**Краткое описание**

**Solc**

Компилятор языка Solidity, который используется для разработки смарт-контрактов Ethereum. Название — сокращение от Solidity compiler.

Вообще говоря, существует два различных компилятора. В [Solc](https://solidity.readthedocs.io/en/v0.4.21/installing-solidity.html) языком программирования выбран C++. Чаще всего ноды используют его в качестве встроенного компилятора, но предлагается также автономный модуль solc. compile () для офлайн-разработки без необходимости подключения к внешней ноде.

Также есть Solc-js, который использует промежуточный язык Emscripten для кросс-компиляции исходного кода Solc C++ и JavaScript. Он работает чуть медленнее и имеет несколько другой интерфейс.

**Установка компилятора Solidity**

**Управление версиями**

Версии Solidity следуют семантическому управлению версиями, и в дополнение к выпускам также доступны сборки для ночной разработки. Работа ночных сборок не гарантируется, и, несмотря на все усилия, они могут содержать недокументированные и / или сломанные изменения. Мы рекомендуем использовать последнюю версию. Установщики пакетов ниже будут использовать последнюю версию.

Установка компилятора Solidity Compiler(далее solc) может быть произведена несколькими способами. Ее можно произвести через NPM если вы собираетесь работать через Node.js. Так же вы можете установить компилятор, через docker, если вы используете docker-образы сборок Solidity доступны с использованием образа solc. Ну и третий способ установки через бинарные пакеты. Далее мы кратко рассмотрим эти три способа установки компилятора Solc:

**npm / Node.js**

Используйте npm для удобного и портативного способа установки solcjs, компилятора Solidity. Программа solcjs имеет меньше возможностей, чем способы доступа к компилятору, описанные ниже на этой странице. Документация по использованию компилятора командной строки предполагает, что вы используете полнофункциональный компилятор solc. Использование solcjs задокументировано в его собственном репозитории.

Примечание. *Проект solc-js является производным от Solc C ++ с использованием Emscripten, что означает, что оба используют один и тот же исходный код компилятора. solc-js можно использовать напрямую в проектах.*

npm install -g solc

Примечание(*Исполняемый файл командной строки называется solcjs.Параметры командной строки solcjs несовместимы с solc и инструментами (такими как geth), ожидая, что поведение solc не будет работать с solcjs.)*

**Докер**

Используйте стабильный тег для последней выпущенной версии и “*nightly”* для потенциально нестабильных изменений в ветке разработки.

Образ Docker запускает исполняемый файл компилятора, поэтому вы можете передать ему все аргументы компилятора. Например, приведенная ниже команда извлекает стабильную версию образа solc (если у вас ее еще нет) и запускает ее в новом контейнере, передавая аргумент –help:

docker run ethereum / solc: stable --help

Так же если вы хотите установить конкретную версию solc вы можете вместо команды stable прописать нужную для вас версию например, для выпуска 0.5.4:

docker run ethereum / solc: 0.4.1 --help

Чтобы использовать образ Docker для компиляции файлов Solidity на главном компьютере, смонтируйте локальную папку для ввода и вывода и укажите контракт для компиляции. Например.

docker run -v / local / path: / sources ethereum / solc: stable -o / sources / output --abi --bin /sources/Contract.sol

Вы также можете использовать стандартный интерфейс JSON (который рекомендуется при использовании компилятора с инструментами). При использовании этого интерфейса нет необходимости монтировать какие-либо каталоги.

docker run ethereum/solc:stable --standard-json < input.json > output.json

**Бинарные пакеты**

Бинарные пакеты Solidity доступны в репозитории по ссылке https://github.com/ethereum/solidity/releases.

Также есть PPA для Ubuntu, вы можете получить последнюю стабильную версию, используя следующие команды:

sudo add-apt-repository ppa: ethereum / ethereum

sudo apt-get update

sudo apt-get install solc

Nightly version можно установить с помощью следующих команд:

sudo add-apt-repository ppa: ethereum / ethereum

sudo add-apt-repository ppa: ethereum / ethereum-dev

sudo apt-get update

sudo apt-get install solc

Компилятор Solidity распространяется через Homebrew как версия для сборки из исходного кода.

brew upgrade

brew tap ethereum/ethereum

brew install solidity

Чтобы установить самую последнюю версию Solidity 0.4.x / 0.5.x, вы также можете использовать brew install solidity @ 4 и brew install solidity @ 5 соответственно.

В Gentoo Linux есть оверлей Ethereum, содержащий пакет solidity. После настройки оверлея solc можно установить в архитектуре x86\_64 следующим образом:

Emerge dev-lang / solidity

**Сборка из исходного кода**

Предварительные требования - все операционные системы

Ниже приведены зависимости для всех сборок Solidity:

* CMake (версия 3.9+) - Генератор файлов кроссплатформенной сборки.
* Boost (версия 1.65+) - библиотеки C ++.
* Git - Инструмент командной строки для получения исходного кода.
* z3 (версия 4.6+, необязательно) - Для использования с программой проверки SMT.
* cvc4 (необязательно) - Для использования с программой проверки SMT.

Примечание(*Версии Solidity до 0.5.10 могут некорректно связываться с Boost версиями 1.70+. Возможный обходной путь - временно переименовать <путь для ускоренной установки> /lib/cmake/Boost-1.70.0 перед запуском команды cmake для настройки твердости.* *Начиная с версии 0.5.10 связывание с Boost 1.70+ должно работать без ручного вмешательства.)*

**Минимальные версии компилятора**

Следующие компиляторы C ++ и их минимальные версии могут создавать базу кода Solidity:

* GCC, версия 5+
* Clang, версия 3.4+
* MSVC, версия 2019+

**Клонировать репозиторий**

Чтобы клонировать исходный код, выполните следующую команду:

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

cd solidity

Если вы хотите помочь в разработке Solidity, вам следует создать вилку Solidity и добавить свою личную вилку в качестве второго пульта дистанционного управления:

git remote add personal git@github.com:[username]/solidity.git

**Сборка из командной строки**

Обязательно установите внешние зависимости перед сборкой.

Проект Solidity использует CMake для настройки сборки. Вы можете установить ccache для ускорения повторяющихся сборок. CMake подберет его автоматически. Построение Solidity очень похоже на Linux, macOS и другие Unix:

mkdir build

cd build

cmake .. && make

или даже проще в Linux и macOS, вы можете запустить:

./scripts/build.sh

Предупреждение:Сборки BSD должны работать, но они не проверены командой Solidity.

И для Windows:

mkdir build

cd build

cmake -G "Visual Studio 16 2019 Win64" ..

Если вы хотите использовать версию boost, установленную через ./scripts/install\_deps.ps1, вам дополнительно потребуется передать -DBoost\_DIR = ".. \ deps \ boost \ lib \ cmake \ Boost- \*" и -DCMAKE\_MSVC\_RUNTIME\_LIBRARY = MultiThreaded в качестве аргументов вызова cmake.

Это должно привести к созданию solidity.sln в этом каталоге сборки. Двойной щелчок по этому файлу должен привести к запуску Visual Studio. Мы предлагаем создать конфигурацию Release, но все остальные работают.

В качестве альтернативы вы можете выполнить сборку для Windows из командной строки, например:

cmake --build. --config Release

**Параметры CMake**

Если вам интересно, какие параметры CMake доступны, запустите cmake .. -LH.

**SMT Solvers**

Solidity может быть построен на основе решателей SMT и будет делать это по умолчанию, если они обнаружены в системе. Каждый решатель можно отключить с помощью параметра cmake.

Примечание. В некоторых случаях это также может быть потенциальным обходным путем при сбоях сборки.

Внутри папки сборки вы можете отключить их, поскольку они включены по умолчанию:

# отключает только Z3 SMT Solver.

cmake .. -DUSE\_Z3=OFF

# отключает только CVC4 SMT Solver.

cmake .. -DUSE\_CVC4=OFF

# отключает Z3 и CVC4

cmake .. -DUSE\_CVC4=OFF -DUSE\_Z3=OFF

**Подробно о строке версии**

Строка версии Solidity состоит из четырех частей:

* номер версии
* тег pre-release, обычно устанавливается на development.YYYY.MM.DD или nightly.YYYY.MM.DD
* коммит в формате commit.GITHASH
* платформа, которая имеет произвольное количество элементов, содержащих сведения о платформе и компиляторе

Если есть локальные изменения, фиксация будет иметь после себя .mod.

Эти части объединены в соответствии с требованиями Semver, где тег предварительного выпуска Solidity соответствует предварительному выпуску Semver, а фиксация Solidity и платформа вместе составляют метаданные сборки Semver.

Пример выпуска: 0.4.8+commit.60cc1668.Emscripten.clang.

Пример предварительной версии: 0.4.9-nightly.2017.1.17 + commit.6ecb4aa3.Emscripten.clang

**Использование компилятора**

**Использование компилятора командной строки**

В данном разделе мы будем говорить только об использовании solc команд. Solcjs в данном разделе не будет описан

**Основное использование**

Одна из целей сборки репозитория Solidity - это solc, компилятор командной строки solidity. Использование solc --help дает вам объяснение всех параметров. Компилятор может производить различные выходные данные, начиная от простых двоичных файлов и сборки абстрактного синтаксического дерева (дерева синтаксического анализа) до оценок использования газа. Если вы хотите скомпилировать только один файл, запустите его как solc --bin sourceFile.sol, и он распечатает двоичный файл. Если вы хотите получить некоторые из более продвинутых вариантов вывода solc, вероятно, лучше указать ему выводить все в отдельные файлы с помощью solc -o outputDirectory --bin --ast-json --asm sourceFile.sol.

**Параметры оптимизатора**

Перед развертыванием контракта активируйте оптимизатор при компиляции с использованием solc --optimize --bin sourceFile.sol. По умолчанию оптимизатор оптимизирует контракт, предполагая, что он вызывается 200 раз за время его существования (более конкретно, он предполагает, что каждый код операции выполняется примерно 200 раз). Если вы хотите, чтобы первоначальное развертывание контракта было дешевле, а последующее выполнение функций было более дорогим, установите для него значение --optimize-run = 1. Если вы ожидаете много транзакций и не заботитесь о более высоких затратах на развертывание и размере вывода, установите для --optimize-run большое число. Этот параметр влияет на следующее (это может измениться в будущем):

* размер двоичного поиска в подпрограмме диспетчеризации функций
* способ хранения констант, таких как большие числа или строки

**Переназначение пути**

Компилятор командной строки автоматически считывает импортированные файлы из файловой системы, но также можно обеспечить перенаправление пути с помощью prefix = path следующим образом:

solc github.com/ethereum/dapp-bin/=/usr/local/lib/dapp-bin/ file.sol

По сути, это указывает компилятору искать все, что начинается с github.com/ethereum/dapp-bin/ в /usr / local / lib / dapp-bin. solc не будет читать файлы из файловой системы, которые находятся за пределами целей переназначения и за пределами каталогов, в которых находятся явно указанные исходные файлы, поэтому такие вещи, как import «/ etc / passwd»; работает только если вы добавляете / = / в качестве переназначения.

Пустой префикс переназначения не допускается.

Если есть несколько совпадений из-за переназначения, выбирается тот, у которого есть самый длинный общий префикс.

При доступе к файловой системе для поиска импорта все пути обрабатываются так, как если бы они были полностью определенными путями. Это поведение можно настроить, добавив параметр командной строки --base-path с путем, который будет добавлен перед каждым доступом к файловой системе для импорта. Кроме того, часть, добавленная с помощью --base-path, не будет отображаться в метаданных контракта.

По соображениям безопасности компилятор имеет ограничения, к каким каталогам он может получить доступ. Пути (и их подкаталоги) к исходным файлам, указанные в командной строке, и пути, определенные повторным назначением, разрешены для операторов импорта, но все остальное отклоняется. Дополнительные пути (и их подкаталоги) могут быть разрешены с помощью переключателя --allow-paths / sample / path, / another / sample / path.

Все внутри пути, указанного с помощью --base-path, всегда разрешено.

Связывание библиотеки

Если в ваших контрактах используются библиотеки, вы заметите, что байт-код содержит подстроки вида «\_\_ $ 53aea86b7d70b31448b230b20ae141a537 $ \_\_». Это заполнители для фактических адресов библиотеки. Заполнитель - это 34-символьный префикс шестнадцатеричного кодирования хэша keccak256 полного имени библиотеки. Файл байт-кода также будет содержать строки вида // <placeholder> -> <fq library name>в конце, чтобы помочь определить, какие библиотеки представляют заполнители. Обратите внимание, что полное имя библиотеки - это путь к исходному файлу и имя библиотеки, разделенные знаком:. Вы можете использовать solc в качестве компоновщика, что означает, что он вставит для вас адреса библиотеки в этих точках:

Либо добавьте --libraries "file.sol: Math: 0x1234567890123456789012345678901234567890 file.sol: Heap: 0xabCD567890123456789012345678901234567890", чтобы указать адрес для каждой библиотеки (используйте запятые или пробелы в качестве разделителей для файлов) line) и запустите solc с помощью --libraries fileName.

Если solc вызывается с параметром --standard-json, он будет ожидать ввода JSON (как описано ниже) на стандартном вводе и вернет вывод JSON на стандартный вывод. Это рекомендуемый интерфейс для более сложных и особенно автоматизированных задач. Процесс всегда будет завершаться в состоянии «успешно» и сообщать обо всех ошибках через выходные данные JSON. Параметр --base-path также обрабатывается в стандартном режиме json.

Если solc вызывается с параметром --link, все входные файлы интерпретируются как несвязанные двоичные файлы (в шестнадцатеричном коде) в формате \_\_ $ 53aea86b7d70b31448b230b20ae141a537 $ \_\_-, указанном выше, и связываются на месте (если вход считывается из стандартного ввода , пишется в stdout). В этом случае все параметры, кроме --libraries (включая -o), игнорируются.

Связывание библиотек вручную по сгенерированному байт-коду не рекомендуется, потому что это не обновляет метаданные контракта. Поскольку метаданные содержат список библиотек, указанных во время компиляции, а байт-код содержит хеш метаданных, вы получите разные двоичные файлы в зависимости от того, когда выполняется связывание.

Вы должны попросить компилятор связать библиотеки во время компиляции контракта, используя параметр --libraries команды solc или ключ библиотек, если вы используете стандартный интерфейс JSON для компилятора.

В качестве заполнителя библиотеки использовалось полное имя самой библиотеки, а не ее хэш. Этот формат по-прежнему поддерживается solc --link, но компилятор больше не будет его выводить. Это изменение было внесено, чтобы снизить вероятность конфликта между библиотеками, поскольку можно было использовать только первые 36 символов полного имени библиотеки.

**Установка целевой версии EVM**

При компиляции кода контракта вы можете указать версию виртуальной машины Ethereum для компиляции, чтобы избежать определенных функций или поведения.

Компиляция для неправильной версии EVM может привести к неправильному, странному и сбойному поведению. Пожалуйста, убедитесь, особенно при использовании частной сети, что вы используете соответствующие версии EVM.

В командной строке вы можете выбрать версию EVM следующим образом:

solc --evm-version <version> contract.sol

В стандартном интерфейсе JSON используйте ключ "evmVersion" в поле "settings":

{

"sources": {...},#указываете источник

"settings": {#В settings указывается значение оптимизатора и версии EVM

"optimizer": {...},

"evmVersion": "<version>"

}

}

**Основные версии**

Ниже приведен список основные версий EVM и относящиеся к компилятору изменения, внесенные в каждую версию. Обратная совместимость между каждой версией не гарантируется.

* *homestead*

(самая старая версия)

* *tangerineWhistle*

Увеличена стоимость Gas для доступа к другим аккаунтам, актуально для оценки Gas и оптимизатора.

По умолчанию весь Gas отправляется на внешние звонки, ранее определенная сумма должна была сохраняться.

* *spuriousDragon*

Стоимость Gas для кода операции exp увеличена, актуально для оценки Gas и оптимизатора.

* *byzantium*

Коды операций Retreatndatacopy, Returnndatasize и staticcall доступны в сборке.

Код операции staticcall используется при вызове небиблиотечного представления или чистых функций, что не позволяет функциям изменять состояние на уровне EVM, т.е. даже применяется, когда вы используете недопустимые преобразования типов.

Можно получить доступ к динамическим данным, возвращаемым из вызовов функций.

введен код операции revert, что означает, что revert () не будет тратить впустую Gas.

* *constantinople*

Коды операций create2, extcodehash, shl, shr и sar доступны в сборке.

Операторы переключения используют коды операций переключения и поэтому требуют меньше Gas.

* petersburg

Компилятор ведет себя так же, как и с константинополем.

* istanbul (по умолчанию)

Коды операций chainid и selfbalance доступны в сборке.

* berlin (экспериментальный)

**Типы ошибок**

* JSONError: ввод JSON не соответствует требуемому формату, например input не является объектом JSON, язык не поддерживается и т. д.
* IOError: ошибки обработки ввода-вывода и импорта, такие как неразрешимый URL-адрес или несоответствие хэша в предоставленных источниках.
* ParserError: исходный код не соответствует языковым правилам.
* DocstringParsingError: теги NatSpec в блоке комментариев не могут быть проанализированы.
* SyntaxError: синтаксическая ошибка. Данная ошибка появляется при неправильной последовательности ввода команд. Например, когда команда continue была использована раньше команды цикла for.
* DeclarationError: недопустимые, неразрешимые или конфликтующие имена идентификаторов. например Идентификатор не найден
* TypeError: ошибка в системе типов, например недопустимые преобразования типов, недопустимые назначения и т. Д.
* UnimplementedFeatureError: эта функция не поддерживается компилятором, но ожидается, что она будет поддерживаться в будущих версиях.
* InternalCompilerError: в компиляторе возникла внутренняя ошибка - об этом следует сообщать как о проблеме. Исключение: неизвестный сбой во время компиляции - об этом следует сообщать как о проблеме.
* CompilerError: недопустимое использование стека компилятора - об этом следует сообщать как о проблеме.
* FatalError: Неустранимая ошибка обрабатывается неправильно - об этом следует сообщать как о проблеме.
* Warning: предупреждение, которое не остановило компиляцию, но должно быть устранено, если возможно.

**Инструменты компилятора**

**solidity-upgrade**

solidity-upgrade может помочь вам в полуавтоматическом режиме обновить ваши контракты до критических изменений языка. Хотя он не может и не может реализовать все необходимые изменения для каждой критической версии, он по-прежнему поддерживает те, которые в противном случае потребовали бы множества повторяющихся ручных настроек.

**Как это устроено**

Вы можете передать (а) исходный файл (ы) Solidity в solidity-upgrade [files]. Если они используют оператор импорта, который ссылается на файлы за пределами каталога текущего исходного файла, вам нужно указать каталоги, из которых разрешено читать и импортировать файлы, передавая --allow-paths [directory]. Вы можете игнорировать отсутствующие файлы, передав --ignore-missing.

solidity-upgrade основан на libsolidity и может анализировать, компилировать и анализировать ваши исходные файлы и может находить в них соответствующие обновления исходного кода.

Обновление исходного кода считается небольшими текстовыми изменениями исходного кода. Они применяются к представлению данных исходных файлов в памяти. Соответствующий исходный файл обновляется по умолчанию, но вы можете передать --dry-run для моделирования всего процесса обновления без записи в какой-либо файл.

Сам процесс обновления состоит из двух этапов. На первом этапе исходные файлы анализируются, и, поскольку невозможно обновить исходный код на этом уровне, ошибки собираются и могут регистрироваться путем передачи --verbose. На данный момент нет доступных обновлений исходного кода.

На втором этапе все исходные коды компилируются, и все активированные модули анализа обновлений запускаются одновременно с компиляцией. По умолчанию все доступные модули активированы. Пожалуйста, прочтите документацию по доступным модулям для получения дополнительной информации.

Это может привести к ошибкам компиляции, которые можно исправить путем обновления исходного кода. Если ошибок не происходит, значит, об обновлениях исходного кода не сообщается, и все готово. Если возникают ошибки и какой-то модуль обновления сообщил об обновлении исходного кода, применяется первый модуль, о котором было сообщено, и снова запускается компиляция для всех заданных исходных файлов. Предыдущий шаг повторяется, пока сообщается об обновлениях исходного кода. Если ошибки все еще возникают, вы можете зарегистрировать их, передав --verbose. Если ошибок не возникает, ваши контракты актуальны и могут быть скомпилированы с помощью последней версии компилятора.

**Доступные модули обновления**

* *constructor* –версия 0.5.0 –описание Конструкторы теперь должны быть определены с помощью ключевого слова конструктора.
* *visibility*–версия 0.5.0 –описание Явная видимость функции теперь обязательна, по умолчанию - public.
* *abstract* –версия 0.6.0–описание Ключевое слово abstract должно использоваться, если контракт не реализует все свои функции.
* *virtual*–версия 0.6.0 –описание Функции без реализации вне интерфейса должны быть помечены как виртуальные.
* *override*–версия 0.6.0–описание При переопределении функции или модификатора необходимо использовать новое ключевое слово override.
* *dotsyntax*–версия 0.7.0–описание Не рекомендуется использовать следующий синтаксис: f.gas (...) (), f.value (...) () и (новый C) .value (...) (). Замените эти вызовы на f {gas: ..., value: ...} () и (new C) {value: ...} ().
* *now*–версия 0.7.0 –описание Ключевое слово now устарело. Вместо этого используйте block.timestamp.
* *constructor-visibility*–версия 0.7.0 –описание Удаляет видимость конструкторов.

Пожалуйста, прочтите примечания к выпуску 0.5.0, примечания к выпуску 0.6.0, примечания к выпуску 0.7.0 и примечания к выпуску 0.8.0 для получения дополнительной информации.

**Допустимые команды для solidity-upgrade**

Использование: solidity-upgrade [команды] contract.sol

Допустимые команды:

* --help Показать справочное сообщение и выйти.
* --version Показать версию и выйти.
* --allow-paths Разрешить заданный путь для импорта. Список путей может быть предоставлен через запятую.
* --ignore-missing Игнорировать отсутствующие файлы.
* --modules модуль (ы) Активировать только определенный модуль обновления. Список модулей можно указывать, разделяя их запятыми.
* --dry-run Применять изменения только в памяти и не записывать на ввод файл.
* --verbose Печатать журналы, ошибки и изменения. Сокращает вывод патчей обновления.
* --unsafe Принять \* небезопасные \* изменения.

**Пример использования solc**

В данном разделе будет описано использование solc- компилятора для языка solidity. Будет использоваться операционная система macOS 10.12 Sierra

Процедура

1. Скомпилируйте один файл:

Введите: solc --bin SimpleStorage.sol

Эта команда показывает следующее:

Предупреждение: это предварительная версия компилятора, пожалуйста, не используйте ее в продакшене.

======= SimpleStorage.sol: SimpleStorage ======= Binary: 6060604052341561000f576000 ....... e27a0029

Примечание:Имя файла «SimpleStorage» и расширение файла «.sol» не имеют значения. Вместо SimpleStorage.sol вы можете назвать файл «test.txt».

1. Если вы хотите развернуть свой контракт, активируйте оптимизатор во время компиляции.

Примечание: двоичный код теперь короче.Введите: solc --optimize --bin SimpleStorage.sol

Эта команда показывает следующее:

Warning: This is a pre-release compiler version, please do not use it in production.  
  
======= SimpleStorage.sol:SimpleStorage ======= Binary: 6060604052341561000f576000.......e27a0029

1. Выведите бинарный исходник в отдельный файл, укажите папку вывода

Введите: solc --optimize -o outputDirectory --bin SimpleStorage.sol

Эта команда создает файл ./outputDirectory/SimpleStorage.bin

Примечание:solc принимает имя контракта в качестве имени файла.

1. Выведите двоичный исходный код и ABI в отдельные файлы.

Введите: solc --optimize -o outputDirectory --abi --bin SimpleStorage.sol

Эта команда создает файлы:

./outputDirectory/SimpleStorage.bin

./outputDirectory/SimpleStorage.abi

Файл abi содержит следующее:

[{"constant":false, .... ,"payable":false,"type":"function"}]

Заметка:

Чтобы перезаписать существующие файлы в outputDirectory, введите:

Введите: solc --optimize --overwrite -o outputDirectory --abi --bin SimpleStorage.sol

1. Файл GreeterCombined.sol содержит два контракта «Mortal» и «Greeter».

Выведите двоичный исходный код и ABI в отдельные файлы.

Введите: solc --optimize -o outputDirectory --abi --bin GreeterCombined.sol

Эта команда создает файлы:

./outputDirectory/Mortal.bin

./outputDirectory/Mortal.abi

./outputDirectory/Greeter.bin

./outputDirectory/Greeter.abi

1. Файл GreeterSimpleWithLibrary.sol содержит один контракт "GreeterSimple" и одну библиотеку "GreeterLib".

Выведите двоичный исходный код и ABI в отдельные файлы.

Введите: solc --optimize -o outputDirectory --abi --bin GreeterSimpleWithLibrary.sol

Эта команда создает файлы:

./outputDirectory/GreeterSimple.bin

./outputDirectory/GreeterSimple.abi

./outputDirectory/GreeterLib.bin

./outputDirectory/GreeterLib.abi

1. Скомпилируйте по два контракта в отдельный файл "Greeter.sol" и "Mortal.sol".

Выведите двоичный исходный код и ABI в отдельные файлы.

Введите: solc --optimize -o outputDirectory --abi --bin Greeter.sol Mortal.sol

Эта команда создает файлы:

./outputDirectory/Mortal.bin

./outputDirectory/Mortal.abi

./outputDirectory/Greeter.bin

./outputDirectory/Greeter.abi

Примечание:

Порядок, в котором вы указываете файлы .sol в команде "solc ... Greeter.sol Mortal.sol", не имеет значения.

Но вы должны использовать оператор импорта в файле Greeter.sol.

Вы можете использовать:

import "Mortal.sol";

или же

import "./Mortal.sol";

1. Скомпилируйте библиотеку GreeterLib.sol.

Выведите двоичный исходный код и ABI в отдельные файлы.

Введите: solc --optimize -o outputDirectory --abi --bin GreeterLib.sol

Эта команда создает файлы:

./outputDirectory/GreeterLib.bin

./outputDirectory/GreeterLib.abi

1. Скомпилируйте контракт и библиотеку в отдельные файлы "GreeterSimple.sol" и "GreeterLib.sol".

Выведите двоичный исходный код и ABI в отдельные файлы.

Введите: solc --optimize -o outputDirectory --abi --bin GreeterSimple.sol GreeterLib.sol

Эта команда создает файлы:

./outputDirectory/GreeterLib.bin

./outputDirectory/GreeterLib.abi

./outputDirectory/GreeterSimple.bin

./outputDirectory/GreeterSimple.abi

Примечание:

Порядок, в котором вы указываете файлы .sol в команде "solc ... GreeterSimple.sol GreeterLib.sol", не имеет значения.

Но вы должны использовать оператор импорта в файле GreeterSimple.sol.

Вы можете использовать:

import "GreeterLib.sol";

или же

import "./GreeterLib.sol";

1. Скомпилируйте один файл с включенным оптимизатором байт-кода, и оценочное количество запусков контракта для настройки оптимизатора установлено на 0:

Введите: solc --optimize --optimize-run 0 --bin SimpleStorage.sol

Эта команда показывает следующее:

Warning: This is a pre-release compiler version, please do not use it in production.  
  
======= SimpleStorage.sol:SimpleStorage ======= Binary: 6060604052341561000f576000.......e27a0029

Примечание:

Если вы не укажете --optimize-run, по умолчанию будет установлено значение --optimize-run 200.