



以太坊综述

Ethereum

2018.10

为什么要学习以太坊

- 庞大的开发者社区，目前最大的区块链开发平台
- 相对较成熟，有代表性，资料众多
- 以应用入手，学习曲线不那么陡峭
- 与JavaScript结合紧密，方便开发人员上手



课程简介

课程名	子课程	主要内容	预计课时
以太坊基础	以太坊综述	以太坊整体介绍	3
	初识以太坊	钱包、测试网、简单交易	4
	以太坊客户端	客户端；Geth的安装和使用；搭建私链	5
深入理解以太坊	以太坊账户和合约	账户详解，合约特性	3
	以太坊交易、gas和EVM	交易详解，EVM简介	3
以太坊编程及应用	Solidity基础	Solidity语法，简单合约	6
	简单投票DApp	ganache，简单投票DApp	6
	web3.js及简单应用	web3.js API，转币脚本，监听脚本	6



课程名	子课程	主要内容	预计课时
深入理解合约工作流	合约工作流	深入理解合约工作流	1
	自动化编译和部署	编写编译脚本和部署脚本	5
	自动化测试	Ganache	4
深入理解以太坊原理	以太坊的理念与实现	白皮书，黄皮书	3
	源码结构及分析	代码结构，MPT，GHOST	3
DApp项目实战	基于token的投票	Truffle，加入token的合约	12
	基于ipfs的去中心化eBay	IPFS，多合约交互	24
	ICO DApp	next.js + react + material-UI + mocha	36

学习目标

- 掌握以太坊的基本概念和工作原理
- 理解以太坊与比特币的联系和区别
- 掌握以太坊客户端的使用
- 深入理解智能合约
- 掌握 Solidity 语法，并能够写出复杂的合约
- 掌握 web3.js 的调用，并能够实现具体的 DApp
- 综合运用各种工具，完成较复杂的项目

主要参考资料

- 《精通以太坊》 (Mastering Ethereum)
<https://github.com/ethereumbook/ethereumbook>
- 《以太坊白皮书》 (A Next-Generation Smart Contract and Decentralized Application Platform)
<https://github.com/ethereum/wiki/wiki/White-Paper>
- 《以太坊黄皮书》 (《以太坊：一种安全去中心化的通用交易账本 拜占庭版本》)
- 以太坊官方文档 (Ethereum Homestead Documentation)
<http://www.ethdocs.org/en/latest/index.html>
- Solidity官方文档
<https://solidity.readthedocs.io/en/latest/>

涉及工具

- MetaMask - 浏览器插件钱包
- Remix - 基于浏览器的 Solidity 在线编辑器
- Geth - 以太坊客户端 (go语言)
- web3.js – 以太坊 javascript API 库
- Ganache – 以太坊客户端 (测试环境私链)
- Truffle – 以太坊开发框架



环境准备

- Chrome浏览器（最新版本 70.0.3538.67）

- Linux 系统或虚拟机（ubuntu 16.04.3）

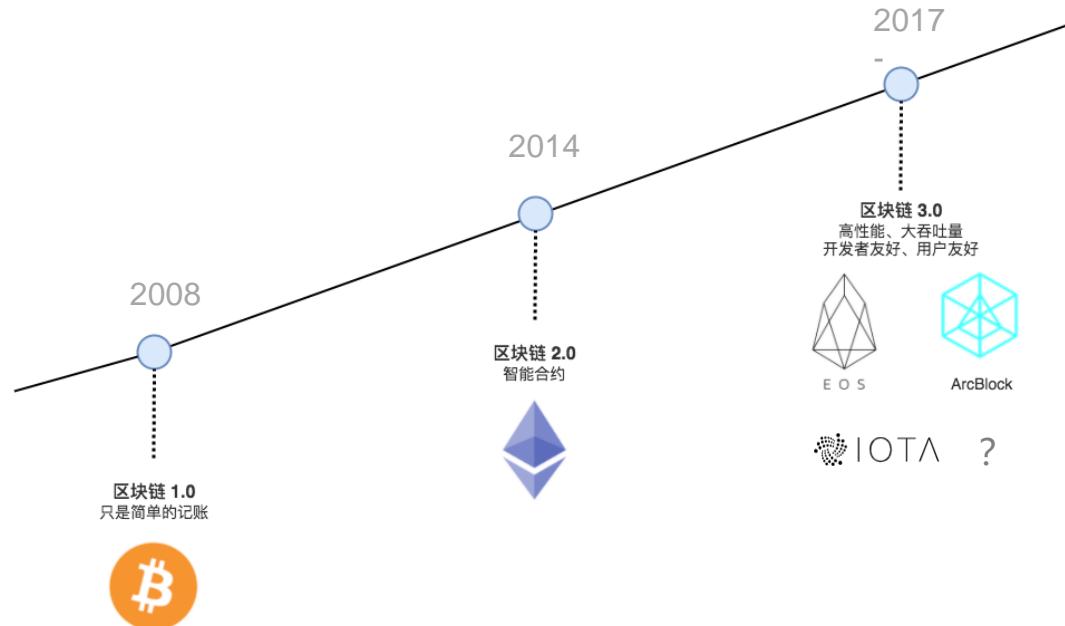
需要安装: *go(1.9), git(2.7.4), node(9.0.0), npm(5.7.1)*

- 文本编辑器（VisualCode）

- 科学上网工具



区块链（公链）发展简史



比特币 (1.0) -- 以太坊 (2.0) -- ? (3.0)



以太坊的出现

- 2014年1月，Vitalik Buterin在自己任编辑的比特币杂志(Bitcoin Magazine)上发表了《以太坊：一个下一代智能合约和去中心化应用平台》
(Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform)
- 2014年的迈阿密比特币会议中，布特林宣布了以太坊项目，并且提出了多项创新性区块链技术，该年7月，启动以太坊众筹募资，募得3.1万枚比特币（当时约合1840万美元）
- 2015年7月30日，当时作为以太坊项目CCO的成员Stephan Tual在官方微博上正式宣布了以太坊系统的诞生，以太坊主网上线



Vitalik Buterin

发展阶段

- “**前沿**” (**Frontier**) – Block #0

以太坊的初始阶段，持续时间为2015年7月30日至2016年3月

- “**家园**” (**Homestead**) - Block #1,150,000

以太坊的第二阶段，于2016年3月推出

- “**大都会**” (**Metropolis**) **Block #4,370,000**

以太坊的第三个阶段，于2017年10月推出的“拜占庭” (**Byzantium**)是 Metropolis 的两个硬分叉中的第一个，也是我们现在所处的阶段。

“**君士坦丁堡**” (**Constantinople**)

Metropolis 阶段的第二部分，计划于2018年推出。预计将包括切换到混合POW/POS 共识算法，以及其他变更。

- “**宁静**” (**Serenity**)

以太坊的第四个也是最后一个阶段。Serenity尚未有计划的发布日期。



重大分叉

- **Block #200,000**

“Ice Age” - 引入指数难度增加的硬分叉，促使向 Proof-of-Stake 过渡。

- **Block #1,192,000**

“The DAO” - 扭转了被攻击的DAO合约并导致以太坊和以太坊经典分裂成两个竞争系统的硬分叉。

- **Block #2,463,000**

“Tangerine Whistle” - 改变某些IO运算的 gas 计算，并从拒绝服务攻击中清除累积状态，该攻击利用了这些操作的低 gas 成本。

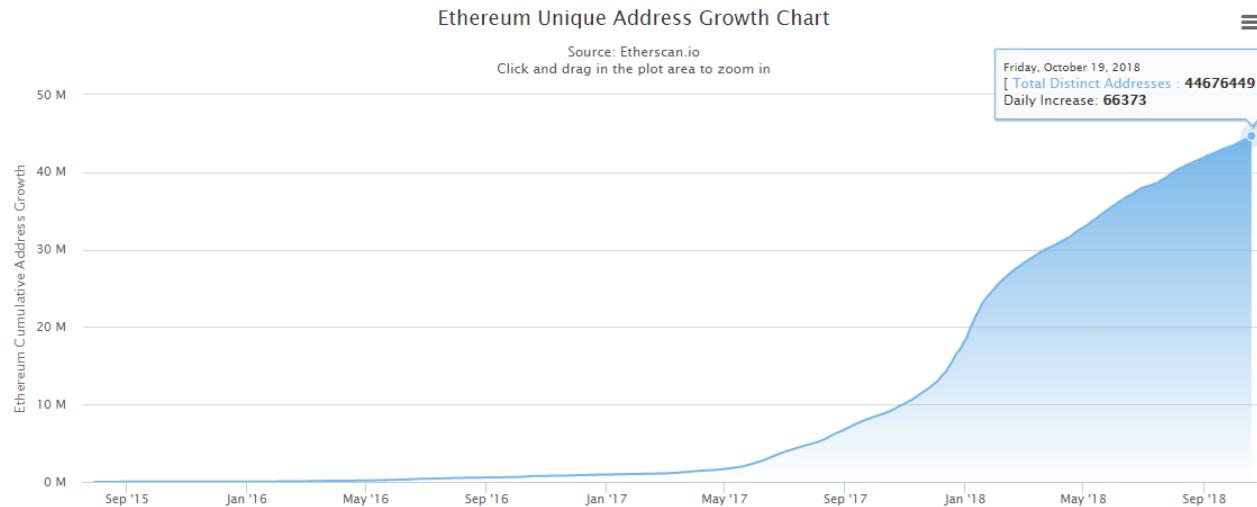
- **Block #2,675,000**

“Spurious Dragon” - 一个解决更多拒绝服务攻击媒介的硬分叉，以及另一种状态清除。此外，还有重放攻击保护机制。



发展现状

- 根据 State of DApps 的统计，目前运行在以太坊上的合约多达 47228 个；而以太坊的地址数也达到了 4000W 以上，如下图：





以太坊特点

- 以太坊是“世界计算机”，这代表它是一个开源的、全球分布的计算基础设施
- 执行称为智能合约 (smart contract) 的程序
- 使用区块链来同步和存储系统状态以及名为以太币 (ether) 的加密货币，以计量和约束执行资源成本
- 本质是一个基于交易的状态机(transaction-based state machine)
- 以太坊平台使开发人员能够构建具有内置经济功能的强大去中心化应用程序 (DApp)；在持续自我正常运行的同时，它还减少或消除了审查，第三方界面和交易对手风险



以太坊的组成部分

- **P2P网络**

以太坊在以太坊主网络上运行，该网络可在TCP端口30303上寻址，并运行一个名为EVP2p的协议。

- **交易 (Transaction)**

以太坊交易是网络消息，其中包括发送者 (sender) , 接收者 (receiver) , 值 (value) 和数据的有效载荷 (payload) 。

- **以太坊虚拟机 (EVM)**

以太坊状态转换由以太坊虚拟机 (EVM) 处理，这是一个执行字节码 (机器语言指令) 的基于堆栈的虚拟机。

- **数据库 (Blockchain)**

以太坊的区块链作为数据库 (通常是 Google 的 LevelDB) 本地存储在每个节点上，包含序列化后的交易和系统状态。

- **客户端**

以太坊有几种可互操作的客户端软件实现，其中最突出的是 Go-Ethereum (Geth) 和 Parity。

没有难学的技术



以太坊中的重要概念

- **账户 (Account)**

包含地址，余额和随机数，以及可选的存储和代码的对象。

- 普通账户 (EOA)，存储和代码均为空
- 合约账户 (Contract)，包含存储和代码

- **地址 (Address)**

一般来说，这代表一个EOA或合约，它可以在区块链上接收或发送交易。

更具体地说，它是ECDSA 公钥的 keccak 散列的最右边的160位。

- **交易 (Transaction)**

- 可以发送以太币和信息
- 向合约发送的交易可以调用合约代码，并以信息数据为函数参数
- 向空用户发送信息，可以自动生成以信息为代码块的合约账户

- **gas**

以太坊用于执行智能合约的虚拟燃料。以太坊虚拟机使用核算机制来衡量 gas 的消耗量并限制计算资源的消耗。



以太坊的货币

以太坊的货币单位称为以太（ether），也可以表示为ETH或符号Ξ。

以太币的发行规则：

- 挖矿前 (Pre-mine, Genesis)

2014年7月/8月间，为众筹大约发行了7200万以太币。这些币有的时候被称之为“矿前”。众筹阶段之后，以太币每年的产量基本稳定，被限制不超过7200万的25%

- 挖矿产出 (Mining)

——区块奖励 (block reward)

——叔块奖励 (uncle reward)

——叔块引用奖励 (uncle referencing reward)

- 以太币产量未来的变化

以太坊出块机制从工作量证明 (PoW) 转换为股权证明 (PoS) 后，以太币的发行会有什么变化尚未有定论。股权证明机制将使用一个称为Casper的协议。在Casper协议下，以太币的发行率将大大低于目前幽灵 (GHOST) 协议下的发行率。



以太坊的挖矿产出

- **区块奖励 (Block rewards)**

每产生一个新区块就会有一笔固定的奖励给矿工，初始是5个以太币，现在是3个。

- **叔块奖励 (Uncle rewards)**

有些区块被挖得稍晚一些，因此不能作为主区块链的组成部分。比特币称这类区块为“孤块”，并且完全舍弃它们。但是，以太币称它们为“叔块”(uncles)，并且在之后的区块中，可以引用它们。如果叔块在之后的区块链中作为叔块被引用，每个叔块会为挖矿者产出血块奖励的 $\frac{7}{8}$ 。这被称之为叔块奖励。

- **叔块引用奖励 (Uncle referencing rewards)**

矿工每引用一个叔块，可以得到区块奖励的 $\frac{1}{32}$ 作为奖励（最多引用两个叔块）

- 这样的一套基于POW的奖励机制，被称为以太坊的“幽灵协议”



以太币供应量



以太币供应量

102,681,639.34

Total Ether Supply

\$20,923,437,648

Market Capitalization

Breakdown By Supply Types



- Genesis (72009990.49948 ETH)
- Block Rewards (28479138.0938 ETH)
- Uncle Rewards (2192510.75 ETH)



以太坊区块收入

- 普通区块收入

- 固定奖励（挖矿奖励），每个普通区块都有
- 区块内包含的所有程序的 gas 花费的总和
- 如果普通区块引用了叔块，每引用一个叔块可以得到固定奖励的

1/32

- 叔块收入

叔块收入只有一项，就是叔块奖励，计算公式为：

叔块奖励 = (叔块高度 + 8 – 引用叔块的区块高度) * 普通区块奖励 / 8



“幽灵” (GHOST) 协议

- 以太坊出块时间：设计为12秒，实际14~15秒左右
- 快速确认会带来区块的高作废率，由此链的安全性也会降低
- “幽灵” 协议：Greedy Heaviest Observed SubTree, "GHOST"
 - 计算工作量证明时，不仅包括当前区块的祖区块，父区块，还要包括祖先块的作废的后代区块（“叔块”），将他们进行综合考虑。
 - 目前的协议要求下探到第七层（最早的简版设计是五层），也就是说，废区块只能以叔区块的身份被其父母的第二代至第七代后辈区块引用，而不能是更远关系的后辈区块。
 - 以太坊付给以“叔区块”身份为新块确认作出贡献的废区块 $\frac{7}{8}$ 的奖励，把它们纳入计算的“侄子区块”将获得区块奖励的 $\frac{1}{32}$ ，不过，交易费用不会奖励给叔区块。



以太坊和图灵完备

- 1936年，英国数学家艾伦·图灵（Alan Turing）创建了一个计算机的数学模型，它由一个控制器、一个读写头和一根无限长的工作带组成。纸带起着存储的作用，被分成一个个的小方格（可以看成磁带）；读写头能够读取纸带上的信息，以及将运算结果写进纸带；控制器则负责根据程序对搜集到的信息进行处理。在每个时刻，机器头都要从当前纸带上读入一个方格信息，然后结合自己的内部状态查找程序表，根据程序输出信息到纸带方格上，并转换自己的内部状态，然后进行移动纸带。
- 如果一个系统可以模拟任何图灵机，它就被定义为“图灵完备”（Turing Complete）的。这种系统称为通用图灵机（UTM）。
- 以太坊能够在称为以太坊虚拟机的状态机中执行存储程序，同时向内存读取和写入数据，使其成为图灵完备系统，因此成为通用图灵机。考虑到有限存储器的限制，以太坊可以计算任何可由任何图灵机计算的算法。
- 简单来说，以太坊中支持循环语句，理论上可以运行“无限循环”的程序。



去中心化应用

- 基于以太坊可以创建**智能合约** (Smart Contract) 来构建

去中心化应用 (Decentralized Application, 简称为 DApp)

- 以太坊的构想是成为 DApps 编程开发的平台

- DApp至少由以下组成：

—— 区块链上的智能合约

—— Web前端用户界面

以太坊应用

- 基于以太坊创建新的加密货币 (CryptoCurrency, 这种能力是 2017 年各种 ICO 泛滥的技术动因)
- 基于以太坊创建域名注册系统、博彩系统
- 基于以太坊开发去中心化的游戏，比如 2017 年底红极一时的以太猫 (CryptoKitties, 最高单只猫售价高达 80W 美元)

代币 (Token)

- 代币 (token) 也称作通证，本意为“令牌”，代表有所有权的资产、货币、权限等在区块链上的抽象
- 可替代性通证 (fungible token)：指的是基于区块链技术发行的，互相可以替代的，可以接近无限拆分的token
- 非同质通证 (non-fungible token)：指的是基于区块链技术发行的，唯一的，不可替代的，大多数情况下不可拆分的token，如加密猫 (CryptoKitties)



名词解释

- **EIP**: Ethereum Improvement Proposals, 以太坊改进建议
- **ERC**: Ethereum Request for Comments的缩写, 以太坊征求意见。
一些EIP被标记为ERC, 表示试图定义以太坊使用的特定标准的提议
- **EOA**: External Owned Account, 外部账户。由以太坊网络的人类用户创建的账户
- **Ethash**: 以太坊1.0 的工作量证明算法。
- **HD钱包**: 使用分层确定性 (HD protocol) 密钥创建和转账协议 (BIP32) 的钱包。
- **Keccak256**: 以太坊中使用的密码哈希函数。Keccak256 被标准化为SHA-3
- **Nonce**: 在密码学中, 术语nonce用于指代只能使用一次的值。以太坊使用两种类型的随机数, 账户随机数和POW随机数



Q&A



尚硅谷



初识以太坊

——钱包、测试网络和简单交易

2018.10



以太币单位

- 以太坊的货币单位称为以太，也称为ETH或符号Ξ
- ether被细分为更小的单位，直到可能的最小单位，称为wei；
$$1 \text{ ether} = 10^{18} \text{ wei}$$
- 以太的值总是在以太坊内部表示为以wei表示的无符号整数值。
- 以太的各种单位都有一个使用国际单位制（SI）的科学名称，和一个口语名称。



以太币各单位名称

值 (wei)	指数	通用名称	SI 名称
1	1	wei	wei
1,000	10^3	babbage	kilowei or femtoether
1,000,000	10^6	lovelace	megawei or picoether
1,000,000,000	10^9	shannon	gigawei or nanoether
1,000,000,000,000	10^{12}	szabo	microether or micro
1,000,000,000,000,000	10^{15}	finney	milliether or milli
1,000,000,000,000,000,000	10^{18}	ether	ether
1,000,000,000,000,000,000,000	10^{21}	grand	kiloether
1,000,000,000,000,000,000,000,000	10^{24}		megaether

以太坊钱包

以太坊钱包是我们进入以太坊系统的门户。它包含了私钥，可以代表我们创建和广播交易。

- MetaMask：一个浏览器扩展钱包，可在浏览器中运行。
- Jaxx：一款多平台、多币种的钱包，可在各种操作系统上运行，包括Android, iOS, Windows, Mac和Linux。
- MyEtherWallet (MEW)：一个基于web的钱包，可以在任何浏览器中运行。
- Emerald Wallet：旨在与 ETC 配合使用，但与其他基于以太坊的区块链兼容。



私钥、公钥和地址

- 私钥 (Private Key)

以太坊私钥事实上只是一个256位的随机数，用于发送以太的交易中创建签名来证明自己对资金的所有权。

- 公钥 (Public Key)

公钥是由私钥通过椭圆曲线加密secp256k1算法单向生成的512位(64字节)数。

- 地址 (Address)

地址是由公钥的 Keccak-256 单向哈希，取最后20个字节 (160位) 派生出来的标识符。



安全须知

- keystore文件就是加密存储的私钥。所以当系统提示你选择密码时：将其设置为强密码，备份并不要共享。如果你没有密码管理器，请将其写下来并将其存放在带锁的抽屉或保险箱中。要访问账户，你必须同时有keystore文件和密码。
- 助记词可以导出私钥，所以可以认为助记词就是私钥。请使用笔和纸进行物理备份。不要把这个任务留给“以后”，你会忘记。
- 切勿以简单形式存储私钥，尤其是以电子方式存储。
- 不要将私钥资料存储在电子文档、数码照片、屏幕截图、在线驱动器、加密PDF等中。使用密码管理器或笔和纸。
- 在转移任何大额金额之前，首先要做一个小小的测试交易（例如，小于1美元）。收到测试交易后，再尝试从该钱包发送。



安装MetaMask

- 打开Google Chrome浏览器并导航至：
- <https://chrome.google.com/webstore/category/extensions>
- 搜索“MetaMask”并单击狐狸的徽标。您应该看到扩展程序的详细信息
页面如下：

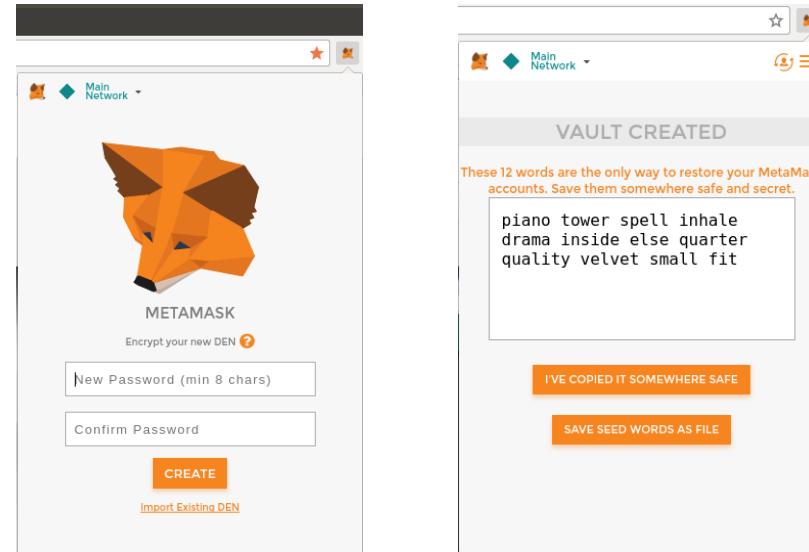


- 验证您是否正在下载真正的MetaMask扩展程序非常重要，因为有时候人们可以通过谷歌的过滤器隐藏恶意扩展。确认您正在查看正确的扩展程序后，请点击“添加到Chrome”进行安装。



第一次使用MetaMask

- 安装MetaMask后，应该在浏览器的工具栏中看到一个新图标（狐狸头）。点击它开始。系统会要求接受条款和条件，然后输入密码来创建新的以太坊钱包：



- 设置密码后，MetaMask将生成一个钱包，并显示由12个英文单词组成的助记符备份。如果MetaMask或计算机出现问题，导致无法打开钱包，我们可以在任何兼容的钱包中使用这些单词来恢复对资金的访问。

怎样安全存储助记词

- 将助记词（12个单词）备份在纸上，两次。
- 将两个纸张备份存放在两个单独的安全位置，例如防火保险箱，锁定抽屉或保险箱。
- 要将纸质备份视为自己在以太坊钱包中存储的等值现金。任何能够访问这些单词的人都可以访问并窃取你的资金。



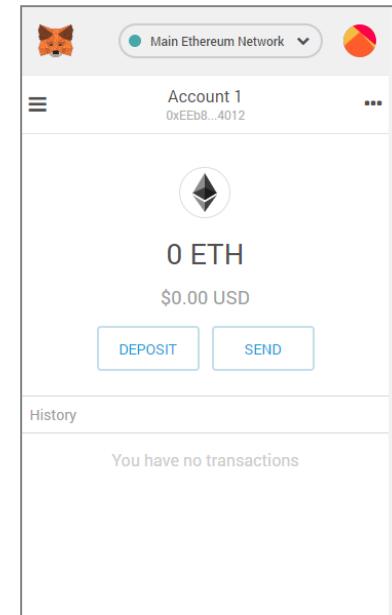
显示账户信息

- 一旦确认已安全存储助记符，MetaMask将显示您的以太坊帐户详细信息：

——账户名称：Account1

——以太坊地址

——账户余额：0 ETH





助记词

- 助记词是明文私钥的另一种表现形式，最早由BIP-39提出，目的是帮助用户记忆复杂的私钥（256位）。
- 技术上该提议可以在任意区块链中实现，比如使用完全相同的助记词在比特币和区块链上生成的地址可以是不同的，用户只需要记住满足一定规则的词组（就是上面说的助记词），钱包软件就可以基于该词组创建一些列的账户，并且保障不论是在什么硬件、什么时间创建出来的账户、公钥、私钥都完全相同，这样既解决了账号识别的问题，也把账户恢复的门槛降低了很多。
- 支持 BIP39 提议的钱包也可以归类为 HD 钱包（Hierarchical Deterministic Wallet），Metamask 当属此类。

切换网络

- **Main Network (Network ID: 1)**
- 主要的、公共的，以太坊区块链。真正的ETH，真正的价值，真正的结果。
- **Ropsten Test Network (Network ID: 3)**
- 以太坊公共测试区块链和网络，使用工作量证明共识（挖矿）。该网络上的ETH没有任何价值。
- **Kovan Test Network (Network ID: 42)**
- 以太坊公共测试区块链和网络，使用“Aura”协议进行权威证明POA共识（联合签名）。该网络上的ETH没有任何价值。此测试网络仅由Parity支持。
- **Rinkeby Test Network (Network ID: 4)**
- 以太坊公共测试区块链和网络，使用“Clique”协议进行权威证明POA共识（联合签名）。该网络上的ETH没有任何价值。
- **localhost 8545**
- 连接到与浏览器在同一台计算机上运行的节点。该节点可以是任何公共区块链（main或testnet）的一部分，也可以是私有testnet。
- **Custom RPC**

获取测试以太

- 钱包有了，地址有了，接下来需要做的就是为我们的钱包充值。
我们不会在主网络上这样做，因为真正的以太坊需要花钱。
- 以太坊测试网络给了我们免费获取测试以太的途径：水龙头
(faucet)
- 现在，我们将尝试把一些测试以太充入我们的钱包。



获取测试以太

- 将 MetaMask 切换到 Ropsten 测试网络。单击 “Deposit”；然后单击 “Ropsten Test Faucet” 。MetaMask 将打开一个新的网页：

faucet

address: 0x81b7e08f65bdf5648606c89998a9cc8164397647
balance: 357445.65 ether

request 1 ether from faucet

user

address: 0xe713963a92c02317a681b9bb3065a8249de124f
balance: 0.00 ether
donate to faucet:

1 ether 10 ether 100 ether

transactions

0xb53dcf15af8d86842c08e53474ed25beba17b122cccb7da77f559d3527c1b2f5

- 按绿色 “request 1 ether from faucet”按钮。您将在页面的下半部分看到一个交易ID。水龙头应用程序创建了一个交易 - 付款给您。交易ID如下所示：



在区块浏览器中查看

- <https://ropsten.etherscan.io/>

The screenshot shows a web browser displaying the Etherscan Testnet transaction details for the URL <https://ropsten.etherscan.io/tx/0x7c7ad5aaea6474adccf6f5c5d6abed11b70a3...>. The page header includes the Etherscan logo, navigation links for HOME, BLOCKCHAIN, ACCOUNT, TOKEN, CHART, and MISC, and a search bar. The main content area shows the transaction details:

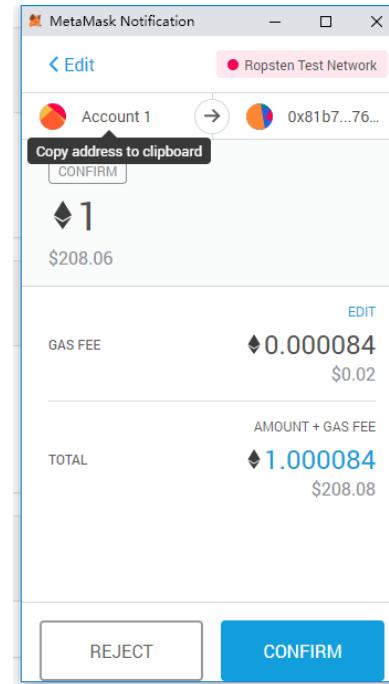
Transaction Information	
TxHash:	0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fb6f9590109e099568090c57
TxReceipt Status:	Success
Block Height:	2546420 (3 block confirmations)
TimeStamp:	1 min ago (Jan-29-2018 05:19:35 PM +UTC)
From:	0x81b7e08f65bdf5648606c89998a9cc8164397647
To:	0x9e713963a92c02317a681b9bb3065a8249de124f
Value:	1 Ether (\$0.00)



从MetaMask发送Ether

- 单击橙色“1 ether”按钮告诉MetaMask创建支付水龙头1 ether的交易。

MetaMask将准备一个交易并弹出一个确认窗口：





Gas编辑选项

- Metamask 计算了最近成功交易的平均 gas 价格为4 GWEI
- 发送基本交易的 gas 成本是21000个 gas单位
- 花费的最大 ETH 量是 $4 * 21000$
 $GWEI = 84000 \text{ GWEI} = 0.000084\text{ETH}$
- 做 1 ETH 交易成本为 1.000084 ETH
- 从水龙头请求多一些的以太，如果有2个ETH的余额，我们就可以再试一次

Customize Gas X

Gas Price (GWEI)

We calculate the suggested gas prices based on network success rates.

^
▼

Gas Limit

We calculate the suggested gas limit based on network success rates.

^
▼



搜索地址的交易记录

The screenshot shows the MetaMask wallet interface. On the left, the wallet dashboard for 'Account 1' (Address: 0xEEb8e564689Ddc6138263078bE00dec5E6AC4012) is displayed. The balance is 9.999958 Ether, and there are 14 transactions. A modal window titled 'Account Details' is open, featuring a 'View on Etherscan' button, which is highlighted with a red box. Below the modal, the account's value is listed as \$2,081.19 USD. At the bottom of the dashboard are 'DEPOSIT' and 'SEND' buttons. On the right, a detailed view of the transaction history is shown, listing 14 recent transactions with columns for TxHash, Block, Age, From, To, Value, and TxFee.

TxHash	Block	Age	From	To	Value	TxFee	
0xb53d15a8b868...	426698	1 hr 27 mins ago	0xeeb8e564689ddc...	OUT	0x81b7e08f65bd56...	1 Ether	0.000021
0xf6a3b052f4ec6f6...	4260513	1 day 35 mins ago	0x81b7e08f65bd56...	IN	0xeeb8e564689ddc...	1 Ether	0.000021
0xb3e475bb5fc58ac...	4260356	1 day 1 hr ago	0xeeb8e564689ddc...	OUT	0x81b7e08f65bd56...	1 Ether	0.000021
0x2d1dd242639c0a...	4260319	1 day 1 hr ago	0x81b7e08f65bd56...	IN	0xeeb8e564689ddc...	1 Ether	0.000021
0x23a7208effdc814...	4260315	1 day 1 hr ago	0x81b7e08f65bd56...	IN	0xeeb8e564689ddc...	1 Ether	0.000021



Q&A



尚硅谷

获取 Rinkeby 测试以太

Ropsten 是以太坊的主测试网，自然是测试练手的首选。不过由于 Ropsten 采用与主网完全一样的 PoW 共识，有时也就会和主网一样拥堵，甚至有过之而无不及。比如前些天 Ropsten 测试君士坦丁堡分叉的时候，一度完全瘫痪，无法发送交易。这时我们可能就需要找别的替代品了。Rinkeby 和 Kovan 采用的是 PoA 机制，所以出块很快而且很稳定。

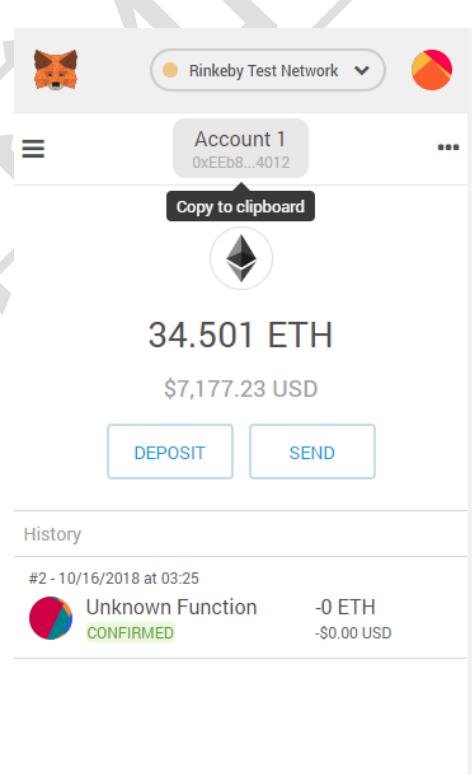
现在我们切换到 Rinkeby 测试网络，再来看一下如何获取免费的测试以太。

关于在 Rinkeby 的 faucet 上获取测试以太，社区已经提供了很不错的解决办法。目前发现的免费充值方法（当然是充值到测试网络中）主要有两个：

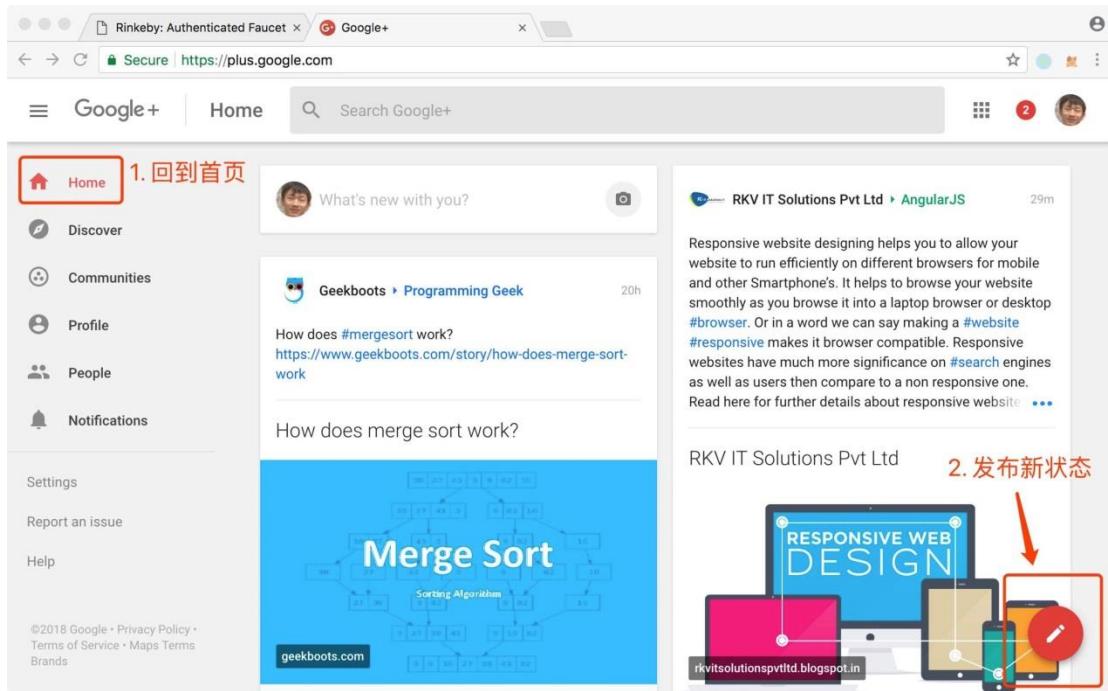
- rinkeby-faucet.com，只要提供账户地址即可充值 0.001 ETH，理论上是可以无限制充值的，但是如果需要充值 1ETH，需要操作 1000 次，太费劲，建议大家直接使用第 2 种方式；
- faucet.rinkeby.io，可以提供多达 18 ETH（/三天）的充值金额，但是为了避免被滥用，要求接受充值的账户持有人必须以太坊账户地址发送到自己的社交网络中（如 Twitter、Facebook、Google Plus），同样，该工具限制了充值的频率；

接下来，逐步跟大家介绍下，如何使用 faucet.rinkeby.io 为 Metamask 里面的账户充值 18 ETH。

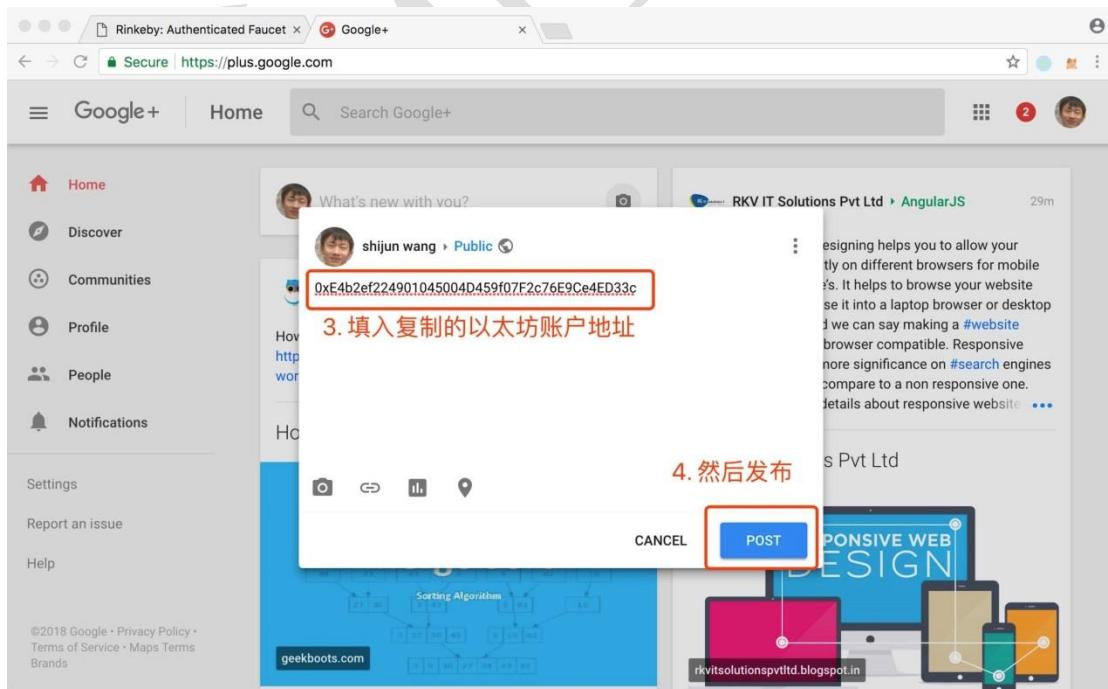
1. 复制 Metamask 账户的地址。点击小狐狸的图标打开钱包（必要的时候需要输入密码解锁钱包），然后点击 "Account 1" 把地址复制到剪贴板，如下图：



2. 打开 plus.google.com, 确保处于登录状态 (如果没有需要先注册 google 账号), 如下图, 按页面右下角的按钮, 准备开始发布新的状态:



3. 把复制到的 Metamask 账户地址粘贴到状态发布输入框里面, 然后点击发布:



4. 单击新发布状态卡片右上角的分享按钮, 会在新标签中打开该状态:



获取 Rinkeby 测试以太

The screenshot shows the Google+ Home page. On the left, there's a sidebar with links: Home, Discover, Communities, Profile, People, and Notifications. The main area shows a post from 'shijun wang' with the text: '5. 点击分享, 会打开新标签'. Below it is another post from 'Geekboots' with the text: 'How does #mergesort work?'. A red box highlights the share icon (a square with a plus sign) next to the first post.

5. 在新标签中复制地址栏中的地址备用:

The screenshot shows a new browser tab with the URL <https://plus.google.com/+shijunwang2010/posts/X5xZEhv7YH2>. A red box highlights the URL in the address bar. The page content is identical to the one in the previous screenshot, showing the same post from 'shijun wang'.

6. 打开 faucet.rinkeby.io, 按下图提示操作:



获取 Rinkeby 测试以太坊

The screenshot shows the Rinkeby Authenticated Faucet website. At the top, it says "Rinkeby Authenticated Faucet". Below that is a form with a URL input field containing "https://plus.google.com/+shijunwang2010/posts/X5xZEhv7YH2". To the right of the URL is a dropdown menu labeled "Give me Ether" with options: "3 Ethers / 8 hours", "7.5 Ethers / 1 day", and "18.75 Ethers / 3 days", with "18.75 Ethers / 3 days" highlighted with a red box. Below the URL input field, there's a note: "7. 填入 google plus 新状态的地址". On the left, there are icons for Twitter, Google+, and Facebook, each with instructions: "To request funds via Twitter, make a tweet with your Ethereum address pasted into the contents (surrounding text doesn't matter). Copy-paste the tweets URL into the above input box and fire away!" for Twitter; "To request funds via Google Plus, publish a new public post with your Ethereum address embedded into the content (surrounding text doesn't matter). Copy-paste the posts URL into the above input box and fire away!" for Google+; and "To request funds via Facebook, publish a new public post with your Ethereum address embedded into the content (surrounding text doesn't matter). Copy-paste the posts URL into the above input box and fire away!" for Facebook. At the bottom left, it says "You can track the current pending requests below the input field to see how much you have to wait until your turn comes." and "The faucet is running invisible reCaptcha protection against bots." There's also a "Privacy - Terms" link.

提交充值申请之后，可能会遇到 Google 的图形验证码，按提示操作即可，等待转账完成，可以看到如下的提示：

The screenshot shows the Rinkeby Authenticated Faucet website after a successful funding. It displays a message "充值成功的提示" (Funding successful) in red. The user's Ethereum address "0xe4b2ef224901045004d459f07f2c76e9ce4ed33c" is shown next to a funded status bar. Below the address, it says "3 peers 2209423 blocks 9.046256971665328e+56 Ethers 98981 funded". The rest of the page content is identical to the previous screenshot, including the social media instructions and the note about reCaptcha protection.

How does this work?

This Ether faucet is running on the Rinkeby network. To prevent malicious actors from exhausting all available funds or accumulating enough Ether to mount long running spam attacks, requests are tied to common 3rd party social network accounts. Anyone having a Twitter, Google+ or Facebook account may request funds within the permitted limits.

- To request funds via Twitter, make a [tweet](#) with your Ethereum address pasted into the contents (surrounding text doesn't matter).
Copy-paste the [tweets URL](#) into the above input box and fire away!
- To request funds via Google Plus, publish a new [public post](#) with your Ethereum address embedded into the content (surrounding text doesn't matter).
Copy-paste the [posts URL](#) into the above input box and fire away!
- To request funds via Facebook, publish a new [public post](#) with your Ethereum address embedded into the content (surrounding text doesn't matter).
Copy-paste the [posts URL](#) into the above input box and fire away!

You can track the current pending requests below the input field to see how much you have to wait until your turn comes.
The faucet is running invisible reCaptcha protection against bots.



重新打开我们的 Metamask 钱包账户，查看账户余额，发现还是 0。

可能你会好奇，刚才明明充值成功了，为什么账户余额还是 0 呢？原因是充值操作只发生在 Rinkeby 测试网络中，而 Metamask 钱包默认链接的是以太坊主网，还记得主网和测试网络的账号可以还记得主网和测试网络的账号可以通用，但是账户中的数据是完全隔离的么？点击 Metamask 钱包界面左上角的 "Main network"，切换到 Rinkeby 测试网络即可。



获取 Rinkeby 测试以太

不出意外，现在可以看到已经有余额了。恭喜，拿到了接近 4000 美金的测试金！





在 Remix 上构建简单的水龙头合约

我们已经创建了一个钱包，而且接收并发送了以太币。到目前为止，我们看到以太坊和比特币一样，也可以看作一种加密货币。但以太坊还有更多功能。事实上，加密货币功能服从于以太坊作为世界计算机的功能；去中心化的智能合约平台。**Ether** 用于支付运行智能合约的费用，智能合约是在称为以太坊虚拟机（**EVM**）的模拟计算机上运行的计算机程序。

EVM 是一个全局单例，意味着它就像是一个全局的单实例计算机一样运行，无处不在。

以太坊网络上的每个节点都运行 **EVM** 的本地副本以验证合约执行，而以太坊区块链在处理交易和智能合约时记录此世界计算机的变化状态。

以太坊有许多不同的高级语言，所有这些语言都可用于编写合约并生成 **EVM** 字节码。

到目前为止，一种高级语言是智能合约编程的主要语言：**Solidity**。**Solidity** 由 **Gavin Wood** 创建，并已成为以太坊及其他地区使用最广泛的语言。我们将使用 **Solidity** 编写我们的第一份合约。

编写水龙头合约

对于我们的第一个例子，我们将编写一个控制水龙头的合约。我们已经在 **Ropsten** 测试网络上使用了一个水龙头来测试 **ether**。水龙头是一件相对简单的事情：它会向任何要求的地址发出以太，并且可以定期重新填充。当然，我们可以将水龙头实施为由人（或 **Web** 服务器）控制的钱包，不过现在我们的目标是学习智能合约，所以我们将编写实施水龙头的 **Solidity** 合同：

Faucet.sol: 实施水龙头的 **Solidity** 合同

```
// Version of Solidity compiler this program was written for  
pragma solidity ^0.4.19;
```

¹

更多 **Java** -**大数据** -**前端** -**python** **人工智能** -**区块链** 资料下载，可访问百度：尚硅谷官网



```
// Our first contract is a faucet!
contract Faucet {

    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {

        // Limit withdrawal amount
        require(withdraw_amount <= 1000000000000000000);

        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }

    // Accept any incoming amount
    function () public payable {}

}
```

这是一个非常简单的合约，尽可能简单。它也是一个有缺陷的合同，表明了一些不良做法和安全漏洞。我们将通过检查后面部分中的所有缺陷来学习。但就目前而言，让我们一步一步地看看这份合约的作用及其运作方式。

第一行是注释：

```
// Version of Solidity compiler this program was written for
```

注释供人阅读，不包含在可执行 EVM 字节码中。我们通常在我们试图解释的代码之前将它们放在行上，或者有时在同一行上。注释以两个正斜杠//开头。从斜线到超出的所有内容，直到该行的结尾，都被视为空行并被忽略。

好的，下一行是我们实际合同开始的地方：

```
contract Faucet {
```

该行声明了一个合约对象，类似于其他面向对象语言（如 JavaScript, Java 或 C ++）中的类声明。合约定义包括定义范围的花括号{}之间的所有行，就像在许多其他编程语言中使用花括号一样。



在 Remix 上构建简单的水龙头合约

接下来，我们声明水龙头合约的第一个功能：

```
function withdraw(uint withdraw_amount) public {
```

该函数名为 `withdraw`，它接受一个名为 `withdraw_amount` 的无符号整数（`uint`）参数。它被声明为公共函数，这意味着它可以被其他合约调用。函数定义遵循花括号：

```
    require(withdraw_amount <= 10000000000000000000);
```

提现功能的第一部分设定了提款限额。它使用内置的 Solidity 函数 `require` 来测试一个前提条件，即 `withdraw_amount` 小于或等于 `10000000000000000000 wei`，这是 `ether` 的基本单位，相当于 `0.1 ether`。如果使用大于该数量的 `withdraw_amount` 调用 `withdraw` 函数，则此处的 `require` 函数将导致合约执行停止并因异常而失败。

这部分合约是我们水龙头的主要逻辑。它通过限制提款来控制合约之外的资金流动。这是一个非常简单的控制，但可以让你一瞥可编程区块链的力量：控制资金的去中心化软件。

接下来是实际提现：

```
    msg.sender.transfer (withdraw_amount);
```

这里有一些神奇的东西：`msg` 对象，这是所有合约都可以访问的输入之一。它表示触发此合约执行的交易。属性 `sender` 是交易的发件人地址。函数传递是一个内置函数，它将以太从合约传递到调用它的地址。向后读，这意味着转移到触发此合约执行的 `msg` 的发送者。传递函数将金额作为其唯一参数。我们将 `withdraw_amount` 值作为参数传递给上面几行声明的 `withdraw` 函数。

下一行是结束大括号，表示我们的 `withdraw` 函数定义的结束。

下面我们再声明一个功能：

```
function () public payable {}
```



此函数是所谓的“回退”或默认函数，如果触发合约的交易未命名合约中的任何已声明函数或任何函数或未包含数据，则调用此函数。合约可以有一个这样的默认函数（没有名称），它通常是接收以太的函数。这就是为什么它被定义为公共和默认函数，这意味着它可以接受以太合约。除了接受以太之外，它没有做任何事情，如花括号{}中的空定义所示。如果我们创建一个将 `ether` 发送到合约地址的交易，就好像它是钱包一样，这个函数将处理它。

在我们的默认函数下面是最后的结束花括号，它表示了合约 `Faucet` 的定义结束。

编译水龙头合约

现在我们有了第一个示例合约，我们需要使用 `Solidity` 编译器将 `Solidity` 代码转换为 `EVM` 字节码，因此它可以由 `EVM` 执行。

`Solidity` 编译器作为独立的可执行文件，作为不同框架的一部分，也捆绑在集成开发环境（`IDE`）中。为了简单起见，我们将使用一种比较流行的 `IDE`，称为 `Remix`。

`Remix` 是以太坊社区开发并开源的、一款非常好用的在线 `Solidity` 集成开发环境，我们可以方便的在其中编写、部署、测试智能合约，`Remix` 提供了强大的自动完成，语法高亮，实时编译检查错误等。

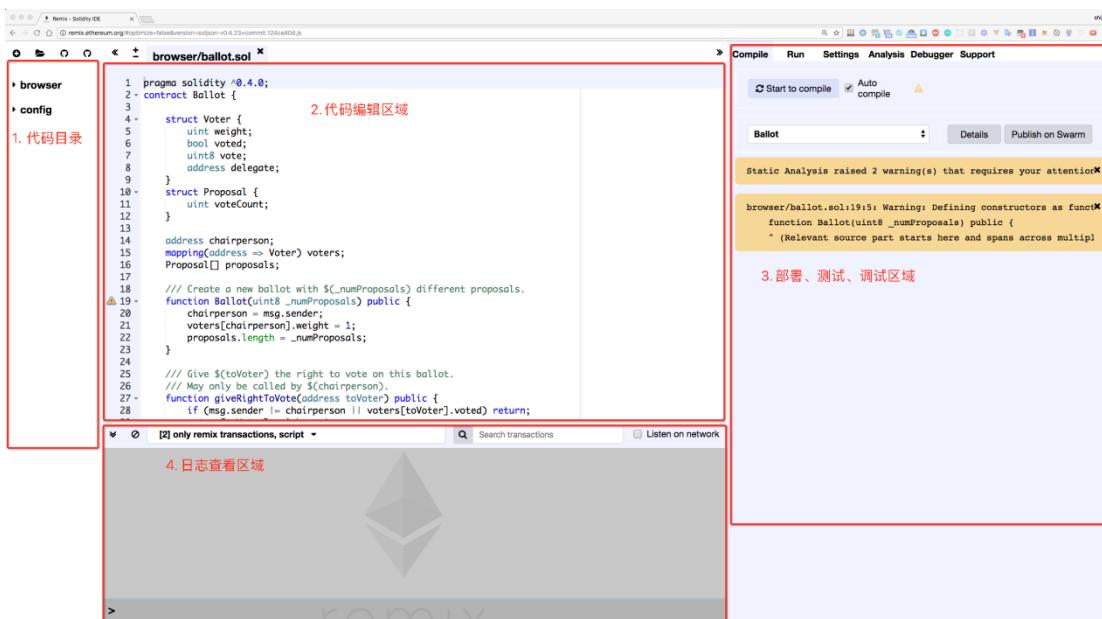
使用 `Chrome` 浏览器导航到 `Remix IDE`:

<https://remix.ethereum.org/>

接下来，我们使用 `Remix` 开发和调试第一个智能合约，初始界面如下图：



在 Remix 上构建简单的水龙头合约

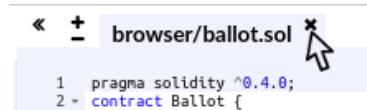


默认的界面可以划分为 4 个区域：

- 文件目录：这里有新建文件、从本地打开文件等按钮，可以直观的看到当前编辑过哪些合约文件，以及删除、重命名这些文件；
- 代码编辑：是使用频繁的区域，提供了语法高亮、自动完成等功能，左上角的加号和减号可以调整编辑器字体，即使刷新浏览器这些代码也不会丢失，因为 Remix 把配置和源代码都保存在了浏览器的 LocalStorage 里面；
- 调试工具：是使用频繁的区域，提供了智能合约的编译、运行、单步调试、编译器选项配置等功能；
- 调试输出：会打印出所有测试活动产生的日志，比如部署智能合约，调用智能合约函数过程中的各种交易； 除了代码区域之外的 3 个区域都是可以折叠起来的，点击要折叠的区域和代码区域相邻边上的双尖括号即可实现折叠。

第一次加载 Remix 时，它将以一个名为 ballot.sol 的示例合约开始。我们不需要它，

所以让我们关闭它，点击选项卡一角的 x：





在 Remix 上构建简单的水龙头合约

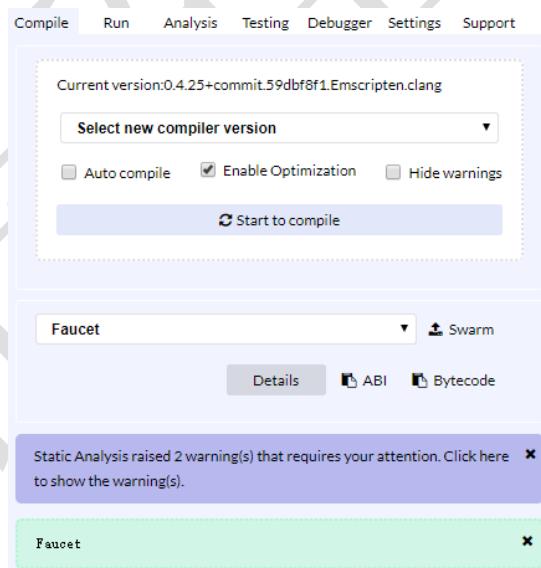
现在，通过单击左侧工具栏中的圆形加号，添加一个新选项卡，命名新文件 `Faucet.sol`:



打开新选项卡后，复制并粘贴我们的示例 `Faucet.sol` 中的代码:

```
// Version of Solidity compiler this program was written for
pragma solidity ^0.4.19;
// Our first contract is a faucet!
contract Faucet {
    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {
```

现在我们已经将 `Faucet.sol` 合约加载到 Remix IDE 中，IDE 将自动编译代码。如果一切顺利，你会在右侧的 **Compile** 选项卡下看到一个带有 “Faucet” 的绿色框，确认编译成功:



如果出现问题，最可能的问题是 Remix IDE 使用的是与 0.4.19 不同的 Solidity 编译器版本。在这种情况下，我们的 `pragma` 指令将阻止 `Faucet.sol` 编译。要更改编译器版本，点击“设置”选项卡，将编译器版本设置为 0.4.19，然后重试。



在 Remix 上构建简单的水龙头合约

Solidity 编译器现在已将 `Faucet.sol` 编译为 EVM 字节码。如果你很好奇，字节码看起来像这样：

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH2 0xF JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST
```

看到这么反人类的字节码，是不是很高兴我们可以使用 Solidity 等高级语言而不是直接在 EVM 字节码中编程？

在区块链上创建合同

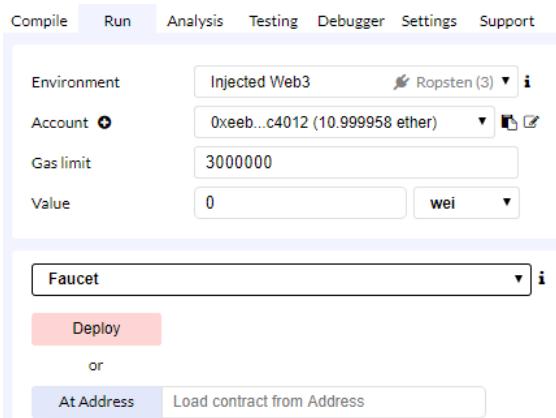
所以我们写了合约。我们把它编译成字节码。现在，我们需要在以太坊区块链上“注册”合约。我们将使用 Ropsten 测试网来测试我们的合约，这就是我们想要记录的区块链。

在区块链上注册合约涉及创建一个特殊交易，其目的地是一个“零地址”，也就是地址为：`0x00`。零地址是一个特殊地址，告诉以太坊区块链我们想要注册合约。不过我们不需要手动输入这么多 0，Remix IDE 将为我们处理所有这些并将交易发送到 MetaMask。

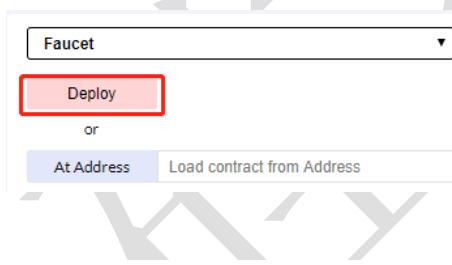
首先，切换到“Run”选项卡，然后在“Environment”下拉选择框中选择“Injected Web3”。这将 Remix IDE 连接到 MetaMask 钱包，并通过 MetaMask 连接到 Ropsten 测试网络。一旦你这样做，你可以在环境下看到“Ropsten”。此外，在帐户选择框中，它显示你的钱包的地址：



在 Remix 上构建简单的水龙头合约



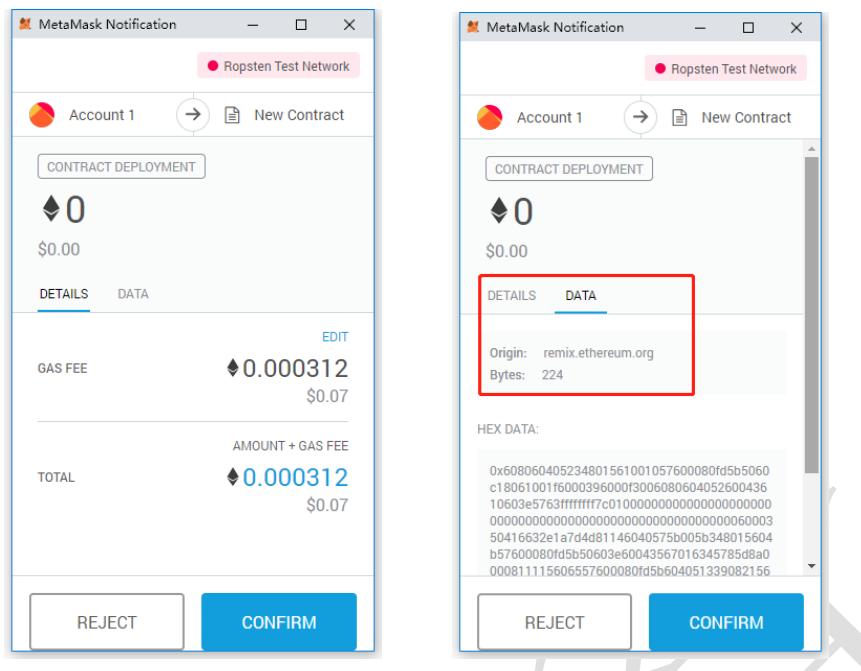
在我们刚刚确认的“Run”设置的正下方，是水龙头合约，准备好了。单击“Deploy”按钮：



Remix IDE 将构建特殊的“Create”交易，MetaMask 将要我们批准它。正如从 MetaMask 中看到的那样，合约创建交易中发送 0 个以太，但它有 224 个字节（已编译的合约）并将消耗余额 0.000312 以太的 gas 费用。点击“Confirm”批准它：

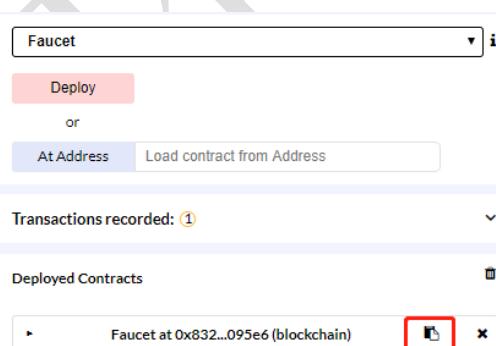


在 Remix 上构建简单的水龙头合约



现在，稍微等一下：在 Ropsten 上部署合约大约需要 15 到 30 秒。还记得 Ropsten 采用什么共识机制吗？它和主网完全一样采用了 PoW 的机制，出块的时间大概是 15 秒。

创建合约后，它将显示在“Run”选项卡的底部：



请注意，水龙头合约现在有一个自己的地址：Remix 将其显示为 Faucet，位于 0x832 ... 095e6。右侧的小剪贴板符号允许你将合约地址复制到剪贴板中。我们将在下一节中使用它。

另外如果细心我们可以在控制台 log 中看到，交易成功发送，我们可以用这里的链接到



etherscan 上查询：

```
• Running JavaScript scripts. The following libraries are accessible:  
  • web3 version 1.0.0  
  • ethers.js  
  • swarm.js  
• Executing common command to interact with the Remix interface (see list of commands above). Note that these commands can also be included and run on a JavaScript script.  
  
creation of Faucet pending...  
  
https://ropsten.etherscan.io/tx/0xddcd2d6d20fcd765e67923507595a4462be05b5118b263efb0b7406345406a7  
[block:4271619 txIndex:3] from:0xeeb...c4012 to:Faucet.(constructor) value:0 wei data:0x608...60029 logs:0 hash:0xddc...406a7  
Debug  
> |
```

与合约交互

让我们回顾一下迄今为止我们学到的东西：以太坊合约是控制资金的程序，它在称为 EVM 的虚拟机内运行。它们由特殊交易创建，该交易提交其字节码以记录在区块链上。一旦他们在区块链上创建，他们就有了一个以太坊地址，就像钱包一样。只要有人将某个交易发送到合约地址，就会导致合约在 EVM 中运行，并将该合约作为其输入。

发送到合约地址的交易可能包含 ether 或数据或两者。如果它们含有 ether，则将其“存入”合约余额。如果它们包含数据，则数据可以在合约中指定命名函数并调用它，将参数传递给函数。

在区块浏览器中查看合同地址

现在，我们已经在 Ropsten 区块链上记录了一份合约，我们可以看到它有一个以太坊地址。让我们在 ropsten.etherscan.io 区块浏览器上查看它，看看合约是什么样的。

在一个标签中打开 Remix，稍后我们会再次回顾它。现在，将浏览器导航到 ropsten.etherscan.io 并将地址粘贴到搜索框中。你应该看到合约的以太坊地址历史：



在 Remix 上构建简单的水龙头合约

The screenshot shows the Etherscan interface for the contract 0x83220cd52F67C40edc15D1B4e8CF20fd8ea095e6. The 'Contract Overview' section displays a balance of 0 Ether and one transaction. The 'Transactions' tab shows a single entry: a 'Contract Creation' transaction from 0xeeb8e564689ddc... at block 4271619, timestamped 30 mins ago, with a value of 0 Ether and a gas fee of 0.000312135. The 'Events' tab is empty.

资助合约

目前，合约在其历史记录中只有一个交易：合约创建交易。我们刚才在 etherscan 上也看到了，合约也还没有以太（零余额）。那是因为我们没有在创建交易中向合约发送任何以太，当然这本来是可以的。

那就让我们现在给合约发一些以太。你仍然应该在剪贴板中包含合约的地址（如果没有，从 Remix 再次复制）。打开 MetaMask，并向其发送 1 个以太，就像你发送给其他任何以太坊地址一样：

The image shows two overlapping MetaMask dialogs. The left dialog is titled 'Send ETH' and shows the transaction details: From: Account 1 (10.999583 ETH, \$2,308.15 USD), To: 0x83220cd52F67C40edc15D1B4e8CF20fd8ea095e6, Amount: 1 ETH (\$209.84 USD), Gas Fee: 0.000095 ETH (\$0.02 USD). The right dialog is titled 'Edit' and shows the total amount: TOTAL 1.000095 ETH (\$209.86). Both dialogs have 'CONFIRM' buttons at the bottom.

在一分钟内，如果你重新加载 etherscan 区块资源管理器，它将显示合约地址的另一个交易以及 1 个以太网的更新余额。

还记得我们的 `Faucet.sol` 代码中未命名的默认公共应付款功能吗？它看起来像这样：



在 Remix 上构建简单的水龙头合约

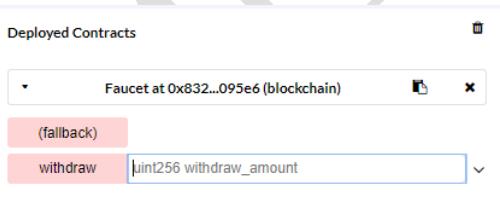
```
function () public payable {}
```

当你将交易发送到合同地址时，没有数据指定要调用的函数，它会调用此默认函数。因为我们将其声明为应付款，所以它接受并将 1 以太币存入合约账户余额。你的交易导致合约在 EVM 中运行，更新其余额。我们已经顺利资助了我们的水龙头！

提现我们的合约

接下来，让我们从水龙头中提取一些资金。要提现，我们必须构造一个调用 `withdraw` 函数的交易，并将 `withdraw_amount` 参数传递给它。为了使事情变得简单，Remix 将为我们构建该交易，MetaMask 将提供它以供我们批准。

返回 Remix 选项卡，查看 “Run” 选项卡下的合约。你应该看到一个标有 “Withdraw”的红色框，其中包含一个标记为 `uint256 withdraw_amount` 的字段条目：



这是合约的 Remix 接口。它允许我们构造调用合约中定义的函数的交易。我们将输入 `withdraw_amount` 并单击 “Withdraw” 按钮以生成交易。

首先，让我们弄清一下 `withdraw_amount`。我们想尝试提现 0.1 以太，这是我们合约允许的最大金额。请记住，以太坊中的所有货币值都在内部以 `wei` 表示，而我们的提现功能期望 `withdraw_amount` 也以 `wei` 计价。我们想要的数量是 0.1 以太，这是

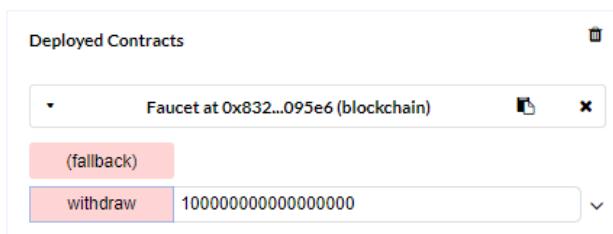


在 Remix 上构建简单的水龙头合约

1000000000000000000 wei (1 后跟 17 个零)。

由于 JavaScript 的限制, Remix 无法处理大到 10^{17} 的数字。相反, 我们将它括在双引号中, 以允许 Remix 将其作为字符串接收并将其作为 BigNumber 进行操作。如果我们不将它括在引号中, 则 Remix IDE 将无法处理它并显示“Error encoding arguments: Error: Assertion failed”, 好在 Remix 会帮我们做自动转换

在 withdraw_amount 框中键入 1000000000000000000, 然后单击 “Withdraw” 按钮:



MetaMask 将弹出一个交易窗口供你批准。点击 “Confirm” 将你的提款调用发送给合约。

等一下, 然后重新加载 etherscan 区块资源管理器, 以查看在水龙头合约地址历史记录中的交易:

Contract Overview														
Balance:	0.9 Ether	Misc:	Contract Creator: 0xeeb8e564689ddc... at txn 0xddcd2d6d20fcda7...	More Options										
Transactions:	4 txns													
Transactions														
Latest 4 txns														
TxHash	Block	Age	From	To	Value	[TxFee]								
0x8adb94e6551c18...	4271822	1 min ago	0xeeb8e564689ddc...	IN	0x83220cd52f67c40...	0 Ether 0.000088281								
0xd23688d42f6e4d0...	4271784	9 mins ago	0xeeb8e564689ddc...	IN	0x83220cd52f67c40...	1 Ether 0.00006312								
0xe9227fe9fb80dc...	4271774	11 mins ago	0xeeb8e564689ddc...	IN	0x83220cd52f67c40...	0 Ether 0.00006312								
0xddcd2d6d20fcda7...	4271619	49 mins ago	0xeeb8e564689ddc...	IN	Contract Creation	0 Ether 0.00031235								

我们现在看到一个新的交易, 其中合约地址为目的地, 零以太。合约余额已经改变, 现在是 0.9 以太, 因为它按要求向我们发送了 0.1 以太。但我们在合约地址历史记录中看不到



“OUT” 交易。

提现的地方在哪里？合约的地址历史记录页面中出现了一个新选项卡，名为 “Internal Transactions”。因为 0.1 以太传输源自合约代码，所以它是内部交易（也称为消息）。单击 “Internal Transactions” 选项卡以查看它：

The screenshot shows the Etherscan interface for a specific Ethereum contract. At the top, it displays the contract address: 0x83220cd52f67c40edc15d1b4e8cf20fd8ea095e6. Below this, there are sections for 'Contract Overview' (Balance: 0.9 Ether, Transactions: 4 txns), 'Misc:' (Contract Creator: 0xeeb8e564689dd..., at tx 0xddcd2d6d20fcda...), and 'More Options'. The 'Transactions' tab is active, but the 'Internal Txns' tab is highlighted with a red box. Under the 'Internal Txns' section, it says 'Internal Transactions as a result of Contract Execution' and 'Latest 1 Internal Transaction'. A table below lists the details of the latest internal transaction:

ParentTxHash	Block	Age	From	To	Value
0x8adb94e6551c18...	4271822	3 mins ago	0x83220cd52f67c40...	→ 0xeeb8e564689dd...	0.1 Ether

这个 “Internal Transactions” 是由合约在这行代码中发出的（来自 `Faucet.sol` 中的提现功能）：

```
msg.sender.transfer (withdraw_amount);
```

回顾一下：我们从 `MetaMask` 钱包发送了一个包含数据指令的交易，调用 `withdraw_amount` 参数为 0.1 ether 的 `withdraw` 函数；该交易导致合约在 EVM 内部运行，当 EVM 运行水龙头合约的提现功能时，首先它调用 `require` 函数并验证我们的金额小于或等于允许的最大提现 0.1 以太；然后它调用传递函数向我们发送以太，运行转账功能会产生一个内部交易，从合约的余额中将 0.1 以太币存入我们的钱包地址；这就是 etherscan 中 “内部交易” 选项卡中显示的那笔交易。



小结

在这次课程中，我们在 Solidity 写了一个水龙头合约，然后使用 Remix IDE 将合约编译为 EVM 字节码；我们使用 Remix 进行交易，并在 Ropsten 区块链上记录了水龙头合约。一旦记录下来，水龙头合约就有一个以太坊地址，我们给它发了一些 ether。最后，我们构建了一个交易来调用 withdraw 函数并成功请求 0.1 ether。合约检查了我们的请求，并通过内部交易向我们发送了 0.1 以太。

它可能看起来不多，但我们刚刚成功地与在分散的世界计算机上控制资金的软件进行交互。

我们将在之后的课程中进行更多智能合约编程，并了解最佳实践和安全注意事项。

以太坊客户端

2018.10



什么是以太坊客户端

- 以太坊客户端是一个软件应用程序，它实现以太坊规范并通过p2p网络与其他以太坊客户端进行通信。如果不同的以太坊客户端符合参考规范和标准化通信协议，则可以进行相互操作。
- 以太坊是一个开源项目，由“黄皮书”正式规范定义。除了各种以太坊改进提案之外，此正式规范还定义了以太坊客户端的标准行为。
- 因为以太坊有明确的正式规范，以太网客户端有了许多独立开发的软件实现，它们之间又可以彼此交互。



基于以太坊规范的网络

- 存在各种基于以太坊规范的网络，这些网络基本符合以太坊“黄皮书”中定义的形式规范，但它们之间可能相互也可能不相互操作。
- 这些基于以太坊的网络中有：以太坊，以太坊经典，Ella，Expanse，Ubiq，Musicoin等等。
- 虽然大多数在协议级别兼容，但这些网络通常具有特殊要求，以太坊客户端软件的维护人员、需要进行微小更改、以支持每个网络的功能或属性。



以太坊的多种客户端

- go-ethereum (Go)

官方推荐，开发使用最多

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

- parity (Rust)

最轻便客户端，在历次以太坊网络攻击中表现卓越

地址: <https://github.com/ethcore/parity/releases>

- cpp-ethereum (C++)

地址: <https://github.com/ethereum/cpp-ethereum>

- pyethapp (python)

地址: <https://github.com/heikoheiko/pyethapp>

- ethereumjs-lib (javascript)

地址: <https://github.com/ethereumjs/ethereumjs-lib>

- EthereumJ / Harmony (Java)

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



以太坊全节点

- 全节点是整个主链的一个副本，存储并维护链上的所有数据，并随时验证新区块的合法性。
- 区块链的健康和扩展弹性，取决于具有许多独立操作和地理上分散的全节点。每个全节点都可以帮助其他新节点获取区块数据，并提供所有交易和合约的独立验证。
- 运行全节点将耗费巨大的成本，包括硬件资源和带宽。
- 以太坊开发不需要在实时网络（主网）上运行的全节点。我们可以使用测试网络的节点来代替，也可以用本地私链，或者使用服务商提供的基于云的以太坊客户端；这些几乎都可以执行所有操作。



远程客户端和轻节点

- 远程客户端

不存储区块链的本地副本或验证块和交易。这些客户端一般只提供钱包的功能，可以创建和广播交易。远程客户端可用于连接到现有网络，MetaMask 就是一个这样的客户端。

- 轻节点

不保存链上的区块历史数据，只保存区块链当前的状态。轻节点可以对块和交易进行验证。



全节点的优缺点

优点

- 为以太坊网络的灵活性和抗审查性提供有力支持。
- 权威地验证所有交易。
- 可以直接与公共区块链上的任何合约交互。
- 可以离线查询区块链状态（帐户，合约等）。
- 可以直接把自己的合约部署到公共区块链中。

缺点

- 需要巨大的硬件和带宽资源，而且会不断增长。
- 第一次下载往往需要几天才能完全同步。
- 必须及时维护、升级并保持在线状态以同步区块。



公共测试网络节点的优缺点

优点

- 一个 testnet 节点需要同步和存储更少的数据，大约10GB，具体取决于不同的网络。
- 一个 testnet 节点一般可以在几个小时内完全同步。
- 部署合约或进行交易只需要发送测试以太，可以从“水龙头”免费获得。
- 测试网络是公共区块链，有许多其他用户和合约运行（区别于私链）。

缺点

- 测试网络上使用测试以太，它没有价值。因此，无法测试交易对手的安全性，因为没有任何利害关系。
- 测试网络上的测试无法涵盖所有的真实主网特性。例如，交易费用虽然是发送交易所必需的，但由于gas免费，因此 testnet 上往往不会考
虑。而且，一般来说，测试网络不会像主网那样仔细地操作。



本地私链的优缺点

优点

- 磁盘上几乎没有数据，也不同步别的数据，是一个完全“干净”的环境。
- 无需获取测试以太，你可以任意分配以太，也可以随时自己挖矿获得。
- 没有其他用户，也没有其他合约，没有任何外部干扰。

缺点

- 没有其他用户意味着与公链的行为不同。发送的交易并不存在空间或交易顺序的竞争。
- 除自己之外没有矿工意味着挖矿更容易预测，因此无法测试公链上发生的某些情况。
- 没有其他合约，意味着你必须部署要测试的所有内容，包括所有的

学习不漫无目的
技术

运行全节点的要求

- **最低要求**
 - 双核以上CPU
 - 硬盘存储可用空间至少80GB
 - 如果是SSD，需要4GB 以上 RAM，如果是HDD，至少8GB RAM
 - 8 MB/s下载带宽
- **推荐配置**
 - 四核以上的快速CPU
 - 16GB 以上 RAM
 - 500GB 以上可用空间的快速SSD
 - 25+ MB/s下载带宽



Geth (Go-Ethereum)

- Geth是由以太坊基金会积极开发的 Go 语言实现，因此被认为是以太坊客户端的“官方”实现。
- 通常，每个基于以太坊的区块链都有自己的Geth实现。
- 以太坊的 Geth github 仓库链接：

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



JSON-RPC

- 以太坊客户端提供了 API 和一组远程调用 (RPC) 命令，这些命令被编码为 JSON。这被称为 JSON-RPC API。本质上，JSON-RPC API 就是一个接口，允许我们编写的程序使用以太坊客户端作为网关，访问以太坊网络和链上数据。
- 通常，RPC 接口作为一个 HTTP 服务，端口设定为 8545。出于安全原因，默认情况下，它仅限于接受来自 localhost 的连接。
- 要访问JSON-RPC API，我们可以使用编程语言编写的专用库，例如JavaScript的 web3.js。
- 或者也可以手动构建HTTP请求并发送/接收JSON编码的请求，如：

```
$ curl -X POST -H "Content-Type: application/json" --data \
'{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":1
}' \ http://localhost:8545
```



Q&A



尚硅谷



用 Geth 搭建以太坊私链

这节课让我们来用 Geth 来搭建一个属于自己的以太坊私链。

安装 Geth

安装 Geth 有很多种方式，这里主要就 Linux 环境给出两种：系统包管理器（apt-get）安装和源码安装。更加推荐大家用源码安装，在整个过程中可以看到 Geth 各组件的构建步骤。

一、apt-get

```
$ sudo apt-get install software-properties-common  
$ sudo add-apt-repository -y ppa:ethereum/ethereum  
$ sudo apt-get update  
$ sudo apt-get install ethereum
```

二、源码安装

1. 克隆 github 仓库

我们的第一步是克隆 git 仓库，以获取源代码的副本。

```
$ git clone https://github.com/ethereum/go-ethereum.git
```

2. 从源码构建 Geth

要构建 Geth，切换到下载源代码的目录并使用 make 命令：

```
$ cd go-ethereum  
$ make geth
```

如果一切顺利，我们将看到 Go 编译器构建每个组件，直到它生成 geth 可执行文件：

```
build/env.sh go run build/ci.go install ./cmd/geth  
>>> /usr/local/go/bin/go install -ldflags -X  
main.gitCommit=58a1e13e6dd7f52a1d5e67bee47d23fd6cfdee5c -v ./cmd/geth  
github.com/ethereum/go-ethereum/common/hexutil  
github.com/ethereum/go-ethereum/common/math  
github.com/ethereum/go-ethereum/crypto/sha3 github.com/ethereum/go-ethereum/rlp  
github.com/ethereum/go-ethereum/crypto/secp256k1  
github.com/ethereum/go-ethereum/common [...]  
github.com/ethereum/go-ethereum/cmd/utils
```

¹

更多 Java -大数据 -前端 -python 人工智能 -区块链资料下载，可访问百度：尚硅谷官网



用 Geth 搭建以太坊私链

```
github.com/ethereum/go-ethereum/cmd/geth Done building. Run "build/bin/geth" to  
launch geth.
```

查看 geth version，确保在真正运行之前安装正常：

```
$ ./build/bin/geth version  
Geth  
Version: 1.8.0-unstable  
Git Commit: e37f7be97e47a032d723db16d8b195998547805a  
Architecture: amd64  
Protocol Versions: [63 62]  
Network Id: 1  
Go Version: go1.9  
Operating System: linux  
GOPATH=/home/ubuntu/project  
GOROOT=/usr/local/go
```

启动节点同步

安装好了 Geth，现在我们可以尝试运行一下它。执行下面的命令，geth 就会开始同步区块，并存储在当前目录下。这里的 --syncmode fast 参数表示我们会以“快速”模式同步区块。在这种模式下，我们只会下载每个区块头和区块体，但不会执行验证所有的交易，直到所有区块同步完毕再去获取一个系统当前的状态。这样就节省了很多交易验证的时间。

```
$ geth --datadir . --syncmode fast
```

通常，在同步以太坊区块链时，客户端会一开始就下载并验证每个块和每个交易，也就是说从创世区块开始。毫无疑问，如果我们不加 --syncmode fast 参数，同步将花费很长时间并且具有很高的资源要求（它将需要更多的 RAM，如果你没有快速存储，则需要很长时间）。

有些文章会把这个参数写成 --fast，这是以前快速同步模式的参数写法，现在已经被 --syncmode fast 取代。

如果我们想同步测试网络的区块，可以用下面的命令：

```
$ geth --testnet --datadir . --syncmode fast
```

--testnet 这个参数会告诉 geth 启动并连接到最新的测试网络，也就是 Ropsten。测试网络的区块和交易数量会明显少于主网，所以会更快一点。但即使是用快速模式同步测试网络，也会需要几个小时的时间。



搭建自己的私有链

因为公共网络的区块数量太多，同步耗时太长，我们为了方便快速了解 Geth，可以试着用它来搭一个只属于自己的私链。

首先，我们需要创建网络的“创世”(genesis)状态，这写在一个小小的 JSON 文件里（例如，我们将其命名为 genesis.json）：

```
{  
  "config": {  
    "chainId": 15  
  },  
  "difficulty": "2000",  
  "gasLimit": "2100000",  
  "alloc": {  
    "7df9a875a174b3bc565e6424a0050ebc1b2d1d82": { "balance": "300000" },  
    "f41c74c9ae680c1aa78f42e5647a62f353b7bdde": { "balance": "400000" }  
  }  
}
```

要创建一条以它作为创世块的区块链，我们可以使用下面的命令：

```
geth --datadir path/to/custom/data/folder init genesis.json
```

在当前目录下运行 geth，就会启动这条私链，注意要将 networked 设置为与创世块配置里的 chainId 一致。

```
geth --datadir path/to/custom/data/folder --networkid 15
```

我们可以看到节点正常启动：

```
WARN [10-23|02:38:19] No etherbase set and no accounts found as default  
INFO [10-23|02:38:19] Starting peer-to-peer node  
instance=Geth/v1.8.0-unstable-e37f7be9/linux-amd64/go1.9  
...  
INFO [10-23|02:38:21] IPC endpoint opened:  
/home/ubuntu/project/go_ethereum_test/geth.ipc  
INFO [10-23|02:38:21] Mapped network port proto=tcp  
extport=30303 intport=30303 interface="UPNP IGDv1-IP1"
```

恭喜！我们已经成功启动了一条自己的私链。



用 Geth 搭建以太坊私链



Geth 控制台命令

Geth Console 是一个交互式的 JavaScript 执行环境，里面内置了一些用来操作以太坊的 JavaScript 对象，我们可以直接调用这些对象来获取区块链上的相关信息。这些对象主要包括：

eth：主要包含对区块链进行访问和交互相关的方法；

net：主要包含查看 p2p 网络状态的方法；

admin：主要包含与管理节点相关的方法；

miner：主要包含挖矿相关的一些方法；

personal：包含账户管理的方法；

txpool：包含查看交易内存池的方法；

web3：包含以上所有对象，还包含一些通用方法。

常用命令有：

personal.newAccount()：创建账户；

personal.unlockAccount()：解锁账户；

eth.accounts：列出系统中的账户；

eth.getBalance()：查看账户余额，返回值的单位是 Wei；

eth.blockNumber：列出当前区块高度；

eth.getTransaction()：获取交易信息；

eth.getBlock()：获取区块信息；

miner.start()：开始挖矿；

miner.stop()：停止挖矿；

web3.fromWei()：Wei 换算成以太币；

web3.toWei()：以太币换算成 Wei；

txpool.status：交易池中的状态；





以太坊账户

Ethereum Accounts

2018.10



从UTXO谈起

- 比特币在基于UTXO的结构中存储有关用户余额的数据：系统的整个状态就是一组UTXO的集合，每个UTXO都有一个所有者和一个面值（就像不同的硬币），而交易会花费若干个输入的UTXO，并根据规则创建若干个新的UTXO：
- 每个引用的输入必须有效且尚未花费；对于一个交易，必须包含有与每个输入的所有者匹配的签名；总输入必须大于等于总输出值
- 所以，系统中用户的余额（balance）是用户具有私钥的UTXO的总值



以太坊的做法

- 以太坊的“状态”，就是系统中所有帐户的列表
- 每个账户都包括了一个余额（balance），和以太坊特殊定义的数据（代码和内部存储）
- 如果发送帐户有足够的余额来支付，则交易有效；在这种情况下发送帐户先扣款，而收款帐户将记入这笔收入
- 如果接收帐户有相关代码，则代码会自动运行，并且它的内部存储也可能被更改，或者代码还可能向其他帐户发送额外的消息，这就会导致进一步的借贷资金关系



优缺点比较

比特币 UTXO 模式优点：

- 更高程度的隐私：如果用户为他们收到的每笔交易使用新地址，那么通常很难将帐户相互链接。这很大程度上适用于货币，但不太适用于任意dapps，因为dapps通常涉及跟踪和用户绑定的复杂状态，可能不存在像货币那样简单的用户状态划分方案。
- 潜在的可扩展性：UTXO在理论上更符合可扩展性要求。因为我们只需要依赖拥有 UTXO 的那些人去维护基于Merkle树的所有权证明就够了，即使包括所有者在内的每个人都决定忘记该数据，那么也只有所有者受到对应UTXO的损失，不影响接下来的交易。而在帐户模式中，如果每个人都丢失了与帐户相对应的Merkle树的部分，那将会使得和该帐户有关的消息完全无法处理，包括发币给它。



优缺点比较 (续)

以太坊账户模式优点：

- 可以节省大量空间：不将 UTXOs 分开存储，而是合为一个账户；每个交易只需要一个输入、一个签名并产生一个输出。
- 更好的可替代性：货币本质上都是同质化、可替代的；UTXO的设计使得货币从来源分成了“可花费”和“不可花费”两类，这在实际应用中很难有对应的模型。
- 更加简单：更容易编码和理解，特别是设计复杂脚本的时候。UTXO 在脚本逻辑复杂时更令人费解。
- 便于维护持久轻节点：只要沿着特定方向扫描状态树，轻节点可以很容易地随时访问账户相关的所有数据。而UTXO的每个交易都会使得状态引用发生改变，这对轻节点来说长时间运行Dapp会有很大压力。



比特币和以太坊的对比

	BitCoin	Ethereum
设计定位	现金系统	去中心化应用平台
数据组成	交易列表（账本）	交易和账户状态
交易对象	UTXO	Accounts
代码控制	脚本	智能合约

以太坊账户类型

- 外部账户 (Externally owned account, EOA)
- 合约账户 (Contract accounts)



EOA

外部账户（用户账户/普通账户）

- 有对应的以太币余额
- 可发送交易（转币或触发合约代码）
- 由用户私钥控制
- 没有关联代码



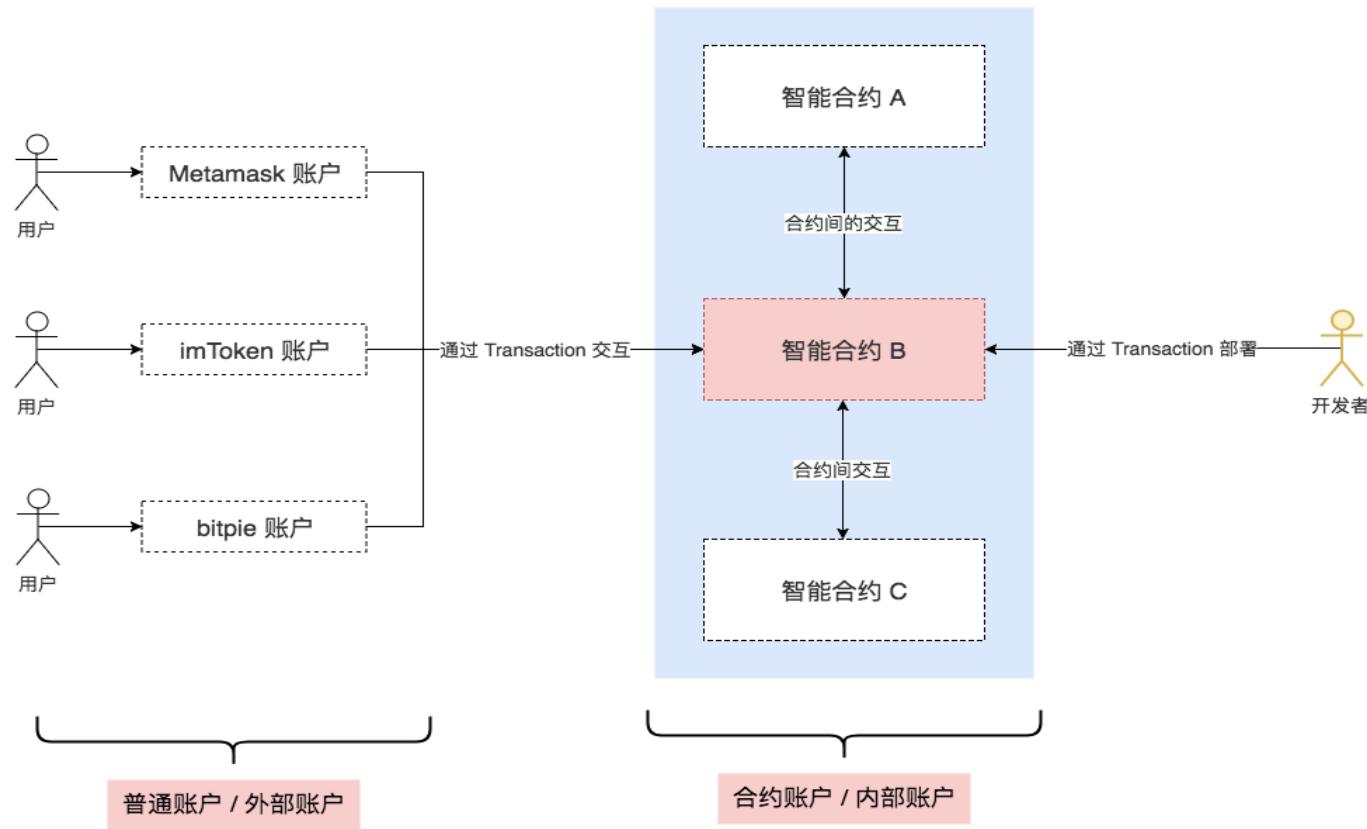
合约账户

外部账户 (用户账户/普通账户)

- 有对应的以太币余额
- 有关联代码
- 由代码控制
- 可通过交易或来自其它合约的调用消息来触发代码执行
- 执行代码时可以操作自己的存储空间，也可以调用其它合约



以太坊网络: Rinkeby



以太坊交易 (Transaction)

签名的数据包，由EOA发送到另一个账户

- 消息的接收方地址
- 发送方签名
- 金额 (VALUE)
- 数据 (DATA, 可选)
- START GAS
- GAS PRICE



消息 (Message)

- 合约可以向其它合约发送“消息”
- 消息是不会被序列化的虚拟对象，只存在于以太坊执行环境 (EVM) 中
- 可以看作函数调用
 - 消息发送方
 - 消息接收方
 - 金额 (VALUE)
 - 数据 (DATA, 可选)
 - START GAS



合约 (Contract)

- 可以读/写自己的内部存储 (32字节key-value的数据库)
- 可向其他合约发送消息，依次触发执行
- 一旦合约运行结束，并且由它发送的消息触发的所有子执行 (sub-execution) 结束，EVM就会中止运行，直到下次交易被唤醒



合约应用一

- 维护一个数据存储（账本），存放对其他合约或外部世界有用的内容
- 最典型的例子是模拟货币的合约（代币）



合约应用二

- 通过合约实现一种具有更复杂的访问策略的普通账户（EOA），这被称为“转发合同”：只有在满足某些条件时才会将传入的消息重新发送到某个所需的目的地址；例如，一个人可以拥有一份转发合约，该合约会等待直到给定三个私钥中的两个确认之后，再重新发送特定消息
- 钱包合约是这类应用中很好的例子

合约应用三

- 管理多个用户之间的持续合同或关系
- 这方面的例子包括金融合同，以及某些特定的托管合同或某种保险



Q&A



尚硅谷



以太坊交易详解 Ethereum Transactions

2018.10



交易的本质

- 交易是由外部拥有的账户发起的签名消息，由以太坊网络传输，并被序列化后记录在以太坊区块链上。
- 交易是唯一可以触发状态更改或导致合约在EVM中执行的事物。
- 以太坊是一个全局单例状态机，交易是唯一可以改变其状态的东西。
- 合约不是自己运行的，以太坊也不会“在后台”运行。以太坊上的一切变化都始于交易。



交易数据结构

交易是包含以下数据的序列化二进制消息：

- **nonce**: 由发起人EOA发出的序列号，用于防止交易消息重播。
- **gas price**: 交易发起人愿意支付的gas单价（wei）。
- **start gas**: 交易发起人愿意支付的最大gas量。
- **to**: 目的以太坊地址。
- **value**: 要发送到目的地的以太数量。
- **data**: 可变长度二进制数据负载（payload）。
- **v,r,s**: 发起人EOA的ECDSA签名的三个组成部分。
- 交易消息的结构使用递归长度前缀（RLP）编码方案进行序列化，该方案专为在以太坊中准确和字节完美的数据序列化而创建。



交易中的nonce

- 黄皮书定义：一个标量值，等于从这个地址发送的交易数，或者对于关联 code的帐户来说，是这个帐户创建合约的数量。
- nonce不会明确存储为区块链中帐户状态的一部分。相反，它是通过计算发送地址的已确认交易的数量来动态计算的。
- nonce值还用于防止错误计算账户余额。nonce强制来自任何地址的交易按顺序处理，没有间隔，无论节点接收它们的顺序如何。
- 使用nonce确保所有节点计算相同的余额和正确的序列交易，等同于用于防止比特币“双重支付”（“重放攻击”）的机制。但是，由于以太坊跟踪账户余额并且不单独跟踪 UTXO，因此只有在错误地计算账户余额时才会发生“双重支付”。nonce机制可以防止这种情况发生。



并发和nonce

- 以太坊是一个允许操作（节点，客户端，DApps）并发的系统，但强制执行单例状态。例如，出块的时候只有一个系统状态。
- 假如我们有多个独立的钱包应用或客户端，比如 MetaMask和 Geth，它们可以使用相同的地址生成交易。如果我们希望它们都够同时发送交易，该怎么设置交易的nonce呢？
- 用一台服务器为各个应用分配nonce，先来先服务——可能出现单点故障，并且失败的交易会将后续交易阻塞。
- 生成交易后不分配nonce，也不签名，而是把它放入一个队列等待。另起一个节点跟踪nonce并签名交易。同样会有单点故障的可能，而且跟踪nonce和签名的节点是无法实现真正并发的。



交易中的gas

- 当由于交易或消息触发 EVM 运行时，每个指令都会在网络的每个节点上执行。这具有成本：对于每个执行的操作，都存在固定的成本，我们把这个成本用一定量的 gas 表示。
- gas 是交易发起人需要为 EVM 上的每项操作支付的成本名称。发起交易时，我们需要从执行代码的矿工那里用以太币购买 gas 。
- gas 与消耗的系统资源对应，这是具有自然成本的。因此在设计上 gas 和 ether 有意地解耦，消耗的 gas 数量代表了对资源的占用，而对应的交易费用则还跟 gas 对以太的单价有关。这两者是由自由市场调节的：gas 的价格实际上是由矿工决定的，他们可以拒绝处理 gas 价格低于最低限额的交易。我们不需要专门购买 gas ，只需将以太币添加到帐户即可，客户端在发送交易时会自动用以太币购买汽油。而以太币本身的价格通常由于市场力量而波动。



gas的计算

- 发起交易时的 gas limit 并不是要支付的 gas 数量，而只是给定了一个消耗 gas 的上限，相当于“押金”
- 实际支付的 gas 数量是执行过程中消耗的 gas (gasUsed)，gas limit 中剩余的部分会返回给发送人
- 最终支付的 gas 费用是 gasUsed 对应的以太币费用，单价由设定的 gasPrice 而定
- 最终支付费用 $\text{totalCost} = \text{gasPrice} * \text{gasUsed}$
- totalCost 会作为交易手续费 (Tx fee) 支付给矿工



交易的接收者 (to)

- 交易接收者在to字段中指定，是一个20字节的以太坊地址。地址可以是EOA或合约地址。
- 以太坊没有进一步的验证，任何20字节的值都被认为是有效的。如果20字节值对应于没有相应私钥的地址，或不存在的合约，则该交易仍然有效。以太坊无法知道地址是否是从公钥正确派生的。
- 如果将交易发送到无效地址，将销毁发送的以太，使其永远无法访问。
- 验证接收人地址是否有效的工作，应该在用户界面一层完成。



交易的 value 和 data

- 交易的主要“有效负载”包含在两个字段中：value 和 data。交易可以同时有 value 和 data，仅有 value，仅有 data，或者既没有 value 也没有 data。所有四种组合都有效。
- 仅有 value 的交易就是一笔以太的付款
- 仅有 data 的交易一般是合约调用
- 进行合约调用的同时，我们除了传输 data，还可以发送以太，从而交易中同时包含 data 和 value
- 没有 value 也没有 data 的交易，只是在浪费 gas，但它是有效的



向 EOA 或合约传递 data

- 当交易包含数据有效负载时，它很可能是发送到合约地址的，但它同样可以发送给 EOA
- 如果发送 data 给 EOA，数据负载（data payload）的解释取决于钱包
- 如果发送数据负载给合约地址，EVM 会解释为函数调用，从 payload 里解码出函数名称和参数，调用该函数并传入参数
- 发送给合约的数据有效负载是32字节的十六进制序列化编码：
 - 函数选择器：函数原型的 Keccak256 哈希的前4个字节。这允许 EVM 明确地识别将要调用的函数。
 - 函数参数：根据 EVM 定义的各种基本类型的规则进行编码。



特殊交易：创建（部署）合约

- 有一中特殊的交易，具有数据负载且没有 value，那就是一个创建新合约的交易。
- 合约创建交易被发送到特殊目的地地址，即零地址0x0。该地址既不代表 EOA 也不代表合约。它永远不会花费以太或发起交易，它仅用作目的地，具有特殊含义“创建合约”。
- 虽然零地址仅用于合同注册，但它有时会收到来自各种地址的付款。这种情况要么是偶然误操作，导致失去以太；要么是故意销毁以太。
- 合约注册交易不应包含以太值，只包含合约的已编译字节码的数据有效负载。此交易的唯一效果是注册合约。



Q&A



尚硅谷



以太坊虚拟机 (EVM) 简介

2018.9



以太坊虚拟机 (EVM)

- 以太坊虚拟机 EVM 是智能合约的运行环境
- 作为区块验证协议的一部分，参与网络的每个节点都会运行 EVM。他们会检查正在验证的块中列出的交易，并运行由 EVM中的交易触发的代码
- EVM不仅是沙盒封装的，而且是完全隔离的，也就是说在 EVM 中运行的代码是无法访问网络、文件系统和其他进程的，甚至智能合约之间的访问也是受限的
- 合约以字节码的格式 (EVM bytecode) 存在于区块链上
- 合约通常以高级语言 (solidity) 编写，通过EVM编译器编译为字节码，最终通过客户端上载部署到区块链网络中



EVM和账户

- 以太坊中有两类账户：**外部账户** 和 **合约账户**，它们共用 EVM中同一个地址空间
- 无论帐户是否存储代码，这两类账户对 EVM 来说处理方式是完全一样的
- 每个账户在EVM中都有一个键值对形式的持久化存储。其中 key 和 value 的长度都是256位，称之为 **存储空间** (storage)



EVM和交易

- 交易可以看作是从一个帐户发送到另一个帐户的消息，它可以包含二进制数据（payload）和以太币
- 如果目标账户含有代码，此代码会在EVM中执行，并以 payload 作为入参，这就是合约的调用
- 如果目标账户是零账户（账户地址为 0），此交易就将创建一个 **新合约**，这个用来创建合约的交易的 payload 会被转换为 EVM 字节码并执行，执行的输出作为合约代码永久存储



EVM和gas

- 合约被交易触发调用时，指令会在全网的每个节点上执行：这需要消耗算力成本；每一个指令的执行都有特定的消耗，**gas** 就用来量化表示这个成本消耗
- 一经创建，每笔交易都按照一定数量的 gas 预付一笔费用，目的是限制执行交易所需要的工作量和为交易支付手续费
- EVM 执行交易时，gas 将按特定规则逐渐耗尽
- **gas price** 是交易发送者设置的一个值，作为发送者预付手续费的单价。如果交易执行后还有剩余，gas 会原路返还
- 无论执行到什么位置，一旦 gas 被耗尽（比如降为负值），将会触发一个 out-of-gas 异常。当前调用帧（call frame）所做的所有状态修改都将被回滚



EVM数据存储

Storage

- 每个账户都有一块持久化的存储空间，称为 storage，这是一个将256位字映射到256位字的 key-value 存储区，可以理解为合约的数据库
- 永久储存在区块链中，由于会永久保存合约状态变量，所以读写的 gas 开销也最大

Memory (内存)

- 每一次消息调用，合约会临时获取一块干净的内存空间
- 生命周期仅为整个方法执行期间，函数调用后回收，因为仅保存临时变量，故读写 gas 开销较小

Stack (栈)

- EVM 不是基于寄存器的，而是基于栈的，因此所有的计算都在一个被称为栈（stack）的区域执行
- 存放部分局部值类型变量，几乎免费使用的内存，但有数量限制



EVM指令集

- 所有的指令都是针对"256位的字 (word) "这个基本的数据类型来进行操作
- 具备常用的算术、位、逻辑和比较操作，也可以做到有条件和无条件跳转
- 合约可以访问当前区块的相关属性，比如它的块高度和时间戳

消息调用 (Message Calls)

- 合约可以通过消息调用的方式来调用其它合约或者发送以太币到非合约账户
- 合约可以决定在其内部的消息调用中，对于剩余的 gas，应发送和保留多少
- 如果在内部消息调用时发生了 out-of-gas 异常（或其他任何异常），这将由一个被压入栈顶的错误值所指明；此时只有与该内部消息调用一起发送的 gas 会被消耗掉



委托调用 (Delegatecall)

- 一种特殊类型的消息调用
- 目标地址的代码将在发起调用的合约的上下文中执行，并且 msg.sender 和 msg.value 不变
- 可以由此实现 “库” (library)：可复用的代码库可以放在一个合约的存储上，通过委托调用引入相应代码



合约的创建和自毁

- 通过一个特殊的消息调用 **create calls**，合约可以创建其他合约（不是简单的调用零地址）
- 合约代码从区块链上移除的唯一方式是合约在合约地址上的执行自毁操作 **selfdestruct**；合约账户上剩余的以太币会发送给指定的目标，然后其存储和代码从状态中被移除



Q&A



尚硅谷



Solidity 简介

2018.10



Solidity是什么

- Solidity 是一门面向合约的、为实现智能合约而创建的高级编程语言。这门语言受到了 C++, Python 和 Javascript 语言的影响，设计的目的是能在以太坊虚拟机（EVM）上运行。
- Solidity 是静态类型语言，支持继承、库和复杂的用户定义类型等特性。
- 内含的类型除了常见编程语言中的标准类型，还包括 address 等以太坊独有的类型，Solidity 源码文件通常以 .sol 作为扩展名
- 目前尝试 Solidity 编程的最好的方式是使用 [Remix](#)。Remix 是一个基于 Web 浏览器的 IDE，它可以让你编写 Solidity 智能合约，然后部署并运行该智能合约。



Solidity语言特性

Solidity的语法接近于JavaScript，是一种面向对象的语言。但作为一种真正意义上运行在网络上的去中心合约，它又有很多的不同：

- 以太坊底层基于帐户，而不是 UTXO，所以增加了一个特殊的数据类型 address 用于定位用户和合约账户。
- 语言内嵌框架支持支付。提供了 payable 等关键字，可以在语言层面直接支持支付。
- 使用区块链进行数据存储。数据的每一个状态都可以永久存储，所以在使用时需要确定变量使用内存，还是区块链存储。
- 运行环境是在去中心化的网络上，所以需要强调合约或函数执行的调用的方式。
- 不同的异常机制。一旦出现异常，所有的执行都将会被回撤，这主要是为了保证合约执行的原子性，以避免中间状态出现的数据不一致。

Solidity源码和智能合约

- Solidity 源代码要成为可以运行在以太坊上的智能合约需要经历如下的步骤：
 1. 用 Solidity 编写的智能合约源代码需要先使用编译器编译为字节码 (Bytecode) , 编译过程中会同时产生智能合约的二进制接口规范 (Application Binary Interface, 简称为 ABI) ;
 2. 通过交易 (Transaction) 的方式将字节码部署到以太坊网络, 每次成功部署都会产生一个新的智能合约账户;
 3. 使用 Javascript 编写的 DApp 通常通过 web3.js + ABI去调用智能合约中的函数来实现数据的读取和修改。

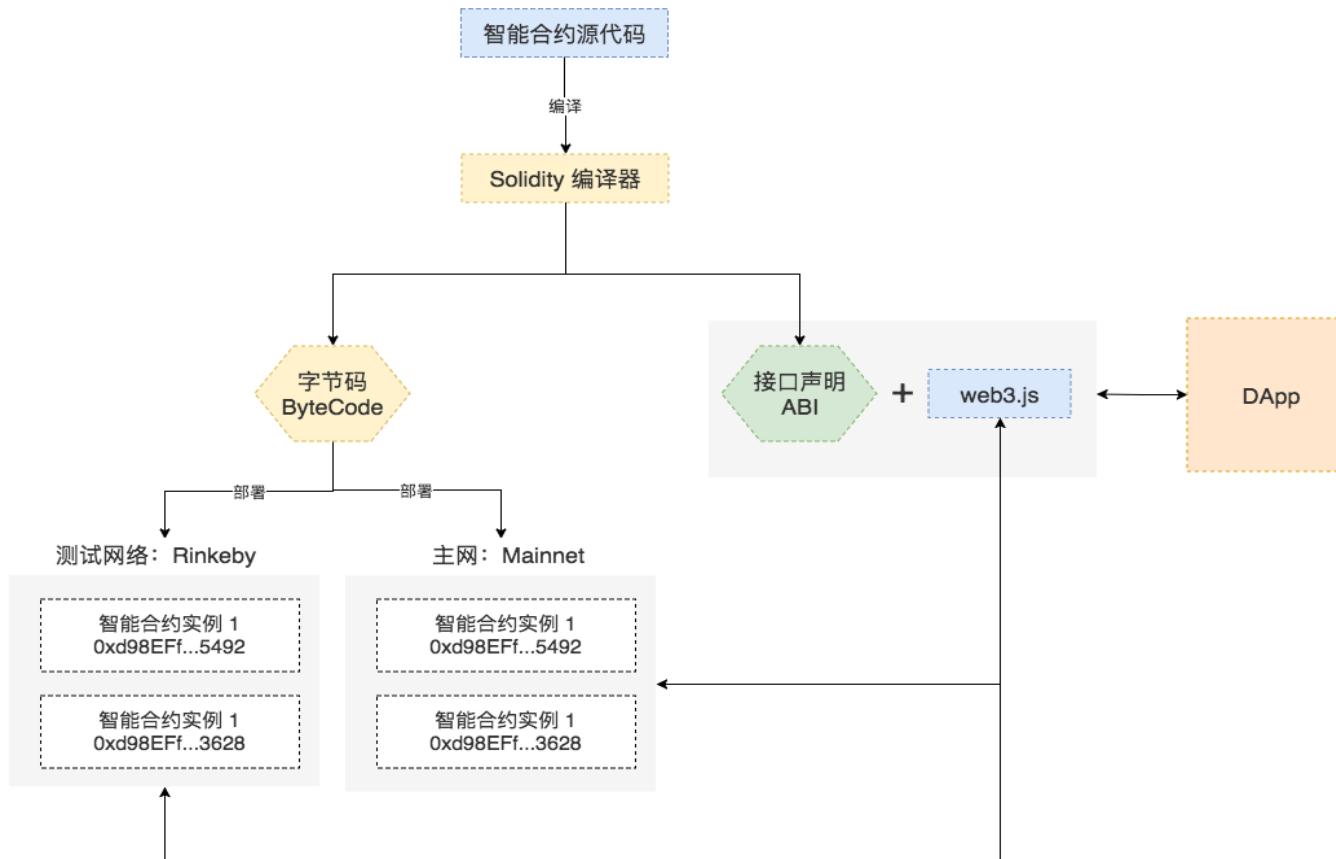
Solidity编译器

Remix

- Remix 是一个基于 Web 浏览器的 Solidity IDE；可在线使用而无需安装任何东西
- <http://remix.ethereum.org>

solcjs

- solc 是 Solidity 源码库的构建目标之一，它是 Solidity 的命令行编译器
- 使用 *npm* 可以便捷地安装 Solidity 编译器 solcjs
- *npm install -g solc*





一个简单的智能合约

```
pragma solidity ^0.4.0;

contract SimpleStorage {

    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```



智能合约概述

Solidity中合约

- 一组代码（合约的函数）和数据（合约的状态），它们位于以太坊区块链的一个特定地址上
- 代码行 `uint storedData;` 声明一个类型为 `uint` (256位无符号整数) 的状态变量，叫做 `storedData`
- 函数 `set` 和 `get` 可以用来变更或取出变量的值



合约结构

- 状态变量 (State Variables)

作为合约状态的一部分，值会永久保存在存储空间内。

- 函数 (Functions)

合约中可执行的代码块。

- 函数修饰器 (Function Modifiers)

用在函数声明中，用来补充修饰函数的语义。

- 事件 (Events)

非常方便的 EVM 日志工具接口。



智能合约练习

pragma solidity >0.4.22;

```
contract Car {  
    string public brand;  
    uint public price;  
    constructor(string initBrand, uint initPrice){  
        brand = initBrand;  
        price = initPrice;  
    };  
    function setBrand(string newBrand) public {  
        brand = newBrand;  
    }  
    function setPrice(uint newPrice)(uint) {  
        price = newPrice;  
    }  
}
```



pragma solidity >0.4.22 <0.6.0; 另一个例子——子货币
contract Coin {

address public minter;

mapping (address => uint) public balances;

event Sent(address from, address to, uint amount);

constructor() public { minter = msg.sender; }

function mint(address receiver, uint amount) public {

require(msg.sender == minter);

balances[receiver] += amount;

}

function send(address receiver, uint amount) public {

require(amount <= balances[msg.sender]);

balances[msg.sender] -= amount;

balances[receiver] += amount;

emit Sent(msg.sender, receiver, amount);

让天下没有难学的技术



合约代码解读

address public minter;

- 这一行声明了一个可以被公开访问的 address 类型的状态变量。
- 关键字 public 自动生成一个函数，允许你在这个合约之外访问这个状态变量的当前值。

mapping(address => uint) public balances;

- 也创建一个公共状态变量，但它是一个更复杂的数据类型，该类型将 address 映射为无符号整数。
- *mappings* 可以看作是一个**哈希表**，它会执行虚拟初始化，把所有可能存在的键都映射到一个字节表示为全零的值。



合约代码解读

event Sent(*address from, address to, uint amount*);

- 声明了一个“事件”(event)，它会在 send 函数的最后一行触发
- 用户可以监听区块链上正在发送的事件，而不会花费太多成本。一旦它被发出，监听该事件的listener都将收到通知
- 所有的事件都包含了 from, to 和 amount 三个参数，可方便追踪事务

emit Sent(*msg.sender, receiver, amount*);

- 触发Sent事件，并将参数传入



事件的监听

```
Coin.Sent().watch({}, "", function(error, result) {  
    if (!error) {  
        console.log("Coin transfer: " + result.args.amount +  
            "coins were sent from " + result.args.from +  
            " to " + result.args.to + ".");  
        console.log("Balances now:\n" +  
            "Sender: " +  
            Coin.balances.call(result.args.from) +  
            "Receiver: " +  
            Coin.balances.call(result.args.to));  
    }  
});
```

```
pragma solidity >0.4.22 <0.6.0;

contract Coin {

    address public minter;

    mapping (address => uint) public balances;

    event Sent(address from, address to, uint amount);

    constructor() public { minter = msg.sender; }

    function mint(address receiver, uint amount) public {

        require(msg.sender == minter);

        balances[receiver] += amount;

    }

    function send(address receiver, uint amount) public {

        require(amount <= balances[msg.sender]);

        balances[msg.sender] -= amount;

        balances[receiver] += amount;

        emit Sent(msg.sender, receiver, amount);

    }

}
```



Ballot -- 一个简单的投票合约

- 电子投票的主要问题是如何将投票权分配给正确的人员以及如何防止被操纵。这个合约展示了如何进行委托投票，同时，计票又是 **自动和完全透明的**
- 为每个（投票）表决创建一份合约，然后作为合约的创造者——即主席，将给予每个独立的地址以投票权
- 地址后面的人可以选择自己投票，或者委托给他们信任的人来投票
- 在投票时间结束时，`winningProposal()` 将返回获得最多投票的提案



Q&A



尚硅谷



Solidity 深入理解

2018.10



Solidity源文件布局

pragma (版本杂注)

- 源文件可以被版本 杂注**pragma**所注解，表明要求的编译器版本
- 例如： **pragma solidity ^0.4.0;**
- 源文件将既不允许低于 0.4.0 版本的编译器编译，也不允许高于（包含） 0.5.0 版本的编译器编译（第二个条件因使用 ^ 被添加）

import (导入其它源文件)

- Solidity 所支持的导入语句**import**，语法同 JavaScript（从 ES6 起）非常类似

Solidity源文件布局-- import

import "filename";

- 从 “filename” 中导入所有的全局符号到当前全局作用域中

import * as symbolName from "filename";

- 创建一个新的全局符号 symbolName， 其成员均来自 “filename” 中全局符号

import {symbol1 as alias, symbol2} from "filename";

- 创建新的全局符号 alias 和 symbol2， 分别从 "filename" 引用 symbol1 和 symbol2

import "filename" as symbolName;

- 这条语句等同于 `import * as symbolName from "filename";`



Solidity值类型

- 布尔 (bool) : 可能的取值为字符常量值 true 或 false
- 整型 (int/uint) : 分别表示有符号和无符号的不同位数的整型变量； 支持关键字 uint8 到 uint256 (无符号, 从 8 位到 256 位) 以及 int8 到 int256, 以 8 位为步长递增
- 定长浮点型 (fixed / ufixed) : 表示各种大小的有符号和无符号的定长浮点型；在关键字 ufixedMxN 和 fixedMxN 中, M 表示该类型占用的位数, N 表示可用的小数位数
- 地址 (address) : 存储一个 20 字节的值 (以太坊地址大小)
- 定长字节数组：关键字有 bytes1, bytes2, bytes3, ..., bytes32
- 枚举 (enum) : 一种用户可以定义类型的方法，与C语言类似，默认从0开始递增，一般用来模拟合约的状态
- 函数 (function) : 一种表示函数的类型



Solidity引用类型

数组 (Array)

- 数组可以在声明时指定长度（定长数组），也可以动态调整大小（变长数组、动态数组）
- 对于存储型（storage）的数组来说，元素类型可以是任意的（即元素也可以是数组类型，映射类型或者结构体）；对于内存型（memory）的数组来说，元素类型不能是映射（mapping）类型
- Solidity 支持通过构造结构体的形式定义新的类型
- 映射（Mapping）
- 映射可以视作 哈希表，在实际的初始化过程中创建每个可能的 key，并将其映射到字节形式全是零的值（类型默认值）



Solidity地址类型

address

- 地址类型存储一个 20 字节的值（以太坊地址的大小）；地址类型也有成员变量，并作为所有合约的基础

address payable (v0.5.0引入)

- 与地址类型基本相同，不过多出了 transfer 和 send 两个成员变量

两者区别和转换

- Payable 地址是可以发送 ether 的地址，而普通 address 不能
- 允许从 payable address 到 address 的隐式转换，而反过来的直接转换是不可能的（唯一方法是通过uint160来进行中间转换）
- 从0.5.0版本起，合约不再是从地址类型派生而来，但如果它有payable的回退函数，那同样可以显式转换为 address 或者 address payable 类型



地址类型成员变量

<address>.balance (uint256)

- 该地址的 ether 余额，以Wei为单位

<address payable>.transfer(uint256 amount)

- 向指定地址发送数量为 amount 的 ether (以Wei为单位) , 失败时抛出异常, 发送 2300 gas 的矿工费, 不可调节

<address payable>.send(uint256 amount) returns (bool)

- 向指定地址发送数量为 amount 的 ether (以Wei为单位) , 失败时返回 false, 发送 2300 gas 的矿工费用, 不可调节

<address>.call(bytes memory) returns (bool, bytes memory)

- 发出底层函数 CALL, 失败时返回 false, 发送所有可用 gas, 可调节

<address>.delegatecall(bytes memory) returns (bool, bytes memory)

- 发出底层函数 DELEGATECALL, 失败时返回 false, 发送所有可用 gas, 可调节

<address>.staticcall(bytes memory) returns (bool, bytes memory)

- 发出底层函数 STATICCALL , 失败时返回 false, 发送所有可用 gas, 可调节



地址成员变量用法

balance 和 transfer

- 可以使用 balance 属性来查询一个地址的余额，可以使用 transfer 函数向一个 payable 地址发送以太币 Ether (以 wei 为单位)

```
address payable x = address(0x123);
address myAddress = address(this);
if (x.balance < 10 && myAddress.balance >= 10)
    x.transfer(10);
```

send

- send 是 transfer 的低级版本。如果执行失败，当前的合约不会因为异常而终止，但 send 会返回 false

call

- 也可以用 call 来实现转币的操作，通过添加 .gas() 和 .value() 修饰器：

```
nameReg.call.gas(1000000).value(1
    ether)(abi.encodeWithSignature("register(string)", "MyName"));
```

没有难学的技术



字符数组 (Byte Arrays)

定长字符数组

- 属于值类型，bytes1, bytes2, ..., bytes32分别代表了长度为1到32的字节序列
- 有一个.length属性，返回数组长度（只读）

变长字符数组

- 属于引用类型，包括 bytes和string，不同的是bytes是Hex字符串，而string是UTF-8编码的字符串



枚举 (Enum)

- 枚举类型用来用户自定义一组常量值
- 与C语言的枚举类型非常相似，对应整型值

```
pragma solidity >=0.4.0 <0.6.0;
```

```
contract Purchase {  
    enum State { Created, Locked, Inactive }  
}
```



数组 (Array)

- 固定大小k和元素类型T的数组被写为T [k]，动态大小的数组为T []。例如，一个由5个uint动态数组组成的数组是uint [] [5]
- 要访问第三个动态数组中的第二个uint，可以使用x [2] [1]
- 越界访问数组，会导致调用失败回退
- 如果要添加新元素，则必须使用.push () 或将.length增大
- 变长的storage数组和bytes（不包括string）有一个push()方法。可以将一个新元素附加到数组末端，返回值为当前长度



数组示例

```
pragma solidity >=0.4.16 <0.6.0;

contract C {

    function f(uint len) public pure {

        uint[] memory a = new uint[](7);

        bytes memory b = new bytes(len);

        assert(a.length == 7);

        assert(b.length == len);

        a[6] = 8;

    }

}
```



结构 (Struct)

- 结构类型可以在映射和数组中使用，它们本身可以包含映射和数组。
- 结构不能包含自己类型的成员，但可以作为自己数组成员的类型，也可以作为自己映射成员的值类型

```
pragma solidity >=0.4.0 <0.6.0;
contract Ballot {
    struct Voter {
        uint weight;
        bool voted;
        uint vote;
    }
}
```



映射 (Mapping)

- 声明一个映射: `mapping (_KeyType => _ValueType)`
- `_KeyType`可以是任何基本类型。这意味着它可以是任何内置值类型加上字节和字符串。不允许使用用户定义的或复杂的类型, 如枚举, 映射, 结构以及除`bytes`和`string`之外的任何数组类型。
- `_ValueType`可以是任何类型, 包括映射。

```
pragma solidity >=0.4.0 <0.6.0;

contract MappingExample {
    mapping(address => uint) public balances;
    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}

contract MappingUser {
    function f() public returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
        return m.balances(address(this));
    }
}
```



Solidity 数据位置

- 所有的复杂类型，即数组、结构和映射类型，都有一个额外属性，“数据位置”，用来说明数据是保存在内存 memory 中还是存储 storage 中
- 根据上下文不同，大多数时候数据有默认的位置，但也可以通过在类型名后增加关键字 storage 或 memory 进行修改
- 函数参数（包括返回的参数）的数据位置默认是 memory，局部变量的数据位置默认是 storage，状态变量的数据位置强制是 storage
- 另外还存在第三种数据位置，calldata，这是一块只读的，且不会永久存储的位置，用来存储函数参数。外部函数的参数（非返回参数）的数据位置被强制指定为 calldata，效果跟 memory 差不多



数据位置总结

强制指定的数据位置

- 外部函数的参数（不包括返回参数）： calldata；
- 状态变量： storage

默认数据位置

- 函数参数（包括返回参数）： memory；
- 引用类型的局部变量： storage
- 值类型的局部变量： 栈（stack）

特别要求

- 公开可见（publicly visible）的函数参数一定是 memory 类型，如果要求是 storage 类型则必须是 private 或者 internal 函数，这是为了防止随意的公开调用占用资源

```
// 一个简单的例子
pragma solidity ^0.4.0;
contract C {
    uint[] data1;
    uint[] data2;
    function appendOne() public {
        append(data1);
    }
    function appendTwo() public {
        append(data2);
    }
    function append(uint[] storage d) internal {
        d.push(1);
    }
}
```

/// 下面代码包含一个错误

```
pragma solidity ^0.4.0;

contract C {

    uint someVariable;

    uint[] data;

    function f() public {

        uint[] x;

        x.push(2);

        data = x;

    }

}
```

```
// 下面代码编译错误
pragma solidity ^0.4.0;
contract C {
    uint[] x;
    function f(uint[] memoryArray) public {
        x = memoryArray;
        uint[] y = x;
        y[7];
        y.length = 2;
        delete x;
        y = memoryArray;
        delete y;
        g(x);
        h(x);
    }
    function g(uint[] storage storageArray) internal {}
    function h(uint[] memoryArray) public {} }
```

```
// 下面我们一起来玩一个猜数字游戏
pragma solidity >0.4.22;

contract Honeypot{
    uint luckyNum = 52;
    uint public last;
    struct Guess{ address player; uint number; }
    Guess[] public guessHistory;
    address owner = msg.sender;
    function guess(uint _num) public payable{
        Guess newGuess;
        newGuess.player = msg.sender;
        newGuess.number = _num;
        guessHistory.push( newGuess );
        if( _num == luckyNum )
            msg.sender.transfer( msg.value * 2 );
        last = now;
    }
}
```

Solidity 函数声明和类型

```
function getBrand() public view returns (string) {  
    return brand;  
}
```

函数名称

函数类型

返回类型

函数的值类型有两类： - 内部 (*internal*) 函数和 外部 (*external*) 函数

- 内部函数只能在当前合约内被调用（更具体来说，在当前代码块内，包括内部库函数和继承的函数中），因为它们不能在当前合约上下文的外部被执行。调用一个内部函数是通过跳转到它的入口标签来实现的，就像在当前合约的内部调用一个函数。
- 外部函数由一个地址和一个函数签名组成，可以通过外部函数调用传递或者返回
- 调用内部函数：直接使用名字 f
- 调用外部函数：this.f (当前合约) , a.f (外部合约)



Solidity 函数可见性

函数的可见性可以指定为 external, public , internal 或者 private;

对于状态变量，不能设置为 external ， 默认是 internal。

- external : 外部函数作为合约接口的一部分，意味着我们可以从其他合约和交易中调用。一个外部函数 f 不能从内部调用（即 f 不起作用，但 this.f() 可以）。当收到大量数据的时候，外部函数有时候会更有效率。
- public : public 函数是合约接口的一部分，可以在内部或通过消息调用。对于 public 状态变量，会自动生成一个 getter 函数。
- internal : 这些函数和状态变量只能是内部访问（即从当前合约内部或从它派生的合约访问），不使用 this 调用。
- private : private 函数和状态变量仅在当前定义它们的合约中使用，并且不能被派生合约使用。

// 以下代码编译错误

```
pragma solidity >=0.4.0 <0.6.0;

contract C {
    uint private data;

    function f(uint a) private pure returns(uint b) { return a + 1; }

    function setData(uint a) public { data = a; }

    function getData() public view returns(uint) { return data; }

    function compute(uint a, uint b) internal pure returns (uint) { return a + b; }
}

contract D {
    function readData() public {
        C c = new C();
        uint local = c.f(7);
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5);
    }
}

contract E is C {
    function g() public {
        C c = new C();
        uint val = compute(3, 5);
    }
}
```



函数可见性示例

```
pragma solidity >=0.4.16 <0.6.0;

contract C {

    function f(uint a) private pure returns (uint b) {
        return a + 1;
    }

    function setData(uint a) internal {
        data = a;
    }

    uint public data;

    function x() public {
        data = 3; // 内部访问
        uint val = this.data(); // 外部访问
        uint val2 = f(data);
    }
}
```



// 下面代码编译错误

```
pragma solidity ^0.4.0;

contract C {
    uint private data;

    function f(uint a) private returns(uint b) { return a + 1; }

    function setData(uint a) public { data = a; }

    function getData() public returns(uint) { return data; }

    function compute(uint a, uint b) internal returns (uint) { return a+b; }
}

contract D {
    function readData() public {
        C c = new C();
        uint local = c.f(7); // 错误：成员 `f` 不可见
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5); // 错误：成员 `compute` 不可见
    }
}

contract E is C {
```



Solidity函数状态可变性

- `pure`: 纯函数，不允许修改或访问状态
- `view`: 不允许修改状态
- `payable`: 允许从消息调用中接收以太币Ether。
- `constant`: 与`view`相同，一般只修饰状态变量，不允许赋值
(除初始化以外)



Solidity 函数状态可变性

以下情况被认为是修改状态：

- 修改状态变量。
- 产生事件。
- 创建其它合约。
- 使用 `selfdestruct`。
- 通过调用发送以太币。
- 调用任何没有标记为 `view` 或者 `pure` 的函数。
- 使用低级调用。
- 使用包含特定操作码的内联汇编。



Solidity 函数状态可变性

以下被认为是从状态中进行读取：

- 读取状态变量。
- 访问 `this.balance` 或者 `<address>.balance`。
- 访问 `block`, `tx`, `msg` 中任意成员 (除 `msg.sig` 和 `msg.data` 之外) 。
- 调用任何未标记为 `pure` 的函数。
- 使用包含某些操作码的内联汇编。



函数修饰器 (modifier)

- 使用 修饰器modifier 可以轻松改变函数的行为。例如，它们可以在执行函数之前自动检查某个条件。 修饰器modifier 是合约的可继承属性， 并可能被派生合约覆盖
- 如果同一个函数有多个 修饰器modifier， 它们之间以空格隔开， 修饰器modifier 会依次检查执行。



Modifier示例

```
pragma solidity >=0.4.22 <0.6.0;

contract Purchase {

    address public seller;

    modifier onlySeller() { // Modifier
        require( msg.sender == seller, "Only seller can call." );
        _;
    }

    function abort() public view onlySeller {
        // Modifier usage
        // ...
    }
}
```



回退函数 (fallback)

- 回退函数 (fallback function) 是合约中的特殊函数；没有名字，不能有参数也不能有返回值
- 如果在一个到合约的调用中，没有其他函数与给定的函数标识符匹配（或没有提供调用数据），那么这个函数 (fallback 函数) 会被执行
- 每当合约收到以太币（没有任何数据），回退函数就会执行。此外，为了接收以太币，fallback 函数必须标记为 payable。如果不存在这样的函数，则合约不能通过常规交易接收以太币
- 在上下文中通常只有很少的 gas 可以用来完成回退函数的调用，所以使 fallback 函数的调用尽量廉价很重要

```
pragma solidity >0.4.99 <0.6.0;

contract Sink {

    function() external payable { }

}

contract Test {

    function() external { x = 1; }

    uint x;

}

contract Caller {

    function callTest(Test test) public returns (bool) {

        (bool success,) = address(test).call(abi.encodeWithSignature("nonExistingFunction()));

        require(success);

        address payable testPayable = address(uint160(address(test)));

        return testPayable.send(2 ether);

    }

}
```



事件 (event)

- 事件是以太坊EVM提供的一种日志基础设施。事件可以用来做操作记录，存储为日志。也可以用来实现一些交互功能，比如通知UI，返回函数调用结果等
- 当定义的事件触发时，我们可以将事件存储到EVM的交易日志中，日志是区块链中的一种特殊数据结构；日志与合约关联，与合约的存储合并存入区块链中；只要某个区块可以访问，其相关的日志就可以访问；但在合约中，我们不能直接访问日志和事件数据
- 可以通过日志实现简单支付验证 SPV (Simplified Payment Verification)，如果一个外部实体提供了一个带有这种证明的合约，它可以检查日志是否真实存在于区块链中



Solidity异常处理

- Solidity使用“状态恢复异常”来处理异常。这样的异常将撤消对当前调用（及其所有子调用）中的状态所做的所有更改，并且向调用者返回错误。
- 函数assert和require可用于判断条件，并在不满足条件时抛出异常
- assert()一般只应用于测试内部错误，并检查常量
- require()应用于确保满足有效条件（如输入或合约状态变量），或验证调用外部合约的返回值
- revert()用于抛出异常，它可以标记一个错误并将当前调用回退



Solidity中的单位

以太币 (ether)

- 以太币 Ether 单位之间的换算就是在数字后边加上 wei、finney、szabo 或 ether 来实现的，如果后面没有单位，缺省为 Wei。例如 2 ether == 2000 finney 的逻辑判断值为 true

Unit	Wei Value	Wei
wei	1	1 wei
Kwei (babbage)	1e3 wei	1,000
Mwei (lovelace)	1e6 wei	1,000,000
Gwei (shannon)	1e9 wei	1,000,000,000
microether (szabo)	1e12 wei	1,000,000,000,000
milliether (finney)	1e15 wei	1,000,000,000,000,000
ether	1e18 wei	1,000,000,000,000,000,000



Solidity中的单位

时间

秒是缺省时间单位，在时间单位之间，数字后面带

有 seconds、minutes、hours、days、weeks 和 years 的可以进行换算，基本换算关系如下：

- $1 == 1 \text{ seconds}$
- $1 \text{ minutes} == 60 \text{ seconds}$
- $1 \text{ hours} == 60 \text{ minutes}$
- $1 \text{ days} == 24 \text{ hours}$
- $1 \text{ weeks} == 7 \text{ days}$
- $1 \text{ years} == 365 \text{ days}$

这些后缀不能直接用在变量后边。如果想用时间单位（例如 days）来将输入变量换算为时间，你可以用如下方式来完成：

```
function f(uint start, uint daysAfter) public {  
    if (now >= start + daysAfter * 1 days) { // ... }  
}
```



Q&A



尚硅谷



ERC20 代币合约

```
pragma solidity ^0.4.16;

interface tokenRecipient { function receiveApproval(address _from, uint256 _value, address _token, bytes _extraData) external; }

contract TokenERC20 {
    // Public variables of the token
    string public name;
    string public symbol;
    uint8 public decimals = 18;
    // 18 decimals is the strongly suggested default, avoid changing it
    uint256 public totalSupply;

    // This creates an array with all balances
    mapping (address => uint256) public balanceOf;
    mapping (address => mapping (address => uint256)) public allowance;

    // This generates a public event on the blockchain that will notify clients
    event Transfer(address indexed from, address indexed to, uint256 value);

    // This generates a public event on the blockchain that will notify clients
    event Approval(address indexed _owner, address indexed _spender, uint256 _value);

    // This notifies clients about the amount burnt
    event Burn(address indexed from, uint256 value);

    /**
     * Constructor function
     *
     * Initializes contract with initial supply tokens to the creator of the contract
     */
    function TokenERC20(
        uint256 initialSupply,
        string tokenName,
        string tokenSymbol
    ) public {
        totalSupply = initialSupply * 10 ** uint256(decimals); /*Update total supply with the
                                                               decimal amount*/
    }
}
```



```
balanceOf[msg.sender] = totalSupply;           // Give the creator all initial tokens
name = tokenName;                            // Set the name for display purposes
symbol = tokenSymbol;                         // Set the symbol for display purposes
}

/**
 * Internal transfer, only can be called by this contract
 */
function _transfer(address _from, address _to, uint _value) internal {
    // Prevent transfer to 0x0 address. Use burn() instead
    require(_to != 0x0);
    // Check if the sender has enough
    require(balanceOf[_from] >= _value);
    // Check for overflows
    require(balanceOf[_to] + _value >= balanceOf[_to]);
    // Save this for an assertion in the future
    uint previousBalances = balanceOf[_from] + balanceOf[_to];
    // Subtract from the sender
    balanceOf[_from] -= _value;
    // Add the same to the recipient
    balanceOf[_to] += _value;
    emit Transfer(_from, _to, _value);
    // Asserts are used to use static analysis to find bugs in your code. They should never
fail
    assert(balanceOf[_from] + balanceOf[_to] == previousBalances);
}

/**
 * Transfer tokens
 *
 * Send `_value` tokens to `_to` from your account
 *
 * @param _to The address of the recipient
 * @param _value the amount to send
 */
function transfer(address _to, uint256 _value) public returns (bool success) {
    _transfer(msg.sender, _to, _value);
    return true;
}

/**
 * Transfer tokens from other address
 *
```



```
* Send `_value` tokens to `_to` on behalf of `_from`
*
* @param _from The address of the sender
* @param _to The address of the recipient
* @param _value the amount to send
*/
function transferFrom(address _from, address _to, uint256 _value) public returns (bool
success) {
    require(_value <= allowance[_from][msg.sender]);          // Check allowance
    allowance[_from][msg.sender] -= _value;
    _transfer(_from, _to, _value);
    return true;
}

/**
 * Set allowance for other address
 *
 * Allows `_spender` to spend no more than `_value` tokens on your behalf
 *
 * @param _spender The address authorized to spend
 * @param _value the max amount they can spend
 */
function approve(address _spender, uint256 _value) public
    returns (bool success) {
    allowance[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
}

/**
 * Set allowance for other address and notify
 *
 * Allows `_spender` to spend no more than `_value` tokens on your behalf, and then ping
the contract about it
 *
 * @param _spender The address authorized to spend
 * @param _value the max amount they can spend
 * @param _extraData some extra information to send to the approved contract
 */
function approveAndCall(address _spender, uint256 _value, bytes _extraData)
    public
    returns (bool success) {
    tokenRecipient spender = tokenRecipient(_spender);
```



```
if (approve(_spender, _value)) {
    spender.receiveApproval(msg.sender, _value, this, _extraData);
    return true;
}

/***
 * Destroy tokens
 *
 * Remove `_value` tokens from the system irreversibly
 *
 * @param _value the amount of money to burn
 */
function burn(uint256 _value) public returns (bool success) {
    require(balanceOf[msg.sender] >= _value); // Check if the sender has enough
    balanceOf[msg.sender] -= _value;           // Subtract from the sender
    totalSupply -= _value;                    // Updates totalSupply
    emit Burn(msg.sender, _value);
    return true;
}

/***
 * Destroy tokens from other account
 *
 * Remove `_value` tokens from the system irreversibly on behalf of `_from`.
 *
 * @param _from the address of the sender
 * @param _value the amount of money to burn
 */
function burnFrom(address _from, uint256 _value) public returns (bool success) {
    require(balanceOf[_from] >= _value);        // Check if the targeted balance is enough
    require(_value <= allowance[_from][msg.sender]); // Check allowance
    balanceOf[_from] -= _value;                  // Subtract from the targeted balance
    allowance[_from][msg.sender] -= _value;       // Subtract from the sender's allowance
    totalSupply -= _value;                      // Update totalSupply
    emit Burn(_from, _value);
    return true;
}
```

web3.js 简介

2018.10



web3.js

- Web3 JavaScript app API
- web3.js 是一个JavaScript API库。要使DApp在以太坊上运行，我们可以使用web3.js库提供的web3对象
- web3.js 通过RPC调用与本地节点通信，它可以用于任何暴露了RPC层的以太坊节点
- web3 包含 eth 对象 - web3.eth (专门与以太坊区块链交互) 和 shh 对象 - web3.shh (用于与 Whisper 交互)



web3 模块加载

- 首先需要将 web3 模块安装在项目中：

```
npm install web3@0.20.1
```

- 然后创建一个 web3 实例，设置一个 “provider”
- 为了保证我们的 MetaMask 设置好的 provider 不被覆盖掉，在引入 web3 之前我们一般要做当前环境检查（以v0.20.1为例）：

```
if (typeof web3 !== 'undefined') {  
    web3 = new Web3(web3.currentProvider);  
}  
else {  
    web3 = new Web3(new Web3.providers  
        .HttpProvider("http://localhost:8545"));  
}
```



异步回调 (callback)

- web3js API 设计的最初目的，主要是为了和本地 RPC 节点共同使用，所以默认情况下发送的是同步 HTTP 请求
- 如果要发送异步请求，可以在函数的最后一个参数位置上，传入一个回调函数。回调函数是可选 (optional) 的
- 我们一般采用的回调风格是所谓的“错误优先”，例如：

```
web3.eth.getBlock(48, function(error, result){  
  if(!error)  
    console.log(JSON.stringify(result));  
  else  
    console.error(error);  
});
```



回调 Promise 事件 (v1.0.0)

- 为了帮助 web3 集成到不同标准的所有类型项目中，1.0.0 版本提供了多种方式来处理异步函数。大多数的 web3 对象允许将一个回调函数作为最后一个函数参数传入，同时会返回一个 promise 用于链式函数调用。
- 以太坊作为一个区块链系统，一次请求具有不同的结束阶段。为了满足这样的要求，1.0.0 版本将这类函数调用的返回值包成一个“承诺事件”(promiseEvent)，这是一个 promise 和 EventEmitter 的结合体。
- PromiEvent 的用法就像 promise 一样，另外还加入了.on, .once 和.off方法

```
web3.eth.sendTransaction({from: '0x123...', data: '0x432...'})  
  .once('transactionHash', function(hash){ ... })  
  .once('receipt', function(receipt){ ... })  
  .on('confirmation', function(confNumber, receipt){ ... })  
  .on('error', function(error){ ... })  
  .then(function(receipt){ // will be fired once the receipt is mined });
```



应用二进制接口 (ABI)

- web3.js 通过以太坊智能合约的 json 接口 (Application Binary Interface, ABI) 创建一个 JavaScript 对象，用在 js 代码中描述
- 函数 (functions)
 - type: 函数类型，默认 “function”，也可能是 “constructor”
 - constant, payable, stateMutability: 函数的状态可变性
 - inputs, outputs: 函数输入、输出参数描述列表
- 事件 (events)
 - type: 类型，总是 “event”
 - inputs: 输入对象列表，包括 name、type、indexed



批处理请求 (batch requests)

- 批处理请求允许我们将请求排序，然后一起处理它们。
- 注意：批量请求不会更快。实际上，在某些情况下，一次性地发出许多请求会更快，因为请求是异步处理的。
- 批处理请求主要用于确保请求的顺序，并串行处理。

```
var batch = web3.createBatch();
batch.add(web3.eth.getBalance.request('0x00000000000000000000
00000000000000000000000000000000', 'latest', callback));
batch.add(web3.eth.contract(abi).at(address).balance.request(a
ddress, callback2));
batch.execute();
```



大数处理 (big numbers)

- JavaScript 中默认的数字精度较小，所以 web3.js 会自动添加一个依赖库 BigNumber，专门用于大数处理
- 对于数值，我们应该习惯把它转换成 BigNumber 对象来处理

```
var balance = new  
  BigNumber('131242344353464564564574574567456');  
  
// or var balance = web3.eth.getBalance(someAddress);  
  
balance.plus(21).toString(10);  
  
//"131242344353464564564574574567477"
```

- BigNumber.toString(10) 对小数只保留20位浮点精度。所以推荐的做法是，我们内部总是用 wei 来表示余额（大整数），只有在需要显示给用户看的时候才转换为ether或其它单位



常用 API —— 基本信息查询

查看 web3 版本

- v0.2x.x: `web3.version.api`
- v1.0.0: `web3.version`

查看 web3 连接到的节点版本 (`clientVersion`)

- 同步: `web3.version.node`
- 异步:

```
web3.version.getNode((error,result)=>{console.log(result)})
```

- v1.0.0: `web3.eth.getNodeInfo().then(console.log)`

基本信息查询

获取 *network id*

- 同步: `web3.version.network`
- 异步: `web3.version.getNetwork((err, res)=>{console.log(res)})`
- v1.0.0: `web3.eth.net.getId().then(console.log)`

获取节点的以太坊协议版本

- 同步: `web3.version.ethereum`
- 异步: `web3.version.getEthereum((err, res)=>{console.log(res)})`
- v1.0.0: `web3.eth.getProtocolVersion().then(console.log)`



网络状态查询

是否有节点连接/监听，返回true/false

- 同步： web3.isConnected() 或者 web3.net.listening
- 异步： web3.net.getListening((err,res)=>console.log(res))
- v1.0.0： web3.eth.net.isListening().then(console.log)

查看当前连接的peer 节点

- 同步： web3.net.peerCount
- 异步： web3.net.getPeerCount((err,res)=>console.log(res))
- v1.0.0： web3.eth.net.getPeerCount().then(console.log)

Provider

查看当前设置的 web3 provider

- `web3.currentProvider`

查看浏览器环境设置的 web3 provider (v1.0.0)

- `web3.givenProvider`

设置 provider

- `web3.setProvider(provider)`

- `web3.setProvider(new web3.providers.HttpProvider('http://localhost:8545'))`



web3 通用工具方法

以太单位转换

- web3.fromWei web3.toWei

数据类型转换

- web3.toString web3.toDecimal web3.toBigNumber

字符编码转换

- web3.toHex web3.toAscii web3.toUtf8 web3.fromUtf8

地址相关

- web3.isAddress web3.toChecksumAddress



web3.eth – 账户相关

coinbase 查询

- 同步: web3.eth.coinbase
- 异步: web3.eth.getCoinbase((err, res)=>console.log(res))
- v1.0.0: web3.eth.getCoinbase().then(console.log)

账户查询

- 同步: web3.eth.accounts
- 异步: web3.eth.getAccounts((err, res)=>console.log(res))
- v1.0.0: web3.eth.getAccounts().then(console.log)

区块相关

区块高度查询

- 同步: web3.eth.blockNumber
- 异步: web3.eth.getBlockNumber(*callback*)

gasPrice 查询

- 同步: web3.eth.gasPrice
- 异步: web3.eth.getGasPrice(*callback*)



区块相关

区块查询

- 同步: `web3.eth.getBlockNumber(hashStringOrBlockNumber [, returnTransactionObjects])`
- 异步: `web3.eth.getBlockNumber(hashStringOrBlockNumber, callback)`

块中交易数量查询

- 同步:
`web3.eth.getBlockTransactionCount(hashStringOrBlockNumber)`
- 异步:
`web3.eth.getBlockTransactionCount(hashStringOrBlockNumber, callback)`



交易相关

余额查询

- 同步: web3.eth.getBalance(*addressHexString* [, *defaultBlock*])
- 异步: web3.eth.getBalance(*addressHexString* [, *defaultBlock*]
[, *callback*])

交易查询

- 同步: web3.eth.getTransaction(*transactionHash*)
- 异步: web3.eth.getTransaction(*transactionHash* [, *callback*])



交易执行相关

- 交易收据查询（已进块）
 - 同步：web3.eth.getTransactionReceipt(*hashString*)
 - 异步：web3.eth.getTransactionReceipt(*hashString* [, *callback*])
- 估计 gas 消耗量
 - 同步：web3.eth.estimateGas(*callObject*)
 - 异步：web3.eth.estimateGas(*callObject* [, *callback*])



发送交易

- `web3.eth.sendTransaction(transactionObject [, callback])`
- 交易对象：
 - `from`: 发送地址
 - `to`: 接收地址，如果是创建合约交易，可不填
 - `value`: 交易金额，以wei为单位，可选
 - `gas`: 交易消耗 gas 上限，可选
 - `gasPrice`: 交易 gas 单价，可选
 - `data`: 交易携带的字串数据，可选
 - `nonce`: 整数 nonce 值，可选



消息调用

- `web3.eth.call(callObject [, defaultBlock] [, callback])`

• 参数：

- 调用对象：与交易对象相同，只是from也是可选的
 - 默认区块：默认 “latest” ， 可以传入指定的区块高度
 - 回调函数，如果没有则为同步调用



日志过滤（事件监听）

- web3.eth.filter(*filterOptions* [, *callback*])

// *filterString* 可以是 'latest' or 'pending'

```
var filter = web3.eth.filter(filterString);
```

// 或者可以填入一个日志过滤 options

```
var filter = web3.eth.filter(options);
```

// 监听日志变化

```
filter.watch(function(error, result){ if (!error) console.log(result); });
```

// 还可以用传入回调函数的方法，立刻开始监听日志

```
web3.eth.filter(options, function(error, result){
```

```
    if (!error) console.log(result);
```

```
});
```



合约相关 —— 创建合约

- web3.eth.contract

```
var MyContract = web3.eth.contract(abiArray);  
// 通过地址初始化合约实例  
  
var contractInstance = MyContract.at(address);  
// 或者部署一个新合约  
  
var contractInstance = MyContract.new([constructorParam1]  
[, constructorParam2], {data: '0x12345...', from:  
myAccount, gas: 1000000});
```



调用合约函数

- 可以通过已创建的合约实例，直接调用合约函数

// 直接调用，自动按函数类型决定用 sendTransaction 还是 call

```
myContractInstance.myMethod(param1 [, param2, ...] [,  
    transactionObject] [, defaultBlock] [, callback]);
```

// 显式以消息调用形式 call 该函数

```
myContractInstance.myMethod.call(param1 [, param2, ...] [,  
    transactionObject] [, defaultBlock] [, callback]);
```

// 显式以发送交易形式调用该函数

```
myContractInstance.myMethod.sendTransaction(param1 [,  
    param2, ...] [, transactionObject] [, callback]);
```



监听合约事件

- 合约的 event 类似于 filter，可以设置过滤选项来监听

```
var event = myContractInstance.MyEvent({valueA: 23}  
[, additionalFilterObject])  
  
// 监听事件  
  
event.watch(function(error, result){ if (!error) console.log(result); });  
  
//还可以用传入回调函数的方法，立刻开始监听事件  
  
var event = myContractInstance.MyEvent([{valueA: 23}]  
[, additionalFilterObject] , function(error, result){  
  
    if (!error) console.log(result);  
  
})  
);
```



Q&A



尚硅谷



简单投票 DApp

接下来我们要开始真正做一个 DApp，尽管它这是很简单的一个投票应用，但会包含完整的工作流程和交互页面。构建这个应用的主要步骤如下：

1. 我们首先安装一个叫做 `ganache` 的模拟区块链，能够让我们的程序在开发环境中运行。
2. 写一个合约并部署到 `ganache` 上。
3. 然后我们会通过命令行和网页与 `ganache` 进行交互。



我们与区块链进行通信的方式是通过 RPC (Remote Procedure Call)。
`web3js` 是一个 JavaScript 库，它抽象出了所有的 RPC 调用，以便于你可以通过 JavaScript 与区块链进行交互。另一个好处是，`web3js` 能够让你使用你最喜欢的 JavaScript 框架构建非常棒的 web 应用。

开发准备-Linux

下面是基于 Linux 的安装指南。这要求我们预先安装 `nodejs` 和 `npm`，再用 `npm` 安装 `ganache-cli`、`web3` 和 `solc`，就可以继续项目的下一步了。

```
mkdir simple_voting_dapp
cd simple_voting_dapp
npm init
npm install ganache-cli web3@0.20.1 solc
node_modules/.bin/ganache-cli
```

如果安装成功，运行命令 `node_modules/.bin/ganache-cli`，应该能够看到下图所示的输出。

¹ 更多 Java -大数据 -前端 -python 人工智能 -区块链资料下载，可访问百度：尚硅谷官网



```
Ganache CLI v6.0.3 (ganache-core: 2.0.2)

Available Accounts
=====
(0) 0x5c252a0c0475f9711b56ab160a1999729eccce97
(1) 0x353d310bed379b2d1df3b727645e200997016ba3
(2) 0xa3ddc09b5e49d654a43e161cae3f865261cabd23
(3) 0xa8a188c6d97ec8cf905cc1dd1cd318e887249ec5
(4) 0xc0aa5f8b79db71335dacc7cd116f357d7ecd2798
(5) 0xda695959ff85f0581ca924e549567390a0034058
(6) 0xd4ee63452555a87048dcfe2a039208d113323790
(7) 0xc60c8a7b752d38e35e0359e25a2e0f6692b10d14
(8) 0xba7ec95286334e8634e89760fab8d2ec1226bf42
(9) 0x208e02303fe29be3698732e92ca32b88d80a2d36

Private Keys
=====
(0) a6de9563d3db157ed9926a993559dc177be74a23fd88ff5776ff0505d21fed2b
(1) 17f71d31360fbafbc90cad906723430e9694daed3c24e1e9e186b4e3ccf4d603
(2) ad2b90ce116945c11eaf081f60976d5d1d52f721e659887fcebc5c81ee6ce99
(3) 68e2288df55cbc3a13a2953508c8e0457e1e71cd8ae62f0c78c3a5c929f35430
(4) 9753b05bd606e2ffc65a190420524f2efc8b16edb8489e734a607f589f0b67a8
(5) 6e8e8c468cf75fd4de0406a1a32819036b9fa64163e8be5bb6f7914ac71251cc
(6) c287c82e2040d271b9a4e071190715d40c0b861eb248d5a671874f3ca6d978a9
(7) cec41ef9ccf6cb3007c759bf3fce8ca485239af1092065aa52b703fd04803c9d
(8) c890580206f0bbea67542246d09ab4bef7eeaa22c3448dc7253ac2414a5362a
(9) eb8841a5ae34ff3f4248586e73fc8274a7f5dd2dc07b352d2c4b71132b3c73f0

HD Wallet
=====
Mnemonic: cancel better shock lady capable main crunch alcohol derive alarm duck umb
Base HD Path: m/44'/60'/0'/0/{account_index}

Listening on localhost:8545
```

为了便于测试，ganache 默认会创建 10 个账户，每个账户有 100 个以太。。。你需要用其中一个账户创建交易，发送、接收以太。

当然，你也可以安装 GUI 版本的 ganache 而不是命令行版本，在这里下载 GUI 版本：<http://truffleframework.com/ganache/>

Solidity 合约

我们会写一个叫做 Voting 的合约，这个合约有以下内容：

- 一个构造函数，用来初始化一些候选者。
- 一个用来投票的方法（对投票数加 1）



- 一个返回候选者所获得的总票数的方法

当你把合约部署到区块链的时候，就会调用构造函数，并只调用一次。与 web 世界里每次部署代码都会覆盖旧代码不同，在区块链上部署的合约是不可改变的，也就是说，如果你更新合约并再次部署，旧的合约仍然会在区块链上存在，并且数据仍在。新的部署将会创建合约的一个新的实例。

代码和解释

```
pragma solidity ^0.4.22;

contract Voting {
    mapping (bytes32 => uint8) public votesReceived;
    bytes32[] public candidateList;
    constructor(bytes32[] candidateNames) public {
        candidateList = candidateNames;
    }
    function totalVotesFor(bytes32 candidate) view public
    returns (uint8) {
        require(validCandidate(candidate));
        return votesReceived[candidate];
    }
    function voteForCandidate(bytes32 candidate) public {
        require(validCandidate(candidate));
        votesReceived[candidate] += 1;
    }
    function validCandidate(bytes32 candidate) view public
    returns (bool) {
        for(uint i = 0; i < candidateList.length; i++) {
            if (candidateList[i] == candidate) {
                return true;
            }
        }
    }
}
```



```
        }
    }
    return false;
}
}
```

Line 1. 我们必须指定代码将会哪个版本的编译器进行编译

Line 3. `mapping` 相当于一个关联数组或者是字典，是一个键值对。`mapping votesReceived` 的键是候选者的名字，类型为 `bytes32`。`mapping` 的值是一个未赋值的整型，存储的是投票数。

Line 4. 在很多编程语言中（例如 `java`、`python` 中的字典`<HashTable` 继承自字典`>`），仅仅通过 `votesReceived.keys` 就可以获取所有的候选者姓名。但是，但是在 `solidity` 中没有这样的方法，所以我们必须单独管理一个候选者数组 `candidateList`。

Line 14. 注意到 `votesReceived[key]` 有一个默认值 `0`，所以你不需要将其初始化为 `0`，直接加 `1` 即可。

你也会注意到每个函数有个可见性说明符（`visibility specifier`）（比如本例中的 `public`）。这意味着，函数可以从合约外调用。如果你不想要其他任何人调用这个函数，你可以把它设置为私有（`private`）函数。如果你不指定可见性，编译器会抛出一个警告。最近 `solidity` 编译器进行了一些改进，如果用户忘记了对私有函数进行标记导致了外部可以调用私有函数，编译器会捕获这个问题。

你也会在一些函数上看到一个修饰符 `view`。它通常用来告诉编译器函数是只读的（也就是说，调用该函数，区块链状态并不会更新）。

接下来，我们将会使用上一节安装的 `solc` 库来编译代码。如果你还记得的话，之前我们提到过 `web3js` 是一个库，它能够让你通过 `RPC` 与区块链进行交互。我们将会在 `node` 控制台里用这个库部署合约，并与区块链进行交互。



编译合约

```
In the node console> Web3 = require('web3')
> web3 = new Web3(new
Web3.providers.HttpProvider("http://localhost:8545"));
> web3.eth.accounts
[ '0x5c252a0c0475f9711b56ab160a1999729eccce97'
'0x353d310bed379b2d1df3b727645e200997016ba3' ]
> code = fs.readFileSync('Voting.sol').toString()
> solc = require('solc')
> compiledCode = solc.compile(code)
```

首先，在终端中运行 `node` 进入 `node` 控制台，初始化 `web3` 对象，并向区块链查询获取所有的账户。

确保与此同时 `ganache` 已经在另一个窗口中运行

为了编译合约，先从 `Voting.sol` 中加载代码并绑定到一个 `string` 类型的变量，然后像右边这样对合约进行编译。

当你成功地编译好合约，打印 `compiledCode` 对象（直接在 `node` 控制台输入 `compiledCode` 就可以看到内容），你会注意到有两个重要的字段，它们很重要，你必须要理解：

1. `compiledCode.contracts[':Voting'].bytecode`: 这就是 `Voting.sol` 编译好后的字节码。也是要部署到区块链上的代码。
2. `compiledCode.contracts[':Voting'].interface`: 这是一个合约的接口或者说模板（叫做 `abi` 定义），它告诉了用户在这个合约里有哪些方法。在未来无论何时你想要跟任意一个合约进行交互，你都会需要这个 `abi` 定义。你可以在[这里](#) 看到 `ABI` 的更多内容。

在以后的项目中，我们将会使用 `truffle` 框架来管理编译和与区块链的交互。但是，在使用任何框架之前，深入了解它的工作方式还是大有裨益的，因为框架会将这些内容抽象出去。



部署合约

让我们继续课程，现在将合约部署到区块链上。为此，你必须先通过传入 `abi` 定义来创建一个合约对象 `VotingContract`。然后用这个对象在链上部署并初始化合约。

```
Execute this in your node console:  
> abiDefinition =  
JSON.parse(compiledCode.contracts[':Voting'].interface)  
> VotingContract = web3.eth.contract(abiDefinition)  
> byteCode = compiledCode.contracts[':Voting'].bytecode  
> deployedContract =  
VotingContract.new(['Alice','Bob','Cary'],{data: byteCode, from:  
web3.eth.accounts[0], gas: 4700000})  
> deployedContract.address  
'0x0396d2b97871144f75ba9a9c8ae12bf6c019f610'  
// Your address will be different  
> contractInstance = VotingContract.at(deployedContract.address)
```

`VotingContract.new` 将合约部署到区块链。

第一个参数是一个候选者数组，候选者们会竞争选举，这很容易理解。让我们来看一下第二个参数里面都是些什么：

1. **data:** 这是我们编译后部署到区块链上的字节码。



2. **from:** 区块链必须跟踪是谁部署了这个合约。在这种情况下，我们仅仅是
从调用 `web3.eth.accounts` 返回的第一个账户，作为部署这个合约的账
户。记住，`web3.eth.accounts` 返回一个 `ganache` 所创建 10 个测试账
号的数组。在交易之前，你必须拥有这个账户，并对其进行解锁。创建一个账
户时，你会被要求输入一个密码，这就是你用来证明你对账户所有权的东
西。在下一节，我们将会进行详细介绍。为了方便起见，`ganache` 默认
会解锁 10 个账户。
3. **gas:** 与区块链进行交互需要花费金钱。这笔钱用来付给矿工，因为他们
帮你把代码包含了在区块链里面。你必须指定你愿意花费多少钱让你的代
码包含在区块链中，也就是设定“gas”的值。你的“from”账户里面的
ETH 余额将被用来购买 gas。gas 的价格由网络设定。

我们已经部署了合约，并有了一个合约实例（变量 `contractInstance`），我
们可以用这个实例与合约进行交互。

在区块链上有上千个合约。那么，如何识别你的合约已经上链了呢？

答案是找到已部署合约的地址：`deployedContract.address`. 当你需要跟合
约进行交互时，就需要这个部署地址和我们之前谈到的 `abi` 定义。

控制台交互

```
In your node console:  
> contractInstance.totalVotesFor.call('Rama')  
{ [String: '0'] s: 1, e: 0, c: [ 0 ] }  
> contractInstance.voteForCandidate('Rama', {from:  
web3.eth.accounts[0]})
```



```
'0xdedc7ae544c3dde74ab5a0b07422c5a51b5240603d31074f5b75c0ebc78  
6bf53'  
> contractInstance.voteForCandidate('Rama', {from:  
web3.eth.accounts[0]})  
'0x02c054d238038d68b65d55770fabfca592a5cf6590229ab91bbe7cd72da  
46de9'  
> contractInstance.voteForCandidate('Rama', {from:  
web3.eth.accounts[0]})  
'0x3da069a09577514f2baaa11bc3015a16edf26aad28dffbcd126bde2e71f  
2b76f'  
>  
contractInstance.totalVotesFor.call('Rama').toLocaleString()'3  
'
```

{ [String: '0'] s: 1, e: 0, c: [0] } 是数字 0 的科学计数法表示。这里返回的值是一个 bigNumber 对象，可以用它的的.toNumber()方法来显示数字：

```
contractInstance.totalVotesFor.call('Alice').toNumber()  
web3.fromWei(web3.eth.getBalance(web3.eth.accounts[1]).toNumber(),'ether'  
)
```

BigNumber 的值以符号，指数和系数的形式,以十进制浮点格式进行存储。

s 是 sign 符号，也就是正负；

e 是 exponent 指数，表示最高位后有几个零；

c 是 coefficient 系数，也就是实际的有效数字； bignumber 构造函数的入参位数限制为 14 位，所以系数表示是从后向前截取的一个数组，14 位截取一次。

为候选者投票并查看投票数

继续课程，在你的 node 控制台里调用 voteForCandidate 和 totalVotesFor 方法并查看结果。



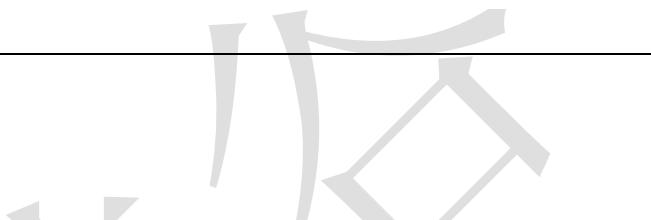
每为一位候选者投一次票，你就会得到一个交易 id:

比如：

‘0xdedc7ae544c3dde74ab5a0b07422c5a51b5240603d31074f5b75c0ebc786bf53’。这个交易 id 就是交易发生的凭据，你可以在将来的任何时候引用这笔交易。这笔交易是不可改变的。

对于以太坊这样的区块链，不可改变是其主要特性之一。在接下来的章节，我们将会利用这一特性构建应用。

网页交互



至此，大部分的工作都已完成，我们还需要做的事情就是创建一个简单的 html，里面有候选者姓名并调用投票命令（我们已经在 nodejs 控制台里试过）。你可以在右侧找到 html 代码和 js 代码。将它们放到 chapter1 目录，并在浏览器中打开 index.html。

index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Voting DApp</title>
  <link
    href='https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css' rel='stylesheet' type='text/css'>
</head>
<body class="container">
  <h1>A Simple Voting Application</h1>
  <div class="table-responsive">
```



```
<table class="table table-bordered">
  <thead>
    <tr>
      <th>Candidate</th>
      <th>Votes</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Alice</td>
      <td id="candidate-1"></td>
    </tr>
    <tr>
      <td>Bob</td>
      <td id="candidate-2"></td>
    </tr>
    <tr>
      <td>Cary</td>
      <td id="candidate-3"></td>
    </tr>
  </tbody>
</table>
</div>
<input type="text" id="candidate" />
<a href="#" onclick="voteForCandidate()" class="btn btn-primary">Vote</a>
</body>
<script
  src="https://cdn.jsdelivr.net/gh/ethereum/web3.js/dist/web3.min.js">
</script>
```



```
<script src="https://code.jquery.com/jquery-3.1.1.slim.min.js">
</script>
<script src="./index.js"></script>
</html>
```

Tips:

1. <head>中用 link 形式引入 bootstrap 的 css 类型库，以下 container、table-responsive 等 class 均来自 bootstrap
2. <th>表头单元格，<td>表单元格，候选人名字后的单元格为得票数，用 id 区分以方便写入，之后 js 中写死了对应关系
3. <input>一个输入框，定义 id 方便在 js 中取值
4. <a>超链接形式的按键 btn， href="#" 为跳转至本页，即不跳转； onclick 指向 js 中方法

为了简化项目，我们已经硬编码了候选者姓名。如果你喜欢的话，可以调整代码使其动态选择候选者。

index.js

```
web3 = new Web3(new
Web3.providers.HttpProvider("http://localhost:8545"));
abi = JSON.parse('[{"constant":false,...}]')
VotingContract = web3.eth.contract(abi);
contractInstance =
VotingContract.at('0x329f5c190380ebcf640a90d06eb1db2d68503a53'
);
candidates = {"Alice": "candidate-1", "Bob": "candidate-2",
"Cary": "candidate-3"};
```



```
function voteForCandidate(candidate) {
    candidateName = $("#candidate").val();
    try {
        contractInstance.voteForCandidate(candidateName,
            {from: web3.eth.accounts[0]},
            function() {
                let div_id = candidates[candidateName];
                $("#" + div_id).html(
                    contractInstance.totalVotesFor
                        .call(candidateName)
                        .toString());
            }
        );
    } catch (err) {
    }
}
$(document).ready(function() {
    candidateNames = Object.keys(candidates);
    for (var i = 0; i < candidateNames.length; i++) {
        let name = candidateNames[i];
        let val = contractInstance.totalVotesFor
            .call(name).toString()
        $("#" + candidates[name]).html(val);
    }
});
```

在第 4 行，用你自己的合约地址替换代码中的合约地址。合约地址是之前的 `deployedContract.address`

12

更多 Java -大数据 -前端 -python 人工智能 -区块链资料下载，可访问百度：尚硅谷官网



如果一切顺利的话，你应该能够在文本框中输入候选者姓名，然后投票数应该加 1。

注意：由于网络原因，`web3.js` 可能无法获取，可自行下载到本地导入。

如果你可以看到页面，为候选者投票，然后看到投票数增加，那就已经成功创建了第一个合约，恭喜！所有投票都会保存到区块链上，并且是不可改变的。任何人都可以独立验证每个候选者获得了多少投票。当然，我们所有的事情都是在一个模拟的区块链上（`ganache`）完成，在接下来的课程中，我们会将这个合约部署到真正的公链上。在 Part 2，我们会把合约部署到叫做 `Ropsten testnet` 的公链，同时也会学习如何使用 `truffle` 框架构建合约，管理 `dapp`。

总结一下，下面是你到目前为止已经完成的事情：

1. 通过安装 `node`, `npm` 和 `ganache`, 你已经配置好了开发环境。
2. 你编码了一个简单的投票合约，编译并部署到区块链上。
3. 你通过 `nodejs` 控制台与网页与合约进行了交互。



编写合约的编译脚本

之前的课程中，我们已经熟悉了智能合约的编译。编译是对合约进行部署和测试的前置步骤，编译步骤的目标是把源代码转成 **ABI** 和 **Bytecode**，并且能够处理编译时抛出的错误，确保不会在包含错误的源代码上进行编译。

开始我们的编译方式是用 **solc** 工具做命令行编译，这个过程中牵涉到大段内容的复制粘贴，很容易出错；之后在项目中引入 **solc** 模块，可以在 **node** 命令行中自动编译并读取结果内容。于是我们自然会想到，能不能将这个过程写成脚本，自动完成这些过程呢？这节课我们就来完成这个任务。

目录结构

首先新建一个项目目录，可以叫做 **contract_workflow**。

```
mkdir contract_workflow  
cd contract_workflow
```

为了存放不同目的不同类型的文件，我们先在项目根目录下新建 4 个子目录：

```
mkdir contracts  
mkdir scripts  
mkdir compiled  
mkdir tests
```

其中 **contracts** 目录存放合约源代码，**scripts** 目录存放编译脚本，**compiled** 目录存放编译结果，**tests** 目录存放测试文件。

准备合约源码

为了简化工作，我们可以直接复制以前的 **solidity** 代码，也可以自己写一个简单的合约。比如，这里用到了我们最初写的简单合约 **Car.sol**：

¹

更多 Java -大数据 -前端 -python 人工智能 -区块链资料下载，可访问百度：尚硅谷官网



```
pragma solidity ^0.4.22;
contract Car {
    string public brand;
    constructor(string initialBrand) public {
        brand = initialBrand;
    }
    function setBrand(string newBrand) public {
        brand = newBrand;
    }
}
```

将它放到 `contracts` 目录下。

准备编译工具

我们用 `solc` 作为编译的基础工具。用 `npm` 将 `solc` 安装到本地目录中：

```
npm install solc
```

开发编译脚本

我们已经熟悉了命令行编译的流程，现在我们试图将它脚本中。在 `scripts` 目录下新建文件 `compile.js`

```
const fs = require('fs');
const path = require('path');
const solc = require('solc');
const contractPath = path.resolve(__dirname, '../contracts',
'Car.sol');
const contractSource = fs.readFileSync(contractPath, 'utf8');
const result = solc.compile(contractSource, 1);
```



```
console.log(result);
```

我们把合约源码从文件中读出来，然后传给 `solc` 编译器，等待同步编译完成之后，把编译结果输出到控制台。

其中 `solc.compile()` 的第二个参数给 `1`，表示启用 `solc` 的编译优化器。

编译结果是一个嵌套的 `js` 对象，其中可以看到 `contracts` 属性包含了所有找到的合约（当然，我们的源码中只有一个 `Car`）。每个合约下面包含了 `assembly`、`bytecode`、`interface`、`metadata`、`opcodes` 等字段，我们最关心的当然是这两个：

- `bytecode`: 字节码，部署合约到以太坊区块链上时需要使用；
- `interface`: 二进制应用接口（ABI），使用 `web3` 初始化智能合约交互实例的时候需要使用。

其中 `interface` 是被 `JSON.stringify` 过的字符串，我们用 `JSON.parse` 反解出来并格式化，就可以拿到合约的 `abi` 对象。

保存编译结果

让我们继续课程，现在将合约部署到区块链上。为此，你必须先通过传入 `abi` 定义来创建一个合约对象 `VotingContract`。然后用这个对象在链上部署并初始化合约。为了方便后续的部署和测试过程直接使用编译结果，需要把编译结果保存到文件系统中，在做改动之前，我们引入一个非常好用的小工具 `fs-extra`，在脚本中使用 `fs-extra` 直接替换到 `fs`，然后在脚本中加入以下代码：

```
Object.keys(result.contracts).forEach( name => {
    const contractName = name.replace(/:/, '');
    const filePath = path.resolve(__dirname, '../compiled',
        `${contractName}.json`);
    fs.writeFileSync(filePath, result.contracts[name]);
    console.log(`save compiled contract ${contractName} to
        ${filePath}`);
}
```



```
});
```

然后重新运行编译脚本，确保 `compiled` 目录下包含了新生成的 `Car.json`。

类似于前端构建流程中的编译步骤，我们编译前通常需要把之前的结果清空，然后把最新的编译结果保存下来，这对保障一致性非常重要。所以继续对编译脚本做如下改动：

在脚本执行的开始加入清除编译结果的代码：

```
// cleanup
const compiledDir = path.resolve(__dirname, '../compiled');
fs.removeSync(compiledDir);
fs.ensureDirSync(compiledDir);
```

这里专门定义了 `compiledDir`，所以后面的 `filePath` 也可以改为：

```
const filePath =
  path.resolve(compiledDir, `${contractName}.json`);
```

新增的 `cleanup` 代码段的作用就是准备全新的目录，修改完之后，需要重新运行编译脚本，确保一切正常。

处理编译错误

现在的编译脚本只处理了最常见的情况，即 `Solidity` 源代码没问题，这个假设其实是不成立的。如果源代码有问题，我们在编译阶段就应该报出来，而不应该把错误的结果写入到文件系统，因为这样会导致后续步骤失败。为了搞清楚编译器 `solc` 遇到错误时的行为，我们人为在源代码中引入错误（例如把 `function` 关键字写成 `functio`），看看脚本的表现如何。

重新运行编译脚本，发现它并没有报错，而是把错误作为输出内容打印出来，其中错误的可读性比较差。



所以我们要对编译脚本稍作改动，在编译完成之后就检查 error，让它能够在出错时直接抛出错误：

```
// check errors  
if (Array.isArray(result.errors) && result.errors.length) {  
    throw new Error(result.errors[0]);  
}
```

重新运行编译脚本，可以看到我们得到了可读性更好的错误提示。

```
/home/ubuntu/project/contract_workflow/node_modules/solc/soljson.js:1  
(function (exports, require, module, __filename, __dirname) { var Module;if(!Module)Module=(typeof M  
wnProperty(key)){moduleOverrides[key]=Module[key]}}var ENVIRONMENT_IS_WEB=typeof window==="object";v  
==="object"&&typeof require==="function"&&!ENVIRONMENT_IS_WEB&&!ENVIRONMENT_IS_WORKER;var ENVIRONM  
DE;if(!Module["print"]){Module["print"]=function print(x){process["stdout"].write(x+"\n")};if(!Modul  
=require("fs");var nodePath=require("path");Module["read"]=function read(filename,binary){filename=r  
Error: :7:18: ParserError: Expected ';' but got '{'  
        functio setBrand(string newBrand) public {  
  
            at Object.<anonymous> (/home/ubuntu/project/contract_workflow/scripts/compile.js:17:8)  
            at Module._compile (module.js:641:30)  
            at Object.Module._extensions..js (module.js:652:10)  
            at Module.load (module.js:560:32)  
            at tryModuleLoad (module.js:503:12)  
            at Function.Module._load (module.js:495:3)  
            at Function.Module.runMain (module.js:682:10)  
            at startup (bootstrap_node.js:191:16)  
            at bootstrap_node.js:613:3
```

最终版编译脚本

编译脚本的最终版如下：

```
const fs = require('fs-extra');  
const path = require('path');  
const solc = require('solc');  
  
// cleanup  
const compiledDir = path.resolve(__dirname, '../compiled');  
fs.removeSync(compiledDir);  
fs.ensureDirSync(compiledDir);
```



```
// compile const contractPath = path.resolve(__dirname,
    '../contracts', 'Car.sol');

const contractSource = fs.readFileSync(contractPath, 'utf8');
const result = solc.compile(contractSource, 1);

// check errors
if (Array.isArray(result.errors) && result.errors.length) {
    throw new Error(result.errors[0]);
}

// save to disk
Object.keys(result.contracts).forEach(name => {
    const contractName = name.replace(/^:/, '');
    const filePath = path.resolve(compiledDir,
        `${contractName}.json`);
    fs.outputJsonSync(filePath, result.contracts[name]);
    console.log(`save compiled contract ${contractName} to
        ${filePath}`); });
}
```





编写合约的部署脚本

上节课我们已经通过编译从 `solidity` 源码得到了字节码，接下来我们会试图完成一个自动化脚本，将合约部署到区块链网络中。

部署的必要条件

与以太坊节点的通信连接

我们需要启动一个以太坊节点，连接到想要的网络，然后开放 `HTTP-RPC` 的 `API`（默认 `8545` 端口）给外部调用；或者也可以用第三方提供的可用节点入口，以太坊社区有人专门为开发者提供了节点服务。目前我们直接用 `ganache`，不需要考虑这些问题，但如果配置其它网络，这个配置就是必要的。

余额大于 0 的账户

因为以太坊上的任何交易都需要账户发起，账户中必须有足够的余额来支付手续费（`Transaction Fee`），如果余额为 0 部署会失败。当然，我们目前用的是 `ganache`，里面默认有 10 个账户，每个账户 `100ETH`，不存在这个问题，但如果要部署到其它网络（私链、测试网络、主网）就必须考虑这个问题。

安装依赖

搞清楚部署的必要条件之后，我们需要安装必要的依赖包。首先是 `web3.js`，`web3.js` 的 `1.0.0` 版本尚未发布，但是相比 `0.2x.x` 版本变化非常大，`1.x` 中大量使用了 `Promise`，可以结合 `async/await` 使用，而 `0.x` 版本只支持回调，因为使用 `async/await` 能让代码可读性更好，我们这次选择使用 `1.0.0` 版本。

```
npm install web3
```



编写部署脚本

做好准备工作之后，我们开始编写合约部署脚本，在 `scripts` 目录下新建脚本文件 `deploy.js`:

```
const path = require('path');
const Web3 = require('web3');
const web3 = new Web3(new Web3.providers
    .HttpProvider('http://localhost:8545'));
// 1. 拿到 abi 和 bytecode
const contractPath = path.resolve(__dirname,
    '../compiled/Car.json');
const { interface, bytecode } = require(contractPath);

(async () => {
    // 2. 获取钱包里面的账户
    const accounts = await web3.eth.getAccounts();
    console.log('部署合约的账户: ', accounts[0]);
    // 3. 创建合约实例并且部署
    var result = await new
        web3.eth.Contract(JSON.parse(interface))
        .deploy({ data: bytecode, arguments: ['AUDI'] })
        .send({ from: accounts[0], gas: '1000000' });
    console.log('合约部署成功: ', result);
})();
```

我们来熟悉一下 v1.0.0 版本中的部署操作。由于 1.0.0 版本中调用返回全部是 promise，所以我们这里用到了 ES7 中的 `async/await` 来处理所有异步调用。

第二步获取钱包账户，存为本地变量，然后选取 `accounts[0]` 作为部署合约的账户；我们应该确保这个账户中以太余额充足。



第三步中，我们用 `promise` 的链式调用完成了创建抽象合约对象、创建部署交易对象（`deploy`）和发送部署交易三个步骤，其中只有 `send` 一步是真正的异步请求调用。分开写就是这样：

```
const contract = new web3.eth.Contract(JSON.parse(interface));
const transaction = contract.deploy({ data: bytecode, arguments:
                                         ['AUDI'] });
const result = await transaction.send({ from: accounts[0], gas:
                                         1000000 });
```

运行脚本

在根目录下运行写好的部署脚本：

```
node scripts/deploy.js
```

查看结果，可以看到合约已经成功部署。我们发现返回结果有些复杂，所以可以对代码稍作改进，截取 `address` 返回，并计算一下部署花了多少时间：

```
const path = require('path');
const Web3 = require('web3');
const web3 = new Web3(new Web3.providers
                      .HttpProvider('http://localhost:8545'));

// 1. 拿到 bytecode
const contractPath = path.resolve(__dirname,
                                    '../compiled/Car.json');

const { interface, bytecode } = require(contractPath);

(async () => {
    // 2. 获取钱包里面的账户
    const accounts = await web3.eth.getAccounts();
    console.log('部署合约账户：', accounts[0]);
    // 3. 创建合约实例并且部署
})()
```



```
console.time('合约部署耗时');
var result = await new
    web3.eth.Contract(JSON.parse(interface))
    .deploy({ data: bytecode, arguments: [ 'AUDI' ] })
    .send({ from: accounts[0], gas: '1000000' });
console.timeEnd('合约部署耗时');
console.log('合约部署成功:', result.options.address);
})();
```





编写合约测试脚本

我们已经实现了合约的编译和部署的自动化，这将大大提升我们开发的效率。但流程的自动化并不能保证我们的代码质量。质量意识是靠谱工程师的基本职业素养，在智能合约领域也不例外：任何代码如果不做充分的测试，问题发现时通常都已为时太晚；如果代码不做自动化测试，问题发现的成本就会越来越高。

在编写合约时，我们可以利用 `remix` 部署后的页面调用合约函数，进行单元测试；还可以将合约部署到私链，用 `geth` 控制台或者 `node` 命令行进行交互测试。但这有很大的随意性，并不能形成标准化测试流程；而且手动一步步操作，比较繁琐，不易保证重复一致。

于是我们想到，是否可以利用现成的前端技术栈实现合约的自动化测试呢？当然是可以的，`mocha` 就是这样一个 `JavaScript` 测试框架。

安装依赖

开始编写测试脚本之前，我们首先需要安装依赖：测试框架 `mocha`。当然，作为对合约的测试，模拟节点 `ganache` 和 `web3` 都是不可缺少的；不过我们在上节课编写部署脚本时，已经安装了这些依赖（我们的 `web3` 依然是 1.0.0 版本）。

```
npm install mocha -save-dev
```

进行单元测试，比较重要的一点是保证测试的独立性和隔离性，所以我们并不需要测试网络这种有复杂交互的环境，甚至不需要本地私链保存测试历史。而 `ganache` 基于内存模拟以太坊节点行为，每次启动都是一个干净的空白环境，所以非常适合我们做开发时的单元测试。还记得 `ganache` 的前身叫什么吗？就是大名鼎鼎的 `testRPC`。

`mocha` 简介

¹

更多 Java -大数据 -前端 -python 人工智能 -区块链资料下载，可访问百度：尚硅谷官网



mocha 是 JavaScript 的一个单元测试框架，既可以在浏览器环境中运行，也可以在 node.js 环境下运行。我们只需要编写测试用例，mocha 会将测试自动运行并给出测试结果。

mocha 的主要特点有：

- 既可以测试简单的 JavaScript 函数，又可以测试异步代码；
- 可以自动运行所有测试，也可以只运行特定的测试；
- 可以支持 before、after、beforeEach 和 afterEach 来编写初始化代码。

测试脚本示例

假设我们编写了一个 sum.js，并且输出一个简单的求和函数：

```
module.exports = function (...rest) {  
    var sum = 0;  
    for (let n of rest) {  
        sum += n;  
    }  
    return sum;  
};
```

这个函数非常简单，就是对输入的任意参数求和并返回结果。

如果我们想对这个函数进行测试，可以写一个 test.js，然后使用 Node.js 提供的 assert 模块进行断言：

```
const assert = require('assert');  
const sum = require('./sum');  
  
assert.strictEqual(sum(), 0);  
assert.strictEqual(sum(1), 1);  
assert.strictEqual(sum(1, 2), 3);  
assert.strictEqual(sum(1, 2, 3), 6);
```



`assert` 模块非常简单，它断言一个表达式为 `true`。如果断言失败，就抛出 `Error`。

单独写一个 `test.js` 的缺点是没法自动运行测试，而且，如果第一个 `assert` 报错，后面的测试也执行不了了。

如果有很多测试需要运行，就必须把这些测试全部组织起来，然后统一执行，并且得到执行结果。这就是我们为什么要用 `mocha` 来编写并运行测试。

我们利用 `mocha` 修改后的测试脚本如下：

```
const assert = require('assert');
const sum = require('../sum');

describe('#sum.js', () => {

    describe('#sum()', () => {
        it('sum() should return 0', () => {
            assert.strictEqual(sum(), 0);
        });

        it('sum(1) should return 1', () => {
            assert.strictEqual(sum(1), 1);
        });

        it('sum(1, 2) should return 3', () => {
            assert.strictEqual(sum(1, 2), 3);
        });

        it('sum(1, 2, 3) should return 6', () => {
            assert.strictEqual(sum(1, 2, 3), 6);
        });
    });
});
```



```
});
```

这里我们使用 mocha 默认的 BDD-style 的测试。describe 可以任意嵌套，以便把相关测试看成一组测试。

describe 可以任意嵌套，以便把相关测试看成一组测试；而其中的每个 it 就代表一个测试。

每个 it("name", function() {...}) 就代表一个测试。例如，为了测试 sum(1, 2)，我们这样写：

```
it('sum(1, 2) should return 3', () => {
  assert.strictEqual(sum(1, 2), 3);
});
```

编写测试的原则是，一次只测一种情况，且测试代码要非常简单。我们编写多个测试来分别测试不同的输入，并使用 assert 判断输出是否是我们所期望的。

运行测试脚本

下一步，我们就可以用 mocha 运行测试了。打开命令提示符，切换到项目目录，然后创建文件夹 test，将 test.js 放入 test 文件夹下，执行命令：

```
./node_modules/mocha/bin/mocha
```

mocha 就会自动执行 test 文件夹下所有测试，然后输出如下：

```
#sum.js
#sum()
  ✓ sum() should return 0
  ✓ sum(1) should return 1
  ✓ sum(1, 2) should return 3
  ✓ sum(1, 2, 3) should return 6
```



```
4 passing (7ms)
```

这说明我们编写的 4 个测试全部通过。如果没有通过，要么修改测试代码，要么修改 `hello.js`，直到测试全部通过为止。

编写合约测试脚本

测试时我们通常会把每次测试运行的环境隔离开，以保证互不影响。对应到合约测试，我们每次测试都需要部署新的合约实例，然后针对新的实例做功能测试。`Car` 合约的功能比较简单，我们只要设计 2 个测试用例：

- 合约部署时传入的 `brand` 属性被正确存储；
- 调用 `setBrand` 之后合约的 `brand` 属性被正确更新；

新建测试文件 `tests/car.spec.js`，完整的测试代码如下。

```
const path = require('path');
const assert = require('assert');
const ganache = require('ganache-cli');
const Web3 = require('web3');

// 1. 配置 provider
const web3 = new Web3(ganache.provider());

// 2. 拿到 abi 和 bytecode
const contractPath = path.resolve(__dirname,
    '../compiled/Car.json');
const { interface, bytecode } = require(contractPath);

let accounts;
let contract;
const initialBrand = 'BMW';
```



```
describe('contract', () => {
    // 3. 每次跑单测时需要部署全新的合约实例，起到隔离的作用
    beforeEach(async () => {
        accounts = await web3.eth.getAccounts();
        console.log('合约部署账户: ', accounts[0]);
        contract = await new
            web3.eth.Contract(JSON.parse(interface))
            .deploy({ data: bytecode, arguments: [initialBrand] })
            .send({ from: accounts[0], gas: '1000000' });
        console.log('合约部署成功: ',
            contract.options.address); });

    // 4. 编写单元测试
    it('deployed contract', () => {
        assert.ok(contract.options.address);
    });

    it('should has initial brand', async () => {
        const brand = await contract.methods.brand().call();
        assert.equal(brand, initialBrand);
    });

    it('can change the brand', async ()=>{
        const newBrand = 'Benz';
        await contract.methods.setBrand(newBrand)
            .send({from: accounts[0]});

        const brand = await contract.methods.brand().call();
        assert.equal(brand, newBrand);
    });
});
```



整个测试代码使用的断言库是 Node.js 内置的 `assert` 模块，`assert.ok()` 用于判断表达式真值，等同于 `assert()`，如果为 `false` 则抛出 `error`；`assert.equal()` 用于判断实际值和期望值是否相等（`==`），如果不相等则抛出 `error`。

`beforeEach` 是 mocha 里提供的声明周期方法，表示每次运行时每个 `test` 执行前都要做的准备操作。因为我们知道，在测试前初始化资源，测试后释放资源是非常常见的，所以 mocha 提供了 `before`、`after`、`beforeEach` 和 `afterEach` 来实现这些功能。

测试的关键步骤也用编号的数字做了注释，其中步骤 1、2、3 在合约部署脚本中已经比较熟悉，需要注意的是 `ganache-cli provider` 的创建方式。我们在脚本中引入 `ganache`，将模拟以太坊节点嵌入测试中，就不会影响我们外部运行的节点环境了。

测试中我们用到了 `web3.js` 中两个与合约实例交互的方法，之前我们已经接触过，以后在 DApp 开发时会大量使用：

- `contract.methods.brand().call()`，调用合约上的方法，通常是取数据，立即返回，与 v0.20.1 版本中的 `.call()` 相同；
- `contract.methods.setBrand('xxx').send()`，对合约发起交易，通常是修改数据，返回的是交易 Hash，相当于 v0.20.1 中的 `sendTransaction()`；
`send` 必须指定发起的账户地址，而 `call` 可以直接调用。

注意在 v1.0.0 中，`contract` 后面要加上`.methods` 然后才能跟合约函数名，这与 v0.20.1 不同；类似，v1.0.0 中事件的监听也要 `contract` 后面加`.events`。

运行测试脚本

有了测试代码，就可以运行并观察结果。mocha 默认会执行 `test` 目录下的所有脚本，但我们也一样可以传入脚本路径，指定执行目录。如果你环境中全局安装了 mocha，可以使用如下命令运行测试：

```
mocha tests
```

如果没有全局安装 mocha，就使用如下命令运行测试：



编写合约测试脚本

```
./node_modules/.bin/mocha tests
```

如果一切正常，我们可以看到这样的输出结果：

```
contract
合约部署账户: 0x1dD5C293Daf399Df299A7896Ce618142cAd0378f
(node:354) MaxListenersExceededWarning: Possible EventEmitter memory
emitters added. Use emitter.setMaxListeners() to increase limit
合约部署成功: 0x4c3a244d8529927aD44c8707b302B30671DB2473
    ✓ deployed contract
合约部署账户: 0x1dD5C293Daf399Df299A7896Ce618142cAd0378f
合约部署成功: 0x222246dF0990178391c9B0CC4cF2D86E34E23B42
    ✓ should has initial brand
合约部署账户: 0x1dD5C293Daf399Df299A7896Ce618142cAd0378f
合约部署成功: 0xf559425569829a606123f27aa27AA1AC61B1d1ab
    ✓ can change the brand (117ms)

  3 passing (748ms)
```

完整的工作流

到目前为止，我们已经熟悉了智能合约的开发、编译、部署、测试，而在实际工作中，把这些过程串起来才能算作是真正意义上的工作流。比如修改了合约代码需要重新运行测试，但是重新运行测试之前需要重新编译，而部署的过程也是类似的，每次部署的都要是最新的合约代码。

通过 `npm script` 机制，我们可以把智能合约的工作流串起来，让能自动化的尽可能自动化，在 `package.json` 中作如下修改：

```
"scripts": {
  "compile": "node scripts/compile.js",
  "pretest": "npm run compile",
  "test": "mocha tests/",
  "predeploy": "npm run compile",
  "deploy": "node scripts/deploy.js"
},
```



编写合约测试脚本

上面的改动中，我们为项目增加了 3 条命令：compile、test、deploy，其中 pretest、predeploy 是利用了 npm script 的生命周期机制，把我们的 compile、test、deploy 串起来。

接下来我们可以使用 npm run test 运行测试，结果如下：

```
> contract_workflow@1.0.0 pretest /home/ubuntu/project/workflow_test
> npm run compile

> contract_workflow@1.0.0 compile /home/ubuntu/project/workflow_test
> node scripts/compile.js

Saving json file to /home/ubuntu/project/workflow_test/compiled/Car.json

> contract_workflow@1.0.0 test /home/ubuntu/project/workflow_test
> ./node_modules/mocha/bin/mocha tests/

contract
合约部署账户: 0xF32E39c8b69999a4305Dd878bA56e0c6E29b0ef0
(node:423) MaxListenersExceededWarning: Possible EventEmitter memory leak
nners added. Use emitter.setMaxListeners() to increase limit
合约部署成功: 0xA8bC40d15f863627D7ae21ccF384A7F6FF73e645
    ✓ deployed contract
合约部署账户: 0xF32E39c8b69999a4305Dd878bA56e0c6E29b0ef0
合约部署成功: 0x2bF2b282C02b4f59132f52f9c3BC3f43973Afff9
    ✓ should has initial brand (38ms)
合约部署账户: 0xF32E39c8b69999a4305Dd878bA56e0c6E29b0ef0
合约部署成功: 0xBa792454BBe4892d7668F8C027dC79C2d07a2836
    ✓ can change the brand (155ms)

3 passing (860ms)
```

同理我们可以使用 npm run deploy 部署合约，结果如下：

```
> contract_workflow@1.0.0 predeploy /home/ubuntu/project/workflow_test
> npm run compile

> contract_workflow@1.0.0 compile /home/ubuntu/project/workflow_test
> node scripts/compile.js

Saving json file to /home/ubuntu/project/workflow_test/compiled/Car.json

> contract_workflow@1.0.0 deploy /home/ubuntu/project/workflow_test
> node scripts/deploy.js

deploy time: 177.083ms
contract address: 0x00aC688114723873766aa7D9903750b11d31ae89
```



梅克尔-帕特里夏树

Merkel-Patricia Tree

(MPT)

2018.10

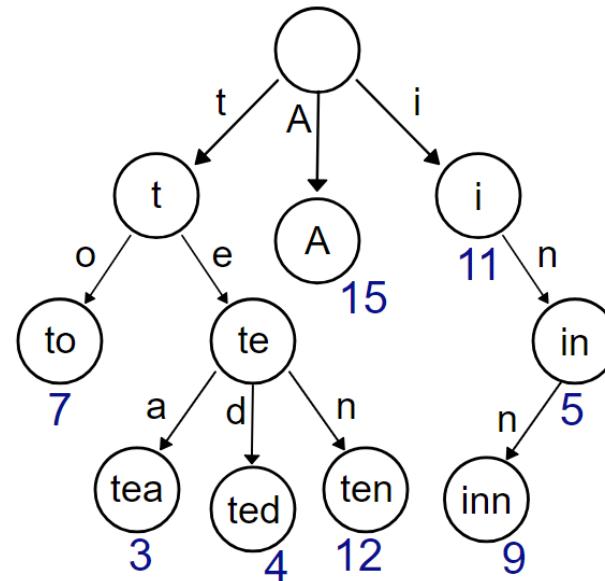


MPT是什么

- Merkle Patricia Tree (MPT)，翻译为梅克尔-帕特里夏树
- MPT 提供了一个基于密码学验证的底层数据结构，用来存储键值对 (key-value) 关系
- MPT 是完全确定性的，这是指在一颗 MPT 上一组键值对是唯一确定的，相同内容的键可以保证找到同样的值，并且有同样的根哈希 (root hash)
- MPT 的插入、查找、删除操作的时间复杂度都是 $O(\log(n))$ ，相对于其它基于复杂比较的树结构（比如红黑树），MPT 更容易理解，也更易于编码实现



从字典树 (Trie) 说起

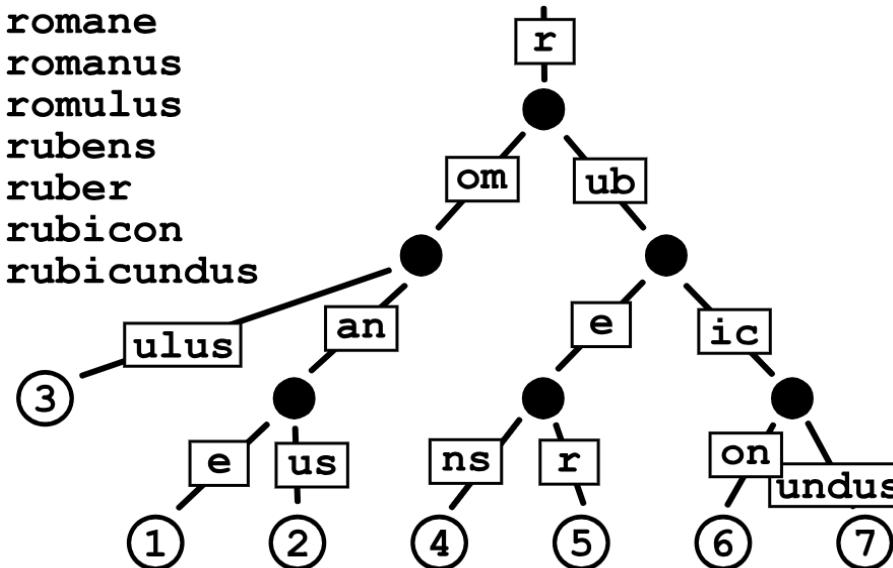


- 字典树 (Trie) 也称前缀树 (prefix tree) , 属于搜索树，是一种有序的树数据结构
- 字典树用于存储动态的集合或映射，其中的键通常是字符串

基数树 (Radix Tree)

- 基数树又叫压缩前缀树 (compact prefix tree) , 是一种空间优化后的字典树，其中如果一个节点只有唯一的子节点，那么这个子节点就会与父节点合并存储

1 romane
2 romanus
3 romulus
4 rubens
5 ruber
6 rubicon
7 rubicundus





基数树节点

在一个标准的基数树里，每个节点存储的数据如下：

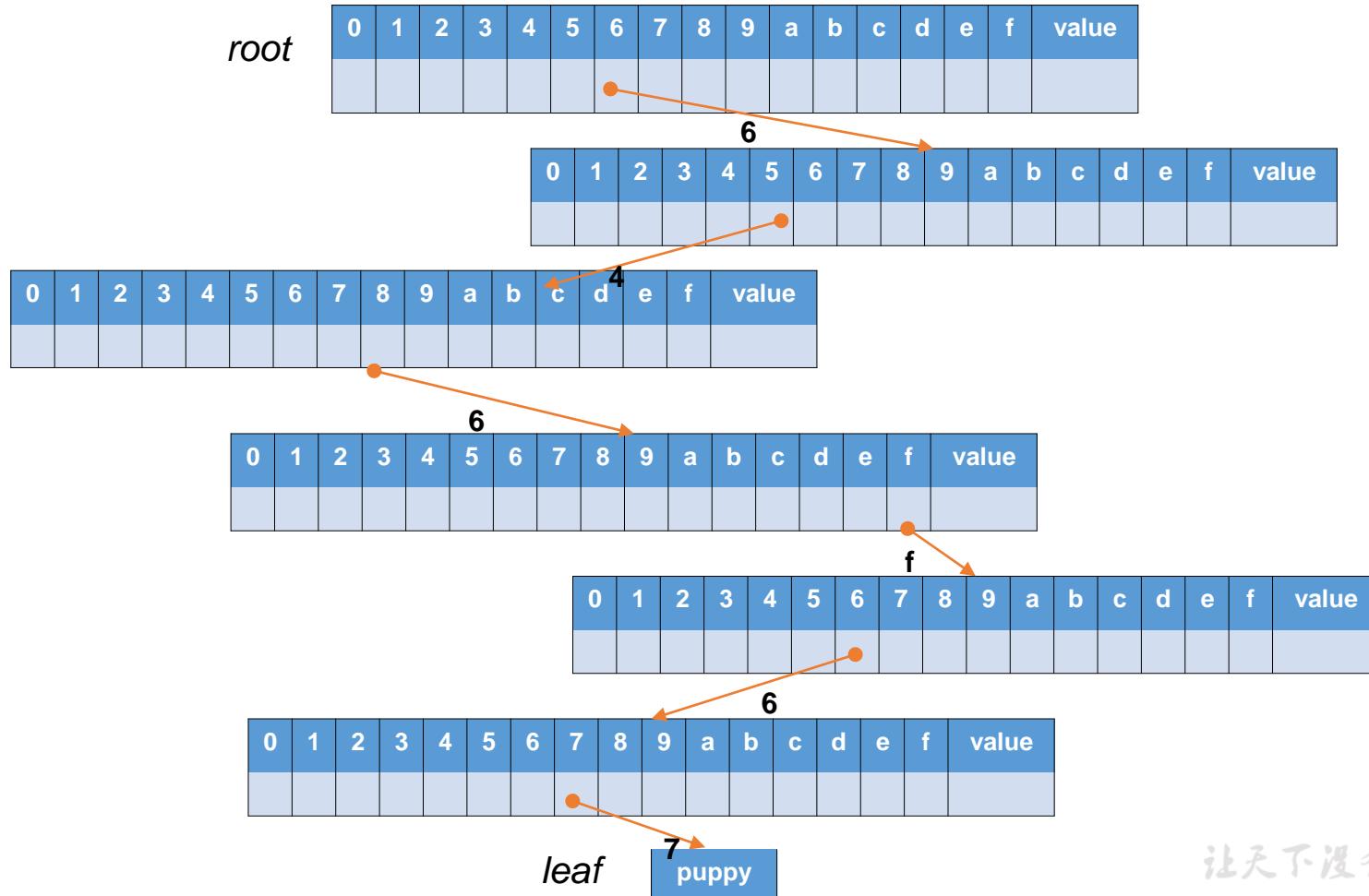
[i0, i1, ... in, value]

- 这里的 i_0, i_1, \dots, i_n 表示定义好的字母表中的字符，字母表中一共有 $n+1$ 个字符，这颗树的基数 (radix) 就是 $n+1$
- $value$ 表示这个节点中最终存储的值
- 每一个 i_0 到 i_n 的“槽位”，存储的或者是 $null$ ，或者是指向另一节点的指针
- 用节点的访问路径表示 key ，用节点的最末位置存储 $value$ ，这就实现了一个基本的键值对存储



示例

- 我们有一个键值对{ “dog”: “puppy” }，现在希望通过键 dog 访问它的值；我们采用16进制的 Hex 字符作为字符集
- 首先我们将 “dog” 转换成 ASCII 码，这样就得到了字符集中的表示 64 6f 67，这就是树结构中对应的键
- 按照键的字母序，即 6->4->6->f->6->7，构建树中的访问路径
- 从树的根节点 (root) 出发，首先读取索引值 (index) 为 6 的插槽中存储的值，以它为键访问到对应的子节点
- 然后取出子节点索引值为 4 的插槽中的值，以它为键访问下一层节点，直到访问完所需要的路径
- 最终访问到的叶子节点，就存储了我们想要查找的值，即“puppy”





基数树的问题

数据校验

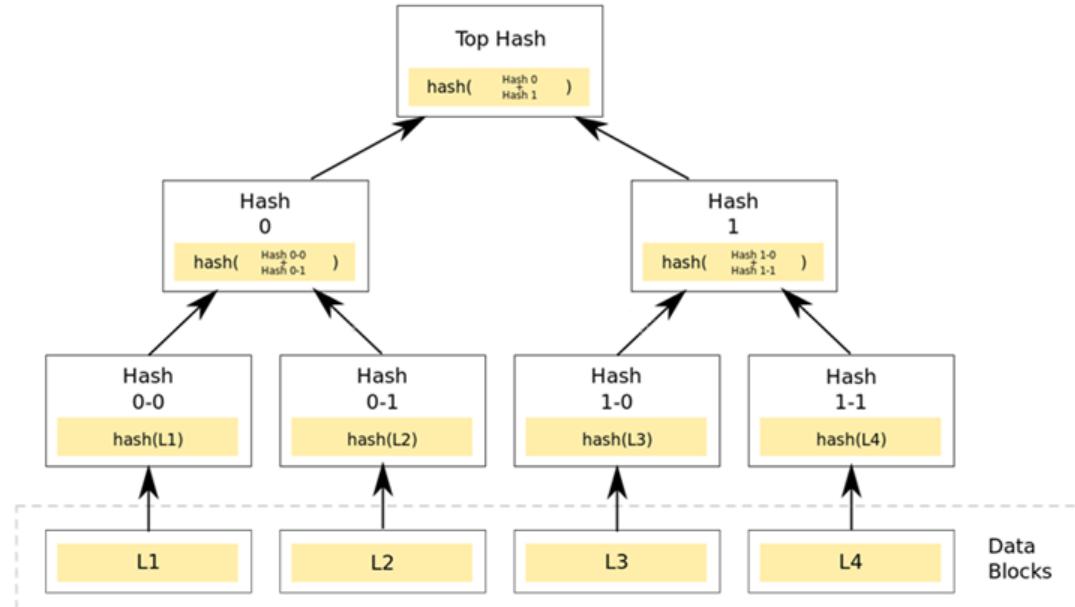
- 基数树节点之间的连接方式是指针，一般用32位或64位的内存地址作为指针的值，比如C语言就是这么做的。但这种直接存地址的方式无法提供对数据内容的校验，而这在区块链这样的分布式系统中非常重要。

访问效率

- 基数树的另一个问题是低效。如果我们只想存一个 bytes32 类型的键值对，访问路径长度就是64（在以太坊定义的 Hex 字符集下）；每一级访问的节点都至少需要存储 16 个字节，这样就需要至少 1k 字节的额外空间，而且每次查找和删除都必须完整地执行 64 次下探访问。

梅克尔树 (Merkel Tree)

- 也被称作哈希树 (Hash Tree)，以数据块的 hash 值作为叶子节点存储值。梅克尔树的非叶子节点存储其子节点内容串联拼接后的 hash 值。





帕特里夏树 (Patricia Tree)

- 如果一个基数树的“基数”(radix)为2或2的整数次幂，就被称为“帕特里夏树”，有时也直接认为帕特里夏树就是基数树
- 以太坊中采用 Hex 字符作为 key 的字符集，也就是基数为 16 的帕特里夏树
- 以太坊中的树结构，每个节点可以有最多 16 个子节点，再加上 value，所以共有 17 个“插槽”(slot)位置
- 以太坊中的帕特里夏树加入了一些额外的数据结构，主要是为了解决效率问题



MPT (Merkel Patricia Tree)

- 梅克尔-帕特里夏树是梅克尔树和帕特里夏树的结合
- 以太坊中的实现，对 key 采用 Hex 编码，每个 Hex 字符就是一个 nibble (半字节)
- 遍历路径时对一个节点只访问它的一个 nibble，大多数节点是一个包含17个元素的数组；其中16个分别以 hex字符作为索引值，存储路径中下一个 nibble 的指针；另一个存储如果路径到此已遍历结束，需要返回的最终值。这样的节点叫做“分支节点”（branch node）
- 分支节点的每个元素存储的是指向下一级节点的指针。与传统做法不同，MPT 是用所指向节点的 hash 来代表这个指针的；每个节点将下个节点的 hash 作为自己存储内容的一部分，这样就实现了 Merkle 树结构，保证了数据校验的有效性



MPT 节点分类

MPT 中的节点有以下几类：

- 空节点 (NULL)
 - 表示空字符串
- 分支节点 (branch)
 - 17 个元素的节点，结构为 [v0 ... v15, vt]
- 叶子节点 (leaf)
 - 拥有两个元素，编码路径 encodedPath 和值 value
- 扩展节点 (extension)
 - 拥有两个元素，编码路径 encodedPath 和键 key

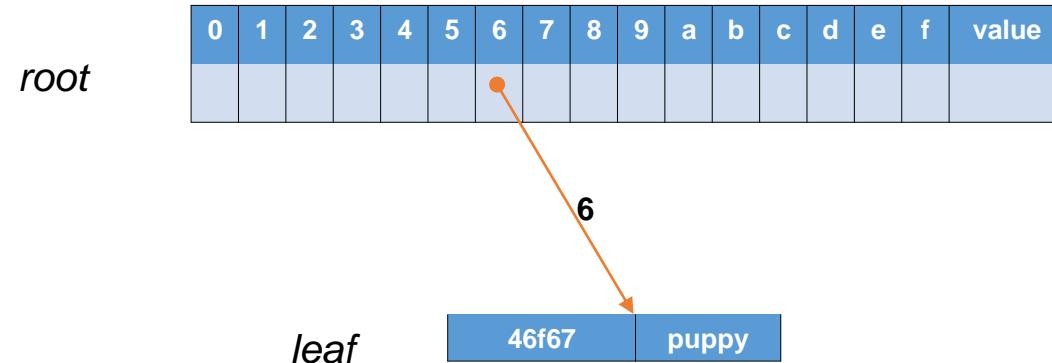


MPT 中数据结构的优化

- 对于64个字符的路径长度，很有可能在某个节点处会发现，下面至少有一段路径没有分叉；这很难避免
- 我们当然可以依然用标准的分支节点来表示，强制要求这个节点必须有完整的16个索引，并给没有用到的那15个位置全部赋空值；但这样有点蠢
- 通过设置“扩展节点”，就可以有效地缩短访问路径，将冗长的层级关系压缩成一个键值对，避免不必要的空间浪费
- 扩展节点（extension node）的内容形式是 [encodedPath, key]，其中 encodedPath 包含了下面不分叉的那部分路径，key 是指向下一个节点的指针（hash，也即在底层db中的存储位置）
- 叶子节点（leaf node）：如果在某节点后就没有了分叉路径，那这是一个叶子节点，它的第二个元素就是自己的 value



压缩之后的 “dog” 路径





紧凑编码 (compact coding)

- 路径压缩的处理相当于实现了压缩前缀树的功能；不过路径表示是 Hex 字符串 (nibbles)，而存储却是以字节 (byte) 为单位的，这相当于浪费了一倍的存储空间
- 我们可以采用一种紧凑编码 (compact coding) 方式，将两个 nibble 整合在一个字节中保存，这就避免了不必要的浪费
- 这里就会带来一个问题：有可能 nibble 总数是一个奇数，而数据总是以字节形式存储的，所以无法区分 nibble 1 和 nibbles 01；这就使我们必须分别处理奇偶两种情况
- 为了区分路径长度的奇偶性，我们在 encodedPath 中引入标识位



Hex 序列的压缩编码规则

- 我们在 encodedPath 中，加入一个 nibble 作为前缀，它的后两位用来标识节点类型和路径长度的奇偶性

Hex 字符	二进制位 (bits)	节点类型	路径长度
0	0000	扩展	偶
1	0001	扩展	奇
2	0010	叶子	偶
3	0011	叶子	奇

- MPT 中还有一个可选的“结束标记”（用T表示），值为 0x10 (十进制的16)，它仅能在路径末尾出现，代表节点是一个最终节点（叶子节点）
- 如果路径是奇数，就与前缀 nibble 凑成整字节；如果是偶数，则前缀 nibble 后补 0000 构成整字节



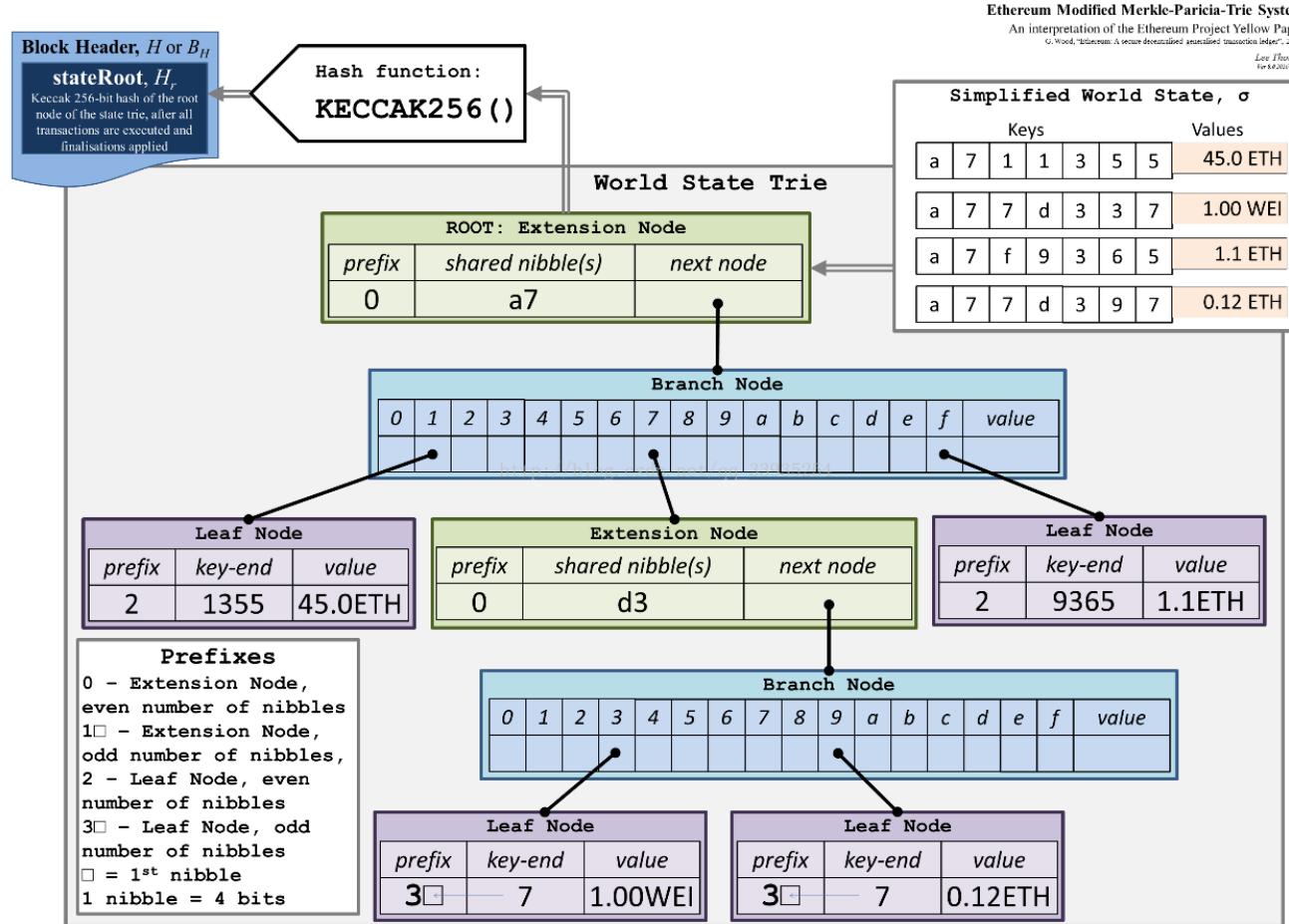
编码示例

- > [1, 2, 3, 4, 5, ...] 不带结束位, 奇路径
 - '11 23 45'
- > [0, 1, 2, 3, 4, 5, ...] 不带结束位, 偶路径
 - '00 01 23 45'
- > [0, f, 1, c, b, 8, 10] 带结束位 T 的偶路径
 - '20 0f 1c b8'
- > [f, 1, c, b, 8, 10] 带结束位 T 的奇路径
 - '3f 1c b8'



MPT 树结构示例

- 假设我们现在要构建一个存储了以下键值对的 MPT 树：
 - ('do', 'verb'), ('dog', 'puppy'), ('doge', 'coin'), ('horse', 'stallion')
- 首先我们会把所有的路径 (path) 转成 ASCII 码表示的 bytes：
 - <64 6f> : 'verb'
 - <64 6f 67> : 'puppy'
 - <64 6f 67 65> : 'coin'
 - <68 6f 72 73 65> : 'stallion'
- 然后我们就可以用在底层db中存储的以下键值对，构建出 MPT 树：
 - rootHash: [<16>, hashA]
 - hashA: [<>, <>, <>, <>, hashB, <>, <>, <>, hashC, <>, <>, <>, <>, <>, <>, <>, <>]
 - hashC: [<20 6f 72 73 65>, 'stallion']
 - hashB: [<00 6f>, hashD]
 - hashD: [<>, <>, <>, <>, <>, hashE, <>, <>, <>, <>, <>, <>, <>, <>, <>, 'verb']
 - hashE: [<17>, hashF]
 - hashF: [<>, <>, <>, <>, <>, hashG, <>, <>, <>, <>, <>, <>, <>, <>, <>, <>, 'puppy']
 - hashG: [<35>, 'coin']



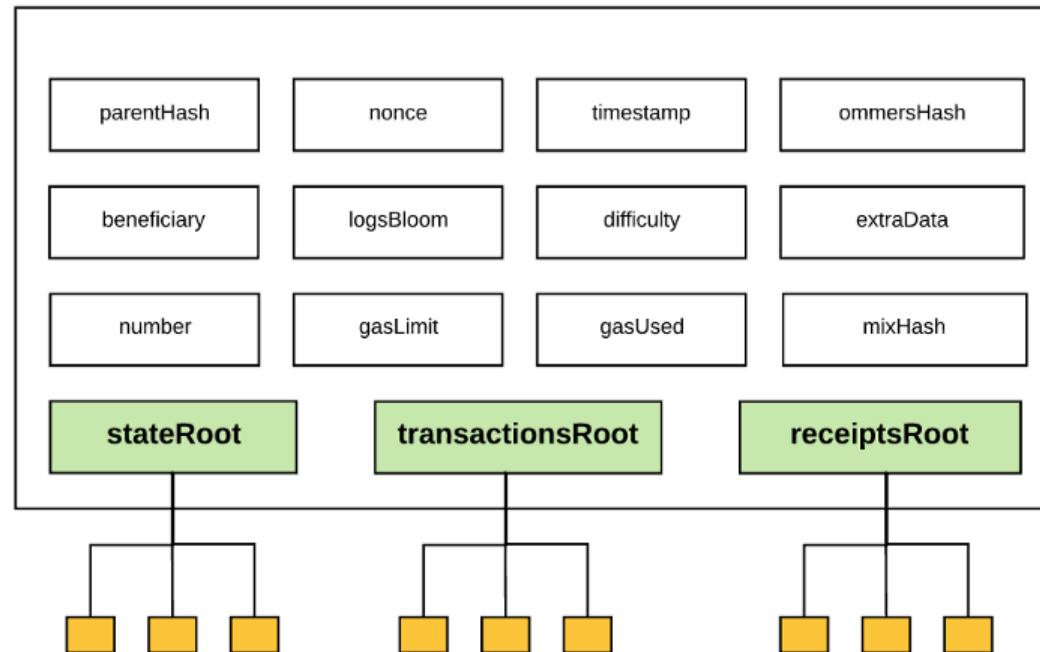


以太坊中树结构

- 以太坊中所有的 merkel 树都是 MPT
- 在一个区块的头部 (block head) 中，有三颗 MPT 的树根：
 - stateRoot
 - 状态树的树根
 - transactionRoot
 - 交易树的树根
 - receiptsRoot
 - 收据树的树根



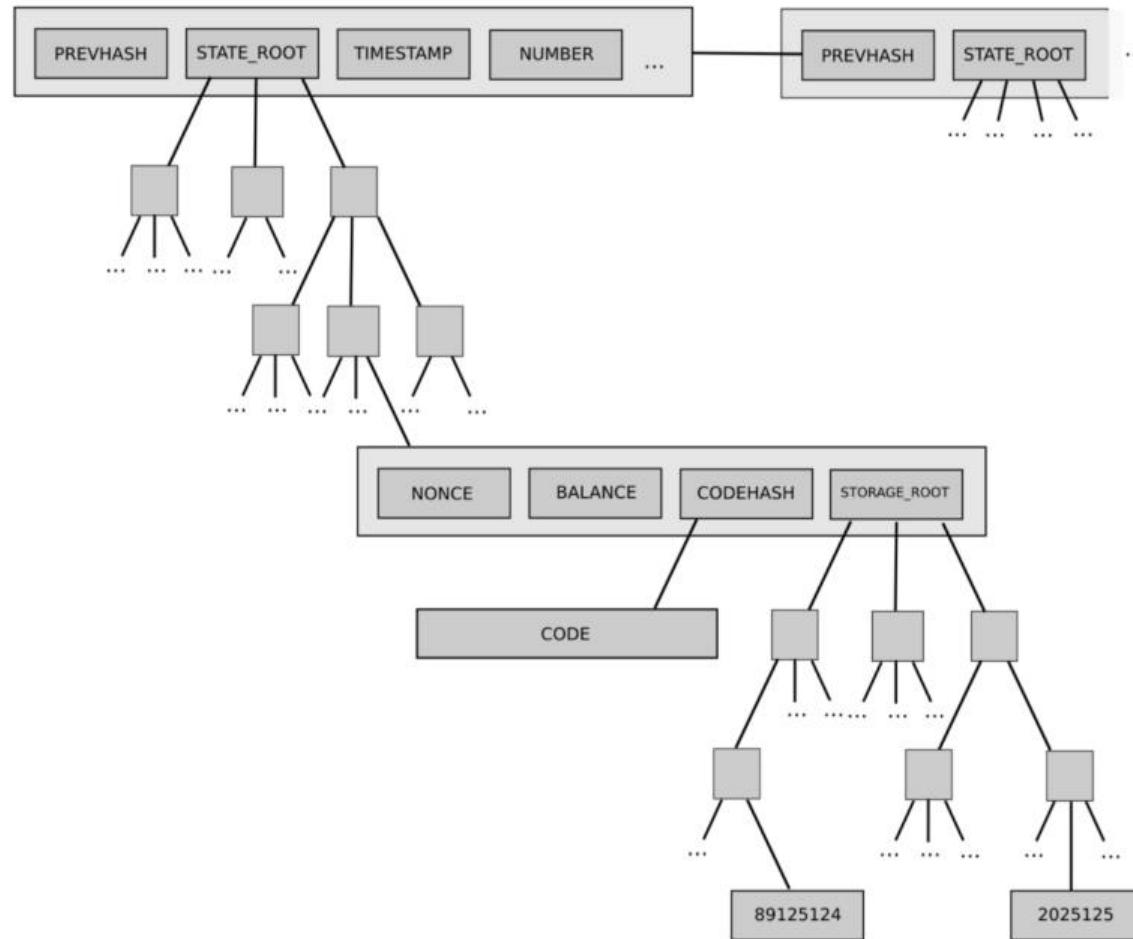
Block header





以太坊中树结构

- 状态树 (state trie)
 - 世界状态树，随时更新；它存储的键值对 (path, value) 可以表示为 (sha3(ethereumAddress), rlp(ethereumAccount))
 - 这里的 account 是4个元素构成的数组：[nonce, balance, storageRoot, codeHash]
- 存储树 (storage trie)
 - 存储树是保存所有合约数据的地方；每个合约账户都有一个独立隔离的存储空间
- 交易树 (transaction trie)
 - 每个区块都会有单独的交易树；它的路径 (path) 是 rlp(transactionIndex)，只有在挖矿时才能确定；一旦出块，不再更改
- 收据树 (receipts trie)
 - 每个区块也有自己的收据树；路径也表示为 rlp(transactionIndex)



下没有难学的技术



Q&A



尚硅谷



以太坊难度调整

2018.10



什么是难度

- **难度(Difficulty)**一词来源于区块链技术的先驱比特币，用来度量挖出一个区块平均需要的运算次数。
- 挖矿本质上就是在求解一个谜题，不同的电子币设置了不同的谜题。比特币使用SHA-256、莱特币使用Scrypt、以太坊使用Ethash。一个谜题的解的所有可能取值被称为解的空间，挖矿就是在这些可能的取值中寻找一个解
- 这些谜题都有如下共同的特点：
 - 没有比穷举法更有效的求解方法
 - 解在空间中均匀分布，从而使每一次穷举尝试找到一个解的概率基本一致
 - 解的空间足够大，保证一定能够找到解



什么是难度

- 现在我们为谜题设置一个参数 **Difficulty**，那么谜题就变成了求解某个空间内符合 $x < \text{Difficulty}$ 的 x ，这个参数 **Difficulty** 就是所谓的**难度**
- 难度(Difficulty) 通过控制合格的解在空间中的数量来控制平均求解所需要尝试的次数，也就可以间接的控制产生一个区块需要的时间，这样就可以使区块以一个合理而稳定的速度产生
- 当挖矿的人很多，单位时间能够尝试更多次时，难度就会增大，当挖矿的人减少，单位时间能够尝试的次数变少时，难度就降低。这样产生一个区块需要的时间就可以做到稳定



以太坊中的难度计算

- 难度计算的规则

- 以太坊中有三种计算难度的规则，分别对应着以太坊中三个不同阶段的版本：Frontier, Homestead 和 Metropolis，现在用的方法叫做 calcDifficultyByzantium ()，对应大都会的拜占庭阶段

- 计算一个区块的难度时，需要以下输入：

- parent_timestamp：上一个区块产生的时间
 - parent_diff：上一个区块的难度
 - block_timestamp：当前区块产生的时间
 - block_number：当前区块的序号



以太坊中的难度计算

计算步骤：

- $\text{block_diff} = \text{parent_diff} + \text{难度调整} + \text{难度炸弹}$
- 难度调整 = $\text{parent_diff} / 2048 * \max((2 \text{ if } \text{len}(\text{parent.uncles}) \\ \text{else } 1) - ((\text{timestamp} - \text{parent.timestamp}) // 9), -99))$
- 难度炸弹 = $2^{(\text{parent.Number} - \text{bombDelay}) // 100000} - 2$
- 目前拜占庭阶段， $\text{bombDelay} = 3000000$
- 另外，区块难度不能低于以太坊的创世区块，创世区块的难度为131072，这是以太坊难度的下限。



Q&A



尚硅谷



基于 token 的投票（一）

——用 truffle 构建简单投票 DApp

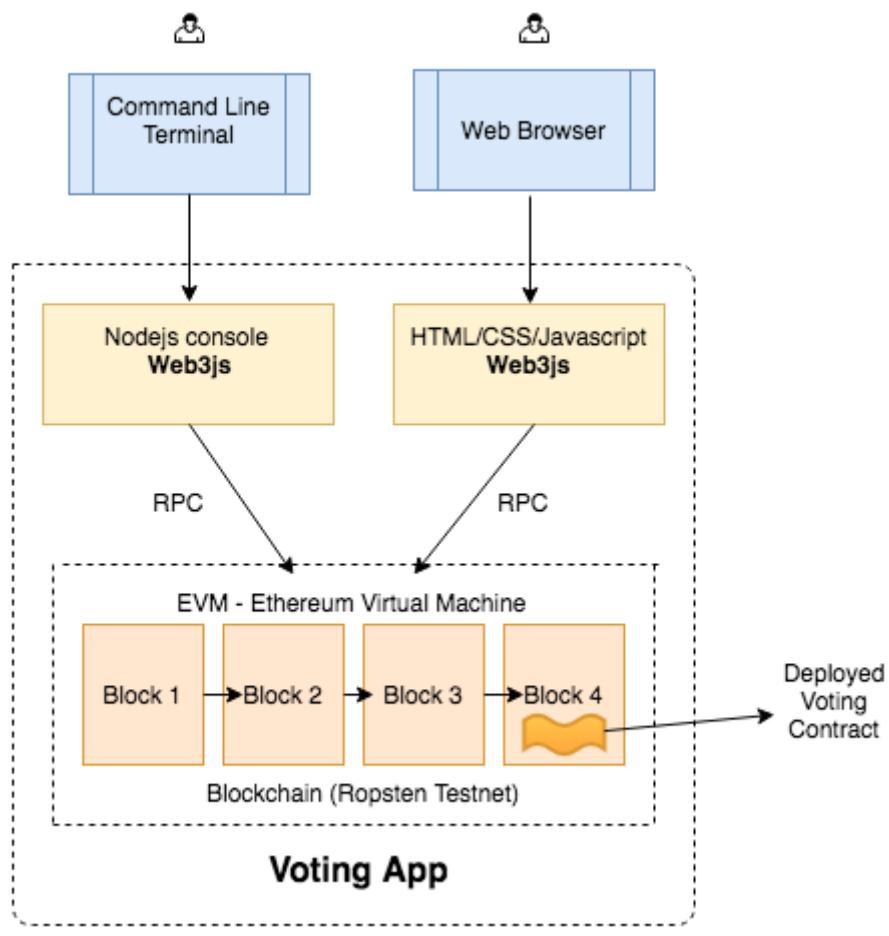
在课程“简单投票 Dapp”中，你已经在一个模拟的区块链（ganache）上实现了一个投票合约，并且成功地通过 nodejs 控制台和网页与合约进行了交互。

在接下来的项目学习中，我们将会实现以下内容：

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

这篇文档将主要带领大家完成 1~4 的阶段。





准备工作

用 Geth 启动私链

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

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

来看一下启动 geth 节点时传入参数代表的意思。

--datadir: 指定区块链数据的存储目录，这里我们就在当前目录启动。



--rpc 启用 HTTP-RPC 服务器。

--rpcapi db,eth,net,web3,personal,miner: 基于 HTTP-RPC 接口提供的 API。

这是告诉 geth 通过 RPC 接收请求，同时启用我们将会在之后使用的一些 API。

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

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

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

如果我们想到直接连接到测试网络，可以用下面的命令：

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

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

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

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



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

用 Rinkeby 替换 Ropsten

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

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

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

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

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

工作流 (Workflow)

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

Development (开发环境) : Ganache

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

Production (生产环境) : Mainnet



Truffle

安装

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

你可以像这样使用 npm 安装 truffle：

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

然后我们创建一个空目录，在下面创建 truffle 项目：

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

Unbox 的过程相对会长一点，完成之后应该看到这样的提示：

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

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

```
Downloading...  
Unpacking...  
Setting up...  
Unbox successful. Sweet!  
  
Commands:  
  
Compile:           truffle compile  
Migrate:          truffle migrate  
Test contracts:   truffle test  
Run linter:        npm run lint  
Run dev server:   npm run dev  
Build for production: npm run build
```

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



webpack: 一个流行的前端资源依赖管理和打包工具。

Truffle 简介

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

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

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

Truffle 的客户端

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

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

- Ganache
- truffle develop



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

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

```
>truffle develop
```

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

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

创建 Voting 项目

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

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

- **app/** - 你的应用文件运行的默认目录。这里面包括推荐的 **javascript** 文件和 **css** 样式文件目录，但你可以完全决定如何使用这些目录。
- **contract/** - Truffle 默认的合约文件存放目录。
- **migrations/** - 部署脚本文件的存放目录



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

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

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

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

Migration

migration 的概念

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

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

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



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

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

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

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

像所有其它在 `Truffle` 中的代码一样，`Truffle` 提供了我们自己代码的合约抽象层(`contract abstractions`)，并且进行了初始化，以方便你可以便利的与以太坊的网络交互。这些抽象接口都是部署流程的一部分。

更新 `migration` 文件

将 `2_deploy_contracts.js` 的内容更新为以下信息：

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

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



更新 truffle 配置文件

像下面这样更新 `truffle.js` 的内容:

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

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

合约代码

`Voting.sol`

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

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



创建账户（可用 metamask 上账户转币）

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

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

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

部署

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

```
>truffle compile
```



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

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

可能出现的问题和解决方案

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

控制台和网页交互

如果部署顺利，你可以通过控制台和网页与合约进行交互。



app/index.html

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

在标题<h1>下加 address <div id="address"></div>

在表格</table>下加 msg <div id="msg"></div>

app/scripts/index.js

新建 JavaScript 文件 app/scripts/index.js

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



```
contractInstance.totalVotesFor
    .call(candidateName).then(function(v) {
        $("#" + div_id).html(v.toString());
        $("#msg").html("");
    });
});

} catch (err) {
    console.log(err);
}

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

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



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

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

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

控制台交互

需要重新打开一个新的 console

```
>truffle console  
truffle(default)>  
Voting.deployed().then(function(contractInstance)  
{contractInstance.voteForCandidate('Alice').then(function(v)  
{console.log(v)}))}  
  
{ blockHash:  
'0x7229f668db0ac335cdd0c4c86e0394a35dd471a1095b8fafb52ebd76714  
33156',  
blockNumber: 469628,
```



```
contractAddress: null,  
....  
....  
truffle(default)>  
Voting.deployed().then(function(contractInstance)  
{contractInstance.totalVotesFor.call('Alice').then(function(v)  
{console.log(v)}))}  
  
{ [String: '1'] s: 1, e: 0, c: [ 1 ] }
```

在调用 `voteForCandidate` 方法之后需要稍等一下，因为发送交易需要时间；注意，`truffle` 的所有调用都会返回一个 `promise`，这就是为什么能看到每个响应被包装在 `then()` 函数下面；另外 `totalVoteFor()` 方法也可以不加 `.call()` 直接调用，不会发送交易。

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

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

网页交互

在控制台用 `webpack` 启动服务器：

```
>npm run dev
```

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

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



到目前为止，我们已经用 truffle 构建了一个真正的 Dapp。





基于 token 的投票 (二)

——基于 token 的投票 DApp

之前的课程中我们已经学习了用 truffle 来构建 DApp 并部署到 Rinkeby 测试网络，接下来我们就会在原先简单投票的基础上将合约进行扩展，实现一个基于 token 的投票 DApp。

代币和支付

在以太坊中，你会遇到的一个重要概念就是 token（代币）。token 就是在以太坊上构建的数字资产。token 可以代表物理世界里的一些东西，比如黄金，或者可以是自己的数字资产（就像货币一样）。token 实际上就是智能合约，并没有什么神奇之处。

1. **Gold Token** (黄金代币)：银行可以有 1 千克的黄金储备，然后发行 1 千的 token。买 100 个 token 就等于买 100 克的黄金。
2. **Shares in a company** (公司股票)：公司股票可以用以太坊上的 token 来表示。通过支付以太，人们可以购买公司 token (股票)。
3. **Gaming currency** (游戏货币)：你可以有一个多玩家的游戏，游戏者可以用以太购买 token，并在游戏购买中进行花费。
4. **Golem Token**: 这是一个以太坊项目的真实 token，你可以通过租售空闲的 CPU 来赚取 token。
5. **Loyalty Points** (忠诚度)：当你在一个商店购物，商店可以发行 token 作为忠诚度点数，它可以在将来作为现金回收，或是在第三方市场售卖。

在合约中如何实现 token，实际上并没有限制。但是，有一个叫做 ERC20 的 token 标准，该标准也会不断进化。ERC20 token 的优点是很容易其他的 ERC20 token 互换。同时，也更容易将你的 token 集成到其他 dapp 中。

¹

更多 Java -大数据 -前端 -python 人工智能 -区块链资料下载，可访问百度：尚硅谷官网



在接下来的课程中，我们向 Voting 项目中包含 token 和支付。总的来说，我们会覆盖以下内容：

1. 使用 struct 来定义更加复杂的数据类型，在区块链上组织和存储数据
2. 实现投票的 token 化表达
3. 连接 token、投票应用和以太坊上的支付，构建完整的 DApp。

项目描述

一提到投票，你通常会想起普通的选举，你会通过投票来选出国家的首相或总统。在这种情况下，每个公民都会有一票，可以投给他们看中的候选者。

还有另外一种叫做加权投票 (weighted voting) 的投票方式，它常常用于公开上市交易的公司。在这些公司，股东使用它们的股票进行投票。比如，如果你拥有 10,000 股公司股票，你就有 10,000 个投票权（而不是普通选举中的一票）。我们会实现加权投票。

项目细节



比如说，我们有一个叫做 Block 的上市公司。公司有 3 个职位空闲，分别是总裁，副总裁和部长。这几个职位有 3 个竞争人选。公司想要进行选举，股东决定哪个候选人得到哪个职位。拥有最高投票的候选人将会成为总裁，然后是副总裁，最后是部长。我们会构建一个项目，并发行公司股票，允许任何人购买股票。基于所拥有的股票数，他们可以为候选人投票。比如，如果你有 10,000 股，你可以一个候选人投 5,000 股，另一个候选人 3,000 股，第三个候选人 2,000 股。

接下来，我们将会勾勒出实现框架，并随后实现构建完整应用的所有组件。

实现计划

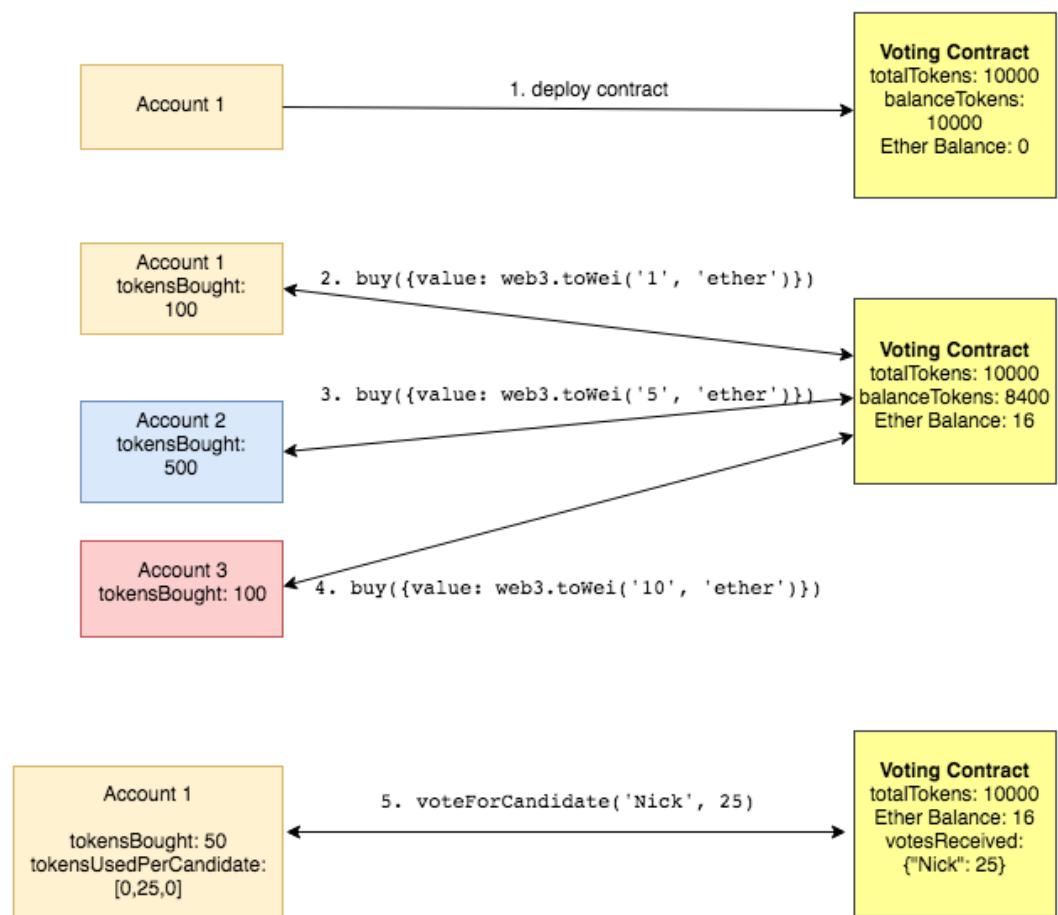
1. 我们首先会创建一个与之前类似新的 truffle 项目。并且再次与 2_deploy_contracts.js, Voting.sol, index.html, app.js 和 app.css 打交道。



2. 我们会初始化在选举中竞争的候选者。从之前的课程中，我们已经知道了如何实现这一点。我们将会在 `2_deploy_contracts.js` 中完成这个任务。
3. 对于投票的股东，他们需要持有公司股票。所以，我们会先初始化公司股票。这些股票就是构成公司的数字资产。在以太坊的世界中，这些数字资产就叫做 `token`。从现在开始，我们将会把这些股票称为 `token`。除了候选者，我们还会 `deployment` 文件里的合约构造函数里初始化所有的 `token`。（提示，股票可以看做是 `token`，但是并非所有的以太坊 `token` 都是股票。股票仅仅是我们前一节中提到的 `token` 使用场景的一种）
4. 我们会向合约中引入一个新的方法，让任何人购买这些 `token`，他们会用这些 `token` 给候选人投票。
5. 我们也会加入一个函数来查询投票人信息，以及他们已经给谁投了票，有多少 `token`，他们的 `token` 余额。
6. 为了跟踪所有这些数据，我们会用到几个 `mapping` 字段，并会引入一个新的数据结构 `struct` 来组织投票信息。

下图是我们将要在本课程实现应用的图示。现在并不需要理解图示中的所有内容。在后面我们将会进一步阐释。





初始化 truffle 项目

在之前的学习中，你已经在系统里安装好了 webpack 和 truffle。如下所示，初始化 truffle 项目，并从 contracts 目录下移除 MetaCoin.sol。

```

>mkdir token_based_voting_dapp
>cd token_based_voting_dapp
>truffle unbox webpack
>ls
README.md      contracts      node_modules      test
webpack.config.js  truffle.jsapp      migrations
package.json
>ls app/

```



```
index.html javascripts stylesheets  
>ls contracts/  
ConvertLib.sol MetaCoin.sol Migrations.sol  
>ls migrations/  
1_initial_migration.js 2_deploy_contracts.js  
>rm contracts/ConvertLib.sol contracts/MetaCoin.sol
```

投票合约

创建合约代码 Voting.sol。下面会给出详细的代码解释。

```
pragma solidity ^0.4.18;  
contract Voting {  
    struct voter {  
        address voterAddress;  
        uint tokensBought;  
        uint[] tokensUsedPerCandidate;  
    }  
    mapping (address => voter) public voterInfo;  
    mapping (bytes32 => uint) public votesReceived;  
    bytes32[] public candidateList;  
    uint public totalTokens;  
    uint public balanceTokens;  
    uint public tokenPrice;  
    constructor(uint tokens, uint pricePerToken, bytes32[] candidateNames) public {  
        candidateList = candidateNames;  
        totalTokens = tokens;  
        balanceTokens = tokens;  
        tokenPrice = pricePerToken;  
    }  
}
```



```
function buy() payable public returns (uint) {
    uint tokensToBuy = msg.value / tokenPrice;
    require(tokensToBuy <= balanceTokens);
    voterInfo[msg.sender].voterAddress = msg.sender;
    voterInfo[msg.sender].tokensBought += tokensToBuy;
    balanceTokens -= tokensToBuy;
    return tokensToBuy;
}

function totalVotesFor(bytes32 candidate) view public
returns (uint) {
    return votesReceived[candidate];
}

function voteForCandidate(bytes32 candidate, uint
votesInTokens) public {
    uint index = indexOfCandidate(candidate);
    require(index != uint(-1));
    if ( voterInfo[msg.sender].
        tokensUsedPerCandidate.length == 0) {
        for(uint i = 0; i < candidateList.length
            ;i++) {
            voterInfo[msg.sender]
                .tokensUsedPerCandidate
                .push(0);
        }
    }
    uint availableTokens =
        voterInfo[msg.sender].tokensBought -
        totalTokensUsed(voterInfo[msg.sender]
            .tokensUsedPerCandidate);
    require (availableTokens >= votesInTokens);
    votesReceived[candidate] += votesInTokens;
}
```



```
voterInfo[msg.sender]
.tokensUsedPerCandidate[index] += votesInTokens;
}
function totalTokensUsed(uint[] _tokensUsedPerCandidate)
private pure returns (uint) {
    uint totalUsedTokens = 0;
    for(uint i = 0; i < _tokensUsedPerCandidate.length;
        i++) {
        totalUsedTokens +=
            _tokensUsedPerCandidate[i];
    }
    return totalUsedTokens;
}
function indexOfCandidate(bytes32 candidate) view public
returns (uint) {
    for(uint i = 0; i < candidateList.length; i++) {
        if (candidateList[i] == candidate) {
            return i;
        }
    }
    return uint(-1);
}
function tokensSold() view public returns (uint) {
    return totalTokens - balanceTokens;
}
function voterDetails(address user) view public returns
(uint, uint[])
{
    return (voterInfo[user].tokensBought
        , voterInfo[user].tokensUsedPerCandidate);
}
function transferTo(address account) public {
    account.transfer(this.balance);
```



```
    }

    function allCandidates() view public returns (bytes32[]) {
        return candidateList;
    }

}
```

之前，我们仅仅有 2 个合约属性：一个数组 `candidateList` 存储所有的候选者，一个 `mapping votesReceived` 跟踪每个候选者获得的投票。

在这个合约中，我们必须再额外跟踪几个值：

- 每个投票人的信息: `solidity` 有个叫做 `struct` 的数据类型，它可以用来一组相关数据。用 `struct` 来存储投票人信息非常好（如果你之前没有听过 `struct`, 把它想成一个面向对象的类即可，里面有 `getter` 和 `setter` 方法来获取这些属性）。我们会用 `struct` 存储投票人的地址，他们已经购买的所有 `token` 和给每个候选者投票所用的 `token`。（Line 5-9）
- 查询投票人信息的 `mapping`: 给定一个投票人的账户地址，我们想要显示他的信息。我们会使用 `voterInfo` 字段来存储信息。（Line 11）
- `Tokens`: 我们需要有存储发行 `token` 总量的合约变量，还需要存储所有剩余的 `token` 和每个 `token` 的价格。（Line 17-19）

Line 21: 像上一节一样初始化构造函数。因为我们会发行任何人都可以购买的 `token`，除了候选者，我们必须设置所有售卖的 `token` 和每个 `token` 的价格。

Line 28: `buy` 函数用于购买 `token`。注意关键字 “`payable`”。通过向一个函数添加一个关键字，任何人调用这个函数，你的合约就可以接受支付(通过以太)。

Line 28 - 35: 当你调用合约的 `buy` 方法时，在请求里设置你想要用于购买 `token` 的所有以太。以太的值通过 `msg.value`。基于以太的值和 `token` 价格，你就可以计算出所有的 `token`，并将这些 `token` 赋予购买人。购买人的地址通过 `msg.sender` 可以获取。



下面是从 truffle 控制台调用 `buy` 的一个案例，参数传入一个 `options` 对象，这是 web3 v0.2x 的用法：

```
truffle(development)> Voting.deployed().then(function(contract){contract.buy({value: web3.toWei('1', 'ether'), from: web3.eth.accounts[1]})})
```

它相当于 web3 v1.0 中的

```
contract.buy().send({options})
```

如果是消息调用的话就应该是

```
contract.method(parameters).call({options})
```

Line 41 - 56: `voteForCandidate` 方法现在有一点复杂，因为我们不仅要增加候选人的投票数，还是跟踪投票人的信息，比如投票人是谁（即他们的账户地址），给每个候选人投了多少票。

Line 83 - 85: 当一个用户调用 `buy` 方法发送以太来购买 `token` 时，所有的以太去了哪里？所有以太都在合约里。每个合约都有它自己的地址，这个地址里面存储了这些钱。可这些钱怎么拿出来呢？我们已经在这里定义了 `transferTo` 函数，它可以让你转移所有钱到指定的账户。该方法目前所定义的方式，任何人都可以调用，并向他们的账户转移以太，这并不是一个好的选择。你可以给谁能取钱上施加一些限制。虽然这已经超过了本课程的内容，但是我们推荐在未来实现这一点。

合约里面剩下的方法都是 `getter` 方法，仅仅返回合约变量的值。



注意方法上的 `view` 修改符，比如 `tokensSold`, `voterDetails` 等等。这些方法并不会改变区块链状态，也就是说这些是只读的方法。执行这些交易不会耗费任何 `gas`。

合约部署

与之前类似，更新 `migrations/2_deploy_contracts.js`，不过这次你需要传入两个额外的参数 “`total tokens to issue`”（示例给了 `10000`）和每个 `token` 的成本（`0.01` 以太）。所有的价格需要以 `Wei` 为单位计价，所以我们需要用 `toWei` 将 `Ether` 转换为 `Wei`。

```
var Voting = artifacts.require("./Voting.sol");
module.exports = function(deployer) {
    deployer.deploy(Voting, 10000,
        web3.toWei('0.01', 'ether'),
        ['Alice', 'Bob', 'Cary']);
};
```

让我们将合约部署到 `ganache`，测试与交互，确保代码如期工作。然后我们会把合约部署到公共的测试网。如果已经运行了 `geth`，停止 `geth` 然后启动 `ganache`。记得将 `truffle.js` 里的 `ganache` 改为 `development`, `port` 改为 `8545`；之后继续并将合约部署到网络上。

```
> truffle compile
Compiling Migrations.sol...
Compiling Voting.sol...
Writing artifacts to ./build/contracts
> truffle migrate
Running migration: 1_initial_migration.js
```



```
Deploying Migrations...
Migrations: 0x3cee101c94f8a06d549334372181bc5a7b3a8bee
Saving successful migration to network...
Saving artifacts...
Running migration: 2_deploy_contracts.js
Deploying Voting...
Voting: 0xd24a32f0ee12f5e9d233a2ebab5a53d4d4986203
Saving successful migration to network...
Saving artifacts...
```

控制台交互

```
> truffle console
```

接下来我们做一个控制台交互测试。如果成功地将合约部署到了 ganache，启动 truffle 控制台并执行以下操作，在 truffle 控制台打印（console.log）：

1. 一个候选人（比如 Alice）有多少投票？
2. 一共初始化了多少 token？
3. 已经售出了多少 token？
4. 购买 100 token
5. 购买以后账户余额是多少？
6. 已经售出了多少？
7. 给 Alice 投 25 个 token，给 Bob 和 Cary 各投 10 个 token。
8. 查询你所投账户的投票人信息（除非用了其他账户，否则你的账户默认是 web3.eth.accounts[0]）
9. 现在每个候选人有多少投票？
10. 合约里有多少 ETH？（当你通过 ETH 购买 token 时，合约接收到的 ETH）



```
truffle(development)> Voting.deployed().then(function(instance)
{instance.totalVotesFor.call('Alice').then(function(i)
{console.log(i)}))}

truffle(development)> Voting.deployed().then(function(instance)
{console.log(instance.totalTokens.call().then(function(v)
{console.log(v))))})

truffle(development)> Voting.deployed().then(function(instance)
{console.log(instance.tokensSold.call().then(function(v)
{console.log(v))))})

truffle(development)> Voting.deployed().then(function(instance)
{console.log(instance.buy({value: web3.toWei('1',
'ether')})).then(function(v) {console.log(v)}))})

truffle(development)> web3.eth.getBalance(web3.eth.accounts[0])

truffle(development)> Voting.deployed().then(function(instance)
{console.log(instance.tokensSold.call().then(function(v)
{console.log(v))))})

truffle(development)> Voting.deployed().then(function(instance)
{console.log(instance.voteForCandidate('Alice',
25).then(function(v) {console.log(v)}))})

truffle(development)> Voting.deployed().then(function(instance)
{console.log(instance.voteForCandidate('Bob',
10).then(function(v) {console.log(v)}))})

truffle(development)> Voting.deployed().then(function(instance)
{console.log(instance.voteForCandidate('Cary',
10).then(function(v) {console.log(v)}))})

truffle(development)> Voting.deployed().then(function(instance)
{console.log(instance.voterDetails.call(web3.eth.accounts[0]).then(function(v) {console.log(v)}))})
```



```
truffle(development)> Voting.deployed().then(function(instance)
{instance.totalVotesFor.call('Alice').then(function(i)
{console.log(i)}))}

truffle(development)>
web3.eth.getBalance(Voting.address).toNumber()
```

Html 视图

现在，我们已经知道了合约如期工作。让我们来构建前端逻辑，以便于能够通过网页浏览器与合约交互。

将下面内容拷贝到 `app/index.html`。

```
<!DOCTYPE html>

<html>

<head>
  <title>Decentralized Voting App</title>
  <link
    href='https://fonts.googleapis.com/css?family=Open+Sans:400,700' rel='stylesheet' type='text/css'>
  <link
    href='https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css' rel='stylesheet'
    type='text/css'>
  <style></style>
</head>
<body class="row">
  <h1 class="text-center banner">Decentralized Voting
  Application (Ropsten Testnet)</h1>
  <div class="container">
```



```
<div class="row margin-top-3">
    <div class="col-sm-12">
        <h3>How to use the app</h3>
        <strong>Step 1</strong>: Install the
        <a href="https://metamask.io/">
            metamask plugin</a>
        and create an account on Ropsten Test Network
        and load some Ether.
        <br>
        <strong>Step 2</strong>: Purchase tokens below by
        entering the total number of tokens you like
        to buy.
        <br>
        <strong>Step 3</strong>: Vote for candidates by entering
        their name and no. of tokens to vote with.
        <br>
        <strong>Step 4</strong>: Enter your account address to
        look up your voting activity.
    </div>
</div>
<div class="row margin-top-3">
    <div class="col-sm-7">
        <h2>Candidates</h2>
        <div class="table-responsive">
            <table class="table table-bordered">
                <thead>
                    <tr>
                        <th>Candidate</th>
                        <th>Votes</th>
                    </tr>
                <tbody>
```



```
</thead>
<tbody id="candidate-rows">
</tbody>
</table>
</div>
</div>

<div class="col-sm-offset-1 col-sm-4">
<h2>Tokens</h2>
<div class="table-responsive">
<table class="table table-bordered">
<tr>
<th>Tokens Info</th>
<th>Value</th>
</tr>
<tr>
<td>Tokens For Sale</td>
<td id="tokens-total"></td>
</tr>
<tr>
<td>Tokens Sold</td>
<td id="tokens-sold"></td>
</tr>
<tr>
<td>Price Per Token</td>
<td id="token-cost"></td>
</tr>
<tr>
<td>Balance in the contract</td>
<td id="contract-balance"></td>
</tr>
```



```
</table>

</div>

</div>

<hr>

<div class="row margin-bottom-3">

    <div class="col-sm-7 form">

        <h2>Vote for Candidate</h2>

        <div id="msg"></div>

        <input type="text" id="candidate" class="form-control"
               placeholder="Enter the candidate name"/>

        <br>

        <br>

        <input type="text" id="vote-tokens" class="form-control"
               placeholder="Total no. of tokens to vote"/>

        <br>

        <br>

        <a href="#" onclick="voteForCandidate(); return false;" class="btn btn-primary">Vote</a>

    </div>

    <div class="col-sm-offset-1 col-sm-4">

        <div class="col-sm-12 form">

            <h2>Purchase Tokens</h2>

            <div id="buy-msg"></div>

            <input type="text" id="buy" class="col-sm-8"
                   placeholder="Number of tokens to buy"/>

            <a href="#" onclick="buyTokens(); return false;" class="btn btn-primary">Buy</a>

        </div>

        <div class="col-sm-12 margin-top-3 form">
```



```
<h2 class="">Lookup Voter Info</h2>
<input type="text" id="voter-info", class="col-sm-8"
placeholder="Enter the voter address" />
<a href="#" onclick="lookupVoterInfo(); return
false;" class="btn btn-primary">Lookup</a>
<div class="voter-details row text-left">
<div id="tokens-bought" class="margin-top-3
col-md-12"></div>
<div id="votes-cast" class="col-md-12"></div>
</div>
</div>
</div>
</body>
<script
src="https://code.jquery.com/jquery-3.1.1.slim.min.js">
</script>
<script src="app.js"></script>
</html>
```

如果仔细看代码的话，你会发现已经没有硬编码的值了。候选者的名字会通过向部署好的合约查询进行填充。

它也会显示公司所发行的所有 token，已售出和剩余的 token。

有一节，你可以输入一个账户地址（投票人的地址），观察他们的投票行为和 token。



JavaScript

通过移除候选者姓名等等的硬编码，我们已经大幅改进了 `HTML` 文件。我们会使用 `javascript/web3js` 来填充 `html` 里面的所有值，并实现购买 `token` 的查询投票人信息的额外功能。

我们推荐用 `JavaScript` 自己实现，代码仅作参考之用。按照下述指引帮助实现：

- 创建一个 `Voting` 合约的实例
- 在页面加载时，初始化并创建 `web3` 对象。（第一步和第二步与之前的课程一模一样）
- 创建一个在页面加载时调用的函数，它需要：
- 使用 `Voting` 合约对象，向区块链查询来获取所有的候选者姓名并填充表格。
- 再次查询区块链得到每个候选人所获得的所有投票并填充表格的列。
- 填充 `token` 信息，比如所有初始化的 `token`，剩余 `token`，已售出的 `token` 以及 `token` 成本。
- 实现 `buyTokens` 函数，它在上一节的 `html` 里面调用。你已经在控制台交互一节中购买了 `token`。`buyTokens` 代码与那一节一样不可或缺。
- 类似地，实现 `lookupVoterInfo` 函数来打印一个投票人的细节。

网页交互

CSS: `app/styles/app.css.`

在命令行中，使用 `npm run dev` 启动 `web` 服务器，完后你应该看到下面的内容。

如果一切顺利，你应该可以购买更多的 `token`，为任意候选者投票并查看投票人信息。



Candidate	Votes
Rama	25
Nick	25
Jose	10

Vote for Candidate

Enter the candidate name:
Total no. of tokens to vote:

Token Stats

Tokens For Sale	10000
Tokens Sold	100
Price Per Token	0.01 Ether
Balance in the contract	1 Ether

Purchase Tokens

Number of tokens to buy: **Buy**

Lookup Voter Info

Address: **Lookup**

Total Tokens bought: 100
Votes cast per candidate:
Rama: 25
Nick: 25
Jose: 10

测试网络

现在，你可以关闭 `ganache`，再次启动 `geth` 并运行 `truffle` 部署到测试网。

鉴于这是一个部署在区块链上的去中心化应用，任何人都可以接入你的应用并与之交互。如果你还记得上一课，你需要将 `ABI` 和合约地址分享给那些想要接入你的应用的人。你可以在 `truffle` 的 `build/contracts/Voting.json` 找到 `ABI` 和合约地址。（这会让任何人通过命令行进行交互。如果喜欢其他人通过 `GUI` 使用你的应用，你仍然需要托管 `web` 前端。）

练习

现在合约的实现方式，用户购买 `token` 并用 `token` 投票。但是他们投票的方式是向合约发送 `token`。如果他们还需要在未来的选举中投票怎么办？每次投票都需要购买 `token` 显然是不合理的，而他们所有的 `token` 都会保留在合约中，并不在自己手上。进一步改善合约的方式是，加入一个方式以便于用户能够收回他们的 `token`。你必须实现这样一个方法，查询用户投票的所有 `token`，并将这些 `token` 返回给他们。



测试

Truffle 自带了一个自动化的测试框架，这使得测试合约非常容易。你可以通过两种方式用这个框架来写测试：

1. Solidity

2. Javascript

一般的经验是用 solidity 写单元测试，用 JavaScript 写功能测试。但是，从我们的经验来看，大部分开发者常常只会用 JavaScript 写测试。在这一章节中，你将会学习如何用这两种方式编写测试。

在这一小节中，我们会写一些 Solidity 的测试，并在下一个章节中涉及 JavaScript 测试。

当你创建好 truffle 项目后，truffle 就会在 test 目录下自动创建好 TestMetacoin.sol 和 metacoin.js。因为我们已经不再需要这些示例合约了，所以删除这些文件。

Solidity 测试

下面是 solidity 测试文件，File: TestVoting.sol

```
pragma solidity ^0.4.16;
import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/Voting.sol";
contract TestVoting {
    uint public initialBalance = 2 ether;
    function testInitialTokenBalanceUsingDeployedContract()
        public {
        Voting voting = Voting(DeployedAddresses.Voting());
        uint expected = 10000;
        Assert.equal(voting.balanceTokens(), expected, "10000
            Tokens not initialized for sale");
    }

    function testBuyTokens() public {
```



```
Voting voting = Voting(DeployedAddresses.Voting());
voting.buy.value(1 ether)();
Assert.equal(voting.balanceTokens(), 9900, "9900
tokens should have been available");
}
}
```

解释如下：

1. 测试文件应该像这样命名 “`Test.sol`”. 这样，`truffle` 框架才能知道这是我们要测试合约对应的测试文件。
2. Line 2: `Truffle` 框架提供了一个断言的库 `Assert.sol`, 你可以用它来断言合约相关的任何值。它有一些函数用来断言 `equal`, `notEqual`, `isAbove`, `isBelow`, `isAtLeast`, `isAtMost`, `isZero` 和 `isNotZero`.
3. Line 3: 每当运行一个测试时，`truffle` 都会部署你的合约。`DeployedAddress` 是一个 `truffle` 框架的帮助库。通过调用 `DeployedAddress.()` 即可获取部署合约的地址。
4. Line 6: 在这个测试文件中，你会用 `TestVoting` 合约与实际的 `Voting` 合约进行交互。为了测试合约能够执行函数，它需要以太。声明一个 `initialBalance` 共有变量，并初始化一些以太。
5. Line 7 - 11: 在 `testInitialTokenBalanceUsingDeployedContract` 中，我们是测试当部署合约后，确保初始化了 10000 个代币。如果你还记得的话，代币的数量在 `migrations/2_deploy_contracts.js` 是在进行了指定。
6. Line 12 - 16: 在 `testBuyTokens` 中，智能合约购买代币，我们断言确保售出 100 个代币。记住，如果你不提供 `initialBalance`，测试合约就没有以太来购买代币，交易就会失败。

如下所示运行测试，如果你的合约代码没有任何 bug，那么测试应该会通过。我们鼓励大家多写几个测试来练习其他合约函数。

```
>truffle test test/TestVoting.sol
```

Javascript 测试



下面的 JavaScript 测试代码对你来说可能看着比较熟悉，因为我们这就是我们通过 truffle 控制台和 app.js 与合约交互的方式。Truffle 使用了 Mocha 测试框架和 Chai 用于断言。

File: voting.js

```
var Voting = artifacts.require("./Voting.sol");

contract('Voting', function(accounts) {
  it("should be able to buy tokens", function() {
    var instance;
    var tokensSold;
    var userTokens;
    return Voting.deployed().then(function(i) {
      instance = i;
      return i.buy({value: web3.toWei(1, 'ether')});
    }).then(function() {
      return instance.tokensSold.call();
    }).then(function(balance) {
      tokensSold = balance;
      return instance.voterDetails
        .call(web3.eth.accounts[0]);
    }).then(function(tokenDetails) {
      userTokens = tokenDetails[0];
    });
    assert.equal(balance.valueOf(), 100, "100 tokens were not sold");
    assert.equal(userTokens.valueOf(), 100, "100 tokens were not
      sold"); });

  it("should be able to vote for candidates", function() {
    var instance;
    return Voting.deployed().then(function(i) {
      instance = i;
      return i.buy({value: web3.toWei(1, 'ether')});
    }).then(function() {
      return instance.voteForCandidate('Alice', 3);
    }).then(function() {
      return instance.voterDetails
        .call(web3.eth.accounts[0]);
    }).then(function(tokenDetails) {
    assert.equal(tokenDetails[1][0].valueOf(), 3, "3 tokens were
      not used for voting to Alice");
  });
});
```



```
    });
  });
});
```

在 `test` 目录下创建一个叫做 `voting.js` 的文件，并将右侧代码拷贝进去。

我们有了两个测试，用于测试购买代币和为候选者投票的功能测试，并检测投票是否正确。

```
>truffle test test/voting.js
```

如果你对 JavaScript 和 promises 不太熟悉，你可能会觉得代码块中的 `return` 语句有点看不懂。实际上，当这些代码成功执行后，返回值会进入 `then` 代码块。

第 14 行的 `balance` 是 13 行代码的返回值。注意我们在第 12 行并没有保存任何值，因为 `buy` 函数没有返回任何值。`20 -21` 行断言确保了售出 100 个代币，并且用户拥有这 100 个代币。当出现错误时，测试就会失败，并输出一些信息（`assert.equal` 函数的第 3 个参数）。

