加密僵尸：高级Solidity理论

智能协议的永固性

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
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。

Ownable Contracts

OpenZeppelin库的`Ownable` 合约

```
/**
 * @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;
  }
}
```

- 构造函数：`function Ownable()`是一个 **_ constructor_** (构造函数)，构造函数不是必须的，它与合约同名，构造函数一生中唯一的一次执行，就
最初被创建的时候。
- 函数修饰符：`modifier onlyOwner()`。 修饰符跟函数很类似，不过是用来修饰其他已有函数用的， 在其他语句执行前，为它检查下先验条件。
中，我们就可以写个修饰符 `onlyOwner` 检查下调用者，确保只有合约的主人才能运行本函数。

`Ownable` 合约基本都会这么干：

1. 合约创建，构造函数先行，将其 `owner` 设置为`msg.sender`（其部署者）
2. 为它加上一个修饰符 `onlyOwner`，它会限制陌生人的访问，将访问某些函数的权限锁定在 `owner` 上。
3. 允许将合约所有权转让给他人。

onlyOwner 函数修饰符

```
/**
 * @dev 调用者不是‘主人’，就会抛出异常
 */
modifier onlyOwner() {
  require(msg.sender == owner);
  _;
}
```

onlyOwner 函数修饰符是这么用的：

```
contract MyContract is Ownable {
  event LaughManiacally(string laughter);

  //注意！ `onlyOwner`上场 :
  function likeABoss() external onlyOwner {
    LaughManiacally("Muahahahaha");
  }
}
```

注意 likeABoss 函数上的 onlyOwner 修饰符。 当你调用 likeABoss 时，**首先执行** onlyOwner 中的代码， 执行到 onlyOwner 中的 _; 语句时，程序再执行 likeABoss 中的代码。

Gas

Gas - 驱动以太坊DApps的能源。

在 Solidity 中，你的用户想要每次执行你的 DApp 都需要支付一定的 ***gas***，gas 可以用以太币购买，因此，用户每次跑 DApp 都得花费以太币。

一个 DApp 收取多少 gas 取决于功能逻辑的复杂程度。每个操作背后，都在计算完成这个操作所需要的计算资源，（比如，存储数据就比做个加法运多）， 一次操作所需要花费的 ***gas*** 等于这个操作背后的所有运算花销的总和。

为什么要用 *gas* 来驱动?

以太坊就像一个巨大、缓慢、但非常安全的电脑。当你运行一个程序的时候，网络上的每一个节点都在进行相同的运算，以验证它的输出 —— 这就“中心化” 由于数以千计的节点同时在验证着每个功能的运行，这可以确保它的数据不会被被监控，或者被刻意修改。

可能会有用户用无限循环堵塞网络，抑或用密集运算来占用大量的网络资源，为了防止这种事情的发生，以太坊的创建者为以太坊上的资源制定了价以太坊上运算或者存储，你需要先付费。

省 gas 的招数：结构封装 （Struct packing）

```
struct NormalStruct {
  uint a;
  uint b;
  uint c;
}

struct MiniMe {
  uint32 a;
  uint32 b;
  uint c;
}

// 因为使用了结构打包，`mini` 比 `normal` 占用的空间更少
NormalStruct normal = NormalStruct(10, 20, 30);
MiniMe mini = MiniMe(10, 20, 30);
```

时间单位

变量 now 将返回当前的unix时间戳（自1970年1月1日以来经过的秒数）。我写这句话时 unix 时间是 1515527488。

注意：Unix时间传统用一个32位的整数进行存储。这会导致“2038年”问题，当这个32位的unix时间戳不够用，产生溢出，使用这个时间的遗留系统就以，如果我们想让我们的 DApp 跑够20年，我们可以使用64位整数表示时间，但为此我们的用户又得支付更多的 gas。

```
uint lastUpdated;

// 将‘上次更新时间’ 设置为 ‘现在’
```

```
function updateTimestamp() public {
    lastUpdated = now;
}

// 如果到上次`updateTimestamp` 超过5分钟，返回 'true'
// 不到5分钟返回 'false'
function fiveMinutesHavePassed() public view returns (bool) {
  return (now >= (lastUpdated + 5 minutes));
}
```

带参数的函数修饰符

```
// 存储用户年龄的映射
mapping (uint => uint) public age;

// 限定用户年龄的修饰符
modifier olderThan(uint _age, uint _userId) {
  require(age[_userId] >= _age);
  _;
}

// 必须年满16周岁才允许开车 (至少在美国是这样的).
// 我们可以用如下参数调用`olderThan` 修饰符:
function driveCar(uint _userId) public olderThan(16, _userId) {
  // 其余的程序逻辑
}
```

利用 'View' 函数节省 Gas

当玩家从外部调用一个view函数，是不需要支付一分 gas 的。

这是因为 view 函数不会真正改变区块链上的任何数据 - 它们只是读取。因此用 view 标记一个函数，意味着告诉 web3.js，运行这个函数只需要查询
太坊节点，而不需要在区块链上创建一个事务（事务需要运行在每个节点上，因此花费 gas）。

> 注意：如果一个 view 函数在另一个函数的内部被调用，而调用函数与 view 函数的不属于同一个合约，也会产生调用成本。这是因为如果主调函
> 坊创建了一个事务，它仍然需要逐个节点去验证。所以标记为 view 的函数只有在外部调用时才是免费的。

存储非常昂贵

Solidity 使用storage(存储)是相当昂贵的，”写入“操作尤其贵。

这是因为，无论是写入还是更改一段数据， 这都将永久性地写入区块链。”永久性“啊！需要在全球数千个节点的硬盘上存入这些数据，随着区块链的
份数更多，存储量也就越大。这是需要成本的！

为了降低成本，不到万不得已，避免将数据写入存储。这也会导致效率低下的编程逻辑 - 比如每次调用一个函数，都需要在 memory(内存) 中重建一
不是简单地将上次计算的数组给存储下来以便快速查找。

在大多数编程语言中，遍历大数据集合都是昂贵的。但是在 Solidity 中，使用一个标记了external view的函数，遍历比 storage 要便宜太多，因为
会产生任何花销。

以下是申明一个内存数组的例子：

```
function getArray() external pure returns(uint[]) {
  // 初始化一个长度为3的内存数组
  uint[] memory values = new uint[](3);
  // 赋值
  values.push(1);
  values.push(2);
  values.push(3);
  // 返回数组
  return values;
}
```