实验六：Solidity与智能合约在线编程

1、实验概述

本实验参考自以太坊中的以太猫游戏，通过构建一个“僵尸游戏”来学习智能合约的编写。

2、预备知识

- **Solidity**：Solidity是一种静态类型的编程语言，用于开发在EVM上运行的智能合约。 Solidity被编译为可在EVM上运行的字节码。借由Solidity，开发人员能够编写出可自我运行其欲实现之商业逻辑的应用程序，该程序可被视为一份具权威性且永不可悔改的交易合约。对已具备程序编辑能力的人而言，编写Solidity的难易度就如同编写一般的编程语言。

  附：有需要的同学可以另查阅Solidity官方文档。

  Gavin Wood最初在规划Solidity语言时引用了ECMAScript的语法概念，使其对现有的Web开发者更容易入门；与ECMAScript不同的地方在于Solidity具有静态类型和可变返回类型。而与当前其他EVM目标语言（如Serpent和Mutan）相比，其重要的差异在于Solidity具有一组复杂的成员变量使得合约可支持任意层次结构的映射和结构。Solidity也支持继承，包含C3线性化多重继承。 另外还引入了一个应用程序二进制接口（ABI），该接口（ABI）可在单一合同中实现多种类型安全的功能。

  以下为使用Solidity编写的程序示例：

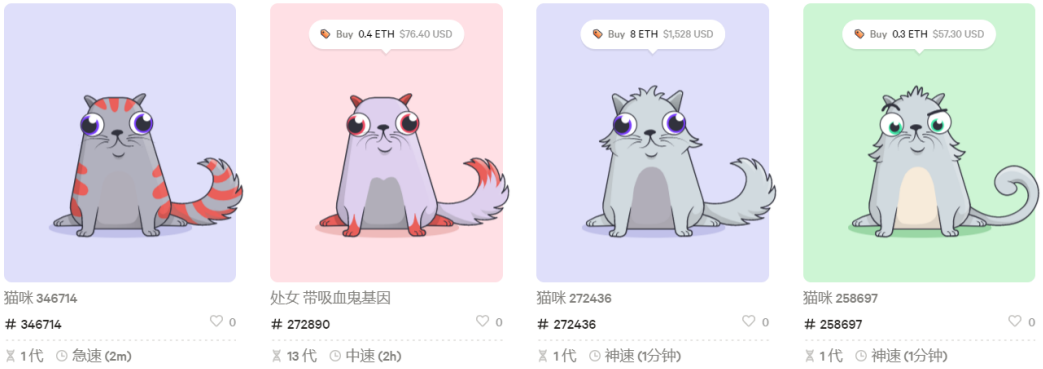
```
contract GavCoin
{
  mapping(address=>uint) balances;
  uint constant totalCoins = 100000000000;

  /// Endows creator of contract with 1m GAV.
  function GavCoin(){
      balances[msg.sender] = totalCoins;
  }

  /// Send $((valueInmGAV / 1000).fixed(0,3)) GAV from the account of
$(message.caller.address()), to an account accessible only by $(to.address()).
  function send(address to, uint256 valueInmGAV) {
    if (balances[msg.sender] >= valueInmGAV) {
      balances[to] += valueInmGAV;
      balances[msg.sender] -= valueInmGAV;
    }
  }

  /// getter function for the balance
  function balance(address who) constant returns (uint256 balanceInmGAV) {
    balanceInmGAV = balances[who];
  }
}
```

- **Remix**：Remix是以太坊提供的一个开发Solidity智能合约的网络版开发软件。合约的开发者在Remix里提供的JavaScript虚拟机上开发，调试好合约后，可以发布到以太坊，或者任何支持Solidity智能合约的区块链上。
- **以太猫**：在以太坊中，每只以太猫都有自己的一个独一无二的DNA，



每只"以太猫"有两个关键部分构成：

1、一段存储在以太坊转账中的文本，用于表示猫的 DNA 序列.
2、一个可爱的图形化的猫的形象，这个猫咪形象由游戏的发行公司 Axiom Zen 提供。

先来看看第一部分中的转账记录。首先，每条"以太猫"转账记录都绑定了一个以太坊钱包地址（这个地址就是猫的主人的钱包地址）。转账中还记录了猫的 DNA 序列，这个序列定义了猫的一系列属性，比如胡须形状，眼睛形状，毛色，条纹等等。

显然，这些属性是需要依赖第二个部分，也就是 Axiom Zen 这家公司提供的图形化界面来将"以太猫"显示在你的浏览器中的。"以太猫"的 DNA 其实就是一串文本字符，比如下面这个就是这只标号为 207454 的猫的 DNA：
`000042d28314c7305c97b94300c0c31c44638c92c0b228ca6104217a52e4b56b`

3、实验准备

- 最新版Chrome或Firefox浏览器即可。

4、实验内容

在本次实验中，每一个小实验所需的基础知识将记录在**《实验六：Solidity实验基础知识》**中，其中包含每个实验所需的知识点，可及时翻阅查看。

4.0 Remix

原版Remix： https://remix.ethereum.org

中文版Remix： http://remix.hubwiz.com

建议使用原版Remix，中文版功能并没有原版全面，仅作为无法打开时的备选方案。

- 在`compile/编译`下，设置编译器为`0.4.19+commit`，选择自动编译。
- 在`deploy/运行`下，设置环境为`Javascript VM`

4.1 实验一：Solidity基础——搭建僵尸工厂

实验一目的创造一个"僵尸工厂"， 用它建立一支僵尸部队。具体包括：

- 工厂会把部队中所有的僵尸保存到数据库中
- 工厂会有一个函数能产生新的僵尸
- 每个僵尸会有一个随机的独一无二的样子

僵尸DNA

参考以太猫的DNA，僵尸的面孔同样取决于它的DNA。本实验定义的DNA很简单，由一个16位的整数组成：

```
8356281049284737
```

如同真正的DNA, 这个数字的不同部分会对应不同的特点。 前2位代表头型，紧接着的2位代表眼睛，等等。

Ps：我们并没有真实的图片，僵尸样貌可以存在于各位的想象中 XD

4.1.1 实验内容

1. 清空原有编译文件，新建文件`ZombieFactory.sol`
2. 为了建立僵尸部队，先建立一个基础合约 `ZombieFactory`，并指定Solidity编译器版本为 `0.4.19`。本实验所有代码均在该版本下运行。
3. 在合约中定义 `dnaDigits` 为 `uint` 数据类型, 并赋值 `16` (因为我们的僵尸DNA由十六位数字组成)。
4. 为了保证我们的僵尸的DNA只含有16个字符，我们先造一个`uint`数据，让它等于10^16。这样以后可以用模运算来获取一个16位整数。
    - 建立一个`uint`类型的变量，名字叫`dnaModulus`, 令其等于 10 的 `dnaDigits` 次方.
5. 创建僵尸结构
    - 建立一个`struct` 命名为 `Zombie`. 有两个属性： `name` (类型为 `string`), 和 `dna` (类型为 `uint`)。
6. 首先需要将僵尸部队保存在合约中，并且能够让其它合约看到这些僵尸，因此需要一个公共数组。
    - 创建一个数据类型为 `Zombie` 的结构体数组，用 `public` 修饰，命名为`zombies`.
7. 定义一个事件 `NewZombie`。 它有3个参数: `zombieId` (`uint`)， `name` (`string`)， 和 `dna` (`uint`)。
8. 定义 创建僵尸函数：能够创建僵尸并添加入`zombies`数组中

- 建立一个私有函数 `_createZombie`。 它有两个参数: `_name` (类型为`string`), 和 `_dna` (类型为 `uint`)。
- 在函数体里新创建一个 `Zombie`， 然后把它加入 `zombies` 数组中。 新创建的僵尸的 `name` 和 `dna`，来自于函数的入参，同时将`zombies`的索引值存为僵尸ID`zombieId`。
- 在函数结束触发事件 `NewZombie`。

9. 定义 DNA生成函数：能够根据字符串随机生成一个DNA。

- 创建一个 `private` 函数 `_generateRandomDna`。它只接收一个输入变量 `_str` (类型 `string`), 返回一个 `uint` 类型的数值。此函数只读取我们合约中的一些变量，所以标记为`view`。
- 使用以太坊内部的哈希函数`keccak256`，根据输入参数`_str`来生成一个十六进制数，类型转换为`uint`后，返回该值的后`dnaModulus`位。

> 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。 这在Ethereum 中有很多应用，但是现在我们只是用它造一个伪随机数。
>
> 使用样例：`keccak256("abcdefg")`

10. 定义 第一个**公共**函数来把部件组合起来： 能够接收僵尸的名字，之后用它生成僵尸的DNA
    - 创建一个 `public` 函数，命名为`createRandomZombie`. 它将被传入一个变量 `_name` (数据类型是 `string`)。
    - 首先调用 `_generateRandomDna` 函数，传入 `_name` 参数来生成一个DNA。
    - 调用 `_createZombie` 函数存入僵尸， 传入参数： `_name` 和 `randDna`。

4.1.2 需实现效果

在JavaScript VM环境下，部署`ZombieFactory`合约。创建三个分别叫`Drogon`、`Rheagal`、`Viserion`的僵尸，向助教展示其DNA。

4.2 实验二：Solidity进阶——僵尸猎食系统

实验一中创建了一个函数用来生成僵尸，并且将它放入区块链上的僵尸数据库中。 在实验二里，会让我们的智能合约看起来更像一个游戏： 它得支持多玩家，并且采用更加有趣，而不仅仅使用随机哈希的方式，来生成新的僵尸。

生成僵尸的方法很简单：作为僵尸，自然是要吃人的。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。

PS：这也是以太猫的繁殖机制。

4.2.1 实验内容

1. 首先通过给合约中的僵尸指定“主人”， 来支持“多玩家”模式。 为了存储僵尸的所有权，我们会使用到两个映射：一个记录僵尸拥有者的地址，另一个记录某地址所拥有僵尸的数量。
    - 创建一个叫做 `zombieToOwner` 的映射。其键是一个`uint`（我们将根据它的 id 存储和查找僵尸），值为 `address`。映射属性为`public`。
    - 创建一个名为 `ownerZombieCount` 的映射，其中键是 `address`，值是 `uint`。
2. 修改`_createZombie`函数来使用映射
    - 首先，在得到新的僵尸 `zombieId` 后，更新 `zombieToOwner` 映射，在 `zombieId` 下面存入 `msg.sender`。
    - 然后，我们为这个 `msg.sender` 名下的 `ownerZombieCount` 加 1。
3. 在`createRandomZombie`开头使用 `require` 来确保这个函数只有在每个用户第一次调用它的时候执行，用以创建初始僵尸。 判断方式：判断该用户的僵尸数是否为0
4. 创建新文件`ZombieFeeding.sol`文件，在其中创建`ZombieFeeding`合约，继承自`ZombieFactory`。记得设置编译版本与import
5. 在`ZombieFeeding`合约中，增加繁殖功能： 当一个僵尸猎食人类时，它自身的DNA将与人类的DNA结合在一起，形成一个新的僵尸DNA，
    - 创建一个名为 `feedAndMultiply` 的函数。 包含三个参数：`_zombieId`（`uint`）、`_targetDna`（`uint`），分别表示猎食僵尸、被吃人类。 设置属性为 `public`。
    - 显然我们不希望别人用我们的僵尸去捕猎。 首先，我们确保对自己僵尸的所有权。因此在函数开始添加一个`require`语句来确保 `msg.sender` 只能是这个僵尸的主人。

- 声明一个名为 `myZombie` 数据类型为 `Zombie` 的本地变量（这是一个 `storage` 型的指针），将其值设定为在 `zombies` 数组中索引为 `_zombieId` 所指向的值。

6. 完成 `feedAndMultiply` 函数：
   - 取得 `_targetDna` 的后 `dnaModulus` 位
   - 生成新的僵尸DNA：计算捕食僵尸与被吃人类DNA的平均值
   - 为僵尸添加标识：将新的僵尸DNA最后两位改为“99”。
   - 调用 `_createZombie` 生成新僵尸，新僵尸名字为“No-one”。**（需要修改 `_createZombie` 函数属性使对继承可见）**。
7. 在 `ZombieFeeding` 合约中增加抓人函数：

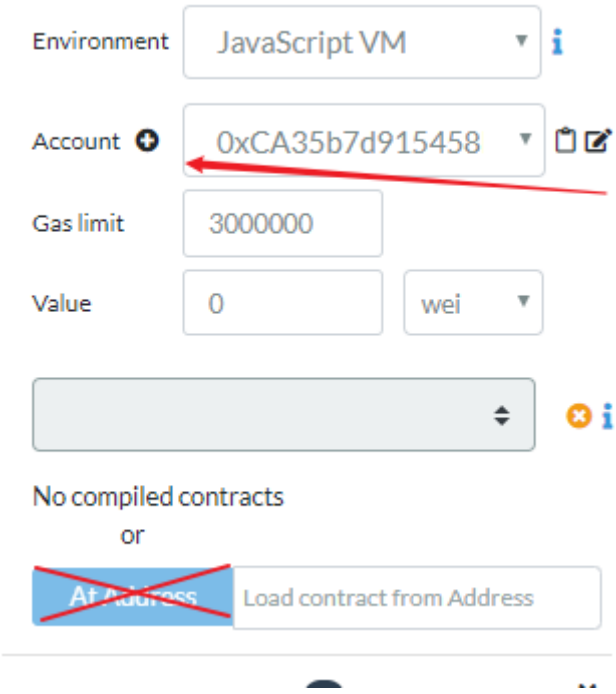
```
function _catchAHuman(uint _name) internal pure returns (uint) {
      uint rand = uint(keccak256(_name));
      return rand;
}
```

8. 在 `ZombieFeeding` 合约中，增加捕食功能：僵尸抓来一个人类，然后繁殖出一个僵尸（雾）
   - 创建一个名为 `feedOnHuman` 的函数。它需要2个 `uint` 类型的参数，`_zombieId` 和 `_humanId` ，这是一个 `public` 类型的函数。
   - 调用 `_catchAHuman` 函数，获得一个人类DNA。
   - 调用 `feedAndMultiply` 函数。传入僵尸id和人类dna

4.2.2 需实现效果

部署 `ZombieFeeding` 合约，实现以下效果：

- 同一账户只可调用一次 `createRandomZombie`
- 以三个用户身份添加 `Drogon`、`Rheagal`、`Viserion` 的僵尸



注：如图所示，账户位置为箭头处。

- 让 `Drogon` 僵尸繁殖一次，展示新僵尸的主人与 `Drogon` 相同

4.3 实验三：Solidity高阶理论

到现在为止，用到的 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。

首先，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。

在实验三中，我们将通过运用一些Solidity高阶理论来实现一些真正的DApp必须要知道的的： **智能协议的所有权，Gas的花费，代码优化，和代码安全。** 具体到我们的僵尸上，本节将会实现为僵尸添加等级系统与“捕食cd”。

4.3.1 实验内容

1. 将 `Ownable` 合约的代码复制一份到新文件 `ownable.sol` 中。 接下来，使得`ZombieFactory`继承 `Ownable`。

   注：由于继承的传承性，意味着`ZombieFeeding`也是`Ownable`子类

2. 给`createRandomZombie`函数添加`onlyOwner`，部署后用不同的账户来调用该函数，感受一下这一修饰符的作用。

   然后删掉这一修饰符

3. 回到 `zombiefactory.sol` 。给僵尸添2个新属性： `level`（`uint32`）和 `readyTime`（`uint32`） - 后者是用来实现一个“冷却定时器”，以限制僵尸猎食的频率。要求元素顺序尽可能节约Gas消耗。

4. 给DApp添加一个“冷却周期”的设定，让僵尸两次攻击或捕猎之间必须等待 **1min**。

   - 声明一个名为 `cooldownTime` 的`uint`，并将其设置为1分钟
   - 修改`_createZombie`中的 `zombies.push` 那一行。

   注： `now` *返回类型* `uint256` ，需要类型转换。

   再来到`ZombieFeeding.sol`的 `feedAndMultiply` 函数：

   - 修改可见性`internal` 以保障合约安全。
   - 在`_targetDna`计算前，检查该僵尸是否已经冷却完毕。提示：require。
   - 在函数结束时重新设置僵尸冷却周期，以表示其捕食行为重新进入cd。

5. 接下来撰写一个属于僵尸自己的函数修饰符，让僵尸能够在达到一定水平后获得特殊能力：

   - 创建一个新的文件`ZombieHelper.sol`，定义合约`ZombieHelper`继承自`ZombieFeeding`
   - 创建一个名为 `aboveLevel` 的`modifier`，它接收2个参数， `_level`(`uint`)以及 `_zombieId`(`uint`)。
   - 函数逻辑确保僵尸 `zombies[_zombieId].level` 大于或等于 `_level`。
   - 修饰符的最后一行为 `_;`，表示修饰符调用结束后返回，并执行调用函数余下的部分。

6. 添加一些使用 `aboveLevel` 修饰符的函数来作为达到level的奖励。 作为游戏，得有一些措施激励玩家们去升级他们的僵尸：

   - 创建一个名为 `changeName` 的函数。它接收2个参数：`_zombieId`（`uint`类型）以及 `_newName`（`string`类型），可见性为 `external`。它带有一个 `aboveLevel` 修饰符，调

用的时候通过 `_level` 参数传入 `2`， 当然，别忘了同时传 `_zombieId` 参数。 在函数中使用 `require` 检查 `msg.sender` 是否是僵尸主人，如果是则将僵尸名改为 `_newName`
    - 在 `changeName` 下创建另一个名为 `changeDna` 的函数。它的定义和内容几乎和 `changeName` 相同，不过它第二个参数是 `_newDna`（`uint` 类型），在修饰符 `aboveLevel` 的 `_level` 参数中传递 `20` 。在函数中可以把僵尸的 `dna` 设置为 `_newDna` 。
7. 定义一个新函数 `getZombiesByOwner` 来获取某个玩家的所有僵尸：
    - 函数有一个参数 `_owner(address)`，声明为 `external view` 属性，返回一个 `uint数组` 。
    - 声明一个名为 `result` 的 `uint [] memory` （内存变量数组），其长度为该 `_owner` 拥有的僵尸数量。
    - 使用 `for` 循环遍历 `zombies` 数组，将主人为 `_owner` 的僵尸添加入 `result`
    - 返回 `result`

    这样能够使得 `getZombiesByOwner` 不花费任何gas。

4.3.2 需实现效果

部署 `ZombieHelper` 合约

- 展示僵尸的冷却cd
- 展示 `getZombiesByOwner` 函数效果
- 展示在 `zombiefactory.sol` 中的僵尸结构，说明为什么你的结构能够节省Gas。

4.4 扩展实验一：支付系统

4.4.1 实验内容

在 `ZombieHelper` 中，

1. 添加支付系统——玩家可以通过支付ETH来升级他们的僵尸。ETH将存储在你拥有的合约中，向你展示你可以通过自己的游戏赚钱。
    - 定义一个 `uint` ，命名为 `levelUpFee`，将值设定为 `0.001 ether`。
    - 定义一个名为 `levelUp` 的函数。 它将接收一个 `uint` 参数 `_zombieId`。 函数应该修饰为 `external` 以及 `payable`。
    - 这个函数首先应该 `require` 确保 `msg.value` 等于 `levelUpFee`。
    - 然后它应该增加僵尸的 `level`：`zombies[_zombieId].level++`。
2. 添加提现系统
    - 创建一个 `withdraw` 函数，它应该几乎和上面的 `GetPaid` 样例一样。
    - 以太的价格在过去几年内不停地波动，所以应该再创建一个函数，允许合约拥有者来设置 `levelUpFee`。

      a. 创建一个函数，名为 `setLevelUpFee`， 其接收一个参数 `uint _fee`，是 `external` 并使用修饰符 `onlyOwner`。

      b. 这个函数应该设置 `levelUpFee` 等于 `_fee`。

4.4.2 需实现效果

- 氪金使我快乐：成功花钱升级，并改名
- 实现一次提现操作。

4.5 扩展实验二：战斗升级系统

实验效果要求

新建一个文件`ZombieAttack.sol`，在其中新建合约`ZombieAttack`，继承自`ZombieHelper`。在这之下编辑新的合约的主要部分。

最终目的是达到这样一个流程：

- 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。

5. 实验报告

实验报告要求

简要记录实验流程，每一个子实验结束后需要备份代码上交，以及回答各部分练习中的问题与相应的感悟体会。