深入浅出以太坊

V1.0

蓝莲花

目录

第一章 以太坊是什么？ 5

1.1 以太坊是什么？ 5

1.2 下一代区块链 5

1.3 以太坊虚拟机 5

1.4 以太坊如何工作？ 6

第二章 以太坊账户管理 7

2.1 账户 7

2.2 钥匙文件 8

2.3 创建账号 8

第三章 更新、备份、恢复账号 11

3.1 更新账号 11

3.2 账号备份和恢复 12

第四章 公有链、联盟链、私有链和网络 14

4.1 以太坊网络 14

4.2 公有链、私有链和联盟链 14

4.3 如何连接 15

4.4 更快下载区块链 17

4.5 静态节点，信任节点和启动节点 18

第五章 搭建测试网络和私有链 19

5.1 Morden 测试网 19

5.2 设置本地私有测试网 20

第六章 账户、交易核心概念及投注合约 24

6.1 外有账户 vs 合约账户 24

6.2 什么是交易？ 25

6.3 什么是信息？ 25

6.4 什么是 gas? 26

6.5 估算交易成本 27

6.6 账户交互示例 — 投注合约 28

第七章 深入浅出智能合约 31

7.1 什么是合约？ 31

7.2 以太坊高级语言 32

7.3 写合约 32

7.4 编译合约 33

7.5 创建和部署合约 36

7.6 与合约互动 36

7.7 合约元数据 37

7.8 测试合约和交易 39

第八章 如何部署、调用智能合约 40

8.1 RPC 40

8.2 惯例 40

8.3 部署合约 41

8.4 和智能合约互动 43

8.5 Web3.js 45

8.6 控制台 46

8.7 查看合约与交易 46

第九章 智能合约案例实战 47

9.1 安装 truffle 47

9.2 依赖环境 47

9.3 新建第一个项目 47

10 总结 50

10.1 写这本书的初衷 50

第一章 以太坊是什么？

1.1 以太坊是什么？

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

1.2 下一代区块链

区块链技术是比特币的底层技术，这一技术第一次被描述是在中本聪 2008 年发表的白皮书“比特币：点对点电子现金系统”中。区块链技术更多的一般性用途在原书中已经有所讨论，但直到几年后，区块链技术才作为通用术语出现。一个区块链是一个分布式计算架构，里面的每个网络节点执行并记录相同的交易，交易被分组为区块。一次只能增加一个区块，每个区块有一个数学证明来保证新的区块与之前的区块保持先后顺序。这样一来，区块链的“分布式数据库”就能和整个网络保持一致。个体用户与总账的互动（交易）受到安全的密码保护。由数学执行并编码到协议中的经济激励因素刺激着维持和验证网络的节点。

在比特币中，分布式数据库被设想为一个账户余额表，一个总账，交易就是通过比特币的转移以实现个体之间无需信任基础的金融活动。但是随着比特币吸引了越来越多开发者和技术专家的注意，新的项目开始将比特币网络用于有价代币转移之外的其他用途。其中很多都采用了“代币”的形式——以原始比特币协议为基础，增加了新的特征或功能，采用各自加密货币的独立区块链。在 2013 年末，以太坊的发明者 Vitalik Buterin 建议能够通过程序重组来运行任意复杂运算的单个区块链应该包含其他的程序。

2014 年，以太坊的创始人 Vitalik Buterin, Gavin Wood 和 Jeffrey Wilcke 开始研究新一代区块链，试图实现一个总体上完全无需信任基础的智能合约平台。

1.3 以太坊虚拟机

以太坊是可编程的区块链。它并不是给用户一系列预先设定好的操作（例如比特币交易），而是允许用户按照自己的意愿创建复杂的操作。这样一来，它就可以作为多种类型去中心化区块链应用的平台，包括加密货币在内但并不仅限于此。

以太坊狭义上是指一系列定义去中心化应用平台的协议，它的核心是以太坊虚拟机（“EVM”），可以执行任意复杂算法的编码。在计算机科学术语中，以太坊

是“图灵完备的”。开发者能够使用现有的 JavaScript 和 Python 等语言为模型的其他友好的编程语言，创建出在以太坊模拟机上运行的应用。

和其他区块链一样，以太坊也有一个点对点网络协议。以太坊区块链数据库由众多连接到网络的节点来维护和更新。每个网络节点都运行着以太坊模拟机并执行相同的指令。因此，人们有时形象地称以太坊为“世界电脑”。

这个贯穿整个以太坊网络的大规模并行运算并不是为了使运算更高效。实际上，这个过程使得在以太坊上的运算比在传统“电脑”上更慢更昂贵。然而，每个以太坊节点都运行着以太坊虚拟机是为了保持整个区块链的一致性。去中心化的一致使以太坊有极高的故障容错性，保证零停机，而且可以使存储在区块链上的数据保持永远不变且抗审查。

以太坊平台本身没有特点，没有价值性。和编程语言相似，它由企业家和开发者决定其用途。不过很明显，某些应用类型较之其他更能从以太坊的功能中获益。以太坊尤其适合那些在点与点之间自动进行直接交互或者跨网络促进小组协调活动的应用。例如，协调点对点市场的应用，或是复杂财务合同的自动化。比特币使个体能够不借助金融机构、银行或政府等其他中介来进行货币交换。以太坊的影响可能更为深远。理论上，任何复杂的金融活动或交易都能在以太坊上用编码自动且可靠地进行。除金融类应用外，任何对信任、安全和持久性要求较高的应用场景——比如资产注册、投票、管理和物联网——都会大规模地受到以太坊平台影响。

1.4 以太坊如何工作？

以太坊合并了很多对比特币用户来说十分熟悉的特征和技术，同时自己也进行了很多修正和创新。比特币区块链纯粹是一个关于交易的列表，而以太坊的基础单元是账户。以太坊区块链跟踪每个账户的状态，所有以太坊区块链上的状态转换都是账户之间价值和信息的转移。账户分为两类：

- 外部账户（EOA），由私人密码控制
- 合同账户，由它们的合同编码控制，只能由外部账户“激活”

对于大部分用户来说，两者基本的区别在于外部账户是由人类用户掌控——因为他们能够控制私钥，进而控制外部账户。而合同账户则是由内部编码管控。如果他们是被人类用户“控制”的，那也是因为程序设定它们被具有特定地址的外部账户控制，进而被持有私钥控制外部账户的人控制着。“智能合约”这个流行的术语指的是在合同账户中编码——交易被发送给该账户时所运行的程序。用户可以通过在区块链中部署编码来创建新的合约。

只有当外部账户发出指令时，合同账户才会执行相应的操作。所以合约账户不可能自发地执行诸如任意数码生成或应用程序界面调用等操作--只有受外部账户提示时，它才会做这些事。这是因为以太坊要求节点能够与运算结果保持一致，这就要求保证严格确定执行。

和比特币一样，以太坊用户必须向网络支付少量交易费用。这可以使以太坊区块链免受无关紧要或恶意的运算任务干扰，比如分布式拒绝服务（DDoS）攻击或无限循环 。交易的发送者必须在激活的“程序”每一步付款，包括运算和记忆储存。费用通过以太坊自有的有价代币，以太币的形式支付。

交易费用由节点收集，节点使网络生效。这些“矿工”就是以太坊网络中收集、传播、确认和执行交易的节点。矿工们将交易分组——包括许多以太坊区块链中账户“状态”的更新——分成的组被称为“区块”，矿工们会互相竞争，以使他们的区块可以添加到下一个区块链上。矿工们每挖到一个成功的区块就会得到以太币奖励。这就为人们带来了经济激励，促使人们为以太坊网络贡献硬件和电力。

和比特币网络一样，矿工们有解决复杂数学问题的任务以便成功地“挖”到区块。这被称为“工作量证明”。一个运算问题，如果在算法上解决，比验证解决方法需要更多数量级的资源，那么它就是工作证明的极佳选择。为防止比特币网络中已经发生的，专门硬件（例如特定用途集成电路）造成的中心化现象，以太坊选择了难以存储的运算问题。如果问题需要存储器和 CPU，事实上理想的硬件是普通的电脑。这就使以太坊的工作量证明具有抗特定用途集成电路性，和比特币这种由专门硬件控制挖矿的区块链相比，能够带来更加去中心化的安全分布。

第二章 以太坊账户管理

2.1 账户

账户在以太坊中发挥着中心作用。共有两种账户类型：外部账户（ EOAs）和合约账户。我们这里重点讲一下外部账户，以下会简称为*账户*。合约账户简称为*合约*， *在合约章节具体讨论*。把外部账户和合约账户都归入到帐户的一般概念是合理的，因为这些实体都是所谓的*状态对象*。这些实体都有状态：账户有余额，合约既有余额也有合约储存。所有账户的状态正是以太坊网络的状态，以太坊网络和每个区块一起更新，网络需要达成关于以太坊的共识。对于用户通过交易和以太坊区块链互动来说，账户是必不可少的。

如果我们把以太坊限制为只有外部账户，只允许外部账户之间进行交易，我们就会进入到“代币”系统，“代币”系统不如比特币本身有力，只能用于转移以太币。

账户代表着外部代理人（例如人物角色，挖矿节点 ，或是自动代理人）的身份。账户运用公钥加密图像来签署交易以便以太坊虚拟机可以安全地验证交易发送者身份。

2.2 钥匙文件

每个账户都由一对钥匙定义，一个私钥和一个公钥。 账户以地址为索引，地址由公钥衍生而来，取公钥的最后 20 个字节。每对私钥 /地址都编码在一个*钥匙文件*里。钥匙文件是 JSON 文本文件，可以用任何文本编辑器打开和浏览。钥匙文件的关键部分，账户私钥，通常用你创建帐户时设置的密码进行加密。钥匙文件可以在以太坊节点数据目录的 keystore 子目录下找到。确保经常给钥匙文件备份！查看*备份和恢复账号*章节了解更多。创建钥匙和创建帐户是一样的。

- 不必告诉任何人你的操作。
- 不必和区块链同步。
- 不必运行客户端。
- 甚至不必连接到网络。

当然新账户不包含任何以太币。但它将会是你的，你大可放心，没有你的钥匙和密码，没有人能进入。

转换整个目录或任何以太坊节点之间的个人钥匙文件都是安全的。

警告：请注意万一你从一个不同的节点向另一个节点添加钥匙文件， 账户的顺序可能发生改变。确保不要回复或改变手稿中的索引或代码片段。

2.3 创建账号

警告：记住密码并“备份钥匙文件<backup-and-restore-accounts>”。为了从账号发送交易，包括发送以太币，你必须同时有钥匙文件和密码。确保钥匙文件有个备份并牢记密码，尽可能安全地存储它们。这里没有逃亡路径，如果钥匙文件丢失或忘记密码，就会丢失所有的以太币。没有密码不可能进入账号，也没有*忘记密码*选项。所以一定不要忘记密码。

使用 geth account new

一旦安装了 geth 客户端，创建账号就只是在终端执行 geth account new 指令的问题了。

注意不必运行 geth 客户端或者和区块链同步来使用 geth account 指令。

```
$ geth account new

Your new account is locked with a password. Please give a password. Do
not forget this password.

Passphrase:
```

```
Repeat Passphrase:

Address: {168bc315a2ee09042d83d7c5811b533620531f67}
```

对于非交互式使用，你可以提供纯文本密码文件作为--password 标志的变元。文件中的数据包含密码的原始字节，后面可选择单独跟着新的一行。

```
$ geth --password /path/to/password account new
```

警告：用--password 标志只是为了测试或在信任的环境中自动操作。不建议将密码保存在文件中或以任何其他方式暴露。如果你用密码文件来使用--password 标志，要确保文件只对你自己可阅读和列表。你可以在 Mac/Linux 系统中通过以下指令实现：

```
touch /path/to/password

chmod 600 /path/to/password

cat > /path/to/password

>I type my pass
```

要列出目前在你的 keystore 文件夹中的钥匙文件的所有账号，使用 geth account 指令的 list 子指令：

```
$ geth account list

account #0: {a94f5374fce5edbc8e2a8697c15331677e6ebf0b}

account #1: {c385233b188811c9f355d4caec14df86d6248235}

account #2: {7f444580bfef4b9bc7e14eb7fb2a029336b07c9d}
```

钥匙文件的文件名格式为 UTC--<created_at UTC ISO8601>-<address hex>。账号列出时是按字母顺序排列，但是由于时间戳格式，实际上它是按创建顺序排列。

使用 geth 控制台

为了用 geth 创建新账号，我们必须先在控制台模式开启 geth（或者可以用 geth attach 将控制台依附在已经运行着的事例上）：

```
> geth console 2>> file_to_log_output
```

```
instance: Geth/v1.4.0-unstable/linux/go1.5.1

coinbase: coinbase: [object Object]

at block: 865174 (Mon, 18 Jan 2016 02:58:53 GMT)

datadir: /home/USERNAME/.ethereum
```

控制台使你能够通过发出指令与本地节点互相作用。比如，试一下这个列出账号的指令：

```
> eth.accounts

{

code: -32000,

message: "no keys in store"

}
```

这就表明你没有账号。你也可以从控制台创建一个账号：

```
> personal.newAccount()

Passphrase:

Repeat passphrase:

"0xb2f69ddf70297958e582a0cc98bce43294f1007d"
```

> 注意：记得用一个安全性强、随机生成的密码。

我们刚刚创建了第一个账号。如果我们再次试着列出账号，就可以看到新创建的账号了。

```
> eth.accounts

["0xb2f69ddf70297958e582a0cc98bce43294f1007d"]
```

使用 Mist 以太坊钱包

对于相反的命令行，现在有一个基于 GUI 的选项可以用来创建账号：“官方”Mist 以太坊钱包。 Mist 以太坊钱包，和它的父项目 Mist, 是在以太坊基金会的赞助下开发，因此是“官方”地位。钱包应用有 Linux, Mac OS X 和 Windows 可用的版本。

> 警告：Mist 钱包是试用软件，使用需风险自担。

用 GUI Mist 以太坊钱包创建账号再容易不过了。事实上，第一个账号在应用安装期间就创建出来了。

1.根据你的操作程序下载钱包应用最新版本。由于你实际上会运行一个完整的 geth 节点，打开钱包应用就会开始同步复制你电脑上的整个以太坊区块链。

2. 解锁下载的文件夹，运行以太坊钱包可执行文件。

3. 等待区块链完全同步，按照屏幕上的说明操作，第一个账号就创建出来了。

4. 第一次登录 Mist 以太坊钱包，你会看到自己在安装过程中创建的账号。它会被默认命名为主账号（以太库）
5. 再另外创建账号很容易；只需点击应用主界面上的添加账号，输入所需的密码即可。

> 注意： Mist 钱包仍在开发中，以上列出的具体步骤可能会随着更新有所变更。

第三章 更新、备份、恢复账号

3.1 更新账号

你可以把钥匙文件更新到最新的钥匙文件格式并且/或者升级钥匙文件密码。

使用 geth

你可以在命令行用更新子命令更新现在的账号，可以使用账号地址或者索引作为参数。记住账号索引反映了创建顺序（按字母顺序排列的钥匙文件名包含了创建时间）。

```
geth account update b0047c606f3af7392e073ed13253f8f4710b08b6
```

或者

```
geth account update 2
```

例如：

```
$ geth account update a94f5374fce5edbc8e2a8697c15331677e6ebf0b
Unlocking account a94f5374fce5edbc8e2a8697c15331677e6ebf0b | Attempt 1/3
Passphrase:
0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b
account 'a94f5374fce5edbc8e2a8697c15331677e6ebf0b' unlocked.
Please give a new password. Do not forget this password.
Passphrase:
Repeat Passphrase:
0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b
```

账户以加密的形式储存在最新版本，它会提示你需要一个密码来解锁账户，另一个密码来保存更新的文件。同一个指令还可以用在将弃用格式的账户变成最新版本或者改变账户密码。
对于非交互式使用，密码可以用 —password 标志详细说明：

```
geth --password <passwordfile> account update
a94f5374fce5edbc8e2a8697c15331677e6ebf0bs
```

由于只能给出一个密码，所以只能执行格式更新，修改密码只在交互式的情况下才有可能。

注意：账号更新有个副作用就是会引起账号顺序变化。更新成功后，同一钥匙所有之前的格式/版本都会被移除！

3.2 账号备份和恢复

手动备份/恢复

要从账号发送交易，需要有账号钥匙文件。钥匙文件可以在以太坊节点数据目录的钥匙商店（keystore）子目录下找到。默认数据目录的位置与平台相关：

- Windows: C:\Users\username\%appdata%\Roaming\Ethereum\keystore
- Linux: ~/.ethereum/keystore
- Mac: ~/Library/Ethereum/keystore

要备份钥匙文件（账号），在 keystore 子目录中复制单独的钥匙文件或复制整个 keystore 文件夹。

要恢复钥匙文件（账号），将钥匙文件重新复制到 keystore 子目录，即其原始地址。

导入未加密私钥

导入未加密私钥由 geth 支持

```
geth account import /path/to/<keyfile>
```

这个指令从纯文本文件<keyfile>导入未加密私钥并创建新账号和打印地址。钥匙文件被假定包含未加密私钥作为编码到十六进制的标准 EC 原始字节。账号以加密的形式储存，会提示你输入密码。你需要记住密码用于以后解锁账号。

下面给出一个例子，详细说明数据目录。如果 —datadir 标志没有使用，新账户就会被创建在默认数据目录里，例如钥匙文件会被放在数据目录的钥匙文件子目录里。

```
$ geth --datadir /someOtherEthDataDir account import ./key.prv
The new account will be encrypted with a passphrase.
Please enter a passphrase now.
Passphrase:
Repeat Passphrase:
Address: {7f444580bfef4b9bc7e14eb7fb2a029336b07c9d}
```

对于非交互式使用，密码可以用 —password 标志详细说明：

```
geth --password <passwordfile> account import <keyfile>
```

注意：因为你可以直接把加密账户复制到另一个以太坊事例中，在节点之间转移账号的时候就不需要这个导入/导出机制了。

警告：当你往已存在节点的 keystore 里复制钥匙的时候，你习惯的账户顺序可能会改变。因此要保证你不依赖于账户顺序，否则就要进行复核并更新脚本中使用的索引。

第四章 公有链、联盟链、私有链和网络

4.1 以太坊网络

去中心化共识的基础是参与节点的点对点网络，节点维持和保护着区块链（维护并保证区块链网络的安全）。参见*挖矿*。

以太坊网络数据

EthStats.net 是以太坊网络实时数据的仪表板，这个仪表板展示重要信息，诸如现在的区块，散表难度，gas 价格和 gas 花费等。页面上显示的节点只是精选了网络上的实际节点。任何人都可以在 EthStats 仪表板上添加他们的节点。Github 上的 Eth-Netstats README 描述了如何连接。

EtherNodes.com 展示了节点数的当前和历史数据以及以太坊主网络和 Morden 测试网络上的其他信息。

当前实时网络上客户端实现分配 - EtherChain 上的实时数据。

4.2 公有链、私有链和联盟链

当今大多数以太坊项目都依靠以太坊作为公有链，公有链可以访问到更多用户，网络节点，货币和市场。然而通常有理由更偏好私有链或联盟链（在一群值得信任的参与者中）。例如，银行领域的很多公司都希望以太坊作为他们私有链的平台。

以下是博客发文《关于公有链和私有链》的摘录，它解释了三种区块链在许可方面的区别：

• 公有链：世界上所有人都可以阅读和发送交易。如果他们合法都有希望看到自己被包括在内。世界上任何人都能参与到共识形成过程——决定在链条上添加什么区块以及现状是怎样的。作为中心化或准中心化信任的替代品，公有链受加密经济的保护，加密经济是经济激励和加密图形验证的结合，用类似工作量证明或

权益证明的机制，遵循的总原则是人们影响共识形成的程度和他们能够影响的经济资源数量成正比。这类区块链通常被认为是“完全去中心化“。

• 联盟链：共识形成过程由预先选择的一系列的节点所掌控，例如，设想一个有15 个金融机构的团体，每个机构都操作一个节点，为了使区块生效，其中的 10 个必须签署那个区块。阅读区块链的权利可能是公开的，或仅限于参与者，也有混合的路径，比如区块的根散表和应用程序编程接口一起公开，使公共成员可以进行一定量的查询，重获一部分区块链状态的加密图形证明。这类区块链被认为是“部分去中心化”。

• 私有链：书写许可对一个组织保持中心化。阅读许可可能是公开的或者限制在任意程度。应用很可能包含对单个公司内部的数据库管理，审查等，因此公共的可读性在很多情况下根本不必要，但在另一些情况下人们又想要公共可读性。

私有链/联盟链可能和公有链毫无联系，他们仍然通过投资以太坊软件开发，对以太坊整体生态系统有利。经过一段时间，这会转变成软件改善，知识共享和工作机会。

4.3 如何连接

Geth 会持续尝试在网络上连接到其他节点，直到有了端点为止。如果你在路由器上有可用的 UPnP 或者在面向因特网的服务器上运行以太坊，它也会接受其他节点的连接。

Geth 通过*发现协议*找到对等端。在发现协议中，节点互相闲聊发现网络上的其他节点。最开始，geth 会使用一系列辅助程序节点，这些辅助程序节点的端点记录在源代码中。

检查连接和 ENODE 身份

要检查客户端在交互控制台上连接了多少对等端点，net 模块有两个属性可以提供信息，告诉你对等端点的数量以及你是否在监听的节点。

```
> net.listening
true
> net.peerCount
4
```

了解更多关于连接对等端点的信息，比如 IP 地址、端口号和支持协议，用管理员对象的 peers()功能。admin.peers()会返回到现在已连接的对等端点列表。

```
> admin.peers
```

```
[{
ID:
'a4de274d3a159e10c2c9a68c326511236381b84c9ec52e72ad732eb0b2b1a2277938
f78593cdbe734e6002bf23114d434a085d260514ab336d4acdc312db671b',
Name: 'Geth/v0.9.14/linux/go1.4.2',
Caps: 'eth/60',
RemoteAddress: '5.9.150.40:30301',
LocalAddress: '192.168.0.28:39219'
}, {
ID:
'a979fb575495b8d6db44f750317d0f4622bf4c2aa3365d6af7c284339968eef29b69
ad0dce72a4d8db5ebb4968de0e3bec910127f134779fbcb0cb6d3331163c',
Name: 'Geth/v0.9.15/linux/go1.4.2',
Caps: 'eth/60',
RemoteAddress: '52.16.188.185:30303',
LocalAddress: '192.168.0.28:50995'
}, {
ID:
'f6ba1f1d9241d48138136ccf5baa6c2c8b008435a1c2bd009ca52fb8edbbc991eba3
6376beaee9d45f16d5dcbf2ed0bc23006c505d57ffcf70921bd94aa7a172',
Name: 'pyethapp_dd52/v0.9.13/linux2/py2.7.9',
Caps: 'eth/60, p2p/3',
RemoteAddress: '144.76.62.101:30303',
LocalAddress: '192.168.0.28:40454'
}, {
ID:
'f4642fa65af50cfdea8fa7414a5def7bb7991478b768e296f5e4a54e8b995de102e0
ceae2e826f293c481b5325f89be6d207b003382e18a8ecba66fbaf6416c0',
Name: '++eth/Zeppelin/Rascal/v0.9.14/Release/Darwin/clang/int',
Caps: 'eth/60, shh/2',
RemoteAddress: '129.16.191.64:30303',
LocalAddress: '192.168.0.28:39705'
} ]
```

要检查 geth 使用的端口，发现你自己的 enode URI 执行：

```
> admin.nodeInfo
{
Name: 'Geth/v0.9.14/darwin/go1.4.2',
NodeUrl:
'enode://3414c01c19aa75a34f2dbd2f8d0898dc79d6b219ad77f8155abf1a287ce2
ba60f14998a3a98c0cf14915eabfdacf914a92b27a01769de18fa2d049dbf4c17694@
```

```
[::]:30303',
NodeID:
'3414c01c19aa75a34f2dbd2f8d0898dc79d6b219ad77f8155abf1a287ce2ba60f149
98a3a98c0cf14915eabfdacf914a92b27a01769de18fa2d049dbf4c17694',
IP: '::',
DiscPort: 30303,
TCPPort: 30303,
Td: '2044952618444',
ListenAddr: '[::]:30303'
}
```

4.4 更快下载区块链

启动以太坊客户端时，会自动下载以太坊区块链。用于下载以太坊区块链的时间会根据客户端、客户端设置、连接速度和可用的端点数量变化。下面是更快获取以太坊区块链的一些选项。

使用 geth

如果你在用 geth 客户端，你可以做些什么来加速下载以太坊区块的时间。如果你用—fast 标志来执行以太坊快速同步，不会保留过去的交易数据。

注意：你不能在执行所有或者部分正常的同步操作之后再使用这个标志，也就是说在用这个指令之前，不能下载以太坊区块链的任何部分。查看这个 Ethereum Stack.Exchange answer 了解更多。

下面是想要更快同步客户端时使用的一些标志。

--fast
这个标志使通过状态下载而不是下载整个区块数据来实现快速同步成为可能。这样也能大幅减少区块链尺寸。注意：--fast 只在从头开始同步区块链，并且是出于安全原因第一次下载区块链时，才会运行。查看 Reddit 发文了解更多。

--cache=1024
分配到内部缓存的千兆内存（最少 16MB / 数据库）。默认是 16MB，所以根据你电脑内存多少，增加到 256, 512, 1024 (1GB)或者 2048 (2GB)会带来不同。

--jitvm
这个标志可以激活 JIT VM。
完整的控制台命令示例：

```
geth --fast --cache=1024 --jitvm console
```

了解更多关于快速同步和区块链下载次数的讨论，查看这篇 Reddit 发文。

导出/导入区块链

如果你已经同步了整个以太坊节点，可以从完全同步的节点中导出区块链数据并将其导入新节点。你可以在 geth 中用 geth export filename 指令导出所有节点，并用 geth import filename 将区块链导入节点，来实现这一目的。

4.5 静态节点，信任节点和启动节点

Geth 支持一个叫静态节点的特征，如果你有特定的端点，你会一直想与静态节点连接。如果断开连接，静态节点会再次连接。你可以配置永久性静态节点，方法是将如下所说的放进<datadir>/static-nodes.json（这应该是和 chaindata 以及 keystone 在同一个文件夹）

```
[
"enode://f4642fa65af50cfdea8fa7414a5def7bb7991478b768e296f5e4a54e8b995de102e0ceae2e826f293c481b5325f89be6d207b003382e18a8ecba66fbaf6416c0@33.4.2.1:30303",
"enode://pubkey@ip:port"
]
```

你也可以在运行期间通过 Javascript 使用 admin.addPeer()加入静态节点。

```
>
admin.addPeer("enode://f4642fa65af50cfdea8fa7414a5def7bb7991478b768e296f5e4a54e8b995de102e0ceae
```

连接的常见问题

有时候可能无法连接，最常见的原因有：

- 本地时间不正确。要参与到以太坊网络中，需要精确的时钟。检查 OS 如何同步时钟（例如 sudo ntpdate -s time.nist.gov），即便只快了 12 秒也有可能导致 0 端点。

• 有的防火墙配置可能会阻止 UDP 流通。可以用静态节点功能或者控制台上的 admin.addPeer() 来手动配置连接。

不使用发现协议来启动 geth，你可以用—nodiscover 参数。你只会在运行测试节点或有固定节点的实验测试网络时才想要这样做。

第五章 搭建测试网络和私有链

5.1 Morden 测试网

Morden 是公开的以太坊替代测试网。它会贯穿于整个软件里程碑 Frontier 和 Homestead。

用法

eth（C++客户端） 0.9.93 及以上版本自动支持。比如开启以下任意客户端时，通过—morden 参数。

PyEthApp（Python 客户端） PyEthApp 支持 v1.0.5 以后的 morden 网络。

geth（Go 客户端）

细节

除以下几条，所有参数都和主要的以太坊网络相同：
• 网络名称：**Morden**
• 网络身份：2
• genesis.json（如下）；
• 初始账户随机数（IAN）是 2^20 （不像之前的网络中是 0）
  - 状态树形结构中的所有账户都有随机数>= IAN。
  - 账户被插入到状态树形结构中时，都会被赋予一个初始随机数= IAN。
• 初始通用区块散表：
0cd786a2425d16f152c658316c423e6ce1181e15c3295826d7c9904cba9ce303
• 初始通用状态根：
f3f4696bbf3b3b07775128eb7a3763279a394e382130f27c21e70233e04946a9

Morden 的 genesis.json

获取 Morden 测试网以太币

有两种方法可以获取 Morden 测试网以太币：
• 用 CPU/GPU 挖矿（参见*挖矿*）。
• 用以太坊 wei 龙头。

5.2 设置本地私有测试网

eth (C++ 客户端)

可以使用 - genesis 和 - config 连接到或创建一个新的网络。

可以同时使用 - config 和 - genesis。

那样的话， - config 提供的初始区块描述会被 - genesis 选项覆盖。

注意：<filename>包含一个网络的 JSON 描述。

• sealEngine (用来在区块挖矿的引擎)
“Ethash”是以太坊工作量证明引擎(用于实时网络)。
“NoProof” 在区块挖矿不需要工作量。
• params (诸如 minGasLimit, minimumDifficulty, blockReward, networkID 等一般的网络信息)
• genesis (初始区块描述)
• accounts (设置包含账户/合约的初始状态)

这是一个 Config 的例子(用于 Olympic 网络)：

注意：<filename>包含一个网络的 JSON 描述。

内容与’config’参数提供的初始领域相同。

geth (Go 客户端)

你可以在私有测试网上生成或挖掘自己的以太币。这个试验以太坊方法很划算，可以避免不得不挖矿，或找到 Morden 测试网络的以太币。

在私有链中需要详细说明的事件有：

- 定制初始文件
- 定制数据目录
- 定制网络 ID
- （推荐）废弃节点发现

初始文件

初始区块是区块链的起始 — 第一个区块，区块 0，唯一没有指向前面区块的一个区块。协议确保其他节点不会和你的区块链一致，除非他们和你有相同的初始区块，这样你想创建多少私有测试网区块链，就可以创建多少！

```
{
"nonce": "0x0000000000000042", "timestamp": "0x0",
"parentHash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
"extraData": "0x0", "gasLimit": "0x8000000", "difficulty": "0x400",
"mixhash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
"coinbase": "0x3333333333333333333333333333333333333333", "alloc": { }
}
```

存储文件为 CustomGenesis.json。用下面的标志启动 geth 节点的时候，你会引用到这个。

```
--genesis /path/to/CustomGenesis.json
```

私有网络的命令行参数

有一些必需的命令行选项（又称为“标志”）来确保你的网络是私有的。我们已经谈到了初始标志，下面还有几个。注意所有下面的指令都会用在 geth 以太坊客户端。

--nodiscover
使用这个命令可以确保你的节点不会被非手动添加你的人发现。否则，你的节点可能被陌生人的区块链无意添加，如果他和你有相同的初始文件和网络 ID。

--maxpeers 0
如果你不希望其他人连接到你的测试链，可以使用 maxpeers 0。反之，如果你确切知道希望多少人连接到你的节点，你也可以通过调整数字来实现。

--rpc
这个指令可以激活你节点上的 RPC 界面。它在 geth 中通常被默认激活。

--rpcapi "db,eth,net,web3"
这个命令可以决定允许什么 API 通过 RPC 进入。在默认情况下，geth 可以在 RPC 激活 web3 界面。

重要信息：请注意在 RPC/IPC 界面提供 API，会使每个可以进入这个界面（例如 dapp’ s）的人都有权限访问这个 API。注意你激活的是哪个 API。Geth 会默认激活 IPC 界面上所有的 API，以及 RPC 界面上的 db,eth,net 和 web3 API。

--rpcport "8080"
将 8000 改变为你网络上开放的任何端口。Geth 的默认设置是 8080.

--rpccorsdomain "http://chriseth.github.io/browser-solidity/"
这个可以指示什么 URL 能连接到你的节点来执行 RPC 定制端任务。务必谨慎，输入一个特定的 URL 而不是 wildcard (*)，后者会使所有的 URL 都能连接到你的 RPC 实例。

--datadir "/home/TestChain1"
这是你的私有链数据所储存在的数据目录（在 nubits 下）。选择一个与你以太坊公有链文件夹分开的位置。

--identity "TestnetMainNode"
这会为你的节点设置一个身份，使之更容易在端点列表中被辨认出来。这个例子说明了这些身份如何在网络上出现。

发布 geth

你创建了定制初始区块 JSON 并建立区块链数据目录后，在控制台输入以下指令，进入 geth：

```
geth --identity "MyNodeName" --genesis /path/to/CustomGenesis.json --rpc
--rpcport "8080" --rpcco
```

注意：请改变标志与定制设置匹配。

每次想要进入定制链的时候，你都需要用定制链指令启动 geth 实例。如果你只在控制台输入“geth”，它不会记住你设置的所有标志。

给账户预分配以太币

“0x400”难度能让你再私有测试网链上快速挖以太币。如果你创建了自己的链，开始挖矿，你应该几分钟就会有上百个以太币，远远超过了在网络上测试交易所需的数量。如果你还想给账户预分配以太币，就需要：

1. 创建私有链以后再创建新的以太坊账户。
2. 复制新的账户地址。
3. 在 Custom_Genesis.json 文件中添加以下指令：

```
"alloc":
{
"<your account address e.g. 0x1fb891f92eb557f4d688463d0d7c560552263b5a>":
{ "balance": "20000000000000000000" }
}
```

注意：用你的账户地址取代 0x1fb891f92eb557f4d688463d0
d7c560552263b5a

保存初始文件，重新运行私有链指令。Geth 完整装载以后，关闭它。

我们想指派一个地址给变量 primary，查看它的余额。

在终端运行 geth account list 指令，查看指派给你的新地址账户号码是什么。

```
> geth account list
Account #0: {d1ade25ccd3d550a7eb532ac759cac7be09c2719}
Account #1: {da65665fc30803cb1fb7e6d86691e20b1826dee0}
Account #2: {e470b1a7d2c9c5c6f03bbaa8fa20db6d404a0c32}
Account #3: {f4dd5c3794f1fd0cdc0327a83aa472609c806e99}
```

记录你预分配以太币的账户号码。或者，可以用 geth console（和最先启动 geth 时保持一样的参数）启动控制台。提示出现以后，输入

```
> eth.accounts
```

这会返回到你拥有的账户地址排列。

```
> primary = eth.accounts[0]
```

注意：用你的账户指数取代 0，这个控制台指令会返回到你第一个以太坊地址。

输入以下指令：

```
> balance = web3.fromWei(eth.getBalance(primary), "ether");
```

这应该会返回到 7.5，意味着你账户里有那么多以太币。我们必须在你初始文件的分区里放那么多数量是因为“余额”领域以 wei 为单位取一个数字，wei 是以太坊货币以太币的最小面额（参见以太币）。

- https://www.reddit.com/r/ethereum/comments/3kdnus/question_about_private_chain_mining_dont_upvote/
- http://adeduke.com/2015/08/how-to-create-a-private-ethereum-chain/

第六章 账户、交易核心概念及投注合约

6.1 外有账户 vs 合约账户

以太坊中有两种类型的账户

- 外有（外部）账户
- 合约账户

它们的区别在 Serenity 版本中可能会消失。

外有账户（EOA）

外有账户

- 有以太币余额，
- 可以发送交易（以太币交易或引发合约代码），
- 由私钥控制，
- 没有相关代码。

合约账户

合约

- 有以太币余额，
- 有相关代码，
- 代码执行由从其他合约接收的交易或信息（调用）触发，
- 执行的时候—执行任意复杂的操作（图灵完备的）—操控它自己的永久存储，例如，可以有自己的持久状态—可以调用其他合约

以太坊区块链上的所有行为都由外有账户引发的交易调动。每次合约账户接收到交易时，它的代码都按照输入参数的指示执行，作为交易的一部分发送。合约代码由参与到网络的每个节点上的以太坊虚拟机执行，作为验证新区块的一部分。

这个执行需要是完全确定性的，它唯一的语境是区块链上区块的位置和所有可见的数据。区块链上的区块代表时间单位，区块链本身是时间维度，代表在链上区块指定的离散的时间点上状态的整个历史。

所有的以太币余额和价值都以 wei 为单位命名：1 个以太币是 1e18 wei。

注意：以太坊中的“合约”不应该被看做是要“实现”或“遵守”的事物；它更像是在以太坊执行环境中生存的“自治代理”，被信息或交易“戳到”的时候，总会执行特定的代码片段，并且对自己的以太币余额和钥匙/价值商店有直接的控制，以储存永久状态。

6.2 什么是交易？

“交易”这个术语用在以太坊中来指代签署的数据包，数据包存储着要从外有账户发送到区块链上另一账户的信息。

交易包括：

- 信息接收人，
- 一个签字，用以确认发送方身份，证明通过区块链向接收者发送信息的意图，
- VALUE 域—从发送方向接收方转移的 wei 的数量，
- 可选数据域，包括发送到合约的信息，
- 一个 STARTGAS 值，代表交易执行允许采取的运算步骤的最大数量，
- 一个 GASPRICE 值，代表发送人愿意支付的 gas 费用。一个 gas 单位对应着一个原子指令执行，比如运算步骤。

6.3 什么是信息？

合约能够向其他合约发送“信息”。信息是虚拟的事物，永远不能序列化，只存在于以太坊执行环境中。他们可以被想象为功能调用。

信息包括：

- 信息发送方（内含的）
- 信息接收方
- VALUE 域—和发送到合约地址的信息一起转移的 wei 数量，
- 可选数据域，即发送到合约的实际数据
- 一个 STARTGAS 值，限制了信息可以触发的代码执行的 gas 最大值。

本质上来说，一个信息就像一个交易，只不过信息是由合约而不是由外在因素创造的。当正在执行代码的合约执行 CALL 或 DELEGATECALL 操作码时，信息就产生了。和交易一样，信息可能会导致接收方账户运行代码。因此，就像和外在因素建立关系一样，合约也能以同样的方式和其他合约建立关系。

6.4 什么是 gas?

以太坊虚拟机(EVM)是以太坊区块链上的可执行环境。每个参与到网络的节点运行 EVM，作为区块验证协议的一部分。他们检查列在区块上的、他们验证的交易，运行 EVM 内部交易触发的代码。每个网络上的完整节点进行同样的运算，储存相同的值。很明显以太坊并不是关于运算效率的优化。它类似的进程是多余的。它的目的是提供一个有效的方式，在系统状态上不需要信任的第三方、准则或暴力垄断就能达成共识。但重要的是他们不是为了优化运算而存在。事实上，合约执行在节点之间被多余地重复，自然使之更昂贵，通常会鼓励人们如果能在链外操作运算，就不在区块链里进行。

运行去中心化应用(dapp)时，它和区块链互动来阅读和修正状态，但是去中心化应用会很典型地只把业务逻辑和对共识至关重要的状态放在区块链上。

当信息或交易触发结果执行时，每个指令在每个网络节点被执行。这有一个代价：每个执行的操作都有特定的成本，以一定量的 gas 单元表现。

Gas 是交易发送方需要为每个以太坊区块链上发生的操作所支付的执行花费。这个名字 gas 的灵感来自这样一个观点，这笔花费就像加密燃料，驱使智能合约产生。Gas 从执行代码的矿工处购买以获得以太币。Gas 和以太币被故意分开，因为 gas 单位与具备自然成本的运算单位一致，而以太币的价格通常会由于市场力量产生波动。二者由自由市场调和：gas 价格实际上由矿工决定，矿工可以拒绝以低于最低限度的 gas 价格进行交易。为了获得 gas，你只需要向账户中添加以太币。以太坊客户端会自动为你的以太币购买 gas，数量是你指定的交易最大支出。

在合约或交易执行的每个运算步骤，以太坊协议都要收费，以防止以太坊网络上发生蓄意攻击或滥用。每个交易都必须包含一个 gas 限度和每 gas 愿意支付的花费。矿工可以选择是否将交易包括在内和收集花费。如果交易产生的、用于运算步骤的 gas 总量，包括原始信息和可能引发的子信息，少于或等于 gas 限额，那么交易就会进行。如果 gas 总量超过 gas 限额，那么所有的变化都复原，但是交易仍然有效，矿工仍然可以收集花费。未用于交易执行的、所有多余的 gas 都会以以太币的形式偿还给发送方。不必担心超支，因为只会对你消费的 gas 收费。这意味着以高于预估的 gas 限额发送交易也是有效和安全的。

6.5 估算交易成本

交易花费的以太币总量基于两个因素：
gasUsed 是交易消费的 gas 总量
gasPrice 交易中指定的一个 gas 单元的价格(换算成以太币)
总成本= gasUsed * gasPrice

gasUsed

以太坊虚拟机上的每个操作都会被指派消费的 gas 数量。gasUsed 是所有执行的操作所需的 gas 总额。有个电子表格可以看到背后的一些统计。
对于估算 gasUsed，可以用 estimate Gas API 但是有些警告说明。

gasPrice

用户建构并签署交易，每个用户可以说明自己想要的 gasPrice，可以是零。然而 Frontier 发布的以太坊客户端默认 gasPrice 是 0.05e12 wei。由于矿工会使收入最优化，如果大部分交易都以 0.05e12 wei 的 gasPrice 提交，就很难说服矿工接受价格更低或为 0 的交易。

示例交易成本

我们来做一个只添加 2 个数字的合约。EVM OPCODE ADD 消费 3 gas。
大概的成本，以默认 gas 价格计算 (2016 年 1 月)是：
3 * 0.05e12 = 1.5e11 wei
1 以太币是 1e18 wei，总成本就是 0.00000015 以太币。
这是个简化的计算，因为忽略了一些成本，比如将 2 个数字转移给合约的成本，在他们可以被添加之前。

- question
- gas 费用
- gas 成本计算器
- 以太坊 Gas 价格

操作名称	gas 成本	备注
step	1	每个执行循环的默认数量
stop	0	免费
suicide	0	免费
sha3	20	
sload	20	移出永久存储

sstore	100	放进永久存储
balance	20	
create	100	合约创建
call	20	发起一个只读调用
memory	1	扩展内存时每个额外的词
txdata	5	一个交易的每个数据字节或编码
transaction	500	基础费用交易
contract creation	53000	在 homestead 中从 21000 变化而来

6.6 账户交互示例 — 投注合约

之前提到过，有两种类型的账户：
• 外有账户 (EOA)：由私钥控制的账户，如果你有和外有账户相关的私钥，就能从账户发送以太币和信息。
• 合约： 有自己代码的账户，受代码控制。

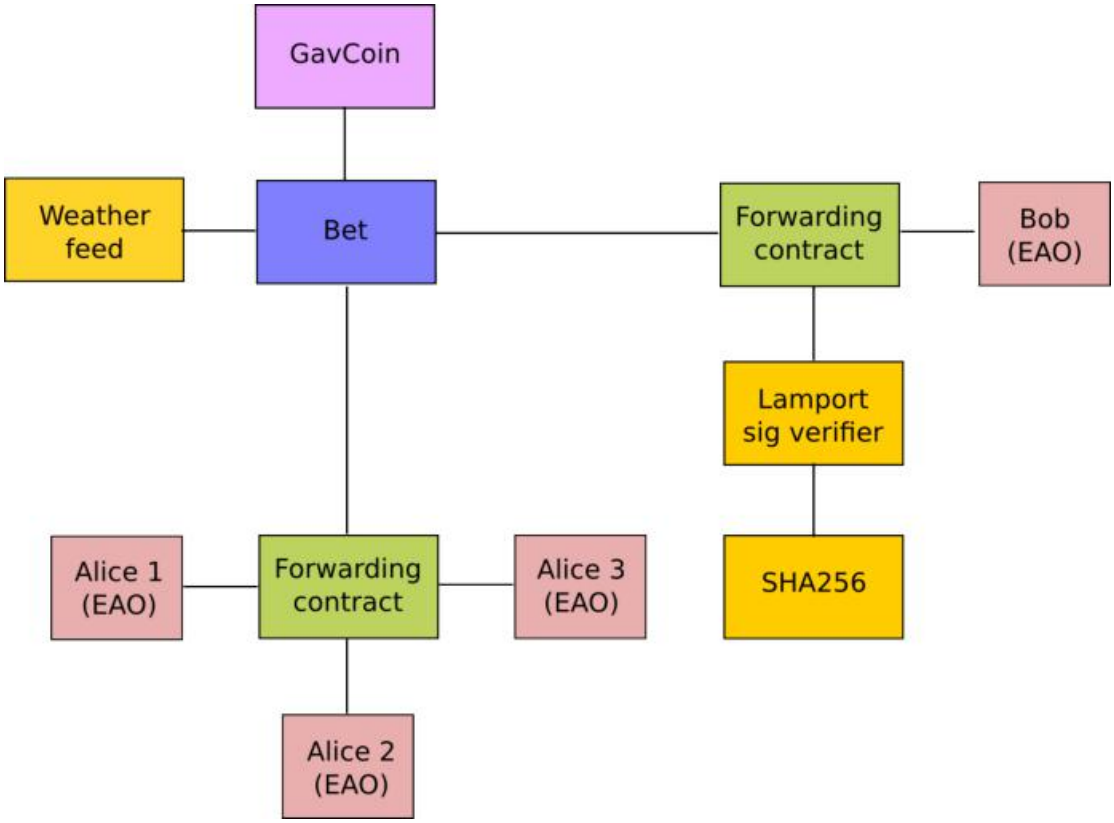
以太坊默认的执行环境是没有生命的，什么都不会发生，每个账户的状态保持相同。但是，每个用户都可以通过从外有账户发送交易来触发行动，启动以太坊。如果交易的目的地是其他外有账户，交易可能会转移一些以太币，否则什么也不会做。但如果目的地是个合约， 反之合约会激活，自动运行代码。
代码有能力读/写自己的内部存储(一个将 32 字节钥匙映射到 32 字节价值的数据库)，阅读存储的接收信息，给其他合约发送信息，转而触发执行。一旦执行停止，合约发送的信息所触发的所有的子执行都会停止(这些都以决定好的同步的顺序发生，比如，子调用在父调用进一步之前完全完成)，执行环境立即再次停止，直到被下一个交易唤醒。

合约通常服务于四个目的：
• 保持数据库代表着对其他合约或外部世界有用的东西；一个例子是合约激励货币，另一个例子是合约在特定的组织里记录会员。

• 作为某种具有更复杂访问政策的外有账户，被称为“前向合约”，典型地只在特定条件下，把进来的信息转发给到指定目的地；例如，前向合约可能会等到指定 3 个私钥中的 2 个都确认了特定的信息之后才会进行转发(例如，多重签名)。更复杂的前向合约基于要发送的信息会有不同的条件。最简单的一个功能使用案例就是撤回限制，在一些更复杂的访问政策中难以驾驭。钱包合约就是个很好的例子。
• 在多个用户之间管理一个正在进行的合约或关系。例子包括金融合约，有特定中介的第三方保管合约，或一些保险。也可以是开放合约，一方对其他方的随时参与保持开放；一个例子是自动给提交数学问题有效解决方案或是证明提供了一些运算资源的人发奖金的合约。
• 给其他合约提供功能，本质上作为软件库。
合约通过被交替叫做“调用”或“发送信息”的活动进行互动。“信息”是包含一定量以太币，任何大小的数据字节串，发送方和接收方地址的事物。合约接收信息时，可以选择返还一些数据，信息本来的发送方可以立即使用。 这样发送信息就和调用一个功能一样。

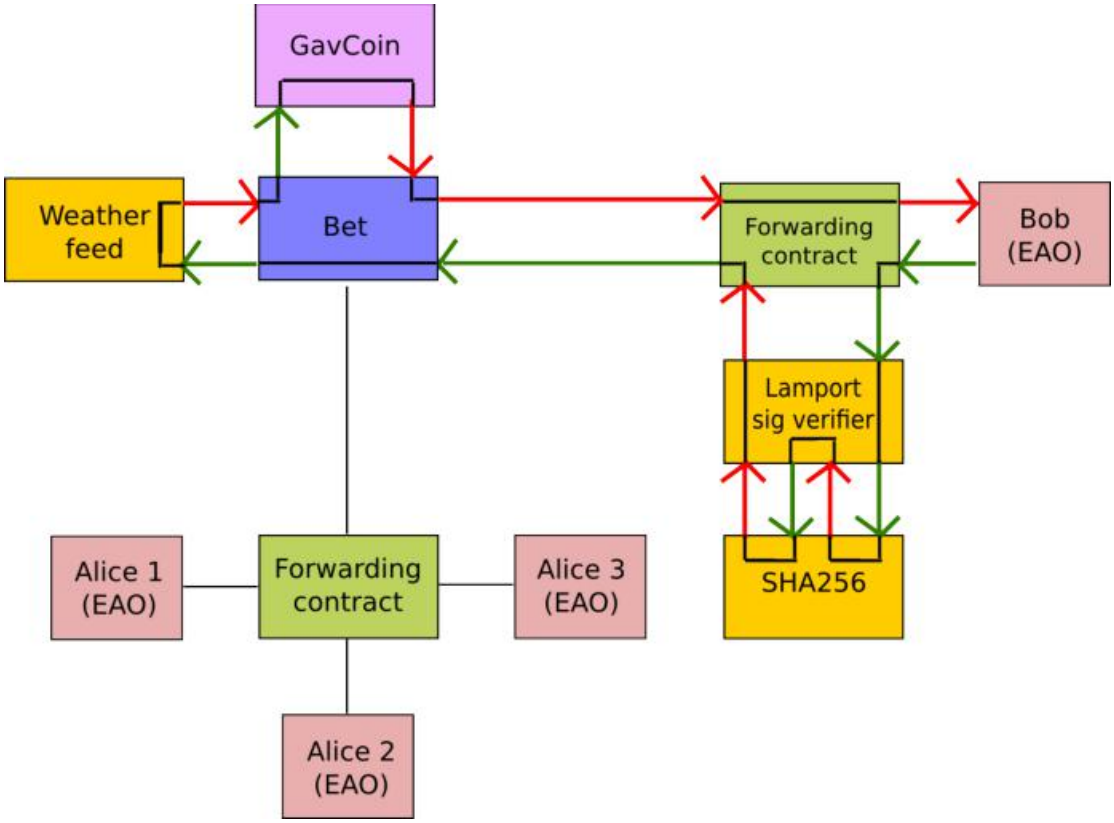
因为合约有这样的作用，我们期望合约可以彼此互动。举个例子，设想一个情景，Alice 和 Bob 赌 100 Gav 币，明年旧金山的温度不会超过 35°C。但是 Alice 非常有安全意识，她的第一个账户使用的前向合约，只有在 3 个私钥中的 2 个都批准的情况下才可以发送信息。Bob 偏执于量子加密图形，他使用的前向合约，只传递有 Lamport 签名和传统 ECDSA 的信息(但是因为他很老派，所以更偏向于用基于 SHA256 的 Lamport 签名版本，以太坊不直接支持)。

投注合约本身需要从一些合约中取得旧金山天气的数据，当它想要实际发送 Gav 币给 Alice 或 Bob 时，也需要和 Gav 币合约交谈(或者，更准确地说，Alice 或 Bob 的前向合约)。因次我们可以这样表示账户之间的关系：



Bob 想最终决定赌注的时候，就会发生以下的步骤：
1. 交易被发出，触发信息从 Bob 的外有账户发送到他的前向合约。
2. Bob 的前向合约给合约发送信息散表和 Lamport 签名，发挥 Lamport 签名确认库的作用。
3. Lamport 签名确认库看到 Bob 想要基于 SHA256 的 Lamport 签名，于是给 SHA256 库多次发调用来确认签名。
4. Lamport 签名确认库一旦回到 1，表明签名已确认，他就会给代表赌注的合约发送信息。
5. 赌注合约检查提供旧金山天气的合约，查看天气如何。
6. 赌注合约看到对信息的回应显示天气高于 35°C，就会给 Gav 币合约发送信息，将 Gav 币从它的账户转移到 Bob 的前向合约。

注意 Gav 币在 Gav 币合约的数据库中作为一个整体“储存”；第 6 步语境中“账户”的意思只是说在 Gav 币合约储存中有数据入口，有钥匙可以进入赌注合约的地址和余额值。接收到信息后，Gav 币合约值上减少，与 Bob 前向账户对应的入口值增加。
我们可以在下表看到这些步骤：



第七章 深入浅出智能合约

7.1 什么是合约？

合约是代码（它的功能）和数据（它的状态）的集合，存在于以太坊区块链的特定地址。 合约账户能够在彼此之间传递信息，进行图灵完备的运算。合约依靠被称作以太坊虚拟机(EVM) 字节代码（以太坊特有的二进制格式）上的区块链运行。
合约很典型地用诸如 Solidity 等高级语言写成，然后编译成字节代码上传到区块链上。

也存在其他语言， 尤其是 Serpent 和 LLL，在此文本的以太坊高级语言章节会进一步阐述。去中心化应用开发资源列出了综合的开发环境，帮助你用这些语言开发的开发者工具，提供测试，和部署支持等功能。

7.2 以太坊高级语言

合约依靠被称作以太坊虚拟机(EVM) 字节代码（以太坊特有的二进制格式）上的区块链运行。然而，合约很典型地用诸如 Solidity 等高级语言写成，然后用以太坊虚拟机编译器编译成字节代码上传到区块链。
下面是开发者可以用来为以太坊写智能合约的高级语言。

Solidity

Solidity 是和 JavaScript 相似的语言，你可以用它来开发合约并编译成以太坊虚拟机字节代码。
它目前是以太坊最受欢迎的语言。

- Solidity 文本 - Solidity 是以太坊的旗舰高级语言，用于写合约。
- Solidity 在线实时编译器
- 标准合约 API
- 有用的去中心化模式 - 用于去中心化应用开发的代码片段。

Serpent

Serpent 是和 Python 类似的语言，可以用于开发合约编译成以太坊虚拟机字节代码。它力求简洁， 将低级语言在效率方面的优点和编程风格的操作简易相结合，同时合约编程增加了独特的领域特定功能。Serpent 用 LLL 编译。

- 以太坊维基百科上的 Serpent
- Serpent 以太坊虚拟机编译器

LLL

Lisp Like Language (LLL)是和 Assembly 类似的低级语言。它追求极简；本质上只是直接对以太坊虚拟机的一点包装。

- GitHub 上的 LIBLLL
- LLL 实例

Mutan (弃用)

Mutan 是个静态类型，由 Jeffrey Wilcke 开发设计的 C 类语言。它已经不再受到维护。

7.3 写合约

没有 Hello World 程序，语言就不完整。Solidity 在以太坊环境内操作，没有明显的“输出”字符串的方式。我们能做的最接近的事就是用日志记录事件来把字符串放进区块链：

```
contract HelloWorld {
event Print(string out);
function() { Print("Hello, World!"); }
}
```

每次执行时，这个合约都会在区块链创建一个日志入口，印着“Hello,World!”参数。

另请参阅：

Solidity docs 里有更多写 Solidity 代码的示例和指导。

7.4 编译合约

solidity 合约的编译可以通过很多机制完成。

- 通过命令行使用 solc 编译器。
- 在 geth 或 eth 提供的 javascript 控制台使用 web3.eth.compile.solidity (这仍然需要安装 solc 编译器)。
- 在线 Solidity 实时编译器。
- 建立 solidity 合约的 Meteor dapp Cosmo。
- Mix IDE。
- 以太坊钱包。

注意：关于 solc 和编译 Solidity 合约代码的更多信息可在此查看。

在 geth 设置 solidity 编译器

如果你启动了 geth 节点，就可以查看哪个编译器可用。

```
> web3.eth.getCompilers();
["lll", "solidity", "serpent"]
```

这一指令会返回到显示当前哪个编译器可用的字符串。

注意：solc 编译器和 cpp- ethereum 一起安装。或者，你可以自己创建。

如果你的 solc 可执行文件不在标准位置，可以用—solc 标志为 solc 可执行文件指定一个定制路线。

```
$ geth --solc /usr/local/bin/solc
```

或者你可以通过控制台在执行期间设置这个选项：

```
> admin.setSolc("/usr/local/bin/solc")
solc, the solidity compiler commandline interface
Version: 0.2.2-02bb315d/.-Darwin/appleclang/JIT linked to
libethereum-1.2.0-8007cef0/.-Darwin/appleclang/JIT
path: /usr/local/bin/solc
```

编译一个简单合约

让我们编译一个简单的合约源：

```
> source = "contract test { function multiply(uint a)
returns(uint d) { return a * 7; } }"
```

这个合约提供了一个单一方法 **multiply**，它和一个正整数 a 调用并返回到 a*7。你准备在 geth JS 控制台用 eth.compile.solidity()编译 solidity 代码：

```
> contract = eth.compile.solidity(source).test
{
code:
'605280600c6000396000f3006000357c010000000000000000000000000000000000
0000000000000000000000090048063c6888fa114602e57005b6037604356041565b8
060005260206000f35b60006007820290506004d565b91905056',
info: {
language: 'Solidity',
languageVersion: '0',
compilerVersion: '0.9.13',
abiDefinition: [{
constant: false,
inputs: [{
name: 'a',
type: 'uint256'
} ],
name: 'multiply',
outputs: [{
name: 'd',
type: 'uint256'
} ],
type: 'function'
} ],
userDoc: {
methods: {
}
},
developerDoc: {
methods: {
}
},
source: 'contract test { function multiply(uint a) returns(uint d)
{ return a
*
7; } }'
```

```
}
}
```

注意：编译器通过 RPC 因此也能通过 web3.js，对浏览器内任何通过 RPC/IPC 连接到 geth 的Ðapp 可用。

下面的例子会向你展示如何通过 JSON-RPC 接合 geth 来使用编译器。

```
$ geth --datadir ~/eth/ --loglevel 6 --logtostderr=true --rpc --rpcport
8100 --rpccorsdomain ' * ' --mine console 2>> ~/eth/eth.log
$ curl -X POST --data
'{"jsonrpc":"2.0","method":"eth_compileSolidity","params":["contract
test {
```

单源编译器输出会给出你合约对象，每个都代表一个单独的合约。eth.compile.solidity 的实际返还值是合约名字到合约对象的映射。由于合约名字是 test，eth.compile.solidity(source).test 会给出包含下列领域的测试合约对：

Code 编译的以太坊虚拟机字节代码
Info 从编译器输出的额外元数据
Source 源代码
Language 合约语言 (Solidity, Serpent, LLL)
LanguageVersion 合约语言版本
compilerVersion 用于编译这个合约的 solidity 编译器版本。
abiDefinition 应用的二进制界面定义
userDoc 用户的 NatSpec Doc。
developerDoc 开发者的 NatSpec Doc。

编译器输出的直接结构化(到 code 和 info)反映了两种非常不同的**部署**路径。编译的以太坊虚拟机代码和一个合约创建交易被发送到区块，剩下的(info)在理想状态下会存活在去中心化云上，公开验证的元数据则执行区块链上的代码。

如果你的源包含多个合约，输出会包括每个合约一个入口，对应的合约信息对象可以用作为属性名称的合约名字检索到。你可以通过检测当前的 GlobalRegistrar 代码来试一下：

```
contracts = eth.compile.solidity(globalRegistrarSrc)
```

7.5 创建和部署合约

开始这一章节之前，确保你有解锁的账户和一些资金。
你现在会在区块链上创建一个合约，方法是用上一章节的以太坊虚拟机代码作为数据给空地址发送交易。

注意：用在线 Solidity 实时编译器或 Mix IDE 程序会更容易完成。

```
var primaryAddress = eth.accounts[0]
var abi = [{ constant: false, inputs: [{ name: 'a', type:
'uint256' } ]
var MyContract = eth.contract(abi)
var contract = MyContract.new(arg1, arg2, ..., {from:
primaryAddress, data: evmByteCodeFromPrevio
```

所有的二进制数据都以十六进制的格式序列化。十六进制字符串总会有一个十六进制前缀 0x。

注意：注意 arg1, arg2, ...是合约构造函数参数，以备它要接受参数。如果合约不需要构造函数参数，就可以忽略这些参数。

值得指出的是，这一步骤需要你支付执行。一旦交易成功进入到区块，你的账户余额(你作为发送方放在 from 领域)会根据以太坊虚拟机的 gas 规则被扣减。一段时间以后，你的交易会在一个区块中出现，确认它带来的状态是共识。你的合约现在存在于区块链上。
以不同步的方式做同样的事看起来是这样：

```
MyContract.new([arg1, arg2, ...,]{from: primaryAccount, data:
evmCode}, function(err, contract) {
if (!err && contract.address)
console.log(contract.address);
});
```

7.6 与合约互动

与合约互动典型的做法是用诸如 eth.contract()功能的抽象层，它会返回到 javascript 对象，和所有可用的合约功能一起，作为可调用的 javascript 功能。

描述合约可用功能的标准方式是 ABI 定义。这个对象是一个字符串，它描述了调用签名和每个可用合约功能的返回值。

```
var Multiply7 = eth.contract(contract.info.abiDefinition);
var myMultiply7 = Multiply7.at(address);
```

现在 ABI 中具体说明的所有功能调用都在合约实例中可用。你可以用两种方法中的一种来调用这些合约实例上的方法。

```
> myMultiply7.multiply.sendTransaction(3, {from: address})
"0x12345"
> myMultiply7.multiply.call(3)
21
```

当用 sendTransaction 被调用的时候，功能调用通过发送交易来执行。需要花费以太币来发送，调用会永久记录在区块链上。用这种方式进行的调用返回值是交易散表。
当用 call 被调用的时候，功能在以太坊虚拟机被本地执行，功能返回值和功能一起返回。用这种方式进行的调用不会记录在区块链上，因此也不会改变合约内部状态。这种调用方式被称为**恒定**功能调用。以这种方式进行的调用不花费以太币。
如果你只对返回值感兴趣，那么你应该用 call 。如果你只关心合约状态的副作用，就应该用 sendTransaction。
在上面的例子中，不会产生副作用，因此 sendTransaction 只会烧 gas，增加宇宙的熵。

7.7 合约元数据

在之前的章节，我们揭示了怎样在区块链上创建合约。现在我们来处理剩下的编译器输出，合约元数据或者说合约信息。
当与不是你创建的合约互动时，你可能会想要文档或者查看源代码。合约作者被鼓励提供这样的可见信息，他们可以在区块链上登记或者借助第三方服务，比如说 EtherChain。管理员 API 为所有选择登记的合约提供便利的方法来获取这个捆绑。

```
// get the contract info for contract address to do manual verification
var info = admin.getContractInfo(address) // lookup, fetch, decode
var source = info.source;
var abiDef = info.abiDefinition
```

这项工作的潜在机制是：

- 合约信息被可以公开访问的 URI 上传到可辨认的地方
- 任何人都可以只知道合约地址就找到是什么 URI

仅通过 2 个步骤的区块链注册就可以实现这些要求。第一步是在被称作 HashReg 的合约中用内容散表注册合约代码（散表）。第二步是在 UrlHint 合约用内容散表注册一个 url。这些注册合约是 Frontier 版本的一部分，已经参与到 Homestead 中。

要知道合约地址来查询 url，获取实际合约元数据信息包，使用这一机制就足够了。

如果你是个尽职的合约创建者，请遵循以下步骤：

1. 将合约本身部署到区块链
2. 获取合约信息 json 文件
3. 将合约信息 json 文件部署到你选择的任意 url
4. 注册代码散表 ->内容散表 -> url

JS API 通过提供助手把这个过程变得非常容易。 调用 admin.register 从合约中提取信息，在指定文件中写出 json 序列，运算文件的内容散表，最终将这个内容散表注册到合约代码散表。一旦将那个文件部署到任意 url，你就能用 admin.registerUrl 来注册 url 和你区块链上的内容散表(注意一旦固定的内容选址模式被用作文件商店，url-hint 不再必要了。)

```
source = "contract test { function multiply(uint a) returns(uint d)
{ return a
*
7; } }"
// compile with solc
contract = eth.compile.solidity(source).test
// create contract object
var MyContract = eth.contract(contract.info.abiDefinition)
// extracts info from contract, save the json serialisation in the given
file,
contenthash = admin.saveInfo(contract.info,
"~/dapps/shared/contracts/test/info.json")// send off the contract to
the blockchain
MyContract.new({from: primaryAccount, data: contract.code},
function(error, contract){
if(!error && contract.address) {
// calculates the content hash and registers it with the code hash in
`HashReg`
// it uses address to send the transaction.
// returns the content hash that we use to register a url
admin.register(primaryAccount, contract.address, contenthash)
// here you deploy ~/dapps/shared/contracts/test/info.json to a url
admin.registerUrl(primaryAccount, hash, url)
}
});
```

7.8 测试合约和交易

你通常需要低级的测试策略，为交易和合约排除故障。这一章节介绍了一些你可以用到的排错工作和做法。为了测试合约和交易而不产生实际的后果，你最好在私有区块链上测试。这可以通过配置一个替代网络 ID (选择一个特别的数字)和/或不能用的端点来实现。推荐做法是，为了测试你用一个替代数据目录和端口 ，这样就不会意外地和实时运行的节点冲突(假定用默认运行。在虚拟机排错模式开启 geth，推荐性能分析和最高的日志冗余级别 ：

```
geth --datadir ~/dapps/testing/00/ --port 30310 --rpcport 8110
--networkid 4567890 --nodiscover -
```

提交交易之前，你需要创建私有测试链。参阅*测试网络*。

```
// create account. will prompt for password
personal.newAccount();
// name your primary account, will often use it
primary = eth.accounts[0];
// check your balance (denominated in ether)
balance = web3.fromWei(eth.getBalance(primary), "ether");
```

```
// assume an existing unlocked primary account
primary = eth.accounts[0];
// mine 10 blocks to generate ether
// starting miner
miner.start(4);
// sleep for 10 blocks (this can take quite some time).
admin.sleepBlocks(10);
// then stop mining (just not to burn heat in vain)
miner.stop();
balance = web3.fromWei(eth.getBalance(primary), "ether");
```

创建交易之后，你可以用下面的命令来强制运行：

```
miner.start(1);
admin.sleepBlocks(1);
miner.stop();
```

你可以用以下命令查看即将发生的交易：

```
// shows transaction pool
txpool.status
// number of pending txs
eth.getBlockTransactionCount("pending");
// print all pending txs
eth.getBlock("pending", true).transactions
```

如果你提交合约创建交易，可以检查想要的代码是否实际上嵌入到当前的区块链：

```
txhash = eth.sendTansaction({from:primary, data: code})
//... mining
contractaddress = eth.getTransactionReceipt(txhash);
eth.getCode(contractaddress)
```

第八章 如何部署、调用智能合约

8.1 RPC

之前的章节中我们看到了怎么写、部署合约以及与合约互动。现在该讲讲与以太坊网络和智能合约沟通的细节了。

一个以太坊节点提供一个 RPC 界面。这个界面给Ðapp 访问以太坊区块链的权限和节点提供的功能，比如编译智能合约代码，它用 JSON-RPC 2.0 规范(不支持提醒和命名的参数) 的子集作为序列化协议，在 HTTP 和 IPC (linux/OSX 上的 unix 域接口，在 Windows 上叫 pipe’s)上可用。

如果你对细节不感兴趣，正在寻找使用 javascript 库的简便方法，你可以略过下面的章节，从 *Using Web3* 继续。

8.2 惯例

RPC 界面使用一些惯例，它们不是 JSON-RPC 2.0 规范的一部分：

- 数字是十六进制编码。做这个决定是因为有些语言对运行极大的数字没有或有很少的限制。为了防止这些错误数字类型是十六进制编码，由开发者来分析这些数字并正确处理它们。在维基页百科查看十六进制编码章节查看案例。
- 默认区块数字，几个 RPC 方法接受区块数字。在一些情况下，给出区块数字是不可能的或者不太方便。 在那样的情况下，默认区块数字可以是以下字符串

中的一个[”earliest”，“latest”，“pending”]。在维基页面查看使用默认区块参数的 RPC 方法列表。

8.3 部署合约

我们会通过不同的步骤来部署下面的合约，只用到 RPC 界面。

```
contract Multiply7 {
event Print(uint);
function multiply(uint input) returns (uint) {
Print(input
*
7);
return input
*
7;
}
}
```

要做的第一件事是确保 HTTP RPC 界面可用。这意味着我们在开始为 geth 供应—rpc 标志，为 eth 提供-j 标志。在这个例子中我们用私有开发链上的 geth 节点。通过这种方法，我们就不需要真实网络上的以太币了。

```
> geth --rpc --dev --mine --minerthreads 1 --unlock 0 console 2>>geth.log
```

这会在 http://localhost:8545 上启动 HTTP RPC 界面。

注意：geth 支持 CORS 查看—rpccorsdomain 标志了解更多。

我们可以通过用 curl 检索 coinbase 地址和余额来证明界面正在运行。请注意这些例子中的数据在你本地的节点上会有所不同。如果你想要试试这些参数，视情况替换需要的参数。

```
> curl --data '{"jsonrpc":"2.0","method":"eth_coinbase", "id":1}'
localhost:8545
{"id":1,"jsonrpc":"2.0","result":["0xeb85a5557e5bdc18ee1934a89d8bb402
398ee26a"]}
> curl --data '{"jsonrpc":"2.0","method":"eth_getBalance", "params":
["0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a"], "id":2}'
localhost:8545
{"id":2,"jsonrpc":"2.0","result":"0x1639e49bba16280000"}
```

记不记得我们说过数字是十六进制编码？在这个情况下，余额作为十六进制字符串以 Wei 的形式返还。如果我们希望余额作为数字以太币为单位，我们可以从控制台用 web3。

```
> web3.fromWei("0x1639e49bba16280000", "ether")
"410"
```

现在我们在私有开发链上有一些以太币，我们就可以部署合约了。第一步是验证 solidity 编译器可用。我们可以用 eth_getCompilers RPC method 方法来检索可用的编译器。

```
> curl --data '{"jsonrpc":"2.0","method": "eth_getCompilers", "id": 3}'
localhost:8545
{"id":3,"jsonrpc":"2.0","result":["Solidity"]}
```

我们可以看到 solidity 编译器可用。如果不可用，按照这些说明操作。
下一步是把 Multiply7 合约编译到可以发送给以太坊虚拟机的字节代码。

```
> curl --data '{"jsonrpc":"2.0","method": "eth_compileSolidity",
"params": ["contract Multiply7 { event Print(uint); function
multiply(uint input) returns (uint) { Print(input
{"id":4,"jsonrpc":"2.0","result":{"Multiply7":{"code":"0x606060405260
5f80601060003960000f360606040
```

现在我们有了编译代码，需要决定花多少 gas 去部署它。RPC 界面有 eth_estimateGas 方法，会给我们一个预估数量。

```
> curl --data '{"jsonrpc":"2.0","method": "eth_estimateGas", "params":
[{"from": "0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a", "data":
"0x6060604052605f80601060003960000f3606060405260e060020a6000350463c688
8fa18114601a575b005b60586004356007810260609081526000907f24abdb5865df5
079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da90602090a150600702
90565b5060206060f3"}], "id": 5}' localhost:8545
{"id":5,"jsonrpc":"2.0","result":"0xb8a9"}
```

最后部署合约。

```
> curl --data '{"jsonrpc":"2.0","method": "eth_sendTransaction",
"params": [{"from": "0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a",
"gas": "0xb8a9", "data":
"0x6060604052605f80601060003960000f3606060405260e060020a6000350463c688
8fa18114601a575b005b60586004356007810260609081526000907f24abdb5865df5
079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da90602090a150600702
90565b5060206060f3"}], "id": 6}' localhost:8545
```

```
{"id":6,"jsonrpc":"2.0","result":"0x3a90b5face52c4c5f30d507ccf51b0209
ca628c6824d0532bcd6283df7c08
```

交易由节点接受，交易散表被返还。我们可以用这个散表来跟踪交易。
下一步是决定部署合约的地址。每个执行的交易都会创建一个接收。这个接收包含交易的各种信息，比如交易被包含在哪个区块，以太坊虚拟机用掉多少 gas。如果交易创建了一个合约，它也会包含合约地址。我们可以用 eth_getTransactionReceipt RPC 方法检索接收。

```
> curl --data '{"jsonrpc":"2.0","method": "eth_getTransactionReceipt",
"params":
["0x3a90b5face52c4c5f30d507ccf51b0209ca628c6824d0532bcd6283df7c08a7c"
], "id": 7}' localhost:8545
{"id":7,"jsonrpc":"2.0","result":{"transactionHash":"0x3a90b5face52c4
c5f30d507ccf51b0209ca628c682
```

我们可以看到合约在 0x6ff93b4b46b41c0c3c9baee01c255d3b4675963d 上被创建。如果你得到了零而不是接收，说明还没有被纳入区块。等一下，检查看看你的矿工是否在运行，重新试一遍。

8.4 和智能合约互动

现在已经部署了合约，我们可以和它互动了。有两种方法，发送交易或*像之前所介绍的那样使用调用*。在这个例子中，我们会发送交易到合约的 multiply 方法。
如果我们看 eth_sendTransaction 的档案，可以发现我们需要提供几个参数。在我们的实例中，需要具体说明 from, to 和 data 参数。From 是我们账户的公共地址，to 是合约地址。Data 参数有一点困难。它包括了规定调用哪个方法和哪个参数的负载量。这就需要 ABI 发挥作用了。ABI 规定了如何为以太坊虚拟机规定和编码数据。你可以在这儿阅读 ABI 的所有具体信息。
负载量的字节是功能选择符，规定了调用哪个方法。它取 Keccak 散表的头 4 个字节，涵盖功能名称参数类型，并进行十六进制编码。multiply 功能接受一个*单元*，也就是 *uint256* 的别名。这就让我们进行：

```
> web3.sha3("multiply(uint256)").substring(0, 8)
"c6888fa1"
```

在此页查看细节。
下一步是编码参数。我们只有一个 unit256，让我们假定提供了值 6。ABI 有一个章节规定了怎么编码 uint 字节。

int<M>: enc(X) is the big-endian two's complement encoding of X, padded on the higher-oder (left) side with 0xff for negative X and with zero 字节s for positive X such that the length is a multiple of 32 bytes.
它编码到
0000000000000000000000000000000000000000000000000000000000000006.
将功能选择符和编码参数结合起来，数据就会变成0xc6888fa1000000000000000000000000000000000000000000000000000000000000006.
我们来试一下：

```
> curl --data '{"jsonrpc":"2.0","method": "eth_sendTransaction",
"params": [{"from": "0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a",
"to": "0x6ff93b4b46b41c0c3c9baee01c255d3b4675963d", "data":
"0xc6888fa1000000000000000000000000000000000000000000000000000000000000006"}], "id": 8}' localhost:8545
{"id":8,"jsonrpc":"2.0","result":"0x759cf065cbc22e9d779748dc53763854e5376eea07409e590c990eafc0869
```

由于我们发送了交易，于是有交易散表返回。如果我们检索接收，可以看到一些新内容：

```
{
blockHash:
"0xbf0a347307b8c63dd8c1d3d7cbdc0b463e6e7c9bf0a35be40393588242f01d55",
blockNumber: 268,
contractAddress: null,
cumulativeGasUsed: 22631,
gasUsed: 22631,
logs: [{
address: "0x6ff93b4b46b41c0c3c9baee01c255d3b4675963d",
blockHash:
"0xbf0a347307b8c63dd8c1d3d7cbdc0b463e6e7c9bf0a35be40393588242f01d55",
blockNumber: 268,
data:
"0x000000000000000000000000000000000000000000000000000000000000002a",
logIndex: 0,
topics:
["0x24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da"
],
transactionHash:
"0x759cf065cbc22e9d779748dc53763854e5376eea07409e590c990eafc0869d74",
transactionIndex: 0
}],
transactionHash:
```

```
"0x759cf065cbc22e9d779748dc53763854e5376eea07409e590c990eafc0869d74",
transactionIndex: 0
}
```

接收包含一个日志。日志由以太坊虚拟机在交易执行时生成，包含接收。如果我们看 Multiply 功能，可以看到打印事件和输入次数 7 一起被提出。由于打印事件的参数是 uint256，我们可以根据 ABI 规则对它进行编码，这样我们就会得到预期的十进制 42.。除数据外，主题用于决定什么事件来创建日志，是毫无意义的：

```
> web3.sha3("Print(uint256)")
"24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da"
```

你可以在 Solidity 教程中阅读更多关于事件，主题和索引的内容。
这只是对一些最常见任务的简单介绍。在 RPC 维基页面查看可用 RPC 方法的完整列表。

8.5 Web3.js

正如我们在之前的案例所见，使用 JSON-RPC 界面相当单调乏味且容易出错，尤其是在处理 ABI 的时候。Web3.js 是 javascript 库，在以太坊 RPC 界面顶端。它的目标是提供更友好的界面，减少出错机会。
用 web3 部署 Multiply7 合约看起来是这样：

```
var source = 'contract Multiply7 { event Print(uint); function
multiply(uint input) returns (uint) { Print(input
var compiled = web3.eth.compile.solidity(source);
var code = compiled.Multiply7.code;
var abi = compiled.Multiply7.info.abiDefinition;
web3.eth.contract(abi).new({from:
"0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a", data: code}, function
(err, contract) {
if (!err && contract.address)
console.log("deployed on:", contract.address);
}
);
deployed on: 0x0ab60714033847ad7f0677cc7514db48313976e2
```

装载一个部署的合约，发送交易：

```
var source = 'contract Multiply7 { event Print(uint); function
```

```
multiply(uint input) returns (uint) { Print(input
var compiled = web3.eth.compile.solidity(source);
var Multiply7 =
web3.eth.contract(compiled.Multiply7.info.abiDefinition);
var multi = Multiply7.at("0x0ab60714033847ad7f0677cc7514db48313976e2")
multi.multiply.sendTransaction(6, {from:
"0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a"})
```

注册一个回调，打印事件创建日志的时候会被调用。

```
multi.Print(function(err, data) { console.log(JSON.stringify(data)) })
{"address":"0x0ab60714033847ad7f0677cc7514db48313976e2","args":
{"":"21"},"blockHash":"0x259c7dc0
```

在 web3.js 维基页面查看更多信息。

8.6 控制台

geth 控制台提供命令行界面和 javascript 执行时间。它可以连接到本地或远程的 geth 或 eth 节点。它会装载用户能使用的 web3.js 库。这会允许用户从控制台用 web3.js 部署智能合约并和智能合约互动。实际上 Web3.js 章节的例子可以被复制进控制台。

8.7 查看合约与交易

有几个可用的在线区块链浏览器，能让你查询以太坊区块链。
查看列表：*区块链浏览器*。

在线区块链浏览器

- EtherChain
- EtherCamp
- EtherScan （为测试网）

其他资源

- EtherNodes - 节点的地理分配，由客户端区分
- EtherListen - 实时以太坊交易可视器和可听器

第九章 智能合约案例实战

以太坊是区块链开发领域最好的编程平台，而 truffle 是以太坊（Ethereum）最受欢迎的一个开发框架，这是我们介绍 truffle 的原因，实战是最重要的事情，这篇文章不讲原理，只搭建环境，运行第一个区块链程序（Dapp）。

9.1 安装 truffle

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

9.2 依赖环境

NodeJS
访问 https://nodejs.org 官方网站下载安装

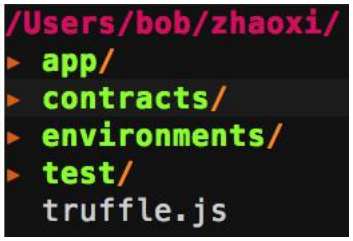
系统：Windows, Linux or Mac OS X，推荐 Mac OS X，不建议使用 Windows，会碰到各种各样的问题，导致放弃。

需要安装 Ethereum 客户端，来支持 JSON RPC API 调用
开发环境，推荐使用 EthereumJS
TestRPC: https://github.com/ethereumjs/testrpc

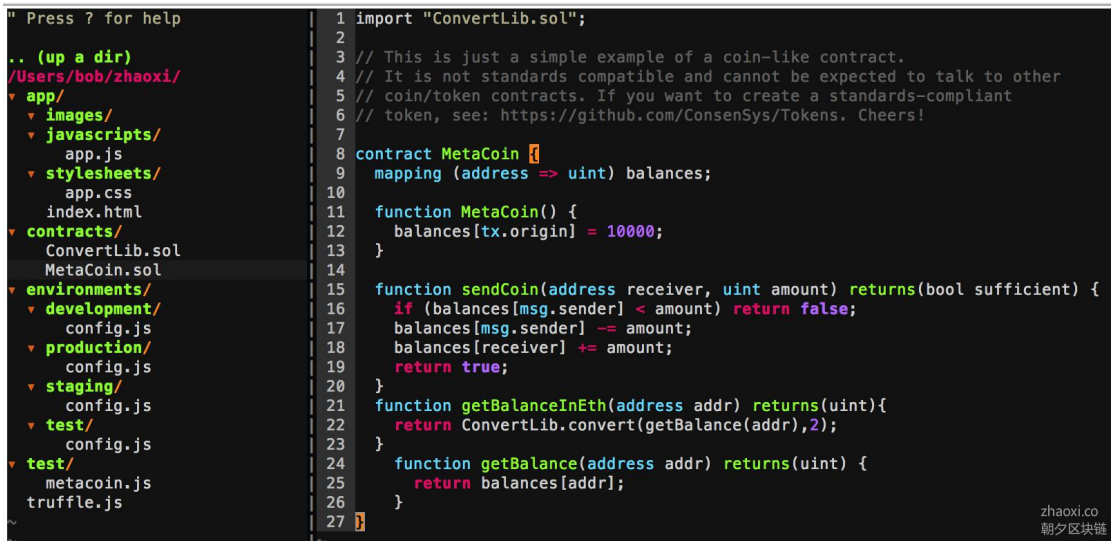
安装命令：
```
$ npm install -g ethereumjs-testrpc
```

9.3 新建第一个项目

```
$ mkdir zhaoxi
$ cd zhaoxi
$ truffle init
```
默认会生成一个 MetaCoin 的 demo，可以从这个 demo 中学习 truffle 的架构
项目目录结构如图：



项目所有文件目录如图：



编译项目

$ truffle compile

部署项目

部署之前先启动 TestRPC

$ testrpc

$ truffle deploy（在 Truffle 2.0 以上版本中，命令变成了：truffle migrate）

$ truffle migrate 执行结果

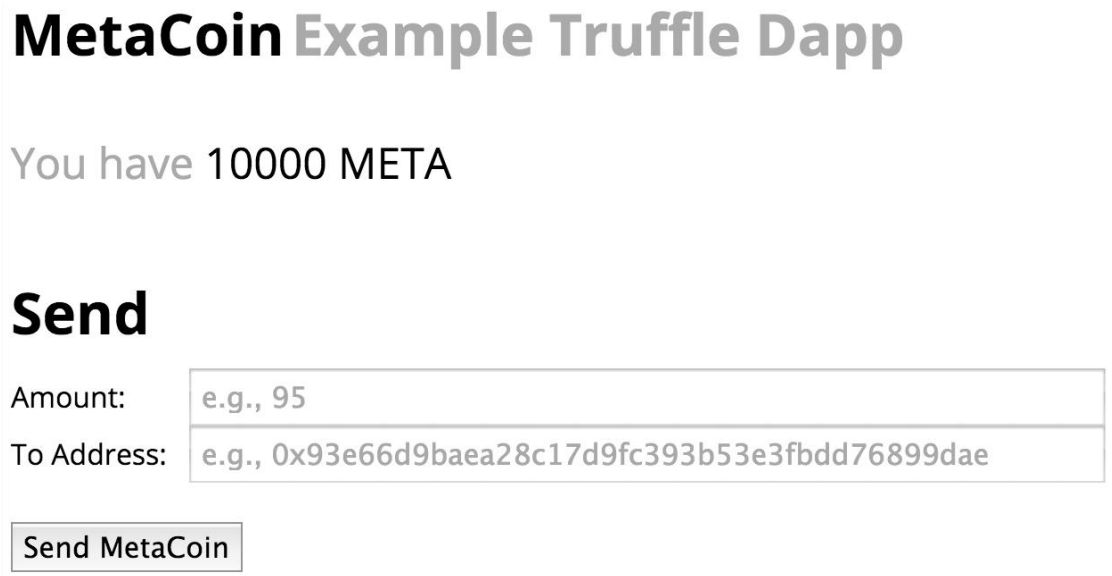
```
[bob@192 zhaoxi % truffle migrate
Running migration: 1_initial_migration.js
  Deploying Migrations...
  Migrations: 0x34e94dd89ed596077d5a4f9d8481dbd0633a454c
Saving successful migration to network...
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying ConvertLib...
  ConvertLib: 0x62038717aeb35113425683d8eb3dfab65239162c
  Linking ConvertLib to MetaCoin
  Deploying MetaCoin...
  MetaCoin: 0xbeb2aaac8e7c85426dcb120667d5fe49358581b0
Saving successful migration to network...
Saving artifacts...
```

启动服务

$ truffle serve

```
bob@192 zhaoxi % truffle serve
Using environment development.
Serving app on port 8080...
Rebuilding...
Completed without errors on Sun May 01 2016 05:53:12 GMT+0800 (CST)
```

启动服务后，可以在浏览器访问项目：
http://localhost:8080/ ，网页界面如下：



好了，第一个区块链程序跑起来了，后面可以不断地实践深入学习了。

10 总结

10.1 写这本书的初衷

因为接触区块链比较早，并写了一些文章，所以很多朋友加我微信、QQ 聊，其中包括开发者、大小公司的创始人、高管。问了很多问题，因为时间关系，我也无法一一作答。另外，多家出版社希望我写一本区块链书籍，我知道出纸质书流程很长，对文字也要字斟句酌，对于创业者来说，确实没有那么多时间去独自写一本书，因此婉言谢绝了。

基于以上原因，综合考虑下来决定写一本以太坊电子书，很多人有这个需求，所以会很有价值。因为区块链技术发展日新月异，因此《深入浅出以太坊》是一本不断更新的电子书，为了时刻保持对最新技术的跟进。后面还会出视频教程，更好的帮助大家学习。以后会更新到如下网址（目前尚未上线）：

书籍：book.wangxiaoming.com
视频：ethcast.com