第5章 Solidity 高级编程

第4章介绍了 Solidity 最基础、最常用的一些语法特性，本章将介绍其更高级的用法。例如在编写一些复杂或大型的合约时，可能需要使用到合约继承、接口、库等特性，把一个合约拆分为多个合约，另外，还会介绍如何在合约中动态创建、如何使用通过 ABI 编码与合约交互、怎么节约合约调用的 gas、如何使用 Solidity 内联汇编。

5.1 合约继承

继承是大多数高级语言都具有的特性，Solidity 同样支持继承，Solidity 继承使用关键字 is(类似于 Java 等语言的 extends 或 implements)。例如，contract B is A 表示合约 B 继承合约 A，称 A 为父合约，B 为子合约或派生合约。

当一个合约从多个合约继承时，在区块链上只创建一个子合约，所有父合约的代码被编译到创建的合约中，并不会连带部署父合约。因此当使用 super.f()调用父合约时，也不是进行消息调用，而仅仅是在本合约内进行代码跳转。

举个例子来说明继承的用法，示例代码如下：

```
pragma solidity >= 0.5.0;

contract Owned {
    constructor() public { owner = msg.sender; }
    address payable owner;

    function setOwner(address _owner) public virtual {
        owner = payable(_owner);
    }
}

// 使用 is 表示继承
contract Mortal is Owned {
```

```
    event SetOwner(address indexed owner);

    function kill() public {
        if (msg.sender == owner) selfdestruct(owner);
    }

    function setOwner(address _owner) public override {
        super.setOwner(_owner);
        emit SetOwner(_owner);
    }
}
```

在 4.6.1 节已介绍过，子合约可以访问父合约内的所有非私有成员，因此内部(internal)函数和状态变量在子合约里是可以直接使用的，比如上面示例代码中的父合约状态变量 owner 可以在子合约直接使用。

状态变量不能在子合约中覆盖。例如，上面示例代码中的子合约 Mortal 不可以再次声明状态变量 owner，因为父合约中已经存在该状态变量。但是可以通过重写函数来更改父合约中函数的行为，例如上面示例代码中的子合约 Mortal 中的 setOwner()函数。

5.1.1 多重继承

Solidity 支持多重继承，即可以从多个父合约继承，直接在 is 后面接多个父合约，例如：

```
contract Named is Owned, Mortal {

}
```

注意：如果多个父合约之间也有继承关系，那么 is 后面的合约的书写顺序就很重要，顺序应该是，父合约在前，子合约在后。例如，下面的代码将无法编译：

```
pragma solidity >= 0.4.0;

contract X {}
contract A is X {}
// 编译出错
contract C is A, X {}
```

5.1.2 父合约构造函数

子合约继承父合约时，如果实现了构造函数，父合约的构造函数代码会被编译器

复制到子合约的构造函数中，先看看最简单的情况，也就是构造函数没有参数的情况：

```
contract A {
    uint public a;
    constructor() {
        a = 1;
    }
}

contract B is A {
    uint public b ;
    constructor() {
        b = 2;
    }
}
```

在部署合约 B 时，可以看到 a 为 1，b 为 2。

父合约构造函数如果有参数，会复杂一些，对构造函数传参有两种方式。

(1) 在继承列表中指定参数，即通过 contract B is A(1)的方式对构造函数传参进行初始化，示例代码如下：

```
abstract contract A {
    uint public a;

    constructor(uint _a) {
        a = _a;
    }
}

contract B is A(1) {
    uint public b ;
    constructor() {
        b = 2;
    }
}
```

(2) 在子合约构造函数中使用修饰符方式调用父合约，此时利用部署合约 B 的参数，传入到合约 A 中，示例代码如下：

```
contract B is A {
   uint public b ;

   constructor() A(1) {
       b = 2;
```

```
    }
}
// 或者是
   constructor(uint_b) A(_b / 2) {
       b = _b;
   }
```

5.1.3 抽象合约

如果一个合约包含没有实现的函数，需要将合约标记为抽象合约，使用关键字 abstract 定义抽象合约，不过即使实现了所有功能，合约也可能被标记为 abstract。抽象合约是无法成功部署的，它们通常用作父合约。下面是抽象合约的示例代码：

```
abstract contract A {
    uint public a;

    constructor(uint _a) {
        a = _a;
    }
}
```

抽象合约可以声明一个纯虚函数，纯虚函数没有具体实现代码的函数，其函数声明用“；”结尾，而不是"{ }"，例如：

```
pragma solidity >= 0.5.0;

abstract contract A {
    function get() virtual public ;
}
```

纯虚函数和用关键字 virtual 修饰的虚函数略有区别。关键字 virtual 只表示该函数可以被重写，关键字 virtual 可以修饰除私有可见性(private)函数的任何函数上，无论函数是纯虚函数还是普通的函数，即便是重写的函数，也依然可以用关键字 virtual 修饰，表示该重写的函数可以被再次重写。

如果合约继承自抽象合约，并且没有通过重写实现所有未实现的函数，那么这个合约依旧是抽象的。

5.1.4 函数重写

父合约中的虚函数(使用关键字 virtual 修饰的函数)可以在子合约被重写，以更改

它们在父合约中的行为。重写的函数需要使用关键字 override 修饰，示例代码如下：

```
pragma solidity >= 0.6.0;

contract Base {
    function foo() virtual public {}
}

contract Middle is Base {}
contract Inherited is Middle {
    function foo() public override {}
}
```

对于多重继承，如果有多个父合约有相同定义的函数，关键字 override 后必须指定所有的父合约名，示例代码如下：

```
pragma solidity >= 0.6.0;

contract Base1 {
    function foo() virtual public {}
}

contract Base2 {
    function foo() virtual public {}
}

contract Inherited is Base1, Base2 {
    // 继承自隔两个父合约定义的 foo(), 必须显式地指定 override
    function foo() public override(Base1, Base2) {}
}
```

如果函数没有标记为 virtual(5.2 节介绍的接口除外，因为接口里面所有的函数会自动标记为 virtual)，那么就不能被子合约重写。另外私有函数不可以标记为 virtual。如果 getter()函数的参数和返回值都和外部函数一致，外部函数是可以被 public 的状态变量重写的，示例代码如下：

```
pragma solidity >= 0.6.0;

contract A {
    function f() external pure virtual returns(uint) { return 5; }
}

contract B is A {
    uint public override f;
}
```

但是公共的状态变量不能被重写。

如果函数在多个父合约都有实现，可以通过合约名指定调用哪一个父合约的实现，示例代码如下：

```
pragma solidity >= 0.5.0;
contract X {
    uint public x;
    function setX() public virtual {
        x = 1;
    }
}
contract A is X {
    function setX() public virtual override {
        x = 2;
    }
}

contract C is X, A {
    function setX() public override(X, A) {
        X.setX();
        // super.setX(); 将调用 A 的 setX.
    }
}
```

上述代码，合约 C 的 setX()函数指定调用合约 X 的 setX()函数。如何使用 super 来调用父合约函数，则会根据继承关系图谱，调用紧挨着的父合约，此例中继承关系图谱为(子合约到父合约序列)C、A、X，因此将调用 A 的 setX()函数。

5.2 接　　口

接口和抽象合约类似，不同的是接口不实现任何函数，同时还具有以下限制。

(1) 无法继承其他合约或接口。

(2) 无法定义构造函数。

(3) 无法定义变量。

(4) 无法定义结构体。

(5) 无法定义枚举。

接口由关键字 interface 表示，示例代码如下：

```
interface IToken {
    function transfer(address recipient, uint amount) external;
}
```

就像继承其他合约一样，合约可以继承接口，接口中的函数都会隐式地标记为virtual，意味着它们需要被重写。

除了接口的抽象功能外，接口广泛应用于合约之间的通信，即一个合约调用另一个合约的接口。例如，合约 SimpleToken 实现了上面的接口 IToken：

```
contract SimpleToken is IToken {
function transfer(address recipient, uint256 amount) public override {
...
}
```

另外一个合约(假设合约名为 Award)则通过合约 SimpleToken 给用户发送奖金，奖金就是合约 SimpleToken 表示的代币，这时合约 Award 就需要与合约 SimpleToken 通信(外部函数调用)，示例代码如下：

```
contract Award {
  IToken immutable token;
  // 部署时传入合约 SimpleToken 的地址
  constrcutor(IToken t) {
     token = t;
  }
  function sendBonus(address user) public {
     token.transfer(user, 100);
  }
}
```

函数 sendBonus()用于发送奖金，通过接口函数调用 SimpleToken 实现转账。

5.3 库

在开发合约的时候，通常会有一些函数经常被多个合约调用，这个时候可以把这些函数封装为一个库，实现代码复用。库使用关键字 library 来定义，例如，定义库 SafeMath 的示例代码如下：

```
pragma solidity >= 0.5.0;
library SafeMath {
```

```
    function add(uint a, uint b) internal pure returns (uint) {
        uint c = a + b;
        require(c >= a, "SafeMath: addition overflow");
        return c;
    }
}
```

库 SafeMath 实现了一个加法函数 add(),它可以在多个合约中复用。例如,合约 AddTest 使用 SafeMath 的 add()函数实现加法,代码如下:

```
import "./SafeMath.sol";
contract AddTest {
    function add (uint x, uint y) public pure returns (uint) {
        return SafeMath.add(x, y);
    }
}
```

库是一个很好的代码复用手段。不过要注意,库仅仅是由函数构成的,它不能有自己的状态变量。根据场景不同,库有两种使用方式:一种是库代码嵌入引用的合约内部署(可以称为内嵌库);另一种是作为库合约单独部署(可以称为链接库)。

5.3.1 内嵌库

如果合约引用的库函数都是内部函数,那么在编译合约的时候,编译器会把库函数的代码嵌入合约里,就像合约自己实现了这些函数,这时的库并不会单独部署,前面的合约 AddTest 引用库 SafeMath 就属于这个情况。

5.3.2 链接库

如果库代码内有公共或外部函数,库就可以被单独部署,它在以太坊链上有自己的地址,在部署合约的时候,需要通过库地址把库链接进合约里,合约通过委托调用的方式调用库函数。

前面提到,库没有自己的状态,在委托调用的方式下库合约函数是在发起调用的合约(下文称为主调合约)的上下文中执行的,因此库合约函数中使用的变量(如果有的话)都来自主调合约的变量,库合约函数使用的 this 也是主调合约的地址。

从另一个角度来理解为什么库不能有自己的状态。库是单独部署的,而它又会被多个合约引用(这也是库最主要的功能:避免在多个合约里重复部署,可以节约 gas),如果库拥有自己的状态,那它一定会被多个调用合约修改状态,这将无法保证调用库

函数输出结果的确定性。

把前面的库 SafeMath 的函数 add()修改为外部函数，就可以通过链接库的方式使用，示例代码如下：

```
pragma solidity >= 0.5.0;
library SafeMath {
  function add(uint a, uint b) external pure returns (uint) {
     uint c = a + b;
     require(c >= a, "SafeMath: addition overflow");
     return c;
  }
}
```

合约 AddTest 的代码不用作任何更改，因为库 SafeMath 是独立部署的，合约 AddTest 要调用库 SafeMath 就必须先知道后者的地址，这相当于合约 AddTest 会依赖库 SafeMath，因此部署合约 AddTest 会有一点不同，需要一个合约 AddTest 与库 SafeMath 建立连接的步骤。

先来回顾一下合约的部署过程：第一步由编译器生成合约的字节码，第二步把字节码作为交易的附加数据提交交易。

编译器在编译引用了库 SafeMath 的合约 AddTest 时，编译出来的字节码会留一个空，部署合约 AddTest 时，需要用库 SafeMath 的地址填充此空，这就是链接过程。

感兴趣的读者可以用命令行编译器 solc 操作一下，使用命令 solc--optimize--bin AddTest.sol 可以生成合约 AddTest 的字节码，其中有一段用双下画线留出的空，类似__$239d231e517799327d948ebf93f0befb5c98$__，这个空就需要用库 SafeMath 的地址替换，该占位符是完整的库名称的 Keccak-256 哈希的十六进制编码的 34 个字符的前缀。

5.3.3 using for

在 5.3.2 节中，案例通过 SafeMath.add(x, y)调用库函数，还有一个方式是使用 using LibA for B，它表示把所有 LibA 的库函数关联到类型 B。这样就可以在类型 B 直接调用库函数，示例代码如下：

```
contract testLib {
    using SafeMath for uint;
    function add (uint x, uint y) public pure returns (uint) {
        return x.add(y);
    }
}
```

使用 using SafeMath for uint 后，就可以直接在 uint 类型的 x 上调用 x.add(y)，代码明显更加简洁了。

using LibA for * 则表示 LibA 中的函数可以关联到任意的类型上。使用 using…for…看上去就像扩展了类型的能力。例如，可以给数组添加一个函数 indexOf()，查看一个元素在数组中的位置，示例代码如下：

```
pragma solidity >= 0.4.16;

library Search {
    function indexOf(uint[] storage self, uint value)
        public
        view
        returns (uint)
    {
        for (uint i = 0; i < self.length; i++)
            if (self[i] == value) return i;
        return uint(-1);
    }
}

contract C {
    using Search for uint[];
    uint[] data;

    function append(uint value) public {
        data.push(value);
    }

    function replace(uint _old, uint _new) public {
        // 执行库函数调用
        uint index = data.indexOf(_old);
        if (index == uint(-1))
            data.push(_new);
        else
            data[index] = _new;
    }
}
```

这段代码中函数 indexOf()的第一个参数存储变量 self，实际上对应合约 C 的变量 data。

5.4 应用程序二进制接口

在以太坊(Ethereum)生态系统中,ABI 是从区块链外部与合约进行交互,以及合约与合约之间进行交互的一种标准方式。

5.4.1 ABI 编码

前面在介绍以太坊交易和比特币交易的不同时提到,以太坊交易多了一个 DATA 字段,DATA 的内容会解析为对函数的消息调用,DATA 的内容其实就是 ABI 编码。以下面这个简单的合约为例进行理解。

```
pragma solidity ^0.5.0;
contract Counter {
    uint counter;

    constructor() public {
        counter = 0;
    }
    function count() public {
        counter = counter + 1;
    }

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

按照第 6 章的方法,把合约部署到以太坊测试网络 Ropsten 上,并调用函数 count(),然后查看实际调用附带的输入数据,在区块链浏览器 etherscan 上交易的信息地址为 https://ropsten.etherscan.io/tx/0xafcf79373cb38081743fe5f0ba745c6846c6b08f375-fda028556b4e52330088b,如图 5-1 所示。

如图 5-1 所示,交易通过携带数据 0x06661abd 表示调用合约函数 count(),0x06661abd 被称为函数选择器(function selector)。

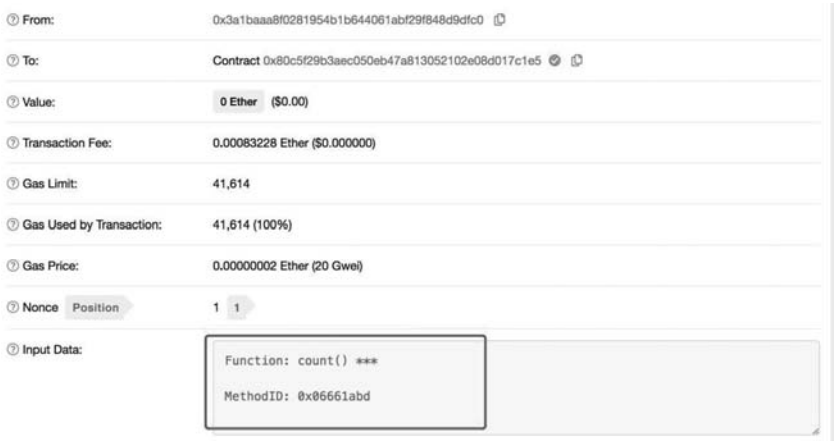

图 5-1　调用信息截图

5.4.2　函数选择器

在调用函数时，用前面 4 字节的函数选择器指定要调用的函数，函数选择器是某个函数签名的 Keccak(SHA-3)哈希的前 4 字节，即：

```
bytes4(keccak256("count()"))
```

count()的 Keccak 的哈希结果是 06661abdecfcab6f8e8cf2e41182a05dfd130c76cb32b448d9306aa9791f3899，开发者可以用在线哈希工具(https://emn178.github.io/online-tools/keccak_256.htm)验证，取出前面 4 个字节就是 0x06661abd。

函数签名是函数名及参数类型的字符串(函数的返回类型并不是这个函数签名的一部分)。比如 count()就是函数签名，当函数有参数时，使用参数的基本类型，并且不需要变量名，因此函数 add(uinti)的签名是 add(uint256)。如果有多个参数，使用“,”隔开，并且要去掉表达式中的所有空格。因此，foo(uint a,bool b)函数的签名是 foo(uint256,bool)，函数选择器计算则是：

```
bytes4(keccak256("foo(uint256,bool)"))
```

公有或外部(public/external)函数都包含一个成员属性.selector 的函数选择器。

5.4.3　参数编码

如果一个函数带有参数，编码的第 5 字节开始是函数的参数。在前面的合约 Counter 中添加一个带参数的方法：

```
function add(uint i) public {
      counter = counter + i;
}
```

重新部署之后，使用 16 作为参数调用函数 add()，调用方法如图 5-2 所示。

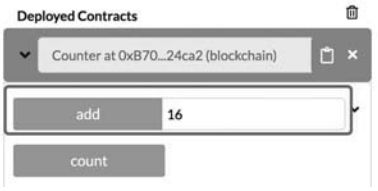

图 5-2　Remix 调用 add()函数

在 etherscan 上查看交易附加的输入数据，查询地址为 https://ropsten.etherscan.io/tx/0x5f2a2c6d94aff3461c1e8251ebc5204619acfef66e53955dd2cb81fcc57e12b6，如图 5-3 所示。

图 5-3　函数调用的 ABI 编码

输入数据为 0x1003e2d20000000000000000000000000000000000000000000000000000000000000010。其中，前 4 字节 0x1003e2d2 为函数 add()的函数选择器，后面的 32 字节是参数 16 的二进制表示，会补充到 32 字节长度。不同的类型，参数编码方式会有所不同，详细的编码方式可以参考 ABI 编码规范（https://learnblockchain.cn/docs/solidity/abi-spec.html）。

5.4.4　通过 ABI 编码调用函数

通常，在合约中调用合约 Counter 的函数 count()的形式是 Counter.count()，第 4 章介绍过底层调用函数 call()，因此也可以直接通过函数 call()和 ABI 编码来调用函数 count()：

```
(bool success, bytes memory returnData) =
address(c).call("0x06661abd"); //c 为 Counter 合约的地址,0x06661abd
require(success);
```

其中，c 为 Counter 合约的地址，0x06661abd 是函数 count()的编码，如果函数 count()发生异常，调用 call()会返回 false，因此需要检查返回值。

使用底层调用可以非常灵活地调用不同合约的不同的函数，在编写合约时，并不需要提前知道目标函数的合约地址及函数。例如，定义一个合约 Task，它可以调用任意合约，代码如下：

```
contract Task {
    function execute(address target, uint value, bytes memory data) public payable
returns (bytes memory) {
        (bool success, bytes memory returnData) = target.call{value: value}(data);
        require(success, "execute: Transaction execution reverted.");
        return returnData;
    }
}
```

5.4.5 ABI 接口描述

ABI 接口描述是由编译器编译代码之后，生成的一个对合约所有函数和事件描述的 JSON 文件。一个描述函数的 JSON 包含以下字段。

(1) type：可取值有 function、constructor、fallback，默认为 function。

(2) name：函数名称。

(3) inputs：一系列对象，每个对象包含属性 name(参数名称)和 type(参数类型)。

(4) components：给元组类型使用，当 type 是元组(tuple)时，components 列出元组中每个元素的名称(name)和类型(type)。

(5) outputs：一系列类似 inputs 的对象，无返回值时，可以省略。

(6) payable：true 表示函数可以接收以太币，否则表示不能接收，默认值为 false。

(7) stateMutability：函数的可变性状态，可取值有 pure、view、nonpayable、payable。

(8) constant：如果函数被指定为 pure 或 view，则为 true。

一个描述事件的 JSON 包含以下字段。

(1) type：总是 event。

(2) name：事件名称。

(3) inputs：对象数组，每个数组对象会包含属性 name(参数名称)和 type(参数类型)。

(4) components：供元组类型使用。

(5) indexed：如果此字段是日志的一个主题，则为 true，否则为 false。

(6) anonymous：如果事件被声明为 anonymous，则为 true。

在 Remix 的编译器页面，编译输出的 ABI 接口描述文件，可以用来查看合约 Counter 的接口描述，只需要在如图 5-4 所示方框处单击 ABI 按钮，ABI 描述就

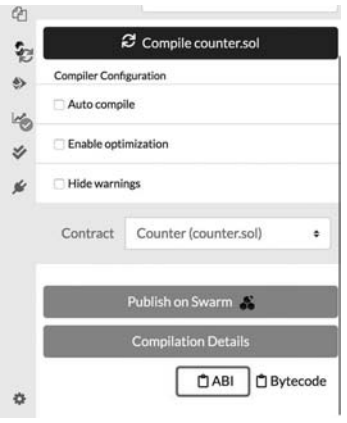

图 5-4　获取 ABI 信息

会复制到剪切板上。

下面是 ABI 描述代码示例：

```
{
    "constant": false,
    "inputs": [],
    "name": "count",
    "outputs": [],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "function"
},
{
    "constant": true,
    "inputs": [],
    "name": "get",
    "outputs": [
       {
          "internalType": "uint256",
          "name": "",
          "type": "uint256"
       }
    ],
    "payable": false,
    "stateMutability": "view",
    "type": "function"
},
{
    "inputs": [],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "constructor"
}
```

JSON 数组中包含了 3 个函数描述，描述合约所有接口方法，在合约外部（如 DApp）调用合约方法时，就需要利用这个描述获得合约的方法，第 7 章会进一步介绍 ABI JSON 的应用。不过，DApp 开发人员并不需要使用 ABI 编码调用函数，只需要提供 ABI 的接口描述 JSON 文件，编码由 Web3 或 ether.js 库来完成。

5.5 gas 优化

gas 优化是开发以太坊智能合约一项非常有挑战性的任务。以太坊上的计算资源是有限的，每个区块可用的 gas 是有上限的（2021 年单个区块区块限制约为 1250

万)。随着链上去中心化金融应用的兴起,以太坊的利用率逐渐增长,由于矿工是以gas竞价排名的方式打包区块,当以太坊的利用率非常高时,只有gas价格高的交易才能得到打包的机会,从而导致交易手续费一直居高不下。在这样的背景下,优化合约交易的gas耗用量就显得更加重要。

5.5.1 变量打包

合约内总是用连续32字节(256位)的插槽来存储状态变量。当操作者在一个插槽中放置多个变量时,被称为变量打包。

存储操作码指令(SSTORE)消耗的gas成本非常高,首次写时,每32字节的成本是20 000gas,而后续每次修改则为5000gas,变量打包可以减少SSTORE指令的使用。

变量打包就像俄罗斯方块游戏。如果打包的变量超过当前槽的32字节限制,它将被存储在一个新的插槽中。操作者必须找出哪些变量最适合放在一起,以最小化浪费的空间。

因为使用每个插槽都需要消耗gas,变量打包通过减少合约要求的插槽数量,帮助优化gas的使用,例如以下变量:

```
uint128 a;
uint256 b;
uint128 c;
```

这些变量无法打包。如果b和a打包在一起,那么就会超过32字节的限制,所以会被放在新的一个储存插槽中。同样c和b打包也如此。使用下面的顺序定义变量,效果更好:

```
uint128 a;
uint128 c;
uint256 b;
```

因为c和a打包之后不会超过32字节,所以可以被存放在一个插槽中。

在选择数据类型时,如果刚好可以与其他变量打包放入一个储存插槽中,那么使用一个小数据类型是不错的。

但是当变量无法和合并打包时,应该尽量使用256位的变量,例如,uint256和bytes32,因为EVM的运行时,总是一次处理32字节,如果只存储一个uint8,EVM其

实会用零填充所有缺少的数字，这会耗费 gas。同时，EVM 执行计算时也执行类型转化，将变量转化为 uint256，因此使用 256 位的变量会更有效率。

5.5.2 选择适合的数据类型

1. 使用常量或不可变量

如果在合约运行中，一个数据的值一直是固定的，则应该使用常量（constant）或不可变量（immutable），这两种类型将数据包含在智能合约的字节码中，则用于加载和存储数据的 gas 消耗将大大减少。定义常量（constant）或不可变量（immutable）的代码如下：

```
contract C {
    uint constant X = 32 ** 22 + 8;  // 定义常量
    string constant TEXT = "abc";

    uint immutable decimals;          // 定义不可变量

    constructor(uint _d) {
      decimals = _d;
    }
}
```

常量与不可变量的区别是，常量在编译期确定值，不可变量在部署时确定值。

2. 固定长度比变长更好

如果能确定一个数组有多少元素，应该优先采用固定大小的方式，例如，下面是一个按月存储的数据，可以使用 12 个元素的数组：

```
uint256[12] monthlys;
```

这同样也适用于字符型，一个 string 或者 bytes 变量是变长的。如果一个字符串很短，则应该使用固定长度的 bytes1～bytes32 类型。

3. 映射和数组

大多数的情况下，映射的 gas 消耗会优于数组，映射存储、读取、删除的 gas 消耗都是固定的，而数组的 gas 消耗会随数组长度增长而线性增长。不过，当元素类型是较小的数据类型时，数组是一个不错的选择时。数组元素会像其他存储变量一样被打包，这样可节省存储空间以弥补昂贵的数组操作。如果必须要设计一个动态数组，也

需要尽量让数组保持末尾递增，避免数组的移位。

5.5.3 内存和存储

在内存中操作数据，比在存储中操作状态变量数据方便得多。减少存储操作的一种常见方法是在分配给存储变量之前，使用内存变量进行操作。例如，下面的 Solidity 代码中，num 变量是一个存储型状态变量，那么在每次循环中都操作 num 很浪费 gas。

```
uint num = 0;
function expensiveLoop(uint x) public {
  for(uint i = 0; i < x; i++) {
    num += 1;
  }
}
```

可以创建一个临时变量，来代替上述全局变量参与循环，然后在循环结束后重新将临时变量的值赋给全局状态变量，代码如下：

```
uint num = 0;
function lessExpensiveLoop(uint x) public {
  uint temp = num;
  for(uint i = 0; i < x; i++) {
    temp += 1;
  }
  num = temp;
}
```

5.5.4 减少存储

1. 清理存储

根据 EVM 的规则，在删除状态变量时，EVM 会返还一部分 gas，返还的 gas 可以用来抵消交易消耗的 gas。尤其在清理大数据变量时，返还的 gas 将相当可观，最高可达交易消耗 gas 的一半，代码如下：

```
contract DelC {
  uint [] bigArr;
```

```
  function doSome() public {
    // do some
    delete bigArr;
  }
}
```

同样的道理，当有一个合约不再使用时，可以把合约销毁返还 gas。

2. 使用事件储存数据

那些不需要在链上被访问的数据可以存放在事件中达到节省 gas 的目的。

触发事件的 LOG 指令基础费用是 375 gas，远小于 SSTORE 指令。

例如，要记录文档的注册记录，很多时候不假思索，会这样写：

```
contract Registry {
  mapping (uint256 => address) public documents;
  function register(uint256 hash) public {
    documents[hash] = msg.sender;
  }
}
```

其实下面的代码更高效：

```
contract DocumentRegistry {
  event Registered(uint256 hash, address sender);
  function register(uint256 hash) public {
    emit Registered(hash, msg.sender);
  }
}
```

这个合约没有任何变量存储，但是实现了同样的功能，事件记录同样会在区块链上永久保存，在需要查询数据时，可以通过订阅事件把数据缓存到数据库查询记录。

5.5.5 其他建议

1. 初始化

在 Solidity 中，每个变量的赋值都要消耗 gas。在初始化变量时，避免使用默认值初始化，例如“uint256 value”比“uint256 value=0”消耗的 gas 少。

2. Require 字符串

如果操作者在 require 中增加语句，可以通过限制字符串长度为 32 字节降低 gas 消耗。

3. 精确的声明函数可见性

在 Solidity 合约开发中，显式声明函数的可见性不仅可以提高智能合约的安全性，同时也有利于优化合约执行的 gas 成本。例如，仅会通过外部执行的函数应该显式地标记函数为外部函数(external)而不是笼统地使用公共函数(public)。

4. 链下计算

例如在排序列表中，向列表中添加元素后，依旧要确保其仍是有序的。经验不足时需要在整个集合中进行迭代，以找到合适的位置进行插入。一种更有效的方法是在链下计算合适的位置，仅在链上进行相应的验证(例如，添加的值位于其相邻元素之间)，这可以防止成本随数据结构的变化呈线性增长。

5. 警惕循环

当合约中存在依赖时间、依赖数据大小(如数组长度)的循环时，很可能导致潜在的漏洞。随着数据量的增多(或时间的增长)，gas 的消耗就可能对应地线性增长，很可能突破区块限制导致无法打包。

首先要尽可能避免使用这种循环，将依靠循环的线性增长的计算量尽可能转化为固定大小的计算量(或常量)。如果没法做到转换，那就要考虑限制循环的次数，即限制总数据的大小，把一个大数据分拆为多个分段的小数据(即想办法限制单次的计算量大小)，比如依靠时间长度计算收益的质押合约，可以设置质押有效期，比如设置质押有效期最长为一年，一年到期之后，用户需提取再质押。

5.6 使用内联汇编

本节的内容在智能合约开发中使用较少，读者也可以选择跳过，本节亦是抛砖引玉，内联汇编语言 Yul 仍然在不断地进化，对这部分内容感兴趣的读者可以阅读官方资料。

5.6.1 汇编基础概念

实际上很多高级语言(例如 C、Go 或 Java)编写的程序，在执行之前都将先编译为

汇编语言。汇编语言与 CPU 或虚拟机绑定实现指令集，通过指令告诉 CPU 或虚拟机执行一些基本任务。

Solidity 语言可以理解为是以太坊虚拟机 EVM 指令集的抽象，让操作者更容易编写智能合约更容易。而汇编语言则是 Solidity 语言和 EVM 指令集的一个中间形态，Solidity 也支持直接使用内联汇编，下面是在 Solidity 代码中使用汇编代码的例子。

```
contract Assembler {
 function do_something_cpu() public {
   assembly {
   // 编写汇编代码
   }
 }
}
```

在 Solidity 中使用汇编代码有如下的好处。

（1）进行细粒度控制。可以在汇编代码使用汇编操作码直接与 EVM 进行交互，从而对智能合约执行的操作进行更精细的控制。汇编提供了更多的控制权执行某些仅靠 Solidity 不可能实现的逻辑，例如控制指向特定的内存插槽。

（2）更少的 gas 消耗。此处通过一个简单的加法运算对比两个版本的 gas 消耗，一个版本是仅使用 Solidity 代码，一个版本是仅使用内联汇编。

```
function addAssembly(uint x, uint y) public pure returns (uint) {
    assembly {
        let result := add(x, y)       // x + y
        mstore(0x0, result)           // 在内存中保存结果
        return(0x0, 32)               // 从内存中返回 32 字节
    }
}

function addSolidity(uint x, uint y) public pure returns (uint) {
   return x + y;
}
```

gas 的消耗如图 5-5 所示。从图 5-5 可以看到，使用内联汇编可以节省 86gas。对于这个简单的加法操作来说，减少的 gas 并不多，但已经表明直接使用内联汇编将消耗更少的 gas，更复杂的逻辑能更显著地节省 gas。

[call] from:0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to:AssemblyLanguage.addSolidity(uint256,uint256) data:0xde9...00023 Debug

transaction hash	0x2273b95302994d01f4d313ff57775baa99eb9666ccf2a5ca44f975d5e547de0a
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	AssemblyLanguage.addSolidity(uint256,uint256) 0xf2bd5de8b57ebfc45dcee97524a7a08fccc80aef
transaction cost	21996 gas Cost only applies when called by a contract)
execution cost	340 gas (Cost only applies when called by a contract)
hash	0x2273b95302994d01f4d313ff57775baa99eb9666ccf2a5ca44f975d5e547de0a
input	0xde9...00023
decoded input	{ "uint256 x": "25", "uint256 y": "35" }
decoded output	{ "0": "uint256: 60" }
logs	[]

(a) 使用Solidity的gas消耗

[call] from:0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to:AssemblyLanguage.addAssembly(uint256,uint256) data:0x08b...00023 Debug

transaction hash	0x2b217b695efb969190fc249b22bca41505f3fe9f3ccd283eb7d66b8558c28a72
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	AssemblyLanguage.addAssembly(uint256,uint256) 0xf2bd5de8b57ebfc45dcee97524a7a08fccc80aef
transaction cost	21910 gas Cost only applies when called by a contract)
execution cost	254 gas (Cost only applies when called by a contract)
hash	0x2b217b695efb969190fc249b22bca41505f3fe9f3ccd283eb7d66b8558c28a72
input	0x08b...00023
decoded input	{ "uint256 x": "25", "uint256 y": "35" }
decoded output	{ "0": "uint256: 60" }
logs	[]

(b) 使用内联汇编的gas消耗

图 5-5　gas 消耗对比图

5.6.2　Solidity 中引入汇编

可以在 Solidity 中使用 assembly{}嵌入汇编代码段，这被称为内联汇编，代码如下：

```
assembly {
 // some assembly code here
}
```

在 assembly 块内的代码开发语言被称为 Yul。

Solidity 可以引入多个汇编代码块，不过汇编代码块之间不能通信，也就是说在一个汇编代码块里定义的变量，在另一个汇编代码块中不可以访问。

因此以下这段代码的 b 无法获取到 a 的值：

```
assembly {
    let a := 2
}

assembly {
    let b := a          // Error
}
```

再看一个使用内联汇编代码完成加法的例子，重写函数 addSolidity()，代码如下：

```
function addSolidity(uint x, uint y) public pure returns (uint) {
 assembly {
     let result := add(x, y)      // ① x + y
     mstore(0x0, result)          // ② 将结果存入内存
     return(0x0, 32)              // ③
 }
}
```

对上面这段代码做一个简单的说明。

(1) 创建一个新的变量 result，通过操作码 add 计算 x+y，并将计算结果赋值给变量 result。

(2) 使用操作码 mstore 将变量 result 的值存入地址 0x0 的内存位置。

(3) 表示从内存地址 0x0 返回 32 字节。

5.6.3 汇编变量定义与赋值

在 Yul 语言中，使用关键字 let 定义变量。使用操作符“:=”给变量赋值。

```
assembly {
 let x := 2
}
```

由于 Solidity 只需要用“=”,因此在 Yul 不要忘了“:”。如果没有给变量赋值,那么变量会被初始化为 0,代码如下:

```
assembly {
  let x           // 自动初始化为 x = 0
  x := 5          // x 现在的值是 5
}
```

也可以使用表达式给变量赋值,代码如下:

```
assembly {
  let a := add(x, 3)
}
```

5.6.4 汇编中的块和作用域

在 Yul 汇编语言中,用{}表示一个代码块,变量的作用域是当前的代码块,即变量在当前的代码块中有效,代码如下:

```
assembly {
    let x := 3          // 变量 x 一直可见
 {
    let y := x          // 正确
 }                      // 到此处会销毁 y

 {
    let z := y          // 错误
 }
}
```

在上面的示例代码中,y 和 z 都是仅在所在块内有效,因此 z 获取不到 y 的值。不过在函数和循环中,作用域规则有一些不一样,将在 5.6.6 节及 5.6.9 节中介绍。

5.6.5 汇编中访问变量

汇编中只需要使用变量名就可以访问局部变量(指在函数内部定义的变量),无论该变量是定义在汇编块中,还是在 Solidity 代码中,示例代码如下:

```
function localvar() public pure {
 uint b = 5;
```

```
    assembly {
        let x := add(2, 3)
        let y := mul(x, b)      // 使用了外面的 b
        let z := add(x, y)      // 访问了内部定义的 x,y
    }
}
```

5.6.6 for 循环

Yul 汇编语言同样支持 for 循环，例如，value+2 计算 n 次的示例代码如下：

```
function forloop(uint n, uint value) public pure returns (uint) {
    assembly {
      for { let i := 0 } lt(i, n) { i := add(i, 1) } {
          value := add(2, value)
      }
      mstore(0x0, value)
      return(0x0, 32)
  }
}
```

for 循环的条件部分包含 3 个元素。

(1) 初始化条件：let i:=0。

(2) 判断条件：lt(i,n)，这是函数式风格，表示 i 小于 n。

(3) 迭代后续步骤：add(i, 1)。

for 循环中变量的作用范围和前面介绍的作用域略有不同。在初始化部分定义的变量在循环条件的其他部分都有效。在 for 循环的其他部分中声明的变量依旧遵守 4.6.4 节介绍的作用域规则。此外，Yul 汇编语言中没有 while 循环。

5.6.7 if 判断语句

Yul 汇编语言支持使用 if 语句设置代码执行的条件，但是没有 else 分支，同时每个条件对应的执行代码都需要用“{}”，示例代码如下：

```
assembly {
    if slt(x, 0) { x := sub(0, x) }     // 正确
    if eq(value, 0) revert(0, 0)        // 错误, 需要{}
}
```

5.6.8 汇编 switch 语句

汇编语言中也有 switch 语句，它将一个表达式的值与多个常量进行对比，并选择相应的代码分支来执行。switch 语句支持一个默认分支 default，当表达式的值不匹配任何其他分支条件时，将执行默认分支的代码，示例代码如下：

```
assembly {
    let x := 0
    switch calldataload(4)
    case 0 {
        x := calldataload(0x24)
    }
    default {
        x := calldataload(0x44)
    }
    sstore(0, div(x, 2))
}
```

switch 语句的分支条件需要具有相同的类型、不同的值。如果分支条件已经涵盖所有可能的值，那么不允许再出现 default 条件。要注意的是，Solidity 语言中是没有 switch 语句的。

5.6.9 汇编函数

可以在内联汇编中定义自定义底层函数，调用这些自定义的函数和使用内置的操作码一样。下面的汇编函数用来分配指定长度(length)的内存，并返回内存指针 pos，代码如下：

```
assembly {
    function alloc(length) -> pos {          // ①
        pos := mload(0x40)
        mstore(0x40, add(pos, length))
    }
    let free_memory_pointer := alloc(64)  // ②
}
```

代码说明：①定义了一个 alloc()函数，函数使用“->”指定返回值变量，不需要显式 return 返回语句；②使用了定义的函数。

定义函数不需要指定汇编函数的可见性，因为它们仅在定义的汇编代码块内有效。

5.6.10 元组

汇编函数可以返回多个值，它们被称为元组，可以通过元组一次给多个变量赋值，示例代码如下：

```
assembly {
    function f() -> a, b {}
    let c, d := f()
}
```

5.6.11 汇编缺点

上面的内容介绍了汇编语言的一些基本语法，可以帮助操作者在智能合约中实现简单的内联汇编代码。不过，一定要谨记，内联汇编是一种以较低级别访问以太坊虚拟机的方法。它会绕过 Solidity 编译器的安全检查。只有在操作者对自身能力非常有信心且必需时才使用它。