

第一章 智能合约数据类型

1. 布尔类型

布尔类型导入

1. 布尔类型

像任何编程语言一样，Solidity 提供了一种布尔数据类型。布尔类型用来判断条件是否成立，例如 true 或 false、1 或 0 等。常用于解决结果为二分类的问题。例如：地址是否合法？交易是否成功？整数是否为素数？

布尔类型使用 **bool** 关键字声明，此数据类型的有效值为 **true** 和 **false**。true 为真，false 为假。

值得注意的是，Solidity 中的布尔不能转换为整数，就像它们在其他编程语言中一样。它是一个值类型，任何赋值给其他的布尔变量都会创建一个新副本。Solidity 中 bool 的默认值为 false。

声明和赋值 bool 数据类型的代码如下：

```
bool isActive; //不带默认值  
bool isActive=false; //带默认值
```

2. 支持的运算符

bool 类型支持的运算符是逻辑运算符和关系运算符。

支持的逻辑运算符包括：

!	逻辑非
&&	逻辑与 (and)

	逻辑或 (or)
--	----------

支持的关系运算符包括：

==	等于
!=	不等于

使用运算符的代码如下：

```
bool public _bool = true;

bool public _bool1 = !_bool; //取非

bool public _bool2 = _bool && _bool1; //与

bool public _bool3 = _bool || _bool1; //或

bool public _bool4 = _bool == _bool1; //相等

bool public _bool5 = _bool != _bool1; //不相等
```

输出：

```
_bool: true
_bool1: false
_bool2: false
_bool3: true
_bool4: false
_bool5: true
```

解析：

变量 `_bool` 的取值是 `true`；`_bool1` 是 `_bool` 的非，为 `false`；`_bool && _bool1` 为 `false`；`_bool || _bool1` 为 `true`；`_bool == _bool1` 为 `false`；`_bool != _bool1` 为 `true`。

3. 短路运算符

在 C++ 中，逻辑运算符和表达式中存在短路问题。和 C++ 中一样，Solidity 中的运算符 `||` 和 `&&` 都遵循同样的短路规则：

$f||g$ 若 f 为 $true$ ，则结果一定为 $true$ ，那么 g 就不会被执行。

$f\&\&g$ ，若 f 为 $false$ ，则结果一定为 $false$ ，那么 g 就不会被执行。

有多个语句也是一样：

逻辑或中，前面有 $true$ ，则 $true$ 后面的不执行。

逻辑与中，前面有 $false$ ，则 $false$ 后面的不执行。



布尔类型分析

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.4.23;

contract BooleanTest{

    bool _a;

    function getBool() public view returns(bool){

        return !_a;

    }

}
```

代码解析：

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.4.23;
```

声明合约版本需兼容 0.4.23 以上版本。

```
contract BooleanTest{
```

定义了一个名为 BooleanTest 的合约。contract 为合约声明关键字，BooleanTest 为自定义合约名称。

```
bool _a;
```

声明了一个名为_a 的变量。bool 为布尔类型声明关键字，_a 为自定义变量名称。

```
function getBool() public view returns(bool){
```

声明一个函数用于得到返回值，`returns(bool)`需要定义为返回值的类型 `bool`，`getBool` 为自定义函数名称。

```
return !_a;
```

返回!`_a` 的值，`_a` 声明时未规定值，因此默认值为 `false`，逻辑运算符! 代表非，即非 `false`，返回值为 `true`。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。



2. 整型

整型导入

1. 整型

整数有助于将数字存储在合约中。Solidity 提供以下两种类型的整数：

有符号的整数 (int)：带符号的整数可以同时具有负值和正值。

无符号整数 (uint)：无符号整数只能保持正值和零。

整型的关键字有 **int8**、**int16**、到 **int256**，数字以 8 步进。对应的**无符号整型**有 **uint8** 到 **uint256**，**uint** 和 **int** 默认对应的是 **uint256** 和 **int256**。

根据要求，应选择适当大小的整数。例如，当存储 0~255 之间的值时，uint8 是合适的，而存储介于-128~127 之间则 int8 更合适。对于更高的值，可以使用更大的整数。

有符号和无符号整数的缺省值为零，在声明时它们会自动初始化。

声明方式如下：

```
int8 x = -1;
uint16 y = 1;
uint32 z;
```

1.1 有符号整型 int

```
int8 x = -1;
int256 a=8;
```

int 分为 int8、int16、int24，8 位一步，直到 int256。如果后面的数字省略，则默认为 int256。

int8 后面的 8 是什么意思呢？

即用 8 位二进制来存储 (其中左边第 1 位用来表示 + - 符号), 代表可存储的数字范围, 如:

11111111 ~ 01111111, 左边第 1 位用来表示符号, 1 表示负, 0 表示正, 换算成十进制则为 $-(1 + 2 + 4 + 8 + 16 + 32 + 64) \sim (1 + 2 + 4 + 8 + 16 + 32 + 64)$ 。即可存储 -127 ~ 127 之间 255 个数字。也可以理解成存储范围为 2 的多少次方。

1.2 无符号整型 uint

```
uint16 y = 0;  
uint32 z = 200;
```

uint 分为 uint8、uint16、uint24, 8 位一步, 直到 uint256, 如果后面的数字省略, 则默认为 uint256。

uint8 代表可存储的数字范围, 用 8 位二进制来存储 (左边第 1 位不用来表示 + - 符号, 即大于等于 0 的数), 如:

00000000 ~ 11111111, 即可存储 0 ~ 255 之间 255 个数字。也可以理解成存储范围为 2 的多少次方。

2. 支持的运算符

整型支持的运算操作有**比较运算符**、**位运算符**和**算术运算符**三种。

比较运算符 (返回布尔值 true 或 false)	位运算符	算术运算符
<= (小于等于)	& (与)	+ (加)
< (小于)	(或)	- (减)
== (等于)	^ (异或)	一元运算符 "-"

!= (不等于)	~ (位取反)	一元运算符 "+"
>= (大于等于)		* (乘)
> (大于)		/ (除)
		% (取余)
		** (幂)
		<< (左移位)
		>> (右移位)

<<(左移) , >>(右移), 左移和右移操作, $a \ll b$, 可以理解为 a 乘以 2 的 b 次方, 表示为 $a * 2^b$, 同理右移 $a \gg b$ 表示为 $a / 2^b$ 。

位运算不是原数值直接运算, 而是转化为二进制数据, 在数字的二进制补码表示上执行。

使用运算符的代码如下:

```
uint256 public _number = 20220330; // 256 位正整数
uint256 public _number1 = _number + 1; // +
uint256 public _number2 = 2**2; // 指数
uint256 public _number3 = 7 % 2; // 取余数
bool public _numberbool = _number2 > _number3; // 比大小
```

输出:

_number: 20220330

_number1: 20220331

_number2: 4

_number3: 1

_numberbool: true

3. 注意

(1) 整数除法总是截断的。单如果运算的是常量（常量后面的内容会讲到），则不会截断。

(2) 整数除 0 会抛异常，即 $x/0$ 为非法的。

(3) 右移位和除是等价的，如 $x * 2^{**}y$ 是相等的。移位运算的结果的正负取决于运算符左边的数。右移位一个负数边的数。右移位一个负数，向下取整时会为 0。

(4) 不能进行负移位，即运算符右边的数不可以为负数，否则会抛出运行时异常，如 $3 >> -1$ 为非法的。

(5) 有符号整数是不能够使用 `***` 操作。



整型分析

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.4.23;

contract UintTest{

    uint a=10;

    uint b=2;

    function getUint() public view returns(uint){

        uint sum=a*b+3%2;

        return sum;

    }

}
```

代码解析:

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.4.23;
```

声明合约版本需兼容 0.4.23 以上版本。

```
contract UintTest{
```

定义了一个名为 UintTest 的合约。contract 为合约声明关键字，UintTest 为自定义合约名称。

```
uint a=10;
```

声明了一个名为 a 的变量并赋值为 10。uint 为整型声明关键字，a 为自定义变量名称。

```
uint b=2;
```

声明了一个名为 b 的变量并赋值为 2。uint 为整型声明关键字，b 为自定义变量名称。

```
function getUint() public view returns(uint){
```

声明一个函数用于得到返回值，returns(uint)需要定义为返回值的类型 uint，getUint 为自定义函数名称。

```
uint sum=a**b+3%2;
```

声明了一个名为 sum 的变量并赋值为 $a**b+3\%2$ 的计算结果。uint 为整型声明关键字，sum 为自定义变量名称。

```
return sum;
```

返回 sum 的值，返回值为 $a**b+3\%2$ 的计算结果 101。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。

3. 地址类型

地址类型导入

1. 地址类型

地址类型是以太坊的一个特有类型，它是一个 **160 位二进制的数**，不同于 Bitcoin 的 UTXO 模型，以太坊交易是以其特定的账户模型，每个账户都有其对应的地址。

在常见的编程语言里并没有地址这样的类型，但是地址类型在以太坊中却非常重要。因为以太坊的账户需要用地址来表示。

私钥是一个 32 个字节的数、256 位的二进制数，也就是 64 位的十六进制数。

公钥是由私钥生成的，私钥是对一个信息进行签名，公钥用来验证合法性。

地址（这里指的是 ETH 地址）是把公钥用 sha256 hash 之后，**取其后 160 位生成的 16 进制字符串（40 个字符）再加上前缀“0x”（总共 42 位）**。

地址是所有合约的基础，所有的合约都会继承地址对象，通过合约的地址串，调用合约内的函数。

2. 地址类型的声明

地址类型有两种形式，他们大致相同：

address：保存一个 160 位(20 个字节)，一般表示成 40 个十六进制数（以太坊地址的大小），由于一个字节等于 8 位，所以地址也可以使用 uint160 来声明。

address payable：可支付地址，与 address 相同，不过有成员函数 transfer 和 send。

这种区别背后的思想是 **address payable** 可以接受以太币的地址，如果在函数中涉

及到以太币的转移，需要使用到 payable 关键词。意味着可以在调用这笔函数的消息中附带以太币。而一个普通的 address 则不能。使用 payable 关键字声明的函数能够接受来自调用者的以太币。如果发送者没有提供以太币，则调用将会失败。如果在函数中涉及到以太币的转移，需要使用到 payable 关键词。意味着可以在调用这笔函数的消息中附带以太币。如果你需要 address 类型的变量，并计划发送以太币给这个地址，那么声明类型为 address payable 可以明确表达出你的需求。

声明方式如下：

```
address addr = 0xca35b7d915458ef540ade6068dfe2f44e8fa733c;  
address payable addr;
```

address 和 address payable 的区别是在 0.5.0 版本引入的，同样从这个版本开始，合约类型不再继承自地址类型。

不过如果合约有可支付的回退（ payable fallback ）函数或 receive 函数，合约类型仍然可以显式转换为 address 或 address payable 。

3. 校验地址合法性

因为 ETH 的地址是按照规则产生的，所以我们可以去校验一个 ETH 地址是否合法，步骤如下：

- (1) 先判断地址非空和是否 0x 开头
- (2) 把 16 进制字符串转成 10 进制数，看是否能否转换成功
- (3) 判断是否长度是 40 位（去掉 0x）
- (4) 如果它具有所有这些属性并且它是全部小型大写字母或全部大写字母，则它是有效的地址
- (5) 如果地址具有大写字母和小写字母的组合，则必须检查该地址的校验和，校验

和，在数据处理和数据通信领域中，用于校验目的地一组数据项的和。它通常是以十六进制为数制表示的形式。如果校验和的数值超过十六进制的 FF，也就是 255，就要求其补码作为校验和。通常用来在通信中，尤其是远距离通信中保证数据的完整性和准确性。

4. 支持的运算符

地址类型也有成员。地址是所有合约的基础（基类），即合约也可以是一个类并且继承自地址类型。从 Solidity 0.5.0 版本开始，合约将不再继承自地址类型，但会保留显式转换为地址。

地址类型支持的运算符有： \leq 、 $<$ 、 $=$ 、 $!=$ 、 \geq 、 $>$ 。



地址类型分析

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.4.0;

contract Addr {

    address public addr=0x8cC758f5a41bdF6Ab749944c737F86e6a4B4070B;

    function getsenderaddr() public view returns(address){

        return msg.sender;

    }

}
```

代码解析:

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.4.0;
```

声明合约版本需兼容 0.4.0 以上版本。

```
contract Addr {
```

定义了一个名为 Addr 的合约。contract 为合约声明关键字，Addr 为自定义合约名称。

```
address public addr=0x8cC758f5a41bdF6Ab749944c737F86e6a4B4070B;
```

声明了一个地址类型名为 addr 的变量并赋值为一个地址。address 为地址类型声明关键字，public 表示变量是公开的，addr 为自定义变量名称。

```
function getsenderaddr() public view returns(address){
```

声明一个函数用于得到返回值，returns(address)需要定义为返回值的类型 address，
getsenderaddr 为自定义函数名称。

```
return msg.sender;
```

返回 msg.sender 的值，msg.sender 是一个全局变量，可以获得当前合约的地址，
全局变量是全局范围工作的变量，是 Solidity 预留关键字。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成
一个复合语句，最后一个大括号表示程序结束。



4. 数组

数组导入

1. 数组

数组是一种数据结构，它是存储同类元素的有序集合。数组中的特定元素由索引访问，索引值从 0 开始。

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**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[]。

2. 创建数组

2.1 使用字面量创建数组

字面量创建数组的格式：

```
type [arraySize] arrayName;
```

通过数组字面量，创建的数组是 memory 的，同时还是定长的，例：

```
uint [10] balance;
```

数组可以存储元素，但元素类型需要是指定存储的元素类型，若声明的数组是定长的，需要实际长度与声明的相匹配，否则编译器会报错：

```
uint [3] a=[1,2,3];
```

数组通过下标进行访问，序号是从 0 开始的。例如，访问第 1 个元素时使用 testarray[0]，对元素赋值 testarray[0] = 1。

数组字面量是写作表达式形式的数组，并且不会立即赋值给变量，如下例子，需要声明第一个元素的类型，不然默认用存储空间最小的类型 uint8：

```
uint[5] x = [uint(1), 3, 4];
```

2.2 使用 new 关键字创建数组

可以省略数组长度，定义为变长数组：

```
uint [] balance = [1, 2, 3];
```

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，但是必须声明长度，并且声明后长度不能改变。

```
uint [] testarray = new uint[](7);  
bytes public _data = new bytes(10);  
string [] adarr1 = new string[](4);
```

变长数组需要一个一个元素的赋值。

```
uint[] memory x = new uint[](3);  
x[0] = 1;  
x[1] = 3;  
x[2] = 4;
```

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。

```
uint[] memory a = new uint[](7);  
a.length=8; //不可调整数组大小  
a.push(32); // 大小不可调整
```

类型为数组的状态变量，可以标记为 public 类型，从而让 Solidity 创建一个访问器，如果要访问数组的某个元素，指定数字下标就好了。

```
uint [] public stateVar=[uint(1)];
```

3. 多维数组

还可以声明一个多维数组。例如声明一个类型为 uint、长度为 5 的变长数组 (5 个元素都是变长数组), 则可以声明为:

```
uint[][5] x;
```

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][] x;
```

`a[i][j]` 代表第 `j` 个元素块, `i` 代表第 `j` 个元素块的第 `i` 个。

注意, 这种多维数组的声明方式可能与其他语言不一致。比如用 Java 声明一个包含 5 个元素, 每个元素都是数组的方式为 `int[5][]`。

要访问第 3 个动态数组的第 2 个元素, 使用 `x[2][1]` 即可。数组的序号是从 0 开始的, 访问数组使用的序号顺序与定义相反, 数组访问的方式和非区块链语言一致。

4. 数组的成员

数组的成员有 `length`、`push()`、`push(x)`、`pop()`。

5.1 length 属性

数组有一个 `length` 的成员属性, 表示当前的数组长度。对于存储在 `storage` 的变长数组, 可以通过给 `length` 赋值调整数组长度。memory 数组的长度在创建后是固定的。

注意: 不能通过访问超出当前数组长度的方式, 来自动实现改变数组长度。存储在 memory 的数组虽然可以通过参数灵活指定长度, 但一旦创建, 长度便不可调整。

```
uint[] memory a = new uint[](7);
```

```
a.length=8 //这是不允许的
```

```
uint[] storage sa;
```

```
sa.push(32); //只有 storage 可以 push, 因为他的大小不固定, 而 memory 中一旦声明就固定下来了
```

5.2 push 方法

存储在 storage 的变长数组有 push 成员方法，该方法用来向数组末尾添加新数据，返回值为添加元素后数组的长度。

注意：存储在 memory 的数组不支持 push 方法。

5.3 push(x)方法

变长数组拥有 push(x)成员，可以在数组最后添加一个 x 元素。

5.4 pop()方法

变长数组拥有 pop()成员，可以移除数组最后一个元素。



数组分析

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.5;

contract ArrayTypes {

    uint[8] array1;

    bytes1[5] array2;

    address[100] array3;

    uint[] array4;

    bytes1[] array5;

    address[] array6;

    bytes array7;

    uint[] array8 = new uint[](5);

    bytes array9 = new bytes(9);

    function initArray() public pure returns(uint[] memory){

        uint[] memory x = new uint[](3);

        x[0] = 1;

        x[1] = 3;

        x[2] = 4;

        return(x);

    }

    function arrayPush() public returns(uint[] memory){

        uint[2] memory a = [uint(1),2];

        array4 = a;

        array4.push(3);

        return array4;

    }

}
```

代码解析：

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract ArrayTypes{
```

定义了一个名为 ArrayTypes 的合约。contract 为合约声明关键字，ArrayTypes 为自定义合约名称。

```
uint[8] array1;  
bytes1[5] array2;  
address[100] array3;
```

声明 uint、bytes1、address 类型的固定长度的数组，声明时已经固定数组长度，array1~ array3 表示数组名称。

```
uint[] array4;  
bytes1[] array5;  
address[] array6;  
bytes array7;
```

声明 uint、bytes1、address、bytes 类型的变长数组，未固定数组长度，array4~array7 表示数组名称。

```
uint[] array8 = new uint[](5);
```

```
bytes array9 = new bytes(9);
```

初始化变长数组 array8、array9，通过 new 关键字来初始化一个数组，但是必须声明长度，并且声明后长度不能改变，声明 array8 的长度为 5，array9 的长度为 9。

```
function initArray() public pure returns(uint[] memory){
```

声明一个函数返回 uint[] 类型，返回数组类型时需要指明存储位置为 memory。

```
uint[] memory x = new uint[](3);
```

使用 new 声明并初始化变长数组 x，指定长度为 3，在函数中需要指定存储位置 memory。

```
x[0] = 1;
```

```
x[1] = 3;
```

```
x[2] = 4;
```

变长数组需要一个一个元素的为其赋值。

```
return(x);
```

返回 x 的值 1,3,4。

```
function arrayPush() public returns(uint[] memory){
```

声明一个函数返回 uint[] 类型，返回数组类型时需要指明存储位置为 memory。

```
uint[2] memory a = [uint(1),2];
```

通过数组字面量创建一个长度为 2 的 uint 类型的定长数组 a 并赋值，声明第一个元素

的类型是 uint256，不然默认是 uint8。

```
array4 = a;
```

将 a 赋值给变长数组 array4，此时 array4 的值为 1,2。

```
array4.push(3);
```

在数组 array4 最后添加一个元素 3。数组 array4 是存储在 storage 的变长数组，有成员 push。

```
return array4;
```

返回 array4 的值 1,2,3。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。

5. 变长字节数组

变长字节数组导入

1. 变长字节数组

数组的大小既可以有固定的，也可以实现动态改变。在不同的场景下要合理选择需要的类型。**bytes 和 string 是一种特殊的数组**。bytes 类似于 bytes[]，但在外部函数作为参数调用中，会进行压缩打包，更省空间，**所以应该尽量使用 bytes 而不是 bytes[]**。

bytes1~bytes32 表示创建固定字节大小的数组，不可修改。在不确定字节数据大小的情况下，通常可以使用 string 或者 bytes。如果预先已经了解了字符数组，可以通过 bytes1~bytes32 表示，应该避免使用 string 或 bytes，应使用 bytes1~bytes32，降低存储成本。

bytes 和 string 都可以用来表达字符串，他们的区别是 **bytes 用来存储任意长度的字节数据，string 用来存储任意长度（UTF-8 编码）的字符串数据**。

2. bytes 和 string 的成员

2.1 length 属性

bytes 拥有 length 属性，string 可以转换为 bytes 以通过 length 获得它的字节长度。

2.2 push 方法

存储在 storage 的 bytes 有 push 成员方法，string 没有 push 方法。

2.3 push(x)方法

bytes 拥有 push(x)成员。string 可以通过索引来修改对应的字节内容，通过 push 方法来增加字节内容。

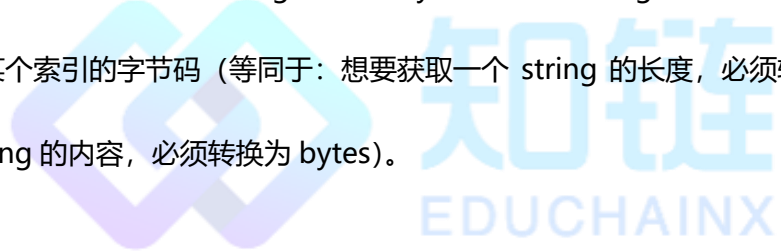
2.4 pop()方法

bytes 拥有 pop()成员，可以移除数组最后一个元素。

3. 常规字符串 string 转换为 bytes

可以将字符串 string 通过 bytes(s)这种显示转换的方式转为一个 bytes，通过 bytes(s).length 获取长度，bytes(s)[n]获取对应的 UTF-8 编码。通过下标访问获取到的不是对应字符串，而是 UTF-8 编码，比如中文的编码是变长的多字节，因此通过下标访问中文字符串得到的只是其中一部分编码。

string 字符串中没有提供 length 方法获取字符串长度，也没有提供方法修改某个索引的字节码，不过我们可以将 string 转换为 bytes，再调用 length 方法获取字节长度，也可以修改某个索引的字节码（等同于：想要获取一个 string 的长度，必须转为 bytes；想要修改 string 的内容，必须转换为 bytes）。



变长字节数组分析

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.5;

contract ElongateArray {

    string name1="a!+&500";

    bytes[] name2=new bytes[](1);

    function name() public view returns(bytes memory){

        return bytes(name1);

    }

    function namelength() public view returns(uint,uint){

        return (bytes(name1).length,name2.length);

    }

}
```

代码解析:

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract ElongateArray{
```

定义了一个名为 ElongateArray 的合约。contract 为合约声明关键字，ElongateArray 为自定义合约名称。

```
    string name1="a!+&500";
```

```
bytes[] name2=new bytes[](1);
```

声明 string、bytes 类型的变长数组。 name1 赋值为 a!+&500, name2 规定数组长度为 1。

```
function name() public view returns(bytes memory){  
    return bytes(name1);  
}
```

声明一个函数用于返回 name1 的值, name 为 string 类型, 可以通过 bytes(s)获取对应的 UTF-8 编码。返回值为 0x61212b26353030。

```
function namelength() public view returns(uint,uint){  
    return (bytes(name1).length,name2.length);  
}
```

声明一个函数用于返回两个字符串的长度, name1 是 string 类型没有 length 属性, 需要转换成拥有 length 属性的 bytes 类型用以读取长度, name2 是 bytes 类型则可以直接读取长度。返回值为 7, 1。

```
}
```

大括号是一组语句的组合, {}一一对应, 成对出现, 大括号的作用是将多条语句合成一个复合语句, 最后一个大括号表示程序结束。

6. 结构体

结构体导入

1. 结构体

结构体是一种数据类型，该数据类型由一组称为成员的不同数据组成，其中每个成员可以具有不同的类型。结构体类型不是由系统定义好的，而是需要程序设计者自己定义的。

结构体通常用来表示类型不同但是又相关的若干数据。结构体和其他类型基础数据类型一样，例如 int 类型、bool 类型，只不过结构体可以做成你想要的数据类型。以方便日后的使用。

类似于 C 语言，Solidity 使用 **struct** 关键字来实现类型的自定义，自定义的类型是引用类型。struct 除了可以使用基本类型作为成员以外，还可以使用数组、结构体、映射作为成员。

2. 结构体的声明及初始化

结构体的声明方式：

```
struct struct_name {  
    type1 type_name_1;  
    type2 type_name_2;  
    type3 type_name_3;  
}
```

例：

```
struct test1 {  
    bool memberBool;  
    uint memberUInt;
```

```
}
```

需要注意的是，不能在声明一个 struct 的同时将自身作为成员，因为结构体的大小必须是有限的。

```
struct test1 {  
    test1 t; // 不能将自身作为成员  
}
```

但声明的 struct 可以作为 mapping 的值类型成员。

```
struct test1 {  
    bool memberBool;  
    uint memberUInt;  
    mapping(string=>test1) testMapping;  
}
```

使用结构体声明变量及初始化的几种方式：

(1) 仅声明变量而不初始化，此时会使用默认值创建结构体变量。

```
test1 tt1; // 仅声明变量而不初始化
```

(2) 按成员循序（结构体声明时的顺序）初始化。这种方式需要特别注意参数的类型及数量的匹配。另外，如果结构体中有 mapping，则需要跳过对 mapping 的初始化。

```
// 按成员循序（结构体声明时的顺序）初始化  
test1 tt1 = test1(true, 2); // 只能作为状态变量这样使用  
test1 memory tt2 = test1(true, 2); // 在函数内声明  
test2 memory tt = test2(true, 2);
```

(3) 命名方式初始化。使用命名方式初始化可以不按定义的顺序初始化。参数的个数需要保持和定义时一致，如果有 mapping 类型，则同样需要忽略。

```
test1 memory tt = test1 ({memberBool:true, memberUInt:2}); // 命名方式初始化
```

3. 结构体的限制

结构体目前仅支持在合约内部使用或继承合约内使用，如果要在参数和返回值中使用结构体，函数必须声明 `internal`。目前，如果想在合约之间传递结构体，则必须对结构体成员进行拆解才行。

```
//合法
function interFun(test1 tt) internal {}

//非法
function exteFunc(test1 tt) public {}
```

4. 访问结构体成员

要访问结构的任何成员，使用成员访问操作符(`.`)。



结构体分析

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.5;

contract structure {

    struct Book {

        string title;

        string author;

        uint book_id;

    }

    Book book;

    function setBook() public {

        book = Book('LearnSolidity', 'ProgrammingLanguage', 1);

    }

    function getBookId() public view returns (uint) {

        return book.book_id;

    }

}
```

代码解析：

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract structure{
```


定义了一个名为 `structure` 的合约。`contract` 为合约声明关键字，`structure` 为自定义合约名称。

```
struct Book {
```

在合约中声明一个结构体 `Book`。

```
string title;  
string author;  
uint book_id;
```

结构体通常用来表示类型不同，所以可以在结构体中用不同的数据类型定义书的信息：
`string` 类型的名称（`title`）、`string` 类型的其他信息（`author`）、`uint` 类型的书号（`book_id`）。

```
Book book;
```

声明结构体变量 `book`。

```
function setBook() public {  
    book = Book('LearnSolidity', 'ProgrammingLanguage', 1);  
}
```

声明一个函数用于将结构体初始化，按成员循序（结构体声明时的顺序）初始化。

```
function getBookId() public view returns (uint) {  
    return book.book_id;  
}
```

声明一个函数访问结构体，属性 `book` 使用访问操作符.访问结构体中 `book_id` 的值，

使用 return 返回结构体的值。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。



7. 映射

映射导入

1. 映射

Solidity 映射类型和 Java 的 Map 或 Python 的 Dict 功能相似，是一种**键值对的映射关系存储结构**，定义方式中，**_KeyType** 可以是任何内置类型，或者 **bytes** 和字符串。不允许使用引用类型、合约和枚举。**_ValueType** 可以是任何类型。**mapping** 为映射关键字，**name** 为映射变量名。

```
mapping(_KeyType=>_KeyValue) name;
```

映射能够看做是哈希表，它们被虚拟地初始化，使得每一个可能的键都存在并被映射到其字节，映射类型的默认值为零。

映射是一种使用广泛的类型，经常在合约中充当一个类数据库的角色，比如在代币合约中用映射来存储账户的余额，在游戏合约里可以用映射来存储每个账号的级别：

```
mapping(address => uint) public balances;  
mapping(address => uint) public userLevel;
```

2. 映射的规则

规则 1：映射的 **_KeyType** 只能选择 Solidity 默认的类型，比如 **uint**，**address** 等，不能用自定义的结构体。而 **_ValueType** 可以使用自定义的类型。下面这个例子会报错，因为 **_KeyType** 使用了我们自定义的结构体：

```
// 我们定义一个结构体 Struct  
  
struct Student{  
  
    uint256 id;  
  
    uint256 score;
```

```
}  
  
mapping(Student => uint) public testVar;
```

规则 2：映射的存储位置必须是 storage，因此可以用于合约的状态变量、函数中的 storage 变量，也能定义在结构体中，但存储位置仍是 Strong。不能用于 public 函数的参数或返回结果中，因为 mapping 记录的是一种映射关系。

规则 3：如果映射声明为 public，那么 Solidity 会自动给你创建一个 getter 函数，可以通过 Key 来查询对应的 Value。

规则 4：给映射新增的键值对的语法为 `_Var[_Key] = _Value`，其中 `_Var` 是映射变量名，`_Key` 和 `_Value` 对应新增的键值对。例子：

```
function writeMap (uint _Key, address _Value) public {  
    idToAddress[_Key] = _Value;  
}
```



映射分析

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.5;

contract MappingTest {

    mapping(address => uint) public balances;

    function update(uint amount) public returns (address){

        balances[msg.sender] = amount;

        return msg.sender;

    }

}
```

代码解析:

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract MappingTest{
```

定义了一个名为 MappingTest 的合约。contract 为合约声明关键字，MappingTest 为自定义合约名称。

```
mapping(address => uint) public balances;
```

声明一个名为 balances 的映射，可见性为 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。

```
function update(uint amount) public returns (address){
```

声明一个函数，输入整型参数 `amount`，返回地址类型的值。

```
balances[msg.sender] = amount;
```

`msg.sender` 是合约的账户地址，本语句表示将参数 `amount` 的值和 `msg.sender` 这个地址对应起来。

```
return msg.sender;
```

当输入 `amount` 的值，检测此合约地址，当输入的是本合约地址，返回 `amount` 的值。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。

第二章 变量

1. Solidity 中的变量

Solidity 中的变量

1. Solidity 中的变量

Solidity 是一种静态类型语言，这意味着需要在声明期间指定变量类型。每个变量声明时，都有一个基于其类型的默认值。没有 undefined 或 null 的概念。

Solidity 中支持的三种变量类型：

- **状态变量** – 变量值永久保存在合约存储空间中的变量。
- **局部变量** – 变量值仅在函数执行过程中有效的变量，函数退出后，变量无效。
- **全局变量** – 保存在全局命名空间，用于获取区块链相关信息的特殊变量，提供有关区块链和交易属性的信息。

```
contract Example {  
  
    uint storedXlbData; // 状态变量  
  
    function getResult() public view returns(uint){  
  
        uint storedXlbData; // 局部变量  
  
        storedXlbData= msg.value; // 全局变量  
  
    }  
  
}
```

2. 命名规则

变量是自命名的，但在为变量命名时，请记住以下规则：

- **Solidity 不能使用保留关键字作为变量名**

例如，break 或 boolean 变量名无效。

- **不应以数字(0-9)开头，必须以字母或下划线开头**

例如，123test 是一个无效的变量名，但是_123test 是一个有效的变量名。

- **变量名区分大小写**

例如，Name 和 name 是两个不同的变量。

注： 习惯上函数里的变量都是以(_)开头 (但不是硬性规定) 以区别全局变量。在函数名字后面使用关键字 private 即可，私有函数的名字用()下划线起始，而公有函数不需要下划线开头，这属于命名规范，我们可以通过这些区分合约中那些函数是私有的。

3. 变量的默认值

变量在声明后会用全零字节作为默认值，也就是所有类型的默认值都是零态，比如 bool 的默认值是 false，uint 和 int 默认值是 0，对于变长的数组 bytes 和 string，默认值则为空数组和空字符串。

2. 状态变量

状态变量导入

1. 状态变量的声明

状态变量声明在合约中，函数外。

```
contract Example { // 合约

    uint public storedXlbData; // 状态变量

    function test() public {} // 函数
}
```

2. 状态变量的存储范围

状态变量是被永久地保存在合约中的，也就是说它们被写入以太坊区块链中。状态变量是数据存储在链上的变量，所有合约内函数都可以访问，gas 消耗高。

```
contract Example {

    // 这个无符号整数将会永久的被保存在区块链中

    uint public myUnsignedInteger = 100; // 定义 myUnsignedInteger 为 uint 类型，并赋值 100
}
```

3. 状态变量的作用域

状态变量可以在声明时进行初始化，并且具有以下可见性：

- **private**：状态变量仅在定义的合约里可见。
- **public**：状态变量也可以在定义合约的外部访问，因为编译器会自动创建一个与该变量同名的 getter 函数。
- **internal**：状态变量在定义的合约以及所有继承合约都是可见的。

可见性指示符放在状态变量的类型之后，**如果未指定，则状态变量将被视为 internal。**

```
contract Example {  
    uint public mytest1 = 100;  
    uint internal mytest2 = 100;  
    uint private mytest2 = 100;  
}
```



状态变量分析

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.5;

contract StateVariable {

    uint public stateVar1 = 10;

    uint stateVar2 = 20;

    function doSomething() public view returns (uint) {

        return stateVar2;

    }

}
```

代码解析:

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract StateVariable{
```

定义了一个名为 StateVariable 的合约。contract 为合约声明关键字，StateVariable 为自定义合约名称。

```
uint public stateVar1 = 10;
```

声明一个公开可见的状态变量 stateVar1，赋值为 10。

```
uint stateVar2 = 20;
```

声明一个状态变量 `stateVar2`，赋值为 20，未指定可见性，状态变量被视为 `internal` 可见性。

```
function doSomething() public view returns (uint) {
```

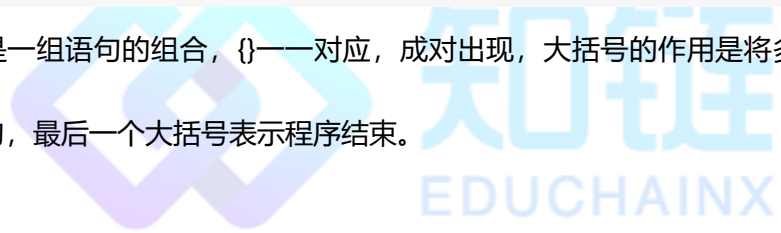
声明一个函数用于返回值。

```
return stateVar2;
```

返回 `stateVar2` 的值为 20。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。



3. 局部变量

局部变量导入

1. 局部变量的声明

变量值仅在函数执行过程中有效的变量为局部变量，局部变量声明在函数内。

```
contract Example {  
  
    function getResult() public view returns(uint){  
  
        uint storedXlbData; // 局部变量  
  
    }  
  
}
```

2. 局部变量的存储范围

局部变量是被声明在函数内部的变量，函数退出后，变量无效，不会写入到区块链中。

局部变量的数据存储在内存里，不上链，gas 低。

```
contract Example {  
  
    function returnuint() public pure returns(uint){  
  
        uint a=5; // 局部变量  
  
    }  
  
}
```

3. 局部变量的作用域

局部变量的作用域仅限于定义它们的函数。

局部变量分析

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.5;

contract LocalVariable{

    function returnuint() public pure returns(uint){

        uint a =5;

        return a;

    }

}
```

代码解析：

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract LocalVariable{
```

定义了一个名为 LocalVariable 的合约。contract 为合约声明关键字，LocalVariable 为自定义合约名称。

```
function returnuint () public pure returns (uint) {
```

声明一个函数，局部变量声明在函数内。

```
uint a =5;
```

声明一个局部变量 a，赋值为 5。

```
return a;
```

返回 a 的值为 5。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。



4. 全局变量

全局变量导入

1. 全局变量

全局变量保存在全局命名空间，用于**获取区块链相关信息的特殊变量**，提供有关区块链和交易属性的信息。

全局变量是**全局范围工作的变量**，都是 **Solidity 预留关键字**。他们可以在函数内不声明直接使用。

2. 全局变量的使用

```
address sender = msg.sender;
```

在上面例子里，我们使用了全局变量：`msg.sender` 他们代表请求发起地址，下面是 Solidity 中的全局变量：

<code>block.basefee (uint)</code>	当前区块的基础费用
<code>block.coninbase (address)</code>	当前块矿工的地址，括号中表示返回值的类型
<code>block.chainid (uint)</code>	当前链 id
<code>block.difficulty (uint)</code>	当前块的难度
<code>block.gaslimit (uint)</code>	当前块的 gaslimit
<code>block.number (uint)</code>	当前区块的块号
<code>block.timestamp (uint)</code>	当前块 Unix 时间戳(从 1970/1/1 00:00:00 UTC 开始所经过的秒数)
<code>msg.sig (bytes4)</code>	调用数据(calldata)的前四个字节
<code>msg.data (bytes)</code>	完整地调用数据 (calldata)

msg.sender (address)	当前调用发起人的地址
msg.value (uint)	这个消息所附带的以太币，单位为 wei
tx.gasprice (uint)	交易的 gas 价格
tx.origin (address)	交易的发送者（全链调用）



全局变量分析

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.5;

contract GlobalVariable{

    address public coinbase;

    function t () payable public {

        coinbase = block.coinbase ;

    }

}
```

代码解析:

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract GlobalVariable{
```

定义了一个名为 GlobalVariable 的合约。contract 为合约声明关键字，GlobalVariable 为自定义合约名称。

```
address public coinbase;
```

声明一个公开可见的地址类型的变量。

```
function t () payable public {
```

声明一个函数，使用 payable 关键字说明和交易有关，函数中应用全局变量，全局变量 block.coinbase 用于获取当前块矿工的地址，和交易属性相关。

```
coinbase = block.coinbase ;
```

获得当前块矿工的地址，block.coinbase 获得的值为 address 类型，所以 coinbase 也需要是 address 类型。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。



第三章 函数

1. 函数参数

函数参数导入

1. 函数的声明

Solidity 中，函数写在合约内部，一个合约中可以有多个函数，函数由关键字

function 声明，后面跟函数名、函数参数、可见性、返回值的定义。定义函数的语法如下：

```
//函数修饰词、returns（函数参数）可省略  
//参数类型 参数名 可省略  
function 函数名称（参数类型 参数名称） 可见性 函数修饰词 returns（返回参数类型 返回  
参数名称） {  
    //函数体  
}
```

```
function getBrand() public view returns (string) {  
    return brand;  
}
```

2. 函数参数

与 javascript 一样，函数可以提供参数作为输入。与 JavaScript 和 C 不同的是，

Solidity 还可以返回任意数量的参数做为输出。

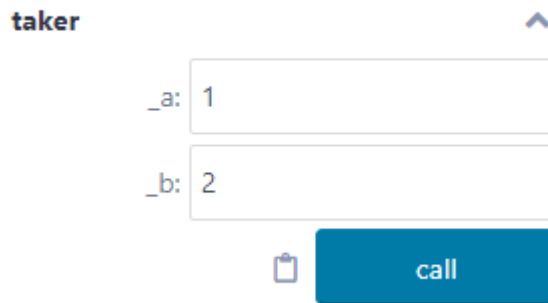
2.1 输入参数

输入参数的声明方式与变量相同，**未使用的参数可以省略变量名称。**

假设希望合约接受一种带有两个整数参数的外部调用：

```
function taker(uint _a,uint _b) public pure{  
    //使用_a,_b  
}
```

输入：



2.2 输出参数

在 `returns` 关键字之后，可以用与输入参数相同的语法声明输出参数，输出参数的变量名可以省略。输出值可以使用 `return` 语句指定，返回参数初始化为 0，如果未显式设置，它们会保持为 0。与参数类型相反，返回类型不能为空 —— 如果函数类型不需要返回，则需要删除整个 `returns (<return types>)` 部分。

假设我们希望输出两个结果，即两个给定整数的和、积，则可以这样写：

```
function arithmetics(uint _a,uint _b) public pure returns(uint,uint){  
    uint o_sum=_a+_b;  
    uint o_product=_a*_b;  
}
```

输出：

arithmetics

_a: 1

_b: 2



call

0: uint256: 0

1: uint256: 0

当函数需要使用多个值，参数的数量需要和声明时候一致。

```
return (v0, v1, ..., vn)
```

假设我们希望显式的返回两个结果，即两个给定整数的和、积，则可以这样写：

```
function arithmetics(uint _a,uint _b) public pure returns(uint,uint){  
    uint o_sum=_a+_b;  
    uint o_product=_a*_b;  
    return (o_sum,o_product);  
}
```

输出：

arithmetics

_a: 5

_b: 2



call

0: uint256: o_sum 7

1: uint256: o_product 10

函数参数分析

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.7;

contract Param{

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

        uint o_sum=_a+_b;

        uint o_product=_a*_b;

        return (o_sum,o_product);

    }

}
```

代码解析:

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract Param{
```

定义了一个名为 Param 的合约。contract 为合约声明关键字，Param 为自定义合约名称。

```
function arithmetics(uint _a,uint _b) public pure returns(uint,uint){
```

在函数内部声明的 uint 类型的参数_a、_b 是输入参数，返回值内定义了返回两个值，参数类型是 uint。

```
uint o_sum=_a+_b;  
uint o_product=_a*_b;
```

计算出两个结果，即两个给定整数的和、积。

```
return (o_sum,o_product);
```

显式的返回两个结果，函数需要使用多个值，参数的数量需要和声明时候一致。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。



2. 函数可见性

函数可见性导入

1. 函数可见性

Solidity 对函数和状态变量提供了四种可见性：分别是 **external**、**public**、**internal**、**private**。其中函数默认的可见性是 **public**；状态变量默认的可见性是 **internal**，状态变量不能设置为 **external**。

函数可见性中的 **external**、**internal** 与函数调用中的 **external**、**internal** 是不同，一个代表可见性，一个代表调用方式。

当使用 **public** 函数时，Solidity 会立即复制数组参数到内存，而 **external** 函数则是从 **calldata** 读取的，分配内存的开销比直接从 **calldata** 读取的开销要大。**public** 函数要复制数组参数到内存的原因是：**public** 函数可能会被内部调用，而内部调用数组的参数是当成指向一块内存的指针。同时，**external** 函数不允许内部调用，它直接从 **calldata** 读取数据，省去了复制的过程。所以，如果一个函数需要在外部访问，使用 **external** 函数。

类似的，通过 **this.f()** 模式调用，会有一个花费很大开销的 **CALL** 调用，并且它传参的方法也比内部传递的开销更大。因此，**当需要内部调用的时候，建议使用 public 函数。**

- **private** 意味着它只能被合约内部调用；
- **internal** 就像 **private** 但是也能被继承的合约调用；
- **external** 只能从合约外部调用；
- **public** 可以在任何地方调用，不管是内部还是外部。

2. external

外部函数是合约接口的一部分，所以我们可以从其他合约或通过交易来发起调用。而一个外部函数 `f` 不能通过内部的方式来发起调用。外部函数在接收大的数组数据时更加有效。

3. public

公共状态变量可以在内部访问，也可以通过外部访问。对于公共状态变量，将生成一个自动 `getter` 函数。

合约中的函数本身默认是 `public` 的，只有当它被当做类型名称时，默认才是内部函数，声明为 `public` 的函数允许以外部调用和内部调用两种方式进行调用。

4. internal

内部状态变量只能从当前合约或其派生合约内访问。需要注意的是访问时不能加前缀 `this`，前缀 `this` 表示的是通过外部方式访问。

声明为 `internal` 的函数在定义合约和子合约中均只能以 `internal` 的方式进行调用。

5. private

私有函数和状态变量仅在当前合约中可以访问，在继承的合约内不可以访问。所有在合约内的东西对外部观察者来说都是可见的，将某些标记为 `private` 仅仅阻止了其他合约来进行访问和修改，但并不能阻止其他人看到相关的信息。

声明为 `private` 的函数只能在定义的合约中以 `internal` 的方式进行调用。

函数可见性分析

1. 函数可见性——external

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.5;

contract ExternalTest{

    function externalFunc() external pure returns(uint a){

        return 10;

    }

    function callFunc() public view returns(uint){

        //1. 以 internal 的方式调用报错

        //return externalFunc();

        //2. 以 external 的方式调用函数

        return this.externalFunc();

    }

}
```

代码解析：

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract ExternalTest{
```

定义了一个名为 ExternalTest 的合约。contract 为合约声明关键字，ExternalTest 为自定义合约名称。

```
function externalFunc() external pure returns(uint a){  
    return 10;  
}
```

声明一个外部函数 `externalFunc()`，定义一个参数 `a`，返回 `a` 的值。

```
function callFunc() public view returns(uint){
```

声明一个函数用于调用。

```
//return externalFunc();
```

外部函数不能以 `internal` 的方式调用。

```
return this.externalFunc();
```

以 `external` 的方式调用外部函数。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。

2. 函数可见性——public

```
// SPDX-License-Identifier: GPL-3.0  
  
pragma solidity ^0.8.5;  
  
contract ExternalTest{  
    uint public test=20;  
  
    function publicFunc() public pure returns(uint){
```

```

        return 10;
    }

    function callFunc1() public pure returns(uint){
        //1. 以 internal 的方式调用函数
        return publicFunc();
    }

    function callFunc2() public view returns(uint){
        //2. 以 external 的方式调用函数
        return this.publicFunc();
    }

    function callFunc3() public view returns(uint){
        //3. 以 internal 的方式调用状态变量
        return test;
    }

    function callFunc4() public view returns(uint){
        //4. 以 external 的方式调用状态变量
        return this.test();
    }
}

```

代码解析：

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract ExternalTest{
```

定义了一个名为 ExternalTest 的合约。contract 为合约声明关键字，ExternalTest 为自定义合约名称。

```
uint public test=20;
```

声明一个公开可见的状态变量并赋值为 20，状态变量默认是 internal。

```
function publicFunc() public pure returns(uint){  
    return 10;  
}
```

声明一个公开函数 publicFunc()，返回 uint 类型的 10，合约中的函数默认是 public，但不可以省略可见性。

```
function callFunc1() public pure returns(uint){  
    return publicFunc();  
}
```

声明一个公开函数 callFunc1，以 internal 的方式调用函数 publicFunc()，可以看到声明为 public 的 publicFunc()允许 internal 调用。

```
function callFunc2() public view returns(uint){  
    return this.publicFunc();  
}
```

声明一个公开函数 callFunc2，以 external 的方式调用函数 publicFunc()，可以看到声明为 public 的 publicFunc()允许 external 调用。

```
function callFunc3() public view returns(uint){
```

```
        return test;
    }
```

声明一个公开函数 `callFunc2`，以 `internal` 的方式调用状态变量 `test`，可以看到声明为 `public` 的 `test` 允许 `internal` 调用。

```
function callFunc4() public view returns(uint){
    return this.test();
}
```

声明一个公开函数 `callFunc4`，以 `external` 的方式调用状态变量 `test`，可以看到声明为 `public` 的 `test` 允许 `external` 调用。

```
}
```

大括号是一组语句的组合，`{}`——对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。

3. 函数可见性——`internal`

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.5;
contract ExternalTest{
    uint internal test=20;

    function InternalFunc() internal pure returns(uint){
        return 10;
    }

    function callFunc1() public pure returns(uint){
        //以 internal 的方式调用函数
        return InternalFunc();
    }
}
```

```

    }

    function callFunc2() public view returns(uint){

        //以 internal 的方式调用状态变量

        return test;

    }
}

//派生合约
contract ExternalCall is ExternalTest{

    function callFunc3() public pure returns(uint){

        //以 internal 的方式调用函数

        return InternalFunc();

    }

    function callFunc4() public view returns(uint){

        //以 internal 的方式调用状态变量

        return test;

    }

}

```

代码解析:

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract ExternalTest{
```

定义了一个名为 ExternalTest 的合约。contract 为合约声明关键字，ExternalTest 为

自定义合约名称。

```
uint internal test=20;
```

声明一个可见性为 internal 的状态变量并赋值为 20。

```
function InternalFunc() internal pure returns(uint){  
    return 10;  
}
```

声明一个可见性为 internal 的函数 InternalFunc(), 返回 uint 类型的 10。

```
function callFunc1() public pure returns(uint){  
    return InternalFunc();  
}
```

声明一个公开函数 callFunc1, 以 internal 的方式调用函数 InternalFunc(), 声明为 internal 的 InternalFunc()在定义合约和派生合约中均只能以 internal 的方式可以进行调用。

```
function callFunc2() public view returns(uint){  
    return test;  
}
```

声明一个公开函数 callFunc2, 以 internal 的方式调用状态变量 test, 内部状态变量只能在当前合约和派生合约中以 internal 的方式访问。

```
contract ExternalCall is ExternalTest{
```

声明一个合约 ExternalCall 继承自 ExternalTest, ExternalCall 是 ExternalTest 的派

生合约。

```
function callFunc3() public pure returns(uint){  
    return InternalFunc();  
}
```

声明一个公开函数 callFunc3，以 internal 的方式调用函数 InternalFunc()，声明为 internal 的 InternalFunc()在定义合约和派生合约中均只能以 internal 的方式可以进行调用。

```
function callFunc4() public view returns(uint){  
    return test;  
}
```

声明一个公开函数 callFunc4，以 internal 的方式调用状态变量 test，内部状态变量只能在当前合约和派生合约中以 internal 的方式访问。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。

4. 函数可见性——private

```
// SPDX-License-Identifier: GPL-3.0  
  
pragma solidity ^0.8.5;  
  
contract ExternalTest{  
    uint private test=20;  
  
    function privateFunc() private pure returns(uint){
```

```

        return 10;
    }

    function callFunc1() public pure returns(uint){
        //以 internal 的方式调用函数
        return privateFunc();
    }

    function callFunc2() public view returns(uint){
        //以 internal 的方式调用状态变量
        return test;
    }
}

```

代码解析:

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract ExternalTest{
```

定义了一个名为 ExternalTest 的合约。contract 为合约声明关键字，ExternalTest 为自定义合约名称。

```
uint private test=20;
```

声明一个可见性为 private 的状态变量并赋值为 20。

```
function privateFunc() private pure returns(uint){  
    return 10;  
}
```

声明一个可见性为 `private` 的函数 `privateFunc ()`，返回 `uint` 类型的 10。

```
function callFunc1() public pure returns(uint){  
    return privateFunc();  
}
```

声明一个公开函数 `callFunc1`，以 `internal` 的方式调用函数 `privateFunc ()`，声明为 `private` 的函数仅在当前合约中以 `internal` 的方式可以访问。

```
function callFunc2() public view returns(uint){  
    return test;  
}
```

声明一个公开函数 `callFunc2`，以 `internal` 的方式调用状态变量 `test`，声明为 `private` 的状态变量仅在当前合约中以 `internal` 的方式可以访问。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。

3. view 函数

view 函数导入

在 Solidity 中，如果一个函数不修改合约状态，它只是读取区块链的状态而不改变它，那么可以将该函数应该声明为 view（视图）函数，使用 view 函数不消耗 gas。若有函数有以下操作，则不能用 view 声明：

- 修改状态变量
- 产生事件
- 创建其它合约
- 使用 selfdestruct
- 通过调用发送以太币
- 调用任何没有标记为 view 或者 pure 的函数
- 使用底层调用
- 使用包含特定操作码的内联汇编

```
function f() public view returns (uint) {  
}
```

注意：

1. 在 Solidity 0.5.0 版本以前，constant 作为 view 的别名存在，使用 constant 和 view 效果等价，不过在更新的版本中，该特性已被删除，因为 constant 作为状态常量的关键字经常会被混淆。

2. 编译器为 public 的变量创建的访问器函数都是 view 类型的。

3. 编译器中 Solidity 低版本虽然没有强制 view 方法不能修改状态。虽然可以通过编

译，但这种方式是错误做法。

view 函数分析

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.5;

contract ViewTest{

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

        return a * (b + 42) + block.timestamp;

    }

}
```

代码解析：

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract ViewTest{
```

定义一个名为 ViewTest 的合约。

```
function f(uint a, uint b) public view returns (uint) {
```

声明函数 f(), 函数修饰词为 view, view 表示读取区块链的状态而不改变它。

```
return a * (b + 42) + block.timestamp;
```

这里 a 与 b 是保存在 memory 中的，不存在读取或改变区块链状态的行为，block.timestamp 是全局变量，全局变量的作用是保存区块（或链）的信息，因此只是读取区块链的状态而不改变它的行为，所以函数修饰词使用 view。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。



4. pure 函数

pure 函数导入

Solidity 语言有两类和状态读写有关的函数类型，一类是 view 函数，另一类是 pure (纯) 函数。他们的区别是 **view 函数不修改状态，pure 函数不修改状态也不读取状态。**和 view 函数一样，使用 pure 函数也不消耗 gas。如果函数中存在以下语句，则被视为读取状态：

- 读取状态变量。
- 访问 address(this).balance 或者 <address>.balance。
- 访问 block, tx, msg 中任意成员 (除 msg.sig 和 msg.data 之外)。
- 调用任何未标记为 pure 的函数。
- 使用包含某些操作码的内联汇编。

```
function f() public pure returns (uint) {  
}
```

注意：

1. 同 view 函数一样编译器没有强制 pure 方法不能读取状态。尽管 view 和 pure 修饰符编译器并未强制要求使用，view 和 pure 修饰也不会带来 gas 消耗的改变，但是更好的编码习惯有助于发现智能合约中的错误。

2. 如果发生错误，pure 函数可以使用 revert()和 require()函数来还原潜在的状态更改。

pure 函数分析

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.5;

contract PureTest{

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

        return a * (b + 42);

    }

}
```

代码解析：

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract PureTest{
```

定义一个名为 PureTest 的合约。

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

声明函数 f(), 函数修饰词为 pure, pure 表示既不读取也不修改区块链的状态。

```
return a * (b + 42);
```

这里 a 与 b 是保存在 memory 中的, 不存在读取或修改区块链状态的行为, 所以函数修饰词使用 pure。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。



5. 构造函数

构造函数导入

1. 构造函数

构造函数是特殊的函数，在部署合约的时候，就会被调用。而且只能在此时被调用。常常用于对于某一些状态变量的初始化。

在 Solidity 0.4.22 之前，构造函数是和合约名字同名的：

```
contract test{  
    function test(){}  
}
```

由于这种旧写法容易使开发者在书写时发生疏漏（例如合约名叫 Parents，构造函数名写成 parents），使得构造函数变成普通函数，引发漏洞，所以 0.4.22 版本及之后，采用了全新的 constructor 写法，使用关键词 constructor 作为构造函数：

```
contract test{  
    constructor(){}  
}
```

构造函数不需要指明可见性，构造函数的可见性只可以是 public，但不需要指明。

2. 构造函数的特性

构造函数有以下重要特性：

- (1) 一个合约只能有一个构造函数。
- (2) 构造函数在创建合约时执行一次，用于初始化合约状态。
- (3) 在执行构造函数之后，合约最终代码被部署到区块链。合约最终代码包括公共函数和可通过公共函数访问的代码。构造函数代码或仅由构造函数使用的任何内部方法不包括

在最终代码中。

(4) 构造函数可以是公共的，也可以是内部的。

(5) 内部构造函数将合约标记为抽象合约。

(6) 如果没有定义构造函数，则使用默认构造函数。

```
contract Test {  
  
    constructor() public {}//默认构造函数  
  
}
```

(7) 如果基合约具有带参数的构造函数，则每个派生/继承的合约也都必须包含参数。

```
contract Base {  
  
    uint data;  
  
    constructor(uint _data) public {  
  
        data = _data;  
  
    }  
  
}  
  
contract Derived is Base (5) {  
  
    constructor() public {}  
  
}
```

(8) 如果派生合约没有将参数传递给基合约构造函数，则派生合约将成为抽象合约。

```
contract Base {  
  
    uint public a;  
  
    address public owner;  
  
    constructor(uint _a){  
  
        a = _a;  
  
    }  
  
}  
  
abstract contract Derived3 is Base {  
  
    constructor(){  
  
        owner = msg.sender;  
  
    }  
  
}
```

```
}  
  
}
```

构造函数分析

```
// SPDX-License-Identifier: GPL-3.0  
  
pragma solidity ^0.8.5;  
  
contract MyContract{  
    string str;  
    constructor() {  
        str = "Hello world";  
    }  
    function getValue() public view returns (string memory) {  
        return str;  
    }  
}
```

代码解析：

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract MyContract{
```

定义一个名为 MyContract 的合约。

```
string str;
```

声明状态变量 str。

```
constructor() {  
    str = "Hello world";  
}
```

初始化状态变量 str 的值为 Hello world。

```
function getValue() public view returns (string memory) {  
    return str;  
}
```

返回 str 的值为 Hello world，说明合约在创建时，已为 str 赋值。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。

第四章 数据存储位置

1. 数据存储位置

数据存储位置导入

1. 数据存储位置

在介绍了值类型后，接下来介绍引用类型。在此之前，我们需要介绍一下数据存储位置。因为引用类型占用的空间通常超过 256 位，拷贝时开销很大，因此需要考虑将它们存储在什么位置。

在合约中声明和使用的变量都有一个数据位置，指明变量值应该存储在哪里。合约变量的数据位置将会影响 Gas 消耗量。在 Solidity 中，**有两个地方可以存储变量：存储 (Storage) 以及内存 (Memory)。**

Storage 变量是指永久存储在区块链中的变量。Memory 变量则是临时的，当外部函数对某合约调用完成时，内存型变量即被移除。内存(memory)位置还包含 2 种类型的存储数据位置，一种是 calldata，一种是栈 (stack)。

数据的存储位置非常重要，因为他们影响着赋值行为。**在 memory 和 storage 之间或与状态变量之间相互赋值，总是会创建一个完全独立的拷贝。而将一个 storage 的状态变量，赋值给一个 storage 的局部变量，则是通过引用传递完成的。**所以对于局部变量的修改，要同时修改关联的状态变量。另一方面，**将一个 memory 的引用类型赋值给另一个 memory 的引用，是不会创建拷贝的，即 memory 之间是通过引用传递完成的。**

1.2 Storage

该存储位置存储永久数据，这意味着该数据可以被合约中的所有函数访问。可以把它视为计算机的硬盘数据，所有数据都永久存储。保存在存储区(Storage)中的变量，以智

能合约的状态存储，并且在函数调用之间保持持久性。与其他数据位置相比，存储区数据位置的成本较高。

1.3 Memory

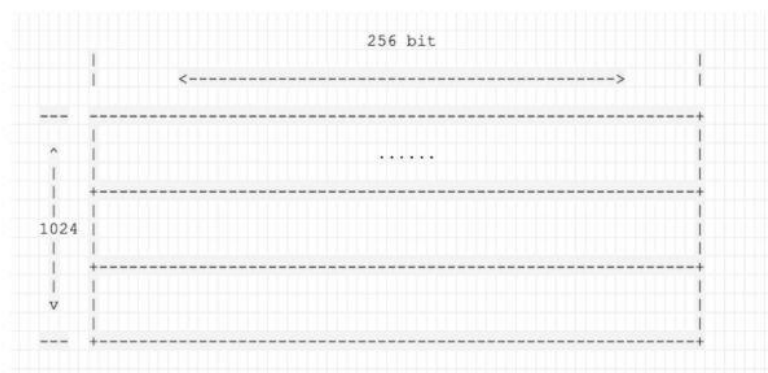
内存位置是临时数据，比存储位置便宜。它只能在函数中访问。通常，内存数据用于保存临时变量，以便在函数执行期间进行计算。一旦函数执行完毕，它的内容就会被丢弃。你可以把它想象成每个单独函数的内存(RAM)。不能将 memory 赋值给局部变量。对于值类型，总是会进行拷贝的。

1.4 Calldata

Calldata 是只读的、不可修改的非持久性数据位置，所有传递给函数的值，都存储在这里。此外，Calldata 是外部函数的参数(而不是返回参数)的默认位置。

1.5 Stack

堆栈是由 EVM 维护的非持久性数据。EVM 使用堆栈数据位置在执行期间加载变量。堆栈位置最多有 1024 个级别的限制，其中每个单元是 32 byte，值类型的局部变量是存储在栈上。stack 即所谓的“运行栈”，用来保存 EVM 指令的输入和输出数据。可以免费使用，没有 gas 消耗，数量被限制在 16 个。



2. 强制制定的数据位置

外部函数的参数（不包括返回参数）强制指定的数据位置为 `calldata`。状态变量强制指定的数据位置为 `storage`。

3. 默认数据位置

函数参数（包括返回的参数）默认存储位置是 `memory`。局部复杂类型变量（`local variables`）和状态变量存储位置是 `storage`。

`storage` 存储结构是在合约创建的时候就确定好，他取决于合约所声明的状态变量。但是内容可以被调用（交易）改变。

Solidity 称这个为状态改变，这也是合约级变量称为状态变量的原因。也可以简单理解为状态变量都是 `storage`。**`memory` 只能用于函数内部**，`memory` 声明用来告知 EVM 在运行时创建一块（固定大小）内存区域给变量使用。

`storage` 在区块链中是用 `key/value` 的形式存储的，而 `memory` 则表现为字节数组。

4. 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。

`memory` 仅保存临时变量，函数调用之后释放，开销很小。

`stack` 保存很小的局部变量，免费使用，但有数量限制(16 个变量)。

`calldata` 的数据包含消息体的数据，其计算需要增加 $n \times 68$ 的 GAS 费用。

数据存储位置分析

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.5;

contract Storagelocation{

    uint[] x;

    function f(uint[] memory Array) public{

        x=Array;

        uint[] storage y=x;

        y[7];

        g(x);

        h(x);

    }

    function g(uint[] storage storagArray) internal{}

    function h(uint[] memory Array) public{}

}
```

代码解析：

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract Storagelocation {
```

定义了一个名为 Storagelocation 的合约。contract 为合约声明关键字，Storagelocation 为自定义合约名称。

```
uint[] x;
```

状态变量强制为 storage。storage 存储结构是在合约创建的时候就确定好了的，它取决于合约所声明状态变量。x 的存储位置是 storage。uint[] 是一个数组，是引用类型。

```
function f(uint[] memory Array) public {
```

memory 只能用于函数内部，memory 声明用来告知 EVM 在运行时创建一块（固定大小）内存区域给变量使用。Array 的存储位置是 memory。

```
x=Array;  
    uint[] storage y=x;  
y[7];
```

状态变量的内容可以被调用改变。x=Array 表示从 memory 复制到 storage，uint[] storage y=x storage 表示引用传递局部变量 y (y 是一个 storage 引用)，y[7] 表示返回第 8 个元素。

```
g(x);  
h(x);
```

外部函数的参数强制为 calldata。g(x) 是引用传递，g 可以改变 x 的内容。h(x) 是拷贝到 memory，h 无法改变 x 的内容。

```
function g(uint[] storage Array) internal {}
```

复杂类型的局部变量：storage。

```
function h(uint[] memory Array) public {}
```

普通局部变量：memory。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。



第五章 错误及异常处理

1. 错误及异常处理

错误及异常处理导入

1. 错误处理的概念

错误处理是指在程序发生错误时的处理方式。**Solidity 处理错误与大多数高级编程语言不同，绝大部分语言通过捕获异常来处理错误，而 Solidity 通过回退状态方式来处理错误，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。**

Solidity 这样处理错误的原因与区块链的特性有关。可以把区块链理解为全球共享的分布式事务性数据库。全球共享意味着参与这个网络的每一个人都可以读写其中的记录。如果想修改这个数据库中的内容，就必须创建一个事务。**该事务必须要满足原子性，即要么全做要么全不做。Solidity 错误处理就是要保证每次调用事务的原子性。**

2. 错误处理的方式

Solidity 提供了 assert 和 require 两个函数来进行条件检查。assert 函数通常用来检查（测试）内部错误，而 require 函数用来检查输入变量或合约状态变量是否满足条件，以及验证调用外部合约的返回值。另外，如果正确使用 assert 函数，一些 Solidity 分析工具（如 SMTChecker）可以帮我们分析出智能合约中的错误。

除了可以用 assert 和 require 两个函数来进行条件检查以外，还有两种方式可以触发异常：

(1) revert 函数可以用来标记错误并回退当前调用，当前剩余的 gas 会返回给调用

者。

(2) 使用 `throw` 关键字抛出异常（从 0.4.13 版本，`throw` 关键字已被弃用）回滚所有状态改变，返回“无效操作代码错误”，而且消耗掉剩下的 `gas`。

例如，此 `if` 语句在功能上等价以下语句：

```
if(msg.sender != owner){  
    throw;  
}  
//if 语句等价于以下两个语句  
assert(msg.sender == owner);  
require(msg.sender == owner);
```

当调用中发生异常时，异常会自动向上“冒泡”。不过，也有例外的情况。例如，`send` 和底层函数调用 `call`、`delegatecall`、`callcode`，当发生异常时，这些函数会返回 `false`。需要注意的是，在一个不存在的地址上调用底层函数 `call`、`delegatecall`、`callcode`，会成功返回，所以在进行调试时需优先进行函数存在性检查。

3. `assert` 与 `require` 类型异常

在下述场景中，Solidity 将自动产生 `assert` 类型的异常：

- (1) 越界或负的序号值访问数组，如 `i >= x.length` 或 `i < 0` 时访问 `x[i]`。
- (2) 序号越界或负的序号值访问一个定长的 `bytesN`。
- (3) 被除数为 0，如 `5/0` 或 `23%0`。
- (4) 对一个二进制数移动一个负的值，如 `5 << -1`。
- (5) 证书进行显式转换为枚举时，将过大值、负值转为枚举类型并抛出异常。
- (6) 调用未初始化内部函数类型的变量。
- (7) 调用 `assert` 的参数为 `false`。

在下述场景中，Solidity 将自动产生 `require` 类型的异常：

- (1) 调用 `throw`。
- (2) 调用 `require` 的参数为 `false`。
- (3) 通过消息调用一个函数名，但调用过程并没有正确结束。
- (4) 在使用 `new` 创建一个新合约时因为 (3) 的原因而没有正常完成。
- (5) 调用外部函数时，被调用的对象不包含代码。
- (6) 合约中没有 `payable` 修饰符的 `public` 函数在接收以太币时（包括构造函数和回退函数）会出现异常。
- (7) 合约通过 `public` 的 `getter` 函数接收以太币。
- (8) `.transfer()` 执行失败。

两种异常类型的相同点是异常都会撤销所有操作。同时也有如下不同点：

- (1) `gas` 消耗不同，`assert` 类型的异常会消耗掉所有剩余的 `gas`，而 `require` 不会消耗剩余 `gas`，会将剩余 `gas` 返还给调用者。
- (2) 操作符不同，当发生 `assert` 类型异常时，Solidity 会执行一个无效操作（无效指令 `0xfe`），当发生 `require` 类型的异常时，Solidity 会执行一个回退操作（`REVERT` 指令 `0xfd`）。

4. 如何选择

`require()` 函数用于：

- (1) 确认有效条件
- (2) 确认合约声明变量是一致的

(3) 从调用到外部合约返回有效值

revert()函数用于：

处理与 `require()` 同样的类型，但是需要更复杂处理逻辑的场景，如果有复杂的 `if/else` 逻辑流，那么应该考虑使用 `revert()` 函数而不是 `require()`。

assert()函数用于：

预防本不该发生的事情，如果发生就意味着合约中存在需要修复的 `bug`（比如 `assert(1 > 2)`）。一般地，尽量少使用 `assert` 调用，一般 `assert` 应该在函数结尾处使用。



错误及异常处理分析

```
// SPDX-License-Identifier:3.0

pragma solidity ^0.8.5;

contract ErrorTest {

    bool public flag;

    function setFlag(address a) public {

        require(msg.sender == a);

        flag = true;

    }

}
```

代码解析:

```
// SPDX-License-Identifier: GPL-3.0
```

声明机器可读许可证。

```
pragma solidity ^0.8.5;
```

声明合约版本需兼容 0.8.5 以上版本。

```
contract ErrorTest {
```

定义了一个名为 ErrorTest 的合约。

```
    bool public flag;
```

声明一个公开可见的变量，未赋值则默认值为 false。

```
    function setFlag(address a) public {
```

声明一个输入函数，参数为地址类型的 a。

```
require(msg.sender == a);
```

判断调用者的发送地址，是否等于当前地址，成立则继续运行。

```
flag = true;
```

如果等于当前地址，flag 的值为 true，否则为 false。

```
}
```

大括号是一组语句的组合，{}一一对应，成对出现，大括号的作用是将多条语句合成一个复合语句，最后一个大括号表示程序结束。



第六章 开发环境使用

1. 开发环境的注册与认证

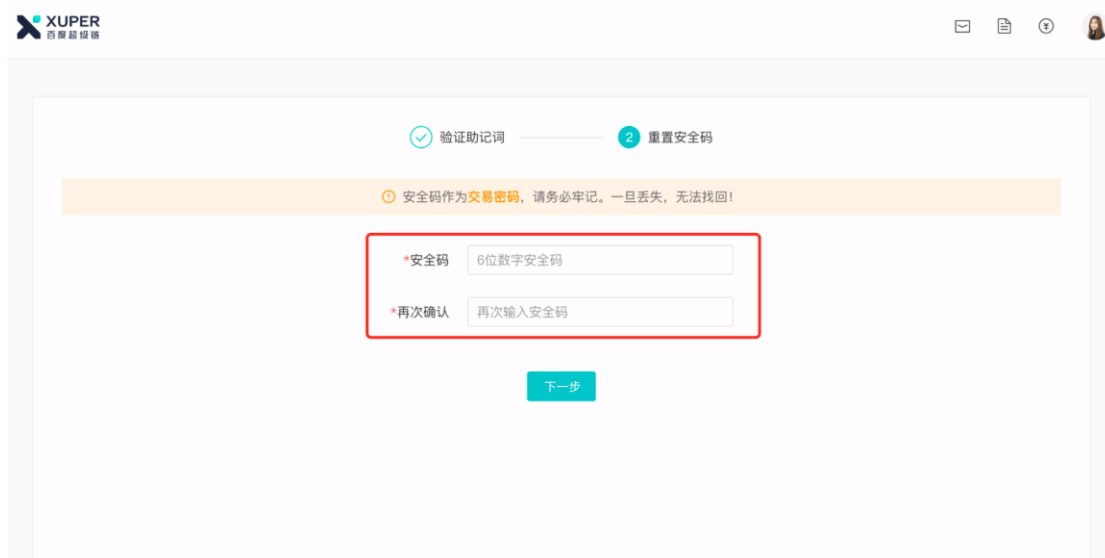
注册

1. 使用网址：<https://xchain.baidu.com/n/console#/xuperos/info> 进入百度超级链开放网络，使用平台前，请先注册百度账号，若已有百度账号，登录即可

2. 点击右上角进入「工作台」



3. 进入设置账户安全码页，安全码作为交易密码，请务必牢记。平台无法提供安全码找回功能。设置完成后，点击「下一步」

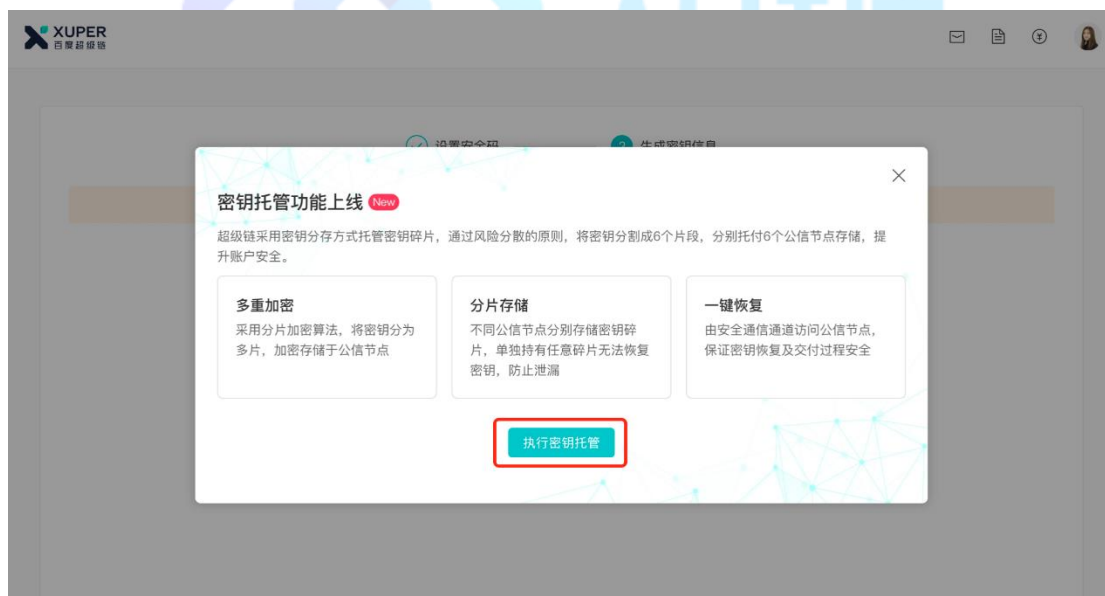


4. 进入记录超级链账户页，请务必按照页面指引，下载账户私密钥、记下助记词，一旦遗失，会导致无法找回账户。平台无法提供找回私密钥、助记词功能。点击「进入工作台」，进入工作台页后，即注册平台账号成功！



5. 超级链账户页，点击【密钥托管】进入密钥托管页面，点击【执行密钥托管】即可完成密钥托管，完成后可进入控制台操作；

注：推荐使用密钥托管，后续重置/找回安全码将自动验证信息



认证

使用平台功能前，需完成实名认证：

方式一

1. 在工作台点击头像，点击「未实名认证」，可跳到认证信息填写页
2. 进入认证页面，按要求完成认证流程，认证立即生效



方式二

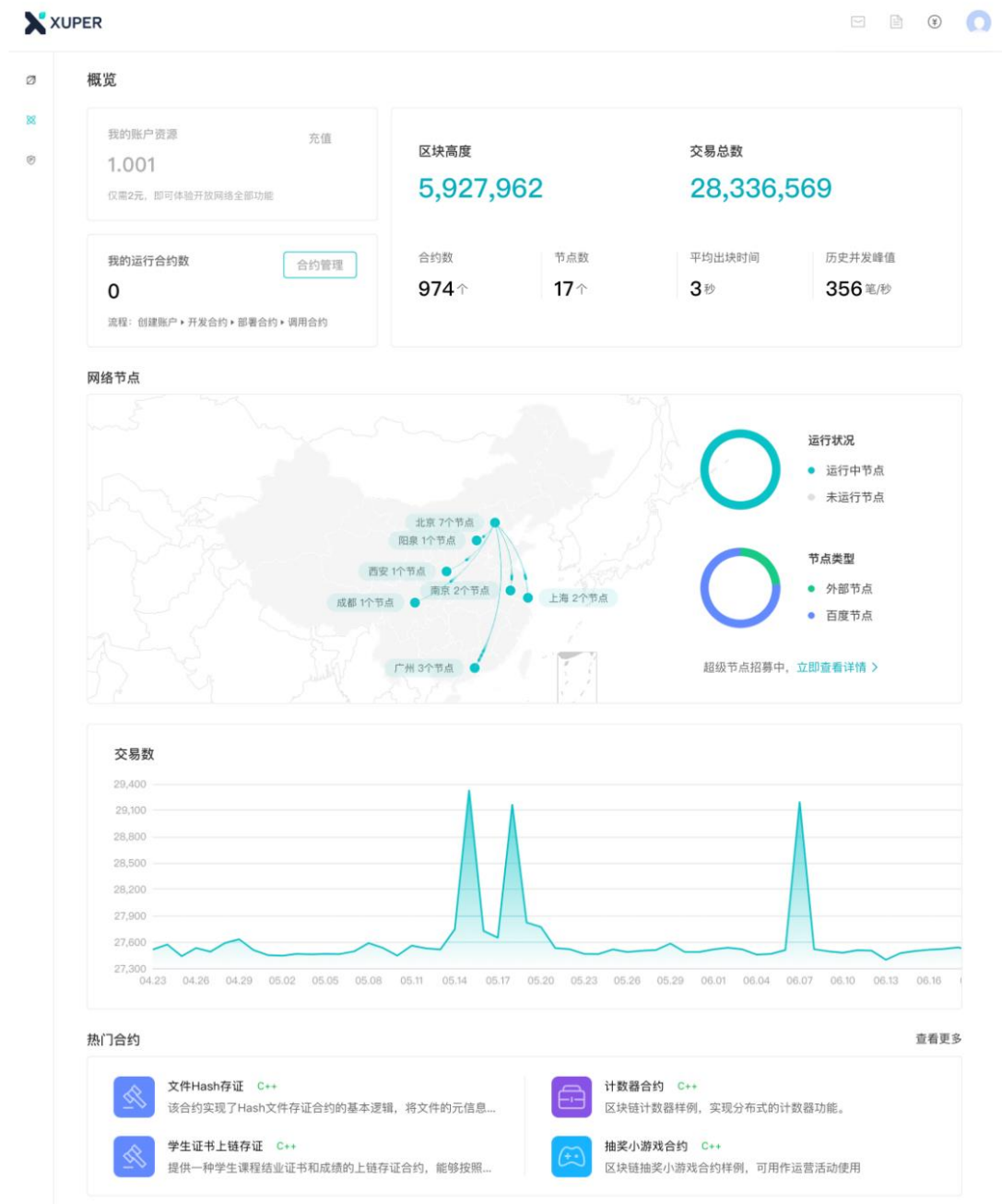
1. 在工作台相关实名认证弹框内，点击「立即认证」，可跳到认证信息填写页
2. 进入认证页面，按要求完成认证流程，认证立即生效



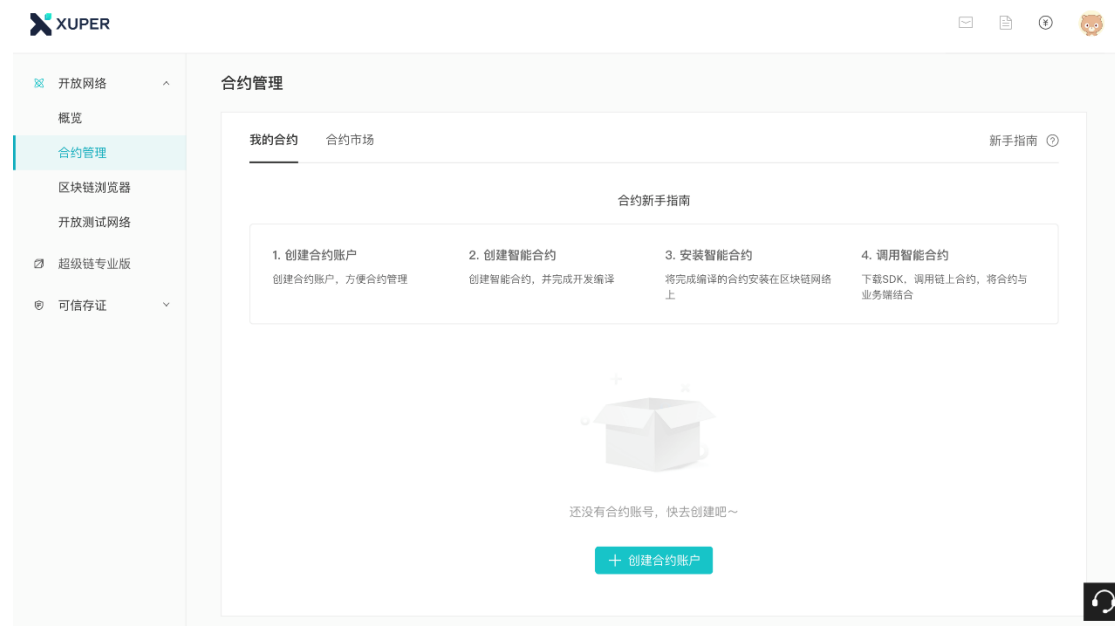
2. 创建智能合约账户

创建智能合约账户概览

1. 在工作台，选择【开放网络 —> 概览】，进入当前页
2. 概览页作为您的个人数据以及节点状况仪表盘。且提供了各功能快速入口，包含：合约管理、热门合约等。目的是让您更好、更便捷的使用开放网络



3. 在工作台，选择「开放网络 —> 合约管理」，点击「创建合约账户」



4. 进入创建合约账户页，输入安全码后点击「确认创建」，系统自动生成账户名称后，即创建完毕

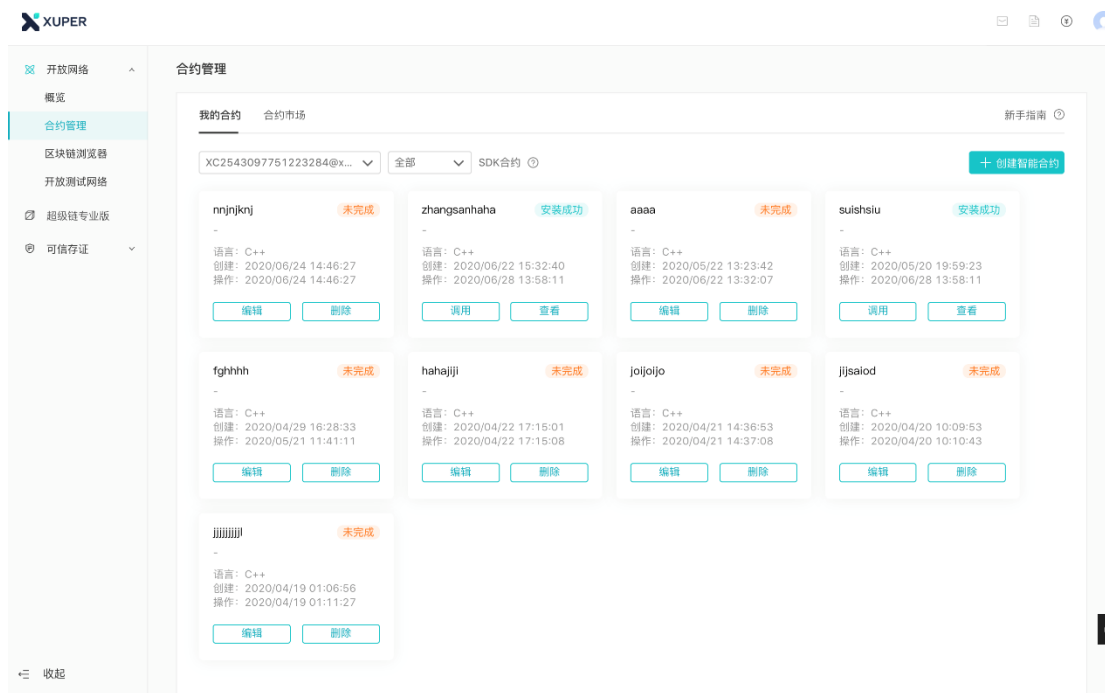




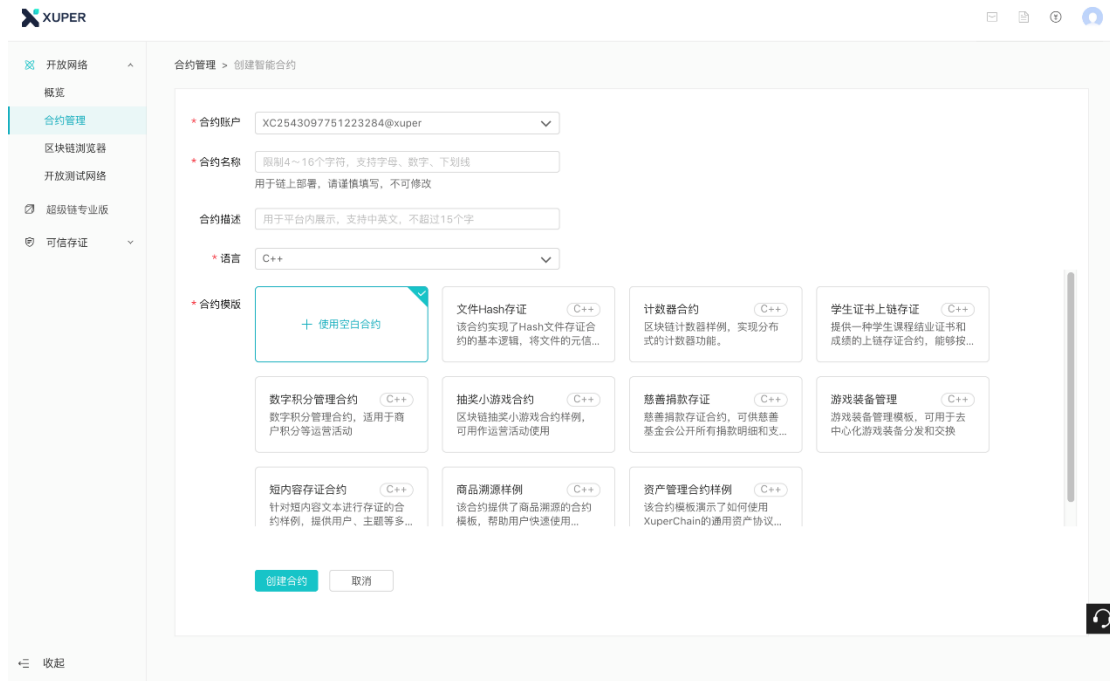
3. 创建与安装智能合约

创建智能合约

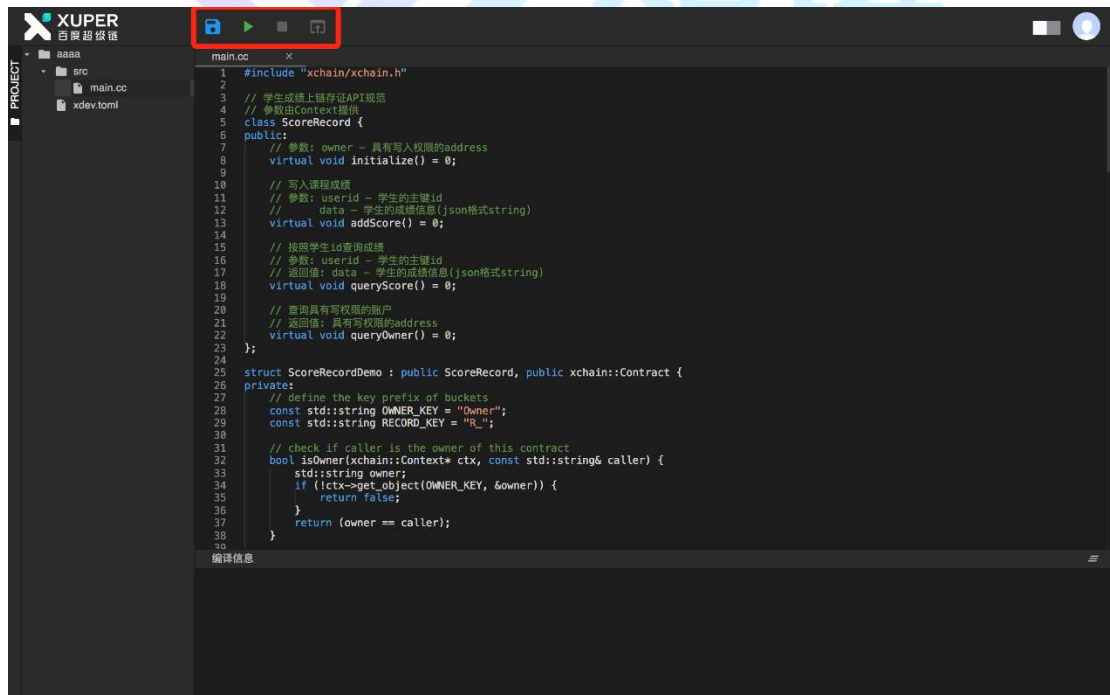
1. 在工作台，选择「开放网络 —> 合约管理」，点击「创建智能合约」



2. 进入创建合约页，按要求填写合约基本信息、选择合约模板，点击创建合约。即创建成功。【合约模板已包含基本的业务逻辑，用户只需在模板代码中填写相关参数即可（参考模板详情完成参数填写）】

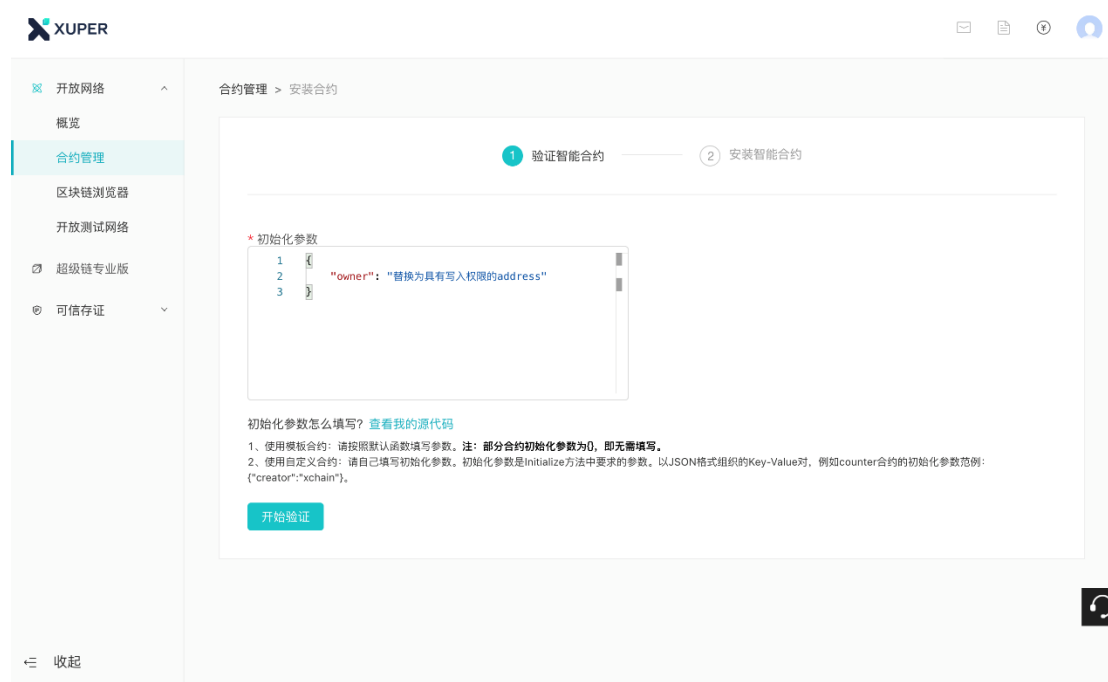


3. 创建成功后可根据页面指引，进入在线 IDE 进行合约开发。或在合约列表页点击编辑按钮。在 IDE 内，选择左侧合约文件，对合约进行编辑、编译操作，编译成功后，点击「安装」按钮，跳转进入安装（部署）环节

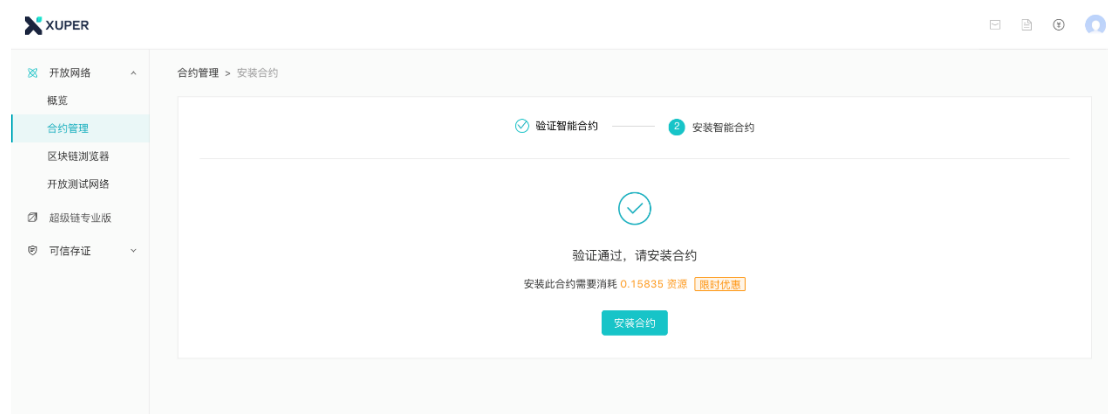


安装（部署）智能合约

1. 进入安装流程，用户需按合约代码完成预执行操作。点击「开始验证」，执行通过会进入安装确认页
 - a. 模板合约；系统会提供模板的函数，只需填写参数即可（可参考模板详情）
 - b. 自定义合约；根据页面操作说明，完成函数、参数填写



2. 进入确认安装页，页面显示安装合约预计消耗的余额。点击「安装合约」将合约上链，上链过程需要等待 20S 左右。



3. 返回首页时，可看到合约状态变更为 '安装成功'，即该合约已完成安装。若未看到合

约状态变更，请刷新当前页面。



4. JAVA SDK 调用智能合约

JAVA SDK 调用智能合约

Java SDK 详情文档: <https://github.com/xuperchain/xuper-java-sdk>

使用说明:

添加项目依赖

```
<dependency>
<groupId>com.baidu.xuper</groupId>
<artifactId>xuper-java-sdk</artifactId>
<version>0.1.1</version>
</dependency>
```

配置文件

如果使用背书功能, 请按如下所示设置配置文件: :

```
Config.setConfigPath("./conf/sdk.yaml");
```

正式网络配置文件: src/main/java/com/baidu/xuper/conf/sdk.yaml.

测试网络配置文件: src/main/java/com/baidu/xuper/conf/sdk.testnet.yaml.

创建 client

```
XuperClient client = new XuperClient("127.0.0.1:37101");
```

导入账户

```
// Import account from local keys
Account account = Account.create("./keys");
```

创建账户

```
// Import account from local keys
Account account = Account.create(1, 2);
```

```
System.out.println(account.getAddress());  
System.out.println(account.getMnemonic());
```

创建合约账户

```
// The account name is XC1111111111111111@xuper  
client.createContractAccount(account, "111111111111111");
```

转账

```
client.transfer(account, "XC1111111111111111@xuper", BigInteger.valueOf(1000000), "1");
```

查询账户余额

```
BigInteger result = client.getBalance("XC1111111111111111@xuper");
```

查询账户余额详细信息

```
XuperClient.BalDetails[] result = client.getBalanceDetails("XC1111111111111111@xuper");
```

部署合约

```
// Using a contract account to deploy contract
account.setContractAccount("XC1111111111111111@xuper");

Map<String, byte[]> args = new HashMap<>();

args.put("creator", "icexin".getBytes());

String codePath = "./counter.wasm";

byte[] code = Files.readAllBytes(Paths.get(codePath));

// the runtime is c

client.deployWasmContract(account, code, "counter", "c", args);

调用合约

Map<String, byte[]> args = new HashMap<>();

args.put("key", "icexin".getBytes());

Transaction tx = client.invokeContract(account, "wasm", "counter", "increase", args);

System.out.println("txid: " + tx.getTxid());

System.out.println("response: " + tx.getContractResponse().getBodyStr());

System.out.println("gas: " + tx.getGasUsed());
```

EVM 合约

```
String abi = "[{\"inputs\": [{\"internalType\": \"uint256\" .....\"};  
  
String bin = \"6080604.....\";  
  
Map<String, String> args = new HashMap<>();  
args.put(\"num\", \"5889\");  
  
Transaction t = client.deployEVMContract(account, bin.getBytes(), abi.getBytes(), contractName,  
args);  
System.out.println(\"txID:\" + t.getTxid());  
  
// storagepay is a payable method. Amount param can be NULL if there is no need to transfer to  
the contract.  
  
Transaction t1 = xuperClient.invokeEVMContract(account, contractName, \"storepay\", args,  
BigInteger.ONE);  
System.out.println(\"txID:\" + t1.getTxid());  
System.out.println(\"tx gas:\" + t1.getGasUsed());  
  
Transaction t2 = xuperClient.queryEVMContract(account, contractName, \"retrieve\", null);  
System.out.println(\"tx res getMessage:\" + t2.getContractResponse().getMessage());  
System.out.println(\"tx res getBodyStr:\" + t2.getContractResponse().getBodyStr());
```