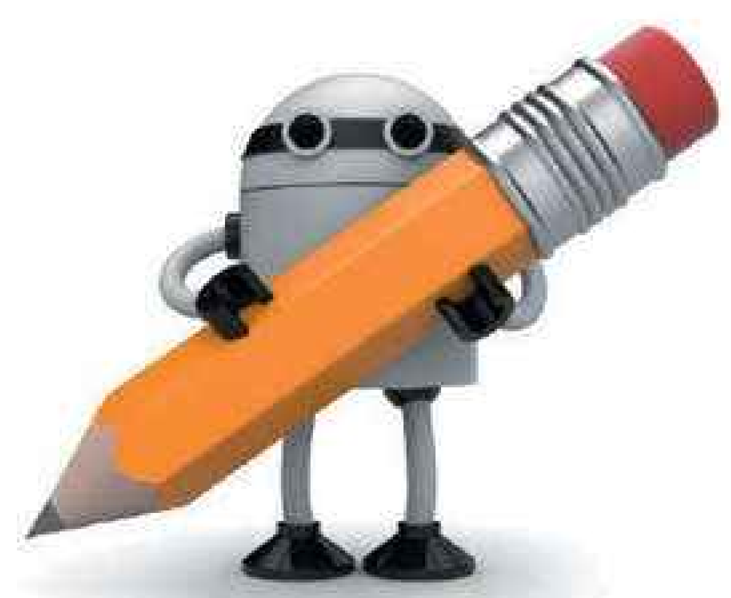
The Pragmatic Programmers

Практическое использование Vim

Редактируйте текст со скоростью мысли



Дрю Нейл

Дрю Нейл

Практическое использование Vim

Drew Neil

Practical Vim

Second Edition

Edit Text ap the Speed of Thought

Foreword by Tim Pope

The Pragmatic Bookshelf
Dalls, Texas • Raleigh, North Carolina

Дрю Нейл

Практическое использование Vim

Второе издание

Редактируйте текст со скоростью мысли

Предисловие Тима Поупа

Москва, 2017

УДК 004.912Vim
ББК 32.973.26-018.2
Н38

Нейл Д.

Н38 Практическое использование Vim / пер. с англ. Киселева А. Н. – 2-е изд. – М.: ДМК Пресс, 2017. – 392 с.: ил.

ISBN 978-5-97060-420-5

Vim – быстрый и эффективный текстовый редактор, способный повысить скорость и эффективность разработки. С помощью более чем 100 рецептов вы быстро освоите основные возможности Vim и сможете заняться решением своих самых необычных задач, связанных с созданием и правкой текста.

В данной книге вы найдете новые и эффективные приемы работы с редактором независимо от того, являетесь ли вы начинающим или опытным пользователем Vim. В обновленном издании исправлены ошибки и добавлены новые рецепты, использующие улучшенные возможности, появившиеся в версии Vim 7.4.110.

Книга предназначена для всех пользователей Vim – как начинающих, так и опытных.

УДК 004.912Vim
ББК 32.973.26-018.2

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-68050-127-8 (анг.)
Copyright © 2015 The Pragmatic Programmers, LLC.

ISBN 978-5-97060-420-5 (рус.)
© Оформление, перевод, ДМК Пресс, 2017

Что говорят читатели о книге «Практическое использование Vim»

Из этой книги я узнал о Vim намного больше, чем из других ресурсов.

➢ **Роберт Эванс (Robert Evans)**,
инженер-программист, Code Wranglers

Прочитав лишь пару глав из книги «Практическое использование Vim», я понял, насколько мало я знаю. За тридцать минут я превратился в начинающего пользователя!

➢ **Хенрик Них (Henrik Nyh)**,
инженер-программист

Книга «Практическое использование Vim» перевернула мои представления о том, что должен уметь текстовый редактор.

➢ **Джон П. Дэйгл (John P. Daigle)**,
разработчик, ThoughtWorks, Inc.

В этой книге Дрю продолжил свою работу, начатую в серии обучающих видеороликов Vimcast. Книгу обязательно следует прочитать всем, кто пользуется редактором Vim.

➢ **Андерс Джанмайр (Anders Janmyr)**,
разработчик, Jayway

Книга «Практическое использование Vim» проложила мост между официальной документацией и тем, как действительно следует использовать Vim. После прочтения нескольких глав Vim стал моим основным редактором. Я уже не смогу отказаться от него.

➢ **Хавьер Колладо (Javier Collado)**,
инженер по автоматизации контроля качества, Canonical Ltd.

Дрю Нейл (Drew Neil) не просто показывает правильный выбор инструмента для работы – он подробно знакомит читателя с философией, стоящей за каждым решением. В своей книге автор не ждет, что вы запомните все и вся, – вместо этого он методично учит вас думать на Vim.

- **Мислав Мароник (Mislav Marohnic)**,
  консультант

Я пользовался редактором Vim при администрировании сервера более пятнадцати лет, но лишь недавно я стал применять его для разработки программного обеспечения. Я думал, что я неплохо знаю Vim, однако книга «Практическое использование Vim» помогла мне значительно повысить свою продуктивность.

- **Грейм Мэтисон (Graeme Mathieson)**,
  инженер-программист, Rubaidh Ltd.

Книга «Практическое использование Vim» показала мне, как много я еще не знаю о Vim. Каждый рецепт, что приводится в книге, можно сразу же применять на практике и немедленно получить прирост производительности труда.

- **Матиас Мейер (Mathias Meyer)**,
  писатель, Riak Handbook

Книга «Практическое использование Vim» – настоящая сокровищница знаний о редакторе Vim. Я ежедневно пользуюсь Vim вот уже два года, но эта книга стала для меня настоящим открытием.

- **Феликс Гайзендорфер (Felix Geisendörfer)**,
  соучредитель, Transloadit

Содержание

Благодарности 20

Предисловие к первому изданию 23

Это надо знать 25

Прочитайте всеми забытое руководство 28

Глава 1. Путь Vim 39

Рецепт 1. Встречайте: команда «точка» 39
  Команда «точка» – микромакроопределение 42
Рецепт 2. Не повторяйся 42
  Избавляйтесь от лишних перемещений 43
Рецепт 3. Шаг назад, три вперед 45
  Делайте изменения повторяемыми 45
  Делайте перемещения повторяемыми 46
  Теперь все вместе 46
Рецепт 4. Действие, повтор, возврат 46
Рецепт 5. Поиск и замена вручную 48
  Будьте экономны: выполняйте поиск без ввода лишних символов ... 49
  Делайте изменения повторяемыми 50
  Теперь все вместе 50
Рецепт 6. Формула точки 50
  Обзор решений трех задач редактирования с помощью команды «точка» ... 51
  Идеальное решение: одна клавиша для перехода и одна для изменения ... 51

Часть I. РЕЖИМЫ 52

Глава 2. Командный режим 53

Рецепт 7. Оторвите кисть от холста 53

Рецепт 8. Группируйте изменения для возможной отмены 54
Рецепт 9. Составляйте повторяемые изменения 56
Удаление назад 56
Удаление вперед 56
Удаление целого слова 57
Дополнительная оценка: какой вариант более повторим? 57
Обсуждение 58
Рецепт 10. Используйте счетчики для простых арифметических операций 58
Рецепт 11. Не занимайтесь подсчетами, если можно выполнить повторение 60
Используйте счетчик, когда в этом есть смысл 62
Рецепт 12. Объединяй и властвуй 63
Оператор + команда перемещения = Действие 63
Расширение возможностей Vim 64

Глава 3. Режим вставки 67

Рецепт 13. Исправляйте ошибки, не выходя из режима вставки 67
Рецепт 14. Возвращайтесь в командный режим 68
Встречайте: командный подрежим режима вставки 69
Рецепт 15. Вставка из регистра, не покидая режима вставки 70
Используйте <C-r>{register} для доступа к регистрам 71
Рецепт 16. Выполняйте простые вычисления на месте 72
Рецепт 17. Вставка необычных символов по их кодам 73
Рецепт 18. Вставка необычных символов, соответствующих парам символов 74
Рецепт 19. Правка текста в режиме замены 74
Затирайте символы табуляции в виртуальном режиме замены 75

Глава 4. Визуальный режим 77

Рецепт 20. Знакомство с визуальным режимом 77
Рецепт 21. Выделение текста в визуальном режиме 79
Включение визуальных режимов 80
Переключение между визуальными режимами 80
Переключение свободного конца выделения 81
Рецепт 22. Повторение команд построчного визуального режима 81
Подготовка 82
Выполните отступ один раз, а затем повторите 82

Рецепт 23. Используйте операторы вместо команд визуального режима, если это возможно 84
Использование визуального оператора 84
Использование операторов командного режима 85
Обсуждение 85
Рецепт 24. Правка табличных данных в блочном визуальном режиме 86
Рецепт 25. Изменение колонок текста 88
Рецепт 26. Добавление текста после непрямоугольного блока 90

Глава 5. Режим командной строки 92

Рецепт 27. Встречайте: режим командной строки 92
Специальные ключи в режиме командной строки 94
Команды Ex стреляют дальше и накрывают большую площадь 95
Рецепт 28. Выполнение команд для одной строки или для группы смежных строк 96
Номера строк 96
Определяйте диапазон строк с использованием их номеров 97
Определяйте диапазон строк посредством визуального выделения 98
Определяйте диапазон строк с помощью шаблонов 99
Изменяйте адрес с помощью смещения 100
Обсуждение 100
Рецепт 29. Копируйте и перемещайте строки с помощью команд ':t' и ':m' 101
Копируйте строки командой :t 101
Перемещайте строки командой ':m' 103
Рецепт 30. Применение команд командного режима к диапазону строк 104
Рецепт 31. Повторяйте последнюю команду Ex 106
Рецепт 32. Автодополнение команд Ex 108
Выбор из нескольких совпадений 109
Рецепт 33. Вставка текущего слова в командную строку 110
Рецепт 34. Повторный вызов команд из истории 111
Встречайте: окно режима командной строки 112
Рецепт 35. Выполнение команд в оболочке 114
Запуск программ в командной оболочке 115
Передача содержимого буфера на вход командам и сохранение вывода команд в буфере 117
Фильтрация содержимого буфера с помощью внешней команды 118

Обсуждение 119
Рецепт 36. Выполнение сразу нескольких команд Ex 119
Выполнение команд Ex по одной 120
Запись команд Ex в сценарий и его выполнение 121
Запуск сценария для изменения нескольких файлов 122

Часть II. ФАЙЛЫ 124

Глава 6. Управление несколькими файлами 125

Рецепт 37. Слежение за открытыми файлами с помощью списка буферов 125
Различия между файлами и буферами 125
Встречайте: список буферов 126
Пользуйтесь списком буферов 127
Удаление буферов 128
Рецепт 38. Группировка буферов в коллекции с помощью списка аргументов 129
Заполнение списка аргументов 130
Определение файлов по именам 130
Определение шаблонов имен файлов 130
Определение файлов с помощью обратных кавычек 131
Использование списка аргументов 132
Рецепт 39. Управление скрытыми файлами 132
Обработка скрытых буферов при выходе из редактора 134
Включите параметр настройки hidden перед вызовом команды :argdo или :bufdo 134
Рецепт 40. Деление рабочего пространства на окна 136
Создание окон 136
Переключение фокуса ввода между окнами 137
Закрытие окон 138
Изменение размеров и переупорядочение окон 138
Рецепт 41. Организация размещения окон с помощью вкладок 139
Как пользоваться вкладками 140
Открытие и закрытие вкладок 141
Переключение между вкладками 141
Переупорядочение вкладок 142

Глава 7. Открытие файлов и их сохранение на диск 143

Рецепт 42. Открытие файла по его имени с использованием команды :edit 143

Как открыть файл, указав путь относительно текущего рабочего каталога ... 144
Как открыть файл, указав путь относительно активного каталога ... 145
Рецепт 43. Открытие файла по его имени с применением команды :find ... 146
Подготовка ... 146
Настройка параметра path ... 147
Используйте команду :find для поиска файлов по именам ... 148
Рецепт 44. Исследование файловой системы с помощью netrw ... 148
Подготовка ... 149
Встречайте: netrw – встроенный обозреватель файлов Vim ... 149
Открытие обозревателя файлов ... 150
Работа с окнами ... 151
Дополнительные возможности netrw ... 152
Рецепт 45. Сохранение файлов в несуществующие каталоги ... 153
Рецепт 46. Сохранение файла с привилегиями суперпользователя ... 153

Часть III. БЫСТРАЯ НАВИГАЦИЯ ... 156

Глава 8. Навигация внутри файлов ... 157

Рецепт 47. Не убирайте руки из основной позиции ... 158
Оставьте свою правую руку там, где она должна быть ... 159
Рецепт 48. Разница между фактическими строками и строками на экране ... 160
Рецепт 49. Перемещение по словам ... 162
Отличайте слова и СЛОВА ... 164
Рецепт 50. Поиск символов ... 165
Поиск символа может выполняться включительно или исключительно ... 168
Думайте как при игре в «Балду» ... 169
Рецепт 51. Поиск с целью навигации ... 169
Используйте команды поиска в операциях ... 171
Рецепт 52. Выделение фрагментов с применением текстовых объектов ... 172
Выполнение операций с текстовыми объектами ... 175
Обсуждение ... 175
Рецепт 53. Удаление, включая ограничители, и изменение внутри ограничителей ... 176
Рецепт 54. Установка меток и возврат к ним ... 178

Автоматическая расстановка меток ... 179
Рецепт 55. Переход между парными скобками ... 180
Переход между парными ключевыми словами ... 181

Глава 9. Навигация между файлами ... 183

Рецепт 56. Обход списка переходов ... 183
Рецепт 57. Обход списка изменений ... 185
Отметка последнего изменения ... 186
Рецепт 58. Переход к файлу с именем под курсором ... 187
Определение расширения файла ... 188
Определение списка каталогов для поиска ... 188
Обсуждение ... 189
Рецепт 59. Переключение между файлами с помощью глобальных меток ... 190
Устанавливайте глобальную метку перед погружением в код ... 191

Часть IV. РЕГИСТРЫ ... 192

Глава 10. Копирование и вставка ... 193

Рецепт 60. Удаление, копирование и вставка с применением неименованного регистра ... 193
Перестановка символов ... 194
Перестановка строк ... 194
Создание дубликатов строк ... 195
Ой! Я затер свою копию ... 195
Рецепт 61. Знакомство с регистрами Vim ... 196
Адресация регистров ... 197
Неименованный регистр ("") ... 198
Регистр захвата («0) ... 199
Именованные регистры ("a–"z) ... 199
Регистр «черной дыры» ("_) ... 200
Системный буфер обмена ("+) и регистр выделенного фрагмента ("*) ... 201
Регистр выражений ("=) ... 202
Дополнительные регистры ... 202
Рецепт 62. Замена выделенного текста содержимым регистра ... 203
Как поменять слова местами ... 204
Рецепт 63. Вставка из регистра ... 205
Вставка последовательностей символов ... 205

Вставка строк 206
Обсуждение 207
Рецепт 64. Взаимодействие с системным буфером обмена 207
Подготовка 208
Вызов системной команды вставки 208
Использование системной команды вставки в режиме вставки 209
Используйте регистр "+, чтобы исключить необходимость переключения параметра paste 210

Глава 11. Макросы 211

Рецепт 65. Запись и выполнение макроса 212
Запись последовательности команд в виде макроса 212
Повторное выполнение последовательности команд вызовом макроса 213
Последовательное выполнение макроса 214
Параллельное выполнение макроса 215
Рецепт 66. Товсь! Цельсь! Отставить! 215
Нормализация позиции курсора 215
Устанавливайте курсор повторяемыми командами перемещения 216
Используйте возможность прерывания при неудачном перемещении курсора 216
Рецепт 67. Выполнение со счетчиком 217
Рецепт 68. Повторение изменений в последовательности строк 219
Запись единицы правки 219
Последовательное выполнение макроса 221
Параллельное выполнение макроса 222
Выбор: последовательно или параллельно 223
Рецепт 69. Добавление команд в макросы 223
Обсуждение 224
Рецепт 70. Выполнение операций над множеством файлов 225
Подготовка 225
Создание списка целевых файлов 226
Запись единицы правки 226
Параллельное выполнение макроса 227
Последовательное выполнение макроса 228
Сохранение изменений во всех файлах 229
Обсуждение 229

Рецепт 71. Использование итератора для нумерации элементов в списке 230
Простой язык сценариев Vim 231
Запись макроса 231
Выполнение макроса 232
Рецепт 72. Правка содержимого макроса 232
Задача: Нестандартное форматирование 233
Вставка макроса в документ 234
Правка макроса 234
Копирование макроса из документа обратно в регистр 234
Обсуждение 235

Часть V. ШАБЛОНЫ 237

Глава 12. Поиск по шаблону и поиск точного совпадения 238

Рецепт 73. Настройка чувствительности к регистру в шаблонах 238
Глобальная настройка чувствительности к регистру 239
Настройка чувствительности к регистру для каждой операции поиска 239
Интеллектуальное определение чувствительности к регистру 239
Рецепт 74. Включение поддержки регулярных выражений 240
Поиск шестнадцатеричных кодов цвета в расширенном режиме 241
Поиск шестнадцатеричных кодов цвета в самом волшебном режиме 242
Использование класса шестнадцатеричных цифр 242
Обсуждение 242
Рецепт 75. Ключ \V включает режим поиска точного совпадения 244
Рецепт 76. Использование круглых скобок для захвата совпадений с подвыражениями 246
Рецепт 77. Границы слова 247
Рецепт 78. Границы совпадения 249
Рецепт 79. Экранирование проблемных символов 251
Экранируйте символы «/», выполняя поиск вперед 252
Экранируйте символы «?», выполняя поиск назад 253
Всегда экранируйте символы «\» 254
Экранируйте символы программно 255

Глава 13. Поиск 257

Рецепт 80. Встречайте: команда поиска 257
Выполнение команды поиска.............................. 258
Определение направления поиска 258
Повторение последней команды поиска.................... 258
История поиска 260
Рецепт 81. Подсветка совпадений.............................. 260
Отключение подсветки совпадений 260
Рецепт 82. Предварительный просмотр первого совпадения 261
Проверка существования совпадения 262
Автодополнение поля ввода шаблона с опорой
на предварительные результаты поиска 262
Рецепт 83. Смещение курсора относительно конца
совпадения 263
Рецепт 84. Выполнение операций над полным совпадением...... 265
Усовершенствованная формула точки 267
Рецепт 85. Создание сложных шаблонов с использованием
истории поиска 268
1: Максимальное совпадение 269
2: Доработка... 269
3: Еще один цикл 270
4: Последний штрих....................................... 271
Обсуждение 272
Рецепт 86. Подсчет совпадений с текущим шаблоном............... 272
Подсчет количества совпадений командой ':substitute'.......... 273
Подсчет количества совпадений командой ':vimgrep'............ 273
Рецепт 87. Поиск текущего визуального выделения 274
Поиск текущего слова в визуальном режиме 274
Поиск текущего выделения (прототип) 275
Поиск текущего выделения (окончательная версия) 276

Глава 14. Подстановка.................................. 277

Рецепт 88. Встречайте: команда подстановки 277
Настройка поведения команды подстановки с помощью
флагов.. 278
Специальные символы в строке замены 279
Рецепт 89. Поиск и замена всех совпадений в файле 279
Рецепт 90. Подтверждение каждой подстановки 281
Обсуждение 282
Рецепт 91. Повторное использование последнего шаблона
поиска.. 283

Этот прием подходит не всегда 284
Влияние на историю команд 284
Рецепт 92. Замена содержимым регистра 285
Передача по значению 285
Передача по ссылке 286
Сравнение 286
Рецепт 93. Повторение предыдущей команды подстановки 287
Повторение команды подстановки в строке ко всему файлу 288
Изменение диапазона в команде подстановки 288
Обсуждение 290
Рецепт 94. Переупорядочение полей в файле CSV 291
Рецепт 95. Выполнение арифметических операций в строке замены 292
Шаблон поиска 293
Команда подстановки 293
Рецепт 96. Перемена местами двух и более слов 294
Возврат другого слова 295
Поиск совпадений с двумя словами 295
Все вместе 295
Обсуждение 296
Рецепт 97. Поиск и замена в нескольких файлах 297
Команда подстановки 297
Поиск во всех файлах в текущем проекте с использованием ':vimgrep' 298
Применение команды подстановки ко всем файлам в текущем проекте с использованием ':cfdo' 298
В заключение 300

Глава 15. Глобализация команд 301

Рецепт 98. Встречайте: команда :global 301
Рецепт 99. Удаление строк, соответствующих шаблону 302
Удаление соответствующих строк командой :g/re/d 303
Сохранение только соответствующих строк командой :v/re/d 304
Рецепт 100. Выборка комментариев TODO в регистр 304
Обсуждение 306
Рецепт 101. Сортировка свойств в правилах CSS 307
Сортировка свойств внутри одного блока 307
Сортировка свойств во всех блоках 308
Обсуждение 310

Часть VI. ИНСТРУМЕНТЫ 311

Глава 16. Индексирование исходного кода и навигация по нему с помощью ctags 312

Рецепт 102. Встречайте: ctags 312
Установка exuberant ctags 312
Индексирование исходного кода проекта с помощью ctags 313
Анатомия индексного файла 313
Ключевые слова адресуются шаблонами, а не номерами строк 314
Индексирование ключевых слов с помощью метаданных 315
Рецепт 103. Настройка Vim для работы с программой ctags 316
Настройка поиска индексного файла в Vim 316
Создание индексного файла 316
Обсуждение 318
Рецепт 104. Навигация по определениям ключевых слов 318
Переход к определению ключевого слова 318
Определение точки перехода при наличии нескольких совпадений с ключевым словом 320
Использование команд Ex 321

Глава 17. Компиляция кода и обзор ошибок с помощью Quickfix List 323

Рецепт 105. Компиляция кода, не покидая Vim 324
Подготовка 324
Компиляция проекта в командной оболочке 324
Компиляция проекта в Vim 325
Рецепт 106. Навигация по списку с результатами 327
Основы навигации по списку с результатами 328
Быстрые переходы вперед и назад 329
Окно Quickfix 329
Рецепт 107. Восстановление прежнего списка с результатами 330
Рецепт 108. Настройка внешнего компилятора 330
Настройка вызова программы Nodelint командой :make 331
Заполнение списка с результатами на основе вывода Nodelint 332
Настройка makeprg и errorformat единственной командой 333

Глава 18. Поиск в пределах проекта с помощью команд grep, vimgrep и других 335

Рецепт 109. Вызов grep, не покидая Vim 335

Использование grep из командной строки 336
Вызов grep из редактора Vim 336
Рецепт 110. Настройка программы grep 337
Настройки по умолчанию команды :grep 337
Настройка :grep на вызов команды ack 338
Переход к строке и позиции в строке при использовании ack 340
Рецепт 111. Поиск с использованием внутреннего механизма поиска Vim 341
Выбор файлов для поиска 342
Поиск в файле с последующим поиском в проекте 343
История поиска и :vimgrep 343

Глава 19. Набери X и пользуйся автодополнением 345

Рецепт 112. Встречайте: автодополнение ключевых слов........... 345
Вызов функции автодополнения.. 347
Рецепт 113. Работа с меню функции автодополнения................ 347
Обзор списка слов без изменения документа.......................... 348
Изменение документа по мере прокручивания списка слов 349
Отклонение вариантов выбора 349
Фильтрация списка по мере ввода 349
Рецепт 114. Источники ключевых слов 350
Буфер... 350
Подключаемые файлы 351
Индексные файлы 351
Объединяем все вместе 352
Рецепт 115. Автодополнение словами из словаря 353
Рецепт 116. Автодополнение целых строк 354
Рецепт 117. Автодополнение последовательностей слов 355
Рецепт 118. Автодополнение имен файлов............................... 357
Рецепт 119. Контекстное автодополнение................................ 359

Глава 20. Поиск и исправление опечаток в Vim 361

Рецепт 120. Проверьте свой текст 361
Принцип действия механизма проверки орфографии в Vim 362
Рецепт 121. Использование альтернативных орфографических словарей 363
Настройка диалекта языка... 363
Получение словарей для других языков................................. 364
Рецепт 122. Добавление слов в орфографический словарь........ 364

Создание словаря для специальных терминов 365
Рецепт 123. Исправление орфографических ошибок
в режиме вставки 366
Подготовка 366
Обычный способ: выход в командный режим 366
Быстрый способ: использовать орфографическое
автодополнение 366

Глава 21. Что дальше? 368

Продолжайте практиковаться! 368
Настраивайте Vim под себя 368
Узнайте, как пользоваться пилой, и только потом точите ее 369

**Приложение 1. Настройка Vim в соответствии
с личными предпочтениями 371**

Изменяйте настройки на лету 371
Сохраняйте настройки в файле vimrc 373
Применение настроек для определенных типов файлов 375

Предметный указатель 377

Благодарности

Спасибо Браму Моленару (Bram Moolenaar) за создание редактора Vim и всем, кто участвовал в его разработке. Это – образец программного обеспечения на все времена, и я с нетерпением жду появления новых его версий.

Спасибо всем сотрудникам издательства Pragmatic Bookshelf, работавшим со мной над этой книгой. Особое спасибо Кею Кепплеру (Kay Keppler), моему научному редактору, за то, что помог мне стать писателем и написать эту книгу, невзирая на сложности и мои вспышки раздражения. Спасибо также Катарине Дворак (Katharine Dvorak), моему литературному редактору, за работу над этим изданием. Я также хочу поблагодарить Дэвида Келли (David Kelly) за удовлетворение всех моих необычных пожеланий по форматированию текста.

Изначально книга «Практическое использование Vim» задумывалась совсем не как сборник рецептов, но Сюзанна Фалзер (Susannah Pfalzer) сказала, что будет лучше придерживаться именно этого формата. Было довольно сложно переписать такой объем информации, но мне удалось сделать это с первого раза, чем я был очень доволен. Сюзанна знает, как будет лучше, и я благодарен ей за ее видение.

Спасибо Дэйву Томасу (Dave Thomas) и Энди Ханту (Andy Hunt) за создание издательства Pragmatic Bookshelf. Я рад, что представлен именно этим издательством, и для меня большая честь, что моя книга находится в их каталоге.

Книга «Практическое использование Vim» едва ли появилась бы на свет без моих технических обозревателей. Каждый из вас внес свой вклад и помог в создании книги. Спасибо вам, Адам МакКри (Adam McCrea), Алан Гарднер (Alan Gardner), Алекс Кан (Alex Kahn), Али Олвейсити (Ali Alwasity), Андерс Джанмайр (Anders Janmyr), Эндрю Дональдсон (Andrew Donaldson), Ангус Нейл (Angus Neil), Чарли Танксли (Charlie Tanksley), Чез Мартин (Ches

Martin), Дэниел Бретой (Daniel Bretoi), Дэвид Моррис (David Morris), Денис Горин (Denis Gorin), Элиза Мендес Ризенд (Ely zer Mendes Rezende), Эрик Сент Мартин (Erik St. Martin), Федерико Галасси (Federico Galassi), Феликс Гейзендорфер (Felix Geisendörfer), Флориан Вален (Florian Vallen), Грэм Матисон (Graeme Mathieson), Ханс Хассельберг (Hans Hasselberg), Хенрик Них (Henrik Nyh), Хавьер Колладо (Javier Collado), Джефф Холланд (Jeff Holland), Джош Салливан (Josh Sullivan), Джошуа Флэнаган (Joshua Flanagan), Кана Натсуно (Kana Natsuno), Кент Фрайзер (Kent Frazier), Луис Мерино (Luis Merino), Матиас Мейер (Mathias Meyer), Мэтт Соузерден (Matt Southerden), Мислав Мароник (Mislav Marohnic), Митч Гатри (Mitch Guthrie), Морган Прайор (Morgan Prior), Пол Бэрри (Paul Barry), Питер Аронофф (Peter Aronoff), Питер Рин (Peter Rihn), Филип Робертс (Philip Roberts), Роберт Эванс (Robert Evans), Райан Стенхауз (Ryan Stenhouse), Чтивен Рагнарок (Steven Ragnarök), Тибор Симик (Tibor Simic), Тим Чейз (Tim Chase), Тип Поуп (Tim Pope), Тим Тирелл (Tim Tyrrell) и Тобиас Сайлер (Tobias Sailer).

Несмотря на всемерную помощью моих научных редакторов, в тексте могли остаться незамеченными некоторые ошибки. Я буду благодарен всем, кто сообщит мне об ошибках в книге, поможет их найти и исправить.

Документация к редактору Vim – потрясающий источник знаний, и я часто буду ссылаться на нее на протяжении всей книги. Я хотел бы поблагодарить Карло Тюбнера (Carlo Teubner) за публикацию документации к Vim на сайте vimhelp.appspot.com и за постоянное ее обновление.

Некоторые советы в первом издании оказались неудачными, но я все равно оставил их, потому что считал важными. В этом, пересмотренном издании я смог переписать их. Спасибо Кристиану Брабандту (Christian Brabandt) за реализацию качественно новой команды `gn`, благодаря которой я смог поправить совет 84 «Выполнение операций над полным совпадением». Спасибо Йегаппану Лакшманану (Yegappan Lakshmanan) за реализацию команды `cfdo` (и родственных ей), благодаря которой я смог поправить совет 97 «Поиск и замена в нескольких файлах». Я также хочу выразить благодарность Дэвиду Баргину (David Bürgin) за редакцию 7.3.850, что исправило мою «любимую» ошибку с командой `vimgrep`.

Я также хотел бы выразить благодарность всему сообществу пользователей Vim за распространение их знаний по всему Интернету. Многие рецепты, что приводятся в этой книге, были найдены

по тегу «Vim» на сайте StackOverflow и в списке рассылки vim_use.

Расширение rails.vim Тима Поупа (Tim Pope) оказало существенное влияние на серьезность моего отношения к Vim. Многие другие его расширения также стали обязательной частью Vim для меня. Немалую помощь в понимании Vim мне оказали также расширения, написанные Каном Натсуно (Kana Natsuno), текстовые объекты которых я считаю лучшим расширением базовой функциональности. Спасибо вам обоим, что помогли мне увидеть дополнительные преимущества.

Спасибо Джо Рознеру (Joe Rozner) за предоставленный исходный код, который я использовал для демонстрации команды `:make`. Спасибо Олегу Ефимову (Oleg Efimov) за его мгновенные ответы по проблемам nodelint. Спасибо Бену Кормаку (Ben Cormack) за иллюстрации.

В январе 2012-го мы приехали в Берлин, где сообщество технических специалистов вдохновило меня на создание данной книги. Я благодарен Грегору Шмидту (Gregor Schmidt) за основание Берлинской группы пользователей Vim и Яну Шульцу-Хофену (Jan Schulz-Hofen) за предоставленное место для встречи. Общение с пользователями Vim по-настоящему помогло мне привести свои мысли в порядок, поэтому я благодарен всем, кто посещал наши встречи в Берлине. Спасибо вам, Дэниел и Нина Холле (Daniel и Nina Holle), за то, что предоставили нам свой дом. Это замечательное место, где можно жить и активно работать.

В марте 2011-го мне потребовалась хирургическая операция на кишечнике. К сожалению, в этот момент я оказался далеко от дома, но, к счастью, моя жена была со мной. Ханна (Hannah) признала, что в южно-синайской больнице мне был обеспечен превосходный уход. Я хочу поблагодарить всех сотрудников больницы за помощь и доктора Шавкета Гергеса (Shawket Gerges) за удачно проведенную операцию.

Когда моя мама узнала, что мне нужна операция, она бросила все и ближайшим рейсом вылетела в Египет, несмотря на начавшуюся там революцию. Это был смелый поступок! Мне трудно представить, как мы с Ханной справились бы с трудностями без поддержки моей мамы и ее жизненного опыта. Я безмерно счастлив, что в моей жизни есть две такие замечательные женщины.

Предисловие к первому изданию

Существует расхожее мнение, что редактор Vim сложен в освоении. Я думаю, большинство пользователей Vim не согласятся с этим. Конечно, на первом этапе будут возникать некоторые затруднения, но как только вы пройдете обучение с помощью `vimtutor` и познакомитесь с устройством конфигурационного файла *vimrc*, вы получите почти все необходимые знания, обладая которыми, уже можно хоть как-то работать.

А что дальше? Интернет отвечает на этот вопрос: «читайте советы» – рецепты решения конкретных проблем. Вы можете искать конкретные рецепты, когда чувствуете, что текущее решение не является оптимальным, или наоборот – активно читать сборники популярных рецептов. Такой подход дает свои плоды – именно так я изучал работу с Vim, – но он очень медленный. Разумеется, полезно знать, что нажатие `*` запускает поиск слова под курсором, но едва ли это знание поможет вам думать на уровне знатока Vim.

Когда книга «Практическое использование Vim» попала в мои руки, я испытывал некоторый скепсис в отношении ее. Как пара сотен рецептов может помочь достичь того же уровня, для чего мне понадобилось изучить несколько тысяч рецептов? Однако, прочитав несколько страниц, я понял, что мое понятие «рецепт» было слишком узким. Вопреки моим ожиданиям рецепты в книге «Практическое использование Vim» не следуют шаблону проблема/решение, а представляют собой уроки, учащие думать категориями опытного пользователя Vim. В некотором смысле они больше похожи на притчи, чем на рецепты. Первые несколько рецептов учат широко использовать команду `.` – основной инструмент любого опытного пользователя Vim, для обнаружения которой без посторонней помощи мне потребовались годы.

Именно этим меня порадовала книга «Практическое использование Vim». Теперь, когда начинающие пользователи спрашивают меня: «Что дальше?», – я знаю, что им ответить. В конце концов, книга «Практическое использование Vim» даже мне преподнесла несколько приятных сюрпризов.

Тим Поуп (Tim Pope),
член основной команды разработчиков Vim,
апрель 2012 года

Это надо знать

Книга «Практическое использование Vim» предназначена для программистов, желающих повысить свой уровень. Возможно, вам доводилось слышать, что в руках мастера Vim кромсает текст со скоростью мысли. Прочитав данную книгу, вы сделаете еще один шаг к овладению этим мастерством.

Книга «Практическое использование Vim» – это самый короткий путь к овладению Vim. Она не требует наличия опыта использования Vim, но для начинающих пользователей совсем нелишним будет получить предварительные знания, воспользовавшись интерактивной обучающей программой `vimtutor`, распространяемой вместе с Vim[1]. «Практическое использование Vim» основана на этих базовых сведениях и демонстрирует приемы идиоматического использования.

Редактор Vim имеет огромное количество настроек. Однако настройки сильно зависят от личных предпочтений, поэтому я старался избегать каких-либо рекомендаций о том, что должно быть или чего не должно быть в файле *vimrc*. Вместо этого я расскажу об основных функциональных возможностях редактора, поддерживаемых им всегда, независимо от того, работаете ли вы с Vim удаленно, через SSH, или используете локальную версию GVim с массой дополнительных расширений. Овладев основами Vim, вы получите мощный и переносимый инструмент редактирования текста.

Организация книги

Книга «Практическое использование Vim» – это сборник рецептов. Ее не обязательно читать по порядку, от начала до конца. (Я изначально предполагал такую возможность! В начале следующей главы я сразу посоветую вам пропустить ее и немедленно перейти к действию.) Каждая глава – это сборник рецептов, объединенных

---

[1] http://vimdoc.sourceforge.net/htmldoc/usr_01.html#vimtutor

общей темой, а каждый рецепт демонстрирует применение какой-то одной особенности. Некоторые рецепты не связаны с другими. Иные требуют знания сведений, которые приводятся где-то в другом месте в этой книге. Такие рецепты включают ссылки на необходимые сведения, поэтому вы легко сможете найти их.

В целом книга «Практическое использование Vim» не следует шаблону «от простого к сложному», но каждая глава использует именно такой порядок изложения. Малоопытные читатели могут сначала просто пробежаться по книге и прочитать лишь первые рецепты в каждой главе. Более опытные пользователи могут сосредоточиться на более сложных рецептах или выбирать разделы для чтения по своему усмотрению.

Примечания к примерам

Любую задачу в Vim можно решить несколькими способами. Например, в главе 1 «Путь Vim» собраны задачи, на которых иллюстрируется применение команды «точка», но каждую из этих задач можно также решить с помощью команды `:substitute`.

Время от времени при знакомстве с моими решениями у вас может возникнуть мысль: «Разве *такое-то и такое-то* решение не будет проще?» И вы будете правы! Мои решения иллюстрируют конкретные приемы. Не заостряйте внимания на их простоте, а попробуйте отыскать сходство с задачами, которые вам приходится решать ежедневно. Именно в этом случае описываемые приемы помогут вам сэкономить свое время.

От переводчика: *в процессе работы над книгой я опробовал приводимые здесь примеры в версиях Vim/gVim для Linux и MS Windows и обнаружил одно неприятное обстоятельство: в версиях для ОС Windows некоторые комбинации клавиш действуют не так, как описывается в этой книге (а иногда вообще не действуют). Поэтому будьте внимательны, если предполагаете пользоваться редактором Vim/gVim в Windows.*

Научитесь печатать вслепую, а затем изучайте Vim

Если вы постоянно будете смотреть на клавиатуру, вы нескоро начнете получать выгоды от знания Vim. Владение слепой печатью является обязательным условием.

Свою родословную Vim ведет от классических текстовых редакторов ОС Unix, `vi` и `ed` (см. врезку «О происхождении Vim (и семейства)» в главе 5). Они были созданы задолго до появления мыши и интерфейсов типа «укажи и щелкни». В редакторе Vim все операции можно выполнить с клавиатуры. Для владеющих слепой печатью это означает, что Vim *ускоряет* работу.

Прочитайте всеми забытое руководство

В книге «Практическое использование Vim» я буду показывать примеры, а не описывать их. Это непростая задача, если использовать только письменное слово. Чтобы показать шаги, выполняемые в ходе сеанса редактирования, я буду использовать простую нотацию, иллюстрирующую нажатия клавиш и содержимое буфера Vim.

Если вы горите желанием немедленно приступить к действию, просто пропустите этот раздел. Он описывает соглашения, используемые в книге «Практическое использование Vim», многие из которых и без того понятны. Возможно, в какой-то момент вы столкнетесь с непонятным вам символом, тогда вы сможете вернуться к этому разделу и поискать нужное вам описание.

Ознакомьтесь со встроенной документацией Vim

Самый простой способ ознакомиться с документацией Vim – потратить некоторое время на ее чтение. Чтобы помочь вам в этом, я включил в текст книги «гиперссылки» на разделы документации. Например, вот «гиперссылка» на учебник Vim: `:h vimtutor` ⓘ http://vimdoc.sourceforge.net/htmldoc/usr_01.html#vimtutor.

Ярлык несет двойную функцию. Во-первых, он служит визуальным признаком полезной справочной информации. Во-вторых, если вы читаете электронную версию книги на устройстве, подключенном к Интернету, вы можете щелкнуть на ярлыке и открыть соответствующий раздел электронной документации Vim. В этом смысле ярлык является настоящей гиперссылкой.

Как быть, если вы читаете печатное издание книги? Не волнуйтесь. Если у вас имеется редактор Vim под рукой, просто введите команду, что приводится перед ярлыком.

Например, введите `:h vimtutor` (`:h` – это сокращенная версия команды `:help`). Считайте строку `vimtutor` уникальным адресом в документации, своеобразной строкой URL. В этом смысле команду получения справки можно считать гиперссылкой во встроенной документации Vim.

Нотация представления Vim на странице

Модальный интерфейс Vim отличается от интерфейса большинства других текстовых редакторов. Если провести аналогию с музыкальными инструментами, клавиатуру компьютера можно сравнить с клавиатурой фортепиано. Пианист может проигрывать мелодию, нажимая клавиши по одной или целыми группами, извлекая аккорды. В большинстве текстовых редакторов клавиатурные сокращения соответствуют нажатию какой-то клавиши при одной или более удерживаемых клавишах-модификаторах, таких как **Control** и **Command**. Такие клавиатурные сокращения являются эквивалентом аккордов.

Некоторые команды Vim также запускаются проигрыванием аккордов, но в режиме Normal (командный режим, или режим по умолчанию) команды вводятся как последовательные нажатия клавиш. Это компьютерный эквивалент проигрыванию мелодии нажатием клавиш фортепиано по одной.

`Ctrl`-`S` – это типичное соглашение, используемое для обозначения команд, вызываемых аккордами из комбинаций клавиш. В данном случае эта последовательность означает: «Нажать клавиши **Control** и **S** одновременно». Но подобное соглашение плохо подходит для описания команд Vim. В этом разделе мы увидим нотацию, используемую по всей книге «Практическое использование Vim» для иллюстрации приемов работы с редактором.

Проигрывание мелодий

В командном режиме команды вводятся последовательным нажатием одной или более клавиш. Выглядят эти команды следующим образом:

Нотация	Значение
`x`	Нажать одну клавишу `x`
`dw`	Нажать сначала `d`, потом `w`
`dap`	Нажать сначала `d`, потом `a`, потом `p`

Большинство таких последовательностей состоят из двух или трех нажатий на клавиши, но есть и более длинные. Расшифровка значения команд в командном режиме может оказаться непростым делом, но, немного попрактиковавшись, вы легко будете справляться с этим.

Аккорды

Обозначения, такие как <C-p>, не означают: «Нажать <, затем C, затем - и т. д.». Запись <C-p> эквивалентна записи Ctrl-p, то есть: «Нажать клавиши <Ctrl> и p одновременно».

Я выбрал такую форму записи вполне обоснованно. Она используется во встроенной документации Vim (:h key-notation ⓘ http://vimdoc.sourceforge.net/htmldoc/intro.html#key-notation), и мы также будем использовать ее для определения комбинаций клавиш. Некоторые команды Vim запускаются как комбинации аккордов и последовательных нажатий на клавиши, и такая форма записи отлично описывает их, например:

Нотация	Значение
<C-n>	Нажать одновременно <Ctrl> и n
g<C-]>	Нажать g, затем одновременно нажать <Ctrl> и]
<C-r>0	Нажать одновременно <Ctrl> и r, затем нажать 0
<C-w><C-=>	Нажать одновременно <Ctrl> и w, затем нажать одновременно <Ctrl> и =

Символы-заполнители

Многие команды Vim требуют последовательного нажатия двух или более клавиш. Некоторые команды должны сопровождаться вводом некоторой последовательности символов, тогда как другие – вводом единственного символа. Символы, которые должны вводиться после команды, я буду обозначать как последовательности, заключенные в фигурные скобки, например:

Нотация	Значение
f{char}	Нажать f, затем ввести любой символ
`{a-z}	Нажать `, затем ввести любой символ нижнего регистра
m{a-zA-Z}	Нажать m, затем ввести любой символ нижнего или верхнего регистра
d{motion}	Нажать d, затем ввести любую команду перемещения курсора
<C-r>{register}	Нажать одновременно <Ctrl> и r, затем ввести адрес регистра

Обозначение специальных клавиш

Некоторые клавиши имеют специальные обозначения. Некоторые из них перечислены в таблице ниже:

Нотация	Значение
`<Esc>`	Клавиша Escape (Esc)
`<CR>`	Клавиша возврата каретки (также известна как <Enter>)
`<Ctrl>`	Клавиша Control (Ctrl)
`<Tab>`	Клавиша табуляции (Tab)
`<Shift>`	Клавиша Shift
`<S-Tab>`	Одновременное нажатие клавиш <Shift> и <Tab>
`<Up>`	Клавиша со стрелкой «вверх»
`<Down>`	Клавиша со стрелкой «вниз»
`␣`	Пробел

Обратите внимание, что клавиша пробела обозначается как `␣`. Команда `f{char}` с этой клавишей будет записываться так: `f␣`.

Обозначение команд в разных режимах

При работе с редактором Vim довольно часто приходится переключаться между командным режимом и режимом вставки. В разных режимах одна и та же клавиша может вызывать разные действия. Нажатия клавиш в режиме **Insert** (режиме вставки) я буду обозначать иначе, чтобы их проще было отличать от нажатий клавиш в командном режиме.

Взгляните на такой пример: `cwreplacement<Esc>`. В командном режиме команда `cw` удалит текст до конца текущего слова и переключит редактор в режим вставки. Затем мы вводим слово «replacement» в режиме вставки и нажимаем клавишу `<Esc>`, чтобы переключиться обратно в командный режим.

Форма представления нажатий клавиш в командном режиме также будет использоваться для обозначения нажатий клавиш в режиме **Visual** (визуальный режим), а форма представления нажатий клавиш в режиме вставки будет использоваться для обозначения нажатий клавиш в режимах **Command-Line** (командной строки) и **Replace** (замены). Какой режим является текущим, будет очевидно из контекста.

Режим командной строки

В некоторых рецептах мы будем выполнять команды из командной строки – либо команды командной оболочки, либо внутренние

команды Vim. Вот как выглядит запуск команды `grep` в командной оболочке:

```
⇒ $ grep -n Waldo *
```

А так выглядит запуск встроенной в Vim команды `:grep`:

```
⇒ :grep Waldo *
```

В примерах в этой книге символ `$` будет указывать, что команда выполняется внешней командной оболочкой, а символ `:` будет указывать, что команда выполняется во внутреннем режиме командной строки. Иногда мы будем видеть и другие приглашения к вводу, включая следующие:

Приглашение	Значение
$	Введенная команда будет выполнена внешней командной оболочкой
:	Используется режим командной строки редактора Ex
/	Используется режим командной строки для поиска вперед
?	Используется режим командной строки для поиска назад
=	Используется режим командной строки для вычисления выражения на языке сценариев Vim

Всякий раз, когда в тексте будет встречаться команда редактора Ex, такая как `:write`, будет предполагаться, что она завершается нажатием клавиши `<CR>`, иначе команда просто не будет выполнена, то есть нажатие клавиши `<CR>` будет подразумеваться неявно.

Напротив, команда поиска в Vim отыскивает первое совпадение до нажатия клавиши `<CR>` (см. рецепт 82 в главе 13). Когда в тексте будет встречаться команда поиска, такая как `/pattern<CR>`, нажатие клавиши `<CR>` будет указываться явно. Если `<CR>` будет отсутствовать в команде поиска, знайте, что это сделано преднамеренно и означает, что вы не должны нажимать клавишу **Enter**.

Отображение позиции курсора в буфере

При просмотре содержимого буфера иногда полезно иметь возможность увидеть, где находится курсор. В следующем примере показано, что курсор находится в позиции первой буквы в слове «One»:

```
One two three
```

Когда мы в несколько этапов вносим некоторое изменение, содержимое буфера проходит через последовательность промежуточных состояний. Для отображения этого процесса я буду использовать таблицу, в левом столбце которой будут показаны выполняемые команды, а в правом – содержимое буфера. Например:

Нажатия клавиш	Содержимое буфера
{start}	One two three
dw	two three

Во второй строке выполняется команда dw, удаляющая слово под курсором. В этой же строке показано, как изменилось состояние буфера сразу после выполнения команды.

Подсветка совпадений, найденных при поиске

При демонстрации команды поиска полезно иметь возможность выделить все совпадения, найденные в буфере. В следующем примере выполняется поиск строки «the» и обнаруживаются четыре совпадения, выделенные цветом:

Нажатия клавиш	Содержимое буфера
{start}	the problem with these new recruits is that they don't keep their boots clean.
/the<CR>	the problem with these new recruits is that they don't keep their boots clean.

Загляните в рецепт 81 в главе 13, где рассказывается, как включить в Vim подсветку совпадений, найденных при поиске.

Выделение текста в визуальном режиме

В визуальном режиме имеется возможность выделить текст в буфере и затем выполнить с ним какие-либо операции. В следующем примере для выделения содержимого тега <a> используется текстовый объект it:

Нажатия клавиш	Содержимое буфера
`{start}`	`<a href="http://pragprog.com/dnvim/">Practical Vim</a>`
`vit`	`<a href=»http://pragprog.com/dnvim/»>Practical Vim</a>`

Обратите внимание, что выделение в визуальном режиме выглядит так же, как подсветка совпадений в режиме поиска. Когда в книге будет приводиться такое оформление, из контекста должно быть понятно, представляет оно выделение в визуальном режиме или подсветку совпадений в режиме поиска.

Загружаемые примеры

Примеры в книге обычно будут начинаться с демонстрации содержимого файла *до* его изменения. Заголовки листингов будут также включать путь к файлу:

macros/incremental.txt
http://media.pragprog.com/titles/dnvim/code/macros/incremental.txt

```
partridge in a pear tree
turtle doves
French hens
calling birds
golden rings
```

Каждый раз, когда вы увидите такое имя файла со строкой пути к нему, это будет означать, что данный пример можно загрузить. В каждом таком случае я рекомендую открыть файл в Vim и попытаться выполнить упражнение самостоятельно. Это самый лучший способ изучения!

Прежде чем продолжить, загрузите все примеры и исходные коды с сайта издательства Pragmatic Bookshelf[1]. Если вы читаете электронную версию книги на устройстве, подключенном к Интернету, вы можете загрузить любой файл отдельно, просто щелкнув на имени файла. Попробуйте сделать это в примере выше.

Используйте настройки Vim по умолчанию

Редактор Vim может настраиваться в очень широких пределах. Если какие-то настройки по умолчанию не устраивают вас,

[1] http://pragprog.com/titles/dnvim/source_code

измените их. В этом нет ничего плохого, но это может вызвать путаницу, если вы собираетесь следовать за примерами в данной книге, используя версию Vim, настроенную по своему вкусу. Вы можете неожиданно обнаружить, что что-то работает совсем не так, как описывается в книге. Если вы подозреваете, что причиной являются произведенные вами настройки, выполните следующий простой тест. Попробуйте запустить Vim со следующими параметрами:

```
⇒ $ vim -u NONE -N
```

Флаг `-u NONE` сообщает редактору Vim, что он не должен загружать настройки из файла *vimrc*. То есть настройки не будут выполнены, а расширения будут отключены. Когда Vim запускается таким способом, он возвращается к режиму совместимости с `vi`, в котором отключены многие полезные особенности. Флаг `-N` предотвращает это, устанавливая «несовместимый» режим.

Для большинства примеров в этой книге трюк с командой `vim -u NONE -N` гарантирует, что примеры будут выполняться в точности как описывается, кроме пары исключений. Некоторые встроенные особенности Vim реализованы в виде сценариев Vim, то есть они будут работать, только когда включена поддержка расширений. Следующий файл содержит минимальный набор настроек, необходимых для включения встроенных расширений Vim:

essential.vim
http://media.pragprog.com/titles/dnvim/code/essential.vim

```
set nocompatible
filetype plugin on
```

Чтобы использовать этот файл с настройками вместо стандартного *vimrc*, редактор Vim следует запустить командой:

```
⇒ $ vim -u code/essential.vim
```

При этом не забудьте исправить путь *code/essential.vim*, если необходимо. Включив встроенные расширения, вы сможете использовать такие особенности Vim, как `netrw` (см. рецепт 44 в главе 7) и `omni-completion` (см. рецепт 119 в главе 19), а также многие другие. Под настройками по умолчанию я подразумеваю включенную поддержку расширений и выключенный режим совместимости с `vi`.

Обязательно заглядывайте в раздел «Подготовка» в начале каждого рецепта. Чтобы воспроизвести пример, который приводится в рецепте, необходимо настроить Vim соответственно. Запуская Vim с настройками по умолчанию и применяя дополнительные настройки «на лету», вы сможете воспроизвести шаги, описываемые в рецепте, без каких-либо проблем.

Если вы по-прежнему будете испытывать проблемы, загляните в раздел «О версиях Vim» ниже.

О роли сценариев Vim

Сценарии Vim дают возможность добавлять в редактор новые функциональные возможности или изменять имеющиеся. Vim поддерживает полноценный язык сценариев, достойный отдельной книги, но «Практическое использование Vim» – не та книга.

Однако мы не будем сторониться этой темы полностью. Поддержка сценариев Vim всегда к нашим услугам, и мы рассмотрим несколько примеров использования их в повседневной работе в рецептах: 16 (глава 3), 71 (глава 11), 95 и 96 (глава 14).

Эта книга описывает приемы использования базовой функциональности Vim. Иными словами, мы не будем касаться сторонних расширений. Однако в рецепте 87 (глава 13) я сделаю исключение. Здесь я буду рекомендовать расширение `visual-star.vim`, добавляющее новые особенности, которые я считаю совершенно необходимыми. Его применение потребует написать совсем немного кода – менее десятка строк. Этот пример продемонстрирует, как легко расширяется функциональность Vim. Реализация `visual-star.vim` будет представлена без дополнительных пояснений. Взглянув на содержимое этих файлов, вы получите представление о том, как выглядят сценарии Vim и чего можно добиться с их помощью. Если эта демонстрация разбудит в вас интерес, тем лучше.

О версиях Vim

Все примеры в книге «Практическое использование Vim» были протестированы с последней версией Vim, каковой на момент написания этих строк была версия 7.3. Однако большинство примеров с успехом будут работать в любой версии 7.x, а многие особенности, обсуждаемые здесь, также доступны в версиях 6.x.

Некоторые особенности Vim могут отключаться на этапе компиляции. Например, при подготовке к сборке можно указать параметр `--with-features=tiny`, который отключит все особенности, кроме самых основных (существуют также наборы особенностей, помеченные как `small`, `normal`, `big` и `huge`). Посмотреть список особенностей можно с помощью команды `:h +feature-list` ⓘ http://vimdoc.sourceforge.net/htmldoc/various.html#+feature-list.

Если вы столкнетесь с отсутствием какой-либо особенности, обсуждаемой в книге, возможно, вы пользуетесь минимальной сборкой Vim. Проверьте доступность особенности с помощью команды `:version`:

```
⇛ :version
  VIM - Vi IMproved 7.3 (2010 Aug 15, compiled Sep 18 2011 16:00:17)
  Huge version with MacVim GUI. Features included (+) or not (-):
  +arabic +autocmd +balloon_eval +browse +builtin_terms +byte_offset
  +cindent +clientserver +clipboard +cmdline_compl +cmdline_hist
  +cmdline_info +comments
  ...
```

В современных компьютерах нет причин использовать набор особенностей Vim меньше, чем `huge`!.

Vim в терминале или Vim с графическим интерфейсом? Выбирать вам!

Традиционно Vim используется в терминале, без графического интерфейса, можно даже сказать, что Vim имеет только текстовый интерфейс. Если большую часть дня вы проводите в командной строке, для вас это будет выглядеть вполне естественно.

Если вы привыкли пользоваться текстовыми редакторами с графическим интерфейсом, тогда роль моста в мир Vim для вас может сыграть GVim (или MacVim в OS X) (см. `:h gui` ⓘ http://vimdoc.sourceforge.net/htmldoc/gui.html#gui). GVim поддерживает большое количество шрифтов и больше цветов для подсветки синтаксиса. Кроме того, он позволяет пользоваться мышью и следует некоторым соглашениям, принятым в операционной системе. Например, в MacVim поддерживаются: доступ к системному буферу обмена, с помощью комбинаций `Cmd`-`X` и `Cmd`-`V`, возможность сохранения документа комбинацией клавиш `Cmd`-`S` и закрытия окна комбинацией

клавиш `Cmd-W`. Вы можете использовать их, если вам это удобно, но знайте, что всегда есть более удачный путь.

Для нашего обсуждения совершенно не важно, какой версией вы будете пользоваться, редактором Vim в терминале или редактором GVim с графическим интерфейсом. Все наше внимание мы сконцентрируем на базовых командах, которые доступны в любом из этих редакторов. Мы узнаем, как выполнять свою работу, следуя *путем Vim*.

Глава 1. Путь Vim

Выполняемая нами работа содержит множество повторений. Будь то внесение одних и тех же небольших изменений в нескольких местах или перемещение между похожими фрагментами документа, мы повторно выполняем одни и те же действия. Все, что сможет упростить выполнение повторяющихся операций, поможет нам сэкономить наше время.

Редактор Vim оптимизирует такие повторяющиеся действия. Его эффективность во многом обусловлена возможностью отслеживать последовательность последних выполняемых операций. Мы всегда можем воспроизвести последние изменения нажатием одной клавиши. Это действительно очень мощная особенность, но она становится совершенно бесполезной для тех, кто не умеет фиксировать свои действия так, чтобы они выполняли полезную работу при повторном воспроизведении. Овладение данной концепцией является ключом к эффективному использованию Vim.

Команда «точка» станет нашей отправной точкой. Эта, казалось бы, простая команда – самый мощный инструмент в нашем арсенале и первый шаг на пути к овладению редактором Vim. Мы рассмотрим несколько простых задач редактирования, которые с помощью команды «точка» могут быть решены в мгновение ока. Хотя все задачи отличаются друг от друга, их решения во многом сходны. В каждом случае мы определим единую формулу редактирования, для выполнения которой требуется нажатие всего одной клавиши.

Рецепт 1. Встречайте: команда «точка»

Команда «точка» позволяет повторять последнее изменение. Это самая мощная и гибкая команда в редакторе Vim.

Документация к редактору Vim просто отмечает, что команда «точка»: «повторяет последнее изменение» (см. `:h .` ⓘ http://

vimdoc.sourceforge.net/htmldoc/repeat.html#). Вроде бы ничего особенного, но за этим простым определением скрывается мощная основа, делающая модель модального редактирования в Vim такой эффективной. Для начала выясним, что это за «последнее изменение».

Чтобы осознать всю мощь команды «точка», нужно понимать, что под «последним изменением» может скрываться все, что угодно. Изменением могут быть действия над отдельными символами, строками или даже над целым файлом.

Для демонстрации воспользуемся следующим фрагментом текста:

the_vim_way/0_mechanics.txt
http://media.pragprog.com/titles/dnvim/code/the_vim_way/0_mechanics.txt

```
Line one
Line two
Line three
Line four
```

Команда `x` удаляет символ под курсором. Когда команда «точка» используется в этом контексте, под «последним изменением» Vim будет понимать удаление символа под курсором:

Нажатия клавиш	Содержимое буфера
{start}	Line one Line two Line three Line four
x	ine one Line two Line three Line four
.	ne one Line two Line three Line four
..	one Line two Line three Line four

Вернуть файл в исходное состояние можно, нажав клавишу `u` несколько раз, чтобы отменить изменения.

Команда `dd` тоже выполняет удаление, но она удаляет текущую строку целиком. Если использовать команду «точка» после команды

dd, тогда под «последним изменением» Vim будет понимать удаление текущей строки:

Нажатия клавиш	Содержимое буфера
{start}	Line one Line two Line three Line four
dd	ine one Line two Line three Line four
.	Line three Line four

Наконец, команда >G увеличивает отступ, начиная с текущей строки, до конца файла. После выполнения этой команды под «последним изменением» Vim будет понимать увеличение отступа от текущей строки до конца файла. Этот пример мы начали с того, что поместили курсор в начало второй строки, чтобы подчеркнуть отличия.

Нажатия клавиш	Содержимое буфера
{start}	Line one Line two Line three Line four

>G	Line one
	  Line two
	  Line three
	  Line four

j	Line one
	  Line two
	  Line three
	  Line four

.	Line one
	  Line two
	    Line three
	    Line four

j.	Line one
	  Line two
	    Line three
	      Line four

Все команды – x, dd и > – выполняются в командном режиме, но изменения также создаются каждый раз при входе в режим вставки.

От момента входа в режим вставки (например, нажатием клавиши `i`) до выхода в командный режим (нажатием `<Esc>`) Vim записывает все нажатия клавиш. После создания такого изменения команда «точка» будет повторять эти нажатия (но прочитайте предупреждение во врезке «Перемещение по тексту в режиме вставки закрывает последнее изменение» в главе 2).

Команда «точка» – микромакроопределение

Ниже, в главе 11 «Макросы», будет показано, как в редакторе Vim организовать запись последовательности из произвольного количества нажатий клавиш для повторного воспроизведения ее в будущем. Эта возможность позволяет сохранять повторяющиеся операции и воспроизводить их нажатием одной клавиши. Команду «точка» можно считать миниатюрным, или «микро-» (если хотите), макроопределением.

На протяжении этой главы будет показано несколько вариантов применения команды «точка». Кроме того, пара приемов работы с командой «точка» будет показана в рецептах 9 (глава 2) и 43 (глава 4).

Рецепт 2. Не повторяйся

Для такой типичной операции, как добавление точки с запятой в конец строки, Vim предоставляет отдельную команду, объединяющую два шага в один.

Допустим, что у нас имеется следующий фрагмент кода на JavaScript:

the_vim_way/2_foo_bar.js
http://media.pragprog.com/titles/dnvim/code/the_vim_way/2_foo_bar.js

```
var foo = 1
var bar = 'a'
var foobar = foo + bar
```

Нам требуется добавить точку с запятой в конец каждой строки. Для этого нужно переместить курсор в конец текущей строки и затем перейти в режим вставки, чтобы выполнить изменение. Команда `$` выполнит такое перемещение, после чего мы сможем нажать последовательность `a;<Esc>`, чтобы выполнить изменение.

Чтобы выполнить задание полностью, нам могло бы потребоваться повторить ту же самую последовательность нажатий с остальными двумя строками, но это дело слишком уж нехитрое. Команда «точка» повторяет последнее изменение, поэтому мы можем просто выполнить `j$.` дважды. Одно нажатие (`.`) экономит три (`a;<Esc>`). Экономия получается не очень большая, но эффективность приема будет расти с ростом количества повторений.

А теперь рассмотрим поближе этот шаблон: `j$.`. Команда `j` перемещает курсор на одну строку вниз, а затем команда `$` перемещает его в конец строки. Нам пришлось нажать две клавиши, только чтобы вывести курсор в позицию, где можно применить команду «точка». Не кажется ли вам, что здесь есть что улучшить?

Избавляйтесь от лишних перемещений

Команда `a` переходит в режим вставки в позицию, находящуюся за позицией курсора. Однако есть команда `A`, выполняющая переход в режим вставки в позицию, находящуюся в конце текущей строки. Где бы ни находился курсор, нажатие на клавишу `A` выполнит переход в режим вставки и переместит курсор в конец строки. Иными словами, эта команда совмещает в себе последовательность команд `$a`. Во врезке «Две по цене одной» ниже мы узнаем, что в редакторе Vim имеется несколько подобных составных команд.

Ниже показано измененное решение предыдущей задачи:

Нажатия клавиш	Содержимое буфера

`{start}`	`var foo = 1`
	`var bar = 'a'`
	`var foobar = foo + bar`

`A;<Esc>`	`var foo = 1;`
	`var bar = 'a'`
	`var foobar = foo + bar`

`j`	`var foo = 1;`
	`var bar = 'a'`
	`var foobar = foo + bar`

`.`	`var foo = 1;`
	`var bar = 'a';`
	`var foobar = foo + bar`

`j.`	`var foo = 1;`
	`var bar = 'a';`
	`var foobar = foo + bar;`

Используя `A` вместо `$a`, мы расширяем команду «точка». Вместо того чтобы перемещать курсор в *конец* строки, которую требуется изменить, мы просто перемещаем курсор в эту строку (в *любую позицию* в ней!). Теперь мы можем внести изменения в последовательность строк, просто нажимая `j.` столько раз, сколько потребуется.

Одно нажатие – чтобы переместить курсор, и одно нажатие – чтобы выполнить изменение. Разве это не здорово?! Запомните этот прием, потому что мы встретимся с ним еще в нескольких примерах.

Несмотря на свою потрясающую простоту, эта формула не является универсальным решением. Представьте, что нам потребовалось добавить точку с запятой в пятьдесят строк, следующих друг за другом. В этом случае нажатие пары клавиш `j.` превращается в утомительный труд. Альтернативное решение этой задачи вы найдете в рецепте 30 в главе 5.

Две по цене одной

Мы могли бы сказать, что команда `A` состоит из двух действий (`$a`), совмещенных в одном нажатии клавиши. Однако она не является единственной командой такого рода. Многие команды Vim можно рассматривать как одноклавишные версии последовательностей из двух или более команд. В таблице ниже перечислено несколько таких команд. Сможете ли вы определить, что их объединяет?

Составная команда	Эквивалентная последовательность
`C`	`c$`
`s`	`cl`
`S`	`^C`
`I`	`^i`
`A`	`$a`
`o`	`A<CR>`
`O`	`ko`

Если вы вдруг обнаружите, что все время выполняете последовательность `ko` (или хуже того, `k$a<CR>`), остановитесь! Подумайте о том, что вы делаете. Затем вспомните, что можно было бы использовать более простую команду `O`.

Сумеете ли вы заметить другую общую черту, объединяющую эти команды? Все они выполняют переход из командного режима в режим вставки. Поразмышляйте над этим и над тем, как это может влиять на команду «точка».

Рецепт 3. Шаг назад, три вперед

Мы можем дополнить единственный символ пробелами (один слева, другой справа), используя идиоматическое для Vim решение. Сначала оно будет казаться странным, но это решение дает возможность повторения, что позволяет выполнить задание с минимумом усилий.

Представьте, что имеется такая строка кода:

the_vim_way/3_concat.js
http://media.pragprog.com/titles/dnvim/code/the_vim_way/3_concat.js

```
var foo = "method("+argument1+","+argument2+")";
```

Конкатенация строк в JavaScript никогда не выглядела особенно удобочитаемо, тем не менее мы могли бы повысить удобочитаемость, окружая каждый знак + пробелами, чтобы строка кода выглядела, как показано ниже:

```
var foo = "method(" + argument1 + "," + argument2 + ")";
```

Делайте изменения повторяемыми

Описываемая задача имеет следующее идиоматическое решение:

Нажатия клавиш	Содержимое буфера
`{start}`	`var foo = "method("+argument1+","+argument2+")";`
`f+`	`var foo = "method("+argument1+","+argument2+")";`
`s␣+␣<Esc>`	`var foo = "method(" +argument1+","+argument2+")";`
`;`	`var foo = "method(" + argument1+","+argument2+")";`
`.`	`var foo = "method(" + argument1 + ","+argument2+")";`
`;.`	`var foo = "method(" + argument1 + "," +argument2+")";`
`;.`	`var foo = "method(" + argument1 + "," + argument2 +")";`

Команда `s` объединяет два шага в один: она удаляет символ под курсором и переходит в режим вставки. После удаления символа + мы вводим `␣+␣` и покидаем режим вставки.

Один шаг назад и затем три вперед. Этот странный танец на клавиатуре сначала выглядит малопонятным, но он дает нам большой выигрыш: мы можем повторять изменения с помощью команды «точка»: все, что для этого нужно, – переместить курсор к следующему символу +, после чего команда «точка» повторит этот маленький танец.

Делайте перемещения повторяемыми

В этом примере присутствует еще одна хитрость. Команда `f{char}` сообщает редактору Vim, что он должен найти следующее вхождение указанного символа и переместить курсор в его позицию (см. `:h f` ⓘ http://vimdoc.sourceforge.net/htmldoc/motion.html#f). Поэтому, когда вводится последовательность `f+`, курсор перепрыгивает к следующему символу `+`. Подробнее о команде `f{char}` рассказывается в рецепте 50 в главе 8.

Выполнив первое изменение, мы могли бы перейти к следующему символу `+`, повторив команду `f+`, но есть более простой путь. Команда `;` повторяет последний поиск, произведенный с помощью команды `f`. Поэтому вместо команды `f+` мы можем использовать эту команду.

Теперь все вместе

Команда `;` переносит курсор в позицию следующего символа `+`, а команда `.` повторяет последнее изменение, поэтому мы можем внести оставшиеся изменения, трижды введя последовательность `;.`. Не кажется ли вам такой алгоритм действий знакомым?

Вместо того чтобы бороться с модальной моделью ввода в Vim, мы используем ее преимущества и убеждаемся, насколько она способна упростить решение поставленной задачи.

Рецепт 4. Действие, повтор, возврат

Сталкиваясь с повторяющимися задачами, мы можем выработать еще более оптимальную стратегию редактирования, делая одновременно перемещения и изменения повторяющимися. В редакторе Vim имеется все необходимое для этого. Он запоминает наши действия и сохраняет их так, чтобы мы легко могли воспользоваться ими. В этом рецепте мы познакомимся с каждым из действий, которые Vim способен повторить, и узнаем, как отменить их.

Мы уже знаем, что команда «точка» повторяет последнее *изменение*. Поскольку существует огромное количество команд, которые могут интерпретироваться как изменения, команда «точка» оказывается чрезвычайно универсальной. Но некоторые команды могут повторяться другими средствами. Например, `@:` может повторить любую команду редактора Ex (как описывается в рецепте 31 в главе 5). Точно так же, нажав `&` (см. рецепт 93 в главе 14), можно повто-

рить последнюю команду подстановки `:substitute` (которая также является командой редактора Ex).

Если известно, как повторять действия, избегая необходимости вводить их каждый раз, мы можем добиться высочайшей эффективности. Сначала мы выполняем действие, затем повторяем его.

Но когда многого можно добиться лишь несколькими нажатиями клавиш, приходится быть внимательными. Слишком легко допустить ошибку. Многократное повторение `j.j.j.` снова и снова начинает напоминать барабанную дробь. Что, если по ошибке мы нажмем клавишу `j` два раза подряд? Или хуже того, два раза нажмем клавишу `.`?

Всякий раз, когда редактор Vim упрощает повторение действия или перемещения, он всегда предоставляет способ отступить назад, если, войдя в раж, мы зайдем слишком далеко. В случае с командой «точка» мы всегда можем нажать клавишу `u`, чтобы отменить последнее изменение. Если лишний раз нажать клавишу `;` после использования команды `f{char}`, можно пропустить искомый символ. Но мы можем вернуться назад, нажав клавишу `,`, которая повторяет последнюю команду поиска `f{char}` в обратном направлении (см. рецепт 50 в главе 8).

Всегда полезно знать, как включается обратная передача, на случай, если по ошибке мы зайдем слишком далеко. В табл. 1.1 перечислены команды повторения, поддерживаемые редактором Vim, и соответствующие им обратные команды. В большинстве случаев можно использовать команду отмены. Неудивительно, что самая потертая клавиша на моей клавиатуре – это клавиша `u`!

Таблица 1.1. Команды повторения и обратные им команды

Цель	Команда	Повторение	Возврат
Выполнить изменение	`{edit}`	`.`	`u`
Найти следующий символ в строке	`f{char}`/`t{char}`	`;`	`,`
Найти предыдущий символ в строке	`F{char}`/`T{char}`	`;`	`,`
Найти следующее совпадение в документе	`/pattern<CR>`	`n`	`N`
Найти предыдущее совпадение в документе	`?pattern<CR>`	`n`	`N`
Выполнить подстановку	`:s/target/replacement`	`&`	`u`
Выполнить последовательность изменений	`qx{changes}q`	`@x`	`u`

Рецепт 5. Поиск и замена вручную

Для выполнения поиска с заменой в Vim имеется команда `:substitute`, но существует альтернативный прием, когда мы изменяем первое вхождение вручную, а затем отыскиваем и заменяем каждое последующее вхождение по одному. Команда «точка» могла бы спасти нас от переутомления, но существует другая, более привлекательная команда, осуществляющая переход между совпадениями.

В следующем фрагменте текста в каждой строке встречается слово «content»:

the_vim_way/1_copy_content.txt
http://media.pragprog.com/titles/dnvim/code/the_vim_way/1_copy_content.txt

```
...We're waiting for content before the site can go live...
...If you are content with this, let's go ahead with it...
...We'll launch as soon as we have the content...
```

Допустим, что нам понадобилось заменить каждое слово «content» словом «copy». Вы можете подумать, что сделать это очень просто – достаточно лишь воспользоваться командой подстановки:

```
➾ :%s/content/copy/g
```

Но минутку! В этом случае фраза «If you are content with this» («Если вас это устраивает») превратится во фразу «If you are 'copy' with this» («Если вас это копия»), лишенную смысла!

Мы столкнулись с проблемой, потому что слово «content» имеет два смысла. Один из них – синоним слова «copy» (копия), а другой – синоним слова «happy» (довольный). Технически мы имеем дело с омонимами (словами, разными по значению, но одинаковыми по написанию), но не это здесь главное. Главное здесь то, что вы должны следить за своими действиями.

Мы не можем слепо заменить каждое вхождение слова «content» словом «copy». Мы должны оценить каждое из найденных вхождений и ответить «да» или «нет» на вопрос о замене. Команда подстановки имеет свою область применения, и мы познакомимся с ней поближе в рецепте 90 в главе 14. А пока займемся альтернативным решением, соответствующим теме данной главы.

Будьте экономны: выполняйте поиск без ввода лишних символов

Возможно, вы уже догадались, что команда «точка» – моя любимица. На втором месте у меня стоит команда `*`. Она выполняет поиск слова под курсором (см. `:h *` ⓘ http://vimdoc.sourceforge.net/htmldoc/pattern.html#star).

Мы можем выполнить поиск слова «content», напечатав его в строке приглашения к вводу:

```
⇛ /content
```

или просто поместив курсор в требуемое слово и нажав клавишу `*`. Взгляните, как выглядит весь процесс:

Нажатия клавиш	Содержимое буфера

`{start}`	`...We're waiting for content before the site can go live...`
	`...If you are content with this, let's go ahead with it...`
	`...We'll launch as soon as we have the content...`

`*`	`...We're waiting for content before the site can go live...`
	`...If you are content with this, let's go ahead with it...`
	`...We'll launch as soon as we have the content...`

`cwcopy<Esc>`	`...We're waiting for content before the site can go live...`
	`...If you are content with this, let's go ahead with it...`
	`...We'll launch as soon as we have the copy...`

`n`	`...We're waiting for content before the site can go live...`
	`...If you are content with this, let's go ahead with it...`
	`...We'll launch as soon as we have the content...`

`.`	`...We're waiting for copy before the site can go live...`
	`...If you are content with this, let's go ahead with it...`
	`...We'll launch as soon as we have the content...`

Процесс начинается с установки курсора в слово «content», после чего вызывается команда `*` поиска. Попробуйте сами выполнить это упражнение. В результате случится следующее: курсор перепрыгнет вперед, к следующему совпадению, и все найденные вхождения будут выделены цветом. Если вы не увидите подсвеченные совпадения, попробуйте выполнить команду `:set hls` и затем обратиться к рецепту 81 в главе 13 за более подробными разъяснениями.

Выполнив поиск слова «content», перейти к следующему совпадению можно простым нажатием клавиши `n`. В данном случае нажатие `*nn` выполнит обход всех совпадений и вернет нас туда, откуда мы начали.

Делайте изменения повторяемыми

После установки курсора в начало слова «content» мы можем изменить его. Для этого требуется выполнить два шага: удалить слово «content» и затем ввести слово замены. Команда `cw` удаляет символы от курсора до конца слова и выполняет переход в режим вставки, после чего можно ввести слово «copy». Vim записывает все нажатия клавиш, вплоть до выхода из режима вставки, поэтому вся последовательность `cwcopy<Esc>` рассматривается как единое изменение. Нажатие клавиши `.` удалит символы до конца текущего слова и заменит его словом «copy».

Теперь все вместе

Теперь все готово к действию! При каждом нажатии клавиши `n` курсор перемещается к следующему вхождению слова «content». А при нажатии клавиши `.` слово под курсором замещается словом «copy».

Если бы нам потребовалось слепо заменить все вхождения, мы могли бы просто отстучать на клавиатуре `n.n.n.` столько раз, сколько потребовалось бы для замены всех вхождений (хотя в этом случае можно было бы также использовать команду `:%s/content/copy/g`). Но нам нужно пропускать слова, не подлежащие замене. Поэтому после нажатия клавиши `n` мы проверяем смысл найденного совпадения и решаем, следует ли заменить его словом «copy». Если слово должно быть заменено, нажимаем клавишу `.`. Если нет – не нажимаем. Независимо от принятого решения, чтобы перейти к следующему вхождению, снова нажимаем клавишу `n`. И так, пока не будет выполнена замена всех требуемых вхождений.

Рецепт 6. Формула точки

Мы рассмотрели три простые задачи редактирования. Несмотря на различия между ними, мы смогли решить их с помощью команды «точка». В данном рецепте мы сравним найденные решения и выявим общий шаблон – оптимальную стратегию редактирования, которую я назвал «формула точки».

Обзор решений трех задач редактирования с помощью команды «точка»

В рецепте 2 нам потребовалось добавить точку с запятой в конец каждой строки. Мы изменили первую строку нажатием клавиш `A;<Esc>`, благодаря чему появилась возможность повторить изменение в каждой последующей строке с помощью команды «точка». Мы могли бы перемещаться между строками, выполняя команду `j`, и выполнить оставшиеся изменения, просто последовательно нажимая `j.` столько раз, сколько потребуется.

В рецепте 3 нам понадобилось окружить каждый символ `+` пробелами с обеих сторон. Чтобы найти этот символ в тексте, мы воспользовались командой `f+` и затем задействовали команду `s`, чтобы заменить один символ тремя. Благодаря такому подходу мы получили возможность выполнить поставленную задачу, нажав клавиши `;.` несколько раз.

В рецепте 5 нам нужно было заменить каждое вхождение слова «content» словом «copy». Для инициации поиска требуемого слова мы использовали команду `*` и затем выполнили команду `cw` для изменения первого вхождения. После этого мы получили возможность использовать команду `n` для перехода к следующему вхождению и команду `.` для применения того же изменения. Мы могли бы выполнить поставленную задачу, просто нажимая клавиши `n.` столько раз, сколько потребуется.

Идеальное решение: одна клавиша для перехода и одна для изменения

Во всех примерах мы использовали команду «точка», чтобы повторить последнее изменение. Но это не единственная общая черта. Для перемещения к следующей позиции применения изменений также потребовалось единственное нажатие клавиши.

Мы использовали одно нажатие для перемещения курсора и одно нажатие для выполнения изменения. Разве можно придумать что-то еще более удобное? Это – идеальное решение. Мы еще не раз столкнемся с этой стратегией редактирования, поэтому для краткости будем называть ее «формула точки».

Часть I. РЕЖИМЫ

Редактор Vim имеет модальный пользовательский интерфейс. То есть результат нажатия любой клавиши зависит от режима, активного в данный момент. Очень важно знать, какой режим активен в данный момент и как переключать режимы Vim. В этой части книги вы узнаете, как действует каждый режим и для каких целей они используются.

Глава 2. Командный режим

Командный режим – это естественное состояние Vim. Если эта глава покажется вам на удивление короткой, то только потому, что большая часть книги посвящена описанию особенностей работы в командном режиме! Однако здесь рассматриваются самые основные понятия и даются самые универсальные рецепты.

Другие текстовые редакторы большую часть времени находятся в режиме, напоминающем режим вставки. Начинающим осваивать Vim часто кажется странным, что командный режим является режимом по умолчанию. В рецепте 7 ниже объясняется, почему именно так действует Vim, по аналогии с холстом картины.

Многие команды командного режима можно запускать со счетчиком, чтобы выполнить их определенное количество раз. В рецепте 10 далее мы познакомимся с парой команд, увеличивающих и уменьшающих числовые значения, и посмотрим, как эти команды можно объединять со счетчиком для выполнения простых арифметических операций.

Возможность с помощью счетчиков сэкономить на нажатиях клавиш не означает, что этот прием обязательно следует использовать. Мы увидим несколько примеров, когда проще повторить команду, чем тратить время на ввод значения счетчика.

Широта возможностей командного режима в значительной степени обусловлена способом объединения операторов с командами перемещения курсора. И закончим мы эту главу рассмотрением последствий такой возможности.

Рецепт 7. Оторвите кисть от холста

Тем, кто никогда не пользовался редактором Vim, кажется странным, что командный режим выбран в качестве режима по умолчанию. Но опытные пользователи Vim представить себе не могут

иного способа работы. Для иллюстрации пути Vim в этом рецепте используется аналогия.

Как вы считаете, сколько времени кисть художника находится в контакте с холстом? Разумеется, у разных художников это время будет разным, но я буду удивлен, если мне встретится художник, у которого это время составляет хотя бы половину времени работы над картиной.

Только представьте, сколько всего делают художники, помимо нанесения мазков. Они изучают изображаемый предмет, оценивают освещение и смешивают краски, подбирая нужные оттенки. А когда приходит момент нанести краску на холст, они не всегда используют кисти. Художник может использовать для этого шпатель, чтобы добиться определенной текстуры, или растереть уже нанесенную краску ватным тампоном.

Художник не удерживает кисть в постоянном контакте с холстом. То же происходит и при работе с редактором Vim. Командный режим – естественное его состояние.

Так же, как художники тратят лишь часть времени на нанесение краски, программисты тратят лишь часть времени на набор кода. Большую часть времени они тратят на раздумья, чтение документации и перемещение от одного участка программы к другому. И кто сказал, что мы должны переходить в режим вставки, чтобы внести изменения? Мы можем изменять форматирование кода, копировать, перемещать или удалять его. Для этого можно использовать множество инструментов, доступных в командном режиме.

Рецепт 8. Группируйте изменения для возможной отмены

В других текстовых редакторах вызов команды отмены после ввода нескольких слов может отменить ввод последнего слова или символа. В Vim мы можем управлять избирательностью команды отмены.

Нажатие на клавишу `u` вызывает команду отмены ввода, которая отменяет самое последнее изменение. Под изменением понимаются любые команды, модифицирующие текст документа, включая команды, выполняемые в командном или визуальном режиме, или в режиме командной строки, а также ввод (или удаление) текста в режиме вставки. То есть можно сказать, что последовательность `i{вставка некоторого текста}<Esc>` составляет одно изменение.

Команда отмены в немодальных текстовых редакторах, после ввода нескольких слов, может делать одно из двух: отменить ввод последнего символа или множества символов, составляющих слово.

Редактор Vim позволяет управлять избирательностью команды отмены. От момента входа в режим вставки и до момента выхода в командный режим любые последовательности операций ввода (или удаления) текста рассматриваются как единственное изменение. Благодаря этому можно заставить команду отмены оперировать словами, предложениями или абзацами, используя для этого клавишу `<Esc>`.

Так как же часто следует покидать режим вставки? Это зависит от личных предпочтений, но лично я привык, что «блоки отмены» соответствуют одной законченной мысли. Когда я пишу эти строки (в Vim, конечно же!), я часто останавливаюсь в конце предложения, чтобы подумать о том, что я напишу дальше. Не важно, насколько коротки эти паузы, – любая из них является естественной остановкой, дающей возможность покинуть режим вставки. Когда я готов продолжить, я нажимаю `A` и возвращаюсь к месту, где закончил предыдущую мысль.

Если я решу, что использовал неправильный оборот, то перейду в командный режим и нажму клавишу `u`. Каждый раз, когда я выполняю отмену, воздействию подвергаются фрагменты текста, соответствующие моему образу мышления. Это означает, что я легко могу написать одно-два предложения и затем удалить их парой нажатий на клавиши.

Проще всего начать новую строку, находясь в режиме вставки, – нажать `<CR>`. И все же я предпочитаю нажать клавиши `<Esc>o` просто потому, что мне хочется получить чуть больше избирательности от команды отмены. Если такой способ покажется вам чрезмерным, не волнуйтесь. По мере овладения мастерством работы с Vim переключение режимов для вас будет становиться все более и более естественным делом.

Главное правило, если вы приостанавливаетесь на достаточно длинный промежуток времени, чтобы задаться вопросом: «А не следует ли мне выйти из режима вставки?», – выполните такой выход.

Перемещение по тексту в режиме вставки закрывает последнее изменение

Когда я говорил, что команда отмены отменит любые изменения в тексте, выполненные в промежутке между входом в режим вставки и выхо-

дом из него, я умолчал об одной маленькой детали. При каждом нажатии клавиш со стрелками – <Up>, <Down>, <Left> или <Right> – создается новый блок отмены изменений. Действие этих клавиш равноценно переключению в командный режим и использованию команд `h`, `j`, `k` и `l` для перемещения по тексту, только при этом редактор остается в режиме вставки. Эта особенность также влияет на действие команды «точка».

Рецепт 9. Составляйте повторяемые изменения

Редактор Vim оптимизирован для выполнения повторяющихся операций. Чтобы использовать эту его возможность, следует внимательнее относиться к самим изменениям.

В Vim всегда доступно более одного способа выполнения каких-то операций. Самой важной оценкой этих способов является эффективность, то есть лучшим считается тот способ, который требует меньше нажатий клавиш (своеобразная игра в Vim-гольф ℹ http://vimgolf.com/). Но как выявить победителя в случае ничейного счета?

Представьте, что курсор находится в позиции символа «h», в конце следующей строки текста, и нам требуется удалить слово «nigh».

normal_mode/the_end.txt
http://media.pragprog.com/titles/dnvim/code/normal_mode/the_end.txt

```
The end is nigh
```

Удаление назад

Так как курсор уже находится в конце слова, можно воспользоваться командой удаления назад.

Нажатия клавиш	Содержимое буфера
{start}	The end is nigh
db	The end is h
x	The end is

Команда `db` удалит все символы, находящиеся левее курсора, до начала слова, но оставит символ «h» под курсором. Удалить этот символ можно командой `x`. Эта операция зарабатывает три очка.

Удаление вперед

Теперь попробуем выполнить удаление вперед.

Нажатия клавиш	Содержимое буфера
`{start}`	`The end is nigh`
`b`	`The end is nigh`
`dw`	`The end is`

Мы совершили маневр курсором, выполнив команду `b`. После чего удалили слово единственной командой `dw`. Эта операция также зарабатывает три очка.

Удаление целого слова

Оба решения, представленные выше, включают операцию подготовки к удалению. Однако у нас есть возможность использовать более тонкий хирургический инструмент – текстовый объект `aw` (см. `:h aw` ⓘ http://vimdoc.sourceforge.net/htmldoc/motion.html#aw):

Нажатия клавиш	Содержимое буфера
`{start}`	`The end is nigh`
`daw`	`The end is`

Команда `daw` легко запоминается благодаря мнемонике *delete a word* (удалить слово). Подробнее о текстовых объектах мы поговорим в рецептах 52 и 53 (глава 8).

Дополнительная оценка: какой вариант более повторим?

Итак, мы опробовали три способа удаления слова: `dbx`, `bdw` и `daw`. Каждый из них набрал три очка. Так как же выявить победителя?

Вспомните, как выше говорилось, что редактор Vim оптимизирован для выполнения повторяющихся операций. Давайте окинем еще раз взглядом эти три способа. На этот раз попробуем задействовать команду «точка» и посмотрим, что из этого получится. Я настоятельно рекомендую вам самим попробовать проделать это.

Прием удаления назад состоит из двух операций: `db` удаляет слово до начала, затем `x` удаляет единственный символ. Если после этого вызвать команду «точка», она повторит операцию удаления единственного символа (`.` == `x`). Я бы не назвал это победой.

Прием удаления вперед также состоит из двух операций, где `b` выполняет перемещение курсора, а `dw` производит изменение. Команда «точка» повторит команду `dw` и удалит символы от позиции курсора до начала следующего слова. Так случилось, что после удаления мы

оказались в конце строки, и далее в тексте нет никакого «следующего слова», поэтому в данном случае команда «точка» оказывается бесполезной. Но она хотя бы является теперь коротким вариантом более длинной команды (`.` == `dw`).

Последнее решение состоит из единственной операции: `daw`. Эта команда удаляет не только слово, но и пробельный символ. Как результат курсор оказался в позиции последнего символа в слове «is». Если теперь выполнить команду «точка», она повторит операцию удаления слова. На этот раз команда «точка» оказалась куда полезней (`.` == `daw`).

Обсуждение

Прием на основе команды `daw` наделил команду «точка» более широкими возможностями, поэтому я объявляю его победителем данного раунда.

Часто, чтобы повысить эффективность команды «точка», требуется проявить некоторую предусмотрительность. Если вы обнаруживаете, что одно и то же изменение необходимо выполнить в нескольких местах, попробуйте скомпоновать изменения так, чтобы их можно было повторить с помощью команды «точка». Чтобы научиться этому, нужна практика. Но как только вы выработаете привычку компоновать повторяемые изменения, Vim вознаградит вас.

Иногда я просто не замечаю возможности применить команду «точка». Но после внесения изменения – обнаружив место, где требуется аналогичная правка, – я, к своему удовольствию, замечаю, что команда «точка» готова выполнить всю работу за меня, и каждый раз это вызывает у меня улыбку.

Рецепт 10. Используйте счетчики для простых арифметических операций

Большинство команд командного режима предусматривает возможность выполнения со счетчиком. Эту особенность можно использовать для простых арифметических вычислений.

Многие команды, доступные в командном режиме могут снабжаться счетчиком. Вместо одного раза Vim попытается выполнить команду указанное количество раз (см. `:h count` ⓘ http://vimdoc.sourceforge.net/htmldoc/intro.html#count).

Команды `<C-a>` и `<C-x>` выполняют сложение и вычитание чисел. При выполнении без счетчика они увеличивают и уменьшают

число под курсором соответственно, но если им предшествует ввод числа, с их помощью можно складывать и вычитать любые целые числа. Например, если установить курсор в позицию символа «5» и выполнить `10<C-a>`, число 5 заменит число 15.

А если под курсором окажется не цифра? В документации говорится, что команда `<C-a>` *«добавит [счетчик] к числу под курсором или после курсора»* (см. `:h ctrl-a` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#CTRL-A). То есть если курсор находится за пределами числа, команда `<C-a>` попытается найти число правее курсора в этой же строке. Если число будет найдено, курсор будет перемещен на него. Мы можем использовать эту особенность для своей выгоды.

Ниже приводится фрагмент таблицы стилей CSS:

normal_mode/sprite.css
http://media.pragprog.com/titles/dnvim/code/normal_mode/sprite.css

```
.blog, .news { background-image: url(/sprite.png); }
.blog { background-position: 0px 0px }
```

Сейчас мы скопируем последнюю строку и внесем два небольших изменения в копию: заменим слово «blog» словом «news» и «0px» на «-180px». Мы можем скопировать строку командой `yyp` и затем с помощью `cw` изменить первое слово. Но как лучше поступить с числом?

Можно просто перейти к цифре командой `f0` и затем, войдя в режим вставки, изменить число вручную: `i-18<Esc>`. Но гораздо быстрее получится ввести `180<C-x>`. Поскольку курсор находится не на цифре, команда выполнит поиск числа правее курсора. Это избавит нас от необходимости перемещать курсор вручную. Рассмотрим поближе весь процесс:

Нажатия клавиш	Содержимое буфера
`{start}`	`.blog, .news { background-image: url(/sprite.png); }` `.blog { background-position: 0px 0px }`

`yyp`	`.blog, .news { background-image: url(/sprite.png); }`
	`.blog { background-position: 0px 0px }`
	`.blog { background-position: 0px 0px }`

`cW.news<Esc>`	`.blog, .news { background-image: url(/sprite.png); }`
	`.blog { background-position: 0px 0px }`
	`.news { background-position: 0px 0px }`

`180<C-x>`	`.blog, .news { background-image: url(/sprite.png); }`
	`.blog { background-position: 0px 0px }`
	`.news { background-position: -180px 0px }`

В этом примере мы скопировали лишь одну строку и изменили ее. А теперь представьте, что нам нужно скопировать строку 10 раз и вычесть 180 из числа в каждой следующей копии. Если бы мы были вынуждены переключаться в режим вставки, чтобы исправить каждое число, нам пришлось бы вводить в каждой копии разные числа (–180, затем –360 и т. д.). Но, воспользовавшись командой `180<C-x>`, на каждом этапе мы будем выполнять одни и те же действия. Мы можем даже записать последовательность нажатий клавиш, сохранить ее в виде макроса (см. главу 11 «Макросы») и затем воспроизводить ее столько раз, сколько потребуется.

Числовые форматы

Что следует за 007? Нет, я не спрашиваю про соратника Джеймса Бонда; я спрашиваю – что получится, если к числу 007 прибавить единицу.

Если вы считаете, что получится число 008, то будете удивлены, попробовав применить команду `<C-a>` к любому числу, начинающемуся с нуля. Как и многие языки программирования, Vim интерпретирует последовательности цифр, начинающиеся с нуля, как восьмеричные числа. В восьмеричной системе счисления `007 + 001 = 010`. Результат похож на десятичное число 10, но в действительности это восьмеричная восьмерка. Удивлены?

Если вам часто приходится работать с восьмеричными числами, поведение по умолчанию редактора Vim может вполне устраивать вас. А если нет, возможно, у вас появится желание добавить следующую строку в свой файл *vimrc*:

```
set nrformats=
```

Она заставит Vim интерпретировать все числа как десятичные, независимо от наличия ведущего нуля.

Рецепт 11. Не занимайтесь подсчетами, если можно выполнить повторение

Поддержка счетчиков позволяет уменьшить количество нажатий на клавиши при решении некоторых задач, но это не значит, что мы обязательно должны пользоваться ею. Рассмотрим «за» и «против» счетчиков в сравнении с простым повторением.

Допустим, что в буфере имеется следующий текст:

```
Delete more than one word
```

Нам требуется сделать то, о чем говорится в этой строке (удалить несколько слов), чтобы в буфере остался текст «Delete one word». То есть нам нужно удалить два слова.

Добиться этого можно несколькими способами: `d2w` и `2dw`. Последовательность `d2w` вызывает команду удаления и затем передает ей `2w` как объем удаления. Ее можно прочитать как «delete 2 words» (удалить 2 слова). Последовательность `2dw` переворачивает все с ног на голову. На этот раз счетчик относится к команде удаления, которой передается объем удаления, равный одному слову. Данную последовательность можно прочитать как «delete a word two times» (дважды удалить слово). Невзирая на разную семантику, мы получаем одинаковый результат.

А теперь представьте альтернативу: `dw.`. Эту последовательность можно прочитать как «delete a word and then repeat» (удалить слово и повторить).

Итак, у нас имеются три варианта: `d2w`, `2dw` и `dw.` – каждый состоит из трех нажатий на клавиши. Но какой из них лучший?

С позиции обсуждаемой темы, последовательности `d2w` и `2dw` можно считать идентичными. После выполнения любой из них мы можем нажать клавишу `u`, и два удаленных слова появятся снова. Или вместо отмены мы могли бы воспользоваться командой «точка», чтобы удалить следующие два слова.

В случае последовательности `dw.` действие команд `u` и `.` несколько отличается. Здесь изменение выполняет команда `dw` – «delete word» (удалить слово). Поэтому, чтобы восстановить два удаленных слова, потребуется нажать клавишу `u` дважды: `uu` (или `2u`, если хотите). Команда «точка» удалит только одно слово, а не два.

Теперь представьте, что первоначально мы хотели удалить не два, а три слова. Из-за маленькой оплошности вместо `d3w` мы ввели `d2w`. И что дальше? Мы уже не можем использовать команду «точка», потому что тогда в общей сложности будет удалено четыре слова. Из-за этого нам придется вернуться обратно и указать иное значение счетчика (`ud3w`) или просто удалить следующее слово (`dw`).

Но если бы мы сразу использовали команду `dw.`, то могли бы ограничиться единственным повторением команды «точка». Поскольку в таком случае изменение было бы выполнено простой ко-

мандой `dw`, команды `u` и `.` получили бы большую точность. Каждая из них воздействовала бы только на одно слово.

Теперь допустим, что требуется удалить семь слов. Можно было бы выполнить команду `d7w` или `dw......` (то есть команду `dw` и шесть команд «точка»). По количеству нажатий на клавиши первый способ одерживает чистую победу. Но уверены ли вы на все сто процентов, что не ошиблись в подсчетах?

Заниматься подсчетами довольно утомительно. Я лучше шесть раз нажму на точку, чем потрачу то же самое время на подсчет слов, чтобы потом сэкономить на нажатиях клавиш. Что случится, если я нажму на точку лишний раз? Ничего страшного, я просто нажму один раз на клавишу `u`.

Вспомните мантру из рецепта 4 в главе 1: *действие, повтор, возврат*. Здесь она показана в действии.

Используйте счетчик, когда в этом есть смысл

Допустим, нам требуется изменить текст «I have a couple of questions» (у меня есть пара вопросов), чтобы получилось «I have some more questions» (у меня есть еще несколько вопросов). Сделать это можно так, например:

Нажатия клавиш	Содержимое буфера
`{start}`	`I have a couple of questions.`
`c3wsome more<Esc>`	`I have some more questions.`

В данном случае нет особого смысла использовать команду «точка». Мы могли бы удалить одно слово, потом другое (с помощью команды «точка»), но тогда нам пришлось бы переключать передачи и входить в режим вставки (командой `i` или `cw`, например). Для меня, который и так достаточно неуклюж, этого уже достаточно, чтобы заглянуть вперед и подсчитать слова.

Но у счетчиков есть еще одно важное преимущество: они обеспечивают ясную и согласованную историю отмен. Выполнив такое изменение, мы сможем отменить его одним нажатием клавиши `u`, что согласуется с обсуждением в рецепте 8 выше.

Тот же аргумент можно привести в пользу подсчета (`d5w`) перед повторением (`dw....`), поэтому я могу показаться непоследовательным в своих предпочтениях. Постепенно вы придете к собственному мнению, в зависимости от того, насколько важно для вас иметь

ясную историю отмен и насколько для вас утомительно заниматься подсчетами.

Рецепт 12. Объединяй и властвуй

Широта возможностей редактора Vim в немалой степени обусловлена его способностью объединять команды операций с командами перемещения курсора (motion). В этом рецепте мы увидим, как это делается, и остановимся на следствиях.

Оператор + команда перемещения = Действие

Команда `d{motion}` может оперировать единственным символом (`dl`), словом (`daw`) или целым абзацем (`dap`). Это достигается сочетанием оператора с командой перемещения. То же относится к командам `c{motion}`, `y{motion}` и некоторым другим. Все вместе эти команды называются *операторами*. Полный список операторов можно найти в справке `:h operator` ⓘ http://vimdoc.sourceforge.net/htmldoc/motion.html#operator, однако наиболее употребительные из них приводятся в табл. 2.1 ниже.

Команды `g~`, `gu` и `gU` вызываются нажатием двух клавиш. В каждом из этих случаев символ `g` можно рассматривать как префикс, изменяющий поведение последующей клавиши. Дальнейшее обсуждение приводится во врезке «Режим ожидающего оператора» ниже.

Объединение операторов с командами перемещения курсора образует своеобразную грамматику. Основное правило легко запомнить: действие определяется оператором, за которым указывается команда перемещения курсора. Изучение новых команд перемещения и операторов напоминает изучение словаря Vim. Следуя простым грамматическим правилам, мы сможем свободнее выражать свои мысли по мере увеличения словаря.

Допустим, что мы уже знаем, как удалить слово командой `daw`, и потом мы узнали о команде `gU` (см. `:h gU` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#gU). Это тоже оператор, поэтому мы можем вызвать команду `gUaw`, чтобы преобразовать текущее слово в КРИЧАЩИЙ регистр. Если потом наш словарь расширится и в нем появится команда перемещения `ap`, определяющая абзац, мы обнаружим две новые операции: `dap` – выполняющую удаление и `gUap` – переводящую в верхний регистр целый абзац.

Грамматика Vim включает еще одно правило: когда вызывается команда-оператор два раза подряд, она воздействует на текущую строку. То есть `dd` удалит текущую строку, а `>>` добавит в нее отступ. Команда `gU` – особый случай. Мы можем заставить ее воздействовать на текущую строку, либо нажав последовательность клавиш `gUgU`, либо воспользовавшись более коротким эквивалентом `gUU`.

Таблица 2.1. Команды-операторы Vim

Команда	Выполняемая операция
`c`	Изменить
`d`	Удалить
`y`	Копировать в регистр
`g~`	Поменять регистр символов
`gu`	Преобразовать в нижний регистр
`gU`	Преобразовать в верхний регистр
`>`	Сдвинуть вправо
`<`	Сдвинуть влево
`=`	Выравнивать автоматически
`!`	Фильтровать {motion} строки внешней программой

Расширение возможностей Vim

Количество операций, которые можно выполнить с применением операторов и команд перемещения, поддерживаемых редактором Vim по умолчанию, чрезвычайно велико. Но мы можем увеличить их количество еще больше, определяя собственные операторы и команды перемещения. Давайте посмотрим, что из этого следует.

Собственные операторы получают поддержку существующих команд перемещения

Стандартный набор операторов, поддерживаемых редактором Vim, относительно невелик, но у нас имеется возможность расширять его, определяя собственные операторы. Отличным примером является расширение *commentary.vim* Тима Поупа (Tim Pope)[1].Оно добавляет команду, с помощью которой можно закомментировать или раскомментировать строки кода на любом языке программирования, поддерживаемом редактором Vim.

Команда вызывается нажатием клавиш `gc{motion}` и переключает состояние указанных строк. Это – команда-оператор, поэтому ее

---

1 https://github.com/tpope/vim-commentary

можно использовать в сочетании с любыми командами перемещения курсора. `gcap` переключит состояние текущего абзаца. `gcG` – от начала текущей строки до конца файла. `gcc` переключит состояние текущей строки.

Если вас заинтересовала возможность создания собственных операторов, начните с чтения справки `:h :map-operator` ⓘ http://vimdoc.sourceforge.net/htmldoc/map.html#:map-operator.

Собственные команды перемещения получают поддержку существующих операторов

Стандартный набор команд перемещения в Vim является достаточно полным, но мы можем еще больше расширить его новыми командами перемещения и текстовыми объектами.

Отличным примером может служить расширение *textobj-entire*, созданное Кана Натсуно (Kana Natsuno)[1]. Оно добавляет в Vim поддержку новых текстовых объектов: `ie` и `ae`, – воздействующих на весь файл.

Если нам потребуется применить автоматическое оформление отступов ко всему файлу с помощью команды `=`, мы могли бы выполнить команду `gg=G` (то есть `gg`, чтобы перейти в начало файла, и `=G`, чтобы оформить отступы от текущей позиции курсора до конца файла). Но если установить расширение *textobj-entire*, то же самое можно выполнить более простой командой `=ae`. Ей безразлична текущая позиция курсора; она всегда воздействует на файл целиком.

Обратите внимание, что если установить оба расширения, *commentary* и *textobj-entire*, мы сможем использовать их совместно. Команда `gcae` переключит состояние строк во всем файле.

Если вас заинтересовала возможность создания собственных команд перемещения, начните с чтения справки `:h omap-info` ⓘ http://vimdoc.sourceforge.net/htmldoc/map.html#omap-info.

Режим ожидающего оператора

Командный режим, режим вставки и визуальный режим обычно легко идентифицируются, но Vim поддерживает и другие режимы, которые легко не заметить. Показательным примером является режим ожидающего оператора (operator-pending mode). Мы используем его десятки раз на дню, но обычно редактор находится в этом режиме доли секунды. Например, редактор входит в этот режим, когда мы вызываем команду `dw`. Он длится ровно столько, сколько требуется, чтобы после

[1] https://github.com/kana/vim-textobj-entire

клавиши d нажать клавишу w. Короткий миг – вы даже не заметили этого!

Если рассматривать Vim как конечный автомат, режим ожидающего оператора – это состояние, в котором принимаются только команды перемещения курсора. Он активируется, когда вызывается команда-оператор, и продолжается, пока не будет введена команда перемещения, завершающая операцию. Находясь в режиме ожидающего оператора, можно вернуться в командный режим обычным способом, нажав клавишу <Esc>, прервав тем самым выполнение операции.

Многие команды вызываются двумя или более нажатиями на клавиши (например, загляните в справку `:h g` ⓘ http://vimdoc.sourceforge.net/htmldoc/vimindex.html#g, `:h z` ⓘ http://vimdoc.sourceforge.net/htmldoc/vimindex.html#z, `:h ctrl-w` ⓘ http://vimdoc.sourceforge.net/htmldoc/vimindex.html#ctrl-w или `:h [` ⓘ http://vimdoc.sourceforge.net/htmldoc/vimindex.html#[), но в большинстве случаев первая клавиша служит префиксом для второй. Эти команды не активируют режим ожидающего оператора. Вместо этого их можно интерпретировать как пространства имен, расширяющие круг доступных команд. Режим ожидающего оператора инициируется только командами-операторами.

Кто-то может задаться вопросом: «Зачем на тот краткий миг между вызовом оператора и вводом команды перемещения курсора инициируется отдельный режим, тогда как команды с префиксом пространства имен являются простыми расширениями командного режима?» Хороший вопрос! Затем, чтобы имелась возможность создавать собственные команды, инициирующие или действующие в режиме ожидающего оператора. Иными словами, такой подход позволяет создавать собственные операторы и команды перемещения и тем самым расширять словарь Vim.

Глава 3. Режим вставки

Большинство команд Vim выполняется в других режимах, но все же в режиме вставки тоже доступна некоторая функциональность. В данной главе мы исследуем эти команды. Хотя все команды удаления, копирования и вставки вызываются из командного режима, мы увидим, что имеется удобный способ, позволяющий вставлять текст из регистра, не покидая режима вставки. Мы также узнаем, что Vim поддерживает два простых способа вставки необычных символов, не имеющих соответствующих клавиш на клавиатуре.

Режим замены – частный случай режима вставки, в котором происходит затирание существующих символов. Мы узнаем, как вызывать этот режим, и рассмотрим некоторые ситуации, где он может пригодиться. Мы также познакомимся с командным подрежимом режима вставки, позволяющим выполнить одну команду и вернуться в режим вставки.

Автодополнение – одна из самых мощных особенностей, доступных в режиме вставки. Мы подробно рассмотрим ее в главе 19 «Набери X и пользуйся автодополнением».

Рецепт 13. Исправляйте ошибки, не выходя из режима вставки

Если при наборе текста в режиме вставки была допущена ошибка, ее можно тут же исправить. Для этого не нужно изменять режим. Чтобы быстро исправить ошибку в режиме вставки, помимо клавиши забоя (Backspace) можно использовать еще пару команд.

Печать вслепую – это не просто набор текста, не глядя на клавиатуру; это еще и сенсорное восприятие набираемого текста. Когда опытные наборщики совершают опечатки, они узнают о ней еще до того, как увидят ее. Они чувствуют ошибки своими пальцами, как какое-то неверное движение.

Когда мы допускаем опечатку, то можем воспользоваться клавишей забоя, чтобы стереть ее и внести исправления. Если ошибка оказывается совсем рядом с концом слова, такой способ исправления может оказаться самым быстрым. Но что, если ошибка была допущена в начале слова?

Опытные наборщики рекомендуют радикальный подход: удалить все слово, а затем ввести его снова. Если вы способны печатать со скоростью 60 слов в минуту, повторный набор слова займет всего одну секунду. Если вы не способны печатать с такой скоростью, рассматривайте этот прием как возможность попрактиковаться! Существуют некоторые слова, в которых я с завидным постоянством допускаю опечатки. Как только я начал следовать этому совету, я точнее запомнил, какие слова сбивают меня с толку. В результате теперь я делаю намного меньше ошибок.

Как вариант можно переключиться в командный режим, перейти в начало слова, исправить ошибку, нажать `A`, чтобы вернуться к месту, где вы вышли из режима вставки. На выполнение этого маленького танца уйдет значительно больше времени, чем одна секунда, и он никак не скажется на ваших навыках слепой печати. Наличие возможности переключаться между режимами не означает, что вы должны ею пользоваться.

В режиме вставки клавиша забоя действует именно так, как можно было бы ожидать: она удаляет символ перед курсором. Кроме того, в нашем распоряжении имеются следующие аккорды:

Комбинация	Действие
`<C-h>`	Удалит символ слева от курсора (забой)
`<C-w>`	Удалит слово слева от курсора
`<C-u>`	Удалит текст слева от курсора до начала строки

Эти команды не являются уникальными для режима вставки или даже для Vim. Их можно использовать в командной строке Vim, а также в командной оболочке bash.

Рецепт 14. Возвращайтесь в командный режим

Режим вставки предназначен для выполнения одной задачи – ввода текста, тогда как командный режим является режимом по умолчанию, в котором вы будете находиться большую часть време-

ни. Поэтому очень важно иметь возможность быстро переключаться между ними. Этот рецепт демонстрирует пару приемов, которые помогут ускорить переключение между режимами.

Классический способ вернуться обратно в командный режим – нажать клавишу `<Esc>`, но на многих клавиатурах руке придется проделать длинный путь, чтобы достать ее. Однако существует иной способ – комбинация `<C-[>`, которая производит точно такой же эффект (см. `:h i_CTRL-[` ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#i_CTRL-[).

Комбинация	Действие
`<Esc>`	Переход в командный режим
`<C-[>`	Переход в командный режим
`<C-o>`	Переход в командный подрежим режима вставки

Начинающие осваивать Vim часто жалуются на необходимость постоянно переключаться между режимами, но по мере накопления опыта они начинают чувствовать себя более естественно. Модальная природа Vim может показаться неудобной лишь в одном случае – когда в режиме вставки возникает потребность выполнить единственную команду командного режима и затем продолжить набор текста. В редакторе Vim имеется неплохое средство, позволяющее сгладить эту шероховатость: командный подрежим режима вставки.

Встречайте: командный подрежим режима вставки

Командный подрежим режима вставки – это особая версия командного режима, которая дает нам всего один патрон. Мы можем выстрелить единственную команду, после чего произойдет возврат в режим вставки. Переход в командный подрежим режима вставки выполняется нажатием `<C-o>` (`:h i_CTRL-O` ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#i_CTRL-O).

Когда находишься в самой верхней или в самой нижней строке, иногда бывает необходимо прокрутить окно, чтобы увидеть невидимый текст. Команда `zz` прокручивает окно так, что текущая строка оказывается в середине окна, что дает возможность увидеть текст на полэкрана выше и ниже текущей строки. Я часто пользуюсь этой командой, вызывая ее из командного подрежима режима вставки: `<C-o>zz`. После этого я тут же возвращаюсь в режим вставки и могу продолжить ввод.

Рецепт 15. Вставка из регистра, не покидая режима вставки

Операции копирования текста в регистр и вставки из регистра обычно выполняются в командном режиме, но иногда может возникнуть желание вставить текст в документ, не покидая режима вставки.

Ниже приводится незавершенный фрагмент текста:

insert_mode/practical-vim.txt
http://media.pragprog.com/titles/dnvim/code/insert_mode/practical-vim.txt

```
Practical Vim, by Drew Neil
Read Drew Neil's
```

Переназначение клавиши Caps Lock

Для пользователей Vim клавиша **Caps Lock** представляет определенную угрозу. Если нажать клавишу **Caps Lock** и попробовать использовать клавиши `k` и `j` для перемещения курсора, в результате будут выполнены команды `K` и `J`. Проще говоря: команда `K` найдет страницу справочного руководства (man) для слова под курсором (`:h K` ⓘ http://vimdoc.sourceforge.net/htmldoc/various.html#K), а команда `J` объединит текущую и следующую строки (`:h J` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#J). Это просто удивительно, как быстро можно испортить текст в буфере, случайно нажав клавишу **Caps Lock**!

Многие пользователи Vim переназначают действие клавиши **Caps Lock** на другую клавишу, например `<Esc>` или `<Ctrl>`. На современных клавиатурах клавиша `<Esc>` находится достаточно далеко, тогда как **Caps Lock** всегда под рукой. Переназначение клавиши **Caps Lock**, чтобы она вела себя как клавиша `<Esc>`, может сэкономить массу сил и времени, особенно если учесть, что клавиша `<Esc>` очень часто используется в Vim. Лично я предпочитаю назначать клавише **Caps Lock** поведение клавиши `<Ctrl>`. Комбинация `<C-[>` действует идентично клавише `<Esc>`, и ее легко вводить, когда клавиша `<Ctrl>` оказывается под рукой. Кроме того, клавишу `<Ctrl>` можно использовать и для отображений многих других функций, не только в Vim, но и в других программах.

Простейший способ переназначить клавишу **Caps Lock** – выполнить эту операцию на системном уровне. Методы могут отличаться в OS X, Linux и Windows, поэтому я не буду приводить инструкции для каждой системы, а предложу проконсультироваться у Google. Обратите внимание, что эта настройка затронет не только Vim: она отразится на всей системе. Если вы воспользуетесь моим советом, то откажитесь от клавиши **Caps Lock** навсегда. И вы никогда не будете жалеть об этом, обещаю.

Нам нужно дополнить последнюю строку, вставив название этой книги. Так как этот текст уже присутствует в начале первой строки, мы скопируем его в регистр и затем вставим в конец последней строки в режиме вставки:

Нажатия клавиш	Содержимое буфера
yt,	Practical Vim, by Drew Neil Read Drew Neil's
jA␣	Practical Vim, by Drew Neil Read Drew Neil's
<C-r>0	Practical Vim, by Drew Neil Read Drew Neil's Practical Vim
.<Esc>	Practical Vim, by Drew Neil Read Drew Neil's Practical Vim.

Команда yt, копирует слова *Practical Vim* в регистр (в рецепте 50 (глава 8) мы познакомимся с командой t{char}). Вставить текст из регистра в позицию курсора, находясь в режиме вставки, можно нажатием клавиш <C-r>0. Регистры и операции копирования мы достаточно подробно рассмотрим в главе 10 «Копирование и вставка».

В общем случае команда имеет вид <C-r>{register}, где {register} – это адрес регистра (см. :h i_CTRL-R ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#i_CTRL-R).

Используйте <C-r>{register} для доступа к регистрам

Команду <C-r>{register} удобно использовать для копирования нескольких слов в режиме вставки. Если регистр содержит достаточно большой фрагмент текста, вы можете заметить небольшую задержку перед тем, как текст на экране обновится. Это объясняется тем, что Vim вставляет текст из регистра, как если бы он вводился по одному символу. Если включена настройка textwidth или autoindent, в тексте могут появиться нежелательные разрывы строк или дополнительные отступы.

Команда <C-r><C-p>{register} действует более интеллектуально. Она вставляет текст буквально и исправляет все непредусмотренные отступы (см. :h i_CTRL-R_CTRL-P ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#i_CTRL-R_CTRL-P). Но она так утомительна! Если мне требуется вставить из регистра многострочный текст, я предпочитаю переключиться в командный режим и использовать одну из команд вставки (см. рецепт 62 в главе 10).

Рецепт 16. Выполняйте простые вычисления на месте

Регистр выражений позволяет выполнять вычисления и вставлять результат непосредственно в документ. В этом рецепте вы увидите одно из применений этой мощной возможности.

Большинство регистров в Vim хранят текст либо в виде последовательности символов, либо в виде целых строк текста. Команды удаления и копирования в регистр позволяют сохранять содержимое в регистре, а команда вставки – извлекать содержимое регистра и вставлять его в документ.

Регистр выражений – это нечто иное. Он может вычислять выражения на языке сценариев Vim и возвращать результат. Мы можем использовать его как калькулятор. Передача простого выражения, такого как `1+1`, даст результат `2`. Возвращаемое значение можно использовать как текстовое содержимое обычного регистра.

Регистр выражений адресуется символом `=`. Получить доступ к регистру выражений из режима вставки можно командой `<C-r>=`. В результате в нижней части экрана появится строка ввода, где можно ввести требуемое выражение. Закончив ввод, нажмите `<CR>`, и Vim вставит результат в документ, в позицию курсора.

Допустим, что мы только что набрали текст:

insert_mode/back-of-envelope.txt
http://media.pragprog.com/titles/dnvim/code/insert_mode/back-of-envelope.txt

```
6 chairs, each costing $35, totals $
```

Не нужно вычислять общую стоимость на клочке бумаги. Vim может выполнить вычисления за нас, и нам даже не придется покидать режим вставки:

Нажатия клавиш	Содержимое буфера
`A`	`6 chairs, each costing $35, totals $`
`<C-r>=6*35<CR>`	`6 chairs, each costing $35, totals $210`

Регистр может выполнять не только простые арифметические операции. Более сложный пример использования этого регистра будет представлен в рецепте 71 в главе 11.

Рецепт 17. Вставка необычных символов по их кодам

Редактор Vim способен вставлять любые символы по их числовым кодам. Эта особенность может пригодиться для ввода символов, отсутствующих на клавиатуре.

Мы можем попросить Vim вставить любой символ, если знаем его числовой код. В режиме вставки достаточно просто ввести <C-v>{code}, где {code} – числовой код символа, который требуется вставить.

Vim ожидает получить код, состоящий из трех цифр. Допустим, например, что нам требуется вставить символ «A» из латинского алфавита. Этому символу соответствует код 65, поэтому мы могли бы ввести его как <C-v>065.

Но что, если код вставляемого символа содержит больше трех цифр? Например, базовая таблица символов Юникода содержит 65 535 символов. Как оказывается, любой символ из этой таблицы можно ввести как код из четырех шестнадцатеричных цифр: <C-v>u{1234} (обратите внимание на клавишу u перед цифровым кодом). Представим, что нам нужно вставить символ перевернутого знака вопроса («¿»), который имеет код 00bf. Для этого достаточно в режиме вставки нажать последовательность <C-v>u00bf. Подробности ищите в справке :h i_CTRL-V_digit ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#i_CTRL-V_digit.

Если необходимо определить числовой код для некоторого символа в документе, просто поместите курсор на него и выполните команду ga. В результате в нижней части экрана будет выведено сообщение с кодом символа в десятичной и шестнадцатеричной записи (см. :h ga ⓘ http://vimdoc.sourceforge.net/htmldoc/various.html#ga). Разумеется, это не поможет узнать код символа, отсутствующего в документе. В этом случае вы можете заглянуть в таблицы символов Юникода.

Если за <C-v> последует нажатие любой нецифровой клавиши, в документ будет вставлен символ, соответствующий этой клавише. Например, если включена настройка expandtab, нажатие клавиши <Tab> будет приводить к вставке пробелов вместо символа табуляции. Но нажатие <C-v><Tab> всегда будет приводить к вставке самого символа табуляции, независимо от значения настройки expandtab.

В табл. 3.1 перечислены команды ввода необычных символов.

Таблица 3.1. Команды вставки необычных символов

Комбинация	Действие
<C-v>{123}	Вставит символ, соответствующий десятичному коду
<C-v>u{1234}	Вставит символ, соответствующий шестнадцатеричному коду
<C-v>{nondigit}	Вставит сам нецифровой символ {nondigit}
<C-k>{char1}{char2}	Вставит символ, соответствующий паре символов {char1}{char2}, или диграфу

Рецепт 18. Вставка необычных символов, соответствующих парам символов

Хотя редактор Vim позволяет вставлять любые символы по их числовым кодам, иногда бывает сложно запомнить эти коды. Необычные символы можно также вставлять как пары символов (диграфы), которые проще для запоминания.

Описываемый здесь способ достаточно прост в использовании. Нужно лишь ввести в режиме вставки последовательность <C-k>{char1}{char2}. То есть если потребуется вставить символ «¿», который может быть представлен парой символов «?I», мы могли бы просто ввести <C-k>?I.

Пары букв, образующие диграфы, подобраны по определенным правилам, чтобы их проще было запомнить или даже угадать. Например, кавычкам-елочкам – « и » – соответствуют диграфы << и >>; рациональные дроби – ½, ¼ и ¾ – можно представить диграфами 12, 14 и 34 соответственно. Диграфы, поддерживаемые редактором Vim по умолчанию, подчиняются определенным соглашениям, с которыми можно ознакомиться в справке :h digraphs-default ⓘ http://vimdoc.sourceforge.net/htmldoc/digraph.html#digraphs-default.

Список всех имеющихся диграфов можно увидеть, выполнив команду :digraphs, но этот список сложен для восприятия. Более удобочитаемый список можно найти в справке :h digraph-table ⓘ http://vimdoc.sourceforge.net/htmldoc/digraph.html#digraph-table.

Рецепт 19. Правка текста в режиме замены

Режим замены идентичен режиму вставки, за исключением того, что в этом режиме вводимые символы затирают имеющиеся.

Допустим, что у нас имеется такой фрагмент текста:

insert_mode/replace.txt
http://media.pragprog.com/titles/dnvim/code/insert_mode/replace.txt

```
Typing in Insert mode extends the line. But in Replace mode
the line length doesn't change.
```

Нам необходимо объединить два предложения в одно, заменив точку запятой. Кроме того, требуется символ «B» в слове «But» заменить символом «b». Следующий пример демонстрирует, как можно выполнить эту правку в режиме замены.

Нажатия клавиш	Содержимое буфера
`{start}`	`Typing in Insert mode extends the line. But in Replace mode the line length doesn't change.`
`f.`	`Typing in Insert mode extends the line. But in Replace mode the line length doesn't change.`
`R,␣b<Esc>`	`Typing in Insert mode extends the line, but in Replace mode the line length doesn't change.`

Переход в режим замены из командного режима выполняется командой `R`. Как видно в примере выше, вводимые символы «, b» затирают имеющиеся «. B». Закончив правку в режиме замены, можно нажать клавишу `<Esc>` и вернуться в командный режим. Не на всех клавиатурах имеется клавиша `<Insert>`, но если у вас она есть, вы можете использовать ее для переключения между режимами вставки и замены.

Затирайте символы табуляции в виртуальном режиме замены

Некоторые символы могут осложнять работу в режиме замены. Примером таких символов могут служить символы табуляции. В файле символ табуляции представлен единственным символом, но на экране он может занимать несколько знакомест, согласно настройке `tabstop` (см. `:h 'tabstop'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'tabstop'). Если поместить курсор в позицию символа табуляции и инициировать режим замены, следующий введенный символ затрет символ табуляции. Допустим, что параметр настройки `tabstop` имеет значение 8 (по умолчанию), тогда может создаться ощущение, что один символ заменил восемь пробелов, вызвав значительное сокращение длины текущей строки.

В Vim имеется еще один вариант режима замены – *виртуальный режим замены*, который инициируется командой `gR`. В этом режиме символы табуляции интерпретируются как последовательности пробелов. Допустим, что курсор находится в позиции символа табуляции, занимающего восемь знакомест на экране. Переключившись в виртуальный режим замены, мы сможем ввести до семи символов, каждый из которых будет вставлен перед символом табуляции. Если затем ввести восьмой символ, он заменит символ табуляции.

В виртуальном режиме замены происходит затирание знакомест на экране, а не фактических символов, которые могут храниться в файле. Так как в результате происходит меньше неожиданностей, я рекомендую использовать виртуальный режим замены всегда, когда это уместно.

Редактор Vim поддерживает также однократную версию режима замены и виртуального режима замены. Команды `r{char}` и `gr{char}` дают возможность затереть единственный символ и переключиться обратно в командный режим (`:h r` ℹ http://vimdoc.sourceforge.net/htmldoc/change.html#r).

Глава 4. Визуальный режим

Визуальный режим редактора Vim позволяет выделять произвольные фрагменты текста и применять к ним какие-либо операции. Работа в этом режиме проста и понятна, потому что в его основе лежит модель, которой следует большинство программ-редакторов. Но Vim привносит в эту модель свои характерные особенности, поэтому для начала мы познакомимся с визуальным режимом (рецепт 20).

Редактор Vim поддерживает три разновидности визуального режима, позволяющие оперировать символами, строками и прямоугольными блоками текста. Мы исследуем способы переключения между этими режимами, а также с некоторыми интересными приемами изменения границ выделенного текста (рецепт 21).

Далее вы узнаете, что команда «точка» может использоваться также для повторения команд визуального режима и что она наиболее эффективна при использовании в построчном визуальном режиме (или в режиме визуальной строки). При работе с посимвольными областями выделения команда «точка» иногда может не оправдывать наши ожидания. В таких ситуациях команды-операторы могут оказаться более предпочтительными.

Блочный визуальный режим (или режим визуального блока) отличается возможностью выполнять операции с прямоугольными блоками текста. Вы сможете найти самые разные варианты использования этой особенности, но в этой книге мы рассмотрим лишь три рецепта, демонстрирующие некоторые ее возможности.

Рецепт 20. Знакомство с визуальным режимом

Визуальный режим позволяет выделить фрагмент текста и применить к нему какие-либо операции. Однако в своем подходе к выделению текста Vim отличается от других редакторов.

Представим на минуту, что мы работаем не с Vim, а заполняем текстовую область в веб-странице. Мы хотели написать слово «April», но вместо него написали «March», поэтому мы дважды щелкаем левой кнопкой мыши на слове, чтобы выделить его. Выделив слово, мы можем нажать клавишу забоя, чтобы удалить его, и затем ввести правильное название месяца.

Возможно, вы уже знаете, что в данном случае нет необходимости нажимать клавишу забоя. После выделения слова «March» достаточно просто ввести букву «A», и она заменит выделенное слово, расчистив путь для ввода остальных символов в слове «April». Вроде бы ничего особенного, но одно нажатие клавиши экономится.

Если вы ожидаете подобного поведения от визуального режима в редакторе Vim, то будете удивлены. Ключ находится в самом названии: визуальный *режим* – это просто иной режим, в котором клавиши выполняют другие функции.

Многие команды, знакомые вам по командному режиму, действую точно так же и в визуальном режиме. Для перемещения курсора можно использовать клавиши `h`, `j`, `k` и `l`. Для перехода к определенному символу в строке можно использовать команду `f{char}`, а затем повторять поиск вперед или назад, выполняя команды `;` и `,` соответственно. Мы можем даже использовать команду поиска (и команды `n`/`N`) для перемещения между совпадениями. Всякий раз, перемещая курсор в визуальном режиме, мы изменяем границы выделения.

Некоторые команды визуального режима выполняют те же функции, что и в командном режиме, но с некоторыми побочными эффектами. Например, команда `c` действует одинаково в обоих режимах, удаляя указанный текст и затем переключаясь в режим вставки. Разница состоит в том, как указывается область действия команды. В командном режиме сначала вводится команда изменения текста, а затем определяется область ее действия. Такие команды, если вспомнить рецепт 12 в главе 2, называются командами-операторами. В визуальном режиме, напротив, сначала выделяется фрагмент текста, а затем вызывается команда его изменения. Такая инверсия управления распространяется на все команды-операторы (см. табл. 2.1 в главе 2). Для многих работа в визуальном режиме кажется более простой и понятной.

Вернемся к примеру, где требуется заменить слово «March» словом «April». На этот раз оставим текстовую область в веб-странице и вернемся в уютный редактор Vim. Мы поместим курсор где-то

в пределах слова «March» и выполним команду `viw`, чтобы визуально выделить слово. Теперь мы не можем просто взять и ввести слово «April», потому что первый символ будет воспринят как команда `A`, а остальные просто будут добавлены в текст! Вместо этого, чтобы изменить выделенный текст, следует выполнить команду `c`, которая удалит выделенное слово и перенесет нас в режим вставки, где мы сможем ввести слово «April». Этот шаблон напоминает первый пример с текстовой областью в веб-странице, только вместо клавиши забоя здесь используется клавиша `c`.

Встречайте: режим выделения

В типичных текстовых редакторах выделенный текст удаляется при вводе любого печатаемого символа. Визуальный режим редактора Vim не следует этому соглашению, а вот режим выделения – следует. Как описывается во встроенной документации Vim, он «напоминает режим выделения в Microsoft Windows» (см. `:h Select-mode` ⓘ http://vimdoc.sourceforge.net/htmldoc/visual.html#Select-mode). Ввод печатаемого символа в этом режиме вызывает удаление выделенного текста, перевод редактора Vim в режим вставки и добавление введенного символа.

Переключаться между визуальным режимом и режимом выделения можно нажатием клавиш `<C-g>`. Единственное видимое отличие между этими режимами – сообщение внизу окна, обозначающее режим: `-- VISUAL --` (`-- ВИЗУАЛЬНЫЙ РЕЖИМ --`) или `-- SELECT --` (`-- ВЫДЕЛЕНИЕ --`). Но если в режиме выделения ввести любой печатаемый символ, он заменит выделенный текст и переключит редактор в режим вставки. Конечно, точно так же можно было бы нажать клавишу `c` в визуальном режиме, чтобы перейти к изменению выделенного текста.

Когда вы освоитесь с модальной природой Vim, вы почти не будете пользоваться режимом выделения, который поддерживает под руки пользователей, которым хотелось бы, чтобы Vim действовал подобно многим другим текстовым редакторам. Лично я могу вспомнить только одну ситуацию, где я постоянно использую режим выделения: при работе с расширением, имитирующим функциональность редактора TextMate, режим выделения подсвечивает активную метку-заполнитель.

Рецепт 21. Выделение текста в визуальном режиме

Визуальный режим имеет три подрежима для работы с разными типами текста. В этом рецепте мы познакомимся со способами пере-

ключения между ними.

Редактор Vim поддерживает три разновидности визуального режима. Посимвольный визуальный режим позволяет выделять любые фрагменты текста, от одного символа до их группы, которая может включать несколько строк. Этот подрежим можно использовать для выполнения операций с отдельными словами или фразами. Для работы с целыми строками можно использовать построчный визуальный режим (или режим визуальной строки). Наконец, визуальный режим работы с блоками позволяет выполнять операции с прямоугольными областями текста в документе. Блочный визуальный режим (или режим визуального блока) имеет массу специфических особенностей, поэтому мы рассмотрим его более подробно в рецептах 24, 25 и 26.

Включение визуальных режимов

Путь в визуальный режим лежит через клавишу `v`. При нажатии клавиши `v` в командном режиме Vim переходит в посимвольный визуальный режим. Переход в построчный визуальный режим осуществляется нажатием `V` (с клавишей **Shift**), а переход в блочный визуальный режим – нажатием `<C-v>` (с клавишей **Control**). Эти команды перечислены в таблице ниже:

Команда	Действие
`v`	Активирует посимвольный визуальный режим
`V`	Активирует построчный визуальный режим
`<C-v>`	Активирует блочный визуальный режим
`gv`	Повторно выделяет последнее визуальное выделение

Команда `gv` – это удобное клавиатурное сокращение. Она повторно выделяет фрагмент текста, который был выделен последним в визуальном режиме. Не важно, в каком из подрежимов было сделано это выделение – в посимвольном, построчном или блочном, – команда `gv` сделает все правильно. Единственная ситуация, когда может возникнуть путаница, – если прежде выделенный текст был удален.

Переключение между визуальными режимами

Переключение между разновидностями визуального режима производится точно так же, как их активация из командного режима. Переход из посимвольного визуального режима в построчный вы-

полняется нажатием клавиши V, переход в блочный визуальный режим – нажатием <C-v>. Но если в посимвольном визуальном режиме нажать v, произойдет возврат в командный режим. Поэтому клавишу v можно считать переключателем между командным и посимвольным визуальным режимом. Клавиши V и <C-v> также действуют как переключатели между командным режимом и соответствующими разновидностями визуального режима. Разумеется, вы всегда можете явно перейти в командный режим, нажав <Esc> или <C-[> (по аналогии с выходом из режима вставки). Команды переключения между визуальными режимами перечислены в следующей таблице:

Команда	Действие
<Esc> / <C-[>	Выполняет переход в командный режим
v / V / <C-v>	Выполняет переход в командный режим (когда нажимается в посимвольном, построчном или блочном визуальном режиме соответственно)
v	Выполняет переход в посимвольный визуальный режим
V	Выполняет переход в построчный визуальный режим
<C-v>	Выполняет переход в блочный визуальный режим
o	Перемещает курсор в конец выделенного фрагмента текста

Переключение свободного конца выделения

Выделение в визуальном режиме имеет два конца: *фиксированный* и *свободный* (перемещаемый вместе с курсором). Клавиша o позволяет переключать свободный конец. Это очень удобно, если на полпути выделения фрагмента обнаруживается, что он начинается не с того места. Вместо выхода из визуального режима и попытки начать все сначала можно просто нажать клавишу o и изменить границы области выделения. Ниже показано, как можно использовать этот прием:

Нажатия клавиш	Содержимое буфера
{start}	Select from here to here.
vbb	Select from here to here.
o	Select from here to here.
e	Select from here to here.

Рецепт 22. Повторение команд построчного визуального режима

При использовании команды «точка» для повторения изменений в визуальном режиме она применяет эти изменения к фрагменту

текста того же размера. В этом рецепте мы выполним изменение выделенного текста в построчном визуальном режиме и затем повторим это изменение с помощью команды «точка».

После выполнения команды в визуальном режиме редактор автоматически возвращается в командный режим, а выделение, сделанное в визуальном режиме, снимается. А как быть, если нам потребуется применить другую команду визуального режима к тому же фрагменту текста? Допустим, что у нас имеется следующий код на языке Python, содержащий ошибку форматирования:

visual_mode/fibonacci-alformed.py
http://media.pragprog.com/titles/dnvim/code/visual_mode/fibonacci-malformed.py

```
def fib(n):
    a, b = 0, 1
    while a < n:
print a,
a, b = b, a+b
fib(42)
```

В этом фрагменте кода величина отступа составляет четыре пробела. Сначала нам следует настроить Vim на использование этого стиля.

Подготовка

Чтобы обеспечить правильную работу команд `<` и `>`, необходимо присвоить параметрам настройки `shiftwidth` и `softtabstop` значение 4 и включить параметр `expandtab`. Если вам интересно, как действуют эти настройки, прочитайте статью «Tabs and Spaces» на сайте Vimcasts.org[1]. Выполнить необходимые настройки можно одной командой:

```
➾ :set shiftwidth=4 softtabstop=4 expandtab
```

Выполните отступ один раз, а затем повторите

В представленном выше фрагменте кода на языке Python две строки, следующие за ключевым словом `while`, должны иметь два отступа. Мы могли бы попробовать исправить ошибку, выделив эти

[1] http://vimcasts.org/e/2

две строки в визуальном режиме и выполнив команду `>`. Но в этом случае будет добавлен только один отступ, после чего произойдет возврат в командный режим.

После этого можно было бы повторно выделить тот же текст командой `gv` и вызвать команду добавления отступа еще раз. Но если вы начали интуитивно понимать, как действует редактор Vim, в ответ на необходимость подобных манипуляций в вашей голове должен прозвенеть звонок.

Когда мне нужно что-то повторить, я часто использую команду «точка». Вместо того чтобы снова выделять тот же фрагмент текста и повторять ту же команду вручную, можно просто нажать клавишу `.` в командном режиме. Вот как это выглядит на практике:

Нажатия клавиш	Содержимое буфера

{start}	def fib(n):
	    a, b = 0, 1
	    while a < n:
	print a,
	a, b = b, a+b
	fib(42)

`Vj`	def fib(n):
	    a, b = 0, 1
	    while a < n:
	print a,
	a, b = b, a+b
	fib(42)

`>.`	def fib(n):
	    a, b = 0, 1
	    while a < n:
	        print a,
	        a, b = b, a+b
	fib(42)

Если вам удобно пользоваться счетчиками, вы можете предпочесть достигнуть поставленной цели одним ударом, выполнив команду `2>` в визуальном режиме. Лично я предпочитаю пользоваться командой «точка», потому что она дает мне чувство постоянной обратной связи. Если мне потребуется повторить добавление отступа, я просто нажму `.` еще раз. Или если я впопыхах нажму клавишу лишний раз, я просто нажму клавишу `u`. Более подробно разница между этими подходами обсуждается в рецепте 11 (глава 2).

Когда команда «точка» используется для повторения команд визуального режима, она воздействует на тот же объем текста, который

был помечен последней операцией выделения. Такое поведение может приносить пользу, когда текст выделяется в построчном визуальном режиме, но оно же может давать неожиданные результаты при выделении в посимвольном визуальном режиме. Рассмотрим пример, иллюстрирующий это.

Рецепт 23. Используйте операторы вместо команд визуального режима, если это возможно

Возможно, работа в визуальном режиме более понятна, чем в командном режиме, но в ней есть свои недостатки: команда «точка» не всегда действует непротиворечиво с визуальным режимом. Этот недостаток часто можно обойти с помощью операторов командного режима.

Допустим, что нам требуется изменить следующий список гиперссылок, преобразовав смволы в верхний регистр:

visual_mode/list-of-links.html
http://media.pragprog.com/titles/dnvim/code/visual_mode/list-of-links.html

```
<a href="#">one</a>
<a href="#">two</a>
<a href="#">three</a>
```

Мы можем выделить внутреннее содержимое тега командой `vit`, которую можно интерпретировать как *visually* select *inside* the *tag* (визуально выделить содержимое тега). Эта команда представляет собой особую разновидность области действия, называемую *текстовым объектом* (text object). Подробнее о текстовых объектах рассказывается в рецепте 52 в главе 8.

Использование визуального оператора

В визуальном режиме можно выделить текст и выполнить операцию над ним. В данном случае нам нужно выполнить команду `U`, которая преобразует выделенные символы в верхний регистр (`:h v_U` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#v_U). См. табл. 4.1 ниже.

Выполнив преобразование в первой строке, нам нужно выполнить то же самое преобразование в следующих двух строках. А не воспользоваться ли нам формулой «точки»?

Таблица 4.1. Преобразование в верхний регистр в визуальном режиме

Нажатия клавиш	Содержимое буфера

{start}	`<a href="#">one</a>`
	`<a href="#">two</a>`
	`<a href="#">three</a>`

vit	`<a href="#">one</a>`
	`<a href="#">two</a>`
	`<a href="#">three</a>`

U	`<a href="#">ONE</a>`
	`<a href="#">two</a>`
	`<a href="#">three</a>`

Команда `j.` переместит курсор на следующую строку и повторит последнее изменение. Со второй строкой никаких проблем не возникает, но если повторить эту команду, мы увидим странный результат:

```
<a href="#">ONE</a>
<a href="#">TWO</a>
<a href="#">THRee</a>
```

Вам понятно, что случилось? Когда повторяется команда визуального режима, она применяется к фрагменту текста той же длины (см. `:h visual-repeat` ⓘ http://vimdoc.sourceforge.net/htmldoc/visual.html#visual-repeat). В данном случае первый раз команда применялась к слову длиной в три символа. При повторном применении этой команды ко второй строке не произошло ничего необычного, потому что требуемое слово также содержит три символа, но при попытке применить ту же команду к слову из пяти символов мы потерпели фиаско.

Использование операторов командного режима

Для команды `U` визуального режима в командном режиме имеется эквивалентная команда: `gU{motion}` (`:h gU` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#gU). Если для первого изменения использовать эту команду, мы сможем выполнить последующие изменения, применив «формулу точки», как показано в табл. 4.2.

Обсуждение

В обоих случаях требуется нажать всего четыре клавиши: `vitU` и `gUit`, но семантика этих команд совершенно отлична. В визуаль-

ном режиме эти четыре нажатия интерпретируются как две разные команды: `vit` выделяет текст, а `U` преобразует его. Последовательность нажатий `gUit`, напротив, можно считать единственной командой, состоящей из оператора (`gU`) и области его действия (`it`).

Если вы хотите подготовить команду «точка», чтобы она выполняла некоторое полезное действие, тогда лучше покинуть визуальный режим. Как правило, всегда, когда дело касается повторения изменений, следует отдавать предпочтение командам-операторам командного режима перед их эквивалентами визуального режима.

Таблица 4.2. Оператор командного режима

Нажатия клавиш	Содержимое буфера
{start}	one two three
gUit	ONE two three
j.	ONE TWO three
j.	ONE TWO THREE

Нельзя сказать, что визуальный режим не имеет практического применения. Он с успехом может использоваться в разных ситуациях. В процессе правки текста не всегда возникает необходимость повторять изменения, поэтому визуальный режим вполне подходит для выполнения однократных изменений. И даже при том, что области действия Vim обеспечивают хирургическую точность, иногда бывает необходимо править фрагменты текста, границы которых сложно определить. В таких ситуациях визуальный режим – верный выбор.

Рецепт 24. Правка табличных данных в блочном визуальном режиме

Со строками текста можно работать в любом редакторе, но для выполнения операций над колонками текста требуется более специализированный инструмент. Vim поддерживает такую возможность в виде блочного визуального режима, который мы будем использовать для преобразования простых текстовых таблиц.

Допустим, что у нас имеется простая текстовая таблица, как показано ниже:

visual_mode/chapter-table.txt
http://media.pragprog.com/titles/dnvim/code/visual_mode/chapter-table.txt

```
Chapter                 Page
Normal  mode              15
Insert  mode              31
Visual  mode              44
```

Нам требуется нарисовать вертикальную линию, чтобы разделить колонки и сделать таблицу больше похожей на настоящую таблицу. Но сначала нам нужно уменьшить количество пробелов между колонками, которые сейчас отстоят друг от друга слишком далеко. Оба эти изменения можно выполнить в блочном визуальном режиме, как показано в табл. 4.3.

Таблица 4.3. Добавление вертикальной линии между колонками

Нажатия клавиш	Содержимое буфера

`{start}`	`Chapter          Page`
	`Normal mode        15`
	`Insert mode        31`
	`Visual mode        44`

`<C-v>3j`	`Chapter          Page`
	`Normal mode        15`
	`Insert mode        31`
	`Visual mode        44`

`x...`	`Chapter      Page`
	`Normal mode     15`
	`Insert mode     31`
	`Visual mode     44`

`gv`	`Chapter      Page`
	`Normal mode     15`
	`Insert mode     31`
	`Visual mode     44`

`r|`	`Chapter     |   Page`
	`Normal mode |     15`
	`Insert mode |     31`
	`Visual mode |     44`

`yyp`	`Chapter     |   Page`
	`Chapter     |   Page`
	`Normal mode |     15`
	`Insert mode |     31`
	`Visual mode |     44`

Таблица 4.3. Добавление вертикальной линии между колонками (окончание)

Нажатия клавиш	Содержимое буфера

`Vr-`	<pre>Chapter | Page
	------------------ 	
	Normal mode | 15
	Insert mode | 31
	Visual mode | 44</pre>

Сначала активируем блочный визуальный режим командой `<C-v>`; затем выделим колонку, переместив курсор на несколько строк вниз. Нажмем клавишу `x`, чтобы удалить колонку, и повторим удаление несколько раз командой «точка». Удаление повторяется, пока колонки не окажутся достаточно близко друг к другу.

Вместо использования команды «точка» можно было бы раздвинуть границы выделения так, чтобы выделенным оказался прямоугольный фрагмент текста, переместив курсор на две-три позиции вправо. После этого достаточно было бы выполнить одну команду удаления. Лично я предпочитаю иметь постоянную обратную связь, поэтому я выполнил удаление единственной колонки и повторил его столько раз, сколько посчитал нужным.

Теперь, когда мы расположили колонки на нужном расстоянии друг от друга, можно приступать к рисованию вертикальной линии между ними. Для этого можно повторно выделить тот же фрагмент в тексте с помощью команды `gv` и затем нажать `r|`, чтобы заменить каждый выделенный символ символом вертикальной черты.

Пока мы не ушли далеко, можно также нарисовать горизонтальную линию, отделяющую заголовок таблицы. Для этого мы копируем первую строку целиком командой `yyp` и затем замещаем все имеющиеся в ней символы дефисом (`Vr-`).

Рецепт 25. Изменение колонок текста

Блочный визуальный режим можно использовать для вставки текста в несколько строк одновременно. Блочный визуальный режим можно с успехом использовать не только для работы с табличными данными.

Очень часто из этого режима можно извлечь пользу, работая с программным кодом. Например, возьмем следующий фрагмент CSS:

visual_mode/sprite.css
http://media.pragprog.com/titles/dnvim/code/visual_mode/sprite.css

```
li.one    a{ background-image: url('/images/sprite.png'); }
li.two    a{ background-image: url('/images/sprite.png'); }
li.three  a{ background-image: url('/images/sprite.png'); }
```

Допустим, что файл *sprite.png* был перемещен из каталога *images/* в каталог *components/*. Нам необходимо изменить все ссылки на этот файл, указав новое его местоположение. Мы могли бы сделать это в блочном визуальном режиме, как показано в табл. 4.4.

Эта процедура не должна вызвать осложнений. Сначала следует выделить требуемый фрагмент текста в блочном визуальном режиме, который как раз имеет прямоугольную форму. Затем нажать клавишу `c`, в результате чего выделенный текст будет удален и редактор перейдет в режим вставки.

При вводе слова «components» в режиме вставки оно будет появляться только в верхней строке. В остальных строках ничего происходить не будет. Мы увидим введенный текст в этих строках, только когда нажмем клавишу `<Esc>`, чтобы вернуться в командный режим.

Поведение команды изменения текста в блочном визуальном режиме может показаться странным. Кажется непоследовательным, что удаление воздействует на все выделенные строки сразу, а вставка – только на самую верхнюю строку (по крайней мере, пока редактор находится в режиме вставки). Некоторые текстовые редакторы, поддерживающие похожую функциональность, изменяют сразу все выбранные строки. Если вы привыкли к такому поведению (как и я в былые времена), вам может показаться, что реализация Vim недостаточно хорошо продумана.

Таблица 4.4. Вставка в несколько строк

Нажатия клавиш	Содержимое буфера

`{start}`
Командный режим
	`li.one   a{ background-image: url('/images/sprite.png'); }`
	`li.two   a{ background-image: url('/images/sprite.png'); }`
	`li.three a{ background-image: url('/images/sprite.png'); }`

`<C-v>jje`
Визуальный режим
	`li.one   a{ background-image: url('/images/sprite.png'); }`
	`li.two   a{ background-image: url('/images/sprite.png'); }`
	`li.three a{ background-image: url('/images/sprite.png'); }`

`c`
Режим вставки
	`li.one   a{ background-image: url('//sprite.png'); }`
	`li.two   a{ background-image: url('//sprite.png'); }`
	`li.three a{ background-image: url('//sprite.png'); }`

Таблица 4.4. Вставка в несколько строк (окончание)

Нажатия клавиш	Содержимое буфера

`components`
 	
Режим вставки
	`li.one   a{ background-image: url('/ components/sprite.png'); }`
	`li.two   a{ background-image: url('//sprite.png'); }`
	`li.three a{ background-image: url('//sprite.png'); }`

`<Esc>`
Командный режим
	`li.one   a{ background-image: url('/components/sprite.png'); }`
	`li.two   a{ background-image: url('/components/sprite.png'); }`
	`li.three a{ background-image: url('/components/sprite.png'); }`

Но на конечном результате это никак не отражается. Пока вы используете режим вставки только для ввода коротких фрагментов, вы не будете замечать подобных странностей.

Рецепт 26. Добавление текста после непрямоугольного блока

Блочный визуальный режим отлично подходит для выполнения операций с прямоугольными фрагментами текста, такими как строки и колонки, но позволяет работать и с непрямоугольными фрагментами.

Мы уже встречались со следующим фрагментом кода на JavaScript:

the_vim_way/2_foo_bar.js
http://media.pragprog.com/titles/dnvim/code/the_vim_way/2_foo_bar.js

```
var foo = 1
var bar = 'a'
var foobar = foo + bar
```

Фрагмент состоит из трех строк разной длины. Нам требуется добавить точку с запятой в конец каждой из них. В рецепте 2 (глава 1) мы решили эту задачу с помощью команды «точка», но ее также можно решить с использованием блочного визуального режима, как показано в табл. 4.5.

После перехода в блочный визуальный режим мы расширяем область выделения до конца каждой строки, нажимая `$`. На первый взгляд это может показаться непросто, потому что все строки имеют разную длину. Но в данном случае Vim понимает, что мы желаем расширить область выделения до конца каждой строки. Это позволяет выделить непрямоугольную область со ступенчатым правым краем.

Таблица 4.5. Добавление точки с запятой в конец нескольких строк в блочном визуальном режиме

Нажатия клавиш	Содержимое буфера
{start} *Командный режим*	var foo = 1 var bar = 'a' var foobar = foo + bar
<C-v>jj$ *Визуальный режим*	var foo = 1 var bar = 'a' var foobar = foo + bar
A; *Режим вставки*	var foo = 1; var bar = 'a' var foobar = foo + bar

<Esc>
 	
Командный режим
	var foo = 1;
	var bar = 'a';
	var foobar = foo + bar;

Выделив фрагмент, мы можем добавить текст в конец каждой выделенной строки, воспользовавшись командой A (см. врезку «Соглашения для клавиш 'i' и 'a'» ниже). Она переводит редактор в режим вставки в самой верхней строке выделенного фрагмента. Все, что будет вводиться после этого (до выхода из режима вставки), будет появляться только в этой строке, но сразу после перехода в командный режим все изменения будут автоматически распространены на все остальные строки, попавшие в выделенную область.

Соглашения для клавиш 'i' и 'a'

В Vim имеется пара соглашений, касающихся переключения из командного режима в режим вставки. Команды i и a обе выполняют такое переключение, устанавливая курсор перед и после текущего символа соответственно. Команды I и A действуют аналогично, кроме того что устанавливают курсор в начало или в конец текущей *строки*.

Аналогичные соглашения поддерживаются также для организации перехода из блочного визуального режима в режим вставки. Команды I и A обе выполняют такое переключение, устанавливая курсор перед и после *выделения* соответственно. А как же команды i и a; как они действуют в визуальном режиме?

В визуальном режиме и в режиме ожидающего оператора клавиши i и a следуют иному соглашению: они образуют первую половину *текстового объекта*. Подробнее о текстовых объектах рассказывается в рецепте 52 (глава 8). Если вы, выделив текст в блочном визуальном режиме, не можете перейти в режим вставки клавишей i, попробуйте нажать I.

Глава 5. Режим командной строки

Редактор Vim ведет свою родословную от редактора vi, в котором была реализована модальная парадигма редактирования. Редактор vi, в свою очередь, ведет родословную от строчного редактора с названием ex, от которого по наследству достались команды Ex. ДНК этих ранних текстовых редакторов для Unix сохранились и в современном Vim. Для решения некоторых задач команды Ex все еще остаются лучшим инструментом. В этой главе вы узнаете, как пользоваться режимом командной строки, который возвращает нас к истокам – к редактору ex.

Рецепт 27. Встречайте: режим командной строки

Режим командной строки позволяет вводить команды редактора Ex, шаблоны поиска или выражения. В данном рецепте мы познакомимся с командами выбора, воздействующими на текст в буфере, и с некоторыми специальными комбинациями клавиш для использования в этом режиме.

Когда мы нажимаем клавишу `:`, Vim переключается в режим командной строки. Этот режим имеет некоторое сходство с командной строкой оболочки. В этом режиме можно вводить команды и запускать их нажатием клавиши `<CR>`. Нажав клавишу `<Esc>`, можно в любой момент покинуть режим командной строки и вернуться в командный режим.

Традиционно команды, выполняемые в режиме командной строки, называют *командами редактора Ex* (см. врезку «О происхождении редактора Vim (и его семейства)» ниже). Режим командной строки можно также активировать нажатием клавиши `/`, после чего

появится приглашение к вводу шаблона для поиска или <C-r>= – для доступа к регистру выражений (см. рецепт 16 в главе 3). Некоторые советы из этой главы в равной степени применимы ко всем вариантам командной строки, но большая часть советов будет касаться только режима командной строки для ввода команд Ex.

Таблица 5.1. Команды Ex для выполнения операций с текстом в буфере

Команда	Действие
`:[range]delete[x]`	Удалит указанные строки [в регистр x]
`:[range]yank[x]`	Скопирует указанные строки [в регистр x]
`:[line]put[x]`	Вставит текст из регистра x после указанной строки
`:[range]copy{address}`	Скопирует указанные строки ниже строки с номером {address}
`:[range]move{address}`	Переместит указанные строки ниже строки с номером {address}
`:[range]join`	Объединит указанные строки
`:[range]normal{commands}`	Выполнит команду {commands} командного режима для каждой указанной строки
`:[range]substitute/{pattern}/{string}/[flags]`	Заменит вхождения {pattern} последовательностью символов {string} в каждой указанной строке
`:[range]global/{pattern}/[cmd]`	Выполнит команду [cmd] для всех строк из числа указанных, где найдено вхождение {pattern}

Команды Ex можно использовать для чтения и сохранения файлов (`:edit` и `:write`), для создания новых вкладок (`:tabnew`) или разделения окна (`:split`), а также для выполнения операций со списком аргументов (`:prev`/`:next`) или списком буферов (`:bprev`/`:bnext`). Фактически с помощью команд Ex можно выполнять любые действия (полный список можно найти в справке `:h ex-cmd-index` ⓘ http://vimdoc.sourceforge.net/htmldoc/index.html#ex-cmd-index).

В этой главе мы сосредоточимся в основном на командах Ex, используемых для редактирования текста. В табл. 5.1 перечислены наиболее часто употребимые из них.

Большинство из этих команд предваряются префиксом, определяющим диапазон их действия (`[range]`). Подробнее о диапазонах будет рассказываться в рецепте 28 ниже. Команда `:copy` позволяет быстро скопировать строку (см. также раздел «Копирование строк командой ':t'» в рецепте 29). Команда `:normal` дает удобную возможность применить одно и то же изменение к группе строк, как будет показано в рецепте 30.

Мы также познакомимся с командами `:delete`, `:yank` и `:put` в главе 10 «Копирование и вставка». Команды `:substitute` и `:global` обладают настолько широкими возможностями, что каждой из них посвящена отдельная глава (глава 14 «Подстановка» и глава 15 «Глобализация команд»).

Специальные ключи в режиме командной строки

Режим командной строки напоминает режим вставки, в том смысле что при нажатии на большинство клавиш на клавиатуре вводятся простые символы. Но, в отличие от режима вставки, в котором введенный текст помещается в буфер, в режиме командной строки текст вводится в специальную строку ввода. В обоих режимах можно использовать управляющие комбинации клавиш для выполнения команд.

О происхождении редактора Vim (и его семейства)

Редактор ed был первым текстовым редактором для Unix. Он был написан во времена, когда дисплеи были редки. Исходный код программ обычно печатался на рулонной бумаге и правился на телетайпе[1]. Вводимые команды отправлялись на ЭВМ для обработки, а вывод каждой команды печатался на принтере. В те дни передача данных между телетайпом и ЭВМ осуществлялась на очень низкой скорости, настолько низкой, что опытный специалист мог вводить команды быстрее, чем они успевали передаваться на ЭВМ. В этой ситуации было важно обеспечить в редакторе ed краткий синтаксис. Например, команда `p` выводит текущую строку, а команда `%p` выводит весь файл.

Редактор ed пережил несколько поколений усовершенствований, включая em (сокращение от англ. «editor for mortals» – редактор для простых смертных), en и, наконец, ex[2]. Со временем дисплеи получили широкое распространение. В редактор ex были добавлены возможности, превратившие экран терминала в интерактивное окно, отображающее содержимое файла. Стало возможным видеть изменения на экране в режиме реального времени. Режим экранного редактирования можно было активировать вводом команды `:visual` или просто `:vi`. Именно от этой команды и произошло имя редактора `vi`.

Название Vim происходит от англ. «vi improved» (улучшенный vi). Улучшенный – это весьма скромная оценка. Лично я терпеть не мог регулярные выражения в `vi`! Загляните в справку `:h vi-differences`

---

1 http://ru.wikipedia.org/wiki/Телетайп

2 http://www.theregister.co.uk/2003/09/11/bill_joys_greatest_gift/

http://vimdoc.sourceforge.net/htmldoc/vi_diff.html#vi-differences, где приводится перечень функциональных возможностей Vim, недоступных в `vi`. Улучшения в Vim огромны, но в нем довольно много осталось от прошлого. Ограничения, накладывавшиеся на предков редактора Vim, способствовали появлению набора весьма эффективных команд, не потерявших свою ценность до настоящего времени.

Некоторые из этих команд действуют одинаково в режиме вставки и в режиме командной строки. Например, `<C-w>` и `<C-u>` удаляют символы левее курсора до начала слова или текущей строки соответственно. Для вставки символов, отсутствующих на клавиатуре, можно использовать команды `<C-v>` и `<C-k>`. А команду `<C-r>{register}` можно использовать для вставки в командную строку содержимого любого регистра, как было показано в рецепте 15 (глава 3). В режиме командной строки имеется множество команд, недоступных в режиме вставки. С некоторыми из них мы познакомимся в рецепте 33 ниже.

В режиме командной строки поддерживается также ограниченная возможность перемещения курсора. Клавиши со стрелками влево и вправо перемещают курсор на один символ в соответствующем направлении. В сравнении с богатейшим набором определений областей действия команд, доступных в командном режиме, это может показаться существенным ограничением. Но, как будет показано в рецепте 34 ниже, командная строка Vim поддерживает все необходимые средства для составления сложных команд.

Команды Ex стреляют дальше и накрывают большую площадь

Иногда бывает проще выполнить команду Ex, чем производить те же операции в командном режиме Vim. Например, область действия команд командного режима обычно начинается с текущего символа или с текущей строки, тогда как команды Ex могут выполняться независимо от текущей позиции курсора. Это означает, что команды Ex можно использовать для внесения изменений без перемещения курсора. Но самое большое отличие команд Ex – в том, что они могут применяться сразу к нескольким строкам.

Можно смело утверждать, что команды Ex имеют более широкую область действия и способны воздействовать сразу на множество строк. Или, образно говоря, команды Ex стреляют дальше и накрывают большую площадь.

Рецепт 28. Выполнение команд для одной строки или для группы смежных строк

Многие команды редактора Ex принимают диапазон строк [range], к которым они применяются. Первую и последнюю строки диапазона можно указать в виде номеров строк, меток или шаблона.

Одной из сильных сторон команд Ex является возможность применения их ко множеству строк. Рассмотрим, как можно использовать ее на примере следующего фрагмента разметки HTML:

ex_mode/practical-vim.html
http://media.pragprog.com/titles/dnvim/code/ex_mode/practical-vim.html

```
1 <!DOCTYPE html>
2 <html>
3   <head><title>Practical Vim</title></head>
4   <body><h1>Practical Vim</h1></body>
5 </html>
```

Возьмем для демонстрации команду :print, которая просто выводит указанные строки ниже командной строки. Эта команда не выполняет никакой полезной работы, но с ее помощью легко продемонстрировать порядок определения диапазона. Попробуйте заменить :print в каждом из последующих примеров такой командой, как :delete, :join, :substitute или :normal, и вы получите представление о том, насколько полезными могут быть команды Ex.

Номера строк

Если ввести команду Ex, состоящую из единственного числа, Vim будет интерпретировать ее как адрес и переместит курсор в указанную строку. Мы можем перейти в начало файла, выполнив следующие команды:

```
⇒ :1
⇒ :print
  1 <!DOCTYPE html>
```

Наш фрагмент содержит всего пять строк. Если потребуется перейти в конец файла, можно ввести команду :5 или использовать специальный символ $:

```
⇒ :$
⇒ :p
  5 </html>
```

Здесь использована команда `:p` – более короткая форма команды `:print`. Подобные команды можно объединять в одну:

```
⇒ :3p
  3 <head><title>Practical Vim</title></head>
```

Эта команда переместит курсор в строку 3 и выведет ее содержимое. Не забывайте, что команда `:p` используется здесь исключительно для иллюстрации. Если выполнить команду `:3d`, мы одним движением заставим редактор Vim перейти в строку 3 и удалить ее. Чтобы выполнить то же действие в командном режиме, нам потребовалось бы выполнить команду `3G`, а затем команду `dd`. Это один из примеров, когда команды Ex могут оказаться более удобными, чем команды командного режима.

Определяйте диапазон строк с использованием их номеров

До сих пор мы использовали номер, чтобы указать номер единственной строки. Но точно так же можно указать диапазон строк. Например:

```
⇒ :2,5p
  2 <html>
  3   <head><title>Practical Vim</title></head>
  4   <body><h1>Practical Vim</h1></body>
  5 </html>
```

Эта команда выведет строки со 2 по 5 включительно. Обратите внимание, что после выполнения команды курсор переместился в строку с номером 5. В общем случае диапазон строк определяется как:

```
:{start},{end}
```

Обратите внимание, что оба параметра – `{start}` и `{end}` – являются адресами строк. До сих пор в качестве адресов использовались номера строк, но чуть ниже мы увидим, что в качестве адресов могут также использоваться шаблоны и метки.

В качестве адреса, представляющего текущую строку, можно использовать символ точки (`.`). То есть мы легко можем выразить диапазон строк от текущей строки и до конца файла:

```
⇒ :2
⇒ :.,$p
  2 <html>
  3    <head><title>Practical Vim</title></head>
  4    <body><h1>Practical Vim</h1></body>
  5 </html>
```

Специальное значение имеет также символ `%` – он обозначает все строки в текущем файле:

```
⇒ :%p
  1 <!DOCTYPE html>
  2 <html>
  3    <head><title>Practical Vim</title></head>
  4    <body><h1>Practical Vim</h1></body>
  5 </html>
```

Аналогичное действие имеет команда `:1,$p`. Это сокращение часто используется в сочетании с командой `:substitute`:

```
⇒ :%s/Practical/Pragmatic/
```

Данная команда потребует от Vim заменить первое вхождение слова «Practical» в каждой строке на «Pragmatic». Подробнее об этой команде будет рассказываться в главе 14 «Подстановка».

Определяйте диапазон строк посредством визуального выделения

Вместо адресации строк их номерами можно использовать выделение в визуальном режиме. Если выполнить команду `2G` и сразу за ней команду `VG`, будет создано визуальное выделение, как показано ниже:

```
<!DOCTYPE html>
<html>
   <head><title>Practical Vim</title></head>
   <body><h1>Practical Vim</h1></body>
</html>
```

Если теперь нажать клавишу `:`, в командной строке появится диапазон `:'<,'>`. Выглядит немного странно, но вы можете считать эту последовательность символов обычным диапазоном, описывающим

визуальное выделение. Вслед за ним можно указать команду Ex и применить ее к каждой выделенной строке:

```
⇒ :'<,'>p
  2 <html>
  3   <head><title>Practical Vim</title></head>
  4   <body><h1>Practical Vim</h1></body>
  5 </html>
```

Такой способ определения диапазона строк может пригодиться для применения команды `:substitute` к подмножеству строк в файле.

Последовательность символов `'<` соответствует первой строке в визуальном выделении, а последовательность `'>` – последней строке в этом же выделении (дополнительную информацию о метках можно найти в рецепте 54 в главе 8). Эти метки продолжают существовать даже после выхода из визуального режима. Если попытаться выполнить команду `:'<,'>p` непосредственно в командном режиме, она всегда будет применяться к строкам, выделенным последними в визуальном режиме.

Определяйте диапазон строк с помощью шаблонов

В качестве адресов строк для команд Ex можно также использовать шаблоны, как показано ниже:

```
⇒ :/<html>/,/<\/html>/p
  2 <html>
  3   <head><title>Practical Vim</title></head>
  4   <body><h1>Practical Vim</h1></body>
  5 </html>
```

Такой способ выглядит достаточно сложным, но в нем используется обычная форма определения диапазона: `:{start},{end}`. Роль адреса `{start}` в данном случае играет шаблон `/<html>/`, а роль адреса `{end}` – шаблон `/<\/html>/`. Иными словами, диапазон начинается со строки, содержащей открывающий тег `<html>`, и заканчивается строкой, содержащей соответствующий закрывающий тег.

В данном конкретном примере тот же результат можно было бы получить, используя адрес `:2,5`, который короче, но более тесно связан с количеством строк в файле. Если для определения диапазона, включающего элемент `<html></html>` целиком, использовать шаблоны, становится неважно, сколько строк содержится в файле.

Изменяйте адрес с помощью смещения

Допустим, что необходимо выполнить команду Ex для всех строк внутри элемента `<html></html>`, кроме строк, содержащих теги `<html>` и `</html>`. Добиться этого можно, определив смещение:

```
⇒ :/<html>/+1,/<\/html>/-1p
  3 <head><title>Practical Vim</title></head>
  4 <body><h1>Practical Vim</h1></body>
```

В общем случае смещение определяется как:

```
:{address}+n
```

Если значение `n` отсутствует, по умолчанию оно принимается равным 1. Адрес `{address}` может быть номером строки, меткой или шаблоном.

Допустим, что нам потребовалось применить команду к конкретному количеству строк, начиная с текущей. Для этого можно было бы указать смещения относительно текущей строки:

```
⇒  :2
⇒  :.,.+3p
```

Символ точки (`.`) обозначает текущую строку, то есть команда `:.,.+3` в данном примере эквивалентна команде `:2,5`.

Обсуждение

Синтаксис определения диапазонов обладает очень большой гибкостью. Мы можем смешивать и сопоставлять номера строк, метки и шаблоны, а также применять смещения к любым из них. Ниже перечислены некоторые символы, которые можно использовать в качестве адресов и диапазонов в командах Ex:

Символ	Адрес
`1`	Первая строка в файле
`$`	Последняя строка в файле
`0`	(Ноль) Виртуальная строка в файле, выше первой строки
`.`	Текущая строка, где находится курсор
`'m`	Строка с меткой m
`'<`	Начало визуального выделения
`'>`	Конец визуального выделения
`%`	Весь файл (более краткая форма записи диапазона `:1,$`

Строка 0 в действительности не существует, но в некоторых ситуациях такой адрес может оказаться как нельзя кстати. В частности, его можно использовать в качестве последнего аргумента команд `:copy{address}` и `:move{address}`, когда требуется скопировать или переместить диапазон строк в начало файла. Примеры этих команд будут показаны в следующих двух рецептах.

Когда определяется диапазон строк, он всегда представляет множество строк, следующих друг за другом. Однако существует возможность применять команды Ex к множеству строк, не следующих друг за другом. Этой цели служит команда `:global`. Подробнее о ней будет рассказываться в главе 15 «Глобализация команд».

Рецепт 29. Копируйте и перемещайте строки с помощью команд ':t' и ':m'

Команда `:copy` (и ее более краткая форма `:t`) позволяет скопировать одну или более строк из одной части документа в другую, а команда `:move` – переместить их.

Для демонстрации возьмем список покупок, составленный перед походом в магазин:

ex_mode/shopping-list.todo
http://media.pragprog.com/titles/dnvim/code/ex_mode/shopping-list.todo

```
1 Shopping list
2     Hardware Store
3         Buy new hammer
4     Beauty Parlor
5         Buy nail polish remover
6         Buy nails
```

Копируйте строки командой :t

Наш список покупок неполон: в хозяйственном магазине нам также требуется купить гвозди (nails). Чтобы добавить новую строку в список, скопируем последнюю строку в файле в раздел «Hardware Store» (хозяйственный магазин). Для этого можно воспользоваться командой `:copy`:

Нажатия клавиш	Содержимое буфера

{start}	`Shopping list`
	`    Hardware Store`
	`        Buy new hammer`
	`    Beauty Parlor`
	`        Buy nail polish remover`
	`        Buy nails`

:6copy.	`Shopping list`
	`    Hardware Store`
	`        Buy nails`
	`        Buy new hammer`
	`    Beauty Parlor`
	`        Buy nail polish remover`
	`        Buy nails`

В общем виде команда копирования имеет следующий формат (см. `:h :copy` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#:copy):

```
:[range]copy{address}
```

В этом примере диапазон `[range]` определяет строку с номером 6, а в качестве адреса `{address}` используется символ точки (`.`), которому соответствует текущая строка. То есть команду `:6copy.` можно прочитать как: «Скопировать строку 6 и вставить ее ниже текущей строки».

Команду `:copy` можно сократить до двух символов, `:co`. А чтобы обеспечить еще большую краткость, можно воспользоваться командой `:t`, которая является синонимом команды `:copy`. Для запоминания ее можно читать как: «copy TO» (копировать в). Ниже приводится несколько примеров использования команды `:t`:

Команда	Действие
:6t.	Вставит копию 6-й строки за текущей
:t6	Скопирует текущую строку и вставит ее за 6-й строкой
:t.	Скопирует текущую строку и вставит ее за текущей (аналог команды yyp в командном режиме)
:t$	Скопирует текущую строку и вставит ее в конец файла
:'<,'>t0	Скопирует выделенные строки в начало файла

Обратите внимание, что команда `:t.` дублирует текущую строку. Тот же эффект дает последовательность команд копирования и вставки в командном режиме (yyp). Единственное существенное различие между этими двумя приемами состоит в том, что коман-

да yyp использует регистр, тогда как команда :t. – нет. Я иногда пользуюсь командой :t. для дублирования текущей строки, когда желательно сохранить текущее содержимое регистра по умолчанию.

В данном примере при желании можно было бы скопировать текущую строку командой yyp, но для этого потребовалось бы предварительно установить курсор. Нам потребовалось бы перейти к требуемой строке (6G), скопировать ее в регистр (yy), вернуться к месту, куда требуется вставить строку (<C-o>), и выполнить команду вставки (p). Команда :t. обычно лучше подходит для дублирования удаленных строк.

В разделе «Команды Ex стреляют дальше и накрывают большую площадь» выше мы видели, что команды командного режима действуют локально, а команды Ex могут воздействовать на более обширные диапазоны строк. Этот пример демонстрирует данный принцип в действии.

Перемещайте строки командой ':m'

Команда :move выглядит так же, как команда :copy (см. :h :move http://vimdoc.sourceforge.net/htmldoc/change.html#:move):

```
:[range]move{address}
```

Ее имя можно сократить до одной буквы: :m. Допустим, что нам потребовалось переместить раздел «Hardware Store» (хозяйственный магазин) после раздела «Beauty Parlor» (салон красоты). Мы могли бы сделать это с помощью команды :move, как показано в табл. 5.2.

При наличии визуального выделения достаточно просто выполнить команду :'<,'>m$. В противном случае можно воспользоваться командой dGp, которая действует так: d удалит визуальное выделение, G выполнит переход в конец файла и p вставит удаленный перед этим текст.

Таблица 5.2. Перемещение группы строк с помощью команды ':m'

Нажатия клавиш	Содержимое буфера

{start}	Shopping list
	    Hardware Store
	        Buy nails
	        Buy new hammer
	    Beauty Parlor
	        Buy nail polish remover
	        Buy nails

Таблица 5.2. Перемещение группы строк с помощью команды ':m' (окончание)

Нажатия клавиш	Содержимое буфера

Vjj	Shopping list
	    Hardware Store
	        Buy nails
	        Buy new hammer
	    Beauty Parlor
	        Buy nail polish remover
	        Buy nails

:'<,'>m$	Shopping list
	    Beauty Parlor
	        Buy nail polish remover
	        Buy nails
	    Hardware Store
	        Buy nails
	        Buy new hammer

Не забывайте, что диапазон `'<,'>` соответствует визуальному выделению. Мы легко можем определить другое визуальное выделение и повторить команду `:'<,'>m$` для перемещения выделенного текста в конец файла. Чтобы повторить последнюю команду Ex, достаточно нажать `@:` (см. рецепт 31 ниже, где приводится еще один пример), то есть такое перемещение проще повторить, чем при использовании команд командного режима.

Рецепт 30. Применение команд командного режима к диапазону строк

Если потребуется применить команду командного режима к последовательности строк, для этого можно воспользоваться командой `:normal`. В комбинации с командой «точка» или макросом этот прием значительно упрощает выполнение повторяющихся операций.

Вернемся к примеру из рецепта 2 в главе 1, где требовалось добавить точку с запятой в конец каждой строки из указанной последовательности. Применение формулы «точки» позволило нам быстро справиться с этой задачей, но в этом примере нам нужно было изменить всего три строки. А как быть, если то же самое потребуется проделать в 50 строках? Следуя формуле «точки», можно было бы нажать `j.` 50 раз, то есть выполнить 100 нажатий на клавиши!

Однако существует более простой путь. Для демонстрации добавим точку с запятой в конец каждой строки в этом файле. Для

экономии места в книге я приведу только пять строк, но если вы сможете представить на их месте 50 строк, то получите более полное представление о возможностях описываемого приема.

ex_mode/foobar.js
http://media.pragprog.com/titles/dnvim/code/ex_mode/foobar.js

```
var foo = 1
var bar = 'a'
var baz = 'z'
var foobar = foo + bar
var foobarbaz = foo + bar + baz
```

Начнем, как и прежде, с изменения первой строки:

Нажатия клавиш	Содержимое буфера

`{start}`	`var foo = 1`
	`var bar = 'a'`
	`var baz = 'z'`
	`var foobar = foo + bar`
	`var foobarbaz = foo + bar + baz`

`A;<Esc>`	`var foo = 1;`
	`var bar = 'a'`
	`var baz = 'z'`
	`var foobar = foo + bar`
	`var foobarbaz = foo + bar + baz`

Нам нужно применить команду «точка» не к одной строке, а к целому диапазону. Для этого можно использовать команду `:normal`:

Нажатия клавиш	Содержимое буфера

`jVG`	`var foo = 1;`
	`var bar = 'a'`
	`var baz = 'z'`
	`var foobar = foo + bar`
	`var foobarbaz = foo + bar + baz`

`:'<,'>normal .`	`var foo = 1;`
	`var bar = 'a';`
	`var baz = 'z';`
	`var foobar = foo + bar;`
	`var foobarbaz = foo + bar + baz;`

Команду `:'<,'>normal.` можно прочитать как: «Для каждой строки в визуальном выделении выполнить команду `.` командного режима». Этот прием будет работать одинаково и для пяти строк, и для пятидесяти. Но истинная его прелесть состоит в том, что нам не нужно даже считать строки – мы можем избежать необходимости выделять их в визуальном режиме.

В этом случае мы использовали команду `:normal` для выполнения команды «точка», но точно так же мы можем выполнить любую команду командного режима. Например, решить поставленную задачу можно единственной командой:

```
⇒ :%normal A;
```

Символ `%`, при использовании в качестве диапазона представляет файл целиком. То есть команда `:%normal A;` сообщает редактору Vim, что он должен добавить точку с запятой в конец каждой строки в файле. Чтобы выполнить эти изменения, редактору приходится переключаться в режим вставки, но после этого он автоматически возвращается в командный режим.

Прежде чем выполнить указанную команду командного режима для каждой строки, Vim перемещает курсор в начало строки. Поэтому нам не приходится беспокоиться о позиционировании курсора при выполнении команды. Следующей одиночной командой можно закомментировать весь файл с кодом на JavaScript:

```
⇒ :%normal i//
```

Хотя команду `:normal` можно использовать в комбинации с любыми командами командного режима, но, на мой взгляд, наибольшего эффекта можно добиться при использовании команд повторения: либо `:normal.` для применения простых изменений, либо `:normal @q` для решения более сложных задач (пару примеров такого подхода вы найдете в рецептах 68 и 70 главы 11).

В разделе «Команды Ex стреляют дальше и накрывают большую площадь» выше уже отмечалось, что команды Ex способны одновременно изменять несколько строк. Команда `:normal` позволяет объединить выразительность командного режима с диапазонами команд Ex. В результате получается весьма мощная комбинация!

Еще одно альтернативное решение задачи, рассматривавшейся в этом примере, можно найти в рецепте 26 в главе 4.

Рецепт 31. Повторяйте последнюю команду Ex

В командном режиме для повторения последней команды можно использовать команду `.`, а для повторения последней команды

Ex – команду `@:`. Знание, как отменить последнюю команду, всегда полезно, поэтому мы рассмотрим и эту тему.

В главе 1 «Путь Vim» мы видели, что команду `.` можно использовать для повторения последнего изменения. Но команда «точка» не повторяет изменений, выполненных в режиме командной строки. Вместо нее для этой цели следует использовать команду `@:` (см. `:h @:` ⓘ http://vimdoc.sourceforge.net/htmldoc/repeat.html#@:).

Например, эта команда может пригодиться для обхода элементов в списке буферов. Выполнить обход списка в прямом направлении можно с помощью команды `:bn[ext]`, а в обратном – с помощью команды `:bp[revious]` (более подробное обсуждение списка буферов приводится в рецепте 37 в главе 6). Допустим, что в списке буферов имеется с десяток элементов, и нам необходимо осмотреть каждый из них. Мы могли бы ввести следующую команду один раз:

```
⇛ :bnext
```

А затем использовать `@:` для повторения команды. Обратите внимание на сходство между этой командой и способом выполнения макроса (см. раздел «Повторное выполнение последовательности команд вызовом макроса» в главе 11). Отметьте также, что регистр `:` всегда хранит самую последнюю команду, выполненную в режиме командной строки (см. `:h quote_:` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#quote_:). После выполнения команды `@:` первый раз ее можно повторить командой `@@`.

Допустим, что мы, набравшись терпения, выполнили команду `@:` очень много раз и в конце концов выполнили ее лишний раз. Как теперь исправить промах? Можно было бы выполнить команду `:bprevious`. Но что произойдет, если после этого снова воспользоваться командой `@:`? Она вернет нас на еще один шаг назад в списке буферов, в противоположность тому, что она делала прежде. Это может привести в замешательство.

В данном случае лучше всего было бы использовать команду `<C-o>` (см. рецепт 56 в главе 9). При каждом выполнении команды `:bnext` (или ее повторении с помощью команды `@:`) в список переходов добавляется новая запись. Команда `<C-o>` производит переход к предыдущей записи в этом списке.

Мы могли бы выполнить `:bnext` один раз и затем повторять с помощью `@:`. При необходимости вернуться можно было бы воспользоваться командой `<C-o>`. А затем, чтобы продолжить движение впе-

ред по списку буферов, мы снова могли бы использовать команду `@:`. Напомню мантру из рецепта 4 в главе 1: действие, повтор, возврат.

В редакторе Vim с помощью команд Ex можно выполнить любые операции. Несмотря на наличие возможности повторить последнюю команду Ex нажатием на клавиши `@:`, обратить действие выполненной команды не всегда так просто, как хотелось бы. Команда `<C-o>`, описанная в этом рецепте, также может применяться для обращения действия команд `:next`, `:cnext` и `:tnext` (и других). Отменить действие любой из команд, перечисленных в табл. 5.1, можно, также нажав `u`.

Рецепт 32. Автодополнение команд Ex

Как и в командной оболочке, для автодополнения команд в режиме командной строки можно использовать клавишу `<Tab>`.

Для выбора предлагаемых вариантов в Vim используется интеллектуальный алгоритм. Он определяет контекст, в котором находится командная строка, и конструирует список наиболее подходящих подсказок. Например, допустим, что мы ввели:

```
⇒ :col<C-d>
  colder      colorscheme
```

Команда `<C-d>` запрашивает у Vim список возможных продолжений команды (см. `:h c_CTRL-D` ⓘ http://vimdoc.sourceforge.net/htmldoc/cmdline.html#c_CTRL-D). Если нажимать клавишу `<Tab>`, в строке ввода последовательно будут появляться команды `colder`, `colorscheme` и снова `col`. Можно выполнить обход подсказок в обратном направлении с помощью клавиш `<S-Tab>`.

Допустим, что нам нужно изменить цветовую схему, но мы не можем вспомнить ее название. Мы могли бы с помощью команды `<C-d>` получить список всех доступных вариантов:

```
⇒ :colorscheme <C-d>
  blackboard  desert        morning     shine
  blue        elflord       murphy      slate
  darkblue    evening       pablo       solarized
  default     koehler       peachpuff   torte
  delek       mac_classic   ron         zellner
```

На этот раз команда `<C-d>` отобразила список подсказок, опираясь на доступные цветовые схемы. Если бы нам потребовалось

активировать схему «solarized», мы могли бы просто ввести буквы «so» и нажать клавишу табуляции, чтобы дополнить команду.

Во многих ситуациях функция автодополнения с помощью клавиши табуляции работает безупречно. Если вводится команда, ожидающая получить в аргументе имя файла (такая как `:edit` или `:write`), нажатие на клавишу <Tab> поможет подставить имена файлов и каталогов относительно текущего рабочего каталога. При вводе команды `:tag` мы получим возможность автодополнения имен тегов. Команды `:set` и `:help` знают о существовании всех параметров настройки Vim.

Имеется даже возможность определять поведение функции автодополнения при создании собственных команд Ex. Чтобы поближе познакомиться с доступными возможностями, обращайтесь к справочному разделу `:h :command-complete` http://vimdoc.sourceforge.net/htmldoc/map.html#:command-complete.

Выбор из нескольких совпадений

Когда Vim находит только одну подсказку для функции автодополнения, он использует найденное совпадение. Но если Vim обнаруживает множество вариантов, тогда события могут развиваться по одному из следующих сценариев. По умолчанию, когда клавиша табуляции нажимается в первый раз, Vim подставляет первую подсказку. При каждом следующем нажатии клавиши табуляции выполняется подстановка остальных подсказок.

Мы можем изменить такое поведение, настроив параметр `wildmode` (см. `:h 'wildmode'` http://vimdoc.sourceforge.net/htmldoc/options.html#%27wildmode%27). Если вы привыкли использовать аналогичную функцию командной оболочки bash, тогда следующие настройки будут максимально соответствовать вашим ожиданиям:

```
set wildmode=longest,list
```

Если вы привыкли использовать меню автодополнения в zsh, возможно, вам понравятся следующие настройки:

```
set wildmenu
set wildmode=full
```

С включенным параметром `wildmenu` Vim будет отображать список с подсказками на выбор, по которому можно перемещаться

вперед, нажимая <Tab>, <C-n> или <Right> (клавиша со стрелкой вправо), или назад, нажимая <S-Tab>, <C-p> или <Left> (клавиша со стрелкой влево).

Рецепт 33. Вставка текущего слова в командную строку

Даже находясь в режиме, Vim всегда знает, где находится курсор и какая часть окна (если оно разбито на части) активна. Для экономии времени имеется возможность вставить текущее слово (или СЛОВО) из активного документа в командную строку.

Если в командной строке Vim нажать последовательность клавиш <C-r><C-w>, он скопирует слово, находящееся под курсором, и вставит его в командную строку. Мы можем использовать эту возможность, чтобы уменьшить объем ввода.

Допустим, что в следующем фрагменте нам требуется переименовать переменную `tally` в `counter`:

ex_mode/loop.js
http://media.pragprog.com/titles/dnvim/code/ex_mode/loop.js

```
var tally;
for (tally=1; tally <= 10; tally++) {
  // выполнить некоторые операции с переменной tally
};
```

Если установить курсор на слово `tally`, мы могли бы выполнить команду `*`, чтобы найти все вхождения этого имени. (Команда `*` эквивалентна последовательности нажатий /\<<C-r><C-w>\><CR>. Роль меток \< и \> в шаблонах подробно описывается в рецепте 77 в главе 12.)

Нажатия клавиш	Содержимое буфера

{start}	var tally;
	for (tally=1; tally <= 10; tally++) {
	    // выполнить некоторые операции с переменной tally
	};

*	var tally;
	for (tally=1; tally <= 10; tally++) {
	    // выполнить некоторые операции с переменной tally
	};

cwcounter<Esc>	var tally;
	for (counter=1; tally <= 10; tally++) {
	    // выполнить некоторые операции с переменной tally
	};

Когда нажимается клавиша *, курсор перепрыгивает к следующему совпадению, но в конечном итоге возвращается к тому же самому слову. Последовательность cwcounter<Esc> вносит изменение.

Остальные изменения в этом примере мы выполним с помощью команды :substitute. Так как курсор теперь находится на слове «counter», нам не требуется вводить его снова. Мы можем воспользоваться последовательностью <C-r><C-w> для заполнения поля, определяющего строку замены:

```
⇒ :%s//<C-r><C-w>/g
```

Эта команда на книжной странице не выглядит короткой, но два нажатия на клавиши, чтобы вставить слово, – очень неплохо. Благодаря ранее выполненной команде * нам не пришлось даже вводить шаблон поиска. Чтобы понять, почему поле шаблона можно оставить пустым, как в данном примере, обращайтесь к рецепту 91 в главе 14.

Команда <C-r><C-w> позволяет получить *слово* под курсором, а чтобы получить СЛОВО, можно воспользоваться командой <C-r><C-a> (описание см. в рецепте 49 в главе 8). За дополнительной информацией обращайтесь к справке :h c_CTRL-R_CTRL-W ℹ http://vimdoc.sourceforge.net/htmldoc/cmdline.html#c_CTRL-R_CTRL-W. В этом примере мы использовали команду :substitute, но описанные комбинации клавиш можно применять с любыми командами Ex.

В качестве упражнения откройте файл *vimrc*, установите курсор на название какого-либо параметра настройки и затем введите :help <C-r><C-w>, чтобы перейти к справке с описанием этого параметра.

Рецепт 34. Повторный вызов команд из истории

Редактор Vim записывает команды, выполнявшиеся в режиме командной строки, и предоставляет два способа повторного их вызова: посредством прокрутки клавишами управления курсором или посредством ввода номера команды.

Vim запоминает команды, выполнявшиеся в режиме командной строки. Мы легко можем повторно вызвать любую команду, вызывавшуюся прежде, и тем самым избежать необходимости повторного ввода длинных команд Ex.

Для начала войдите в режим командной строки, нажав клавишу `:`. Оставьте строку ввода пустой и нажмите клавишу со стрелкой вверх `<Up>`. Командная строка должна заполниться самой последней выполненной командой Ex. Нажмите клавишу `<Up>` еще раз, чтобы перейти к следующей команде в истории, или клавишу `<Down>`, чтобы двинуться в обратном направлении.

Теперь попробуйте ввести `:help` и понажимать клавишу `<Up>`. И снова в строке ввода должны последовательно появляться прежде выполнявшиеся команды Ex, но на этот раз не все подряд, а только начинающиеся со слова «help».

По умолчанию Vim записывает последние 20 команд. Учитывая, что современные компьютеры снабжаются огромными объемами памяти, мы можем позволить себе увеличить этот предел, изменив параметр настройки `history`. Попробуйте добавить следующую строку в файл *vimrc*:

```
set history=200
```

Имейте в виду, что история команд сохраняется не только для текущего сеанса работы с редактором. Она сохранится, даже если закрыть и вновь запустить Vim (см. `:h viminfo` http://vimdoc.sourceforge.net/htmldoc/starting.html#viminfo). Увеличение количества команд, сохраняемых в истории, может пригодиться вам в дальнейшем.

Помимо сохранения истории выполнения команд Ex, Vim отдельно запоминает историю операций поиска. Если нажать `/`, чтобы открыть строку ввода шаблона для поиска, с помощью клавиш `<Up>` и `<Down>` можно будет просматривать историю поиска. В конце концов, строка ввода шаблона – это всего лишь еще одна разновидность режима командной строки.

Встречайте: окно режима командной строки

Как и режим вставки, режим командной строки отлично подходит для ввода текста, что называется «с нуля», но он не очень удобен для редактирования текста.

Допустим, что мы правим простой сценарий на языке Ruby. Каждый раз, внося изменения, мы обнаруживаем, что выполняем следующие две команды:

```
⇒ :write
⇒ :!ruby %
```

Выполнив эти две команды пару раз, мы замечаем, что могли сэкономить на усилиях, заключив обе команды в одну командную строку. Благодаря этому мы сможем выбрать полную команду из истории и повторно выполнить ее:

```
⇒ :write | !ruby %
```

Каждая из этих команд по отдельности уже находится в истории, поэтому нам не нужно вводить полную команду «с нуля». Но как объединить две записи из истории в одну? Нажмите `q:`, и на экране появится окно с историей команд (см. `:h cmdwin` ⓘ http://vimdoc.sourceforge.net/htmldoc/cmdline.html#cmdwin).

Окно с историей командной строки напоминает обычный буфер, где каждая строка содержит элемент из истории. Мы можем перемещаться по истории, нажимая клавиши `k` и `j`. Или использовать средства поиска в Vim, чтобы найти нужную команду. Нажатие клавиши `<CR>` приведет к выполнению содержимого текущей строки как команды Ex.

Вся прелесть окна режима командной строки – в том, что оно позволяет изменять строки в истории команд, используя для этого всю мощь редактора Vim. Мы можем перемещаться по окну, применяя команды перемещения, доступные в командном режиме, использовать механизм визуального выделения или переключаться в режим вставки. Мы можем даже применять команды Ex к содержимому окна режима командной строки!

Открыв окно режима командной строки нажатием `q:`, мы можем решить нашу проблему, как показано ниже:

Нажатия клавиш	Содержимое буфера
{start}	write !ruby %
A␣|<Esc>	write | !ruby %
J	write	!ruby %
:s/write/update	update	!ruby %

Если теперь нажать клавишу `<CR>`, редактор выполнит команду `:update | !ruby %`, как если бы мы ввели ее в режиме командной строки.

Когда окно режима командной строки открыто, оно всегда получает фокус ввода. Это означает, что мы не сможем переключиться

в другое окно, не закрыв этого. Закрыть окно режима командной строки можно командой `:q` (как любое другое окно Vim) или нажав `<CR>`.

Имейте в виду, что при нажатии `<CR>` в окне режима командной строки команда выполняется в контексте активного окна, то есть в контексте окна, которое было активно до открытия окна режима командной строки. Редактор Vim никак не обозначает окно, которое было активным перед открытием окна режима командной строки, поэтому будьте внимательны, если пользуетесь функцией разбиения окон!

А как быть, если в середине ввода команды Ex мы обнаруживаем, что нам нужны более мощные возможности правки? Находясь в режиме командной строки, можно нажать комбинацию `<C-f>`, чтобы переключиться в окно режима командной строки, сохранив при этом наполовину введенную команду. Ниже перечислено несколько способов открытия окна режима командной строки:

Команда	Действие
`q/`	Открыть окно режима командной строки с историей поиска
`q:`	Открыть окно режима командной строки с историей команд Ex
`C-f`	Перейти из командной строки в окно режима командной строки

Команды `q:` и `:q` легко перепутать. Я уверен, каждый, кто когда-либо пользовался Vim, случайно открывал окно режима командной строки, пытаясь закрыть редактор! Окно режима командной строки действительно очень полезно, но многие сильно расстраиваются, столкнувшись с ним впервые (случайно). Прочитайте рецепт 86 в главе 13, где приводится еще один пример использования окна режима командной строки.

Рецепт 35. Выполнение команд в оболочке

Мы легко можем запускать внешние программы, не покидая Vim. Но самое замечательное – мы можем передавать содержимое буфера на вход команды или сохранять вывод команды в буфере.

Команды, обсуждаемые в этом рецепте, лучше всего использовать в редакторе Vim, выполняющемся в окне терминала. Если вы используете GVim (или MacVim), некоторые команды могут работать не так, как описывается здесь. Это не должно быть большим сюр-

призом. Редактору Vim намного проще делегировать выполнение операций командной оболочке, когда он сам выполняется в командной оболочке. GVim может делать что-то лучше, чем Vim, но выполнение команд оболочки – это одна из областей, где Vim оказывается впереди.

Запуск программ в командной оболочке

Находясь в режиме командной строки, можно запускать внешние программы, добавляя в начало команд восклицательный знак (см. `:h :!` http://vimdoc.sourceforge.net/htmldoc/various.html#:!). Например, если потребуется выяснить содержимое текущего каталога, можно выполнить следующую команду:

```
⇒ :!ls
  duplicate.todo          loop.js
  emails.csv              practical-vim.html
  foobar.js               shopping-list.todo
  history-scrollers.vim

  Press ENTER or type command to continue
```

Обратите внимание на разницу между командами `:!ls` и `:ls` – первая из них вызовет внешнюю команду `ls` в командной оболочке, а команда `:ls` вызовет встроенную команду Vim, которая выведет список буферов.

Символ `%` в командной строке Vim является сокращенной формой имени текущего файла (см. `:h cmdline-special` http://vimdoc.sourceforge.net/htmldoc/cmdline.html#cmdline-special). Мы можем использовать это обстоятельство для выполнения каких-либо операций с текущим файлом с помощью внешних команд. Например, редактируя сценарий на языке Ruby, его можно выполнить следующей командой:

```
⇒ :!ruby %
```

Редактор Vim предоставляет также множество модификаторов, позволяющих извлекать информацию из имени текущего файла, такую как путь к файлу или расширение (см. `:h filename-modifiers` http://vimdoc.sourceforge.net/htmldoc/cmdline.html#filename-modifiers). Загляните в рецепт 45 (глава 7), где дается пример их использования.

Синтаксис `:!{cmd}` отлично подходит для запуска одиночных команд, но как быть, если потребуется выполнить несколько команд в командной оболочке? В этом случае можно воспользоваться командой Vim `:shell`, которая запускает интерактивный сеанс командной оболочки (см. `:h :shell` ⓘ http://vimdoc.sourceforge.net/htmldoc/various.html#:shell):

```
⇛ :shell
⇛ $ pwd
  /Users/drew/books/PracticalVim/code/ex_mode
⇛ $ ls
  duplicate.todo          loop.js
  emails.csv              practical-vim.html
  foobar.js               shopping-list.todo
  history-scrollers.vim
⇛ $ exit
```

Команда `exit` закроет сеанс и вернет нас в Vim.

Перевод редактора в фоновый режим работы

Команда `:shell` позволяет переключиться в интерактивный сеанс командной оболочки. Но если Vim уже выполняется в терминале, нам становятся доступны встроенные команды оболочки. Например, поддержка команд управления заданиями в оболочке bash, которые позволяют приостанавливать задания, переводя их в фоновый режим, и возобновлять – переводя их обратно на передний план.

Допустим, что мы запустили Vim в командной оболочке bash и нам нужно выполнить серию команд оболочки. Нажатие комбинации `Ctrl`-`z` приостановит процесс, в котором выполняется Vim, и передаст управление оболочке bash. Процесс Vim будет простаивать в фоновом режиме, позволяя нам взаимодействовать с командной оболочкой как обычно. Получить список заданий можно с помощью следующей команды:

```
⇛ $ jobs
  [1]+ Stopped      vim
```

Возобновить приостановленное задание в оболочке bash можно командой `fg`, которая переведет указанное задание на передний план. Она вернет Vim к жизни точно в том состоянии, в каком мы его оставили. Команды `Ctrl`-`z` и `fg` более просты в использовании, чем эквивалентные им команды Vim `:shell` и `exit`. Чтобы получить дополнительную информацию, выполните команду `man bash` и прочитайте раздел об управлении заданиями.

Передача содержимого буфера на вход командам и сохранение вывода команд в буфере

При использовании синтаксиса `:!{cmd}` Vim повторяет вывод команды `{cmd}`. Все хорошо, пока команда выводит не так много информации, но иногда команды могут выводить большое количество строк. В таких ситуациях можно использовать команду `:read !{cmd}`, которая сохранит вывод команды `{cmd}` в текущем буфере (см. `:h read!` ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#read!).

Команда `:read !{cmd}` переадресует стандартный вывод выполняемой команды в буфер. Как вы уже наверняка догадались, команда `:write !{cmd}` выполняет обратное действие: она передает содержимое буфера на стандартный ввод указанной команды `{cmd}` (см. `:h :write_c` ⓘ http://vimdoc.sourceforge.net/htmldoc/editing.html#:write_c). Загляните в рецепт 46 (глава 7), где приводится пример использования этой особенности.

Восклицательный знак может придавать командам разный смысл в зависимости от местоположения в командной строке. Сравните следующие команды:

```
⇛ :write !sh
⇛ :write ! sh
⇛ :write! sh
```

Первые две предают содержимое буфера на стандартный ввод внешней команде `sh`. Последняя команда записывает содержимое буфера в файл с именем `sh`, вызывая команду `:write!`. В последнем случае восклицательный знак сообщает редактору Vim, что он должен перезаписать файл `sh`, если таковой существует. Как видите, местоположение восклицательного знака оказывает существенное влияние на результат. Будьте внимательны при составлении команд такого рода!

В результате вызова команды `:write !sh` в командной оболочке будет выполнена каждая строка из текущего буфера. Отличный пример использования этой команды можно найти в справке `:h rename-files` ⓘ http://vimdoc.sourceforge.net/htmldoc/tips.html#rename-files.

Фильтрация содержимого буфера с помощью внешней команды

Команда `:!{cmd}` приобретает иной смысл, когда ей передается диапазон строк. Строки, определяемые диапазоном `[range]`, передаются на стандартный ввод команды `{cmd}`, а ее вывод затирает оригинальное содержимое диапазона. Или, говоря иными словами, команда `{cmd}` фильтрует указанный диапазон `[range]` строк (см. `:h :range!` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#:range!). Редактор Vim определяет фильтр как «программу, принимающую текст со стандартного ввода, изменяющую его некоторым способом и возвращающую результат в стандартный вывод».

Для демонстрации воспользуемся внешней командой `sort` и переупорядочим записи в файле CSV:

ex_mode/emails.csv
media.pragprog.com/titles/dnvim/code/ex_mode/emails.csv

```
first name,last name,email
john,smith,john@example.com
drew,neil,drew@vimcasts.org
jane,doe,jane@example.com
```

Отсортируем записи по второму полю: по фамилии. Сообщить команде `sort`, что поля разделены запятой, можно с помощью ключа `-t','`, а указать номер поля, по которому следует выполнять сортировку, – с помощью ключа `-k2`.

Первая строка в файле содержит заголовочную информацию. Ее желательно оставить в начале файла, поэтому мы исключим ее из операции сортировки, указав диапазон `:2,$`. Требуемую нам сортировку выполнит следующая команда:

```
⇒ :2,$!sort -t',' -k2
```

Теперь записи в нашем файле CSV должны быть отсортированы по фамилии:

```
first name,last name,email
jane,doe,jane@example.com
drew,neil,drew@vimcasts.org
john,smith,john@example.com
```

Редактор Vim поддерживает удобную краткую форму определения диапазона команды `:[range]!{filter}`. Команда-оператор

!{motion} переведет редактор в режим командной строки и подставит в поле [range] диапазон, соответствующий указанному аргументу {motion} (см. :h ! http://vimdoc.sourceforge.net/htmldoc/change.html#!). Например, если поместить курсор в строку 2 и выполнить !G, Vim откроет командную строку и подставит в нее диапазон :.,$!. Правда, нам все еще нужно ввести остаток команды {filter}, но все-таки нам удалось сэкономить на нажатиях клавиш.

Обсуждение

При работе в Vim мы никогда не будем удаляться от командной оболочки дальше, чем на пару нажатий на клавиши, чтобы выполнить команду оболочки. В следующей таблице приводится список наиболее полезных приемов вызова внешних команд:

Команда	Действие
:shell	Запускает командную оболочку (возврат в Vim выполняется командой exit)
:!{cmd}	Выполняет команду {cmd} в командной оболочке
:read !{cmd}	Выполняет команду {cmd} в командной оболочке и вставляет ее вывод ниже курсора
:[range]write !{cmd}	Выполняет команду {cmd} в командной оболочке и передает на ее стандартный ввод указанный диапазон строк [range]
:[range]!{filter}	Фильтрует указанный диапазон строк [range] с помощью внешней команды {filter}

Некоторые команды интерпретируются редактором Vim особо. Например, для команд make и grep имеются отдельные команды-обертки. Они не только упрощают запуск этих команд внутри Vim, но и анализируют их вывод и используют его для заполнения списка результатов (quickfix list). Более подробно эти команды рассматриваются в главе 17 «Компиляция кода и обзор ошибок с помощью Quickfix List» и в главе 18 «Поиск в пределах проекта с помощью команд grep, vimgrep и других».

Рецепт 36. Выполнение сразу нескольких команд Ex

Если потребуется выполнить последовательность из нескольких команд Ex, есть возможность сэкономить силы и время, оформив эту последовательность в виде сценария. Когда в следующий раз понадобится выполнить эту же последовательность команд, доста-

точно будет просто запустить сценарий, вместо того чтобы вводить команды по одной.

Следующий файл содержит ссылки на первую пару эпизодов из архива Vimcasts:

cmdline_mode/vimcasts/episodes-1.html
http://media.pragprog.com/titles/dnvim2/code/cmdline_mode/vimcasts/episodes-1.html

```
<ol>
  <li>
    <a href="/episodes/show-invisibles/">
      Show invisibles
    </a>
  </li>
  <li>
    <a href="/episodes/tabs-and-spaces/">
      Tabs and Spaces
    </a>
  </li>
</ol>
```

Эту разметку нужно преобразовать в простой текстовый формат с названием и адресом URL, следующим за ним:

cmdline_mode/vimcasts-episodes-1.txt
http://media.pragprog.com/titles/dnvim2/code/cmdline_mode/vimcasts-episodes-1.txt

```
Show invisibles: http://vimcasts.org/episodes/show-invisibles/
Tabs and Spaces: http://vimcasts.org/episodes/tabs-and-spaces/
```

Допустим, что данное преобразование нам необходимо применить к целой последовательности файлов, имеющих похожий формат. Далее мы рассмотрим два разных способа решения поставленной задачи.

Выполнение команд Ex по одной

Требуемое преобразование можно выполнить единственной командой `:substitute`, но я предпочел разбить ее на несколько команд, решающих более мелкие задачи. Следующая последовательность команд Ex – одно из возможных решений:

```
⇒ :g/href/j
⇒ :v/href/d
на 8 строк меньше
⇒ :%norm A: http://vimcasts.org
⇒ :%norm yi"$p
⇒ :%s/\v^[^\>]+\>\s//g
```

Необязательно точно понимать, что делает каждая команда, чтобы продолжить знакомство с рецептом, но для тех, кому интересно, коротко опишу происходящее. Команды `:global` и `:vglobal` вместе свертывают файл до четырех строк, содержащих необходимую нам информацию, пусть и в неправильном порядке (см. рецепт 99 в главе 15). Команды `:normal` добавляют URL в конец строки (см. рецепт 30 в главе 5). И команда `:substitute` удаляет открывающий тег <a href="">. Как всегда, лучший способ понять команды – опробовать их.

Запись команд Ex в сценарий и его выполнение

Вместо выполнения команд по одной их можно поместить в файл на диске. Назовем этот файл `batch.vim`. (Расширение `.vim` позволит редактору Vim правильно выделить синтаксис.) Каждая строка в этом файле соответствует команде из последовательности, представленной выше. Внутри сценария не требуется предварять каждую команду двоеточием (`:`). Лично я предпочитаю использовать длинные имена команд Ex в сценариях. Удобочитаемость сценария дороже экономии на нажатиях клавиш.

cmdline_mode/batch.vim
http://media.pragprog.com/titles/dnvim2/code/cmdline_mode/batch.vim

```
global/href/join
vglobal/href/delete
%normal A: http://vimcasts.org
%normal yi"$p
%substitute/\v^[^\>]+\>\s//g
```

Запустить сценарий `batch.vim` можно командой :source (см. `:h source` http://vimhelp.appspot.com/repeat.txt.html#%3Asource). Каждая строка сценария выполняется как команда Ex, как если бы она вводилась в командной строке Vim. Возможно, вам приходилось использовать команду `:source` прежде, в других контекстах: она часто применяется для загрузки настроек из файла `vimrc` во время работы. (За дополнительной информацией обращайтесь к разделу «Сохраняйте настройки в файле vimrc» в приложении A.)

Я предлагаю опробовать этот сценарий прямо сейчас. Исходный код сценария можно загрузить на сайте издательства Pragmatic Bookshelf, на странице книги «Practical Vim». Прежде чем открыть

Vim, перейдите в каталог `cmdline_mode`, где хранятся оба файла, `batch.vim` и `episodes-1.html`.

```
⇒ $ pwd
  ~/dnvim2/code/cmdline_mode
⇒ $ ls *.vim
    batch.vim history-scrollers.vim
⇒ $ vim vimcasts/episodes-1.html
```

Теперь можно выполнить сценарий:

```
⇒ :source batch.vim
```

Всего одной командой мы выполнили все команды Ex в файле `batch.vim`. Если вдруг вы передумаете, изменения можно отменить, нажав клавишу `u` один раз.

Запуск сценария для изменения нескольких файлов

Нет смысла сохранять последовательность команд Ex в файле, если сценарий будет выполняться только один раз. Этот прием особенно полезен, когда ту же последовательность команд требуется выполнить многократно.

В пакете примеров к данной книге имеется несколько файлов того же формата, что и файл `episodes-1.html`. Перейдите в каталог `cmdline_mode` и запустите Vim:

```
⇒ $ pwd
  ~/dnvim2/code/cmdline_mode
⇒ $ ls vimcasts
  episodes-1.html episodes-2.html episodes-3.html
⇒ $ vim vimcasts/*.html
```

Когда Vim запускается с групповым символом, командная оболочка заполнит список аргументов файлами, соответствующими указанному шаблону. Мы могли бы обойти файлы по одному и выполнить `batch.vim` отдельно для каждого из них:

```
⇒ :args
  [vimcasts/episodes-1.html] vimcasts/episodes-2.html vimcasts/episodes-3.html
⇒ :first
⇒ :source batch.vim
⇒ :next
⇒ :source batch.vim
  и так далее.
```

Но лучше воспользоваться командой `:argdo` (см. `:h :argdo` http://vimhelp.appspot.com/editing.txt.html#%3Aargdo):

```
⇒ :argdo source batch.vim
```

Бум! Одной командой мы применили все команды Ex в `batch.vim` ко всем файлам, перечисленным в списке аргументов.

Я выбрал такой сложный путь со множеством команд Ex, только чтобы продемонстрировать саму возможность. На практике я использую этот подход в основном для выполнения одной или нескольких команд `:substitute`, если обнаруживаю, что использую их снова и снова. Я часто удаляю файл `batch.vim` после использования, но точно так же могу сохранить в репозитории, если решу, что он может пригодиться в будущем.

Часть II. ФАЙЛЫ

В этой части книги мы познакомимся с особенностями работы с файлами и буферами. Редактор Vim позволяет управлять сразу несколькими файлами в рамках одного сеанса. Их можно просматривать по одному либо разбивать рабочее пространство на окна или вкладки, содержащие отдельные буферы. Мы увидим несколько разных способов открытия файлов из Vim. Мы также познакомимся с парой приемов решения проблем, способных помешать сохранению буферов в файлах.

Глава 6. Управление несколькими файлами

Редактор Vim позволяет работать с несколькими файлами одновременно. Список буферов помогает следить за тем, какие файлы были открыты в ходе сеанса. В рецепте 37 мы узнаем, как управлять этим списком, а также научимся различать файлы и буферы.

Список аргументов – самое лучшее дополнение к списку буферов. В рецепте 38 мы увидим, как пользоваться командой `:args` для группировки файлов из списка буферов в коллекции, которые дают возможность выполнять обход списка или применять команды Ex ко всем членам коллекций с помощью команды `:argdo`.

Редактор Vim позволяет делить рабочее пространство на окна, о чем будет рассказываться в рецепте 40. Затем в рецепте 40 мы познакомимся с интерфейсом вкладок, используемым для организации окон в коллекции.

Рецепт 37. Слежение за открытыми файлами с помощью списка буферов

В процессе сеанса работы с редактором Vim можно загрузить множество файлов и управлять ими с помощью списка буферов.

Различия между файлами и буферами

Как и любой другой текстовый редактор, Vim позволяет читать файлы, править их и сохранять изменения. При обсуждении процесса работы мы позволяем себе вольность говорить, что мы редактируем файл, хотя в действительности это не так. На самом деле мы правим представление файла в памяти, которое в терминологии Vim называется *буфером*.

Файлы хранятся на диске, тогда как буфер хранится в памяти. Когда мы открываем файл в Vim, его содержимое читается в буфер с тем же именем. Изначально содержимое буфера идентично содержимому файла, но оно начинает отличаться с того момента, как мы начинаем вносить изменения в буфер. Если мы пожелаем сохранить изменения, мы можем записать содержимое буфера обратно в файл. Большинство команд в Vim оперирует буферами, но есть и такие, которые оперируют файлами, включая команды `:write`, `:update` и `:saveas`.

Встречайте: список буферов

Редактор Vim позволяет работать с несколькими буферами одновременно. Давайте откроем несколько файлов, выполнив следующие команды в командной оболочке:

```
⇒ $ cd code/files
⇒ $ vim *.txt
  2 files to edit
```

Шаблону `*.txt` в данном случае соответствуют два файла, находящиеся в текущем каталоге: *a.txt* и *b.txt*. Эта команда предлагает редактору Vim открыть оба файла. Когда Vim запустится, он отобразит единственное окно с буфером, в котором хранится содержимое первого из двух файлов. Второй буфер останется невидимым, но в него будет загружено содержимое второго файла, в чем можно убедиться, выполнив следующую команду:

```
⇒ :ls
  1 %a  "a.txt"         line 1
  2     "b.txt"         line 0
```

Команда `:ls` выводит список всех буферов, присутствующих в памяти (`:h :ls` ⓘ http://vimdoc.sourceforge.net/htmldoc/windows.html#:ls). Переключиться на следующий буфер в списке можно командой `:bnext`:

```
⇒ :bnext
⇒ :ls
  1 #   "a.txt"         line 1
  2 %a  "b.txt"         line 1
```

Символ `%` показывает, какой из буферов отображается в текущем окне, а символом `#` отмечается альтернативный файл. Мы легко мо-

жем переключаться между текущим и альтернативным файлами, нажимая `<C-^>`. Нажав эту комбинацию один раз, мы переключимся на файл *a.txt*; нажав ее второй раз, мы вернемся к файлу *b.txt*.

Пользуйтесь списком буферов

Для обхода элементов списка буферов предусмотрено четыре команды – `:bprev` и `:bnext` перемещаются вперед и назад по списку, а команды `:bfirst` и `:blast` выполняют переход в начало и в конец списка. Многие предпочитают отобразить эти команды на горячие клавиши, как описывается во врезке «Назначение горячих клавиш для обхода списков в Vim» ниже.

Каждый пункт списка, выводимого командой `:ls`, начинается с его порядкового номера, который присваивается каждому созданному буферу автоматически. Мы можем перейти к буферу непосредственно, выполнив команду `:buffer N` (см. `:h :b` ⓘ http://vimdoc.sourceforge.net/htmldoc/windows.html#:b). Можно также использовать более понятную форму команды `:buffer {bufname}`. В имени `{bufname}` можно указать лишь первые символы имени, уникально идентифицирующие буфер. Кроме того, после ввода строки, которой соответствует более одного элемента в списке буферов, можно воспользоваться функцией автодополнения для выбора нужного буфера (рецепт 32 в главе 5).

Команда `:bufdo` дает возможность применить команду Ex ко всем буферам, перечисляемым командой `:ls` (`:h :bufdo` ⓘ http://vimdoc.sourceforge.net/htmldoc/windows.html#:bufdo). Однако я предпочитаю пользоваться командой `:argdo`, с которой мы познакомимся в рецепте 39 ниже.

Назначение горячих клавиш для обхода списков в Vim

Кому-то ввод команд `:bn` и `:bp` может показаться слишком утомительным. Чтобы немного ускорить навигацию по буферам, я использую следующие назначения горячих клавиш из расширения unimpaired.vim[1] Тима Поупа (Tim Pope):

```
nnoremap <silent> [b :bprevious<CR>
nnoremap <silent> ]b :bnext<CR>
nnoremap <silent> [B :bfirst<CR>
nnoremap <silent> ]B :blast<CR>
```

[1] https://github.com/tpope/vim-unimpaired

Клавиши `[` и `]` уже используются редактором Vim для целой группы похожих команд (см. `:h [` http://vimdoc.sourceforge.net/htmldoc/index.html#[), поэтому эти назначения достаточно естественно вписываются в их круг. Расширение unimpaired.vim включает похожие назначения горячих клавиш для прокрутки списка аргументов (`[a` и `]a`), результатов (`[q` и `]q`), местоположений (`[l` и `]l`) и тегов (`[t` и `]t`). Попробуйте их.

Удаление буферов

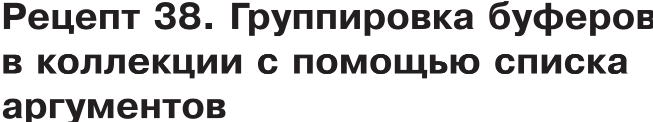
Каждый раз, когда открывается файл, Vim создает новый буфер. С некоторыми способами открытия файлов мы познакомимся в главе 7 «Открытие файлов и сохранение их на диске». Если вам потребуется удалить буфер, это можно сделать с помощью команды `:bdelete`. Она имеет следующие разновидности:

```
:bdelete N1 N2 N3
:N,M bdelete
```

Имейте в виду, что операция удаления буфера никак не сказывается на связанном с ним файле. Она просто удаляет представление файла в памяти. Если потребуется удалить буферы с 5 по 10 включительно, это можно сделать, выполнив команду `:5,10bd`. Но если при этом потребуется оставить буфер с номером 8, это можно сделать командой `:bd 5 6 7 9 10`.

Номера присваиваются буферам автоматически, и Vim не предусматривает способов их изменения вручную. Поэтому если вам потребуется удалить один или более буферов, сначала определите их номера. Данная процедура требует достаточно много времени. Если у меня нет веских причин удалить буфер, я обычно оставляю его как есть. Как результат команда `:ls` позволяет получить список всех файлов, открытых в ходе сеанса работы с редактором.

Встроенные средства Vim управления списком буферов испытывают определенный недостаток гибкости. С их помощью нельзя изменить порядок следования буферов по своему вкусу. Для этих целей лучше использовать разбиение рабочего пространства на окна, вкладки или список аргументов. Как это делается, я расскажу в следующих нескольких рецептах.



Рецепт 38. Группировка буферов в коллекции с помощью списка аргументов

Список аргументов в редакторе Vim обладает развитыми средствами управления и может пригодиться для группировки файлов с целью упрощения навигации по ним. К каждому элементу списка аргументов можно применять команды Ex, используя для этого команду `:argdo`.

Начнем с того, что откроем в Vim несколько файлов:

```
⇒ $ cd code/files/letters
⇒ $ vim *.txt
  5 files to edit
```

В рецепте 37 выше был показано, как получить список буферов с помощью команды `:ls`. А теперь давайте исследуем список аргументов:

```
⇒ :args
  [a.txt] b.txt c.txt. d.txt e.txt
```

Список аргументов представляет список файлов, перечисленных при выполнении команды `vim`. В нашем случае мы указали единственный аргумент, `*.txt`, но командная оболочка развернула групповой символ * и фактически передала команде `vim` пять файлов, которые можно наблюдать в списке аргументов. Символы `[]` указывают, какой из файлов в списке аргументов является активным.

В сравнении со списком буферов, возвращаемым командой `:ls`, вывод команды `:args` выглядит беднее. Наверное, никто не удивится, узнав, что список аргументов был унаследован от редактора `vi`, а список буферов – это расширение, привнесенное редактором Vim. Но дайте списку аргументов шанс, и вы увидите, что он является отличным дополнением к списку буферов.

Функциональность списка аргументов, как и многих других особенностей, в Vim была существенно расширена, хотя оригинальное название осталось. Мы в любой момент можем изменить содержимое списка аргументов, а это означает, что вывод команды `:args` не обязательно отражает список, переданный команде `vim` в момент запуска редактора. Не следует воспринимать это имя буквально! (См. также врезку «Команды :compiler и :make служат не только для разработки программ на компилируемых языках» в главе 17.)

Заполнение списка аргументов

Когда команда Ex `:args` вызывается без аргументов, она выводит содержимое списка аргументов. Используя следующую форму этой команды, можно изменять содержимое списка аргументов (`:h :args_f` ⓘ http://vimdoc.sourceforge.net/htmldoc/editing.html#:args_f):

```
:args {arglist}
```

Аргумент `{arglist}` может включать имена файлов, групповые символы и даже вывод команды оболочки. Для демонстрации воспользуемся каталогом *files/mvc*, который можно найти в загружаемом пакете исходных файлов для этой книги. Если вы желаете следовать за примерами в этом рецепте, перейдите в указанный каталог и запустите Vim:

```
⇛ $ cd code/files/mvc
⇛ $ vim
```

Увидеть структуру упоминаемого здесь дерева каталогов можно в рецепте 41 (глава 7).

Определение файлов по именам

Самый простой способ заполнить список аргументов – перечислить имена файлов, как показано ниже:

```
⇛ :args index.html app.js
⇛ :args
  [index.html] app.js
```

Этот прием удобно использовать, только когда необходимо открыть небольшое количество буферов. Его преимущество состоит в том, что позволяет явно определить порядок следования файлов, но выполнять эту операцию вручную может быть утомительно. При необходимости добавить в список аргументов много файлов можно воспользоваться групповыми символами.

Определение шаблонов имен файлов

Для определения шаблонов имен файлов и каталогов можно использовать групповые символы. Символ * соответствует нулю

или более символам, но только в пределах указанного каталога (`:h wildcard` ⓘ http://vimdoc.sourceforge.net/htmldoc/editing.html#wildcard). Групповой символ ** также соответствует нулю или более символам, но он позволяет углубляться во вложенные подкаталоги (`:h starstar-wildcard` ⓘ http://vimdoc.sourceforge.net/htmldoc/editing.html#starstar-wildcard).

Допускается объединять эти групповые символы с фрагментами имен файлов и каталогов для составления шаблонов, соответствующих множеству файлов, представляющих для нас интерес. В таблице ниже приводится список некоторых (но не всех) файлов в каталоге *files/mvc*, имена которых соответствуют указанным шаблонам.

Шаблон	Файлы, соответствующие шаблону
`:args *.*`	`index.html` `app.js`

`:args **/*.js`	`app.js`
	`lib/framework.js`
	`app/controllers/Mailer.js`
	`...`

`:args **/*.*`	`app.js`
	`index.js`
	`lib/framework.js`
	`lib/theme.css`
	`app/controllers/Mailer.js`
	`...`

Так же как в `{arglist}` можно указать несколько имен файлов, можно указать и несколько шаблонов. Если потребуется определить список аргументов, содержащий только файлы *.js* и *.css*, можно воспользоваться следующими шаблонами:

```
⇒ :args **/*.js **/*.css
```

Определение файлов с помощью обратных кавычек

В процессе работы над этой книгой мне иногда требовалось заполнять список аргументов главами, следующими в том же порядке, в каком они располагаются в оглавлении. Для этой цели я создал простой текстовый файл, содержащий по одному имени файла в каждой строке. Ниже приводится выдержка из этого файла:

files/.chapters
http://media.pragprog.com/titles/dnvim/code/files/.chapters

```
the_vim_way.pml
normal_mode.pml
insert_mode.pml
visual_mode.pml
```

Заполнить список аргументов из этого файла можно следующей командой:

```
⇒ :args `cat .chapters`
```

Текст, находящийся в обратных кавычках, Vim выполнит в командной оболочке, а вывод передаст как аргумент команде `:args`. Здесь для получения содержимого файла *.chapters* была использована команда `cat`, но точно так же можно использовать любую другую команду, доступную в командной оболочке. Однако эта возможность доступна не во всех системах. Дополнительные подробности см. в справке `:h backtick-expansion` ⓘ http://vimdoc.sourceforge.net/htmldoc/editing.html#backtick-expansion.

Использование списка аргументов

Список аргументов более прост в управлении, чем список буферов, что делает его идеальным средством группировки буферов в коллекции. Единственной командой `:args {arglist}` можно очистить и тут же заполнить список аргументов новыми элементами. Команды `:next` и `:prev` обеспечивают возможность навигации по списку аргументов. Команда `:argdo` позволяет применить одну и ту же команду к каждому буферу в коллекции.

Список буферов вызывает у меня ассоциацию с моим рабочим столом: на нем всегда жуткий беспорядок. Список аргументов напоминает отдельное рабочее место, которое я всегда держу в чистоте. Некоторые примеры использования списка аргументов можно найти в рецепте 36 (глава 5) и в рецепте 70 (глава 11).

Рецепт 39. Управление скрытыми файлами

Когда содержимое буфера изменяется, Vim прикладывает дополнительные усилия, чтобы предотвратить случайный выход из ре-

дактора без сохранения изменений. В этом рецепте вы узнаете, как обрабатывать скрытые буферы при завершении Vim.

Выполните следующие команды в командной оболочке, чтобы запустить Vim:

```
⇒ $ cd code/files
⇒ $ ls
  a.txt      b.txt
⇒ $ vim *.txt
  2 files to edit
```

Измените содержимое буфера `a.txt`: просто выполните команду `Go`, чтобы добавить пустую строку в конец буфера. Не сохраняя изменений, заглянем в список буферов:

```
⇒ :ls
  1 %a + "a.txt"          line 1
  2      "b.txt"          line 0
```

Буфер, представляющий файл *a.txt*, был помечен знаком `+`, указывающим, что буфер содержит несохраненные изменения. Если теперь сохранить буфер, его содержимое будет записано в файл на диске и знак `+` исчезнет. Но давайте пока не будем сохранять его. Вместо этого попробуем переключиться на следующий буфер:

```
⇒ :bnext
  E37: No write since last change (add ! to override)
```

Vim вывел сообщение об ошибке, указав, что текущий буфер содержит несохраненные изменения. Попробуем последовать совету в круглых скобках[1] и добавим восклицательный знак в конце:

```
⇒ :bnext!
⇒ :ls
  1 #h + "a.txt"            line 1
  2 %a   "b.txt"            line 1
```

Восклицательный знак заставил редактор переключить буферы, несмотря на то что текущий буфер содержит несохраненные изменения. Если теперь выполнить команду `:ls`, буфер `b.txt` будет помечен буквой `a` как *активный*, а буфер `a.txt` – буквой `h` как *скрытый* (hidden).

[1] add ! to override (добавьте !, чтобы обойти проверку). – *Прим. перев.*

Обработка скрытых буферов при выходе из редактора

Наличие скрытых буферов никак не отражается на работе Vim. Мы так же можем открывать другие буферы, изменять их, сохранять и выполнять другие операции без каких-либо последствий – но только до момента, когда будет предпринята попытка завершить сеанс работы с редактором. В этот момент Vim напомнит, что имеются несохраненные изменения в одном из буферов:

```
⇒ :quit
  E37: No write since last change (add ! to override)
  E162: No write since last change for buffer "a.txt"
```

При этом Vim загрузит первый скрытый буфер с изменениями в текущее окно, благодаря чему мы сможем решить, что с ним делать. Если потребуется сохранить изменения, можно выполнить команду `:write`. Или, если потребуется отменить изменения, можно выполнить команду `:edit!`, которая прочитает файл с диска и затрет содержимое буфера. Приведя содержимое буфера в соответствие с содержимым файла на диске, можно попытаться еще раз выполнить команду `:quit`.

При наличии другого несохраненного буфера Vim активирует следующий такой буфер, и так каждый раз при попытке выполнить команду `:quit`. Напомню, что мы можем использовать команду `:write` или `:edit!`, чтобы сохранить или отменить изменения. Этот цикл будет продолжаться, пока не будет принято решение по каждому скрытому буферу с имеющимися изменениями. Когда в редакторе не останется окон и скрытых буферов с изменениями, команда `:q` закроет редактор Vim.

Чтобы закрыть Vim, невзирая на наличие изменений, можно выполнить команду `:qall!`. Если появится желание сохранить все буферы сразу, не просматривая их один за другим, можно воспользоваться командой `:wall`. Все эти возможности перечислены в табл. 6.1 ниже.

Включите параметр настройки hidden перед вызовом команды :argdo или :bufdo

По умолчанию редактор Vim не позволяет покидать измененные буферы с помощью команд `:next`, `:bnext`, `:cnext` и других, если не указать в конце команды восклицательный знак. Vim будет ворчать

на нас, выводя сообщение об ошибке: «No write since last change»[1]. В большинстве случаев это сообщение является полезным напоминанием. Но в одной ситуации оно превращается в неприятность.

Таблица 6.1. Способы обработки скрытых буферов при завершении работы с редактором

Команда	Действие
`:w[rite]`	Запишет содержимое буфера на диск
`:e[dit]!`	Прочитает файл с диска в буфер (то есть затрет все изменения)
`:qa[ll]!`	Закроет все окна, отменит все изменения без лишних предупреждений
`:wa[ll]`	Запишет содержимое всех измененных буферов на диск

Рассмотрим поближе команды `:argdo` и `:bufdo`. Команда `:argdo {cmd}` действует примерно следующим образом:

```
⇒ :first
⇒ :{cmd}
⇒ :next
⇒ :{cmd}
```

и так далее.

Если выбранная команда `{cmd}` изменит содержимое первого буфера, следующая за ней команда `:next` потерпит неудачу. Vim не позволит перейти ко второму пункту в списке аргументов, пока не будут сохранены изменения в первом. Это так неудобно! Если в настройках включен параметр `hidden` (см. `:h 'hidden'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'hidden'), мы сможем использовать команды `:next`, `:bnext`, `:cnext` (и другие) без завершающего восклицательного знака. Если активный буфер окажется изменен, Vim автоматически скроет его и перейдет к следующему. Параметр настройки `hidden` позволяет использовать команды `:argdo` и `:bufdo` для изменения буферов в коллекции единственной командой.

После выполнения команды `:argdo {cmd}` может потребоваться сохранить изменения, внесенные в каждый элемент списка аргументов. Мы могли бы сохранить буферы по одному, выполнив `:first` и затем `:wn`, получив попутную возможность просмотреть каждый файл. Или, если нет никаких сомнений, что все в порядке, можно выполнить команду `:argdo write` (или `:wall`), чтобы сохранить сразу все буферы.

---

[1] Изменения не сохранены. – *Прим. перев.*

Рецепт 40. Деление рабочего пространства на окна

Редактор Vim позволяет одновременно видеть сразу несколько буферов, предоставляя возможность разбивать рабочее пространство на окна.

В терминологии Vim *окно* – это ограниченная область рабочего пространства, в которой отображается содержимое буфера (`:h window` http://vimdoc.sourceforge.net/htmldoc/windows.html#window). Мы можем открыть множество окон, каждое из которых будет отображать содержимое одного и того же буфера, или загрузить в окна разные буферы. Система управления окнами в редакторе Vim обладает значительной гибкостью, разрешая компоновать рабочее пространство в соответствии с нашими требованиями и предпочтениями.

Создание окон

В момент запуска Vim открывает единственное окно. Мы можем разделить это окно по горизонтали, выполнив команду `<C-w>s`, которая создаст два окна одинаковой высоты, или по вертикали, выполнив команду `<C-w>v`, которая создаст два окна одинаковой ширины. Эти команды можно повторять столько раз, сколько потребуется, разбивая рабочее пространство снова и снова.

На рис. 6.1 изображен один из возможных результатов. В каждом случае заштрихованный прямоугольник представляет активное окно.

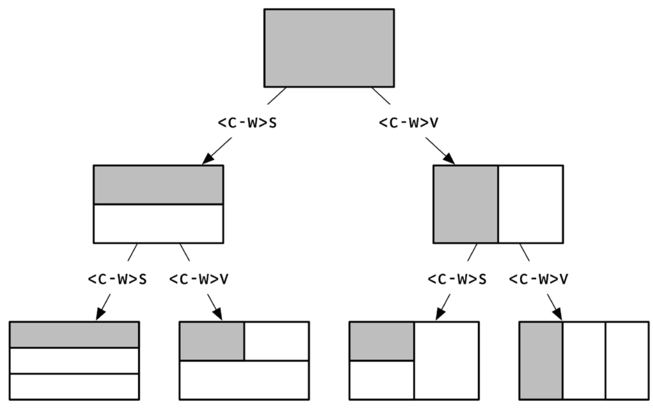


Рис. 6.1. Разные варианты разбиения рабочего пространства на окна

Каждый раз, выполняя команду `<C-w>s` или `<C-w>v`, мы получаем два окна, отображающие содержимое буфера, который отображался в оригинальном окне перед его делением. Возможность отображения одного и того же буфера в отдельных окнах может быть очень полезной, особенно при работе с длинными файлами. Например, в одном окне можно вывести один фрагмент буфера, который желательно иметь перед глазами, внося изменения в другом фрагменте этого же буфера, отображаемого в другом окне.

Для загрузки другого файла в активное окно можно использовать команду `:edit`. Если последовательно выполнить команды `<C-w>s` и `:edit {filename}`, рабочее пространство будет разделено пополам – в одном окне будет отображаться содержимое оригинального буфера, а в другое загрузится содержимое другого файла, указанного в команде `:edit {filename}`. Можно также использовать команду `:split {filename}`, объединяющую эти две команды. Ниже перечислены способы разбиения рабочего пространства на окна:

Команда	Действие
`<C-w>s`	Разобьет текущее окно по горизонтали, отобразив во втором окне тот же буфер
`<C-w>v`	Разобьет текущее окно по вертикали, отобразив во втором окне тот же буфер
`:sp[lit] {file}`	Разобьет текущее окно по горизонтали и загрузит файл `{file}` в новое окно
`:vsp[lit] {file}`	Разобьет текущее окно по вертикали и загрузит файл `{file}` в новое окно

Переключение фокуса ввода между окнами

Vim поддерживает несколько команд переключения фокуса ввода между окнами. В таблице ниже перечислены наиболее примечательные из них (полный список можно найти в справке `:h window-move-cursor` ⓘ http://vimdoc.sourceforge.net/htmldoc/windows.html#window-move-cursor):

Команда	Действие
`<C-w>w`	Передает фокус ввода между окнами в цикле
`<C-w>h`	Передает фокус ввода окну слева
`<C-w>j`	Передает фокус ввода окну ниже
`<C-w>k`	Передает фокус ввода окну выше
`<C-w>l`	Передает фокус ввода окну справа

Фактически команда `<C-w><C-w>` делает то же, что и команда `<C-w>w`. Это означает, что можно нажать клавишу `<Ctrl>` и, удерживая ее, нажать `ww` (или `wj`, или любую другую комбинацию из таблицы выше), чтобы сменить активное окно. Гораздо проще ввести команду `<C-w><C-w>`, чем `<C-w>w`, даже при том, что на книжной странице она выглядит длиннее. Если вы активно используете возможность разбиения рабочего пространства на окна, подумайте о переназначении этих команд на другие клавиши, более удобные для вас.

Если ваш терминал поддерживает операции с мышью или если вы пользуетесь версией редактора GVim, вы можете активировать окна щелчком мышью на них. Если у вас что-то не получится – проверьте параметр настройки `mouse` (`:h 'mouse'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'mouse').

Закрытие окон

Если потребуется уменьшить количество открытых окон, можно пойти одним из двух путей: воспользоваться командой `:close`, чтобы закрыть активное окно, или закрыть все окна, кроме активного, командой `:only`. В следующей таблице перечислены команды и соответствующие им комбинации клавиш, которые можно использовать в командном режиме:

Команда Ex	Команда командного режима	Действие
`clo[se]`	`<C-w>c`	Закроет активное окно
`:on[ly]`	`<C-w>o`	Закроет все окна, кроме активного

Изменение размеров и переупорядочение окон

Vim поддерживает несколько комбинаций клавиш для изменения размеров окна. Полный список можно найти в справке `:h window-resize` ⓘ http://vimdoc.sourceforge.net/htmldoc/windows.html#window-resize. В следующей таблице перечислены наиболее удобные из них:

<table>
<tr><th>Команда</th><th>Действие</th></tr>
<tr><td><C-w>=</td><td>Выровнит ширину и высоту всех окон</td></tr>
<tr><td><C-w>_</td><td>Увеличит высоту активного окна до предела</td></tr>
<tr><td><C-w>|</td><td>Увеличит ширину активного окна до предела</td></tr>
<tr><td>[N]<C-w>_</td><td>Установит высоту активного окна равной [N] строкам</td></tr>
<tr><td>[N]<C-w>|</td><td>Установит ширину активного окна равной [N] столбцам</td></tr>
</table>

Изменение размеров окон – одна из немногих операций, которые я предпочитаю выполнять мышью. Делается это очень просто: ухватите левой кнопкой мыши границу, разделяющую два окна, и перемещайте, пока окна не приобретут желаемый размер, а затем отпустите мышь. Такой способ можно использовать, только если терминал поддерживает мышь, или если вы пользуетесь редактором GVim.

Кроме того, Vim поддерживает команды переупорядочения окон, но, вместо того чтобы обсуждать, как это делается, я предлагаю взглянуть на скринкасты на сайте Vimcasts.org, наглядно демонстрирующие эту особенность[1]. Дополнительную информацию можно также найти в справке `:h window-moving` ⓘ http://vimdoc.sourceforge.net/htmldoc/windows.html#window-moving.

Рецепт 41. Организация размещения окон с помощью вкладок

Интерфейс вкладок в Vim отличается от используемого во многих других текстовых редакторах. Поддержка вкладок позволяет организовать окна в коллекцию рабочих пространств.

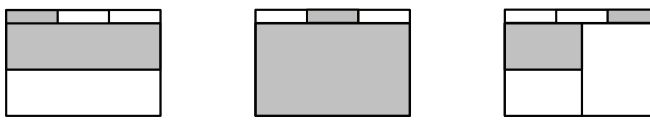


Рис. 6.2. Организация размещения окон с помощью вкладок

В Vim *вкладка* – это контейнер, способный хранить коллекцию окон (`:h tabpage` ⓘ http://vimdoc.sourceforge.net/htmldoc/tabpage.html#tabpage). Если вы привыкли пользоваться другими редакторами, интерфейс вкладок в редакторе Vim, на первый взгляд, может показаться странным. Давайте начнем с того, что посмотрим, как действуют вкладки в большинстве других редакторов.

Классический графический интерфейс редактирования текста включает главное рабочее пространство, где выполняется правка текста, и боковую панель с деревом каталогов текущего проекта. Если щелкнуть на файле в боковой панели, редактор откроет новую

[1] http://vimcasts.org/e/7

вкладку в главной рабочей области и отобразит в ней содержимое выбранного файла. Для каждого открываемого файла создается новая вкладка. Можно сказать, что в этой модели вкладки представляют множество открытых файлов.

Когда с помощью команды `:edit` в Vim открывается новый файл, для него не создается новая вкладка. Вместо этого создается новый буфер, куда загружается содержимое файла, и он отображается в текущем окне. Vim запоминает открытые файлы в списке буферов, как описывалось в рецепте 37 выше.

Вкладки в Vim не связаны с буферами отношением «один к одному». Они являются своеобразными контейнерами, способными хранить коллекции окон. На рис. 6.2 изображено рабочее пространство с тремя вкладками, каждая из которых содержит одно или более окон. В каждом случае заштрихованные прямоугольники представляют активное окно и активную вкладку.

Вкладки доступны как в GVim, так и в терминальной версии Vim. GVim отрисовывает вкладки как часть пользовательского интерфейса, придавая им внешний вид, напоминающий вкладки в веб-браузере или в любой другой программе, предоставляющий интерфейс с вкладками. При выполнении в терминале Vim отрисовывает полосу с вкладками как часть текстового интерфейса. Кроме чисто внешних различий, во всем остальном вкладки в графическом и текстовом интерфейсах идентичны.

Как пользоваться вкладками

Вкладки в Vim можно использовать для логического распределения файлов по разным рабочим пространствам. Вкладки больше напоминают виртуальные рабочие столы в Linux, чем привычные вкладки в других текстовых редакторах.

Представьте, что вы работаете над проектом, разбив рабочее пространство на пять окон. Внезапно поступает какое-то срочное задание, и вам необходимо переключиться на другую работу. Вместо того чтобы открывать новые файлы в текущей вкладке, что испортило бы уютную организацию рабочего пространства, можно создать новую вкладку и выполнить срочную работу в ней. По окончании, чтобы вернуться к отложенным делам, достаточно будет просто переключиться в первоначальную вкладку, где все наши окна сохранятся в том состоянии, в каком мы их оставили.

Команда `:lcd {path}` дает возможность установить текущий локальный рабочий каталог для текущего окна. Если создать новую

вкладку и выполнить в ней команду `:lcd`, чтобы перейти в другой каталог, мы сможем комфортно работать с разными проектами во вкладках. Обратите внимание, что команда `:lcd` применяется к *текущему окну*, а не к текущей вкладке. При наличии во вкладке двух или более окон можно определить текущий рабочий каталог для всех их одной командой `:windo lcd {path}`. Посмотрите девятый скринкаст на сайте Vimcasts.org[1].

Открытие и закрытие вкладок

Новую вкладку можно открыть командой `:tabedit {filename}`. Если опустить аргумент `{filename}`, Vim создаст новую вкладку с пустым буфером.

С другой стороны, если текущая вкладка содержит несколько окон, можно выполнить команду `<C-w>T`, которая переместит текущее окно в новую вкладку (см. `:h CTRL-W_T` ⓘ http://vimdoc.sourceforge.net/htmldoc/windows.html#CTRL-W_T).

Если активная вкладка содержит единственное окно, команда `:close` закроет окно и вместе с ним вкладку. Можно также воспользоваться командой `:tabclose`, которая закроет текущую вкладку независимо от количества окон в ней. Наконец, командой `:tabonly` можно закрыть все вкладки, кроме текущей.

Команда	Действие
`:tabe[dit] {filename}`	Откроет `{filename}` в новой вкладке
`<C-w>T`	Переместит текущее окно в новую вкладку
`:tabc[lose]`	Закроет текущую вкладку и все окна в ней
`:tabo[nly]`	Закроет все вкладки, кроме текущей

Переключение между вкладками

Вкладкам присваиваются номера, начиная с 1. Мы можем переключаться между вкладками с помощью команды `{N}gt`, которую можно запомнить по мнемонике *goto tab{N}* (*перейти к вкладке {N}*). Когда эта команда предваряется числом, Vim переходит к указанной вкладке, но если номер отсутствует, Vim переходит к следующей вкладке. Команда `gT` выполняет переход в обратном направлении.

---

[1] http://vimcasts.org/e/9

Команда Ex	Команда командного режима	Действие
:tabn[ext] {N}	{N}gt	Переключится на вкладку {N}
:tabn[ext]	gt	Переключится на следующую вкладку
:tabp[revious]	gT	Переключится на предыдущую вкладку

Переупорядочение вкладок

С помощью команды Ex `:tabmove [N]` можно переупорядочить вкладки. Когда в аргументе `[N]` передается число 0, текущая вкладка переносится в начало, если опустить аргумент `[N]`, текущая вкладка переместится в конец. Если терминал поддерживает операции с мышью или если вы пользуетесь редактором GVim, переупорядочивать вкладки можно с помощью мыши.

Глава 7. Открытие файлов и их сохранение на диск

Редактор Vim поддерживает несколько способов открытия файлов. В рецепте 42 ниже мы познакомимся с командой `:edit`, которую можно использовать для открытия любых файлов по именам.

При работе с файлами, находящимися двумя или более уровнями ниже в дереве каталогов проекта, может быть утомительным указывать полные пути к этим файлам. В рецепте 43 рассказывается, как настроить параметр `path`, чтобы обеспечить возможность автоматического использования команды `:find`. Это избавит нас от необходимости указывать полные пути к файлам и упростит ввод имен файлов.

Расширение `netrw`, распространяемое в составе Vim, дает возможность исследовать содержимое дерева каталогов. Как им пользоваться, мы узнаем в рецепте 44.

Команда `:write` позволяет сохранить содержимое буфера на диск. Пользоваться ей достаточно просто, но она может вызывать затруднения при попытке сохранить буфер в несуществующий каталог или при отсутствии права на запись в файл. В рецептах 45 и 46 я расскажу, как справиться с этими ситуациями.

Рецепт 42. Открытие файла по его имени с использованием команды :edit

Команда `:edit` позволяет открывать файлы из редактора Vim, указывая полное имя файла, содержащее абсолютный или относительный путь к нему. Здесь мы также узнаем, как указать путь относительно активного буфера.

Для демонстрации будем использовать каталог *files/mvc* из пакета загружаемых файлов к книге. Он содержит следующие каталоги и файлы:

```
app.js
index.html
app/
    controllers/
        Mailer.js
        Main.js
        Navigation.js
    models/
        User.js
    views/
        Home.js
        Main.js
        Settings.js
    lib/
        framework.js
        theme.css
```

В командной оболочке выполните переход в каталог *files/mvc* и затем запустите Vim:

```
⇒ $ cd code/files/mvc
⇒ $ vim index.html
```

Как открыть файл, указав путь относительно текущего рабочего каталога

В редакторе Vim, так же как в bash и других командных оболочках, существует понятие *текущего рабочего каталога*. Когда Vim запускается, его текущим рабочим каталогом становится текущий рабочий каталог оболочки. Проверить это можно с помощью команды Ex `:pwd`, название которой (как и в bash) происходит от англ. «print working directory» (вывести текущий рабочий каталог):

```
⇒ :pwd
  /Users/drew/practical-vim/code/files/mvc
```

Команда `:edit {file}` может принимать пути к файлам, откладываемые относительно текущего рабочего каталога. Например, чтобы открыть файл *lib/framework.js*, достаточно выполнить следующую команду:

```
⇒ :edit lib/framework.js
```

А открыть файл *app/controllers/Navigation.js* можно командой:

```
⇒ :edit app/controllers/Navigation.js
```

В процессе набора пути можно использовать функцию автодополнения (см. рецепт 32 в главе 5), вызываемую нажатием на клавишу табуляции. То есть если потребуется открыть файл *Navigation.js*, можно просто ввести `:edit a<Tab>c<Tab>N<Tab>`.

Как открыть файл, указав путь относительно активного каталога

Допустим, что мы правим файл *app/controllers/Navigation.js* и решили открыть файл *Main.js* в том же каталоге. Мы могли бы указать к нему путь, лежащий через текущий рабочий каталог, но это решение кажется избыточным. Файл, который требуется открыть, находится в том же каталоге, что и файл, открытый в активном буфере. Было бы идеально, если бы имелась возможность использовать контекст активного буфера в качестве отправной точки. Попробуйте выполнить такую команду:

```
⇒ :edit %<Tab>
```

Символ `%` в данном случае представляет путь к файлу в активном буфере (см. `:h cmdline-special` ⓘ http://vimdoc.sourceforge.net/htmldoc/cmdline.html#cmdline-special). Нажатие на клавишу `<Tab>` расширит путь до полного, абсолютного пути к файлу в активном буфере. Это не то, что нам хотелось бы, но достаточно близко. Теперь попробуйте такую команду:

```
⇒ :edit %:h<Tab>
```

Модификатор удалит имя файла, оставив путь к нему в неприкосновенности (см. `:h ::h` ⓘ http://vimdoc.sourceforge.net/htmldoc/cmdline.html#::h). В нашем случае после ввода `%:h<Tab>` в командной строке появится полный путь к каталогу с текущим файлом:

```
⇒ :edit app/controllers/
```

Теперь можно ввести имя файла *Main.js* (или снова воспользоваться функцией автодополнения), и Vim откроет указанный файл. Всего нам потребовалось нажать следующие клавиши:

```
⇒ :edit %:h<Tab>M<Tab><Tab>
```

Вообще говоря, комбинация `%:h` не так полезна, чтобы отображать ее на горячие клавиши. Более интересное решение описывается во врезке «Простой способ извлечения имени каталога активного файла» ниже.

Простой способ извлечения имени каталога активного файла

Попробуйте добавить следующую строку в свой файл *vimrc*:

```
cnoremap <expr> %% getcmdtype() == ':' ? expand('%:h').'/' : '%%'
```

Теперь, когда вы будете вводить последовательность `%%` в командной строке Vim, она автоматически будет замещаться путем к каталогу, где хранится активный файл, как если бы вы ввели `%:h<Tab>`. Помимо удобства при работе с командой `:edit`, эта последовательность с успехом может использоваться и в иных командах Ex, таких как `:write`, `:saveas` и `:read`.

Другие примеры использования этого отображения можно увидеть в скринкасте на сайте Vimcasts.org :edit command[1].

Рецепт 43. Открытие файла по его имени с применением команды :find

Команда `:find` позволяет открывать файлы по их именам, без необходимости указывать пути к ним. Чтобы задействовать эту возможность, сначала необходимо настроить параметр настройки `path`.

Мы всегда можем использовать команду `:edit`, чтобы открыть файл, указав полный путь к нему. Но как быть тем, кто работает с проектами, имеющими несколько уровней вложенных подкаталогов? Ввод полного пути каждый раз, когда потребуется открыть файл, может оказаться чересчур утомительным делом. В таких ситуациях на помощь к вам может прийти команда `:find`.

Подготовка

Для демонстрации будем использовать каталог *files/mvc* из загружаемого пакета файлов для этой книги. Запустите Vim в командной оболочке из каталога *files/mvc*:

---

[1] http://vimcasts.org/episodes/the-edit-command/

```
⇛ $ cd code/files/mvc
⇛ $ vim index.html
```

Давайте посмотрим, что произойдет, если попытаться выполнить команду `:find` прямо сейчас:

```
⇛ :find Main.js
  E345: Can't find file "Main.js" in path
```

В сообщении об ошибке говорится, что файл *Main.js* не найден в пути поиска. Давайте решим эту проблему.

Настройка параметра path

В параметре `path` можно указать множество каталогов, в которых команда `:find` будет выполнять поиск (см. `:h 'path'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html'path'). В нашем случае желательно упростить поиск файлов в каталогах *app/controllers* и *app/views*. Мы можем добавить эти каталоги в параметр `path`, выполнив следующую команду:

```
⇛ :set path+=app/**
```

Групповой символ ** соответствует всем подкаталогам, вложенным в каталог *app/*. Мы уже обсуждали групповые символы в разделе «Заполнение списка аргументов» (глава 6), но в контексте параметра `path` значение символов * и ** несколько отличается (см. `:h file-searching` ⓘ http://vimdoc.sourceforge.net/htmldoc/editing.html#file-searching). Групповые символы в Vim обрабатываются иначе, чем в командной оболочке.

Управление путями поиска с помощью расширения rails.vim

Расширение rails.vim Типа Поупа (Tim Pope) значительно упрощает навигацию в проектах Rails[1]. Оно автоматически настраивает параметр `path`, включая в него все каталоги, имеющиеся в проекте Rails. Это означает возможность использовать команду `:find` без дополнительной настройки параметра `path`.

Но настройка параметра `path` – не единственная особенность расширения rails.vim. Оно также предоставляет дополнительные команды,

[1] https://github.com/tpope/vim-rails

такие как :Rcontroller, :Rmodel, :Rview и другие, каждая из которых действует как специализированная версия команды :find, ограничивая область поиска соответствующим каталогом.

Используйте команду :find для поиска файлов по именам

Теперь, после настройки параметра path, мы можем открывать файлы в указанных каталогах по простым именам. Например, если потребуется открыть файл *app/controllers/Navigation.js*, достаточно выполнить команду:

```
➾ :find Navigation.js
```

Мы можем использовать клавишу <Tab> для автодополнения имен файлов, то есть мы можем получить желаемый результат, просто введя :find nav<Tab> и нажав клавишу **Enter**.

Кого-то может заинтересовать вопрос: что произойдет, если указанное имя файла окажется неуникальным. Давайте посмотрим. В нашем пакете имеются два файла *Main.js*: один – в каталоге *app/controllers*, а другой – в каталоге *app/views*.

```
➾ :find Main.js<Tab>
```

Если ввести имя *Main.js* и нажать клавишу <Tab>, Vim выведет полный путь для первого совпадения: *./app/controllers/Main.js*. Нажмите клавишу <Tab> еще раз, и Vim выведет следующее совпадение: *./app/views/Main.js*. Когда нажимается клавиша **Enter**, Vim использует полный путь, полученный в результате нажатий клавиши <Tab>, или первое найденное совпадение, если клавиша <Tab> не нажималась.

Присвоив параметру wildmode другое значение, отличное от значения по умолчанию full, можно наблюдать иное поведение функции автодополнения. Дополнительные подробности можно найти в рецепте 32 в главе 5.

Рецепт 44. Исследование файловой системы с помощью netrw

В дополнение к возможности просматривать (и править) содержимое файлов Vim также поддерживает возможность просматри-

вать содержимое каталогов. Эту возможность реализует расширение netrw, входящее в состав дистрибутива Vim.

Подготовка

Функциональность, описываемая в этом рецепте, не входит в базовый набор функций Vim, но поддерживается расширением netrw. Это расширение распространяется в составе стандартного дистрибутива Vim, поэтому вам не придется заниматься его установкой – только настроить загрузку расширений в Vim. Ниже приводятся строки, которые следует добавить в файл *vimrc*:

essential.vim
http://media.pragprog.com/titles/dnvim/code/essential.vim

```
set nocompatible
filetype plugin on
```

Встречайте: netrw – встроенный обозреватель файлов Vim

Если запустить Vim, передав ему путь к каталогу, а не к файлу, он откроет окно обозревателя файлов:

```
⇒ $ cd code/file/mvc
⇒ $ ls
   app          app.js    index.html       lib
⇒ $ vim .
```

На рис. 7.1 показано, как выглядит этот обозреватель. Это обычный буфер Vim, но вместо содержимого файла он представляет содержимое каталога.

Мы можем перемещать курсор вверх и вниз клавишами `k` и `j`. Нажатие клавиши `<CR>` открывает элемент под курсором. Если курсор оказывается на имени каталога, окно обозревателя перерисовывается, и в нем отображается содержимое этого каталога. Если курсор оказывается на имени файла, этот файл загружается в текущий буфер, заменяя обозреватель. Перейти в родительский каталог можно нажатием клавиши `-` или установив курсор на пункт .. и нажав `<CR>`.

Осуществлять навигацию по содержимому каталога можно не только с помощью клавиш `j` и `k`. Поддерживаются также все команды перемещений курсора, доступные в обычных буферах Vim.

```
" ============================================================
" Netrw Directory Listing
"   /Users/drew/code/mvc
"   Sorted by      name
"   Sort sequence: [\/]$,\<core\%(\.\d\+\)\=\>,\.h$,\.c$,\.c
"   Quick Help: <F1>:help  -:go up dir  D:delete  R:rename
" ============================================================
../
app/
lib/
app.js
index.html
~
~
/Users/drew/code/mvc [RO]                    8,1           All
```

Рис. 7.1. netrw – встроенный обозреватель файлов

Например, чтобы открыть файл *index.html*, можно выполнить поиск /html<CR>, в результате которого курсор окажется на имени требуемого файла.

Открытие обозревателя файлов

Открыть окно обозревателя файлов можно командой :edit {path}, передав в аргументе {path} имя каталога (вместо имени файла). Символ точки (.) соответствует текущему рабочему каталогу, поэтому, если ввести команду :edit ., в окне обозревателя отобразится содержимое текущего рабочего каталога.

Если потребуется открыть в обозревателе каталог, где находится текущий файл, это можно сделать с помощью команды :edit %:h (см. врезку «Простой способ извлечения имени каталога активного файла» выше). Но расширение netrw поддерживает более удобный способ сделать то же самое – с помощью команды :Explore (см. :h :Explore ⓘ http://vimdoc.sourceforge.net/htmldoc/pi_netrw.html#:Explore).

Обе эти команды имеют сокращенные формы. Вместо команды :edit . можно использовать команду :e. – без пробела перед точкой. А команду :Explore можно сократить до :E. В следующей таблице перечислены длинные и короткие формы команд:

Команда Ex	Короткая форма	Действие
:edit .	:e.	Откроет в обозревателе файлов текущий рабочий каталог
:Explore	:E	Откроет в обозревателе файлов каталог текущего активного буфера

В дополнение к команде `:Explore` расширение netrw поддерживает также команды `:Sexplore` и `:Vexplore`, открывающие обозреватель в новом окне, разбив текущее по горизонтали или по вертикали соответственно.

Работа с окнами

Классические графические интерфейсы текстовых редакторов отображают обозреватель файлов в боковой панели, который иногда называют *обозревателем проекта*. Если вы привыкли пользоваться подобными интерфейсами, тогда вам может показаться странным, что команды `:E` и `:e.` открывают обозреватель файлов в текущем окне, замещая прежнее его содержимое. Однако на то есть веская причина: подобное поведение отлично подходит при разбиении рабочей области на окна.

Взгляните на первый кадр, на рис. 7.2.

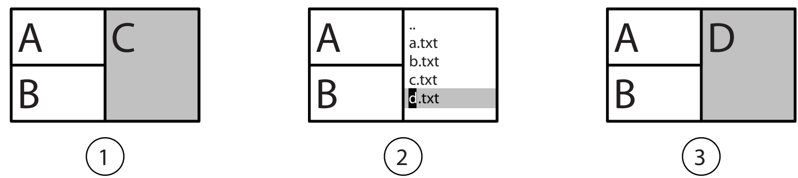


Рис. 7.2. Пример использования обозревателя файлов при работе с несколькими окнами

Здесь мы видим три окна, каждое из которых отображает свой буфер. Представьте на мгновение, что обозреватель проекта, содержащий обозреватель файлов, «прикручен» к интерфейсу Vim в виде боковой панели. Если нам потребуется открыть файл щелчком мыши на его имени в обозревателе проекта, где он должен быть открыт?

Активным является окно с меткой *C* (заштриховано на рис. 7.2), поэтому кажется вполне естественным открыть файл в нем. Но взаимоотношения между обозревателем проекта и активным окном неочевидны. Вы легко можете забыть, какое окно является активным, что может привести к неожиданным результатам при выборе файла в обозревателе проекта – он может быть открыт в неожиданном для вас окне.

Теперь удалим воображаемый обозреватель проекта и познакомимся с тем, как в действительности работает Vim. Если выполнить

команду `:Explore`, текущее содержимое активного окна будет замещено обозревателем файлов, как показано на втором кадре, на рис. 7.2. Вне всяких сомнений, после выбора файла его содержимое будет загружено в то же самое окно.

Представьте, что каждое окно является своеобразной игральной картой. С одной стороны карты отображается содержимое файла, а с другой – обозреватель файлов. Когда выполняется команда `:Explore`, карта переворачивается, и на экране отображается сторона с обозревателем файлов (кадр 2 на рис. 7.2). После выбора файла мы нажимаем `<CR>`, и карта снова переворачивается, на этот раз отображая содержимое выбранного файла (кадр 3 на рис. 7.2). Если после вызова обозревателя файлов мы решим переключиться обратно в буфер, не открывая нового файла, это можно будет сделать с помощью команды `<C-^>`.

С определенными допущениями можно сказать, что окна в редакторе Vim имеют два режима: один – для работы с файлами, другой – для работы с каталогами. Такая модель работы отлично согласуется с оконным интерфейсом Vim, тогда как отдельное понятие обозревателя проекта – нет.

Дополнительные возможности netrw

Расширение netrw позволяет не только обозревать файловую систему. С его помощью можно также создавать новые файлы (`:h etrw-%` http://vimdoc.sourceforge.net/htmldoc/pi_netrw.html#netrw-%) или каталоги (`:h netrw-d` http://vimdoc.sourceforge.net/htmldoc/pi_netrw.html#netrw-d), переименовывать существующие (`:h netrw-rename` http://vimdoc.sourceforge.net/htmldoc/pi_netrw.html#netrw-rename) и удалять их (`:h netrw-del` http://vimdoc.sourceforge.net/htmldoc/pi_netrw.html#netrw-del$). Наглядную демонстрацию выполнения этих операций можно найти на сайте Vimcasts.org[1].

Мы еще не коснулись самой убийственной особенности, давшей название расширению: netrw позволяет читать и писать файлы через сеть. Расширение поддерживает множество протоколов, включая `scp`, `ftp`, `curl` и `wget`. За более подробной информацией обращайтесь к справке `:h netrw-ref` http://vimdoc.sourceforge.net/htmldoc/pi_netrw.html#netrw-ref.

---

[1] http://vimcasts.org/e/15

Рецепт 45. Сохранение файлов в несуществующие каталоги

Редактор Vim с радостью позволяет править файлы, пути к которым включают отсутствующие каталоги. Но только пока не будет предпринята записать буфер в такой файл. В этом рецепте рассказывается, как решить данную проблему.

Для открытия уже существующих файлов чаще других используется команда `:edit {file}`. Но если указать имя несуществующего файла, Vim создаст новый пустой буфер. Если нажать комбинацию <C-g>, буфер будет помечен как «new file» («новый файл»). Команда <C-g> выводит имя и состояние текущего файла (см. `:h ctrl-G` http://vimdoc.sourceforge.net/htmldoc/editing.html#ctrl-G). Если затем выполнить команду `:write`, Vim попытается записать содержимое буфера в новый файл по указанному пути.

Если в команде `:edit {file}` был указан путь, содержащий несуществующие каталоги, редактор выведет сообщение об ошибке:

```
⇒  :edit madeup/dir/doesnotexist.yet
⇒  :write
   "madeup/dir/doesnotexist.yet" E212: Can't open file for writing
```

В данном случае каталоги *madeup/dir* не существуют в файловой системе. Vim создаст новый буфер в любом случае, который будет помечен как «new DIRECTORY» («новый КАТАЛОГ»). Но ошибка возникнет только при попытке записать буфер на диск. Мы можем исправить ситуацию, вызвав внешнюю команду `mkdir`:

```
⇒  :!mkdir -p %:h
⇒  :write
```

Флаг `-p` указывает команде `mkdir` на необходимость создавать промежуточные каталоги. Описание значений символов `%:h` приводится в разделе «Как открыть файл, указав путь относительно активного каталога» выше.

Рецепт 46. Сохранение файла с привилегиями суперпользователя

Обычно не принято запускать редактор Vim с привилегиями суперпользователя, но иногда бывает необходимо сохранить изменения в файл, доступный для записи только суперпользователю. Это

можно сделать, и не перезапуская Vim, предоставив эту задачу процессу командной оболочки, запущенной с помощью команды `sudo`.

Данный рецепт может быть неприменим к редактору GVim и точно не работает в Windows. Его можно применять в системах Unix, где редактор Vim выполняется в окне терминала, что является достаточно обычным делом, чтобы включить этот рецепт в книгу.

Продемонстрируем данный рецепт на примере файла */etc/hosts*. Владельцем этого файла является root, но мы зашли в систему как пользователь «drew», поэтому мы можем только читать этот файл:

```
⇛ $ ls -al /etc/ | grep hosts
  -rw-r--r--    1 root wheel       634 6 Apr 15:59 hosts
⇛ $ whoami
  drew
```

Откроем этот файл в Vim:

```
⇛ $ vim /etc/hosts
```

Первое, что бросается в глаза после нажатия комбинации `<C-g>`, – статус файла, Vim пометил его как «[readonly]» («[только для чтения]»).

Попробуем изменить его и посмотрим, что из этого получится. Выполним команду `Go`, чтобы добавить пустую строку в конец файла. В ответ Vim выведет сообщение: «W10: Warning: Changing a readonly file» («W10: Внимание: изменен файл, доступный только для чтения»). Считайте это сообщение дружеским напоминанием, а не ошибкой. После вывода сообщения Vim выполнит запрошенное изменение.

Vim не мешает вносить изменения в буфер, соответствующий файлу, доступному только для чтения, но он не даст нам сохранить его на диск обычным способом:

```
⇛ :write
  E45: 'readonly' option is set (add ! to override)
```

Давайте последуем совету в сообщении[1] и повторим команду, добавив восклицательный знак в конец команды (что может означать: «На этот раз я понимаю, что делаю!»):

---

[1] Установлен параметр readonly (добавьте !, чтобы обойти проверку). – *Прим. перев.*

```
⇛ :write!
  "/etc/hosts" E212: Can't open file for writing
```

Проблема в том, что у нас отсутствует право на запись в файл */etc/hosts file*. Если вы помните, владельцем файла является root, а мы запустили Vim с привилегиями пользователя drew. Решить эту проблему можно несколько странной, на первый взгляд, командой:

```
⇛ :w !sudo tee % > /dev/null
  Password:
  W12: Warning: File "hosts" has changed and the buffer was
  changed in Vim as well
  [O]k, (L)oad File, Load (A)ll, (I)gnore All:
```

Vim взаимодействует с нами в два этапа: сначала он требует ввести пароль пользователя drew (отвернитесь, пока я ввожу его); затем Vim предупреждает, что файл будет изменен, и предлагает меню с несколькими вариантами на выбор. Я рекомендую нажать `l`, чтобы снова загрузить файл в буфер.

Теперь разберемся, как все это работает. Команда `:write !{cmd}` посылает содержимое буфера на стандартный ввод указанной команде `{cmd}`, которая может быть любой внешней программой (см. `:h :write_c` ⓘ http://vimdoc.sourceforge.net/htmldoc/editing.html#:write_c). Сам редактор Vim по-прежнему выполняется с привилегиями пользователя drew, но мы можем сообщить, что внешний процесс должен быть запущен с привилегиями суперпользователя. В данном случае всю работу выполняет утилита `tee`, которая запускается с помощью команды `sudo`, благодаря чему она получает возможность выполнить запись в файл */etc/hosts*.

Символ `%` имеет специальное значение в командной строке Vim: он замещается строкой пути к файлу в текущем буфере (см. `:h :_%` ⓘ http://vimdoc.sourceforge.net/htmldoc/cmdline.html#:_%), в данном случае – `/etc/hosts`. То есть заключительная часть команды выглядит так: `tee /etc/hosts > /dev/null`. Данная команда принимает содержимое буфера через стандартный ввод и записывает его в файл */etc/hosts*.

Vim обнаруживает, что файл был изменен внешней программой. Обычно это означает рассинхронизацию содержимого буфера и файла, и именно поэтому Vim предлагает сохранить версию в буфере или загрузить версию с диска. В данном примере так получилось, что буфер и файл содержат один и тот же текст.

Часть III. БЫСТРАЯ НАВИГАЦИЯ

Команды перемещения являются одними из важнейших при работе с редактором Vim. Они не только позволяют перемещать курсор из одного места в другое, но и дают возможность определять область действия в режиме ожидающего оператора. В этой части книги мы познакомимся с наиболее практичными командами перемещения, а также узнаем о командах перехода, обеспечивающих быструю навигацию между файлами.

Глава 8. Навигация внутри файлов

Редактор Vim поддерживает множество способов перемещения внутри документа, а также имеет команды для перехода между буферами. В этой главе мы сосредоточимся на командах перемещения внутри документа.

Пожалуй, самый простой способ навигации в документе – использовать клавиши управления курсором. Vim позволяет перемещать курсор вверх, вниз, влево и вправо, не убирая руки из основной позиции на клавиатуре, как будет показано в рецепте 47. Но это только начало – существуют более быстрые способы перемещения: в рецепте 49 будет показано, как перемещаться между словами; в рецепте 50 будет показано, как быстро переместиться к любому желаемому символу в текущей строке, а в рецепте 51 – как перемещаться с помощью команды поиска.

Команды перемещения – это не только средство навигации по документу. Они могут также использоваться в режиме ожидающего оператора для выполнения настоящей работы, как обсуждалось в рецепте 12 в главе 2. В этой главе вашему вниманию будут представлены примеры использования команд перемещений вместе с командами-операторами. Суперзвездами режима ожидающего оператора являются текстовые объекты, о которых мы поговорим в рецептах 52 и 53.

Vim поддерживает множество команд перемещений. Их все невозможно охватить в одной главе, поэтому я рекомендую обратиться к разделу справки `:h motion.txt` ⓘ http://vimdoc.sourceforge.net/htmldoc/motion.html, где приводится полный список таких команд. Поставьте перед собой цель добавлять в свой арсенал пару команд перемещений каждую неделю.

Рецепт 47. Не убирайте руки из основной позиции

Vim ориентирован на работу с клавиатурой. Узнайте, как перемещаться по тексту, не убирая рук из основной позиции, и вы сможете выполнять свою работу в Vim намного быстрее.

Первое, что должен усвоить тот, кто желает освоить метод слепой печати, – пальцы рук всегда должны находиться в основной позиции на основном ряду клавиш. На стандартной клавиатуре с раскладкой QWERTY[1] основными клавишами для пальцев левой руки являются `a`, `s`, `d` и `f`, а для правой – `j`, `k`, `l` и `;`. Когда руки находятся в этой позиции, мы можем достать пальцами любую клавишу на клавиатуре, не перемещая рук и не глядя на пальцы. Это – идеальная позиция для слепой печати.

Как и любой текстовый редактор, Vim позволяет использовать клавиши со стрелками для перемещения курсора, но при этом поддерживает альтернативную возможность делать то же самое с помощью клавиш `h`, `j`, `k` и `l`. Действуют они следующим образом:

Команда	Перемещает курсор
h	На одну позицию влево
l	На одну позицию вправо
j	На одну строку вниз
k	На одну строку вверх

Понятно, что такой способ управления курсором не так очевиден, как с помощью клавиш со стрелками. Клавиши `j` и `k` находятся рядом друг с другом, и из-за этого сложнее запомнить, какая из них перемещает курсор вверх, а какая вниз. И клавиша `l` перемещает курсор не влево, а вправо! Такая привязка клавиш объясняется историческими причинами, поэтому не ищите какое-то логическое объяснение[2].

Следующие несколько замечаний помогут вам запомнить, какая клавиша и что делает: буква `j` немного напоминает стрелку вниз. На клавиатуре с раскладкой QWERTY клавиши `h` и `l` расположены слева и справа друг от друга, отражая направление, в котором они перемещают курсор.

Несмотря на неочевидность выполняемых ими функций, вам все же стоит научиться пользоваться клавишами `h`, `j`, `k` и `l`. Чтобы до-

[1] ЙЦУКЕН, если говорить о кириллической клавиатуре. – *Прим. перев.*
[2] http://www.catonmat.net/blog/why-vim-uses-hjkl-as-arrow-keys/

стать клавиши со стрелками, необходимо убрать правую руку из основной позиции. А так как клавиши `h`, `j`, `k` и `l` остаются легкодостижимыми, вы можете перемещать курсор в Vim, не покидая основной позиции.

На первый взгляд экономия может показаться незначительной, но это только на первый взгляд. Как только вы приучите себя пользоваться клавишами `h`, `j`, `k` и `l` для перемещения по документу, работа в любых других редакторах, опирающихся на использование клавиш со стрелками, будет вызывать у вас ощущение неудобства. Вы будете удивляться, что так долго мирились с этим!

Оставьте свою правую руку там, где она должна быть

На клавиатуре с раскладкой QWERTY клавиши `j`, `k` и `l` как раз оказываются в основной позиции указательного, среднего и безымянного пальцев правой руки. По правилам слепой печати клавиша `h` должна нажиматься указательным пальцем, но для этого его нужно убрать с основной позиции и переместить влево. Некоторые видят в этом некоторое неудобство и смещают пальцы правой руки на одну клавишу влево, чтобы все клавиши `h`, `j`, `k` и `l` постоянно находились под пальцами. Пожалуйста, не делайте этого.

Как будет показано далее в этой главе, Vim поддерживает гораздо более скоростные способы перемещения по тексту. Если вы нажимаете клавишу `h` более двух раз подряд для перемещения в одной и той же строке, значит, вы тратите свое время впустую. Когда дело доходит до перемещения по горизонтали, вы можете перемещаться быстрее, перепрыгивая от слова к слову или пользуясь командой поиска символа (рецепты 49 и 50 ниже).

Я использую клавиши `h` и `l` для более точного позиционирования курсора, когда с помощью более грубых приемов промахиваюсь мимо цели. Кроме того, я довольно редко пользуюсь ими. Если учитывать, насколько редко я использую клавишу `h`, я рад, что до нее приходится тянуться на клавиатуре с раскладной QWERTY. С другой стороны, я часто пользуюсь командами поиска символа (рецепт 50), поэтому я еще больше рад, что клавиша `;` как раз находится под мизинцем.

Избавьтесь от привычки пользоваться клавишами со стрелками

Если вам сложно избавиться от привычки пользоваться клавишами со стрелками, попробуйте добавить следующие строки в свой файл *vimrc*:

motions/disable-arrowkeys.vim
http://media.pragprog.com/titles/dnvim/code/motions/disable-arrowkeys.vim

```
noremap <Up> <Nop>
noremap <Down> <Nop>
noremap <Left> <Nop>
noremap <Right> <Nop>
```

Они отключают функциональность клавиш со стрелками. Каждый раз, когда вы будете тянуться к клавишам со стрелками, вы будете вспоминать, что должны оставлять руки в основной позиции. Приобретение нового навыка использовать клавиши `h`, `j`, `k` и `l` займет совсем немного времени.

Я не рекомендую оставлять эти строки в файле *vimrc* навсегда, оставьте только на время, пока вы не привыкнете перемещать курсор клавишами `h`, `j`, `k` и `l`. В конце концов, вы всегда сможете найти клавишам со стрелками какое-нибудь более практичное применение.

Рецепт 48. Разница между фактическими строками и строками на экране

Старайтесь не путать фактические строки текста со строками на экране. Vim позволяет оперировать и теми, и другими.

В отличие от большинства текстовых редакторов, Vim различает фактические строки и строки на экране. Когда параметр `wrap` включен (он включен по умолчанию), каждая строка текста, длина которой превышает ширину окна, переносится на следующую строку на экране, благодаря чему текст в таких строках всегда будет виден на экране. Как результат одна строка в файле может занимать несколько строк на экране.

Самый простой способ обеспечить возможность различения таких строк – включить параметр `number`, управляющий нумерацией строк. Нумероваться при этом будут только фактические строки. Если строка на экране будет образована в результате переноса длин-

ной фактической строки, она окажется без номера. На рис. 8.1 изображено содержимое буфера Vim с тремя фактическими строками (имеющими номера) и девятью экранными строками:

```
  1 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pra
    esent ut sapien nulla, ac bibendum diam. Suspendisse rutrum
    euismod tincidunt.
  2 Duis leo eros, cursus a vehicula accumsan, venenatis nec mas
    sa. Maecenas porttitor, nulla vel congue euismod, neque puru
    s lobortis nisi, id placerat enim sapien nec enim.
  3 Vestibulum ante ipsum primis in faucibus orci luctus et ultr
    ices posuere cubilia Curae. Nullam pulvinar tempor mollis. M
    auris ac blandit turpis.
:set number                                  2,85          All
```

Рис. 8.1. Три фактические строки и девять экранных строк

Понимание разницы между фактическими и экранными строками играет важную роль, потому что Vim поддерживает команды перемещений для строк обоих типов. Команды `j` и `k` перемещают курсор вниз и вверх по фактическим строкам, тогда как команды `gj` и `gk` выполняют перемещение по экранным строкам.

Рассмотрим рис. 8.1 поближе. Допустим, что нам требуется переместить курсор вверх, в позицию слова «vehicular». Наша задача – переместить курсор вверх на одну экранную строку. Для достижения поставленной цели можно было бы нажать `gk`. Команда `k` переместит курсор вверх на одну фактическую строку, в позицию слова «ac», а это не то, что нам нужно в данном случае.

Vim также предоставляет команды для непосредственного перехода к первому или последнему символу в строке. В следующей таблице приводится перечень команд для перемещения по фактическим и экранным строкам:

Команда	Перемещает курсор
`j`	На одну фактическую строку вниз
`gj`	На одну экранную строку вниз
`k`	На одну фактическую строку вверх
`gk`	На одну экранную строку вверх
`0`	К первому символу в фактической строке
`go`	К первому символу в экранной строке
`^`	К первому непробельному символу в фактической строке
`g^`	К первому непробельному символу в экранной строке
`$`	К концу фактической строки
`g$`	К концу экранной строки

Обратите внимание на сходство: команды `j`, `k`, `0` и `$` оперируют фактическими строками, а с префиксом `g` те же самые команды оперируют экранными строками.

Большинство других текстовых редакторов не поддерживают понятие фактической строки и всегда оперируют только экранными строками. Первое время вы будете испытывать неудобства из-за поддержки двух понятий строк в редакторе Vim. Но когда вы научитесь пользоваться командами `gj` и `gk`, то оцените способность команд `j` и `k` покрывать большее расстояние одним нажатием на клавишу.

Переназначение команд перемещения между строками

Если для вас предпочтительнее, чтобы команды `j` и `k` оперировали экранными строками, а не фактическими, вы всегда можете переназначить их. Попробуйте добавить следующие строки в свой файл *vimrc*:

motions/cursor-maps.vim
http://media.pragprog.com/titles/dnvim/code/motions/cursor-maps.vim

```
nnoremap k gk
nnoremap gk k
nnoremap j gj
nnoremap gj j
```

Благодаря им клавиши `j` и `k` будут перемещать курсор вверх и вниз на одну экранную строку, а команды `gj` и `gk` – на одну фактическую строку (в противоположность настройкам Vim по умолчанию). Однако я не рекомендую вам использовать эти строки, если вам предстоит работать с редактором Vim на множестве разных машин. В такой ситуации луче приучить себя к поведению редактора, принятому по умолчанию.

Рецепт 49. Перемещение по словам

Редактор Vim имеет две скорости перемещения по словам. Обе позволяют одним нажатием перемещать курсор дальше, чем на один символ.

Vim поддерживает несколько команд перемещений, дающих возможность передвигать курсор вперед и назад сразу на одно слово (см. `:h word-motions` ⓘ http://vimdoc.sourceforge.net/htmldoc/motion.html#word-motions). Все они перечислены в таблице ниже:

Команда	Перемещает курсор
w	Вперед, к началу следующего слова
b	Назад, к началу текущего/предыдущего слова
e	Вперед, к концу текущего/следующего слова
ge	Назад, к концу предыдущего слова

Мысленно эти команды можно объединить в пары: команды w и b нацелены на начало слова, а команды e и ge – на конец слова. Команды w и e двигают курсор вперед, а команды b и ge – назад. Эту матрицу цель/направление иллюстрирует рис. 8.2.

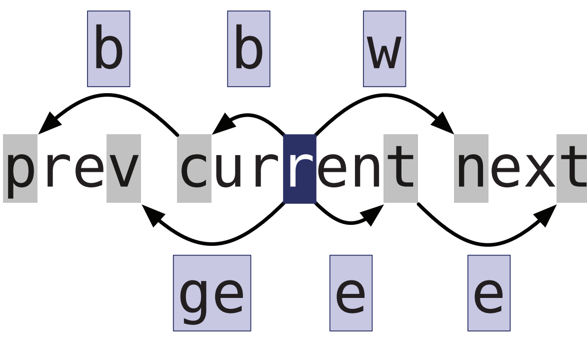


Рис. 8.2. Цель и направление
команд перемещения курсора на целое слово

Запомнить особенности действия этих команд непросто, и я не рекомендую заучивать их. Для начала попробуйте пользоваться командами w и b (если вам нужна мнемоника, запомните их как «(for-)*word*» и «*back*-word»[1]). Вы обязательно заметите, что перемещение вперед и назад на целое слово существенно ускоряет передвижение курсора в сравнении с клавишами h и l, передвигающими курсор на один символ.

Команды e и ge дополняют множество, но первое время вы вполне сможете обойтись и без них. Рано или поздно вы обнаружите, что иногда было бы удобно иметь возможность переходить сразу в конец слова одним нажатием на клавишу. Например, представьте, что требуется превратить слово «fast» в слово «faster»:

[1] Здесь автором обыгрывается сходство предложенных мнемоник со словами «forward» (вперед) и «backward» (назад). – *Прим. перев.*

Нажатия клавиш	Содержимое буфера
{start}	Go fast.
eaer<Esc>	Go faster.

Команды ea можно читать как «Append at the end of the current word» (добавить в конец текущего слова). Я пользуюсь комбинацией ea достаточно часто, чтобы воспринимать ее как единую команду. Иногда полезной может оказаться команда gea, которую можно читать как «append at the end of the previous word» (добавить в конец предыдущего слова»).

Отличайте слова и СЛОВА

Мы много говорим о словах, но до сих пор не дали определение этому понятию. В Vim поддерживаются и различаются два определения: «слово» и «СЛОВО». Для всех команд перемещения курсора между словами имеются эквивалентные команды для перемещения между СЛОВАМИ: W, B, E и gE.

Слово «слово» состоит из последовательности букв, цифр и символов подчеркивания, или из последовательности других непробельных символов, окруженной пробельными символами (см. :h word ⓘ http://vimdoc.sourceforge.net/htmldoc/motion.html#word). Определение слова «СЛОВО» проще – это последовательность непробельных символов, окруженная пробельными символами (см. :h WORD ⓘ http://vimdoc.sourceforge.net/htmldoc/motion.html#WORD).

Хорошо, но что это в действительности означает? Оставим детали разработчикам Vim и посмотрим с точки зрения пользователя: СЛОВА – больше, чем слова! Взгляните на следующий текст и подсчитайте в нем слова:

```
e.g. we're going too slow
```

Сколько слов вы насчитали? Пять? Десять? Или какое-то другое их число между пятью и десятью? В действительности в этом фрагменте пять СЛОВ и десять слов. Точки и апострофы тоже считаются словами, поэтому если мы решим двигаться через эту строку с помощью команды w, это будет медленное движение:

Нажатия клавиш	Содержимое буфера
{start}	e.g. we're going too slow
wwww	e.g. we're going too slow
www	e.g. we're going too slow

Но если использовать команды перемещения по СЛОВАМ, мы достигнем той же точки лишь парой нажатий.

Нажатия клавиш	Содержимое буфера
{start}	e.g. we're going too slow
W	e.g. we're going too slow
W	e.g. we're going too slow

В этом примере команды перемещения по СЛОВАМ выглядят предпочтительнее, но так бывает не всегда. Иногда бывает желательно, чтобы, скажем, последовательность «we» интерпретировалась как слово. Например, представьте, что слово «we» нужно заменить словом «you»:

Нажатия клавиш	Содержимое буфера
{start}	e.g. we're going too slow
cwyou<ESC>	e.g. you're going too slow

Иногда, наоборот, бывает желательно интерпретировать последовательность «we're» как СЛОВО. Например, представьте: это СЛОВО необходимо заменить на «it's»:

Нажатия клавиш	Содержимое буфера
{start}	e.g. we're going too slow
cWit's<ESC>	e.g. it's going too slow

Когда требуется двигаться быстро, используйте команды перемещения по СЛОВАМ, а когда требуется более высокая точность, используйте команды перемещения по словам. Поиграйте с ними, и вы интуитивно поймете, когда какие команды использовать, даже не зная всех тонкостей реализации.

Рецепт 50. Поиск символов

Команды поиска символов в Vim позволяют перемещаться вдоль строки еще быстрее, и они прекрасно работают в режиме ожидающего оператора.

Команда `f{char}` – один из самых быстрых способов перемещения по тексту в Vim. Она пытается найти указанный символ, начиная от текущей позиции курсора и до конца текущей строки. Если совпадение будет найдено, курсор переместится к найденному символу. В противном случае курсор останется на месте (см. `:h f` ⓘ http://vimdoc.sourceforge.net/htmldoc/motion.html#f).

На первый взгляд, этот прием выглядит слишком сложным, но на практике все оказывается гораздо проще. Взгляните:

Нажатия клавиш	Содержимое буфера
{start}	Find the first occurrence of {char} and move to it.
fx	Find the first occurrence of {char} and move to it.
fo	Find the first occurrence of {char} and move to it.

В данном примере команда fx ничего не сделала. Vim попытался найти символ «x» правее курсора в текущей строке, но не нашел и оставил курсор на месте. Команда fo нашла символ «o», поэтому курсор переместился в позицию первого найденного совпадения.

В данном примере вы не сможете поместить курсор в начало слова «occurrence», нажав меньше двух клавиш. Команда f{char} весьма эффективна, и когда она срабатывает, как в данном примере, возникает ощущение, что Vim может читать наши мысли.

Но команда f{char} не всегда так эффективна, как в примере выше. Представьте, что нам потребовалось переместить курсор в позицию символа «c» в начале слова «char». Взгляните, что получится, если использовать команду fc:

Нажатия клавиш	Содержимое буфера
{start}	Find the first occurrence of {char} and move to it.
fc	Find the first occurrence of {char} and move to it.
;	Find the first occurrence of {char} and move to it.
;	Find the first occurrence of {char} and move to it.
;	Find the first occurrence of {char} and move to it.

Символ «c» встречается несколько раз в этой строке, поэтому мы не сразу попали в цель. Пришлось предпринять несколько попыток, чтобы курсор встал туда, куда надо. К счастью, нам не пришлось при этом явно повторять команду fc. Vim запоминает последнюю команду поиска f{char} и позволяет повторить ее с помощью команды ; (см. :h ; ℹ http://vimdoc.sourceforge.net/htmldoc/motion.html#;). В данном примере нам пришлось нажать клавишу ; трижды, чтобы попасть в нужную позицию.

Команды f{char} и ; составляют мощную комбинацию и позволяют покрыть большое расстояние несколькими нажатиями на клавиши. Но где окажется курсор после очередного нажатия, не всегда очевидно. Как результат легко лишний раз нажать клавишу ; и промазать мимо цели. Например, представьте, что нам требуется поместить курсор в начало слова «of»:

Нажатия клавиш	Содержимое буфера
{start}	Find the first occurrence of {char} and move to it.
fo	Find the first occurrence of {char} and move to it.
;;	Find the first occurrence of {char} and move to it.
;	Find the first occurrence of {char} and move to it.

Случайно промахнувшись, мы можем вернуться назад командой `,`. Она повторит последний поиск `f{char}`, но в обратном направлении (см. `:h ,` http://vimdoc.sourceforge.net/htmldoc/motion.html#,).

Вспомните мантру из рецепта 4 в главе 1: действие, повтор, возврат. Я думаю о `,` как о подушке безопасности, когда чересчур рьяно нажимаю клавишу `;`.

Не забывайте команду повторения поиска символа в обратном направлении

Vim практически каждой клавише на клавиатуре присваивает какую-нибудь функцию. Если вы желаете создать собственный набор привязок к клавишам, как это реализовать? Vim предоставляет клавишу `<Leader>`, играющую роль пространства имен команд, определяемых пользователем. Ниже показано, как создать собственные привязки с использованием `<Leader>`:

```
noremap <Leader>n nzz
noremap <Leader>N Nzz
```

По умолчанию роль клавиши `<Leader>` играет клавиша `\`, поэтому определенные выше команды можно вызывать нажатием последовательностей `\n` и `\N`. Чтобы узнать, что делают эти новые команды, загляните в справку `:h zz` http://vimdoc.sourceforge.net/htmldoc/scroll.html#zz.

На некоторых клавиатурах клавиша `\` расположена очень неудобно, поэтому Vim упрощает замену клавиши `<Leader>` (см. `:h mapleader` http://vimdoc.sourceforge.net/htmldoc/map.html#mapleader). Обычно эту клавишу привязывают к клавише «запятая». Если вы пойдете таким путем, надо привязать функцию повторения поиска символа в обратном направлении к какой-нибудь другой клавише. Например:

```
let mapleader=","
noremap \ ,
```

Команды `;` и `,` дополняют друг друга. Если одну из них выбросить, все семейство команд поиска символа станет менее ценным.

Поиск символа может выполняться включительно или исключительно

Команды `f{char}`, `;` и `,` являются лишь частью множества команд поиска символа. Полный список этих команд приводится в таблице ниже:

Команда	Действие
`f{char}`	Поиск вперед, до следующего вхождения `{char}`
`F{char}`	Поиск назад, до предыдущего вхождения `{char}`
`t{char}`	Поиск вперед, до символа, стоящего перед следующим вхождением `{char}`
`T{char}`	Поиск назад, до символа, стоящего после предыдущего вхождения `{char}`
`;`	Повторить последнюю команду поиска символа `{char}`
`,`	Повторить последнюю команду поиска символа `{char}` в обратном направлении `{char}`

Команды `t{char}` и `T{char}` можно интерпретировать как поиск *до* (*till*) указанного символа. Курсор останавливается *за один символ до искомого символа* `{char}`, а при использовании команд `f{char}` и `F{char}` курсор останавливается *на* искомом символе.

Возможно, пока непонятно, зачем могли бы потребоваться две разновидности поиска символа. Но давайте рассмотрим следующий пример, демонстрирующий их в действии:

Нажатия клавиш	Содержимое буфера
`{start}`	`I've been expecting you, Mister Bond.`
`f,`	`I've been expecting you, Mister Bond.`
`dt.`	`I've been expecting you.`

Сначала нужно установить курсор непосредственно на символ запятой. Сделать это можно с помощью команды `f,`. Далее нужно удалить весь текст до конца предложения, кроме завершающей точки. С этой работой отлично справляется команда `dt.`.

Как вариант можно было бы использовать команду `dfd`, которая удалит все, от курсора до последнего символа в слове «Bond». В обоих случаях результат будет тем же самым, но, на мой взгляд, команда `dt.` требует меньше концентрации внимания. Удаление до символа «d»[1] – нетипичный шаблон, тогда как удаление до конца предложения[2] – операция настолько распространенная, что последовательность нажатий `f,dt.` можно рассматривать как своеобразное макроопределение на пальцах.

---

[1] Именно так «расшифровывается» команда `dfd`. – *Прим. перев.*

[2] А так «расшифровывается» команда `dt.`. – *Прим. перев.*

Вообще говоря, когда в командном режиме мне требуется быстро переместить курсор в текущей строке, я обычно использую команды `f{char}` и `F{char}`, а команды поиска `t{char}` и `T{char}` я чаще применяю в сочетании с командой `d{motion}` или `c{motion}`. Проще говоря, команды `f` and `F` я использую в командном режиме, а команды `t` и `T` – в режиме ожидающего оператора. За дополнительными подробностями обращайтесь к рецепту 12 в главе 2 (и к врезке «Режим ожидающего оператора»).

Думайте как при игре в «Балду»

Применяя команды поиска символов, можно существенно сэкономить на нажатиях клавиш, но их эффективность во многом зависит от выбранной цели. Любой опытный игрок в «Балду»[1] скажет вам, что одни буквы встречаются чаще, другие – реже. Если вы привыкнете выбирать менее употребимые буквы для использования в команде `f{char}`, то чаще будете поражать свою цель с первой попытки.

Допустим, что нам требуется удалить единственное прилагательное в следующем приложении:

```
Improve your writing by deleting excellent adjectives.
```

Какие команды можно было бы использовать, чтобы установить курсор на слово «excellent»? Если поставить перед собой целью установить курсор на первую букву слова командой `fe`, тогда нам придется нажать `;;;`, чтобы пропустить промежуточные символы «e». Лучшим решением является команда `fx`, которая достигает цели одним движением. Из этой точки мы сможем удалить слово командой `daw` (подробнее о команде `aw` рассказывается в рецепте 52 ниже).

Взгляните на текст, который вы только что прочитали. Он практически полностью составлен из строчных букв. Заглавные буквы встречаются намного реже, как и знаки пунктуации. При использовании команд поиска символа лучше выбирать редко встречающиеся символы. С практикой вы научитесь быстро различать их.

Рецепт 51. Поиск с целью навигации

Команды поиска позволяют быстро покрывать расстояния, большие и маленькие, всего лишь несколькими нажатиями на клавиши.

[1] http://ru.wikipedia.org/wiki/Балда_(игра) – *Прим. перев.*

Команды поиска символа (`f{char}`, `t{char}` и др.) просты и удобны, но они имеют серьезное ограничение. Эти команды могут выполнять поиск только одного символа и только в текущей строке. Поэтому, чтобы отыскать последовательность из нескольких символов или переместиться за пределы текущей строки, следует использовать команду глобального поиска.

Допустим, что нам требуется установить курсор на слово «takes» в следующем фрагменте:

motions/search-haiku.txt
http://media.pragprog.com/titles/dnvim/code/motions/search-haiku.txt

```
search for your target
it only takes a moment
to get where you want
```

Мы могли бы просто отыскать слово: `/takes<CR>`. В этом коротком фрагменте оно присутствует в единственном экземпляре, поэтому мы достигнем поставленной цели единственной командой. Но давайте посмотрим, можно ли решить ту же задачу меньшим количеством нажатий клавиш:

Нажатия клавиш	Содержимое буфера

`{start}`	`search for your target`
	`it only takes a moment`
	`to get where you want`

`/ta<CR>`	`search for your target`
	`it only takes a moment`
	`to get where you want`

`/tak<CR>`	`search for your target`
	`it only takes a moment`
	`to get where you want`

Поиск двухсимвольной последовательности «ta» дал два совпадения, а трехсимвольной последовательности «tak» – одно. В этом примере операция поиска выполняет короткий переход, но в больших документах этот прием можно использовать для покрытия больших расстояний всего несколькими нажатиями на клавиши. Команда глобального поиска – весьма экономичный способ навигации.

Даже если, выполняя поиск двухсимвольной последовательности «ta», курсор не достигает нужного места, мы всегда можем перейти к следующему вхождению, повторив последнюю команду поиска нажатием на клавишу `n`. Кроме того, если так получится, что вы нажмете клавишу `n` лишний раз, вы сможете вернуться назад командой

N. Мантра из рецепта 4 в главе 1 «действие, повтор, возврат» должна быть уже хорошо знакома вам.

В предыдущем рецепте мы видели, что команда fe редко оказывается полезной, потому что буква *e* достаточно часто встречается в словах. Мы можем избавиться от этого недостатка, выполняя поиск двухсимвольных или более длинных последовательностей. Буква *e* довольно часто встречается в английском языке, но намного реже встречается комбинация *re*. Просто удивительно, как часто можно перепрыгнуть к нужному слову, просто выполнив поиск по нескольким первым символам.

В примере со словом «takes» я включил параметр настройки hlsearch, отвечающий за подсветку найденных совпадений. При поиске короткой последовательности символов в тексте часто обнаруживается несколько совпадений. Результаты поиска при включенном параметре hlsearch могут вызывать рябь в глазах, поэтому, когда функция поиска часто используется для навигации по документам, бывает желательно отключить этот параметр (выключен по умолчанию). Однако параметр incsearch в таких ситуациях оказывается очень полезным. За дополнительной информацией обращайтесь к рецепту 82 в главе 13.

Используйте команды поиска в операциях

Команды поиска можно использовать не только в командном режиме. Их можно также применять в визуальном режиме и в режиме ожидающего оператора. Например, допустим, что нам требуется удалить текст «takes time but eventually» из следующей фразы:

Нажатия клавиш	Содержимое буфера
v	This phrase takes time but eventually gets to the point.
/ge<CR>	This phrase takes time but eventually gets to the point.
h	This phrase takes time but eventually gets to the point.
d	This phrase gets to the point.

Сначала командой v выполняется переход в визуальный режим. Затем выделенная область расширяется операцией поиска последовательности «ge», которая сразу переместила курсор в начало требуемого слова. Почти хорошо – осталось лишь устранить небольшой промах: выделение включает символ «g» в начале слова, но удалять его не тре-

буется, поэтому далее клавишей h выполняется перемещение на один символ влево. По окончании выделения оно удаляется командой d.

Ниже приводится еще более быстрый способ решения этой задачи:

Нажатия клавиш	Содержимое буфера
{start}	This phrase takes time but eventually gets to the point.
d/ge<CR>	This phrase gets to the point.

Здесь мы использовали команду перемещения /ge<CR>, чтобы сообщить команде d{motion}, какой фрагмент следует удалить. Использованная команда поиска не включает конечную цель, то есть даже при том, что курсор перемещается в позицию символа «g» в начале слова «gets», этот символ не включается в выделение и не подпадает под действие операции удаления (см. :h exclusive http://vimdoc.sourceforge.net/htmldoc/motion.html#exclusive).

Оставаясь в визуальном режиме, мы избавились от двух лишних нажатий на клавиши (см. также рецепт 23 в главе 4). Комбинирование оператора d{motion} с операциями поиска требует некоторой привычки, но этот прием обладает огромным потенциалом. Используйте его и поразите своих друзей и коллег.

Рецепт 52. Выделение фрагментов с применением текстовых объектов

Текстовые объекты позволяют выполнять операции со скобками, кавычками, тегами XML и другими распространенными текстовыми элементами.

Взгляните на следующий фрагмент кода:

motions/template.js
http://media.pragprog.com/titles/dnvim/code/motions/template.js

```
var tpl = [
    '<a href="{url}">{title}</a>'
]
```

Для каждой открывающей скобки { имеется закрывающая скобка }. То же относится к скобкам [и], < и > и к открывающим и закрывающим тегам HTML, <a> и . Данный фрагмент также содержит парные одиночные и двойные кавычки.

Vim распознает подобного рода структуры и дает возможность оперировать фрагментами текста, заключенными в них. *Текстовые объекты* определяют фрагменты текста по их структуре (см. `:h text-objects` http://vimdoc.sourceforge.net/htmldoc/motion.html#text-objects). Всего лишь парой нажатий на клавиши мы можем задействовать их, чтобы выделить фрагмент или выполнить некоторую операцию над ним.

Допустим, что курсор находится внутри фигурных скобок и нам требуется выделить текст, ограниченный символами `{}`. Это можно сделать, последовательно нажав клавиши `vi}`:

Нажатия клавиш	**Содержимое буфера**
{start}	var tpl = ['{title}']
vi}	var tpl = ['{title}']
a»	var tpl = ['{title}']
i>	var tpl = ['{title}']
it	var tpl = ['{title}']
at	var tpl = ['{title}']
a]	var tpl = ['{title}']

Обычно при работе в визуальном режиме один конец выделения закреплен за определенным символом, а другой – свободно перемещается. Когда используются такие команды перемещения курсора, как `l`, `w` и `f{char}`, мы можем расширить или сократить область выделения, перемещая свободный ее конец.

При работе с текстовыми объектами ситуация в корне иная. Когда нажимается последовательность `vi}`, Vim переходит в визуальный режим и выделяет все символы, заключенные в фигурные скобки `{}`. Где находился курсор перед этим, не имеет никакого значения, если к моменту вызова текстового объекта `i}` он находился внутри фигурных скобок.

Мы можем расширить область выделения с помощью другого текстового объекта. Например, текстовый объект a" выделяет последовательность символов, заключенную в двойные кавычки. Текстовый объект i> выделяет все, что находится между угловыми скобками.

Имена текстовых объектов в Vim состоят из двух символов, первый из которых всегда i или a. В общем случае можно сказать, что текстовые объекты, имена которых начинаются с символа i, выделяют последовательности символов *внутри* (*inside*) символов-ограничителей, а объекты, имена которых начинаются с символа a, включают в выделение также и символы-ограничители. Чтобы проще было запомнить, интерпретируйте символ i как *inside* (*внутри*), а символ a – *around* (*вокруг*) или *all* (*все*).

Рассмотрим пример выше с другой стороны, обратив особое внимание на символ в начале имени каждого текстового объекта, i или a. В частности, отметьте разницу между действием объектов it и at. Обратите также внимание, что a] распространяет выделение на несколько строк.

В табл. 8.1 ниже приводится неполный список текстовых объектов, поддерживаемых в Vim. В интересах экономии из списка были исключены некоторые дубликаты. Например, i(и i) действуют совершенно эквивалентно, то же относится к текстовым объектам a[и a]. Вы можете использовать те имена объектов, которые больше соответствуют вашему стилю.

Таблица 8.1. Текстовые объекты

Текстовый объект	Выделяет
a) или ab	Пару (круглых скобок) с содержимым
i) или ib	Область внутри (круглых скобок)
a} или aB	Пару {фигурных скобок} с содержимым
i} или iB	Область внутри {фигурных скобок}
a]	Пару [квадратных скобок] с содержимым
i]	Область внутри [квадратных скобок]
a>	Пару <угловых скобок> с содержимым
i>	Область внутри <угловых скобок>
a'	Пару 'одиночных кавычек'
i'	Область внутри 'одиночных кавычек'
a"	Пару "двойных кавычек"
i"	Область внутри "двойных кавычек"
a`	Область внутри `обратных апострофов`
i`	Область внутри `обратных апострофов`
at	Пару <xml>тегов</xml>
it	Область внутри <xml>тегов</xml>

Выполнение операций с текстовыми объектами

Для знакомства с текстовыми объектами замечательно подходит визуальный режим, потому что он позволяет наглядно увидеть происходящее. Но истинная мощь текстовых объектов проявляется в режиме ожидающего оператора.

Текстовые объекты не являются командами перемещения курсора: их нельзя использовать для навигации по документу. Но текстовые объекты с успехом можно применять в визуальном режиме и в режиме ожидающего оператора. Запомните: всякий раз, когда вы видите команду, синтаксис которой включает `{motion}`, в этой команде можно использовать текстовые объекты. Типичными примерами таких команд являются: `d{motion}`, `c{motion}` и `y{motion}` (см. табл. 2.1).

Рассмотрим в качестве примера команду `c{motion}`, которая удаляет указанный текст и переключает редактор в режим вставки (`:h c` http://vimdoc.sourceforge.net/htmldoc/change.html#c). Мы воспользуемся ею, чтобы заменить `{url}` на символ `#`, а затем с ее же помощью заменим `{title}` некоторым текстом:

Нажатия клавиш	Содержимое буфера
`{start}`	`'<a href="{url}">{title}</a>'`
`ci"#<Esc>`	`'<a href="#">{title}</a>'`
`citclick here<Esc>`	`'<a href="#">click here</a>'`

Команду `ci"` можно интерпретировать как «change inside the double quotes» (изменить текст в двойных кавычках). Команду `cit` можно интерпретировать как «change inside the tag» (изменить текст внутри тега). С таким же успехом мы могли бы использовать команду `yit`, чтобы скопировать текст внутри тега, или команду `dit`, чтобы удалить его.

Обсуждение

Каждая из этих команд требует нажать всего три клавиши и, несмотря на свою краткость, обладает определенной выразительностью. Я мог бы даже сказать, что их назначение достаточно прозрачно видно из их имен. Это обусловлено тем, что данные команды следуют простым правилам грамматики Vim, о которых рассказывалось в рецепте 12 главы 2.

В рецептах 50 и 51 выше мы познакомились с парой приемов, позволяющих перемещать курсор с высокой точностью. Используем ли мы команду `f{char}` поиска единственного символа или команду `/target<CR>` поиска последовательности символов, принцип остается тем же: мы выбираем подходящую цель, прицеливаемся и стреляем. В случае успеха мы поразим цель первым же выстрелом. Эти приемы позволяют нам покрывать большие расстояния с минимумом усилий.

Текстовые объекты – это следующий уровень. Если команды `f{char}` и `/pattern<CR>` сравнить с ударом одной ногой в прыжке, то текстовые объекты можно сравнить с ударом двумя ногами, поражающим сразу две цели, как показано на рис. 8.3.

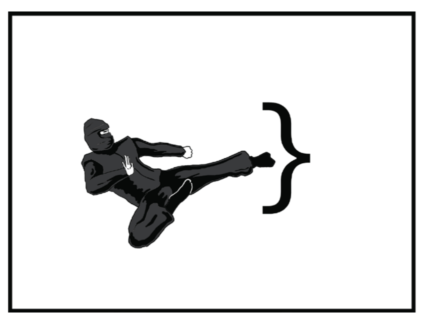
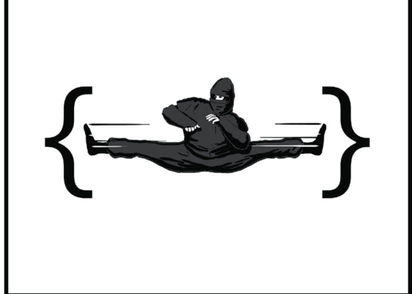



Рис. 8.3. Команды перемещений и текстовые объекты Vim способны поражать цель с непревзойденной точностью

Рецепт 53. Удаление, включая ограничители, и изменение внутри ограничителей

Текстовые объекты обычно определяются парами: один определяет область действия внутри соответствующих ограничителей, а другой – включая эти ограничители. В этом рецепте мы исследуем типичные случаи применения обеих разновидностей текстовых объектов.

Текстовые объекты Vim делятся на две категории: действующие с парами ограничителей, такие как `i)`, `i"` и `it`, и действующие с фрагментами текста, такими как слова, предложения и абзацы.

Некоторые из объектов, входящих во вторую категорию, перечислены в табл. 8.2.

Таблица 8.2. Связанные текстовые объекты

Текстовый объект	Выделяет
`iw`	Текущее слово
`aw`	Текущее слово плюс один пробел
`iW`	Текущее СЛОВО
`aW`	Текущее СЛОВО плюс один пробел
`is`	Текущее предложение
`as`	Текущее предложение плюс один пробел
`ip`	Текущий абзац
`ap`	Текущий абзац плюс одна пустая строка

Первую категорию текстовых объектов я называю *текстовыми объектами с ограничителями*, потому что соответствующие им фрагменты текста начинаются и заканчиваются парными символами-ограничителями. Слова, предложения и абзацы – это связанные, цельные фрагменты текста, поэтому текстовые объекты, соответствующие этим фрагментам, я называю *связанными текстовыми объектами* (в документации Vim эти текстовые объекты называются «блочными» и «неблочными», но мне кажется, что эти определения не отражают всю полноту различий).

Давайте сравним текстовые объекты `iw` и `aw`. Вспомнив, о чем говорилось в предыдущем рецепте, мы можем подумать, что один из них действует *внутри* слова, а другой – *вокруг* слова, включая ограничители соответственно. Но что это означает?

Текстовый объект `iw` определяет область действия от первого до последнего символа текущего слова. Текстовый объект `aw` определяет ту же область действия плюс один пробельный символ до или после слова, если имеется. Правила определения границ слов Vim приводятся в рецепте 49 выше.

Разница между `iw` и `aw` очень тонкая, и не сразу понятно, зачем могут потребоваться два почти идентичных текстовых объекта, поэтому давайте рассмотрим типичные случаи их применения.

Допустим, что нам необходимо удалить слово «excellent» из следующего предложения. Это легко сделать с помощью команды `daw`:

Нажатия клавиш	Содержимое буфера
`{start}`	`Improve your writing by deleting excellent adjectives.`
`daw`	`Improve your writing by deleting adjectives.`

Она удалит слово плюс один пробел, давая желаемый результат. Если бы вместо нее мы использовали команду `diw`, в предложении остались бы два пробела, идущих подряд, что, вероятно, не совсем то, чего нам хотелось бы.

Теперь допустим, что нам потребовалось заменить это слово другим. На этот раз мы воспользуемся командой `ciw`:

Нажатия клавиш	Содержимое буфера
`{start}`	`Improve your writing by deleting excellent adjectives.`
`ciwmost<Esc>`	`Improve your writing by deleting most adjectives.`

Команда `ciw` удалит слово, не захватывая дополнительного пробела, и перейдет в режим вставки. Это именно то, что нам требуется. Если вместо нее использовать команду `caw`, после ввода нового слова мы получили «mostadjectives». Эту ошибку легко исправить, но лучше, конечно, вообще не допускать ее.

В общем случае в комбинации с командой `d{motion}` часто бывает лучше использовать текстовые объекты `aw`, `as` и `ap`, а с командой `c{motion}` – текстовые объекты `iw` и подобные ему.

Рецепт 54. Установка меток и возврат к ним

Метки редактора Vim позволяют быстро перемещаться между определенными точками в тексте. Метки можно устанавливать вручную, но Vim также способен автоматически определять, какие точки представляют для нас наибольший интерес.

Команда `m{a-zA-Z}` устанавливает метку в текущую позицию курсора, назначая ей указанную букву (`:h m` ⓘ http://vimdoc.sourceforge.net/htmldoc/motion.html#m). Метки с буквами в нижнем регистре являются локальными для буфера, тогда как метки с буквами в верхнем регистре доступны глобально. Мы познакомимся с ними поближе в рецепте 59 в главе 9. Vim никак не выделяет позиции меток в тексте, но если вы все сделали правильно, то сможете переходить к меткам несколькими нажатиями клавиш из любого места в буфере.

Для перехода к меткам в командном режиме Vim имеется две команды. (Будьте внимательны, они очень похожи!) Команда `'{mark}` выполняет переход к строке с меткой и устанавливает курсор на первый непробельный символ. Команда `` `{mark} `` выполняет переход к точной позиции метки, устанавливая курсор в строку и в позицию

в строке, где эта метка была установлена (см. `:h mark-motions` http://vimdoc.sourceforge.net/htmldoc/motion.html#mark-motions).

Если вы собираетесь пользоваться только одной из этих команд, выбирайте `` `{mark} ``. Эта команда всегда будет соответствовать вашим ожиданиям – и когда вам потребуется точно вернуться в позицию метки, и когда достаточно будет вернуться в строку с меткой. Единственное, когда действительно потребуется использовать команду `'{mark}`, – в контексте команд Ex (см. рецепт 28 в главе 5).

Команды `mm` и `` `m `` составляют очень удобную пару. Первая устанавливает метку, а вторая выполняет переход к ней. В разделе «Как поменять два слова местами» в рецепте 61 (глава 10) приводится пример использования этих команд, чтобы быстро поменять два слова местами.

Автоматическая расстановка меток

В каждом буфере можно установить до 26 меток – каждая буква алфавита соответствует одной метке, и этого более чем достаточно! В редакторе `vi`, предшественнике Vim, отсутствовало такое понятие, как визуальный режим. В те времена метки играли куда более важную роль, чем сейчас. Многие задачи, требовавшие применения меток в `vi`, в редакторе Vim могут быть решены с применением визуального режима.

Но метки не потеряли свою актуальность в Vim; они сохранили за собой определенную область применения. В частности, метки, которые Vim устанавливает автоматически, могут сослужить неплохую службу. В их число входят метки, перечисленные в табл. 8.3 ниже.

Таблица 8.3. Автоматические метки в Vim

Метка	Место установки
` `` `	Позиция перед последним переходом в текущем файле
`` `. ``	Место последнего изменения
`` `^ ``	Место последней вставки
`` `[ ``	Начало последнего изменения или скопированного блока
`` `] ``	Конец последнего изменения или скопированного блока
`` `< ``	Начало последнего выделения, сделанного в визуальном режиме
`` `> ``	Конец последнего выделения, сделанного в визуальном режиме

Метка ` `` ` дополняет список переходов (рецепт 56 в главе 9), и мы увидим, как она действует, в следующем рецепте. Метка `` `. `` дополня-

ет список изменений, о котором будет рассказываться в рецепте 57 главы 9.

Начало и конец последнего выделения, сделанного в визуальном режиме, сохраняются в виде меток автоматически, поэтому визуальный режим можно даже рассматривать как своеобразный интерфейс к меткам.

Рецепт 55. Переход между парными скобками

Редактор Vim поддерживает команды перемещения, позволяющие перемещаться между парными открывающими и закрывающими скобками. Включив расширение matchit.vim, такое поведение редактора Vim можно распространить на пары тегов XML, а также на ключевые слова в некоторых языках программирования.

Команда % выполняет переходы между открывающими и закрывающими скобками (см. `:h %` http://vimdoc.sourceforge.net/htmldoc/motion.html#%). Она различает скобки (), {}, and [], как показано в следующем примере:

Нажатия клавиш	Содержимое буфера
{start}	`console.log([{'a':1},{'b':2}])`
%	`console.log([{'a':1},{'b':2}])`
h	`console.log([{'a':1},{'b':2}])`
%	`console.log([{'a':1},{'b':2}])`
l	`console.log([{'a':1},{'b':2}])`
%	`console.log([{'a':1},{'b':2}])`

Чтобы понять, как можно использовать команду % на практике, поэкспериментируем с коротким фрагментом кода на языке Ruby:

motions/parentheses.rb
http://media.pragprog.com/titles/dnvim/code/motions/parentheses.rb

```
cities = %w{London Berlin New\ York}
```

Допустим, что нам требуется преобразовать фрагмент `%w{London Berlin New\ York}` в определение обычного списка: `[«London», «Berlin», «New York»]`. Для этого нам нужно заменить фигурные скобки квадратными. Кому-то может показаться, что это отличный случай пустить в ход команду %. И они были бы правы, если бы не одно «НО»!

Допустим, что мы установили курсор на открывающую фигурную скобку и вводим команду `r[`, чтобы заменить ее открывающей квадратной скобкой. В результате у нас получилась странная конструкция: `[London Berlin New\ York}`. Команда `%` может работать только с парными скобками, поэтому ее нельзя использовать, чтобы перейти к закрывающей фигурной скобке `}`.

Вся хитрость состоит в том, чтобы использовать команду `%` *до* внесения каких-либо изменений. Когда вызывается команда `%`, Vim автоматически устанавливает метку в позицию, откуда выполняется переход. Благодаря этому появляется возможность вернуться назад командой ` `` `. Ниже приводится частичное решение для нашего примера рефакторинга кода:

Нажатия клавиш	Содержимое буфера
`{start}`	`cities = %w{London Berlin New\ York}`
`dt{`	`cities = {London Berlin New\ York}`
`%`	`cities = {London Berlin New\ York}`
`r]`	`cities = {London Berlin New\ York]`
` `` `	`cities = {London Berlin New\ York]`
`r[`	`cities = [London Berlin New\ York]`

Обратите внимание, что в этом примере вместо команды ` `` ` можно было бы использовать команду `<C-o>` (см. рецепт 56 в главе 9). Расширение surround.vim поддерживает команды, еще более упрощающие эту задачу. Подробнее об этом рассказывается во врезке «Surround.vim» ниже.

Переход между парными ключевыми словами

В состав дистрибутива Vim входит расширение `matchit`, увеличивающее возможности команды `%`. Когда это расширение включено, с помощью команды `%` оказывается возможным переходить между парными ключевыми словами. Например, в тексте с разметкой HTML команда `%` сможет переходить между открывающими и закрывающими тегами. В коде на языке Ruby она сможет переходить между парами ключевых слов `class`/`end`, `def`/`end` и `if`/`end`.

Но даже при том, что расширение `matchit` входит в состав дистрибутива Vim, оно выключено по умолчанию. Следующие строки в файле `vimrc` обеспечат автоматическую загрузку расширения `matchit` при запуске Vim:

```
set nocompatible
filetype plugin on
runtime macros/matchit.vim
```

Возможности, поддерживаемые этим расширением, имеют большую практическую ценность, поэтому я рекомендую включить его у себя. За дополнительной информацией обращайтесь к разделу справки `:h matchit-install` ⓘ http://vimdoc.sourceforge.net/htmldoc/usr_05.html#matchit-install.

Surround.vim

Расширение surround.vim, написанное Тимом Поупом (Tim Pope)[1], является одним из моих любимых расширений. Оно позволяет легко заключить выделенный фрагмент в пару символов-ограничителей. Например, ниже показано, как с его помощью можно было бы заключить пару слов `New York` в кавычки:

Нажатия клавиш	Содержимое буфера
`{start}`	`cities = ["London", "Berlin", New York]`
`vee`	`cities = ["London", "Berlin", New York]`
`S"`	`cities = ["London", "Berlin", "New York"]`

Расширение surround.vim предоставляет команду `S"`, которую можно интерпретировать как «Surround the selection with a pair of double quote marks» (окружить выделение парой двойных кавычек). Точно так же можно окружить выделение парой круглых или фигурных скобок, применив команду `S)` или `S}`.

С помощью surround.vim можно даже *заменить* имеющиеся символы-ограничители. Например, заменить `{London}` на `[London]` можно с помощью команды `cs}]`, которую можно интерпретировать как «Change surrounding {} braces to [] brackets» (заменить окружающие скобки {} на []). Обратную замену можно выполнить командой `cs]}`. Это очень мощное расширение, обязательно установите его себе.

[1] http://github.com/tpope/vim-surround

Глава 9. Навигация между файлами

Как рассказывалось в предыдущей главе, Vim поддерживает команды, позволяющие перемещаться по содержимому файла. Однако существуют также команды, дающие возможность перемещаться между файлами. Редактор Vim предоставляет несколько команд, которые превращают ключевые слова в документе в своеобразные «червоточины», обеспечивающие возможность быстрого перемещения от одного документа к другому. Как ни странно, но Vim всегда следит за нашими перемещениями, чтобы мы легко могли вернуться обратно тем же путем.

Рецепт 56. Обход списка переходов

Vim сохраняет местоположение до и после перехода и предоставляет несколько команд, с помощью которых можно повторить эти переходы.

Мы давно уже привыкли пользоваться кнопкой **Back** (Назад) в веб-браузерах, чтобы вернуться на страницы, посещенные ранее. Редактор Vim поддерживает аналогичную возможность посредством *списка переходов* (jump list): команда `<C-o>` действует подобно кнопке **Back** (Назад), а дополняющая ее команда `<C-i>` – подобно кнопке **Forward** (Вперед). Эти команды дают возможность путешествовать по списку переходов Vim, но что такое *переход*?

Для начала обозначим некоторые отличия: команда перемещения вызывает перемещение курсора *внутри файла*, тогда как команды перехода выполняют переходы *между файлами* (впрочем, как будет показано ниже, некоторые команды перемещения также могут классифицироваться как переходы). Исследовать содержимое списка переходов можно с помощью следующей команды:

```
⇛ :jumps
    jump line   col file/text
      4     12     2 <recipe id="sec.jump.list">
      3    114     2 <recipe id="sec.change.list">
      2    169     2 <recipe id="sec.gf">
      1    290     2 <recipe id="sec.global.marks">
>
Press Enter or type command to continue
```

Переходом может называться любая команда, выполняющая замену активного файла в текущем окне. В списке переходов Vim запоминает позицию курсора до и после выполнения такой команды. Например, если выполнить команду :edit, чтобы открыть новый файл (см. рецепт 42 в главе7), мы получим возможность использовать команды <C-o> и <C-i> для перемещения вперед и назад между двумя файлами.

Непосредственное перемещение к строке с определенным номером ([count]G) также считается переходом, но перемещение вверх или вниз на одну строку переходом не является. Перемещения между предложениями и абзацами являются переходами, а перемещение между символами и словами – нет. Обобщая, можно сказать, что перемещения на дальние дистанции могут классифицироваться как переходы, а перемещения на короткие дистанции рассматриваются только как перемещения.

В следующей таблице перечислены некоторые команды переходов:

Команда	Действие
[count]G	Переход к строке с указанным номером
/pattern<CR>/ ?pattern<CR>/n/N	Переход к следующему/предыдущему вхождению шаблона pattern
%	Переход к парной скобке
(/)	Переход к началу следующего/предыдущего предложения
{/}	Переход к началу следующего/предыдущего абзаца
H/M/L	Переход к верхнему краю/середине/нижнему краю экрана
gf	Переход к файлу с именем под курсором
<C-]>	Переход к определению ключевого слова под курсором
'{mark}/`{mark}	Переход к метке

Сами команды <C-o> и <C-i> никогда не интерпретируются как перемещения. То есть их нельзя использовать, чтобы выполнить выделение в визуальном режиме или определить область действия какой-либо команды в режиме ожидающего оператора. Я склонен воспринимать список переходов как некоторый маршрут с контрольны-

ми точками, упрощающий перемещение по файлам, посещавшимся в течение сеанса редактирования.

Редактор Vim поддерживает сразу несколько списков переходов. В действительности каждое отдельное окно имеет собственный список переходов. Если разбить рабочую область на несколько окон или вкладок, команды <C-o> и <C-i> всегда будут использовать список переходов активного окна.

Остерегайтесь изменять привязку клавиши табуляции

Нажав комбинацию <C-i> в режиме вставки, можно заметить, что она имеет тот же эффект, что и клавиша <Tab>. Это обусловлено тем, что при нажатии комбинации <C-i> и клавиши <Tab> редактор Vim получает одинаковые коды.

Помните, что, переназначив клавишу <Tab>, связав ее с какой-то другой функцией, вы также переназначите комбинацию <C-i> (и наоборот). На первый взгляд, в этом не видно никаких проблем, но представьте: если вы свяжете клавишу <Tab> с какой-то другой функцией, тем самым вы измените поведение по умолчанию комбинации <C-i>. Подумайте как следует, действительно ли это стоящий обмен. Выгоды от списка переходов будут намного меньше, если вы сможете перемещаться по нему только в одном направлении.

Рецепт 57. Обход списка изменений

Vim сохраняет местоположение курсора после каждого изменения документа. Обход этого списка выполняется очень просто и может служить еще одним способом навигации по документу.

Приходилось ли вам когда-нибудь выполнять команду отмены и сразу за ней команду повтора изменений (команды undo и redo)? Эти две команды являются взаимно противоположными, но производят побочный эффект, помещая курсор в позицию последнего изменения. Это может пригодиться для перемещения к фрагменту документа, редактировавшемуся последним. Хотя команда u<C-r> и является грубым приемом, но она позволяет получить желаемый результат.

Как оказывается, редактор Vim поддерживает для каждого буфера отдельный список модификаций, выполняемых в течение сеанса редактирования. Он называется *списком изменений* (см. `:h changelist` ⓘ http://vimdoc.sourceforge.net/htmldoc/motion.html#changelist), а его содержимое можно исследовать, выполнив следующую команду:

```
➾  :changes
   change line   col  text
       3       1     8 Line one
       2       2     7 Line two
       1       3     9 Line three
>
Press ENTER or type command to continue
```

Этот пример показывает, что Vim сохраняет номер строки и номер позиции в строке для каждого изменения. С помощью команд `g;` и `g,` мы можем перемещаться взад и вперед по списку изменений. Напоминанием о действиях команд `g;` и `g,` могут служить команды `;` и `,`, используемые для повторения поиска командой `f{char}` в прямом и обратном направлении (см. рецепт 50 в главе 8).

Чтобы вернуться обратно к последнему изменению в документе, можно нажать `g;`. Эта команда поместит курсор в строку на символ, где он оказался после предыдущего изменения. Результат будет тем же, как после нажатия последовательности `u<C-r>`, только при этом не будет производиться временных изменений в документе.

Отметка последнего изменения

Vim автоматически создает две метки, дополняющие список изменений. Метка `` `. `` всегда ссылается на позицию последнего изменения (`` :h `. `` ⓘ http://vimdoc.sourceforge.net/htmldoc/motion.html#%60.), а метка `` `^ `` – на позицию курсора, где произошел выход из режима вставки (`` :h `^ `` ⓘ http://vimdoc.sourceforge.net/htmldoc/motion.html#%60^).

В большинстве случаев переход к метке `` `. `` производит тот же эффект, что и команда `g;`. Однако метка может ссылаться только на позицию самого последнего изменения, а список изменений может хранить множество местоположений. Мы можем нажимать `g;` снова и снова, и каждый раз оказываться в другом месте. Метка `` `. ``, напротив, всегда переносит курсор в позицию последнего изменения.

Метка `` `^ `` ссылается на позицию последней *вставки*, которая несколько отличается от последнего *изменения*. Если выйти из режима вставки и затем перейти в другое место документа, мы легко можем вернуться обратно командой `gi` (`:h gi` ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#gi). Одним движением, воспользовавшись меткой `` `^ ``, можно вернуть курсор в предыдущую позицию и переключиться в режим вставки. Это позволит сэкономить массу времени!

Для каждого буфера поддерживается свой список изменений, в отличие от списка переходов, который поддерживается для каждого окна.

Рецепт 58. Переход к файлу с именем под курсором

Имена файлов в документе могут интерпретироваться редактором Vim как своеобразные гиперссылки. При правильных настройках можно использовать команду `gf` для перехода к файлам с именами под курсором.

Для демонстрации воспользуемся каталогом *jumps*, который находится в пакете с загружаемыми примерами к книге. Он содержит следующее дерево подкаталогов:

```
practical_vim.rb
practical_vim/
  core.rb
  jumps.rb
  more.rb
  motions.rb
```

Сначала перейдите в каталог *jumps* в командной оболочке, а затем запустите редактор Vim. Для данной демонстрации я рекомендую использовать флаги `-u NONE -N`, обеспечивающие запуск Vim без загрузки дополнительных расширений:

```
⇒ $ cd code/jumps
⇒ $ vim -u NONE -N practical_vim.rb
```

Файл *practical_vim.rb* содержит только инструкции загрузки файлов *core.rb* и *more.rb*:

jumps/practical_vim.rb
http://media.pragprog.com/titles/dnvim/code/jumps/practical_vim.rb

```
require 'practical_vim/core'
require 'practical_vim/more'
```

Разве не было бы полезно иметь возможность быстро взглянуть на содержимое файла, указанного в директиве `require`? Именно для этих целей существует команда `gf`. Ее имя можно расшифровать как «go to file» (перейти к файлу) (`:h gf` ⓘ http://vimdoc.sourceforge.net/htmldoc/editing.html#gf).

Попробуем воспользоваться ею. Сначала поместим курсор где-нибудь в пределах строки `'practical_vim/core'` (например, нажав `fp`). Если теперь попытаться выполнить команду `gf`, мы получим ошибку: «E447: Can't find file 'practical_vim/core' in path» (файл *practical_vim/core* не найден).

Редактор Vim попытался открыть файл с именем *practical_vim/core* и сообщил, что он не найден, но нам требовалось открыть файл *practical_vim/core.rb* (обратите внимание на расширение). Нам требуется каким-то способом сообщить редактору Vim, что перед попыткой открыть файл с именем под курсором он должен добавить расширение *.rb*. Сделать это можно с помощью параметра настройки `suffixesadd`.

Определение расширения файла

Параметр `suffixesadd` позволяет определить одно или более расширений имен файлов, которые Vim будет пытаться использовать при поиске файла командой `gf` (`:h 'suffixesadd'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'suffixesadd'). Выполнить необходимую настройку можно командой:

```
⇒ :set suffixesadd+=.rb
```

Теперь, когда мы попытаемся выполнить команду `gf`, Vim перейдет непосредственно к файлу с именем под курсором. Попробуйте сейчас открыть файл *more.rb*. В этом файле имеется несколько других объявлений. Выберите одно из них и откройте соответствующий файл командой `gf`.

Каждый раз, когда вызывается команда `gf`, Vim добавляет новую запись в список переходов, поэтому мы всегда можем вернуться обратно, откуда пришли, выполнив команду `<C-o>` (см. рецепт 56 выше). В данном случае первая команда `<C-o>` вернет нас обратно в файл *more.rb*, а вторая – в файл *practical_vim.rb*.

Определение списка каталогов для поиска

В примере выше все файлы в инструкциях `require` находились в дереве каталогов с корнем в текущем рабочем каталоге. Но что, если инструкция `require` ссылается на файл в сторонней библиотеке, такой как `rubygem`?

Решить эту проблему поможет параметр настройки `path` (`:h 'path'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'path').

В этом параметре можно указать список каталогов, разделенных точкой с запятой. При вызове команды gf Vim проверит каждый из каталогов, перечисленных в параметре path, на наличие файла с именем под курсором. Параметр path также используется командой :find, о которой рассказывалось в рецепте 43 в главе 7.

Исследовать значение параметра path можно командой

```
⇛ :set path?
  path=.,/usr/include,,
```

В данном контексте точка (.) обозначает каталог текущего файла, а пустая строка (ограниченная двумя запятыми) – рабочий каталог. Значение по умолчанию отлично подходит для простых случаев, но для работы с большими проектами может потребоваться включить в параметр path дополнительные каталоги.

Например, было бы удобно, если бы параметр path включал каталоги всех библиотек, используемых в проекте на Ruby. Благодаря этому можно было бы использовать команду gf для открытия любых модулей, указанных в инструкциях require. Существует и автоматизированное решение в виде расширения bundler.vim Тима Поупа (Tim Pope)[1], который использует файл *Gemfile* проекта для заполнения параметра path.

Обсуждение

В самом начале описания этого рецепта я рекомендовал запустить Vim с отключенными расширениями. Это обусловлено тем, что вместе с Vim обычно распространяется расширение vim-ruby для работы с файлами на Ruby, которое автоматически настраивает параметры suffixesadd и path. Если вам часто приходится работать с программным кодом на Ruby, я рекомендую получить последнюю версию этого расширения на сайте Github, потому что оно очень активно развивается[2].

Параметры suffixesadd и path можно настраивать отдельно для каждого буфера, благодаря чему можно иметь разные настройки для файлов разных типов. Кроме расширения для Ruby, в состав Vim входит также множество расширений для других языков, поэтому на практике вам нечасто придется настраивать описываемые параме-

---

1 https://github.com/tpope/vim-bundler
2 https://github.com/vim-ruby/vim-ruby

тры вручную. Но даже в этом случае важно понимать, как работает команда `gf`. Она превращает каждое имя файла в документе в гиперссылку, упрощая навигацию по компонентам проекта.

Похожую функцию выполняет команда `<C-]>`. Для правильной работы она требует выполнить дополнительные настройки (как описывается в рецепте 103 в главе 16), но взамен позволяет переходить из точки вызова любого метода к его определению. Загляните в рецепт 104 (глава 16), где демонстрируется действие этой команды.

Если списки переходов и изменений действуют как некоторый маршрут с контрольными точками, позволяя нам возвращаться обратно по своим следам, то команды `gf` и `<C-]>` служат своеобразными «червоточинами», дающими возможность перемещаться между файлами проекта.

Рецепт 59. Переключение между файлами с помощью глобальных меток

Глобальная метка – это своеобразная закладка, позволяющая переключаться между файлами. Метки особенно удобно использовать, чтобы обеспечить возможность возврата в файл после изучения некоторых других файлов.

Команда `m{letter}` создает метку в текущей позиции курсора (`:h m` http://vimdoc.sourceforge.net/htmldoc/motion.html#m). Если указать букву нижнего регистра, команда создаст локальную метку, доступную только в пределах буфера, а если указать букву верхнего регистра, будет создана глобальная метка. Установив метку, к ней можно вернуться командой `` `{letter} `` (`` :h ` `` http://vimdoc.sourceforge.net/htmldoc/motion.html#`).

Попробуйте проделать следующее: откройте файл *vimrc* и нажмите последовательность `mV`, чтобы установить глобальную метку (мнемоника: *V* для *vimrc*). Переключитесь в любой другой файл и затем нажмите последовательность `` `V `` – вы должны вернуться назад к глобальной метке в файле *vimrc*. По умолчанию глобальные метки сохраняются между сеансами правки (хотя это поведение можно изменить; см. `:h 'viminfo'` http://vimdoc.sourceforge.net/htmldoc/options.html'viminfo'). Теперь вы всегда сможете открыть файл *vimrc* двумя нажатиями на клавиши, если, конечно, вы не установите глобальную метку *V* в другое местоположение.

Устанавливайте глобальную метку перед погружением в код

Глобальные метки особенно удобны, когда возникает необходимость заглянуть в несколько файлов, а затем вернуться назад. Представьте, что вы работаете с некоторым кодом и хотите отыскать все вызовы метода `fooBar()` в проекте. Для этого можно воспользоваться командой `:vimgrep`, описываемой в рецепте 111 в главе 18):

```
⇒ :vimgrep /fooBar/ **
```

По умолчанию команда `:vimgrep` автоматически переходит к первому найденному совпадению, что может означать переход к другому файлу. Из этой точки мы можем вернуться обратно командой `<C-o>`.

Допустим, что в проекте имеются десятки совпадений с шаблоном `fooBar`. Для каждого найденного совпадения команда `:vimgrep` создает запись в списке результатов (quickfix list). Теперь допустим, что мы потратили несколько минут на обход этого списка, пока не нашли то, что искали. И нужно вернуться обратно в точку, где была выполнена команда `:vimgrep`. Как это сделать?

Мы могли бы воспользоваться командой `<C-o>`, чтобы обойти список переходов в обратном порядке, но это может потребовать некоторого времени. В этой ситуации нам очень пригодилась бы глобальная метка. Если перед командой `:vimgrep` выполнить команду `mM`, мы сможем вернуться обратно одним движением, выполнив команду `` `M ``.

Советы редко воспринимаются положительно, когда даются в форме: «Вы должны были предварительно выполнить команду *X*». Глобальные метки приносят пользу, только когда они устанавливаются предварительно. С опытом вы научитесь отличать ситуации, когда было бы полезно установить глобальную метку.

Попробуйте выработать привычку устанавливать глобальную метку всегда, когда собираетесь использовать команды, взаимодействующие со списком результатов, такие как `:grep`, `:vimgrep` и `:make`. То же относится к командам, взаимодействующим со списками буферов и аргументов, таким как `:args{arglist}` и `:argdo` (см. рецепт 38 в главе 6).

Помните, что вы можете установить до 26 глобальных меток, чего более чем достаточно. Не ограничивайте себя в их использовании; устанавливайте глобальные метки всякий раз, когда видите что-то, к чему наверняка потребуется вернуться позднее.

Часть IV. РЕГИСТРЫ

Регистры Vim – это всего лишь контейнеры для хранения текста. Их можно использовать как своеобразные буферы обмена, копируя текст в регистры и вставляя его из регистров, или для записи макросов, сохраняя в них последовательности нажатий на клавиши. В этой части книги мы познакомимся с основными возможностями регистров.

Глава 10. Копирование и вставка

Функции вырезания, копирования и вставки в редакторе Vim отличаются от тех, что вы привыкли использовать. В первую очередь отличается используемая терминология, как описывается во врезке «Терминология редактора Vim» ниже. В рецепте 60 мы узнаем, как пользоваться командами удаления, копирования и вставки в наиболее типичных ситуациях.

Вместо единственного общесистемного буфера обмена Vim поддерживает несколько десятков регистров, где можно сохранить фрагменты текста. Поближе с регистрами Vim мы познакомимся в рецепте 61. Команда вставки в редакторе Vim различает режимы вставки строк и символов, как будет показано в рецепте 63. Она проявляет некоторые интересные особенности при использовании в визуальном режиме, как будет описано в рецепте 62.

Наконец, в рецепте 64 мы узнаем, как использовать системную команду вставки в Vim и избежать появления странных эффектов.

Рецепт 60. Удаление, копирование и вставка с применением неименованного регистра

Команды удаления, копирования и вставки в редакторе Vim были созданы с целью упрощения решения типичных задач. Мы познакомимся с некоторыми из них, которые легко решаются с применением неименованного регистра, а затем рассмотрим задачи, требующие более глубокого понимания особенностей работы регистров.

Обычно в разговорах об операциях вырезания, копирования и вставки подразумевается буфер обмена. При работе с редактором Vim чаще всего не требуется использовать буфер обмена – эту роль вместо него выполняют *регистры*. В рецепте 61 мы увидим, что Vim имеет множество регистров, и узнаем, что имеем возможность вы-

бирать тот или иной регистр по желанию. Но для начала познакомимся с *неименованным регистром*.

Перестановка символов

Я постоянно допускаю орфографические ошибки в некоторых словах. Со временем я могу заметить, что допускаю одну и ту же ошибку, и приучить себя не допускать ее. Но некоторые ошибки, скорее, относятся к случайным опечаткам. Чаще всего я допускаю опечатку, непреднамеренно меняя порядок следования символов. Редактор Vim позволяет легко исправлять подобные ошибки.

Представьте, что, набирая название этой книги, мы допустили такую ошибку, поменяв два символа местами:

Нажатия клавиш	Содержимое буфера
`{start}`	`Practica lvim`
`F␣`	`Practica lvim`
`x`	`Practicalvim`
`p`	`Practical vim`

Здесь мы нажали клавишу пробела слишком рано, но эту ошибку легко исправить. Команда `F␣` поместит курсор на первый из двух символов, которые требуется поменять местами (см. рецепт 50 в главе 8). Команда `x` вырежет символ под курсором, сохранив его в неименованном регистре. А следующая команда `p` вставит содержимое неименованного регистра в позицию правее курсора.

Обе команды вместе `xp` можно рассматривать как единую команду: «Переставить следующие два символа».

Перестановка строк

Точно так же просто выполняется перестановка строк. Только вместо команды `x`, вырезающей текущий символ, следует использовать команду `dd`, вырезающую текущую строку:

Нажатия клавиш	Содержимое буфера

`{start}`	`2) line two`
	`1) line one`
	`3) line three`

`dd`	`1) line one` `3) line three`

`p`	`1) line one`
	`2) line two`
	`3) line three`

Команда `p` понимает, что на этот раз она имеет дело с целыми строками, поэтому результат получается вполне ожидаемый: она вставляет содержимое неименованного регистра после текущей *строки*. Напомню, что в предыдущем примере, где демонстрировалась последовательность нажатий `xp`, команда `p` вставляла содержимое регистра после текущего *символа*.

Последовательность `ddp` можно рассматривать как единую команду: «Переставить местами следующие две строки».

Создание дубликатов строк

Если потребуется создать новую строку текста, содержимое которой повторяет содержимое предыдущей строки с небольшими отличиями, мы можем начать с создания дубликата строки и использовать его как шаблон. Эту операцию в Vim можно выполнить, применив команду копирования строки с последующей командой вставки:

Нажатия клавиш	Содержимое буфера
`{start}`	`1) line one` `2) line two`

`yyp`	`1) line one`
	`2) line two`
	`2) line two`

Обратите внимание на сходство этих двух последовательностей, `ddp` и `yyp`. Первая производит вырезание и вставку строки, фактически выполняя перестановку двух строк местами. А вторая производит копирование и вставку строки, фактически дублируя ее.

Ой! Я затер свою копию

До сих пор команды удаления, копирования и вставки выглядели вполне просто и понятно. Они существенно упрощают решение многих типичных задач, возникающих при редактировании. А теперь рассмотрим ситуацию, когда эти команды работают не так слаженно, как хотелось бы. Взгляните на следующий фрагмент:

copy_and_paste/collection.js
http://media.pragprog.com/titles/dnvim/code/copy_and_paste/collection.js

```
collection = getCollection();
process(somethingInTheWay, target);
```

Мы собираемся скопировать слово `collection` в неименованный регистр и заменить им слово `somethingInTheWay`. В табл. 10.1 приводится первая попытка.

Изначально курсор уже находится на слове, которое нужно скопировать, поэтому мы можем скопировать его в неименованный регистр командой `yiw`.

Затем мы перемещаем курсор в место, куда требуется вставить слово `collection`, но перед вставкой слова мы захотели очистить место для него и выполнили команду `diw`, чтобы удалить `somethingInTheWay`.

Таблица 10.1. Копирование и вставка – первая попытка

Нажатия клавиш	Содержимое буфера
`yiw`	`collection = getCollection();` `process(somethingInTheWay, target);`
`jww`	`collection = getCollection();` `process(somethingInTheWay, target);`
`diw`	`collection = getCollection();` `process(, target);`
`P`	`collection = getCollection();` `process(somethingInTheWay, target);`

Теперь можно нажать клавишу `P`, чтобы вставить содержимое неименованного регистра перед курсором. Но... вместо слова `collection`, скопированного прежде, в текст было вставлено слово `somethingInTheWay`. Что же случилось?

Команда `diw` не просто удаляет слово – она копирует его в неименованный регистр. Или, используя более привычную терминологию, `diw` *вырезает* слово. (См. врезку «Терминология редактора Vim» ниже).

Теперь понятно, что именно было сделано неправильно. Выполнив команду `diw`, мы затерли содержимое неименованного регистра. Поэтому, нажав клавишу `P`, мы снова вставили только что удаленное слово вместо скопированного ранее.

Чтобы решить эту проблему, необходимо получить более полное представление об особенностях работы регистров.

Рецепт 61. Знакомство с регистрами Vim

Вместо единственного буфера обмена, используемого всеми операциями вырезания, копирования и вставки, Vim предоставляет воз-

можность использовать множество регистров. Выполняя команды удаления, копирования и вставки, мы можем указать, какой регистр следует использовать.

Адресация регистров

Команды удаления, копирования и вставки взаимодействуют с одним из регистров Vim. Выполняя эти команды, можно указать, какой регистр использовать, предварив их префиксом `"{register}`. Если регистр не указан явно, Vim будет использовать неименованный регистр.

Терминология редактора Vim

Термины «вырезать» (cut), «скопировать» (copy) и «вставить» (paste) просты и понятны, и описываемые ими операции доступны во многих окружениях рабочего стола и операционных системах. Редактор Vim также поддерживает аналогичные операции, но использует иную терминологию: «удалить» (delete), «захватить» (yank) и «вложить» (put).

Команда Vim «вложить» (put) полностью идентична операции вставки (paste) operation. К счастью, оба слова в английском языке начинаются с буквы «p», поэтому мнемонически команда действует одинаково, независимо от используемой терминологии.

Команда Vim «захватить» (yank) эквивалентна операции копирования (copy). Так сложилось, что команда `c` традиционно была связана с операцией *change* (изменить), поэтому авторам `vi` пришлось подыскивать альтернативное имя. Свободной оставалась клавиша `y`, поэтому операция копирования (copy) превратилась в команду *yank* (захватить).

Команда удаления (delete) в Vim эквивалентна стандартной операции вырезания (cut). То есть она копирует указанный текст в регистр и затем удаляет его из документа. Особенность работы этой команды очень важно понимать, чтобы не попасть в ловушку, описанную в разделе «Ой! Я затер свою копию» выше.

Возможно, вам интересно узнать, какая команда в Vim просто удаляет текст, то есть команда, которая удаляет текст, не копируя его в какой-либо регистр. Для этих целей в Vim предусмотрен специальный регистр, называемый *«черной дырой»*, который ничего не возвращает. Этот регистр адресуется символом подчеркивания (_) (см. `:h quote_` http://vimdoc.sourceforge.net/htmldoc/change.html#quote_), соответственно, простое удаление выполняется командой `"_d{motion}`.

Например, если потребуется захватить текущее слово в регистр `a`, можно нажать последовательность `"ayiw`. Или если потребуется вырезать текущую строку в регистр `b`, можно нажать последователь-

ность `"bdd`. После этого вставить слово из регистра `a` можно будет нажатием последовательности `"ap`, а вставить строку из регистра `b` – нажатием последовательности `"bp`.

В дополнение к командам командного режима Vim предоставляет также команды Ex удаления, захвата и вложения. Вырезать текущую строку в регистр `c` можно командой `:delete c`, а вставить ее за текущей строкой – командой `:put c`. Эти команды могут показаться слишком многословными в сравнении с командами командного режима, но они с успехом могут использоваться в комбинации с другими командами Ex и в сценариях Vim. Например, в рецепте 100 (глава 15) демонстрируется, как можно использовать команду `:yank` совместно с командой `:global`.

Неименованный регистр ("")

Если регистр не указывается, по умолчанию используется неименованный регистр, адресуемый символом `"` (см. `:h quote_quote` http://vimdoc.sourceforge.net/htmldoc/change.html#quote_quote). Чтобы адресовать этот регистр явно, следует добавить две двойные кавычки, например: команда `""p` полностью эквивалентна команде `p`.

Команды `x`, `s`, `d{motion}`, `c{motion}` и `y{motion}` (а также их эквиваленты в верхнем регистре) все сохраняют текст в неименованном регистре. Каждой из них можно явно указать другой регистр, добавив префикс `"{register}`, но все они по умолчанию используют неименованный регистр. Именно поэтому так легко затереть содержимое неименованного регистра и столкнуться с проблемами при неосторожном обращении с этими командами.

Вернемся к примеру из раздела «Ой! Я затер свою копию» выше. Мы начали с того, что захватили в регистр некоторый текст (слово «collection») с намерением вставить его в другое место. Прежде чем вставить слово, мы решили очистить место для него, удалив некоторый текст, в результате чего мы затерли содержимое неименованного регистра. Когда мы выполнили команду `p`, она вставила только что *удаленный* текст, а не текст, *захваченный* ранее.

Выбор терминологии в Vim был сделан не совсем удачно. Команды `x` и `d{motion}` часто называют командами «удаления». Это не совсем так. Их следует интерпретировать как команды «вырезания» (cut). Неименованный регистр часто содержит не тот текст, который я ожидал, но, к счастью, в нашем распоряжении имеется регистр захвата (с которым мы познакомимся ниже).

Регистр захвата («0)

При выполнении команды y{motion} текст копируется не только в неименованный регистр, но также в дополнительный регистр захвата, который адресуется символом 0 (см. :h quote0 ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#quote0).

Как следует из названия, регистр захвата изменяется только командой y{motion}. Иными словами, его содержимое не изменяется командами x, s, c{motion} и d{motion}. Если вы захватили некоторый текст, то можете быть уверены, что он сохранится в регистре 0, пока явно не будет затерт другой командой захвата. Регистр захвата обеспечивает надежность в ситуациях, когда нельзя положиться на неименованный регистр.

Регистр захвата с успехом можно использовать для решения проблемы, с которой мы столкнулись в разделе «Ой! Я затер свою копию»:

Нажатия клавиш	Содержимое буфера
yiw	collection = getCollection(); process(somethingInTheWay, target);
jww	collection = getCollection(); process(somethingInTheWay, target);
diw	collection = getCollection(); process(, target);
"0P	collection = getCollection(); process(collection, target);

Команда diw все еще затирает неименованный регистр, но оставляет регистр захвата нетронутым. Мы без опаски можем вставить текст из регистра захвата, нажав последовательность "0P, и Vim вставит тот текст, который нам необходим.

Если попробовать исследовать содержимое неименованного регистра и регистра захвата, мы увидим в них удаленный и захваченный текст соответственно:

```
⇒ :reg "0
  --- Registers ---
  "" somethingInTheWay
  "0 collection
```

Именованные регистры ("a–"z)

В редакторе Vim имеется по одному именованному регистру для каждой буквы латинского алфавита (см. :h quote_alpha ⓘ

http://vimdoc.sourceforge.net/htmldoc/change.html#quote_alpha). Это означает, что мы можем вырезать (`"ad{motion}`), скопировать (`"ay{motion}`) или вставить (`"ap`) 26 фрагментов текста.

Для решения проблемы из раздела «Ой! Я затер свою копию» мы с успехом могли бы использовать именованные регистры:

Нажатия клавиш	Содержимое буфера
`"ayiw`	`collection = getCollection();` `process(somethingInTheWay, target);`
`jww`	`collection = getCollection();` `process(somethingInTheWay, target);`
`diw`	`collection = getCollection();` `process(, target);`
`"aP`	`collection = getCollection();` `process(collection, target);`

Использование именованного регистра требует дополнительных нажатий на клавиши, поэтому для такой простой ситуации, как в данном примере, лучше использовать регистр захвата (`"0`). Удобство именованных регистров начинает проявляться, когда требуется вставить несколько фрагментов текста в разные места.

Когда производится ссылка на именованный регистр с помощью буквы нижнего регистра, команда *затирает* содержимое указанного регистра, а при ссылке с помощью буквы верхнего регистра команда добавляет новый текст *в конец* содержимого регистра. Загляните в рецепт 100 (глава 15), где демонстрируется прием добавления нового текста в конец имеющегося содержимого регистра.

Регистр «черной дыры» ("_)

Регистр «черной дыры» – это место, откуда ничего не возвращается. Он адресуется символом подчеркивания (см. `:h quote_` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#quote_). Если выполнить команду `"_d{motion}`, Vim удалит указанный фрагмент без сохранения его копии. Это может пригодиться, когда желательно удалить текст так, чтобы не затереть содержимое неименованного регистра.

Мы могли бы использовать регистр «черной дыры» для решения проблемы из раздела «Ой! Я затер свою копию»:

Нажатия клавиш	Содержимое буфера
`yiw`	`collection = getCollection();` `process(somethingInTheWay, target);`
`jww`	`collection = getCollection();` `process(somethingInTheWay, target);`
`"_diw`	`collection = getCollection();` `process(, target);`
`P`	`collection = getCollection();` `process(collection, target);`

Системный буфер обмена ("+) и регистр выделенного фрагмента ("*)

Все регистры, обсуждавшиеся выше, доступны только внутри редактора Vim. Но если потребуется скопировать некоторый текст внутри Vim и вставить его в другой программе (или наоборот), придется использовать один из системных буферов обмена.

Регистр Vim, адресуемый символом «плюс» (+), является ссылкой на системный буфер обмена (см. `:h quote+` ⓘ http://vimdoc.sourceforge.net/htmldoc/gui_x11.html#quote+).

Если некоторый фрагмент текста был скопирован или вырезан в буфер обмена в какой-то внешней программе, его можно вставить в редакторе Vim командой `"+p` (или `<C-r>+` в режиме вставки). Напротив, если дополнить команды Vim захвата или удаления префиксом `"+`, соответствующий фрагмент текста будет сохранен в системном буфере обмена. То есть его легко можно будет вставить в другом приложении.

Оконная система X11 имеет еще один буфер обмена, который называется *первичным* (primary). В нем сохраняется последний фрагмент текста, выделенный мышью. Вставить текст из этого буфера обмена можно щелчком средней кнопки мыши (или одновременным нажатием левой и правой кнопок, если мышь двухкнопочная). С этим буфером обмена в Vim связан регистр `"*` (`:h quotestar` ⓘ http://vimdoc.sourceforge.net/htmldoc/gui.html#quotestar).

Регистр	Описание
`"+`	Буфер обмена, используется командами вырезания, копирования и вставки
`"*`	Первичный буфер обмена X11, используется при выполнении щелчка средней кнопки мыши

В Windows и Mac OS X первичный буфер обмена не поддерживается, поэтому в этих операционных системах регистры `"+` и `"*`

можно использовать взаимозаменяемо: они оба представляют системный буфер обмена.

Редактор Vim может быть скомпилирован без поддержки первичного буфера обмена X11. Чтобы выяснить это, выполните команду `:version` и поищите в ее выводе строку `xterm_clipboard`. Если ей предшествует знак «минус», ваша версия Vim не поддерживает эту особенность. Знак «плюс» означает, что эта функциональная возможность поддерживается.

Регистр выражений ("=)

Регистры Vim можно представлять как простые контейнеры, хранящие блоки текста. Исключением из этого правила является регистр выражений, адресуемый символом `=` (`:h quote=` ℹ http://vimdoc.sourceforge.net/htmldoc/change.html#quote=). При извлечении содержимого из регистра выражений Vim переходит в режим командной строки, отображая приглашение к вводу `=`. В этой строке можно ввести выражение с соблюдением правил синтаксиса Vim и нажать клавишу `<CR>`, чтобы выполнить его. Если выражение вернет строку (или значение, которое может быть преобразовано в строку), Vim будет использовать ее.

Примеры использования регистра выражений можно найти в рецепте 16 (глава 3), в рецептах 96 и 95 (глава 14) и в рецепте 71 (глава 11).

Дополнительные регистры

В командах удаления и захвата можно явно указывать неименованный регистр, регистр захвата и именованные регистры. Кроме того, Vim поддерживает еще несколько регистров, в которых текст сохраняется неявно. Они известны под общим названием *регистры только для чтения* (`:h quote.` ℹ http://vimdoc.sourceforge.net/htmldoc/change.html#quote.). Они перечислены в следующей таблице:

Регистр	Описание
`"%`	Имя текущего файла
`"#`	Имя альтернативного файла
`".`	Последний вставленный текст
`":`	Последняя команда Ex
`"/`	Последний шаблон поиска

Технически регистр `"/` доступен не только для чтения – его содержимое можно определить явно, выполнив команду `:let` (см. `:h quote/` http://vimdoc.sourceforge.net/htmldoc/change.html#quote), тем не менее я включил его в таблицу для полноты картины.

Рецепт 62. Замена выделенного текста содержимым регистра

При работе в визуальном режиме команда вложения (вставки) приобретает не совсем обычные свойства. В этом рецепте мы посмотрим, как можно использовать их в своих интересах.

При попытке выполнить команду `p` в визуальном режиме Vim *заменит* выделенный фрагмент содержимым указанного регистра (см. `:h v_p` http://vimdoc.sourceforge.net/htmldoc/change.html#v_p). Эту особенность можно использовать для решения проблемы из раздела «Ой! Я затер свою копию»:

Нажатия клавиш	Содержимое буфера
`yiw`	`collection = getCollection();` `process(somethingInTheWay, target);`
`jww`	`collection = getCollection();` `process(somethingInTheWay, target);`
`ve`	`collection = getCollection();` `process(somethingInTheWay, target);`
`P`	`collection = getCollection();` `process(collection, target);`

Это мое любимое решение данной конкретной проблемы. Оно позволяет избежать неприятностей, связанных с использованием неименованного регистра обеими операциями, захвата (копирования) и вложения (вставки), потому что здесь отсутствует этап удаления. В данном решении команды удаления и вставки объединены в один шаг – замену выделенного фрагмента.

Однако этот прием имеет один побочный эффект. Попробуйте нажать `u`, чтобы отменить последнее изменение, затем `gv`, чтобы повторно выделить фрагмент текста и снова нажать `p`. Что произошло? Абсолютно ничего!

Для получения желаемого результата повторно необходимо нажать последовательность `"0p`, чтобы заменить выделенный фрагмент содержимым регистра захвата. В первый раз проблема не проявилась просто потому, что неименованный регистр, так получилось,

содержал желаемый текст. Во второй раз в неименованном регистре оказался замещенный текст.

Чтобы продемонстрировать необычность ситуации, рассмотрим воображаемый API модели стандартных команд вырезания, копирования и вставки. Этот API включает методы `setClipboard()` и `getClipboard()`. Операции вырезания и копирования вызывают метод `setClipboard()`, а операция вставки вызывает метод `getClipboard()`. При выполнении в визуальном режиме команда `p` извлекает содержимое из неименованного регистра и затем сохраняет в нем новое значение.

Проще говоря: выделенный фрагмент текста *меняется местами* с текстом в регистре. Что это? Достоинство? Недостаток? Решать вам!

Как поменять слова местами

Эту особенность поведения операции вставки в визуальном режиме можно использовать в своих интересах. Допустим, что нам требуется изменить порядок следования двух слов, чтобы они читались как «fish and chips»:

Нажатия клавиш	Содержимое буфера
`{start}`	`I like chips and fish.`
`fc`	`I like chips and fish.`
`de`	`I like  and fish.`
`mm`	`I like  and fish.`
`ww`	`I like  and fish.`
`ve`	`I like  and fish.`
`p`	`I like  and chips.`
`` `m ``	`I like  and chips.`
`P`	`I like fish and chips.`

Мы выполнили команду `de`, чтобы вырезать слово «chips» и скопировать его в неименованный регистр. Затем мы выделили слово «fish», чтобы заменить его. Когда мы выполнили команду `p`, слово «chips» появилось в документе, а слово «fish» было скопировано в неименованный регистр. Затем мы вернулись к пустому месту и вставили слово «fish» из неименованного регистра.

В данном случае быстрее было бы удалить «chips and fish» и затем ввести «fish and chips», воспользовавшись командой `c3w`, к примеру. Но этот прием я продемонстрировал по той простой причине, что его с успехом можно использовать, чтобы поменять местами более длинные фразы.

Команда `m{char}` устанавливает метку, а команда `` `{char} `` выполняет переход к метке. За дополнительной информацией обращайтесь к рецепту 54 в главе 8.

Рецепт 63. Вставка из регистра

В командном режиме команда вставки может действовать по-разному, в зависимости от природы вставляемого текста. Для вставки содержимого как последовательности символов или как целых строк можно использовать разные стратегии.

В рецепте 60 выше было показано, как командой `xp` поменять местами два символа, а командой `ddp` – две строки. В обоих случаях используется одна и та же команда `p`, но действует она немного по-разному.

Команда `p` вставляет текст из регистра в позицию правее курсора (`:h p` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#p). В дополнение к ней Vim поддерживает команду `P` (в верхнем регистре), которая вставляет текст левее курсора. В зависимости от содержимого регистра позиции *левее* и *правее* курсора могут интерпретироваться по-разному.

В случае с командой `xp` регистр содержит единственный символ, поэтому команда `p` вставит его непосредственно после *символа, на котором находится курсор*.

В случае с командой `ddp` регистр содержит полную строку, поэтому команда `p` вставит содержимое регистра *после строки, в которой находится курсор*.

Как понять, вставит ли команда `p` содержимое регистра правее текущего символа или после текущей строки? Это зависит от того, какое содержимое было сохранено в регистре. Операция захвата или удаления строки (такая как `dd`, `yy` или `dap`) сохранит в регистре полную строку, операция захвата или удаления символа (такая как `x`, `diw` или `das`) сохранит в регистре символ. В общем случае результат действия команды `p` очевиден из контекста (см. `:h linewise-register` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#linewise-register).

Вставка последовательностей символов

Допустим, что регистр по умолчанию содержит текст `collection` и нам необходимо вставить его в вызов метода на место первого ар-

гумента. Выбор команды между p и P зависит от позиции курсора. Взгляните на следующее содержимое буфера:

```
collection = getCollection();
process(, target);
```

И сравните его с:

```
collection = getCollection();
process(, target);
```

В первом случае следует использовать команду p, а во втором – команду P. На мой взгляд, ситуация не так очевидна, как кажется. В действительности я допускаю подоные ошибки настолько часто, что последовательности puP и Pup у меня выполняются уже на уровне рефлекса!

Я не люблю задумываться над тем, куда будет вставлена последовательность символов – левее или правее курсора. Поэтому я иногда предпочитаю вставлять символы в режиме вставки с помощью команды <C-r>{register} и избегаю пользоваться командами p и P в командном режиме. При использовании этого приема текст вставляется левее курсора, как если бы он вводился вручную.

Вставить текст из неименованного регистра в режиме вставки можно командой <C-r>", а из регистра захвата – командой <C-r>0 (см. рецепт 15 в главе 3). Этот прием можно использовать и для решения проблемы из раздела «Ой! Я затер свою копию»:

Нажатия клавиш	Содержимое буфера
yiw	collection = getCollection(); process(somethingInTheWay, target);
jww	collection = getCollection(); process(somethingInTheWay, target);
ciw<C-r>0<Esc>	collection = getCollection(); process(collection, target);

Применение команды ciw дает нам дополнительное преимущество: после ее выполнения команда «точка» будет заменять текущее слово словом «collection».

Вставка строк

Если в регистре сохранена целая строка, команды p и P будут вставлять текст ниже или выше текущей строки. Такое поведение более понятно, чем в случае вставки символов.

Стоит также отметить, что редактор Vim дополнительно поддерживает команды `gp` и `gP`. Они тоже вставляют текст после текущей строки или перед ней, но они перемещают курсор в конец вставленного фрагмента, а не в начало. Команда `gP` особенно удобна, когда требуется продублировать группу строк, как показано ниже:

Нажатия клавиш	Содержимое буфера
`yap`	<table> <tr> <td>Keystrokes</td> <td>Buffer Contents</td> </tr> </table>
`gP`	<table> <tr> <td>Keystrokes</td> <td>Buffer Contents</td> </tr> <tr> <td>Keystrokes</td> <td>Buffer Contents</td> </tr> </table>

Скопированные строки можно использовать как шаблон, изменяя содержимое ячеек таблицы по мере необходимости. Для этой цели с успехом можно было бы использовать обе команды, `P` и `gP`, только первая из них будет оставлять курсор выше вставленного текста. Команда `gP`, напротив, переносит курсор во вторую копию, что несколько облегчает последующее изменение только что вставленного текста.

Обсуждение

Команды `p` и `P` отлично подходят для вставки многострочных фрагментов. Но при работе с короткими фрагментами, состоящими из последовательностей символов, удобнее пользоваться командой `<C-r>{register}`.

Рецепт 64. Взаимодействие с системным буфером обмена

Помимо встроенных команд вставки иногда можно использовать системные команды вставки. Однако порой эти команды могут давать неожиданные результаты при выполнении Vim внутри терми-

нала. Мы можем избежать этих проблем, включив параметр paste перед использованием системной команды вставки.

Подготовка

Этот рецепт применим только для случая, когда Vim выполняется в окне терминала, поэтому те, кто пользуется версией GVim, могут просто пропустить его. Для начала запустим Vim в терминале:

```
⇒ $ vim -u NONE -N
```

Включение параметра autoindent гарантирует избавление от малопонятных эффектов, возникающих при вставке текста из системного буфера обмена:

```
⇒ :set autoindent
```

Наконец, необходимо скопировать следующий код в системный буфер обмена. Копирование текста из документа в формате PDF может давать странные результаты, поэтому я рекомендую загрузить примеры кода для книги, открыть файл в другом текстовом редакторе (или в веб-браузере) и уже оттуда выполнить копирование в системный буфер обмена:

copy_and_paste/fizz.rb
http://media.pragprog.com/titles/dnvim/code/copy_and_paste/fizz.rb

```
[1,2,3,4,5,6,7,8,9,10].each do |n|
  if n%5==0
    puts "fizz"
  else
    puts n
  end
end
```

Вызов системной команды вставки

В ходе этого рецепта мы постоянно будем встречать упоминание о *системной команде вставки*, и для ее вызова вы можете использовать комбинацию клавиш, поддерживаемую вашей системой. В OS X системная команда вставки вызывается комбинацией клавиш Cmd-v. Ее можно использовать при работе в окне терминала или в MacVim. Эта команда вставляет содержимое системного буфера обмена.

В Linux и Windows все не так просто – системная команда вставки обычно вызывается комбинацией `Ctrl`-`v`. В командном режиме эта комбинация вызывает переход в режим визуального блока (рецепт 21 в главе 4), а в режиме обеспечивает вставку символов буквально или по их кодам (рецепт 17 в главе 3).

Некоторые версии эмуляторов терминала в Linux поддерживают модифицированную версию команды `Ctrl`-`v` для вставки текста из системного буфера обмена. Она может быть привязана к комбинации `Ctrl`-`Shift`-`v` или `Ctrl`-`Alt`-`v`, в зависимости от системы. Не волнуйтесь, если вам не удастся выяснить, с какой комбинацией клавиш связана системная команда вставки. В крайнем случае вы можете пользоваться регистром `"*`.

Использование системной команды вставки в режиме вставки

Если переключиться в режим вставки и вызвать системную команду вставки, получится странный результат:

```
[1,2,3,4,5,6,7,8,9,10].each do |n|
  if n%5==0
      puts "fizz"
         else
      puts n
         end
         end
```

Что-то нарушило выравнивание текста. Когда системная команда вставки используется в режиме вставки, Vim действует так, как если бы символы вводились вручную. Когда параметр `autoindent` включен, Vim сохраняет отступы при создании каждой новой строки. Начальные пробелы в каждой строке из буфера обмена добавляются к автоматически установленным отступам, и в результате каждая следующая строка смещается все дальше и дальше вправо.

Версия редактора GVim отличает ситуацию, когда текст вставляется из буфера обмена и корректирует такое поведение, но когда Vim выполняется внутри терминала, эта информация ему недоступна. Параметр `paste` позволяет нам явно предупредить Vim, что мы будем использовать системную команду вставки. Когда параметр `paste` включен, Vim отключает в режиме вставки все привязки клавиш и сокращения и сбрасывает некоторые основные параметры, включая `autoindent` (см. `:h 'paste'` ⓘ http://vimdoc.sourceforge.

net/htmldoc/options.html#'paste'). Это дает возможность безопасно вставлять текст из системного буфера обмена без неожиданных сюрпризов.

Завершив использование системной команды вставки, можно снова отключить параметр `paste`. Для этого необходимо переключиться в командный режим и затем выполнить команду Ex `:set paste!`. Не кажется ли вам, что было бы здорово иметь возможность переключать этот параметр, не покидая режима вставки?

Особенности поведения редактора Vim при включенном параметре `paste` не позволят использовать привычный способ привязки этой команды к какой-либо комбинации клавиш и ее применения в режиме вставки. Но есть другой путь: мы можем связать с комбинацией клавиш изменение параметра `pastetoggle` (`:h 'pastetoggle'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'pastetoggle'):

```
⇛ :set pastetoggle=<f5>
```

Попробуйте выполнить эту команду: она свяжет с клавишей `<f5>` переключение параметра `paste`. Она должна работать в обоих режимах – в командном режиме и в режиме вставки. Если вам понравится пользоваться этой клавишей, добавьте строку выше в свой файл *vimrc*.

Используйте регистр "+, чтобы исключить необходимость переключения параметра paste

При использовании версии Vim, поддерживающей интеграцию с системным буфером обмена, можно вообще избежать необходимости вспоминать про параметр `paste`. Команда `"+p` командного режима вставляет содержимое регистра `"+`, являющегося отражением системного буфера обмена (см. раздел «Системный буфер обмена ("+) и регистр выделенного фрагмента ("*)» выше). Эта команда сохраняет оригинальные отступы в тексте и не принесет неприятных сюрпризов, независимо от состояния параметров `paste` и `autoindent`.

Глава 11. Макросы

Редактор Vim поддерживает более одного способа повторения операций. Мы уже познакомились с командой «точка», очень удобной для повторения небольших изменений. Но когда требуется организовать повторение чего-то более существенного, следует использовать макроопределения Vim. С помощью механизма макросов можно записать последовательность нажатий на клавиши любой длины и затем воспроизводить ее.

Макросы идеально подходят для повторения операций над похожими строками, абзацами и даже файлами. В этой главе вы узнаете, что есть два способа многократного выполнения макросов – последовательно или параллельно – и в каком случае применяется каждый из этих способов.

При записи последовательности команд всегда есть вероятность допустить ошибку. Но это не означает, что всякий раз нужно будет начинать все сначала. Мы легко можем добавлять новые команды в конец существующего макроса. Для внесения более обширных поправок можно даже вставить макрос в документ, отредактировать последовательность команд и затем скопировать ее обратно в регистр.

Иногда возникает необходимость вставлять в текст последовательные числа. В рецепте 67 мы узнаем, как это делается с применением комбинации элементарного сценария на языке Vim и регистра выражений.

Подобно игре «Отелло» (реверси)[1], нужно потратить всего несколько минут, чтобы научиться пользоваться макросами Vim, и целую жизнь, чтобы овладеть ими. Но любой, от начинающего до эксперта, сможет извлечь немало выгод от применения этого инструмента для автоматизации задач. Давайте посмотрим, как.

---

[1] http://ru.wikipedia.org/wiki/Реверси – *Прим. перев.*

Рецепт 65. Запись и выполнение макроса

Механизм поддержки макросов позволяет записывать последовательности нажатий на клавиши и затем воспроизводить их. Данный рецепт демонстрирует, как это делается.

Многие повторяющиеся задачи включают множество изменений. При желании мы можем автоматизировать их, записав макрос и затем выполняя его многократно.

Запись последовательности команд в виде макроса

Клавиша `q` действует как своеобразная кнопка «Запись» и одновременно как кнопка «Стоп». Чтобы начать запись последовательности нажатий на клавиши, необходимо ввести `q{register}`, указав имя регистра, где будет сохранен макрос. Если все было сделано правильно, в строке состояния появится слово «recording» («запись»). После этого будет производиться запись всех выполняемых команд, пока повторно не будет нажата клавиша `q`, останавливающая запись.

Давайте рассмотрим, как это выполняется на практике:

Нажатия клавиш	Содержимое буфера

`qa`	`foo = 1`
	`bar = 'a'`
	`foobar = foo + bar`

`A;<Esc>`	`foo = 1;`
	`bar = 'a'`
	`foobar = foo + bar`

`Ivar␣<Esc>`	`var foo = 1;`
	`bar = 'a'`
	`foobar = foo + bar`

`q`	`var foo = 1;`
	`bar = 'a'`
	`foobar = foo + bar`

Последовательность `qa` запускает запись макроса в регистр `a`. Затем выполняются два изменения в первой строке: добавление в конец точки с запятой и вставка в начало слова `var`. После внесения изменений запись останавливается нажатием `q` (`:h q` ⓘ http://vimdoc.sourceforge.net/htmldoc/repeat.html#q).

Мы можем исследовать содержимое регистра, выполнив следующую команду:

```
⇛ :reg a
  --- Registers ---
  "a   A;^[Ivar ^[
```

Не самая простая последовательность для чтения, но в ней без труда угадываются команды, выполненные несколькими мгновениями раньше. Единственным сюрпризом может показаться символ `^[`, используемый для обозначения клавиши **Escape**. Подробнее об этом рассказывается во врезке «Коды клавиш в макросах» ниже.

Повторное выполнение последовательности команд вызовом макроса

Команда `@{register}` служит для выполнения содержимого указанного регистра (см. `:h @` ⓘ http://vimdoc.sourceforge.net/htmldoc/repeat.html#@). Можно также использовать команду `@@`, которая повторит макрос, вызывавшийся последним.

Например:

Нажатия клавиш	Содержимое буфера

`{start}`	`var foo = 1;`
	`bar = 'a'`
	`foobar = foo + bar`

`j`	`var foo = 1;`
	`bar = 'a'`
	`foobar = foo + bar`

`@a`	`var foo = 1;`
	`var bar = 'a';`
	`foobar = foo + bar`

`j@@`	`var foo = 1;`
	`var bar = 'a';`
	`var foobar = foo + bar;`

Мы выполнили только что записанный макрос, повторив те же изменения с каждой из последующих строк. Обратите внимание, что к первой строке мы применили команду `@a`, а затем повторно выполнили тот же макрос командой `@@`.

В этом примере мы выполнили макрос командой `j@a` (и в следующей строке – командой `j@@`). Такой подход чем-то напоминает «формулу точки». Он требует выполнить одно нажатие (`j`), чтобы перейти к следующей строке, и два нажатия (`@a`), чтобы выполнить

фактическое действие. Для начала неплохо, но у нас есть возможность для дальнейшего усовершенствования.

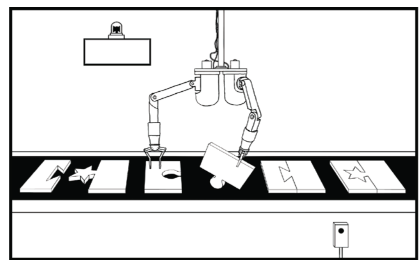
В нашем распоряжении имеется пара приемов, позволяющих выполнять макрос многократно. Подготовительные операции в обоих случаях несколько отличаются, и, что более важно, они по-разному реагируют на встречающиеся ошибки. Я постараюсь объяснить различия, опираясь на аналогию с елочными гирляндами.

Если вы приобрели недорогую елочную гирлянду, лампочки в ней, скорее всего, будут соединены последовательно. Если одна из них перегорит, светиться перестанет вся гирлянда. Если вы приобрели дорогую гирлянду, лампочки в ней наверняка будут соединены параллельно. То есть выход из строя одной лампочки не скажется на работоспособности всей гирлянды.

Я заимствовал из области электротехники слова «последовательно» и «параллельно», чтобы описать различия между двумя способами многократного выполнения макроса. Прием, опирающийся на последовательный подход, прекращает повторное выполнение макроса после первой же встретившейся ошибки. Подобно недорогой елочной гирлянде, он легко ломается. Прием, опирающийся на параллельный подход, более устойчив к ошибкам.

Последовательное выполнение макроса

На рис. 11.1 изображены робот и конвейерная лента с деталями, которые обрабатываются роботом. Запись макроса сродни процессу программирования робота на выполнение некоторой операции. На заключительном этапе мы даем роботу команду запустить конвейер и взять следующую ближайшую деталь. При таком подходе у нас может быть только один робот, выполняющий одну и ту же операцию.



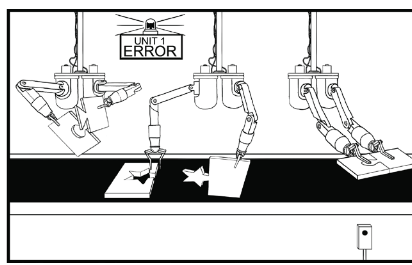


Рис. 11.1. Макросы Vim позволяют быстро выполнять повторяющиеся операции

Как следствие при последовательном подходе, столкнувшись с какими-либо неожиданностями, робот выдает сигнал тревоги и прекращает работу. Даже если на конвейерной ленте имеются детали, требующие обработки, работа все равно прекращается.

Параллельное выполнение макроса

При параллельном выполнении макроса понятие конвейера вообще исчезает. Вместо этого мы развертываем целый комплекс роботов, каждый из которых запрограммирован на выполнение одной и той же операции. Каждому дается единственная деталь. Если робот успешно справился с работой – хорошо. Если нет – ничего страшного.

В действительности Vim всегда выполняет макросы последовательно, независимо от выбранного нами подхода. Термин «параллельно» использовался, только чтобы провести аналогию с надежностью параллельного соединения. Это вовсе не означает, что Vim выполняет множество изменений параллельно.

В рецептах 68 и 70 ниже будут представлены примеры последовательного и параллельного выполнения макросов.

Рецепт 66. Товсь! Цельсь! Отставить!

Иногда в результате выполнения макроса могут получиться неожиданные результаты, но мы сможем избежать их, если будем следовать некоторым рекомендациям.

Выполняя макрос, Vim слепо повторяет записанную последовательность нажатий клавиш. Если не соблюдать меры предосторожности, результат выполнения макроса может отличаться от ожидаемого. Но у нас есть возможность конструировать более гибкие макросы, адаптирующие свое поведение в разных контекстах.

Главное правило: при записи макроса используйте только повторяемые команды.

Нормализация позиции курсора

Начиная записывать макрос, задайте себе вопрос: «Где я, откуда я пришел и куда направляюсь?» Прежде чем приступать к выполнению операций, убедитесь, что установили курсор в нужную позицию, чтобы команда выполнила необходимое действие именно там, где она должна его выполнить.

Это может подразумевать необходимость перемещения курсора к следующему совпадению (`n`), к началу текущей строки (`0`) или, может быть, к первой строке текущего файла (`gg`). Всегда тщательно прицеливайтесь, чтобы каждый раз поражать правильную цель.

Устанавливайте курсор повторяемыми командами перемещения

Редактор Vim имеет богатый набор команд перемещения курсора. Используйте их с умом.

Не пользуйтесь клавишей `l` для установки курсора в нужную позицию. Не забывайте, что Vim слепо выполняет ваши команды. Перемещение курсора вправо на десять позиций, возможно, установит курсор туда, куда нужно в момент записи макроса, но попадете ли вы в нужное место, когда будете воспроизводить макрос позднее? В другой ситуации перемещение курсора вправо на десять позиций может оказаться излишним или недостаточным.

Команды перемещения по словам, такие как `w`, `b`, `e` и `ge`, обычно позволяют добиться большей гибкости, чем команды перемещения по символам `h` и `l`. Если вы записали команду перемещения `0` и вслед за ней `e`, то смело можете ожидать получения одного и того же результата при каждом выполнении макроса. В этом случае курсор окажется на последнем символе первого слова в текущей строке. И не важно, сколько символов в этом слове, главное, чтобы строка содержала хотя бы одно слово.

Используйте команды поиска. Используйте текстовые объекты. Используйте весь арсенал команд перемещения курсора, чтобы сделать макрос максимально гибким. Помните: при записи макроса *нельзя пользоваться мышью*!

Используйте возможность прерывания при неудачном перемещении курсора

Команды перемещения курсора в редакторе Vim могут терпеть неудачу. Например, если курсор находится в первой строке в файле, команда `k` никуда не переместит его. То же относится к команде `j`, если курсор находится в последней строке в файле. По умолчанию Vim издает звуковой сигнал, когда команда перемещения терпит неудачу, который, впрочем, можно запретить с помощью параметра настройки `visualbell` (см. `:h 'visualbell'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'visualbell').

Если команда перемещения потерпит неудачу в ходе выполнения макроса, Vim прервет его выполнение. Рассматривайте это как особенность, а не как ошибку. Команды перемещения можно использовать как простой индикатор возможности выполнения макроса в текущем контексте.

Рассмотрим простой пример: начнем с того, что выполним поиск по шаблону. Допустим, что документ содержит десять совпадений. Запись макроса начинается с выполнения команды `n`, повторяющей последнюю операцию поиска. После установки курсора в начало совпадения мы выполняем некоторые операции и останавливаем запись макроса. В результате нашей правки измененный текст больше не совпадает с искомым шаблоном. Теперь документ содержит всего девять совпадений.

Когда мы запустим макрос, он перейдет к следующему совпадению и выполнит те же самые операции. Теперь в документе осталось восемь совпадений. Мы выполняем макрос снова и снова, пока не будет изменено последнее совпадение. Если теперь попытаться выполнить макрос еще раз, команда `n` потерпит неудачу из-за отсутствия совпадений и выполнение макроса прервется.

Допустим, что макрос был сохранен в регистре `a`. Вместо того чтобы десять раз выполнить команду `@a`, мы можем снабдить ее счетчиком: `10@a`. Вся прелесть этого приема в том, что от нас не требуется высокая точность. Вы не знаете, сколько будет совпадений? Не беда! Вы можете выполнить команду `100@a` или даже `1000@a` – результат будет тем же.

Рецепт 67. Выполнение со счетчиком

«Формула точки» является весьма эффективной стратегией редактирования для небольшого количества повторений, но она не позволяет использовать счетчик. Преодолеть это ограничение можно, записав небольшой макрос и выполнив его нужное количество раз.

В рецепте 3 (глава 1) мы использовали «формулу точки» для редактирования фрагмента:

the_vim_way/3_concat.js
http://media.pragprog.com/titles/dnvim/code/the_vim_way/3_concat.js

```
var foo = "method("+argument1+","+argument2+")";
```

Чтобы он выглядел так:

```
var foo = "method(" + argument1 + "," + argument2 + ")";
```

«Формула точки» подразумевает достижение желаемого результата многократным повторением команды ;.. А что, если ту же задачу требуется решить в более крупном масштабе?

```
x = ")"+a+","+b+","+c+","+d+","+e+")";
```

Мы можем применить то же самое решение. Но когда для достижения желаемого результата команду ;. требуется выполнить больше двух раз, возникает ощущение, что мы делаем лишнюю работу. Нет ли какого-нибудь способа задействовать счетчик?

Можно было бы подумать, что команда 11;. решит поставленную задачу, но это не так. Данная команда выполнит команду ; одиннадцать раз и затем один раз выполнит команду .. Похожей ошибкой было бы пытаться выполнить команду ;11., которая один раз выполнит команду ; и затем одиннадцать раз выполнит команду .. Нам же требуется одиннадцать раз выполнить последовательность ;..

Мы можем добиться желаемого, записав простейший макрос: qq;.q. Здесь команда qq включает запись макроса в регистр q. Затем вводятся требуемые команды ;., и последнее нажатие на клавишу q завершает запись макроса. Теперь мы можем выполнить макрос со счетчиком: 11@q. Эта команда выполнит ;. одиннадцать раз.

Давайте объединим все вместе (взгляните на табл. 11.1).

Команда ; повторяет поиск f+. Когда курсор окажется за последним символом + в строке, команда ; потерпит неудачу и выполнение макроса прервется.

Таблица 11.1. Запись макроса и его выполнение со счетчиком

Нажатия клавиш	Содержимое буфера
{start}	`x = "("+a+","+b+","+c+","+d+","+e+")";`
f+	`x = "("+a+","+b+","+c+","+d+","+e+")";`
s + <Esc>	`x = "(" + a+","+b+","+c+","+d+","+e+")";`
qq;.q	`x = "(" + a + ","+b+","+c+","+d+","+e+")";`
22@q	`x = "(" + a + "," + b + "," + c + "," + d + "," + e + ")";`

В данном примере достаточно было выполнить макрос десять раз. Но если выполнить его одиннадцать раз, последняя попытка будет

прервана. Иными словами, мы можем добиться требуемого результата, вызвав макрос десять или более раз.

Есть желающие точно подсчитать, сколько раз требуется выполнить этот макрос? Только не я! Лучше я укажу заведомо большое значение счетчика. Я часто использую число 22, потому что я достаточно ленив, а набрать это число на клавиатуре легче легкого. На моей клавиатуре символы `@` и `2` находятся на одной и той же клавише.

Обратите внимание, что не всегда можно позволить себе вольность со счетчиком. В данном случае это простительно, потому что макрос имеет встроенный предохранитель: команда перемещения `;` потерпит неудачу после выхода за последний символ `+` в текущей строке. Подробности смотрите в разделе «Прерывайте выполнение, если перемещение курсора не удалось» выше.

Рецепт 68. Повторение изменений в последовательности строк

Мы можем здорово упростить себе жизнь, выполняя один и тот же набор операций с группой строк, записав макрос и выполнив его для каждой строки. Добиться этого можно двумя способами: выполнить макрос последовательно или параллельно.

В качестве примера отредактируем следующий текст:

macros/consecutive-lines.txt
http://media.pragprog.com/titles/dnvim/code/macros/consecutive-lines.txt

```
1. one
2. two
3. three
4. four
```

Чтобы он выглядел так:

```
1) One
2) Two
3) Three
4) Four
```

На первый взгляд, задача выглядит тривиальной, но в действительности она имеет две интересные особенности.

Запись единицы правки

Для начала запишем все изменения, которые необходимо выполнить в первой строке:

Нажатия клавиш	Содержимое буфера
qa	1. one 2. two
0f.	1. one 2. two
r)	1) one 2. two
w~	1) One 2. two
j	1) One 2. two
q	1) One 2. two

Обратите внимание на команды перемещения курсора в этом макросе. Мы начинаем запись с команды `0`, которая нормализует позицию курсора, помещая его в начало строки. Это означает, что следующее перемещение курсора всегда будет начинаться с одного и того же места, чем обеспечивается более высокая повторяемость.

Кто-то, взглянув на следующую команду перемещения, `f.`, мог бы посчитать ее слишком расточительной. Она перемещает курсор всего на одну позицию вправо, как и команда `l`. Зачем нажимать на клавиши два раза, если достаточно одного?

Повторюсь еще раз – это сделано в угоду повторяемости. В нашем примере имеются всего четыре строки, пронумерованные от одного до четырех. А теперь представьте, что в тексте имеются строки, начинающиеся с двух цифр.

```
1. one
2. two
...
10. ten
11. eleven
```

В первых девяти строках команда `0l` переместит курсор на второй символ строки – точку. Но, начиная с десятой строки, эта же команда не будет достигать цели, тогда как команда `f.` прекрасно справится со всеми этими строками и любыми другими, начинающимися с любого количества цифр.

Кроме того, команда `f.` является еще одним «предохранителем». Если в строке отсутствует символ точки (`.`), команда `f.` закончится неудачей, и выполнение макроса прервется. Мы задействуем эту особенность чуть ниже, поэтому попридержите пока эту мысль в голове.

Последовательное выполнение макроса

Выполнить только что записанный макрос можно командой @a. Она выполнит следующие действия: переместит курсор на первый символ точки (.) в строке, заменит его скобкой), переведет в верхний регистр первую букву следующего слова и переместит курсор на следующую строку.

Чтобы решить поставленную задачу, можно трижды выполнить команду @a, но команда 3@a выглядит короче:

Нажатия клавиш	Содержимое буфера
{start}	1) One 2. two 3. three 4. four
3@a	1) One 2) Two 3) Three 4) Four

Давайте усложним задачу. Пусть последовательность строк, на которую требуется воздействовать, разбита на фрагменты случайными комментариями:

macros/broken-lines.txt
http://media.pragprog.com/titles/dnvim/code/macros/broken-lines.txt

```
1. one
2. two
// break up the monotony
3. three
4. four
```

Теперь взгляните, что получится, если применить тот же самый макрос к этому файлу (табл. 11.2).

Макрос замер на третьей строке, содержащей комментарий. Команда f. не обнаружила символа точки в этой строке, и выполнение макроса было прервано. Предохранитель сработал, и это хорошо, потому что если бы макрос продолжил выполнение в этой строке, он внес бы нежелательные для нас изменения.

Таблица 11.2. Воспроизведение макроса

Нажатия клавиш	Содержимое буфера
{start}	1. one 2. two // break up the monotony 3. three 4. four
5@a	1) One 2) Two // break up the monotony 3. three 4. four

Но задача фактически не была решена. Мы просили Vim выполнить макрос пять раз, а он сбежал на третьем повторении. Поэтому нам придется вызвать макрос еще раз, со следующей строки, чтобы выполнить работу до конца. Давайте посмотрим на альтернативное решение.

Параллельное выполнение макроса

В рецепте 30 (глава 5) демонстрировался способ применения команды «точка» к непрерывной последовательности строк. Этот же прием можно использовать и в данном случае:

Нажатия клавиш	Содержимое буфера
qa	1. one
0f.r)w~	1) One
q	1) One
jVG	1) One 2. two // break up the monotony 3. three 4. four
:'<,'>normal @a	1) One 2) Two // break up the monotony 3) Three 4) Four

Мы повторно записали макрос с самого начала. Он получился практически идентичным предыдущему, кроме того что мы опустили заключительную команду j, перемещающую курсор на следующую строку. На этот раз она не нужна.

Команда `:normal @a` выполнит макрос для каждой строки выделенной строки. Как и прежде, макрос благополучно выполнится в первых двух строках и прервется в третьей строке, но на этот раз он не остановится – он продолжит выполнение. Почему?

В предыдущем примере мы поставили в очередь пять повторений, выполнив команду `5@a`. Когда в третьем повторении возникла ошибка, она прервала выполнение всей цепочки. На этот раз мы расположили пять итераций *параллельно*. Каждый вызов макроса не зависит от результатов предыдущих вызовов. Поэтому когда третья итерация потерпела неудачу, это произошло в изоляции от других.

Выбор: последовательно или параллельно

Какой способ лучше, последовательный или параллельный? Ответ (как всегда): это зависит от ситуации.

Независимое применение макроса к нескольким элементам обеспечивает более высокую надежность. В данном примере это оказалось более удачным решением. Но если ошибки, возникающие при выполнении макроса, играют для вас важную роль, возможно, более удачным решением будет последовательное применение макроса. В последнем случае мы ясно сможем увидеть, где и почему возникают ошибки.

Попробуйте использовать оба способа, и вы разовьете в себе способность быстро определять, в каких случаях какой подход использовать.

Рецепт 69. Добавление команд в макросы

Иногда в процессе записи макроса мы по ошибке пропускаем важную операцию. Обнаружив это, не нужно бросаться тут же переписывать макрос с самого начала – мы всегда можем добавить необходимые команды в конец существующего макроса.

Представьте, что мы записали следующий макрос (заимствованный из рецепта 68):

Нажатия клавиш	Содержимое буфера
`qa`	`1. one` `2. two`
`0f.r)w~`	`1) One` `2. two`
`q`	`1) One` `2. two`

Сразу после остановки записи нажатием на клавишу `q` мы заметили, что забыли завершить выполнение командой `j` для перехода на следующую строку.

Прежде чем исправить ошибку, рассмотрим содержимое регистра `a`:

```
⇒ :reg a
  "a         0f.r)w~
```

Если ввести команду `qa`, Vim начнет запись нажатий на клавиши в регистр `a`, *затирая* прежнее содержимое этого регистра. Если ввести команду `qA`, Vim начнет запись нажатий на клавиши, *добавляя* их в конец содержимого регистра `a`. Мы можем исправить нашу ошибку так:

Нажатия клавиш	Содержимое буфера
`qA`	`1) One` `2. two`
`j`	`1) One` `2. two`
`q`	`1) One` `2. two`

Посмотрим, что теперь хранится в регистре `a`:

```
⇒ :reg a
  "a         0f.r)w~j
```

Все команды, записанные прежде, остались на месте, но теперь в конце появилась команда `j`.

Обсуждение

Эта небольшая хитрость позволяет экономить время, исключая необходимость повторной записи макроса с самого начала. Но мы можем добавлять команды только в конец макроса. Если потре-

буется добавить команду в начало или в середину макроса, мы не сможем воспользоваться этим способом. В рецепте 72 ниже описывается более мощный прием исправления макроса после его записи.

Рецепт 70. Выполнение операций над множеством файлов

До сих пор мы имели дело с операциями, повторно выполняемыми в пределах одного файла, однако макросы с тем же успехом можно применять к коллекциям файлов. В этом рецепте мы снова будем рассматривать приемы последовательного и параллельного выполнения макросов.

Допустим, что у нас имеется множество файлов, содержащих примерно такой текст:

macros/ruby_module/animal.rb
http://media.pragprog.com/titles/dnvim/code/macros/ruby_module/animal.rb

```
# ...[end of copyright notice]
class Animal
    # implementation
end
```

Нам нужно заключить определение класса в модуль, как показано ниже:

```
# ...[end of copyright notice]
module Rank
    class Animal
        # implementation...
    end
end
```

Подготовка

Для успешного воспроизведения примеров в этом рецепте добавьте следующие строки в свой конфигурационный файл:

macros/rc.vim
http://media.pragprog.com/titles/dnvim/code/macros/rc.vim

```
set nocompatible
filetype plugin indent on
set hidden
if has("autocmd")
  autocmd FileType ruby setlocal ts=2 sts=2 sw=2 expandtab
endif
```

Параметр `hidden` подробно рассматривался в разделе «Включите параметр настройки hidden перед вызовом команды :argdo или :bufdo» в главе 6.

Если вы собираетесь следовать за примерами в этом рецепте, прочитайте раздел «Загружаемые примеры» во вступлении. Комплект файлов, с которыми мы будем здесь работать, находится в каталоге *code/macros/ruby_module*.

Создание списка целевых файлов

Давайте обозначим границы наших владений, создав список целевых файлов. Мы будем перемещаться по ним с применением списка аргументов (рецепт 38 в главе 6):

```
⇒ :cd code/macros/ruby_module
⇒ :args *.rb
```

Если вызвать команду `:args` без аргументов, она выведет содержимое списка:

```
⇒ :args
  [animal.rb] banker.rb frog.rb person.rb
```

Перемещаться по списку файлов можно с помощью команд `:first`, `:last`, `:prev` и `:next`.

Запись единицы правки

Перед началом убедимся, что находимся в начале списка аргументов:

```
⇒ :first
```

Теперь запишем в макрос необходимые операции:

Нажатия клавиш	Содержимое буфера

`qa`	`# ...[end of copyright notice]`
	`class Animal`
	`    # implementation...`
	`end`

`gg/class<CR>`	`# ...[end of copyright notice]`
	`class Animal`
	`    # implementation...`
	`end`

Нажатия клавиш	Содержимое буфера
Omodule Rank<Esc>	# ...[end of copyright notice] module Rank class Animal # implementation... end

j>G	# ...[end of copyright notice]
	module Rank
	    class Animal
	        # implementation...
	    end

Goend<Esc>	# ...[end of copyright notice]
	module Rank
	    class Animal
	        # implementation...
	    end
	end

q	# ...[end of copyright notice]
	module Rank
	    class Animal
	        # implementation...
	    end
	end

Каждый из файлов начинается с упоминания об авторских правах, поэтому нам необходимо очень тщательно подойти к нормализации позиции курсора. Команда gg поместит курсор в начало файла, а команда /class<CR> передвинет его к первому совпадению со словом «class». После выполнения этих подготовительных шагов мы можем приступать к внесению изменений.

С помощью команды O мы вставляем новую строку над курсором и добавляем в нее текст. Затем переносим курсор на следующую строку, где командой >G оформляем отступы во всех строках ниже. Наконец, командой G мы переходим в конец файла, с помощью команды o создаем новую строку ниже курсора и добавляем в нее ключевое слово end.

Если вы следовали за примером, выполняя те же действия в своем редакторе, постарайтесь воспротивиться желанию немедленно сохранить изменения в файле командой :w. Почему следует воздержаться от нее, я объясню чуть ниже.

Параллельное выполнение макроса

Команда :argdo позволяет выполнить команду Ex для каждого буфера в списке аргументов (см. :h :argdo ⓘ http://vimdoc.sourceforge.

net/htmldoc/editing.html#:argdo). Но если выполнить команду `:argdo normal @a` прямо сейчас, мы получим побочный эффект.

Представьте: команда `:argdo normal @a` выполнит только что записанный макрос для всех буферов в списке аргументов, включая и первый, в который мы уже внесли изменения во время записи макроса. Как результат содержимое первого буфера будет обернуто объявлением дважды.

Чтобы избежать этого, мы отменим все изменения, выполненные в первом буфере, командой `:edit!` (см. `:h :edit!` ⓘ http://vimdoc.sourceforge.net/htmldoc/editing.html#:edit!):

```
⇛ :edit!
```

Если вы уже сохранили изменения в файле, команда `:edit!` не даст желаемого результата. В этом случае просто выполните несколько раз команду `u`, пока содержимое файла не примет первоначальный вид.

Теперь можно выполнить макрос во всех буферах в списке аргументов:

```
⇛ :argdo normal @a
```

Описанный выше прием требует выполнения подготовительных операций, но зато позволяет одной командой выполнить массу работы. А теперь посмотрим, как можно адаптировать этот макрос для последовательного выполнения.

Последовательное выполнение макроса

Наш макрос выполняет единицу правки в одном буфере. Если необходимо обеспечить его воздействие на несколько буферов, следует добавить в конец команду, выполняющую переход к следующему буферу в списке (см. табл. 11.3).

Мы могли бы вызвать команду `3@a`, чтобы выполнить макрос для каждого из оставшихся файлов в списке буферов, однако нам не нужна такая точность. Достигнув последнего буфера в списке аргументов, команда `:next` потерпит неудачу и прервет выполнение макроса. Поэтому вместо точного значения счетчика можно указать любое число, заведомо превышающее количество буферов: 22 как раз является таким числом, к тому же оно легко вводится.

Таблица 11.3. Последовательное выполнение макроса

Нажатия клавиш	Содержимое буфера

qA	`module Rank`
	`    class Animal`
	`        # implementation...`
	`    end`
	`end`

:next	`class Banker`
	`    # implementation...`
	`end`

q	`class Banker`
	`    # implementation...`
	`end`

22@a	`class Banker`
	`    # implementation...`
	`end`

Сохранение изменений во всех файлах

Мы изменили четыре файла, но пока не сохранили их. Мы можем выполнить команду `:argdo write`, чтобы сохранить сразу все файлы в списке аргументов, но существует более простой способ:

```
⇛ :wall
```

Обратите внимание, что эта команда сохранит все файлы в списке буферов, поэтому она не является точным эквивалентом команды `:argdo write` (см. `:h :wa` ⓘ http://vimdoc.sourceforge.net/htmldoc/editing.html#:wa).

Еще одна полезная команда: `:wnext` (см. `:h :wn` ⓘ http://vimdoc.sourceforge.net/htmldoc/editing.html#:wn), эквивалентная вызову команды `:write` с последующей командой `:next`. Если вы применяете макрос последовательно к нескольким файлам в списке аргументов, возможно, вы предпочтете ее.

Обсуждение

Допустим, что во время выполнения макроса в третьем буфере в списке аргументов возникла какая-то ошибка, прервавшая его. Если бы мы использовали команду `:argdo normal @a`, макрос прервался бы только в этом буфере, а если бы мы запустили макрос последовательно, указав счетчик, выполнение остановилось бы на буфере, вызвавшем ошибку, и все остальные остались бы без изменений.

Мы уже наблюдали этот эффект в рецепте 68 выше. Но на сей раз выводы получаются несколько другими. При применении одной и той же операции к последовательности строк мы можем охватить блок одним взглядом. Если что-то пошло не так, мы сразу заметим это.

На этот раз мы работали со множеством файлов, поэтому мы не можем видеть все сразу. Когда макрос выполняется последовательно и терпит неудачу, он останавливается в месте, где возникла ошибка, тогда как при параллельном выполнении макроса, чтобы обнаружить буфер, где возникла ошибка, нам придется последовательно обойти файлы в списке аргументов.

Если в процессе параллельного выполнения возникнет ошибка, обработка может закончиться раньше, но полезная информация будет скрыта.

Рецепт 71. Использование итератора для нумерации элементов в списке

Возможность вставлять значение, изменяющееся при каждом вызове макроса, может оказаться весьма полезной. В данном рецепте мы познакомимся с приемом последовательного увеличения числа с момента записи макроса, чтобы пронумеровать строки от 1 до 5.

Допустим, что нам необходимо создать нумерованный список из последовательности строк. В качестве отправной точки возьмем следующий текст:

macros/incremental.txt
http://media.pragprog.com/titles/dnvim/code/macros/incremental.txt

```
partridge in a pear tree
turtle doves
French hens
calling birds
golden rings
```

и преобразуем его в список:

```
1) partridge in a pear tree
2) turtle doves
3) French hens
4) calling birds
5) golden rings
```

Мы уже знаем пару приемов, как выполнять простые арифметические операции в Vim. Мы можем использовать команды `<C-a>` и `<C-x>` (см. рецепт 10 в главе 2) или задействовать регистр выражений (см. рецепт 16 в главе 3). В этом рецепте мы будем использовать регистр выражений и слегка коснемся языка сценариев Vim.

Простой язык сценариев Vim

Начнем с того, что последовательно выполним несколько команд в режиме командной строки. Ключевое слово `let` позволяет нам создать переменную `i` и присвоить ей значение 0. Команда `:echo` дает возможность вывести текущее значение переменной.

```
⇒ :let i=0
⇒ :echo i
  0
```

Мы можем увеличить значение `i`:

```
⇒ :let i += 1
⇒ :echo i
  1
```

Команда `:echo` отлично подходит, чтобы получить значение переменной, но в идеале нам нужен инструмент, позволяющий вставить значение в документ. Сделать это можно посредством регистра выражений. В рецепте 16 (глава 3) было показано, что регистр выражений можно использовать для вычисления простых сумм и вставки результатов в документ. Вставить значение переменной `i` можно, выполнив команду `<C-r>=i<CR>` в режиме вставки.

Запись макроса

Теперь объединим все вместе:

Нажатия клавиш	Содержимое буфера
`:let i=1`	`partridge in a pear tree`
`qa`	`partridge in a pear tree`
`I<C-r>=i<CR>)` `<Esc>`	`1) partridge in a pear tree`
`:let i += 1`	`1) partridge in a pear tree`
`q`	`1) partridge in a pear tree`

Перед началом записи макроса мы присвоили переменной `i` значение 1. Для вставки значения `i` в текст во время записи макроса

использовался регистр выражений. Затем перед остановкой записи макроса мы увеличили значение, хранящееся в переменной, которое теперь должно быть равно 2.

Выполнение макроса

Сейчас можно применить макрос к остальным строкам, как показано в табл. 11.4.

Команда `:normal @a` применит макрос к каждой строке, попавшей в выделенный фрагмент (см. раздел «Параллельное выполнение макроса» в рецепте 68). В начале процесса переменная `i` имеет значение 2, но оно увеличивается с каждым вызовом макроса. В конечном итоге к каждой выделенной строке будет добавлен ее порядковый номер.

Таблица 11.4. Нумерация элементов списка с помощью макроса

Нажатия клавиш	Содержимое буфера
{start}	1) partridge in a pear tree turtle doves French hens calling birds golden rings
jVG	1) partridge in a pear tree turtle doves French hens calling birds golden rings
:'<,'>normal @a	1) partridge in a pear tree 2) turtle doves 3) French hens 4) calling birds 5) golden rings

Для решения этой задачи также можно было бы использовать команды копирования, вставки и <C-a>. Опробовать такую возможность я оставляю вам в качестве самостоятельного упражнения!

Рецепт 72. Правка содержимого макроса

В рецепте 69 было показано, как добавлять новые команды в конец макроса. Но как быть, если потребуется удалить последнюю

команду? Или изменить что-то в начале макроса? В этом рецепте мы узнаем, как можно отредактировать содержимое макроса, как если бы это был простой текст.

Задача: Нестандартное форматирование

Допустим, что мы только что, следуя за примером в разделе «Запись единицы правки» в рецепте 67, записали макрос в регистр a. А теперь мы столкнулись с файлом, отформатированным немного иначе:

macros/mixed-lines.txt
http://media.pragprog.com/titles/dnvim/code/macros/mixed-lines.txt

```
1. One
2. Two
3. three
4. four
```

Одни строки начинаются с символов в верхнем регистре, другие – с символов в нижнем регистре. В нашем макросе мы использовали команду ~ переключения регистра символа под курсором (см. :h ~ ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#~). Давайте заменим команду ~ на команду vU, которая переводит символ под курсором в верхний регистр (см. :h v_U ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#v_U).

Коды клавиш в макросах

В данном примере мы работаем с относительно простым макросом. Но с увеличением макросов в размерах править их становится намного сложнее. Например, взгляните, как выглядит макрос, записанный в рецепте 70:

```
⇒ :reg a
  --- Registers ---
  "a      0moul<80>kb<80>kbdule Rank^[j>GGoend^[
```

Ничего странного не заметили? Прежде всего в теле макроса дважды появляется символ ^[. Не важно, какую клавишу вы нажмете, <Esc> или <C-[>, обе они будут представлены в теле макроса именно так.

Еще более странным выглядит символ <80>kb, представляющий клавишу забоя (backspace). Найдите время и изучите коды клавиш. Когда я записывал этот макрос, я ввел буквы «moul», заметил свою ошибку, дважды нажал клавишу забоя (backspace) и ввел оставшуюся часть «dule» слова «module».

Это действие практически не имеет никаких последствий. При выполнении макроса Vim воспроизведет мою ошибку и последующее ее исправление. Конечный результат получится правильным. Но подобные опечатки и исправления усложняют чтение и правку макроса.

Вставка макроса в документ

Запись макросов осуществляется в те же регистры, с которыми работают операции копирования и вставки. То есть при желании изменить макрос в регистре `a` достаточно просто вставить его в документ и отредактировать, как обычный текст.

Давайте нажмем клавишу `G` и перейдем в конец текущего документа. Нам нужно вставить содержимое регистра в новую строку. Для этого проще всего воспользоваться командой `:put`:

```
⇛ :put a
```

Почему бы не воспользоваться командой `"ap`? В данном случае команда `p` вставит содержимое регистра правее курсора в текущую строку. Команда `:put`, напротив, всегда вставляет содержимое в новую строку, ниже текущей, независимо от того, что было сохранено в регистре, последовательность символов или целая строка.

Правка макроса

Теперь мы можем отредактировать макрос, как обычный текст. Последовательность команд, показанная в табл. 11.5, замещает символ ~ парой символов `vU`.

Таблица 11.5. Правка макроса

Нажатия клавиш	Содержимое буфера
`{start}`	`0f.r)w~j`
`f~`	`0f.r)w~j`
`svU<Esc>`	`0f.r)wvUj`

Копирование макроса из документа обратно в регистр

В нашем документе имеется исправленная последовательность команд, и нам требуется скопировать ее обратно в регистр. Проще всего это сделать с помощью команды `"add` (или `:d a`), но это может

вызвать проблемы позднее. Команда dd выполняет удаление целой строки. В результате в конец макроса добавляется символ ^J:

```
⇒ :reg a
  0f.r)wvUj^J
```

Это символ перевода строки, присутствие которого в большинстве случаев не играет никакой роли. Но иногда завершающий символ перевода строки может изменять значение макроса. Для предупреждения неожиданностей всегда копируйте макросы из документа в регистры как последовательности символов:

Нажатия клавиш	Содержимое буфера
{start}	// last line of the file proper 0f.r)wvUj
0	// last line of the file proper 0f.r)wvUj
"ay$	// last line of the file proper 0f.r)wvUj
dd	// last line of the file proper

Когда вслед за командой 0 выполняется команда "ay$, она захватывает все символы в строке, кроме символа перевода строки. Сохранив все необходимое в регистре a, можно выполнить команду dd, чтобы удалить строку. Копия удаленной строки сохранится в регистре по умолчанию, но мы не собираемся использовать его.

После выполнения этих шагов в регистре окажется новый, улучшенный макрос. Мы можем использовать его, чтобы изменить фрагмент текста, представленный в начале этого рецепта.

Обсуждение

Возможность вставить макрос в документ, отредактировать его и скопировать обратно в регистр является неоспоримым удобством. Но тело макроса может оказаться трудночитаемым по причинам, описанным во врезке «Коды клавиш в макросах» выше. Если требуется лишь добавить команду в конец макроса, проще следовать процедуре, описанной в рецепте 69.

Так как регистры в редакторе Vim – не более чем контейнеры, предназначенные для хранения текста, ими можно манипулировать программно, используя сценарии на языке Vim. Например, для правки макроса, описывавшегося выше, можно было бы использовать

функцию `substitute()` (которая не имеет ничего общего с командой `:substitute`! См. `:h substitute()` ⓘ http://vimdoc.sourceforge.net/htmldoc/eval.html#substitute()):

```
⇛ :let @a=substitute(@a, '\~', 'vU', 'g')
```

Если вас заинтересовала такая возможность, загляните в справку `:h function-list` ⓘ http://vimdoc.sourceforge.net/htmldoc/usr_41.html#function-list, где можно найти дополнительную информацию.

Часть V. ШАБЛОНЫ

Эта часть книги посвящена шаблонам, являющимся составной частью некоторых, одних из самых мощных команд Vim. Мы познакомимся с некоторыми хитростями, упрощающими составление регулярных выражений и поиск литерального текста. Изучим особенности работы самого механизма поиска и затем исследуем две мощные команды Ex: `:substitute`, позволяющую находить совпадения с шаблоном и заменять их чем-то другим, и `:global`, дающую возможность выполнить любую команду Ex для каждой строки, соответствующей указанному шаблону.

Глава 12. Поиск по шаблону и поиск точного совпадения

В этой части книги мы будем знакомиться с командами поиска, подстановки и применения других команд к совпадениям. Но сначала рассмотрим общий механизм, лежащий в их основе: механизм поиска в редакторе Vim. Приходилось ли вам задаваться вопросом о том, как действуют регулярные выражения в Vim или как отключить их?

Механизм регулярных выражений в редакторе Vim может отличаться от механизма, к которому вы могли привыкнуть. В этой главе вы узнаете, что многие расхождения можно сгладить переключателем *самого волшебного режима* (very magic) интерпретации шаблонов. *По умолчанию* в командах поиска некоторые символы интерпретируются редактором Vim особым образом, что может осложнять попытки отыскать точное совпадение. Мы узнаем, как отключить интерпретацию специальных символов с помощью переключателя *самого простого режима* (very nomagic) интерпретации шаблонов.

Особое внимание будет уделено двум специальным элементам, которые можно использовать в шаблонах Vim: разделителям нулевой ширины, обозначающим границы слов или границы поиска. В заключение мы подробно обсудим особенности использования некоторых символов, сохраняющих свое специальное значение, даже при использовании переключателя `\V` *самого простого режима*.

Рецепт 73. Настройка чувствительности к регистру в шаблонах

Настраивать чувствительность к регистру символов в операциях поиска можно глобально или для каждой операции отдельно.

Глобальная настройка чувствительности к регистру

Обеспечить *нечувствительность* операций поиска к регистру символов в редакторе Vim можно с помощью параметра настройки `ignorecase`. Однако этот параметр имеет побочный эффект, влияющий на поведение функции автодополнения, как описывается во врезке «Автодополнение и чувствительность к регистру» в главе 19.

Настройка чувствительности к регистру для каждой операции поиска

Изменять чувствительность к регистру символов в шаблонах можно с помощью символов `\c` и `\C`. Символ `\c` (в нижнем регистре) выключает чувствительность к регистру, а символ `\C` (в верхнем регистре), напротив, включает чувствительность к регистру. Эти символы имеют более высокий приоритет, чем параметр настройки `ignorecase`.

Примечательно, что эти символы можно использовать в любом месте в шаблонах. Если вы обнаружите необходимость включить чувствительность к регистру уже *после* ввода полного шаблона, просто добавьте `\C` в конец, и он будет воздействовать на весь шаблон.

Интеллектуальное определение чувствительности к регистру

В Vim имеется дополнительный параметр настройки, включающий интеллектуальный механизм, пытающийся предсказать желаемый нами режим чувствительности к регистру. Этот параметр называется `smartcase`. При добавлении параметра `smartcase` в настройки он отменяет действие параметра `ignorecase` всякий раз, когда в шаблон поиска включается символ верхнего регистра. Иными словами, если шаблон содержит только символы нижнего регистра, поиск будет выполняться без учета регистра символов. Но если мы включим в шаблон символ верхнего регистра, поиск будет выполняться с учетом регистра символов.

Вам не кажется это слишком сложным? Попробуйте применить этот параметр настройки на практике, и вы поймете, что здесь нет ничего сложного. И помните, что за нами сохраняется возможность принудительно включить режим чувствительности/нечувствитель-

ности к регистру в любой операции поиска, добавив в шаблон символ `\C` или `\c`. В табл. 12.1 приводится матрица параметров, управляющих чувствительностью к регистру. Аналогичную таблицу можно найти во встроенной документации Vim (`:h /ignorecase` ⓘ http://vimdoc.sourceforge.net/htmldoc/pattern.html#/ignorecase).

Таблица 12.1. Настройка чувствительности к регистру в операциях поиска

Шаблон	ignorecase	smartcase	Совпадения
foo	выключен	–	foo
foo	включен	–	foo Foo FOO
foo	включен	включен	foo Foo FOO
Foo	включен	включен	Foo
Foo	включен	выключен	foo Foo FOO
\cfoo	–	–	foo Foo FOO
foo\C	–	–	foo

Рецепт 74. Включение поддержки регулярных выражений

Синтаксис регулярных выражений в Vim ближе к описываемому стандартом POSIX, чем к поддерживаемому языком Perl. Для программистов, уже имеющих опыт использования регулярных выражений в Perl, это обстоятельство может превратиться в источник расстройств. Однако с помощью переключателя самого волшебного режима (very magic) интерпретации шаблонов можно включить поддержку более знакомого синтаксиса регулярных выражений.

Допустим, что нам нужно сконструировать регулярное выражение, которому будут соответствовать все коды цвета в следующем фрагменте CSS:

patterns/color.css
$http://media.pragprog.com/titles/dnvim/code/patterns/color.css

```
body   { color: #3c3c3c; }
a      { color: #0000EE; }
strong { color: #000; }
```

Регулярное выражение должно совпадать с символом `#`, за которым следуют три или шесть шестнадцатеричных цифр (все десятичные цифры плюс буквы от A до F включительно, в верхнем или в нижнем регистре).

Поиск шестнадцатеричных кодов цвета в расширенном режиме

Описанным требованиям соответствует следующее регулярное выражение:

```
⇒ /#\([0-9a-fA-F]\{6}\|[0-9a-fA-F]\{3}\)
```

Попробуйте его, если хотите. Оно прекрасно работает, но взгляните на эти символы обратного слэша, которых в данном выражении целых пять штук!

В этом регулярном выражении используются три типа скобок. Квадратные скобки по умолчанию имеют особое значение, поэтому их не требуется экранировать. Круглые скобки соответствуют символам `(` и `)` буквально, поэтому их требуется экранировать, чтобы придать им особое значение. То же относится к фигурным скобкам, но обратите внимание: экранировать требуется только открывающую скобку. Закрывающую скобку можно оставить неэкранированной, и Vim поймет наши намерения. Но это *не относится* к круглым скобкам – экранировать необходимо обе скобки в паре, открывающую и закрывающую.

Каждый из трех типов скобок подчиняется разным правилам. Прочитайте предыдущий абзац еще несколько раз и зазубрите его. Я подожду. Впрочем, не стоит!

Два механизма регулярных выражений

В версии Vim 7.4 появилась поддержка нового механизма регулярных выражений (см. `:h new-regexp-engine` ⓘ http://vimhelp.appspot.com/version7.txt.html#new-regexp-engine). Если прежний механизм использует алгоритм с возвратами, то новый реализует конечный автомат, лучше справляющийся со сложными шаблонами и длинными фрагментами текста. Это усовершенствование, в свою очередь, привело к увеличению скорости работы всех функций, использующих регулярные выражения, таких как подсветка синтаксиса, команды поиска и `vimgrep`.

Новый механизм регулярных выражений действует по умолчанию в Vim 7.4, но старый механизм все еще доступен. Некоторые функции Vim, использующие регулярные выражения, не поддерживаются новым механизмом. Для таких функций Vim автоматически выбирает старый механизм. За дополнительной информацией обращайтесь в справочный раздел `:h two-engines` ⓘ http://vimhelp.appspot.com/pattern.txt.html#two-engines.

Поиск шестнадцатеричных кодов цвета в самом волшебном режиме

Мы можем унифицировать правила для всех специальных символов с помощью ключа `\v`. Он включает *самый волшебный* (very magic) режим поиска, когда все символы интерпретируются как специальные, кроме символа подчеркивания («_»), букв верхнего и нижнего регистров и цифр от `0` до `9` (см. `:h \v` ⓘ http://vimdoc.sourceforge.net/htmldoc/pattern.html#\v).

Ключ `\v` делает поведение механизма регулярных выражений Vim более похожим на механизм регулярных выражений в языках Perl, Python и Ruby. Однако существуют некоторые отличия, к которым мы не раз будем привлекать ваше внимание на протяжении всей этой главы, но запомнить их гораздо проще, чем правила экранирования символов.

Давайте перепишем регулярное выражение, соответствующее шестнадцатеричным кодам цвета, но на этот раз с применением ключа `\v`:

```
⇒ /\v#([0-9a-fA-F]{6}|[0-9a-fA-F]{3})
```

Ключ `\v` в начале регулярного выражения придает всем последующим символам специальное значение. Теперь регулярное выражение выглядит более опрятно, без всех этих символов обратного слэша, вам так не кажется?

Использование класса шестнадцатеричных цифр

Мы можем еще больше упростить регулярное выражение: вместо перечисления всех символов `[0-9a-fA-F]` можно использовать символьный класс `\x` (см. `:h /character-classes` ⓘ http://vimdoc.sourceforge.net/htmldoc/pattern.html#/character-classes). Этот шаблон действует точно так же, как и предыдущий:

```
⇒ /\v#(\x{6}|\x{3})
```

Обсуждение

В следующей таблице перечислены все регулярные выражения, созданные в этом рецепте, чтобы проще было сравнить их:

Шаблон	Примечания
`#\([0-9a-fA-F]\{6}\\|[0-9a-fA-F]\{3}\)`	Используя расширенный режим, мы вынуждены экранировать символы `(`, `)`, `|` и `{`, чтобы придать им специальное значение
`\v#([0-9a-fA-F]{6}|[0-9a-fA-F]{3})`	Применение ключа `\v` обеспечивает специальную интерпретацию символов `(`, `)`, `|` и `{` по умолчанию
`\v#(\x{6}|\x{3})`	Задействовав символьный класс `\x`, который является эквивалентом класса `[0-9A-Fa-f]`, можно сделать регулярное выражение еще короче

Одно заключительное замечание: символ # не имеет специального значения, поэтому в регулярных выражениях он соответствует сам себе. Вспомните, как выше говорилось, что в *самом волшебном режиме* (very magic) специальное значение имеют все символы, кроме подчеркивания («_»), букв и цифр. Похоже, что мы нашли исключение из этого правила!

В ответ на это Vim отвечает: «Любые символы, не имеющие специального значения, зарезервированы для расширения в будущем» (см. `:h /\\` ⓘ http://vimdoc.sourceforge.net/htmldoc/pattern.html#/\\). Иными словами, отсутствие специального значения у символа # сегодня вовсе не означает, что он не получит его в будущих версиях. Если бы символ # имел специальное значение, нам пришлось бы экранировать его, чтобы обеспечить соответствие самому себе. Однако не позволяйте этой мысли беспокоить вас по ночам.

Урок истории: наследственность синтаксиса шаблонов в Vim

Помимо синтаксисов шаблонов, поддержка которых активируется ключами `\v` и `\V`, в Vim поддерживаются еще два более старых синтаксиса. По умолчанию в Vim используется *расширенный* (magic) синтаксис, имеется также *стандартный* (nomagic) синтаксис, имитирующий поведение `vi`. Они могут включаться ключами и `\m` и `\M` соответственно.

Ключ `\M` стандартного (nomagic) режима производит эффект, аналогичный ключу `\V` литерального режима (или режима точного сопоставления), за исключением пары символов, `^` и `$`, которым автоматически придается специальное значение.

В расширенном режиме специальное значение приобретают еще несколько символов, таких как `.`, `*` и квадратные скобки. Расширенный режим поиска был реализован, чтобы упростить создание несложных регулярных выражений. Но такие символы, как `+`, `?`, круглые и фигурные

скобки, в этом режиме не имеют специального значения, поэтому их необходимо экранировать, чтобы использовать для специальных целей.

Расширенный режим остановился на полпути к достижению простоты в создании регулярных выражений. Как результат набор правил экранирования выглядит беспорядочным, что усложняет его запоминание. Ключ \v придает специальное значение всем символам, кроме _, цифр и букв. Правила этого режима легко запоминаются, и они более полно соответствуют правилам составления регулярных выражений в языке Perl.

Рецепт 75. Ключ \V включает режим поиска точного совпадения

Специальные символы регулярных выражений удобно использовать, когда поиск выполняется по шаблону, но они могут мешать, когда требуется найти точное совпадение с искомым текстом. С помощью переключателя самого простого режима (very nomagic) можно выключить интерпретацию большинства специальных символов, таких как `.`, `*` и `?`.

Возьмем для демонстрации следующий фрагмент текста:

patterns/excerpt-also-known-as.txt
http://media.pragprog.com/titles/dnvim/code/patterns/excerpt-also-known-as.txt

```
The N key searches backward...
...the \v pattern switch (a.k.a. very magic search)...
```

Допустим, что нам нужно перейти к вхождению подстроки «a.k.a.» (от англ. «also known as» – «известный также как»), выполнив ее поиск. Самой первой мыслью будет воспользоваться командой

```
⇒ /a.k.a.
```

Но, нажав клавишу **Enter**, мы обнаружим, что в этом фрагменте было найдено несколько совпадений. Символ «.» имеет специальное значение: ему соответствует любой символ. Например, слово «backward» содержит фрагмент, соответствующий нашему шаблону. Следующая таблица иллюстрирует наши попытки найти совпадение с требуемым фрагментом с первой попытки.

Нажатия клавиш	Содержимое буфера
`{start}`	`The N key searches backward...` `...the \v pattern switch (a.k.a. very magic search)...`

`/a.k.a.<CR>`	`The N key searches backward...` `...the \v pattern switch (a.k.a. very magic search)...`
`/a\.k\.a\.<CR>`	`The N key searches backward...` `...the \v pattern switch (a.k.a. very magic search)...`
`/\Va.k.a.<CR>`	`The N key searches backward...` `...the \v pattern switch (a.k.a. very magic search)...`

Попытки найти совпадение в примере выше всего лишь вызывают раздражение из-за задержки. Мы можем перейти к следующему совпадению, надеясь, что это и будет конечная цель, просто нажимая клавишу `n`. Но в некоторых ситуациях ошибочное совпадение может иметь гораздо более разрушительные последствия. Представьте, что мы собрались выполнить команду подстановки, такую как `:%s//also known as/g`, упустив из виду, что наш шаблон имеет гораздо более широкое толкование, чем мы подразумеваем (пустое поле шаблона в команде `:substitute` сообщает редактору Vim, что он должен использовать последний применявшийся шаблон, как описывается в рецепте 90 в главе 14). Это может приводить к серьезным ошибкам!

Мы можем отменить специальное значение символа точки (`.`), экранировав его. Следующий шаблон не обнаружит совпадения в слове «backward», но все еще будет находить подстроку «a.k.a.»:

```
⇛ /a\.k\.a\.
```

Как вариант можно было бы использовать ключ `\V`, включающий *самый простой режим* (very nomagic) поиска точного совпадения:

```
⇛ /\Va.k.a.
```

Как описывается в документации Vim: «при использовании ключа `\V` все специальные символы, следующие после него, утрачивают свое специальное значение, кроме символа обратного слэша» (см. `:h /\V` ⓘ http://vimdoc.sourceforge.net/htmldoc/pattern.html#/\V). Как будет показано в рецепте 79 ниже, это несколько упрощенное представление, но этого достаточно для нашего обсуждения.

В режиме поиска точного совпадения все еще возможно использовать регулярные выражения, но это достаточно утомительно, потому что требуется предварять символом обратного слэша каждый специальный символ. Обычно, когда требуется произвести поиск с применением регулярного выражения, используется ключ `\v`, а когда требуется найти точное совпадение, – ключ `\V`.

Рецепт 76. Использование круглых скобок для захвата совпадений с подвыражениями

При работе с регулярными выражениями имеется возможность сохранять совпадения с подвыражениями и затем ссылаться на них. Эта возможность особенно удобна в комбинации с командой подстановки, но ее также можно использовать для поиска повторяющихся слов.

Возьмем для демонстрации следующий фрагмент текста:

patterns/springtime.txt
http://media.pragprog.com/titles/dnvim/code/patterns/springtime.txt

```
I love Paris in the
the springtime.
```

Заметили грамматическую ошибку? Просто удивительно, как сложно порой бывает заметить такие ошибки из-за того, что наш мозг неосознанно исправляет ее сам, но она станет более заметной, если выделить ее: «Paris in *the the* springtime.». Когда последовательность из двух одинаковых слов разрывается переносом строки, наш мозг не замечает повторения. Этот эффект называется лексической иллюзией[1].

Следующее регулярное выражение позволяет находить повторяющиеся слова:

```
⇒ /\v<(\w+)\_s+\1>
```

Попробуйте выполнить поиск по этому шаблону во фрагменте текста, представленном выше, и вы увидите, что в результате поиска пара слов «the the» окажется подсвечена. Теперь попробуйте объединить строки (например, командой `vipJ`) – регулярное выражение все равно будет находить совпадение. Самое замечательное, что этот шаблон позволяет найти не только последовательность «the the», но и любых других двух одинаковых слов. Давайте разберем это регулярное выражение на составные части и посмотрим, как оно работает.

Вся хитрость поиска повторяющихся слов заключается в комбинации `()` и `\1`. Все, что совпадет с подвыражением в круглых скобках, автоматически будет сохранено во временном буфере. Мы можем сослаться на совпавший фрагмент с помощью конструкции

\1. Если шаблон содержит более одной пары круглых скобок, на совпадения с подвыражениями в каждой паре () можно сослаться с помощью конструкций \1, \2 и т. д., вплоть до \9. Элемент \0 всегда ссылается на совпадение со всем регулярным выражением, независимо от наличия в нем круглых скобок.

Регулярное выражение для поиска лексических иллюзий содержит еще несколько хитростей. Мы уже видели в рецепте 74 выше, что ключ \v включает *самый волшебный* (very magic) режим интерпретации регулярных выражений. Символы < и > соответствуют границам слова, как описывается в рецепте 77 ниже. Наконец, последовательность _s соответствует пробельным символам и разрывам строк (см. :h /_ ⓘ http://vimdoc.sourceforge.net/htmldoc/pattern.html#/_ и :h 27.8 ⓘ http://vimdoc.sourceforge.net/htmldoc/usr_27.html#27.8 соответственно).

На практике довольно редко встречаются ситуации, когда возникает необходимость использовать совпадения с подвыражениями в регулярных выражениях. Единственное, что приходит на ум: поиск открывающих и закрывающих пар тегов XML или HTML. Однако, как будет показано в рецепте 94 (глава 14), совпадения с подвыражениями могут также пригодиться в строках замены, в команде :substitute.

Рецепт 77. Границы слова

В регулярном выражении может пригодиться возможность определять границы слова. Специальные символы, представляющие границы слова, позволяют это.

Некоторые слова, особенно короткие, часто могут входить в состав более длинных слов. Например, слово «the» встречается внутри слов «these», «they», «their» и многих других. Поэтому, выполняя поиск короткого фрагмента командой /the<CR>, мы можем найти гораздо больше совпадений, чем предполагали:

```
the problem with these new recruits is that
they don't keep their boots clean.
```

Если требуется найти именно слово, а не последовательность символов «the», можно воспользоваться специальными символами, обозначающими границы слова. В *самом волшебном* (very magic) режиме поиска эту роль играют символы < и >. То есть если

для поиска в тексте, представленном выше, использовать шаблон `/\v<the><CR>`, мы найдем только одно совпадение.

Эти специальные символы имеют *нулевую ширину*. То есть они не соответствуют каким-либо символам в тексте, а представляют границы между словом и пробельными символами или знаками препинания.

Действие символов `<` и `>` можно имитировать, комбинируя классы символов `\w` и `\W` с разделителями `\zs` и `\ze` (с которыми мы познакомимся в рецепте 78 ниже). Класс `\w` соответствует любым символам слова, включая буквы, цифры и знак подчеркивания «_», а класс `\W` соответствует любым символам, кроме символов слова.

Использование круглых скобок без сохранения совпадений

Иногда бывает желательно использовать круглые скобки для группировки, и при этом не требуется сохранять совпадения с подвыражениями в них. Примером может служить следующий шаблон, совпадающий с любой формой написания моего имени:

```
⇛ /\v(And|D)rew Neil
```

Здесь круглые скобки использовались, чтобы обеспечить совпадение с формой написания «Andrew» или «Drew», но в данном случае не требуется сохранять совпадение «And» или «D» с подвыражением в круглых скобках. Мы можем сообщить Vim, что он может не сохранять совпадение в регистре `\1`, добавив перед скобками знак процента `%`:

```
⇛ /\v%(And|D)rew Neil
```

И что это нам дает? В простейшем случае поиск по такому шаблону будет выполняться быстрее, хотя вы вряд ли это заметите. Кроме того, такая возможность может пригодиться, когда вы будете использовать в шаблоне несколько пар круглых скобок. Представьте, что потребовалось заменить все вхождения `FIRSTNAME LASTNAME` на `LASTNAME, FIRSTNAME` во всех формах написания моего имени. Сделать это можно так:

```
⇛ /\v(%(And|D)rew) (Neil)
⇛ :%s//\2, \1/g
```

Регулярное выражение сохранит в регистре `\1` либо «Andrew», либо «Drew», а в регистре `\2` – «Neil». Если бы в этом выражении не использовались простые группирующие скобки `%()`, механизм поиска сохранил бы ненужный нам фрагмент и поле замены выглядело бы иначе.

Специальный символ <, например, можно выразить как \W\zs\w, а специальный символ > – как \w\ze\W.

В *самом волшебном* (very magic) режиме поиска границы слова можно обозначать неэкранированными символами < и >, но в расширенном и стандартном режимах, а также в режиме поиска точного совпадения их необходимо экранировать. Поэтому, чтобы отыскать описание этих специальных символов в справке Vim, их следует экранировать символом обратного слэша: :h /\< http://vimdoc.sourceforge.net/htmldoc/pattern.html#/\<. Имейте в виду, что если потребуется отыскать сам символ угловой скобки в *самом волшебном* (very magic) режиме поиска, его необходимо экранировать.

Даже если вы не используете явно специальные символы границ слова в своих шаблонах, вы все равно используете их косвенно, вызывая команду * или # (см. :h * http://vimdoc.sourceforge.net/htmldoc/pattern.html#star). Эти команды выполняют поиск слова под курсором вперед и назад соответственно. Если вывести историю поиска после использования этих команд (нажав /<Up>), можно увидеть, что последний шаблон поиска заключен в специальные символы границ слова. Кстати, команды g* и g# выполняют поиск слова под курсором без использования границ.

Рецепт 78. Границы совпадения

Иногда бывает желательно определить более универсальный шаблон и затем сконцентрироваться на подмножестве совпадений. Такая возможность в редакторе Vim обеспечивается элементами \zs и \ze.

До настоящего момента мы рассматривали шаблоны и совпадения с ними как единое целое. Настало время разделить эти понятия. Для начала дадим необходимые определения. Говоря о *шаблоне*, мы подразумеваем регулярное выражение (или простой текст), которое указывается в команде поиска. Говоря о *совпадении*, мы подразумеваем любой текст в документе, подсвеченный в результате выполнения операции поиска (предполагается, что параметр hlsearch включен).

Границы совпадения обычно соответствуют началу и концу шаблона. Но для сужения круга совпадений мы можем использовать специальные символы \zs и \ze (см. :h /\zs http://vimdoc.sourceforge.net/htmldoc/pattern.html#/\zs). Специальный символ \zs соответствует началу совпадения, а специальный символ \ze – концу совпадения. Вместе они позволяют определять шаблоны,

совпадающие с некоторой последовательностью символов, а затем ограничить подмножество совпадений. Так же как и границы слова (о которых рассказывалось в предыдущем рецепте), символы \zs и \ze имеют нулевую ширину.

Понять назначение этих символов проще всего на примере. Если выполнить поиск по шаблону /Practical Vim<CR>, в документе будут подсвечены все вхождения фразы «Practical Vim». Если выполнить поиск по шаблону /Practical \zsVim<CR>, подсвечены будут только вхождения слова «Vim». Слово «Practical» будет исключено из совпадений, несмотря на то что оно присутствует в шаблоне. То есть подсвечены будут только вхождения слова «Vim», следующие за словом «Practical», любые другие вхождения слова «Vim» не будут найдены. Этот результат совершенно отличается от результатов поиска по шаблону /Vim<CR>.

Ниже приводится еще один пример, на этот раз с использованием обеих границ, \zs и \ze, чтобы обозначить начало и конец совпадения:

Нажатия клавиш	Содержимое буфера
{start}	Match "quoted words"---not quote marks.
/\v"[^"]+"<CR>	Match "quoted words"---not quote marks.
/\v"\zs[^"]+\ze"<CR>	Match "quoted words"---not quote marks.

Основу шаблона составляет типичная идиома: "[^"]+". Шаблон начинается и заканчивается двойными кавычками и соответствует любому одному или более символам в них, кроме двойных кавычек. После открывающей кавычки в шаблоне стоит специальный символ \zs, а перед закрывающей – специальный символ \ze. Благодаря им кавычки исключаются из совпадения и остается только текст, содержащийся в них. Отметьте, что кавычки остаются важным элементом шаблона, даже при том, что они исключаются из совпадения.

Выражения проверок

Концептуально специальные символы \zs и \ze, поддерживаемые редактором Vim, напоминают опережающие и ретроспективные проверки в языке Perl[1]. Синтаксис регулярных выражений в Perl и Vim отличается, тем не менее легко можно заметить, что специальный символ \zs является аналогом *положительной ретроспективной проверки* в Perl, а символ \ze – аналогом *положительной опережающей проверки*.

Нетрудно догадаться, что Perl также поддерживает отрицательные варианты проверок. Эти элементы, имеющие нулевую ширину, соответствуют, только если совпадения с указанными шаблонами отсутствуют.

Редактор Vim тоже поддерживает полный набор отрицательных/положительных опережающих/ретроспективных проверок, но опять же их синтаксис отличается от синтаксиса проверок в Perl. Чтобы увидеть, как выглядят эти проверки в Perl и их аналоги в Vim, загляните в раздел справки `:h perl-patterns` http://vimdoc.sourceforge.net/htmldoc/pattern.html#perl-patterns.

Шаблон `/\v"\zs[^"]+\ze"<CR>` из рецепта 78 можно было переписать с использованием положительных проверок вместо символов `\zs` и `\ze`:

```
⇛ /\v"@<=[^"]+"@=
```

Не знаю, как вам, но мне версия с символами `\zs` и `\ze` кажется проще. Отрицательные проверки широко используются в редакторе Vim, в определениях подсветки синтаксиса, но мне кажется, что они не имеют большой практической ценности в повседневной работе. Положительным проверкам, напротив, можно найти массу применений, поэтому вполне логично, что они имеют отдельные символы, а именно `\zs` и `\ze`.

Рецепт 79. Экранирование проблемных символов

Переключатель `\V` самого простого режима упрощает поиск точного соответствия, потому что отключает специальное поведение символов `.`, `+` и `*`, но существует несколько символов, чье специальное значение нельзя отключить. В этом рецепте мы узнаем, как использовать данные символы.

Выражения позиционной проверки

Метасимволы `\zs` и `\ze` в Vim концептуально схожи с позиционными проверками в Perl (lookaround assertions)[1]. Несмотря на то что синтаксис регулярных выражений в Perl и Vim отличается, метасимвол `\zs` можно считать примерным эквивалентом *позитивной ретроспективной проверки*, а метасимвол `\ze` – эквивалентом *позитивной опережающей проверки*.

Как нетрудно догадаться, Perl поддерживает также негативные варианты позиционных проверок. Это элементы нулевой ширины, которые совпадают, только если заданный шаблон отсутствует в тексте. Vim так же поддерживает полный спектр негативных/позитивных опережающих/ретроспективных проверок, но его синтаксис регулярных выра-

[1] http://www.regular-expressions.info/lookaround.html

жений отличается от используемого в языке Perl. Наглядное сравнение можно найти в разделе справки `:h perl-patterns` ⓘ http://vimhelp.appspot.com/pattern.txt.html#perl-patterns.

Вместо метасимволов `\zs` и `\ze` в регулярном выражении `/\v"\zs[^"]+\ze"<CR>` из рецепта 78 можно было бы использовать позитивные позиционные проверки Vim:

```
⇒ /\v"@<=[^"]+"@=
```

Вы можете не согласиться со мной, но мне кажется, что версия с метасимволами `\zs` и `\ze` проще воспринимается на глаз. Негативные позиционные проверки широко применяются в некоторых определениях, используемых механизмом подсветки синтаксиса в Vim, но мне в моей практике редко приходилось их использовать. Однако я нашел множество применений позитивным позиционным проверкам, поэтому кажется вполне разумным, что для них были определены собственные сокращения, а именно `\zs` и `\ze`.

Экранируйте символы «/», выполняя поиск вперед

Возьмем для демонстрации следующий фрагмент документа (имеющий URL vimdoc.sourceforge.net[1], но сокращенный здесь, чтобы уместить по ширине книжной страницы):

patterns/search-url.markdown
http://media.pragprog.com/titles/dnvim/code/patterns/search-url.markdown

```
Search items: [http://vimdoc.net/search?q=/\\][s]
...
[s]: http://vimdoc.net/search?q=/\\
```

Допустим, что нам требуется найти все вхождения URL `http://vimdoc.net/search?q=/\\`. Вместо того чтобы вводить эту строку URL полностью, мы просто скопируем ее в регистр, чтобы затем просто вставлять ее в команду поиска. Так как нам нужно найти точное соответствие, будем использовать ключ `\V`.

Если предположить, что курсор находится где-то внутри квадратных скобок, мы можем скопировать строку URL в регистр `u` командой `"uyi[` (чтобы было проще запомнить: *u* означает *URL*). Далее, чтобы вставить содержимое регистра `u` в команду поиска, можно

[1] http://vimdoc.sourceforge.net/search.php?search=/\\&docs=help

ввести /\V<C-r>u<CR>. Получившаяся команда поиска должна выглядеть так:

```
⇛ /\Vhttp://vimdoc.net/search?q=/\\
```

Запустив ее, мы получим следующий результат:

```
Search items: [http://vimdoc.net/search?q=/\\][s]
...
[s]: http://vimdoc.net/search?q=/\\
```

Что случилось, почему мы не получили ожидаемого результата? Когда мы вставляли полную строку URL в команду поиска, Vim интерпретировал первый символ / как конец поля искомого шаблона (см. врезку «Символы завершения шаблона поиска» ниже). Все, что оказалось за первым символом слэша, было просто проигнорировано, поэтому фактически мы выполнили поиск строки http:.

При выполнении поиска вперед необходимо экранировать символы слэша. Это требуется в любом режиме поиска – и в самом волшебном (с ключом \v), и в самом простом (с ключом \V). Давайте исправим ошибку в предыдущей попытке поиска, вставив обратный слэш перед каждым символом /:

```
⇛ /\Vhttp:\/\/vimdoc.net\/search?q=\/\\
```

На этот раз результат получился ближе к ожидаемому:

```
Search items: [http://vimdoc.net/search?q=/\\][s]
...
[s]: http://vimdoc.net/search?q=/\\
```

Но он все еще неверный. Последний обратный слэш не попал в совпадение. Вскоре мы узнаем причину такого непонятного поведения, но сначала познакомимся с правилами поиска назад.

Экранируйте символы «?», выполняя поиск назад

При поиске назад символ ? играет роль границы поля шаблона в команде поиска. То есть при поиске назад не требуется экранировать символы /, зато требуется экранировать символы ?. Взгляните, что получится, если выполнить поиск назад строки URL из регистра u:

```
⇛ ?http://vimdoc.net/search?q=/\\
```

Без экранирования каких-либо символов Vim будет искать совпадения со строкой «http://vimdoc.net/search»:

```
Search items: [http://vimdoc.net/search?q=/\\][s]
...
[s]: http://vimdoc.net/search?q=/\\
```

Этот результат выглядит лучше, чем результат попытки поиска вперед без экранирования символов, но мы все еще не получили совпадения с полной строкой URL. Результат можно улучшить, если вставить обратный слэш перед символом ?:

```
⇛ ?http://vimdoc.net/search\?q=/\\
```

Эта попытка найдет следующие совпадения:

```
Search items: [http://vimdoc.net/search?q=/\\][s]
...
[s]: http://vimdoc.net/search?q=/\\
```

Всегда экранируйте символы «\»

Существует еще один символ, который нужно экранировать в командах поиска: обратный слэш. Обычно символ \ указывает, что следующий за ним символ имеет некоторое специальное значение. Если подряд следуют два символа обратного слэша (\\), первый из них отменяет специальную интерпретацию второго. В итоге такая последовательность требует от Vim найти единственный символ обратного слэша.

В данном примере искомая строка URL включает два символа обратного слэша, следующих подряд. Нам нужно добавить в команду поиска еще по одному символу обратного слэша перед каждым из них. То есть команда поиска вперед должна выглядеть так:

```
⇛ /\Vhttp:\/\/vimdoc.net\/search?q=\/\\\\
```

Наконец-то! Мы получили совпадения с полной строкой URL:

```
Search items: [http://vimdoc.net/search?q=/\\][s]
...
[s]: http://vimdoc.net/search?q=/\\
```

Символы обратного слэша должны экранироваться всегда, независимо от направления поиска, вперед или назад.

Экранируйте символы программно

Экранирование символов вручную – довольно утомительная процедура, при выполнении которой легко можно ошибиться. К счастью, существует библиотека функций на языке сценариев Vim, которые могут взять всю тяжелую работу на себя: `escape({string}, {chars})` (см. `:h escape()` http://vimdoc.sourceforge.net/htmldoc/eval.html#escape()).

Аргумент `{chars}` определяет экранируемый символ. Выполняя поиск вперед, можно произвести вызов `escape(@u, '/\')`, который добавит обратный слэш перед каждым символом `/` и `\`. Выполняя поиск назад, можно произвести вызов `escape(@u, '?\')`.

Прежде всего убедимся, что искомая строка URL все еще хранится в регистре `u`. Затем перейдем в строку ввода команды поиска, нажав `/` или `?`; для демонстрации подойдет любая из них. Введите ключ `\V` и затем нажмите `<C-r>=`. Последняя команда переключит редактор из режима ввода команды поиска в режим ввода выражения. Теперь введите следующую строку:

```
⇒ =escape(@u, getcmdtype().'\')
```

После нажатия клавиши `<CR>` редактор выполнит функцию `escape()` и вставит результат в команду поиска. Функция `getcmdtype()` просто возвращает символ `/`, если выполняется поиск вперед, и символ `?`, если выполняется поиск назад (см. `:h getcmdtype()` http://vimdoc.sourceforge.net/htmldoc/eval.html#getcmdtype()). В языке сценариев Vim оператор `.` выполняет конкатенацию строк, то есть выражение `getcmdtype().'\'` вернет строку «/\», если выполняется поиск вперед, и строку «?\», если выполняется поиск назад. В результате, независимо от направления поиска, это выражение экранирует содержимое регистра `u` так, что его можно использовать в команде поиска.

Переключение в режим ввода выражения и вызов функции `escape()` вручную требуют большого количества нажатий на клавиши. Однако решение этой задачи можно автоматизировать, написав небольшой сценарий Vim. Пример такого сценария вы найдете в рецепте 87 в главе 13.

Символы завершения шаблона поиска

Кого-то может волновать вопрос: «Зачем в команде поиска вообще нужен символ завершения шаблона? Почему бы просто не взять всю строку до конца и не использовать ее?». Дело в том, что команда поиска в Vim поддерживает возможность настройки ее поведения путем добавления некоторых флагов в конец после символа завершения шаблона. Например, если выполнить команду `/vim/e<CR>`, Vim будет устанавливать курсор в конец совпадения, а не в начало. В рецепте 83 (глава 13) будет показано, как можно использовать эту особенность.

Существует лишь одна возможность ввода шаблона, чтобы не беспокоиться о символах завершения шаблона, но она поддерживается лишь в GVim, – команда `:promptfind` (см. `:h :promptfind` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#:promptfind). Эта команда выводит диалоговое окно с полем **Find** (**Что**), куда можно вводить символы `/` и `?`, не экранируя их. Однако символы `\` и перевода строки все еще будут вызывать проблемы.

Глава 13. Поиск

После знакомства с поддержкой регулярных выражений в предыдущей главе можно переходить к их применению в командах поиска. Для начала познакомимся с основами: как инициировать поиск, как обеспечить подсветку совпадений и как выполнять переход между ними. Затем я познакомлю вас с возможностью инкрементального поиска, которая не только дает мгновенную обратную связь, но и позволяет уменьшить объем ввода за счет автодополнения на основе результатов предварительного поиска. В этой главе вы также узнаете, как подсчитать количество совпадений, найденных в документе.

Поддержка смещений в совпадениях позволяет позиционировать курсор относительно границ совпадений. Мы рассмотрим одну из ситуаций, когда эта поддержка позволяет спрямить путь к желаемой цели. Затем мы увидим, как можно использовать поддержку смещений для выполнения операций с полными совпадениями.

Конструирование регулярных выражений – и их понимание – часто требует нескольких попыток, поэтому очень важно иметь возможность итеративной разработки шаблонов. В этой главе мы познакомимся с двумя способами выполнения повторных попыток: обращением к истории поиска и использованием окна с командной строкой.

Хотели бы вы иметь простую возможность искать текст, уже присутствующий в документе? Эту главу мы закончим определением простой настройки команды * для поиска текущего визуального выделения.

Рецепт 80. Встречайте: команда поиска

В этом рецепте мы познакомимся с основами использования команды поиска, включая определение направления поиска, повторение последней операции поиска и работу с историей поиска.

Выполнение команды поиска

Команда / в командном режиме вызывает строку ввода шаблона поиска. В этой строке можно ввести шаблон или буквальный текст, который требуется найти. Поиск не начнется, пока не будет нажата клавиша <CR>. Если нажать клавишу <Esc>, строка ввода шаблона закроется и редактор вернется в командный режим.

В ходе выполнения операции поиска Vim сканирует содержимое документа вперед, начиная от текущей позиции курсора, и останавливается на первом найденном совпадении. Если совпадений не будет найдено до конца документа, Vim проинформирует нас сообщением «search hit BOTTOM, continuing at TOP» (достигнут КОНЕЦ документа, поиск продолжен СНАЧАЛА). Это значит, что поиск вперед может привести нас назад. Такое поведение не должно сбивать вас с толку. Просто помните, что, достигнув конца документа, команда поиска переходит в начало и продолжает поиск.

Если когда-нибудь вам потребуется *ограничить* область поиска текущей позицией курсора и концом документа, выключите параметр настройки `wrapscan` (см. `:h 'wrapscan'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'wrapscan').

Определение направления поиска

Когда поиск инициируется нажатием на клавишу /, Vim будет выполнять сканирование документа вперед. Если строка ввода шаблона была вызвана нажатием на клавишу ?, сканирование будет выполняться в обратном направлении. Строка ввода шаблона всегда начинается с символа / или ?, указывающего направление поиска.

Повторение последней команды поиска

Команда n выполняет переход к следующему совпадению, а команда N – к предыдущему. С помощью команд n и N легко можно перемещаться между совпадениями в текущем документе. Но определение понятия «следующее совпадение» зависит от контекста.

Команда n выполняет перемещение в направлении поиска и соблюдает все применявшиеся настройки смещений (со смещениями мы познакомимся в рецепте 83 ниже). То есть если поиск был запущен в прямом направлении командой /, тогда команда n будет продолжать поиск в прямом направлении. Напротив, если исполь-

зовалась команда ?, тогда команда n будет перемещаться между совпадениями в обратном направлении. Команда N всегда вызывает перемещение в направлении, обратном первоначальному направлению поиска.

Иногда может понадобиться повторить поиск с использованием того же шаблона, но в другом направлении или с другим смещением. В таких случаях полезно помнить, что при выполнении команды поиска без шаблона Vim будет использовать шаблон из предыдущей попытки поиска. В табл. 13.1 перечисляются возможные способы выполнения повторного поиска.

Таблица 13.1. Способы повторного поиска

Команда	Действие
n	Переход к следующему совпадению с сохранением направления поиска и смещения
N	Переход к предыдущему совпадению с сохранением направления поиска и смещения
/<CR>	Переход вперед, к следующему совпадению с тем же шаблоном
?<CR>	Переход назад, к предыдущему совпадению с тем же шаблоном

Допустим, что поиск был инициирован командой ?. После перехода назад, к предыдущему совпадению, мы решили двинуться вперед и просмотреть все имеющиеся совпадения. Мы могли бы продолжить с помощью клавиши N, но это вызывает ощущение выполнения поиска шиворот-навыворот. Вместо этого мы можем выполнить команду /<CR>, инициирующую поиск в прямом направлении с тем же шаблоном, а затем использовать клавишу n для обхода оставшихся совпадений.

Команды n и N переносят курсор, помещая его в начало совпадения с текущим шаблоном. Но как быть, если потребуется выделить совпавший текст в визуальном режиме, чтобы в нем можно было выполнить некоторые правки? В этом случае вам пригодится команда gn. При использовании в командном режиме gn переносит курсор к следующему совпадению, затем включает визуальный режим и выделяет совпавший текст. Если курсор уже находится внутри совпадения, команда gn просто выделит совпадение, не перемещая курсора. Более детально эта команда будет рассматриваться в рецепте 84 в главе 13.

История поиска

Редактор Vim сохраняет шаблоны поиска, чтобы их легко можно было использовать повторно. Когда видна строка ввода шаблона, мы можем прокручивать прежде использовавшиеся шаблоны, нажимая клавишу `<Up>` (со стрелкой вверх). В действительности интерфейс просмотра истории поиска в точности повторяет интерфейс просмотра истории командной строки, который подробно рассматривался в рецепте 34 главы 5. Мы воспользуемся им в рецепте 85.

Рецепт 81. Подсветка совпадений

Редактор Vim может подсвечивать совпадения, но эта особенность отключена по умолчанию. Узнайте, как включать ее и (что не менее важно) как выключить, когда подсветка начинает отвлекать внимание.

Команды поиска дают возможность быстро перемещаться между совпадениями, но по умолчанию Vim никак не выделяет их визуально. Исправить этот недостаток можно, включив параметр `hlsearch` (см. `:h 'hlsearch'` http://vimdoc.sourceforge.net/htmldoc/options.html#'hlsearch'), который отвечает за подсветку совпадений в активном документе и в любых других окнах, открытых в текущей рабочей области.

Отключение подсветки совпадений

Подсветка совпадений – полезная особенность, но иногда она может оказаться нежелательной. Например, когда в результате поиска некоторой типичной строки или шаблона обнаруживаются сотни совпадений, от обилия пятен желтого цвета (или любого другого, в зависимости от настройки цветовой схемы) в рабочей области может рябить в глазах.

В подобных ситуациях можно воспользоваться командой `:set nohlsearch`, чтобы выключить подсветку (также можно использовать команды `:se nohls` и `:se hls!`). Но когда приходит время выполнить следующий поиск, у нас может появиться желание вновь включить подсветку.

В Vim существует более элегантное решение. Временно выключить подсветку можно командой `:nohlsearch` (см. `:h :noh` http://vimdoc.sourceforge.net/htmldoc/pattern.html#:noh). Она будет

оставаться отключенной до следующей команды поиска. Во врезке «Определение комбинации клавиш для отключения подсветки» описывается решение, основанное на определении собственной комбинации клавиш.

Определение комбинации клавиш для отключения подсветки

Ввод команды :noh<CR> для выключения подсветки выглядит довольно трудоемким. Но мы можем упростить доступ к этой функции, определив свою комбинацию клавиш, как показано ниже:

```
nnoremap <silent> <C-l> :<C-u>nohlsearch<CR><C-l>
```

Обычно комбинация <C-l> очищает и перерисовывает экран (см. :h CTRL-L ⓘ http://vimdoc.sourceforge.net/htmldoc/various.html#CTRL-L). Эта комбинация определена на основе команды отключения подсветки совпадений.

Рецепт 82. Предварительный просмотр первого совпадения

Команда поиска в редакторе Vim особенно удобна, когда включена поддержка инкрементального поиска. В этом рецепте описывается пара примеров, когда она может положительно сказываться на вашей работе.

По умолчанию Vim «сидит тихо», пока мы вводим шаблон поиска, и вступает в дело только после нажатия на клавишу <CR>. Мое любимое расширение включается с помощью параметра incsearch (см. :h 'incsearch' ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'incsearch'). Когда этот параметр включен, Vim показывает первое совпадение, опираясь на введенную часть шаблона поиска. Каждый раз, когда вводится новый символ, Vim обновляет результат предварительного поиска. Следующая таблица демонстрирует работу этой особенности:

Нажатия клавиш	Содержимое буфера
{start}	The car was the color of a carrot.
/car	The car was the color of a carrot.
/carr	The car was the color of a carrot.
/carr<CR>	The car was the color of a carrot.

После ввода символов «car» Vim подсветил первое совпадение – в данном случае слово «car». Сразу после ввода символа «r» Vim перемещается вперед, к следующему совпадению. На этот раз совпадение было найдено в слове «carrot». Если в этот момент нажать клавишу <Esc>, строка ввода шаблона будет скрыта, и курсор вернется в первоначальное положение. Но мы нажали клавишу <CR>, чтобы выполнить команду, и наш курсор перепрыгнул в начало слова «carrot».

Немедленная обратная связь дает возможность увидеть, где находится цель поиска. Если бы нашей целью было переместить курсор в начало слова «carrot», нам не пришлось бы вводить искомое слово целиком. В данном примере достаточно было бы ввести /carr<CR>. При выключенном параметре `incsearch` неизвестно, позволит ли введенный шаблон достигнуть цели, пока команда поиска не будет выполнена нажатием <CR>.

Проверка существования совпадения

В нашем примере имеются два совпадения со словом «car» в одной и той же строке. Но представьте, что слова «car» и «carrot» разделены несколькими сотнями других слов. В момент ввода еще одного символа «r» в строку ввода шаблона редактору придется прокрутить содержимое документа, чтобы поле «carrot» оказалось в видимой области, что, собственно, и происходит на практике.

Допустим, что нам нужно всего лишь узнать, присутствует ли слово «carrot» в текущем документе, не перемещая курсора. Если включен параметр `incsearch`, мы могли бы просто набрать в поле ввода шаблона столько символов из слова «carrot», сколько необходимо, чтобы убедиться в его присутствии, а затем просто нажать клавишу <Esc>, чтобы вернуться в исходную позицию. При таком подходе не нужно прерывать ход своих мыслей.

Автодополнение поля ввода шаблона с опорой на предварительные результаты поиска

В последнем примере выше мы выполнили команду поиска до того, как ввели полное слово «carrot». Такой прием достаточно хорош, если целью поиска является всего лишь перемещение курсора к первому совпадению. Но представьте, что необходимо найти со-

впадение с полным словом «carrot»: например, если вслед за командой поиска предполагается выполнить команду подстановки.

Конечно, мы могли бы просто ввести слово «carrot» целиком. Но существует более удобный способ – комбинация `<C-r><C-w>`. Она автоматически дополнит содержимое строки ввода шаблона текущим совпадением. Если вызвать эту команду после ввода «carr», в строку ввода будут добавлены символы «ot», обеспечив совпадение с полным словом «carrot».

Обратите внимание, что команда автодополнения `<C-r><C-w>` страдает некоторой ненадежностью при подобном применении. Если вставить в начало шаблона ключ `\v`, команда `<C-r><C-w>` вставит в строку ввода все слово под курсором (в результате, например, получится шаблон `/\vcarrcarrot<CR>`), а не только недостающий остаток. Когда выполняется поиск точного совпадения, а не по шаблону, функция автодополнения инкрементального режима поиска может сэкономить немало времени.

Рецепт 83. Смещение курсора относительно конца совпадения

Смещение относительно концов совпадения можно использовать для позиционирования курсора относительно начала или конца совпадения на фиксированное число символов. В этом рецепте мы изучим пример, где благодаря возможности позиционирования курсора относительно конца совпадения мы сможем выполнить последовательность изменений, используя «формулу точки».

Каждый раз, выполняя команду поиска, курсор устанавливается на первый символ в совпадении. Такое поведение по умолчанию выглядит вполне разумным, но иногда нам может потребоваться, чтобы курсор устанавливался на последний символ в совпадении. В редакторе Vim подобная возможность поддерживается с помощью смещений в совпадениях (см. `:h search-offset` ⓘ http://vimdoc.sourceforge.net/htmldoc/pattern.html#search-offset).

Рассмотрим следующий пример. В этом фрагменте автор постоянно сокращает слово «language» до «lang»:

search/langs.txt
http://media.pragprog.com/titles/dnvim/code/search/langs.txt

```
Aim to learn a new programming lang each year.
Which lang did you pick up last year?
Which langs would you like to learn?
```

Как бы вы решили задачу дополнения всех трех вхождений слова «lang» до «language»? Одно из решений – воспользоваться командой подстановки: :%s/lang/language/g. Но давайте поищем другой способ, к которому можно было бы применить «формулу точки». Попутно мы узнаем немало интересного.

Для начала попробуем решить поставленную задачу без использования смещений в совпадениях. Итак, прежде всего найдем строку, которую требуется изменить: /lang<CR>. Эта команда поместит курсор в начало первого совпадения. Из этой позиции можно выполнить вставку в конец слова командой eauage<Esc>. Добавление в конец слова – настолько часто используемая операция, что последовательность ea должна нажиматься на инстинктивном уровне, как единая команда.

Теперь нужно переместить курсор в правильную позицию, а все остальное выполнит команда «точка». Чтобы исправить следующее вхождение слова «lang», можно выполнить команду ne. – n выполнит переход к следующему совпадению; e переместит курсор в конец слова, а «точка» (.) добавит необходимые символы. Итого три нажатия на клавиши. Мы не достигли идеальной «формулы точки», но решили поставленную задачу, так или иначе.

Или не решили? Если выполнить ту же последовательность команд, ne., второй раз, она исказит последнее слово. Сможете объяснить причину? Последнее вхождение «lang» в действительности является сокращением от слова «languages». Поэтому если слепо повторить нашу неидеальную «формулу точки», она создаст жуткое слово «langsuage». Очевидно, что это один из примеров, когда предпочтительнее было иметь возможность устанавливать курсор в конец совпадения, а не в конец слова.

В табл. 13.2 демонстрируется улучшенное решение.

Здесь выполняется поиск командой /lang/e<CR>, которая помещает курсор в конец совпадения, именно так, как нам нужно. Теперь каждый раз, когда будет выполняться команда n, курсор будет устанавливаться в конец найденного совпадения, подготавливая почву для команды «точка». Итак, использование смещения в совпадении помогло нам получить идеальную «формулу точки».

На практике не всегда очевидно, когда применение смещений в совпадениях может принести выгоду. Представьте, что мы начали выполнять команду поиска без смещения, но, выполнив команду n пару раз, поняли, что было бы удобно, если бы курсор устанавливался в конец совпадения. Эта проблема легко решается, достаточно

выполнить команду `//e<CR>`. Когда поле шаблона остается пустым, как в данном случае, Vim будет повторно использовать предыдущий шаблон. То есть данная команда повторит предыдущую команду поиска, но со смещением.

Таблица 13.2. Улучшенное решение с использованием «формулы точки»

Нажатия клавиш	Содержимое буфера

`{start}`	`Aim to learn a new programming lang each year.`
	`Which lang did you pick up last year?`
	`Which langs would you like to learn?`

`/lang/e<CR>`	`Aim to learn a new programming lang each year.`
	`Which lang did you pick up last year?`
	`Which langs would you like to learn?`

`auage<Esc>`	`Aim to learn a new programming language each year.`
	`Which lang did you pick up last year?`
	`Which langs would you like to learn?`

`n`	`Aim to learn a new programming language each year.`
	`Which lang did you pick up last year?`
	`Which langs would you like to learn?`

`.`	`Aim to learn a new programming language each year.`
	`Which language did you pick up last year?`
	`Which langs would you like to learn?`

`n.`	`Aim to learn a new programming language each year.`
	`Which language did you pick up last year?`
	`Which languages would you like to learn?`

Рецепт 84. Выполнение операций над полным совпадением

Команда поиска в редакторе Vim позволяет подсвечивать совпадения и быстро перемещаться между ними. Кроме того, имеется возможность оперировать фрагментами теста, совпавшими с текущим шаблоном, с помощью команды `gn`.

Команду поиска в редакторе Vim удобно использовать для перемещения между совпадениями с шаблоном, но что, если нам потребуется изменить каждое из них? Прежде это была очень непростая задача, но с появлением команды `gn` (в версии 7.4.110) открылась возможность достаточно эффективно манипулировать результатами поиска.

Рассмотрим эту ситуацию на примере. В следующем фрагменте мы будем изменять имена классов `XmlDocument`, `XhtmlDocument`, `XmlTag` и `XhtmlTag`.

search/tag-heirarchy.rb
http://media.pragprog.com/titles/dnvim/code/search/tag-heirarchy.rb

```
class XhtmlDocument < XmlDocument; end
class XhtmlTag < XmlTag; end
```

Допустим, что нам требуется переименовать все классы, как показано ниже:

```
class XHTMLDocument < XMLDocument; end
class XHTMLTag < XMLTag; end
```

Мы могли бы использовать команду `gU{motion}` для преобразования фрагмента текста в верхний регистр (см. `:h gU` ⓘ http://vimhelp.appspot.com/change.txt.html#gU), а для перемещения – команду `gn`, которая воздействует на следующее совпадение (см. `:h gn` ⓘ http://vimhelp.appspot.com/visual.txt.html#gn). Если курсор находится в пределах совпадения, тогда команда `gn` будет воздействовать на текущее совпадение. Но если курсор находится за границами совпадения, тогда `gn` перепрыгнет вперед к следующему совпадению и применит указанную операцию. В табл. 13.3 показано, как пользоваться этой командой.

Таблица 13.3. Выполнение операций над полным совпадением

Нажатия клавиш	Содержимое буфера
`{start}`	`class XhtmlDocument < XmlDocument; end` `class XhtmlTag < XmlTag; end`
`/\vX(ht)?ml\C<CR>`	`class XhtmlDocument < XmlDocument; end` `class XhtmlTag < XmlTag; end`
`gUgn`	`class XHTMLDocument < XmlDocument; end` `class XhtmlTag < XmlTag; end`
`n`	`class XHTMLDocument < XmlDocument; end` `class XhtmlTag < XmlTag; end`
`.`	`class XHTMLDocument < XMLDocument; end` `class XhtmlTag < XmlTag; end`
`n.`	`class XHTMLDocument < XMLDocument; end` `class XHTMLTag < XmlTag; end`
`.`	`class XHTMLDocument < XMLDocument; end` `class XHTMLTag < XmlTag; end`

Прежде всего напишем регулярное выражение, совпадающее с «Xml» или «Xhtml». Это достаточно просто: команда поиска `/\vX(ht)?ml\C<CR>` обеспечит достижение желаемого результата.

Специальный символ `\C` включает чувствительность к регистру, поэтому шаблон не будет совпадать с `XML` или `XHTML`. После выполнения поиска с применением этого шаблона подсвеченными оказываются четыре фрагмента, которые требуется изменить, и курсор устанавливается в начало первого совпадения.

Команда `gUgn` преобразует совпавший текст в верхний регистр. Но главное ее достоинство в том, что она легко повторяется. Мы можем перейти к следующему совпадению, просто нажав `n` и повторив изменение, нажав `.`. Всего два нажатия на клавиши на одно изменение – наша классическая «формула точки». Но это редчайший случай, когда есть возможность еще больше сэкономить на нажатиях клавиш.

Усовершенствованная формула точки

Операцию `gUgn` можно было бы описать как: преобразовать *следующее совпадение* в верхний регистр. Если курсор уже находится в пределах совпадения, тогда нажатие `.` будет воздействовать на совпадение под курсором. Но если курсор находится вне совпадения, тогда `.` выполнит переход к следующему совпадению и применит указанную операцию. Нам не нужно даже нажимать клавишу `n`. Достаточно просто нажимать `.`, а это означает, что для выполнения каждого изменения достаточно нажатия одной клавиши.

Классическая формула точки делится на две части: одно нажатие для перемещения, другое – для выполнения изменений. Команда `gn` позволяет уместить эти два шага в один, потому что `gn` воздействует на *следующее совпадение*, а не на текст в позиции курсора. Если есть возможность организовать работу так, что команда `.` будет выполнять переход к следующему совпадению *и* применять к нему последнюю операцию, тогда мы сможем повторить изменение нажатием единственной клавиши. Я называю это «усовершенствованной формулой точки».

Попробуйте повторить этот пример с шаблоном, не учитывающим регистра символов, заменив `\C` на `\c`. Вы увидите, что возможность повторять каждое изменение нажатием `n.` сохранилась (классическая формула точки), но нажатие одной только клавиши `.` не выполняет перехода к следующему совпадению в документе. Это объясняется тем, что шаблон, нечувствительный к регистру, совпадает с текстом до и после выполнения команды `gUgn`: например, он совпадает с `Xml` и `XML`. В этом случае команда `.` будет применять

изменения к совпадению под курсором, вместо того чтобы выполнить переход к следующему совпадению. Вы не увидите никаких изменений, потому что преобразование `XML` в верхний регистр визуально ничего не изменяет.

Чтобы усовершенствованная формула точки могла действовать, шаблон должен совпадать с искомым текстом до его изменения, но не совпадать после изменения. В данном конкретном примере операция `gU` изменяет регистр символов целевого текста, поэтому так важно было сделать шаблон чувствительным к регистру, но для успешного применения усовершенствованной формулы точки это требуется не всегда.

Попробуйте повторить пример, но на этот раз используйте команду `dgn`, чтобы удалить совпавший текст. Или используйте `cgnJson<Esc>`, чтобы заменить каждое совпадение текстом `Json`. В обоих случаях должна появиться возможность повторять изменения совпадений нажатием одной клавиши `.`. Пока целевой текст изменяется так, что перестает совпадать с шаблоном поиска, мы можем использовать усовершенствованную форму точки.

Будьте осторожны, применяя усовершенствованную форму точки к большим файлам, где совпадения могут далеко отстоять друг от друга. При использовании `n.` есть возможность приостановиться между двумя нажатиями и решить, требуется ли применить изменение нажатием `.`. Тогда как при использовании `.` без предварительного перехода к следующему совпадению вы можете не видеть всех совпадений перед их изменением.

После выхода версии Vim 7.4.110 команда `gn` стала основным моим инструментом. Если вы пользуетесь более старой версией Vim, эта команда является веской причиной, чтобы обновиться!

Рецепт 85. Создание сложных шаблонов с использованием истории поиска

Разработка регулярных выражений – достаточно сложное дело. Редко когда получается написать желаемое регулярное выражение с первого раза, поэтому всегда полезно иметь возможность вести разработку поэтапно. И помочь нам в этом сможет история поиска.

В центре нашего внимания, в следующем примере, будет находиться символ кавычки:

search/quoted-strings.txt
http://media.pragprog.com/titles/dnvim/code/search/quoted-strings.txt

```
This string contains a 'quoted' word.
This string contains 'two' quoted 'words.'
This 'string doesn't make things easy.'
```

Нам требуется составить регулярное выражение, соответствующее каждой строке в кавычках. Для этого понадобится выполнить несколько попыток, зато потом, когда у нас будет работоспособный шаблон, мы сможем выполнить команду подстановки, замещающую одиночные кавычки двойными, как в следующем фрагменте:

```
This string contains a "quoted" word.
This string contains "two" quoted "words."
This "string doesn't make things easy."
```

1: Максимальное совпадение

Начнем с самого простого шаблона:

```
⇒ /\v'.+'
```

Ему соответствует единственный символ `'`, за которым следуют любой символ один или более раз и заключительный символ `'`. После выполнения этой команды поиска документ приобретает вид, как показано ниже:

```
This string contains a 'quoted' word.
This string contains 'two' quoted 'words.'
This 'string doesn't make things easy.'
```

Первое совпадение соответствует желаемому, но второе – нет. Элемент `.+` в шаблоне старается найти *максимальное* совпадение, в том смысле что старается охватить как можно больше символов. Но нам нужно, чтобы в этой строке было найдено два совпадения: по одному для каждого слова в кавычках. Вернемся назад к чертежной доске.

2: Доработка

Попробуем вместо символа `.`, соответствующего любому символу, использовать что-нибудь более специализированное. В действительности нам требуется совпадение с любым символом, кроме `'`, для

чего можно использовать `[^']+`. Попробуем доработать шаблон, чтобы он выглядел так:

```
⇛ /\v'[^']+'
```

Необязательно вводить это выражение с самого начала. Достаточно просто ввести `/<Up>`, в результате чего поле шаблона будет заполнено предыдущим регулярным выражением. После этого остается только внести небольшое изменение: нажать клавишу `<Left>` и клавишу забоя (backspace), чтобы удалить символ `.` из шаблона, и затем ввести его замену. Попытка поиска с новым шаблоном дает следующий результат:

```
This string contains a 'quoted' word.
This string contains 'two' quoted 'words.'
This 'string doesn't make things easy.'
```

Неплохо! Первые две строки соответствуют нашим желаниям, но у нас появилась новая проблема в третьей строке. Здесь присутствует символ `'`, используемый в качестве апострофа, который не должен завершать совпадение. Требуется дополнительная доработка шаблона.

3: Еще один цикл

Теперь нам нужно понять, что отличает апостроф от закрывающей одиночной кавычки. Взгляните на следующие примеры: «won't», «don't» и «we're». В каждом случае сразу за апострофом следует буква – не пробел и не знак препинания. Изменим шаблон так, чтобы он продолжал искать соответствие, пока за символом `'` следует символ слова. Ниже приводится следующая версия:

```
⇛ /\v'([^']|'\w)+'
```

На этот раз потребовалось внести более существенные изменения: не только добавить дополнительный элемент `'\w`, но также заключить две альтернативы в круглые скобки и разделить их вертикальной чертой. Самое время достать из арсенала нашу большую пушку.

Вместо нажатия на клавиши `/<Up>`, чтобы вставить в команду поиска предыдущий шаблон, нажмите `q/`, чтобы вывести окно с командной строкой. Оно действует подобно обычному буферу, но содержит всю историю команд поиска, по одной в каждой строке

(см. раздел «Встречайте: окно режима командной строки» в главе 5). Здесь можно использовать все возможности модального редактирования, чтобы исправить последний шаблон.

Ниже приводится последовательность команд редактирования, демонстрирующая, как внести данное конкретное изменение. Если вы забыли, что делает команда `c%(<C-r>")<Esc>`, обращайтесь к рецептам 55 (глава 8) и 15 (глава 2).

Нажатия клавиш	Содержимое буфера
`{start}`	`\v'[^']+'`
`f[`	`\v'[^']+'`
`c%(<C-r>")<Esc>`	`\v'([^'])+'`
`i|'\w<Esc>`	`\v'([^']|'\w)+'`

Отредактировав шаблон, достаточно просто нажать клавишу `<CR>`, чтобы выполнить поиск. Теперь документ должен выглядеть так:

```
This string contains a 'quoted' word.
This string contains 'two' quoted 'words.'
This 'string doesn't make things easy.'
```

Отлично!

4: Последний штрих

Наш шаблон обнаруживает все желаемые фрагменты, но, прежде чем выполнить команду подстановки, необходимо добавить последний штрих. Нам нужно сохранить все, что заключено в одиночные кавычки, заключив подвыражение в круглые скобки. В окончательном виде наш шаблон выглядит, как показано ниже:

```
⇒ /\v'(([^']|'\w)+)'
```

Здесь можно было бы ввести команду `/<Up>` и отредактировать шаблон в команде поиска, или воспользоваться командой `q/` и внести изменения в окне командной строки. Используйте тот способ, который вам больше по душе. Результаты поиска с применением последней версии шаблона не отличаются от предыдущих, но теперь текст внутри одиночных кавычек сохраняется в регистре захвата `\1`. Благодаря этому можно выполнить следующую команду подстановки:

```
⇒ :%s//"\1"/g
```

Мы можем оставить поле шаблона пустым, и Vim будет повторно использовать последнюю команду поиска (за дополнительными подробностями обращайтесь к рецепту 91 в главе 14). Ниже приводится результат выполнения команды подстановки:

```
This string contains a "quoted" word.
This string contains "two" quoted "words."
This "string doesn't make things easy."
```

Обсуждение

Все, что было сделано выше, фактически идентично команде:

```
⇒ :%s/\v'(([^']|'\w)+)'/"\1"/g
```

Смогли бы вы ввести такую команду с первой попытки и без ошибок?

Не волнуйтесь, если вам не удастся создать нужное регулярное выражение с первой попытки. Редактор Vim сохранит шаблон на расстоянии двух нажатий на клавиши, и вы легко сможете восстановить его. Начинайте с поиска максимального совпадения, затем продолжайте доработку в несколько этапов, пока не достигнете желаемого результата.

Возможность редактирования содержимого командной строки здесь как нельзя кстати. Если включить параметр `incsearch`, появится дополнительное преимущество в виде немедленной обратной связи по мере ввода команды. Конечно, это преимущество теряется с переходом в окно командной строки. Но это – небольшая цена за дополнительные возможности модального редактирования.

Рецепт 86. Подсчет совпадений с текущим шаблоном

В этом рецепте демонстрируется пара способов подсчета числа совпадений с шаблоном.

Представьте, что нам нужно найти число вхождений слова «buttons» в следующем фрагменте:

search/buttons.js
http://media.pragprog.com/titles/dnvim2/code/search/buttons.js

```
var buttons = viewport.buttons;
viewport.buttons.previous.show();
```

```
viewport.buttons.next.show();
viewport.buttons.index.hide();
```

Начнем с поиска этого слова:

```
⇒ /\<buttons\>
```

Теперь мы можем переходить от одного совпадения к другому, нажимая клавиши `n` и `N`, но команды поиска в Vim не сообщают количества совпадений, найденных в текущем документе. Подсчитать количество совпадений можно командой `:substitute` или `:vimgrep`.

Подсчет количества совпадений командой ':substitute'

Подсчитать количество совпадений можно, выполнив следующую команду:

```
⇒ /\<buttons\>
⇒ :%s///gn
   5 matches on 4 lines
```

Мы вызвали команду `:substitute`, подавив ее обычное поведение с помощью флага `n`. Вместо замены каждого совпадения она просто подсчитала количество совпадений и вывела результат в командной строке. Оставив поле искомого шаблона пустым, мы сообщили Vim, что тот должен использовать текущий шаблон. Поле со строкой замены игнорируется в любом случае (из-за флага `n`), поэтому его тоже можно оставить пустым.

Обратите внимание, что команда содержит последовательность из трех символов слэша `/`. Первый и второй ограничивают поле шаблона, а второй и третий – поле со строкой замены. Будьте внимательны, чтобы не упустить один из символов `/`. Команда `:%s//gn` заменит все совпадения парой символов `gn`!

Подсчет количества совпадений командой ':vimgrep'

Флаг `n` в команде `:substitute` помогает узнать общее количество совпадений с шаблоном. Но иногда бывает полезно знать, что данное конкретное совпадение является, например, третьим из пяти. Эту информацию можно получить с помощью команды `:vimgrep`:

```
⇒ /\<buttons\>
⇒ :vimgrep //g %
  (1 of 5) var buttons = viewport.buttons;
```

Эта команда заполнит список результатов (quickfix list) совпадениями, найденными в текущем буфере. Команда `:vimgrep` способна выполнять поиск во множестве файлов, но здесь нас интересует единственный файл. Символ `%` развертывается в путь к файлу в текущем буфере (см. `:h cmdline-special` ⓘ http://vimhelp.appspot.com/cmdline.txt.html#cmdline-special). Оставив поле шаблона пустым, мы потребовали от команды `:vimgrep` использовать текущий шаблон поиска.

Теперь мы можем перемещаться между совпадениями не только с помощью клавиш `n` и `N`, но и с помощью команд `:cnext` и `:cprev`:

```
⇒ :cnext
  (2 of 5) var buttons = viewport.buttons;
⇒ :cnext
  (3 of 5) viewport.buttons.previous.show();
⇒ :cprev
  (2 of 5) var buttons = viewport.buttons;
```

Этот прием кажется мне более предпочтительным, чем с применением команды `:substitute`, когда требуется просмотреть каждое совпадение и, может быть, внести какие-то изменения. Очень полезно видеть сноску `(1 of 5)`, затем `(2 of 5)` и т. д., которая подсказывает, как много еще осталось сделать.

Поддержка списка результатов – важнейшая из функций в Vim, на которой основано множество приемов работы с редактором. Подробнее об этой функции рассказывается в главе 17 «Компиляция кода и обзор ошибок с помощью Quickfix List».

Рецепт 87. Поиск текущего визуального выделения

В командном режиме команда `*` позволяет выполнять поиск слова под курсором. С помощью небольшого сценария можно переопределить поведение команды `*` так, что в визуальном режиме она будет искать не текущее слово, а текущий выделенный фрагмент.

Поиск текущего слова в визуальном режиме

В визуальном режиме команда `*` выполняет поиск слова под курсором. Например:

Нажатия клавиш	Содержимое буфера
{start}	She sells sea shells by the sea shore.
*	She sells sea shells by the sea shore.

Первоначально редактор находится в визуальном режиме, в документе выделены три первых слова и курсор находится на слове «sea». Если вызвать команду *, она найдет следующее вхождение слова «sea» и расширит выделенную область. Хотя такое поведение команды * вполне согласуется с поведением этой же команды в командном режиме, я редко нахожу его удобным.

До того, как я был пленен редактором Vim, я пользовался другим текстовым редактором, имевшим команду «Искать выделенный текст». У меня была настроена горячая комбинация клавиш для доступа к этой команде, и она использовалась мною настолько часто, что мои пальцы вызывали ее на инстинктивном уровне. Когда я начал осваивать Vim, я был удивлен отсутствием подобной возможности в этом редакторе. Мне всегда казалось, что в визуальном режиме команда * должна искать текущее выделение, а не текущее слово. Однако эту возможность легко добавить в Vim, написав небольшой сценарий.

Поиск текущего выделения (прототип)

Если заглянуть в раздел справки `:h visual-search` ⓘ http://vimdoc.sourceforge.net/htmldoc/visual.html#visual-search, можно увидеть следующее предложение:

> Ниже предлагается вариант реализации поиска выделенного текста и назначения горячей клавиши:
>
> ```
> :vmap X y/<C-R>"<CR>
> ```
>
> Обратите внимание, что присутствие специальных символов в тексте (таких как «.» и «*») будет вызывать проблемы.

Команда `y` захватывает (копирует) текущее выделение, а последовательность `/<C-r>"<CR>` инициирует ввод команды поиска, вставляет содержимое регистра по умолчанию и выполняет поиск. Решение достаточно простое, но, согласно предупреждению в справке, имеет некоторые ограничения.

В рецепте 79 (глава 12) мы узнали, как преодолевать подобные ограничения. Теперь попробуем совместить теорию с практикой и создать привязку к клавише, обеспечивающую возможность поиска

текущего выделения, решающую проблему наличия специальных символов.

Поиск текущего выделения (окончательная версия)

Следующий сценарий Vim решает поставленную задачу:

patterns/visual-star.vim
http://media.pragprog.com/titles/dnvim/code/patterns/visual-star.vim

```
xnoremap * :<C-u>call <SID>VSetSearch()<CR>/<C-R>=@/<CR><CR>
xnoremap # :<C-u>call <SID>VSetSearch()<CR>?<C-R>=@/<CR><CR>

function! s:VSetSearch()
  let temp = @s
  norm! gv"sy
  let @/ = '\V' . substitute(escape(@s, a:cmdtype. '|'), '\n', '\\n', 'g')
  let @s = temp
endfunction
```

Этот фрагмент можно вставить в файл *vimrc* непосредственно или установить расширение, реализующее данную функциональность[1].

Помимо команды `*`, мы также переопределили команду `#`, которая теперь будет выполнять поиск текущего выделения в обратном направлении. Ключевое слово `xnoremap` указывает, что данная привязка должна действовать в визуальном режиме, но не должна действовать в режиме выделения (см. `:h mapmode-x` ⓘ http://vimdoc.sourceforge.net/htmldoc/map.html#mapmode-x).

[1] https://github.com/nelstrom/vim-visual-star-search

Глава 14. Подстановка

Кому-то может показаться, что команда подстановки просто выполняет операции поиска и замены, но это не так. В действительности это одна из самых мощных команд Ex. К концу этой главы вы познакомитесь со всеми обличьями команды подстановки, от самых простых до весьма сложных.

Вашему вниманию будет представлено несколько советов и рекомендаций, которые позволят конструировать команды подстановки быстрее за счет использования последнего шаблона. Вы также познакомитесь с особым случаем, когда Vim позволяет просмотреть каждое совпадение перед подстановкой. Затем вы узнаете, как заполнять поле со строкой замены без ввода символов, и изучите некоторые особенности, которыми обладает поле замены. Кроме того, вы увидите, как повторить последнюю команду подстановки с другим диапазоном, без повторного ввода всей команды.

В поле замены поддерживается возможность выполнения выражений на языке сценариев Vim. Вы познакомитесь с примерами использования этой возможности для выполнения арифметических операций с последовательностями числовых совпадений. Затем вы узнаете, как одной командой подстановки менять местами два или более слов.

Завершится эта глава обзором двух стратегий поиска с заменой в нескольких файлах.

Рецепт 88. Встречайте: команда подстановки

Команда `:substitute` является одной из самых сложных: кроме шаблона поиска и строки замены, в ней можно указать диапазон символов, над которыми следует выполнить операцию. Дополни-

тельно в команде подстановки можно указать флаги для настройки ее поведения.

Команда подстановки позволяет найти и заменить один фрагмент текста другим. Она имеет следующий синтаксис:

```
:[range]s[ubstitute]/{pattern}/{string}/[flags]
```

Команда подстановки состоит из множества частей. При определении элемента `[range]` используются те же правила, что и в других командах Ex, которые подробно рассматривались в рецепте 28 (глава 5). То же относится и к элементу `{pattern}`, о котором рассказывалось в главе 12 «Поиск по шаблону и поиск точного совпадения».

Настройка поведения команды подстановки с помощью флагов

Поведение команды подстановки можно настраивать с помощью флагов. Проще всего разобраться с флагами – посмотреть на них в действии, но прежде чем двинуться дальше, давайте коротко ознакомимся с определениями флагов, которые будут использоваться в рецептах, следующих ниже. (Полное описание можно найти в справке `:h :s_flags` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#:s_flags.)

Флаг `g` включает *глобальный* (globally) режим действия команды подстановки, обеспечивая изменение всех совпадений, а не только первого найденного. Этот флаг встретится нам в рецепте 89.

Флаг `c` дает возможность *подтвердить* или отвергнуть замену найденного совпадения. Этот флаг встретится нам в рецепте 90.

Флаг `n` подавляет типичное поведение команды подстановки, заставляя ее сообщать *количество* найденных совпадений. Пример использования этого флага приводился в рецепте 86 в главе 13.

Если выполнить команду подстановки с шаблоном, не имеющим совпадений в текущем файле, Vim сообщит об ошибке: «E486: Pattern not found» («Шаблон не найден»). Отключить вывод этого сообщения можно с помощью флага `e`.

Флаг `&` просто предписывает редактору Vim повторно использовать тот же набор флагов, что использовался в предыдущей команде подстановки. Где может пригодиться такая возможность, рассказывается в рецепте 93.

Специальные символы в строке замены

В главе 12 «Поиск по шаблону и поиск точного совпадения» мы видели, что некоторые символы имеют специальное значение в шаблонах. Некоторые символы в строках замены также приобретают специальное значение. Некоторые из них перечислены в таблице ниже, а полный их перечень можно найти в разделе справки `:h sub-replace-special` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#sub-replace-special:

Символ	Значение
`\r`	Вставляет возврат каретки
`\t`	Вставляет символ табуляции
`\\`	Вставляет один символ обратного слэша
`\1`	Вставляет совпадение с первым подвыражением в скобках
`\2`	Вставляет совпадение со вторым подвыражением в скобках
`\0`	Вставляет полное совпадение с шаблоном
`&`	Вставляет полное совпадение с шаблоном
`~`	Использует строку замены `{string}` из предыдущей команды подстановки
`\={Vim script}`	Вычисляет выражение `{Vim script}`; использует результат в качестве строки замены `{string}`

Специальные символы `\r`, `\t` и `\\` многим знакомы очень хорошо. С символом ~ мы познакомимся поближе в рецепте 93, где также будут описаны два сокращения, позволяющие повторно вызывать команду подстановки еще быстрее. Примеры использования специальных символов `\1` и `\2` будут продемонстрированы в рецепте 94.

Комбинация `\={Vim script}` обладает широчайшими возможностями. Она позволяет выполнять программный код и использовать результат в качестве строки замены `{string}`. Пару примеров использования этой комбинации мы рассмотрим в рецептах 95 и 96.

Рецепт 89. Поиск и замена всех совпадений в файле

По умолчанию команда подстановки воздействует на текущую строку и изменяет только первое совпадение. Чтобы изменить все совпадения в файле, необходимо определить диапазон `[range]` и указать флаг `g`.

Для демонстрации воспользуемся следующим фрагментом:

substitution/get-rolling.txt
http://media.pragprog.com/titles/dnvim/code/substitution/get-rolling.txt

```
When the going gets tough, the tough get going.
If you are going through hell, keep going.
```

Попробуем заменить все вхождения слова «going» на «rolling». Для начала включим параметр `hlsearch`, чтобы можно было видеть происходящее (см. рецепт 81 в главе 13):

```
➾ :set hlsearch
```

В простейшем виде команда подстановки включает шаблон поиска `{pattern}` и строку замены `{string}`:

Нажатия клавиш	Содержимое буфера
`:s/going/rolling`	`When the rolling gets tough, the tough get going.` `If you are going through hell, keep going.`

Заметили, что произошло? Редактор заменил первое вхождение «going» на «rolling» и оставил все остальные совпадения нетронутыми.

Чтобы понять причину, представьте, что файл представляет собой двумерное поле из символов, в котором ось X направлена слева направо, а ось Y – сверху вниз. По умолчанию команда подстановки воздействует только на первое совпадение в текущей строке. Давайте посмотрим, что необходимо сделать, чтобы полностью охватить обе оси координат, X и Y.

Чтобы команда продолжила движение по оси X, в нее необходимо включить флаг `g`. Название этого флага происходит от английского *global* (глобальный, всеобщий), что, вообще говоря, является не совсем правильным. Всегда найдется кто-то, кто будет ожидать, что данный флаг должен обеспечивать распространение действия команды подстановки на весь файл, но в действительности этот флаг означает «глобальное действие в пределах текущей строки». Такая интерпретация приобретает определенный смысл, если вспомнить, что Vim является прямым потомком редактора `ed`, как описывается во врезке «О происхождении редактора Vim (и его семейства)» в главе 5.

Нажмите клавишу `u`, чтобы отменить последнее изменение, и давайте попробуем выполнить другую разновидность команды подстановки – с флагом `/g` в конце:

Нажатия клавиш	Содержимое буфера
:s/going/rolling/g	When the rolling gets tough, the tough get rolling. If you are going through hell, keep going.

На этот раз изменились все вхождения слова «going» в текущей строке, но в файле осталась еще пара совпадений, которые требуется изменить. Как подсказать команде подстановки, что она должна продолжить обработку файла еще и по оси Y?

Ответ на этот вопрос: указать диапазон. Если перед командой подстановки добавить префикс %, она выполнит обход всех строк в файле:

Нажатия клавиш	Содержимое буфера
:%s/going/rolling/g	When the rolling gets tough, the tough get rolling. If you are rolling through hell, keep rolling.

Команда подстановки – это лишь одна из множества команд Ex, каждая из которых может принимать диапазон строк. Подробнее о диапазонах рассказывается в рецепте 28 в главе 5.

Итак, если необходимо найти и заменить все совпадения с шаблоном в текущем файле, следует явно сообщить команде подстановки, что она должна просматривать содержимое файла по обеим осям, X и Y. Флаг g управляет просмотром по оси X, а адрес % – по оси Y.

Подобные детали легко забываются, поэтому в рецепте 93 ниже будет продемонстрирована пара приемов многократного повторения команды подстановки.

Рецепт 90. Подтверждение каждой подстановки

Слепой поиск и замена всех совпадений с шаблоном не всегда оказываются лучшим решением. Иногда бывает желательно оценить каждое совпадение и подтвердить или отвергнуть подстановку. Эту возможность обеспечивает флаг c.

Помните пример из рецепта 5 в главе 1?

the_vim_way/1_copy_content.txt
http://media.pragprog.com/titles/dnvim/code/the_vim_way/1_copy_content.txt

```
...We're waiting for content before the site can go live...
...If you are content with this, let's go ahead with it...
...We'll launch as soon as we have the content...
```

В этом примере мы не могли использовать поиск с заменой, чтобы заменить слова «content» словами «copy». Тогда мы использовали «формулу точки». Однако эту задачу можно было бы решить с помощью команды подстановки, добавив в нее флаг c:

```
⇒ :%s/content/copy/gc
```

Флаг c вынуждает редактор Vim показать каждое совпадение и спросить: «Replace with copy?» («Заменить на copy?»). В ответ на этот вопрос можно нажать y, чтобы подтвердить замену, или n, чтобы пропустить совпадение. Редактор выполнит наше требование, перейдет к следующему совпадению и снова задаст вопрос.

В данном примере мы могли бы ответить yny, подтвердив подстановку для первого и последнего совпадений и оставив второе совпадение нетронутым.

Мы не ограничены двумя вариантами ответов. В действительности Vim услужливо напоминает в строке запроса все доступные варианты ответов «y/n/a/q/l/^E/^Y», значения которых описываются в таблице ниже:

Вариант ответа	Результат
y	Выполнить подстановку для данного совпадения
n	Пропустить данное совпадение
q	Прервать выполнение команды
l	«last» (последнее) – Выполнить подстановку для данного совпадения и прервать выполнение команды
a	«all» (все) – Выполнить подстановку для всех оставшихся совпадений
<C-e>	Прокрутить экран вверх
<C-y>	Прокрутить экран вниз

Это описание можно также найти в разделе справки :h :s_c ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#:s_c.

Обсуждение

Большинство клавиш на клавиатуре не выполняет никаких действий, когда Vim находится в режиме подстановки с подтверждением, что довольно необычно. Как всегда, клавиша <Esc> позволяет вернуться в командный режим, но остальные клавиши, кроме упомянутых выше, не дают никакого эффекта.

С одной стороны, такое поведение дает нам возможность выполнить операцию с минимумом нажатий на клавиши. Но с другой –

никакие другие функциональные возможности в этом режиме недоступны. «Формула точки» (как было показано в рецепте 5), напротив, позволяет нам все время оставаться в старом добром командном режиме, сохраняя доступ ко всем возможностям редактора.

Попробуйте использовать оба метода и выберите для себя тот, что лучше соответствует вашим чаяньям.

Рецепт 91. Повторное использование последнего шаблона поиска

Если в команде подстановки оставить поле шаблона пустым, она задействует последний использовавшийся шаблон поиска. Мы можем воспользоваться этой особенностью, чтобы спрямить путь к достижению результата.

Честно говоря, чтобы выполнить команду подстановки, требуется много раз нажать на клавиши. Сначала нужно определить диапазон, затем заполнить поля шаблона и строки замены и, наконец, добавить необходимые флаги. Такой объем работы требует вдумчивого отношения, а любая ошибка может кардинально изменить результат.

Однако не все так плохо: если оставить поле шаблона пустым, Vim будет использовать текущий шаблон.

Возьмем для примера следующую монолитную команду подстановки (из рецепта 85 в главе 13):

```
⇒ :%s/\v'(([^']|'\w)+)'/"\1"/g
```

Она эквивалентна следующим двум отдельным командам:

```
⇒ /\v'(([^']|'\w)+)'
⇒ :%s//"\1"/g
```

Ну и что? Какой бы путь мы ни выбрали, нам все равно придется вводить шаблон целиком, разве не так? Так, да не совсем. В большинстве случаев команда подстановки создается в два этапа: создание шаблона и определение строки замены. Второй подход позволяет нам разделить одну большую задачу на две.

При создании нетривиальных регулярных выражений часто приходится выполнить несколько попыток, прежде чем будет достигнут желаемый результат. Если тестирование регулярного выражения выполнять в команде подстановки, каждая попытка будет вносить

изменения в документ. Это совершенно неприемлемо. Когда тестирование выполняется с помощью команды поиска, документ не изменяется, благодаря чему мы можем позволить себе ошибаться. В рецепте 85 (глава 13) мы познакомились с эффективным способом создания регулярных выражений. Разделение задач упрощает их решение. У нас появляется возможность семь раз отмерить и один раз отрезать.

Кроме того, кто сказал, что мы должны вводить шаблон? В рецепте 87 (глава 13) был продемонстрирован сценарий, реализующий эквивалент команды *, работающий в визуальном режиме. Этот сценарий дает возможность выделить любой текст в документе и найти совпадения с ним, нажав клавишу *. Полученный таким способом шаблон можно использовать в команде подстановки для замены выделенного текста чем-то другим. Впору поспорить, кто окажется ленивее!

Этот прием подходит не всегда

Я не говорил, что вы никогда не должны заполнять поле шаблона в команде подстановки. Ниже приводится пример команды подстановки, объединяющей все строки в файле, с замещением всех символов перевода строки запятыми:

```
⇒ :%s/\n/,
```

Нет смысла разбивать создание такой команды на два этапа. Более того, такое разбиение только добавит работы.

Влияние на историю команд

Важно также помнить, что при выполнении команды подстановки с пустым полем шаблона создается неполная запись в истории команд. Шаблоны сохраняются редактором в истории поиска, а сами команды подстановки – в истории команд Ex (`:h cmdline-history` ⓘ http://vimdoc.sourceforge.net/htmldoc/cmdline.html#cmdline-history). Выполняя деление на две задачи, поиска и замены, мы разносим информацию по двум разным хранилищам, что может вызвать сложности, если позднее потребуется повторно выполнить команду подстановки.

Если вам кажется, что позднее может потребоваться восстановить команду подстановки в полном ее виде из истории команд, вы

всегда можете заполнить поле шаблона явно. Нажатие комбинации `<C-r>/` в строке ввода вставит последний использовавшийся шаблон в команду. Так, следующая последовательность добавит в историю команд полную запись:

```
⇒ :%s/<C-r>//"\1"/g
```

Иногда бывает нежелательно оставлять пустым поле шаблона в команде подстановки. Иногда – наоборот. Используйте оба способа, и со временем вы научитесь определять, когда и какой способ использовать. Как всегда, полагайтесь на свои суждения.

Рецепт 92. Замена содержимым регистра

Мы не обязаны вводить строку замены вручную. Если требуемый текст уже имеется в документе, его можно скопировать в регистр и использовать в качестве строки замены. При этом содержимое регистра можно передавать как по значению, так и по ссылке.

В рецепте 91 выше мы видели, что редактор способен делать вполне разумные выводы при вызове команды подстановки с пустым полем шаблона. Исходя из этого, можно было бы подумать, что если оставить пустым поле строки замены, то редактор автоматически будет использовать строку замены из предыдущей команды подстановки, но это не так. Если оставить поле строки замены пустым, редактор просто заменит все совпадения пустой строкой. Иными словами, он удалит все найденные совпадения.

Передача по значению

Содержимое регистра можно вставить командой `<C-r>{register}`. Допустим, что мы уже скопировали текст, который требуется вставить в поле строки замены команды подстановки. Теперь мы можем вставить содержимое регистра в команду, как показано ниже:

```
⇒ :%s//<C-r>0/g
```

Когда мы вводим `<C-r>0`, Vim вставляет содержимое регистра 0, благодаря чему мы сможем проверить команду замены перед выполнением. Во многих ситуациях это вполне приемлемое решение, но иногда такой подход может вызывать сложности.

Если текст в регистре 0 содержит какие-либо символы, имеющие специальное значение в строке замены (например, такие как & или ~), нам может потребоваться вручную отредактировать команду, чтобы экранировать такие символы. Кроме того, если регистр 0 содержит многострочный текст, он может не уместиться в командную строку.

Чтобы избежать подобных неприятностей, можно вместо содержимого регистра передать ссылку на этот регистр.

Передача по ссылке

Допустим, что мы сохранили многострочный текст в регистре 0. Теперь нам нужно использовать этот текст в качестве строки замены в команде подстановки. Мы могли бы ввести такую команду:

```
➾ :%s//\=@0/g
```

Последовательность `\=` в поле замены предписывает редактору выполнить выражение. В выражениях можно указывать ссылки на регистры, имеющие вид: `@{register}`. Ссылка `@0` вернет содержимое регистра захвата, а ссылка `@"`» вернет содержимое регистра по умолчанию (неименованный регистр). То есть выражение `:%s//\=@0/g` предписывает редактору заменить совпадения с последним шаблоном поиска содержимым регистра захвата.

Сравнение

Взгляните на следующую команду:

```
➾ :%s/Pragmatic Vim/Practical Vim/g
```

Сравните ее с последовательностью команд:

```
➾ :let @/='Pragmatic Vim'
➾ :let @a='Practical Vim'
➾ :%s//\=@a/g
```

Команда `:let @/='Pragmatic Vim'` – это удобный способ определения шаблона поиска. Она производит тот же эффект, что и команда поиска `/Pragmatic Vim<CR>` (за исключением того, что команда `:let @/='Pragmatic Vim'` не создает запись в истории поиска).

Аналогично команда `:let @a='Practical Vim'` сохраняет содержимое в регистре `a`. Конечный результат получается тем же,

как если бы мы выделили текст «Practical Vim» в визуальном режиме и затем ввели `"ay`, чтобы скопировать выделенный текст в регистр `a`.

Обе команды подстановки делают одно и то же – они замещают все вхождения фразы «Pragmatic Vim» на «Practical Vim». А теперь подумайте о последствиях, которые влечет каждый из подходов.

В первом случае в истории команд создается запись `:%s/Pragmatic Vim/Practical Vim/g`, не допускающая неоднозначностей. Если позднее, в этом же сеансе редактирования, нам потребуется повторить эту команду подстановки, мы сможем вызвать ее из истории команд и выполнить еще раз. Без каких-либо неожиданностей.

Во втором случае в истории команд создается запись `:%s//\=@a/g`. Выглядит довольно туманно, не находите?

Когда эта команда выполнялась в первый раз, роль шаблона играла строка «Pragmatic Vim», а регистр `a` хранил текст «Practical Vim». Но за полчаса-час текущий шаблон поиска мог измениться десять раз, и содержимое регистра могло быть затерто чем-то другим. То есть если спустя какое-то время повторить команду `:%s//\=@a/g`, мы можем получить совершенно другой эффект!

Однако эту особенность можно использовать в своих интересах. Мы могли бы найти текст, который можно было бы использовать в качестве строки замены, и скопировать его в регистр `a`. Затем повторить команду `:%s//\=@a/g`, автоматически использующую подготовленные значения `@/` и `@a`. Потом мы могли бы отыскать какой-то другой текст, определить его в качестве строки замены, сохранив в регистре `a`, и вновь повторить команду `:%s//\=@a/g`, которая на этот раз произвела бы совершенно другой эффект.

Попробуйте! Такой прием может нравиться или, наоборот, вызывать неприязнь, но в любом случае это довольно интересный трюк!

Рецепт 93. Повторение предыдущей команды подстановки

Иногда вам может потребоваться изменять диапазон действия команды подстановки. Например, если была допущена ошибка при первой попытке или при необходимости выполнить ту же команду подстановки с другим буфером. Существует пара приемов, упрощающих повторной вызов команды подстановки.

Повторение команды подстановки в строке ко всему файлу

Допустим, что мы только что выполнили следующую команду, воздействующую на текущую строку:

```
⇛ :s/target/replacement/g
```

А после этого заметили ошибку: мы забыли добавить префикс %. Но нет причин для расстройства. Мы можем повторить команду и применить ее ко всему файлу, просто нажав g& (см. :h g& ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#g&), что эквивалентно следующей команде:

```
⇛ :%s//~/&
```

Она расшифровывается как «повторить последнюю команду подстановки с теми же флагами, с той же строкой замены и с текущим шаблоном поиска, но использовать диапазон %. Другими словами: применить последнюю команду подстановки ко всему файлу.

В следующий раз, когда вы поймаете себя на желании добавить префикс % к команде подстановки из истории команд, попробуйте просто нажать клавиши g&.

Изменение диапазона в команде подстановки

В качестве примера возьмем следующий код:

substitution/mixin.js
http://media.pragprog.com/titles/dnvim/code/substitution/mixin.js

```
mixin = {
     applyName: function(config) {
          return Factory(config, this.getName());
     },
}
```

Допустим, что его потребовалось дополнить, как показано ниже:

```
mixin = {
     applyName: function(config) {
          return Factory(config, this.getName());
     },
     applyNumber: function(config) {
          return Factory(config, this.getNumber());
     },
}
```

Новая функция практически идентична прежней. Поэтому начнем с того, что создадим копию функции `applyName`, а затем с помощью команды подстановки заменим вхождения слова «Name» на «Number». Весь процесс показан в табл. 14.1.

Сможете ли вы найти здесь ошибку? Мы использовали диапазон `%`, из-за чего команда заменила все вхождения «Name» на «Number», тогда как нам следовало ограничить область действия команды второй функцией (копией).

Не волнуйтесь. Мы легко можем отменить изменения и исправить ошибку (как показано в табл. 14.2).

Команда `gv` выполняет переход в визуальный режим и восстанавливает последнее выделение (об этом рассказывалось в рецепте 21 в главе 4). Затем мы нажимаем `:` в визуальном режиме, и командная строка заполняется диапазоном `:'<,'>`, который ограничит область действия следующей за ним команды Ex выделенными строками.

Таблица 14.1. Ошибка при использовании команды подстановки

Нажатия клавиш	Содержимое буфера

`{start}`	`mixin = {`
	`    applyName: function(config) {`
	`        return Factory(config, this.getName());`
	`    },`
	`}`

`Vjj`	`mixin = {`
	`    applyName: function(config) {`
	`        return Factory(config, this.getName());`
	`    },`
	`}`

`yP`	`mixin = {`
	`    applyName: function(config) {`
	`        return Factory(config, this.getName());`
	`    },`
	`    applyName: function(config) {`
	`        return Factory(config, this.getName());`
	`    },`
	`}`

`:%s/Name/Number/g`	`mixin = {`
	`    applyNumber: function(config) {`
	`        return Factory(config, this.getNumber());`
	`    },`
	`    applyNumber: function(config) {`
	`        return Factory(config, this.getNumber());`
	`    },`
	`}`

Команда :&& требует дополнительных пояснений, поскольку первый и второй символы & в ней имеют разное значение. Первый из них образует команду Ex :&, которая повторяет последнюю команду :substitute (см. :h :& ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#%3A%26), а второй указывает, что должны использоваться флаги из предыдущей команды :s.

Обсуждение

Мы всегда можем указать новый диапазон и повторить подстановку командой :&&. Не важно, какой диапазон использовался в последний раз. Сама команда :&& воздействует на текущую строку, команда :'<,'>&& воздействует на визуальное выделение, и команда :%&& воздействует на весь файл. Как мы уже видели, команда g& является удобным сокращением для команды :%&&.

Таблица 14.2. Изменение диапазона для команды подстановки

Нажатия клавиш	Содержимое буфера

u	`mixin = {`
	`    applyName: function(config) {`
	`        return Factory(config, this.getName());`
	`    },`
	`    applyName: function(config) {`
	`        return Factory(config, this.getName());`
	`    },`
	`}`

gv	`mixin = {`
	`    applyName: function(config) {`
	`        return Factory(config, this.getName());`
	`    },`
	`    applyName: function(config) {`
	`        return Factory(config, this.getName());`
	`    },`
	`}`

:'<,'>&&	`mixin = {`
	`    applyName: function(config) {`
	`        return Factory(config, this.getName());`
	`    },`
	`    applyNumber: function(config) {`
	`        return Factory(config, this.getNumber());`
	`    },`
	`}`

Исправление команды &

Команда & является синонимом команды :s, которая выполняет последнюю подстановку. К сожалению, если использовались какие-либо флаги, команда & игнорирует их, а это означает, что ее результат может отличаться от предыдущей команды подстановки.

Команда :&&, основанная на команде &, выглядит предпочтительнее. Она сохраняет флаги и потому производит более ожидаемые результаты. Следующие привязки исправляют поведение команды & в командном режиме и создают эквивалент для визуального режима:

```
nnoremap & :&&<CR>
xnoremap & :&&<CR>
```

Рецепт 94. Переупорядочение полей в файле CSV

В этом рецепте рассказывается, как можно использовать сохраняемые совпадения с подвыражениями в круглых скобках в поле со строкой замены.

Допустим, у нас имеется файл в формате CSV, содержащий список электронных адресов с именами и фамилиями адресатов:

substitution/subscribers.csv
http://media.pragprog.com/titles/dnvim/code/substitution/subscribers.csv

```
last name,first name,email
neil,drew,drew@vimcasts.org
doe,john,john@example.com
```

Теперь допустим, что нам требуется изменить порядок следования полей так, чтобы электронный адрес находился в первом поле, имя – во втором, а фамилия – в третьем. Для решения этой задачи можно было бы использовать следующую команду подстановки:

```
➾ /\v^([^,]*),([^,]*),([^,]*)$
➾ :%s//\3,\2,\1
```

В этом шаблоне класс [^,] соответствует любым символам, кроме запятой. Поэтому подвыражение ([^,]*) соответствует последовательности из нуля или более символов, не являющихся запятыми, и сохраняет ее (см. рецепт 76 в главе 12). Это подвыражение повторяется трижды, чтобы захватить каждое поле в файле CSV.

На совпадения с подвыражениями можно ссылаться с помощью нотации `\n`. То есть в строке замены ссылка `\1` будет ссылаться на поле с именем, `\2` – на поле с фамилией и `\3` – на поле с электронным адресом. Разделив каждую строку на отдельные поля, мы можем переупорядочить их в соответствии с условиями задачи, то есть `\3,\2\,1` – электронный адрес, фамилия, имя.

Ниже приводится результат выполнения этой команды:

```
email,first name,last name
drew@vimcasts.org,drew,neil
john@example.com,john,doe
```

Рецепт 95. Выполнение арифметических операций в строке замены

Поле замены в команде подстановки необязательно должно содержать простую строку. В нем можно указывать выражения на языке сценариев Vim и использовать результат в качестве строки замены. То есть единственной командой мы можем преобразовать все теги заголовков в документе HTML.

Допустим, что у нас имеется такой HTML-документ:

substitution/headings.html
http://media.pragprog.com/titles/dnvim/code/substitution/headings.html

```
<h2>Heading number 1</h2>
<h3>Number 2 heading</h3>
<h4>Another heading</h4>
```

Нам нужно увеличить уровень каждого заголовка, то есть заголовок `<h2>` превратить `<h1>`, `<h3>` – в `<h2>` и т. д. Проще говоря, нужно вычесть единицу из числовой части всех HTML-тегов заголовков.

Для решения этой задачи воспользуемся командой подстановки. Вот основная идея: мы разработаем шаблон, соответствующий числовым частям HTML-тегов заголовков. Затем разработаем команду подстановки, использующую выражение на языке сценариев Vim для вычитания единицы из сохраненной числовой части. Когда такая команда будет применена ко всему файлу, она изменит все HTML-теги заголовков в файле.

Шаблон поиска

Единственное, что нам нужно изменить, – числовую часть в тегах заголовков, поэтому в идеале нам нужно создать шаблон, который будет совпадать только с ней и ни с чем больше. Нам не нужно совпадение со случайными цифрами в документе – только с теми, которым предшествуют символы `<h` или `</h`. Необходимое совпадение обеспечивает следующий шаблон:

```
⇒ /\v\<\/?h\zs\d
```

Специальный символ `\zs` позволяет ограничить совпадение. Для простоты можно сказать, что шаблон `h\zs\d` будет совпадать с буквой «h», за которой следует любая цифра («h1», «h2» и т. д.). Местоположение специального символа `\zs` указывает, что сам символ «h» должен исключаться из совпадения, даже при том что он является составной частью шаблона (мы уже встречались со специальным символом `\zs` в рецепте 77 главы 12, где сравнивали его с положительной ретроспективной проверкой в языке Perl). Наш шаблон выглядит немного сложнее только потому, что нам требуется, чтобы он совпадал не просто с «h1» и «h2», а с «<h1», «</h1», «<h2», «</h2» и т. д.

Попробуйте выполнить поиск по этому шаблону у себя. В результате все числовые части тегов заголовков должны быть подсвечены, а остальные цифры, имеющиеся в документе, – нет.

Команда подстановки

Нам нужно выполнить арифметическую операцию в поле замены команды подстановки. Для этого следует использовать выражение на языке сценариев Vim. Извлечь текущее совпадение можно с вызовом функции `submatch(0)`. Так как наш шаблон соответствует только цифре и ничему другому, можно смело полагать, что `submatch(0)` вернет число. Нам остается только вычесть из него единицу и вернуть результат, чтобы команда подстановки вставила его на место совпадения.

С этой работой должна справиться следующая команда подстановки:

```
⇒ :%s//\=submatch(0)-1/g
```

После выполнения команды поиска и вслед за ней команды подстановки мы получим следующий результат:

```
<h1>Heading number 1</h1>
<h2>Number 2 heading</h2>
<h3>Another heading</h3>
```

Все HTML-теги заголовков изменились, а посторонние цифры, имевшиеся в документе, остались нетронутыми.

Рецепт 96. Перемена местами двух и более слов

Мы можем создать команду подстановки, которая будет менять местами все вхождения одного слова с вхождениями другого, используя регистр выражений и структуру-словарь на языке Vim.

Возьмем для примера следующий фрагмент:

substitution/who-bites.txt
http://media.pragprog.com/titles/dnvim/code/substitution/who-bites.txt

```
The dog bit the man.
```

Допустим, что нам нужно поменять местами слова «dog» и «man». Мы могли бы, конечно, использовать последовательность операций копирования и вставки, как описывалось в разделе «Как поменять слова местами» в главе 10. Но сейчас нам интересно выяснить, можно ли реализовать то же самое с помощью команды подстановки.

Ниже приводится первая, наивная попытка решить задачу:

```
➾ :%s/dog/man/g
➾ :%s/man/dog/g
```

Первая команда замещает слово «dog» словом «man», в результате чего получается фраза «the man bit the man». Вторая команда замещает оба вхождения слова «man» словом «dog», что в конечном итоге дает фразу «the dog bit the dog». Очевидно, что нам придется продолжить попытки.

Двухпроходное решение оказалось неудачным, поэтому нам необходимо создать такую команду подстановки, которая будет выполнять все необходимое за один проход. Самое простое здесь – написать шаблон, обнаруживающий слова «dog» и «man». Самое

сложное – написать выражение, принимающее одно слово и возвращающее другое. Решим эту часть головоломки первой.

Возврат другого слова

Нам даже не нужно создавать функцию, чтобы решить задачу. Достаточно будет воспользоваться структурой-словарем, создав две пары ключ/значение. Попробуйте ввести в Vim следующие команды:

```
➾ :let swapper={"dog":"man","man":"dog"}
➾ :echo swapper["dog"]
  man
➾ :echo swapper["man"]
  dog
```

Если теперь передать словарю `swapper` ключ `"dog"`, он вернет `"man"`, и наоборот.

Поиск совпадений с двумя словами

Вы уже придумали шаблон для поиска? Вот он:

```
➾ /\v(<man>|<dog>)
```

Этот шаблон просто совпадает с целым словом «man» или с целым словом «dog». Круглые скобки используются для сохранения совпадения, чтобы на него можно было сослаться в строке замены.

Все вместе

Теперь объединим все вместе. Для начала выполним команду поиска. Она должна найти все вхождения слов «dog» и «man». Затем, когда мы попробуем выполнить команду подстановки, можно будет оставить поле шаблона пустым (как описывалось в рецепте 91 выше).

Для строки замены нам необходимо подготовить небольшое выражение на языке сценариев Vim. То есть мы будем использовать специальный символ `\=`. На этот раз мы не будем присваивать словарь переменной, а просто создадим временный словарь для однократного использования.

Обычно для ссылки на совпадения с подвыражениями в Vim используется форма записи `\1`, `\2` (и т. д.). Но в выражениях сохраненные совпадения необходимо извлекать вызовом функции `submatch()` (см. `:h submatch()` ⓘ http://vimdoc.sourceforge.net/htmldoc/eval.html#submatch()).

В окончательном виде решение поставленной задачи выглядит, как показано ниже:

```
⇛ /\v(<man>|<dog>)
⇛ :%s//\={"dog":"man","man":"dog"}[submatch(1)]/g
```

Обсуждение

Это был не самый лучший пример! Нам пришлось трижды вводить слова «man» и «dog». Очевидно, что гораздо быстрее получилось бы просто заменить два слова в документе. Но при работе с большими объемами текста, где каждое слово встречается множество раз, подобные дополнительные усилия быстро окупаются. Обратите внимание, что это решение легко можно адаптировать для случая перестановки трех и более слов за один проход.

Проблема, связанная с большим объемом ручного ввода, осталась. Однако, написав чуть больше кода на языке сценариев Vim, мы могли бы реализовать собственную команду с более дружественным интерфейсом, которая выполняла бы всю рутинную работу автоматически. Эта тема выходит далеко за рамки данной книги, тем не менее прочитайте врезку «Abolish.vim: перегруженная команда подстановки» ниже.

Abolish.vim: перегруженная команда подстановки

Расширение Abolish.vim, написанное Тимом Поупом (Tim Pope), является одним из моих самых любимых[1]. Оно реализует новую команду `:Subvert` (или `:S`), которая действует как перегруженная версия команды `:substitute`. С помощью этого расширения мы могли бы поменять местами слова «man» и «dog», выполнив следующую команду:

```
⇛ :%S/{man,dog}/{dog,man}/g
```

Она не только короче, но и гораздо гибче нашего варианта. Помимо слов «man» и «dog», она также поменяет местами слова «MAN» и «DOG», «Man» и «Dog». Этот коротенький пример – лишь верхушка айсберга возможностей данного потрясающего расширения. Я настоятельно рекомендую найти время и исследовать другие его возможности.

---

[1] https://github.com/tpope/vim-abolish

Рецепт 97. Поиск и замена в нескольких файлах

Команда подстановки действует в пределах текущего файла. А как быть, если нам потребуется применить ту же команду подстановки ко всему проекту? Подобная необходимость часто возникает на практике, однако редактор Vim не имеет отдельной команды поиска с заменой, которую можно было применить ко всему проекту. Впрочем, в этом нет необходимости, данную функциональность можно получить, объединив пару простых команд.

Начнем со знакомства с сырым, но эффективным решением, а затем посмотрим, как его улучшить. Для демонстрации мы будем использовать каталог *refactor-project*, который можно найти в пакете с загружаемыми примерами к книге. Ниже приводятся файлы, находящиеся в нем, и их содержимое:

```
refactor-project/
     about.txt
     Pragmatic Vim is a hands-on guide to working with Vim.

     credits.txt
     Pragmatic Vim is written by Drew Neil.

     license.txt
     The Pragmatic Bookshelf holds the copyright for this book.

     extra/
          praise.txt
          What people are saying about Pragmatic Vim...

          titles.txt
          Other titles from the Pragmatic Bookshelf...
```

Каждый из этих файлов содержит слово «Pragmatic» либо как часть фразы «Pragmatic Bookshelf», либо как часть фразы «Pragmatic Vim». Нам нужно найти каждое вхождение фразы «Pragmatic Vim» и заменить ее на «Practical Vim», оставив фразы «Pragmatic Bookshelf» нетронутыми.

Если вы желаете опробовать примеры ниже, загрузите пакет с исходными текстами со страницы книги на сайте Pragmatic Bookshelf. Прежде чем открыть Vim, перейдите в каталог *refactor-project*.

Команда подстановки

Начнем с создания команды подстановки. Нам нужно сконструировать шаблон, который совпадал бы со словом «Pragmatic» во фра-

зе «Pragmatic Vim» и нигде в другом месте. Этим условиям отвечает следующий шаблон:

```
⇒ /Pragmatic\ze Vim
```

В нем используется специальный символ `\ze`, исключающий слово «Vim» из совпадения (см. рецепт 78 в главе 12). Выполнив поиск с этим шаблоном, можно вызвать следующую команду подстановки:

```
⇒ :%s//Practical/g
```

Теперь нам осталось выяснить, как применить эту команду ко всему проекту.

Мы сделаем это в два этапа. Сначала по всему проекту выполним поиск совпадений с заданным шаблоном. Затем применим команду подстановки к файлам, где будут найдены совпадения.

Поиск во всех файлах в текущем проекте с использованием ':vimgrep'

Чтобы выполнить поиск во всех файлах в проекте, мы воспользуемся командой `:vimgrep` (см. рецепт 111 в главе 18). Поскольку она использует встроенный механизм поиска Vim, мы применим все тот же шаблон. Попробуйте выполнить следующие команды:

```
⇒ /Pragmatic\ze Vim
⇒ :vimgrep // **/*.txt
```

Поле шаблона в этой команде ограничено двумя соседними символами `/`. Мы оставили поле пустым, потребовав от редактора Vim выполнить `:vimgrep` с применением последнего шаблона. Шаблон `**/*.txt` требует от `vimgrep` просмотреть содержимое всех файлов с расширением `.txt` в текущем каталоге.

Применение команды подстановки ко всем файлам в текущем проекте с использованием ':cfdo'

Каждое совпадение, найденное командой `vimgrep`, сохраняется в списке результатов (quickfix list, см. главу 17 «Компиляция кода и обзор ошибок с помощью Quickfix List»). Ознакомиться с резуль-

татами поиска можно с помощью команды `:copen`, которая открывает окно со списком результатов. Но нам не нужны результаты как таковые, нам нужно применить команду подстановки ко всем файлам, оказавшимся в этом списке. Сделать это можно, применив команду `:cfdo` (см. `:h :cfdo` ⓘ http://vimhelp.appspot.com/quickfix.txt.html#%3Acfdo).

Прежде чем применить команду `:cfdo`, убедимся, что параметр `hidden` включен:

```
⇒ :set hidden
```

Этот параметр позволяет выходить из измененного файла без предварительного его сохранения. Более подробное обсуждение вы найдете в рецепте 38 в главе 6.

Теперь можно применить команду подстановки ко всем файлам в списке результатов (quickfix list):

```
⇒ :cfdo %s//Practical/gc
```

Флаг c здесь является необязательным. Он позволяет осмотреть каждое совпадение и решить – выполнять подстановку или нет (см. рецепт 90). Завершим, записав изменения на диск:

```
⇒ :cfdo update
```

Команда `:update` сохраняет файл, но только если он был изменен (см. `:h update` ⓘ http://vimhelp.appspot.com/editing.txt.html#%3Aupdate).

Обратите внимание, что последние две команды можно объединить в одну:

```
⇒ :cfdo %s//Practical/g | update
```

Символ `|` имеет иной смысл в командах Vim, чем в командной оболочке. В Unix символ вертикальной черты передает стандартный вывод одной команды в стандартный ввод следующей команды (создает «конвейер»). В Vim символ `|` действует как простой разделитель команд, эквивалентный точке с запятой в командной оболочке Unix. Подробности см. в разделе справки `:h :bar` ⓘ http://vimhelp.appspot.com/cmdline.txt.html#%3Abar.

В заключение

Ниже приводится полная последовательность команд:

```
⇒ /Pragmatic\ze Vim
⇒ :vimgrep // **/*.txt
⇒ :cfdo %s//Practical/gc
⇒ :cfdo update
```

Сначала мы составили шаблон поиска и убедились, что он работает в текущем буфере. Затем мы использовали `:vimgrep` для поиска по всем файлам в проекте по тому же шаблону и заполнили список результатов (quickfix list). Далее мы выполнили обход файлов в списке quickfix list, воспользовавшись командой `:cfdo` для выполнения команд `:substitute` и `:update`.

Глава 15. Глобализация команд

Команда `:global` соединяет в себе мощь команд Ex с возможностями поиска по шаблону. Ее можно использовать для применения команд Ex к каждой строке, соответствующей указанному шаблону. Наряду с «формулой точки» и макросами команда `:global` является одним из самых мощных инструментов Vim, предназначенных для эффективного выполнения повторяющихся действий.

Рецепт 98. Встречайте: команда :global

Команда `:global` дает возможность применить команду Ex к каждой строке, соответствующей определенному шаблону. Для начала познакомимся с синтаксисом команды.

Команда `:global` имеет следующий синтаксис (см. `:h :g` ⓘ http://vimdoc.sourceforge.net/htmldoc/repeat.html#:g):

```
:[range] global[!] /{pattern}/ [cmd]
```

Диапазоном `[range]` для команды `:global` по умолчанию является весь файл (`%`). Этим она отличается от других команд Ex, включая `:delete`, `:substitute` и `:normal`, которые по умолчанию воздействуют на текущую строку (`.`).

Поле шаблона `{pattern}` интегрируется с историей поиска, в том смысле что его можно оставить пустым, в этом случае будет использоваться текущий шаблон поиска.

В поле команды `[cmd]` можно указать любую команду Ex, кроме самой команды `:global`. На практике чаще всего используются команды Ex, выполняющие операции с текстом документа, как те, что перечислены в табл. 5.1. Если команда в поле `[cmd]` не указана, по умолчанию будет использоваться команда `:print`.

Чтобы изменить смысл команды `:global` на противоположный, ее можно вызвать как `:global!` или `:vglobal` (мнемоника: *inVert* –

обратное значение). Каждая из них применит указанную команду `[cmd]` к каждой строке, *не соответствующей* заданному шаблону. В следующем рецепте будут представлены примеры использования обеих команд, `:global` и `:vglobal`.

Команда `:global` выполняет обработку строк в указанном диапазоне `[range]` в два прохода. На первом проходе отмечаются строки, соответствующие шаблону `{pattern}`. На втором проходе к отмеченным строкам применяется команда `[cmd]`. Команда `[cmd]` может принимать собственный диапазон, что дает возможность воздействовать на многострочные области. Этот мощнейший прием демонстрируется в рецепте 101.

Рецепт 99. Удаление строк, соответствующих шаблону

Комбинация команд `:global` и `:delete` позволяет быстро сокращать файлы в размерах. С помощью этой комбинации можно удалить или оставить строки, соответствующие шаблону `{pattern}`.

Следующий файл содержит ссылки на первые несколько видеоуроков в архиве сайта Vimcasts.org:

global/episodes.html
http://media.pragprog.com/titles/dnvim/code/global/episodes.html

```
<ol>
  <li>
    <a href="/episodes/show-invisibles/">
      Show invisibles
    </a>
  </li>
  <li>
    <a href="/episodes/tabs-and-spaces/">
      Tabs and Spaces
    </a>
  </li>
  <li>
    <a href="/episodes/whitespace-preferences-and-filetypes/">
      Whitespace preferences and filetypes
    </a>
  </li>
</ol>
```

Каждый элемент списка состоит из двух частей: названия видеоурока и его адреса URL. Мы собираемся использовать команду `:global` для выявления каждой из частей.

Удаление соответствующих строк командой :g/re/d

Допустим, что нам требуется выбросить все, кроме содержимого тегов <a>. В этом файле содержимое каждой ссылки находится в отдельной строке, а во всех остальных строках только открывающие и закрывающие теги. То есть если мы сможем создать шаблон, соответствующий любому тегу HTML, мы сможем использовать в команде `:global` для удаления любых строк, соответствующих этому шаблону.

О происхождении слова «grep»

Взгляните на следующую сокращенную форму команды `:global`:

```
➾ :g/re/p
```

Слово `re` в данном случае означает «regular expression» (регулярное выражение), а «p» соответствует команде `:print`, которая по умолчанию подставляется в пустое поле `[cmd]`. Если удалить символы `/`, мы получим слово «grep».

Все это сделают следующие команды:

```
➾ /\v\<\/?\w+>
➾ :g//d
```

Если применить эти две команды к файлу выше, в нем останутся только строки:

```
Show invisibles
Tabs and Spaces
Whitespace preferences and filetypes
```

В команде `:global` можно оставить поле шаблона пустым, как и в команде `:substitute`, и Vim будет повторно использовать последний шаблон (см. рецепт 91 в главе 14). Это означает, что конструирование регулярного выражения можно начать с поиска более широкого совпадения и затем уточнить его в несколько этапов, как было показано в рецепте 85 в главе 13.

Регулярное выражение в команде `:global` сопоставляется в *самом волшебном режиме* (very magic), описанном в рецепте 74 в главе 12. В данном случае совпадение должно начинаться открывающей угловой скобкой (\<), за которой следуют необязательный слэш

(\/?) и один или более символов слова (\w+), и заканчиваться разделителем (>). Это регулярное выражение не является универсальным в смысле поиска тегов HTML, но оно достаточно хорошо подходит для данного конкретного случая.

Сохранение только соответствующих строк командой :v/re/d

Теперь перевернем все с ног на голову. Команда `:vglobal`, или ее краткая форма `:v`, является противоположностью команде `:g`. То есть она применит указанную команду к каждой строке, не соответствующей заданному шаблону.

Строки, содержащие адреса URL, легко идентифицировать: все они содержат атрибут `href`. Оставить только эти строки можно командой

```
➾ :v/href/d
```

Ее можно прочитать как «Удалить все строки, *не содержащие* `href`». В результате в файле останутся строки:

```
<a href="/episodes/show-invisibles/">
<a href="/episodes/tabs-and-spaces/">
<a href="/episodes/whitespace-preferences-and-filetypes/">
```

Одной командой мы оставили в файле только интересующие нас строки.

Рецепт 100. Выборка комментариев TODO в регистр

Комбинация команд `:global` и `:yank` позволяет сохранить в регистре все строки, соответствующие указанному шаблону.

Следующий фрагмент кода содержит пару комментариев, начинающихся со слова «TODO»:

global/markdown.js
http://media.pragprog.com/titles/dnvim/code/global/markdown.js

```
Markdown.dialects.Gruber = {
    lists: function() {
        // TODO: Cache this regexp for certain depths.
```

```
            function regex_for_depth(depth) { /* implementation */ }
        },
        "`": function inlineCode( text ) {
            var m = text.match( /(`+)(([\s\S]*?)\1)/ );
            if ( m && m[2] )
                return [ m[1].length + m[2].length ];
            else {
                // TODO: No matching end code found - warn!
                return [ 1, "`" ];
            }
        }
}
```

Допустим, что нам требуется собрать все комментарии TODO в одном месте. Вывести их все можно одной командой:

```
⇒  :g/TODO
    // TODO: Cache this regexp for certain depths.
       // TODO: No matching end code found - warn!
```

Не забывайте, что команда `:print` используется командой `:global` по умолчанию, если поле `[cmd]` оставить пустым. Она просто выведет все строки, содержащие слово «TODO». Впрочем, это не самое удачное решение, потому что сообщения исчезнут, как только будет выполнена другая команда.

Можно поступить иначе: скопировать каждую строку со словом «TODO» в регистр, а затем вставить содержимое этого регистра в другой файл и сохранить для последующего использования.

В примере ниже мы будем использовать регистр `a`. Прежде всего его следует очистить командой `qaq`, где `qa` предписывает редактору Vim начать запись макроса в регистр `a`, а `q` останавливает запись. В процессе записи макроса ничего не вводилось, поэтому в результате регистр очищается. Убедиться в этом можно с помощью команды

```
⇒  :reg a
   --- Registers ---
   "a
```

Теперь можно приступать к копированию комментариев TODO в регистр:

```
⇒  :g/TODO/yank A
⇒  :reg a
   "a // TODO: Cache this regexp for certain depths.
         // TODO: No matching end code found - warn!
```

Вся хитрость здесь заключена в адресации регистра заглавной буквой `A`. Таким способом мы сообщаем редактору, что он должен *добавлять* новый текст в конец регистра, тогда как при адресации строчной буквой `a` Vim будет затирать прежнее содержимое регистра в каждой операции копирования (захвата). Эту команду `:global` можно интерпретировать как «Каждую строку, соответствующую шаблону `/TODO/`, добавить в регистр `a`".

Если теперь выполнить команду `:reg a`, мы увидим, что регистр содержит два комментария TODO. (Для ясности я отформатировал вывод команды `:reg a`, в действительности же Vim выводит все в одну строку, вставляя символы `^J` на место символов перевода строки.) После этого можно было бы открыть новый буфер, создав еще одно окно в рабочей области, и командой `"ap` вставить содержимое регистра `a` в новый документ.

Обсуждение

В этом примере мы просто выбрали два комментария TODO, что достаточно быстро можно было бы сделать и вручную. Но описанный прием хорошо масштабируется. Если документ будет содержать десятки комментариев TODO, на их выборку будет затрачено ровно столько же усилий с нашей стороны.

Мы могли бы даже объединить команду `:global` с командой `:bufdo` или `:argdo`, чтобы выбрать все комментарии TODO из множества файлов. Я оставляю создание такой команды вам в качестве самостоятельного упражнения, но некоторые подсказки, касающиеся процесса разработки, вы найдете в рецепте 36 в главе 5.

Ниже приводится альтернативное решение:

```
⇒ :g/TODO/t$
```

В нем используется команда `:t`, с которой мы познакомились в рецепте 29 в главе 5. Вместо добавления каждого комментария TODO в регистр она просто копирует их в конец файла. После выполнения этой команды можно перейти в конец файла и просмотреть список извлеченных комментариев TODO. Этот прием выглядит более простым, потому что не вынуждает выполнять операции с регистрами. Но он плохо адаптируется для выполнения в комбинации с командами `:argdo` и `:bufdo`.

Рецепт 101. Сортировка свойств в правилах CSS

Объединяя команду Ex с командой `:global`, можно также определить диапазон для команды `[cmd]`. Vim позволяет определять диапазон динамически, с использованием `:g/{pattern}` в качестве точки отсчета. В этом рецепте будет показано, как использовать данную особенность для сортировки свойств в блоках, внутри файла CSS.

Для демонстрации будет использоваться следующий файл:

global/unsorted.css
http://media.pragprog.com/titles/dnvim/code/global/unsorted.css

```
html {
  margin: 0;
  padding: 0;
  border: 0;
  font-size: 100%;
  font: inherit;
  vertical-align: baseline;
}
body {
  line-height: 1.5;
  color: black;
  background: white;
}
```

Допустим, что нам требуется отсортировать свойства внутри каждого правила в алфавитном порядке. Для этого можно было бы использовать встроенную команду `:sort` (`:h :sort` ⓘ http://vimdoc.sourceforge.net/htmldoc/change.html#:sort).

Сортировка свойств внутри одного блока

Давайте для начала попробуем отсортировать с помощью команды `:sort` лишь часть файла (см. табл. 15.1).

Строки внутри фигурных скобок `{}` легко можно выбрать с помощью текстового объекта `vi{`. Последующая команда `:'<,'>sort` отсортирует строки в алфавитном порядке. Этот прием замечательно подходит, когда требуется отсортировать содержимое только одного блока, но давайте полагать, что наша таблица стилей содержит несколько сотен правил. Возможно ли как-то автоматизировать этот процесс?

Таблица 15.1. Сортировка фрагмента файла

<table>
<tr><th>Нажатия клавиш</th><th>Содержимое буфера</th></tr>
<tr><td><pre>{start}</pre></td><td><pre>html {
  margin: 0;
  padding: 0;
  border: 0;
  font-size: 100%;
  font: inherit;
  vertical-align: baseline;
}</pre></td></tr>
<tr><td><pre>vi{</pre></td><td><pre>html {
  margin: 0;
  padding: 0;
  border: 0;
  font-size: 100%;
  font: inherit;
  vertical-align: baseline;
}</pre></td></tr>
<tr><td><pre>:'<,'>sort</pre></td><td><pre>html {
  border: 0;
  font-size: 100%;
  font: inherit;
  margin: 0;
  padding: 0;
  vertical-align: baseline;
}</pre></td></tr>
</table>

Сортировка свойств во всех блоках

Мы можем отсортировать свойства во всех блоках единственной командой `:global`. Например, если применить следующую команду к нашей таблице стилей:

```
⇒ :g/{/ .+1,/}/-1 sort
```

мы получим:

```
html {
  border: 0;
  font-size: 100%;
  font: inherit;
  margin: 0;
  padding: 0;
  vertical-align: baseline;
}
body {
  background: white;
  color: black;
  line-height: 1.5;
}
```

Команда сортировки выполняется внутри блоков `{}`. Наша демонстрационная таблица стилей содержит всего десяток строк, но этот прием с успехом можно применять к значительно более длинным файлам CSS.

Команда `:global` достаточно сложна, поэтому очень важно понять, как она действует, чтобы осознать всю скрытую в ней мощь. Стандартный синтаксис команды имеет следующий вид:

```
:g/{pattern}/[cmd]
```

Напомню, что команды Ex обычно могут принимать собственные диапазоны (как описывалось в рецепте 28 в главе 5). Такая возможность сохраняется и в контексте команды `:global`. Следовательно, шаблон можно развернуть, как показано ниже:

```
:g/{pattern}/[range][cmd]
```

Диапазон `[range]` для нашей команды `[cmd]` может устанавливаться динамически, с использованием `:g/{pattern}`. Обычно диапазон `.` соответствует строке, в которой находится курсор. Но в контексте команды `:global` он означает: «для каждой строки, соответствующей шаблону `{pattern}`».

Мы можем разбить нашу команду на две отдельные команды Ex. Давайте вернемся пройденным путем в обратном направлении. Ниже приводится вполне допустимая команда Ex:

```
➾ :.+1,/}/-1 sort
```

Если выбросить смещения, команда примет вид: `.,/}/`. Ее можно интерпретировать как: «От текущей строки вверх, пока не встретится следующая строка, соответствующая шаблону `/}/`». Смещения +1 и –1 просто сужают диапазон применения команды до содержимого блока в фигурных скобках `{}`. Если поместить курсор в строку 1 или 9 оригинального, несортированного файла CSS, данная команда Ex отсортирует содержимое соответствующего блока `{}`.

Чтобы отсортировать содержимое блока `{}`, достаточно лишь установить курсор в его начало и выполнить команду `:.,/}/sort`. Правильно? Теперь попробуем выполнить поиск с использованием шаблона `{pattern}` из команды `:global`:

```
➾ /{/
```

Она установит курсор в начало блока `{}`, именно туда, куда требуется. Теперь снова объединим команды `:global` и `[cmd]`:

```
⇒ :g/{/ .+1,/}/-1 sort
```

Шаблону `{` соответствуют первые строки в каждом блоке `{}`. Для каждой такой строки выполняется команда `:sort` с диапазоном `[range]`, конец которого совпадает с концом блока `{}`. В результате свойства в каждом блоке внутри файла CSS оказываются отсортированы в алфавитном порядке.

Обсуждение

Обобщенная форма данной команды `:global` имеет следующий вид:

```
:g/{start}/ .,{finish} [cmd]
```

Ее можно интерпретировать как «Для каждого диапазона строк, начинающегося в точке `{start}` и заканчивающегося в точке `{finish}`, выполнить указанную команду `[cmd]`».

Эту форму команды `:global` можно использовать в сочетании с любой командой Ex. Например, допустим, что в указанных диапазонах требуется оформить отступы. Мы могли бы решить эту задачу с помощью команды Ex `:>` (см. `:h :>` http://vimdoc.sourceforge.net/htmldoc/change.html#:>):

```
⇒ :g/{/ .+1,/}/-1 >
  6 lines >ed 1 time
  3 lines >ed 1 time
```

Обратите внимание, что при каждом вызове команда `:>` выводит сообщение, а команда `:sort` – нет. Мы можем подавить вывод сообщений, добавив в команду `[cmd]` префикс `:silent` (см. `:h :sil` http://vimdoc.sourceforge.net/htmldoc/various.html#:sil):

```
⇒ :g/{/sil .+1,/}/-1 >
```

Этот прием особенно полезен, когда команда `:g/{pattern}` обнаруживает совпадение с большим количеством строк.

Часть VI. ИНСТРУМЕНТЫ

Фраза «Пусть каждая программа делает что-то одно, но хорошо» составляет основу философии Unix. Редактор Vim предоставляет обертки команд, упрощающие вызов внешних программ, таких как `make` или `grep`. Для решения некоторых задач требуется более тесная интеграция с текстовым редактором, поэтому Vim предоставляет собственные инструменты для проверки орфографии и автодополнения при вводе, а также встроенную команду `:vimgrep`. В этой части книги мы познакомимся с инструментами из комплекта Vim и его интерфейсом взаимодействия с внешними инструментами.

Глава 16. Индексирование исходного кода и навигация по нему с помощью ctags

`ctags` – это внешняя программа, сканирующая исходный код и генерирующая алфавитный указатель (индекс) по ключевым словам. Первоначально она была встроена в Vim, но начиная с версии Vim 6 стала самостоятельной программой. Эта наследственность все еще достаточно ярко проявляется в тесной интеграции с Vim.

Поддержка программы `ctags` в редакторе Vim дает возможность быстро перемещаться между объявлениями функций и классов в исходном коде проектов. Как это делается, будет рассказано в рецепте 104. Еще одно преимущество такой интеграции проявляется в возможности использовать вывод программы `ctags` с целью создания списка слов для функции автодополнения, как будет показано в разделе «Индексные файлы» в главе 19.

Навигация и автодополнение невозможны, если Vim не будет знать, где искать индексный файл. В рецепте 103 вы узнаете, как настроить Vim на работу с программой `ctags`. Но сначала давайте посмотрим, как установить и использовать `ctags`.

Рецепт 102. Встречайте: ctags

Чтобы воспользоваться средствами навигации, встроенными в редактор Vim, необходимо сначала установить программу `ctags`. Затем в этом рецепте я расскажу, как запустить эту программу и как использовать индексный файл, сгенерированный ею.

Установка exuberant ctags

Пользователи Linux могут загрузить и установить программу с помощью диспетчера пакетов. Например, установить эту программу в Ubuntu можно следующей командой:

```
⇒ $ sudo apt-get install exuberant-ctags
```

В составе OS X уже имеется предустановленная версия программы с именем `ctags`. Будьте внимательны: это не то же самое, что Exuberant Ctags. Вам придется установить Exuberant Ctags самостоятельно. Сделать это можно с помощью команды:

```
⇒ $ brew install ctags
```

После установки убедитесь, что программа установлена и находится в пути поиска выполняемых файлов, вызвав команду:

```
⇒ $ ctags --version
  Exuberant Ctags 5.8, Copyright (C) 1996-2009 Darren Hiebert
  Compiled: Dec 18 2010, 22:44:26
  ...
```

Если вы не увидели это сообщение, возможно, вам следует изменить переменную `$PATH`. Убедитесь, что путь к каталогу */usr/local/bin* в ней предшествует пути к каталогу */usr/bin*.

Индексирование исходного кода проекта с помощью ctags

Программу `ctags` можно запустить из командной строки, указав ей путь к одному или нескольким файлам, для которых требуется создать индексный файл. В состав загружаемого пакета с примерами для этой книги входит небольшая демонстрационная программа, состоящая из трех файлов с исходным кодом на языке Ruby. Давайте попробуем проиндексировать их с помощью `ctags`:

```
⇒ $ cd code/ctags
⇒ $ ls
  anglophone.rb francophone.rb speaker.rb
⇒ $ ctags *.rb
⇒ $ ls
  anglophone.rb francophone.rb speaker.rb tags
```

Обратите внимание, что программа `ctags` создала простой текстовый файл с именем *tags*. Он содержит алфавитный указатель ключевых слов из трех исходных файлов, проанализированных с помощью `ctags`.

Анатомия индексного файла

Давайте заглянем внутрь только что созданного файла `tags`. Отметьте, что некоторые строки были усечены, чтобы уместить их по ширине страницы:

ctags/tags-abridged
http://media.pragprog.com/titles/dnvim/code/ctags/tags-abridged

```
!_TAG_FILE_FORMAT       2     /extended format/
!_TAG_FILE_SORTED       1     /0=unsorted, 1=sorted, 2=foldcase/
!_TAG_PROGRAM_AUTHOR    Darren Hiebert  //
!_TAG_PROGRAM_NAME      Exuberant Ctags //
!_TAG_PROGRAM_URL       http://ctags.sourceforge.net  /official site/
!_TAG_PROGRAM_VERSION   5.8   //
Anglophone    anglophone.rb   /^class Anglophone < Speaker$/;"     c
Francophone   francophone.rb  /^class Francophone < Speaker$/;"    c
Speaker speaker.rb      /^class Speaker$/;"   c
initialize    speaker.rb      /^  def initialize(name)$/;"   f
speak   anglophone.rb   /^  def speak$/;"    f   class:Anglophone
speak   francophone.rb  /^  def speak$/;"    f   class:Francophone
speak   speaker.rb      /^  def speak$/;"    f   class:Speaker
```

Индексный файл начинается несколькими строками с метаданными. Вслед за ними следует список ключевых слов, по одному в каждой строке. Каждый элемент списка содержит ключевое слово, а также имя файла и адрес в файле, где это ключевое слово встречается. Ключевые слова отсортированы в алфавитном порядке, поэтому Vim (или любой другой текстовый редактор) может быстро отыскивать их в файле, используя поиск методом дихотомии.

Ключевые слова адресуются шаблонами, а не номерами строк

Спецификация формата индексного файла определяет, что роль адреса может играть любая команда Ex[1]. В этом качестве вполне можно было бы использовать абсолютные номера строк. Например, к строке с номером 42 можно перейти с помощью команды Ex `:42`. А теперь подумайте, насколько «хрупким» получится такой файл индексов. Добавив всего одну строку в начало файла с исходными текстами, мы разрушим весь индекс.

Поэтому для адресации ключевых слов `ctags` использует команду поиска (если вы сомневаетесь, что команда поиска является командой Ex, попробуйте ввести `:/pattern`). Такой способ более устойчив, но все еще далек от идеала. А что, если команда поиска будет обнаруживать несколько совпадений в файле с исходным кодом?

Такая ситуация не должна возникать на практике, потому что в шаблон включается столько строк кода, сколько необходимо,

[1] http://ctags.sourceforge.net/FORMAT

чтобы обеспечить уникальность адреса. Пока длина строки не превышает 512 символов, индексный файл будет обратно совместим с редактором `vi`. Конечно, с увеличением длины шаблона поиска увеличивается и «хрупкость» индексного файла.

Индексирование ключевых слов с помощью метаданных

Классический формат индексного файла требует наличия лишь трех полей, разделенных символами табуляции: ключевое слово, имя файла и адрес. Однако в настоящее время используется расширенный формат индексных файлов, включающий дополнительные поля в конце записей, где хранятся метаданные, описывающие ключевые слова. В данном примере можно видеть, что ключевые слова `Anglophone`, `Francophone` и `Speaker` помечены символом `c` как имена классов, а ключевые слова `initialize` и `speak` помечены символом `f` как имена функций.

Расширение ctags или использование совместимых генераторов тегов

Программа ctags может дополняться поддержкой языков, которые не поддерживаются по умолчанию. Используя параметры `--regex`, `--langdef` и `--langmap`, можно определить регулярные выражения и создать простые правила индексирования ключевых конструкций на любом языке. Кроме того, можно опуститься уровнем ниже и написать парсер на C. Парсеры, написанные на C, обычно действуют быстрее, чем парсеры, определенные в виде регулярных выражений, поэтому, если вам приходится работать с большими проектами, это отличие может оказаться очень важным.

Еще одна возможность заключается в создании специализированного инструмента индексирования исходного кода на выбранном языке вместо расширения `ctags`. Например, `gotags` – это `ctags`-совместимый генератор тегов для Go, сам реализованный на Go[1]. Он производит вывод в том же формате, что и программа `ctags`, поэтому без осложнений может использоваться с редактором Vim.

Формат файлов тегов открыт – это простые текстовые файлы. Любой желающий может написать сценарий, генерирующий файлы с тегами, понятные редактору Vim.

---

[1] https://github.com/jstemmer/gotags

Рецепт 103. Настройка Vim для работы с программой ctags

Чтобы использовать команды навигации `ctag` в Vim, необходимо иметь актуальный индексный файл и сообщить редактору, где его искать.

Настройка поиска индексного файла в Vim

Путь к индексному файлу определяется с помощью параметра настройки `tags` (`:h 'tags'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'tags'). Если в параметр `tags` указать путь ./, Vim заменит его строкой пути к текущему активному файлу. Значения по умолчанию можно вывести, как показано ниже:

```
⇒ :set tags?
  tags=./tags,tags
```

С этими настройками Vim будет искать файл *tags* в каталоге с текущим файлом и в текущем рабочем каталоге. В данном конкретном случае, если совпадение будет найдено в первом файле *tags*, Vim не будет просматривать содержимое второго файла (см. `:h tags-option` ⓘ http://vimdoc.sourceforge.net/htmldoc/tagsrch.html#tags-option). При использовании настроек по умолчанию мы можем сохранить файлы *tags* в каждом подкаталоге проекта или создать один общий файл *tags* в корневом каталоге проекта.

Если запускать программу `ctags` достаточно часто, для поддержки актуальности индексного файла (или файлов), он может загружаться при каждой попытке получить исходный код из репозитория. Чтобы избежать этого, сообщите своей системе контроля версий, что она должна игнорировать файлы *tags*.

Создание индексного файла

Как мы видели в разделе «Индексирование исходного кода проекта с помощью ctags» выше, программу `ctags` можно запускать из командной строки. Но нам совсем не нужно покидать Vim, чтобы обновить файл *tags*.

Простейший случай: запуск ctags вручную

Программу `ctags` можно запустить непосредственно из Vim, выполнив команду:

```
⇛ :!ctags -R
```

Начиная с текущего рабочего каталога, эта команда рекурсивно обойдет все подкаталоги и проиндексирует каждый файл. Получившийся в результате файл *tags* будет сохранен в текущем рабочем каталоге.

Если в команду необходимо добавить такой ключ, как `--exclude=.git` или `--languages=-sql`, можно сэкономить на нажатиях клавиш, создав соответствующие привязки к клавишам:

```
⇛ :nnoremap <f5> :!ctags -R<CR>
```

Данная привязка позволит перестроить индекс простым нажатием на клавишу `<F5>`, но при этом мы все еще должны помнить о необходимости периодически обновлять файл *tags*. Поэтому давайте познакомимся с парой возможностей автоматизировать этот процесс.

Автоматический запуск ctags при сохранении файлов

Механизм автоматического запуска команд в редакторе Vim позволяет вызывать команды по событиям, таким как создание буфера, открытие или сохранение файла. Мы могли бы настроить автоматический запуск `ctags` при каждом сохранении файла:

```
⇛ :autocmd BufWritePost * call system("ctags -R")
```

и производить индексирование исходного кода проекта при сохранении любого файла.

Автоматический запуск ctags из обработчиков системы управления версиями

Большинство систем управления версиями поддерживает возможность запуска сценариев в ответ на события, происходящие в репозитории. Эту возможность можно использовать для индексирования файлов после каждой операции отправки исходных кодов в репозиторий.

В своей статье «Effortless Ctags with Git» («Интеграция ctags с Git без усилий») Тип Поуп (Tim Pope) демонстрирует, как организовать выполнение сценариев по событиям `post-commit`, `post-merge` и `post-checkout`[1]. Вся прелесть этого решения – в том, что

---

[1] http://tbaggery.com/2011/08/08/effortless-ctags-with-git.html

оно использует глобальные события, поэтому вам не придется настраивать каждый репозиторий в отдельности.

Обсуждение

Любая стратегия индексирования исходных кодов имеет свои достоинства и недостатки. Индексирование вручную проще, но требует помнить о необходимости обновления индексов, чтобы поддерживать их в актуальном состоянии.

Использование решения на основе автоматического вызова ctags при сохранении каждого буфера гарантирует постоянную актуальность индексного файла, но какой ценой? Для небольшого проекта время выполнения программы ctags может быть совсем невелико, но в больших проектах такой подход может вызывать значительные задержки в рабочем процессе. Кроме того, данное решение не учитывает возможности изменения исходных кодов вне редактора.

Прием повторного индексирования при каждой отправке в репозиторий выглядит более сбалансированным. Конечно, в процессе правки рабочей копии файл *tags* может утрачивать актуальность, но это вполне терпимо. Код, над которым мы активно работаем, едва ли будет тем кодом, для навигации по которому хотелось бы использовать индексы. И не забывайте, что ключевые слова в файле *tags* адресуются командами поиска (см. раздел «Анатомия индексного файла» выше), что повышает устойчивость индексов к изменениям в исходных текстах.

Рецепт 104. Навигация по определениям ключевых слов

Интеграция редактора Vim с программой ctags превращает ключевые слова в своеобразные гиперссылки, давая возможность быстро переходить к определениям. В этом рецепте вы узнаете, как пользоваться командами <C-]> и g<C-]> командного режима и аналогичными им командами Ex.

Переход к определению ключевого слова

Нажатие комбинации <C-]> вызывает переход к определению ключевого слова под курсором. Например:

Нажатия клавиш	Содержимое буфера

{start}	require './speaker.rb'
	class Anglophone < Speaker
	  def speak
	    puts "Hello, my name is #{@name}"
	  end
	end
	Anglophone.new(‘Jack’).speak

<C-]>	require './speaker.rb'
	class Anglophone < Speaker
	  def speak
	    puts "Hello, my name is #{@name}"
	  end
	end
	Anglophone.new('Jack').speak

В данном случае определение класса `Anglophone` оказалось в том же буфере, но если установить курсор на ключевое слово `Speaker` и вызвать ту же команду еще раз, будет открыт буфер с определением этого класса:

Нажатия клавиш	Содержимое буфера

fS	require './speaker.rb'
	class Anglophone < Speaker
	  def speak
	    puts "Hello, my name is #{@name}"
	  end
	end
	Anglophone.new(‘Jack’).speak

<C-]>	class Speaker
	  def initialize(name)
	    @name = name
	  end
	  def speak
	    puts "#{name}"
	  end
	end

При навигации по исходным кодам таким способом Vim запоминает историю посещавшихся индексов. Команда `<C-t>` действует подобно кнопке **Back** (Назад) браузера, возвращая нас назад на один шаг. Если нажать ее прямо сейчас, редактор вернется к определению `Anglophone`, а если нажать ее еще раз, он вернется в то место, откуда мы начали. За дополнительной информацией по приемам работы со списком переходов по индексам обращайтесь к разделу справки `:h tag-stack` ⓘ http://vimdoc.sourceforge.net/htmldoc/tagsrch.html#tag-stack.

Определение точки перехода при наличии нескольких совпадений с ключевым словом

В предыдущем примере мы не испытывали никаких сложностей, потому что исходный код демонстрационного проекта содержал только по одному определению классов `Speaker` и `Anglophone`. Но представьте, что курсор находится в точке вызова метода `speak`, как показано ниже:

```
Anglophone.new('Jack').speak
```

Метод с таким именем определен во всех трех классах: `Speaker`, `Francophone` и `Anglophone`. Какой из них будет выбран редактором при попытке выполнить команду `<C-]>`? Попробуйте выяснить это сами.

Если искомый индекс присутствует в текущем буфере, он имеет наивысший приоритет. Поэтому в данном случае будет выполнен переход к определению метода `speak` в классе `Anglophone`. Дополнительную информацию о ранжировании меток по приоритетам можно найти в справке `:h tag-priority` ⓘ http://vimdoc.sourceforge.net/htmldoc/tagsrch.html#tag-priority.

Вместо команды `<C-]>` можно также использовать команду `g<C-]>`. Обе команды действуют одинаково, если ключевое слово имеет единственное совпадение. Но когда существует несколько совпадений, команда `g<C-]>` выведет список вариантов на выбор:

```
# pri kind tag               file
1 F  C f    speak             anglophone.rb
              class:Anglophone
              def speak
2 F    f    speak             francophone.rb
              class:Francophone
              def speak
3 F    f    speak             speaker.rb
              class:Speaker
              def speak
Type number and <Enter> (empty cancels):
```

Как явствует из приглашения к вводу («Введите число и нажмите <Enter> (пустая строка – отмена)»), мы можем выбрать точку перехода, введя число и нажав клавишу `<CR>`.

Допустим, что в результате выполнения команды `g<C-]>` мы перешли не к тому определению. Мы можем воспользоваться коман-

дой `:tselect`, чтобы повторно вызвать меню выбора со списком индексов, или командой `:tnext`, чтобы перейти к следующему индексу из списка, минуя вывод меню. Как вы уже, наверное, поняли, существуют родственные ей команды `:tprev`, `:tfirst` и `:tlast`. Предложения по определению горячих комбинаций для этих команд можно найти во врезке «Назначение горячих клавиш для обхода списков в Vim» в главе 6.

Использование команд Ex

Необязательно устанавливать курсор на ключевое слово, чтобы перейти к его определению. С тем же успехом можно пользоваться командами Ex. Например,, команды `:tag {keyword}` и `:tjump {keyword}` действуют подобно командам `<C-]>` и `g<C-]>` соответственно (см. `:h :tag` ⓘ http://vimdoc.sourceforge.net/htmldoc/tagsrch.html#:tag и `:h :tjump` ⓘ http://vimdoc.sourceforge.net/htmldoc/tagsrch.html#:tjump).

Иногда эти команды оказываются удобнее, чем маневрирование курсором по ключевым словам в документе, особенно если учесть, что Vim поддерживает автоматическое дополнение для всех ключевых слов, имеющихся в индексном файле. Например, можно ввести `:tag Fran<Tab>`, и Vim услужливо подставит имя `Francophone` в команду.

Кроме того, эти команды Ex принимают также регулярные выражения в форме `:tag /{pattern}` и `:tjump /{pattern}` (обратите внимание на символ `/` перед `{pattern}`). Например, для обхода всех определений, оканчивающихся на `phone`, можно вызвать такую команду:

```
⇛ :tjump /phone$
# pri kind  tag
1 F C c     Anglophone        anglophone.rb
              class Anglophone < Speaker
2 F   c     Francophone       francophone.rb
              class Francophone < Speaker
Type number and <Enter> (empty cancels):
```

В следующей таблице перечислены команды, которые можно использовать для навигации в исходном коде с использованием индексов:

Команда	Действие
`<C-]>`	Выполняет переход к первому индексу, соответствующему ключевому слову под курсором
`g<C-]>`	Предлагает пользователю выбрать из множества совпадений со словом под курсором. Если имеется только одно совпадение, переход выполняется без обращения к пользователю
`:tag {keyword}`	Выполняет переход к первому индексу, соответствующему `{keyword}`
`:tjump {keyword}`	Предлагает пользователю выбрать из множества совпадений с `{keyword}`. Если имеется только одно совпадение, переход выполняется без обращения к пользователю
`:pop` или `<C-t>`	Выполняет переход по истории использования индексов в обратном направлении
`:tag`	Выполняет переход по истории использования индексов в прямом направлении
`:tnext`	Выполняет переход к следующему соответствующему индексу
`:tprev`	Выполняет переход к следующему предыдущему индексу
`:tfirst`	Выполняет переход к первому соответствующему индексу
`:tlast`	Выполняет переход к последнему соответствующему индексу
`:tselect`	Предлагает пользователю выбрать одно из множества совпадений

Глава 17. Компиляция кода и обзор ошибок с помощью Quickfix List

Список результатов (quickfix list) является одной из основных особенностей Vim, обеспечивающих возможность интеграции с внешними инструментами. В простейшем случае он поддерживает последовательность аннотированных адресов, включающих имя файла, номер строки, номер символа в строке (необязательно) и текст сообщения. Традиционно в этом списке сохраняются сообщения компилятора об ошибках, но в нем также могут сохраняться предупреждения программы проверки синтаксиса, сообщения компоновщика или любого другого инструмента, производящего подобный вывод.

Эту главу мы начнем со знакомства с примером компиляции проекта: выполнения команды `make` во внешней командной оболочке и навигации по сообщениям об ошибках вручную. Затем я представлю вам команду `:make`, и мы посмотрим, как она может упростить нам жизнь за счет встроенной поддержки анализа сообщений компилятора об ошибках и возможности навигации по ним.

В рецепте 106 мы рассмотрим наиболее полезные команды навигации по результатам выполнения команды `:make`. А затем, в рецепте 107, исследуем команду отмены, поддерживаемую списком результатов.

В рецепте 108 мы пройдем по всем этапам настройки Vim, чтобы вызов команды `:make` запускал программу JSLint для проверки файлов JavaScript и генерировал список результатов (quickfix list), пригодный для навигации.

Рецепт 105. Компиляция кода, не покидая Vim

Возможность вызова внешнего компилятора из Vim позволяет нам не покидать редактор. А если компилятор сообщит о каких-либо ошибках, Vim позволит нам быстро переходить к строкам, где они были обнаружены.

Подготовка

Для демонстрации мы будем использовать небольшую программу на языке C. Файл с исходным кодом программы входит в состав загружаемого пакета примеров к книге (за подробностями обращайтесь к разделу «Загружаемые примеры» в предисловии). Прежде чем запустить редактор, перейдите в каталог *code/quickfix/wakeup*:

```
⇒ $ cd code/quickfix/wakeup
```

Чтобы скомпилировать эту программу, у вас должен иметься компилятор `gcc`, но вам совсем необязательно устанавливать компилятор, чтобы опробовать приводимые здесь примеры у себя. Последовательность операций, приводимая здесь, демонстрирует задачу, для решения которой изначально задумывался список результатов (откуда он и получил свое название quickfix list – список быстрых результатов). Как будет показано ниже, он имеет массу других применений.

Компиляция проекта в командной оболочке

Демонстрируемая здесь программа `wakeup` состоит из трех файлов: *Makefile*, *wakeup.c* и *wakeup.h*. Чтобы скомпилировать ее, достаточно в командной оболочке вызвать команду `make`:

```
⇒ $ make
  gcc    -c -o wakeup.o wakeup.c
  wakeup.c:68: error: conflicting types for 'generatePacket'
  wakeup.h:3: error: previous declaration of 'generatePacket' was here
  make: *** [wakeup.o] Error 1
```

Как видите, компилятор нашел пару ошибок. Безусловно, очень полезно иметь подобную информацию в окне терминала, но давайте

попробуем заставить Vim распознавать их, чтобы мы могли быстро исправить их в Vim.

Компиляция проекта в Vim

Вместо того чтобы вызывать команду `make` в командной оболочке, попробуем выполнить компиляцию проекта непосредственно из редактора Vim. Убедитесь, что находитесь в каталоге *code/quickfix/wakeup* и в нем присутствует файл *Makefile*, а затем запустите редактор:

```
⇛ $ pwd; ls
  ~/code/quickfix/wakeup
  Makefile wakeup.c wakeup.h
⇛ $ vim -u NONE -N wakeup.c
```

Теперь в редакторе Vim можно вызвать команду `:make`:

```
⇛ :make
  gcc    -c -o wakeup.o wakeup.c
  wakeup.c:68: error: conflicting types for 'generatePacket'
  wakeup.h:3: error: previous declaration of 'generatePacket' was here
  make: *** [wakeup.o] Error 1
  Press ENTER or type command to continue
```

Мы получили тот же результат, что и выше, за исключением того, что Vim автоматически проанализировал полученный вывод. Редактор не просто вывел результаты работы команды `make` на экран, он проанализировал каждую строку, извлек имена файлов, номера строк и тексты сообщений об ошибках, и для каждой ошибки или предупреждения создал запись в списке результатов. Мы можем перемещаться по этим записям взад и вперед, а Vim будет переходить к строкам в исходном коде, соответствующим этим записям. Согласно документации Vim (`:h quickfix` ⓘ http://vimdoc.sourceforge.net/htmldoc/quickfix.html#quickfix), это дает возможность «ускорить цикл правка – компиляция – правка».

После выполнения команды `:make` Vim переходит к первой записи в списке результатов. В данном случае вы заметите, что он установит курсор в файле *wakeup.c* в начало следующей функции:

```
void generatePacket(uint8_t *mac, uint8_t *packet)
{
    int i, j, k;
    k = 6;
    for (i = 0; i <= 15; i++)
```

```
    {
        for (j = 0; j <= 5; j++, k++)
        {
            packet[k] = mac[j];
        }
    }
}
```

Сообщение об ошибке «conflicting types for 'generatePacket'» сообщает о конфликте типов в функции `generatePacket`. Перейти к следующей записи в списке результатов можно с помощью команды `:cnext`. В данном случае она перенесет нас в файл *wakeup.h*, в строку:

```
void generatePacket(char *, char *);
```

Теперь понятно, на что «жалуется» компилятор: сигнатура функции в заголовочном файле не совпадает с сигнатурой в файле реализации. Давайте изменим строку в заголовочном файле, заменив `char` на `uint8_t`:

```
void generatePacket(uint8_t *, uint8_t *);
```

Сохраним изменения и снова вызовем команду `:make`:

```
⇛ :write
⇛ :make
  gcc -c -o wakeup.o wakeup.c
  gcc -o wakeup wakeup.o
```

На этот раз программа благополучно скомпилировалась. Список с результатами обновился и теперь отражает вывод последнего вызова команды `make`. Редактор не обнаружил здесь сообщений об ошибках и оставил курсор там, где он находился.

Не заблудитесь!

После выполнения команды `:make` Vim автоматически переходит к месту обнаружения первой ошибки (если ошибки действительно были обнаружены). Если вы предпочитаете, чтобы курсор оставался на месте, используйте следующую команду:

```
⇛ $ make!
```

Завершающий символ `!` предписывает редактору обновить список с результатами, но не выполнять переход к первому элементу в нем.

Теперь представьте, что мы запустили `:make` и только после этого заметили, что забыли добавить восклицательный знак. Как вернуться туда, где находился курсор на момент запуска команды `:make`? Все просто: воспользуйтесь командой `<C-o>` перехода в предыдущую позицию в списке переходов. Подробнее об этом списке рассказывается в рецепте 56 в главе 9.

Рецепт 106. Навигация по списку с результатами

Список с результатами хранит коллекцию записей, ссылающихся на один или несколько файлов. Каждая запись может соответствовать ошибке, обнаруженной компилятором в процессе выполнения команды `:make`, или совпадению, найденному командой `:grep`. Не важно, как был получен этот список, важно, что мы получаем возможность навигации по записям в нем. В этом рецепте вы познакомитесь со способами навигации по списку с результатами.

Исчерпывающий перечень команд навигации по списку с результатами можно найти в разделе справки `:h quickfix` ⓘ http://vimdoc.sourceforge.net/htmldoc/quickfix.html#quickfix. Некоторые из наиболее употребимых команд приводятся в табл. 17.1.

Все команды начинаются с префикса `:c`. Список адресов (см. врезку «Встречайте: список адресов» ниже) имеет эквивалентные команды, начинающиеся с префикса `:l`, такие как `:lnext`, `:lprev` и т. д. Команда `:ll N`, выполняющая переход к *n*-й записи в списке адресов, является наглядной демонстрацией следования этому шаблону.

Таблица 17.1. Команды навигации по списку с результатами Quickfix List

Команда	Действие
`:cnext`	Выполняет переход к следующей записи
`:cprev`	Выполняет переход к предыдущей записи
`:cfirst`	Выполняет переход к первой записи
`:clast`	Выполняет переход к последней записи
`:cnfile`	Выполняет переход к первой записи в следующем файле
`:cpfile`	Выполняет переход к последней записи в предыдущем файле
`:cc N`	Выполняет переход к n-й записи
`:copen`	Открывает окно со списком с результатами
`:cclose`	Закрывает окно со списком с результатами

Таблица 17.1. Команды навигации по списку с результатами Quickfix List (окончание)

Команда	Действие
`:cdo {cmd}`	Применяет команду `{cmd}` к каждой строке в списке результатов (quickfix list)
`:cfdo {cmd}`	Применяет команду `{cmd}` один раз к каждому файлу в списке результатов (quickfix list)

Встречайте: список адресов

Для каждой команды, заполняющей список с результатами, имеется аналог, помещающий результаты в список адресов. Если команды `:make`, `:grep` и `:vimgrep` заполняют список с результатами, то команды `:lmake`, `:lgrep` и `:lvimgrep` заполняют список адресов. В чем разница? В каждый конкретный момент список с результатами всегда один, а списков адресов может быть несколько.

Допустим, что мы выполнили все шаги, описываемые в рецепте 108 ниже, настроив команду `:make` на проверку файлов JavaScript с помощью JSLint. А теперь представьте, что в разных окнах у нас открыты два разных файла JavaScript. Мы можем выполнить команду `:lmake`, чтобы проверить содержимое активного окна, и тем самым сохранить сообщения об ошибках в списке адресов. Затем мы можем переключиться в другое окно и снова выполнить команду `:lmake`. В итоге Vim не затрет список адресов, полученный в результате выполнения первой команды, а создаст новый. После этого у нас будут два списка адресов, каждый из которых будет содержать список ошибок для своего файла JavaScript.

Любые команды, взаимодействующие со списками адресов (`:lnext`, `:lprev` и др.), будут оперировать списком, связанным с текущим активным окном. Сравните это с использованием списка с результатами (quickfix list), который является глобальным: не важно, какая вкладка или какое окно активно, команда `:copen` всегда будет отображать один и тот же список.

Основы навигации по списку с результатами

Мы можем перемещаться между записями в списке с результатами с помощью команд `:cnext` и `:cprevious`. Перепрыгнуть в начало или в конец списка можно с помощью команд `:cfirst` и `:clast`. Мы часто будем пользоваться этими командами, поэтому совсем нелишним будет привязать их к горячим комбинациям клавиш, чтобы проще было пользоваться ими. Предлагаемые варианты вы найдете во врезке «Назначение горячих клавиш для обхода списков в Vim» в главе 6.

Быстрые переходы вперед и назад

Обе команды, `:cnext` и `:cprev`, могут снабжаться счетчиками. То есть вместо последовательного перехода от одной записи к другой можно, например, перепрыгнуть сразу через пять записей:

```
⇒ :5cnext
```

Допустим, что мы просматриваем список с результатами и натыкаемся на файл, которому в списке соответствуют десятки записей, ни одна из которых не представляет особого интереса. В этом случае вместо последовательного обхода записей по одной (или даже по десятку) можно просто пропустить все записи, относящиеся к этому файлу, и перейти к первой записи, соответствующей следующему файлу, для чего достаточно вызвать команду `:cnfile`. Как вы уже наверняка догадались, команда `:cpfile` выполняет аналогичное действие, но в обратном направлении – переходит к последней записи, соответствующей предыдущему файлу.

Окно Quickfix

С помощью команды `:copen` можно открыть окно с содержимым списка с результатами. До определенной степени это окно действует так же, как обычный буфер. Его можно прокручивать клавишами `k` и `j`, в нем можно даже выполнять обычные операции поиска.

Однако окно списка с результатами имеет свои особенности. Если установить курсор на какую-либо запись и нажать клавишу `<CR>`, будет открыт файл, соответствующий этой записи. Обычно файл открывается в окне, находящемся непосредственно над окном **Quickfix**, но если файл уже был открыт в некотором окне, в текущей вкладке, будет использован открытый буфер.

Имейте в виду, что каждая строка в данном окне соответствует записи в списке с результатами. Если выполнить команду `:cnext`, курсор переместится вниз на одну строку, даже если окно **Quickfix** не было активным. Напротив, если переход осуществлялся посредством выбора строки в окне **Quickfix**, следующая команда `:cnext` выполнит переход к строке, следующей за выбранной. Выбор строки в окне **Quickfix** близко напоминает команду `:cc [nr]`, но обеспечивает более понятный визуальный интерфейс.

Закрыть окно **Quickfix**, когда оно активно, можно привычной командой `:q`. А находясь в другом окне, закрыть его можно командой `:cclose`.

Рецепт 107. Восстановление прежнего списка с результатами

Обновляя список с результатами, Vim не затирает предыдущего его содержимого, а сохраняет в более старых списках, позволяя восстанавливать его.

Прежнюю версию списка с результатами (Vim хранит последние десять списков) можно восстановить командой `:colder` (к сожалению, в редакторе отсутствует команда `:warmer`[1]). Вернуться обратно, к более новой версии списка с результатами, можно командой `:cnewer`. Обратите внимание, что обе команды, `:colder` и `:cnewer`, могут принимать счетчик, обеспечивающий выполнение указанной команды заданное число раз.

Если после вызова команды `:colder` или `:cnewer` открыть окно **Quickfix**, строка состояния в нем будет содержать команду, сгенерировавшую этот список.

Команды `:colder` и `:cnewer` можно рассматривать как своеобразные команды отмены и повторения ввода (undo и redo) для списка с результатами. Это означает, что вы без опаски можете выполнять команды, заполняющие список с результатами, потому что у вас всегда сохраняется возможность вызвать `:colder` и вернуться к предыдущему списку. Кроме того, вместо повторного выполнения команды `:make` или `:grep` можно просто вернуться к уже имеющимся результатам (конечно, если с того момента никакие файлы не изменялись). Это поможет сэкономить массу времени, особенно если на выполнение таких команд требуется достаточно много времени.

Рецепт 108. Настройка внешнего компилятора

Команда `:make` может вызывать не только внешнюю программу `make`; она способна вызвать любой компилятор, имеющийся в системе. (Обратите внимание, что Vim интерпретирует понятие «компилятор» более широко, чем вы, возможно, привыкли; см. врезку «Команды :compiler и :make служат не только для разработки программ на компилируемых языках» ниже.) В этом рецепте вы узнаете, как настроить команду `:make`, чтобы она выполняла проверку

[1] Шутка. Здесь автор обыгрывает слова «colder» – холоднее и «warmer» – теплее. – *Прим. перев.*

файлов JavaScript с помощью JSLint и сохраняла вывод в списке с результатами.

Для начала настроим Vim, чтобы команда `:make` вызывала программу `nodelint`[1], командный интерфейс JSLint[2]. Для этого необходимо установить Node.js с помощью диспетчера пакетов NPM, просто выполнив следующую команду[3]:

```
⇒ $ npm install nodelint -g
```

Для демонстрации мы будем использовать следующий файл JavaScript:

quickfix/fizzbuzz.js
http://media.pragprog.com/titles/dnvim/code/quickfix/fizzbuzz.js

```
var i;
for (i=1; i <= 100; i++) {
    if(i % 15 == 0) {
        console.log('Fizzbuzz');
    } else if(i % 5 == 0) {
        console.log('Buzz');
    } else if(i % 3 == 0) {
        console.log('Fizz');
    } else {
        console.log(i);
    }
};
```

Настройка вызова программы Nodelint командой :make

Определить программу, которую должна вызывать команда `:make`, можно с помощью параметра настройки `makeprg` (см. `:h 'makeprg'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'makeprg'), как показано ниже:

```
⇒ :setlocal makeprg=NODE_DISABLE_COLORS=1\ nodelint\ %
```

На место символа `%` будет поставлен путь к текущему файлу. То есть, если редактируется файл *~/quickfix/fizzbuzz.js*, вызов команды `:make` в редакторе Vim будет эквивалентен вызову следующей команды в оболочке:

---

[1] https://github.com/tav/nodelint

[2] http://jslint.com/

[3] http://nodejs.org/ и http://npmjs.org/ соответственно.

```
⇛ $ export NODE_DISABLE_COLORS=1
⇛ $ nodelint ~/quickfix/fizzbuzz.js
  ~/quickfix/fizzbuzz.js, line 2, character 22: Unexpected '++'.
  for (i=1; i <= 100; i++) {
  ~/quickfix/fizzbuzz.js, line 3, character 15: Expected '===' ...
  if(i % 15 == 0) {
  ~/quickfix/fizzbuzz.js, line 5, character 21: Expected '===' ...
  } else if(i % 5 == 0) {
  ~/quickfix/fizzbuzz.js, line 7, character 21: Expected '===' ...
  } else if(i % 3 == 0) {
  ~/quickfix/fizzbuzz.js, line 12, character 2: Unexpected ';'.
  };
  5 errors
```

По умолчанию `nodelint` подсвечивает ошибки красным цветом, используя ANSI-коды цвета. Настройка `NODE_DISABLE_COLORS=1` отключает выделение цветом, что упрощает поиск ошибок в выводе.

Далее нужно настроить Vim, чтобы он смог разобрать вывод программы `nodelint` и на его основе построить список результатов. Решить эту проблему можно двумя способами: настроить программу `nodelint`, чтобы выводимая ею информация напоминала вывод программы `make`, которую Vim уже понимает, или научить Vim распознавать вывод программы `nodelint` с настройками по умолчанию. Поскольку эта книга посвящена редактору Vim, реализуем второе решение.

Заполнение списка с результатами на основе вывода Nodelint

Научить Vim распознавать вывод, генерируемый командой `:make`, можно с помощью параметра настройки `errorformat` (см. `:h 'errorformat'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'errorformat'). Значение по умолчанию этого параметра можно получить следующей командой:

```
⇛ :setglobal errorformat?
  errorformat=%*[^"]"%f"%*\D%l: %m,"%f"%*\D%l: %m, ...[abridged]...
```

Знакомые с функцией `scanf` (в языке C) сразу поймут основную идею. Каждый символ с предшествующим знаком процента имеет специальное значение: `%f` соответствует имени файла, `%l` – номеру строки, а `%m` – тексту сообщения об ошибке. Полный список специальных символов можно найти в разделе справки `:h errorformat` ⓘ http://vimdoc.sourceforge.net/htmldoc/quickfix.html#errorformat.

Для анализа вывода программы `nodelint` можно использовать следующую строку формата:

```
⇒ :setlocal efm=%A%f\,\ line\ %l\,\ character\ %c:%m,%Z%.%#,%-G%.%#
```

Теперь при вызове команды `:make` Vim будет использовать эту строку формата для анализа вывода программы `nodelint`. Для каждого предупреждения он будет извлекать имя файла, номер строки и номер позиции в строке и на их основе генерировать адрес для записи в списке в результатами (quickfix list). Благодаря этому мы сможем перемещаться между предупреждениями с помощью команд, обсуждавшихся в рецепте 106 выше.

Настройка makeprg и errorformat единственной командой

Строка для параметра настройки `errorformat` достаточно сложна, чтобы удерживать ее в памяти. Однако ее можно сохранить в файл и затем активировать командой `:compiler`, которая является более краткой формой настройки параметров `makeprg` и `errorformat` (см. `:h :compiler` ⓘ http://vimdoc.sourceforge.net/htmldoc/quickfix.html#:compiler):

```
⇒ :compiler nodelint
```

Команда `:compiler` активирует *расширение компилятора* (compiler plugin), которое настраивает параметры `makeprg` и `errorformat` на работу с программой `nodelint`. Она является примерным эквивалентом встраивания следующих строк в файл с настройками:

quickfix/ftplugin.javascript.vim
http://media.pragprog.com/titles/dnvim/code/quickfix/ftplugin.javascript.vim

```
setlocal makeprg=NODE_DISABLE_COLORS=1\ nodelint\ %

let &l:efm='%A'
let &l:efm.='%f\, '
let &l:efm.='line %l\, '
let &l:efm.='character %c:'
let &l:efm.='%m' . ','
let &l:efm.='%Z%.%#' . ','
let &l:efm.='%-G%.%#'
```

Внутреннее устройство расширения компилятора намного сложнее, но данная аппроксимация достаточно близко описывает его.

Ознакомиться со списком имеющихся расширений компиляторов можно с помощью команды

```
⇒ :args $VIMRUNTIME/compiler/*.vim
```

Обратите внимание, что Vim-расширение компилятора для поддержки `nodelint` не входит в состав стандартного дистрибутива Vim, но его легко можно установить[1]. Если предполагается постоянное использование программы `nodelint` в качестве «компилятора» для файлов JavaScript, можно воспользоваться расширением автоматического выполнения команд или расширением настройки окружения в зависимости от типа файла. Чтобы узнать, как это делается, загляните в раздел «Применение настроек для определенных типов файлов» в приложении А.

Команды :compiler и :make служат не только для разработки программ на компилируемых языках

Слова «make» (сборка) и «compile» (компиляция) имеют определенное значение в контексте компилирующих языков программирования. Но в редакторе Vim соответствующие команды `:make` и `:compile` имеют более широкую интерпретацию, что позволяет с успехом применять их к файлам с кодом на интерпретирующих языках и языках разметки.

Например, при работе с документами LaTeX можно настроить Vim так, что команда `:make` будет компилировать файлы *.tex* в формат PDF. Или при разработке программ на интерпретирующих языках, таких как JavaScript, можно настроить команду `:make` на выполнение проверки синтаксиса исходного кода с помощью JSLint или какой-либо другой программы. Как вариант командой `:make` можно было бы запускать тестирование.

В терминологии Vim компилятором является любая внешняя программа, выполняющая некоторые операции с документом и производящая список ошибок или предупреждений. Команда `:make` просто вызывает внешний компилятор и затем анализирует его вывод с целью создать список с результатами для навигации по ним.

[1] https://github.com/bigfish/vim-nodelint

Глава 18. Поиск в пределах проекта с помощью команд grep, vimgrep и других

Команды поиска в редакторе Vim отлично подходят для поиска всех вхождений шаблона в пределах файла. Но как быть, если необходимо выполнить поиск совпадений по всему проекту? Для этого придется просмотреть множество файлов. Традиционно эта задача решается с помощью специализированного инструмента `grep`.

В этой главе мы исследуем команду Vim `:grep`, дающую возможность вызвать внешнюю команду, не покидая редактора. Хотя эта команда и носит имя `grep` (если она доступна), ее легко можно адаптировать для запуска других внешних программ, таких как `ack`.

Один из недостатков применения внешних программ состоит в том, что синтаксис регулярных выражений, реализованных в них, может оказаться несовместимым с синтаксисом регулярных выражений, поддерживаемым редактором Vim. Мы увидим, что команда `:vimgrep` позволяет использовать встроенный механизм Vim для поиска совпадений с шаблонами в нескольких файлах. Однако это удобство имеет свою цену: `:vimgrep` действует не так быстро, как внешние программы.

Рецепт 109. Вызов grep, не покидая Vim

Команда `:grep` редактора Vim действует как обертка к внешней программе `grep` (или любой ей подобной). С помощью этой обертки можно вызвать программу `grep` для поиска по шаблону во множестве файлов, не покидая Vim, а затем выполнить обход результатов с применением списка результатов.

Сначала мы рассмотрим ситуацию, когда `grep` и Vim действуют независимо, никак не взаимодействуя друг с другом. Мы исследуем слабые стороны этого подхода, а затем перейдем к изучению интегрированного решения, лишенного подобных проблем.

Использование grep из командной строки

Допустим, что в процессе работы в Vim нам потребовалось отыскать каждое вхождение слова «Waldo» во всех файлах в текущем каталоге. Оставляя Vim, мы запускаем следующую команду:

```
⇒ $ grep -n Waldo *
  department-store.txt:1:Waldo is beside the boot counter.
  goldrush.txt:6:Waldo is studying his clipboard.
  goldrush.txt:9:The penny farthing is 10 paces ahead of Waldo.
```

По умолчанию программа `grep` выводит по одной строке на каждое найденное совпадение, отображая содержимое строки с совпадением и имя файла. Флаг `-n` предписывает программе `grep` включать номера строк в вывод.

Итак, что мы можем сделать с этим выводом? Скажем так: мы могли бы интерпретировать его как оглавление. Опираясь на информацию в любой строке, мы могли бы открыть соответствующий файл и перейти к строке с указанным номером. Например, чтобы открыть файл *goldrush.txt* и перейти к строке 9, мы могли бы выполнить следующую команду оболочки:

```
⇒ $ vim goldrush.txt +9
```

Разумеется, имеющиеся у нас инструменты способны поддерживать более тесную интеграцию.

Вызов grep из редактора Vim

В редакторе Vim имеется команда `:grep`, являющаяся оберткой для внешней программы `grep` (см. `:h :grep` ⓘ http://vimdoc.sourceforge.net/htmldoc/quickfix.html#:grep). Вместо вызова команды `grep` из командной оболочки мы можем выполнить ее, не покидая редактора:

```
⇒ :grep Waldo *
```

За кулисами Vim выполнит команду `grep -n Waldo *`. Но вместо вывода результатов работы `grep` Vim проанализирует их и сконстру-

ирует список результатов (quickfix list). В результате мы получим возможность навигации по списку с помощью команд `:cnext`/`:cprev` и применения всех остальных приемов, описанных в главе главе 17 «Компиляция кода и обзор ошибок с помощью Quickfix List».

Даже при том, что мы просто вызвали `:grep Waldo *`, Vim автоматически добавит флаг `-n`, чтобы обеспечить включение номеров строк в вывод. Именно благодаря этому, осуществляя навигацию по списку с результатами, редактор оказывается в состоянии переместить курсор непосредственно в строку с совпадением.

Допустим, что нам нужно выполнить поиск без учета регистра символов. Для этого следует передать команде флаг `-i`:

```
⇒ :grep -i Waldo *
```

За кулисами Vim выполнит команду `grep -n -i Waldo *`. Обратите внимание, что флаг по умолчанию `-n` все еще присутствует в команде. Таким же способом мы можем передавать программе `grep` любые другие флаги, чтобы оказать влияние на ее поведение.

Рецепт 110. Настройка программы grep

Команда `:grep` – это всего лишь обертка для внешней программы `grep`. Мы можем настроить ее так, что Vim будет делегировать решение задачи поиска любому другому инструменту, устанавливая значения двух параметров настройки: `grepprg` и `grepformat`. Сначала мы познакомимся со значениями по умолчанию этих параметров, а потом посмотрим, как можно настроить их так, чтобы обеспечить решение задачи с помощью любой другой подходящей внешней программы.

Настройки по умолчанию команды :grep

Параметр настройки `grepprg` определяет команду оболочки, которую должна запускать команда Vim `:grep` (`:h 'grepprg'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'grepprg'). Параметр настройки `grepformat` обеспечивает анализ вывода, возвращаемого командой `:grep` (см. `:h 'grepformat'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'grepformat'). В системах Unix эти параметры имеют следующие значения по умолчанию:

```
grepprg="grep -n $* /dev/null"
grepformat="%f:%l:%m,%f:%l%m,%f  %l%m"
```

Символ `$*` в значении параметра `grepprg` является меткой-заполнителем, замещаемой любыми аргументами, переданными команде `:grep`.

Значением параметра `grepformat` является строка, содержащая спецификаторы формата, которые описывают вывод, возвращаемый командой `:grep`. В строке `grepformat` используются те же спецификаторы формата, что и в параметре `errorformat`, с которым мы познакомились в разделе «Заполнение списка с результатами на основе вывода Nodelint» в главе 17. Полный перечень спецификаторов можно найти в разделе справки `:h errorformat` ⓘ http://vimdoc.sourceforge.net/htmldoc/quickfix.html#errorformat.

Давайте посмотрим, как формат по умолчанию `%f:%l %m` обеспечивает анализ вывода команды `grep`:

```
department-store.txt:1:Waldo is beside the boot counter.
goldrush.txt:6:Waldo is studying his clipboard.
goldrush.txt:9:The penny farthing is 10 paces ahead of Waldo.
```

Для каждой строки спецификатор `%f` извлекает имя файла (*department-store.txt* или *goldrush.txt*), `%l` извлекает номер строки, а `%m` – оставшийся текст.

Строка `grepformat` может содержать множество спецификаторов формата, разделенных запятыми. По умолчанию вывод команды `grep` соответствует либо набору спецификаторов `%f:%l %m`, либо `%f %l%m`. Vim будет использовать первый набор, подошедший для анализа вывода `:grep`.

Настройка :grep на вызов команды ack

Программа `ack` – это альтернатива программе `grep`, которая предназначена в первую очередь для использования программистами. Если вам интересно ознакомиться со сравнительными характеристиками программ `ack` и `grep`, обязательно посетите страницу: http://betterthangrep.com.

Прежде всего необходимо установить программу `ack`. В Ubuntu это можно сделать с помощью следующей команды:

```
⇒ $ sudo apt-get install ack-grep
⇒ $ sudo ln -s /usr/bin/ack-grep /usr/local/bin/ack
```

Первая из этих команд установит программу, что даст нам возможность вызывать ее как `ack-grep`. А вторая создаст символическую ссылку на нее с именем `ack`.

В OS X программу `ack` можно установить с помощью Homebrew:

```
⇒ $ brew install ack
```

Давайте посмотрим, как настроить параметры `grepprg` и `grepformat`, чтобы команда `:grep` вызывала `ack` вместо `grep`. По умолчанию программа `ack` выводит имя файла в отдельной строке, за которой следует строка с номером и содержимым строки в файле, содержащей искомое совпадение:

```
⇒ $ ack Waldo *
  department-store.txt
  1:Waldo is beside the boot counter.

  goldrush.txt
  6:Waldo is studying his clipboard.
  9:The penny farthing is 10 paces ahead of Waldo.
```

Мы легко можем организовать вывод результатов в формате, напоминающем формат вывода команды `grep -n`, вызвав команду `ack` с ключом `--nogroup`:

```
⇒ $ ack --nogroup Waldo *
  department-store.txt:1:Waldo is beside the boot counter.
  goldrush.txt:6:Waldo is studying his clipboard.
  goldrush.txt:9:The penny farthing is 10 paces ahead of Waldo.
```

Этот формат вывода в точности соответствует формату вывода команды `grep -n`, а так как значение по умолчанию параметра `grepformat` уже прекрасно справляется с анализом этого формата, нам не требуется изменять его. То есть нам остается всего лишь подменить вызов команды `grep` командой `ack`, установив новое значение в параметре `grepprg`:

```
⇒ :set grepprg=ack\ --nogroup\ $*
```

Альтернативные расширения для поиска в файлах

Редактор Vim упрощает поиск совпадений во множестве файлов с применением внешней программы. Нам нужно только изменить параметры `grepprg` и `grepformat`, и можно начинать пользоваться командой `:grep`, которая точно так же будет сохранять вывод программы в списке результатов. Не важно, какая программа будет вызываться в действительности, интерфейс доступа к результатам остается неизменным.

Однако есть ряд важных особенностей, которые требуется учитывать. Программа `grep` использует регулярные выражения POSIX, тогда как `ack` – регулярные выражения Perl. Если настроить использование программы `ack` командой `:grep`, увеличивается вероятность допустить ошибку при составлении регулярных выражений. Может, лучше было бы создать собственную команду `:Ack`, имя которой не вводило бы в заблуждение?

Именно этой стратегии следует расширение Ack.vim, а также расширение fugitive.vim[1], которое добавляет собственную команду `:Ggrep`, вызывающую `git-grep`. Мы можем установить несколько расширений, подобных этим, а так как каждый из них создает отдельную команду, не переопределяя команду `:grep`, они прекрасно могут сосуществовать, не конфликтуя друг с другом. Мы не ограничены какой-то одной `grep`-подобной программой и можем использовать любые другие, которые наиболее удобны для нас.

Переход к строке и позиции в строке при использовании ack

Но программа `ack` имеет еще одну особенность. Когда она вызывается с ключом `--column`, `ack` выводит не только номер строки с совпадением, но *и номер позиции в строке*. Взгляните:

```
⇒ $ ack --nogroup --column Waldo *
  department-store.txt:1:1:Waldo is beside the boot counter.
  goldrush.txt:6:1:Waldo is studying his clipboard.
  goldrush.txt:9:41:The penny farthing is 10 paces ahead of Waldo.
```

Если изменить значение параметра `grepformat` так, что включится извлечение дополнительной информации, при навигации по списку результатов мы сможем перемещаться непосредственно к найденному совпадению. Этого можно добиться при следующих настройках:

```
⇒ :set grepprg=ack\ --nogroup\ --column\ $*
⇒ :set grepformat=%f:%l:%c:%m
```

Спецификатор `%c` соответствует позиции в строке.

1 https://github.com/mileszs/ack.vim и https://github.com/tpope/vim-fugitive соответственно.

Рецепт 111. Поиск с использованием внутреннего механизма поиска Vim

Команда `:vimgrep` дает возможность осуществлять поиск по множеству файлов с использованием механизма регулярных выражений, встроенного в редактор Vim.

Для демонстрации мы будем использовать файлы в каталоге *grep/quotes*, который можно найти в пакете загружаемых примеров к книге. Каталог содержит следующие файлы (далее приводится также их содержимое):

quotes/about.txt
```
Don't watch the clock; do what it does. Keep going.
```

tough.txt
```
When the going gets tough, the tough get going.
```

where.txt
```
If you don't know where you are going,
you might wind up someplace else.
```

Каждый из этих файлов содержит, по меньшей мере, одно вхождение слова «going». Мы можем потребовать от Vim выполнить поиск этого слова в каждом из перечисленных файлов, вызвав команду `:vimgrep`:

```
⇒ :vimgrep /going/ clock.txt tough.txt where.txt
  (1 of 3): Don't watch the clock; do what it does. Keep going.
⇒ :cnext
  (2 of 3): When the going gets tough, the tough get going.
⇒ :cnext
  (3 of 3): If you don't know where you are going,
```

Команда `:vimgrep` заполнит список результатов (quickfix list), включив в него все строки, содержащие совпадение. Обход результатов можно выполнять с помощью команд `:cnext` и `:cprev` (см. рецепт 106 в главе 17).

Файл *tough.txt* содержит два вхождения слова «going», но команда `:vimgrep` подсчитала только первое совпадение. Если после шаблона добавить в команду флаг `g`, команда `:vimgrep` создаст отдельные записи для всех совпадений с шаблоном в строке, а не только для первых:

```
⇒ :vim /going/g clock.txt tough.txt where.txt
  (1 of 4): Don't watch the clock; do what it does. Keep going.
```

На этот раз в списке результатов содержатся все вхождения слова «going». Таким своим поведением `:vimgrep` напоминает команду `:substitute`: по умолчанию она отыскивает только первые совпадения в строках, но при наличии флага `g` будет отыскивать все. Когда я использую команду `:substitute` или `:vimgrep`, я почти всегда ожидаю от них поведения, которое определяет флаг `g`.

Выбор файлов для поиска

Команда `:vimgrep` имеет следующий синтаксис (`:h :vimgrep` ⓘ http://vimhelp.appspot.com/quickfix.txt.html#%3Avimgrep):

```
:vim[grep][!] /{pattern}/[g][j] {file} ...
```

Аргумент `{file}` не должен оставаться пустым. В нем можно передавать имена файлов, групповые символы, выражения в обратных апострофах и их комбинации. Здесь также можно использовать любые приемы, которые допустимы при заполнении списка аргументов (подробно обсуждалось в разделе «Заполнение списка аргументов» в главе 6).

В предыдущих примерах мы явно указывали имена всех файлов. Однако тот же результат можно получить с применением групповых символов:

```
⇒ :vim /going/g *.txt
  (1 of 4): Don't watch the clock; do what it does. Keep going.
```

Помимо групповых символов * и **, можно также использовать символ ##, представляющий имена файлов в списке аргументов (`:h cmdline-special` ⓘ http://vimhelp.appspot.com/cmdline.txt.html#cmdline-special). Это позволяет несколько иначе организовать поиск. Сначала нужно заполнить список аргументов именами требуемых файлов, а затем применить `:vimgrep` ко всем файлам в этом списке:

```
⇒ :args *.txt
⇒ :vim /going/g ##
  (1 of 4): Don't watch the clock; do what it does. Keep going.
```

На первый взгляд, кажется, что этот подход более трудоемкий, потому что требует выполнить две отдельные Ex-команды. Но я часто предпочитаю использовать `:vimgrep` именно таким способом,

потому что он позволяет получить ответы на два отдельных вопроса: «По каким файлам выполнить поиск?» и «Что искать в этих файлах?». Заполнив список аргументов требуемыми файлами, мы можем использовать его в команде `:vimgrep` столько раз, сколько понадобится.

Поиск в файле с последующим поиском в проекте

Поле шаблона можно оставить пустым и тем самым потребовать от `:vimgrep` использовать текущий шаблон поиска. Тот же трюк можно использовать с командой `:substitute` (как обсуждалось в рецепте 91 в главе 14) и с командой `:global`. Это очень удобно, когда требуется выполнить поиск по регулярному выражению в нескольких файлах. Мы можем сначала сконструировать регулярное выражение и опробовать его на текущем файле, а затем передать этот же шаблон команде `:vimgrep`, чтобы выполнить поиск в нескольких файлах. Например, ниже выполняется поиск в текущем файле по регулярному выражению, совпадающему со словами «don't» и «Don't».

```
⇒ /[Dd]on't
⇒ :vim //g *.txt
  (1 of 2): Don't watch the clock; do what it does. Keep going.
```

Главное достоинство команды `:vimgrep` – в том, что она принимает те же шаблоны, что и команда поиска в Vim. Если вам потребуется использовать `:grep`, чтобы выполнить поиск с тем же шаблоном по всем файлам в проекте, вам придется сначала преобразовать этот шаблон в POSIX-совместимое регулярное выражение. Для таких простых шаблонов, как этот, это не займет много времени, но мало у кого возникнет желание проделать то же самое со сложным регулярным выражением, как то, что было сконструировано в рецепте 85 в главе 13.

История поиска и :vimgrep

Я часто пользуюсь следующей командой, которая ищет совпадения с текущим шаблоном в каждом файле из списка аргументов:

```
⇒ :vim //g ##
  (1 of 2): Don't watch the clock; do what it does. Keep going.
```

Единственное, о чем следует помнить, работая с этой командой, – она всегда использует текущие значения из списка аргументов и истории поиска. Если эту команду выполнить повторно спустя некоторое время, ее результаты могут отличаться от ожидаемых в зависимости от содержимого списка аргументов и истории поиска.

Как вариант можно вручную заполнять поле поиска текущим шаблоном, нажимая `<C-r>/`. В этом случае поиск даст те же результаты, что и прежде, но история команд изменится.

```
⇒ :vim /<C-r>//g ##
```

Если вы думаете, что та же команда `:vimgrep` может понадобиться позднее, тогда сохраните этот шаблон в истории команд.

Глава 19. Набери X и пользуйся автодополнением

Функция автодополнения позволяет сэкономить на вводе целых слов. После ввода нескольких первых символов редактор Vim создает список возможных вариантов продолжения и дает возможность выбрать один из них. Использовать этот список или нет, зависит от того, насколько уверенно вы будете чувствовать себя при взаимодействии с этим списком. В рецепте 113 вы познакомитесь с некоторыми особенностями работы с меню выбора вариантов автодополнения.

В рецепте 112 будут представлены основные комбинации клавиш доступа к функции автодополнения. Список автодополнения создается сканированием всех файлов, которые открывались в текущем сеансе работы с редактором, а также всех подключаемых и индексных файлов. Подробнее об этом рассказывается в рецепте 114, где также будет показано, как уменьшить список подсказок.

Кроме поиска ключевых слов, Vim имеет другие способы создания списка вариантов автодополнения. В табл. 19.1 перечислены некоторые из них, описываемые в рецептах из этой главы.

Чтобы извлечь максимум из функции автодополнения в редакторе Vim, необходимо понимать две вещи: как организовать запоминание наиболее подходящих вариантов и как выбрать правильное слово из списка. Обе эти темы будут рассматриваться в следующих рецептах.

Рецепт 112. Встречайте: автодополнение ключевых слов

Функция автодополнения ключевых слов в редакторе Vim пытается угадать набираемое слово, что позволяет избежать необходимости ввода его вручную.

Функция автодополнения вызывается из режима вставки. В момент вызова Vim создает список слов, исходя из содержимого буфера и символов, находящихся слева от курсора. После создания списка он фильтруется – из него исключаются все слова, не начинающиеся с символов, слева от курсора. Получившийся список выводится в форме меню, из которого выбирается желаемый вариант.

Автодополнение и чувствительность к регистру

Команда поиска в редакторе Vim интерпретирует символы верхнего и нижнего регистров как эквивалентные, если включен параметр `ignorecase` (как описывается в рецепте 73 в главе 12), но этот параметр имеет еще один побочный эффект: функция автодополнения также перестает учитывать регистр символов.

В примере «She sells sea shells» ниже слово «She» не включается в список слов, потому что оно начинается с символа верхнего регистра «S». Но если бы параметр `ignorecase` был включен, слово «She» появилось бы в списке. Однако, так как был введен символ нижнего регистра «s», такое расширение списка слов по большей мере не имеет смысла.

Мы можем исправить подобное поведение, включив параметр `infercase` (см. `:h 'infercase'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'infercase'). В этом случае в список будет добавлено слово «she», начинающееся с символа нижнего регистра «s».

На рис. 19.1 представлены два скриншота, первый сделан до, а второй – после вызова функции автодополнения.

```
She sells sea shells by the s
~
~
She sells sea shells by the sea
~                           sells
~                           sea
~                           shells
-- match 2 of 3
```

Рис. 19.1. Функция автодополнения
в редакторе Vim

В данном случае список фильтруется буквой «s», и у нас на выбор остаются три варианта: «sells», «sea» и «shells». Если вам интересно узнать, почему в списке отсутствует слово «She», прочитайте врезку «Автодополнение и чувствительность к регистру» выше.

Вызов функции автодополнения

Вызвать функцию автодополнения в режиме вставки можно комбинациями клавиш <C-p> и <C-n>, которые выбирают предыдущее и следующее слово в списке соответственно.

Обе команды, <C-p> и <C-n>, вызывают обобщенную функцию автодополнения. Всего в Vim имеется несколько форм автодополнения, и все они вызываются комбинациями клавиш, начинающимися с аккорда <C-x>. В той главе мы обсудим методы, перечисленные в табл. 19.1 (полный список можно найти в разделе справки :h ins-completion ℹ http://vimdoc.sourceforge.net/htmldoc/insert.html#ins-completion).

Таблица 19.1. Наиболее часто используемые команды вызова функции автодополнения

Команда	Форма автодополнения
<C-n>	Обобщенное автодополнение
<C-x><C-n>	Автодополнение по ключевым словам в текущем буфере
<C-x><C-i>	Автодополнение по ключевым словам в подключаемом файле
<C-x><C-]>	Автодополнение по ключевым словам в индексном файле tags
<C-x><C-k>	Автодополнение по словарю
<C-x><C-l>	Автодополнение целой строки
<C-x><C-f>	Автодополнение имени файла
<C-x><C-o>	Контекстное автодополнение (omni-completion)

Если в примере «She sells sea shells» использовать последовательность <C-x><C-n>, должен появиться тот же список слов, как показано на рис. 19.1. Однако <C-n> может предлагать на выбор больше вариантов, потому что список слов заполняется не только из текущего буфера, но и из других источников. Подробнее о том, как заполняется список слов, мы поговорим в рецепте 114.

Независимо от выбранной формы автодополнения команды выбора слов из списка остаются одними и теми же. Они будут исследоваться в следующем рецепте.

Рецепт 113. Работа с меню функции автодополнения

Чтобы получить максимум от команд автодополнения, необходимо знать, как управлять раскрывающимся меню автодополнения. Либо сузьте выбор и затем укажите нужный вариант, либо закройте список, если в нем отсутствует необходимое слово.

Когда вызывается функция автодополнения, Vim выводит раскрывающееся меню со списком слов. Мы можем управлять этим меню с помощью команд, перечисленных в табл. 19.2.

Более подробное описание этих команд можно найти в разделе справки: `:h popupmenu-completion` http://vimdoc.sourceforge.net/htmldoc/insert.html#popupmenu-completion.

Независимо от используемой формы автодополнения после вывода меню автодополнения комбинации `<C-n>` и `<C-p>` всегда выполняет переход к предыдущему или к следующему элементу в меню. Напротив, в режиме вставки команды `<C-n>` и `<C-p>` вызывают обобщенное автодополнение.

Таблица 19.2. Команды управления меню автодополнения

Команда	Действие
`<C-n>`	Выбрать следующий (next) вариант в списке слов
`<C-p>`	Выбрать предыдущий (previous) вариант в списке слов
`<Down>`	Выбрать следующий вариант в списке слов
`<Up>`	Выбрать предыдущий вариант в списке слов
`<C-y>`	Принять текущий вариант (yes – да)
`<C-e>`	Вернуться к оригинальному тексту (exit – выйти из меню автодополнения)
`<C-h> (и <BS>)`	Удалить один символ из текущего совпадения
`<C-l>`	Добавить один символ из текущего совпадения
`{char}`	Закрыть меню и вставить символ `{char}`

Обе комбинации, `<C-n>` и `<Down>`, выбирают следующий элемент меню, а комбинации `<C-p>` и `<Up>` – предыдущий. Однако механика работы `<C-p>`/`<C-n>` и `<Up>`/`<Down>` несколько отличается.

Обзор списка слов без изменения документа

Когда нажимается клавиша `<Down>`, меню обновляется – выбирается следующий его элемент, но текст документа при этом не изменяется. С помощью клавиш `<Up>` и `<Down>` можно прокручивать содержимое меню со списком слов, пока не будет найдено требуемое слово. После этого найденное слово можно будет вставить в документ нажатием клавиши `<CR>` или комбинации `<C-y>`.

Изменение документа по мере прокручивания списка слов

Комбинация <C-n>, напротив, не только выбирает следующий элемент меню, но и вставляет выбранное слово в документ. Это означает, что вам не придется нажимать клавишу <CR>, чтобы подтвердить свой выбор, потому что текст документа всегда синхронизируется в соответствии с выбранным элементом меню. Закончив выбор, можно просто продолжить ввод текста, и меню закроется автоматически.

Я предпочитаю использовать комбинации <C-p> и <C-n> по двум веским причинам. При использовании этих комбинаций не нужно убирать руки из основной позиции. А кроме того, не требуется подтверждать свой выбор нажатием <CR> или <C-y>, потому что текст вставляется в документ автоматически. И снова мы столкнулись с мантрой из рецепта 47 главы 8: не убирайте руки из основной позиции на клавиатуре.

Отклонение вариантов выбора

После вызова меню с вариантами автодополнения может появиться желание закрыть его. Например, если список содержит слишком много вариантов дополнения, может оказаться быстрее ввести слово вручную. Закрыть меню можно нажатием комбинации <C-e>, которая не только закроет меню, но и восстановит прежний текст перед курсором.

Фильтрация списка по мере ввода

В этом разделе я расскажу об одном из самых моих любимых приемов при работе с меню функции автодополнения. Попробуйте нажать последовательность <C-n><C-p>. То есть выполните две отдельные команды: <C-n> и вслед за ней <C-p>. Первая команда вызовет функцию автодополнения, откроет меню и выберет первый элемент в списке слов. Вторая команда выберет предыдущий элемент в списке, вернув первоначальный вариант, но при этом оставит меню открытым. После этого можно продолжать ввод, в ходе которого Vim будет фильтровать список слов.

Этот прием особенно удобно использовать, когда список вариантов слишком велик, чтобы уместиться целиком в видимой области меню. Допустим, что мы ввели только два символа и список содержит двадцать вариантов. Если ввести третий символ, список немедленно сократится. Мы можем продолжать ввод символов, пока список не окажется достаточно коротким, чтобы можно было сделать выбор парой движений.

Этот трюк отлично работает и с другими формами автодополнения. Например, можно нажать последовательность <C-x><C-o><C-p>, чтобы обеспечить динамическую фильтрацию при использовании варианта контекстного автодополнения, или <C-x><C-f><C-p>, чтобы получить тот же эффект с автодополнением имени файла.

Рецепт 114. Источники ключевых слов

Функция обобщенного автодополнения отбирает слова в список из нескольких источников. У нас есть возможность точно указать, какие источники следует использовать при создании списка слов.

При создании списка слов некоторые формы автодополнения используют определенный файл или файлы. Функция обобщенного автодополнения использует объединение всех этих списков. Чтобы лучше понять, откуда функция обобщенного автодополнения получает слова, необходимо сначала разобраться со всеми более специализированными формами автодополнения.

Буфер

Самый простой случай заполнения списка слов – использование текста из текущего буфера. Функция автодополнения по ключевым словам в текущем буфере, вызываемая последовательностью нажатий <C-x><C-n> (см. :h compl-current ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#compl-current), именно так и поступает. Эта форма автодополнения может пригодиться, когда обобщенная форма генерирует слишком много вариантов и заранее известно, что требуемое слово имеется в текущем буфере.

Но автодополнение по ключевым словам в текущем буфере может предложить совсем немного вариантов, когда в текущем буфере содержится небольшое количество слов. Чтобы расширить список, можно указать редактору Vim, что он должен извлекать слова из всех открытых буферов. Получить список открытых буферов можно командой

```
⇒ :ls!
```

Ключевые слова для автодополнения будут извлекаться из каждого файла в этом списке, представляющем все файлы, открытые в текущем сеансе работы с редактором Vim. Как будет показано ниже, чтобы включить содержимое файла в список слов, его даже необязательно открывать.

Подключаемые файлы

Большинство языков программирования поддерживают некоторый способ загрузки кода из внешних файлов или библиотек. В языке C эту функцию выполняет директива `#include`, в Python – `import`, в Ruby – `require`. При редактировании файла, подключающего код из другой библиотеки, может оказаться полезным, если Vim будет читать содержимое подключаемых файлов при создании списка слов для автодополнения. Именно это и происходит, когда функция автодополнения вызывается нажатием последовательности `<C-x><C-i>` (см. `:h compl-keyword` ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#compl-keyword).

Редактор Vim с распознает подключаемые файлы на языке C, но его необходимо обучить распознавать соответствующие директивы в других языках, настроив параметр `include` (см. `:h 'include'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'include'). Обычно эта операция выполняется автоматически расширением распознавания типов файлов. Есть еще одно приятное обстоятельство: стандартный дистрибутив Vim уже поддерживает множество языков программирования, поэтому вам не придется изменять этот параметр настройки, если только вы не работаете с неподдерживаемым языком. Попробуйте открыть файл с исходным кодом на языке Ruby или Python и выполнить команду `:set include?`. Вы должны увидеть, что Vim уже знает, как выглядит инструкция подключения файлов в этих языках.

Индексные файлы

В главе 16 «Индексирование исходного кода и навигация по нему с помощью ctags» мы познакомились с программой Exuberant Ctags, которая сканирует файлы с исходным кодом в поисках ключевых слов, таких как имена функций, классов и любых других конструкций, имеющих особое значение в том или ином языке программиро-

вания. Обрабатывая файлы с исходными текстами, программа `ctags` генерирует алфавитный указатель (индекс) ключевых слов. В соответствии с соглашениями индекс сохраняется в файле с именем *tags*.

Основная цель индексирования проектов с помощью `ctags` – упростить навигацию по проекту, но индексный файл имеет очень удобное побочное применение: он может использоваться функцией автодополнения. Задействовать этот файл можно командой `<C-x><C-]>` (см. `:h compl-tag` ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#compl-tag).

Если слово, которое вы пытаетесь дополнить, является объектом языка (таким как имя функции или класса), автодополнение по ключевым словам в индексном файле позволит вам получить отличное соотношение сигнал/шум.

Объединяем все вместе

Форма обобщенного автодополнения генерирует варианты, объединяя списки слов, составленные на основе текущего буфера, подключаемых файлов и индексного файла. Если вам потребуется изменить такое ее поведение, прочитайте врезку «Настройка функции обобщенного автодополнения» ниже. Не забывайте, что обобщенное автодополнение вызывается простым нажатием комбинации `<C-n>`, тогда как вызов остальных, более специализированных форм начинается с комбинации `<C-x>`, за которой следует другая комбинация.

Настройка функции обобщенного автодополнения

Определить перечень источников ключевых слов, которые будут использоваться функцией обобщенного автодополнения, можно с помощью параметра настройки `complete`. Этот параметр хранит список однобуквенных флагов, разделенных запятыми, определяющих конкретные источники слов для автодополнения. По умолчанию параметр имеет значение `complete=.,w,b,u,t,i`. Мы можем запретить просмотр подключаемых файлов, выполнив следующую команду:

```
⇒ :set complete-=i
```

Или разрешить использование слов из орфографического словаря:

```
⇒ :set complete+=k
```

Описание каждого флага можно найти в разделе справки `:h 'complete'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'complete'.

Рецепт 115. Автодополнение словами из словаря

Список вариантов автодополнения конструируется редактором Vim из одного или нескольких списков слов. Мы можем настроить Vim так, что список вариантов будет конструироваться на основе орфографического словаря, входящего в состав программы проверки орфографии.

Иногда возникает желание задействовать в функции автодополнения слова, отсутствующие в открытых буферах, подключаемых файлах или индексе. В этом случае всегда можно обратиться к орфографическому словарю. Делается это с помощью команды `<C-x><C-k>` (см. `:h compl-dictionary` ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#compl-dictionary).

Чтобы включить эту возможность, необходимо указать редактору Vim, где находится подходящий список слов. Проще всего это сделать с помощью команды `:set spell`, включающей проверку орфографии в Vim (подробности смотрите в главе 20 «Поиск и исправление опечаток в Vim»). После этого все слова в орфографическом словаре будут доступны посредством команды `<C-x><C-k>`.

Если вы не желаете включать проверку орфографии, можно также установить значение параметра настройки `dictionary`, указав в нем один или несколько файлов со списками слов (см. `:h 'dictionary'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'dictionary').

Автодополнение по орфографическому словарю особенно полезно, когда требуется дополнить слишком длинное слово или слово, имеющее сложное написание. Пример одного из таких слов приводится на рис. 19.2.

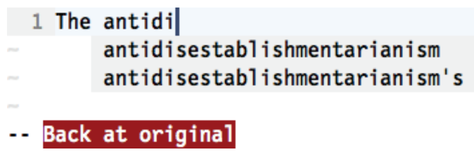


Рис. 19.2. Пример автодополнения длинного и сложного слова

Существует еще одна форма автодополнения, использующая орфографический словарь. О ее использовании рассказывается в рецепте 123 в главе 20.

Рецепт 116. Автодополнение целых строк

Во всех примерах, приводившихся до сих пор, демонстрировалось дополнение отдельных слов, но Vim способен также производить дополнение целых строк.

Режим автодополнения целых строк активируется нажатием <C-x><C-l> (см. :h compl-whole-line ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#compl-whole-line).

Допустим, у нас имеется следующий фрагмент:

auto_complete/bg-colors.css
http://media.pragprog.com/titles/dnvim/code/auto_complete/bg-colors.css

```
.top {
     background-color: #ef66ef; }
.bottom {
```

Нам требуется вставить в конец файла копию второй строки. Ниже показано, как этого добиться с применением автодополнения целых строк (табл. 19.3).

Таблица 19.3. Добавление в конец файла копии второй строки

Нажатия клавиш	Содержимое буфера

{start}	.top {
	    background-color: #ef66ef; }
	.bottom {

oba	.top {
	    background-color: #ef66ef; }
	.bottom {
	    ba

<C-x><C-l><Esc>	.top {
	    background-color: #ef66ef; }
	.bottom {
	    background-color: #ef66ef; }

Функция автодополнения целых строк использует те же источники, что и функция обобщенного автодополнения (см. рецепт 114 выше), а также генерирует список целых строк. При этом Vim игнорирует любые отступы в начале строки.

Вся прелесть автодополнения целыми строками – в отсутствии необходимости знать, где находится требуемая строка, которую тре-

буется скопировать. Достаточно лишь знать, что она существует. После ввода нескольких первых символов нажимаете `<C-x><C-l>` – и вуаля! Все остальное Vim сделает за вас!

Мы видели два других способа копирования строк с применением регистра (раздел «Создание дубликатов строк» в главе 10) и с помощью команды Ex (раздел «Копируйте строки командой :t» в главе 5). Каждый из них имеет свои достоинства и недостатки. Попробуйте самостоятельно определить, в каких ситуациях эти приемы оказываются наиболее оптимальными, и применяйте их соответственно.

Рецепт 117. Автодополнение последовательностей слов

Когда функция автодополнения используется для дополнения слова, Vim запоминает контекст, из которого это слово было извлечено. Если вызвать функцию автодополнения повторно, Vim вставит слово, следующее за исходным дополненным словом. Эту операцию можно повторять снова и снова, чтобы воспроизвести целую последовательность слов. Это часто позволяет получить желаемый результат быстрее, чем команды копирования и вставки.

Допустим, что мы работаем со следующим документом:

auto_complete/help-refs.xml
http://media.pragprog.com/titles/dnvim2/code/auto_complete/help-refs.xml

```
Here's the "hyperlink" for the Vim tutor:
<vimref href="http://vimhelp.appspot.com/usr_01.txt.html#tutor">tutor</
vimref>.
For more information on autocompletion see:
<vimr
```

Здесь тег `<vimref>` – нестандартный элемент, который я использовал повсюду в XML-рукописи с текстом этой книги, чтобы отметить ссылки на страницы справки Vim. Нам нужно вставить тег `<vimref>` в последней строке этого документа, чтобы получить элемент `:h ins-completion`, ссылающийся на раздел документации к редактору Vim. Тег будет выглядеть в точности как имеющийся тег `<vimref>`, только вместо `usr_01` нам нужно вставить `insert`, а вместо `tutor` – `ins-completion`.

Добиться желаемого можно, быстро продублировав имеющийся тег `<vimref>` и изменив необходимые части. Для этого можно переместить курсор к существующему тегу `<vimref>`, скопировать его в буфер обмена, затем вернуться обратно и вставить скопированный текст. В данном документе это не вызовет больших проблем, потому что он очень короткий. Но если бы мы работали с документом, содержащим тысячи слов, в поисках экземпляра `<vimref>` для копирования мы могли бы уйти очень далеко от точки, куда требуется выполнить вставку.

Как вариант для вставки копии тега `<vimref>` можно использовать функцию автодополнения, никуда не перемещая курсора:

Нажатие клавиш	Содержимое буфера
`{start}`	`<vimr`
`a<C-x><C-p>`	`<vimref`
`<C-x><C-p>`	`<vimref>.`
`<C-p>`	`<vimref href`
`<C-x><C-p>`	`<vimref href"http`
`<C-x><C-p>`	`<vimref href"http://vimhelp`

Сначала мы нажали `a`, чтобы переключиться в режим вставки. Теперь можно использовать `<C-x><C-p>` для автоматического дополнения части слова `vimr` до `vimref`. (В этом примере можно было бы просто использовать `<C-x><C-n>` повсюду.)

После второго нажатия `<C-x><C-p>` получился интересный результат. Vim запоминает контекст, в котором находится слово `vimref` для автодополнения. Когда мы вызвали команду автодополнения второй раз, Vim дополнил слово, следующее за `vimref`. В данном случае есть два возможных варианта: слово `vimref` может присутствовать в открывающем и закрывающем тегах. Vim вывел меню с вариантами автодополнения, предлагая выбрать контекст, который следует использовать для дальнейшего дополнения. Нажав `<C-p>` во второй раз, мы получили желаемый результат.

Теперь можно продолжить нажимать `<C-x><C-p>` снова и снова. С каждым вызовом этой команды Vim будет вставлять следующее слово из того, что обнаружит в контексте, откуда заимствуется текст для автодополнения. При таком подходе заполнение оставшейся части тега XML не займет много времени. После чего можно приступать к редактированию тега вручную и заменить `usr_01` на `insert` и `tutor` на `ins-completion`.

Функция автодополнения в Vim позволяет вставлять не только последовательности слов. Она также может работать с последовательностями строк. Если многократно использовать команду `<C-x><C-l>` (с которой мы познакомились в рецепте 116 выше), с ее помощью можно вставить последовательность смежных строк из любого места в документе.

Возможность вставлять последовательности слов и строк часто позволяет дублировать текст быстрее, чем команды копирования и вставки. Когда ваши коллеги увидят, как вы пользуетесь этим приемом, они обязательно спросят, как вы это делаете!

Рецепт 118. Автодополнение имен файлов

При работе в командной строке можно нажать клавишу `<Tab>` и автоматически дополнить путь к файлу или каталогу. С помощью функции автодополнения имен файлов можно делать то же самое в редакторе Vim.

Автодополнение имен файлов вызывается командой `<C-x><C-f>` (см. `:h compl-filename` ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#compl-filename).

Редактор Vim всегда помнит путь к текущему рабочему каталогу, подобно командной оболочке. Получить его можно командой `:pwd` (print working directory – вывести рабочий каталог), а изменить – командой `:cd {path}` (change directory – изменить каталог). Важно понимать, что функция автодополнения имен файлов в редакторе Vim всегда дополняет пути относительно текущего рабочего каталога, а не относительно каталога, в котором находится текущий редактируемый файл.

Допустим, что мы разрабатываем небольшое веб-приложение, состоящее из следующих файлов:

```
webapp/
  config.ru
  public/
    index.html
    js/
      application.js
```

и в данный момент занимаемся правкой файла *index.html*:

auto_complete/webapp/public/index.html
http://media.pragprog.com/titles/dnvim/code/auto_complete/webapp/public/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Practical Vim - the app</title>
    <script src="" type="text/javascript"></script>
  </head>
  <body></body>
</html>
```

Нам нужно подставить в атрибут `src=""` ссылку на файл *application.js*. Но если воспользоваться функцией автодополнения имен файлов, у нас возникнут сложности:

```
⇒ :pwd
  webapp
```

Если вызвать автодополнение имен файлов прямо сейчас, она добавит относительный путь к файлу, начинающийся в каталоге *webapp*, дав нам в результате: `src="public/js/application.js"`. А нам в действительности требуется такая ссылка: `src="js/application.js"`. Чтобы воспользоваться функцией автодополнения имен файлов в данной ситуации, необходимо сначала перейти в каталог *public*:

```
⇒ :cd public
```

Теперь можно вызвать функцию автодополнения имен файлов, и на этот раз она будет подставлять пути относительно каталога *webapp/public*:

Нажатия клавиш	Содержимое буфера
`i`	`<script src=""/>`
`js/ap`	`<script src="js/ap"/>`
`<C-x><C-f>`	`<script src="js/application.js"/>`

После вставки пути к файлу можно вернуться обратно, в первоначальный рабочий каталог:

```
⇒ :cd -
```

Как и в командной оболочке, команда `:cd-` выполнит переход в предыдущий рабочий каталог (см. `:h :cd-` ⓘ http://vimdoc.sourceforge.net/htmldoc/editing.html#:cd-).

В документации с описанием автодополнения имен файлов отмечается, «что параметр `path` не используется (пока)». Возможно, в будущих версиях Vim не будет требовать переходить в другой каталог для использования этой функции, как в нашем гипотетическом веб-приложении.

Рецепт 119. Контекстное автодополнение

Контекстное автодополнение (omni-completion) – это ответ Vim на появление механизма IntelliSense. Оно формирует список вариантов, исходя из контекста, определяемого местоположением курсора. В данном рецепте мы посмотрим, как действует эта форма автодополнения в контексте файла CSS.

Функция контекстного автодополнения вызывается командой `<C-x><C-o>` (см. `:h compl-omni` ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#compl-omni). Данная функциональность реализована в виде расширения, определяющего тип файла, поэтому, чтобы активировать ее поддержку, необходимо добавить следующие строки в конфигурационный файл:

essential.vim
http://media.pragprog.com/titles/dnvim/code/essential.vim

```
set nocompatible
filetype plugin on
```

Нам также придется установить расширение, реализующее контекстное автодополнение для языка, с которым мы работаем. В состав дистрибутива Vim включена поддержка около десятка языков, включая HTML, CSS, JavaScript, PHP и SQL. Полный список можно найти в разделе справки `:h compl-omni-filetypes` ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#compl-omni-filetypes.

На рис. 19.3 показаны результаты вызова функции контекстного автодополнения для файла CSS в двух разных контекстах. В первом случае для фрагмента «ba» при вводе имени свойства CSS выводится список вариантов, включая `background`, `background-attachment` и еще несколько других. В этом примере выбирается имя `background-color`. Во втором случае функция контекстного автодополнения была вызвана без ввода начального фрагмента, но Vim смог, исходя из контекста, определить, что ожидается значение цвета, поэтому он предложил три варианта: `#`, `rgb(` и `transparent`.

```
1 h1 { ba
       background:
       background-attachment:
       background-color:

1 h1 { background-color: 
                         transparent
                         rgb(
                         #
```

Рис. 19.3. Контекстное автодополнение свойств и значений CSS

Статическая природа CSS хорошо подходит для контекстного автодополнения, но если попробовать использовать эту функцию в исходном коде на каком-нибудь языке программирования, список предлагаемых вариантов может вырасти значительно. Если вас не устраивает поддержка кого-то конкретного языка, поищите альтернативное расширение или напишите свое. Прекрасной отправной точкой для тех, кто решится написать свое расширение, может служить раздел справки `:h complete-functions` ⓘ http://vimdoc.sourceforge.net/htmldoc/insert.html#complete-functions.

Глава 20. Поиск и исправление опечаток в Vim

Механизм проверки орфографии в Vim упрощает поиск и исправление орфографических ошибок. В рецепте 120 я расскажу, как этот механизм действует в командном режиме, а в рецепте 123 вы узнаете, что он также может выполнять свои функции и в режиме вставки.

Обычно в состав дистрибутива Vim входит только файл орфографического словаря для английского языка, но вы с легкостью сможете установить поддержку других языков. Кроме того, как будет показано в рецепте 121, имеется возможность переключаться между американским и британским диалектами английского языка (кроме прочих). А если слово помечено как ошибочное, но вы уверены в его правильности, добавить такое слово в словарь проще простого, как будет показано в рецепте 122.

Рецепт 120. Проверьте свой текст

При включенной поддержке проверки орфографии Vim помечает слова, отсутствующие в словаре. Вы легко можете перемещаться между ошибками и исправлять их, как предлагает Vim.

Взгляните на следующий фрагмент текста:

spell_check/yoru-moustache.txt
http://media.pragprog.com/titles/dnvim/code/spell_check/yoru-moustache.txt

```
Yoru mum has a moustache.
```

Первое слово, по всей видимости, написано с ошибкой. Мы можем заставить Vim подсвечивать такие слова, включив встроенный механизм проверки орфографии:

```
⇒ :set spell
```

Слово «Yoru» должно быть помечено как ошибочное. Обычно это выражается в подчеркивании красной волнистой линией, впрочем, стиль выделения зависит от используемой цветовой схемы.

По умолчанию для проверки используется орфографический словарь английского языка. В рецепте 121 будет показано, как настроить поддержку других языков, а пока будем пользоваться настройками по умолчанию.

Принцип действия механизма проверки орфографии в Vim

Перемещаться взад и вперед между орфографическими ошибками можно с помощью команд [s и]s соответственно (см. :h]s ⓘ http://vimdoc.sourceforge.net/htmldoc/spell.html#]s). Установив курсор на ошибочное слово, можно запросить у редактора список вариантов исправления ошибки, выполнив команду z= (см. :h z= ⓘ http://vimdoc.sourceforge.net/htmldoc/spell.html#z=). На рис. 20.1 представлены скриншоты, выполненные до и после вызова команды z=.

```
1 Yoru mum has a moustache.
~
```

```
Change "Yoru" to:
 1 "Your"
 2 "Yore"
 3 "York"
Type number and <Enter>:
```

Рис. 20.1. Вызов вариантов исправления орфографической ошибки

Как показано на рис. 20.1, мы можем заменить слово с ошибкой вариантом «Your», введя 1<CR>. Если список не содержит необходимого варианта, можно нажать клавишу <Esc>, чтобы скрыть список.

Можно также вообще предотвратить вывод списка, снабдив команду z= счетчиком, значение которого будет определять выбранный вариант. Если есть уверенность, что первый вариант будет именно тем, что необходим, можно выполнить команду 1z=, чтобы исправить ошибку одним движением.

В процессе работы над текстом я предпочитаю отделять этап ввода от этапа проверки. Я часто отключаю проверку орфографии, когда набираю текст, чтобы не отвлекаться каждый раз, когда я допускаю опечатку. Закончив ввод текста, я выполняю заключительный этап – произвожу поиск орфографических ошибок в документе и исправляю их.

В следующей таблице перечислены основные команды командного режима для работы с механизмом проверки орфографии:

Команда	Действие
`]s`	Выполняет переход к следующей орфографической ошибке
`[s`	Выполняет переход к предыдущей орфографической ошибке
`z=`	Предлагает варианты исправления ошибки
`zg`	Добавляет текущее слово в орфографический словарь
`zw`	Удаляет текущее слово из орфографического словаря
`zug`	Отменяет команду `zg` или `zu`, выполненную для текущего слова

Мы еще встретимся с командами `zg`, `zw` и `zug` в рецепте 122 ниже.

Рецепт 121. Использование альтернативных орфографических словарей

По умолчанию механизм проверки орфографии в Vim поддерживает региональные диалекты английского языка. В этом рецепте я расскажу, как указать диалект и получить словари для других языков.

Когда механизм проверки орфографии активен, по умолчанию он пытается искать слова в орфографическом словаре английского языка. Изменить язык можно с помощью параметра настройки `spelllang` (см. `:h 'spelllang'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'spelllang'). Действие этого параметра распространяется только на текущий буфер. То есть можно одновременно работать с двумя или более документами и использовать разные словари – очень удобно для тех, кому приходится набирать тексты на двух и более языках.

Настройка диалекта языка

Механизм проверки орфографии в редакторе Vim поддерживает несколько диалектов английского языка. С настройкой `spelllang=en` по умолчанию допустимыми считаются любые слова, присутствующие в любом из диалектов английского языка. Не важно, как будет записано слово: «moustache» (британский английский) или «mustache» (американский английский), оба они будут считаться правильными.

Мы можем сообщить Vim, что проверка орфографии должна осуществляться с правилами правописания американского английского:

```
➾ :set spell
➾ :set spelllang=en_us
```

После этого слово «moustache» будет помечаться как ошибочное, а «mustache» будет рассматриваться как правильное. В число других поддерживаемых диалектов входят: `en_au`, `en_ca`, `en_gb` и `en_nz`. Подробности смотрите в разделе справки `:h spell-remarks` ⓘ http://vimdoc.sourceforge.net/htmldoc/spell.html#spell-remarks.

Получение словарей для других языков

В состав дистрибутива Vim входит только орфографический словарь английского языка, однако вы можете найти десятки словарей для других языков по адресу http://ftp.vim.org/vim/runtime/spell/.

Если попытаться настроить проверку орфографии языка, для которого отсутствует файл словаря, Vim предложит загрузить и установить его:

```
➾ :set spell
➾ :set spellang=ru
  Cannot find spell file for "ru" in utf-8
  Do you want me to try downloading it?
  (Y)es, [N]o:
➾ Y
  Downloading ru.utf-8.spl
  In which directory do you want to write the file:
  1. /Users/drew/.vim/spell
  2. /Applications/MacVim.app/Contents/Resources/vim/runtime/spell
  [C]ancel, (1), (2):
```

Эта возможность поддерживается посредством расширения с именем spellfile.vim, который входит в состав дистрибутива Vim (см. `:h spellfile.vim` ⓘ http://vimdoc.sourceforge.net/htmldoc/spell.html#spellfile.vim). Чтобы он заработал, добавьте следующие две строки в файл *vimrc* (в самый конец):

```
set nocompatible
plugin on
```

Рецепт 122. Добавление слов в орфографический словарь

Орфографические словари не претендуют на абсолютную полноту, и у нас есть возможность добавлять в них новые слова.

Иногда Vim может ошибаться и отмечать как неправильные вполне допустимые слова только потому, что они отсутствуют в словаре. Вы можете обучать Vim распознавать слова с помощью команды `zg` (`:h zg` ⓘ http://vimdoc.sourceforge.net/htmldoc/spell.html#zg), которая добавляет слово под курсором в словарь.

Редактор поддерживает также парную ей команду `zw`, помечающую слово под курсором как ошибочное. В действительности эта команда удаляет текущее слово из словаря. Если команда `zg` или `zw` окажется вызванной случайно, мы по ошибке добавим слово в словарь или удалим его оттуда. Как раз для таких случаев Vim поддерживает отдельную команду отмены `zug`, которая отменяет команду `zg` или `zw` для слова под курсором.

Vim записывает добавляемые слова непосредственно в файл словаря, поэтому они сохраняются между сеансами редактирования. Имя файла словаря конструируется из названия языка и кодировки символов в файле.

Например, команда `zg` будет сохранять слова английского языка в кодировке UTF-8 в файле с именем *~/.vim/spell/en.utf-8.add*.

Создание словаря для специальных терминов

В параметре настройки `spellfile` можно указать путь к файлу, куда Vim будет добавлять слова при вызове команды `zg` и откуда он будет удалять при вызове команды `zw` (см. `:h 'spellfile'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'spellfile').

Мы можем указать сразу несколько файлов словарей, то есть поддерживать несколько списков слов.

Эта книга содержит множество текстовых элементов, которые не считаются словами, включая команды Vim (такие как `ciw`) и имена параметров настройки (такие как `spelllang`). Я не хотел, чтобы Vim помечал их как ошибочные, но при этом я не хотел также, чтобы они считались допустимыми словами английского языка. В качестве компромиссного решения я настроил поддержку отдельного списка слов с терминами Vim. Я в любой момент мог загрузить этот словарь, создававшийся в процессе работы над книгой.

По окончании работы над очередной главой я подключал файл, содержащий следующие настройки:

spell_check/spellfile.vim

```
setlocal spelllang=en_us
setlocal spellfile=~/.vim/spell/en.utf-8.add
setlocal spellfile+=~/books/practical_vim/jargon.utf-8.add
```

Файл *~/.vim/spell/en.utf-8.add* служит хранилищем по умолчанию, где Vim хранит слова, добавляемые командой `zg`. Файл *~/books/practical_vim/jargon.utf-8.add* хранится в репозитории с исходными текстами книги, куда я записываю термины Vim.

Теперь для каждого слова, которое механизм проверки орфографии помечает как ошибочное, у меня есть на выбор два варианта действий. Я могу вызвать команду `2zg`, чтобы добавить его в список терминов Vim, или `1zg`, чтобы добавить его в список слов, используемый по умолчанию.

Рецепт 123. Исправление орфографических ошибок в режиме вставки

Механизм проверки орфографии в редакторе Vim позволяет исправлять опечатки, не покидая режима вставки.

Представьте следующую ситуацию: мы только что ввели строку текста и затем заметили, что допустили опечатку в ее начале. Что делать?

Подготовка

Этот прием зависит от установки параметра `spell`:

```
⇒ :set spell
```

Обычный способ: выход в командный режим

Чтобы исправить ошибку, можно выйти в командный режим, вызвать команду `[s`, чтобы вернуться назад к ошибке, и затем командой `1z=` исправить ее. Выполнив исправление, мы можем вернуться в режим вставки командой `A` и продолжить с того места, где мы прервались.

Быстрый способ: использовать орфографическое автодополнение

Ошибку можно также исправить, не покидая режима вставки, вызвав команду `<C-x>s`, которая запускает специальную разновид-

ность функции автодополнения (см. `:h compl-spelling` http://vimdoc.sourceforge.net/htmldoc/insert.html#compl-spelling). С той же целью можно было бы вызвать команду `<C-x><C-s>`, которая вводится немного удобнее. На рис. 20.2 представлены скриншоты, полученные до и после вызова команды `<C-x>s`. Обратите внимание, что оба скриншота получены в режиме вставки.

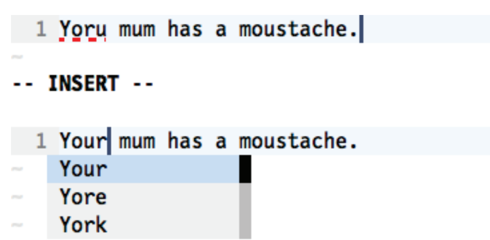


Рис. 20.2. Вызов функции автодополнения в режиме вставки

Меню со списком слов содержит те же варианты, что мы видели в рецепте 120 выше, когда использовали команду `z=`.

Когда вызывается команда автодополнения, Vim обычно предлагает варианты дополнения слова под курсором. Но когда вызывается команда `<C-x>s`, Vim сканирует текст назад от позиции курсора и останавливается в месте обнаружения орфографической ошибки. Затем он конструирует список слов с вариантами исправления и выводит их в виде меню функции автодополнения. Тот или иной вариант можно выбрать с применением любого приема из числа описанных в рецепте 113 главы 19.

Команда `<C-x>s` по-настоящему удобна, когда в строке имеется более одной ошибки. Продолжая пример, представленный на рис. 20.2: если была выполнена команда `:set spelllang=en_us`, слово «moustache» также было бы отмечено как ошибочное. Находясь в конце строки, в режиме вставки, мы могли бы исправить обе ошибки, просто дважды нажав `<C-x>s`. Попробуйте сами, это так здорово!

Глава 21. Что дальше?

Поздравляю, вы добрались до конца книги «Практическое использование Vim»! И что дальше?

Продолжайте практиковаться!

Продолжайте работать с этим замечательным редактором. Со временем многое из того, что казалось необычным, превратится во вторую натуру. Поставьте перед собой цель достичь такого уровня владения редактором Vim, когда вы не будете задумываться о своих действиях. Добившись этого, вы сможете править текст со скоростью мысли.

При создании книги «Практическое использование Vim» я вовсе не рассчитывал, что вы будете наращивать свои навыки по мере ее прочтения, поэтому вы наверняка не сможете освоить все за один проход. Некоторые из рецептов достаточно просты, тогда как другие рассчитаны на опытных пользователей. Я полагаю, что вы еще не раз возьмете эту книгу в руки и каждый раз будете узнавать что-то новое.

Настраивайте Vim под себя

В основном в этой книге мы использовали Vim с настройками по умолчанию, какие он получает сразу после установки. Эти настройки могут служить неплохой отправной точкой, обеспечивая универсальный набор поддерживаемых особенностей, но я не собираюсь предлагать вам продолжать придерживаться их. Некоторые настройки в редакторе Vim имеют довольно неудачные значения по умолчанию. Если вы задаетесь вопросом, почему та или иная функция действует именно так, а не иначе, обычный ответ на него: «потому что так работал vi!».

Но вы совершенно не обязаны придерживаться настроек по умолчанию. Вы можете настраивать поведение Vim под свои привычки и желания. Если вы сохраните свои настройки в файле *vimrc*, редактор всегда будет настроен так, чтобы лучше соответствовать привычному для вас рабочему процессу. В приложении 1 «Настройка Vim в соответствии с личными предпочтениями» приводится простой пример, который вы можете взять за основу.

Узнайте, как пользоваться пилой, и только потом точите ее

В своем классическом эссе «Семь привычек эффективного редактирования текстов» Брэм Муленаар (Bram Moolenaar) советует тратить время на оттачивание навыков[1]. Построение файла *vimrc* является одним из таких навыков, но при этом очень важно, чтобы вы понимали, как действует базовая функциональность редактора, прежде чем на ее основе возводить свое здание. Сначала узнайте, как пользоваться пилой. Затем наточите ее.

Я встречал людей, настраивавших Vim так, что эти настройки мешали оттачиванию навыков. Мне встречались даже такие, кто «точил пилу» абсолютно неправильно! Не волнуйтесь, прочитав книгу «Практическое использование Vim», вы не допустите таких ошибок. Вы уже познакомились с основами Vim.

Выстраивайте свой файл *vimrc* с нуля. В качестве основы можно использовать файл *essential.vim*, входящий в состав загружаемого пакета примеров. Многие пользователи Vim выкладывают свои файлы *vimrc* в Интернете, и они могут служить отличным источником вдохновения. Копируйте из них фрагменты, помогающие вам решать ваши проблемы, но не трогайте то, что вам не нужно. Ваш файл *vimrc* должен быть только вашим. То есть вы должны понимать, все, что в нем находится.

Я упорно сопротивлялся соблазну поделиться лучшими фрагментами из моего файла *vimrc* (в противном случае объем книги удвоился бы), но я добавил в книгу свои подсказки в виде врезок. Мой файл *vimrc* можно найти на сайте Github наряду со многими моими конфигурационными файлами[2]. Кроме того, некоторые свои настройки я задокументировал на сайте Vimcasts.org, где я также

---

1 http://www.moolenaar.net/habits_ru.html
2 http://github.com/nelstrom/dotfiles

рекламирую некоторые из моих самых любимых расширений[1].

Редактор Vim распространяется на условиях не совсем обычной лицензии: charityware (благотворительная) (`:h license` ⓘ http://vimdoc.sourceforge.net/htmldoc/uganda.html#license). Это означает, что вы бесплатно можете пользоваться редактором, но приветствуются любые посильные пожертвования фонду ICCF Holland[2], который помогает нуждающимся детям в Уганде. Пожалуйста, выкажите свое уважение к авторам Vim, пожертвовав на это благородное дело.

⇛ `:x`

---

1 http://vimcasts.org/
2 http://iccf-holland.org/

Приложение 1. Настройка Vim в соответствии с личными предпочтениями

Центром внимания этой книги является овладение базовой функциональностью редактора Vim, но некоторые настройки по умолчанию могут не соответствовать вашим личным предпочтениям. Vim может настраиваться в очень широких пределах; и мы можем конфигурировать его в соответствии с личными предпочтениями.

Изменяйте настройки на лету

Редактор Vim имеет сотни параметров, позволяющих нам настраивать его поведение (полный список можно найти в разделе `:h option-list` ⓘ http://vimdoc.sourceforge.net/htmldoc/quickref.html#option-list). Для их изменения используется команда `:set`.

Давайте возьмем в качестве примера параметр `ignorecase` (обсуждается в рецепте 73 в главе 12). Это – логический параметр: он может быть либо включен, либо выключен. Включить его можно командой

```
⇒ :set ignorecase
```

Чтобы выключить его, к имени параметра нужно добавить приставку «no»:

```
⇒ :set noignorecase
```

Если в конец имени логического параметра добавить восклицательный знак, его значение можно переключить на противоположное:

```
⇒ :set ignorecase!
```

Если добавить вопросительный знак, команда `:set` выведет текущее состояние параметра:

```
⇒ :set ignorecase?
  ignorecase
```

В конец имени можно также добавить знак амперсанда &, чтобы сбросить параметр в значение по умолчанию:

```
⇒ :set ignorecase&
⇒ :set ignorecase?
  noignorecase
```

Некоторые параметры настройки принимают строковое или числовое значение. Например, параметр `tabstop` определяет ширину символа табуляции в количестве знакомест (`:h 'tabstop'` ⓘ http://vimdoc.sourceforge.net/htmldoc/options.html#'tabstop'). Значение этого параметра можно установить, как показано ниже:

```
⇒ :set tabstop=2
```

В одной команде `:set` можно присваивать значения нескольким параметрам:

```
⇒ :set ts=2 sts=2 sw=2 et
```

Параметры `softtabstop`, `shiftwidth` и `expandtab` обеспечивают дополнительную гибкость в настройке отступов. Подробнее о табуляции и пробелах можно узнать в обучающем видеоролике на сайте Vimcasts[1].

Для большинства параметров настройки Vim имеются сокращенные варианты. Так, имя параметра `ignorecase` можно сократить до `ic`, то есть переключить этот параметр можно командой `:se ic!` или выключить командой `:se noic`. При настройке поведения Vim на лету я предпочитаю пользоваться краткими именами параметров для простоты, а в файле *vimrc* использую в основном длинные имена для удобочитаемости.

Обычно параметры настройки Vim действуют глобально, но есть и такие, область действия которых ограничивается окном или бу-

[1] http://vimcasts.org/e/2

фером. Например, настройка, выполняемая командой `:setlocal tabstop=4`, применяется только к активному буферу. То есть мы можем открыть несколько файлов и определить значение параметра `tabstop` для каждого из них в отдельности. Если потребуется применить некоторое значение ко всем открытым буферам, можно выполнить следующую команду:

```
⇒ :bufdo setlocal tabstop=4
```

Параметр `number` может настраиваться для каждого окна в отдельности. Когда выполняется команда `:setlocal number`, она включает нумерацию строк в активном окне. Если необходимо включить нумерацию во всех окнах, следует выполнить следующую команду:

```
⇒ :windo setlocal number
```

Команда `:setlocal` ограничивает область действия изменяемого параметра текущим окном или буфером (если только параметр не имеет исключительно глобальную область действия). Если выполнить команду `:set number`, она включит нумерацию строк в текущем окне и запомнит это значение как новое глобальное значение по умолчанию. Уже открытые окна сохранят свои настройки, но новые будут получать новую глобальную настройку.

Сохраняйте настройки в файле vimrc

Изменение настроек на лету прекрасно подходит для удовлетворения сиюминутных потребностей, но если у вас появится желание установить особенно полюбившиеся вам настройки, разве не будет лучше обеспечить их сохранность между сеансами?

Для этого настройки можно сохранить в файле, а затем восстанавливать их командой `:source {file}` из указанного файла `{file}` (`:h :source` ⓘ http://vimdoc.sourceforge.net/htmldoc/repeat.html#:source). Когда выполняется загрузка файла таким способом, Vim выполняет каждую строку как команду Ex, как если бы она была введена в режиме командной строки.

Допустим, что мы часто работаем с файлами, где отступы оформляются двумя пробелами. Мы могли бы создать файл с соответствующими настройками и сохранить его на диске:

customizations/two-space-indent.vim
http://media.pragprog.com/titles/dnvim/code/customizations/two-space-indent.vim

```
"" Оформлять отступы двумя пробелами
set tabstop=2
set softtabstop=2
set shiftwidth=2
set expandtab
```

Всякий раз, когда понадобится применить эти настройки к текущему буферу, достаточно будет выполнить следующую команду:

```
⇒ :source two-space-indent.vim
```

При изменении настроек на лету мы начинаем ввод с символа двоеточия, чтобы переключиться в режим командной строки. Однако, когда настройки сохраняются в файле, они необязательно должны начинаться с двоеточия, потому что команда `:source` предполагает, что каждая строка в указанном файле должна выполняться как команда Ex.

В момент запуска редактор Vim проверяет наличие файла с именем *vimrc*. Если файл найден, Vim автоматически загружает его содержимое. Благодаря этому избранные настройки можно хранить в файле *vimrc*, и они автоматически будут применяться при каждом запуске Vim.

Vim пытается найти файл *vimrc* в нескольких местах (см. `:h vimrc` ⓘ http://vimdoc.sourceforge.net/htmldoc/starting.html#vimrc). В системах Unix пытается найти файл с именем *~/.vimrc*. В Windows – файл `$HOME/`*_vimrc*. Не важно, какой системой вы пользуетесь, вы можете открыть этот файл прямо из редактора Vim следующей командой:

```
⇒ :edit $MYVIMRC
```

`$MYVIMRC` – это переменная окружения в Vim, хранящая путь к файлу *vimrc*. После сохранения изменений в файле *vimrc* мы можем загрузить новую конфигурацию командой

```
⇒ :source $MYVIMRC
```

Если файл *vimrc* открыт в активном буфере, эту команду можно сократить до `:so %`.

Применение настроек для определенных типов файлов

Наши предпочтения могут отличаться для файлов разных типов. Например, допустим, что мы следуем рекомендациям по оформлению исходных текстов, которые в языке Ruby советуют использовать отступы шириной два пробела, а в языке JavaScript – четыре. Мы можем воплотить эти рекомендации, добавив следующие строки в файл *vimrc*:

customizations/filetype-indentation.vim
http://media.pragprog.com/titles/dnvim/code/customizations/filetype-indentation.vim

```
if has("autocmd")
  filetype on
  autocmd FileType ruby setlocal ts=2 sts=2 sw=2 et
  autocmd FileType javascript setlocal ts=4 sts=4 sw=4 noet
endif
```

Объявление `autocmd` сообщает редактору организовать прием указанного события и выполнять указанную команду, как только оно появится (`:h :autocmd` ⓘ http://vimdoc.sourceforge.net/htmldoc/autocmd.html#:autocmd). В данном случае настраивается реакция на событие `FileType`, которое генерируется, когда Vim определяет тип текущего файла.

Для одного и того же события можно предусмотреть выполнение нескольких команд `autocmd`. Допустим, что нам требуется использовать программу `nodelint` в качестве компилятора для файлов JavaScript. С этой целью мы могли бы добавить следующую строку в пример выше:

```
autocmd FileType javascript compiler nodelint
```

Обе команды будут выполняться всякий раз, когда событие `FileType` окажется сгенерировано для файла JavaScript.

Добавление подобных команд в файл *vimrc* отлично подходит для случая, когда вам необходимо изменить один-два параметра настройки для файлов некоторого типа. Но если требуется применить массу настроек, файл начинает разрастаться непомерно. Решением проблемы может стать расширение ftplugin – альтернативный механизм применения настроек, зависящих от типов файлов. Вместо

объявления наших предпочтений в отношении файлов JavaScript в файле *vimrc* с помощью `autocmd` мы можем поместить их в файл с именем `~/.vim/after/ftplugin/javascript.vim`:

customizations/ftplugin/javascript.vim
http://media.pragprog.com/titles/dnvim/code/customizations/ftplugin/javascript.vim

```
setlocal ts=4 sts=4 sw=4 noet
compiler nodelint
```

Этот файл имеет точно такую же организацию, как и файл *vimrc*, за исключением того, что настройки из него применяются только к файлам JavaScript. Точно так же мы могли бы создать файл *ftplugin/ruby.vim* с целью хранения настроек для языка Ruby, как и для каждого типа файлов, с которыми приходится работать регулярно. Дополнительные подробности можно найти в разделе справки `:h ftplugin-name` ⓘ http://vimdoc.sourceforge.net/htmldoc/usr_05.html#ftplugin-name.

Для работы механизма ftplugin необходимо гарантировать включение обоих механизмов: определения типа файлов и поддержку расширений. Убедитесь, что в вашем файле *vimrc* присутствует следующая строка:

```
filetype plugin on
```

Предметный указатель

Символы

! (восклицательный знак)
в командах, 130
!{motion}, команда-оператор, 64
"+p, команда, 198
"0p, команда, 200
"ay$, команда, 232
(символ решетки)
  в регулярных выражениях, 239
  значение в списке буферов, 123
$, команда, 43
$MYVIMRC, переменная
окружения, 366
% (процент)
  значение в списке буферов, 123
  как диапазон, 283
  как диапазон, включающий весь файл, 98
%, команда, 177
&, команда, 286
&, флаг, команды подстановки, 273
() (круглые скобки)
  в регулярных выражениях, 238, 242
  поиск без сохранения совпадений, 245
* (звездочка), команда, 110, 269
  поиск слова под курсором, 49
** групповой символ, 144
, (запятая), команда, 47
, (запятая), команда повторения
поиска в обратном направлении, 164
. (точка), команда, 39
  в визуальном режиме, 83
  микромакроопределение, 42
  повторение добавления отступов, 82
:, команда, переход в режим
командной строки, 92
:!ls, команда, 115
:&&, команда, 285
:>, команда, 306
:argdo, команда, 129, 132, 224
  в комбинации с командой :global, 302
:args, команда, 126
:bdelete, команда, 125
:bnext, команда, 93, 123, 124
повторение, 107
:bprev, команда, 93, 124
:bprevious, команда, 107
:bufdo, команда, 124, 132
  в комбинации с командой :global, 302
:buffer, команда, 124
:cc N, команда, 323
:cclose, команда, 323
:cd {path}, команда, 349
:cfirst, команда, 323
:clast, команда, 323
:close, команда, 135
:cnewer, команда, 326
:cnext, команда, 323
:cnfile, команда, 323
:colder, команда, 326
:compiler, команда, 329
:copen, команда, 323

:copy, команда Ex, 101
:cpfile, команда, 323
:cprev, команда, 323
:delete, команда, в комбинации с командой :global, 297
:E, команда, 147
:e., команда, 147
:echo, команда, 228
:edit, команда, 93, 140, 141
:Explore, команда, 147
:find, команда, 143, 145
:global, команда, 297
    в комбинации с командами Ex, 305
    в комбинации с командой :argdo, 302
    в комбинации с командой :bufdo, 302
    в комбинации с командой :delete, 297
    в комбинации с командой :yank, 300
:grep, команда, 331
    настройка, 333
:lcd, команда, 137
:let, команда, для сохранения содержимого в регистре, 281
:lgrep, команда, список адресов, 324
:lmake, команда, список адресов, 324
:ls, команда, 115
:lvimgrep, команда, список адресов, 324
:m, команда Ex, 103
:make, команда, 319, 321
    настройка внешнего компилятора, 326
    настройка на вызов программы nodelint, 327
:move, команда, 103
:move, команда Ex, 101
:next, команда, 93, 129
:normal, команда, 105, 220
:only, команда, 135
:prev, команда, 93, 129
:print, команда, 96
    как команда, выполняемая по умолчанию командой :global, 301
:put, команда, 231
:pwd, команда, 141
:quit, команда, 131
:Rcontroller, команда, 144
:read, команда, 117
:Rmodel, команда, 144
:Rview, команда, 144
:set nohlsearch, команда, 256
:set spell, команда, 347
:set, команда, 364
:setlocal, команда, 365
:Sexplore, команда, 147
:shell, команда, 116
:sort, команда, 303
:source {file}, команда, 365
:split, команда, 134
:substitute, команда, 98, 259, 273
    выполнение арифметических операций в строке замены, 287
    замена содержимым регистра, 280
    конструирование списка файлов, содержащих совпадения с шаблоном, 294
    основы, 273
    перемена местами двух и более слов, 289
    переупорядочение полей в файле CSV, 286
    повторение предыдущей команды подстановки, 283
    повторное использование последнего шаблона поиска, 278
    подтверждение каждой подстановки, 276
    поиск и замена в нескольких файлах, 292
    поиск и замена всех совпадений в файле, 274
    специальные символы в строке замены, 274
:substitute, команда, 47
:Subvert, команда, 291
:t, команда Ex, 102
:tabclose, команда, 138
:tabedit, команда, 138
:tabmove, команда, 139

:tabnew, команда, 93
:tabonly, команда, 138
:tag /{pattern}, команда, 317
:tag {keyword}, команда, 317
:tjump /{keyword}, команда, 317
:tjump {keyword}, команда, 317
:tnext, команда, 317
:tselect, команда, 316
:Vexplore, команда, 147
:vglobal, команда, 297
:vimgrep, команда, 188, 331, 337
  и история поиска, 338
:visual, команда, 94
:vsplit, команда, 134
:wall, команда, 226
:windo, команда, 138
:wnext, команда, 226
:write, команда, 93, 117, 140
:yank, команда, в комбинации с командой :global, 300
; (точка с запятой), команда, повторный поиск, 46
;, команда повторения поиска, 163
@, команда, 46
@:, команда, повторение команд Ex, 107
@@, команда, повторения, 107
@{register}, команда выполнения макроса, 210
[] (квадратные скобки) в регулярных выражениях, 238, 242
[N]<C-w>_, команда, 135
[N]<C-w>|, команда, 135
[s, команда перехода к предыдущему ошибочному слову, 354
\ (обратный слеш) в строке замены команды подстановки, 274
\\{motion}, команда-оператор, 64
\v, ключ, 238, 241
\V, ключ, 238, 241
\x, класс шестнадцатеричных цифр, в регулярных выражениях, 239
]s, команда перехода к следующему ошибочному слову, 354
^[, символ, 230
^J, символ, 232
`., метка, 183
`^, метка, 183
`{letter}, команда, 187
`m, команда, 176
{} (фигурные скобки в регулярных выражениях), 238, 242
~, команда, 230
<, команда, 82
'<, символ, 99
<80>kb, символ, 230
<C-[>, команда перехода в командный режим, 69
<C-]>, команда, 187
<C-]>, команда для навигации по ключевым словам, 314
<C-a>, команда, 59
<C-d>, команда, 108
<C-g>, команда, 150
<C-h>, команда, 68
<C-i>, команда, 180
<C-k>, команда, 95
<C-k>{char1}{char2}, команда, 74
<C-n>, команда, вызов функции автодополнения, 341
<C-o>, команда, 107, 180, 185
<C-p>, команда, вызов функции автодополнения, 341
<C-r>", команда, 203
<C-r>{register}, команда, 71, 95, 203
<C-r>+, команда, 198
<C-r><C-p>{register}, команда, 71
<C-r><C-w>, команда, 111
<C-r>=, команда, регистр выражений, 72
<C-r>0, команда, 71, 203
<C-t>, команда, 315
<C-u>, команда, 68, 95
<C-v>, команда, 80, 95
<C-v>{code}, команда, 73
<C-w>, команда, 68, 95
<C-w>_, команда, 135
<C-w>|, команда, 135
<C-w><C-w>, команда, 135
<C-w>=, команда, 135
<C-w>s, команда, 134
<C-w>T, команда, 138
<C-w>v, команда, 134

<C-w>w, команда, 135
<C-x>, команда, 58, 341
<C-x>s, команда, 359
<Esc>, команда для выхода в командный режим, 92
<Esc>, клавиша
  и избирательность команды отмены, 55
  команда перехода в командный режим, 69
<Leader>, клавиша, роль пространства имен, 164
<Tab>, клавиша, автодополнение путей к файлам, 142
>, команда, 41, 82
'>, символ, 99

A

a", текстовый объект, 171
a, команда, 42
A, команда, 43
a], текстовый объект, 171
Abolish, расширение, 291
ack, программа
  вызов командой :grep, 334
  установка, 334
Ack.vim, расширение, 336
ap, команда перемещения, 63
at, текстовый объект, 171
autocmd, команда, 367
aw, текстовый объект, 57, 174

B

b, команда, 159
bash, командная оболочка, 109
bdw, команда, 57

C

c, флаг, команды подстановки, 273
c{motion}, команда, 172
c3w, команда, 201
commentary, расширение, 64
ctags, программа, 308
  автоматический запуск, 313
  вызов вручную, 312
  дополнительные инструменты, 309
  индексирование проектов, 309
  навигация по ключевым словам, 314
  настройка Vim, 312
  установка, 308
cw, команда, 50

D

d{motion}, команда, 63
dap, команда, 202
daw, команда, 57, 166
db, команда, 56
dbx, команда, 57
dd, команда, 40, 191, 232
ddp, команда, 192
de, команда, 201
diw, команда и неименованный регистр, 193
Doctor JS, проект, 309
dw, команда, 57

E

e, команда, 159
e, флаг, команды подстановки, 273
em, редактор, 94
en, редактор, 94
errorformat, параметр настройки, 328
essential.vim, файл, 361
ex, редактор, 92
expandtab
  настройка, 80
  параметр настройки, 364
exuberant ctags, программа, установка, 308

F

f{char}, команда, 46, 162
F{char}, команда, 165
FileType, событие, 367
ftplugin, расширение, 367
fugitive.vim, расширение, 336

G

g, флаг, команды подстановки, 273
g&, команда, 283
g, команда, 183
g;, команда, 183
g~, команда, 63
ga, команда, 73
ge, команда, 159
gf, команда, 184
gi, команда, 183
gj, команда, 158
gk, команда, 158
gP, команда, 204
gr{char}, команда, 76
grep, программа
  ack программа, как
  альтернатива, 334
  вызов из командной строки, 332
  вызов, не покидая Vim, 332
  настройка, 333
  поиск с учетом регистра
  символов, 333
  происхождение термина, 299
grepformat, параметр настройки, 333
grepprg, параметр настройки, 333
gt, команда, 138
gT, команда, 138
gu, команда, 63
gU, команда, 63
gU{motion}, команда, 85
gUfl, команда, 264
gUtD, команда, 263
gv, команда, 80, 200, 284

H

history, параметр настройки, 112
hlsearch, параметр настройки, 256

I

i/, текстовый объект, 265
i}, текстовый объект, 170
i>, текстовый объект, 171
ICCF Holland, фонд, 362
ignorecase, параметр
настройки, 236, 363
  и функция автодополнения, 340
include, параметр настройки, 345
incsearch, параметр настройки, 257
infercase, параметр настройки
и функция автодополнения, 340
it, текстовый объект, 171
iw, текстовый объект, 174

J, L

j, команда, 158
jsctags, программа, 309
let, ключевое слово, 228

M

M, ключ стандартного режима
поиска, 240
m{letter}, команда, 187
make, команда, 319, 321
makeprg, параметр настройки, 327
matchit, расширение, 178
  загрузка при запуске Vim, 178
mm, команда, 176

N

n, команда, 49, 254
N, команда, 254
n, флаг, команды подстановки, 273
netrw, расширение, 140, 146
  дополнительные
  возможности, 149
nodelint, программа, 326
nohlsearch, параметр настройки, 256
number, параметр настройки, 365
  для нумерации строк, 157

O, P, Q, R

o, команда, 81
p, команда, 191
path, параметр настройки, 144, 185
q, команда записи макроса, 209
q:, команда, 113
qa, команда, 221

qA, команда, 221
R, команда, 75
r{char}, команда, 76
rails.vim, расширение, 144

S

s, команда, 45
shiftwidth
  настройка, 82
  параметр настройки, 364
smartcase, параметр настройки, 236
softtabstop
  настройка, 82
  параметр настройки, 364
spellfile, параметр настройки, 357
spelllang, параметр настройки, 355
suffixesadd, параметр настройки, 185
surround.vim, расширение, 179

T

t{char}, команда, 165
T{char}, команда, 165
tabstop, параметр настройки, 364
tags
  индексный файл, 310
  создание файла, 312
textobj-entire, расширение, 65
TODO комментарии, выборка
в регистр, 300

U

u, команда, 40, 54
U, команда, 85

V

vi, редактор, 92
vi{, команда, 303
Vim
  изменение настроек на лету, 363
  компиляция кода, 320
  лицензия, 362
  настройка, 363
  настройка для работы
  с программой ctags, 312
  настройка под себя, 360
  настройки по умолчанию, 360
Vim, редактор
  о происхождении редактора Vim
  (и его семейства), 94
  перевод редактора в фоновый
  режим работы, 116
  регистры, 194
  терминология, 194
vimrc, файл, 361
  set nrformats=, 60
visualbell, параметр настройки, 213
vU, команда, 231

W

w, команда, 159
wakeup, программа, 320
wildmode, параметр
настройки, 109, 145
wrap, параметр настройки, 157
wrapscan, параметр настройки, 254

X, Y, Z

x, команда, 40, 191
yiw, команда, 193
yt, команда, 71
yyp, команда, 103, 192
z=, команда, 354
zg, команда, 357
zug, команда, 357
zw, команда, 357
zz, команда, 69

А

Автодополнение, 339
  вызов, 341
  и чувствительность к регистру,
  340
  из словаря, 347
  имен файлов, 349
  исправление орфографических
  ошибок, 358
  источники слов, 344

команд Ex, 108
выбор из нескольких совпадений, 109
контекстное, 351
меню, 342
настройка обобщенного автодополнения, 346
перечень методов, 341
по ключевым словам, 339
целых строк, 348
Автоматическая расстановка меток, 176
Активный каталог, 142
Арифметические операции, в строке замены команды подстановки, 287, 288

Б

Благотворительная лицензия, 362
Блочный визуальный режим, 77, 80
включение, 80
вставка текста, 88
добавление текста после непрямоугольного блока, 90
правка табличных данных, 87
Брэм Муленаар (Bram Moolenaar), 361
Буфер обмена, 190, 205
системный, 198
Буферы
активные, 130
группировка в список аргументов, 126
и вкладки, 137
и файлы, различия, 123
как источник слов для автодополнения, 344
назначение горячих клавиш для обхода списков, 124
открытие файлов, 123
скрытые, 130
обработка при выходе из редактора, 131
удаление, 125

В

Визуальный режим, 77
блока, 77, 80
включение, 80
команда «точка», 83
определение диапазона строк, 98
переключение между подрежимами, 80
переключение свободного конца, 81
поиск текущего выделения, 270
поиск текущего слова, 269
построчный, 77, 80
применение текстовых объектов для выделения фрагментов, 169
управление курсором, 78
Виртуальный режим замены, 75
и символы табуляции, 75
Вкладки, 136
и буферы, 137
использование, 137
переключение между вкладками, 138
переупорядочение, 139
переупорядочение мышью, 139
Внешние программы, вызов, 308
Внешний компилятор,
настройка, 326
вызов из Vim, 320
Возврат каретки, в строке замены команды подстановки, 274
Восьмеричные числа, 60
Вставка
из регистра, 70
макросов в документ, 231
символов по диграфам, 74
символов по числовому коду, 73
Вставки, команда, 190
из регистра, 202
последовательностей символов, 202
системная, 205
строк, 203
Выбор из нескольких совпадений, 109

Выделение, переключение свободного конца, 81
Выполнение команд в оболочке, 115
 перевод редактора в фоновый режим работы, 116
 передача содержимого буфера на вход команды, 117
 сохранение вывода команды в буфере, 117
 фильтрация буфера внешними, командами, 118
Выражения проверок, в регулярных выражениях, 247
Вычисления
 регистр выражений, 72
 в строке замены команды подстановки, 288
Вычитание чисел, 58

Г

Глобальный файл tags, 312
Границы слова, в регулярных выражениях, 244
Границы совпадения, в регулярных выражениях, 246

Д

Дерево каталогов, пример, 140
Десятичные числа, настройка интерпретации в Vim, 60
Диапазоны для команды подстановки, 275
Диграфы, для вставки необычных символов, 74
Добавление
 в конец регистра, 302
 команд в конец макроса, 220
 текста после непрямоугольного блока, 90
Документы, обновление при прокрутке меню автодополнения, 343
Дополнительные регистры, 199

З, И

Запись макросов, 209
Избавьтесь от привычки пользоваться клавишами со стрелками, 157
Избирательность команды отмены, 55
Изменение настроек Vim на лету, 363
Изменения
 закрытие при перемещении курсора в режиме вставки, 55
 повторяемые, 56
Имена файлов, автодополнение, 349
Именованные регистры, 196
Инверсия команды :global, 297
Индексные файлы как источник слов для функции автодополнения, 345
История команд, шаблон поиска в истории, 279
История
 отмен и счетчики, 62
 поиска, 255, 266
  и команда :vimgrep, 338
Итеративная разработка шаблонов, 253, 265
Итераторы, применение для нумерации элементов списков, 227

К

Каталоги
 определение списка, 185
 ссылка на текущий рабочий каталог, 349
Клавиатура, 70
Клавиша забоя (Backspace), 68
Клавиши со стрелками
 для перемещения курсора, 155
 избавьтесь от привычки пользоваться, 157
 переназначение, чтобы избавиться от привычки пользоваться, 157

Клавиши управления курсором
  в командном режиме, 95
  в окне режима командной строки, 112
Класс шестнадцатеричных цифр, в регулярных выражениях, 239
Ключевые слова
  адресация командами поиска, 310
  метаданные для адресации, 311
  переход между парными, 178
Коды клавиш, в макросах, 230
Коды символов, для вставки необычных символов, 73
Коллекции файлов, выполнение операций с помощью макросов, 222
Колонки текста, правка, 88
Команда расстановки комментариев, 64
Командная строка, использование grep, 332
Командный режим, 53
  в сравнении с режимом командной строки, 95
  команды проверки орфографии, 355
  сравнение использования счетчиков с возможностью повторения, 61
Команды
  запись последовательностей в виде макросов, 209
  объединение с, командами перемещения, 63
  составные, 44
Команды Ex, 92, 95
  @:, команда, для повторения, 107
  автодополнение, 108
    выбор из нескольких совпадений, 109
  в комбинации с командой :global, 305
  вставка текущего слова в командную строку, 110
  выполнение с помощью команды :global, 297
  диапазоны строк, 97
  для навигации по ключевым словам, 317
  номера строк, 96
  определение диапазонов
    с помощью визуального выделения, 98
    с помощью смещений, 100
    с помощью шаблонов, 99
  повторение, 104, 107
  применение к диапазону строк, 104
  символы для определения диапазонов и смещений, 100
Команды-обертки, 308
Команды-операторы, 64
  собственные, 64
Команды перемещения, 65, 154
  в режиме ожидающего оператора, 154
  для навигации внутри файлов, 154
  объединение с, командами, 63
  собственные, 65
Компиляция кода, не покидая Vim, 320
Конструирование списка файлов, содержащих совпадения с шаблоном, 294
Контекстное автодополнения, 351
Копирование
  и вставка, 190
  макроса из документа в регистр, 231
  строк, 59
  строк текста, 101
Копирования, команда, 190
Копирования, операция
  с помощью функции автодополнения, 348
  строк, 192
Курсор
  местоположение после каждого изменения, 182
  нормализация позиции в макросах, 212

перемещение в конец совпадения, 262
переход к файлу с именем под курсором, 184
положение после выполнения команды :make, 322
смещение в конец совпадения, 260
управление клавишами в основной позиции, 156
установка повторяемыми командами в макросах, 213

Л, М

Лицензия Vim, 362
Макросы, 208
вставка в документ, 231
добавление команд в конец, 220
запись и выполнение, 209, 228
и коллекции файлов, 222
и мышь, 213
и «Формула точки», 215
коды клавиш, 230
копирование из документа в регистр, 231
многократное выполнение со счетчиком, 214
нормализация позиции курсора, 212
параллельное выполнение, 212, 219, 224
повторение изменений в последовательности строк, 216
последовательное выполнение, 211, 218, 225
правка, 231
правка содержимого, 230
прерывание, 213
сохранение изменений во всех файлах, 226
список целевых файлов, 223
установка курсора повторяемыми командами, 213
Максимальное совпадение, 266
Меню, функции автодополнения, 342
Метки, 175
автоматическая расстановка, 176
глобальные, как закладки, 187
глобальные, установка перед погружением в код, 188
последнего изменения, 183
установка, 175
Модальный пользовательский интерфейс, 52
Мышь и макросы, 213

Н

Навигация
внутри файлов, 154
между файлами, 180
по ключевым словам, 314
по списку с результатами, 323
посредством поиска, 167
Направление, поиска, 254
Настройка внешнего компилятора, 326
Настройка Vim, 363
Неименованный регистр, 190, 195

О

Обобщенное автодополнение, настройка, 346
Ожидающего оператора, режим, 65
Окна, 133
Quickfix, 325
закрытие, 135
изменение размеров, 135
организация во вкладках, 136
переключение фокуса вода, 134
переупорядочение, 135
прокрутка содержимого, 69
Окно командной строки, 267
Операторы, 64
собственные, 64
Операции, выполнение с применением текстовых объектов, 172
Опережающие проверки в регулярных выражениях, 247
Основная позиция рук на клавиатуре, 156

Отключение подсветки совпадений, 256
Открытие файлов, 140
Отступы, повторение командой «точка», 82
Очистка, регистра, 301

П

Парные скобки, переход между, 177
Передача содержимого регистра по значению, в команде подстановки, 280
Передача содержимого регистра по ссылке, в команде подстановки, 281
Переключение свободного конца, в визуальном режиме, 81
Перемещение
  по словам, 159
  строк текста, 103
Переназначение
  команд перемещения между строками, 159
  клавиши Caps Lock, 70
Перестановка
  символов, 191
  слов, 201
  строк, 191
Переупорядочение полей в файле CSV, 286
Переход
  между парными ключевыми словами, 178
  между парными скобками, 177
Переходы
  для навигации между файлами, 180
  к первому или последнему в строке, 158
  к файлу с именем под курсором, 184
  между орфографическими ошибками, 354
  по ключевым словам, 314
Печать вслепую, 67
Повторение
  команд Ex, 104, 107
  команд в построчном визуальном режиме, 82
Повторение изменений в последовательности строк, 216
Повторяемые изменения, 56
Повторяемые команды в макросах, 212
Повторяющиеся изменения, макросы, 208
Повторяющиеся слова, поиск, 244
Подсветка орфографических ошибок, 353
Подсветка совпадений, 256
  отключение, 256
Подсчет совпадений, 259
Подтверждение, в команде подстановки, 273
Поиск, 253
  вперед, 248, 254
  вызов истории, 255
  выполнение операций над полным совпадением, 262
  история, 266
  команды поиска в операциях, 168
  команды поиска в режиме ожидающего оператора, 168
  назад, 249, 254
  направление, 254
  основы, 253
  повторение, 254
  предварительный просмотр совпадений, 257
  символов, 162
  с целью навигации, 167
  текущего выделения в визуальном режиме, 270
  текущего слова в визуальном режиме, 269
Поиск с заменой вручную, 48
Построчный визуальный режим, 77, 80
  включение, 80
  повторное выполнение команд, 82

Правка
в режиме вставки, 67
макросов, 231
табличных данных в блочном визуальном режиме, 87
текста в режиме замены, 75
Предварительный обзор, перед подтверждением подстановки, 276
Прерывание макросов, 213
Проверка орфографии, 353
автодополнение, 358
альтернативные словари, 355
подсветка ошибок, 353
специальная терминология, 357
Проверка существования совпадения, 258
Проекты
индексирование с помощью ctags, 309
поиск и замена в нескольких файлах, 292
Прокрутка окна, 69

Р

Рабочее пространство, 133
вкладки, 136
использование, 137
переключение между вкладками, 138
переупорядочение, 139
деление на окна, 133
закрытие окон, 135
изменение размеров окон, 135
переключение фокуса ввода между окнами, 134
переупорядочение окон, 135
Рабочий каталог
как пустая строка между запятыми, 186
Регистр
выделенного фрагмента, 198
выражений, 199, 228
захвата, 196
символов, переключение, 230
«черной дыры», 197
Регистры, 190, 194
TODO комментарии, выборка, 300
адресация, 194
вставка, 202
выражений, 72
добавление в конец, 302
дополнительные, 199
замена выделенного текста, 200
именованные, 196
использование в строке замены команды подстановки, 280
исследование содержимого макроса, 209
неименованный регистр, 190, 195
очистка, 301
передача содержимого по значению, 280
передача содержимого по ссылке, 281
регистр выделенного фрагмента, 198
регистр выражений, 199, 228
регистр захвата, 196
регистр «черной дыры», 197
текстовые, 71
только для чтения, 199
Регулярные выражения
\\v, ключ, 238, 241
\\V, ключ, 238, 241
в командах Ex, 317
выражении проверок, 247
границы слова, 244
границы совпадения, 246
для команд подстановки, 279
квадратные скобки, 238, 242
класс шестнадцатеричных цифр, 239
круглые скобки, 238, 242
опережающие проверки, 247
ретроспективные проверки, 247
совпадение со строкой в кавычках, 265
фигурные скобки, 238, 242
экранирование символов, 238, 248, 249, 250, 251

Режим визуального блока, 77, 80
включение, 80
вставка текста, 88
добавление текста после непрямоугольного блока, 90
правка табличных данных, 87
Режим визуальной строки, 77, 80
включение, 80
Режим вставки, 67
автодополнение, 340
возврат в командный режим, 69
вставка из регистра, 70
командный подрежим, 69
правка, 67
Режим выделения, 79
Режим замены, 67
виртуальный режим замены, 75
и символы табуляции, 75
правка текста, 75
Режим командной строки, 92
в сравнении с командным режимом, 95
выполнение команд в оболочке, 115
история команд, 111
повторный вызов команд, 111
окно режима командной строки, 112
клавиши управления курсором, 112
объединение команд, 113
специальные ключи, 94
Режим экранного редактирования, 94
Режимы ожидающего оператора, 65
Ретроспективные проверки в регулярных выражениях, 247

С

Свойства CSS
контекстное автодополнение, 352
сортировка, 303
Символ табуляции, в строке замены команды подстановки, 274
Символы
вставка необычных символов, 73, 74
переключение регистра, 230
перестановка, 191
переход к первому или последнему в строке, 158
поиск, 162
преобразование в верхний регистр, 84
удаление, 68
Символы завершения шаблона поиска, 252
Системная, команда вставки, 205
Слова
автодополнение из словаря, 347
добавление в орфографический словарь, 357
и СЛОВА, 161
источники для функции автодополнения, 344
перестановка, 201
удаление, 68
Словари
для механизма проверки орфографии, 355
добавление слов, 357
как источник слов для функции автодополнения, 347
Слово, удаление, 57
Сложение чисел, 58
Смещение курсора в конец совпадения, 260
Совпадения
выполнение операций, 262
границы, 246
перемещение курсора в конец, 262
подсветка, 256
подсчет количества, 259
предварительный просмотр, 257
проверка существования, 258
смещение курсора в конец, 260
Сортировка свойств в правилах CSS, 303
Составные команды, 44

Сохранение
изменений во всех файлах, 226
файлов в несуществующие каталоги, 150
Специальная терминология, проверка орфографии, 357
Специальные ключи в режиме командной строки, 94
Специальные символы в строке замены команды подстановки, 274
Списки, нумерация элементов с помощью итератора, 227
Список адресов, 324
Список аргументов, 126
использование, 129
определение файлов по именам, 127
определение файлов с помощью обратных кавычек, 128
определение шаблонов имен файлов, 127
Список буферов, 123
Список изменений, обход, 182
Список переходов, обход, 180
Список результатов, 319
восстановление прежней версии, 326
навигация, 323
Список слов, прокрутка, 342
Список целевых файлов, создание для обработки макросами, 223
Строки
вставка, 203
перестановка, 191
повторение изменений в последовательности с помощью макросов, 216
создание дубликатов, 192
Строки текста
автодополнение, 348
копирование, 59
нумерация, 157
переход к первому или последнему символу, 158
удаление до начала, 68
фактические и экранные, 157
Счетчики
для команд, 58
и история отмен, 62
многократное выполнение макросов, 214
сравнение использования счетчиков с возможностью повторения, 61

Т

Таблицы символов Юникода, 73
Теги HTML, шаблон для команды :global, 300
Текстовые объекты
выделение фрагментов с применением, 169
выполнение операций, 172
категории, 171
связанные, 174
с ограничителями, 174
Текстовые регистры, 71
Текущая строка, адрес, 98
Текущий рабочий каталог, 141
ссылка, 349

У

Удаление
вперед, 57
команды, 68
назад, 56
символов, 68
целого слова, 57
Удаления, команда, 190
Управление курсором в визуальном режиме, 78
Управление файлами, 122
как открыть файл, указав путь относительно активного каталога, 142
как открыть файл, указав путь относительно текущего рабочего каталога, 141
навигация внутри файлов, 154
навигация в файловой системе, 146

открытие файлов, 140
открытие файлов с применением команды :find, 143
поиск файлов по именам, 145
простой способ извлечения имени каталога активного файла, 143
расширение rails.vim, 144
скрытыми, 130
сохранение в несуществующие каталоги, 150
сохранение изменений, 226
Урок истории, наследственность синтаксиса шаблонов в Vim, 240

Ф

Файловая система, навигация, 146
Файлы
глобальные метки
для переключения, 188
и буферы, различия, 122
как источники слов для функции автодополнения, 345
конструирование списка файлов, содержащих совпадения с шаблоном, 294
навигация между файлами, 180
открытие файлов, 140
переход к файлу с именем под курсором, 184
поиск и замена в нескольких файлах, 292
поиск по именам, 145
скрытые, 130
сохранение в несуществующие каталоги, 150
список аргументов, 126
Фильтрация буфера внешними командами, 118
флаги, команды подстановки, 273
формула точки, 51, 104
«Формула точки», 261, 265
Формула точки и макросы, 215

Ч

Числа
с ведущим нулем, 60
сложение и вычитание, 58
Чувствительность к регистру
автоматическое определение, 236
и автодополнение, 340
настройка, 236
шаблоны, 236
Чувствительность к регистру символов и команда grep, 333

Ш, Э

Шаблоны, 235
итеративная разработка, 253, 265
поиска тегов HTML для команды :global, 300
чувствительность к регистру, 236
автоматическое определение, 236
настройка, 236
Экранирование, символов в регулярных выражениях, 238, 248, 249, 250, 251

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Дрю Нейл

Практическое использование Vim

Второе издание

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 36,75. Тираж 100 экз.

Веб-сайт издательства: www.дмк.рф