СОЗДАНИЕ СМАРТ-КОНТРАКТОВ
SOLIDITY ДЛЯ БЛОКЧЕЙНА

ETHEREUM

ПРАКТИЧЕСКОЕ РУКОВОДСТВО

Александр Фролов

12+

Александр Фролов

Создание смарт-контрактов Solidity для блокчейна Ethereum. Практическое руководство

«ЛитРес: Самиздат»

2019

Фролов А. В.

Создание смарт-контрактов Solidity для блокчейна Ethereum. Практическое руководство / А. В. Фролов — «ЛитРес: Самиздат», 2019

Эта книга поможет быстро приступить к созданию смарт-контактов Solidity и распределенных приложений DApp для блокчейна Ethereum. Она состоит из 12 уроков с практическими заданиями. Выполнив их, читатель сможет создавать собственные локальные узлы Ethereum, публиковать смарт-контракты и вызывать их методы, обмениваться данными между реальным миром и смарт-контрактами с помощью оракулов, работать с сетью Rinkeby. Книга адресована всем, кто интересуется передовыми технологиями в области блокчейнов и хочет быстро получить знания, позволяющие заниматься интересной и перспективной работой.

Содержание

Введение

Наша книга предназначена для тех, кто хочет не только понять принципы работы блокчейна Ethereum, но и получить практические навыки в создании распределенных приложений DApp на языке программирования Solidity для этой сети.

Эту книгу лучше не просто читать, а работать с ней, выполняя практические задания, описанные в уроках. Для работы вам потребуются локальный компьютер, виртуальный или облачный сервер с установленной ОС Debian или Ubuntu. Также для выполнения многих заданий можно использовать Raspberry Pi.

На первом уроке мы рассмотрим принципы работы блокчейна Ethereum и основную терминологию, а также расскажем о том, где можно использовать этот блокчейн.

Цель **второго урока** – создать узел приватного блокчейна Ethereum для дальнейшей работы в рамках этого курса на сервере Ubuntu и Debian. Мы рассмотрим особенности установки основных утилит, таких как geth, обеспечивающего работу узла нашего блокчейна, а также демона децентрализованного хранилища данных swarm.

Третий урок научит вас проводить эксперименты с Ethereum на недорогом микрокомпьютере Raspberry Pi. Вы установите операционную систему (ОС) Rasberian на Raspberry Pi, утилиту Geth, обеспечивающую работу узла блокчейна, а также демон децентрализованного хранилища данных Swarm.

Четвертый урок посвящен аккаунтам и криптовалютным единицам в сети Ethereum, а также способам перевода средств с одного аккаунта на другой из консоли Geth. Вы научитесь создавать аккаунты, инициировать транзакции перевода средств, получать состояние транзакции и ее квитанцию.

На пятом уроке вы познакомитесь со смарт-контрактами в сети Ethereum, узнаете об их выполнении виртуальной машиной Ethereum.

Вы создадите и опубликуете в приватной сети Ethereum свой первый смарт-контракт и научитесь вызывать его функции. Для этого вы будете использовать среду разработки Remix Solidity IDE. Кроме того, вы научитесь устанавливать и использовать пакетный компилятор solc.

Также мы расскажем о так называемом бинарном интерфейсе приложения Application Binary Interface (ABI) и научим его использовать.

Шестой урок посвящен созданию скриптов JavaScript, работающих под управлением Node.js и выполняющих операции со смарт-контрактами Solidity.

Вы установите Node.js в ОС Ubuntu, Debian и Rasberian, напишете скрипты для публикации смарт-контракта в локальной сети Ethereum и вызова его функций.

Кроме того, вы научитесь переводить с помощью скриптов средства между обычными аккаунтами, а также зачислять их на аккаунты смарт-контрактов.

На седьмом уроке вы научитесь устанавливать и использовать популярную среди разработчиков смарт-контрактов Solidity интегрированную среду Truffle. Вы научитесь создавать скрипты JavaScript, вызывающие функции контрактов с помощью модуля truffle-contract, а также выполните тестирование своего смарт-контракта средствами Truffle.

Восьмой урок посвящен типам данных Solidity. Вы напишете смарт-контракты, работающие с такими типами данных, как знаковые и беззнаковые целые числа, числа со знаком, строки, адреса, переменные сложных типов, массивы, перечисления, структуры и словари.

На девятом уроке вы приблизитесь еще на шаг к созданию смарт-контрактов для основной сети Ethereum. Вы научитесь публиковать контракты при помощи Truffle в приватной сети Geth, а также в тестовой сети Rinkeby. Отладка смарт-контракта в сети Rinkeby очень полезна перед его публикацией в основной сети – там практически все по-настоящему, но бесплатно.

В рамках урока вы создадите узел тестовой сети Rinkeby, пополните его средствами и опубликуете смарт-контракт.

Урок 10 посвящен распределенным хранилищам данных Ethereum Swarm. Используя распределенные хранилища, вы экономите на хранении данных большого объема в блокчейне Ethereum.

В рамках этого урока вы создадите локальное хранилище Swarm, выполните операции записи и чтения файлов, а также каталогов с файлами. Далее вы научитесь работать с публичным шлюзом Swarm, напишите скрипты для обращения к Swarm из Node.js, а также с помощью модуля Perl Net::Ethereum::Swarm.

Цель урока 11 – освоить работу со смарт-контрактами Solidity с применением популярного языка программирования Python и фреймворка Web3.py. Вы установите этот фреймворк, напишите скрипты для компиляции и публикации смарт-контракта, а также для вызова его функций. При этом Web3.py будет использован как сам по себе, так и совместно с интегрированной средой разработки Truffle.

На 12 уроке вы научитесь передавать данные между смарт-контрактами и реальным миром при помощи оракулов. Это пригодится вам для получения данных с Web-сайтов, устройств интернета вещей IoT, различных приборов и датчиков, и отправки данных из смарт-контрактов на эти устройства. В практической части урока вы создадите оракул и смарт-контракт, получающий актуальный курс обмена USD на рубли с сайта ЦБ РФ.

Обратная связь

Вы можете связаться с автором этой книги по адресам sbook@frolov-lib.ru или https://www.facebook.com/frolov.shop2you.

Пожалуйста, отправляйте ваши замечания и рекомендации, автор постарается их учесть при переиздании книги.

Урок 1. Кратко о блокчейне и сети Ethereum

Цель урока: познакомиться с принципами работы блокчейна Ethereum, областями его применения и основной терминологией.

Практические задания: в этом уроке не предусмотрены.

Едва ли сегодня есть разработчик программного обеспечения (ПО), который бы ничего не слышал о технологии блокчейн (Blockchain), криптовалютах (Cryptocurrency или Crypto Currency), биткоинах (Bitcoin), первичном размещении монет (ICO, Initial coin offering), смарт-контрактах (Smart Contract), а также других понятиях и терминах, имеющих отношение к блокчейну.

Технология блокчейна открывает новые рынки и создает рабочие места для программистов. Если вы постигнете все тонкости криптовалютных технологий и технологий смарт-контрактов, то у вас не должно быть проблем с применением этих знаний на практике.

Надо сказать, что вокруг криптовалют и блокчейнов возникает много спекуляций. Мы оставим в стороне рассуждения об изменениях курсов криптовалют, о создании пирамид, о тонкостях криптовалютного законодательства и т.п. В нашем учебном курсе мы сосредоточимся главным образом на технических аспектах применения смарт-контрактов блокчейна Ethereum (эфириум, эфир) и разработки так называемых децентрализованных приложений (Distributed Application, DApp).

Что такое блокчейн

Блокчейн (Blockchain, Block Chain) представляет собой цепочку блоков данных, связанных друг с другом определенным образом. В начале цепочки находится первый блок, который называется первичным блоком (genesis block) или блоком генезиса. За ним следует второй, потом третий и так далее.

Все эти блоки данных автоматически дублируются на многочисленных узлах сети блокчейна. Таким образом обеспечивается децентрализованное хранение данных блокчейна.

Вы можете представить себе систему блокчейна как большое количество узлов (физических или виртуальных серверов), объединенных в сеть и реплицирующих все изменения в цепочке блоков данных. Это как бы гигантский мультисерверный компьютер, причем узлы такого компьютера (серверы) могут быть разбросаны по всему миру. И вы тоже можете добавить свой компьютер в сеть блокчейна.

Распределенная база данных

Блокчейн можно представить себе как распределенную базу данных, реплицируемую на все узлы сети блокчейна. В теории блокчейн будет работоспособен до тех пор, пока работает хотя бы один узел, хранящий все блоки блокчейна.

Распределенный реестр данных

Блокчейн можно представить себе как распределенный реестр данных и операций (транзакций). Еще одно название такого реестра – гроссбух.

В распределенный реестр можно добавлять данные, но невозможно их изменять или удалять. Такая невозможность достигается, в частности, применением криптографических алгоритмов, специальных алгоритмов добавления блоков в цепочку и децентрализованным хранением данных.

При добавлении блоков и выполнении операций (транзакций) используются приватные и публичные ключи. Они ограничивают пользователей блокчейна, предоставляя им доступ только к своим блокам данных.

Транзакции

Блокчейн хранит информацию об операциях (транзакциях) в блоках. При этом старые, уже выполненные транзакции невозможно откатить или изменить. Новые транзакции хранятся в новых, добавленных блоках.

Таким образом в блокчейне может быть записана в неизменном виде вся история транзакций. Поэтому блокчейн может быть использован, например, для надежного хранения банковских операций, сведений об авторском праве, истории изменений владельцев объектов недвижимости и т.п.

Блокчейн Ethereum содержит так называемые состояния системы. По мере выполнения транзакций состояние изменяется от начального до текущего. Транзакции записываются в блоки.

Публичные и приватные блокчейны

Тут нужно отметить, что все сказанное верно только для так называемых *публичных* сетей блокчейн, которые не могут контролироваться никакими отдельными физическими или юридическими лицами, государственными органами или правительствами.

Так называемые *приватные* сети блокчейн находятся под полным контролем их создателей, и там возможно все, например, полная замена всех блоков цепочки.

Практические применения блокчейна

Для чего может пригодиться блокчейн?

Если кратко, блокчейн позволяет безопасным образом проводить операции (сделки) между не доверяющими друг другу персонами или компаниями. Записанные в блокчейн данные (транзакции, персональные данные, документы, свидетельства, договоры, накладные и т.п.) невозможно подделать или заменить после записи. Поэтому на базе блокчейна можно создавать, например, доверенные распределенные реестры различного рода документов.

Конечно, вы знаете, что на базе блокчейнов создаются криптовалютные системы, призванные заменить обычные бумажные деньги. Бумажные деньги еще называют фиатными (от Fiat Money).

Блокчейн обеспечивает хранение и неизменность транзакций, записанных в блоки, поэтому его и можно использовать для создания криптовалютных систем. Он содержит всю историю передачи крипто-средств между различными пользователями (аккаунтами), причем любая операция может быть отслежена.

Хотя операции внутри криптовалютных систем могут быть анонимными, вывод криптовалюты и обмен ее на фиатные деньги обычно приводит к раскрытию личности владельца криптовалютного актива.

Так называемые смарт-контракты, представляющие собой программное обеспечение, работающее в сети Ethereum, позволяют автоматизировать процесс заключения сделок и контроль их выполнения. Особенно это эффективно, если оплата по сделке проводится криптовалютой Ether (эфир).

Блокчейн Ethereum и смарт-контракты Ethereum, написанные на языке программирования Solidity, могут использоваться, например, в таких областях:

- альтернатива нотариальному заверению документов;
- хранение реестра объектов недвижимости и сведений об операциях с объектами недвижимости;
- хранение сведений об авторских правах на объекты интеллектуальной собственности (книги, изображения, музыкальные произведения и т.п.);
- создание систем независимого голосования;
- сфера финансов и банковская деятельность;
- логистика в международных масштабах, отслеживание перемещения грузов;
- хранение персональных данных как аналог системе удостоверений личности;
- безопасные сделки в коммерческой области;
- хранение результатов медицинских обследований, а также истории назначенных процедур.

Проблемы с блокчейном

Но, разумеется, не все так просто, как могло бы показаться!

Есть проблемы с верификацией данных перед их добавлением в блокчейн (например, не поддельные ли они?), проблемы в безопасности системного и прикладного ПО, применяемого для работы с блокчейном, проблемы с возможностью использования методов социальной инженерии для похищения доступа к кошелькам с криптовалютой и т.п.

Опять же, если речь идет не о публичном блокчейне, узлы которого разбросаны по всему миру, а о приватном блокчейне, принадлежащем персоне или организации, то уровень доверия здесь будет не выше, чем уровень доверия к этой персоне или этой организации.

Также следует учитывать, что данные, записанные в блокчейн, становятся *доступны для всех*. В этом смысле блокчейн (особенно публичный) не подходит для хранения конфиденциальной информации. Однако тот факт, что информацию в блокчейне невозможно изменить, может помочь предотвратить или расследовать различного рода мошеннические действия.

Децентрализованные приложения Ethereum будут удобны, если платить за их использование криптовалютой. Чем больше людей, владеющих криптовалютой или готовых ее приобрести, тем большую популярность получат приложения DApp и смарт-контракты.

Среди общих проблем блокчейна, затрудняющих его практическое применение, можно упомянуть ограниченную скорость добавления новых блоков и относительно высокую стоимость транзакций. Но технологии в этой области активно развиваются, и есть надежды, что технические проблемы со временем будут решены.

Еще одна проблема заключается в том, что смарт-контракты блокчейна Ethereum работают в изолированной среде виртуальных машин и не имеют доступа к данным реального мира. В частности, программа смарт-контракта не может сама прочитать данные с сайтов или каких-либо физических устройств (датчики, контакты и т.п.), а также не может вывести данные на какие-либо внешние устройства. Эту проблему и способы ее решения мы будем обсуждать на уроке, посвященном так называемым Оракулам – информационным посредникам смарт-контрактов.

Также есть ограничения в области законодательства. В некоторых странах, например, запрещается использовать криптовалюту как платежное средство, но можно владеть ей как неким цифровым активом, наподобие ценных бумаг. Такие активы можно покупать и продавать на бирже. В любом случае, при создании проекта, работающего с криптовалютами, необходимо ознакомиться с законодательством той страны, под юрисдикцию которой попадает ваш проект.

Как формируется цепочка блокчейна

Как мы уже говорили, блокчейн представляет собой простую цепочку блоков данных. Сначала формируется первый блок этой цепочки, потом к нему добавляется второй и так далее. Предполагается, что данные транзакций хранятся в блоках и добавляются в самый последний блок.

На рис. 1.1. мы показали простейший вариант последовательности блоков, где первый блок ссылается на следующий.

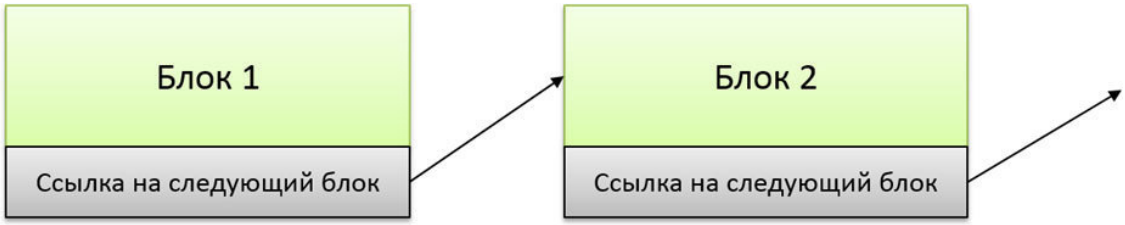

Рис. 1.1. Простая последовательность блоков

В таком варианте, однако, можно очень легко подделать содержимое любого блока цепочки, так как блоки не содержат никакой информации для защиты от изменений. Учитывая, что блокчейн предназначен для использований людей и компаний, между которыми нет доверия, можно сделать вывод, что такой способ хранения данных для блокчейна не подходит.

Давайте займемся защитой блоков от подделки. На первом этапе мы попробуем защитить каждый блок контрольной суммой (рис. 1.2.).

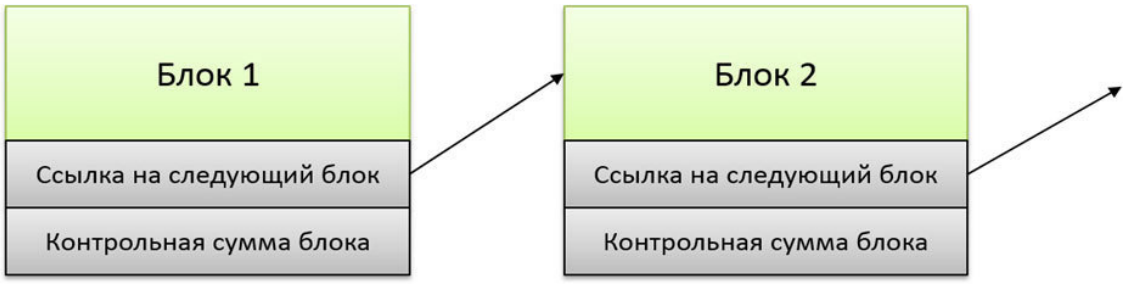

Рис. 1.2. Добавляем защиту данных блоков контрольной суммой

Теперь злоумышленник не может просто так изменить блок, так как в нем находится контрольная сумма данных блока. Проверка контрольной суммы покажет, что данные были изменены.

Для вычисления контрольной суммы можно использовать одну из функций хеширования, такую как MD-5, SHA-1, SHA-256 и т.п. Хеш-функции вычисляют некоторое значение (например, в виде текстовой строки постоянной длины) в результате выполнения необратимых операций над блоком данных. Операции зависят от вида хеш-функции.

Даже при небольшом изменении содержимого блока данных значение хеш-функции также изменится. Анализируя значение хеш-функции, невозможно восстановить блок данных, для которого она была вычислена.

Будет ли достаточна такая защита? К сожалению, нет.

В этой схеме контрольная сумма (хеш-функция) защищает только отдельные блоки, но не всю цепочку блоков. Зная алгоритм вычисления хеш-функции, злоумышленник может легко

подменить содержимое блока. Также ничто не помешает ему удалить блоки из цепочки или добавить новые.

Чтобы защитить всю цепочку в целом, можно хранить в каждом блоке вместе с данными еще и хеш данных предыдущего блока (рис. 1.3.).

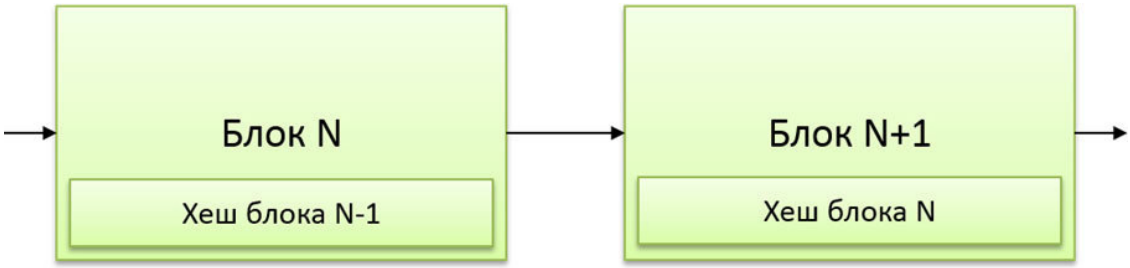

Рис. 1.3. Добавляем в блок данных хеш предыдущего блока

В этой схеме, чтобы изменить какой-нибудь блок, нужно пересчитать хеш-функции всех последующих блоков. Казалось бы, в чем проблема?

В реальных блокчейнах дополнительно созданы искусственные трудности для добавления новых блоков – используются алгоритмы, требующие много вычислительных ресурсов. С учетом того, что для внесения изменений в блок пересчитать нужно не один этот блок, а все последующие, сделать это будет крайне сложно.

Вспомним также, что данные блокчейна хранятся (дублируются) на многочисленных узлах сети, т.е. используется децентрализованное хранение. А это многократно усложняет подделку блока, т.к. изменения требуется внести на все узлы сети.

Так как блоки хранят информацию о предыдущем блоке, то можно проверить содержимое всех блоков цепочки.

Блокчейн Ethereum

Блокчейн Ethereum представляет собой платформу, на базе которой можно создавать распределенные приложения DApp. В отличие от других платформ, Ethereum позволяет использовать так называемые умные контракты (*смарт-контракты*, smart contracts), написанные на языке программирования Solidity.

Эта платформа была создана в 2013 году Виталиком Бутериным, основателем журнала Bitcoin Magazine, и запущена в 2015 году. Все, что мы будем изучать или делать в нашем учебном курсе, имеет отношение именно к блокчейну Ethereum и смарт-контрактам Solidity.

Майнинг, или Как создаются блоки

Майнинг (mining) представляет собой довольно сложный и ресурсоемкий процесс добавления новых блоков в цепочку блокчейна, а вовсе не «добычу криптовалют». Майнинг обеспечивает работоспособность блокчейна, т.к. именно этот процесс отвечает за добавление транзакций в блокчейн Ethereum.

Люди и организации, занимающиеся добавлением блоков, называются майнерами (miner).

Программное обеспечение (ПО), работающее на узлах майнеров, пытается подобрать для последнего блока параметр хеширования с названием Nonce, чтобы получилось определенное значение хеш-функции, заданной сетью. Алгоритм хеширования Ethash, применяемый в Ethereum, позволяет получить значение Nonce только путем последовательного перебора.

Если узел майнера нашел правильное значение Nonce, то это является так называемым доказательством работы (PoW, Proof-of-work). В этом случае, если блок будет добавлен в сеть Ethereum, майнер получает определенное вознаграждение в валюте сети – Ether. На момент подготовки нашей книги вознаграждение составляет 5 Ether, однако со временем оно будет снижено.

Таким образом, майнеры Ethereum обеспечивают работу сети, добавляя блоки, и получают за это криптовалютные деньги. В интернете вы найдете массу информации о майнерах и майнинге, а мы сосредоточимся на создании контрактов Solidity и децентрализованных приложений DApp в сети Ethereum.

Итоги урока

На первом уроке вы познакомились с блокчейном и узнали, что он представляет собой особым образом составленную последовательностью блоков. Содержимое ранее записанных блоков невозможно изменить, так как для этого придется пересчитать все последующие блоки на многих узлах сети, на что нужно очень много ресурсов и времени.

Блокчейн может применяться для сохранения результатов выполнения транзакций. Его главное назначение – организация безопасного выполнения транзакций между сторонами (персонами и организациями), между которыми нет доверия. Вы узнали, в каких конкретно областях бизнеса и в каких сферах можно использовать блокчейн Ethereum и смарт-контракты Solidity. Это банковская сфера, регистрация прав собственности, документов и т.п.

Вы также узнали, что при использовании блокчейна могут возникать различные проблемы. Это проблемы верификации информации, добавляемой в блокчейн, скорость работы блокчейна, стоимость транзакций, проблема обмена данными между смарт-контрактами и реальным миром, а также потенциально возможные атаки злоумышленников, направленные на похищение криптовалютных средств с аккаунтов пользователей.

Также мы кратко рассказали о майнинге как о процессе добавления новых блоков в блокчейн. Майнинг необходим для выполнения транзакций. Те, кто занимается майнингом, обеспечивают работоспособность блокчейна и получают за это вознаграждение в криптовалюте.

Урок 2. Подготовка рабочей среды в ОС Ubuntu и Debian

Цель урока: создать узел собственного приватного блокчейна Ethereum для дальнейшей работы в рамках этого курса на сервере Ubuntu и Debian.

Практические задания: установка операционной системы Ubuntu и Debian, установка ПО geth, обеспечивающего работу узла нашего блокчейна, а также демона децентрализованного хранилища данных swarm. Вам будет нужно создать собственный приватный блокчейн Ethereum, выполнить его инициализацию. Подключившись к узлу блокчейна, вы научитесь выдавать простейшие команды в приглашении geth.

Прежде чем мы займемся изучением смарт-контрактов Ethereum, нам необходимо подготовить рабочую среду – установить операционную систему (ОС) и необходимое программное обеспечение (ПО).

Мы могли бы приступить к работе сразу в какой-либо интегрированной среде разработки (IDE, Integrated Development Environment), например, Remix или Truffle. Возможно, это был бы самый быстрый путь к изучению смарт-контрактов Solidity. Однако для того, чтобы глубже разобраться в том, как работает Ethereum, мы начнем с базовых инструментов.

Выбор операционной системы

Свой первый узел сети Ethereum мы будем делать на базе клиента Go Ethereum (https://geth.ethereum.org/). Это ПО представляет собой реализацию протокола Ethereum и реализовано на языке программирования Go и доступно в виде программы Geth. На базе Geth можно создать полнофункциональный узел сети Ethereum.

Для работы с узлом Geth можно использовать интерфейс командной строки, а также программный интерфейс (API, Application Programming Interface) JSON RPC. Используя этот интерфейс и различные фреймворки, вы сможете создавать ПО, работающее с узлами Ethereum, практически на всех современных языках программирования.

Клиент Geth может работать на платформах, где имеется Go (это, например, Linux, Mac OSX, Windows, Raspberry Pi, Android OS, iOS). На странице загрузки https://geth.ethereum.org/downloads/ доступны реализации для Linux, macOS и Windows. Также вы можете загрузить исходные коды Geth.

При работе над книгой мы использовали ОС Ubuntu Live Server 18.04.2, Ubuntu 18.10 cosmic, Debian версий 9 и 10 Alfa 5, хотя Geth можно установить и на другие сборки Linux. На следующем уроке мы выполним установку Geth на ОС Rasberian для микрокомпьютера Raspberry Pi 3.

Для изучения Ethereum можно арендовать виртуальный или облачный сервер у одного из провайдеров. Также вы можете установить эти ОС на свой настольный компьютер, непосредственно на его диск или на виртуальную машину, например, VMware Workstation, или воспользоваться другой системой виртуализации.

Описание процесса установки Ubuntu и Debian выходит за рамки нашей книги, но в интернете есть немало достаточно подробных руководств, посвященных этому вопросу. Кроме того, при аренде виртуального или облачного сервера провайдер обязательно поможет установить на него ОС. Учтите, что вам нужна 64-разрядная версия ОС Ubuntu и Debian.

Для работы над книгой мы использовали облачный виртуальный сервер в следующей конфигурации:

- 4 ядра CPU с тактовой частотой 2 GHz;
- 2 GB RAM;
- 20 GB Disk SSD.

Мы также установили ОС Ubuntu 18.10 cosmic в виртуальную машину VMware Workstation с такой конфигурацией:

- 8 ядер CPU с тактовой частотой 2 GHz;
- 8 GB RAM;
- 100 GB Disk SATA.

Конфигурация виртуального сервера влияет на скорость майнинга новых блоков. Чтобы нам было удобно работать, эта скорость не должна быть слишком низкой. Не рекомендуем, в частности, использовать менее 2 Гбайт оперативной памяти.

Для микрокомпьютеров Raspberri Pi, где памяти меньше, есть решение, о котором мы расскажем на следующем уроке.

Сразу после установки ОС Ubuntu или Debian обновите пакеты при помощи команды apt-get:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

Если в Ubuntu Server обновилось ядро, может потребоваться перезагрузка ОС. Сообщение об этом вы увидите при подключении к консоли:

```
*** System restart required ***
```

В этом случае перед продолжением работы выполните перезагрузку:

```
$ sudo shutdown -r now
```

Пакеты Ubuntu можно также обновлять также через менеджер обновлений в графическом интерфейсе.

Обновление пакетов нужно делать с правами пользователя root.

Установка необходимых утилит

В ОС Ubuntu установите сервис ssh, если вы планируете подключаться к консоли удаленно (по умолчанию в десктопной версии сервис ssh не устанавливается, при установке Ubuntu Live Server нужно отметить соответствующий флажок).

В качестве имени пользователя при начальной установке ОС укажите book, чтобы у этого пользователя сразу была возможность работать с командой sudo.

Вы также можете создать пользователя book уже после установки Ubuntu. В этом случае при помощи команды visudo добавьте этому пользователю возможность работать с командой sudo. Для этого запустите с правами пользователя root такую команду:

```
# visudo
```

Откроется редактор файла /etc/sudoers. Вам нужно добавить в конец этого файла следующую строку:

```
book ALL=(ALL) ALL
```

Для установки сервиса ssh введите следующую команду:

```
$ sudo apt-get install ssh
```

Далее в Ubuntu и Debian установите редактор vim (если вам удобно в нем работать), утилиты sudo (если она не установлена), git, curl, gcc и mc (mc устанавливать не обязательно, пригодится, если только вы привыкли работать с Midnight Commander):

```
$ sudo apt-get install vim sudo git curl gcc mc
```

Для того чтобы обезопасить ваш сервер от атак типа брутфорса (перебор паролей) на порт SSH, установите fail2ban:

```
$ sudo apt-get install fail2ban
```

Мы настоятельно рекомендуем использовать эту утилиту в рабочем окружении вместе с брандмауэром.

Установка Geth и Swarm в Ubuntu

Далее мы перейдем к установке Geth, а также ПО узла распределенного хранилища данных Swarm (потребуется позже, на 10 уроке).

Проще всего установить Geth в ОС Ubuntu. Процедура описана здесь: https://github.com/ethereum/go-ethereum/wiki/Installation-Instructions-for-Ubuntu.

Для установки выполните следующие команды:

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

Вы также можете установить девелоперскую (нестабильную версию Geth), для чего выполните такую команду:

```
$ sudo apt-get install ethereum-unstable
```

После установки проверьте версию Geth:

```
$ geth version
Geth
Version: 1.8.23-stable
Git Commit: c942700427557e3ff6de3aaf6b916e2f056c1ec2
Architecture: amd64
Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.10.4
Operating System: linux
GOPATH=
GOROOT=/usr/lib/go-1.10
```

Как видите, здесь мы установили Geth стабильной версии 1.8.23 и Go версии 1.10.4.

Для установки распределенного хранилища данных Swarm на локальный тестовый узел используйте следующую команду:

```
$ sudo apt-get install ethereum-swarm
```

После установки проверьте версию Swarm:

```
$ swarm version
Swarm
Version: 0.3.11-stable
Git Commit: c942700427557e3ff6de3aaf6b916e2f056c1ec2
Go Version: go1.10.4
OS: linux
```

Если установка прошла успешно, переходите к разделу урока, посвященного созданию приватного блокчейна.

В том случае, когда при установке произошли ошибки, попробуйте найти решение в поисковой системе Google. Заметим, что ошибки часто связаны с обновлением версий устанавливаемого ПО.

Установка Geth и Swarm в Debian

Установку Geth и Swarm в ОС Debian нужно выполнять из исходных текстов. При этом вначале нужно будет установить Go, а затем уже собственно Geth и Swarm.

На момент создания нашей книги была доступна версия Go 1.12.1. Заметим, что Geth и Swarm находятся в состоянии постоянного совершенствования. Не исключено, что к моменту, когда вы начнете работу над этой книгой, для них придется устанавливать новую версию Go.

Предварительная подготовка

Прежде всего обновите пакеты и установите необходимые утилиты:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ apt-get install vim sudo git curl gcc mc
```

Если вы при установке ОС не добавили пользователю book возможность работы с командой sudo, сделайте это аналогично тому, как это было описано ранее для Ubuntu.

После этого можно переходить к установке Go и Geth.

Загрузка дистрибутива Go

Дистрибутивы Go различных версий и для различных платформ можно найти здесь: https://golang.org/dl/.

Прежде всего подключимся к нашему серверу (физическому или виртуальному) пользователем book и загрузим архив Go нужной версии:

```
$ curl -O https://dl.google.com/go/go1.12.1.linux-amd64.tar.gz
```

Теперь, находясь в консоли с правами пользователя book, распаковываем загруженный архив в каталог /usr/local:

```
$ sudo tar -C /usr/local -xzf go1.12.1.linux-amd64.tar.gz
```

У вас будет запрошен пароль пользователя book. Команда sudo необходима, так как обычному пользователю запрещена запись файлов в каталог /usr/local.

Установка переменных окружения

Далее мы создаем в домашнем каталоге пользователя book каталог go и устанавливаем переменные окружения:

```
$ mkdir -p ~/go; echo "export GOPATH=$HOME/go" >> ~/.bashrc
$ echo "export PATH=$PATH:$HOME/go/bin:/usr/local/go/bin" >>
~/.bashrc
$ source ~/.bashrc
```

Проверяем, что переменные окружения установлены:

```
$ printenv | grep go

GOPATH=/root/go
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin:/root/go/bin:/usr/local/go/bin
```

Проверка версии Go

Прежде чем перейти собственно к установке Geth и Swarm, нужно проверить версию go:

```
$ go version
go version go1.12.1 linux/amd64
```

Если у вас версия 1.12.1, то все нормально. Но если ранее по каким-то причинам на вашем сервере была установлена старая версия go из репозитория, удаляем ее так:

```
sudo apt-get remove golang-go
sudo apt-get remove –auto-remove golang-go
```

Установка Geth и Swarm

Первым шагом загрузите исходный код Geth из репозитория на GitHub:

```
$ mkdir -p $GOPATH/src/github.com/ethereum
$ cd $GOPATH/src/github.com/ethereum
$ git clone https://github.com/ethereum/go-ethereum
$ cd go-ethereum
$ git checkout master

$ go get github.com/ethereum/go-ethereum
```

Далее запустите компиляцию клиента Geth и Swarm:

```
go install -v ./cmd/geth
go install -v ./cmd/swarm
```

Если при компиляции появились ошибки, попробуйте установить Go другой версии. Перед этим удалите все каталоги, созданные в процессе предыдущей установки.

Если же все хорошо, то осталось только проверить версию установленной Geth и Swarm:

```
$ geth version
Geth
Version: 1.9.0-unstable
Architecture: amd64
Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.12.1
Operating System: linux
```

```
GOPATH=/home/book/go
GOROOT=/usr/local/go

$ swarm version
Swarm
Version: 0.3.12-unstable
Go Version: go1.12.1
OS: linux
```

Как видите, были установлены нестабильные версии Geth и Swarm. С помощью whereis вы можете определить, в какой каталог была выполнена установка:

```
$ whereis geth
geth: /home/book/go/bin/geth
```

Чтобы установить стабильную версию, загрузите ее бинарный код с сайта https://geth.ethereum.org/downloads/. Затем извлеките из архива программу geth и скопируйте в отдельный каталог.

Актуальную инструкцию по установке Geth и Swarm можно найти по адресу https://media.readthedocs.org/pdf/swarm-guide/latest/swarm-guide.pdf.

Создаем приватный блокчейн

Для того чтобы быстро освоить процесс создания смарт-контрактов, нам нужен блокчейн. Вначале создадим приватный блокчейн на своем сервере, так как с ним проще всего работать. Далее мы будем использовать тестовую сеть Rinkeby, работающую в точности как Ethereum и пригодную для отладки «настоящих» контрактов перед публикацией в Ethereum.

Готовим файл genesis.json

Прежде всего создайте в домашнем каталоге пользователя book файл genesis.json (листинг 2.1.).

Листинг 2.1. Файл genesis.json

```
{
 "config": {
    "chainId": 1999,
    "homesteadBlock": 0,
    "eip155Block": 0,
    "eip158Block": 0
 },
 "difficulty": "10",
 "gasLimit": "5100000",
 "alloc": {}
}
```

Этот файл описывает первичный блок (genesis block) цепочки блоков – самый первый блок блокчейна. Так как мы создаем наш приватный блокчейн, то и о первом блоке мы должны позаботиться сами.

Хорошее описание первичного блока сети Ethereum можно найти, например, здесь: https://arvanaghi.com/blog/explaining-the-genesis-block-in-ethereum/.

Блок **config** файла genesis.json содержит переменные конфигурации сети Ethereum.

Поле **chainId** содержит идентификатор блокчейна. Он используется для защиты от репликации, т.е. от неавторизованных действий пользователей, пытающихся действовать от имени настоящего отправителя данных.

Если речь идет о главной сети Ethereum, то ее идентификатор равен 1. Идентификатор тестовой сети Rinkeby равен 4. При создании нашей приватной сети мы можем указать любое значение, отличное от известных. Мы выбрали значение 1999.

Значение 0 в поле **homesteadBlock** указывает на то, что мы будем использовать релиз сети Ethereum под названием Homestead. Homestead представляет собой второй релиз сети Ethereum (первый релиз назывался Frontier). Нулевое значение этого параметра используется и в основной сети Ethereum.

Немного о параметрах с префиксом EIP. При реализации Homestead был обновлен протокол, причем без обратной совместимости с предыдущим релизом Ethereum. Соответствующие изменения отражены в документах «Предложения по улучшению Ethereum» (Ethereum Improvement Proposals, EIPs), опубликованных на сайте https://eips.ethereum.org/.

Реализация изменений может потребовать выполнения так называемой процедуры хардфорка (hard-forking), или обновления ПО узлов. В результате этой процедуры может меняться

структура блока, появляется возможность использовать ранее недоступные блоки и вносить различные изменения в протокол.

Так как мы не будем делать хардфорк, то установим значения соответствующих параметров **eip155Block** и **eip158Block** равными нулю.

Параметр **difficulty** важен для нас в практическом смысле. Этот параметр имеет прямое влияние на время генерации новых блоков блокчейна. Для нашего «учебного» блокчейна мы установим очень маленькое значение этого параметра, равное 10, чтобы скорость добавления новых блоков была приемлемой даже на виртуальных серверах небольшой производительности.

С помощью параметра **gasLimit** мы задаем в рамках блокчейна предел расхода так называемого газа (Gas). Газ Ethereum представляет собой расходный ресурс, который тратится на выполнение таких операций, как отправка транзакций, публикация и выполнение контрактов и т.п. Далее мы расскажем об этом подробнее. В нашей приватной тестовой сети мы устанавливаем достаточно большое значение, чтобы не возникали ограничения при запуске тестовых программ.

Параметр **alloc** позволяет при инициализации блокчейна создать кошельки и заранее наполнить их тестовым эфиром. В первых примерах мы не будем использовать эту возможность.

Создаем каталог для работы

Создайте в домашнем каталоге подкаталог node1:

```
$ mkdir node1
```

В этом каталоге будут располагаться данные блокчейна.

Создаем аккаунт

Теперь перейдем к созданию нашего приватного блокчейна. Прежде всего войдите в домашний каталог пользователя book и создайте новый аккаунт:

```
$ geth –datadir node1 account new
```

При создании аккаунта будет запрошен пароль, который нужно сохранить в безопасном месте:

```
    INFO         [02-13|08:42:28.798]         Maximum         peer
count                        ETH=25 LES=0 total=25
    Your new account is locked with a password. Please give a
password. Do not forget this password.
    Passphrase:
    Repeat passphrase:
    Address: { 4f744742ac711fd111c7a983176db1d48d29f413 }
```

Команда **account new** выведет на консоль в фигурных скобках так называемый адрес узла. В нашем случае это адрес 4f744742ac711fd111c7a983176db1d48d29f413. Мы будем указывать его в различных командах.

Параметр **datadir** команды geth указывает путь к рабочему каталогу. Мы используем каталог /home/book/node1.

Запускаем инициализацию узла

После создания аккаунта нам нужно выполнить инициализацию узла, выполняем ее из домашнего каталога пользователя book:

```
$ geth –datadir node1 init genesis.json
```

Здесь мы с помощью параметра **datadir** должны указать путь к рабочему каталогу, а в параметре **init** – путь к файлу первичного блока genesis.json.

Команда выполнит инициализацию и выведет на консоль результаты своей работы:

```
    INFO [02-13|08:43:53.934] Maximum peer
count                      ETH=25 LES=0 total=25
    INFO [02-13|08:43:53.936] Allocated cache and file
handles         database=/home/book/node1/geth/chaindata cache=16
handles=16
    INFO [02-13|08:43:53.950] Writing custom genesis block
    INFO [02-13|08:43:53.950] Persisted trie from memory
database           nodes=0 size=0.00B time=28.058µs gcnodes=0
gcsize=0.00B gctime=0s livenodes=1 livesize=0.00B
    INFO [02-13|08:43:53.951] Successfully wrote genesis
state       database=chaindata                hash=a5e5bc…3f490e
    INFO [02-13|08:43:53.951] Allocated cache and file
handles            database=/home/book/node1/geth/lightchaindata
cache=16 handles=16
    INFO [02-13|08:43:53.955] Writing custom genesis block
    INFO [02-13|08:43:53.955] Persisted trie from memory
database           nodes=0 size=0.00B time=1.778µs   gcnodes=0
gcsize=0.00B gctime=0s livenodes=1 livesize=0.00B
    INFO [02-13|08:43:53.956] Successfully wrote genesis
state       database=lightchaindata                 hash=a5e5bc…
3f490e
```

Для работы с узлом вам нужно будет открыть две консоли, подключившись в каждой консоли пользователем book.

Чтобы запустить узел, выполните в первой консоли следующую команду:

```
    $ geth -etherbase
"0x4f744742ac711fd111c7a983176db1d48d29f413" -datadir node1 -
nodiscover -mine -minerthreads 1 -maxpeers 0 -verbosity 3
-networkid 98760 -rpc -rpcapi="db,eth,net,web3,personal,web3"
console
```

В качестве параметра –etherbase нужно ввести адрес узла, полученный при первоначальном создании аккаунта.

На экране появится множество сообщений о ходе инициализации. В ходе этого процесса будет запущена генерация файла DAG. Вам нужно будет дождаться завершения процесса генерации:

```
    INFO [02-13|08:51:16.647] Maximum peer
count                      ETH=0 LES=0 total=0
    INFO [02-13|08:51:16.649] Starting peer-to-peer
node                 instance=Geth/v1.8.22-stable-7fa3509e/linux-
amd64/go1.10.4
    INFO [02-13|08:51:16.649] Allocated cache and file
handles         database=/home/book/node1/geth/chaindata cache=512
handles=524288
    INFO [02-13|08:51:16.662] Initialised chain
configuration       config="{ChainID: 1999 Homestead: 0 DAO: <nil>
DAOSupport: false EIP150: <nil> EIP155: 0 EIP158: 0 Byzantium:
<nil> Constantinople: <nil>  ConstantinopleFix: <nil> Engine:
unknown}"
    INFO [02-13|08:51:16.663] Disk storage enabled for ethash
caches   dir=/home/book/node1/geth/ethash count=3
    INFO [02-13|08:51:16.663] Disk storage enabled for ethash
DAGs     dir=/home/book/.ethash          count=2
    INFO [02-13|08:51:16.663] Initialising Ethereum
protocol           versions="[63 62]" network=98760
    INFO [02-13|08:51:16.724] Loaded most recent local
header          number=0 hash=a5e5bc…3f490e td=10 age=49y10mo16h
    INFO [02-13|08:51:16.724] Loaded most recent local full
block      number=0 hash=a5e5bc…3f490e td=10 age=49y10mo16h
    INFO [02-13|08:51:16.724] Loaded most recent local fast
block      number=0 hash=a5e5bc…3f490e td=10 age=49y10mo16h
    INFO [02-13|08:51:16.724] Loaded local transaction
journal         transactions=0 dropped=0
    INFO [02-13|08:51:16.725] Regenerated local transaction
journal    transactions=0 accounts=0
    INFO [02-13|08:51:16.732] New local node
record                    seq=3 id=eae4aa1f2059eed4 ip=127.0.0.1
udp=0 tcp=30303
    INFO [02-13|08:51:16.732] Started P2P
networking                enode://e26ca322fc935ef5b6b042ebbb6126f5bd7d6cf6903c1e19600cf7f6c8b5@127.0.0.1:30303?
discport=0"
    INFO [02-13|08:51:16.732] IPC endpoint
opened                     url=/home/book/node1/geth.ipc
    INFO [02-13|08:51:16.733] HTTP endpoint
opened                      url=http://127.0.0.1:8545     cors=
vhosts=localhost
    INFO [02-13|08:51:16.733] Transaction pool price threshold
updated price=1000000000
    INFO [02-13|08:51:16.733] Updated mining
threads                   threads=1
```

```
    INFO [02-13|08:51:16.733] Transaction pool price threshold
updated price=1000000000
    INFO [02-13|08:51:16.734] Commit new mining
work                    number=1 sealhash=5c4116…c8c1bf uncles=0
txs=0 gas=0 fees=0 elapsed=562.141µs
    INFO [02-13|08:51:16.779] Mapped network
port                        proto=tcp extport=30303 intport=30303
interface=NAT-PMP(192.168.0.1)
    Welcome to the Geth JavaScript console!

    instance: Geth/v1.8.22-stable-7fa3509e/linux-amd64/go1.10.4
    coinbase: 0x4f744742ac711fd111c7a983176db1d48d29f413
    at block: 0 (Wed, 31 Dec 1969 16:00:00 PST)
    datadir: /home/book/node1
    modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0
net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

    > INFO [02-13|08:51:18.459] Generating DAG in
progress               epoch=0 percentage=0 elapsed=983.779ms
    INFO [02-13|08:51:19.420] Generating DAG in
progress               epoch=0 percentage=1 elapsed=1.944s
    INFO [02-13|08:51:20.395] Generating DAG in
progress               epoch=0 percentage=2 elapsed=2.919s
    INFO [02-13|08:51:21.440] Generating DAG in
progress               epoch=0 percentage=3 elapsed=3.963s
    …

    INFO [02-13|08:55:56.193] Successfully sealed new
block               number=46 sealhash=fccbc1…5cc27f hash=776967…
3700c9 elapsed=7.306s
    INFO [02-13|08:55:56.193] 🔗 block reached canonical
chain          number=39 hash=9c6ba7…c95452

    INFO [02-13|08:55:56.193] ⚒ mined potential
block                  number=46 hash=776967…3700c9
    INFO [02-13|08:55:56.193] Commit new mining
work                    number=47 sealhash=41558f…3ef931 uncles=0
txs=0 gas=0 fees=0 elapsed=97.707µs
    > [1;5FINFO [02-13|08:56:01.030] Successfully sealed new
block               number=47 sealhash=41558f…3ef931 hash=f67a9b…
773bfd elapsed=4.837s
    INFO [02-13|08:56:01.030] 🔗 block reached canonical
chain          number=40 hash=3a6600…bde7e0

    INFO [02-13|08:56:01.030] ⚒ mined potential
block                  number=47 hash=f67a9b…773bfd
    INFO [02-13|08:56:01.030] Commit new mining
work                    number=48 sealhash=d4ab02…7151c7 uncles=0
txs=0 gas=0 fees=0 elapsed=97.374µs
```

В зависимости от производительности вашего виртуального или физического сервера генерация файла DAG может занять несколько минут или дольше.

Файл DAG содержит направленный ациклический граф (Directed Acyclic Graph). Он используется для добавления блоков в Ethereum с помощью алгоритма с названием Ethash. Его размер может составлять более 1 Гбайта. Размер этого блока увеличивается по мере роста сети Ethereum. Текущий размер блока можно узнать, например, на сайте https://investoon.com/tools/dag_size. На момент создания этого учебного курса файл DAG для основной сети Ethereum был размером 2.95 Гбайт.

Чем больше файл DAG, тем труднее выполнить майнинг. Если для майнинга используются видеокарты, то данные DAG должны полностью поместиться в память видеокарты, иначе применение видеокарты для майнинга будет бесполезным.

Так как запуск узла вы будете выполнять часто, рекомендуем подготовить пакетный файл для запуска с именем, например, start_node.sh (листинг 2.2.).

Листинг 2.2. Файл start_node.sh

```
    geth –etherbase "0x4f744742ac711fd111c7a983176db1d48d29f413" –
datadir    node1    –nodiscover    –mine    –minerthreads    1    –
maxpeers    0    –verbosity    3    –networkid    98760    –rpc    –
rpcapi="db,eth,net,web3,personal,web3" console
```

Параметры запуска узла

Расскажем о параметрах geth, которые мы использовали при запуске узла. Эти параметры были выбраны исходя из назначения нашего узла – мы создаем узел для учебной приватной сети Ethereum. Когда вы будете создавать узел для работы с тестовой сетью Rinkeby или с основной сетью Ethereum, нужно будет указывать другие параметры.

Чтобы получить краткую справку по всем командам и параметрам geth, запустите ее следующим образом:

```
$ geth –h
```

С параметром **datadir**, который указывает путь к каталогу блокчейна, вы уже знакомы. При запуске узла укажите тот же каталог, что мы использовали при инициализации приватного блокчейна.

Параметр **etherbase** задает публичный адрес, на который будет отправлено вознаграждение за майнинг.

Параметр **nodiscover** отключает поиск других узлов сети. Мы указали его, так как пока будем работать только с одним узлом блокчейна.

Мы также указали значение параметра **maxpeers**, равное нулю. Таким способом мы фактически отключили обмен по сети между узлами нашего блокчейна.

С помощью параметра **mine** мы запускаем так называемый майнинг – процесс создания новых блоков в нашем блокчейне. Это необходимо, так как без появления новых блоков выполнение транзакций и публикация смарт-контрактов будут невозможны.

Параметр **minerthreads** указывает количество потоков, используемых для майнинга. Если ресурсы вашего сервера позволяют и там установлен многоядерный процессор, то для ускорения майнинга можно увеличить значение этого параметра.

Очень важный параметр – **networkid**. Это идентификатор сети. Здесь мы должны указать уникальный идентификатор 98760 нашего приватного блокчейна.

Параметр **verbosity** задает детализацию журнала:
• 0 – не записывать данные в журнал;
• 1 – записывать сообщения об ошибках;
• 2 – записывать предупреждающие сообщения;
• 3 – записывать информационные сообщения;
• 4 – записывать отладочную информацию;
• 5 – записывать детальную информацию.
По умолчанию используется значение 3.

Так как мы будем работать с узлом с помощью протокола JSON RPC, нам необходимо включить такую возможность, указав параметр **rpc**. Дополнительно с помощью параметра **rpcapi** мы перечисляем, какие программные интерфейсы должен предоставить нам узел. Здесь мы указали такой набор: db, eth, net, web3, personal, web3. Подробнее об этом мы расскажем позже.

При запуске geth мы указываем команду **console**. Эта команда запускает интерактивную консоль JavaScript, где мы сможем выдавать команды.

Подключаемся к нашему узлу

Теперь откройте вторую консоль и введите в ней следующую команду:

```
$ geth –datadir node1 –networkid 98760 attach ipc://home/book/
node1/geth.ipc
```

Эта команда откроет консоль geth и подключится к вашему приватному узлу.

Здесь необходимо указать те же значения параметров **datadir** и **networked**, что и при запуске узла. Команда **attach** подключается к узлу, адрес которого задан после нее, и запускает интерактивную консоль JavaScript. В адресе нам нужно указать полный путь к рабочему каталогу нашего приватного блокчейна.

Запишите команду подключения в файл attach_node.sh для удобства (листинг 2.3.).

Листинг 2.3. Файл attach_node.sh

```
geth –datadir node1 –networkid 98760 attach ipc://home/book/
node1/geth.ipc
```

Теперь запустите этот файл, и вы увидите приглашение консоли geth:

```
$ sh attach_node.sh
Welcome to the Geth JavaScript console!

instance: Geth/v1.8.22-stable-7fa3509e/linux-amd64/go1.10.4
coinbase: 0x3cd46aab0631305437842cf639218e41ce946baa
at block: 379 (Wed, 13 Feb 2019 09:12:48 PST)
datadir: /home/book/node1
modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0
net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

>
```

Введите в этом приглашении команду web3.eth.accounts:

```
> web3.eth.accounts
["0x4f744742ac711fd111c7a983176db1d48d29f413"]
```

Вы увидите идентификатор (адрес) аккаунта, который мы создали ранее, указав для него пароль. У вас этот идентификатор будет другой.

Попробуйте также ввести команду web3.version. Эта команда позволяет посмотреть версию фреймворка Web3, с помощью которого мы будем работать с контрактами, версию geth, а также номер сети. Мы задали номер нашей приветной сети, равный 98760.

Для стабильного релиза Geth версии 1.8.22 на консоль будет выведено сообщение:

```
> web3.version
{
 api: "0.20.1",
 ethereum: "0x3f",
 network: "98760",
 node: "Geth/v1.8.22-stable-7fa3509e/linux-amd64/go1.10.4",
 whisper: undefined,
 getEthereum: function(callback),
 getNetwork: function(callback),
 getNode: function(callback),
 getWhisper: function(callback)
}
```

Использование нестабильной версии Geth будет отмечено в поле node:

```
> web3.version
{
 api: "0.20.1",
 ethereum: "0x3f",
 network: "98760",
 node: "Geth/v1.8.11-unstable/linux-amd64/go1.9.6",
 whisper: undefined,
 getEthereum: function(callback),
 getNetwork: function(callback),
 getNode: function(callback),
 getWhisper: function(callback)
}
```

Управление майнингом и проверка баланса

При запуске узла мы автоматически запускаем майнинг. Текущий баланс вы можете проверить при помощи следующей команды:

```
> web3.fromWei( eth.getBalance(eth.coinbase) )
0
```

Позже мы опишем экономику Ethereum и эти команды детальнее.

Сразу после инициализации сети баланс нашего аккаунта равен нулю. Однако по мере того, как будут «добыты» новые блоки, баланс будет расти:

```
> web3.fromWei( eth.getBalance(eth.coinbase) )
15
```

В первой консоли, где мы запустили узел, добавление каждого блока будет отмечено такими сообщениями:

```
    INFO [02-13|09:37:10.577] 🔗 block reached canonical
chain          number=45 hash=924ce1…d8b5a2

    INFO [02-13|09:37:10.577] 🔨 mined potential
block                  number=52 hash=a80a36…153593
    INFO [02-13|09:37:10.577] Commit new mining
work                   number=53 sealhash=3acb6c…0ecd19 uncles=0
txs=0 gas=0 fees=0 elapsed=130.557µs
    INFO [02-13|09:37:11.223] Successfully sealed new
block               number=53 sealhash=3acb6c…0ecd19 hash=14e0fa…
575494 elapsed=645.999ms
    INFO [02-13|09:37:11.223] 🔗 block reached canonical
chain          number=46 hash=c5ff7a…da8069

    INFO [02-13|09:37:11.224] 🔨 mined potential
block                  number=53 hash=14e0fa…575494
    INFO [02-13|09:37:11.224] Commit new mining
work                   number=54 sealhash=96235b…f3fc50 uncles=0
txs=0 gas=0 fees=0 elapsed=124.053µs
    INFO [02-13|09:37:11.723] Successfully sealed new
block               number=54 sealhash=96235b…f3fc50 hash=e5438e…
2f6f2e elapsed=498.975ms
```

С помощью команд miner.start и miner.stop можно запускать и останавливать майнинг.

При ручном запуске майнинга нужно указать количество потоков для поиска новых блоков:

```
> miner.start(4)
```

Здесь мы запускаем майнинг на четырех ядрах виртуальной машины. Перед запуском проверьте, сколько ядер доступно на вашем сервере.

Заметим, что при отладке смарт-контрактов процесс майнинга останавливать не нужно, иначе ваш узел не сможет обрабатывать транзакции, публиковать контракты и вызывать методы контрактов. Тем не менее, вы всегда сможете остановить майнинг с помощью такой команды:

```
> miner.stop()
```

Если скорость майнинга недостаточна и вам приходится ждать появления новых блоков более 20-30 секунд, попробуйте увеличить размер оперативной памяти и количество процес-

сорных ядер на виртуальной машине. Облачные хостинги, как правило, позволяют сделать это очень просто через Web-интерфейс вашего личного кабинета.

Завершение работы консоли Geth

Для завершения работы Geth введите в приглашении команду exit:

```
> exit
```

Итоги урока

На втором уроке вы подготовили рабочую среду, необходимую для дальнейшего изучения Ethereum и смарт-контрактов Solidity. Вы создали сервер с ОС Ubuntu или Debian, установили Go, Geth и программу распределенного хранилища данных Swarm.

Далее вы создали приватный блокчейн, состоящий из одного узла, выполнили инициализацию этого узла и убедились, что в вашем блокчейне работает майнинг. Вы научились запускать свой узел Ethereum и подключаться к нему в консоли. Вы также научились выдавать простейшие консольные команды Geth и теперь готовы для первых экспериментов с вашим приватным блокчейном.

Урок 3. Подготовка рабочей среды на Raspberry Pi 3

Цель урока: создать узел собственного приватного блокчейна Ethereum для дальнейшей работы в рамках этого курса на микрокомпьютере Raspberry Pi 3 с операционной системой Rasberian. Изучить способ запуска узла приватной сети Ethereum при помощи параметра –dev утилиты Geth.

Практические задания: установка операционной системы Rasberian на Raspberry Pi 3, установка ПО Geth, обеспечивающего работу узла нашего блокчейна, а также демона децентрализованного хранилища данных Swarm. Запуск узла приватного блокчейна Ethereum с помощью параметра –dev утилиты geth.

На предыдущем уроке вы научились создавать свой приватный блокчейн Ethereum на сервере с ОС Ubuntu и Debian. Возможно, для первых опытов и отладки смарт-контрактов вам больше подойдет очень недорогой микрокомпьютер Raspberry Pi 3 (или его более новая модель Raspberry Pi 3 model B), который создавался как раз для обучения.

К сожалению, ресурсы микрокомпьютера Raspberry Pi довольно ограничены, в частности, объем оперативной памяти составляет всего 1 Гбайт (уже есть в продаже Raspberry Pi 4 с 4 Гбайт памяти на борту). Поэтому на Raspberry Pi не получится запустить майнинг в Geth аналогично тому, как мы это делали на предыдущем уроке в ОС Ubuntu и Debian.

Тем не менее, выход есть.

Если раньше вы создавали файл первичного блока, инициализировали блокчейн и запускали майнинг, то на этом уроке мы будем использовать другой, возможно, более удобный способ создания тестовой сети. Мы создадим ее, запустив Geth с параметром –dev, специально предназначенным для этого случая. Разумеется, такой способ можно применять и на обычных серверах, физических и виртуальных.

Подготовка Raspberry Pi 3 к работе

Для первоначальной установки ОС Rasberian вам нужно подключить к микрокомпьютеру блок питания, монитор (лучше всего через интерфейс HDMI), клавиатуру (через интерфейс USB). Также нужно подключить микрокомпьютер кабелем Ethernet к вашему роутеру.

Установите в микрокомпьютер карту памяти MicroSD объемом не меньше 8 Гбайт.

После того как установка и настройка Rasberian будут завершены, монитор и клавиатуру можно будет отключить.

Хорошую инструкцию по начальной подготовке Raspberry Pi 3 к работе можно найти здесь: http://dmitrysnotes.ru/raspberry-pi-3-obzor-i-nachalo-raboty.

Установка Rasberian

Прежде всего нужно отформатировать карту памяти MicroSD. В ОС Microsoft Windows воспользуйтесь для этого бесплатной утилитой SD Memory Card Formatter, загрузив ее с сайта https://www.sdcard.org/downloads/formatter_4/. Стандартные средства форматирования ОС Microsoft Windows не подойдут.

Далее нужно скачать дистрибутив Raspbian с сайта https://www.raspberrypi.org/downloads/raspbian/. Для наших целей подойдет версия Raspbian Stretch Lite. Загрузите архив Zip и распакуйте его. В итоге вы получите файл с расширением имени img, например, 2018-04-18-raspbian-stretch-lite.img.

Запишите этот файл на предварительно отформатированную карту памяти MicroSD. Это можно сделать с помощью бесплатной утилиты Rufus. Загрузите ее с сайта https://rufus.akeo.ie/.

Далее установите подготовленную таким способом карту в Raspberry Pi и включите питание. Проследите процесс начальной загрузки Rasberian на мониторе.

В первый раз для подключения через консоль используйте имя пользователя pi и пароль raspberry. После подключения сразу смените пароль с помощью утилиты passwd.

Установка обновлений

Сразу после установки микрокомпьютер получит динамический адрес IP от сервера DHCP вашего роутера, и вы сможете продолжить установку.

Прежде всего установите обновления пакетов до актуальной версии:

```
sudo apt-get update
sudo apt-get upgrade
```

Включение доступа SSH

Чтобы с микрокомпьютером можно было работать удаленно через SSH, нужно включить такую возможность. Для этого в консоли введите следующую команду:

```
sudo raspi-config
```

Выберите в списке строку **5. Interfacing options**, найдите в списке **SSH** и включите.

Установка статического адреса IP

Для удаленной работы через SSH будет удобно, если микрокомпьютеру будет выделен статический адрес IP. Чтобы это сделать, отредактируйте файл /etc/dhcpcd.conf:

```
sudo nano /etc/dhcpcd.conf
```

Уберите символы комментария # со следующих строк:

```
interface eth0
static ip_address=192.168.0.38/24
static ip6_address=fd51:42f8:caae:d92e::ff/64
static routers=192.168.0.1
static domain_name_servers=192.168.0.1 8.8.8.8 fd51:42f8:caae:d92e:
```

Также в строке static ip_address укажите нужный адрес IP. Заметим, что этот адрес нужно исключить из DHCP соответствующей настройкой вашего роутера.

После изменения файла перезагрузите микрокомпьютер:

```
sudo shutdown –r now
```

Теперь вы можете подключиться к нему удаленно с помощью любого клиента SSH по статическому адресу. Если удаленный доступ работает, вы можете отключить клавиатуру и монитор от Raspberry Pi и продолжить работу через SSH.

Дальнейшие действия очень похожи на то, что мы делали в предыдущем уроке. Однако есть и отличия.

Установка необходимых утилит

Как и на прошлом уроке, мы рекомендуем вам установить редактор vim, утилиты sudo и git:

```
# apt-get install vim
# apt-get install sudo
# apt-get install git
```

Создайте пользователя book и при помощи команды visudo добавьте этому пользователю возможность работать с командой sudo, как мы это делали на предыдущем уроке.

Установка Go

Мы будем работать с Go версии 1.9.6. Установка для Rasbian очень похожа на установку для Debian, однако вам потребуется дистрибутив, рассчитанный на архитектуру процессора ARM.

Загрузка дистрибутива Go

Как мы уже говорили, дистрибутивы Go различных версий и для различных платформ можно найти здесь: https://golang.org/dl/.

Подключимся к нашему микрокомпьютеру Raspberry Pi пользователем book и загрузим архив Go нужной версии:

```
$ curl -O https://dl.google.com/go/go1.9.6.linux-armv6l.tar.gz
```

Как и на предыдущем уроке, вам нужно добавить возможность работать пользователю book с командой sudo, для чего нужно воспользоваться командой visudo.

Добавьте в конец файла /etc/sudoers строку:

```
book ALL=(ALL) ALL
```

Подключитесь к консоли с правами пользователя book и распакуйте загруженный архив в каталог /usr/local:

```
$ sudo tar -C /usr/local -xzf go1.9.6.linux-armv6l.tar.gz
```

Обратите внимание, что здесь мы распаковываем файл дистрибутива для процессоров ARM.

Установка переменных окружения

Создайте в домашнем каталоге пользователя book каталог go и установите переменные окружения:

```
$ mkdir -p ~/go; echo "export GOPATH=$HOME/go" >> ~/.bashrc
$ echo "export PATH=$PATH:$HOME/go/bin:/usr/local/go/bin" >>
~/.bashrc
$ source ~/.bashrc
```

Проверьте, что переменные окружения установлены:

```
$ printenv | grep go

GOPATH=/root/go
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin:/root/go/bin:/usr/local/go/bin
```

Проверка версии Go

Прежде чем перейти собственно к установке Geth и Swarm, нужно проверить версию go. Теперь должно быть указано, что установлена версия 1.9.6 для процессоров ARM:

```
$ go version
go version go1.9.6 linux/arm
```

Установка Geth и Swarm

Установка Geth и Swarm выполняется аналогично тому, как мы это делали на предыдущем уроке.

Загрузите исходный код Geth из репозитория на GitHub:

```
$ mkdir -p $GOPATH/src/github.com/ethereum
$ cd $GOPATH/src/github.com/ethereum

$ git clone https://github.com/ethereum/go-ethereum
$ cd go-ethereum
$ git checkout master

$ go get github.com/ethereum/go-ethereum
```

Запустите компиляцию клиента Geth и Swarm:

```
go install -v ./cmd/geth
go install -v ./cmd/swarm
```

Если при компиляции появились ошибки, попробуйте установить Go другой версии. Перед этим удалите все каталоги, созданные в процессе предыдущей установки.

Если же все хорошо, то осталось только проверить версию установленного Geth и Swarm:

```
$ geth version
Geth
Version: 1.8.9-unstable
Architecture: arm

Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.9.6
Operating System: linux
GOPATH=/home/book/go
GOROOT=/usr/local/go

$ swarm version
Swarm
Version: 1.8.9-unstable
Network Id: 0
Go Version: go1.9.6
OS: linux
GOPATH=/home/book/go
GOROOT=/usr/local/go
```

Обратите внимание, что команда geth version сообщает о том, что она работает на процессоре с архитектурой ARM.

Создаем приватный блокчейн

На предыдущем уроке мы создавали приватный блокчейн, подготовив для него первичный блок в файле genesis.json. Затем мы создали в домашнем каталоге пользователя book рабочий каталог node1, создали аккаунт, запустили инициализацию узла и, наконец, запустили узел нашего блокчейна. При этом был создан файл DAG с направленным ациклическим графом и запущен майнинг. В другой консоли мы подключились к нашему узлу и выдали там несколько команд Web3.

На этот раз мы сделаем все намного проще. Создадим рабочий каталог node1 для размещения данных блокчейна:

```
$ mkdir node1
```

Теперь запустим узел приватной сети при помощи следующей команды:

```
$ geth –datadir node1 –networkid 98760 –dev –rpc –
rpcapi="db,eth,net,web3,personal,web3" console
```

В окне консоли появятся сообщения о запуске узле сети.

Сохраните команду запуска узла в пакетном файле с именем start_node.sh (листинг 3.1.).

Листинг 3.1. Файл start_node.sh

```
geth –datadir node1 –networkid 98760 –dev –rpc –
rpcapi="db,eth,net,web3,personal,web3" console
```

Обратите внимание, что мы указали здесь те же значения параметров datadir, networked, rpc и rpcapi, что и на предыдущем уроке.

Для подключения к запущенному таким способом узлу вы можете использовать ту же самую команду, что и раньше:

```
$ geth –datadir node1 –networkid 98760 attach ipc://home/book/
node1/geth.ipc
```

Запишите команду подключения в файл attach_node.sh для удобства (листинг 3.2.).

Листинг 3.2. Файл attach_node.sh

```
geth –datadir node1 –networkid 98760 attach ipc://home/book/
node1/geth.ipc
```

Проверка учетной записи и баланса

Запустите файл attach_node.sh и в приглашении консоли Geth введите команду accounts:

```
> web3.eth.accounts
["0xd902f8405a6108e8bd9343d1bfccf21a081d2897"]
```

Оказывается, в нашем тестовом узле уже определен пользователь, хотя мы не создавали его явным образом и не задавали пароль, как на прошлом уроке. У этого пользователя пустой пароль.

Попробуйте также ввести команду web3.version:

```
> web3.version
{
 api: "0.20.1",
 ethereum: "0x3f",
 network: "98760",
 node: "Geth/v1.8.9-unstable/linux-arm/go1.9.6",
 whisper: "6.0",
 getEthereum: function(callback),
 getNetwork: function(callback),
 getNode: function(callback),
 getWhisper: function(callback)
}
```

Здесь будут показаны те же самые данные приватной сети, что и на предыдущем уроке.

Проверьте текущий баланс при помощи следующей команды:

```
> web3.fromWei( eth.getBalance(eth.coinbase) )
1.157920892373161954235709850086879078532699846656405640394575840007
+59
```

Как видите, на нашем аккаунте есть уже довольно много монет Ether, хотя мы и не запускали майнинг явным образом. Это все благодаря тестовому режиму geth, выбранному с помощью параметра –dev.

Итоги урока

На третьем уроке вы научились устанавливать узел приватного блокчейна Ethereum на микрокомпьютер Raspberry Pi 3, обладающего оперативной памятью размеров всего 1 Гбайт.

Хотя на этом компьютере не удается запустить майнинг утилитой Geth, вы использовали параметр –dev для создания тестового узла. На таком узле майнинг запускать не нужно, т.к. там уже предусмотрен аккаунт с очень большим балансом.

Урок 4. Учетные записи и перевод средств между аккаунтами

Цель урока: научиться просматривать список аккаунтов, создавать новые аккаунты, познакомиться с криптовалютными единицами в сети Ethereum, а также научиться переводить средства с одного аккаунта на другой с помощью транзакции из консоли Geth.

Практические задания: нужно просмотреть список аккаунтов, добавить новый аккаунт и перевести на него средства с другого аккаунта. Далее нужно посмотреть состояние транзакции и получить ее квитанцию.

К четвертому уроку мы подготовили серверы Ubuntu и Debian и установили на них Go Ethereum (программу Geth), создали узел приватной сети. Мы проделали аналогичную операцию с микрокомпьютером Raspberry Pi 3, установив на него ОС Rasberian и Geth для запуска сети в тестовом режиме с параметром –dev.

Мы научились подключаться к узлу сети в отдельном окне консоли, получив приглашение Geth в виде символа “>” (далее мы будем назвать это приглашение консолью Geth). Напомним, что для выхода из консоли Geth и возврата к системному приглашению ОС нужно ввести команду exit.

Детальное описание программного интерфейса фреймворка Web3 JavaScript API версии 0.2x.x, вы найдете здесь: https://github.com/ethereum/wiki/wiki/JavaScript-API.

На момент подготовки книги версия 1.0 еще не реализована, но вы можете изучить документацию на нее по адресу http://web3js.readthedocs.io/en/1.0/. В нашем руководстве мы будем приводить примеры использования и стабильной версии 0.2x.x, и бета-версий 1.0-beta.xx.

Просмотр и добавление аккаунтов

В сети Ethereum, как тестовой, так и «настоящей», аккаунты играют очень важную роль. Именно аккаунтам принадлежат криптовалютные средства, необходимые для выполнения транзакций. Все взаимодействия в сети Ethereum происходят в виде транзакций между аккаунтами.

Заметим, что помимо аккаунтов пользователей, в сети Ethereum предусмотрены аккаунты смарт-контрактов, загруженных в блокчейн, но об этом мы будем говорить позже.

В процессе инициализации приватной узла сети Ethereum на втором уроке мы добавляли первую учетную запись (аккаунт) явным образом, задавая для нее пароль:

```
$ geth –datadir node1 account new
```

На третьем уроке мы запускали тестовую приватную сеть, задавая geth параметр –dev. При этом на узле уже была создана одна учетная запись, к тому же с большим положительным балансом.

Когда вы добавляете аккаунт, создаются приватный и публичный ключ. Приватный ключ хранится на диске вашего сервера, публичный используется для идентификации аккаунта другими пользователями.

Просмотр списка аккаунтов

Откройте две консоли и подключитесь в каждой из них пользователем book. Далее в первой консоли запустите узел следующей командой:

```
$ sh start_node.sh
```

Во второй консоли подключитесь к запущенному узлу:

```
$ sh attach_node.sh
```

Скрипты start_node.sh и attach_node.sh мы подготовили ранее на втором уроке нашего курса.

Теперь, чтобы посмотреть список аккаунтов, достаточно выдать такую команду в консоли второй Web3:

```
> web3.eth.accounts
["0x4f744742ac711fd111c7a983176db1d48d29f413"]
```

Если аккаунтов несколько, они будут показаны через запятую как элементы массива:

```
> web3.eth.accounts
["0x4f744742ac711fd111c7a983176db1d48d29f413",
"0xf212d0180b331a88bd3cafbd77bbd0d56398ae00"]
```

Каждый аккаунт обладает своим идентификатором, или адресом, представляющим собой длинное шестнадцатеричное число. Именно эти адреса и показывает команда web3.eth.accounts.

Добавление аккаунта

Вы можете добавить аккаунты в консоли Geth или в консоли ОС, запустив там Geth с соответствующим параметром.

Давайте добавим новый аккаунт, указав для него пароль test (никогда не используйте такие простые пароли в рабочих проектах):

```
> web3.personal.newAccount("test")
"0x346cc69a63f9b84c45f17e337574c0150ab6bc03"
```

Здесь мы обратились к объекту web3.personal. Чтобы такие обращения были возможны, при запуске узла сети мы указали название этого объекта наряду с другими объектами в параметре rpcapi (см. урок 2). При использовании параметра –dev параметр rpcapi задавать явным образом не требуется.

Если раньше массив аккаунтов содержал два элемента, то теперь в нем будет на один элемент больше:

```
> web3.eth.accounts
["0x4f744742ac711fd111c7a983176db1d48d29f413",
"0xf212d0180b331a88bd3cafbd77bbd0d56398ae00",
"0x346cc69a63f9b84c45f17e337574c0150ab6bc03"]
```

Для каждого аккаунта на узле создается ключ. Другой неотъемлемый параметр аккаунта – это его пароль. Пароль при создании аккаунта никуда не записывается, поэтому его невозможно восстановить.

Теперь для добавления аккаунта воспользуемся командой geth account с параметром new. Откройте новое консольное окно пользователем book и введите там следующую команду:

```
$ geth –datadir node1 account new
```

Будет добавлен новый аккаунт, а его идентификатор (адрес), будет показан в фигурных скобках:

```
INFO [02-17|23:01:28.855] Maximum peer
count ETH=25 LES=0 total=25
Your new account is locked with a password. Please give a
password. Do not forget this password.
Passphrase:
Repeat passphrase:
Address: {ae7bb3649a5c597d44f812b4a636f3cc21ee98e1}
```

При добавлении аккаунта у вас будет запрошен пароль.

Обратите внимание, что во всех этих командах мы указывали путь к рабочему каталогу нашего приватного блокчейна с помощью параметра datadir.

Теперь если вы вернетесь в консольное окно Geth и посмотрите там список аккаунтов, то увидите, что теперь на вашем узле определено целых четыре аккаунта:

```
> web3.eth.accounts
```

```
    ["0x4f744742ac711fd111c7a983176db1d48d29f413",
"0xf212d0180b331a88bd3cafbd77bbd0d56398ae00",
"0x346cc69a63f9b84c45f17e337574c0150ab6bc03",
"0xae7bb3649a5c597d44f812b4a636f3cc21ee98e1"]
```

Параметры команды geth account

Запуская команду geth account с параметрами в консоли ОС, вы можете полностью управлять аккаунтами. Это самый безопасный способ, так как в процессе управления пользователями вы не обращаетесь ни к каким фреймворкам, а работаете с Geth напрямую.

Команде geth account можно передавать параметры, перечисленные в табл. 4.1.

Табл. 4.1. Параметры команды geth account

Параметр	Описание
list	Вывод списка аккаунтов
new	Создание нового аккаунта
update	Обновление существующего аккаунта (в том числе смена пароля)
import	Импорт приватного ключа в новый аккаунт

Хорошее описание управления аккаунтами с примерами можно найти здесь: https://github.com/ethereum/go-ethereum/wiki/Managing-your-accounts.

Пароли аккаунтов

Обращаем ваше внимание, что пароли аккаунтов **невозможно восстановить**. Если вы создаете узел для реального проекта или собираетесь использовать аккаунт для хранения криптовалютных средств, обращайтесь с паролем очень осторожно.

Лучше всего запомнить пароль или записать его на обычной бумаге и хранить в безопасном месте. Доступ к вашему аккаунту обеспечивается приватным ключом и паролем.

Приватный ключ хранится на сервере узла и может быть похищен, например, с использованием какой-нибудь уязвимости в ОС или в ПО, установленном на сервере. Но чтобы использовать похищенный ключ, злоумышленнику потребуется еще и пароль. Мы не рекомендуем хранить пароль в компьютере (на сервере, ноутбуке и любом другом), так как он может быть похищен вместе с ключом.

Также учтите, что вредоносные программы могут перехватить управление клавиатурой, а также получить доступ к буферу обмена данных Clipboard. В этом случае они смогут похитить пароль в процессе его ввода с клавиатурой. Используйте антивирусные программы для защиты от вирусов и вредоносных программ различных типов.

Если злоумышленник сможет похитить приватный ключ и пароль, он получит полный доступ к вашему аккаунту и сможет, например, перевести все имеющиеся там криптовалютные средства на свой аккаунт.

Криптовалюта в Ethereum

Как мы уже говорили, для того чтобы владелец аккаунта мог проводить транзакции, публиковать и запускать смарт-контракты, на аккаунте должны быть средства. В тестовой сети, которую мы создали на втором уроке, эти средства можно получить при помощи майнинга. Если вы создаете тестовую сеть с помощью Geth, передавая ей параметр –dev, уже будет создан аккаунт, имеющий на балансе значительное количество криптовалюты.

Далее вы можете создавать новые аккаунты и переводить на них средства с тех аккаунтов тестовой сети, где уже имеются монеты.

Однако в основной сети Ethereum майнинг – дорогое и долгое занятие. Есть другая возможность, а именно: приобретение криптовалютных средств у владельцев других аккаунтов на биржах и обменниках. При этом владелец аккаунта может перевести средства на ваш аккаунт. Вы должны будете сообщить адрес вашего аккаунта, и если вы ошибетесь, то не будет никакой возможности вернуть ваши средства.

На этом уроке мы будем выполнять операцию перевода средств (разумеется, тестовых) в нашей приватной сети.

Денежные единицы Ethereum

Внутренняя валюта сети Ethereum называется Ether, или эфир. Когда владелец аккаунта выполняет транзакции, валюта, которая имеется на счету аккаунта, тратится.

Помимо Ether, существуют более крупные и более мелкие единицы криптовалюты Ethereum (аналогично тому, как в фиатных деньгах существуют рубли и копейки, доллары и центы).

Самая мелкая единица – это Wei. В одном эфире (т.е. в одном Ether) содержится целых 1 000 000 000 000 000 000 единиц Wei.

Как вы увидите далее, единица Wei удобнее, чем Ether, когда речь идет об оплате транзакций, не отнимающих много ресурсов.

В таблице 4.2. мы привели полный список денежных единиц Ethereum и их ценность в единицах Wei.

Таблица 4.2. Денежные единицы Ethereum

	Значение, Wei
Wei	1
Kwei, Ada, Femtoether	$1\,000 = 10^3$
Mwei, Babbage, Picoether	$1\,000\,000 = 10^6$
Gwei, Shannon, Nanoether, Nano	$1\,000\,000\,000 = 10^9$
Szabo, Microether,Micro	$1\,000\,000\,000\,000 = 10^{12}$
Finney, Milliether,Milli	$1\,000\,000\,000\,000\,000 = 10^{15}$
Ether	$1\,000\,000\,000\,000\,000\,000 = 10^{18}$
Kether, Grand,Einstein	$1\,000\,000\,000\,000\,000\,000\,000 = 10^{21}$
Mether	$1\,000\,000\,000\,000\,000\,000\,000\,000 = 10^{24}$
Gether	$1\,000\,000\,000\,000\,000\,000\,000\,000\,000 = 10^{27}$
Tether	$1\,000\,000\,000\,000\,000\,000\,000\,000\,000\,000 = 10^{30}$

В интернете можно найти сайты с конвертами криптовалют Ethereum, вот один из них: https://etherconverter.online/. Здесь же показывается текущий курс Ether по отношению к доллару и евро.

Определяем текущий баланс наших аккаунтов

Выше мы добавили в нашу приватную сеть несколько аккаунтов. Полный список аккаунтов всегда можно посмотреть в консоли Geth при помощи команды web3.eth.accounts.

С помощью функции web3.eth.getBalance мы можем посмотреть баланс аккаунта в единицах Wei. Адрес аккаунта нужно передать функции в качестве параметра:

```
> web3.eth.getBalance("0x4f744742ac711fd111c7a983176db1d48d29f413")
2.3085e+22
```

Функция web3.eth.getBalance возвращает достаточно большое число. Вообще при работе с API фреймворков Ethereum мы часто будем иметь дело с очень большими числами. Когда вы будете разрабатывать свое децентрализованное приложение (DApp), это нужно будет учитывать.

Для того чтобы узнать баланс нужного вам аккаунта в Ether, используйте функцию fromWei:

```
> web3.fromWei(eth.getBalance("0x4f744742ac711fd111c7a983176db1d48d29f413
23110
```

При этом функции eth.getBalance нужно передать адрес проверяемого аккаунта.

Перевод средств с одного аккаунта на другой

Если подобным образом проверить баланс для аккаунтов, созданных нами дополнительно, то окажется, что сразу после создания он равен нулю, например:

```
    >
web3.fromWei( eth.getBalance("0xf212d0180b331a88bd3cafbd77bbd0d56398ae0
    0
```

Это неудивительно, ведь мы еще не переводили средства на эти аккаунты. Но можно перевести деньги с основного аккаунта нашей приватной сети, которую мы создали на втором уроке. Для нее был запущен майнинг, поэтому там уже должны быть средства.

Метод eth.sendTransaction

Давайте посмотрим, какие у нас есть аккаунты и какой на них баланс:

```
    > web3.eth.accounts
    ["0x4f744742ac711fd111c7a983176db1d48d29f413",
"0xf212d0180b331a88bd3cafbd77bbd0d56398ae00",
"0x346cc69a63f9b84c45f17e337574c0150ab6bc03",
"0xae7bb3649a5c597d44f812b4a636f3cc21ee98e1"]

    >
web3.fromWei( eth.getBalance("0x4f744742ac711fd111c7a983176db1d48d29f41
    23135
    >
web3.fromWei( eth.getBalance("0xf212d0180b331a88bd3cafbd77bbd0d56398ae0
    0
    >
web3.fromWei( eth.getBalance("0x346cc69a63f9b84c45f17e337574c0150ab6bc0
    0
    >
web3.fromWei( eth.getBalance("0xae7bb3649a5c597d44f812b4a636f3cc21ee98e
    0
```

Как видите, на первом из этих аккаунтов средства есть, а на остальных – ничего нет.

Давайте переведем часть средств, а именно 0.05 Ether, с первого их этих аккаунтов на другой, где средств нет. Воспользуемся для этого методом eth.sendTransaction:

```
    >
eth.sendTransaction({from:"0x4f744742ac711fd111c7a983176db1d48d29f413",
to:"0xf212d0180b331a88bd3cafbd77bbd0d56398ae00",          value:
web3.toWei(0.05, "ether")})
```

При попытке выполнить эту операцию вы, однако, получите сообщение об ошибке:

```
    Error: authentication needed: password or unlock
```

```
        at web3.js:3143:20
        at web3.js:6347:15
        at web3.js:5081:36
        at <anonymous>:1:1
```

Перед выполнением такой операции необходимо разблокировать исходный аккаунт, с которого отправляются средства. Для разблокировки введите такую команду:

```
    >
web3.personal.unlockAccount("0x4f744742ac711fd111c7a983176db1d48d29f413
"*********")
    true
```

Вместо звездочек укажите пароль, с которым данный аккаунт создавался. Если аккаунт и пароль были указаны правильно, на консоли вы увидите true.

Теперь повторите вызов функции eth.sendTransaction:

```
    >
eth.sendTransaction({from:"0x4f744742ac711fd111c7a983176db1d48d29f413",
to:"0xf212d0180b331a88bd3cafbd77bbd0d56398ae00",          value:
web3.toWei(0.05, "ether")})
    "0xb6d13a5e915c3af1feabad7caec7b45348146695973b32285df287639717e916
```

На этот раз функция выполнится успешно и вернет нам так называемый хеш транзакции (Transaction Hash) со значением:

```
    0xb6d13a5e915c3af1feabad7caec7b45348146695973b32285df287639717e916
```

Чтобы перевод средств произошел успешно, эта транзакция должна быть выполнена. Кроме того, нужно дождаться, когда в вашу тестовую сеть будет добавлен новый блок. После этого средства появятся на целевом счету.

Посмотрите на консоль, где мы запустили наш узел. Видим, что там есть сообщение о запуске транзакции с указанным выше хешем, а также о том, что был добавлен новый блок:

```
    INFO              [02-17|23:20:50.917]              Submitted
fullhash=0xb6d13a5e915c3af1feabad7caec7b45348146695973b32285df287639717e916
        recipient=0xF212D0180B331a88BD3CafbD77bBd0D56398aE00
    INFO      [02-17|23:20:53.018]      Commit      new      mining
    work                  number=4643 sealhash=0f860c…d73ae1 uncles=0
    txs=1 gas=21000 fees=2.1e-05 elapsed=36.186ms
    INFO    [02-17|23:22:10.119]    Successfully    sealed    new
    block             number=4643 sealhash=0f860c…d73ae1 hash=3c9761…
    8b0eea elapsed=1m17.116s
    INFO   [02-17|23:22:10.119]   block   reached   canonical
    chain          number=4636 hash=3b5237…0e8761
    INFO     [02-17|23:22:10.119]      mined     potential
    block                  number=4643 hash=3c9761…8b0eea
```

Теперь, если проверить баланс второго аккаунта, на который мы переводили средства, то окажется, что он как раз равен 0.05 Ether, как и должно быть:

```
    >
web3.fromWei( eth.getBalance("0xf212d0180b331a88bd3cafbd77bbd0d56398ae0
    0.05
```

Функции eth.sendTransaction мы передали в параметрах from и to адреса исходного и целевого аккаунта соответственно. В параметре value мы задали количество переводимых средств в единицах Wei. Так как нам удобнее указывать размер средств в более крупных единицах Ether, для перевода в Wei мы воспользовались функцией web3.toWei.

Заметим, что в реальной сети вместе с транзакцией нам нужно указать средства, которые пойдут на обработку этой транзакции (так называемый газ). Для этого команде eth.sendTransaction нужно задать параметры gas (количество газа, заданное для выполнения операции) и gasPrice (стоимость газа в Wei):

```
    eth.sendTransaction({from:"0x208970e5e3d48a6eab968e64ba3447f6181310
to:"0x82a4165f21d8f1867d536e81537fc0085e5470a1",value:
web3.toWei(5, "ether"), gas: 120000, gasPrice: 80000000000})
```

Просмотр состояния транзакции

Зная хеш транзакции, вы можете получить о ней определенную информацию. Это можно сделать при помощи метода web3.eth.getTransaction, передав ему в качестве параметра строку хеша транзакции.

В ответ вы получите объект следующего вида:

```
    >
web3.eth.getTransaction("0xb6d13a5e915c3af1feabad7caec7b45348146695973b
    {
                                                        blockHash:
"0x3c9761fefa52a0bc563733d87163828c5fe1316d78ca89be8af18d9c818b0eea",
     blockNumber: 4643,
     from: "0x4f744742ac711fd111c7a983176db1d48d29f413",
     gas: 90000,
     gasPrice: 1000000000,
                                                      hash:
"0xb6d13a5e915c3af1feabad7caec7b45348146695973b32285df287639717e916",
     input: "0x",
     nonce: 0,
                                                        r:
"0x1e3519fbca45cc5f6a0804232c8f0362d42c8abfeaf5225536867651f53787fd",
                                                        s:
"0x69e617eceec461b727a0997fd837264e02242fa16f61491e58974faaf20c49c7",
     to: "0xf212d0180b331a88bd3cafbd77bbd0d56398ae00",
     transactionIndex: 0,
     v: "0xfc2",
     value: 50000000000000000
    }
```

Поля объекта состояния транзакции перечислены в табл. 4.3.

Таблица 4.3. Состояние транзакции

Поле	Тип данных	Описание
hash	String, 32 байта	Хеш транзакции
nonce	Number	Количество транзакций, совершенных отправителем до этой транзакции
blockHash	String, 32 байта	Хеш блока, в котором размещена транзакция, или значение null, если транзакция все еще находится в состоянии ожидания
blockNumber	Number	Номер блока, в котором размещена транзакция, или значение null, если транзакция все еще находится в состоянии ожидания
transactionIndex	Number	Целочисленное значение индекса позиции транзакции в блоке или значение null, если транзакция все еще находится в состоянии ожидания
from	String, 20 байт	Адрес отправителя
to	String, 20 байт	Адрес получателя или null, если это транзакция создания смарт-контракта
value	BigNumber	Количество Wei, переведенных с одного аккаунта на другой в рамках данной транзакции
gasPrice	BigNumber	Стоимость газа в единицах Wei, заданная отправителем
gas	Number	Количество газа, заданное отправителем для выполнения данной транзакции
input	String	Данные, передаваемые вместе с транзакцией
v, r, s	String, 32 байта	Значения для подписи транзакции

Как видите, здесь есть адреса отправителя **from** и получателя **to**, а также объем переведенных средств в Wei.

Когда вы запускаете транзакцию на выполнение, она выполняется не сразу. Вначале транзакция находится в состоянии ожидания, пока майнеры не создадут для нее новый блок. Анализируя поля номера блока **blockNumber**, в котором размещается транзакция, можно определить, была ли запущена транзакция или еще нет.

Поля v, r, s содержат значения для подписи транзакции. Их содержимое можно использовать для получения публичного ключа аккаунта. Способ их использования с этой целью обсуждается здесь: https://ethereum.stackexchange.com/questions/13778/get-public-key-of-any-ethereum-account.

Содержимое полей **gasPrice** и **gas** имеет отношение к стоимости транзакции. Пока мы работаем с тестовой сетью, об этом можно не беспокоиться. Если кратко, то **gasPrice** содержит так называемую стоимость газа, который тратится на выполнение транзакции, а поле **gas** – количество газа, которое выделил отправитель для выполнения транзакции.

Вы можете думать о газе, как о бензине, который тратится на выполнение транзакций. Стоимость этого бензина может меняться, а на разные транзакции этого бензина тратится разное количество.

Квитанция транзакции

Когда транзакция выполнилась (о чем можно узнать при помощи только что рассмотренного метода web3.eth.getTransaction), вы можете получить так называемую квитанцию об ее выполнении. Для этого предназначен метод web3.eth.getTransactionReceipt.

Передайте этому методу в качестве параметра хеш завершенной транзакции, и вы получите квитанцию в виде объекта:

```
    >
web3.eth.getTransactionReceipt("0xb6d13a5e915c3af1feabad7caec7b45348146
    {
                                                          blockHash:
"0x3c9761fefa52a0bc563733d87163828c5fe1316d78ca89be8af18d9c818b0eea",
     blockNumber: 4643,
     contractAddress: null,
     cumulativeGasUsed: 21000,
     from: "0x4f744742ac711fd111c7a983176db1d48d29f413",
     gasUsed: 21000,
     logs: [],
                                                          logsBloom:
"0x00000000000000000000000000000000000000000000000000000000000000000
                                                             root:
"0xa5cdcd909d837e937189f4cffc52840111f4430bf871f07094eaea96c47d682d",
     to: "0xf212d0180b331a88bd3cafbd77bbd0d56398ae00",
                                                    transactionHash:
"0xb6d13a5e915c3af1feabad7caec7b45348146695973b32285df287639717e916",
     transactionIndex: 0
    }
```

Поля объекта квитанции перечислены в табл. 4.4.

Таблица 4.4. Квитанция выполненной транзакции

Поле	Тип данных	Описание
blockHash	String, 32 байта	Хеш блока, в котором размещена транзакция
blockNumber	Number	Номер блока, в котором размещена транзакция
transactionHash	String, 32 байта	Хеш транзакции
transactionIndex	Number	Целочисленное значение индекса позиции транзакции в блоке
from	String, 20 байт	Адрес отправителя
to	String, 20 байт	Адрес получателя или null, если это транзакция создания смарт-контракта
cumulativeGasUsed	Number	Общее количество газа, использованное при выполнении транзакции в блоке
gasUsed	Number	Количество газа, использованное только этой конкретной транзакцией
contractAddress	String - 20 байт	Адрес созданного смарт-контракта, если транзакция использовалась для создания смарт-контракта, или значение null
logs	Array	Массив объектов журнала, созданного данной транзакцией
logsBloom	String, 256 байт	Так называемый цветной фильтр, используется для экономии места при хранении журналов, а также для поиска журнала нужного смарт-контракта
status	String	Если в этом поле находится значение "0x1", это означает, что транзакция выполнена успешно. При ошибке здесь находится значение "0x0"
root	String, 32 байта	Хеш корня дерева состояний

Анализируя квитанцию, вы можете узнать номер блока, в котором размещена транзакция **blockNumber**, а также индекс позиции размещения транзакции в блоке **transactionIndex**.

При помощи полей **cumulativeGasUsed** и **gasUsed** вы сможете понять, сколько газа ушло на выполнение транзакции. При запуске транзакции вы можете выделить определенное количество газа, но если газа будет больше, израсходуется только часть ваших средств. В реальной сети Ethereum, если газа будет выделено слишком мало, транзакция будет очень долго находиться в состоянии ожидания или не будет выполнена вовсе.

Также обратите внимание на поле **contractAddress**. Когда мы займемся публикацией смарт-контрактов, нам потребуется адрес размещенного смарт-контракта из этого поля.

Итоги урока

На четвертом уроке вы научились управлять аккаунтами, а также переводить средства с одного аккаунта на другой. Попутно вы изучили криптовалютные единицы, которые применяются в сети Ethereum.

Вы узнали, что созданные транзакции сначала попадают в состояние ожидания и находятся в нем до тех пор, пока в блокчейн не будет добавлен новый блок. Проверку состояния транзакции можно выполнить методом web3.eth.getTransaction.

Когда транзакция выполнена, то с помощью метода web3.eth.getTransactionReceipt вы можете получить ее квитанцию. Из квитанции можно определить, сколько газа было потрачено на выполнение транзакции. Если транзакция была связана с публикацией контракта, то этот метод поможет вам получить адрес опубликованного контракта. Адрес контракта необходим, как вы скоро узнаете, для вызова его методов.

Урок 5. Публикация первого контракта

Цель урока: на этом уроке вы узнаете о смарт-контрактах в сети Ethereum и о том, как они выполняются виртуальной машиной Ethereum. Мы расскажем, как создать смарт-контракт на языке Solidity и как опубликовать его в приватной сети. Также вы научитесь устанавливать и запускать пакетный компилятор solc. Вам нужно узнать о так называемом бинарном интерфейсе приложения Application Binary Interface (ABI) и научится его использовать.

Практические задания: на пятом уроке вы создадите первый контракт HelloSol в среде разработки Remix Solidity IDE. Этот контракт вам предстоит опубликовать в приватной сети и проверить в работе. Также вы установите пакетный компилятор solc и откомпилируете им свой смарт-контракт HelloSol.

Как вы, конечно, знаете, обычные контракты заключаются между двумя (или несколькими сторонами) в бумажном виде. Например, в контракте может быть написано, что одна сторона берет на себя обязательство поставить какой-нибудь товар, а другая обязуется после получения товара выполнить оплату. Либо контракт может предусматривать предоплату, и тогда товар поставляется после получения денег.

В контрактах также обычно предусматривают некие штрафные санкции за невыполнение условий. Это может быть, например, штраф за задержку поставки или оплаты. Если условия контракта нарушаются, пострадавшая сторона может обратиться в суд, при этом в дело включатся юристы. Вся эта процедура может отнять немало времени и средств.

Смарт-контракты в Ethereum

В сети Ethereum имеется возможность создавать так называемые смарт-контракты (Smart Contracts). Смарт-контракт представляет собой программный код, работающий в среде виртуальной машины Ethereum. Он запускается на всех узлах сети, и результаты его работы также реплицируются на все узлы.

С помощью смарт-контрактов очень удобно отслеживать выполнение транзакций, например, имеющих отношение к поставке товаров (особенно электронных, таких как файлы книг или доступы к сервисам), автоматизируя оплату при помощи криптовалютных средств. Если смарт-контракт получил тем или иным способом подтверждение выполнения условий сделки, то он может сам, автоматически, перевести средства поставщику.

Если условия сделки были выполнены не полностью или не выполнены вовсе, смарт-контракт может вернуть средства покупателю или перевести сумму штрафа на счет пострадавшей стороны.

Смарт-контракт может хранить данные, например, значения баланса, флаги, строки и числа, идентификаторы документов, загруженных в Ethereum, таких как сканы прав владения чем-либо, выписки и т.п. Поэтому смарт-контракты могут применяться для заверения документов вместо услуг обычных нотариусов.

Одной особенностью смарт-контракта является то, что его нельзя оспорить. Если логика смарт-контракта сработает таким образом, что средства будут переведены, эти средства уже невозможно будет вернуть.

Достоинством смарт-контрактов является то, что на их работу требуется очень мало средств. Сравните это с оплатой юристов и различного рода пошлин в суде. Поэтому смарт-контракты можно использовать для поддержки очень недорогих сделок.

Но вот разработка смарт-контракта может потребовать определенных затрат, в том числе и на юристов, с которыми имеет смысл согласовать логику работы смарт-контракта.

Кроме того, если с обычными контрактами при возникновении проблем вам могут помочь юристы и суды, то результаты выполнения смарт-контрактов будет очень сложно оспорить, если вообще возможно.

Выполнение смарт-контракта

Код смарт-контрактов составляют на языке программирования Solidity, напоминающем JavaScript. Больше всего код смарт-контракта похож на определение класса в объектно-ориентированных языках программирования – там есть свойства, методы, конструктор.

Перед публикацией смарт-контракта в сети Ethereum его необходимо компилировать в байт-код. Далее этот код сохраняется в сети с помощью транзакции. На этом уроке мы выполним такие действия.

Когда контракт опубликован, становится доступен его адрес (похожий на адрес аккаунта). С помощью этого адреса децентрализованные приложения DApp (и любые другие приложения) могут вызывать методы контракта.

Виртуальная машина Ethereum

Программный код смарт-контрактов выполняется в песочнице на виртуальной машине Ethereum (EVM, Ethereum Virtual Machine). Песочница полностью изолирует смарт-контракт, запрещая ему прямой доступ к оборудованию и программным интерфейсам узлов Ethereum.

Таким образом, смарт-контракт не может, например, читать и писать файлы, выполнять передачу данных по сети, вводить данные с клавиатуры или отображать их на экране.

При необходимости смарт-контракты могут вызывать другие смарт-контракты (далее просто контракты), однако такие вызовы могут привести к проблемам с безопасностью.

Если смарт-контракту нужно получать данные из-каких либо внешних сервисов, например, с Web-сайтов или от аппаратных датчиков, то напрямую это невозможно. На 12-м уроке мы расскажем, как смарт-контракты могут обмениваться данными с объектами реального мира с помощью так называемых оракулов – информационных посредников.

Интегрированная среда разработки Remix Solidity IDE

Изучение основ языка смарт-контрактов Solidity, а также отладку первых проектов очень удобно делать в интегрированной среде разработки Remix Solidity (Integrated Development Environment, IDE). Для того чтобы приступить к работе в Remix Solidity IDE, вам не нужно ничего устанавливать. Просто откройте браузер и загрузите в него сайт http://remix.ethereum.org.

На экране появится главная страница Remix Solidity IDE, в левой части которой будет показан исходный код контракта ballot.sol (рис. 5.1). Этот контракт реализует возможность голосования, и пока мы не будем его рассматривать.

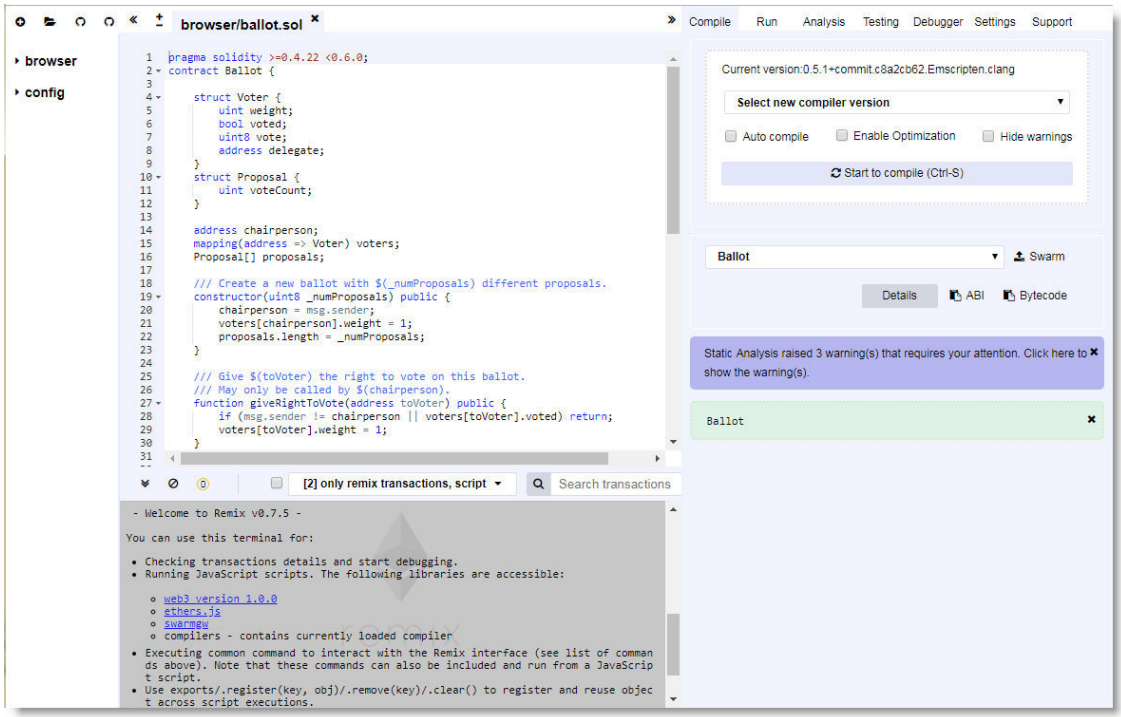

Рис. 5.1. Главная страница Solidity IDE

Щелкните кнопку в виде круга со знаком «+» внутри, и в появившемся на экране диалоговом окне введите имя файла для нашего первого контракта HelloSol.sol (расширение имени для программ Solidity должно быть sol).

Далее скопируйте исходный код контракта HelloSol.sol (листинг 5.1.) в окно, где раньше был исходный текст контракта ballot.sol.

Листинг 5.1. Контракт HelloSol.sol

```
pragma solidity ^0.5.0;

contract HelloSol {
   string savedString;
   uint savedValue;

   function setString( string memory newString ) public {
       savedString = newString;
   }
```

```
    function getString() public view returns( string memory ) {
        return savedString;
    }
    function setValue( uint newValue ) public {
        savedValue = newValue;
    }
    function getValue() public view returns( uint ) {
        return savedValue;
    }
}
```

Этот контракт позволяет хранить текстовую строку и числовое значение. С помощью методов setString и getString можно соответственно записывать и читать строки. Аналогичные методы setValue и getValue предусмотрены в нашем контракте для числовых значений.

Обратите внимание, что в первой строке контракта pragma solidity мы указали версию Solidity как ^0.5.0. Таким способом мы сообщаем, что для этого файла исходного текста нужен компилятор Solidity версии не ранней, чем 0.5.0. Символ ^ означает, что для компиляции нельзя использовать версию 0.6.0.

Язык Solidity, как и все, имеющее отношение к Ethereum, стремительно развивается, так что следите за версиями.

Ключевое слово contract определяет класс контракта. В нашем случае контракт называется HelloSol, а файл исходного текста контракта должен называться HelloSol.sol.

Поле savedString типа string будет хранить значение текстовой строки, а поле savedValue типа uint – целочисленное значение. Позже мы расскажем о типах данных в Solidity подробнее.

Запуск компиляции

Для запуска компиляции нашего файла HelloSol.sol щелкните кнопку **Start to compile**, расположенную сверху в правой части окна Remix Solidity IDE.

Обратите внимание, что в правой части окна Remix Solidity IDE при компиляции появился блок с сообщением о наличии предупреждений, выделенный фиолетовым цветом (рис. 5.2.).

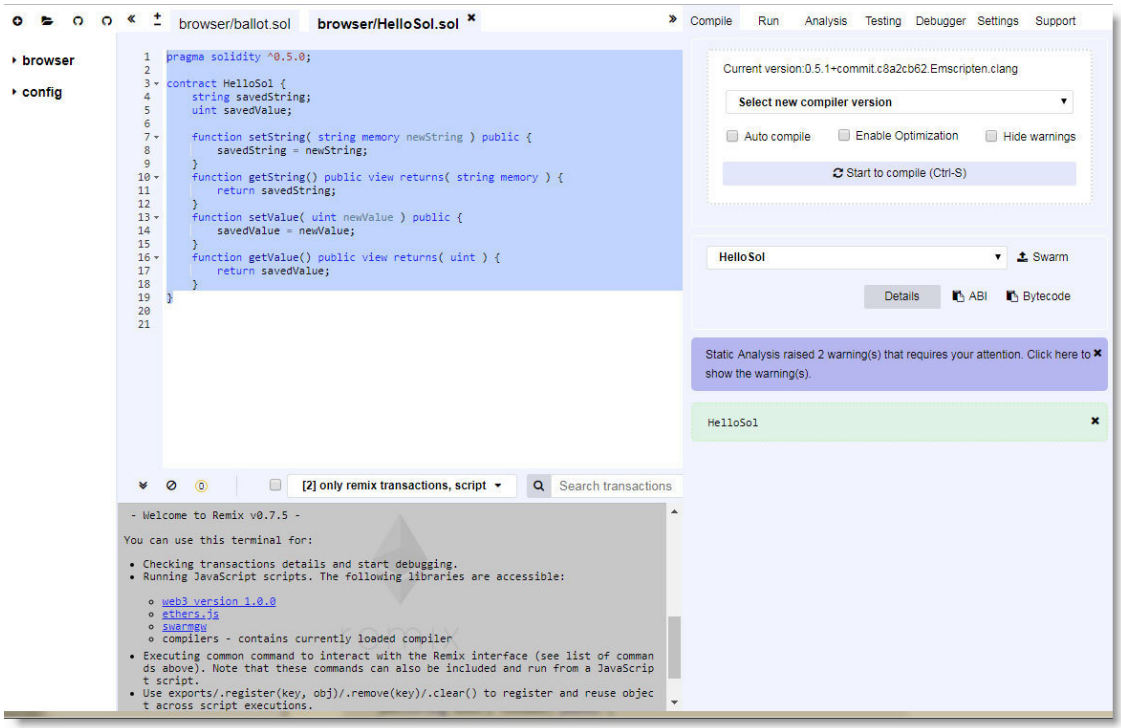

Рис. 5.2. Предупреждения при компиляции

Щелкните этот блок, чтобы открыть вкладку **Analysis**, и пролистайте его вниз. Там вы увидите текст предупреждений:

```
Gas requirement of function HelloSol.getString() high: infinite. If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
```

Аналогичное предупреждение будет выведено и для функции setString.

В данном случае нас предупреждают о невозможности определения количества газа, которое потребуется для выполнения функций HelloSol.getString() и HelloSol.setString(). Это потому, что заранее не известен размер строк, возвращаемых данными функциями.

Аналогичное предупреждение выводится и для функции HelloSol.getString(), по причине того, что неизвестно, какого размера строку нужно будет сохранить в блокчейне.

Сейчас, на этапе запуска нашего первого контракта, можно проигнорировать эти предупреждения. В рабочих контрактах вы, конечно, должны оптимизировать функции таким образом, чтобы они при выполнении расходовали как можно меньше газа.

Вызов функций контракта

Теперь, когда мы выполнили компиляцию исходного текста нашего контракта, давайте опубликуем его в тестовой сети и займемся вызовом функций контракта.

На вкладке Run щелкните кнопку **Deploy**. В окне Solidity IDE появится адрес опубликованного контракта. Это строка **HelloSol at 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a**, она сокращена, но может быть скопирована полностью соответствующей кнопкой.

Щелкните упомянутую только что строку, чтобы увидеть имена функций, определенных в нашем контракте (рис. 5.3.).

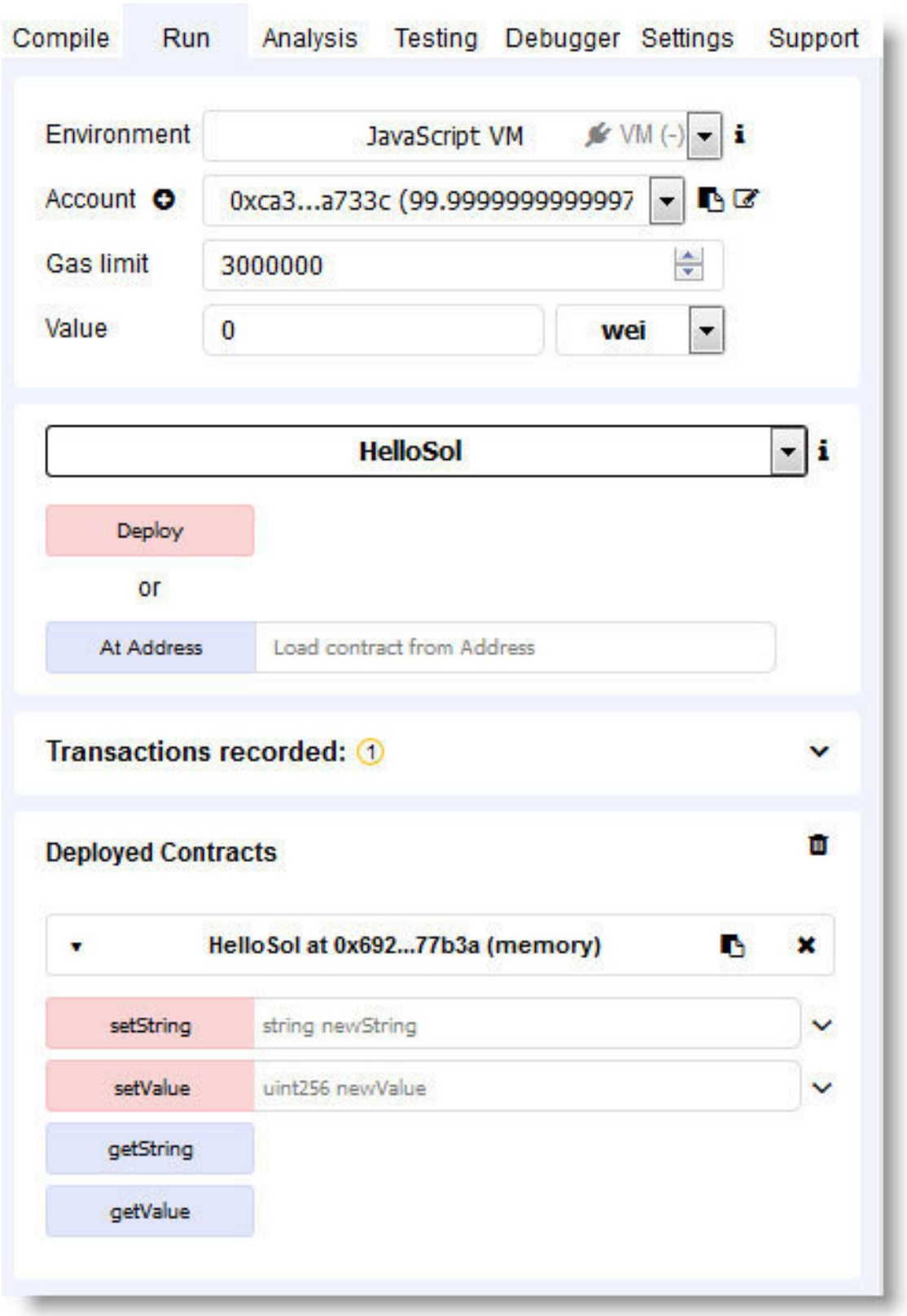

Рис. 5.3. Контракт опубликован

Теперь вы можете попробовать вызывать функции, передавая им параметры. Чтобы сохранить текстовую строку, введите ее в кавычках в поле справа от кнопки **setString**, а затем щелкните эту кнопку. Выполните аналогичное действие с кнопкой **setValue**, но теперь нужно ввести любое целочисленное значение.

Далее щелкните по очереди кнопки **getString** и **getValue**, и вы увидите сохраненную строку и число соответственно (рис. 5.4.).

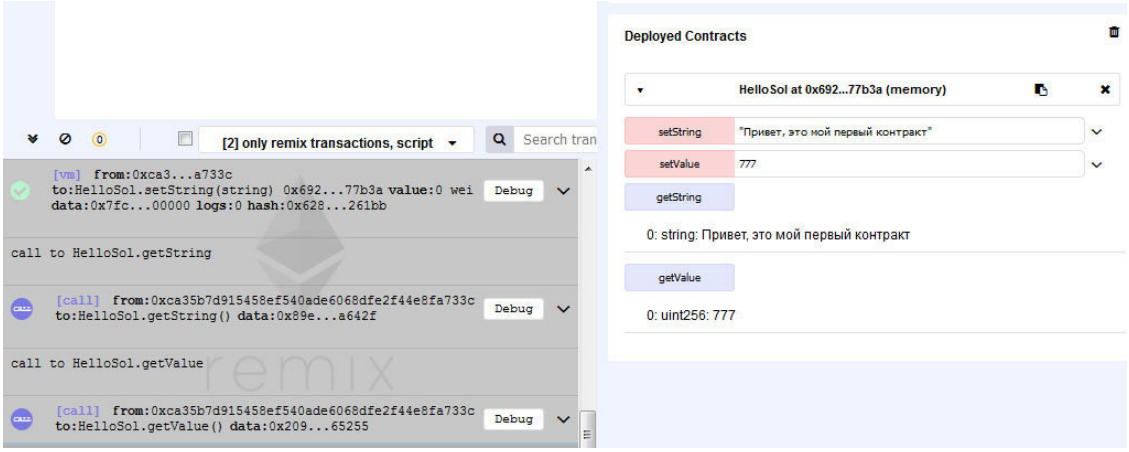

Рис. 5.4. Результаты вызова методов контракта

Обратите внимание, что в левой части окна отображается информация о выполненных транзакциях. Она может помочь при отладке контракта.

Подробную документацию о работе с Remix Solidity IDE вы найдете здесь: https://remix.readthedocs.io/en/latest/index.html.

Публикация контракта в приватной сети

На втором и третьем уроках мы создавали свою приватную сеть Ethereum, состоящую из одного узла. Теперь мы опубликуем наш контракт HelloSol.sol в этой приватной сети и будем работать с ним из консоли Geth.

При компиляции исходного текста контракта Solidity создается не только бинарный файл программы, но и так называемый бинарный интерфейс приложения Application Binary Interface (ABI).

Для запуска контракта в нашей приватной сети потребуется и то, и другое.

Интерфейс ABI представляет собой спецификацию параметров и возвращаемых значений методов контракта. Подробнее об этом можно почитать здесь: https://solidity.readthedocs.io/en/develop/abi-spec.html.

Интерфейс ABI и бинарный код программы на этом уроке мы получим при помощи все того же Remix Solidity IDE.

Получаем определение ABI и двоичный код контракта

Чтобы получить ABI и двоичный код, после компиляции на вкладке **Compile** щелкните кнопку **Details**. Прокрутив содержимое появившегося окна вниз, найдите блок **WEB3DEPLOY** (рис. 5.5.).

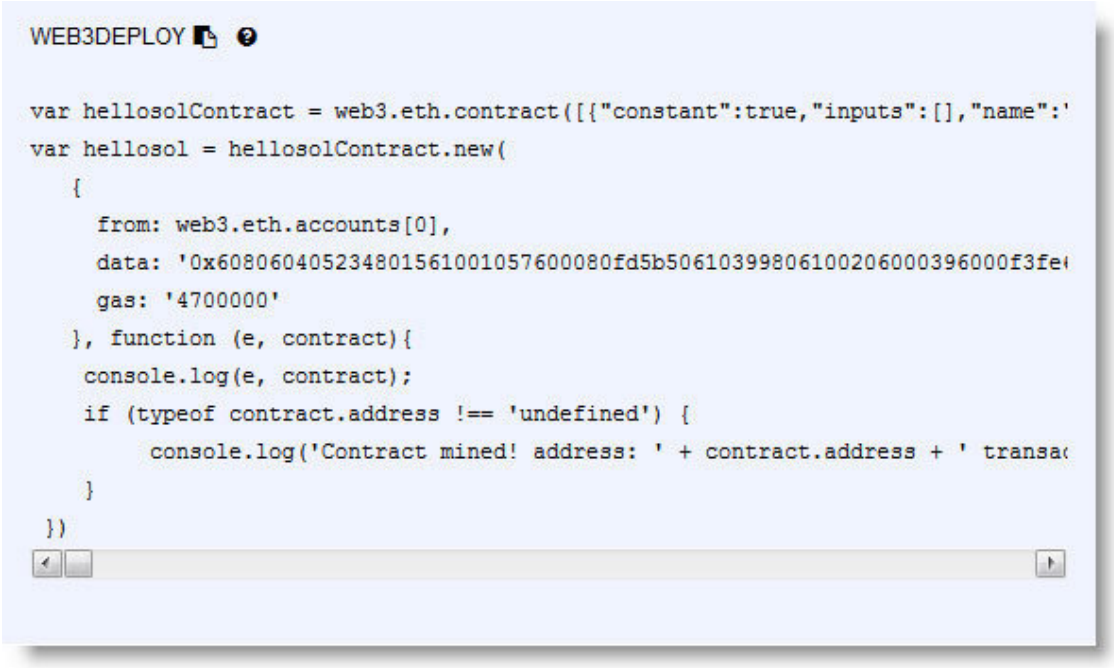

Рис. 5.5. Блок WEB3DEPLOY

Скопируйте отсюда блок кода при помощи кнопки **Copy value to clipboard** и вставьте этот текст в какой-нибудь текстовый редактор, например, в редактор Sublime или Vim. Вот что вы увидите (мы сократили байт-код в поле data для лучшей читаемости текста):

```
    var                    hellosolContract                    =
web3.eth.contract([{"constant":true,"inputs":
[],"name":"getValue","outputs":
```

```
[{"name":"","type":"uint256"}],"payable":false,"stateMutability":"view"
{"constant":false,"inputs":
[{"name":"newValue","type":"uint256"}],"name":"setValue","outputs":
[],"payable":false,"stateMutability":"nonpayable","type":"function"},
{"constant":false,"inputs":
[{"name":"newString","type":"string"}],"name":"setString","outputs":
[],"payable":false,"stateMutability":"nonpayable","type":"function"},
{"constant":true,"inputs":[],"name":"getString","outputs":
[{"name":"","type":"string"}],"payable":false,"stateMutability":"view",
    var hellosol = hellosolContract.new(
      {
        from: web3.eth.accounts[0],
                                                              data:
'0x6080604052348015610010576000800fd5b506103998061002060003960000f3fe6080
    ...
    …    10161034e565b5090565b9056fea165627a7a72305820ea427b1809a23dd1a
        gas: '4700000'
      }, function (e, contract){
       console.log(e, contract);
       if (typeof contract.address !== 'undefined') {
                                  console.log('Contract    mined!
address:   '   +   contract.address   +   '   transactionHash:   '
+ contract.transactionHash);
       }
    })
```

Скопированный код представляет собой программу JavaScript. Эта программа вызывает метод web3.eth.contract, предназначенный для создания объекта контракта.

В качестве параметра методу web3.eth.contract передается массив ABI, в котором определены функции нашего контракта.

В том же окне **Details** есть вкладка ABI, где вы можете посмотреть объект ABI в более читаемом виде. На рис. рис. 5.6. мы показали этот блок в сокращенном виде, только для функций setString() и setValue().

Рис. 5.6. Блок ABI

Как видите, для каждой функции в ABI содержатся ее имя, определение типов входных и выходных параметров и другие данные, которыми мы займемся позже.

Публикация контракта

Прежде чем опубликовать контракт, необходимо разблокировать аккаунт при помощи функции web3.personal.unlockAccount:

```
> web3.personal.unlockAccount("0x4f744742ac711fd111c7a983176db1d48d29f413","**********")
true
```

Мы уже выполняли эту операцию на 4-м уроке, когда переводили средства с одного аккаунта на другой.

Для того чтобы опубликовать контракт, вставьте в консоль Web3 вашего приватного блокчейна блок кода, скопированный из блока **WEB3DEPLOY**, и затем нажмите клавишу Enter.

Как только вы это проделаете, то увидите в консоли следующее:

```
    null [object Object]
    undefined
    > null [object Object]
```

Это нормально. Теперь нужно подождать, когда контракт опубликуется. Если все сделано правильно, через несколько минут вы увидите в консоли сообщение:

```
    Contract                    mined!                    address:
0x1c766c3cd0114771c0e9e45e54f4924b3f81f0d8      transactionHash:
0xcf6af22b774045e7730cd8862d9bef36b1d05dac46c57a491eea2a386377e97e
```

Теперь ваш контракт опубликован в вашей приватной сети, и ему присвоен адрес `0x1c766c3cd0114771c0e9e45e54f4924b3f81f0d8`. Пользуясь этим адресом, вы сможете обращаться к контракту, вызывая его функции.

Пока вы ожидаете публикации контракта, загляните в первое окно, в котором вы запустили узел сети. Там вы увидите сообщение о запуске транзакции, публикующей контракт, хеш этой транзакции и адрес контракта:

```
    NFO       [02-18|03:23:22.740]       Submitted       contract
fullhash=0xcf6af22b774045e7730cd8862d9bef36b1d05dac46c57a491eea2a386377e97e
contract=0x1C766C3CD0114771C0e9E45e54F4924B3F81f0d8
```

Проверка состояния транзакции публикации контракта

Теперь проверьте состояние транзакции функцией web3.eth.getTransaction, передав ей хеш транзакции:

```
    >
web3.eth.getTransaction("0xcf6af22b774045e7730cd8862d9bef36b1d05dac46c5
    {
                                                          blockHash:
"0x86332e89d7018dbe7f61e395e02e805f070e2eb8fc352a985351f172776b0505",
     blockNumber: 5430,
     from: "0x4f744742ac711fd111c7a983176db1d48d29f413",
     gas: 4700000,
     gasPrice: 1000000000,
                                                            hash:
"0xcf6af22b774045e7730cd8862d9bef36b1d05dac46c57a491eea2a386377e97e",
                                                          input:
"0x60806040523480156100105760008fd5b506103998061002060003960006f3fe6080
     nonce: 1,
                                                              r:
"0x75a67c3579640d25dfc13323a0940f33956513db2b1996907679a9c00dbea0dc",
                                                              s:
"0x665cca9e3aaf9e1b71343a00f8c2d9b5d2704adfc2be2537afe173ca52446428",
     to: null,
     transactionIndex: 0,
     v: "0xfc2",
     value: 0
    }
```

Здесь видно, что транзакция завершена и размещена в блоке с номером 5430.

Теперь получите квитанцию на транзакцию по ее хешу:

```
    >
web3.eth.getTransactionReceipt("0xcf6af22b774045e7730cd8862d9bef36b1d05
    {
                                                          blockHash:
"0x86332e89d7018dbe7f61e395e02e805f070e2eb8fc352a985351f172776b0505",
     blockNumber: 5430,
    contractAddress:
"0x1c766c3cd0114771c0e9e45e54f4924b3f81f0d8",

     cumulativeGasUsed: 297174,
     from: "0x4f744742ac711fd111c7a983176db1d48d29f413",
     gasUsed: 297174,
     logs: [],
                                                          logsBloom:
"0x00000000000000000000000000000000000000000000000000000000000000000
                                                               root:
"0x8de6cde102bd347b60bd5c53fce668179a13e2f36905cb30c3c755c4ce97aeba",
     to: null,
                                                   transactionHash:
"0xcf6af22b774045e7730cd8862d9bef36b1d05dac46c57a491eea2a386377e97e",
     transactionIndex: 0
    }
```

Из квитанции можно узнать адрес опубликованного контракта: 0x1c766c3cd0114771c0e9e45e54f4924b3f81f0d8. Этот адрес будет нам нужен для вызова функций контракта.

Вызов функций контракта

Теперь займемся самым интересным – взаимодействием с контрактом. Мы будем вызывать его методы из консоли Geth, используя API JavaScript Web3.

Прежде всего в консольном приглашении Geth введите следующую строку:

```
    var hellosolContract =
web3.eth.contract([{"constant":true,"inputs":
[],"name":"getValue","outputs":
[{"name":"","type":"uint256"}],"payable":false,"stateMutability":"view"
{"constant":false,"inputs":
[{"name":"newValue","type":"uint256"}],"name":"setValue","outputs":
[],"payable":false,"stateMutability":"nonpayable","type":"function"},
{"constant":false,"inputs":
[{"name":"newString","type":"string"}],"name":"setString","outputs":
[],"payable":false,"stateMutability":"nonpayable","type":"function"},
{"constant":true,"inputs":[],"name":"getString","outputs":
[{"name":"","type":"string"}],"payable":false,"stateMutability":"view",
```

Здесь на основе ABI будет создан объект нашего контракта. Вызовем для него метод at, передав этому методу в качестве параметра адрес контракта 0x1c766c3cd0114771c0e9e45e54f4924b3f81f0d8, полученный нами на шаге публикации контракта:

```
    > var HelloSol =
hellosolContract.at("0x1c766c3cd0114771c0e9e45e54f4924b3f81f0d8")
```

Теперь можно вызывать методы контракта. Как и ожидается, метод getValue возвращает нулевое значение, т.к. мы еще не сохраняли в базе контракта никаких значений:

```
    > HelloSol.getValue()
    0
```

Перед тем как вызывать метод setValue, нам необходимо разблокировать аккаунт при помощи метода unlockAccount:

```
    >
web3.personal.unlockAccount("0x4f744742ac711fd111c7a983176db1d48d29f413
    true
```

Если вы помните, ранее мы разблокировали аккаунт перед публикацией контракта.

После того как аккаунт разблокирован, вызываем метод setValue:

```
    > HelloSol.setValue(777, {from:
"0x4f744742ac711fd111c7a983176db1d48d29f413"})
    "0x429eb0bbbc50cca55c8446adca8bdd5ed4c7df6fdc32930f7e14ce7ccb51ea51
```

Обратите внимание, что в качестве первого параметра мы передаем методу значение, которое должно быть сохранено, а в качестве второго – адрес аккаунта, от имени которого будет вызван этот метод.

Напомним, что список адресов аккаунтов можно получить в консоли Geth командой web3.eth.accounts.

В первом окне, где запущен узел сети, появится сообщение о запуске транзакции, вызвавшей фукнцию setValue(), хеш этой транзакции и адрес контракта:

```
    INFO [02-18|03:35:26.877] Submitted
fullhash=0x4a9eb0bbb50cca55c8446adca8bdd5ed4c7df6fdc32930f7e14ce7ccb51ea51
    recipient=0x1C766C3CD0114771C0e9E45e54F4924B3F81f0d8
```

Теперь попытаемся получить значение методом getValue:

```
    > HelloSol.getValue()
    0
    > HelloSol.getValue()
    777
```

Вначале этот метод будет возвращать нулевое значение, и только через некоторое время, когда в сеть будет добавлен новый блок, мы получим наше новое значение.

Аналогичный эксперимент мы можем провести и с методами getString и setString, которые извлекают из базы контракта и изменяют в базе текстовую строку соответственно:

```
    > HelloSol.getString()
    ""
    >
web3.personal.unlockAccount("0x4f744742ac711fd111c7a983176db1d48d29f413
    Passphrase:
    true
    >       HelloSol.setString("Моя       строка",       {from:
"0x4f744742ac711fd111c7a983176db1d48d29f413"})
    "0xaecf9112e7ae36f4f28ccd36716538760672d3936620db7c05bf994ddcc6a024
    > HelloSol.getString()
    ""
    > HelloSol.getString()
    ""
    > HelloSol.getString()
    "Моя строка"
```

Здесь в качестве второго параметра мы передали функции setString структуру, в поле from которой находится адрес аккаунта, вызвавшего данную функцию.

Метод getString вернет новое значение после успешного майнинга нового блока.

Пакетный компилятор solc

Небольшие контракты очень удобно отлаживать при помощи Solidity IDE. Однако есть и альтернатива – пакетный компилятор solc, который можно запускать в командной строке. Документация по этому компилятору находится здесь: http://solidity.readthedocs.io/en/develop/using-the-compiler.html.

Установка solc в Ubuntu

Для установки solc в Ubuntu достаточно ввести одну команду:

```
$ sudo snap install solc
[sudo] password for book:
solc v0.5.7 from Ethereum Build Automation (builds-c) installed
```

Вы можете проверить версию установленного компилятора solc:

```
$ solc -version
solc, the solidity compiler commandline interface
Version: 0.5.7+commit.6da8b019.Linux.g++
```

Установка solc в Debian

Для установки solc в ОС Debian тоже можно использовать snap. Но сначала вам нужно установить демон snapd:

```
$ sudo apt install snapd
$ sudo snap install solc
```

После установки добавьте путь /snap/bin в переменную $PATH:

```
$ echo "export PATH=$PATH:/snap/bin " >> ~/.bashrc
$ source ~/.bashrc
```

Проверяем, что переменные окружения установлены:

```
$ printenv | grep snap
```

Теперь можно узнать версию компилятора solc:

```
$ solc -version
solc, the solidity compiler commandline interface
Version: 0.5.7+commit.6da8b019.Linux.g++
```

Компиляция контракта HelloSol

Теперь мы выполним компиляцию нашего контракта HelloSol, получив для него файлы, содержащие ABI и двоичный код. Запишите исходный текст контракта в файл HelloSol.sol (листинг 5.1), а затем выполните следующую команду:

```
$ solc –bin –abi HelloSol.sol –o build –overwrite
Compiler run successful. Artifact(s) can be found in directory build.
```

После ее выполнения в подкаталоге build текущего каталога будут созданы два файла – HelloSol.abi и HelloSol.bin.

```
$ ls –1 build
HelloSol.abi
HelloSol.bin
```

Первый из этих файлов содержит спецификацию интерфейса контракта, ABI, а второй – двоичный код откомпилированного контракта.

Публикация контракта

Для публикации контракта используйте приведенный выше код из Solidity IDE. Подставьте в него результат компиляции – содержимое файлов HelloSol.abi и HelloSol.bin.

Обязательно проследите за тем, чтобы при копировании не было символов перевода строки внутри содержимого этих файлов.

Конечно, такая процедура публикации контракта и вызова его функций достаточно трудоемкая. Мы проделали ее для того, чтобы вы смогли понять, как все это работает. Далее на уроке 7 мы рассмотрим такой инструмент, как Truffle. С его помощью компиляция, отладка и публикация смарт-контрактов выполняются намного удобнее.

Установка solc на Rasberian

Установку solc на Rasberian придется выполнять из исходных текстов. К сожалению, это довольно длительная процедура. Ее описание вы можете найти здесь: https://media.consensys.net/installing-ethereum-compilers-61d701e78f6

Прежде всего установите необходимые зависимости:

```
$ sudo apt install -y build-essential cmake libboost-all-dev
```

Далее загрузите из репозитория исходные тексты Solidity:

```
$ git clone -recursive https://github.com/ethereum/solidity.git
$ cd solidity
```

Создайте каталог для сборки и сделайте его текущим:

```
$ mkdir build && cd build
```

Создайте файлы конфигурации, необходимые для сборки:

```
$ cmake ..
```

Запустите процесс компиляции (на Raspberry Pi 3 будет идти довольно долго):

```
$ make
```

Запустите установку solc в системный каталог, используя sudo:

```
$ sudo make install
```

Добавьте путь /usr/local/lib к переменной окружения PATH. Для этого добавьте следующую строку в конец файла ~/.bashrc:

```
export PATH="$PATH:/usr/local/lib"
```

Затем введите команду:

```
source ~/.bashrc
```

Теперь вы можете проверить версию установленного компилятора solc:

```
$ solc -version
solc, the solidity compiler commandline interface
Version: 0.4.25-develop.2018.5.30+commit.3f3d6df2.Linux.g++
```

Можете опубликовать в своей приватной сети описанный выше контракт HelloSol и попробовать вызвать его функции.

Итоги урока

На пятом уроке вы запустили свой первый смарт-контракт в интегрированной среде разработки Remix Solidity. Вы научились вызывать его функции, передавая параметры.

Также вы опубликовали свой контракт в созданной ранее приватной сети Ethereum, вызвали его методы.

Вы установили пакетный компилятор solc и научились с его помощью компилировать смарт-контракты Solidity.

Урок 6. Смарт-контракты и Node.js

Цель урока: на этом уроке вы должны научиться создавать скрипты JavaScript, работающие под управлением Node.js и выполняющие операции со смарт-контрактами Solidity.

Практические задания: нужно будет установить Node.js в ОС Ubuntu, Debian и Rasberian. Вы также напишете скрипты для публикации смарт-контракта в локальной сети Ethereum и вызова его функций. При этом вы сможете проверить работу скриптов с фреймворком Web3 стабильной, а также нестабильной версий. Вы также научитесь переводить с помощью скриптов средства между обычными аккаунтами, а также зачислять их на аккаунты смарт-контрактов.

Для связи смарт-контрактов Solidity с реальным миром мы должны использовать программы, которые обращаются к API узлов Ethereum по сети. Это могут быть программы, работающие через API JSON RPC либо через фреймворки, такие как Web3.

На этом уроке мы научимся работать с узлами Ethereum с помощью скриптов JavaScript, работающих под управлением Node.js. При этом мы будем использовать фреймворк Web3, с которым вы немного знакомы по предыдущим урокам, а также узел Ganache CLI (вместо Geth).

Установка Node.js

Мы установим Node.js в ОС Ubuntu, Debian и Rasberian. Процедуры отличаются.

Установка в Ubuntu

Устанавливать Node.js в ОС Ubuntu нужно с помощью скрипта менеджера управления версиями NVM (Node Version Manager). Эта процедура описана здесь: https://linuxize.com/post/how-to-install-node-js-on-ubuntu-18.04/.

Прежде всего скопируйте скрипт NVM из репозитория Github:

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.34.0/install.sh | bash
```

Закройте и заново откройте консоль либо выполните следующие действия:

• добавьте путь к каталогу .nvm в профиль Bash:

```
export NVM_DIR="$HOME/.nvm"
```

• загрузите nvm:

```
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion"
```

После завершения установки проверьте версию NVM:

```
$ nvm -version
0.33.11
```

Чтобы установить какую-либо конкретную версию Node.js, получите список всех доступных версий:

```
$ nvm ls-remote
```

Теперь установите Node.js следующей командой (здесь мы установили самую новую на момент написания этого руководства версию 10.15.3 с длительной поддержкой LTS):

```
$ nvm install 10.15.3
```

После установки проверьте версии Node.js и npm:

```
$ node -v
v10.15.3

$ npm -v
6.4.1
```

Установка в Debian

Установите Node.js следующим образом:

```
    $ sudo curl -sL https://deb.nodesource.com/setup_8.x | sudo -
E bash -
    $ sudo apt-get install -y nodejs
```

Поле установки проверьте версию установленного Node.js:

```
    $ node -v
    v8.15.0
```

Также установите npm – менеджер пакетов Node.js, и проверьте его версию:

```
    $ sudo apt install npm
    $ npm -v
    6.4.1
```

Далее установите git, если это еще не сделано:

```
    sudo apt-get install git
```

Установка и запуск Ganache-cli

На этом уроке мы будем использовать вместо Geth другое ПО, очень удобное для создания тестового узла Ethereum, а именно Ganache CLI. С его помощью можно очень легко создать персональный блокчейн, удобный для отладки смарт-контрактов, в том числе с помощью такого инструмента, как Truffle.

Установить Ganache CLI очень просто:

```
    $ npm install -g ganache-cli
```

После установки запустите Ganache CLI в отдельном консольном окне при помощи такой команды:

```
    $ ganache-cli
```

Программа создаст при запуске 10 аккаунтов, причем у каждого аккаунта на счету уже будет 100 Ether. Информацию об аккаунтах и вашей приватной сети она выведет на консоль:

```
    Ganache CLI v6.4.1 (ganache-core: 2.5.3)

    Available Accounts

    (0) 0xc0bace993ec94fa15a12327194314f3d7f5512d3 (~100 ETH)
    (1) 0xd8a5722aafd35af59e623f25940eec83e3a68db8 (~100 ETH)
    (2) 0x89f12885a5220d51b10f482077641b7ec0056028 (~100 ETH)
```

```
    (3) 0xe8ab1460f5dbdb651d47385758602c8309871a20 (~100 ETH)
    (4) 0xe39047ad37107ba8121b62effa3320441ef1de03 (~100 ETH)
    (5) 0xcfa907664f08af1152721f434e7919cb6c3e7c8c (~100 ETH)
    (6) 0x85c7e0653260fea923cdb1fac6ff6a7d68671fa7 (~100 ETH)
    (7) 0xbbd61a8744554c33ec3428200d3dd6020d06789b (~100 ETH)
    (8) 0xb1fae37034458b8e02e087306c67c5e4ae540b47 (~100 ETH)
    (9) 0x001b6712ad94b07e7dee1e46c8ef79b35ba01a12 (~100 ETH)

    Private Keys

    (0)
0x1c4d39dc1c1ec842617647e3ac166a6c3c9b7b9f15777bf7ecf85924b283f9d9
    (1)
0x767815291daa28204dcbe8b1b5ceaf91561d5e6fc9535790da5e3257e3209fc9
    (2)
0x2343230b1c6023facb16a51322e7eda232f58dda1f00cc7e03b4c4ba8189d22a
    (3)
0x9a9c8bfaab45f707f026efa2013e36ea18f28f7d2f282c54be6b466093878e87
    (4)
0xd3d12a19f522392be87a302fad106f62964263bbabaf1fa786154d454d02fcb0
    (5)
0x2e7295f07fef91ec0b4be1274219fd561a28d22f23f97a496dca88705f28fbf3
    (6)
0x8b452860a466394fe3188815a269eb7a17e02da65e73655092ae86075a38f01d
    (7)
0x59bd6aa51723eb4a4aeb902c2ac657794c0480eb4c864cceb3220b1cf6532988
    (8)
0xf5fd25f951c151fc725fd7193285ef82a840546e3c295f31eea3b7e83ff37790
    (9)
0xc744da8c004833dbf50bbae4956ea848944f76acf2ea142afd0ebfbff8bc0a55

    HD Wallet

    Mnemonic:        hat loud toddler same forest service train
diamond disagree aim whip habit
    Base HD Path:  m/44'/60'/0'/0/{account_index}

    Gas Price

    20000000000

    Gas Limit

    6721975

    Listening on 127.0.0.1:8545
```

Сейчас нам будут интересны адреса созданных аккаунтов, а также тот факт, что вы можете создать соединение с вашим локальным узлом на порту 8545, как и раньше, когда мы использовали Geth.

Установка Web3

Сначала мы используем Web3 стабильной версии, а в конце этого урока будем использовать нестабильную версию 1.x.

По умолчанию устанавливается нестабильная версия 1.x, поэтому сначала удалим текущую версию, а потом установим нужную нам версию 0.20.6:

```
$ npm uninstall web3
$ npm install web3@0.20.6 -save
```

Для проверки результатов установки запустите консоль Node.js командой node:

```
$ node
>
```

Убедитесь, что у вас запущен узел вашей локальной сети Ethereum на базе Ganache CLI. Далее в консольном приглашении Node.js введите следующие команды:

```
> var Web3 = require('web3')
undefined
> var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:8545"));
undefined
> console.log(web3.eth.accounts);
[ '0x88d7d7aee70b8735a2133d562c711b480ed97a17',
 '0x3729883d60ba3510ca2aaaef6d016bbf1339b8af',
 '0x6cf7f569a4052d6f84d269d99972b16c29b12652',
 '0xbe0944eee29670270213501214ea3e278336cf4b',
 '0x19ca68549517aa2fedeba4b5ee77037bc91d046b',
 '0x5554c63d8d721117ee0370e5b8da5cfdbf9a79a0',
 '0xc3123e9a9a8ae920d09f70edbdfe73eb9dbed7a9',
 '0xb43de04e2f5d2c7fae0fed8413688f5ef01fd3b8',
 '0x20ff14447328e646c553c0a5feef04f83c975a83',
 '0x3424d3b949b8f0469a2df646902506f1c36e1e9a' ]
```

Здесь мы подключились к узлу Ethereum нашей приватной сети с помощью так называемого провайдера Web3.providers.HttpProvider (с использованием протокола HTTP) и посмотрели список аккаунтов, созданных на этом узле.

Если в консоли Node.js появился список аккаунтов, значит, все было установлено правильно.

Установка solc

Проверьте, что у вас установлен solc. Вы должны были это сделать на пятом уроке. Для проверки введите команду:

```
$ solc -version
solc, the solidity compiler commandline interface
Version: 0.5.7+commit.6da8b019.Linux.g++
```

Установка Node.js на Rasberian

Чтобы установить Node.js в ОС Rasberian на Raspberry Pi 3, лучше всего воспользоваться готовыми бинарными сборками для процессора с архитектурой ARM.

Прежде всего определите архитектуру ARM вашего процессора:

```
$ uname -m
armv7l
```

Далее зайдите на сайт https://nodejs.org/en/download/ и найдите ссылку на архив нужной вам архитектуры. На момент создания этого учебного курса там были архивы для ARMv6, ARMv7 и ARMv8. В нашем случае нужен архив для архитектуры ARMv7.

Копируем с сайта соответствующую ссылку, загружаем и распаковываем архив:

```
$ wget https://nodejs.org/dist/v8.11.2/node-v8.11.2-linux-armv7l.tar.xz
$ tar xvf node-v8.11.2-linux-armv7l.tar.xz
```

Далее копируем бинарные программные файлы в каталог /usr/bin/:

```
$ cd node-v8.11.2-linux-armv7l
$ sudo cp -R * /usr/bin/
```

Теперь проверяем версию Node.js и пакетного менеджера npm:

```
$ node -v
v8.11.2
$ npm -v
5.6.0
```

Теперь убедитесь, что у вас запущен узел вашей локальной сети Ethereum, и попробуйте в приглашении Node.js обратиться к локальному узлу вашей приватной сети Ethereum, как мы это сделали в предыдущем разделе этого урока.

Вы должны в итоге получить список аккаунтов, созданных на локальном узле приватной сети Ethereum:

```
$ node
> var Web3 = require('web3')
undefined
> var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
undefined
> console.log(web3.eth.accounts);
[ '0xd902f8405a6108e8bd9343d1bfccf21a081d2897',
```

```
 '0x468da0fa573995e99307f29794b411bf5090e82d',
 '0x8fe467c2a5a8d7a5adc47bb0a2df16683a90f5a5',
 '0x6f3477695e4c13d03f92e08bdc2a3320b2b905a5' ]
```

Чтобы выйти из консоли Node.js, введите команду .exit:

```
> .exit
```

Скрипт для получения списка аккаунтов в консоли

Давайте сделаем первый шаг на пути создания скриптов JavaScript, работающих с узлом нашей сети Ethereum и смарт-контрактами. Мы создадим и запустим простой скрипт, который выводит на консоль ОС список аккаунтов, созданных на локальном узле Ethereum.

Создайте файл list_accounts.js, как показано в листинге 6.1.

Листинг 6.1. Файл list_accounts.js

```
var Web3 = require('web3');
var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:8545"));
console.log(web3.eth.accounts);
```

Для запуска передайте путь к файлу программе node:

```
$ node list_accounts.js
[ '0x88d7d7aee70b8735a2133d562c711b480ed97a17',
 '0x3729883d60ba3510ca2aaaef6d016bbf1339b8af',
 '0x6cf7f569a4052d6f84d269d99972b16c29b12652',
 '0xbe0944eee29670270213501214ea3e278336cf4b',
 '0x19ca68549517aa2fedeba4b5ee77037bc91d046b',
 '0x5554c63d8d721117ee0370e5b8da5cfdbf9a79a0',
 '0xc3123e9a9a8ae920d09f70edbdfe73eb9dbed7a9',
 '0xb43de04e2f5d2c7fae0fed8413688f5ef01fd3b8',
 '0x20ff14447328e646c553c0a5feef04f83c975a83',
 '0x3424d3b949b8f0469a2df646902506f1c36e1e9a' ]
```

Если узел Ethereum запущен, наш скрипт выведет на консоль нужный нам список аккаунтов.

В первом консольном окне должен быть запущен узел Ganache CLI. А во втором консольном окне мы запускаем на выполнение наш скрипт получения списка аккаунтов list_accounts.js.

Заметим, что подобным образом к сети Ethereum может обращаться любой Web-сайт, сделанный с использованием Node.js.

Скрипт для публикации смарт-контракта

Отлаживать контракты, компилируя их с помощью Remix Solidity IDE, довольно неудобно. Давайте сделаем скрипт JavaScript, работающий под управлением Node.js, который займется компиляцией и публикацией контракта.

Для компиляции этот скрипт будет использовать пакетный компилятор solc, а публикация будет осуществляться функциями Web3.

Исходный текст нашего контракта запишем в файл deploy_contract.js (листинг 6.2.).

Листинг 6.2. Файл deploy_contract.js

```
var contract_to_deploy = process.argv[2];
var unlock_password = process.argv[3];

console.log('Contract Deploy script: ' + contract_to_deploy);

var solc_cmd ='solc -bin -abi ' + contract_to_deploy + '.sol
-o build -overwrite';
var abi_path = 'build/' + contract_to_deploy + '.abi';
var bin_path = 'build/' + contract_to_deploy + '.bin';

const { exec } = require('child_process');
exec(solc_cmd, (err, stdout, stderr) => {
 if(err) {
   console.log(`exec error: ${err}`);
   return;
 }
});

var fs = require("fs");
var Web3 = require('web3');
var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:8545"));

if(web3.personal.unlockAccount(web3.eth.coinbase,
unlock_password)) {
   console.log(`${web3.eth.coinbase} is unlocked`);
 }else{
   console.log(`unlock failed, ${web3.eth.coinbase}`);
}

var abi = JSON.parse(fs.readFileSync(abi_path), 'utf8');
//console.log(JSON.stringify(abi, undefined, 2));

var bin = fs.readFileSync(bin_path);

var gasEstimate = web3.eth.estimateGas({data: '0x' + bin});
console.log('gasEstimate = ' + gasEstimate);
```

```
    var contractObject = web3.eth.contract(abi);

    var contractInstance = contractObject.new( {
     from: web3.eth.coinbase, gas: 4700000, data: '0x'+ bin
    }, function (err, newContract) {
        if(!err) {
          if (!newContract.address) {
                      console.log(`newContract.transactionHash  =
${newContract.transactionHash}`);
          } else {
                               console.log(`newContract.address   =
${newContract.address}`);
          }
       } else {
         console.log(err);
       }
    });
```

Рассмотрим работу этого скрипта в деталях.

Запуск и получение параметров

Скрипт запускается из консоли ОС следующим образом:

```
    $ node deploy_contract.js HelloSol ""
    Contract Deploy script: HelloSol
    0x88d7d7aee70b8735a2133d562c711b480ed97a17 is unlocked
    gasEstimate = 344227
    newContract.transactionHash                                    =
0xeb317e6cd9f4571b6d67822a70d2ca9890d85a67697849b1ae6b94ac0b15056b
    newContract.address                                            =
0x5ff460b69b0f01cbbe91af80c37b8fd29ad78abd
```

В качестве первого параметра мы передаем Node.js путь к запускаемому скрипту JavaScript, в нашем случае это deploy_contract.js.

Второй параметр – имя класса смарт-контракта HelloSol, с которым вы уже имели дело. Этот контракт должен находиться в текущем рабочем каталоге, в файле с именем HelloSol.sol.

И наконец, третий параметр – пароль, необходимый для разблокировки основного аккаунта, который будет публиковать контракт. При использовании Ganache CLI пароль нужно задать в виде пустой строки.

После запуска скрипт deploy_contract.js получает параметры запуска, компилирует контракт, разблокирует аккаунт, загружает ABI и бинарный код контракта и оценивает количество газа, необходимое для публикации контракта. Далее выполняется публикация контракта.

Получение параметров запуска

Параметры запуска скрипта deploy_contract.js мы получаем следующим образом:

```
    var contract_to_deploy = process.argv[2];
```

```
var unlock_password = process.argv[3];
```

Компиляция контракта

Следующий шаг – компиляция контракта. В переменной solc_cmd мы сохраняем команду компиляции, а в переменных abi_path и bin_path – путь к результатам компиляции, т.к. к файлу ABI и бинарному файлу контракта:

```
var solc_cmd ='solc -bin -abi ' + contract_to_deploy + '.sol
-o build -overwrite';
var abi_path = 'build/' + contract_to_deploy + '.abi';
var bin_path = 'build/' + contract_to_deploy + '.bin';
```

После этого мы запускаем компилятор на выполнение:

```
const { exec } = require('child_process');
exec(solc_cmd, (err, stdout, stderr) => {
 if(err) {
   console.log(`exec error: ${err}`);
   return;
 }
});
```

Разблокировка аккаунта

Для публикации контракта нам нужно разблокировать аккаунт, который будет публиковать контракт. Мы делаем это с помощью уже знакомой вам функции web3.personal.unlockAccount:

```
if(web3.personal.unlockAccount(web3.eth.coinbase,
unlock_password)) {
   console.log(`${web3.eth.coinbase} is unlocked`);
 }else{
   console.log(`unlock failed, ${web3.eth.coinbase}`);
}
```

Пароль для разблокировки unlock_password мы передаем в качестве параметра строки запуска.

Загрузка ABI и бинарного кода контракта

Теперь наша задача – загрузить из соответствующих файлов ABI и бинарный код контракта:

```
var abi = JSON.parse(fs.readFileSync(abi_path), 'utf8');
//console.log(JSON.stringify(abi, undefined, 2));

var bin = fs.readFileSync(bin_path);
```

При помощи строки, закрытой в этом фрагменте кода символом комментария, вы можете вывести на консоль загруженный ABI в удобном для просмотра виде.

Оценка необходимого количества газа

Функция web3.eth.estimateGas позволяет оценить количество газа, которое нужно для публикации контракта:

```
var gasEstimate = web3.eth.estimateGas({data: '0x' + bin});
console.log('gasEstimate = ' + gasEstimate);
```

Мы выведем эту информацию на консоль.

Создание объекта и запуск публикации контракта

Теперь мы готовы к публикации контракта. Прежде всего необходимо создать объект контракта, передав функции web3.eth.contract объекту ABI, загруженному ранее из файла:

```
var contractObject = web3.eth.contract(abi);
```

Далее мы запускаем процесс создания экземпляра контракта, который будет опубликован в нашей приватной сети Ethereum:

```
var contractInstance = contractObject.new( {
 from: web3.eth.coinbase, gas: 4700000, data: '0x'+ bin
}, function (err, newContract) {
    if(!err) {
      if (!newContract.address) {
                  console.log(`newContract.transactionHash  =
${newContract.transactionHash}`);
      } else {
                           console.log(`newContract.address   =
${newContract.address}`);
      }
    } else {
      console.log(err);
    }
});
```

Конструктору contractObject.new передаются параметры публикации. В поле from мы указываем адрес основного аккаунта нашей сети как web3.eth.coinbase. Далее в поле gas мы разрешаем при публикации контракта использовать достаточно много газа – 4700000 единиц. И наконец, поле data содержит бинарный код контракта, к которому в качестве префикса добавлена строка ‘0x’.

При вызове contractObject.new мы определили функцию обратного вызова с параметрами err и newContract:

```
function (err, newContract) {
    if(!err) {
```

```
          if (!newContract.address) {
                      console.log(`newContract.transactionHash  =
${newContract.transactionHash}`);
          } else {
                                console.log(`newContract.address   =
${newContract.address}`);
          }
       } else {
         console.log(err);
       }
    }
```

Эта функция будет вызвана два раза. В первый раз она получит управление, когда будет получен хеш транзакции transactionHash. На этом этапе контракт еще не опубликован. Второй раз эта функция будет вызвана уже после публикации контракта, когда будет доступен адрес контракта.

Запуск скрипта публикации контракта

После того как вы запустите скрипт публикации контракта, в консоли ОС, где запущен узел Ganache CLI вашей сети, появится сообщение о запуске транзакции:

```
    eth_sendTransaction

                                                        Transaction:
0xeb317e6cd9f4571b6d67822a70d2ca9890d85a67697849b1ae6b94ac0b15056b
     Contract created: 0x5ff460b69b0f01cbbe91af80c37b8fd29ad78abd
     Gas usage: 344227
     Block Number: 1
     Block Time: Thu Apr 04 2019 14:49:03 GMT+0000 (Coordinated
Universal Time)
```

В той же консоли, где вы запускали скрипт публикации, после майнинга новых блоков вы увидите хеш транзакции публикации контракта и адрес опубликованного контракта:

```
    $ node deploy_contract.js HelloSol ""
    Contract Deploy script: HelloSol
    0x88d7d7aee70b8735a2133d562c711b480ed97a17 is unlocked
    gasEstimate = 344227
    newContract.transactionHash                                  =
0xeb317e6cd9f4571b6d67822a70d2ca9890d85a67697849b1ae6b94ac0b15056b
    newContract.address                                          =
0x5ff460b69b0f01cbbe91af80c37b8fd29ad78abd
```

Вызов функций смарт-контракта

В предыдущем разделе мы рассказали, как создать скрипт для компилирования и публикации смарт-контракта. Теперь займемся скриптами, которые умеют вызывать методы опубликованного контракта.

Исходный текст первого из таких скриптов call_contract_set.js вы найдете в листинге 6.3.

Листинг 6.3. Файл call_contract_set.js

```
var contract_name = process.argv[2];
var contract_address = process.argv[3];
var unlock_password = process.argv[4];

console.log('Contract script: ' + contract_name);
var abi_path = 'build/' + contract_name + '.abi';

var fs = require("fs");
var Web3 = require('web3');
var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:8545"));

if(web3.personal.unlockAccount(web3.eth.coinbase,
unlock_password)) {
   console.log(`${web3.eth.coinbase} is unlocked`);
 }else{
   console.log(`unlock failed, ${web3.eth.coinbase}`);
}

var abi = JSON.parse(fs.readFileSync(abi_path), 'utf8');

var HelloSolContract = web3.eth.contract(abi);
var HelloSol = HelloSolContract.at(contract_address);

HelloSol.setValue(444, {from: web3.eth.coinbase});

HelloSol.setString.sendTransaction("My String 1234567",{
  from:web3.eth.coinbase,
  gas:4000000},function (error, result){
    if(!error){
      console.log(result);
    } else{
      console.log(error);
    }
});
```

При запуске скрипту call_contract_set.js нужно передать имя класса контракта, адрес опубликованного контракта, а также пароль для разблокировки аккаунта:

```
    $ node call_contract_set.js HelloSol
0x5ff460b69b0f01cbbe91af80c37b8fd29ad78abd ""
    Contract script: HelloSol
    0x88d7d7aee70b8735a2133d562c711b480ed97a17 is unlocked
    0xe0c1e1b64213ad881078e51b02ee20f1394b09b0ddfab3884cb8688f29beef23
```

Имя класса контракта используется для загрузки ABI из файла, созданного при компиляции описанным ранее скриптом deploy_contract.js.

Так как наш скрипт будет вызывать функции, изменяющие состояние контракта, нам нужно разблокировать аккаунт:

```
    if(web3.personal.unlockAccount(web3.eth.coinbase,
unlock_password)) {
       console.log(`${web3.eth.coinbase} is unlocked`);
     }else{
       console.log(`unlock failed, ${web3.eth.coinbase}`);
    }
```

После этого мы загружаем файл ABI, создаем на его основе объект контракта:

```
    var abi = JSON.parse(fs.readFileSync(abi_path), 'utf8');
    var HelloSolContract = web3.eth.contract(abi);
    var HelloSol = HelloSolContract.at(contract_address);
```

Теперь можно приступить к вызову методов.

Для того чтобы вызвать методы HelloSol.setString и HelloSol.setValue, изменяющие состояние сети, нужно указывать аккаунт, от имени которого будут отправлены транзакции:

```
    HelloSol.setValue(444, {from: web3.eth.coinbase});
```

Для того чтобы транзакции в реальной сети могли быть выполнены, нужно указать количество газа, которое мы готовы потратить на обработку транзакции. Это можно сделать, вызывая методы при помощи sendTransaction, например:

```
    HelloSol.setString.sendTransaction("My String 1234567",{
      from:web3.eth.coinbase,
      gas:4000000},function (error, result){
        if(!error){
          console.log(result);
        } else{
          console.log(error);
        }
    });
```

Обратите внимание, что вызов методов HelloSol.setString и HelloSol.setValue не приведет к немедленному изменению состояния, т.к. необходимо дождаться, пока в сети появятся новые блоки.

Как и при публикации контракта, на консоли, в которой запущен узел сети Ganache CLI, появятся хеши транзакций, отправленных нашим скриптом:

```
    eth_coinbase
    eth_sendTransaction

                                                        Transaction:
0xe0c1e1b64213ad881078e51b02ee20f1394b09b0ddfab3884cb8688f29beef23
     Gas usage: 34079
     Block Number: 10
     Block Time: Thu Apr 04 2019 15:00:13 GMT+0000 (Coordinated
Universal Time)
```

В листинге 6.4. мы привели исходный текст второго скрипта call_contract_get.js, который вызывает функции getValue и getString нашего контракта.

Листинг 6.4. Файл call_contract_get.js

```
    var contract_name = process.argv[2];
    var contract_address = process.argv[3];

    console.log('Contract script: ' + contract_name);
    var abi_path = 'build/' + contract_name + '.abi';

    var fs = require("fs");
    var Web3 = require('web3');
    var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:8545"));

    var abi = JSON.parse(fs.readFileSync(abi_path), 'utf8');

    var HelloSolContract = web3.eth.contract(abi);
    var HelloSol = HelloSolContract.at(contract_address);

    console.log(HelloSol.getValue().toString(10));

    var str = HelloSol.getString();
    console.log(str);
```

Здесь мы вызываем методы, которые не изменяют состояние контракта, а лишь возвращают сохраненные ранее значения, поэтому транзакции не создаются:

```
    console.log(HelloSol.getValue().toString(10));
    console.log(HelloSol.getString());
```

Каждый раз при вызове такой функции в консольном окне Ganache CLI появляются соответствующие сообщения:

```
    eth_call
    eth_call
```

Это очень удобно при отладке ваших программ и контрактов.

Возможно ли обновление опубликованного смарт-контракта

В процессе отладки смарт-контракта вы будете вносить в него всякого рода исправления и дополнения. Но что если ошибка обнаружена уже после публикации контракта? Можно ли ее исправить?

Оказывается, нет, нельзя!

Вспомните, все данные, записанные в блокчейн, хранятся там в неизменном виде неограниченно долго. Это относится и к смарт-контрактам.

Когда вы исправите ошибку или внесете какие-либо изменения в смарт-контракт, его нужно будет опубликовать. При этом новая опубликованная версия получит уже новый адрес, по которому нужно будет вызывать функции вашего смарт-контракта.

Именно поэтому смарт-контракты лучше всего отлаживать в тестовой приватной сети, наподобие той, что сделали мы. Эта процедура быстрая и бесплатная.

Когда смарт-контракт отлажен, его можно опубликовать для окончательной проверки в тестовой сети Rinkeby, о чем мы еще расскажем. И только после этого вы можете публиковать контракт в основной сети Ethereum.

Работа с Web3 версии 1.0.x

Во время работы над этим руководством были доступны только бета-версии фреймворка Web3 1.0.x. Они еще не отлажены и не рекомендуются для использования в коммерческих проектах.

Тем не менее вам будет полезно узнать, какие изменения ожидаются в новом фреймворке. В частности, вместо функций обратного вызова теперь можно использовать промисы (promise). Если вы никогда не работали с промисами в JavaScript, рекомендуем их изучить, например, по этой документации: https://learn.javascript.ru/promise.

Изменения значительные, поэтому мы рекомендуем вам разрабатывать и отлаживать новые проекты не только на стабильной версии Web3, но и на новых бета-версиях.

Текущую версию документации для фреймворка Web3 версии 1.0.x можно почитать здесь: https://web3js.readthedocs.io/en/1.0/web3-eth.html.

Чтобы установить самую новую версию фреймворка Web3, выполните следующие команды:

```
$ npm uninstall web3
$ npm install web3
```

Если нужно установить какую-либо конкретную версию, вы можете сделать это так:

```
$ npm install web3@1.0.0-beta.48
```

Здесь мы установили версию 1.0.0-beta.48, которую использовали для скриптов, использованных на этом уроке.

Получаем список аккаунтов

В листинге 6.4. мы привели исходный текст скрипта, получающего список аккаунтов локального узла и баланс каждого аккаунта в единицах wei и ether.

Листинг 6.4. Файл list_accounts_promise.js

```
var Web3 = require('web3')
var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:8545"));

web3.eth.getAccounts()
.then(accList => {
 return accList;
})
.then(function (accounts) {
 var balancePromeses = [];
 for(let i = 0; i < accounts.length; i++) {
   balancePromeses[i] = web3.eth.getBalance(accounts[i]);
 }

 Promise.all(balancePromeses).then(values => {
   for(let i = 0; i < values.length; i++) {
```

```
            console.log('Account: ', accounts[i], 'balance:
', values[i], 'wei, ', web3.utils.fromWei(values[i], 'ether'),
'ether');
       }
     });
    })
    .catch(function (error) {
     console.error(error);
    });
```

Список аккаунтов мы получаем с помощью функции web3.eth.getAccounts. Обработку аккаунтов мы выполняем в цикле:

```
    var balancePromeses = [];
    for(let i = 0; i < accounts.length; i++) {
     balancePromeses[i] = web3.eth.getBalance(accounts[i]);
    }
```

Обратите внимание, что в версии 1.0.x фреймворка Web3 та функция web3.eth.getBalance возвращает не значение баланса, а промис. Если у нас несколько аккаунтов, то мы получаем массив промисов.

Далее наш скрипт дожидается всех промисов с помощью функции Promise.all, получающей в качестве параметра массив промисов:

```
    Promise.all(balancePromeses).then(values => {
     for(let i = 0; i < values.length; i++) {
          console.log('Account: ', accounts[i], 'balance: ',
values[i], 'wei, ', web3.utils.fromWei(values[i], 'ether'),
'ether');
     }
```

Когда все промисы будут получены, скрипт выводит балансы аккаунтов на консоль в цикле:

```
    $ node list_accounts_promise.js
    Account:          0xf49C317B2C1B73675D141d870cA4e55ED92ABe86
balance:  100000000000000000000 wei,  100 ether
    Account:          0x0787ce52d2A75d816e3E55C7DFAcf438aA20Ac6f
balance:  100000000000000000000 wei,  100 ether
    Account:          0xD8e118A68F71bCCA3F9B2d2CB454647aa792a821
balance:  100000000000000000000 wei,  100 ether
    Account:          0xcA5819B17018B05f307E5Cc8E8215B739314356a
balance:  100000000000000000000 wei,  100 ether
    Account:          0x75aF7A607aae16a02d13c2a634e12F3213C60d5F
balance:  100000000000000000000 wei,  100 ether
    Account:          0x1b50941936a15dC63e0721C1Ba7ea7592f6c5727
balance:  100000000000000000000 wei,  100 ether
    Account:          0x5F6B4b75B5D09c3268CAF736566421F6eda891d7
balance:  100000000000000000000 wei,  100 ether
```

```
    Account:          0x52fda7De2105F38CA0811423E9738ec3Ea17c38d
balance:  100000000000000000000 wei,  100 ether
    Account:          0x16257fc1965105D1AE3B7B1d59BCadCa90745825
balance:  100000000000000000000 wei,  100 ether
    Account:          0x9bA3490448B15EaAC91FB7cB5c656aA83d95cBc4
balance:  100000000000000000000 wei,  100 ether
```

Публикация контракта

Для публикации контрактов с использованием фреймворка Web3 версии 1.0.x мы подготовили скрипт deploy_contract_promise.js (листинг 6.5.).

Листинг 6.5. Файл deploy_contract_promise.js

```
    // node deploy_contract_promise.js HelloSol *******

    var contract_to_deploy = process.argv[2];
    var unlock_password = process.argv[3];

    console.log('Contract Deploy script: ' + contract_to_deploy);

    var solc_cmd ='solc -bin -abi ' + contract_to_deploy + '.sol
-o build -overwrite';
    var abi_path = 'build/' + contract_to_deploy + '.abi';
    var bin_path = 'build/' + contract_to_deploy + '.bin';

    var abi;
    var bin;
    var contractObject;
    var myCoinbase;

    const { exec } = require('child_process');
    exec(solc_cmd, (err, stdout, stderr) => {
     if(err) {
       console.log(`exec error: ${err}`);
       return;
     }
    });

    var fs = require("fs");
    var Web3 = require('web3');
    var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:8545"));

    web3.eth.getCoinbase()
    .then(coinbase => {
     myCoinbase = coinbase;
     return coinbase;
    })
    .then(function (account) {
```

```
            return        web3.eth.personal.unlockAccount(account,
unlock_password, 600)
    })
    .then(function (unlocked) {
     console.log('Unlocked: ' + unlocked);

     abi = JSON.parse(fs.readFileSync(abi_path), 'utf8');
     bin = fs.readFileSync(bin_path);

     return web3.eth.estimateGas({ data: '0x' + bin })
    })
    .then(function (gas) {
     console.log('Gas estimation: ' + gas);
     contractObject = new web3.eth.Contract(abi, myCoinbase, {
       from: myCoinbase, gas: 4700000, data: '0x'+ bin
     });

     contractObject.deploy({
         data: '0x'+ bin
     })
     .send({
         from: myCoinbase,
         gas: 1500000,
         gasPrice: '30000000000000'
     }, (error, transactionHash) => { console.log('transactionHash:
' + transactionHash) })
     .on('error', (error) => { console.log('Error: ' + error) })
     .on('transactionHash', (transactionHash) => { console.log('on
transactionHash: ' + transactionHash) })
     .on('receipt', (receipt) => {
        console.log('contractAddress: ' + receipt.contractAddress)
     })
     .on('confirmation', (confirmationNumber, receipt) => {
       console.log('on confirmation: ' + confirmationNumber)
     })
     .then((newContractInstance) => {
       console.log('Contract Deployed');
       console.log('coinbase: ' + myCoinbase);
                  console.log('New    Contract    address:    '
+ newContractInstance.options.address);

             fs.writeFileSync(contract_to_deploy   +   ".address",
newContractInstance.options.address);
     });
    })
    .catch(function (error) {
     console.error(error);
    });
```

При запуске мы должны передать скрипту два параметра – имя смарт-контракта и пароль для разблокировки аккаунта.

Как и раньше, контракт компилируется утилитой solc.

Перед публикацией контракта мы разблокируем аккаунт. Теперь это нужно делать с использованием промисов:

```
    web3.eth.getCoinbase()
    .then(coinbase => {
     myCoinbase = coinbase;
     return coinbase;
    })
    .then(function (account) {
           return       web3.eth.personal.unlockAccount(account,
unlock_password, 600)
    })
```

Если аккаунт разблокирован успешно, получаем интерфейс ABI и бинарный код контракта, созданные на этапе компиляции, а также оцениваем количество газа, необходимого для публикации контракта:

```
    .then(function (unlocked) {
     console.log('Unlocked: ' + unlocked);

     abi = JSON.parse(fs.readFileSync(abi_path), 'utf8');
     bin = fs.readFileSync(bin_path);

     return web3.eth.estimateGas({ data: '0x' + bin })
    })
```

Теперь можно выполнять публикацию контракта с помощью функции deploy:

```
    .then(function (gas) {
     console.log('Gas estimation: ' + gas);
     contractObject = new web3.eth.Contract(abi, myCoinbase, {
       from: myCoinbase, gas: 4700000, data: '0x'+ bin
     });

     contractObject.deploy({
         data: '0x'+ bin
     })
     .send({
         from: myCoinbase,
         gas: 1500000,
         gasPrice: '30000000000000'
     }, (error, transactionHash) => { console.log('transactionHash:
' + transactionHash) })
     .on('error', (error) => { console.log('Error: ' + error) })
     .on('transactionHash', (transactionHash) => { console.log('on
transactionHash: ' + transactionHash) })
```

```
     .on('receipt', (receipt) => {
        console.log('contractAddress: ' + receipt.contractAddress)
     })
     .on('confirmation', (confirmationNumber, receipt) => {
       console.log('on confirmation: ' + confirmationNumber)
     })
     .then((newContractInstance) => {
       console.log('Contract Deployed');
       console.log('coinbase: ' + myCoinbase);
                  console.log('New    Contract    address:    '
+ newContractInstance.options.address);

            fs.writeFileSync(contract_to_deploy   +   ".address",
newContractInstance.options.address);
     });
    })
    .catch(function (error) {
     console.error(error);
    });
```

Здесь определены обработчики on, позволяющие получать сообщения об ошибках, хеш транзакции, квитанцию для выполненной транзакции, отслеживать подтверждения транзакции, а также получить адрес опубликованного контракта.

Обратите внимание, что после получения адреса контракта мы записываем его в файл с таким же именем, как имя контракта, и с расширением имени .address. Этот файл потребуется нам в следующих скриптах, которые будут выполнять обращение к контракту. Они будут считывать адрес контракта из файла, а не из командной строки, что упростит отладку скриптов и контракта.

Запуск публикации контракта выполняется следующим образом:

```
$ node deploy_contract_promise.js HelloSol ""
Contract Deploy script: HelloSol
Unlocked: true
Gas estimation: 344227
transactionHash:
0x43acb2a8a9a005bbc73ba9f045cf40c994cb4c941260b3c659fd810586849e3f
on                                              transactionHash:
0x43acb2a8a9a005bbc73ba9f045cf40c994cb4c941260b3c659fd810586849e3f
transactionHash: [object Object]
contractAddress: 0x2441501236a21d459c2F9677330738b55Bafb166
Contract Deployed
coinbase: 0xf49c317b2c1b73675d141d870ca4e55ed92abe86
New                      Contract                        address:
0x2441501236a21d459c2F9677330738b55Bafb166
```

Как видите, после публикации контракта мы получили его адрес.

Вызов функций контракта

Для того чтобы показать, как можно вызывать методы контракта с помощью фреймворка Web3 версии 1.0.x, мы подготовили два скрипта. Один из них вызывает функции контракта, устанавливающие числовое и строчное значение (листинг 6.6.), другой – получающие значения из контракта (листинг 6.7.).

Листинг 6.6. Файл call_contract_set_promise.js

```
// node call_contract_set_promise.js HelloSol *******

var contract_to_deploy = process.argv[2];
var unlock_password = process.argv[3];

console.log('Contract script: ' + contract_to_deploy);
var abi_path = 'build/' + contract_to_deploy + '.abi';
var bin_path = 'build/' + contract_to_deploy + '.bin';

var abi;
var bin;
var myCoinbase;

var fs = require("fs");
var Web3 = require('web3');
var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:8545"));

var version = web3.version;
console.log('Web3 version: ' + version);

var contract_address = fs.readFileSync(contract_to_deploy +
'.address').toString();
console.log('contract_address: ' + contract_address);

web3.eth.getCoinbase()
.then(coinbase => {
 myCoinbase = coinbase;
 console.log('coinbase: ' + coinbase);
 return coinbase;
})
.then(function (account) {
        return       web3.eth.personal.unlockAccount(account,
unlock_password, 600)
})
.then(function (unlocked) {
 console.log('Unlocked: ' + unlocked);
 abi = JSON.parse(fs.readFileSync(abi_path), 'utf8');
 bin = fs.readFileSync(bin_path);
```

```
 var myContract = new web3.eth.Contract(abi, contract_address);

myContract.methods.setValue(307309).send({from: myCoinbase})
.once('setValue transactionHash', (hash) => {
 console.log('hash: ' + hash);
})
.on('setValue confirmation', (confNumber) => {
 console.log('confNumber: ' + confNumber);
})
.on('receipt', (receipt) => {
 console.log(JSON.stringify(receipt, undefined, 2));
})
.then(function () {

myContract.methods.setString("Test                    string
307309").send({from: myCoinbase})
.once('transactionHash', (hash) => {
 console.log('setString hash: ' + hash);
})
.on('confirmation', (confNumber) => {
 console.log('setString confNumber: ' + confNumber);
})
.on('receipt', (receipt) => {
 console.log(JSON.stringify(receipt, undefined, 2));
})
})

})
.catch(function (error) {
 console.error(error);
});
```

Сразу после запуска скрипт call_contract_set_promise.js определяет и выводит на консоль версию фреймворка Web3:

```
var version = web3.version;
console.log('Web3 version: ' + version);
```

Далее скрипт читает адрес контракта из файла с расширением имени .address, куда он был записан при публикации контракта скриптом deploy_contract_promise.js (листинг 6.5.):

```
var contract_address = fs.readFileSync(contract_to_deploy +
'.address').toString();
console.log('contract_address: ' + contract_address);
```

Далее нам нужно разблокировать аккаунт, так как при вызове функции записи текстовой строки будет запущена транзакция:

```
web3.eth.getCoinbase()
```

```
    .then(coinbase => {
     myCoinbase = coinbase;
     console.log('coinbase: ' + coinbase);
     return coinbase;
    })
    .then(function (account) {
           return       web3.eth.personal.unlockAccount(account,
unlock_password, 600)
    })
```

После разблокировки скрипт создает объект контракта и вызывает функции setString и setValue:

```
    .then(function (unlocked) {
     console.log('Unlocked: ' + unlocked);
     abi = JSON.parse(fs.readFileSync(abi_path), 'utf8');
     bin = fs.readFileSync(bin_path);

     var myContract = new web3.eth.Contract(abi, contract_address);

    myContract.methods.setValue(307309).send({from: myCoinbase})
    .once('setValue transactionHash', (hash) => {
     console.log('hash: ' + hash);
    })
    .on('setValue confirmation', (confNumber) => {
     console.log('confNumber: ' + confNumber);
    })
    .on('receipt', (receipt) => {
     console.log(JSON.stringify(receipt, undefined, 2));
    })
    .then(function () {

    myContract.methods.setString("Test                    string
307309").send({from: myCoinbase})
    .once('transactionHash', (hash) => {
     console.log('setString hash: ' + hash);
    })
    .on('confirmation', (confNumber) => {
     console.log('setString confNumber: ' + confNumber);
    })
    .on('receipt', (receipt) => {
     console.log(JSON.stringify(receipt, undefined, 2));
    })
    })
    })
    .catch(function (error) {
     console.error(error);
    });
```

В этом фрагменте кода мы использовали промисы. Вы также можете вызывать эти методы с помощью функций обратного вызова:

```
    myContract.methods.setValue(307309).estimateGas({gas: 50000},
function(error, gasAmount){
     console.log('gasAmount setValue: ' + gasAmount);
     if(gasAmount == 50000)
         console.log('Method setValue ran out of gas');
    });

    myContract.methods.setString("Test                     string
307309").estimateGas({gas: 50000}, function(error, gasAmount){
     console.log('gasAmount setString: ' + gasAmount);
     if(gasAmount == 50000)
         console.log('Method setString ran out of gas');
    });
```

При запуске скрипта call_contract_set_promise.js нужно указать имя контракта и пароль аккаунта (для Ganache CLI он должен быть пустой), после чего можно будет наблюдать результат:

```
    $ node call_contract_set_promise.js HelloSol ""
    Contract script: HelloSol
    Web3 version: 1.0.0-beta.48
    contract_address: 0x2441501236a21d459c2F9677330738b55Bafb166
    coinbase: 0xf49c317b2c1b73675d141d870ca4e55ed92abe86
    Unlocked: true
    {
                                                  "transactionHash":
"0xcaffbd40a7e1bfd6aef58e08970d360163c41641ccdcbbea2ad320db4c38a16c",
     "transactionIndex": 0,
                                                        "blockHash":
"0xb6e1744ee73a1129a98f11c6f03be55c8a8d851a98976db27520b7659b494e0f",
     "blockNumber": 3,
     "from": "0xf49c317b2c1b73675d141d870ca4e55ed92abe86",
     "to": "0x2441501236a21d459c2f9677330738b55bafb166",
     "gasUsed": 41834,
     "cumulativeGasUsed": 41834,
     "contractAddress": null,
     "status": true,
                                                        "logsBloom":
"0x0000000000000000000000000000000000000000000000000000000000000000000
     "v": "0x1b",
                                                                "r":
"0x7c68fbbc9fca19ad637324e5282fe26e7e36d24c59790f314c16fe6af143f212",
                                                                "s":
"0x245e7b0c98cd6d9a186d16b040e87c6a6f91f8df32d8970276a5d89afe4aa01b",
     "events": {}
    }
```

```
    setString                                                    hash:
0x6c1f806e0a0821a54fed6bc87b9a52ff04ad1d487001d7bbfcf6227a8526be6b
    {
                                                    "transactionHash":
"0x6c1f806e0a0821a54fed6bc87b9a52ff04ad1d487001d7bbfcf6227a8526be6b",
     "transactionIndex": 0,
                                                      "blockHash":
"0xd7bfebc658b877d5417dbce946f45df82b462d94b7ff406aac1d528845a264da",
     "blockNumber": 4,
     "from": "0xf49c317b2c1b73675d141d870ca4e55ed92abe86",
     "to": "0x2441501236a21d459c2f9677330738b55bafb166",
     "gasUsed": 44086,
     "cumulativeGasUsed": 44086,
     "contractAddress": null,
     "status": true,
                                                      "logsBloom":
"0x00000000000000000000000000000000000000000000000000000000000000000
     "v": "0x1c",
                                                           "r":
"0x84119d92db4d76ba8e54318c4fc9431dabdbe79d9eef7720da8f2c65540deba8",
                                                           "s":
"0x2cb43decc69585460209272192 1a1a22e14e456a163eaceeb4d4f81849ce0f77",
     "events": {}
    }
```

Скрипт, вызывающий функции getValue и getString приведен в листинге 6.7.

Листинг 6.7. Файл call_contract_get_promise.js

```
    // node call_contract_get_promise.js HelloSol

    var contract_to_deploy = process.argv[2];

    console.log('Contract script: ' + contract_to_deploy);
    var abi_path = 'build/' + contract_to_deploy + '.abi';
    var bin_path = 'build/' + contract_to_deploy + '.bin';

    var abi;
    var bin;
    var myCoinbase;

    var fs = require("fs");
    var Web3 = require('web3');
    var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:8545"));

    var version = web3.version;
    console.log('Web3 version: ' + version);
```

```
    var contract_address = fs.readFileSync(contract_to_deploy +
'.address').toString();
    console.log('contract_address: ' + contract_address);

    web3.eth.getCoinbase()
    .then(coinbase => {
     myCoinbase = coinbase;
     console.log('coinbase: ' + coinbase);
     return coinbase;
    })
    .then(function () {
     abi = JSON.parse(fs.readFileSync(abi_path), 'utf8');
     bin = fs.readFileSync(bin_path);

     var myContract = new web3.eth.Contract(abi, contract_address);

    //  console.log(JSON.stringify(myContract, undefined, 2));

     myContract.methods.getValue().call({from: contract_address},
(error, result) =>
     {
        if(!error){
          console.log('getValue result: ' + result);
          console.log(JSON.stringify(result, undefined, 2));
        } else{
          console.log(error);
        }
     });

                         myContract.methods.getString().call({from:
contract_address}, (error, result) =>
     {
        if(!error){
          console.log('getString result: ' + result);
    //       console.log(JSON.stringify(result, undefined, 2));
        } else{
          console.log(error);
        }
     });
    })
    .catch(function (error) {
     console.error(error);
    });
```

Здесь не нужно разблокировать аккаунт, т.к. транзакция не требуется.
При запуске скрипта укажите имя контракта:

```
$ node call_contract_get_promise.js HelloSol
Contract script: HelloSol
```

```
Web3 version: 1.0.0-beta.48
contract_address: 0x2441501236a21d459c2F9677330738b55Bafb166
coinbase: 0xf49c317b2c1b73675d141d870ca4e55ed92abe86
getValue result: 307309
"307309"
getString result: Test string 307309
```

Перевод средств с одного аккаунта на другой

Для того чтобы продемонстрировать перевод средств с одного аккаунта на другой, мы подготовили скрипт send_ether_promise.js (листинг 6.8.).

Листинг 6.8. Файл send_ether_promise.js

```
// node send_ether_promise.js HelloSol
0x6cbbA0d2bC126aEE51Acc4614E530b12B7abb4a1
0xDfaEFc30fB1Ac864Be7c5F44dffBb90C9ED2bA8d *******

var from_address = process.argv[2];
var to_address = process.argv[3];
var unlock_password = process.argv[4];

var myCoinbase;

var fs = require("fs");
var Web3 = require('web3');
var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:8545"));

var version = web3.version;
console.log('Web3 version: ' + version);

web3.eth.getCoinbase()
.then(coinbase => {
 myCoinbase = coinbase;
 console.log('coinbase: ' + coinbase);
 return coinbase;
})
.then(function (account) {
       return web3.eth.personal.unlockAccount(account,
unlock_password, 600)
})
.then(function (unlocked) {
 console.log('Unlocked: ' + unlocked);
 console.log('from_address: ' + from_address);
 console.log('to_address: ' + to_address);

web3.eth.sendTransaction({
   from: from_address,
   to: to_address,
```

```
       value: '1000000000000000'
    })
    .on('transactionHash', function(hash){
     console.log(hash);
    })
    .on('receipt', function(receipt){
     console.log(JSON.stringify(receipt, undefined, 2));
    })
    .on('confirmation',   function(confirmationNumber,   receipt)
{         console.log('sendTransaction   confirmationNumber:   '
+ confirmationNumber); })
    .on('error', console.error);
    })
    .then(function (hash) {
     console.log(hash);
    })
    .catch(function (error) {
     console.error(error);
    });
```

В качестве параметров при запуске скрипту нужно передать адрес аккаунта, с которого передаются средства, адрес аккаунта, получающего средства, а также пароль для разблокировки первого из этих аккаунтов:

```
    var from_address = process.argv[2];
    var to_address = process.argv[3];
    var unlock_password = process.argv[4];
```

После того как исходный аккаунт будет разблокирован, скрипт вызывает функцию web3.eth.sendTransaction:

```
    web3.eth.sendTransaction({
       from: from_address,
       to: to_address,
       value: '1000000000000000'
    })
    .on('transactionHash', function(hash){
     console.log(hash);
    })
    .on('receipt', function(receipt){
     console.log(JSON.stringify(receipt, undefined, 2));
    })
    .on('confirmation',   function(confirmationNumber,   receipt)
{         console.log('sendTransaction   confirmationNumber:   '
+ confirmationNumber); })
    .on('error', console.error);
    })
    .then(function (hash) {
     console.log(hash);
```

```
})
```

Этой функции мы передаем адрес источника и адрес приемника, а также передаваемую сумму в единицах wei.

Перед запуском скрипта убедитесь, что на вашем узле имеются по крайней мере два аккаунта. Это можно сделать, например, запустив скрипт list_accounts_promise.js:

```
$ node list_accounts_promise.js
Account:          0xf49C317B2C1B73675D141d870cA4e55ED92ABe86
balance:  79344661600000000000 wei,  79.3446616 ether

Account:          0x0787ce52d2A75d816e3E55C7DFAcf438aA20Ac6f
balance:  100000000000000000000 wei,  100 ether

Account:          0xD8e118A68F71bCCA3F9B2d2CB454647aa792a821
balance:  100000000000000000000 wei,  100 ether
Account:          0xcA5819B17018B05f307E5Cc8E8215B739314356a
balance:  100000000000000000000 wei,  100 ether
Account:          0x75aF7A607aae16a02d13c2a634e12F3213C60d5F
balance:  100000000000000000000 wei,  100 ether
Account:          0x1b50941936a15dC63e0721C1Ba7ea7592f6c5727
balance:  100000000000000000000 wei,  100 ether
Account:          0x5F6B4b75B5D09c3268CAF736566421F6eda891d7
balance:  100000000000000000000 wei,  100 ether
Account:          0x52fda7De2105F38CA0811423E9738ec3Ea17c38d
balance:  100000000000000000000 wei,  100 ether
Account:          0x16257fc1965105D1AE3B7B1d59BCadCa90745825
balance:  100000000000000000000 wei,  100 ether
Account:          0x9bA3490448B15EaAC91FB7cB5c656aA83d95cBc4
balance:  100000000000000000000 wei,  100 ether
```

Здесь на балансе всех аккаунтов, кроме первого, уже есть 100 ether. На первом аккаунте средств меньше.

Теперь запустите скрипт send_ether_promise.js:

```
$                 node                 send_ether_promise.js
0xf49C317B2C1B73675D141d870cA4e55ED92ABe86
0x0787ce52d2A75d816e3E55C7DFAcf438aA20Ac6f
Web3 version: 1.0.0-beta.48
coinbase: 0xf49c317b2c1b73675d141d870ca4e55ed92abe86
Unlocked: true
from_address: 0xf49C317B2C1B73675D141d870cA4e55ED92ABe86
to_address: 0x0787ce52d2A75d816e3E55C7DFAcf438aA20Ac6f
undefined
0x4f8af8d856bc29c708a1112cc4676afc7b827c55b1c45bd25b2b55371e0c3d9f
{
                                          "transactionHash":
"0x4f8af8d856bc29c708a1112cc4676afc7b827c55b1c45bd25b2b55371e0c3d9f",
 "transactionIndex": 0,
```

```
                                                                "blockHash":
"0x3d5b182f8c53e19149d481deaf8a1869bc8f058edfd684c78a45eac92eb4cf4c",
     "blockNumber": 5,
     "from": "0xf49c317b2c1b73675d141d870ca4e55ed92abe86",
     "to": "0x0787ce52d2a75d816e3e55c7dfacf438aa20ac6f",
     "gasUsed": 21000,
     "cumulativeGasUsed": 21000,
     "contractAddress": null,
     "logs": [],
     "status": true,
                                                                "logsBloom":
"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000
     "v": "0x1b",
                                                                    "r":
"0xc9e7bfae24ecb079b917ad18b7c38581dc438c9d98c5cf8d05d8378a2ac6df8e",
                                                                    "s":
"0x5da1dfbbb7fce02c7275eb589f93a6a4dddbe9862edec94642d803749097c188"
    }
```

После того как он отработает, проверьте балансы аккаунтов, они должны измениться:

```
    $ node list_accounts_promise.js
    Account:           0xf49C317B2C1B73675D141d870cA4e55ED92ABe86
balance:  79343241600000000000 wei,  79.3432416 ether

    Account:           0x0787ce52d2A75d816e3E55C7DFAcf438aA20Ac6f
balance:  100001000000000000000 wei,  100.001 ether

    Account:           0xD8e118A68F71bCCA3F9B2d2CB454647aa792a821
balance:  100000000000000000000 wei,  100 ether
    Account:           0xcA5819B17018B05f307E5Cc8E8215B739314356a
balance:  100000000000000000000 wei,  100 ether
    Account:           0x75aF7A607aae16a02d13c2a634e12F3213C60d5F
balance:  100000000000000000000 wei,  100 ether
    Account:           0x1b50941936a15dC63e0721C1Ba7ea7592f6c5727
balance:  100000000000000000000 wei,  100 ether
    Account:           0x5F6B4b75B5D09c3268CAF736566421F6eda891d7
balance:  100000000000000000000 wei,  100 ether
    Account:           0x52fda7De2105F38CA0811423E9738ec3Ea17c38d
balance:  100000000000000000000 wei,  100 ether
    Account:           0x16257fc1965105D1AE3B7B1d59BCadCa90745825
balance:  100000000000000000000 wei,  100 ether
    Account:           0x9bA3490448B15EaAC91FB7cB5c656aA83d95cBc4
balance:  100000000000000000000 wei,  100 ether
```

Подробное описание функции web3.eth.sendTransaction вы найдете здесь: https://web3js.readthedocs.io/en/1.0/web3-eth.html#eth-sendtransaction.

Перевод средств на аккаунт контракта

В предыдущем разделе мы написали скрипт send_ether_promise.js (листинг 6.8.), с помощью которого можно переводить средства с одного аккаунта на другой. Для теста мы переводили средства между аккаунтами, созданными Ganache CLI.

Однако мы говорили ранее, что для опубликованных смарт-контрактов тоже создается специальный аккаунт и на него тоже можно переводить средства нашим скриптом.

Обновляем смарт-контракт HelloSol

Для того чтобы смарт-контракт Solidity мог получать средства на свой аккаунт, в нем необходимо определить функцию с ключевым словом payable.

Добавьте в наш смарт-контракт следующие две функции:

```
function() external payable { }

function getBalance() public view returns( uint256 ) {
 return address(this).balance;
}
```

Первая из этих функций – так называемая резервная (fallback) функция. У нее нет имени, и вы можете создать только одну такую функцию в смарт-контракте. Для такой функции вы также должны определить параметр видимости external. Это означает, что она может быть вызвана из других смарт-контрактов или в результате выполнения транзакции.

Когда средства переводятся на аккаунт смарт-контракта, наша fallback-функция будет вызвана. В нашем примере эта функция ничего не делает, но ее наличие необходимо для того, чтобы была возможность пополнить аккаунт смарт-контракта средствами через отправку транзакции.

Что же касается функции getBalance, то она возвращает текущий баланс аккаунта смарт-контракта address(this).balance. Здесь balance – это функция, которая возвращает баланс аккаунта по его адресу. В данном случае мы используем в качестве адреса this, который представляет собой адрес смарт-контракта.

Начиная с версии 5.0 Solidity необходимо использовать явное преобразование этого адреса к типу address.

Обновленный смарт-контракт HelloSol.sol представлен в листинге 6.9.

Листинг 6.9. Файл HelloSol.sol

```
pragma solidity ^0.5.0;

contract HelloSol {
   string savedString;
   uint savedValue;

   function() external payable { }
   function getBalance() public view returns( uint256 ) {
     return address(this).balance;
   }
   function setString( string memory newString ) public {
```

```
            savedString = newString;
        }
        function getString() public view returns( string memory ) {
            return savedString;
        }
        function setValue( uint newValue ) public {
            savedValue = newValue;
        }
        function getValue() public view returns( uint ) {
            return savedValue;
        }
    }
```

Создаем скрипт для просмотра баланса аккаунта

Для удобного просмотра баланса аккаунта (обычного или смарт-контракта) вы можете использовать скрипт get_account_balance_promise.js (листинг 6.10.).

Листинг 6.10. Файл get_account_balance_promise.js

```
    var account = process.argv[2];
    var Web3 = require('web3')
    var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:8545"));
    web3.eth.getBalance(account)
    .then(function (balance) {
     console.log('Account: ', account, 'balance: ', balance, 'wei,
', web3.utils.fromWei(balance, 'ether'), 'ether');
    })
```

При запуске в качестве параметра этому скрипту нужно передать адрес аккаунта, баланс которого необходимо определить.

Добавляем вызов функции getBalance в скрипт call_contract_get_promise.js

Также мы добавим вызов функции getBalance нашего обновленного смарт-контракта HelloSol. Вставьте этот код в скрипт call_contract_get_promise.js (листинг 6.7.):

```
    myContract.methods.getBalance().call({from:
contract_address}, (error, result) =>
    {
      if(!error){
           console.log('Contract address: ', contract_address,
'balance: ', result, 'wei, ', web3.utils.fromWei(result, 'ether'),
'ether');
      } else{
        console.log(error);
      }
    });
```

Значение, полученное от функции getBalance, выводится на консоль как в исходном виде (баланс в единицах Wei), так и в преобразованном (в единицах Ether).

Пополняем счет смарт-контракта

Теперь, когда все готово, займемся переводом средств.

Прежде всего запустим в консоли узел Ganache CLI. Вы увидите список из 10 аккаунтов, на каждом из которых имеется по 100 Ether.

Чтобы увидеть список аккаунтов и их баланс, запустите скрипт list_accounts_promise.js (листинг 6.4.):

```
    $ node list_accounts_promise.js
    Account:           0xC48289db6c43eFf603D2284e6089E8Ece416BcF3
balance:  100000000000000000000 wei,  100 ether
    Account:           0x952e689B1308bF4d9fBE721fd83e0053D67F2965
balance:  100000000000000000000 wei,  100 ether
    Account:           0xA4801E83d92C3dA7b978BeBD07e03390fB6FA315
balance:  100000000000000000000 wei,  100 ether
    Account:           0x58F7e2d8b60086f8c424c5D2bE6BA1621DB3C424
balance:  100000000000000000000 wei,  100 ether
    Account:           0x481ac41A44b8B2eE1F5219E5c106286835a69139
balance:  100000000000000000000 wei,  100 ether
    Account:           0x2839221d12159Bbd8Cc9bB23EF1dc6B5400EbbbB
balance:  100000000000000000000 wei,  100 ether
    Account:           0x4465672F9428e9b4892BF2c711Cf8Af0CF074CA9
balance:  100000000000000000000 wei,  100 ether
    Account:           0xeCdF340e3e0d7ce8Da19bBcB513c8b0480bc12a3
balance:  100000000000000000000 wei,  100 ether
    Account:           0x5cFf1b92CDdFdbA8827B7504Bd1a7205777f5fc4
balance:  100000000000000000000 wei,  100 ether
    Account:           0x51EEc443F4e62846A130643175947b465f33e834
balance:  100000000000000000000 wei,  100 ether
```

Теперь опубликуем наш смарт-контракт с помощью скрипта deploy_contract_promise.js (листинг 6.5.):

```
    ~$ node deploy_contract_promise.js HelloSol ""
    Contract Deploy script: HelloSol
    Unlocked: true
    Gas estimation: 323129
    transactionHash:
0x8aed5fbdb60d738ba17b3ecca41445a77b92a23119cc7ecb5a63c2213f0f4c5b
    on                                                transactionHash:
0x8aed5fbdb60d738ba17b3ecca41445a77b92a23119cc7ecb5a63c2213f0f4c5b
    transactionHash: [object Object]
    contractAddress: 0xeD8d81991cfbB4f094b45C43C5E3A8780C134792
    Contract Deployed
    coinbase: 0xc48289db6c43eff603d2284e6089e8ece416bcf3
```

```
    New                    Contract                    address:
0xeD8d81991cfbB4f094b45C43C5E3A8780C134792
```

В результате публикации мы получили адрес нашего смарт-контракта, на который будем переводить средства: 0xeD8d81991cfbB4f094b45C43C5E3A8780C134792. При этом баланс основного аккаунта уменьшился, что можно увидеть с помощью скрипта get_account_balance_promise.js:

```
    $          node          get_account_balance_promise.js
0xC48289db6c43eFf603D2284e6089E8Ece416BcF3
    Account:          0xC48289db6c43eFf603D2284e6089E8Ece416BcF3
balance:  90306130000000000000 wei,  90.30613 ether
```

Баланс аккаунта смарт-контракта на данный момент равен нулю:

```
    $          node          get_account_balance_promise.js
0xeD8d81991cfbB4f094b45C43C5E3A8780C134792
    Account:          0xeD8d81991cfbB4f094b45C43C5E3A8780C134792
balance:  0 wei,  0 ether
```

Давайте запустим скрипт send_ether_promise.js (листинг 6.8.), описанный в предыдущем разделе. В качестве первого параметра передадим скрипту адрес нашего основного аккаунта, а в качестве второго – адрес смарт-контракта:

```
    $              node              send_ether_promise.js
0xC48289db6c43eFf603D2284e6089E8Ece416BcF3
0xeD8d81991cfbB4f094b45C43C5E3A8780C134792
```

Теперь мы видим, что на балансе смарт-контракта есть средства в количестве 10 Ether, как это и должно быть:

```
    $          node          get_account_balance_promise.js
0xeD8d81991cfbB4f094b45C43C5E3A8780C134792
    Account:          0xeD8d81991cfbB4f094b45C43C5E3A8780C134792
balance:  10000000000000000000 wei,  10 ether
```

Аналогичный результат мы получим и при запуске скрипта call_contract_get_promise.js

```
    $ node call_contract_get_promise.js HelloSol
    Contract script: HelloSol
    Web3 version: 1.0.0-beta.48
    contract_address: 0xeD8d81991cfbB4f094b45C43C5E3A8780C134792
    coinbase: 0xc48289db6c43eff603d2284e6089e8ece416bcf3
    getValue result: 0
    "0"
    Contract address:  0xeD8d81991cfbB4f094b45C43C5E3A8780C134792
balance:  9865560500000000000 wei,  9.8655605 ether
    getString result:
```

Таким образом вы можете перечислять средства на аккаунт смарт-контракта, после чего смарт-контракт может тратить их на выполнение транзакций или перечислять на другие аккаунты.

Итоги урока

На шестом уроке вы сделали первые шаги к практическому использованию смарт-контрактов, вызывая их из Node.js. Вы установили Node.js в ОС Ubuntu, Debian и Rasberian, написали скрипт JavaScript для публикации смарт-контракта и вызова его функций.

В работе вы использовали Web3 стабильной и новой, пока еще нестабильной, версий, составляя программы с применением промисов JavaScript.

Вы также научились создавать скрипты, способные переводить средства между обычными аккаунтами и зачислять их на аккаунты смарт-контрактов.

Урок 7. Введение в Truffle

Цель урока: научиться устанавливать интегрированную среду разработки смарт-контрактов Truffle, научиться компилировать, публиковать и тестировать смарт-контракты в Truffle.

Практические задания: на этом уроке вы должны будете установить IDE Truffle версий 4.x и 5.x, создать смарт-контракт, запустить его на компиляцию и опубликовать в сети разработки с помощью Truffle. Далее вы научитесь писать скрипты JavaScript, вызывающие функции контрактов с помощью модуля truffle-contract, а также выполните тестирование своего смарт-контракта. Таким образом вы сделаете первые шаги на пути к использованию Truffle для разработки смарт-контрактов Solidity.

Из предыдущих уроков вы получили некоторое представление о том, как можно публиковать и вызывать контракты Solidity. Однако на практике намного удобнее пользоваться интегрированными средами разработки. На этом уроке мы изучим IDE Truffle, очень популярную среди программистов Solidity. Она проста в установке и удобна на практике.

Вы можете найти подробную информацию об установке и использовании Truffle на сайте http://truffleframework.com/.

Установка Truffle

Перед тем как устанавливать Truffle, убедитесь, что вы уже установили Node.js и менеджер пакетов npm. Процедура установки была описана на предыдущем уроке.

Проверить версии Node.js и npm можно командами:

```
$ node -v
v10.15.3

$ npm -v
6.4.1
```

Если с установкой Node.js все в порядке, запустите установку Truffle:

```
$ npm install -g truffle
```

И это все! Можно переходить к созданию нашего первого проекта.

Определить текущую установленную версию Truffle можно с помощью следующей команды:

```
$ truffle version
Truffle v5.0.12 (core: 5.0.12)
Solidity v0.5.0 (solc-js)
Node v10.15.3
Web3.js v1.0.0-beta.37
```

Если вам нужно установить предыдущую или какую-либо определенную версию Truffel, это можно сделать так:

```
$ npm install -g truffle@4.1.15
```

Здесь мы устанавливаем версию 4.1.15. Обновить версию до 4.1.15 можно следующей командой:

`$ npm i -g truffle@4.1.15`

И наконец, чтобы удалить старую и поставить самую новую версию Truffel, воспользуйтесь такими командами:

```
$ npm uninstall -g truffle
$ npm install -g truffle
```

Создаем проект HelloSol

Мы создадим проект HelloSol для публикации и отладки смарт-контракта HelloSol.sol, с которым мы уже работали на предыдущих уроках (листинг 7.1.).

Листинг 7.1. Контракт HelloSol.sol

```
pragma solidity ^0.4.10;

contract HelloSol {
   string savedString;
   uint savedValue;

   function setString( string newString ) public {
       savedString = newString;
   }
   function getString() public constant returns( string ) {
       return savedString;
   }
   function setValue( uint newValue ) public {
       savedValue = newValue;
   }
   function getValue() public constant returns( uint ) {
       return savedValue;
   }
}
```

Обратите внимание, что в первой строке контракта указана версия компилятора 0.4.10. Это необходимо для использования Truffle версии 4.x, о чем мы расскажем подробнее позже на этом уроке.

Создание каталога и файлов проекта

Прежде всего вам нужно создать в домашнем каталоге рабочий каталог проекта HelloSol и сделать его текущим:

```
$ mkdir HelloSol
$ cd HelloSol
```

Далее запустите инициализацию проекта:

```
$ truffle init
```

Вы увидите сообщения об инициализации, а также краткую подсказку по дальнейшим действиям:

```
√ Preparing  to download
√ Downloading
√ Cleaning up temporary files
√ Setting up box
```

```
Unbox successful. Sweet!

Commands:

 Compile:        truffle compile
 Migrate:        truffle migrate
 Test contracts: truffle test
```

После выполнения инициализации будут созданы три каталога и два файла с расширением имени js:

- contracts
- migrations
- test
- truffle-config.js

Расскажем об этих каталогах и файлах подробнее.

Каталог contracts

Каталог contracts предназначен для хранения контрактов. Сразу после инициализации нового проекта в нем находится один служебный контракт Migrations.sol. Он нужен для управления и обновления статуса публикации вашего смарт-контракта, и на данном этапе мы не будем его редактировать.

Создайте с помощью редактора nano или vim в этом каталоге файл HelloSol.sol и запишите в него текст нашего смарт-контракта из листинга 7.1.

Каталог migrations

Этот каталог содержит скрипты, предназначенные для публикации контрактов в сети.

Сразу после создания нового проекта в каталоге migrations находится файл 1_initial_migration.js, предназначенный для публикации служебного контракта Migrations.sol.

Вам нужно будет добавить в этот каталог файл сценария публикации нашего контракта HelloSol.sol с именем 2_hellosol_migrations.js (листинг 7.2.).

Листинг 7.2. Сценарий миграции 2_hellosol_migrations.js

```
var MyContract = artifacts.require("HelloSol");

module.exports = function(deployer) {
 // deployment steps
 deployer.deploy(MyContract);
};
```

Имя файла сценария публикации должно начинаться с номера. Этот номер используется для сохранения записи о том, что публикация была выполнена успешно.

Каталог test

В каталоге test должны находиться скрипты тестирования, составленные на JavaScript или Solidity. Пока не добавляйте в каталог test никакие файлы.

Файл truffle-config.js

Файл truffle-config.js также создаются автоматически при инициализации нового проекта. В нем находятся параметры конфигурации Truffle (листинг 7.3.).

Листинг 7.3. Файл truffle-config.js

```
module.exports = {
    //    See    <http://truffleframework.com/docs/advanced/
configuration>
 // to customize your Truffle configuration!
   networks: {
   development: {
     host: "127.0.0.1",
     port: 8545,
     network_id: "*" // Match any network id
   }
 }
};
```

Компиляция контракта HelloSol

Для компиляции контракта введите следующую команду, находясь в рабочем каталоге проекта:

```
$ truffle compile
```

Команда выполнит компиляцию контрактов, расположенных в папке contracts, и запишет результаты в свой внутренний рабочий каталог ./build/contracts:

```
Compiling ./contracts/HelloSol.sol…
Compiling ./contracts/Migrations.sol…
Writing artifacts to ./build/contracts
```

Заметим, что компилироваться будут только те контракты, в которые мы внесли изменения.

Результаты компиляции будут записаны в файлы с расширением имени json:

```
$ ls build/contracts/
HelloSol.json  Migrations.json
```

Эти файлы будут содержать, кроме всего прочего, ABI и бинарный код контрактов.

В процессе своей работы Truffle создает файлы, которые называются артефактами, вроде только что упомянутых файлов *.json. Эти файлы, конечно, можно просматривать, но не нужно редактировать. При выполнении операций Truffle перезапишет эти файлы и затрет внесенные вами изменения.

Запуск публикации контракта

Прежде чем запускать публикацию контракта, давайте запустим отладочный приватный блокчейн, встроенный в Truffle. Он называется Truffle Develop и запускается очень просто:

```
$ truffle develop
```

После запуска на консоли появится сообщение, в котором будет указан сетевой интерфейс, предварительно созданные 10 аккаунтов и их приватные ключи, а также мнемонические обозначения аккаунтов:

```
Truffle Develop started at http://127.0.0.1:9545/

Accounts:
(0) 0x532a094f3c4d2547037275a0d94041f471e756b6
(1) 0x938b0fc1c7ff674fbb4b343ca9cdef14a595a395
(2) 0x3d829baeffd13c9fd919123a903cd8d9a2926699
(3) 0x520e62abaf3a9a85a45efbb18ffbe5da0d10d5c3
(4) 0xe79f908588396b6eccb9749794198cf915a07d87
(5) 0xe4b93db98d0833e67e538501a31a925e88f68c72
(6) 0xb8e131e804aec737caf19b6f0d2999dde8e3a7be
(7) 0x4c4390f2c09ab798c313af252562dd7a26e6705e
(8) 0x6fab43b6a6eeb7ec67d9d25deba0dc5ef9a3c7b4
(9) 0x51c1e17be601f118f23064e501098fe2107d64ab

Private Keys:
(0)
845f31e64a2604f35b235fbc039c049a540f6e60f56639baadebaf86508a8d28
(1)
a3980fa82a05f4f39362782e4ec1eba77cd1bfd3411757738c964e9c10d630da
(2)
3d895cf36f0a46ccf82aa7f4ecc59f7c86f8b011a5b4a4d7491ee90f8bd731f8
(3)
bde4532f31b638aaf6dd663fd4e020cbd1440adc88d1c9a41bdda06c58304cfc
(4)
d363b7e19ae6289d1c76d55b6373f364769ec96e625f8ffd88e926484dc71e70
(5)
96c0081ca7d4546d5dc51a925148c30c58bb99cf79eb00fdceca6c731f5d157d
(6)
50d7930046db3e69e319d61e173f5078041753e3988c9139864a81f417fafd9c
(7)
7cf88f4bf05ef2f7ba242ae968d2f398d12d5bd9809793133a573dc786d16b6a
(8)
1129f0e0ae1c50f832087608c7af5c8e73fa3522405047633fb0d24b8cf07ec1
(9)
706c4ee5187e913dc0ce834234b64fbca7dd3e8fa7d5a3c568f73c79a095dd67
```

```
    Mnemonic: candy maple cake sugar pudding cream honey rich smooth
crumble sweet treat
```

⚠ Important ⚠ : This mnemonic was created for you by Truffle. It is not secure.
Ensure you do not use it on production blockchains, or else you risk losing funds.

После вы увидите командное приглашение truffle (develop):

```
    truffle(develop)>
```

Здесь можно вводить все те же команды truffle, что мы уже использовали на этом уроке.
Для публикации наших контрактов введите команду migrate:

```
    truffle(develop)> migrate
```

По умолчанию будет выполнена публикация в приватную сеть (блокчейн) Truffle с названием develop:

```
    > migrate
    ⚠  Important ⚠

    If you're using an HDWalletProvider, it must be Web3 1.0 enabled
or your migration will hang.

    Starting migrations…
    =
    > Network name:    'develop'
    > Network id:      4447
    > Block gas limit: 6721975

    1_initial_migration.js
    =

      Deploying 'Migrations'
      -
                                       >                transaction
hash:    0xbcd3c2def96ac78785f8eaa3a37f9d58fb57f6d0c5b687994f9a75fe5b3a
      > Blocks: 0            Seconds: 0
                                        >                contract
address:    0x4b27b93eF2F4496e67504d2d057a1b7fB6589481
                                                                    >
account:             0x532A094f3C4D2547037275a0d94041f471E756B6
      > balance:             99.99430184
      > gas used:            284908
      > gas price:           20 gwei
      > value sent:          0 ETH
      > total cost:          0.00569816 ETH

      > Saving migration to chain.
```

```
      > Saving artifacts
      -
      > Total cost:          0.00569816 ETH

    2_hellosol_migrations.js

      Deploying 'HelloSol'
      -
                                       >                transaction
hash:    0xfbb981bc2bc66d563a623104b805ae493d333428eac0e91e70988ef3f8e3
      > Blocks: 0            Seconds: 0
                                        >                  contract
address:    0xb5C65092baeede3b7b83938fCc36636B4944993e
                                                                    >
account:             0x532A094f3C4D2547037275a0d94041f471E756B6
      > balance:             99.98728008
      > gas used:            309054
      > gas price:           20 gwei
      > value sent:          0 ETH
      > total cost:          0.00618108 ETH

      > Saving migration to chain.
      > Saving artifacts
      -
      > Total cost:          0.00618108 ETH
    …
```

Вы увидите идентификаторы транзакций и адреса опубликованных контрактов.
Повторную публикацию контракта выполняйте с параметром –reset:

```
migrate –reset
```

Заметим, что Truffle позволяет легко публиковать контракты в разные сети, например, в тестовую или в основную сеть Ethereum. Для этого нужно настроить конфигурацию Truffle.

Вызов функций контракта HelloSol в приглашении Truffle

Теперь давайте вызовем функции контракта в приглашении Truffle. Начнем с функции getValue:

```
    truffle(develop)> HelloSol.deployed().then(function(instance)
{return instance.getValue()});
    <BN: 0>
```

Здесь полученное числовое значение выводится во внутреннем формате BigNumber, позволяющем хранить очень большие числа. Конвертируем его в обычное число с помощью функции toNumber:

```
    truffle(develop)> HelloSol.deployed().then(function(instance)
{return         instance.getValue()}).then(function(value){return
value.toNumber()});
    0
```

Учтите, что такое преобразование не будет работать, если число действительно очень большое. Надежнее конвертировать очень большие числа в строки с помощью функции toString:

```
    truffle(develop)> HelloSol.deployed().then(function(instance)
{return         instance.getValue()}).then(function(value){return
value.toString()});
    '0'
```

Обратите внимание, что теперь в качестве результата мы получили строку в кавычках.

Теперь давайте запишем в сеть значение 777 с помощью функции SetValue:

```
    truffle(develop)> HelloSol.deployed().then(function(instance)
{return instance.setValue(777)});
    { tx:
      '0x735c4f6275d898ac31219fa9be05216aeb6e59b0a86768f654b2c9302f3e2c
     receipt:
      { transactionHash:
         '0x735c4f6275d898ac31219fa9be05216aeb6e59b0a86768f654b2c9302f3
        transactionIndex: 0,
        blockHash:
         '0x7156850ca23bed75d656939f60c766f58c5586c28952bab07291ac55c5a
        blockNumber: 13,
        from: '0x532a094f3c4d2547037275a0d94041f471e756b6',
        to: '0x12321cf58d3a206d33543038c5a8e9ca6e6fd86f',
        gasUsed: 41781,
        cumulativeGasUsed: 41781,
        contractAddress: null,
        logs: [],
        status: true,
```

```
        logsBloom:
         '0x00000000000000000000000000000000000000000000000000000000000000
        v: '0x1c',
        r:
         '0x9184f8da2b7bf5a0c53d68f8ae815bf3daa583562d149a783e119eabcc8
        s:
         '0x1ce2d33acbc6f223a61cfb7d43fb869756d26a70081c6cf308936a6579e
        rawLogs: [] },
     logs: [] }
```

Вы увидите хеш транзакции, номер блока, в котором эта транзакция была размещена, а также количество использованного газа.

Если теперь вызвать функцию getValue, то мы увидим, что значение изменилось:

```
    truffle(develop)> HelloSol.deployed().then(function(instance)
{return instance.getValue()});
    <BN: 309>
```

Или с преобразованием в строку:

```
    truffle(develop)> HelloSol.deployed().then(function(instance)
{return         instance.getValue()}).then(function(value){return
value.toString()});
    '777'
```

Аналогично вызовем функции getString и setString. Сначала мы получаем текущее значение строки, хранящейся в контракте:

```
    truffle(develop)> HelloSol.deployed().then(function(instance)
{return instance.getString()});
    ''
```

Теперь мы записываем в контракт новую строку:

```
    truffle(develop)> HelloSol.deployed().then(function(instance)
{return instance.setString('New string')});
    { tx:
      '0xd567697498bd69797c1774585457f07453d9102f2a60b33b2b2f84a8edc14b
     receipt:
      { transactionHash:
         '0xd567697498bd69797c1774585457f07453d9102f2a60b33b2b2f84a8edc
        transactionIndex: 0,
        blockHash:
         '0x370a14c5b500b084cb426e039824ef95b63e75a9b3101bc8f17a172d6da
        blockNumber: 14,
        from: '0x532a094f3c4d2547037275a0d94041f471e756b6',
        to: '0x12321cf58d3a206d33543038c5a8e9ca6e6fd86f',
        gasUsed: 43563,
        cumulativeGasUsed: 43563,
```

```
        contractAddress: null,
        logs: [],
        status: true,
        logsBloom:
         '0x00000000000000000000000000000000000000000000000000000000000000000
        v: '0x1b',
        r:
         '0x9179841f39a3aafa17ef95129632a537634ec4875a0013d2da59398f992
        s:
         '0x68a35adfc39b13b1d490023c260458c1274c2c33d90d9e352411373ff0c
        rawLogs: [] },
     logs: [] }
```

На консоль выводится хеш транзакции, а также количество использованного газа.
Теперь прочитаем строку:

```
    truffle(develop)> HelloSol.deployed().then(function(instance)
{return instance.getString()});
    'New string'
```

Как видите, строка изменилась.

Вызов функций контракта HelloSol из скрипта JavaScript под управлением Node.js

Когда вы будете создавать реальное децентрализованное приложение DApp с использованием Node.js, то вам нужно будет встроить вызовы функций смарт-контрактов в скрипты JavaScript.

В этом вам поможет модуль truffle-contract, представляющий собой оболочку над Web3. Этот модуль упрощает взаимодействие с функциями смарт-контрактов, выполняя, например, ожидание завершения транзакций с помощью так называемых промисов (Promises).

На сайте https://github.com/trufflesuite/truffle-contract вы найдете руководство по использованию этого модуля.

Скрипты, приведенные в этом разделе, были отлажены с использованием Truffle версии 4.1.15, а также фреймворка Web3 версии 0.20.6. Далее мы приведем скрипты, работающие с Web3 версии 1.0.x.

Как мы уже говорили, версию Truffle и компилятора solc-js (используется Truffle по умолчанию) можно определить с помощью следующей команды:

```
$ truffle version
Truffle v4.1.15 (core: 4.1.15)
Solidity v0.4.25 (solc-js)
```

Если у вас установлен Truffle версии 4.1.x, то он использует фреймворк Web3 версии 0.20.x. В этом случае вы сможете определить версию фреймворка в командном приглашении truffle develop:

```
> console.log(web3.version.api);
0.20.6
```

Что же касается Truffle версии 5.x, то вместе с ним устанавливается бета-версия фреймворка Web3. При подготовке руководства была доступна версия Truffle 5.0.7:

```
$ truffle version
Truffle v5.0.7 (core: 5.0.7)
Solidity v0.5.0 (solc-js)
Node v10.15.3
```

В этом случае версию Web3 нужно определять по-другому:

```
> web3.version
'1.0.0-beta.37'
```

Скрипты из этого руководства проверялись на бета-версиях Web3 1.0.0-beta.37 и 1.0.0-beta.48. Чтобы установить нужную вам версию Web3, используйте команду следующего вида:

```
$ npm install web3@1.0.0-beta.48
```

Установка модуля truffle-contract

Чтобы установить модуль truffle-contract, воспользуйтесь менеджером пакетов npm:

```
$ npm install truffle-contract
```

Вызов функций контракта getValue и getString

Функции контракта могут только читать свойства и переменные контракта, не изменяя состояние сети блокчейна, либо они могут изменять это состояние, выполняя транзакции.

В листинге 7.4. мы привели исходный текст скрипта hello_sol_get.js, который вызывает функции getValue и getString нашего контракта HelloSol.

Листинг 7.4. Файл hello_sol_get.js

```
// node hello_sol_get.js HelloSol

var contract_name = process.argv[2];
var contract_address = process.argv[3];

var Web3 = require('web3');
var contract = require("truffle-contract");
var path = require('path');

var provider = new Web3.providers.HttpProvider("http://
localhost:9545");

var contractJSON = require(path.join(__dirname, 'build/
contracts/' + contract_name + '.json'));
var contract = contract(contractJSON);
contract.setProvider(provider);

contract.deployed().then(function(instance) {
   return instance.getValue.call();
}).then(function(result) {
   console.log('getValue() returns: ' + result);
}, function(error) {
   console.log(error);
});

contract.deployed().then(function(instance) {
   return instance.getString.call();
}).then(function(result) {
   console.log('getString() returns: ' + result);
}, function(error) {
   console.log(error);
});
```

В качестве параметра мы передаем скрипту имя класса контракта. Сценарий выводит на консоль текущие значения числа и строки, записанные в поля контракта:

```
$ node hello_sol_get.js HelloSol
getValue() returns: 777
```

```
getString() returns: New string
```

Рассмотрим работу этого скрипта. Прежде всего мы подключаем нужные нам модули:

```
var Web3 = require('web3');
var contract = require("truffle-contract");
var path = require('path');
```

Далее мы получаем сетевой интерфейс для взаимодействия с отладочным приватным блокчейном Truffle Develop, встроенным в Truffle:

```
var provider = new Web3.providers.HttpProvider("http://
localhost:9545");
```

Теперь нам нужно загрузить из артефактов json-файл, созданный из нашего смарт-контракта на этапе компиляции.

```
var contractJSON = require(path.join(__dirname, 'build/
contracts/' + contract_name + '.json'));
```

Загрузив этот файл, мы создаем на его основе объект контракта и устанавливаем для этого объекта сетевое взаимодействие с отладочным приватным блокчейном:

```
var contract = contract(contractJSON);
contract.setProvider(provider);
```

Теперь мы готовы вызывать методы getValue и getString:

```
contract.deployed().then(function(instance) {
   return instance.getValue.call();
}).then(function(result) {
   console.log('getValue() returns: ' + result);
}, function(error) {
   console.log(error);
});

contract.deployed().then(function(instance) {
   return instance.getString.call();
}).then(function(result) {
   console.log('getString() returns: ' + result);
}, function(error) {
   console.log(error);
});
```

Обратите внимание, что, так как функции контракта getValue и getString не изменяют состояние блокчейна, здесь мы при вызове функций контракта используем функцию call без параметров.

Также обращаем ваше внимание, что скрипт hello_sol_get.js нигде **явным образом не обращается к адресу контракта**. Из артефакта HelloSol.json всегда берется адрес самого

нового опубликованного контракта и подставляется при вызове методов Web3. Это очень удобно для отладки, когда вы в локальной сети многократно изменяете свой контракт, запускаете его компиляцию и публикацию.

Вызов функций контракта setValue и setString

Теперь посмотрим, как оболочка truffle-contract упрощает вызов функций, изменяющих состояние сети и выполняющих транзакции.

Мы подготовили скрипт hello_sol_set.js, вызывающий методы setValue и setString контракта HelloSol (листинг 7.5.).

Листинг 7.5. Файл hello_sol_set.js

```
// node hello_sol_set.js HelloSol  888 "strstr"
var contract_name = process.argv[2];
var newValue = process.argv[3];
var newString = process.argv[4];

var Web3 = require('web3');
var contract = require("truffle-contract");
var path = require('path');
var provider = new
Web3.providers.HttpProvider("http://127.0.0.1:9545/");

var contractJSON = require(path.join(__dirname, 'build/
contracts/' + contract_name + '.json'));
var contract = contract(contractJSON);
contract.setProvider(provider);

var web3 = new Web3(provider);

var myAccount;
web3.eth.getAccounts().then(function(result){
 myAccount = result[0];
}, function(error) {
 console.log(error);
});

contract.deployed().then(function(instance) {
       return instance.setValue(newValue, {from:myAccount,
gas:4000000});
}).then(function(result) {
    console.log('setValue() tx: ' + result.tx + ' Gas used:'
+ result.receipt.gasUsed);
   //console.log("setValue result: %o", result);
}, function(error) {
   console.log(error);
});

contract.deployed().then(function(instance) {
```

```
          return  instance.setString(newString,  {from:myAccount,
gas:4000000});
    }).then(function(result) {
     console.log('setString()  tx:  ' + result.tx + ' Gas used:'
+ result.receipt.gasUsed);
     console.log("setString result: %o", result);
    }, function(error) {
       console.log(error);
    });
```

При запуске скрипта hello_sol_set.js ему нужно передать два параметра, первый из которых должен быть числом, а второй – строкой:

```
node hello_sol_set.js HelloSol  888 "Строка 3"
```

Вначале этот скрипт, как и описанный выше hello_sol_get.js (листинг 7.3.), подключает необходимые модули, создает объект провайдера сетевого соединения, загружает файл HelloSol.json из артефактов, создает объект контракта и подключает к нему провайдер.

Далее нам нужно определить адрес основного аккаунта тестового блокчейна Truffle:

```
var myAccount;
web3.eth.getAccounts().then(function(result){
 myAccount = result[0];
}, function(error) {
 console.log(error);
});
```

Здесь базе провайдера мы создаем объект Web3. Функция web3.eth.getAccounts возвращает промис (promise). С учетом этого приведенный выше фрагмент кода сохраняет в переменной myAccount адрес первого аккаунта тестового блокчейна.

Далее мы вызываем метод setValue, передавая ему в качестве первого параметра новое значение, которое нужно будет записать в поле контракта, а в качестве второго – структуру, определяющую аккаунт, от которого будет выполнена транзакция, а также количество газа, которое мы готовы потратить на выполнение транзакции:

```
contract.deployed().then(function(instance) {
       return  instance.setValue(newValue,  {from:myAccount,
gas:4000000});
}).then(function(result) {
    console.log('setValue()  tx:  ' + result.tx + ' Gas used:'
+ result.receipt.gasUsed);
    // console.log("setValue result: %o", result);
}, function(error) {
    console.log(error);
});
```

Обратите внимание, что оболочка truffle-contract обеспечивает здесь для нас ожидание выполнения транзакции.

После вызова функции setValue объект result будет содержать хеш транзакции, квитанцию о выполнении транзакции и массив записей журнала. Эта информация выводится на консоль:

```
    console.log("setString result: %o", result);
```

Вот что вы увидите при запуске:

```
    $ node hello_sol_set.js HelloSol  888 "Строка 3"
    myAccount ->undefined
    setValue()                                                    tx:
0x305c4e392f4f046301c498f8098f6dc65483f45002c35c3423c031c5f1698598
Gas used:26781
    setString()                                                   tx:
0x0de4ba46d8344b2329598b4a6717e66dd812a68d4bcf15760601cc93bf0e70a5
Gas used:33876
    setString result: { tx:
      '0x0de4ba46d8344b2329598b4a6717e66dd812a68d4bcf15760601cc93bf0e70
     receipt:
      { transactionHash:
         '0x0de4ba46d8344b2329598b4a6717e66dd812a68d4bcf15760601cc93bf0
        transactionIndex: 0,
        blockHash:
         '0xea11d10f5f11427b1319594bf13c62fd1e1ee0e06f8fd32af741338b998
        blockNumber: 14,
        from: '0x532a094f3c4d2547037275a0d94041f471e756b6',
        to: '0x19497e3afb1f2ad1b6d3179a52598f5beabb03d7',
        gasUsed: 33876,
        cumulativeGasUsed: 33876,
        contractAddress: null,
        logs: [ [length]: 0 ],
        status: true,
        logsBloom:
         '0x00000000000000000000000000000000000000000000000000000000000000000
        v: '0x1b',
        r:
         '0xc895946082a67548a447854a422568945e20c2e3ea0b88fe9f4ed469378
        s:
         '0x7bab153fd218262877d7690b18cc7ed11c3a3cc2c44be899730082875df
        rawLogs: [ [length]: 0 ] },
     logs: [ [length]: 0 ] }
```

Обратите внимание на квитанцию о выполнении транзакции, полученную от функции web3.eth.getTransactionReceipt(hash). Мы рассказывали об этом на уроке 5 нашего курса.

Мы также выводим на консоль хеш транзакции и количество использованного газа при вызове метода setValue.

Аналогичным образом мы вызываем метод setString:

```
    contract.deployed().then(function(instance) {
```

```
          return instance.setString(newString, {from:myAccount,
gas:4000000});
    }).then(function(result) {
       console.log('setString() tx: ' + result.tx + ' Gas used:'
+ result.receipt.gasUsed);
    }, function(error) {
       console.log(error);
    });
```

Изменение контракта и повторная публикация

На следующем шаге мы внесем изменения в контракт HelloSol.sol, а затем выполним его повторную компиляцию и публикацию. Убедимся, что это очень просто.

Внесем небольшое изменение в метод getValue исходного текста контракта HelloSol.sol (листинг 7.6.).

Листинг 7.6. Файл HelloSol.sol

```
pragma solidity ^0.4.10;

contract HelloSol {
   string savedString;
   uint savedValue;

   function setString( string memory newString ) public {
       savedString = newString;
   }
   function getString() public view returns( string memory ) {
       return savedString;
   }
   function setValue( uint newValue ) public {
       savedValue = newValue;
   }
   function getValue() public view returns( uint ) {
       return savedValue + 1;
   }
}
```

Теперь метод getValue возвращает сохраненное числовое значение, увеличенное на единицу.

После того как вы внесете это изменение и сохраните текст контракта, запустите компиляцию в приглашении Truffle:

```
truffle(develop)> compile
Compiling ./contracts/HelloSol.sol…
Writing artifacts to ./build/contracts
```

Когда компиляция завершится, опубликуйте контракт командой migrate с параметром –reset:

```
truffle(develop)> migrate –reset
Using network 'develop'.
```

Убедитесь, что теперь наш контракт возвращает сохраненное числовое значение, увеличенное на единицу:

```
$ $ node hello_sol_set.js HelloSol 12345 'test string 12345'
```

```
    setValue()                                                    tx:
0xe73236396af416a3bbc28274366e84bbca5585344a01a48ae3aa6a6a714be8a2
Gas used:41781
    setString()                                                   tx:
0xeab83eb0dc5dbeac4f0123a6974941b78f4d9e7ceac7224a0557229da33e2a6d
Gas used:44011
    setString result: { tx:
      '0xeab83eb0dc5dbeac4f0123a6974941b78f4d9e7ceac7224a0557229da33e2a
     receipt:
      { transactionHash:
         '0xeab83eb0dc5dbeac4f0123a6974941b78f4d9e7ceac7224a0557229da33
        transactionIndex: 0,
        blockHash:
         '0x70b3480272083f0033499a8cb2cd836c14b96303d1488fb43d4eaefba0b
        blockNumber: 22,
        from: '0x532a094f3c4d2547037275a0d94041f471e756b6',
        to: '0x7c1be32e6523c837fd258393e2bd26972c5c25bf',
        gasUsed: 44011,
        cumulativeGasUsed: 44011,
        contractAddress: null,
        logs: [ [length]: 0 ],
        status: true,
        logsBloom:
         '0x00000000000000000000000000000000000000000000000000000000000
        v: '0x1b',
        r:
         '0x47b12603a4031e72970ca42ef7b5e5085be8e8378399eb37620645cce55
        s:
         '0x26806e1c38521546ca315d86a265ca2f3a7086c25fcf779ef9160a0cac2
        rawLogs: [ [length]: 0 ] },
     logs: [ [length]: 0 ] }

    $ node hello_sol_get.js HelloSol
    getValue() returns: 12346
    getString() returns: test string 12345
```

Как видите, процесс компиляции и публикации контракта очень прост и сводится к запуску всего двух команд в приглашении Truffle.

Работа с Web3 версии 1.0.x

По умолчанию устанавливается среда разработки Truffle 5.0.x и фреймворк Web3 версии 1.0.x (как мы уже говорили, на момент создания этого руководства была доступна только бета-версия данного фреймворка).

Для того чтобы показать, как скрипты JavaScript, работающие под управлением Node.js, могут обращаться к фреймворку Web3 версии 1.0.x, мы подготовили два скрипта. Первый из них обращается к функциям смарт-контракта, изменяющим состояние сети, а второй – не изменяющим их.

Вносим изменения в смарт-контракт HelloSol

Вместе с Truffle версии 5.0.x устанавливается компилятор solc версии 0.5.x:

```
$ truffle version
Truffle v5.0.11 (core: 5.0.11)
Solidity v0.5.0 (solc-js)
Node v10.15.3
Web3.js v1.0.0-beta.37
```

Поэтому нам нужно внести небольшие изменения в наш смарт-контракт HelloSol, а именно в его первую строку. В ней нам нужно указать версию компилятора как ^5.0.0 (листинг 7.7.).

Листинг 7.7. Файл HelloSol.sol для solc версии 5.0.x

```
pragma solidity ^0.5.0;

contract HelloSol {
   string savedString;
   uint savedValue;

   function setString( string memory newString ) public {
       savedString = newString;
   }
   function getString() public view returns( string memory ) {
       return savedString;
   }
   function setValue( uint newValue ) public {
       savedValue = newValue;
   }
   function getValue() public view returns( uint ) {
       return savedValue;
   }
}
```

Скрипты для вызова методов контракта

В скрипте hello_sol_set_tr.js (листинг 7.8) мы вызываем функции setValue и setString нашего контракта.

Листинг 7.8. Файл hello_sol_set_tr.js

```
// node hello_sol_set_tr.js HelloSol 5777 ""
let Web3 = require('web3')
let fs = require('fs')
let request = require('request')

var contract_name = process.argv[2];
var network_id = process.argv[3];
var unlock_password = process.argv[4];

console.log('Contract script: ' + contract_name);

var path = require('path');
var contractJSON = require(path.join(__dirname, 'build/
contracts/' + contract_name + '.json'));
var decoded = JSON.parse(JSON.stringify(contractJSON.networks,
undefined, 2));
var contract_address = decoded[network_id].address;
var abi = contractJSON.abi;
console.log('contract_address: ' + contract_address);

var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:9545"));

var version = web3.version;
console.log('Web3 version: ' + version);

let myContract = new web3.eth.Contract(abi, contract_address);

web3.eth.getCoinbase()
.then(coinbase => {
 myCoinbase = coinbase;
 console.log('coinbase: ' + coinbase);
 return coinbase;
})
.then(function (account) {
       return       web3.eth.personal.unlockAccount(account,
unlock_password, 600)
})
.then(function (unlocked) {
 console.log('Unlocked: ' + unlocked);

 myContract.methods.setValue(307309).send({from: myCoinbase})
```

```
     .once('setValue transactionHash', (hash) => {
       console.log('hash: ' + hash);
     })
     .on('setValue confirmation', (confNumber) => {
       console.log('confNumber: ' + confNumber);
     })
     .on('receipt', (receipt) => {
       console.log(JSON.stringify(receipt, undefined, 2));
     })
     .then(function () {

               myContract.methods.setString("Test          string
307309").send({from: myCoinbase})
     .once('transactionHash', (hash) => {
       console.log('setString hash: ' + hash);
     })
     .on('confirmation', (confNumber) => {
       console.log('setString confNumber: ' + confNumber);
     })
     .on('receipt', (receipt) => {
       console.log(JSON.stringify(receipt, undefined, 2));
     })
    })

    })
    .catch(function (error) {
     console.error(error);
    });
```

В целом скрипт аналогичен скрипту call_contract_set_promise.js, приведенному в листинге 6.6. предыдущего урока. Отличия лишь в том, что этот скрипт получает необходимые ему параметры работы (адрес контракта и abi) из артефактов Truffle.

В качестве параметров при запуске скрипту нужно передать имя смарт-контракта, а также идентификатор сети. Для сети Truffle этот идентификатор равен 5777. Пароль нужно указывать как пустую строку:

```
$ node hello_sol_set_tr.js HelloSol 5777 ""
```

Скрипт извлекает адрес контракта и abi из файла, имя которого передается ему в качестве параметра, и расширением имени json. Для извлечения адреса контракта используется идентификатор сети network_id:

```
var path = require('path');
var contractJSON = require(path.join(__dirname, 'build/
contracts/' + contract_name + '.json'));
var decoded = JSON.parse(JSON.stringify(contractJSON.networks,
undefined, 2));
var contract_address = decoded[network_id].address;
var abi = contractJSON.abi;
```

```
    console.log('contract_address: ' +  contract_address);
```

Скрипт hello_sol_get_tr.js вызывает функции контракта getValue и getString (листинг 7.9.).

Листинг 7.9. Файл hello_sol_get_tr.js

```
    // node hello_sol_get_tr.js HelloSol 5777
    let Web3 = require('web3')
    let fs = require('fs')
    let request = require('request')

    var contract_name = process.argv[2];
    var network_id = process.argv[3];

    console.log('Contract script: ' + contract_name);

    var path = require('path');
    var  contractJSON  =  require(path.join(__dirname,  'build/
contracts/' + contract_name + '.json'));
    var decoded = JSON.parse(JSON.stringify(contractJSON.networks,
undefined, 2));
    var contract_address = decoded[network_id].address;
    var abi = contractJSON.abi;
    console.log('contract_address: ' +  contract_address);

    var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:9545"));

    var version = web3.version;
    console.log('Web3 version: ' + version);

    let myContract = new web3.eth.Contract(abi, contract_address);

    web3.eth.getCoinbase()
    .then(coinbase => {
     myCoinbase = coinbase;
     console.log('coinbase: ' + coinbase);
     return coinbase;
    })
    .then(function () {

     myContract.methods.getValue().call({from: contract_address},
(error, result) =>
     {
        if(!error){
          console.log('getValue result: ' + result);
          console.log(JSON.stringify(result, undefined, 2));
        } else{
          console.log(error);
```

```
        }
     });

                        myContract.methods.getString().call({from:
contract_address}, (error, result) =>
     {
        if(!error){
          console.log('getString result: ' + result);
    //       console.log(JSON.stringify(result, undefined, 2));
        } else{
          console.log(error);
        }
     });
    })
    .catch(function (error) {
     console.error(error);
    });
```

Он аналогичен скрипту call_contract_get_promise.js, приведенному в листинге 6.7. предыдущего урока. Отличия лишь в том, что скрипт hello_sol_get_tr.js получает адрес контракта и abi из артефактов Truffle.

Тестирование в Truffle

Среда разработки Truffle содержит очень удобные средства тестирования смарт-контрактов. Они помогут вам в отладке ваших распределенных приложений DApp.

Вы можете создавать тесты как на Solidity, так и на JavaScript. Подробное описание методик составления тестов для смарт-контрактов вы найдете в документации Truffle по адресу https://truffleframework.com/docs/truffle/testing/testing-your-contracts. На этом уроке мы приведем два небольших сценария тестирования нашего смарт-контракта HelloSol.

Тест на Solidity

Первый тест мы составим на языке Solidity. Запишите файл TestHelloSol.sol (листинг 7.10.) в подкаталог test каталога HelloSol, созданного нами для работы со смарт-контрактом HelloSol.sol.

Листинг 7.10. Файл TestHelloSol.sol

```
pragma solidity ^0.5.0;
import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/HelloSol.sol";

contract TestHelloSol {
 function testSetValue() public {
  HelloSol helloSol = HelloSol(DeployedAddresses.HelloSol());
   helloSol.setValue(77);
   uint expectedValue = 77;
   uint actualValue = helloSol.getValue();
    Assert.equal(actualValue, expectedValue, "Value should be
equal 77");
 }

 function testSetString() public {
  HelloSol helloSol = HelloSol(DeployedAddresses.HelloSol());
   helloSol.setString("My test string");
   string memory expectedString = "My test string";
   string memory actualString = helloSol.getString();
    Assert.equal(actualString, expectedString, "String should
be equal 'My test string'");
 }
}
```

Мы будем тестировать работу функций setValue, getValue, setString и getString. Суть тестов заключается в том, что мы сначала вызываем функцию setValue, передавая ей в качестве значения число 77, а затем проверяем, что оно установилось правильно при помощи функции getValue.

Аналогичные проверки выполняются для функций setString и getString, при этом используется тестовая строка My test string.

Первая строка файла TestHelloSol.sol задает версию компилятора solc. Она аналогична первой строке нашего смарт-контракта HelloSol.sol.

Далее мы импортируем библиотеки функций Assert.sol, DeployedAddresses.sol и, наконец, наш смарт-контракт HelloSol.sol:

```
import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/HelloSol.sol";
```

Библиотека Assert.sol нужна для выполнения проверочных действий (assertion functions). В документации есть ссылка на перечень всех доступных функций.

С помощью библиотеки DeployedAddresses.sol можно работать с адресами опубликованных (мигрированных) контрактов.

Заметим, что с помощью ключевого слова import вы можете подключать к своему смарт-контракту различные готовые библиотеки функций. Это позволит вам быстро добавлять в смарт-контракт необходимую функциональность. Кроме того, применяя готовые отлаженные библиотеки, вы уменьшите риск внесения ошибок в свой смарт-контракт.

Функция testSetValue получает и сохраняет в переменной helloSol адрес нашего смарт-контракта, а затем вызывает функцию setValue, устанавливая значение 77:

```
function testSetValue() public {
 HelloSol helloSol = HelloSol(DeployedAddresses.HelloSol());
 helloSol.setValue(77);
 uint expectedValue = 77;
 uint actualValue = helloSol.getValue();
  Assert.equal(actualValue, expectedValue, "Value should be
equal 77");
}
```

Далее вызывается функция getValue, и полученное от нее значение сравнивается при помощи Assert.equal. Если значения не совпадут, на консоли будет показано предупреждающее сообщение.

Функция testSetString выполняет аналогичный тест с текстовыми строками.

Как запустить тесты на выполнение?

Очень просто, достаточно ввести команду test в приглашении Truffle develop:

```
truffle(develop)> test
Using network 'develop'.

Compiling your contracts…

> Compiling ./test/TestHelloSol.sol
> Artifacts written to /tmp/test-11938-2475-1olwk2l.sw3e
> Compiled successfully using:
  - solc: 0.5.0+commit.1d4f565a.Emscripten.clang

 TestHelloSol
√ testSetValue (70ms)
√ testSetString (90ms)
```

```
  2 passing (5s)
```

Ниже мы показали, какое будет сообщение, если тесты будут провалены:

```
  0 passing (5s)
  2 failing

  1) TestHelloSol
       testSetValue:
Error: Value should be equal 77 (Tested: 77, Against: 78)

      at result.logs.forEach.log (/home/book/.nvm/versions/node/
v10.15.3/lib/node_modules/truffle/build/webpack:/packages/
truffle-core/lib/testing/soliditytest.js:70:1)
       at Array.forEach (<anonymous>)
      at processResult (/home/book/.nvm/versions/node/v10.15.3/
lib/node_modules/truffle/build/webpack:/packages/truffle-core/
lib/testing/soliditytest.js:68:1)
                at   process._tickCallback   (internal/process/
next_tick.js:68:7)

  2) TestHelloSol
       testSetString:
Error: String should be equal 'My test string' (Tested: My test
string, Against: My test string 8)

      at result.logs.forEach.log (/home/book/.nvm/versions/node/
v10.15.3/lib/node_modules/truffle/build/webpack:/packages/
truffle-core/lib/testing/soliditytest.js:70:1)
       at Array.forEach (<anonymous>)
      at processResult (/home/book/.nvm/versions/node/v10.15.3/
lib/node_modules/truffle/build/webpack:/packages/truffle-core/
lib/testing/soliditytest.js:68:1)
                at   process._tickCallback   (internal/process/
next_tick.js:68:7)
```

Тут мы намеренно внесли ошибки в контракт тестирования TestHelloSol.sol.

Ниже мы представили еще несколько функций из библиотеки truffle/Assert.sol, чтобы у вас было некоторое представление о том, какие есть возможности для тестирования:

- equal(string)
- notEqual(string)
- isEmpty(string)
- isNotEmpty(string)
- equal(bytes32)
- notEqual(bytes32)
- isZero(bytes32)
- isNotZero(bytes32)
- equal(address)

• notEqual(address)
• balanceEqual(address a, uint b, string memory message)
• balanceIsZero(address a, string memory message)

Подобные проверочные функции есть для всех типов данных Solidity. Полный список функций можно найти здесь:

https://github.com/trufflesuite/truffle/blob/develop/packages/truffle-core/lib/testing/Assert.sol

Тест на JavaScript

Теперь давайте добавим тесты, составленные на языке JavaScript. Добавьте в каталог test вашего проекта файл TestHelloSol.js (листинг 7.11.).

Листинг 7.11. Файл TestHelloSol.js

```
const HelloSol = artifacts.require("HelloSol");

contract("getValue test", async accounts => {
 it("getValue should return 777", async () => {
   let instance = await HelloSol.deployed();
   await instance.setValue(777);
   let val = await instance.getValue.call();
   assert.equal(val.valueOf(), 777);
 });
 it("getString should return 'My test string'", async () => {
   let instance = await HelloSol.deployed();
   await instance.setString("My test string");
   let str = await instance.getString.call();
   assert.equal(str, "My test string");
 });
});
```

Как видите, здесь выполняются точно такие же тесты с функциями getValue, setValue, getString, setString, что и в предыдущем разделе.

Вызов artifacts.require позволяет указать скрипту тестирования, с каким контрактом нужно работать. Мы по очереди запускаем два теста, дожидаясь завершения выполнения функций нашего контракта с помощью await.

Вы также можете использовать для составления тестов JavaScript и промисы, это описано в документации. Но, на наш взгляд, применение await делает исходный текст тестов более наглядным.

Итоги урока

На этом уроке вы приступили к изучению интегрированной среды разработки смарт-контрактов Truffle. Вы научились устанавливать этот инструмент, создавать с его помощью смарт-контракты, публиковать, вызывать функции смарт-контрактов, а также тестировать смарт-контракты.

Основной целью предыдущих уроков было изучение принципов разработки и публикации смарт-контрактов, чтобы понять, как это все работает. На седьмом же уроке вы приступили к использованию Truffle – очень удобного инструмента программиста, создающего смарт-контракты Solidity для реальных проектов.

Урок 8. Типы данных Solidity

Цель урока: изучить основные типы данных Solidity – логические типы данных, беззнаковые целые числа и числа со знаком, адреса, переменные сложных типов, массивы, перечисления, структуры и словари.

Практические задания: написать смарт-контракты, использующие основные типы данных Solidity.

В первых семи уроках нашего курса вы научились создавать приватный блокчейн Ethereum, публиковать в нем простейшие контракты и вызывать их функции. Теперь, пользуясь IDE Truffle, мы займемся изучением типов данных в языке программирования Solidity, предназначенном для создания смарт-контрактов Ethereum.

Solidity **представляет собой статически типизированный язык. В нем реализованы наследование,** библиотеки, пользовательские типы данных и другие полезные возможности.

Если вы программировали на таких языках, как JavaScript, C++ или Python, то без особого труда сможете освоить и Solidity.

Как мы уже говорили, программы Solidity работают в среде виртуальной машины EVM (Ethereum Virtual Machine). Вы можете представить себе EVM как огромный компьютер, узлы которого разбросаны по всему миру. Они выполняют синхронно код смарт-контрактов на всех узлах, при этом данные, доступные программам смарт-контрактов, также реплицируются.

Также EVM можно представить себе как глобальную базу данных, которая хранит и реплицирует состояния. При выполнении транзакций состояние сети изменяется.

EVM выполняет не исходный код Solidity, а компилированный бинарный код opcodes (operation codes). На предыдущих уроках вы уже получали бинарный код смарт-контрактов при помощи компилятора solc.

В рамках нашего краткого курса мы, конечно, не сможем охватить все тонкости использования Solidity. При необходимости обращайтесь к документации. Для Solidity версии 0.5.7 ее можно найти здесь: https://solidity.readthedocs.io/en/v0.5.7/.

Надо сказать, что EVM представляет собой 256-разрядную машину. Это позволяет работать с очень большими числами, но в ряде случаев может привести к неэкономному расходу памяти.

Контракт для изучения типов данных

Для изучения типов данных мы подготовим новый контракт SolDataTypes.sol. Скопируйте его из контракта HelloSol.sol и расположите в том же каталоге /home/book/HelloSol/contracts.

Исходный текст контракта SolDataTypes.sol вы найдете в листинге 8.1.

Листинг 8.1. Контракт SolDataTypes1.sol (вариант 1)

```
pragma solidity ^0.5.0;

contract SolDataTypes {
   bool isReady;
   string savedString;
   uint savedValue;
   uint8[10] boxPrice;
   uint8[] dBoxPrice;

   address owner;

   enum doorState { Open, Closed, Blocked }
   doorState myDoor;

   function openDoor() public {
       myDoor = doorState.Open;
   }

   function closeDoor() public {
       myDoor = doorState.Closed;
   }

   function blockDoor() public {
       myDoor = doorState.Blocked;
   }

   function getDoorState() public view returns ( doorState ) {
       return myDoor;
   }

   constructor() public {
       owner = msg.sender;
   }

   function setDBoxPrice(uint8 price) public {
       dBoxPrice.push(price);
   }

     function getDBoxPrices() public view returns( uint8[]
memory ) {
```

```
            return dBoxPrice;
        }

        function setBoxPrice(uint idBox, uint8 price) public {
            boxPrice[idBox] = price;
        }

         function getBoxPrices() public view returns( uint8[10]
memory ) {
            return boxPrice;
        }

        function init() public {
            isReady = true;
        }

        function isInitialized() public view returns ( bool ) {
            return isReady;
        }

        function getBalance() public view returns ( uint ) {
            address myAddress = msg.sender;
            uint balance = myAddress.balance;
            return balance;
        }

        function setString( string memory newString ) public {
            savedString = newString;
        }
        function getString() public view returns( string memory ) {
            return savedString;
        }
        function setValue( uint newValue ) public {
            savedValue = newValue;
        }
        function getValue() public view returns( uint ) {
            return savedValue;
        }
    }
```

Таким образом мы добавили в наш проект еще один смарт-контракт. Чтобы Truffle смог его публиковать, создайте в каталоге /home/book/HelloSol/migrations файл 3_soldatatypes_migrations.js (листинг 8.2.).

Листинг 8.2. Файл 3_soldatatypes_migrations.js

```
var MyContract = artifacts.require("SolDataTypes");

module.exports = function(deployer) {
 // deployment steps
```

```
 deployer.deploy(MyContract);
};
```

Теперь в приглашении Truffle выполните повторную компиляцию контрактов и повторную их публикацию (миграцию):

```
truffle(develop)> compile –all
Compiling ./contracts/SolDataTypes.sol…
Writing artifacts to ./build/contracts

truffle(develop)> migrate –reset
⚠  Important ⚠

If you're using an HDWalletProvider, it must be Web3 1.0 enabled
or your migration will hang.

Starting migrations…
=
> Network name:    'develop'
> Network id:      4447
> Block gas limit: 6721975

1_initial_migration.js
=

  Replacing 'Migrations'
  –
                                   >                transaction
hash:    0x52d244c6dade300c55d37a7550489edf99c6ba47ebb1254dc19ca6995aae
  > Blocks: 0            Seconds: 0
                                     >                  contract
address:    0x2F577F828d8A4D91Eed8717932D312f79102f141
                                                          >
account:             0x532A094f3C4D2547037275a0d94041f471E756B6
  > balance:             99.941922892
  > gas used:            284908
  > gas price:           20 gwei
  > value sent:          0 ETH
  > total cost:          0.00569816 ETH

  > Saving migration to chain.
  > Saving artifacts
  –
  > Total cost:          0.00569816 ETH

2_hellosol_migrations.js

  Replacing 'HelloSol'
  –
```

```
                                   >                transaction
hash:    0x0ab074db0e13d435e4c9c23f2ab19d61172b3d3fc9f216bd4cef5d7e1af2
      > Blocks: 0            Seconds: 0
                                    >                  contract
address:    0x50D217412d1f8ee915104E29cbB5f7F1f8309C85
                                                                  >
account:            0x532A094f3C4D2547037275a0d94041f471E756B6
      > balance:            99.934890412
      > gas used:           309590
      > gas price:          20 gwei
      > value sent:         0 ETH
      > total cost:         0.0061918 ETH

      > Saving migration to chain.
      > Saving artifacts
      -
      > Total cost:          0.0061918 ETH

    3_soldatatypes_migrations.js
    =

      Replacing 'SolDataTypes'
      -
                                   >                transaction
hash:    0xe5e23fa032e1f3807a815f82ee5ea1abe199f40e8dad0aaded09ca2f8dd9
      > Blocks: 0            Seconds: 0
                                    >                  contract
address:    0xeE31C6a39F49688186A0192D21A4c4fF8452e68D
                                                                  >
account:            0x532A094f3C4D2547037275a0d94041f471E756B6
      > balance:            99.922302532
      > gas used:           602360
      > gas price:          20 gwei
      > value sent:         0 ETH
      > total cost:         0.0120472 ETH

      > Saving migration to chain.
      > Saving artifacts
      -
      > Total cost:          0.0120472 ETH

    Summary
    =
    > Total deployments:   3
    > Final cost:          0.02393716 ETH
```

Функция init нашего смарт-контракта SolDataTypes просто записывает в поле isReady значение true, имитируя некую инициализацию. С помощью функции isInitialized можно проверить, была выполнена инициализация, или нет.

После публикации контракта вызовите функцию isInitialized в приглашении Truffle:

```
    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.isInitialized()});
    false
```

Как видите, сразу после публикации контракта эта функция вернула нам значение false. Давайте теперь выполним инициализацию, вызвав функцию init:

```
    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.init()});
    { tx:
      '0x9a8acc754039252adc9ceb3e7b355a01eb4dfb34760e01a0b399928d2b767a
     receipt:
      { transactionHash:
         '0x9a8acc754039252adc9ceb3e7b355a01eb4dfb34760e01a0b399928d2b7
        transactionIndex: 0,
        blockHash:
         '0x4699e65d0f386997a8ad83e8301bfb821d1be9a003b9bcdedc1fbe11527
        blockNumber: 29,
        from: '0x532a094f3c4d2547037275a0d94041f471e756b6',
        to: '0xee31c6a39f49688186a0192d21a4c4ff8452e68d',
        gasUsed: 41892,
        cumulativeGasUsed: 41892,
        contractAddress: null,
        logs: [],
        status: true,
        logsBloom:
         '0x00000000000000000000000000000000000000000000000000000000000000
        v: '0x1b',
        r:
         '0x8208aecee9431f97d9ece1ce36f9c81d5e9f7f2a369a294d1e21da7adab
        s:
         '0x3a3ce0c524bd65ffd82566368ba205871ed452d77e7b157bc978112e87b
        rawLogs: [] },
     logs: [] }
```

Теперь если проверить инициализацию, то функция isInitialized вернет нам значение true:

```
    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.isInitialized()});
    true
```

Контракт готов, и мы можем переходить к изучению типов данных в Solidity.

Логические типы данных

Логические типы данных могут принимать значения, равные true или false. Для их обозначения зарезервировано ключевое слово bool.

В нашем контракте SolDataTypes.sol определено поле isReady типа bool:

```
pragma solidity ^0.5.0;
contract SolDataTypes {
   bool isReady;
…
}
```

Функция init записывает значение true в поле isReay, а функция isInitialized – возвращает значение из этого поля:

```
function init() public {
  isReady = true;
}

function isInitialized() public view returns ( bool ) {
  return isReady;
}
```

Обратите внимание, что в функции isInitialized указан тип возвращаемого значения как bool.

Для логических типов данных определены следующие операторы:

- == (равно)
- != (не равно)
- ! (логическое отрицание НЕ)
- && (логическая операция И)
- || (логическая операция ИЛИ)

Беззнаковые целые числа и целые числа со знаком

Для обозначения беззнаковых целых чисел зарезервировано ключевое слово uint. Целые числа со знаком обозначаются как int. Вы также можете указывать необходимую разрядность числовых значений, например int8, int16, и т.д. с шагом 8 до int265 (аналогично для uint). При этом int и uint обозначают то же самое, что и int256 и uint256 соответственно.

В контракте SolDataTypes.sol мы определили поле savedValue типа uint, а также функции setValue и getValue, с помощью которых можно установить и прочитать содержимое этого поля соответственно:

```
contract SolDataTypes {
…
    uint savedValue;
…
    function setValue( uint newValue ) public {
        savedValue = newValue;
    }
    function getValue() public view returns( uint ) {
        return savedValue;
    }
}
```

В определении функции gatValue указан тип возвращаемого значения как uint.

Для целых чисел определены операторы:

• сравнения <=, <, ==, !=, >=, >

• битовые операторы `&`, `|`, ^ (Исключающее ИЛИ), ~ (НЕ)

• арифметические операторы +, −, унарный −, унарный +, *, /, `%` (остаток от деления), ** (возведение в степень), << (сдвиг влево), >> (сдвиг вправо)

Числа с фиксированной запятой

Для обозначения чисел с фиксированной запятой зарезервированы ключевые слова fixed (знаковые) и ufixed (беззнаковые). На момент создания этого курса в Solidity еще не была добавлена поддержка чисел с фиксированной запятой.

Если попытаться использовать эти типы данных, при компиляции в консоли появится сообщение об ошибке:

```
    UnimplementedFeatureError: Not yet implemented - FixedPointType.
```

Адрес

Переменные типа address хранят значения адресов Ethereum длиной 20 байт. В них можно записать 40 шестнадцатеричных символов адреса, или 160 бит.

Для переменных типа address определены следующие операции: <=, <, ==, !=, >=, >. Обратите внимание, что с переменными address нельзя выполнять арифметические операции.

Тип данных address представляет собой класс, в котором определены функции. Например, с помощью функции balance можно определить баланс аккаунта, адрес которого сохранен в переменной типа address.

В нашем тестовом контракте SolDataTypes.sol определено поле owner типа address, а также функция getBalance:

```
contract SolDataTypes {
…
    address owner;
    function getBalance() public view returns ( uint ) {
        address myAddress = msg.sender;
        uint balance = myAddress.balance;
        return balance;
    }
…
}
```

Прежде всего эта функция обращается к специальной переменной msg.sender, которая содержит адрес аккаунта, вызвавшего функцию контракта. Далее функция определяет баланс этого аккаунта в единицах Wei и возвращает баланс.

Для тестирования вы можете вызвать функцию getBalance следующим образом:

```
    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.getBalance()}).then(function(value){return
value.toString()});
    '99759787472000000000'
```

Здесь полученное значение баланса преобразуется в текстовую строку функцией value.toString и затем выводится на консоль.

Для хранения адресов, на которые могут быть переведены средства, используется тип данных address payable. Этот тип данных определен в solidity, начиная с версии 0.5.0. Помимо balance в классе address payable определены функции transfer и send.

Функция transfer позволяет передать определенное количество средств на заданный адрес. Аналогичное назначение и у функции send, однако она выполняет эту операцию на более низком уровне.

Все, что касается перечисления средств между аккаунтами, заслуживает отдельного рассмотрения, так как это очень важная тема. Если реализовать функционал передачи средств неправильно, то этим могут воспользоваться злоумышленники с целью похищения криптовалюты.

На шестом уроке мы привели примеры скриптов JavaScript, которые перечисляют средства с одного аккаунта на другой, а также рассказали об особенностях перевода средств на аккаунт смарт-контракта.

Переменные сложных типов

Переменные сложных типов, такие как массивы, хеши и структуры, не помещаются в 256 бит. Они могут занимать много места, что может привести к существенным затратам газа при выполнении контракта. Такие переменные могут храниться в оперативной памяти (тип memory) или в хранилище (storage) и передаются через параметры функций по ссылке, а не по значению.

Место для хранения переменных обычно выбирается компилятором по умолчанию, однако вы можете использовать ключевые слова storage и memory для явного указания расположения.

Параметры функций и значения, возвращаемые функциями, по умолчанию располагаются в memory, в то время как локальные переменные и переменные состояния сохраняются в постоянной памяти storage.

Содержимое оперативной памяти(и соответственно данные с расположением memory) не сохраняется в блокчейне, и это нужно учитывать при разработке контрактов.

Массивы фиксированного размера

Контракты Solidity могут работать с массивами фиксированного размера, а также с динамическими массивами. Массивы могут быть одномерными и многомерными.

В нашем контракте SolDataTypes.sol определен массив фиксированного размера boxPrice, предназначенный, например, для хранения арендных цен гаражных блоков, а также функции setBoxPrice и getBoxPrices:

```
    uint8[10] boxPrice;

    function setBoxPrice(uint idBox, uint8 price) public {
       boxPrice[idBox] = price;
    }

    function getBoxPrices() public view returns( uint8[10] memory )
{
       return boxPrice;
    }
```

Первая из этих функций записывает в массив цену блока, заданного своим номером, а вторая возвращает весь массив цен.

Сразу после публикации контракта массив цен будет пустым:

```
    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.getBoxPrices()}).then(function(value){return
value.toString()});
    '0,0,0,0,0,0,0,0,0,0'
```

Запишем в него несколько значений:

```
    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.setBoxPrice(0, 10)});
    { tx:
      '0x58f1b7206a09d8879b42eb022a05d5f617266de4e634708256a2fb32e8c570
     receipt:
      { transactionHash:
         '0x58f1b7206a09d8879b42eb022a05d5f617266de4e634708256a2fb32e8c
        transactionIndex: 0,
        blockHash:
         '0x63718856883b9fa2dcb24185ec8095b6342e55af11427772f59dcfb9d65
        blockNumber: 37,
        from: '0x532a094f3c4d2547037275a0d94041f471e756b6',
        to: '0x105f568f029eaa522529b340eff24214046d1c4d',
        gasUsed: 42283,
        cumulativeGasUsed: 42283,
        contractAddress: null,
```

```
        logs: [],
        status: true,
        logsBloom:
         '0x00000000000000000000000000000000000000000000000000000000000000000
        v: '0x1c',
        r:
         '0x59a1ea609420eaeb2e868fe32bf1acfa173f98ad960364185d267641e25
        s:
         '0x474d7b9e4bd18fd918cc69ced849e8301fbc7eaf7bf453465ac2baec8f5
        rawLogs: [] },
     logs: [] }
```

Аналогично:

```
    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.setBoxPrice(1, 20)});
    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.setBoxPrice(2, 30)});
```

Теперь при выводе на консоли показываются добавленные нами значения:

```
    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.getBoxPrices()}).then(function(value){return
value.toString()});
    '10,20,30,0,0,0,0,0,0,0'
```

Учтите, что функцию setBoxPrice может вызывать кто угодно, знающий адрес контракта, поэтому такой способ записи цен не является безопасным. В нашем простом примере мы это не учитываем.

Динамические массивы

Динамические массивы удобно использовать, когда вы не можете заранее определить размерность создаваемого массива.

В нашем контракте SolDataTypes.sol мы определили динамический массив dBoxPrice, содержащий 8-битные значения, а также функции setDBoxPrice и getDBoxPrice:

```
    uint8[] dBoxPrice;

    function setDBoxPrice(uint8 price) public {
       dBoxPrice.push(price);
    }

    function getDBoxPrices() public view returns( uint8[] memory )
{
       return dBoxPrice;
    }
```

Первая из этих функций добавляет в динамический массив новое значение с помощью функции push, а вторая возвращает весь массив.

Добавьте в динамический массив данные следующим образом:

```
    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.setDBoxPrice(10)});
    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.setDBoxPrice(20)});
```

Если теперь вызвать функцию getDBoxPrice(), то мы увидим в консоли значения, добавленные ранее в динамический массив:

```
    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.getDBoxPrices()}).then(function(value){return
value.toString()});
    '10,20'
```

Перечисление

С помощью типа данных enum (перечисление) вы можете создавать в Solidity свои собственные типы данных.

Например, в нашем контракте мы создаем тип данных doorState, который хранит состояние двери. Дверь может находиться в одном из трех состояний: Open (открыта), Closed (закрыта) и Blocked (заблокирована).

В нашем контракте мы определили также четыре функции, три из которых могут изменять состояние двери, а четвертая – возвращает текущее состояние:

```
enum doorState { Open, Closed, Blocked }
doorState myDoor;

function openDoor() public {
   myDoor = doorState.Open;
}
function closeDoor() public {
   myDoor = doorState.Closed;
}
function blockDoor() public {
   myDoor = doorState.Blocked;
}
function getDoorState() public view returns ( doorState ) {
   return myDoor;
}
```

Чтобы открыть дверь, вызовите функцию openDoor:

```
truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.openDoor ()});
```

Теперь если вызывать функцию getDoorState, то она вернет значение 0, соответствующее открытой двери:

```
truffle(develop)>
SolDataTypes.deployed().then(function(instance){return  instance.
getDoorState ()}).then(function(value){return value.toString()});
'0'
```

Здесь мы преобразовали возвращаемое значение в текстовую строку функцией toString.

Аналогично можно установить для двери другие состояния и узнать, какие значения им соответствуют:

```
truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.closeDoor ()});
```

```
    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.getDoorState            ()}).then(function(value){return
value.toString()});
    '1'

    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.blockDoor ()});

    truffle(develop)>
SolDataTypes.deployed().then(function(instance){return
instance.getDoorState            ()}).then(function(value){return
value.toString()});
    '2'
```

В качестве упражнения добавьте в контракт функцию unblockDoor, которая разблокирует дверь, а также модифицируйте функции openDoor и closeDoor таким образом, чтобы они возвращали false, если дверь заблокирована, и не изменяли состояние двери, пока она не будет разблокирована.

Структуры

Как и во многих других языках программирования, в Solidity предусмотрена возможность создавать собственные типы данных с помощью структур. В структурах могут быть определены поля различных типов данных, но не программный код.

Для демонстрации использования структур и словарей (mapping) мы подготовили контракт StructSample.sol (листинг 8.2.), а также скрипт для выполнения миграции (публикации контракта) 4_sructsample_migrations.js (листинг 8.3.).

Листинг 8.2. Файл StructSample.sol

```
pragma solidity ^0.5.0;
pragma experimental ABIEncoderV2;

contract StructSample {

   address public owner;

   struct parkingMember {
       uint id;
       string name;
       uint balance;
       bool isActive;
   }

   parkingMember parkingStruct;
   parkingMember[] public parkingMembersList;

   mapping(uint => parkingMember) public parkingMemberMap;

   constructor() payable public {
       owner = msg.sender;

       parkingStruct = parkingMember({id:1, name:"Alexandre",
balance:200, isActive:true});
       parkingMembersList.push(parkingStruct);

       parkingMemberMap[1] = parkingStruct;
   }

     function getMapMemberName(uint id) view public returns
( string memory) {
       string memory memberName = parkingMemberMap[id].name;
       return memberName;
   }

   function getMemberName(uint id) view public returns ( string
memory) {
      string memory memberName = parkingMembersList[id].name;
```

```
            return memberName;
        }

       function listMembers() view public returns ( parkingMember[]
memory) {
            return parkingMembersList;
        }

      function addMember(uint id, string memory name, uint balance)
payable public returns ( bool) {
                parkingStruct = parkingMember({id:id, name:name,
balance:balance, isActive:true});

            parkingMembersList.push(parkingStruct);
            parkingMemberMap[id] = parkingStruct;

            return(true);
        }
    }
```

Листинг 8.3. Файл 4_sructsample_migrations.js

```
var MyContract = artifacts.require("StructSample");

module.exports = function(deployer) {
 // deployment steps
 deployer.deploy(MyContract);
};
```

В контракте StructSample.sol мы определили структуру parkingMember для хранения информации о владельце парковочного места в воображаемом гараже:

```
struct parkingMember {
   uint id;
   string name;
   uint balance;
   bool isActive;
}
```

Здесь хранится некоторый числовой идентификатор владельца id, его имя name, баланс на счету balance и флаг активности аккаунта isActive. Разумеется, этого недостаточно для реальной парковки, но для начала мы ограничимся только такой информацией.

В контракте мы определили переменную parkingStruct типа parkingMember, где будут храниться данные структуры, а также массив parkingMembersList для хранения данных владельцев парковочных мест:

```
parkingMember parkingStruct;
parkingMember[] public parkingMembersList;
```

В нашем контракте мы определили конструктор, который получает управление сразу после запуска скрипта контракта:

```
    constructor() payable public {
     owner = msg.sender;

       parkingStruct  =  parkingMember({id:1,  name:"Alexandre",
balance:200, isActive:true});
     parkingMembersList.push(parkingStruct);
    }
```

Функция конструктора определяется как constructor. В предыдущих версиях Solidity конструктор должен был иметь такое же имя, что и файл, содержащий скрипт контракта, но теперь нужно использовать название constructor.

Наш конструктор записывает адрес владельца контракта в переменную owner типа address. Таким образом мы сохраняем адрес узла, который опубликовал контракт.

Кроме того, мы инициализируем структуру parkingStruct, записывая в нее данные о встроенной учетной записи владельца парковки с идентификатором 1, именем Alexandre и балансом 200 неких условных единиц.

Далее проинициализированная таким образом структура добавляется в массив учетных записей владельцев парковочных мест parkingMembersList с помощью функции push.

Таким образом, сразу после публикации нашего контракта массив владельцев мест уже будет содержать одну запись.

С помощью функции listMembers можно получить содержимое всего массива владельцев парковочных мест:

```
    function listMembers() view public returns ( parkingMember[]
memory) {
     return parkingMembersList;
    }
```

Обратите внимание, что здесь функция возвращает массив структур. В версии Solidity 0.5.x эта возможность является экспериментальной, и для ее использования мы добавили в начало исходного текста контракта следующую строку:

```
    pragma experimental ABIEncoderV2;
```

При компиляции контракта мы получим предупреждающее сообщение об использовании экспериментальной возможности:

```
    /home/book/HelloSol/contracts/StructSample.sol:2:1:  Warning:
Experimental features are turned on. Do not use experimental
features on live deployments.
    pragma experimental ABIEncoderV2;
    ^_^
```

Разумеется, вы не должны использовать подобное в рабочем контракте. Мы рассказали об этом потому, что в одной из следующих версий Solidity эта экспериментальная возможность может уже стать доступной в нормальном режиме.

А как же получить данные из массива структур владельцев парковочных мест?

Это можно сделать, например, по идентификатору, запрашивая отдельные поля соответствующей структуры. Например, в функции getMemberName мы получаем имя владельца по его идентификатору:

```
    function getMemberName(uint id) view public returns ( string
memory) {
       string memory memberName = parkingMembersList[id].name;
       return memberName;
    }
```

Чтобы добавить нового владельца в список, предназначена функция addMember:

```
    function addMember(uint id, string memory name, uint balance)
payable public returns ( bool) {
            parkingStruct   =   parkingMember({id:id,   name:name,
balance:balance, isActive:true});

       parkingMembersList.push(parkingStruct);
       return(true);
    }
```

Здесь мы инициализируем структуру данными, полученными через параметры функции, а затем добавляем эту структуру функцией push в массив структур parkingMembersList.

Для того чтобы проверить этот контракт в работе, выполните его компиляцию и публикацию:

```
    truffle(develop)> compile
    truffle(develop)> migrate –reset
```

Сначала давайте вызовем функцию listMembers, чтобы посмотреть список владельцев парковки сразу после публикации контракта:

```
    truffle(develop)>
StructSample.deployed().then(function(instance){return
instance.listMembers()});
    [ [ '1',
       'Alexandre',
       '200',
       true,
       id: '1',
       name: 'Alexandre',
       balance: '200',
       isActive: true ] ]
```

Как видите, в списке есть один владелец, добавленный в конструкторе контракта.

На следующем шаге добавим второго владельца:

```
    truffle(develop)>
StructSample.deployed().then(function(instance){return
instance.addMember(2, "Иван Иваныч", 333)});
```

У него будет идентификатор, равный 2, имя «Иван Иваныч» и начальный баланс, равный 333.

Теперь, если опять посмотреть список владельцев, получим следующее:

```
    truffle(develop)>
StructSample.deployed().then(function(instance){return
instance.listMembers()});
    [ [ '1',
       'Alexandre',
       '200',
       true,
       id: '1',
       name: 'Alexandre',
       balance: '200',
       isActive: true ],
     [ '2',
       'Иван Иваныч',
       '333',
       true,
       id: '2',
       name: 'Иван Иваныч',
       balance: '333',
       isActive: true ] ]
```

Как и следовало ожидать, теперь в списке находится два владельца, один из которых был добавлен конструктором, а второй – функцией addMember.

Чтобы проверить в работе функцию getMemberName, возвращающей имя владельца по его идентификатору, используйте такую команду:

```
    truffle(develop)>
StructSample.deployed().then(function(instance){return
instance.getMemberName(0)});
    'Alexandre'
```

Для извлечения имени второго владельца используйте такой код:

```
    truffle(develop)>
StructSample.deployed().then(function(instance){return
instance.getMemberName(1)});
    'Иван Иваныч'
```

Если в качестве параметра передать функции getMemberName несуществующий идентификатор, мы получим исключение во времени работы контракта. Здесь мы передали идентификатор 2, но массивы нумеруются, начиная с 0, поэтому получаем ошибку:

```
    truffle(develop)>
StructSample.deployed().then(function(instance){return
instance.getMemberName(2)});
    Thrown:
    Error: Returned error: VM Exception while processing
transaction: invalid opcode
              at XMLHttpRequest._onHttpResponseEnd (/usr/lib/
node_modules/truffle/build/webpack:/~/xhr2-cookies/dist/xml-
http-request.js:318:1)
         at XMLHttpRequest._setReadyState (/usr/lib/node_modules/
truffle/build/webpack:/~/xhr2-cookies/dist/xml-http-
request.js:208:1)
           at XMLHttpRequestEventTarget.dispatchEvent (/usr/lib/
node_modules/truffle/build/webpack:/~/xhr2-cookies/dist/xml-
http-request-event-target.js:34:1)
          at XMLHttpRequest.request.onreadystatechange (/usr/lib/
node_modules/truffle/build/webpack:/~/web3/~/web3-providers-
http/src/index.js:96:1)
         at /usr/lib/node_modules/truffle/build/webpack:/packages/
truffle-provider/wrapper.js:112:1
       at /usr/lib/node_modules/truffle/build/webpack:/~/web3-eth-
contract/~/web3-core-requestmanager/src/index.js:140:1
          at Object.ErrorResponse (/usr/lib/node_modules/truffle/
build/webpack:/~/web3-eth-contract/~/web3-core-helpers/src/
errors.js:29:1)
```

Словари mapping

Вы, скорее всего, знаете про такие структуры данных, как хеши и словари данных. С их помощью можно устанавливать соответствие между ключом и данными. Задавая ключ, вы можете сохранять данные в словаре, а также извлекать данные из словаря.

В языке Solidity словари создаются при помощи ключевого слова mapping:

```
mapping(uint => parkingMember) public parkingMemberMap;
```

Здесь в качестве ключа используется тип данных uint, а в качестве данных – массив структур parkingMember.

Solidity позволяет использовать в качестве ключа только встроенные типы данных, а также данные типа bytes и string. Что же касается значений, то в mapping можно сохранять данные любого типа, в частности, массивы и mappings.

Первый элемент добавляется в словарь parkingMemberMap в конструкторе:

```
    constructor() payable public {
       owner = msg.sender;

         parkingStruct = parkingMember({id:1, name:"Alexandre",
balance:200, isActive:true});
       parkingMembersList.push(parkingStruct);

       parkingMemberMap[1] = parkingStruct;
    }
```

В функции addMember мы также добавляем структуру с информацией о владельце парковочного места parkingStruct в словарь parkingMemberMap, используя в качестве ключа идентификатор владельца:

```
    function addMember(uint id, string memory name, uint balance)
payable public returns ( bool) {
          parkingStruct = parkingMember({id:id, name:name,
balance:balance, isActive:true});

      parkingMembersList.push(parkingStruct);
      parkingMemberMap[id] = parkingStruct;
      return(true);
    }
```

Сразу после публикации контракта в словаре имеется только одна запись, добавленная конструктором:

```
    truffle(develop)>
StructSample.deployed().then(function(instance){return
instance.getMapMemberName(1)});
    'Alexandre'
```

Обратите внимание, что если при вызове функции getMapMemberName указать значение ключа, для которого в словаре нет записи, исключения не произойдет:

```
    truffle(develop)>
StructSample.deployed().then(function(instance){return
instance.getMapMemberName(10)});
    ''
```

Функция просто вернет пустую строку.

Теперь добавим в словарь одну запись, как мы это делали раньше при работе с массивами:

```
    truffle(develop)>
StructSample.deployed().then(function(instance){return
instance.addMember(2, "Иван Иваныч", 333)});
```

Если теперь попробовать вызывать getMapMemberName и передать ей идентификатор, равной 2, то из словаря типа mapping будет извлечено правильное значение:

```
    truffle(develop)>
StructSample.deployed().then(function(instance){return
instance.getMapMemberName(2)});
    'Иван Иваныч'
```

Итоги урока

На восьмом уроке вы изучили основные типы данных Solidity, такие как целые числа, адреса, строки, массивы, перечисления, структуры и словари. Вы также написали смарт-контракты, использующие эти типы данных.

Хорошее владение типами данных необходимо для успешного составления смарт-контрактов.

Урок 9. Миграция контрактов в приватную сеть и в сеть Rinkeby

Цель урока: научиться публиковать контракты при помощи Truffle в приватной сети Geth, а также в тестовой сети Rinkeby.

Практические задания: подготовка узла Geth для публикации, запуск и публикация (миграция) контракта в приветную сеть Geth, а также в тестовую сеть Rinkeby. Пополнение аккаунта Rinkeby средствами.

На этом уроке мы займемся миграцией смарт-контракта из Truffle в приватную сеть, созданную с помощью Geth. Мы также создадим узел тестовой сети Rinkeby, пополним его средствами и выполним публикацию смарт-контракта.

Публикация контракта из Truffle в приватную сеть Geth

Среда разработки смарт-контрактов Truffle удобна своими средствами миграции смарт-контрактов. Подготовив несложный файл конфигурации truffle.js, вы с помощью простых команд сможете публиковать (мигрировать) смарт-контракты в различные сети, приватные, тестовые, а также в основную сеть Ethereum.

На этом уроке мы сначала займемся публикацией в приватной сети Geth, а затем в отладочной сети Rinkeby.

Подготовка узла приватной сети

Прежде всего необходимо установить на узел клиент Geth, если вы это не сделали на предыдущих уроках. Здесь мы приведем краткую инструкцию.

Выполните обновление Ubuntu и установите необходимые пакеты:

```
$ sudo apt update
$ sudo apt upgrade
$ sudo apt-get install build-essential
$ sudo apt install python
$ sudo add-apt-repository -y ppa:ethereum/ethereum
$ sudo apt-get update
$ sudo apt-get install ethereum
```

Проверьте версию geth:

```
$ geth version
Geth
Version: 1.8.24-stable
Git Commit: 4e13a09c5033b4cf073db6aeaaa7d159dcf07f30
Architecture: amd64
Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.10.4
Operating System: linux
GOPATH=
GOROOT=/usr/lib/go-1.10
```

Установите Node.js версии 10.15.3:

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/
v0.34.0/install.sh | bash
```

Закройте и откройте заново окно консоли.

```
$ nvm install 10.15.3
```

Установите фреймворк Web3 самой новой версии:

```
$ npm install web3
```

Создайте рабочий каталог node1 для размещения данных блокчейна:

```
$ mkdir node1
```

Затем подготовьте два пакетных файла с именами start_node_dev.sh и attach_node.sh.

С помощью файла start_node_dev.sh (листинг 9.1.) вы будете запускать узел приватной сети в режиме разработчика.

Листинг 9.1. start_node_dev.sh

```
geth -datadir node1 -networkid 98760 -dev -rpc -
rpcapi="db,eth,net,web3,personal,web3" console
```

Файл attach_node.sh (листинг 9.2.) позволит вам подключиться к запущенному узлу из другого окна консоли.

Листинг 9.2. attach _node_dev.sh

```
geth -datadir node1 -networkid 98760 attach ipc://home/book/
node1/geth.ipc
```

Для проверки запустите первый из этих пакетных файлов в одном окне консоли, а второй – в другом окне консоли.

Получите список аккаунтов в окне второй консоли:

```
> web3.eth.accounts
["0xcc54cb029dc5b6dc27170deb3db8c1d412871055",
"0xc7d93f3e11da0d7dd882a917bcc29da0cdbff0a6"]
```

Если вы только что создали сеть, у вас будет только один аккаунт.

Проверьте версию фреймворка Web3 в консоли Node.js:

```
$ node
> var Web3 = require('web3')
undefined
> var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:8545"));
undefined
> web3.version
'1.0.0-beta.48'
```

Установите Truffle самой новой версии:

```
$ npm install -g truffle
```

Проверьте версию Truffle, должна быть не ниже 5.0.7:

```
$ truffle version
Truffle v5.0.7 (core: 5.0.7)
```

```
Solidity v0.5.0 (solc-js)
Node v10.15.3
```

Также установите модуль truffle-contract:

```
$ npm install truffle-contract
```

Подготовка контракта для работы

На этом уроке мы будем использовать все тот же контракт HelloSol.
Создайте файлы проекта:

```
$ mkdir HelloSol
$ cd HelloSol
$ truffle init
```

Добавьте в каталог contracts файл HelloSol.sol (листинг 9.3.).

Листинг 9.3. Файл HelloSol.sol

```
pragma solidity ^0.5.0;

contract HelloSol {
   string savedString;
   uint savedValue;

   function setString( string memory newString ) public {
       savedString = newString;
   }
   function getString() public view returns( string memory ) {
       return savedString;
   }
   function setValue( uint newValue ) public {
       savedValue = newValue;
   }
   function getValue() public view returns( uint ) {
       return savedValue;
   }
}
```

В каталог migrations добавьте файл 2_hellosol_migrations.js (листинг 9.4.).

Листинг 9.4. Файл 2_hellosol_migrations.js

```
var MyContract = artifacts.require("HelloSol");

module.exports = function(deployer) {
 // deployment steps
 deployer.deploy(MyContract);
};
```

И наконец, в каталоге проекта создайте файл truffle.js (листинг 9.5.).

Листинг 9.5. Файл truffle.js

```
module.exports = {
   networks: {
   development: {
     host: "127.0.0.1",
     port: 8545,
     network_id: "*"
   },
   live: {
     host: "127.0.0.1",
     port: 8545,
     network_id: 98760,
     gas: 5000000,
   }
 }
};
```

Обратите внимание, что в файл truffle.js мы добавили блок live, содержащий параметры доступа к нашей приватной сети.

Компиляция и миграция контракта в сеть Truffle

Запустите в отдельном консольном окне консоль Truffle Develop:

```
$ truffle develop
Truffle Develop started at http://127.0.0.1:9545/
```

Теперь у нас на хосте работают две отладочные сети. Первая из них была запущена пакетным файлом start_node_dev.sh и работает на порту 8545, вторая – запущена Truffle и работает на порту 9545.

Запустим компиляцию и миграцию контракта в сеть Truffle:

```
truffle(develop)> compile
truffle(develop)> migrate
```

Проверим, что можно вызывать функции контракта.

Сначала вызываем функцию setValue:

```
truffle(develop)> HelloSol.deployed().then(function(instance)
{return instance.setValue(777)});
{ tx:
  '0x0a8aecaa612eec53f151a903b43bb2d5b1a3b52d4363dbc859756c0f720e15
 receipt:
  { transactionHash:
     '0x0a8aecaa612eec53f151a903b43bb2d5b1a3b52d4363dbc859756c0f720
    transactionIndex: 0,
    blockHash:
```

```
        '0xd6c111efe0b9589e2d15c0dbf0fce9a4dd3085a2b750efd7150c44541d9
      blockNumber: 5,
      from: '0xf5ac3b9b4d78a6ec5b0afe13000c9b0679cab7de',
      to: '0x065daed48e78238bf010c65e2ba66471f4e8d6b7',
      gasUsed: 41803,
      cumulativeGasUsed: 41803,
      contractAddress: null,
      logs: [],
      status: true,
      logsBloom:
       '0x00000000000000000000000000000000000000000000000000000000000000
      v: '0x1c',
      r:
       '0x9b715d5f93dd7bfda1c16ad158070421dd52a89bd325593b0e88ffaa370
      s:
       '0x4fea80a3c369c1431f4d3a47f19a05c95f2d1325298b999562a48194166
      rawLogs: [] },
   logs: [] }
```

Затем вызываем функцию getValue, чтобы проверить записанное значение:

```
    truffle(develop)> HelloSol.deployed().then(function(instance)
{return         instance.getValue()}).then(function(value){return
value.toNumber()});
    777
```

Запуск миграции в локальную сеть geth

Для того чтобы опубликовать контракт в вашей локальной сети geth, достаточно выполнить миграцию из Truffle следующей командой:

```
    truffle(develop)> migrate –network live
```

И это все, контракт будет опубликован в соответствии с секцией live файлf truffle.js (листинг 9.5.). Дальше вы можете вызывать функции этого контракта, как мы это делали на седьмом уроке.

А что если доступ к контракту нужно организовать отдельно, вне Truffle?

Это тоже возможно, но требуется определить адрес контракта, опубликованного при помощи Truffle. Этот адрес можно взять из артефактов проекта Truffle.

Добываем артефакты Truffle

Результаты миграции контракта, сам контракт, его интерфейс abi и многое другое Truffle записывает в файл build/contracts/<имя контракта>.json. В нашем случае это будет файл /home/book/HelloSol/build/contracts/HelloSol.json.

Откройте этот файл в текстовом редакторе, таком как Sublime или Vim, и вы увидите там много интересного.

В частности, там есть адреса контрактов для сетей с различными идентификаторами:

```
    "networks": {
       "5777": {
         "events": {},
         "links": {},
         "address": "0x065dAED48E78238bf010c65E2BA66471f4e8d6b7",
                                                  "transactionHash":
"0xd8e2bee79abf001d73bddc8d8af58a2a5621f28c7146d86a78a525af6ec99824"
       },
       "98760": {
         "events": {},
         "links": {},
         "address": "0xa65B88734fbe5942f970cB5814ae976d5ce5f00d",
                                                  "transactionHash":
"0xc46a87e0b00512069f765aa5904ba856066db2e200382a258900e87069a5b191"
       }
```

Нас интересует сеть с идентификатором 98760, который мы задали в пакетном файле запуска узла start_node_dev.sh:

```
    geth -datadir node1 -networkid 98760 -dev -rpc -
rpcapi="db,eth,net,web3,personal,web3" console
```

В листинге 9.6. мы привели скрипт call_contract_1.0.js, который «добывает» адрес контракта из файла артефакта HelloSol.json и использует его для вызова функций контракта.

Листинг 9.6. Файл call_contract_1.0.js

```
    // node call_contract_1.0.js HelloSol 98760

    var contract_name = process.argv[2];
    var network_id = process.argv[3];
    var unlock_password = process.argv[4];

    console.log('Contract script: ' + contract_name);

    var path = require('path');
    var contractJSON = require(path.join(__dirname, 'build/
contracts/' + contract_name + '.json'));

    var decoded = JSON.parse(JSON.stringify(contractJSON.networks,
undefined, 2));
    var contract_address = decoded[network_id].address;
    var abi;

    console.log('contract_address: ' + contract_address);

    var Web3 = require('web3');
    var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:8545"));
```

```
    var myCoinbase;
    web3.eth.getCoinbase()
    .then(coinbase => {
     myCoinbase = coinbase;
     return coinbase;
    })
    .then(function (account) {
           return       web3.eth.personal.unlockAccount(account,
unlock_password, 600)
    })
    .then(function (unlock_result) {
     console.log(`${myCoinbase} unlock result: ` + unlock_result);

     abi = contractJSON.abi;
     var myContract = new web3.eth.Contract(abi, contract_address);

     myContract.methods.getValue().call({from: contract_address},
(error, result) =>
     {
        if(!error){
          console.log('getValue result: ' + result);
          console.log(JSON.stringify(result, undefined, 2));
        } else{
          console.log(error);
        }
     });

                        myContract.methods.getString().call({from:
contract_address}, (error, result) =>
     {
        if(!error){
          console.log('getString result: ' + result);
        } else{
          console.log(error);
        }
     });

               myContract.methods.setValue(555307309).send({from:
myCoinbase})
     .once('setValue transactionHash', (hash) => {
       console.log('hash: ' + hash);
     })
     .on('setValue confirmation', (confNumber) => {
       console.log('confNumber: ' + confNumber);
     })
     .on('receipt', (receipt) => {
       console.log(JSON.stringify(receipt, undefined, 2));
     })
     .then(function () {
```

```
              myContract.methods.setString("Test           string
555307309").send({from: myCoinbase})
     .once('transactionHash', (hash) => {
       console.log('setString hash: ' + hash);
     })
     .on('confirmation', (confNumber) => {
       console.log('setString confNumber: ' + confNumber);
     })
     .on('receipt', (receipt) => {
       console.log(JSON.stringify(receipt, undefined, 2));
     })
     });
    });
```

В качестве параметров при запуске нашему скрипту необходимо передать имя контракта, идентификатор сети и пароль для разблокировки аккаунта:

```
$ node call_contract_1.0.js HelloSol 98760 ""
```

Идентификатор сети нужно указывать, т.к. в файле артефактов находится информация для разных сетей.

В случае запуска узла geth в режиме разработчика с параметром –dev пароль разблокировки аккаунта нужно указывать пустым. Если же вы публикуете контракт в обычную сеть geth, потребуется правильный пароль для разблокировки исходного аккаунта.

Адрес опубликованного контракта мы получаем следующим образом:

```
var path = require('path');
var  contractJSON  =  require(path.join(__dirname,  'build/
contracts/' + contract_name + '.json'));

var decoded = JSON.parse(JSON.stringify(contractJSON.networks,
undefined, 2));
var contract_address = decoded[network_id].address;
```

Здесь мы вначале загружаем файл артефактов, указывая его полный путь. Далее мы раскодируем раздел сетей networks, а затем извлекаем адрес сети с нужным нам идентификатором.

Для вызова функций контракта мы используем приемы с промисами, которые работают с фреймворком Web3 версии 1.0.x (проверено на версии 1.0.47).

Вначале мы получаем доступ к Web3, указывая порт 8584:

```
var Web3 = require('web3');
var web3 = new Web3(new Web3.providers.HttpProvider("http://
localhost:8545"));
```

Далее мы определяем адрес локального узла и разблокируем аккаунт:

```
var myCoinbase;
web3.eth.getCoinbase()
```

```
    .then(coinbase => {
     myCoinbase = coinbase;
     return coinbase;
    })
    .then(function (account) {
           return       web3.eth.personal.unlockAccount(account,
unlock_password, 600)
    })
```

После разблокировки аккаунта создаем объект контракта, извлекая abi из загруженного файла артефактов:

```
    abi = contractJSON.abi;
    var myContract = new web3.eth.Contract(abi, contract_address);
```

Пользуясь объектом контракта, вызываем функции getValue и getString нашего контракта, опубликованного в приватной сети:

```
    myContract.methods.getValue().call({from:  contract_address},
(error, result) =>
     {
        if(!error){
          console.log('getValue result: ' + result);
          console.log(JSON.stringify(result, undefined, 2));
        } else{
          console.log(error);
        }
     });

                        myContract.methods.getString().call({from:
contract_address}, (error, result) =>
     {
        if(!error){
          console.log('getString result: ' + result);
        } else{
          console.log(error);
        }
     });
```

Аналогично вызываем функции setValue и setString:

```
                myContract.methods.setValue(555307309).send({from:
myCoinbase})
     .once('setValue transactionHash', (hash) => {
       console.log('hash: ' + hash);
     })
     .on('setValue confirmation', (confNumber) => {
       console.log('confNumber: ' + confNumber);
     })
     .on('receipt', (receipt) => {
```

```
      console.log(JSON.stringify(receipt, undefined, 2));
    })
    .then(function () {

            myContract.methods.setString("Test          string
555307309").send({from: myCoinbase})
    .once('transactionHash', (hash) => {
      console.log('setString hash: ' + hash);
    })
    .on('confirmation', (confNumber) => {
      console.log('setString confNumber: ' + confNumber);
    })
    .on('receipt', (receipt) => {
      console.log(JSON.stringify(receipt, undefined, 2));
    })
    });
  });
```

Первая из этих функций не изменяет состояние сети, и ее вызов не приводит к возникновению транзакции. Вторая функция вызывает транзакцию.

Публикация контракта из Truffle в тестовой сети Rinkeby

Прежде чем вы приступите к публикации контрактов в основной сети Ethereum, мы рекомендуем вам выполнить тестирование в сети Rinkeby. Она очень похожа на основную сеть, но работа в Rinkeby не вызовет у вас реальных затрат.

Подготовка узла Geth для работы с Rinkeby

Прежде всего скопируйте файл rinkeby.json в домашний каталог пользователя book:

```
$ wget https://www.rinkeby.io/rinkeby.json
```

Далее проинициализируйте узел Rinkeby:

```
$ geth –datadir=$HOME/.rinkeby init rinkeby.json
```

Запустите узел следующей командой:

```
$ geth –networkid=4 –datadir=$HOME/.rinkeby –cache=2048
--rpc –rpcaddr "0.0.0.0" –rpcapi
"admin,debug,miner,shh,txpool,personal,eth,net,web3" –
ethstats='shop2you:Respect my authoritah!@stats.rinkeby.io' –
bootnodes=enode://
a24ac7c5484ef4ed0c5eb2d36620ba4e4aa13b8c84684e1b4aab0cebea2ae45cb4d375b
```

Описание параметров команды geth вы можете найти по адресу https://github.com/ethereum/go-ethereum/wiki/Command-Line-Options. Также можно посмотреть справку по параметрам geth с помощью команды:

```
geth –help
```

Параметр –networkid=4 указывает, что мы используем тестовую сеть Rinkeby.

С помощью параметра –datadir мы задаем расположение каталога для хранения базы данных узла и ключей.

Параметр –rpc разрешает работу сервера HTTP-RPC. Мы будем подключаться к нашему узлу из второго консольного окна через этот сервер.

С помощью параметра –rpcaddr мы указываем, что сервер работает на всех сетевых интерфейсах узла (по умолчанию только на localhost).

При помощи параметра –rpcapi мы указываем, какие API будут доступны через интерфейс HTTP-RPC.

Параметр –ethstats указывает URL-адрес отчета службы ethstats и задается в формате nodename:secret@host:port.

И наконец, параметр –bootnodes задает URL-адреса для загрузки P2P (Comma separated enode URLs for P2P discovery bootstrap).

Итак, мы запустили узел Geth для сети Rinkeby в первом консольном окне. Во **втором** консольном окне подключитесь к этому узлу с помощью такой команды:

```
    $ geth -networkid=4 -datadir=/home/book/.rinkeby attach ipc://
home/book/.rinkeby/geth.ipc
```

После подключения вы увидите командное приглашение утилиты geth. Введите в нем следующую команду:

```
    > web3.version
    {
     api: "0.20.1",
     ethereum: "0x3f",
     network: "4",
     node: "Geth/v1.7.3-stable-4bb3c89d/linux-amd64/go1.9.1",
     whisper: undefined,
     getEthereum: function(callback),
     getNetwork: function(callback),
     getNode: function(callback),
     getWhisper: function(callback)
    }
```

Эта команда покажет версию программного интерфейса Web3, с помощью которого мы будем работать с контрактами, версию АПИ Geth, а также номер сети, который для Rinkeby равен 4.

Синхронизация узла

Сразу после запуска узла начнется его синхронизация с другими узлами сети.

Процесс синхронизации можно наблюдать в первом консольном окне, в котором мы запустили узел Geth:

```
    …
    INFO     [03-12|13:19:51.129]     Imported     new     block
receipts              count=274  elapsed=174.167ms number=2610338
hash=b70f02…a14fd5 age=8mo5d1h    size=3.94mB
    INFO     [03-12|13:19:51.420]     Imported     new     block
receipts              count=407  elapsed=284.216ms number=2610745
hash=2a4c1a…e59724 age=8mo4d23h   size=6.82mB
    INFO     [03-12|13:19:51.502]     Imported     new     block
receipts              count=132  elapsed=79.944ms  number=2610877
hash=74bc59…38cbc6 age=8mo4d22h   size=2.00mB
    INFO     [03-12|13:20:06.297]     Imported     new     block
headers               count=2048 elapsed=16.929s   number=2613354
hash=5c7e66…774942 age=8mo4d12h
    INFO     [03-12|13:20:06.882]     Imported     new     state
entries             count=608  elapsed=3.708ms   processed=3967595
pending=17868 retry=0   duplicate=0 unexpected=764
    INFO     [03-12|13:20:07.060]     Imported     new     block
receipts              count=144  elapsed=113.803ms number=2611450
hash=003251…297e29 age=8mo4d20h   size=1.32mB
```

```
    INFO     [03-12|13:20:07.340]     Imported     new     block
receipts              count=411  elapsed=190.825ms number=2611861
hash=ded385…cf615a age=8mo4d18h   size=4.42mB
    INFO     [03-12|13:20:08.423]     Imported     new     block
receipts              count=1493 elapsed=630.276ms number=2613354
hash=5c7e66…774942 age=8mo4d12h   size=15.43mB
    INFO     [03-12|13:20:12.945]     Imported     new     state
entries             count=701  elapsed=5.364ms   processed=3968296
pending=18176 retry=0   duplicate=0 unexpected=764
    …
```

Синхронизация начинается автоматически, но не сразу после запуска узла geth, а через некоторое время. На ее завершение может уйти достаточно продолжительное время, порядка нескольких часов, а то и дней.

Чтобы синхронизация шла быстрее, рекомендуется использовать диски SSD, быстрый процессор и оперативную память объемом не меньше 4 Гбайт. Для подключения к интернету лучше использовать канал шириной не менее 100 Мбит/с.

В процессе синхронизации geth будет периодически упаковывать базу данных, чтобы она занимала меньше места на диске. При этом на консоли будут появляться такие сообщения:

```
    WARN   [03-12|14:23:02.918]   Database   compacting,   degraded
performance database=/home/book/.rinkeby/geth/chaindata
```

На время упаковки синхронизация будет замедляться.

Прежде чем продолжить работу, необходимо дождаться окончания синхронизации узла. Запустите во втором окне в приглашении geth команду eth.syncing:

```
> eth.syncing
{
 currentBlock: 148226,
 highestBlock: 4178118,
 knownStates: 300686,
 pulledStates: 284817,
 startingBlock: 0
}
```

Повторяя время от времени запуск команды, сможете отслеживать ход процесса. Когда синхронизация будет завершена, команда вернет false:

```
> eth.syncing
false
```

Пока идет синхронизация, команда eth.syncing возвращает объект с полями, отражающими ход синхронизации:

- currentBlock – номер блока, который уже синхронизирован узлом;
- highestBlock – результат оценки номера блока, до которого нужно синхронизировать узел;
- knownStates – количество объектов (состояний) префиксного дерева (state trie entries), о которых известно узлу;

• pulledStates – количество загруженных узлов префиксного дерева;
• startingBlock – номер блока, с которого началась синхронизация.

Процесс синхронизации будет завершен после загрузки всех состояний префиксного дерева.

Так как в сети постоянно появляются новые блоки и новые состояния, узел должен продолжать выполнение синхронизации.

Номер самого последнего обработанного блока можно определить при помощи следующей команды:

```
> eth.blockNumber
4027994
```

Пока начальная синхронизация узла не завершится, эта команда будет возвращать нулевое значение.

Добавление аккаунтов

Сразу после установки нового узла вам нужно добавить в него аккаунты. Для добавления аккаунта введите в приглашении geth команду personal.newAccount():

```
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0x37ff26a52677e4dd1bab543ddab5905a87f49162"
```

Здесь вам нужно будет ввести пароль. Храните пароль в надежном месте, чтобы его не скопировали вирусы или троянские программы.

Пароль защищает ваш приватный ключ. Если злоумышленник сможет похитить пароль и приватный ключ с компьютера, он завладеет всеми средствами, которые есть у данного аккаунта. Не существует никакого способа восстановить пароль, так что терять его нельзя.

После ввода пароля на консоли появится адрес аккаунта, в нашем случае это «0x704f35da18df4404937747b1c98201cb9d63c706» (у вас будет другой адрес). Заметим, что адреса созданных аккаунтов записывать не обязательно, т.к. их всегда можно посмотреть при помощи команды eth.accounts:

```
> eth.accounts
["0x37ff26a52677e4dd1bab543ddab5905a87f49162",
"0x704f35da18df4404937747b1c98201cb9d63c706"]
```

Вы можете добавить подобным образом несколько аккаунтов. В этом случае команда выведет массив идентификаторов для всех аккаунтов.

Для каждого аккаунта создается приватный ключ. В нашем случае все ключи находятся в каталоге /home/book/.rinkeby/keystore. Скопируйте и сохраните эти ключи в безопасном месте, особенно если они относятся к аккаунтам основной, а не тестовой сети.

Пополнение аккаунта Rinkeby эфиром

Для того чтобы можно было публиковать контракты и вызывать их методы, необходимо пополнить кошелек эфиром (Ether). Текущий баланс можно проверить следующей командой:

```
> web3.fromWei( eth.getBalance(eth.coinbase) )
0
```

Этой команде также можно предавать идентификатор аккаунта:

```
>
web3.fromWei( eth.getBalance("0x37ff26a52677e4dd1bab543ddab5905a87f4916
0
```

Чтобы пополнить кошелек в тестовой сети Rinkeby, вам не потребуются реальные бумажные деньги или криптовалюта. Воспользуйтесь сайтом https://faucet.rinkeby.io/. Вам также потребуется аккаунт в Google+, Facebook или Twitter.

Сделайте публикацию в одной из перечисленных социальных сетей, содержащую адрес вашего аккаунта, такой как 0x37ff26a52677e4dd1bab543ddab5905a87f49162 (у вас будет другой адрес).

Далее скопируйте адрес публикации в поле **Social network URL containing your Ethereum address** и выберите из списка **Give me Ether** одно из значений.

Здесь вы можете получать 3 Ethers каждые 8 часов, 7.5 Ethers каждый день или 18.75 Ethers каждые три дня. Для начала работы вам вполне хватит 3 Ethers, так что можете выбирать первый вариант.

Если вы все сделали правильно, через некоторое время средства поступят на ваш аккаунт. Разумеется, эти средства вы сможете потратить только в тестовой сети Rinkeby.

Убедиться в поступлении средств можно при помощи следующей команды:

```
> web3.fromWei( eth.getBalance(eth.coinbase) )
3
```

Учтите, что данная команда вернет реальное значение баланса только после завершения начальной синхронизации узла. Пока идет синхронизация, она будет возвращать нулевое значение.

Теперь, когда на нашем аккаунте имеются средства, мы можем продолжить работу.

Запуск миграции контракта в сеть Rinkeby

Добавьте блок rinkeby в файл truffle.js, как это показано в листинге 9.7.

Листинг 9.7. Файл truffle.js с блоком для Rinkeby

```
module.exports = {
   networks: {
```

```
      development: {
        host: "127.0.0.1",
        port: 8545,
        network_id: "*" // Match any network id
      },
      live: {
       host: "127.0.0.1", // Random IP for example purposes (do
not use)
      port: 8545,
      network_id: 98760,        // Ethereum public network
      gas: 5000000,
      },
      rinkeby: {
        host: "localhost", // Connect to geth on the specified
        port: 8545,
        from: "0x37ff26a52677e4dd1bab543ddab5905a87f49162",
        network_id: 4,
        gas: 4612388 // Gas limit used for deploys
      }
    }
   };
```

Перед запуском миграции необходимо разблокировать аккаунт. Это можно сделать в консоли geth, которую вы использовали для контроля завершения синхронизации. Введите там такую команду, указав пароль, заданный при создании аккаунта:

```
    >     personal.unlockAccount(web3.eth.coinbase,   "password",
15000)
```

Затем запустите Truffle и из консоли разработчика выдайте команду миграции с параметром –network rinkeby:

```
truffle(develop)> migrate –network rinkeby
```

Вы увидите результат миграции:

```
Compiling your contracts…

> Everything is up to date, there is nothing to compile.

Migrations dry-run (simulation)
=
> Network name:    'rinkeby-fork'
> Network id:      4
> Block gas limit: 7000000

1_initial_migration.js
=
```

```
      Deploying 'Migrations'
      -
                                                                    >
account:             0x37Ff26a52677e4dD1BaB543dDAb5905a87F49162
      > balance:             2.999460184
      > gas used:            269908
      > gas price:           2 gwei
      > value sent:          0 ETH
      > total cost:          0.000539816 ETH

      -
      > Total cost:         0.000539816 ETH

    2_hellosol_migrations.js

      Deploying 'HelloSol'
      -
                                                                    >
account:             0x37Ff26a52677e4dD1BaB543dDAb5905a87F49162
      > balance:             2.998651462
      > gas used:            377327
      > gas price:           2 gwei
      > value sent:          0 ETH
      > total cost:          0.000754654 ETH

      -
      > Total cost:         0.000754654 ETH

    Summary
    =
    > Total deployments:   2
    > Final cost:          0.00129447 ETH

    Starting migrations…
    =
    > Network name:    'rinkeby'
    > Network id:      4
    > Block gas limit: 7000000

    1_initial_migration.js
    =

      Deploying 'Migrations'
      -
                                        >                transaction
hash:    0x2549079f8fc78d9f777c64cff1565363714ea44ea3a31d1244dce46ceb73
      > Blocks: 1            Seconds: 16
```

```
                                          >                  contract
address:    0xE20DB07dE2145d63Bfc6c9d97d63B3b4aC3F7894
                                                                    >
account:             0x37Ff26a52677e4dD1BaB543dDAb5905a87F49162
      > balance:             2.99430184
      > gas used:            284908
      > gas price:           20 gwei
      > value sent:          0 ETH
      > total cost:          0.00569816 ETH

      > Saving migration to chain.
      > Saving artifacts
      -
      > Total cost:          0.00569816 ETH

    2_hellosol_migrations.js

      Deploying 'HelloSol'
      -
                                        >                  transaction
hash:    0xe0a7e8a3e65a553bd530f4c1d9b812763e0cbbf3f1e66c06690036518693

      > Blocks: 0            Seconds: 12
                                        >                  contract
address:    0x456003f63e7632bf6bf40D8a5220ebE90F8fF0e5

                                                                    >
account:             0x37Ff26a52677e4dD1BaB543dDAb5905a87F49162
      > balance:             2.98591462
      > gas used:            377327
      > gas price:           20 gwei
      > value sent:          0 ETH
      > total cost:          0.00754654 ETH

      > Saving migration to chain.
      > Saving artifacts
      -
      > Total cost:          0.00754654 ETH

    Summary
    =
    > Total deployments:   2
    > Final cost:          0.0132447 ETH
```

Обратите внимание, что на консоли появится адрес контракта 0x456003f63e7632bf6bf40D8a5220ebE90F8fF0e5 и хеш транзакции 0xe0a7e8a3e65a553bd530f4c1d9b812763e0cbbf3f1e66c066900365186932826. Далее мы будем использовать эти значения.

Просмотр информации о контракте в сети Rinkeby

Для работы с тестовой сетью Rinkeby используйте сайт Rinkeby: Ethereum Testnet, расположенный по адресу https://www.rinkeby.io/ (рис. 9.1.).

Рис. 9.1. Сайт Rinkeby: Ethereum Tetnet

Здесь вы найдете много информации о контрактах, транзакциях, сможете узнать, как пополнить свой тестовый аккаунт средствами, а также найдете инструкцию по запуску узла с помощью приложения geth.

Откройте этот сайт и щелкните в левой панели значок **Block Explorer**. Затем введите в строке поиска (находится справа вверху) адрес опубликованного контракта и щелкните кнопку **GO**. Мы ввели адрес 0x456003f63e7632bf6bf40D8a5220ebE90F8fF0e5, у вас будет другой.

Если адрес введен правильно, то вы увидите информацию о контракте (рис. 9.2.).

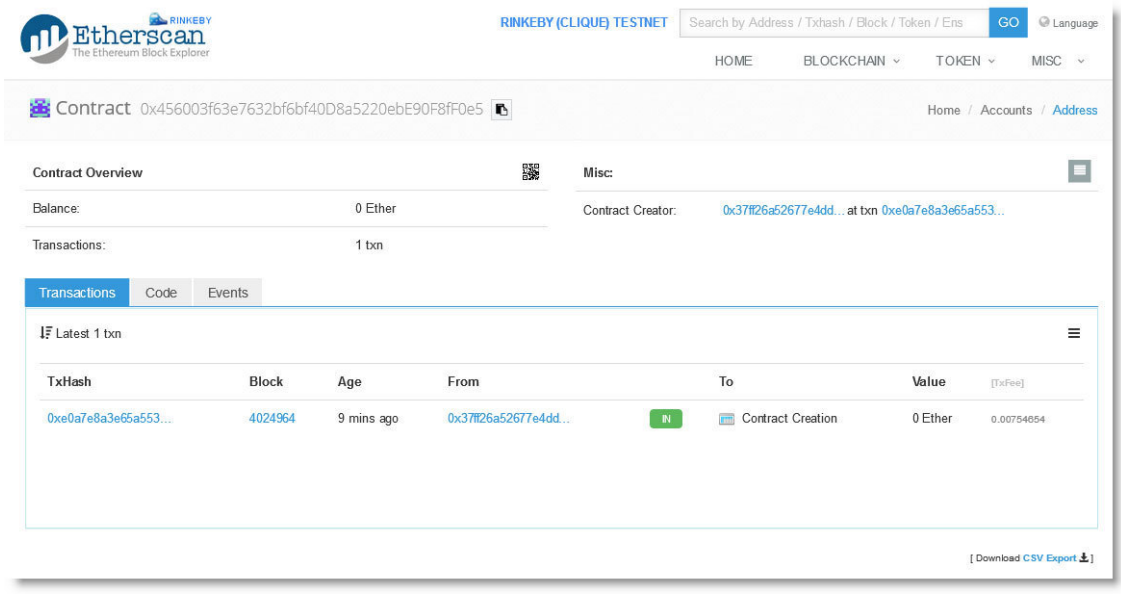

Рис. 9.2. Информация о только что опубликованном контракте

В верхней части страницы будет показан адрес контракта. Вы увидите, что баланс контракта равен нулю (поле **Balance** в блоке **Contract Overview**). Это правильно, т.к. мы еще не перечисляли средства на адрес нашего контракта.

В поле Contract Creator виден адрес узла, опубликовавшего контракт, а также соответствующий хеш транзакции. Сравните их со значениями, полученными в результате выполнения команды миграции migrate –network rinkeby.

Адрес и хеш оформлены в виде ссылок. Если их щелкнуть, вы перейдете на страницы с информацией об адресе узла и транзакции соответственно.

На вкладке **Transactions** находится строка информации о контракте, в которой есть несколько столбцов:

- хеш транзакции **TxHash**;
- номер блока **Block**, содержащего транзакцию;
- возраст транзакции **Age** (время, которое прошло с момента выполнения транзакции);
- адрес узла **From**, инициировавшего транзакцию;
- тип транзакции (в нашем случае это создание контракта: Contract Creation);
- баланс контракта **Value**;
- средства, потраченные на выполнение транзакции **TxFee.**

Если щелкнуть хеш транзакции в столбце TxHash, можно получить подробную информацию о выполненной транзакции (рис. 9.3.).

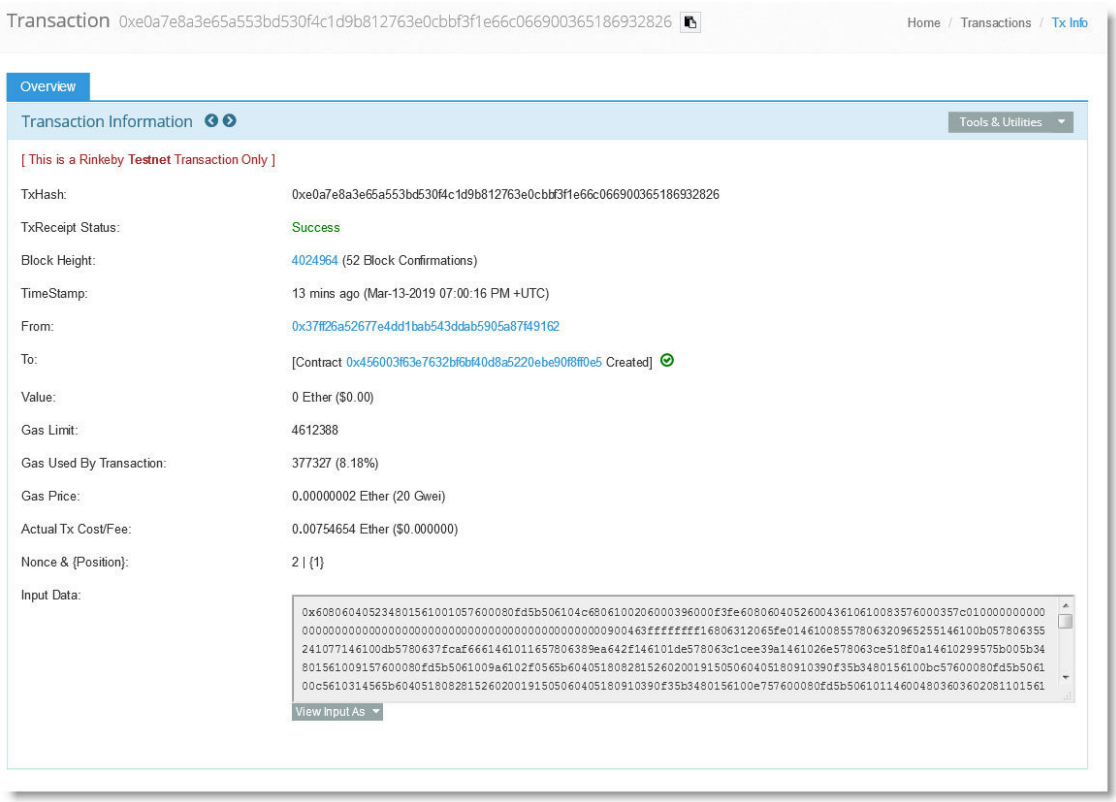

Рис. 9.3. Информация о транзакции

В частности, в поле **Gas Used By Transaction** здесь можно увидеть количество газа, использованного транзакцией, стоимость газа **Gas Price**, а также актуальную стоимость выполнения транзакции Actual Tx Cost/Fee.

Щелкнув в информации о контракте адреса узла, опубликовавшего контракт, можно увидеть подробную информацию о соответствующем аккаунте и список выполненных транзакций (рис. 9.4.).

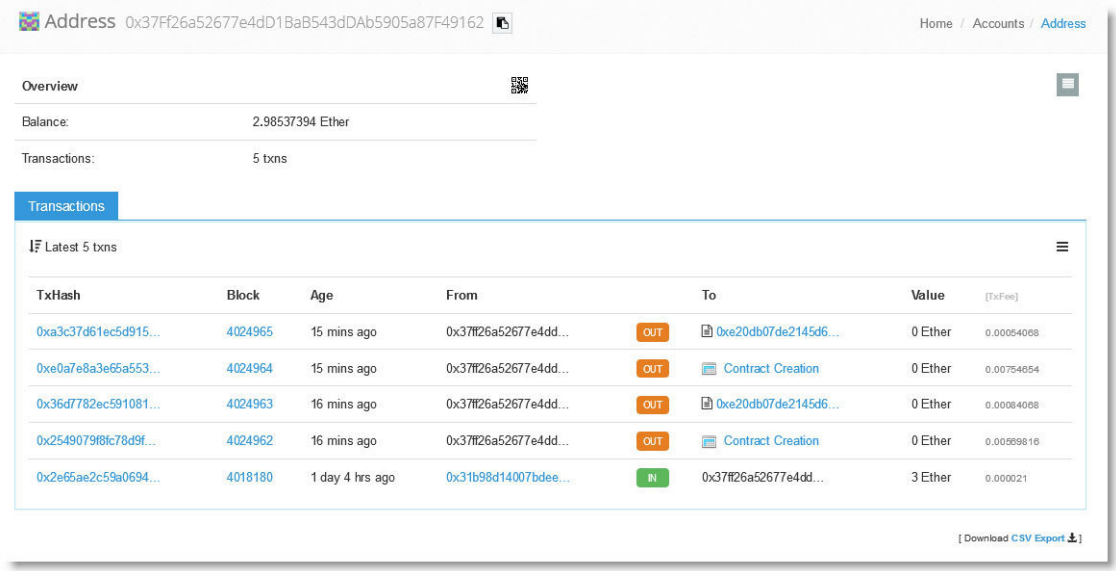

Рис. 9.4. Информация об аккаунте

Также можно щелкнуть номер блока и просмотреть подробную информацию об этом блоке (рис. 9.5.).

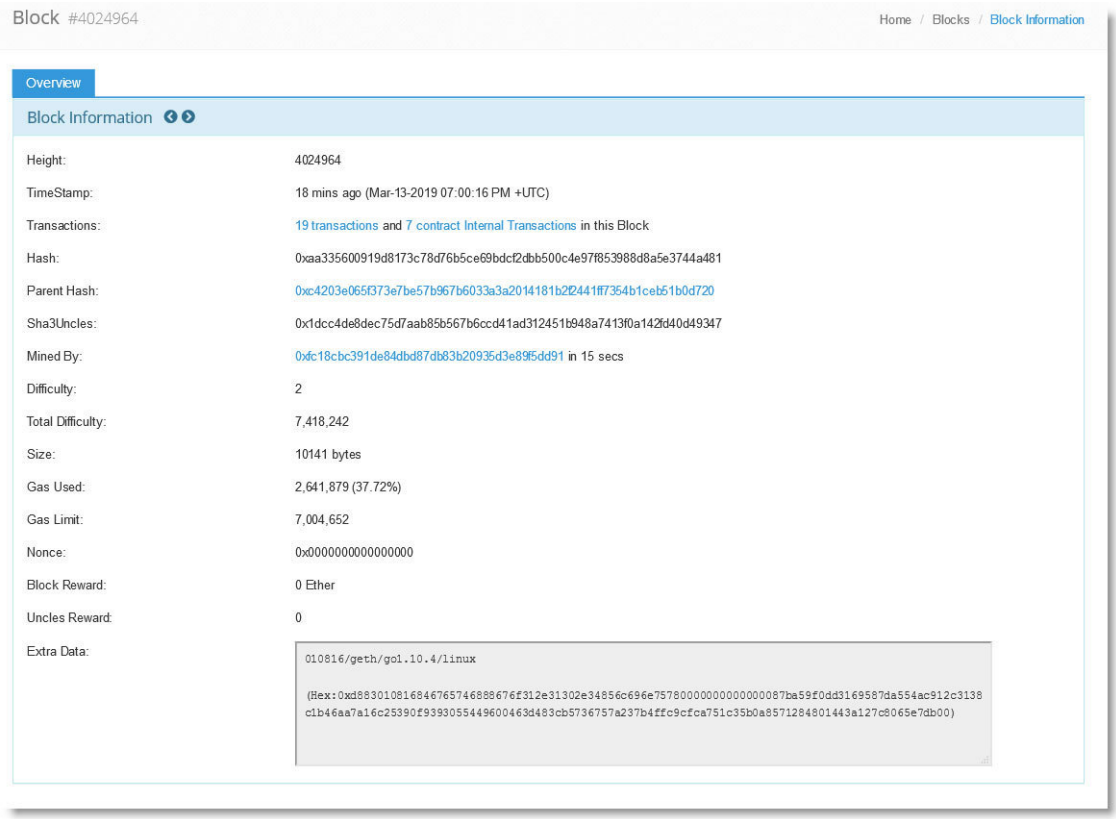

Рис. 9.5. Информация о блоке

Как видно из рис. 9.5., блок содержит данные 19 транзакций и семи внутренних транзакций контрактов. Щелкнув соответствующие ссылки, вы можете увидеть данные этих транзакций.

На вкладке Code можно просмотреть код контракта (рис. 9.6.).

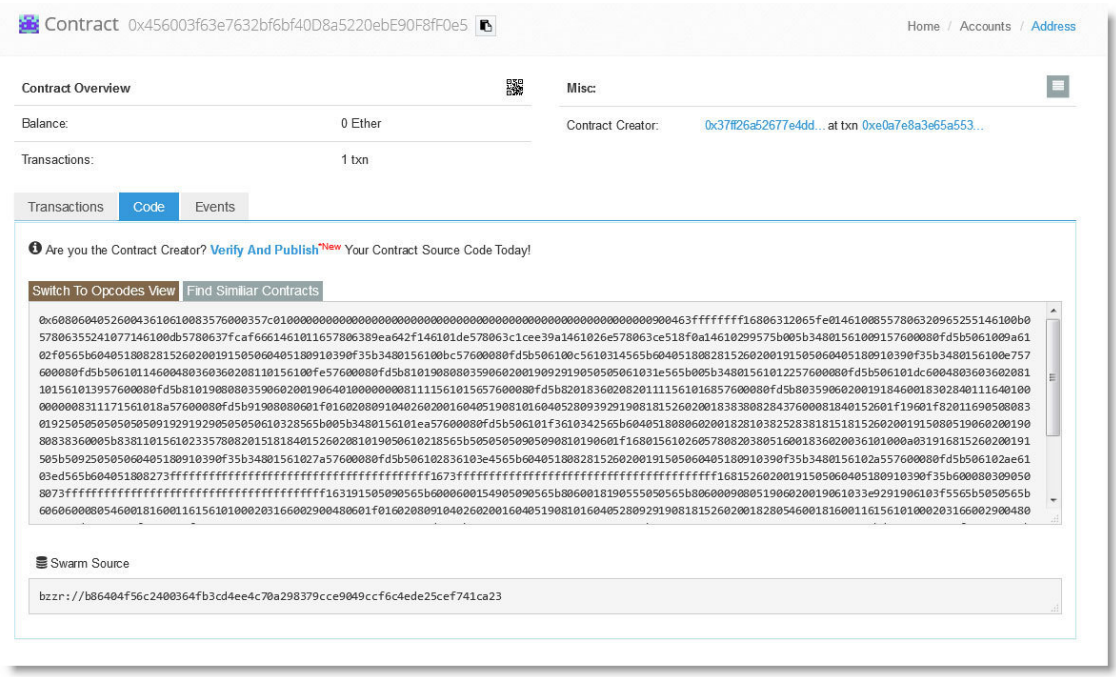

Рис. 9.6. Просмотр кода контракта

Если щелкнуть вкладку Switch To Opcodes View, то можно просмотреть коды операций (operation code) контракта (рис. 9.7.).

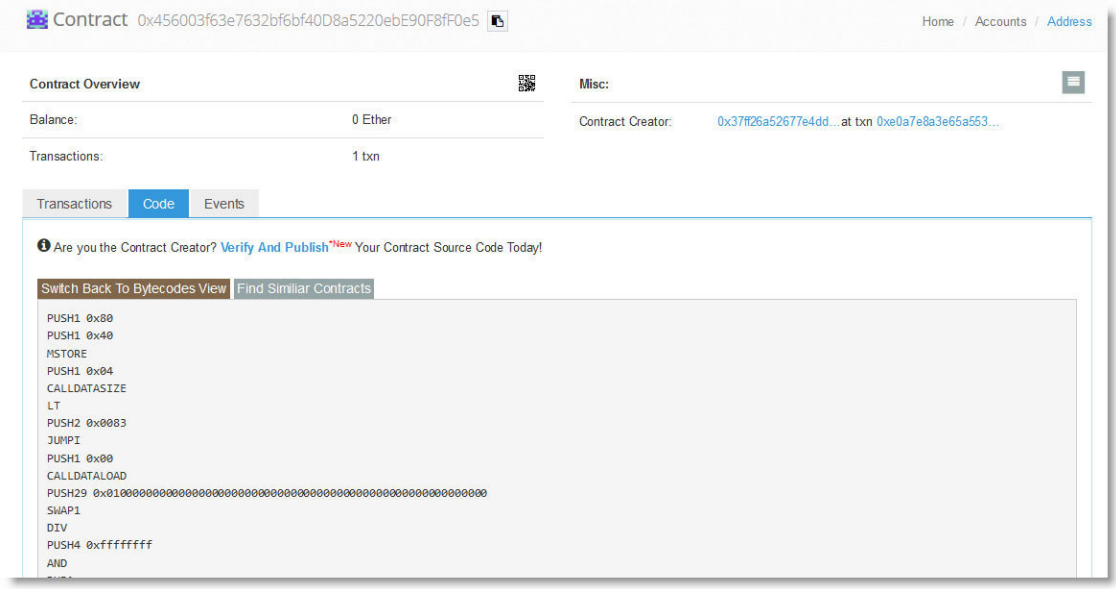

Рис. 9.7. Просмотр кода операций контракта

Как видите, любой может просмотреть код любого контракта.

Консоль Truffle для сети Rinkeby

После того как мы выполнили миграцию контракта в сеть Rinkeby, можно вызывать его методы прямо из консоли Truffle. Для этого запустите консоль с параметром –network rinkeby:

```
$ truffle console –network rinkeby
truffle(rinkeby)>
```

Далее проверьте баланс аккаунта:

```
truffle(rinkeby)>
web3.eth.getBalance("0x37Ff26a52677e4dD1BaB543dDAb5905a87F49162")
'29707478800000000000'
```

Теперь, чтобы вызывать методы контракта HelloSol, вам нужно разблокировать аккаунт. Сделайте это таким же образом, что и перед публикацией (миграцией) контракта в Rinkeby:

```
>     personal.unlockAccount(web3.eth.coinbase,   "password",
15000)
```

Теперь в консоли Truffle вы сможете вызывать функции нашего контракта.
Сначала мы вызовем функцию setValue:

```
truffle(rinkeby)> HelloSol.deployed().then(function(instance)
{return instance.setValue(777)});
{ tx:
  '0x270c69215c2bed4b9d6b0137760480d161e95baea5dfd0c3a0304c66b5304a
 receipt:
  { blockHash:
     '0xab64575f8e6d932fc42f73c1a6cd18eab6e6c32fd181cd8908c8205f811
    blockNumber: 4028328,
    contractAddress: null,
    cumulativeGasUsed: 265153,
    from: '0x37ff26a52677e4dd1bab543ddab5905a87f49162',
    gasUsed: 41803,
    logs: [],
    logsBloom:
     '0x00000000000000000000000000000000000000000000000000000000000000
    status: true,
    to: '0xe77d0eaf85e0f7f5a786affdc9fc1240678c84d9',
    transactionHash:
     '0x270c69215c2bed4b9d6b0137760480d161e95baea5dfd0c3a0304c66b53
    transactionIndex: 1,
    rawLogs: [] },
 logs: [] }
```

Затем вызовем функцию getValue:

```
    truffle(rinkeby)>
    undefined
    truffle(rinkeby)> HelloSol.deployed().then(function(instance)
{return          instance.getValue()}).then(function(value){return
value.toNumber()});
    777
```

Теперь снова откройте сайт Rinkeby: Ethereum Testnet и посмотрите информацию об аккаунте (рис. 9.8.).

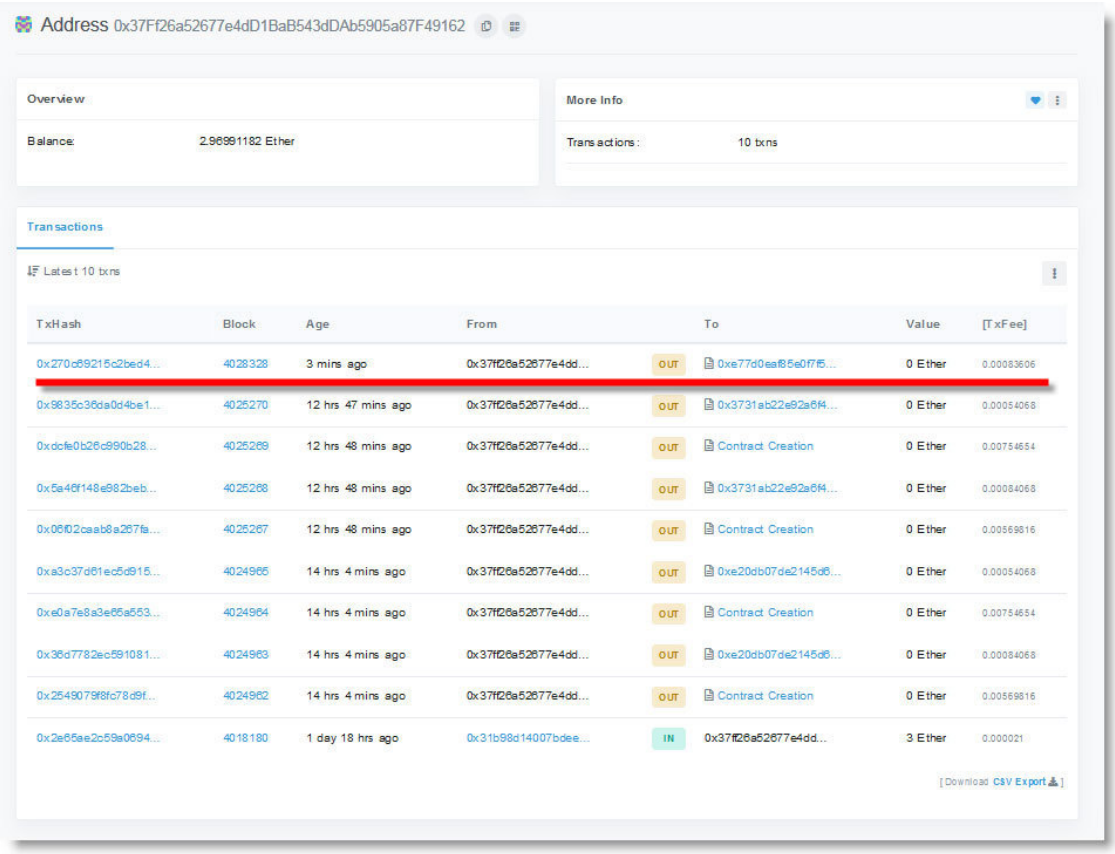

Рис. 9.8. Информация об аккаунте после вызова метода контракта SetValue

Теперь в верхней части списка транзакций появилась новая транзакция, которая была создана в результате вызова метода SetValue. На рис. 9.8. соответствующая строка подчеркнута линией красного цвета.

Если теперь проверить баланс аккаунта, то вы увидите, что он немного уменьшился:

```
    truffle(rinkeby)>
web3.eth.getBalance("0x37Ff26a52677e4dD1BaB543dDAb5905a87F49162")
    '2969911820000000000'
```

Аналогичным образом можно вызвать функции setString и getString нашего контракта:

```
    truffle(rinkeby)> HelloSol.deployed().then(function(instance)
{return instance.setString('New string')});
    { tx:
```

```
      '0x74209264d1d194abe5b613b07fbdf06444717b41feaeec1e52e11befd8504d
     receipt:
      { blockHash:
         '0x21a7ba04d3a573ffa59ef08db4e6f966650e87f8aedacc975e2989e3c10
        blockNumber: 4028372,
        contractAddress: null,
        cumulativeGasUsed: 43585,
        from: '0x37ff26a52677e4dd1bab543ddab5905a87f49162',
        gasUsed: 43585,
        logs: [],
        logsBloom:
         '0x00000000000000000000000000000000000000000000000000000000000000
        status: true,
        to: '0xe77d0eaf85e0f7f5a786affdc9fc1240678c84d9',
        transactionHash:
         '0x74209264d1d194abe5b613b07fbdf06444717b41feaeec1e52e11befd85
        transactionIndex: 0,
        rawLogs: [] },
     logs: [] }

    truffle(rinkeby)> HelloSol.deployed().then(function(instance)
{return instance.getString()});
    'New string'
```

После вызова функции SetString, изменяющей состояние сети, баланс аккаунта снова уменьшится:

```
    truffle(rinkeby)>
web3.eth.getBalance("0x37Ff26a52677e4dD1BaB543dDAb5905a87F49162")
    '2969040120000000000'
```

Более простой способ вызова функций контракта

Фреймворк Truffle позволяет вам вызывать функции контракта и более простым способом.

Вначале вы можете создать объект контракта с применением ключевого слова await:

```
truffle(rinkeby)> let instance = await HelloSol.deployed()
undefined
```

С помощью await можно вызывать асинхронные функции как синхронные. Это может облегчить процесс тестирования.

Для начала вызовем функцию getString:

```
truffle(rinkeby)> instance.getString()
'New string'
```

Как и следует ожидать, функция вернет строку, записанную нами ранее.

Теперь вызовем функцию setString, передав ей в качестве параметра новую строку:

```
truffle(rinkeby)> instance.setString("Новая строка")
{ tx:
  '0xec126a5f1570a6b37ff3146c10b0b3c7c15ff77f9af72d2dab630e2d3016bc
 receipt:
  { blockHash:
     '0x7287088a9500415626188203bf3d6ac44b1202a7a106192297c068f656d
    blockNumber: 4028387,
    contractAddress: null,
    cumulativeGasUsed: 29674,
    from: '0x37ff26a52677e4dd1bab543ddab5905a87f49162',
    gasUsed: 29674,
    logs: [],
    logsBloom:
     '0x00000000000000000000000000000000000000000000000000000000000000
    status: true,
    to: '0xe77d0eaf85e0f7f5a786affdc9fc1240678c84d9',
    transactionHash:
     '0xec126a5f1570a6b37ff3146c10b0b3c7c15ff77f9af72d2dab630e2d301
    transactionIndex: 0,
    rawLogs: [] },
 logs: [] }
```

Будет создана транзакция. После ее выполнения функция getString вернет новое значение:

```
truffle(rinkeby)> instance.getString()
'Новая строка'
```

Вот так в приглашении Truffle вы можете посмотреть список аккаунтов узла сети Rinkeby:

```
truffle(rinkeby)> let accounts = await web3.eth.getAccounts()
undefined
truffle(rinkeby)> accounts
[ '0x37Ff26a52677e4dD1BaB543dDAb5905a87F49162',
 '0x704F35Da18Df4404937747B1C98201cb9d63c706' ]
```

Вызов методов контракта при помощи Node.js

Если контракт опубликован в сети Rinkeby с помощью Truffle, то вы можете вызывать методы контракта из скриптов, аналогично тому, как мы это делали на седьмом уроке, или извлекая адрес контракта из артефактов Truffle.

Напомним, что артефакты Truffle записывает в файл build/contracts/<имя контракта>.json. У нас это файл /home/book/HelloSol/build/contracts/HelloSol.json.

Теперь в разделе networks появится блок для сети с идентификатором 4:

```
    "networks": {
       "4": {
         "events": {},
         "links": {},
         "address": "0xE77D0EAF85E0F7f5A786aFfDc9fC1240678c84D9",
                                                "transactionHash":
"0xdcfe0b26c990b287d77cf23bee00759364657e75127d67c50ceccb4713c56bd2"
       },
       "5777": {
         "events": {},
         "links": {},
         "address": "0x065dAED48E78238bf010c65E2BA66471f4e8d6b7",
                                                "transactionHash":
"0xd8e2bee79abf001d73bddc8d8af58a2a5621f28c7146d86a78a525af6ec99824"
       },
       "98760": {
         "events": {},
         "links": {},
         "address": "0xa65B88734fbe5942f970cB5814ae976d5ce5f00d",
                                                "transactionHash":
"0xc46a87e0b00512069f765aa5904ba856066db2e200382a258900e87069a5b191"
       }
    }
```

Этот идентификатор соответствует сети Rinkeby. Основная же сеть Ethereum имеет идентификатор, равный 1.

С помощью команды truffle networks вы можете посмотреть список сетей, сконфигурированных в Truffle:

```
    $ truffle networks

    The following networks are configured to match any network id
('*'):

       development

    Closely inspect the deployed networks below, and use `truffle
networks --clean` to remove any networks that don't match your
configuration. You should not use the wildcard configuration ('*')
```

```
for staging and production networks for which you intend to deploy
your application.

    Network: UNKNOWN (id: 5777)
     HelloSol: 0x065dAED48E78238bf010c65E2BA66471f4e8d6b7
     Migrations: 0xe540aE5716b487E1bACBB737F08aa35323DaFfA8

    Network: live (id: 98760)
     HelloSol: 0xa65B88734fbe5942f970cB5814ae976d5ce5f00d
     Migrations: 0x98d1331be0Ce4ae7f072A1778a44334042eAEE5f

    Network: rinkeby (id: 4)
     HelloSol: 0xE77D0EAF85E0F7f5A786aFfDc9fC1240678c84D9
     Migrations: 0x3731Ab22E92a6f46237b3722CFd1839d2F7e9eEE
```

Здесь для каждой сети показывается адрес публикации контрактов. Это наш контракт HelloSol и служебный контракт Migrations, который используется Truffle при выполнении миграций.

Перевод средств между аккаунтами в консоли Truffle для Rinkby

При необходимости вы можете перевести средства с одного аккаунта Rinkeby на другой прямо из командного приглашения Truffle:

```
    truffle(rinkeby)>            web3.eth.sendTransaction({from:
'0x37Ff26a52677e4dD1BaB543dDAb5905a87F49162',                to:
'0x704F35Da18Df4404937747B1C98201cb9d63c706', value: 1000})
    { blockHash:
      '0x38a37a6a2e9f40dfa2257d1a385c2dbd84e42565db0681dafeef2639e9a77b
     blockNumber: 4028952,
     contractAddress: null,
     cumulativeGasUsed: 2892913,
     from: '0x37ff26a52677e4dd1bab543ddab5905a87f49162',
     gasUsed: 21000,
     logs: [],
     logsBloom:
      '0x00000000000000000000000000000000000000000000000000000000000000000
     status: true,
     to: '0x704f35da18df4404937747b1c98201cb9d63c706',
     transactionHash:
      '0x43a8410758f2dcce2bebbdb5ec1bb990ad9948c93d2599c6cb5c736c52e413
     transactionIndex: 9 }
```

Здесь мы перевели с одного аккаунта на другой 1000 Wei. Результат можно проверить с помощью вызова web3.eth.getBalance. Баланс первого аккаунта должен уменьшиться на 1000 Wei, а второго – увеличиться на эту же сумму:

```
    truffle(rinkeby)>
web3.eth.getBalance("0x37Ff26a52677e4dD1BaB543dDAb5905a87F49162")
    '2967829599999999000'

    truffle(rinkeby)>
web3.eth.getBalance("0x704F35Da18Df4404937747B1C98201cb9d63c706")
    '1000'
```

Итоги урока

На девятом уроке вы научились работать с тестовой сетью Rinkeby. При этом для разработки смарт-контрактов вы применяли Truffle и Geth.

Вы создали с помощью Geth узел сети Rinkeby и приложили немало терпения, ожидая его синхронизации. Далее вы пополнили свой аккаунт Rinkeby средствами, опубликовали свой смарт-контракт, а также научились переводить средства на другие аккаунты Rinkeby.

Урок 10. Децентрализованное хранилище данных Ethereum Swarm

Цель урока: научиться работать с распределенным хранилищем данных Ethereum Swarm, устанавливать локальный узел Ethereum Swarm для приватной сети Ethereum с целью тестирования технологии и разработки децентрализованных приложений, хранящих данные в Ethereum Swarm.

Практические задания: вам предстоит создать локальное хранилище Swarm, выполнить операции записи и чтения файлов, а также каталогов с файлами. Далее вы научитесь работать с публичным шлюзом Swarm, напишите скрипты для обращения к Swarm из Node.js, а также с помощью модуля Perl Net::Ethereum::Swarm.

Если вы создаете на базе блокчейна Ethereum децентрализованное приложение DApp (Decentralized Application), то ему необходимо децентрализованное хранилище данных.

Почему мы не можем просто хранить все данные непосредственно в блокчейне, записывая их при помощи функций смарт-контрактов?

В теории можем, но, к сожалению, хранение данных большого объема может стоить довольно дорого.

Конечно, вы можете хранить данные в любом облачном сервисе, а в блокчейн записывать только их контрольные суммы для верификации. Однако при этом данные будут храниться только в одном месте, что противоречит принципам работы децентрализованных хранилищ.

На помощь приходят такие децентрализованные хранилища, как Ethereum Swarm («swarm» переводится как «рой», «куча»). Если кратко, то Ethereum Swarm представляет собой программный код, работающий на пиринговой сети Ethereum. Он обеспечивает децентрализованное хранение данных на дисках узлов, владельцы которых отдают свои ресурсы в общее пользование.

Помимо Ethereum Swarm, существуют и другие децентрализованные хранилища данных, например, Sia, Storj, MadeSAFE. За их использование может взиматься плата.

Как работает Ethereum Swarm

По всему миру разбросаны десятки тысяч узлов сети Ethereum Swarm. Они предоставляют свои ресурсы для хранения данных, загруженных пользователями. Предполагается, что владельцы узлов будут получать вознаграждение за предоставление ресурсов, при этом стоимость размещения данных будет ниже, чем в традиционных облачных хранилищах.

Когда пользователь загружает файл в сеть Ethereum Swarm, этот файл сначала попадает на один из узлов. Далее файл реплицируется на остальные узлы сети в процессе синхронизации. При этом используется протокол bzz, работающий поверх сети Ethereum.

До тех пор пока работает хоть один узел Ethereum Swarm, загруженный файл остается доступным. Это обеспечивает надежность хранения данных, т.к. практически невозможно вывести из строя или заблокировать огромное количество узлов Ethereum Swarm.

Также необходимо учесть, что **данные, загруженные в публичную сеть Ethereum Swarm, останутся там навсегда** и нет никакого способа удалить их оттуда. Будьте осторожны, загружая в Swarm персональные данные, личные фотографии и любую другую чувствительную информацию.

Подробнее о Swarm вы сможете прочитать на сайте проекта https://swarm-guide.readthedocs.io.

На момент подготовки этого руководства был доступен Ethereum Swarm стабильной версии 0.3.12, созданный как доказательство концепции (POC, proof of concept). Пока, к сожалению, сохранность данных, загруженных в Swarm, не гарантируется. Также пока не создана система вознаграждений за предоставление ресурсов сети Ethereum Swarm.

Установка и запуск Swarm

Документацию по Swarm вы найдете здесь: https://swarm-guide.readthedocs.io/en/latest/introduction.html.

Перед установкой Swarm вам необходимо установить узел Geth, пользуясь материалом предыдущих уроков.

Также мы рекомендуем установить программу jq, которая поможет нам просматривать данные JSON в форматированном виде:

```
$ sudo apt install jq
```

Описание утилиты jq вы найдете здесь: https://stedolan.github.io/jq/.

Для установки распределенного хранилища Swarm на локальный тестовый узел используйте следующую команду:

```
$ sudo apt-get install ethereum-swarm
```

После установки проверьте версию Swarm:

```
$ swarm version
Swarm
Version: 0.3.12-stable
Git Commit: 4e13a09c5033b4cf073db6aeaaa7d159dcf07f30
Go Version: go1.10.4
OS: linux
```

Для запуска Swarm потребуется адрес аккаунта, созданного на этапе инициализации узла. Чтобы узнать этот адрес, в консоли geth, открытой с помощью скрипта attach_node.sh, выдайте следующую команду:

```
> web3.eth.accounts
["0xaebde32aa52e8f85a0c96a237b9c4678d901cd52"]
```

Для запуска демона Ethereum Swarm в режиме единственного узла (Singleton) подготовьте пакетный файл swarm_start.sh (листинг 10.1.).

Листинг 10.1. Файл swarm_start.sh

```
swarm -bzzaccount aebde32aa52e8f85a0c96a237b9c4678d901cd52 -
datadir "/home/book/node1" -maxpeers 0 -bzznetworkid 17 -verbosity
4 -ens-api /home/book/node1/geth.ipc
```

В параметре bzzaccount укажите здесь адрес аккаунта, созданного на вашем узле, без «0x». Параметр bzznetworkid задается в виде произвольного числа от 15 до 255. Он необходим для работы в режиме Singleton, когда ваш узел не будет подключаться к другим узлам сети Swarm.

При запуске демона вам будет нужно ввести пароль от созданного ранее аккаунта:

```
    $ sh swarm_start.sh
    Unlocking swarm account
0xCD9Fcb450C858D1A7678a2bCCf36EA5decd2B09B [1/3]
    Passphrase:
```

Если вы используете тестовую сеть geth, созданную с параметром –dev, вместо ввода пароля просто нажмите клавишу Enter.

Операции с файлами и каталогами

С помощью простых команд вы можете загружать файлы из каталога в Swarm, просматривать информацию о загруженных данных, а также читать данные из хранилища Swarm. Приведем некоторые примеры таких операций.

Загрузка файла в Ethereum Swarm

Проще всего загрузить файл с помощью команды swarm с параметром up. Дополнительно нужно указать путь к загружаемому файлу.

Для загрузки мы подготовили текстовый файл с именем testfile.txt (листинг 10.2.). Вы, конечно, можете использовать вместо него свой файл с любым содержимым.

Листинг 10.2. Файл testfile.txt

```
«SkyNet всюду и нигде единого центра нет. Отключать нечего.» (Терминатор-3: Восстание машин)
```

Чтобы загрузить файл testfile.txt в Swarm, используйте следующую команду:

```
book@bookether:~$ swarm up testfile.txt
5a0320520501491c7a0538e3a0584a97a6f6603a6b1949784d977309c4c83148
```

Команда вернет хеш загруженного файла. Хеш можно использовать для чтения файла.

Чтение файла из Ethereum Swarm

Прочитать файл из хранилища Swarm нетрудно с помощью команд wget или curl, передав им в качестве параметра хеш загруженного ранее файла:

```
~$ wget http://localhost:8500/bzz:/5a0320520501491c7a0538e3a0584a97a6f6603a6b1949784d977309c4c83148/ -O get.txt
--2019-03-14 13:38:19- http://localhost:8500/bzz:/5a0320520501491c7a0538e3a0584a97a6f6603a6b1949784d977309c4c83148/
Resolving localhost (localhost)… ::1, 127.0.0.1
Connecting to localhost (localhost)|::1|:8500… failed: Connection refused.
Connecting to localhost (localhost)|127.0.0.1|:8500… connected.
HTTP request sent, awaiting response… 200 OK
Length: 161 [text/plain]
Saving to: ‘get.txt’

get.txt 100%[=>] 161 -.-KB/s in 0s

2019-03-14 13:38:19 (28.2 MB/s) - ‘get.txt’ saved [161/161]
```

Параметр -O задает имя файла, в котором будут сохранены прочитанные данные. Вы можете просмотреть содержимое этого файла, например, при помощи команды cat:

```
    book@bookether:~$ cat get.txt
    «SkyNet всюду и нигде единого центра нет. Отключать
нечего.» (Терминатор-3: Восстание машин)
```

Для загрузки данных из Swarm можно также использовать команду curl:

```
    $ curl http://localhost:8500/
bzz:/5a0320520501491c7a0538e3a0584a97a6f6603a6b1949784d977309c4c83148/
    «SkyNet всюду и нигде единого центра нет. Отключать
нечего.» (Терминатор-3: Восстание машин)
```

Эта команда выведет содержимое файла на консоль, поэтому в таком виде ее не следует использовать для просмотра содержимого бинарных файлов. В конце URL необходим слеш, иначе произойдет редирект.

Когда файл загружается в Ethereum Swarm описанным выше образом, для него создается и сохраняется так называемый манифест. Это заголовок, описывающий содержимое, доступное в хранилище по заданному идентификатору.

Просмотр манифеста загруженного файла

Для просмотра манифеста загруженного ранее файла нужно указать протокол bzz-list:

```
    $ curl http://localhost:8500/bzz-
list:/5a0320520501491c7a0538e3a0584a97a6f6603a6b1949784d977309c4c83148/
    {"entries":
[{"hash":"6c4082c04143ff4399f1e9b65df45465342841b173d7d5665f7a24604b4fb
plain;
charset=utf-8","mode":436,"size":161,"mod_time":"2019-03-14T13:34:07Z"}
```

Манифест будет показан в формате JSON. Используйте утилиту jq для форматирования результата:

```
    $ curl http://localhost:8500/bzz-
list:/5a0320520501491c7a0538e3a0584a97a6f6603a6b1949784d977309c4c83148/
| jq .
       %   Total            %   Received   %   Xferd      Average
Speed   Time    Time     Time  Current
                           Dload  Upload  Total  Spent   Left  Speed
    100   199  100   199    0    0   7960     0 -:-:- -:-:- -:-:-  7960
    {
     "entries": [
       {
                                                          "hash":
"6c4082c04143ff4399f1e9b65df45465342841b173d7d5665f7a24604b4fbcd9",
         "path": "/",
         "contentType": "text/plain; charset=utf-8",
```

```
        "mode": 436,
        "size": 161,
        "mod_time": "2019-03-14T13:34:07Z"
      }
    ]
  }
```

В манифесте хранятся пусть к файлу (имя файла), его размер, тип (Content Type), кодировка charset, дата и время модификации, размер, а также хеш файла.

Загрузка каталогов с подкаталогами

Для рекурсивной загрузки каталога вместе с его содержимым в хранилище Ethereum Swarm укажите параметр –recursive:

```
$ swarm –recursive up HelloSol
0a1afa1c86a009b3e380238c2c46ad495fcf708ff020407c419173b0c561b66d
```

В манифесте будет показана информация обо всех файлах загруженного подкаталога:

```
$ curl http://localhost:8500/bzz-list:/0a1afa1c86a009b3e380238c2c46ad495fcf708ff020407c419173b0c561b66d/ | jq .
  %   Total    %   Received   %   Xferd   Average Speed   Time    Time     Time  Current
                                 Dload  Upload  Total  Spent   Left  Speed
100   663  100   663    0    0   215k     0 -:-:- -:-:- -:-:-  215k
{
  "common_prefixes": [
    "build/",
    "contracts/",
    "migrations/"
  ],
  "entries": [
    {
                                                                "hash": "c7e1a4d55577bbddb176c0e8f3270d55a08b5d1e6be38a31b339124aa5826e5d",
      "path": "call_contract_1.0.js",
      "contentType": "application/javascript",
      "mode": 436,
      "size": 2387,
      "mod_time": "2019-03-12T11:39:05Z"
    },
    {
                                                                "hash": "2e25aebc6936a1ba63d66c5e0084e9d914fe368c3c4731931aa0e795dc3eecf6",
      "path": "truffle-config.js",
      "contentType": "application/javascript",
      "mode": 436,
```

```
        "size": 1040,
        "mod_time": "2019-03-12T08:52:04Z"
      },
      {
                                                          "hash":
"06ab08c595cb65baea4304187e31ea0699bffc067391d16d8d8972dd87f46268",
        "path": "truffle.js",
        "contentType": "application/javascript",
        "mode": 436,
        "size": 1368,
        "mod_time": "2019-03-13T18:48:48Z"
      }
    ]
  }
```

Чтение файла из загруженного каталога

Чтобы прочитать файл из загруженного каталога, нужно указать хеш каталога, а также имя файла:

```
  $ wget http://localhost:8500/
bzz:/0a1afa1c86a009b3e380238c2c46ad495fcf708ff020407c419173b0c561b66d/
truffle.js -O truffle.js
```

Здесь мы восстановили файл truffle.js.

Использование публичного шлюза Swarm

В предыдущей части этого урока мы загружали файлы в свой локальный узел сети Ethereum. Однако существует публичный шлюз этого распределенного хранилища, работающий пока в тестовом режиме. В нем существуют ограничения на размеры файлов, загружаемых в публичную сеть Swarm, но позже эти ограничения обещают снять.

Адрес шлюза публичной сети Swarm https://swarm-gateways.net нужно указывать при помощи параметра –bzzapi.

Следующая команда загружает в публичное хранилище Swarm файл testfile.txt:

```
$ swarm –bzzapi https://swarm-gateways.net up testfile.txt
5a0320520501491c7a0538e3a0584a97a6f6603a6b1949784d977309c4c83148
```

Команда возвращает хеш, который можно использовать для чтения файла:

```
swarm –bzzapi https://swarm-gateways.net down
bzz:/5a0320520501491c7a0538e3a0584a97a6f6603a6b1949784d977309c4c83148
file.tmp
```

Когда вы будете пробовать эти команды, помните, что файл, загруженный в публичную сеть Swarm, **невозможно удалить**.

После того как вы загрузили файл, откройте сайт https://swarm-gateways.net в браузере (рис. 10.1.).

Рис. 10.1. Сайт публичного интерфейса Swarm

Введите в поисковой строке хеш загруженного файла, и вы увидите его содержимое. Зная хеш файла, любой может получить к нему доступ через этот сайт.

Вы также можете работать с публичным хранилищем Swarm с помощью утилит curl или wget, указывая адрес шлюза https://swarm-gateways.net.

Следующая команда просматривает манифест файла по его хешу:

```
    $             curl             https://swarm-gateways.net/bzz-
list:/5a0320520501491c7a0538e3a0584a97a6f6603a6b1949784d977309c4c83148/
| jq
       %   Total            %   Received   %   Xferd      Average
Speed   Time    Time     Time  Current
                           Dload  Upload  Total  Spent   Left  Speed
    100   199  100   199    0    0   646     0 -:-:- -:-:- -:-:-   646
    {
     "entries": [
       {
                                                                "hash":
"6c4082c04143ff4399f1e9b65df45465342841b173d7d5665f7a24604b4fbcd9",
         "path": "/",
         "contentType": "text/plain; charset=utf-8",
         "mode": 436,
         "size": 161,
         "mod_time": "2019-03-14T13:34:07Z"
       }
     ]
    }
```

Чтобы прочитать загруженный файл из публичного хранилища Swarm, можно использовать такую команду:

```
    $          wget                    https://swarm-gateways.net/
bzz:/5a0320520501491c7a0538e3a0584a97a6f6603a6b1949784d977309c4c83148/
-O get.txt
```

Если вы работаете с публичным хранилищем Swarm, то вам не нужно запускать ПО хранилища (Geth и Swarm) на локальном узле.

Обращение к Swarm из скриптов Node.js

Для того чтобы записывать данные в Swarm и считывать их из этой сети, можно использовать один из пакетов JavaScript.

Мы будем использовать на этом уроке пакет swarmgw. Установите его следующим образом:

```
$ npm install -g swarmgw
```

Далее подготовьте два скрипта. Скрипт swarmgw_put.js будет записывать данные в сеть Swarm (листинг 10.3.), а скрипт swarmgw_get.js (листинг 10.4.) – читать их.

Листинг 10.3. Файл swarmgw_put.js

```
var str_to_put = process.argv[2];
const swarmgw = require('swarmgw')( ['mode', 'https'],
['gateway', 'localhost:8500'])

swarmgw.put(str_to_put, function (err, ret) {
 if (err) {
   console.log('Failed to upload: ' + err)
 } else {
   console.log('Swarm hash: ' + ret)
 }
})
```

Листинг 10.4. Файл swarmgw_get.js

```
var swarm_hash = process.argv[2];
const swarmgw = require('swarmgw')( ['mode', 'https'],
['gateway', 'localhost:8500'])

swarmgw.get("bzz-raw://" + swarm_hash, function (err, ret) {
 if (err) {
   abort('Failed to download: ' + err)
 } else {
   console.log(ret)
 }
})
```

Подключая пакет swarmgw, вы можете указывать два параметра – mode и gateway:

```
const swarmgw = require('swarmgw')( ['mode', 'https'],
['gateway', 'localhost:8500'])
```

Первый из них определяет протокол (http или https), а второй задает адрес шлюза сети Swarm. Эти параметры можно не задавать. При этом по умолчанию будет использован протокол https и шлюз swarm-gateways.net:

```
const swarmgw = require('swarmgw')()
```

При запуске передайте скрипту swarmgw_put.js произвольную текстовую строку, которую нужно записать в хранилище Swarm:

```
$ node swarmgw_put.js "testing string"
Swarm hash:
d24b21f1f24942274d9fa84b585aa8bc7d3fb8cb826d6d2e328646db2c9f069c
```

В ответ скрипт выведет на консоль хеш, с помощью которого можно прочитать данные из Swarm.

Передайте этот хеш при запуске скрипту swarmgw_get.js, и он вернет записанные ранее данные:

```
$ node swarmgw_get.js
d24b21f1f24942274d9fa84b585aa8bc7d3fb8cb826d6d2e328646db2c9f069c
testing string
```

Описание пакета swarmgw и его исходные тексты вы найдете здесь: https://github.com/axic/swarmgw.

Модуль Perl Net::Ethereum::Swarm

Взаимодействие через HTTP-запросы можно легко реализовать практически на любом языке программирования.

Для того чтобы работать с децентрализованным хранилищем данных Ethereum Swarm в системах, написанных на языке Perl, автор этого руководства разработал и выложил на CPAN модуль Net::Ethereum::Swarm.

С помощью модуля Net::Ethereum::Swarm вы сможете загружать в Ethereum Swarm текстовые и бинарные файлы, получать манифест для загруженных данных, а также загружать файлы из Ethereum Swarm по их идентификатору.

Модуль Net::Ethereum::Swarm работает с узлом Ethereum Swarm с помощью HTTP-запросов GET и POST. Использование запросов описано в разделе The HTTP API документации Swarm.

Установка модуля Net::Ethereum::Swarm

Чтобы установить модуль Net::Ethereum::Swarm, используйте следующую команду:

```
$ sudo cpan Net::Ethereum::Swarm
```

Описание модуля вы найдете здесь: https://metacpan.org/pod/Net::Ethereum::Swarm.

Запись и чтение данных

Чтобы записать файл в Swarm с помощью модуля Net::Ethereum::Swarm, мы подготовили скрипт swarm_put.pl (листинг 10.5.).

Листинг 10.5. Файл swarm_put.pl

```
use Net::Ethereum::Swarm;
use Data::Dumper;

my $uploaded_file_path = $ARGV[0];
#my     $sw_node     =     Net::Ethereum::Swarm->new('http://
localhost:8500/');
my    $sw_node    =    Net::Ethereum::Swarm->new('http://swarm-
gateways.net/');

my                $hash                =                $sw_node-
>_swarp_node_upload_text_file($uploaded_file_path,    'plain/text;
charset=UTF-8');

print("Hash: ", $hash, "\n");

my $rc = $sw_node->_swarp_node_get_manifest($hash);
print Dumper($rc), "\n";
```

Здесь строкой комментария мы закрыли обращение к локальному узлу Swarm, поэтому данный вариант будет записывать файл в публичное хранилище Swarm.

После записи файла в хранилище скрипт swarm_put.pl показывает на консоли хеш файла, а также его манифест:

```
    $ perl swarm_put.pl testfile.txt
    Hash:
1b133708e9e7f388a7fe6d2413fd7723deb323b98c45271673869e0c29361bbc
    $VAR1 = {
     'entries' => [
      {
       'mode' => 420,
       'mod_time' => '0001-01-01T00:00:00Z',
        path' => '/',
                                                hash'          =>
'6c4082c04143ff4399f1e9b65df45465342841b173d7d5665f7a24604b4fbcd9',
        'contentType' => 'plain/text; charset=UTF-8',
        'size' => 161
       }
      ]
    };
```

Для чтения файла используйте скрипт swarm_get.pl (листинг 10.6.).

Листинг 10.6. Файл swarm_get.pl

```
    use Net::Ethereum::Swarm;
    use Data::Dumper;

    my $manifest_id = $ARGV[0];
    my $file_path_to_save = $ARGV[1];

    #my     $sw_node     =     Net::Ethereum::Swarm->new('http://
localhost:8500/');
    my    $sw_node    =    Net::Ethereum::Swarm->new('http://swarm-
gateways.net/');

    my    $rc    =    $sw_node->_swarp_node_get_file($manifest_id,
$file_path_to_save, 'plain/text; charset=UTF-8');
    print Dumper($rc), "\n";
```

В качестве параметра этому скрипту нужно передать хеш файла, выведенный на консоль скриптом swarm_put.pl:

```
    book@bookether:~$              perl              swarm_get.pl
1b133708e9e7f388a7fe6d2413fd7723deb323b98c45271673869e0c29361bbc
    $VAR1 = '«SkyNet всюду и нигде единого центра нет. Отключать
нечего.» (Терминатор-3: Восстание машин)
    ';
```

После запуска вы увидите на консоли содержимое файла, прочитанного из хранилища.

Итоги урока

На 10-м уроке вы узнали, что хранение данных большого объема в блокчейне может стоить дорого. Вы научились решать эту проблему с помощью распределенного хранилища данных Ethereum Swarm.

Вы создали собственное локальное хранилище Swarm, научились записывать в него файлы и каталоги с файлами, а также получать эти данные обратно. Эти же операции вы проделали и с публичным шлюзом Swarm.

Вы составили скрипты для работы с хранилищем Swarm из Node.js, а также из программ Perl при помощи модуля Net::Ethereum::Swarm.

Урок 11. Фреймворк Web3.py для работы с Ethereum на Python

Цель урока: научиться устанавливать и использовать фреймворк Web3.py, предназначенный для работы с узлами Ethereum с помощью языка программирования Python.

Практические задания: вам предстоит установить фреймворк Web3.py, научиться с его помощью компилировать и публиковать смарт-контракты, а также вызывать функции смарт-контрактов. Вам также предстоит использовать Web3.py совместно с Truffle.

Очень многие разработчики децентрализованных приложений Solidity используют рассмотренный нами ранее фреймворк Web3, предполагающий использование JavaScript. Однако существуют фреймворки и для других языков программирования, например, C++, Python, Go, Perl и т.д.

На этом уроке мы займемся установкой и изучением фреймворка Web3.py, который будет интересен пользователям Python.

На сайте https://web3py.readthedocs.io/en/stable/index.html вы найдете документацию по фреймворку Web3.py.

Установка Web3.py

Для работы с фреймворком Web3.py нужно установить Python версии 3.x. Мы будем работать с версией 3.6.7.

Обновление и установка необходимых пакетов

Обновите пакеты и выполните установку следующим образом (предполагается, что мы работаем в ОС Ubuntu):

```
$ sudo apt update
$ sudo apt upgrade
$ sudo apt install software-properties-common
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt install python3
$ sudo apt-get install python3-pip
```

После установки проверьте версию python:

```
$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
```

Далее с помощью pip3 установите фреймворк Web3.py:

```
$ pip3 install web3
```

Чтобы работать с локальной сетью geth, запущенной с параметром –dev, также нужно установить «песочницу» eth-tester, специально предназначенную для тестирования приложений Ethereum:

```
$ pip3 install –U web3[tester]
```

Описание «песочницы» находится здесь: https://github.com/ethereum/eth-tester.
Результаты установки можно проверить в командном приглашении python3:

```
>>> from web3 import Web3, IPCProvider
>>> w3 = Web3(IPCProvider('/home/book/node1/geth.ipc'))
>>> from web3.middleware import geth_poa_middleware
>>> w3.middleware_stack.inject(geth_poa_middleware, layer=0)
```

Здесь после импорта Web3 и провайдера IPCProvider, необходимого для подключения к узлу geth, мы загрузили модуль geth_poa_middleware, необходимый для совместимости с Geth. Обсуждение этого модуля вы можете найти здесь: https://web3py.readthedocs.io/en/stable/middleware.html

Прежде всего проверьте версию узла Geth:

```
>>> w3.version.node
'Geth/v1.8.24-stable-4e13a09c/linux-amd64/go1.10.4'
```

Далее вы можете просмотреть номер последнего блока в вашей отладочной сети:

```
>>> w3.eth.blockNumber
18
```

И наконец, посмотрите информацию о самом последнем блоке с помощью вызова функции getBlock:

```
    >>> w3.eth.getBlock('latest')
    AttributeDict({'difficulty':    2,    'proofOfAuthorityData':
HexBytes('0xd883010817846765746888676f312e31302e34856c696e7578000000000
'gasLimit':      6394525,      'gasUsed':      34346,      'hash':
HexBytes('0x60523bda1c1d568057efd92daeadd9a2ffb9873ae5857e68e2a3972d547
'logsBloom':
HexBytes('0x00000000000000000000000000000000000000000000000000000000000000000
'miner': '0x0000000000000000000000000000000000000000', 'mixHash':
HexBytes('0x00000000000000000000000000000000000000000000000000000000000000000
'nonce':     HexBytes('0x0000000000000000'),     'number':     18,
'parentHash':
HexBytes('0x1b8c7b701d774fe8484a1e791e2f717d6ea384fbc70ad2f02665bf66bda
'receiptsRoot':
HexBytes('0xa731f923f7a7fe37172fc1e9217dd5a1d30c6fe5b3f9380f2e71c078a82
'sha3Uncles':
HexBytes('0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d
'size':                      815,                      'stateRoot':
HexBytes('0x2389b804de29e109e9b9b5db7d7d6ac347e2c662cc9f0a97aed55119b44
'timestamp': 1552390748, 'totalDifficulty': 37, 'transactions':
[HexBytes('0x3d80af81692cc4815c63ffee6c39a43a1cb734fbee581be2e9794ad1e8
'transactionsRoot':
HexBytes('0xbc67847e399048034857af35152340ca01a112c7fc84509843cc5d59369
'uncles': []})
```

Если все эти команды отработали нормально, можно продолжить изучение фреймворка Web3.py.

Установка модуля easysolc

На момент создания этого руководства была доступна версия фреймворка Web3.py, предназначенная для работы с компилятором solc версии 0.4.x. Для того чтобы использовать компилятор версии 0.5.x, нам нужно будет дополнительно установить модуль easysolc. Описание и исходные тексты этого модуля доступны по адресу https://github.com/brunneis/easysolc.

При помощи следующих команд установите необходимые модули:

```
$ pip3 install py-solc
$ pip3 install var_dump
$ pip3 install easysolc
$ pip3 install json
```

Публикация контракта с помощью Web3.py

Фреймворк Web3.py содержит функциональность, необходимую для компиляции и публикации контракта. В листинге 11.1. мы привели исходный текст скрипта, который получает в качестве параметра запуска имя контракта, компилирует и публикует контракт.

Листинг 11.1. Файл migrate_5.py

```
import json
import web3
import sys

from web3 import Web3
from solc import compile_source
from web3.middleware import geth_poa_middleware
from var_dump import var_dump

from easysolc import Solc
solc = Solc()

contract_name = sys.argv[1]
contract_dict = solc.compile(contract_name + '.sol')

w3      =      Web3(Web3.HTTPProvider("http://127.0.0.1:8545",
request_kwargs={'timeout': 60}))
w3.middleware_stack.inject(geth_poa_middleware, layer=0)
w3.eth.defaultAccount = w3.eth.accounts[0]

abi = contract_dict[contract_name]["abi"]
bytecode = contract_dict[contract_name]["bytecode"]
#var_dump(abi)

w3      =      Web3(Web3.HTTPProvider("http://127.0.0.1:8545",
request_kwargs={'timeout': 60}))
w3.middleware_stack.inject(geth_poa_middleware, layer=0)
w3.eth.defaultAccount = w3.eth.accounts[0]

HelloSol = w3.eth.contract(abi=abi, bytecode=bytecode)
tx_hash = HelloSol.constructor().transact()
tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)
var_dump(tx_receipt)

hello_sol = w3.eth.contract(
   address=tx_receipt.contractAddress,
   abi=abi,
)

contract_address = tx_receipt.contractAddress
print("Contract address: " + tx_receipt.contractAddress)
```

```
vars = {
   'contract_address' : contract_address,
   'contract_abi' : abi
}
with open(contract_name + '.json', 'w') as outfile:
   json.dump(vars, outfile)
```

Расскажем, что же делает этот скрипт.

Компиляция контракта

Прежде всего наш скрипт создает объект Solc, с помощью которого будет компилироваться наш смарт-контракт:

```
from easysolc import Solc
solc = Solc()
```

Имя контракта передается скрипту в качестве параметра. Далее при помощи метода solc.compile запускается компиляция контракта:

```
contract_name = sys.argv[1]
contract_dict = solc.compile(contract_name + '.sol')
```

Результаты компиляции (бинарный код и abi) сохраняются в словаре contract_dict.

Подключение к провайдеру

После того как объект интерфейса контракта получен, нам необходимо получить объект провайдера. Мы используем здесь провайдер Web3.HTTPProvider, указывая ему IP и порт, предоставляемый программой Geth, которая была запущена в режиме разработки с ключом –dev:

```
w3 = Web3(Web3.HTTPProvider("http://127.0.0.1:8545",
request_kwargs={'timeout': 60}))
w3.middleware_stack.inject(geth_poa_middleware, layer=0)
```

Из-за особенностей реализации Web3.py для работы с Geth необходимо подключить w3.middleware_stack. Подробнее об этом можно причитать здесь: https://web3py.readthedocs.io/en/stable/middleware.html.

Далее мы сохраняем в объекте w3 адрес основного аккаунта нашего локального узла Ethereum, от которого будет выполняться публикация контракта:

```
w3.eth.defaultAccount = w3.eth.accounts[0]
```

В переменных abi и bytecode сохраняются соответственно интерфейс ABI и байт-код откомпилированного контракта:

```
abi = contract_dict[contract_name]["abi"]
```

```
bytecode = contract_dict[contract_name]["bytecode"]
```

Выполнение публикации контракта

Чтобы выполнить публикацию, мы создаем объект контракта при помощи w3.eth.contract, передавая конструктору интерфейс abi, а также байт-код контракта:

```
HelloSol = w3.eth.contract(abi=abi, bytecode=bytecode)
```

Теперь остается только инициировать транзакцию, в рамках которой будет выполнена публикация, и дождаться ее завершения:

```
tx_hash = HelloSol.constructor().transact()
tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)
```

Когда транзакция завершится, мы выводим на консоль ее квитанцию:

```
var_dump(tx_receipt)
```

Также показываем на консоли адрес опубликованного контракта:

```
contract_address = tx_receipt.contractAddress
print("Contract address: " + tx_receipt.contractAddress)
```

Сохранение адреса контракта и abi в файле

Перед завершением своей работы скрипт сохраняет адрес опубликованного контракта и abi в текстовом файле JSON:

```
vars = {
   'contract_address' : contract_address,
   'contract_abi' : abi
}
with open(contract_name + '.json', 'w') as outfile:
   json.dump(vars, outfile)
```

Имя у этого файла такое же, как имя контракта, расширение имени .json. Файл с адресом контракта и abi будет нам нужен в следующем разделе, где мы займемся вызовом методов опубликованного контракта.

Запуск скрипта публикации контракта

Как мы уже говорили, при запуске мы передаем скрипту migrate.py имя контракта:

```
$ python3 migrate_5.py HelloSol
2019-04-11      07:59:42      [INFO]      Solc      version:
0.5.7+commit.6da8b019.Linux.g++
#0 object(AttributeDict) (12)
   blockHash => object(HexBytes) (0)
```

```
       blockNumber => int(30)
                          contractAddress       =>       str(42)
"0x7771735a6d07aC1A5D9F9C363C3F9f4CE9140074"
       cumulativeGasUsed => int(323129)
      from => str(42) "0xea685835e37e0495b63cd99f4943a5e2ca131eda"
       gasUsed => int(323129)
       logs => list(0)
       logsBloom => object(HexBytes) (0)
       status => int(1)
       to => NoneType(None)
       transactionHash => object(HexBytes) (0)
       transactionIndex => int(0)
    Contract address: 0x7771735a6d07aC1A5D9F9C363C3F9f4CE9140074
```

Когда контракт будет опубликован, вы увидите на консоли квитанцию, а также адрес опубликованного контракта. Из квитанции можно узнать, в частности, сколько газа было использовано на публикацию контракта.

Вызов методов контракта

Скрипт call_set.py (листинг 11.2.) демонстрирует вызов методов контракта, опубликованного при помощи скрипта migrate.py (листинг 11.1.).

Листинг 11.2. Файл call_set.py

```
import json
import web3
import sys
import json
from web3 import Web3
from web3.middleware import geth_poa_middleware
from var_dump import var_dump

contract_name = sys.argv[1]
print(contract_name);

with open (contract_name + '.json', 'r') as read_file:
   vars = json.load(read_file)
#print(json.dumps(vars, sort_keys=True, indent=2))

contract_address = vars['contract_address']
abi = vars['contract_abi']

#var_dump(abi)
print('Contract address: ' + contract_address)

w3     =     Web3(Web3.HTTPProvider("http://127.0.0.1:8545",
request_kwargs={'timeout': 60}))
w3.middleware_stack.inject(geth_poa_middleware, layer=0)
w3.eth.defaultAccount = w3.eth.accounts[0]

hello_sol    =      w3.eth.contract(address=contract_address,
abi=abi)

tx_hash = hello_sol.functions.setValue(777888).transact()
w3.eth.waitForTransactionReceipt(tx_hash)

tx_hash1    =    hello_sol.functions.setString('Test    string
888').transact()
w3.eth.waitForTransactionReceipt(tx_hash1)

print(              "getValue:              %d"              %
(hello_sol.functions.getValue().call()) )

print('getString: {}'.format(
   hello_sol.functions.getString().call()
))
```

Рассмотрим работу нашего скрипта.

Чтение адреса и abi контракта из файла JSON

При запуске скрипту нужно передать имя контракта:

```
$ python3 call_set.py HelloSol
HelloSol
Contract address: 0x7f24CE42D3fD283841ce274145F1DB3D3629502b
getValue: 777888
getString: Test string 888
```

Скрипт call_set.py получает из командной строки имя контракта и читает соответствующий файл JSON, содержащий адрес контракта и abi:

```
contract_name = sys.argv[1]
print(contract_name);

with open (contract_name + '.json', 'r') as read_file:
   vars = json.load(read_file)

contract_address = vars['contract_address']
abi = vars['contract_abi']
print('Contract address: ' + contract_address)
```

Адрес контракта выводится на консоль для контроля.

Подключение к провайдеру

На следующем шаге скрипт подключается к узлу через провайдера Web3.HTTPProvider:

```
w3 = Web3(Web3.HTTPProvider("http://127.0.0.1:8545",
request_kwargs={'timeout': 60}))
w3.middleware_stack.inject(geth_poa_middleware, layer=0)
w3.eth.defaultAccount = w3.eth.accounts[0]
```

Эта операция выполняется таким же образом, что и в скрипте публикации контракта migrate.py (листинг 11.1.).

Создание объекта контракта

Прежде чем мы сможем вызывать функции контракта, необходимо создать соответствующий объект. Для этого мы передаем конструктору w3.eth.contract адрес и abi контракта, загруженные из файла JSON:

```
hello_sol = w3.eth.contract(address=contract_address,
abi=abi)
```

Вызов методов контракта

Когда объект контракта создан, мы первым делом вызываем функции контракта setValue и setString. Эти функции сохраняют в контракте числовое и строковое значения соответственно:

```
    tx_hash = hello_sol.functions.setValue(777888).transact()
    w3.eth.waitForTransactionReceipt(tx_hash)

    tx_hash1    =    hello_sol.functions.setString('Test    string
888').transact()
    w3.eth.waitForTransactionReceipt(tx_hash1)
```

Далее мы читаем сохраненные значения, вызывая функции getValue и getString, отображая полученные значение на консоли:

```
    print(              "getValue:              %d"              %
(hello_sol.functions.getValue().call()) )

    print('getString: {}'.format(
       hello_sol.functions.getString().call()
    ))
```

Truffle и Web3.py

Если вы используете для отладки интегрированную среду разработки смарт-контрактов Truffle, но вызов функций контракта вам нужно выполнять из программ Python, то это тоже возможно при помощи фреймворка Web3.py.

В листинге 11.3. мы извлекаем из результата миграции (артефактов Truffle) адрес контракта и abi, а потом используем эти данные для вызова функции контракта.

Листинг 11.3. Файл call_contract.py

```
import json
import web3
import sys
import json
from web3 import Web3
from web3.middleware import geth_poa_middleware
from var_dump import var_dump

contract_name = sys.argv[1]
network_id = sys.argv[2]
print(contract_name);

with open("/home/book/HelloSol/build/contracts/"
+ contract_name + '.json') as f:
    info_json = json.load(f)

abi = info_json["abi"]
contract_address = info_json["networks"][network_id]
['address']

print('Contract address: ' + contract_address)

w3 = Web3(Web3.HTTPProvider("http://127.0.0.1:9545",
request_kwargs={'timeout': 60}))
w3.middleware_stack.inject(geth_poa_middleware, layer=0)
w3.eth.defaultAccount = w3.eth.accounts[0]

hello_sol = w3.eth.contract(address=contract_address,
abi=abi)

tx_hash = hello_sol.functions.setValue(777888).transact()
w3.eth.waitForTransactionReceipt(tx_hash)

tx_hash1 = hello_sol.functions.setString('Test string
888').transact()
w3.eth.waitForTransactionReceipt(tx_hash1)

print( "getValue: %d" %
(hello_sol.functions.getValue().call()) )
```

```
print('getString: {}'.format(
   hello_sol.functions.getString().call()
))
```

При запуске скрипту call_contract.py нужно передать два параметра – имя контракта, а также идентификатор сети Ethereum.

Получив эти параметры, наш скрипт сначала загружает файл артефактов из каталога build/contracts/ проекта Truffle:

```
contract_name = sys.argv[1]
network_id = sys.argv[2]
print(contract_name);

with              open("/home/book/HelloSol/build/contracts/"
+ contract_name + '.json') as f:
    info_json = json.load(f)
```

Далее скрипт call_contract.py извлекает из загруженного файла JSON необходимые данные и выводит адрес контракта на консоль:

```
abi = info_json["abi"]
contract_address     =     info_json["networks"][network_id]
['address']

print('Contract address: ' + contract_address)
```

Дальнейшие действия скрипта call_contract.py полностью аналогичны действиям, выполняемым в скрипте call_set.py (листинг 11.2.), описанном в предыдущем разделе. Создается провайдер соединения с локальным узлом Ethereum, создается объект контракта, и вызываются его функции.

Вот что вы увидите на консоли при запуске скрипта:

```
$ python3 call_contract.py HelloSol 5777
HelloSol
Contract address: 0xCe080D343895757ca9123D2c9953e243fB61a4fc
getValue: 777888
getString: Test string 888
```

Итоги урока

На 11-м уроке вы освоили работу с узлами Ethereum с помощью программ Python и фреймворка Web3.py.

Вы научились устанавливать фреймворк Web3.py, компилировать и публиковать смарт-контракты, вызывать функции смарт-контрактов, а также использовать Web3.py совместно с Truffle.

Урок 12. Оракулы

Цель урока: научиться обмениваться данными между смарт-контрактами и реальным миром при помощи оракулов.

Практические задания: создать оракул и смарт-контракт, получающий актуальный курс обмена USD на рубли с сайта ЦБ РФ.

Как вы, наверное, уже знаете, виртуальная машина Ethereum и, соответственно, контракты Solidity не имеют никакого доступа к данным реального мира. Вся информация, такая как данные, полученные с Web-сайтов, устройств интернета вещей IoT, различных приборов и датчиков, может быть добавлена в сеть Ethereum только при помощи внешних программ и API виртуальной машины Ethereum. Для этого можно использовать различные фреймворки, такие, например, как Web3.js.

Может ли смарт-контракт доверять данным из внешнего мира

Распределенная сеть Ethereum призвана решать проблемы взаимодействия не доверяющих друг другу сторон. Однако ввод в эту сеть данных реального мира в плане доверия представляет собой узкое место.

Почему это так?

Главным образом потому, что данные из реального мира (курсы валют, метеорологические данные, ставки и т.п.) поступают через ограниченное количество каналов. Каждый из этих каналов кем-то контролируется, а также может быть скомпрометирован злоумышленниками с целью искажения данных или задержки их передачи.

Если вам, например, нужно узнать биржевые курсы ценных бумаг, то какой из источников использовать? Где взять самый точный прогноз погоды или курс обмена валюты?

Второй момент: вам нужно создать некоторое программное обеспечение (ПО), которое будет получать информацию с этого сайта и сохранять ее в смарт-контракте, используя фреймворк web3.js или какой-либо другой.

Тут встает вопрос доверия к ПО и серверному оборудованию, на котором такое ПО работает. Ведь в ПО могут быть ошибки, а сервер может быть «взломан» и захвачен злоумышленниками.

Выходом из данной ситуации, возможно, будет использование нескольких независимых источников данных, доступ к которым осуществляется разным ПО, работающим на разных серверах. Получив данные из нескольких источников, смарт-контракт может отбросить заведомо ложные данные, пользуясь каким-либо алгоритмом поиска консенсуса.

Оракулы как информационные посредники блокчейна

Для того чтобы передать данные из реального мира в блокчейн и обратно, используются так называемые оракулы (oracles). Это алгоритмы, созданные для обмена данными между смарт-контрактами и реальным миром.

Для обмена данными со смарт-контрактами используются следующие компоненты:

• источник данных;

• код для предоставления данных из источника;

• оракул, реализующий алгоритм преобразования данных в формат, понятный смарт-контрактам.

Рассмотрим компоненты на конкретных примерах.

Источник данных

Например, пусть нам нужно создать смарт-контракт, получающий актуальный курс обмена USD на рубли. В качестве достоверной информации мы будем использовать сайт ЦБ РФ.

По адресу https://www.cbr-xml-daily.ru/ опубликована информация, которая поможет нам получить обменные курсы фиатных валют. Воспользуемся способом, который возвращает данные в формате JSON, т.к. этот формат очень простой в использовании.

Если открыть в браузере URL https://www.cbr-xml-daily.ru/daily_json.js, то можно увидеть обменные курсы валют, доступные с сайта ЦБ РФ. Ниже мы привели фрагменты для американского доллара и евро:

```
    {
       "Date": "2019-03-30T11:30:00+03:00",
       "PreviousDate": "2019-03-29T11:30:00+03:00",
              "PreviousURL":   "\/\/www.cbr-xml-daily.ru\/archive
\/2019\/03\/29\/daily_json.js",
       "Timestamp": "2019-03-29T23:00:00+03:00",
       "Valute": {
    ...
    "USD": {
                "ID": "R01235",
                "NumCode": "840",
                "CharCode": "USD",
                "Nominal": 1,
                "Name": "Доллар США",
                "Value": 64.7347,
                "Previous": 64.8012
            },
    "EUR": {
                "ID": "R01239",
                "NumCode": "978",
                "CharCode": "EUR",
                "Nominal": 1,
                "Name": "Евро",
                "Value": 72.723,
```

```
            "Previous": 72.8884
        },
...
}
```

Обратите внимание на поле Timestamp, содержащее время и дату, когда указанные обменные курсы будут актуальны.

Насколько этот источник данных достоверный?

Очевидно, что если вы используете данные в тех случаях, когда обмен валют должен проводиться по курсу ЦБ РФ, то этот источник является достоверным. Если, конечно, сайт не взломан и он возвращает действительно актуальную информацию.

Если же вы будете менять, например, наличную валюту в отделениях банков или обменных пунктах, то там обменные курсы могут заметно отличаться от курсов ЦБ РФ.

Существует ряд коммерческих сервисов, предлагающих решение упомянутой выше проблемы как раз с применением независимых источников информации. Среди них самый известный – Oraclize. Но на этом уроке мы расскажем, как можно организовать ввод внешних данных в смарт-контракты своими силами.

Код для представления данных из источника

На этом уроке мы будем создавать оракул, работающий со смарт-контрактами про помощи скриптов JavaScript, запущенных под управлением Node.js. Эти скрипты будут обращаться к функциям смарт-контрактов при помощи фреймворка Web3.js версии 1.0.47. Напомним, что это бета-версия, релиз 1.0.x на момент создания данного руководства еще не был готов.

Ниже в листинге 12.1. мы привели код скрипта get_usd_rate.js. Он загружает данные курса валют с сайта ЦБ РФ и преобразует их в формат, который можно использовать для смарт-контрактов Solidity.

Листинг 12.1. Скрипт get_usd_rate.js

```
let request = require('request')
let cbr_url = "https://www.cbr-xml-daily.ru/daily_json.js"
request(cbr_url, function(error, response, body) {
 if(error)
   console.log("Request error: " + error)
 if(response.statusCode != 200)
          console.log("Error.   CBR   responce   code:   "
+ response.statusCode)

 let rates = JSON.parse(body)
 let rates_timestamp = rates.Timestamp

 let unix_timestump = new Date(rates_timestamp)
 unix_timestump = unix_timestump.getTime()

   console.log('Rates   Timestamp   from   CBR   site:   '
+ rates_timestamp)
 console.log('Unix Rates Timestump: ' + unix_timestump)
```

```
 let str_timestump = new Date(unix_timestump)
 console.log('Formatted Timestump: ' + str_timestump)

 let USD_rate = rates.Valute.USD.Value

 console.log('CBR USD Exchange Rate from CBR site: ' + USD_rate)
 USD_rate = (USD_rate * 10000).toFixed()
 console.log('CBR USD Exchange Rate * 10000 =  ' + USD_rate)
})
```

Чтобы загрузить данные с сайта, мы используем модуль request. Одноименный метод модуля возвращает промис, функции обработчика которого передается код ошибки error, данные кода ответа response, а также загруженные данные body.

При обработке мы проверяем, нет ли ошибки и возвращается ли код состояния, равный 200 (что должно быть при правильной работе).

Далее с помощью метода JSON.parse мы разбираем данные, полученные с сайта CBR, а затем извлекаем отметку времени и обменный курс доллара:

```
let rates = JSON.parse(body)
let rates_timestamp = rates.Timestamp
…

let USD_rate = rates.Valute.USD.Value
```

Здесь мы получаем отметку времени в виде текстовой строки, а обменный курс доллара – в виде дробного числа с точностью до сотой доли копейки, как в исходных данных:

```
"Timestamp": "2019-03-29T23:00:00+03:00"
"Value": 64.7347
```

И вот теперь мы сталкиваемся с необходимостью преобразования типов полученных данных перед их сохранением в смарт-контракте Solidity.

Прежде всего на момент создания этого руководства в Solidity версии 5.0.x не было возможности работать с дробными числами.

Что же касается текстовых строк, то их, конечно, можно сохранять в контракте, однако с датой, представленной в таком виде, работать будет крайне неудобно. Кроме того, хранение строк в блокчейне требует больших затрат по сравнению с хранением целых чисел.

Таким образом, было бы хорошо преобразовать дату, время и обменный курс в целые числа.

Для хранения даты и времени мы будем использовать формат Unix Time. Вы, наверное, знаете, что в этом формате время представлено в виде целого числа и представляет собой количество секунд, прошедших с 00:00:00 1 января 1970 года по шкале всемирного времени UTC.

Чтобы преобразовать исходную текстовую строку в формат времени Unix Time, мы используем встроенный объект Date и его метод getTime():

```
let unix_timestump = new Date(rates_timestamp)
unix_timestump = unix_timestump.getTime()
```

Обратное преобразование также возможно:

```
let str_timestump = new Date(unix_timestump)
console.log('Formatted Timestump: ' + str_timestump)
```

Что же касается перевода дробного обменного курса доллара в целое число, то мы это делаем очень просто. Прежде всего мы умножаем курс на 10000 (т.к. он представлен в сотых долях копейки), а затем преобразуем в целое значение с помощью метода toFixed:

```
USD_rate = (USD_rate * 10000).toFixed()
```

Ниже мы показали консольный вывод скрипта get_usd_rate.js:

```
$ node get_usd_rate.js
Rates Timestamp from CBR site: 2019-03-30T15:00:00+03:00
Unix Rates Timestump: 1553947200000
Formatted Timestump: Sat Mar 30 2019 12:00:00 GMT+0000
(Coordinated Universal Time)
CBR USD Exchange Rate from CBR site: 64.7347
CBR USD Exchange Rate * 10000 =  647347
```

Здесь скрипт вывел на консоль сначала исходное значение отметки времени, затем преобразованное в формат Unix Time и, наконец, преобразованное обратно в текстовую строку.

Далее скрипт выводит исходный дробный обменный курс доллара и его целочисленный эквивалент.

Оракул для записи обменного курса в блокчейн

Итак, мы подобрали источник данных для получения обменного курса фиатных валют, а также написали скрипт, который может преобразовать эти данные в формат, пригодный для записи в смарт-контракт.

Теперь нам потребуются средства для сохранения полученного обменного курса доллара в блокчейне.

Мы создадим смарт-контракт USDRateOracle.sol, а также два скрипта start_oracle.js и call_oracle.js.

Смарт-контракт USDRateOracle.sol будет хранить обменный курс доллара, а также инициировать процесс его обновления.

Скрипт start_oracle.js займется обновлением курса доллара и будет играть роль шлюза, передающего данные обменного курса из реального мира (с сайта ЦБ РФ) в память смарт-контракта USDRateOracle.sol.

И наконец, скрипт call_oracle.js будет инициировать обновление значения текущего обменного курса доллара в памяти смарт-контракта, а также показывать обновленное значение и временной штамп на консоли.

Контракт USDRateOracle

Смарт-контракт USDRateOracle нашего оракула довольно простой и представлен в листинге 12.2.

Листинг 12.2. Смарт-контракт USDRateOracle.sol

```
pragma solidity ^0.5.0;
contract USDRateOracle {
 uint public USD_rate;
 string public timestamp;
 uint public unix_timestump;

 event RateUpdate(address sender);
 event UpdatedRate(uint req_rate);

 constructor() public {
     USD_rate = 0;
     timestamp = "";
     unix_timestump = 0;
 }
 function requestNewRate() public {
     emit RateUpdate(msg.sender);
 }
 function saveNewRate(uint req_rate, uint uix_timestump, string
memory tstamp) public {
     timestamp = tstamp;
     unix_timestump = uix_timestump;
     USD_rate = req_rate;
     emit UpdatedRate(req_rate);
 }
 function getRate() public view returns( uint, uint, string
memory) {
     return(USD_rate, unix_timestump, timestamp);
 }
}
```

Несмотря на простоту, в нем реализована обработка событий (events), чем мы на наших уроках до сих пор не занимались.

Конструктор нашего смарт-контракта обнуляет значения полей USD_rate и unix_timestump, хранящих текущий курс обмена доллара и временную отметку соответственно. Также конструктор записывает пустую текстовую строку в поле timestamp, где хранится оригинальное значение временной отметки, полученное нами с сайта CBR.

Функция getRate не делает ничего необычного. Она просто возвращает текущее значение только что перечисленных полей.

Для обновления обменного курса доллара и временной отметки, хранящейся в смарт-контракте, мы определили два события RateUpdate и UpdatedRate:

```
event RateUpdate(address sender);
```

```
event UpdatedRate(uint req_rate);
```

Первое из них возникает, когда нужно обновить сохраненные значения, а второе – когда обновление завершено.

Также в нашем смарт-контракте определены функции requestNewRate и saveNewRate, способные создавать перечисленные выше события.

Функция requestNewRate создает событие RateUpdate с помощью ключевого слова emit:

```
function requestNewRate() public {
   emit RateUpdate(msg.sender);
}
```

Что же касается функции saveNewRate, то она вызывается уже после того, как будут получены новые значения обменного курса и временной отметки. Ее задача – сохранить эти новые значения в полях смарт-контракта, после чего вызывать событие UpdatedRate, сигнализирующее о том, что сохраненные значения обновлены:

```
function saveNewRate(uint req_rate, uint uix_timestump, string
memory tstamp) public {
   timestamp = tstamp;
   unix_timestump = uix_timestump;
   USD_rate = req_rate;
   emit UpdatedRate(req_rate);
}
```

Как это все работает?

Сначала нужно запустить скрипт start_oracle.js. После запуска он переходит в ожидание события RateUpdate, т.е. ждет, когда нужно будет получить новый курс обмена доллара и временную отметку.

Далее в отдельном окне нужно запустить скрипт call_oracle.js. Этот скрипт переходит в ожидание события UpdatedRate (курс и временная отметка обновлены), а затем вызывает функцию контракта requestNewRate.

Именно функция requestNewRate вызывает появление события RateUpdate, которого ждет скрипт start_oracle.js. Обработка этого события сводится к получению от сайта CBR новых данных и обновлению соответствующих полей смарт-контракта с помощью метода saveNewRate.

Далее метод saveNewRate создает событие UpdatedRate, говорящее о том, что новые значения получены и сохранены в контракте. Событие UpdatedRate, в свою очередь, ожидает скрипт call_oracle.js. Получив его, этот скрипт отображает на консоли новые данные.

Далее мы расскажем о скриптах start_oracle.js и call_oracle.js подробнее.

Обновление обменного курса в смарт-контракте

Как мы уже сказали, скрипт start_oracle.js (листинг 12.3.) взаимодействует со смарт-контрактом USDRateOracle.sol (листинг 12.2.), обеспечивая обновление в памяти контракта обменного курса доллара и временной отметки.

Листинг 12.3. Скрипт start_oracle.js

```
// node start_oracle.js USDRateOracle 5777

let Web3 = require('web3')
let fs = require('fs')
let request = require('request')

var contract_name = process.argv[2];
var network_id = process.argv[3];
var unlock_password = process.argv[4];
console.log('Contract script: ' + contract_name);

var path = require('path');
var contractJSON = require(path.join(__dirname, 'build/
contracts/' + contract_name + '.json'));
var decoded = JSON.parse(JSON.stringify(contractJSON.networks,
undefined, 2));
var contract_address = decoded[network_id].address;
var abi = contractJSON.abi;
console.log('contract_address: ' + contract_address);

const web3 = new Web3(new
Web3.providers.WebsocketProvider('ws://127.0.0.1:9545'))

var version = web3.version;
console.log('Web3 version: ' + version);

var myContract = new web3.eth.Contract(abi, contract_address);
var account

web3.eth.getAccounts()
.then(accList => {
 return accList;
})
.then(function (accounts) {
 account = accounts[0];
 console.log('Account: ' + account)

   console.log("Waiting for events on contract "
+ contract_address + ' …')
 startEventListener(contract_address, account)
```

```
    }, function(err) {
       console.log('Error: ' + err)
    })
    .catch(function (error) {
     console.error(error);
    });

    function startEventListener(address, account) {
       myContract.events.RateUpdate(
    //      myContract.events.RateUpdate({fromBlock: 0, toBlock:
'latest'},
         function(error, event){
           console.log(">>> Event: " + event.event) })
           .on('data', (event) => {
               console.log('- >>> Event RateUpdate fired')
    //                    console.log("RateUpdate event data: "
+ JSON.stringify(event, undefined, 2))
               RateUpdateHandler(address, account)
           })
           .on('changed', (event) => {
               console.log(' > changed')
               console.log(event)
           })
           .on('error', console.error);
    }

    function RateUpdateHandler(address, account) {

    let cbr_url = "https://www.cbr-xml-daily.ru/daily_json.js"
     request(cbr_url, function(error, response, body) {
       if(error)
         console.log("Request error: " + error)
       if(response.statusCode != 200)
                 console.log("Error! CBR responce code: "
+ response.statusCode)

       let rates = JSON.parse(body)
       let rates_timestamp = rates.Timestamp
       let USD_rate = rates.Valute.USD.Value

       let unix_timestump = new Date(rates_timestamp)
       unix_timestump = unix_timestump.getTime()

      console.log('Rates Timestamp from CBR: ' + rates_timestamp)
          console.log('Rates Timestamp Unix Time format: '
+ unix_timestump)

       console.log('USD Exchange Rate from CBR: ' + USD_rate)
       USD_rate = (USD_rate * 10000).toFixed()
```

```
      console.log('CBR USD Exchange Rate, fixed uint: ' + USD_rate)

        myContract.methods.saveNewRate(USD_rate, unix_timestump,
rates_timestamp).send({from: account, gasPrice: 18000000000, gas:
5000000})
       .once('transactionHash', (hash) => {
         console.log('hash: ' + hash);
       })
       .on('confirmation', (confNumber) => {
         console.log('confNumber: ' + confNumber);
       })
       .on('receipt', (receipt) => {
                                      var      ret_result      =
receipt.events.UpdatedRate.returnValues[0]
         console.log('Receipt result: ' + ret_result)
    //        console.log('Receipt: ' + receipt)

         if (USD_rate == ret_result)
             console.log(" > Successful update")
       })
       .on('error', console.error);
     })
    }
```

Прежде чем запускать этот скрипт, откройте консоль, войдите в каталог проекта и запустите Truffle с параметром develop:

```
$ cd HelloSol/
$ truffle develop
```

Далее запустите компиляцию и миграцию смарт-контракта:

```
truffle(develop)> compile –all
truffle(develop)> migrate –reset
```

При запуске скрипта start_oracle.js нужно передать ему имя смарт-контракта USDRateOracle, номер сети 5777 и пароль для разблокировки аккаунта (пустая строка для отладочной сети Truffle).

Скрипт извлечет из артефактов Truffle адрес контракта для сети 5777 (используется Truffle при миграции), а также интерфейс ABI смарт-контракта:

```
var contract_name = process.argv[2];
var network_id = process.argv[3];
var unlock_password = process.argv[4];
console.log('Contract script: ' + contract_name);
var path = require('path');
var  contractJSON  =  require(path.join(__dirname,  'build/
contracts/' + contract_name + '.json'));
```

```
    var decoded = JSON.parse(JSON.stringify(contractJSON.networks,
undefined, 2));
    var contract_address = decoded[network_id].address;
    var abi = contractJSON.abi;
    console.log('contract_address: ' +  contract_address);
```

Все это мы уже проделывали на предыдущих уроках.

Использование провайдера Web Socket

Очень важно, что, в отличие от примеров из предыдущих уроков, для подключения к узлу Ethereum мы используем не HTTP, а **протокол Web Socket**:

```
    const          web3          =          new          Web3(new
Web3.providers.WebsocketProvider('ws://127.0.0.1:9545'))
```

Дело в том, что наш скрипт обрабатывает события, создаваемые смарт-контрактом. Обработка таких событий возможна только в том случае, если мы используем протоколы Web Socket или RPC. В нашем примере для создания соединения мы используем провайдер Web3.providers.WebsocketProvider.

Так как наш пример работает с бета-версией Web3.js, выведем на консоль версию Web3:

```
    var version = web3.version;
    console.log('Web3 version: ' + version);
```

Мы отлаживали скрипт в версии 1.0.0-beta.48.

Ожидание события RateUpdate

Показав версию Web3 на консоли, наш скрипт получает список аккаунтов и вызывает функцию startEventListener, ожидающую запрос на обновление обменного курса с сайта ЦБ РФ:

```
    web3.eth.getAccounts()
    .then(accList => {
     return accList;
    })
    .then(function (accounts) {
     account = accounts[0];
     console.log('Account: ' + account)
        console.log("Waiting   for   events   on   contract   "
+ contract_address + ' …')
     startEventListener(contract_address, account)
    }, function(err) {
       console.log('Error: ' + err)
    })
    .catch(function (error) {
     console.error(error);
    });
```

Функции startEventListener передают адрес опубликованного смарт-контракта и основной аккаунт узла Ethereum.

Получив управление, эта функция подписывается на событие RateUpdate, ожидая, пока это событие произойдет:

```
function startEventListener(address, account) {
   myContract.events.RateUpdate(
     function(error, event){
       console.log(">>> Event: " + event.event) })
       .on('data', (event) => {
           console.log('- >>> Event RateUpdate fired')
           RateUpdateHandler(address, account)
       })
       .on('changed', (event) => {
           console.log(' > changed')
           console.log(event)
       })
       .on('error', console.error);
}
```

Напомним, что появление события RateUpdate означает, что смарт-контракт должен обновить хранящиеся в его полях данные обменного курса доллара.

Заметим, что функция startEventListener не вернет управление, а перейдет в режим ожидания события. На консоли вы увидите соответствующее сообщение, которое скрипт выведет перед вызовом этой функции:

```
$ node start_oracle.js USDRateOracle 5777
Contract script: USDRateOracle
contract_address: 0x926cfD5a5E65AA88E8CEAc4c2398a9E4d92f4A30
Web3 version: 1.0.0-beta.48
Account: 0xe5561B146B256CdE78f28C8e79783E95112764FA
Waiting for events on contract
0x926cfD5a5E65AA88E8CEAc4c2398a9E4d92f4A30 …
```

Событие RateUpdate будет инициировано скриптом call_oracle.js, который мы рассмотрим в следующем разделе.

Обработка события RateUpdate

Когда произойдет событие RateUpdate, в функции startEventListener получит управление блок .on('data', (event):

```
.on('data', (event) => {
    console.log('- >>> Event RateUpdate fired')
    RateUpdateHandler(address, account)
})
```

Здесь вызывается функция RateUpdateHandler. Ее задачей является получение нового значения обменного курса, отметки времени, а также обновление этих данных в памяти смарт-контракта.

Функции RateUpdateHandler передаются адрес контракта и основной аккаунт, необходимый для вызова метода контракта, обновляющего данные.

Получив управление, прежде всего эта функция загружает данные обменных курсов валют с сайта ЦБ РФ и преобразует их в формат, пригодный для сохранения в памяти смарт-контракта:

```
    let cbr_url = "https://www.cbr-xml-daily.ru/daily_json.js"
     request(cbr_url, function(error, response, body) {
       if(error)
         console.log("Request error: " + error)
       if(response.statusCode != 200)
                 console.log("Error!   CBR   responce   code:   "
+ response.statusCode)
       let rates = JSON.parse(body)
       let rates_timestamp = rates.Timestamp
       let USD_rate = rates.Valute.USD.Value
       let unix_timestump = new Date(rates_timestamp)
       unix_timestump = unix_timestump.getTime()
      console.log('Rates Timestamp from CBR: ' + rates_timestamp)
          console.log('Rates  Timestamp  Unix  Time  format:  '
+ unix_timestump)
       console.log('USD Exchange Rate from CBR: ' + USD_rate)
       USD_rate = (USD_rate * 10000).toFixed()
     console.log('CBR USD Exchange Rate, fixed uint: ' + USD_rate)
```

Процесс преобразования мы уже описывали в начале этого урока.

Для сохранения полученных и преобразованных данных мы вызываем функцию смарт-контракта saveNewRate:

```
    myContract.methods.saveNewRate(USD_rate,      unix_timestump,
rates_timestamp).send({from: account, gasPrice: 18000000000, gas:
5000000})
    .once('transactionHash', (hash) => {
     console.log('hash: ' + hash);
    })
    .on('confirmation', (confNumber) => {
     console.log('confNumber: ' + confNumber);
    })
    .on('receipt', (receipt) => {
     var ret_result = receipt.events.UpdatedRate.returnValues[0]
     console.log('Receipt result: ' + ret_result)
     if (USD_rate == ret_result)
         console.log(" > Successful update")
    })
    .on('error', console.error);
```

Так как эта функция изменяет состояние блокчейна, для ее вызова нужно инициировать транзакцию методом send.

После получения чека транзакции в блоке .on('receipt', (receipt) мы извлекаем из чека новый обменный курс доллара, и если он совпадает с тем, что мы сохраняем в контракте, выводим на консоль сообщение об успешном обновлении.

Функция RateUpdateHandler также выводит на консоль новое значение обменного курса и отметку времени, загруженные с сайта ЦБ РФ:

```
    -- >>> Event RateUpdate fired
    Rates Timestamp from CBR: 2019-03-30T16:00:00+03:00
    Rates Timestamp Unix Time format: 1553950800000
    USD Exchange Rate from CBR: 64.7347
    CBR USD Exchange Rate, fixed uint: 647347
    hash:
0x20bfd45153f4e8a70c9c1f2ee26e866893c2866acfa48e202cf01aaa436d8544
    Receipt result: 647347
    > Successful update
```

Итак, мы рассказали о том, как смарт-контракт USDRateOracle взаимодействует со скриптом start_oracle.js. Скрипт ожидает поступления от смарт-контракта события RateUpdate и при его появлении скачивает с сайта ЦБ РФ новые данные и обновляет сохраненные значения в смарт-контракте.

Но как возникает событие RateUpdate?

Это событие создает скрипт call_oracle.js, о котором мы расскажем в следующем разделе урока.

Инициирование обновления данных в смарт-контракте

Исходный текст скрипта call_oracle.js вы найдете в листинге 12.4.

Листинг 12.4. Скрипт call_oracle.js

```
// node call_oracle.js USDRateOracle 5777
let Web3 = require('web3')
let fs = require('fs')
let request = require('request')

var contract_name = process.argv[2];
var network_id = process.argv[3];
var unlock_password = process.argv[4];

console.log('Contract script: ' + contract_name);

var path = require('path');
var contractJSON = require(path.join(__dirname, 'build/
contracts/' + contract_name + '.json'));
var decoded = JSON.parse(JSON.stringify(contractJSON.networks,
undefined, 2));
var contract_address = decoded[network_id].address;
var abi = contractJSON.abi;
console.log('contract_address: ' + contract_address);

const web3 = new Web3(new
Web3.providers.WebsocketProvider('ws://127.0.0.1:9545'))
var version = web3.version;
console.log('Web3 version: ' + version);

let myContract = new web3.eth.Contract(abi, contract_address);

web3.eth.getAccounts()
.then(accList => {
 return accList;
})
.then(function (accounts) {
 account = accounts[0];

//  myContract.events.UpdatedRate({fromBlock: 0, toBlock:
'latest'}, function(error, event){ console.log(">>> " + event) })
    myContract.events.UpdatedRate(function(error, event)
{ console.log(">>> " + event) })
   .on('data', (log) => {
       console.log('- >>> Event UpdatedRate fired >>>')
//                 console.log("UpdatedRate event data: "
+ JSON.stringify(log, undefined, 2))
```

```
                          myContract.methods.getRate().call({from:
contract_address}, (error, result) =>
         {
           if(!error){
           console.log('getRate on UpdatedRate result: ' + result);
             console.log(JSON.stringify(result, undefined, 2));
             console.log('USD Rate: ' + result[0]/10000)

             let ux_time = parseInt(result[1])
             let str_timestump = new Date(ux_time)
             console.log('Formatted Timestump: ' + str_timestump)
             process.exit();
           } else{
              console.log(error);
           }
         });
       })
       .on('changed', (event) => {
           console.log(' > changed')
           console.log(event)
       })
       .on('error', console.error);

        myContract.methods.requestNewRate().send({from: account,
gas: 100000, gasPrice: "2000000000"})
       .once('transactionHash', (hash) => {
         console.log('hash: ' + hash);
       })
       .on('confirmation', (confNumber) => {
         console.log('confNumber: ' + confNumber);
       })
       .on('receipt', (receipt) => {
     //    console.log(JSON.stringify(receipt, undefined, 2));
       })
       .on('error', console.error);
     })
     .then(function (receipt) {
    });
```

Аналогично start_oracle.js, скрипт call_oracle.js получает в качестве параметров имя контракта, номер сети и пароль для разблокировки аккаунта.

После запуска скрипт извлекает из артефактов Truffle адрес контракта, а также интерфейс ABI.

Получив эти данные, скрипт создает соединение с вашим тестовым узлом Ethereum с помощью провайдера Web3.providers.WebsocketProvider. Это необходимо для обработки события UpdatedRate, возникающего после завершения обновления обменного курса валют и временной отметки в смарт-контракте.

После создания объекта контракта скрипт call_oracle.js подписывается на событие UpdatedRate:

```
    myContract.events.UpdatedRate(function(error,          event)
{ console.log(">>> " + event) })
     .on('data', (log) => {
       console.log('- >>> Event UpdatedRate fired >>>')
      myContract.methods.getRate().call({from: contract_address},
(error, result) =>
       {
         if(!error){
          console.log('getRate on UpdatedRate result: ' + result);
           console.log(JSON.stringify(result, undefined, 2));
           console.log('USD Rate: ' + result[0]/10000)

           let ux_time = parseInt(result[1])
           let str_timestump = new Date(ux_time)
           console.log('Formatted Timestump: ' + str_timestump)

           process.exit();

         } else{
            console.log(error);
         }
     });
    })
```

Напомним, что это событие возникает после того, как наш смарт-контракт обновит обменный курс доллара и отметку времени.

Обработчик события UpdatedRate получает новые значения с помощью функции контракта getRate, вызывая ее при помощи метода call. Транзакция не создается, т.к. функция getRate не изменяет состояние сети.

После вывода полученных результатов на консоль скрипт завершает работу Node.js при помощи метода process.exit.

Но когда же возникнет событие UpdatedRate?

Сразу после того, как скрипт call_oracle.js установит обработчик этого события, он его и инициирует. Для этого создается транзакция для вызова метода requestNewRate:

```
    myContract.methods.requestNewRate().send({from: account, gas:
100000, gasPrice: "2000000000"})
    .once('transactionHash', (hash) => {
     console.log('hash: ' + hash);
    })
    .on('confirmation', (confNumber) => {
     console.log('confNumber: ' + confNumber);
    })
    .on('receipt', (receipt) => {
    // console.log(JSON.stringify(receipt, undefined, 2));
    })
    .on('error', console.error);
```

Таким образом, скрипт call_oracle.js вначале готовится обработать событие UpdatedRate, а потом вызывает метод requestNewRate нашего смарт-контракта, обновляющий курс доллара и отметку времени в памяти смарт-контракта.

Для чего мы используем событие UpdatedRate? Ведь мы могли бы просто запустить обновление данных в смарт-контракте, вызвав функцию requestNewRate, а затем получить новое значение при помощи метода getRate?

Дело в том, что на выполнение транзакции требуется определенное время. Если не дожидаться события UpdatedRate, то функция getRate вернет старые значения обменного курса и старую метку времени.

Скрипт call_oracle.js при обработке события UpdatedRate выводит на консоль данные, полученные от функции getRate, в исходном и преобразованном виде:

```
    $ node call_oracle.js USDRateOracle 5777 ""
    Contract script: USDRateOracle
    contract_address: 0x926cfD5a5E65AA88E8CEAc4c2398a9E4d92f4A30
    Web3 version: 1.0.0-beta.48
    hash:
0x072f63a2270107fc5d75c244da60aa222617698472c3c1efffa83b066cff7d7
    -- >>> Event UpdatedRate fired >>>
    getRate on UpdatedRate result: [object Object]
    {
     "0": "647347",
     "1": "1553950800000",
     "2": "2019-03-30T16:00:00+03:00"
    }
    USD Rate: 64.7347
    Formatted Timestump: Sat Mar 30 2019 13:00:00 GMT+0000
(Coordinated Universal Time)
```

Итоги урока

На 12-м уроке вы узнали, каким способом данные из реального мира могут попадать в память смарт-контрактов. Вы создали собственный оракул, способный получать и сохранять в смарт-контракте текущий курс обмена USD на рубли с сайта ЦБ РФ.

Вы также научились инициировать и обрабатывать события в смарт-контрактах Solidity.