Нараян Прасти

Блокчейн

Разработка приложений

Разработка децентрализованных приложений в реальном времени на платформе Ethereum

bhv®

Packt›

Building Blockchain Projects

Develop real-time practical DApps using Ethereum and JavaScript

Narayan Prusty

BIRMINGHAM - MUMBAI

Нараян Прасти

Блокчейн
Разработка приложений

Разработка децентрализованных приложений в реальном времени на платформе Ethereum

Санкт-Петербург
«БХВ-Петербург»
2018

УДК 004.75
ББК 32.973.26-018
П70

Прасти Н.

П70 Блокчейн. Разработка приложений: Пер. с англ. — СПб.: БХВ-Петербург, 2018. — 256 с.: ил.

ISBN 978-5-9775-3976-0

Рассказано о том, что такое децентрализованные приложения и как они работают. Рассмотрены принципы работы платформы Ethereum. Показано, как писать смарт-контракты и использовать интерактивную консоль Geth для размещения и передачи транзакций. Описана библиотека web3.js, ее импорт, подключение к Geth и использование в среде Node.js или на стороне клиента. Продемонстрировано, как создать сервис кошелька и управлять им, как компилировать смарт-контракты и развертывать их при помощи web3.js и EthereumJS. Описаны язык программирования Solidity и среда разработки Truffle. Приведено руководство по разработке собственного блокчейна и децентрализованных приложений корпоративного уровня.

Для программистов, преподавателей и студентов,
а также специалистов отделов развития компаний и банков

УДК 004.75
ББК 32.973.26-018

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Капалыгина*
Перевод с английского	*Валерия Яценкова*
Компьютерная верстка	*Ольги Сергиенко*
Оформление обложки	*Марины Дамбиевой*

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-1-78712-214-7 (англ.)
ISBN 978-5-9775-3976-0 (рус.)

Оглавление

Предисловие к русскому изданию **11**

Об авторе **13**

О рецензентах **15**

Издательство «Packt» **17**

Предисловие **19**
О чем эта книга? 19
Что понадобится в дополнение к этой книге? 20
Для кого эта книга? 20
Обозначения 20
Обратная связь 21
Поддержка читателей 22
Скачивание исходных кодов программ 22
Цветные иллюстрации 22
Электронный архив файлов для русского издания 23
Исправления 23
Пиратство 23
Вопросы 23

Глава 1. Децентрализованные приложения **25**
Что такое ДП? 25
Достоинства децентрализованных приложений 27
Недостатки децентрализованных приложений 27
Децентрализованная автономная организация 27
Личность пользователя в ДП 28
Учетные записи пользователей 30
Доступ к централизованным приложениям 30
Внутренняя валюта приложений 31
Недостатки внутренней валюты 32

Приложения с контролируемым доступом 32
Популярные приложения 32
Bitcoin 32
Что такое регистр? 33
Что такое блокчейн? 33
Легален ли биткойн? 33
Почему мы используем биткойн? 34
Ethereum 34
Hyperledger 35
IPFS 35
Как работает IPFS? 36
Filecoin 36
Namecoin 37
Домены в зоне *.bit* 37
Dash 38
Децентрализованное управление и бюджетирование 39
Децентрализованные услуги 39
BigChainDB 40
OpenBazaar 40
Ripple 40
Заключение 43

Глава 2. Принципы работы Ethereum 44
Знакомство с Ethereum 44
Учетная запись Ethereum 45
Транзакции 45
Консенсус 46
Метка времени 48
Число nonce 48
Время блока 49
Ветвление 51
Генезис 52
Деноминация эфира 52
Виртуальная машина Ethereum 53
Газ 53
Обнаружение узлов 54
Протоколы Whisper и Swarm 55
Geth 55
Установка Geth 56
OS X 56
Ubuntu 56
Windows 56
JSON-RPC и консоль JavaScript 57
Подкоманды и опции 57
Подключение к сети *mainnet* 57
Создание частной сети 57
Создание аккаунта 58
Майнинг 58
Быстрая синхронизация 59

Ethereum Wallet ... 59
Mist ... 61
Уязвимости Ethereum ... 62
Атака Сибиллы ... 62
Атака 51% ... 62
Обновление Serenity ... 62
Платежные каналы и каналы состояния ... 63
Протокол консенсуса Casper ... 63
Разделение данных ... 64
Заключение ... 64

Глава 3. Разработка смарт-контрактов ... 65
Файлы исходного кода Solidity ... 65
Структура смарт-контракта ... 66
Расположение данных ... 67
Что такое типы данных? ... 68
Массивы ... 69
Строки ... 70
Структуры ... 71
Перечисление ... 72
Сопоставление ... 72
Оператор *delete* ... 73
Преобразование элементарных типов ... 74
Ключевое слово *var* ... 74
Управляющие структуры ... 75
Оператор *new* и создание контракта ... 76
Исключения ... 77
Вызов внешних функций ... 77
Свойства контракта ... 79
Видимость ... 79
Модификаторы ... 81
Резервная функция ... 83
Наследование ... 83
Ключевое слово *super* ... 85
Абстрактные контракты ... 86
Библиотеки ... 86
Конструкция *using ... for ...* ... 88
Возврат нескольких значений ... 89
Импорт файлов исходных кодов Solidity ... 89
Глобальные переменные ... 90
Свойства блока и транзакции ... 90
Свойства, связанные с адресом ... 91
Переменные, связанные с контрактом ... 91
Единицы эфира ... 91
Доказательство наличия, целостности и принадлежности файла ... 91
Компиляция и развертывание контракта ... 93
Заключение ... 96

Глава 4. Учимся работать с web3.js 97
Введение в web3.js 97
Импортирование web3.js 98
Подключение к узлу 98
Структура API 99
Библиотека BigNumber.js 100
Конвертация денежных единиц 101
Запрос цены газа, баланса и деталей транзакции 101
Отправка эфира 103
Работа с контрактами 104
Отслеживание событий контракта 106
Разработка клиентского приложения для контракта 109
Структура проекта 110
Разработка серверной части 110
Разработка клиентской части 112
Тестирование клиентской части 116
Заключение 119

Глава 5. Разработка сервиса кошелька 120
Различие между онлайн- и оффлайн-кошельками 120
Библиотеки hooked-web3-provider и ethereumjs-tx 121
Что такое HD-кошелек? 124
Введение в функции формирования ключа 125
Знакомство с LightWallet 126
Путь вывода HD-кошелька 127
Разработка сервиса кошелька 127
Предварительная подготовка 127
Структура проекта 128
Разработка серверной части 128
Разработка клиентской части 129
Тестирование 136
Заключение 142

Глава 6. Разработка платформы для смарт-контрактов 143
Вычисление *nonce* для транзакции 144
Знакомство с solcjs 145
Установка solcjs 145
API solcjs 146
Использование различных версий компилятора 147
Связывание библиотек 148
Обновление ABI 149
Разработка платформы для развертывания контрактов 149
Структура проекта 150
Разработка серверной части 150
Разработка клиентской части 156
Тестирование 160
Заключение 161

Глава 7. Приложение для ставок на результат матча 162
Знакомство с Oraclize 163
Как работает Oraclize? 163
Источники данных 163
Доказательство подлинности 164
Стоимость услуг Oraclize 166
Основы работы с API Oraclize 167
Настройка типа и места хранения доказательства 167
Отправка запросов 167
Отложенные запросы 168
Расходование газа 168
Функции обратного вызова 169
Синтаксический разбор результатов 170
Получение цены запроса 171
Шифрование запросов 171
Расшифровка источника данных 171
IDE Oraclize 172
Работа со строками 172
Разработка контракта для ставок на спорт 174
Разработка приложения для ставок 177
Разработка структуры приложения 178
Разработка серверной части 178
Разработка клиентской части 181
Тестирование приложения 189
Заключение 193

Глава 8. Разработка смарт-контрактов уровня предприятия 194
Знакомство с *ethereumjs-testrpc* 195
Установка и использование *ethereumjs-testrpc* 195
Приложение командной строки *testrpc* 195
Использование *ethereumjs-testrpc* в качестве провайдера *web3* или HTTP-сервера 197
Доступные методы RPC 198
Что такое заголовки событий? 199
Знакомство с пакетом *truffle-contract* 201
Установка и импорт *truffle-contract* 202
Настройка тестового окружения 203
API *truffle-contract* 203
API абстракции контракта 204
Создание экземпляра контракта 209
API экземпляра контракта 211
Введение в Truffle 212
Установка Truffle 212
Инициализация Truffle 212
Компиляция контрактов 214
Файлы конфигурации 214
Развертывание контрактов 215
Файлы переноса 216
Написание кода переноса 216

Юнит-тесты контрактов 218
Написание тестов на JavaScript 219
Написание тестов на Solidity 221
Как перевести валюту на тестовый контракт? 224
Запуск тестов 224
Управление пакетами 225
Управление пакетами через NPM 225
Управление пакетами через EthPM 225
Использование контрактов из пакета 227
Использование артефактов пакета в коде JavaScript 227
Доступ к адресам развернутых контрактов пакета в Solidity 227
Работа с консолью Truffle 228
Запуск внешних скриптов в контексте Truffle 229
Создание клиента в Truffle 229
Запуск внешних команд 230
Запуск пользовательских функций 230
Конструктор Truffle по умолчанию 231
Создание клиента 233
Сервер Truffle 236
Заключение 238

Глава 9. Разработка блокчейна для консорциума 239
Что такое блокчейн консорциума? 240
Что такое консенсус с доказательством полномочий? 240
Введение в Parity 241
Принципы работы Aura 241
Начинаем работу с Parity 243
Установка Rust 243
Скачивание, установка и запуск Parity 244
Создание частной сети 244
Создание аккаунтов 244
Создание файла спецификации 244
Запуск узлов 248
Подключение узлов 249
Полномочия и приватность 249
Заключение 250

Приложение. Описание электронного файлового архива 251

Предметный указатель 253

Предисловие к русскому изданию

Уважаемые читатели!

Нам повезло жить в интересное время. На наших глазах и при нашем участии рождаются новые технологии, которые изменят мир: блокчейн и децентрализованные приложения. Следует понимать, что сам по себе блокчейн ничего не значит. Революционные перемены зависят лишь от того, каким способом и в каких областях мы будем его применять. Книга, которую вы держите в руках, — один из редких образцов руководства по практическому применению технологии блокчейна.

Блокчейн вошел в нашу жизнь совсем недавно. Для этой технологии пока нет общепринятых законов и правил. Не удивительно, что возникают причудливые сочетания языков программирования и сред разработки, буквально в течение года успевают родиться и умереть новые протоколы и сервисы. Но уже настало время делиться опытом и говорить об устоявшихся подходах. Автор этой книги, Нараян Прасти, знает технологию блокчейна изнутри, потому что работает предметным экспертом по блокчейну в Национальном банке Арабских Эмиратов и последние пять лет занимается прикладной разработкой децентрализованных приложений.

Для продуктивной работы с материалом этой книги вам потребуются базовые знания и навыки в области программирования:

- основы объектно-ориентированного и функционального программирования. Следует знать, что такое объекты, классы, методы, конструкторы, функции. Полезно иметь навыки программирования на языке C#;
- навык программирования на языке JavaScript. Это обязательное условие для продуктивной работы с материалом книги. Программированию на JavaScript посвящено много хороших книг, и для вас не составит большого труда получить необходимые знания;
- умение работать со средой разработки и выполнения скриптов Node.js. Вы можете приступить к освоению Node.js параллельно с чтением двух первых глав этой книги.

При подготовке перевода мы тщательно проверили доступность всех ссылок, номера версий программ и обновили их по мере необходимости. В сносках указаны дополнительные ссылки и примечания, которые помогут в изучении материала. В сопровождающем книгу файловом архиве вы найдете исходные коды упражнений к главам книги, предоставленные издательством «Packt», и листинги исходных кодов примеров программ из текста.

Желаем успеха в практическом освоении новейшей технологии!

Переводчик и научный редактор русского издания
Валерий Яценков

Об авторе

Нараян Прасти (Narayan Prusty) — разработчик полного цикла, который в течение последних пяти лет специализируется на технологиях блокчейн и JavaScript. С присущей ему целеустремленностью он разрабатывал масштабируемые приложения для стартапов, правительства и предприятий в Индии, Сингапуре, США и ОАЭ.

Сегодня Нараян Прасти регулярно создает децентрализованные приложения на основе Ethereum, Bitcoin, Hyperledger, IPFS, Ripple и других протоколов. Он работает предметным экспертом по технологии блокчейн в Национальном банке Арабских Эмиратов, Дубаи.

Нараян Прасти уже написал две книги: Learning ECMAScript 6 и Modern JavaScript Applications. Обе они опубликованы издательством «Packt»[1].

Он немедленно приступает к работе, если видит интересную захватывающую проблему. В возрасте 18 лет он разработал поисковый движок для файлов MP3 и до сегодняшнего дня создал много различных приложений, которыми пользуются люди по всей Земле. Его отличает уникальное умение разрабатывать масштабные проекты от начала до конца.

Сейчас Нараян видит свое предназначение в том, чтобы делать многие вещи проще, быстрее и дешевле при помощи технологии блокчейн. Он также ищет возможности предотвратить коррупцию, мошенничество и добивается открытости во всем мире при помощи блокчейн-приложений.

Вы можете больше узнать о авторе в его блоге: **http://qnimate.com** и найти его в сообществе LinkedIn: **https://www.linkedin.com/in/narayanprusty/**.

---

[1] На русском языке издана одна из этих книг: Введение в ECMAScript 6. М.: ДМК Пресс, 2016. — *Здесь и далее примечания переводчика, если не указано иное.*

О рецензентах

Имран Башир (Imran Bashir) получил степень магистра информационной безопасности в Королевском колледже Холлоуэй (Royal Holloway) при Лондонском университете и обладает опытом разработки программного обеспечения, управления инфраструктурой и ИТ-службами. Он входит в сообщества Института инженеров по электротехнике и электронике (Institute of Electrical and Electronics Engineers, IEEE) и Британского компьютерного общества (British Computer Society, BSC). Имран имеет 16-летний опыт работы в государственном и финансовом секторах. Прежде чем перейти в индустрию финансовых услуг, он работал над крупными ИТ-проектами для государственного сектора и занимал разные технические должности в Лондоне — экономической столице Европы. Сейчас он вице-президент по технологиям в одном из инвестиционных банков Лондона.

Дэниел Крафт (Daniel Kraft) изучал математику и физику и получил степень доктора прикладной математики в Университете Граца, Австрия. Он подключился к разработке криптовалют в 2013-м году, был ведущим разработчиком и научным руководителем проектов Namecoin и Huntercoin в 2014-м году и опубликовал две исследовательских статьи в рецензируемых журналах. Дэниел работает инженером-программистом и является соучредителем компании Crypto Realities Ltd, которая создает многопользовательские игровые миры на основе технологии блокчейн.

Гуран Торвекар (Gaurang Torvekar) получил степень доктора по информационным системам в Сингапурском университете управления. Он соучредитель и технический директор компании Attores, расположенной в Сингапуре и обслуживающей смарт-контракты. Гуран имеет богатый опыт разработки приложений для Ethereum и Hyperledger. Он был докладчиком на нескольких конференциях по блокчейну, провел множество курсов по технологии блокчейн в Политехническом университете Сингапура и является ментором по блокчейну хакатона[1] Angelhack.

---

[1] Хакатон — форум разработчиков, во время которого специалисты из разных областей разработки программного обеспечения (программисты, дизайнеры, менеджеры) сообща работают над решением какой-либо проблемы. — *Ред.*

Издательство «Packt»

Знаете ли вы, что издательство «Packt» для каждой опубликованной книги предлагает электронные версии в форматах PDF и ePub? Купив печатную книгу, вы сможете на сайте **www.packtpub.com** получить ее электронное издание со скидкой для владельцев печатной версии[1]. С вопросами обращайтесь по адресу; **customercare@packtpub.com**.

На сайте **www.packtpub.com** вы также можете найти множество бесплатных технических публикаций, подписаться на бесплатные рассылки и получать уникальные скидки и предложения.

Наиболее востребованные навыки разработки программного обеспечения вы можете получить на странице «Mapt» сайта издательства «Packt» по адресу: **https://www.packtpub.com/mapt**.

«Mapt» предоставляет полный доступ ко всем книгам и видеокурсам издательства «Packt», а также передовые инструменты, которые помогут вам в личном развитии и карьерном росте.

Что дает подписка?

- Полнотекстовый поиск по всем книгам издательства «Packt».
- Возможность ставить закладки, копировать и распечатывать фрагменты текста из книг.
- Печать книг по запросу и доступ к ним при помощи браузера.

[1] Разумеется, это относится только к покупателям исходного, английского издания книги. — *Ред.*

Предисловие

Блокчейн — это децентрализованный регистр, который содержит постоянно растущий список записей, защищенных от подделки и внесения изменений. Каждый пользователь может подключаться к сети, отправлять в нее новые транзакции, проверять транзакции и создавать новые блоки.

В этой книге рассказано, что такое блокчейн, как он обеспечивает целостность данных, и как создавать прикладные блокчейн-проекты на платформе Ethereum. На примере интересных реальных проектов вы узнаете, как писать смарт-контракты, которые выполняются именно так, как запрограммировано, без малейших шансов на мошенничество, цензуру или вмешательство третьей стороны, а также создавать приложения со сквозным шифрованием для блокчейна. Вы изучите такие основы, как криптография в криптовалютах, майнинг, смарт-контракты и язык программирования Solidity.

Блокчейн стал главным техническим новшеством биткойна, для которого служит публичным регистром транзакций.

О чем эта книга?

- **Глава 1. Децентрализованные приложения** — рассказывает о том, что такое децентрализованные приложения и как они работают.
- **Глава 2. Принципы работы Ethereum** — рассказывает о том, как работает Ethereum.
- **Глава 3. Разработка смарт-контрактов** — показывает, как писать смарт-контракты и использовать интерактивную консоль Geth для размещения и передачи транзакций.
- **Глава 4. Учимся работать с web3.js** — знакомит с библиотекой web3.js, рассказывает о том, как импортировать библиотеку, подключить ее к Geth и использовать в среде Node.js или на стороне клиента.

- **Глава 5. Разработка сервиса кошелька** — демонстрирует, как создать сервис кошелька и легко управлять Ethereum Wallets даже в режиме оффлайн. Мы воспользуемся для этого библиотекой LightWallet.
- **Глава 6. Разработка платформы для смарт-контрактов** — рассказывает о том, как компилировать смарт-контракты при помощи web3.js и развертывать их при помощи web3.js и EthereumJS.
- **Глава 7. Приложение для ставок на результат матча** — поясняет, как можно использовать сервис Oraclize для выполнения HTTP-запросов из смарт-контрактов Ethereum, чтобы получить доступ к данным в Интернете. Мы также узнаем, как получать доступ к файлам, сохраненным в IPFS, научимся использовать строковую библиотеку и получим другие полезные навыки.
- **Глава 8. Разработка смарт-контрактов уровня предприятия** — подробно рассказывает, как использовать среду Truffle, которая значительно упрощает разработку децентрализованных приложений корпоративного уровня.
- **Глава 9. Разработка блокчейна для консорциума** — рассказывает о специфических особенностях разработки блокчейна для консорциумов.

Что понадобится в дополнение к этой книге?

Вам понадобится персональный компьютер с операционной системой Windows 7 SP1+, 8, 10 или Mac OS X 10.8+.

Для кого эта книга?

Эта книга предназначена для JavaScript-разработчиков, которые хотят создавать защищенные приложения для управления транзакциями и данными на основе блокчейна и Ethereum. Читатели, которые интересуются криптовалютами и доверенными хранилищами данных, найдут эту книгу чрезвычайно полезной.

Обозначения

В этой книге вы найдете несколько вариантов оформления текста, которые соответствуют различным типам информации. Продемонстрируем несколько стилей текста и поясним, что они обозначают.

Отдельные директивы программного кода, имена таблиц баз данных, имена файлов, расширения имен файлов, вводимые пользователем данные и строки Twitter обозначены шрифтом `Courier`. Например:

> Затем запустите приложение при помощи команды `node app.js` внутри каталога `Final`.

Фрагменты кода выглядят так:

```
var solc = require("solc");
var input = "contract x { function g() {} }";
var output = solc.compile(input, 1); // 1 activates the optimizer
for (var contractName in output.contracts) {
// logging code and ABI
console.log(contractName + ": " +
output.contracts[contractName].bytecode);
console.log(contractName + "; " +
JSON.parse(output.contracts[contractName].interface));
}
```

Текст, вводимый и выводимый в командной строке терминальных программ, обозначен **полужирным шрифтом Courier**, например:

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
  /etc/asterisk/cdr_mysql.conf
```

Надписи на кнопках и элементах меню программ обозначены **полужирным шрифтом**, например:

Снова выберите тот же самый файл и нажмите кнопку **Get Info**.

Таким значком обозначены важные примечания и комментарии.

Таким значком обозначены полезные советы и подсказки.

Обратная связь

Мы всегда благодарны читателям за отзывы. Расскажите нам, что вы думаете об этой книге, что вам понравилось или не понравилось. Отзывы читателей помогают нам готовить издания, которые действительно будут для вас полезны.

Для отправки отзыва общего плана достаточно написать нам по адресу электронной почты **feedback@packtpub.com**, указав название книги в теме письма.

Если вы хорошо разбираетесь в какой-либо теме и хотели бы написать книгу или стать соавтором, прочтите руководство для авторов: **www.packtpub.com/authors**.

Для читателей русского издания

Читатели русского издания могут оставлять свои отзывы на странице книги на сайте издательства «БХВ-Петербург» по адресу: **www.bhv.ru** или писать в издательство по адресу электронной почты: **mail@bhv.ru**.

Поддержка читателей

Поскольку вы стали правомочным обладателем книги издательства «Packt», мы поможем вам извлечь максимальную пользу из покупки.

Скачивание исходных кодов программ

Вы можете скачать исходные коды программ после регистрации на сайте **www.packtpub.com**. Независимо от места приобретения книги, зарегистрируйтесь по адресу: **https://www.packtpub.com/books/content/support** и получите файлы непосредственно на свою электронную почту[1].

Для скачивания исходных кодов с сайта издательства «Packt» выполните следующие шаги:

1. Войдите под своим именем или зарегистрируйтесь на сайте.
2. Наведите указатель мыши на закладку **SUPPORT** в верхней части сайта.
3. Щелкните на пункте **Code Download & Errata**.
4. Введите название книги или часть названия в поле **Search**.
5. Выберите нужную книгу в результатах поиска.
6. Выберите в раскрывающемся поле место покупки книги.
7. Щелкните на ссылке **Code Download**, которая появится ниже этого поля.

После скачивания воспользуйтесь одним из следующих архиваторов для извлечения файлов из архива:

- WinRAR или 7-ZIP — для Windows;
- Zipeg, iZip или UnRarX — для Mac;
- 7-Zip или PeaZip — для Linux.

Набор исходных кодов для этой книги также хранится на сайте GitHub по адресу: **https://github.com/PacktPublishing/Building-Blockchain-Projects**.

У издательства «Packt» есть и другие пакеты исходных кодов для обширного каталога их книг и видеокурсов. Они доступны по адресу:

https://github.com/PacktPublishing/.

Цветные иллюстрации

Мы также предоставляем вам PDF-файл с цветными изображениями скриншотов и диаграмм, приведенных в этой книге. Цветные изображения помогут вам лучше понять, что имеется в виду. Вы можете скачать этот файл по ссылке:

https://www.packtpub.com/sites/default/files/downloads/BuildingBlockchainProjects_ColorImages.pdf.

[1] Напомним, что эти предложения относятся только к покупателям исходного, английского издания книги. — *Ред.*

Электронный архив файлов для русского издания

Электронный архив с материалами к этой книге можно скачать с FTP-сервера издательства «БХВ-Петербург» по ссылке **ftp://ftp.bhv.ru/9785977539760.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение*).

Исправления

Несмотря на все усилия и аккуратную работу над книгами, ошибки все-таки иногда проникают в текст. Если вы нашли ошибку в одной из наших книг — в тексте или в программе — мы будем признательны за сообщение о ней. Сделав это, вы убережете других читателей от огорчения и поможете нам улучшить следующее издание книги. Если вы нашли ошибку, пожалуйста, сообщите нам, зайдя на сайт по адресу: **http://www.packtpub.com/submit-errata**. Выберите вашу книгу, щелкните на ссылке: **Errata Submission Form** и введите текст своего замечания. После проверки ваша поправка будет одобрена и размещена на сайте в разделе для соответствующей книги.

Для просмотра ранее внесенных поправок перейдите по адресу: **www.packtpub.com/books/content/support** и введите название книги в поле поиска. Информация о правках размещена под заголовком **Errata**.

Пиратство

Хищение авторских материалов в Интернете стало общей проблемой для всех средств массовой информации. В издательстве «Packt» очень серьезно относятся к защите своих авторских прав и лицензий. Если вы обнаружили незаконную копию одного из наших изданий в любой форме, пожалуйста, незамедлительно сообщите нам физический адрес этого места или адрес веб-сайта, чтобы мы смогли принять меры.

Связаться с нами для отправки ссылки на нелегальные материалы можно по адресу электронной почты: **copyright@packtpub.com**.

Мы благодарны вам за помощь в защите наших авторов и наших усилий по разработке полезных материалов, которые мы создаем для вас.

Вопросы

Если у вас возникли затруднения с любыми аспектами использования этой книги, обращайтесь по адресу: **questions@packtpub.com**, и мы постараемся переадресовать ваш вопрос для наилучшего решения проблемы.

Читатели русского перевода книги могут обращаться с вопросами и пожеланиями по адресу издательства «БХВ-Петербург»: **mail@bhv.ru**.

1

Децентрализованные приложения

Почти все интернет-приложения, с которыми мы имели дело, являются *централизованными*, то есть серверы такого приложения принадлежат определенной компании или лицу. В течение длительного времени разработчики создавали централизованные приложения, а пользователи их применяли. Но у такого подхода есть проблемы, мешающие создавать определенные типы приложений. Централизованные приложения менее прозрачны, имеют конкретную точку отказа, не могут противостоять сетевой цензуре и т. д. На фоне этих проблем возникла новая технология разработки интернет-приложений, которые называются *децентрализованными приложениями* (Decentralized Applications, DApps).

В этой главе мы будем говорить о децентрализованных приложениях (ДП) и рассмотрим следующие темы:

- что такое ДП?
- в чем разница между децентрализованными, централизованными и распределенными приложениями?
- преимущества и недостатки централизованных и децентрализованных приложений;
- обзор структуры данных, алгоритмов и протоколов, применяемых в наиболее популярных ДП;
- некоторые популярные ДП, выстроенные поверх других децентрализованных приложений.

Что такое ДП?

Децентрализованные приложения — это особая разновидность интернет-приложений, основанных на одноранговой сети (peer-to-peer network) и имеющих открытый исходный код. Ни один узел сети не имеет полного контроля над ДП.

Структура и способ хранения данных ДП зависят от его функционального назначения. Например, Bitcoin[1] использует структуру данных в виде блокчейна.

Узлом одноранговой сети может стать любой компьютер, подключенный к Интернету. Это создает серьезную проблему, связанную с необходимостью обнаруживать и блокировать узлы, вносящие ошибочные изменения в данные и распространяющие недостоверную информацию остальным узлам. Следовательно, мы должны прийти к соглашению между узлами относительно того, являются ли достоверными данные, опубликованные определенным узлом. В структуре ДП нет главного сервера, который координирует узлы и принимает решение о достоверности данных. Для решения этой сложной задачи предназначены особые *протоколы консенсуса*. Протоколы консенсуса разрабатывают под конкретную структуру данных ДП. Например, Bitcoin для достижения консенсуса использует *протокол доказательства работы* (proof-of-work protocol, PoW).

Каждое ДП имеет клиентскую часть (программа-клиент), с которой работает пользователь. Чтобы получить доступ к ДП, мы должны стать узлом одноранговой сети и запустить на своем компьютере специальный сервер узла, а затем подключить к этому серверу клиент. Узлы приложения предоставляют только API (Application Programming Interface, интерфейс прикладного программирования) и позволяют сообществу разработчиков создавать различные клиентские программы. Некоторые разработчики ДП предоставляют пользователям официальную версию клиента. Клиенты должны иметь открытый исходный код и быть доступными для скачивания без ограничений. В противном случае вся идея децентрализации теряет смысл.

Однако клиент-серверная архитектура может оказаться трудной для установки, особенно если пользователь не имеет навыков разработчика. Поэтому обычно клиент и/или сервер просто запускаются как службы.

Что такое распределенное приложение?

Распределенные приложения работают на множестве серверов одновременно. Это необходимо при наличии большого сетевого трафика и объема данных, особенно если отказ в обслуживании приложения недопустим. В распределенных приложениях данные реплицируются между разными серверами, чтобы обеспечить высокую доступность данных. Централизованные приложения могут быть распределенными, а могут и не быть таковыми. Но децентрализованные приложения всегда распределенные. Например, Google, Facebook, Slack, Dropbox и им подобные являются распределенными, а простой сайт с портфолио или персональный блог — нет, пока трафик не слишком большой.

[1] Следует различать приложение Bitcoin и криптовалюту биткойн. В этой главе автор говорит о децентрализованном приложении Bitcoin.

Достоинства децентрализованных приложений

Приведем некоторые достоинства ДП:

- устойчивость к отказам, поскольку уязвимая точка отказа отсутствует по определению;
- неуязвимость от сетевой цензуры, поскольку нет центрального органа, на который правительство могло бы надавить и потребовать удалить данные. Правительство не может даже заблокировать домен или IP-адреса приложения, поскольку ДП не нуждается в конкретных адресах или доменах. Конечно же, правительство может отследить отдельного пользователя сети по IP-адресу и отключить его. Но если одноранговая сеть достаточно велика, то полное отключение приложения становится непосильной задачей, особенно если узлы разбросаны по разным странам;
- пользователям легче доверять приложению, которое не управляется одним руководящим органом, способным обманывать пользователей для своей выгоды.

Недостатки децентрализованных приложений

Разумеется, каждая система кроме достоинств имеет и недостатки. Рассмотрим некоторые недостатки ДП:

- трудности с обновлением и устранением ошибок. Приложение должно быть обновлено на каждом узле одноранговой сети;
- иногда приложения требуют подтвердить *личность пользователя* (user identity). Поскольку нет центрального органа, заверяющего личность пользователя, разработка некоторых ДП превращается в серьезную проблему;
- трудность разработки — необходимо применять очень сложные протоколы достижения консенсуса и предусмотреть масштабирование с самого начала. Иначе говоря, мы не можем наспех реализовать идею, надеясь когда-нибудь потом добавлять новые функции и расширять приложение;
- обычно приложения не нуждаются в сторонних API для сохранения или получения данных. Ваши ДП не должны зависеть от API централизованных приложений, но при этом они могут зависеть от других ДП. В теории это звучит неплохо, но с большим трудом реализуется на практике.

Децентрализованная автономная организация

Мы привыкли к тому, что организацию олицетворяет наличие подписанных документов, и правительство может на нее влиять. В зависимости от типа организации, она может иметь или не иметь акционеров.

Децентрализованная автономная организация (Decentralized Autonomous Organization, DAO, ДАО) — это организация, которая представлена обществу компьютерной программой (то есть действует в соответствии с правилами, заложенными в программу), полностью прозрачна, полностью подконтрольна акционерам и не зависит от правительства.

Для достижения этих целей мы должны разрабатывать ДАО как децентрализованное приложение. Поэтому можно сказать, что ДАО является подклассом ДП.

Что такое децентрализованная автономная корпорация (ДАК, DAC)?
Между понятиями ДАО и ДАК нет явного различия. Одни утверждают, что это одно и то же, другие считают, что ДАК — это разновидность ДАО, созданная специально для получения прибыли акционерами.

Личность пользователя в ДП

Одним из основных преимуществ ДП является гарантия анонимности пользователя. Но многие приложения нуждаются в проверке личности пользователя перед началом работы. Поскольку центрального органа верификации не существует, подтверждение личности пользователя становится проблемой.

В централизованных приложениях человек подтверждает свою личность отправкой отсканированных документов, звонком по телефону и т. д. Этот процесс получил название «знай своего клиента» (Know Your Customer, KYC). Но в структуре ДП нет человека, который бы проверял личность. Поэтому приложение вынуждено делать это самостоятельно. Очевидно, ДП не может понять и проверить отсканированные документы[2] или отправить SMS. Поэтому мы должны предоставить приложению цифровой идентификатор, который ДП сможет понять и проверить. Основная проблема сегодня заключается в том, что не каждое ДП работает с цифровой идентификацией, и лишь редкие пользователи знают, как получить цифровой идентификатор.

Существуют различные формы цифровых идентификаторов. На сегодняшний день наиболее популярной и рекомендованной формой является *цифровой сертификат*, который также называют *сертификатом открытого ключа* (public key certificate) или *удостоверением личности* (identity certificate). Это электронный документ, устанавливающий право собственности на открытый ключ. В общем случае пользователь имеет закрытый ключ, открытый ключ и цифровой сертификат. Пользователь хранит закрытый ключ в тайне и никому его не показывает. Открытый ключ может быть показан кому угодно. Цифровой сертификат содержит публичный ключ и информацию о том, кто владелец ключа. Изготовить такой сертификат не

[2] Программно распознать отсканированный текст — не проблема. Но ДП по определению не может обратиться в некий централизованный орган для сверки *подлинности* документов, иначе перестанет быть децентрализованным. Именно отсутствие центральной точки доступа или точки отказа является главным достоинством ДП.

составляет труда. Поэтому цифровой сертификат всегда выдается уполномоченным лицом, которому вы можете доверять, — *заверяющим органом*. Цифровой сертификат содержит поле, зашифрованное закрытым ключом заверяющего органа. Чтобы удостовериться в подлинности сертификата, нам достаточно расшифровать поле открытым ключом заверяющего органа. Если расшифровка прошла успешно, мы убеждаемся в подлинности сертификата.

Даже если пользователь успешно получил цифровой идентификатор, и он принят приложением, это не решает основную проблему. Существуют различные заверяющие органы, выпускающие свои сертификаты. В самом деле, трудно вовремя добавлять или обновлять ключи, чтобы поддерживать актуальный набор ключей всех заверяющих органов. По этой причине проверку цифрового идентификатора обычно реализуют на стороне клиента — клиентское приложение проще обновить. Простой перенос проверки на сторону клиента не полностью решает проблему, потому что существует множество заверяющих органов, и механизм добавления их ключей на клиентской стороне получается довольно громоздким.

Почему пользователи не могут проверять идентичность друг друга?

Зачастую, совершая торговые сделки в реальной жизни, мы самостоятельно проверяем идентичность другого лица или доверяем эту работу другим. Такой же подход можно применить и к ДП. Пользователи перед тем, как совершить сделку, могут проверить идентификаторы друг друга. Этот подход работает лишь для определенных типов ДП, в которых люди торгуют друг с другом. Например, если ДП является децентрализованной социальной сетью, то идентификатор не может быть проверен подобным образом. Но если ДП предназначен для людей, которые что-то покупают/продают, то перед оплатой покупатель и продавец могут взаимно проверить идентификаторы. Эта идея привлекательно выглядит, но с трудом реализуется на практике, потому что вам не захочется выполнять процедуру проверки при каждой сделке, и не каждый знает, как это делается. Например, если ДП используется для заказа такси, то вы, очевидно, не захотите выполнять проверку при каждом бронировании. Но если вы изредка совершаете торговые сделки и знаете, как проверять идентификатор, будет вполне приемлемо следовать этой процедуре.

Из-за этих проблем единственный вариант, который нам остался, — это проверка личности пользователя вручную уполномоченным лицом компании, предоставляющей программу-клиент. Например, при создании учетной записи Bitcoin нам не нужна идентификация, но при конвертации биткойна в фиатные деньги[3] обменные биржи запрашивают подтверждение личности. Программа-клиент может игнорировать непроверенных пользователей и оставаться открытой для тех, чья личность подтверждена. Такое решение тоже приводит к небольшим проблемам. Если вы переходите к использованию другого клиента, то теряете прежний набор пользова-

[3] Фиатные деньги также носят наименование *деньги декретные*. Происходит это название от латинского слова *fiat* — указ, декрет, дословное значение «Да будет так». Существуют в различных видах: бумажные банкноты и безналичные деньги, деньги электронные и монеты. — *Ред.*

телей для взаимодействия, потому что разные клиенты имеют разные наборы проверенных пользователей. Все пользователи могут захотеть использовать одного клиента, создавая, таким образом, монополию среди приложений. Но это не главная проблема, потому что если один клиент не способен правильно проверить идентификатор, пользователи могут перейти к использованию другого клиента, не теряя свои критические данные, хранящиеся децентрализованно.

Проверка подлинности пользователя в приложении затрудняет уход от ответственности после совершения каких-либо мошеннических действий, предотвращает использование приложения лицами с криминальным прошлым и дает пользователям уверенность, что другой пользователь является тем, за кого себя выдает. Неважно, какая процедура применяется для проверки пользователя — у него всегда остается возможность выдать себя за другого. Цифровые идентификаторы и копии бумажных документов в равной степени могут быть украдены и использованы без согласия владельца. Важно лишь максимально затруднить подмену личности, а также собрать достаточный объем данных для идентификации пользователя и доказательства мошеннической деятельности.

Учетные записи пользователей

Приложения часто используют механизм *учетных записей*. Данные, связанные с учетной записью, должны быть доступны для редактирования только владельцу учетной записи. Децентрализованные приложения не могут обеспечить эту функциональность на основе пары логин-пароль, потому что наличие пароля еще не доказывает, что изменение учетной записи было запрошено владельцем.

Существует несколько способов реализации учетных записей пользователей в ДП. Но самый популярный путь — это использование пары открытого и закрытого ключей для представления владельца учетной записи. Хеш открытого ключа является уникальным идентификатором учетной записи. Чтобы внести правки в учетную запись, пользователь должен подписать изменения своим закрытым ключом. Мы должны полагать, что пользователи хранят свои закрытые ключи надежно и безопасно. Если пользователи теряют свои секретные ключи, они навсегда теряют доступ к своей учетной записи.

Доступ к централизованным приложениям

ДП не должно зависеть от централизованных приложений, потому что такая зависимость станет точкой отказа. Но иногда не остается другого выхода. Например, если приложению нужно получить счет футбольного матча, где оно возьмет эти данные? Разумеется, одни ДП могут зависеть от других ДП, но зачем FIFA создавать децентрализованное приложение? FIFA не станет создавать свое ДП лишь потому, что другие ДП нуждаются в данных. Децентрализованное приложение для предоставления счета матча не имеет смысла, поскольку будет полностью под контролем FIFA.

Итак, в некоторых случаях ДП вынуждено получать данные из централизованного приложения. Но вот проблема — как ДП может убедиться, что данные не подменили при передаче, и они являются актуальным ответом на запрос? Есть несколько способов решить эту проблему в зависимости от архитектуры ДП. Например, в Ethereum смарт-контракты, которые нуждаются в доступе к централизованным API, могут в качестве посредника воспользоваться сервисом Oraclize, потому что смарт-контракты не выполняют прямые HTTP-запросы. А сервис Oraclize предоставляет доказательство подлинности TLSNotary для данных, собираемых с централизованных сервисов.

Внутренняя валюта приложений

Чтобы централизованное приложение сохраняло работоспособность в течение длительного времени, его владельцу нужно как-то извлекать прибыль. У ДП нет владельца, но все равно узлы ДП нуждаются в оборудовании и сетевых ресурсах[4] для поддержания работы. То есть узлы ДП должны приносить какую-то выгоду в обмен на усилия по поддержанию их работоспособности. Именно здесь вступает в игру *внутренняя валюта*. Большинство ДП имеют встроенную внутреннюю валюту. Точнее, большинство *успешных* ДП имеют встроенную внутреннюю валюту.

Протокол консенсуса определяет, сколько валюты получает узел. В зависимости от протокола, только определенные узлы получают валюту. Мы также можем сказать, что лишь те узлы, которые способствуют поддержанию и управлению ДП, зарабатывают валюту. Узлы, которые только считывают данные, не получают ничего. Например, в Bitcoin только майнеры зарабатывают биткойны за успешно добытые блоки.

Самый важный вопрос о цифровой валюте — почему ее вообще кто-то ценит? Согласно экономической теории, все, что имеет достаточный спрос и недостаточное количество, имеет ценность.

Заставляя пользователей расплачиваться внутренней валютой приложения, мы решаем проблему спроса. Чем больше пользователей используют ДП, тем больше спрос на валюту и тем выше она ценится.

Установление фиксированного объема выпускаемой валюты приводит к ее нехватке и повышает стоимость[5]. Обычно валюта вводится в оборот постепенно, чтобы вновь подключившиеся узлы тоже имели возможность заработать.

[4] А также в обслуживающем персонале, который в общем случае не согласен работать бесплатно.

[5] Иногда не только ограничивают объем выпускаемой валюты, но и вводят механизм сделок, при котором уничтожение небольшого количества валюты является условием успешного завершения транзакции. Кроме защиты от массового мошенничества такой метод повышает стоимость валюты.

Недостатки внутренней валюты

Единственным недостатком внутренней валюты является то, что ДП не могут быть бесплатными для всех. Это как раз тот случай, когда централизованные приложения берут верх, потому что они могут быть монетизированы за счет рекламы, предоставления премиальных API для сторонних приложений и т. п. Поэтому они могут быть бесплатными для обычных пользователей. Мы не можем интегрировать рекламу в ДП, потому что некому проверять рекламу на соответствие стандартам[6]. Клиенты могут не отображать рекламу, потому что не имеют выгоды от показа объявлений.

Приложения с контролируемым доступом

До сих пор мы говорили о ДП, которые полностью открыты и не контролируют доступ. Это значит, что любой желающий может подключиться к одноранговой сети без идентификации личности.

С другой стороны, контролируемые ДП (КДП) не распахнуты настежь для всех желающих. КДП наследуют все качества открытых ДП, за исключением того, что вам требуется разрешение для подключения к сети. Способ выдачи разрешений зависит от приложения.

Поскольку вам требуется разрешение, протокол консенсуса открытого приложения может плохо работать в случае КДП. Поэтому КДП используют другие протоколы консенсуса. Кроме того, КДП не имеют внутренней валюты.

Популярные приложения

Итак, у нас есть общие знания о том, что такое ДП и чем они отличаются от централизованных приложений. Давайте познакомимся ближе с некоторыми популярными и полезными ДП. Говоря об этих приложениях, мы ограничимся уровнем, достаточным для понимания принципов работы и основных проблем, но не будем погружаться слишком глубоко.

Bitcoin

Биткойн — это децентрализованная валюта. Кроме того, это децентрализованное приложение, успех которого показал, насколько успешным может быть ДП, и побудил других людей к созданию собственных приложений.

[6] Технически не трудно учредить доверенного рекламного агента в области ДП. Однако не ясно, как обеспечить показ рекламы в клиентах с открытым исходным кодом, где любой желающий может удалить модуль показа рекламы.

Прежде, чем мы начнем подробно обсуждать, как работает протокол Bitcoin, и почему люди и правительства считают биткойн валютой, мы должны узнать, что такое *регистр* (ledger) и *блокчейн* (blockchain).

Что такое регистр?

Регистр — это, в общем случае, просто список транзакций. Регистр отличается от базы данных. В регистр мы можем *только добавить* новую транзакцию, тогда как в базе данных мы можем добавлять, модифицировать и удалять записи о транзакции. База данных может быть использована как реализация регистра.

Что такое блокчейн?

Блокчейн — это особая структура данных, применяемая для создания децентрализованного регистра[7]. Блокчейн состоит из блоков (block), особым образом соединенных в цепочку (chain). Блок содержит набор транзакций, хеш предыдущего блока, метку времени (время создания блока), сумму отчисления майнеру за блок и т. д. Поскольку каждый блок содержит хеш предыдущего блока, они связаны в цепочку. Каждый узел сети хранит полную копию блокчейна.

Для поддержания безопасности блокчейна применяют протоколы с *доказательством выполнения работы* (proof-of-work, PoW), с *доказательством владения долей* (proof-of-stake, PoS) и некоторые другие. Добавление блока происходит поразному — в зависимости от протокола. В случае с протоколом proof-of-work блок создается при помощи процедуры, которая называется *майнингом* и обеспечивает безопасность добавления блока. Майнер должен решить сложную математическую задачу и затратить дорогостоящие вычислительные ресурсы. Далее в этой книге мы более подробно поговорим о блокчейне и протоколах консенсуса.

В сети Bitcoin блокчейн хранит транзакции Bitcoin. Биткойны (монеты) вводятся в оборот путем выплаты вознаграждения майнерам, успешно создавшим новые блоки.

Основным преимуществом блокчейна является автоматизация контроля над безопасностью транзакций. Блокчейн предотвращает мошенничество и злоупотребления и может решить множество других проблем, в зависимости от способа реализации и использования.

Легален ли биткойн?

Прежде всего, биткойн — это не внутренняя валюта. Скорее, это децентрализованная валюта. Внутренние валюты приложений в основном легальны, потому что это условные цифровые активы, применимость которых ограничена.

[7] Децентрализованный распределенный регистр (decentralized and distributed ledger) определяют как способ реализации списка транзакций, хранящихся в децентрализованной форме без привязки к географической или иной принадлежности узлов сети.

Вопрос в другом: легальны ли ДП, созданные исключительно для оборота денег? Простой ответ — легальны в большинстве стран. Редкие страны полностью запретили оборот криптовалюты, а большинство пока не приняли окончательное решение.

Существует несколько причин, по которым некоторые страны уже запретили цифровые валюты, или могут вскоре принять такое решение:

- в связи с очевидной проблемой установления личности в ДП, аккаунт Bitcoin не привязан к личности пользователя и может быть использован для отмывания денег;
- виртуальные валюты очень волатильны, и существует высокий риск потери денег людьми;
- используя виртуальные валюты, можно легко уходить от налогов.

Почему мы используем биткойн?

Сеть Bitcoin предназначена исключительно для того, чтобы отправлять/получать биткойны, и ни для чего иного. Вы можете недоумевать — почему вообще существует потребность в биткойне?

Вот несколько причин, по которым люди используют сеть Bitcoin:

- основная выгода от использования биткойна — это простой и быстрый способ выполнять платежи по всему миру;
- стоимость обычного банковского перевода выше по сравнению с транзакцией в сети Bitcoin;
- хакеры могут украсть вашу платежную информацию у оператора обычной платежной системы, но хищение платежных адресов Bitcoin абсолютно бесполезно, потому что транзакция становится действительной только в том случае, если подписана вашим закрытым ключом, который нет необходимости куда-то передавать, чтобы сделать платеж.

Ethereum

Ethereum — это децентрализованная платформа, на которой можно запускать приложения в виде *смарт-контрактов* (smart contract, умный контракт). Приложение может состоять из одного или нескольких смарт-контрактов.

Смарт-контракт Ethereum — это программа, которая выполняется в сети Ethereum и работает исключительно так, как запрограммировано, без риска простоя, цензуры, мошенничества и вмешательства третьей стороны.

Главное преимущество платформы Ethereum для выполнения смарт-контрактов заключается в том, что контракты могут легко взаимодействовать друг с другом. Более того, вам не надо беспокоиться об интеграции протокола консенсуса и других вещах — напротив, вам всего лишь нужно написать логику приложения. Оче-

видно, что вы не можете создать любую разновидность ДП на основе Ethereum — вы можете построить только приложения, функции которых поддерживаются платформой Ethereum.

Ethereum имеет внутреннюю валюту под названием *эфир* (ether). Валюта необходима для развертывания смарт-контракта или выполнения его функций.

Эта книга посвящена разработке ДП на платформе Ethereum. Прочитав книгу, вы изучите Ethereum до мелочей.

Hyperledger

Hyperledger — это проект, посвященный разработке технологии контролируемых ДП. Hyperledger Fabric (или просто Fabric) — реализация проекта Hyperledger. К другим реализациям относятся Intel Sawtooth и R3 Corda.

Fabric — это контролируемая децентрализованная платформа, которая позволяет выполнять КДП, именуемые *чейнкодами* (chaincode). Мы должны развернуть собственную единицу Fabric и запустить поверх нее КДП. На каждом узле сети работает единица Fabric. Она действует по принципу plug-and-play (подключай и пользуйся) и позволяет с легкостью подключать различные функции и протоколы консенсуса.

Hyperledger использует структуру данных блокчейна. На данный момент блокчейны проекта Hyperledger позволяют выбрать вариант без консенсуса (протокол NoOps) или протокол консенсуса PBFT (Practical Byzantine Fault Tolerance, решение устойчивое к византийской ошибке[8]). Hyperledger имеет специальный удостоверяющий узел (certificate authority), который решает, кто может подключиться к сети и на какие действия имеет право.

IPFS

IPFS (InterPlanetary File System, Межпланетная файловая система) — это децентрализованная файловая система. В основу IPFS заложены принципы DHT (distributed hash table, распределенная хеш-таблица) и Merkle DAG (directed acyclic graph, направленный ациклический граф). IPFS использует такой же протокол, как BitTorrent, чтобы распределять данные по сети. Одной из ключевых опций IPFS является поддержка версий, реализованная на основе механизма Git.

Хотя IPFS называют децентрализованной файловой системой, она не обладает главным свойством файловой системы. Если мы сохранили что-то в обычную файловую систему, то уверены, что файл будет храниться, пока мы его не удалим. Но

[8] Византийская ошибка (задача византийских генералов) — в криптологии задача взаимодействия нескольких удаленных абонентов, которые получили приказы из одного центра. Часть абонентов, включая центр, могут быть злоумышленниками. Нужно выработать единую стратегию действий, которая будет выигрышной для абонентов. — *Ред.*

IPFS работает иначе. Отдельный узел не хранит все файлы. Он содержит только те файлы, которые ему нужны. Таким образом, если файл не популярен, его хранит небольшое количество узлов, и есть опасность вообще потерять к нему сетевой доступ. Поэтому многие пользователи воспринимают IPFS как децентрализованный пиринговый файлообменник. А еще вы можете считать, что IPFS — это BitTorrent, только полностью децентрализованный, не имеющий трекеров и оснащенный продвинутыми функциями.

Как работает IPFS?

Когда мы сохраняем файл в IPFS, он разбивается на фрагменты (chunks) размером менее 256 Кбайт, и для каждого фрагмента вычисляется хеш. Узлы сети хранят файлы, которые им нужны, а также таблицу хешей.

Существует четыре типа файлов IPFS: блоб[9] (blob), список (list), дерево (tree) и снимок состояния (commit). Блоб содержит фрагмент файла, сохраненного в IPFS. Список представляет собой законченный файл, содержащий список блобов и другие списки. Так как список может содержать другие списки, это помогает сжать данные в сети. Дерево представляет собой каталог, который содержит список блобов, списки, другие деревья и снимки состояния. И, наконец, снимок состояния содержит снимок (snapshot) истории изменений любого другого файла. Поскольку списки, деревья и снимки связаны с другими файлами IPFS, они формируют граф Merkle DAG.

Если мы хотим скачать файл из сети, нам нужно знать только хеш файла списка. Или, если мы хотим скачать каталог, нам просто нужен хеш файла дерева.

Поскольку каждый файл идентифицируется хешем, имена невозможно запомнить. Если мы обновим файл, то должны поделиться новым хешем со всеми, кто хочет загрузить этот файл. Чтобы справиться с проблемой, файловая система использует функцию IPNS (InterPlanetary Name System, Межпланетная система имен), которая позволяет присваивать файлам в IPFS имена с самоподписанными сертификатами или имена, удобные для людей.

Filecoin

Главная причина, которая мешает IPFS стать децентрализованной файловой системой, заключается в том, что узлы сети хранят только те файлы, которые им нужны. Filecoin — это децентрализованная файловая система, аналогичная IPFS, но имеющая внутреннюю валюту. Эта валюта поощряет узлы за хранение файлов, что повышает доступность данных и делает Filecoin более похожей на файловую систему.

Узлы сети добывают монеты Filecoin, предоставляя в аренду дисковое пространство, а для хранения/извлечения файлов вы должны заплатить этой валютой[10].

[9] Binary Large Object Block — большой двоичный блок.

[10] По схожему принципу работает файловая система проекта Storj.io.

Наряду с технологиями IPFS, Filecoin использует структуру данных блокчейна и протокол консенсуса с доказательством допустимости.

На момент подготовки этой книги Filecoin все еще находился в разработке и многие моменты оставались непонятными.

Namecoin

Namecoin — это децентрализованная база данных ключ-значение. Она тоже имеет внутреннюю валюту Namecoin, использует блокчейн и протокол доказательства работы.

В базе данных Namecoin вы можете хранить пары данных ключ-значение. Для регистрации пары вы должны потратить валюту. Зарегистрировавшись, вы должны обновлять регистрацию каждые 35 999 блоков, иначе связь ключей с данными устареет. За обновление вы также должны заплатить внутренней валютой. Нет необходимости обновлять сами ключи, то есть вам не надо тратить валюту на сохранность ключей после регистрации.

Namecoin поддерживает пространство имен, и пользователи могут упорядочить различные типы ключей. Кто угодно может создать свое пространство имен или использовать имеющееся для упорядочивания ключей.

К популярным пространствам имен относятся `a` (application-specific data, прикладные данные), `d` (domain name, имя домена), `ds` (secure domain name, имя безопасного домена), `id` (identity, личность), `is` (secure identity, подтвержденная личность), `p` (product, продукт) и некоторые другие.

Домены в зоне *.bit*

Для доступа к сайту браузер первым делом находит IP-адрес, связанный с именем домена. Таблица связей между доменными именами и фактическими IP-адресами хранится на серверах DNS, которые контролируются большими компаниями и правительствами. Следовательно, доменные имена уязвимы для цензуры. Правительства и компании обычно блокируют доменные имена, если сайт связан с чем-то незаконным, наносит какой-либо ущерб или по иным причинам.

В связи с этим существует потребность в децентрализованной базе данных доменных имен. Поскольку Namecoin хранит пары ключ-значение наподобие сервера DNS, то может применяться для создания децентрализованной службы DNS. Так оно и есть на самом деле. Пространства имен `d` и `ds` содержат ключи, которые заканчиваются на `.bit` и представляют доменные имена в зоне `.bit`. Пространство имен не содержит никаких технических соглашений для именования ключей, но все узлы и клиенты сети Namecoin согласились поддерживать это именование. Если вы попытаетесь сохранить в пространствах имен `d` и `ds` неправильные ключи, то клиенты их отфильтруют.

Браузер, который поддерживает домены `.bit`, должен провести поиск в пространствах `d` и `ds`, чтобы найти IP-адрес, соответствующий имени домена в зоне `.bit`.

Разница между пространствами имен `d` и `ds` заключается в том, что `ds` содержит домены, которые поддерживают сертификат TLS (Transport Layer Security, безопасность на уровне передачи), а пространство `d` содержит домены, которые не поддерживают TLS. Мы сделали сервис DNS децентрализованным, и аналогичным способом мы можем выпускать децентрализованные сертификаты TLS.

Как TLS работает в Namecoin? Пользователи выпускают самоподписанные сертификаты и сохраняют хеш сертификата в Namecoin. Когда клиент с поддержкой доменов `.bit` пытается получить доступ к безопасному домену, он сравнивает хеш сертификата, возвращенного сервером с хешем, который хранится в Namecoin. Если хеши совпали, браузер продолжает поддерживать связь с сервером.

Создание децентрализованной службы DNS на основе Namecoin является первым решением для так называемого *треугольника Зуко* (Zooko triangle), согласно которому приложения могут иметь три качества: децентрализованность, полезность для человека и безопасность[11]. Цифровой идентификатор может представлять не только личность, но также домен, компанию или что-то еще.

Dash

Dash — это децентрализованная валюта, аналогичная биткойну. Dash использует блокчейн и протокол консенсуса с подтверждением работы.

Dash решает некоторые из основных проблем, присущих биткойну:

- транзакции Bitcoin занимают несколько минут, но в современном мире нам надо совершать транзакции мгновенно. Причина в том, что сложность майнинга в сети Bitcoin автоматически настраивается таким образом, чтобы новый блок создавался в среднем один раз в 10 минут. Далее в этой книге мы поговорим о майнинге более подробно;
- несмотря на то, что счета Bitcoin не связаны с личностями, торговля биткойнами за реальные деньги на бирже или покупка вещей за биткойны вполне поддается отслеживанию. Следовательно, биржи или продавцы могут идентифицировать вашу личность и передать ее данные третьим лицам, включая правительство. Если вы развернули собственный узел, чтобы отправлять и принимать транзакции, ваш интернет-провайдер может видеть адреса Bitcoin и отследить владельца по IP-адресам, потому что широковещательные сообщения в сети Bitcoin не зашифрованы.

Dash призван решить эти проблемы за счет почти мгновенного совершения сделок и защиты владельца аккаунта от раскрытия личности. Он также не позволяет провайдеру выследить вас.

[11] Ключевой нюанс — согласно гипотезе Зуко, любой протокол способен достичь только двух характеристик из трех желаемых. Поэтому разработчики новых протоколов любят заявлять, что решили проблему Зуко.

В сети Bitcoin существуют два вида узлов: майнеры и обычные узлы. Однако в сети Dash существуют три типа узлов: майнеры, главные узлы (master node) и обычные узлы (ordinary node). Наличие главных узлов является отличительной особенностью Dash.

Децентрализованное управление и бюджетирование

Чтобы развернуть главный узел, вам нужно иметь 1000 монет Dash и статический IP-адрес. В сети Dash как майнеры, так и главные узлы могут зарабатывать валюту. Когда блок вычислен, 45% вознаграждения получает майнер, 45% — главный узел и оставшиеся 10% уходят в систему бюджетирования.

Главные узлы осуществляют децентрализованное управление и бюджетирование. Исходя из этого, мы можем сказать, что Dash — это децентрализованная автономная организация.

Главные узлы сети выступают в качестве акционеров. Они имеют право решать, куда потратить 10% валюты. Обычно эти 10% монет направляют на финансирование других проектов путем голосования. У каждого главного узла есть один голос.

Обсуждение предлагаемых проектов ведется за пределами сети Dash. Но голосование происходит непосредственно в сети.

Главные узлы могут являться вариантом решения проблемы личности пользователя. Главный узел может выбрать узел, которому доверяет проверку идентичности пользователя. Лицо или организация, которая владеет этим узлом, может вручную проверить документы пользователя и получить за это определенное вознаграждение. Если узел не обеспечивает хорошее качество сервиса идентификации, главные узлы могут проголосовать за другой узел. Это могло бы стать хорошим решением проблемы идентификации пользователей.

Децентрализованные услуги

Главные узлы не только утверждают или отвергают предложения, но и образуют сервисный уровень, который предоставляет различные услуги. Причина, по которой главные узлы предоставляют доступ к услугам, состоит в том, что чем больше услуг они предоставляют, тем функциональнее становится сеть, тем больше пользователей приходят в сеть со своими транзакциями, а это повышает ценность монет Dash и увеличивает размер вознаграждения. Таким образом, главные узлы увеличивают свой доход.

Главные узлы предоставляют такие услуги, как PrivateSend (услуга анонимных переводов в смешанных валютах), InstantSend (услуга практически мгновенных транзакций), DAPI (услуга децентрализованного API, благодаря которому обычный пользователь может не запускать собственный узел) и т. д.

В настоящее время могут быть одновременно задействованы только 10 главных узлов. Алгоритм выбора использует хеш текущего блока, чтобы выбрать главные узлы. Затем он запрашивает у них услугу. Одинаковый ответ, полученный от боль-

шинства узлов, подтверждает корректность услуги. Таким образом достигается консенсус относительно услуг, предоставляемых главными узлами.

Протокол подтверждения услуги (proof-of-service) гарантирует, что главные узлы доступны в сети, отвечают и содержат актуальный блокчейн.

BigChainDB

BigChainDB позволяет вам разворачивать собственные контролируемые или открытые децентрализованные базы данных. Он использует структуру данных блокчейна совместно со специфическими структурами баз данных. На момент подготовки этой книги проект BigChainDB находился в процессе разработки, поэтому многие детали оставались неясными.

BigChainDB предоставляет множество функций, таких как расширенные разрешения, запросы, линейное масштабирование, встроенная поддержка мультиресурсов и протокол объединенного консенсуса.

OpenBazaar

OpenBazaar — это децентрализованная платформа для электронной коммерции. Используя OpenBazaar, вы можете покупать и продавать товары. Пользователи сети OpenBazaar не являются анонимными, потому что их IP-адреса записываются. Узел может быть покупателем, продавцом или арбитром (модератором).

OpenBazaar использует структуру распределенной хеш-таблицы по типу проекта Kademlia. Продавец должен постоянно поддерживать работу своего узла, чтобы его товары были видны в сети.

Чтобы предотвратить массовое создание аккаунтов, применяется протокол доказательства работы. OpenBazaar препятствует массовой «накрутке» рейтингов и обзоров при помощи протоколов *доказательства уничтожения* (proof-of-burn) и *доказательства залога* (proof-of-timelock).

Для взаиморасчетов между продавцом и покупателем используют биткойны. Покупатель может пригласить арбитра при соглашении покупки. Арбитр отвечает за рассмотрение споров, которые могут возникнуть между продавцом и покупателем. Арбитром может стать любой желающий. За вынесение решения по спору арбитр получает вознаграждение[12].

Ripple

Ripple — это децентрализованная платежная платформа, позволяющая переводить фиатные деньги, криптовалюты и даже товары[13] (commodity). Она использует блок-

[12] В виде небольшой доли от суммы сделки.

[13] Предметом транзакции может быть любая единица ценности (value), например баррели нефти, мили для пассажиров или минуты мобильной связи.

чейн и собственный протокол консенсуса. В документации Ripple вы не найдете терминов «блок» и «блокчейн». Вместо них используется термин «реестр».

Транзакция Ripple проводится через *цепочку доверия* (trust chain) наподобие созданной ранее сети Hawala. В сети Ripple существуют два типа узлов: шлюзы и обычные узлы. Шлюзы поддерживают депозит и списание в одной или нескольких валютах и/или активах. Чтобы стать шлюзом в сети Ripple, вы должны обладать разрешением должного уровня для формирования цепочки доверия. Шлюзы обычно создаются финансовыми организациями, биржами, продавцами и т. п.

Каждый пользователь и шлюз имеет адрес учетной записи. Пользователь должен сформировать список шлюзов, которым он доверяет, путем добавления адресов шлюзов в *доверительный список* (trust list). Протокол консенсуса для выбора доверенных шлюзов не предусмотрен. Каждый пользователь на свой риск выбирает шлюзы, которым будет доверять. Шлюзы тоже могут составить список шлюзов, которым они доверяют.

Давайте рассмотрим на примере, как пользователь X, живущий в Индии, может перевести 500 долларов пользователю Y, проживающему в США. Допустим, в Индии есть шлюз XX, который принимает деньги (физические наличные или платеж банковской картой через веб-сайт) и формирует ripple-баланс только в индийских рупиях. Пользователь X приходит в офис XX или посещает веб-сайт и вносит на депозит 30 тысяч рупий. После чего шлюз XX запускает в сеть широковещательное сообщение: «Я должен X 30 тысяч рупий». Теперь допустим, что в США есть шлюз YY, который обслуживает только долларовые транзакции и которому доверяет пользователь Y. Однако шлюзы XX и YY не доверяют друг другу[14]. Итак, X и Y не имеют общего доверенного шлюза, XX и YY не доверяют друг другу, и вдобавок XX и YY поддерживают разные валюты. Следовательно, если X хочет перевести деньги Y, ему нужен шлюз-посредник, чтобы сформировать цепочку доверия. Пусть это будет шлюз ZZ, который пользуется доверием XX и YY и работает как с долларами, так и с рупиями. Теперь X может провести транзакцию по переводу 30 тысяч рупий из XX в ZZ, который конвертирует рупии в доллары. Затем ZZ переводит деньги в YY и требует отдать их Y. Теперь получается, что вовсе не X должен 500 долларов Y, а YY должен 500 долларов Y, ZZ должен 500 долларов YY и XX должен 30 тысяч рупий ZZ. Но это вполне нормально, потому что каждый из них доверяет своему контрагенту, даже если X и Y не доверяют друг другу. При этом XX, YY и ZZ могут фактически перевести деньги за пределами ripple-сети когда захотят, либо ждать, пока обратные транзакции обнулят эти суммы[15].

Ripple тоже имеет внутреннюю валюту, обозначаемую как XPR (ripple, рипл). Каждая транзакция, отправленная в сеть, обходится в некоторое количество риплов.

[14] Это не значит, что владельцы шлюзов не верят друг другу. Просто их адреса по какой-то причине оказались не внесены в доверительные списки.

[15] Важный момент — в сети Ripple не пересылаются деньги, а происходит *обмен долговыми расписками* между звеньями доверительной цепочки. Погашение долговых расписок может происходить как угодно. Например, с привлечением обычных платежных систем и банков.

Будучи внутренней валютой, XPR может быть отправлена кому угодно в сети без доверия. Валюту XPR можно использовать при формировании цепочки доверия. Помните, что каждый шлюз имеет собственный обменный курс. Риплы не добывают майнингом. Напротив, при запуске проекта было сгенерировано 100 миллиардов монет, которые принадлежат самой компании. Монеты вводятся в оборот вручную в зависимости от разных факторов.

Все транзакции хранятся в децентрализованном реестре, который формирует неизменяемую историю. Консенсус гарантирует, что в определенный момент времени все узлы имеют одинаковый реестр. В протоколе Ripple предусмотрен третий тип узла — валидаторы, которые являются частью протокола консенсуса. Валидаторы отвечают за проверку транзакций. Любой узел имеет право стать валидатором, но другие узлы хранят список валидаторов, которым доверяют. Этот список называется UNL (unique node list, уникальный список узлов). Валидаторы тоже хранят UNL, то есть узлы, которые являются валидаторами, должны достичь консенсус между собой. В настоящее время Ripple назначает список доверенных валидаторов, но если сеть решает, что кто-то недостоин доверия, любой узел может модифицировать свой экземпляр списка.

Вы можете сформировать реестр, взяв предыдущий реестр и применив все транзакции, которые произошли с тех пор. Чтобы принять текущий реестр, узлы должны принять предыдущий реестр и набор транзакций. После создания нового реестра обычные узлы и валидаторы запускают таймер (приблизительно на 5 секунд) и собирают транзакции, которые поступили с момента создания предыдущего реестра. Когда время таймера истекло, узлы берут те транзакции, с которыми согласны не менее 80% валидаторов из списка UNL, и формируют новый реестр. Валидаторы рассылают по сети *предложение* (список транзакций, которые по их мнению можно включить в следующий реестр). Валидаторы могут рассылать предложения несколько раз подряд и с разным набором транзакций, если они вдруг решили изменить список действительных транзакций в зависимости от предложений других членов списка UNL или других факторов. Иными словами, вам надо подождать 5–10 секунд, пока ваша транзакция будет подтверждена сетью.

Некоторые люди задаются вопросом: если каждый узел может иметь свою версию UNL, не приведет ли это к появлению множества версий реестра? До тех пор, пока существует минимальная связь между разными UNL, консенсус будет быстро достигнут. По большей части эта уверенность основана на том, что первейшей целью каждого честного узла является достижение консенсуса[16].

[16] Критики протокола Ripple считают, что опасность ветвления реестра заложена в саму основу протокола и ссылаются на пример компании Stellar Foundation, которая в 2014 году пережила крах сети. Если консенсус не достигнут, то может возникнуть ответвление реестра, созданное частью сети, несогласной с транзакциями.

Заключение

В этой главе мы узнали, что такое децентрализованные приложения и в общих чертах рассмотрели принципы их работы. Мы увидели основные проблемы, стоящие перед приложениями, и различные способы их решения. Наконец, мы познакомились с популярными приложениями и узнали, в чем их особенности и как они функционируют. Теперь вы с легкостью можете рассказать, что такое децентрализованное приложение и как оно работает.

2

Принципы работы Ethereum

Из предыдущей главы мы узнали, что такое децентрализованные приложения, и познакомились с обзором некоторых популярных приложений. Одним из них является Ethereum. Сегодня Ethereum занимает второе место по популярности после Bitcoin. В этой главе мы детально разберемся в том, как работает Ethereum, и какие приложения мы можем разработать на его основе. Мы также рассмотрим наиболее важные клиентские программы и реализации узлов платформы Ethereum.

В этой главе будут раскрыты следующие темы:

- учетные записи пользователей Ethereum;
- что такое смарт-контракты и как они работают?
- виртуальная машина Ethereum;
- как работает майнинг в протоколе консенсуса?
- использование команд консольного приложения Geth;
- настройка приложений-клиентов Ethereum Wallet и Mist;
- обзор протокола связи Whisper и платформы для хранения данных Swarm;
- перспективы Ethereum.

Знакомство с Ethereum

Ethereum — это децентрализованная платформа, поверх которой можно разворачивать децентрализованные приложения. Смарт-контракт (smart contract, умный контракт) — это программа, которая выполняется исключительно так, как запрограммировано, без какой-либо возможности простоя, цензуры, мошенничества и вмешательства третьей стороны. Смарт-контракты для платформы Ethereum могут быть написаны на различных языках программирования, включая Solidity, LLL и Serpent.

Наиболее популярен язык Solidity. Ethereum имеет внутреннюю валюту, которая называется *эфир* (ether). Для того чтобы развернуть на платформе смарт-контракт или вызвать его методы, нам нужен эфир. Может существовать несколько экземпляров смарт-контракта или приложения, и каждый экземпляр идентифицируется по его уникальному адресу. Как счета пользователей, так и смарт-контракты могут хранить эфир.

Ethereum основан на структуре данных блокчейна и протоколе консенсуса с доказательством выполнения работы. Метод смарт-контракта может быть вызван через транзакцию или через другой метод. В сети есть два типа узлов: майнеры и обычные узлы. Обычные узлы просто хранят копию блокчейна, а майнеры строят блокчейн, вырабатывая блоки.

Учетная запись Ethereum

Для создания учетной записи Ethereum нам требуется пара асимметричных ключей. Ключи шифрования могут генерироваться на основе различных алгоритмов. Ethereum использует криптографию на эллиптических кривых (Elliptic Curve Cryptography, ECC). Алгоритм ECC имеет различные параметры, от которых зависят скорость и безопасность. В Ethereum использована эллиптическая кривая `secp256k1`. Погружение в ECC и его параметры требует наличия серьезных математических знаний, но в этом нет необходимости, если мы хотим строить децентрализованные приложения на основе Ethereum.

Ethereum использует 256-битное шифрование. Открытый и закрытый ключи Ethereum представляют собой 256-битные числа. Поскольку процессор не может обработать настолько большие числа целиком, их представляют в виде шестнадцатеричной строки из 64 символов.

Каждая учетная запись представлена адресом. Когда у нас есть ключи, мы должны сгенерировать адрес. Процедура генерации адреса происходит следующим образом:

1. Генерируем хеш открытого ключа по алгоритму `keccak-256`. Это дает вам 256-битное число.
2. Отбрасываем первые 96 битов (12 байтов) — теперь у вас должно остаться 160 битов (20 байтов).
3. Затем кодируем адрес в шестнадцатеричную строку. Итак, вы получили строку из 40 символов — это и есть ваш адрес.

Теперь любой желающий может перевести эфир на этот адрес.

Транзакции

Транзакция — это подписанный пакет данных, предназначенный для перемещения эфира из одного счета на другой счет или в контракт, вызова метода контракта или развертывания нового контракта. Транзакция подписана цифровой подписью

ECDSA (Elliptic Curve Digital Signature Algorithm, алгоритм цифровой подписи на эллиптических кривых). Транзакция содержит указатель на получателя сообщения, подпись отправителя, подтверждающую его личность и намерения, количество эфира для передачи, максимальное количество вычислительных шагов, разрешенных для выполнения транзакции (так называемый *лимит газа*), и стоимость, которую отправитель транзакции готов заплатить за каждый вычислительный шаг (так называемая *цена газа*). Если назначение транзакции заключается в вызове метода контракта, она также содержит входные данные, а если она предназначена для развертывания контракта, то может содержать код инициализации. Количество газа и его цена называются *сбором за транзакцию*. Чтобы отправить эфир или выполнить метод контракта, вам необходимо транслировать транзакцию в сеть. Отправитель должен подписать транзакцию при помощи закрытого ключа.

Транзакция считается подтвержденной, если мы уверены, что она навсегда осталась в блокчейне[1]. Рекомендуется подождать 15 подтвержденных блоков, прежде чем считать транзакцию подтвержденной.

Консенсус

Каждый узел в сети Ethereum хранит копию реестра транзакций. Мы должны быть уверены, что узел не вмешался в блокчейн, и нам нужен механизм проверки достоверности блоков. А также, если мы столкнемся с двумя разными действительными блокчейнами, у нас должен быть способ узнать, какой из них выбрать.

Ethereum использует протокол с доказательством выполнения работы, чтобы защитить блокчейн от фальсификации. Принцип доказательства работы предусматривает при создании нового блока решение сложной вычислительной задачи. Решение задачи должно требовать расходования значительных вычислительных ресурсов, что делает создание нового блока трудной работой. Процесс создания нового блока называется *майнингом*. Майнеры — это узлы сети, которые вырабатывают новые блоки. Все децентрализованные приложения, которые используют доказательство выполнения работы, не используют абсолютно одинаковый набор алгоритмов. Они могут различаться в деталях относительно того, какую задачу должен решить майнер, насколько сложной должна быть задача, сколько времени занимает решение и т. п. Мы будем рассматривать алгоритм доказательства работы применительно к Ethereum.

Любой участник сети может стать майнером. Каждый майнер решает задачу индивидуально. Первый майнер, который решил задачу, становится победителем и получает пять эфиров и сборы от всех транзакций, вошедших в блок. Если ваш процессор мощнее, чем у остальных узлов сети, это не означает, что вы всегда будете

[1] Существует вероятность, что транзакция станет недействительной, если сеть придет к соглашению, что недействительным является блок транзакций, уже помещенный в блокчейн. Отмена более 15 блоков подряд технически невозможна по причине огромного объема вычислений.

победителем, потому что параметры задачи не одинаковые для всех майнеров. Однако, если ваш процессор мощнее, у вас больше шансов на успех. Алгоритм доказательства работы похож на лотерею, а вычислительная мощность процессора соответствует количеству купленных лотерейных билетов. Безопасность сети зависит не от количества майнеров, а от совокупной вычислительной мощности сети.

Не существует ограничения по количеству блоков в блокчейне и по количеству эфира, который можно выработать. Как только майнер создал блок, он транслирует его всем остальным узлам сети. Блок содержит заголовок и набор транзакций. Каждый блок содержит хеш предыдущего блока, образуя неразрывную цепь.

Давайте разберемся, какую задачу должен решить майнер и как она решается в общем виде. Чтобы создать блок, майнер первым делом собирает необработанные (сырые) транзакции, которые выложены в сеть, проверяет их и отбрасывает некорректные. Корректные транзакции должны быть правильно подписаны с использованием закрытого ключа, счет отправителя должен содержать достаточную сумму на балансе, чтобы провести транзакцию, и т. д. Далее майнер создает блок, у которого есть заголовок и содержимое. Содержимое состоит из списка транзакций, которые включены в блок. Заголовок содержит такие объекты, как хеш предыдущего блока, номер блока, одноразовое число (nonce[2]), целевое число (target), метку времени (timestamp), сложность (difficulty), адрес майнера и некоторые другие объекты. Метка времени содержит время добавления блока. Nonce здесь — это ничего не значащее число, которое майнер должен найти путем перебора, чтобы получить решение задачи. Обычно задача заключается в том, чтобы найти такое число nonce, при котором хеш готового блока меньше целевого числа или равен ему. Ethereum использует алгоритм хеширования Ethash[3] (Ethereum hash). Единственный способ решения задачи заключается в переборе всех возможных значений nonce. Целевое число — это 256-битное число, которое вычисляется исходя из различных факторов. Значение сложности в заголовке является другим представлением целевого числа. Чем меньше целевое число, тем больше времени требуется для нахождения подходящего значения nonce, и наоборот, чем больше целевое число, тем быстрее находится nonce. Так выглядит формула для вычисления сложности задачи:

```
current_block_difficulty = previous_block_difficulty +
previous_block_difficulty / 2048 * max(1 - (current_block_timestamp -
previous_blocktimestamp) / 10, -99) + int(2 ** ((current_block_number /
100000) - 2))
```

Когда майнер решил задачу и подсоединил к блокчейну новый блок, любой узел сети может проверить, является ли блокчейн правильным. Для этого он проверяет, корректны ли транзакции блока, правильно ли указана метка времени, правильно ли найдено число nonce относительно целевого числа target для всех блоков, правильно ли майнер указал величину своего вознаграждения и т. д.

---

[2] Производное от «**n**umber used **on**ly on**ce**» — одноразовое число.

[3] См. **https://github.com/ethereum/wiki/wiki/Ethash**.

Если узел сети получает два действительных блокчейна, то принимается версия, у которой совокупная сложность блоков выше.

Например, если узел сети заменяет несколько транзакций в блоке, он должен затем вычислить nonce всех последующих блоков. К тому времени, когда он найдет заново число nonce для последующих блоков, сеть создаст еще много новых блоков, поэтому сеть отвергает эту цепочку.

Метка времени

Для формулы, по которой вычисляется целевое число, требуется текущая метка времени. Кроме этого, каждый блок содержит метку времени в заголовке. Ничто не мешает майнеру использовать фиктивную метку времени, пока он добывает новый блок, но обычно они так не поступают, потому что не пройдут проверку метки времени, и другие узлы не примут этот блок. Следовательно, майнер напрасно потратит ресурсы. Когда майнер публикует новый блок, его метка времени проверяется сравнением, превышает ли она метку времени предыдущего блока. Если майнер использует метку времени, которая больше, чем текущая метка, параметр сложности (difficulty) будет меньше, потому что его значение обратно пропорционально значению метки времени. Следовательно, если другой майнер предложит блок, у которого метка времени совпадает с текущей меткой, то сеть предпочтет именно этот блок, т. к. у него сложность выше. Если майнер использует метку времени, которая больше, чем у предыдущего блока, но меньше, чем текущее время, то сложность будет выше, что потребует больше времени на решение задачи. За время, пока добывается блок, сеть произведет другие блоки, и этот блок будет отклонен, как имеющий меньшую сложность. Вот почему майнеры всегда используют точные метки времени, иначе они ничего не заработают.

Число nonce

Nonce — это 64–битное беззнаковое целое число. Оно является решением задачи. Майнер продолжает последовательно увеличивать число, пока не найдет решение. Сейчас вы можете предположить, что майнер, располагающий самой большой вычислительной мощью, всегда будет побеждать в гонке. Нет, это не так.

Хеш блока, который добывает майнер, всегда разный для разных майнеров, потому что он зависит от таких параметров, как метка времени, адрес майнера и прочих, которые не могут быть одинаковыми у всех майнеров. Следовательно, это не гонка за решением. Скорее, это лотерея. Разумеется, шансы майнера на победу зависят от вычислительной мощности, но это не значит, что самый мощный майнер всегда будет находить следующий блок.

Время блока

Формула вычисления сложности, которую вы видели раньше, содержит 10-секундную задержку, которая гарантирует, что разница времени майнинга между родительским и дочерним блоком будет находиться в пределах 10–20 секунд. Но почему именно 10–20 секунд и никакое другое значение? И зачем вводить ограничение с постоянной разностью времени вместо того, чтобы установить постоянную сложность?

Представьте, что мы имеем постоянную сложность, и майнерам просто нужно найти число nonce, при котором хеш блока меньше или равен значению target. Предположим, сложность задачи высокая. В таком случае пользователи сети не смогут даже приблизительно предположить, сколько времени займет передача эфира от одного пользователя другому. Если вычислительной мощности сети недостаточно, поиск решения (и проводка транзакции) может длиться весьма долго. Иногда сети может повезти, и тогда решение будет найдено очень быстро. Но такой сети будет очень сложно привлекать пользователей, потому что люди всегда хотят знать, сколько времени займет транзакция, аналогично тому, как при переводе с одного банковского счета на другой банковский счет мы представляем, сколько времени занимает процедура перевода. Если сложность задачи низкая, это угрожает безопасности блокчейна, потому что мощные майнеры будут добывать блоки намного быстрее слабых майнеров и могут захватить управление децентрализованным приложением. Невозможно найти постоянное значение сложности, которое гарантирует стабильность сети, потому что вычислительная мощность сети непостоянна.

Теперь вы понимаете, почему мы всегда должны знать среднее значение времени, в течение которого сеть вырабатывает новый блок. Вопрос в том, какое усредненное время считать приемлемым в диапазоне от одной секунды до бесконечности? Маленькое усредненное время достигается снижением сложности, большое усредненное время достигается увеличением сложности. Но каковы достоинства и недостатки маленького и большого усредненного времени? Прежде, чем говорить об этом, мы должны понять, что такое *устаревшие блоки*.

Что произойдет, если два майнера добудут следующий блок приблизительно в одно и то же время? Оба блока будут вполне корректными, но блокчейн не может содержать два блока с одинаковым номером, и два майнера не могут получить вознаграждение. Хотя это частая проблема, у нее есть простое решение. Сетью будет принят блок, у которого больше сложность. Соответственно, отвергнутый блок будет назван *устаревшим* блоком.

Общее количество устаревших блоков, производимых сетью, обратно пропорционально средней длительности выработки нового блока. Маленькая длительность выработки блока означает, что новые блоки будут чаще транслироваться в сеть, и возрастает вероятность того, что решение задачи будет найдено более чем одним майнером. Иначе говоря, за время, пока блок распространяется по сети, другие майнеры могут найти решение и тоже его разослать, но это будут устаревшие блоки. Но если средняя длительность выработки блока велика, это уменьшает вероят-

ность того, что несколько майнеров смогут найти решение. Даже если это возможно, то между решениями будет разрыв по времени, в течение которого первый успешный блок будет распространен по сети, другие майнеры узнают об этом, прекратят добывать этот блок и примутся за работу над следующим блоком. Если устаревшие блоки слишком часто появляются в сети, они вызывают серьезные проблемы, но если они появляются редко, то не навредят.

Но в чем заключается проблема, которую создают устаревшие блоки? Они тормозят подтверждение транзакции. Если два майнера выработали блоки приблизительно одновременно, они могут содержать разный набор транзакций, и если наши транзакции содержатся в одном из них, то мы не можем считать их подтвержденными, потому что блок могут признать устаревшим. Нам придется ждать, пока не будет найдено несколько новых блоков. По вине устаревших блоков среднее время подтверждения транзакции не равно среднему времени выработки блока.

Снижают ли устаревшие блоки безопасность блокчейна? Да, снижают. Мы знаем, что безопасность сети определяется совокупной вычислительной мощностью всех майнеров, работающих в сети. Когда вычислительная мощность растет, сложность задачи возрастает, чтобы гарантировать, что блоки не вырабатываются быстрее, чем за установленное среднее время. Следовательно, более высокая сложность означает более высокую безопасность, потому что вмешательство злонамеренного узла в блокчейн потребует наличия у злоумышленника значительной мощности хеширования. Если два блока предложены приблизительно одновременно, они делят сеть пополам, где каждая половина проверяет свою версию блокчейна. Но будет принята только одна версия, а вторая половина сети напрасно потратит вычислительную мощность на выполнение бесполезной работы. Это выглядит так, словно понизилась полезная вычислительная мощность сети, поэтому время выработки следующего блока будет сокращено за счет снижения сложности. Снижение сложности уменьшает общий уровень безопасности сети. Если устаревших блоков много, это чрезвычайно плохо влияет на безопасность блокчейна.

Ethereum борется с проблемой устаревших блоков при помощи протокола под названием GHOST (Greedy Heaviest Observed SubTree), используя модифицированную версию этого протокола. Протокол GHOST решает проблему добавлением устаревших блоков в основной блокчейн. Тем самым он увеличивает общую сложность блокчейна, т. к. совокупная сложность блокчейна включает в себя значения сложности устаревших блоков. Но как устаревшие блоки вставляют в блокчейн без конфликта между транзакциями? Суть в том, что любой блок может содержать ноль или больше устаревших блоков. Чтобы побудить майнеров включать в блокчейн устаревшие блоки, им выплачивают вознаграждение. Кроме этого, майнеры устаревших блоков тоже получают вознаграждение. Транзакции устаревших блоков не участвуют в расчете подтверждения, и майнеры устаревших блоков не получают комиссию за транзакции. Отметим, что в протоколе Ethereum старые блоки называются uncle[4] blocks.

[4] Игра слов. Кроме указания на родственную связь второго уровня (дядя-племянник) одно из переносных значений слова uncle — старик, пожилой человек, что намекает на старость блока.

Существует специальная формула, по которой вычисляется вознаграждение майнеру за прием устаревшего блока в переработку. Остаток вознаграждения переходит к блоку-племяннику (nephew block), который вбирает в себя блок-дядюшку (uncle block):

```
(uncle_block_number + 8 - block_number) * 5 / 8
```

Вы можете спросить: если отсутствие вознаграждения майнерам устаревших блоков никак не влияет на безопасность, почему они получают вознаграждение? Существует еще одна проблема, связанная с частым появлением устаревших блоков. Майнер должен получать вознаграждение пропорционально вычислительной мощности, которую он вносит в систему. Если два блока найдены приблизительно одновременно, то, вероятнее всего, в финальный блокчейн будет включен блок, добытый более мощным майнером, а слабый майнер останется без награды. Если доля старых блоков невелика, то и проблемы нет — мощный майнер получит лишь немного больше. Однако если доля устаревших блоков велика, то мощный майнер будет забирать себе намного больше, чем заслуживает, исходя из вычислительной мощности[5]. Протокол GHOST устраняет дисбаланс, выдавая вознаграждение майнерам старых блоков. Мощный майнер не получает вознаграждение целиком, но все же заметно больше, чем вообще без протокола GHOST. Поэтому мы можем и майнерам старых блоков не выдавать полное вознаграждение, а лишь часть от обычного вознаграждения[6]. Приведенная ранее формула прекрасно поддерживает баланс.

Протокол GHOST ограничивает максимальное количество старых блоков, которые может вобрать в себя блок-племянник, поэтому майнеры не могут просто вхолостую генерировать старые блоки, тормозя блокчейн.

Таким образом, если в сети появляется старый блок, он так или иначе влияет на сеть. Чем чаще появляются старые блоки, тем сильнее они влияют на сеть.

Ветвление

Если случается конфликт между узлами, подтвердившими подлинность блокчейна, и возникает больше одного подлинного блокчейна в сети, причем все они подтверждены несколькими майнерами, то говорят, что произошло *ветвление* (forking). Принято различать три вида ветвления: простое (regular fork), мягкое (soft fork, софтфорк) и жесткое (hard fork, хардфорк).

Простое ветвление возникает, если два или более майнеров выработали блок почти одновременно. Проблема решается за счет того, что сложность одного блока больше, чем у остальных.

---

[5] При этом мелкие майнеры просто потеряют всякий стимул к работе, а картель «жирных майнеров» может захватить контроль над сетью.

[6] На момент работы над переводом майнер старого блока получал 87,5% вознаграждения, а майнер блока, поглотившего старый блок, получал оставшиеся 12,5% дополнительно к вознаграждению за новый блок.

После смены исходного кода узла могут возникнуть конфликты версий узлов. В зависимости от типа конфликта может потребоваться обязательное обновление узлов у майнеров, чья совокупная вычислительная мощность превышает 50% мощности сети, либо обновление абсолютно всех узлов сети. Первый тип конфликта называется *мягким ветвлением*, а второй — *жестким*. Примером мягкого ветвления является такое обновление программного обеспечения (реализации протокола), при котором старые блоки/транзакции теряют валидность. Но если у майнеров, которые обновили свои узлы, совокупная мощность более 50%, то их новая версия блокчейна будет иметь более высокую сложность и в конечном итоге будет принята всей сетью. Примером жесткого ветвления является такое обновление исходного кода, при котором меняется механизм расчета вознаграждения, и для разрешения конфликта все майнеры должны обновиться.

Ethereum с момента запуска прошел через несколько жестких и мягких ветвлений[7].

Генезис

Первый блок блокчейна называют *генезисом* (genesis block) или блоком-прародителем. Он имеет нулевой номер. Это единственный блок в блокчейне, который не ссылается на предыдущий блок. Он не содержит транзакции, потому что к этому моменту не выпущен ни один эфир.

Два узла сети могут установить связь между собой, если имеют одинаковый генезис, т. е. синхронизация блоков между узлами может происходить только в том случае, если они оба хранят одинаковый блок-прародитель. Другой генезис, имеющий более высокую сложность, не может заменить менее сложный генезис. Во многих сетях код блока-прародителя вписан в исходный код клиентского приложения.

Деноминация эфира

Эфир, как и любая другая валюта, имеет свою шкалу деноминации:

- 1 Ether = 1000000000000000000 Wei;
- 1 Ether = 1000000000000000 Kwei;
- 1 Ether = 1000000000000 Mwei;
- 1 Ether = 1000000000 Gwei;
- 1 Ether = 1000000 Szabo;
- 1 Ether = 1000 Finney;

[7] Во время работы над переводом книги был активирован плановый хардфорк Ethereum, который стартовал 16 октября 2017 г. в 5:22 UTC. В результате успешного обновления в работу введен новый протокол под названием Byzantium. Это пятый по счету хардфорк Ethereum (предыдущий состоялся 16 ноября 2016 года).

- 1 Ether = 0.001 Kether;
- 1 Ether = 0.000001 Mether;
- 1 Ether = 0.000000001 Gether;
- 1 Ether = 0.000000000001 Tether.

Виртуальная машина Ethereum

Виртуальная машина Ethereum (Ethereum Virtual Machine, EVM) — это среда выполнения байт-кода контрактов. EVM работает на каждом узле сети. Все узлы сети выполняют все транзакции, на которые указывают смарт-контракты, поэтому каждый узел выполняет одни и те же вычисления и хранит одинаковые значения. Транзакции, которые только перемещают эфир, тоже требуют некоторых вычислений, таких как определение достаточности баланса на счете или вычитание суммы из баланса.

Каждый узел выполняет транзакции и хранит окончательное состояние по нескольким причинам. Например, существует смарт-контракт, который хранит имена и данные гостей, приглашенных на вечеринку. Каждый раз, когда добавляют нового гостя, новая транзакция транслируется в сеть. Любой узел может ознакомиться с данными каждого приглашенного на вечеринку. Для этого достаточно прочитать финальное состояние контракта.

Для любой транзакции необходимы вычисления и хранилище в сети. Следовательно, должны существовать затраты на транзакцию, иначе вся сеть будет переполнена транзакционным спамом, а у майнеров не будет стимула включать транзакции в блок, и они начнут генерировать пустые блоки. Каждая транзакция требует разных ресурсов для обработки и хранения, следовательно, разные транзакции должны иметь различную стоимость.

Существуют две реализации виртуальной машины: на исполняемом байт-коде и JIT-VM[8]. На момент подготовки этой книги JIT-VM была доступна для использования, но оставалась в состоянии разработки. В обоих случаях исходный код Solidity компилируется в байт-код. В случае JIT-VM байт-код подвергается дополнительной компиляции непосредственно в исполняемый код, поэтому JIT-VM работает быстрее своего коллеги.

Газ

Газ (gas) — это единица измерения вычислительных шагов. Каждая транзакция должна содержать лимит газа и величину вознаграждения за газ (за единичное вычисление). У майнера есть возможность выбирать транзакции и собирать вознаграждение. Если количество затраченного газа меньше или равно лимиту газа, то тран-

[8] Just-in-Time, компиляция на ходу.

закция выполняется. Если затраты газа превышают лимит, то все изменения отменяются, кроме валидности транзакции, но вознаграждение за транзакцию майнером может быть получено.

Майнеры самостоятельно устанавливают стоимость газа (цену за единичное вычисление на своем узле). Если транзакция предлагает меньшую цену газа, чем запросил майнер, то майнер отвергает транзакцию. Стоимость газа измеряется в единицах Wei (см. ранее *разд. «Деноминация эфира»*). Иными словами, майнер может отказаться включать транзакцию в блок, если предложенная стоимость газа меньше, чем ему нужно.

С каждой операцией виртуальной машины Ethereum связано количество газа, необходимое для выполнения операции.

Транзакционные издержки влияют на максимальное количество эфира, который может быть переведен с одного счета на другой. Например, если на счете хранится пять эфиров, то невозможно эти пять эфиров полностью перевести на другой счет, потому что после перевода на исходном счете не останется ничего для оплаты комиссии за транзакцию.

Если транзакция вызывает метод контракта, и метод отправляет некоторое количество монет или вызывает другой метод контракта, комиссия за транзакцию вычитается со счета, который вызвал метод.

Обнаружение узлов

Чтобы узел стал частью сети, он должен установить соединение с остальными узлами сети, благодаря чему сможет транслировать транзакции/блоки и слушать трансляцию транзакций/блоков. Узел не нуждается в соединении со всеми существующими узлами сети. Напротив, узел соединяется лишь с несколькими узлами. В свою очередь, каждый из таких узлов соединяется с несколькими другими узлами. Так формируется сеть взаимосвязанных узлов.

Но каким образом узел может найти в сети другие узлы, если не существует центральный сервер, с которым любой желающий мог бы обменяться информацией? Ethereum использует для решения этой проблемы собственный протокол обнаружения узлов, основанный на протоколе Kademlia. В соответствии с этим протоколом, в сети есть специальные узлы, которые называют *загрузочными* (bootstrap node). Загрузочный узел формирует список узлов, которые подключались к нему в течение определенного периода времени. Загрузочный узел не хранит блокчейн. Когда узел пользователя подключается к сети, он первым делом соединяется с загрузочным узлом и получает список узлов, которые соединялись с ним в течение заранее заданного периода времени. Затем новый узел соединяется и синхронизируется с узлами из списка.

Могут существовать различные экземпляры Ethereum, т. е. различные сети, каждая из которых имеет свой идентификатор. Две основные сети Ethereum называются

mainnet (главная сеть) и testnet (тестовая сеть). Сеть mainnet служит для торговли эфиром на биржах, а сеть testnet используется разработчиками для тестирования. Все, что мы изучили до сих пор, относится к блокчейну сети mainnet.

Bootnode — это наиболее популярная реализация загрузочного узла Ethereum. Если вы хотите создать собственный загрузочный узел, то можете использовать Bootnode.

Протоколы Whisper и Swarm

Whisper (шепот) и Swarm (рой) — это, соответственно, децентрализованный протокол связи и децентрализованная платформа для хранения данных, созданные разработчиками Ethereum.

Whisper помогает узлам общаться друг с другом. Он поддерживает широковещательные сообщения, связь пользователь-пользователь, шифрованные сообщения и многое другое. Он не предназначен для передачи большого объема данных.

Вы можете прочитать больше о протоколе Whisper по адресу: **https://github.com/ethereum/wiki/wiki/Whisper** и ознакомиться с обзором примеров кода по адресу: **https://github.com/ethereum/wiki/wiki/Whisper-Overview**.

Swarm похожа на Filecoin и отличается, в основном, техническими деталями и поощрением. Filecoin не оплачивает хранение, тогда как Swarm выплачивает вознаграждение — это увеличивает доступность информации. Вы можете спросить, как реализовано поощрение в Swarm? Имеет ли она внутреннюю валюту? На самом деле, у Swarm нет для выплаты вознаграждения другой валюты, кроме эфира. Существует смарт-контракт Ethereum, который отслеживает поощрения. Очевидно, что смарт-контракт не может общаться со Swarm. Наоборот, Swarm может обращаться к контракту. В общем, вы платите хранилищу через смарт-контракт, и платеж зачисляется хранилищу по наступлении заданной даты. Вы также можете сообщить контракту о пропаже файла, и контракт взыщет штраф с соответствующего хранилища.

Вы можете узнать больше о различиях между Swarm и IPFS/Filecoin по адресу: **https://github.com/ethersphere/go-ethereum/wiki/IPFS-&-SWARM** и получить исходный код смарт-контракта по адресу: **https://github.com/ethersphere/go-ethereum/blob/bzz-config/bzz/bzzcontract/swarm.sol**.

На момент работы над книгой Whisper и Swarm находились в разработке, поэтому многие тонкости пока остаются нераскрытыми.

Geth

Geth (или Go-Ethereum) — это реализация узлов Ethereum, Whisper и Swarm. Geth может быть использован как компонент всех трех протоколов или только для любого из них. Причина использования Geth заключается в потребности сделать так,

чтобы все три децентрализованных приложения вместе выглядели как единое целое, а клиент имел доступ ко всем трем приложениям через один узел сети.

Geth — это консольное приложение, которое написано на языке Go. Оно доступно для всех основных операционных систем. Текущая версия Geth не поддерживает Swarm и поддерживает только некоторые функции Whisper.

Установка Geth

Geth доступен для OS X, Linux и Windows. Он поддерживает два типа установки: бинарный файл и установка при помощи сценария (установочного скрипта). Во время подготовки книги актуальной была версия Geth 1.4.13[9]. Давайте рассмотрим установку приложения на разные операционные системы при помощи исполняемого бинарного файла. Сценарий установки применяется, если вы внесли изменения в исходный код. Мы не собираемся вносить изменения в исходный код, поэтому воспользуемся бинарным файлом.

OS X

Рекомендуем устанавливать Geth под OS X при помощи установщика Brew. Выполните две команды в окне терминала, чтобы установить Geth:

```
brew tap ethereum/ethereum
brew install ethereum
```

Ubuntu

Для установки под Ubuntu рекомендуем использовать apt-get. Выполните эти команды в окне терминала, чтобы установить Geth:

```
sudo apt-get install software-properties-common
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install ethereum
```

Windows

Geth для Windows распространяется в виде исполняемого файла. Скачайте ZIP-архив по адресу: **https://github.com/ethereum/go-ethereum/wiki/Installation-instructions-for-Windows** и распакуйте его. Внутри вы найдете исполняемый файл `geth.exe`.

Узнать больше про установку Geth для различных операционных систем можно по адресу: **https://github.com/ethereum/go-ethereum/wiki/Building-Ethereum**.

[9] Во время работы над переводом была доступна версия Geth 1.7.2.

JSON-RPC и консоль JavaScript

Geth предоставляет API JSON-RPC[10] для взаимодействия с другими приложениями и обслуживает API через HTTP, WebSocket и другие протоколы. API JSON-RPC подразделяется на следующие категории: `admin`, `debug`, `eth`, `miner`, `net`, `personal`, `shh`, `txpool` и `web3`. Вы можете найти более подробную информацию по адресу: **https://github.com/ethereum/go-ethereum/wiki/JavaScript-Console**.

Geth также предоставляет интерактивную консоль JavaScript для программного взаимодействия через API JavaScript. Эта интерактивная консоль использует JSON-RPC поверх IPC (Inter Process Communication, связь между процессами) для связи с Geth. Мы расскажем подробнее про JSON-RPC и API JavaScript в следующих главах.

Подкоманды и опции

Давайте познакомимся на примерах с важными подкомандами и опциями команд Geth. Вы можете получить перечень всех подкоманд и опций при помощи подкоманды `help`. Мы расскажем намного больше про Geth и его команды в следующих главах.

Подключение к сети *mainnet*

Узлы сети Ethereum по умолчанию общаются через порт 30303. Но узлы имеют право выбрать любой другой порт.

Для подключения к сети `mainnet` вам достаточно выполнить команду `geth`. Вот пример того, как в явном виде указать идентификатор сети и пользовательский каталог, в котором будет храниться скачанный блокчейн:

```
geth --datadir "/users/packt/ethereum" --networkid 1
```

Опция **`--datadir`** указывает путь к месту хранения блокчейна. Если он не указан, то по умолчанию используется путь `$HOME/.ethereum`.

Опция **`--networkid`** указывает идентификатор сети. Номер `1` соответствует сети `mainnet`. Если номер не указан, то по умолчанию используется `1`. Идентификатор сети `testnet` — `2`.

Создание частной сети

Для создания частной сети вам достаточно указать случайный идентификатор. Частные сети обычно создают для нужд отладки. Geth располагает различными флагами для логирования и отладки, что полезно во время разработки. Поэтому,

[10] Remote Procedure Call (RPC) — протокол удаленного вызова процедур, не сохраняющий содержание вызова.

вместо задания случайного идентификатора сети и настройки различных флагов логирования и отладки, вы можете просто использовать опцию **--dev**, которая запустит частную сеть с уже включенными флагами.

Создание аккаунта

Geth позволяет создавать счета (аккаунты), т. е. генерировать ключи и связанные с ними адреса. Для создания счета используйте команду:

```
geth account new
```

Когда вы запустите эту команду, вас попросят ввести пароль для шифрования счета. Если вы забудете ваш пароль, не останется ни малейшего шанса вернуть доступ к своему счету.

Чтобы получить список всех счетов в вашем локальном кошельке, используйте команду:

```
geth account list
```

Эта команда выведет на печать список адресов всех счетов. Ключи по умолчанию хранятся в каталоге, заданном опцией **--datadir**, но вы можете использовать опцию **-keystore**, чтобы назначить другой каталог.

Майнинг

По умолчанию Geth не запускает процесс майнинга. Чтобы дать Geth команду начать майнинг, нужно использовать опцию **--mine**. Вот еще несколько опций, относящихся к майнингу:

```
geth --mine --minerthreads 16 --minergpus '0,1,2' --etherbase
'489b4e22aab35053ecd393b9f9c35f4f1de7b194' --unlock
'489b4e22aab35053ecd393b9f9c35f4f1de7b194'
```

Здесь после опции **--mine** мы указали несколько других опций:

- опция **--minerthreads** означает общее количество потоков, задействованных при хешировании. По умолчанию используются восемь потоков[11];
- опция **--etherbase** означает адрес (счет), на который будет поступать вознаграждение. По умолчанию счет зашифрован. Чтобы получить доступ к эфиру на балансе, мы должны расшифровать счет. Расшифровка нужна, чтобы расшифровать закрытый ключ, ассоциированный со счетом. Чтобы начать майнинг, нам нет необходимости расшифровывать ключ, потому что для получения вознаграждения достаточно лишь указать адрес;
- при помощи опции **--unlock** можно разблокировать один или несколько счетов. Несколько адресов разделяются запятыми;

[11] В версии 1.7.2 по умолчанию запускаются четыре потока.

- опция `--minergpus` используется, чтобы перечислить видеокарты (GPU), которые заняты в майнинге. Чтобы получить список доступных видеокарт, используйте команду `geth gpuinfo`. На каждую видеокарту должно приходиться 1–2 Гбайт оперативной памяти компьютера. По умолчанию используются не видеокарты, а центральный процессор компьютера (CPU).

Быстрая синхронизация

На момент подготовки этой книги размер блокчейна составлял примерно 30 Гбайт[12]. Скачивание такого объема данных может занять несколько часов или даже дней, если у вас медленное подключение к Интернету. Для Ethereum реализован алгоритм быстрой синхронизации, который может ускорить скачивание блокчейна.

Для быстрой синхронизации не требуется скачивание полных блоков — загружаются только заголовки блоков, квитанции транзакций и недавняя база данных состояний. Иными словами, нам не надо скачивать и воспроизводить все транзакции. Для проверки целостности блокчейна алгоритм скачивает один полный блок через каждое определенное количество блоков. Вы можете прочитать больше про алгоритм быстрой синхронизации по адресу: **https://github.com/ethereum/go-ethereum/pull/1889**.

Чтобы воспользоваться быстрой синхронизацией при скачивании блокчейна, следует при запуске Geth воспользоваться опцией `--fast`[13].

В целях безопасности быстрая синхронизация разрешена только при начальной синхронизации (т. е., когда собственный блокчейн узла пуст). После того как узел успешно синхронизировался с сетью, быстрая синхронизация навсегда отключается. В качестве дополнительного средства безопасности, если сбой синхронизации произошел рядом с точкой ветвления или сразу после нее, быстрая синхронизация отключается, и узел переходит в режим полной синхронизации с обработкой всех блоков.

Ethereum Wallet

Ethereum Wallet (бумажник) — это приложение-клиент пользовательского интерфейса Ethereum, позволяющее создавать аккаунт, отправлять эфир, разворачивать контракты, вызывать методы контрактов и многое другое. На рис. 2.1 показано, как выглядит окно Ethereum Wallet.

Ethereum Wallet работает в паре с Geth. Когда вы запускаете Ethereum Wallet, он пытается найти локальный экземпляр Geth и соединиться с ним. Если он не находит работающий Geth, то запускает собственный узел Geth. Ethereum Wallet обме-

[12] На момент подготовки перевода размер блокчейна вырос до 108 Гбайт. Данные получены на сайте: **https://bitinfocharts.com/ethereum/**.

[13] В новой версии Geth следует использовать опцию `--syncmode "fast"`.

нивается данными с Geth через IPC (Inter-Process Communication, связь между процессами). Geth поддерживает файловый вариант IPC[14].

Если вы меняете каталог хранения данных при работающем Geth, то должны также изменить путь к файлу IPC. Чтобы Ethereum Wallet подключился к экземпляру Geth, вам следует использовать опцию `--ipcpath`, указывающую на расположение файла IPC по умолчанию. В противном случае Ethereum Wallet не сможет найти файл обмена и запустит собственный экземпляр Geth. Чтобы узнать путь к файлу IPC по умолчанию, выполните команду `geth help`, и она покажет путь по умолчанию после опции `--ipcpath`.

Ethereum Wallet доступен для операционных систем Linux, OS X и Windows. Вы можете скачать его по адресу: **https://github.com/ethereum/mist/releases**.

По аналогии с Geth, у него есть два режима установки: бинарный файл и установка с помощью сценария.

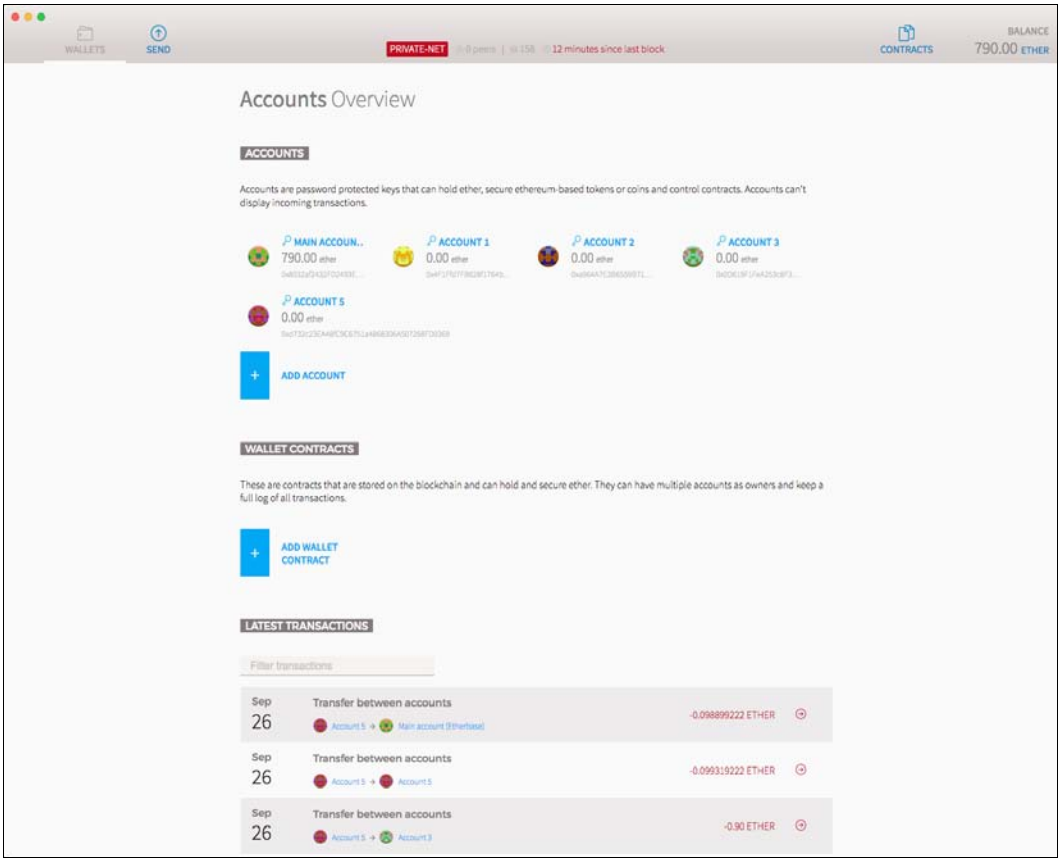

Рис. 2.1. Окно клиента Ethereum Wallet

[14] Обмен данными между процессами через общий файл.

Mist

Mist (туман) — это приложение-клиент для Ethereum, Whisper и Swarm. Mist позволяет выполнять транзакции, отправлять сообщения Whisper, проверять блокчейн и т. д. На рис. 2.2 показано, как выглядит окно Mist.

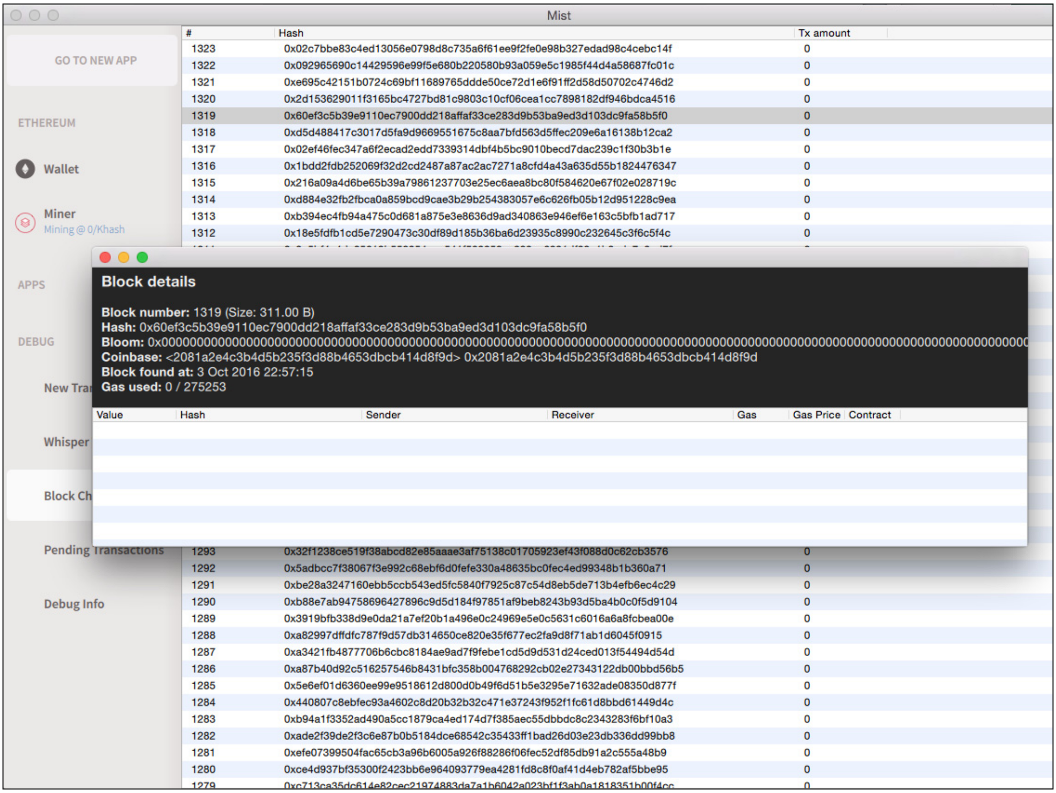

Рис. 2.2. Окно клиента Mist со встроенным браузером блоков

Взаимодействие между Mist и Geth устроено так же, как между Ethereum Wallet и Geth.

Самая популярная функция Mist — встроенный браузер блоков. В настоящее время внешний интерфейс JavaScript, работающий в браузере, может обращаться к API web3 с помощью узла Geth, используя библиотеку web3.js (библиотека, которая предоставляет интерфейсы консоли JavaScript Ethereum другим приложениям для связи с Geth).

Главная идея Mist заключается в создании третьего поколения Web (Web 3.0), которое должно ликвидировать потребность в обладании серверами за счет использования вместо централизованных серверов Ethereum, Whisper и Swarm.

Уязвимости Ethereum

Любая система имеет уязвимости. Ethereum тоже не лишена проблем с безопасностью. Очевидно, как и любая другая программа, исходный код Ethereum может содержать ошибки. И как любое другое сетевое приложение, Ethereum подвержена DoS-атакам. Но давайте обратим внимание на уникальные и более важные уязвимости Ethereum.

Атака Сибиллы

Атакующий может наполнить сеть обычными узлами, которые он единолично контролирует. После этого вы с большой вероятностью соединитесь только с узлами злоумышленника. Если вы соединились лишь с такими узлами, атакующий может отказаться пересылать вам блоки и транзакции от кого бы то ни было, фактически отключив вас от сети. При этом атакующий может пересылать вам только те блоки, которые он создал, т. е. поместить вас в отдельную сеть, и т. д.

Атака 51%

Если атакующий контролирует больше половины вычислительной мощности сети, он может генерировать блоки быстрее, чем остальная сеть. Атакующий может просто организовать ветвление блокчейна и сохранять свою частную ветку до тех пор, пока она не станет длиннее (и сложнее), чем ветка добропорядочной сети, а затем распространить ее.

Обладая мощностью более 50%, майнер может отменять транзакции, мешать всем или некоторым транзакциям быть обработанными и препятствовать включению в блокчейн блоков от других майнеров.

Обновление Serenity

Serenity — название следующего основного обновления Ethereum. Во время работы над этой книгой обновление Serenity оставалось в разработке. Для реализации обновления необходим хардфорк. Serenity изменит протокол консенсуса на Casper и будет включать в себя каналы состояния и разделение данных (sharding). Точные детали того, как это будет работать, в настоящее время остаются неясными. Давайте ограничимся поверхностным обзором обновления.

Уточнение от переводчика

Реализация обновлений отстала от графика, который был объявлен во время работы автора над книгой. Очередность основных обновлений во время работы над переводом книги выглядела так: Frontier — Homestead — Metropolis — Constantinople — Serenity. Ожидалось, что обновление Serenity будет реализовано в 2017 году. Фактически, в октябре 2017 года был проведен успешный хардфорк, который соответствует лишь половине пути к обновлению Metropolis. Таким обра-

зом, протокол Serenity вряд ли будет введен в действие ранее 2019 года. Ключевым отличием Serenity будет замена доказательства работы proof-of-work на доказательство владения долей proof-of-stake в протоколе консенсуса. Как обещают разработчики, новый протокол значительно снизит потребление энергии и распахнет двери в сеть Ethereum всем желающим, включая школьников и домохозяек.

Платежные каналы и каналы состояния

Прежде, чем говорить о каналах состояния, следует знать, что представляют собой платежные каналы. *Платежный канал* — это функция, которая позволяет нам объединить более двух транзакций по отправке эфира и передать их всего двумя транзакциями. Вот как это работает. Допустим, X — владелец веб-сайта с платными видеоканалами, а Y — пользователь. X списывает один эфир за каждую минуту видео. Соответственно, X хочет, чтобы Y платил ему каждую минуту во время просмотра видео. Разумеется, Y может транслировать транзакции каждую минуту, но при этом возникают проблемы. Например, X может ждать подтверждения транзакции, и на это время будет прерываться трансляция видео. Платежные каналы решают эту проблему. Используя платежные каналы, пользователь Y может путем отправки блокирующей транзакции депонировать несколько эфиров (допустим, даже 100 эфиров) на определенный срок (допустим, на 24 часа) в пользу X. Теперь, после просмотра одной минуты видео, пользователь Y отправляет *заверенную запись* с указанием разблокировать резерв и отдать один эфир на счет X. Спустя еще одну минуту пользователь Y транслирует еще одну заверенную запись, что можно разблокировать счет и отправить на счет X два эфира. Этот процесс продолжается до тех пор, пока Y смотрит видео на сайте. В результате, если пользователь Y просмотрел 100 минут видео и исчерпал резерв (или прошли 24 часа), владелец сайта X отправляет заверенный запрос на списание финальной суммы со счета Y. Если X не смог подать запрос на списание в оговоренные ранее 24 часа, то вся сумма резерва возвращается пользователю Y. Таким образом, сеть фактически обрабатывает всего две транзакции: блокирование и разблокирование.

Как можно видеть, платежные каналы относятся к переводам эфира. А каналы состояния позволяют объединять транзакции, относящиеся к смарт-контрактам.

Протокол консенсуса Casper

Говоря о протоколе консенсуса под названием Casper, мы должны понимать, как работает протокол подтверждения доли (подтверждение участия, proof-of-stake).

Подтверждение доли — это наиболее распространенная замена протокола с подтверждением работы, потому что подтверждение работы напрасно расходует огромные вычислительные ресурсы. Для подтверждения доли майнеру не нужно решать сложную вычислительную задачу. Чтобы создать блок, он должен подтвердить владение долей активов сети. В системе с подтверждением доли количество эфира на счете майнера рассматривается как доля, а вероятность нахождения ново-

го блока прямо пропорциональна величине доли. Поэтому, если майнер владеет 10% доли эфира в сети, он будет добывать 10% блоков.

Но вот вопрос: как мы узнаем, кто будет майнером следующего блока? Ведь нельзя позволить майнеру с наибольшей долей всегда добывать следующий блок, потому что это породит централизацию. Существуют различные алгоритмы выбора майнера следующего блока — такие, как случайный выбор блока и выбор по возрасту монет (coin-age-based selection).

Casper — это модифицированная версия протокола с подтверждением доли, которая исключает различные проблемы базовой версии.

Разделение данных

В настоящее время каждый узел должен загружать все транзакции, объем которых огромен. Если учесть, что размер блокчейна быстро увеличивается, через несколько лет станет трудно загружать весь блокчейн и поддерживать синхронизацию.

Если вы знакомы с архитектурой распределенных баз данных, то должны быть знакомы с разделением данных (sharding, фрагментация/репликация данных). Если нет, то вам достаточно знать, что это метод распределения данных по множеству компьютеров. Ethereum будет делить блокчейн на части и распределять его по узлам.

Вы можете подробнее прочитать про распределение блокчейна по адресу: **https://github.com/ethereum/wiki/wiki/Sharding-FAQ**.

Заключение

В этой главе мы детально изучили, как работает Ethereum. Мы узнали, как время выработки блока влияет на безопасность и в чем заключаются уязвимости Ethereum. Мы поговорили про Mist и Ethereum Wallet и узнали, как их установить. Мы познакомились с наиболее важными командами Geth. Наконец, мы узнали, какие изменения принесет в Ethereum новый протокол Serenity.

В следующей главе мы поговорим о различных способах хранения и защиты валюты Ethereum — эфира.

3

Разработка смарт-контрактов

Из предыдущей главы мы узнали, как работает блокчейн и как консенсус с подтверждением работы (PoW) сохраняет его безопасность. Теперь пришло время начать писать смарт-контракты, потому что мы уже достаточно хорошо разобрались в том, как работает Ethereum. Для написания смарт-контрактов Ethereum могут использоваться разные языки, но самым популярным является язык программирования Solidity, и в этой главе мы изучим его основы. А в завершение главы напишем децентрализованное приложение для доказательства наличия, целостности и права собственности на файл в заданное время. Иными словами, это будет децентрализованное приложение, которое может доказать, что файл принадлежал определенному владельцу в указанное время.

В этой главе мы рассмотрим следующие темы:

- расположение данных Solidity;
- типы данных Solidity;
- специальные переменные и функции контракта;
- управляющие структуры;
- строение и свойства контрактов;
- компиляция и развертывание контрактов.

Файлы исходного кода Solidity

Файлы исходного кода Solidity обозначаются расширением `.sol`. Как и любой другой язык программирования, Solidity имеет разные версии. Во время подготовки книги самая свежая версия имела номер 0.4.2[1].

[1] К моменту начала работы над переводом была доступна версия 0.4.18.

При помощи директивы `pragma Solidity` можно указать, для какой версии компилятора предназначен исходный код. Например, рассмотрим такую строку:

```
pragma Solidity ^0.4.2;
```

Теперь исходный код не может быть обработан компилятором с версией раньшей, чем 0.4.2, а также будет отвергнут компиляторами, начиная с версии 0.5.0 (на это указывает символ ^). Версии компиляторов в диапазоне с 0.4.2 по 0.5.0, скорее всего, будут включать только исправления ошибок, а не существенные изменения.

Существует возможность задать более сложные правила использования компилятора.

Структура смарт-контракта

Контракт похож на класс. Он содержит статические переменные, функции, модификаторы функций, события, структуры и перечисления. Контракт также поддерживает наследование, которое осуществляется копированием кода при компиляции. Смарт-контракты также поддерживают полиморфизм.

Давайте рассмотрим пример смарт-контракта, чтобы получить представление о том, как он выглядит (листинг 3.1).

Листинг 3.1. Пример смарт-контракта

```
contract Sample
{
        //статические переменные
        uint256 data;
        address owner;

        //определение события
        event logData(uint256 dataToLog);

        //модификатор
        modifier onlyOwner() {
                if (msg.sender != owner) throw;
                _;
        }

        //конструктор
        function Sample(uint256 initData, address initOwner){
                data = initData;
                owner = initOwner;
        }
```

```
        //функции
        function getData() returns (uint256 returnedData){
                return data;
        }

        function setData(uint256 newData) onlyOwner{
                logData(newData);
                data = newData;
        }
}
```

Поясним, как работает код из листинга 3.1:

1. Мы объявляем контракт при помощи ключевого слова `contract`.
2. Затем объявляем две статические переменные: переменная `data` хранит некоторые данные, а переменная `owner` — адрес бумажника Ethereum Wallet владельца. Это адрес, по которому будет развернут контракт.
3. Далее мы объявляем событие. События нужны для уведомления клиента о чем-либо. Мы будем запускать событие каждый раз, когда меняется значение `data`. Все события хранятся в блокчейне.
4. Затем мы объявляем модификатор функции. Модификаторы применяются для автоматической проверки условия перед выполнением функции. В данном случае модификатор проверяет, ссылается ли владелец контракта на функцию. Если нет, это порождает исключение.
5. Далее у нас расположен конструктор контракта. При развертывании контракта вызывается конструктор. Конструктор служит для инициализации переменных состояния.
6. Наконец, мы определяем два метода: первый метод получает значение переменной `data`, а второй — изменяет значение этой переменной.

Прежде, чем мы погрузимся в изучение функций смарт-контрактов, следует рассмотреть несколько других важных понятий, имеющих отношение к Solidity. А затем мы снова вернемся к контрактам.

Расположение данных

Все языки программирования, которые вы когда-либо изучали, хранят данные в памяти. Но Solidity хранит данные в памяти, а файловую систему — в зависимости от ситуации.

В зависимости от контекста, всегда есть расположение по умолчанию. Но для сложных типов данных, таких как строки, массивы и структуры, его можно переопределить в явном виде, добавив к объявлению типа ключевое слово `storage` или `memory`. По умолчанию для параметров функции (включая возвращаемые парамет-

ры) — это memory, по умолчанию для локальных переменных — это storage. Для статических переменных обычно также назначается расположение storage.

Расположение данных имеет значение, поскольку от него зависит поведение операторов присваивания:

- присваивание между переменными в хранилище (storage) и переменными в памяти (memory) всегда создает независимую копию данных. Но операция присваивания между сложными типами данных, если они оба расположены в памяти, не приводит к созданию копии;
- присваивание статической переменной (даже если источником является другая статическая переменная) всегда создает независимую копию данных;
- вы не можете присвоить данные сложного типа, находящиеся в памяти, статической переменной, находящейся в локальном хранилище;
- в случае, когда статическая переменная присваивается локальной переменной, локальная переменная получает указатель на статическую переменную. Это значит, что локальная переменная становится указателем.

Что такое типы данных?

Solidity — это язык со статической типизацией, то есть типы переменных должны быть заранее определены. По умолчанию все биты переменных сброшены в ноль. Если переменная объявлена внутри функции, областью видимости переменной является вся функция, независимо от места объявления.

Рассмотрим различные типы данных в языке Solidity:

- самый простой тип данных — bool. Он может хранить только булевы значения true или false;
- типы uint8, uint16, uint24 … uint256 применяются для хранения беззнаковых целых чисел длиной 8 битов, 16 битов, 24 бита … 256 битов соответственно. Аналогично, int8, int16 … int256 хранят целые числа со знаком длиной 8 битов, 16 битов … 256 битов соответственно. Служебные слова uint и int являются псевдонимами для uint256 и int256. Служебные слова ufixed и fixed обозначают дробные числа. Типы ufixed0x8, ufixed0x16 … ufixed0x256 — это беззнаковые дробные числа длиной 8 битов, 16 битов … 256 битов соответственно. Если требуется сохранить число с разрядностью большей, чем 256, то используется 256-битовый тип, и в этом случае хранится приближенное (усеченное) значение;
- тип address применяется для хранения шестнадцатеричного литерала длиной до 20 байтов, используемого для адресов Ethereum. Тип address расширяется свойствами balance и send: свойство balance применяется при проверке баланса определенного адреса Ethereum, а свойство send — при переводе эфира на указанный адрес. Метод send получает сумму в Wei, которую надо перевести, и возвращает значение true или false в зависимости от того, был ли перевод успешно выпол-

нен. Сумма в Wei вычитается из контракта, который вызвал метод `send`. Вы можете использовать в коде Solidity префикс `0x` для обозначения шестнадцатеричного представления значений переменных.

Массивы

Solidity поддерживает обычные и байтовые массивы. Размер массива может быть постоянным или динамически изменяемым. Также поддерживаются многомерные массивы.

`bytes1`, `bytes2`, `bytes3`, ..., `bytes32` — это типы байтовых массивов. `byte` является псевдонимом типа `bytes1`.

Рассмотрим пример использования обычных массивов (листинг 3.2).

Листинг 3.2. Пример использования массивов

```
contract sample{
        //массив с динамическим размером
        //везде, где виден литерал массива, создается новый массив
        //здесь myArray хранит массив с размером [0, 0].
        //тип массива определяется типом его значений,
        //следовательно, вы не можете назначить пустой литерал массива

        int[] myArray = [0, 0];

        function sample(uint index, int value){
                //индекс массива должен иметь тип uint256
                myArray[index] = value;

                //myArray2 хранит указатель на myArray
                int[] myArray2 = myArray;

                //массив постоянного размера в памяти
                //здесь мы вынуждены применить uint24, потому что
               //максимальное значение 99999,
                //и требуется 24 бита для хранения
                //данное ограничение действует для литералов в памяти,
               //потому что стоимость памяти высока.
                //так как [1, 2, 99999] - это тип uint24, следовательно
               //myArray3 тоже
                //должен иметь такой же тип, чтобы хранить указатель

                uint24[3] memory myArray3 = [1, 2, 99999];
```

```
                //вызовет исключение во время компиляции,
                //так как myArray4 не может быть
                //назначен комплексному типу, хранящемуся в памяти
                uint8[2] myArray4 = [1, 2];
        }
}
```

Вам будет полезно знать о некоторых тонкостях работы с массивами:

- массивы имеют свойство `length`. Вы можете присвоить свое значение свойству `length`, чтобы изменить размер массива. Однако вы не можете изменить размер массива, хранящегося в памяти, или размер статического массива;
- если вы попытаетесь обратиться к несуществующему индексу, это вызовет исключение.

Запомните, что массивы, структуры и таблицы сопоставлений не могут быть параметрами функций, а также не могут быть возвращены функциями.

Строки

В Solidity существует два способа создать строки: используя объявления `bytes` и `string`. Объявление `bytes` применяется для создания неформатированной строки, а `string` — для создания строки в кодировке UTF-8. Длина строки всегда динамическая.

В листинге 3.3 приведен пример синтаксиса строк.

Листинг 3.3. Пример синтаксиса при использовании строк

```
contract sample{

        //создаем пустую строку myString
        string myString = "";
        //создаем неформатированную строку
        bytes myRawString;

        function sample(string initString, bytes rawStringInit){
                myString = initString;

                //myString2 содержит указатель на myString
                string myString2 = myString;

                //myString3 - это строка в памяти
                string memory myString3 = "ABCDE";
```

```
        //меняем длину и содержимое строки
        myString3 = "XYZ";

        //присваиваем значение неформатированной строке
        myRawString = rawStringInit;

        //увеличиваем длину myRawString
        myRawString.length++;

        //вызовет исключение при компиляции,
        //потому что строка расположена не в памяти
        string myString4 = "Example";

        //вызовет исключение при компиляции
        string myString5 = initString;
    }
}
```

Структуры

Solidity поддерживает структуры. В листинге 3.4 приведен пример синтаксиса при использовании структур.

Листинг 3.4. Пример синтаксиса при использовании структур

```
contract sample{
    struct myStruct {
        bool myBool;
        string myString;
    }
    myStruct s1;

    myStruct s2 = myStruct(true, ""); //синтаксис структурного метода

    function sample(bool initBool, string initString){

        //создаем экземпляр структуры в хранилище (storage)
        s1 = myStruct(initBool, initString);

        //создаем экземпляр в памяти (memory)
        myStruct memory s3 = myStruct(initBool, initString);
    }
}
```

Учтите, что параметр функции не может быть структурой, и функция не может возвращать структуру.

Перечисление

Solidity поддерживает перечисления. В листинге 3.5 приведен пример синтаксиса при использовании перечислений.

Листинг 3.5. Пример синтаксиса при использовании перечислений

```
contract sample {

        enum OS { Windows, Linux, OSX, UNIX }

        OS choice;

        function sample(OS chosen){
                choice = chosen;
        }

        function setLinuxOS(){
                choice = OS.Linux;
        }

        function getChoice() returns (OS chosenOS){
                return choice;
        }
}
```

Сопоставление

Сопоставление (отображение, mapping) представляет собой хеш-таблицу[2]. Сопоставление может находиться только в хранилище, но не в памяти, поэтому оно может быть объявлено только как статическая переменная. Сопоставление можно рассматривать как набор данных, состоящий из пар ключ-значение. Но ключ на самом деле не хранится. Для поиска значений используется хеш ключа, вычисленный по алгоритму `keccak256`. Сопоставления не имеют длины и не могут быть присвоены другому сопоставлению.

В листинге 3.6 показан пример создания и использования сопоставления.

Листинг 3.6. Пример создания и использования сопоставления

```
contract sample{
        mapping (int => string) myMap;
```

[2] Сопоставление (Mapping) реализует ассоциативный массив. В C++ это `map`, в языке Python это называется `dict` (словарь).

```
        function sample(int key, string value){
                myMap[key] = value;
                //myMap2 это ссылка на myMap
                mapping (int => string) myMap2 = myMap;
        }
}
```

Если вы попытаетесь получить доступ к несуществующему ключу, все биты результата будут нулевыми.

Оператор *delete*

Оператор `delete` можно применить к любой переменной, чтобы сбросить ее в состояние по умолчанию, когда все биты равны нулю.

Если мы применяем оператор `delete` к динамическому массиву, он удаляет все элементы массива, и длина массива становится равной нулю. Но если применить его к статическому массиву, то обнуляются все элементы массива.

Если применить оператор `delete` к сопоставлению целиком, то ничего не произойдет. Но если вы примените `delete` к ключу сопоставления, то значение, связанное с ключом, будет удалено (обнулено).

В листинге 3.7 приведен пример, демонстрирующий применение оператора `delete`.

Листинг 3.7. Пример использования оператора `delete`

```
contract sample {

        struct Struct {
                mapping (int => int) myMap;
                int myNumber;
        }

        int[] myArray;
        Struct myStruct;

        function sample(int key, int value, int number, int[] array) {
                //сопоставления не могут быть присвоены, поэтому при
                //конструировании структуры мы игнорируем сопоставления
                myStruct = Struct(number);

                //задаем соответствие key/value
                myStruct.myMap[key] = value;
                myArray = array;
        }
```

```
        function reset(){
                //теперь длина myArray равна нулю
                delete myArray;
                //myNumber теперь будет обнулен,
                //но myMap не изменится
                delete myStruct;
        }

        function deleteKey(int key){
                //здесь мы удаляем ключ
                delete myStruct.myMap[key];
        }
}
```

Преобразование элементарных типов

Все сущности языка Solidity, которые не являются массивами, строками, структурами, перечислениями и сопоставлениями, относятся к элементарным типам.

Если в операторе встречаются два разных элементарных типа, то компилятор пытается выполнить неявное преобразование типа одного из операндов к типу другого операнда. В общем случае неявное преобразование возможно, если оно имеет семантический смысл и не приводит к потере информации. Например, `uint8` можно преобразовать в `uint16`, а `uint128` можно преобразовать в `uint256`. Но `int8` нельзя преобразовать в `uint256`, потому что `uint256` не может хранить отрицательные числа. Кроме того, целые числа без знака могут быть преобразованы в байты того же или большего размера, но не наоборот. Любой тип, который может быть преобразован в `uint160`, также может быть преобразован в тип `address`.

Solidity поддерживает и явное преобразование типов. Если компилятор не допускает неявное преобразование типов, вы можете попробовать применить явное преобразование. Тем не менее, настоятельно рекомендуется избегать явного преобразования, потому что оно может дать неожиданные результаты.

Давайте рассмотрим пример явного преобразования:

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // теперь b = 0x5678
```

Здесь мы преобразовываем `uint32` в `uint16` в явном виде, то есть превращаем больший тип в меньший. Следовательно, старшие разряды числа отбрасываются.

Ключевое слово *var*

В Solidity есть ключевое слово `var`, служащее для объявления переменных. В этом случае тип переменной определяется автоматически, в зависимости от типа первого значения, которое будет присвоено переменной. Как только первое значение при-

своено, тип переменной фиксируется. Если вы захотите присвоить этой переменной значение другого типа, это вызовет преобразование типа.

Вот простой пример, который демонстрирует применение `var`:

```
int256 x = 12;
//тип переменной y будет тоже int256
var y = x;
uint256 z= 9;
//здесь возникнет исключение, потому что
//неявное преобразование типа int в uint невозможно
y = z;
```

Помните, что ключевое слово `var` нельзя использовать при объявлении массивов и сопоставлений, а также при объявлении параметров функции и статических переменных.

Управляющие структуры

Solidity поддерживает управляющие структуры `if`, `else`, `while`, `for`, `break`, `continue`, `return` и конструкцию `? :` [3].

В листинге 3.8 приведены примеры использования управляющих структур.

Листинг 3.8. Примеры управляющих структур

```
contract sample{
        int a = 12;
        int[] b;
        function sample()
        {
        //оператор "==" вызовет исключение для сложных типов данных
                if(a == 12)
                {
                }
                else if(a == 34)
                {
                }
                else
                {
                }

                var temp = 10;
```

[3] Тернарный оператор условия.

```
                while(temp < 20)
                {
                        if(temp == 17)
                        {
                                break;
                        }
                        else
                        {
                                continue;
                        }

                        temp++;
                }

                for(var iii = 0; iii < b.length; iii++)
                {
                        //операторы цикла for
                }
        }
}
```

Оператор *new* и создание контракта

Контракт может создать новый контракт при помощи оператора `new`. Должен быть известен полный код создаваемого контракта.

Далее приведен пример создания контракта. Мы не станем выделять этот простой пример в отдельный листинг.

```
contract sample1
{
        int a;

        function assign(int b)
        {
                a = b;
        }
}

contract sample2{
        function sample2()
        {
                sample1 s = new sample1();
                s.assign(12);
        }
}
```

Исключения

В некоторых случаях исключения генерируются автоматически. Вы можете использовать оператор `throw`, чтобы принудительно вбросить исключение. Действие исключения состоит в том, что выполнение всех текущих вызовов прекращается и ревертируется (возвращается к исходному состоянию, какие-либо изменения баланса и статуса счета не происходят). Перехват исключений невозможен. Так выглядит пример принудительного вброса исключения:

```
contract sample
{
        function myFunction()
        {
                throw;
        }
}
```

Вызов внешних функций

В Solidity существуют два вида вызова функций: внутренний и внешний. Внутренний вызов происходит, когда одна функция вызывает другую функцию внутри того же самого контракта.

При внешнем вызове происходит вызов функции другого контракта. Давайте рассмотрим пример, приведенный в листинге 3.9.

Листинг 3.9. Пример вызова внешней функции

```
contract sample1
{
        int a;
        //"payable" - это встроенный модификатор
        //этот модификатор требуется, если другой контракт отправляет эфир,
        //когда вызывает метод
        function sample1(int b) payable
        {
                a = b;
        }

        function assign(int c)
        {
                a = c;
        }
```

```
        function makePayment(int d) payable
        {
                a = d;
        }
}

contract sample2{
        function hello()
        {
        }

        function sample2(address addressOfContract)
        {
                //отправка 12 wei при создании контракта
                sample1 s = (new sample1).value(12)(23);

                s.makePayment(22);

                //также отправляем эфир
                s.makePayment.value(45)(12);

                //назначаем стоимость газа
                s.makePayment.gas(895)(12);

                //отправляем эфир, а также назначаем стоимость газа
                s.makePayment.value(4).gas(900)(12);

                //hello() - это внутренний вызов, this.hello() - это
                //внешний вызов
                this.hello();

                //указатель на контракт, который уже развернут
                sample1 s2 = sample1(addressOfContract);

                s2.makePayment(112);
        }
}
```

Вызовы с ключевым словом `this` выполняются как внешние. Ключевое слово `this` внутри функции ссылается на текущий экземпляр контракта.

Свойства контракта

Пришло время подробнее изучить контракты. Мы узнаем о некоторых новых их свойствах, а также более детально рассмотрим уже знакомые опции.

Видимость

Видимость статических переменных или функций определяет, кто их может видеть. Существует четыре типа видимости функций и статических переменных: `external`, `public`, `internal` и `private`.

По умолчанию функции имеют видимость `public`, а статические переменные — видимость `internal`. Давайте разберемся, что означает каждая разновидность видимости:

- `external` — внешние функции (external functions) могут быть вызваны только из другого контракта или через транзакцию. Внешняя функция `f` не может быть вызвана внутренним вызовом. Это означает, что вызов `f()` не будет работать, но вызов `this.f()` сработает. Вы не можете применить видимость `external` к статическим переменным;
- `public` — общие функции и статические переменные могут быть доступны любыми возможными способами. Созданные компилятором функции доступа включают все статические переменные. Вы не можете создавать собственные функции доступа (аксессоры). Фактически компилятор генерирует только геттеры[4], но не сеттеры[5];
- `internal` — внутренние функции и статические переменные могут быть доступны только внутри текущего контракта и контрактов, наследующих его. Вы не можете использовать для доступа ключевое слово `this`;
- `private` — частные функции и статические переменные похожи на внутренние, но они недоступны для наследующих контрактов.

В листинге 3.10 приведен пример, демонстрирующий видимость и функции доступа.

Листинг 3.10. Пример различной видимости и функций доступа

```
contract sample1
{
        int public b = 78;
        int internal c = 90;
```

[4] Геттер (getter) — метод, возвращающий значение переменной.

[5] Сеттер (setter) — метод, присваивающий значение переменной.

```
        function sample1()
        {
                //внешний доступ
                this.a();

                //ошибка компиляции
                a();

                //внутренний доступ
                b = 21;

                //внешний доступ
                this.b;

                //внешний вызов
                this.b();

                //ошибка компиляции
                this.b(8);

                //ошибка компиляции
                this.c();

                //внутренний доступ
                c = 9;
        }

        function a() external
        {
        }
}

contract sample2
{
        int internal d = 9;
        int private e = 90;
}

//sample3 наследует sample2
contract sample3 is sample2
{
        sample1 s;

        function sample3()
        {
                s = new sample1();
```

```
                //внешний доступ
                s.a();

                //внешний доступ
                var f = s.b;

                //ошибка компиляции, так как аксессор нельзя использовать
                //для присвоения значения (в качестве сеттера)
                s.b = 18;

                //ошибка компиляции
                s.c();

                //внутренний доступ
                d = 8;

                //ошибка компиляции
                e = 7;
        }
}
```

Модификаторы

Мы уже видели раньше модификатор[6] функции и даже использовали его встроенный вариант (см. листинг 3.9). Теперь изучим модификаторы подробнее.

Модификаторы наследуются дочерними контрактами и могут быть переназначены ими. К одной функции можно применить несколько модификаторов, разделенных пробелами, при этом модификаторы будут задействованы в порядке перечисления. Вы также можете передавать аргументы в модификаторы.

Внутри модификатора следующий модификатор (или тело функции, в зависимости от того, что будет дальше) начинается после символов `_;`.

Давайте рассмотрим более сложный пример модификаторов функций (листинг 3.11).

Листинг 3.11. Пример использования модификаторов функций

```
contract sample
{
        int a = 90;.

        modifier myModifier1(int b) {
                int c = b;
                _;
```

[6] Модификаторы служат для изменения объявлений типов и их членов.

```
                c = a;
                a = 8;
        }

        modifier myModifier2 {
                int c = a;
                _;
        }

        modifier myModifier3 {
                a = 96;
                return;
                _;
                a = 99;
        }

        modifier myModifier4 {
                int c = a;
                _;
        }

        function myFunction() myModifier1(a) myModifier2 myModifier3 returns (int d)
        {
                a = 1;
                return a;
        }
}
```

В этом примере мы описали несколько модификаторов и применили эти модификаторы к функции `myFunction()`. Вот как будет выполняться функция `myFunction()` на самом деле:

```
int c = b;
        int c = a;
                a = 96;
                return;
                        int c = a;
                                a = 1;
                                return a;
                a = 99;
c = a;
a = 8;
```

В данном случае вызов `myFunction()` возвратит ноль. Но если затем вы прочитаете значение статической переменной `a`, то окажется, что ее значение равно восьми.

Оператор `return` в модификаторе или теле функции приводит к немедленному выходу из функции и возвращает значения переменных в том виде, как есть.

Но в случае нашей функции с модификаторами код после оператора `return` продолжает выполняться даже после того, как завершен вызываемый код. То есть, при наличии модификатора, код после метки `_;` в предыдущем модификаторе выполнен уже после завершения вызываемого кода. В предыдущем фрагменте (который соответствует фактическому коду функции) строки `5`, `6` и `7` никогда не выполняются. После строки номер `4` будут выполнены строки `8`, `9` и `10`.

Оператор `return` внутри модификатора не может иметь связанное значение — он всегда возвращает только ноль.

Резервная функция

Контракт может содержать только одну безымянную функцию, которая называется *резервной функцией* (fallback function). Эта функция не может иметь аргументы и ничего не возвращает. Она выполняется при обращении к контракту, если ни одно имя функции в контракте не совпадает с указанным при вызове идентификатором.

Эта функция также выполняется, как только контракт получает эфир без обращения к функциям (транзакция переводит эфир контракту, но не вызывает ни одного метода контракта). В данном случае для вызова функции доступно очень мало газа (точнее, 2300 единиц), поэтому важно делать резервные функции как можно дешевле[7].

Вот небольшой пример резервной функции:

```
contract sample
{
        function() payable
        {
                //например, код для уведомления о том,
                //сколько эфиров было получено
        }
}
```

Наследование

Solidity поддерживает множественное наследование путем копирования кода, включая полиморфизм. Если контракт наследует от нескольких разных контрактов, то в блокчейне создается только один контракт. Код родительских контрактов всегда копируется в финальный контракт.

[7] На практике резервную функцию часто используют в простых смарт-контрактах, единственное назначение которых — получать или отправлять дискретные платежи. Например, это может быть контракт для первоначального размещения токенов, который получает оплату за токен и создает запись в реестре акционеров. Прочитать больше о резервной функции можно по адресу:

https://github.com/ConsenSys/Ethereum-Development-Best-Practices/wiki/Fallback-functions-and-the-fundamental-limitations-of-using-send()-in-Ethereum-&-Solidity.

В листинге 3.12 приведен пример наследования.

Листинг 3.12. Пример наследования контрактов

```
contract sample1
{
        function a(){}
        function b(){}
}

//контракт sample2 наследует sample1
//и переопределяет функцию b()
contract sample2 is sample1
{
        function b(){}
}

contract sample3
{
        function sample3(int b)
        {
                //код функции
        }
}

//контракт sample4 наследует sample1 и sample2,
//но контракт sample1 - это родитель для sample2, по сути мы получаем
//лишь реализацию сущности sample1
contract sample4 is sample1, sample2
{
        function a(){}
        function c(){

                //выполняем метод a() контракта sample4
                a();

                //выполняем метод a() контракта sample1
                sample1.a();

                //выполняем метод sample2.b(), потому что он последний
                //в списке родителей и перекрывает собой sample1.b()
                b();
        }
}
```

```
//если конструктор содержит аргумент, он должен быть представлен
//в конструкторе дочернего контракта,
//но в Solidity дочерний конструктор не вызывает родительский
//конструктор, а инициализирует и копирует родителя
contract sample5 is sample3(122)
{
}
```

Ключевое слово *super*

Ключевое слово `super` применяется для ссылки на следующий контракт в окончательной последовательности наследования. Для более глубокого понимания рассмотрим пример из листинга 3.13.

Листинг 3.13. Пример использования ключевого слова `super`

```
contract sample1
{
}

contract sample2
{
}

contract sample3 is sample2
{
}

contract sample4 is sample2
{
}

contract sample5 is sample4
{
        function myFunc()
        {
        }
}

contract sample6 is sample1, sample2, sample3, sample5
{
        function myFunc()
        {
                //соответствует ссылке sample5.myFunc()
                super.myFunc();
        }
}
```

По отношению к контракту `sample6` цепочка наследования имеет вид:

```
sample6, sample5, sample4, sample2, sample3, sample1
```

Цепочка наследования начинается с самого младшего контракта и заканчивается самым старшим.

Абстрактные контракты

Контракты, которые содержат только прототипы функций, но не их реализацию, называются *абстрактными контрактами*. Такие контракты не могут быть скомпилированы (даже если они содержат реализованные функции наряду с нереализованными функциями). Если контракт наследуется от абстрактного контракта и не реализует все нереализованные функции путем переопределения, то такой контракт сам становится абстрактным.

Абстрактные контракты объявляются только для того, чтобы сделать интерфейс известным компилятору. Это полезно, когда вы обращаетесь к развернутому контракту и вызываете его функции. Вот простой пример, который демонстрирует такую ситуацию:

```
contract sample1
{
        function a() returns (int b);
}

contract sample2
{
        function myFunc()
        {
        sample1 s = sample1(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);
        //без абстрактного контракта эта часть кода
        //вызовет ошибку компиляции
        s.a();
        }
}
```

Библиотеки

Библиотеки почти не отличаются от контрактов, но их назначение состоит в том, что они размещаются только один раз по определенному адресу, а затем повторно используются различными контрактами. Это означает, что если вызываются функции библиотеки, то их код выполняется в контексте вызывающего контракта. Ключевое слово `this` будет указывать на вызывающий контракт, и, что важно, можно получить доступ к хранилищу вызывающего контракта. Поскольку библиотека является изолированным фрагментом исходного кода, она может получить доступ только к статическим переменным вызывающего контракта и только в том случае,

если доступ предоставлен в явном виде (в противном случае не будет иного способа обращения).

Библиотеки не могут иметь свои статические переменные. Они не поддерживают наследование и не могут получать эфир. Библиотеки могут содержать структуры и перечисления.

Как только библиотека Solidity размещена в блокчейне, ей может воспользоваться любой желающий, — если знает ее адрес и имеет на руках исходный код (только прототипы или полную реализацию). Исходный код нужен компилятору Solidity, чтобы убедиться, что методы, которые вы собираетесь вызывать, действительно содержатся в библиотеке.

Ознакомьтесь с небольшим примером кода:

```
library math
{
        function addInt(int a, int b) returns (int c)
        {
                return a + b;
        }
}

contract sample
{
        function data() returns (int d)
        {
                return math.addInt(1, 2);
        }
}
```

Мы не можем вставить адрес библиотеки в исходный код контракта. Вместо этого мы должны сообщить компилятору адрес библиотеки в процессе компиляции.

Библиотеки имеют много вариантов применения. Рассмотрим два основных случая использования библиотек:

- если у вас есть много контрактов, которые содержат общий код, вы можете поместить общий код в виде библиотеки. Это позволит сэкономить газ, потому что расход газа зависит в том числе и от размера контракта. Следовательно, мы можем в некотором смысле воспринимать библиотеку как родительский контракт по отношению к контрактам, которые ее используют. Но использование родительских контрактов вместо библиотек не приводит к экономии газа, потому что при наследовании происходит копирование родительского кода. Поскольку библиотеки рассматриваются как родительские контракты, библиотечные функции с ограниченной внутренней видимостью копируются в контракт, который их использует. В противном случае внутренние функции библиотеки не могут быть вызваны контрактом — ведь для этого потребуется внешний вызов, а внутренние функции не могут быть вызваны при помощи внешнего вызова;

- библиотеки могут быть использованы для добавления функций принадлежности к типам данных.

Конструкция *using ... for ...*

Конструкцию `using ... for ...` можно использовать для подключения библиотечных функций (из библиотеки A к любому типу B). Эти функции будут получать объект, который их вызвал, в качестве своего первого параметра.

Результатом работы команды вида `using A for *;` будет присоединение функций из библиотеки A ко всем типам.

В листинге 3.14 приведен пример использования конструкции `using ... for ...`

Листинг 3.14. Пример использования конструкции `using … for …`

```
library math
{
        struct myStruct1 {
                int a;
        }

        struct myStruct2 {
                int a;
        }

        //Здесь мы должны обратиться к хранилищу s, чтобы получить ссылку,
        //в противном случае addInt будет обращаться не к тому
        //экземпляру myStruct1, который его вызывает
        function addInt(myStruct1 storage s, int b) returns (int c)
        {
                return s.a + b;
        }

        function subInt(myStruct2 storage s, int b) returns (int c)
        {
                return s.a + b;
        }
}

contract sample
{
        //"*" присоединяет функции библиотеки ко всем структурам
        using math for *;
        math.myStruct1 s1;
        math.myStruct2 s2;
```

```
        function sample()
        {
                s1 = math.myStruct1(9);
                s2 = math.myStruct2(9);
                s1.addInt(2);
                //ошибка компиляции, потому что функция addInt
                //не подключена к myStruct2
                s2.addInt(1);
        }
}
```

Возврат нескольких значений

Solidity позволяет функциям возвращать несколько значений. Вот небольшой пример, который это демонстрирует:

```
contract sample
{
        function a() returns (int a, string c)
        {
                return (1, "ss");
        }

        function b()
        {
                int A;
                string memory B;

                //A получит 1 и B получит "ss"
                (A, B) = a();

                //A получит 1
                (A,) = a();

                //B получит "ss"
                (, B) = a();
        }
}
```

Импорт файлов исходных кодов Solidity

Solidity позволяет импортировать в исходный код содержимое других файлов с исходным кодом. Вот короткий пример, который демонстрирует эту возможность:

```
//Этот оператор импортирует все глобальные символы из "filename"
//(и символы, импортированные там) в текущую глобальную область.
```

```
//"filename" может содержать абсолютный или относительный путь.
//Это может быть только HTTP-адрес
import "filename";

//создает новый глобальный символ symbolName, который является членом
//глобального пространства символов "filename"
import * as symbolName from "filename";

//creates new global symbols alias and symbol2 which reference symbol1 and symbol2
from "filename", respectively.

import {symbol1 as alias, symbol2} from "filename";

//this is equivalent to import * as symbolName from "filename";.
import "filename" as symbolName;
```

Глобальные переменные

Существуют особые переменные и функции, которые всегда доступны глобально. Они рассмотрены в следующих разделах.

Свойства блока и транзакции

Блок и транзакция имеют следующие стандартные свойства:

- `block.blockhash(uint blockNumber) returns (bytes32)` — хеш данного блока, работает только для 256 последних блоков;
- `block.coinbase (address)` — адрес майнера текущего блока;
- `block.difficulty (uint)` — сложность текущего блока;
- `block.gaslimit (uint)` — лимит газа текущего блока. Определяет максимальное количество газа, которое можно потратить на все транзакции блока. Его назначение заключается в поддержании низкого времени распространения и обработки блока. Майнеры имеют право принять лимит газа для текущего блока в диапазоне ~0,0975% ($^{1}/_{1024}$) от лимита газа последнего блока, поэтому лимит газа должен быть усредненным относительно требований майнеров;
- `block.number (uint)` — номер текущего блока;
- `block.timestamp (uint)` — метка времени текущего блока;
- `msg.data (bytes)` — полные данные вызова. Содержат функцию, вызываемую транзакцией, и ее аргументы;
- `msg.gas (uint)` — остаток газа;
- `msg.sender (address)` — адрес отправителя сообщения (текущего вызова);

- `msg.sig (bytes4)` — четыре первых байта вызова (идентификатор функции);
- `msg.value (uint)` — количество Wei, переданных с сообщением;
- `now (uint)` — метка времени текущего блока (псевдоним для `block.timestamp`);
- `tx.gasprice (uint)` — цена газа для транзакции;
- `tx.origin (address)` — адрес отправителя транзакции (полная цепочка вызовов).

Свойства, связанные с адресом

Свойства, связанные с адресом, входят в следующий перечень:

- `<address>.balance (uint256)` — баланс заданного адреса в Wei;
- `<address>.send(uint256 amount) returns (bool)` — отправка[8] заданного количества Wei на заданный `address`. Возвращает `false` при неудаче.

Переменные, связанные с контрактом

Переменные, связанные с контрактом:

- `this` — указатель на текущий контракт, в явном виде преобразуется в тип `address`;
- `selfdestruct(address recipient)` — уничтожает текущий контракт и переводит все средства контракта на указанный адрес[9].

Единицы эфира

Сумма валюты может иметь суффикс `wei`, `finney`, `szabo` или `Ether` для конвертации между субноминациями эфира, где сумма без указания суффикса номинирована в wei. Например, сопоставление `2 Ether == 2000 finney` вернет `true`.

Доказательство наличия, целостности и принадлежности файла

Давайте напишем на языке Solidity контракт, который может подтвердить принадлежность файла, не раскрывая содержимое самого файла. Он может доказать, что файл существовал в определенное время, и проверить целостность документа.

---

[8] Разумеется, в данном случае метод, выполняющий стандартное действие, с натяжкой можно отнести к свойствам адреса. Скорее речь идет о свойствах результата действия, которое применили к адресу. Оно может закончиться либо удачно (true), либо неудачно (false).

[9] Снова в качестве свойства приводится стандартный метод. И в данном контексте мы тоже подразумеваем результат действия, которое завершено либо удачно (true), либо неудачно (false).

При этом мы получим доказательство владения, сохранив пару, состоящую из хеша файла и имени владельца. Мы получим доказательство существования, сохранив пару, состоящую из хеша файла и метки времени блока. Наконец, сохранение самого хеша доказывает целостность файла. Если файл изменить, то его хеш тоже изменится, и контракт больше не сможет найти файл, что послужит доказательством изменения файла.

В листинге 3.15 приведен исходный код контракта, который решает эту задачу[10].

Листинг 3.15. Пример контракта, доказывающего владение файлом

```
contract Proof
{
        struct FileDetails
        {
                uint timestamp;
                string owner;
        }

        mapping (string => FileDetails) files;

        event logFileAddedStatus(bool status, uint timestamp, string owner,
        string fileHash);

        //функция для сохранения владельца файла и метки времени блока
        function set(string owner, string fileHash)
        {
                //не существует корректного способа проверить,
                //есть ли уже такой ключ,
                //поэтому мы проверяем значение по умолчанию (все биты нулевые)
                if(files[fileHash].timestamp == 0)
                {
                        files[fileHash] = FileDetails(block.timestamp, owner);

                //мы создаем событие, благодаря которому программа-оболочка
                        //узнает от контракта, что наличие файла
                        //и принадлежности сохранено
                        logFileAddedStatus(true, block.timestamp, owner, fileHash);
                }
```

[10] Дополнительно напомним начинающим разработчикам, что смарт-контракт сам по себе не общается с пользователем. Это back-end (серверная часть). Пользователь же имеет дело с приложением front-end, то есть с приложением уровня клиентского интерфейса. Например, это может быть оболочка кошелька, приложение для голосования или веб-интерфейс. Приложения front-end вызывают методы контракта и получают от него ответные данные.

```
            else
            {
                    //здесь мы сообщаем программе-оболочке, что
                    //подтверждение существования
                    //и принадлежности не могут быть сохранены, потому
                    //что данные файла
                    //уже были сохранены раньше
                    logFileAddedStatus(false, block.timestamp, owner, fileHash);
            }
    }

    //функция для получения сведений о файле
    function get(string fileHash) returns (uint timestamp, string owner)
    {
            return (files[fileHash].timestamp, files[fileHash].owner);
    }
}
```

Компиляция и развертывание контракта

Ethereum имеет собственный компилятор SOLC (SOLidity Compiler), взаимодействие с которым происходит через командную строку. По адресу: **http://solidity.readthedocs.io/en/develop/installing-solidity.html#binary-packages** вы найдете руководство по установке, а по адресу: **https://solidity.readthedocs.io/en/develop/using-the-compiler.html** получите инструкции по использованию компилятора.

Но мы не станем работать с компилятором напрямую. Мы воспользуемся инструментами, которые называются solcjs и браузер Solidity. Утилита solcjs позволяет компилировать код Solidity в среде Node.js, а браузер Solidity — это среда разработки (IDE), которая подходит для небольших контрактов.

Давайте прямо сейчас скомпилируем контракт из листинга 3.15 при помощи браузера Solidity, который нам предоставляет платформа Ethereum. Прочитать больше об этом браузере можно по адресу: **https://Ethereum.github.io/browser-Solidity/**[11]. Вы также можете по ссылке: **https://github.com/Ethereum/browser-Solidity/tree/gh-pages** скачать исходный код браузера и использовать его локально, без подключения к сети.

Главное преимущество браузера Solidity заключается в том, что он имеет встроенный редактор и генерирует код для развертывания контракта.

Итак, скопируйте и вставьте в редактор исходный код из листинга 3.15 — вы увидите, что контракт компилируется и получается код web3.js для развертывания контракта при помощи интерактивной консоли Geth (см. *главу 2*).

[11] На момент подготовки перевода ссылка не работала. Воспользуйтесь общей ссылкой: **https://github.com/ethereum/**, чтобы выбрать нужную тему.

Вы получите на выходе следующий код:

```
var proofContract =
web3.eth.contract([{"constant":false,"inputs":[{"name":"fileHash","type":"s
tring"}],"name":"get","outputs":[{"name":"timestamp","type":"uint256"},{"na
me":"owner","type":"string"}],"payable":false,"type":"function"},{"constant
":false,"inputs":[{"name":"owner","type":"string"},{"name":"fileHash","type
":"string"}],"name":"set","outputs":[],"payable":false,"type":"function"},{
"anonymous":false,"inputs":[{"indexed":false,"name":"status","type":"bool"}
,{"indexed":false,"name":"timestamp","type":"uint256"},{"indexed":false,"na
me":"owner","type":"string"},{"indexed":false,"name":"fileHash","type":"str
ing"}],"name":"logFileAddedStatus","type":"event"}]);
var proof = proofContract.new(
        {
                from: web3.eth.accounts[0],
                data: '60606040526......,
                gas: 4700000
        }, function (e, contract){
                console.log(e, contract);
        if (typeof contract.address !== 'undefined') {
                console.log('Contract mined! address: ' + contract.address + '
        transactionHash: ' + contract.transactionHash);
        }
})
```

Блок `data` (выделен полужирным шрифтом) представляет собой байт-код (скомпилированную версию контракта), который понятен виртуальной машине Ethereum. Исходный код сначала конвертируется в операционный код (opcode, мнемонические коды операций), а затем в байт-код виртуальной машины. Каждый операционный код имеет связанный с ним газ.

Первый аргумент в `web3.eth.contract` — это определение ABI (Application Binary Interface, двоичный интерфейс приложения). Определение ABI применяется при создании транзакций и содержит объявление прототипов всех методов.

Теперь запустите Geth в режиме разработчика, *обязательно с включенным майнингом*. Для этого введите в окне командной строки[12]:

```
geth --dev --mine --etherbase 'здесь подставьте ваш адрес Ethereum'
```

Откройте еще одно окно командной строки и введите команду запуска интерактивной консоли JavaScript с подключением к работающему узлу:

```
geth attach
```

После этого консоль JavaScript должна подключиться к процессу, запущенному в другом окне.

[12] Новая версия Geth не запустит режим майнинга даже в отладочном режиме без указания вашего адреса в опции `–etherbase`.

В правой панели браузера Solidity скопируйте содержимое текстового поля **web3 deploy**, вставьте его в интерактивную консоль и нажмите клавишу <Enter>. Сначала вы получите *хеш транзакции*, а после некоторого ожидания, которое займет майнинг транзакции, вы получите *адрес контракта*. Каждый развернутый контракт имеет уникальный адрес, по которому определяется в блокчейне.

Адрес контракта определенным образом вычисляется из адреса создателя (адрес `from`) и количества отправленных транзакций (transaction nonce). Два этих числа подвергаются RLP-кодированию[13], а затем хешируются по алгоритму Keccak-256. Мы рассмотрим понятие *transaction nonce* позже. Про RLP-кодирование можно подробнее прочитать по адресу: **https://github.com/ethereum/wiki/wiki/RLP**.

Теперь давайте сохраним информацию о файле и запросим ее обратно.

Вставьте в консоль код из листинга 3.16, чтобы передать транзакцию сохранения информации о файле.

Листинг 3.16. Код для отправки транзакции с данными файла

```
var contract_obj =
proofContract.at("0x9220c8ec6489a4298b06c2183cf04fb7e8fbd6d4");
contract_obj.set.sendTransaction("Owner Name",
"e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855", {
from: web3.eth.accounts[0],
}, function(error, transactionHash){
if (!err)
console.log(transactionHash);
})
```

В этом листинге замените адрес контракта на адрес, который вы только что получили. Первый аргумент метода `proofContract.at()` — это и есть адрес контракта. Здесь мы не указали количество газа, в таком случае это значение вычисляется автоматически.

Теперь давайте извлечем информацию о файле. Введите в консоль этот код[14]:

```
contract_obj.get.call("e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852
b855");
```

Вы получите следующий ответ:

```
[1477591434, "Owner Name"]
```

Метод `call()` применяется для вызова метода контракта из текущего расположения. Он не транслирует транзакцию. Для чтения данных нам не требуется трансляция, потому что у нас есть собственная копия блокчейна.

Более подробно о пакете библиотек web3.js вы узнаете в следующих главах.

[13] Recursive Length Prefix, основной метод кодирования для сериализации объектов Ethereum.

[14] Вы можете скопировать хеш файла из листинга 3.16.

Заключение

В этой главе мы познакомились с языком программирования Solidity. Мы узнали о расположении данных, типах данных и расширенных свойствах контрактов. Мы опробовали самый быстрый и простой способ развертывания смарт-контракта. Теперь вы можете с комфортом работать над контрактами.

В следующей главе мы создадим оболочку для смарт-контракта, благодаря которой будет удобно запускать контракт и проводить транзакции.

4

Учимся работать с web3.js

Из предыдущей главы мы узнали, как писать смарт-контракты и использовать интерактивную консоль Geth для размещения и трансляции транзакций при помощи пакета библиотек web3.js. В этой главе мы более детально изучим web3.js и узнаем, как подключаться к Geth и использовать библиотеки web3.js в среде Node.js и в программах JavaScript на стороне клиента. Мы также узнаем, как при помощи web3.js создать клиентскую оболочку для смарт-контракта, разработанного нами в *главе 3*.

В этой главе мы рассмотрим следующие темы:

- импорт web3.js в среду Node.js и клиентские программы JavaScript;
- подключение к Geth;
- обзор возможностей, которые дает нам web3.js;
- исследование наиболее востребованных API web3.js;
- разработка в среде Node.js приложения для смарт-контракта.

Введение в web3.js

web3.js — это пакет библиотек, которые предоставляют нам API для взаимодействия с Geth на уровне протокола JSON-RPC[1]. Библиотеки web3.js могут также работать с любой другой реализацией узла Ethereum при условии поддержки протокола JSON-RPC. web3.js поддерживает не только API, относящиеся к узлу сети Ethereum, но и API приложений Whisper и Swarm.

---

[1] Remote Procedure Call (RPC) — протокол удаленного вызова процедур, не сохраняющий содержание вызова.

По мере того, как мы станем разрабатывать различные проекты, вы будете узнавать про web3.js все больше и больше. Но сейчас мы обсудим наиболее востребованные API пакета web3.js, а затем разработаем оболочку для нашего смарт-контракта с применением web3.js.

На момент работы над книгой актуальной была версия web3.js 0.16.0[2]. В дальнейшем мы будем подразумевать использование этой версии.

Пакет web3.js находится по адресу: **https://github.com/ethereum/web3.js**, а полная документация к нему доступна по адресу: **https://github.com/ethereum/wiki/wiki/JavaScript-API**.

Импортирование web3.js

Чтобы использовать web3.js в среде Node.js, вам достаточно, находясь внутри каталога проекта, выполнить команду:

```
npm install web3
```

В исходном же коде вы можете выполнить импорт web3.js при помощи директивы:

```
require("web3");
```

Чтобы использовать web3.js на стороне клиентского приложения JavaScript, вы можете подключить файл `web3.js`, который находится в каталоге `dist` исходного кода проекта. Теперь у вас есть объект `Web3`, доступный глобально.

Подключение к узлу

web3.js может взаимодействовать с узлами по протоколу HTTP или IPC. Для установления связи с узлом мы воспользуемся протоколом HTTP. Допускается установление связи с несколькими узлами, при этом каждый экземпляр `web3` представляет соединение с узлом и предоставляет набор API.

Когда приложение запущено внутри приложения-клиента Mist, оно автоматически создает доступный экземпляр `web3`, подключенный к узлу Mist. Имя переменной экземпляра `web3`.

Так выглядит базовый код для подключения к узлу:

```
if (typeof web3 !== 'undefined') {
       web3 = new Web3(new
Web3.providers.HttpProvider("http://localhost:8545"));
}
```

[2] Вероятно, автор имел в виду версию API web3 Ethereum, которая в период работы над переводом книги имела номер 0.20.0. Сам пакет web3.js во время подготовки книги имел версию 0.13.0, а во время работы над переводом была актуальна уже версия 0.14.0.

Сначала мы уточняем, запущен ли код внутри Mist. Для этого мы проверяем тип `web3` — является ли он неопределенным (`undefined`). Если тип `web3` определен, мы воспользуемся уже существующим экземпляром. В ином случае мы создаем экземпляр путем подключения к нашему пользовательскому узлу. Если вы хотите подключиться к узлу независимо от наличия работающего приложения внутри Mist, просто удалите условие `if` из приведенного здесь кода. В данном примере мы подразумеваем, что узел доступен локально через порт 8545.

Объект `Web3.providers` предоставляет конструкторы (в данном контексте называемые *провайдерами*), чтобы устанавливать соединения и передавать сообщения с использованием различных протоколов:

- `Web3.providers.HttpProvider` — устанавливает HTTP-соединение;
- `Web3.providers.IpcProvider` — устанавливает IPC-соединение.

Свойству `Web3.providers.IpcProvider` автоматически присваивается текущий экземпляр провайдера. После создания экземпляра `web3` вы можете изменить его провайдера при помощи метода `web3.setProvider()`. Он получает единственный аргумент — экземпляр нового провайдера.

По умолчанию при запуске Geth протокол HTTP-RPC отключен. Чтобы включить его, добавьте в строку запуска Geth опцию **`--rpc`**. По умолчанию HTTP-RPC использует порт 8545.

Метод `isConnected()` применяется для проверки наличия подключения к узлу и возвращает `true` или `false`.

Структура API

Экземпляр `web3` содержит объект `eth` (`web3.eth`), предназначенный специально для взаимодействия с блокчейном Ethereum, и объект `shh` (`web3.shh`) — для взаимодействия с Whisper. Большая часть API пакета web3.js представлена внутри двух этих объектов.

Все API по умолчанию синхронные. Если вы хотите выполнить асинхронный запрос, то для большинства функций API должны добавить функцию обратного вызова (callback-функцию) в качестве последнего параметра. Все функции обратного вызова используют формат вызова «error-first»[3].

Некоторые API имеют псевдоним для асинхронных запросов. Например, `web3.eth.coinbase()` является синхронным, а `web3.eth.getCoinbase()` — асинхронным.

[3] Простое описание формата вызова «error-first» читатели могут найти по адресу: **http://fredkschott.com/post/2014/03/understanding-error-first-callbacks-in-node-js**. Если коротко, первый аргумент функции обратного вызова зарезервирован под объект ошибки `err`. Когда случается ошибка, функция возвращает объект `err`, который можно использовать для получения описания ошибки.

Приведем простой пример:

```
//синхронный запрос
try
{
        console.log(web3.eth.getBlock(48));
}
catch(e)
{
        console.log(e);
}

//асинхронный запрос
web3.eth.getBlock(48, function(error, result){
        if(!error)
                console.log(result)
        else
                console.error(error);
})
```

Метод `getBlock()` возвращает информацию о блоке по его порядковому номеру или хешу. Также можно вместо номера использовать ключевые слова `earliest` (генезисный блок), `latest` (последний добавленный блок) или `pending` (блок в майнинге). Если вы опускаете аргумент, по умолчанию запрашивается `web3.eth.defaultBlock`, которому соответствует псевдоним `latest`.

Все API, которые нуждаются в идентификации блока, могут принимать в качестве входного аргумента число, хеш или строку. Если аргумент отсутствует, все эти API по умолчанию используют `web3.eth.defaultBlock`.

Библиотека BigNumber.js

Язык JavaScript изначально плохо работает с большими числами. Поэтому приложения, которым приходится иметь дело с большими числами и делать точные вычисления, используют библиотеку BigNumber.js.

Пакет web3.js тоже зависит от BigNumber.js (эта библиотека добавляется автоматически). web3.js всегда возвращает объект типа `BigNumber` для числовых значений. Он может принимать числа JavaScript, строковые представления чисел и экземпляры `BigNumber` в качестве входного параметра.

Вот пример, который это демонстрирует:

```
web3.eth.getBalance("0x27E829fB34d14f3384646F938165dfcD30cFfB7c").toString();
```

Здесь мы используем метод `web3.eth.getBalance()`, чтобы получить баланс указанного адреса (счета) Ethereum. Этот метод возвращает объект типа `BigNumber`. Мы должны вызвать метод `toString()` объекта `BigNumber`, чтобы конвертировать его в строковое представление числа.

Библиотека BigNumber.js не может обрабатывать числа, у которых более 20 знаков после запятой, поэтому рекомендуется хранить баланс в единицах Wei, а для отображения конвертировать в другие единицы. По умолчанию web3.js всегда возвращает и получает баланс в Wei. Например, метод `getBalance()` возвратит баланс счета в Wei.

Конвертация денежных единиц

web3.js предоставляет возможность конвертировать Wei в другие номиналы эфира и, наоборот, любой другой номинал эфира конвертировать в Wei.

Метод `web3.fromWei()` предназначен для конвертации суммы, номинированной в Wei в другой номинал, а метод `web3.toWei()` выполняет обратную операцию. Вот два примера, которые демонстрируют конвертацию:

```
web3.fromWei("1000000000000000000", "ether");
web3.toWei("0.000000000000000001", "ether");
```

В первой строке мы конвертируем Wei в Ether (один эфир), а во второй строке конвертируем Ether в Wei. Второй аргумент в обоих методах может быть одной из следующих строк:

- `kwei/ada`;
- `mwei/babbage`;
- `gwei/shannon`;
- `szabo`;
- `finney`;
- `ether`;
- `kether/grand/einstein`;
- `mether`;
- `gether`;
- `tether`.

Запрос цены газа, баланса и деталей транзакции

Теперь давайте рассмотрим API, применяемые для запроса цены газа, баланса на счете и информации о транзакции (квитанции):

```
//Это синхронный запрос.
//Для асинхронного запроса используйте getGasPrice
console.log(web3.eth.gasPrice.toString());

console.log(web3.eth.getBalance("0x407d73d8a49eeb85d32cf465507dd71d507100c1",
45).toString());

console.log(web3.eth.getTransactionReceipt("0x9fc76417374aa880d4449a1f7f31ec597f00b
1f6f3dd2d66f4c9c6c445836d8b"));
```

Возвращаемая этим запросом информация имеет следующий вид:

```
20000000000
30000000000
```

```
{
"transactionHash":
"0x9fc76417374aa880d4449a1f7f31ec597f00b1f6f3dd2d66f4c9c6c445836d8b ",
"transactionIndex": 0,
"blockHash":
"0xef95f2f1ed3ca60b048b4bf67cde2195961e0bba6f70bcbea9a2c4e133e34b46",
"blockNumber": 3,
"contractAddress": "0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b",
"cumulativeGasUsed": 314159,
"gasUsed": 30234
}
```

Рассмотрим подробнее, как работает приведенный здесь метод:

- `web3.eth.gasPrice()` — возвращает цену газа, как медианное значение цены для `x` последних блоков;
- `web3.ethgetBalance()` — возвращает баланс указанного счета (адреса). Все хеши должны быть отправлены в API web3.js как шестнадцатеричные строки, а не шестнадцатеричные литералы. Входные значения для типа `address` в Solidity тоже должны быть шестнадцатеричными строками;
- `web3.eth.getTransactionReceipt()` — применяется для получения информации о транзакции (квитанции) с использованием хеша. Он возвращает объект квитанции, если транзакция была найдена в блокчейне. В противном случае метод возвращает `null`. Объект квитанции содержит следующие свойства:
  - `blockHash` — хеш блока, в который помещена транзакция;
  - `blockNumber` — номер блока, в который помещена транзакция;
  - `transactionHash` — хеш транзакции;
  - `transactionIndex` — целое число, индекс позиции в блоке;
  - `from` — адрес отправителя;
  - `to` — адрес получателя, `null` — если это транзакция создания блока;
  - `cumulativeGasUsed` — совокупное количество газа, которое расходуется при выполнении этой транзакции в блоке;
  - `gasUsed` — количество газа, расходуемое при выполнении данной одиночной транзакции;
  - `contractAddress` — адрес контракта, созданного транзакцией, если это была транзакция, создающая контракт. В противном случае `null`;
  - `logs` — массив объектов лога, который сгенерировала данная транзакция.

Отправка эфира

Давайте научимся отправлять эфир на любой адрес. Для перевода эфира вам потребуется метод `web3.eth.sendTransaction()`. Этот метод можно использовать для отправки любых транзакций, но в основном его применяют для перевода эфира, потому что развертывание контракта или вызов метода контракта при помощи этого метода являются громоздкими и требуют самостоятельной генерации данных транзакции вместо автоматического создания. Метод принимает объект транзакции, который обладает следующими свойствами:

- `from` — адрес счета отправителя. Если не указан, по умолчанию используется свойство `web3.eth.defaultAccount`;
- `to` — необязательное свойство. Остается неопределенным для транзакций создания контракта;
- `value` — необязательное свойство. Сумма перевода в Wei либо сумма вознаграждения при создании контракта;
- `gas` — необязательное свойство. Количество газа, расходуемого на транзакцию (неиспользованный газ возвращается). Если не указано, задается автоматически;
- `gasPrice` — необязательное свойство. Цена газа для данной транзакции в Wei, которая определяется по значению цены газа в сети;
- `data` — необязательное свойство. Это либо байтовая строка, содержащая присоединенное сообщение, либо — в случае создания контракта — код инициализации;
- `nonce` — необязательное свойство. Каждая транзакция имеет связанное с ней целое число `nonce`. Это счетчик, который показывает число транзакций, переданных отправителем транзакции. Если свойство не определено, то задается автоматически. Это свойство позволяет предотвратить атаку массовыми транзакциями. Напомним, что данное свойство `nonce` — это вовсе не число `nonce`, которое имеет отношение к майнингу блока. Если мы используем значение `nonce`, которое больше, чем должна была бы иметь транзакция, то эта транзакция встает в очередь, пока не поступят другие транзакции. Например, если свойство `nonce` следующей транзакции должно иметь значение `4`, а мы установили значение `10`, то Geth будет ждать шесть промежуточных транзакций, прежде чем транслировать данную транзакцию. Поэтому наша транзакция, у которой `nonce` равно десяти, называется *отложенной транзакцией* (queued transaction), и не является *транзакцией в обработке* (pending transaction).

Вот простой пример того, как отправляют эфир по заданному адресу:

```
var txnHash = web3.eth.sendTransaction({
        from: web3.eth.accounts[0],
        to: web3.eth.accounts[1],
        value: web3.toWei("1", "ether")
});
```

Здесь мы переводим один эфир со счета номер `0` на счет номер `1`. Убедитесь, что оба счета разблокированы опцией **`--unlock`** при запуске Geth. В интерактивной консоли Geth будут запрошены пароли, но API web3.js вне интерактивной консоли вернут ошибку, если счет заблокирован. Этот метод возвращает хеш транзакции. Затем вы при помощи метода `getTransactionReceipt()` можете проверить, обработана ли транзакция майнером.

Вы также можете использовать API: `web3.personal.listAccounts()`, `web3.personal.unlockAccount(addr, pwd)` и `web3.personal.newAccount(pwd)` для динамического управления счетами.

Работа с контрактами

Теперь давайте рассмотрим работу с контрактами — развертывание нового контракта, получение ссылки на контракт по его адресу, отправку эфира на контракт, отправку транзакции, вызывающей метод контракта, и оценку количества газа, нужного для вызова метода.

Для развертывания нового контракта или для получения ссылки на уже развернутый контракт вам надо, прежде всего, создать объект контракта при помощи метода `web3.eth.contract()`. Он получает ABI контракта как аргумент и возвращает объект контракта.

Так выглядит код[4] для создания объекта контракта:

```
var proofContract =
web3.eth.contract([{"constant":false,"inputs":[{"name":"fileHash","type":"s
tring"}],"name":"get","outputs":[{"name":"timestamp","type":"uint256"},{"na
me":"owner","type":"string"}],"payable":false,"type":"function"},{"constant
":false,"inputs":[{"name":"owner","type":"string"},{"name":"fileHash","type
":"string"}],"name":"set","outputs":[],"payable":false,"type":"function"},{
"anonymous":false,"inputs":[{"indexed":false,"name":"status","type":"bool"}
,{"indexed":false,"name":"timestamp","type":"uint256"},{"indexed":false,"na
me":"owner","type":"string"},{"indexed":false,"name":"fileHash","type":"str
ing"}],"name":"logFileAddedStatus","type":"event"}]);
```

Когда ваш контракт готов, вы можете развернуть его при помощи метода `new()` объекта контракта или получить ссылку на уже развернутый контракт при помощи метода `at()`.

Давайте рассмотрим пример того, как развертывают новый контракт:

```
var proof = proofContract.new({
        from: web3.eth.accounts[0],
        data: "0x606060405261068...",
        gas: "4700000"
        },
```

[4] О том, как получить этот код, рассказано в *главе 3*.

```
        function (e, contract){
        if(e)
        {
        console.log("Error " + e);
}
else if(contract.address != undefined)
        {
                console.log("Contract Address: " + contract.address);
        }
else
        {
                console.log("Txn Hash: " + contract.transactionHash)
        }
})
```

В данном случае метод `new()` вызывается асинхронно, поэтому, если транзакция создана и успешно передана в сеть, обратный вызов срабатывает дважды: сначала — когда транзакция передана, а затем — когда она обработана майнером. Если вы не предусмотрели обратный вызов, то поле `address` переменной `proof` будет установлено в `undefined`. Как только майнинг контракта завершен, присваивается значение полю `address`.

В нашем контракте `proof` нет конструктора, но если конструктор имеется, то аргументы конструктора должны быть помещены в начало метода `new()`. Объект, который мы передали, содержит адрес `from`, байт-код контракта и доступное количество газа. Эти три параметра должны быть обязательно заданы, иначе транзакция не будет создана. В данном случае `data` — это байт-код контракта, а свойство `to` игнорируется.

Для получения ссылки на развернутый контракт вы можете воспользоваться методом `at()`. Пример кода, который демонстрирует получение ссылки:

```
var proof = proofContract.at("0xd45e541ca2622386cd820d1d3be74a86531c14a1");
```

Теперь посмотрим, как отправить транзакцию, которая вызывает метод контракта. Вот простой пример, демонстрирующий эту процедуру:

```
proof.set.sendTransaction("Owner Name",
"e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855", {

from: web3.eth.accounts[0],
}, function(error, transactionHash){

if (!err)

console.log(transactionHash);
})
```

Здесь мы вызываем метод `sendTransaction()`. Объект, переданный этому методу, имеет такие же параметры, как и объект, передаваемый методу `web3.eth.sendTransaction()`, за исключением того, что игнорируются параметры `data` и `to`.

Если вы хотите вызывать некий метод непосредственно на своем узле, вместо того, чтобы создавать и транслировать транзакцию, то вместо `sendTransaction()` используйте `call()`, например:

```
var returnValue =
proof.get.call("e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855");
```

Иногда необходимо узнать, какое количество газа потребуется для вызова метода, чтобы принять решение, стоит ли вообще вызывать этот метод. Для решения такой оценочной задачи можно использовать `web3.eth.estimateGas()`. Однако прямое использование этого метода вынуждает вас генерировать данные для транзакции. Следовательно, мы можем воспользоваться методом `estimateGas()` нашего объекта, например:

```
var estimatedGas =
proof.get.estimateGas("e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852
b855");
```

Если вы хотите просто перевести несколько эфиров на контракт без вызова каких-либо методов, то можете напрямую воспользоваться методом `web3.eth.sendTransaction`.

Отслеживание событий контракта

Сейчас мы узнаем, как отслеживать события, которые генерирует контракт. Отслеживание событий имеет большое значение, потому что результаты вызова методов обычно возвращаются путем переключения триггеров событий.

Прежде, чем приступить к изучению отслеживания событий, нам следует узнать, что такое *индексированные параметры событий*. Иметь атрибут `indexed` могут не более трех параметров события. Этот атрибут применяется для уведомления узла о том, что клиентское приложение может искать события по совпадению возвращаемых значений. Если вы не используете атрибут `indexed`, вам придется запрашивать все события узла и отфильтровывать нужные. Например, вы можете описать событие `logFileAddedStatus` таким образом:

```
event logFileAddedStatus(bool indexed status, uint indexed timestamp,
string owner, string indexed fileHash);
```

Пример, приведенный в листинге 4.1, демонстрирует, как прослушивать события контракта.

Листинг 4.1. Прослушивание событий контракта

```
var event = proof.logFileAddedStatus(null, {
fromBlock: 0,
toBlock: "latest"
});
event.get(function(error, result){
if(!error)
{
        console.log(result);
}
else
{
        console.log(error);
}
})
event.watch(function(error, result){
if(!error)
{
        console.log(result.args.status);
}
else
{
        console.log(error);
}
})
setTimeout(function(){
event.stopWatching();
}, 60000)

var events = proof.allEvents({
fromBlock: 0,
toBlock: "latest"
});
events.get(function(error, result){
if(!error)
{
        console.log(result);
}
else
{
        console.log(error);
}
})
```

```
events.watch(function(error, result){
if(!error)
{
        console.log(result.args.status);
}
else
{
        console.log(error);
}
})
setTimeout(function(){
events.stopWatching();
}, 60000)
```

Давайте рассмотрим подробнее, как работает этот код:

1. Сначала мы получаем объект события путем вызова метода, одноименного с экземпляром контракта. Этот метод получает два объекта как аргументы, которые использует для фильтрации событий:
   - первый объект служит для фильтрации событий по индексированным значениям — например: `{'valueA': 1, 'valueB': [myFirstAddress, mySecondAddress]}`. По умолчанию все значения фильтров установлены в `null`. Это означает, что они будут совпадать со всеми событиями данного типа, поступающими от этого контракта;
   - второй объект может содержать три свойства: `fromBlock` (самый старый блок, по умолчанию `"latest"`), `toBlock` (самый новый блок, по умолчанию `"latest"`) и `address` (список адресов, откуда следует получать логи событий, по умолчанию это адрес контракта).
2. Объект `event` предоставляет нам три метода: `get`, `watch` и `stopWatching`. Метод `get` применяется для получения всех событий в заданном диапазоне блоков. Метод `watch` подобен методу `get`, но следит за изменениями после получения событий. Метод `stopWatching` прекращает отслеживание изменений.
3. Далее, у нас есть метод `allEvents` экземпляра контракта. Он служит для извлечения всех событий контракта.
4. Каждое событие представлено объектом, который имеет следующие свойства:
   - `args` — объект аргументов, которые предоставило событие;
   - `event` — строка, содержащая имя события;
   - `logIndex` — целое число, представляющее позицию индекса (указателя) лога в блоке;
   - `transactionIndex` — целое число, представляющее транзакцию, по которой был сгенерирован лог;

- `transactionHash` — строка, представляющая хеш транзакции, по которой был сгенерирован данный лог;
- `address` — строка, представляющая адрес, из которого поступил лог;
- `blockHash` — строка, представляющая хеш блока, в котором был данный лог, и `null`, если блок в ожидании подтверждения;
- `blockNumber` — номер блока, в котором был лог, и `null`, если блок в ожидании подтверждения.

Пакет web3.js предоставляет API `web3.eth.filter` для извлечения и просмотра событий. Вы можете использовать этот API, но описанный ранее метод намного проще. Про это можно прочитать больше по адресу: **https://github.com/ethereum/wiki/wiki/JavaScript-API#web3ethfilter**.

Разработка клиентского приложения для контракта

В *главе 3* мы написали код на языке Solidity, а также на протяжении двух последних глав мы изучаем web3.js и вызов методов при помощи web3.js. Настало время написать клиентскую оболочку для нашего смарт-контракта, чтобы пользователям было легче им пользоваться.

Мы разработаем клиент, в котором пользователь выбирает файл, вводит данные владельца файла и нажимает кнопку **Submit** (Отправить), чтобы передать в сеть транзакцию, вызывающую метод `set` с хешем и данными владельца. Когда транзакция успешно передана, мы выведем на дисплей ее хеш. Кроме того, пользователь сможет выбрать файл и получить для него сведения о владельце из смарт-контракта. А еще клиент будет показывать последние транзакции `set`, подтвержденные в режиме реального времени.

Мы воспользуемся при разработке следующими фреймворками и модулями:

- sha1.js — для получения хеша файла на стороне клиента;
- jQuery — для манипуляции объектами DOM;
- Bootstrap 4 — для создания адаптивного макета.

На серверной стороне мы задействуем express.js и web3.js. При этом нам надо сделать так, чтобы серверная часть отправляла подтвержденные транзакции клиентскому интерфейсу без постоянных запросов со стороны клиента. Для этого мы воспользуемся сервисом **socket.io**.

Пакет web3.js может быть задействован и на клиентской стороне. Но в данном случае это создаст проблемы с безопасностью. Мы пользуемся счетами, сохраненными в Geth, и раскрываем URL-адрес узла Geth клиентскому интерфейсу, который создает угрозу для эфира, хранящегося на других счетах.

Структура проекта

В папке файлов упражнений, дополняющих эту главу (см. *приложение*), вы найдете два каталога: `Final` и `Initial`. Каталог `Final` содержит окончательный исходный код проекта, в то время как `Initial` содержит пустые файлы исходного кода и библиотеки, что позволяет быстро начать работу над проектом.

Для проверки каталога `Final` вам понадобится выполнить команду терминала **`npm install`** внутри каталога и заменить заданный в коде файла `app.js` адрес контракта на адрес, полученный вами после развертывания вашего контракта. После этого запустите приложение командой терминала **`node app.js`** внутри каталога `Final`.

В каталоге `Initial` находится вложенный каталог `public` и два файла: `app.js` и `package.json`. Второй файл содержит зависимости для вашего приложения, а файл `app.js` — это место, в котором будет храниться серверная часть проекта.

Каталог `public` содержит файлы, относящиеся к пользовательскому интерфейсу (клиентская часть). Внутри `public/css` вы найдете библиотеку фреймворка Bootstrap `bootstrap.min.css`, а внутрь `public/html` вы поместите HTML-код вашего клиентского приложения. В каталоге `public/js` находятся JavaScript-файлы для jQuery, sha1 и socket.io. Также внутри этого каталога вы найдете файл `main.js`, в который вы поместите код JavaScript для клиентской части проекта.

Разработка серверной части

Давайте сначала создадим серверную часть нашего приложения. Первым делом выполним команду **`npm install`** внутри каталога `Initial` — чтобы установить необходимые зависимости для серверной части. Прежде чем приступить к написанию кода серверной части, убедитесь, что Geth работает, а RPC включен. Если вы запустили Geth в частной сети, также убедитесь, что включен режим майнинга. Наконец, удостоверьтесь, что счет 0 существует и разблокирован. Вы можете запустить Geth в частной сети с включенным RPC и майнингом и разблокированным счетом 0 при помощи единственной команды:

```
geth --dev --mine --rpc --unlock=0
```

Осталось лишь развернуть в сети контракт, о котором мы говорили в *главе 3*, и скопировать адрес контракта.

Теперь создадим автономный сервер (single server), который будет обслуживать HTML-запросы браузера и принимать подключения сервиса socket.io:

```
var express = require("express");
var app = express();
var server = require("http").createServer(app);
var io = require("socket.io")(server);
server.listen(8080);
```

Здесь мы объединили серверы `express` и `socket.io` в один сервер с портом `8080`.

Далее мы объявим пути к статическим файлам и домашней странице приложения. Вот код, который это делает:

```
app.use(express.static("public"));
app.get("/", function(req, res){
res.sendFile(__dirname + "/public/html/index.html");
})
```

Здесь мы используем промежуточный интерфейс `express.static` для доступа к статическим файлам и указываем ему искать статические файлы в каталоге `public`.

Теперь подключимся к узлу Geth и получим ссылку на развернутый контракт, благодаря чему сможем отправлять транзакции и следить за событиями. Вот соответствующий фрагмент кода:

```
var Web3 = require("web3");
web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
var proofContract =
web3.eth.contract([{"constant":false,"inputs":[{"name":"fileHash","type":"s
tring"}],"name":"get","outputs":[{"name":"timestamp","type":"uint256"},{"na
me":"owner","type":"string"}],"payable":false,"type":"function"},{"constant
":false,"inputs":[{"name":"owner","type":"string"},{"name":"fileHash","type
":"string"}],"name":"set","outputs":[],"payable":false,"type":"function"},{
"anonymous":false,"inputs":[{"indexed":false,"name":"status","type":"bool"}
,{"indexed":false,"name":"timestamp","type":"uint256"},{"indexed":false,"na
me":"owner","type":"string"},{"indexed":false,"name":"fileHash","type":"str
ing"}],"name":"logFileAddedStatus","type":"event"}]);
var proof = proofContract.at("0xf7f02f65d5cd874d180c3575cb8813a9e7736066");
```

Мы уже несколько раз упоминали эту часть кода раньше, и она не требует пояснений. Просто замените номер контракта на тот, который получили сами.

Сформируем пути для вещания транзакций и получения сведений о файле:

```
app.get("/submit", function(req, res){
var fileHash = req.query.hash;
var owner = req.query.owner;
proof.set.sendTransaction(owner, fileHash, {
from: web3.eth.accounts[0],
}, function(error, transactionHash){
if (!error)
{
        res.send(transactionHash);
}
else
{
        res.send("Error");
}
})
})
```

```
app.get("/getInfo", function(req, res){
var fileHash = req.query.hash;
var details = proof.get.call(fileHash);
res.send(details);
})
```

В данном случае для создания и трансляции транзакции мы используем путь `/submit`. Как только станет известен хеш транзакции, мы сразу отправим его в клиентскую часть приложения, — мы не ждем, пока транзакция находится в процессе майнинга. Путь `/getInfo` служит для вызова метода контракта, расположенного на нашем узле, вместо того, чтобы создавать транзакцию. Он просто пересылает ответ, который получил от узла.

Теперь приступим к слежению за событиями контракта и трансляции их всем подключившимся клиентам:

```
proof.logFileAddedStatus().watch(function(error, result){
if(!error)
{
        if(result.args.status == true)
        {
                io.send(result);
        }
}
})
```

Здесь мы проверяем статус результата. Если он равен `true`, то единственное, что мы делаем, — пересылаем событие подключенным клиентам сервиса socket.io.

Разработка клиентской части

Приступим к разработке HTML-кода приложения. Поместите в файл `index.html` код из листинга 4.2.

Листинг 4.2. HTML-код клиентского интерфейса

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
<link rel="stylesheet" href="/css/bootstrap.min.css">
</head>
<body>
<div class="container">
<div class="row">
<div class="col-md-6 offset-md-3 text-xs-center">
```

```
<br>
<h3>Upload any file</h3>
<br>
<div>
<div class="form-group">
<label class="custom-file text-xs-left">
<input type="file" id="file" class="customfile-
input">
<span class="custom-file-control"></span>
</label>
</div>
<div class="form-group">
<label for="owner">Введите имя владельца</label>
<input type="text" class="form-control"
id="owner">
</div>
<button onclick="submit()" class="btn btnprimary">Submit</button>
<button onclick="getInfo()" class="btn btnprimary">Get Info</button>
<br><br>
<div class="alert alert-info" role="alert"
id="message">
Вы можете ввести сведения о файле либо получить их.
</div>
</div>
</div>
</div>
<div class="row">
<div class="col-md-6 offset-md-3 text-xs-center">
<br>
<h3>Транзакции, подтвержденные недавно</h3>
<br>
<ol id="events_list">Транзакции не найдены</ol>
</div>
</div>
</div>
<script type="text/javascript" src="/js/sha1.min.js"></script>
<script type="text/javascript" src="/js/jquery.min.js"></script>
<script type="text/javascript" src="/js/socket.io.min.js"></script>
<script type="text/javascript" src="/js/main.js"></script>
</body>
</html>
```

Кратко поясним, как работает этот код:

1. Сначала мы отображаем поле ввода, в котором пользователь может выбрать файл.

2. Затем мы отображаем текстовое поле, в которое пользователь может ввести данные владельца файла.
3. Далее, у нас есть две кнопки: первая кнопка сохраняет хеш файла и данные пользователя в контракт, а вторая кнопка предназначена для получения сведений о файле из контракта. Нажатие на кнопку **Submit** вызывает метод `submit()`, а нажатие на кнопку **Get Info**, соответственно, вызывает метод `getInfo()`.
4. В окне приложения предусмотрена область, в которой отображаются сообщения.
5. Наконец, мы отображаем упорядоченный список транзакций контракта, которые были подтверждены, пока пользователь находится на странице приложения.

Теперь напишем реализации для методов `getInfo()` и `submit()`, установления соединения с socket.io и прослушивания сообщений socket.io, поступающих с сервера. Далее приведен код, который выполняет указанные задачи. Он должен находиться в файле `main.js` (листинг 4.3).

Листинг 4.3. Содержимое файла `main.js`

```
function submit()
{
        var file = document.getElementById("file").files[0];
        if(file)
        {
                var owner = document.getElementById("owner").value;
                if(owner == "")
                {
                        alert("Введите имя владельцаа");
                }
        else
        {
                var reader = new FileReader();
                reader.onload = function (event) {
                var hash = sha1(event.target.result);
                $.get("/submit?hash=" + hash + "&owner=" + owner, function(data){
                if(data == "Error")
                {
                        $("#message").text("Произошла ошибка.");
                }
                else
                {
                        $("#message").html("Хеш транзакции: " + data);
                }
        });
        };
```

```
        reader.readAsArrayBuffer(file);
                }
        }
                else
                {
                        alert("Выберите файл.");
                }
}

function getInfo()
{
        var file = document.getElementById("file").files[0];
        if(file)
        {
        var reader = new FileReader();
        reader.onload = function (event) {
        var hash = sha1(event.target.result);
        $.get("/getInfo?hash=" + hash, function(data){
        if(data[0] == 0 && data[1] == "")
        {
                $("#message").html("Файл не найден");
        }
        else
        {
                $("#message").html("Метка времени: " + data[0] + " Владелец: " +
data[1]);
        }
        });
};
        reader.readAsArrayBuffer(file);
}
else
        {
                alert("Please select a file");
        }
}
var socket = io("http://localhost:8080");
socket.on("connect", function () {
socket.on("message", function (msg) {
if($("#events_list").text() == "Транзакции не найдены.")
{
        $("#events_list").html("<li>Txn Hash: " + msg.transactionHash +
        "nOwner: " + msg.args.owner + "nFile Hash: " + msg.args.fileHash +
        "</li>");
}
```

```
else
{
        $("#events_list").prepend("<li>Txn Hash: " + msg.transactionHash +
        "nOwner: " + msg.args.owner + "nFile Hash: " + msg.args.fileHash +
        "</li>");
}
        });
});
```

Код из листинга 4.3 работает следующим образом:

1. Сначала мы определяем метод `submit()`. В этом методе мы убеждаемся, что выбранный файл существует, и текстовое поле не пустое. Затем мы считываем содержимое файла в буферный массив и передаем этот массив методу `sha1()`, предоставленному библиотекой sha1.js, чтобы получить хеш содержимого буферного массива. Когда получен хеш, мы используем фреймворк jQuery для формирования AJAX-запроса по адресу `/route`, после чего отображаем хеш транзакции в блоке сообщений.
2. Затем мы определяем метод `getInfo()`. Прежде всего он проверяет, выбран ли файл. Далее он генерирует хеш аналогично тому, как это делалось в первый раз, и отправляет запрос по адресу `/getInfo` для получения сведений о файле.
3. Наконец, мы устанавливаем соединение с сервисом socket.io при помощи метода `io()` из библиотеки socket.io. Ждем, пока сработает триггер по событию соединения. Это событие означает, что соединение установлено. Теперь мы слушаем сообщения сервера и отображаем пользователю информацию о транзакциях.

Мы не сохраняем файл в блокчейн Ethereum, потому что хранение файлов обходится очень дорого и расходует большое количество газа. В данном случае нам и не нужно хранить файл, потому что каждый узел сети сможет его увидеть. Следовательно, вы не сможете сохранить в секрете содержимое файла, если выложите его в блокчейн. Назначение нашего приложения — удостоверять сведения о владельце, а не хранить файлы в облачном сервисе.

Тестирование клиентской части

Запустите приложение `app.js` в качестве серверной части приложения. Откройте ваш любимый браузер и перейдите по адресу: **http://localhost:8080/**. В окне браузера отобразится интерфейс пользователя (рис. 4.1).

Теперь выберите файл, введите имя владельца и нажмите кнопку **Submit**. Содержимое экрана изменится и будет выглядеть, как показано на рис. 4.2.

Вы можете видеть, что отобразился хеш транзакции. Теперь подождите, пока транзакция не пройдет процесс майнинга. Когда майнинг завершен, вы можете увидеть ее хеш в «живом» списке обработанных транзакций (рис. 4.3).

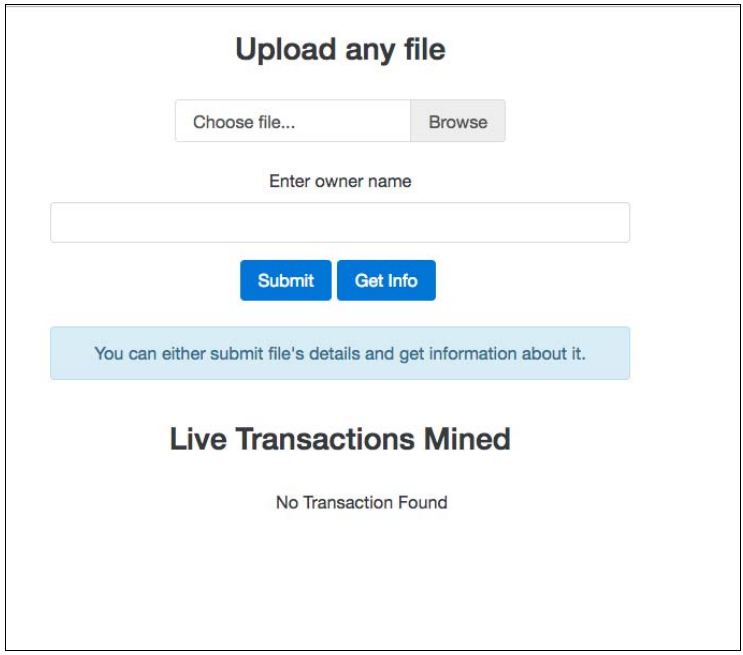

Рис. 4.1. Интерфейс пользователя: начальная страница

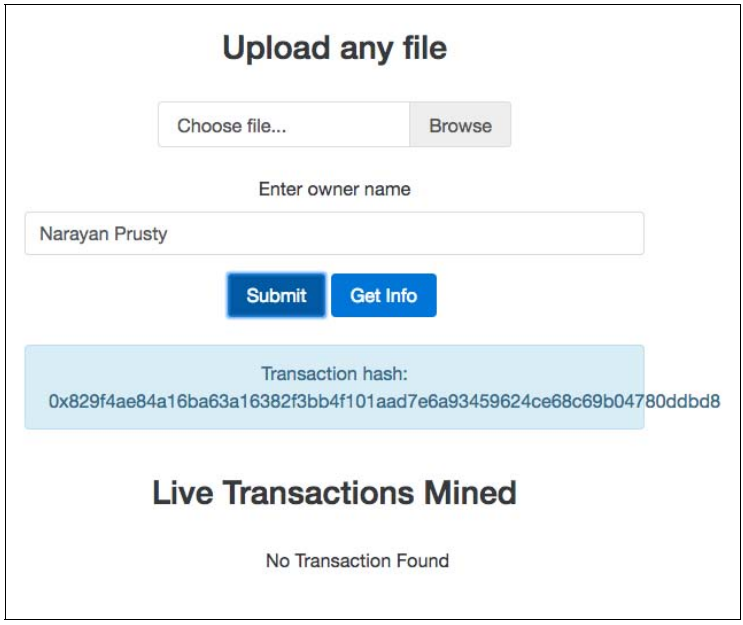

Рис. 4.2. Отображение хеша транзакции после отправки

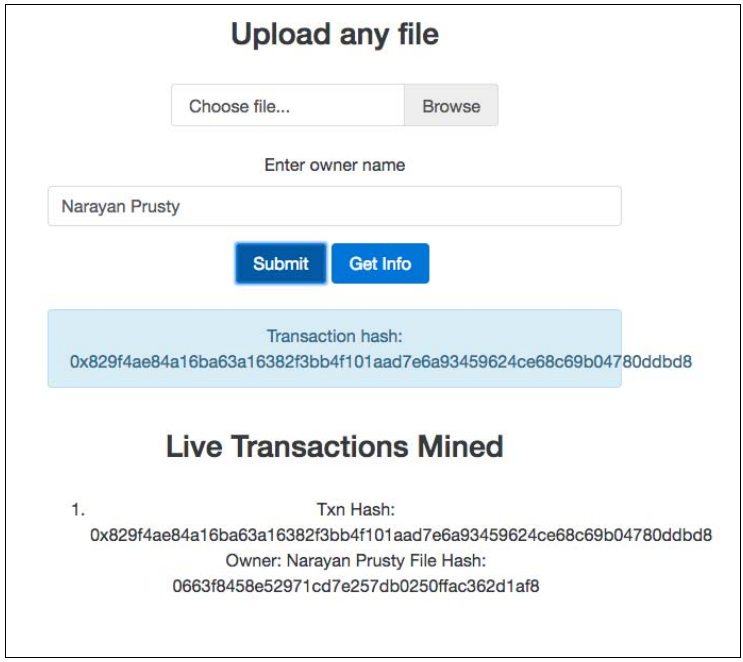

Рис. 4.3. Отображение хеша и сведений о владельце файла

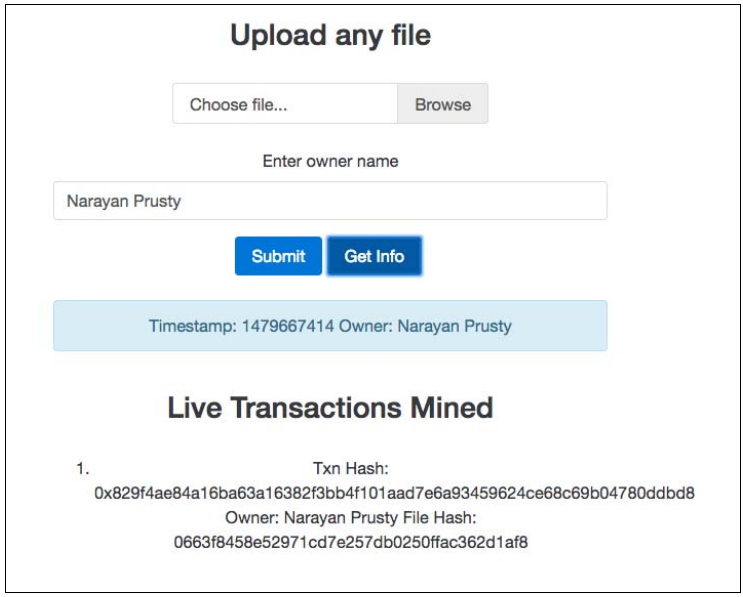

Рис. 4.4. Сведения о файле, хранящиеся в блокчейне

Теперь снова выберите тот же файл и нажмите кнопку **Get Info** — вы увидите сведения о файле (рис. 4.4). Эти сведения содержат метку времени и сведения о владельце.

Итак, мы закончили работу над клиентской частью нашего первого децентрализованного приложения!

Заключение

В этой главе мы на примерах изучили основы web3.js. Мы узнали, как подключиться к узлу, освоили базовые API и отправку различных типов транзакций, а также научились отслеживать события. В завершение главы мы разработали клиентское приложение для нашего контракта, заверяющего владение файлом. Теперь вам будет проще писать смарт-контракты и разрабатывать для них интерфейсные оболочки, которые значительно облегчают пользование контрактами.

В следующей главе мы разработаем сервис бумажника (wallet), благодаря которому пользователям будет проще создавать бумажники Ethereum Wallet и управлять ими даже без доступа к сети.

5

Разработка сервиса кошелька

Сервис кошелька предназначен для отправки и получения средств. Главной задачей при создании такого сервиса является обеспечение безопасности и доверия: пользователь должен быть уверен, что его средства находятся в безопасности и что их не украдет администратор сервиса. Приложение, которое мы с вами разработаем, решает обе проблемы.

В этой главе мы рассмотрим следующие темы:

- различие между онлайн- и оффлайн-кошельком;
- использование библиотек hooked-web3-provider и ethereumjs-tx для простого создания и подписания транзакций из аккаунтов, которые не расположены непосредственно на узле Ethereum;
- понимание того, что такое HD-кошелек и как им пользоваться;
- создание HD-кошелька и заверителя транзакций при помощи библиотеки LightWallet;
- создание сервиса кошелька.

Различие между онлайн- и оффлайн-кошельками

Кошелек — это набор счетов, а счет — это комбинация адреса и связанного с ним закрытого ключа.

Мы называем кошелек *онлайновым*, если он подключен к Интернету. Например, кошельки, которые хранятся в Geth, и им подобные являются онлайн-кошельками. Такие кошельки не рекомендуется использовать для хранения большого количества эфира или хранить в них эфир в течение длительного времени, потому что это рис-

кованно. Более того — в зависимости от того, где хранится сам кошелек, — возможно, придется довериться третьей стороне.

Например, многие популярные сервисы кошельков хранят закрытые ключи кошельков у себя и предоставляют вам доступ при помощи электронной почты и пароля. По сути, вы не имеете полного контроля над своим кошельком, и они могут украсть ваши деньги, когда захотят.

И наоборот, если кошелек не имеет соединения с Интернетом, мы называем его *оффлайновым*. Это, например, кошельки, которые хранятся на флэш-накопителе, в виде записей на бумаге, в виде текстовых файлов и т. п.[1]. Оффлайн-кошельки еще называют «холодными» (cold wallet). Оффлайн-кошельки более безопасны, потому что для кражи средств необходимо получить физический доступ к хранилищу. Проблема использования оффлайн-кошелька состоит в том, что вы должны найти такие место и способ хранения, которые исключают случайное его уничтожение, утерю или доступ к нему посторонних. Многие люди хранят данные кошелька на бумаге и кладут эти бумаги в сейф, который обеспечивает безопасное хранение в течение длительного времени. Если вам надо часто отправлять средства со своего счета, вы записываете файлы кошелька на защищенный паролем флэш-накопитель, который все равно кладете в сейф. Риск хранения кошелька на цифровом носителе немного выше, потому что электронное устройство может в любой момент сломаться, и вы потеряете доступ к данным на нем. Вот почему имеет смысл хранить в сейфе также и копию устройства. Вы можете найти и лучший способ, исходя из своих потребностей, но в любом случае надо быть уверенным, что кошелек лежит в безопасном месте, а вы не потеряете доступ к нему.

Библиотеки hooked-web3-provider и ethereumjs-tx

До сих пор все примеры метода `sendTransaction()` из библиотеки web3.js использовали адрес `from`, который находится в узле Ethereum, — следовательно, узел сети Ethereum мог подписывать транзакции перед трансляцией в сеть. Но если ваш закрытый персональный ключ хранится где-то в другом месте, Geth не сможет его найти. Значит, в данном случае для передачи транзакции вы должны использовать метод `web3.eth.sendRawTransaction()`.

Метод `web3.eth.sendRawTransaction()` применяется для передачи «сырой», необработанной, транзакции, т. е. вам придется написать код для создания и подписания такой транзакции. А узел сети Ethereum напрямую передаст ее в сеть, не совершая над ней никаких иных действий. Однако написать код для передачи транзакции ме-

[1] Иными словами, мы говорим о «кошельке» в самом широком смысле этого слова — как о некоем способе хранения каких-либо ценностей, включающем бухгалтерские записи, долговые расписки и т. д. Такой подход обоснован, потому что децентрализованный реестр предназначен для надежного хранения записей о любых ценностях.

тодом `web3.eth.sendRawTransaction()` достаточно трудно, потому что надо сгенерировать блок данных, создать сырую транзакцию и подписать ее.

hooked-web3-provider — это библиотека, предоставляющая нам *прикладной провайдер*[2], который общается с Geth по протоколу HTTP. И уникальность данного провайдера состоит в том, что он позволяет нам подписывать вызовы `sendTransaction()` при помощи наших ключей. Следовательно, нам больше не надо создавать блок данных транзакции. Провайдер, по сути, переопределяет реализацию метода `web3.eth.sendTransaction()`. В общем, он позволяет нам подписывать как вызов `sendTransaction()`, так и вызов `web3.eth.sendTransaction()`.

Метод `sendTransaction()` экземпляра контракта осуществляет внутреннее формирование данных транзакции, а метод `web3.eth.sendTransaction()` вещает транзакцию в сеть.

ethereumjs — это коллекция библиотек JavaScript, относящихся к Ethereum. В свою очередь, ethereumjs-tx — это одна из библиотек коллекции. Она предоставляет API, относящиеся к транзакциям. Например, она помогает создать необработанную транзакцию, подписать эту транзакцию, проверить, правильным ли ключом подписана эта транзакция, и т. д.

Обе упомянутых библиотеки доступны и как компоненты Node.js, и как клиентская часть JavaScript. Скачайте hooked-web3-provider по адресу: **https://www.npmjs.com/package/hooked-web3-provider**, а ethereumjs-tx — по адресу: **https://www.npmjs.com/package/ethereumjs-tx**.

На момент подготовки книги была доступна версия hooked-web3-provider 1.0.0 и версия ethereumjs-tx 1.1.4[3].

Давайте на небольшом примере посмотрим, как использовать обе эти библиотеки вместе и отправлять транзакции без прямого участия Geth:

```
var provider = new HookedWeb3Provider({
host: "http://localhost:8545",
transaction_signer: {
        hasAddress: function(address, callback){
        callback(null, true);
        },
signTransaction: function(tx_params, callback){
var rawTx = {
        gasPrice: web3.toHex(tx_params.gasPrice),
        gasLimit: web3.toHex(tx_params.gas),
        value: web3.toHex(tx_params.value)
        from: tx_params.from,
```

[2] От англ. *custom provider* — интерфейс-посредник, средство доступа к функциям и методам другой программы или процесса.

[3] Во время работы над переводом книги версия Hooked-Web3-Provider оставалась прежней, а версия EthereumJS-TX обновилась до 1.3.3.

```
        to: tx_params.to,
        nonce: web3.toHex(tx_params.nonce)
        };

var privateKey =
        EthJS.Util.toBuffer('0x1a56e47492bf3df9c9563fa7f66e4e032c661de9d68c3f36f358
        e6bc9a9f69f2', 'hex');

var tx = new EthJS.Tx(rawTx);
        tx.sign(privateKey);

        callback(null, tx.serialize().toString('hex'));
        }
}
});

var web3 = new Web3(provider);

web3.eth.sendTransaction({
        from: "0xba6406ddf8817620393ab1310ab4d0c2deda714d",
        to: "0x2bdbec0ccd70307a00c66de02789e394c2c7d549",
        value: web3.toWei("0.1", "ether"),
        gasPrice: "20000000000",
        gas: "21000"
        }, function(error, result){
        console.log(error, result)
})
```

Этот код работает следующим образом:

1. Сначала мы создаем экземпляр объекта `HookedWeb3Provider` — он предоставлен библиотекой hooked-web3-provider. Конструктор создает объект с двумя свойствами:
   - `host` — это HTTP-адрес узла;
   - `transaction_signer` — ссылка на объект, с которым взаимодействует прикладной провайдер, чтобы подписать транзакцию.
2. В свою очередь, объект `transaction_signer` содержит два метода:
   - `hasAddress()` — служит для проверки, подписана ли транзакция (имеет ли подписант транзакции персональный закрытый ключ для аккаунта с адресом `from`). Этот метод принимает адрес и функцию обратного вызова. Обратный вызов должен сработать, если первый параметр ответа — это сообщение об ошибке, а второй параметр имеет значение `false` (закрытый ключ по указанному адресу не существует). Если ключ найден, первый параметр ответа имеет значение `null`, а второй — `true`;

- `signTransaction()` — этот метод вызывается для подписания транзакции, если найден закрытый ключ по указанному адресу. Метод принимает два аргумента: параметры транзакции и функцию обратного вызова. Внутри метода параметры нашей транзакции сначала преобразуются в параметры сырой транзакции, а затем значения этих параметров кодируются в шестнадцатеричные строки. Затем мы создаем буфер для хранения закрытого ключа. Буфер создается при помощи метода `EthJS.Util.toBuffer()`, который является частью библиотеки ethereumjs-util. Библиотека ethereumjs-util импортируется библиотекой ethereumjs-tx. Далее мы создаем и подписываем сырую транзакцию, сериализуем ее и конвертируем в шестнадцатеричную строку. Наконец, нам надо передать шестнадцатеричную строку подписанной сырой транзакции в прикладной провайдер при помощи функции обратного вызова. В случае, если внутри метода произошла ошибка, первым аргументом обратного вызова будет сообщение об ошибке.

3. Далее провайдер берет сырую транзакцию и транслирует ее в сеть при помощи метода `web3.eth.sendRawTransaction()`.
4. Наконец, мы вызываем функцию `web3.eth.sendTransaction` для отправки некоторой суммы эфира на другой счет. Здесь мы должны предоставить все параметры транзакции, кроме `nonce`, потому что провайдер может сам вычислить `nonce`. Раньше многие из параметров были необязательными, потому что мы доверяли их формирование узлу Ethereum. Но теперь, поскольку мы сами подписываем транзакцию, мы должны предоставить все параметры. Параметр `gas` всегда имеет значение `21000`, если транзакция не несет связанные с ней данные.

А где же открытый ключ?

В приведенном здесь коде нет упоминания об открытом ключе, который связан с адресом отправителя, подписавшего транзакцию. Вы должны были задуматься: как майнер проверит аутентичность транзакции без публичного ключа? Дело в том, что майнеры используют уникальное свойство ECDSA[4], которое позволяет им вычислить открытый ключ из сообщения и подписи. В транзакции сообщение указывает на назначение транзакции, а подпись используется для определения того, была ли транзакция подписана корректным закрытым ключом. Это является особенностью ECDSA. Библиотека ethereumjs-tx предоставляет API для проверки транзакций.

Что такое HD-кошелек?

HD-Wallet (Hierarchical Deterministic Wallet, иерархический детерминированный кошелек) — это система извлечения адресов и ключей из единственного источника, называемого *сидом* (от англ. *seed* — семя). *Детерминированность* означает, что из одинаковых сидов всегда будут сгенерированы одинаковые адреса и ключи. *Иерар-*

[4] Elliptic Curve Digital Signature Algorithm (ECDSA) — алгоритм построения цифровой подписи с использованием эллиптических кривых.

хия означает, что адреса и ключи будут генерироваться в одинаковом порядке. Такой подход упрощает резервное копирование и хранение набора счетов — ведь вам достаточно сохранить только сид, а не набор ключей и адресов.

Почему пользователи нуждаются в нескольких счетах?

Вам интересно, зачем пользователям несколько счетов? Обычно причина заключается в необходимости скрыть свое богатство. Баланс счета публично доступен через блокчейн. Таким образом, если пользователь A сообщает пользователю B адрес для получения некоторой суммы, то пользователь B может проверить, какая сумма эфира хранится по этому адресу. Поэтому пользователи обычно распределяют свое состояние по различным счетам.

Существуют разные типы HD-кошельков, которые различаются форматом сидов и алгоритмом генерации адресов и ключей, — например: BIP32, Armory, Coinkite, Coinb.in и др.

Что такое BIP32, BIP44 и BIP39?

BIP (Bitcoin Improvement Proposal, предложение по улучшению Bitcoin) — это проектный документ, предоставляющий информацию сообществу Bitcoin или описывающий новые функции Bitcoin, а также процессы и окружение. BIP должен излагать краткую техническую спецификацию функции и логическое обоснование этой функции. На момент подготовки этой книги имелось 152 предложения[5] по улучшению Bitcoin. BIP32 и BIP39 содержат, соответственно, информацию об алгоритме реализации HD-кошелька и спецификацию сида. Вы можете прочитать об этом больше по адресу: **https://github.com/bitcoin/bips**.

Введение в функции формирования ключа

Асимметричные криптографические алгоритмы определяют характер ключей и то, как они должны генерироваться, поскольку ключи должны быть связаны. Например, для генерации ключа RSA применяется детерминированный алгоритм.

Симметричные криптографические алгоритмы определяют только размер ключей, а генерировать ключи мы должны сами. Для генерации ключей существуют различные алгоритмы, одним из которых является KDF (Key Derivation Function, функция формирования ключа) — детерминированный алгоритм формирования симметричного ключа на основе секретного значения (мастер-ключа, пароля или парольной фразы). Существуют различные типы KDF, такие как bcrypt, crypt, PBKDF2, scrypt, HKDF и др. Вы можете прочитать больше про KDF по адресу: **https://en.wikipedia.org/wiki/Key_derivation_function**[6].

Основанная на пароле функция формирования ключа берет пароль и генерирует симметричный ключ. Примером парольной функции формирования ключа является

[5] На момент подготовки перевода было уже 174 предложения.

[6] На русском языке подобную информацию можно найти по адресу: **https://ru.wikipedia.org/wiki/Функция_формирования_ключа**.

тип PBKDF2. Как известно, пользователи обычно придумывают слабые пароли, поэтому парольные функции формирования ключей специально делают медленными и занимающими много памяти. Это осложняет атаку перебором паролей и другие типы атак. Основанные на пароле функции формирования ключа применяются так широко, потому что простой пароль запомнить намного проще, чем длинный криптоключ. Но если где-то хранить записанный криптоключ, то есть риск, что его похитят.

Мастер-ключ или парольная фраза труднее поддаются атаке перебором. Следовательно, для генерации симметричного ключа на основе мастер-ключа или парольной фразы вы можете использовать беспарольную функцию формирования ключа, например HKDF. Эта функция работает намного быстрее.

Почему вместо KDF не используют хеш-функцию?

Выходное значение хеш-функции можно использовать в качестве симметричного ключа. Поэтому вам должно быть интересно, зачем тогда существуют KDF? Если вы гарантированно используете парольную фразу или сложный пароль, вам достаточно применить к нему хеш-функцию. Например, HKDF просто использует хеш-функцию для генерации ключа. Но если вы не уверены, что пользователь придумает сложный пароль, то лучше задействовать KDF.

Знакомство с LightWallet

LightWallet — это HD-кошелек, в котором реализованы предложения BIP32, BIP39 и BIP44. LightWallet предоставляет API для создания и подписания транзакций или шифрования и расшифровки данных при помощи адресов и ключей на их основе.

API LightWallet подразделяется на четыре пространства имен[7]: `keystore`, `signing`, `encryption` и `txutils`.

Пространства имен `signing`, `encryption` и `txutils` предоставляют, соответственно, API для подписания, несимметричного шифрования и создания транзакций.

Пространство `keystore` используется при создании объекта `keystore`, генерации сидов и тому подобных операций. `keystore` — это объект, который содержит сид и ключи в зашифрованном виде. Пространство имен `keystore` реализует методы подписывания транзакций, которые требуют подписывать вызовы `we3.eth.sendTransaction()`, если мы используем библиотеку hooked-web3-provider. Поэтому пространство имен `keystore` способно автоматически создавать и подписывать транзакции для адресов, которые может в нем найти. В общем-то, LightWallet в первую очередь и создан в качестве подписывающего провайдера для hooked-web3-provider.

[7] Пространство имен (namespace) — это способ организации множества идентификаторов (имен), схожий с папками в файловой системе. Имя пространства формирует уникальное имя сущности. Например, `Signing.test` и `Encryption.test` — это разные классы `test`, потому что они относятся к разным пространствам имен.

Экземпляр `keystore` может быть настроен как для создания и подписания транзакций, так и для шифрования и дешифровки данных. Для подписания транзакций он использует параметр `secp256k`, а для шифрования/дешифровки — параметр `curve25519`.

Сидом для LightWallet служит мнемокод из 12 слов, который легко запомнить, но трудно взломать. Но это не могут быть любые 12 слов — сид должен быть сгенерирован самим LightWallet. Сгенерированный сид соответствует требованиям к подбору слов и другим параметрам.

Путь вывода HD-кошелька

Путь вывода HD-кошелька — это строка, которая упрощает обработку нескольких криптовалют (при допущении, что все они используют одинаковые алгоритмы подписи), множественных блокчейнов, множественных счетов и т. д.

Путь вывода может содержать столько параметров, сколько нам понадобится. Используя разные значения параметров, мы можем создать разные группы адресов и связанных с ними ключей.

По умолчанию LightWallet использует путь вывода: `m/0'/0'/0'`. Здесь `/n` является параметром, а `n` — значением этого параметра.

Каждый путь вывода содержит параметры `curve` и `purpose`. В свою очередь, `purpose` может принимать значения `sign` или `asymEncrypt`: `sign` показывает, что путь используется для подписания транзакций, а `asymEncrypt` — что путь используется для шифрования/дешифрования. `curve` обозначает параметр ECC (Elliptic Curve Cryptography). Для подписания транзакций это `secp256k`, а для асимметричного шифрования — `curve25519`, потому что LightWallet требует использовать именно эти параметры.

Разработка сервиса кошелька

Итак, мы в достаточной мере изучили основы LightWallet и настало время разработать сервис кошелька, используя LightWallet и hooked-web3-provider. Наш сервис предоставит пользователю возможности генерировать уникальный сид, отображать адреса и связанные с ними балансы, и, наконец, сервис позволит пользователю отправлять эфир на другие счета. Все операции выполняются на клиентской стороне, поэтому пользователь может доверять сервису как самому себе. Пользователь может запомнить сид или сохранить в надежном месте.

Предварительная подготовка

Прежде, чем приступить к разработке сервиса кошелька, убедитесь, что Geth запущен в режиме разработчика, майнинг включен, сервер HTTP-RPC запущен и при-

нимает клиентские запросы с любого домена, а аккаунт 0 разблокирован. Для этого вы можете воспользоваться следующей строкой запуска в терминале:

```
geth --dev --rpc --rpccorsdomain "*" --rpcaddr "0.0.0.0" --rpcport "8545"
--mine --unlock=0
```

Здесь опция **--rpccorsdomain** разрешает доступ с определенного домена. Мы должны предоставить ей список доменов, разделенных пробелом, — например: **"http://localhost:8080 https://mySite.com *"**. Можно использовать символ подстановки **"*"**, чтобы разрешить любые домены. Опция **--rpcaddr** показывает, по какому адресу доступен сервер Geth. По умолчанию это адрес `127.0.0.1`. Следовательно, если сервер расположен на стороннем хостинге, вы не сможете использовать публичный IP-адрес сервера. Поэтому мы указали адрес **0.0.0.0**, который означает, что сервер может быть расположен и доступен по любому адресу.

Структура проекта

В файлах папки упражнений, дополняющих эту главу (см. *приложение*), вы найдете два каталога: `Final` и `Initial`. Каталог `Final` содержит окончательный исходный код проекта, в то время как `Initial` содержит пустые файлы исходного кода и библиотеки, что позволяет быстро начать работу над проектом.

Для тестирования содержимого каталога `Final` следует выполнить команду терминала **npm install** внутри каталога. После этого запустите приложение командой терминала **node app.js** внутри каталога `Final`.

В каталоге `Initial` находится вложенный каталог `public` и два файла: `app.js` и `package.json`. Второй файл содержит зависимости для вашего приложения, а файл `app.js` — это место, в котором будет храниться серверная часть проекта.

Каталог `public` содержит файлы, относящиеся к пользовательскому интерфейсу (клиентская часть). Внутри `public/css` вы найдете библиотеку фреймворка Bootstrap `bootstrap.min.css`, а внутрь `public/html` вы поместите HTML-код вашего клиентского приложения. В каталоге `public/js` находятся JavaScript-файлы для hooked-web3-provider, web3js и LightWallet. Также внутри этого каталога вы найдете файл `main.js`, в который вы поместите код JavaScript для клиентской части проекта.

Разработка серверной части

Приступим к разработке серверной части приложения. Прежде всего выполните команду **npm install** внутри каталога `Initial`, чтобы установить зависимости для нашей серверной части.

Далее приведен простой и короткий код, который запускает службу `express` и готовит к выдаче файл `index.html` и статические файлы:

```
var express = require("express");
var app = express();
```

```
app.use(express.static("public"));

app.get("/", function(req, res){
res.sendFile(__dirname + "/public/html/index.html");
})
app.listen(8080);
```

Этот код не требует дополнительных пояснений.

Разработка клиентской части

Теперь создадим клиентскую часть приложения. Она будет поддерживать все основные функции: генерацию сида, отображение адресов сида и отправку эфира.

Вставьте HTML-код из листинга 5.1 в файл `index.html`.

Листинг 5.1. Содержимое файла `index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initialscale=1,
                                                                shrink-to-fit=no">
        <meta http-equiv="x-ua-compatible" content="ie=edge">
        <link rel="stylesheet" href="/css/bootstrap.min.css">
</head>

<body>
        <div class="container">
        <div class="row">
        <div class="col-md-6 offset-md-3">
        <br>
        <div class="alert alert-info" id="info" role="alert">
        Создание или использование кошелька
        </div>
        <form>
                <div class="form-group">
                <label for="seed">Введите сид (12 слов)</label>
                <input type="text" class="form-control"
                id="seed">
                </div>
                <button type="button" class="btn btn-primary"
onclick="generate_addresses()">Генерировать адреса</button>
                <button type="button" class="btn btn-primary"
                onclick="generate_seed()">Генерировать сид</button>
                </form>
```

```
<hr>
<h2 class="text-xs-center">Адреса, ключи и балансы сида</h2>
<ol id="list">
</ol>
<hr>
<h2 class="text-xs-center">Отправить эфир</h2>

<form>
        <div class="form-group">
        <label for="address1">Исходящий адрес</label>
        <input type="text" class="form-control" id="address1">
        </div>
        <div class="form-group">
        <label for="address2">Адрес получателя</label>
        <input type="text" class="form-control" id="address2">
        </div>
        <div class="form-group">
        <label for="ether">Эфир</label>
        <input type="text" class="form-control" id="ether">
        </div>
        <button type="button" class="btn btn-primary"
onclick="send_ether()">Отправить эфир</button>
</form>
</div>
</div>
</div>

<script src="/js/web3.min.js"></script>
<script src="/js/hooked-web3-provider.min.js"></script>
<script src="/js/lightwallet.min.js"></script>
<script src="/js/main.js"></script>

</body>
</html>
```

Код из листинга 5.1 работает следующим образом:

1. Сначала мы запрашиваем набор стилей Bootstrap 4.
2. Затем отображаем информационное поле, в котором будем показывать пользователю различные сообщения.
3. Далее, у нас отображается форма, которая содержит поле ввода и две кнопки. В поле ввода мы вводим имеющийся сид. Это же поле используется для отображения нового сида, если мы запросили генерацию сида.
4. Кнопка **Генерировать адреса** предназначена для отображения адресов, а кнопка **Генерировать сид** — для генерации нового уникального сида. Нажатие кнопки

Генерировать адреса вызывает метод `generate_addresses()`, а нажатие кнопки **Генерировать сид** — метод `generate_seed()`.

5. Далее у нас расположен неупорядоченный список. Пока он пустой. Здесь мы будем динамически отображать адреса, их балансы и связанные закрытые ключи сидов, когда пользователь нажмет кнопку **Генерировать адреса**.
6. Наконец, у нас есть еще одна форма, которая содержит адрес отправителя, адрес получателя и количество эфира для перевода. Адрес отправителя должен быть одним из адресов, которые отображаются в неупорядоченном списке.

Теперь напишем реализацию каждой функции, которую вызывает HTML-код. Сначала создадим код, который генерирует новый сид. Поместите этот короткий код в файл `main.js`:

```
function generate_seed()
{
        var new_seed = lightwallet.keystore.generateRandomSeed();
        document.getElementById("seed").value = new_seed;
        generate_addresses(new_seed);
}
```

Метод `generateRandomSeed()` из пространства имен `keystore` служит для генерации случайных сидов. Он может получать дополнительный строковый параметр, который содержит внешний источник энтропии (случайных данных).

Энтропия — это неупорядоченные данные[8], которые приложение собирает по определенному алгоритму из разных мест для последующего применения там, где требуются случайные данные. Обычно энтропию получают из аппаратных источников — например, из движений компьютерной мыши или специально сконструированных генераторов.

Для генерации уникального сида нам требуется по-настоящему высокая энтропия. LightWallet содержит встроенные методы генерации уникальных сидов, причем алгоритм LightWallet генерирует энтропию на основе окружающей среды. Но если вы хотите получить энтропию получше, то можете передать сгенерированную энтропию в метод `generateRandomSeed()`, и она будет сложена с энтропией, которая создана внутри метода `generateRandomSeed()`.

После генерации случайного сида мы вызываем метод `generate_addresses()`. Этот метод берет сид и показывает принадлежащие ему адреса. Прежде чем генерировать адреса, метод спрашивает у пользователя, сколько адресов ему нужно.

В листинге 5.2 содержится код реализации метода `generate_addresses()`. Добавьте этот код в файл `main.js`.

[8] Энтропией также называют меру неупорядоченности случайных данных. Чем выше энтропия, тем меньше упорядочены данные и тем надежнее будет ключ шифрования на их основе.

Листинг 5.2. Реализация метода `generate_addresses()`

```
var totalAddresses = 0;

function generate_addresses(seed)
     {
          if(seed == undefined)
          {
          seed = document.getElementById("seed").value;
          }
          if(!lightwallet.keystore.isSeedValid(seed))
          {
          document.getElementById("info").innerHTML = "Пожалуйста, введите правильный
                                                                                  сид";
          eturn;
     }

totalAddresses = prompt("Сколько адресов вы хотите сгенерировать?");

     if(!Number.isInteger(parseInt(totalAddresses)))
     {
          document.getElementById("info").innerHTML = "Пожалуйста, введите
                                                        допустимое число адресов";
          return;
     }

var password = Math.random().toString();

lightwallet.keystore.createVault({
password: password,
seedPhrase: seed
     }, function (err, ks) {
     ks.keyFromPassword(password, function (err, pwDerivedKey) {
     if(err)
     {
          document.getElementById("info").innerHTML = err;
     }
     else
     {
          ks.generateNewAddress(pwDerivedKey, totalAddresses);
          var addresses = ks.getAddresses();
          var web3 = new Web3(new
          Web3.providers.HttpProvider("http://localhost:8545"));
          var html = "";
          for(var count = 0; count < addresses.length; count++)
```

```
        {
            var address = addresses[count];
            var private_key = ks.exportPrivateKey(address, pwDerivedKey);
            var balance = web3.eth.getBalance("0x" + address);
            html = html + "<li>";
            html = html + "<p><b>Address: </b>0x" + address + "</p>";
            html = html + "<p><b>Private Key: </b>0x" + private_key + "</p>";
            html = html + "<p><b>Balance: </b>" + web3.fromWei(balance, "ether") +
            " ether</p>";
            html = html + "</li>";
        }
        document.getElementById("list").innerHTML = html;
        }
    });
    });
}
```

Код из листинга 5.2 работает следующим образом:

1. Сначала мы получаем значение переменной `totalAddresses` — она содержит количество адресов, которые мы хотим сгенерировать.
2. Затем мы проверяем, определен ли параметр `seed`. Если он определен, то считываем значение из поля ввода. Мы это делаем, чтобы иметь возможность использовать метод `generate_addressess()` для отображения информации о сиде при генерации нового сида, а также, если пользователь нажимает кнопку **Генерировать адреса**.
3. Далее мы проверяем правильность сида при помощи метода `isSeedValid()` из пространства имен `keystore`.
4. Спрашиваем у пользователя, сколько адресов надо создать, и проверяем введенное значение.
5. Закрытые ключи в пространстве имен `keystore` всегда хранятся в зашифрованном виде. При генерации ключей мы должны их зашифровать, а при подписании транзакций расшифровать обратно. Пароль для формирования симметричного ключа шифрования может быть придуман пользователем, а может быть сгенерирован в виде случайной строки. Для удобства пользователей мы генерируем случайную строку и используем ее в качестве пароля. Симметричный ключ не хранится в пространстве имен `keystore`. Поэтому мы должны генерировать ключ из пароля всякий раз, когда мы выполняем операции, связанные с закрытым ключом: генерацию ключей, получение доступа к ключам и т. д.
6. Затем мы используем метод `createVault()`, чтобы создать экземпляр `keystore`. Метод `createVault()` получает объект и функцию обратного вызова. Объект может иметь четыре свойства: `password`, `seedPhrase`, `salt` и `hdPathString`. Обязательным является только `password`, а остальные три свойства не обязательны. Если вы

не задали параметр `seedPhrase`, метод сгенерирует и будет использовать случайный сид. Параметр `salt` конкатенируется (сцепляется) с паролем для повышения безопасности симметричного ключа, поскольку атакующему придется найти не только пароль, но и значение `salt`. Если значение `salt` не задано, оно генерируется случайным образом. Пространство имен `keystore` хранит `salt` в открытом виде. Параметр `hdPathString` задает путь вывода для пространства `keystore`, т. е. при генерации адресов, подписании транзакций и т. д. Если вы не задали параметр `hdPathString`, то значением по умолчанию будет `m/0'/0'/0'`. Назначением по умолчанию для данного пути является `sign`. Вы можете создать новый путь вывода или изменить назначение текущего пути при помощи метода `addHdDerivationPath()` экземпляра `keystore`. Также вы можете сменить путь по умолчанию при помощи метода `setDefaultHdDerivationPath()` экземпляра `keystore`. Когда создано пространство имен `keystore`, экземпляр возвращается через обратный вызов. Таким образом, мы создали `keystore` лишь на основе пароля и сида.

7. Теперь мы должны сгенерировать для пользователя нужное количество адресов и связанных с ними ключей. Мы можем сгенерировать миллионы адресов на основе одного сида, поэтому `keystore` не генерирует никакие адреса до тех пор, пока пользователь не укажет количество. После создания экземпляра `keystore` мы генерируем симметричный ключ из пароля при помощи метода `keyFromPassword`. А затем вызываем метод `generateNewAddress()` для формирования адресов и связанных с ними ключей.

8. Метод `generateNewAddress()` получает три аргумента: ключ, сформированный из пароля, количество адресов для генерации и путь вывода. Так как мы не задали путь вывода, метод использует путь по умолчанию для хранилища ключей. Если вы вызываете метод `generateNewAddress()` несколько раз, то он возобновляет работу с адресов, созданных при последнем вызове. Например, если вы вызывали метод дважды и каждый раз создавали по два адреса, у вас получится четыре адреса.

9. Затем мы вызываем метод `getAddresses()`, чтобы получить все адреса, хранящиеся в `keystore`.

10. Мы расшифровываем и извлекаем закрытые ключи адресов при помощи метода `exportPrivateKey`.

11. При помощи метода `web3.eth.getBalance()` получаем балансы адресов.

12. И, наконец, отображаем всю информацию в виде списка.

Теперь мы знаем, как генерировать адреса и закрытые ключи из сида. Давайте напишем реализацию метода `send_ether()`, который будет отправлять эфир с одного из адресов, которые мы сгенерировали из сида.

Возьмите код из листинга 5.3 и добавьте его в файл `main.js`.

Листинг 5.3. Реализация метода `send_ether()`

```
function send_ether()
{
     var seed = document.getElementById("seed").value;
     if(!lightwallet.keystore.isSeedValid(seed))
     {
           document.getElementById("info").innerHTML = "Введите правильное значение";
           return;
     }

     var password = Math.random().toString();

     lightwallet.keystore.createVault({
     password: password,
     seedPhrase: seed
     }, function (err, ks) {
     ks.keyFromPassword(password, function (err, pwDerivedKey) {
     if(err)
     {
           document.getElementById("info").innerHTML = err;
     }
     else
     {
           ks.generateNewAddress(pwDerivedKey, totalAddresses);
           ks.passwordProvider = function (callback) {
           callback(null, password);
     };

     var provider = new HookedWeb3Provider({
     host: "http://localhost:8545",
     transaction_signer: ks
     });

     var web3 = new Web3(provider);

     var from = document.getElementById("address1").value;

     var to = document.getElementById("address2").value;

     var value = web3.toWei(document.getElementById("ether").value, "ether");
     web3.eth.sendTransaction({
     from: from,
     to: to,
     value: value,
```

```
    gas: 21000
    }, function(error, result){
    if(error)
    {
          document.getElementById("info").innerHTML = error;
    }
    else
    {
          document.getElementById("info").innerHTML = "Txn hash: " + result;
    }
    })
    }
    });
});
}
```

Фрагмент кода, который выполняет генерацию адресов из сида, не нуждается в пояснениях. Затем мы присоединяем обратный вызов к свойству `passwordProvider`. Этот вызов срабатывает с целью получения пароля и расшифровки закрытого ключа во время подписания транзакции. Если мы не задали этот пароль, LightWallet запросит пароль у пользователя. Далее мы создаем экземпляр `HookedWeb3Provider`, передавая экземпляр `keystore` в качестве заверителя транзакции. Теперь, если провайдеру надо подписать транзакцию, он вызывает методы `hasAddress` и `signTransactions`. Если адрес подписываемой транзакции не является тем адресом, который мы создавали, провайдеру будет возвращена ошибка. И, наконец, мы пересылаем какое-то количество эфира при помощи метода `web3.eth.sendTransaction`.

Тестирование

Мы закончили разработку нашего сервиса кошелька. Настало время проверить, работает ли он так, как задумано. Выполните команду терминала **`node app.js`** внутри начального каталога и введите `http://localhost:8080` в адресную строку вашего любимого браузера. Вы должны увидеть окно, аналогичное изображенному на рис. 5.1.

Теперь нажмите кнопку **Generate New Seed** (Генерировать сид), чтобы создать новый сид. Вам придется ввести количество адресов, которые нужно сгенерировать, — вы можете ввести любое число, но для нужд тестирования введите число больше единицы. После этого окно приложения должно приобрести вид, показанный на рис. 5.2.

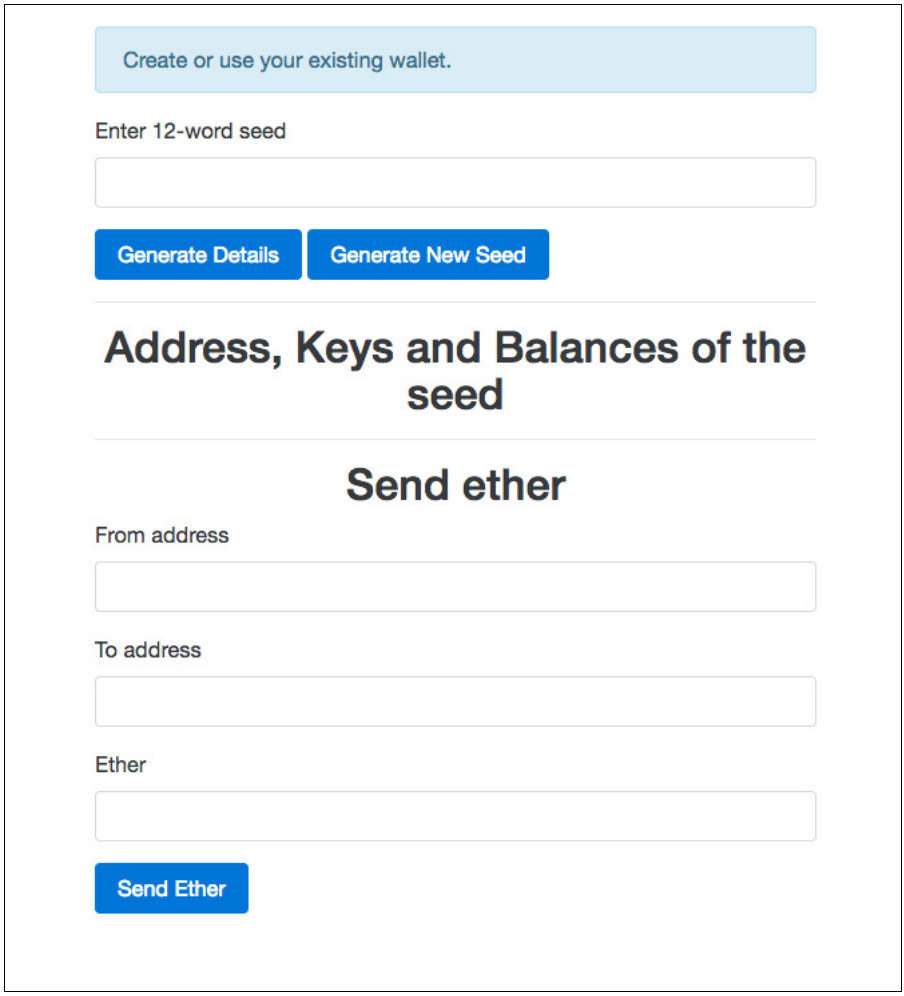

Рис. 5.1. Начальное окно клиентской части приложения

Теперь для тестирования перевода валюты вам надо перечислить какую-то сумму на один из вновь созданных адресов со своего основного счета. Когда вы это сделаете, нажмите кнопку **Generate Details** (Генерировать адреса), чтобы обновить информацию интерфейса, хотя это и не обязательно. Удостоверьтесь, что из сида развернуты (сгенерированы) те же самые адреса, что и на предыдущем шаге. Теперь окно приложения должно выглядеть приблизительно так, как показано на рис. 5.3.

В адресное поле **From address** (адрес отправителя) введите адрес, с которого хотите перевести средства. В поле **To address** (адрес получателя) введите адрес, на который должна быть зачислена валюта. Для тестирования мы укажем один из наших собственных адресов. Введите сумму перевода, которая меньше или равна балансу отправителя. Теперь окно приложения должно выглядеть, как показано на рис. 5.4.

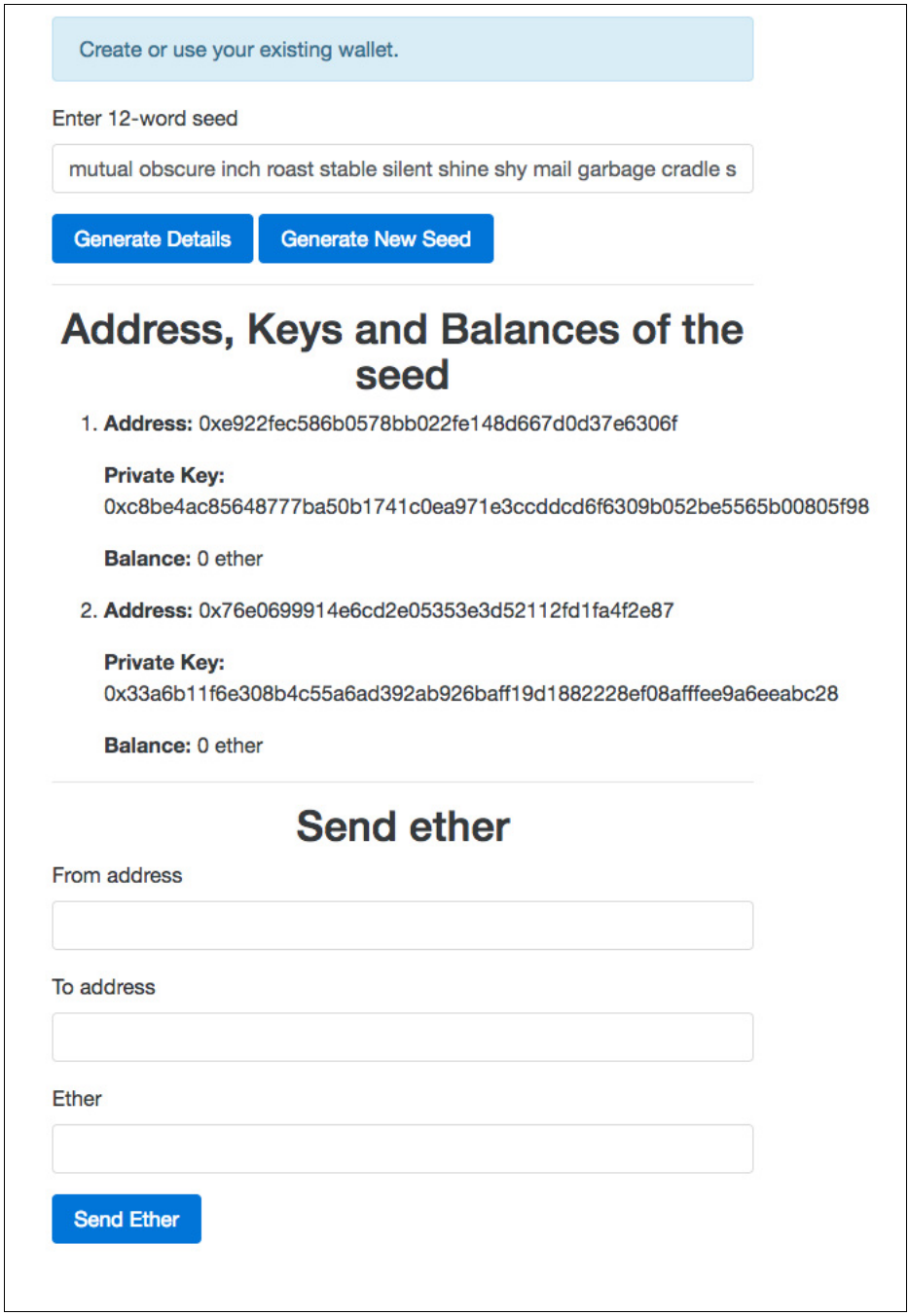

Рис. 5.2. Окно приложения после генерации адресов

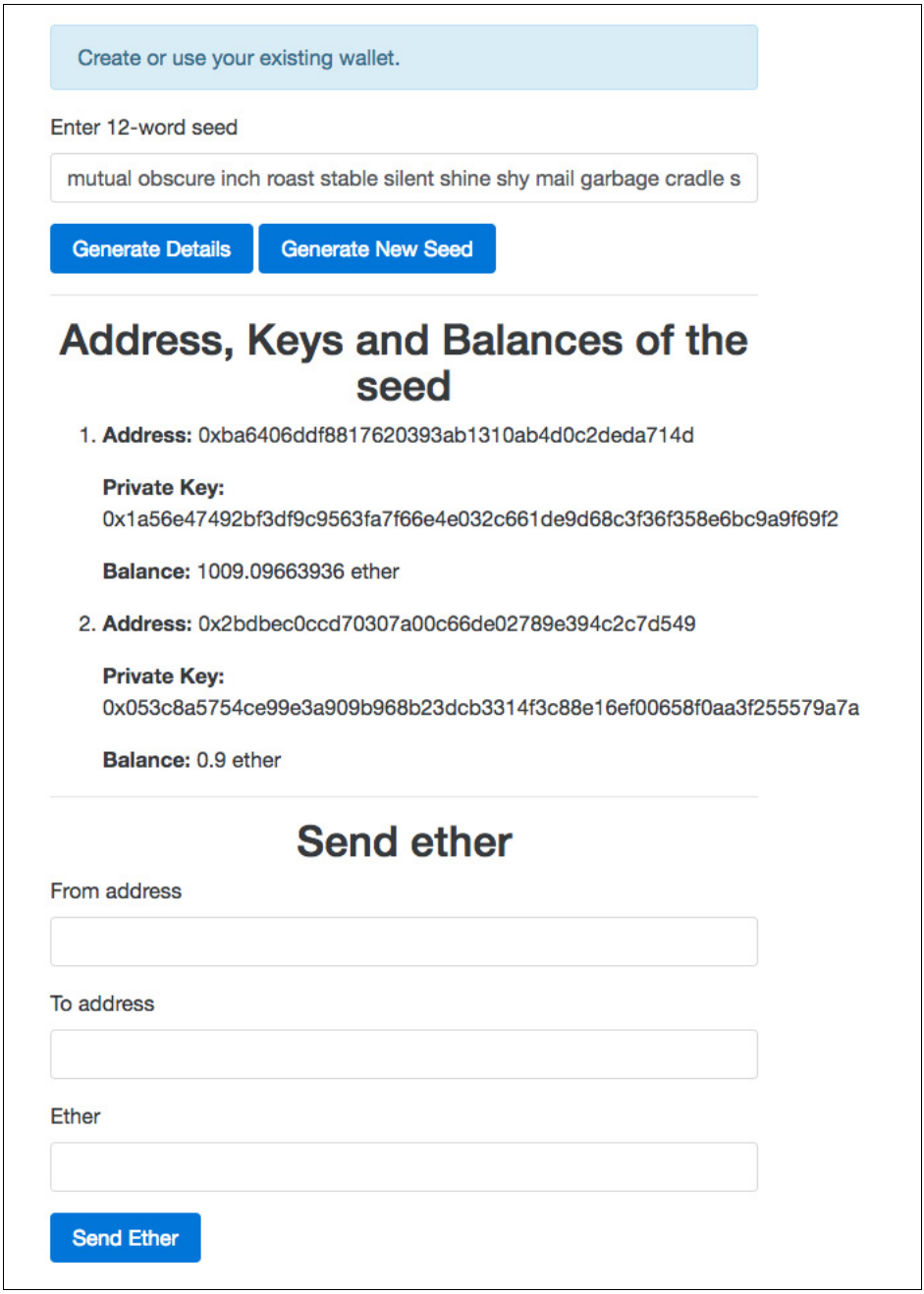

Рис. 5.3. Окно приложения после получения эфира на счет

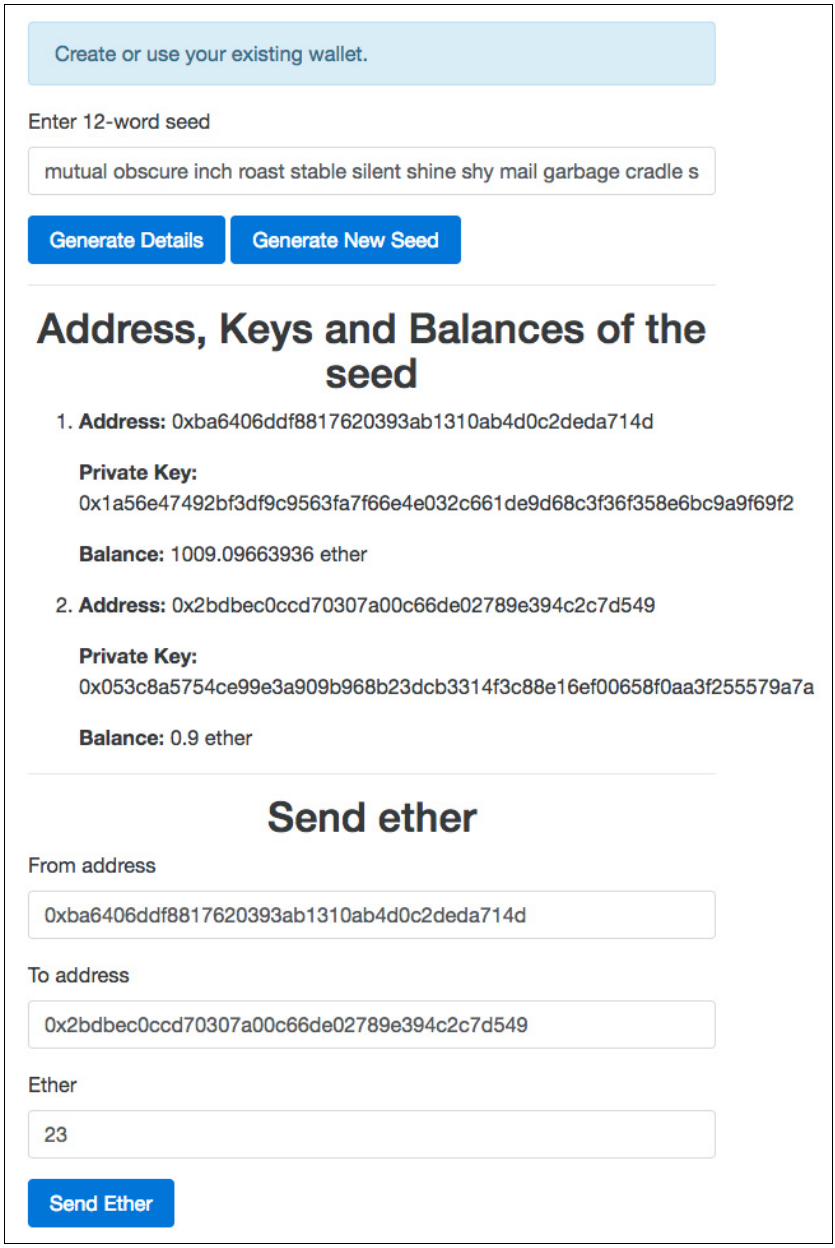

Рис. 5.4. Приложение готово к переводу валюты со счета на счет

Нажмите кнопку **Send Ether** (отправить эфир). В информационном блоке вы должны увидеть хеш транзакции. Подождите некоторое время, пока она обрабатывается. Вы можете проверить, завершен ли майнинг, нажимая кнопку **Generate Details** через короткие промежутки времени. Как только майнинг транзакции завершен, вы увидите окно приложения, аналогичное показанному на рис. 5.5.

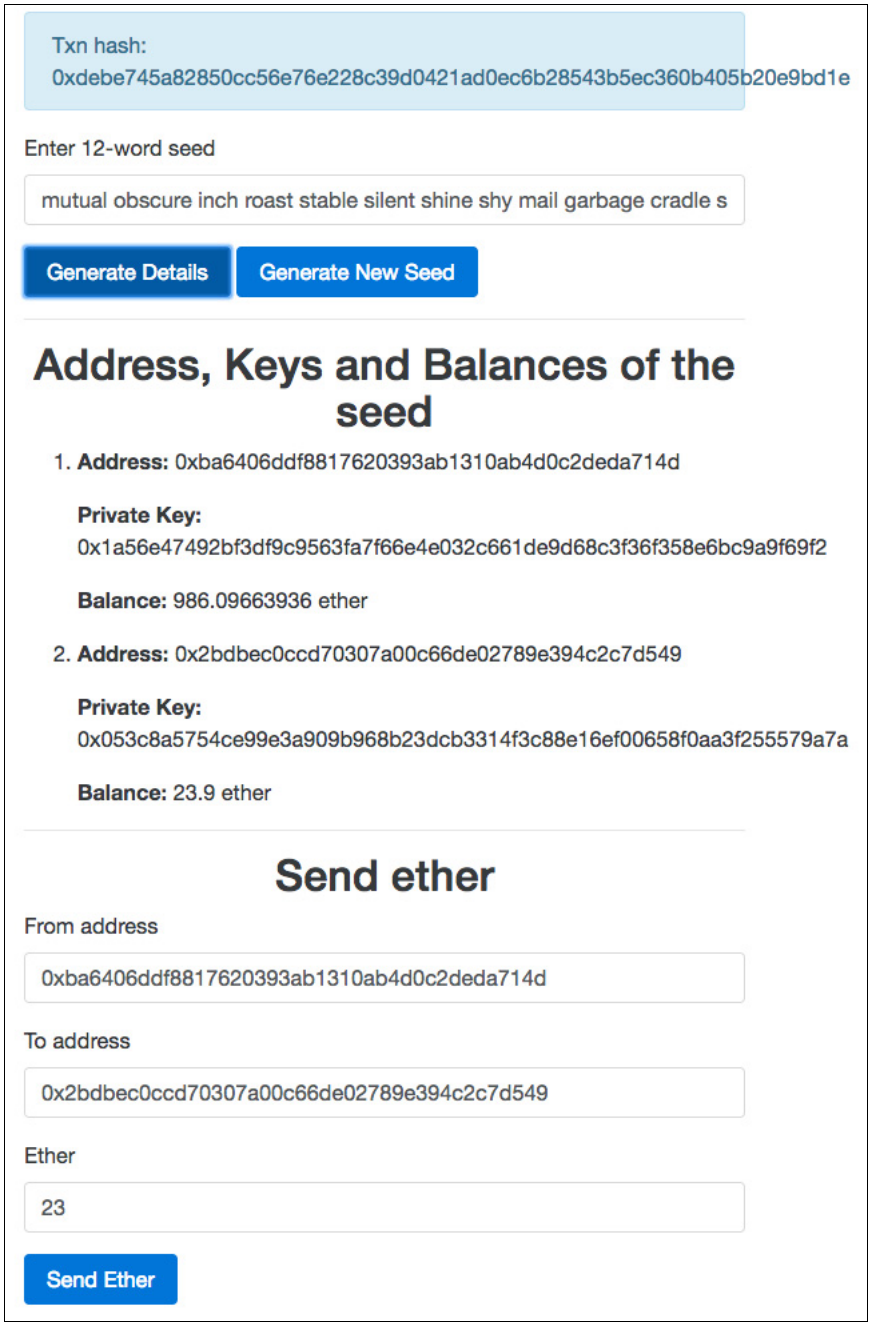

Рис. 5.5. Окно приложения после завершения майнинга транзакции

Если все сработало, как ожидалось, созданный вами сервис кошелька готов к работе. Вы можете поместить это приложение на своем домене и сделать доступным для общего пользования. Приложение полностью безопасно, и пользователи будут доверять ему.

Заключение

В этой главе вы узнали про три важных библиотеки Ethereum: hooked-web3-provider, ethereumjs-tx и LightWallet. Эти библиотеки используются для управления счетами и подписания транзакций вне узла Ethereum. Занимаясь разработкой клиентской части различных децентрализованных приложений, вы убедитесь в полезности этих библиотек.

В завершение главы мы создали сервис кошелька, который помогает пользователю управлять счетами и пересылать закрытые ключи и другую информацию кошелька в серверную часть сервиса.

В следующей главе мы построим платформу для разработки и развертывания смарт-контрактов.

6

Разработка платформы для смарт-контрактов

Клиентам некоторых приложений нужно компилировать и размещать контракты «на ходу», в режиме реального времени. В нашем приложении для подтверждения владения файлом (см. *главы 3* и *4*) мы размещали контракт вручную и жестко вписывали адрес контракта в код клиентской части. Но некоторым клиентам это не подходит. Например, если клиентская часть приложения позволяет отмечать посещаемость учащихся в блокчейне, то приложению придется развертывать смарт-контракт каждый раз, когда зарегистрирована новая школа, потому что каждая школа должна иметь полный доступ к смарт-контракту. Из материала этой главы мы узнаем, как скомпилировать смарт-контракты с помощью web3.js и развернуть их с помощью web3.js и ethereumjs.

В этой главе будут рассмотрены следующие темы:

- вычисление `nonce`[1] для транзакции;
- использование пула транзакций API JSON-RPC;
- формирование данных транзакции для создания контракта;
- оценка количества газа, необходимого для транзакции;
- нахождение текущего доступного баланса счета;
- компиляция смарт-контракта при помощи solcjs;
- разработка платформы для написания, компиляции и развертывания смарт-контрактов.

---

[1] Напомним, что число `nonce` для транзакции — это не то число nonce, которое определяется в процессе майнинга (см. *главу 2*).

Вычисление *nonce* для транзакции

Если счета размещены на узле под управлением Geth, мы не заботимся о вычислении `nonce`, потому что Geth сам добавляет корректное значение `nonce` к транзакции и подписывает ее. Если же мы имеем дело со счетами, работающими независимо от Geth, то должны вычислять `nonce` самостоятельно.

Для самостоятельного вычисления `nonce` мы можем использовать принадлежащий Geth метод `getTransactionCount()`. Первый аргумент метода представляет собой адрес транзакции, для которой производится вычисление `nonce`. Второй аргумент — это блок, в который помещена транзакция. Мы можем применить строку `"pending"` для обозначения блока, который пока находится в процессе майнинга. Как мы уже говорили в предыдущих главах, Geth формирует накопитель транзакций (transaction pool), в котором он хранит ожидающие и поставленные в очередь транзакции. Для майнинга блока Geth берет из накопителя ожидающие транзакции и запускает выработку нового блока. До тех пор, пока майнинг блока не завершен, транзакции остаются в накопителе. По завершении майнинга транзакции из накопителя удаляются. Новые транзакции, поступившие во время майнинга блока, помещаются в накопитель и обрабатываются в следующем блоке. Следовательно, если мы указываем строку `"pending"` в качестве второго аргумента вызова `getTransactionCount()`, метод не заглядывает внутрь накопителя, а рассматривает транзакции в обрабатываемом блоке.

Итак, если вы отправляете транзакции со счета, который не обслуживается на узле Geth, то вычисляете общее количество транзакций данного счета в блокчейне и складываете его с количеством транзакций, ждущих в накопителе. Если вы попытаетесь учесть ожидающие транзакции из обрабатываемого блока, то не получите корректное значение `nonce` в случае, если транзакции отправлены в Geth с интервалом в несколько секунд, потому что для включения транзакции в блокчейн требуется в среднем 12 секунд.

В предыдущей главе для добавления `nonce` к транзакции мы обращались к провайдеру hooked-web3-provider. К сожалению, этот провайдер не умеет получать значение `nonce` правильным способом: он создает счетчик для каждого счета и инкрементирует его каждый раз, когда вы отправляете транзакцию со счета. Но если транзакция оказалась ошибочной (например, если транзакция пытается перевести денег больше, чем есть на счете), то обратное уменьшение счетчика не происходит. Следовательно, остальные транзакции счета будут поставлены в очередь, но не будут обработаны майнером до тех пор, пока не сброшен hooked-web3-provider, т. е. пока не перезапущена клиентская часть приложения. Если вы создали несколько экземпляров hooked-web3-provider, они не способны синхронизировать текущее значение `nonce` между собой, поэтому вы легко можете получить некорректное значение. Но перед тем как включить `nonce` в транзакцию, hooked-web3-provider всегда получает количество транзакций, которые еще не попали в ожидающий майнинга блок, сравнивает его со своим счетчиком и использует большее значение. Следовательно, если транзакция со счета, управляемого hooked-web3-provider, отправлена

с другого узла сети и уже включена в ожидающий блок, провайдер увидит это. Но мы не можем целиком полагаться на hooked-web3-provider в задаче вычисления `nonce`. Он прекрасно подходит лишь для быстрого прототипирования клиентской части приложений или для тех приложений, в которых пользователь может видеть и повторно отправлять транзакции, если они не отправлены в сеть, а провайдер часто перезапускается. Например, в нашем сервисе кошелька (см. *главу 5*) пользователь часто перезагружает страницу, соответственно часто создается новый объект hooked-web3-provider. Если транзакция не передана, ошибочна или не обработана, пользователь может обновить страницу и отправить транзакцию заново.

Знакомство с solcjs

solcjs — это библиотека Node.js и инструмент командной строки, который предназначен для компиляции программ на языке Solidity. solcjs не использует компилятор Solc. Напротив, он выполняет компиляцию исключительно при помощи JavaScript, поэтому заметно проще в установке, чем Solc.

Solc — это компилятор для языка Solidity, написанный на C++. Затем код C++ скомпилирован в JavaScript при помощи межплатформенного компилятора Emscripten[2]. Каждая версия solc компилируется в JavaScript. По адресу: **https://github.com/ethereum/solc-bin/tree/gh-pages/bin** вы можете найти компиляторы на базе JavaScript для любой версии Solidity. Инструмент solcjs просто использует один из этих компиляторов для компиляции исходного кода Solidity. JavaScript-компиляторы могут быть запущены как в браузере Solidity, так и в среде Node.js.

Браузер Solidity использует упомянутые компиляторы на основе JavaScript для компиляции исходного кода Solidity.

Установка solcjs

Компилятор solcjs доступен в виде NPM-пакета под именем solc. Вы можете выполнить установку пакета локально или глобально, аналогично другим пакетам NPM. Если пакет установлен глобально, то будет доступен инструмент командной строки solcjs. Поэтому для установки инструмента командной строки выполните следующую команду терминала:

```
npm install -g solc
```

Теперь выполните следующую команду, чтобы узнать, как компилировать файлы Solidity при помощи командной строки:

```
solcjs -help
```

[2] Об Emscripten — компиляторе из LLVM-байт-кода в JavaScript — можно прочитать здесь: **https://en.wikipedia.org/wiki/Emscripten**.

В этой книге мы не будем использовать инструмент командной строки, а воспользуемся API solcjs для компиляции файлов Solidity.

По умолчанию solcjs использует компилятор, версия которого совпадает с его собственной версией. Например, если вы используете solcjs версии 0.4.8, то по умолчанию он применит компилятор версии 0.4.8. Но solcjs можно настроить для использования другой версии компилятора. На момент работы над книгой последней была версия solcjs 0.4.8.

API solcjs

solcjs предоставляет метод `compile()`, который применяется для компиляции кода Solidity. Этот метод может быть использован двумя разными способами в зависимости от того, содержит ли исходный код импорты. Если исходный код не содержит ни одного импорта, то метод получает два аргумента: первый аргумент — это собственно исходный код Solidity в виде строки, а второй — булева переменная, которая указывает, оптимизировать байт-код или нет. Если строка исходного кода содержит несколько контрактов, то все они будут скомпилированы.

Вот простой пример, который это демонстрирует:

```
var solc = require("solc");
var input = "contract x { function g() {} }";
var output = solc.compile(input, 1); // 1 activates the optimiser
for (var contractName in output.contracts) {
        //выводим в консоль код и ABI
        console.log(contractName + ": " +
output.contracts[contractName].bytecode);
        console.log(contractName + "; " +
JSON.parse(output.contracts[contractName].interface));
}
```

Если исходный код содержит импорты, то первым аргументом будет объект, чьи ключи являются именами файлов, а значения — содержимым файлов. Поэтому всякий раз, когда компилятор видит оператор `import`, он не ищет файл в файловой системе. Вместо этого он ищет содержимое файла в объекте, сопоставляя имя файла с ключами. Вот пример, который это демонстрирует:

```
var solc = require("solc");
var input = {
"lib.sol": "library L { function f() returns (uint) { return 7; } }",
"cont.sol": "import 'lib.sol'; contract x { function g() { L.f(); } }"
};
var output = solc.compile({sources: input}, 1);
for (var contractName in output.contracts)
console.log(contractName + ": " +
output.contracts[contractName].bytecode);
```

Если вы хотите прочитать содержимое импортированного файла из файловой системы во время компиляции или распознать содержимое файла при компиляции, то для таких случаев метод `compile()` поддерживает третий аргумент, который сам является ссылкой на метод, получающий имя файла и возвращающий содержимое этого файла.

Вот пример, который демонстрирует такой вызов метода:

```
var solc = require("solc");
var input = {
       "cont.sol": "import 'lib.sol'; contract x { function g() { L.f(); } }"
};
function findImports(path) {
       if (path === "lib.sol")
       return { contents: "library L { function f() returns (uint) {
return 7; } }" }
       else
       return { error: "Файл не найден" }
}
var output = solc.compile({sources: input}, 1, findImports);
for (var contractName in output.contracts)
       console.log(contractName + ": " +
output.contracts[contractName].bytecode);
```

Использование различных версий компилятора

Для компиляции контракта с использованием различных версий Solidity вам следует применять метод `useVersion()`, который возвращает ссылку на другой компилятор. Метод `useVersion()` получает строку с именем файла JavaScript, содержащего компилятор, и ищет файл в каталоге `/node_modules/solc/bin`.

solcjs также содержит метод с именем `loadRemoteVersion()`, который получает имя компилятора, ищет файл с таким именем в репозитории: **https://github.com/ethereum/solc-bin/tree/gh-pages/bin**, скачивает этот файл и использует его.

Наконец, solcjs предоставляет метод `setupMethods()`, который аналогичен методу `useVersion()`, но может загружать компилятор из любого каталога.

Рассмотрим пример, который демонстрирует все три метода (листинг 6.1).

Листинг 6.1. Демонстрация методов для работы с версиями компилятора

```
var solc = require("solc");

var solcV047 = solc.useVersion("v0.4.7.commit.822622cf");
var output = solcV011.compile("contract t { function g() {} }", 1);
```

```
solc.loadRemoteVersion('soljson-v0.4.5.commit.b318366e', function(err,
solcV045) {
        if (err) {
        // Сообщаем об ошибке и завершаем работу
        }

        var output = solcV045.compile("contract t { function g() {} }", 1);
});

var solcV048 = solc.setupMethods(require("/my/local/0.4.8.js"));
var output = solcV048.compile("contract t { function g() {} }", 1);

solc.loadRemoteVersion('latest', function(err, latestVersion) {
        if (err) {
        // Сообщаем об ошибке и завершаем работу
        }
var output = latestVersion.compile("contract t { function g() {} }",
1);
});
```

Чтобы выполнить код из листинга 6.1, вы сначала должны скачать файл `v0.4.7.commit.822622cf.js` из репозитория и поместить его в каталог `node_modules/solc/bin`. Затем скачать файл компилятора Solidity версии 0.4.8, поместить его где угодно в файловой системе и указать путь к нему в аргументе метода `setupMethods()`.

Связывание библиотек

Если исходный код Solidity ссылается на библиотеки, то сгенерированный байт-код будет содержать поля-заполнители (placeholders) для реальных адресов библиотек, на которые имеются ссылки. Перед развертыванием контракта заполнители должны быть обновлены при помощи процесса, который называется *связыванием*.

Для связывания библиотек solcjs предоставляет метод `linkByteCode()`, выполняющий привязку адресов библиотек к байт-коду.

Вот пример, который демонстрирует применение метода `linkByteCode()`:

```
var solc = require("solc");
var input = {
"lib.sol": "library L { function f() returns (uint) { return 7; } }",
"cont.sol": "import 'lib.sol'; contract x { function g() { L.f(); } }"
};
var output = solc.compile({sources: input}, 1);
var finalByteCode = solc.linkBytecode(output.contracts["x"].bytecode, {
'L': '0x123456...' });
```

Обновление ABI

ABI (Application Binary Interface, двоичный интерфейс приложения) контракта кроме реализации методов предоставляет и другие виды информации. ABI, созданные двумя разными версиями компилятора, могут различаться. Новая версия компилятора поддерживает больше возможностей Solidity, чем старая, и расширяет набор опций ABI. Например, резервная функция (см. *главу 3*) была заявлена в версии 0.4.0 языка Solidity. Если использовать компилятор, версия которого меньше, чем 0.4.0, то в ABI не будет включена информация о резервной функции, и контракт будет выглядеть так, словно резервная функция имеет пустое тело и модификатор `payable`. Следовательно, ABI нашего контракта должен быть обновлен, чтобы приложения, которые созданы для новой версии Solidity, могли получить более полную информацию о контракте.

solcjs располагает средствами для обновления ABI. Ознакомьтесь с примером, который демонстрирует эту возможность:

```
var abi = require("solc/abi");
var inputABI =
[{"constant":false,"inputs":[],"name":"hello","outputs":[{"name":"","type":
"string"}],"payable":false,"type":"function"}];
var outputABI = abi.update("0.3.6", inputABI)
```

Здесь строка `"0.3.6"` показывает, что ABI был сгенерирован компилятором версии 0.3.6. Поскольку мы используем solcjs версии 0.4.8, то ABI будет обновлен, чтобы соответствовать версии 0.4.8 (но не выше).

Результат компиляции приведенного здесь кода будет следующим:

```
[{"constant":false,"inputs":[],"name":"hello","outputs":[{"name":"","type":"string"
}],"payable":true,"type":"function"},{"type":"fallback","payable":true}]
```

Разработка платформы для развертывания контрактов

Теперь, когда мы знаем, как применять solcjs для компиляции исходного кода Solidity, пришло время создать платформу для написания, компиляции и развертывания контрактов. Наша платформа позволит пользователям вводить свой адрес и закрытый ключ, с помощью которых платформа будет размещать контракты в сети.

Прежде чем приступить к разработке сервиса кошелька, убедитесь, что Geth запущен в режиме разработчика, майнинг включен, RPC работает и предоставляет API `eth`, `web3` и `txpool` через сервер HTTP-RPC. Для этого вы можете воспользоваться следующей строкой запуска в терминале:

```
geth --dev --rpc --rpccorsdomain "*" --rpcaddr "0.0.0.0" --rpcport "8545" --mine
--rpcapi "eth,txpool,web3"
```

Структура проекта

В файлах папки упражнений, дополняющих эту главу (см. *приложение*), вы найдете два каталога: `Final` и `Initial`. Каталог `Final` содержит окончательный исходный код проекта, в то время как `Initial` содержит пустые файлы исходного кода и библиотеки, что позволяет быстро начать самостоятельную работу над проектом.

Для тестирования содержимого каталога `Final` следует выполнить команду терминала **`npm install`** внутри каталога. После этого запустите приложение командой терминала **`node app.js`** внутри каталога `Final`.

В каталоге `Initial` находится вложенный каталог `public` и два файла: `app.js` и `package.json`. Второй файл содержит зависимости для вашего приложения, а файл `app.js` — это место, в котором будет храниться серверная часть проекта.

Каталог `public` содержит файлы, относящиеся к пользовательскому интерфейсу (клиентская часть). Внутри `public/css` вы найдете библиотеку фреймворка Bootstrap `bootstrap.min.css`, а также файл `codemirror.css`, который содержит таблицу стилей библиотеки Codemirror. Внутри `public/html` находится файл `index.html`, в который вы поместите HTML-код вашего клиентского приложения. В каталоге `public/js` находятся JavaScript-файлы для web3js и Codemirror. Также внутри этого каталога вы найдете файл `main.js`, в который вы поместите код JavaScript для клиентской части проекта.

Разработка серверной части

Приступим к разработке серверной части приложения. Прежде всего выполните команду **`npm install`** внутри каталога `Initial`, чтобы установить зависимости для нашей серверной части.

Далее приведен простой и короткий код, который запускает службу `express` и готовит к выдаче файл `index.html` и статические файлы:

```
var express = require("express");
var app = express();

app.use(express.static("public"));

app.get("/", function(req, res){
res.sendFile(__dirname + "/public/html/index.html");
})
app.listen(8080);
```

Этот код не требует дополнительных пояснений. Теперь пойдем дальше. У нашего приложения будет две кнопки: **Compile** (Компилировать) и **Deploy** (Развернуть). Соответственно, когда пользователь нажимает кнопку **Compile**, контракт компилируется, а когда нажимает кнопку **Deploy** — контракт размещается в сети.

Мы будем выполнять компиляцию и развертывание контрактов на серверной стороне приложения. Несмотря на то, что эти операции можно выполнить и на клиентской стороне, мы воспользуемся именно серверной частью, потому что solcjs доступен только для Node.js (в то время как JavaScript-компиляторы, которые он использует, работают на клиентской стороне).

Чтобы узнать, как выполнять компиляцию на стороне клиента, вам следует разобраться с исходным кодом solcjs, в котором вы найдете упоминания об API компиляторов, основанных на JavaScript.

Когда пользователь нажимает кнопку **Compile**, клиентская часть формирует GET-запрос по пути `/compile` и передает исходный код контракта:

```
var solc = require("solc");
app.get("/compile", function(req, res){
var output = solc.compile(req.query.code, 1);
res.send(output);
})
```

Сначала мы импортируем библиотеку solcjs. Затем задаем путь `/compile` и внутри функции обратного вызова просто компилируем с включенным оптимизатором отправленный клиентом код. Далее получаем результат метода `solc.compile()`, отправляем его в клиентскую часть и даем пользователю возможность проверить, была ли компиляция успешной.

Когда пользователь нажимает на кнопку **Deploy**, клиентская часть формирует GET-запрос по пути `/deploy` и передает исходный код контракта и аргументы конструктора: адрес и закрытый ключ. После нажатия этой кнопки происходит развертывание контракта, а пользователю возвращается хеш транзакции.

В листинге 6.2 приведен код, который выполняет эту работу.

Листинг 6.2. Код обработки нажатия кнопки Deploy и развертывания контракта

```
var Web3 = require("web3");
var BigNumber = require("bignumber.js");
var ethereumjsUtil = require("ethereumjs-util");
var ethereumjsTx = require("ethereumjs-tx");

var web3 = new Web3(new
Web3.providers.HttpProvider("http://localhost:8545"));

function etherSpentInPendingTransactions(address, callback)
{
        web3.currentProvider.sendAsync({
        method: "txpool_content",
        params: [],
        jsonrpc: "2.0",
```

```
id: new Date().getTime()
        }, function (error, result) {
        if(result.result.pending)
        {
        if(result.result.pending[address])
        {
                var txns = result.result.pending[address];
                var cost = new BigNumber(0);
                for(var txn in txns)
                {
                cost = cost.add((new BigNumber(parseInt(txns[txn].value))).add((new
                BigNumber(parseInt(txns[txn].gas))).mul(new
                BigNumber(parseInt(txns[txn].gasPrice)))));
        }
        callback(null, web3.fromWei(cost, "ether"));
        }
        else
        {
                callback(null, "0");
        }
        }
        else
        {
                callback(null, "0");
        }
        })
}

function getNonce(address, callback)
{
        web3.eth.getTransactionCount(address, function(error, result){
        var txnsCount = result;
        web3.currentProvider.sendAsync({
        method: "txpool_content",
        params: [],
        jsonrpc: "2.0",
        id: new Date().getTime()
        }, function (error, result) {
        if(result.result.pending)
        {
                if(result.result.pending[address])
                {
                        txnsCount = txnsCount +
                        Object.keys(result.result.pending[address]).length;
                        callback(null, txnsCount);
                }
```

```
                else
                {
                        callback(null, txnsCount);
                }
        }
        else
        {
                callback(null, txnsCount);
        }
        })
        })
}

app.get("/deploy", function(req, res){
var code = req.query.code;
var arguments = JSON.parse(req.query.arguments);
var address = req.query.address;

var output = solc.compile(code, 1);

var contracts = output.contracts;

for(var contractName in contracts)
{
        var abi = JSON.parse(contracts[contractName].interface);
        var byteCode = contracts[contractName].bytecode;
        var contract = web3.eth.contract(abi);

        var data = contract.new.getData.call(null, ...arguments, {
        data: byteCode
        });

        var gasRequired = web3.eth.estimateGas({
        data: "0x" + data
        });

        web3.eth.getBalance(address, function(error, balance){
        var etherAvailable = web3.fromWei(balance, "ether");
        etherSpentInPendingTransactions(address, function(error, balance){
        etherAvailable = etherAvailable.sub(balance)

        if(etherAvailable.gte(web3.fromWei(new
        BigNumber(web3.eth.gasPrice).mul(gasRequired), "ether")))
        {
                getNonce(address, function(error, nonce){
                var rawTx = {
                gasPrice: web3.toHex(web3.eth.gasPrice),
```

```
                gasLimit: web3.toHex(gasRequired),
                from: address,
                nonce: web3.toHex(nonce),
                data: "0x" + data
        };

        var privateKey = ethereumjsUtil.toBuffer(req.query.key, 'hex');

        var tx = new ethereumjsTx(rawTx);

        tx.sign(privateKey);

        web3.eth.sendRawTransaction("0x" + tx.serialize().toString('hex'),

        function(err, hash) {
                res.send({result: {
                hash: hash,
        }});
        });
        })
        }
        else
        {
                res.send({error: "Недостаточно средств на балансе"});
        }
        })
        })
        break;
        }
})
```

Давайте пошагово рассмотрим, как работает код из листинга 6.2:

1. Сначала мы импортируем библиотеки `web3.js`, `BigNumber.js`, `ethereumjs-util` и `ethereumjs-tx`. Затем создаем экземпляр `web3`.
2. Определяем функцию с именем `etherInSpentPendingTransactions()`. Эта функция вычисляет общую сумму эфира, которая будет израсходована ожидающими транзакциями данного адреса. Библиотека `web3.js` не предоставляет JavaScript API для пула транзакций, поэтому мы выполняем непосредственный вызов JSON-RPC посредством `web3.currentProvider.sendAsync()`. Использованный здесь метод `sendAsync()` применяется для асинхронных вызовов JSON-RPC. Если вы хотите выполнить синхронный вызов, то используйте метод `send()`. Во время вычисления общей суммы эфира всех ждущих транзакций данного адреса мы ищем ожидающие транзакции в пуле транзакций, но не в ожидающем блоке, по

причине проблемы, которую уже обсудили в этой главе ранее[3]. При вычислении общей суммы мы прибавляем к ней газ каждой транзакции, потому что газ тоже вычитается из баланса счета.

3. Далее мы определяем функцию `getNonce()`. Она получает значение `nonce` по методике, которую мы обсудили ранее. Эта функция просто складывает общее количество обработанных транзакций с общим количеством ожидающих транзакций.

4. Наконец, мы объявляем конечную точку `/deploy`. Сначала мы компилируем контракт. Затем размещаем только первый контракт. Наша платформа разработана для размещения первого контракта, если в исходном коде обнаружено несколько контрактов. Потом вы можете доработать приложение, чтобы развернуть все скомпилированные контракты, а не только первый. Далее мы создаем объект контракта при помощи `web3.eth.contract`.

5. Мы не используем hooked-web3-provider или другие средства для перехвата вызова `sendTransactions` и преобразования его в вызов `sendRawTransaction`, поэтому для развертывания контракта мы теперь должны сгенерировать блок данных транзакции, который будет содержать байт-код контракта и аргументы конструктора, представленные в виде шестнадцатеричной строки. Объект контракта позволяет нам получить блок данных транзакции. Мы достигаем этого вызовом метода `getData()`. Если вы хотите развернуть контракт, то выполняете вызов `contract.new.getData()`, а если хотите вызвать функцию контракта, то вызываете `contract.functionName.getData()`. В обоих случаях не забудьте указать аргументы метода `getData()`. Итак, для формирования блока данных транзакции вам нужен только ABI контракта. Для более глубокого изучения вопроса, как имя и аргументы функции упаковываются в блок данных, посетите страницу по адресу: **https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI#examples**, но в этом нет необходимости, если вам доступен ABI контракта или вы знаете, как создать его собственноручно.

6. Затем мы применяем `web3.eth.estimateGas()`, чтобы вычислить количество газа, которое необходимо для развертывания контракта.

7. Далее мы проверяем, хватает ли на счете средств для оплаты газа, который будет затрачен на развертывание контракта. Для этого мы запрашиваем наличие средств на балансе адреса, вычитаем из него средства, потраченные ожидающими транзакциями, и убеждаемся, что остаток на балансе больше или равен стоимости газа.

8. В завершение процесса мы получаем `nonce`, подписываем и отправляем транзакции, а затем просто возвращаем хеш транзакции в клиентскую часть приложения.

---

[3] См. ранее *разд. «Вычисление `nonce` для транзакции»*.

Разработка клиентской части

Теперь приступим к разработке клиентской части приложения. Она будет содержать редактор, и пользователи смогут писать исходный код контракта. А когда пользователь нажмет кнопку **Compile**, мы будем динамически отображать ему поля ввода, которые соответствуют аргументам конструктора. По нажатию кнопки **Deploy** значения аргументов конструктора считываются из этих полей. Пользователь должен вводить в эти поля строку в формате JSON.

Для добавления редактора в клиентскую часть мы используем библиотеку Codemirror. Вы можете узнать о ней больше по адресу: **http://codemirror.net/**.

В листинге 6.3 приведен HTML-код клиентской части приложения. Поместите его в файл `index.html`.

Листинг 6.3. HTML-код клиентской части приложения для файла `index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1,
                                                                shrink-to-fit=no">
        <meta http-equiv="x-ua-compatible" content="ie=edge">
        <link rel="stylesheet" href="/css/bootstrap.min.css">
        <link rel="stylesheet" href="/css/codemirror.css">
        <style type="text/css">
        .CodeMirror
        {
                height: auto;
        }
        </style>
</head>

<body>
<div class="container">
        <div class="row">
        <div class="col-md-6">
                <br>
                <textarea id="editor"></textarea>
                <br>
                <span id="errors"></span>
                <button type="button" id="compile" class="btn
btnprimary">Compile</button>
        </div>
        <div class="col-md-6">
```

```
                <br>
                <form>
                <div class="form-group">
                <label for="address">Address</label>
                <input type="text" class="form-control" id="address"
placeholder="Prefixed with 0x">
                </div>
                <div class="form-group">
                <label for="key">Private Key</label>
                <input type="text" class="form-control" id="key"
placeholder="Prefixed with 0x">
                </div>
                <hr>
                <div id="arguments"></div>
                <hr>
                <button type="button" id="deploy" class="btn
btnprimary">Deploy</button>
                </form>
        </div>
        </div>
        </div>
        <script src="/js/codemirror.js"></script>
        <script src="/js/main.js"></script>
</body>
</html>
```

В этом коде только тег `<textarea>` требует отдельного комментария. В этом теге располагается редактор Codemirror, в котором пользователь будет работать с кодом контракта. Остальной код страницы понятен без пояснений.

В листинге 6.4 приведен код JavaScript для клиентской части. Этот код следует поместить в файл `main.js`.

Листинг 6.4. Код JavaScript для файла `main.js`

```
var editor = CodeMirror.fromTextArea(document.getElementById("editor"), {
        lineNumbers: true,
});

var argumentsCount = 0;

document.getElementById("compile").addEventListener("click", function(){
        editor.save();
        var xhttp = new XMLHttpRequest();

        xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
```

```
if(JSON.parse(xhttp.responseText).errors != undefined)
{
        document.getElementById("errors").innerHTML =
        JSON.parse(xhttp.responseText).errors + "<br><br>";
}
else
{
        document.getElementById("errors").innerHTML = "";
}

var contracts = JSON.parse(xhttp.responseText).contracts;

for(var contractName in contracts)
{
        var abi = JSON.parse(contracts[contractName].interface);

        document.getElementById("arguments").innerHTML = "";
for(var count1 = 0; count1 < abi.length; count1++)
{
if(abi[count1].type == "constructor")
{
        argumentsCount = abi[count1].inputs.length;
        document.getElementById("arguments").innerHTML =
        '<label>Arguments</label>';
for(var count2 = 0; count2 < abi[count1].inputs.length; count2++)
{
var inputElement = document.createElement("input");
inputElement.setAttribute("type", "text");
inputElement.setAttribute("class", "form-control");
inputElement.setAttribute("placeholder",
abi[count1].inputs[count2].type);
inputElement.setAttribute("id", "arguments-" + (count2 + 1));
var br = document.createElement("br");
document.getElementById("arguments").appendChild(br);
document.getElementById("arguments").appendChild(inputElement);
}
break;
}
}
break;
}
}
};

xhttp.open("GET", "/compile?code=" +
encodeURIComponent(document.getElementById("editor").value), true);
```

```
        xhttp.send();
})

document.getElementById("deploy").addEventListener("click", function(){
        editor.save();

        var arguments = [];

        for(var count = 1; count <= argumentsCount; count++)
        {
                arguments[count - 1] =
JSON.parse(document.getElementById("arguments-" + count).value);
        }

        var xhttp = new XMLHttpRequest();

        xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200)
        {
                var res = JSON.parse(xhttp.responseText);
                if(res.error)
        {
        alert("Error: " + res.error)
        }
                else
        {
        alert("Txn Hash: " + res.result.hash);
        }
        }
        else if(this.readyState == 4)
        {
                alert("Произошла ошибка.");
        }
        };

        xhttp.open("GET", "/deploy?code=" +
        encodeURIComponent(document.getElementById("editor").value) + "&arguments="
        + encodeURIComponent(JSON.stringify(arguments)) + "&address=" +
        document.getElementById("address").value + "&key=" +
        document.getElementById("key").value, true);
        xhttp.send();
})
```

Код из листинга 6.4 работает следующим образом:

1. Сначала мы добавляем на страницу редактор кода. Он расположен в области, определяемой тегом `<textarea>`, и эта область пока свернута.

2. Далее, у нас есть обработчик нажатия на кнопку **Compile**. Внутри обработчика мы сохраняем содержимое редактора. Когда нажата кнопка **Compile**, мы отправляем запрос к конечной точке `/compile`. Как только получен результат, мы проводим его парсинг и отображаем поля, в которые пользователь может ввести аргументы конструктора. В данном случае мы считываем аргументы конструктора только для первого контракта. Но вы можете доработать интерфейс, чтобы отображать поля ввода для всех контрактов, если их больше одного.
3. Наконец, у нас есть обработчик нажатия кнопки **Deploy**. Мы считываем значения аргументов конструктора и помещаем их в массив. Затем мы формируем запрос к конечной точке `/deploy` и отправляем адрес, ключ, код и значение аргумента. Если возникла ошибка, мы отображаем ее во всплывающем окне. В ином случае отображаем во всплывающем окне хеш транзакции.

Тестирование

Для тестирования приложения запустите скрипт `app.js`, который находится в каталоге `Initial`, и введите в адресной строке браузера: `localhost:8080`. Вы увидите рабочее поле (рис. 6.1).

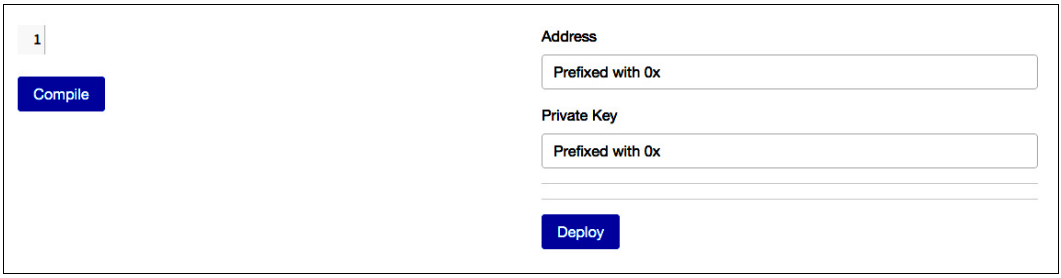

Рис. 6.1. Начальный экран приложения

Теперь введите код какого-нибудь контракта на языке Solidity и нажмите кнопку **Compile** — на правой стороне рабочей панели вы увидите новое поле. В качестве примера на рис. 6.2 показан снимок соответствующего экрана.

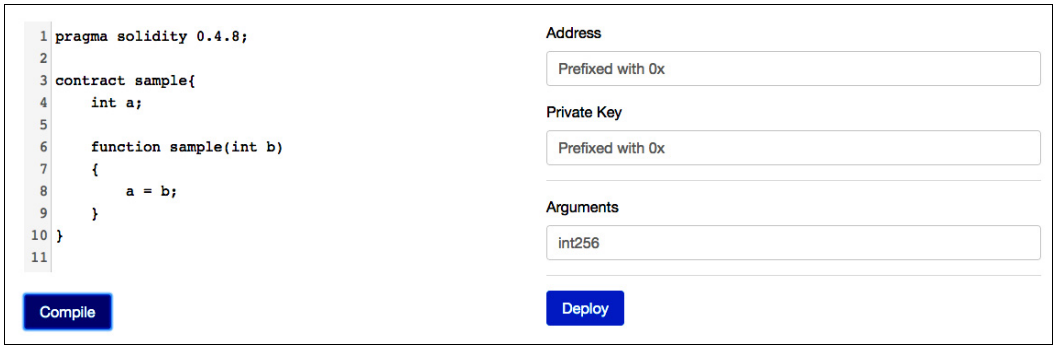

Рис. 6.2. Экран приложения после ввода кода контракта и нажатия кнопки **Compile**

Введите корректный адрес и связанный с ним закрытый ключ. Затем введите значения аргументов конструктора и нажмите кнопку **Deploy**. Если все прошло, как задумано, вы увидите всплывающее сообщение с хешем транзакции (рис. 6.3).

Рис. 6.3. Экран приложения после нажатия кнопки **Deploy**

Заключение

Из этой главы вы узнали, как использовать API пула транзакций, как определять правильное значение `nonce`, как вычислять расходуемый баланс, генерировать данные транзакций, компилировать контракты и т. д. Затем мы создали законченную платформу для компиляции и развертывания контракта. Теперь вы можете двигаться дальше и доработать приложение, чтобы компилировать все контракты, найденные в редакторе, обрабатывать импорты и добавлять библиотеки.

В следующей главе в процессе создания децентрализованного приложения для ставок на результаты матчей вы познакомитесь с сервисом Oraclize.

7

Приложение для ставок на результат матча

Иногда смарт-контракту необходимо получить данные от других децентрализованных приложений или сайтов Интернета. Увы, эта простая операция превращается в сложную задачу из-за проблем с консенсусом и прочих технических трудностей. Поэтому в настоящее время смарт-контракты Ethereum не имеют встроенной поддержки доступа к внешним данным. Но существуют сторонние решения для организации доступа к некоторым популярным приложениям и сайтам. В этой главе мы научимся использовать сервис Oraclize для реализации HTTP-запросов от смарт-контрактов. Мы также узнаем, как получать доступ к файлам, сохраненным в IPFS, как использовать библиотеку для работы со строками и многое другое. Со всем этим мы разберемся в процессе работы над смарт-контрактом для ставок на результаты футбольного матча.

Предупреждение от редакции русского издания

В Российской Федерации, как и в большинстве других стран, организация приема ставок на спорт, а также продажа и эксплуатация соответствующего программного обеспечения и оборудования строго регулируются и ограничиваются государством. Материал этой главы предназначен для учебных целей и демонстрирует методику получения доступа к открытым данным о результатах футбольных матчей. Вы не должны использовать приложения с функциями тотализатора в коммерческих целях, если у вас нет специальной лицензии.

В этой главе будут рассмотрены следующие темы:

- как работает Oraclize?
- что такое источники данных Oraclize и как они работают?
- как работает консенсус в Oraclize?
- интеграция сервиса Oraclize в смарт-контракты;
- использование библиотеки Solidity для работы со строками;
- разработка приложения для ставок на результаты футбольных матчей.

Знакомство с Oraclize

Oraclize — это сервис, который позволяет смарт-контрактам получать доступ к данным из других блокчейнов и Всемирной сети Интернет. Этот сервис работает в сети Bitcoin, а также обеих подсетях Ethereum: testnet и mainnet. Особенностью Oraclize является то, что этот сервис представляет доказательство подлинности данных, и сейчас нам предстоит узнать, как смарт-контракты Ethereum используют Oraclize для получения внешних данных из Интернета.

Как работает Oraclize?

Для получения данных, расположенных вне блокчейна Ethereum — из другого блокчейна или иного источника — смарт-контракт должен отправить в Oraclize запрос, содержащий указание на источник данных и входные аргументы для источника данных (т. е. указание на конкретные данные источника).

Отправка запроса в сервис Oraclize фактически означает вызов смарт-контракта Oraclize, расположенного в блокчейне Ethereum.

Сервер Oraclize постоянно следит за входящими запросами к его смарт-контракту, и как только обнаружен новый запрос, он формирует результат и возвращает его вашему контракту при помощи вызова служебного метода `__callback()` вашего контракта.

Источники данных

Сервис Oraclize позволяет получать данные из следующих источников[1]:

- `URL` — источник типа URL позволяет делать запросы HTTP GET или HTTP POST и получать данные от любого сервера в сети Интернет по заданному адресу;
- `WolframAlpha` — отправка запроса в базу знаний WolframAlpha[2];
- `Blockchain` — получение данных из других блокчейнов. Источнику типа `Blockchain` можно отправить запросы `bitcoin blockchain height`, `litecoin hashrate`, `bitcoin difficulty`, `balance` и некоторые другие[3];
- `IPFS` — получение содержимого файла, сохраненного в файловой системе IPFS;
- `Nested` — это источник метаданных. Он не предоставляет доступ к другим сервисам и реализует простую логику агрегирования, позволяя одному запросу использовать вложенные (nested) подзапросы к любым доступным источникам данных в виде единой строки, например:

```
[WolframAlpha] temperature in ${[IPFS]
QmP2ZkdsJG7LTw7jBbizTTgY1ZBeen64PqMgCAWz2koJBL}.
```

[1] См. **https://docs.oraclize.it/#general-concepts-data-source-types**.

[2] См. **https://ru.wikipedia.org/wiki/WolframAlpha**.

[3] Этот источник данных отсутствовал в документации Oraclize в период подготовки перевода.

- `Computation` — позволяет запустить контролируемое выполнение заданного приложения в безопасном окружении вне цепочки блокчейна. Иными словами, этот источник позволяет получить результат работы приложения вне цепочки. Приложение должно вывести результат в последней строке стандартного вывода прежде, чем завершит работу. Контекст выполнения должен описываться файлом Docker, а его создание и запуск должны немедленно запускать основное приложение. Инициализация файла Docker и выполнение приложения должны быть завершены как можно скорее. Максимальное ожидание выполнения для экземпляра `AWS t2.micro` не превышает пяти минут.

  В данном случае речь идет об экземпляре `AWS t2.micro`, потому что Oraclize использует его для выполнения приложения. Входным параметром этого источника является мультихеш ZIP-архива в файловой системе IPFS, который содержит файл Docker и файлы внешних зависимостей (файл Docker помещают в корень архива). Поэтому вы должны заранее позаботиться о создании этого архива и поместить его в IPFS.

Указанные источники данных были доступны на момент подготовки книги. Однако, скорее всего, в будущем появятся новые источники.

Дополнение от переводчика

К моменту начала работы над переводом книги в документации Oraclize говорилось о новых источниках данных, в книге не упомянутых:

- `Random` — неискаженная случайная последовательность байтов, поступающая от защищенного приложения, запущенного на аппаратном кошельке Ledger Nano S (см. **https://www.ledgerwallet.com/products/ledger-nano-s)**;
- `Identity` — возвращает запрос;
- `Decrypt` — расшифровывает строку, зашифрованную закрытым ключом Oraclize.

Доказательство подлинности

Несмотря на то, что Oraclize — это доверенный сервис, вы можете по-прежнему испытывать потребность удостовериться в подлинности полученных данных. Вдруг кто-то посторонний модифицировал данные по пути от Oraclize к вам, или сам сервис Oraclize манипулирует данными?

Oraclize опционально поддерживает доказательство подлинности TLSNotary для результатов, возвращенных источниками типа `URL`, `blockchain`, `nested` и `computation`, но для источников `WolframAlpha` и `IPFS` такое доказательство недоступно. Сейчас Oraclize поддерживает только доказательство TLSNotary, но в будущем он может использовать другие способы подтверждения подлинности. Пока что доказательство TLSNotary требует выполнения действий по проверке вручную, но Oraclize работает над верификацией через блокчейн. Иными словами, ваш смарт-контракт сможет использовать доказательство TLSNotary самостоятельно и отвергать доказательство, если сервис пометил его, как некорректное.

Oraclize предоставляет нам программный инструмент с открытым кодом для проверки доказательства TLSNotary. Он расположен по адресу: **https://github.com/Oraclize/proof-verification-tool**.

Инструмент для проверки доказательства TLSNotary представляет собой приложение с открытым исходным кодом. Любой вредоносный код в его составе был бы моментально обнаружен экспертами сообщества программистов, поэтому инструменту можно доверять. Для использования Oraclize или проверки доказательства подлинности не требуется понимания того, как работает TLSNotary.

Рассмотрим все же, хотя бы в общих чертах, как работает TLSNotary. Но сначала следует понять, как работает протокол TLS. Протокол TLS обеспечивает зашифрованную сессию клиент-сервер, и посторонние лица не могут манипулировать данными обмена между клиентом и сервером. Сервер отправляет клиенту свой сертификат безопасности, который выпускает владелец домена при участии удостоверяющего органа. Сертификат содержит открытый ключ сервера. Клиент расшифровывает сертификат при помощи открытого ключа *удостоверяющего органа* и, убедившись таким образом, что сертификат действительно выдан удостоверяющим органом, извлекает из сертификата открытый ключ *сервера*. Затем клиент генерирует свой симметричный ключ и MAC-ключ (Message Authentication Code, код проверки сообщения), шифрует их при помощи открытого ключа сервера и отправляет серверу. Сервер может расшифровать это сообщение, потому что у него тоже есть этот открытый ключ. Теперь клиент и сервер оба имеют одинаковый симметричный ключ и MAC-ключ, но никому из посторонних эти ключи неизвестны. Поэтому клиент и сервер могут безопасно обмениваться данными. Симметричный ключ применяется для шифрования и расшифровки сообщений, а MAC-ключ вместе с симметричным ключом применяется для генерации уникальной подписи зашифрованного сообщения. Если третья сторона вмешается и модифицирует сообщение, участники сессии смогут узнать об этом.

TLSNotary — это специальная модификация TLS. Сервис Oraclize представляет криптографическое доказательство того, что данные, полученные вашим смарт-контрактом, ничем не отличаются от данных Oraclize в определенный момент времени. Фактически, протокол TLSNotary — это технология с открытым кодом, разработанная и применяемая проектом PageSigner.

TLSNotary работает путем разделения симметричного ключа и MAC-ключа между тремя сторонами, т. е. сервером, аудируемой стороной[4] и аудитором. Основная идея TLSNotary заключается в том, что аудируемый может доказать аудитору, что определенный результат был возвращен сервером в данный момент времени.

Как TLSNotary решает эту задачу? Допустим, банк передал данные смарт-контракту через сервис Oraclize. Мы хотим иметь уверенность, что Oraclize не подменил данные. Аудитор вычисляет симметричный ключ и MAC-ключ, но передает в Oraclize только симметричный ключ. В данном случае Oraclize не нуждается

[4] Сторона, подвергнутая аудиторской проверке. В нашем случае аудируемым является сервис Oraclize.

в MAC-ключе, потому что цифровая подпись на основе MAC показывает, что данные не были модифицированы по пути из банка. Имея симметричный ключ, Oraclize теперь может расшифровать данные банковского сервера и отдать их смарт-контракту. Иными словами, все сообщения «заверены» банком при помощи MAC-ключа, но только банк и аудитор знают этот ключ. Поэтому наличие корректной MAC-подписи на стороне получателя данных дает доказательство, что посредник (в лице Oraclize) не подменил данные.

В нашем случае объектом аудиторской проверки является сервис Oraclize, а в роли аудитора выступает специальный экземпляр AWS (Amazon Web Services) — защищенного облачного сервиса Amazon с открытым исходным кодом.

Кроме доказательства подлинности сообщений, переданных через TLSNotary, аудитор предоставляет дополнительные доказательства того, что программное обеспечение самого экземпляра AWS не было модифицировано с момента инициализации[5].

Стоимость услуг Oraclize

Первый запрос к Oraclize с любого адреса Ethereum — бесплатный. Кроме того, бесплатными являются все запросы из тестового сегмента testnet. Но это применимо только для ограниченного пользования в тестовых целях.

Начиная со второго запроса, вы должны платить за услуги сервиса. При обращении к сервису Oraclize определенная сумма эфира вычитается из вызывающего контракта в пользу контракта Oraclize. Величина платежа зависит от источника данных и типа доказательства.

В табл. 7.1 приведена стоимость одного запроса для разных источников данных.

Таблица 7.1. *Стоимость услуг сервиса Oraclize в долларах США*[6]

Источник данных	Базовая стоимость	Тип доказательства			
		Нет	TLSNotary	Android	Ledger
URL	0.01	+0.0	+0.04	+0.04	—
WolframAlpha	0.03	+0.0	—	—	—
IPFS	0.01	+0.0	—	—	—
random	0.05	+0.0	—	—	+0.0
computation	0.50	+0.0	+0.04	+0.04	—

[5] «Кто заверит заверяющего?» — это общая проблема всех систем, использующих заверение и аттестацию. В данном случае гарантией честности аудитора является репутация компании Amazon, стоимость которой на много порядков больше, чем выгода от манипуляций с данными Oraclize.

[6] Расценки с официального сайта **https://docs.oraclize.it** на момент подготовки перевода. Стоимость указана в долларах США, но оплата взимается в номинале Wei по текущему обменному курсу эфир/доллар.

Например, если вы делаете HTTP-запрос и нуждаетесь в доказательстве TLSNotary, то вызывающий контракт должен заплатить 0,05 доллара, в противном случае будет сгенерировано исключение (отказ по ошибке оплаты).

Основы работы с API Oraclize

Контракт, который использует сервис Oraclize, должен наследовать контракт `usingOraclize`. Вы можете найти этот контракт по адресу: **https://github.com/Oraclize/Ethereum-api**.

Контракт `usingOraclize` выступает в качестве посредника для контрактов `OraclizeI` и `OraclizeAddrResolverI`. Фактически `usingOraclize` упрощает вызовы контрактов `OraclizeI` и `OraclizeAddrResolverI`, потому что предлагает простой API. Но вы, если хотите, можете вызывать контракты `OraclizeI` и `OraclizeAddrResolverI` напрямую — возьмите исходный код этих контрактов и найдите там все компоненты API. Мы же в этой книге рассмотрим только самые необходимые функции.

Давайте узнаем, как указать тип доказательства и место его хранения, как делать запросы, как находить стоимость запроса и пр.

Настройка типа и места хранения доказательства

Независимо от того, собираетесь вы использовать TLSNotary или нет, перед отправкой запросов вы должны задать тип доказательства и хранилище доказательства.

Если вам не требуется доказательство подлинности, поместите в свой контракт такую строку кода:

```
oraclize_setProof(proofType_NONE)
```

А если доказательство требуется, помещаемая в свой контракт строка кода должна выглядеть так:

```
oraclize_setProof(proofType_TLSNotary | proofStorage_IPFS)
```

В настоящее время в качестве хранилища доказательства можно использовать только IPFS, поэтому мы указываем хранилище `proofStorage_IPFS`.

Вы можете выполнить любой из этих методов однократно — например, в конструкторе или в любое другое время (если доказательство требуется только для некоторых запросов).

Отправка запросов

Для отправки запроса в Oraclize вам потребуется функция `oraclize_query()`. Эта функция ожидает, как минимум, два аргумента: источник данных и входные параметры для источника. Параметры источника данных не чувствительны к регистру.

Рассмотрим несколько простых примеров вызова функции `oraclize_query()`:

1. `oraclize_query("WolframAlpha", "random number between 0 and 100");`
2. `oraclize_query("URL", "https://api.kraken.com/0/public/Ticker?pair=ETHXBT");`
3. `oraclize_query("IPFS","QmdEJwJG1T9rzHvBD8i69HHuJaRgXRKEQCP7Bh1BVttZbU");`
4.
```
oraclize_query("URL", "https://xyz.io/makePayment",
                                          '{"currency": "USD","amount": "1"}');
```

Фрагменты кода из примеров работают следующим образом:

- если первый аргумент — строка, то подразумевается, что это источник данных, а второй аргумент считается входным параметром источника данных. Так, в первом примере источником данных является WolframAlpha, а параметром запроса к источнику данных — строка `"random number between 0 and 100"` («случайное число в диапазоне от 0 до 100»)[7];
- во втором примере мы выполняем запрос `HTTP GET` по адресу, указанному во втором аргументе;
- в третьем примере мы получаем содержимое файла с идентификатором `QmdEJwJG1T9rzHvBD8i69HHuJaRgXRKEQCP7Bh1BVttZbU` из IPFS;
- если после источника данных два аргумента подряд являются строками, это считается POST-запросом. В четвертом примере мы выполняем запрос HTTP POST по адресу `https://xyz.io/makePayment` и тело POST-запроса содержит строку из третьего аргумента. Сервис Oraclize достаточно разумен, чтобы распознать тип запроса по формату строки.

Отложенные запросы

Если вы хотите, чтобы Oraclize выполнил запрос не сразу, а в запланированное время, просто укажите в первом аргументе запроса задержку в секундах относительно текущего времени:

```
oraclize_query(60, "WolframAlpha", "random number between 0 and 100");
```

Этот запрос будет выполнен спустя 60 секунд после получения. Следовательно, если первым аргументом запроса является число, это отложенный запрос.

Расходование газа

Ответные транзакции, приходящие из Oraclize в адрес вашей функции обратного вызова `__callback`, расходуют на оплату услуг майнеров некоторое количество газа, как и любая другая транзакция. Майнеры получают оплату, номинированную в эфирах. Величина оплаты вычисляется как количество газа, необходимого для

[7] Сервис WolframAlpha может работать с запросами в свободной речевой форме. В данном случае он вернет нам случайное число из указанного диапазона. — *Прим. пер.*

выполнения транзакции, умноженного на стоимость единицы газа. Oraclize по умолчанию закладывает в обеспечение транзакции 200 000 единиц газа[8].

Вы можете по своему усмотрению назначить лимит газа[9]. Для задания пользовательского количества газа используется аргумент `_gasLimit` функции `oraclize_query()`, например:

```
oraclize_query("WolframAlpha", "random number between 0 and 100", 500000);
```

В этом примере мы увеличили лимит газа транзакции до 500 000 единиц. Поскольку мы указали сервису Oraclize потратить больше газа, с вашего баланса будет списано больше эфира.

Учтите, что неизрасходованный газ возвращается Oraclize, а не вам!

Разработчик смарт-контракта должен позаботиться о том, чтобы минимизировать расход газа.

Дополнение от переводчика

Для задания пользовательской стоимости газа применяется метод `oraclize_setCustomGasPrice()`, например:

```
oraclize_setCustomGasPrice(4000000000 wei);
```

В этом примере мы задали цену газа 4 GWei. По неизвестной причине упоминание об этом методе отсутствует в авторском тексте книги.

Имейте в виду: если вы установили недостаточно большой лимит газа, а ваша функция обратного вызова ресурсоемкая, вы рискуете никогда не получить ответ. Также учтите, что пользовательский лимит газа должен быть больше 200 000 единиц.

Функции обратного вызова

Когда результат запроса готов, Oraclize отправляет транзакцию обратно вашему контракту и вызывает один из трех методов:

- `__callback(bytes32 myid, string result)`. Здесь `myid` — это уникальный идентификатор запроса, который возвращается методом `oraclize_query()`. Если ваш контракт сделал несколько вызовов метода `oraclize_query()`, то по идентификатору вы можете определить, для какого запроса предназначен ответ;
- если вы запросили доказательство TLSNotary, то обратный вызов должен иметь вид: `__callback(bytes32 myid, string result, bytes proof)`;
- наконец, если в вашем контракте отсутствуют другие методы, будет вызвана резервная функция `function()` (см. *главу 3*).

[8] При стоимости газа 20 GWei. На сегодняшний день — это наибольшее значение в спектре рыночных цен, зато оно способствует быстрой обработке транзакции. Вознаграждение майнера списывается с баланса контракта, когда ответная транзакция обработана.

[9] И его стоимость.

Рассмотрим пример использования функции `__callback`:

```
function __callback(bytes32 myid, string result) {
//убеждаемся, что ответ получен от авторизованного сервиса Oraclize
if (msg.sender != oraclize_cbAddress()) throw;
//делаем что-нибудь с результатом...
}
```

Синтаксический разбор результатов

Если мы отправили HTTP-запрос, то ответ стороннего источника может поступить в формате HTML, JSON, XML, в двоичном виде и т. д. Синтаксический разбор ответа в контракте на языке Solidity труден и дорого стоит. По этой причине Oraclize предоставляет нам *средства разбора* (parsing helper), способные выполнять синтаксический разбор на серверах Oraclize, а вы получаете нужную часть ответа в готовом виде.

Чтобы попросить Oraclize провести разбор ответа, вы должны «обернуть» URL в одно из следующих средств разбора:

- средства `xml(...)` и `json(...)` позволяют запросить у Oraclize только часть разобранного ответа в формате JSON или XML, например:
  - чтобы получить ответ целиком, мы используем источник данных типа `URL` и аргумент: `api.kraken.com/0/public/Ticker?pair=ETHUSD`;
  - если вы хотите получить только последнее поле ответа, следует использовать средство разбора JSON:
    `json(api.kraken.com/0/public/Ticker?pair=ETHUSD).result.XETHZUSD.c.0`;
- средство `html(...).xpath(...)` применяется для обработки HTML. Просто укажите XPATH, который вам нужен, в качестве аргумента `xpath(...)`. Например, чтобы получить текст определенного твита:

  ```
  html(https://twitter.com/oraclizeit/status/671316655893561344).
  xpath(//*[contains(@class, 'tweettext')]/text());
  ```

- средство `binary(...)` позволяет выделять из ответа двоичные файлы, например файлы сертификатов. Чтобы получить только часть двоичного файла, вы можете использовать метод `slice(offset,length)`. Первый параметр здесь — смещение, а второй — длина фрагмента. Оба параметра указывают в байтах. Например, извлечем первые 300 байтов из двоичного сертификата CRL:

  ```
  binary(https://www.sk.ee/crls/esteid/esteid2015.crl).slice(0,300).
  ```

  Средство `binary` должно применяться только с опцией `slice` и обрабатывает только двоичные файлы.

Если запрошенный вами сервер не отвечает или недоступен, то сервис Oraclize возвращает пустой ответ. Для проверки своих запросов вы можете воспользоваться специальной тестовой формой по адресу:
http://app.Oraclize.it/home/test_query.

Получение цены запроса

Если вы хотите узнать, сколько будет стоить запрос, воспользуйтесь функцией `Oraclize.getPrice()`. Первый аргумент содержит указание на источник данных, второй (опциональный) аргумент задает пользовательское значение газа.

Обычно эту функцию применяют, чтобы предложить пользователю пополнить баланс контракта, если его недостаточно для выполнения запроса.

Шифрование запросов

Исходный код смарт-контракта доступен для просмотра посторонними лицами. Но иногда необходимо скрыть от посторонних глаз источник данных контракта или параметры, передаваемые контракту. Например, вы, будучи разработчиком, не хотите раскрывать пользователям смарт-контракта личный ключ API для определенного сайта. Сервис Oraclize позволяет включать в исходный код смарт-контракта зашифрованные запросы, и только у сервера Oraclize есть ключ для их расшифровки.

Oraclize предоставляет нам инструмент на языке Python (**https://github.com/Oraclize/encrypted-queries**), который можно использовать для шифрования адреса источника и/или аргументов запроса. Он генерирует недетерминированную зашифрованную строку.

Команда терминала для шифрования произвольной строки текста выглядит приблизительно так:

```
python encrypted_queries_tools.py -e -p
044992e9473b7d90ca54d2886c7addd14a61109af202f1c95e218b0c99eb060c7134c4ae46345d0383a
c996185762f04997d6fd6c393c86e4325c469741e64eca9 "YOUR DATASOURCE or INPUT"
```

Длинная шестнадцатеричная последовательность — это открытый ключ сервера Oraclize. Теперь вы можете взять результат работы этой команды и поместить его в свой код на место источника данных и/или аргументов источника.

Чтобы предотвратить некорректное использование шифрованных запросов (т. е. «атаку повторением»), законным получателем данных считается только первый контракт, отправивший шифрованный запрос. Любой другой контракт, повторно отправивший такую же шифрованную строку, получит пустой ответ. Поэтому не забывайте генерировать новую строку шифрованного запроса для использования в других контрактах.

Расшифровка источника данных

Существует особый источник данных, который называется `decrypt`. Он используется для расшифровки зашифрованных строк запросов. Но этот источник не возвращает никакого результата, в противном случае любой желающий смог бы расшифровать источник данных и его аргументы.

Источник `decrypt` разработан специально для обработки вложенных (`nested`) источников данных с частичным шифрованием запроса. В этом заключается его единственное предназначение.

IDE Oraclize

Oraclize располагает онлайновой средой разработки (IDE), при помощи которой вы можете писать, компилировать и тестировать приложения, основанные на сервисе Oraclize. Среда разработки расположена по адресу: **http://dapps.Oraclize.it/browser-Solidity/**[10].

Если вы воспользуетесь средой разработки Oraclize, то увидите, что она похожа на браузер Solidity. Это и есть браузер Solidity с дополнительными опциями. Чтобы понять, что это за опции, следует узнать больше о браузере Solidity.

Браузер Solidity позволяет не только писать, компилировать и генерировать код web3.js для наших контрактов, но и тестировать эти контракты внутри себя. До сих пор, чтобы протестировать контракт, мы запускали собственный узел Ethereum и отправляли ему транзакцию. Но браузер Solidity может выполнять контракты без соединения с каким-либо узлом, и все действия происходят в памяти. Это возможно благодаря EthereumJS-VM — реализации виртуальной машины Ethereum на языке JavaScript. При помощи EthereumJS-VM вы можете создать собственную виртуальную машину и выполнить байт-код. Если мы захотим, то можем настроить браузер Solidity для подключения к узлу Ethereum, просто указав URL для подключения. Интерфейс браузера очень информативен, поэтому вы можете испробовать все возможности самостоятельно.

Особенностью Oraclize Web IDE является размещение контракта Oraclize во внутреннем пространстве памяти — следовательно, нет необходимости подключаться к сети testnet или mainnet. Но если вы используете браузер Solidity, вам нужно подключиться к узлу сети testnet или mainnet для проверки взаимодействия с API Oraclize.

Работа со строками

Работа со строками в языке Solidity не столь проста, как в других языках высокого уровня типа JavaScript, Python и др. Поэтому программисты придумывают различные библиотеки и контракты, чтобы облегчить работу со строками.

Библиотека strings — одна из наиболее популярных библиотек для обработки строк. Она позволяет объединять, разделять, сравнивать строки и выполнять другие операции путем преобразования строки в так называемый *срез* (slice). Срез — это структура, которая содержит длину строки и адрес строки в памяти. Поскольку срез строки обозначает только смещение и длину, копирование и манипуляции со срезами обходятся намного дешевле, чем аналогичные действия над соответствующими строками.

Чтобы дополнительно снизить затраты на газ, большинство функций, работающих со срезом и возвращающих срез, изменяют оригинал вместо создания новой строки.

[10] В период работы над русским переводом эта ссылка перестала работать. Теперь разработчикам контрактов доступен Центр разработок Oraclize по адресу **https://dev.oraclize.it/**.

Например, функция `s.split(".")` вернет текст до первого символа "." и урежет исходную строку `s`. В ситуации, когда вы не хотите изменять исходную строку, можете сделать ее копию при помощи функции `copy()` — например: `s.copy().split(".")`. Но старайтесь избегать использования этой конструкции в циклах — Solidity не имеет управления памятью, и создание копий в цикле приведет к замусориванию памяти множеством недолговечных срезов, которые позже пропадут.

Функции, которые должны копировать строковые данные, возвращают строки, а не срезы. При необходимости мы можем получить срезы этих строк для дальнейшей обработки.

Рассмотрим несколько простых примеров использования библиотеки strings:

```
pragma Solidity ^0.4.0;
import "github.com/Arachnid/Solidity-stringutils/strings.sol";
contract Contract
{
        using strings for *;

        function Contract()
        {

        //конвертируем строку в срез
        var slice = "xyz abc".toSlice();

        //длина строки
        var length = slice.len();

        /разделяем строку
        //подсрез = xyz
        //срез = abc
        var subslice = slice.split(" ".toSlice());

        //преобразуем строку в массив
        var s = "www.google.com".toSlice();
        var delim = ".".toSlice();
        var parts = new string[](s.count(delim));
        for(uint i = 0; i < parts.length; i++) {
        parts[i] = s.split(delim).toString();
        }

        //преобразуем срез обратно в строку
        var myString = slice.toString();

        //конкатенация строк
        var finalSlice = subslice.concat(slice);
```

```
        //проверка равенства двух строк
        if(slice.equals(subslice))
        {
        }
        }
}
```

Код этого примера не нуждается в пояснениях.

Функции, которые используют два среза, существуют в двух вариантах: функция в *неразмещающем* (nonallocating) варианте получает второй срез как аргумент и модифицирует на исходном месте, функция в *размещающем* (allocating) варианте размещает в памяти и возвращает второй срез. Рассмотрим следующий код в качестве примера:

```
var slice1 = "abc".toSlice();

//переносит указатель среза строки slice1 на следующий символ
//и возвращает срез, который содержит только один символ
var slice2 = slice1.nextRune();

var slice3 = "abc".toSlice();
var slice4 = "".toSlice();

//извлекает первый символ из среза slice3 в срез slice4,
//смещает указатель на следующий символ и возвращает срез slice4
var slice5 = slice3.nextRune(slice4);
```

Вы можете прочитать больше о библиотеке strings по адресу: **https://github.com/Arachnid/Solidity-stringutils**.

Разработка контракта для ставок на спорт

В нашем приложении два человека могут сделать ставку на результат футбольного матча в форме пари: один участник ставит на домашнюю команду, а второй — на команду гостей. Оба участника должны поставить одинаковую сумму, и победитель забирает все деньги. Если матч закончился вничью, оба участника забирают свои деньги обратно.

Для получения результатов матчей мы воспользуемся API сервиса FastestLiveScores. Этот сервис предоставляет бесплатный[11] API для 100 запросов в час. Сначала создайте учетную запись на этом сервисе и получите ключ API. Для регистрации

[11] Бесплатный пробный доступ действует только 30 дней. Постарайтесь успеть завершить работу над своим контрактом за это время.

перейдите по адресу: **https://customer.fastestlivescores.com/register**. После завершения регистрации ваш ключ будет виден по адресу:

https://customer.fastestlivescores.com/.

Мы будем развертывать отдельный контракт для каждого пари между двумя участниками. Контракт должен содержать идентификатор матча, полученный от API FastestLiveScores, количество валюты в Wei, которое ставит каждый участник, и Ethereum-адреса участников. Как только обе стороны внесли ставку в контракт, он проверяет результат матча. Если матч еще не завершился, контракт будет проверять результат каждые 24 часа.

В листинге 7.1 приведен исходный код контракта.

Листинг 7.1. Контракт для заключения пари на результат матча

```
pragma Solidity ^0.4.0;

import "github.com/Oraclize/Ethereum-api/oraclizeAPI.sol";
import "github.com/Arachnid/Solidity-stringutils/strings.sol";

contract Betting is usingOraclize
{
        using strings for *;
        string public matchId;
        uint public amount;
        string public url;
        address public homeBet;
        address public awayBet;
        function Betting(string _matchId, uint _amount, string _url)
        {
                matchId = _matchId;
                amount = _amount;
                url = _url;
                oraclize_setProof(proofType_TLSNotary | proofStorage_IPFS);
        }

        //1 обозначает домашнюю команду
        //2 обозначает гостевую команду
        function betOnTeam(uint team) payable
        {
                if(team == 1)
                {
                        if(homeBet == 0)
                {
                if(msg.value == amount)
                {
                        homeBet = msg.sender;
```

```
        if(homeBet != 0 && awayBet != 0)
        {
                oraclize_query("URL", url);
        }
        }
        else
        {
                throw;
        }
        }
        else
        {
                throw;
        }
        }
        else if(team == 2)
        {
                if(awayBet == 0)
        {
        if(msg.value == amount)
        {
                awayBet = msg.sender;
                if(homeBet != 0 && awayBet != 0)
                {
                        oraclize_query("URL", url);
                }
        }
        else
        {
                throw;
        }
        }
        else
        {
                throw;
        }
        }
        else
        {
                throw;
        }
}

function __callback(bytes32 myid, string result, bytes proof) {
        if (msg.sender != oraclize_cbAddress())
```

```
                {
                        throw;
                }
                else
                {
                        if(result.toSlice().equals("home".toSlice()))
                        {
                                homeBet.send(this.balance);
                        }
                else if(result.toSlice().equals("away".toSlice()))
                {
                        awayBet.send(this.balance);
                }
                else if(result.toSlice().equals("draw".toSlice()))
                {
                        homeBet.send(this.balance / 2);
                        awayBet.send(this.balance / 2);
                }
                else
                {
                        if (Oraclize.getPrice("URL") < this.balance)
                        {
                                oraclize_query(86400, "URL", url);
                        }
                }
                }
        }
}
```

Код этого контракта не нуждается в дополнительных пояснениях. Теперь скомпилируйте контракт при помощи solc.js или браузера Solidity — как вам удобнее. Во втором случае можно будет не импортировать библиотеку strings, потому что все функции будут находиться во внутренней области видимости.

Если вы работаете с браузером Solidity и указали для импорта библиотеки или контракта HTTP-адрес, убедитесь, что он расположен на домене GitHub. В противном случае импорт не сработает. В адресе файла удалите как фрагмент протокола **https//:**, так и фрагмент **blob/{branch-name}**.

Разработка приложения для ставок

Чтобы упростить поиск матча по идентификатору, развертывание контракта и оплату ставок, необходимо разработать приложение, которое реализует интерфейс пользователя. Вот мы и разработаем приложение, которое будет использовать два пути: домашний путь — для развертывания контракта и ставок на матчи, и внеш-

ний путь — для получения списка матчей. Мы дадим возможность пользователю развертывать контракты и делать ставки, используя свои оффлайновые счета[12], поэтому приложение получится вполне децентрализованным, и никто не сможет жульничать.

Прежде чем приступить к разработке приложения, убедитесь, что вы синхронизировали свой узел с подсетью testnet, потому что сервис Oraclize работает только с сетью Ethereum, но не с частными сетями. Вы можете переключиться в testnet, заменив опцию `--dev` на `--testnet` и скачав блокчейн тестовой подсети. Например, команда запуска Geth может выглядеть так:

```
geth --testnet --rpc --rpccorsdomain "*" --rpcaddr "0.0.0.0" --rpcport "8545"
```

Разработка структуры приложения

В файлах папки упражнений, дополняющих эту главу (см. *приложение*), вы найдете два каталога: `Final` и `Initial`. Каталог `Final` содержит окончательный исходный код проекта, в то время как `Initial` содержит пустые файлы исходного кода и библиотеки, что позволяет быстро начать работу над проектом.

Для тестирования содержимого каталога `Final` следует выполнить команду терминала **`npm install`** внутри каталога. После этого запустите приложение командой терминала **`node app.js`** внутри каталога `Final`.

В каталоге `Initial` находится вложенный каталог `public` и два файла: `app.js` и `package.json`. Второй файл содержит зависимости для вашего приложения, а файл `app.js` — это место, в котором будет храниться серверная часть проекта.

Каталог `public` содержит файлы, относящиеся к пользовательскому интерфейсу (клиентская часть). Внутри `public/css` вы найдете библиотеку фреймворка Bootstrap `bootstrap.min.css`, а внутри `public/html` находятся файлы `index.html` и `matches.ejs`. В каталоге `public/js` находятся JavaScript-файлы для web3js и ethereumjs-tx. Также внутри этого каталога вы найдете файл `main.js`, в который поместите код JavaScript для клиентской части проекта. Дополнительно, в каталоге `Final` находится инструмент на языке Python для шифрования запросов.

Разработка серверной части

Приступим к разработке серверной части приложения. Прежде всего, выполните команду **`npm install`** внутри каталога `Initial`, чтобы установить зависимости для нашей серверной части.

Далее приведен простой и короткий код, который запускает службу `express`, готовит к выдаче файл `index.html` и статические файлы и запускает службу визуализации (view engine):

[12] См. в *главе 5 разд. «Различие между онлайн- и оффлайн-кошельками».*

```
var express = require("express");
var app = express();

app.set("view engine", "ejs");

app.use(express.static("public"));

app.listen(8080);

app.get("/", function(req, res) {
        res.sendFile(__dirname + "/public/html/index.html");
})
```

Этот код не требует пояснений, поэтому продолжим нашу работу. У приложения будет еще одна страница, отображающая список последних матчей, их номера и результаты тех матчей, которые завершились. Вот фрагмент кода, который отвечает за отображение этой страницы:

```
var request = require("request");
var moment = require("moment");

app.get("/matches", function(req, res) {
request("https://api.crowdscores.com/v1/matches?api_key=7b7a988932de4eaab4e
d1b4dcdc1a82a", function(error, response, body) {
        if (!error && response.statusCode == 200) {
        body = JSON.parse(body);

        for (var i = 0; i < body.length; i++) {
        body[i].start = moment.unix(body[i].start /
        1000).format("YYYY MMM DD hh:mm:ss");
        }

        res.render(__dirname + "/public/html/matches.ejs", {
        matches: body
        });
        } else {
                res.send("Произошла ошибка");
                }
        })
})
```

В этой части кода мы создаем запрос к API для получения списка последних матчей и передаем результат в движок визуализации `matches.ejs`, который формирует интерфейс, удобный для восприятия человеком. API возвращает время начала матча в виде метки времени, поэтому мы используем библиотеку `moment` для конвертации в понятный человеку формат. Мы выполняем запрос на стороне сервера, а не из клиентской части, поэтому пользователи не могут увидеть ключ API.

В свою очередь, наша серверная часть предоставляет API для клиентской части. Используя этот API, клиент может зашифровать запрос перед развертыванием контракта. Наше приложение не станет предлагать пользователю создать ключ API, поскольку это плохое решение, которое портит впечатление от интерфейса. В данном случае, если разработчик приложения сам управляет ключами интерфейса, это не вызовет опасения, потому что разработчик не может модифицировать результаты, поступающие с сервера. Следовательно, пользователь может доверять разработчику, который знает ключи API.

Ознакомьтесь с фрагментом кода для шифрования запроса:

```
var PythonShell = require("python-shell");

app.get("/getURL", function(req, res) {
var matchId = req.query.matchId;

var options = {
args: ["-e", "-p",
"044992e9473b7d90ca54d2886c7addd14a61109af202f1c95e218b0c99eb060c7134c4ae46
345d0383ac996185762f04997d6fd6c393c86e4325c469741e64eca9",
"json(https://api.crowdscores.com/v1/matches/" + matchId +
"?api_key=7b7a988932de4eaab4ed1b4dcdc1a82a).outcome.winner"],
scriptPath: __dirname
};

PythonShell.run("encrypted_queries_tools.py", options, function
(err, results) {
       if(err)
       {
              res.send("An error occured");
       }
       else
       {
              res.send(results[0]);
       }
       });
})
```

Мы уже знаем, как использовать этот инструмент. Для успешного запуска в вашей системе должен быть установлен Python. Но даже если Python и установлен, этот код может возвратить ошибку, которая означает, что не установлены модули `cryptography` и `base58`. Установите их, если инструмент шифрования выдаст вам запрос на установку.

Разработка клиентской части

Приступим к разработке клиентской части приложения. Она позволит пользователю просматривать список последних матчей, развертывать контракты ставок, делать ставки на игру, а затем проверять состояние контракта.

Сначала создадим файл `matches.ejs`, который визуализирует список последних матчей (листинг 7.2).

Листинг 7.2. Полный код файла визуализации `matches.ejs`

```
<!DOCTYPE html>
<html lang="en">
<head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initialscale=1,
                                                                shrink-to-fit=no">
        <meta http-equiv="x-ua-compatible" content="ie=edge">
        <link rel="stylesheet" href="/css/bootstrap.min.css">
</head>

<body>
        <div class="container">
        <br>
        <div class="row m-t-1">
        <div class="col-md-12">
        <a href="/">Home</a>
        </div>
        </div>
        <br>
        <div class="row">
        <div class="col-md-12">
        <table class="table table-inverse">
        <thead>
        <tr>
                <th>Номер матча</th>
                <th>Время начала</th>
                <th>Домашняя команда</th>
                <th>Гостевая команда</th>
                <th>Победитель</th>
        </tr>
        </thead>
        <tbody>
        <% for(var i=0; i < matches.length; i++) {%>
        <tr>
        <td><%= matches[i].dbid %></td>
```

```
        <% if (matches[i].start) { %>
        <td><%= matches[i].start %></td>
        <% } else { %>
        <td>Time not finalized</td>
        <% } %>
        <td><%= matches[i].homeTeam.name%></td>
        <td><%= matches[i].awayTeam.name%></td>
        <% if (matches[i].outcome) { %>
        <td><%= matches[i].outcome.winner%></td>
        <% } else { %>
        <td>Матч не закончился</td>
        <% } %>
        </tr>
        <% } %>
        </tbody>
        </table>
        </div>
        </div>
        </div>
</body>
</html>
```

Код из листинга 7.2 очевиден и не требует пояснений. Теперь напишем HTML-код домашней страницы. Она будет отображать три формы: первая форма предназначена для размещения контракта ставки, вторая форма — для внесения суммы ставки и третья форма отображает информацию о размещенном контракте.

В листинге 7.3 приведен исходный HTML-код файла `index.html`.

Листинг 7.3. Исходный код файла `index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initialscale=1,
                                                        shrink-to-fit=no">
        <meta http-equiv="x-ua-compatible" content="ie=edge">
        <link rel="stylesheet" href="/css/bootstrap.min.css">
</head>

<body>
        <div class="container">
        <br>
        <div class="row m-t-1">
        <div class="col-md-12">
                <a href="/matches">Матчи</a>
```

```
</div>
</div>
<br>
<div class="row">
<div class="col-md-4">
<h3>Развернуть контракт</h3>

<form id="deploy">
<div class="form-group">
<label>Адрес: </label>
<input type="text" class="form-control" id="fromAddress">
</div>
<div class="form-group">
<label>Закрытый ключ: </label>
<input type="text" class="form-control" id="privateKey">
</div>
<div class="form-group">
<label>Номер матча: </label>
<input type="text" class="form-control" id="matchId">
</div>
<div class="form-group">
<label>Ставка (сумма эфира): </label>
<input type="text" class="form-control" id="betAmount">
</div>
<p id="message" style="word-wrap: break-word"></p>
<input type="submit" value="Deploy" class="btn btn-primary" />
</form>
</div>

<div class="col-md-4">
<h3>Ставка для контракта</h3>

<form id="bet">
<div class="form-group">
<label>Адрес: </label>
<input type="text" class="form-control" id="fromAddress">
</div>
<div class="form-group">
<label>Закрытый ключ: </label>
<input type="text" class="form-control" id="privateKey">
</div>
<div class="form-group">
<label>Адрес контракта: </label>
<input type="text" class="form-control" id="contractAddress">
</div>
```

```
        <div class="form-group">
        <label>Команда: </label>
        <select class="form-control" id="team">
        <option>Домашняя</option>
        <option>Гостевая</option>
        </select>
        </div>
        <p id="message" style="word-wrap: break-word"></p>
        <input type="submit" value="Bet" class="btn btnprimary"/>
        </form>
        </div>

        <div class="col-md-4">
        <h3>Показать контракт</h3>

        <form id="find">
        <div class="form-group">
        <label>Адрес контракта: </label>
        <input type="text" class="form-control" d="contractAddress">
        </div>
        <p id="message"></p>
        <input type="submit" value="Find" class="btn btnprimary"/>
        </form>
        </div>
        </div>
        </div>

<script type="text/javascript" src="/js/web3.min.js"></script>
<script type="text/javascript" src="/js/ethereumjstx.js"></script>
<script type="text/javascript" src="/js/main.js"></script>

</body>
</html>
```

Этот код тоже не требует пояснений. Теперь мы напишем код JavaScript, который, собственно, и выполняет действия по развертыванию контракта, внесению средств на контракт и отображению информации о контракте.

Исходный код JavaScript приведен в листинге 7.4. Поместите его в файл с именем `main.js`.

Листинг 7.4. Исходный код файла `main.js`

```
var bettingContractByteCode = "6060604...";

var bettingContractABI =
[{"constant":false,"inputs":[{"name":"team","type":"uint256"}],"name":"betOnTeam","
outputs":[],"payable":true,"type":"function"},{"constant":false,"inputs":[{"name":"
```

```
myid","type":"bytes32"},{"name":"result","type":"string"}],"name":"__callback","out
puts":[],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"my
id","type":"bytes32"},{"name":"result","type":"string"},{"name":"proof","type":"byt
es"}],"name":"__callback","outputs":[],"payable":false,"type":"function"},{"constan
t":true,"inputs":[],"name":"url","outputs":[{"name":"","type":"string"}],"payable":
false,"type":"function"},{"constant":true,"inputs":[],"name":"matchId","outputs":[{
"name":"","type":"string"}],"payable":false,"type":"function"},{"constant":true,"in
puts":[],"name":"amount","outputs":[{"name":"","type":"uint256"}],"payable":false,"
type":"function"},{"constant":true,"inputs":[],"name":"homeBet","outputs":[{"name":
"","type":"address"}],"payable":false,"type":"function"},{"constant":true,"inputs":
[],"name":"awayBet","outputs":[{"name":"","type":"address"}],"payable":false,"type"
:"function"},{"inputs":[{"name":"_matchId","type":"string"},{"name":"_amount","type
":"uint256"},{"name":"_url","type":"string"}],"payable":false,"type":"constructor"}
];

var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));

function getAJAXObject()
{
        var request;
        if (window.XMLHttpRequest) {
        request = new XMLHttpRequest();
        } else if (window.ActiveXObject) {
        try {
        request = new ActiveXObject("Msxml2.XMLHTTP");
        } catch (e) {
        try {
        request = new ActiveXObject("Microsoft.XMLHTTP");
        } catch (e) {}
        }
        }
        return request;
}

document.getElementById("deploy").addEventListener("submit",
function(e){e.preventDefault();

var fromAddress = document.querySelector("#deploy #fromAddress").value;
var privateKey = document.querySelector("#deploy #privateKey").value;
var matchId = document.querySelector("#deploy #matchId").value;
var betAmount = document.querySelector("#deploy #betAmount").value;
var url = "/getURL?matchId=" + matchId;
var request = getAJAXObject();

request.open("GET", url);
request.onreadystatechange = function() {
        if (request.readyState == 4) {
        if (request.status == 200) {
```

```
if(request.responseText != "Произошла ошибка")
{
        var queryURL = request.responseText;
        var contract = web3.eth.contract(bettingContractABI);
        var data = contract.new.getData(matchId,
        web3.toWei(betAmount, "ether"), queryURL, {
        data: bettingContractByteCode
        });
        var gasRequired = web3.eth.estimateGas({ data: "0x" + data
        });
        web3.eth.getTransactionCount(fromAddress, function(error, nonce){
        var rawTx = {
        gasPrice: web3.toHex(web3.eth.gasPrice),
        gasLimit: web3.toHex(gasRequired),
        from: fromAddress,
        nonce: web3.toHex(nonce),
        data: "0x" + data,
        };
        privateKey = EthJS.Util.toBuffer(privateKey, "hex");
        var tx = new EthJS.Tx(rawTx);
        tx.sign(privateKey);
        web3.eth.sendRawTransaction("0x" +
        tx.serialize().toString("hex"), function(err, hash) {
        if(!err)
        {document.querySelector("#deploy #message").
        innerHTML = "Хеш транзакции: " + hash + ".
        Идет майнинг транзакции...";

        var timer = window.setInterval(function(){
        web3.eth.getTransactionReceipt(hash, function(err, result){
        if(result){window.clearInterval(timer);
        document.querySelector("#deploy #message").innerHTML =
        "Хеш транзакции: " + hash + " и адрес контракта: " +
        result.contractAddress;}
        })
        }, 10000)
        }
        else
        {document.querySelector("#deploy #message").innerHTML = err;
        }
        });
        })
        }
}
}
};
```

```
        request.send(null);
        }, false)
        document.getElementById("bet").addEventListener("submit",
                                                    function(e){e.preventDefault();

var fromAddress = document.querySelector("#bet #fromAddress").value;
var privateKey = document.querySelector("#bet #privateKey").value;
var contractAddress = document.querySelector("#bet #contractAddress").value;
var team = document.querySelector("#bet #team").value;

        if(team == "Home")
        {
                team = 1;
        }
        else
        {
                team = 2;
        }
        var contract = web3.eth.contract(bettingContractABI).at(contractAddress);
        var amount = contract.amount();
        var data = contract.betOnTeam.getData(team);

        var gasRequired = contract.betOnTeam.estimateGas(team, {
        from: fromAddress,
        value: amount,
        to: contractAddress
        })

        web3.eth.getTransactionCount(fromAddress, function(error, nonce){
        var rawTx = {
        gasPrice: web3.toHex(web3.eth.gasPrice),
        gasLimit: web3.toHex(gasRequired),
        from: fromAddress,
        nonce: web3.toHex(nonce),
        data: data,
        to: contractAddress,
        value: web3.toHex(amount)
        };

        privateKey = EthJS.Util.toBuffer(privateKey, "hex");
        var tx = new EthJS.Tx(rawTx);
        tx.sign(privateKey);
        web3.eth.sendRawTransaction("0x" + tx.serialize().toString("hex"),
        function(err, hash) {
        if(!err)
```

```
        {
        document.querySelector("#bet #message").innerHTML = "Хеш транзакции: " +  hash;
        }
        else
        {
        document.querySelector("#bet #message").innerHTML = err;
        }
        })
        })
        }, false)
        document.getElementById("find").addEventListener("submit", function(e){
        e.preventDefault();
        var contractAddress = document.querySelector("#find
        #contractAddress").value;+

        var contract = web3.eth.contract(bettingContractABI).at(contractAddress);
        var matchId = contract.matchId();
        var amount = contract.amount();
        var homeAddress = contract.homeBet();
        var awayAddress = contract.awayBet();
        document.querySelector("#find #message").innerHTML = "Баланс контракта: "
        + web3.fromWei(web3.eth.getBalance(contractAddress), "эфира") + ",
        Номер матча: " + matchId + ", сумма ставки: " + web3.fromWei(amount,
        "эфира") + " ETH, " + homeAddress + " поставлено на домашнюю команду " +
        awayAddress + " поставлено на гостевую команду";
}, false)
```

Этот код работает следующим образом:

1. Мы сохраняем байт-код и ABI контракта в переменных: `bettingContractByteCode` и `bettingContractABI`.
2. Создаем экземпляр `web3`, который соединяется с нашим узлом сети `testnet`.
3. Функция `getAJAXObject` (кроссбраузерная функция) возвращает нам объект AJAX.
4. Далее мы подключаем слушатель события `submit` к первой форме, которая служит для развертывания контракта. В функции обратного вызова слушателя выполняем вызов по адресу `/getURL` путем передачи `matchId` для получения зашифрованной строки запроса. Затем генерируем необходимые данные для развертывания контракта и оцениваем необходимое количество газа при помощи функции `gasRequired`. Для вычисления количества газа мы используем метод `estimateGas()`, но вы также можете использовать метод `web3.eth.estimateGas()`. Они различаются лишь аргументами — в данном случае нет необходимости передавать блок данных транзакции. Имейте в виду — если вызов функции порождает исключение, то `gasRequired` возвратит лимит газа для блока. Затем мы вычисляем `nonce`. Для этого используем метод `getTransactionCount()`, а не про-

цедуру, которая была описана в *главе 6*. Далее мы создаем сырую транзакцию, подписываем и отправляем ее. Когда майнинг транзакции завершен, мы отображаем адрес контракта.

5. Подключаем слушатель события `submit` ко второй форме, которая служит для внесения суммы ставки. Здесь мы генерируем блок данных транзакции, вычисляем необходимое количество газа, создаем сырую транзакцию, подписываем и передаем ее. При вычислении количества газа мы передаем адрес отправителя, потому что от него зависит количество газа. Учитывайте, что количество газа, необходимое для вызова функции контракта, зависит от параметров `to`, `from` и `value`.
6. Наконец, подключаем слушатель события `submit` к третьей форме, которая отображает информацию о развернутом контракте.

Тестирование приложения

Мы завершили разработку приложения для ставок на результаты матчей, и пришло время его испытать. Прежде чем приступить к тестированию, убедитесь, что блокчейн подсети testnet полностью загружен и ждет новые блоки.

Используя сервис кошелька, который был разработан ранее (см. *главу 5*), создайте три счета. Добавьте по одному тестовому эфиру на каждый счет при помощи сайта **http://faucet.ropsten.be:3001/**[13].

Выполните команду `node app.js` в каталоге `Initial` и перейдите по адресу: **http://localhost:8080/matches**. Вы должны увидеть таблицу, аналогичную представленной на рис. 7.1.

Скопируйте номер любого матча. Допустим, это будет матч под номером 123945. Перейдите по адресу **http://localhost:8080**, и вы увидите три формы, приведенные на рис. 7.2.

Заполните поля первой формы, использовав первый счет для развертывания контракта, и нажмите кнопку **Deploy** (рис. 7.3).

Теперь сделаем ставку на домашнюю команду (home team) со второго счета и на гостевую команду (away team) с третьего счета, как показано на рис. 7.4.

Введите адрес контракта в третью форму и нажмите кнопку **Find** — вы увидите информацию о своем контракте (рис. 7.5).

Когда завершится майнинг обеих транзакций, еще раз нажмите на кнопку **Find** — вы увидите, что сведения о контракте изменились (рис. 7.6): баланс контракта обнулен, потому что два эфира перечислены на счет, с которого мы сделали ставку на победившую команду.

[13] Сервис ropsten.be позволял получить тестовые эфиры для работы в сети testnet, но весной 2017 года прекратил работу. Воспользуйтесь альтернативным источником — например: **https://www.rinkeby.io/#faucet**.

Рис. 7.1. Начальный экран приложения с таблицей матчей

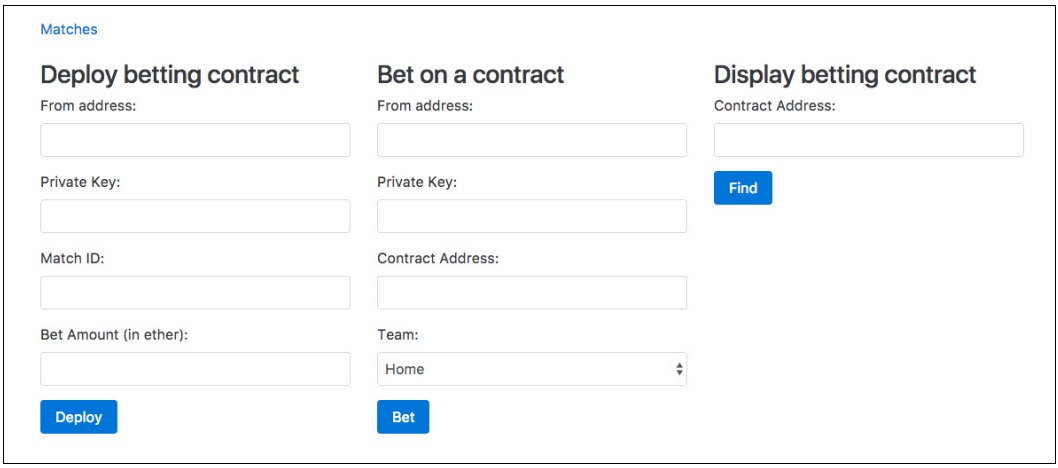

Рис. 7.2. Формы ввода для развертывания контракта и ставок

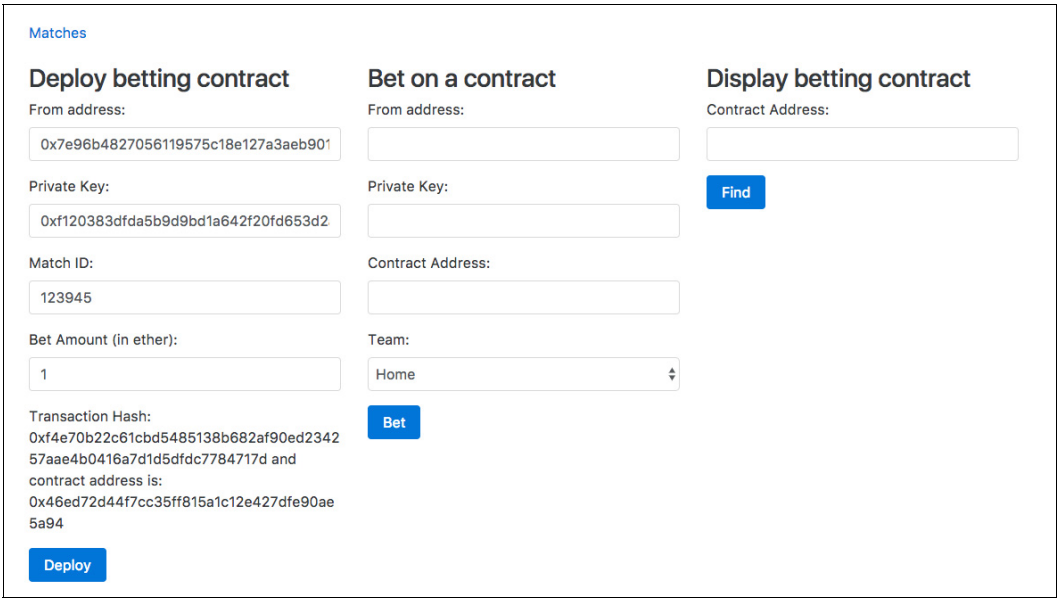

Рис. 7.3. Развертывание контракта

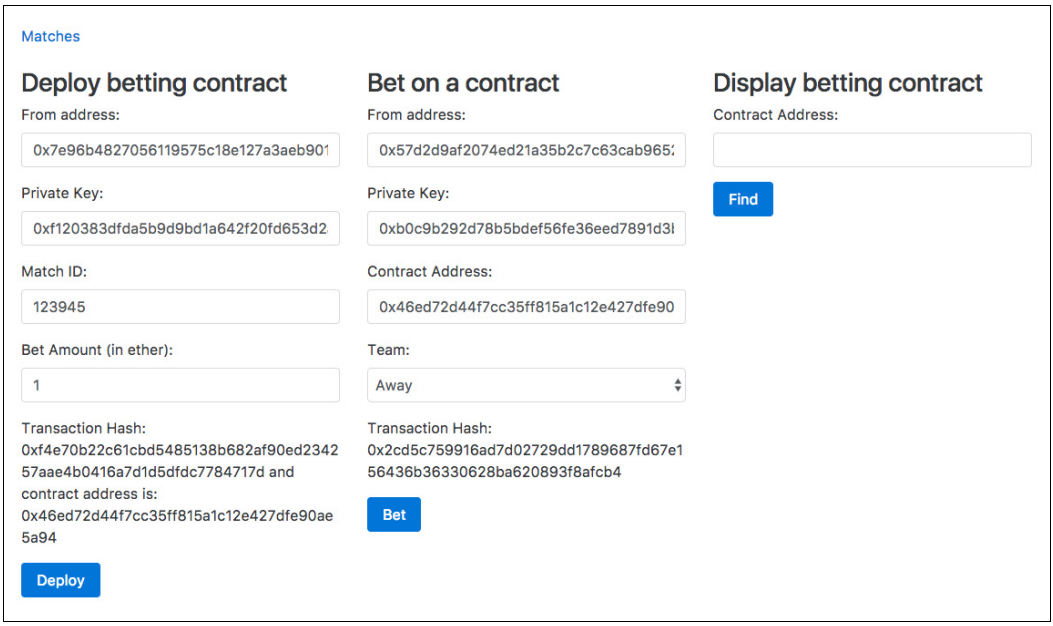

Рис. 7.4. Размещение ставок на команды

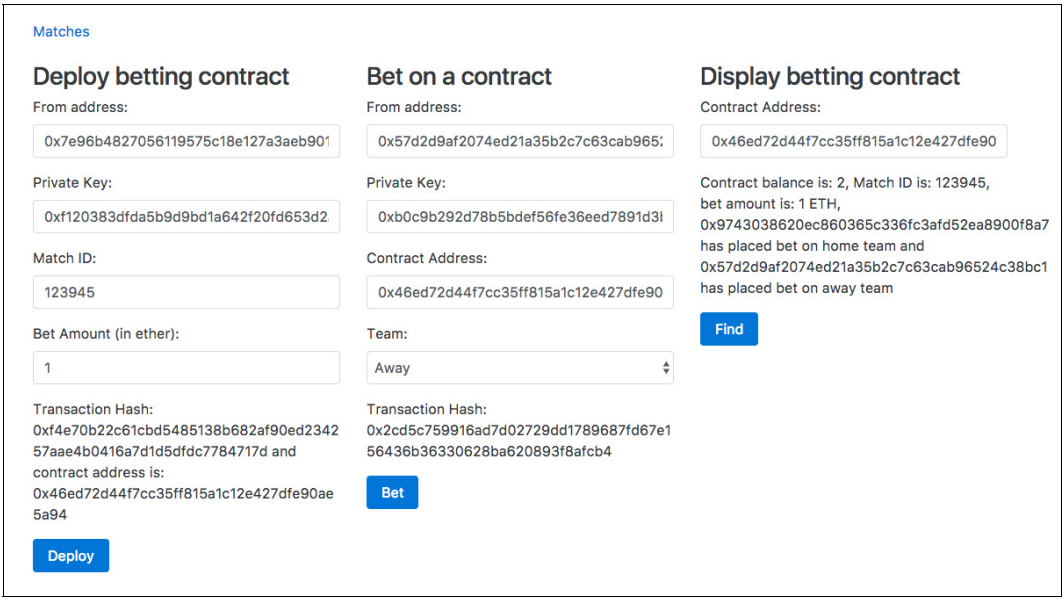

Рис. 7.5. Отображение информации о заданном контракте

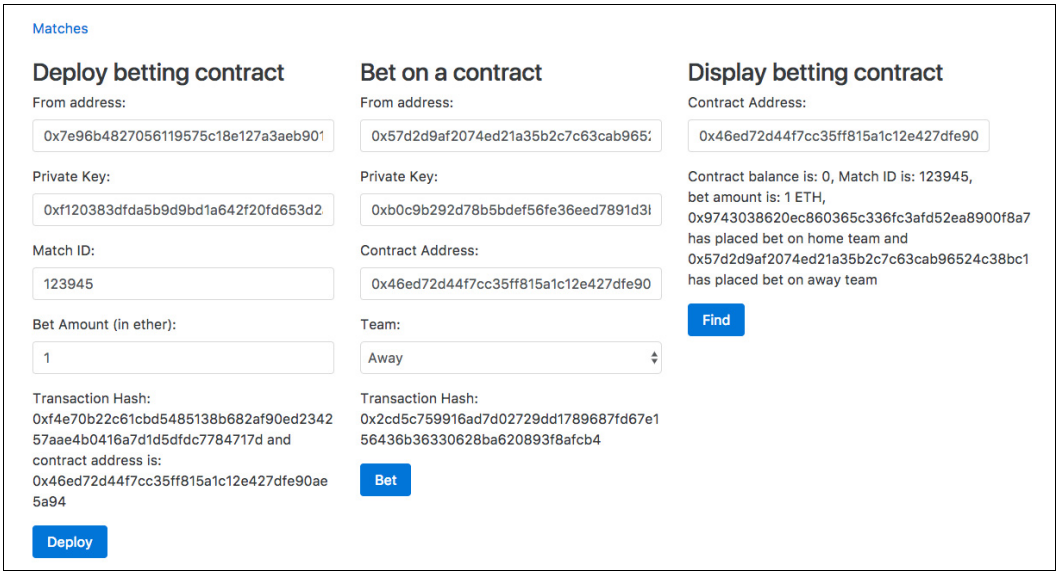

Рис. 7.6. Информация о контракте после выплаты выигрыша

Заключение

В этой главе мы детально изучили сервис Oraclize и библиотеку strings и использовали их при разработке децентрализованной платформы ставок. Теперь вы можете пойти дальше и самостоятельно доработать контракт и клиентскую часть в соответствии со своими потребностями. Чтобы расширить функции приложения, вы можете добавить в контракт события и отображать уведомления в интерфейсе клиентской части. Цель главы заключалась в том, чтобы понять базовую архитектуру децентрализованного приложения ставок.

В следующей главе мы узнаем, как создавать смарт-контракты Ethereum на уровне предприятия, используя фреймворк Truffle и собственную криптовалюту.

8

Разработка смарт-контрактов уровня предприятия

До сих пор для компиляции исходного кода контракта мы использовали браузер Solidity, а тестирование контрактов проводили при помощи web3.js. Можно было также применить и онлайновый инструмент Solidity IDE. Все это хорошо выглядит, пока мы компилируем одиночный смарт-контракт, который содержит лишь несколько импортов. Но как только понадобится скомпилировать большой и сложный контракт, вы столкнетесь с проблемами при компиляции и тестировании. Здесь мы рассмотрим фреймворк Truffle[1], который значительно упрощает разработку децентрализованных приложений корпоративного уровня путем создания *альткоинов*. Альткоинами принято называть все остальные валюты, кроме биткойна.

В этой главе будут раскрыты следующие темы:

- что такое узел `ethereumjs-testrpc` и как им пользоваться?
- что такое заголовки событий?
- работа с контрактами при помощи пакета `truffle-contract`;
- установка Truffle и изучение командной строки и файла конфигурации;
- компиляция, развертывание и тестирование кода Solidity при помощи Truffle;
- управление пакетами при помощи NPM и EthPM;
- использование консоли Truffle и написание внешних скриптов;
- разработка клиентской части приложения при помощи Truffle.

---

[1] См. **http://truffleframework.com/**.

Знакомство с *ethereumjs-testrpc*

`ethereumjs-testrpc` — это узел Ethereum на основе Node.js, предназначенный для разработки и тестирования. Он имитирует поведение полноценного узла и значительно ускоряет процесс разработки. Узел включает в себя все популярные функции RPC (такие, как события).

Программное обеспечение узла написано на языке JavaScript и распространяется в виде пакета NPM. На момент подготовки книги актуальной была версия 3.0.3, а для запуска узла требовалась версия Node.js 6.9.1.

Все данные в процессе работы узла хранятся в памяти. Следовательно, после перезагрузки узла он возвращается в исходное состояние.

Установка и использование *ethereumjs-testrpc*

Существуют три разных способа имитировать узел Ethereum при помощи `ethereumjs-testrpc`. Каждый из этих способов предназначен для определенной ситуации. Давайте их рассмотрим.

Приложение командной строки *testrpc*

Для имитации узла Ethereum можно использовать приложение командной строки `testrpc`. Чтобы установить это приложение, необходимо выполнить глобальную установку `ethereumjs-testrpc` при помощи команды:

```
npm install -g ethereumjs-testrpc
```

Для этой команды доступны следующие опции:

- **`-a`** или **`--accounts`** — количество счетов (аккаунтов), которые генерируются при запуске;
- **`-b`** или **`--blocktime`** — устанавливает время `blocktime` в секундах для автоматического майнинга. По умолчанию значение `0`, и автомайнинг выключен;
- **`-d`** или **`--deterministic`** — всякий раз, когда узел запущен, он будет генерировать 10 детерминированных адресов, т. е., когда вы используете эту опцию, каждый раз генерируется один и тот же набор адресов. Эта опция может использоваться для генерации детерминированных адресов на основе предопределенной мнемоники;
- **`-n`** или **`--secure`** — по умолчанию блокирует все счета. Если эта опция использована без опции **`--unlock`**, то HD-кошелек не будет создан;
- **`-m`** или **`--mnemonic`** — задает мнемоническую фразу для генерации начальных адресов;
- **`-p`** или **`--port`** — порт для подключения. По умолчанию 8545;
- **`-h`** или **`--hostname`** — имя хоста. Задает значение, возвращаемое методом `server.listen()` по умолчанию;

- **-s** or **--seed** — произвольные исходные данные (сид) для генерации мнемоники HD-кошелька (см. *разд. «Что такое HD-кошелек?» главы 5*);
- **-g** или **--gasPrice** — задает пользовательскую цену газа (по умолчанию 1). Если цена газа не объявлена при отправке транзакции на узел, то используется это значение;
- **-l** или **--gasLimit** — задает пользовательский лимит газа (по умолчанию `0x47E7C4`). Если лимит газа не объявлен при отправке транзакции на узел, то используется это значение;
- **-f** или **--fork** — ветвление от другого работающего узла Ethereum для определенного блока. Параметр должен содержать HTTP-адрес и порт другого клиента — например: `http://localhost:8545`. Опционально вы можете указать номер конкретного блока для ветвления при помощи символа `@` — например: `http://localhost:8545@1599200`;
- **--debug** — выводит коды операций виртуальной машины для отладки;
- **--account** — применяется для импорта счетов. Опция может использоваться в строке произвольное число раз, и каждый раз в ней передается пара из закрытого ключа и связанного с ним счета. Например:

  ```
  testrpc --account="privatekey,balance" [--account="privatekey,balance"] ...
  ```

  Если вы использовали эту опцию, то HD-кошелек не будет создан;
- **-u** или **--unlock** — разблокирует заданный счет. Может использоваться несколько раз — по числу счетов, которые надо разблокировать. Если используется в связке с опцией **--secure**, то блокировка указанных счетов будет проигнорирована:

  ```
  testrpc --secure --unlock "0x1234..." --unlock "0xabcd..."
  ```

  В этом примере будут заблокированы все счета, кроме указанных в опции **--unlock**. Вместо длинного адреса можно указать индексы счетов, которые надо разблокировать:

  ```
  testrpc --secure -u 0 -u 1
  ```

  Эта хитрость может быть использована для того, чтобы скрыть от посторонних глаз полные номера счетов. При использовании этой опции вместе с опцией **--fork** вы можете выполнять транзакции для любого адреса из блокчейна, что очень удобно при тестировании и динамическом анализе;
- **--networkId** — задает идентификатор сети, частью которой является ваш узел.

Закрытый ключ имеет длину 64 символа и вводится как шестнадцатеричная строка с префиксом `0x`. Баланс счета тоже должен быть представлен шестнадцатеричным числом с префиксом `0x` в номинале Wei.

Использование *ethereumjs-testrpc* в качестве провайдера *web3* или HTTP-сервера

Вы можете использовать `ethereumjs-testrpc` в качестве `web3`-провайдера следующим образом:

```
var TestRPC = require("ethereumjs-testrpc");
web3.setProvider(TestRPC.provider());
```

Вы можете использовать `ethereumjs-testrpc` в качестве HTTP-сервера следующим образом:

```
var TestRPC = require("ethereumjs-testrpc");
var server = TestRPC.server();
server.listen(port, function(err, blockchain) {});
```

Оба метода: `provider()` и `server()` — получают необязательный входной параметр, который определяет поведение `ethereumjs-testrpc`. Доступны следующие параметры:

- `accounts` — массив объектов. Каждый объект должен иметь значение баланса в виде шестнадцатеричного значения (`balanceKey`). Также можно указать закрытый ключ счета (`secretKey`). Если значение `secretKey` не задано, автоматически генерируется адрес, содержащий указанный баланс. Если закрытый ключ указан, то он используется для определения адреса счета;
- `debug` — вывод операционных кодов виртуальной машины для отладки;
- `logger` — значение является объектом, который реализует функцию `log()`;
- `mnemonic` — использует заданную мнемонику HD-кошелька для генерации начального адреса;
- `port` — порт входящих подключений при работе в режиме сервера;
- `seed` — произвольные данные для генерации мнемоники HD-кошелька;
- `total_accounts` — количество счетов (аккаунтов), которые надо сгенерировать при запуске;
- `fork` — то же самое, что опция командной строки **`--fork`** в предыдущем разделе;
- `network_id` — то же самое, что опция командной строки **`--networkId`** в предыдущем разделе;
- `time` — дата, с которой должен начинаться первый блок. Используйте этот параметр вместе с методом `evm_increaseTime` для тестирования функции `time-dependent`;
- `locked` — определяет, будут ли счета заблокированы по умолчанию;
- `unlocked_accounts` — массив адресов, или индексов адресов, которые не должны быть заблокированы.

Доступные методы RPC

Ознакомьтесь с полным перечнем методов RPC, которые реализованы в `ethereumjs-testrpc`:

- `eth_accounts;`
- `eth_blockNumber;`
- `eth_call;`
- `eth_coinbase;`
- `eth_compileSolidity;`
- `eth_estimateGas;`
- `eth_gasPrice;`
- `eth_getBalance;`
- `eth_getBlockByNumber;`
- `eth_getBlockByHash;`
- `eth_getBlockTransactionCountByHash;`
- `eth_getBlockTransactionCountByNumber;`
- `eth_getCode (only supports block number "latest");`
- `eth_getCompilers;`
- `eth_getFilterChanges;`
- `eth_getFilterLogs;`
- `eth_getLogs;`
- `eth_getStorageAt;`
- `eth_getTransactionByHash;`
- `eth_getTransactionByBlockHashAndIndex;`
- `eth_getTransactionByBlockNumberAndIndex;`
- `eth_getTransactionCount;`
- `eth_getTransactionReceipt;`
- `eth_hashrate;`
- `eth_mining;`
- `eth_newBlockFilter;`
- `eth_newFilter (includes log/event filters);`
- `eth_sendTransaction;`
- `eth_sendRawTransaction;`
- `eth_sign;`
- `eth_syncing;`
- `eth_uninstallFilter;`
- `net_listening;`

- net_peerCount;
- net_version;
- miner_start;
- miner_stop;
- rpc_modules;
- web3_clientVersion;
- web3_sha3.

Имеются также и нестандартные методы, которые не включены в стандартную спецификацию RPC:

- evm_snapshot — снимок состояния блокчейна на момент добавления текущего блока. Не получает параметры. Возвращает целочисленный идентификатор созданного снимка;
- evm_revert — возвращает блокчейн к состоянию предыдущего снимка. Получает единственный параметр — идентификатор снимка, к состоянию которого надо вернуть блокчейн. Если идентификатор не указан, происходит возврат к последнему снимку. В случае успеха возвращает true;
- evm_increaseTime — перескакивает вперед во времени. Получает единственный параметр — интервал времени в секундах. Возвращает текущую настройку времени в секундах;
- evm_mine — принудительно запускает майнинг блока. Не получает параметры. Выполняет майнинг блока независимо от того, запущен или остановлен майнинг.

Что такое заголовки событий?

Заголовки событий (topics) — это значения, применяемые для индексирования событий. Вы не можете искать события без заголовков. Когда возникает событие, для него генерируется заголовок по умолчанию, который считается первым заголовком события. У события может быть до четырех заголовков. Заголовки всегда формируются в одинаковом порядке. Вы можете искать событие, используя один или более заголовков.

Первый заголовок представляет собой сигнатуру события. Три остальных заголовка — это значения индексированных параметров. Если параметр типа string, bytes или array, то в заголовке представлен его хеш по keccak-256.

Рассмотрим пример, поясняющий, что такое заголовок. Допустим, событие имеет вид:

```
event ping(string indexed a, int indexed b, uint256 indexed c, string d, int e);
//вызов события
ping("Random String", 12, 23, "Random String", 45);
```

В данном случае будут сформированы четыре заголовка:

- `0xb62a11697c0f56e93f3957c088d492b505b9edd7fb6e7872a93b41cdb2020644` — это первый заголовок. Он сгенерирован методом:

  ```
  web3.sha3("ping(string,int256,uint256,string,int256)")
  ```

  Мы видим, что здесь все данные использованы в исходной канонической форме;

- `0x30ee7c926ebaf578d95b278d78bc0cde445887b0638870a26dcab901ba21d3f2` — это второй заголовок. Он сгенерирован методом:

  ```
  web3.sha3("Random String")
  ```

- третий и четвертый заголовки имеют вид:

  ```
  0x000000000000000000000000000000000000000000000000000000000000000c
  0x0000000000000000000000000000000000000000000000000000000000000017
  ```

  Это шестнадцатеричные представления параметров. Они вычислены при помощи методов:

  `EthJS.Util.bufferToHex(EthJS.Util.setLengthLeft(12,32))` и
  `EthJS.Util.bufferToHex(EthJS.Util.setLengthLeft(23,32))`.

Ваш узел Ethereum будет создавать индексы на основе заголовков, поэтому вы сможете легко находить события по их сигнатуре и значениям с индексами.

Допустим, вы хотите получать вызовы по заданному событию, у которого первый аргумент `"Random String"`, а третий аргумент принимает значения `23` или `78`. Вы можете найти событие, используя метод `web3.eth.getFilter()` следующим образом:

```
var filter = web3.eth.filter({
        fromBlock: 0,
        toBlock: "latest",
        address: "0x853cdcb4af7a6995808308b08bb78a74de1ef899",
        topics:
["0xb62a11697c0f56e93f3957c088d492b505b9edd7fb6e7872a93b41cdb2020644",
"0x30ee7c926ebaf578d95b278d78bc0cde445887b0638870a26dcab901ba21d3f2",
null,[EthJS.Util.bufferToHex(EthJS.Util.setLengthLeft(23, 32)),
EthJS.Util.bufferToHex(EthJS.Util.setLengthLeft(78, 32))]]
});
filter.get(function(error, result){
        if (!error)
                console.log(result);
});
```

В этом примере мы просим узел извлечь из блокчейна все события, которые порождены контрактом с адресом:

```
0x853cdcb4af7a6995808308b08bb78a74de1ef899,
```

у которых:

- первый заголовок:

  ```
  0xb62a11697c0f56e93f3957c088d492b505b9edd7fb6e7872a93b41cdb2020644;
  ```

- второй заголовок:

```
0x30ee7c926ebaf578d95b278d78bc0cde445887b0638870a26dcab901ba21d3f2;
```

- а третий заголовок:

```
0x0000000000000000000000000000000000000000000000000000000000000017
```

или

```
0x000000000000000000000000000000000000000000000000000000000000004e.
```

В приведенном примере обратите внимание на порядок следования значений в массиве `topics`. В данном случае важно соблюдать порядок следования значений.

Знакомство с пакетом *truffle-contract*

Прежде чем начать изучение фреймворка Truffle, нам следует ближе познакомиться с пакетом `truffle-contract`, который глубоко интегрирован в Truffle. Многие функции Truffle — такие как развертывание кода контракта, тестирование, взаимодействие между Truffle и контрактом — реализованы при помощи `truffle-contract`.

API `truffle-contract` значительно облегчает работу со смарт-контрактами. До сих пор для развертывания и вызова смарт-контрактов мы использовали web3.js, но `truffle-contract` помогает выполнить эту работу намного проще и быстрее. При работе со смарт-контрактами `truffle-contract` обладает рядом преимуществ по сравнению с web3.js:

- синхронизация транзакций для лучшего управления потоком выполнения (т. е. транзакции не будут завершены, пока нет гарантии, что завершен майнинг);
- API на основе JavaScript Promise[2] — попрощайтесь с кошмаром обратных вызовов;
- значения по умолчанию для транзакций, такие как `address` или `gas`;
- возвращает логи, подтверждение и хеш для каждой синхронизированной транзакции.

Прежде чем углубиться в изучение пакета `truffle-contract`, вам следует знать, что он не позволяет подписывать транзакции при помощи счетов, которые хранятся за пределами узла Ethereum. Это значит, что пакет не содержит ничего похожего на `sendRawTransaction`. API `truffle-contract` подразумевает, что каждый пользователь вашего децентрализованного приложения имеет работающий узел, и его счета расположены на этом узле. На самом деле, приложение так и должно работать, потому что если позволить каждому клиенту произвольно создавать счета и управлять ими, это будет удобно для пользователей, но станет изрядной головной болью для раз-

[2] См. **https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises**.

работчика. Разработчику придется разрабатывать менеджер кошелька каждый раз, под каждого нового клиента[3]. А теперь вопрос — как клиенты узнают, где пользователь хранит свои счета и какой у них формат? Из соображений переносимости рекомендуется полагать, что пользователи хранят счета на своем персональном узле и для управления счетами используют что-то аналогичное приложению Ethereum Wallet. Поскольку счета, хранящиеся на узле сети Ethereum, подписаны самим узлом, то нам не требуется ничего похожего на метод `sendRawTransaction`. Каждый пользователь должен иметь собственный узел и не может пустить на этот узел еще кого-то. Когда счет разблокирован, он полностью открыт для любого, кто пожелает им воспользоваться, и другие пользователи могут украсть средства и выполнить транзакции с чужого счета.

Если вы используете приложение, которое требует от вас поддерживать собственный узел и управлять счетами на этом узле, то убедитесь, что никто не может выполнить запрос JSON-RPC к этому узлу. Наоборот, такой запрос должен быть доступен только для локальных приложений. Также постарайтесь не держать слишком долго счет в разблокированном состоянии, и блокируйте счет сразу же, как только в нем отпала необходимость.

Если ваше приложение нуждается в функциях создания и подписания сырых транзакций, то вы можете использовать `truffle-contract` только для разработки и тестирования смарт-контрактов, а в своем приложении потом взаимодействовать с контрактами так, как было показано в предыдущих главах.

Установка и импорт *truffle-contract*

На момент подготовки книги актуальной была версия API `truffle-contract` 1.1.10[4]. Прежде, чем импортировать `truffle-contract`, сначала следует импортировать web3.js, т. к. вам придется создать провайдер для работы с API `truffle-contract`. В свою очередь, `truffle-contract` будет использовать провайдер для внутренних вызовов JSON-RPC.

Для установки `truffle-contract` при помощи менеджера пакетов NPM достаточно выполнить команду внутри каталога вашего приложения:

```
npm install truffle-contract
```

Затем в своем коде вы сможете использовать импорт:

```
var TruffleContract = require("truffle-contract");
```

Для использования `truffle-contract` в браузере вы можете скачать браузерный вариант из каталога **/dist** по адресу: **https://github.com/trufflesuite/truffle-contract**.

[3] Под клиентом подразумевается клиентская часть приложения, которая может быть создана сторонним разработчиком или модифицирована пользователем на основе открытого исходного кода.

[4] На момент подготовки перевода по адресу: **https://github.com/trufflesuite/truffle-contract** была доступна уже версия 3.0.1.

В HTML-коде вы можете воспользоваться следующим путем доступа:

```
<script type="text/javascript" src="./dist/trufflecontract.min.js"></script>
```

Теперь вам доступна глобальная переменная `TruffleContract`.

Настройка тестового окружения

Мы уже почти готовы приступить к изучению API `truffle-contract`, но сначала должны настроить тестовое окружение, которое поможет нам проверять код в процессе обучения.

Запустите узел `ethereumjs-testrpc`, представляющий сеть с идентификатором `10` при помощи команды: **`testrpc --networkId 10`**. Мы наугад выбрали этот идентификатор, поэтому вы можете указать любой другой номер. Только не указывайте номер `1`, потому что это подсеть mainnet, которая всегда используется только реально работающими приложениями и не предназначена для нужд тестирования и отладки.

Создайте файл и поместите в него следующий HTML-код:

```
<!doctype html>
<html>
        <body>
        <script type="text/javascript" src="./web3.min.js"></script>
        <script type="text/javascript" src="./trufflecontract.min.js">
        </script>
        <script type="text/javascript">
        //здесь поместите свой код
        </script>
        </body>
</html>
```

Скачайте файлы `web3.min.js` и `truffle-contract.min.js`. Вы можете найти их по адресу: **https://github.com/trufflesuite/truffle-contract/tree/master/dist**.

API *truffle-contract*

Пакет `truffle-contract` предоставляет нам два API: на уровне абстракции[5] контракта и на уровне экземпляра контракта:

- API абстракции предоставляет различные виды информации о контракте (или библиотеке), такие как ABI, несвязанный байт-код, развернут ли контракт, адрес контракта в различных сетях Ethereum, адреса библиотек в различных сетях Ethereum, события контракта. API на уровне абстракции — это набор функций, которые существуют для всех абстракций контракта;

[5] См. **https://en.wikipedia.org/wiki/Abstraction_layer**.

- экземпляр контракта представляет собой развернутый в определенной сети контракт. Соответственно, API на уровне экземпляра — это API, предоставляемые экземпляром контракта. Они создаются динамически, в зависимости от функций, доступных в исходном коде Solidity. Экземпляр контракта создается из абстракции контракта, представляющей этот контракт.

API абстракции контракта

API абстракции контракта — вот чем пакет `truffle-contract` значительно отличается от web3.js. Основные отличия заключаются в следующем:

- API автоматически получает значения по умолчанию — такие как адреса библиотек, адреса контрактов и другие, в зависимости от того, к какой сети подключен узел. Следовательно, вам не надо редактировать исходный код всякий раз, когда вы меняете сеть;
- вы можете, при желании, слушать только заданные события и только в заданной сети;
- API позволяет без проблем подключать библиотеки к байт-коду прямо во время выполнения.

Есть и другие выгоды от использования API уровня абстракции, о которых вы узнаете позже, во время изучения способов работы с API.

Прежде чем узнать, как создать абстракцию контракта и ее методы, давайте напишем простой контракт, который будет представлять абстракция. Исходный код контракта представлен в листинге 8.1.

Листинг 8.1. Исходный код контракта для изучения абстракции

```
pragma Solidity ^0.4.0;

import "github.com/pipermerriam/ethereum-string-utils/contracts/StringLib.sol";

//если этот путь не сработает, попробуйте полный вариант
//import "github.com/pipermerriam/ethereum-string-
utils/tree/master/contracts/StringLib.sol";

contract Sample
{
        using StringLib for *;
        event ping(string status);
        function Sample()
        {
                uint a = 23;
                bytes32 b = a.uintToBytes();
```

```
                bytes32 c = "12";
                uint d = c.bytesToUInt();

                ping("Conversion Done");
        }
}
```

Этот контракт конвертирует значение типа `uint` в значение типа `bytes32` и наоборот при помощи библиотеки `StringLib`. Библиотека `StringLib` доступна по адресу: `0xcca8353a18e7ab7b3d094ee1f9ddc91bdf2ca6a4` в сети mainnet, но при тестировании контракта в другой сети необходимо разместить ее самостоятельно. Прежде, чем мы пойдем дальше, скомпилируйте контракт в браузере Solidity, потому что нам нужны ABI и байт-код контракта.

Теперь давайте создадим абстракцию контракта, представляющую контракт `SampleContract` и библиотеку `StringLib`. Исходный код представлен в листинге 8.2. Поместите код в файл с расширением `html`.

Листинг 8.2. Исходный код примера абстракции контракта

```
var provider = new Web3.providers.HttpProvider("http://localhost:8545");
var web3 = new Web3(provider);

var SampleContract = TruffleContract({
abi:
[{"inputs":[],"payable":false,"type":"constructor"},{"anonymous":false,
"inputs":[{"indexed":false,"name":"status","type":"string"}],"name":"ping",
"type":"event"}],
unlinked_binary:
"6060604052341561000c57fe5b5b60006000600060006000601793508373__StringLib__6394e
8767d909160006040516020015260405182638ffffffff167c0100000000000000000000000000
0000000000000000000000000000000000028152600401808281526020019150506020604051
083038186803b151561008b57fe5b60325a03f4151561009857fe5b505050604051805190505
92507f313200000000000000000000000000000000000000000000000000000000000091508
16000191673__StringLib__6381a33a6f909160006040516020015260405182638ffffffff1
67c0100000000000000000000000000000000000000000000000000000000000002815260040180
826000191660001916815260200191505060206040518083038186803b151561014557fe5b6
0325a03f4151561015257fe5b505050604051805190509050 7f3adb191b3dee3c3ccbe8c657
275f608902f13e3a020028b12c0d825510439e56604051808060200182810382526 00f81526
02001807f436f6e76657273696f6e20446f6e650000000000000000000000000000000000081
525060200191505060405180910390a15b505050505b6033806101da6000396000f30060606
040525bfe00a165627a7a7230582056ebda5c1e4ba935e5ad61a271ce8d59c95e0e4bca4ad2
0e7f07d804801e95c60029",
networks: {
        1: {
        links: {
```

```
        "StringLib": "0xcca8353a18e7ab7b3d094ee1f9ddc91bdf2ca6a4"
        },
        events: {
        "0x3adb191b3dee3c3ccbe8c657275f608902f13e3a020028b12c0d825510439e56": {
        "anonymous": false,
        "inputs": [
                {
                "indexed": false,
                "name": "status",
                "type": "string"
                }
                ],
                "name": "ping",
                "type": "event"
        }
        }
},

        10: {
        events: {
        "0x3adb191b3dee3c3ccbe8c657275f608902f13e3a020028b12c0d825510439e56": {
        "anonymous": false,
        "inputs": [
                {
                "indexed": false,
                "name": "status",
                "type": "string"
                }
                ],
                "name": "ping",
                "type": "event"
        }
        }
        }
        },
        contract_name: "SampleContract",
});

SampleContract.setProvider(provider);
SampleContract.detectNetwork();

SampleContract.defaults({
from: web3.eth.accounts[0],
gas: "900000",
gasPrice: web3.eth.gasPrice,
})
```

```
var StringLib = TruffleContract({
abi:
[{"constant":true,"inputs":[{"name":"v","type":"bytes32"}],"name":"bytesToUInt",
"outputs":[{"name":"ret","type":"uint256"}],"payable":false,"type":
"function"},{"constant":true,"inputs":[{"name":"v","type":"uint256"}],"name":
"uintToBytes","outputs":[{"name":"ret","type":"bytes32"}],"payable":false,
"type":"function"}],
unlinked_binary:
"6060604052341561000c57fe5b5b6102178061001c6000396000f30060606040526000357c
0100000000000000000000000000000000000000000000000000000000900463ffffffff168
06381a33a6f1461004657806394e8767d14610076575bfe5b61006600480803560001916900
6020019091905050610100aa565b604051808281526020019150506040518090390f35b61008
c60048080359060200190919050506101405b604051808260001916600019168152602001
91505060405180910390f35b60006000600060006001028460001916141561006c5576100005
65b600090505b602081101561013557760ff81601f0360080260020a8560019004811515610
ed57fe5b041691506000821415610100ff57610135565b603082108061010e5750603982115b1
561011857610000565b5b600a830292506030820383019250 5b808060010191505061006ca56
5b8292505b5050919050565b60006000821415610173577f3000000000000000000000000000
0000000000000000000000000000000000000090506101e2565b5b60008211561016e1576101
008160019004811515610018e57fe5b0460010290507f0100000000000000000000000000000000
0000000000000000000000000000000006030600a848115615610 1c357fe5b06010260010281
179050600a828115615610 1d957fe5b049150610174565b5b8090505b9190505600a165627a7
a72305820d2897c98df4e1a3a71aefc5c486aed29c47c80cfe77e38328ef5f4cb5efcf2f100
29",

networks: {
        1: {
                address: "0xcca8353a18e7ab7b3d094ee1f9ddc91bdf2ca6a4"
                }
        },
        contract_name: "StringLib",
})

StringLib.setProvider(provider);
StringLib.detectNetwork();

StringLib.defaults({
        from: web3.eth.accounts[0],
        gas: "900000",
        gasPrice: web3.eth.gasPrice,
})
```

Код из листинга 8.2 работает следующим образом:

1. Сначала мы создаем провайдер. При помощи этого провайдера `truffle-contract` будет взаимодействовать с узлом.

2. Создаем абстракцию для контракта `SampleContract` при помощи функции `TruffleContract`. Функция получает объект, который содержит различную информацию о контракте. Этот объект мы будем называть *артефактом* (artifacts object[6]). Свойства `abi` и `unlinked_binary` являются обязательными, а остальные свойства объекта опциональные. Свойство `abi` указывает на ABI контракта, а свойство `unlinked_binary` — на несвязанный двоичный код контракта.
3. В коде есть свойство `networks`, которое указывает на различную информацию о контракте в разных сетях. В данном случае мы говорим о том, что у нас сеть номер `1` (`mainnet`), зависимость `StringLib` размещена по адресу: `0xcca8353a18e7ab7b3d094ee1f9ddc91bdf2ca6a4`, поэтому при развертывании контракта в сети `1` она будет подключена автоматически. В свойство `networks` вложено свойство `address`, говорящее о том, что контракт уже развернут в данной сети, и это адрес контракта. Там же мы видим свойство `events`, обозначающее события контракта, которые мы хотим перехватывать. Ключами `events` являются заголовки событий, а значениями — ABI событий.
4. Затем мы вызываем метод `setProvider()` объекта `SampleContract` и передаем ему новый экземпляр провайдера. Это нужно для того, чтобы `truffle-contract` мог общаться с узлом. API `truffle-contract` не позволяет установить глобальный провайдер. Вы должны установить провайдер для каждой абстракции контракта. Такой подход позволяет нам без труда подключаться и работать с несколькими сетями одновременно.
5. Потом мы вызываем метод `detectNetwork()` объекта `SampleContract`. Здесь мы определяем идентификатор сети, которую представляет текущая абстракция контракта. Иными словами, во всех операциях с абстракцией контракта используются значения, соответствующие подключенной сети. Этот метод определяет идентификатор сети, к которой подключен наш узел, и автоматически устанавливает его для контракта. Если вы хотите вручную задать идентификатор или изменить его в процессе выполнения, то можете использовать метод: `SampleContract.setNetwork(network_id)`. После смены идентификатора убедитесь, что провайдер ссылается на узел этой же сети, иначе `truffle-contract` не сможет использовать корректные ссылки, события и т. д.
6. Устанавливаем значения по умолчанию для транзакций контракта. Метод `defaults()` получает значения по умолчанию, но опционально может и установить их. Если метод вызван без аргументов, то он вернет текущие значения. Если в метод переданы аргументы, то будут установлены новые значения по умолчанию.
7. Повторяем указанные действия с библиотекой `StringLib`, чтобы создать для нее абстракцию контракта.

[6] Полужаргонный американизм «artifacts object» можно перевести по смыслу как «объект промежуточных значений» или «набор рабочих сущностей», но мы не будем загромождать текст и воспользуемся прямой «калькой»: «артефакты».

Создание экземпляра контракта

Экземпляр контракта представляет контракт, развернутый в определенной сети. Используя абстракцию контракта, мы должны создать экземпляр контракта. Существует три метода создания экземпляра контракта:

1. `SampleContract.new([arg1, arg2, ...], [tx params])` — эта функция получает параметры, которые требуются контракту, и развертывает новый экземпляр контракта в той сети, для использования которой настроена абстракция контракта. У функции есть необязательный последний аргумент, который можно использовать для передачи параметров транзакции, включая адрес `from`, лимит газа и цену газа. Функция возвращает «обещание» (promise), что когда майнинг транзакции по развертыванию контракта будет завершен, вновь созданный адрес будет ассоциирован с абстракцией контракта. Функция не вносит изменения в артефакт, который представляет абстракция контракта. Перед использованием функции убедитесь, что она может найти адреса библиотек, для которых байт-код зависит от используемой сети.
2. `SampleContract.at(address)` — эта функция создает новую абстракцию контракта, представляющую контракт по указанному в аргументе адресу. Функция возвращает «обусловленный»[7] объект (но не фактическое обещание для обратной совместимости). Этот объект превращается в экземпляр абстракции после проверки, что код действительно имеется в наличии по указанному адресу в используемой сети.
3. `SampleContract.deployed()` — эта функция похожа на `at()`, но адрес запрашивается из артефакта, и эта функция возвращает «обусловленный» объект, который превращается в экземпляр контракта после подтверждения того, что код контракта существует по данному адресу, и адрес находится именно в рабочей сети.
4. Давайте развернем `SampleContract` и получим экземпляр контракта. В сети номер `10` мы сначала должны применить метод `new()` для размещения библиотеки `StringLib`, а затем добавить адрес размещенной библиотеки в абстракцию `StringLib` и связать эту абстракцию с абстракцией контракта `SampleContract`. Далее, при помощи метода `new()` мы развертываем контракт и получаем экземпляр контракта. Но в сети номер `1` нам достаточно только развернуть `SampleContract` и получить его экземпляр, потому что библиотека `StringLib` уже размещена в этой сети. В листинге 8.3 приведен код, который выполняет эти действия.

Листинг 8.3. Развертывание контракта и получение его экземпляра

```
web3.version.getNetwork(function(err, network_id) {
if(network_id == 1)
```

[7] В английском тексте автор использовал непереводимое жаргонное слово «thenable», обозначающее объект, зависящий от чего-то, что должно случиться позже. В коде программы действия, осуществляемые после выполнения условия, обозначают ключевым словом `then` (тогда, в таком случае).

```
        {
                var SampleContract_Instance = null;
                SampleContract.new().then(function(instance){
                SampleContract.networks[SampleContract.network_id]
                ["address"] = instance.address;
                SampleContract_Instance = instance;
                })
        }
else if(network_id == 10)
        {
                var StringLib_Instance = null;
                var SampleContract_Instance = null;
                StringLib.new().then(function(instance){
                StringLib_Instance = instance;
                }).then(function(){
                StringLib.networks[StringLib.network_id] = {};
                StringLib.networks[StringLib.network_id]["address"] =
                StringLib_Instance.address;
                SampleContract.link(StringLib);
                }).then(function(result){
                return SampleContract.new();
                }).then(function(instance){
                SampleContract.networks[SampleContract.network_id]
                ["address"] = instance.address;
                SampleContract_Instance = instance;
                })
        }
});
```

Код из листинга 8.3 работает следующим образом:

1. Сначала мы определяем идентификатор сети. Если это сеть номер `10`, то развертываем контракт и библиотеку, а если сеть номер `1`, то развертываем только контракт.
2. В сети номер `10` развертываем контракт библиотеки[8] `StringLib` и получаем экземпляр контракта.
3. Обновляем абстракцию `StringLib`, поскольку известно, какой адрес контракта в текущей сети она представляет. Если вы подключены к сети номер `1`, то будет просто переписан адрес `StringLib`, который уже задан.
4. Затем мы связываем `StringLib` с абстракцией `SampleContract`. Связывание обновляет ссылки и копирует события библиотеки в абстракцию `SampleContract` для

[8] Напомним, что библиотека — это, по сути, тоже контракт.

текущей сети. Библиотеки могут быть подключены несколько раз, при этом будут переписаны их предыдущие ссылки.

5. Развертываем `SampleContract` в текущей сети.
6. Обновляем абстракцию `SampleContract`, чтобы сохранить адрес контракта в текущей сети. Благодаря этому, позднее мы сможем использовать метод `deployed()`, чтобы получить экземпляр.
7. В случае подключения к сети номер `1` мы просто развертываем контракт `SampleContract`, и этого достаточно.
8. Теперь мы можем просто сменить сеть, к которой подключен узел, и перезагрузить приложение. Ваше приложение будет вести себя надлежащим образом. Например, на компьютере разработчика приложение будет подключено к тестовой сети, а на рабочем сервере — к сети mainnet. Очевидно, что вы не захотите развертывать контракты каждый раз, когда запускается файл приложения, поэтому вы можете фактически обновлять артефакты, когда контракты развернуты, а в коде приложения просто проверить, развернут контракт или нет. Если контракт не развернут, его следует развернуть. Вместо того чтобы обновлять артефакты вручную, вы можете сохранить их в базе данных или в файле и написать код для автоматического обновления после того, как завершено развертывание контракта.

API экземпляра контракта

Каждый экземпляр контракта отличается от исходного кода Solidity, и API создается динамически. Нам доступны следующие функции API экземпляра контракта:

- `allEvents` — это функция экземпляра контракта, которая принимает обратный вызов, возникающий всякий раз, когда событие инициируется контрактом, соответствующим сигнатуре события в текущей сети. Вы также можете использовать зависящую от имени функцию `event()` для перехвата определенного события, а не получать их все. Например, в предыдущем контракте (см. листинг 8.2) для перехвата события «пинг контракта» вы можете применить команду: `SampleContract_Instance.ping(function(e, r){});`
- `send` — эта функция применяется для отправки эфира в контракт. Она получает два аргумента: первый аргумент — сумма перевода в Wei, второй аргумент необязательный и может содержать адрес `from`, показывающий, с какого адреса выполнен перевод. Функция возвращает «обещание» (promise), которое выполняется в виде сведений о транзакции, когда завершен майнинг.

Мы можем вызвать любой метод контракта при помощи функций `SampleContract.functionName()` или `SampleContract.functionName.call()`. Первая функция отправляет транзакцию, а вторая лишь вызывает метод на виртуальной машине Ethereum и не сохраняет изменения в блокчейне. Оба вызова возвращают «обещание». В первом случае обещание выполняется в виде результата транзакции — т. е. в виде объекта, содержащего хеш транзакции, логи и квитанцию транзакции. Во

втором случае обещание реализуется в виде значения, возвращенного методом `call()`. Оба метода получают обязательные аргументы — ссылки на методы контракта, и необязательный последний аргумент — объект, который содержит параметры транзакции (`from`, `gas`, `value` и т. д.).

Введение в Truffle

Truffle — это *среда разработки* (инструмент командной строки для компиляции, развертывания, тестирования и сборки), *фреймворк* (набор пакетов для простого написания тестов, развертывания кода, сборки клиентов и т. д.), а также *канал обмена ресурсами* (публикация своих пакетов и использование пакетов, опубликованных другими). Truffle предназначен для разработки децентрализованных приложений на платформе Ethereum.

Установка Truffle

Truffle работает в среде OS X, Linux и Windows и требует предварительно установленную платформу Node.js версии 5.0 или выше. На момент подготовки этой книги последней стабильной версией Truffle была версия 3.1.2, и мы будем использовать эту версию[9].

Для установки Truffle при помощи менеджера пакетов введите команду:

```
npm install -g truffle
```

Прежде, чем двигаться дальше, убедитесь, что запустили узел `ethereumjs-testrpc` в сети номер `10` (см. *разд. «Настройка тестового окружения»*).

Инициализация Truffle

Начнем с создания каталога для нашего приложения и назовем этот каталог `altcoin`. Внутри каталога `altcoin` выполните команду инициализации вашего проекта:

```
truffle init
```

После завершения инициализации вы получите структуру проекта, состоящую из следующих компонентов:

- `contracts` — каталог, в котором Truffle ожидает найти контракты на языке Solidity;
- `migrations` — каталог для хранения файлов, содержащих код развертывания контрактов;
- `test` — место для тестовых файлов, применяемых при тестировании ваших смарт-контрактов;
- `truffle.js` — главный файл конфигурации Truffle;

[9] На момент подготовки перевода была доступна версия 4.0.1 по адресу: **https://github.com/trufflesuite/truffle**.

По умолчанию команда формирует набор демонстрационных контрактов (`MetaCoin` и `ConvertLib`), которые предлагают нам простую альтернативную валюту (альткоин) на платформе Ethereum.

В листинге 8.4 для ознакомления приведен исходный код смарт-контракта `MetaCoin`.

Листинг 8.4. Исходный код контракта `MetaCoin`

```
pragma Solidity ^0.4.4;

import "./ConvertLib.sol";

contract MetaCoin {
        mapping (address => uint) balances;
        event Transfer(address indexed _from, address indexed _to, uint256 _value);

        function MetaCoin() {
                balances[tx.origin] = 10000;
        }

        function sendCoin(address receiver, uint amount) returns(bool sufficient) {
                if (balances[msg.sender] < amount) return false;
                balances[msg.sender] -= amount;
                balances[receiver] += amount;
                Transfer(msg.sender, receiver, amount);
                return true;
        }

        function getBalanceInEth(address addr) returns(uint){
                return ConvertLib.convert(getBalance(addr),2);
        }

        function getBalance(address addr) returns(uint) {
                return balances[addr];
        }
}
```

Смарт-контракт `MetaCoin` помещает 10 000 метакоинов на счет по адресу, на котором развернут контракт. Теперь пользователь может отправить эти метакоины кому-нибудь при помощи функции `sendCoin()`. В любой момент вы можете узнать текущий баланс при помощи функции `getBalance()`. Можно получить баланс в эфирах при помощи функции `getBalanceInEth()`. При этом считается, что курс метакоина фиксированный и равен двум эфирам.

Библиотека `ConvertLib` применяется для конвертации суммы метакоинов в эфир. Для этой цели доступен метод `convert()`.

Компиляция контрактов

Компиляция контрактов завершается созданием артефактов, содержащих набор из `abi` и `unlinked_binary`. Для компиляции введите команду:

```
truffle compile
```

Truffle будет компилировать только те контракты, которые были изменены с момента последней компиляции. Если вы хотите изменить такое поведение, добавьте к предыдущей команде опцию: **`--all`**.

После компиляции вы найдете артефакты в каталоге `build/contracts`. Вы вправе редактировать эти файлы по своему усмотрению. Файлы модифицируются во время выполнения команд **`compile`** и **`migrate`**.

Есть несколько нюансов, на которые следует обратить внимание перед компиляцией:

- Truffle ожидает, что имя файла совпадает с именем контракта. Например, если у вас есть файл с именем `MyContract.sol`, то он должен содержать контракт `MyContract{}` или библиотеку `myContract{}`;
- имена файлов чувствительны к регистру. Например, если имя файла не содержит заглавные буквы, то и имя контракта не может содержать заглавных букв;
- вы можете объявить зависимости контракта при помощи команды **`import`**. Truffle будет компилировать контракты в правильном порядке и подключать библиотеки по мере необходимости. Зависимости должны быть объявлены относительно текущего расположения файла Solidity и начинаться с пути `./` или `../`.

Truffle версии 3.1.2 использует компилятор версии 0.4.8. В настоящее время Truffle не поддерживает изменение версии компилятора.

Файлы конфигурации

Файл `truffle.js` предназначен для конфигурирования проекта. Этот файл может содержать любой код, необходимый для создания конфигурации. Он должен экспортировать объект, представляющий конфигурацию вашего проекта. По умолчанию файл содержит следующий исходный код:

```
module.exports = {
       networks: {
              development: {
                     host: "localhost",
                     port: 8545,
                     network_id: "*" // Match any network id
              }
       }
};
```

Объект может содержать различные свойства. Но наиболее востребованным является свойство `networks`. Оно определяет как сеть, доступную для развертывания

контракта, так и специальные параметры транзакции (`gasPrice`, `from`, `gas` и т. д.) По умолчанию `gasPrice` = `100 000 000 000` Wei, `gas` = `4712388`, адрес `from` определяется по адресу первого доступного контракта в клиенте.

Вы можете описать свойства любого количества сетей. Отредактируйте файл конфигурации следующим образом:

```
module.exports = {
       networks: {
              development: {
                     host: "localhost",
                     port: 8545,
                     network_id: "10"
              },
              live: {
                     host: "localhost",
                     port: 8545,
                     network_id: "1"
              }
       }
};
```

В этом примере кода мы определили две сети с именами: `development` и `live`.

При использовании стандартной командной строки Windows имя файла конфигурации, заданное по умолчанию, может вызвать конфликт с исполняемым файлом Truffle. В таком случае рекомендуем использовать Windows PowerShell или Git BASH, т. к. в этих оболочках конфликт не возникает. В качестве альтернативного варианта можно изменить имя файла конфигурации на `truffle-config.js`.

Развертывание контрактов

Даже самый маленький проект будет взаимодействовать, по крайней мере, с двумя блокчейнами: один — на машине разработчика, такой как TestRPC EthereumJS, и другой в сети, в которой будет развернуто приложение. Это может быть, например, главная сеть Ethereum или частная корпоративная сеть.

Поскольку рабочая сеть определяется автоматически абстракциями контракта во время выполнения, это означает, что вам достаточно один раз развернуть контракт или клиентскую часть. Когда приложение запущено, работающий клиент сети Ethereum будет определять, какой артефакт контракта используется, и это придаст вашему приложению большую гибкость.

Файлы JavaScript, которые содержат код для развертывания контракта в сети Ethereum, называются *файлами переноса* (migration files). Эти файлы отвечают за поэтапную постановку задач развертывания и пишутся исходя из предположения, что ваши требования к развертыванию будут меняться со временем. По мере развития вашего проекта вы будете создавать новые скрипты переноса и вносить изме-

нения в блокчейн. История всех запусков переноса записывается в блокчейн при помощи специального контракта `Migrations`. Если вы просматривали содержимое каталогов `contracts` и `build/contracts`, то должны были заметить, что там уже есть контракт `Migrations`. Этот контракт всегда должен там находиться, и его не следует трогать без необходимости и ясного понимания своих действий.

Файлы переноса

В каталоге `migrations` вы можете видеть, что имена файлов начинаются с числового префикса — например, `1_initial_migration.js` и `2_deploy_contracts.js`. Числовой префикс необходим, чтобы контролировать порядок выполнения переноса.

Контракт `Migrations` сохраняет в `last_completed_migration` номер, соответствующий последнему использованному скрипту переноса из каталога `migrations`. Контракт `Migrations` всегда выполняется первым. Для имен файлов принято соглашение вида `x_script_name.js`, где значения `x` начинаются с `1`. Контракты вашего приложения обычно идут в скриптах, начиная с номера `2`.

Поскольку контракт `Migrations` хранит номер последнего выполненного скрипта, Truffle не может выполнить этот скрипт повторно. С другой стороны, в будущем ваше приложение может нуждаться в доработке или развертывании нового контракта. В таком случае вы можете создать новый скрипт со следующим порядковым номером. После выполнения этот скрипт тоже станет недоступным для повторного запуска.

Написание кода переноса

В начале файла переноса мы сообщаем Truffle, какой контракт будет взаимодействовать с ним через метод `artifacts.require()`. Этот метод совпадает с методом `require()` Node.js, но в нашем случае он особым образом возвращает абстракцию контракта, которую мы будем использовать далее, вплоть до окончания скрипта.

Все скрипты миграции должны экспортировать (предоставлять нам для использования) функцию через `module.exports`. Функция должна получать объект `deployer` в качестве первого аргумента. Этот объект помогает нам в развертывании и предоставляет простой и понятный API для развертывания смарт-контрактов, а также выполняет обычную работу — сохранение артефактов в файлах для последующего использования, связывание библиотек и т. д. Объект `deployer` — это ваш основной интерфейс для постановки задач развертывания.

Объект `deployer` предоставляет нам несколько методов. Все методы синхронные:

- `deployer.deploy(contractAbstraction, args..., options)` — развертывает контракт, определяемый объектом абстракции контракта с необязательными аргументами конструктора. Это удобно для одиночных контрактов, когда для вашего децентрализованного приложения существует только один экземпляр этого контракта. Метод определит адрес контракта после развертывания (т. е. свойство `address` в файле артефакта получит значение нового адреса), и будут переписаны любые

предыдущие значения адреса. Вы можете передать в метод массив контрактов (или массив массивов), чтобы ускорить развертывание контрактов. Дополнительно, последний аргумент является необязательным объектом, состоящим из единственного ключа `overwrite`. Если ключ `overwrite` установлен в состояние `false`, то метод не станет развертывать контракт заново, если он уже был развернут. Метод возвращает обещание (подтверждаемое ожидание выполнения);

- `deployer.link(library, destinations)` — подключает ранее размещенную библиотеку к одному или нескольким контрактам. Аргумент `destinations` может быть одиночной абстракцией контракта или массивом абстракций нескольких контрактов. Если контракт не ссылается на подключаемую библиотеку, `deployer` игнорирует этот контракт. Метод возвращает обещание;
- `deployer.then(function(){})` — метод применяется для запуска произвольного шага развертывания. В процессе выполнения скрипта переноса вы можете вызвать определенные функции контракта для добавления, редактирования или реорганизации данных контракта. Для развертывания и подключения контрактов внутри функции обратного вызова вы должны использовать API абстракции контракта.

Существует возможность выполнять условные шаги развертывания, которые зависят от сети назначения. Чтобы организовать условие выполнения шагов развертывания, ваш скрипт переноса должен получать второй параметр — `network`. Например, в сети mainnet уже развернуто множество популярных библиотек. Следовательно, при развертывании контракта в этой библиотеке вам не надо развертывать библиотеки — достаточно лишь подключить их. Вот небольшой пример такого подхода:

```
module.exports = function(deployer, network) {
if (network != "live") {
        // Выполняем различные действия, если имя сети не "live"
        } else {
        // Делаем что-то специальное для сети с именем "live"
        }
}
```

В заготовке проекта вы найдете два файла переноса: `1_initial_migration.js` и `2_deploy_contracts.js`. Первый файл не следует трогать, если вы не имеете четкого понимания, что делаете. Второй файл можно редактировать по своему усмотрению. Код файла `2_deploy_contracts.js` выглядит так:

```
var ConvertLib = artifacts.require("./ConvertLib.sol");
var MetaCoin = artifacts.require("./MetaCoin.sol");

module.exports = function(deployer) {
        deployer.deploy(ConvertLib);
        deployer.link(ConvertLib, MetaCoin);
        deployer.deploy(MetaCoin);
};
```

В этом коде мы создаем абстракции для библиотеки `ConvertLib` и контракта `MetaCoin`. Независимо от того, какую сеть будем использовать, мы размещаем библиотеку `ConvertLib`, подключаем ее к контракту `MetaCoin` и, наконец, развертываем контракт `MetaCoin`.

Для запуска скрипта переноса выполните команду:

```
truffle migrate --network development
```

В данном случае мы указываем Truffle выполнить перенос в сеть `development`. Впрочем, если не указать опцию **`--network`**, то сеть `development` будет использована по умолчанию.

После запуска команды вы получите уведомление о том, что Truffle автоматически обновит адреса библиотеки `ConvertLib` и контракта `MetaCoin` в файлах артефактов, а также обновит ссылки.

Есть еще две важные опции, на которые стоит обратить внимание:

- **`--reset`** — выполняет все скрипты переноса с самого начала, а не с последнего завершенного переноса;
- **`-f <number>`** — выполняет скрипт, начиная с указанного номера.

Вы можете в любое время найти адреса контрактов и библиотек вашего проекта при помощи команды **`truffle networks`**.

Юнит-тесты контрактов

Юнит-тест (поузловой тест) — один из методов тестирования приложения. Это процесс, при котором приложение разбивается на минимально возможные функциональные фрагменты, именуемые *юнитами* (unit), а затем каждый юнит подвергается тестированию по отдельности. Юнит-тестирование можно выполнять вручную, хотя обычно его автоматизируют.

Truffle по умолчанию распространяется с фреймворком для автоматизации тестирования ваших контрактов. Он обеспечивает чистую среду выполнения для запуска тестовых файлов — Truffle будет перезапускать ваши скрипты переноса для каждого тестового файла, чтобы поддерживать свежий набор контрактов, подвергаемых тестированию.

Truffle позволяет нам писать простые и управляемые тесты двумя способами:

- на языке JavaScript — чтобы проверять контракт из клиентской части приложения;
- на языке Solidity — чтобы проверять ваш контракт из других контрактов.

Оба подхода имеют свои достоинства и недостатки, поэтому мы изучим два способа разработки тестов.

Все тесты должны располагаться в каталоге `/test`. Truffle будет выполнять только файлы с расширениями `js`, `es`, `es6`, `jsx` и `sol`. Все остальные файлы игнорируются.

При запуске автоматических тестов `ethereumjs-testrpc` работает заметно быстрее, чем другие клиенты. Более того, `testrpc` содержит специальные опции, которые повышают скорость выполнения тестов Truffle почти на 90 процентов. В общем, мы рекомендуем использовать `testrpc` в процессе обычной разработки и тестирования, а потом непосредственно перед развертыванием еще раз однократно выполнить тесты при помощи `go-ethereum` или иного официального клиента Ethereum.

Написание тестов на JavaScript

Тестовый фреймворк Truffle является надстройкой над Mocha. В свою очередь, Mocha — это среда JavaScript для написания тестов, а Chai — библиотека проверки утверждений (assertion library).

Тестовые фреймворки применяются для организации и выполнения тестов, а библиотеки проверки утверждений содержат утилиты для проверки правильности различных утверждений. Библиотеки значительно упрощают тестирование вашего кода, потому что вам не приходится выполнять тысячи проверок оператором `if`. Большинство тестовых фреймворков не содержат библиотек проверки утверждений и разрешают пользователю подключать их по мере необходимости.

Прежде чем продолжить чтение, вам следует научиться писать тесты с использованием Mocha и Chai. Для изучения Mocha посетите адрес: **https://mochajs.org/**. Для изучения Chai посетите адрес: **http://chaijs.com/**.

Ваши тесты должны располагаться в каталоге `./test` и названия файлов должны заканчиваться расширением `js`.

Абстракции контракта являются основой возможного взаимодействия с JavaScript. Поскольку у Truffle нет способа заранее определить, для каких контрактов вам потребуется взаимодействие с тестами, вам придется запросить контракты в явном виде. Это делается при помощи метода `artifacts.require()`. Итак, первое, что нужно сделать в тестовых файлах, — это создать абстракции для контрактов, которые вы хотите проверить.

Затем следует написать тесты. Ваши тесты должны перейти из Mocha почти без структурных изменений. Файлы тестов должны содержать код, который Mocha распознает как автоматизированный тест. Тесты Truffle отличаются от Mocha наличием функции `contract()`. Эта функция в целом работает аналогично `describe()`, за исключением того, что указывает Truffle выполнить все скрипты переноса. Функция `contract()` работает следующим образом:

- перед выполнением каждой функции `contract()` ваши контракты заново разворачиваются на узле Ethereum, поэтому все тесты запускаются для исходного состояния контракта;
- функция `contract()` предоставляет перечень счетов, доступных на вашем узле и пригодных для написания тестов.

Поскольку «под капотом» у Truffle скрывается Mocha, вы можете продолжать использовать `describe()` для запуска обычных тестов Mocha, когда не нужны функции Truffle.

В листинге 8.5 приведен исходный код теста, который по умолчанию генерирует Truffle для тестирования контракта `MetaCoin`. Вы найдете этот код в файле `metacoin.js`.

Листинг 8.5. Исходный код файла теста `metacoin.js`

```
// Запрашиваем абстракцию для MetaCoin.sol
var MetaCoin = artifacts.require("./MetaCoin.sol");

contract('MetaCoin', function(accounts) {
        it("на первом счете должно быть 10000 MetaCoin", function() {
        return MetaCoin.deployed().then(function(instance) {
        return instance.getBalance.call(accounts[0]);
        }).then(function(balance) {
        assert.equal(balance.valueOf(), 10000, "10000 не оказалось на первом счете");
});
});
it("следует корректно отправить средства", function() {
        var meta;

        // Получаем начальные балансы первого и второго счета.
        var account_one = accounts[0];
        var account_two = accounts[1];
        var account_one_starting_balance;
        var account_two_starting_balance;
        var account_one_ending_balance;
        var account_two_ending_balance;

        var amount = 10;

        return MetaCoin.deployed().then(function(instance) {
        meta = instance;
        return meta.getBalance.call(account_one);
        }).then(function(balance) {
        account_one_starting_balance = balance.toNumber();
        return meta.getBalance.call(account_two);
        }).then(function(balance) {
        account_two_starting_balance = balance.toNumber();
        return meta.sendCoin(account_two, amount, {from: account_one});
        }).then(function() {
        return meta.getBalance.call(account_one);
        }).then(function(balance) {
        account_one_ending_balance = balance.toNumber();
```

```
        return meta.getBalance.call(account_two);
        }).then(function(balance) {
        account_two_ending_balance = balance.toNumber();

        assert.equal(account_one_ending_balance, account_one_starting_balance - amount,
                                        "Сумма некорректно получена от отправителя");
        assert.equal(account_two_ending_balance, account_two_starting_balance +
                                amount, "Сумма некорректно отправлена получателю");
        });
        });
});
```

В коде листинга 8.5 вы можете видеть, что взаимодействие контрактов полностью реализовано на основе библиотеки `truffle-contract`. В остальном код очевиден и не требует дополнительных пояснений.

Наконец, Truffle дает нам доступ к конфигурации Mocha, поэтому мы можем управлять поведением Mocha. Конфигурация Mocha находится в свойстве `mocha` в файле `truffle.js` экспортированного объекта. Например, конфигурация может выглядеть так:

```
mocha: {
        useColors: true
}
```

Написание тестов на Solidity

Тестовые файлы на языке Solidity имеют расширение `sol`. Перед тем как приступить к написанию тестов на языке Solidity, обратите внимание на некоторые особенности:

- тесты Solidity не должны являться расширением какого-либо контракта. Это минимизирует ваши тесты насколько возможно и дает вам полный контроль над контрактами, которые вы пишете;
- Truffle содержит готовую библиотеку проверки утверждений, но вы можете изменять и дорабатывать эту библиотеку по мере необходимости;
- вы должны иметь возможность запустить ваши тесты Solidity в любом клиенте Ethereum.

Чтобы научиться писать тесты на Solidity, рассмотрите исходный код стандартного теста, который по умолчанию генерирует Truffle. Код хранится в файле `TestMetacoin.sol` и показан в листинге 8.6.

Листинг 8.6. Исходный код файла теста `TestMetacoin.sol`

```
pragma Solidity ^0.4.2;

import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/MetaCoin.sol";
```

```
contract TestMetacoin {
        function testInitialBalanceUsingDeployedContract() {
        MetaCoin meta = MetaCoin(DeployedAddresses.MetaCoin());

        uint expected = 10000;

        Assert.equal(meta.getBalance(tx.origin), expected, "Владелец счета должен
                                                            иметь 10000 MetaCoin");
        }

        function testInitialBalanceWithNewMetaCoin() {
        MetaCoin meta = new MetaCoin();

        uint expected = 10000;

        Assert.equal(meta.getBalance(tx.origin), expected, "Владелец счета должен
                                                            иметь 10000 MetaCoin");
        }
}
```

Код из листинга 8.6 работает следующим образом:

1. Функции утверждений, такие как `Assert.equal()`, представлены в библиотеке `truffle/Assert.sol`. Это стандартная библиотека проверки утверждений, но вы можете подключить собственную библиотеку, если она свободно интегрируется со средой выполнения тестов Truffle и вызывает правильные события утверждений. Функции утверждений вызывают события, которые затем Truffle перехватывает и отображает соответствующую информацию. Такова архитектура библиотек проверки утверждений Solidity в среде Truffle. Вы можете найти полный перечень функций утверждений в файле `Assert.sol` по адресу: **https://github.com/ConsenSys/truffle/blob/beta/lib/testing/Assert.sol**.
2. Если мы используем путь импорта `truffle/Assert.sol`, то `truffle` — это имя пакета. Мы будем изучать пакеты немного позже.
3. Адреса ваших развернутых контрактов (т. е. контрактов, которые были развернуты в процессе переноса) доступны при помощи библиотеки `truffle/DeployedAddresses.sol`. Эта библиотека тоже входит в состав Truffle. Она заново компилируется и подключается перед каждым запуском тестового пакета. Библиотека предоставляет доступ ко всем вашим развернутым контрактам в виде `DeployedAddresses.<contract name>()`. Вызов возвращает адрес контракта, который в дальнейшем можно использовать для доступа к контракту.
4. Чтобы использовать развернутый контракт, вам придется импортировать код контракта в свой тестовый пакет. Обратите внимание в рассматриваемом примере на строку:

   ```
   import "../contracts/MetaCoin.sol";
   ```

Этот импорт принадлежит тестовому контракту, который расположен в каталоге `./test`, но вынужден выйти за пределы каталога `./test`, чтобы найти контракт `MetaCoin`. Затем он использует этот контракт для преобразования адреса в тип `MetaCoin`.

5. Имена всех тестовых контрактов должны начинаться со слова `Test` (с заглавной буквы `T`). Это помогает отличить файл теста от вспомогательных файлов и контрактов, подвергаемых тестированию, и указывает среде тестирования, какой файл содержит набор тестов.
6. По аналогии с именем файла все тестовые функции должны начинаться со слова `test`, но в нижнем регистре. Каждая функция выполняется как одиночная транзакция и в том порядке, в каком встречается в тестовом файле. Функции утверждений из библиотеки `truffle/Assert.sol` инициируют события, которые среда тестирования использует для определения результата теста. Функции утверждения возвращают логическое (булево) значение, которое представляет результат утверждения. Вы можете использовать его для выхода из теста раньше, чем возникнет ошибка выполнения (ошибка, которую сгенерирует `testrpc`).
7. Вам доступно несколько средств тестирования (test hooks, хуков), которые показаны в следующем примере. Это хуки с названиями: `beforeAll`, `beforeEach`, `afterAll` и `afterEach`, которые не отличаются от таких же средств Mocha в тестах на JavaScript. Вы можете использовать эти хуки для назначения и отмены действий до и после каждого теста, либо до и после каждого набора тестов. Аналогично функции, каждый хук выполняется через одиночную транзакцию. Имейте в виду, что некоторые сложные тесты могут нуждаться в большом количестве действий по настройке событий и превышать лимит газа транзакции. Вы можете обойти лимит газа, создав несколько хуков с различными суффиксами, как показано в примере:

```
import "truffle/Assert.sol";
contract TestHooks {
uint someValue;
function beforeEach() {
     someValue = 5;
}
function beforeEachAgain() {
     someValue += 1;
}
function testSomeValueIsSix() {
     uint expected = 6;
     Assert.equal(someValue, expected, "someValue должно быть = 6");
}
}
```

8. Вы можете задать данные контракта перед выполнением теста и сбросить их в ходе подготовки к следующему тесту. Аналогично тестам на JavaScript, ваша

следующая функция тестирования продолжит работу с тем состоянием контракта, которое осталось после завершения работы предыдущей функции.

Truffle не располагает возможностью проверить, должен ли ваш контракт вбрасывать исключения (речь идет о контрактах, которые используют вбрасывание исключений для уведомления о предсказуемых ошибках). Но вы можете найти оригинальное решение проблемы по адресу: **http://truffleframework.com/tutorials/testing-for-throws-in-Solidity-tests**.

Как перевести валюту на тестовый контракт?

Чтобы перевести валюту на тестовый контракт, в нем должна быть публичная функция `initialBalance()`, которая возвращает значение `uint`. Она может быть написана непосредственно как функция или как публичная переменная. Когда ваш тестовый контракт развернут в сети, Truffle отправит указанную сумму с тестового счета на тестовый контракт. Теперь ваш тестовый контракт может использовать полученную валюту для проведения операций с контрактом, который вы тестируете. Функция `initialBalance()` не является обязательной.

В качестве примера рассмотрим следующий небольшой код:

```
import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/MyContract.sol";

contract TestContract {
// Truffle отправит на TestContract один Эфир после развертывания контракта
public uint initialBalance = 1 ether;

function testInitialBalanceUsingDeployedContract() {
MyContract myContract = MyContract(DeployedAddresses.MyContract());
// выполняем отправку суммы на myContract, затем проверяем результат
myContract.send(...);
}

function () {
// Эта функция не выполняется, если валюта успешно отправлена
}
}
```

Truffle переводит валюту таким способом, который не выполняет вызов резервной функции. Поэтому вы можете продолжать использовать резервную функцию для нужд тестирования в сложных случаях.

Запуск тестов

Для запуска тестовых скриптов достаточно выполнить команду:

```
truffle test
```

Можно указать путь к определенному файлу, который вы хотите запустить, например:

```
truffle test ./path/to/test/file.js
```

Управление пакетами

Пакет представляет собой набор смарт-контрактов и их артефактов. Пакет может зависеть от других пакетов, из которых вы используете смарт-контракты или артефакты. При использовании пакетов в своем проекте важно учитывать, что существуют два места, в которых мы будем использовать контракты и их артефакты: внутри контрактов проекта и внутри кода JavaScript проекта (скрипты переноса и тесты).

Проекты, созданные Truffle, по умолчанию имеют определенную структуру, которая позволяет использовать их в качестве пакетов. Наиболее важными каталогами в пакете `truffle` являются следующие каталоги:

- `/contracts`;
- `/build/contracts` (создается Truffle).

Первый каталог — это каталог ваших контрактов, где хранятся «сырые» контракты Solidity. Во втором каталоге хранятся артефакты сборки в виде файлов `.json`.

Truffle поддерживает два варианта управления пакетами: NPM и EthPM. Вы уже должны знать, что такое пакеты NPM, но пакеты EthPM следует пояснить отдельно. Это конфигурация пакетов в сети Ethereum. Вы можете прочитать больше о EthPM по адресу: **https://www.ethpm.com/**. Стандарт пакетов основан на спецификации ERC190, описывающей публикацию и применение пакетов смарт-контрактов: **https://github.com/ethereum/EIPs/issues/190**

Управление пакетами через NPM

Truffle по умолчанию поставляется с поддержкой NPM, поэтому правильно трактует назначение каталога `node_modules`, если он имеется в вашем проекте. Это означает, что вы можете распространять контракты и библиотеки при помощи механизма NPM и сделать свой код доступным для окружающих, а вам будет доступен их код. В своем проекте вы можете использовать файл `package.json`. Вы можете просто установить произвольный NPM-пакет в свой проект и импортировать его в любые файлы JavaScript, но если он содержит два упомянутых ранее каталога, то обязательно должен называться `truffle`. Установка NPM-пакета в проект Truffle ничем не отличается от установки NPM-пакета в любое приложение Node.js.

Управление пакетами через EthPM

При установке пакетов EthPM автоматически создается каталог `installed_contracts`, если он не существовал ранее. Этот каталог можно использовать аналогично каталогу `node_modules`, о котором мы только что говорили.

Установка пакетов при помощи EthPM столь же проста, как установка NPM-пакетов. Достаточно выполнить команду:

```
truffle install <package name>
```

Можно установить пакет определенной версии:

```
truffle install <package name>@<version>
```

Нумерация пакетов NPM и EthPM соответствует спецификации SemVer[10]. В вашем проекте может присутствовать файл `ethpm.json`, который совпадает с файлом `package.json` для NPM-пакетов. Для установки всех зависимостей, упомянутых в файле `ethpm.json`, выполните команду:

```
truffle install
```

Содержимое файла `ethpm.json` может выглядеть приблизительно так, как показано в этом примере:

```
{
"package_name": "adder",
"version": "0.0.3",
"description": "Simple contract to add two numbers",
"authors": [
"Tim Coulter <tim.coulter@consensys.net>"
],
"keywords": [
"ethereum",
"addition"
],
"dependencies": {
"owned": "^0.0.1"
},
"license": "MIT"
}
```

Создание и публикация NPM-пакетов для Truffle ничем не отличается от процесса создания любых других NPM-пакетов. Чтобы узнать, как создавать и публиковать EthPM-пакеты, прочтите руководство по адресу:
http://truffleframework.com/docs/getting_started/packages-ethpm#publishing-your-own-package.

Независимо от того, какой пакет вы создаете: NPM или EthPM — необходимо выполнить команду:

```
truffle networks --clean
```

Запуск этой команды приводит к удалению всех артефактов для тех идентификаторов сети, которые имеют обобщающий символ `*` в файле конфигурации. Скорее всего, эти сети являются частными (закрытыми) и используются только для

[10] SemVer (Semantic Versioning, семантическая версификация) — спецификация и механизм управления версиями пакетов в Node.js (см. **http://nodesource.com/blog/semver-a-primer/)**.

нужд отладки, поэтому адреса в таких сетях будут недействительными для других проектов. Не используйте эту команду без ясного понимания, что вы делаете. Она не удалит артефакты для частных сетей, указанных как константа, поэтому вам нужно удалить их вручную.

Использование контрактов из пакета

Чтобы задействовать контракты из состава пакета в своем контракте, достаточно использовать оператор `import` языка Solidity. Когда путь импорта не является относительным или абсолютным в явном виде, Truffle полагает, что вы ищете файл из определенного именованного пакета. Рассмотрим пример из `example-truffle-library` по адресу: **https://github.com/ConsenSys/example-truffle-library**:

```
import "example-truffle-library/contracts/SimpleNameRegistry.sol";
```

Поскольку путь не начинается с символов `./`, Truffle понимает, что надо заглянуть в папки `node_modules` или `installed_contracts` каталога `example-truffle-library`. Таким образом Truffle находит путь к контракту, который вы запросили.

Использование артефактов пакета в коде JavaScript

Для обращения к артефактам пакета в коде JavaScript нужно запросить файл `.json` соответствующего пакета при помощи оператора `require` и указать на `truffle-contract`, чтобы преобразовать его в прикладную абстракцию:

```
var data = require("example-trufflelibrary/build/contracts/SimpleNameRegistry.json");
var contract = require("truffle-contract");
var SimpleNameRegistry = contract(data);
```

Доступ к адресам развернутых контрактов пакета в Solidity

Может случиться так, что вашему контракту понадобится взаимодействовать с пакетами ранее развернутых контрактов. Поскольку адреса развернутых контрактов находятся внутри файлов `.json` соответствующих пакетов, код Solidity не может напрямую прочитать содержимое этих файлов. Чтобы код Solidity получил доступ к адресам в файлах `.json`, можно поступить следующим образом: определить в коде Solidity функции, которые определяют адреса контрактов-зависимостей, а когда контракт развернут, вызывать эти функции в коде JavaScript.

Например, вы можете написать код контракта наподобие этого:

```
import "example-truffle-library/contracts/SimpleNameRegistry.sol";

contract MyContract {
        SimpleNameRegistry registry;
        address public owner;

        function MyContract {
        owner = msg.sender;
        }
```

```
// Простой пример использования реестра из пакета
        function getModule(bytes32 name) returns (address) {
        return registry.names(name);
        }

// Назначить реестр, если вы его владелец
        function setRegistry(address addr) {
        if (msg.sender != owner) throw;
        registry = SimpleNameRegistry(addr);
        }
}
```

Так будет выглядеть соответствующий скрипт переноса:

```
var SimpleNameRegistry = artifacts.require("example-trufflelibrary/
contracts/SimpleNameRegistry.sol");

module.exports = function(deployer) {
// Разворачиваем контракт и определяем адрес реестра
        deployer.deploy(MyContract).then(function() {
        return MyContract.deployed();
}).then(function(deployed) {
        return deployed.setRegistry(SimpleNameRegistry.address);
        });
};
```

Работа с консолью Truffle

Иногда бывает полезно тестировать и отлаживать контракт в интерактивном режиме или вручную выполнять транзакции. Благодаря Truffle, у нас есть простой способ выполнять эти действия в интерактивной консоли, когда ваш контракт готов к использованию.

Чтобы открыть консоль, выполните команду:

```
truffle console
```

Консоль подключается к узлу Ethereum, определенному в конфигурации проекта. Команда может также содержать опцию **`--network`**, чтобы задать определенный узел.

Обратите внимание на следующие особенности консоли:

- в консоли можно выполнять команды Truffle. Например, если вы введете в консоли команду:

  ```
  migrate --reset
  ```

  она будет интерпретирована так же, как команда:

  ```
  truffle migrate --reset
  ```

- введенная за пределами консоли;
- все ваши контракты должны быть доступны и готовы к использованию;
- после каждой команды (такой, как **`migrate --reset`**) ваши контракты обновляются, т. е. вы можете немедленно начинать использование новых адресов и бинарного кода;
- все объекты `web3` доступны и настроены на подключение к вашему узлу Ethereum;
- все команды, которые возвращают обещание, будут автоматически завершены, а результат выведен на печать. Это делает ненужным использование оператора `.then()` для простых команд. Например, вы можете ввести код наподобие такого:

```
MyContract.at("0xabcd...").getValue.call();
```

Запуск внешних скриптов в контексте Truffle

Часто возникает необходимость запустить внешние скрипты, которые будут взаимодействовать с вашими контрактами. Благодаря Truffle, у нас есть простой способ это сделать: загрузить контракты в нужной сети и автоматически подключиться к узлу Ethereum в соответствии с конфигурацией нашего проекта.

Для запуска внешнего скрипта выполните команду:

```
truffle exec <path/to/file.js>
```

Для правильного выполнения внешних скриптов Truffle ожидает, что они будут экспортировать функцию, которая принимает один параметр в качестве обратного вызова. Вы можете делать что угодно в этом скрипте, пока выполняется обратный вызов и пока не завершится скрипт. Обратный вызов принимает ошибку как первый и единственный параметр. Если получена ошибка, выполнение остановится, и процесс вернет ненулевой код выхода.

Структура, которой должны соответствовать внешние скрипты, выглядит следующим образом:

```
module.exports = function(callback) {
        // здесь выполняем действия
        callback();
}
```

Создание клиента в Truffle

Теперь, когда вы знаете, как компилировать, развертывать и тестировать смарт-контракты при помощи Truffle, настало время скомпоновать приложение-клиент для наших альткоинов. Прежде чем начать изучение этой темы, напомним, что Truffle не позволяет подписывать транзакции при помощи аккаунтов, находящихся вне вашего узла Ethereum. Иными словами, он не располагает функциональностью наподобие `sendRawTransaction`.

Создание клиента при помощи Truffle означает интеграцию артефактов в исходный код клиента и подготовку исходного кода клиента к развертыванию.

Для создания клиента следует выполнить команду:

```
truffle build
```

При выполнении этой команды Truffle обращается к свойству `build` в файле конфигурации проекта.

Запуск внешних команд

Для создания клиента можно использовать инструмент командной строки. Если свойство `build` является строкой, Truffle предполагает, что мы хотим запустить команду для создания клиента, поэтому он выполняет строку как команду. Команде доступен обширный набор переменных среды, которые можно интегрировать с Truffle.

Вы можете заставить Truffle запустить инструмент командной строки для создания клиента при помощи кода конфигурации, аналогичного следующему примеру:

```
module.exports = {
// This will run the &grave;webpack&grave; command on each build.
//
// The following environment variables will be set when running the
command:
// WORKING_DIRECTORY: root location of the project
// BUILD_DESTINATION_DIRECTORY: expected destination of built assets
// BUILD_CONTRACTS_DIRECTORY: root location of your build contract files
(.sol.js)
//
build: "webpack"
}
```

Запуск пользовательских функций

Для создания клиента можно использовать функцию JavaScript. Если свойство `build` является функцией, Truffle будет выполнять эту функцию всякий раз при создании клиента. Функция дает много информации о проекте, которая может быть интегрирована с Truffle.

Вы можете заставить Truffle выполнить функцию для создания клиента при помощи кода конфигурации, аналогичного следующему примеру:

```
module.exports = {
       build: function(options, callback) {
       // Do something when a build is required. &grave;options&grave;
       contains these values:
       //
```

```
        // working_directory: root location of the project
        // contracts_directory: root directory of .sol files
        // destination_directory: directory where truffle expects the built
        assets (important for &grave;truffle serve&grave;)
        }
}
```

Вы также можете создать объект, который содержит метод создания, наподобие показанного здесь. Это отлично подходит для тех, кто хочет опубликовать пакет для создания клиента.

Конструктор Truffle по умолчанию

Truffle содержит NPM-пакет truffle-default-builder, который является конструктором Truffle по умолчанию. Этот конструктор экспортирует объект, содержащий метод создания клиента, работающий полностью идентично методу, о котором шла речь в предыдущем разделе.

Конструктор по умолчанию можно использовать для создания веб-клиента вашего децентрализованного приложения, когда сервер обслуживает только статичные файлы, а вся функциональность обеспечена клиентской частью.

Прежде, чем приступить к углубленному изучению конструктора по умолчанию, следует установить его командой:

```
npm install truffle-default-builder --save
```

Теперь измените файл конфигурации, чтобы он выглядел следующим образом:

```
var DefaultBuilder = require("truffle-default-builder");

        module.exports = {
                networks: {
                        development: {
                        host: "localhost",
                        port: 8545,
                        network_id: "10"
                },
                live: {
                        host: "localhost",
                        port: 8545,
                        network_id: "1"
                        }
                },
                build: new DefaultBuilder({
                "index.html": "index.html",
                "app.js": [
                "javascripts/index.js"
                ],
```

```
        "bootstrap.min.css": "stylesheets/bootstrap.min.css"
        })
};
```

Конструктор по умолчанию дает вам полный контроль над тем, как организована структура файлов и папок в вашем клиенте.

Эта конфигурация описывает пункт назначения `targets` (левая часть) с указанием содержимого в виде файлов, папок и массивов файлов (правая часть). Каждый пункт назначения создается обработкой файлов, указанных в правой части, с учетом их расширений, конкатенации результатов и сохранения результирующего файла (target file) в место создания. В нашем случае вместо массива указана строка, этот файл будет обработан при необходимости, а затем скопирован в указанное место. Если строка заканчивается символом «/», он будет интерпретирован как указание на каталог, и этот каталог будет просто скопирован без обработки. Все пути в правой части указываются относительно каталога `app/`.

Вы можете в любое время изменить этот файл конфигурации и структуру каталогов. Вовсе не обязательно иметь каталоги `javascripts` и `stylesheets`, но позаботьтесь соответствующим образом исправить вашу конфигурацию.

Если вы хотите использовать конструктор по умолчанию для интеграции Truffle с клиентской частью вашего веб-приложения, позаботьтесь о создании целевого файла `app.js`, к которому конструктор будет присоединять код. Любое другое имя файла не подойдет.

Конструктор по умолчанию обладает следующими возможностями:

- автоматически импортирует артефакты скомпилированных контрактов, информацию о развернутых контрактах и конфигурацию узла Ethereum в исходный код клиента;
- подключает рекомендованные зависимости, включая `web3` и `truffle-contract`;
- компилирует файлы `ES6` и `JSX`;
- компилирует файлы `SASS`;
- минифицирует[11] файлы `asset`.

Для отслеживания изменений в каталогах `contracts`, `app` и файле конфигурации можно использовать команду:

```
truffle watch
```

Как только обнаружено изменение, команда перекомпилирует контракты, создает новые файлы артефактов и пересобирает клиент. Но при этом она не запускает скрипты переноса и тестирования.

[11] Минификация — уменьшение размера исходного кода путем удаления ненужных символов без потери функциональности.

Создание клиента

Давайте напишем клиентскую часть нашего децентрализованного приложения, а затем выполним его компоновку при помощи конструктора Truffle. Сначала создайте файлы и каталоги на основе рассмотренной ранее конфигурации: создайте каталог `app` и внутри него создайте файл `index.html`, а также каталоги `javascripts` и `stylesheets`. Внутри каталога `javascripts` создайте файл `index.js`, а внутрь каталога `stylesheets` скачайте и сохраните файл CSS для Bootstrap 4. Вы можете найти его по адресу: **https://v4-alpha.getbootstrap.com/getting-started/download/#bootstrap-css-and-js**.

В файл `index.html` поместите код из листинга 8.7.

Листинг 8.7. Код файла `index.html`

```
<!doctype html>
<html>
<head>
<link rel="stylesheet" type="text/css" href="bootstrap.min.css">
</head>

<body>
        <div class="container">
        <div class="row">
        <div class="col-md-6">
        <br>
        <h2>Отправить Метакоины</h2>
        <hr>

        <form id="sendForm">
        <div class="form-group">
        <label for="fromAddress">Выбрать адрес счета</label>
        <select class="form-control" id="fromAddress">
        </select>
        </div>
        <div class="form-group">
        <label for="amount">Сколько метакоинов вы хотите отправить?
        </label>
        <input type="text" class="form-control" id="amount">
        </div>
        <div class="form-group">
        <label for="toAddress">Введите адрес, на который хотите отправить
                                                                метакоины</label>
        <input type="text" class="form-control" id="toAddress" placeholder="Prefixed
                                                                with 0x">
        </div>
```

```
        <button type="submit" class="btn btn-primary">Отправить</button>
        </form>
        </div>

        <div class="col-md-6">
        <br>
        <h2>Найти баланс</h2>
        <hr>

        <form id="findBalanceForm">
        <div class="form-group">
        <label for="address">Выбрать адрес счета</label>
        <select class="form-control" id="address">
        </select>
        </div>
        <button type="submit" class="btn btn-primary"></button>
        </form>
        </div>
        </div>
        </div>
        <script type="text/javascript" src="/app.js"></script>
        </body>
</html>
```

В коде из листинга 8.7 мы загружаем файлы `bootstrap.min.css` и `app.js`. Далее, у нас есть две формы: одна из них — для отправки метакоинов на другой счет, а вторая — для проверки баланса своего счета. В первой форме пользователь должен выбрать счет, ввести количество метакоинов и адрес получателя. Во второй форме пользователь просто выбирает адрес счета, баланс которого хочет проверить.

В файл `index.js` поместите код из листинга 8.8.

Листинг 8.8. Код файла `index.js`

```
window.addEventListener("load", function(){
var accounts = web3.eth.accounts;

var html = "";

for(var count = 0; count < accounts.length; count++)
{
        html = html + "<option>" + accounts[count] + "</option>";
}

document.getElementById("fromAddress").innerHTML = html;
document.getElementById("address").innerHTML = html;
```

```
MetaCoin.detectNetwork();
})

document.getElementById("sendForm").addEventListener("submit", function(e){
        e.preventDefault();
        MetaCoin.deployed().then(function(instance){
        return
        instance.sendCoin(document.getElementById("toAddress").value,
        document.getElementById("amount").value, {
        from:
        document.getElementById("fromAddress").options[document.getElementById("from
Address").selectedIndex].value
        });
}).then(function(result){
        alert("Транзакция успешно обработана. Txn Hash: " + result.tx);
}).catch(function(e){
        alert("Произошла ошибка");
})
})

document.getElementById("findBalanceForm").addEventListener("submit",
        function(e){
        e.preventDefault();
        MetaCoin.deployed().then(function(instance){
        return
        instance.getBalance.call(document.getElementById("address").value);
        }).then(function(result){
        console.log(result);
        alert("Баланс: " + result.toString() + " метакоинов");
        }).catch(function(e){
        alert("Произошла ошибка");
})
})
```

Код из листинга 8.8 работает так:

1. Конструктор по умолчанию создает артефакты, доступные через глобальный объект `__contracts__`.
2. Также конструктор делает доступными абстракции всех контрактов. Они доступны как глобальные переменные, имена которых совпадают с именами контрактов.
3. Конструктор предоставляет объект `web3` через уже существующий провайдер, а также определяет провайдера абстракций контракта. Он подключает объект `web3` к сети с именем `development` и адресом подключения по умолчанию **http://localhost:8545** (если не задан иной адрес в явном виде).

4. Мы дожидаемся полной загрузки страницы. Когда загрузка завершена, мы запрашиваем список счетов подключенного узла и отображаем его в обеих формах, а также вызываем метод `detectNetwork()` абстракции `MetaCoin`.
5. Далее, у нас имеется обработчик событий кнопки **Отправить** в обеих формах. Они делают именно то, чего от них ожидают, а результат отображается во всплывающем окне.
6. Когда передано содержимое первой формы, мы обращаемся к экземпляру развернутого контракта `MetaCoin` и вызываем метод `sendCoin()` с корректными аргументами.
7. Когда передано содержимое второй формы, мы запрашиваем баланс выбранного счета вызовом метода `getBalance()` на виртуальной машине Ethereum, вместо того, чтобы транслировать транзакцию в сеть.

Теперь идем дальше и выполняем команду: **`truffle build`**. Вы увидите, что Truffle создаст файлы `index.html`, `app.js` и `bootstrap.min.css` в каталоге `build` и поместит в них окончательный код развертывания клиента.

Сервер Truffle

Truffle распространяется со встроенным веб-сервером. Этот сервер лишь выдает файлы из каталога `build` с правильным заголовком типа MIME. Помимо этого, он не приспособлен к выполнению чего-либо еще.

Для запуска сервера выполните команду:

```
truffle serve
```

По умолчанию сервер доступен через порт 8080, но вы можете использовать опцию **`-p`** для назначения другого номера порта.

По аналогии с командой **`truffle watch`**, этот веб-сервер также отслеживает изменения в каталогах `contracts`, `app` и в файле конфигурации. Если обнаружено изменение, он перекомпилирует контракты, генерирует новые файлы артефактов и пересобирает клиента. Но веб-сервер не запускает скрипты переноса и тесты.

Поскольку конструктор `truffle-default-builder` помещает финальный код развертывания в каталог `build`, для получения доступа к этим файлам через сеть достаточно выполнить команду запуска сервера: **`truffle serve`**.

Давайте протестируем наш веб-клиент. Наберите в адресной строке браузера адрес `http://localhost:8080`, и вы должны увидеть окно, показанное на рис. 8.1.

Разумеется, у вас будут отображаться другие адреса счетов. При развертывании контракта он помещает все метакоины по адресу развертывания этого контракта. По этой причине на первом счете у нас имеется 10 000 метакоинов. Теперь давайте отправим пять метакоинов с первого счета на второй счет и нажмем кнопку **Submit**. Вы должны увидеть окно, аналогичное показанному на рис. 8.2.

Теперь проверьте баланс второго счета. Для этого в списке правой формы выберите второй счет и нажмите кнопку **Check Balance** (рис. 8.3).

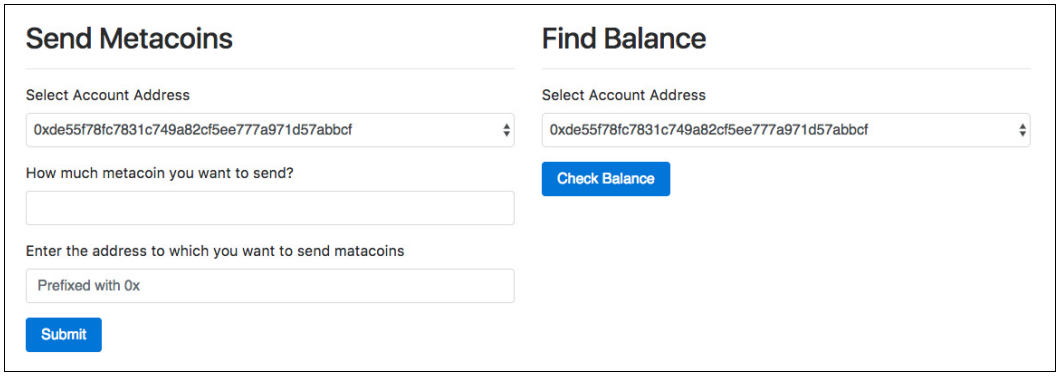

Рис. 8.1. Начальный экран веб-клиента

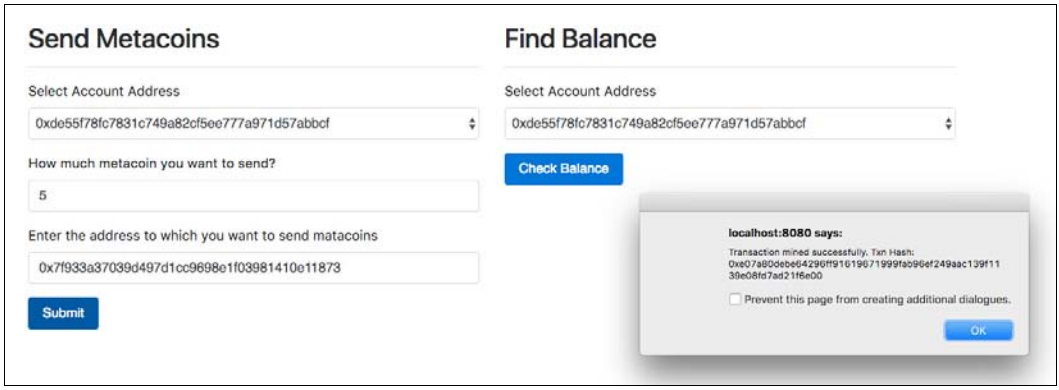

Рис. 8.2. Окно клиента после отправки метакоинов

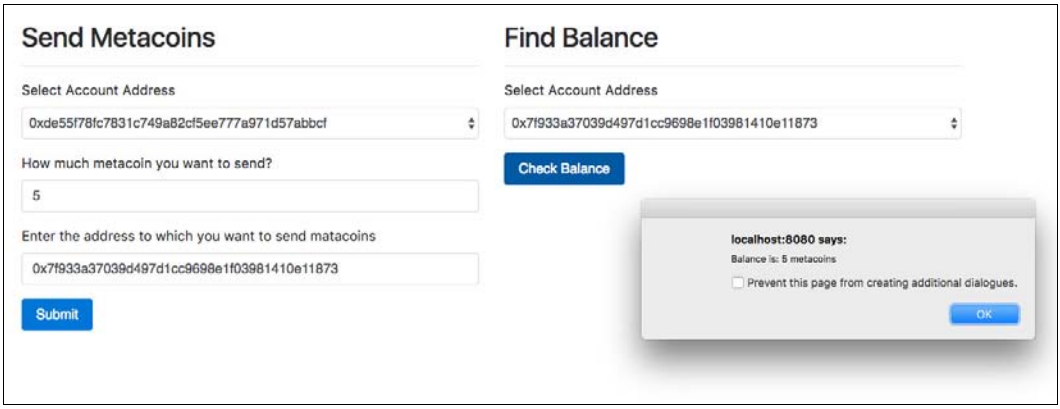

Рис. 8.3. Окно проверки баланса второго счета

Заключение

В этой главе мы подробно изучили разработку децентрализованных приложений и соответствующих веб-клиентов в среде Truffle. Мы убедились, что Truffle действительно упрощает написание, компиляцию, развертывание и тестирование приложений. Мы также увидели, как просто переключаться между сетями прямо в работающем клиенте при помощи `truffle-contract`, ничего не меняя в исходном коде. Теперь мы готовы приступить к созданию блокчейна на корпоративном уровне.

9

Разработка блокчейна для консорциума

Консорциумы (профильные объединения, обычно состоящие из нескольких участников, таких как банки, сайты электронной коммерции, правительственные учреждения, больницы и т. п.) могут использовать технологию блокчейна, чтобы решить ряд проблем и сделать свою работу быстрее и дешевле. Хотя они и представляют в общих чертах, как им может помочь блокчейн, реализация блокчейна Ethereum подходит им не во всех случаях. Тем не менее, несмотря на то, что существуют варианты блокчейна специально для консорциумов (например, Hyperledger), в этой книге мы рассматриваем прежде всего Ethereum. Поэтому сейчас мы расскажем, как приспособить Ethereum для создания блокчейна консорциума.

В основу нашего блокчейна будет заложен сетевой узел под названием Parity. Существуют и альтернативные решения, такие как J.P. Morgan Quorum, но мы воспользуемся именно Parity, потому что к моменту работы над книгой он уже достаточно долго существовал, и его использовали многие предприятия, а другие решения применялись лишь ограниченным кругом организаций. Однако имейте в виду, что Parity не обязательно станет лучшим выбором для вашего предприятия. Обязательно изучите другие варианты, чтобы выбрать наиболее подходящее решение.

В этой главе будут рассмотрены следующие темы:

- почему Ethereum плохо подходит для блокчейна консорциума?
- что такое узел Parity и каковы его особенности?
- что такое консенсус с доказательством полномочий, и какие типы консенсуса поддерживает Parity?
- как работает протокол консенсуса Aura?
- скачивание и установка Parity;
- создание блокчейна консорциума на основе Parity.

Что такое блокчейн консорциума?

Чтобы лучше понять, что такое блокчейн консорциума, или, вернее, какая разновидность блокчейна нужна консорциуму, рассмотрим простой пример. Банк хочет создать блокчейн, чтобы сделать денежные переводы проще, быстрее и дешевле. В этом случае банк выдвигает такие требования:

- *скорость* — сеть, в которой работает блокчейн, должна подтверждать транзакции практически в режиме реального времени. Однако в сети Ethereum блокирующий интервал между транзакциями составляет 12 секунд, вдобавок клиент обычно ждет подтверждения транзакции несколько минут;
- *контроль доступа* — банки хотят, чтобы доступ к блокчейну был строго ограничен, и контролируемый доступ подразумевает различные ограничения. Например, необходимость получить разрешение на подключение к сети, иметь полномочия на создание блоков, полномочия на отправку определенных транзакций и т. п.;
- *безопасность* — консенсус на основе доказательства выполнения работы не гарантирует безопасность блокчейна в частных сетях, потому что они состоят из слишком маленького числа участников и не обладают достаточно большой совокупной вычислительной мощностью. Следовательно, нам нужен другой протокол консенсуса;
- *приватность* — несмотря на то, что сеть является частной, внутри этой сети все равно надо сохранять приватность. Существуют два типа приватности:
  - *приватность участника* — остальные участники сети не должны иметь возможности отследить кого-либо[1] через его транзакции. В сети Ethereum эту проблему решают созданием нескольких аккаунтов, но тогда смарт-контракты не смогут правильно работать, потому что нет способа доказать контракту, что несколько аккаунтов принадлежат одному владельцу;
  - *приватность данных* — иногда нам нужно, чтобы некоторые данные могли видеть только строго заданные узлы, а не каждый узел сети.

В этой главе мы разберемся, как можно решить указанные проблемы в сети Ethereum.

Что такое консенсус с доказательством полномочий?

Доказательство полномочий (Proof-of-Authority, PoA) — это механизм достижения консенсуса, при котором происходит обращение к списку *валидаторов*[2]. Валидато-

[1] То есть получить его приватную информацию.

[2] В привычном мире бумажного документооборота валидаторами (заверителями) являются нотариусы, паспортные службы и прочие так называемые *удостоверяющие органы*.

ры — это особая группа аккаунтов/узлов, которым разрешено формировать консенсус. Они заверяют транзакции и блоки.

В протоколе PoA не задействован механизм майнинга. Существуют разные типы протоколов PoA, и они различаются принципами работы. Например, Hyperledger и Parity основаны на PoA, но главное различие между ними состоит в том, что Hyperledger использует PBFT, а Parity — итеративный процесс.

Введение в Parity

Parity — это узел Ethereum, написанный с нуля специально для обеспечения корректности/проверяемости, модульности, низкой нагрузки и высокой производительности. Он написан на языке Rust — это гибридный язык[3] с акцентом на производительность, специально разработанный Parity Technologies Ltd. На момент подготовки книги самой новой была версия Parity 1.7.0, и далее будет рассмотрена именно эта версия[4]. Мы изучим Parity настолько, насколько это потребуется для создания блокчейна консорциума. Для более глубокого изучения обратитесь к официальной документации на сайте **https://www.parity.io/**.

По сравнению с Go-Ethereum, Parity обладает более обширным набором инструментов, например браузером приложений web3, развитыми средствами управления аккаунтами и другими подобными опциями. Но ключевой особенностью Parity является использование протокола доказательства полномочий (PoA) вместо протокола доказательства работы (PoW). На данный момент Parity поддерживает PoA-протоколы Aura и Tendermint. Рекомендуем использовать протокол Aura, т. к. Tendermint находится на стадии разработки.

Aura намного лучше, чем PoW, подходит для блокчейна с ограниченным доступом, потому что обеспечивает быструю обработку блоков и значительно более высокую безопасность в частных сетях.

Принципы работы Aura

Давайте рассмотрим в общих чертах, как работает Aura. Прежде всего, необходимо, чтобы для каждого узла сети был назначен один и тот же список валидаторов — это список адресов аккаунтов, которые достигают консенсуса. Узел может быть или не быть валидатором. Но даже единственный узел-валидатор должен хранить список валидаторов и поэтому может достигать консенсуса с самим собой.

Если список валидаторов должен всегда оставаться неизменным, то его можно предоставить в статическом виде внутри генезисного блока. Динамически изменяемый

[3] Rust — это гибрид концепций императивного, объектно-ориентированного и функционального подхода в одном языке программирования.

[4] Во время работы над русским переводом была выпущена версия 1.8.3. Версии обычно обновляются каждые шесть недель и пока поддерживают обратную совместимость.

список можно предоставить в смарт-контракте, и каждый узел будет знать об изменениях. В смарт-контракте можно реализовать различные стратегии добавления новых валидаторов.

Время генерации блока задается в генезисном файле и полностью зависит от вашего решения. Для частных сетей хорошо подходит интервал в три секунды. В протоколе Aura каждые три секунды выбирается новый валидатор, и он отвечает за создание, проверку, подписание и трансляцию блока в сеть. Нам нет нужды глубоко вникать в алгоритм выбора валидаторов, потому что он никак не связан с разработкой децентрализованных приложений. Но в общем виде формула выбора нового валидатора имеет следующий вид:

```
UNIX_TIMESTAMP / BLOCK_TIME %NUMBER_OF_TOTAL_VALIDATORS
```

Алгоритм выбора достаточно разумен и предоставляет всем валидаторам равные шансы. Когда другой узел получает блок, он проверяет, является ли действующий валидатор отправителем этого блока. Если нет, то блок отвергается. В отличие от алгоритма PoW, когда валидатор создает блок, он не получает вознаграждение. В протоколе Aura только от нашего решения зависит, генерировать ли пустой блок, если к моменту генерации нет ожидающих транзакций.

Вы можете спросить, что произойдет, если очередной валидатор по какой-то причине не сможет создать и транслировать следующий блок. В качестве примера допустим, что A — валидатор пятого блока и B — валидатор шестого блока. Интервал между блоками пять секунд. Если A не смог создать и передать блок, то через пять секунд придет валидатор B и создаст этот блок. Так что не случится ничего серьезного. Особенности создания блока будут отражены в метке времени.

Также вы можете спросить, существует ли вероятность того, что в сети возникнут несколько разных блокчейнов, как это случается в протоколе PoW, если два майнера выработали блоки одновременно. Да, это может случиться в силу разных причин. В качестве примера рассмотрим один из вариантов развития событий и покажем, как сеть автоматически решает проблему. Итак, существуют пять валидаторов: A, B, C, D и E. Интервал между блоками пять секунд. Допустим, валидатор A был выбран первым, и передал готовый блок в сеть. Но этот блок по какой-то причине не дошел до D и E, поэтому они решили, что A не передал блок. Далее алгоритм выбрал валидатора B для генерации следующего блока. Валидатор B генерирует следующий блок, вставляет его в блокчейн после блока, созданного узлом A, и передает в сеть. Теперь D и E получили новый блок, но отвергают его, потому что не получили предыдущий блок. По этой причине D и E сформируют свою цепочку блоков, которая отличается от цепочки, сформированной узлами A, B и C. Теперь узлы A, B и C будут отвергать блоки, созданные узлами D и E, и наоборот. Для решения этой проблемы применяется понятие *уровень точности* (accuracy score). Уровень точности блокчейна валидаторов A, B и C будет выше, поэтому валидаторы D и E откажутся от своей версии и обновят блокчейн до версии, заверенной узлами A, B и C. Например, когда валидатор B передает в сеть свой блок, вместе с ним он передает уровень точности нового блокчейна. Если точность этого блок-

чейна выше, то D и E будут должны заменить свои блокчейны на версию, предложенную узлом B. Таким образом, проблема решена. Уровень точности блокчейна вычисляется по формуле:

```
U128_max * BLOCK_NUMBER_OF_LATEST_BLOCK -
(UNIX_TIMESTAMP_OF_LATEST_BLOCK / BLOCK_TIME)
```

Сначала точность блокчейна сравнивается по его длине — чем больше блоков, тем лучше. Если длина совпадает, то предпочтение отдается версии с более старым последним блоком.

Вы можете более глубоко изучить протокол Aura, перейдя по ссылке: **https://github.com/paritytech/parity/wiki/Aura**.

Начинаем работу с Parity

Для работы с Parity потребуется Rust версии 1.16.0[5].

Установка Rust

Способы установки Rust зависят от операционной системы вашего компьютера.

Для Linux

В Linux-системах выполните команду терминала:

```
curl https://sh.rustup.rs -sSf | sh
```

Кроме этого, для Parity необходимо установить пакеты `gcc`, `g++`, `libssl-dev/openssl`, `libudev-dev`, и `pkg-config`, если они не были установлены ранее.

Для OS X

В операционной системе OS X выполните команду терминала:

```
curl https://sh.rustup.rs -sSf | sh
```

Кроме этого, для Parity потребуется компилятор Clang, который поставляется вместе с инструментом командной строки Xcode или может быть установлен отдельно при помощи Homebrew.

Для Windows

Прежде всего, у вас должна быть установлена среда разработки Visual Studio 2015 с поддержкой C++. Затем скачайте и запустите установщик, доступный по адресу: **https://static.rust-lang.org/rustup/dist/x86_64-pc-windows-msvc/rustupinit.exe**.

Запустите приложение VS2015 x64 Native Tools Command Prompt и выполните следующую команду для установки и настройки пакета средств MSVC:

```
rustup default stable-x86_64-pc-windows-msvc
```

[5] Для работы с Parity 1.8.3 требуется Rust не ниже 1.21.0.

Скачивание, установка и запуск Parity

После установки Rust в своей операционной системе вы можете выполнить однострочную команду установки Parity:

```
cargo install --git https://github.com/paritytech/parity.git parity
```

Для проверки установки Parity введите команду:

```
parity --help
```

Если установка прошла успешно, вы увидите полный список команд и опций.

Создание частной сети

Пришло время создать наш блокчейн для консорциума. Мы создадим два заверяющих узла, соединенных друг с другом и использующих протокол Aura для достижения консенсуса. Оба узла будут запущены на одном компьютере.

Создание аккаунтов

Откройте два терминала командной строки: первый терминал — для первого валидатора, а второй — для второго валидатора. Первый узел будет содержать два аккаунта, а второй узел — только один аккаунт. Второй аккаунт первого узла будет располагать некоторой начальной суммой в эфирах, следовательно в сети будет обращаться некоторое количество валюты.

В первом терминале *дважды* выполните такую команду:

```
parity account new -d ./validator0
```

Оба раза вам будет предложено ввести пароль. В данном случае просто введите одинаковый пароль.

Во втором терминале однократно выполните команду:

```
parity account new -d ./validator1
```

Введите пароль для этого аккаунта.

Создание файла спецификации

Узлы любой сети обращаются к общему файлу спецификации. Этот файл рассказывает узлу о генезисном блоке, кто является валидаторами и т. п. Мы создадим смарт-контракт, который содержит список валидаторов. Существуют два типа таких контрактов: *рапортующий* (reporting contract) и *не рапортующий* (non-reporting contract) — мы должны выбрать только один вариант.

Различие между этими двумя типами контракта состоит в следующем. Контракт без рапорта просто возвращает список валидаторов. Рапортующий же контракт может принимать меры против неумышленного неправильного поведения узла (например, такой узел может просто не получать блок от назначенного валидатора)

или против злоумышленного неправильного поведения (например, выпуска двух разных блоков для одного и того же шага).

Интерфейс контракта без рапорта должен иметь, как минимум, следующий интерфейс:

```
{"constant":true,"inputs":[],"name":"getValidators","outputs":[{"name":"","type":"a
ddress[]"}],"payable":false,"type":"function"}
```

Функция `getValidators` будет вызываться для определения списка валидаторов при появлении каждого блока. Правила переключения между валидаторами определяются контрактом, реализующим эту функцию.

Рапортующий контракт должен иметь, как минимум, следующий интерфейс:

```
[
{"constant":true,"inputs":[],"name":"getValidators","outputs":[{"name":"","type":"a
ddress[]"}],"payable":false,"type":"function"},
{"constant":false,"inputs":[{"name":"validator","type":"address"}],"name":"reportMa
liciors","outputs":[],"payable":false,"type":"function"},
{"constant":false,"inputs":[{"name":"validator","type":"address"}],"name":"reportBe
nign","outputs":[],"payable":false,"type":"function"}
]
```

При обнаружении неумышленных или злоумышленных отклонений поведения механизм консенсуса вызывает, соответственно, функцию `reportBenign` или `reportMalicious`.

Мы создадим рапортующий контракт. Базовый пример такого контракта приведен в листинге 9.1.

Листинг 9.1. Базовый пример рапортующего контракта

```
contract ReportingContract {
address[] public validators =
[0x831647ec69be4ca44ea4bd1b9909debfbaaef55c,
0x12a6bda0d5f58538167b2efce5519e316863f9fd];
mapping(address => uint) indices;
address public disliked;

function ReportingContract() {
        for (uint i = 0; i < validators.length; i++) {
        indices[validators[i]] = i;
        }
}

// Вызывается при каждом блоке для обновления списка валидаторов
function getValidators() constant returns (address[]) {
        return validators;
}
```

```
// Расширяет список валидаторов
function addValidator(address validator) {
        validators.push(validator);
}

// Удаляет валидатора из списка
function reportMalicious(address validator) {
        validators[indices[validator]] = validators[validators.length-1];
        delete indices[validator];
        delete validators[validators.length-1];
        validators.length--;
}

function reportBenign(address validator) {
        disliked = validator;
}
}
```

Этот код не требует пояснений. Убедитесь, что в массиве адресов этого кода вы заменили исходные адреса на реальные адреса узлов `validator1` и `validator2`, поскольку мы будем использовать эти адреса для заверения. Теперь скомпилируйте этот контракт при помощи любого из изученных ранее компиляторов.

Теперь создадим файл конфигурации. Создайте файл с именем `spec.json` и поместите в него код из листинга 9.2.

Листинг 9.2. Исходный код файла `spec.json`

```
{
"name": "ethereum",
"engine": {
        "authorityRound": {
        "params": {
                "gasLimitBoundDivisor": "0x400",
                "stepDuration": "5",
                "validators" : {
                "contract": "0x0000000000000000000000000000000000000005"
                }
                }
        }
},
"params": {
        "maximumExtraDataSize": "0x20",
        "minGasLimit": "0x1388",
        "networkID" : "0x2323"
},
```

```
"genesis": {
        "seal": {
                "authorityRound": {
                "step": "0x0",
                "signature": "0x0000000000000000000000000000000000000000000000000
                0000000000000000000000000000000000000000000000000000000000000000
                0000000000000000000000000000000000"
        }
        },
                "difficulty": "0x20000",
                "gasLimit": "0x5B8D80"
        },
        "accounts": {
"0x0000000000000000000000000000000000000001": { "balance": "1",
"builtin": { "name": "ecrecover", "pricing": { "linear": { "base": 3000,
"word": 0 } } } },
"0x0000000000000000000000000000000000000002": { "balance": "1",
"builtin": { "name": "sha256", "pricing": { "linear": { "base": 60, "word":
12 } } } },
"0x0000000000000000000000000000000000000003": { "balance": "1",
"builtin": { "name": "ripemd160", "pricing": { "linear": { "base": 600,
"word": 120 } } } },
"0x0000000000000000000000000000000000000004": { "balance": "1",
"builtin": { "name": "identity", "pricing": { "linear": { "base": 15,
"word": 3 } } } },
"0x0000000000000000000000000000000000000005": { "balance": "1",
"constructor" : "0x6060604052604060405190810160405280738316471647" },
"0x004ec07d2329997267Ec62b4166639513386F32E": { "balance":
"1000000000000000000000" }
        }
}
```

Код из листинга 9.2 работает следующим образом:

1. Свойство `engine` служит для задания протокола и его специфических параметров. Наш движок протокола называется `authorityRound`, что соответствует протоколу Aura. Свойство `gasLimitBoundDivisor` определяет настройку лимита газа и содержит обычное значение `ethereum`. В свойство `validators` вложен параметр `contract`, который содержит адрес уведомляющего контракта. Параметр `stepDuration` — это интервал между блоками в секундах.
2. В свойстве `params` имеет значение только идентификатор сети. Остальные параметры стандартные для всех цепочек.
3. Свойство `genesis` имеет несколько стандартных значений для консенсуса `authorityRound`.

4. Свойство `accounts` применяется для перечисления исходных аккаунтов и контрактов, которые находятся в сети. Первые четыре — это стандартные встроенные контракты Ethereum. Они должны быть добавлены для использования контрактов на языке Solidity. Пятый контракт — это уведомляющий контракт. Убедитесь, что вы не забыли подставить байт-код *своего* контракта в параметр `constructor`. Последний аккаунт сгенерирован в терминале, относящемся к `validator1`. Он применяется для отправки эфиров в сеть. Подставьте туда свой номер.

Прежде, чем мы продолжим, создайте файл с именем `node.pwds`. Поместите в этот файл пароль к созданным вами аккаунтам. Этот файл будут использовать валидаторы, чтобы разблокировать аккаунты и подписывать блоки.

Запуск узлов

Теперь у нас есть все необходимое для запуска узлов-валидаторов. В первом терминале выполните следующую команду для запуска первого валидатора:

```
parity --chain spec.json -d ./validator0 --force-sealing --engine-signer
"0x831647ec69be4ca44ea4bd1b9909debfbaaef55c" --port 30300 --jsonrpc-port
8540 --ui-port 8180 --dapps-port 8080 --ws-port 8546 --jsonrpc-apis
web3,eth,net,personal,parity,parity_set,traces,rpc,parity_accounts --
password "node.pwds"
```

Команда устроена следующим образом:

- опция **`--chain`** указывает путь к файлу спецификации;
- опция **`-d`** указывает на каталог данных;
- опция **`--force-sealing`** обеспечивает создание блоков, даже если они не содержат транзакции;
- опция **`--engine-signer`** определяет адрес аккаунта, который будет применять узел для подписания блоков, т. е. адрес валидатора. В случае злонамеренного заверения пригодится опция **`--force-sealing`**. Она гарантирует, что правильная цепочка длиннее. Убедитесь, что вы подставили в команду свой сгенерированный адрес. Это должен быть первый адрес, сгенерированный ранее в терминале;
- опция **`--password`** задает файл пароля.

Во втором терминале выполните команду для запуска второго валидатора:

```
parity --chain spec.json -d ./validator1 --force-sealing --engine-signer
"0x12a6bda0d5f58538167b2efce5519e316863f9fd" --port 30301 --jsonrpc-port
8541 --ui-port 8181 --dapps-port 8081 --ws-port 8547 --jsonrpc-apis
web3,eth,net,personal,parity,parity_set,traces,rpc,parity_accounts --
password "/Users/narayanprusty/Desktop/node.pwds"
```

Убедитесь, что вы правильно указали адрес своего аккаунта и путь к файлу пароля на вашем компьютере.

Подключение узлов

Теперь нам надо соединить наши узлы между собой. Откройте новое окно терминала и выполните следующую команду:

```
curl --data '{"jsonrpc":"2.0","method":"parity_enode","params":[],"id":0}' -H
"Content-Type: application/json" -X POST localhost:8541
```

Эта команда должна найти URL для подключения второго узла. Вывод команды будет выглядеть приблизительно так:

```
{"jsonrpc":"2.0","result":"enode://7bac3c8cf914903904a408ecd71635966331990c5c9f7c7a
291b531d5912ac3b52e8b174994b93cab1bf14118c2f24a16f75c49e83b93e0864eb099996ec1af9@
[::0.0.1.0]:30301","id":0}
```

Далее выполните следующую команду, заменив URL-адрес и IP-адрес `enode` на 127.0.0.1:

```
curl --data '{"jsonrpc":"2.0","method":"parity_addReservedPeer","params":["enode:
//7ba..."],"id":0}' -H "Content-Type: application/json" -X POST localhost:8540
```

Вы должны получить следующий ответ:

```
{"jsonrpc":"2.0","result":true,"id":0}
```

В консоли узлы должны выводить сообщение `0/1/25 peers` (рис. 9.1), которое означает, что узлы соединились друг с другом.

```
2017-04-19 00:29:59  Imported #868 bc6f…dfa8 (0 txs, 0.00 Mgas, 0.57 ms, 0.56 KiB)
2017-04-19 00:30:04  Imported #869 080b…7964 (0 txs, 0.00 Mgas, 0.60 ms, 0.56 KiB)
2017-04-19 00:30:10  Imported #870 552c…4fd8 (0 txs, 0.00 Mgas, 0.51 ms, 0.56 KiB)
2017-04-19 00:30:15  Imported #871 2fed…27d4 (0 txs, 0.00 Mgas, 0.58 ms, 0.56 KiB)
2017-04-19 00:30:17     0/ 1/25 peers   309 KiB db  302 KiB chain  0 bytes queue   17 KiB
2017-04-19 00:30:19  Imported #872 834c…9d78 (0 txs, 0.00 Mgas, 0.49 ms, 0.56 KiB)
2017-04-19 00:30:25  Imported #873 62ee…6335 (0 txs, 0.00 Mgas, 0.48 ms, 0.56 KiB)
2017-04-19 00:30:29  Imported #874 8043…7a5d (0 txs, 0.00 Mgas, 0.51 ms, 0.56 KiB)
2017-04-19 00:30:35  Imported #875 9b7d…a9c9 (0 txs, 0.00 Mgas, 0.46 ms, 0.56 KiB)
2017-04-19 00:30:40  Imported #876 493b…9cc6 (0 txs, 0.00 Mgas, 0.65 ms, 0.56 KiB)
2017-04-19 00:30:45  Imported #877 a672…f06f (0 txs, 0.00 Mgas, 0.53 ms, 0.56 KiB)
2017-04-19 00:30:47     0/ 1/25 peers   311 KiB db  302 KiB chain  0 bytes queue   17 KiB
2017-04-19 00:30:49  Imported #878 cedf…1ee5 (0 txs, 0.00 Mgas, 0.47 ms, 0.56 KiB)
2017-04-19 00:30:55  Imported #879 4381…8fcc (0 txs, 0.00 Mgas, 0.58 ms, 0.56 KiB)
2017-04-19 00:30:59  Imported #880 b383…ef90 (0 txs, 0.00 Mgas, 0.53 ms, 0.56 KiB)
2017-04-19 00:31:05  Imported #881 25cf…aeeb (0 txs, 0.00 Mgas, 0.46 ms, 0.56 KiB)
2017-04-19 00:31:10  Imported #882 8dee…ca2c (0 txs, 0.00 Mgas, 0.53 ms, 0.56 KiB)
2017-04-19 00:31:15  Imported #883 770a…f85b (0 txs, 0.00 Mgas, 0.53 ms, 0.56 KiB)
```

Рис. 9.1. Рабочая консоль узла после подключения

Полномочия и приватность

Мы показали, как Parity решает проблемы скорости и безопасности. В настоящее время Parity не имеет никаких особенностей в плане приватности и полномочий.

- **Полномочия** — Parity может реализовать механизм разграничения полномочий, чтобы решать, кто может или не может подключиться, путем конфигурирования

сервера каждого узла и разрешения подключений только с заданных IP-адресов. Даже если IP-адрес не заблокирован, для подключения к узлу сети новый узел должен знать адрес `enode`, как было показано ранее (и это вполне очевидно). По сути, это элементы базовой защиты, но от них мало пользы. Каждый узел сети должен самостоятельно обеспечивать безопасность. Разрешения для тех, кто имеет право создавать блоки, могут быть определены через смарт-контракт. Наконец, на сегодняшний день вообще не предусмотрена настройка того, какие типы транзакций имеет право отправлять конкретный узел.

- **Защита личной информации** — существует механизм обеспечения защиты личных данных через постоянную проверку права доступа. Во время установления права принадлежности (начальной идентификации) владелец должен задать открытый ключ недетерминированной асимметричной криптографии. Всякий раз, когда надо подтвердить доступ к личным данным, он предоставляет зашифрованный текст, который расшифровывается контрактом. Успешная расшифровка подтверждает доступ к данным. Контракт должен следить за тем, чтобы одни и те же зашифрованные данные не проверялись дважды.
- **Защита данных** — если вы используете блокчейн только для хранения данных, то можете применять симметричную криптографию для шифрования данных и выдавать ключ тем, кому разрешен доступ к данным. Но какие-либо операции над зашифрованными данными невозможны. И если вам необходимо выполнять операции над поступающими данными, но при этом сохранять конфиденциальность, придется создать для этого полностью независимый блокчейн.

Заключение

На протяжении этой главы мы изучали, как использовать Parity, как работает Aura, а также некоторые механизмы обеспечения конфиденциальности и безопасности. Теперь вы достаточно подготовлены для реализации, как минимум, протокола Proof-of-Concept в блокчейне корпоративного уровня. Вы можете работать над собой далее и перейти к изучению других решений, таких как Hyperledger или Quorum. В настоящее время разработчики Ethereum трудятся над более полной официальной поддержкой консорциумов. Поэтому вы должны внимательно следить за всеми новостями в области блокчейна и не пропустить новинки на рынке новых технологий.

ПРИЛОЖЕНИЕ

Описание электронного файлового архива

Электронный архив с материалами к этой книге можно скачать с FTP-сервера издательства «БХВ-Петербург» по ссылке **ftp://ftp.bhv.ru/9785977539760.zip** или со страницы книги на сайте **www.bhv.ru**.

Электронный архив состоит из двух частей:

- в папке `OriginalPaktCode` находятся файлы упражнений к *главам 4–9*. Каждое упражнение разбито на два подкаталога с названиями `Initial` и `Final`. В папке `Initial` расположены пустые файлы-заготовки, которые читатель может заполнить самостоятельно. На случай, если у читателя недостаточно квалификации для самостоятельного написания кода или проект работает неправильно, в папке `Final` находятся полностью готовые и проверенные автором книги файлы проекта соответствующей главы;
- в корневом каталоге архива помещены файлы листингов примеров из текста книги.

Перечень файлов (папок) приведен в табл. П.1.

Таблица П.1. *Содержание файлового архива книги*

Имя файла (папки)	Описание файла (папки)
`OriginalPactCode`	Папка с примерами упражнений к *главам 4–9*
`listing_3.1.sol`	Пример смарт-контракта
`listing_3.2.sol`	Пример использования массивов
`listing_3.3.sol`	Пример синтаксиса при использовании строк
`listing_3.4.sol`	Пример синтаксиса при использовании структур
`listing_3.5.sol`	Пример синтаксиса при использовании перечислений
`listing_3.6.sol`	Пример создания и использования сопоставления
`listing_3.7.sol`	Пример использования оператора `delete`

Таблица П.1 *(окончание)*

Имя файла (папки)	Описание файла (папки)
listing_3.8.sol	Примеры управляющих структур
listing_3.9.sol	Пример вызова внешней функции
listing_3.10.sol	Пример различной видимости и функций доступа
listing_3.11.sol	Пример использования модификаторов функций
listing_3.12.sol	Пример наследования контрактов
listing_3.13.sol	Пример использования ключевого слова `super`
listing_3.14.sol	Пример использования конструкции `using … for …`
listing_3.15.sol	Пример контракта, доказывающего владение файлом
listing_3.16.sol	Код для отправки транзакции с данными файла
listing_4.1.js	Прослушивание событий контракта
listing_4.2.html	HTML-код клиентского интерфейса
listing_4.3.js	Содержимое файла `main.js`
listing_5.1.html	Содержимое файла `index.html`
listing_5.2.js	Реализация метода `generate_addresses()`
listing_5.3.js	Реализация метода `send_ether()`
listing_6.1.js	Демонстрация методов для работы с версиями компилятора
listing_6.2.js	Код обработки нажатия кнопки **Deploy** и развертывания контракта
listing_6.3.html	HTML-код клиентской части приложения для файла `index.html`
listing_6.4.js	Код JavaScript для файла `main.js`
listing_7.1.sol	Контракт для заключения пари на результат матча
listing_7.2.ejs	Полный код файла визуализации `matches.ejs`
listing_7.3.html	Исходный код файла `index.html`
listing_7.4.js	Исходный код файла `main.js`
listing_8.1.sol	Исходный код контракта для изучения абстракции
listing_8.2.html	Исходный код примера абстракции контракта
listing_8.3.sol	Развертывание контракта и получение его экземпляра
listing_8.4.sol	Исходный код контракта MetaCoin
listing_8.5.js	Исходный код файла теста `metacoin.js`
listing_8.6.sol	Исходный код файла теста `TestMetacoin.sol`
listing_8.7.html	Код файла `index.html`
listing_8.8.js	Код файла `index.js`
listing_9.1.sol	Базовый пример рапортующего контракта
listing_9.2.json	Исходный код файла `spec.json`

Предметный указатель

A

ABI, Application Binary Interface 149
Aura 241
AWS, Amazon Web Services 166

B

BigChainDB 40
BIP 125
Bitcoin 32
Bootstrap node *См.* Загрузочный узел

C

Casper 62, 64
Codemirror 150

D

Dash 38
DHT, Distributed Hash Table 35

E

ECC, Elliptic Curve Criptography 45
ECDSA, Elliptic Curve Digital Signature Algorithm 46, 124
Error-first, формат вызова 99
Ethereum 34, 45
Ethereum Wallet 59
EVM *См.* Виртуальная машина Ethereum

F

Fabric 35
Filecoin 36
Fork 51
◊ hard fork 51
◊ regular fork 51
◊ soft fork 51

G

Gas *См.* Газ
Genesis 52
Geth 55
GHOST, Greedy Heaviest Observed SubTree 50

H

HD-Wallet 124
Hyperledger 35

I

IPFS 35

J

JSON-RPC 97

K

Kademlia 54
KDF, Key Derivation Function 125

L

LightWallet 126

M

MAC, Message Authetication Code 165
Mist 61
Mocha 219

N

Namecoin 37
Nephew block 51
Nonce 47

O

OpenBazaar 40
Oraclize 162

P

Parity 239
Parsing helper 170
Proof-of-Authority 240

Q

Quorum 239

R

Ripple 40
◊ валидатор 42
◊ доверительный список 41
◊ транзакция 41
◊ цепочка доверия 41
◊ шлюз 41
Rust 241
◊ установка 243

S

Serenity 62
Sharding *См.* Разделение данных
SOLC 145
SOLCJS 145
Solidity 45, 65
◊ исключения 77
◊ массивы 69
◊ оператор delete 73
◊ перечисление 72
◊ преобразование типов 74
◊ расположение данных 67
◊ создание контракта 76
◊ сопоставление 72
◊ строки 70
◊ структура контракта 66
◊ структуры 71
◊ типы данных 68
◊ файлы исходного кода 65
Swarm 55

T

Tendermint 241
testRPC 197
TLSNotary 164
Truffle 194
◊ truffle-contract, API 203
◊ truffle-contract, пакет 201
◊ внешние команды 230
◊ инициализация 212
◊ компиляция контракта 214
◊ консоль 228
◊ конструктор по умолчанию 231
◊ сервер 236
◊ установка 212
◊ файлы конфигурации 214

U

Uncle block 50

W

Whisper 55
WolframAlpha 163

Z

Zooko triangle *См.* Треугольник Зуко

А

Абстракция контракта 203
Альткоин 194
Арбитр 40
Атака 51% 62
◊ Сибиллы 62

Б

Библиотеки 86
Биткойн *См.* Bitcoin
Блокчейн 33

В

Валидатор 240
Ветвление 51
Виртуальная машина Ethereum 53
Время блока 49

Г

Газ 53
Генезис *См.* Genesis

Д

Деноминация 52
Децентрализованная автономная организация 28
Доказательство полномочий *См.* Proof-of-Authority

З

Заголовок события 199
Загрузочный узел 54

И

Индексированные параметры событий 106

К

Канал состояния 63
Конвертация денежных единиц 101
Консенсус 46
Консорциум 239
Контракт
◊ абстрактные контракты 86
◊ резервная функция 83
◊ свойства контракта 79

М

Майнер 46
Майнинг 46, 58
Метка времени 47
Модификатор функции 81

Н

Накопитель транзакций 144

П

Платежный канал 63
Поузловой тест *См.* Юнит-тест
Прикладной провайдер 122
Приложение
◊ внутренняя валюта 31
◊ децентрализованное 25
◊ распределенное 26
◊ централизованное 25
Пространство имен 126
Протокол
◊ доказательства работы 26
◊ доказательства владения долей 33
◊ доказательства залога 40
◊ доказательства уничтожения 40
◊ консенсуса 26
◊ подтверждения услуги 40
Путь вывода 127

Р

Разделение данных 64
Рапортующий контракт 244
Реестр 33

С

Сертификат, цифровой 28
Сигнатура события 199
Синхронизация 59
Сложность 47
Средства разбора строк 170
Срез 172

Т

Транзакция 45
◊ в обработке 103
◊ отложенная 103
Треугольник Зуко 38

У

Удостоверение личности 28
Умный контракт 34
Управление пакетами 225
Устаревший блок 49
Учетная запись 30

Ф

Файлы переноса 216

Ц

Целевое число 47

Ч

Частная сеть 244

Э

Экземпляр контракта 209
◊ API экземпляра 211
Эфир 45
Эфириум *См.* Ethereum

Ю

Юнит-тест 218