加密僵尸：课程所有代码

```
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;
}
```

```
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;
  }
}
```

```
pragma solidity ^0.4.19;
/**
 * @title Ownable
```

LEVI_104

```
 * @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;
  }

}
```

```
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 _n
    uint id = zombies.push(Zombie(
    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);
  }

}
```

```
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");
  }
}
```

```
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;
  }

}
```

```
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
  }

```

```
  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);
    }
  }
}
```

```
pragma solidity ^0.4.19;

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

/// TODO: 把这里变成 natspec 标准的注释把
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);
  }
}
```