

以太坊学习：第七天

目录

Solidity的深入理解

- solidity源文件布局
- Solidity源文件布局-- import
- Solidity值类型
- Solidity引用类型
- 地址类型成员变量
- 地址成员变量用法
- 字符数组 (Byte Arrays)
- 枚举 (Enum)
- 数组 (Array)
- 结构 (Struct)
- 映射 (Mapping)
- Solidity数据位置
- 数据位置总结
- Solidity函数声明和类型
- Solidity函数可见性
- 函数可见性例子
- Solidity函数状态可变性
- Solidity函数状态可变性
- 函数修饰器 (modifier)
- 回退函数 (fallback)
- 事件 (event)
- Solidity异常处理
- Solidity中的单位
 - 以太坊 (ether)
 - 时间

Solidity的深入理解

solidity源文件布局

pragma (版本杂注)

- 源文件可以被版本 杂注pragma所注解，表明要求的编译 器版本
- 例如： `pragma solidity ^0.4.0;`
- 源文件将既不允许低于 0.4.0 版本的编译器编译，也不允 许高于（包含） 0.5.0 版本的编译器编译（第二个条件因 使用 ^ 被添加） `import`（导 件）
- Solidity 所支持的导入语句`import`，语法同 JavaScript（从 ES6 起）非常类似

Solidity源文件布局-- import

- `import "filename"`：从“filename”中导入所有的全局符号到当前全局作用域中
- `import * as symbolName from "filename"`：创建一个新的全局符号 `symbolName`，其成员均来自“filename” 中全局符号
- `import {symbol1 as alias, symbol2} from "filename"`：创建新的全局符号 `alias` 和 `symbol2`，分别从 "filename" 引 用 `symbol1` 和 `symbol2`
- `import "filename" as symbolName`：这条语句等同于 `import * as symbolName from "filename"`

Solidity值类型

- 布尔 (bool)：可能的取值为字符常量值 `true` 或 `false`
- 整型 (int/uint)：分别表示有符号和无符号的不同位数的整型变量；支持 关键字 `uint8` 到 `uint256`（无符号，从 8 位到 256 位）以及 `int8` 到 `int256`（有符号，从 8 位到 256 位）位为步长递增
- 定长浮点型 (fixed / ufixed)：表示各种大 可用的小数位



LEVI_104

👍 0 🗨️ 0 ⭐ 0

- 地址 (address) : 存储一个 20 字节的值 (以太坊地址大小)
- 定长字节数组: 关键字有 bytes1, bytes2, bytes3, ..., bytes32
- 枚举 (enum) : 一种用户可以定义类型的方法, 与C语言类似, 默认从0 开始递增, 一般用来模拟合约的状态
- 函数 (function) : 一种表示函数的类型

Solidity引用类型

数组 (Array)

- 数组可以在声明时指定长度 (定长数组), 也可以动态调整大小 (变长数组、动态数组)
- 对于存储型 (storage) 的数组来说, 元素类型可以是任意的 (即元素也可以是数组类型, 映射类型或者结构体); 对于内存型 (memory) 的元素类型不能是映射 (mapping) 类型

结构 (Struct) : Solidity 支持通过构造结构体的形式定义新的类型

映射 (Mapping) : 映射可以视作 哈希表, 在实际的初始化过程中创建每个可能的 key, 并将其映射到字节形式全是零的值 (类型默认值)

address: 地址类型存储一个 20 字节的值 (以太坊地址的大小); 地址类型也有成员变量, 并作为所有合约的基础

address payable (v0.5.0引入) : 与地址类型基本相同, 不过多出了 transfer 和 send 两个成员变量

address与address payable的区别

两者区别和转换

- Payable 地址是可以发送 ether 的地址, 而普通 address 不能
- 允许从 payable address 到 address 的隐式转换, 而反过来的直接转换是不可能的 (唯一方法是通过uint160来进行中间转换)
- 从0.5.0版本起, 合约不再是从地址类型派生而来, 但如果它有payable的回退函数, 那同样可以显式转换为 address 或者 address payable 类型

地址类型成员变量

- **<address>.balance (uint256)**: 该地址的 ether 余额, 以Wei为单位。
- **<address payable>.transfer(uint256 amount)**: 向指定地址发送数量为 amount 的 ether (以Wei为单位), 失败时抛出异常, 发送 2300 gas 费, 不可调节
- **<address payable>.send(uint256 amount) returns (bool)**: 向指定地址发送数量为 amount 的 ether (以Wei为单位), 失败时返回 false, 发 gas 的矿工费用, 不可调节
- **<address>.call(bytes memory) returns (bool, bytes memory)**: 发出底层函数 CALL, 失败时返回 false, 发送所有可用 gas, 可调节
- **<address>.delegatecall(bytes memory) returns (bool, bytes memory)**: 发出底层函数 DELEGATECALL, 失败时返回 false, 发送所有可用 gas, 可调节
- **<address>.staticcall(bytes memory) returns (bool, bytes memory)**: 发出底层函数 STATICCALL, 失败时返回 false, 发送所有可用 gas, 可调节

地址成员变量用法

- **balance 和 transfe**: 可以使用 balance 属性来查询一个地址的余额, 可以使用 transfer 函数向一个payable地址发送 以太币Ether (以 wei 为

```

1 | address payable x = address(0x123);
2 |
3 | address myAddress = address(this);
4 |
5 | if (x.balance < 10 && myAddress.balance >= 10)
6 |
7 | x.transfer(10);

```

- **send**: send 是 transfer 的低级版本。如果执行失败, 当前的合约不会因为异常而终止, 但 send 会返回 false
- **call**: 也可以用call来实现转币的操作, 通过添加.gas()和.value()修饰器:

```

1 | nameReg.call.gas(1000000).value(
2 |
3 |     1 ether)(abi.encodeWith

```



LEVI_104

👍 0 🗨️ 0 ⭐ 0

字符数组 (Byte Arrays)

定长字符数组

- 属于值类型，bytes1, bytes2, ..., bytes32分别代表了长度为1到32的字节序列
- 有一个.length属性，返回数组长度（只读）

变长字符数组

- 属于引用类型，包括 bytes和string，不同的是bytes是Hex字符串，而string 是UTF-8编码的字符串

枚举 (Enum)

- 枚举类型用来用户自定义一组常量值
- 与C语言的枚举类型非常相似，对应整型值

```
1 | pragma solidity >=0.4.0 <0.6.0;
2 |
3 | contract Purchase {
4 |
5 |     enum State { Created, Locked, Inactive }
6 |
7 | }
```

数组 (Array)

- 固定大小k和元素类型T的数组被写为T [k]，动态大小的数组 为T []。例如，一个由5个uint动态数组组成的数组是uint [] [5]
- 要访问第三个动态数组中的第二个uint，可以使用x [2] [1]
- 越界访问数组，会导致调用失败回退
- 如果要添加新元素，则必须使用.push () 或将.length增大
- 变长的storage数组和bytes（不包括string）有一个push()方法。可以将一个新元素附加到数组末端，返回值为当前长度
- 例子

```
1 | pragma solidity >=0.4.16 <0.6.0;
2 | contract C {
3 |     function f(uint len) public pure {
4 |         uint[] memory a = new uint[](7);
5 |         bytes memory b = new bytes(len);
6 |         assert(a.length == 7);
7 |         assert(b.length == len);
8 |         a[6] = 8;
9 |     }
10 | }
```

结构 (Struct)

- 结构类型可以在映射和数组中使用，它们本身可以包含映射和数组。
- 结构不能包含自己类型的成员，但可以作为自己数组成员的类型，也可以作为自己映射成员的值类型
-

```
1 | pragma solidity >=0.4.0 <0.6.0;
2 | contract Ballot {
3 |     struct Voter {
4 |         uint weight;
5 |         bool voted;
6 |         uint vote;
7 |     }
8 | }
```

映射 (Mapping)

- 声明一个映射：mapping (_KeyType => _)



LEVI_104

👍 0 🗨 0 ⭐ 0

- `_KeyType`可以是任何基本类型。这意味着它可以是任何内置值类型加上字节和字符串。不允许使用用户定义的或复杂的类型，如枚举，映射，`bytes`和`string`之外的任何数组类型。
- `_ValueType`可以是任何类型，包括映射。
- ```

1 | pragma solidity >=0.4.0 <0.6.0;
2 | contract MappingExample {
3 | mapping(address => uint) public balances;
4 | function update(uint newBalance) public {
5 | balances[msg.sender] = newBalance;
6 | } }
7 |
8 | contract MappingUser {
9 | function f() public returns (uint) {
10 | MappingExample m = new MappingExample();
11 | m.update(100);
12 | return m.balances(address(this));
13 | } }

```

## Solidity数据位置

- 所有的复杂类型，即数组、结构和映射类型，都有一个额外属性，“数据位置”，用来说明数据是保存在内存 `memory` 中还是存储 `storage` 中
- 根据上下文不同，大多数时候数据有默认的位置，但也可以通过在类型名后增加关键字 `storage` 或 `memory` 进行修改
- 函数参数（包括返回的参数）的数据位置默认是 `memory`，局部变量的数据位置默认是 `storage`，状态变量的数据位置强制是 `storage`
- 另外还存在第三种数据位置，`calldata`，这是一块只读的，且不会永久存储的位置，用来存储函数参数。外部函数的参数（非返回参数）的强制指定为 `calldata`，效果跟 `memory` 差不多

## 数据位置总结

### 强制指定的数据位置

- 外部函数的参数（不包括返回参数）：`calldata`;
- 状态变量：`storage`

### 默认数据位置

- 函数参数（包括返回参数）：`memory`;
- 引用类型的局部变量：`storage`
- 值类型的局部变量：栈（`stack`）

### 特别要求

- 公开可见（`publicly visible`）的函数参数一定是 `memory` 类型，如果要求是 `storage` 类型则必须是 `private` 或者 `internal` 函数，这是为了防止滥用占用资源
- 例子：

```

1 | pragma solidity ^0.4.0;
2 | contract C {
3 | uint[] data1;
4 | uint[] data2;
5 | function appendOne() public {
6 | append(data1);
7 | }
8 | function appendTwo() public {
9 | append(data2);
10 | }
11 | function append(uint[] storage d) internal {
12 | d.push(1);
13 | }
14 | }

```

```

1 | // // 下面代码包含一个错误
2 | pragma solidity ^0.4.0;
3 | contract C {

```



LEVI\_104

0
 
 0
 0

```

4 | uint someVariable; 5 | uint[] data;
6 | function f() public {
7 | uint[] x;
8 | x.push(2);
9 | data = x;
10 | }
11 | }

```

```

1 | // 下面代码编译错误
2 | pragma solidity ^0.4.0;
3 | contract C {
4 | uint[] x;
5 | function f(uint[] memoryArray) public {
6 | x = memoryArray;
7 | uint[] y = x;
8 | y[7];
9 | y.length = 2;
10 | delete x;
11 | y = memoryArray;
12 | delete y;
13 | g(x);
14 | h(x);
15 | }
16 | function g(uint[] storage storageArray) internal {}
17 | function h(uint[] memoryArray) public {} }

```

```

1 | // 下面我们一起来玩一个猜数字游戏
2 | pragma solidity >0.4.22;
3 | contract HoneyPot{
4 | uint luckyNum = 52;
5 | uint public last;
6 | struct Guess{ address player; uint number; }
7 | Guess[] public guessHistory;
8 | address owner = msg.sender;
9 | function guess(uint _num) public payable{
10 | Guess newGuess;
11 | newGuess.player = msg.sender;
12 | newGuess.number = _num;
13 | guessHistory.push(newGuess);
14 | if(_num == luckyNum)
15 | msg.sender.transfer(msg.value * 2);
16 | last = now;
17 | }
18 | }

```

## Solidity函数声明和类型

函数名称      函数类型      返回类型

```

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

```

CSDN @LEVI\_104

图1：来自尚硅谷

函数的值类型有两类：- 内部（internal）函数和 外部（external）函数

- 内部函数只能在当前合约内被调用（更具体来说，在当前代码块内，包括内部库函数和继承的函数中），因为它们不能在当前合约上下文的外部调用一个内部函数是通过跳转到它的入口标签来实现的，就像在当前合约的内部调用一个函数。
- 外部函数由一个地址和一个函数签名组成，可以通过外部函数调用传递或者 返回
- 调用内部函数：直接使用名字 f
- 调用外部函数：this.f（当前合约），a.f（外部合约）

## Solidity函数可见性



LEVI\_104

👍 0 🗨 0 ⭐ 0

函数的可见性可以指定为 `external` , `public` , `internal` 或者 `private` ; 对于状态变量, 不能设置为 `external` , 默认是 `internal`。

- **external** : 外部函数作为合约接口的一部分, 意味着我们可以从其他合约和交易中调用。一个外部函数 `f` 不能从内部调用 (即 `f` 不起作用, 但以)。当收到大量数据的时候, 外部函数有时候会更有效率。
- **public** : `public` 函数是合约接口的一部分, 可以在内部或通过消息调用。对于 `public` 状态变量, 会自动生成一个 `getter` 函数。
- **internal** : 这些函数和状态变量只能是内部访问 (即从当前合约内部或 从它派生的合约访问), 不使用 `this` 调用。
- **private** : `private` 函数和状态变量仅在当前定义它们的合约中使用, 并且不能被派生合约使用。

```

1 // 以下代码编译错误
2 pragma solidity >=0.4.0 <0.6.0;
3
4 contract C {
5 uint private data;
6 function f(uint a) private pure returns(uint b) { return a + 1; }
7 function setData(uint a) public { data = a; }
8 function getData() public view returns(uint) { return data; }
9 function compute(uint a, uint b) internal pure returns (uint) { return a + b; }
10 }
11
12 contract D {
13 function readData() public {
14 C c = new C();
15 uint local = c.f(7);
16 c.setData(3);
17 local = c.getData();
18 local = c.compute(3, 5);
19 }
20 }
21
22 contract E is C {
23 function g() public {
24 C c = new C();
25 uint val = compute(3, 5);
26 }
27 }

```

## 函数可见性例子

```

1 pragma solidity >=0.4.16 <0.6.0;
2 contract C {
3 function f(uint a) private pure returns (uint b) {
4 return a + 1;
5 }
6 function setData(uint a) internal {
7 data = a;
8 }
9 uint public data;
10 function x() public {
11 data = 3; // 内部访问
12 uint val = this.data(); // 外部访问
13 uint val2 = f(data);
14 }
15 }

```

## Solidity函数状态可变性

- **pure**: 纯函数, 不允许修改或访问状态
- **view**: 不允许修改状态
- **payable**: 允许从消息调用中接收以太币Ether。
- **constant**: 与`view`相同, 一般只修饰状态变量, 不允许赋值 (除初始化以外)

## Solidity函数状态可变性

Solidity函数状态可变性



LEVI\_104

👍 0 🗨 0 ⭐ 0

- 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。
- 使用低级调用。
- 使用包含特定操作码的内联汇编。

以下被认为是从状态中进行读取：

- 读取状态变量。
- 访问 `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。

## 函数修饰器 (modifier)

- 使用 修饰器 `modifier` 可以轻松改变函数的行为。例如，它们可以在 执行函数之前自动检查某个条件。修饰器 `modifier` 是合约的可继承属性，可以覆盖基合约覆盖
- 如果同一个函数有多个 修饰器 `modifier`，它们之间以空格隔开，修饰器 `modifier` 会依次检查执行。
- 例子：

```
1 pragma solidity >=0.4.22 <0.6.0;
2 contract Purchase {
3 address public seller;
4 modifier onlySeller() { // Modifier
5 require(msg.sender == seller, "Only seller can call.");
6 _;
7 }
8 function abort() public view onlySeller {
9 // Modifier usage
10 // ...
11 }
12 }
```

## 回退函数 (fallback)

- 回退函数 (fallback function) 是合约中的特殊函数；没有名字，不能有参数也不能有返回值
- 如果在一个到合约的调用中，没有其他函数与给定的函数标识符匹配（或没有提供调用数据），那么这个函数 (fallback 函数) 会被执行
- 每当合约收到以太币（没有任何数据），回退函数就会执行。此外，为了接收以太币，fallback 函数必须标记为 `payable`。如果不存在这样的函数，合约不能通过常规交易接收以太币
- 在上下文中通常只有很少的 gas 可以用来完成回退函数的调用，所以使 fallback 函数的调用尽量廉价很重要
- 例子：

```
1 pragma solidity >0.4.99 <0.6.0;
2
3 contract Sink {
4 function() external payable { }
5 }
6
7 contract Test {
8 function() external { x = 1; }
9 uint x;
10 }
11
12 contract Caller {
13 function callTest(Test test)
14 (bool success,) =
```



LEVI\_104

👍 0 🗨 0 ⭐ 0

```
15 | address(test).call(abi.encodeWithSignature("nonExistingFunction()"));16 | require(success);
17 | address payable testPayable = address(uint160(address(test)));
18 | return testPayable.send(2 ether);
19 | }
20 | }
```

事件 (event)

- 事件是以太坊EVM提供的一种日志基础设施。事件可以用来做操作记录，存储为日志。也可以用来实现一些交互功能，比如通知UI，返回函数
- 当定义的事件触发时，我们可以将事件存储到EVM的交易日志中，日志是区块链中的一种特殊数据结构；日志与合约关联，与合约的存储合并中；只要某个区块可以访问，其相关的日志就可以访问；但在合约中，我们不能直接访问日志和事件数据
- 可以通过日志实现简单支付验证 SPV (Simplified Payment Verification)，如果一个外部实体提供了一个带有这种证明的合约，它可以检查日志存在于区块链中

Solidity异常处理

- Solidity使用“状态恢复异常”来处理异常。这样的异常将撤消对当前调用（及其所有子调用）中的状态所做的所有更改，并且向调用者返回错误
- 函数assert和require可用于判断条件，并在不满足条件时抛出异常
- assert() 一般只应用于测试内部错误，并检查常量
- require() 应用于确保满足有效条件（如输入或合约状态变量），或验证调用外部合约的返回值
- revert() 用于抛出异常，它可以标记一个错误并将当前调用回退

Solidity中的单位

以太币 (ether)

- 以太币 Ether 单位之间的换算就是在数字后边加上 wei、finney、szabo 或 ether 来实现的，如果后面没有单位，缺省为 Wei。例如 2 ether = 2000000000000000000 wei 的逻辑判断值为 true

| Unit                | Wei Value | Wei                                         |
|---------------------|-----------|---------------------------------------------|
| wei                 | 1         | 1 wei                                       |
| Kwei (babbage)      | 1e3 wei   | 1,000                                       |
| Mwei (lovelace)     | 1e6 wei   | 1,000,000                                   |
| Gwei (shannon)      | 1e9 wei   | 1,000,000,000                               |
| microether (szabo)  | 1e12 wei  | 1,000,000,000,000                           |
| milliether (finney) | 1e15 wei  | 1,000,000,000,000,000                       |
| ether               | 1e18 wei  | 1,000,000,000,000,000,000 <small>CS</small> |

图2：来自尚硅谷

时间

秒是缺省时间单位，在时间单位之间，数字后面带有 seconds、minutes、hours、days、weeks 和 years 的可以进行换算，基本换算关系如下：

- 1 == 1 seconds
- 1 minutes == 60 seconds
- 1 hours == 60 minutes
- 1 days == 24 hours
- 1 weeks == 7 days
- 1 years == 365 days

这些后缀不能直接用在变量后边。如果想用时间

 LEVI\_104

👍 0

👎 0

💬 0




🌟 0



```
1 function f(uint start, uint daysAfter) public {
2 if (now >= start + daysAfter * 1 days) {
3 // ...
4 }
5 }
```

“相关推荐”对你有帮助么？

-  非常没帮助
-  没帮助
-  一般
-  有帮助
-  非常有帮助

关于我们 招贤纳士 商务合作 寻求报道  400-660-0108  kefu@csdn.net  在线客服 工作时间 8:30-22:00

公安备案号11010502030143 京ICP备19004658号 京网文〔2020〕1039-165号 经营性网站备案信息 北京互联网违法和不良信息举报中心 家长监护 网络110报警服务 中国互联网举报中心 Chrome商店下载 ©1999-2022北京创新乐知网络技术有限公司 版权与免责声明 版权申诉 出版物许可证 营业执照



LEVI\_104

 0  0  0  0