加密僵尸：僵尸攻击人类

映射（Mapping）和地址（Address）

Addresses （地址）

以太坊区块链由 _ **account** _ (账户)组成，你可以把它想象成银行账户。一个帐户的余额是 **_以太_** （在以太坊区块链上使用的币种），你可以和其
支付和接受以太币，就像你的银行帐户可以电汇资金到其他银行帐户一样。

每个帐户都有一个“地址”，你可以把它想象成银行账号。这是账户唯一的标识符，它看起来长这样：

0x0cE446255506E92DF41614C46F1d6df9Cc969183

Mapping（映射）

```
//对于金融应用程序，将用户的余额保存在一个 uint类型的变量中：
mapping (address => uint) public accountBalance;
//或者可以用来通过userId 存储/查找的用户名
mapping (uint => string) userIdToName;
```

映射本质上是存储和查找数据所用的键-值对。在第一个例子中，键是一个 address，值是一个 uint，在第二个例子中，键是一个uint，值是一个 st

Msg.sender

在 Solidity 中，有一些全局变量可以被所有函数调用。 其中一个就是 msg.sender，它指的是当前调用者（或智能合约）的 address。

注意：在 Solidity 中，功能执行始终需要从外部调用者开始。 一个合约只会在区块链上什么也不做，除非有人调用其中的函数。所以 msg.sender总

以下是使用 msg.sender 来更新 mapping 的例子：

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

function setMyNumber(uint _myNumber) public {
  // 更新我们的 `favoriteNumber` 映射来将 `_myNumber`存储在 `msg.sender`名下
  favoriteNumber[msg.sender] = _myNumber;
  // 存储数据至映射的方法和将数据存储在数组相似
}

function whatIsMyNumber() public view returns (uint) {
  // 拿到存储在调用者地址名下的值
  // 若调用者还没调用 setMyNumber， 则值为 `0`
  return favoriteNumber[msg.sender];
}
```

在这个小小的例子中，任何人都可以调用 setMyNumber 在我们的合约中存下一个 uint 并且与他们的地址相绑定。 然后，他们调用 whatIsMyNumber 们存储的 uint。

使用 msg.sender 很安全，因为它具有以太坊区块链的安全保障 —— 除非窃取与以太坊地址相关联的私钥，否则是没有办法修改其他人的数据的。

Require

```
function sayHiToVitalik(string _name) public returns (string) {
  // 比较 _name 是否等于 "Vitalik". 如果不成立，抛出异常并终止程序
  // (敲黑板: Solidity 并不支持原生的字符串比较，我们只能通过比较
  // 两字符串的 keccak256 哈希值来进行判断)
  require(keccak256(_name) == keccak256("Vitalik"));
  // 如果返回 true，运行如下语句
  return "Hi!";
}
```

如果你这样调用函数 sayHiToVitalik（“Vitalik”），它会返回“Hi！”。而如果调用的时候使用了其他参数，它则会抛出错误并停止执行。

继承（Inheritance）

```
contract Doge {
  function catchphrase() public returns (string) {
    return "So Wow CryptoDoge";
  }
}

contract BabyDoge is Doge {
  function anotherCatchphrase() public returns (string) {
    return "Such Moon BabyDoge";
  }
}
```

引入（Import）

import "./someothercontract.sol";

这样当我们在合约（contract）目录下有一个名为 someothercontract.sol 的文件（./ 就是同一目录的意思），它就会被编译器导入。

Storage与Memory

Storage 变量是指永久存储在区块链中的变量。 ***Memory*** 变量则是临时的，当外部函数对某合约调用完成时，内存型变量即被移除。 你可以把它想你电脑的硬盘或是RAM中数据的关系。

大多数时候你都用不到这些关键字，默认情况下 Solidity 会自动处理它们。 状态变量（在函数之外声明的变量）默认为“存储”形式，并永久写入区块数内部声明的变量是“内存”型的，它们函数调用结束后消失。

```
contract SandwichFactory {
  struct Sandwich {
    string name;
    string status;
  }

  Sandwich[] sandwiches;

  function eatSandwich(uint _index) public {
    // Sandwich mySandwich = sandwiches[_index];

    // ^ 看上去很直接，不过 Solidity 将会给出警告
    // 告诉你应该明确在这里定义 `storage` 或者 `memory`。

    // 所以你应该明确定义 `storage`:
    Sandwich storage mySandwich = sandwiches[_index];
    // ...这样 `mySandwich` 是指向 `sandwiches[_index]`的指针
    // 在存储里，另外...
    mySandwich.status = "Eaten!";
    // ...这将永久把 `sandwiches[_index]` 变为区块链上的存储

    // 如果你只想要一个副本，可以使用`memory`:
    Sandwich memory anotherSandwich = sandwiches[_index + 1];
    // ...这样 `anotherSandwich` 就仅仅是一个内存里的副本了
    // 另外
    anotherSandwich.status = "Eaten!";
    // ...将仅仅修改临时变量，对 `sandwiches[_index + 1]` 没有任何影响
    // 不过你可以这样做:
    sandwiches[_index + 1] = anotherSandwich;
    // ...如果你想把副本的改动保存回区块链存储
  }
}
```

internal 和 external

除 public 和 private 属性之外，Solidity 还使用了另外两个描述函数可见性的修饰词：internal（内部） 和 external（外部）。

internal 和 private 类似，不过， 如果某个合约继承自其父合约，这个合约即可以访问父合约中定义的“内部”函数。（嘿，这听起来正是我们想要的样！）。

external 与public 类似，只不过这些函数只能在合约之外调用 - 它们不能被合约内的其他函数调用。稍后我们将讨论什么时候使用 external 和 pub

声明函数 internal 或 external 类型的语法，与

```
contract Sandwich {
  uint private sandwichesEaten = 0;

  function eat() internal {
    sandwichesEaten++;
  }
}

contract BLT is Sandwich {
  uint private baconSandwichesEaten = 0;

  function eatWithBacon() public returns (string) {
    baconSandwichesEaten++;
    // 因为eat() 是internal 的，所以我们能在这里调用
    eat();
  }
}
```

使用接口

继续前面 NumberInterface 的例子，我们既然将接口定义为：

```
contract NumberInterface {
  function getNum(address _myAddress) public view returns (uint);
}
```

我们可以在合约中这样使用：

```
contract MyContract {
  address NumberInterfaceAddress = 0xab38...;
  // ^ 这是FavoriteNumber合约在以太坊上的地址
  NumberInterface numberContract = NumberInterface(NumberInterfaceAddress);
  // 现在变量 `numberContract` 指向另一个合约对象

  function someFunction() public {
    // 现在我们可以调用在那个合约中声明的 `getNum`函数:
    uint num = numberContract.getNum(msg.sender);
    // ...在这儿使用 `num`变量做些什么
  }
}
```

通过这种方式，只要将您合约的可见性设置为public(公共)或external(外部)，它们就可以与以太坊区块链上的任何其他合约进行交互。

处理多返回值

```
function multipleReturns() internal returns(uint a, uint b, uint c) {
  return (1, 2, 3);
}

function processMultipleReturns() external {
  uint a;
  uint b;
  uint c;
  // 这样来做批量赋值:
  (a, b, c) = multipleReturns();
}

// 或者如果我们只想返回其中一个变量:
function getLastReturnValue() external {
  uint c;
  // 可以对其他字段留空:
  (,,c) = multipleReturns();
}
```

本章节全部代码

```
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 {

  address ckAddress = 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d;
  KittyInterface kittyContract = KittyInterface(ckAddress);

  function feedAndMultiply(uint _zombieId, uint _targetDna, string _species) public {
    require(msg.sender == zombieToOwner[_zombieId]);
    Zombie storage myZombie = zombies[_zombieId];
    _targetDna = _targetDna % dnaModulus;
    uint newDna = (myZombie.dna + _targetDna) / 2;
    if (keccak256(_species) == keccak256("kitty")) {
      newDna = newDna - newDna % 100 + 99;
    }
    _createZombie("NoName", newDna);
  }

  function feedOnKitty(uint _zombieId, uint _kittyId) public {
    uint kittyDna;
    (,,,,,,,,,kittyDna) = kittyContract.getKitty(_kittyId);
    feedAndMultiply(_zombieId, kittyDna, "kitty");
  }

}
```