实验五：认识智能合约&线上 IDE 实现 Solidity 合约

一、实验概述

本实验参考自 Loom Network 团队的智能合约教学案例和以太坊中的以太猫游戏，进行 Solidity 智能合约入门与 remix 在线 IDE 使用练习，通过构建一个“宠物游戏”来学习智能合约的编写，在实验中穿插 Solidity 基础知识。

二、预备知识

- **Solidity**：Solidity 是一种静态类型的编程语言，用于开发在 EVM 上运行的智能合约。 Solidity 被编译为可在 EVM 上运行的字节码。借由 Solidity，开发人员能够编写出可自我运行其欲实现之商业逻辑的应用程序，该程序可被视为一份具权威性且永不可悔改的交易合约。对已具备程序编辑能力的人而言，编写 Solidity 的难易度就如同编写一般的编程语言。

  Gavin Wood 最初在规划 Solidity 语言时引用了 ECMAScript 的语法概念，使其对现有的 Web 开发者更容易入门；与 ECMAScript 不同的地方在于 Solidity 具有静态类型和可变返回类型。而与当前其他 EVM 目标语言（如 Serpent 和 Mutan）相比，其重要的差异在于 Solidity 具有一组复杂的成员变量使得合约可支持任意层次结构的映射和结构。Solidity 也支持继承，包含 C3 线性化多重继承。 另外还引入了一个应用程序二进制接口（ABI），该接口（ABI）可在单一合同中实现多种类型安全的功能。

  以下为使用 Solidity 编写的程序示例：

```
contract GavCoin
{
  mapping(address=>uint) balances;
  uint constant totalCoins = 100000000000;

  /// Endows creator of contract with 1m GAV.
  function GavCoin(){
      balances[msg.sender] = totalCoins;
  }

  /// Send $((valueInmGAV / 1000).fixed(0,3)) GAV from the account of
$(message.caller.address()), to an account accessible only by $(to.addr
ess()).
  function send(address to, uint256 valueInmGAV) {
    if (balances[msg.sender] >= valueInmGAV) {
      balances[to] += valueInmGAV;
      balances[msg.sender] -= valueInmGAV;
    }
  }
```

```
  /// getter function for the balance
  function balance(address who) constant returns (uint256 balanceInmGA
V) {
    balanceInmGAV = balances[who];
  }
}
```

- **Remix**：Remix 是以太坊提供的一个开发 Solidity 智能合约的网络版开发软件。合约的开发者在 Remix 里提供的 JavaScript 虚拟机上开发，调试好合约后，可以发布到以太坊，或者任何支持 Solidity 智能合约的区块链上。

- **以太猫**：在以太坊中，每只以太猫都有自己的一个独一无二的 DNA，每只“以太猫”有两个关键部分构成：

  （1）一段存储在以太坊转账中的文本，用于表示猫的 DNA 序列.

  （2） 一个可爱的图形化的猫的形象，这个猫咪形象由游戏的发行公司 Axiom Zen 提供。

首先，每条“以太猫”转账记录都绑定了一个以太坊钱包地址（这个地址就是猫的主人的钱包地址）。转账中还记录了猫的 DNA 序列，这个序列定义了猫的一系列属性，比如胡须形状，眼睛形状，毛色，条纹等等。

显然，这些属性是需要依赖第二个部分，也就是 Axiom Zen 这家公司提供的图形化界面来将“以太猫”显示在你的浏览器中的。“以太猫”的 DNA 其实就是一串文本字符，比如下面这个就是这只标号为 207454 的猫的 DNA：`000042d28314c7305c97b94300c0c31c44638c92c0b228ca6104217a52e4b56b`

三、实验准备

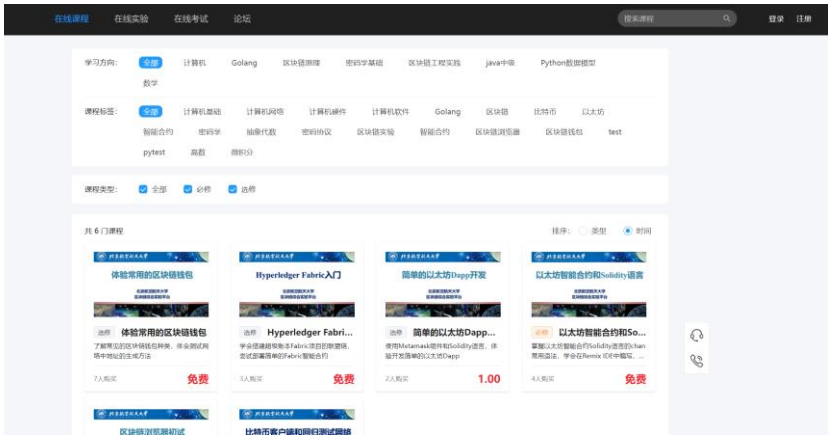
- 最新版 Chrome 浏览器。在实验材料中，提供了 chrome 浏览器的 windows 安装包，有需要的同学可以自行安装。

四、在线实验平台使用说明

本实验提供了区块链在线实验平台供同学们 进行在线实验，下面介绍在线实验平台的使用方法。

1、注册和登录

1a.打开 Chrome 浏览器，转到网址 http://www.knowonchain.com/，会看到如下页面。

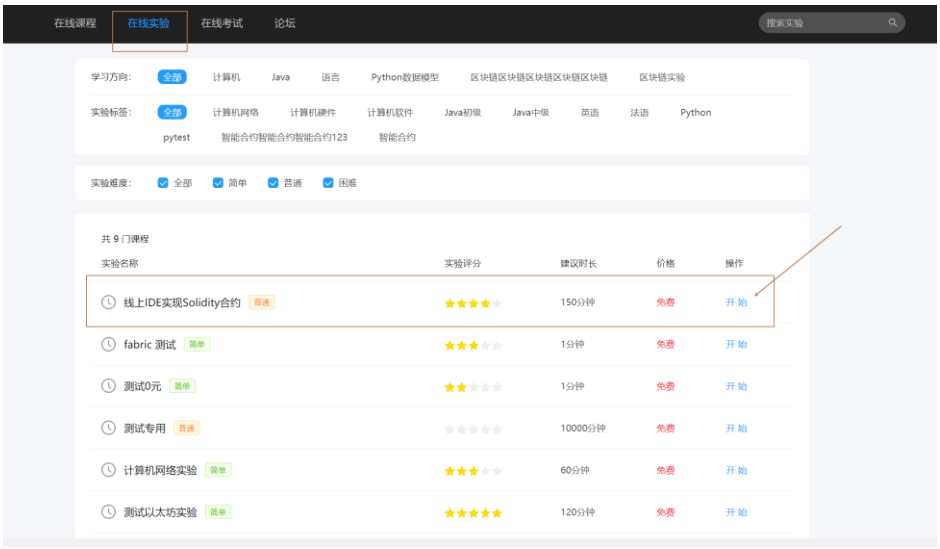


1b.点击右上角的“注册”，进入新用户注册页面。同学们可以选择手机或者邮箱方式进行注册。

1c.注册成功后，系统会自动跳转到登录界面，此时输入注册使用的手机号/邮箱和密码即可登录进系统。

2、进入实验

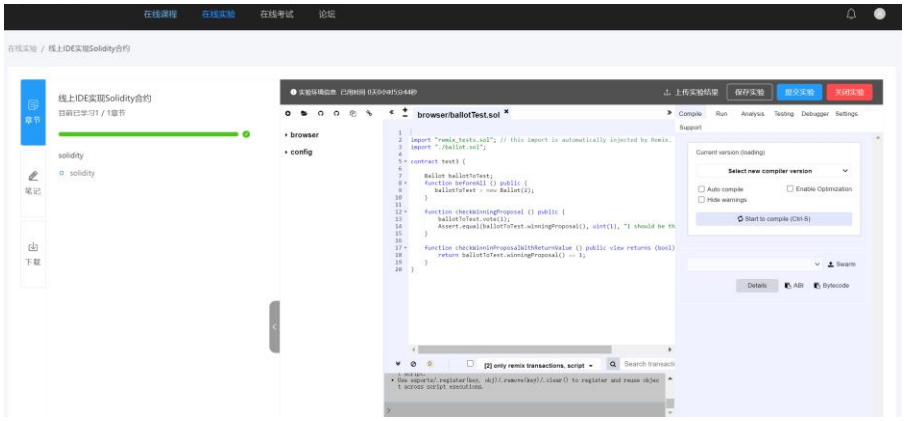
2a.登录进系统后，选择最上面一栏的“在线实验”，在实验列表中选择“线上 IDE 实现 Solidity 合约”，点击“开始”。



2b.点击“创建资源”。

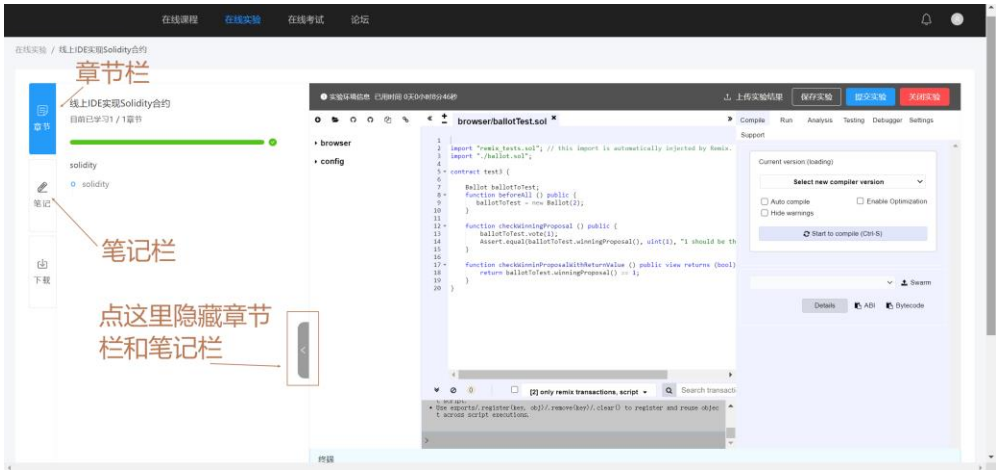
2c.点击“立即学习”。

2d.看到此界面即表示成功进入实验环境。

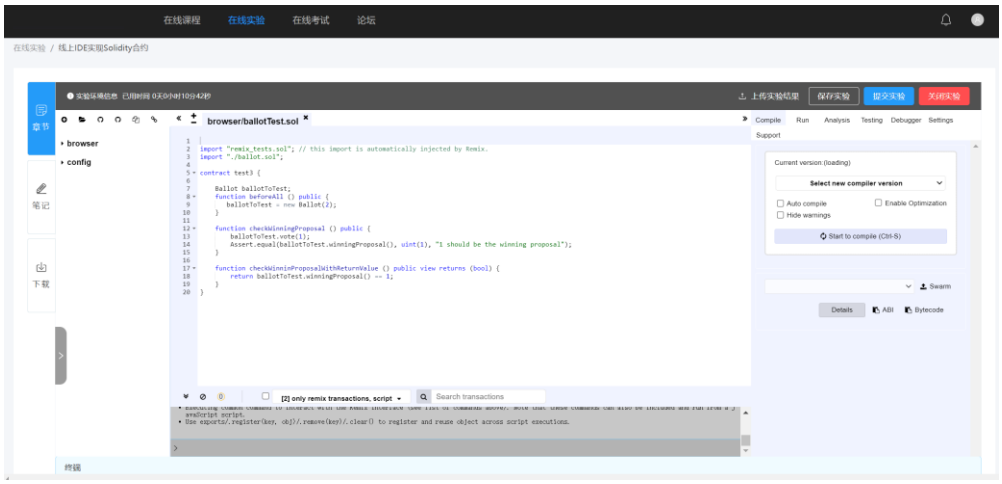


3、IDE 使用指南

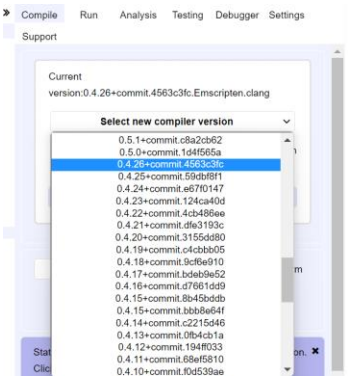
3a.左边是实验章节和笔记栏，在笔记栏可以自己编辑笔记，点击伸缩按钮可以隐藏章节和笔记栏。



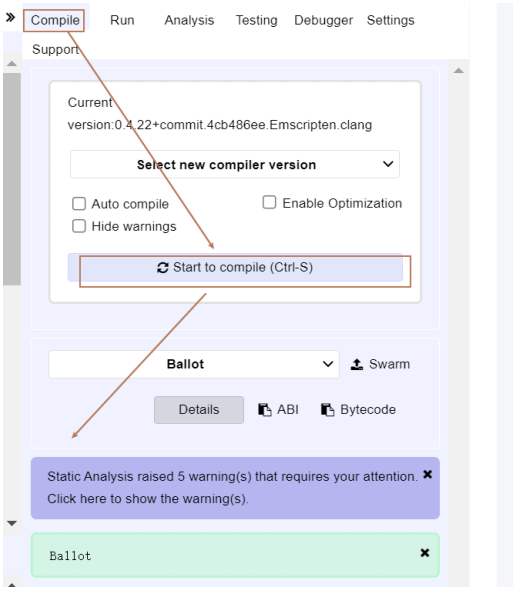
3b.这是章节栏和笔记栏隐藏之后的效果。



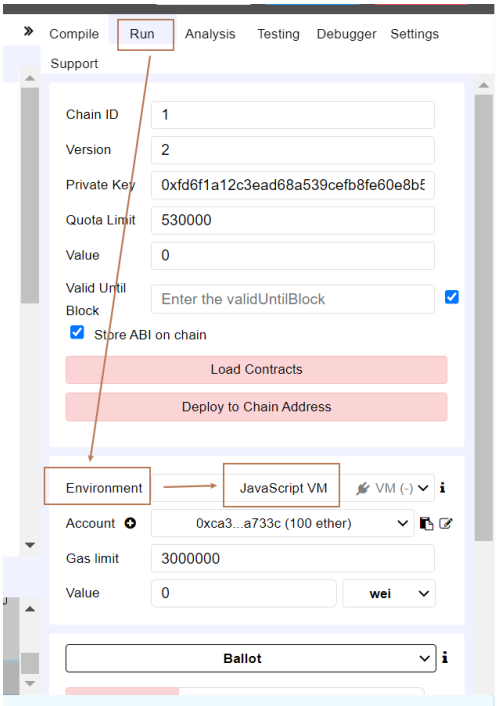
3c.选择编译器版本为 0.4.26。



3d.点击 Compile 模块中的 Start to Compile，即可实现对合约代码的编译，如果没有出现红色的提示框，即表示合约编译通过。



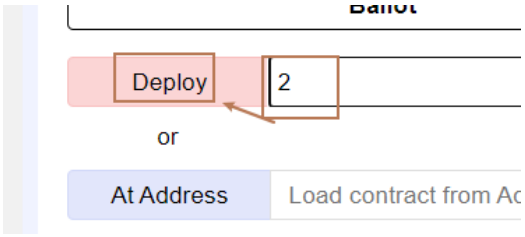
3e.点击 Run 模块中的 Environment 部分的下拉框，选择 JavaScript VM。这意味着我们现在拥有了一条本地的私链环境，有助于在后续实验流程中的合约快速部署。有兴趣的同学也可以选择接入以太坊的测试网络，结合 metamask 插件进行合约部署和调用，但这样会使得合约交易的相应速度很慢，不利于实验的进行。

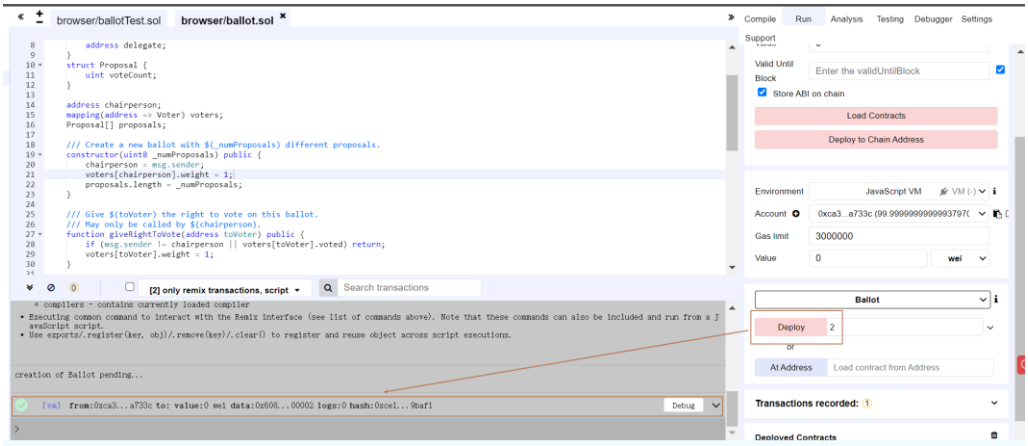


3f.第一张图：合约代码里的 constructor 函数是在部署合约时要运行的初始化函数，可以看到，在右边的部署合约（Deploy）选项里，系统已经自动识别出需要我们输入的参数（变量名：_numProposals，类型：uint8）

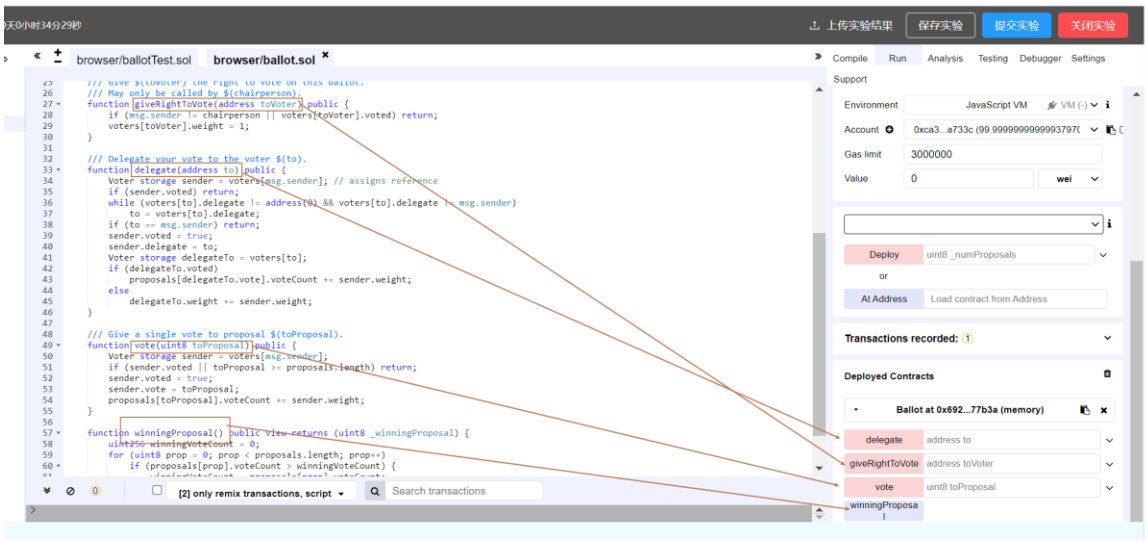
第二张图：输入变量的值，在这里我们输入 2，再点击 Deploy，就可以实现对合约的部署。

第三张图：部署合约之后，可以看到在左下方的日状态栏中，出现了一笔成功的交易单，这个交易单就对应着刚才我们部署的合约。

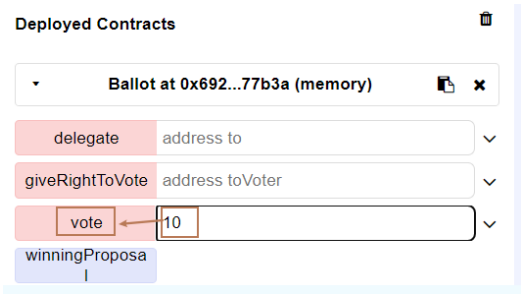




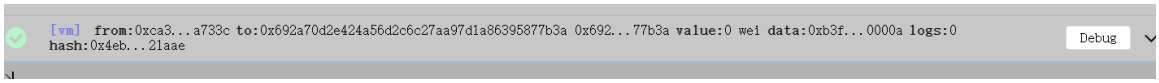
3f.已经部署的合约可以在 Run 模块的 Deployed Contracts 看到，系统已经识别出我们在合约代码中定义的 public 类型的函数的输入格式。



3g.尝试调用 vote 函数，输入 10，点击函数名。（点开右方的展开箭头，可以看到此函数的更详细的输入信息，一般在一个函数有多个输入时使用）



3h.可以通过新的成功交易初步判断调用成功。



五、实验内容

在本实验中，每一个小实验将分为“基础知识”与“实验内容”两个部分，基础知识中将包含实验所需知识点，可及时向前翻阅。

实验 1：Solidity 基础

电子宠物很多人都知道，乃至养过，其中以拓麻歌子最为著名。本实验则以电子宠物为背景，来用智能合约的方式创造一个新的”电子宠物世界“。

实验一目的创造一个“宠物孵化池”， 通过孵化池，就能够从中诞生一个全新的宠物。那么这个孵化池需要满足这么几个功能：

- 数据库中能够记录所有的宠物信息
- 应该在孵化池中存在这样一个接口，让我们能够从中孵化宠物
- 每个宠物都应该是独一无二的

宠物 DNA

参考以太猫的 DNA，宠物的独立标识则为他的 DNA。本实验定义的 DNA 很简单，由一个 16 位的整数组成：

```
8356281049284737
```

事实上，每一只电子宠物也是由这样一串数字构成，他的各位则表示了不同的属性。比如前 2 位表示物种，紧接着的 2 位表示有没有翅膀，等等。

Ps：我们并没有真实的图片，宠物样貌可以存在于各位的想象中 XD

实验 1 的基础知识

合约

Solidity 的代码都在**合约**里面. 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” — 这目的是标明 Solidity 编译器的版本. 从而避免将来新的编译器可能破坏你的代码。

例如: `pragma solidity ^0.5.12;`

我们也可以指定一个版本区间，比如 `pragma solidity >=0.4.12 <0.6.0;`

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 ^0.5.12;

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; // 等价于 5^2 = 25
```

结构体

Solidity 中也提供了 **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 alice = Person(199, "Alice");
people.push(alice);
```

你也可以用一行代码:

```
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 (stri
ng) {
}
```

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。

显然不是总需要 Public 属性的函数，而且这样的合约易于受到攻击，所以可以把自己的函数定义为 private 属性，当需要外部调用它时才将它设置为 public。

定义一个私有函数只需要在其后添加 private 关键字。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：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;
}
```

上面的函数没有改变 Solidity 里的状态，只是进行读取，这种情况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，返回值完全取决于输入参数，例如：

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

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，仅编写一遍作为了解。

实验 1 的实验内容

1. 清空原有.sol 文件，新建文件 AnimalIncubators.sol
2. 为了建立宠物部队，先建立一个基础合约 AnimalIncubators，并指定 Solidity 编译器版本。
3. 因为我们的宠物 DNA 由十六位数字组成，所以首先，需要在合约中定义 dnaDigits 为 uint 数据类型, 并赋值 16，用来表示 dna 的位数。
4. 为了保证我们的宠物的 DNA 只含有 16 个字符，我们需要一个 uint 变量等于 10^16。便于后续取模。
   - 建立一个 dnaLength 变量, 令其等于 10 的 dnaDigits 次方.
5. 创建宠物结构体
   - 建立一个名为 Animal 的结构体. 在其中有两个属性： name (类型为 string), 和 dna (类型为 uint)。
6. 准备工作完成后，首先需要将宠物们保存在合约中，并且让其它合约也能够看到这些宠物们，因此需要一个公共数组。
   - 创建一个 Animal 的结构体数组，用 public 修饰，命名为 animals.
7. 定义一个事件 NewAnimal。 它有 3 个参数: AnimalId (uint)， name (string)，和 dna (uint)。
8. 定义一个 孵化宠物函数，其功能为：孵化一个新宠物并添加入 animals 数组中

- 建立一个私有函数 _createAnimal。 它有两个参数: _name (类型为 string), 和 _dna (类型为 uint)。
- 在函数体里新创建一个 Animal， 然后把它加入 animals 数组中。 很显然新创建的宠物属性就来自于函数的入参，同时将 animals 的索引值记录下来为 animalId。
- 在函数结束触发事件 NewAnimal。

9. 定义 DNA 生成函数：能够根据字符串随机生成一个 DNA。

- 创建一个函数 _generateRandomDna，将属性标记为 private。它只接收一个输入变量 _str (类型 string), 返回一个 uint 类型的数值。此函数只读取我们合约中的一些变量，所以可以标记 view 属性。
- 使用以太坊内部的哈希函数 keccak256，根据输入参数_str 来生成一个十六进制数，类型转换为 uint 后，返回该值的后 dnaLength 位。

  Ethereum 内部有一个散列函数 keccak256，它用了 SHA3 版本。一个散列函数基本上就是把一个字符串转换为一个 256 位的 16 进制数字。字符串的一个微小变化会引起散列数据极大变化。 这在 Ethereum 中有很多应用，但是现在我们只是用它造一个伪随机数。

使用样例：keccak256("abcdefg")

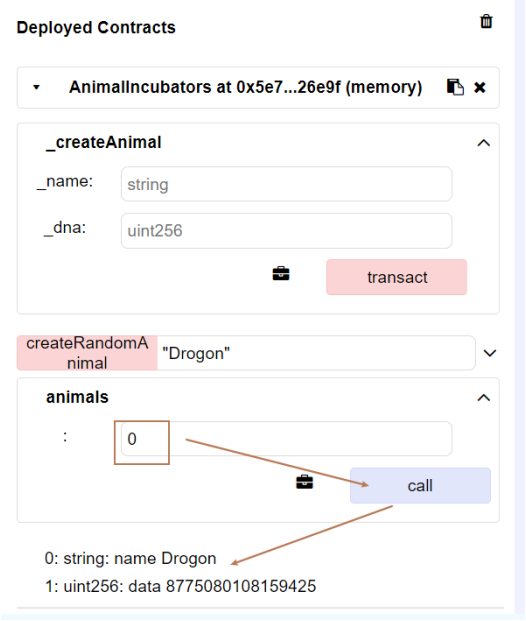
10. 定义 第一个**公共**函数来把上面定义的若干部件组合起来，使得实现这样的一个功能：该函数能够接收宠物的名字，然后用这个名字来生成宠物的 DNA

    - 创建一个 public 函数，命名为 createRandomAnimal. 它将接收一个变量 _name (数据类型是 string)。
    - 首先调用 _generateRandomDna 函数，传入 _name 参数来生成一个 DNA，取名为 randDna。
    - 调用 _createAnimal 函数，将这个新生成的宠物记录下来， 传入参数： _name 和 randDna。

实验 1 需实现效果

在 JavaScript VM 环境下，部署 AnimalIncubators 合约。创建三个分别叫 Drogon、Rheagal、Viserion 的宠物，向助教展示其 DNA。

下图中，创建了第一个宠物后，可以 call 一下 animals 数组，会直接显示返回结果。



实验 2：Solidity 进阶——宠物成长系统

实验一中创建了一个函数用来生成宠物，并且存入区块链上的宠物数据库中。 实验二中，我们会模拟以太猫的繁殖机制，创建一个宠物成长系统，让宠物可以通过进食进行成长，系统会通过宠物和食物的 DNA 计算出新宠物的 DNA，

实验 2 的基础知识

Addresses （地址）

以太坊没有采用比特币中的 UTXO 模型，而是由类似银行账号的 ***account*** (账户)组成。账户的余额是 ***以太***。每个账户都有一个“地址”作为其唯一标识符，比如：

```
0xa5a82d26dbb1dd2f35dba16b1aadb2fda77d71be
```

地址可以属于外部用户也可以属于智能合约。我们可以指定“地址”作为宠物主人的 ID。当用户调用合约创建新的宠物时，新宠物的所有权就分配给了调用者的以太坊地址。

Mapping（映射）

除了 ***结构体*** 和 ***数组***，***映射*** 也是一种在 Solidity 中存储有组织数据的方法。

映射是这样定义的：

```
//对于银行账户，将用户的余额保存在一个 uint 类型的变量中：
mapping (address => uint) public Balance;
//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，有些全局变量所有函数都可以调用，比如 msg.sender，它指的是当前调用者（或智能合约）的 `address`，`solidity` 中，所有合约都要从被外部调用才会执行，所以总会有 `msg.sender` 存在。

以下是使用 `msg.sender` 来更新 `mapping` 的例子：

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

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

```
function whatIsMyid() 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 setMyid， 则值为 `0`
  return id[msg.sender];
}
```

这个例子中，每个人都可以调用 setMyid 存储自己最喜欢的 id，然后再调用 whatIsMyid 来查询自己的 id，这样是很安全的，因为在区块链中，没有你的私钥任何人都不能篡改你的地址关联的数据。

require（要求）

require 使得函数必须满足某些条件才能执行，否则抛出错误，并停止执行：

```
function sayHiTo Satoshi (string _name) public returns (string) {
  // 比较 _name 是否等于 " Satoshi ". 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("Satoshi "));
  // 如果返回 true，运行如下语句
  return "Hi!";
}
```

如果你这样调用函数 sayHiTo Satoshi（“Satoshi”）,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。

所以在调用函数前常常会使用 require 来检查是否满足条件。

Inheritance（继承）

当代码过于冗长的时候，可以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，所以 Doge 和 BabyDoge 中的函数都可以访问，需要这种逻辑继承时可以使用，也可以为了将代码分散到不同合约中便于组织。

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。

一般情况下不用管这俩关键字， 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， 子合约可以访问父合约中定义的内部函数。

`external` 与 `public` 类似，只不过这些函数只能在合约之外调用 ，而不能被合约内的其他函数调用。

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 2 的实验内容

1. 首先使用地址给宠物指定“主人”，为了存储宠物的所有权，我们会使用到两个映射：一个记录宠物拥有者的地址，另一个记录某地址所拥有宠物的数量。

   - 创建一个叫做 AnimalToOwner 的映射。其键是一个 uint（我们将根据它的 id 存储和查找宠物），值为 address。映射属性为 public。
   - 创建一个名为 ownerAnimalCount 的映射，其中键是 address，值是 uint。

2. 修改_createAnimal 函数来使用映射

   - 首先，在得到新的宠物 animalId 后，更新 AnimalToOwner 映射，在 AnimalId 下面存入 msg.sender。
   - 然后，我们为这个 msg.sender 名下的 ownerAnimalCount 加 1。

3. 在 createRandomAnimal 开头使用 require 来确保这个函数只有在每个用户第一次调用它的时候执行，用以创建初始宠物。 判断方式：判断该用户的宠物数是否为 0

4. 创建新文件 AnimalFeeding.sol 文件，在其中创建 AnimalFeeding 合约，继承自 AnimalIncubators。记得设置编译版本与 import。

5. 在 AnimalFeeding 合约中，增加进食功能： 当一个宠物进食后，它自身的 DNA 将与食物的 DNA 结合在一起，形成一个新的宠物 DNA，

   - 创建一个名为 feedAndGrow 的函数。 包含两个参数：_AnimalId （ uint ）、_targetDna （ uint ），分别表示宠物、食物。 设置属性为 public 。
   - 要求只有宠物的主人才能给宠物喂食，在函数开始添加一个 require 语句来确保 msg.sender 只能是这个宠物的主人。
   - 声明一个名为 myAnimal 数据类型为 Animal 的本地变量（这是一个 storage 型的指针），将其值设定为在 animals 数组中索引为 _AnimalId 所指向的值。

6. 完成 feedAndGrow 函数：

   - 取得_targetDna 的后 dnaDigits 位
   - 生成新的宠物 DNA：计算宠物与食物 DNA 的平均值
   - 为宠物添加标识：将新的宠物 DNA 最后两位改为“99”。
   - 调用_createAnimal 生成新宠物，新宠物名字为“No-one”。**（需要修改 _createAnimal 函数属性使对继承可见）**。

7. 在 AnimalFeeding 合约中增加捕食函数：

```
function _catchFood(uint _name) internal pure returns (uint) {
      uint rand = uint(keccak256(_name));
      return rand;
}
```

8. 在 AnimalFeeding 合约中，增加进食功能：宠物抓住食物后进食，然后成长为一个新宠物
   - 创建一个名为 feedOnFood 的函数。它需要 2 个 uint 类型的参数，_AnimalId 和_FoodId，这是一个 public 类型的函数。
   - 调用_catchFood 函数，获得一个食物 DNA。
   - 调用 feedAndGrow 函数。传入宠物 id 和食物 dna，调用 feedAndGrow。

实验 2 需实现效果

部署 AnimalFeeding 合约，实现以下效果：

- 同一账户只可调用一次 createRandomAnimal
- 以三个用户身份添加 Drogon、Rheagal、Viserion 的宠物
- 让 Drogon 宠物进食成长一次，展示新宠物的主人与 Drogon 相同

实验 3：Solidity 高阶理论

Solidity 很像 JavaScript，但是以太坊上的合约与普通的程序最大的区别就是以太坊上的合约代码一旦上链就无法更改，即使合约出现 bug，也无法进行修改，之恶能放弃这个合约让用户去使用一个新的被修复过的合约，虽然这可能会造成使用上的不便，但这也是智能合约的优势之一，一旦你的代码上链，就无法被他人恶意篡改，其他人调用也只能以预设的逻辑一直执行下去。

实验 3 的基础知识

合约所有权

因为智能合约无法篡改，所以开发者可能需要留一些后门来处理 bug，但是由于合约上链之后一般就要被公开，所以这个后门也就暴露给了所有人，开发者当然不希望所有人都能随意调用后门，所以开发者会给这个合约指定所有权，只有自己才能调用这个后门。

OpenZeppelin 库的 Ownable 合约

OpenZeppelin 库的 `Ownable` 合约

下面 `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 a
uthorization control
 * functions, this simplifies the implementation of "user permissions".
 */
contract Ownable {
  address public owner;
  event OwnershipTransferred(address indexed previousOwner, address ind
exed newOwner);

  /**
   * @dev The Ownable constructor sets the original `owner` of the cont
ract 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;
  }
}
```

其中包括：

- 构造函数：function Ownable()是一个 **constructor** (构造函数)，构造函数与合约同名，只在合约第一次创建的时候执行一次，而且这个构造函数不是必须的。
- 函数修饰符：modifier onlyOwner()。修饰符看起来跟函数很像，不过是跟在“宿主”函数后面用来修饰“宿主”函数的，可以在函数执行前，为它检查下先验条件。比如可以写个修饰符 onlyOwner 检查调用者，确保只有合约的所有者才能运行此函数。我们之后会详细讲述修饰符，以及那个奇怪的 _;。

Ownable 合约大概的流程就是这样：

1. 创建合约，先写构造函数，将 msg.sender（其部署者）设置为 owner。
2. 为它加上一个修饰符 onlyOwner，只有合约的所有者 owner 才能进行访问。
3. 允许将合约所有权转让给他人。

onlyOwner 非常常见，很多合约的开头都有这段合约代码，然后从 Ownable 继承子类，再进行功能和逻辑的开发。

函数修饰符

函数修饰符看着跟函数很像，不过编译器会识别这个关键字 modifier，知道它不是函数，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 type(string language);

  function speak() external onlyOwner {
    type("English");
  }
}
```

注意 speak 函数后的 onlyOwner 修饰符。 当你调用 speak 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 speak 中的代码。

函数修饰符最常见的用途就是放在函数执行之前添加快速的 require 检查。

添加修饰符 onlyOwner 后，只有**合约的 owner**（也就是开发者）才能调用它。

> 注意：虽然开发者拥有后门是由正当理由的，但用户也要小心，认真阅读源代码，避免开发者留下恶意后门，比如偷走你的宠物；而作为开发者，既要给自己留些后门又要让用户安心地使用合约。

带参数的函数修饰符

之前介绍了一个简单的函数修饰符：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 18 周岁才允许开车.
// 我们可以用如下参数调用`olderThan` 修饰符:
function driveCar(uint _userId) public olderThan(18, _userId) {
  // 其余的程序逻辑
}
```

olderThan 通过“宿主”函数 driveCar 传递参数。

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

Solidity 中，用户每次执行合约都会花费 gas，而 gas 需要用以太币购买，DApp 消耗多少 gas 取决于程序的复杂程度，比如存储操作就会比计算花销多，每个操作都会计算其花费的计算资源，最后计算所有的 gas 花费。

因为运行合约需要花费 gas，也就是相当于花钱运行，所以智能合约特别强调优化，同样的功能，用户肯定会选择优化更好花费更少的合约。

为什么要用 *gas* 来驱动？

以太坊就像一个去中心化的巨大缓慢但非常安全的电脑。当用户运行合约的时候，以太坊上的所有节点都会进行运算并验证，从而确保链上的数据不会被恶意篡改。为了防止代码无限循环或者进行密集运算阻塞网络，想在以太坊上运算或者存储数据都需要花费 gas。

省 gas 的招数：结构封装 （Struct packing）

除了基本的 `uint` 外，还有其他变种 `uint`：`uint8`，`uint16`，`uint32` 等。

一般我们不会使用 `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 `uint256` 不会节省任何 gas。

而如果把 `uint` 绑定到 `struct` 里面，即一个 `struct` 中有多个 `uint`，那么编译器就会尽可能使用较小的 `uint`，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，在 struct 中定义 uint 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”

当用户外部调用 view 函数时不需要花费 gas，因为 view 函数只会读取数据，而不会改变链上的数据，当编译器读取 view 函数时，它会知道这个函数只是查询本地节点的数据，而不需要创建新的事务，所以不需要花费 gas，所以你需要在只进行读取的函数上标明 external view，从而减少合约中的 gas 花费。

> 注意：如果一个 view 函数被另一个不属于同一合约的函数调用，是要花费 gas 的。这是因为主调函数会在以太坊创建一个事务，网络上的节点仍然要去计算验证。

减少存储开销

Solidity 使用 storage(存储)是相当昂贵的，因为写入或是更改一段数据，都会永久性地记录上链，全世界的数千个节点都会进行存储 。所以为了降低成本，应尽量避免将数据写入存储，这可能会导致程序上的不变，比如每次运行都要重建数组并计算，而不是直接读取已经存储的数组，在很多编程语言中遍历数据开销都会很大，不过还好 solidity 中有 external view 函数，可以不花 gas 就能进行数据遍历。

如何在内存中声明数组

在数组后面加上 memory 关键字，代表这个数组仅在内存中使用，使用完就舍弃无需上链，相比存储在链上，可以大大节省开销。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;
}
```

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）。

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 private 或 internal 的函数可以接收结构体的指针作为参数，所以结构体可以在多个函数之间相互传递。语法如下：

```
function _doStuff(Animal storage _Animal) internal {
  // do stuff with _Animal
}
```

这样我们可以将某宠物的引用直接传递给一个函数，而不用是通过参数传入宠物 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。

实验 3 的实验内容

1. 将 Ownable 代码复制一份到新文件 ownable.sol 中，让 AnimalIncubators 作为子类继承 Ownable。

2. 给 createRandomAnimal 函数添加 onlyOwner，再用不同的账户进行调用，看看有什么区别，完成后再删掉它，毕竟其他玩家也要调用。

3. 在 AnimalIncubators.sol 中给宠物添 2 个新属性： level（uint32）和 readyTime（uint32） - 一个是等级，一个是进食的冷却时间。

4. 给 DApp 添加一个"冷却周期"的设定，让宠物两次进食之间必须等待 **1min**。

   - 声明一个名为 cooldownTime 的 uint，并将其设置为 1 分钟
   - 修改_createAnimal 中添加。

   注： now 返回类型 uint256 ， 需要类型转换。

   再来到 AnimalFeeding.sol 的 feedAndGrow 函数：

   - 修改可见性 internal 以保障合约安全。
   - 在_targetDna 计算前，检查该宠物是否已经冷却完毕。
   - 在函数结束时重新设置宠物冷却周期，以表示其捕食行为重新进入冷却。

5. 接下来撰写一个属于宠物自己的函数修饰符，让宠物能够在达到一定水平后获得特殊能力：

   - 创建一个新的文件 AnimalHelper.sol, 定义合约 AnimalHelper 继承自 AnimalFeeding
   - 创建一个名为 aboveLevel 的 modifier，它接收 2 个参数， _level (uint) 以及 _AnimalId (uint)。
   - 函数逻辑确保宠物 Animals[_AnimalId].level 大于或等于 _level。
   - 修饰符的最后一行为 _;，表示修饰符调用结束后返回，并执行调用函数余下的部分。

6. 添加一些使用 aboveLevel 修饰符的函数来作为达到 level 的奖励。激励玩家们去升级他们的宠物：

   - 创建一个名为 changeName 的函数。它接收 2 个参数：_AnimalId（uint 类型）以及 _newName（string 类型），可见性为 external。它带有一个 aboveLevel 修饰符，调用的时候通过 _level 参数传入 2，当然，别忘了同时传 _AnimalId 参数。 在函数中使用 require 检查 msg.sender 是否是宠物主人，如果是则将宠物名改为_newName
   - 在 changeName 下创建另一个名为 changeDna 的函数。它的定义和内容几乎和 changeName 相同，不过它第二个参数是 _newDna（uint 类

型），在修饰符 aboveLevel 的 _level 参数中传递 20 。在函数中可以把宠物的 dna 设置为 _newDna 。

7. 定义一个新函数 getAnimalsByOwner 来获取某个玩家的所有宠物：
   - 函数有一个参数_owner(address)，声明为 external view 属性，返回一个 uint 数组。
   - 声明一个名为 result 的 uint [] memory （内存变量数组），其长度为该_owner 拥有的宠物数量。
   - 使用 for 循环遍历 Animals 数组，将主人为_owner 的宠物添加入 result
   - 返回 result

   这样 getAnimalsByOwner 不花费任何 gas。

实验 3 需实现效果

- 展示宠物的冷却时间。
- 成功实现 getAnimalsByOwner 函数。

实验 4：拓展实验

此部分为拓展实验，不强制要求完成。完成了此部分的同学可以获得额外的加分。

请先完成实验 1 至实验 3，再做拓展实验。

拓展实验 4：solidity 高阶篇

拓展实验 4 的基础知识

payable 修饰符

在函数上增加 payable 标识，即可接受以太。对，他能让一次函数调用变得非常 cool——普通网站的 api 函数可没有这样的功能。

在以太坊中， 因为数据 (*事务负载*)，钱 (*以太*), 以及合约代码本身都存在于以太坊网络中。所以，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。

事实上，一些人会调用这个函数 (比如 DApp 的前端) :

```
// 假设 `OnlineStore` 在以太坊上指向你的合约:
OnlineStore.buySomething().send(from: web3.eth.defaultAccount, value: w
eb3.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

那么在发送之后的以太，会流入到哪里呢？

事实上，发送出来的以太，将被存储进以合约所属的以太坊账户中， 并冻结在哪里，直到所属者通过另一种方式将他提取出来——提现。

我们可以通过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)`。

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;
```

拓展实验 4 的实验内容

1. 在 AnimalHelper 中，添加支付系统——宠物主人可以通过支付 ETH（对，氪金）的方式来强化他们的宠物。这些支付的 ETH 将存储在你拥有的合约中，向你展示如何通过你的合约赚钱。

   - 定义一个名为 powerUpFee 的 uint 类型变量, 将值设定为 0.001 ether。
   - 定义一个名为 powerUp 的函数。 它将接收一个 uint 参数 _animalId。函数记得修饰 external 以及 payable 属性。
   - 这个函数首先应该 require 确保 msg.value 等于 powerUpFee。
   - 函数将增加指定宠物的 ATK 属性: animals[_animalId].ATK++。

   需实现效果：成功升级，并改名。

2. 在 AnimalHelper 中，添加提现系统

   - 创建一个 withdraw 函数，参考上面的 GetPaid 样例。
   - 以太的价格在过去几年内不停地波动，所以我们应该再创建一个函数，允许我们以合约拥有者的身份来设置 powerUpFee。

     a. 创建一个函数，名为 setPowerUpFee， 其接收一个参数 uint _fee，是 external 并标记为仅 owner 可用。

     b. 这个函数会将 powerUpFee 等于 _fee。

   需实现效果：实现一次提现操作。

3. 实现战斗升级系统

   新建一个文件 AnimalAttack.sol，在其中新建一个继承自 AnimalHelper 的合约 AnimalAttack，在这之下编辑新的合约的主要部分。

   需实现效果：

- 你选择一只自己的宠物，然后选择一个对手的宠物去战斗。
- 如果你是战斗发起方（先手），你将有 75%的几率获胜，防守方将有 25%的几率获胜。
- 每一只宠物（攻守双方）都会有一个 winCount 和一个 lossCount，用来记录该宠物的战斗结果。
- 若发起方获胜，这只宠物的 ATK 将增加。
- 如果攻击方失败，除了失败次数将加一外，什么都不会发生。
- 无论输赢，当前宠物的战斗冷却时间都将被重置（很显然，饿了嘛）。

六、参考文献

[1] https://cryptozombies.io/，访问时间：2021.10.19

[2] https://ethereum.github.io/yellowpaper/paper.pdf，访问时间：2021.10.19

[3] https://docs.soliditylang.org/en/v0.8.11/，访问时间：2021.10.19

[4] https://ethereum.org/en/developers/docs/standards/tokens/erc-20/，访问时间：2021.10.19

[5]https://drive.google.com/file/d/1ikZHnJXyaCHmjmAb5qubaDFREMX1F6PK/view，访问时间：2021.10.19