Solidity面向对象编程

一、Solidity基本语法

1.1 通过solidity实现一个类

编写智能合约可以直接在http://remix.ethereum.org里编写。

```
pragma solidity ^0.4.4;
/*
pragma:版本声明
solidity:开发语言
0.4.4:当前合约的版本,0.4代表主版本,.4代表修复bug的升级版
^:代表向上兼容,0.4.4~0.4.9可以对我们当前的代码进行编译
*/

contract Person {
    uint _height;
    uint _age;
    address _owner; //合约拥有者

    //方法名和合约名相同就属于构造函数
    function Person() {
        _height = 180;
        _age = 23;
        _owner = msg.sender;
    }

    function owner() constant returns (address) {
        return _owner;
    }

    //set方法，可以修改_height属性
    function setHeight(uint height) {
        _height = height;
    }

    //读取_height属性,constant代表方法只读
    function height() constant returns (uint) {
        return _height;
    }
```

```
    function setAge(uint age) {
        _age = age;
    }

    function age() constant returns (uint) {
        return _age;
    }

    function kill() {
        if (_owner == msg.sender) {
            //析构函数
            selfdestruct(msg.sender);
        }
    }
}
```

> 在网页中将Run的环境切换到JavaScript VM
>
> `msg.sender`是部署合约的钱包地址
>
> 带`constant`的方法会在一开始就被调用，因此如果在kill方法加constant在一开始就会调用它，导致销毁合约出错。

1.2 Solidity合约中属性和行为的访问权限

1.2.1 属性的访问权限

属性：状态变量

```
pragma solidity ^0.4.4;

// public > internal > private
contract Person {
    //internal 是合约属性默认的访问权限
    uint _age;
    uint _weight;
    uint private _height;
    uint public _money;

    function _money() constant returns (uint) {
        return 120;
    }
}
```

> 属性默认类型为`internal`，`internal`和`private`类型的属性都不能被外部访问，当属性的类型为`public`类型时，会生成一个和属性名相同并且返回值就是当前属性的geth函数。get函数会覆盖掉`public`类型属性自动生成的get函数。
>
> 如上述代码，_money的返回值为120，而不是0。

1.2.2 方法/行为访问权限

方法/行为：合约里面的函数。

```
pragma solidity ^0.4.4;

contract Animal {
    uint internal _age;
    uint _weight;
    uint private _height;
    uint public _money;

    // public类型
    function test() constant returns (uint) {
        return _weight;
    }

    function test1() constant public returns (uint) {
        return _height;
    }

    function test2() constant internal returns (uint) {
        return _age;
    }

    function test3() constant private returns (uint) {
        return _money;
    }

    function testInternal() constant returns (uint) {
        return this.test2();
    }

    function testPublic() constant returns (uint) {
        return this.test1();
    }

    function testPrivate() constant returns (uint) {
        return test3();
    }
}
```

> 合约方法默认权限为`public`，只有public类型的属性才可能供外部访问，internal和private类型的函数不能够通过指针进行访问，哪怕是在内部使用this进行调用也会报错，但可以直接通过方法名直接来调用。
>
> **备注**：不管是属性还是方法，只有public类型，才可以通过合约地址进行访问，合约内部的this就是当前合约的地址。

1.2.3 Solidity合约继承

- 单继承

```
contract Dog is Animal {

    function testWeight() constant returns (uint) {
        return _weight;
    }
}
```

> 子合约只可以继承`public`类型的函数，而子合约可以继承`public`和`internal`类型的属性。

- 多继承

```
contract Dog is Animal,Animal1 {

}
```

1.3 Solidity基本类型

1.3.1 值类型

- 布尔(Booleans)
- 整型(Integer)
- 地址(Address)
- 定长字节数组(fixed byte arrays)
- 有理数和整型(Rational and Integer Literals, String Literals)
- 枚举类型(Enums)
- 函数(Function Types)

> 值类型与其他语言一样，只是拷贝内容，而不是拷贝指针。

```
int a = 1;
int b = a;
b = 300;  //此时a还是1
```

1.3.2 引用类型

- 不定长字节数组(bytes)
- 字符串(String)
- 数组(Array)
- 结构体(Struct)

> 引用类型赋值时，可以是`值传递`，也可以是`地址传递`。

```
pragma solidity ^0.4.4;
```

```
contract Person {

    string _name;

    function Person(string name) {
        _name = name;
    }

    function f() {
        modify(_name);
    }

    function modify(string storage name) internal {
        bytes(name)[0] = "W";
    }

    function name() constant returns (string) {
        return _name;
    }
}
```

> 当创建合约时给`_name`赋值为“wjt”，当调用f函数后`_name`变成"Wjt"。

- string memory name此时name是值传递(默认memory)
- string storage name此时name是地址传递(如果形参是这种类型，只能内部调用此形参的函数，如`internal`或`private`)

1.3.3 各类型详解

布尔类型

bool类型可能的取值为常量`true`和`false`。

其所支持的运算符有：

- `!`逻辑非
- `&&`逻辑与
- `||`逻辑或
- `==`等于
- `!=`不等于

> 逻辑运算符`&&`和`||`与其他语言一样有`短路`现象。

整型

`int\uint`类型为边长的有符号或无符号整型。变量支持的步长以`8`递增，从int8\uint8到int256\uint256。

`var`类型可以存储任何类型，var类型变量的类型取决于值的类型。

其所支持的运算符有：

- 比较：`<=`，`<`，`==`，`!=`，`>=`，`>`
- 位运算符：`&(与)`，`|(或)`，`^(异或)`，`~(非)`
- 数学运算符：`+`，`-`，`*`，`/`，`%`，`**(a**b代表a的b次方)`

```
pragma solidity ^0.4.4;

contract Test {
    int8 a;
    int8 b;

    function Test() {
        a = 5;
        b = 8;
    }

    function addAB() constant returns (int8) {
        return a + b;
    }

    function subAB() constant returns (int8) {
        return a - b;
    }

    function mulAB() constant returns (int8) {
        return a * b;
    }

    function divAB() constant returns (int8) {
        return a / b;
    }

    function modAB() constant returns (int8) {
        return a % b;
    }

    function modAB() constant returns (int8) {
        return a ** b;
    }
}
```

- 移位运算符：`<<`，`>>`

```
pragma solidity ^0.4.4;

contract Test {

   uint8 a;
```

```
    function Test() {
        a = 3;
    }

    function leftShift(uint8 b) constant returns (uint8) {

        return a << b;
    }

    function rightShift(uint8 b) constant returns (uint8) {
        return a >> b;
    }
}
```

- 字面量

  所谓的字面量就是，在计算过程中数值的大小可以随意写(尽管超过数值的范围)，只要最后的结果在数值范围内就可以。

```
pragma solidity ^0.4.4;

contract Test {

  function test() constant returns (uint) {

      return 999999999999999999999 - 999999999999999999999;
  }
}
```

1.4 Solidity地址类型

1.4.1 以太坊钱包地址

以太坊中的地址长度为20字节，一共160位，所以`address`类型其实可以用`uint160`类型来声明。

1.4.2 msg.sender和this

`msg.sender`是当前使用者合约的地址。

```
pragma solidity ^0.4.4;

contract Test {

    address public _owner;

    uint public _number;

    function Test() {
        _owner = msg.sender;
```

```
        _number = 100;
    }

    function msgSenderAddress() constant returns (address) {
        return msg.sender;
    }

    function setNumberAdd1() {
        _number = _number + 5;
    }

    function setNumberAdd2() {
    //此时如果使用非合约发布账户，msg.sender就不是合约发布账户的地址，因此与_owner不同
        if (_owner == msg.sender) {
            _number = _number + 10;
        }
    }
}
```

> 如果有部分操作只想允许合约发布者使用，可以采用if语句进行判断msg.sender是否还是合约发布者。

`this`在合约中是当前合约地址。

1.4.3 address变量运算符使用

与平常的整数运算一样，地址可以比较大小。

1.4.4 address变量的成员变量与函数

balance

如果我们需要查看一个地址的余额，我们可以使用`balance`属性进行查看。

```
pragma solidity ^0.4.4;

contract addressBalance{

    function getBalance(address addr) constant returns (uint) {
        return addr.balance;
    }

    function getCurrentAddressBalance() constant returns (uint) {
    //当前合约地址余额
        return this.balance;
    }
}
```

transfer

从合约发起方向某个地址转入以太币(单位wei)，地址无效或者合约发起方余额不足，代码会抛出异常停止转账。

```
pragma solidity ^0.4.4;

contract PayableKeyword{
    //凡是与转账有关的方法都需要添加payable关键字。
    function deposit() payable {

        //address Account1 = 0xca35b7d915458ef540ade6068dfe2f44e8fa733c;
        address Account2 = 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c;
        //当前账户向Account2账户进行转账，msg.value是全局值(即转账值)
        Account2.transfer(msg.value);
    }

    function getAccount2Balance() constant returns (uint) {

        address Account2 = 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c;
        return Account2.balance;
    }

    function getOwnerBalance() constant returns (uint) {

        address Owner = msg.sender;
        return Owner.balance;

    }
}
```

send(执行时有一些风险)

- 调用递归深度不能超过1024
- 入股gas不够，执行会失败
- 所以使用这个方法要检查成功与否
- `tansfer`相对`send`较安全

1.5 字符串类型

字符串可以通过`""`或者`''`来表示字符串的值，Solidity中的`string`不像C语言一样以`\0`结尾。

```
pragma solidity ^0.4.4;

contract StringLiterals {

    string _name;

    function StringLiterals() public {
        _name = "wjt-joten";
    }
```

```
    function setString(string name) public {
        _name = name;
    }

    function name() constant public returns (string) {
        return _name;
    }
}
```

> 在solidity中`string`字符串不能通过`length`方法获取其长度。

1.6 字节数组

1.6.1 固定大小字节数组

固定大小字节数组可以通过`byte1`，`bytes2`，……，`bytes32`来进行声明。`byte`的别名就是`byte1`。

> bytex代表是x个字节。

比较运算符和位运算符的操作与其他类型一样，这里不再说明。

索引访问

```
pragma solidity ^0.4.4;

contract C {

    bytes9 b9 = 0x6c697989994568e1a1;

    function readIndex5Byte() constant returns (byte) {
        return b9[5];
        //返回0x45
    }
}
```

获取字节数据长度

`.length`返回字节的个数(只读)。

```
pragma solidity ^0.4.4;

contract C {

    bytes9 b9 = 0x6c697989994568e1a1;

    function bb9ByteLength) constant returns (uint) {

        return b9.length;
    }
}
```

不可变深度解析

- 长度不可修改

```
pragma solidity ^0.4.4;

contract C {

    bytes9 b9 = 0x6c697989994568e1a1;

    function setNameLength(uint length) {
        //此时会报错，无法修改长度
        name.length = length;
    }
}
```

- 内部字节不可修改

```
pragma solidity ^0.4.4;

contract C {

    bytes9 b9 = 0x6c697989994568e1a1;

    function setNameFirstByte(byte b) {
        //此时会报错，无法修改长度
        name[0] = b;
    }
}
```

1.6.2 动态大小字节数组

常规字符串string转换为bytes

```
pragma solidity ^0.4.4;
```

```
contract C {
    bytes9 public g = 0x7865673876e9a72222;

    string public name = "wangjitao";

    function gByteLength() constant returns (uint) {
        return g.length;
    }
    //将string类型转换成bytes类型
    function nameBytes() constant returns (bytes) {
        return bytes(name);
    }
    //将string类型转换成bytes类型之后再获取长度
    function nameLength() constant returns (uint) {
        return bytes(name).length;
    }
    //将string类型转换成bytes类型之后再修改
    function setNameFirstByteForL(bytes1 z) {
        bytes(name)[0] = z;
    }
}
```

汉子字符串或特殊字符的字符串转换成bytes

- 特殊字符：一个特殊字符对应一个字节
- 中文字符：一个中文字符对应三个字节

bytes可变数组length使用

```
pragma solidity ^0.4.4;

contract C {

    bytes public b = new bytes(1);

    function bLength() constant returns (uint) {
        return b.length;
    }

    function setBLength(uint len) {
        b.length = len;
    }

    function setIndexByte(byte data, uint index) {
        b[index] = data;
    }

    function clearBBytes() {
        //b.length = 0;
        delete b;
```

```
    }
}
```

bytes可变数组push使用

`push`方法可以将内容添加到bytes可变数组的末尾，并且`length`也会随之改变。

```
pragma solidity ^0.4.4;

contract C {

    bytes public name = new bytes(2);

    function nameLength() constant returns (uint) {
        return name.length;
    }

    function setNameLength(uint len) {
        name.length = len;
    }

    function pushAByte(byte b) {
        name.push(b);
    }
}
```

1.6.3 字节数组总结

对比分析：

- 不可变字节数组

用`bytex`来声明的字节数组为不可变字节数组，字节不可修改，字节数组长度也不可修改。

- 可变字节数组

可通过`bytes name = new bytes(length)`定义可变字节数组，其可以修改字节数组长度和字节内容。

> 其中`length`是字节数组长度。

1.7 字节数组相关转换

1.7.1 固定大小字节数组之间的转换

固定大小字节数组我们用`bytes1~byte32`来声明，固定大小字节数组长度不可变，内容不可修改。

```
pragma solidity ^0.4.4;

contract C {
```

```
    bytes9 public b = 0x6c;

    function b1_length() constant returns (uint) {

        return b.length;
    }

    function bToBytes2() constant returns (bytes2) {

        return bytes2(b);
    }

    function bToBytes32() constant returns (bytes32) {

        return bytes32(b);
    }
}
```

1.7.2 固定大小字节数转动态大小字节数组

通过将固定大小字节数组的内容赋值给一个新的动态大小字节数组即可。

```
pragma solidity ^0.4.4;

contract C {

    bytes9 public b = 0x6c;

    function zhuanHuan() constant returns (bytes) {

        bytes memory b1 = new bytes(b.length);

        for (uint i = 0; i < b.length; i++) {

            b1[i] = b[i];
        }

        return b1;
    }
}
```

1.7.3 动态大小字节数组转string

string类型是特殊的动态字节数组，所以string类型只能和动态大小字节数组之间进行转换，不能和固定字节数字进行转换。

```
pragma solidity ^0.4.4;

contract C {
```

```
    bytes names = new bytes(2);

    function C() {

        names[0] = 0x11;
        names[1] = 0x22;
    }

    function nameToString() constant returns (string) {

        return string(names);
    }
}
```

1.7.4 固定大小字节数组转string实现

```
pragma solidity ^0.4.4;

contract C {

   function bytes32ToString(bytes32 x) constant returns (string) {
       //以二进制看该字节数组，通过每次向左移位8位，然后判断移位后的字节数组是否为全0，从而得到该字节数组开始为0的位置。
       bytes memory bytesString = new bytes(32);
       uint charCount = 0;
       for (uint i = 0; i < 32; i++) {
           byte char = byte(bytes32(uint(x) * 2 ** (8 * i)));

           if (char != 0) {
               bytesString[charCount] = char;
                charCount++;
           }
       }

       bytes memory bytesStringTrimmed = new bytes(charCount);
       for (i = 0; i < charCount; i++) {
           bytesStringTrimmed[i] = bytesString[i];
       }

       return string(bytesStringTrimmed);
   }
}
```

总结

`string`本身是一个特殊的动态字节数组，所以它只能和`bytes`之间进行转换，不能和固定大小字节数组进行直接转换，如果是固定字节大小数组，需要将其转换为动态字节大小数组才能进行转换。

1.8 Solidity数组

1.8.1 一维数组

创建可变长度数组方法一

```
pragma solidity ^0.4.4;

contract C {

   uint [] T = [1, 2, 3, 4, 5];

   function T_Length() constant returns (uint) {

       return T.Length;
   }

}
```

创建可变长度数组方法二(推荐使用)

```
 pragma solidity ^0.4.4;

 contract C {

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

     function C() {

         for (uint i = 0; i < 5; i++) {

             T[i] = i + 1;
         }
     }
 }
```

1.8.2 二维数组

不可变二维数组

```
pragma solidity ^0.4.4;

contract C {
    //solidity里的二维数组与一般语言相反，按列写
    uint [2][3] T = [[1,2], [3,4], [5,6]];

    function T_len() constant returns (uint) {
        //二维数组长度是列数
        return T.length;
    }
}
```

可变二维数组

```
pragma solidity ^0.4.4;

contract C {
    uint [2][] T1 = new uint [2][](5);

    function pushArrToT1(uint [2] _t) {

        T1.push(_t);
    }

    function T_len() constant returns (uint) {
        //二维数组长度是列数
        return T.length;
    }
}
```

1.8.3 创建Memory Arrays

创建一个长度为`length`的`memory`类型的数组可以通过`new`关键字来创建。`memory`数组一旦创建，它不可通过`length`修改其长度。

1.9 solidity枚举类型

```
pragma solidity ^0.4.4;

contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill}
    ActionChoices _choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight(ActionChoices choice) public {
        _choice = choice;
    }
```

```
    function getChoice() constant public returns (ActionChoices) {
        return _choice;
    }

    function getDefaultChoice() pure public returns (uint) {
        return uint(defaultChoice);
    }
}
```

> `ActionChoices`是一个自定义的整型，当枚举数不够多时，它默认的类型为`uint8`，当枚举数足够多时，它会自动编程`uint16`，如上`GoLeft == 0`，`GoRight = 1`，以此类推。
> 在`setGoStraight`方法中，我们传入的参数的值可以是`0-3`，当超出范围就会报错。

1.10 solidity结构体/字典类型

1.10.1 结构体

```
pragma solidity ^0.4.4;

contract Students {

    struct Person {
        uint age;
        uint stuID;
        string name;
    }
    Person [] persons = new Person[](3);
}
```

1.10.2字典/映射

`字典/映射`就是一个键值对。

> 同一个映射中，可以有多个相同的值，但是键必须具备唯一性。

```
pragma solidity ^0.4.4;

contract MappingExample {

    mapping(string => uint) name;

    function update(string s, uint newID) public {
        name[s] = newID;
    }

    function searchID(string s) constant public returns (uint) {
        return name[s];
    }
```

```
}
```

1.10.3 结构体与字典综合案例

案例是一个`集资`合约的案例，里面有两个角色，一个是投资人`Funder`，也就是`出资者`。另一个角色是运动员`Campaign`，被赞助者。一个`Funder`可以给多个`Campaign`赞助，一个`Campaign`也可以被多个`Funder`赞助。

```
pragma solidity ^0.4.4;

contract CrowdFunding {

    //0x14723a09acff6d2a60dcdf7aa4aff308fddc160c
    //0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db
    struct Funder {
        address addr;       //出资人地址
        uint amount;     //出资总额
    }

    struct Campaign {
        address beneficiary;   //受益人钱包地址
        uint fundingGoal;   //需要赞助的总额度
        uint numFunders;    //被多少人赞助
        uint amount;    //已集资总金额
        mapping (uint => Funder) funders;   //用字典存放出资人信息
    }

    uint public numCampaigns;   //集资人总数
    mapping (uint => Campaign) campaigns;   //用字典存放集资人信息

    //产生新的集资人，传入集资人钱包地址和集资需求总额
    function newCampaign(address beneficiary, uint goal) public returns
(uint campaignID) {
        campaignID = numCampaigns++;

        campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
    }

    //通过集资人ID给某个集资人赞助
    function contribute(uint campaignID) public payable {
        Campaign storage c = campaigns[campaignID];
        c.funders[c.numFunders++] = Funder({addr:msg.sender,
amount:msg.value});
        c.amount += msg.value;
        c.beneficiary.transfer(msg.value);
    }

    //判断某个集资人是否已经达到集资总额
    function checkGoalReached(uint campaignID) public view returns (bool) {
```

```
        Campaign storage c = campaigns[campaignID];
        if (c.amount < c.fundingGoal) {
            return false;
        }
        return true;
    }

}
```

1.11 solidity中单位与全局变量

1.11.1 货币单位

一个整数的后面可以跟一个单位，`ether`，`finney`，`szabo`或者`wei`。

- 1 ether == 1000 finney
- 1 ether == 1000000 szabo
- 1 ether == 10 ** 18 wei

1.11.2 时间单位

时间单位有`second`，`minutes`，`hours`，`days`，`weeks`和`years`。

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

1.11.3 特殊的变量和函数

区块和交易属性

- `block.blockhash(uint blockNumber) returns (bytes32)`:某个区块的区块链hash值。
- `block.coinbase (address)`:当前区块的挖矿地址。
- `block.difficulty (uint)`:当前区块的难度。
- `block.gaslimit (uint)`:当前区块的gaslimit。
- `block.number (uint)`:当前区块编号。
- `block.timestamp (uint)`:当前区块时间戳。
- `msg.data (bytes)`:参数。
- `msg.gas (uint)`:剩余的gas。
- `msg.sender (address)`:当前发送消息的地址。
- `msg.sig (byte4)`:方法ID。
- `msg.value (uint)`:伴随消息附带的以太币数量。
- `now (uint)`:时间戳，与block.timestamp等价。
- `tx.gasprice (uint)`:交易的gas单价。

- `tx.origin (address)`:交易发送地址。

错误处理

- `assert(bool condition)`:不满足条件，将抛出异常。
- `require(bool condition)`:不满足条件，将抛出异常。
- `revert()`:抛出异常

```
if (msg.sender != owner) { revert(); }
assert(msg.sender == owner);
requir4e(msg.sender == owner);
```

1.12 solidity元组

1.12.1 什么是元组

```
pragma solidity ^0.4.4;

contract Simple {

    function arithmetics(uint _a, uint _b) constant returns (uint, uint) {
        return (_a + _b, _a * _b);
    }

    function f() constant returns (uint, bool, uint) {
        return (7, true, 1);
    }

    function g() constant returns (uint, bool, uint) {
        var (x, b, y) = f();

        return (x, b, y);
    }
```

二、智能合约编写

2.1 truffle安装初始化

- 打开终端，执行如下命令安装truffle

```
$ sudo npm install -g truffle
```

- 创建一个truffle项目(即初始化)

```
$ truffle init
```

- 执行完初始化命令后，会产生一系列文件供用户使用，如下图

```
# joten @ wangjitaodeMBP in ~/区块链技术/truffle项目开发/demo1 [10:54:48]
$ ls
contracts         test              truffle.js
migrations        truffle-config.js
```

> `contracts`文件夹用于存储所写的智能合约代码
>
> `migrations`文件夹用于存储与智能合约相对应的`js`文件，该文件用于检测合约是否发生变化
>
> `test`文件夹用于存储测试案例
>
> `truffle-config.js`为配置文件

2.2 truffle中编写/编译智能合约

2.2.1 编写智能合约

我们可以在`contracts`文件夹中创建`.sol`文件(智能合约文件)，写如下HelloWorld文件作为测试。

```
pragma solidity ^0.4.4;

contract HelloWorld {

  function test() pure public returns (string) {

    return "HelloWorld";
  }
}
```

2.2.2 编译智能合约

- 启动truffle

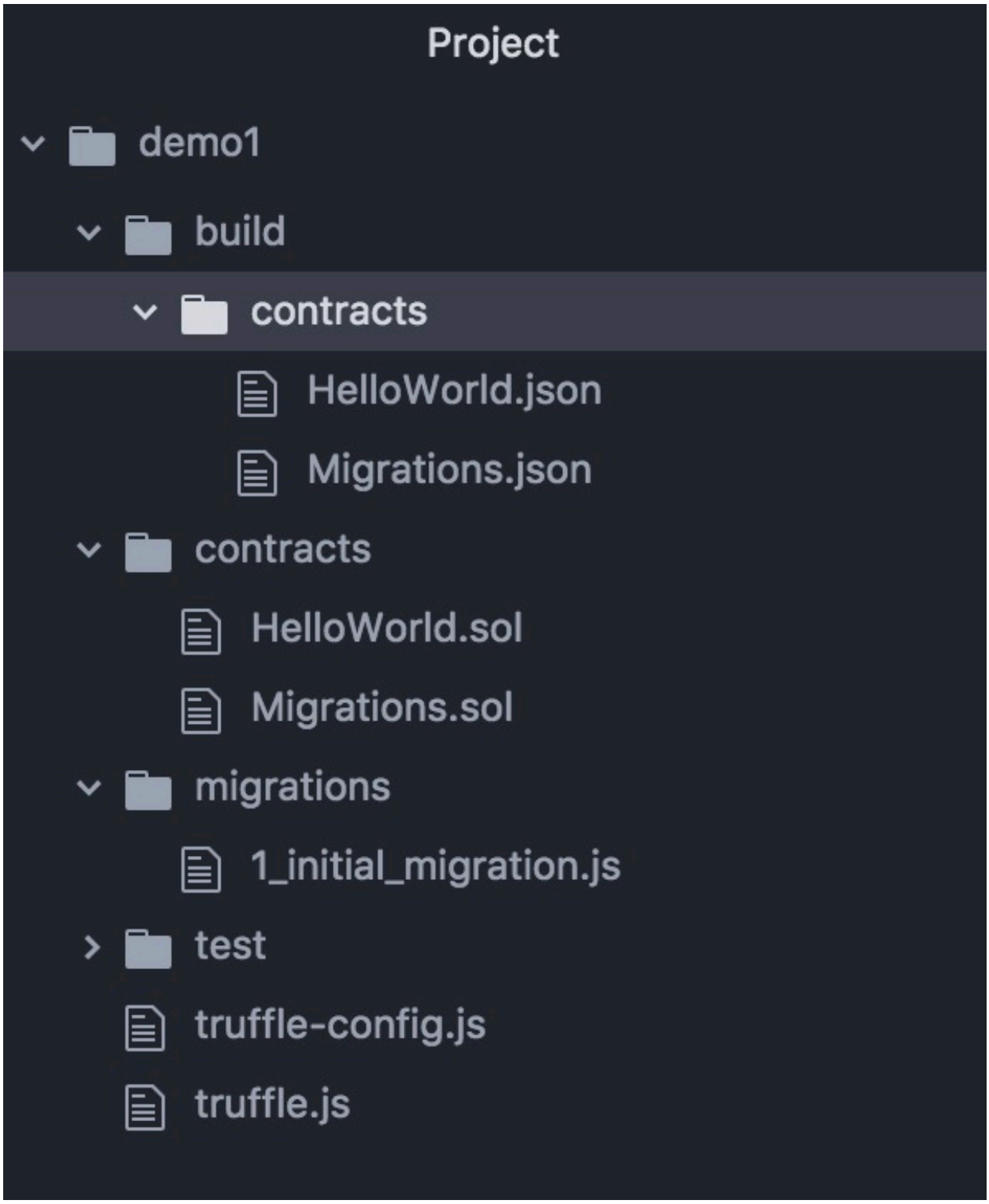
```
# joten @ wangjitaodeMBP in ~/区块链技术/truffle项目开发/demo1 [13:32:35]
$ truffle develop
Truffle Develop started at http://127.0.0.1:9545/

Accounts:
(0) 0x627306090abab3a6e1400e9345bc60c78a8bef57
(1) 0xf17f52151ebef6c7334fad080c5704d77216b732
(2) 0xc5fdf4076b8f3a5357c5e395ab970b5b54098fef
(3) 0x821aea9a577a9b44299b9c15c88cf3087f3b5544
(4) 0x0d1d4e623d10f9fba5db95830f7d3839406c6af2
(5) 0x2932b7a2355d6fecc4b5c0b6bd44cc31df247a2e
(6) 0x2191ef87e392377ec08e7c08eb105ef5448eced5
(7) 0x0f4f2ac550a1b4e2280d04c21cea7ebd822934b5
(8) 0x6330a553fc93768f612722bb8c2ec78ac90b3bbc
(9) 0x5aeda56215b167893e80b4fe645ba6d5bab767de

Private Keys:
(0) c87509a1c067bbde78beb793e6fa76530b6382a4c0241e5e4a9ec0a0f44dc0d3
(1) ae6ae8e5ccbfb04590405997ee2d52d2b330726137b875053c36d94e974d162f
(2) 0dbbe8e4ae425a6d2687f1a7e3ba17bc98c673636790f1b8ad91193c05875ef1
(3) c88b703fb08cbea894b6aeff5a544fb92e78a18e19814cd85da83b71f772aa6c
(4) 388c684f0ba1ef5017716adb5d21a053ea8e90277d0868337519f97bede61418
(5) 659cbb0e2411a44db63778987b1e22153c086a95eb6b18bdf89de078917abc63
(6) 82d052c865f5763aad42add438569276c00d3d88a2d062d36b2bae914d58b8c8
(7) aa3680d5d48a8283413f7a108367c7299ca73f553735860a87b08f39395618b7
(8) 0f62d96d6675f32685bbdb8ac13cda7c23436f63efbb9d07700d8669ff12b7c4
(9) 8d5366123cb560bb606379f90a0bfd4769eecc0557f1b362dcae9012b548b1e5
```

> 执行完该命令之后会自动生成10个账户，各自都分配有一定的以太币。

- 编译合约

```
truffle(develop)> compile
Compiling ./contracts/HelloWorld.sol...
Compiling ./contracts/Migrations.sol...
```

编译成功之后，truffle项目下会生成一个新的`build`文件夹，此时项目目录如下：



2.3 部署合约

- 在`migrations`文件夹中创建一个新的`js`文件

```
var HelloWorld = artifacts.require("./HelloWorld.sol");

module.exports = function(deployer) {
  deployer.deploy(HelloWorld);
};
```

- 在truffle终端中执行如下命令

```
truffle(develop)> migrate
Compiling ./contracts/HelloWorld.sol...
Writing artifacts to ./build/contracts

Using network 'develop'.

Running migration: 2_delopy_helloworld.js
  Deploying HelloWorld...
  ... 0xed62615c17c1e863808f64e4ab0fb92c8a5e1ddfb0bd24ff409dbdce9b9ee014
  HelloWorld: 0x345ca3e014aaf5dca488057592ee47305d9b3e10
Saving successful migration to network...
  ... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0
Saving artifacts...
```

- 创建合约变量，产生实例

```
truffle(develop)> let contract;
undefined
truffle(develop)> HelloWorld.deployed().then(instance => contract = instance)
```

- 调用方法

```
truffle(develop)> contract.test();
'HelloWorld'
```

2.4 truffle调用合约方法注意事项

我们在合约中增加两个方法，如下：

```
pragma solidity ^0.4.4;

contract HelloWorld {

  function test() pure public returns (string) {

    return "HelloWorld";
  }

  function test1() public returns (string) {

    return "WorldHello";
  }

  function echo(string s) public returns (string) {
    return s;
  }
}
```

> `test1()`和`echo()`方法我们不加`pure`关键字！！

重新编译合约产生新的合约实例，当我们调用`test()`方法时可正常输出结果，但当我们调用`test1()`方法时并不是我们想要的结果，如下：

```
truffle(develop)> contract.test();
'HelloWorld'
truffle(develop)> contract.test1();
{ tx:
   '0xc0f4c4d9e46b0a4576ca3556d8194ac657eb78ee4935be01243d098ecfcf1bf7',
  receipt:
   { transactionHash:
      '0xc0f4c4d9e46b0a4576ca3556d8194ac657eb78ee4935be01243d098ecfcf1bf7',
     transactionIndex: 0,
     blockHash:
      '0x23488feafa5560b93965891db1b858dcf5441accd6e8a49d5cd388c889ec067e',
     blockNumber: 5,
     gasUsed: 21877,
     cumulativeGasUsed: 21877,
     contractAddress: null,
     logs: [],
     status: '0x01',
     logsBloom:
      '0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000' },
  logs: [] }
```

此时我们需要利用call()方法来得到我们正确的输出结果：

```
truffle(develop)> contract.test1.call();
'WorldHello'
truffle(develop)> contract.echo.call("wjt");
'wjt'
```

> 当我们的合约方法不加 `pure` 关键字，我们调用该方法时需要利用call()方法来进行调用。

2.5 编写代币合约实例

2.5.1 代币合约基本概念

使用者在代币合约中，用自己的 `以太币账户地址` 当作银行账户，可以透过代币合约执行转账，查询余额等原本由银行负责的工作。因为合约部署在公开区块链上，所有的交易都是公开透明的，可供检验的。

2.5.2 创建代币合约项目

```
pragma solidity ^0.4.4;

contract EncryptedToken {
  uint256 INITIAL_SUPPLY = 888888;
  mapping(address => uint256) balances;

  //设置最初的代币供应量
  function EncryptedToken() {
    balances[msg.sender] = INITIAL_SUPPLY;
  }
  //转账
```

```
  function transfer(address _to, uint256 _amount) {
    assert(balances[msg.sender] >= _amount);
    balances[msg.sender] -= _amount;
    balances[_to] += _amount;
  }
  //查询代币数量
  function balanceOf(address _owner) constant returns (uint256) {
    return balances[_owner];
  }
}
```

2.6 建立标准代币部落币

- 创建一个BLC文件夹，在文件中初始化truffle目录

```
$ truffle init
$ npm init
```

- 安装`zeppelin-solidity`模块

```
$ sudo npm install zeppelin-solidity
```

- 合约代码BloggerCoin.sol

```
pragma solidity ^0.4.4;
import "/Users/joten/区块链技术/truffle项目开发/BLC/node_modules/zeppelin-
solidity/contracts/token/ERC20/StandardToken.sol";

contract BloggerCoin is StandardToken {
  string public name = "BloggerCoin";
  string public symbol = "BLC";
  uint8 public decimals = 4;
  uint256 public INITIAL_SUPPLY = 6666666;

  function BloggerCoin() {
    totalSupply_ = INITIAL_SUPPLY;
    balances[msg.sender] = INITIAL_SUPPLY;
  }
}
```

- 发布合约的js文件

```
var BloggerCoin = artifacts.require("./BloggerCoin.sol");

module.exports = function(deployer) {
  deployer.deploy(BloggerCoin);
};
```

- 编译合约(compile)/发布合约(migrate)
- 发布合约，就可以调用方法，常用方法如下：

函数	方法
totalSupply()	代币发行的总量
balanceOf(A)	查询A账户下的代币数目
transfer(A,x)	发送x个代币到A账户
transferFrom(A,x)	从A账户提取x个代币
approve(A,x)	同意A账户从我的账户中提取代币
allowance(A,B)	查询B账户可以从A账户提取多少代币