北京大学
PEKING UNIVERSITY

区块链课程

孙惠平
sunhp@ss.pku.edu.cn

北京大学 软件与微电子学院
School of Software and Microelectronics, Peking University

北京大学
PEKING UNIVERSITY

PART 第六章

智能合约

目 录

CONTENTS

第1节

智能合约概述

01 智能合约的概念

区块链是一个在非安全条件下的分布式安全数据库，它维护着一个体量不断增长的数据列表。我们已经知道，一旦将某数据写入区块链中，除非节点之间达成修改该数据所在区块的所有后续区块数据的共识，那么任何一个节点都无法自行修改或删除该数据。由于区块链具备**共识**的特征，因此区块链可以在**没有第三方管理机构的基础上**为用户提供一个**可靠的安全交易环境**。

区块链技术存在局限性，其中最明显的一点是数据被写入区块链的操作缺少其他的约束条件。为了改善该缺点，Vitalik Buterin发表了以太坊白皮书，他在白皮书中详细描述了以太坊的技术设计和智能合约的结构。

01 智能合约的概念

北京大学 PEKING UNIVERSITY

- 我们给出智能合约的具体定义：

定义

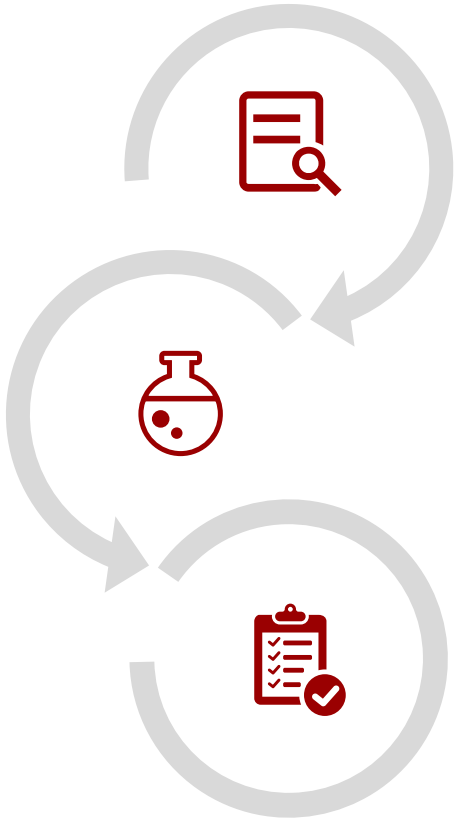
智能合约（Smart Contract）是一种在区块链上运行的计算机程序，这段计算机程序是预先设计好的一套数字化规则，它包含真实世界经多方协商达成一致的业务逻辑，通常是由高级计算机语言（Solidity语言、Go语言、JavaScript语言等）编写出来的。

解读

计算机系统会自动强制执行智能合约中的逻辑，除此之外，网络中的所有节点都会在共识过程中复制并执行这段逻辑。需要注意的是，智能合约在区块链中扮演的角色不只是一个可以自动执行的计算机程序，其本身也可以被看作成区块链系统的参与者。

02 智能合约的功能

- 总的来说，智能合约就像一个可以被信任的节点，它具有以下功能：



存储数据

智能合约可以对接收到的数据进行存储和处理，也可以向外发送数据。

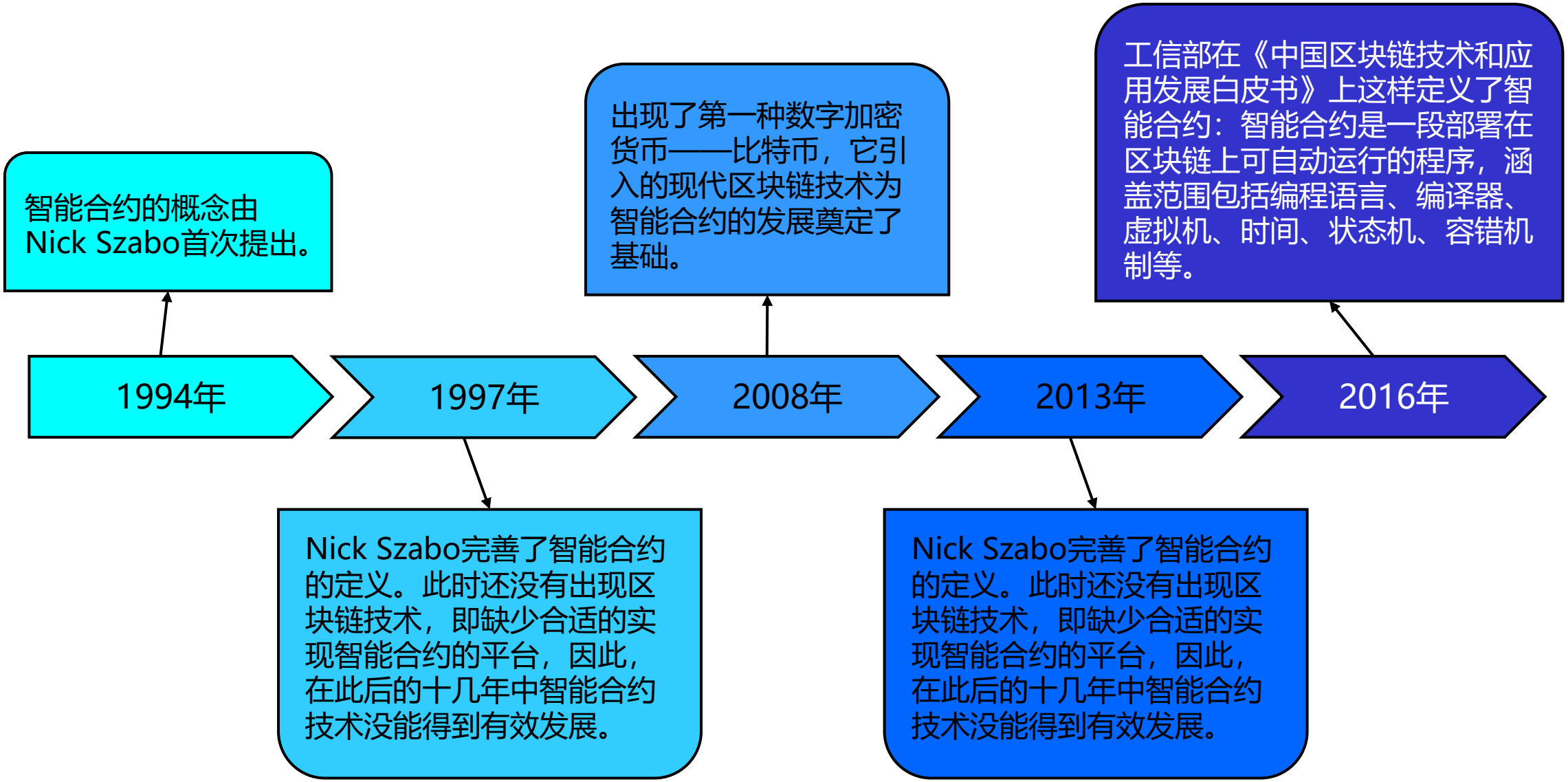
验证数据

智能合约可以验证数据的发送方是否是某个特定的节点。

强制执行事先预定的规则

一旦存在输入数据，智能合约就可以通过强制执行预定的规则产生输出数据，并经过共识将数据保存到区块链上。

03 智能合约的发展历史

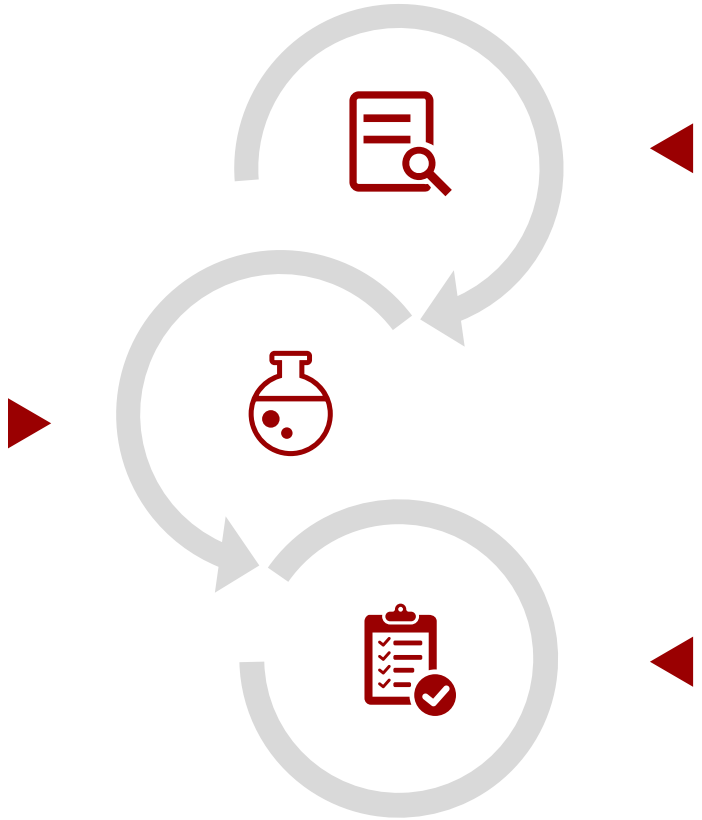


04 智能合约的技术挑战

- 尽管智能合约具有很多强大的功能，但是它仍然存在很多技术挑战等待人们去改进，其中最主要的问题包括以下三个问题：

隐私问题

智能合约需要在多个参与节点上运行，每笔交易的输入和输出数据都需要被参与的节点读取并验证。那么，智能合约隐私问题主要表现在：**如果智能合约上公开的交易数据能够反应用户的行为和敏感信息，就会存在隐私泄露的问题。**



安全问题

智能合约的安全问题主要来自于两个方面：编写的代码和智能合约语言。智能合约需要按照开发者的意图来执行，**如果智能合约中代码的逻辑存在漏洞，那么可能会造成重大损失。**

性能问题

完整执行一次智能合约的时间包括两个方面：**智能合约的运行时间**和**达成共识并将数据写入区块的时间**。在执行智能合约时往往会受到以下三个条件的制约：虚拟机的性能限制、写入区块数据的数量限制和共识机制的等待时间限制，因此智能合约的执行难以达到很高的性能。

05 智能合约的开发平台

- 在现有的众多区块链平台中，很多平台都支持智能合约的开发。其中，使用较广的平台有：以太坊、“超级账本”、Corda平台和Stellar平台。

以太坊

以太坊是一个支持智能合约的去中心化平台，它采用了基于账户的数据模型，每个参与者都由其数字钱包识别。为了支持智能合约的开发，以太坊开发了Solidity语言。

“超级账本”

“超级账本”的名称是Hyperledger Fabric，它是一个用于运行智能合约的分布式账本平台。与以太坊的运行方式不同，“超级账本”采用Docker容器技术来执行代码，这种技术可以降低智能合约运行时的开销。

Corda平台

与以太坊等平台相比，Corda平台专门用于数字货币服务，它作为一个分布式账本平台，用于保存和处理历史数字资产记录。

Stellar平台

与Corda平台类似，Stellar平台是用于数字货币应用程序的专用平台。与以太坊相比，Stellar平台更简便快捷、更易于访问。

06 智能合约的应用

- 智能合约在很多领域都有广泛的应用，包括：金融行业、保险行业、博彩行业、公共事业、供应链、政府、企业等。例如：

政府监管企业

政府的监管机构和企业之间可以签署一份智能合约，它可以在简化监管流程的同时保证企业和政府之间不会出现行贿受贿等经济问题，政府可以将调查的数据上传到智能合约中，智能合约对数据进行处理并根据结果给出企业是否合格的结论。

电力系统

智能电表可以监控用户的设备运行时间，并计算企业和个人的用电情况，电力系统可以通过智能电表读取的用电数据上传到智能合约中来产生电费单，如超过上限则触发罚款机制等，以此防止偷电等不法行为。

06 智能合约的应用

保险行业

保险行业的信任程度非常低，投保人可能在投保时为了个人利益而谎报实情。在保险公司处理理赔中的争议问题时，为了证实投保人的理赔诉求的真实性，通常需要耗费大量人力。如果将智能合约应用到保险行业，可以增强保险公司对客户真实情况的判断能力，并提高整个保险行业的信任水平。

数字货币支付

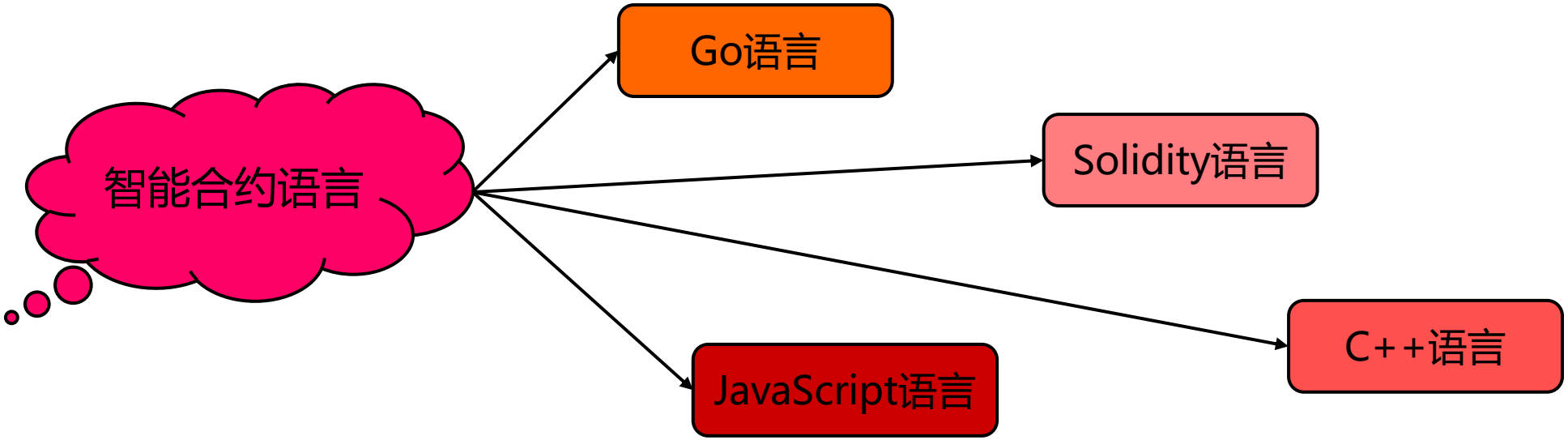
如果智能合约能够访问现有的银行系统，那么可以实现传统金融系统中无法实现的应用程序。例如：开发者可以将全球范围内各大银行中消费者的账户信息、存款信息和业务信息等所有数据整合起来，形成一个数据库，据此可以开发出方便快捷的跨境支付功能。

第 2 节

Solidity语言基础

01 Solidity语言概述

Solidity语言是在以太坊虚拟机（EVM）上运行的一种面向智能合约的高级编程语言，它是在2014年8月由Gavin Wood提出，后来这个项目被以太坊团队接手。需要注意的是，编写智能合约可以使用的语言不仅限于Solidity语言，很多其他语言同样可以用来实现智能合约。如今，Solidity语言已经成为了以太坊平台上开发智能合约的首选语言。



01 Solidity语言概述

北京大学 PEKING UNIVERSITY

- Solidity语言是一种面向对象的语言，从**特征的角度**来说它具备以下两种特性：

封装

封装是指将物理实体从真实世界中抽象出来，通常把抽象出来的物理实体称为“合约”。每个合约都是一个独立的单位，它对应一个真实世界的物理实体，这样做可以使每个物理实体包含的数据保护在合约的内部，尽可能地隐藏合约内部的细节。如果其他用户想要访问这个合约内部的数据，即使他不知道合约内部的实现细节，也可以通过合约中预留的接口发生数据交互。

继承

继承是指使用已存在的合约定义作为基础建立新合约的过程，新的合约可以继承已存在合约中的数据和功能，也可以在此基础上定义新的数据和功能。在这个过程中，通常把已存在的合约称为“父合约”、把继承父合约的新合约称为“子合约”。

01 Solidity语言概述

北京大学 PEKING UNIVERSITY

- Solidity语言是一种面向对象的语言，从**语法的角度**来说它具备以下六个特征：

静态类型语言

与C++语言相同，Solidity语言是静态类型语言，即必须将代码编译通过后才能执行、部署智能合约。

运行在服务器端

与传统的面向对象的语言不同，C++语言通常运行在本地或者服务器端，而Solidity语言是运行在以太坊上的，需要考虑用户和账户等问题，因此Solidity语言提供了地址型变量，它用于定位用户的地址。

可交易性

Solidity语言需要考虑账户支付的问题，因此它提供了payable等关键字，用于用户之间进行交易。

考虑变量的位置

与C++语言不同，Solidity语言需要考虑变量的存储位置。C++语言通常将变量保存在内存中，而Solidity语言需要区分数据的存储位置，因此Solidity语言提供了关键字memory、关键字storage等，它用于为变量指定特定的存储位置。

函数调用方式

由于Solidity语言是运行在以太坊上的语言，因此它需要考虑函数调用方式，Solidity语言提供了关键字view、关键字pure等，用于描述函数具体的调用方式，这样可以提高代码在节点中的执行效率。

异常机制

Solidity语言的异常机制与C++语言也有很大不同。在C++语言中，如果程序抛出异常，通常会执行异常检测的操作，而Solidity语言一旦出现异常，所有的执行都将会被撤回。这主要是为了保证智能合约在节点上执行时的原子性，以免节点间的数据不一致的问题。

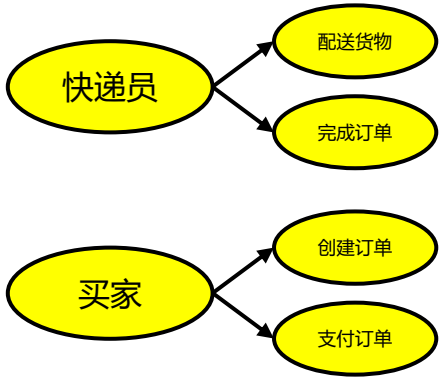
01 Solidity语言概述

在第2节的开始，提出一个智能合约的应用场景：**快递员配送用户从网络上购买的商品。**请思考：实现这样一个智能合约需要考虑哪些用户？每个用户应具备哪些功能？

北京大学 PEKING UNIVERSITY

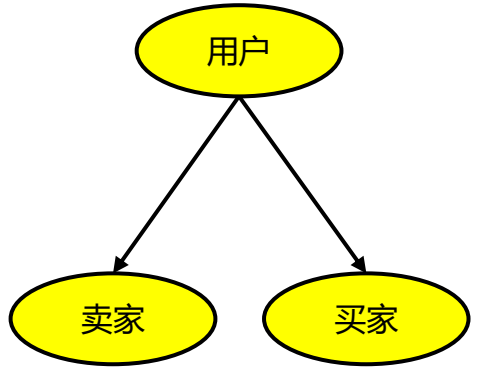
封装

封装就是把多个功能打包成一个功能的过程。例如：在“快递员配送用户从网络上购买的商品”的场景中包含以下三个物理实体有：快递员、买家和卖家，因此实现这个智能合约时我们需要为这三个物理实体设置三个合约。



继承

继承是以一个功能为基础衍生出多个功能的过程。例如：在上述场景中，由于买家和卖家都是购物平台的用户，每个用户都应该具备一些基础数据，包括：用户名、密码等。因此实现这个智能合约时可以首先实现“用户”合约的功能，然后令“买家”合约和“卖家”合约继承“用户”合约。



02 数据类型——值类型

在“快递员配送用户从网络上购买的商品”的场景中，每个合约中都应该包含描述该物理实体的数据。例如：描述快递员的数据包括姓名、性别、ID、历史订单数量、历史订单信息等。为了能够全面合理地描述一位快递员的信息，就首先需要在“快递员”合约中声明这些数据对应的数据类型和名称。“快递员”合约的声明如右图：

```
contract deliveryman
{
    string name;
    bool gender;
    address ID;
    uint orderNum;
    bytes32[] orders;
}
```

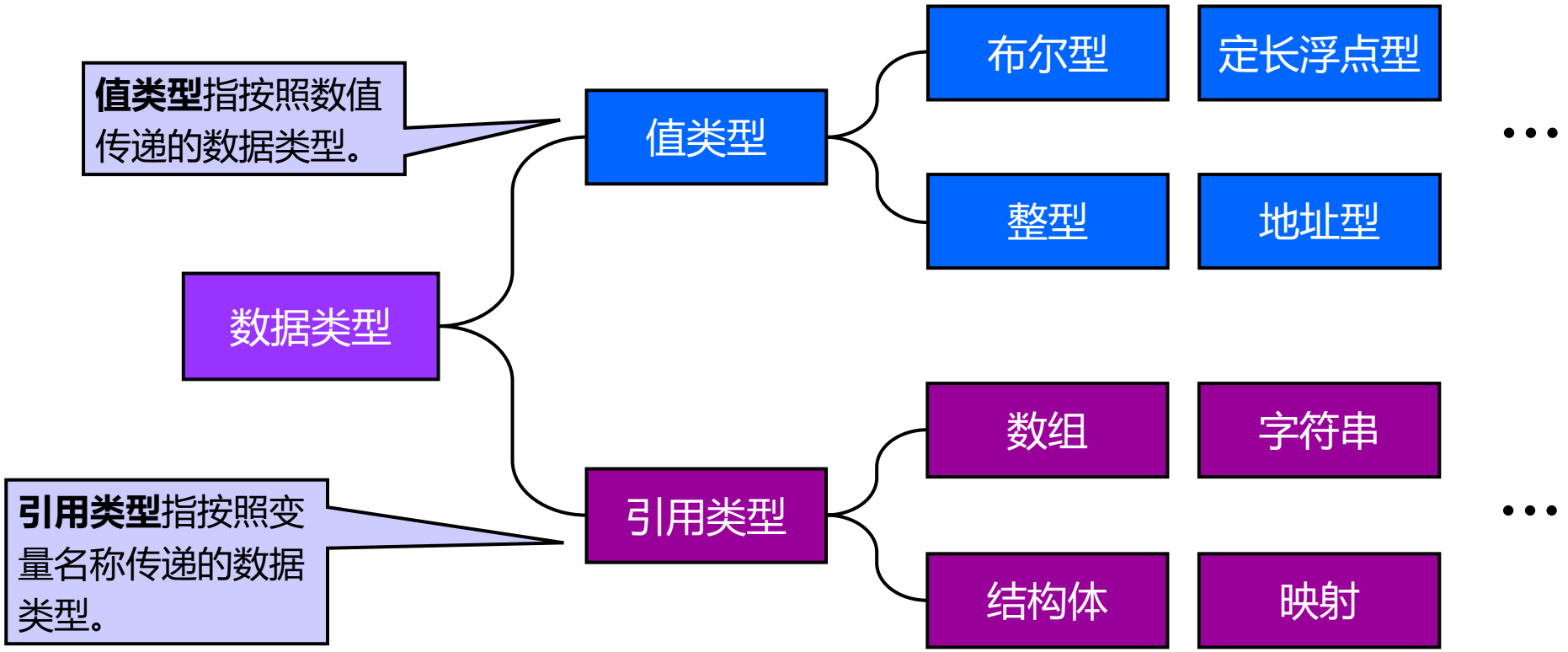
```
string name;
bool gender;
address ID;
uint orderNum;
bytes32[] orders;
```

左图是5个描述快递员信息的数据，它们都对应不同的数据类型，即计算机保存这些数据的方式不同。例如：

- 由于快递员的姓名一般是一串汉字，因此通常选用**字符串型变量**（string类型）来存储；
- 由于快递员的性别只有“男”和“女”两种可能，对于取值只有两种的变量通常选用**布尔型变量**（bool类型）来存储。

02 数据类型——值类型

- Solidity语言提供的数据类型主要可以分为两类：值类型和引用类型。其中，值类型和引用类型还包括若干种类型的变量，下面会一一讲到。



02 数据类型——值类型

- **布尔型变量**的定义和应用场景如下：

定义

名称	布尔型变量
关键字	bool
取值	它有两种取值：true和false，即数学意义上的1和0。

应用场景

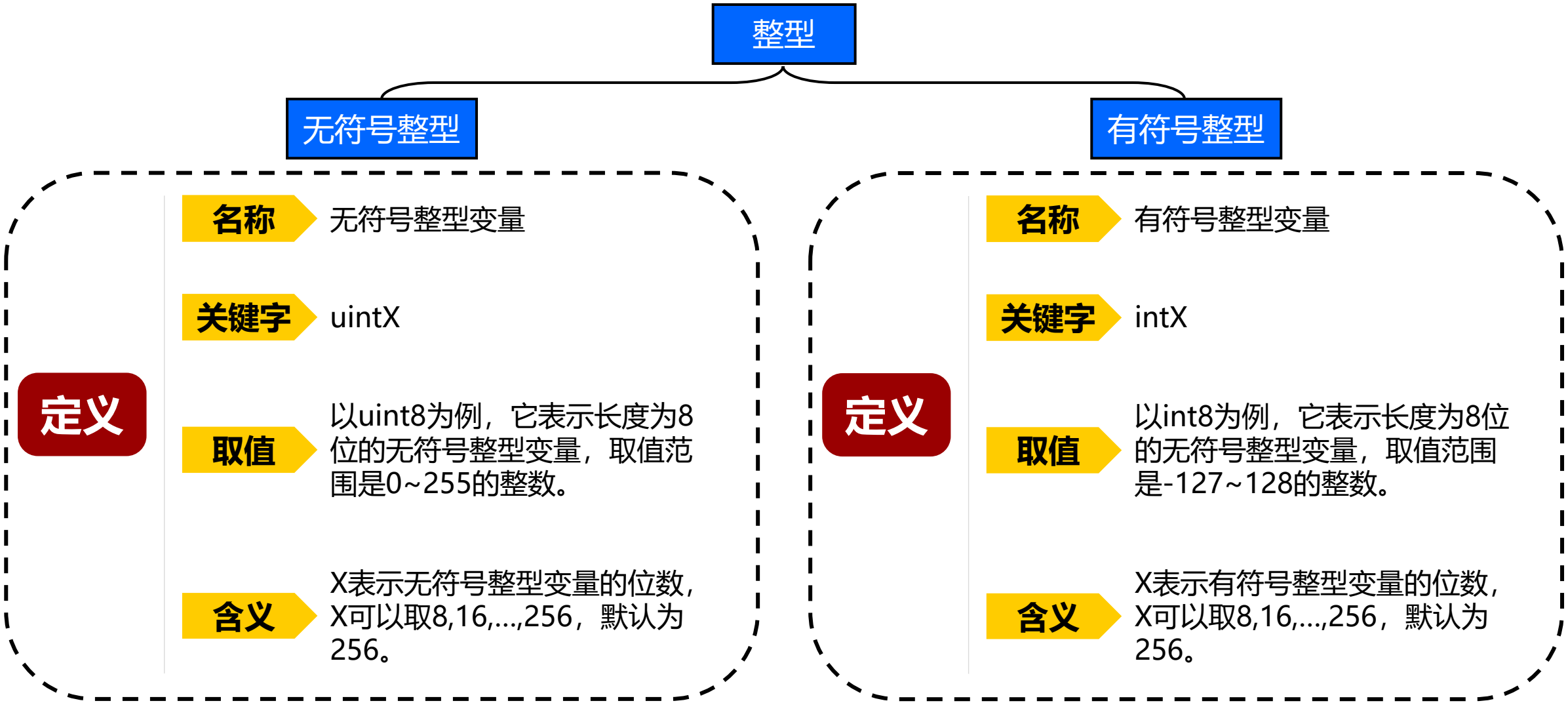
如果数据只有两种取值，那么可以使用布尔型变量来描述它。

- “快递员”合约中描述“性别”的变量就使用了布尔型变量。

```
bool gender;
```

02 数据类型——值类型

- **整型变量**包括**无符号整型变量**和**有符号整型变量**，它的定义如下：



02 数据类型——值类型

- **整型变量**的应用场景如下：

应用场景

如果数据的取值为整数，那么可以使用整型变量来描述它。

- “快递员”合约中描述 “历史订单数量”的变量就使用了整型变量。

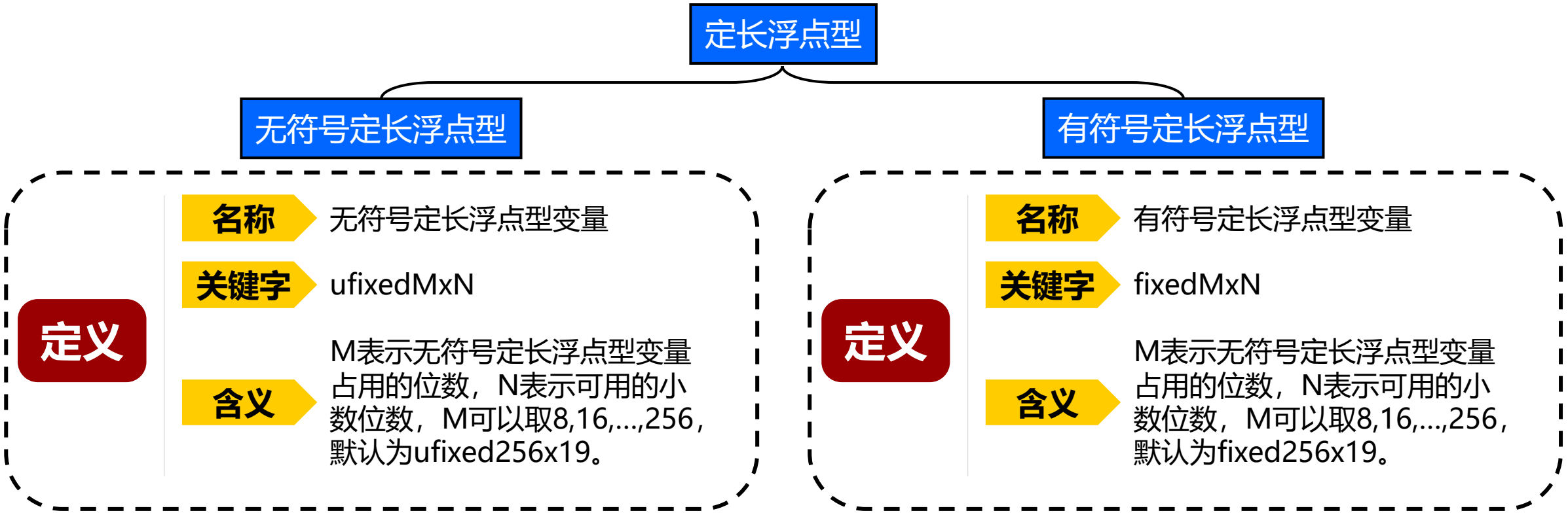
```
uint orderNum;
```

需要注意的是：

- 如果数据的取值有可能取到负数，那么可以使用有符号整型变量来描述；如果数据的取值不会取到负数，那么可以使用无符号整型变量来描述。
- 至于整型的位数，通常由数据取值的大小来判定。

02 数据类型——值类型

- **定长浮点型变量**包括**无符号定长浮点型变量**和**有符号定长浮点型变量**，它的定义如下：



由于目前还没有完全支持定长浮点型变量，因此使用定长浮点型变量的情况较少。

02 数据类型——值类型

- **地址型变量**的定义和应用场景如下：

定义

名称	地址型变量
关键字	address
取值	长度为20字节的值。

在计算机中，1个字节包含8个二进制位，即2个十六进制数。地址型变量都是由十六进制数表示的：

```
ID = 0x72ba7d8e73fe8eb666ea66babc8116a41bfb10e2;
```

在该地址中，“0x”是十六进制数的标识符，后面40个十六进制数表示用户的地址，对应20字节的长度。

应用场景

如果数据表示用户的身份，那么可以使用地址型变量来描述它。

- “快递员”合约中描述“ID”的变量就使用了地址型变量。

```
address ID;
```

02 数据类型——值类型

- **枚举型变量**的定义和应用场景如下：

定义

名称 枚举型变量

关键字 enum

取值 它的取值可以自定义。

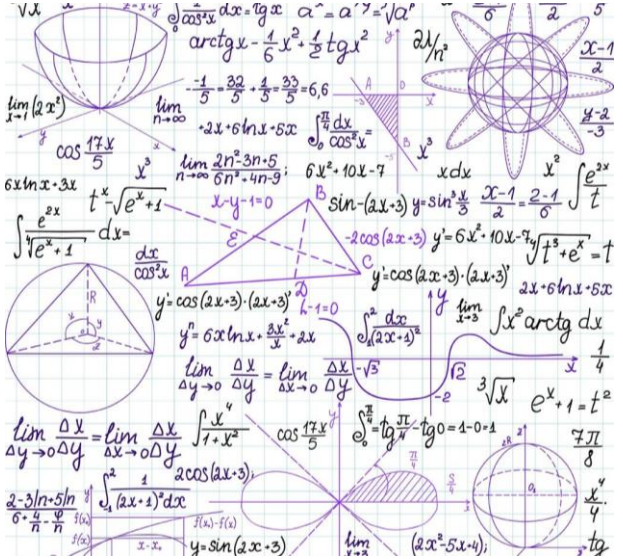
应用场景

如果数据存在多种固定的取值，那么可以使用枚举型变量来描述。

- 商品在配送过程中的状态往往有固定三种，包括“未寄出”、“运输中”和“已送达”，则可以使用枚举型变量来描述这种状态：

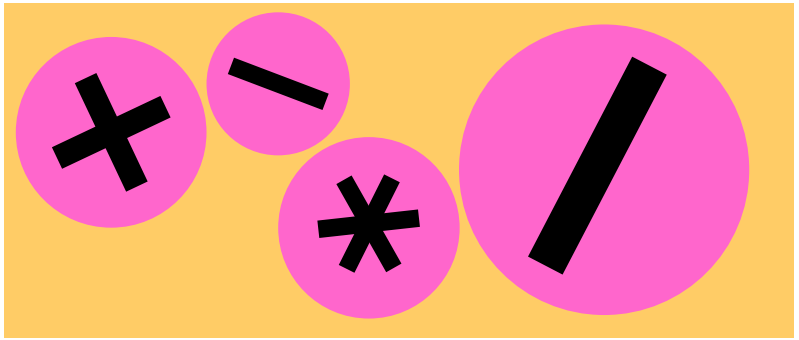
```
enum goodsState {Unsent, Transporting, Delivered} // 商品的状态。
goodsState defaultGoods;
defaultGoods = goodsState.Unsent; // 将初始商品的状态设为“未寄出”。
```

03 运算符



计算机语言通常可以使用运算符对变量进行操作或判定。

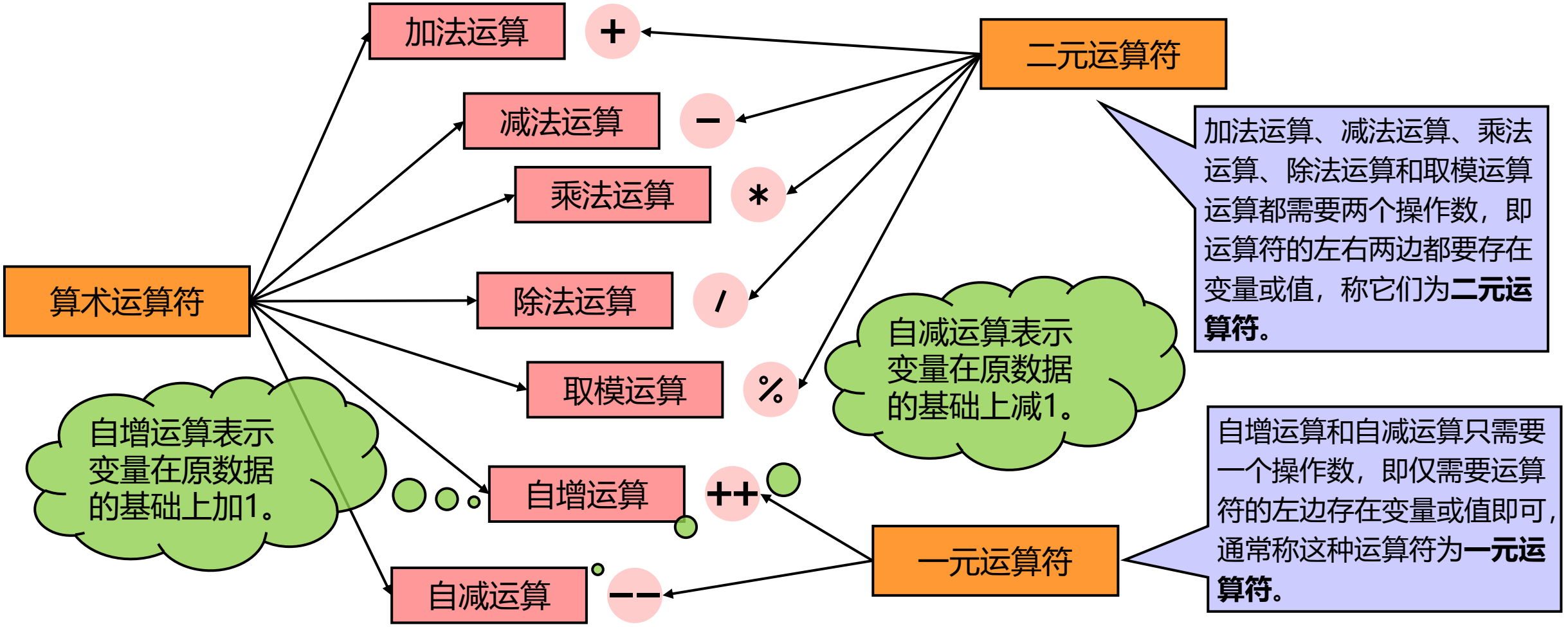
- 如果要统计买家的支出总和，就需要对其每个订单支付的费用进行累加运算，此过程就涉及了加法运算；
- 如果用户一次性购买了多项商品，在计算支出的费用时需要将涉及到乘法运算；
- 在逻辑关系的判定时，也涉及到多项逻辑运算。



在Solidity语言中，运算符可以分为六种：**算术运算符**、**比较运算符**、**赋值运算符**、**逻辑运算符**、**位运算符**和**访问运算符**。

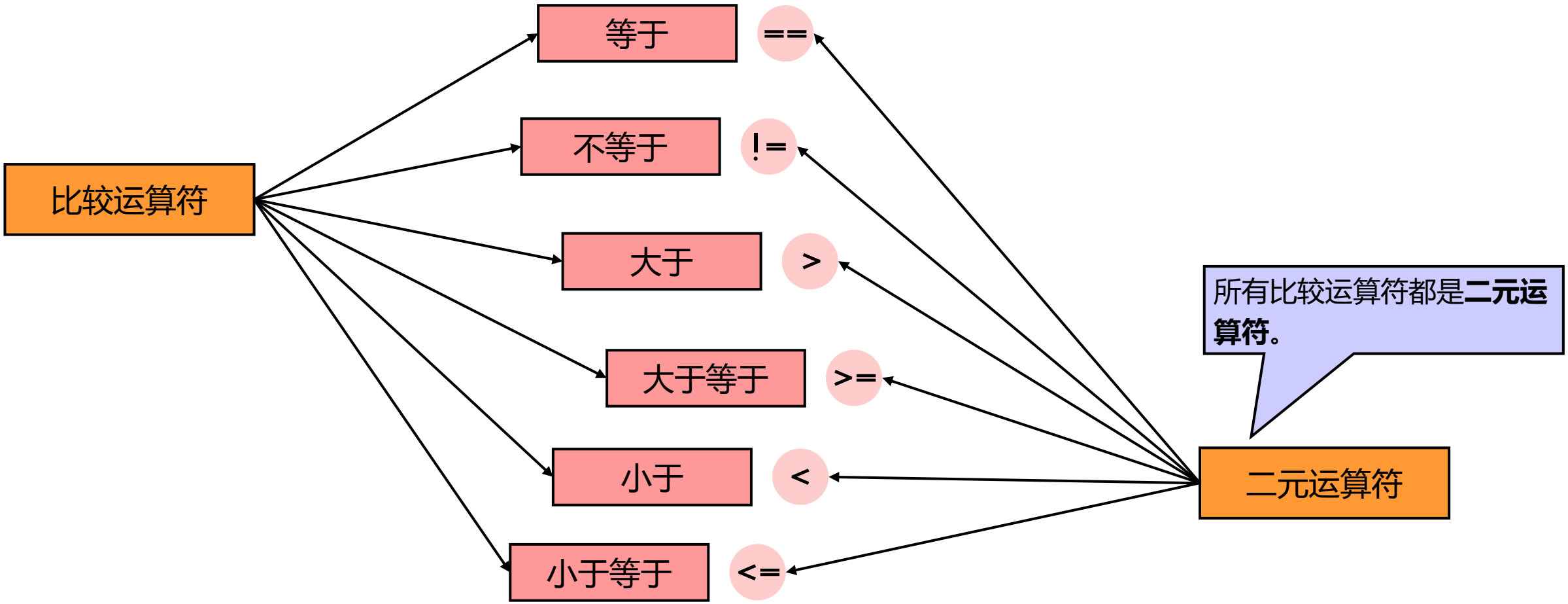
03 运算符

- **算术运算符**包括七种运算：



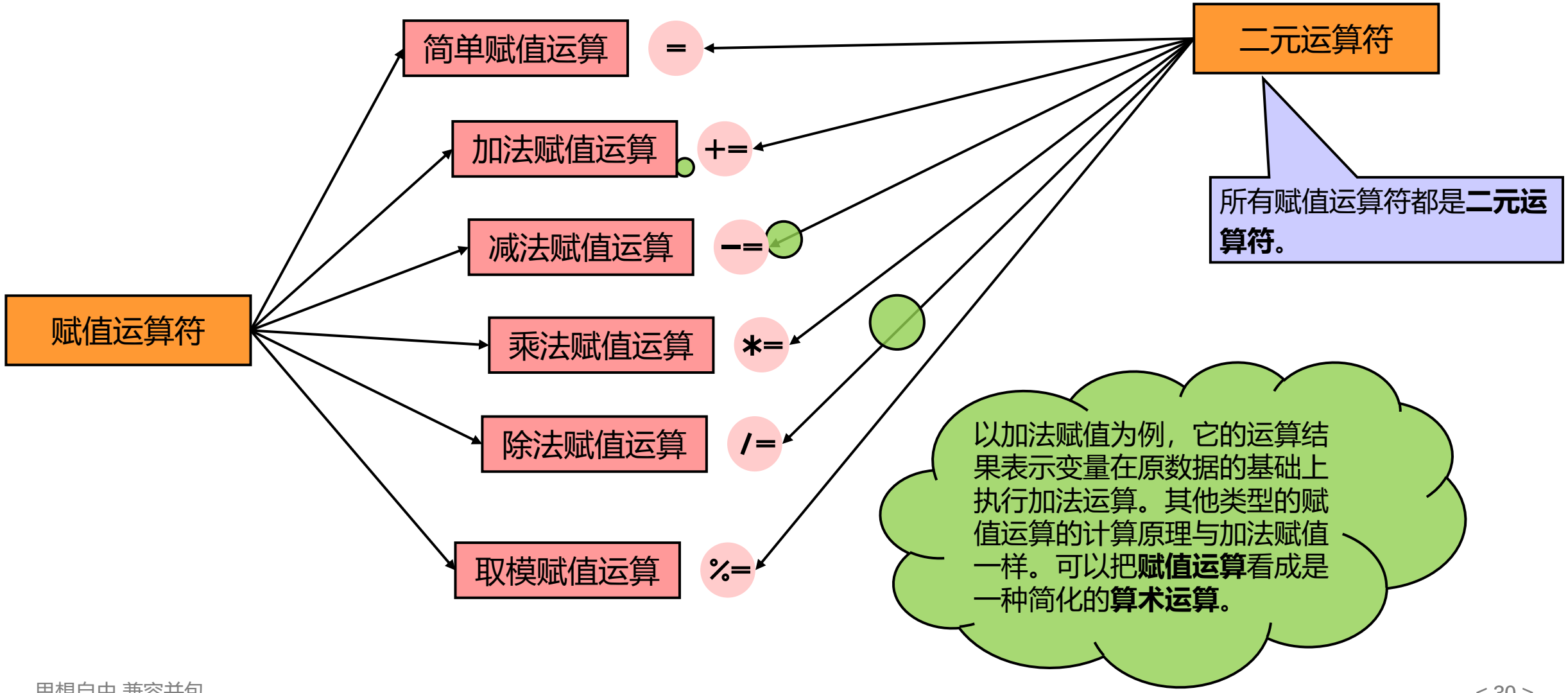
北京大学 PEKING UNIVERSITY

- **比较运算符**包括六种运算：

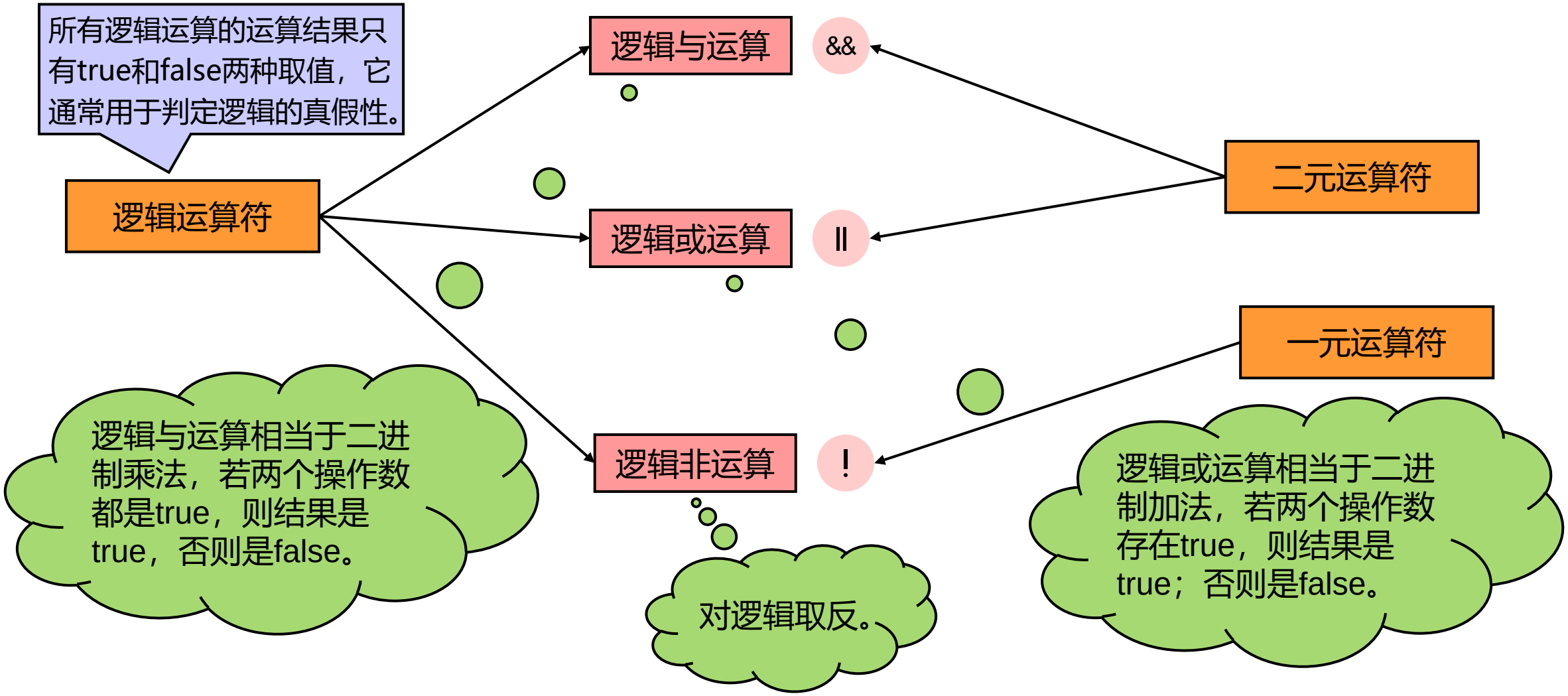


03 运算符

- **赋值运算符**包括七种运算：

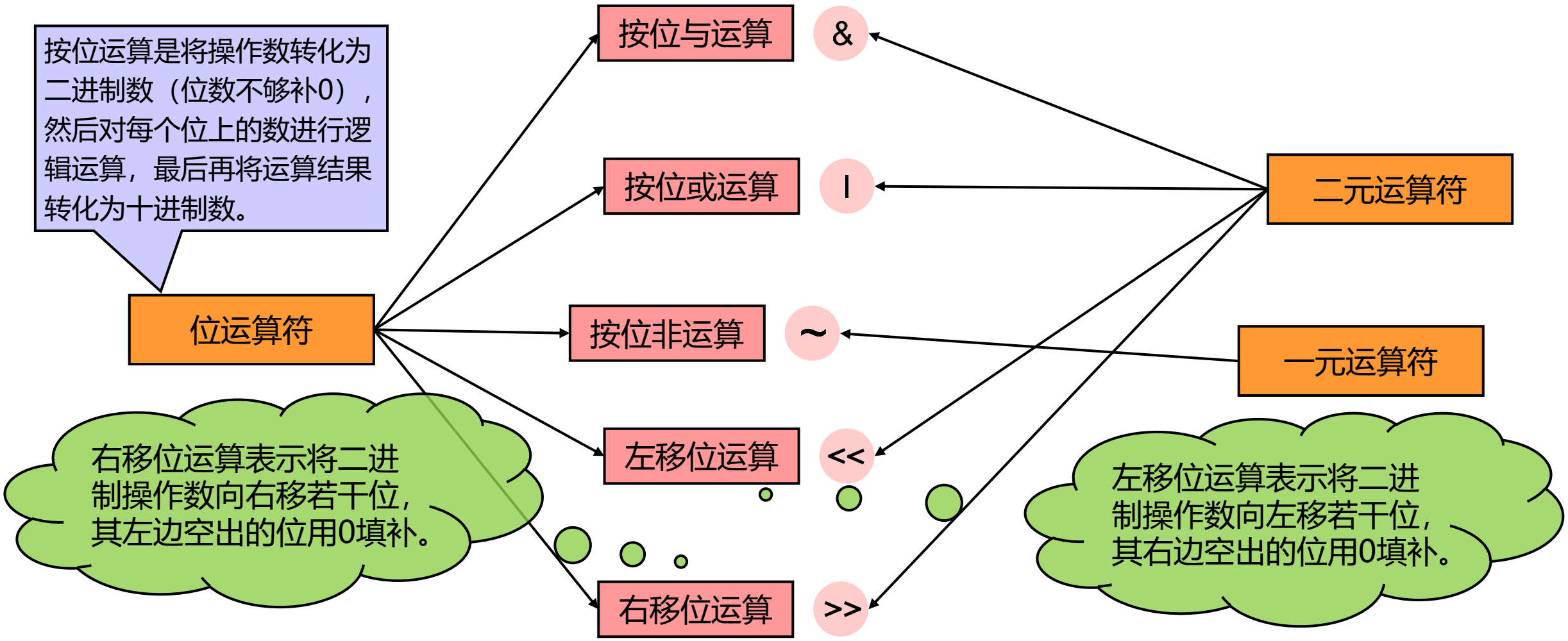


- **逻辑运算符**包括三种运算：



03 运算符

- **位运算符**包括五种运算：



04 数据类型——引用类型

- **数组**与**定长字节数组**的定义如下：

数组

…

定长字节数组

定义

名称 数组

含义 数组就像一张表格，普通变量往往只能存储单个元素，而数组可以存储若干个相同类型的元素。

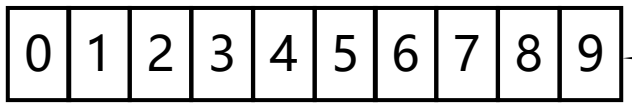
定义

名称 定长字节数组

关键字 bytesM

含义 M表示定长字节数组占用的字节数，M可以取1,2,…,32。

数组的存储方式就像酒店的一个个房间，它们占用的是连续的内存。例如，名为num整型数组就是这样存储的：

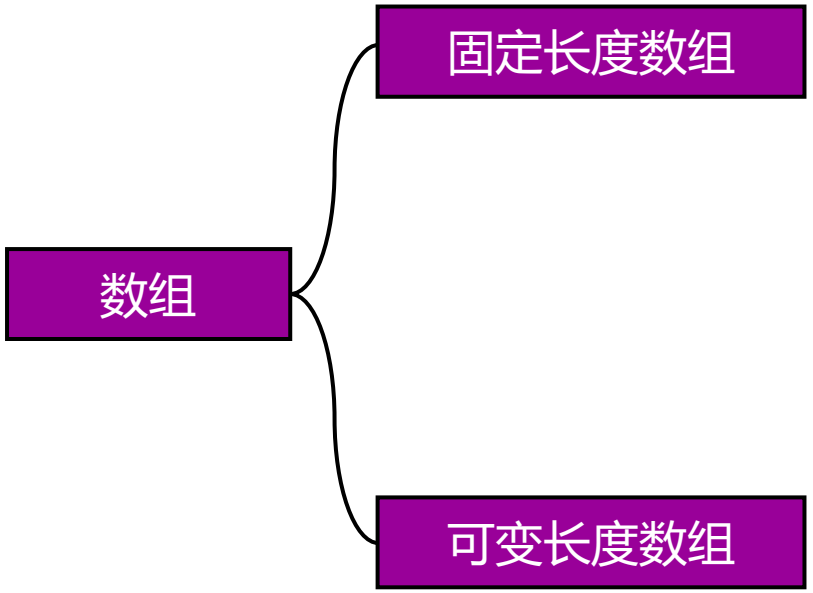


可以通过变量的位置来访问数组中的元素。例如：

```
num[4] = 4 // num数组的第四个位置里的数字是4
```

04 数据类型——引用类型

- **数组**还可以分为**固定长度数组**和**可变长度数组**：



- 在声明固定长度数组时，需要明确它保存元素的数量。对于固定长度数组，它的长度是不可更改的，因此在对固定长度数组进行赋值时，可以直接根据下标来赋值
- 在声明可变长度数组时，不需要明确它保存元素的数量。在对可变长度数组进行赋值时，为了防止数组越界的情况，通常可以使用push函数将元素添加到数组的末尾。
- 数组本身还具有成员变量length，可以通过.length获取数组的长度。可以通过更改length的值来改变可变长度数组的长度。

04 数据类型——引用类型

- **结构体变量**的定义和应用场景如下：

定义

名称 结构体变量

关键字 struct

取值 结构体通常包含多项成员变量，可以理解为它是多个变量的总称。

描述结构体的变量称作该结构体的**成员变量**，可以使用**访问运算符**来访问结构体中的变量。例如：

```
order.ID
```

称“.”为访问运算符。

应用场景

如果数据包含很多子数据，那么可以使用结构体变量来描述。

- 购买商品的订单包含很多子数据，包括：订单ID、商品名称、支付金额、下单时间、送达时间等，则可以使用结构体变量来描述订单：

```
struct order
{
    bytes32 ID; // 订单ID。
    string name; // 商品名称。
    uint price; // 支付金额。
    string orderTime; // 下单时间。
    string deliveryTime; // 送达时间。
}
```

04 数据类型——引用类型

- **字符串**的定义和应用场景如下：

定义

名称 字符串

关键字 string

取值 它的取值是一串数字、字母或汉字。

应用场景

如果数据的取值是一串字符，那么可以使用字符串来描述。

- “快递员”合约中描述“姓名”的变量就使用了字符串。

```
string name;
```

字符串只能被整体地访问，不能通过位置访问某一个字符。

04 数据类型——引用类型

- **映射**的定义和应用场景如下：

定义

名称	映射
关键字	mapping
解读	它是一种特殊的数组，数组是通过序号来访问元素的，映射可以通过关键字来访问元素。

应用场景

如果想通过数据关键字查找而不是通过数据位置查找，可以使用映射来保存数据。

- 如果想查找某个历史订单，仅凭序号查找显然是不合理的，如果能通过订单ID来查找订单会更好，那么就可以声明这样一个映射：

```
mapping(bytes32 => order) orderList;
```

在映射orderList中，它的关键字类型是bytes32类型的数据，每个bytes32类型的数据对应一个订单。

05 基础逻辑

北京大学 PEKING UNIVERSITY

- 在“快递员配送用户从网络上购买的商品”的场景中，考虑实现一种**筛选历史订单**的功能：买家想查找支出大于100元的订单。
- 想要实现这个功能，首先要访问所有的历史订单，然后通过设置条件来查找符合的订单。
- 上述过程就涉及了两种基本逻辑。

05 基础逻辑

北京大学 PEKING UNIVERSITY

- Solidity语言提供了两种基本的逻辑：**循环逻辑**和**条件逻辑**。

循环逻辑

执行逻辑

循环逻辑通过判定条件是否成立，若成立则循环执行这些代码，直到条件不满足或被打断为止。

for语句

```
for (初始条件; 条件式; 循环语句)
{
    若条件式为真，则执行此处语句，然后执行循环语句。再判定条件式是否为真，若为真则循环执行，直到条件为假。
}
```

while语句

```
while (条件式)
{
    若条件式为真，则执行此处语句。再判定条件式是否为真，若为真则再执行，直到条件为假。
}
```

条件逻辑

执行逻辑

条件逻辑通过判定条件是否成立（或不成立），分别执行后续的代码。

if语句

```
if (条件式1)
{
    若条件式1为真，则执行此处语句。
}
else if (条件式2)
{
    若条件式2为真，则执行此处语句。
}
else
{
    若上述条件均为假，则执行此处语句。
}
```

06 常见关键字

数据类型——值类型	bool	uint	int	ufixed	fixed	address	enum
数据类型——引用类型	struct	bytes	mapping	contract			
常见逻辑	if	else	else if	require	while	for	
数据的存储位置	storage	memory	calldata				
访问权限修饰符	public	private					
函数及函数修饰符	function	returns	view	pure	payable		
变量的操作	new	delete					
智能合约文件	pragma	import					

第 3 节

Solidity语言函数

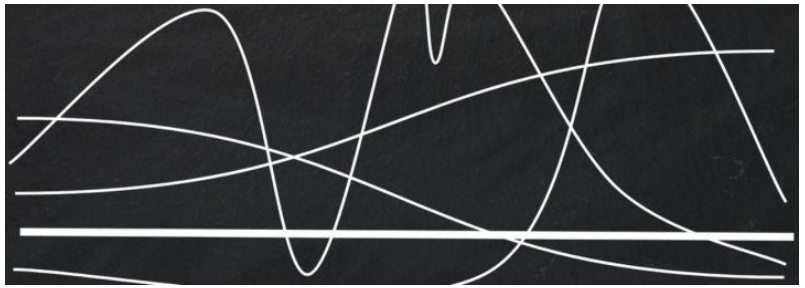
01 函数的定义

02 函数的调用

03 函数的特性

04 常见内置函数

01 函数的定义

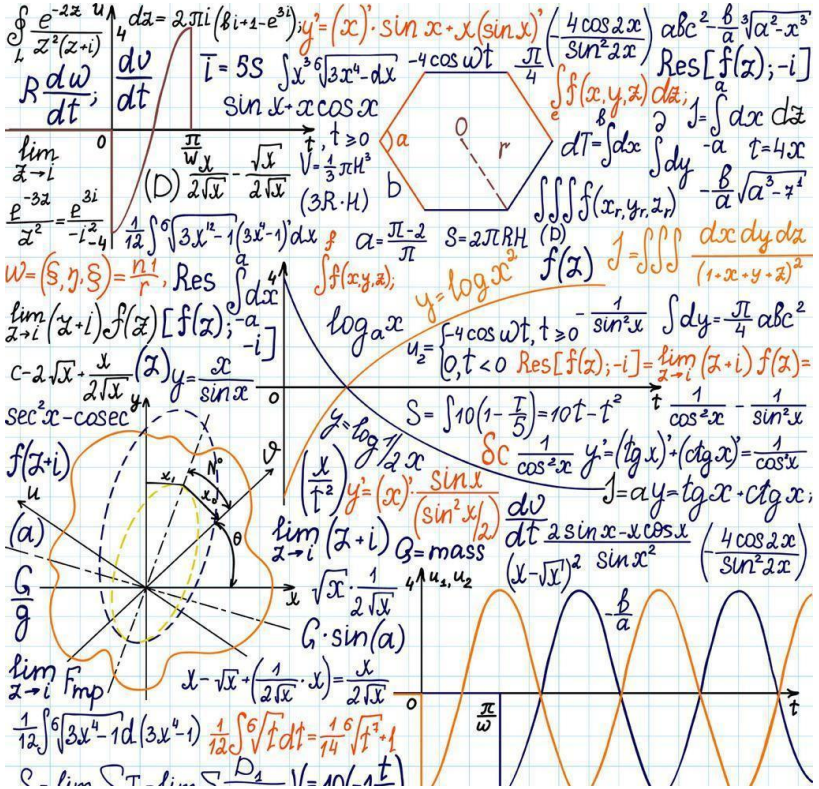


在Solidity语言中，一项基本的功能一般是以函数为单位来编写的。

计算机中“函数”的概念和数学中“函数”的概念有些类似：

- 在数学中，函数$y=f(x)$表示自变量x经过对应法则f后能够对应因变量y；
- 在计算机中，自变量x指的是输入数据，因变量y指的是输出数据，对应法则f指的是需要通过代码编写的逻辑。

通常称输入数据为**参数**、称输出数据为**返回值**。在Solidity语言中，为了防止栈溢出等情况，最多可以支持8个参数。



01 函数的定义

北京大学 PEKING UNIVERSITY

- 在“快递员配送用户从网络上购买的商品”的场景中，考虑实现一种**筛选历史订单**的功能：买家想查找支出大于一定数目的订单。
- 通过分析可以知道，这个函数的参数应该包括：价格的阈值和所有的历史订单，返回值应该是满足该条件所有订单。

01 函数的定义

- **函数**的定义过程如下：

名称 函数

关键字 function

定义

```
function 函数名称(参数1, 参数2, …) 函数特性 returns(返回值)
{
    函数逻辑
}
```

在函数getOrderByPrice中：

- **参数**：价格的阈值由整型变量_price表示、所有的历史订单由order数组_allOrder表示。
- **返回值**：order数组。
- **函数逻辑**：首先获取_allOrder的长度_numAllOrder，然后创建一个order数组_resultOrder作为预定的返回值，接着执行逻辑部分：遍历所有的历史订单，根据订单的成员变量price来筛选符合的订单，将符合的订单添加到order数组_resultOrder即可。最后返回_resultOrder。

```
function getOrderByPrice(uint _price, order[] _allOrder) returns(order[])
{
    uint _numAllOrder = _allOrder.length;
    order[] _resultOrder;
    for(uint _i = 0; _i < _numAllOrder; _i++)
    {
        if(_allOrder[i].price > _price)
        {
            _resultOrder.push(_allOrder[_i])
        }
    }
    return _resultOrder;
}
```

02 函数的调用

函数就像一个黑盒子，在调用函数的时候无需关注函数内部的实现过程，只需知道函数的参数类型和返回值类型即可。

函数的调用过程非常简单，只需要在对应位置的参数上赋予实际意义的数据即可。

- 在定义函数时，由于参数只有含义而没有具体数据，通常把定义函数中的参数称为“形式参数”；
- 在调用函数时，由于参数就有了具体的数据，通常把调用函数中的参数称为“实际参数”。

右图是调用getOrderByPrice函数的过程。

```
order[] result;
uint price = 100;
result = getOrderByPrice(price, result);
```

03 函数的特性

- “函数特性”的概念类似于名词和形容词的关系：形容词用来形容名词的一些性质，即“函数特性”用来限定函数的特征。函数的特性主要包括两个方面：**函数的可见性**和**数据的可见性。**

函数的可见性

函数的可见性主要表示该函数是否能被其他智能合约调用。

- 如果函数可以被其他智能合约调用，则使用关键字public描述它的可见性；
- 如果函数不可以被其他智能合约调用，则使用关键字private描述它的可见性。

数据的可见性

在Solidity语言中，还为函数是否修改了相关数据提供了两个关键字：pure和view。

- 关键字pure表示该函数不会与区块链的数据发生交互，即不修改也不读取区块链上保存的数据；
- 关键字view表示该函数不会更改和保存任何智能合约中的数据，即它仅仅会读取智能合约中的数据。

在初次编写代码的过程中，编译器往往会提醒你需要为函数加上关键字pure或关键字view，因此你不必担心用错了这些关键字。

04 常见内置函数

- Solidity语言提供了很多内置函数，这些函数都是系统已经为用户编写好的，用户只需要调用就可以了。本节介绍四类常见的内置函数：

数学函数

- addmod函数：计算 $(x+y)\%k$ 。

```
addmod(uint x, uint y, uint k) returns(uint)
```

- mulmod函数：计算 $(x \times y)\%k$ 。

```
mulmod(uint x, uint y, uint k) returns(uint)
```

密码学加密函数

- keccak256函数：计算某数据的SHA-3哈希值。

```
keccak256(若干个数据类型) returns(bytes32)
```

- sha256函数：计算某数据的SHA-256哈希值。

```
sha256(若干个数据类型) returns(bytes32)
```

- ripemd160函数：计算某数据的RIPEMD-160哈希

```
ripemd160(若干个数据类型) returns(bytes20)
```

04 常见内置函数

ABI编码函数

- abi.encode函数：对若干个参数进行编码。

```
abi.encode(参数1, 参数2, …) returns(bytes)
```

- abi.decode函数：对编码结果进行解码。

```
abi.decode(编码结果)
```

地址型变量成员函数

- balance变量：用户的账户余额（以wei为单位）。

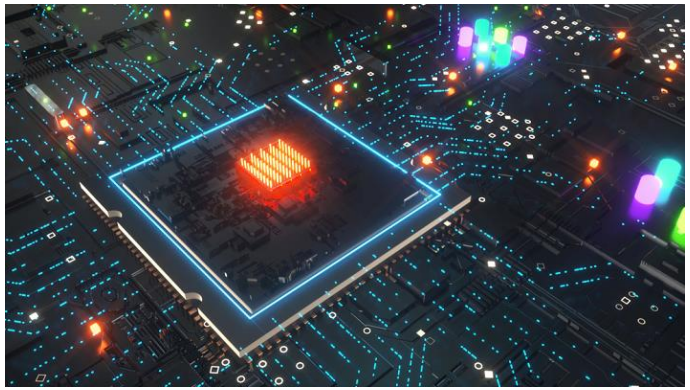
```
<address>.balance (uint256)
```

- transfer函数：向其他地址型变量发送以太币，若函数执行失败则抛出异常。

```
<address>.transfer(uint256 以太币的数量)
```

- send函数：向其他地址型变量发送以太币，若函数执行失败则返回false。

```
<address>.send(uint256 以太币的数量)
```



第 4 节

Solidity语言扩展

01 局部变量与全局变量

02 编译器与集成开发环境

03 编码习惯

04 编写智能合约的思路

01 局部变量与全局变量

北京大学 PEKING UNIVERSITY

- 在Solidity语言中（很多编程语言都是如此），智能合约中所有的变量可以分为**局部变量**和**全局变量**。

局部变量

在函数内部定义的变量称为局部变量，局部变量仅在函数内部生效，通常用关键字memory描述局部变量。

全局变量

在函数外部定义的变量称为全局变量，全安居变量在整个文件中都生效，通常用关键字storage描述全局变量。

例如，在函数getOrderByPrice内部声明的变量都属于局部变量，在该函数执行完毕后就会被销毁。而在调用函数getOrderByPrice时，变量result、变量price会一直存在，它们属于全局变量。

```
function getOrderByPrice(uint _price, order[] _allOrder) returns(order[])
{
    uint _numAllOrder = _allOrder.length;
    order[] _resultOrder;
    for(uint _i = 0; _i < _numAllOrder; _i++)
    {
        if(_allOrder[i].price > _price)
        {
            _resultOrder.push(_allOrder[_i])
        }
    }
    return _resultOrder;
}
```

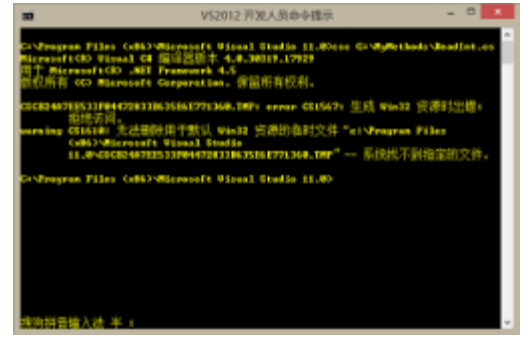
```
order[] result;
uint price = 100;
result = getOrderByPrice(price, result);
```

02 编译器与集成开发环境

- 在计算机中，编译器和集成开发环境是易混淆的两组概念，在这里需要进行说明。

编译器

我们编写的智能合约是通过高级编程语言开发的，即人类可以通过高级编程语言理解代码的内容，但是计算机是无法理解这样的代码的，这个时候就需要一类程序将高级编程语言转化为计算机可以理解的二进制代码，通常称这样的程序为**编译器**。

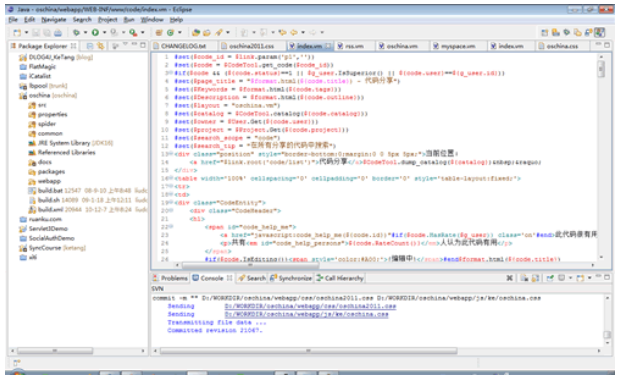


集成开发环境

在实际的开发过程中，除了编译器外，开发人员往往还需要很多其他软件来辅助开发的工作，包括：

- 代码提示器
- 调试器
- 项目管理工具

集成开发环境就是一系列开发工具的集合。可以说，集成开发环境包括了编译器等众多开发工具。



03 编码习惯

北京大学 PEKING UNIVERSITY

- 在学习编写代码的时候需要养成良好的编码习惯，这有利于代码的阅读和修改。

代码结构

- 每个缩进级别使用4个空格作为一个单位。例如：在使用大括号作为代码块时，应该保持缩进的距离相等。
- 合理使用空行。例如：在函数与函数之间、结构与结构体之间，应该中间加一行空行。
- 合理使用空格。例如：通常在运算符前后各加上1个空格，以明确该运算对应的操作数。
- 合理书写注释。例如：在每个函数开头简要注明该函数中每个参数、每个返回值的含义和数据格式。

变量命名

- 变量的命名应该遵循驼峰命名法，即混合使用大小写字母来构成变量和函数的名字，第一个单词的首字母一般小写，后面单词的首字母一般大写。
- 一些特殊的变量在命名时第一个单词的首字母也通常大写，例如：枚举型变量、结构体变量、映射变量、合约等。枚举变量的值和常量的命名往往是全部大写。函数参数和函数中局部变量的命名往往在第一个单词前加上“_”。

函数顺序

在一个智能合约中通常会存在很多函数，那么应该以合理的顺序安排这些函数，具体的顺序建议是：

- 构造函数
- 外部函数
- 公共函数
- 内部函数

04 编写智能合约的思路

北京大学 PEKING UNIVERSITY

- 在实际编写智能合约时，主要有两种思路：**从局部到整体**和**从整体到局部。**

从局部到整体

- **从局部到整体**是指先编写智能合约要实现的函数，然后根据每个函数的需求声明全局变量或编写辅助函数。
- 这种方法可以应用到编写规模较小、功能较为简单的智能合约上，它缺乏对智能合约的合理规划和统筹布局，很像“拍脑袋”编写出来的程序，因此十分不推荐这种编写思路。

从整体到局部

- **从整体到局部**是指先规划好智能合约的全局变量和功能函数，然后按照构造函数、外部函数、公共函数、内部函数的顺序编写智能合约。这种方法可以应用到任意规模智能合约上，可以通过有计划地布局吧大型智能合约拆分成若干个子模块交给不同的开发人员编写。
- 在编写函数时一定要明确调用函数的主体，即哪类用户有可能调用这个函数，可以根据调用实体的变化调整内部函数的顺序，这有利于其他开发人员阅读并修改代码。因此非常推荐这种编写思路。

第5节

区块链中的智能合约

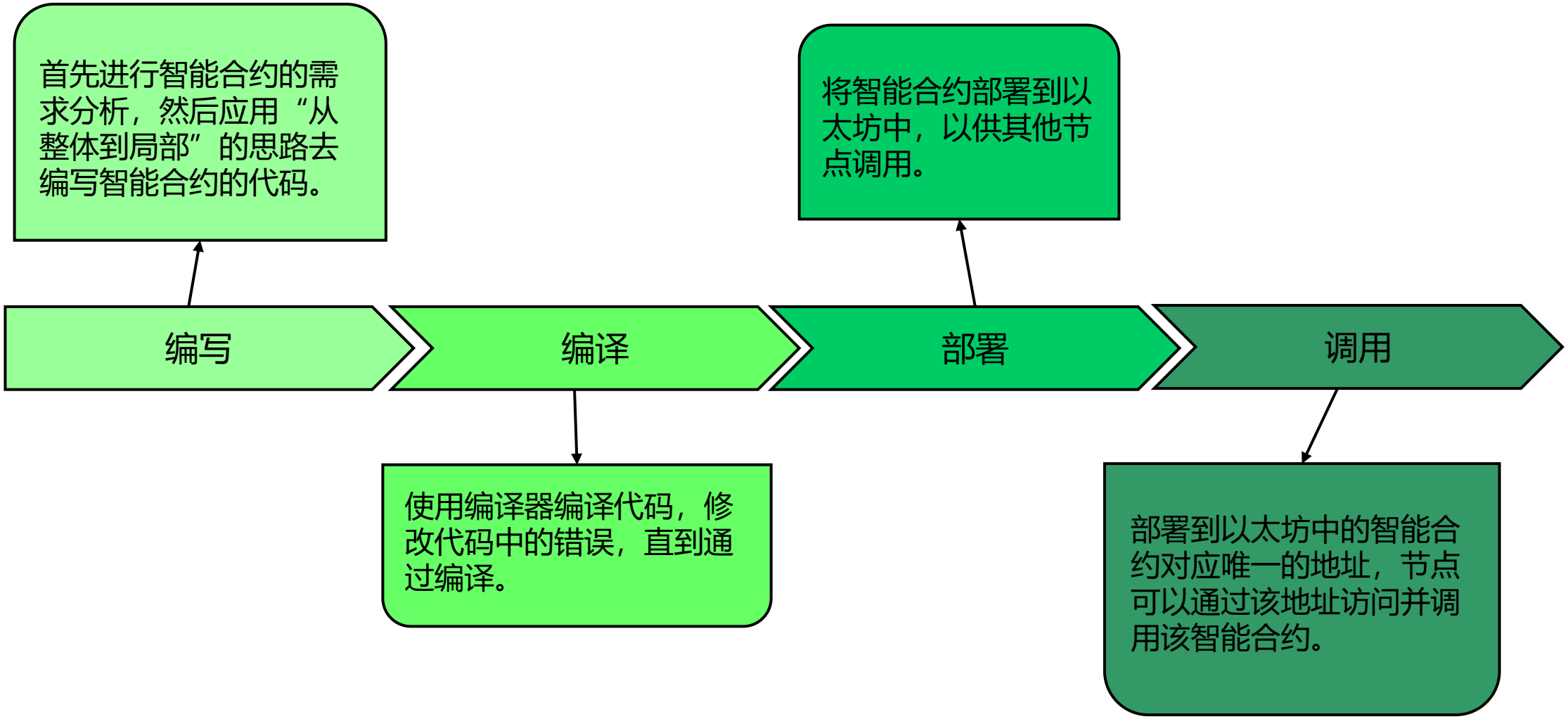
01 发布智能合约的过程

02 智能合约与区块链的交互

03 Remix开发平台

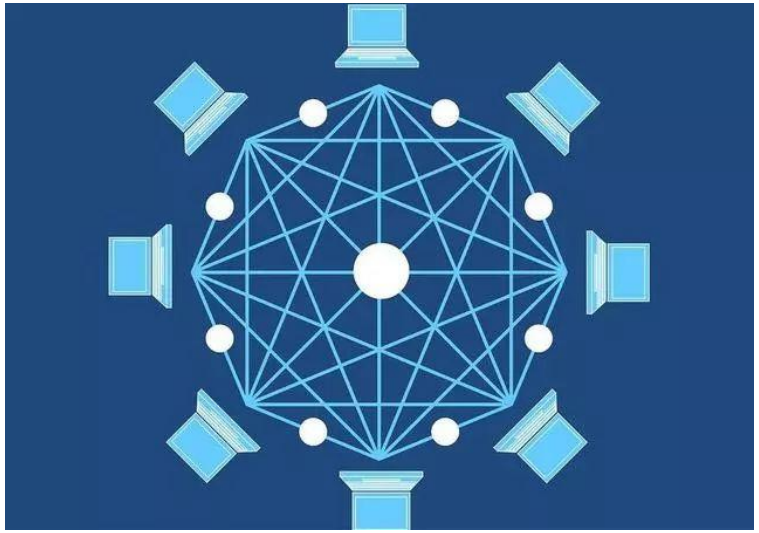
04 智能合约实例

01 发布智能合约的过程



02 智能合约与区块链的交互

当智能合约被发布到以太坊中以后，以太坊网络之间的所有节点会通过共识保存这份智能合约的代码。当节点需要调用智能合约中的某个函数时，该节点首先需要支付调用该函数的费用，然后此次调用及其支付的费用会作为一笔交易打包到以太坊中的某区块上。



一旦以太坊网络之间的所有节点对该区块形成共识，就意味着节点对该函数的调用是合法的，那么以太坊就会执行该函数，其产生的数据也会保存在以太坊中，这个过程称为“数据上链”。由于区块的生成遵循PoW共识机制，因此其他节点只能访问链上的数据而不能修改链上的数据。

02 智能合约与区块链的交互

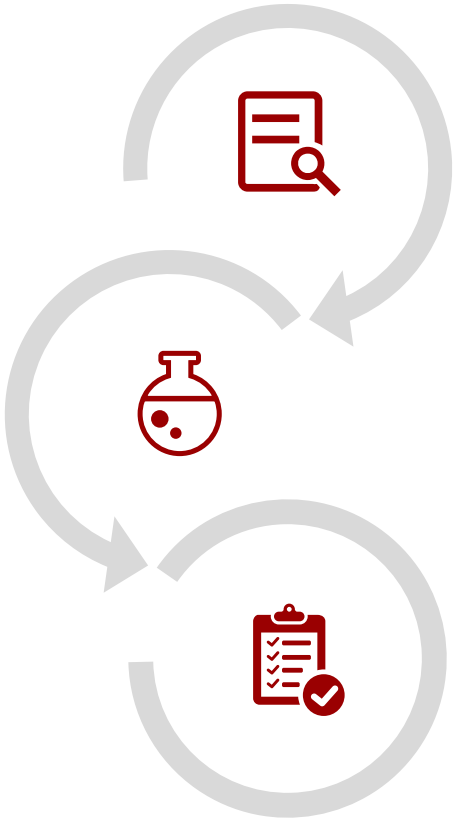
- 通常情况下，以太坊上的交易有以下三种：普通交易、只有发送方的交易、将以太币转移到智能合约的交易。

普通交易

例如：两个用户之间发生了以太币的转移，即一个用户将以太币支付到了另一个用户中，这类交易属于普通交易。

只有发送方的交易

如果一笔交易中只有发送方而没有接收方，这意味着此次交易是用于在以太坊中中创建一个智能合约。

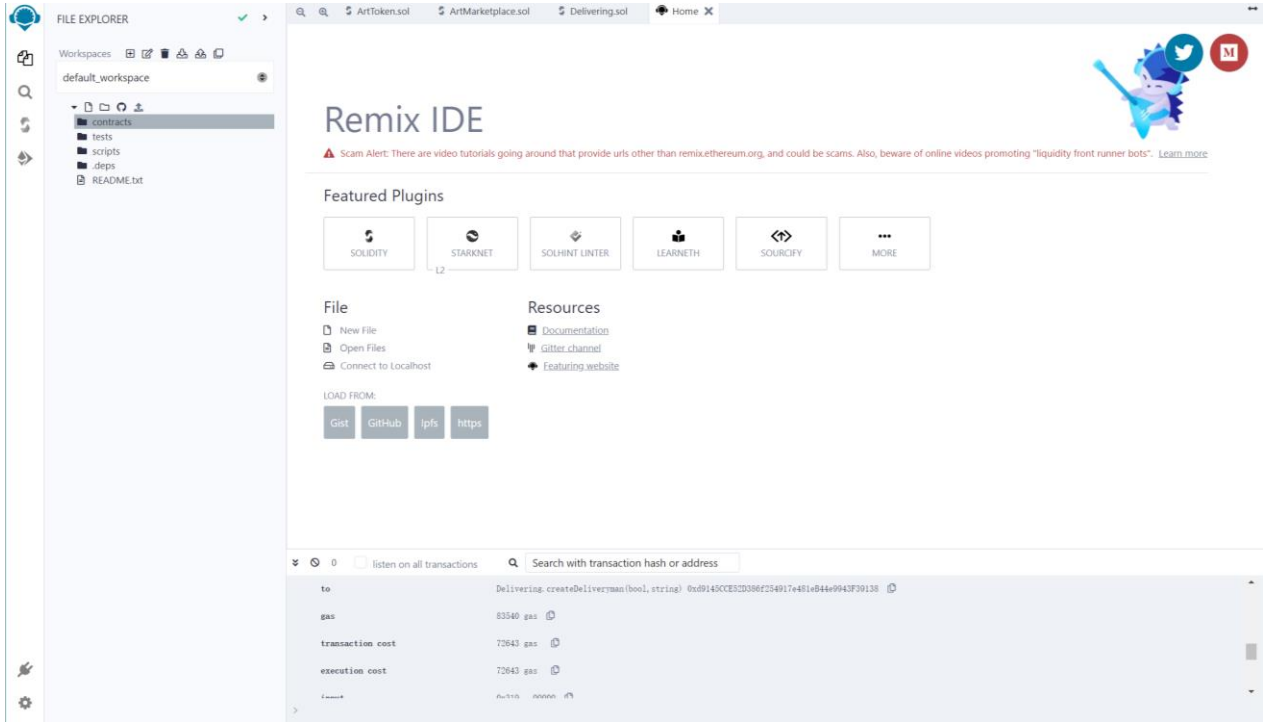


将以太币转移到智能合约的交易

如果一笔交易中接收方的地址是一份智能合约，这意味着此次交易是发送方调用该智能合约时产生的交易。

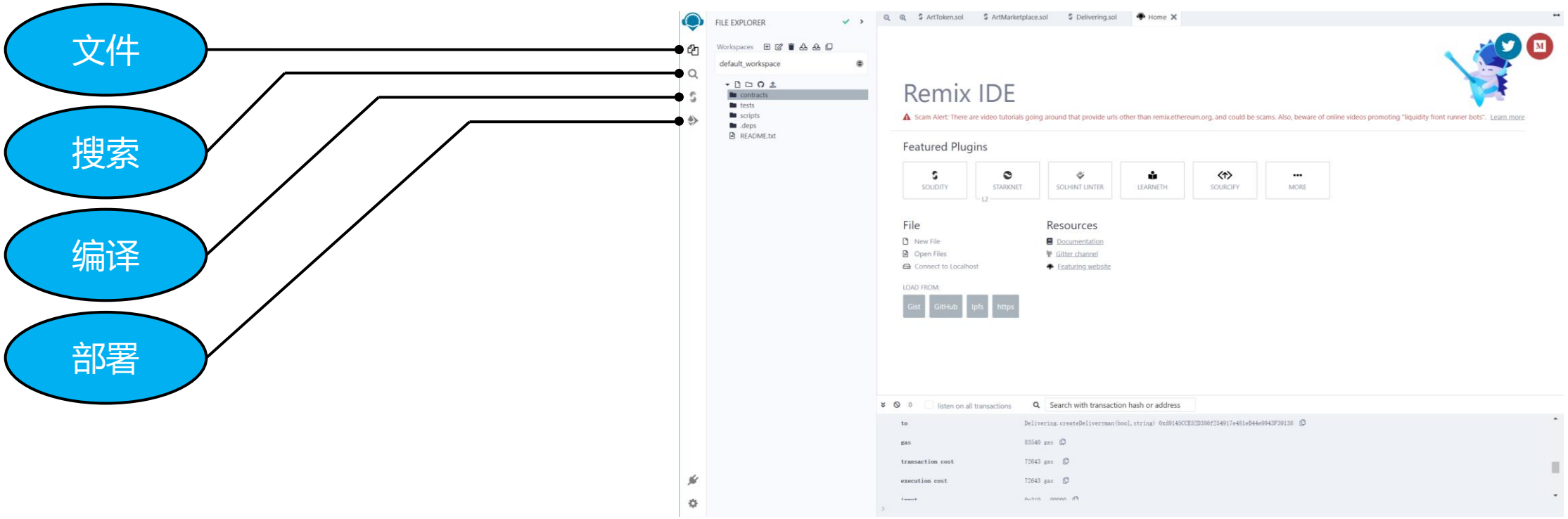
03 Remix开发平台

- Remix开发平台是一个在线的智能合约开发平台，它提供基本的编译、部署至本地或测试网络、执行合约等功能。其网址是：http://remix.ethereum.org/。



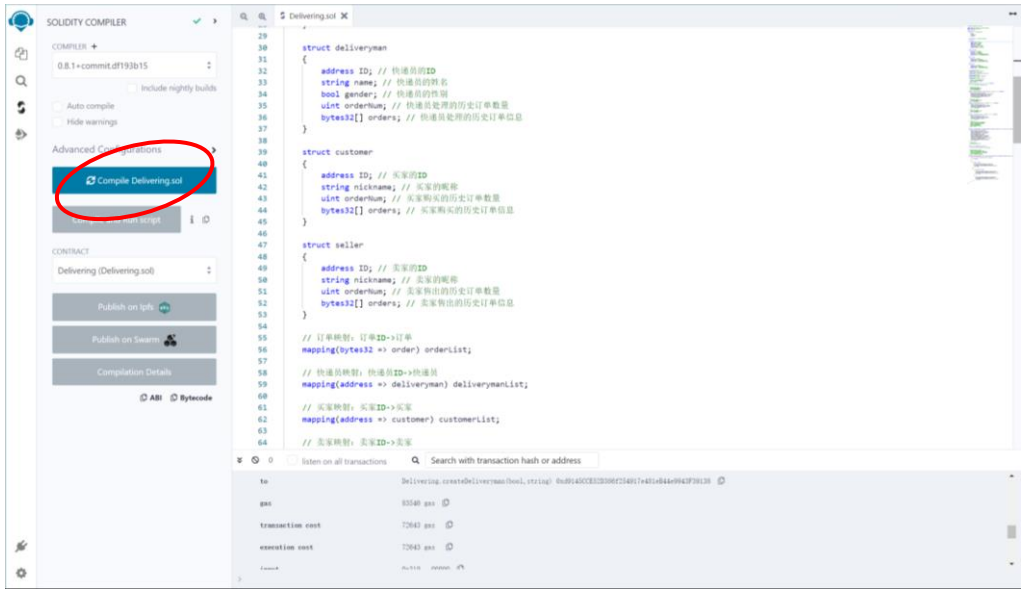
03 Remix开发平台

- Remix开发平台是一个在线的智能合约开发平台，它提供基本的编译、部署至本地或测试网络、执行合约等功能。其网址是：http://remix.ethereum.org/。
- 我们可以看到，Remix开发平台左侧有四个导航栏，第一个是“文件”导航栏、第二个是“搜索”导航栏、第三个是“编译”导航栏、第四个是“部署”导航栏。

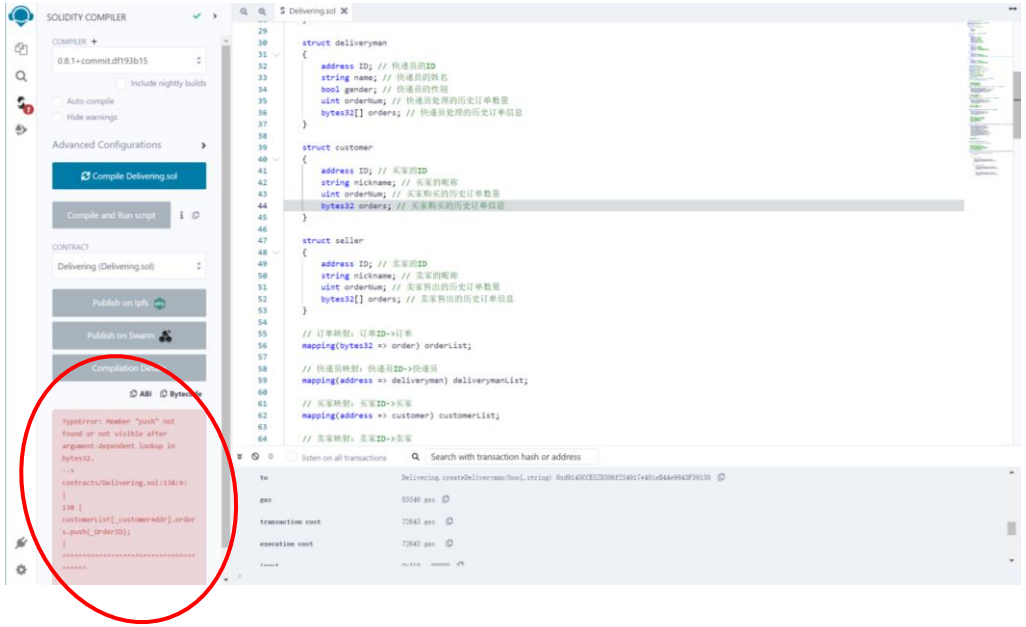


03 Remix开发平台

- 以“配送商品”合约Delivering.sol为例，可以先把该合约的代码编写上传到开发平台中，点击左侧导航栏的第三个按钮，即可进入“编译”菜单，点击“Complie Delivering.sol” 按钮即可编译合约。

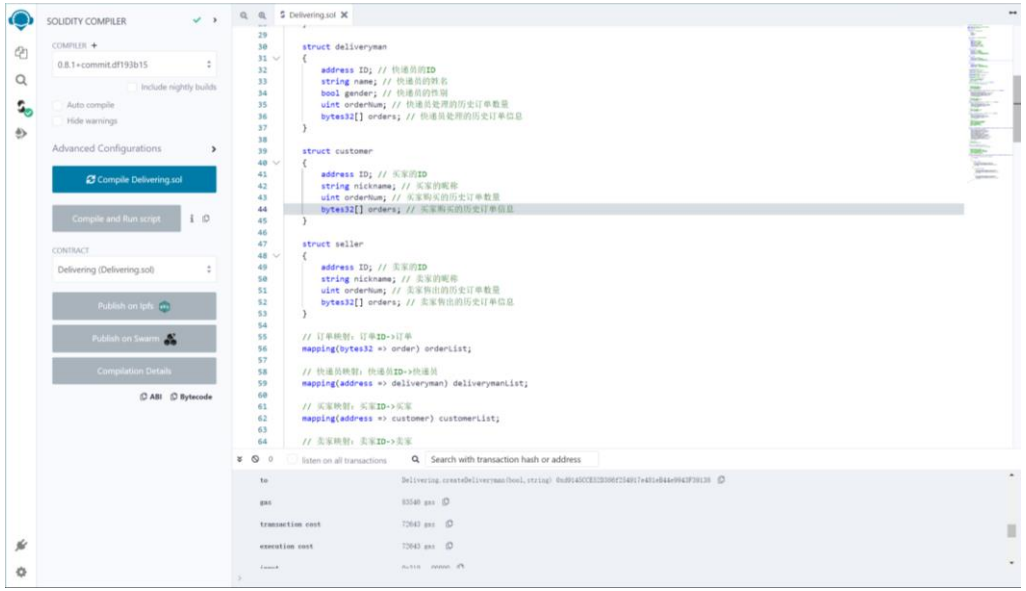


- 若合约中的代码有警告或错误，Remix 开发平台会在下方显示各种警告信息或错误信息。警告信息是允许存在的，但是它不耽误正常部署；如果存在错误信息，那么您需要找出错误之处并将其修改正确。

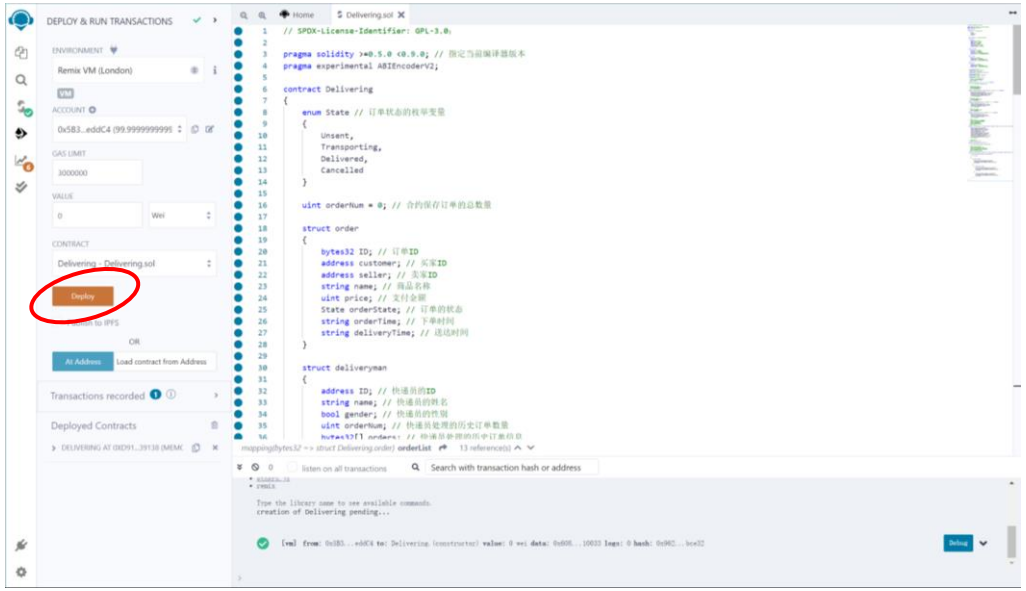


03 Remix开发平台

- 若合约中的代码没有警告或错误，这说明它可以通过编译，下面是通过编译的显示情况，此时您可以进行后续的部署和测试等操作。

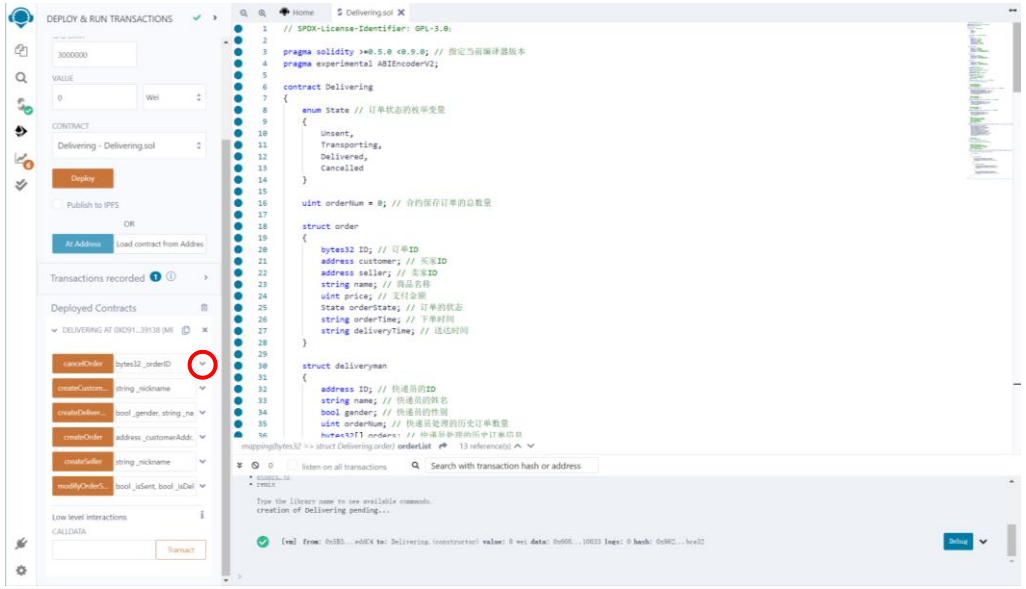


- 点击左侧导航栏的第四个按钮，即可进入“部署”菜单，点击“Deploy”按钮，即可将刚才编译好的智能合约部署到测试网络中。部署完成后，您可以在右下方的交互界面看到合约的状态。



03 Remix开发平台

- 点击左下角“Deployed Contracts”旁边的小箭头，你可以看到在测试网中可以执行的函数，此时你可以测试函数的功能。



- 通过以上的介绍，您一定可以完成相关的操作，本课程将会以实验的形式将编写智能合约的任务布置下去。

04 智能合约实例

- 将以“快递员配送用户从网络上购买的商品”为例向您展示这些知识是如何有机地串联起来以形成一个完整的智能合约的。首先展示智能合约的实现代码，然后分析每个部分的具体意义与涉及的知识点。

1. 首先声明编译器的版本、导入相关的库文件或外部智能合约。

```
// SPDX-License-Identifier: GPL-3.0;

pragma solidity >=0.5.0 <0.9.0; // 指定当前编译器版本
pragma experimental ABIEncoderV2;
```

2. 声明若干个全局变量。

```
enum State // 订单状态的枚举变量
{
    Unsent,
    Transporting,
    Delivered,
    Cancelled
}

uint orderNum = 0; // 合约保存订单的总数量
```

北京大学 PEKING UNIVERSITY

3. 声明订单结构体、快递员结构体、买家结构体和卖家结构体。

```
struct order
{
    bytes32 ID; // 订单ID
    address customer; // 买家ID
    address seller; // 卖家ID
    string name; // 商品名称
    uint price; // 支付金额
    State orderState; // 订单的状态
    string orderTime; // 下单时间
    string deliveryTime; // 送达时间
}
```

```
struct deliveryman
{
    address ID; // 快递员的ID
    string name; // 快递员的姓名
    bool gender; // 快递员的性别
    uint orderNum; // 快递员处理的历史订单数量
    bytes32[] orders; // 快递员处理的历史订单信息
}
```

```
struct customer
{
    address ID; // 买家的ID
    string nickname; // 买家的昵称
    uint orderNum; // 买家购买的历史订单数量
    bytes32[] orders; // 买家购买的历史订单信息
}
```

```
struct seller
{
    address ID; // 卖家的ID
    string nickname; // 卖家的昵称
    uint orderNum; // 卖家售出的历史订单数量
    bytes32[] orders; // 卖家售出的历史订单信息
}
```

04 智能合约实例

4. 声明了四个映射，以供用户通过ID访问到相关信息。

```
// 订单映射：订单ID->订单
mapping(bytes32 => order) orderList;

// 快递员映射：快递员ID->快递员
mapping(address => deliveryman) deliverymanList;

// 买家映射：买家ID->买家
mapping(address => customer) customerList;

// 卖家映射：卖家ID->卖家
mapping(address => seller) sellerList;
```

5. 编写私有函数getOrderID，该函数的功能是利用合约中订单的总数量生成订单ID。

```
/* 利用合约中订单的总数量生成订单ID
 * 参数1：_orderNum表示订单的序号
 * 返回值：订单ID
 */
function getOrderID(uint _num) private pure returns(bytes32)
{
    return keccak256(abi.encodePacked(_num));
}
```

北京大学 PEKING UNIVERSITY

6. 编写三个注册函数：函数createDeliveryman、函数createCustomer和函数createSeller。

```
    /* 函数1：快递员注册账户
     * 参数1：_gender表示快递员性别
     * 参数2：_name表示快递员姓名
     * 返回值：快递员账户地址
    */
    function createDeliveryman(bool _gender, string memory _name) public
returns(address)
    {
        address _deliverymanAddr = msg.sender;
        deliverymanList[_deliverymanAddr].ID = _deliverymanAddr;
        deliverymanList[_deliverymanAddr].name = _name;
        deliverymanList[_deliverymanAddr].gender = _gender;
        deliverymanList[_deliverymanAddr].orderNum = 0;
        return _deliverymanAddr;
    }
```

```
    /* 函数2：买家注册账户
     * 参数1：_nickname表示买家昵称
     * 返回值：买家账户地址
     */
    function createCustomer(string memory _nickname) public
returns(address)
    {
        address _customerAddr = msg.sender;
        customerList[_customerAddr].ID = _customerAddr;
        customerList[_customerAddr].nickname = _nickname;
        customerList[_customerAddr].orderNum = 0;
        return _customerAddr;
    }
```

```
/* 函数3：卖家注册账户
 * 参数1：_nickname表示卖家昵称
 * 返回值：卖家账户地址
 */
function createSeller(string memory _nickname) public returns(address)
{
    address _sellerAddr = msg.sender;
    sellerList[_sellerAddr].ID = _sellerAddr;
    sellerList[_sellerAddr].nickname = _nickname;
    sellerList[_sellerAddr].orderNum = 0;
    return _sellerAddr;
}
```

04 智能合约实例

北京大学 PEKING UNIVERSITY

7. 编写三个订单操作函数：函数createOrder、函数cancelOrder和函数modifyOrderState。

```
    /* 函数4：创建订单
     * 参数1：_customerAddr表示买家账户地址
     * 参数2：_sellerAddr表示卖家账户地址
     * 参数3：_name表示商品名称
     * 参数4：_price表示支付金额
     * 参数5：_orderTime表示下单时间
     * 返回值：订单ID
     */
    function createOrder(address _customerAddr, address _sellerAddr, string
memory _name, uint _price, string memory _orderTime) public
returns(bytes32)
    {
        bytes32 _OrderID = getOrderID(orderNum);
        orderList[_OrderID].ID = _OrderID;
        orderList[_OrderID].customer = _customerAddr;
        orderList[_OrderID].seller = _sellerAddr;
        orderList[_OrderID].name = _name;
        orderList[_OrderID].price = _price;
        orderList[_OrderID].orderState = State.Unsent;
        orderList[_OrderID].orderTime = _orderTime;
        customerList[_customerAddr].orderNum += 1;
        customerList[_customerAddr].orders.push(_OrderID);
        sellerList[_sellerAddr].orderNum += 1;
        sellerList[_sellerAddr].orders.push(_OrderID);
        return _OrderID;
    }
```

```
   /* 函数5：取消订单
    * 参数1：_orderID表示订单ID
    */
   function cancelOrder(bytes32 _orderID) public
   {
       orderList[_orderID].orderState = State.Cancelled;
   }
```

```
   /* 函数6：快递员修改订单的状态
    * 参数1：_isSent表示快递是否开始运输
    * 参数2：_isDelivered表示快递是否送达至用户
    * 参数3：_deliverymanAddr表示快递员账户地址
    * 参数4：_orderID表示订单ID
    * 返回值：订单的状态是否修改成功
    */
   function modifyOrderState(bool _isSent, bool _isDelivered, address
deliverymanAddr, bytes32 _orderID) public returns(bool)
   {
       if(orderList[_orderID].orderState == State.Cancelled)
       {
           return false;
       }
       else
       {
           if(_isSent == false)
           {
               orderList[_orderID].orderState = State.Unsent;
               deliverymanList[_deliverymanAddr].orderNum += 1;
               deliverymanList[_deliverymanAddr].orders.push(_orderID);
               return true;
           }
           else
           {
               if(_isDelivered == false)
               {
                   orderList[_orderID].orderState = State.Transporting;
                   deliverymanList[_deliverymanAddr].orderNum += 1;

deliverymanList[_deliverymanAddr].orders.push(_orderID);
                   return true;
               }
               else
               {
                   orderList[_orderID].orderState = State.Delivered;
                   deliverymanList[_deliverymanAddr].orderNum += 1;

deliverymanList[_deliverymanAddr].orders.push(_orderID);
                   return true;
               }
           }
       }
   }
```

本章小结

本章主要围绕智能合约展开。

- 第1节从智能合约的概念、功能、发展历史、技术挑战、开发平台和应用等角度简要介绍了智能合约，然后提出了“快递员配送用户从网络上购买的商品”的应用场景。
- 第2节简要介绍了Solidity语言中基本概念、数据类型、运算符、基础逻辑、常见关键字。
- 第3节初步介绍了Solidity语言中函数的概念和应用。
- 第4节介绍了Solidity语言中一些其他概念，包括局部变量与全局变量、编译器与集成开发环境、编码习惯和编写智能合约的思路。
- 第5节介绍了智能合约与区块链的交互，以及在区块链上部署智能合约的过程，最后举了一个智能合约的实例。

北京大学
PEKING UNIVERSITY

感谢您的观看