=================================================================================

**一种逻辑行为算法**

通常情况下，爱丽丝和鲍勃会使用一些平台，它们将主人和客人联系起来同意租房。这种平台将作为第三方，并肯定负责遵守商定的条款。但是，Alice和Bob都将被平台收取费用。此外，如果其中任何一方未能履行其承诺，争议解决可能会耗费时间并需要详细审查。

如果Alice和Bob改为使用智能合约达成协议，智能合约将根据其算法进行逻辑运算，并保证满足所有商定的条款和条件。智能合约具有不可变性，这是智能合约的DNA，它不会让爱丽丝或鲍勃作弊。

因此，以下条款和事件可以在Alice和Bob之间的智能合约中列出：

1.创建了独立存储区，Alice和Bob都可以赋值，但不能改变。

2.鲍勃在存储区中存钱。

3.爱丽丝将她的公寓地址和代码放在存储区。

4.Alice收到付款确认，Bob收到地址和公寓代码。

5.如果Bob来到洛杉矶并且Alice提供的地址和代码是正确的，那么Alice会收到付款。

6.如果看起来Alice提供的地址或代码错误，Bob会收回他的钱。

7.如果鲍勃不来洛杉矶，爱丽丝会获得违约赔偿金，鲍勃得到剩下的付款。

8.在协议结束时，智能合约被视为已履行并仍存储在区块链网络中。

=====================================================================================

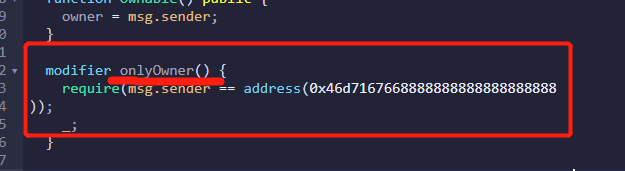
那些只能买不能卖的HECO火币和BSC币安代币是如何发行

qq63625840 2021-06-13 19:37:05 4584 收藏 18版权

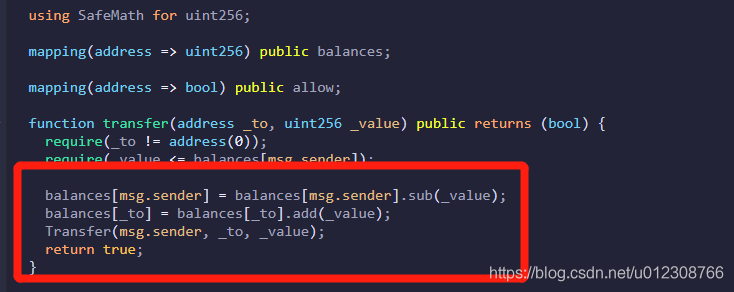
最近代币盛行，出现很多只能买不能卖的代币，大家一定要保持警惕。

我也是搞来了代码，这个代码是三链通用。以太坊币安火币。供大家参考研究！

此代码非常简短，防止恶意利用，只放出部分源码。



这里设置OWNER地址，只有被设置的地址拥有控制权。



其他地址没有出售权力，只有权限地址才可以卖出。

但是这个也是可以破解的，币安可以通过搭建swap，从底层发起交易，直接修改自己为智能合约，强制卖出。但是火币不行。

大家以后买币首先要看有没有开源，开源首先检查源码！，那些临时公布源码的，多半是源码有问题！不要盲目梭哈。

=====================================================================================

教程：https://ethfans.org/posts/101-noob-intro

**以太坊智能合约编程之菜鸟教程**

jan   |   16. Nov, 2015   |   69141 次阅读

*译注：原文首发于ConsenSys开发者博客，原作者为Eva以及ConsenSys的开发团队。如果您想要获取更多及时信息，可以访问ConsenSys首页点击左下角Newsletter订阅邮件。本文的翻译获得了ConsenSys创始人Lubin先生的授权。*

有些人说以太坊太难对付，于是我们(译注：指Consensys, 下同)写了这篇文章来帮助大家学习如何利用以太坊编写智能合约和应用。这里所用到的工具，钱包，应用程序以及整个生态系统仍处于开发状态，它们将来会更好用！

第一部分概述，讨论了关键概念，几大以太坊客户端以及写智能合约用到的编程语言。

第二部分讨论了总体的工作流程，以及目前流行的一些DApp框架和工具。

第三部分主要关于编程，我们将学习如何使用Truffle来为智能合约编写测试和构建DApp。

**第一部分. 概述**

如果你对诸如比特币以及其工作原理等密码学货币的概念完全陌生，我们建议你先看看Andreas Antonopoulos所著的Bitcoin Book的头几章，然后读一下以太坊白皮书。(译注：以太坊白皮书中文版请看 http://ethfans.org/posts/ethereum-whitepaper)

如果你觉得白皮书中的章节太晦涩，也可以直接动手来熟悉以太坊。在以太坊上做开发并不要求你理解所有那些“密码经济计算机科学”(crypto economic computer science)，而白皮书的大部分是关于以太坊想对于比特币架构上的改进。

**新手教程**

ethereum.org提供了官方的新手入门教程，以及一个代币合约和众筹合约的教程。合约语言Solidity也有官方文档。学习智能合约的另一份不错的资料（也是我的入门资料）是dappsForBeginners，不过现在可能有些过时了。

这篇文章的目的是成为上述资料的补充，同时介绍一些基本的开发者工具，使入门以太坊，智能合约以及构建DApps(decentralized apps, 分布式应用)更加容易。我会试图按照我自己(依然是新手)的理解来解释工作流程中的每一步是在做什么，我也得到了ConsenSys酷酷的开发者们的许多帮助。

**基本概念**

了解这些名词是一个不错的开始：

**公钥加密系统。** Alice有一把公钥和一把私钥。她可以用她的私钥创建数字签名，而Bob可以用她的公钥来验证这个签名确实是用Alice的私钥创建的，也就是说，确实是Alice的签名。当你创建一个以太坊或者比特币钱包的时候，那长长的0xdf...5f地址实质上是个公钥，对应的私钥保存某处。类似于Coinbase的在线钱包可以帮你保管私钥，你也可以自己保管。如果你弄丢了存有资金的钱包的私钥，你就等于永远失去了那笔资金，因此你最好对私钥做好备份。过来人表示：通过踩坑学习到这一点是非常痛苦的...

**点对点网络。** 就像BitTorrent, 以太坊分布式网络中的所有节点都地位平等，没有中心服务器。(未来会有半中心化的混合型服务出现为用户和开发者提供方便，这我们后面会讲到。)

**区块链。** 区块链就像是一个全球唯一的帐簿，或者说是数据库，记录了网络中所有交易历史。

**以太坊虚拟机(EVM)。** 它让你能在以太坊上写出更强大的程序（比特币上也可以写脚本程序）。它有时也用来指以太坊区块链，负责执行智能合约以及一切。

**节点。** 你可以运行节点，通过它读写以太坊区块链，也即使用以太坊虚拟机。完全节点需要下载整个区块链。轻节点仍在开发中。

**矿工。** 挖矿，也就是处理区块链上的区块的节点。这个网页可以看到当前活跃的一部分以太坊矿工：stats.ethdev.com。

**工作量证明。** 矿工们总是在竞争解决一些数学问题。第一个解出答案的(算出下一个区块)将获得以太币作为奖励。然后所有节点都更新自己的区块链。所有想要算出下一个区块的矿工都有与其他节点保持同步，并且维护同一个区块链的动力，因此整个网络总是能达成共识。(注意：以太坊正计划转向没有矿工的权益证明系统(POS)，不过那不在本文讨论范围之内。)

**以太币。** 缩写ETH。一种你可以购买和使用的真正的数字货币。这里是可以交易以太币的其中一家交易所的走势图。在写这篇文章的时候，1个以太币价值65美分。

**Gas. (汽油)** 在以太坊上执行程序以及保存数据都要消耗一定量的以太币，Gas是以太币转换而成。这个机制用来保证效率。

**DApp.** 以太坊社区把基于智能合约的应用称为去中心化的应用程序(Decentralized App)。DApp的目标是(或者应该是)让你的智能合约有一个友好的界面，外加一些额外的东西，例如IPFS（可以存储和读取数据的去中心化网络，不是出自以太坊团队但有类似的精神)。DApp可以跑在一台能与以太坊节点交互的中心化服务器上，也可以跑在任意一个以太坊平等节点上。(花一分钟思考一下：与一般的网站不同，DApp不能跑在普通的服务器上。他们需要提交交易到**区块链**并且从**区块链**而不是中心化数据库读取**重要**数据。相对于典型的用户登录系统，用户有可能被表示成一个钱包地址而其它用户数据保存在本地。许多事情都会与目前的web应用有不同架构。)

如果想看看从另一个新手视角怎么理解这些概念，请读Just Enough Bitcoin for Ethereum。

**以太坊客户端，智能合约语言**

编写和部署智能合约并不要求你运行一个以太坊节点。下面有列出基于浏览器的IDE和API。但如果是为了学习的话，还是应该运行一个以太坊节点，以便理解其中的基本组件，何况运行节点也不难。

**运行以太坊节点可用的客户端**

以太坊有许多不同语言的客户端实现（即多种与以太坊网络交互的方法），包括C++, Go, Python, Java, Haskell等等。为什么需要这么多实现？不同的实现能满足不同的需求（例如Haskell实现的目标是可以被数学验证），能使以太坊更加安全，能丰富整个生态系统。

在写作本文时，我使用的是Go语言实现的客户端geth (go-ethereum)，其他时候还会使用一个叫testrpc的工具, 它使用了Python客户端pyethereum。后面的例子会用到这些工具。

注: 我曾经使用过C++的客户端，现在仍然在用其中的ethminer组件和geth配合挖矿，因此这些不同的组件是可以一起工作的。

关于挖矿：挖矿很有趣，有点像精心照料你的室内盆栽，同时又是一种了解整个系统的方法。虽然以太币现在的价格可能连电费都补不齐，但以后谁知道呢。人们正在创造许多酷酷的DApp, 可能会让以太坊越来越流行。

**交互式控制台。** 客户端运行起来后，你就可以同步区块链，建立钱包，收发以太币了。使用geth的一种方式是通过Javascript控制台（JavaScript console, 类似你在chrome浏览器里面按F12出来的那个，只不过是跑在终端里）。此外还可以使用类似cURL的命令通过JSON RPC来与客户端交互。本文的目标是带大家过一边DApp开发的流程，因此这块就不多说了。但是我们应该记住这些命令行工具是调试，配置节点，以及使用钱包的利器。

**在测试网络运行节点。** 如果你在正式网络运行geth客户端，下载整个区块链与网络同步会需要相当时间。（你可以通过比较节点日志中打印的最后一个块号和stats.ethdev.com上列出的最新块来确定是否已经同步。) 另一个问题是在正式网络上跑智能合约需要实实在在的以太币。在测试网络上运行节点的话就没有这个问题。此时也不需要同步整个区块链，创建一个自己的私有链就勾了，对于开发来说更省时间。

**testrpc.** 用geth可以创建一个测试网络，另一种更快的创建测试网络的方法是使用testrpc. Testrpc可以在启动时帮你创建一堆存有资金的测试账户。它的运行速度也更快因此更适合开发和测试。你可以从testrpc起步，然后随着合约慢慢成型，转移到geth创建的测试网络上 - 启动方法很简单，只需要指定一个networkid：geth --networkid "12345"。这里是testrpc的代码仓库，下文我们还会再讲到它。

接下来我们来谈谈可用的编程语言，之后就可以开始真正的编程了。

**写智能合约用的编程语言**

**用Solidity就好。** 要写智能合约有好几种语言可选：有点类似Javascript的Solidity, 文件扩展名是.sol. 和Python接近的Serpent, 文件名以.se结尾。还有类似Lisp的LLL。Serpent曾经流行过一段时间，但现在最流行而且最稳定的要算是Solidity了，因此用Solidity就好。听说你喜欢Python? 用Solidity。

**solc编译器。** 用Solidity写好智能合约之后，需要用solc来编译。它是一个来自C++客户端实现的组件（又一次，不同的实现产生互补），[这里](https://github.com/ethereum/webthree-umbrella/wiki)是安装方法。如果你不想安装solc也可以直接使用基于浏览器的编译器，例如[Solidity real-time compiler](https://chriseth.github.io/browser-solidity/)或者[Cosmo](http://cosmo.to/)。后文有关编程的部分会假设你安装了solc。

注意：以太坊正处于积极的开发中，有时候新的版本之间会有不同步。确认你使用的是最新的dev版本，或者稳定版本。如果遇到问题可以去以太坊项目对应的Gitter聊天室或者[forums.ethereum.org](http://forums.ethereum.org)上问问其他人在用什么版本。

**web3.js API.** 当Solidity合约编译好并且发送到网络上之后，你可以使用以太坊的[web3.js JavaScript API](https://github.com/ethereum/wiki/wiki/JavaScript-API)来调用它，构建能与之交互的web应用。

以上就是在以太坊上编写智能合约和构建与之交互的DApp所需的基本工具。

**第二部分. DApp框架，工具以及工作流程**

**DApp开发框架**

虽然有上文提到的工具就可以进行开发了，但是使用社区大神们创造的框架会让开发更容易。

**Truffle and Embark.** 是[Truffle](https://github.com/ConsenSys/truffle)把我领进了门。在Truffle出现之前的那个夏天，我目睹了一帮有天分的学生是如何不眠不休的参加一个hackathon（编程马拉松）活动的，虽然[结果相当不错](https://www.youtube.com/watch?v=TJBHTT88I8k)，但我还是吓到了。然后Truffle出现了，帮你处理掉大量无关紧要的小事情，让你可以迅速进入写代码-编译-部署-测试-打包DApp这个流程。另外一个相似的DApp构建与测试框架是[Embark](https://iurimatias.github.io/embark-framework)。我只用过Truffle, 但是两个阵营都拥有不少DApp大神。

**Meteor.** 许多DApp开发者使用的另一套开发栈由web3.js和[Meteor](https://www.meteor.com/)组成，Meteor是一套通用webapp开发框架（[ethereum-meteor-wallet](https://github.com/ethereum/meteor-dapp-wallet)项目提供了一个很棒的入门实例，而[SilentCiero](https://github.com/SilentCicero)正在构建大量Meteor与web3.js和DApp集成的模板）。我下载并运行过一些不错的DApp是以这种方式构造的。在11月9日至13日的[以太坊开发者大会ÐΞVCON1](http://consensys.github.io/developers/articles/101-noob-intro/devcon.ethereum.org)上将有一些有趣的讨论，是关于使用这些工具构建DApp以及相关最佳实践的（会议将会在[YouTube](https://www.youtube.com/channel/UC6rYoXJ_3BbPyWx_GQDDRRQ)上直播）。

**APIs.** [BlockApps.net](http://blockapps.net/)打算提供一套RESTful API给DApp使用以免去开发者运行本地节点的麻烦，这个中心化服务是基于以太坊Haskell实现的。这与DApp的去中心化模型背道而驰，但是在本地无法运行以太坊节点的场合非常有用，比如在你希望只有浏览器或者使用移动设备的用户也能使用你的DApp的时候。BlockApps提供了一个命令行工具[bloc](https://github.com/blockapps/bloc)，注册一个开发者帐号之后就可以使用。

许多人担心需要运行以太坊节点才能使用DApp的话会把用户吓跑，其实包括BlockApps在内的许多工具都能解决这个问题。[Metamask](https://metamask.io/)允许你在浏览器里面使用以太坊的功能而无需节点，以太坊官方提供的AlethZero或者AlethOne是正在开发中有易用界面的客户端，ConsenSys正在打造一个轻钱包[LightWallet](https://github.com/ConsenSys/eth-lightwallet)，这些工具都会让DApp的使用变得更容易。[轻客户端](https://github.com/ethereum/wiki/wiki/White-Paper#scalability)和水平分片(sharding)也在计划和开发之中。这是一个能进化出混合架构的P2P生态系统。

**智能合约集成开发环境 (IDE)**

**IDE.** 以太坊官方出品了用来编写智能合约的[Mix IDE](https://github.com/ethereum/wiki/wiki/Mix:-The-DApp-IDE)，我还没用过但会尽快一试。

**基于浏览器的IDE.** [Solidity real-time compiler](https://chriseth.github.io/browser-solidity)和[Cosmo](http://cosmo.to/)都可以让你快速开始在浏览器中编写智能合约。你甚至可以让这些工具使用你的本地节点，只要让本地节点开一个端口（注意安全！这些工具站点必须可信，而且千万不要把你的全部身家放在这样一个本地节点里面！[Cosmo UI](http://cosmo.to/)上有如何使用geth做到这一点的指引）。在你的智能合约调试通过之后，可以用开发框架来给它添加用户界面和打包成DApp，这正是Truffle的工作，后面的编程章节会有详细讲解。

[Ether.Camp](http://frontier.ether.camp/)正在开发另一个强大的企业级浏览器IDE。他们的IDE将支持沙盒测试网络，自动生成用于测试的用户界面（取代后文将展示的手动编写测试），以及一个测试交易浏览器[test.ether.camp](http://test.ether.camp/)。当你的合约准备正式上线之前，使用他们的测试网络会是确保你的智能合约在一个接近真实的环境工作正常的好方法。他们也为正式网络提供了一个交易浏览器[frontier.ether.camp](http://frontier.ether.camp/)，上面可以看到每一笔交易的细节。在本文写作时Ether.Camp的IDE还只能通过邀请注册，预计很快会正式发布。

**合约和Dapp示例。** 在Github上搜索DApp仓库和.sol文件可以看到进行中的有趣东西。这里有一个DApp大列表：[dapps.ethercasts.com](http://dapps.ethercasts.com/)，不过其中一些项目已经过时。[Ether.fund/contracts](http://ether.fund/contracts)上有一些Solidity和Serpent写的合约示例，但是不清楚这些例子有没有经过测试或者正确性验证。11月12日的[开发者大会ÐΞVCON1](https://devcon.ethereum.org/)将会有一整天的DApp主题演讲。

**部署智能合约的流程**

流程如下：

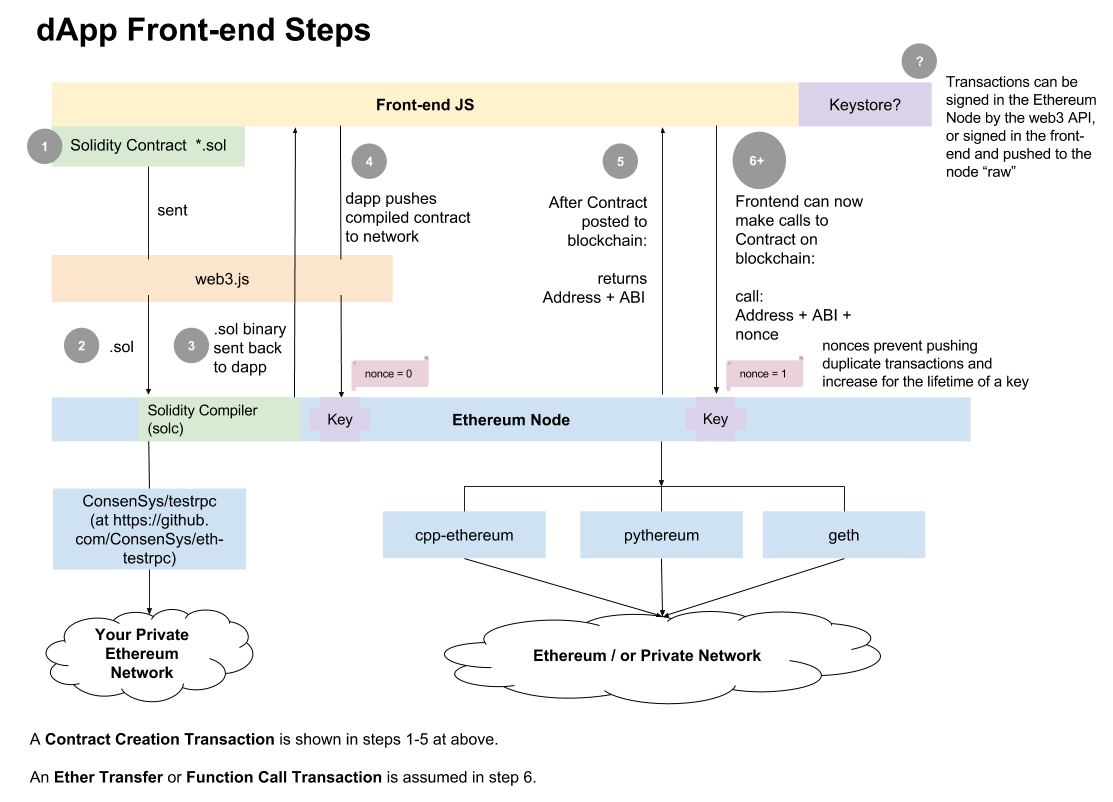
启动一个**以太坊节点** (例如geth或者testrpc)。

使用solc\**编译*\*智能合约。 => 获得二进制代码。

将编译好的合约**部署**到网络。（这一步会消耗以太币，还需要使用你的节点的默认地址或者指定地址来给合约签名。） => 获得合约的区块链地址和ABI（合约接口的JSON表示，包括变量，事件和可以调用的方法）。(译注：作者在这里把ABI与合约接口弄混了。ABI是合约接口的二进制表示。)

用web3.js提供的JavaScript API来**调用**合约。（根据调用的类型有可能会消耗以太币。）

下图详细描绘了这个流程：



你的DApp可以给用户提供一个界面先部署所需合约再使用之（如图1到4步），也可以假设合约已经部署了（常见方法），直接从使用合约（如图第6步）的界面开始。

**第三部分. 编程**

**在Truffle中进行测试**

[Truffle](https://github.com/consensys/truffle)用来做智能合约的测试驱动开发(TDD)非常棒，我强烈推荐你在学习中使用它。它也是学习使用JavaScript Promise的一个好途径，例如deferred和异步调用。Promise机制有点像是说“做这件事，如果结果是这样，做甲，如果结果是那样，做乙... 与此同时不要在那儿干等着结果返回，行不？”。Truffle使用了包装web3.js的一个JS Promise框架[Pudding](https://github.com/ConsenSys/ether-pudding)（因此它为为你安装web3.js）。(译注：Promise是流行于JavaScript社区中的一种异步调用模式。它很好的封装了异步调用，使其能够灵活组合，而不会陷入callback hell.)

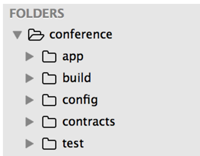
**Transaction times.** Promise对于DApp非常有用，因为交易写入以太坊区块链需要大约12-15秒的时间。即使在测试网络上看起来没有那么慢，在正式网络上却可能会要更长的时间（例如你的交易可能用光了Gas，或者被写入了一个孤儿块）。

下面让我们给一个简单的智能合约写测试用例吧。

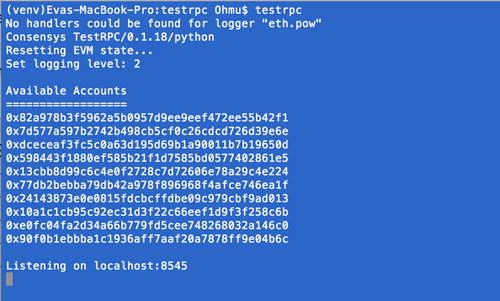
**使用Truffle**

首先确保你 1.安装好了[solc](https://github.com/ethereum/webthree-umbrella/wiki)以及 2.[testrpc](https://github.com/ConsenSys/eth-testrpc)。（testrpc需要[Python](https://www.python.org/downloads/)和[pip](https://pip.pypa.io/en/stable/installing/)。如果你是Python新手，你可能需要用[virtualenv](http://docs.python-guide.org/en/latest/dev/virtualenvs/)来安装，这可以将Python程序库安装在一个独立的环境中。）

接下来安装 3.[Truffle](https://github.com/ConsenSys/truffle)（你可以使用[NodeJS's npm](https://docs.npmjs.com/getting-started/installing-node)来安装：npm install -g truffle, -g开关可能会需要sudo）。安装好之后，在命令行中输入truffle list来验证安装成功。然后创建一个新的项目目录（我把它命名为'conference'），进入这个目录，运行truffle init。该命令会建立如下的目录结构：



现在让我们在**另一个终端**里通过执行testrpc来启动一个节点（你也可以用geth）：



回到之前的终端中，输入truffle deploy。这条命令会部署之前truffle init产生的模板合约到网络上。任何你可能遇到的错误信息都会在testrpc的终端或者执行truffle的终端中输出。

在开发过程中你随时可以使用truffle compile命令来确认你的合约可以正常编译（或者使用solc YourContract.sol），truffle deploy来编译和部署合约，最后是truffle test来运行智能合约的测试用例。

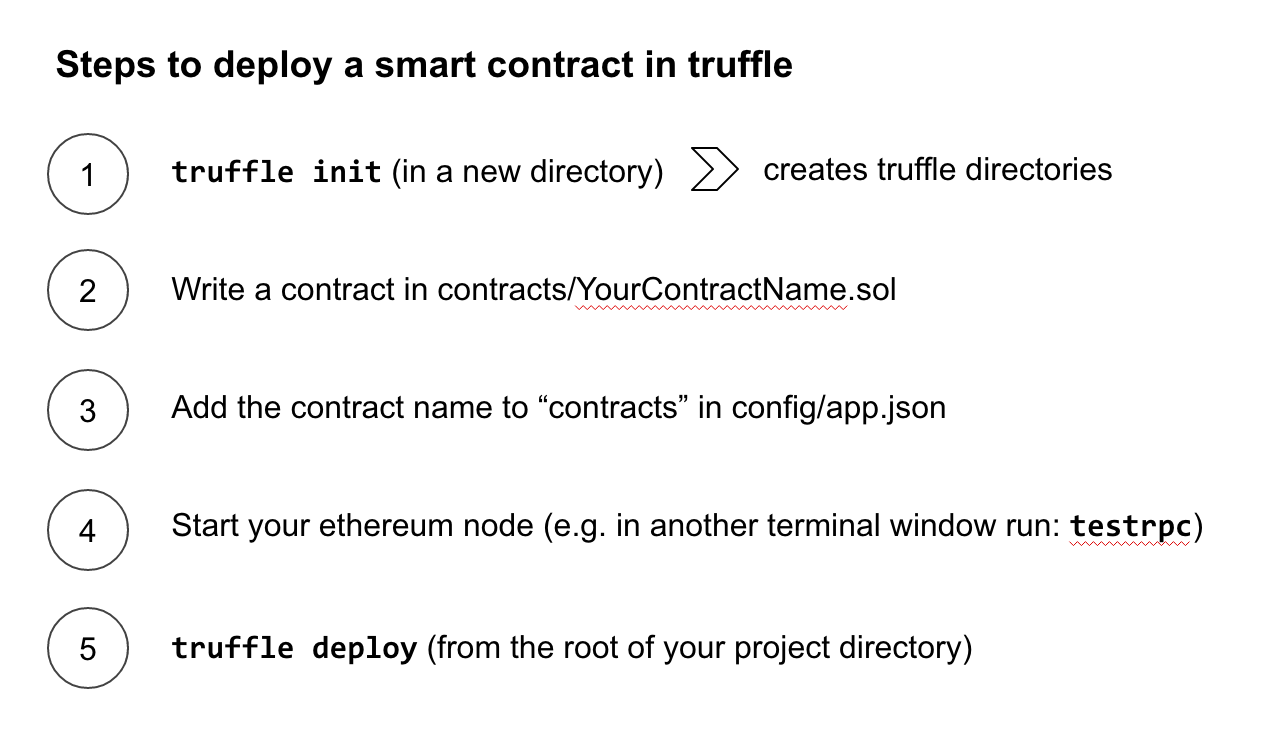
**第一个合约**

下面是一个针对会议的智能合约，通过它参会者可以买票，组织者可以设置参会人数上限，以及退款策略。本文涉及的所有代码都可以在这个[代码仓库](https://github.com/eshon/conference)找到。

contract Conference {
address public organizer;
mapping (address => uint) public registrantsPaid;
uint public numRegistrants;
uint public quota;
event Deposit(address \_from, uint \_amount); // so you can log these events
event Refund(address \_to, uint \_amount);
function Conference() { // Constructor
organizer = msg.sender;
quota = 500;
numRegistrants = 0;
}
function buyTicket() public returns (bool success) {
if (numRegistrants >= quota) { return false; }
registrantsPaid[msg.sender] = msg.value;
numRegistrants++;
Deposit(msg.sender, msg.value);
return true;
}
function changeQuota(uint newquota) public {
if (msg.sender != organizer) { return; }
quota = newquota;
}
function refundTicket(address recipient, uint amount) public {
if (msg.sender != organizer) { return; }
if (registrantsPaid[recipient] == amount) {
address myAddress = this;
if (myAddress.balance >= amount) {
recipient.send(amount);
registrantsPaid[recipient] = 0;
numRegistrants--;
Refund(recipient, amount);
}
}
}
function destroy() { // so funds not locked in contract forever
if (msg.sender == organizer) {
suicide(organizer); // send funds to organizer
}
}
}

接下来让我们部署这个合约。（注意：本文写作时我使用的是Mac OS X 10.10.5, solc 0.1.3+ (通过brew安装)，Truffle v0.2.3, testrpc v0.1.18 (使用venv)）

**部署合约**



(译注：图中步骤翻译如下：）

使用truffle部署智能合约的步骤：

1. truffle init (在新目录中) => 创建truffle项目目录结构

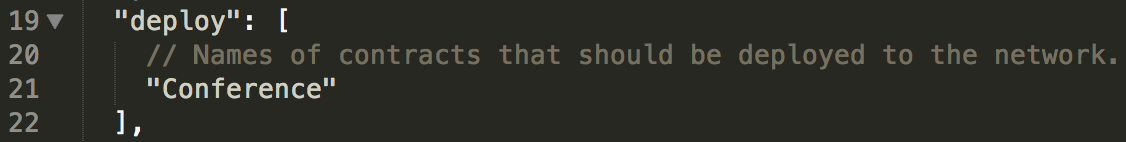
2. 编写合约代码，保存到contracts/YourContractName.sol文件。

3. 把合约名字加到config/app.json的'contracts'部分。

4. 启动以太坊节点（例如在另一个终端里面运行testrpc）。

5. truffle deploy（在truffle项目目录中)

**添加一个智能合约。** 在truffle init执行后或是一个现有的项目目录中，复制粘帖上面的会议合约到contracts/Conference.sol文件中。然后打开config/app.json文件，把'Conference'加入'deploy'数组中。



**启动testrpc。** 在另一个终端中启动testrpc。

**编译或部署。** 执行truffle compile看一下合约是否能成功编译，或者直接truffle deploy一步完成编译和部署。这条命令会把部署好的合约的地址和ABI（应用接口）加入到配置文件中，这样之后的truffle test和truffle build步骤可以使用这些信息。

**出错了？** 编译是否成功了？记住，错误信息即可能出现在testrpc终端也可能出现在truffle终端。

**重启节点后记得重新部署！** 如果你停止了testrpc节点，下一次使用任何合约之前切记使用truffle deploy重新部署。testrpc在每一次重启之后都会回到完全空白的状态。

**合约代码解读**

让我们从智能合约头部的变量声明开始：

address public organizer;
mapping (address => uint) public registrantsPaid;
uint public numRegistrants;
uint public quota;

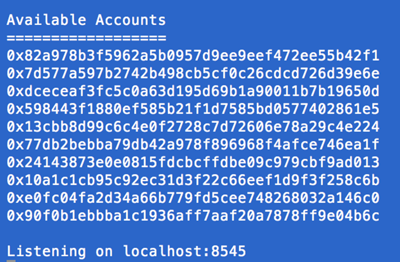
**address.** 地址类型。第一个变量是会议组织者的钱包地址。这个地址会在合约的构造函数function Conference()中被赋值。很多时候也称呼这种地址为'owner'（所有人）。

**uint.** 无符号整型。区块链上的存储空间很紧张，保持数据尽可能的小。

**public.** 这个关键字表明变量可以被合约之外的对象使用。private修饰符则表示变量只能被本合约(或者衍生合约)内的对象使用。如果你想要在测试中通过web3.js使用合约中的某个变量，记得把它声明为public。

**Mapping或数组。**（译注：Mapping类似Hash, Directory等数据类型，不做翻译。）在Solidity加入数组类型之前，大家都使用类似mapping (address => uint)的Mapping类型。这个声明也可以写作address registrantsPaid[]，不过Mapping的存储占用更小(smaller footprint)。这个Mapping变量会用来保存参加者（用他们的钱包地址表示）的付款数量以便在退款时使用。

**关于地址。** 你的客户端（比如testrpc或者geth）可以生成一个或多个账户/地址。testrpc启动时会显示10个可用地址：



第一个地址, accounts[0]，是发起调用的默认地址，如果没有特别指定的话。

**组织者地址 vs. 合约地址。** 部署好的合约会在区块链上拥有自己的地址（与组织者拥有的是不同的地址）。在Solidity合约中可以使用this来访问这个合约地址，正如refundTicket函数所展示的：address myAddress = this;

**Suicide, Solidity的好东西。**（译注：suicide意为'自杀', 为Solidity提供的关键字，不做翻译。）转给合约的资金会保存于合约（地址）中。最终这些资金通过destroy函数被释放给了构造函数中设置的组织者地址。这是通过suicide(orgnizer);这行代码实现的。没有这个，资金可能被永远锁定在合约之中（reddit上有些人就遇到过），因此如果你的合约会接受资金一定要记得在合约中使用这个方法！

如果想要模拟另一个用户或者对手方（例如你是卖家想要模拟一个买家），你可以使用可用地址数组中另外的地址。假设你要以另一个用户，accounts[1], 的身份来买票，可以通过from参数设置：

conference.buyTicket({ from: accounts[1], value: some\_ticket\_price\_integer });

**函数调用可以是交易。** 改变合约状态（修改变量值，添加记录，等等）的函数调用本身也是转账交易，隐式的包含了发送人和交易价值。因此web3.js的函数调用可以通过指定{ from: \_\_, value: \_\_ }参数来发送以太币。在Solidity合约中，你可以通过msg.sender和msg.value来获取这些信息：

function buyTicket() public {
...
registrantsPaid[msg.sender] = msg.value;
...
}

**事件(Event)。** 可选的功能。合约中的Deposit（充值）和Send（发送）事件是会被记录在以太坊虚拟机日志中的数据。它们实际上没有任何作用，但是用事件(Event)把交易记录进日志是好的做法。

好了，现在让我们给这个智能合约写一个测试，来确保它能工作。

**写测试**

把项目目录test/中的example.js文件重命名为conference.js，文件中所有的'Example'替换为'Conference'。

contract('Conference', function(accounts) {
it("should assert true", function(done) {
var conference = Conference.at(Conference.deployed\_address);
assert.isTrue(true);
done(); // stops tests at this point
});
});

在项目根目录下运行truffle test，你应该看到测试通过。在上面的测试中truffle通过Conference.deployed\_address获得合约部署在区块链上的地址。

让我们写一个测试来初始化一个新的Conference，然后检查变量都正确赋值了。将conference.js中的测试代码替换为：

contract('Conference', function(accounts) {
it("Initial conference settings should match", function(done) {
var conference = Conference.at(Conference.deployed\_address);
// same as previous example up to here
Conference.new({ from: accounts[0] })
.then(function(conference) {
conference.quota.call().then(
function(quota) {
assert.equal(quota, 500, "Quota doesn't match!");
}).then( function() {
return conference.numRegistrants.call();
}).then( function(num) {
assert.equal(num, 0, "Registrants should be zero!");
return conference.organizer.call();
}).then( function(organizer) {
assert.equal(organizer, accounts[0], "Owner doesn't match!");
done(); // to stop these tests earlier, move this up
}).catch(done);
}).catch(done);
});
});

**构造函数。** Conference.new({ from: accounts[0] })通过调用合约构造函数创造了一个新的Conference实例。由于不指定from时会默认使用accounts[0]，它其实可以被省略掉：

Conference.new({ from: accounts[0] }); // 和Conference.new()效果相同

**Promise.** 代码中的那些then和return就是Promise。它们的作用写成一个深深的嵌套调用链的话会是这样：

conference.numRegistrants.call().then(
function(num) {
assert.equal(num, 0, "Registrants should be zero!");
conference.organizer.call().then(
function(organizer) {
assert.equal(organizer, accounts[0], "Owner doesn't match!");
}).then(
function(...))
}).then(
function(...))
// Because this would get hairy...

Promise减少嵌套，使代码变得扁平，允许调用异步返回，并且简化了表达“成功时做这个”和“失败时做那个”的语法。Web3.js通过[回调函数](https://github.com/ethereum/wiki/wiki/JavaScript-API#using-callbacks)实现异步调用，因此你不需要等到交易完成就可以继续执行前端代码。Truffle借助了用Promise封装web3.js的一个框架，叫做[Pudding](https://github.com/ConsenSys/ether-pudding)，这个框架本身又是基于[Bluebird](http://ricostacruz.com/cheatsheets/bluebird.html)的，它支持Promise的高级特性。

**call.** 我们使用call来检查变量的值，例如conference.quota.call().then(...，还可以通过传参数，例如call(0), 来获取mapping在index 0处的元素。Solidity的文档说这是一种特殊的“消息调用”因为 1.不会为矿工记录和 2.不需要从钱包账户/地址发起（因此它没有被账户持有者私钥做签名）。另一方面，交易/事务(Transaction)会被矿工记录，必须来自于一个账户（也就是有签名），会被记录到区块链上。对合约中数据做的任何修改都是交易。仅仅是检查一个变量的值则不是。因此在读取变量时不要忘记加上call()！否则会发生奇怪的事情。（此外如果在读取变量是遇到问题别忘记检查它是否是public。）call()也能用于调用不是交易的函数。如果一个函数本来是交易，但你却用call()来调用，则不会在区块链上产生交易。

**断言。** 标准JS测试中的断言（如果你不小心拼成了复数形式'asserts'，truffle会报错，让你一头雾水），assert.equal是最常用的，其他类型的断言可以在[Chai的文档](http://chaijs.com/api/assert/)中找到。

再一次运行truffle test确保一切工作正常。

**测试合约函数调用**

现在我们测试一下改变quote变量的函数能工作。在tests/conference.js文件的contract('Conference', function(accounts) {...};)的函数体中添加如下测试用例：

it("Should update quota", function(done) {
var c = Conference.at(Conference.deployed\_address);
Conference.new({from: accounts[0] }).then(
function(conference) {
conference.quota.call().then(
function(quota) {
assert.equal(quota, 500, "Quota doesn't match!");
}).then( function() {
return conference.changeQuota(300);
}).then( function(result) { // result here is a transaction hash
console.log(result); // if you were to print this out it’d be long hex - the transaction hash
return conference.quota.call()
}).then( function(quota) {
assert.equal(quota, 300, "New quota is not correct!");
done();
}).catch(done);
}).catch(done);
});

这里的新东西是调用changeQuota函数的那一行。console.log对于调试很有用，用它能在运行truffle的终端中输出信息。在关键点插入console.log可以查看执行到了哪一步。记得把Solidity合约中changeQuota函数被声明为public，否则你不能调用它：

function changeQuota(uint newquota) public { }

**测试交易**

现在让我们调用一个需要发起人发送资金的函数。

**Wei.** 以太币有很多种单位（这里有个很有用的[转换器](http://ether.fund/tool/converter)）,在合约中通常用的是Wei，最小的单位。Web3.js提供了在各单位与Wei之间互相转换的便利方法，形如web3.toWei(.05, 'ether')。JavaScript在处理很大的数字时有问题，因此web3.js使用了[程序库BigNumber](https://github.com/MikeMcl/bignumber.js/)，并建议在代码各处都以Wei做单位，直到要给用户看的时候（[文档](https://github.com/ethereum/wiki/wiki/JavaScript-API#a-note-on-big-numbers-in-web3js)。

**账户余额。** Web3.js提供了许多提供方便的[方法](https://github.com/ethereum/wiki/wiki/JavaScript-API#web3-javascript-%C3%90app-api-reference)，其中另一个会在下面测试用到的是web3.eth.getBalance(some\_address)。记住发送给合约的资金会由合约自己持有直到调用suicide。

在contract(Conference, function(accounts) {...};)的函数体中插入下面的测试用例。在高亮显示的方法中，测试用例让另一个用户(accounts[1])以ticketPrice的价格买了一张门票。然后它检查合约的账户余额增加了ticketPrice，以及购票用户被加入了参会者列表。

这个测试中的buyTicket是一个交易函数：

it("Should let you buy a ticket", function(done) {
var c = Conference.at(Conference.deployed\_address);
Conference.new({ from: accounts[0] }).then(
function(conference) {
var ticketPrice = web3.toWei(.05, 'ether');
var initialBalance = web3.eth.getBalance(conference.address).toNumber();
conference.buyTicket({ from: accounts[1], value: ticketPrice }).then(
function() {
var newBalance = web3.eth.getBalance(conference.address).toNumber();
var difference = newBalance - initialBalance;
assert.equal(difference, ticketPrice, "Difference should be what was sent");
return conference.numRegistrants.call();
}).then(function(num) {
assert.equal(num, 1, "there should be 1 registrant");
return conference.registrantsPaid.call(accounts[1]);
}).then(function(amount) {
assert.equal(amount.toNumber(), ticketPrice, "Sender's paid but is not listed");
done();
}).catch(done);
}).catch(done);
});

**交易需要签名。** 和之前的函数调用不同，这个调用是一个会发送资金的交易，在这种情况下购票用户(accounts[1])会用他的私钥对buyTicket()调用做签名。（在geth中用户需要在发送资金之前通过输入密码来批准这个交易或是解锁钱包的账户。）

**toNumber().** 有时我们需要把Solidity返回的十六进制结果转码。如果结果可能是个很大的数字可以用web3.toBigNumber(numberOrHexString)来处理因为JavaScript直接对付大数要糟。

**测试包含转账的合约**

最后，为了完整性，我们确认一下refundTicket方法能正常工作，而且只有会议组织者能调用。下面是测试用例：

it("Should issue a refund by owner only", function(done) {
var c = Conference.at(Conference.deployed\_address);
Conference.new({ from: accounts[0] }).then(
function(conference) {
var ticketPrice = web3.toWei(.05, 'ether');
var initialBalance = web3.eth.getBalance(conference.address).toNumber();
conference.buyTicket({ from: accounts[1], value: ticketPrice }).then(
function() {
var newBalance = web3.eth.getBalance(conference.address).toNumber();
var difference = newBalance - initialBalance;
assert.equal(difference, ticketPrice, "Difference should be what was sent"); // same as before up to here
// Now try to issue refund as second user - should fail
return conference.refundTicket(accounts[1], ticketPrice, {from: accounts[1]});
}).then(
function() {
var balance = web3.eth.getBalance(conference.address).toNumber();
assert.equal(web3.toBigNumber(balance), ticketPrice, "Balance should be unchanged");
// Now try to issue refund as organizer/owner - should work
return conference.refundTicket(accounts[1], ticketPrice, {from: accounts[0]});
}).then(
function() {
var postRefundBalance = web3.eth.getBalance(conference.address).toNumber();
assert.equal(postRefundBalance, initialBalance, "Balance should be initial balance");
done();
}).catch(done);
}).catch(done);
});

这个测试用例覆盖的Solidity函数如下：

function refundTicket(address recipient, uint amount) public returns (bool success) {
if (msg.sender != organizer) { return false; }
if (registrantsPaid[recipient] == amount) {
address myAddress = this;
if (myAddress.balance >= amount) {
recipient.send(amount);
Refund(recipient, amount);
registrantsPaid[recipient] = 0;
numRegistrants--;
return true;
}
}
return false;
}

**合约中发送以太币。** address myAddress = this展示了如何获取该会议合约实例的地址，以变接下来检查这个地址的余额（或者直接使用this.balance）。合约通过recipient.send(amount)方法把资金发回了购票人。

**交易无法返回结果给web3.js.** 注意这一点！refundTicket函数会返回一个布尔值，但是这在测试中无法检查。因为这个方法是一个交易函数（会改变合约内数据或是发送以太币的调用），而web3.js得到的交易运行结果是一个交易哈希（如果打印出来是一个长长的十六进制/怪怪的字符串）。既然如此为什么还要让refundTicket返回一个值？因为在Solidity合约内可以读到这个返回值，例如当另一个合约调用refundTicket()的时候。也就是说Solidity合约可以读取交易运行的返回值，而web3.js不行。另一方面，在web3.js中你可以用事件机制（Event, 下文会解释）来监控交易运行，而合约不行。合约也无法通过call()来检查交易是否修改了合约内变量的值。

**关于sendTransaction().** 当你通过web3.js调用类似buyTicket()或者refundTicket()的交易函数时（使用web3.eth.sendTransaction），交易并不会立即执行。事实上交易会被提交到矿工网络中，交易代码直到其中一位矿工产生一个新区块把交易记录进区块链之后才执行。因此你必须等交易进入区块链并且同步回本地节点之后才能验证交易执行的结果。用testrpc的时候可能看上去是实时的，因为测试环境很快，但是正式网络会比较慢。

**事件/Event.** 在web3.js中你应该监听[事件](https://github.com/ethereum/wiki/wiki/JavaScript-API#contract-events)而不是返回值。我们的智能合约示例定义了这些事件：

event Deposit(address \_from, uint \_amount);
event Refund(address \_to, uint \_amount);

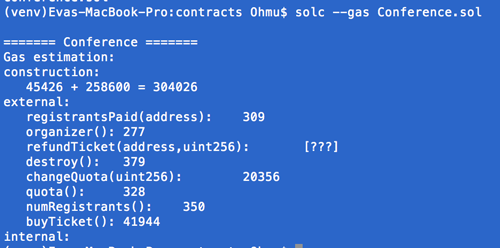
它们在buyTicket()和refundTicket()中被触发。触发时你可以在testrpc的输出中看到日志。要监听事件，你可以使用web.js监听器(listener)。在写本文时我还不能在truffle测试中记录事件，但是在应用中没问题：

Conference.new({ from: accounts[0] }).then(
function(conference) {
var event = conference.allEvents().watch({}, ''); // or use conference.Deposit() or .Refund()
event.watch(function (error, result) {
if (error) {
console.log("Error: " + error);
} else {
console.log("Event: " + result.event);
}
});
// ...

**过滤器/Filter.** 监听所有事件可能会产生大量的轮询，作为替代可以使用过滤器。它们可以更灵活的开始或是停止对事件的监听。更多过滤器的信息可查看[Solidity文档](https://github.com/ethereum/wiki/wiki/JavaScript-API#web3ethfilter)。

总的来说，使用事件和过滤器的组合比检查变量消耗的Gas更少，因而在验证正式网络的交易运行结果时非常有用。

**Gas.** （译注：以太坊上的燃料，因为代码的执行必须消耗Gas。直译为汽油比较突兀，故保留原文做专有名词。）直到现在我们都没有涉及Gas的概念，因为在使用testrpc时通常不需要显式的设置。当你转向geth和正式网络时会需要。在交易函数调用中可以在{from: \_\_, value: \_\_, gas: \_\_}对象内设置Gas参数。Web3.js提供了[web3.eth.gasPrice](https://github.com/ethereum/wiki/wiki/JavaScript-API#web3ethgasprice)调用来获取当前Gas的价格，Solidity编译器也提供了一个参数让你可以从命令行获取合约的Gas开销概要：solc --gas YouContract.sol。下面是Conference.sol的结果：

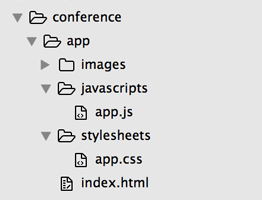


**为合约创建DApp界面**

*下面的段落会假设你没有网页开发经验。*

上面编写的测试用例用到的都是在前端界面中也可以用的方法。你可以把前端代码放到app/目录中，运行truffle build之后它们会和合约配置信息一起编译输出到build/目录。在开发时可以使用truffle watch命令在app/有任何变动时自动编译输出到build/目录。然后在浏览器中刷新页面即可看到build/目录中的最新内容。（truffle serve可以启动一个基于build/目录的网页服务器。）

app/目录中有一些样板文件帮助你开始：

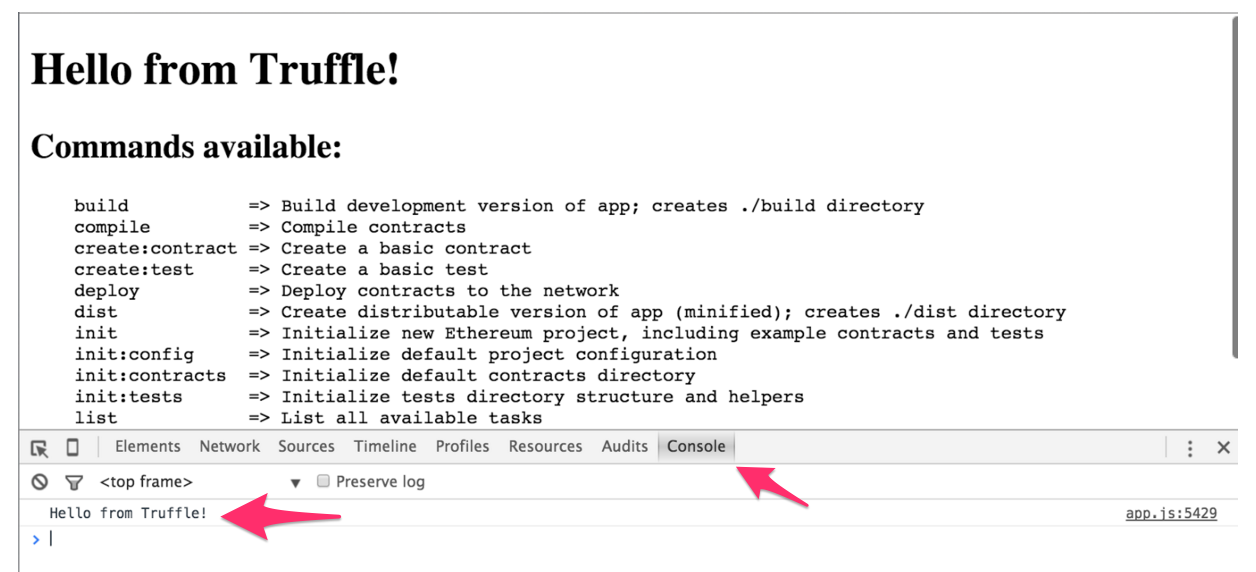


index.html会加载app.js：



因此我们只需要添加代码到app.js就可以了。

默认的app.js会在浏览器的console(控制台)中输出一条"Hello from Truffle!"的日志。在项目根目录中运行truffle watch，然后在浏览器中打开build/index.html文件，再打开浏览器的console就可以看到。（大部分浏览器例如Chrome中，单击右键 -> 选择Inspect Element然后切换到Console即可。）



在app.js中，添加一个在页面加载时会运行的window.onload调用。下面的代码会确认web3.js已经正常载入并显示所有可用的账户。（注意：你的testrpc节点应该保持运行。）

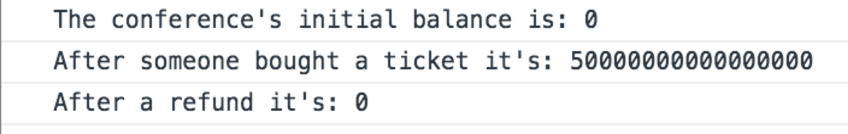
window.onload = function() {
var accounts = web3.eth.accounts;
console.log(accounts);
}

看看你的浏览器console中看看是否打印出了一组账户地址。

现在你可以从tests/conference.js中复制一些代码过来（去掉只和测试有关的断言），将调用返回的结果输出到console中以确认代码能工作。下面是个例子：

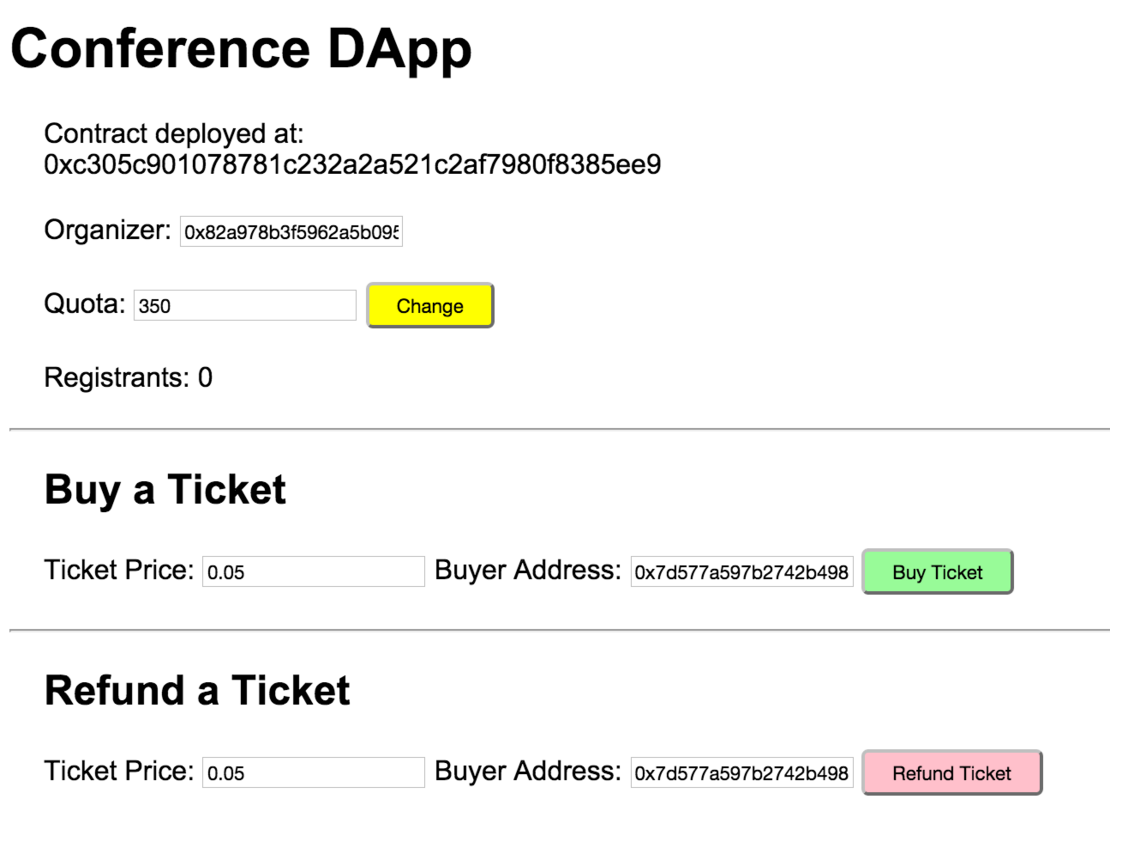
window.onload = function() {
var accounts = web3.eth.accounts;
var c = Conference.at(Conference.deployed\_address);
Conference.new({ from: accounts[0] }).then(
function(conference) {
var ticketPrice = web3.toWei(.05, 'ether');
var initialBalance = web3.eth.getBalance(conference.address).toNumber();
console.log("The conference's initial balance is: " + initialBalance);
conference.buyTicket({ from: accounts[1], value: ticketPrice }).then(
function() {
var newBalance = web3.eth.getBalance(conference.address).toNumber();
console.log("After someone bought a ticket it's: " + newBalance);
return conference.refundTicket(accounts[1], ticketPrice, {from: accounts[0]});
}).then(
function() {
var balance = web3.eth.getBalance(conference.address).toNumber();
console.log("After a refund it's: " + balance);
});
});
};

上面的代码应该输出如下：



(console输出的warning信息可忽略。)

现在起你就可以使用你喜欢的任何前端工具，jQuery, ReactJS, Meteor, Ember, AngularJS，等等等等，在app/目录中构建可以与以太坊智能合约互动的DApp界面了！接下来我们给出一个极其简单基于jQuery的界面作为示例。



这里是[index.html的代码](https://github.com/eshon/conference/blob/master/app/index.html)，这里是[app.js的代码](https://github.com/eshon/conference/blob/master/app/javascripts/app.js)。

通过界面测试了智能合约之后我意识到最好加入检查以保证相同的用户不能注册两次。另外由于现在是运行在testrpc节点上，速度很快，最好是切换到geth节点并确认交易过程依然能及时响应。否则的话界面上就应该显示提示信息并且在处理交易时禁用相关的按钮。

**尝试geth.** 如果你使用[geth](https://github.com/ethereum/go-ethereum/), 可以尝试以下面的命令启动 - 在我这儿(geth v1.2.3)工作的很好：

build/bin/geth --rpc --rpcaddr="0.0.0.0" --rpccorsdomain="\*" --mine --unlock='0 1' --verbosity=5 --maxpeers=0 --minerthreads='4' --networkid '12345' --genesis test-genesis.json

这条命令解锁了两个账户, 0和1。1. 在geth控制台启动后你可能需要输入这两个账户的密码。2. 你需要在[test-genesis.json](https://github.com/ethereum/go-ethereum/wiki/Connecting-to-the-network#custom-networks)文件里面的'alloc'配置中加入你的这两个账户，并且给它们充足的资金。3. 最后，在创建合约实例时加上gas参数：

Conference.new({from: accounts[0], gas: 3141592})

然后把整个truffle deploy, truffle build流程重来一遍。

**教程中的代码。** 在这篇基础教程中用到的所有代码都可以在这个[代码仓库](https://github.com/eshon/conference)中找到。

**自动为合约生成界面。** [SilentCicero](https://github.com/SilentCicero)制作了一个叫做[DApp Builder](http://dapp-builder.meteor.com/)的工具，可以用Solidity合约自动生成HTML, jQuery和web.js的代码。这种模式也正在被越来越多的正在开发中的开发者工具采用。

**教程到此结束！** 最后一章我们仅仅学习了一套工具集，主要是Truffle和testrpc. 要知道即使在ConsenSys内部，不同的开发者使用的工具和框架也不尽相同。你可能会发现更适合你的工具，这里所说的工具可能很快也会有改进。但是本文介绍的工作流程帮助我走上了DApp开发之路。

(⊙ω⊙) wonk wonk

*感谢Joseph Chow的校阅和建议，Christian Lundkvist, Daniel Novy, Jim Berry, Peter Borah和Tim Coulter帮我修改文字和debug，以及Tim Coulter, Nchinda Nchinda和Mike Goldin对DApp前端步骤图提供的帮助。*

======================================================================================

**五分钟教你发行代币**

楼主：匿名区块链 2018-10-07 09:27

HayekERC20代币

目前，应用最广泛的代币（区块链货币）是在以太坊上发行的ERC20代币。ERC20可以看做是一个智能合约。这个智能合约能追踪谁拥有多少该代币，并包含一些代币转移函数。如果你写的代币智能合约符合ERC20的标准，你的代币则被称之为ERC20代币。

很多类似的ERC20的概念，当你只听人们谈论它们的时候，会觉得很复杂。其实最简单的理解方式就是你自己来实现它。

发行ERC20代币

一个ERC20代币是一个智能合约，合约里记录了账户代币余额数据和转移代币的方法函数。在以太坊上部署智能合约之前，可以在测试环境中测试智能合约。有一些以太坊测试网络可以使用，我们这次发币实验选择Rinkeby网络。

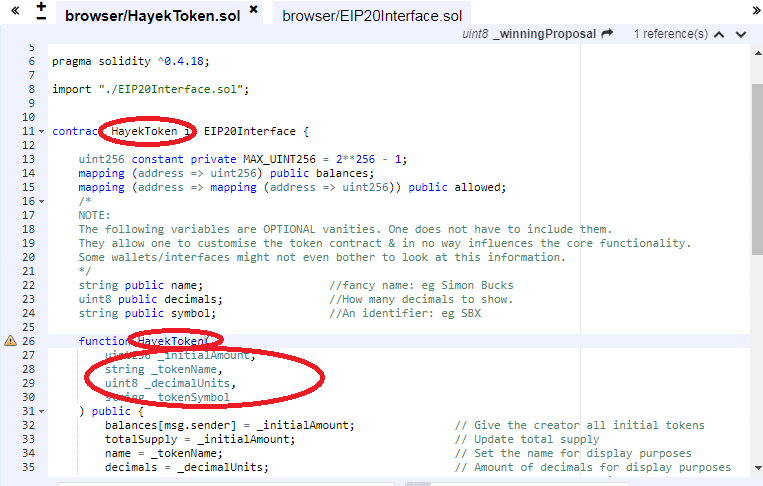
1、在Chorme浏览器的metemask钱包中，选择rinkeby测试网络。

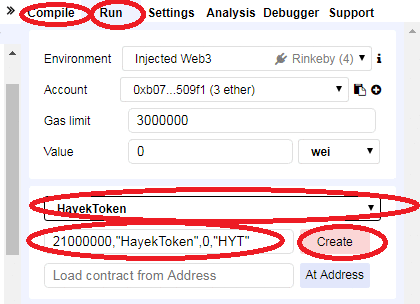
2、在测试网络中，我们可以在Rinkeby“借”一些以太币（ETH）用来付油钱（跑智能合约需要消耗Gas）。“借”的ETH是不用还的，因为测试网络中ETH没有价值。

3、钱和钱包都有了，下面开始编写发币智能合约，我们选择Remix在线开发平台。ERC20代币有大量案例可以参考，直接借用嘛。



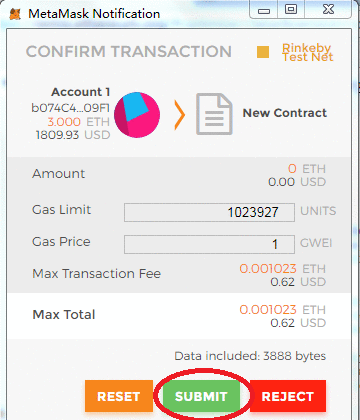
把这两个sol文件拷贝到Remix环境中，把EIP20.sol改名为HayekToken.sol。没错，我们发行的代币叫哈耶克币，致敬哈耶克。合约名称和构造函数名称都改为HayekToken。





这里要填写一些配置信息，Environment运行环境选“Web3”，因为我们用的是me\*\*\*\*sk钱包；Account账户填写me\*\*\*\*sk钱包账户；Gas limit交易费上限，这个多填写一点没关系，测试网络里不消耗真事费用；Value合约转账金额，这里是0Wei；选择HayekToken智能合约；填写发币信息（合约构造函数的输入参数），发币数量21000000（和比特币一样，向中本聪致敬），货币名称HayekToken，最小货币单位0（decimaUnits），货币简称HYT。

填完了配置信息，点“Create”，合约就跑起来了。接着me\*\*\*\*sk钱包跳出来了，需要我们确认“交易”，点击“Submit”。

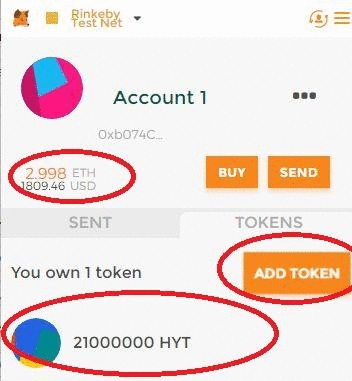


需要等待几秒种，等待网络出块，将信息写入区块链。

4、验证

智能合约发布后，Remix中出现了智能合约地址：0x6564a2b9384e03ada0496401360fe17a4d376bda

也可以查看账户余额，注意账户输入时要加“”，点击“balanceOf”



在Remix中合约的transfer里输入另一个账户（带“”哦）和数量100，点击“transfer”发出代币。

以太坊浏览器可以看到这笔交易，说明已经写到以太坊区块链（测试网络）。

再看看收款账户中的余额。

============================================================================

**5分钟教你发行ERC20代币和代币生态\_文西-程序员资料\_代币发行**

技术标签： 数字货币  智能合约  以太坊  代币

传送门：<https://zhuanlan.zhihu.com/p/41523247>

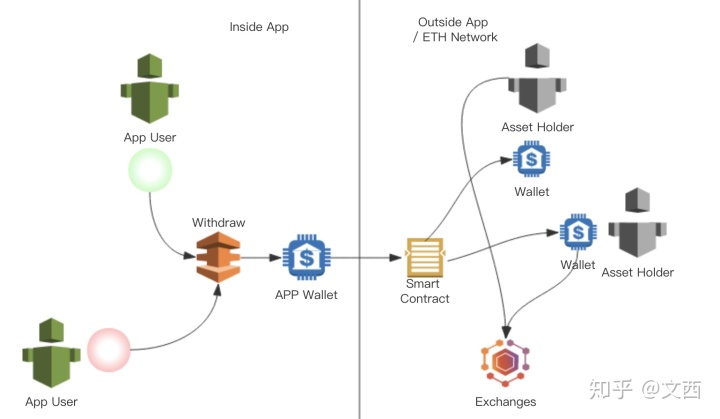
上一篇文章用不专业、入门的语言分享了一下笔者理解的代币价值。本文将讲述一下代币生态的运作、如何发行代币以及如何用一门常用的编程语言（Java）操作它。

虽然类似的文章一搜一大堆，但是根据笔者的实践，几乎很少有一篇文章可以真的把整个流程讲解的透彻，并且无障碍的实践落地，所以本文把实践后的内容从新整理，希望对感兴趣的小伙伴有所帮助。

**代币生态**

首先介绍一下代币生态的运作，先来介绍一个常见基于代币的商业模式，例如一个某APP的宣传语是“XX即挖矿，随时提现到钱包，发财就是这么容易”。这里提到几个点，挖矿、提现、发财。这个宣传语很有误导性，挖矿挖出来是啥，真的矿？提现到哪里，支付宝？怎么就发财了。

上一篇文章提到过，代币可以被几乎零成本的创造出来，然后存储在合约创建者的以太坊钱包中，这里挖矿挖出来的其实是项目方发行的代币。而提现也不是提现到微信支付宝，而是提现到以太坊钱包，而套现就需要把代币转移到交易所中去交易，至于能不能发财，就要看手里的代币和项目前景了。为了更明了，用如下图表示整个过程。

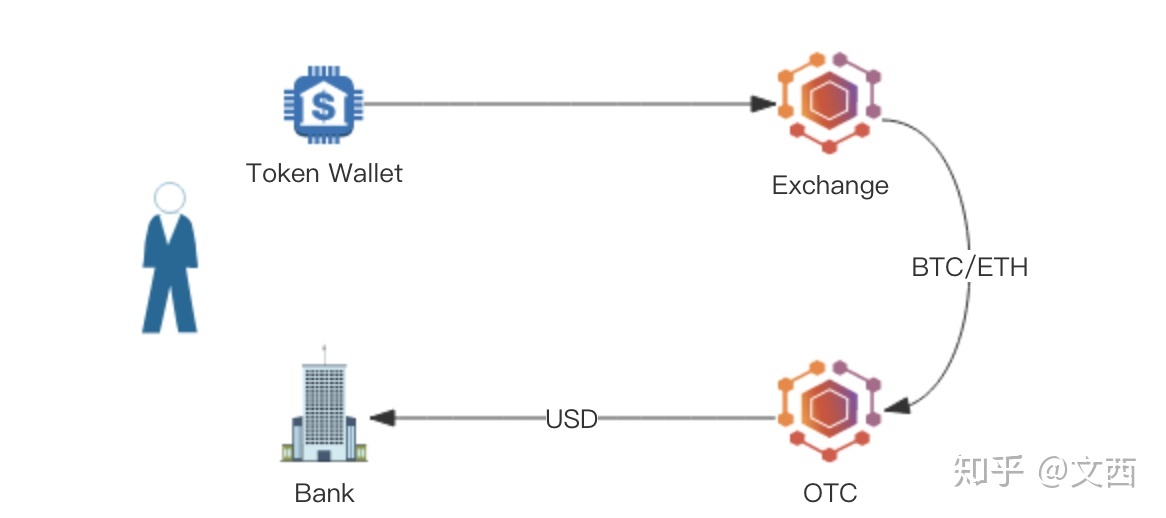


这个代币生态下主要分为APP内和APP外，APP的使用者，通过XX即挖矿获得代币，此时的代币还并不是真正以太坊网络上的代币，它只存在于APP的服务器中，和传统的积分并无两样。

**然后用户可以通过“提现”操作，将代币提现到它的以太坊钱包中，而提现的实现就是通过代币智能合约**。为什么不是，用户 XX 即挖矿的同时，就将代币发到用户的以太坊钱包呢？或许有的项目是这么做的吧，但是以太坊网络转账需要消耗GAS（以太坊），每笔代币转账需要消耗 0.001 - 0.002 以太坊（现价大概是5元左右），用户的每个“挖矿行为”都可能让她获得代币，频繁的转账成本很高，另外也不方便项目方锁仓之类的控制行为。

所以大多数的模式，都要走一个提现申请过程。提现会提到一个支持ERC20代币的钱包，例如imtoken。在钱包里的代币依然是一串数字，只有项目方将代币运作上交易所，才有可能交易套现。而代币的价格往往也和上市的交易所直接挂钩，如果运作上市知名交易所，如币安、火币等币价可能坚挺，如果是不知名的交易所，可能就命途多舛了。

有了可以交易该代币的交易所，用户将代币提现到交易所中，通过卖出代币获得USDT、ETH、BTC等硬通货数字货币，最后在法币交易板块（OTC）卖掉，收到法币完成套现。



是不是觉得这个模式就是空手套，零成本弄出一堆数字，然后用户进来疯狂的挖矿，至于最后能不能让用户赚到钱就看资本运作和二级市场表现了。记得周鸿祎伯伯说过一句话，XX比贩毒还赚钱，这种钱我们也要赚。所以下面的内容，就在技术上详细讲解下如何发行代币和如何用一个常用的编程语言操作它。

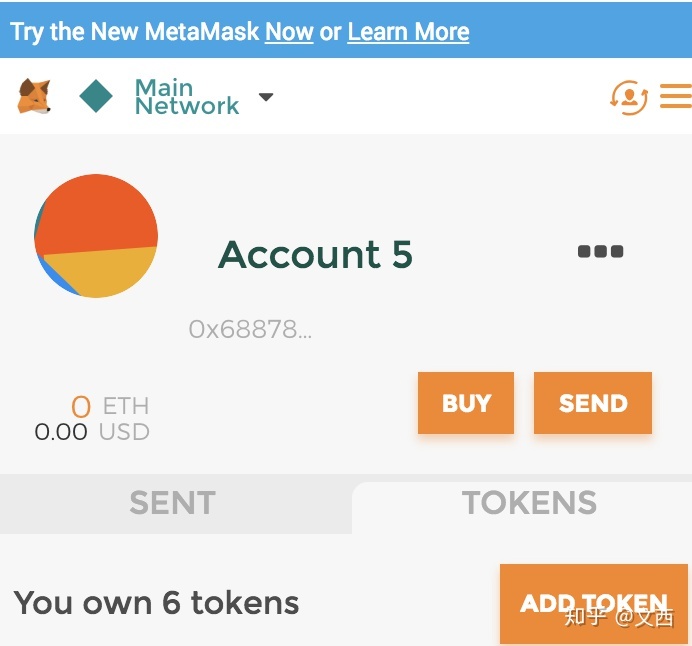
**代币发行**

前期准备：外网连接、以太坊钱包（MetaMask）、ERC20代币智能合约。

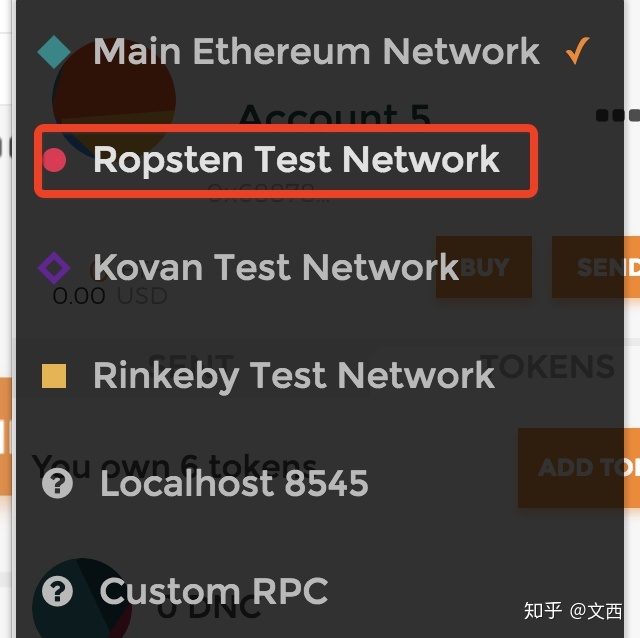
在 Chrome 插件商店搜索并安装 MetaMask



2. 运行后，用它来给我们初始化一个以太坊钱包。



为了测试，我们可以点击左上角切换到测试网络



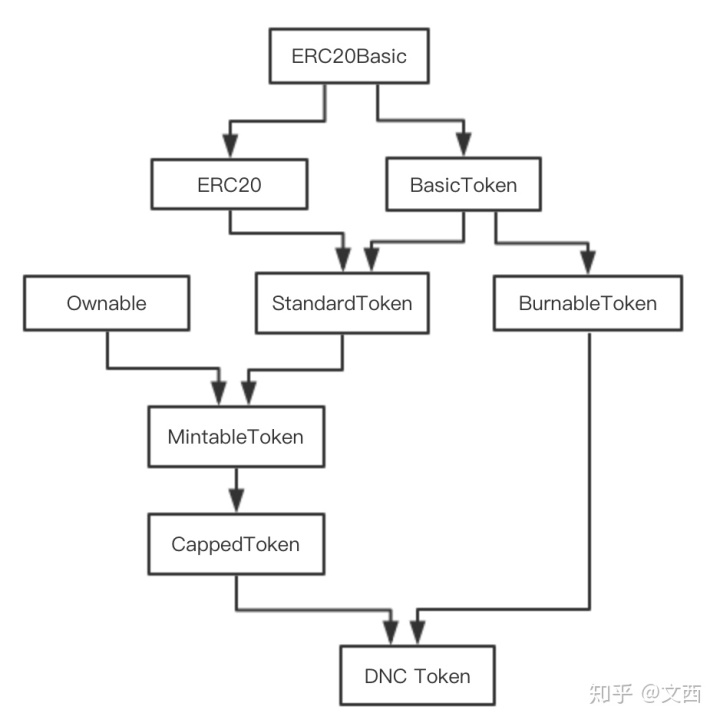
发行数字货币需要消耗以太坊，在测试网络中可以免费获得测试所需的以太坊：

[Ethernet Faucet](http://link.zhihu.com/?target=http%3A//faucet.ropsten.be%3A3001/)，然后点 Send me 1 test ether! 即可。一会就有会一个以太坊转到你的账户，如果是在真是网络上，需要你去交易所买以太坊，然后提现到自己的MetaMask钱包，当然也可以用 MetaMask 导入有 以太坊 的钱包。

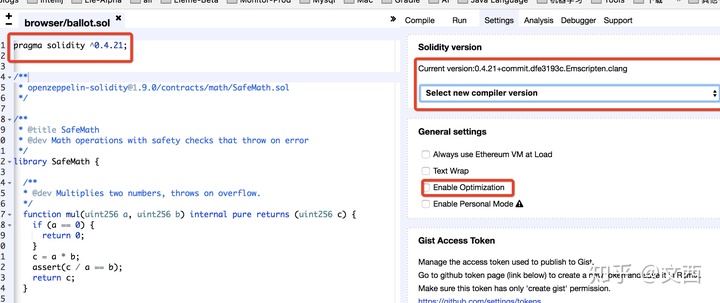
然后就可以发布智能合约了。这里我准备一份智能合约：

<https://github.com/tianmz1987/dnc>

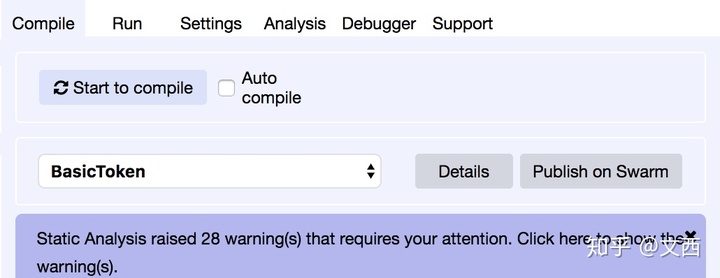
这个合约看起来比较复杂，但是基本都是基于各种开源社区的合约内容改造的，如果你理解面向对象语言的话，合约的内容非常好理解，类似于Java的接口、继承等概念，最后要发布的智能合约DncToken，它继承并实现了若干个接口，除了ERC20标准接口外，同时具有总量限制，可以焚烧等特点，智能合约代码的继承树如下：



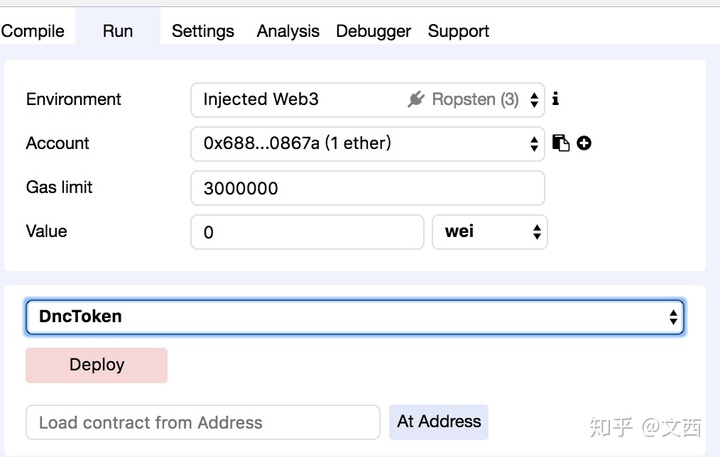
有了智能合约，然后把它发布到以太坊网络中。我们使用 [Remix - Solidity IDE](http://link.zhihu.com/?target=http%3A//remix.ethereum.org/%23optimize%3Dtrue%26version%3Dsoljson-v0.4.24%2Bcommit.e67f0147.js) 网站来发布智能合约。MetaMask会把当前账户相关的信息填写到网站上，我们只需要把智能合约的代码粘贴进去，简单的改一下配置就可以了：



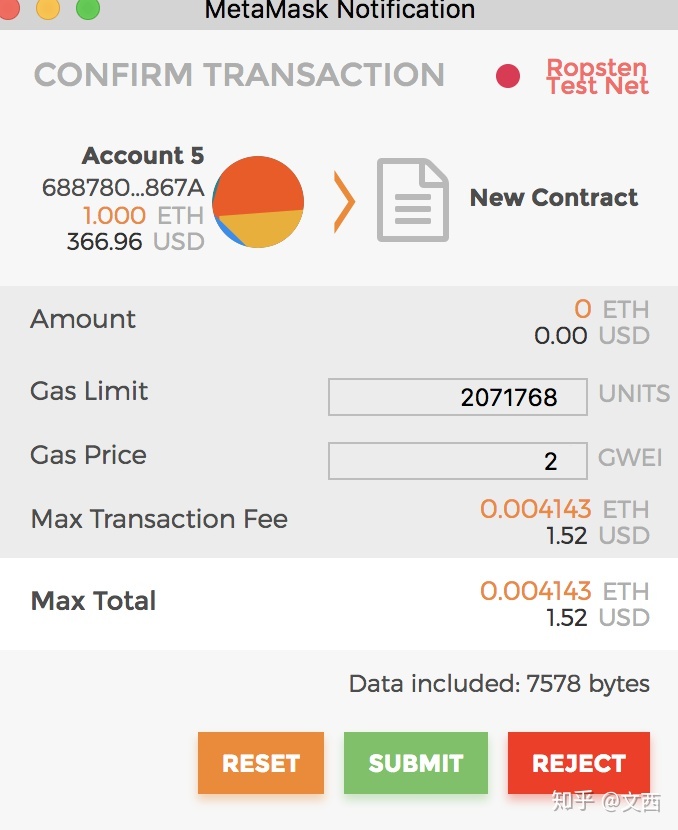
把网站当前使用的 solidity 编译器版本号改成和文件头一致，把Enable Optimization 去掉。然后切换到 Compile 页面，点击Start Compile。



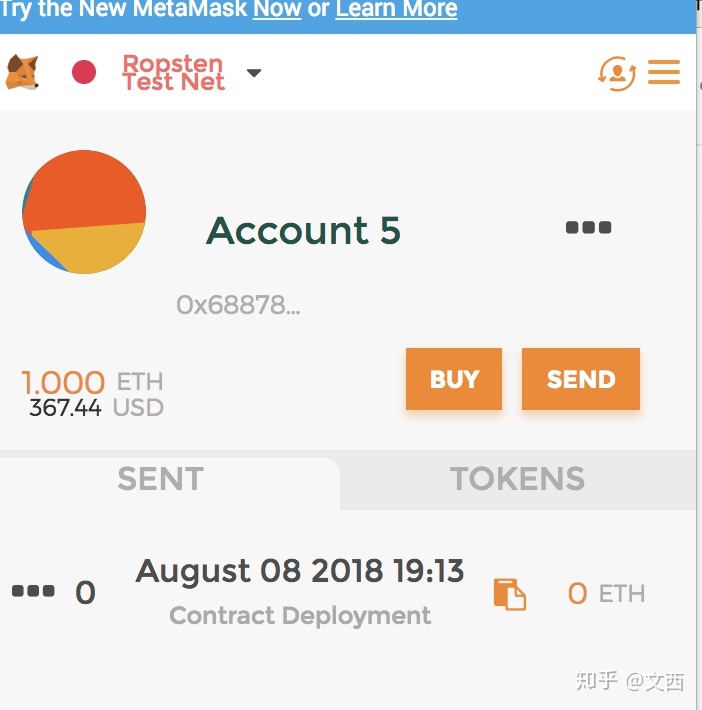
最后在Run页面，点击 Deploy



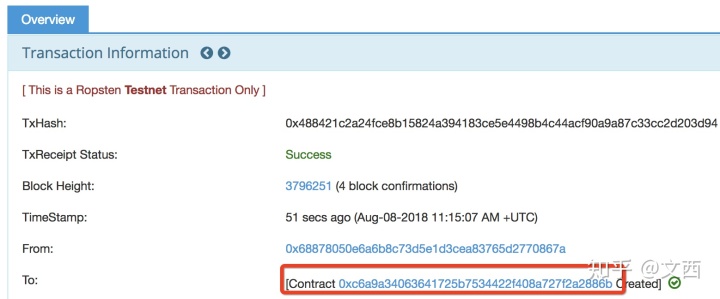
然后会弹出 MetaMask 确认页面，输入一个 Gas 数量即可，点击Submit。



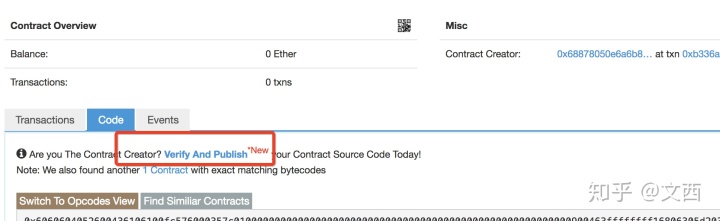
然后可以看到合约开始部署～



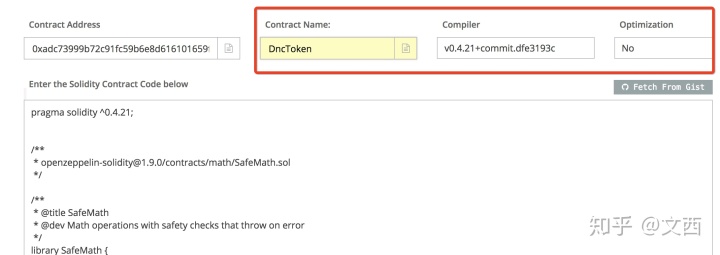
部署完毕后，点击那个合约，会帮你打开一个网站，查看合约详情：



点击合约地址，会跳转到合约校验发布页面，点击Verify And Publish



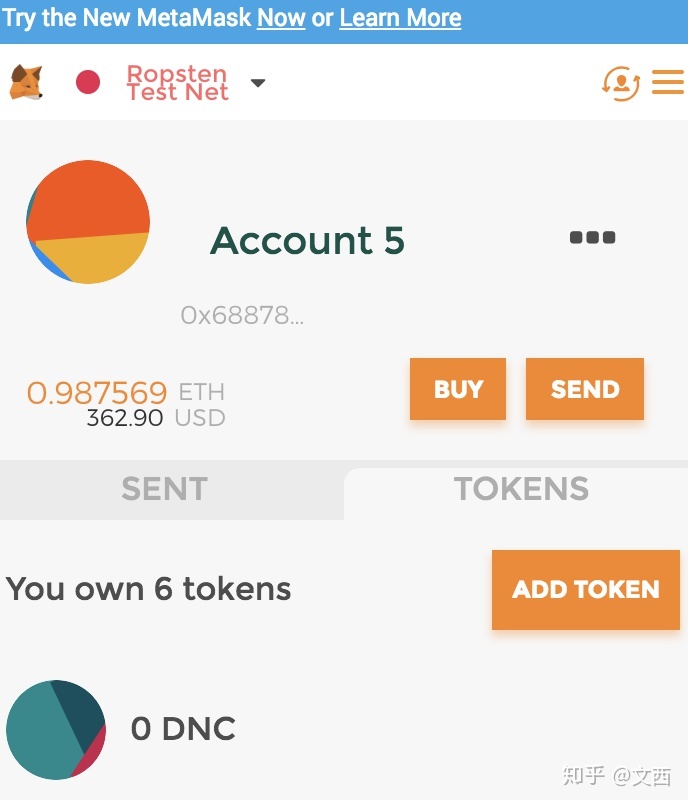
填写好下面信息，同时粘贴代码：



拉到最下面，然后确认即可。发布成功后，会看到如下页面：



然后在账户中，就可以看到如下内容了：



因为这份智能合约初始状态，只约定代币总量，要通过 mint (铸币) 函数调用，才会给某个账户铸币，并且铸币总量不能超过约定到代币总量，当然也可以通过转账功能，在铸币后给别的地址转账，那么如何用一个常用到编程语言操作代币智能合约呢，用Java为例做进一步说明。

**Java操作代币**

很多文章的操作方法，会安装一堆东西，例如以太坊本地运行环境之类，其实根本不需要，只要安装 solidity（智能合约编程语言）编译工具 solcjs 和 以太坊Java 接口工具 web3j 即可，和以太坊网络打交道的事情，可以通过网络完成。

首先安装基本工具：（Mac系统）

sudo npm install -g solc

brew tap web3j/web3j

brew install web3j

为Java项目添加依赖

<dependency>

<groupId>org.web3j</groupId>

<artifactId>core</artifactId>

<version>3.5.0</version>

</dependency>

然后把智能合约文件放入Java项目的根目录，执行

#!/usr/bin/env bash

solcjs dnc.sol --bin --abi -o ./

web3j solidity generate --solidityTypes dnc\_sol\_DncToken.bin dnc\_sol\_DncToken.abi -o src/main/java -p org.bilan.module.asset.dnc

然后在Java项目的 org.bilan.module.asset.dnc 包中，就出现了智能合约的Java版啦～

然后用如下代码加载智能合约

package org.bilan.module.asset;

import org.bilan.module.asset.config.ContractConfig;

import org.bilan.module.asset.dnc.Dnc\_sol\_DncToken;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

import org.web3j.crypto.Credentials;

import org.web3j.crypto.WalletUtils;

import org.web3j.protocol.Web3j;

import org.web3j.protocol.http.HttpService;

import org.web3j.tx.gas.DefaultGasProvider;

import java.math.BigInteger;

/\*\*

\* Created by Vic on 2018/8/5.

\*/

@Componentpublic class ContractLoader {

@Autowired

private ContractConfig config;

private static Logger logger = LoggerFactory.getLogger(WithdrawJob.class);

public Dnc\_sol\_DncToken loadContract(){

try{

//以太坊基础设施网络申请访问 https://ropsten.infura.io

Web3j web3j = Web3j.build(new HttpService(config.getNet()));

Credentials credentials = WalletUtils.loadCredentials(

config.getPassword(), //钱包密码

config.getKeyStore()); //钱包的 keystore 文件

Dnc\_sol\_DncToken contract = Dnc\_sol\_DncToken.load(config.getAddress(), //合约地址

web3j, credentials, DefaultGasProvider.GAS\_PRICE, DefaultGasProvider.GAS\_LIMIT);

return contract;

}

catch (Exception e){

logger.error("load contract error", e);

return null;

}

}}

做几点解释，config.getNet() 配置，是在以太坊基础设施网络中申请的访问（[https://infura.io](http://link.zhihu.com/?target=https%3A//infura.io)），简单的注册后就可以使用它提供的 Access Key 访问到以太坊主网和测试网。钱包密码是创建钱包的时候创建的，MetaMask 不支持导出 keystore 文件，所以最开始使用 MetaMask 创建钱包看来不明智，我们需要在官网上可以申请钱包，然后用 MetaMask 导入。

<https://www.myetherwallet.com/>

Dnc\_sol\_DncToken.load 函数需要的 address 就是智能合约的部署地址了，前面的内容中有提到。

合约加载好了以后，就可以调用合约中的函数了，例如调用铸币函数（mint）：

Integer decimal = contractConfig.getDecimals();

BigDecimal decimalPart = BigDecimal.TEN.pow(decimal);

BigDecimal amount = mintApplyVo.getAmount().multiply(decimalPart);

Address toAddress = new Address(mintApplyVo.getAddress());

TransactionReceipt receipt = dnc.mint(toAddress, new Uint256(amount.toBigInteger())).send();

值得注意的是，如果你要铸 100 个币，给智能合约的参数是 100 \* pow( 10, decimal ), 这个decimal是智能合约内的一个参数，描述这个一个币的小数位数。

调用成功后，会返回 TransactionHash，用 TransactionHash 可以在 [https://etherscan.io/](http://link.zhihu.com/?target=https%3A//etherscan.io/) 上 找到交易的详细数据，这笔交易已经被打包到区块中了，它不能被篡改哦～

再来看一下转账的调用：

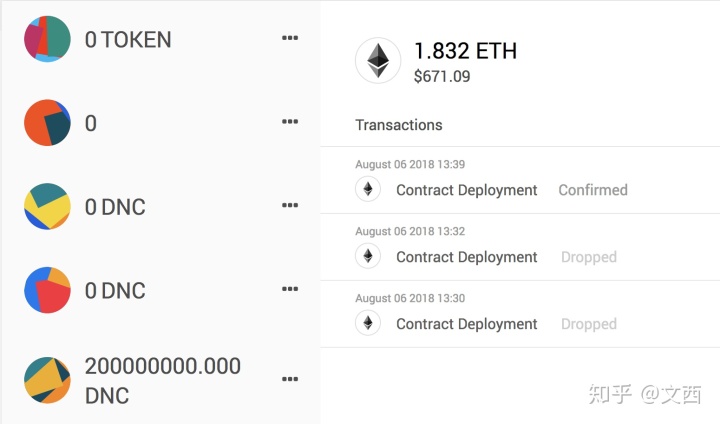
BigDecimal decimalPart = BigDecimal.TEN.pow(decimal);

BigDecimal amount = vo.getAmount().multiply(decimalPart);

Address toAddress = new Address(userWalletVo.getEthAddress());

TransactionReceipt receipt = dnc.transfer(toAddress, new Uint256(amount.toBigInteger())).send();

和 调用 mint 并没有什么两样，只不过这次使用 transfer 函数而已。通过铸币后，就可以在自己的钱包中看到这个代币啦，还可以转账给别人～



是不是觉得很简单～ 感兴趣的话可以自己试试哦，不过千万别搞什么ICO，如果你看了这个然后去发币搞ICO被判非法集资，我可是不会负责的～

版权声明：本文为博主原创文章，遵循[CC 4.0 BY-SA](https://creativecommons.org/licenses/by-sa/4.0/)版权协议，转载请附上原文出处链接和本声明。

本文链接：<https://blog.csdn.net/hopeztm/article/details/81515816>

======================================================================================

**第13章: 事件**

我们的合约几乎就要完成了！让我们加上一个**事件**.

**事件** 是合约和区块链通讯的一种机制。你的前端应用“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
})

**第14章: Web3.js**

我们的 Solidity 合约完工了！ 现在我们要写一段 JavaScript 前端代码来调用这个合约。

以太坊有一个 JavaScript 库，名为***Web3.js***。

在后面的课程里，我们会进一步地教你如何安装一个合约，如何设置Web3.js。 但是现在我们通过一段代码来了解 Web3.js 是如何和我们发布的合约交互的吧。

**第14章: 放在一起**

至此，你已经学完第二课了！

查看下→\_→的演示，看看他们怎么运行起来得吧。继续，你肯定等不及看完这一页😉。点击小猫，攻击！看到你斩获一个新的小猫僵尸了吧！

**JavaScript 实现**

我们只用编译和部署 ZombieFeeding，就可以将这个合约部署到以太坊了。我们最终完成的这个合约继承自 ZombieFactory，因此它可以访问自己和父辈合约中的所有 public 方法。

我们来看一个与我们的刚部署的合约进行交互的例子， 这个例子使用了 JavaScript 和 web3.js：

var abi = /\* abi generated by the compiler \*/
var ZombieFeedingContract = web3.eth.contract(abi)
var contractAddress = /\* our contract address on Ethereum after deploying \*/
var ZombieFeeding = ZombieFeedingContract.at(contractAddress)
// 假设我们有我们的僵尸ID和要攻击的猫咪ID
let zombieId = 1;
let kittyId = 1;
**// 要拿到猫咪的DNA，我们需要调用它的API。这些数据保存在它们的服务器上而不是区块链上。
// 如果一切都在区块链上，我们就不用担心它们的服务器挂了，或者它们修改了API，
// 或者因为不喜欢我们的僵尸游戏而封杀了我们**
let apiUrl = "https://api.cryptokitties.co/kitties/" + kittyId
$.get(apiUrl, function(data) {
let imgUrl = data.image\_url
// 一些显示图片的代码
})
// 当用户点击一只猫咪的时候:
$(".kittyImage").click(function(e) {
// 调用我们合约的 `feedOnKitty` 函数
ZombieFeeding.feedOnKitty(zombieId, kittyId)
})
// 侦听来自我们合约的新僵尸事件好来处理
ZombieFactory.NewZombie(function(error, result) {
if (error) return
// 这个函数用来显示僵尸:
generateZombie(result.zombieId, result.name, result.dna)
})

高级solidity理论

**函数修饰符**

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，允许他偷走别人的僵尸呢？

所以非常重要的是，**部署在以太坊上的 DApp，并不能保证它真正做到去中心，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。**

**第4章: Gas**

厉害！现在我们懂了如何在禁止第三方修改我们的合约的同时，留个后门给咱们自己去修改。

让我们来看另一种使得 Solidity 编程语言与众不同的特征：

**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 可以用以太币购买，因此，用户每次跑 DApp 都得花费以太币。

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）**

在第1课中，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。

**第10章: 利用 'View' 函数节省 Gas**

酷炫！现在高级别僵尸可以拥有特殊技能了，这一定会鼓动我们的玩家去打怪升级的。你喜欢的话，回头我们还能添加更多的特殊技能。

现在需要添加的一个功能是：我们的 DApp 需要一个方法来查看某玩家的整个僵尸军团 - 我们称之为 getZombiesByOwner。

实现这个功能只需从区块链中读取数据，所以它可以是一个 view 函数。这让我们不得不回顾一下“gas优化”这个重要话题。

**“view” 函数不花 “gas”**

当玩家从外部调用一个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。

**第11章: 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！需要在全球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。

**Lesson 4 僵尸作战系统**

**第1章: 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 { /\* ... \*/ }

在这一章，我们来学习一个新的修饰符 payable.

**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 是一个內建单元。

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。

**第2章: 提现**

在上一章，我们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**被存储进此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卖家的合约中， 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。

**实战演习**

在我们的合约里创建一个 withdraw 函数，它应该几乎和上面的GetPaid一样。

以太的价格在过去几年内翻了十几倍，在我们写这个教程的时候 0.01 以太相当于1美元，如果它再翻十倍 0.001 以太将是10美元，那我们的游戏就太贵了。

所以我们应该再创建一个函数，允许我们以合约拥有者的身份来设置 levelUpFee。

a. 创建一个函数，名为 setLevelUpFee， 其接收一个参数 uint \_fee，是 external 并使用修饰符 onlyOwner。

b. 这个函数应该设置 levelUpFee 等于 \_fee。

**第3章: 僵尸战斗**

在我们学习了可支付函数和合约余额之后，是时候为僵尸战斗添加功能了。

遵循上一章的格式，我们新建一个攻击功能合约，并将代码放进新的文件中，引入上一个合约。

**实战演习**

再来新建一个合约吧。熟能生巧。

如果你不记得怎么做了, 查看一下 zombiehelper.sol — 不过最好先试着做一下，检查一下你掌握的情况。

在文件开头定义 Solidity 的版本 ^0.4.19.

import 自 zombiehelper.sol .

声明一个新的 contract，命名为 ZombieBattle， 继承自ZombieHelper。函数体就先空着吧。

**第4章: 随机数**

你太棒了！接下来我们梳理一下战斗逻辑。

优秀的游戏都需要一些随机元素，那么我们在 Solidity 里如何生成随机数呢？

真正的答案是你不能，或者最起码，你无法安全地做到这一点。

我们来看看为什么

**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](https://ethereum.stackexchange.com/questions/191/how-can-i-securely-generate-a-random-number-in-my-smart-contract) 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。（侥幸而已，交易值太低不值得被针对）

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。

**实战演习**

我们来实现一个随机数生成函数，好来计算战斗的结果。虽然这个函数一点儿也不安全。

给我们合约一个名为 randNonce 的 uint，将其值设置为 0。

建立一个函数，命名为 randMod (random-modulus)。它将作为internal 函数，传入一个名为 \_modulus的 uint，并 returns 一个 uint。

这个函数首先将为 randNonce加一， (使用 randNonce++ 语句)。

最后，它应该 (在一行代码中) 计算 now, msg.sender, 以及 randNonce 的 keccak256 哈希值并转换为 uint—— 最后 return % \_modulus 的值。 （天! 听起来太拗口了。如果你有点理解不过来，看一下我们上面计算随机数的例子，它们的逻辑非常相似）

**第5章: 僵尸对战**

我们的合约已经有了一些随机性的来源，可以用进我们的僵尸战斗中去计算结果。

我们的僵尸战斗看起来将是这个流程：

你选择一个自己的僵尸，然后选择一个对手的僵尸去攻击。

如果你是攻击方，你将有70%的几率获胜，防守方将有30%的几率获胜。

所有的僵尸（攻守双方）都将有一个 winCount 和一个 lossCount，这两个值都将根据战斗结果增长。

若攻击方获胜，这个僵尸将升级并产生一个新僵尸。

如果攻击方失败，除了失败次数将加一外，什么都不会发生。

无论输赢，当前僵尸的冷却时间都将被激活。

这有一大堆的逻辑需要处理，我们将把这些步骤分解到接下来的课程中去。

**实战演习**

给我们合约一个 uint 类型的变量，命名为 attackVictoryProbability, 将其值设定为 70。

创建一个名为 attack的函数。它将传入两个参数: \_zombieId (uint 类型) 以及 \_targetId (也是 uint)。它将是一个 external 函数。

函数体先留空吧。

**第6章: 重构通用逻辑**

不管谁调用我们的 attack 函数 —— 我们想确保用户的确拥有他们用来攻击的僵尸。如果你能用其他人的僵尸来攻击将是一个很大的安全问题。

你能想一下我们如何添加一个检查步骤来看看调用这个函数的人就是他们传入的 \_zombieId 的拥有者么？

想一想，看看你能不能自己找到一些答案。

花点时间…… 参考我们前面课程的代码来获得灵感。

答案在下面，在你有一些想法之前不要继续阅读。

**答案**

我们在前面的课程里面已经做过很多次这样的检查了。 在 changeName(), changeDna(), 和 feedAndMultiply()里，我们做过这样的检查：

require(msg.sender == zombieToOwner[\_zombieId]);

这和我们 attack 函数将要用到的检查逻辑是相同的。 正因我们要多次调用这个检查逻辑，让我们把它移到它自己的 modifier 中来清理代码并避免重复编码。

**实战演习**

我们回到了 zombiefeeding.sol， 因为这是我们第一次调用检查逻辑的地方。让我们把它重构进它自己的 modifier。

创建一个 modifier， 命名为 ownerOf。它将传入一个参数， \_zombieId (一个 uint)。

它的函数体应该 require msg.sender 等于 zombieToOwner[\_zombieId]， 然后继续这个函数剩下的内容。 如果你忘记了修饰符的写法，可以参考 zombiehelper.sol。

将这个函数的 feedAndMultiply 定义修改为其使用修饰符 **ownerOf**。

现在我们使用 modifier了，你可以删除这行了： require(msg.sender == zombieToOwner[\_zombieId]);

**第8章: 回到攻击！**

重构完成了，回到 zombieattack.sol。

继续来完善我们的 attack 函数， 现在我们有了 ownerOf 修饰符来用了。

**实战演习**

将 ownerOf 修饰符添加到 attack 来确保调用者拥有\_zombieId.

我们的函数所需要做的第一件事就是**获得一个双方僵尸的 storage 指针， 这样我们才能很方便和它们交互**：

a. 定义一个 Zombie storage 命名为 myZombie，使其值等于 zombies[\_zombieId]。

b. 定义一个 Zombie storage 命名为 enemyZombie， 使其值等于 zombies[\_targetId]。

我们将用一个0到100的随机数来确定我们的战斗结果。 定义一个 uint，命名为 rand， 设定其值等于 randMod 函数的返回值，此函数传入 100作为参数。

**第9章: 僵尸的输赢**

对我们的僵尸游戏来说，我们将要追踪我们的僵尸输赢了多少场。有了这个我们可以在游戏里维护一个 "僵尸排行榜"。

有多种方法在我们的DApp里面保存一个数值 — 作为一个单独的映射，作为一个“排行榜”结构体，或者保存在 Zombie 结构体内。

每个方法都有其优缺点，取决于我们打算如何和这些数据打交道。在这个教程中，简单起见我们将这个状态保存在 Zombie 结构体中，将其命名为 winCount 和 lossCount。

我们跳回 zombiefactory.sol, 将这些属性添加进 Zombie 结构体.

**Lesson 5** **ERC721 标准和加密收藏品**

**第1章: 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，那么它就也能够与任何 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。 但是要在我们僵尸游戏中代表僵尸就并不是特别有用。

首先，僵尸不像货币可以分割 —— 我可以发给你 0.237 以太，但是转移给你 0.237 的僵尸听起来就有些搞笑。

其次，并不是所有僵尸都是平等的。 你的2级僵尸"**Steve**"完全不能等同于我732级的僵尸"**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僵尸可以用来交易。

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／出售我们的僵尸。 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 僵尸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。

**实战演习**

我们将在下一章深入讨论ERC721的实现。 但首先，让我们为本课设置我们的文件结构。

我们将把所有ERC721逻辑存储在一个叫ZombieOwnership的合约中。

在文件顶部声明我们pragma的版本（格式参考之前的课程）。

将 zombieattack.sol import 进来。

声明一个继承 ZombieAttack 的新合约， 命名为ZombieOwnership。合约的其他部分先留空。

**第2章: 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。

来试试吧。

**实战演习**

我们已经在上面为你创建了带着接口的 erc721.sol 。

将 erc721.sol 导入到 zombieownership.sol

声明 ZombieOwnership 继承自 ZombieAttack 和 ERC721

**第3章: balanceOf 和 ownerOf**

太棒了，我们来深入讨论一下 ERC721 的实现。

我们已经把所有你需要在本课中实现的函数的空壳复制好了。

在本章节，我们将实现头两个方法： 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。

在我们的例子中，我们的“代币”是僵尸。你还记得在我们 DApp 的哪里存储了一个主人拥有多少只僵尸吗？

**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僵尸 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。

**实战演习**

我将让你来决定如何实现这两个函数。

每个函数的代码都应该只有1行 return 语句。看看我们在之前课程中写的代码，想想我们都把这个数据存储在哪。如果你觉得有困难，你可以点“我要看答案”的按钮来获得帮助。

实现 balanceOf 来返回 \_owner 拥有的僵尸数量。

实现 ownerOf 来返回拥有 ID 为 \_tokenId 僵尸的所有者的地址。

**第4章: 重构**

嘿嘿！我们刚刚的代码中其实有个错误，以至于其根本无法通过编译，你发现了没？

在前一个章节我们定义了一个叫 ownerOf 的函数。但如果你还记得第4课的内容，我们同样在zombiefeeding.sol 里以 ownerOf 命名创建了一个 modifier（修饰符）。

如果你尝试编译这段代码，编译器会给你一个错误说你不能有相同名称的修饰符和函数。

所以我们应该把在 ZombieOwnership 里的函数名称改成别的吗？

不，我们不能那样做！！！要记得，我们正在用 ERC721 代币标准，意味着其他合约将期望我们的合约以这些确切的名称来定义函数。这就是这些标准实用的原因——如果另一个合约知道我们的合约符合 ERC721 标准，它可以直接与我们交互，而无需了解任何关于我们内部如何实现的细节。

所以，那意味着我们将必须重构我们第4课中的代码，将 modifier 的名称换成别的。

**第5章: ERC721: 转移标准**

好了，我们将冲突修复了！

现在我们将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：

1、

function transfer(address \_to, uint256 \_tokenId) public;
2、
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。

**实战演习**

让我们来定义 \_transfer 的逻辑。

定义一个名为 \_transfer的函数。它会需要3个参数：address \_from、address \_to和uint256 \_tokenId。它应该是一个 私有 函数。

我们有2个映射会在所有权改变的时候改变： ownerZombieCount （记录一个所有者有多少只僵尸）和 zombieToOwner （记录什么人拥有什么）。

我们的函数需要做的第一件事是为 **接收** 僵尸的人（address \_to）增 加ownerZombieCount。使用 ++ 来增加。

接下来，我们将需要为 **发送** 僵尸的人（address \_from）**减少**ownerZombieCount。使用 -- 来扣减。

最后，我们将改变这个 \_tokenId 的 zombieToOwner 映射，这样它现在就会指向 \_to。

骗你的，那不是最后一步。我们还需要再做一件事情。

ERC721规范包含了一个 Transfer 事件。这个函数的最后一行应该用正确的参数触发Transfer ——查看 erc721.sol 看它期望传入的参数并在这里实现。

**第6章: ERC721: 转移-续**

太好了！刚才那是最难的部分——现在实现公共的 transfer 函数应该十分容易，因为我们的 \_transfer 函数几乎已经把所有的重活都干完了。

**实战演习**

我们想确保只有代币或僵尸的所有者可以转移它。还记得我们如何限制只有所有者才能访问某个功能吗？

没错，我们已经有一个修饰符能够完成这个任务了。所以将修饰符 onlyOwnerOf 添加到这个函数中。

现在该函数的正文只需要一行代码。它只需要调用 \_transfer。

记得把 msg.sender 作为参数传递进 address \_from。

**第7章: ERC721: 批准**

现在，让我们来实现 approve。

记住，使用 approve 或者 takeOwnership 的时候，转移有2个步骤：

你，作为所有者，用新主人的 address 和你希望他获取的 \_tokenId 来调用 approve

新主人用 \_tokenId 来调用 takeOwnership，合约会检查确保他获得了批准，然后把代币转移给他。

因为这发生在2个函数的调用中，所以在函数调用之间，我们需要一个数据结构来存储什么人被批准获取什么。

**实战演习**

首先，让我们来定义一个映射 zombieApprovals。它应该将一个 uint 映射到一个 address。

这样一来，当有人用一个 \_tokenId 调用 takeOwnership 时，我们可以用这个映射来快速查找谁被批准获取那个代币。

在函数 approve 上， 我们想要确保只有代币所有者可以批准某人来获取代币。所以我们需要添加修饰符 onlyOwnerOf 到 approve。

函数的正文部分，将 \_tokenId 的 zombieApprovals 设置为和 \_to 相等。

最后，在 ERC721 规范里有一个 Approval 事件。所以我们应该在这个函数的最后触发这个事件。（参考 erc721.sol 来确认传入的参数，并确保 \_owner 是 msg.sender）

**第8章: ERC721: takeOwnership**

太棒了，现在让我们完成最后一个函数来结束 ERC721 的实现。（别担心，这后面我们还会讲更多内容😉）

最后一个函数 takeOwnership， 应该只是简单地检查以确保 msg.sender 已经被批准来提取这个代币或者僵尸。若确认，就调用 **\_transfer**；

**实战演习**

首先，我们要用一个 require 句式来检查 \_tokenId 的 zombieApprovals 和 msg.sender 相等。

这样如果 msg.sender 未被授权来提取这个代币，将抛出一个错误。

为了调用 \_transfer，我们需要知道代币所有者的地址（它需要一个 \_from 来作为参数）。幸运的是我们可以在我们的 ownerOf 函数中来找到这个参数。

所以，定义一个名为 owner 的 address 变量，并使其等于 ownerOf(\_tokenId)。

最后，调用 \_transfer, 并传入所有必须的参数。（在这里你可以用 msg.sender 作为 \_to， 因为代币正是要发送给调用这个函数的人）。

注意： 我们完全可以用一行代码来实现第2、3两步。但是分开写会让代码更易读。一点个人建议 :)

**第9章: 预防溢出**

恭喜你，我们完成了 ERC721 的实现。

并不是很复杂，对吧？很多类似的以太坊概念，当你只听人们谈论它们的时候，会觉得很复杂。所以最简单的理解方式就是你自己来实现它。

不过要记住那只是最简单的实现。还有很多的特性我们也许想加入到我们的实现中来，比如一些额外的检查，来确保用户不会不小心把他们的僵尸转移给0 地址（这被称作 “烧币”, 基本上就是把代币转移到一个谁也没有私钥的地址，让这个代币永远也无法恢复）。 或者在 DApp 中加入一些基本的拍卖逻辑。（你能想出一些实现的方法么？）

但是为了让我们的课程不至于离题太远，所以我们只专注于一些基础实现。如果你想学习一些更深层次的实现，可以在这个教程结束后，去看看 OpenZeppelin 的 ERC721 合约。

**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。

**实战演习**

我们已经帮你把 OpenZeppelin 的 SafeMath 库包含进 safemath.sol了，如果你想看一下代码的话，现在可以看看，不过我们下一章将深入进去。

首先我们来告诉我们的合约要使用 SafeMath。我们将在我们的 ZombieFactory 里调用，这是我们的基础合约 — 这样其他所有继承出去的子合约都可以使用这个库了。

将 safemath.sol 引入到 zombiefactory.sol.

添加定义： using SafeMath for uint256;.

**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);

**实战演习**

在 ZombieOwnership 中有两个地方用到了数学运算，来替换成 SafeMath 方法把。

将 ++ 替换成 SafeMath 方法。

将 -- 替换成 SafeMath 方法。

**Putting it to the Test**

分配：

声明我们将为 uint32 使用SafeMath32。

声明我们将为 uint16 使用SafeMath16。

在 ZombieFactory 里还有一处我们也应该使用 SafeMath 的方法， 我们已经在那里留了注释提醒你。

**第13章: 注释**

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。

**实战演习**

如果你还没注意到：CryptoZombies 的答案检查器在工作的时候将忽略所有的注释。所以这一章我们其实无法检查你的 natspec 注释了。全靠你自己咯。

话说回来，到现在你应该已经是一个 Solidity 小能手了。我们就假定你已经学会这些了。

大胆去做些尝试把，给 ZombieOwnership 加上一些 natspec 标签:

@title — 例如：一个管理转移僵尸所有权的合约

@author — 你的名字

@dev — 例如：符合 OpenZeppelin 对 ERC721 标准草案的实现

**Lesson 6: 应用前端和 Web3.js**

哟，你都学到这里来啦。你真是个了不得的 CryptoZombie...

通过前五课的学习，相信你已经有了扎实的 Solidity 基础.

但是若没有一个互动界面让用户来使用，一个 DApp 就不能称之为完整。

在这一课，我们将来学习如果用一个名为 **Web3.js** 的库来为你的 DApp 创建一个基本的前端界面，和你的智能合约交互。

需要注意的是这个 APP 界面将使用 **JavaScript** 来写，并不是 Solidity。因为我们的课程专注于 Ethereum / Solidity，我们就暂时假定你已经会用HTML, JavaScript(包括 ES6 promises)，以及 JQuery 写网站了。因此我们不会花时间来介绍这些技术的基础知识。

如果你还不会用 HTML/JavaScript 来写网站，你应该先去学习一下这方面的基础知识再来继续接下来的课程。

**第1章: 介绍 Web3.js**

完成第五课以后，我们的僵尸 DApp 的 Solidity 合约部分就完成了。现在我们来做一个基本的网页好让你的用户能玩它。 要做到这一点，我们将使用以太坊基金发布的 JavaScript 库 —— ***Web3.js***.

**什么是 Web3.js?**

还记得么？以太坊网络是由节点组成的，每一个节点都包含了区块链的一份拷贝。当你想要调用(call)一份智能合约的一个方法(function)，你需要从其中一个节点中查找并告诉它:

智能合约的地址

你想调用的方法，以及

你想传入那个方法的参数

以太坊节点只能识别一种叫做 ***JSON-RPC*** 的语言。这种语言直接读起来并不好懂。当你你想调用一个合约的方法的时候，需要发送的查询语句将会是这样的：

// 哈……祝你写所有这样的函数调用的时候都一次通过
// 往右边拉…… ==>
{"jsonrpc":"2.0","method":"eth\_sendTransaction","params":[{"from":"0xb60e8dd61c5d32be8058bb8eb970870f07233155","to":"0xd46e8dd67c5d32be8058bb8eb970870f07244567","gas":"0x76c0","gasPrice":"0x9184e72a000","value":"0x9184e72a","data":"0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675"}],"id":1}

幸运的是 Web3.js 把这些令人讨厌的查询语句都隐藏起来了， 所以你只需要与方便易懂的 JavaScript 界面进行交互即可。

你不需要构建上面的查询语句，在你的代码中调用一个函数看起来将是这样：

CryptoZombies.methods.createRandomZombie("Vitalik Nakamoto 🤔")
.send({ from: "0xb60e8dd61c5d32be8058bb8eb970870f07233155", gas: "3000000" })

我们将在接下来的几章详细解释这些语句，不过首先我们来把 Web3.js 环境搭建起来。

**准备好了么？**

取决于你的项目工作流程和你的爱好，你可以用一些常用工具把 Web3.js 添加进来：

// 用 NPM
npm install web3
// 用 Yarn
yarn add web3
// 用 Bower
bower install web3
// ...或者其他。

甚至，你可以从 github 直接下载压缩后的 .js 文件 然后包含到你的项目文件中：

<script language="javascript" type="text/javascript" src="web3.min.js"></script>

因为我们不想让你花太多在项目环境搭建上，在本教程中我们将使用上面的 script 标签来将 Web3.js 引入。

**实战演习**

我们为你建立了一个HTML 项目空壳 —— index.html。假设在和 index.html 同个文件夹里有一份 web3.min.js

使用上面的 script 标签代码把 web3.js 添加进去以备接下来使用。

**第2章: Web3 提供者**

太棒了。现在我们的项目中有了Web3.js, 来初始化它然后和区块链对话吧。

首先我们需要 ***Web3 Provider***.

要记住，以太坊是由共享同一份数据的相同拷贝的 **\_节点\_** 构成的。 在 Web3.js 里设置 Web3 的 Provider（提供者） 告诉我们的代码应该和 **哪个节点** 交互来处理我们的读写。这就好像在传统的 Web 应用程序中为你的 API 调用设置远程 Web 服务器的网址。

你可以运行你自己的以太坊节点来作为 Provider。 不过，有一个第三方的服务，可以让你的生活变得轻松点，让你不必为了给你的用户提供DApp而维护一个以太坊节点— ***Infura***.

**Infura**

Infura 是一个服务，它维护了很多以太坊节点并提供了一个缓存层来实现高速读取。你可以用他们的 API 来免费访问这个服务。 用 Infura 作为节点提供者，你可以不用自己运营节点就能很可靠地向以太坊发送、接收信息。

你可以通过这样把 Infura 作为你的 Web3 节点提供者：

var web3 = new Web3(new Web3.providers.WebsocketProvider("wss://mainnet.infura.io/ws"));

不过，因为我们的 DApp 将被很多人使用，这些用户不单会从区块链读取信息，还会向区块链 **\_写\_** 入信息，我们需要用一个方法让用户可以用他们的私钥给事务签名。

注意: 以太坊 (以及通常意义上的 blockchains )使用一个公钥/私钥对来对给事务做数字签名。把它想成一个数字签名的异常安全的密码。这样当我修改区块链上的数据的时候，我可以用我的公钥来 **证明** 我就是签名的那个。但是因为没人知道我的私钥，所以没人能伪造我的事务。

加密学非常复杂，所以除非你是个专家并且的确知道自己在做什么，你最好不要在你应用的前端中管理你用户的私钥。

不过幸运的是，你并不需要，已经有可以帮你处理这件事的服务了： ***Metamask***.

**Metamask**

Metamask 是 Chrome 和 Firefox 的浏览器扩展， 它能让用户安全地维护他们的以太坊账户和私钥， 并用他们的账户和使用 Web3.js 的网站互动（如果你还没用过它，你肯定会想去安装的——这样你的浏览器就能使用 Web3.js 了，然后你就可以和任何与以太坊区块链通信的网站交互了）

作为开发者，如果你想让用户从他们的浏览器里通过网站和你的DApp交互（就像我们在 CryptoZombies 游戏里一样），你肯定会想要兼容 Metamask 的。

**注意**: Metamask 默认使用 Infura 的服务器做为 web3 提供者。 就像我们上面做的那样。不过它还为用户提供了选择他们自己 Web3 提供者的选项。所以使用 Metamask 的 web3 提供者，你就给了用户选择权，而自己无需操心这一块。

**使用 Metamask 的 web3 提供者**

Metamask 把它的 web3 提供者注入到浏览器的全局 JavaScript对象web3中。所以你的应用可以检查 web3 是否存在。若存在就使用 web3.currentProvider 作为它的提供者。

这里是一些 Metamask 提供的示例代码，用来检查用户是否安装了MetaMask，如果没有安装就告诉用户需要安装MetaMask来使用我们的应用。

window.addEventListener('load', function() {
// 检查web3是否已经注入到(Mist/MetaMask)
if (typeof web3 !== 'undefined') {
// 使用 Mist/MetaMask 的提供者
web3js = new Web3(web3.currentProvider);
} else {
// 处理用户没安装的情况， 比如显示一个消息
// 告诉他们要安装 MetaMask 来使用我们的应用
}
// 现在你可以启动你的应用并自由访问 Web3.js:
startApp()
})

你可以在你所有的应用中使用这段样板代码，好检查用户是否安装以及告诉用户安装 MetaMask。

注意: 除了MetaMask，你的用户也可能在使用其他他的私钥管理应用，比如 **Mist** 浏览器。不过，它们都实现了相同的模式来注入 web3 变量。所以我这里描述的方法对两者是通用的。

**实战演习**

我们在HTML文件中的 </body> 标签前面放置了一个空的 script 标签。可以把这节课的 JavaScript 代码写在里面。

把上面用来检测 MetaMask 是否安装的模板代码粘贴进来。请粘贴到以 window.addEventListener 开头的代码块中。

**第3章: 和合约对话**

现在，我们已经用 MetaMask 的 Web3 提供者初始化了 Web3.js。接下来就让它和我们的智能合约对话吧。

Web3.js 需要两个东西来和你的合约对话: 它的 **\_地址\_** 和它的 ***ABI（Appli Binary Intface）***。

**合约地址**

在你写完了你的智能合约后，你需要编译它并把它部署到以太坊。我们将在**下一课**中详述**部署**，因为它和写代码是截然不同的过程，所以我们决定打乱顺序，先来讲 Web3.js。

在你部署智能合约以后，它将获得一个以太坊上的永久地址。如果你还记得第二课，CryptoKitties 在以太坊上的地址是 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d。

你需要在部署后复制这个地址以来和你的智能合约对话。

**合约 ABI**

另一个 Web3.js 为了要和你的智能合约对话而需要的东西是 ***ABI***。

ABI 意为应用二进制接口（Application Binary Interface）。 基本上，它是以 JSON 格式表示合约的方法，告诉 Web3.js 如何以合同理解的方式格式化函数调用。

当你编译你的合约向以太坊部署时(我们将在第七课详述)， Solidity 编译器会给你 ABI，所以除了合约地址，你还需要把这个也复制下来。

因为我们这一课不会讲述部署，所以现在我们已经帮你编译了 ABI 并放在了名为cryptozombies\_abi.js的文件中，保存在一个名为 cryptoZombiesABI 的变量中。

如果我们将cryptozombies\_abi.js 包含进我们的项目，我们就能通过那个变量访问 CryptoZombies ABI 。

**实例化 Web3.js**

一旦你有了合约的地址和 ABI，你可以像这样来实例化 Web3.js。

// 实例化 myContract
var myContract = new web3js.eth.Contract(myABI, myContractAddress);

**实战演习**

在文件的 <head> 标签块中，用 script 标签引入cryptozombies\_abi.js，好把 ABI 的定义引入项目。

在 <body> 里的 <script> 开头 , 定义一个var，取名 cryptoZombies， 不过不要对其赋值，稍后我们将用这个这个变量来存储我们实例化合约。

接下来，创建一个名为 startApp() 的 function。 接下来两步来完成这个方法。

startApp() 里应该做的第一件事是定义一个名为cryptoZombiesAddress 的变量并赋值为"你的合约地址" (这是你的合约在以太坊主网上的地址)。

最后，来实例化我们的合约。模仿我们上面的代码，将 cryptoZombies 赋值为 new web3js.eth.Contract (使用我们上面代码中通过 script 引入的 cryptoZombiesABI 和 cryptoZombiesAddress)。

**第4章: 调用和合约函数**

我们的合约配置好了！现在来用 Web3.js 和它对话。

Web3.js 有两个方法来调用我们合约的函数: call and send.

**Call**

call 用来调用 view 和 pure 函数。它只运行在本地节点，不会在区块链上创建事务。

**复习:** view 和 pure 函数是只读的并不会改变区块链的状态。它们也不会消耗任何gas。用户也不会被要求用MetaMask对事务签名。

使用 Web3.js，你可以如下 call 一个名为myMethod的方法并传入一个 123 作为参数：

myContract.methods.myMethod(123).call()

**Send**

send 将创建一个事务并改变区块链上的数据。你需要用 send 来调用任何非 view 或者 pure 的函数。

**注意:** send 一个事务将要求用户支付gas，并会要求弹出对话框请求用户使用 Metamask 对事务签名。在我们使用 Metamask 作为我们的 web3 提供者的时候，所有这一切都会在我们调用 send() 的时候自动发生。而我们自己无需在代码中操心这一切，挺爽的吧。

使用 Web3.js, 你可以像这样 send 一个事务调用myMethod 并传入 123 作为参数：

myContract.methods.myMethod(123).send()

语法几乎 call()一模一样。

**获取僵尸的数据**

来看一个使用 call 读取我们合约数据的真实例子

回忆一下，我们定义我们的僵尸数组为 公开(public):

Zombie[] public zombies;

在 Solidity 里，当你定义一个 public变量的时候， 它将自动定义一个公开的 "getter" 同名方法， 所以如果你像要查看 id 为 15 的僵尸，你可以像一个函数一样调用它： zombies(15).

这是如何在外面的前端界面中写一个 JavaScript 方法来传入一个僵尸 id，在我们的合同中查询那个僵尸并返回结果

注意: 本课中所有的示例代码都使用 Web3.js 的 **1.0 版**，此版本使用的是 Promises 而不是回调函数。你在线上看到的其他教程可能还在使用老版的 Web3.js。在1.0版中，语法改变了不少。如果你从其他教程中复制代码，先确保你们使用的是相同版本的Web3.js。

function getZombieDetails(id) {
return cryptoZombies.methods.zombies(id).call()
}
// 调用函数并做一些其他事情
getZombieDetails(15)
.then(function(result) {
console.log("Zombie 15: " + JSON.stringify(result));
});

我们来看看这里都做了什么

cryptoZombies.methods.zombies(id).call() 将和 Web3 提供者节点通信，告诉它返回从我们的合约中的 Zombie[] public zombies，id为传入参数的僵尸信息。

注意这是 **异步的**，就像从外部服务器中调用API。所以 Web3 在这里返回了一个 Promises. (如果你对 JavaScript的 Promises 不了解，最好先去学习一下这方面知识再继续)。

一旦那个 promise 被 resolve, (意味着我们从 Web3 提供者那里获得了响应)，我们的例子代码将执行 then 语句中的代码，在控制台打出 result。

result 是一个像这样的 JavaScript 对象：

{
"name": "H4XF13LD MORRIS'S COOLER OLDER BROTHER",
"dna": "1337133713371337",
"level": "9999",
"readyTime": "1522498671",
"winCount": "999999999",
"lossCount": "0" // Obviously.
}

我们可以用一些前端逻辑代码来解析这个对象并在前端界面友好展示。

**实战演习**

我们已经帮你把 getZombieDetails 复制进了代码。

先为zombieToOwner 创建一个类似的函数。如果你还记得 ZombieFactory.sol，我们有一个长这样的映射：

mapping (uint => address) public zombieToOwner;

定义一个 JavaScript 方法，起名为 zombieToOwner。和上面的 getZombieDetails 类似， 它将接收一个id 作为参数，并返回一个 Web3.js call 我们合约里的zombieToOwner 。

之后在下面，为 getZombiesByOwner 定义一个方法。如果你还能记起 ZombieHelper.sol，这个方法定义像这样：

function getZombiesByOwner(address \_owner)

我们的 getZombiesByOwner 方法将接收 owner 作为参数，并返回一个对我们函数 getZombiesByOwner的 Web3.js call

**第5章: MetaMask 和账户**

太棒了！你成功地写了一些前端代码来和你的第一个智能合约交互。

接下来我们综合一下——比如我们想让我们应用的首页显示用户的整个僵尸大军。

毫无疑问我们首先需要用 getZombiesByOwner(owner) 来查询当前用户的所有僵尸ID。

但是我们的 Solidity 合约需要 owner 作为 Solidity address。我们如何能知道应用用户的地址呢？

**获得 MetaMask中的用户账户**

MetaMask 允许用户在扩展中管理多个账户。

我们可以通过这样来获取 web3 变量中激活的当前账户：

var userAccount = web3.eth.accounts[0]

因为用户可以随时在 MetaMask 中切换账户，我们的应用需要监控这个变量，一旦改变就要相应更新界面。例如，若用户的首页展示它们的僵尸大军，当他们在 MetaMask 中切换了账号，我们就需要更新页面来展示新选择的账户的僵尸大军。

我们可以通过 setInterval 方法来做:

var accountInterval = setInterval(function() {
// 检查账户是否切换
if (web3.eth.accounts[0] !== userAccount) {
userAccount = web3.eth.accounts[0];
// 调用一些方法来更新界面
updateInterface();
}
}, 100);

这段代码做的是，每100毫秒检查一次 userAccount 是否还等于 web3.eth.accounts[0] (比如：用户是否还激活了那个账户)。若不等，则将 当前激活用户赋值给 userAccount，然后调用一个函数来更新界面。

**实战演习**

我们来让应用在页面第一次加载的时候显示用户的僵尸大军，监控当前 MetaMask 中的激活账户，并在账户发生改变的时候刷新显示。

定义一个名为userAccount的变量，不给任何初始值。

在 startApp()函数的最后，复制粘贴上面样板代码中的 accountInterval 方法进去。

将 updateInterface();替换成一个 getZombiesByOwner 的 call 函数，并传入 userAccount。

在 getZombiesByOwner 后面链式调用then 语句，并将返回的结果传入名为 displayZombies 的函数。 (语句像这样: .then(displayZombies);).

我们还没有 displayZombies 函数，将于下一章实现。

**第6章: 显示僵尸大军**

如果我们不向你展示如何显示你从合约获取的数据，那这个教程就太不完整了。

在实际应用中，你肯定想要在应用中使用诸如 React 或 Vue.js 这样的前端框架来让你的前端开发变得轻松一些。不过要教授 React 或者 Vue.js 知识的话，就大大超出了本教程的范畴——它们本身就需要几节课甚至一整个教程来教学。

所以为了让 CryptoZombies.io 专注于以太坊和智能合约，我们将使用 JQuery 来做一个快速示例，展示如何解析和展示从智能合约中拿到的数据。

**显示僵尸数据 —— 一个粗略的例子**

我们已经在代码中添加了一个空的代码块 <div id="zombies"></div>， 在 displayZombies 方法中也同样有一个。

回忆一下在之前章节中我们在 startApp() 方法内部调用了 displayZombies 并传入了 call getZombiesByOwner 获得的结果，它将被传入一个僵尸ID数组，像这样：

[0, 13, 47]

因为我们想让我们的 displayZombies 方法做这些事:

首先清除 #zombies 的内容以防里面已经有什么内容（这样当用户切换账号的时候，之前账号的僵尸大军数据就会被清除）

循环遍历 id，对每一个id调用 getZombieDetails(id)， 从我们的合约中获得这个僵尸的数据。

将获得的僵尸数据放进一个HTML模板中以格式化显示，追加进 #zombies 里面。

再次声明，我们只用了 JQuery，没有任何模板引擎，所以会非常丑。不过这只是一个如何展示僵尸数据的示例而已。

**如何来展示僵尸元素呢？**

在上面的例子中，我们只是简单地用字符串来显示 DNA。不过在你的 DApp 中，你将需要把 DNA 转换成图片来显示你的僵尸。

我们通过把 DNA 字符串分割成小的字符串来做到这一点，每2位数字代表一个图片，类似这样：

// 得到一个 1-7 的数字来表示僵尸的头:
var head = parseInt(zombie.dna.substring(0, 2)) % 7 + 1
// 我们有7张头部图片：
var headSrc = "../assets/zombieparts/head-" + i + ".png"

每一个模块都用 CSS 绝对定位来显示，在一个上面叠加另外一个。

如果你想看我们的具体实现，我们将用来展示僵尸形象的 Vue.js 模块开源了： [点击这里](https://github.com/loomnetwork/zombie-char-component).

不过，因为那个文件中有太多行代码， 超出了本教程的讨论范围。我们依然还是使用上面超级简单的 JQuery 实现，把美化僵尸的工作作为家庭作业留给你了😉

**实战演习**

我们为你创建了一个空的 displayZombies 方法。来一起实现它。

首先我们需要清空 #zombies 的内容。 用JQuery，你可以这样做： $("#zombies").empty();。

接下来，我们要循环遍历所有的 id，循环这样用： for (id of ids) {

在循环内部，复制粘贴上面的代码，对每一个id调用 getZombieDetails(id)，然后用 $("#zombies").append(...) 把内容追加进我们的 HTML 里面。

**第7章: 发送事务**

这下我们的界面能检测用户的 MetaMask 账户，并自动在首页显示它们的僵尸大军了，有没有很棒？

现在我们来看看用 **send** 函数来修改我们智能合约里面的数据。

相对 call 函数，send 函数有如下主要区别:

send 一个事务需要一个 from 地址来表明谁在调用这个函数（也就是你 Solidity 代码里的 msg.sender )。 我们需要这是我们 DApp 的用户，这样一来 MetaMask 才会弹出提示让他们对事务签名。

send 一个事务将花费 gas

在用户 send 一个事务到该事务对区块链产生实际影响之间有一个不可忽略的延迟。这是因为我们必须等待事务被包含进一个区块里，以太坊上一个区块的时间平均下来是15秒左右。如果当前在以太坊上有大量挂起事务或者用户发送了过低的 gas 价格，我们的事务可能需要等待数个区块才能被包含进去，往往可能花费数分钟。

所以在我们的代码中我们需要编写逻辑来处理这部分异步特性。

**生成一个僵尸**

我们来看一个合约中一个新用户将要调用的第一个函数: createRandomZombie.

作为复习，这里是合约中的 Solidity 代码：

function createRandomZombie(string \_name) public {
require(ownerZombieCount[msg.sender] == 0);
uint randDna = \_generateRandomDna(\_name);
randDna = randDna - randDna % 100;
\_createZombie(\_name, randDna);
}

这是如何在用 MetaMask 在 Web3.js 中调用这个函数的示例:

function createRandomZombie(name) {
// 这将需要一段时间，所以在界面中告诉用户这一点
// 事务被发送出去了
$("#txStatus").text("正在区块链上创建僵尸，这将需要一会儿...");
// 把事务发送到我们的合约:
return cryptoZombies.methods.createRandomZombie(name)
.send({ from: userAccount })
.on("receipt", function(receipt) {
$("#txStatus").text("成功生成了 " + name + "!");
// 事务被区块链接受了，重新渲染界面
getZombiesByOwner(userAccount).then(displayZombies);
})
.on("error", function(error) {
// 告诉用户合约失败了
$("#txStatus").text(error);
});
}

我们的函数 send 一个事务到我们的 Web3 提供者，然后链式添加一些事件监听:

receipt 将在合约被包含进以太坊区块上以后被触发，这意味着僵尸被创建并保存进我们的合约了。

error 将在事务未被成功包含进区块后触发，比如用户未支付足够的 gas。我们需要在界面中通知用户事务失败以便他们可以再次尝试。

注意:你可以在调用 send 时选择指定 gas 和 gasPrice， 例如： .send({ from: userAccount, gas: 3000000 })。如果你不指定，MetaMask 将让用户自己选择数值。

**实战演习**

我们添加了一个div， 指定 ID 为 txStatus — 这样我们可以通过更新这个 div 来通知用户事务的状态。

在 displayZombies下面， 复制粘贴上面 createRandomZombie 的代码。

我们来实现另外一个函数 feedOnKitty：

调用 feedOnKitty 的逻辑几乎一样 — 我们将发送一个事务来调用这个函数，并且成功的事务会为我们创建一个僵尸，所以我们希望在成功后重新绘制界面。

在 createRandomZombie 下面复制粘贴它的代码，改动这些地方:

a) 给其命名为 feedOnKitty， 它将接收两个参数 zombieId 和 kittyId

b) #txStatus 的文本内容将更新为: "正在吃猫咪，这将需要一会儿..."

c) 让其调用我们合约里面的 feedOnKitty 函数并传入相同的参数

d) #txStatus 里面的的成功信息应该是 "吃了一只猫咪并生成了一只新僵尸！"

**第8章: 调用 Payable 函数**

attack, changeName, 以及 changeDna 的逻辑将非常雷同，所以本课将不会花时间在上面。

实际上，在调用这些函数的时候已经有了非常多的重复逻辑。所以最好是重构代码把相同的代码写成一个函数。（并对txStatus使用模板系统——我们已经看到用类似 Vue.js 类的框架是多么整洁）

我们来看看另外一种 Web3.js 中需要特殊对待的函数 —— payable 函数。

**升级！**

回忆一下在 ZombieHelper 里面，我们添加了一个 payable 函数，用户可以用来升级:

function levelUp(uint \_zombieId) external payable {
require(msg.value == levelUpFee);
zombies[\_zombieId].level++;
}

和函数一起发送以太非常简单，只有一点需要注意： 我们需要指定发送多少 wei，而不是以太。

**啥是 Wei?**

一个 wei 是以太的最小单位 — 1 ether 等于 10^18 wei

太多0要数了，不过幸运的是 Web3.js 有一个转换工具来帮我们做这件事：

// 把 1 ETH 转换成 Wei
web3js.utils.toWei("1", "ether");

在我们的 DApp 里， 我们设置了 levelUpFee = 0.001 ether，所以调用 levelUp 方法的时候，我们可以让用户用以下的代码同时发送 0.001 以太:

cryptoZombies.methods.levelUp(zombieId)
.send({ from: userAccount, value: web3js.utils.toWei("0.001","ether") })

**实战演习**

在 feedOnKitty 下面添加一个 levelUp 方法。代码和 feedOnKitty 将非常相似。不过：

函数将接收一个参数, zombieId

在发送事务之前，txStatus 的文本应该是 "正在升级您的僵尸..."

当它调用合约里的levelUp时，它应该发送"0.001" ETH，并用 toWei 转换，如同上面例子里那样。

成功之后应该显示 "不得了了！僵尸成功升级啦！"

我们 **不** 需要在调用 getZombiesByOwner 后重新绘制界面 — 因为在这里我们只是修改了僵尸的级别而已。

**第9章: 订阅事件**

如你所见，通过 Web3.js 和合约交互非常简单直接——一旦你的环境建立起来， call 函数和 send 事务和普通的网络API并没有多少不同。

还有一点东西我们想要讲到——订阅合约事件

**监听新僵尸事件**

如果你还记得 zombiefactory.sol，每次新建一个僵尸后，我们会触发一个 NewZombie 事件：

event NewZombie(uint zombieId, string name, uint dna);

在 Web3.js里， 你可以 **订阅** 一个事件，这样你的 Web3 提供者可以在每次事件发生后触发你的一些代码逻辑：

cryptoZombies.events.NewZombie()
.on("data", function(event) {
let zombie = event.returnValues;
console.log("一个新僵尸诞生了！", zombie.zombieId, zombie.name, zombie.dna);
}).on('error', console.error);

注意这段代码将在 **任何** 僵尸生成的时候激发一个警告信息——而不仅仅是当前用户的僵尸。如果我们只想对当前用户发出提醒呢？

**使用 indexed**

为了筛选仅和当前用户相关的事件，我们的 Solidity 合约将必须使用 indexed 关键字，就像我们在 ERC721 实现中的Transfer 事件中那样：

event Transfer(address indexed \_from, address indexed \_to, uint256 \_tokenId);

在这种情况下， 因为\_from 和 \_to 都是 indexed，这就意味着我们可以在前端事件监听中过滤事件

cryptoZombies.events.Transfer({ filter: { \_to: userAccount } })
.on("data", function(event) {
let data = event.returnValues;
// 当前用户更新了一个僵尸！更新界面来显示
}).on('error', console.error);

看到了吧， 使用 event 和 indexed 字段对于监听合约中的更改并将其反映到 DApp 的前端界面中是非常有用的做法。

**查询过去的事件**

我们甚至可以用 getPastEvents 查询过去的事件，并用过滤器 fromBlock 和 toBlock 给 Solidity 一个事件日志的时间范围("block" 在这里代表以太坊区块编号）：

cryptoZombies.getPastEvents("NewZombie", { fromBlock: 0, toBlock: 'latest' })
.then(function(events) {
// events 是可以用来遍历的 `event` 对象
// 这段代码将返回给我们从开始以来创建的僵尸列表
});

因为你可以用这个方法来查询从最开始起的事件日志，这就有了一个非常有趣的用例： **用事件来作为一种更便宜的存储**。

若你还能记得，在区块链上保存数据是 Solidity 中最贵的操作之一。但是用事件就便宜太多太多了。

这里的短板是，事件不能从智能合约本身读取。但是，如果你有一些数据需要永久性地记录在区块链中以便可以在应用的前端中读取，这将是一个很好的用例。这些数据不会影响智能合约向前的状态。

举个栗子，我们可以用事件来作为僵尸战斗的历史纪录——我们可以在每次僵尸攻击别人以及有一方胜出的时候产生一个事件。智能合约不需要这些数据来计算任何接下来的事情，但是这对我们在前端向用户展示来说是非常有用的东西。

**Web3.js 事件 和 MetaMask**

上面的示例代码是针对 Web3.js 最新版1.0的，此版本使用了 ***WebSockets*** 来订阅事件。

但是，MetaMask 尚且不支持最新的事件 API (尽管如此，他们已经在实现这部分功能了， [点击这里](https://github.com/MetaMask/metamask-extension/issues/3642) 查看进度)

所以现在我们必须使用一个单独 Web3 提供者，它针对事件提供了WebSockets支持。 我们可以用 Infura 来像实例化第二份拷贝：

var web3Infura = new Web3(new Web3.providers.WebsocketProvider("wss://mainnet.infura.io/ws"));
var czEvents = new web3Infura.eth.Contract(cryptoZombiesABI, cryptoZombiesAddress);

然后我们将使用 czEvents.events.Transfer 来监听事件，而不再使用 cryptoZombies.events.Transfer。我们将继续在课程的其他部分使用 cryptoZombies.methods。

将来，在 MetaMask 升级了 API 支持 Web3.js 后，我们就不用这么做了。但是现在我们还是要这么做，以使用 Web3.js 更好的最新语法来监听事件。

**放在一起**

来添加一些代码监听 Transfer 事件，并在当前用户获得一个新僵尸的时候为他更新界面。

我们将需要在 startApp 底部添加代码，以保证在添加事件监听器之前 cryptoZombies 已经初始化了。

在 startApp()底部，为 cryptoZombies.events.Transfer 复制粘贴上面的2行事件监听代码块

复制监听 Transfer 事件的代码块，并用 \_to: userAccount 过滤。要记得把 cryptoZombies 换成 czEvents 好在这 里使用 Infura 而不是 MetaMask 来作为提供者。

用 getZombiesByOwner(userAccount).then(displayZombies); 来更新界面

**第10章: 放在一起**

恭喜啊少年，你已经成功编写了一个 Web3.js 前端界面来和你的智能合约交互

**接下来的步骤**

这节课的内容非常基础。我们想要给你展示和智能合约交互的核心内容，而并不想用太多的时间来教你完整实现。我们也不想花太多时间在HTML/CSS上，因为大部分人都已经知道了。

所以我们把一些实现略去了。这里是你要完整实现所需要完成的基本事项列表：

为 attack, changeName, changeDna 以及 ERC721 函数 transfer, ownerOf, balanceOf 等实现前端函数。这些函数的实现将和我们讲过的 send事务的函数非常相似。

实现一个“管理界面”，在那里你可以调用 setKittyContractAddress, setLevelUpFee, 以及 withdraw。再次，在前端这块没有什么特别的代码——这些实现之间将非常相似。你应该保证从部署合同时候相同的以太坊地址调用这些函数，因为他们都有onlyOwner 修饰符。

在应用里我们还应该实现一些其他的界面：

a. 一个僵尸页面，在那里你可以查看一个特定僵尸的信息并可以分享它的链接。这个页面应该渲染僵尸的外形，展示它的名字，它的所有者（以及用户主页的链接），它的输赢次数，它的战斗记录等等。

b. 一个用户界面，在那里你可以查看用户的僵尸大军并分享它的链接。

c. 一个主页，就是用户页面的变体，可以展示当前用户的僵尸大军（正如我们在index.html）里面实现的那样。

界面中的一些方法允许用户用 CryptoKitties 喂食僵尸。我们可以给每一个僵尸添加一个按钮，叫做“给我投食”，再给一个输入框让用户输入一个猫咪的ID（或者一个猫咪的网址，比如https://www.cryptokitties.co/kitty/578397），它将触发我们的 feedOnKitty 函数。

界面中的一些方法将让用户用来攻击其他用户的僵尸

实现这点的一个方法是，当用户浏览其他用户的页面的时候，可以在对方僵尸旁边显示一个按钮，叫做“攻击这头僵尸”。当用户点击的时候，可以弹出一个模块，展示当前用户的僵尸大军并询问用户“你想用哪头僵尸出战？”

在用户的主页，也可以在每个僵尸旁边显示一个按钮，叫做“攻击一个僵尸”。当用户点击的时候，可以弹出一个模块，展示一个搜索框，可以让用户输入僵尸ID或者网址来搜索，或者也可以有一个按钮叫做“随机攻击一头僵尸”，将随机搜索一头僵尸来。

我们也建议你将在冷却期的僵尸用特殊的颜色显示，比如使其变成灰色。这样界面就能告诉用户不能用冷却期的僵尸来进攻。

在用户的主页，每一个僵尸也应该有选项可以更改名字、DNA、以及升级（通过付费）。若用户等级不到，无法使用的选项应该标灰。

对于新用户，我们应该显示一个欢迎信息，并让其确认用 createRandomZombie()创建一个新僵尸。

也可以为我们的智能合约添加一个包含indexed 的用户地址属性的 Attack 事件。这样就可以创建实时通知了——我们可以在用户的僵尸遭受攻击的时候弹出一条通知，这样他们可以看到谁在用什么僵尸攻击他们并做出报复。

我们也许还想实现一些前端缓存层，这样就不用总是为了相同的数据去访问Infura。（在我们当前实现中，displayZombies 将在每次页面刷新的时候为每一个僵尸调用 getZombieDetails，但是实际中我们将只需要为新加入的僵尸调用这个函数）

一个实时聊天室，这样你就可以在你击溃别人的僵尸大军的同时嘲讽他们？

因为这将需要大量的前端代码来实现全部的界面（HTML， CSS， JavaScript 以及诸如 React 和 Vue.js 这样的框架）。光实现一个这样的前端界面也许会花费多达10节课，所以我们将这个光荣的任务交给你自己去完成。

注意：尽管智能合约是去中心化的。这个用来和DApp交互的前端界面依然需要放在我们中心化的网络服务器上。不过，有了我们正在内测的Loom Network SDK，很快你就可以在应用自己的DApp链上运行前端界面而不是中心化的网络服务器。这样在以太坊和 Loom DApp 链上，你的整个应用都100%运行在区块链上了。

**总结**

学完第六课。你现在有了编写智能合约和前端界面来让用户交互的所需技能了。

在下一课。 我们将涉及这个历程中最后缺失的一部分——将你的智能合约部署到以太坊。

=====================================================================================