三叶草实验室“期末任务”总结（续）(第二课，第三课)

任务一：完成加密僵尸的游戏

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）

我们需要引入2个新的数据类型：mapping（映射） 和 address（地址）。

Addresses （地址）
以太坊区块链由 _ account _ (账户)组成，你可以把它想象成银行账户。一个帐户的余额是 *以太* （在以太坊区块链上使用的币种），你可以和其他帐户之间支付和接受以太币，就像你的银行帐户可以电汇资金到其他银行帐户一样。

每个帐户都有一个“地址”，你可以把它想象成银行账号。这是账户唯一的标识符，它看起来长这样：

0x0cE446255506E92DF41614C46F1d6df9Cc969183

我们将在后面的课程中介绍地址的细节，现在你只需要了解地址属于特定用户（或智能合约）的。

所以我们可以指定“地址”作为僵尸主人的 ID。当用户通过与我们的应用程序交互来创建新的僵尸时，新僵尸的所有权被设置到调用者的以太坊地址下。

Mapping（映射）
映射是这样定义的：

```
//对于金融应用程序，将用户的余额保存在一个 uint类型的变量中：
mapping (address => uint) public accountBalance;
//或者可以用来通过userId 存储/查找的用户名
mapping (uint => string) userIdToName;
```

映射本质上是存储和查找数据所用的键-值对。在第一个例子中，键是一个 address，值是一个 uint，在第二个例子中，键是一个uint，值是一个 string。

实战演习:

```
mapping (uint => address) public zombieToOwner;
mapping (address => uint) ownerZombieCount;
```

③Msg.sender

在 Solidity 中，有一些全局变量可以被所有函数调用。 其中一个就是 msg.sender，它指的是当前调用者（或智能合约）的 address。

注意：在 Solidity 中，功能执行始终需要从外部调用者开始。 一个合约只会在区块链上什么也不做，除非有人调用其中的函数。所以 msg.sender总是存在的。

以下是使用 msg.sender 来更新 mapping 的例子：

```
mapping (address => uint) favoriteNumber;

function setMyNumber(uint _myNumber) public {
  // 更新我们的 `favoriteNumber` 映射来将 `_myNumber`存储在 `msg.sender`名下
  favoriteNumber[msg.sender] = _myNumber;
  // 存储数据至映射的方法和将数据存储在数组相似
}

function whatIsMyNumber() public view returns (uint) {
  // 拿到存储在调用者地址名下的值
  // 若调用者还没调用 setMyNumber， 则值为 `0`
  return favoriteNumber[msg.sender];
}
```

在这个小小的例子中，任何人都可以调用 setMyNumber 在我们的合约中存下一个 uint 并且与他们的地址相绑定。 然后，他们调用 whatIsMyNumber 就会返回他们存储的 uint。

使用 msg.sender 很安全，因为它具有以太坊区块链的安全保障 —— 除非窃取与以太坊地址相关联的私钥，否则是没有办法修改其他人的数据的。

实战演习

```
    mapping (uint => address) public zombieToOwner;
    mapping (address => uint) ownerZombieCount;

    function _createZombie(string _name, uint _dna) private {
        uint id = zombies.push(Zombie(_name, _dna)) - 1;
        zombieToOwner[id] = msg.sender;
        ownerZombieCount[msg.sender]++;
        NewZombie(id, _name, _dna);
    }
```

④Require

require使得函数在执行过程中，当不满足某些条件时抛出错误，并停止执行：

```
function sayHiToVitalik(string _name) public returns (string) {
  // 比较 _name 是否等于 "Vitalik". 如果不成立，抛出异常并终止程序
  // (敲黑板: Solidity 并不支持原生的字符串比较，我们只能通过比较
  // 两字符串的 keccak256 哈希值来进行判断)
  require(keccak256(_name) == keccak256("Vitalik"));
  // 如果返回 true，运行如下语句
  return "Hi!";
}
```

如果你这样调用函数 sayHiToVitalik（“Vitalik”）,它会返回“Hi！”。而如果调用的时候使用了其他参数，它则会抛出错误并停止执行。

因此，在调用一个函数之前，用 require 验证前置条件是非常有必要的。

实战演习

```
function createRandomZombie(string _name) public {
        require(ownerZombieCount[msg.sender] == 0);
        uint randDna = _generateRandomDna(_name);
        _createZombie(_name, randDna);
    }
```

⑤继承（Inheritance）

个让 Solidity 的代码易于管理的功能，就是合约 inheritance (继承)：

```
contract Doge {
  function catchphrase() public returns (string) {
    return "So Wow CryptoDoge";
  }
}

contract BabyDoge is Doge {
  function anotherCatchphrase() public returns (string) {
    return "Such Moon BabyDoge";
  }
}
```

由于 BabyDoge 是从 Doge 那里 inherits （继承)过来的。 这意味着当你编译和部署了 BabyDoge，它将可以访问 catchphrase() 和 anotherCatchphrase()和其他我们在 Doge 中定义的其他公共函数。
实战演习

```
contract ZombieFeeding is ZombieFactory {

}
```

⑥引入（Import)

在 Solidity 中，当你有多个文件并且想把一个文件导入另一个文件时，可以使用 import 语句：

```
import "./someothercontract.sol";

contract newContract is SomeOtherContract {

}
```

这样当我们在合约（contract）目录下有一个名为 someothercontract.sol 的文件（ ./ 就是同一目录的意思），它就会被编译器导入。

实战演习

```
pragma solidity ^0.4.19;

import "./zombiefactory.sol";

contract ZombieFeeding is ZombieFactory {

}
```

⑦Storage与Memory

在 Solidity 中，有两个地方可以存储变量 —— storage 或 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。 你可以把它想象成存储在你电脑的硬盘或是RAM中数据的关系。

大多数时候你都用不到这些关键字，默认情况下 Solidity 会自动处理它们。 状态变量（在函数之外声明的变量）默认为“存储”形式，并永久写入区块链；而在函数内部声明的变量是“内存”型的，它们函数调用结束后消失。

然而也有一些情况下，你需要手动声明存储类型，主要用于处理函数内的 _ 结构体 _ 和 _ 数组 _ 时：

```
contract SandwichFactory {
  struct Sandwich {
    string name;
    string status;
  }

  Sandwich[] sandwiches;

  function eatSandwich(uint _index) public {
    // Sandwich mySandwich = sandwiches[_index];

    // ^ 看上去很直接，不过 Solidity 将会给出警告
    // 告诉你应该明确在这里定义 `storage` 或者 `memory`。

    // 所以你应该明确定义 `storage`：
    Sandwich storage mySandwich = sandwiches[_index];
    // ...这样 `mySandwich` 是指向 `sandwiches[_index]`的指针
    // 在存储里，另外...
    mySandwich.status = "Eaten!";
    // ...这将永久把 `sandwiches[_index]` 变为区块链上的存储

    // 如果你只想要一个副本，可以使用`memory`：
    Sandwich memory anotherSandwich = sandwiches[_index + 1];
    // ...这样 `anotherSandwich` 就仅仅是一个内存里的副本了
    // 另外
    anotherSandwich.status = "Eaten!";
    // ...将仅仅修改临时变量，对 `sandwiches[_index + 1]` 没有任何影响
    // 不过你可以这样做：
    sandwiches[_index + 1] = anotherSandwich;
    // ...如果你想把副本的改动保存回区块链存储
  }
}
```

如果你还没有完全理解究竟应该使用哪一个,Solidity 编译器会发警示提醒你的。

实战演示

```
function feedAndMultiply(uint _zombieId, uint _targetDna) public {
    require(msg.sender == zombieToOwner[_zombieId]);
    Zombie storage myZombie = zombies[_zombieId];
  }
```

⑧僵尸的DNA

获取新的僵尸DNA的公式很简单：计算猎食僵尸DNA和被猎僵尸DNA之间的平均值。

例如：

```
function testDnaSplicing() public {
  uint zombieDna = 2222222222222222;
  uint targetDna = 4444444444444444;
  uint newZombieDna = (zombieDna + targetDna) / 2;
  // newZombieDna 将等于 3333333333333333
}
```

实战演习

```
function feedAndMultiply(uint _zombieId, uint _targetDna) public {
    require(msg.sender == zombieToOwner[_zombieId]);
    Zombie storage myZombie = zombies[_zombieId];
    _targetDna = _targetDna % dnaModulus;
    uint newDna = (myZombie.dna + _targetDna) / 2;
    _createZombie("NoName", newDna);
  }
```

⑨更多关于函数可见性

internal 和 external
除 public 和 private 属性之外，Solidity 还使用了另外两个描述函数可见性的修饰词：internal（内部）和 external（外部）。

internal 和 private 类似，不过， 如果某个合约继承自其父合约，这个合约即可以访问父合约中定义的“内部”函数。

external 与public 类似，只不过这些函数只能在合约之外调用 - 它们不能被合约内的其他函数调用。

声明函数 internal 或 external 类型的语法，与声明 private 和 public类 型相同：

```
contract Sandwich {
  uint private sandwichesEaten = 0;

  function eat() internal {
    sandwichesEaten++;
  }
}

contract BLT is Sandwich {
  uint private baconSandwichesEaten = 0;

  function eatWithBacon() public returns (string) {
    baconSandwichesEaten++;
    // 因为eat() 是internal 的，所以我们能在这里调用
    eat();
  }
}
```

实战演习:将 _createZombie() 函数的属性从 private 改为 internal

⑩僵尸吃什么?

与其他合约的交互：
如果我们的合约需要和区块链上的其他的合约会话，则需先定义一个 interface (接口)。

先举一个简单的栗子。 假设在区块链上有这么一个合约：

```
contract LuckyNumber {
  mapping(address => uint) numbers;

  function setNum(uint _num) public {
    numbers[msg.sender] = _num;
  }

  function getNum(address _myAddress) public view returns (uint) {
    return numbers[_myAddress];
  }
}
```

这是个很简单的合约，您可以用它存储自己的幸运号码，并将其与您的以太坊地址关联。 这样其他人就可以通过您的地址查找您的幸运号码了。

现在假设我们有一个外部合约，使用 getNum 函数可读取其中的数据。

首先，我们定义 LuckyNumber 合约的 interface：

```
contract NumberInterface {
  function getNum(address _myAddress) public view returns (uint);
}
```

请注意，这个过程虽然看起来像在定义一个合约，但其实内里不同：

首先，我们只声明了要与之交互的函数 —— 在本例中为 getNum —— 在其中我们没有使用到任何其他的函数或状态变量。

其次，我们并没有使用大括号（{ 和 }）定义函数体，我们单单用分号（;）结束了函数声明。这使它看起来像一个合约框架。

编译器就是靠这些特征认出它是一个接口的。
实战演习：

```
contract KittyInterface {
  function getKitty(uint256 _id) external view returns (
    bool isGestating,
    bool isReady,
    uint256 cooldownIndex,
    uint256 nextActionAt,
    uint256 siringWithId,
    uint256 birthTime,
    uint256 matronId,
    uint256 sireId,
    uint256 generation,
    uint256 genes
  );
}
```

⑪使用接口

继续前面 NumberInterface 的例子，我们既然将接口定义为：

```
contract NumberInterface {
  function getNum(address _myAddress) public view returns (uint);
}
```

我们可以在合约中这样使用：

```
contract MyContract {
  address NumberInterfaceAddress = 0xab38...;
  // ^ 这是FavoriteNumber合约在以太坊上的地址
  NumberInterface numberContract = NumberInterface(NumberInterfaceAddress);
  // 现在变量 `numberContract` 指向另一个合约对象

  function someFunction() public {
    // 现在我们可以调用在那个合约中声明的 `getNum`函数:
    uint num = numberContract.getNum(msg.sender);
    // ...在这儿使用 `num`变量做些什么
  }
}
```

通过这种方式，只要将您合约的可见性设置为public(公共)或external(外部)，它们就可以与以太坊区块链上的任何其他合约进行交互。

实战演习:

```
KittyInterface kittyContract = KittyInterface(ckAddress);
```

⑫处理多返回值

getKitty 是我们所看到的第一个返回多个值的函数。我们来看看是如何处理的:

```
function multipleReturns() internal returns(uint a, uint b, uint c) {
  return (1, 2, 3);
}

function processMultipleReturns() external {
  uint a;
  uint b;
  uint c;
  // 这样来做批量赋值:
  (a, b, c) = multipleReturns();
}

// 或者如果我们只想返回其中一个变量:
function getLastReturnValue() external {
  uint c;
  // 可以对其他字段留空:
  (,,c) = multipleReturns();
}
```

实战演习:

```
function feedOnKitty(uint _zombieId, uint _kittyId) public {
    uint kittyDna;
    (,,,,,,,,,kittyDna) = kittyContract.getKitty(_kittyId);
    // 并修改函数调用
    feedAndMultiply(_zombieId, kittyDna);
  }
```

⑬奖励: Kitty 基因

还记得吗，第一课中我们提到，我们目前只使用16位DNA的前12位数来指定僵尸的外观。所以现在我们可以使用最后2个数字来处理“特殊”的特征。

这样吧，把猫僵尸DNA的最后两个数字设定为99（因为猫有9条命）。所以在我们这么来写代码：如果这个僵尸是一只猫变来的，就将它DNA的最后两位数字设置为99。

if 语句
if语句的语法在 Solidity 中，与在 JavaScript 中差不多：

```
function eatBLT(string sandwich) public {
  // 看清楚了，当我们比较字符串的时候，需要比较他们的 keccak256 哈希码
  if (keccak256(sandwich) == keccak256("BLT")) {
    eat();
  }
}
```

⑭放在一起

JavaScript 实现
我们只用编译和部署 ZombieFeeding，就可以将这个合约部署到以太坊了。我们最终完成的这个合约继承自 ZombieFactory，因此它可以访问自己和父辈合约中的所有 public 方法。

我们来看一个与我们的刚部署的合约进行交互的例子， 这个例子使用了 JavaScript 和 web3.js：

```
var abi = /* abi generated by the compiler */
var ZombieFeedingContract = web3.eth.contract(abi)
var contractAddress = /* our contract address on Ethereum after deploying */
var ZombieFeeding = ZombieFeedingContract.at(contractAddress)

// 假设我们有我们的僵尸ID和要攻击的猫咪ID
let zombieId = 1;
let kittyId = 1;

// 要拿到猫咪的DNA，我们需要调用它的API。这些数据保存在它们的服务器上而不是区块链上。
// 如果一切都在区块链上，我们就不用担心它们的服务器挂了，或者它们修改了API，
// 或者因为不喜欢我们的僵尸游戏而封杀了我们
let apiUrl = "https://api.cryptokitties.co/kitties/" + kittyId
$.get(apiUrl, function(data) {
  let imgUrl = data.image_url
  // 一些显示图片的代码
})

// 当用户点击一只猫咪的时候：
$(".kittyImage").click(function(e) {
  // 调用我们合约的 `feedOnKitty` 函数
  ZombieFeeding.feedOnKitty(zombieId, kittyId)
})

// 侦听来自我们合约的新僵尸事件好来处理
ZombieFactory.NewZombie(function(error, result) {
  if (error) return
  // 这个函数用来显示僵尸：
  generateZombie(result.zombieId, result.name, result.dna)
})
```

⑮第二课完成

截至第二课完成，所有代码如下：

zombiefactory.sol

```
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;

    mapping (uint => address) public zombieToOwner;
    mapping (address => uint) ownerZombieCount;

    function _createZombie(string _name, uint _dna) internal {
        uint id = zombies.push(Zombie(_name, _dna)) - 1;
        zombieToOwner[id] = msg.sender;
        ownerZombieCount[msg.sender]++;
        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 {
        require(ownerZombieCount[msg.sender] == 0);
        uint randDna = _generateRandomDna(_name);
        randDna = randDna - randDna % 100;
        _createZombie(_name, randDna);
    }
}
```

zombiefeeding.sol

```
pragma solidity ^0.4.19;
import "./zombiefactory.sol";
contract KittyInterface {
  function getKitty(uint256 _id) external view returns (
    bool isGestating,
    bool isReady,
    uint256 cooldownIndex,
    uint256 nextActionAt,
    uint256 siringWithId,
    uint256 birthTime,
    uint256 matronId,
    uint256 sireId,
    uint256 generation,
    uint256 genes
  );
}
contract ZombieFeeding is ZombieFactory {

  address ckAddress = 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d;
  KittyInterface kittyContract = KittyInterface(ckAddress);

  function feedAndMultiply(uint _zombieId, uint _targetDna, string _species) public {
    require(msg.sender == zombieToOwner[_zombieId]);
    Zombie storage myZombie = zombies[_zombieId];
    _targetDna = _targetDna % dnaModulus;
    uint newDna = (myZombie.dna + _targetDna) / 2;
    if (keccak256(_species) == keccak256("kitty")) {
      newDna = newDna - newDna % 100 + 99;
    }
    _createZombie("NoName", newDna);
  }

  function feedOnKitty(uint _zombieId, uint _kittyId) public {
    uint kittyDna;
    (,,,,,,,,,kittyDna) = kittyContract.getKitty(_kittyId);
    feedAndMultiply(_zombieId, kittyDna, "kitty");
  }
}
```

3.高级 Solidity 理论

①智能协议的永固性

到现在为止，我们讲的 Solidity 和其他语言没有质的区别，它长得也很像 JavaScript。

但是，在有几点以太坊上的 DApp 跟普通的应用程序有着天壤之别。

第一个例子，在你把智能协议传上以太坊之后，它就变得不可更改, 这种永固性意味着你的代码永远不能被调整或更新。

你编译的程序会一直，永久的，不可更改的，存在以太坊上。这就是 Solidity 代码的安全性如此重要的一个原因。如果你的智能协议有任何漏洞，即使你发现了也无法补救。你只能让你的用户们放弃这个智能协议，然后转移到一个新的修复后的合约上。

但这恰好也是智能合约的一大优势。代码说明一切。如果你去读智能合约的代码，并验证它，你会发现，一旦函数被定义下来，每一次的运行，程序都会严格遵照函数中原有的代码逻辑一丝不苟地执行，完全不用担心函数被人篡改而得到意外的结果。

外部依赖关系：
在第2课中，我们将加密小猫（CryptoKitties）合约的地址硬编码到 DApp 中去了。有没有想过，如果加密小猫出了点问题，比方说，集体消失了会怎么样？ 虽然这种事情几乎不可能发生，但是，如果小猫没了，我们的 DApp 也会随之失效 -- 因为我们在 DApp 的代码中用“硬编码”的方式指定了加密小猫的地址，如果这个根据地址找不到小猫，我们的僵尸也就吃不到小猫了，而按照前面的描述，我们却没法修改合约去应付这个变化！

因此，我们不能硬编码，而要采用“函数”，以便于 DApp 的关键部分可以以参数形式修改。

比方说，我们不再一开始就把猎物地址给写入代码，而是写个函数 setKittyContractAddress, 运行时再设定猎物的地址，这样我们就可以随时去锁定新的猎物，也不用担心加密小猫集体消失了。

实战演习：

```
KittyInterface kittyContract;

  function setKittyContractAddress(address _address) external {
    kittyContract = KittyInterface(_address);
  }
```

②Ownable Contracts

OpenZeppelin库的Ownable 合约
下面是一个 Ownable 合约的例子： 来自 _ OpenZeppelin _ Solidity 库的 Ownable 合约。
OpenZeppelin 是主打安保和社区审查的智能合约库，您可以在自己的 DApps中引用。等把这一课学完，您不要催我们发布下一课，最好利用这个时间把 OpenZeppelin 的网站看看，保管您会学到很多东西！

把楼下这个合约读读通，是不是还有些没见过代码？别担心，我们随后会解释。

```
/**
 * @title Ownable
 * @dev The Ownable contract has an owner address, and provides basic authorization control
 * functions, this simplifies the implementation of "user permissions".
 */
contract Ownable {
  address public owner;
  event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

  /**
   * @dev The Ownable constructor sets the original `owner` of the contract to the sender
   * account.
   */
  function Ownable() public {
    owner = msg.sender;
  }

  /**
   * @dev Throws if called by any account other than the owner.
   */
  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }

  /**
   * @dev Allows the current owner to transfer control of the contract to a newOwner.
   * @param newOwner The address to transfer ownership to.
   */
  function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0));
    OwnershipTransferred(owner, newOwner);
    owner = newOwner;
  }
}
```

1. 构造函数：function Ownable()是一个 _ constructor_ (构造函数)，构造函数不是必须的，它与合约同名，构造函数一生中唯一的一次执行，就是在合约最初被创建的时候。
2. 函数修饰符：modifier onlyOwner()。 修饰符跟函数很类似，不过是用来修饰其他已有函数用的，在其他语句执行前，为它检查下先验条件。 在这个例子中，我们就可以写个修饰符 onlyOwner 检查下调用者，确保只有合约的主人才能运行本函数。我们下一章中会详细讲述修饰符，以及那个奇怪的_;。
3. indexed 关键字：别担心，我们还用不到它。

所以Ownable 合约基本都会这么干：

1. 合约创建，构造函数先行，将其 owner 设置为msg.sender（其部署者）

2. 为它加上一个修饰符 onlyOwner，它会限制陌生人的访问，将访问某些函数的权限锁定在 owner 上。
3. 允许将合约所有权转让给他人。

onlyOwner 简直人见人爱，大多数人开发自己的 Solidity DApps，都是从复制/粘贴 Ownable 开始的，从它再继承出的子类，并在之上进行功能开发。

既然我们想把 setKittyContractAddress 限制为 onlyOwner ，我们也要做同样的事情。
实战演习：

```
import "./ownable.sol";

contract ZombieFactory is Ownable {
```

③onlyOwner 函数修饰符

现在我们有了个基本版的合约 ZombieFactory 了，它继承自 Ownable 接口，我们也可以给 ZombieFeeding 加上 onlyOwner 函数修饰符。

这就是合约继承的工作原理。记得：

```
ZombieFeeding 是个 ZombieFactory
ZombieFactory 是个 Ownable
```

因此 ZombieFeeding 也是个 Ownable, 并可以通过 Ownable 接口访问父类中的函数/事件/修饰符。往后，ZombieFeeding 的继承者合约们同样也可以这么延续下去。

函数修饰符
函数修饰符看起来跟函数没什么不同，不过关键字modifier 告诉编译器，这是个modifier(修饰符)，而不是个function(函数)。它不能像函数那样被直接调用，只能被添加到函数定义的末尾，用以改变函数的行为。

咱们仔细读读 onlyOwner:

```
/**
 * @dev 调用者不是‘主人’，就会抛出异常
 */
modifier onlyOwner() {
  require(msg.sender == owner);
  _;
}
```

onlyOwner 函数修饰符是这么用的：

```
contract MyContract is Ownable {
  event LaughManiacally(string laughter);

  //注意！ `onlyOwner`上场 :
  function likeABoss() external onlyOwner {
    LaughManiacally("Muahahahaha");
  }
}
```

注意 likeABoss 函数上的 onlyOwner 修饰符。 当你调用 likeABoss 时，首先执行 onlyOwner 中的代码， 执行到 onlyOwner 中的 _; 语句时，程序再返回并执行 likeABoss 中的代码。

可见，尽管函数修饰符也可以应用到各种场合，但最常见的还是放在函数执行之前添加快速的 require 检查。

因为给函数添加了修饰符 onlyOwner，使得唯有合约的主人（也就是部署者）才能调用它。

> 注意：主人对合约享有的特权当然是正当的，不过也可能被恶意使用。比如，万一，主人添加了个后门，允许他偷走别人的僵尸呢?

> 所以非常重要的是，部署在以太坊上的 DApp，并不能保证它真正做到去中心，你需要阅读并理解它的源代码，才能防止其中没有被部署者恶意植入后门；作为开发人员，如何做到既要给自己留下修复 bug 的余地，又要尽量地放权给使用者，以便让他们放心你，从而愿意把数据放在你的 DApp 中，这确实需要个微妙的平衡。
> 实战演习：

```
// 修改这个函数：
function setKittyContractAddress(address _address) external {
//修改后：
function setKittyContractAddress(address _address) external onlyOwner {
```

④Gas

Gas - 驱动以太坊DApps的能源
在 Solidity 中，你的用户想要每次执行你的 DApp 都需要支付一定的 gas，gas 可以用以太币购买，因此，用户每次跑 DApp 都得花费以太币。

一个 DApp 收取多少 gas 取决于功能逻辑的复杂程度。每个操作背后，都在计算完成这个操作所需要的计算资源，（比如，存储数据就比做个加法运算贵得多）， 一次操作所需要花费的 gas 等于这个操作背后的所有运算花销的总和。

由于运行你的程序需要花费用户的真金白银，在以太坊中代码的编程语言，比其他任何编程语言都更强调优化。同样的功能，使用笨拙的代码开发的程序，比起经过精巧优化的代码来，运行花费更高，这显

然会给成千上万的用户带来大量不必要的开销。

为什么要用 gas 来驱动?

以太坊就像一个巨大、缓慢、但非常安全的电脑。当你运行一个程序的时候，网络上的每一个节点都在进行相同的运算，以验证它的输出 —— 这就是所谓的“去中心化” 由于数以千计的节点同时在验证着每个功能的运行，这可以确保它的数据不会被被监控，或者被刻意修改。

可能会有用户用无限循环堵塞网络，抑或用密集运算来占用大量的网络资源，为了防止这种事情的发生，以太坊的创建者为以太坊上的资源制定了价格，想要在以太坊上运算或者存储，你需要先付费。

> 注意：如果你使用侧链，倒是不一定需要付费，比如咱们在 Loom Network 上构建的 CryptoZombies 就免费。你不会想要在以太坊主网上玩儿“魔兽世界”吧？ - 所需要的 gas 可能会买到你破产。但是你可以找个算法理念不同的侧链来玩它。我们将在以后的课程中咱们会讨论到，什么样的 DApp 应该部署在太坊主链上，什么又最好放在侧链。

省 gas 的招数：结构封装 （Struct packing）

在第1课中，我们提到除了基本版的 uint 外，还有其他变种 uint：uint8，uint16，uint32等。

通常情况下我们不会考虑使用 uint 变种，因为无论如何定义 uint的大小，Solidity 为它保留256位的存储空间。例如，使用 uint8 而不是uint（uint256）不会为你节省任何 gas。

除非，把 uint 绑定到 struct 里面。

如果一个 struct 中有多个 uint，则尽可能使用较小的 uint, Solidity 会将这些 uint 打包在一起，从而占用较少的存储空间。例如：

```
struct NormalStruct {
  uint a;
  uint b;
  uint c;
}

struct MiniMe {
  uint32 a;
  uint32 b;
  uint c;
}

// 因为使用了结构打包，`mini` 比 `normal` 占用的空间更少
NormalStruct normal = NormalStruct(10, 20, 30);
MiniMe mini = MiniMe(10, 20, 30);
```

所以，当 uint 定义在一个 struct 中的时候，尽量使用最小的整数子类型以节约空间。 并且把同样类型的变量放一起（即在 struct 中将把变量按照类型依次放置），这样 Solidity 可以将存储空间最小化。例如，有两个 struct：

uint c; uint32 a; uint32 b; 和 uint32 a; uint c; uint32 b;

前者比后者需要的gas更少，因为前者把uint32放一起了。
实战演习：

```
struct Zombie {
        string name;
        uint dna;
        uint32 level;
        uint32 readyTime;
    }
```

⑤时间单位

level 属性表示僵尸的级别。以后，在我们创建的战斗系统中，打胜仗的僵尸会逐渐升级并获得更多的能力。

readyTime 稍微复杂点。我们希望增加一个“冷却周期”，表示僵尸在两次猎食或攻击之之间必须等待的时间。如果没有它，僵尸每天可能会攻击和繁殖1,000次，这样游戏就太简单了。

为了记录僵尸在下一次进击前需要等待的时间，我们使用了 Solidity 的时间单位。
时间单位
Solidity 使用自己的本地时间单位。

变量 now 将返回当前的unix时间戳（自1970年1月1日以来经过的秒数）。我写这句话时 unix 时间是1515527488。

> 注意：Unix时间传统用一个32位的整数进行存储。这会导致“2038年”问题，当这个32位的unix时间戳不够用，产生溢出，使用这个时间的遗留系统就麻烦了。所以，如果我们想让我们的 DApp 跑够20年，我们可以使用64位整数表示时间，但为此我们的用户又得支付更多的 gas。真是个两难的设计啊！

Solidity 还包含秒(seconds)，分钟(minutes)，小时(hours)，天(days)，周(weeks) 和 年(years) 等时间单位。它们都会转换成对应的秒数放入 uint 中。所以 1分钟 就是 60，1小时是 3600（60秒×60分钟），1天是86400（24小时×60分钟×60秒），以此类推。

下面是一些使用时间单位的实用案例：

```
uint lastUpdated;

// 将‘上次更新时间’设置为‘现在’
function updateTimestamp() public {
  lastUpdated = now;
}

// 如果到上次`updateTimestamp` 超过5分钟，返回 'true'
// 不到5分钟返回 'false'
function fiveMinutesHavePassed() public view returns (bool) {
  return (now >= (lastUpdated + 5 minutes));
}
```

有了这些工具，我们可以为僵尸设定“冷静时间”功能。
实战演习：

```
// 1. 在这里定义 `cooldownTime`

// 2. 修改下面这行：
        uint id = zombies.push(Zombie(_name, _dna)) - 1;
// uint cooldownTime = 1 days;
   uint id = zombies.push(Zombie(_name, _dna, 1, uint32(now + cooldownTime))) - 1;
```

⑥僵尸冷却

现在，Zombie 结构体中定义好了一个 readyTime 属性，让我们跳到 zombiefeeding.sol， 去实现一个”冷却周期定时器“。

按照以下步骤修改 feedAndMultiply：

1. ”捕猎“行为会触发僵尸的”冷却周期“
2. 僵尸在这段”冷却周期“结束前不可再捕猎小猫

这将限制僵尸，防止其无限制地捕猎小猫或者整天不停地繁殖。将来，当我们增加战斗功能时，我们同样用”冷却周期“限制僵尸之间打斗的频率。

首先，我们要定义一些辅助函数，设置并检查僵尸的 readyTime。
将结构体作为参数传入
由于结构体的存储指针可以以参数的方式传递给一个 private 或 internal 的函数，因此结构体可以在多个函数之间相互传递。

遵循这样的语法：

```
function _doStuff(Zombie storage _zombie) internal {
  // do stuff with _zombie
}
```

这样我们可以将某僵尸的引用直接传递给一个函数，而不用是通过参数传入僵尸ID后，函数再依据ID去查找。
实战演习：

```
  function _triggerCooldown(Zombie storage _zombie) internal {
    _zombie.readyTime = uint32(now + cooldownTime);
  }

  function _isReady(Zombie storage _zombie) internal view returns (bool) {
      return (_zombie.readyTime <= now);
  }
```

⑦公有函数和安全性

现在来修改 feedAndMultiply ，实现冷却周期。

回顾一下这个函数，前一课上我们将其可见性设置为public。你必须仔细地检查所有声明为 public 和 external的函数，一个个排除用户滥用它们的可能，谨防安全漏洞。请记住，如果这些函数没有类似 onlyOwner 这样的函数修饰符，用户能利用各种可能的参数去调用它们。

检查完这个函数，用户就可以直接调用这个它，并传入他们所希望的 _targetDna 或 species 。

仔细观察，这个函数只需被 feedOnKitty() 调用，因此，想要防止漏洞，最简单的方法就是设其可见性为 internal。
实战演习：

```
// 1. 使这个函数的可见性为 internal
  function feedAndMultiply(uint _zombieId, uint _targetDna, string species) internal {
    require(msg.sender == zombieToOwner[_zombieId]);
    Zombie storage myZombie = zombies[_zombieId];
    require(_isReady(myZombie));// 2. 在这里为 `_isReady` 增加一个检查
    _targetDna = _targetDna % dnaModulus;
    uint newDna = (myZombie.dna + _targetDna) / 2;
    if (keccak256(species) == keccak256("kitty")) {
      newDna = newDna - newDna % 100 + 99;
    }
    _createZombie("NoName", newDna);
    _triggerCooldown(myZombie);// 3. 调用 `_triggerCooldown`
  }
```

⑧进一步了解函数修饰符

带参数的函数修饰符

之前我们已经读过一个简单的函数修饰符了：onlyOwner。函数修饰符也可以带参数。例如：

```
// 存储用户年龄的映射
mapping (uint => uint) public age;

// 限定用户年龄的修饰符
modifier olderThan(uint _age, uint _userId) {
  require(age[_userId] >= _age);
  _;
}

// 必须年满16周岁才允许开车（至少在美国是这样的）.
// 我们可以用如下参数调用`olderThan` 修饰符：
function driveCar(uint _userId) public olderThan(16, _userId) {
  // 其余的程序逻辑
}
```

看到了吧， olderThan 修饰符可以像函数一样接收参数，是“宿主”函数 driveCar 把参数传递给它的修饰符的。

来，我们自己生产一个修饰符，通过传入的level参数来限制僵尸使用某些特殊功能。
实战演习：

```
pragma solidity ^0.4.19;

import "./zombiefeeding.sol";

contract ZombieHelper is ZombieFeeding {

  modifier aboveLevel(uint _level, uint _zombieId) {
    require(zombies[_zombieId].level >= _level);
    _;
  }

}
```

⑨僵尸修饰符

现在让我们设计一些使用 aboveLevel 修饰符的函数。

作为游戏，您得有一些措施激励玩家们去升级他们的僵尸：

- 2级以上的僵尸，玩家可给他们改名。

- 20级以上的僵尸，玩家能给他们定制的 DNA。

是实现这些功能的时候了。以下是上一课的示例代码，供参考：（第八节课的代码）
实战演习：

```
function changeName(uint _zombieId, string _newName) external aboveLevel(2, _zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    zombies[_zombieId].name = _newName;
  }

  function changeDna(uint _zombieId, uint _newDna) external aboveLevel(20, _zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    zombies[_zombieId].dna = _newDna;
  }
```

⑩利用 'View' 函数节省 Gas

现在需要添加的一个功能是：我们的 DApp 需要一个方法来查看某玩家的整个僵尸军团 - 我们称之为 getZombiesByOwner。

实现这个功能只需从区块链中读取数据，所以它可以是一个 view 函数。这让我们不得不回顾一下“gas 优化”这个重要话题。

“view” 函数不花 “gas”
当玩家从外部调用一个view函数，是不需要支付一分 gas 的。

这是因为 view 函数不会真正改变区块链上的任何数据 - 它们只是读取。因此用 view 标记一个函数，意味着告诉 web3.js，运行这个函数只需要查询你的本地以太坊节点，而不需要在区块链上创建一个事务（事务需要运行在每个节点上，因此花费 gas）。

稍后我们将介绍如何在自己的节点上设置 web3.js。但现在，你关键是要记住，在所能只读的函数上标记上表示“只读”的“external view 声明，就能为你的玩家减少在 DApp 中 gas 用量。

> 注意：如果一个 view 函数在另一个函数的内部被调用，而调用函数与 view 函数的不属于同一个合约，也会产生调用成本。这是因为如果主调函数在以太坊创建了一个事务，它仍然需要逐个节点去验证。所以标记为 view 的函数只有在外部调用时才是免费的。
> 实战演习：

```
function getZombiesByOwner(address _owner) external view returns(uint[]) {

  }
```

⑪存储非常昂贵

Solidity 使用storage(存储)是相当昂贵的，”写入“操作尤其贵。

这是因为，无论是写入还是更改一段数据， 这都将永久性地写入区块链。”永久性“啊！需要在全球数千个节点的硬盘上存入这些数据，随着区块链的增长，拷贝份数更多，存储量也就越大。这是需要成本的!

为了降低成本，不到万不得已，避免将数据写入存储。这也会导致效率低下的编程逻辑 - 比如每次调用一个函数，都需要在 memory(内存) 中重建一个数组，而不是简单地将上次计算的数组给存储下来以便快速查找。

在大多数编程语言中，遍历大数据集合都是昂贵的。但是在 Solidity 中，使用一个标记了external view的函数，遍历比 storage 要便宜太多，因为 view 函数不会产生任何花销。

我们将在下一章讨论for循环，现在我们来看一下看如何如何在内存中声明数组。

在内存中声明数组

在数组后面加上 memory关键字， 表明这个数组是仅仅在内存中创建，不需要写入外部存储，并且在函数调用结束时它就解散了。与在程序结束时把数据保存进 storage 的做法相比，内存运算可以大大节省gas开销 -- 把这数组放在view里用，完全不用花钱。

以下是申明一个内存数组的例子：

```
function getArray() external pure returns(uint[]) {
  // 初始化一个长度为3的内存数组
  uint[] memory values = new uint[](3);
  // 赋值
  values.push(1);
  values.push(2);
  values.push(3);
  // 返回数组
  return values;
}
```

这个小例子展示了一些语法规则，下一章中，我们将通过一个实际用例，展示它和 for 循环结合的做法。

> 注意：内存数组 必须 用长度参数（在本例中为3）创建。目前不支持 array.push()之类的方法调整数组大小，在未来的版本可能会支持长度修改。
> 实战演习：

```
function getZombiesByOwner(address _owner) external view returns(uint[]) {
    uint[] memory result = new uint[](ownerZombieCount[_owner]);
    return result;
  }
```

⑫For循环

在之前的章节中，我们提到过，函数中使用的数组是运行时在内存中通过 for 循环实时构建，而不是预先建立在存储中的。

为什么要这样做呢?

为了实现 getZombiesByOwner 函数，一种“无脑式”的解决方案是在 ZombieFactory 中存入”主人“和”僵尸军团“的映射。

```
mapping (address => uint[]) public ownerToZombies
```

然后我们每次创建新僵尸时，执行 ownerToZombies [owner] .push（zombieId） 将其添加到主人的僵尸数组中。而 getZombiesByOwner 函数也非常简单：

```
function getZombiesByOwner(address _owner) external view returns (uint[]) {
  return ownerToZombies[_owner];
}
```

这个做法有问题
做法倒是简单。可是如果我们需要一个函数来把一头僵尸转移到另一个主人名下（我们一定会在后面的课程中实现的），又会发生什么?

这个“换主”函数要做到：

1.将僵尸push到新主人的 ownerToZombies 数组中， 2.从旧主的 ownerToZombies 数组中移除僵尸，3.将旧主僵尸数组中“换主僵尸”之后的的每头僵尸都往前挪一位，把挪走“换主僵尸”后留下的“空槽”填上， 4.将数组长度减1。

但是第三步实在是太贵了！因为每挪动一头僵尸，我们都要执行一次写操作。如果一个主人有20头僵尸，而第一头被挪走了，那为了保持数组的顺序，我们得做19个写操作。

由于写入存储是 Solidity 中最费 gas 的操作之一，使得换主函数的每次调用都非常昂贵。更糟糕的是，每次调用的时候花费的 gas 都不同！具体还取决于用户在原主军团中的僵尸头数，以及移走的僵尸所在的位置。以至于用户都不知道应该支付多少 gas。

> 注意：当然，我们也可以把数组中最后一个僵尸往前挪来填补空槽，并将数组长度减少一。但这样每做一笔交易，都会改变僵尸军团的秩序。

由于从外部调用一个 view 函数是免费的，我们也可以在 getZombiesByOwner 函数中用一个for循环遍历整个僵尸数组，把属于某个主人的僵尸挑出来构建出僵尸数组。那么我们的 transfer 函数将会便宜得多，因为我们不需要挪动存储里的僵尸数组重新排序，总体上这个方法会更便宜，虽然有点反直觉。

使用 for 循环

for循环的语法在 Solidity 和 JavaScript 中类似。

来看一个创建偶数数组的例子：

```
function getEvens() pure external returns(uint[]) {
  uint[] memory evens = new uint[](5);
  // 在新数组中记录序列号
  uint counter = 0;
  // 在循环从1迭代到10：
  for (uint i = 1; i <= 10; i++) {
    // 如果 `i` 是偶数...
    if (i % 2 == 0) {
      // 把它加入偶数数组
      evens[counter] = i;
      //索引加一， 指向下一个空的‘even’
      counter++;
    }
  }
  return evens;
}
```

这个函数将返回一个形为 [2,4,6,8,10] 的数组。

实战演习：

```
function getZombiesByOwner(address _owner) external view returns(uint[]) {
    uint[] memory result = new uint[](ownerZombieCount[_owner]);
    uint counter = 0;
    for (uint i = 0; i < zombies.length; i++) {
      if (zombieToOwner[i] == _owner) {
        result[counter] = i;
        counter++;
      }
    }
    return result;
  }
```

⑬放在一起

让我们回顾一下：

- 添加了一种新方法来修改CryptoKitties合约
- 学会使用 onlyOwner 进行调用权限限制
- 了解了 gas 和 gas 的优化
- 为僵尸添加了 “级别” 和 “冷却周期”属性
- 当僵尸达到一定级别时，允许修改僵尸的名字和 DNA
- 最后，定义了一个函数，用以返回某个玩家的僵尸军团

⑭第三课完成

截至第三课完成，所有代码如下：
zombiehelper.sol:

```
pragma solidity ^0.4.19;

import "./zombiefeeding.sol";

contract ZombieHelper is ZombieFeeding {

  modifier aboveLevel(uint _level, uint _zombieId) {
    require(zombies[_zombieId].level >= _level);
    _;
  }

  function changeName(uint _zombieId, string _newName) external aboveLevel(2, _zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    zombies[_zombieId].name = _newName;
  }

  function changeDna(uint _zombieId, uint _newDna) external aboveLevel(20, _zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    zombies[_zombieId].dna = _newDna;
  }

  function getZombiesByOwner(address _owner) external view returns(uint[]) {
    uint[] memory result = new uint[](ownerZombieCount[_owner]);
    uint counter = 0;
    for (uint i = 0; i < zombies.length; i++) {
      if (zombieToOwner[i] == _owner) {
        result[counter] = i;
        counter++;
      }
    }
    return result;
  }

}
```

zombiefeeding.sol:

```
pragma solidity ^0.4.19;

import "./zombiefactory.sol";

contract KittyInterface {
  function getKitty(uint256 _id) external view returns (
    bool isGestating,
    bool isReady,
    uint256 cooldownIndex,
    uint256 nextActionAt,
    uint256 siringWithId,
    uint256 birthTime,
    uint256 matronId,
    uint256 sireId,
    uint256 generation,
    uint256 genes
  );
}

contract ZombieFeeding is ZombieFactory {

  KittyInterface kittyContract;

  function setKittyContractAddress(address _address) external onlyOwner {
    kittyContract = KittyInterface(_address);
  }

  function _triggerCooldown(Zombie storage _zombie) internal {
    _zombie.readyTime = uint32(now + cooldownTime);
  }

  function _isReady(Zombie storage _zombie) internal view returns (bool) {
      return (_zombie.readyTime <= now);
  }

  function feedAndMultiply(uint _zombieId, uint _targetDna, string _species) internal {
    require(msg.sender == zombieToOwner[_zombieId]);
    Zombie storage myZombie = zombies[_zombieId];
    require(_isReady(myZombie));
    _targetDna = _targetDna % dnaModulus;
    uint newDna = (myZombie.dna + _targetDna) / 2;
    if (keccak256(_species) == keccak256("kitty")) {
      newDna = newDna - newDna % 100 + 99;
    }
    _createZombie("NoName", newDna);
    _triggerCooldown(myZombie);
  }

  function feedOnKitty(uint _zombieId, uint _kittyId) public {
    uint kittyDna;
```

```
    (,,,,,,,,kittyDna) = kittyContract.getKitty(_kittyId);
    feedAndMultiply(_zombieId, kittyDna, "kitty");
  }
}
```

zombiefactory.sol:

```
pragma solidity ^0.4.19;

import "./ownable.sol";

contract ZombieFactory is Ownable {

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

    uint dnaDigits = 16;
    uint dnaModulus = 10 ** dnaDigits;
    uint cooldownTime = 1 days;

    struct Zombie {
      string name;
      uint dna;
      uint32 level;
      uint32 readyTime;
    }

    Zombie[] public zombies;

    mapping (uint => address) public zombieToOwner;
    mapping (address => uint) ownerZombieCount;

    function _createZombie(string _name, uint _dna) internal {
        uint id = zombies.push(Zombie(_name, _dna, 1, uint32(now + cooldownTime))) - 1;
        zombieToOwner[id] = msg.sender;
        ownerZombieCount[msg.sender]++;
        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 {
        require(ownerZombieCount[msg.sender] == 0);
        uint randDna = _generateRandomDna(_name);
        randDna = randDna - randDna % 100;
        _createZombie(_name, randDna);
    }

}
```

ownable.sol:

```
/**
 * @title Ownable
 * @dev The Ownable contract has an owner address, and provides basic authorization control
 * functions, this simplifies the implementation of "user permissions".
 */
contract Ownable {
  address public owner;

  event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

  /**
   * @dev The Ownable constructor sets the original `owner` of the contract to the sender
   * account.
   */
  function Ownable() public {
    owner = msg.sender;
  }


  /**
   * @dev Throws if called by any account other than the owner.
   */
  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }


  /**
   * @dev Allows the current owner to transfer control of the contract to a newOwner.
   * @param newOwner The address to transfer ownership to.
   */
  function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0));
    OwnershipTransferred(owner, newOwner);
    owner = newOwner;
  }

}
```

心得 ① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩ ⑪ ⑫ ⑬ ⑭ ⑮ ⑯ ⑰ ⑱ ⑲ ⑳

①第二课 第7节Storage与Memory

实战演习代码的第三行
require(msg.sender == zombieToOwner[_zombieId]);

== 后面为什么是 zombieToOwner[_zombieId]);
而不是 _zombieId

②第二课 第3节Msg.sender

实战演习中
zombieToOwner[id] = msg.sender;
ownerZombieCount[msg.sender]++;
的这两句代码无法理解

③第一课 第8节

游戏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));

④第一课 第9节

游戏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 {;*

⑤第一课 第11节

游戏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);

⑥第二课第6节

'_isReady'函数里为什么是'return (_zombie.readyTime <= now);'
而不是'require(_zombie.readyTime <= now);'