1. **ABI最大支持字节为32个字节**
2. **抽象接口**

接口(Abstract Contracts)

抽象函数是没有函数实体的的函数。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的实现。

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虚合约(abstract contract)里继承，但却没实现所有函数，那么它也是一个虚合约(abstract contract)

1. **继承**

solidity支持多继承，通过拷贝代码的方式来实现多态。

所有函数调用是虚拟(virtual)的，这意味着最远的派生方式会被调用，除非明确指定了合约。

尽管一个合约从多个其它合约那里继承，但在区块链上仅会创建一个合约，在基合约里的代码往往会复制到最终的合约里。

1. **通过javascript来获取事件**

var abi = /\* abi as generated by the compiler \*/;var ClientReceipt = web3.eth.contract(abi);var clientReceipt = ClientReceipt.at(0x123 /\* address \*/);

var event = clientReceipt.Deposit();

// watch for changes

event.watch(function(error, result){

// result will contain various information

// including the argumets given to the Deposit

// call.

if (!error)

console.log(result);

});

// Or pass a callback to start watching immediatelyvar event = clientReceipt.Deposit(function(error, result) {

if (!error)

console.log(result);

});

1. **常状态变量**

状态变量(state variables)可以被定义为常量(constant)，但暂不支持数组(array)，自定义结构(struct types)，字典(mapping)。

1. **可见性控制**

Solidity有两种函数调用方式，一种是内部调用，不会创建一个EVM调用，也叫做消息调用(message call)，另一种则是外部调用，会创建EVM调用。所以函数和状态变量(state variables)对应就有4种可见性。

函数可以被标记为external,public,internal,private四种，其中默认是public。对于状态变量(state variables),不能是external的，其默认值是internal。

1. **数据存放的位置**

复杂类型，如数组(arrays)和数据结构(struct)在Solidity中有一个额外的属性，数据的存储位置。可选为memory和storage。

memory存储位置同我们普通程序的内存一致。即分配，即使用，越过作用域即不可被访问，等待被回收。而在区块链上，由于底层实现了图灵完备，故而会有非常多的状态需要永久记录下来。比如，参与众筹的所有参与者。那么我们就要使用storage这种类型了，一旦使用这个类型，数据将永远存在。

基于程序的上下文，大多数时候这样的选择是默认的，我们可以通过指定关键字storage和memory修改它。

默认的函数参数，包括返回的参数，他们是memory。默认的局部变量是storage的[1](http://www.tryblockchain.org/Solidity-DataLocation-%E6%95%B0%E6%8D%AE%E4%BD%8D%E7%BD%AE.html" \l "fn1)。而默认的状态变量（合约声明的公有变量）是storage。

另外还有第三个存储位置calldata。它存储的是函数参数，是只读的，不会永久存储的一个数据位置。外部函数的参数（不包括返回参数）被强制指定为calldata。效果与memory差不多。

数据位置指定非常重要，因为不同数据位置变量赋值产生的结果也不同。在memory和storage之间，以及它们和状态变量（即便从另一个状态变量）中相互赋值，总是会创建一个完全不相关的拷贝。

1. **solidity里面的库**

库与合约类似，但它的目的是在一个指定的地址，且仅部署一次，然后通过EVM的特性DELEGATECALL(Homestead之前是用CALLCODE)来复用代码。这意味着库函数调用时，它的代码是在调用合约的上下文中执行。使用this将会指向到调用合约，而且可以访问调用合约的存储(storage)。因为一个合约是一个独立的代码块，它仅可以访问调用合约明确提供的状态变量(state variables)，否则除此之外，没有任何方法去知道这些状态变量。

使用库合约的合约，可以将库合约视为隐式的父合约(base contracts)，当然它们不会显式的出现在继承关系中。但调用库函数的方式非常类似，如库L有函数f()，使用L.f()即可访问。此外，internal的库函数对所有合约可见，如果把库想像成一个父合约就能说得通了。当然调用内部函数使用的是internal的调用惯例，这意味着所有internal类型可以传进去，memory类型则通过引用传递，而不是拷贝的方式。在EVM的实现方式是，internal的库函数，将会被拉取(pull into)到调用合约中，然后执行一个普通的JUMP来代替DELEGATECALL。

下面的例子展示了如何使用库(后续在using for章节有一个更适合的实现Set的例子)。

pragma solidity ^0.4.0;

library Set {

// We define a new struct datatype that will be used to

// hold its data in the calling contract.

struct Data { mapping(uint => bool) flags; }

// Note that the first parameter is of type "storage

// reference" and thus only its storage address and not

// its contents is passed as part of the call. This is a

// special feature of library functions. It is idiomatic

// to call the first parameter 'self', if the function can

// be seen as a method of that object.

function insert(Data storage self, uint value)

returns (bool)

{

if (self.flags[value])

return false; // already there

self.flags[value] = true;

return true;

}

function remove(Data storage self, uint value)

returns (bool)

{

if (!self.flags[value])

return false; // not there

self.flags[value] = false;

return true;

}

function contains(Data storage self, uint value)

returns (bool)

{

return self.flags[value];

}

}

contract C {

Set.Data knownValues;

function register(uint value) {

// The library functions can be called without a

// specific instance of the library, since the

// "instance" will be the current contract.

if (!Set.insert(knownValues, value))

throw;

}

// In this contract, we can also directly access knownValues.flags, if we want.

}

说明

Library定义了一个数据结构体，用来在调用的合约中使用(库本身并未实际存储的数据)。如果函数需要操作数据，这个数据一般是通过库函数的第一个参数传入，按惯例会把参数名定为self。

另外一个需要留意的是上例中self的类型是storage，那么意味着传入的会是一个引用，而不是拷贝的值，那么修改它的值，会同步影响到其它地方，俗称引用传递，非值传递。

库函数的使用不需要实例化，c.register中可以看出是直接使用Set.insert。但实际上当前的这个合约本身就是它的一个实例。

这个例子中，c可以直接访问，knownValues。虽然这个值主要是被库函数使用的。

当然，上述并非是使用库的惯例，可以不需要定义结构体，不需要使用storage类型的参数，还可以有多个storage引用类型的参数在任何位置。

调用Set.contains，Set.remove，Set.insert都会编译为对外部external合约和库的调用(DELEGATECALL)。如果使用库，需要注意的是一个实实在在的外部函数调用发生了。尽管msg.send，msg.value，this还会保持它们在调用中的值(在Homestead之前，由于实际使用的是CALLCODE，msg.sender，msg.value会变化)。