导读

以太坊是什么？

以太坊是一个全新开放的区块链平台，它允许任何人在平台中建立和使用通过区块链技术运行的去中心化应用。就像比特币一样，以太坊不受任何人控制，也不归任何人所有——它是一个开放源代码项目，由全球范围内的很多人共同创建。和比特币协议有所不同的是，以太坊的设计十分灵活，极具适应性。在以太坊平台上创立新的应用十分简便，随着 Homestead 的发布，任何人都可以安全地使用该平台上的应用。

本电子书原文最早发布于区块链技术博客（http:// tryblockchain.org），由汇智网（http://www.hubwiz.com）编目整理，是目前网上流传的最完整的 Truffle 文档中文版。

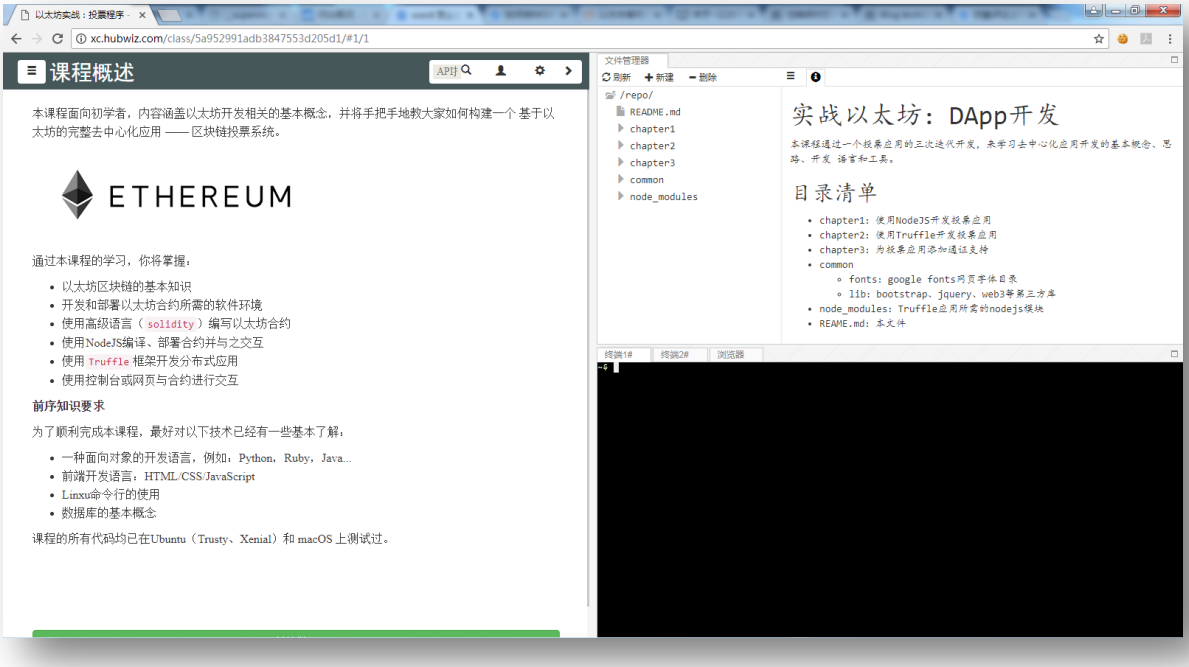
但由于以太坊本身（以及周边生态）的发展非常快，一些实践性内容已经落后于现状。因此编者建议本电子书的读者，在阅读时应注意吸收核心的理念思想，而不要过分关注书中的实践操作环节。

为了弥补这一遗憾，汇智网推出了在线交互式以太坊 DApp 实战开发课程，以去中心化投票应用（Voting DApp）为课程项目，通过三次迭代开发过程的详细讲解与在线实践，并且将区块链的理念与去中心化思想贯穿于课程实践过程中，为希望快速入门区块链开发的开发者提供了一个高效的学习与价值提升途径。读者可以通过以下链接访问《以太坊 DApp 开发实战入门》在线教程：

http://xc.hubwiz.com/course/5a952991adb3847553d205d1?affid=trf

教程预置了开发环境。进入教程后，可以在每一个知识点立刻进行同步实践，

而不必在开发环境的搭建上浪费时间：



汇智网 Hubwiz.com

2018.2

1. Truffle 简介

原文地址：http://truffleframework.com/docs/
Truffle 是一个世界级的开发环境，测试框架，以太坊的资源管理通道，致力于让以太坊上的开发变得简单，Truffle 有以下：

- 内置的智能合约编译，链接，部署和二进制文件的管理。
- 快速开发下的自动合约测试。
- 脚本化的，可扩展的部署与发布框架。
- 部署到不管多少的公网或私网的网络环境管理功能
- 使用 EthPM&NPM 提供的包管理，使用 ERC190 标准。
- 与合约直接通信的直接交互控制台（写完合约就可以命令行里验证了）。
- 可配的构建流程，支持紧密集成。
- 在 Truffle 环境里支持执行外部的脚本。

Truffle 是最流行的开发框架，使命是让开发更容易

安装方式：

```
$ npm install -g truffle
```

环境要求

- NodeJS 5.0+
- Windows，Linux，或 Mac OS X

Truffle 需要以太坊客户端，需要支持标准的 JSON RPC API。对于开发来说，有一些更适合，后续章节会介绍(如：testRPC 提供编码调试时的实时反馈)。

Windows 用户的建议

如果你是 Windows 用户，我们推荐你使用 Powershell 或 Git BASH 来安装和使用 Truffle 框架。这两个 shell 环境相对默认的，提供了更方便的一些特性。

如果你必须使用命令行，可以看这里的关于如何配置 Truffle 的讨论。

2. Truffle 客户端

原文地址：http://truffleframework.com/docs/getting_started/client
有许多的以太坊客户端可以选择。我们推荐在开发和部署时使用不同客户端。

适用开发的客户端

- EtherumJS TestRPC

当开发基于 Truffle 的应用时，我们推荐使用 EthereumJS TestRPC。它是一个完整的在内存中的区块链仅仅存在于你开发的设备上。它在执行交易时是实时返回，而不等待默认的出块时间，这样你可以快速验证你新写的代码，当出现错误时，也能即时反馈给你。它同时还是一个支持自动化测试的功能强大的客户端。Truffle 充分利用它的特性，能将测试运行时间提速近 90%。

适用正式发布的客户端

- Geth (go-ethereum)
- WebThree(cpp-ethereum)
- More

对此有许多官方和非官方的以太坊客户端可供选择。最好使用 TestRPC 客户端充分测试后，再使用这些客户端。这些是完整的客户端实现，包括挖矿，网络，块及交易的处理，Truffle 可以在不需要额外配置的情况下发布到这些客户端。

当发布到私有网络中

私人网络中使用了相同的技术，但却有不同的配置。所以你可以将上面提及的客户端来运行一个私有的网络，部署到这样的网络也是使用同样的方式。

3. 创建一个工程

原文地址：http://truffleframework.com/docs/getting_started/project

创建工程目录

首先创建一个工程目录。你可以使用你喜欢的文件浏览器或使用下面的命令在命令行创建一个目录：

```
$ mkdir myproject
```

初始化你的工程

接下来，通过下面的命令初始化一个 Truffle 工程：

```
$ truffle
Truffle v3.4.11 - a development framework for Ethereum
$ cd myproject
$ truffle init
```

完成后，你将拥有如下目录：

- app/ - 你的应用文件运行的默认目录。这里面包括推荐的 javascript 文件和 css 样式文件目录，但你可以完全决定如何使用这些目录。
- contract/ - Truffle 默认的合约文件存放地址。
- migrations/ - 存放发布脚本文件
- test/ - 用来测试应用和合约的测试文件
- truffle.js - Truffle 的配置文件

默认工程：METACOIN

`truffle init`会默认创建一个构建在以太坊内的代币 demo 应用。我们可以使用这个工程来进行快速的学习，或者你也可以删除这些文件来创建一个你自己的工程。
如果任何问题，欢迎留言批评指正。

4. 编译合约

原文地址：http://truffleframework.com/docs/getting_started/compile

合约位置

所有你的合约应该位于`./contracts`目录。默认我们提供了一个合约文件，一个库文件，均以`.sol`结尾作为示例。尽管库文件有一定的特殊性，但为简单起见，当前均称之为合约。

命令

要编译您的合约，使用：

```
truffle compile
```

Truffle 仅默认编译自上次编译后被修改过的文件，来减少不必要的编译。如果你想编译全部文件，可以使用`--compile-all`选项。

```
truffle compile --compile-all
```

约定

Truffle 需要定义的合约名称和文件名准确匹配。举例来说，如果文件名为`MyContract.sol`，那么合约文件须为如下两者之一：

```
contract MyContract {
  ...
}
// or
library MyContract {
```

```
  ...
}
```

这种匹配是区分大小写的，也就是说大小写也要一致。推荐大写每一个开头字母，如上述代码定义。

依赖

你可以通过使用 import 来声明依赖。Truffle 将会按正确顺序依次编译合约，并在需要的时候自动关联库。

编译目录

编译的输出位于`./build/contracts`目录。如果目录不存在会自动创建。这些编译文件对于 Truffle 框架能否正常工作至关重要。你不应该在正常的编译或发布以外手动修改这些文件。
如果任何问题，欢迎留言批评指正。

5. 移植

原文地址：http://truffleframework.com/docs/getting_started/migrations
移植是由一些 Javascript 文件组成来协助发布到以太坊网络。主要目的是用来缓存你的发布任务，它的存在基于你的发布需求会改变的前提。当你的工程发生了重要的改变，你将创建新的移植脚本来将这些变化带到区块链上。之前运行移植的历史记录通过一个特殊的`Migrations`合约来记录到链上，下面有详细说明。

命令

执行移植，使用下述命令：

```
truffle migrate
```

这个命令会执行所有的位于`migrations`目录内的移植脚本。如果你之前的移植是成功执行的。`truffle migrate`仅会执行新创建的移植。如果没有新的移植脚本，这个命令不同执行任何操作。可以使用选项`--reset`来从头执行移植脚本。

移植脚本文件

一个样例文件如下：

文件名：4_example_migration.js

```
module.exports = function(deployer) {
  // deployment steps
  deployer.deploy(MyContract);
};
```

需要注意的是文件名以数字开头，一个描述性的后缀结尾。数字前缀是必须的，用于记录移植是否成功。后缀仅是为了提高可读性，以方便理解。

移植 js 里的 `exports` 的函数接受一个 `deployer` 对象作为第一个参数。这个对象用于发布过程，提供了一个清晰的语法支持，同时提供一些通过的合约部署职责，比如保存发布的文件以备稍后使用。`deployer` 对象是用来缓存(stage)发布任务的主要操作接口。API 接口见后说明。

像所有其它在 Truffle 中的代码一样，Truffle 为你提供了你自己代码的`合约抽象层(contract abstractions)`，并且进行了初始化，以方便你可以便利的与以太坊的网络交互。这些抽象接口是发布流程的一部分，稍后你将会看到。

初始移植

Truffle 需要一个移植合约来使用移植特性。这个合约内需要指定的接口，但你可以按你的意味修改合约。对大多数工程来说，这个合约会在第一次移植时进行的第一次部署，后续都不会再更新。通过 `truffle init` 创建一个全新工程时，你会获得一个默认的合约。

文件名:contracts/Migration.sol

```
contract Migrations {
  address public owner;

  // A function with the signature `last_completed_migration()`, returning a
uint, is required.
  uint public last_completed_migration;

  modifier restricted() {
    if (msg.sender == owner) _
  }

  function Migrations() {
    owner = msg.sender;
```

```
  }

  // A function with the signature `setCompleted(uint)` is required.
  function setCompleted(uint completed) restricted {
    last_completed_migration = completed;
  }

  function upgrade(address new_address) restricted {
    Migrations upgraded = Migrations(new_address);
    upgraded.setCompleted(last_completed_migration);
  }
}
```

如果你想使用移植特性，你必须在你第一次部署合约时，部署这个合约。可以使用如下方式来创建一次移植。

文件名：migrations/1_initial_migrations.js

```
module.exports = function(deployer) {
  // Deploy the Migrations contract as our only task
  deployer.deploy(Migrations);
};
```

由此，你可以接着创建递增的数字前缀来部署其它合约。

部署器(deployer)

你的移植文件会使用部署器来缓存部署任务。所以，你可以按一定顺序排列发布任务，他们会按正确顺序执行。

```
// Stage deploying A before B
deployer.deploy(A);
deployer.deploy(B);
```

另一选中可选的部署方式是使用 Promise。将部署任务做成一个队列，是否部署依赖于前一个合约的执行情况。

```
// Deploy A, then deploy B, passing in A's newly deployed address
deployer.deploy(A).then(function() {
  return deployer.deploy(B, A.address);
});
```

如果你想更清晰，你也可以选择实现一个 Promise 链。关于部署的 API，在后面进行说明。

网络相关

可以根据发布到的网络的具体情况进行不同的部署流程。这是一个高级特性，你先看 2. 网络与 APP 部署的相关内容后，再继续。

要实现不同条件的不同部署步骤，移植代码中需要第二个参数 `network`。示例如下：

```
module.exports = function(deployer, network) {
  // Add demo data if we're not deploying to the live network.
  if (network != "live") {
    deployer.exec("add_demo_data.js");
  }
}
```

部署 API

部署器有许多的可用函数，用来简化部署流程。

DEPLOYER.DEPLOY(CONTRACT, ARGS...)

发布一个指定的合约，第一参数是合约对象，后面是一些可选的构造器参数。

这个函数适用于单例合约，它只会在你的 dapp 中只创建一个这个合约的实例(单例)。函数会在部署后设置合约的地址(如：Contract.address 将等于新的部署地址)，它将会覆盖之前存储的地址。

你也可以传入一个合约数组，或数组的数组来加速多合约的部署。

需要注意的是如果库的地址可用，`deploy` 会自动为这个部署的合约联接任何需要的库。所以，如果合约依赖某个库，你应该先部署这个库。
例子：

```
// Deploy a single contract without constructor arguments
deployer.deploy(A);

// Deploy a single contract with constructor arguments
deployer.deploy(A, arg1, arg2, ...);

// Deploy multiple contracts, some with arguments and some without.
// This is quicker than writing three `deployer.deploy()` statements as the deployer
// can perform the deployment as a batched request.
deployer.deploy([
  [A, arg1, arg2, ...],
  B,
```

```
  [C, arg1]
]);
```

DEPLOYER.LINK(LIBRARY, DESTINATIONS)

联接一个已经发布的库到一个或多个合约。`destinations`可以是一个合约或多个合约组成的一个数组。如果目标合约并不依赖这个库，部署器会忽略掉这个合约。
这对于在 dapp 中不打算部署的合约（如：非单例）但却需要在使用前先联接的情况下非常有用。

```
// Deploy library LibA, then link LibA to contract B
deployer.deploy(LibA);
deployer.link(LibA, B);

// Link LibA to many contracts
deployer.link(LibA, [B, C, D]);
```

DEPLOYER.AUTOLINK(CONTRACT)

关联合约依赖的所有库。这需要所依赖的库已经部署，或在其前一步部署。

例子：

```
// Assume A depends on a LibB and LibC
deployer.deploy([LibB, LibC]);
deployer.autolink(A);
```

另外你可以省略参数来调用函数`autolink()`。这会自动关联合约依赖的所有库。

需要保证在调用这个函数前，所有被需要的库已经部署了。
例子：

```
// Link *all* libraries to all available contracts
deployer.autolink();
```

DEPLOYER.THEN(FUNCTION() {...})

Promise 语法糖，执行做生意的部署流程。

例子：

```
deployer.then(function() {
  // Create a new version of A
  return A.new();
```

```
}).then(function(instance) {
  // Set the new instance of A's address on B.
  var b = B.deployed();
  return b.setA(instance.address);
});
```

DEPLOYER.EXEC(PATHTOFILE)

执行 `truffle exec` 做为部署的一部分。查看 10. 外部脚本章节了解更多。

例子：

```
// Run the script, relative to the migrations file.
deployer.exec("../path/to/file/demo_data.js");
```

如果任何问题，欢迎留言批评指正。

6. 构建应用

原文地址：http://truffleframework.com/docs/getting_started/build

默认构建

Truffle 集成了默认的构建来方便使用。但也许不适合每个项目，所以你也许需要其它的来打包你的应用。在 3. 构建流程里查看更多信息。默认的构造目标是 web 应用，但也可以很容易的转变为其它的构造流程，比如适用于命令行或库的流程。

特性

默认构建有一些特性来帮助你快速的开始：

- 在浏览器内自动的初始化你的应用，包括引入你编译的合约，部署的合约信息，和以太坊客户端信息配置。
- 包含常见的依赖，如 web3 和 Ether Pudding
- 内置支持 ES6 和 JSX
- SASS 支持
- Uglifyjs 支持

配置

你可以随间的修改默认的构建内容，原始的构建内容目录如下：

```
app/
- javascripts/
  - app.js
- stylesheets/
  - app.css
- images/
- index.html
```

在 1. 配置文件中的构建配置文件如下：

```
{
  "build": {
    // Copy ./app/index.html (right hand side) to ./build/index.html (left hand side).
    "index.html": "index.html",

    // Process all files in the array, concatenating them together
    // to create a resultant app.js
    "app.js": [
      "javascripts/app.js"
    ],

    // Process all files in the array, concatenating them together
    // to create a resultant app.css
    "app.css": [
      "stylesheets/app.scss"
    ],

    // Copy over the whole directory to the build destination.
    "images/": "images/"
  }
}
```

配置文件中的配置键描述了最终的打包目标名称，右边的配置目录或文件数组则是要打包的目录的内容。打包过程根据文件扩展，将文件连接形成一个结果文件，并放到构建的目标位置。如果指定的是一个字符串而不是一个数组，这个字符串代指的文件如果需要会直接拷到对应的构建目录。如果字符串以“/”结尾，则会被识别为一个目录，整个目录会不经调整直接拷贝到对应的目录。所以的指定值都是默认相对于`/app`目录来指定的。

你可以在任何时间改变配置和目录结构。并不强制要求需要`javascript`和`css`文件目录，所以删除构建配置文件中的对应配置就可以了。
特别注意：如果你想默认构建在前端初始化你的应用，务必保证有一个构造目标`app.js`，因为默认构建会将相关代码附加到这个文件，而不是其它文件。

命令

要创建你的前端工程，执行：

```
truffle build
```

构建结果

构建结果存在`./build`目录。所以合约文件则在对应的位置`./build/contracts`。

注意事项

默认构建虽简单易用，但它仍有一些缺点：

- 当前不支持`import`，`require`等。所以不能提供`browserify`，`Webpack`和`CommonJS`这样的工具。由此让依赖管理变得有些困难。
- 这是一套自定义的构建系统，与其它流行构建系统不兼容。
- 它可以扩展，但是自定义的方法和 API。

默认构建在将来可能会被取代，但在较长时间里，都将会是默认的以支持之前构建的 DAPP.Truffle 提供了许多方式来切换到不同的构建流程，可以在 3. 构建流程这里找到更多的例子。

如果任何问题，欢迎留言批评指正。

7. 合约交互

原文地址：http://truffleframework.com/docs/getting_started/contracts

背景

标准的与以太坊网络交互的方法是通过以太坊官方构建的 Web3 库。尽管这个库非常有用，但使用其提供接口与合约交互有些困难,特别是以太坊的新手。为降低学习曲线，Truffle 使用 Ether Pudding 库，它也是基于 Web3 的基础之上，目的是为了让交互更简单。

读写数据

以太坊网络把在网络上读与写数据进行了区分，这个区分对于如何写程序影响很大。通常来说，写数据被称作`交易(transaction)`，读数据被称作`调用(call)`。对于交易与调用，他们分别有如下特性：

交易(Transaction)

交易本质上改变了整个以太坊网络的数据状态。一个交易可以是向另一个帐户发送 ether(以太坊网络代币)这样的简单行为，也可以是执行合约函数，添加一个新合约到以太坊网络这样的复杂行为。交易的典型特征是写入(或修改)数据。交易需要花费 ether，也被称作 gas，交易的执行需要时间。当你通过交易执行一个合约的函数时，你并不能立即得到执行结果，因为交易并不是立即执行的。大多娄情况下，通过执行交易不会返回值；它会返回一个交易的 ID.总的来说，交易具有如下特征：

- 需要 gas（Ether）
- 改变网络的状态
- 不会立即执行
- 不会暴露返回结果(仅有交易 ID)

调用

调用，则与上述的交易非常不同。调用可以在网络上执行代码，但没有数据会被改变(也许仅仅是些临时变量被改变)。调用的执行是免费的，典型的行为就是读取数据。通过调用执行一个合约函数，你会立即得到结果。总的来说，调用具有如下特征：

- 免费（不花费 gas）
- 不改变网络状态
- 立即执行
- 有返回结果。

如果选择，取决于你想干什么，或者说想写数据，还是读数据。

接口(abstract)

为了来体验一下合约接口的作用，我们使用框架自带的默认 metacoin 的合约例子。

```
import "ConvertLib.sol";

contract MetaCoin {
```

```
  mapping (address => uint) balances;

    event Transfer(address indexed _from, address indexed _to, uint256 _value);

    function MetaCoin() {
        balances[tx.origin] = 10000;
    }

    function sendCoin(address receiver, uint amount) returns(bool sufficient)
{
        if (balances[msg.sender] < amount) return false;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        Transfer(msg.sender, receiver, amount);
        return true;
    }
    function getBalanceInEth(address addr) returns(uint){
        return ConvertLib.convert(getBalance(addr),2);
    }
    function getBalance(address addr) returns(uint) {
        return balances[addr];
    }
}
```

合约有三个方法和一个构造方法。所有三个方法可以被执行为交易或调用。

现在我们来看看 Truffle 和 Ether Pudding 为我们提供的叫 `MetaCoin` 的 Javascript 对象，可以在前端中使用：

```
// Print the deployed version of MetaCoin
console.log(MetaCoin.deployed());

// outputs:
//
// Contract
// - address: "0xa9f441a487754e6b27ba044a5a8eb2eec77f6b92"
// - allEvents: ()
// - getBalance: ()
// - getBalanceInEth: ()
// - sendCoin: ()
```

接口层提供了合约中以应的函数名。它还包含一个地址，指向到 MetaCoin 合约的部署版本。

执行合约函数

通过这套框架为我们提供的接口，我们可以简单的在以太坊网络上执行合约函数。

执行交易

在上述例子 MetaCoin 合约中，我们有三个可以执行的函数。如果你对这三个函数稍加分析就会发现，只有 `sendCoin` 会对网络造成更改。`sendCoin` 函数的目标将 Meta Coin 从一个帐户发送到另一些帐户，这些更改需要被永久存下来。

当调用 `sendCoin`，我们将把他们作为一个交易来执行。下面的例子我们来演示下把 10 个币，从一个帐户发到另一个帐户，改变要永久的保存在网络上：

```
var account_one = "0x1234..."; // an address
var account_two = "0xabcd..."; // another address

var meta = MetaCoin.deployed();
meta.sendCoin(account_two, 10, {from: account_one}).then(function(tx_id) {
  // If this callback is called, the transaction was successfully processed.
  // Note that Ether Pudding takes care of watching the network and triggering
  // this callback.
  alert("Transaction successful!")
}).catch(function(e) {
  // There was an error! Handle it.
})
```

上述代码有一些有趣点，我们来了解一下：

- 我们直接调用接口的 `sendCoin` 函数。最终是默认以交易的方式来执行的。
- 交易被成功执行时，回调函数会直到交易被执行时才真正被触发。这样带来的一个好处是你不用一直去检查交易的状态。
- 我们对 `sendCoin` 函数传递了第三个参数，需要注意的是原始合约函数的定义中并没有第三个参数。这里你看到的是一个特殊的对象，用于编辑一些交易中的指定细节，它可以总是做为第三个参数传进。这里，我们设置 `from` 的地址为 `account_one`.

执行调用

继续用 MetaCoin 的例子。其中的 `getBalance` 函数就是一个很好的从网络中读取数据的例子。它压根不需要进行任何数据上的变更，它只是返回传入的地址的帐户余额，我们来简单看一下：

```
var account_one = "0x1234..."; // an address

var meta = MetaCoin.deployed();
```

```
meta.getBalance.call(account_one, {from: account_one}).then(function(balance)
{
  // If this callback is called, the call was successfully executed.
  // Note that this returns immediately without any waiting.
  // Let's print the return value.
  console.log(balance.toNumber());
}).catch(function(e) {
  // There was an error! Handle it.
})
```

一些有意思的地方如下：

- 我们必须通过 `.call()` 来显示的向以太坊网络表明，我们并不会持久化一些数据变化。
- 我们得到了返回结果，而不是一个交易 ID。这里有个需要注意的是，以太坊网网络可以处理非常大的数字，我们被返回了一个 BigNumber 对象，框架再将这个对象转化了一个 number 类型。

警告：我们在上述的例子中将返回值转成了一个 number 类型，是因为例子中的返回值比较小，如果将一个 BigNumber 转换为比 javascript 支持的 number 最大整数都大，你将会出现错误或不可预期的行为。

捕捉事件(Catching Events)

你的合约可以触发事件，你可以进行捕捉以进行更多的控制。事件 API 与 Web3 一样。可以参考 Web3 documentation 来了解更多。

```
var meta = MetaCoin.deployed();
var transfers = meta.Transfer({fromBlock: "latest"});
transfers.watch(function(error, result) {
  // This will catch all Transfer events, regardless of how they originated.
  if (error == null) {
    console.log(result.args);
  }
}
```

METHOD:DEPLOYED()

每一个抽象出来的合约接口都有一个 `deployed()` 方法，上述例子中，你已经见到过。调用这个函数返回一个实例，这个实例代表的是之前部署到网络的合约所对应的抽象接口的实例。

```
var meta = MetaCoin.deployed();
```

警告：这仅对使用 `truffle deploy` 部署的合约，且一定是在 `project configuration` 中配置发布的才有效。如果不是这样，这个函数执行时会抛出异常。

METHOD:AT()

类似于 `deployed()`，你可以通过一个地址来得到一个代表合约的抽象接口实例。当然这个地址一定是这个合约的部署地址。

```
var meta = MetaCoin.at("0x1234...")
```

警告：当你的地址不正确，或地址对应的合约不正确时，这个函数并不会抛出异常。但调用接口时会报错。请保证在使用 `at()` 时输入正确的地址。

METHOD:NEW()

你可以通过这个方法来部署一个完全全新的合约到网络中。

```
MetaCoin.new().then(function(instance) {
  // `instance` is a new instance of the abstraction.
  // If this callback is called, the deployment was successful.
  console.log(instance.address);
}).catch(function(e) {
  // There was an error! Handle it.
});
```

需要注意的是这是一个交易，会改变网络的状态。

如果任何问题，欢迎留言批评指正。

8. 测试合约

原文地址：http://truffleframework.com/docs/getting_started/testing

框架

Truffle 使用 Mocha 测试框架来做自动化测试，使用 Chai 来做断言。这两个库的结合可能让人耳目一新，我们基于这两者之上，提供一种方式来编译简单和可管理的合约自动化测试用例。

位置

测试文件应置于`./tests`目录。Truffle 只会运行以`.js`，`.es`，`.es6`和`.jsx`结尾的测试文件，其它的都会被忽略。

测试用例

每个测试文件至少应该包含至少一个对 Mocha 的`describe()`函数的调用，详情见 Mochajs Documentation。另一种方式是使用 Truffle 自定义的`contract()`函数，作用类型`describe()`函数，但额外添加了一些特性：

- 在每一个`contract()`函数执行前，你的合约都会重部署到以太坊客户端中，这样测试用例会在一个干净状态下执行。
- `contract()`函数支持传入多个可用的帐户做为第二个参数传入，你可以用此来进行测试。

当你需要与你写的合约进行交互时，使用`contract()`，否则使用`describe()`函数。

测试用例示例

`truffle init`命令为我们提供了一个简单的测试用例例子。它会先部署你的合约，然后执行在`it()`块中指定的测试用例。

```
contract('MetaCoin', function(accounts) {
  it("should put 10000 MetaCoin in the first account", function() {
    // Get a reference to the deployed MetaCoin contract, as a JS object.
    var meta = MetaCoin.deployed();

    // Get the MetaCoin balance of the first account and assert that it's 10000.
    return meta.getBalance.call(accounts[0]).then(function(balance) {
      assert.equal(balance.valueOf(), 10000, "10000 wasn't in the first
account");
    });
  });
});
```

需要注意的是在 `contract()` 函数的传入的 `MetaCoin` 仅仅因为用来展示，说明它是 `MetaCoin` 相关的测试，并无本质作用。

合约

Truffle 提供了接口抽象，方便你与合约进行便捷交互。通过 `var meta = MetaCoin.deployed()` 这行。Truffle 设法保证了，你可以在测试用例，前端，移植(Migration)中都能用这种方式与你自己写的合约进行交互。可以在 7. 合约交互章节了解更多。

命令

要执行测试，执行下面的命令：

```
truffle test
```

你也可以对单个文件执行测试：

```
truffle test ./path/to/test/file.js
```

注意事项

EthereumJS TestRPC 在进行自动化测试时相比其它客户端会快非常多。而且，TestRPC 包含了一些，Truffle 可以用来加速测试的特性。作为一个能用流程，我们建议你在开发和测试环节使用 TestRPC。当你筹备好要发布到现上时，才使用 Geth 或其它官方以太坊客户端来进行一次测试。

更多

Truffle 提供了操作 Mocha 的配置文件的入口。参见 1. 配置文件章节来了解更多。

如果任何问题，欢迎留言批评指正。

9. 控制台

原文：http://truffleframework.com/docs/getting_started/console

背景

有时在进行测试和 debug 时，或手动执行交易时与合约进行直接交互是需要的。Truffle 提供了一种更加简单的方式，通过交互式控制台来与你的那些准备好的合约进行交互。

命令

启动控制台，使用：

```
truffle console
```

这会使用默认网络来调起一个控制台，会自动连接到一个运行中的以太坊客户端。

你可以使用选项 `--network` 来修改这个特性，更多细节参见 2. 网络与 APP 部署 和 4. Truffle 命令指南。

当你加载了控制台，你会看到下面的输出：

```
$ truffle console
truffle(default)>
```

`default` 的意思是说，你当前连接到的是默认网络。

特性

控制台支持 Truffle 命令行支持的命令，比如，你可以在控制台中执行 `migrate --reset`，其效果与在命令行中执行 `truffle migrate --reset` 的效果一致。Truffle 的控制台额外增加如下特性：

- 所有已经编译的合约都可用。就像在开发测试，前端代码中，或者移植代码中那样使用。
- 在每个命令后，你的合约会被重新加载。如使用 `migrate --reset` 命令后，你可以立即使用新分配的地址和二进制。
- `web3` 库也可以使用，且也连到你了的以太坊客户端。
- 所有命令返回的 promise，会自动解析，直接打印出结果，你可以不用输入 `then()`，简化了命令。如下：

```
truffle(default)> MyContract.deployed().getValue.call(); //
5
```

如果任何问题，欢迎留言批评指正。

10. 外部脚本

原文：http://truffleframework.com/docs/getting_started/scripts

背景

你也许会经常的执行外部脚本来与你的合约进行交互。Truffle 提供了一个简单的方式来进行这个。首先，启动你的合约，连上你想要的网络，通过 1. 配置文件

命令

要执行外部(external)脚本，执行下述命令：

```
$ truffle exec <path/to/file.js>
```

文件结构

为了外部脚本能正常执行，Truffle 需要它们能通过 Javascript 的模块的方式导出一个函数，且有一个回调函数作为参数：

```
module.exports = function(callback) {
  // perform actions
}
```

脚本内，你可以执行你想要做的任何事，这个回调在脚本执行结束后被调用。回调函数只有一个参数，这个参数传的是错误状态。如果出现错误，整个执行会中止，并返回一个非 0 的退出码(exit code)。

如果任何问题，欢迎留言批评指正。

11.工作流

原文： http://truffleframework.com/docs/getting_started/workflow

命令

我们已经推荐 EtherumJS TestRPC 很多次了，以在开发过程中得到快速的结果反馈。然而，Truffle 更提供了两个命令来让开发更快。

Truffle watch

监控文件系统的文件变化，重编译，重部署你的合约。在被修改后需要的时候，会重构建前端代码。

使用方法：

```
truffle watch
```

查看 4. Truffle 命令指南来了解更多。

Truffle serve

监控文件系统的变化，重编译，部署，构建，并在 `http://localhost:8080/` 提供服务。
使用方式：

```
truffle serve
```

你可以更改端口，参见 4. Truffle 命令指南来了解更多。

如果任何问题，欢迎留言批评指正。

12. 配置文件

http://truffleframework.com/docs/advanced/configuration

位置

你的配置文件是 `truffle.js`。位于项目的根目录下。这个文件是 Javascript 文件，支持执行代码来创建配置。它必须导出一个对象，来代表项目配置，如下面的例子。

WINDOWS 下的命名空间冲突

当使用 Windows 的命令行时，默认的配置文件名与 `truffle` 冲突。这种情况下，我们推荐使用 Windows 的 power Shell 或 Git BASH。你也可以将配置文件重命名为 `truffle-config.js` 来避免冲突。

例子

```
module.exports = {
  build: {
    "index.html": "index.html",
    "app.js": [
      "javascripts/app.js"
    ],
    "app.css": [
      "stylesheets/app.css"
    ],
    "images/": "images/"
  },
  rpc: {
    host: "localhost",
    port: 8545
  }
};
```

默认配置文件已经配置好了，`build` 和 `rpc` 。这些默认和非默认选项，详见后面。

选项

BUILD

这个是前端的构建配置。默认调用默认构建器，在上述构建章节，有所说明。但你也可以自定的构建流程，查看高级构建流程章节来了解更多。

例子：

```
build: {
  "index.html": "index.html",
  "app.js": [
    "javascripts/app.js"
  ],
  "app.css": [
    "stylesheets/app.css"
  ],
  "images/": "images/"
}
```

NETWORKS

指定在移植(Migration)时使用哪个网络。当在某个特定的网络上编译或运行移植时，合约会缓存起来方便后续使用。当你的合约抽象层检查到你连到某个网络上时，它会使用这个这个网络上原有的缓存合约来简化部署流程。网络通过以太坊的 RPC 调用中的 `net_version` 来进行标识。

下述的 `networks` 对象，通过一个网络名做为配置的键，值对应定义了其网络参数。`networks` 的对应选项不是必须的，但如果一旦指定，每个网络必须定义一个对应的 `network_id`。如果你想指定一个默认网络，你可以通过将 `netword_id` 的值标记为 `default` 来实现，当没有匹配到其它的网络时，就会使用默认网络。

需要注意的是整个配置中，应该有且仅有一个 `default` 的网络。一般来说，默认网络主要用于开发，配置，合约等数据没有长期保存的需要，网络 ID 也会因 TestRPC 的重启而频繁改变时。
网络名称用于用户接口调用时使用，在移植中的使用方式如下：

```
$ truffle migrate --network live
```

你还可以选择性的指定 rpc 的配置信息。下面是一个示例：

```
networks: {
  "live": {
    network_id: 1, // Ethereum public network
    // optional config values
    // host - defaults to "localhost"
    // port - defaults to 8545
    // gas
    // gasPrice
    // from - default address to use for any transaction Truffle makes during
migrations
  },
  "morden": {
    network_id: 2,        // Official Ethereum test network
    host: "178.25.19.88", // Random IP for example purposes (do not use)
    port: 80
  },
  "staging": {
    network_id: 1337 // custom private network
    // use default rpc settings
  },
```

```
  "development": {
    network_id: "default"
  }
}
```

RPC

关于如何连接到以太坊客户端的一些细节。`host` 和 `port` 是需要，另外还需要一些其它的。

- `host`：指向以太坊客户端的地址。本机开发时，一般为 `localhost`
- `port`:以太坊客户端接收请求的端口，默认是 `8545`
- `gas`:部署时的 Gas 限制，默认是 `4712388`
- `gasPrice`:部署时的 Gas 价格。默认是 `100000000000` (100 Shannon)
- `from`:移植时使用的源地址。如果没有指定，默认是你的以太坊客户端第一个可用帐户。

示例：

```
rpc: {
  host: "localhost",
  port: 8545
}
```

MOCHA

MochaJS 测试框架的配置选项，详细参考 documentation。

示例：

```
mocha: {
  useColors: true
}
```

如果任何问题，欢迎留言批评指正。

13. 网络与 APP 部署

背景

即使最小的项目也至少会与两个以上的区块链打交道，一个是开发机上的测试链，如 EthereumJS TestRPC.另一个则是比如你最终要部署的网络，如以太坊网络，自己公司内的私链等等。Truffle 提供了一个管理不同网络下的构建和部署资源的系统，来简化最终的部署流程。

配置

详见 1. 配置文件章节了解更多。

指定一个网络

大多数 Truffle 提供的命令根据指定的网络不同而表现不同，会使用对应网络下的合约和配置信息。可以通过 `--network` 选项在参数上进行控制。

```
$ truffle migrate --network live
networks: {
  development: {
    host: "localhost",
    port: 8545,
    network_id: "*" // match any network
  },
  live: {
    host: "178.25.19.88", // Random IP for example purposes (do not use)
    port: 80,
    network_id: 1,        // Ethereum public network
    // optional config values
    // gas
    // gasPrice
    // from - default address to use for any transaction Truffle makes during
migrations
  }
}
```

在上面这个例子中，Truffle 会在 `live` 网络中进行移植。如果配置如上述配置示例的 Example 的章节所指定的内容的话，是最终在以太坊网络上进行部署。

构建资源

正如 4. 编译合约章节中所提到的那样，构建后的资源存储在`./build/contracts`目录下，以`.sol.js`这样的文件存在。当你编译合约文件，或者在某个网络上运行移植时，Truffle 将会更新这些`.sol.js`的文件，文来包含相关网络的信息。当这些资源在后续中被使用了，比如在前端应用中。它会自动检测当前使用的网络，根据网络自动使用对应的合约资源。

应用部署

因为网络是在运行时自动检测的，这意味着你只需要部署你的应用或前端一次。当你的程序运行时，会动态检测当前使用的网络，以调用合适的资源，这让你的程序非常的灵活。

举例来说，如果你将程序发布到`http://mydapp.io/`，使用钱包浏览器时，你的程序能很好的运行。如果你的钱包浏览器运行在正式网络上，你的 dapp 会使用正式网络上部署的合约，如果是在测试网络上，则对应的使用测试网络的资源。如果任何问题，欢迎留言批评指正。

14. 构建流程

自定义构建流程

纵贯 Truffle 的发展历史看来，默认构造器并不适合每一个人。它有一些明显的缺点，且相比其它构建系统显得不太成熟。由此，Truffle 提供了三种方式，来让你扩展默认的构建系统，但让你能体验到绝大部分的 Truffle 的特性。

执行外部命令

如果你希望在每次触发构建时，执行一个外部命令。可以在项目的配置中包含一个选项。

```
module.exports = {
  // This will run the `webpack` command on each build.
  //
  // The following environment variables will be set when running the command:
  // WORKING_DIRECTORY: root location of the project
  // BUILD_DESTINATION_DIRECTORY: expected destination of built assets
(important for `truffle serve`)
```

```
  // BUILD_CONTRACTS_DIRECTORY: root location of your build contract files
(.sol.js)
  // WEB3_PROVIDER_LOCATION: rpc configuration as a string, as a URL needed for
web3's http provider.
  //
  build: "webpack"
}
```

需要注意的是，你需要提供对应的环境变量，来将这些外部的脚本命令集成进 Truffle，详见配置中的备注。

提供一个自定义的函数

你可以提供了一个自定义的构建函数。框架给提供给你工程相关的参数，方便你与 Truffle 进行深度集成。

```
module.exports = {
  build: function(options, callback) {
    // Do something when a build is required. `options` contains these values:
    //
    // working_directory: root location of the project
    // contracts: metadata about your contract files, code, etc.
    // contracts_directory: root directory of .sol files
    // rpc: rpc configuration defined in the configuration
    // destination_directory: directory where truffle expects the built assets
(important for `truffle serve`)
  }
}
```

创建一个自定义的模块

你也可以通过创建一个模块或对象来实现构建接口(一个包含 `build` 函数的对象，就像上一节中那样)。这适用于需要集成 Truffle，但又有自己的发布流程情况。下面是一个使用默认构建模块的一个例子。

```
var DefaultBuilder = require("truffle-default-builder");
module.exports = {
  build: new DefaultBuilder(...) // specify the default builder configuration
here.
}
```

初始化前端

因为你使用了你自己的构建流程，Truffle 不再知道如何初始化你的前端。下面是一个需要做的事清单：

- 引入 `Web3` 库
- 初始化一个 web3 的实例，设置一个 provider 指向到你的以太坊客户端。检查 `web3` 对象是否已经存在是十分重要的，因为如果有人通过钱包浏览器，比如 Metamask 或 Mist，对象很有可能已存在，这时你应该使用这个对象，而不是初始化一个全新的。查看例子了解更多。
- `require` 或 `import` 编译好的 `sol.js` 文件从 `./build/contracts` 目录。对每个文件需要调用 `MyContract.setProvider()` 来设置 `provider`。这需要与 `web3` 实例使用 `provider` 是一致的。可以使用 `web3.currentProvider` 来获得当前的 `provider`。

```
var MyContract = require("./build/contracts/MyContract.sol.js");
MyContract.setProvider(web3.currentProvider);
```

使用 WEBPACK

我们还在致力于与 Webpack 的紧密集成。可以通过这里，来沟通你的想法。

如果任何问题，欢迎留言批评指正。

15. Truffle 命令指南

使用方式

```
truffle [command] [options]
```

命令

build

构建一个开发中的 app 版本，创建 `.build` 目录。

```
truffle build
```

可选参数

- --dist: 创建一个可发布的 app 版本。仅在使用默认构造器时可用。

查看 6. 构建应用章节来了解更多。

console

运行一个控制台，里面包含已初始化，且随时可用的合约对象。

```
truffle console
```

一旦控制台启去吧，你可以使用通过命令行来使用你的合约，就像代码中那样。另外所有 Truffle 的列在这里的命令都可以在控制台使用。

可选参数：

- --network 名称：指定要使用的网络
- --verbose-rpc：输出 Truffle 与 RPC 通信的详细信息。

其它的 9. 控制台章节来了解更多。

compile

智能编译你的合约，仅会编译自上次编译后修改过的合约，除非另外指定强制刷新。

```
truffle compile
```

可选参数：

- --compile-all: 强制编译所有合约。
- --network 名称：指定使用的网络，保存编译的结果到指定的网络上。

create:contract

工具方法使用脚手架来创建一个新合约。名称需要符合驼峰命名：

```
$ truffle create:contract MyContract
```

create:test

工具方法，使用脚手架来创建一个新的测试方法。名称需要符合驼峰命名。

```
$ truffle create:test MyTest
```

migrate

运行工程的移植。详情见`移植`相关的章节。

```
truffle migrate
```

可选的参数：

- --reset: 从头运行所有的移植。
- --network 名称：指定要使用的网络，并将编译后的资料保存到那个网络。
- --to number：将版本从当前版本移植到序号指定的版本。
- --compile-all: 强制编译所有的合约
- --verbose-rpc：打印 Truffle 与 RPC 交互的详细日志。

exec

在 Truffle 的环境下执行一个 Javascript 文件。环境内包含，web3，基于网络设置的默认 provider，作为全局对象的你的合约对象。这个 Javascript 文件需要 export 一个函数，这样 Truffle 才可以执行。查看 10. 外部脚本来了解更多。

```
$ truffle exec /path/to/my/script.js
```

可选参数：

- --network 名称：名称：指定要使用的网络，并将编译后的资料保存到那个网络。

init

在当前目录下初始化一个全新的 APP，一个全新的工程。会自带默认合约和前端配置。

```
$ truffle init
```

list

列出所有可用的命令，与`--help`等同。

```
truffle list
```

serve

在`http://localhost:8080`提供编译的 app 对应的服务，且在需要的时候自动构建，自动部署。与`truffle watch`类似，区别在于这里增加 web 服务器功能。

```
truffle serve
```

可选参数：

- -p port: 指定 http 服务的端口。默认是 8080。
- --network 名称：名称：指定要使用的网络，并将编译后的资料保存到那个网络。

test

运行所有在`./test`目录下的测试用例。或可选的运行单个测试文件。

```
$ truffle test [/path/to/test/file]
```

可选参数：

- --network 名称：指定要使用的网络，并将编译后的资料保存到那个网络。
- --compile-all: 强制编译所有的合约
- --verbose-rpc：打印 Truffle 与 RPC 交互的详细日志。

version

输出版本号然后退出。

```
truffle version
```

watch

Watch 合约，APP，和配置文件变化，在需要时自动构建 APP。

```
truffle watch
```

如果任何问题，欢迎留言批评指正。