

# 第 1 章 Ethereum 的架构与设计理论

## 第 2 章 Ethereum 开发工具与环境准备

本章将介绍 Ethereum 正式开发之前的环境准备及工具安装，包括 node.js 环境，Solidity 开发工具 Remix（IDE），智能合约框架 Truffle，智能合约安全库 OpenZeppelin，基于 MetaMask 创建钱包服务，Ethereum 私有化链 ganache 安装（包括 UI 版和 cli 版）等。

为了顺利完成本课程，最好对以下技术已经有一些基本的了解：

- 一种面向对象的开发语言，例如：Java，Python，Golang，Nodejs...
- 前端开发语言：HTML，CSS，JavaScript
- Linux 命令行的使用
- 数据库的基本概念
- 分布式系统的基本概念

### 环境要求

- Nodejs 5.0+
- Windows，Linux 或 MacOS X

课程的所有代码均在 macOS（版本 10.13.3）上测试通过。

## 11.1 安装 nodejs 环境

Ethereum 很多工具的开发都依赖 node , npm , 以下是应该安装的最低版本 :

npm v5.3.0

node v8.3.0

mac 环境下有两种安装方式 :

1, 通过 homebrew 安装

2, 通过安装包安装

这里我们采用具体介绍第一种安装方式 ;

首先需要安装 homebrew , 可以通过 `brew -v` 来看是否安装了 homebrew , 如果能够显示一下信息 , 说明已经安装 ;

```
bogon:~ qingfeng$ brew -v
Homebrew 1.3.9
Homebrew/homebrew-core (git revision f10b7; last commit 2018-04-17)
```

如果没有安装 homebrew , 通过以下命令进行安装 :

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

安装 homebrew 后 , 用以下命令安装 node :

```
brew install node
```

```
bogon:~ qingfeng$ node -v
v7.0.0
```

```
bogon:~ qingfeng$ npm -v
3.10.8
```

如此 , 说明 node 环境安装成功。

因为我安装时间比较久，node 和 npm 的版本都比较老，所以接下来对 node 版本进行升级，升级的方法如下：

```
brew upgrade node
```

```
bogon:~ qingfeng$ node -v
v9.11.1
bogon:~ qingfeng$ npm -v
5.6.0
```

npm 由于源服务器在国外，下载包的速度较慢，所以我们安装一个淘宝镜像工具 cnpm（国内镜像）

```
npm install -g tnpm --registry=https://registry.npm.taobao.org
```

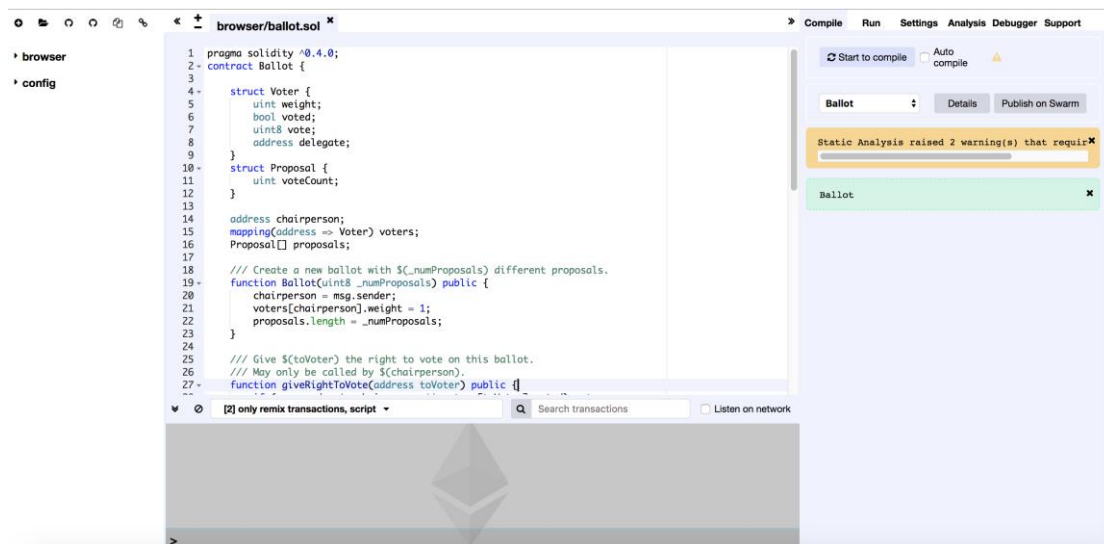
以下是淘宝 npm 镜像网站：

<http://npm.taobao.org/>

## 11.2 Solidity 编辑器：Remix-IDE 安装

Remix 是一款基于浏览器的编译器和 IDE，也是以太坊官方推荐的智能合约开发 IDE，用于 Solidity 语言构建以太坊合约代码，并进行调试和测试。以太坊早起用 Mix 编辑器，目前已经停止更新，项目组也并入了 Remix。

关于 Solidity 语言的讲解以及 Remix 的具体使用在后面章节再讲。



Github 地址：<https://github.com/ethereum/remix-ide>

官方提供的在线编辑地址为：<https://remix.ethereum.org>，用户也可以安装本地运行。

### 方法一：

```
npm install remix-ide -g
```

```
remix-ide
```

通过这种方法安装可能会遇到如下问题：

```
bogon:~ qingfeng$ npm install http-server -g
npm ERR! Darwin 17.4.0
npm ERR! argv "/usr/local/Cellar/node/7.0.0/bin/node" "/usr/local/bin/npm" "install" "http-server" "-g"
npm ERR! node v7.0.0
npm ERR! npm v3.10.8

npm ERR! shasum check failed for /var/folders/nf/7qfcx8rx08lgt8z1940fh3jh0000gn/T/npm-2200-0cdadf4f/registry.npmjs.org/ecstatic/-/ecstatic-3.2.0.tgz
npm ERR! Expected: 1b1aee1ca7c6b99cfb5cf6c9b26b481b90c4409f
npm ERR! Actual:   512ee8cfb43b4793bf70ff28510636d9535da3df
npm ERR! From:     https://registry.npmjs.org/ecstatic/-/ecstatic-3.2.0.tgz
npm ERR!
npm ERR! If you need help, you may report this error at:
npm ERR! <https://github.com/npm/npm/issues>

npm ERR! Please include the following file with any support request:
npm ERR! /Users/qingfeng/npm-debug.log
```

原因是没有安装 node 的 http-server 模块，解决办法如下：

```
npm install http-server -g
```

然后运行remix-ide，成功。

```
bogon:~ qingfeng$ remix-ide
Starting Remix IDE at http://localhost:8080
```

从浏览器中打开 <http://localhost:8080>

### 方法二（源码编译安装）：

```
git clone https://github.com/ethereum/remix-ide.git
cd remix-ide
npm run setupremix # this will clone https://github.com/ethereum/remix for you
and link it to remix-ide
npm install
npm start
```

### 方法三（离线使用）：

#### 下载

wget https://github.com/ethereum/browser-solidity/remix-7013ed1.zip

解压就可以用了。

## 11.3 以太坊开发框架：Truffle 安装

Truffle 是目前最流行的的以太坊开发框架，采用 Javascript 编写，支持智能合约的编译，部署和测试。使用开发框架有助于提升我们的开发效率。

官网地址：<http://truffleframework.com/>

#### 1，安装：

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

#### 2，创建工程

如果想创建一个空的工程，可以用下面的命令：

```
truffle init
```

```
bogon:blockchain qingfeng$ truffle init
```

Downloading...

Error: Something already exists at the destination. Please unbox in an empty folder.

Stopping to prevent overwriting data.

at

```
/usr/local/lib/node_modules/truffle/build/webpack:/~/truffle-box/lib/utils/unbox.js:22:
```

1

at <anonymous>

at process.\_tickCallback (internal/process/next\_tick.js:182:7)

at Function.Module.runMain (internal/modules/cjs/loader.js:697:11)

at startup (internal/bootstrap/node.js:201:19)

at bootstrapNodeJSCore (internal/bootstrap/node.js:516:3)

此问题的原因是执行命令的目录不为空，所以需要在新建的目录执行。

执行完之后会自动下载一些文件到当前的目录：

```
bogon:test qingfeng$ tree
.
├── contracts
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
├── test
├── truffle-config.js
└── truffle.js
```

- contracts 目录中包含 Solidity 合约代码，其中 Migrations.sol 是必须的，其他就是自己写的合约代码。
- migrations 目录中包含合约部署脚本，其中 1\_initial\_migration.js 就是用来部署 Migrations.sol 的，其他的脚本会按照顺序依次执行。

■ test 目录中就是测试代码

在早期版本的 truffle 中，刚刚创建的工程中还会包含 metacoin 的示例代码。新版本 truffle 引入了 box 的概念，所有的示例代码都以 box 的形式提供。因此我们不需要用 truffle init 命令，用下面的命令就可以直接下载 metacoin 的示例代码：

```
truffle unbox metacoin
```

具体可以上 <http://truffleframework.com/boxes/> 进行查看，目前提供了 8 个具体的事例，metacoin 就是其中一个简单的智能合约例子。

## 11.4 以太坊客户端：geth 安装

geth 是以太坊的 go 语言实现版本（go-ethereum），是以太坊协议的实现之一。Geth 可以被安装在 windows，linux，Mac osx，安卓，ios 等操作系统。Geth 实现了以太坊的各种功能，比如新建编辑删除账户，开启挖矿，ether 的转移，智能合约的部署和执行等。

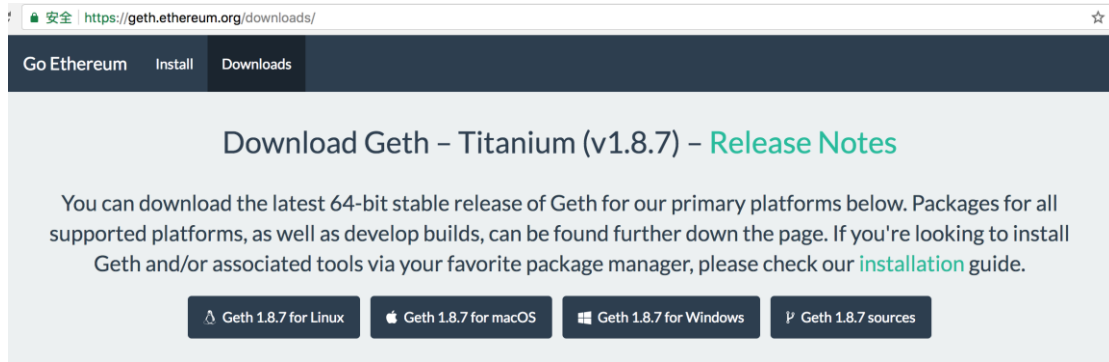
Geth 官网：<https://geth.ethereum.org>

Github 地址：<https://github.com/ethereum/go-ethereum>

### 安装 Go Ethereum

可以根据不同的操作系统平台下载相应的版本进行安装，下载地址：

<https://geth.ethereum.org/downloads/>



其中 macOS 和 Linux 安装很简单，下载解压就可以用，window 下载完成双击安装就行。

也可以从包管理器安装（以 macOS 为例）

通过 HomeBrew 在 macOS 上安装

```
brew tap ethereum/ethereum
```

```
brew install ethereum
```

也可以从源码编译安装，安装步骤如下：

```
git clone https://github.com/ethereum/go-ethereum.git  
  
cd go-ethereum  
  
make geth
```

安装成功之后：

```
Done building.  
Run "/Users/qingfeng/workspace/blockchain/go-ethereum-1.8.10/build/bin/geth" to launch geth.  
bogon:go-ethereum-1.8.10 qingfeng$ cd cmd/
```

```
cd ../go-ethereum-1.8.10/build/bin
```

```
geth -h
```

对于 go 语言开发比较熟悉的朋友，也可以通过 go 命令进行安装：

```
go get -d github.com/ethereum/go-ethereum  
  
go install github.com/ethereum/go-ethereum/cmd/geth
```



安装完成后：geth -h

```
huiqingdembp:bin qingfeng$ ./geth -h
NAME:
  geth - the go-ethereum command line interface

  Copyright 2013-2017 The go-ethereum Authors

USAGE:
  geth [options] command [command options] [arguments...]

VERSION:
  1.8.4-unstable-5909482f

COMMANDS:
  account      Manage accounts
  attach       Start an interactive JavaScript environment (connect to node)
  bug          opens a window to report a bug on the geth repo
  console      Start an interactive JavaScript environment
  copydb       Create a local chain from a target chaindata folder
  dump         Dump a specific block from storage
  dumpconfig   Show configuration values
  export       Export blockchain into file
  export-preimages Export the preimage database into an RLP stream
  import       Import a blockchain file
  import-preimages Import the preimage database from an RLP stream
  init         Bootstrap and initialize a new genesis block
  js           Execute the specified JavaScript files
  license      Display license information
  makecache    Generate ethash verification cache (for testing)
  makedag      Generate ethash mining DAG (for testing)
  monitor      Monitor and visualize node metrics
  removedb     Remove blockchain and state databases
  version      Print version numbers
  wallet       Manage Ethereum presale wallets
  help, h      Shows a list of commands or help for one command

ETHEREUM OPTIONS:
```

其实安装 geth 是非常简单的事情，但是用起来其实是没那么简单的，这里对于 geth 的用法以及各个参数做一个简单的说明。

参考：<https://www.cnblogs.com/tinyxiong/p/7918706.html>

在后面课程中使用的时候，我们再详细讲解。

## 11.5 Ethereum 私有化链：ganache 安装

智能合约必须要部署到链上进行测试。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。

还有一种方式就是部署到私链上，Truffle 官方推荐使用以下两种客户端：

- Ganache ( <https://github.com/trufflesuite/ganache> )
- `truffle develop`

Ganache：创建一个虚拟的以太坊区块链，并生成一些我们将在开发过程中用到的虚拟账号。

Ganache 现在有两个版本，一个是带图形界面的版本，下载地址：

<https://github.com/trufflesuite/ganache/releases>

参见下面的对应关系：

- Windows: Ganache-\*.appx
- Mac: Ganache-\*.dmg
- Linux: Ganache-\*.ApplImage

由于我是用的 Mac，我选择的是 Ganache-1.1.0.dmg，下载到本地，然后双击安装，运行之后：

Ganache				
ACCOUNTS	BLOCKS	TRANSACTIONS	LOGS	SEARCH FOR BLOCK NUMBERS OR TX HASHES
CURRENT BLOCK 0	GAS PRICE 20000000000	GAS LIMIT 6721975	NETWORK ID 5777	RPC SERVER HTTP://127.0.0.1:7545
MINING STATUS AUTOMINING				
MNEMONIC cook funny connect page logic nominee crucial black pyramid slow lens east			HD PATH m/44'/60'/0'/0/account_index	
ADDRESS 0x2EDAcAB0bd06beA74141D3CdEcDb07620F263711	BALANCE 100.00 ETH	TX COUNT 0	INDEX 0	
ADDRESS 0x8EC29c02605c36e997310DafCeeD76AC67DdEf1A	BALANCE 100.00 ETH	TX COUNT 0	INDEX 1	
ADDRESS 0x32Ee44bd97921e7Bc7716FCfD573Cc9A5E54e910	BALANCE 100.00 ETH	TX COUNT 0	INDEX 2	
ADDRESS 0x8171DF58f5E5b2bf379afBe3bDD2E5C7F6e4D018	BALANCE 100.00 ETH	TX COUNT 0	INDEX 3	
ADDRESS 0x51eDF513eE802dc13BaF3c1aF84F6b05A080D13b	BALANCE 100.00 ETH	TX COUNT 0	INDEX 4	
ADDRESS 0x14d21c35976dcdfe36a41ca274AE5Bf8ce4666aA	BALANCE 100.00 ETH	TX COUNT 0	INDEX 5	
ADDRESS 0x0377E5BA2ddE297f654c68E77adfE31215b120fC	BALANCE 100.00 ETH	TX COUNT 0	INDEX 6	

点击右上角的小齿轮可以进行私有链参数设置：

SERVER
ACCOUNTS & KEYS
CHAIN
ADVANCED

HOSTNAME

127.0.0.1

PORT NUMBER

7545

NETWORK ID

5777

AUTOMINE

☐

ERROR ON TRANSACTION FAILURE

☐

The server will accept RPC connections on the following host and port.

Internal blockchain identifier of Ganache server.

Process transactions instantaneously.

When transactions fail, throw an error. If disabled, transaction failures will only be detectable via the "status" flag in the transaction receipt. Disabling this feature will make Ganache handle transaction failures like other Ethereum clients.

CANCEL

RESTART

HOSTNAME

PORT NUMBER

NETWORK ID

点击右上角的 RESTART 重启私有链使配置生效。

另一个就是命令行版本

```
sudo npm install -g ganache-cli
```

具体的命令行参数配置参见 github：

<https://github.com/trufflesuite/ganache-cli>

在后面的 Dapp 开发中还会具体的讲解。

## 11.6 一个构建安全智能合约框架：OpenZeppelin 安装

OpenZeppelin 是一个在 Ethereum 上编写安全智能合约的函数库，里面也提供了兼容 ERC20（见附录 1）标准的智能合约。

Github 地址：

<https://github.com/OpenZeppelin/zeppelin-solidity>

安装 OpenZeppelin

```
$ npm install zeppelin-solidity
```

安装完成后，在目录 node\_modules/ 下会看到 zeppelin-solidity/ 文件夹；在开发代币的时候会用到此库。

最近网上关于 ERC20 代币的漏洞导致项目方损失惨重，如何避免低级别的错误，目前市面上有专门提供解决方案的组织，openzeppelin 就是其中一个，还有其他的组织比如：Experfy，Solidify 等。

## 11.7 基于 MetaMask 创建以太坊轻钱包服务

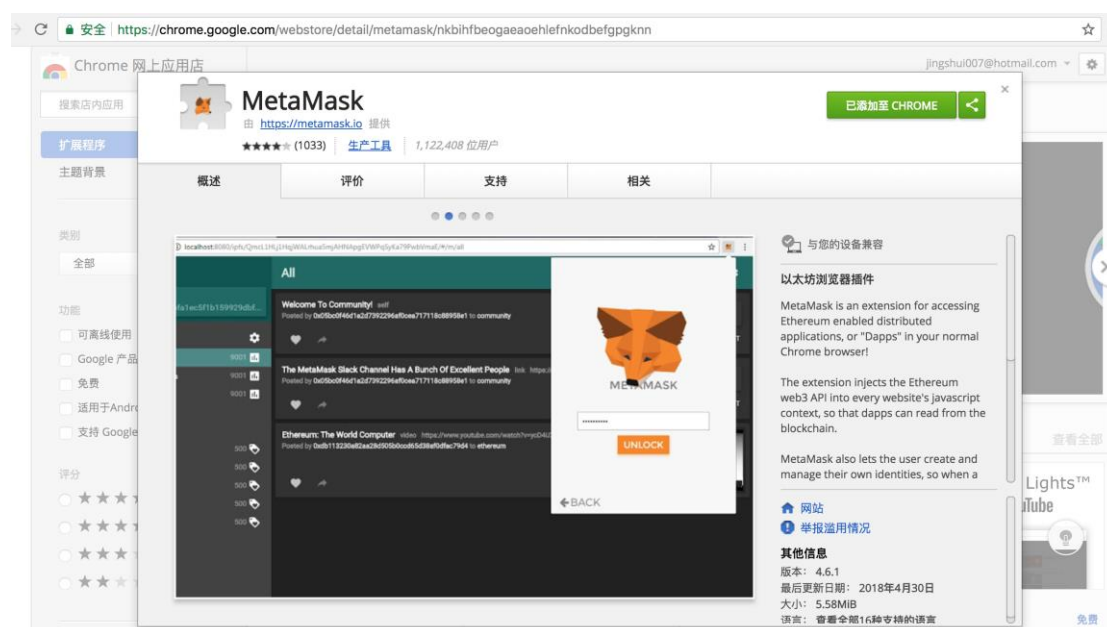
MetaMask 是一个开源的以太坊钱包，能帮助用户方便地管理自己的以太坊数字资产。

### 在线安装：

在这里我用的是 chrome 浏览器

首先点击以下链接安装 MetaMask 插件：

<https://chrome.google.com/webstore/detail/metamask/nkbihfbeogaeaioehlfknodbefgpgknn>



因为我已经安装了，所以右上角显示已添加至 CHROME，没有安装的右上角会显示添加至 CHROME，点击之后弹出一个对话框，点击添加扩展就好了。如果浏览器的右上角出现一个小狐狸的图标说明安装成功了。

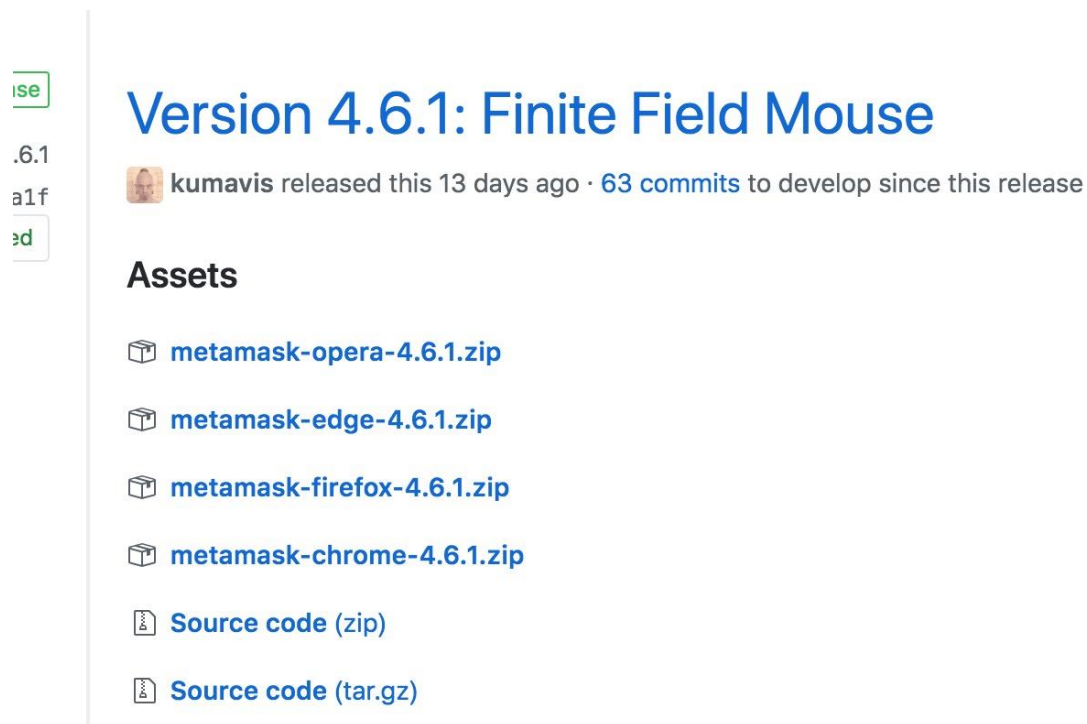


但是，如果你不会科学上网的话，我估计以上的安装方法很难安装成功，接下来介绍一种离线安装的办法。

## 离线安装：

首先，下载插件：

<https://github.com/MetaMask/metamask-extension/releases>

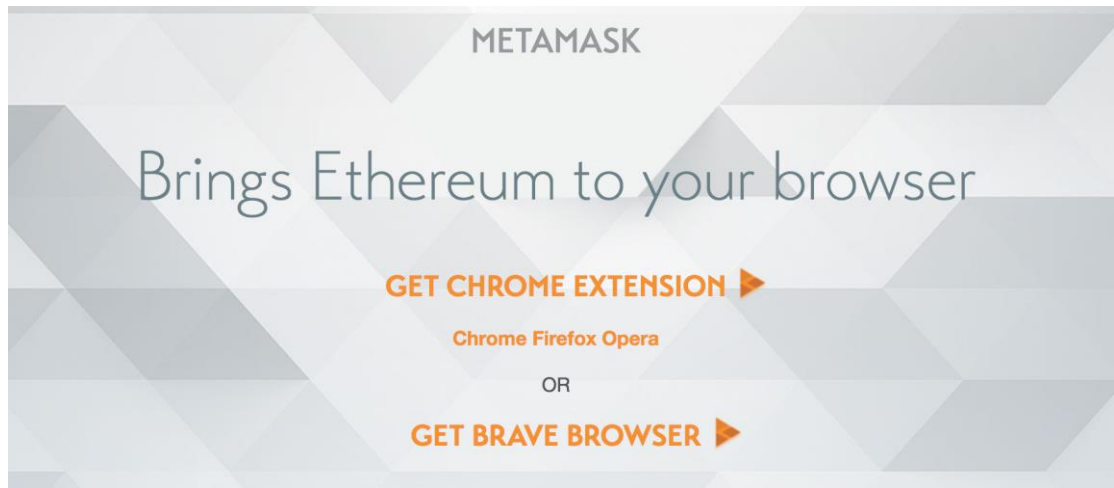


下载 metamask-chrome-4.6.1.zip

然后，用 Chrome 打开链接：chrome://extensions



打开右上角的开发者模式，点击加载已解压的扩展程序，选择刚才下载并解压后的文件；



点击 GET CHROME EXTENSION 进行安装。

以上两种方法都没有成功的，可以打开如下链接：

<https://pan.baidu.com/s/1POsvTZEfi23VCB8aZcaD3g#list/path=%2F>

附录：

1, 什么是 ERC20 token ?

<https://www.cnblogs.com/coinbt/p/8325169.html>

2, ERC721 标准简介

<http://www.aquagemini.com/erc721-non-fungible-token-standard-brief/>

## 第 3 章 智能合约开发语言：Solidity 详解

Solidity 是编写以太坊智能合约的高级语言,专门为以太坊虚拟机( EVM )而生。Solidity 是一门静态语言,同时也是一门像 c++ ,Python 和 JavaScript 的高级语言。它被设计成以编译的方式生成以太坊虚拟机代码。

github 地址：<https://github.com/ethereum/solidity>

此文档以 solidity 0.4.19 版本为基准,不同的版本在语法上可能有一些出入。

### 3.1 从 Hello World 开始

从最基本的开始入手：

Solidity 的代码都包裹在合约里面。一份合约就是以太坊应用的基本模块,所有的变量和函数都属于一份合约,它是你所有应用的起点。

一份名为 HelloWorld 的空合约如下:

```
pragma solidity ^0.4.19;
```



```
contract HelloWorld {  
  
}
```

所有的 solidity 源码都必须加入版本号，比如：pragma solidity ^0.4.19;

## 3.2 状态变量和整数

状态变量是被永久地保存在合约中。被写入以太坊区块链中。

```
pragma solidity ^0.4.19;  
contract HelloWorld {  
    uint myUnsignedInteger = 100; // 被保存在区块链中  
}
```

uint 无符号数据类型，值其值不能是负数，有符号的整数为 int 类型。

Solidity 中，uint 实际上是 uint256 代名词，一个 256 位的无符号整数。也可以定义位数少的 uints — uint8，uint16，uint32 但一般来讲用简单的 uint，除非在某些特殊情况下，在后面会讲到。

有时需要变换数据类型。例如：

```
uint8 a = 5;  
uint b = 6;  
// 将会抛出错误，因为 a * b 返回 uint，而不是 uint8:  
uint8 c = a * b;  
// 我们需要将 b 转换为 uint8:  
uint8 c = a * uint8(b);
```

上面，a \* b 返回类型是 uint，但是当我们尝试用 uint8 类型接收时，就会造成潜在的错误。如果把它的数据类型转换为 uint8，就可以了，编译器也不会出错。

### 3.3 数学运算

加法： $x + y$

减法： $x - y$ ,

乘法： $x * y$

除法： $x / y$

取模 / 求余： $x \% y$  (例如,  $13 \% 5$  余 3, 因为 13 除以 5 , 余 3)

幂方： $x ** y$  (如： $x$  的  $y$  次方)

### 3.4 函数

在 solidity 中函数定义的语法如下：

```
function eatBears(string _name, uint _amount) {  
}
```

这是一个名为 eatBears 的函数，接受两个参数：一个 string 类型和一个 uint 类型。函数体为空。

习惯上函数里的变量都是以( \_ )开头来区别全局变量，虽然这个不是硬性要求，但我们的整个教程都会使用这个习惯。

函数调用：

```
eatBears("songhq", 198)
```

#### 返回值

要想函数返回一个数值，按如下定义：

```
string greeting = "What's up dog";
```

```
function sayHello() public returns (string) {  
    return greeting;  
}
```

Solidity 里，函数的定义里可包含返回值的数据类型(如本例中 string)。

## 函数的修饰符 ( view , pure , modifier )

上面的函数实际上没有改变 Solidity 里的状态，即，它没有改变任何值或者写任何东西。

这种情况下可以把函数定义为 view，意味着它只能读取数据不能更改数据：

```
function sayHello() public view returns (string) {
```

Solidity 还支持 pure 函数，表明这个函数甚至都不访问应用里的数据，例如：

```
function _multiply(uint a, uint b) private pure returns (uint) {  
    return a * b;  
}
```

这个函数甚至都不读取应用里的状态 — 它的返回值完全取决于它的输入参数，在这种情况下可以把函数定义为 pure。

注：可能很难记住何时把函数标记为 pure/view。幸运的是，Solidity 编辑器会给出提示，提醒你使用这些修饰符。

函数修饰符看起来跟函数没什么不同，不过关键字 modifier 告诉编译器，这是个 modifier(修饰符)，而不是个 function(函数)。它不能像函数那样被直接调用，只能被添加到函数定义的末尾，用以改变函数的行为。

咱们仔细读读 onlyOwner：

```
/**  
 * @dev 调用者不是‘主人’，就会抛出异常  
 */  
modifier onlyOwner() {  
    require(msg.sender == owner);  
    _;  
}
```

onlyOwner 函数修饰符是这么用的：

```
contract MyContract is Ownable {  
    event LaughManiacally(string laughter);
```

```
//注意！ `onlyOwner` 上场：
function likeABoss() external onlyOwner {
    LaughManiacally("Muahahahaha");
}
}
```

注意 likeABoss 函数上的 onlyOwner 修饰符。当你调用 likeABoss 时，**首先执行** onlyOwner 中的代码，执行到 onlyOwner 中的 `_;` 语句时，程序再返回并执行 likeABoss 中的代码。

可见，尽管函数修饰符也可以应用到各种场合，但最常见的还是放在函数执行之前添加快速的 require 检查。

因为给函数添加了修饰符 onlyOwner，使得**唯有合约的主人**（也就是部署者）才能调用它。

注意：主人对合约享有的特权当然是正当的，不过也可能被恶意使用。比如，万一，主人添加了个后门，允许他偷走别人的道具呢？

所以非常重要的一点是，部署在以太坊上的 DApp，并不能保证它真正做到去中心化，你需要阅读并理解它的源代码，才能防止其中没有被部署者恶意植入后门；作为开发人员，如何做到既要给自己留下修复 bug 的余地，又要尽量地放权给使用者，以便让他们放心你，从而愿意把数据放在你的 DApp 中，这确实需要个微妙的平衡。

## 带参数的函数修饰符

上一节已经读过一个简单的函数修饰符 `onlyOwner`。函数修饰符也可以带参数。

例如：

// 存储用户年龄的映射

```
mapping (uint => uint) public age;
```

// 限定用户年龄的修饰符

```
modifier olderThan(uint _age, uint _userId) {
    require(age[_userId] >= _age);
    _;
```

```
}
```

// 必须年满 16 周岁才允许开车 (至少在美国是这样的).

// 我们可以用如下参数调用`olderThan` 修饰符:

```
function driveCar(uint _userId) public olderThan(16, _userId) {
```

```
    // 其余的程序逻辑
```

```
}
```

olderThan 修饰符可以像函数一样接收参数，是“宿主”函数 driveCar 把参数传递给它的修饰符的。

## 可支付

截至目前，我们只接触到很少的 函数修饰符。 要记住所有的东西很难，所以我们来个概览：

我们有决定函数何时和被谁调用的可见性修饰符: private 意味着它只能被合约内部调用； internal 就像 private 但是也能被继承的合约调用； external 只能从合约外部调用；最后 public 可以在任何地方调用，不管是内部还是外部。

我们也有状态修饰符， 告诉我们函数如何和区块链交互: view 告诉我们运行这个函数不会更改和保存任何数据； pure 告诉我们这个函数不但不会往区块链写数据，它甚至不从区块链读取数据。这两种在被从合约外部调用的时候都不花费任何 gas（但是它们在被内部其他函数调用的时候将会耗费 gas）。

然后我们有了自定义的 modifiers，例如在第三课学习

的: onlyOwner 和 aboveLevel。 对于这些修饰符我们可以自定义其对函数的约束逻辑。

这些修饰符可以同时作用于一个函数定义上：

```
function test() external view onlyOwner anotherModifier
```

payable 修饰符

payable 方法是让 Solidity 和以太坊变得如此酷的一部分 —— 它们是一种可以接收以太的特殊函数。

先放一下。当你在调用一个普通网站服务器上的 API 函数的时候，你无法用你的函数传送美元——你也不能传送比特币。

但是在以太坊中，因为钱（以太），数据（事务负载），以及合约代码本身都存在于以太坊。你可以在同时调用函数 **并** 付钱给另外一个合约。

这就允许出现很多有趣的逻辑，比如向一个合约要求支付一定的钱来运行一个函数。

来看个例子

```
contract OnlineStore {  
    function buySomething() external payable {  
        // 检查以确定 0.001 以太发送出去来运行函数:  
        require(msg.value == 0.001 ether);  
        // 如果为真，一些用来向函数调用者发送数字内容的逻辑  
        transferThing(msg.sender);  
    }  
}
```

在这里，`msg.value` 是一种可以查看向合约发送了多少以太的方法，另外 `ether` 是一个内建单元。

这里发生的事是，一些人会从 `web3.js` 调用这个函数（从 DApp 的前端），像这样：

// 假设 `OnlineStore` 在以太坊上指向你的合约:

```
OnlineStore.buySomething().send(from: web3.eth.defaultAccount, value:  
web3.utils.toWei(0.001))
```

注意这个 `value` 字段，JavaScript 调用来指定发送多少(0.001)以太。如果把事务想象成一个信封，你发送到函数的参数就是信的内容。添加一个 `value` 很像在信封里面放钱——信件内容和钱同时发送给了接收者。

注意：如果一个函数没标记为 `payable`，而你尝试利用上面的方法发送以太，函数将拒绝你的事务。

提现？

## 函数可见性 ( private , public , internal 和 external )

Solidity 定义的函数的属性默认为公共。这就意味着任何一方 (或其它合约) 都可以调用合约里的函数。

显然,不是什么时候都需要这样,而且这样的合约易于受到攻击。 所以将函数定义为私有是一个好的编程习惯,只有当需要外部世界调用时才将它设置为公共。如何定义一个私有的函数呢?

```
uint[] numbers;
```

```
function _addToArray(uint _number) private {  
    numbers.push(_number);  
}
```

这意味着只有合约中的其它函数才能够调用这个函数,给 numbers 数组添加新成员。

可以看到,在函数名字后面使用关键字 private 即可。和函数的参数类似,私有函数的名字用(下划线)起始。

除 public 和 private 属性之外,Solidity 还使用了另外两个描述函数可见性的修饰词: internal (内部) 和 external (外部)。

internal 和 private 类似,不过,如果某个合约继承自其父合约,这个合约即可以访问父合约中定义的“内部”函数。

external 与 public 类似,只不过这些函数只能在合约之外调用 - 它们不能被合约内的其他函数调用。稍后我们将讨论什么时候使用 external 和 public。

声明函数 internal 或 external 类型的语法,与声明 private 和 public 类型相同:

```
contract Sandwich {  
    uint private sandwichesEaten = 0;  
  
    function eat() internal {  
        sandwichesEaten++;  
    }  
}
```

```

contract BLT is Sandwich {
    uint private baconSandwichesEaten = 0;

    function eatWithBacon() public returns (string) {
        baconSandwichesEaten++;
        // 因为 eat() 是 internal 的，所以我们能在这里调用
        eat();
    }
}

```

## 处理多返回值

```

function multipleReturns() internal returns(uint a, uint b, uint c) {
    return (1, 2, 3);
}

```

```

function processMultipleReturns() external {
    uint a;
    uint b;
    uint c;
    // 这样来做批量赋值:
    (a, b, c) = multipleReturns();
}

```

// 或者如果我们只想返回其中一个变量:

```

function getLastReturnValue() external {
    uint c;
    // 可以对其他字段留空:
    (,c) = multipleReturns();
}

```



## 公有函数和安全性

你必须仔细地检查所有声明为 public 和 external 的函数，一个个排除用户滥用它们的可能，谨防安全漏洞。请记住，如果这些函数没有类似 onlyOwner 这样的函数修饰符，用户能利用各种可能的参数去调用它们。

## 利用‘view’函数节省 Gas

“view” 函数不花 gas

当从外部调用一个 view 函数，是不需要支付一分 gas 的。

这是因为 view 函数不会真正改变区块链上的任何数据 - 它们只是读取。因此用 view 标记一个函数，意味着告诉 web3.js，运行这个函数只需要查询你的本地以太坊节点，而不需要在区块链上创建一个事务（事务需要运行在每个节点上，因此花费 gas）。

稍后我们将介绍如何在自己的节点上设置 web3.js。但现在，你关键是要记住，在所能只读的函数上标记上表示“只读”的“external view 声明，就能为你的玩家减少在 DApp 中 gas 用量。

注意：如果一个 view 函数在另一个函数的内部被调用，而调用函数与 view 函数的不属于同一个合约，也会产生调用成本。这是因为如果主调函数在以太坊创建了一个事务，它仍然需要逐个节点去验证。所以标记为 view 的函数只有在外部调用时才是免费的。

## 3.5 使用结构体和数组

### 结构体

结构体允许你生成一个更复杂的数据类型，它有多个属性。比如；

```
struct Person {  
    uint age;  
    string name;  
}
```

其中，字符串 `string` 用于保存任意长度的 UTF-8 编码数据。如 `string greeting = "Hello World!"`。

创建一个新的 `Person` 结构，然后把它加入到名为 `people` 的数组中。

```
struct Person {  
    uint age;  
    string name;  
}
```

// 创建一个新的 `Person`:

```
Person satoshi = Person(172, "songhq");
```

// 将新创建的 `satoshi` 添加进 `people` 数组:

```
people.push(satoshi);
```

也可以两步并一步，用一行代码更简洁:

```
people.push(Person(16, "songhq"));
```

`array.push()` 在数组的尾部加入新元素，所以元素在数组中的顺序就是我们添加的顺序，如：

```
uint[] numbers;  
numbers.push(5);  
numbers.push(10);  
numbers.push(15);  
// numbers is now equal to [5, 10, 15]
```

## 将结构体作为参数传入

由于结构体的存储指针可以以参数的方式传递给一个 `private` 或 `internal` 的函数，因此结构体可以在多个函数之间相互传递。

遵循这样的语法：

```
function _doStuff(Zombie storage _zombie) internal {  
    // do stuff with _zombie  
}
```

## 数组

建立一个集合可以用数组类型，solidity 支持两种数组：静态数组和动态数组：

```
// 固定长度为 2 的静态数组:
uint[2] fixedArray;

// 固定长度为 5 的 string 类型的静态数组:
string[5] stringArray;

// 动态数组，长度不固定，可以动态添加元素:
uint[] dynamicArray;
```

也可以建立结构体类型的数组，上一节提到的 Person：

```
Person[] people;
```

状态变量被永久保存在区块链中，所以在合约中创建动态数组来保存成结构的数据是非常有意义的。

## 公共数据

可以把数组定义成 public，solidity 会自动创建 getter 方法，语法如下：

```
Person[] public people;
```

其它的合约可以从这个数组读取数据，但是不能写入数据，所以这在合约中是一个有用的保存公共数据的模式。

## 3.6 Keccak256

如何让函数返回一个全(半) 随机的 uint？

Ethereum 内部有一个散列函数 keccak256，它用了 SHA3 版本。一个散列函数基本上就是把一个字符串转换为一个 256 位的 16 进制数字。字符串的一个微小变化会引起散列数据极大变化。

这在 Ethereum 中有很多应用，但是现在我们只是用它造一个伪随机数。

例子：

```
//6e91ec6b618bb462a4a6ee5aa2cb0e9cf30f7a052bb467b0ba58b8748c00d2e5
keccak256("aaaab");
```

```
//b1f078126895a1424524de5321b339ab00408010b7cf0e6ed451514981e58aa9
```

```
keccak256("aaaac");
```

显而易见，输入字符串只改变了一个字母，输出就已经天壤之别了。

注：在区块链中安全地产生一个随机数是一个很难的问题，本例的方法不安全，对于安全性不高的场景，已经很好地满足我们的需要了。

## 用 keccak256 来制造随机数

Solidity 中最好的随机数生成器是 keccak256 哈希函数。

我们可以这样来生成一些随机数

```
// 生成一个 0 到 100 的随机数:
```

```
uint randNonce = 0;
```

```
uint random = uint(keccak256(now, msg.sender, randNonce)) % 100;
```

```
randNonce++;
```

```
uint random2 = uint(keccak256(now, msg.sender, randNonce)) % 100;
```

这个方法首先拿到 now 的时间戳、msg.sender、以及一个自增数 nonce（一个仅会被使用一次的数，这样我们就不会对相同的输入值调用一次以上哈希函数了）。

然后利用 keccak 把输入的值转变为一个哈希值，再将哈希值转换为 uint，然后利用 % 100 来取最后两位，就生成了一个 0 到 100 之间随机数了。

这个方法很容易被不诚实的节点攻击

在以太坊上，当你在和一个合约上调用函数的时候，你会把它广播给一个节点或者在网络上的 transaction 节点们。网络上的节点将收集很多事务，试着成为第一个解决计算密集型数学问题的人，作为“工作证明”，然后将“工作证明”(Proof of Work, PoW)和事务一起作为一个 block 发布在网络上。

一旦一个节点解决了一个 PoW，其他节点就会停止尝试解决这个 PoW，并验证其他节点的事务列表是有效的，然后接受这个节点转而尝试解决下一个节点。

## 这就让我们的随机数函数变得可利用了

我们假设我们有一个硬币翻转合约——正面你赢双倍钱，反面你输掉所有的钱。假如它使用上面的方法来决定是正面还是反面（random >= 50 算正面，random < 50 算反面）。

如果我正运行一个节点,我可以 **只对我自己的节点** 发布一个事务,且不分享它。我可以运行硬币翻转方法来偷窥我的输赢 — 如果我输了,我就不把这个事务包含进我要解决的下一个区块中去。我可以一直运行这个方法,直到我赢得了硬币翻转并解决了下一个区块,然后获利。

所以我们该如何在以太坊上安全地生成随机数呢

因为区块链的全部内容对所有参与者来说是透明的,这就让这个问题变得很难,它的解决方法不在本课程讨论范围,你可以阅读 [这个 StackOverflow 上的讨论](#) 来获得一些主意。一个方法是利用 oracle 来访问以太坊区块链之外的随机数函数。

当然,因为网络上成千上万的以太坊节点都在竞争解决下一个区块,我能成功解决下一个区块的几率非常之低。这将花费我们巨大的计算资源来开发这个获利方法 — 但是如果奖励异常地高(比如我可以在硬币翻转函数中赢得 1 个亿),那就很值得去攻击了。

所以尽管这个方法在以太坊上不安全,在实际中,除非我们的随机函数有一大笔钱在上面,你游戏的用户一般是没有足够的资源去攻击的。

因为在这个教程中,我们只是在编写一个简单的游戏来做演示,也没有真正的钱在里面,所以我们决定接受这个不足之处,使用这个简单的随机数生成函数。但是要谨记它是不安全的。

## 3.8 事件

事件是合约和区块链通讯的一种机制。前端应用“监听”某一些事件,并做出反应。

例子:

事件是合约和区块链通讯的一种机制。前端应用“监听”某些事件,并做出反应。

例子:

// 这里建立事件

```
event IntegersAdded(uint x, uint y, uint result);
```

```
function add(uint _x, uint _y) public {
    uint result = _x + _y;
    //触发事件，通知 app
    IntegersAdded(_x, _y, result);
    return result;
}
```

你的 app 前端可以监听这个事件。JavaScript 实现如下:

```
YourContract.IntegersAdded(function(error, result) {
    // 干些事
})
```

## 3.9 web3.js

后续课程详细介绍

## 3.10 映射 ( Mapping ) 和地址 ( Address )

### Addresses ( 地址 )

以太坊区块链由 account (账户)组成，你可以把它想象成银行账户。一个帐户的余额是以太（在以太坊区块链上使用的币种），你可以和其他帐户之间支付和接受以太币，就像你的银行帐户可以电汇资金到其他银行帐户一样。

每个帐户都有一个“地址”，你可以把它想象成银行账号。这是账户唯一的标识符，它看起来长这样：

```
0xc0910e219cD33BcdE62b77c1b70354233E601e57
```

地址属于特定用户（或智能合约）的。

### Mapping ( 映射 )

映射是另一种在 Solidity 中存储有组织数据的方法。

映射是这样定义的：

```
//对于金融应用程序，将用户的余额保存在一个 uint 类型的变量中：
mapping (address => uint) public accountBalance;
//或者可以用来通过 userId 存储/查找的用户名
mapping (uint => string) userIdToName;
```

映射本质上是存储和查找数据所用的键-值对。在第一个例子中，键是一个 address，值是一个 uint，在第二个例子中，键是一个 uint，值是一个 string。

## 3.11 Msg.sender

在 Solidity 中，有一些全局变量可以被所有函数调用。其中一个就是 msg.sender，它指的是当前调用者（或智能合约）的 address。

注意：在 Solidity 中，功能执行始终需要从外部调用者开始。一个合约只会在区块链上什么也不做，除非有人调用其中的函数。所以 msg.sender 总是存在的。

以下是使用 msg.sender 来更新 mapping 的例子：

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

function setMyNumber(uint _myNumber) public {
    // 更新我们的 `favoriteNumber` 映射来将 `_myNumber` 存储在 `msg.sender`
    名下
    favoriteNumber[msg.sender] = _myNumber;
    // 存储数据至映射的方法和将数据存储在数组相似
}

function whatIsMyNumber() public view returns (uint) {
    // 拿到存储在调用者地址名下的值
    // 若调用者还没调用 setMyNumber，则值为 `0`
    return favoriteNumber[msg.sender];
}
```

在这个小小的例子中，任何人都可以调用 `setMyNumber` 在我们的合约中存下一个 `uint` 并且与他们的地址相绑定。然后，他们调用 `whatIsMyNumber` 就会返回他们存储的 `uint`。

使用 `msg.sender` 很安全，因为它具有以太坊区块链的安全保障——除非窃取与以太坊地址相关联的私钥，否则是没有办法修改其他人的数据的。

## 3.12 Require

`require` 使得函数在执行过程中，当不满足某些条件时抛出错误，并停止执行：

```
function sayHiToSonghq(string _name) public returns (string) {  
    // 比较 _name 是否等于 "Songhq". 如果不成立，抛出异常并终止程序  
    // ( Solidity 并不支持原生的字符串比较, 我们只能通过比较  
    // 两字符串的 keccak256 哈希值来进行判断)  
    require(keccak256(_name) == keccak256("Songhq"));  
    // 如果返回 true, 运行如下语句  
    return "Hi!";  
}
```

如果你这样调用函数 `sayHiToSonghq("Songhq")`，它会返回“Hi！”。而如果调用时使用了其他参数，它则会抛出错误并停止执行。

因此，在调用一个函数之前，用 `require` 验证前置条件是非常有必要的。

## 3.13 继承 ( Inheritance )

有个让 Solidity 的代码易于管理的功能，就是合约 inheritance (继承)：

```
contract Doge {
```



```

function catchphrase() public returns (string) {
    return "So Wow CryptoDoge";
}

contract BabyDoge is Doge {
    function anotherCatchphrase() public returns (string) {
        return "Such Moon BabyDoge";
    }
}

```

由于 BabyDoge 是从 Doge 那里 inherits ( 继承)过来的。这意味着当你编译和部署了 BabyDoge , 它将可以访问 catchphrase() 和 anotherCatchphrase()和其他我们在 Doge 中定义的其他公共函数。

这可以用于逻辑继承 ( 比如表达子类的时候, Cat 是一种 Animal )。 但也可以简单地将类似的逻辑组合到不同的合约中以组织代码。

### 3.14 引入 ( Import )

在 Solidity 中, 当你有多个文件并且想把一个文件导入另一个文件时, 可以使用 import 语句:

```
import "./someothercontract.sol";
```

```

contract newContract is SomeOtherContract {

}

```

这样当我们在合约 ( contract ) 目录下有一个名为 someothercontract.sol 的文件 ( ./ 就是同一目录的意思 ), 它就会被编译器导入。

## 3.15 Storage 与 Memory

在 Solidity 中，有两个地方可以存储变量 —— storage 或 memory。

Storage 变量是指永久存储在区块链中的变量。Memory 变量则是临时的，当外部函数对某合约调用完成时，内存型变量即被移除。可以把它想象成存储在电脑的硬盘或是 RAM 中数据的关系。

大多数时候你都用不到这些关键字，默认情况下 Solidity 会自动处理它们。状态变量（在函数之外声明的变量）默认为“存储”形式，并永久写入区块链；而在函数内部声明的变量是“内存”型的，它们函数调用结束后消失。

然而也有一些情况下，你需要手动声明存储类型，主要用于处理函数内的结构体和数组时：

```
contract SandwichFactory {  
    struct Sandwich {  
        string name;  
        string status;  
    }  
  
    Sandwich[] sandwiches;  
  
    function eatSandwich(uint _index) public {  
        // Sandwich mySandwich = sandwiches[_index];  
  
        // ^ 看上去很直接，不过 Solidity 将会给出警告  
        // 告诉你应该明确在这里定义 `storage` 或者 `memory`。  
  
        // 所以你应该明确定义 `storage`：  
        Sandwich storage mySandwich = sandwiches[_index];  
        // ...这样 `mySandwich` 是指向 `sandwiches[_index]` 的指针  
        // 在存储里，另外...
```

```

mySandwich.status = "Eaten!";
// ...这将永久把 `sandwiches[_index]` 变为区块链上的存储

// 如果你只想要一个副本，可以使用`memory`：
Sandwich memory anotherSandwich = sandwiches[_index + 1];
// ...这样 `anotherSandwich` 就仅仅是一个内存里的副本了
// 另外
anotherSandwich.status = "Eaten!";
// ...将仅仅修改临时变量，对 `sandwiches[_index + 1]` 没有任何影响
// 不过你可以这样做：
sandwiches[_index + 1] = anotherSandwich;
// ...如果你想把副本的改动保存回区块链存储
}
}

```

如果你还没有完全理解究竟应该使用哪一个，也不用担心 —— 在本教程中，我们将告诉你何时使用 `storage` 或是 `memory`，并且当你不得不使用到这些关键字的时候，Solidity 编译器也发警示提醒你的。

现在，只要知道在某些场合下也需要你显式地声明 `storage` 或 `memory` 就够了！

## 3.16 接口

如果我们的合约需要和区块链上的其他的合约会话，则需先定义一个 `interface` (接口)。

先举一个简单的栗子。假设在区块链上有这么一个合约：

```

contract LuckyNumber {
    mapping(address => uint) numbers;

    function setNum(uint _num) public {
        numbers[msg.sender] = _num;
    }
}

```

```

    }

    function getNum(address _myAddress) public view returns (uint) {
        return numbers[_myAddress];
    }
}

```

这是个很简单的合约，你可以用它存储自己的幸运号码，并将其与你的以太坊地址关联。这样其他人就可以通过你的地址查找你的幸运号码了。

现在假设我们有一个外部合约，使用 `getNum` 函数可读取其中的数据。

首先，我们定义 `LuckyNumber` 合约的 `interface`：

```

contract NumberInterface {
    function getNum(address _myAddress) public view returns (uint);
}

```

请注意，这个过程虽然看起来像在定义一个合约，但其实内里不同：

首先，我们只声明了要与之交互的函数——在本例中为 `getNum`——在其中我们没有使用到任何其他的函数或状态变量。

其次，我们并没有使用大括号（`{` 和 `}`）定义函数体，我们单单用分号（`;`）结束了函数声明。这使它看起来像一个合约框架。

编译器就是靠这些特征认出它是一个接口的。

在我们的 `app` 代码中使用这个接口，合约就知道其他合约的函数是怎样的，应该如何调用，以及可期待什么类型的返回值。

## 使用接口

我们可以在合约中这样使用：

```

contract MyContract {
    address NumberInterfaceAddress = 0xab38...;
    // ^ 这是 FavoriteNumber 合约在以太坊上的地址
    NumberInterface numberContract = NumberInterface(NumberInterfaceAddress);
    // 现在变量 `numberContract` 指向另一个合约对象
}

```

```
function someFunction() public {
    // 现在我们可以调用在那个合约中声明的 `getNum` 函数:
    uint num = numberContract.getNum(msg.sender);
    // ...在这儿使用 `num` 变量做些什么
}
}
```

通过这种方式，只要将您合约的可见性设置为 public(公共)或 external(外部)，它们就可以与以太坊区块链上的任何其他合约进行交互。

### 3.17 If 语句

if 语句的语法在 Solidity 中，与在 JavaScript 中差不多：

```
function eatBLT(string sandwich) public {
    // 看清楚了，当我们比较字符串的时候，需要比较他们的 keccak256 哈希码
    if (keccak256(sandwich) == keccak256("BLT")) {
        eat();
    }
}
```

### 3.18 智能协议的永固性

到现在为止，我们讲的 Solidity 和其他语言没有质的区别，它长得也很像 JavaScript.

但是，在有几点以太坊上的 DApp 跟普通的应用程序有着天壤之别。

第一个例子，在你把智能协议传上以太坊之后，它就变得不可更改，这种永固性意味着你的代码永远不能被调整或更新。

你编译的程序会一直，永久的，不可更改的，存在以太网上。这就是 Solidity 代码的安全性如此重要的一个原因。如果你的智能协议有任何漏洞，即使你发现了

也无法补救。你只能让你的用户们放弃这个智能协议，然后转移到一个新的修复后的合约上。

但这恰好也是智能合约的一大优势。代码说明一切。如果你去读智能合约的代码，并验证它，你会发现，一旦函数被定义下来，每一次的运行，程序都会严格遵照函数中原有的代码逻辑一丝不苟地执行，完全不用担心函数被人篡改而得到意外的结果。

外部依赖关系

在编写智能合约的过程中不能硬编码，而要采用“函数”，以便于 DApp 的关键部分可以以参数的形式修改。

## 3.19 OpenZeppelin 库的 Ownable 合约

来自 OpenZeppelin Solidity 库的 Ownable 合约。OpenZeppelin 是主打安全和社区审查的智能合约库，可以在自己的 DApps 中引用。等把这一课学完，不要催我们发布下一课，最好利用这个时间把 OpenZeppelin 的网站看看，保管你会学到很多东西！

```
/**
 * @title Ownable
 * @dev The Ownable contract has an owner address, and provides basic
authorization control
 * functions, this simplifies the implementation of "user permissions".
 */
contract Ownable {
    address public owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed
newOwner);

    /**
```

\* @dev The Ownable constructor sets the original `owner` of the contract to the sender

\* account.

\*/

```
function Ownable() public {  
    owner = msg.sender;  
}
```

/\*\*

\* @dev Throws if called by any account other than the owner.

\*/

```
modifier onlyOwner() {  
    require(msg.sender == owner);  
    _;  
}
```

/\*\*

\* @dev Allows the current owner to transfer control of the contract to a newOwner.

\* @param newOwner The address to transfer ownership to.

\*/

```
function transferOwnership(address newOwner) public onlyOwner {  
    require(newOwner != address(0));  
    OwnershipTransferred(owner, newOwner);  
    owner = newOwner;  
}  
}
```

构造函数：function Ownable()是一个 constructor (构造函数)，构造函数不是必须的，它与合约同名，构造函数一生中唯一的一次执行，就是在合约最初被创建的时候。

函数修饰符：modifier onlyOwner()。 修饰符跟函数很类似，不过是用来修饰其他已有函数用的， 在其他语句执行前，为它检查下先验条件。 在这个例子中，我们就可以写个修饰符 onlyOwner 检查下调用者，确保只有合约的主人才能运行本函数。我们下一章中会详细讲述修饰符，以及那个奇怪的\_;

indexed 关键字：别担心，我们还用不到它。

所以 Ownable 合约基本都会这么干：

合约创建，构造函数先行，将其 owner 设置为 msg.sender ( 其部署者 )

为它加上一个修饰符 onlyOwner，它会限制陌生人的访问，将访问某些函数的权限锁定在 owner 上。

允许将合约所有权转让给他人。

onlyOwner 简直人见人爱，大多数人开发自己的 Solidity DApps，都是从复制/粘贴 Ownable 开始的，从它再继承出的子类，并在之上进行功能开发。

## 3.20 Gas

### Gas——驱动以太坊 DApps 的能源

在 Solidity 中，你的用户想要每次执行你的 DApp 都需要支付一定的 gas，gas 可以用以太坊购买，因此，用户每次跑 DApp 都得花费以太坊。

一个 DApp 收取多少 gas 取决于功能逻辑的复杂程度。每个操作背后，都在计算完成这个操作所需要的计算资源（比如，存储数据就比做个加法运算贵得多），一次操作所需要花费的 gas 等于这个操作背后的所有运算花销的总和。

由于运行你的程序需要花费用户的真金白银，在以太坊中代码的编程语言，比其他任何编程语言都更强调优化。同样的功能，使用笨拙的代码开发的程序，比起经过精巧优化的代码来，运行花费更高，这显然会给成千上万的用户带来大量不必要的开销。

### 为什么要用 gas 来驱动？

以太坊就像一个巨大、缓慢、但非常安全的电脑。当你运行一个程序的时候，网络上的每一个节点都在进行相同的运算，以验证它的输出 —— 这就是所谓的”



去中心化“ 由于数以千计的节点同时在验证着每个功能的运行 ,这可以确保它的数据不会被被监控 , 或者被刻意修改。

可能会有用户用无限循环堵塞网络 , 抑或用密集运算来占用大量的网络资源 , 为了防止这种事情的发生 , 以太坊的创建者为以太坊上的资源制定了价格 , 想要在以太坊上运算或者存储 , 你需要先付费。

注意 : 如果你使用侧链 , 倒是不一定需要付费 , 比如咱们在 Loom Network 上构建的 CryptoZombies 就免费。你不会想要在以太坊主网上玩儿“魔兽世界”吧 ?

- 所需要的 gas 可能会买到你破产。但是你可以找个算法理念不同的侧链来玩它。我们将在以后的课程中咱们会讨论到 , 什么样的 DApp 应该部署在以太坊主链上 , 什么又最好放在侧链。

### **省 gas 的招数 : 结构封装 ( Struct packing )**

我们提到除了基本版的 uint 外 , 还有其他变种 uint : uint8 , uint16 , uint32 等。通常情况下我们不会考虑使用 uint 变种 , 因为无论如何定义 uint 的大小 , Solidity 为它保留 256 位的存储空间。例如 , 使用 uint8 而不是 uint ( uint256 ) 不会为你节省任何 gas。

除非 , 把 uint 绑定到 struct 里面。

如果一个 struct 中有多个 uint , 则尽可能使用较小的 uint , Solidity 会将这些 uint 打包在一起 , 从而占用较少的存储空间。例如 :

```
struct NormalStruct {  
    uint a;  
    uint b;  
    uint c;  
}
```

```
struct MiniMe {  
    uint32 a;  
    uint32 b;  
    uint c;  
}
```

// 因为使用了结构打包, `mini` 比 `normal` 占用的空间更少

```
NormalStruct normal = NormalStruct(10, 20, 30);
```

```
MiniMe mini = MiniMe(10, 20, 30);
```

所以, 当 `uint` 定义在一个 `struct` 中的时候, 尽量使用最小的整数子类型以节约空间。并且把同样类型的变量放一起( 即在 `struct` 中将把变量按照类型依次放置 ), 这样 Solidity 可以将存储空间最小化。例如, 有两个 `struct` :

```
uint c; uint32 a; uint32 b; 和 uint32 a; uint c; uint32 b;
```

前者比后者需要的 `gas` 更少, 因为前者把 `uint32` 放一起了。

## 3.21 时间单位

时间单位

Solidity 使用自己的本地时间单位。

变量 `now` 将返回当前的 unix 时间戳( 自 1970 年 1 月 1 日以来经过的秒数 )。我写这句话时 unix 时间是 1515527488。

注意: Unix 时间传统用一个 32 位的整数进行存储。这会导致“2038 年”问题, 当这个 32 位的 unix 时间戳不够用, 产生溢出, 使用这个时间的遗留系统就麻烦了。

所以, 如果我们想让我们的 DApp 跑够 20 年, 我们可以使用 64 位整数表示时间, 但为此我们的用户又得支付更多的 `gas`。真是两难的设计啊!

Solidity 还包含秒(seconds), 分钟(minutes), 小时(hours), 天(days), 周(weeks) 和 年(years) 等时间单位。它们都会转换成对应的秒数放入 `uint` 中。所以 1 分钟 就是 60, 1 小时是 3600 ( 60 秒×60 分钟 ), 1 天是 86400 ( 24 小时×60 分钟×60 秒 ), 以此类推。

下面是一些使用时间单位的实用案例:

```
uint lastUpdated;
```

```
// 将‘上次更新时间’ 设置为 ‘现在’
```

```
function updateTimestamp() public {
```

```

    lastUpdated = now;
}

// 如果到上次`updateTimestamp` 超过 5 分钟，返回 'true'
// 不到 5 分钟返回 'false'
function fiveMinutesHavePassed() public view returns (bool) {
    return (now >= (lastUpdated + 5 minutes));
}

```

## 3.22 存储非常昂贵

Solidity 使用 storage(存储)是相当昂贵的，”写入“操作尤其贵。

这是因为，无论是写入还是更改一段数据，这都将永久性地写入区块链。”永久性“啊！需要在全世界数千个节点的硬盘上存入这些数据，随着区块链的增长，拷贝份数更多，存储量也就越大。这是需要成本的！

为了降低成本，不到万不得已，避免将数据写入存储。这也会导致效率低下的编程逻辑 - 比如每次调用一个函数，都需要在 memory(内存) 中重建一个数组，而不是简单地将上次计算的数组给存储下来以便快速查找。

在大多数编程语言中，遍历大数据集合都是昂贵的。但是在 Solidity 中，使用一个标记了 external view 的函数，遍历比 storage 要便宜太多，因为 view 函数不会产生任何花销。（gas 可是真金白银啊！）。

我们将在下一章讨论 for 循环，现在我们来看一下看如何如何在内存中声明数组。

### 在内存中声明数组

在数组后面加上 memory 关键字，表明这个数组是仅仅在内存中创建，不需要写入外部存储，并且在函数调用结束时它就解散了。与在程序结束时把数据保存进 storage 的做法相比，内存运算可以大大节省 gas 开销 -- 把这数组放在 view 里用，完全不用花钱。

以下是申明一个内存数组的例子：

```
function getArray() external pure returns(uint[]) {
```

```

// 初始化一个长度为 3 的内存数组
uint[] memory values = new uint[](3);
// 赋值
values.push(1);
values.push(2);
values.push(3);
// 返回数组
return values;
}

```

这个小例子展示了一些语法规则，下一章中，我们将通过一个实际用例，展示它和 for 循环结合的做法。

注意：内存数组 **必须** 用长度参数（在本例中为 3）创建。目前不支持 `array.push()` 之类的方法调整数组大小，在未来的版本可能会支持长度修改。

## For 循环

for 循环的语法在 Solidity 和 JavaScript 中类似。

来看一个创建偶数数组的例子：

```

function getEvens() pure external returns(uint[]) {
    uint[] memory evens = new uint[](5);
    // 在新数组中记录序列号
    uint counter = 0;
    // 在循环从 1 迭代到 10：
    for (uint i = 1; i <= 10; i++) {
        // 如果 `i` 是偶数...
        if (i % 2 == 0) {
            // 把它加入偶数数组
            evens[counter] = i;
            //索引加一，指向下一个空的‘even’
            counter++;
        }
    }
}

```

```
    }  
    return evens;  
}
```

这个函数将返回一个形为 [2,4,6,8,10] 的数组。

### 3.23 提现

学习了如何向合约发送以太，那么在发送之后会发生什么呢？

在你发送以太之后，它将被存储进以合约的以太坊账户中，并冻结在哪里——除非你添加一个函数来从合约中把以太提现。

你可以写一个函数来从合约中提现以太，类似这样：

```
contract GetPaid is Ownable {  
    function withdraw() external onlyOwner {  
        owner.transfer(this.balance);  
    }  
}
```

注意我们使用 Ownable 合约中的 owner 和 onlyOwner，假定它已经被引入了。

你可以通过 transfer 函数向一个地址发送以太，然后 this.balance 将返回当前合约存储了多少以太。所以如果 100 个用户每人向我们支付 1 以太，this.balance 将是 100 以太。

你可以通过 transfer 向任何以太坊地址付钱。比如，你可以有一个函数在 msg.sender 超额付款的时候给他们退钱：

```
uint itemFee = 0.001 ether;  
msg.sender.transfer(msg.value - itemFee);
```

或者在一个有卖家和买家的合约中，你可以把卖家的地址存储起来，当有人买了它的东西的时候，把买家支付的钱发送给它 seller.transfer(msg.value)。

有很多例子来展示什么让以太坊编程如此之酷——你可以拥有一个不被任何人控制的去中心化市场。

## 3.24 以太坊上的代币

让我们来聊聊代币。

如果你对以太坊的世界有一些了解，你很可能听过人们聊到代币——尤其是 ERC20 代币。

一个代币在以太坊基本上就是一个遵循一些共同规则的智能合约——即它实现了所有其他代币合约共享的一组标准函数，例如 `transfer(address _to, uint256 _value)` 和 `balanceOf(address _owner)`。

在智能合约内部，通常有一个映射，`mapping(address => uint256) balances`，用于追踪每个地址还有多少余额。

所以基本上一个代币只是一个追踪谁拥有多少该代币的合约，和一些可以让那些用户将他们的代币转移到其他地址的函数。

它为什么重要呢？

由于所有 ERC20 代币共享具有相同名称的同一组函数，它们都可以以相同的方式进行交互。

这意味着如果你构建的应用程序能够与一个 ERC20 代币进行交互，那么它也能够与任何 ERC20 代币进行交互。这样一来，将来你就可以轻松地将更多的代币添加到你的应用中，而无需进行自定义编码。你可以简单地插入新的代币合约地址，然后哗啦，你的应用程序有另一个它可以使用的代币了。

其中一个例子就是交易所。当交易所添加一个新的 ERC20 代币时，实际上它只需要添加与之对话的另一个智能合约。用户可以让那个合约将代币发送到交易所的钱包地址，然后交易所可以让合约在用户要求取款时将代币发送回给他们。交易所只需要实现这种转移逻辑一次，然后当它想要添加一个新的 ERC20 代币时，只需将新的合约地址添加到它的数据库即可。

其他代币标准

对于像货币一样的代币来说，ERC20 代币非常酷。但是要在我们游戏中代表道具就并不是特别有用。

首先，道具不像货币可以分割——我可以发给你 0.237 以太，但是转移给你 0.237 的道具听起来就有些搞笑。

其次,并不是所有道具都是平等的。你的 2 级道具"Steve"完全不能等同于我 732 级的道具"H4XF13LD MORRIS    ". (你差得远呢, Steve)。

有另一个代币标准更适合如 CryptoZombies 这样的加密收藏品——它们被称为 ERC721 代币。

ERC721 代币是**不能**互换的,因为每个代币都被认为是唯一且不可分割的。你只能以整个单位交易它们,并且每个单位都有唯一的 ID。这些特性正好让我们的道具可以用来交易。

请注意,使用像 ERC721 这样的标准的优势就是,我们不必在我们的合约中实现拍卖或托管逻辑,这决定了玩家能够如何交易 / 出售我们的道具。如果我们符合规范,其他人可以为加密可交易的 ERC721 资产搭建一个交易所平台,我们的 ERC721 道具将可以在该平台上使用。所以使用代币标准相较于使用你自己的交易逻辑有明显的好处。

## 3.25 ERC721 标准,多重继承

让我们来看一看 ERC721 标准:

```
contract ERC721 {  
    event Transfer(address indexed _from, address indexed _to, uint256 _tokenId);  
    event Approval(address indexed _owner, address indexed _approved, uint256  
_tokenId);  
  
    function balanceOf(address _owner) public view returns (uint256 _balance);  
    function ownerOf(uint256 _tokenId) public view returns (address _owner);  
    function transfer(address _to, uint256 _tokenId) public;  
    function approve(address _to, uint256 _tokenId) public;  
    function takeOwnership(uint256 _tokenId) public;  
}
```

这是我们需要实现的方法列表,我们将在接下来的章节中逐个学习。

虽然看起来很多，但不要被吓到了！我们在这里就是准备带着你一步一步了解它们的。

注意：ERC721 目前是一个 *草稿*，还没有正式商定的实现。在本教程中，我们使用的是 OpenZeppelin 库中的当前版本，但在未来正式发布之前它可能会有更改。所以把这 **一个** 可能的实现当作考虑，但不要把它作为 ERC721 代币的官方标准。

### 实现一个代币合约

在实现一个代币合约的时候，我们首先要做的是将接口复制到它自己的 Solidity 文件并导入它，`import ./erc721.sol`。接着，让我们的合约继承它，然后我们用一个函数定义来重写每个方法。

但等一下——ZombieOwnership 已经继承自 ZombieAttack 了——它如何能够也继承于 ERC721 呢？

幸运的是在 Solidity，你的合约可以继承自多个合约，参考如下：

```
contract SatoshiNakamoto is NickSzabo, HalFinney {  
    // 啧啧啧，宇宙的奥秘泄露了  
}
```

正如你所见，当使用多重继承的时候，你只需要用逗号，来隔开几个你想要继承的合约。在上面的例子中，我们的合约继承自 NickSzabo 和 HalFinney。

### balanceOf 和 ownerOf

实现头两个方法：balanceOf 和 ownerOf。

balanceOf

```
function balanceOf(address _owner) public view returns (uint256 _balance);
```

这个函数只需要一个传入 address 参数，然后返回这个 address 拥有多少代币。

ownerOf

```
function ownerOf(uint256 _tokenId) public view returns (address _owner);
```

这个函数需要传入一个代币 ID 作为参数（我们的情况就是一个道具 ID），然后返回该代币拥有者的 address。



同样的，因为在我们的 DApp 里已经有一个 mapping (映射) 存储了这个信息，所以对我们来说这个实现非常直接清晰。我们可以只用一行 return 语句来实现这个函数。

注意：要记得，uint256 等同于 uint。我们从课程的开始一直在代码中使用 uint，但从现在开始我们将在这里用 uint256，因为我们直接从规范中复制粘贴。

### **ERC721:转移标准**

通过学习把所有权从一个人转移给另一个人来继续我们的 ERC721 规范的实现。

注意 ERC721 规范有两种不同的方法来转移代币：

```
function transfer(address _to, uint256 _tokenId) public;
```

```
function approve(address _to, uint256 _tokenId) public;
```

```
function takeOwnership(uint256 _tokenId) public;
```

第一种方法是代币的拥有者调用 transfer 方法，传入他想转移到的 address 和他想转移的代币的 \_tokenId。

第二种方法是代币拥有者首先调用 approve，然后传入与以上相同的参数。接着，该合约会存储谁被允许提取代币，通常存储到一个 mapping (uint256 => address) 里。然后，当有人调用 takeOwnership 时，合约会检查 msg.sender 是否得到拥有者的批准来提取代币，如果是，则将代币转移给他。

你注意到了吗，transfer 和 takeOwnership 都将包含相同的转移逻辑，只是以相反的顺序。（一种情况是代币的发送者调用函数；另一种情况是代币的接收者调用它）。

所以我们把这个逻辑抽象成它自己的私有函数 \_transfer，然后由这两个函数来调用它。这样我们就不用写重复的代码了。

## 3.26 预防溢出

### 合约安全增强：溢出和下溢

我们将来学习你在编写智能合约的时候需要注意的一个主要的安全特性：防止溢出和下溢。

什么是 溢出 (overflow)?

假设我们有一个 uint8, 只能存储 8 bit 数据。这意味着我们能存储的最大数字就是二进制 11111111 (或者说十进制的  $2^8 - 1 = 255$ ).

来看看下面的代码。最后 number 将会是什么值？

```
uint8 number = 255;
```

```
number++;
```

在这个例子中，我们导致了溢出 — 虽然我们加了 1，但是 number 出乎意料地等于 0 了。(如果你给二进制 11111111 加 1，它将被重置为 00000000，就像钟表从 23:59 走向 00:00)。

下溢(underflow)也类似，如果你从一个等于 0 的 uint8 减去 1，它将变成 255 (因为 uint 是无符号的，其不能等于负数)。

虽然我们在这里不使用 uint8，而且每次给一个 uint256 加 1 也不太可能溢出 ( $2^{256}$  真的是一个很大的数了)，在我们的合约中添加一些保护机制依然是非常有必要的，以防我们的 DApp 以后出现什么异常情况。

### 使用 SafeMath

为了防止这些情况，OpenZeppelin 建立了一个叫做 SafeMath 的库(library)，默认情况下可以防止这些问题。

不过在我们使用之前..... 什么叫做库？

一个库是 Solidity 中一种特殊的合约。其中一个有用的功能是给原始数据类型增加一些方法。

比如，使用 SafeMath 库的时候，我们将使用 using SafeMath for uint256 这样的语法。SafeMath 库有四个方法 — add, sub, mul, 以及 div。现在我们可以这样来让 uint256 调用这些方法：

```
using SafeMath for uint256;
```

```
uint256 a = 5;
```

```
uint256 b = a.add(3); // 5 + 3 = 8
```

```
uint256 c = a.mul(2); // 5 * 2 = 10
```

我们将在下一章来学习这些方法 ,不过现在我们先将 SafeMath 库添加进我们的合约。

来看看 SafeMath 的部分代码:

```
library SafeMath {
```

```
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
```

```
        if (a == 0) {
```

```
            return 0;
```

```
        }
```

```
        uint256 c = a * b;
```

```
        assert(c / a == b);
```

```
        return c;
```

```
    }
```

```
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
```

```
        // assert(b > 0); // Solidity automatically throws when dividing by 0
```

```
        uint256 c = a / b;
```

```
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold
```

```
        return c;
```

```
    }
```

```
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
```

```
        assert(b <= a);
```

```
        return a - b;
```

```
    }
```

```

function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
}

```

首先我们有了 `library` 关键字 — 库和合约很相似，但是又有一些不同。就我们的目的而言，库允许我们使用 `using` 关键字，它可以自动把库的所有方法添加给一个数据类型：

```

using SafeMath for uint;
// 这下我们可以为任何 uint 调用这些方法了
uint test = 2;
test = test.mul(3); // test 等于 6 了
test = test.add(5); // test 等于 11 了

```

注意 `mul` 和 `add` 其实都需要两个参数。在我们声明了 `using SafeMath for uint` 后，我们用来调用这些方法的 `uint` 就自动被作为第一个参数传递进去了(在此例中就是 `test`)

我们来看看 `add` 的源代码看 `SafeMath` 做了什么:

```

function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
}

```

基本上 `add` 只是像 `+` 一样对两个 `uint` 相加，但是它用一个 `assert` 语句来确保结果大于 `a`。这样就防止了溢出。

`assert` 和 `require` 相似，若结果为否它就会抛出错误。`assert` 和 `require` 区别在于，`require` 若失败则会返还给用户剩下的 `gas`，`assert` 则不会。所以大部分情况下，你写代码的时候会喜欢 `require`，`assert` 只在代码可能出现严重错误的时候使用，比如 `uint` 溢出。

所以简而言之， SafeMath 的 add , sub , mul , 和 div 方法只做简单的四则运算，然后在发生溢出或下溢的时候抛出错误。

在我们的代码里使用 SafeMath。

为了防止溢出和下溢，我们可以在我们的代码里找 + , - , \* , 或 / , 然后替换为 add, sub, mul, div.

比如，与其这样做：

```
myUint++;
```

我们这样做：

```
myUint = myUint.add(1);
```

太好了，这下我们的 ERC721 实现不会有溢出或者下溢了。

回头看看我们在之前课程写的代码，还有其他几个地方也有可能导致溢出或下溢。

比如，在 ZombieAttack 里面我们有：

```
myZombie.winCount++;
```

```
myZombie.level++;
```

```
enemyZombie.lossCount++;
```

我们同样应该在这些地方防止溢出。（通常情况下，总是使用 SafeMath 而不是普通数学运算是个好主意，也许在以后 Solidity 的新版本里这点会被默认实现，但是现在我们得自己在代码里实现这些额外的安全措施）。

不过我们遇到个小问题 — winCount 和 lossCount 是 uint16，而 level 是 uint32。所以如果我们用这些作为参数传入 SafeMath 的 add 方法。它实际上并不会防止溢出，因为它会把这些变量都转换成 uint256:

```
function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
}
```

// 如果我们在`uint8`上调用`.add`。它将会被转换成`uint256`。

// 所以它不会在  $2^8$  时溢出，因为 256 是一个有效的`uint256`。

这就意味着，我们需要再实现两个库来防止 uint16 和 uint32 溢出或下溢。我们可以将其命名为 SafeMath16 和 SafeMath32。

代码将与 SafeMath 完全相同，除了所有的 uint256 实例都将被替换成 uint32 或 uint16。

我们已经将这些代码帮你写好了，打开 safemath.sol 合约看看代码吧。现在我们需要在 ZombieFactory 里使用它们。

## 3.27 注释

### 注释语法

Solidity 里的注释和 JavaScript 相同。在教程中你已经看到了不少单行注释了：

// 这是一个单行注释，可以理解为给自己或者别人看的笔记

只要在任何地方添加一个 // 就意味着你在注释。如此简单所以你应该经常这么做。



不过我们也知道你的想法：有时候单行注释是不够的。

所以有了多行注释：

```
contract CryptoZombies {  
    /* 这是一个多行注释。  
       对教程有什么建议欢迎提出来。  
    */  
}
```

特别是，最好为你合约中每个方法添加注释来解释它的预期行为。这样其他开发者（或者你自己，在 6 个月以后再回到这个项目中）可以很快地理解你的代码而不需要逐行阅读所有代码。

Solidity 社区所使用的一个标准是使用一种被称作 natspec 的格式，看起来像这样：

```
/// @title 一个简单的基础运算合约  
/// @author H4XF13LD MORRIS    
/// @notice 现在，这个合约只添加一个乘法  
contract Math {  
    /// @notice 两个数相乘
```

```

/// @param x 第一个 uint
/// @param y 第二个 uint
/// @return z (x * y) 的结果
/// @dev 现在这个方法不检查溢出
function multiply(uint x, uint y) returns (uint z) {
    // 这只是个普通的注释，不会被 natspec 解释
    z = x * y;
}
}

```

@title ( 标题 ) 和 @author ( 作者 ) 很直接了。

@notice ( 须知 ) 向 **用户** 解释这个方法或者合约是做什么的。@dev ( 开发者 ) 是向开发者解释更多的细节。

@param ( 参数 ) 和 @return ( 返回 ) 用来描述这个方法需要传入什么参数以及返回什么值。

注意你并不需要每次都用上所有的标签，它们都是可选的。不过最少，写下一个 @dev 注释来解释每个方法是做什么的。

## 第 4 章 智能合约开发框架：Truffle 详解

Truffle 项目 github 地址：<https://github.com/trufflesuite/truffle>

Truffle 官网教程：<http://truffleframework.com/tutorials/>

Truffle 是以太坊智能合约开发测试，以及资源管理框架，旨在让以太坊开发更加简单。通过 Truffle 可以得到：

- 内置智能合约编译，链接，部署以及二进制管理；

- 快速开发下的自动合约测试。
- 脚本化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
- 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。

因为在第二章的时候已经介绍了 Truffle 的安装，所以在这章中就不重复了。本章主要讲解 Truffle 的使用。



## 4.1 创建一个以太坊工程

```
huiqingdembp:myproject qingfeng$ truffle
Truffle v4.1.5 - a development framework for Ethereum

Usage: truffle <command> [options]

Commands:
  init      Initialize new and empty Ethereum project
  compile   Compile contract source files
  migrate   Run migrations to deploy contracts
  deploy    (alias for migrate)
  build     Execute build pipeline (if configuration present)
  test      Run JavaScript and Solidity tests
  debug     Interactively debug any transaction on the blockchain (experimental)
  opcode    Print the compiled opcodes for a given contract
  console   Run a console with contract abstractions and commands available
  develop   Open a console with a local development blockchain
  create    Helper to create new contracts, migrations and tests
  install   Install a package from the Ethereum Package Registry
  publish   Publish a package to the Ethereum Package Registry
  networks  Show addresses for deployed contracts on each network
  watch     Watch filesystem for changes and rebuild the project automatically
  serve     Serve the build directory on localhost and watch for changes
  exec      Execute a JS module within this Truffle environment
  unbox     Download a Truffle Box, a pre-built Truffle project
  version   Show version number and exit

See more at http://truffleframework.com/docs
```

### 4.1.1 创建工程目录

首先在工作目录中新建一个文件夹：

```
mkdir myproject
```

### 4.1.2 初始化工程

```
cd myproject
```

然后执行：truffle init

```

huiqingdembp:myproject qingfeng$ truffle init
Downloading...
Unpacking...
Setting up...
Unbox successful. Sweet!

Commands:

  Compile:      truffle compile
  Migrate:      truffle migrate
  Test contracts: truffle test

```

成功之后，看目录中多了很多的文件：

```

huiqingdembp:myproject qingfeng$ ll
total 16
drwxr-xr-x  7 qingfeng  staff  224  5 23 17:12 ./
drwxr-xr-x 13 qingfeng  staff  416  5 23 17:05 ../
drwxr-xr-x  3 qingfeng  staff   96  5 23 17:12 contracts/
drwxr-xr-x  3 qingfeng  staff   96  5 23 17:12 migrations/
drwxr-xr-x  2 qingfeng  staff   64  5 23 17:12 test/
-rw-r--r--  1 qingfeng  staff  135  5 23 17:12 truffle-config.js
-rw-r--r--  1 qingfeng  staff  135  5 23 17:12 truffle.js

```

contracts/ Truffle 默认的合约文件存放路径；

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

也可以直接用 truffle 提供的现成 box ( demo 程序 ) 框架上进行开发；

目前 truffle 提供的 box 有：

## 4.2 编译合约

参考资料：

调试智能合约

<http://truffleframework.com/tutorials/debugging-a-smart-contract>

智能合约调试指南

<https://zhuanlan.zhihu.com/p/35366945>

用以太坊开发框架 Truffle 开发智能合约实践攻略（代码详解）

<https://mp.weixin.qq.com/s/QF6NYqIKAKDJnZHsA0UNlw>

## 第 5 章 基于 OpenZeppelin 建立标准 代币

宋慧庆 2018/5/30

建立的代币若要能通过以太币钱包收发，必须支持 ERC-20 标准。本教程用 OpenZeppelin 库来简化建立加密代币的过程。

## 5.1 创建项目

```
$ mkdir mic
```

```
$ cd mic
```

```
$ truffle unbox tutorialtoken
```

创建目录 mic ,直接在 truffle 提供的现成 box( tutorialtoken )框架上进行开发。

```
bogon:mic qingfeng$ pwd
/Users/qingfeng/workspace/blockchain/mic
bogon:mic qingfeng$ ll
total 216
drwxr-xr-x  13 qingfeng  staff   416  5 28 16:32 ./
drwxr-xr-x  14 qingfeng  staff   448  5 28 16:30 ../
-rw-r--r--   1 qingfeng  staff  9487  5 28 16:31 box-img-lg.png
-rw-r--r--   1 qingfeng  staff  2426  5 28 16:31 box-img-sm.png
-rw-r--r--   1 qingfeng  staff    68  5 28 16:31 bs-config.json
drwxr-xr-x   3 qingfeng  staff    96  5 28 16:31 contracts/
drwxr-xr-x   3 qingfeng  staff    96  5 28 16:31 migrations/
drwxr-xr-x 216 qingfeng  staff  6912  5 28 16:32 node_modules/
-rw-r--r--   1 qingfeng  staff 80000  5 28 16:32 package-lock.json
-rw-r--r--   1 qingfeng  staff   336  5 28 16:31 package.json
drwxr-xr-x   6 qingfeng  staff   192  5 28 16:31 src/
drwxr-xr-x   3 qingfeng  staff    96  5 28 16:31 test/
-rw-r--r--   1 qingfeng  staff   281  5 28 16:31 truffle.js
bogon:mic qingfeng$
```

## 5.2 什么是 ERC20 标准？

ERC-20 标准是在 2015 年 11 月份推出的，使用这种规则的代币，表现出一种通用的和可预测的方式。简单地说，任何 ERC-20 代币都能立即兼容以太坊钱包（几乎所有支持以太币的钱包，包括 Jaxx、MEW、imToken 等，也支持 erc-20 的代币），由于交易所已经知道这些代币是如何操作的，它们可以很容易地整合这些代币。这就意味着，在很多情况下，这些代币都是可以立即进行交易的。

ERC20 是各个代币的标准接口。ERC20 代币仅仅是以太坊代币的子集。为了充分兼容 ERC20，开发者需要将一组特定的函数（接口）集成到他们的智能合约中，以便在高层面能够执行以下操作：

- 获得代币总供应量
- 获得账户余额
- 转让代币
- 批准花费代币

ERC20 让以太坊区块链上的其他智能合约和去中心化应用之间无缝交互。一些具有部分但非所有 ERC20 标准功能的代币被认为是部分 ERC20 兼容，这还要视其具体缺失的功能而定，但总体是它们仍然很容易与外部交互。

ERC-20 标准还有待完善。其中一个障碍是，将令牌直接发送给令牌的智能合约将导致资金损失。这是因为一个令牌的合约只会跟踪和分配资金。例如，当您从钱包中向另一个用户发送令牌时，该钱包将调用令牌的合约来更新数据库。所以如果您试图将令牌直接传输到令牌的合约中，那么由于该令牌的合约无法响应，所以金钱就“丢失”了。

**ERC20 Token 标准接口：**

```
// https://github.com/ethereum/EIPs/issues/20
contract ERC20 {
    function totalSupply() constant returns (uint totalSupply);
    function balanceOf(address _owner) constant returns (uint balance);
    function transfer(address _to, uint _value) returns (bool success);
    function transferFrom(address _from, address _to, uint _value) returns (bool success);
    function approve(address _spender, uint _value) returns (bool success);
    function allowance(address _owner, address _spender) constant returns (uint remaining);
    event Transfer(address indexed _from, address indexed _to, uint _value);
    event Approval(address indexed _owner, address indexed _spender, uint _value);
}
```

## totalSupply

function totalSupply() constant returns (uint totalSupply)

返回 token 的总供应量

## balanceOf

function balanceOf(address \_owner) constant returns (uint256 balance)

返回地址是\_owner 的账户的账户余额。

## transfer

function transfer(address \_to, uint256 \_value) returns (bool success)

转移\_value 的 token 数量到的地址\_to, 并且必须触发 Transfer 事件。 如果\_from 帐户余额没有足够的令牌来支出, 该函数应该被 throw。

创建新令牌的令牌合同应该在创建令牌时将\_from 地址设置为 0x0 触发传输事件。注意 0 值的传输必须被视为正常传输并触发传输事件。

## transferFrom

function transferFrom(address \_from, address \_to, uint256 \_value) returns (bool success)

从地址\_from 发送数量为\_value 的 token 到地址\_to, 必须触发 Transfer 事件。

transferFrom 方法用于提取工作流，允许合同代您转移 token。这可以用于例如允许合约代您转让代币和/或以子货币收取费用。除了\_from 帐户已经通过某种机制故意地授权消息的发送者之外，该函数应该 throw。

注意 0 值的传输必须被视为正常传输并触发传输事件。

## **approve**

function approve(address \_spender, uint256 \_value) returns (bool success)

允许\_spender 多次取回您的帐户，最高达\_value 金额。如果再次调用此函数，它将以\_value 覆盖当前的余量。

注意：为了阻止向量攻击，客户端需要确认以这样的方式创建用户接口，即将它们设置为 0，然后将其设置为同一个花费者的另一个值。虽然合同本身不应该强制执行，允许向后兼容以前部署的合同兼容性

## **allowance**

function allowance(address \_owner, address \_spender) constant returns (uint256 remaining)

返回\_spender 仍然被允许从\_owner 提取的金额。

## **Transfer**

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

当 token 被转移(包括 0 值)，必须被触发。

## **Approval**

event Approval(address indexed \_owner, address indexed \_spender, uint256 \_value)

当任何成功调用 approve(address \_spender, uint256 \_value)后，必须被触发。

## 5.3 创建智能合约

contracts/ 目录放的是这个项目所包含的所有 solidity 代码。我们在该目录下新建一个 MICToken.sol 的文件。

```
bogon:mic qingfeng$ cd contracts/
bogon:contracts qingfeng$ ll
total 8
drwxr-xr-x  3 qingfeng  staff   96  5 28 16:31 ./
drwxr-xr-x 13 qingfeng  staff  416  5 28 16:32 ../
-rw-r--r--  1 qingfeng  staff  514  5 28 16:31 Migrations.sol
bogon:contracts qingfeng$ vim MICToken.sol
bogon:contracts qingfeng$ ll
total 8
drwxr-xr-x  4 qingfeng  staff  128  5 28 16:59 ./
drwxr-xr-x 13 qingfeng  staff  416  5 28 16:32 ../
-rw-r--r--  1 qingfeng  staff    0  5 28 16:59 MICToken.sol
-rw-r--r--  1 qingfeng  staff  514  5 28 16:31 Migrations.sol
```

然后把以下代码加入其中并保存：

```
pragma solidity ^0.4.2; // 指明目前使用的 solidity 版本 ,不同版本可能会编译出不同的 bytecode
```

```
import 'zeppelin-solidity/contracts/token/ERC20/StandardToken.sol'; // 这是 zeppelin 提供的 ERC20 标准实现
```

```
// contract 关键字类似于其它语言的 class。可以理解为 MICToken 继承了 Contract 类 ,具有智能合约的特性。is 关键字类型于其它语言的 extends ,TutorialToken 继承了 StandardToken 的属性及方法。
```

```
contract MICToken is StandardToken {
    string public name = 'MICToken';        // 设置代币名称
    string public symbol = 'MIC';           // 设置代币代号
    uint8 public decimals = 2;              // 设置代币最小交易单位（精度）
    uint public INITIAL_SUPPLY = 2100000000; // 设置代币发行量
```



```

function MICToken() public {
    totalSupply_ = INITIAL_SUPPLY;
    balances[msg.sender] = INITIAL_SUPPLY;
}
}

```

```

pragma solidity ^0.4.2;

import
'/Users/qingfeng/workspace/blockchain/solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/StandardToken.sol';

contract MICToken is StandardToken {
    string public name = 'MICToken';
    string public symbol = 'MIC';
    uint8 public decimals = 2;
    uint public INITIAL_SUPPLY = 2100000000;

    function MICToken() public {
        totalSupply_ = INITIAL_SUPPLY;
        balances[msg.sender] = INITIAL_SUPPLY;
    }
}

```

首先引入 OpenZeppelin 库的 StandardToken.sol 文件，这里需要注意 import 文件是本地安装的目录才行。然后定义我们的 MICToken 智能合约，继承 StandardToken 智能合约。这样我们定义的 MICToken 智能合约就自动拥有 StandardToken 智能合约的属性及方法，也可以在我们定的智能合约里重写这些属性及方法；

设置我们要发行的代币的属性，包括名字、符号（比特币的符号是 BTC，这里我们定义为 MIC）、精度（最小交易单位，这里我们设置为 2，即最小的交易额为 0.01MIC）、总的发行量（这里我们设置为 2100000000）；

与类名相同的方法为构造函数（是一种特殊的方法，主要功能是初始化一个新建的对象），在这个构造函数中，我们设置代币的总发行量等于我们定义的 INITIAL\_SUPPLY 值，并且将这些代币全部给到部署该智能合约的账号（msg 是一个全局变量，msg.sender 表示调用当前方法的账号）；

这样，我们自己的代币就创建成功了。当然，这只是一个简化的程序，线上的代币都是与金钱挂钩的，会面临巨大的攻击压力，还需要开发非常多的功能及做好健壮的安全工作，在后面的课程中我再详细介绍关于智能合约安全的话题。

接下来修改 app.js 文件

```
cd .. /mic/src/js
```

```
vim app.js
```

把内容替换成以下内容

```
App = {  
  web3Provider: null,  
  contracts: {},  
  
  init: function() {  
    return App.initWeb3();  
  },  
  
  initWeb3: function() {  
    // Initialize web3 and set the provider to the testRPC.  
    if (typeof web3 !== 'undefined') {  
      App.web3Provider = web3.currentProvider;
```

```

        web3 = new Web3(web3.currentProvider);
    } else {
        // set the provider you want from Web3.providers
        App.web3Provider = new Web3.providers.HttpProvider('http://127.0.0.1:9545');
        web3 = new Web3(App.web3Provider);
    }

    return App.initContract();
},

initContract: function() {
    $.getJSON('MICToken.json', function(data) {
        // Get the necessary contract artifact file and instantiate it with truffle-contract.
        var MICTokenArtifact = data;
        App.contracts.MICToken = TruffleContract(MICTokenArtifact);

        // Set the provider for our contract.
        App.contracts.MICToken.setProvider(App.web3Provider);

        // Use our contract to retrieve and mark the adopted pets.
        return App.getBalances();
    });

    return App.bindEvents();
},

bindEvents: function() {
    $(document).on('click', '#transferButton', App.handleTransfer);
},

```

```

handleTransfer: function(event) {

    event.preventDefault();

    var amount = parseInt($('#TTTransferAmount').val());
    var toAddress = $('#TTTransferAddress').val();

    console.log('Transfer ' + amount + ' TT to ' + toAddress);

    var micTokenInstance;

    web3.eth.getAccounts(function(error, accounts) {

        if (error) {

            console.log(error);

        }

        var account = accounts[0];

        App.contracts.MICToken.deployed().then(function(instance) {

            micTokenInstance = instance;

            return micTokenInstance.transfer(toAddress, amount, {from: account});

        }).then(function(result) {

            alert('Transfer Successful!');

            return App.getBalances();

        }).catch(function(err) {

            console.log(err.message);

        });

    });

},

```

```

getBalances: function() {

    console.log('Getting balances...');

    var micTokenInstance;

    web3.eth.getAccounts(function(error, accounts) {

        if (error) {

            console.log(error);

        }

        var account = accounts[0];

        App.contracts.MICToken.deployed().then(function(instance) {

            micTokenInstance = instance;

            return micTokenInstance.balanceOf(account);

        }).then(function(result) {

            balance = result.c[0];

            $('#MICBalance').text(balance);

        }).catch(function(err) {

            console.log(err.message);

        });

    });

};

$(function() {

    $(window).load(function() {

```

```
App.init();  
  
});  
});
```

修改 index.html

cd ../mic/src

vim index.html

替换成以下内容

```
<!DOCTYPE html>  
  
<html lang="en">  
  
  <head>  
  
    <meta charset="utf-8">  
  
    <meta http-equiv="X-UA-Compatible" content="IE=edge">  
  
    <meta name="viewport" content="width=device-width, initial-scale=1">  
  
    <!-- The above 3 meta tags *must* come first in the head; any other head content must come  
*after* these tags -->  
  
    <title>MICToken - Wallet</title>  
  
  
    <!-- Bootstrap -->  
  
    <link href="css/bootstrap.min.css" rel="stylesheet">  
  
  
    <!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media queries -->  
    <!-- WARNING: Respond.js doesn't work if you view the page via file:// -->  
    <!--[if lt IE 9]>  
      <script src="https://oss.maxcdn.com/html5shiv/3.7.3/html5shiv.min.js"> </script>  
      <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"> </script>  
    <![endif]-->  
  
</head>
```

```

<body>

  <div class="container">

    <div class="row">

      <div class="col-xs-12 col-sm-8 col-sm-push-2">

        <h1 class="text-center">MICToken</h1>

        <hr/>

        <br/>

      </div>

    </div>

    <div id="petsRow" class="row">

      <div class="col-sm-6 col-sm-push-3 col-md-4 col-md-push-4">

        <div class="panel panel-default">

          <div class="panel-heading">

            <h3 class="panel-title">My Wallet</h3>

          </div>

          <div class="panel-body">

            <h4>Balance</h4>

            <strong>Balance</strong>: <span id="MICBalance"></span> MIC<br/><br/>

            <h4>Transfer</h4>

            <input type="text" class="form-control" id="TTTransferAddress"
placeholder="Address" />

            <input type="text" class="form-control" id="TTTransferAmount"
placeholder="Amount" />

            <button class="btn btn-primary" id="transferButton"
type="button">Transfer</button>

          </div>

        </div>

      </div>

    </div>

  </div>

```

```
</div>

<!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>
<!-- Include all compiled plugins (below), or include individual files as needed -->
<script src="js/bootstrap.min.js"></script>
<script src="js/web3.min.js"></script>
<script src="js/truffle-contract.js"></script>
<script src="js/app.js"></script>
</body>
</html>
```

修改 package.json

替换以下行（第二行）：

```
"name": "mictoken",
```

修改 package-lock.json

替换以下行（第二行）：

```
"name": "mictoken",
```

## 5.4 编译和部署智能合约

编写完智能合约之后，需要对智能合约进行编译和部署。



## 5.4.1 编译

智能合约是用 Solidity 语言进行开发的，需要编译以后才能在 EVM（以太坊虚拟机）上运行。在项目的根目录运行

```
$ cd mic
```

```
$ truffle compile
```

```
bogon:mic qingfeng$ truffle compile
Compiling ./contracts/MICToken.sol...
Compiling ./contracts/Migrations.sol...
Compiling ../../solidity/node_modules/zeppelin-solidity/contracts/math/SafeMath.sol...
Compiling ../../solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/BasicToken.sol...
Compiling ../../solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/ERC20.sol...
Compiling ../../solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/ERC20Basic.sol...
Compiling ../../solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/StandardToken.sol...

Compilation warnings encountered:

/Users/qingfeng/workspace/blockchain/solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/BasicToken.sol:38:5: Warning: Invoking events without "emit" prefix is deprecated.
    Transfer(msg.sender, _to, _value);
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
/Users/qingfeng/workspace/blockchain/solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/StandardToken.sol:33:5: Warning: Invoking events without "emit" prefix is deprecated.
    Transfer(_from, _to, _value);
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
/Users/qingfeng/workspace/blockchain/solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/StandardToken.sol:49:5: Warning: Invoking events without "emit" prefix is deprecated.
    Approval(msg.sender, _spender, _value);
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
/Users/qingfeng/workspace/blockchain/solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/StandardToken.sol:75:5: Warning: Invoking events without "emit" prefix is deprecated.
    Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
/Users/qingfeng/workspace/blockchain/solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/StandardToken.sol:96:5: Warning: Invoking events without "emit" prefix is deprecated.
    Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Writing artifacts to ./build/contracts
```

编译成功之后进行部署

## 5.4.2 部署（发布合约）

成功编译完智能合约之后，就是部署合约了。

在 migrations/ 目录下，我们可以看到 truffle 框架提供了方便部署合约的脚本。

接下来我们创建 migrations/2\_deploy\_contracts.js 的文件，写入以下内容：

```
1var MICToken = artifacts.require("MICToken");
2module.exports = function(deployer) {
3    deployer.deploy(MICToken);
4};
```

migrations/ 文件夹内的脚本会依照编号的先后顺序执行，例如 2 就会在 1 之后执行，而后面的文字只是为了协助开发者理解使用。

( 代码具体是什么意思需要在这里说明 )

在我们部署智能合约之前，首先需要有一个以太坊区块链测试环境。新开一个命令行工具，输入以下命令：

```
bogon:coin-workspace qingfeng$ ganache-cli
Ganache CLI v6.1.0 (ganache-core: 2.1.0)

Available Accounts
=====
(0) 0xb70cc4036c39ed6ab2596bc264e557b37de05baf
(1) 0xe0bea57c792da18081f37031428ea99ba1e768a3
(2) 0xf884e59a176c81d81dc24678a1c935e32b434c5
(3) 0xb9e7bd0dd114c5a81caf0876a5f5c65acb8f38a3
(4) 0x2f79ee36b885397843dd76a4fd29ebf9c28f3400
(5) 0x1fb59e452c465981d5ef44d194918e469bd92538
(6) 0x23c2ef78e6aed5662621debee5d362b565c6649
(7) 0x2246c2305a2fd3c2d3b99123eb6836fd93194520
(8) 0xd725a6df47631ac1498ab17cbd13f3af097cbbca
(9) 0xe1ff95cbbcd515b18e193010667f749822dead

Private Keys
=====
(0) d7b27831ba4b81da60210acb814fc6817081a0656e42de3cc0b2d82f82948a
(1) c3783448c40e120f8747e04b2f0787d33f21792b591d9e3fd308b806560e4ffa
(2) 01e70265a39b4b2633873feb2134bb86dd44ddc98cf441d45874debad06170e3
(3) 0ebc5019517b5c637efa1c0ddc5ee3264dc6761472c45be8bd4d2f7db0ffb0d3
(4) ecfce89c60ee8639c33d43434e4d9b27ec47d9244f304181b3e67613d0261e9
(5) 3c99344b2293e689fa4f268b3bd273dd1a285fa0a800b163c19790565ed351d3
(6) ebeb52610909a77336edac22d99a096074336469cfffbb8924857fed6528cf81c
(7) 2cdac0789bcf50a834aaceeb21216de01be6b83d0395689df452100d2c4ef430
(8) a2cd3da492af2d8e6340fcc79232dab6765ba5d8e4b406ac5b0dae11ce88e4e2
(9) 8c51e6efa2c365a9dbd1b31d6364986c162ee1408a80b1dee3aa690fbb1e97a0

HD Wallet
=====
Mnemonic:      return odor vessel brave deny clerk minimum cave bottom lion remain rifle
Base HD Path:  m/44'/60'/0'/0/{account_index}

Listening on localhost:8545
```

将会在本地生成以太坊区块链环境，会自动创建 10 个账号（Accounts），以及每个账号对应的私钥（Private Keys），每个账号中拥有 100 个测试用的以太币

(Ether), 我这里本地监听的端口是 8454。这里要留意上面的输出信息 **Mnemonic**, 在与智能合约交互的时候需要用上。

为了与 ganache 连接, 需要配置 `./truffle.js`。如下:

```
1 module.exports = {
2   // See <http://truffleframework.com/docs/advanced/configuration>
3   // for more about customizing your Truffle configuration!
4   networks: {
5     development: {
6       host: "localhost",
7       port: 8545,
8       network_id: "*" // Match any network id
9     }
10  }
11};
```

现在已经可以将智能合约部署到测试的以太坊区块链环境了, 在命令行中输入以下命令:

`cd mic`

```
1$ truffle migrate
```

将会看到类似一下的信息, 说明部署成功。

bogon:coin-workspace qingfeng\$ truffle migrate

Using network 'development'.

Running migration: 1\_initial\_migration.js

Deploying Migrations...

... 0xc36e8c4a7584be4bb555708577db88c77064a9be820fdf2c2f84e58710d6f646

Migrations: 0x6ae87d0ab431f7977c03571d2d97237cce2db708

Saving successful migration to network...

... 0x56c92e1c8adc6e8819b26fbfdd2e9e51750f83a35fb8d16bb33facb78f4f4ace

Saving artifacts...

Running migration: 2\_deploy\_contracts.js

Deploying TutorialToken...

... 0xf3c7b3564780ea2e80457de3c15f3ab67351b0d0e9c9fe7e445781723b5f086f

TutorialToken: 0x406543fec2d1c39c370833a9af13b0114e56bc6d

Saving successful migration to network...

... 0xd2346a80b8b40a73800aa04566bd540fc774e3867e6d9f994de8e70ca62cd26d

Saving artifacts...

```
bogon:mic qingfeng$ truffle migrate
Compiling ./contracts/MICToken.sol...
Compiling ../../solidity/node_modules/zeppelin-solidity/contracts/math/SafeMath.sol...
Compiling ../../solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/BasicToken.sol...
Compiling ../../solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/ERC20.sol...
Compiling ../../solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/ERC20Basic.sol...
Compiling ../../solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/StandardToken.sol...

Compilation warnings encountered:

/Users/qingfeng/workspace/blockchain/solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/BasicToken.sol:38:5: Warning: Invoking events without "emit" prefix is deprecated.
    Transfer(msg.sender, _to, _value);
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
/Users/qingfeng/workspace/blockchain/solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/StandardToken.sol:33:5: Warning: Invoking events without "emit" prefix is deprecated.
    Transfer(_from, _to, _value);
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
/Users/qingfeng/workspace/blockchain/solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/StandardToken.sol:49:5: Warning: Invoking events without "emit" prefix is deprecated.
    Approval(msg.sender, _spender, _value);
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
/Users/qingfeng/workspace/blockchain/solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/StandardToken.sol:75:5: Warning: Invoking events without "emit" prefix is deprecated.
    Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
/Users/qingfeng/workspace/blockchain/solidity/node_modules/zeppelin-solidity/contracts/token/ERC20/StandardToken.sol:96:5: Warning: Invoking events without "emit" prefix is deprecated.
    Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Writing artifacts to ./build/contracts
Using network 'development'.

Running migration: 2_deploy_contracts.js
Deploying MICToken...
... 0xb8bb7719d115e5699276239032646b409b6fa5fb50b00a83db5576558dcd2ee1
MICToken: 0x6554cf5937e6a7b2e457a28eac60580b1d5135a4
Saving successful migration to network...
... 0x5884d72914c13fbaadfabcca50308c5c21e3b3e539787ab00723e957b0baa09c
Saving artifacts...
```

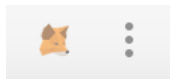
## 5.5 与智能合约进行交互

### 5.5.1 配置 MetaMask

智能合约编译及部署完毕后，就可以通过前端进行与其交互了。

在第 2 章中已经介绍了 MetaMask 以及 Chrome 浏览器插件的安装，这里就不再重复了，直接使用它与智能合约进行交互。

点击浏览器右上角的小狐狸图标



出如下界面：



METAMASK

enter password

LOG IN

[Restore from seed phrase](#)

点击 Restore from send phrase



## RESTORE VAULT

### Wallet Seed

Enter your secret twelve word phrase here to restore your vault.

New Password (min 8 chars)

Confirm Password

CANCEL

OK

在 Wallet Seed 框内填上助记词 ( mnemonic ) , 此助记词是在 ganache 生成测试环境时生成的, 复制填上去。然后下面的框随便输入密码及确认密码后, 点击 OK 继续下一步 :



## ← SETTINGS

**Current RPC** http://localhost:8545

**Save**

**Current Conversion** Updated Thu Apr 19 2018 16:04:05 GMT+0800 (CST)

USD - United States Dollar



State logs contain your public account addresses and sent transactions.

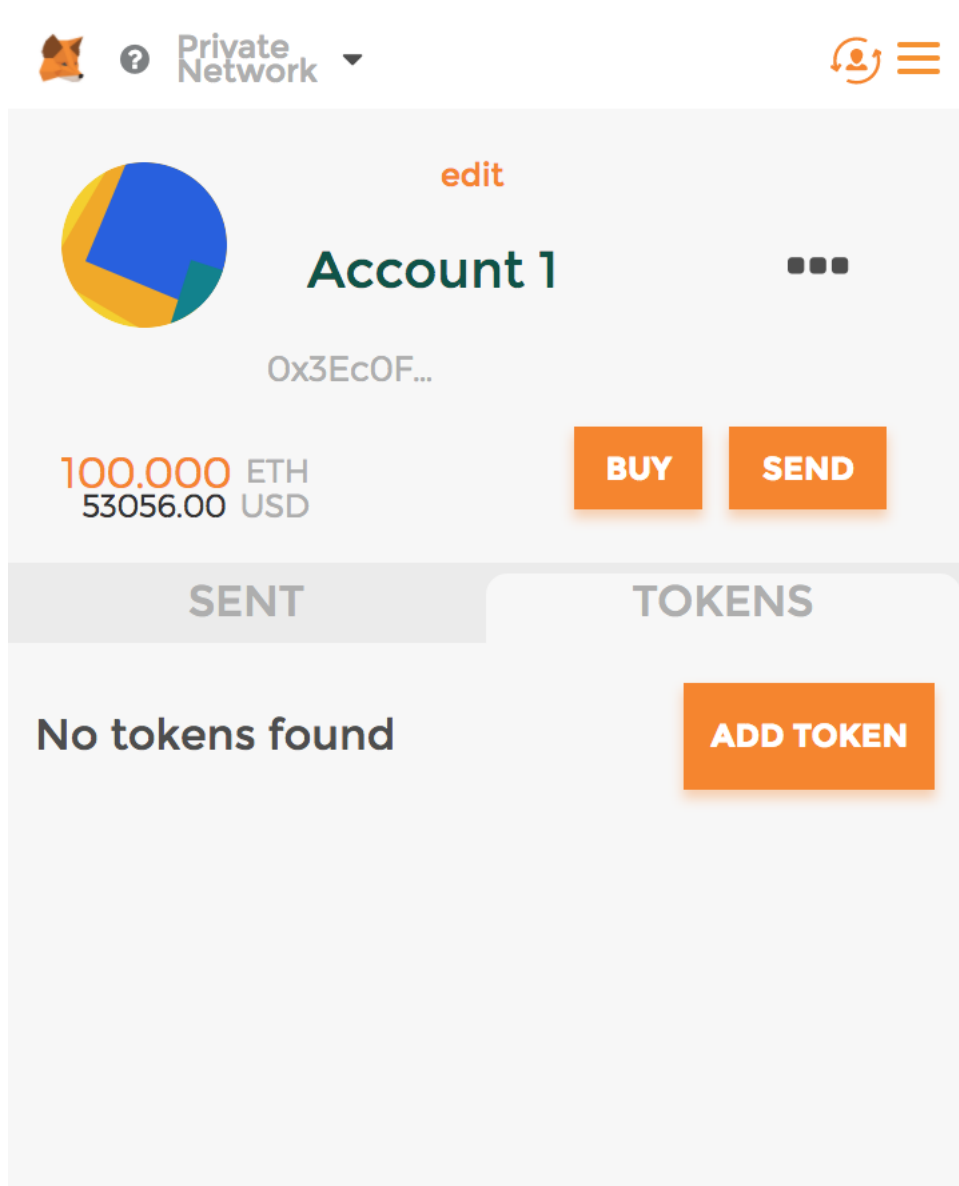
**Download State Logs**

**Reveal Seed Words**

Resetting is for developer use only. [Read more.](#)

**Reset Account**

点击左上角 Main Network 下拉选项框，选择 Custom RPC 进入上述界面，在 Network 框输入 http://localhost:8645 ,该地址的端口要与 ganache 监听的端口一致，点击 SAVE 继续下一步：



这时候你就会看到已经登陆了账号 1 ,其以太币( ETH )少于初始值( 100ETH )。

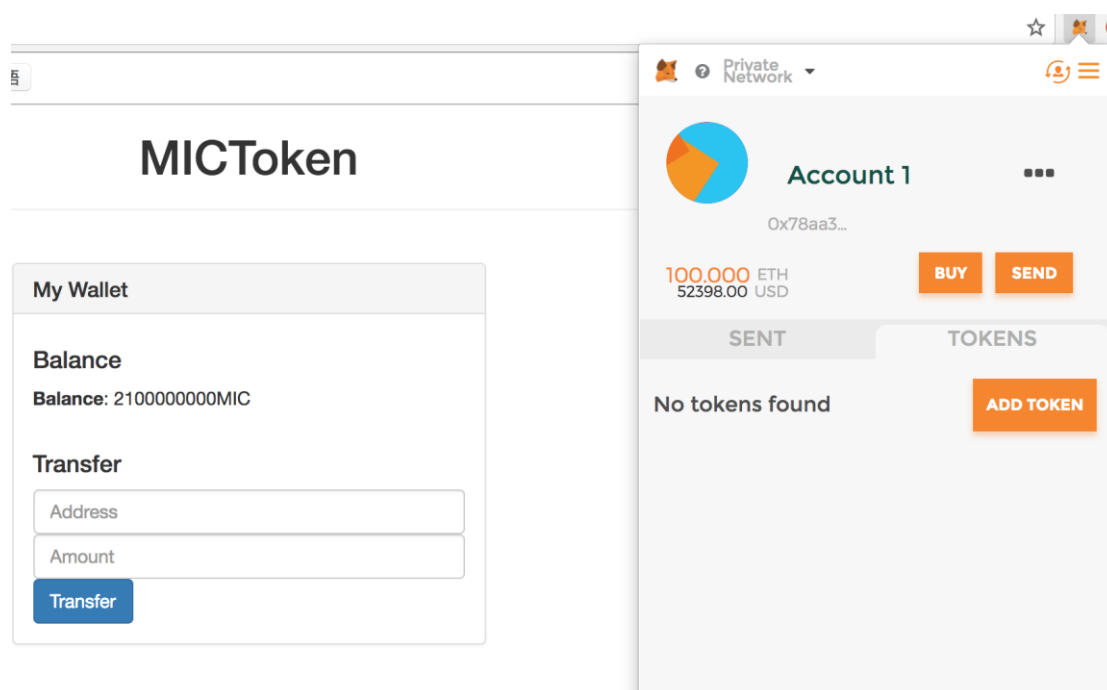
## 5.5.2 使用分布式应用（Dapp）

为了让我们把精力集中到智能合约的开发,此次用的 truffle 的 box 已经包含了分布式应用（Distributed applications），直接拿来用即可。

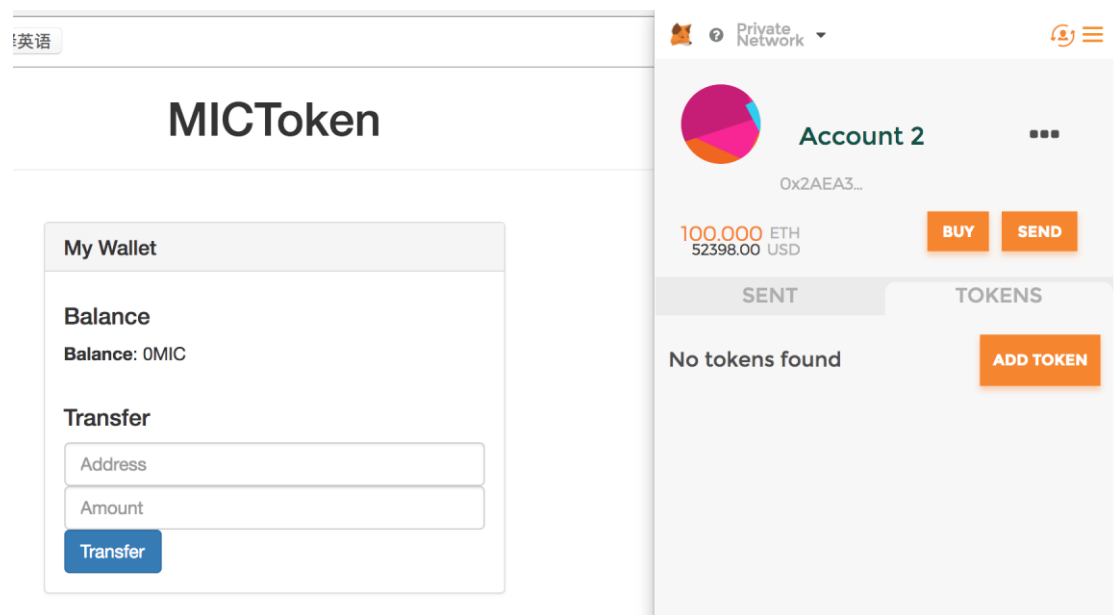
在命令行中输入：

```
1$ npm run dev
```

会自动打开一个新的浏览器窗口，该网页就是一个分布式的运用，如下图所示：

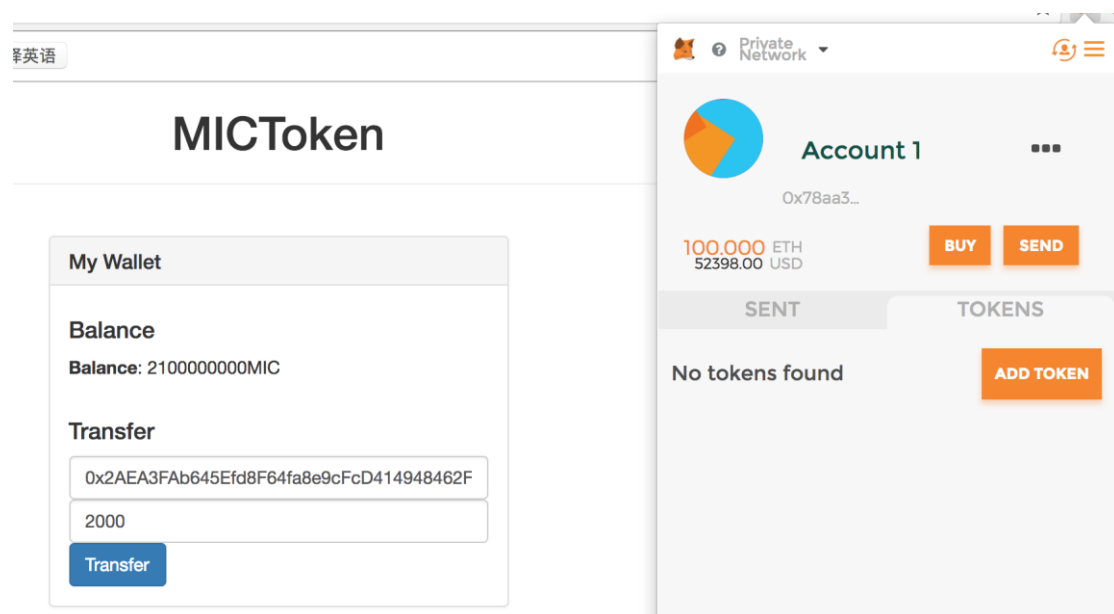


我在 MetaMask 登陆的是 Account1,也就是部署合约的账号,根据合约的规则,初始化时 2100000000 个 MIC 币都是给到部署合约的账号(也即是 Account1),所以在 Balance 一栏可以看到该账号拥有 2100000000 个 MIC 币。此时,我们可以在 MetaMask 中切换为账号 2,再看看其 Balance 值。

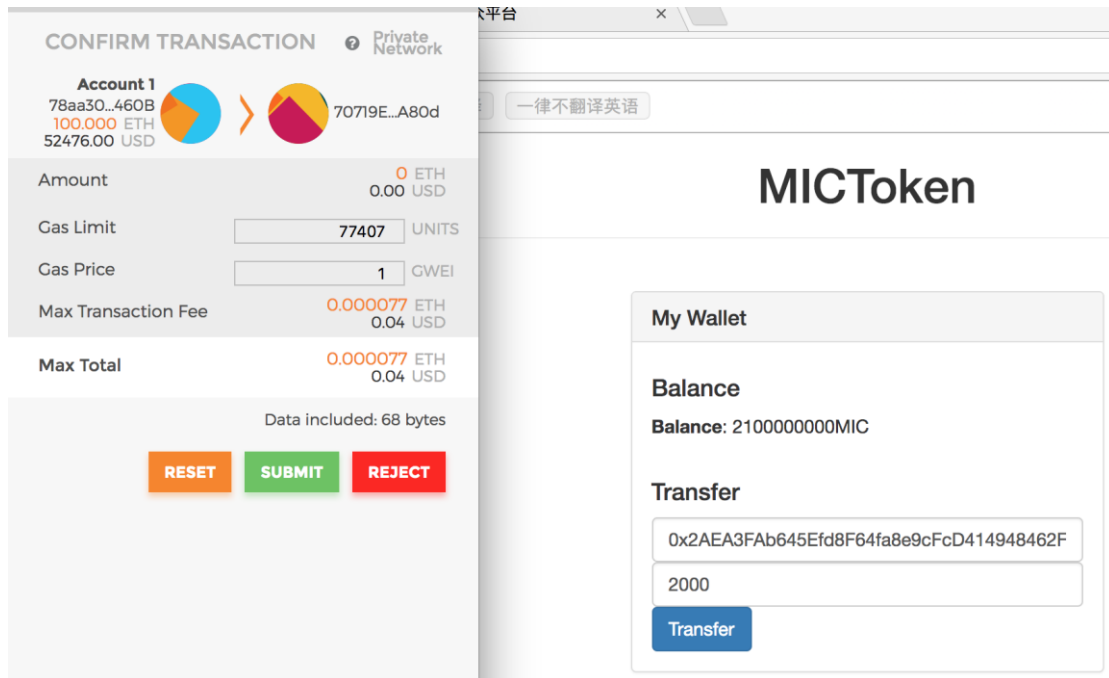


可以看到 Account2 拥有的 MIC 币数量为 0 。其实 ,如果没有进行任何的交易 ,根据合约的设定 ,除了 Account1 一开始有 2100000000 个 MIC 币以外 ,其它账号都是没有 MIC 币的。不信的可以所有账号都切换来看一下。

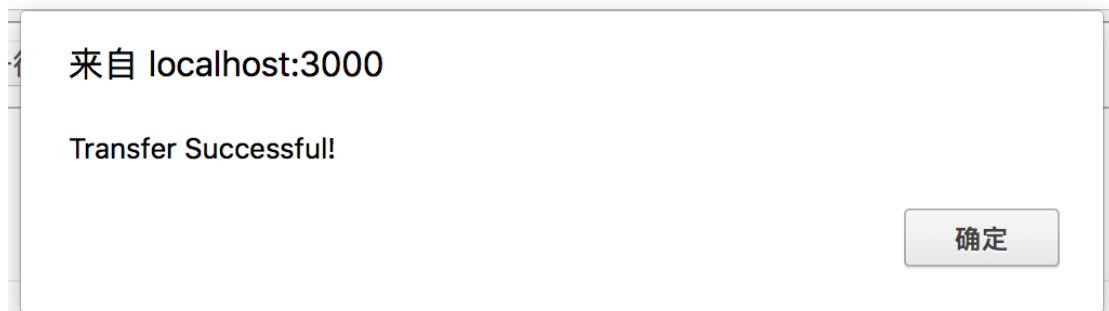
接下来 ,我们模拟一下代币交易。我们从 Account1 账号转出 2000 给 Account2 账号 ( 代币交易的方法是从 StandardToken 合约继承下来的 ) 。



在 Transfer 框输入要发送的地址以及发送的数量 ,然后点击 Transfer 按钮就能完成代币的发送。提交发送的订单 :



发送代币其实就是执行了智能合约，需要消耗 Gas 。点击 SUBMIT



**My Wallet**

**Balance**  
**Balance:** 2100000000MIC

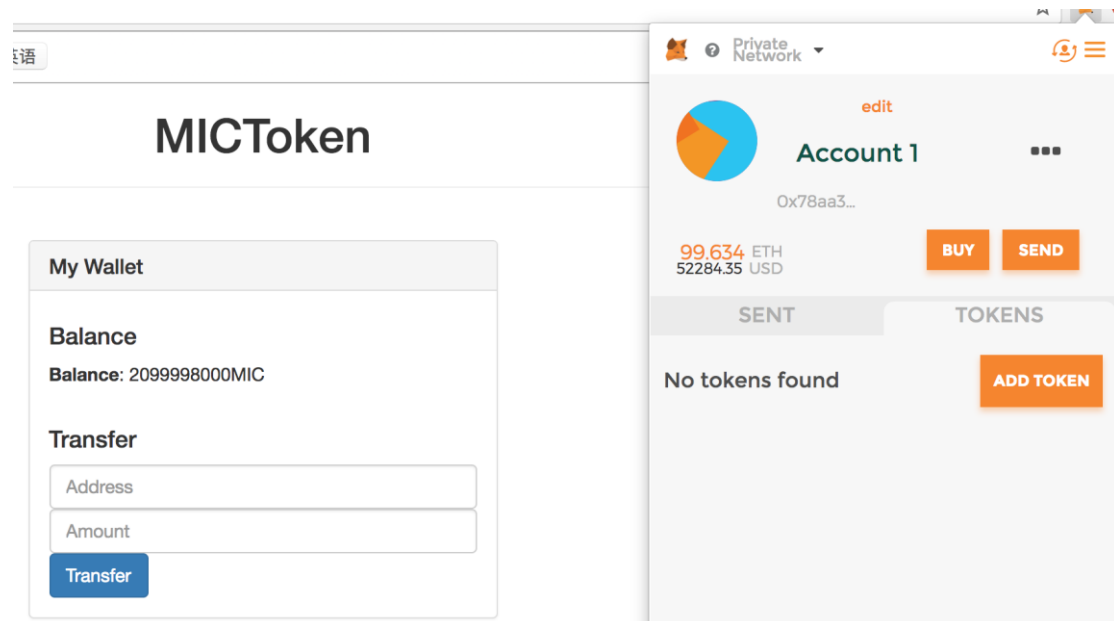
**Transfer**  

0x2AEA3FAb645Efd8F64fa8e9cFcD414948462F

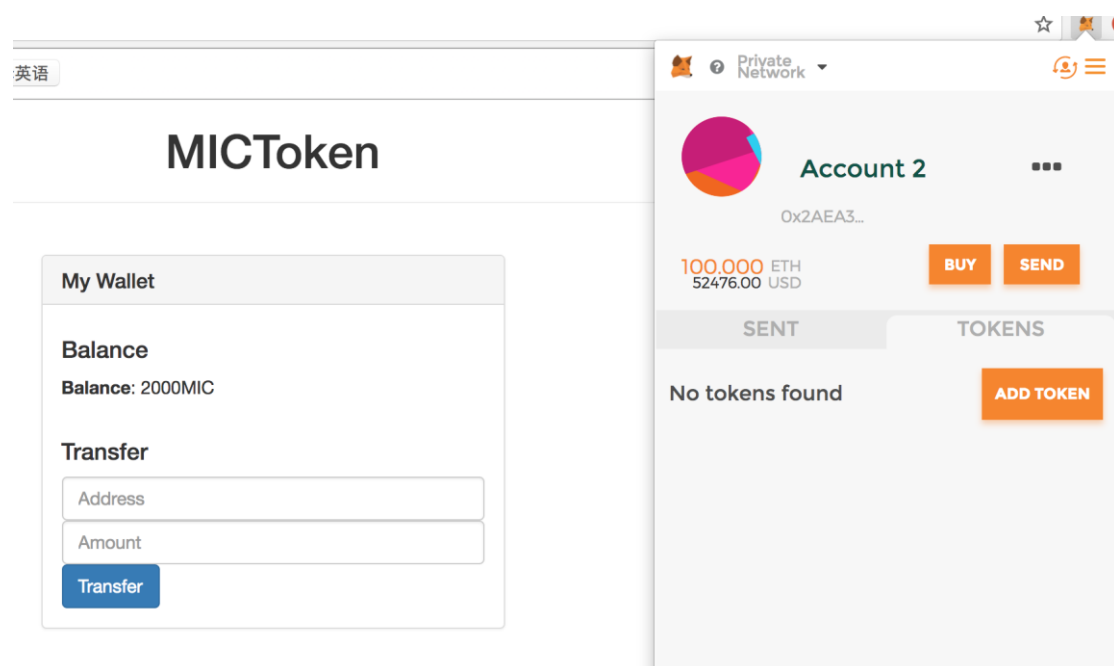
2000

Transfer

可以看到，代币发送后 Account1 账号 MIC 币的余额为 2099998000，右边 SENT 里面有一条发送记录。



此时，Account2 账号已经收到 Account1 账号发送的 2000 MIC 币。



至此，使用 OPENZEPELIN 库创建自己的代币的教程已经完成，当然，这个只是简化版的代币，在正式使用中还要涉及很多方面的内容，以后有机会再慢慢细说。

### 5.5.3 结语

智能合约一旦部署就难以修改，因此合约的安全性及其重要。虽然我们可以借助 OPENZEPPELIN 库创建相对来说比较安全的代币，并且对于开发者来说要避免重复造轮子，但是别人的轮子也不一定是完美的，所以不要过分依赖。在使用的同时要学习其中的思想，并且尝试去优化它。

#### 参考

BUILDING ROBUST SMART CONTRACTS WITH OPENZEPPELIN

<http://truffleframework.com/tutorials/robust-smart-contracts-with-openzeppelin>

基于 OpenZeppelin 建立标准代币

<http://n4sb.me/2018/02/16/%E5%9F%BA%E4%BA%8EOpenZeppelin%E5%BB%BA%E7%AB%8B%E6%A0%87%E5%87%86%E4%BB%A3%E5%B8%81/>

How to Secure Your Smart Contracts: 6 Solidity Vulnerabilities and how to avoid them (Part 1)

<https://medium.com/loom-network/how-to-secure-your-smart-contracts-6-solidity-vulnerabilities-and-how-to-avoid-them-part-1-c33048d4d17d>



