1

​

区块链技术与应用课程设计

​

**题目：基于区块链的高校学生成绩管理系统**

**学院：区块链学院**

**专业：软件工程（区块链方向）**

**姓名：曾尉斌**

**学号：2019000101091**

**指导老师：江海**

**完成时间：2020年1月8日**

​

摘要

​

**中共中央政治局就区块链技术发展现状和趋势进行第十八次集体学习。中共中央总书记习近平在主持学习时强调，区块链技术的集成应用在新的技术革新和产业变革中起着重要作用。我们要把区块链作为核心技术自主创新的重要突破口，明确主攻方向，加大投入力度，着力攻克一批关键核心技术，加快推动区块链技术和产业创新发展。**[1]

**关键词：区块链技术；创新**

**目录**

**摘要**

**第一节 需求分析**

**1.1社会痛点**

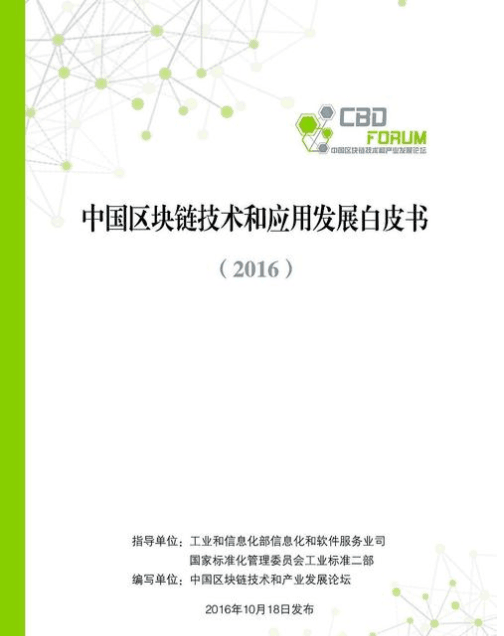
2019年2月8日，翟天临因在直播中回答网友提问时，不知知网为何物，导致他的北大博士学位真实性受到质疑，并由此引发了一场全社会对学术造假的深刻反思。教育作为社会阶层流动的重要通道，其公平性关乎到的不仅仅是个人的荣誉，还有社会的公正，从古代科举到现在的各种考试，无不备受重视。

近些年来，我们的教育在规模和质量上取得了突飞猛进的成果，但是在“开放”和“公信”方面却依然备受质疑。其实，开放是教育的基本理念，知识的公共性决定了教育的开放性。教育的开放意味着教育服务主体的多样化，打通各个教育机构之间的壁垒，突破传统的专业限制和学习时段限制，构建跨校和跨地区的学习环境也是教育开放的另一个重要趋势。

进一步来看，教育的开放是全过程，不仅包括教育资源的开放，也包括教育行为记录、教育评价结果的开放。随着现代远程教育的兴起和大规模开放在线课程（慕课）的推广，开放教育已经从趋势变为了共识。开放教育的形式丰富多样，突破了面授的单一形式。但是，现有的教育系统尚未很好地适应这种模式，面授以外的学习过程和学习结果往往不被公众认可，从而产生了信任危机。即使是在传统的高等教育领域，学生的学历信用记录体系不完整、不透明，导致政府或者企业无法获得完整的有效信息。在求职时，又存在学历造假、简历造假等问题，用人单位和相关院校缺乏简单高效的验证手段。为此，我们急需一种新的机制，来保障人们在享受教育开放带来的便利的同时，保障教育应有的公信力，并进一步地推动教育走向开放。

**1.2探索解决方式**

在此背景下，区块链技术的独特性也逐渐成为教育领域关注的焦点。2016年 10月，工信部颁布《中国区块链技术和应用发展白皮书》，指出“区块链系统的透明化、数据不可篡改等特征，完全适用于学生征信管理、升学就业、学术、资质证明、产学合作等方面，对教育就业的健康发展具有重要的价值”。2018年颁布的《教育信息化2.0行动计划》更是明确指出加快面向下一代网络的高校智能学习体系建设，探索区块链、大数据等新技术在学习效果记录、转移、交换、认证等方面的有效方式，形成泛在化、智能化学习体系。



区块链是一种把区块以链的方式组合在一起的数据结构，具有去中心化、按时序记录数据、集体维护、可编程和安全可信等特点，可以为信用背书、信息加密、智能合约等提供极大的便利，这对于当前的教育领域有很大的应用潜力。

首先，通过开发、应用基因区块链技术的学习管理平台，学习者和教师将成为教学资源和信息的管理和控制者，而不再是学校。因为基于区块链技术的分布式账本能够安全、灵活地管理分布式教学资源和信息，并能够通过数据分析技术的应用，在实现大规模学习认证的同时，扩大学习者的受教育机会。例如阿里云云学院就充分利用互联网及区块链的特点与优势，为学习用户提供云计算、大数据等专业的体系化在线课程与实验环境。进阶式的课程内容、闯关式的节点测试与在线认证相结合，培养兼备理论与实操双能力的技术人才。阿里云云学院为毕业学员颁发阿里云官方认证证书，出具技能评估报告，并给企业推荐。

其次，对那些因工作或其他原因而无法完成传统学校教育的个人学习者来说，通过区块链技术平台，可以把从不同教育机构修来的学分或学习结果绑定、组合在一起，申请认可此学习模式的教育机构的认证，而获得相应毕业或学位证书。英国开放大学已积极实践这一基于区块链技术的新型学习模式，此外，英国开放大学的“知识与媒体研究中心”已开发出组合“微认证”，或者说徽章的创新技术，以适应基于区块链技术平台的学习和认证。

最后，区块链技术的发展和应用能够帮助更多的人取得学习证明、职业资格证书或学位证书，尤其是不发达国家公民，从而有效提高个人在劳动力市场上的竞争力、促进其职业发展，并进而促进创业。例如江西软件职业技术大学为新生颁发基于区块链的录取通知书，该证书具备不可篡改性和与可验证特性。

**第二节 系统分析**

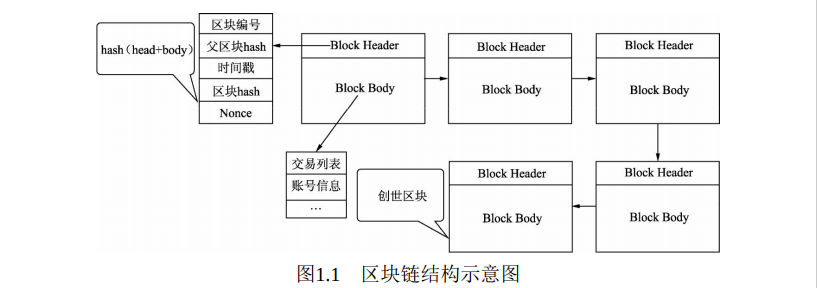
**2.1名词须知**

**区块链**：

2008年比特币诞生，原本只是一个密码学极客之间的玩物，没想到犹如打开的潘多拉盒子，慢慢席卷全球。在08年以前还没有人能成功地研发出一个运行良好的数字货币出来，直到比特币问世；另外，区块链作为比特币的底层技术，在此之前也是闻所未闻。那么区块链到底有什么魔力，让整个世界为之疯狂呢？区块链的本质是什么？是去中心化，还是分布式账本？是非对称加密，还是点对点的信任机制？我们认为都不是，这些都只是手段，不是目的，更不是本质。区块链是使用分布式的账本来建立基于计算机算法的、去中心化的人工市场。它被誉为下一代的人工智能。人工智能是企图了解个体智能的实质，并生产出一种新的能以人类智能相似的方式做出反应的智能机器。而区块链试图掌握市场智能的实质，并创造出一种新的能以市场智能相似的方式做出反应的人工市场。在数字化时代，它将是交易成本最低的基础设施：让计算机能够从事一些过去只有人脑才能完成的交易工作，从而解脱人的一部分繁重的脑力劳动。以AlphaGo为代表的人工智能侧重于发展生产力，而以区块链为代表的人工智能侧重于改造生产关系，形成更聪明的数字化企业，构建更公正和更高效的数字化市场。

因此，从狭义来说，区块链是所有交易的公共总账，由按照时间顺序记录了交易的数据区块的链条所组成。从广义来说，区块链是基于计算机算法的人工市场，是协调所有交易、实现共赢的中枢神经系统。在未来，所有的交易，包括商品交易、雇佣交易、和金融交易都将在区块链上进行，而不仅仅是数字货币的发行和支付，这将更好地促进经济的转型和升级，从而造就更聪明和强大的企业，形成更公正和更高效的市场。

区块链是一个数据集，这个数据集的组织方式比较特殊，它 把数据打包成一个一个的区块，每一个区块是一个区块链数据集的子 集。区块和区块之间的数据相互关联形成逻辑上的链式结构，所以区块 链因此得名，1.1.2节中我们会具体介绍这种链式结构。 区块链技术源于比特币，最开始就是为了解决去中心化的问题，所 以区块链是分布式的，可以把区块链看作一个分布式的账本。区块链上 的数据是每一个参与者的机器上都拥有的，而不是只放在某一台中心机 器上。 使用区块链来做什么呢？其实有太多场景。例如，使用区块链完成 类似于银行的转账职能，具体会在后面介绍。现在我们关心的是区块链 转账的数据如何传递给其他参与者，如果让每一个参与者都拥有区块链 数据，就需要每一个参与者之间能相互通信，因此需考虑怎样组建网 络。 假设已经组建好了网络，每一个参与者之间都能相互通信了，那么 另一个问题来了，每一个参与者都有全部的区块链数据，怎样保证大家 的数据都是一致的，或者说区块链的数据怎样不被篡改呢？ 简单地来说，区块链解决的主要问题就是：分布式网络中每一个参 与者的数据交换和防止数据不被参与者篡改，获得每一个参与者的认同。

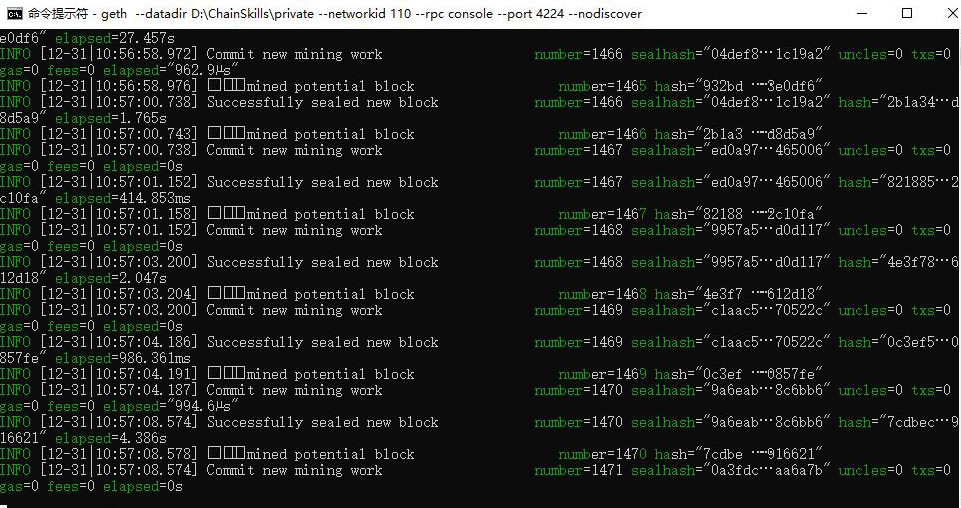


每一个区块（Block）中包含一个区块头（Block Header），区块头 中包含一些块的元数据信息，例如区块编号、区块Hash值、区块创建时 间、父区块的Hash值和Nonce等数据。其中，Nonce是用来做工作量证 明的，它对于使用工作量证明机制的区块链来说很重要，所以后面会反 复介绍。 每一个区块中还包含一个区块体（Block Body），区块体中包含数 据交易信息和账户信息等，不同的区块链有不同的实现。 每一个Block会计算一个Hash值，这个Hash值会被下一个Block包 含，这样前一个Block就被固定了，所以只需要简单地计算前一个Block 的Hash值和自己包含的前一个Block的Hash值是否一致，这样就能判断 前一个Block是否已被修改，每一个Block都确定了前一个Block就形成了 一个链式的结构。

区块链结构时提到区块链除了交易数据，还包含一些其 他的数据，可以称这些数据为Block的元数据（Meta Data）。这些元数 据如下所述。 ·Index：第几区块（创世区块的索引为0）。 ·Hash：当前区块的Hash值。 ·Previous Hash：上一个区块的Hash值。 ·Timestamp：当前区块创建时的时间戳。 ·Data：存储在当前区块上的交易信息。 ·Nonce：参与Hash运算的数值，使区块的Hash值满足指定条件。

**以太坊**：以太坊是一个任何人都可以创建和运行去中心化应用的区块链平 台。但是没有任何人可以控制或者拥有以太坊，它是由很多世界各地的 人建立的开放源码项目。 注意：我们上面提到的以太坊特指以太坊整个生态，而不是指以太币，请读者不要混淆。 以太坊作为和比特币对标的项目，它的设计更具有适应性和灵活 性，可以容易地创建应用在以太坊上运行。以太坊和许多平台一样，例 如Android平台，遵循Android的规范，使用Java或者Kotlin语言就可以开 发了，以太坊则可以使用Solidity或者其他语言就可以进行开发。 应用开发完毕后要发布。像Android这类平台上的应用，开发完成 一般需要发布到第三方平台上，然后用户才能下载、安装，这些平台往 往有各种限制。以太坊不一样，以太坊开发完成之后可以直接部署在以 太坊平台上，没有任何限制。其他人要使用这些开发的应用也不需下 载，只需要调用开发者提供的接口或者开发者建立在接口上的服务就可 以了。另外，如果开发者要使用其他人开发的以太坊应用，还可以看到 其他人的源码。 当然，以太坊像比特币一样，也提供了自己的数字货币——以太 币，可以进行收款、付款、转账等各种交易。

挖矿只是一个比喻，对比于稀缺贵重金属需要实际挖矿而言，数字 货币运算的挖矿指的是一种计算。以太坊中挖矿的单位一般是区块，我 们一般说的矿工挖出一个区块，就是指一个运行以太坊客户端的节点从 现有的交易池中选择出一些交易，记录在一个块中，并且计算出一个指 定的值。 完成了上面的过程会有一些奖励，就是以太币，这就算是挖到矿 了，以太坊中称为挖出一个区块。总结一下，挖矿就是通过在区块链中 创建、验证、发布和传播区块来保护网络的一种方式。 至于为什么能够通过挖矿来保护网络，可以回顾一下第1章介绍的 工作量证明机制的相关内容。



挖矿奖励 前一节我们提到了成功挖出一个区块的矿工会有一些奖励，那么这 个奖励包含哪些部分呢？ ·静态区块奖励，5个以太币。 ·区块中所有交易gas花费，与当前gasPrice相关。 ·包含叔块的额外奖励。 其中，静态区块奖励是会逐渐减少的，最开始是5个以太币，现在 是3个以太币，区块中的交易使用gas并不是以太币，所以会乘以交易设 置的gasPrice。

**以太坊开发的四个阶段：**

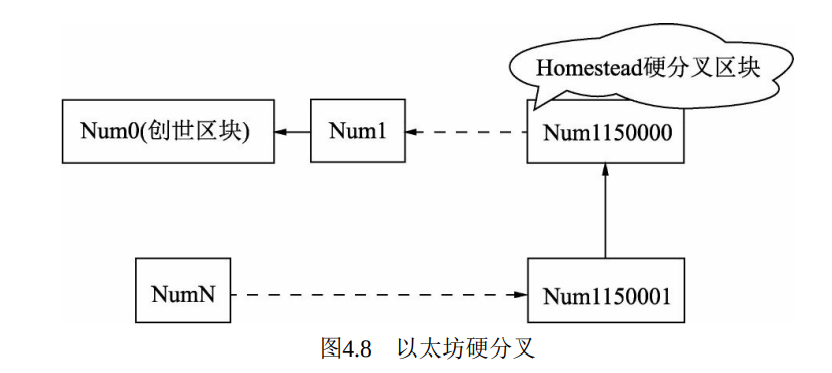
以太坊的诞生是第一阶段的启动，名为“前沿（Frontier）”。以太坊的发展计划分四个阶段进行，每个新阶段都会发生重大变化。每个阶段都可能包含子版本，称为“硬分叉”，它们以不向后兼容的方式改变功能。

四个主要的发展阶段代号为前沿（Frontier），家园（Homestead），大都会（Metropolis）和宁静（Serenity）。中间的硬分叉代号为“冰河时代（Ice Age）”，“DAO”，“蜜桔前哨（Tangerine Whistle）”，“假龙（Spurious Dragon）”，“拜占庭（Byzantium）”和“君士坦丁堡（Constantinople）”。它们在下面列出，以及硬分叉发生的块号

以太坊的GHOST协议是一个非常重要的协议，它需要结合前面的 区块链、挖矿、工作量证明机制的知识才能更好地理解。如果能够理解 GHOST，那么对于以太坊的理解基本能够加深一个层次。正是因为 GHOST比较重要，所以本节我们尽量使用比较简化的模式来梳理一下 GHOST及其相关的知识。

我们使用网银转账，需要确认银行是否转账成功。有的银行需要30 秒就能确认，有的银行需要2小时才能确认。以太坊从根本来说是为了 完成交易，因此也需要确认一笔交易是否成功，我们一般希望交易在一 个比较确定的范围内被确认。 因为交易会被打包在区块之中，所以控制区块产生的时间在一定程 度上可以控制交易被确认的时间。区块产生的时间如何控制呢？我们知 道工作量证明机制中有一个难度值difficulty，这个难度值可以控制区块 产生的时间。中做了 一些调整，难度计算公式如下： block\_diff = parent\_diff + parent\_diff // 2048 \* max(1 - (block\_timestamp - parent\_timestamp) // 10, -99) + int(2\*\*((block.number // 100000) - 2)) 其中，//表示整除。以太坊通过调整区块难度把区块产生时间控制 在15秒左右，所以我们可以认为以太坊的区块时间是15秒，当然这不是 一个绝对的时间，因为工作量证明机制使用的时间是不确定的。 一个区块时间不好控制，但是多个区块的平均时间就好控制了。以 太坊会把30 000区块当作一个纪元，只要把一个纪元的时间控制在125 小时左右就可以了。

区块分叉分为3种，分别是普通分叉、软分叉和硬分叉。其中软分 叉和硬分叉是因为修改源码造成的。如果修改的源码，只要求以太坊网 络中的50%以上的算力升级到新的版本，那么这个分叉叫做软分叉。如 果修改了源码，要求以太坊网络中的所有节点都必须升级到新的版本， 那么这个分叉叫做硬分叉。 通过以太坊的开发路线，知道以太坊的当前版本是Homestead， Homestead就是一个要求硬分叉的版本。应当注意分叉针对的是区块链数据，例如，Homestead版本的主网 要求大于等于1150000的区块号的产生必须是由Homestead版本产生。如 图4.8所示为以太坊区块链硬分叉示意图，就是把区块链数据从区块号 为1150000的区块分成了两部分。



**DAPP**:

去中心化应用 (DApps)

点对点（P2P，peer-to-peer）运动使数百万互联网用户能够连接在一起。USENET是被称为第一个点对点架构的一种分布式消息传递系统，它于1979年成立，是第一个“互联网”ARPANET的继承者。ARPANET是一个客户端 - 服务器网络，参与者运行节点请求和提供内容，但由于除了简单的基于地址的路由，缺乏提供任何上下文的能力，USENET很有希望实施一个分散的控制模型，即客户端 - 服务器模型，分别从用户或客户角度为新闻组服务器提供自组织方法。

1999年，著名的音乐和文件共享应用程序Napster出现了。Napster是点对点网络运动演变为BitTorrent的开始，参与用户建立了一个虚拟网络，完全独立于物理网络，无需遵守任何管理机构或限制。

由于点对点机制可用于访问任何类型的分布式资源，因此它们在去中心化应用程序中起着核心作用。

什么是DApp？

与传统应用程序不同，去中心化应用（DApp）不仅属于单个提供者或服务器，而是整个栈将在P2P网络上以分布式方式部署和操作。

典型的DApp栈包括前端，后端和数据存储。创建DApp有许多优点，典型集中式架构无法提供：

1）弹性：在智能合约上编写业务逻辑意味着DApp后端将在区块链上完全分发和管理。与在中央服务器上部署应用程序不同，DApp不会有停机时间，只要区块链仍在运行，它就会继续存在。

2）透明性：DApp的开源特性允许任何人分叉代码并在区块链上运行相同的应用程序。同样，任何与区块链的互动都将永久存储，任何拥有区块链副本的人都可以获得对它的访问权限。值得注意的是，可能无法将字节码反编译为源码并完全理解合约的代码。寻求提供合约行为完全透明的开发人员必须发布供用户阅读，编译和验证的源代码。

3）抗审查：只要用户可以访问以太坊节点，用户将始终能够与DApp交互而不受集中机构控制的干扰。一旦在网络上部署代码，任何服务提供商，甚至智能合约的所有者都不能更改代码。

**DApp的组件**

**区块链（智能合约）**

智能合约用于存储去中心化应用程序的业务逻辑，状态和计算; 将智能合约视为常规应用程序中的服务器端组件。

在以太坊智能合约上部署服务器端逻辑的一个优点是，你可以构建一个更复杂的架构，智能合约可以在其中相互读取和写入数据。部署智能合约后，未来许多其他开发人员都可以使用你的业务逻辑，而无需你管理和维护代码。

将智能合约作为核心业务逻辑功能运行的一个主要问题是在部署代码后无法更改代码。此外，一个非常庞大的智能合约可能需要耗费大量gas来部署和运行。因此，某些应用程序可能会选择离线计算和外部数据源。请记住，DApp的核心业务逻辑依赖于外部数据或服务器意味着你的用户必须信任这些外部事物。

**前端（Web用户界面(UI)）**

与DApp的业务逻辑需要开发人员了解EVM和新语言（如Solidity）不同，DApp的客户端界面使用基本的Web前端技术（HTML，CSS，JavaScript）。这允许传统的Web开发人员使用他们熟悉的工具，库和框架。与DApp的交互（例如签名消息，发送交易和密钥管理）通常通过浏览器本身使用Mist浏览器或Metamask浏览器扩展等工具进行。

虽然也可以创建移动DApp，但由于缺少可用作具有密钥管理功能的轻客户端的移动客户端，目前没有创建移动DApp前端的最佳实践。

**数据存储**

由于gas成本高，智能合约目前不适合存储大量数据。因此，大多数DApps将利用去中心化存储（如IPFS或Swarm）来存储和分发大型静态资产，如图像，视频和客户端应用程序（HTML，CSS，JavaScript）。

内容的哈希值通常使用键值映射存储为智能合约中的字节。然后，通过你的前端应用程序调用智能合约检索资产，以获取每个资产的URL。

Dapp是开放源代码能够运行在分布式网 络上，通过网络中不同对等节点相互通信进行去中心化操作的应用。 首先，DApp开放源代码，这样才能获得人的信任。例如比特币， 尽管很多人没有读过比特币的源码，但是仍然不影响这些人相信并且持 有比特币，就是因为比特币是开源的，如果有问题肯定会被发现。 去中心化应用肯定是分布式的应用，因为没有中心节点，所以必须 能够在节点之间进行通信。每一个去中心化应用都有自己的通信协议， 使用相同协议的节点共同组成了一个去中心化应用的网络。 注意：网络中的节点都是对等节点（peer），没有能完全控制 整个网络的节点。

DApp优点之一就是保证用户的匿名性，因为各种因素的限制，很 多应用必须经过身份验证才能使用。在中心化的应用中，可以通过要求 用户上传指定文件，如运营许可证等来验证用户的身份和资质。但是对 于去中心化的DApp来说，身份验证是一个比较有挑战性的问题，现阶 段使用的最广泛的方式就是数字证书。 其实最重要的就是私钥，谁拥有私钥，谁就是这个账户的拥有者， 比特币是这样，以太坊也是这样。这种方式就不能避免其他人“碰撞”私 钥，虽然碰撞到的几率很低，但还是存在这种可能性。DApp还没有尝 试次数限制或者短信验证的用户验证机制来避免这个问题。

比特币算知名度最高的DApp，是一个最早的数字货币，也是纯粹 的数字货币。比特币使用的很多技术也基本上成了其他DApp的“标 配”，例如区块链技术和工作量证明机制等。 还有一个大家比较关心的问题，就是比特币的合法性。首先需要明 确的一点是，比特币是合法的，问题的关键在于比特币被定义为什么 类，是一种货币，还是一种商品，亦或是其他分类。 国内比特币被定义为虚拟商品，不能作为货币流通，但是比特币交 易作为一种互联网上的商品买卖行为，普通民众在自担风险的前提下拥 有参与的自由。最重要的是金融机构和支付机构不得以比特币为产品或 服务定价。所以比特币是合法的，在国内只是限制了金融机构和支付机 构把比特币作为投资标的，在美国比特币被当作一种大宗商品，在英国 和欧盟比特币被当作一种货币。

以太坊也是热门的DApp，和比特币不同的是，以太坊的目标是做 一个DApp应用平台，而不仅仅是一个数字货币。最显著的特点就是以 太坊支持智能合约，这就为以太坊在很多应用场景的落地提供了夯实的 基础，例如在众筹、慈善、中介和公证等很多方面智能合约都能够发挥 它的作用。

DApp存在的问题 从辩证主义的角度来看，一件事有好的一面，肯定就会有不好的一 面，DApp也一样存在不完美的一面。我们知道DApp节点分布在网络 中，因此DApp的数据不会被轻易修改，辩证地看，也就让DApp升级困 难。 DApp保证了用户的匿名性，那么通常带来的问题就是用户身份的 验证困难，同样，因为DApp的安全性高，就存在DApp系统更加复杂的 问题。因为DApp的去中心化，所以DApp不能依赖以中心化的服务，那 么生态的建立是非常缓慢的。

**DApp框架**

有许多不同的开发框架和库，以多种语言编写，使得开发人员可以在创建和部署DApp时获得更好的体验。

**Truffle**

Truffle是一种流行的选择，为以太坊提供可管理的开发环境，测试框架和资产管道。

有了Truffle，你会得到：

* 内置智能合约编译，链接，部署和二进制管理。
* 与Mocha和Chai进行自动合约测试。
* 可配置的构建管道，支持自定义构建过程。
* 可编写脚本的部署和迁移框架。
* 用于部署到许多公共和专用网络的网络管理。
* 直接与合约沟通的交互式控制台。
* 在开发过程中即时重建资产。
* 在Truffle环境中执行脚本的外部脚本运行器。

**图灵完备的含义：**

图灵证明，你无法通过在计算机上模拟程序来预测程序是否会终止。简而言之，我们无法预测程序的运行路径。图灵完备系统可以在“无限循环”中运行，这是一个用于描述不终止程序的术语（过分简化地说）。创建一个运行永不结束的循环的程序是微不足道的。但由于起始条件和代码之间存在复杂的相互作用，无意识的无限循环可能会在没有警告的情况下产生。在以太坊中，这提出了一个挑战：每个参与节点（客户端）必须验证每个交易，运行它所调用的任何智能合约。但正如图灵证明的那样，以太坊在没有实际运行（可能永远运行）时，无法预测智能合约是否会终止，或者运行多久。可以意外，或有意地，创建智能合约，使其在节点尝试验证它时永久运行，实际上是拒绝服务攻击。当然，在需要毫秒验证的程序和永远运行的程序之间，存在无限范围的令人讨厌的资源浪费，内存膨胀，CPU过热程序，这些程序只会浪费资源。在世界计算机中，滥用资源的程序会滥用世界资源。如果以太坊无法预测资源使用情况，以太坊如何限制智能合约使用的资源？

为了应对这一挑战，以太坊引入了称为 *燃气* *gas*的计量机制。随着EVM执行智能合约，它会仔细考虑每条指令（计算，数据访问等）。每条指令都有一个以燃气为单位的预定成本。当交易触发智能合约的执行时，它必须包含一定量的燃气，用以设定运行智能合约可消耗的计算上限。如果计算所消耗的燃气量超过交易中可用的天然气量，则EVM将终止执行。Gas是以太坊用于允许图灵完备计算的机制，同时限制任何程序可以使用的资源。

2015年，攻击者利用了一个成本远低于应有成本的EVM指令。这允许攻击者创建使用大量内存的交易，并花几分钟时间进行验证。为了解决这一攻击，以太坊必须在不向前兼容（硬分叉）的更改中改变特定指令的燃气核算公式。但是，即使有这种变化，以太坊客户端也不得不跳过验证这些交易或浪费数周的时间来验证这些交易。

**Tokens**

什么是Token?

单词*Token*来源于古英语“tacen”，意思是符号或符号。常用来表示私人发行的类似硬币的物品，价值不大，例如交通Token，洗衣Token，游乐场Token。

如今，基于区块链的Token将这个词重新定义为基于区块链的抽象概念，可以被拥有，并代表资产，货币或访问权。

“Token”一词与微不足道的价值之间的联系与物理Token的使用限制有很大关系。通常仅限于特定的企业，组织或地点，物理Token不易交换，不能用于多个功能。通过区块链标记，这些限制被删除。这些Token中的许多Token在全球范围内有多种用途，可以在全球流动市场中相互交易或与其他货币交易。随着这些限制的消失，“微不足道的价值”的期望也成为过去。

**如何使用Token？**

Token最明显的用途是作为数字私人货币。但是，这只是一个可能的用途。Token可以被编程为提供许多不同的功能，通常是重叠的。例如，Token可以同时传达投票权，访问权和资源所有权。货币只是第一个“应用程序”。

* 货币

Token可以作为一种货币形式，其价值通过私人交易来确定。例如，ether或bitcoin。

* 资源

Token可以表示在共享经济或资源共享环境中获得或生成的资源。例如，表示可通过网络共享的资源的存储或CPU的Token。

* 资产

Token可以代表内在或外在，有形或无形资产的所有权。例如，黄金，房地产，汽车，石油，能源等

* 访问

Token可以代表访问权限，可以访问数字或实体资产，例如论坛，专属网站，酒店房间，租车。

* 权益

Token可以代表数字组织（例如DAO）或法律虚拟主体（例如公司）中的股东权益

* 投票

Token可以代表数字或法律系统中的投票权。

* 收藏品

Token可以代表数字（例如CryptoPunks）或物理收藏品（例如绘画）

* 身份

Token可以代表数字（例如头像）或合法身份（例如国家ID）。

* 证明

Token可以代表某些机构或去中心化的信用系统（例如婚姻记录，出生证明，大学学位）的认证或事实证明。

* 实际用途

Token可用于访问或支付服务。

通常，单个Token包含其中几个功能。有时它们之间很难辨别，因为物理等价物一直是密不可分的。例如，在物理世界中，驾驶执照（认证）也是身份证件（身份证明），两者不能分开。在数字领域，以前的混合功能可以独立分离和开发（例如匿名认证）。

**ERC20 Token 标准**

第一个标准由Fabian Vogelsteller于2015年11月引入，作为以太坊征求意见（ERC）。它被自动分配了GitHub发行号码20，从而获得了名字“ERC20 Token”。

**ERC20 必须的函数和事件**

* totalSupply

返回当前存在的Token的总单位。ERC20Token可以有固定或可变的供应量。

* balanceOf

给定一个地址，返回该地址的Token余额。

* transfer

给定一个地址和数量，将该数量的Tokens从执行该方法的地址的余额转移到该地址。

* transferFrom

给定发送者，接收人和数量，将Token从一个帐户转移到另一个帐户。与approve结合使用。

* approve

在给定接收者地址和数量的情况下，授权该地址从发布批准的帐户执行多次转账，直到到达指定的数量。

* allowance

给定一个所有者地址和一个消费者地址，返回该消费者被批准从所有者的取款的剩余金额。

* Transfer event

成功转移时触发的事件（调用transfer或transferFrom）（即使对于零值转移）。

* Approval event

成功调用approve时记录的事件。

**ERC20 可选函数**

* name

返回Token的可读名称（例如“US Dollars”）。

* symbol

返回Token的人类可读符号（例如“USD”）。

* decimals

返回用于分割Token数量的小数位数。例如，如果小数为2，则将Token数除以100以获取其用户表示。

**在Solidity中定义的ERC20接口**

以下是在Solidity中ERC20接口规范的样子：

contract ERC20 {  
  function totalSupply() constant returns (uint theTotalSupply);  
  function balanceOf(address \_owner) constant returns (uint balance);  
  function transfer(address \_to, uint \_value) returns (bool success);  
  function transferFrom(address \_from, address \_to, uint \_value) returns (bool success);  
  function approve(address \_spender, uint \_value) returns (bool success);  
  function allowance(address \_owner, address \_spender) constant returns (uint remaining);  
  event Transfer(address indexed \_from, address indexed \_to, uint \_value);  
  event Approval(address indexed \_owner, address indexed \_spender, uint \_value);  
}

**ERC20 数据结构**

如果你检查任何ERC20实现，它将包含两个数据结构，一个用于追踪余额，另一个用于追踪配额（allowances）。在Solidity中，它们使用*data mapping*实现。

第一个data mapping按拥有者实现了Token余额的内部表。这允许Token合约跟踪谁拥有Token。每次转账都是从一个余额中扣除的，并且是对另一个余额的增加。

Balances: a mapping from address (owner) to amount (balance)

mapping(address => uint256) balances;

第二个数据结构是配额的data mapping。正如我们将在[ERC20工作流程：“transfer”和“approve & transferFrom”](https://www.8btc.com/books/834/ethereum-book/_book/%E7%AC%AC%E5%8D%81%E7%AB%A0.html" \l "transfer_workflows)中看到的那样，使用ERC20Token，Token的所有者可以将权限委托给花钱者，允许他们从所有者的余额中花费特定金额（配额）。ERC20合同通过二维映射追踪配额，主关键字是Token所有者的地址，映射到一个花费者地址和配额金额：

Allowances: a mapping from address (owner) to address (spender) to amount (allowance)

mapping (address => mapping (address => uint256)) public allowed;

**ERC20Token的问题**

ERC20Token标准的采用确实是爆炸性的。成千上万的Token已经启动，既可以尝试新的功能，也可以通过各种“众筹”拍卖和初始投币产品（ICO）筹集资金。然而，正如我们在将Token转移到合同地址的问题所看到的那样，存在一些潜在的陷阱。

ERC20Token不太明显的问题之一是它们暴露了Token和ether本身之间的细微差别。如果ether通过以接收者地址为目的地的交易转移，则Token转移发生在 *specific Token contract state* 中，并且将Token合同作为其目的地，而不是接收者的地址。Token合同跟踪余额并发布事件。在Token传输中，实际上没有交易发送给Token的接收者。相反，接收者的地址将被添加到Token合约本身的映射中。将ether发送到地址的交易会改变地址的状态。将Token转移到地址的交易只会改变Token合约的状态，而不会改变接收者地址的状态。即使是支持ERC20Token的钱包，也不会意识到Token的余额，除非用户明确将特定Token合约添加到“监视”中。一些钱包观察最受欢迎的Token合约，以检测由他们控制的地址持有的余额，但这仅限于ERC20合同的一小部分。

事实上，用户不太可能会追踪所有可能的ERC20Token合约中的所有余额。许多ERC20Token更像是垃圾邮件，而不是可用的Token。他们自动为拥有ether活动的帐户创建余额，以吸引用户。如果你有一个活动历史悠久的以太坊地址，特别是如果它是在预售中创建的，你会发现它充满了凭空出现的“垃圾”Tokens。当然，这个地址并不是真的充满了Token，而是那些Token合约有你的地址。如果你用于查看地址的资源管理器或钱包正在监视这些Token合约，才能看到这些余额。

Token不像ether。Ether通过send功能发送，并由合同中的任何payable函数或任何EOA接受。Token仅使用在ERC20合同中存在的transfer 或approve＆transferFrom函数发送，并且不会（至少在ERC20中）触发收款合同中的任何payable函数。Token的功能就像ether这样的加密货币，但它们带有一些细微的区别，可以打破这种错觉。

考虑另一个问题。要发送ether，或使用任何以太坊合同，你需要ether来支付gas。发送Token，你*也需要ether*。你无法用Token支付交易的gas，而Token合同也无法为你支付gas费用。这可能会导致一些相当奇怪的用户体验。例如，假设你使用交易所或Shapeshift将某些比特币转换为Token。你在钱包中“收到”该Token，该钱包会跟踪该Token的合同并显示你的余额。它看起来与你钱包中的任何其他加密货币相同。现在尝试发送Token，你的钱包会通知你，你需要ether才能这样做。你可能会感到困惑 - 毕竟你不需要ether接收Token。也许你没有ether。也许你甚至不知道该Token是以太坊上的ERC20Token，也许你认为这是一个拥有自己的区块链的加密货币。错觉就这样被打破了。

其中一些问题是ERC20Token特有的。其他更一般的问题涉及到以太坊内的抽象和界面边界。有些可以通过更改Token接口来解决，其他可能需要更改以太坊内的基础结构（例如EOAs和合同之间以及交易和消息之间的区别）。有些可能不完全“可解决”，并且可能需要用户界面设计来隐藏细微差别并使用户体验一致，而不管其底层区别如何。

**2.2开发工具**

**Geth:**Geth是一个以太坊客户端，实现以太坊相关的协议，通过Geth来同 步以太坊区块链的数据，进行挖矿。 Geth还可以作为一个完全节点，为其他轻节点提供服务，比如我们 可以通过MetaMask或者web3.js访问Geth节点进行交易、查询等操作。我们可以使用--dev参数让Geth以开发者模式启动，最好还是通过 datadir参数指定一个存储数据的目录。必须使用rpc参数，因为Geth模式 没有开启RPC服务。当然，rpc端口和地址是默认的，端口默认是8545， 地址是127.0.0.1，也可以通过rpcport和rpcaddr修改。 和Ganache不一样，Geth没有默认账户，如果需要默认账户怎么 办？我们可以通过创世区块来指定内置账户。创世区块可以通过一个 JSON文件来指定，下面我们给一个创世区块genesis.json文件内容的示 例：

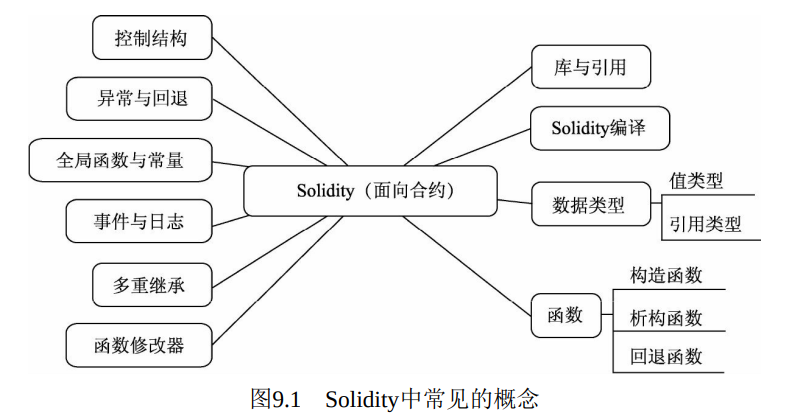
{  
"config": {  
"chainId": 18,  
"homesteadBlock": 0,  
"eip155Block": 0,  
"eip158Block": 0  
},  
"alloc": {  
"0x61f4c9cfd449b4ea1e5cd387eeda89b4f59290f5": {  
"balance": "100"  
},  
"0x34cb6fc3087a54854a64db620d3987792a6a20c5": {  
"balance": "200"  
},  
"0x4ef5c9c79b86a27bf3adb2febb0aadcdd2cca290": {  
"balance": "1000000000000000000"  
}  
},  
"coinbase": "0x0000000000000000000000000000000000000000",  
"difficulty": "0x20000",  
"extraData": "",  
"gasLimit": "0x2fefd8",  
"nonce": "0x0000000000000042",  
"mixhash": "0x000000000000000000000000000000000000000000000000000000  
0000000000",  
"parentHash": "0x000000000000000000000000000000000000000000000000000  
0000000000000",  
"timestamp": "0x00"  
}  
​

启动一个节 点： geth --rpc --datadir nodeA --networkid 18 --rpcapi "db,eth,net,web3,personal, web3" --ipcdisable --port 61910 --rpcport 8110 --rpccorsdomain "*" console 其中最重要的是使用datadir参数指定数据目录，使用networkid参数 指定网络ID，使用ipcdisable参数禁用IPC通信。port参数指定节点发现 协议使用端口。rpccorsdomain参数设置可以使用RPC接口的域名，*表 示没有限制。要想多个节点组成一个网络，需要指定networkid的值相 同，这里的networkid就是指网络ID。

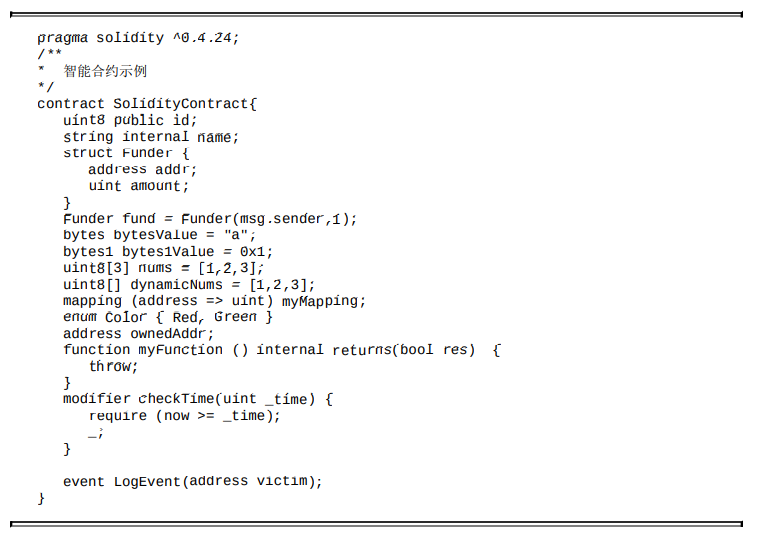
**MetaMask:**一般MetaMask是作为一个浏览器插件使用，可以看做是一个轻量 级的钱包，它提供了很多和以太坊交互的接口，降低了实际项目中与以 太坊交互的开发成本。 本质上MetaMask是使用web3.js和以太坊节点进行交互的，通过 HTTP请求调用以太坊客户端提供的RPC接口。MetaMask的所有功能都 可以通过web3.js来实现，不过使用MetaMask能简化开发工作。

**Solidity**

**:**Solidity是一门面向合约的高级语言，主要用来 编写以太坊智能合约。Solidity受C语言、Python、JavaScript语言的影 响，很多实现都和这些语言相似，但是Solidity为了编译成为以太坊虚拟 机字节码在EVM上执行，很多特性和限制都和EVM相关。 如图9.1所示为Solidity中常见的一些概念。Solidity是一门静态类型 语言，支持继承、库、自定义复杂类型和其他特性。



**智能合约示例:**



上面的例子，最外层是contract关键字，Solidity是一门面向合约的 语言，contract就是一种很好地体现，所有的合约结构都是在contract包 围之中。 像Java这种面向对象的语言，最外层是class包围，不过在Solidity中 不是定义一个类，而是定义一个合约。合约中可以包含状态变量、函 数、事件、自定义类型等，这些都会在后面详细介绍，这里我们需要知 道的是合约的最基本结构。

contract SolidityContract{  
​  
}

上面的示例就是合约的最简结构，首先是contract关键字，后面是 合约名字，花括号“{}”中是合约体。

智能合约中包含的基本结构，本节来了解一下合 约中第一行是什么意思。在Solidity文件中一般第一行是这样的：

pragma solidity ^0.4.24;

pragma是编译指示的意思，不是program。这是为了说明这个 Solidity文件可以使用哪些版本的编译器编译。^与前面5.3.2节中介绍的^ 限制范围含义一样。

状态变量 首先来看状态变量，类似于其他语言中的成员变量，但是在Solidity 中称为状态变量。之所以叫状态变量是因为Solidity是一门面向合约的语 言，以太坊的交易本质其实是一种状态机，从一种状态到另一种状态， 合约的本质也是在合适条件下触发交易。下面我们看一下合约中的状态 变量：

pragma solidity ^0.4.24;  
/\*\*  
\* 状态变量  
\*\*/  
contract StateVariableContract {  
string name;  
uint32 id;  
}  
​

上面代码中，name和id就是状态变量。

局部变量（Local Variable）也叫本地变量，与后面我们说的局部变 量是一样的。注意局部变量不仅仅是函数中的变量，参数也属于局部变 量，包括入参和出参都是局部变量。 如图9.21所示为Solidity的状态变量与局部变量，状态变量只会出现 在一个位置，局部变量可以是函数的入参、出参和函数体中定义的变 量。 为什么要分这么细？因为后面介绍数据位置相关知识时会用上，不 同的变量、不同位置、不同数据位置类型都会影响变量之间的赋值。

**Solidity函数**

function functionName() [returns ()]

与其他语言函数有所区别，如果函数有返回值，必须使用returns关 键字加上函数参数的返回值类型列表。因为Solidity允许返回多值，所以 需要确定返回顺序，具体的实例我们在9.4.4再介绍。 另外需要注意的是Solidity有两个比较相近的关键字，一个是 return，和其他函数一样用于返回值；另一个是returns，用于定义函数返 回参数，注意不要混淆。 最后，在Solidity中参数包含两部分，一部分是函数参数，我们可以 把它叫做入参；另一部分是函数返回参数，我们可以把它叫做出参。如 果没有特别说明，我们说的函数参数就包括出参和入参。

关于返回值，Solidity和其他很多语言不一样，因为Solidity支持返 回多值。下面我们就来看一个关于Solidity返回多值的实例，加深对于 Solidity返回值的认识。

pragma solidity ^0.4.24;  
/\*\*  
\* @title ReturnContract  
\* @dev 返回值测试  
\*/  
contract ReturnContract {  
/\*\*  
\* @dev 返回一个值  
\*/  
function singleReturnTest () public pure returns(uint8){  
uint8 num = 255;  
return num;  
}  
/\*\*  
\* @dev 返回多值  
\*/  
function multipleReturnTestOne () public pure returns(uint8,uint16,  
string){  
uint8 num = 255;  
return (num,num,"multiple");  
}  
/// 返回值类型必须对应  
// function multipleReturnTestTwo () public pure returns(uint8,uint16){  
// uint16 num = 255;  
// return (num,num);  
// }  
/\*\*  
\* @dev 返回值可以直接定义在返回声明中，在函数体中赋值就可以了  
\*/  
function multipleReturnTestThree () public pure returns(uint8 num,  
string name){  
num = 255;  
name = "multiple";  
}  
}  
​

与其他语言不同的是，Solidity函数的返回值不是一个单独的类型， 而是一个类型列表，返回值的声明也不是放在前面，而是放在最后。通 过上面的示例可以看到singleReturnTest只有一个返回值，就是uint8， multipleReturnTestOne返回的是多值。注意，returns类型和return类型的 对应关系，如果return的数据类型不能隐式转换为returns就会出错。 为了上面的类型转换问题，我们可以把类型定义直接写到返回值列 表中，就像示例中的multipleReturnTestThree函数一样，就不需要return 语句。

**构造函数**

与其他语言一样，Solidity也有构造函数，就是和合约同名的函数， 但只允许有一个，在合约创建时调用相关的代码完成初始化工作。如果 读者有其他编程语言开发经验，下面这样的代码相对就很容易：

pragma solidity ^0.4.24;  
/\*\*  
\* 构造函数  
\*\*/  
contract StateVariableContract {  
string name;  
uint32 id;  
function StateVariableContract(uint32 id){  
this.id = id;  
}  
//Solidity中构造函数不允许重载  
function StateVariableContract(uint32 id,string name){  
this.id = id;  
this.name = name;  
}  
function getId() public returns(uint32){  
return id;  
}  
}  
​

但在Solidity中上面的代码不能通过编译，首先是this不能使用，其 次构造函数不能重载，只能有一个。 构造函数可以有参数，但最好不要带参数，除非是为了继承，因为 合约的构造函数是合约在创建时自动调用的，所以没有用参数。虽然构 造函数可以在合约内部调用，但为什么要这样做呢？先来看一个例子：

pragma solidity ^0.4.24;  
/\*\*  
\* 构造函数  
\*\*/  
contract StateVariableContract {  
string name;  
uint32 id;  
function StateVariableContract(uint32 \_id){  
id = \_id;  
}  
//获取状态变量ID  
function getId() public returns(uint32){  
return id;  
}  
//显示调用构造函数  
function callConstructor(uint32 id) public{  
StateVariableContract(id);  
}  
}  
​

上面的代码没有问题，可以通过编译，在callConstructor函数中调用 构造函数StateVariableContract也没有问题，但意义是什么？只是为了修 改ID值吗？完全可以直接修改，不需要这么麻烦。显然，Solidity语言 的开发者也意识到了这个问题，所以同名方式的构造函数已经不被推 荐，而是推荐下面的方式：

pragma solidity ^0.4.24;  
/\*\*  
\* 构造函数  
\*\*/  
contract StateVariableContract {  
string name;  
uint32 id;  
//使用constructor关键字标识构造函数  
constructor(){  
id = 10;  
}  
function getId() returns(uint32){  
return id;  
}  
}  
​

代码中使用关键字constructor作为构造函数，创建完合约执行getId 就可以发现ID已经变为10。 constructor也可以有参数，除非是为了继承，因此一般不要有参 数。constructor只能是public或者internal类的，默认是public。如果被标 注为internal，合约就会被标注为abstract的抽象合约。下面的constructor 不是构造函数，注意其中的区别，不要混淆。

pragma solidity ^0.4.24;  
/\*\*  
\* 构造函数  
\*\*/  
contract StateVariableContract {  
string name;  
uint32 id;  
//添加了function关键字 constructor不再是一个构造函数  
function constructor(uint32 \_id){  
id = 10;  
}  
function getId() returns(uint32){  
return id;  
}  
}  
​

上面代码中虽然使用了constructor，但它并不是一个构造函数，因 为添加了function关键字，所以它只是一个普通函数，而不是一个构造 函数。

**Remix编辑器:**

Remix也是Solidity的编辑器，是一个网页版的Solidity编辑器。 Remix非常强大，并且使用起来也非常方便。

**web3.js:**web3.js是一个JavaScript库，它和以太坊有什么关系？这还需要从 JSON-RPC协议说起。我们知道以太坊的很多客户端都实现了JSONRPC协议，会启动一个服务，其他轻客户端可以通过HTTP请求来查询 相关数据。前面我们接触的Ganache相当于一个以太坊客户端，使用 Postman来发送请求，通过影响的接口获取对应的数据。 web3.js本质和Postman一样，也是使用JSON-RPC协议，通过调用对 应接口的方法来获取数据，不过web3.js是通过浏览器或者Node等来发 送HTTP请求。web3.js本身没有数据，所以要查询数据或者执行操作都要通过指定 的节点，web3.js与节点通信使用的是JSON-RPC协议。

在web3.js中我们使用最多的就是账户、交易和合约的相关接口，本 节就先来看一下账户的相关接口，当然也包含相关交易，因为交易都是 和账户相关联的。

获取账户与账户余额 获取账户和账户余额是常见的两个操作，web3.js当然也提供了这样 的接口，在前面介绍JSON-RPC-API时就介绍了两个方法，即 eth\_accounts和eth\_getBalance方法。所以这两个方法比较简单，只是简 单使用HTTP方式调用eth\_accounts和eth\_getBanlance接口，没有过多的 方法组合。下面来看一个例子：

const Web3 = require("web3");  
let web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:  
8545"))  
//通过getAccounts获取账户，返回Promise  
web3.eth.getAccounts().then(function(accounts){  
//遍历账户  
accounts.forEach(function(account){  
//通过getBalance获取账户余额，返回Promise  
web3.eth.getBalance(account).then(function(ether){  
console.log(account + ":" + ether)  
}).catch(function(err){  
console.log(err)  
})  
})  
}).catch(function(err){  
console.log(err);  
});  
​

getAccounts方法和getBalance方法返回的都是Promise，不 同的是getAccounts方法中Promise的参数是一个数组，而getBalance方法 中Promise返回的是字符串。

创建账户 创建账户有两个方法，一个是：

web3.eth.accounts.create([entropy]);

create方法的参数可选，如果设置，长度至少要32个字符，返回的 是一个对象。该方法只是创建一个账户，并不会把创建的账户添加到以 太坊客户端管理的账户中。另一个方法是：

web3.eth.personal.newAccount(password, [callback])

**Truffle:**Truffle是基于Solidity语言的一套开发框架，它简化了去中心化应用（Dapp）的构建和管理流程。本身是采用Javascript编写，支持智能合约的编译、部署和测试。truffle开发框架提供了很多功能，简化了我们的开发、编译、部署与调试过程。

**Git:**是一个开源的分布式版本控制系统，可以有效、高速地处理从很小到非常大的项目版本管理。

**Node.js:**Node.js是一个基于Chrome V8引擎的JavaScript运行环境，使用了一 个事件驱动、非阻塞式I/O的模型。Node.js的包管理器NPM，是全球最 大的开源库生态系统，因为它让JavaScript可以作为后端语言使用。

**Ganache:**以太坊的很多操作需要gas费用，按照现在以太坊的价格如果在主 链上开发测试的话，成本太高。 有读者可能会说以太坊还有很多测试链，的确如此，但是测试链的 速度很慢，确认交易需要很长时间，所以为了开发，我们需要搭建自己 的私链。 有很多客户端可以帮助我们完成这个工作，本章介绍的Ganache就 是这样一个以太坊客户端。Ganache可以帮助我们快速启动一个以太坊 私链来做开发测试、执行命令、探测区块链状态等。

**以太坊合约应用程序二进制接口（ABI）**

在计算机软件中，应用程序二进制接口（ABI）是两个程序模块之间的接口；通常，一个在机器代码级别，另一个在用户运行的程序级别。ABI定义了如何在**机器码**中访问数据结构和功能；不要与API混淆，API以高级的，通常是人类可读的格式将访问定义为**源代码**。因此，ABI是将数据编码到机器码，和从机器码解码数据的主要方式。

在以太坊中，ABI用于编码EVM的合约调用，并从交易中读取数据。ABI的目的是定义合约中的哪些函数可以被调用，并描述函数如何接受参数并返回数据。

合约ABI的JSON格式由一系列函数描述的数组给出。函数描述是一个JSON对象，它包含type，name，inputs，outputs，constant和payable字段。事件描述对象具有type，name，inputs和anonymous的字段。

函数签名与事件签名 函数签名就是函数的selector，计算方式就是函数签名的keccak256 哈希值的前4个字节。web3.js提供了一个方法encodeFunctionSignature来 计算函数签名。encodeFunction Signature方法接受一个参数，可为对象 或者字符串，对象就是函数对应的ABI，字符串就是函数的签名。 下面结合例子了解一下encodeFunctionSignature方法。假设合约中 有一个方法如下：

function add(uint a,uint b) public pure returns(uint){  
return a + b;  
}

add方法非常简单，其对应的ABI为：

{  
"constant": true,  
"inputs": [  
{  
"name": "a",  
"type": "uint256"  
},  
{  
"name": "b",  
"type": "uint256"  
}  
],  
"name": "add",  
"outputs": [  
{  
"name": "",  
"type": "uint256"  
}  
],  
"payable": false,  
"stateMutability": "pure",  
"type": "function"  
}  
​

获取交易的相关数据

web3.eth.getTransaction(transactionHash [, callback])  
web3.eth.getTransactionFromBlock(hashStringOrNumber, indexNumber [,  
callback])  
web3.eth.getTransactionReceipt(hash [, callback])  
web3.eth.getTransactionCount(address [, defaultBlock] [, callback])

上面的4个方法都和交易相关，最后一个参数是一个可选的回调函 数，返回值是一个Promise。getTransaction是用来获取交易信息，必填 参数是要获取交易的Hash，Promise的resolve参数是一个对象，对象具 体的属性可以参考前面介绍的交易属性相关内容。 getTransactionFromBlock也用来获取交易，不过参数不同，第一个 参数是要获取交易的区块Hash或者区块号，第二个参数是交易在区块中 的索引。 getTransactionReceipt是获取交易收据，参数是交易Hash，最终获取 到的是一个对象，对象中有几个需要注意的属性，一个是status，标识 交易是否成功，logs是一个交易生成的日志，更多的属性可以参考 eth\_getTransactionReceipt方法返回的对象属性。 getTransactionCount用来获取账户交易数量，第一个参数是账户地 址，第二个可选参数是指定区块，可以是区块的Hash或者区块号。需要 我们自己签名交易时会经常使用这个方法。

const Web3 = require("web3")  
let provider = new Web3.providers.HttpProvider("http://localhost:8545")  
let web3 = new Web3(provider)  
let account = '0x93D30d1FfaA4005af593F27742f4C55B87898403'  
let txHash = '0x0e02501ad8ae5c0b9aae9b8af66263b84b8a3bb556a8c80386e0595  
d8e0c7fca'  
let blockHash = '0x5de8e3bbedc426037762be5e7b140c5139322f0bba3601d0080b04  
da2955b502'  
let blockNum = 3  
//根据交易Hash查询交易信息  
web3.eth.getTransaction(txHash).then(function(transation){  
console.log(transation)  
console.log('-------------------------------------------')  
})  
//查询交易收据  
web3.eth.getTransactionReceipt(txHash).then(function(receipt){  
console.log(receipt)  
console.log('-------------------------------------------')  
})  
//根据区块Hash和和交易在区块中的索引查询交易信息  
web3.eth.getTransactionFromBlock(blockHash, 0).then(function(transation){  
console.log(transation)  
console.log('-------------------------------------------')  
})  
//根据区块号和交易在区块中的索引查询交易  
web3.eth.getTransactionFromBlock(blockNum, 0).then(function(transation){  
console.log(transation)  
console.log('-------------------------------------------')  
})  
//根据账户查询账户的交易数量  
web3.eth.getTransactionCount(account).then(function(num){  
console.log(num)  
console.log('-------------------------------------------')  
})  
​

**2.3系统开发环境**

**2.4系统结构图**

**2.5系统登录流程图**

**第三节 代码实现（功能实现）**

**3.1**

**第四节 系统测试**

**4.1测试目的**

**4.2测试方案**

**4.3测试结果**

**第五节 结论**

**5.1**

**参考文献**

[1]习近平：把区块链作为核心技术自主创新重要突破口《 人民日报海外版 》（ 2019年10月26日 第 01 版）

**致谢**