三叶草实验室“期末任务”总结

任务一：完成加密僵尸的游戏

一. Solidity Path:Beginner to Intermediate Smart Contracts

1.搭建僵尸工厂

①课程概述

第一颗要求创建一个僵尸工厂，用它建立一支僵尸部队。
僵尸的面孔由它的DNA决定，DNA由十六位数字组成，数字的不同部分对应不同的特点。
如前两位代表头型，紧接着的2位代表眼睛等。
实战演习：移动所给僵尸的不同基因的滑块，体验数字不同的僵尸的长相。
我的僵尸：头部基因（7）眼部基因（7）上衣基因（2）皮肤基因（0）
眼色基因（67）衣服颜色基因（31）

②合约

合约：一份名为HelloWorld的空合约如下

```
comtract HelloWorld{

}
```

版本指令：所有Solidity源码必须标明Solidity编译器的版本，即新项目的第一段代码，如：

```
pragma solidity ^0.4.19

comtract HelloWorld{

}
```

实战演习：

```
pragma solidity ^0.4.19

contract ZombieFactory{

}
```

③状态变量和整数

例子：

```
contract Example{
    //这个无符号整数将会永久的被保存在区块链中
    uint myUnsignedInteger = 100;
}
```

上面的例子中，定义myUnsignedInteger为uint类型，并赋值100。
uint 无符号数据类型，指其值不能是负数，对于有符号的整数存在名为int的数据类型
实战演习：

```
pragma solidity ^0.4.19

contract ZombieFactory{
    uint dnaDigits = 16;
}
```

④数学运算

Solidity中数学运算与其他程序设计语言相同
Solidity还支持乘方操作：

```
uint x = 5 ** 2;//equal to 5^2 = 25
```

实战演习：

```
pragma solidity ^0.4.19

contract ZombieFactory{
    uint dnaDigits = 16;
    uint dnaModulus = 10 ** dnaDigits;
}
```

⑤结构体

有时需要更复杂的数据类型，Sodlidity提供了结构体：

```
struct Person {
    uint age;
    string name;
}
```

结构体允许生成更复杂的数据类型，它有多个属性。
string类型用于保存任意长度的UTF-8 编码数据。
实战演习：

```
pragma solidity ^0.4.19;

contract ZombieFactory {

    uint dnaDigits = 16;
    uint dnaModulus = 10 ** dnaDigits;

    struct Zombie {
        string name;
        uint dna;
    }
}
```

⑥数组

可以使用数组这样的数据类型建立一个集合，Solidity 支持两种数组: *静态* 数组和_动态_ 数组:

```
// 固定长度为2的静态数组：
uint[2] fixedArray;
// 固定长度为5的string类型的静态数组：
string[5] stringArray;
// 动态数组，长度不固定，可以动态添加元素：
uint[] dynamicArray;
```

也可以建立一个结构体类型的数组，如上一章提到的Person：

```
Person[] people; // 这是动态数组，我们可以不断添加元素
```

公共数组(!!!)
你可以定义 public 数组, Solidity 会自动创建 getter 方法. 语法如下:

```
Person[] public people;
```

其它的合约可以从这个数组读取数据（但不能写入数据），所以这在合约中是一个有用的保存公共数据的模式。

实战演习：

```
pragma solidity ^0.4.19;

contract ZombieFactory {

    uint dnaDigits = 16;
    uint dnaModulus = 10 ** dnaDigits;

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;
}
```

⑦定义函数

在 Solidity 中函数定义的句法如下:

```
function eatHamburgers(string _name, uint _amount) {

}
```

这是一个名为 eatHamburgers 的函数，它接受两个参数：一个 string类型的 和 一个 uint类型的。现在函数内部还是空的。

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。我们整个教程都会沿用这个习惯。
实战演习：

```
pragma solidity ^0.4.19;

contract ZombieFactory {

    uint dnaDigits = 16;
    uint dnaModulus = 10 ** dnaDigits;

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;

    function createZombie(string _name, uint _dna) {

    }
}
```

⑧使用结构体和数组

现在我们学习创建新的 Person 结构，然后把它加入到名为 people 的数组中。

```
// 创建一个新的Person:
Person satoshi = Person(172, "Satoshi");

// 将新创建的satoshi添加进people数组:
people.push(satoshi);
```

你也可以两步并一步，用一行代码更简洁:

```
people.push(Person(16, "Vitalik"));
```

注：array.push()是在数组尾部添加元素，添加的顺序就是我们写代码的顺序。
实战演习：

```
pragma solidity ^0.4.19;

contract ZombieFactory {

    uint dnaDigits = 16;
    uint dnaModulus = 10 ** dnaDigits;

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;

    function createZombie(string _name, uint _dna) {
        zombies.push(Zombie(_name, _dna));
    }
}
```

⑨私有/共有函数

Solidity定义的函数的属性分为公共和私有。公共意味着任何一方（或其他合约）都可以调用你合约里的函数。私有意味着只有合约中的其他函数才能调用这个函数，定义：在函数名字后面使用关键字private即可，如：

```
uint[] numbers;

function _addToArray(uint _number) private {
  numbers.push(_number);
}
```

实战演习：

```
pragma solidity ^0.4.19;

contract ZombieFactory {

    uint dnaDigits = 16;
    uint dnaModulus = 10 ** dnaDigits;

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;

    function _createZombie(string _name, uint _dna) private {
        zombies.push(Zombie(_name, _dna));
    }
}
```

⑩函数的更多属性

Solidity 里，函数的定义里可包含返回值的数据类型(如本例中 string)。

```
string greeting = "What's up dog";

function sayHello() public returns (string) {
  return greeting;
}
```

Solidity支持view函数和pure函数
view函数，意味着它只能读取数据不能更改数据:

```
function sayHello() public view returns (string) {
```

pure 函数, 表明这个函数甚至都不访问应用里的数据，例如：

```
function _multiply(uint a, uint b) private pure returns (uint) {
  return a * b;
}
```

这个函数甚至都不读取应用里的状态 — 它的返回值完全取决于它的输入参数。
注：可能很难记住何时把函数标记为 pure/view。 幸运的是， Solidity 编辑器会给出提示，提醒你使用这些修饰符。
实战演习：

```
pragma solidity ^0.4.19;

contract ZombieFactory {

    uint dnaDigits = 16;
    uint dnaModulus = 10 ** dnaDigits;

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;

    function _createZombie(string _name, uint _dna) private {
        zombies.push(Zombie(_name, _dna));
    }

    function _generateRandomDna(string _str) private view returns (uint) {

    }
}
```

11.Keccak256 和 类型转换

如何让刚才写的_generateRandomDna 函数返回一个全(半) 随机的 uint?
Ethereum 内部有一个散列函数keccak256，它用了SHA3版本。一个散列函数基本上就是把一个字符串转换为一个256位的16进制数字。字符串的一个微小变化会引起散列数据极大变化。

```
//6e91ec6b618bb462a4a6ee5aa2cb0e9cf30f7a052bb467b0ba58b8748c00d2e5
keccak256("aaaab");
//b1f078126895a1424524de5321b339ab00408010b7cf0e6ed451514981e58aa9
keccak256("aaaac");
```

显而易见，输入字符串只改变了一个字母，输出就已经天壤之别了。

类型转换：
有时你需要变换数据类型。例如:

```
uint8 a = 5;
uint b = 6;
// 将会抛出错误，因为 a * b 返回 uint，而不是 uint8:
uint8 c = a * b;
// 我们需要将 b 转换为 uint8:
uint8 c = a * uint8(b);
```

上面, a * b 返回类型是 uint, 但是当我们尝试用 uint8 类型接收时, 就会造成潜在的错误。如果把它的数据类型转换为 uint8, 就可以了，编译器也不会出错。

实战演习：

```
function _generateRandomDna(string _str) private view returns (uint) {
        uint rand = uint(keccak256(_str));
        return rand % dnaModulus;
    }
```

12.放在一起

写一个公共函数，它有一个参数，用来接收僵尸的名字，之后用它生成僵尸的DNA。
实战演习：

```
function createRandomZombie(string _name) public {
        uint randDna = _generateRandomDna(_name);
        _createZombie(_name, randDna);
    }
```

13.事件

事件 是合约和区块链通讯的一种机制。你的前端应用“监听”某些事件，并做出反应。

例子:

```
// 这里建立事件
event IntegersAdded(uint x, uint y, uint result);

function add(uint _x, uint _y) public {
  uint result = _x + _y;
  //触发事件，通知app
  IntegersAdded(_x, _y, result);
  return result;
}
```

你的 app 前端可以监听这个事件。JavaScript 实现如下:

```
YourContract.IntegersAdded(function(error, result) {
  // 干些事
})
```

实战演习：

```
pragma solidity ^0.4.19;

contract ZombieFactory {

    event NewZombie(uint zombieId, string name, uint dna);

    uint dnaDigits = 16;
    uint dnaModulus = 10 ** dnaDigits;

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;

    function _createZombie(string _name, uint _dna) private {
        uint id = zombies.push(Zombie(_name, _dna)) - 1;
        NewZombie(id, _name, _dna);
    }

    function _generateRandomDna(string _str) private view returns (uint) {
        uint rand = uint(keccak256(_str));
        return rand % dnaModulus;
    }

    function createRandomZombie(string _name) public {
        uint randDna = _generateRandomDna(_name);
        _createZombie(_name, randDna);
    }
}
```

14.Web3.js

我们的 Solidity 合约完工了！ 现在我们要写一段 JavaScript 前端代码来调用这个合约。

以太坊有一个 JavaScript 库，名为Web3.js。

在后面的课程里，我们会进一步地教你如何安装一个合约，如何设置Web3.js。 但是现在我们通过一段代码来了解 Web3.js 是如何和我们发布的合约交互的吧。

```
// 下面是调用合约的方式：
var abi = /* abi是由编译器生成的 */
var ZombieFactoryContract = web3.eth.contract(abi)
var contractAddress = /* 发布之后在以太坊上生成的合约地址 */
var ZombieFactory = ZombieFactoryContract.at(contractAddress)
// `ZombieFactory` 能访问公共的函数以及事件

// 某个监听文本输入的监听器：
$("#ourButton").click(function(e) {
  var name = $("#nameInput").val()
  //调用合约的 `createRandomZombie` 函数：
  ZombieFactory.createRandomZombie(name)
})

// 监听 `NewZombie` 事件，并且更新UI
var event = ZombieFactory.NewZombie(function(error, result) {
  if (error) return
  generateZombie(result.zombieId, result.name, result.dna)
})

// 获取 Zombie 的 dna，更新图像
function generateZombie(id, name, dna) {
  let dnaStr = String(dna)
  // 如果dna少于16位,在它前面用0补上
  while (dnaStr.length < 16)
    dnaStr = "0" + dnaStr

  let zombieDetails = {
    // 前两位数构成头部.我们可能有7种头部，所以 % 7
    // 得到的数在0-6,再加上1,数的范围变成1-7
    // 通过这样计算：
    headChoice: dnaStr.substring(0, 2) % 7 + 1,
    // 我们得到的图片名称从head1.png 到 head7.png

    // 接下来的两位数构成眼睛，眼睛变化就对11取模：
    eyeChoice: dnaStr.substring(2, 4) % 11 + 1,
    // 再接下来的两位数构成衣服，衣服变化就对6取模：
    shirtChoice: dnaStr.substring(4, 6) % 6 + 1,
    //最后6位控制颜色．用css选择器：hue-rotate来更新
    // 360度：
    skinColorChoice: parseInt(dnaStr.substring(6, 8) / 100 * 360),
    eyeColorChoice: parseInt(dnaStr.substring(8, 10) / 100 * 360),
    clothesColorChoice: parseInt(dnaStr.substring(10, 12) / 100 * 360),
    zombieName: name,
    zombieDescription: "A Level 1 CryptoZombie",
  }
  return zombieDetails
}
```

我们的 JavaScript 所做的就是获取由zombieDetails 产生的数据, 并且利用浏览器里的 JavaScript 神奇功能 (我们用 Vue.js)，置换出图像以及使用CSS过滤器。在后面的课程中，你可以看到全部的代码。

试一下吧：僵尸名称：2021131104

2.僵尸攻击人类

①第二课概览

僵尸猎食
僵尸猎食的时候，僵尸病毒侵入猎物，这些病毒会将猎物变为新的僵尸，加入你的僵尸大军。系统会通过猎物和猎食者僵尸的DNA计算出新僵尸的DNA。

②映射（Mapping）和地址（Address）

任务二：完成对重入漏洞原理的分析，复现，并提出修复方案

1.

心得

1.

游戏第一章第一课的第八小节的实战演习中
误把 zombies.push(Zombie(_name, _dna)); 写成了zombies.push(Zombie(name, dna));

2.

游戏第一章第一课的第九小节的实战演习中
误把 function createZombie(string _name, uint _dna) private {; 写成了function _createZombie(string _name, uint *dna) private {;*
总结：没在函数名字前加

3.

游戏第一章第一课的第十一小节的实战演习中
把uint rand = uint(keccak256(_str));
写成了uint rand = (uint)keccak256(_str);