基于 token 的投票（一）

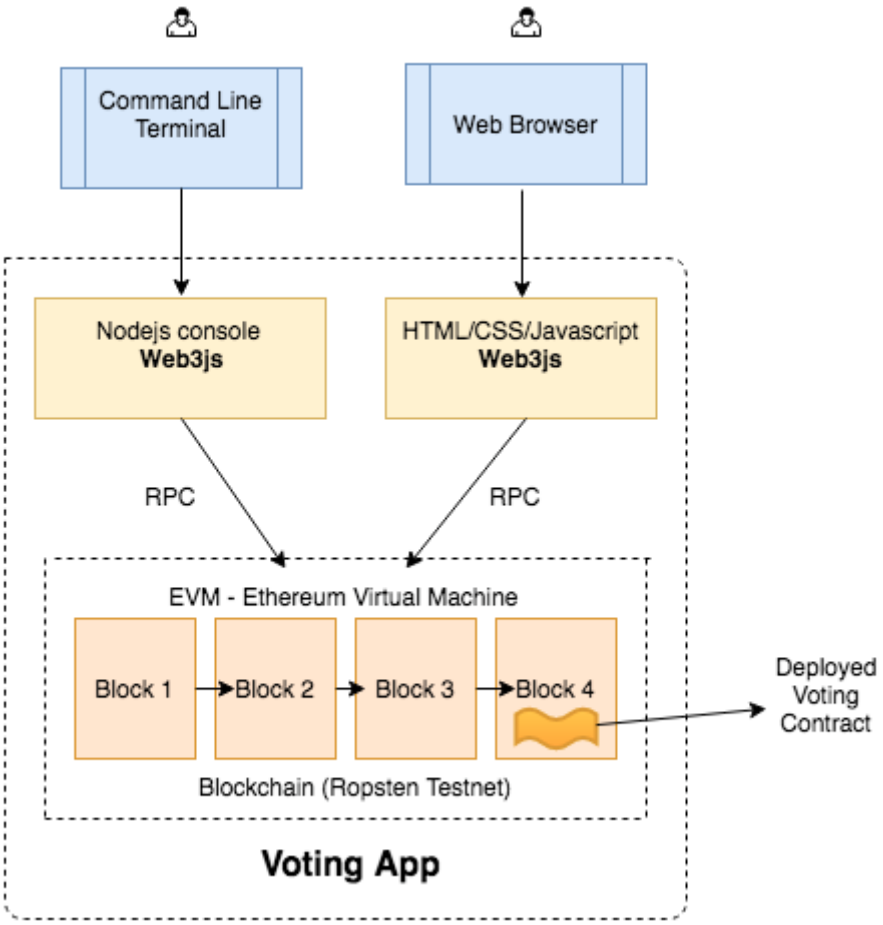
——用 truffle 构建简单投票 DApp

在课程“简单投票 Dapp”中，你已经在一个模拟的区块链（ganache）上实现了一个投票合约，并且成功地通过 nodejs 控制台和网页与合约进行了交互。

在接下来的项目学习中，我们将会实现以下内容：

1. 安装叫做 truffle 的以太坊 dapp 框架，它会被用于编译和部署我们的合约。
2. 在我们之前简单投票 DApp 上做一些小的更新来适配 truffle。
3. 编译合约，并将其部署到自己的测试私链。
4. 通过 truffle 控制台和网页与合约进行交互。
5. 一旦你熟悉 truffle 以后，我们会对合约进行扩展，加入 token 并能够购买 token 的功能。
6. 然后我们会对前端进行扩展，通过网页前端购买 token，并用这些 token 为候选者投票。

这篇文档将主要带领大家完成 1~4 的阶段。



准备工作

用 Geth 启动私链

geth 是用 Go 语言写的一个以太坊客户端，它可以用来连接到以太坊网络。按照之前介绍的方法搭建自己的私链，然后启动（networkid 用自己在 genesis.json 中指定的）：

```
>nohup geth --datadir . --networkid 15 --rpc --rpcapi
db,eth,net,web3,personal,miner  --rpcport 8545 --rpcaddr
127.0.0.1 --rpccorsdomain "*" 2>output.log &
```

来看一下启动 geth 节点时传入参数代表的意思。

--datadir: 指定区块链数据的存储目录，这里我们就在当前目录启动。

--rpc 启用 HTTP-RPC 服务器。

--rpcapi db,eth,net,web3,personal,miner: 基于 HTTP-RPC 接口提供的 API。这是告诉 geth 通过 RPC 接收请求，同时启用我们将会在之后使用的一些 API。

--rpcport 8545 --rpcaddr 127.0.0.1: 这是我们将要用 web3js 库与区块链服务器(geth) 进行通信的服务器主机地址和监听端口。

--rpccorsdomain value 允许跨域请求的域名列表(逗号分隔，浏览器强制)。

注意，课程所提到的节点（node），geth，区块链软件（blockchain software），区块链服务器（blockchain server），客户端（client），实际上指的都是同一个。

如果我们想到直接连接到测试网络，可以用下面的命令：

```
>nohup geth --testnet --syncmode fast --rpc --rpcapi
db,eth,net,web3,personal --cache=1024 --rpcport 8545 --rpcaddr
127.0.0.1 --rpccorsdomain "*" 2>output.log &
```

--testnet: 这是告诉 geth 启动并连接到最新的测试网络。我们所连接的网络是 Ropsten。

--syncmode fast: 我们知道，当用 geth 连接主网或测试网络时，它必须在本地电脑上下载整个区块链。你需要下载完整的区块链并执行每个块里面的每一笔交易，这样你就在本地电脑上拥有了整个历史。这非常耗费时间。不过，也有其他模式或者说优化方法，比如你只需要下载交易收据，而不用执行每一笔交易，这就是“快速”模式。如果我们并不需要整个区块链历史，就可使用这样的 fast 模式同步区块链。

一旦你按照指示启动 geth，它会启动以太坊节点，连接到其他对端节点并开始下载区块链。下载区块链的时间取决于很多因素，比如你的网速，内存，硬盘类型等等。一台 8GB 内存，SSD 硬盘和 10 M 网速的电脑大概需要 7~8 个小时。如果你用快速模式同步 Ropsten，大概需要 6-7 GB 的硬盘空间。

当区块链在同步时，最好知道同步状态，即已经同步了多少块，还有多少块需要同步。可以到 Etherscan 查看当前挖出的最新块。

用 Rinkeby 替换 Ropsten

有些同学在 Ropsten 测试网上运行 geth 会遇到问题。如果耗费时间太长的话，你可以换一个叫做 Rinkeby 的测试网（300 多万个块，下载区块大约 1 个多小时，同步状态大约需要 4~5 个小时，到 Imported new chain segment 即已完成同步）。下面是启动 geth 并同步 Rinkeby 网络的命令。

```
>geth --rinkeby --syncmode "fast" --rpc --rpcapi
db,eth,net,web3,personal --cache=1024 --rpcport 8545 --rpcaddr
127.0.0.1 --rpccorsdomain "*"
```

Full Sync: 从周围节点获取 block headers, block bodies, 并且从初始区块开始重演每一笔交易以验证每一个状态

Fast Sync: 从周围节点获取 block headers, block bodies, 但不会重演交易（只拿 receipts）. 这样就会拿到所有状态的快照（不验证），从此跟全节点一样参与到网络中.

Light Sync: 只拿当前状态（没有历史账本数据）. 如果要验证一笔交易，就必须从另外的全节点处获取历史数据

工作流（Workflow）

如果你正在构建一个基于以太坊的去中心化应用，你的 workflow 可能是像这样：

Development（开发环境）: Ganache

Staging/Testing（模拟/测试环境）: Ropsten, Rinkeby, Kovan or your own private network

Production（生产环境）: Mainnet

Truffle

安装

启动 geth，然后我们来安装 truffle。truffle 是一个 dapp 的开发框架，它可以使得 dapp 的构建和管理非常容易。

你可以像这样使用 npm 安装 truffle：

```
>npm install -g truffle
```

然后我们创建一个空目录，在下面创建 truffle 项目：

```
>mkdir simple_voting_by_truffle_dapp
>cd simple_voting_by_truffle_dapp
>npm install -g webpack
>truffle unbox webpack
```

Unbox 的过程相对会长一点，完成之后应该看到这样的提示：

truffle init: 在当前目录初始化一个新的 truffle 空项目（项目文件只有 truffle-config.js 和 truffle.js；contracts 目录中只有 Migrations.sol；migrations 目录中只有 1_initial_migration.js）

truffle unbox: 直接下载一个 truffle box，即一个预先构建好的 truffle 项目；unbox 的过程相对会长一点，完成之后应该看到这样的提示：

```
Downloading...
Unpacking...
Setting up...
Unbox successful. Sweet!

Commands:

  Compile:              truffle compile
  Migrate:              truffle migrate
  Test contracts:       truffle test
  Run linter:           npm run lint
  Run dev server:       npm run dev
  Build for production: npm run build
```

这里的 webpack 就是一个基于 webpack 构建流程的官方项目框架（truffle box），更多 truffle box 参见 https://truffleframework.com/boxes

webpack: 一个流行的前端资源依赖管理和打包工具。

Truffle 简介

truffle unbox webpack 一条命令由于要下载众多需要的模块，大概耗时 10 分钟左右，所以我们先来了解一下 Truffle。

Truffle 是目前最流行的以太坊 DApp 开发框架，（按照官网说法）是一个世界级的开发环境和测试框架，也是所有使用了 EVM 的区块链的资产管理通道，它基于 JavaScript，致力于让以太坊上的开发变得简单。Truffle 有以下功能：

- 内置的智能合约编译，链接，部署和二进制文件的管理。
- 合约自动测试，方便快速开发。
- 脚本化的、可扩展的部署与发布框架。
- 可部署到任意数量公网或私网的网络环境管理功能
- 使用 EthPM 和 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 的客户端

我们之后写的智能合约必须要部署到链上进行测试，所以 truffle 构建的 DApp 也必须选择一条链来进行部署。我们可以选择部署到一些公共的测试链比如 Rinkeby 或者 Ropsten 上，缺点是部署和测试时间比较长，而且需要花费一定的时间赚取假代币防止 out of gas。当然，对于 DApp 发布的正规流程，staging（模拟环境）还是应该用测试公链的。

还有一种方式就是部署到私链上，这在开发阶段是通常的选择。Truffle 官方推荐使用以下两种客户端：

- Ganache
- truffle develop

Ganache 我们已经接触过了，之前的简单投票小项目就是用它来做模拟区块链的。这里再介绍一点命名背景。它的前身是大名鼎鼎的 testRPC，网上的很多 truffle 教学的老文章里都是用 testRPC。Ganache 是奶油巧克力的意思，而 Truffle 是松露巧克力，一般是以 Ganache 为核，然后上面撒上可可粉，所以这两个产品的名字还是很贴切的。

而 truffle develop 是 truffle 内置的客户端，跟命令行版本的 Ganache 基本类似。在 truffle 目录下 bash 输入：

```
>truffle develop
```

即可开启客户端，和 ganache 一样，它也会给我们自动生成 10 个账户。

唯一要注意的是在 truffle develop 里执行 truffle 命令的时候需要省略前面的“truffle”，比如“truffle compile”只需要敲“compile”就可以了

创建 Voting 项目

初始化一个 truffle 项目时，它会创建运行一个完整 dapp 所有必要的文件和目录。我们直接下载 webpack 这个 truffle box，它里面的目录也是类似的：

```
>ls
README.md       contracts       node_modules      test
webpack.config.js   truffle.js      app          migrations
package.json
>ls app/
index.html javascripts stylesheets
>ls contracts/
ConvertLib.sol MetaCoin.sol Migrations.sol
>ls migrations/
1_initial_migration.js 2_deploy_contracts.js
```

- 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目录。
- migrations/ - 部署脚本文件的存放目录

- test/ - 用来测试应用和合约的测试文件目录
- truffle.js - Truffle 的配置文件

truffle 也会创建一个你可以快速上手的示例应用（在本课程中我们并不会用到该示例应用）。你可以放心地删除项目下面 contracts 目录的 ConvertLib.sol 和 MetaCoin.sol 文件。

```
>rm contracts/ConvertLib.sol contracts/MetaCoin.sol
```

此外，在你的项目目录下查找一个叫做 truffle.js 的配置文件。它里面包含了一个用于开发网络的配置。将端口号从 7545 改为 8545，因为我们的私链及 ganache 默认都会在该端口运行。

Migration

migration 的概念

理解 migrations（迁移）目录的内容非常重要。这些迁移文件用于将合约部署到区块链上。如果你还记得的话，我们在之前的项目中通过在 node 控制台中调用 VotingContract.new 将投票合约部署到区块链上。以后，我们再也不需要这么做了，truffle 将会部署和跟踪所有的部署。

Migrations（迁移）是 JavaScript 文件，这些文件负责暂存我们的部署任务，并且假定部署需求会随着时间推移而改变。随着项目的发展，我们应该创建新的迁移脚本，来改变链上的合约状态。所有运行过的 migration 历史记录，都会通过特殊的迁移合约记录在链上。

第一个迁移 1_initial_migration.js 向区块链部署了一个叫做 Migrations 的合约，并用于存储你已经部署的最新合约。每次你运行 migration 时，truffle 会向区块链查询获取最新已部署好的合约，然后部署尚未部署的任何合约。然后它会更新 Migrations 合约中的 last_completed_migration 字段指向最新部署

的合约。你可以简单地把它当成是一个数据库表，里面有一列 last_completed_migration ，该列总是保持最新状态。

migration 文件的命名有特殊要求：前缀是一个数字（必需），用来标记迁移是否运行成功；后缀是一个描述词汇，只是单纯为了提高可读性，方便理解。

在脚本的开始，我们用 artifacts.require() 方法告诉 truffle 想要进行部署迁移的合约，这跟 node 里的 require 很类似。不过需要注意，最新的官方文档告诫，应该传入定义的合约名称，而不要给文件名称——因为一个.sol 文件中可能包含了多个 contract。

migration js 里的 exports 的函数，需要接收一个 deployer 对象作为第一个参数。这个对象在部署发布的过程中，主要是用来提供清晰的语法支持，同时提供一些通用的合约部署职责，比如保存部署的文件以备稍后使用。deployer 对象是用来暂存(stage)部署任务的主要操作接口。

像所有其它在 Truffle 中的代码一样，Truffle 提供了我们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。这些抽象接口都是部署流程的一部分。

更新 migration 文件

将 `2_deploy_contracts.js` 的内容更新为以下信息：

```
var Voting = artifacts.require("./Voting.sol");
module.exports = function(deployer) {
      deployer.deploy(Voting, ['Alice', 'Bob', 'Cary'], {gas:
                        290000});
};
```

从上面可以看出，部署者希望第一个参数为合约名，跟在构造函数参数后面。在我们的例子中，只有一个参数，就是一个候选者数组。第三个参数是一个哈希，我们用来指定部署代码所需的 gas。gas 数量会随着你的合约大小而变化。对于投票合约， 290000 就足够了。

更新 truffle 配置文件

像下面这样更新 truffle.js 的内容：

```
require('babel-register')
module.exports = {
    networks: {
        development: {
            host: 'localhost',
            port: 8545,
            network_id: '*',
            gas: 470000
            }
    }
}
```

你会注意到，之前的 truffle.js 与我们更新的文件唯一区别在于 gas 选项。这是一个会应用到所有 migration 的全局变量。比如，如果你没有指定 2_deploy_contracts.js gas 值为 290000，migration 就会采用默认值 470000.

合约代码

Voting.sol

之前我们已经完成了编码工作，无须额外改动即可用于 truffle。将文件从 simple_voting_dapp 复制到 contracts 目录即可。

```
>cp ../simple_voting_dapp/Voting.sol contracts/
>ls contracts/
Migrations.sol Voting.sol
```

创建账户（可用 metamask 上账户转币）

在能够部署合约之前，我们需要一个里面有一些以太的账户。当我们用 ganache 的时候，它创建了 10 个测试账户，每个账户里面有 100 个测试以太。但是对于测试网和主网，我们必须自己创建账户，并往里面打一些以太。

在之前的 ganache 应用里，我们曾单独启动了一个 node 控制台，并初始化了 web3 对象。当我们执行 truffle 控制台时，truffle 会帮我们做好所有准备，我们会有一个立即可用的 web3 对象。现在我们有一个账户，地址为 '0x95a94979d86d9c32d1d2ab5ace2dcc8d1b446fa1'（你会得到一个不同的地址），账户余额为 0。

```
>truffle console
// Replace 'verystrongpassword' with a good strong password.
truffle(development)>
web3.personal.newAccount('verystrongpassword')
'0xbaeec91f6390a4eedad8729aea4bf47bf8769b15'
truffle(development)>
web3.eth.getBalance('0xbaeec91f6390a4eedad8729aea4bf47bf8769b15')
{ [String: '0'] s: 1, e: 0, c: [ 0 ] }
truffle(development)>
web3.personal.unlockAccount('0xbaeec91f6390a4eedad8729aea4bf47bf8769b15', 'verystrongpassword', 15000)
```

部署

如果已经有了一些以太，我们就可以继续编译并把合约部署到区块链上。你可以在下面找到相关命令，如果一切顺利，就会出现以下输出。

```
>truffle compile
```

```
Compiling Migrations.sol...Compiling Voting.sol...Writing
artifacts to ./build/contracts
>truffle migrate
Running migration: 1_initial_migration.js
Deploying Migrations...
Migrations: 0x3cee101c94f8a06d549334372181bc5a7b3a8bee
Saving successful migration to network...
Saving artifacts...
Running migration: 2_deploy_contracts.js
Deploying Voting...
Voting: 0xd24a32f0ee12f5e9d233a2ebab5a53d4d4986203
Saving successful migration to network...
Saving artifacts...
```

如果你有多个账户，确保相关账户未被锁定。默认情况，第一个账户 web3.eth.accounts[0] 会用于部署。

可能出现的问题和解决方案

1. 如果由于 gas 不足而部署失败，尝试将 `migrations/2_deploy_contracts.js` 里面的 gas account 增加至 500000。比如：`deployer.deploy(Voting, ['Rama', 'Nick', 'Jose'], {gas: 500000});`
2. 如果你有多个账户，并且更喜欢自选一个账户，而不是 accounts[0]，你可以在 `truffle.js` 中指定想要使用的账户地址。在 network_id 后面添加 `'from: your address'`，truffle 将会使用你指定的地址来部署和交互。

控制台和网页交互

如果部署顺利，你可以通过控制台和网页与合约进行交互。

app/index.html

用之前的 index.html 替换 app/index.html 的内容即可。除了第 40 行包含的 js 文件是 app.js，其他内容与之前基本相同。

在标题<h1>下加 address　

在表格</table>下加 msg　

app/scripts/index.js

新建 JavaScript 文件 app/scripts/index.js

```
// Import the page's CSS. Webpack will know what to do with it.
import "../styles/app.css";
// Import libraries we need.
import { default as Web3} from 'web3';
import { default as contract } from 'truffle-contract'
import voting_artifacts from '../../build/contracts/Voting.json'
var Voting = contract(voting_artifacts);
let candidates = {"Alice": "candidate-1", "Bob": "candidate-2",
"Cary": "candidate-3"}
window.voteForCandidate = function(candidate) {
    let candidateName = $("#candidate").val();
    try {
        $("#msg").html("Vote has been submitted. The vote count
will increment as soon as the vote is recorded on the blockchain.
Please wait.")
        $("#candidate").val("");
        Voting.deployed().then(function(contractInstance) {
            contractInstance.voteForCandidate(candidateName,
                    {gas: 140000,
                    from:web3.eth.accounts[0]})
                    .then(function() {
                      let div_id = candidates[candidateName];
                      return
```

```
                    contractInstance.totalVotesFor
                      .call(candidateName).then(function(v) {
                        $("#" + div_id).html(v.toString());
                        $("#msg").html("");
                      });
                  });
        });
      } catch (err) {
            console.log(err);
      }
}
$( document ).ready(function() {
      if (typeof web3 !== 'undefined') {
            console.warn("Using web3 detected from external
source like Metamask")  // Use Mist/MetaMask's provider
            window.web3 = new Web3(web3.currentProvider);
      } else {
            console.warn("No web3 detected. Falling back to
http://localhost:8545. You should remove this fallback when you
deploy live, as it's inherently insecure. Consider switching to
Metamask for development. More info here:
http://trufflframework.com/tutorials/truffle-and-metamask");
// fallback - use your fallback strategy (local node / hosted node
+ in-dapp id mgmt / fail)
            window.web3 = new Web3(new
                  Web3.providers
                  .HttpProvider("http://localhost:8545"));
      }

      Voting.setProvider(web3.currentProvider);
      let candidateNames = Object.keys(candidates);
      for (var i = 0; i < candidateNames.length; i++) {
      let name = candidateNames[i];
```

```
      Voting.deployed().then(function(contractInstance) {
        contractInstance.totalVotesFor
        .call(name).then(function(v) {
            $("#" + candidates[name])
                .html(v.toString());
          });
      });
    }
});
```

Line 7: 当你编译部署好投票合约时，truffle 会将 abi 和部署好的地址存储到一个 build 目录下面的 json 文件。我们已经在之前讨论了 abi 。我们会用这个信息来启动一个 Voting 抽象。我们将会随后用这个 abstraction 创建一个 Voting 合约的实例。

Line 14: Voting.deployed() 返回一个合约实例。truffle 的每一个调用会返回一个 promise，这就是为什么我们在每一个交易调用时都使用 then().

控制台交互

需要重新打开一个新的 console

```
>truffle console
truffle(default)>
Voting.deployed().then(function(contractInstance)
{contractInstance.voteForCandidate('Alice').then(function(v)
{console.log(v)})})

{ blockHash:
'0x7229f668db0ac335cdd0c4c86e0394a35dd471a1095b8fafb52ebd7671433156',
blockNumber: 469628,
```

```
contractAddress: null,
....
....
truffle(default)>
Voting.deployed().then(function(contractInstance)
{contractInstance.totalVotesFor.call('Alice').then(function(v)
{console.log(v)})})


{ [String: '1'] s: 1, e: 0, c: [ 1] }
```

在调用 voteForCandidate 方法之后需要稍等一下，因为发送交易需要时间；

注意，truffle 的所有调用都会返回一个 promise，这就是为什么会看到每个响应被包装在 then() 函数下面；另外 totalVoteFor() 方法也可以不加.call() 直接调用，不会发送交易。

发出的交易可以在 geth 的 log 输出文件中查到；如果我们连接的是测试网络，可以在 etherscan 上 https://rinkeby.etherscan.io 查询。

可以看到 truffle 默认的 gasPrice 是 100GWei，如果心疼，可以在 truffle.js 中更改，加上 gasPrice: 1000000000 将其改为 1GWei，重启 truffle console 生效。

网页交互

在控制台用 webpack 启动服务器：

```
>npm run dev
```

默认端口 8080，在浏览器访问 localhost:8080 即可看到页面。

如果安装了 metamask，index.js 中会自动检测并使用 metamask 作为 web3 Provider；所以应该注意把 metamask 切换到我们当前连接的网络。

到目前为止，我们已经用 truffle 构建了一个真正的 Dapp。