三叶草实验室“期末任务”总结（续）(第五课)

任务一：完成加密僵尸的游戏

5.ERC721标准和加密收藏品

①以太坊上的代币

让我们来聊聊 *代币*.

如果你对以太坊的世界有一些了解，你很可能听过人们聊到代币——尤其是 ERC20 代币.

一个 *代币* 在以太坊基本上就是一个遵循一些共同规则的智能合约——即它实现了所有其他代币合约共享的一组标准函数，例如 `transfer(address _to, uint256 _value)` 和 `balanceOf(address _owner)`.

在智能合约内部，通常有一个映射， `mapping(address => uint256) balances` ，用于追踪每个地址还有多少余额。

所以基本上一个代币只是一个追踪谁拥有多少该代币的合约，和一些可以让那些用户将他们的代币转移到其他地址的函数。

它为什么重要呢?
由于所有 ERC20 代币共享具有相同名称的同一组函数，它们都可以以相同的方式进行交互。

这意味着如果你构建的应用程序能够与一个 ERC20 代币进行交互，那么它就也能够与任何 ERC20 代币进行交互。 这样一来，将来你就可以轻松地将更多的代币添加到你的应用中，而无需进行自定义编码。 你可以简单地插入新的代币合约地址，然后哗啦，你的应用程序有另一个它可以使用的代币了。

其中一个例子就是交易所。 当交易所添加一个新的 ERC20 代币时，实际上它只需要添加与之对话的另一个智能合约。 用户可以让那个合约将代币发送到交易所的钱包地址，然后交易所可以让合约在用户要求取款时将代币发送回给他们。

交易所只需要实现这种转移逻辑一次，然后当它想要添加一个新的 ERC20 代币时，只需将新的合约地址添加到它的数据库即可。

其他代币标准
对于像货币一样的代币来说，ERC20 代币非常酷。 但是要在我们僵尸游戏中代表僵尸就并不是特别有用。

首先，僵尸不像货币可以分割 —— 我可以发给你 0.237 以太，但是转移给你 0.237 的僵尸听起来就有些搞笑。

其次，并不是所有僵尸都是平等的。 你的2级僵尸"Steve"完全不能等同于我732级的僵尸"H4XF13LD MORRIS 💯 💯 😎 💯 💯"。（你差得远呢，Steve）。

有另一个代币标准更适合如 CryptoZombies 这样的加密收藏品——它们被称为ERC721 代币.

ERC721 代币是不能互换的，因为每个代币都被认为是唯一且不可分割的。 你只能以整个单位交易它们，并且每个单位都有唯一的 ID。 这些特性正好让我们的僵尸可以用来交易。

> 请注意，使用像 ERC721 这样的标准的优势就是，我们不必在我们的合约中实现拍卖或托管逻辑，这决定了玩家能够如何交易 / 出售我们的僵尸。 如果我们符合规范，其他人可以为加密可交易的 ERC721 资产搭建一个交易所平台，我们的 ERC721 僵尸将可以在该平台上使用。 所以使用代币标准相较于使用你自己的交易逻辑有明显的好处。

实战演习:
zombieownership.sol:

```
pragma solidity ^0.4.19;

import "./zombieattack.sol";

contract ZombieOwnership is ZombieAttack {
}
```

②ERC721 标准, 多重继承

让我们来看一看 ERC721 标准：

```
contract ERC721 {
  event Transfer(address indexed _from, address indexed _to, uint256 _tokenId);
  event Approval(address indexed _owner, address indexed _approved, uint256 _tokenId);

  function balanceOf(address _owner) public view returns (uint256 _balance);
  function ownerOf(uint256 _tokenId) public view returns (address _owner);
  function transfer(address _to, uint256 _tokenId) public;
  function approve(address _to, uint256 _tokenId) public;
  function takeOwnership(uint256 _tokenId) public;
}
```

这是我们需要实现的方法列表，我们将在接下来的章节中逐个学习。

虽然看起来很多，但不要被吓到了！我们在这里就是准备带着你一步一步了解它们的。

> 注意： ERC721目前是一个 草稿，还没有正式商定的实现。在本教程中，我们使用的是 OpenZeppelin 库中的当前版本，但在未来正式发布之前它可能会有更改。 所以把这 一个 可能的实现当作考虑，但不要把它作为 ERC721 代币的官方标准。

实现一个代币合约

在实现一个代币合约的时候，我们首先要做的是将接口复制到它自己的 Solidity 文件并导入它， `import "./erc721.sol";` 。 接着，让我们的合约继承它，然后我们用一个函数定义来重写每个方法。

但等一下—— `ZombieOwnership` 已经继承自 `ZombieAttack` 了 —— 它如何能够也继承于 `ERC721` 呢?

幸运的是在Solidity，你的合约可以继承自多个合约，参考如下：

```
contract SatoshiNakamoto is NickSzabo, HalFinney {

}
```

正如你所见，当使用多重继承的时候，你只需要用逗号 , 来隔开几个你想要继承的合约。在上面的例子中，我们的合约继承自 `NickSzabo` 和 `HalFinney` 。

实战演习：

```
pragma solidity ^0.4.19;

import "./zombieattack.sol";
import "./erc721.sol";

contract ZombieOwnership is ZombieAttack, ERC721 {

}
```

③balanceOf 和 ownerOf

在本章节，我们将实现头两个方法： balanceOf 和 ownerOf。

balanceOf

```
function balanceOf(address _owner) public view returns (uint256 _balance);
```

这个函数只需要一个传入 address 参数，然后返回这个 address 拥有多少代币。

在我们的例子中，我们的“代币”是僵尸。你还记得在我们 DApp 的哪里存储了一个主人拥有多少只僵尸吗?

ownerOf

```
function ownerOf(uint256 _tokenId) public view returns (address _owner);
```

这个函数需要传入一个代币 ID 作为参数 (我们的情况就是一个僵尸 ID)，然后返回该代币拥有者的 address。

同样的，因为在我们的 DApp 里已经有一个 mapping (映射) 存储了这个信息，所以对我们来说这个实现非常直接清晰。我们可以只用一行 return 语句来实现这个函数。

> 注意：要记得， uint256 等同于uint。我们从课程的开始一直在代码中使用 uint，但从现在开始我们将在这里用 uint256，因为我们直接从规范中复制粘贴。

实战演习：

```
function balanceOf(address _owner) public view returns (uint256 _balance) {
  return ownerZombieCount[_owner];
}

function ownerOf(uint256 _tokenId) public view returns (address _owner) {
  return zombieToOwner[_tokenId];
}
```

④重构

我们刚刚的代码中其实有个错误，以至于其根本无法通过编译，你发现了没?

在前一个章节我们定义了一个叫 ownerOf 的函数。但如果你还记得第4课的内容，我们同样在 zombiefeeding.sol 里以 ownerOf 命名创建了一个 modifier（修饰符）。

如果你尝试编译这段代码，编译器会给你一个错误说你不能有相同名称的修饰符和函数。

所以我们应该把在 ZombieOwnership 里的函数名称改成别的吗?

不，我们不能那样做！！！要记得，我们正在用 ERC721 代币标准，意味着其他合约将期望我们的合约以这些确切的名称来定义函数。这就是这些标准实用的原因——如果另一个合约知道我们的合约符合 ERC721 标准，它可以直接与我们交互，而无需了解任何关于我们内部如何实现的细节。

所以，那意味着我们将必须重构我们第4课中的代码，将 modifier 的名称换成别的。

实战演习：
我们回到了 zombiefeeding.sol 。我们将把 modifier 的名称从 ownerOf 改成 onlyOwnerOf。

1. 把修饰符定义中的名称改成 onlyOwnerOf
2. 往下滑到使用此修饰符的函数 feedAndMultiply 。我们也需要改这里的名称。

> 注意：我们在 zombiehelper.sol 和 zombieattack.sol 里也使用了这个修饰符，但为了不在这节课的重构里花太多时间，我们已经将那些文件里的修饰符名称为你改好了。

⑤ERC721: 转移标准

现在我们将通过学习把所有权从一个人转移给另一个人来继续我们的 ERC721 规范的实现。

注意 ERC721 规范有两种不同的方法来转移代币：

```
function transfer(address _to, uint256 _tokenId) public;

function approve(address _to, uint256 _tokenId) public;
function takeOwnership(uint256 _tokenId) public;
```

1. 第一种方法是代币的拥有者调用transfer 方法，传入他想转移到的 address 和他想转移的代币的 _tokenId。
2. 第二种方法是代币拥有者首先调用 approve，然后传入与以上相同的参数。接着，该合约会存储谁被允许提取代币，通常存储到一个 mapping (uint256 => address) 里。然后，当有人调用 takeOwnership 时，合约会检查 msg.sender 是否得到拥有者的批准来提取代币，如果是，则将代币转移给他。

你注意到了吗，transfer 和 takeOwnership 都将包含相同的转移逻辑，只是以相反的顺序。 （一种情况是代币的发送者调用函数；另一种情况是代币的接收者调用它）。

所以我们把这个逻辑抽象成它自己的私有函数 _transfer，然后由这两个函数来调用它。 这样我们就不用写重复的代码了。

实战演习:

```
function _transfer(address _from, address _to, uint256 _tokenId) private {
    ownerZombieCount[_to]++;
    ownerZombieCount[_from]--;
    zombieToOwner[_tokenId] = _to;
    Transfer(_from, _to, _tokenId);
  }
```

⑥ERC721: 转移-续

实战演习：

```
function transfer(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenId) {
    _transfer(msg.sender, _to, _tokenId);
  }
```

⑦ERC721: 批准

现在，让我们来实现 approve。

记住，使用 approve 或者 takeOwnership 的时候，转移有2个步骤：

1. 你，作为所有者，用新主人的 address 和你希望他获取的 _tokenId 来调用 approve
2. 新主人用 _tokenId 来调用 takeOwnership，合约会检查确保他获得了批准，然后把代币转移给他。

因为这发生在2个函数的调用中，所以在函数调用之间，我们需要一个数据结构来存储什么人被批准获取什么。

实战演习：

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

function approve(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenId) {
    zombieApprovals[_tokenId] = _to;
    Approval(msg.sender, _to, _tokenId);
  }
```

⑧ERC721: takeOwnership

最后一个函数 takeOwnership， 应该只是简单地检查以确保 msg.sender 已经被批准来提取这个代币或者僵尸。若确认，就调用 _transfer；

实战演习：

```
function takeOwnership(uint256 _tokenId) public {
    require(zombieApprovals[_tokenId] == msg.sender);
    address owner = ownerOf(_tokenId);
    _transfer(owner, msg.sender, _tokenId);
  }
```

⑨预防溢出

我们完成了 ERC721 的实现。

不过要记住那只是最简单的实现。还有很多的特性我们也许想加入到我们的实现中来，比如一些额外的检查，来确保用户不会不小心把他们的僵尸转移给0 地址（这被称作 “烧币”, 基本上就是把代币转移到一个谁也没有私钥的地址，让这个代币永远也无法恢复）。 或者在 DApp 中加入一些基本的拍卖逻辑。（你能想出一些实现的方法么？）

但是为了让我们的课程不至于离题太远，所以我们只专注于一些基础实现。如果你想学习一些更深层次的实现，可以在这个教程结束后，去看看 OpenZeppelin 的 ERC721 合约。

合约安全增强: 溢出和下溢

我们将来学习你在编写智能合约的时候需要注意的一个主要的安全特性：防止溢出和下溢。

什么是 *溢出* (overflow)?

假设我们有一个 uint8, 只能存储8 bit数据。这意味着我们能存储的最大数字就是二进制 11111111 (或者说十进制的 2^8 - 1 = 255).

来看看下面的代码。最后 number 将会是什么值?

```
uint8 number = 255;
number++;
```

在这个例子中，我们导致了溢出 — 虽然我们加了1， 但是 number 出乎意料地等于 0了。 (如果你给二进制 11111111 加1, 它将被重置为 00000000，就像钟表从 23:59 走向 00:00)。

下溢(underflow)也类似，如果你从一个等于 0 的 uint8 减去 1, 它将变成 255 (因为 uint 是无符号的，其不能等于负数)。

虽然我们在这里不使用 uint8，而且每次给一个 uint256 加 1 也不太可能溢出 (2^256 真的是一个很大的数了)，在我们的合约中添加一些保护机制依然是非常有必要的，以防我们的 DApp 以后出现什么异常情况。

使用 SafeMath

为了防止这些情况，OpenZeppelin 建立了一个叫做 SafeMath 的 *库*(library)，默认情况下可以防止这些问题。

不过在我们使用之前…… 什么叫做库?

一个_库_ 是 Solidity 中一种特殊的合约。其中一个有用的功能是给原始数据类型增加一些方法。

比如，使用 SafeMath 库的时候，我们将使用 using SafeMath for uint256 这样的语法。 SafeMath 库有四个方法 — add， sub， mul， 以及 div。现在我们可以这样来让 uint256 调用这些方法：

```
using SafeMath for uint256;

uint256 a = 5;
uint256 b = a.add(3); // 5 + 3 = 8
uint256 c = a.mul(2); // 5 * 2 = 10
```

我们将在下一章来学习这些方法，不过现在我们先将 SafeMath 库添加进我们的合约。

实战演习：

1. 将 safemath.sol 引入到 zombiefactory.sol.
2. 添加定义： using SafeMath for uint256;.

⑩SafeMath第二部分

来看看 SafeMath 的部分代码:

```
library SafeMath {

  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
  }

  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold
    return c;
  }

  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
  }

  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}
```

首先我们有了 library 关键字 — 库和 合约很相似，但是又有一些不同。 就我们的目的而言，库允许我们使用 using 关键字，它可以自动把库的所有方法添加给一个数据类型：

```
using SafeMath for uint;
// 这下我们可以为任何 uint 调用这些方法了
uint test = 2;
test = test.mul(3); // test 等于 6 了
test = test.add(5); // test 等于 11 了
```

注意 mul 和 add 其实都需要两个参数。 在我们声明了 using SafeMath for uint 后，我们用来调用这些方法的 uint 就自动被作为第一个参数传递进去了(在此例中就是 test)

我们来看看 add 的源代码看 SafeMath 做了什么:

```
function add(uint256 a, uint256 b) internal pure returns (uint256) {
  uint256 c = a + b;
  assert(c >= a);
  return c;
}
```

基本上 add 只是像 + 一样对两个 uint 相加， 但是它用一个 assert 语句来确保结果大于 a。这样就防止了溢出。

assert 和 require 相似，若结果为否它就会抛出错误。 assert 和 require 区别在于，require 若失败则会返还给用户剩下的 gas， assert 则不会。所以大部分情况下，你写代码的时候会比较喜欢 require，assert 只在代码可能出现严重错误的时候使用，比如 uint 溢出。

所以简而言之， SafeMath 的 add， sub， mul， 和 div 方法只做简单的四则运算，然后在发生溢出或下溢的时候抛出错误。

在我们的代码里使用 SafeMath。
为了防止溢出和下溢，我们可以在我们的代码里找 +， -， *， 或 /，然后替换为 add, sub, mul, div.

比如，与其这样做: `myUint++;`
我们这样做: `myUint = myUint.add(1);`

实战演习:
// 1. 替换成 SafeMath 的 `add`
ownerZombieCount[_to]++;
// 2. 替换成 SafeMath 的 `sub`
ownerZombieCount[_from]--;

```
    ownerZombieCount[_to] = ownerZombieCount[_to].add(1);
    ownerZombieCount[msg.sender] = ownerZombieCount[msg.sender].sub(1);
```

⑪SafeMath 第三部分

太好了，这下我们的 ERC721 实现不会有溢出或者下溢了。

回头看看我们在之前课程写的代码，还有其他几个地方也有可能导致溢出或下溢。

比如， 在 ZombieAttack 里面我们有：

```
myZombie.winCount++;
myZombie.level++;
enemyZombie.lossCount++;
```

我们同样应该在这些地方防止溢出。（通常情况下，总是使用 SafeMath 而不是普通数学运算是个好主意，也许在以后 Solidity 的新版本里这点会被默认实现，但是现在我们得自己在代码里实现这些额外的安全措施）。

不过我们遇到个小问题 — winCount 和 lossCount 是 uint16， 而 level 是 uint32。 所以如果我们用这些作为参数传入 SafeMath 的 add 方法。 它实际上并不会防止溢出，因为它会把这些变量都转换成 uint256:

```
function add(uint256 a, uint256 b) internal pure returns (uint256) {
  uint256 c = a + b;
  assert(c >= a);
  return c;
}

// 如果我们在`uint8` 上调用 `.add`。它将会被转换成 `uint256`.
// 所以它不会在 2^8 时溢出，因为 256 是一个有效的 `uint256`.
```

这就意味着，我们需要再实现两个库来防止 uint16 和 uint32 溢出或下溢。我们可以将其命名为 SafeMath16 和 SafeMath32。

代码将和 SafeMath 完全相同，除了所有的 uint256 实例都将被替换成 uint32 或 uint16。

我们已经将这些代码帮你写好了，打开 safemath.sol 合约看看代码吧。

现在我们需要在 ZombieFactory 里使用它们。

实战演习：

```
using SafeMath for uint256;
using SafeMath32 for uint32;
using SafeMath16 for uint16;
```

⑫SafeMath 第4部分

真棒，现在我们已经为我们的 DApp 里面用到的 uint 数据类型都实现了 SafeMath 了。

让我们把 ZombieAttack 里所有潜在的问题都修复了吧。 （其实在 ZombieHelper 里也有一处 zombies[_zombieId].level++; 需要修复，不过我们已经帮你做好了，这样我们就不用再来一章了😜）。

⑬注释

僵尸游戏的 Solidity 代码终于完成啦。

在以后的课程中，我们将学习如何将游戏部署到以太坊，以及如何和 Web3.js 交互。

不过在你离开第五课之前，我们来谈谈如何 给你的代码添加注释.

注释语法

Solidity 里的注释和 JavaScript 相同。在我们的课程中你已经看到了不少单行注释了：

```
// 这是一个单行注释，可以理解为给自己或者别人看的笔记
```

只要在任何地方添加一个 // 就意味着你在注释。如此简单所以你应该经常这么做。

不过我们也知道你的想法：有时候单行注释是不够的。毕竟你生来话痨。

所以我们有了多行注释：

```
contract CryptoZombies {
  /* 这是一个多行注释。我想对所有花时间来尝试这个编程课程的人说声谢谢。
  它是免费的，并将永远免费。但是我们依然倾注了我们的心血来让它变得更好。

   要知道这依然只是区块链开发的开始而已，虽然我们已经走了很远，
   仍然有很多种方式来让我们的社区变得更好。
   如果我们在哪个地方出了错，欢迎在我们的 github 提交 PR 或者 issue 来帮助我们改进：
    https://github.com/loomnetwork/cryptozombie-lessons

    或者，如果你有任何的想法、建议甚至仅仅想和我们打声招呼，欢迎来我们的电报群：
     https://t.me/loomnetworkdev
  */
}
```

特别是，最好为你合约中每个方法添加注释来解释它的预期行为。这样其他开发者（或者你自己，在6个月以后再回到这个项目中）可以很快地理解你的代码而不需要逐行阅读所有代码。

Solidity 社区所使用的一个标准是使用一种被称作 natspec 的格式，看起来像这样：

```
/// @title 一个简单的基础运算合约
/// @author H4XF13LD MORRIS 💯💯😎💯💯
/// @notice 现在，这个合约只添加一个乘法
contract Math {
  /// @notice 两个数相乘
  /// @param x 第一个 uint
  /// @param y  第二个 uint
  /// @return z  (x * y) 的结果
  /// @dev 现在这个方法不检查溢出
  function multiply(uint x, uint y) returns (uint z) {
    // 这只是个普通的注释，不会被 natspec 解释
    z = x * y;
  }
}
```

1. @title（标题） 和 @author （作者）很直接了.
2. @notice （须知）向 用户 解释这个方法或者合约是做什么的。 @dev （开发者） 是向开发者解释更多的细节。
3. @param （参数）和 @return （返回） 用来描述这个方法需要传入什么参数以及返回什么值。

注意你并不需要每次都用上所有的标签，它们都是可选的。不过最少，写下一个 @dev 注释来解释每个方法是做什么的。

实战演习：

```
/// @title 一个管理转移僵尸所有权的合约
/// @author formikasaever
/// @dev 符合 OpenZeppelin 对 ERC721 标准草案的实现
contract ZombieOwnership is ZombieAttack, ERC721 {
```

⑭放在一起

总结一下

这节课里面我们学到了

- 代币, ERC721 标准，以及可交易的物件/僵尸
- 库以及如何使用库
- 如何利用 SafeMath 来防止溢出和下溢
- 代码注释和 natspec 标准

这节教程完成了我们游戏的 Solidity 代码（仅针对当下来说，未来的课程我们也许会加入更多进去）。

在接下来的两节课中，我们将学习如何将游戏部署到以太坊以及和 web3.js 交互 （这样你就能为你的 DApp 打造一个界面了 ）。

⑮第五课完成

截至第五课完成，所有代码如下：

zombieownership.sol:

```
pragma solidity ^0.4.19;

import "./zombieattack.sol";
import "./erc721.sol";
import "./safemath.sol";

/// @title 一个管理转移僵尸所有权的合约
/// @author formikasaever
/// @dev 符合 OpenZeppelin 对 ERC721 标准草案的实现
contract ZombieOwnership is ZombieAttack, ERC721 {

  using SafeMath for uint256;

  mapping (uint => address) zombieApprovals;

  function balanceOf(address _owner) public view returns (uint256 _balance) {
    return ownerZombieCount[_owner];
  }

  function ownerOf(uint256 _tokenId) public view returns (address _owner) {
    return zombieToOwner[_tokenId];
  }

  function _transfer(address _from, address _to, uint256 _tokenId) private {
    ownerZombieCount[_to] = ownerZombieCount[_to].add(1);
    ownerZombieCount[msg.sender] = ownerZombieCount[msg.sender].sub(1);
    zombieToOwner[_tokenId] = _to;
    Transfer(_from, _to, _tokenId);
  }

  function transfer(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenId) {
    _transfer(msg.sender, _to, _tokenId);
  }

  function approve(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenId) {
    zombieApprovals[_tokenId] = _to;
    Approval(msg.sender, _to, _tokenId);
  }

  function takeOwnership(uint256 _tokenId) public {
    require(zombieApprovals[_tokenId] == msg.sender);
    address owner = ownerOf(_tokenId);
    _transfer(owner, msg.sender, _tokenId);
  }
}
```

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 onlyOwnerOf(_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) onlyOwnerOf(_zombieId) {
    zombies[_zombieId].name = _newName;
  }

  function changeDna(uint _zombieId, uint _newDna) external aboveLevel(20, _zombieId) onlyOwnerOf(_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 onlyOwnerOf(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 onlyOwnerOf(_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";
import "./safemath.sol";

contract ZombieFactory is Ownable {

  using SafeMath for uint256;

  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;
  }

}
```

safemath.sol:

```
pragma solidity ^0.4.18;

/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
library SafeMath {

  /**
  * @dev Multiplies two numbers, throws on overflow.
  */
  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
  }

  /**
  * @dev Integer division of two numbers, truncating the quotient.
  */
  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold
    return c;
  }

  /**
  * @dev Substracts two numbers, throws on overflow (i.e. if subtrahend is greater than minuend).
  */
  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
  }

  /**
  * @dev Adds two numbers, throws on overflow.
  */
  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}
```

erc721.sol:

```
pragma solidity ^0.4.19;
contract ERC721 {
  event Transfer(address indexed _from, address indexed _to, uint256 _tokenId);
  event Approval(address indexed _owner, address indexed _approved, uint256 _tokenId);

  function balanceOf(address _owner) public view returns (uint256 _balance);
  function ownerOf(uint256 _tokenId) public view returns (address _owner);
  function transfer(address _to, uint256 _tokenId) public;
  function approve(address _to, uint256 _tokenId) public;
  function takeOwnership(uint256 _tokenId) public;
}
```

心得

①