# 1、区块链技术原理及技术介绍

## 1.1 区块链介绍

### 1.1.1区块链简介与发展史

#### 1.1.1.1区块链产生的背景

区块链技术（Block Chain）是指通过去中心化的方式集体维护一个可靠数据库的技术方案。该技术方案主要让区块（Block）通过密码学方法相关联起来，每个数据块包含了一定时间内的系统全部数据信息，并且生成数字签名以验证信息的有效性并链接到下一个数据块形成一条主链（Chain）。

区块（Block）是区块链中的一条记录，包含并确认待处理的交易。

随着技术演进，摩尔定律导致了整个分布式计算和分布式存储成本正在逐渐降低。在某些领域，分布式架构的效率和成本比中心化更有优势，在这样的历史背景之下诞生了区块链。即便没有区块链一定也会有其他的技术，这是商业演化和技术演进的在时空交汇出的结果。

在这样的背景下，2008年9月，中本聪的“一个点对点的电子货币交易的机制”在网络出现，这篇文章创造了一个现在1500亿美元市值的交易网络。区块链是[比特币](http://www.lianmenhu.com/baike/)的底层技术，就是由中本聪的这篇论文而来。而[区块链技术](http://www.lianmenhu.com/apply/)真正被业界所发现是在2013年，一经发现整个世界都在风靡，尤其在[金融](http://jinrong.lianmenhu.com/)领域。

#### 1.1.1.2比特币的诞生

20世纪初期，奥地利经济学派第一次完整的阐述了一种去国家化的非主权货币的构想。代表作品，哈耶克所著的《货币的非国家化》，这可以认为是比特币最早的理论起源。只可惜，当时木有强大的计算机，也木有互联网，如何从技术上完美实现，是个悬而未决的难题。  
  随着时间流逝，计算机出现了，然后，更有爱的互联网忽然席卷了人类社会，于是。。。  
一切开始飞速进步。。。  
  
1、上个世纪80年代，一群天才程序员在密码邮件组探讨这一难题。1982年，大卫·乔姆(David Chaum)提出不可追踪的密码学网络支付系统。8年后，他将此想法扩展为密码学匿名现金系统，即Ecash。  
2、 1998年，戴伟(Wei Dai)的论文阐述了一种匿名的、分布式的电子现金系统：b-money。与此同时，尼克·萨博(Nick Szabo)发明了Bitgold，提出工作量证明机制，用户通过竞争性地解决数学难题，然后将解答的结果用加密算法串联在一起公开发布，构建出一个产权认证系统。哈尔·芬尼(Hal Finney)则把该机制完善为一种“可重复利用的工作量证明”。  
3、 在前人的工作基础之上，2008年，名不见经传的“中本聪”在metzdowd.com的密码学邮件组列表中发表了《比特币：一种点对点的现金支付系统》。2009年1月3日，比特币网络诞生，中本聪本人发布了开源的第一版比特币客户端。  
  从此，人类的货币史，翻开了新的一页。

#### 1.1.1.3比特币与区块链

说到区块链，很多人会想到比特币，因为大多数人是从比特币知道区块链的，区块链和比特币又有什么区别呢？

“区块链技术”——分布式账本

用通俗的话阐述：如果我们把数据库假设成一本账本，读写数据库就可以看做一种记账的行为，区块链技术的原理就是在一段时间内找出记账最快最好的人，由这个人来记账，然后将账本的这一页信息发给整个系统里的其他所有人。这也就相当于改变数据库所有的记录，发给全网的其他每个节点，所以区块链技术也称为分布式账本

“比特币 ”——比特金（虚拟货币）

比特币（BitCoin）的概念最初由中本聪在2009年提出，根据中本聪的思路设计发布的开源软件以及建构其上的P2P网络。比特币是一种P2P形式的数字货币。点对点的传输意味着一个去中心化的支付系统。

本质区别：比特币对于这个世界来说，它是一个基于密码学的数字货币，而区块链我们刚才说过，它是一种价值传递的协议，这两者是有本质区别的，因为一个是数字货币，一个是价值传递的协议。

1.1.2区块链的主要概念

在中本聪的论文《BitCoin：A Peer-to-Peer Electronic Cash System》中定义区块链技术是构建比特币系统的基础，区块链记录着所有元数据和加密交易信息，从而建立了一个完全通过P2P技术实现的电子现金系统，此系统使得在线支付的双方不用通过第三方金融机构而直接进行交易。比特币系统是第一个采用区块链技术作为底层技术构建的系统，他是一个实现了去中心化、去信任化、安全可靠的电子现金系统。

其主要有以下六个特征：

　　去中心化

　　由于使用分布式核算和存储，不存在中心化的硬件或管理机构，任意节点的权利和义务都是均等的，系统中的数据块由整个系统中具有维护功能的节点来共同维护。任一节点停止工作都会不影响系统整体的运作。

　　去信任

　　系统中所有节点之间无需信任也可以进行交易，因为数据库和整个系统的运作是公开透明的，在系统的规则和时间范围内，节点之间无法欺骗彼此。

　　开放性

　　系统是开放的，除了交易各方的私有信息被加密外，区块链的数据对所有人公开，任何人都可以通过公开的接口查询区块链数据和开发相关应用，因此整个系统信息高度透明。

　　自治性

　　区块链采用基于协商一致的规范和协议,使得整个系统中的所有节点能够在去信任的环境自由安全的交换数据，使得对“人”的信任改成了对机器的信任，任何人为的干预不起作用。

　　信息不可篡改

　　一旦信息经过验证并添加至区块链，就会永久的存储起来，除非能够同时控制住系统中超过51%的节点，否则单个节点上对数据库的修改是无效的，因此区块链的数据稳定性和可靠性极高。

　　匿名性

　　由于节点之间的交换遵循固定的算法，其数据交互是无需信任的(区块链中的程序规则会自行判断活动是否有效)，因此交易对手无须通过公开身份的方式让对方自己产生信任，对信用的累积非常有帮助。

### 1.1.3 区块链的基本原理

区块链的基本原理理解起来并不难。基本概念包括：

交易（Transaction）：一次操作，导致账本状态的一次改变，如添加一条记录；

区块（Block）：记录一段时间内发生的交易和状态结果，是对当前账本状态的一次共识；

链（Chain）：由一个个区块按照发生顺序串联而成，是整个状态变化的日志记录。

如果把区块链作为一个状态机，则每次交易就是试图改变一次状态，而每次共识生成的区块，就是参与者对于区块中所有交易内容导致状态改变的结果进行确认。

### 1.1.4 区块链的交易流程

1. 某人发出交易请求
2. 广播交易请求到 P2P 网络
3. 验证，miners验证交易正确性
4. 多个交易组成一个区块
5. 新的区块加入到一个已经存在区块链中
6. 交易完成

### 1.1.5 区块链的主要应用场景

1、区块链资产相关应用

    区块链资产发行、支付、跨境汇兑、交易、买卖等。

    6月25日下午消息，今日港版支付宝AlipayHK上线全球首个基于区块链的电子钱包跨境汇款服务。发布会现场，在港工作的菲律宾人Grace完成了第一笔汇款，耗时仅3秒。

2、记账方式相关应用

    比如银行业、证券业、保险业都可以利用区块链的记账方式来弥补或提高现有的记账和清算效率。在金融行业应用中，主要应用了区块链的去中心化，分布式记账的特点。原来数据放在一个中心化的银行服务器上，如果中心化服务器崩溃就会产生风险，而区块链技术则通过机制保证，参与各方都有一本账本，且无法篡改。

3、公开可信相关应用

    例如众筹、公益、互助保险、数据保全等，涉及到价值也涉及到公信度。

4、可控匿名相关应用

    例如比特币交易，比特币从一个地址转移到另一个地址。但这个地址背后的身份持有人是谁，我们并不知道。很多人利用这个特性做身份认证，尽可能的保护我们的隐私信息。比如我们去银行、医院，这些场景只要证明“我就是我”就够了，不需要我们公开所有信息，可控匿名就是你不需要透漏一些信息，同时又能证明你是这些信息的主人。

## 1.2 钱包和交易所

### 1.2.1 钱包介绍

1. 浏览器钱包

Metamask

2. 手机移动端钱包

Imtoken，trust

3. PC端钱包

Ethereum-Wallet

### 1.2.2 钱包的安装和使用

课程中穿插介绍

### 1.2.3 区块浏览器介绍

Mist

### 1.2.4 区块链交易所

## 1.3 共识算法

“共识机制”，是指通过特殊节点的投票，在很短的时间内完成对交易的验证和确认；对一笔交易，如果利益不相干的若干个节点能够达成共识，我们就可以认为全网对此也能够达成共识。再通俗一点来讲，如果中国一名微博大V、美国一名虚拟币玩家、一名非洲留学生和一名欧洲旅行者互不相识，但他们都一致认为你是个好人，那么基本上就可以断定你这人还不坏。

[区块链](https://baike.baidu.com/item/%E5%8C%BA%E5%9D%97%E9%93%BE)作为一种按时间顺序存储数据的数据结构，可支持不同的共识机制。共识机制是区块链技术的重要组件。区块链共识机制的目标是使所有的诚实节点保存一致的区块链视图，同时满足两个性质：

（1）一致性。所有诚实节点保存的区块链的前缀部分完全相同。

（2）有效性。由某诚实节点发布的信息终将被其他所有诚实节点记录在自己的区块链中。

### 1.3.1 PBFT拜占庭共识

Practical Byzantine Fault Tolerance，实用拜占庭容错算法。

PBFT是一种状态机副本复制算法，即服务作为状态机进行建模，状态机在分布式系统的不同节点进行副本复制。每个状态机的副本都保存了服务的状态，同时也实现了服务的操作。将所有的副本组成的集合使用大写字母R表示，使用0到|R|-1的整数表示每一个副本。为了描述方便，假设|R|=3f+1，这里f是有可能失效的副本的最大个数。尽管可以存在多于3f+1个副本，但是额外的副本除了降低性能之外不能提高可靠性。

### 1.3.2 POW工作量证明

(Proof of Work)：工作量证明机制

PoW工作量证明机制即对于工作量的证明，是生成要加入到区块链中的一笔新的交易信息(即新区块)时必须满足的要求。在基于工作量证明机制构建的区块链网络中，节点通过计算随机哈希散列的数值解争夺记账权，求得正确的数值解以生成区块的能力是节点算力的具体表现。

### 1.3.3 POS 股份证明

(Proof of Stake)：股权证明机制

POS采用工作量证明机制发行新币，采用权益证明机制维护网络安全，这是权益证明机制在加密电子货币中的首次应用。与要求证明人执行一定量的计算工作不同，权益证明要求证明人提供一定数量加密货币的所有权即可。

### 1.3.4 DPOS委托股份证明

（Delegated Proof of Stake）：授权股权证明机制

DPoS股份授权证明机制是一种新的保障网络安全的共识机制。它在尝试解决传统的PoW机制和PoS机制问题的同时，还能通过实施科技式的民主抵消中心化所带来的负面效应。

### 1.3.5 共识算法总结

1、POW 工作量证明机制，类似于按劳分配。

2、POS 股权证明机制。 持有股票的人，就有对应的权利，持有的越多，权利越大 。

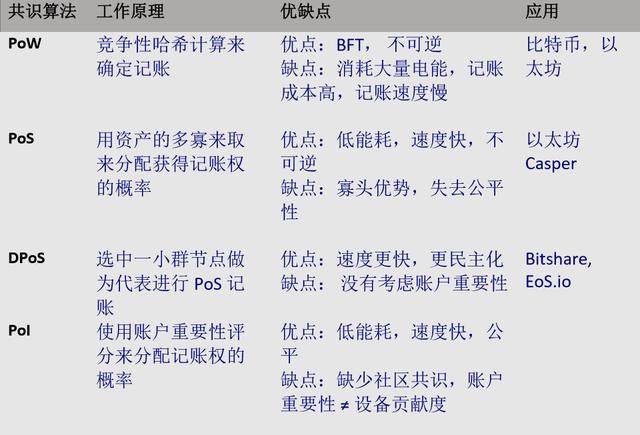
3、DPOS 授权股权证明机制。类似于董事会， 董事会成员数量有限,，由大家选举产生， 被选中的董事会成员可以行使权利。

4、PBFT这是一种基于消息传递的一致性算法，算法经过三个阶段 预准备（pre-prepare）、准备(prepare)和确认(commit)达成一致性，这些阶段可能因为失败而重复进行。

5、POI本质都是POS的变种，就一带而过了。

6、POP是前边所有的几种的升级。

从机制设计上来看. POW 机制更加强调去中心, 更加强调对等.。而DPOS 则是有一个明显的中心, 通过带来部分中心,来得到效率的提升。哪一种机制更好, 有待时间的验证. POW 已经运行快10年, 电力耗费已经非常严重. POP的出现, 有可能让记账这件事情更经济效率, 从而支撑起更多大规模的协作体系。



## 1.4 密码学

### 1.4.1 加密算法介绍

加密算法：[数据加密](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%8A%A0%E5%AF%86/11048982)的基本过程就是对原来为明文的文件或数据按某种[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95/209025)进行处理，使其成为不可读的一段代码，通常称为“密文”，使其只能在输入相应的[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5/101144)之后才能显示出本来内容，通过这样的途径来达到保护数据不被非法人窃取、阅读的目的。 该过程的逆过程为解密，即将该[编码](https://baike.baidu.com/item/%E7%BC%96%E7%A0%81/80092)信息转化为其原来数据的过程。

1、对称加密体系：

对称式加密就是加密和解密使用同一个[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)，通常称之为“Session Key ”这种[加密技术](https://baike.baidu.com/item/%E5%8A%A0%E5%AF%86%E6%8A%80%E6%9C%AF)在当今被广泛采用，如美国政府所采用的DES加密标准就是一种典型的“对称式”[加密法](https://baike.baidu.com/item/%E5%8A%A0%E5%AF%86%E6%B3%95)，它的Session Key长度为56bits

2、非对称加密体系：

非对称式加密就是加密和解密所使用的不是同一个[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)，通常有两个密钥，称为“[公钥](https://baike.baidu.com/item/%E5%85%AC%E9%92%A5)”和“[私钥](https://baike.baidu.com/item/%E7%A7%81%E9%92%A5)”，它们两个必需配对使用，否则不能打开加密文件。这里的“[公钥](https://baike.baidu.com/item/%E5%85%AC%E9%92%A5)”是指可以对外公布的，“[私钥](https://baike.baidu.com/item/%E7%A7%81%E9%92%A5)”则不能，只能由持有人一个人知道。它的优越性就在这里，因为对称式的加密方法如果是在网络上传输加密文件就很难不把[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)告诉对方，不管用什么方法都有可能被别[窃听](https://baike.baidu.com/item/%E7%AA%83%E5%90%AC)到。而非对称式的加密方法有两个[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)，且其中的“[公钥](https://baike.baidu.com/item/%E5%85%AC%E9%92%A5)”是可以公开的，也就不怕别人知道，收件人解密时只要用自己的[私钥](https://baike.baidu.com/item/%E7%A7%81%E9%92%A5)即可以，这样就很好地避免了密钥的传输安全性问题。

**算法:**

一个加密系统S可以用数学符号描述如下：

S={P, C, K, E, D}

其中

P——明文空间，表示全体可能出现的明文集合，

C——密文空间，表示全体可能出现的密文集合，

K——[密钥空间](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5%E7%A9%BA%E9%97%B4)，密钥是加密[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95)中的可变参数，

E——加密[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95)，由一些公式、法则或程序构成，

D——解密[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95)，它是E的逆。

当给定密钥kÎK时，各符号之间有如下关系：

C = Ek(P), 对明文P加密后得到密文C

P = Dk(C) = Dk(Ek(P)), 对密文C解密后得明文P

如用E-1 表示E的逆，D-1表示D的逆，则有：

Ek = Dk-1且Dk = Ek-1

因此，加密设计主要是确定E，D，K。

RSA是Rivest、Shamir和Adleman提出来的基于数论非对称性(公开钥)加密[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95)。 大整数的素因子难分解是[RSA算法](https://baike.baidu.com/item/RSA%E7%AE%97%E6%B3%95)的基础。

RSA在国外早已进入实用阶段，已研制出多种高速的RSA的专用芯片。尽管RSA 的许多特性并不十分理想，但迫于[信息安全](https://baike.baidu.com/item/%E4%BF%A1%E6%81%AF%E5%AE%89%E5%85%A8)的实际需要，许多重要的信息系统还是采 用RSA作为基础加密机制。从RSA提出不久，我国有关部门就一直对它进行研究。从 应用的角度看，软件实现的RSA已经开始用于[计算机网络](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C)加密，用来完成[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)分配、 [数字签名](https://baike.baidu.com/item/%E6%95%B0%E5%AD%97%E7%AD%BE%E5%90%8D)等功能。

除了RSA之外，还有DES([数据加密标准](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%8A%A0%E5%AF%86%E6%A0%87%E5%87%86))。尽管DES公开了其加密[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95)并曾被美国列为“标准”，但很快被废弃。[加密技术](https://baike.baidu.com/item/%E5%8A%A0%E5%AF%86%E6%8A%80%E6%9C%AF)又回归到“[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95)保密”的传统上。

常见的加密算法：

[DES](https://baike.baidu.com/item/DES)（Data Encryption Standard）：[对称算法](https://baike.baidu.com/item/%E5%AF%B9%E7%A7%B0%E7%AE%97%E6%B3%95)，[数据加密标准](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%8A%A0%E5%AF%86%E6%A0%87%E5%87%86)，速度较快，适用于加密大量数据的场合。

[3DES](https://baike.baidu.com/item/3DES)（Triple DES）：是基于DES的[对称算法](https://baike.baidu.com/item/%E5%AF%B9%E7%A7%B0%E7%AE%97%E6%B3%95)，对一块数据用三个不同的[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)进行三次加密，强度更高。

[RC2](https://baike.baidu.com/item/RC2)和[RC4](https://baike.baidu.com/item/RC4)：[对称算法](https://baike.baidu.com/item/%E5%AF%B9%E7%A7%B0%E7%AE%97%E6%B3%95)，用变长[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)对大量数据进行加密，比 DES 快。

[IDEA](https://baike.baidu.com/item/IDEA)（International Data Encryption Algorithm）[国际数据加密算法](https://baike.baidu.com/item/%E5%9B%BD%E9%99%85%E6%95%B0%E6%8D%AE%E5%8A%A0%E5%AF%86%E7%AE%97%E6%B3%95)，使用 128 位[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)提供非常强的安全性。

[RSA](https://baike.baidu.com/item/RSA)：由 RSA 公司发明，是一个支持变长[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)的公共密钥[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95)，需要加密的文件块的长度也是可变的，[非对称算法](https://baike.baidu.com/item/%E9%9D%9E%E5%AF%B9%E7%A7%B0%E7%AE%97%E6%B3%95)。

[DSA](https://baike.baidu.com/item/DSA)（Digital Signature Algorithm）：[数字签名](https://baike.baidu.com/item/%E6%95%B0%E5%AD%97%E7%AD%BE%E5%90%8D)[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95)，是一种标准的 DSS（[数字签名标准](https://baike.baidu.com/item/%E6%95%B0%E5%AD%97%E7%AD%BE%E5%90%8D%E6%A0%87%E5%87%86)），严格来说不算加密算法。

[AES](https://baike.baidu.com/item/AES)（Advanced Encryption Standard）：[高级加密标准](https://baike.baidu.com/item/%E9%AB%98%E7%BA%A7%E5%8A%A0%E5%AF%86%E6%A0%87%E5%87%86)，[对称算法](https://baike.baidu.com/item/%E5%AF%B9%E7%A7%B0%E7%AE%97%E6%B3%95)，是下一代的加密算法标准，速度快，安全级别高，在21世纪AES 标准的一个实现是 Rijndael 算法。

[BLOWFISH](https://baike.baidu.com/item/BLOWFISH)，它使用变长的[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)，长度可达448位，运行速度很快。

MD5：严格来说不算加密[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95)，只能说是摘要算法。

PKCS:The Public-Key Cryptography Standards (PKCS)是由美国RSA数据安全公司及其合作伙伴制定的一组公钥密码学标准，其中包括证书申请、证书更新、证书作废表发布、扩展证书内容以及[数字签名](https://baike.baidu.com/item/%E6%95%B0%E5%AD%97%E7%AD%BE%E5%90%8D)、[数字信封](https://baike.baidu.com/item/%E6%95%B0%E5%AD%97%E4%BF%A1%E5%B0%81)的格式等方面的一系列相关协议。

SSF33，SSF28，SCB2(SM1)：国家密码局的隐蔽不公开的商用[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95)，在国内民用和商用的，除这些都不容许使用外，其他的都可以使用。

### 1.4.2 HASH / SHA256

#### 1.4.2.1、HASH[函数](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0/301912)(计算机算法领域）

Hash，一般翻译做“散列”，也有直接音译为“哈希”的，就是把任意长度的[输入](https://baike.baidu.com/item/%E8%BE%93%E5%85%A5/5481954)（又叫做预映射pre-image）通过散列算法变换成固定长度的[输出](https://baike.baidu.com/item/%E8%BE%93%E5%87%BA/11056752)，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来确定唯一的输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的[消息摘要](https://baike.baidu.com/item/%E6%B6%88%E6%81%AF%E6%91%98%E8%A6%81/4547744)的函数。

**HASH函数性质**：

所有[散列函数](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E5%87%BD%E6%95%B0)都有如下一个基本特性：如果两个散列值是不相同的（根据同一函数），那么这两个散列值的原始输入也是不相同的。这个特性是散列函数具有确定性的结果。但另一方面，散列函数的输入和输出不是一一对应的，如果两个散列值相同，两个输入值很可能是相同的，但不绝对肯定二者一定相等（可能出现哈希碰撞）。输入一些数据计算出散列值，然后部分改变输入值，一个具有强混淆特性的散列函数会产生一个完全不同的散列值。

典型的散列函数都有无限定义域，比如任意长度的字节字符串，和有限的值域，比如固定长度的比特串。在某些情况下，散列函数可以设计成具有相同大小的定义域和值域间的一一对应。一一对应的散列函数也称为排列。可逆性可以通过使用一系列的对于输入值的可逆“混合”运算而得到。

#### 1.4.2.2常用HASH函数

·直接取余法：f(x):= x mod maxM ; maxM一般是不太接近 2^t 的一个质数。

·乘法取整法：f(x):=trunc((x/maxX)\*maxlongit) mod maxM，主要用于实数。

·平方取中法：f(x):=(x\*x div 1000 ) mod 1000000); 平方后取中间的，每位包含信息比较多。

构造方法

[散列函数](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E5%87%BD%E6%95%B0)能使对一个数据序列的访问过程更加迅速有效，通过散列函数，[数据元素](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%85%83%E7%B4%A0)将被更快地定位。

1．[直接寻址](https://baike.baidu.com/item/%E7%9B%B4%E6%8E%A5%E5%AF%BB%E5%9D%80)法：取关键字或关键字的某个线性函数值为散列地址。即H(key)=key或H(key) = a·key + b，其中a和b为常数（这种散列函数叫做自身函数）

2． 数字分析法

3． 平方取中法

4． 折叠法

5． 随机数法

6． 除留余数法：取关键字被某个不大于[散列表](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E8%A1%A8)表长m的数p除后所得的余数为散列地址。即 H(key) = key MOD p,p<=m。不仅可以对关键字直接取模，也可在折叠、平方取中等运算之后取模。对p的选择很重要，一般取素数或m，若p选的不好，容易产生同义词。

**处理冲突方法**

1．[开放寻址法](https://baike.baidu.com/item/%E5%BC%80%E6%94%BE%E5%AF%BB%E5%9D%80%E6%B3%95)；Hi=(H(key) + di) MOD m,i=1,2,…，k(k<=m-1)，其中H(key)为[散列函数](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E5%87%BD%E6%95%B0)，m为[散列表](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E8%A1%A8)长，di为增量序列，可有下列三种取法：

1)． di=1,2,3,…，m-1，称线性探测再散列；

2)． di=1^2,(-1)^2,2^2,(-2)^2,(3)^2,…，±(k)^2,(k<=m/2)称二次探测再散列；

3)． di=[伪随机数](https://baike.baidu.com/item/%E4%BC%AA%E9%9A%8F%E6%9C%BA%E6%95%B0)序列，称伪随机探测再散列。

2． 再[散列法](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E6%B3%95)：Hi=RHi(key),i=1,2,…，k RHi均是不同的散列函数，即在同义词产生地址冲突时计算另一个散列函数地址，直到冲突不再发生，这种方法不易产生“聚集”，但增加了计算时间。

3． 链地址法(拉链法)

4． 建立一个公共溢出区

#### 1.4.2.3常用hash算法的介绍：

（1)MD4

MD4(RFC 1320)是 MIT 的[Ronald L. Rivest](https://baike.baidu.com/item/Ronald%20L.%20Rivest)在 1990 年设计的，MD 是 Message Digest（[消息摘要](https://baike.baidu.com/item/%E6%B6%88%E6%81%AF%E6%91%98%E8%A6%81)） 的缩写。它适用在32位[字长](https://baike.baidu.com/item/%E5%AD%97%E9%95%BF)的处理器上用高速软件实现——它是基于 32[位操作](https://baike.baidu.com/item/%E4%BD%8D%E6%93%8D%E4%BD%9C)数的位操作来实现的。

（2)[MD5](https://baike.baidu.com/item/MD5)

MD5(RFC 1321)是 Rivest 于1991年对MD4的改进版本。它对输入仍以512位分组，其输出是4个32位字的级联，与 MD4 相同。MD5比MD4来得复杂，并且速度较之要慢一点，但更安全，在抗分析和抗差分方面表现更好。

（3)[SHA-1](https://baike.baidu.com/item/SHA-1)及其他

SHA1是由NIST NSA设计为同DSA一起使用的，它对长度小于2^64的输入，产生长度为160bit的散列值，因此抗穷举(brute-force)性更好。SHA-1 设计时基于和MD4相同原理,并且模仿了该算法。

#### 1.4.2.4 HASH函数的应用

**错误校正**

使用一个[散列函数](https://baike.baidu.com/item/%E6%95%A3%E5%88%97%E5%87%BD%E6%95%B0)可以很直观的检测出数据在传输时发生的错误。在数据的发送方，对将要发送的数据应用散列函数，并将计算的结果同原始数据一同发送。在数据的接收方，同样的散列函数被再一次应用到接收到的数据上，如果两次散列函数计算出来的结果不一致，那么就说明数据在传输的过程中某些地方有错误了。这就叫做[冗余校验](https://baike.baidu.com/item/%E5%86%97%E4%BD%99%E6%A0%A1%E9%AA%8C)。

**语音识别**

对于像从一个已知列表中匹配一个MP3文件这样的应用，一种可能的方案是使用传统的散列函数——例如MD5，但是这种方案会对时间平移、CD读取错误、不同的音频压缩算法或者音量调整的实现机制等情况非常敏感。使用一些类似于MD5的方法有利于迅速找到那些严格相同（从音频文件的二进制数据来看）的音频文件，但是要找到全部相同（从音频文件的内容来看）的音频文件就需要使用其他更高级的算法了。

**信息安全**

Hash算法在信息安全方面的应用主要体以下的3个方面：

**（1）**[文件校验](https://baike.baidu.com/item/%E6%96%87%E4%BB%B6%E6%A0%A1%E9%AA%8C)

我们比较熟悉的校验算法有[奇偶校验](https://baike.baidu.com/item/%E5%A5%87%E5%81%B6%E6%A0%A1%E9%AA%8C)和CRC校验，这2种校验并没有抗[数据篡改](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%AF%A1%E6%94%B9)的能力，它们一定程度上能检测并纠正数据传输中的信道误码，但却不能防止对数据的恶意破坏。

MD5 Hash算法的"数字指纹"特性，使它成为应用最广泛的一种文件完整性[校验和](https://baike.baidu.com/item/%E6%A0%A1%E9%AA%8C%E5%92%8C)(Checksum)算法，不少Unix系统有提供计算md5 checksum的命令。

**（2)**[数字签名](https://baike.baidu.com/item/%E6%95%B0%E5%AD%97%E7%AD%BE%E5%90%8D)

Hash 算法也是现代密码体系中的一个重要组成部分。由于[非对称算法](https://baike.baidu.com/item/%E9%9D%9E%E5%AF%B9%E7%A7%B0%E7%AE%97%E6%B3%95)的运算速度较慢，所以在数字签名协议中，[单向散列函数](https://baike.baidu.com/item/%E5%8D%95%E5%90%91%E6%95%A3%E5%88%97%E5%87%BD%E6%95%B0)扮演了一个重要的角色。对 Hash 值，又称"[数字摘要](https://baike.baidu.com/item/%E6%95%B0%E5%AD%97%E6%91%98%E8%A6%81)"进行数字签名，在统计上可以认为与对文件本身进行数字签名是等效的。而且这样的协议还有其他的优点。

**（3) 鉴权协议**

如下的鉴权协议又被称作挑战--认证模式：在传输信道是可被侦听，但不可被篡改的情况下，这是一种简单而安全的方法。以上就是一些关于hash以及其相关的一些基本预备知识。

#### 1.4.2.5 SHA256简介

SHA256是安全散列算法SHA（Secure Hash Algorithm）系列算法之一，其摘要长度为256bits，即32个字节，故称SHA256。SHA系列算法是美国国家安全局 （NSA） 设计，美国国家标准与技术研究院（NIST） 发布的一系列密码散列函数，包括 SHA-1、SHA-224、SHA-256、SHA-384 和 SHA-512 等变体。

主要适用于数字签名标准（DigitalSignature Standard DSS）里面定义的数字签名算法（Digital Signature Algorithm DSA）。下面介绍该算法计算消息摘要的原理。

对于任意长度（按bit计算）的消息，SHA256都会产生一个32个字节长度数据，称作消息摘要。当接收到消息的时候，这个消息摘要可以用来验证数据是否发生改变，即验证其完整性。在传输的过程中，数据很可能会发生变化，那么这时候就会产生不同的消息摘要。

SHA算法有如下特性：   
　1. 不可以从消息摘要中复原信息；   
 2. 两个不同的消息不会产生同样的消息摘要。

　　一、术语和概念

　　（一）位(Bit)，字节（Byte）和字（Word）

　　SHA始终把消息当成一个位（bit）字符串来处理。本文中，一个“字”（Word）是32位，而一个“字节”（Byte）是8位。比如，字符串“abc”可以被转换成一个位字符串：01100001 01100010 01100011。它也可以被表示成16进制字符串:0x616263.

#### 1.4.2.6 SHA256算法描述

　　（一）补位

　　信息必须进行补位，以使其长度在对512取模以后的余数是448。也就是说，（补位后的消息长度）Q2 = 448。即使长度已经满足对512取模后余数是448，补位也必须要进行。

　　补位是这样进行的：先补一个1，然后再补0，直到长度满足对512取模后余数是448。总而言之，补位是至少补一位，最多补512位。以信息“abc”为例显示补位的过程。   
　　   
　　原始信息：01100001 01100010 01100011

　　补位第一步：0110000101100010 01100011 1

　　首先补一个“1”

　　补位第二步：0110000101100010 01100011 10…..0

　　然后补423个“0”

　　我们可以把最后补位完成后的数据用16进制写成下面的样子 

1　　 61626380 0000000000000000 00000000

2　　 00000000 0000000000000000 00000000

3 00000000 0000000000000000 00000000

4 00000000 00000000

　　（三）使用的常量

　　在SHA256算法中，用到64个常量，这些常量是对自然数中前64个质数的立方根的小数部分取前32bit而来。这64个常量如下：

　1　 428a2f98 71374491 b5c0fbcf e9b5dba5

2 3956c25b 59f111f1 923f82a4 ab1c5ed5

3 d807aa98 12835b01 243185be 550c7dc3

4 72be5d74 80deb1fe 9bdc06a7 c19bf174

5 e49b69c1 efbe4786 0fc19dc6 240ca1cc

6 2de92c6f 4a7484aa 5cb0a9dc 76f988da

7 983e5152 a831c66d b00327c8 bf597fc7

8 c6e00bf3 d5a79147 06ca6351 14292967

9 27b70a85 2e1b2138 4d2c6dfc 53380d13

10 650a7354 766a0abb 81c2c92e 92722c85

11 a2bfe8a1 a81a664b c24b8b70 c76c51a3

12 d192e819 d6990624 f40e3585 106aa070

13 19a4c116 1e376c08 2748774c 34b0bcb5

14 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3

15 748f82ee 78a5636f 84c87814 8cc70208

16 90befffa a4506ceb bef9a3f7 c67178f2

　　（四）需要使用的函数 

　1　 CH（x, y, z) = (x AND y) XOR ( (NOT x) AND z)

2 MAJ( x, y, z) = (x AND y) XOR (x AND z) XOR (y AND z)

3 BSIG0(x) = ROTR^2(x) XOR ROTR^13(x) XOR ROTR^22(x)

4 BSIG1(x) = ROTR^6(x) XOR ROTR^11(x) XOR ROTR^25(x)

5 SSIG0(x) = ROTR^7(x) XOR ROTR^18(x) XOR SHR^3(x)

6 SSIG1(x) = ROTR^17(x) XOR ROTR^19(x) XOR SHR^10(x)

其中 x、y、z皆为32bit的字。   
ROTR^2(x)是对x进行循环右移2位。

　　（五）计算消息摘要

基本思想：就是将消息分成N个512bit的数据块，哈希初值H(0)经过第一个数据块得到H（1），H（1）经过第二个数据块得到H（2），……，依次处理，最后得到H（N），然后将H（N）的8个32bit连接成256bit消息摘要

I、哈希初值H（0）

SHA256算法中用到的哈希初值H（0）如下

1 H(0)0 = 6a09e667

2 H(0)1 = bb67ae85

3 H(0)2 = 3c6ef372

4 H(0)3 = a54ff53a

5 H(0)4 = 510e527f

6 H(0)5 = 9b05688c

7 H(0)6 = 1f83d9ab

8 H(0)7 = 5be0cd19

注：这些初值是对自然数中前8个质数3、5、7、11等的平方根的小数部分取前32bit而来。

II、 计算过程中用到的三种中间值

1、64个32bit字的message schedule标记为w0、w1、…、w63。

2、8个32bit字的工作变量标记为a、b、c、d、e、f、g。

3、包括8个32bit字的哈希值标记为H(i)0、…、H(i)7。

### 1.4.3 椭圆曲线签名算法（ECDSA)

#### 1.4.3.1、ECDSA概述

椭圆曲线数字签名算法（ECDSA）是使用椭圆曲线密码（ECC）对数字签名算法（DSA）的模拟。ECDSA于1999年成为ANSI标准，并于2000年成为IEEE和NIST标准。它在1998年既已为ISO所接受，并且包含它的其他一些标准亦在ISO的考虑之中。与普通的离散对数问题（discrete logarithm problem  DLP）和大数分解问题（integer factorization problem  IFP）不同，椭圆曲线离散对数问题（elliptic curve discrete logarithm problem  ECDLP）没有亚指数时间的解决方法。因此椭圆曲线密码的单位比特强度要高于其他公钥体制。  
   
[数字签名算法](http://www.shsxt.com/it/html5/653.html)（DSA）在联邦信息处理标准FIPS中有详细论述，称为数字签名标准。它的安全性基于素域上的离散对数问题。椭圆曲线密码（ECC）由Neal Koblitz和Victor Miller于1985年发明。它可以看作是椭圆曲线对先前基于离散对数问题（DLP）的密码系统的模拟，只是群元素由素域中的元素数换为有限域上的椭圆曲线上的点。椭圆曲线密码体制的安全性基于椭圆曲线离散对数问题（ECDLP）的难解性。椭圆曲线离散对数问题远难于离散对数问题，椭圆曲线密码系统的单位比特强度要远高于传统的离散对数系统。因此在使用较短的密钥的情况下，ECC可以达到于DL系统相同的安全级别。这带来的好处就是计算参数更小，密钥更短，运算速度更快，签名也更加短小。因此椭圆曲线密码尤其适用于处理能力、存储空间、带宽及功耗受限的场合

#### 1.4.3.2、ECDSA原理

ECDSA是ECC与DSA的结合，整个签名过程与DSA类似，所不一样的是签名中采取的算法为ECC，最后签名出来的值也是分为r,s。  
 签名过程如下：  
   1、选择一条椭圆曲线Ep(a,b)，和基点G；  
   2、选择私有密钥k（k<n，n为G的阶），利用基点G计算公开密钥K=kG；  
   3、产生一个随机整数r（r<n），计算点R=rG；  
   4、将原数据和点R的坐标值x,y作为参数，计算SHA1做为hash，即Hash=SHA1(原数据,x,y)；  
   5、计算s≡r - Hash \* k (mod n)  
   6、r和s做为签名值，如果r和s其中一个为0，重新从第3步开始执行  
验证过程如下：  
   1、接受方在收到消息(m)和签名值(r,s)后，进行以下运算  
   2、计算：sG+H(m)P=(x1,y1), r1≡ x1 mod p。  
   3、验证等式：r1 ≡ r mod p。  
   4、如果等式成立，接受签名，否则签名无效。

#### 1.4.3.3、JDK中对于ECDSA的实现

特别注意的是：ECDSA签名算法，只是在JDK1.7之后才有实现，最常见的场景是在微软的产品的安装的产品密钥的设计

**1、KeyPairGenerator**

KeyPairGenerator 类用于生成公钥和私钥对。密钥对生成器是使用 getInstance 工厂方法（返回一个给定类的实例的静态方法）构造的。  
特定算法的密钥对生成器可以创建能够与此算法一起使用的公钥/私钥对。它还可以将特定于算法的参数与每个生成的密钥关联。  
有两种生成密钥对的方式：与算法无关的方式和特定于算法的方式。  
   
  
下面我们将按照指定ECDSA算法去生成秘钥KeyPairGenerator.getInstance("EC");

**2、ECDSAPublicKey**

ECDSA公用密钥的接口

**3、ECDSAPublicKey**

 ECDSA 专用密钥的接口

**4、PKCS8EncodedKeySpec**

PKCS8EncodedKeySpec类继承EncodedKeySpec类，以编码格式来表示私钥。  
PKCS8EncodedKeySpec类使用PKCS#8标准作为密钥规范管理的编码格式

**5、Signature**

Signature 类用来为应用程序提供数字签名算法功能。数字签名用于确保数字数据的验证和完整性。  
   
在所有算法当中，数字签名可以是 NIST 标准的 ECDSA，它使用 ECDSA 和 SHA-1。可以将使用 SHA-1 消息摘要算法的 ECDSA 算法指定为SHA1withECDSA。  
1.4.3.4、实现

其中ECDSA的实现步骤类似于我们之前学习的RSA数字签名算法  
实现步骤  
第一步：初始化化秘钥组，生成ECDSA算法的公钥和私钥  
第二步：执行私钥签名， 使用私钥签名，生成私钥签名  
第三步：执行公钥签名，生成公钥签名  
第四步：使用公钥验证私钥签名  
备注：所谓的公钥与私钥匙成对出现。 遵从的原则就是“私钥签名、公钥验证”。  
   
示例代码如下：

|  |
| --- |
| import java.security.KeyFactory;  import java.security.KeyPair;  import java.security.KeyPairGenerator;  import java.security.PrivateKey;  import java.security.PublicKey;  import java.security.Signature;  import java.security.interfaces.ECPrivateKey;  import java.security.interfaces.ECPublicKey;  import java.security.spec.PKCS8EncodedKeySpec;  import java.security.spec.X509EncodedKeySpec;  /\*\*  \* 椭圆曲线签名算法  \*  \* 速度快 强度高 签名短  \*  \* 实现方 JDK1.7/BC  \*/  public class ECDSAUtil {  private static String str = "hello";  public static void main(String[] args) {  jdkECDSA();  }  public static void jdkECDSA() {  try {  KeyPairGeneratorkeyPairGenerator = KeyPairGenerator.getInstance("EC");  keyPairGenerator.initialize(256);  KeyPair keyPair = keyPairGenerator.generateKeyPair();  ECPublicKey ecPublicKey = (ECPublicKey) keyPair.getPublic();  ECPrivateKey ecPrivateKey = (ECPrivateKey) keyPair.getPrivate();  // 2.执行签名  PKCS8Encoded KeySpecpkcs8EncodedKeySpec = new PKCS8EncodedKeySpec(ecPrivateKey.getEncoded());  KeyFactory keyFactory = KeyFactory.getInstance("EC");  PrivateKey privateKey = keyFactory.generatePrivate(pkcs8EncodedKeySpec);  Signature signature = Signature.getInstance("SHA1withECDSA");  signature.initSign(privateKey);  signature.update(str.getBytes());  byte[] sign = signature.sign();  // 验证签名  X509Encoded KeySpecx509EncodedKeySpec=newX509EncodedKeySpec(ecPublicKey.getEncoded());  keyFactory = KeyFactory.getInstance("EC");  PublicKey publicKey = keyFactory.generatePublic(x509EncodedKeySpec);  signature = Signature.getInstance("SHA1withECDSA");  signature.initVerify(publicKey);  signature.update(str.getBytes());  boolean bool = signature.verify(sign);  System.out.println(bool);  } catch (Exception e) {  e.printStackTrace();  }  }  } |
|  |

#### 1.4.3.5、ECDSA标准

ECDSA的标准和标准草案有很多，其中已经过颁发部门批准的有：ANSI X9.62 ,FIPS 186-2,IEEE 1363-2000,ISO 14888-3。ECDSA也被密码标准化组织（SECG，这是一个从事密码标准通用性潜力研究的组织）加以标准化。

**主要的ECDSA标准**

##### 1.ANSI X9.62

该项目始于1995年，并于1999年正式作为ANSI标准颁布。ANSI X9.62具有高安全性和通用性。它的基域可以是Fp，也可以是F2m。F2m中的元素可以以多项式形式或正规基形式来表示。若用多项式形式，ANSI X9.62要求模多项式为不可约三项式，标准中提供了一些不可约三项式，另外还给出了一个不可约五项式。为了提高通用性，针对每一个域提供了一个模多项式。若使用正规基表示方法，ANSI X9.62规定使用高斯正规基。椭圆曲线最主要的安全因素是n，即基点阶，ANSI X9.62的n大于2160。椭圆曲线是使用随机方法选取的。ANSI X9.62规定使用以字节为单位的字符串形式来表示曲线上的点，ASN.1语法可以清楚地描述域参数，公钥和签名。

##### 2.FIPS 186-2

1997年，NIST开始制定包括椭圆曲线和RSA签名算法的FIPS 186标准。1998年，NIST推出了FIPS186，它包括RSA与DSA数字签名方案，这个方案也称为FIPS 186-1。1999年NIST又面向美国G0vment推出了15种椭圆曲线。这些曲线都遵循ANSI X9.62和IEEE 1363-2000的形式。2000年，包含ANSI X9.62中说明的ECDSA，使用上述曲线的FIPS 186-2问世。

##### 3. IEEE 1363-2000

该标准于2000年作为IEEE标准问世。IEEE 1363的覆盖面很广，包括公钥加密，密钥协商，基于IFP、DLP、ECDLP的数字签名。它与ANSI X9.62和FIPS 186完全不同，它没有最低安全性限制（比如不再对基点阶进行限制），用户可以有充分的自由。

因此IEEE 1363-2000并不是一个安全标准，也不具有良好的通用性，它的意义在于给各种应用提供参照。它的基域可以是，也可以是。 中的元素可以以多项式形式或正规基形式来表示。中元素表示形式是整数，中元素表示形式是字符串。这与ANSI X9. 62和FIPS 186是一致的。

##### 4.ISO/IEC 14888-3

这个标准包含若干签名算法，其中ECDSA部分与ANSI X9.62一致。

### 1.4.4 区块链的公钥以及私钥介绍

公钥（Public Key）与私钥（Private Key）是通过算法得到的一个密钥对（即一个公钥和一个私钥），公钥是密钥对中公开的部分,任何人都可以获得，私钥则是非公开的部分,只有用户自己拥有,一旦泄露就会引发安全风险(例如,私钥可以用于签名,如果私钥泄露那么得到私钥的人就可以伪造用户签名了)。公钥通常用于加密会话密钥、验证数字签名，或加密可以用相应的私钥解密的数据。通过这种算法得到的密钥对能保证在世界范围内是唯一的。使用这个密钥对的时候，如果用其中一个密钥加密一段数据，必须用另一个密钥解密。比如用公钥加密数据就必须用私钥解密，如果用私钥加密也必须用公钥解密，否则解密将不会成功。

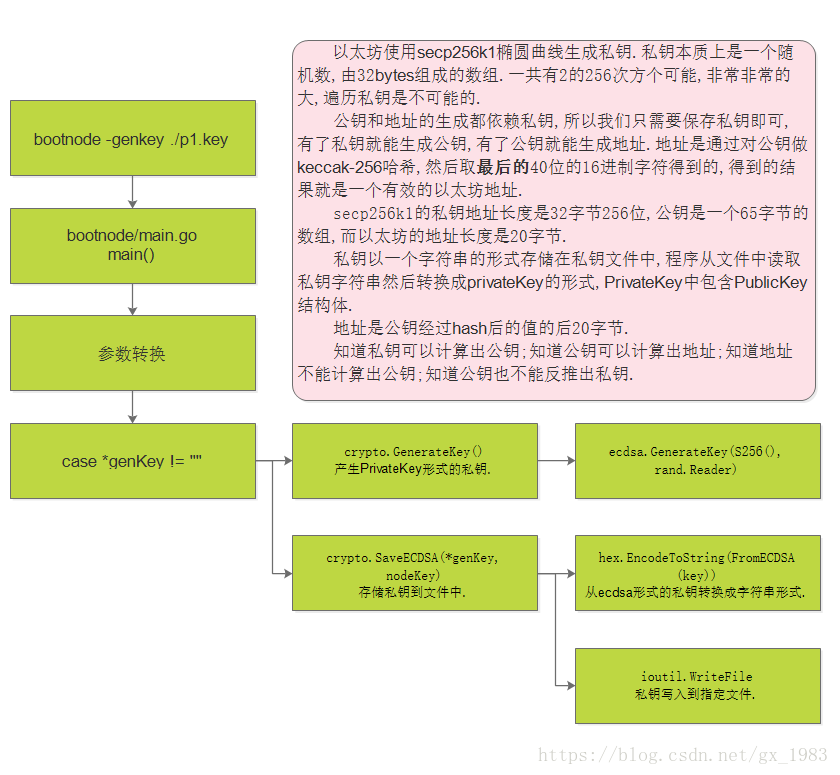
实现分析

使用以太坊的bootnode命令产生私钥文件

通过以太坊的bootnode命令可以生成一个节点的私钥.在p2p发现的过程中,bootnode节点需要这个私钥文件做标识.

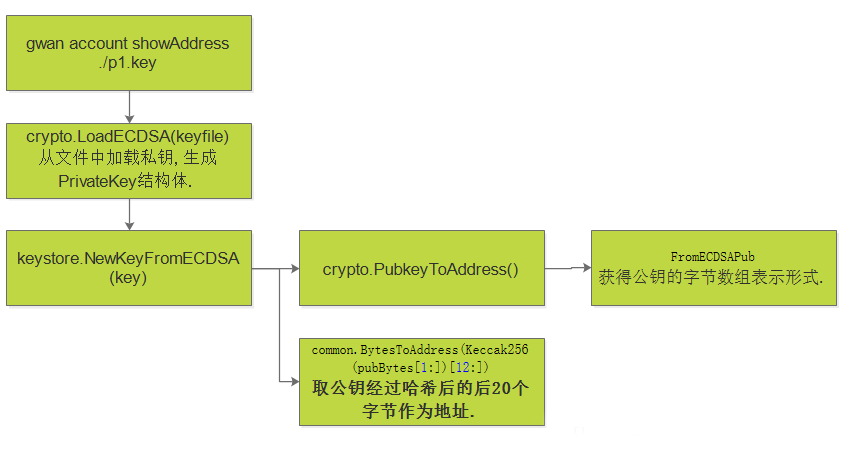
此处的私钥文件不同于keystore文件.keystore文件相关内容参考下述章节.

以太坊bootnode命令生成私钥文件流程图如下:

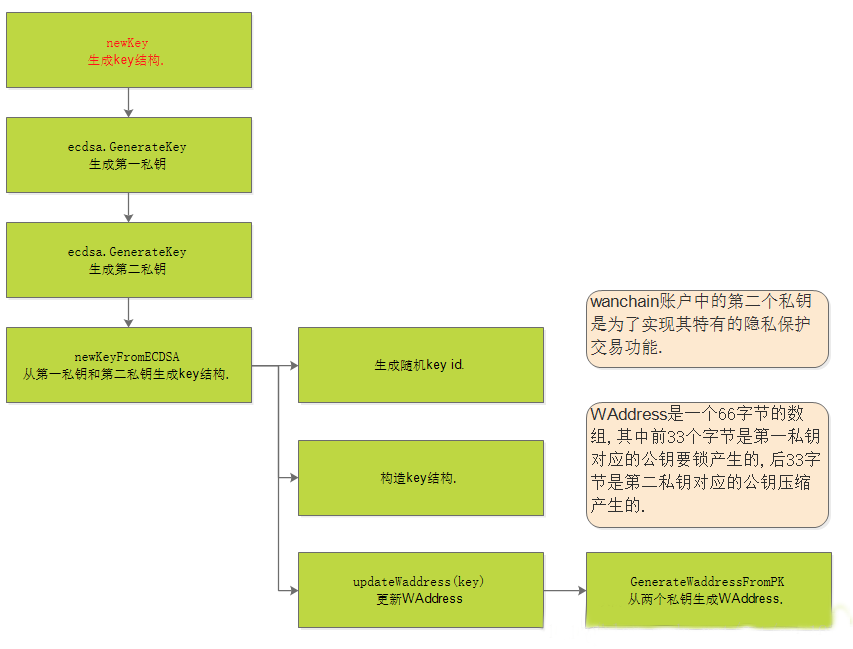


#### 1.4.4.1账户地址的生成过程

* 账户地址是从账户对应的公钥中产生的.
* 截取公钥的后40位(20字节)作为账户对应的地址.
* 获取账户地址的流程如下:



* newKey函数内是真正私钥及将要存储到文件的Key结构的生成过程.在ethereum代码中与wanchain代码中是不同的.wanchain为了实现隐私保护交易在key结构中添加了自己的字段.
* 在以太坊的代码中并Key结构中不存在第二私钥,也没有第二私钥的相关流程.
* Newkey 流程图如



* keystore文件结构
* keystore文件是存储账户相关信息的地方.
* keystore文件实际存储的就是代码生成的Key数据结构,里面保存了用户的私钥及公钥和地址等信息.
  + keystore结构如下:
  + 粉色部分为wanchain代码特有的结构.
  + 蓝色部分为wanchain和以太坊代码共有的结构.
  + wanchain代码的Key中添加了第二私钥的概念,第二私钥及WAddress地址的功能是为了wanchain的隐私保护交易

### 201809111134586341.4.5 数字签名和数字证书

数字签名：

将报文按双方约定的HASH算法计算得到一个固定位数的报文摘要。在数学上保证：只要改动报文中任何一位，重新计算出的报文摘要值就会与原先的值不相符。这样就保证了报文的不可更改性。

将该报文摘要值用发送者的私人密钥加密，然后连同原报文一起发送给接收者，而产生的报文即称数字签名

数字证书：

数字证书就是互联网通讯中标志通讯各方身份信息的一系列数据，提供了一种在Internet上验证您身份的方式，其作用类似于司机的驾驶执照或日常生活中的身份证。它是由一个由权威机构-----CA机构，又称为证书授权（Certificate Authority）中心发行的，人们可以在网上用它来识别对方的身份。数字证书是一个经证书授权中心数字签名的包含公开密钥拥有者信息以及公开密钥的文件。最简单的证书包含一个公开密钥、名称以及证书授权中心的数字签名。

中文案例：

1. 鲍勃有两把钥匙，一把是公钥，另一把是私钥。

2.  鲍勃把公钥送给他的朋友们----帕蒂、道格、苏珊----每人一把。

3.  苏珊要给鲍勃写一封保密的信。她写完后用鲍勃的公钥加密，就可以达到保密的效果

4. 鲍勃收信后，用私钥解密，就看到了信件内容。这里要强调的是，只要鲍勃的私钥不泄露，这封信就是安全的，即使落在别人手里，也无法解密。

5.  鲍勃给苏珊回信，决定采用 "数字签名"。他写完后先用Hash函数，生成信件的摘要（digest）

6. 然后，鲍勃使用私钥，对这个摘要加密，生成"数字签名"（signature）。

7. 鲍勃将这个签名，附在信件下面，一起发给苏珊。

8. 苏珊收信后，取下数字签名，用鲍勃的公钥解密，得到信件的摘要。由此证明，这封信确实是鲍勃发出的。

9. 苏珊再对信件本身使用Hash函数，将得到的结果，与上一步得到的摘要进行对比。如果两者一致，就证明这封信未被修改过。

10.  复杂的情况出现了。道格想欺骗苏珊，他偷偷使用了苏珊的电脑，用自己的公钥换走了鲍勃的公钥。此时，苏珊实际拥有的是道格的公钥，但是还以为这是鲍勃的公钥。因此，道格就可以冒充鲍勃，用自己的私钥做成"数字签名"，写信给苏珊，让苏珊用假的鲍勃公钥进行解密。

11.  后来，苏珊感觉不对劲，发现自己无法确定公钥是否真的属于鲍勃。她想到了一个办法，要求鲍勃去找"证书中心"（certificate authority，简称CA），为公钥做认证。证书中心用自己的私钥，对鲍勃的公钥和一些相关信息一起加密，生成"数字证书"（Digital Certificate）。

12. 鲍勃拿到数字证书以后，就可以放心了。以后再给苏珊写信，只要在签名的同时，再附上数字证书就行了。

13. 苏珊收信后，用CA的公钥解开数字证书，就可以拿到鲍勃真实的公钥了，然后就能证明"数字签名"是否真的是鲍勃签的。

14. 下面，我们看一个应用"数字证书"的实例：https协议。这个协议主要用于网页加密。

15. 首先，客户端向服务器发出加密请求。

16. 服务器用自己的私钥加密网页以后，连同本身的数字证书，一起发送给客户端。

17. 客户端（浏览器）的"证书管理器"，有"受信任的根证书颁发机构"列表。客户端会根据这张列表，查看解开数字证书的公钥是否在列表之内。

18. 如果数字证书记载的网址，与你正在浏览的网址不一致，就说明这张证书可能被冒用，浏览器会发出警告。

19 .如果这张数字证书不是由受信任的机构颁发的，浏览器会发出另一种警告

20.如果数字证书是可靠的，客户端就可以使用证书中的服务器公钥，对信息进行加密，然后与服务器交换加密信息。

# 2、以太坊实战-DApp及智能合约开发

## 2.1以太坊环境搭建

### 2.1.1 搭建本地全节点

### 2.1.2 使用Geth管理节点

### 2.1.3 使用Geth发起和查询交易

### 2.1.4 搭建以太坊私链环境

## 2.2 DApp开发工具Truffle

### 2.2.1 Truffle工具介绍

### 2.2.2 安装开发节点Test RPC

### 2.2.3 安装和配置Truffle

### 2.2.4 使用Truffle自动化测试智能合约

### 2.2.5 使用Truffle部署智能合约

### 2.2.6 使用Truffle开发简单的DApp

## 2.3 Solidity智能合约

### 1、Solidity概述和开发工具的使用

中文教程网站：https://solidity-cn.readthedocs.io/zh/develop/layout-of-source-files.html

#### 1.1 Solidity的概述

Solidity 是一门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。 （静态语言——每个变量包括局部和状态变量都需要在编译时指定变量类型或至少可以推导出类型）

以太坊底层是基于帐户，而非UTXO的，所以有一个特殊的Address的类型。用于定位用户，定位合约，定位合约的代码（合约本身也是一个帐户）。

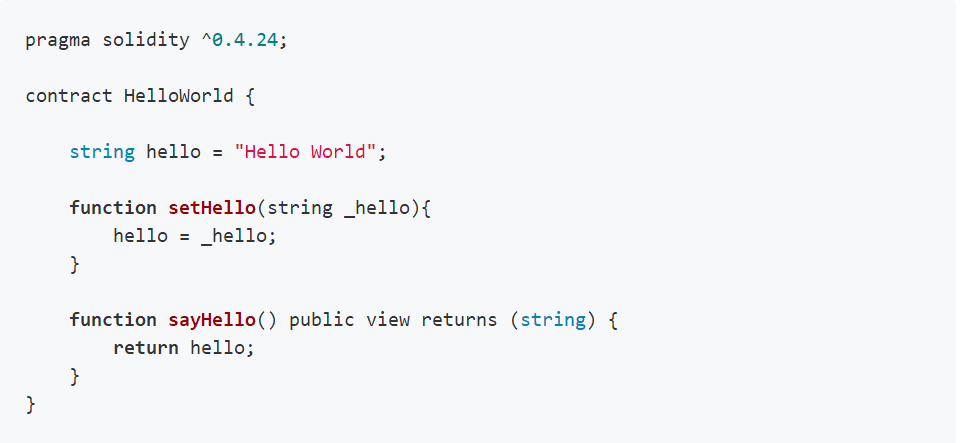
由于语言内嵌框架是支持支付的，所以提供了一些关键字，如payable，可以在语言层面直接支持支付，而且超级简单。

存储是使用网络上的区块链，数据的每一个状态都可以永久存储，所以需要确定变量使用内存，还是区块链。

运行环境是在去中心化的网络上，会比较强调合约或函数执行的调用的方式。因为原来一个简单的函数调用变为了- 一个网络上的节点中的代码执行，分布式的感觉。

最后一个非常大的不同则是它的异常机制，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。

#### 1.2 第一个实例



#### 1.3 如何编写Solidity

目前尝试 Solidity 编程的最好的方式是使用 Remix （需要时间加载，请耐心等待）。Remix 是一个基于 Web 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。

#### 1.4 Solidity开发工具remix的使用

Solidity智能合约开发的IDE有很多，官方推荐的Remix是基于浏览器的。

1.官方工具:

[http://remix.ethereum.org/#optimize=false&version=soljson-v0.4.24+commit.e67f0147.js]

2.操作：手动编译、自动编译、选择合约、查看编译详情、选择运行环境、部署合约、函数调用、调试

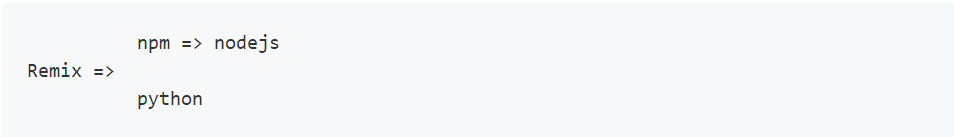
3.Remix分两个版本, 在线版和本地安装版. 在线使用最方便直接用浏览器打开上边的地址。 安装版的界面和可以安装到本地, 安装的方式如下:

#### 1.5 Solidity本地开发工具remix安装

##### 1.5.1 安装nodejs

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。可以在这个环境中运行JavaScript编写的程序。 NPM 是Node.js 的包管理器，是全球最大的开源库生态系统，可以安装各种JavaScript编写的程序。而本地版的remix就是用JavaScript编写的工具，因此依赖nodejs运行环境。另外remix还用到了python语言编写的模块，所以还需要安装python。

因此彼此的依赖关系是：



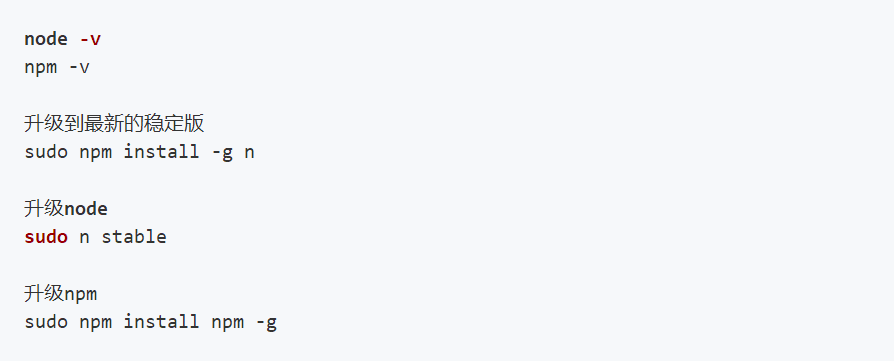
安装Python的步骤：

https://www.runoob.com/python/python-install.html

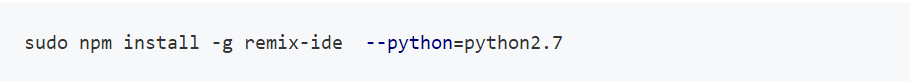
安装nodejs的步骤：

https://www.runoob.com/nodejs/nodejs-install-setup.html

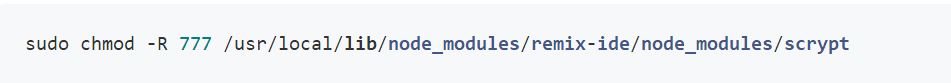
安装成功后打开终端查看版本号：



##### 1.5.2 安装remix-ide

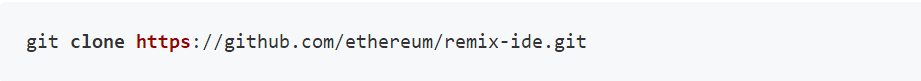


##### 1.5.3 修改文件夹权限



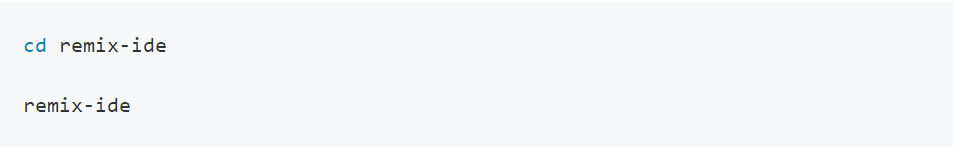
##### 1.5.4 下载安装包

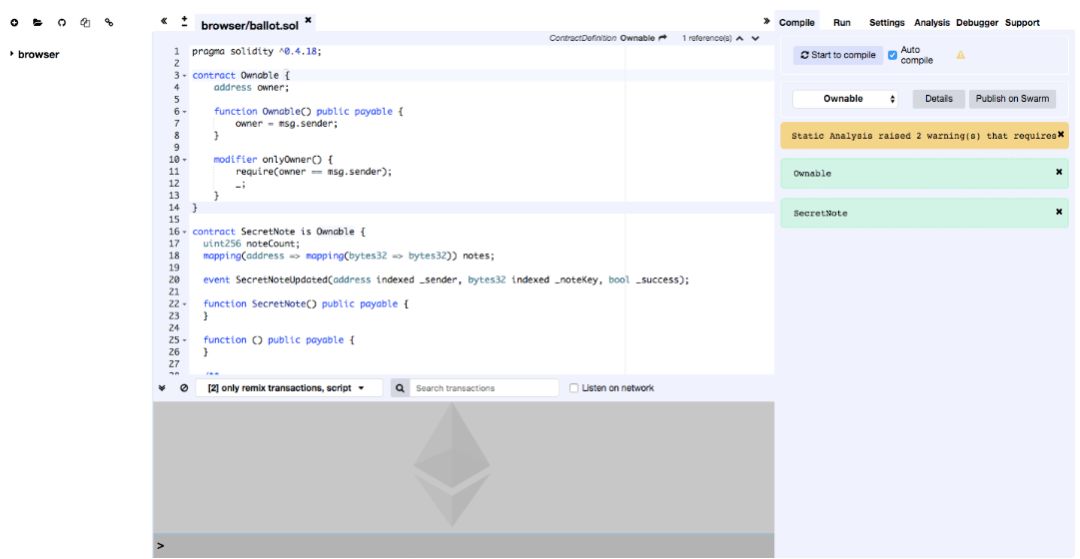
从https://github.com/ethereum/remix-ide下载安装包解压。



##### 1.5.5 启动remix

启动remix，然后打开http://127.0.0.1:8080编辑solidity。





### 2、Solidity源文件布局和合约结构

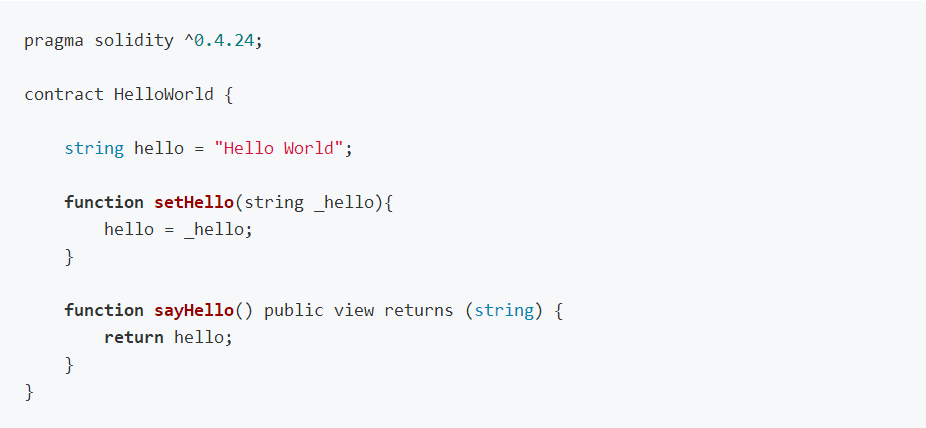
#### 2.1 Solidity源文件的布局——pragma（版本杂注）

说明：

1、版本要高于0.4.24才可以编译；

2、高于0.5的版本则不可编译；

3、第三位的版本号可以变，留出来用做bug可以修复（如0.4.1的编译器有bug，可在0.4.2修复，现有合约不用改代码）。

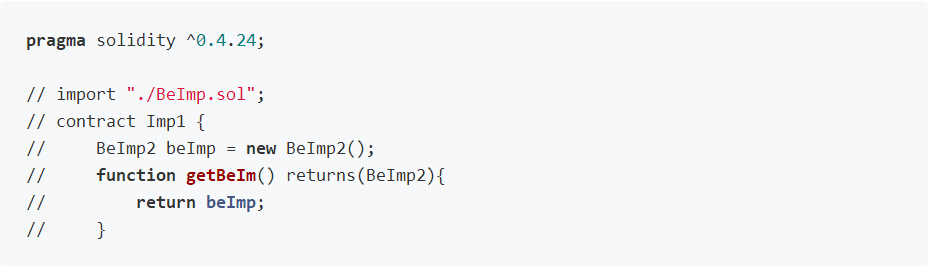


▶源文件应该使用pragma进行注释，防止由于编译器升级导致的不兼容问题。

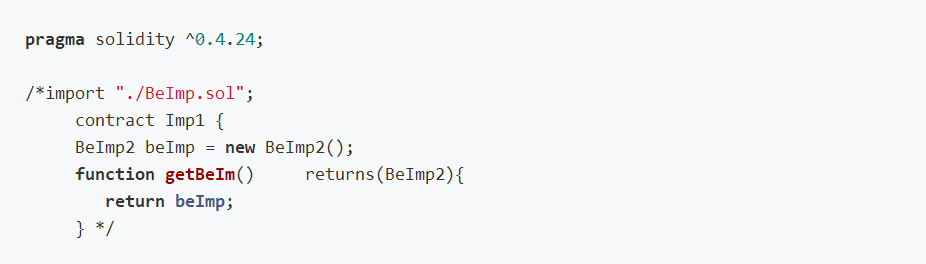
▶源文件可以使用大版本号相同，小版本号高于当前版本的编译器进行编译。

#### 2.2 Solidity源文件的布局——注释

##### 2.2.1 单行注释

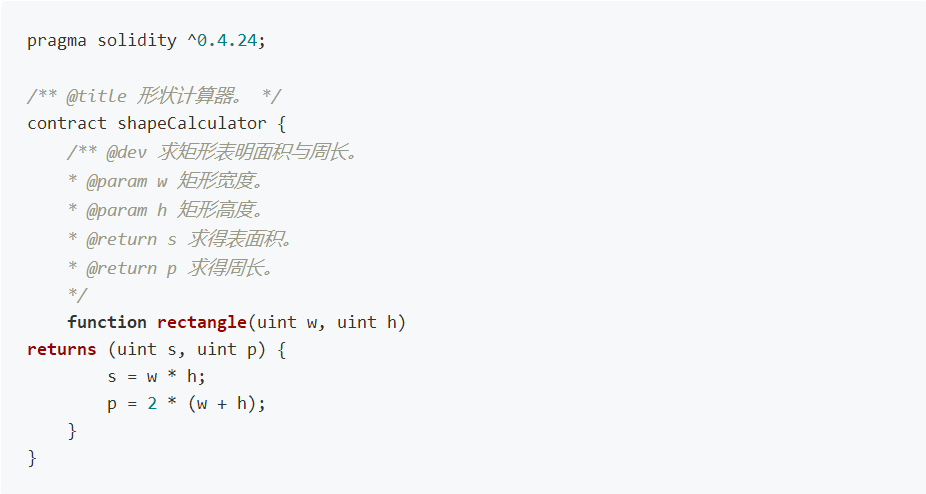


##### 2.2.2 多行注释



##### 2.2.3 natspace注释

natspec 注释，它们是用三个反斜杠（///）或双星号开头的块（/\*\* ... \*/）书写，它们应该直接在函数声明或语句上使用。

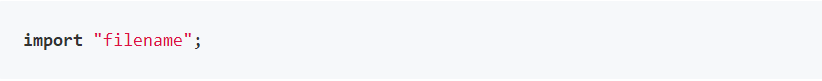


注意：

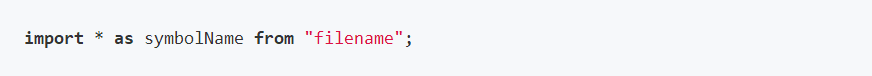
natspec注释在编译过程中会被解析。可以不会使用，但是一定要认识，如果使用，一定要保证参数名与函数一致

#### 2.3 Solidity源文件的布局——import导入其他源文件

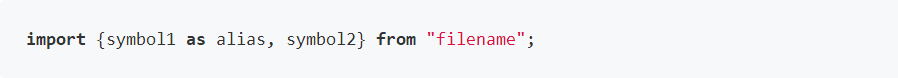
▶在全局层面上导入变量和函数，语句将从“filename”中导入所有的全局符号到当前全局作用域中。



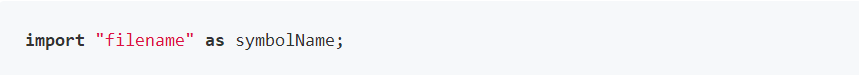
▶创建一个新的全局符号 symbolName，其成员均来自 "filename" 中全局符号。



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



▶这条语句等同于 import \* as symbolName from "filename";



▶关于路径

引入文件路径时要注意，非.打头的路径会被认为是绝对路径，所以要引用同目录下的文件使用。

#### 2.4 编译器解析引用文件机制

各编译器提供了文件前缀映射机制。

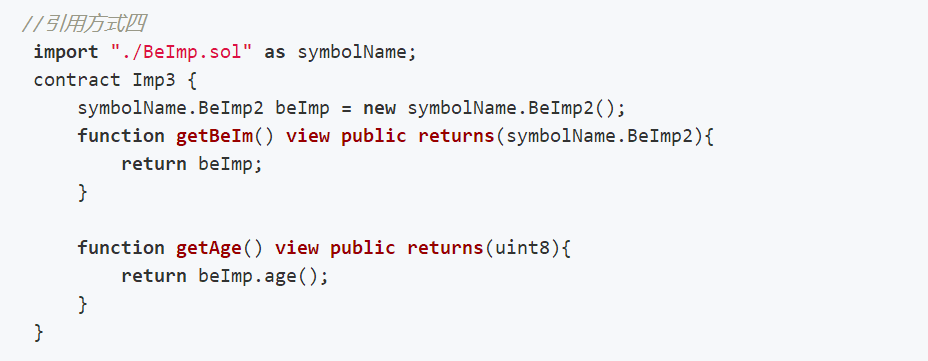
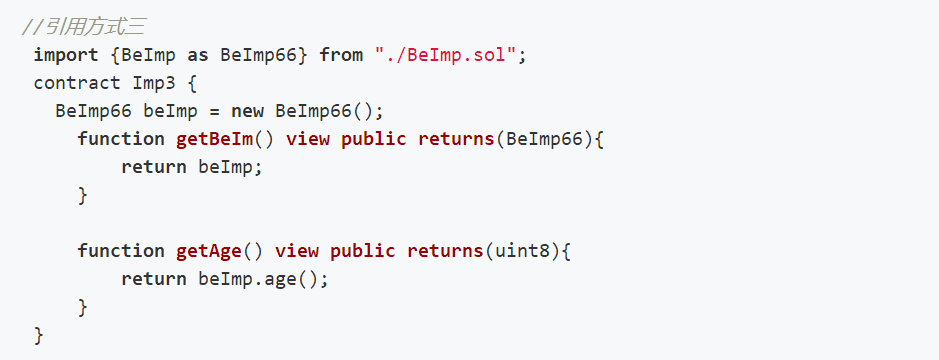
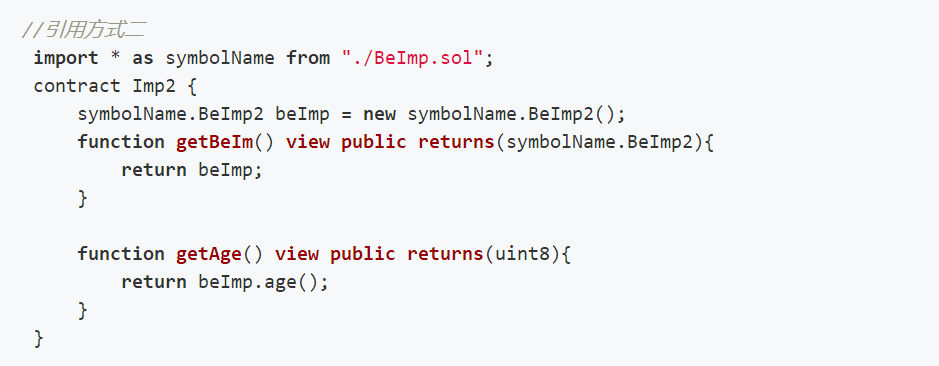
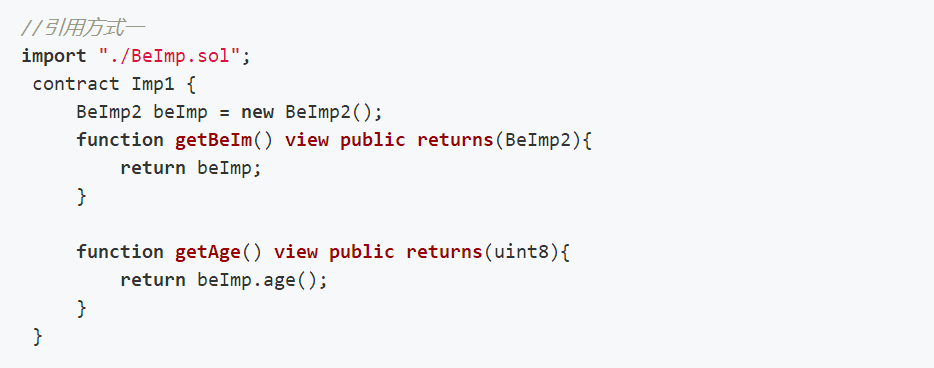
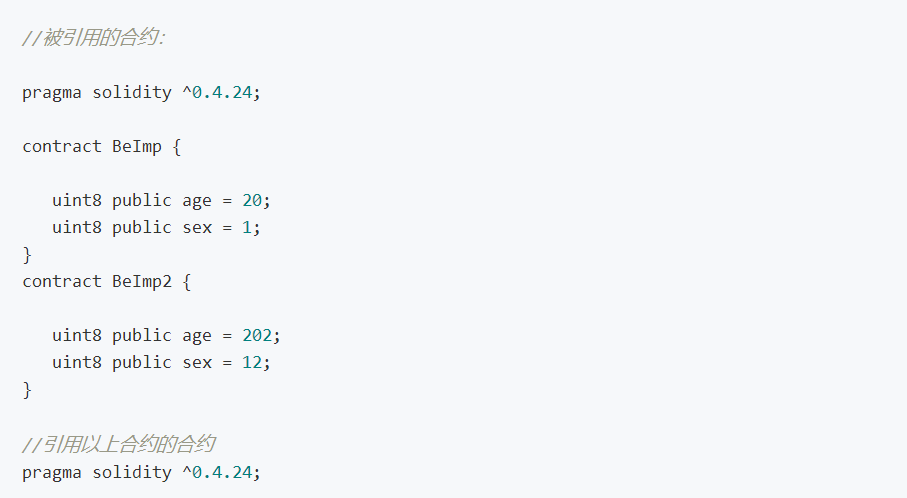
1、可以将一个域名下的文件映射到本地，从而从本地的某个文件中读取；

2、提供对同一实现的不同版本的支持（可能某版本的实现前后不兼容，需要区分）；

3、如果前缀相同，取最长；

4、有一个”fallback-remapping”机制，空串会映射到“/usr/local/include/solidify”。

#### 2.5 import导入其他源文件——实例



注意：

1、导入其他源文件相当于把多个源文件写到一个源文件中；

2、文件名始终被视为具有/作为目录分隔符的路径

▶./作为当前目录

▶../作为父目录（如果没有/ ，则不认为是当前目录或者父目录）

3、如果不明确指定当前目录或者父级目录，引用将从文件系统根目录开始寻找导入文件，这将导致实际导入的文件与目标文件不同（名字相同内容不同）。

#### 2.6 合约结构

在 Solidity 中，合约类似于面向对象编程语言中的类。 每个合约中可以包含：

▶状态变量（State Variables）

▶函数（Functions）

▶函数修改器（Function Modifiers)

▶事件（Events）

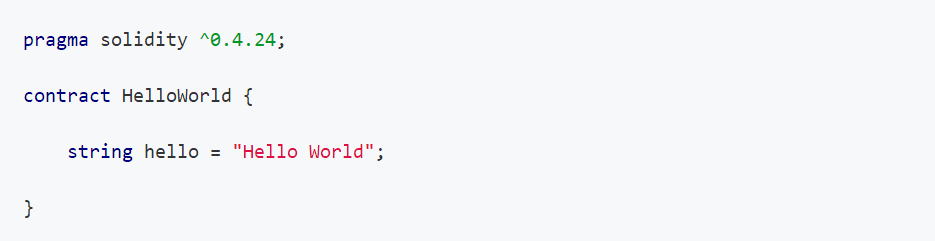
▶结构体（Structs）

▶枚举（Enum ）

（这几个内容本章只是简单介绍一下概念，后续章节会详细讲解。）

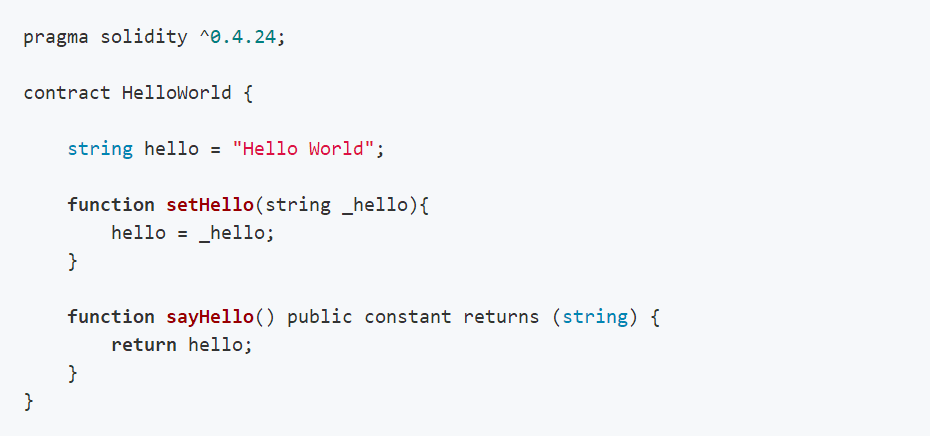
##### 2.6.1 状态变量

状态变量——永久地存储在合约存储中的值。



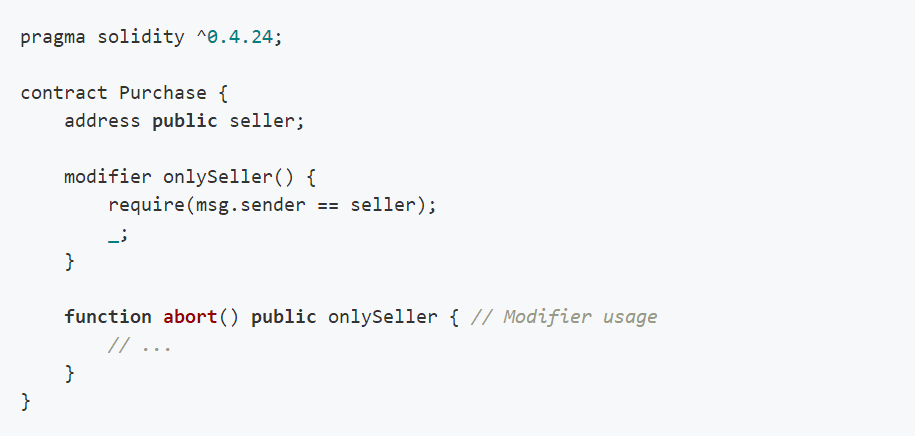
##### 2.6.2 函数

函数——合约中代码的可执行单元。



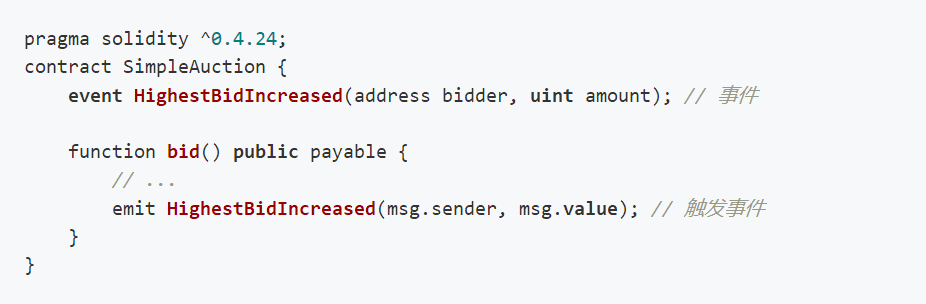
##### 2.6.3 函数修改器

函数修改器——用来以声明的方式改良函数语义。



##### 2.6.4 事件,event

事件——以太坊虚拟机日志工具的方便接口。



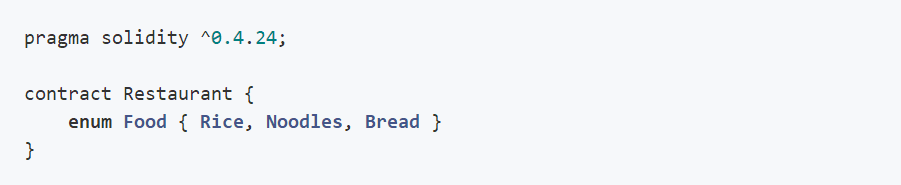
##### 2.6.5 结构体

结构体——可以将几个变量分组的自定义类型（自定义复杂变量的模板）。



##### 2.6.6. 枚举

枚举——枚举可用来创建有一定数量的值的自定义类型。



##### 2.6.7 常量

状态变量可以被定义为constant，常量。这样的话，它必须在编译期间通过一个表达式赋值。赋值的表达式不允许：

▶访问storage；

▶区块链数据，如now，this.balance，block.number；

▶合约执行的中间数据，如msg.gas；

▶向外部合约发起调用。

注意：

不是所有的类型都支持常量，当前支持的仅有值类型和字符串。

#### 2.7 控制结构

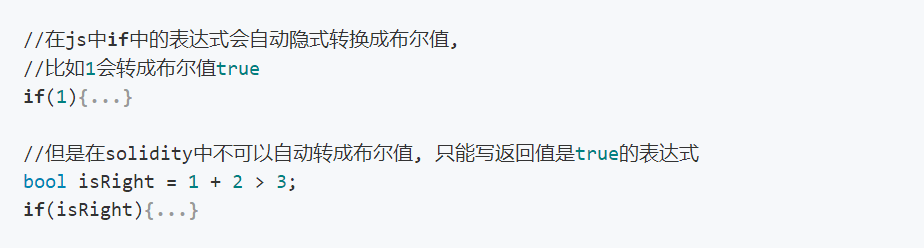
和其它程序语言一样, 编程中不但需要变量常量存储值, 而且需要编写逻辑, 那么逻辑部分就是由这些控制结构负责的：

1、支持：if，else，while，do，for，break，continue，return，二元表达式；

2、不支持：switch和goto；

3、条件判断中的括号不可省略，但在单行语句中的大括号可以省略。 if(true) do('something')；

4、需要注意的是，这里没有像C语言，和javascript里的非Boolean类型的自动转换，比如if(1){...}在Solidity中是无效的。



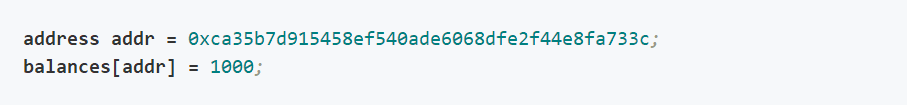
#### 2.8 映射

solidity里的映射可以理解为python里的字典，建立键－值的对应关系，可以通过键来查找值，键必须是唯一的，但值可以重复。

定义方式为：mapping（键类型=>值类型），例如



这个映射的名字是balances，权限类型为public，键的类型是地址address，值的类型是整型uint，在solidity中这个映射的作用一般是通过地址查询余额。键的类型允许除映射外的所有类型。 比如记录一个地址里有1000个eth。



#### 2.9 错误处理

##### 2.9.1 概述

错误处理是指程序发生错误时的处理方式，Solidity处理错误和我们常见的语言不一样，它是通过回退状态的方式处理错误，在发生异常时会撤销当前调用（及其所有子调用）所改变的状态，同时给调用者返回一个错误标识。（没有try catch 不可能捕获异常）。

##### 2.9.2 assert()

assert(bool condition)用于判断内部错误，条件不满足时抛出异常。

特点：注意会消耗掉全部的gas

在下述场景中自动产生assert类型的异常:

▶数组下标越界或者小于0

▶被除数为0， 如5/0 或 23 % 0

▶对一个二进制移动一个负的值。如:5<<i; i为-1时

▶整数进行显式转换为枚举时，过大值、负值

▶调用未初始化内部函数

▶调用assert的参数为false

##### 2.9.3 require（）

require(boolcondition):用于判断输入或外部组件错误，条件不满足时抛出异常。

特点：不会消耗gas

在下述场景中自动产生require类型的异常:

▶调用throw；

▶调用require的参数为false；

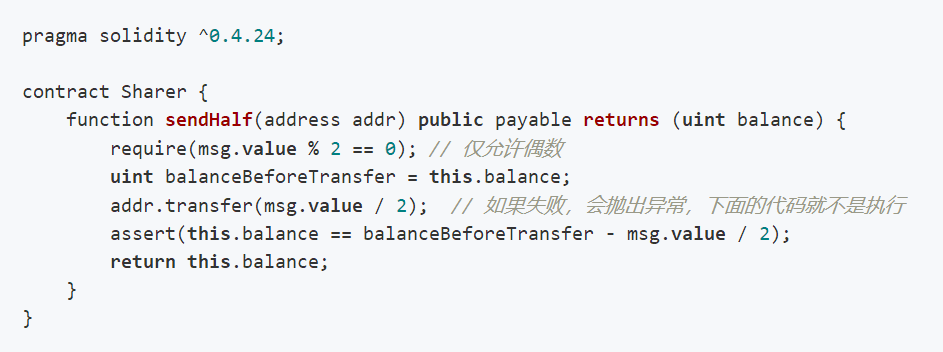
▶调用外部函数调用时，被调用的对象不包含代码；

▶合约没有payable修饰符的public的函数在接收- - 以太币时（包括构造函数，和回退函数）；

▶合约通过一个public的getter函数（public getter funciton）接收以太币；

▶transfer()执行失败。

范例：



##### 2.9.4 其他异常处理方式

除了可以两个函数assert和require来进行条件检查，另外还有两种方式来触发异常：

revert()、revert(string reason)函数可以用来标记错误，终止执行并回退当前调用。



使用throw关键字抛出异常（从0.4.13版本，throw关键字已被弃用，将来会被淘汰。） 当子调用中发生异常时，异常会自动向上“冒泡”。 所谓冒泡就是从当前函数调用逐层向外触发异常。

### 3、Solidity变量类型：值类型

#### 3.1 值类型和引用类型的区别

Solidity变量类型分为两大类——值类型、引用类型

值类型：变量的存储空间存的是变量的数据

引用类型：变量的存储空间存的是变量数据所在的存储空间的地址

注意：值传递和引用传递。值类型的变量，赋值是简单的值传递，即两个变量占有独立的存储区域。引用类型赋值传递的是变量的引用，即两个变量指向同一存储区域

#### 3.2值类型——布尔（bool）

bool: 只有两种值true和false(默认false)。

##### 3.2.1 支持的运算符：

▪ ! 逻辑非

▪ && 逻辑与

▪ || 逻辑或

▪ == 等于

▪ != 不等于

##### 3.2.2 实例

bool a = true;

bool b = !a;

// a == b -> false

// a != b -> true

// a || b -> true

// a && b -> false

逻辑与（&&）和逻辑或（||）都遵循短路原则，即如果根据前一个表达式可以得到运算结果，则不会执行后面的表达式。

#### 3.3 值类型——整型（int/uint）

▪ int(m)：有符号整数

▪ uint(m)：无符号整数

▪ m关键字取值为8~256步幅是8 ，表示在内存中2进制的位数，控制了整数的取值范围，不写默认为256。

▪ uint和int分别是uint256和int256的别名。

▪ m一定要是8的整数倍

##### 3.3.1 操作

比较：<=，<，==，!=，>=，>（结果为bool）

位操作符：&，|，^（按位异或），~（按位取反）

算术运算符：+， - ，一元 - ，一元 +，\*，/，％（取余），\*\*（幂），<<（左移），>>（右移）

注意：

▪ 除法总是截断，但如果两个运算符都是常量（或常量表达式），则它不会截断。

▪ 除零和取余有零引发异常。

▪ 左移几位和右移几位相当于乘以或者除以2的几次方，如果参数为负数的话会引发异常。

▪ 在Solidity中不支持八进制。

#### 3.4 值类型——定点数

##### 3.4.1 定点小数

到现在为止还没有被solidity完全支持，可以被声明，不能被赋值也不能用定点数赋值fixed/ufixed 各种大小的有符号和无符号定点小数，ufixedMxN and fixedMxN关键字M代表定点数占用的二进制位数，N代表定点数能表示多少位小数，M必须是8-256之间的，以8为步幅的整数，N必须是0-80之间的整数，ufixed 和fixed 默认为ufixed128x18和fixed128x18。

▪ 比较运算: <=, <, ==, !=, >=, > (结果为bool)

▪ 算数运算: +, -, 一元-, 一元 +, \*, /, % 取余

#### 3.5 值类型——地址类型（address）

在了解地址之前需要知道一个名词---ABI协议：

Application Binary Interface 应用二进制接口，是从区块链外部与合约进行交互以及合约与合约间进行交互的一种标准方式。ABI是以太坊的一种合约间调用时的一个消息格式。类似Webservice里的SOAP（Simple Object Access Protocol简单对象访问协议）协议一样。

常见格式：

[

{

"constant": false,

"inputs": [

{

"name": "index",

"type": "uint8"

},

{

"name": "s",

"type": "string"

}

],

"name": "removeItem",

"outputs": [

{

"name": "",

"type": "uint256"

}

],

"payable": false,

"stateMutability": "nonpayable",

"type": "function"

}

]

地址：保存一个20字节值，使用40位16进制数表示。地址类型也有成员，并作为所有合约的基础。

##### 3.5.1 地址成员

▪ 以wei位单位返回该地址的余额

<address>.balance(uint256)

▪ 从当前合约地址中给调用函数的地址账户转入amounts数量（以wei为单位）的金额，即从当前合约转账到某账户地址。如果执行失败，将抛出异常，需要支付2300gas的费用，不可以调整。

<address>.transfer(uint256 amount)

其中address就是要把合约中的币转到哪个账号地址。

范例：

pragma solidity ^0.4.24;

contract AddressExample {

//定义一个接受以太币的函数往合约里充值

function AddressExample()payable{}

//函数的参数分别是划转到某账号地址以及转账数量

function giveEthersTo(address \_toAccount,uint amount){

if (this.balance >=amount){

\_toAccount.transfer(amount);

}

}

function getBalance()view returns(uint){

return this.balance;

}

//定义一个匿名的回退函数接收异常情况下退回的以太币

function() payable{}

}

▪ 如果合约地址调用transfer,那么需要合约地址有payable类型的回退函数,并且会随着转账一块执行回退函数中的代码，如果因为回退函数中的代码执行把gas消耗光了，EVM会抛出异常，转账也会被回退。

▪ send是低级对等的转账。执行失败，不会抛出异常，会返回false，需要支付2300gas的费用，不可以调整。推荐使用transfer而不使用send。

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

▪ call(), callcode() 和 delegatecall() 函数。

<address>.call(...) returns (bool)

<address>.callcode(...) returns (bool)

<address>.calldelegate(...) returns (bool)

为了和非ABI协议；也就是定义操作函数签名，参数编码，返回结果编码等的合约进行交互，可以使用call() 函数, 它用来向另一个合约发送原始数据，支持任何类型任意数量的参数，每个参数会按规则(ABI协议)打包成32字节并一一拼接到一起。一个例外是：如果第一个参数恰好4个字节，在这种情况下，会被认为根据ABI协议定义的函数器指定的函数签名而直接使用。如果仅想发送消息体，需要避免第一个参数是4个字节。如下面的例子：

address nameReg = 0x72ba7d8e73fe8eb666ea66babc8116a41bfb10e2;

nameReg.call("register", "MyName");

nameReg.call(bytes4(keccak256("fun(uint256)")), a);

call()

是一个底层的接口，用来向一个合约发送消息，也就是说如果你想实现自己的消息传递，可以使用这个函数。函数支持传入任意类型的任意参数，并将参数打包成32字节，相互拼接后向合约发送这段数据。 call函数返回一个bool值，以表明执行成功与否。正常结束返回true，异常终止返回false。但无法获取到结果数据。还可以提供.gas()修饰器进行调用：

namReg.call.gas(1000000)("register", "MyName");

类似还可以提供附带以太币：

nameReg.call.value(1 ether)("register", "MyName");

修饰器可以混合使用，修饰器调用顺序无所谓。

nameReg.call.gas(1000000).value(1 ether)("register", "MyName");

delegatecall()

call与delegatecall的功能类似，区别仅在于后者仅使用给定地址的代码，仅仅是代码会执行，其它信息则使用当前合约(如存储，余额等等)。

函数的设计目的是为了使用存储在另一个合约的库代码。

所以开发者在提供这样的库时，就要如何安排存储来达到这样的目的。

delegatecall()方法的目的是用来执行另一个合约中的库代码。所以开发者需要保证两个合约中的存储变量能兼容，来保证delegatecall()能顺利执行。

callcode()

未提供对msg.sender，msg.value的访问权限。

call和callcode以及delegateCall的区别：

三者的区别主要体现在三个方面，作用域（上下文）、msg.sender、以及this。

以下是比较示例：

contract D {

uint public n;

address public sender;

function callSetN(address \_e, uint \_n) {

\_e.call(bytes4(sha3("setN(uint256)")), \_n); // 执行结果：改变了E中的变量值， 而当前的同名变量并没改变。

}

function callcodeSetN(address \_e, uint \_n) {

\_e.callcode(bytes4(sha3("setN(uint256)")), \_n); // 执行结果：当前状态变量被改变，而E的并没改变。

}

function delegatecallSetN(address \_e, uint \_n) {

\_e.delegatecall(bytes4(sha3("setN(uint256)")), \_n); //执行结果同callcode

}

}

contract E {

uint public n;

address public sender;

function setN(uint \_n) {

n = \_n;

sender = msg.sender;

// 如果D用call的方式调用，则msg.sender是D的合约地址

//如果D用callcode的方式调用，则msg.sender不会被获取

//如果D用delegateCall的方式调用，msg.sender也不能被获取

// 如果是C中的函数foo()间接调用了. msg.sender则是C

// 如果D用callcode或者delegateCall调用，则this指向D

}

}

contract C {

function foo(D \_d, E \_e, uint \_n) {

\_d.delegatecallSetN(\_e, \_n);

}

}

上面的这三个方法call()，delegatecall()，callcode()都是底层的消息传递调用，最好仅在万不得已才进行使用，因为他们破坏了Solidity的类型安全。 .gas() 在call(), callcode() 和 delegatecall() 函数下都可以使用， delegatecall()不支持.value()。

pragma solidity ^0.4.24;

contract Person{

uint age = 10;

function increaseAge(string name, uint num) returns (uint){

return ++age;

}

function getAge()returns (uint){

return age;

}

}

contract CallTest{

function callByFun(address addr)returns (bool){

bytes4 methodId = bytes4(keccak256("increaseAge(string,uint256)"));

return addr.call(methodId,"jack", 1);

}

}

注意：

合约中使用的this表示当前合约地址；

所有的合约对象都可以被转成地址类型，查询当前合约的余额。

address(this).balance

this.balance

#### 3.6 值类型——定长字节数组

bytes1， ... ，bytes32，允许值以步长1递增。byte默认表示bytes1。

##### 3.6.1 操作

▪ 比较：<=，<，==，!=，>=，>（评估为bool）

▪ 位运算符：&，|，^（按位异或），~（按位取反），<<（左移），>>（右移）

▪ 索引访问：如果x的类型为bytesI，则0 <= k <I的x [k]返回第k个字节（只读）。

##### 3.6.2 成员

.length产生字节数组的固定长度（只读）。

#### 3.7 值类型——有理数和整数字面量

整数常量和有理数常量均支持科学计数法。基数可以是小数，指数必须是整数。 例如2e10,-2e10,2e-10,2.5e10。

##### 3.7.1 有理字面量（特指小数字面量）

▪ 有理数字面量带一个.，在.的两边至少要有一个数字，有效的表示如下1.，.1，1.2

▪ 不允许位运算，小数不能用作指数

▪ 有理数本身支持任意精度，任何运算不会发生溢出或除法截断，当被转换成对应的其他类型，或者与其他类型运算时，不再保证精度。

##### 3.7.2 整数字面量

▪ 由一系列0-9的数字组成的10进制数，存在十六进制表示形式“0x”开头，不存在以“0”八进制的表示形式。

//十六进制表示

0x692a70d2e424a56d2c6c27aa97d1a86395877b3a

▪ 整数常量表达式的除法运算，不做截断处理，结果是有理数。

注意：

数字字面量表达式一旦其中含有非字面量表达式，它就会被转为一个非字面量表达式。

pragma solidity ^0.4.24;

contract IntegerLiteralConvert{

function literalTest(){

uint128 a = 1;

//uint128 b = 2.5 + a + 0.5;

//2.5+a不能转换成一个非字面量表达式

}

}

#### 3.8 值类型——地址字面量

地址字面量表现形式其实就是十六进制整数字面量，如果能够通过地址校验，就会被认为是地址类型，如果不通过则它表示的是一个整数。例如：

0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF

就是一个地址类型的字面量，如果长度为39~40，没有通过地址校验的十六进制整数字面量，会被视为有理数常量，并且会产生一个warning（警告提示）。

#### 3.9 值类型——字符串常量

字符串字面量是用双引号或单引号（“foo”或‘bar’）编写，长度可变。 它们不像C语言那样默认以0结尾; “foo”代表三个不是四个字节。 它们可以隐式转换为bytes1，…，bytes32。

pragma solidity ^0. 4.24;

contract StringLiteralsTest{

bytes15 public name;

function setName(){

name = 'liankuaixueyuan';

}

}

#### 3.10 值类型——十六进制字面量

十六进制字面量，以关键字hex打头，后面紧跟用单或双引号包裹的字符串。如hex"001122ff"。在内部会被表示为二进制流，与字符串的存储形式相同，可以与字符串进行隐式转换，通过下面的例子来理解下是什么意思：

pragma solidity ^0.4.24;

contract HexLiteral{

string name;

bytes nameBytes;

function setName()public{

name = hex"6c69616e6b7561697875657975616e";

nameBytes = hex"6c69616e6b7561697875657975616e";

//nameBytes = hex"a";

//由于一个字节是8位，所以一个hex是由两个[0-9a-z]字符组成的。所以var b = hex“A”;不是成双的，转字符串是会报错的

}

function getName() public view returns (bytes,string){

return (nameBytes,name);

}

}

#### 3.11 值类型——枚举

枚举类型是在Solidity中的一种用户自定义类型。

pragma solidity ^0.4.24;

contract test {

enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }

ActionChoices choice;

ActionChoices constant defaultChoice = ActionChoices.GoStraight;

function setGoStraight() public {

choice = ActionChoices.GoStraight; // 这里通过访问下标的形式得到相同的结果

// choice = ActionChoices(2);

}

function getChoice() public view returns (ActionChoices) {

return choice;

}

function getDefaultChoice() public view returns (uint) {

return uint(defaultChoice);

}

}

注意：

▪ 枚举可以显式的转换与整数进行转换，但不能进行隐式转换。显式的转换会在运行时检查数值范围，如果不匹配，将会引起异常。

▪ 枚举类型应至少有一名成员。

### 4、Solidity函数

#### 4.1 函数的定义

function关键字声明的，合约中的可执行单元，一个函数的完整定义如下：

function (funcName) (<parameter types>) {public|external|internal|private} [constant|view|payable] [returns (<return types>)]

#### 4.2 函数的调用方式

Solidity封装了两种函数的调用方式internal（内部调用）和external（外部调用）。

##### 4.2.1 internal（内部调用方式）

internal调用，实现时转为简单的EVM跳转，所以它能直接使用上下文环境中的数据，对于引用传递时将会变得非常高效（不用拷贝数据）。

在当前的代码单元内，如对合约内函数，引入的库函数，以及父类合约中的函数直接使用即是以internal方式的调用。我们来看个简单的例子：

pragma solidity ^0.4.24;

contract Test {

function f(){}

//以internal的方式调用

function callInternally(){

f();

}

}

在上述代码中，callInternally()以internal的方式对f()函数进行了调用。

简而言之，internal（内部调用方式）就是直接使用函数名去调用函数。

##### 4.2.2 external（外部调用方式）

external调用，实现为合约的外部消息调用。所以在合约初始化时不能external的方式调用自身函数，因为合约还未初始化完成。下面来看一个以external方式调用的例子：

pragma solidity ^0.4.24;

contract A{

function f(){}

}

contract B{

//以external的方式调用另一合约中的函数

function callExternal(A a){

a.f();

}

}

虽然当前合约A和B的代码放在一起，但部署到网络上后，它们是两个完全独立的合约，它们之间的方法调用是通过消息调用。上述代码中，在合约B中的callExternal()以external的方式调用了合约A的f()。

简而言之，external（外部调用方式）就是使用合约实例名.函数名的方式去调用函数。

##### 4.2.3 this

我们可以在合约的调用函数前加this.来强制以external方式的调用。

pragma solidity ^0.4.24;

contract A{

function f() external{}

function callExternally(){

this.f();

}

}

##### 4.2.4 调用方式说明

上面所提到的internal和external指的函数调用方式，请不要与后面的函数可见性声明的external，public，internal，private弄混。声明只是意味着这个函数需要使用相对应的调用方式去调用。

#### 4.3 函数的可见性

Solidity为函数提供了四种可见性，external，public，internal，private。

##### 4.3.1 external（外部函数）

▪ 声明为external的函数可以从其它合约来进行调用，所以声明为external的函数是合约对外接口的一部分。

▪ 不能以internal的方式进行调用。

▪ 有时在接收大的数据数组时性能更好。

pragma3 solidity ^0.4.24;

contract FuntionTest{

function externalFunc() external{}

function callFunc(){

//以`internal`的方式调用函数报错

//Error: Undeclared identifier.

//externalFunc();

//以`external`的方式调用函数

this.externalFunc();

}

}

声明为external的externalFunc()只能以external的方式进行调用，以internal的方式调用会报Error: Undeclared identifier。

##### 4.3.2 pulic（公有函数）

▪ 函数默认声明为public。

▪ public的函数既允许以internal的方式调用，也允许以external的方式调用。

▪ public的函数由于允许被外部合约访问，是合约对外接口的一部分。

pragma solidity ^0.4.24;

contract FuntionTest{

//默认是public函数

function publicFunc(){}

function callFunc(){

//以`internal`的方式调用函数

publicFunc();

//以`external`的方式调用函数

this.publicFunc();

}

}

我们可以看到声明为public的publicFunc()允许两种调用方式。

##### 4.3.3 internal（内部函数）

在当前的合约或继承的合约中，只允许以internal的方式调用。

pragma solidity ^0.4.24;

contract A{

function internalFunc() internal{}

function callFunc(){

//以`internal`的方式调用函数

internalFunc();

}

}

contract B is A{

//子合约中调用

function callFunc(){

internalFunc();

}

}

上述例子中声明为internal的internalFunc()在定义合约，和子合约中均只能以internal的方式可以进行调用。

##### 4.3.4 private（私有函数）

▪ 只能在当前合约中被访问（不可在被继承的合约中访问）。

▪ 即使声明为private，仍能被所有人查看到里面的数据，但是不能修改数据且不能被其它合约访问。

pragma solidity ^0.4.24;

contract A{

function privateFunc() private{}

function callFunc(){

//以`internal`的方式调用函数

privateFunc();

}

}

contract B is A{

//不可调用`private`

function callFunc(){

//privateFunc();

//这里无法调用合约A中的内部函数,

//且在编译阶段就会报错

}

//但是间接调用private函数，但是需要这个private函数处在public中

//function callPrivateByPublicFunc(){

// callFunc();

//}

}

#### 4.4 pure（纯函数）

既不从状态读取数据也不写入数据的函数可以被声明为纯函数 除了之前修改状态数据的情况外，我们认为一下情况属于从状态读取数据。

1. 读取状态变量

2. 调用this.balance或者address.balance

3. 调用block、tx、msg的成员

4. 调用任何非纯函数

5. 使用了包含某些操作码的内联汇编

pragma solidity ^0.4.24;

contract C {

function f(uint a, uint b) public pure returns (uint) {

return a \* (b + 42);

}

}

#### 4.5 constant/view（只读函数）

不改变状态的函数可以被声明为只读函数一下几种情况被视为修改了状态：

1. 修改状态变量

2. 触发事件

3. 创建了其他合约的实例

4. 使用了selfdestruct自我销毁

5. 调用了向合约转账的函数

6. 调用了非只读函数或者纯函数

7. 使用了底层调用

8. 使用了包含某些操作码的内联汇编

注意：

constant是view的一个别名，会在0.5.0版本中遗弃，访问器(getter)方法默认被标记为view调用只读函数。

#### 4.6 函数修改器

在实际情况中，我们经常需要对调用者进行一些限制。比如，只能是合约的所有者才能改变归属。我们一起来看看如何用函数修改器实现这一限制：

pragma solidity ^0.4.24;

contract Ownable {

address public owner = msg.sender;

/// 限制只有创建者才能访问

modifier onlyOwner {

if (msg.sender != owner) throw;

\_;

}

/// 改变合约的所有者 function changeOwner(address \_newOwner)

onlyOwner

{

if(\_newOwner == 0x0) throw;

owner = \_newOwner;

}

}

##### 4.6.1 函数修改器支持参数

pragma solidity ^0.4.24;

contract Parameter{

uint balance = 10;

modifier lowerLimit(uint \_balance, uint \_withdraw){

if( \_withdraw < 0 || \_withdraw > \_balance) throw;

\_;

}

//含参数的函数修改器

function f(uint withdraw) lowerLimit(balance, withdraw) returns (uint){

return balance;

}

}

在上面的例子中，f()函数，有一个函数修改器lowerLimit()，传入了状态变量参数balance，和入参withdraw，以lowerLimit(balance, withdraw)的方式进行调用。最后函数能否正确执行取决于输入的withdraw值大小。

##### 4.6.2 函数修改器参数支持表达式

pragma solidity ^0.4.24;

contract ParameterExpression{

modifier m(uint a){

if(a > 0)

\_;

}

function add(uint a, uint b) private returns(uint){

return a + b;

}

function f() m(add(1, 1)) returns(uint){

return 1;

}

}

#### 4.7 payable（接收以太币函数）

是声明了该函数涉及接收以太币操作，如果函数没有声明为payable,并且在调用过程中有以太币通过被调用的函数转入合约，那么EVM虚拟机将会抛出异常，状态回退。

pragma solidity ^0.4.24;

contract AddressExample {

function AddressExample() payable{}

function giveEthersTo(address \_toAccount,uint amount){

if (this.balance >=amount){

\_toAccount.transfer(amount);

}

}

function getBalance() view returns(uint){

return this.balance;

}

//function() payable{}

}

#### 4.8 回退函数

每一个合约有且仅有一个没有名字的函数。这个函数无参数，也无返回值。如果调用合约时，没有匹配上任何一个函数(或者没有传哪怕一点数据)，就会调用默认的回退函数。

此外，当合约收到ether时（没有任何其它数据），这个函数也会被执行。在此时，一般仅有少量的gas剩余，用于执行这个函数(准确的说，还剩2300gas)。所以应该尽量保证回退函数使用少的gas。

下述提供给回退函数可执行的操作会比常规的花费得多一点。

写入到存储(storage) 创建一个合约 执行一个外部(external)函数调用，会花费非常多的gas 发送ether 请在部署合约到网络前，保证透彻的测试你的回退函数，来保证函数执行的花费控制在2300gas以内。

一个没有定义一个回退函数的合约。如果接收ether，会触发异常，并返还ether（solidity v0.4.0开始）。所以合约要接收ether，必须实现回退函数。下面来看个例子。下面来看个例子：

pragma solidity ^0.4.24;

contract Test {

function() public payable{}

function getX() view returns(uint){

return x;

}

function getBalance() view returns(uint){

return this.balance;

}

}

contract Caller {

function Caller()payable{}

function callTest(Test test) public{

test.call(0xabcdef01);

// test.transfer(2 ether);

}

function getBalance() view returns(uint){

return this.balance;

}

}

如果涉及支付以太币，即回退函数被声明为payable类型，并且通过send或者transfer被调用，那么回退函数仅有你2300gas可以使用，如果回退函数中的代码执行消耗超过2300gas那么被转入的以太币将会退回，修改过的数据状态回退。

以下操作会消耗超过2300gas：

1. 修改状态变量

2. 创建新的合约实例

3. 调用了会消耗gas较多的外部函数

4. 发送以太币

用做接收以太币回退函数内部仅能进行触发事件操作。

#### 4.9 构造函数

构造函数是一个用constructor关键字声明的可选函数，它在创建合约时执行。构造函数可以是public，也可以是internal。如果没有构造函数，则该合约将生成默认构造函数：contructor() public {}。

pragma solidity ^0.4.24;

contract A {

uint public a;

constructor(uint \_a) internal {

a = \_a;

}

}

在版本0.4.22之前，构造函数被定义为与合同名称相同的特殊函数，有且只能有一个，不允许重载。这个函数将在合约创建时，执行一次，用于初始化一些配置。这个语法现在不推荐使用。

pragma solidity ^0.4.24;

contract ContractConstructor{

uint public counter;

function ContractConstructor(){

counter++;

}

}

上述合约在创建成功后，counter的值将为1。说明合约在创建时，被调用了一次。

#### 4.10 函数的输入参数与输出参数

Solidity函数的输入参数的数量是可选的，也可以有任意数量的返回参数。

入参(Input Parameter)与变量的定义方式一致，稍微不同的是，不会用到的参数可以省略变量名称。一种可接受两个整型参数的函数如下:

pragma solidity ^0.4.0;

contract Simple {

function taker(uint \_a, uint) {

// do something with \_a.

}

}

出参（Output Paramets）在returns关键字后定义，语法类似变量的定义方式。返回结果的数量需要与定义的一致。如果给定了参数名，则函数可以不适用return关键字返回，如果没有给定参数名则需要函数体重使用return关键字按照顺序返回。

pragma solidity ^0.4.24;

contract Simple {

//return sum and product

function arithmetics(uint \_a, uint \_b) returns (uint o\_sum, uint o\_product) {

o\_sum = \_a + \_b;

o\_product = \_a \* \_b;

}

}

pragma solidity ^0.4.24;

contract Simple {

//return sum and product

function arithmetics(uint \_a, uint \_b) pure returns (uint , uint ) {

return(\_a + \_b,\_a \* \_b);

}

}

#### 4.11 访问函数

编译器为自动为所有的public的状态变量创建访问函数。下面的合约例子中，编译器会生成一个名叫data的无参，返回值是uint的类型的值data。状态变量的初始化可以在定义时完成。

pragma solidity ^0.4.0;

contract C{

uint public c = 10;

}

contract D{

C c = new C();

function getDataUsingAccessor() returns (uint){

return c.c();

}

}

访问函数有外部(external)可见性。如果通过内部(internal)的方式访问，比如直接访问，你可以直接把它当一个变量进行使用，但如果使用外部(external)的方式来访问，如通过this.，那么它必须通过函数的方式来调用。

pragma solidity ^0.4.0;

contract C{

uint public c = 10;

function accessInternal() returns (uint){

return c;

}

function accessExternal() returns (uint){

return this.c();

}

}

#### 4.12 抽象函数

是没有函数体的的函数。如下：

pragma solidity ^0.4.0;

contract Feline {

function utterance() returns (bytes32);

}

这样的合约不能通过编译，即使合约内也包含一些正常的函数。但它们可以做为基合约被继承。

pragma solidity ^0.4.0;

contract Feline {

function utterance() returns (bytes32);

function getContractName() returns (string){

return "Feline";

}

}

contract Cat is Feline {

function utterance() returns (bytes32) { return "miaow"; }

}

如果一个合约从一个抽象合约里继承，但却没实现所有函数，那么它也是一个抽象合约。

#### 4.13 数字和加密函数

以下函数式solidity自带的函数

asser(bool condition):

如果条件不满足，抛出异常。

addmod(uint x, uint y, uint k) returns (uint):

计算(x + y) % k。加法支持任意的精度。但不超过(wrap around？)2\*\*256。

mulmod(uint x, uint y, uint k) returns (uint):

计算(x \* y) % k。乘法支持任意精度，但不超过(wrap around？)2\*\*256。

keccak256(...) returns (bytes32):

使用以太坊的（Keccak-256）计算HASH值。紧密打包。

sha3(...) returns (bytes32):

等同于keccak256()。紧密打包。

sha256(...) returns (bytes32):

使用SHA-256计算HASH值。紧密打包。

ripemd160(...) returns (bytes20):

使用RIPEMD-160计算HASH值。紧密打包。

ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address):

通过签名信息恢复非对称加密算法公匙地址。如果出错会返回0，例如:

function verify(bytes32 hash, uint8 v, bytes32 r, bytes32 s) constant returns(bool) {

bytes memory prefix = "\x19Ethereum Signed Message:\n32";

bytes32 prefixedHash = keccak256(prefix, hash);

return ecrecover(prefixedHash, v, r, s) == (Your Address);

}

revert()：

取消执行，并回撤状态变化。

需要注意的是参数是“紧密打包（tightly packed）”的，意思是说参数不会补位，就直接连接在一起的。下面来看一个例子:

keccak256("ab", "c")

keccak256("abc")

//hex

keccak256(0x616263)

keccak256(6382179)

//ascii

keccak256(97, 98, 99)

上述例子中，三种表达方式都是一致的。

如果需要补位，需要明确的类型转换，如keccak256("\x00\x12")等同于keccak256(uint16(0x12))

需要注意的是字面量会用，尽可能小的空间来存储它们。比如，keccak256(0) == keccak256(uint8(0))，keccak256(0x12345678) == keccak256(uint32(0x12345678))

注意:

在私链(private blockchain)上运行sha256,ripemd160或ecrecover可能会出现Out-Of-Gas报错。因为它们实现了一种预编译的机制，但合约要在收到第一个消息后才会存在。向一个不存在的合约发送消息，非常昂贵，所以才会导致Out-Of-Gas的问题。一种解决办法是每个在你真正使用它们前，先发送1 wei到这些合约上来完成初始化。在官方和测试链上没有这个问题。

### 5、Solidity变量类型:引用类型

#### 5.1 引用类型——数据位置

不同于之前值类型，引用类型占的空间更大，超过256字节，因为拷贝它们占用更多的空间。由此我们需要考虑将它们存储在什么位置?内存（memory，数据不是永久存在的）或存储(storage，数据永久的保存在数据块上)

##### 5.1.1 数据位置分类

▪memory：存储位置同我们普通程序的内存类似。即分配，即使用，越过作用域即不可被访问，等待被回收。

▪storage：数据将永远存在于区块链上。

▪calldata：一般只有外部函数的参数（不包括返回参数）被强制指定为calldata。这种数据位置是只读的，不会持久化到区块链。

▪stack：

1. 栈实际是内存中的一个数据结构，每个栈元素占256位，栈的最大深度位1024；

2. 值类型的局部变量存储在栈中；

3. 在栈中保存一个很小的局部变量，gas开销最小，几乎免费使用，但是数量有限。

基于程序的上下文，大多数时候数据位置的选择是默认的，我们可以通过在变量名前声明memory还是storage来定义该变量的数据位置。

##### 5.1.2 数据位置默认规则

▪ 函数参数、函数返回参数默认为memory

▪ 局部变量（作用域为局部）以及状态变量（作用域为全局）默认storage类型。

##### 5.1.3 强制的数据位置

▪ 外部函数(External function)的参数(不包括返回参数)强制为：calldata。

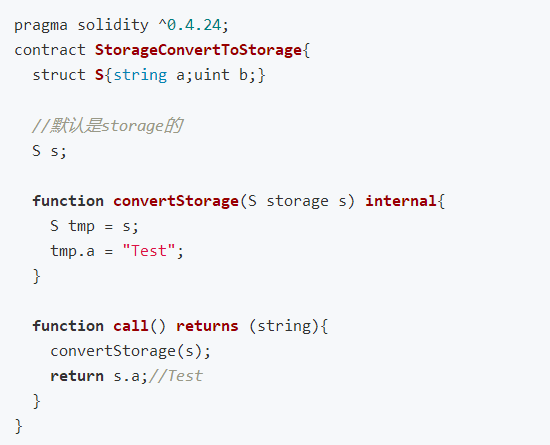
▪ 状态变量(State variables)强制为: storage。

▪ 值类型的局部变量是存储在栈上。

##### 5.1.4 不同数据位置变量间相互赋值

**Storage**

当我们把一个storage类型的变量赋值给另一个storage时，只是修改了它的指针（引用传递）。



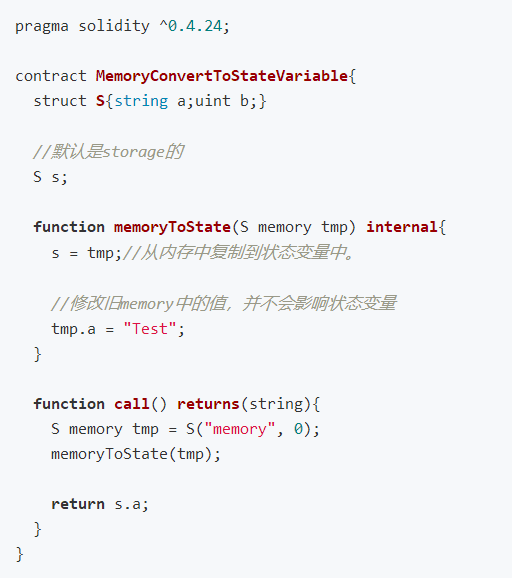
在上面的代码中，我们将传入的storage变量，赋值给另一个临时的storage类型的tmp时，并修改tmp.a = "Test"，最后我们发现合约的状态变量s也被修改了。

**memory给storage赋值**

因为局部变量和状态变量的类型都可能是storage。所以我们要分开来说这两种情况。

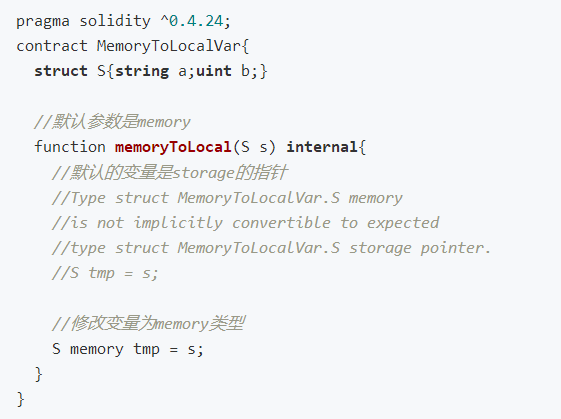
1. memory给storage赋值

将一个memory类型的变量赋值给一个状态变量时，实际是将内存变量拷贝到存储中。



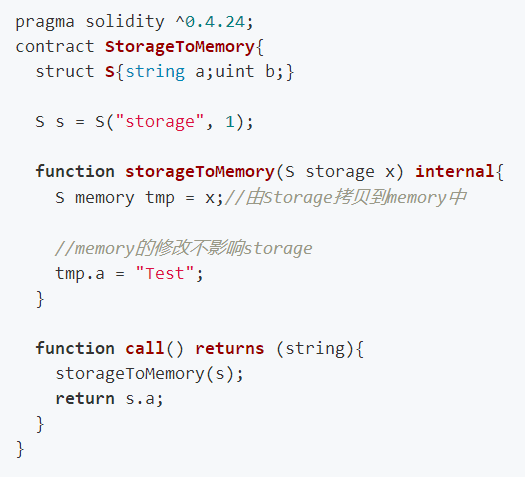
2. memory赋值给状态变量

由于在区块链中，storage必须是静态分配存储空间的。局部变量虽然是一个storage的，但它仅仅是一个storage类型的指针。如果进行这样的赋值，实际会产生一个错误。



通过上面的代码，我们可以看到这样的赋值的确不被允许。你可以通过将变量tmp改为memory来完成这样的赋值。

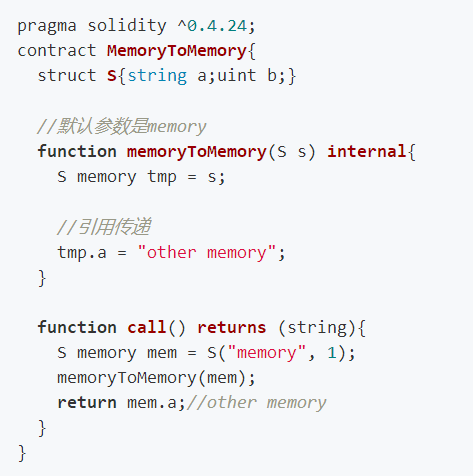
3. storage转为memory



在上面的例子中，我们看到，拷贝后对tmp变量的修改，完全不会影响到原来的storage变量。

4. memory转为memory

memory之间是引用传递，并不会拷贝数据。我们来看看下面的代码。



在上面的代码中，memoryToMemory()传递进来了一个memory类型的变量，在函数内将之赋值给tmp，修改tmp的值，发现外部的memory也被改为了other memory。

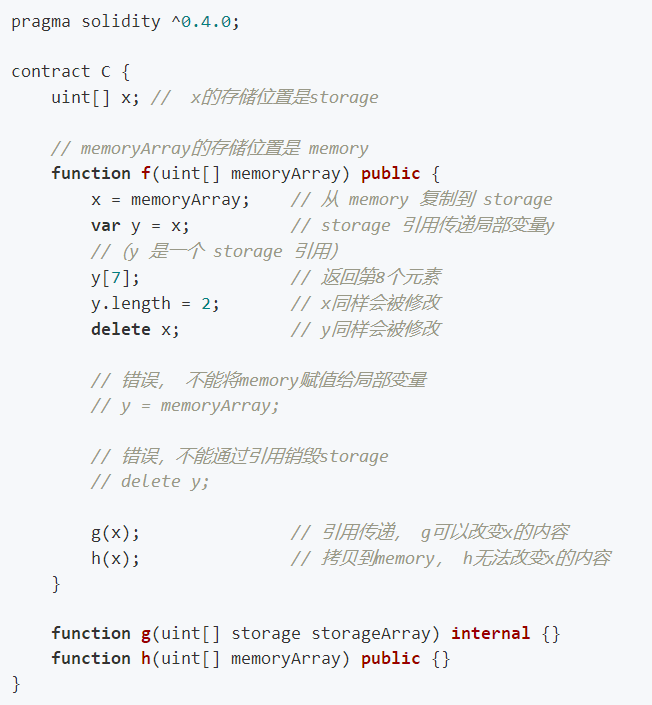
注意：

1. 对于值类型，总是会进行拷贝

2. 不能将memory的函数参数赋值给storage局部变量

3. 不能通过引用销毁storage

##### 5.1.5 实例



##### 5.1.6 不同存储的消耗（gas消耗）

▪ storage 会永久保存合约状态变量，开销最大. 大概5000~20000。

▪ memory 仅保存临时变量，函数调用之后释放，开销很小。

▪ calldata 和memory差不多。

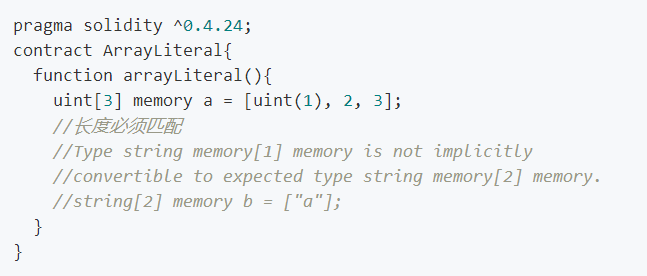
▪ stack 保存很小的局部变量，几乎免费使用，但有数量限制 具体gas消耗值请参考http://yellowpaper.io/。

#### 5.2 引用类型——数组

数组在所有的语言当中都是一种常见类型。在Solidity中，可以支持编译期定长数组和变长数组。一个类型为T，长度为k的数组，可以声明为T[k]，而一个变长的数组则声明为T[]。

##### 5.2.1 使用字面量创建数组

创建数组时，我们可以使用字面量，隐式创建一个定长数组。



通过上面的代码，我们可以发现。

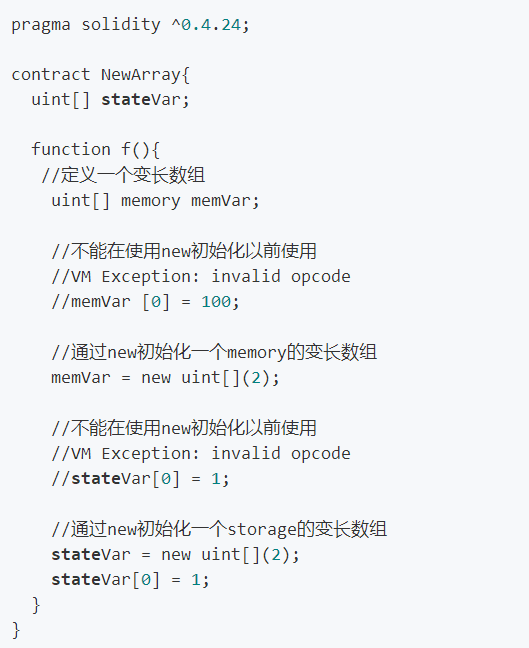
首先元素类型是刚好能存储的元素的类型，比如代码里的[1, 2, 3]，只需要uint8即可存储。但由于我们声明的变量是uint（默认的uint表示的其实是uint256），所以要使用uint(1)来进行显式的类型转换。

其次，字面量方式声明的数组是定长的，且实际长度要与声明的相匹配，否则编译器会报错

Type string memory[1] memory is not implicitly convertible to expected type string memory[2] memory。

##### 5.2.2 使用new关键字创建数组

对于变长数组，在初始化分配空间前不可使用，可以通过new关键字来初始化一个数组。

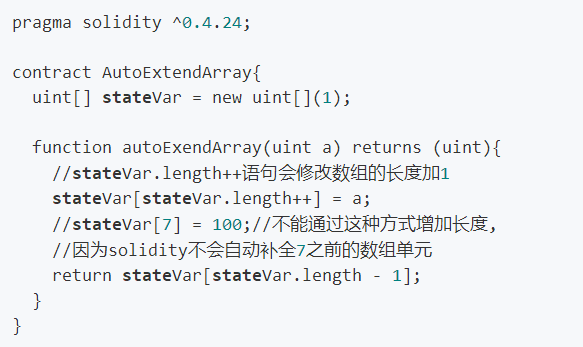


我们声明了一个storage的stateVar，和一个memory的memVar。它们不能在使用new关键字初始化前使用下标方式访问，会报错VM Exception: invalid opcode。可以根据情况使用如例子中的new uint;来进行初始化。

##### 5.2.3 数组属性

1 length属性

数组有一个length属性，表示当前的数组长度。对于storage的变长数组，可以通过给length赋值调整数组长度。



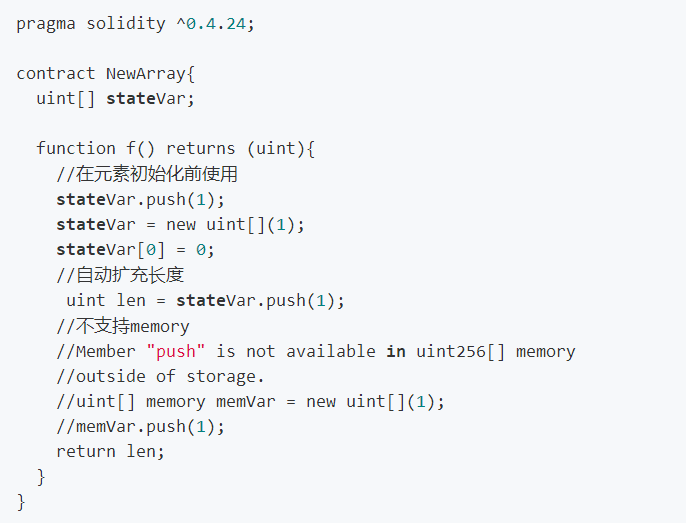
在上面这个例子中，我们可以看到，通过stateVar.length++语句对数组长度进行自增，我们就得到了一个不断变长的数组。 还可以使用后面提到的push()方法，来隐式的调整数组长度。

##### 5.2.4 数组函数

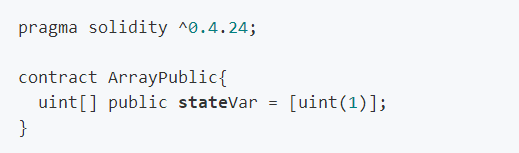
变长的storage数组和bytes（不包括string）有一个push()函数。可以将一个新元素附加到数组末端，返回值为当前长度。push函数支持数组的初始化。

##### 5.2.5 memory数组

对于memory的变长数组，不支持修改length属性，来调整数组大小。memory的变长数组虽然可以通过参数灵活指定大小，但一旦创建，大小不可调整。



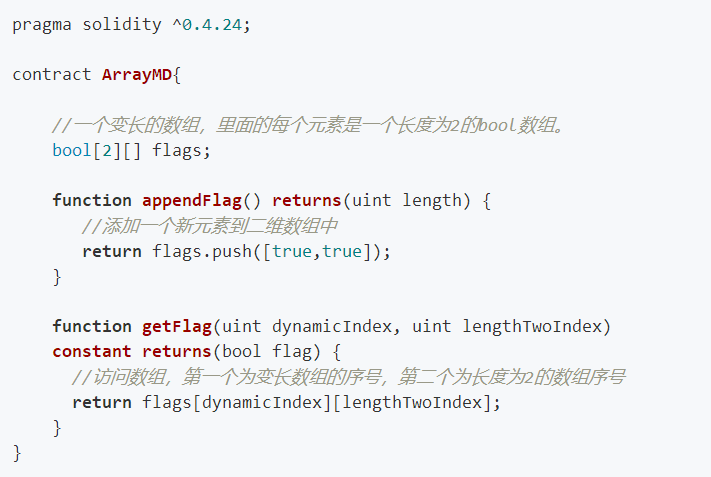
如果状态变量的类型为数组，也可以标记为public类型，从而让Solidity创建一个访问器。(public类型的状态变量都有默认的访问器)访问器对于外界使用者表现为一个函数, 因此可以通过调用函数的方式访问某些值。 另外在remix中表现为一个可以点击的按钮, 可以获取对应的public类型的值。



如上面的合约在Remix运行后，需要我们填入的是一个要访问序号的数字，来访问具体某个元素。

##### 5.2.6 多维数组

我们要创建一个长度为5的数组，每个元素又是一个变长uint数组，将被声明为uint[][5]。 反之, 假如要创建一个变长数组， 每个元素又是一个长度是5的数组, 将被声明为uint[5][]，比如下边的例子:



在上面的代码中，我们声明了一个二维数组，它是一个变长的数组，里面的每个元素是一个长度为2的数组。要访问这个数组flags，第一个下标为变长数组的序号，第二个下标为长度为2的数组序号。

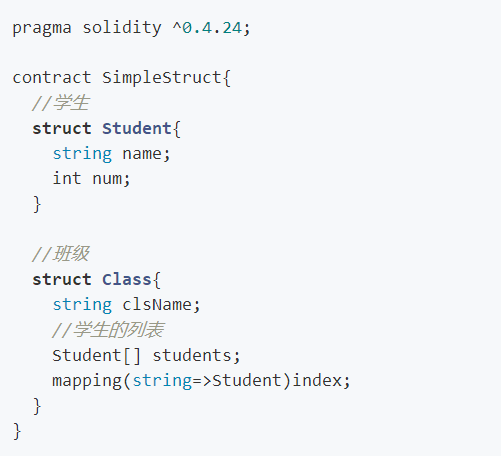
##### 5.2.7 bytes与string

bytes和string是一种特殊的数组。

由于bytes与string，可以自由转换，你可以将字符串s通过bytes(s)转为一个bytes。可以以这种方式获得字符串长度，以及获取字符中字符的UTF-8编码。

#### 5.3 引用类型——结构体

结构体，Solidity中的自定义类型。我们可以使用Solidity的关键字struct来进行自定义。结构体内可以包含字符串，整型等基本数据类型，以及数组，映射，结构体等复杂类型。数组，映射，结构体也支持自定义的结构体。我们来看一个自定义结构体的定义：

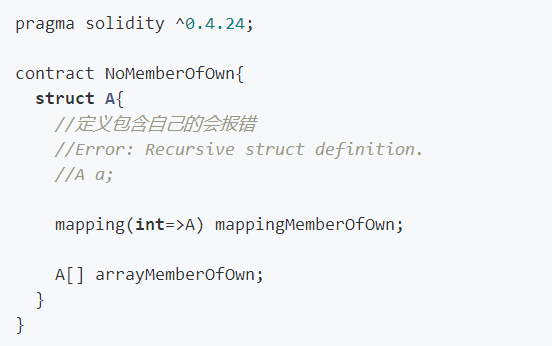


在上面的代码中，我们定义了一个简单的结构体Student，它包含一些基本的数据类型。另外我们还定义了一个稍微复杂一点的结构体Class，它包含了其它结构体Student，以及数组，映射等类型。

数组类型的students和映射类型的index的声明中还使用了结构体。

##### 5.3.1 结构体定义的限制

我们不能在结构中定义一个自己作为类型，这样限制原因是，自定义类型的大小不允许是无限的。我们来看看下述的代码：

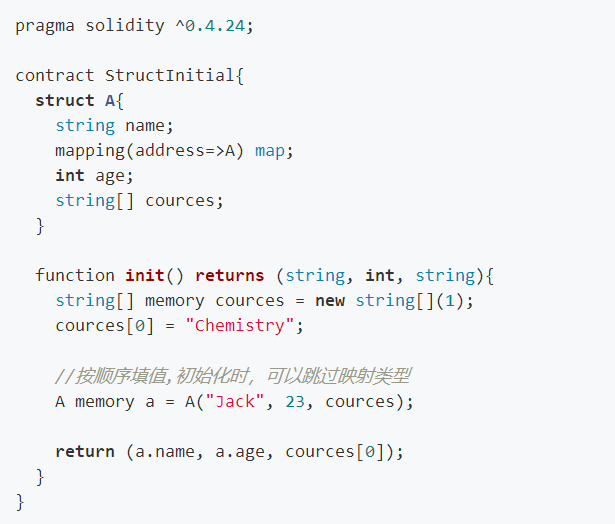


在上面的代码中，我们尝试在A类型中定义一个A a;，将会报错Error: Recursive struct definition.。虽然如此，但我们仍然能在类型内用数组，映射来引用当前定义的类型，如变量mappingMemberOfOwn，arrayMemberOfOwn所示。

##### 5.3.2 初始化

1 直接初始化

如果我们声明的自定义类型为A，我们可以使用A(变量1,变量2, ...)的方式来完成初始化。来看下面的代码：



上面的代码中，我们按定义依次填入值，即可完成了初始化。需要注意的是，参数要与定义的数量匹配。当你填的参数与预计初始化的参数不一致时，会提示Error: Wrong argument count for function call: 2 arguments given but expected 3. Members that have to be skipped in memory: map。另外，在初始化时，需要忽略映射类型，后面有具体说明。

2 命名初始化

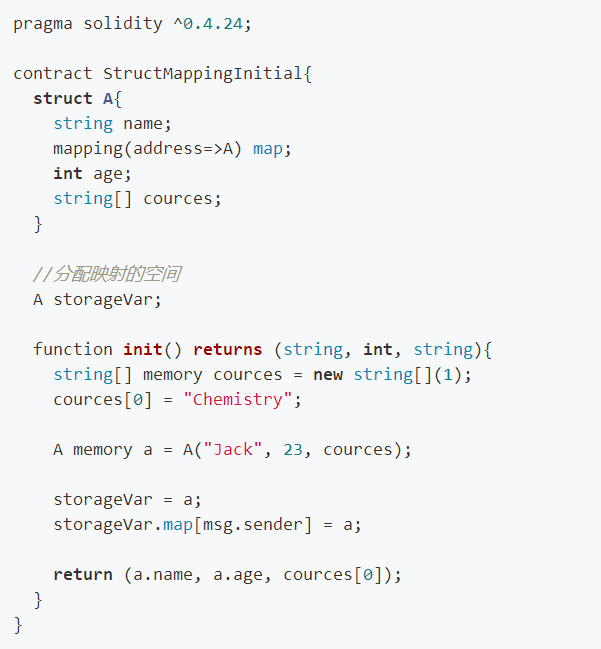
还可以使用类似JavaScript的命名参数初始化的方式，通过传入参数名和对应值的对象。这样做的好处在于可以不按定义的顺序传入值。我们来看看下面的例子：



上面的例子中，通过在参数对象中，指定键为对应的参数名，值为你想要初化的值，我们即完成了初始化。同样需要注意的是，参数要与定义的个数一致，否则会报类似这样的错误Error: Wrong argument count for function call: 2 arguments given but expected 3. Members that have to be skipped in memory: map。另外，在初始化时，需要忽略映射类型，后面有具体说明。

3 结构体中映射的初始化

由于映射是一种特殊的数据结构，所以你可能只能在storage变量中使用它。



上面的例子中，我们定义的了一个storage的状态变量storageVar，完成了映射类型的存储空间分配。然后我们就能对映射类型赋值了。

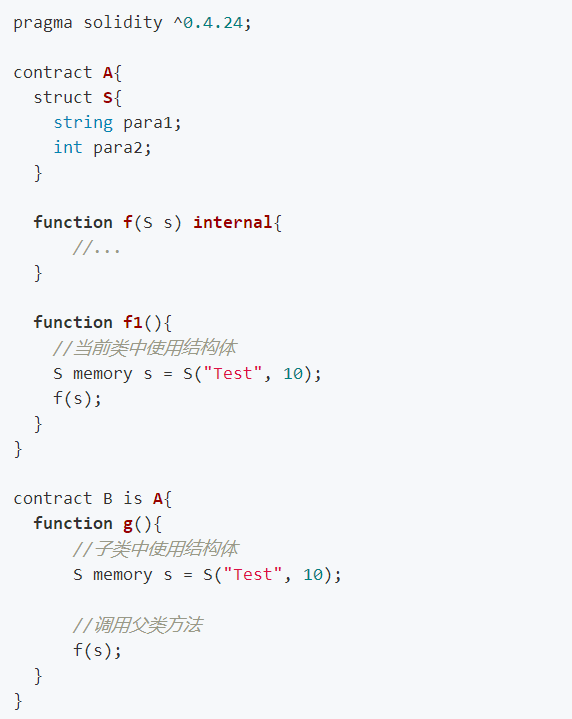
如果你尝试对memory的映射类型赋值，会报错Error: Member "map" is not available in struct StructMappingInitial.A memory outside of storage.。

##### 5.3.3 结构体的可见性

关于可见性，当前只支持internal的，后续不排除放开这个限制。

1 继承中使用

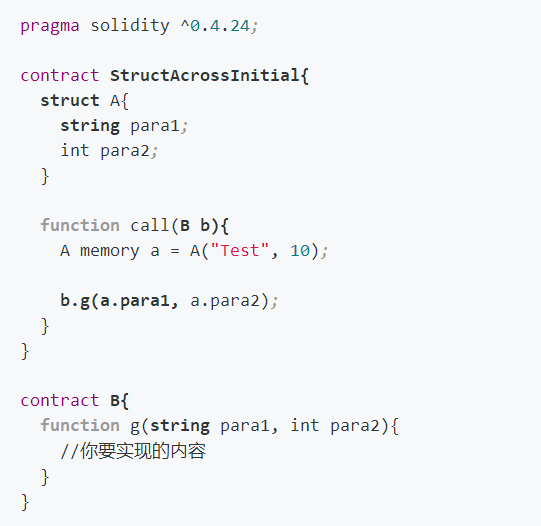
结构体由于是不对外可见的，所以你只可以在当前合约，或合约的子类中使用。包含自定义结构体的函数均需要声明为internal的。



在上面的代码中，我们声明了f(S s)，由于它包含了struct的S，所以不对外可见，需要标记为internal。你可以在当前类中使用它，如f1()所示，你还可以在子类中使用函数和结构体，如B合约的g()方法所示。

2 跨合约的临时解决方案

结构体，由于是动态内容。当前不支持在多个合约间互用，目前一种临时的方案如下：



在上面的例子中，我们手动将要返回的结构体拆解为基本类型进行了返回。

#### 5.4 内存变量的布局

Solidity预留了3个32字节大小的槽位(存储空间)：

▪ 0-64：哈希方法的暂存空间(scratch space)

▪ 64-96：当前已分配内存大小(也称空闲内存指针(free memory pointer))

注: 暂存空间是2个32字节大小，作为一个整体单元。

暂存空间可在语句之间使用（如在内联编译时使用）

Solidity总是在空闲内存指针所在位置创建一个新对象，且对应的内存永远不会被释放(也许未来会改变这种做法)。

有一些在Solidity中的操作需要超过64字节的临时空间，这样就会超过预留的暂存空间。他们就将会分配到空闲内存指针所在的地方，但由于他们自身的特点，生命周期相对较短，且指针本身不能更新，内存也许会，也许不会被清零(zerod out)。因此，大家不应该认为空闲的内存一定已经是清零(zeroed out)的。

#### 5.5. 状态变量的存储模型

大小固定的变量（除了映射，变长数组以外的所有类型）在存储(storage)中是依次连续从位置0开始排列的。如果多个变量占用的大小少于32字节，会尽可能的打包到单个storage槽位里，具体规则如下：

▪ 在storage槽中第一项是按低位对齐存储（lower-order aligned) 。

▪ 基本类型存储时仅占用其实际需要的字节。

▪ 如果基本类型不能放入某个槽位余下的空间，它将被放入下一个槽位。

▪ 结构体和数组总是使用一个全新的槽位，并占用整个槽(但在结构体内或数组内的每个项仍遵从上述规则)。

### 6、Solidity变量类型：其他

#### 6.1 变量类型——映射

映射或字典类型，一种键值对的映射关系存储结构。定义方式为mapping(\_KeyType => \_KeyValue)。键的类型允许除映射外的所有类型，如数组，合约，枚举，结构体。值的类型无限制。

映射可以被视作为一个哈希表，其中所有可能的键已被虚拟化的创建，被映射到一个默认值（二进制表示的零）。但在映射表中，我们并不存储键的数据，仅仅存储它的keccak256哈希值，用来查找值时使用。

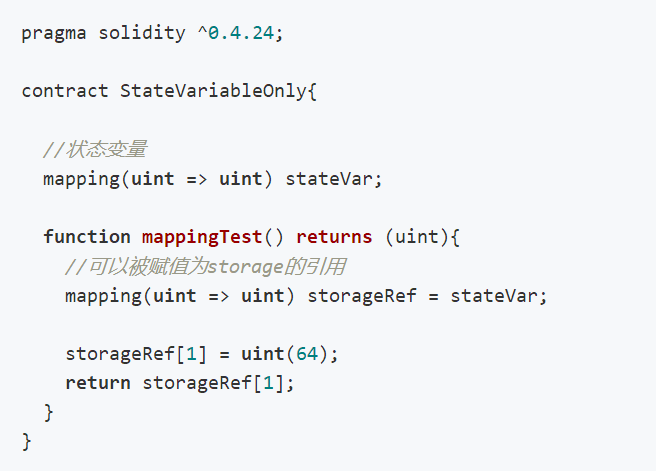
因此，映射并没有长度，键集合（或列表），值集合（或列表）这样的概念。

映射类型，仅能用来定义状态变量，或者是在内部函数中作为storage类型的引用。引用是指你可以声明一个，如var storage mappVal的用于存储状态变量的引用的对象，但你没办法使用非状态变量来初始化这个引用。

可以通过将映射标记为public，来让Solidity创建一个访问器。要想访问这样的映射，需要提供一个键值做为参数。如果映射的值类型也是映射，使用访问器访问时，要提供这个映射值所对应的键，不断重复这个过程。

1 只能是状态变量

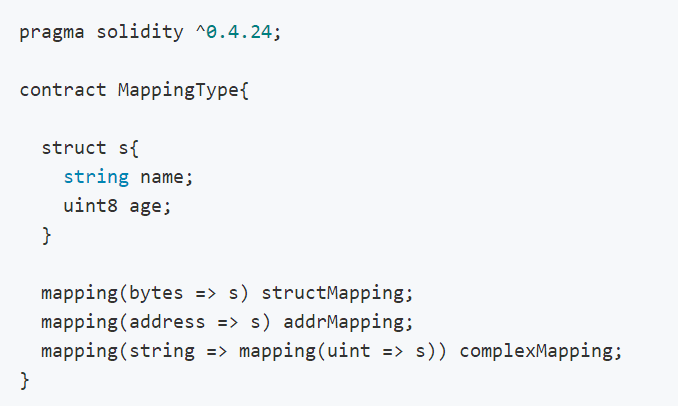
由于在映射中键的数量是任意的，导致映射的大小也是变长的。映射只能声明为storage的状态变量，或被赋值给一个storage的对象引用。我们来看下面的示例：



在上面的示例中，我们声明了storage的状态变量stateVar，可以对其增加新键值对；也能通过引用传递的方式赋值给storage的引用storageRef。

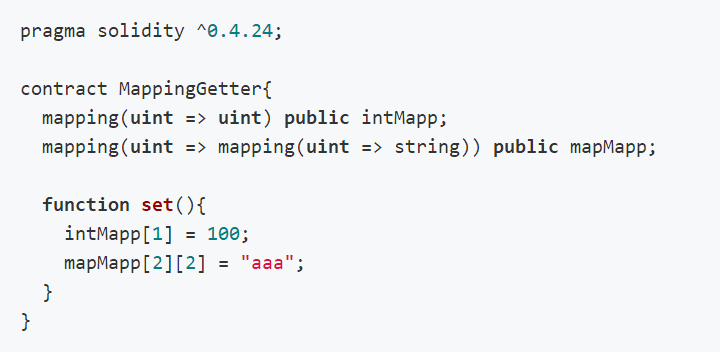
2 支持的类型

映射类型的键支持除映射，变长数组，合约，枚举，结构体以外的任意类型。值则允许任意类型，甚至是映射。下面是一个简单的例子代码：



3 setter方法

对于映射类型，也能标记为public。以让Solidity为我们自动生成访问器。



在上面的例子中，如果要访问intMapp第二个元素，在一对中括号中输入值1即intMapp[1]。而如果要访问嵌套的映射mapMapp[2][2]，则输入两个键对应的值2,2即可。

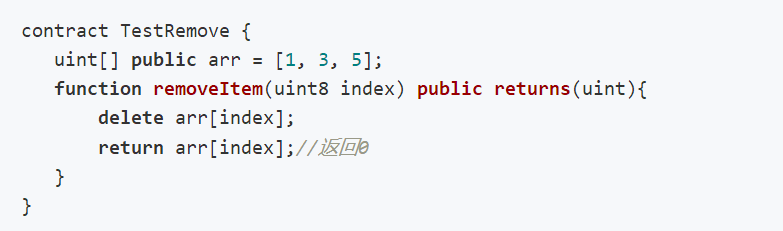
4 getter方法

可以通过将映射标记为public，来让Solidity创建一个访问器。

#### 6.2 变量类型——特殊的运算符delete

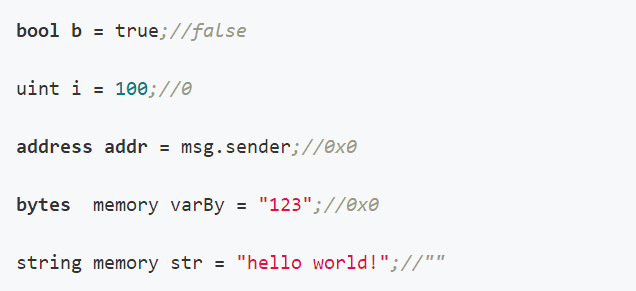
Solidity中有个特殊的操作符delete用于释放空间(特别是对于数组结构体以及映射类型的变量)，因为区块链做为一种公用资源，为避免大家滥用。且鼓励主动对空间的回收，释放空间将会返还一些gas给调用者。

delete关键字的作用是对某个类型值a赋予初始值。比如如果删除整数delete a等同于a = 0。



##### 6.2.1 删除基本类型

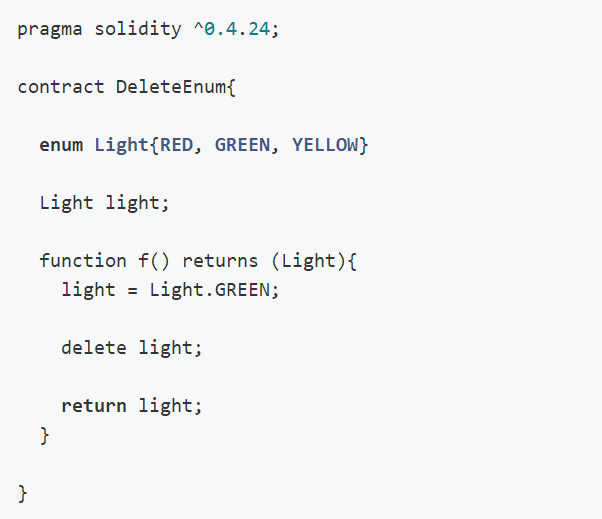
对于基本类型，使用delete会设置为对应的初始值：



删除bool类型是false，变长字节数组是0x0。string则是空串。

##### 6.2.2 删除枚举

删除枚举类型时，会将其值重置为序号为0的值。



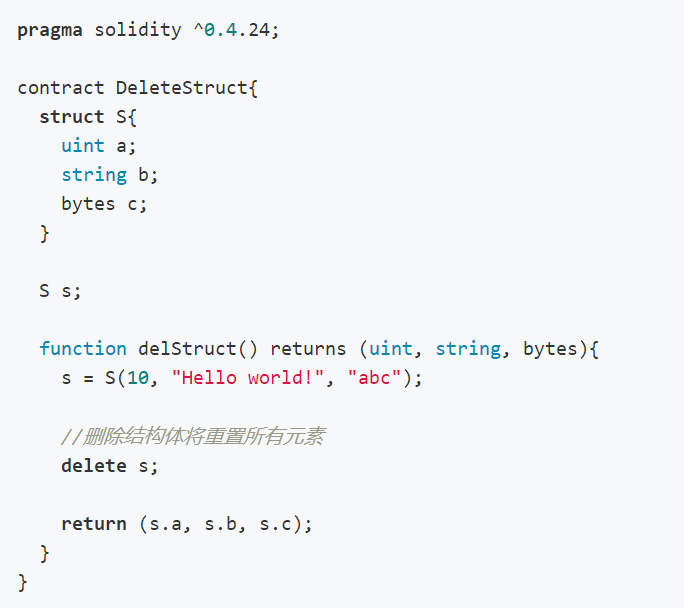
上面的例子中，删除light后，light将被置为序号为0的值即RED。

##### 6.2.3 删除函数

尝试了一下删除函数，会报错Error: Expression has to be an lvalue.，看来不能删除函数。

##### 6.2.4 删除结构体

删除一个结构体，会将其中的所有成员变量一一置为初值，我们来看一个例子。



在上面的例子中，我们声明了结构体s，调用delete s，结构体的值将变为其对应类型uint，string，bytes的初始值0，空串和0x0。

##### 6.2.5 删除映射

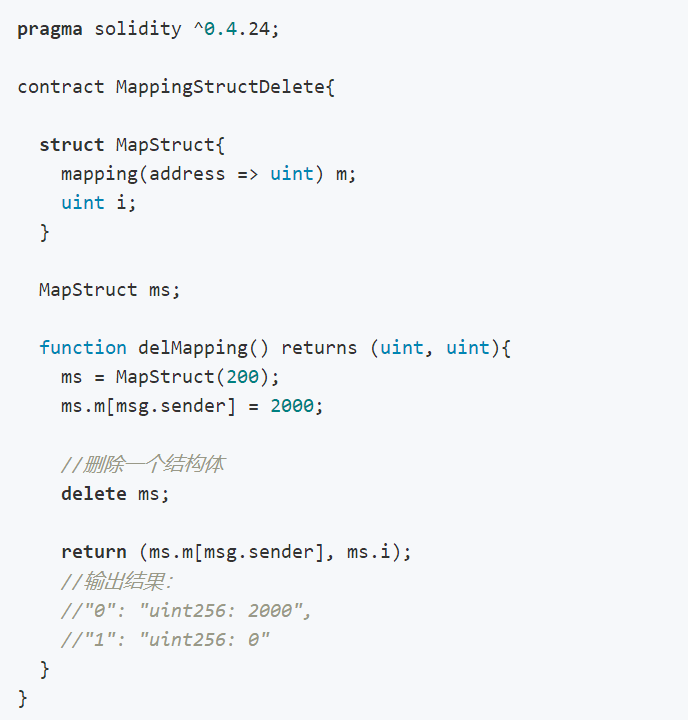
映射是一个特殊的存在，由于映射的键并不总是能有效遍历（数据结构没有提供接口，也并不总是需要关心所有键是什么），所存的键的数量往往是非常大的，所以我们并不能直接删除一个映射。

如果直接删除一个映射会报错Unary operator delete cannot be applied 但我们可以指定键来删除映射中的某一项：



##### 6.2.6 删除结构体中的映射

如果删除一个结构体时，其中含有映射类型，会跳过映射类型。我们来看一个删除含映射的结构体示例：

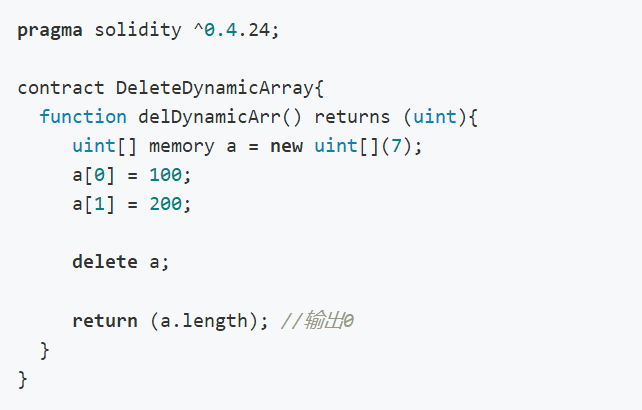


上面的示例中，删除结构体ms，并没有影响其中映射ms.m的值。

##### 6.2.7 删除数组

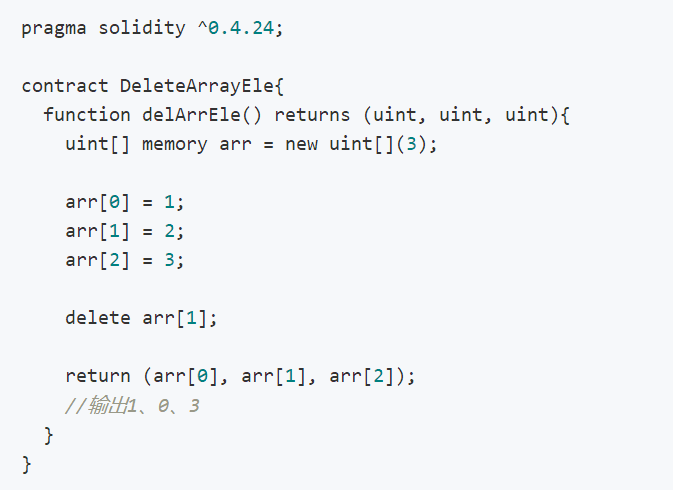
对于定长数组，删除时，是将数组内所有元素置为初值。

而对于变长数组时，则是将长度置为0。



##### 6.2.8 删除数组的一个元素

我们也可以删除数组的一个元素，有一点违反直觉的是，删除一个元素后，数组会留个空隙在那里。比如三个元素的数组，删除了第二个元素，只是将第二个元素置为了初始值，其它没变。



上述的代码运行后，将返回1，0，3。删除只是赋值，并没有移动元素。

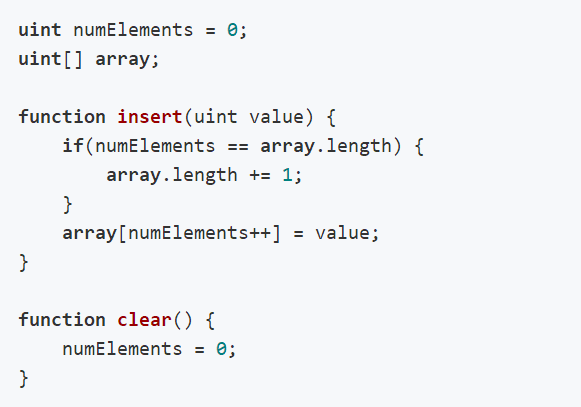
##### 6.2.9 gas使用的考虑

上文中，我们了解到，删除时会忽略映射，以及数组的某个元素被删除后，并不会自动整理数组。这些看起来很不符合常理，其实是基于对gas限制的考虑。因为如果映射或数组非常大的情况下，删除或维护它们将变得非常消耗gas。

不过，清理空间，可以获得gas的返还。但无特别意义的数组的整理和删除，只会消耗更多gas，需要在业务实现上进行权衡, 站在以太坊设计者的角度,因为不清理空间会浪费资源, 而大量遍历和删除操作又会占用cpu以及需要很多节点同步,因此节约空间和减少cpu的消耗都会奖励,两者需要找到一个平衡点才不至于资源浪费消耗太多gas。

##### 6.2.10 清理的最佳实践

由于本身并未提供对映射这样的大对象的清理，所以存储并遍历它们来进行清理，显得特别消耗gas。一种实践就是能复用就复用，一般不主动清理。下面是一个数组的插入实现，比如增加一个计数器，直接忽略已使用过的位置。

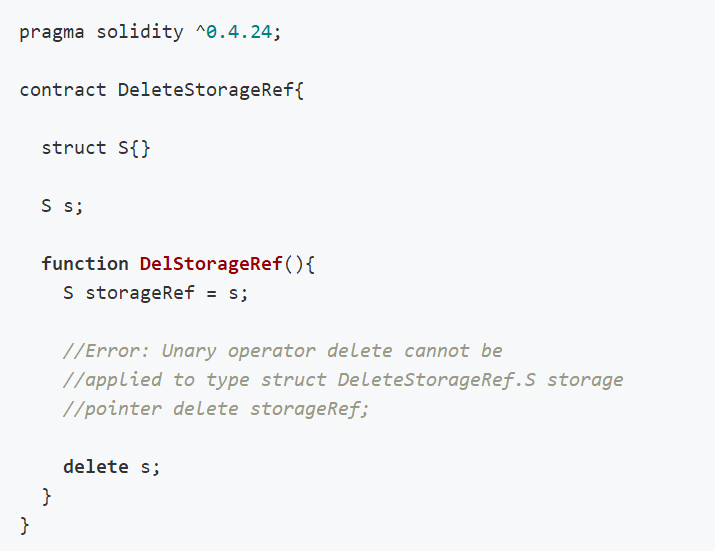


上面的例子中，我们在数组新增时，直接忽略掉已使用过的槽位。而在代码内，我们使用numElements来代替array.length，以获取当前数组所在的位置。

如果这种大对象是在某个事件发生时，一次性使用，然后需要回收的。一个更有效的方式是，在发生某个事件时，创建一个新合约，在新合约完成逻辑，完成后，让合约suicide。清理合约占用空间返还的gas就退还给了调用者，来节省主动遍历删除消耗的额外gas。

##### 6.2.11 删除的注意事项

删除本质是对一个变量赋初值。所以我们删除storage的引用时会报错，因为storage的引用并没有自己已分配的存储空间，所以不能对storage的引用直接赋初值。



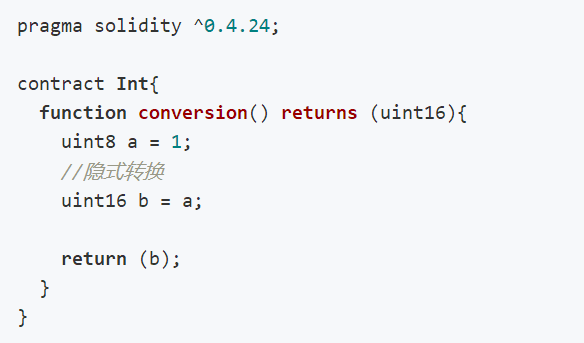
上面的例子中，删除storageRef会报错。

#### 6.3 变量类型——基本类型间的转换

##### 6.3.1 隐式转换

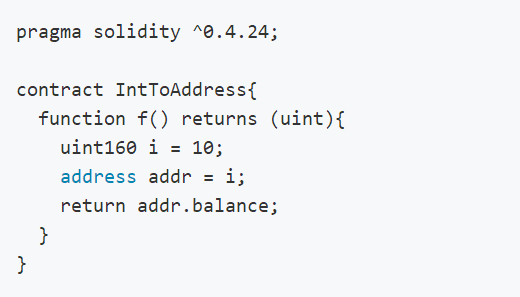
如果一个运算符能支持不同类型。编译器会隐式的尝试将一个操作数的类型，转为另一个操作数的类型，赋值同理。

一般来说，值类型间的互相转换只要不丢失信息，语义可通则可转换。下面，我们来看一个整数转换的例子：



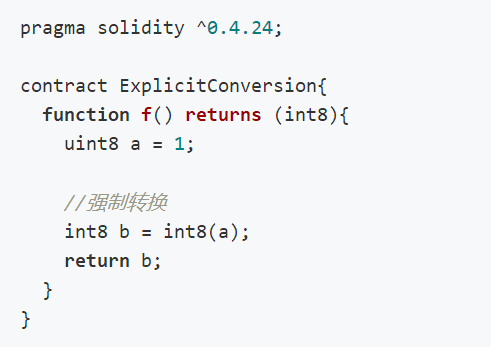
上面的例子中，我们将一个uint8的变量a隐式的转换为了uint16。同理它还支持转为uint32，uint128和uint256。

另外，无符号整数可以被转为同样，或更大的字节的类型。但需要注意的是，不能反过来转换。由于address是20字节大小，所以它与int160大小是一样。



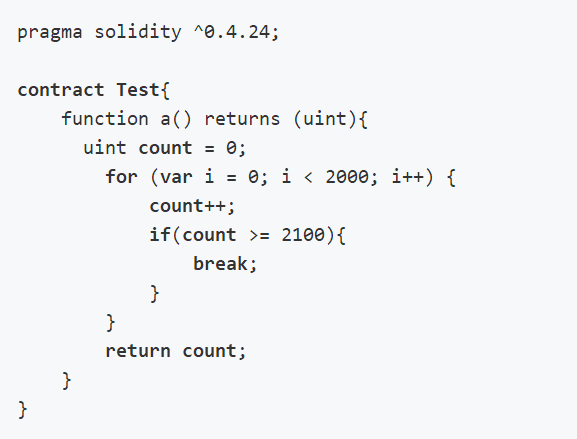
##### 6.3.2 显式转换

编译器不会将语法上不可转换的类型进行隐式转换，此时我们要通过显式转换的方式，比如将一个有符号整数，转为一个无符号整数。



##### 6.3.3 类型推断

有时为了方便，我们不会显式定义类型。但由于编译器，会自动挑选一个最恰当的类型，所以会常常留下坑，我们来看这个例子：



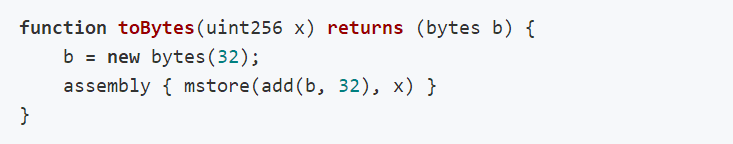
大家可以想想上述代码运行的结果。

上述代码运行的结果实际为2100。原因是因为var i = 0定义时，通过类型推断，i的实际类型为uint8，所以它会一直循环，如果没有count >= 2100这个判断语句，这个循环将永远不会结束。

##### 6.3.4 一些常见的转换方案

1 uint转为bytes

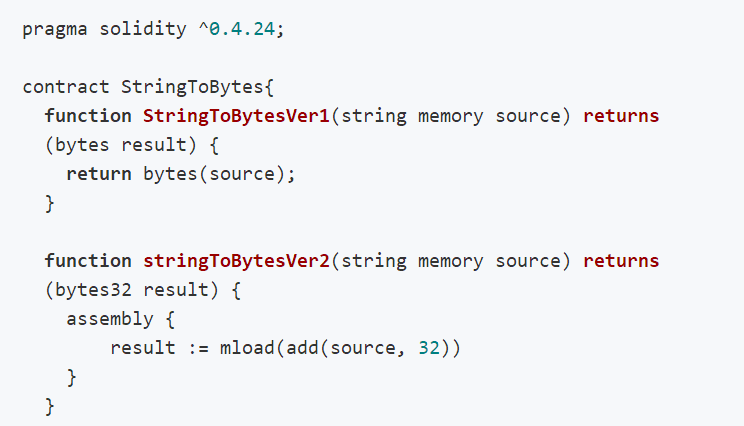
assembly是可以用汇编的方式实现某功能。 将一个uint转换成bytes，可以使用assembly。



上面的转换方式可能是效率最高的方式。

2 string转为bytes

string可以显式的转为bytes。但如果要转为bytes32，可能只能使用assembly。



### 7、 控制结构

### 8、浏览器编辑器Remix

## 2.4、基于ERC20代币发行

##### 2.4.1、 ERC20介绍

##### 2.4.2、 基于ERC20的智能合约开发

##### 2.4.3、 部署智能合约并验证

##### 2.4.4、 开发代币的DApp