实验六：基础知识部分

附：有需要的同学可以另查阅Solidity官方文档。

实验一：Solidity基础——搭建僵尸工厂

合约

Solidity 的代码都包裹在**合约**里面. 一份 `合约` 就是以太应币应用的基本模块， 所有的变量和函数都属于一份合约, 它是你所有应用的起点.

一份名为 `HelloWorld` 的空合约如下:

```
contract HelloWorld {
}
```

版本指令

所有的 Solidity 源码都必须冠以 "version pragma" — 标明 Solidity 编译器的版本. 以避免将来新的编译器可能破坏你的代码。

例如: `pragma solidity ^0.5.13;` (当前 Solidity 的最新稳定版本是 0.5.13).

当然我们也可以指定一个版本区间，比如 `pragma solidity >=0.4.12 <0.6.0;` 很好理解就不解释了。

综上所述， 下面就是一个最基本的合约 — 每次建立一个新的项目时的第一段代码:

```
pragma solidity ^0.5.13;

contract HelloWorld {
}
```

状态变量和整数

状态变量：被永久地保存在合约中。也就是说它们被写入以太币区块链中. 可以想象成写入一个数据库。

```
contract Example {
  // 这个无符号整数将会永久的被保存在区块链中
  uint myUnsignedInteger = 100;
}
```

在上面的例子中，定义 `myUnsignedInteger` 为 `uint` 类型，并赋值100。

无符号整数`uint`： `uint` 无符号数据类型， 指**其值不能是负数**，对于有符号的整数存在名为 `int` 的数据类型。

> 注: Solidity中， `uint` 实际上是 `uint256`代名词， 一个256位的无符号整数。你也可以定义位数少的uints — `uint8`， `uint16`， `uint32`， 等..... 但一般来讲更愿意使用简单的 `uint`， 除非在某些特殊情况下。

字符串`string`： 字符串用于保存任意长度的 UTF-8 编码数据。 如： `string greeting = "Hello world!"`。

数学运算

在 Solidity 中，数学运算很直观明了，与其它程序设计语言相同:

- 加法: `x + y`
- 减法: `x - y`,
- 乘法: `x * y`
- 除法: `x / y`
- 取模 / 求余: `x % y` *(例如, `13 % 5` 余 `3`)*

Solidity 还支持 **乘方操作** (如：x 的 y次方） // 例如： 5 ** 2 = 25

```
uint x = 5 ** 2; // equal to 5^2 = 25
```

结构体

有时你需要更复杂的数据类型，Solidity 提供了 **结构体**:

```
struct Person {
  uint age;
  string name;
}
```

结构体允许你生成一个更复杂的数据类型，它有多个属性。

数组

如果你想建立一个集合，可以用 ***数组***这样的数据类型. Solidity 支持两种数组: ***静态*** 数组和***动态*** 数组:

```
// 固定长度为2的静态数组:
uint[2] fixedArray;
// 固定长度为5的string类型的静态数组:
string[5] stringArray;
// 动态数组，长度不固定，可以动态添加元素:
uint[] dynamicArray;
```

你也可以建立一个 **结构体**类型的数组 例如，比如之前提到的 `Person`:

```
Person[] people; // 这是动态数组，我们可以不断添加元素
```

*记住：状态变量被永久保存在区块链中。所以在你的合约中创建**动态数组**来保存成结构的数据是非常有意义的。*

公共数组

你可以定义 `public` 数组，语法如下:

```
Person[] public people;
```

`public`条目意味着其它的合约可以从这个数组读取数据（但不能写入数据），所以这在合约中是一个有用的保存公共数据的模式。

数组中插入元素

可以定义一个新的 `Person` 结构，然后把它加入到名为 `people` 的数组中.

现在我们学习创建新的 `Person` 结构，然后把它加入到名为 `people` 的数组中.

```
Person satoshi = Person(172, "Satoshi");
people.push(satoshi);
```

你也可以两步并一步，用一行代码更简洁:

```
people.push(Person(16, "Vitalik"));
```

> 注：`array.push()` 在数组的 **尾部** 加入新元素，所以元素在数组中的顺序就是我们添加的顺序，如:

```
uint[] numbers;
numbers.push(5);
numbers.push(10);
numbers.push(15);
// numbers is now equal to [5, 10, 15]
```

> `array.push()` 在完成加入之后，同时会返回数组的长度，类型是`uint`

函数

在 Solidity 中函数定义的句法如下:

```
function eatHamburgers(string _name, uint _amount) public returns (string) {
}
```

这是一个名为 `eatHamburgers` 的函数，它接受两个参数：一个 `string`类型的 和 一个 `uint`类型的，返回一个`string`类型。

> 注：习惯上函数里的变量都是以(`_`)开头 (但不是硬性规定) 以区别全局变量。本实验会沿用这个习惯。

和其他语言一样，函数调用方式如下:

```
string result = eatHamburgers("vitalik", 100);
```

公开、私有函数

Solidity 定义的函数的属性默认为`public`。 这就意味着任何一方 (或其它合约) 都可以调用你合约里的函数。

显然，不是什么时候都需要这样，而且这样的合约易于受到攻击。 所以将自己的函数定义为`private`是一个好的编程习惯，只有当你需要外部世界调用它时才将它设置为`public`。

定义一个私有函数很简单，在其后添加`private`关键字即可。和函数的参数类似，私有函数的名字习惯用(`_`)起始。

```
uint[] numbers;

function _addToArray(uint _number) private {
  numbers.push(_number);
}
```

这意味着只有合约内部才能够调用这个函数，给`numbers`数组添加新成员。

Ps：当然显式指定函数为`public`也是可以的，通常为了可读性，总是会标记函数属性。

更多的函数修饰符

参考这样的代码：

```
string greeting = "What's up dog";

function sayHello() public returns (string) {
    return greeting;
}
```

上面的函数实际上没有改变 Solidity 里的状态，即，它没有改变任何值或者写任何东西。

这种情况下我们可以把函数定义为 **view**, 意味着它只能读取数据不能更改数据:

```
function sayHello() public view returns (string) {}
```

Solidity 还支持 **pure** 函数, 表明这个函数甚至都不访问应用里的数据，例如：

```
function _multiply(uint a, uint b) private pure returns (uint) {
    return a * b;
}
```

这个函数甚至都不读取应用里的状态——它的返回值完全取决于它的输入参数，在这种情况下我们把函数定义为 **pure**.

类型转换

有时你需要变换数据类型。例如:

```
uint8 a = 5;
uint b = 6;
// 将会抛出错误，因为 a * b 返回 uint，而不是 uint8:
uint8 c = a * b;
// 我们需要将 b 转换为 uint8:
uint8 c = a * uint8(b);
```

上面，`a * b` 返回类型是 `uint`，但是当我们尝试用 `uint8` 类型接收时，就会造成潜在的错误。如果把它的数据类型转换为 `uint8`，就可以了，编译器也不会出错。

事件

事件 是合约和区块链通讯的一种机制。能够让前端应用"监听"某些事件，并做出反应。

```
// 这里建立事件
event IntegersAdded(uint x, uint y, uint result);

function add(uint _x, uint _y) public {
  uint result = _x + _y;
  //触发事件，通知app
  IntegersAdded(_x, _y, result);
  return result;
}
```

你的 app 前端可以监听 `IntegersAdded` 事件。JavaScript 实现如下:

```
YourContract.IntegersAdded(function(error, result) {
  // Do something
}
```

注：本实验不涉及前端开发，仅作了解。

实验二：Solidity进阶——僵尸猎食系统

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），你可以和其他帐户之间支付和接受以太币，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
```

地址中包含很多细节，现在你只需要了解**地址属于特定用户（或智能合约）的。**

所以我们可以指定"地址"作为僵尸主人的 ID。当用户通过与我们的应用程序交互来创建新的僵尸时，新僵尸的所有权被设置到调用者的以太坊地址下。

Mapping（映射）

实验一中介绍了 ***结构体*** 和 ***数组*** 。 ***映射*** 是另一种在 Solidity 中存储有组织数据的方法。

映射是这样定义的：

```
//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 `string`。

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：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`总是存在的。

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` 就会返回他们存储的 `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（要求）

`require`使得函数在执行过程中，当不满足某些条件时抛出错误，并停止执行：

```
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。

因此，在调用一个函数之前，用 `require` 验证前置条件是非常有效的手段。

Inheritance（继承）

当代码过于冗长的时候，最好将代码和逻辑分拆到多个不同的合约中，以便于管理。

Solidity 提供了 **inheritance** (继承)来整理代码：

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

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

由于 `BabyDoge` 是从 `Doge` 那里继承过来的。 这意味着当你编译和部署了 `BabyDoge`，它将可以访问 `catchphrase()` 和 `anotherCatchphrase()` 和其他我们在 `Doge` 中定义的其他公共函数。

这可以用于逻辑继承（比如表达子类的时候，`Cat` 是一种 `Animal`）。 但也可以简单地将类似的逻辑组合到不同的合约中以组织代码。

Import

在 Solidity 中，当你有多个文件并且想把一个文件导入另一个文件时，可以使用 `import` 语句：

```
import "./someothercontract.sol";

contract newContract is SomeOtherContract {
}
```

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

在 Solidity 中，有两个地方可以存储变量 —— `storage` 或 `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。 你可以把它想象成存储在你电脑的硬盘或是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，并永久写入区块链；而在函数内部声明的变量是“内存”型的，它们函数调用结束后消失。

然而也有一些情况下，你需要手动声明存储类型，主要用于处理函数内的 **结构体** 和 **数组** 时。

这两个属性主要用于优化代码以节省合约的gas消耗，目前只需知道有时需要显式地声明 `storage` 或 `memory` 即可。

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那样！）。

`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 `public`。

声明函数 `internal` 或 `external` 类型的语法，与声明 `private` 和 `public` 类 型相同：

```
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();
  }
}
```

实验三：Solidity高阶理论——僵尸

合约所有权

由于合约不可更改的特殊性，使得合约开发者通常需要留一些“后门”来应对意外情况，比如外链的某个合约突然作废了或者不能用了。但由于合约部署在以太坊上任何人都能够看到，这个“后门”便被公开了，任何人都能够走这个后门。这对于开发者来说自然是不愿意的，要对付这样的情况，通常的做法是指定合约的“所有权” - 就是说，给它指定一个主人，只有主人对它享有特权。

OpenZeppelin库的 `Ownable` 合约

下面是一个 `Ownable` 合约的例子： 来自 **OpenZeppelin** Solidity 库的 `Ownable` 合约。OpenZeppelin 是主打安保和社区审查的智能合约库，您可以在自己的 DApps中引用。

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

其中有一些很有意思的东西：

- 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，就是在合约最初被创建的时候。
- 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。 在这个例子中，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。我们下一章中会详细讲述修饰符，以及那个奇怪的`_;`。

所以`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` 适用范围及其广泛，大多数人开发自己的 Solidity DApps，都是从复制/粘贴 `Ownable` 开始的，从它再继承出的子类，并在之上进行功能开发。

函数修饰符

函数修饰符看起来跟函数没什么不同，不过关键字`modifier`告诉编译器，这是个`modifier(修饰符)`，而不是个`function(函数)`。它不能像函数那样被直接调用，只能被添加到函数定义的末尾，用以改变函数的行为。

```
/**
 * @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);

  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，程序再返回并执行`likeABoss`中的代码。

可见，尽管函数修饰符也可以应用到各种场合，但最常见的还是放在函数执行之前添加快速的`require`检查。

因为给函数添加了修饰符`onlyOwner`，使得**唯有合约的主人**（也就是部署者）才能调用它。

> 注意：主人对合约享有的特权当然是正当的，不过也可能被恶意使用。比如，万一，主人添加了个后门，允许他偷走别人的僵尸呢？

> 所以非常重要的是，部署在以太坊上的 DApp，并不能保证它真正做到去中心，你需要阅读并理解它的源代码，才能防止其中没有被部署者恶意植入后门；作为开发人员，如何做到既要给自己留下修复 bug 的余地，又要尽量地放权给使用者，以便让他们放心你，从而愿意把数据放在你的 DApp 中，这确实需要个微妙的平衡。

带参数的函数修饰符

之前我们已经读过一个简单的函数修饰符了：`onlyOwner`。函数修饰符也可以带参数。例如：

```
// 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
}
```

`olderThan` 修饰符可以像函数一样接收参数，是"宿主"函数 `driveCar` 把参数传递给它的修饰符的。

Gas

Gas是另一种使得 Solidity 编程语言与众不同的特征：

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算贵得多），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。

由于运行你的程序需要花费用户的真金白银，在以太坊中代码的编程语言，比其他任何编程语言都更强调优化。同样的功能，使用笨拙的代码开发的程序，比起经过精巧优化的代码来，运行花费更高，这显然会给成千上万的用户带来大量不必要的开销。

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 —— 这就是所谓的"去中心化" 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格，想要在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）

目前应该知道了除了基本版的 `uint` 外，还有其他变种 `uint`：`uint8`，`uint16`，`uint32` 等。

通常情况下我们不会考虑使用 `uint` 变种，因为无论如何定义 `uint` 的大小，Solidity 为它保留256位的存储空间。例如，使用 `uint8` 而不是 `uint`（`uint256`）不会为你节省任何 gas。

除非，把 `uint` 绑定到 `struct` 里面。

如果一个 `struct` 中有多个 `uint`，则尽可能使用较小的 `uint`，Solidity 会将这些 `uint` 打包在一起，从而占用较少的存储空间。例如：

```
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);
```

所以，当 `uint` 定义在一个 `struct` 中的时候，尽量使用最小的整数子类型以节约空间。 并且把同样类型的变量放一起（即在 struct 中将把变量按照类型依次放置），这样 Solidity 可以将存储空间最小化。例如，有两个 `struct`：

```
uint c; uint32 a; uint32 b;` 和 `uint32 a; uint c; uint32 b;
```

前者比后者需要的gas更少，因为前者把 `uint32` 放一起了。

“view” 函数不花 “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你的本地以太坊节点，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）。

稍后我们将介绍如何在自己的节点上设置 web3.js。但现在，你关键是要记住，在所能只读的函数上标记上表示“只读”的“`external view` 声明，就能为你的玩家减少在 DApp 中 gas 用量。

> 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数在以太坊创建了一个事务，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。

减少存储开销

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，随着区块链的增长，拷贝份数更多，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) 中重建一个数组，而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 `view` 函数不会产生任何花销。

如何在内存中声明数组

在数组后面加上 `memory` 关键字， 表明这个数组是仅仅在内存中创建，不需要写入外部存储，并且在函数调用结束时它就解散了。与在程序结束时把数据保存进 `storage` 的做法相比，内存运算可以大大节省gas开销 -- 把这数组放在 `view` 里用，完全不用花钱。

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;
}
```

这个小例子展示了一些语法规则，下一章中，我们将通过一个实际用例，展示它和 `for` 循环结合的做法。

> 注意：内存数组 **必须** 用长度参数（在本例中为`3`）创建。目前不支持 `array.push()` 之类的方法调整数组大小，在未来的版本可能会支持长度修改。

时间单位

Solidity 使用自己的本地时间单位。

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麻烦了。所以，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。真是个两难的设计啊！

Solidity 还包含 `秒(seconds)`，`分钟(minutes)`，`小时(hours)`，`天(days)`，`周(weeks)` 和 `年(years)` 等时间单位。它们都会转换成对应的秒数放入 `uint` 中。所以 `1分钟` 就是 `60`，`1小时` 是 `3600`（60秒×60分钟），`1天` 是 `86400`（24小时×60分钟×60秒），以此类推。

下面是一些使用时间单位的实用案例：

```
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));
}
```

将结构体作为参数传入

由于结构体的存储指针可以以参数的方式传递给一个 `private` 或 `internal` 的函数，因此结构体可以在多个函数之间相互传递。遵循这样的语法：

```
function _doStuff(Zombie storage _zombie) internal {
  // do stuff with _zombie
}
```

这样我们可以将某僵尸的引用直接传递给一个函数，而不用是通过参数传入僵尸ID后，函数再依据ID去查找。

使用 `for` 循环

`for` 循环的语法在 Solidity 和 JavaScript 中类似。

来看一个创建偶数数组的例子：

```
function getEvens() pure external returns(uint[]) {
  uint[] memory evens = new uint[](5);
  uint counter = 0;

  for (uint i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
      evens[counter] = i;
      counter++;
    }
  }
  return evens;
}
```

这个函数将返回一个形为 `[2,4,6,8,10]` 的数组。

扩展实验一：支付系统

收付款

`payable` 修饰符

`payable` 方法是让 Solidity 和以太坊变得如此酷的一部分 —— 它们是一种可以接收以太的特殊函数。

先放一下。当你在调用一个普通网站服务器上的API函数的时候，你无法用你的函数传送美元——你也不能传送比特币。

但是在以太坊中， 因为钱 (*以太*), 数据 (*事务负载*)， 以及合约代码本身都存在于以太坊。你可以在同时调用函数 **并**付钱给另外一个合约。

这就允许出现很多有趣的逻辑， 比如向一个合约要求支付一定的钱来运行一个函数。

来看个例子：

```
contract OnlineStore {
  function buySomething() external payable {
    // 检查以确定0.001以太发送出去来运行函数:
    require(msg.value == 0.001 ether);
    // 如果为真，一些用来向函数调用者发送数字内容的逻辑
    transferThing(msg.sender);
  }
}
```

在这里，`msg.value` 是一种可以查看向合约发送了多少以太的方法，另外 `ether` 是一个内建单元。

这里发生的事是，一些人会从 web3.js 调用这个函数 (从DApp的前端)， 像这样 :

```
// 假设 `OnlineStore` 在以太坊上指向你的合约:
OnlineStore.buySomething().send(from: web3.eth.defaultAccount, value:
web3.utils.toWei(0.001))
```

注意这个 `value` 字段， JavaScript 调用来指定发送多少(0.001) `以太` 。如果把事务想象成一个信封，你发送到函数的参数就是信的内容。 添加一个 `value` 很像在信封里面放钱 —— 信件内容和钱同时发送给了接收者。

> 注意： 如果一个函数没标记为 `payable`， 而你尝试利用上面的方法发送以太，函数将拒绝你的事务。

提现

学习了如何向合约发送以太，那么在发送之后会发生什么呢?

在你发送以太之后，它将被存储进以合约的以太坊账户中， 并冻结在哪里 —— 除非你添加一个函数来从合约中把以太提现。

你可以写一个函数来从合约中提现以太，类似这样:

```
contract GetPaid is Ownable {
  function withdraw() external onlyOwner {
    owner.transfer(this.balance);
  }
}
```

注意我们使用 `Ownable` 合约中的 `owner` 和 `onlyOwner`，假定它已经被引入了。

你可以通过 `transfer` 函数向一个地址发送以太， 然后 `this.balance` 将返回当前合约存储了多少以太。 所以如果100个用户每人向我们支付1以太， `this.balance` 将是100以太。

你可以通过 `transfer` 向任何以太坊地址付钱。 比如，你可以有一个函数在 `msg.sender` 超额付款的时候给他们退钱:

```
uint itemFee = 0.001 ether;
msg.sender.transfer(msg.value - itemFee);
```

或者在一个有卖家和卖家的合约中， 你可以把卖家的地址存储起来， 当有人买了它的东西的时候，把买家支付的钱发送给它 `seller.transfer(msg.value)`。

扩展实验二：战斗升级系统

可能用得上的知识

用 `keccak256` 来制造随机数

Solidity 中最好的随机数生成器是 `keccak256` 哈希函数.

我们可以这样来生成一些随机数

```
// 生成一个0到100的随机数:
uint randNonce = 0;
uint random = uint(keccak256(now, msg.sender, randNonce)) % 100;
randNonce++;
uint random2 = uint(keccak256(now, msg.sender, randNonce)) % 100;
```

这个方法首先拿到 `now` 的时间戳、 `msg.sender`、 以及一个自增数 `nonce` （一个仅会被使用一次的数，这样我们就不会对相同的输入值调用一次以上哈希函数了）。

然后利用 `keccak` 把输入的值转变为一个哈希值, 再将哈希值转换为 `uint`, 然后利用 `% 100` 来取最后两位, 就生成了一个0到100之间随机数了。