三叶草实验室“期末任务”总结（续）(第四课)

任务一：完成加密僵尸的游戏

4.僵尸作战系统

①可支付

截至目前，我们只接触到很少的 函数修饰符。 要记住所有的东西很难，所以我们来个概览：

1. 我们有决定函数何时和被谁调用的可见性修饰符: private 意味着它只能被合约内部调用； internal 就像 private 但是也能被继承的合约调用； external 只能从合约外部调用；最后 public 可以在任何地方调用，不管是内部还是外部。
2. 我们也有状态修饰符， 告诉我们函数如何和区块链交互: view 告诉我们运行这个函数不会更改和保存任何数据； pure 告诉我们这个函数不但不会往区块链写数据，它甚至不从区块链读取数据。这两种在被从合约外部调用的时候都不花费任何gas（但是它们在被内部其他函数调用的时候将会耗费gas）。
3. 然后我们有了自定义的 modifiers，例如在第三课学习的: onlyOwner 和 aboveLevel。 对于这些修饰符我们可以自定义其对函数的约束逻辑。

这些修饰符可以同时作用于一个函数定义上：

```
function test() external view onlyOwner anotherModifier { /* ... */ }
```

在这一章，我们来学习一个新的修饰符 payable.

payable 修饰符

payable 方法是让 Solidity 和以太坊变得如此酷的一部分 —— 它们是一种可以接收以太的特殊函数。

先放一下。当你在调用一个普通网站服务器上的API函数的时候，你无法用你的函数传送美元——你也不能传送比特币。

但是在以太坊中， 因为钱 (*以太*), 数据 (事务负载)， 以及合约代码本身都存在于以太坊。你可以在同时调用函数 并付钱给另外一个合约。

这就允许出现很多有趣的逻辑， 比如向一个合约要求支付一定的钱来运行一个函数。

来看个例子

```
contract OnlineStore {
  function buySomething() external payable {
    // 检查以确定0.001以太发送出去来运行函数:
    require(msg.value == 0.001 ether);
    // 如果为真，一些用来向函数调用者发送数字内容的逻辑
    transferThing(msg.sender);
  }
}
```

在这里，msg.value 是一种可以查看向合约发送了多少以太的方法，另外 ether 是一个内建单元。

这里发生的事是，一些人会从 web3.js 调用这个函数 (从DApp的前端)， 像这样 :

```
// 假设 `OnlineStore` 在以太坊上指向你的合约:
OnlineStore.buySomething().send(from: web3.eth.defaultAccount, value: web3.utils.toWei(0.001))
```

注意这个 value 字段， JavaScript 调用来指定发送多少(0.001)以太。如果把事务想象成一个信封，你发送到函数的参数就是信的内容。 添加一个 value 很像在信封里面放钱 —— 信件内容和钱同时发送给了接收者。

> 注意： 如果一个函数没标记为payable， 而你尝试利用上面的方法发送以太，函数将拒绝你的事务。

实战演习:

```
uint levelUpFee = 0.001 ether;

function levelUp(uint _zombieId) external payable {
    require(msg.value == levelUpFee);
    zombies[_zombieId].level++;
}
```

②提现

在你发送以太之后，它将被存储进以合约的以太坊账户中， 并冻结在哪里 —— 除非你添加一个函数来从合约中把以太提现。

你可以写一个函数来从合约中提现以太，类似这样:

```
contract GetPaid is Ownable {
  function withdraw() external onlyOwner {
    owner.transfer(this.balance);
  }
}
```

注意我们使用 Ownable 合约中的 owner 和 onlyOwner，假定它已经被引入了。

你可以通过 transfer 函数向一个地址发送以太， 然后 this.balance 将返回当前合约存储了多少以太。所以如果100个用户每人向我们支付1以太， this.balance 将是100以太。

你可以通过 transfer 向任何以太坊地址付钱。 比如，你可以有一个函数在 msg.sender 超额付款的时候给他们退钱：

```
uint itemFee = 0.001 ether;
msg.sender.transfer(msg.value - itemFee);
```

或者在一个有卖家和卖家的合约中， 你可以把卖家的地址存储起来， 当有人买了它的东西的时候，把买家支付的钱发送给它 seller.transfer(msg.value)。

有很多例子来展示什么让以太坊编程如此之酷 —— 你可以拥有一个不被任何人控制的去中心化市场。

实战演习：

```
function withdraw() external onlyOwner {
    owner.transfer(this.balance);
  }

  function setLevelUpFee(uint _fee) external onlyOwner {
    levelUpFee = _fee;
  }
```

③僵尸战斗

在我们学习了可支付函数和合约余额之后，是时候为僵尸战斗添加功能了。

遵循上一章的格式，我们新建一个攻击功能合约，并将代码放进新的文件中，引入上一个合约。
实战演习：
zombieattack.sol:

```
pragma solidity ^0.4.19;

import "./zombiehelper.sol";

contract ZombieBattle is ZombieHelper {

}
```

④随机数

用 keccak256 来制造随机数。

Solidity 中最好的随机数生成器是 keccak256 哈希函数.

我们可以这样来生成一些随机数

```
// 生成一个0到100的随机数:
uint randNonce = 0;
uint random = uint(keccak256(now, msg.sender, randNonce)) % 100;
randNonce++;
uint random2 = uint(keccak256(now, msg.sender, randNonce)) % 100;
```

这个方法首先拿到 now 的时间戳、 msg.sender、 以及一个自增数 nonce （一个仅会被使用一次的数，这样我们就不会对相同的输入值调用一次以上哈希函数了）。

然后利用 keccak 把输入的值转变为一个哈希值, 再将哈希值转换为 uint, 然后利用 % 100 来取最后两位, 就生成了一个0到100之间随机数了。

这个方法很容易被不诚实的节点攻击

在以太坊上, 当你在和一个合约上调用函数的时候, 你会把它广播给一个节点或者在网络上的 transaction 节点们。 网络上的节点将收集很多事务, 试着成为第一个解决计算密集型数学问题的人，作为“工作证明”，然后将“工作证明”(Proof of Work, PoW)和事务一起作为一个 block 发布在网络上。

一旦一个节点解决了一个PoW, 其他节点就会停止尝试解决这个 PoW, 并验证其他节点的事务列表是有效的，然后接受这个节点转而尝试解决下一个节点。

这就让我们的随机数函数变得可利用了

我们假设我们有一个硬币翻转合约——正面你赢双倍钱，反面你输掉所有的钱。假如它使用上面的方法来决定是正面还是反面 (random >= 50 算正面, random < 50 算反面)。

如果我正运行一个节点，我可以 只对我自己的节点 发布一个事务，且不分享它。 我可以运行硬币翻转方法来偷窥我的输赢 — 如果我输了，我就不把这个事务包含进我要解决的下一个区块中去。我可以一直运行这个方法，直到我赢得了硬币翻转并解决了下一个区块，然后获利。

所以我们该如何在以太坊上安全地生成随机数呢

因为区块链的全部内容对所有参与者来说是透明的， 这就让这个问题变得很难，它的解决方法不在本课程讨论范围，你可以阅读 这个 StackOverflow 上的讨论 来获得一些主意。 一个方法是利用 oracle 来访问以太坊区块链之外的随机数函数。

当然， 因为网络上成千上万的以太坊节点都在竞争解决下一个区块，我能成功解决下一个区块的几率非常之低。 这将花费我们巨大的计算资源来开发这个获利方法 — 但是如果奖励异常地高(比如我可以在硬币翻转函数中赢得 1个亿)， 那就很值得去攻击了。

所以尽管这个方法在以太坊上不安全，在实际中，除非我们的随机函数有一大笔钱在上面，你游戏的用户一般是没有足够的资源去攻击的。

因为在这个教程中，我们只是在编写一个简单的游戏来做演示，也没有真正的钱在里面，所以我们决定接受这个不足之处，使用这个简单的随机数生成函数。但是要谨记它是不安全的。

实战演习：

```
uint randNonce = 0;

  function randMod(uint _modulus) internal returns(uint) {
    randNonce++;
    return uint(keccak256(now, msg.sender, randNonce)) % _modulus;
  }
```

⑤僵尸对决

我们的合约已经有了一些随机性的来源，可以用进我们的僵尸战斗中去计算结果。

我们的僵尸战斗看起来将是这个流程：

- 你选择一个自己的僵尸，然后选择一个对手的僵尸去攻击。
- 如果你是攻击方，你将有70%的几率获胜，防守方将有30%的几率获胜。
- 所有的僵尸（攻守双方）都将有一个 winCount 和一个 lossCount，这两个值都将根据战斗结果增长。
- 若攻击方获胜，这个僵尸将升级并产生一个新僵尸。
- 如果攻击方失败，除了失败次数将加一外，什么都不会发生。
- 无论输赢，当前僵尸的冷却时间都将被激活。

这有一大堆的逻辑需要处理，我们将把这些步骤分解到接下来的课程中去。

实战演习：

```
// 在这里创建 attackVictoryProbability
uint attackVictoryProbability = 70;

// 在这里创建新函数
function attack(uint _zombieId, uint _targetId) external {
  }
```

⑥重构通用逻辑

不管谁调用我们的 attack 函数 —— 我们想确保用户的确拥有他们用来攻击的僵尸。如果你能用其他人的僵尸来攻击将是一个很大的安全问题。

你能想一下我们如何添加一个检查步骤来看看调用这个函数的人就是他们传入的 _zombieId 的拥有者么?

想一想，看看你能不能自己找到一些答案。

花点时间…… 参考我们前面课程的代码来获得灵感。

答案
我们在前面的课程里面已经做过很多次这样的检查了。 在 changeName(), changeDna(), 和 feedAndMultiply()里，我们做过这样的检查:

require(msg.sender == zombieToOwner[_zombieId]);
这和我们 attack 函数将要用到的检查逻辑是相同的。 正因我们要多次调用这个检查逻辑，让我们把它移到它自己的 modifier 中来清理代码并避免重复编码。

实战演习:

```
// 1. 在这里创建 modifier
modifier ownerOf(uint _zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    _;
  }

// 2. 在函数定义时增加 modifier :
  function feedAndMultiply(uint _zombieId, uint _targetDna, string _species) internal ownerOf(_zombieId) {
    // 3. 移除这一行
    require(msg.sender == zombieToOwner[_zombieId]);
```

⑦更多重构

在 zombiehelper.sol里有几处地方，需要我们实现我们新的 modifier—— ownerOf。

实战演习:

1. 修改 changeName() 使其使用 ownerOf
2. 修改 changeDna() 使其使用 ownerOf

在changeName()函数与changeDna()函数的aboveLevel修饰符后面
写上ownerOf(_zombieId)
并删除两个函数中的
require(msg.sender == zombieToOwner[_zombieId]);
这一行代码即可

⑧回到攻击

重构完成了，回到 zombieattack.sol。

继续来完善我们的 attack 函数， 现在我们有了 ownerOf 修饰符来用了。

实战演习：

```
function attack(uint _zombieId, uint _targetId) external ownerOf(_zombieId) {
    Zombie storage myZombie = zombies[_zombieId];
    Zombie storage enemyZombie = zombies[_targetId];
    uint rand = randMod(100);
  }
```

⑨僵尸的输赢

对我们的僵尸游戏来说，我们将要追踪我们的僵尸输赢了多少场。有了这个我们可以在游戏里维护一个"僵尸排行榜"。

有多种方法在我们的DApp里面保存一个数值 — 作为一个单独的映射，作为一个“排行榜”结构体，或者保存在 Zombie 结构体内。

每个方法都有其优缺点，取决于我们打算如何和这些数据打交道。在这个教程中，简单起见我们将这个状态保存在 Zombie 结构体中，将其命名为 winCount 和 lossCount。

我们跳回 zombiefactory.sol, 将这些属性添加进 Zombie 结构体.

实战演习：

```
struct Zombie {
      string name;
      uint dna;
      uint32 level;
      uint32 readyTime;
      uint16 winCount;
      uint16 lossCount;
    }
// 2. 在这里修改修改新僵尸的创建：
uint id = zombies.push(Zombie(_name, _dna, 1, uint32(now + cooldownTime), 0, 0)) - 1;
```

⑩僵尸胜利了😄

有了 winCount 和 lossCount，我们可以根据僵尸哪个僵尸赢了战斗来更新它们了。

在第六章我们计算出来一个0到100的随机数。现在让我们用那个数来决定那谁赢了战斗，并以此更新我们的状态。

实战演习：

```
    if (rand <= attackVictoryProbability) {
      myZombie.winCount++;
      myZombie.level++;
      enemyZombie.lossCount++;
      feedAndMultiply(_zombieId, enemyZombie.dna, "zombie");
    }
```

⑪僵尸失败😞

我们已经编写了你的僵尸赢了之后会发生什么， 该看看 输了 的时候要怎么做了。

在我们的游戏中，僵尸输了后并不会降级 —— 只是简单地给 lossCount 加一，并触发冷却，等待一天后才能再次参战。

要实现这个逻辑，我们需要一个 else 语句。

else 语句和 JavaScript 以及很多其他语言的 else 语句一样。

```
if (zombieCoins[msg.sender] > 100000000) {
  // 你好有钱!!!
} else {
  // 我们需要更多的僵尸币...
}
```

实战演习：

```
    else{
      myZombie.lossCount++;
      enemyZombie.winCount++;
      _triggerCooldown(myZombie);
    }
```

⑫放在一起

认领你的战利品

在赢了战斗之后：

1. 你的僵尸将会升级
2. 你僵尸的 winCount 将会增加
3. 你将为你的僵尸大军获得一个新的僵尸

⑬Lesson 4 Complete！

截至第四课完成，所有代码如下：

zombieattack.sol:

```
pragma solidity ^0.4.19;

import "./zombiehelper.sol";

contract ZombieBattle is ZombieHelper {
  uint randNonce = 0;
  uint attackVictoryProbability = 70;

  function randMod(uint _modulus) internal returns(uint) {
    randNonce++;
    return uint(keccak256(now, msg.sender, randNonce)) % _modulus;
  }

  function attack(uint _zombieId, uint _targetId) external ownerOf(_zombieId) {
    Zombie storage myZombie = zombies[_zombieId];
    Zombie storage enemyZombie = zombies[_targetId];
    uint rand = randMod(100);
    if (rand <= attackVictoryProbability) {
      myZombie.winCount++;
      myZombie.level++;
      enemyZombie.lossCount++;
      feedAndMultiply(_zombieId, enemyZombie.dna, "zombie");
    } else {
      myZombie.lossCount++;
      enemyZombie.winCount++;
      _triggerCooldown(myZombie);
    }
  }
}
```

zombiehelper.sol:

```
pragma solidity ^0.4.19;

import "./zombiefeeding.sol";

contract ZombieHelper is ZombieFeeding {

  uint levelUpFee = 0.001 ether;

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

  function withdraw() external onlyOwner {
    owner.transfer(this.balance);
  }

  function setLevelUpFee(uint _fee) external onlyOwner {
    levelUpFee = _fee;
  }

  function levelUp(uint _zombieId) external payable {
    require(msg.value == levelUpFee);
    zombies[_zombieId].level++;
  }

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

  function changeDna(uint _zombieId, uint _newDna) external aboveLevel(20, _zombieId) ownerOf(_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;

  modifier ownerOf(uint _zombieId) {
    require(msg.sender == zombieToOwner[_zombieId]);
    _;
  }

  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 ownerOf(_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;
      uint16 winCount;
      uint16 lossCount;
    }

    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), 0, 0)) - 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:

```
pragma solidity ^0.4.19;

/**
 * @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第四课 第⑩节

实战演习中if语句的最后一行
feedAndMultiply(_zombieId, enemyZombie.dna, "zombie");
第二个传入'feedAndMultiply'函数的参数
为什么是enemyZombie.dna