智能合约概述

简单的智能合约

让我们先看一下最基本的例子。现在就算你都不理解也不要紧，后面我们会有更深入的讲解。

存储

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData;

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

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

第一行就是告诉大家源代码使用Solidity版本0.4.0写的，并且使用0.4.0以上版本运行也没问题（最高到0.5.0，但是不包含0.5.0）。这是为了确保合约不会在新的编译器版本中突然行为异常。关键字 `pragma` 的含义是，一般来说，pragmas（编译指令）是告知编译器如何处理源代码的指令的（例如， pragma once ）。

Solidity中合约的含义就是一组代码（它的 *函数*)和数据（它的 *状态* ），它们位于以太坊区块链的一个特定地址上。 代码行 `uint storedData;` 声明一个类型为 `uint` (256位无符号整数)的状态变量，叫做 `storedData` 。 你可以认为它是数据库里的一个位置，可以通过调用管理数据库代码的函数进行查询和变更。对于以太坊来说，上述的合约就是拥有合约（owning contract）。在这种情况下，函数 `set` 和 `get` 可以用来变更或取出变量的值。

要访问一个状态变量，并不需要像 `this.` 这样的前缀，虽然这是其他语言常见的做法。

该合约能完成的事情并不多（由于以太坊构建的基础架构的原因）：它能允许任何人在合约中存储一个单独的数字，并且这个数字可以被世界上任何人访问，且没有可行的办法阻止你发布这个数字。当然，任何人都可以再次调用 `set` ，传入不同的值，覆盖你的数字，但是这个数字仍会被存储在区块链的历史记录中。随后，我们会看到怎样施加访问限制，以确保只有你才能改变这个数字。

注解

所有的标识符（合约名称，函数名称和变量名称）都只能使用ASCII字符集。UTF-8编码的数据可以用字符串变量的形式存储。

警告

小心使用Unicode文本，因为有些字符虽然长得相像（甚至一样），但其字符码是不同的，其编码后的字符数组也会不一样。

子货币（Subcurrency）例子

下面的合约实现了一个最简单的加密货币。这里，币确实可以无中生有地产生，但是只有创建合约的人才能做到（实现一个不同的发行计划也不难）。而且，任何人都可以给其他人转币，不需要注册用户名和密码 —— 所需要的只是以太坊密钥对。

```
pragma solidity ^0.4.21;

contract Coin {
    // 关键字“public”让这些变量可以从外部读取
    address public minter;
    mapping (address => uint) public balances;

    // 轻客户端可以通过事件针对变化作出高效的反应
    event Sent(address from, address to, uint amount);

    // 这是构造函数，只有当合约创建时运行
    function Coin() public {
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) public {
        if (msg.sender != minter) return;
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) public {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

这个合约引入了一些新的概念，让我们逐一解读。

`address public minter;` 这一行声明了一个可以被公开访问的 `address` 类型的状态变量。`address` 类型是一个160位的值，且不允许任何算数操作。这种类型适合存储合约地址或外部人员的密钥对。关键字 `public` 自动生成一个函数，允许你在这个合约之外访问这个状态变量的当前值。如果没有这个关键字，其他的合约没有办法访问这个变量。由编译器生成的函数的代码大致如下所示：

```
function minter() returns (address) { return minter; }
```

当然，加一个和上面完全一样的函数是行不通的，因为我们会有同名的一个函数和一个变量，这里，主要是希望你能明白——编译器已经帮你实现了。

下一行，`mapping (address => uint) public balances;` 也创建一个公共状态变量，但它是一个更复杂的数据类型。 该类型将address映射为无符号整数。 Mappings 可以看作是一个 哈希表 它会执行虚拟初始化，以使所有可能存在的键都映射到一个字节表示为全零的值。 但是，这种类比并不太恰当，因为它既不能获得映射的所有键的列表，也不能获得所有值的列表。 因此，要么记住你添加到mapping中的数据（使用列表或更高级的数据类型会更好），要么在不需要键列表或值列表的上下文中使用它，就如本例。 而由 `public` 关键字创建的getter函数 getter function 则是更复杂一些的情况， 它大致如下所示：

```
function balances(address _account) public view returns (uint) {
    return balances[_account];
}
```

正如你所看到的，你可以通过该函数轻松地查询到账户的余额。

`event Sent(address from, address to, uint amount);` 这行声明了一个所谓的“事件（event）”，它会在 `send` 函数的最后一行被发出。用户界面（当然也包括服务器应用程序）可以监听区块链上正在发送的事件，而不会花费太多成本。一旦它被发出，监听该事件的listener都将收到通知。而所有的事件都包含了 `from` ， `to` 和 `amount` 三个参数，可方便追踪事务。 为了监听这个事件，你可以使用如下代码：

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

这里请注意自动生成的 `balances` 函数是如何从用户界面调用的。

特殊函数 `Coin` 是在创建合约期间运行的构造函数，不能在事后调用。 它永久存储创建合约的人的地址: `msg` (以及 `tx` 和 `block`) 是一个神奇的全局变量，其中包含一些允许访问区块链的属性。 `msg.sender` 始终是当前（外部）函数调用的来源地址。

最后，真正被用户或其他合约所调用的，以完成本合约功能的方法是 `mint` 和 `send`。 如果 `mint` 被合约创建者外的其他人调用则什么也不会发生。 另一方面， `send` 函数可被任何人用于向他人发送币 (当然，前提是发送者拥有这些币)。记住，如果你使用合约发送币给一个地址，当你在区块链浏览器上查看该地址时是看不到任何相关信息的。因为，实际上你发送币和更改余额的信息仅仅存储在特定合约的数据存储器中。通过使用事件，你可以非常简单地为你的新币创建一个“区块链浏览器”来追踪交易和余额。

区块链基础

对于程序员来说，区块链这个概念并不难理解，这是因为大多数难懂的东西 (挖矿, 哈希 ， 椭圆曲线密码学 ， 点对点网络（P2P） 等) 都只是用于提供特定的功能和承诺。你只需接受这些既有的特性功能，不必关心底层技术，比如，难道你必须知道亚马逊的 AWS 内部原理，你才能使用它吗?

交易/事务

区块链是全球共享的事务性数据库，这意味着每个人都可加入网络来阅读数据库中的记录。如果你想改变数据库中的某些东西，你必须创建一个被所有其他人所接受的事务。事务一词意味着你想做的（假设您想要同时更改两个值），要么一点没做，要么全部完成。此外，当你的事务被应用到数据库时，其他事务不能修改数据库。

举个例子，设想一张表，列出电子货币中所有账户的余额。如果请求从一个账户转移到另一个账户，数据库的事务特性确保了如果从一个账户扣除金额，它总被添加到另一个账户。如果由于某些原因，无法添加金额到目标账户时，源账户也不会发生任何变化。

此外，交易总是由发送人（创建者）签名。

这样，就可非常简单地为数据库的特定修改增加访问保护机制。在电子货币的例子中，一个简单的检查可以确保只有持有账户密钥的人才能从中转账。

区块

在比特币中，要解决的一个主要难题，被称为“双花攻击 (double-spend attack)”：如果网络存在两笔交易，都想花光同一个账户的钱时（即所谓的冲突）会发生什么情况？交易互相冲突?

简单的回答是你不必在乎此问题。网络会为你自动选择一条交易序列，并打包到所谓的“区块”中，然后它们将在所有参与节点中执行和分发。如果两笔交易互相矛盾，那么最终被确认为后发生的交易将被拒绝，不会被包含到区块中。

这些块按时间形成了一个线性序列，这正是“区块链”这个词的来源。区块以一定的时间间隔添加到链上 —— 对于以太坊，这间隔大约是17秒。

作为“顺序选择机制”（也就是所谓的“挖矿”）的一部分，可能有时会发生块（blocks）被回滚的情况，但仅在链的“末端”。末端增加的块越多，其发生回滚的概率越小。因此你的交易被回滚甚至从区块链中抹除，这是可能的，但等待的时间越长，这种情况发生的概率就越小。

以太坊虚拟机

概述

以太坊虚拟机 EVM 是智能合约的运行环境。它不仅是沙盒封装的，而且是完全隔离的，也就是说在 EVM 中运行代码是无法访问网络、文件系统和其他进程的。甚至智能合约之间的访问也是受限的。

账户

以太坊中有两类账户（它们共用同一个地址空间）： **外部账户** 由公钥-私钥对（也就是人）控制； **合约账户** 由和账户一起存储的代码控制.

外部账户的地址是由公钥决定的，而合约账户的地址是在创建该合约时确定的（这个地址通过合约创建者的地址和从该地址发出过的交易数量计算得到的，也就是所谓的“nonce”）

无论帐户是否存储代码，这两类账户对 EVM 来说是一样的。

每个账户都有一个键值对形式的持久化存储。其中 key 和 value 的长度都是256位，我们称之为 **存储** 。

此外，每个账户有一个以太币余额（ **balance** ）（单位是“Wei”），余额会因为发送包含以太币的交易而改变。

交易

交易可以看作是从一个帐户发送到另一个帐户的消息（这里的账户，可能是相同的或特殊的零帐户，请参阅下文）。它能包含一个二进制数据（合约负载）和以太币。

如果目标账户含有代码，此代码会被执行，并以 payload 作为入参。

如果目标账户是零账户（账户地址为 `0` ），此交易将创建一个 **新合约** 。 如前文所述，合约的地址不是零地址，而是通过合约创建者的地址和从该地址发出过的交易数量计算得到的（所谓的“nonce”）。 这个用来创建合约的交易的 payload 会被转换为 EVM 字节码并执行。执行的输出将作为合约代码被永久存储。这意味着，为创建一个合约，你不需要发送实际的合约代码，而是发送能够产生合约代码的代码。

> **注解**
>
> 在合约创建的过程中，它的代码还是空的。所以直到构造函数执行结束，你都不应该在其中调用合约自己函数。

Gas

一经创建，每笔交易都收取一定数量的 **gas** ， 目的是限制执行交易所需要的工作量和为交易支付手续费。EVM 执行交易时，gas 将按特定规则逐渐耗尽。

gas price 是交易发送者设置的一个值，发送者账户需要预付的手续费= `gas_price * gas` 。如果交易执行后还有剩余， gas 会原路返还。

无论执行到什么位置，一旦 gas 被耗尽（比如降为负值），将会触发一个 out-of-gas 异常。当前调用帧（call frame）所做的所有状态修改都将被回滚。

译者注：调用帧（call frame），指的是下文讲到的EVM的运行栈（stack）中当前操作所需要的若干元素。

存储，内存和栈

每个账户有一块持久化内存区称为 **存储** 。 存储是将256位字映射到256位字的键值存储区。在合约中枚举存储是不可能的，且读存储的相对开销很高，修改存储的开销甚至更高。合约只能读写存储区内属于自己的部分。

第二个内存区称为 **内存** ，合约会试图为每一次消息调用获取一块被重新擦拭干净的内存实例。 内存是线性的，可按字节级寻址，但读的长度被限制为256位，而写的长度可以是8位或256位。当访问（无论是读还是写）之前从未访问过的内存字（word）时（无论是偏移到该字内的任何位置），内存将按字进行扩展（每个字是256位）。扩容也将消耗一定的gas。 随着内存使用量的增长，其费用也会增高（以平方级别）。

EVM 不是基于寄存器的，而是基于栈的，因此所有的计算都在一个被称为 **栈（stack）** 的区域执行。 栈最大有1024个元素，每个元素长度是一个字（256位）。对栈的访问只限于其顶端，限制方式为：允许拷贝最顶端的16个元素中的一个到栈顶，或者是交换栈顶元素和下面16个元素中的一个。所有其他操作都只能取最顶的两个（或一个，或更多，取决于具体的操作）元素，运算后，把结果压入栈顶。当然可以把栈上的元素放到存储或内存中。但是无法只访问栈上指定深度的那个元素，除非先从栈顶移除其他元素。

指令集

EVM的指令集量应尽量少，以最大限度地避免可能导致共识问题的错误实现。所有的指令都是针对"256位的字（word）"这个基本的数据类型来进行操作。具备常用的算术、位、逻辑和比较操作。也可以做到有条件和无条件跳转。此外，合约可以访问当前区块的相关属性，比如它的编号和时间戳。

消息调用

合约可以通过消息调用的方式来调用其它合约或者发送以太币到非合约账户。消息调用和交易非常类似，它们都有一个源、目标、数据、以太币、gas和返回数据。事实上每个交易都由一个顶层消息调用组成，这个消息调用又可创建更多的消息调用。

合约可以决定在其内部的消息调用中，对于剩余的 gas ，应发送和保留多少。如果在内部消息调用时发生了out-of-gas异常（或其他任何异常），这将由一个被压入栈顶的错误值所指明。此时，只有与该内部消息调用一起发送的gas会被消耗掉。并且，Solidity中，发起调用的合约默认会触发一个手工的异常，以便异常可以从调用栈里“冒泡出来”。 如前文所述，被调用的合约（可以和调用者是同一个合约）会获得一块刚刚清空过的内存，并可以访问调用的payload——由被称为 calldata 的独立区域所提供的数据。调用执行结束后，返回数据将被存放在调用方预先分配好的一块内存中。 调用深度被 **限制** 为 1024 ，因此对于更加复杂的操作，我们应使用循环而不是递归。

委托调用/代码调用和库

有一种特殊类型的消息调用，被称为 **委托调用(delegatecall)** 。它和一般的消息调用的区别在于，目标地址的代码将在发起调用的合约的上下文中执行，并且 `msg.sender` 和 `msg.value` 不变。 这意味着一个合约可以在运行时从另外一个地址动态加载代码。存储、当前地址和余额都指向发起调用的合约，只有代码是从被调用地址获取的。 这使得 Solidity 可以实现”库“能力：可复用的代码库可以放在一个合约的存储上，如用来实现复杂的数据结构的库。

日志

有一种特殊的可索引的数据结构，其存储的数据可以一路映射直到区块层级。这个特性被称为 **日志(logs)** ，Solidity用它来实现 **事件(events)** 。合约创建之后就无法访问日志数据，但是这些数据可以从区块链外高效的访问。因为部分日志数据被存储在 布隆过滤器（Bloom filter）中，我们可以高效并且加密安全地搜索日志，所以那些没有下载整个区块链的网络节点（轻客户端）也可以找到这些日志。

创建

合约甚至可以通过一个特殊的指令来创建其他合约（不是简单的调用零地址）。创建合约的调用 **create calls** 和普通消息调用的唯一区别在于，负载会被执行，执行的结果被存储为合约代码，调用者/创建者在栈上得到新合约的地址。

自毁

合约代码从区块链上移除的唯一方式是合约在合约地址上的执行自毁操作 `selfdestruct` 。合约账户上剩余的以太币会发送给指定的目标，然后其存储和代码从状态中被移除。

警告

尽管一个合约的代码中没有显式地调用 `selfdestruct` ，它仍然有可能通过 `delegatecall` 或 `callcode` 执行自毁操作。

注解

旧合约的删减可能会，也可能不会被以太坊的各种客户端程序实现。另外，归档节点可选择无限期保留合约存储和代码。

注解

目前， **外部账户** 不能从状态中移除。