solidity 语法学习

基于 cryptozombies.io

ZombieFactory

```
pragma solidity ^0.4.19;

contract ZombieFactory {

    // 事件， web3.js 可以监控它
    event NewZombie(uint zombieId, string name, uint dna);

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

    // 定义结构体
    struct Zombie {
        string name;
        uint dna;
    }

    // 定义数组
    Zombie[] public zombies;

    // 定义 mapping 结构， 可理解为 python 里面的 dict
    mapping (uint => address) public zombieToOwner;
    mapping (address => uint) ownerZombieCount;

    function _createZombie(string _name, uint _dna) internal {
        uint id = zombies.push(Zombie(_name, _dna)) - 1; // 获取刚刚放到数组的元素的 id

        // msg.sender 为调用者的 地址。
        zombieToOwner[id] = msg.sender;
        ownerZombieCount[msg.sender]++;

        // 触发事件
        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 {

        // 要求每个账户只能有一只僵尸，否则退出
        require(ownerZombieCount[msg.sender] == 0);

        uint randDna = _generateRandomDna(_name);
        randDna = randDna - randDna % 100;
        _createZombie(_name, randDna);
    }

}
```

学到了

- 函数的定义
- 数组的使用
- mapping 的使用
- require的使用
- 事件的使用

ZombieFeeding

```
pragma solidity ^0.4.19;

import "./zombiefactory.sol";

// 定义一个合约接口，通过这种方式可以调用其他合约的公开方法
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
  );
}

// 继承
contract ZombieFeeding is ZombieFactory {

  KittyInterface kittyContract;

  // 使用了函数修饰符，确保只有 合约账户本身可以调用该方法
  function setKittyContractAddress(address _address) external onlyOwner {
    // 通过 合约地址实例化接口，以后可以通过这个接口调用该合约的方法
    kittyContract = KittyInterface(_address);
  }

  // 传输结构体指针
  function _triggerCooldown(Zombie storage _zombie) internal {
    _zombie.readyTime = uint32(now + cooldownTime);
  }

  // 返回 bool 类型
  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")) { // if 语句的使用
      newDna = newDna - newDna % 100 + 99;
    }
    _createZombie("NoName", newDna);
    _triggerCooldown(myZombie);
  }

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

```
    uint kittyDna;
    // 调用其他合约的方法
    (,,,,,,,,kittyDna) = kittyContract.getKitty(_kittyId);  // 多个返回值的获取

    feedAndMultiply(_zombieId, kittyDna, "kitty");
  }

}
```

学到了

- 调用其他合约的方法
- 结构体传值
- 接收多个返回值的方法
- 函数修饰符的使用

ZombieHelper

```
pragma solidity ^0.4.19;

import "./zombiefeeding.sol";

contract ZombieHelper is ZombieFeeding {

  uint levelUpFee = 0.001 ether;

  // 定义修饰函数，会在被修饰函数调用前调用
  modifier aboveLevel(uint _level, uint _zombieId) {
    // 如果指定僵尸的 level 小于 _level 就会退出，否则继续执行被修饰的函数
    require(zombies[_zombieId].level >= _level);
    _;
  }

  // 用于提出 以太坊里面的 eth
  function withdraw() external onlyOwner {
    owner.transfer(this.balance);
  }

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

```
  // payable 修饰符表示，可以往这个方法发送 eth
  function levelUp(uint _zombieId) external payable {
    require(msg.value == levelUpFee);  // 判断 eth 的值
    zombies[_zombieId].level++;
  }

  // 使用了修饰符，当级别大于 2 时才能修改名字
  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;
  }

  // 返回一个列表
  function getZombiesByOwner(address _owner) external view returns(uint[]) {

    // 定义 memory 数组，节省 gas
    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;
  }

}
```

学到了

- 定义修饰函数，以及往修饰函数传参
- 接收，提取 `eth`
- 返回 `uint[]`
- `memory` 变量， `for` 循环的使用

后面接着又了解了 `SafeMath` 的使用

```
pragma solidity ^0.4.19;

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

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);
  }
}
```

来源： https://www.cnblogs.com/hac425/p/9761077.html