

类、类型、抽象数据类型 异同

- 都是 (A, Op) 值集和值上的操作集，在这个意义上，它们通用
- 类型和抽象数据类型都是类型。类型沿用历史理解是基元类型 + 简单结构类型（预定义），抽象数据类型是用户可以用基本类型构造复杂类型（用户定义）
- 类和抽象数据类型都是在类型体系结构上定义操作集，类是用用户定义的程序对象的概括抽象。程序对象是封装的自主的，主动实施类中定义的操作，加工私有数据。抽象数据类型只说明程序对象的类型，被动地接受该类型允许的操作。

<对象名>.<操作名> ADT:V 抽象数据类型的变量（名）

C:V 类的实例对象（名）

<操作名>.(<变量名>) OP1 (V) V是该类型值的引用

V.OP1 V是实例对象名，不涉及值

- 类/类型和子类/子类型形差异更大
子类是类的特化，内涵增加
子类型是类型的子集内涵所小，外延都减小

引用对象

- 在不引起歧义情况下，有时也可称“对象引用”为“某某对象”

- Array a;
- if(a[i] == 0)...
- a[i] = 1;

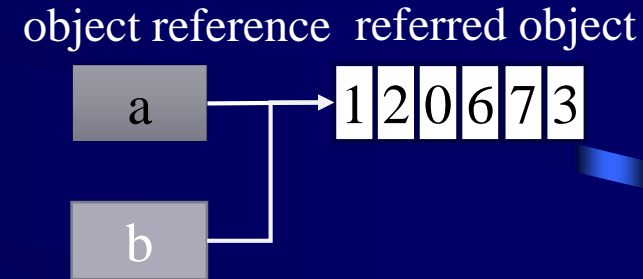
- 多个对象引用可指向一个相同对象

- Array b = a;

- 多个指向同一对象的引用实际产生了对象共享

- 如果被引用对象不可变，共享不会产生风险
- 如果被引用对象可变，共享有可能产生不可预期的效果

- b[3] = 9;



对象共享与对象复制

- 对象引用复制即产生共享
- 如果想要复制对象，而不是共享
 - 使用对象的clone方法产生新的对象
 - 或者，new一个新对象，拷贝对象的属性值（要求对象的属性值外部可访问）

```
public class Poly{  
    private int[] terms;  
    private int deg;  
    ...  
}
```



```
public class Poly{  
    private int[] terms;  
    private int deg;  
    public Poly clone(){  
        Poly p = new Poly();  
        p.terms = this.terms;  
        p.deg = this.deg;  
        return p;    }  
}
```

- 延伸问题：（作业）
- Java的对象如何算相同

第四章 面向对象程序设计语言

4.1 Smalltalk语言

对象的思想最早源于人工智能研究，60年代末描述智能对象的框架 (frame) 即封装了许多槽 (slot)，槽既可以是属性 (数据) 也可以是行为 (操作) 和 (约束)。但最早见诸文献是sketchpad提到的OO图形学 (1963)。

60年代挪威的Dahl和Nygaard为模拟系统研制了SIMULA-67语言，首先提出封装的类和动态生成实例对象的概念。

60年代末，美国犹他大学Alan Kay到Xerox公司PaloAlto研究中心参加了Dynabook项目。该项目的硬件是Star (个人机的前驱) 软件是Smalltalk。



Smalltalk语言关键贡献

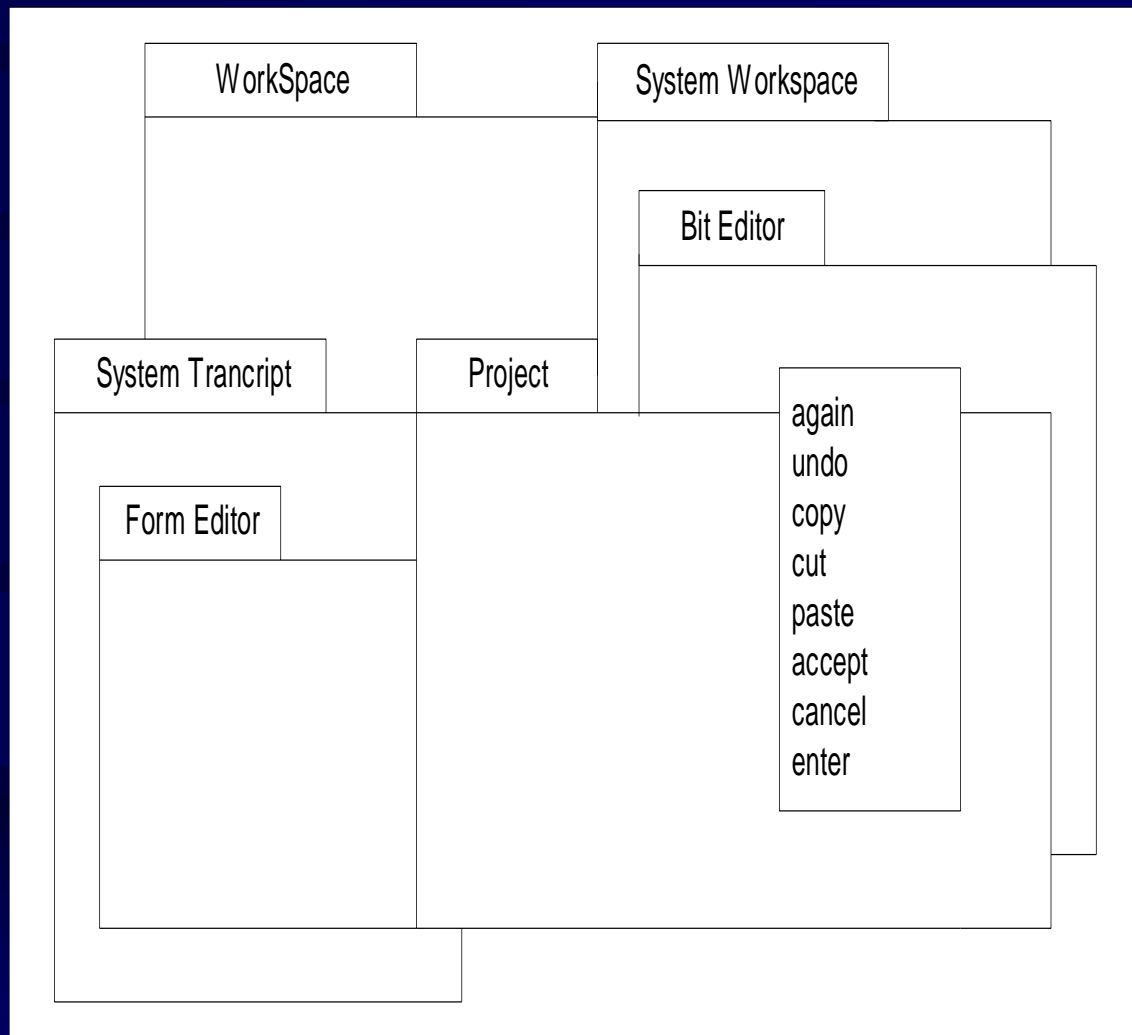
- 1972年Dan Ingalls完成Smalltalk-72第一个实用版
 - 语言完全基于Simula的类和消息的概念。
 - 语言没有固定的语法，语法分析由类本身完成。
- 1976版本
 - 引入了继承和子类的概念。
 - 确定了语言的语法，这使得编译器能够产生高效、可执行、精炼的二进制代码。
 - Larry Tesler设计了浏览器，这极大地提高了Smalltalk程序员的编程效率。
- 1980版本（正式发行版本）
 - 引入了元类的概念。
 - 引入MVC（Model-View-Controller）系统以方便交互式应用软件的开发。

4.1.1 Smalltalk系统

- 语言核心 (Kernel)
- 程序设计系统
- 程序设计范型 (Paradigm)
- 用户界面模型 (User Interface Model)

4.1.2 用户界面模型

- 系统工作空间 (System WorkSpace)
- 工作空间 (WorkSpace)
- 系统副本 (System Transcript)
- 项目 (Project)
- 两种图形编辑窗 (Form和Bit)



- 系统浏览器 (System Browser) 窗

System Browser			
□□□□□ CLASS CATEGORIES MENU	□□□□ CLASSNAMES MENU		□□□□□□ MESSAGE CATEGORIES MENU
	Instance	Class	
正文 TEXT			

- 用户就是按浏览窗中显示的模板填写程序。

4.1.3 语言核心

(1) 保留字

只有五个nil, true, false, self, super

(2) 字面量

字符字面量 / 数字面量 / 符号字面量 / 数组字面量

(3) 限定符和特殊符号

" ' \$ # #() , ; : | :=或← ↑ [] () { }

(4) 变量

实例变量 / 类变量 / 临时变量 / 全局变量 / 汇聚变量 / 参数

(5) 消息表达式与语句

消息表达式的一般格式是：

对象 选择子 参数



Smalltalk的消息表达式有三种：

·单目的 不带参数

tree class	消息class 发向tree，得到tree的类。
0.3 sin	消息sin 发向0.3，得sin(0.3)
Array new	消息new 发向Array，创建-Array 的实例

· 双目的

3+4 消息 '+' 带参数4发向对象3，得对象7。

100@ 50 消息 '@' 带参数50发向对象100，得(100, 50)

(sum/count) * reserve amount

双目，括号优先

单目优先

双目

·关键字消息表达式

用关键字 (带有 ‘:’ 的选择子) 描述的双目表达式, 也是自左至右释义。

```
anArray  at: 3 put:100  
finances totalSpentOn: 'food'
```

·赋值 变量在不同时间可赋以不同对象, 任何表达式加上赋值前缀 ‘←’

```
quantity←19.  
name←'chapter 1'。
```

```
foo ← array at:4。 数组第4元素与 ‘foo’ 同名
```

·块表达式

```
[ :x:y | BicPen goto: x@y ]  
[ :x:y | BicPen goto: x@y ] value: 100 value: 250  
BicPen goto 100@ 250  
| aBlock |  
aBlock ← [ 'This is a String' displayAt: 500@ 500 ].  
Display white.  
aBlock value
```

(6) 控制结构

条件选择一般形式是：

布尔子表达式

ifTrue: [‘真’ 块执行]

ifFalse: [‘假’ 块执行] “可以不出现”

如：number<0

ifTrue: [absValue←number negated]

ifFalse: [absValue←number]

条件重复一般形式是：

[布尔块表达式]

whileTrue: | whileFalse: [重复块]

如：[index>listSize]

whileFalse: [list at:index put: 0.
index←index+1]

(7) 消息/方法

消息模式 | 临时变量 | 语句组

```
newAt: initialLocation | newBox |
```

```
newBox←self new.
```

```
newBox setLoc: initiaPLocation tilt: 0 size: 100 scribe: pen new.
```

```
newBox show.
```

```
setLoc: newLoc tilt: newTilt size:newSize scribe: newScribe | |
```

```
Loc←newLoc. titl←newTilt.
```

```
size←newSize. scribe← new Scribe
```


4.1.4 Smalltalk文件系统与虚机

Smalltalk是编译—解释执行的，Smalltalk源程序经编译器得到虚映象 (Virtual image)，虚映象由字节代码中间语言编写，由Smalltalk虚机解释执行。相应的文件系统管理三种文件：源文件、变更文件、映象文件。

由于Smalltalk是交互式的，被编译的方法在执行期间出了问题要反应到源程序，则要对映象文件施行反编译 (decompilation)

Smalltalk的虚机是一个新软件，它有三个功能部分：

- 存储管理器
- 虚映象解释器
- 基本例程 用汇编码写出的底层方法实现

4.1.5 Smalltalk程序设计范型

- 程序设计在类的层次上进行，由类静态 (于工作空间指明向类发出消息) 或动态 (方法运行时) 生成实例对象。每个对象当接受某消息并执行其方法的消息表达式时都是在自向其它对象发消息。

4.1.5.1 一个简单的Smalltalk程序

统计字母出现频率

| s f | “定义了两个临时变量”

s ← Prompter prompt: 'enter line' default: '' .

“s是Prompter的实例，将关键字表达式的结果束定于s”

“意即输入一行字符串，若不输入，S为空串”

f ← Bag new. “f是Bag的实例”

s do: [:c | c isLetter ifTrue: [f add: c asLowerCase]]

“s在Prompter中找方法do: 的模式，若找不到，找prompter的”

“父类直到Object. C是块变量，意从S中拿出某字符，isLetter”

“是消息模式，判C是否字符，若为真执行内块”。

“内块中f找add: 消息模式，从Bag直至上层父类，找到先执”

“行右边子表达式”。

c asLowerCase是单目表达式，同样要在Prompter中找asLowerCase匹配，也是不成向上找。它返回是“第k个”小写字母，add: 把它发送到对象f的第k个位置上并与原数相加。

↑ f

“返回f中的值”。

这个程序一共四句。如果掀鼠标使菜单项 'doit' 工作并输入：

“Smalltalk is a programming Language for developing soluions to both simple and complex problem.”

则输出的f值是：

7	1	1	2	4	1	5	1	5	1	7	4	4	7	3	3	6	3	2	1
a	b	c	d	e	f	g	h	i	k	l	m	n	o	p	r	s	t	u	v

例 字频统计对比程序

Pascal

```
PROGRAM Frequency
```

```
CONST Size=80;
```

```
VAR s: string[size];
```

```
    k,i: Integer;
```

```
    c:Char;
```

```
f:ARRAY[1..26] OF Integer;
```

```
BEGIN
```

```
    Writeln('enter line');
```

```
    Readln(s);
```

```
    FOR i:=1 TO 26 DO
```

```
        f[i]:=0;
```

```
    FOR i:=1 To size DO
```

```
        BEGIN
```

```
            c:=asLowerCase(s[i]);
```

```
            if isLetter (c) THEN
```

```
                BEGIN
```

```
                    k:=ord(c)-ord('a')+1;
```

```
                    f[k]:=f[k]+1
```

```
                END
```

```
            END;
```

```
    FOR i:=1 To 26 DO
```

```
        Write(f[i], ' ')
```

```
END.
```

Smalltalk

“无消息模式方法，宜写算法”

```
| s c f k |
```

“定义了四个临时变量”

```
f←Array new: 26.
```

“f是Arrey实例长度26”

```
s←Prompter
```

```
    prompt: 'enterline'
```

```
    default: ' '.
```

“S是Prompter的实例，装输入字符串”

```
1 to:26 do:[:I|f at:I put:0].
```

```
1 to: size do:[:I|
```

```
    c←(s at:i) asLowerCase.
```

```
    c isLetter ifTrue: [
```

```
        k←c asciiValue
```

```
        - $a asciiValue + 1.
```

```
        f at:k put: (f at:k) + 1
```

```
    ]
```

```
].
```

```
↑ f
```

4.1.5.2 类协议

Smalltalk编程定义的方法是在各个类协议(protocol)中定义。这里有三种情况:

- 如果所有类的变量和方法都不合要求, 则作类定义时写超类名为Object, 继承它的最一般操作, 如new, at:等, 其它全由程序员设计。
- 如果有一个类很接近你需要的类, 则以它为超类, 增、删、派生类的变量和方法。对于‘删去’的变量和方法是以同名变量和方法‘覆盖’, 重新定义或置空(动作)。
- 如果一个类完全满足你的需要, 则在工作窗直接向它发消息, 生成实例并初始化, 并将它绑定到实例变量名上。

4.1.5.2 类协议

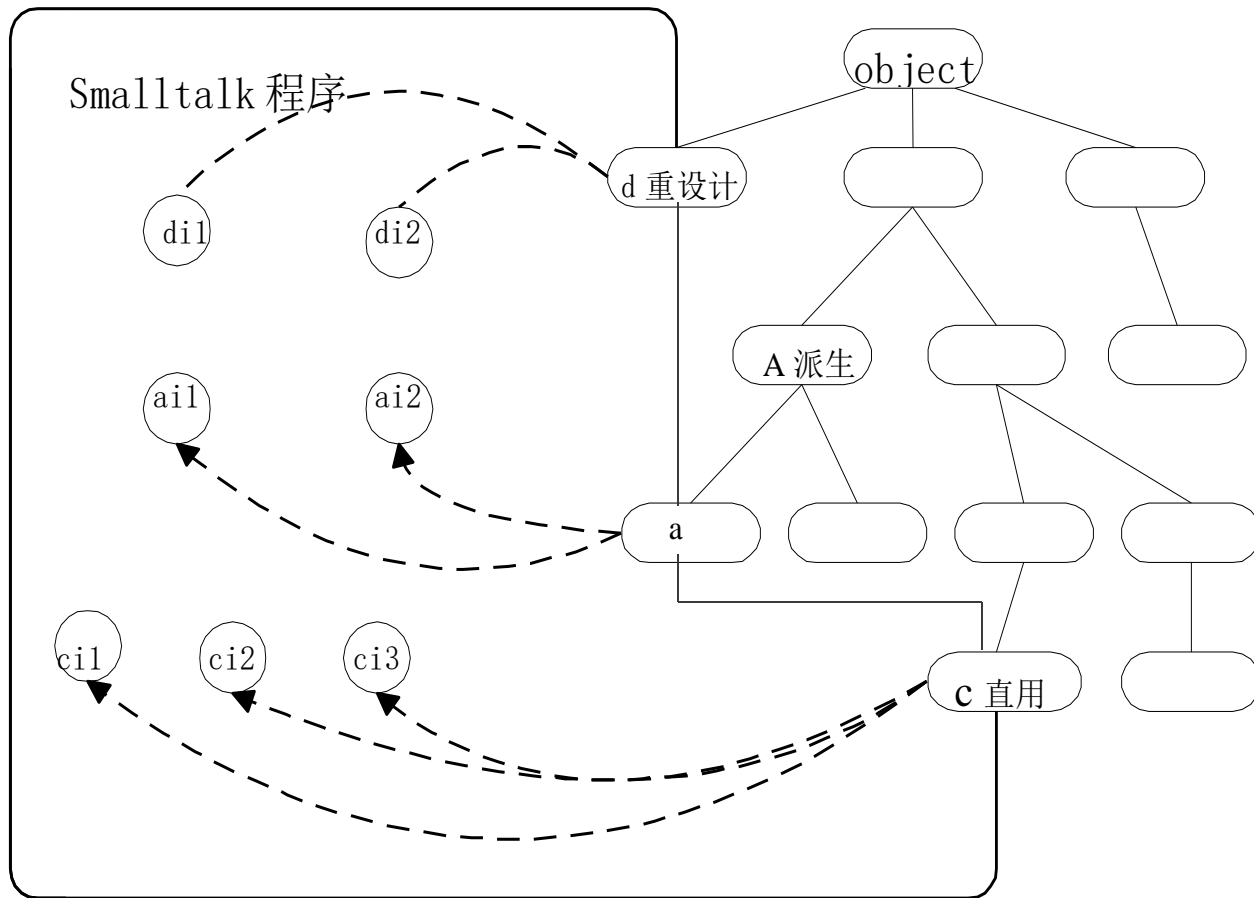
类名 超类名 实例变量名 类变量名 汇集变量名	标识符 标识符 标识符表 标识符表 标识符
类方法: 方法 1 方法 2 . . 方法 n	
实例方法: 方法 1 方法 2 . . 方法 n	

单继承，只有一个
用于实例对象
用于类对象
充作若干类共享的汇聚字

用于创建实例，并初始化

如上例 Array 中的方法 new:

刻画实例对象行为
如上例中 asLowerCase,
at:put:, isLetter 是对象
s, f, c 的方法。



4.1.5.3 一个完整的Smalltalk程序

家庭财务帐目

建立全部流水帐类，直接挂在Object上

class name FinancialHistory

superclass Object

instance variable names 'cashOnHand incomes expenditures'

category 'Financial Tools'

class method

initialBalance:amount | | “建立流水帐本初始为amount（元）”

↑super new setinitialBalance: amount

new | | “建立流水帐本初始为0（元）”

↑super new setinitialBalance:0

instance method

receive: amount from: source | |

incomes at: source put: (self total ReceivedFrom:source)+amount.
“从来源source接收到的钱数，因而手头现金增加”.

cashOnHand←cashOnHand + amount.

incomes changed

spend: amount for: reason | |

“为事由reason支付的钱数，因而手头现金减少。”

expenditures at: reason put: (self totalSpentFor: reason) + amount.


```
cashOnHand ← cashOnHand - amount.
```

```
expenditures changed
```

```
CashOnHand | |
```

```
↑ cashOnHand
```

```
expenditures | |
```

```
↑ expenditures
```

```
incomes | |
```

```
↑ incomes
```

```
totalReceiveFrom: source | |
```

```
(incomes includesKey: source)
```

```
ifTrue: [↑ incomes at: source]
```

```
ifFalse: [↑ 0]
```

```
totalSpentFor: reason | |
```

```
(expenditures includesKey: reason)
```

```
ifTrue: [↑ expenditures at: reason]
```

```
ifFalse: [↑ 0]
```

```
private
```

```
SetInitialBalance: amount | |
```

```
cashOnHand ← amount.
```

```
incomes ← Dictionary new.
```

```
expenditures ← Dictionary new
```

“回答当前手头现金”

“回答支出细目”

“回答收入细目”

“回答自source收钱总数”

“回答在reason项上总支出”

“实例变量初始化”

```
Smalltalk at: # HouseholdFinances put: nil.  
HouseholdFinances←FinancialHistory initialBalance: 1560  
HouseholdFinances spend: 700 for: 'rent'.  
HouseholdFinances spend: 78.53 for : 'food'.  
HouseholdFinances receive: 820 from: 'pay'.  
HouseholdFinances receive: 22.15 from: 'interest'.  
HouseholdFinances spend: 135.65 for: 'utilities'.  
HouseholdFinances spend: 146.14 for: 'food'.
```

4.1.6 Smalltalk程序设计系统

在Smalltalk中，系统支持程序也是作为类挂在Object之下，包括算术运算、数据和控制结构的实现、输入/出、随机数生成器等。

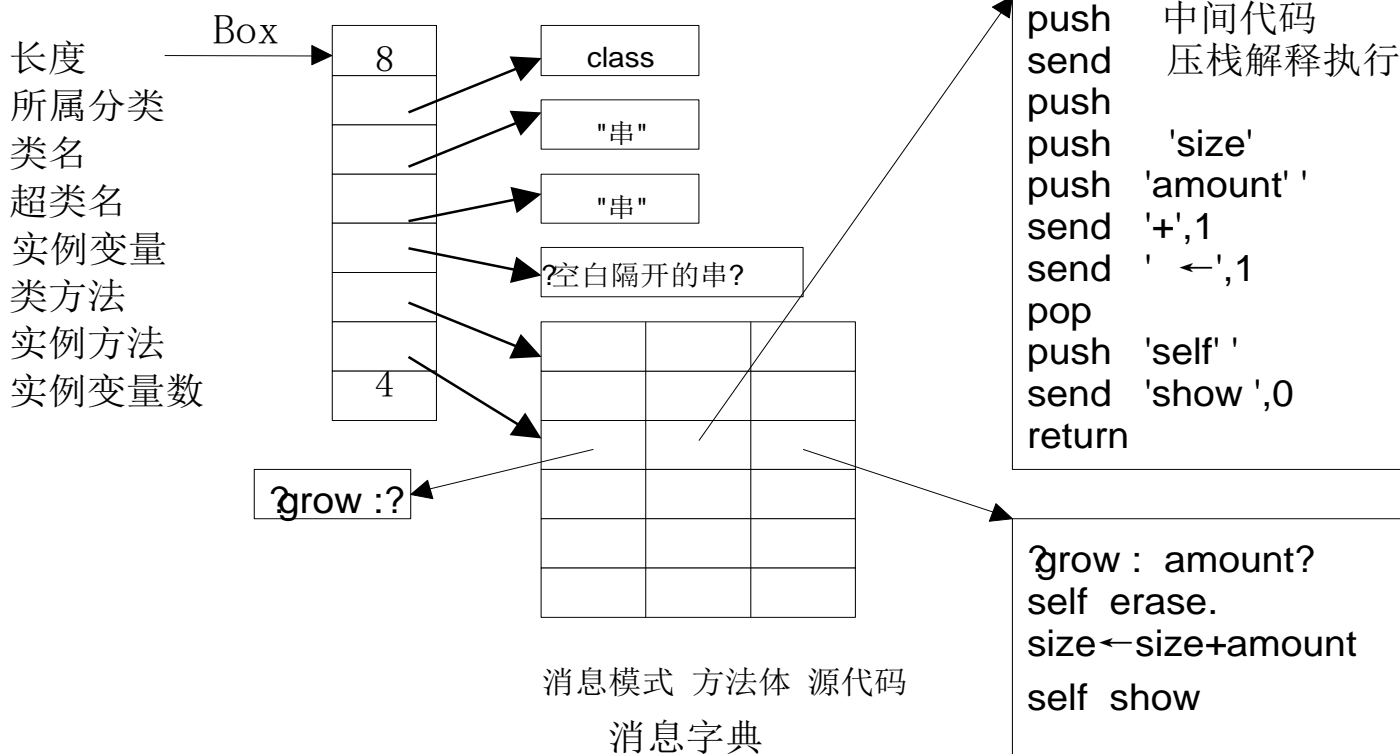
有一些类是辅助程序设计过程的，语法分析器、编译器、解释器、反编译器这些对象的方法都有源代码，目标码两种形式。

还有一些对象表示类和方法的结构，以便程序员追踪系统。

还有将方法和向其发消息的对象联结起来的对象。这些对象统称环境(contexts)类似其他语言实现中的堆栈帧和活动记录。

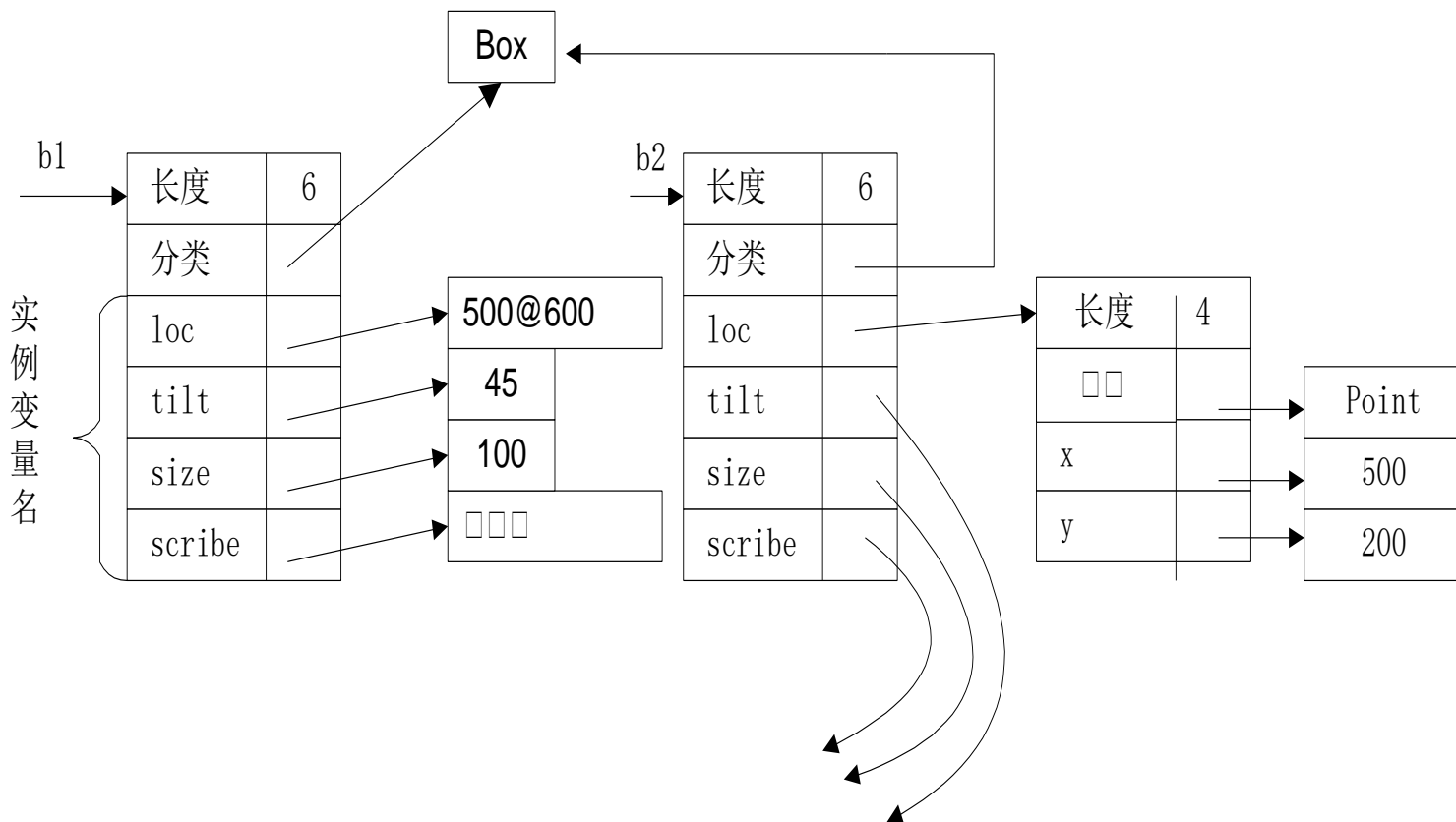
4.1.7 Smalltalk的对象、类、方法的实现

类的存储



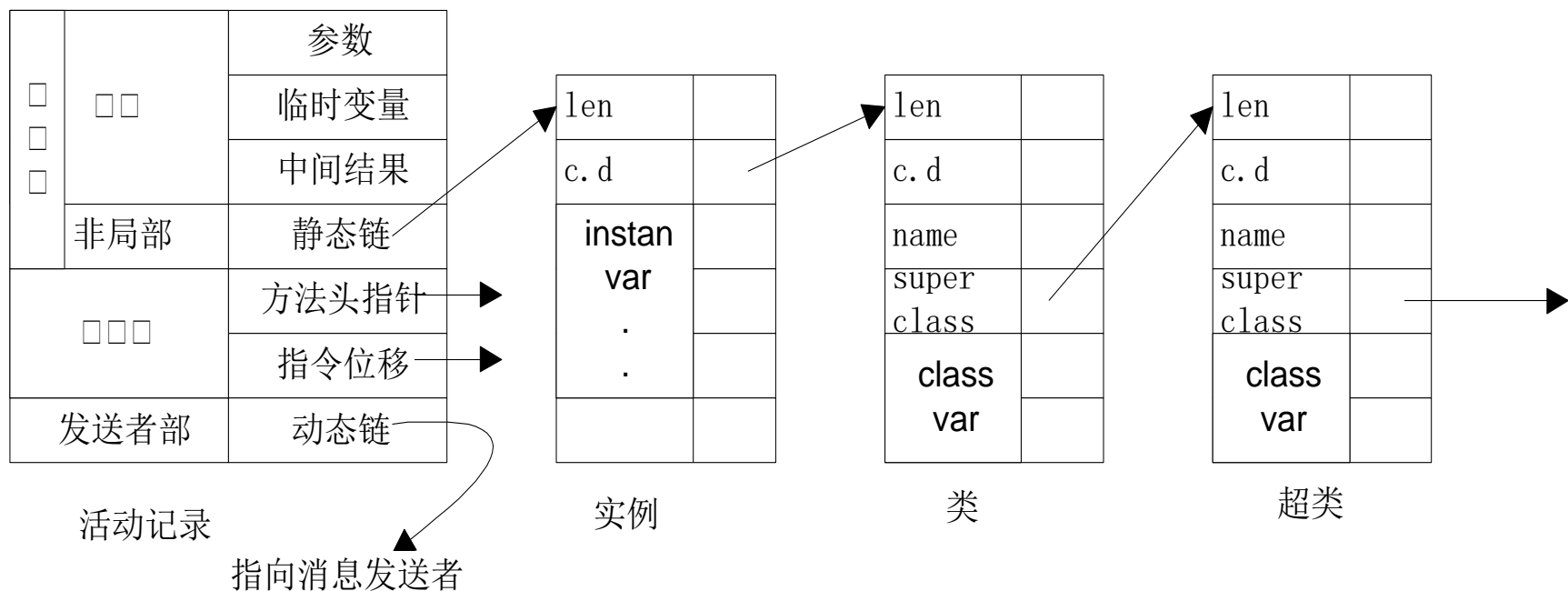
实例对象的存储

实例对象只存放数据，其存储格式如下图：



活动记录

- 环境部分
- 指令部分
- 发送者部分



4.2 面向对象的发展与概念

- 为什么需要面向对象？
- OO 语言的发展
- 面向对象的基本概念

重用的问题

- 实践中人们认识到重用已有开发结果的重要性，提出了软件重用的概念
 - 最早的重用单元是子程序，如 **Fortran** 的子程序库
 - 子程序是纯粹的过程抽象，基于子程序的重用有很大局限性
 - 模块是更合适的重用单元，因为模块可以包装任何功能，更灵活
- 重用中有一种常见情况：软件开发中遇到的新问题常与解决过的问题（可以重用的库提供的功能）类似，但又不完全相同
 - 已有模块的功能与需要有差异，无法以其“现有”形式直接使用
 - 如果模块功能的改变只能通过修改源代码的方式进行，程序员就只能拷贝这个模块的源代码，深入研究后再设法修改，以满足新需求
- 但问题是有没有可以使用的源代码？常常没有：
 - 模块可能是购入的，提供商不提供源代码
 - 模块可能是过去的遗产，源代码已经丢失或部分缺失

模块和程序组织

- 常规的程序单元缺乏弹性，定义好的子程序/模块都是固定功能的实体，难以提供“定制”的方式部分地改变功能以满足实际需要的变化
- 通过模块定义的抽象数据类型是相互独立的，不同模块之间无任何关系
 - 而实际情况中，常常需要定义和使用一些相互有关的类型，可能需要把它们送给同一个函数/过程去处理，以同样方式存储
 - 变体和联合机制就是为了迎合这方面的需要，但它们没有类型安全性，且未能提供解决类似问题的统一框架，难用于应付更复杂的情况
 - 支持相关类型，可能给程序的结构组织带来新的可能性
- 如何在抽象数据类型的框架中提供这一类功能，也是需要解决的问题

OO 发展史

- **OO** 技术和思想中的一个基本方面是数据和操作的封装
 - 这方面的基本想法：一组数据与关联之上相关的操作形成一个对象。其内部数据构成对象的状态，操作确定对象与外界交互的方式
 - **OO** 并不是从模块化程序设计发展出来的，它有自己的发展历程
 - **OO** 的思想与模块化的思想是并行发展，一直相互影响、相互借鉴
- **Simula 67** 是 **OO** 概念的鼻祖，其设计目标是扩充 **Algol 60**，以更好地支持计算机在模拟方面的应用。**60** 年代在挪威计算中心设计和实现，主持其工作的 **Ole-Johan Dahl** 和 **Kristen Nygaard** 获得 **2001** 年图灵奖
 - **OO** 的三个基本要素：封装、继承和动态方法绑定都源于 **Simula**
 - 类的概念源自 **Simula**，其设计中提出用类定义把一组操作与一组数据包装起来。**Simula** 的这些重要想法是模块概念和 **OO** 的起源
 - **Simula** 只提供了基本封装，并没有对封装的保护，也没有信息隐藏

OO 发展史

软件实践也需要 OO 的思想，并逐渐开发了相关的支撑技术，包括：

- 封装的思想在面向模块的语言里发展，提出了许多重要概念和想法，如
 - 作用域规则，开的或者闭的作用域
 - 界面与实现
 - 透明类型与隐晦类型，访问控制，等等
- 数据驱动的程序设计技术：
 - 将计算功能（子程序）束定于程序里处理的数据（结构），使我们在程序里可以从数据对象出发去启动相应的计算过程
 - 在一些非常规的语言（如函数式语言）里，可以通过引用的概念提供函数/过程与数据之间的束定
 - 常规语言（如 C）引进了指向函数的指针，在实现数据驱动程序设计的过程中起到了重要作用，也成为面向对象语言实现的技术基础

OO 发展史

继承和动态束定等被 **Smalltalk** 发展，形成目前 **OO** 的基本概念框架

- 程序里以类的方式定义各种数据抽象
- 类可以通过继承的方式扩充新功能，这样定义的新类（子类，派生类）自动继承已有类（基类，超类，父类）的功能
- 对象是类的实例，是程序运行时的基本数据单元
- 派生类的对象也看作是原有基类的对象，可以当作基类的对象使用（子类就是子类型，**Liskov** 代换原理，**2008** 年图灵奖）
- 类定义了对象的状态成分（数据成员）和一组相关操作（称为方法）
- 方法调用总是针对某个对象进行的，将方法调用看作是给相应对象送一个消息，对象通过执行相应操作的方式对消息做出响应
- 对一个消息执行什么方法，由接收消息的对象的类型确定（根据该对象所属的类确定，这就是动态束定）
- 计算，就是一组对象相互通讯的整体效果（对计算的另一种看法）

OO 发展史

Smalltalk 还有一些独特的东西：

- 变量采用引用模型，变量无类型，可以引用任何对象
- 语言里的一切都是对象：
 - 类也是对象，通过给类送 **new** 消息的方式要求创建类的实例
 - 各种控制结构也是通过消息概念建立的
 - 条件和逻辑循环是逻辑对象对特定消息的响应
 - 枚举循环是整数对象对特定消息的响应
- 采用单根的分类层次结构，以类 **Object** 作为所有类的超类
- 提供了块（**block**）的概念，作为控制结构的抽象机制
- 提出了容器的概念，开发了一个功能丰富的类库
- 与程序开发环境的紧密结合，并开发了 **GUI** 的基本概念和相关技术

OO 发展史

随着 **Smalltalk** 的成功，人们看到了 **OO** 的潜在威力

- 许多人开始研究如何把 **OO** 概念有效集成到常规语言里，提出了一批已有语言的 **OO** 扩充和许多新 **OO** 语言，如 **Object-Pascal**、**Object-C** 等
- 其中前期最成功并得到广泛应用的是 **C++**。**C++** 在 **OO** 概念的广泛接受和应用方面功不可没（具体理由见后面讨论）。原因：
 - 在面向对象和高效程序之间取得较好的平衡
 - **OO** 概念与常规语言的合理集成（在当时），支持数据抽象和面向对象的系统设计和程序设计，支持多泛型程序设计的结合，使与数据抽象和 **OO** 有关的许多新概念和新技术逐渐被实际软件工作者接受
- 随后是 **OO** 分析、**OO** 设计和基于 **OO** 的软件开发等等
- 后来的其他成功语言包括 **Java**，微软提出 **C#**，等等
- 出现了一些基于对象的脚本语言，如 **Python**，**Ruby** 等

面向对象程序设计的核心问题

- 代码重用是基本驱动
 - 提高生产率
 - 原有程序语言无法在原有类型系统进行便捷的修改
 - 所有数据类型的定义都是独立的，并在同一个层次上，不可能构造一个适合问题空间的程序。
- 继承是面向对象程序设计语言的核心
 - 抽象数据类型是基础
 - 抽象数据类型形成新类型
 - 类型成员必然继承新类型的共同部分
 - 继承提供了一种解决抽象数据类型的复用所面临的修改问题，以及程序组织问题的办法。
- 动态束定是实现手段
 - 与谁束定（也是多态问题）
 - 接受者决定一个消息的含义

面向对象程序设计的关注点

- 派生类/子类，父类/超类
- 方法，消息
- 消息协议/消息接口
- 公有、私有与保护类型
- 覆盖方法
- 单继承，多继承
- 多态引用，抽象方法
- 构造，析构
- 动态束定，静态束定
-

面向对象程序设计的设计的关注点

- 父类和子类的常见区别
 - 父类可以定义不能被子类访问的私有变量或方法
 - 子类可以在继承父类成员的基础上再添加成员
 - 子类可以对一个或多个继承来的方法进行修改。修改后的方法与原方法有相同的名字，通常具有与原方法相同的协议。
- 子类对比子类型的概念
- 把子类看作是子类型（通常），如果 **D** 是 **B** 的子类，那么：
 - 若 **o** 是 **D** 类型的对象，那么 **o** 也看作是 **B** 类型的对象
 - 若变量 **x** 可以引用 **B** 类的对象，那么它也可以引用 **D** 类的对象
- 单继承与多继承
 - 支持多继承好吗？
 - 菱形继承（共享继承）

面向对象的语言

虽然基本框架类似，不同面向对象语言之间也存在很大差异：

基本问题：采用什么样的对象模型

- 采用单根类层次结构，还是任意的类层次结构？
- 提供哪些继承方式？
 - 例如 **C++** 里提供了三种继承方式
- 允许多重继承？还是只允许单继承？
- 是否提供丰富完善的访问控制机制？
- 基于类的模型，还是基于对象或原型的模型
（如 **JavaScript**）
- 对象本身的独立性（是否允许不属于任何一个类的对象）
- 类本身是不是对象？

面向对象的语言

其他情况：

- 是不是追求“纯粹”的面向对象语言？
 - **Smalltalk** 尽可能追求“面向对象”理想，完全是重新设计的新语言
 - **Java** 是接近理想的语言，但希望在形式上尽可能靠近常规语言
 - **C++** 设法在支持系统程序设计的过程性语言 **C** 上“扩充”支持面向对象的机制，是一种多范型语言，支持多种程序设计方式
- 采用值模型还是引用模型。
 - 从本质上说，只有采用引用模型才能支持方法的动态绑定，因此大多数面向对象语言采用引用模型
 - **C++** 采用值模型，可以创建静态对象或栈对象，但只有通过对象引用或指向对象的指针才能实现面向对象的动态绑定行为
 - **Java** 只能把 **OO** 功能应用于用户定义类型，基本类型采用值模型

面向对象的语言

- 是否允许静态对象或者堆栈对象（自动对象）？多数面向对象语言只支持堆对象（通过动态存储分配创建的对象）
 - **C++** 支持静态对象和自动对象，这种设计是希望尽可能借助于作用域规则来管理对象，避免依赖自动存储管理系统（**GC**）
 - 为在这种环境下编程，人们开发了许多利用自动对象的对象管理技术，如句柄对象，对象的“创建即初始化”技术等
- 是否依赖自动垃圾收集（**GC**）。由于 **OO** 程序常（显式或隐式地）创建和丢弃对象，对象之间常存在复杂的相互引用关系，由人来完成对象的管理和回收很困难。大多数 **OO** 语言都依赖于自动存储回收系统
 - **GC** 的引入将带来显著的性能损失，还会造成程序行为更多的不可预见性（**GC** 发生的时刻无法预见，其持续时间长短也无法预计）
 - **Java** 等许多语言都需要内置的自动垃圾收集系统
 - **C++** 是例外，其设计目标之一是尽可能避免对自动存储回收的依赖，以支持系统程序设计，提高效率，减少运行时间上的不确定性

面向对象的语言

- 是否所有方法都采用动态束定？
 - 动态束定很重要，但调用时会带来一些额外的开销，如果需要调用的方法能够静态确定，采用静态束定有速度优势
 - 大部分语言里的所有方法都采用动态束定
 - **C++** 和 **Ada** 提供静态束定（默认）和动态束定两种方式
- 一些脚本语言也支持面向对象的概念。例如，
 - **Ruby** 是一个纯面向对象的脚本语言，其中的一切都是对象，全局环境看作一个匿名的大对象，全局环境里的函数看作这个对象的成员函数。它还有另外一些独特性质
 - **JavaScript** 支持一种基于对象和原型的面向对象模型。

作业：为什么JavaScript中有类吗？举例说明你的观点

面向对象的语言

- 总而言之，虽然今天面向对象的模型和语言已成为主流程序设计方法和主流程序语言，还正在发展和研究中
 - 许多语言的 **OO** 机制非常复杂，实际还不断提出一些新要求，使一些 **OO** 语言在发展中变得越来越复杂
 - 如何提供一集足够强大，而且又简洁清晰的机制支持 **OO** 的概念和程序设计，还是这个领域中需要继续研究的问题
 - **OO** 语言有关的理论研究还处在研究与应用阶段

OO 语言需要提供的新机制

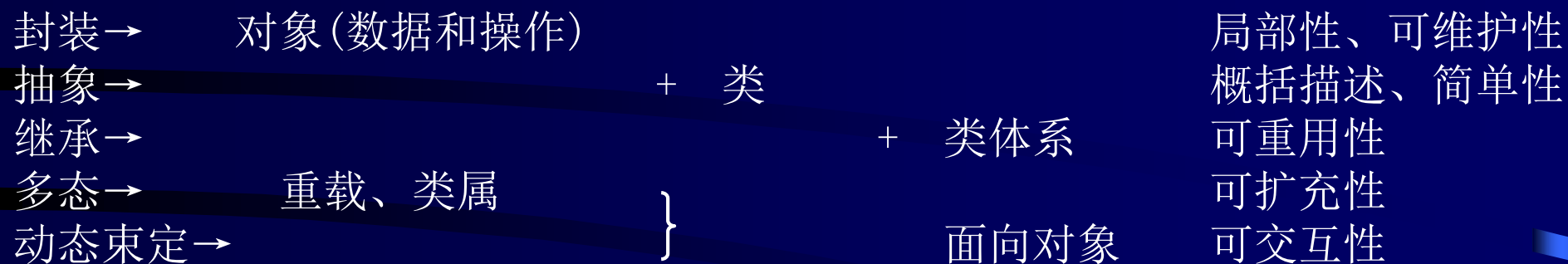
- 定义类的语言机制（语言提供特殊的描述结构）
- 描述或定义对象的机制
- 继承机制，描述类之间的继承关系。可能定义继承关系的性质（如 C++ 里的类继承有 **public**、**protected** 和 **private** 三种方式）
- 与对象交互的机制（方法调用，消息传递）
- 初始化新对象的机制（最好能自动进行，避免未初始化就使用的错误）
- 类型对象的动态转换机制（转换对一个具体对象的观点）
- 控制类成员的访问权限的机制
- 对象销毁前的临终处理机制（最好能自动进行）
- 对象的存储管理机制

其他机制：

- 运行中判断对象的类属关系的机制、自反等等

4.3 面向对象语言的基本特征与实现

P. Wegner总结了OO语言的发展, 给出以下图示澄清了概念:



基于对象的语言

Ada 83, Actor

基于类的语言

CLU

simula 67

面向对象语言

Smalltalk、Eiffel

C++, Ada 95, Java

OO 程序

- 先看一点 OO 程序，复习一下基本 OO 程序的特征
- 这里看一段定义了几个类的 C++ 代码

```
class list_err {                                // exception
public:
    char *description;
    list_err (char *s) {description = s;}
};

class list_node {
    list_node* prev;
    list_node* next;
    list_node* head_node;
public:
    int val;                                     // the actual data in a node
    list_node () {                               // constructor
        prev = next = head_node = this;        // point to self
        val = 0;                               // default value
    }
    list_node* predecessor () {
        if (prev == this || prev == head_node) return 0;
        return prev;
    }
    list_node* successor () {
        if (next == this || next == head_node) return 0;
        return next;
    }
};
```

定义 list_node 类，用于实现带头结点的双向循环链接表

每个结点里有一个域指向表头结点

00 程序

```
int singleton () {
    return (prev == this);
}

void insert_before (list_node* new_node) {
    if (!new_node->singleton ())
        throw new list_err ("attempt to insert node already on list");
    prev->next = new_node;
    new_node->prev = prev;
    new_node->next = this;
    prev = new_node;
    new_node->head_node = head_node;
}

void remove () {
    if (singleton ())
        throw new list_err ("attempt to remove node not currently on list");
    prev->next = next;
    next->prev = prev;
    prev = next = head_node = this;    // point to self
}

~list_node () {
    // destructor
    if (!singleton ())
        throw new list_err ("attempt to delete node still on list");
}

};
```

00 程序

定义一个list类

```
class list {
    list_node header;
public:
    // no explicit constructor required;
    // implicit construction of 'header' suffices
    int empty () {
        return (header.singleton ());
    }
    list_node* head () {
        return header.successor ();
    }
    void append (list_node *new_node) {
        header.insert_before (new_node);
    }
    ~list () {
        // destructor
        if (!header.singleton ())
            throw new list_err ("attempt to delete non-empty list");
    }
};
```

注意: header 是个 list_node
定义的是有头结点的循环链表

OO 程序

通过继承定义 **queue** 类。（只是作为示例）

```
class queue : public list {                                // derive from list
public:
    // no specialized constructor or destructor required
    void enqueue (list_node* new_node) {
        append (new_node);
    }
    list_node* dequeue () {
        if (empty ())
            throw new list_err ("attempt to dequeue from empty queue");
        list_node* p = head ();
        p->remove ();
        return p;
    }
};
```

00 程序

- 还可以定义通用的容器类：
 - 基本容器类没有具体数据域，不保存具体类型的元素，只实现容器操作，如：一些基本判断谓词，插入删除等等
 - 通过继承实现存储具体类型的元素的具体容器

00 程序

```
class gp_list_node {
    gp_list_node* prev;
    gp_list_node* next;
    gp_list_node* head_node;
public:
    gp_list_node ();          // assume method bodies given separately
    gp_list_node* predecessor ();
    gp_list_node* successor ();
    int singleton ();
    void insert_before (gp_list_node* new_node);
    void remove ();
    ~gp_list_node ();
};

class int_list_node : public gp_list_node {
public:
    int val;                  // the actual data in a node
    int_list_node () {
        val = 0;
    }
    int_list_node (int v) {
        val = v;
    }
};
```

通用的表结点类

派生的 int 表结点类

使用这种 int 表的问题:

如果需要访问结点的数据内容, 必须对取出的结点做强制

面向对象特征的实现

- 实现面向对象的语言，需要考虑它的几个标志性特征的实现
- 封装是一种静态机制，如 **C++/Java** 一类语言的各种访问控制机制也是静态的，都可以通过在符号表里记录信息，在编译中检查和处理
- 方法的实现与以模块为类型时局部子程序的实现一样。由于每个方法调用有一个调用对象，因此方法需要一个隐含指针，被调用时指向调用对象，所有对该对象的数据成员的访问都通过这个指针和静态确定的偏移量进行
- 许多语言以这一指针作为一个伪变量，称为 **this** 或者 **self**，通过这种指针
- 访问调用对象，方式上与通过指针访问普通结构一样
- 实现面向对象语言的关键是两个问题：
 - 继承的实现，使派生类型的对象能当作基类的对象使用
 - 动态束定的实现，能够从（作为变量的值或者被变量引用的）对象出发，找到这个对象所属的类里定义的方法

封装

- 封装是一种静态机制，仅仅在程序加工阶段起作用，有关情况与模块机制类似，在加工后的程序里（可执行程序里）完全没有关于封装的信息
- 不同语言里对类的访问控制可能不同：
 - 作为“开模块”（允许以特定方式任意访问类成员）
 - 作为“闭模块”（凡是没有明确声明可访问的都不可访问）

对基本封装机制的扩充是引进进一步的控制

- C++ 引进成员的 **public**、**protected** 和 **private** 属性，提供细致的访问控制
- C++ 还允许定义派生类的不同继承方式，控制对基类成员的访问：
 - **public** 继承
 - **protected** 继承，使基类的 **public** 成员变成派生类的 **protected** 成员
 - **private** 继承，使基类的所有成员变成派生类的 **private** 成员

Java 类的作用域——作业

```
class TalkingClock
{
    private int interval;
    private boolean beep;
    public TalkingClock(int interval, boolean beep){...}
    public void start(){...}
```

```
public class TimePrinter implements ActionListener
{
    ...
}
```

```
class TimePrinter implements ActionListener
```

```
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("At the tone, the time is " + new Date());
        if(beep) Toolkit.getDefaultToolkit().beep();
    }
}
```

静态域和静态方法

许多面向对象语言的类里可以定义静态域和静态方法

- **C++/Java** 允许类里定义静态数据域
- **Smalltalk** 把普通的对象域称为实例变量，表示在这个类的每个实例里都有这些成分的一份拷贝；把静态数据域称为类变量
- 类的静态数据域并不出现在实例对象里，它们是类封装的静态数据成分，提出具有静态生存期，在类的作用域里可直接访问。类外能否访问由语言确定（提出有与其他成员一样的访问控制）

静态方法和静态域的一些情况：

- 类的静态数据成员可以在静态区实现，在程序运行之前静态分配，在程序的整个执行期间保持其存储
- 类的静态方法可访问静态数据成员，其他方法也可以访问静态数据成员
- 可以把静态数据成员看作本类的所有对象共享的信息
- 类对象可以通过静态数据成员交换或者共享信息
- 静态成员是静态创建的，其初始化在程序开始执行前完成（或者在语言定义的适当时刻完成），只做一次
- 静态成员的初始化中不能调用类的普通方法（因为没有对象）

静态域和静态方法

静态方法相当于普通子程序，只是具有类封装（类作用域）。特点：

- 没有调用对象
- 不能引用 **this/self**，不能引用类定义的普通数据成员（如 **Smalltalk** 里不能引用实例变量），只能引用本类的静态数据成员
- 通常采用某种基于定义类的语法形式调用

仅有静态数据成员和静态方法的类，相当于一个简单模块

- 提供模块的内部状态，可以通过所提供的方法修改状态
- 不能生成有用的实例（生成的是空实例，没有局部的实例状态）
- 静态数据成员的静态方法的封装，可能定义内部数据和操作

对象和继承：数据布局

继承关系的数据部分通过对象的适当存储布局实现

- 对象的实际表现就是数据成员的存储
- 假定 **B** 是一个类，有自己的数据成员
- **D**是**B**的派生类，增加了数据成员。**D**类对象的前部仍是**B**类的所有成员，扩充的成员排在后面
- 在**D**类对象里，所有**B**类成员相对于对象开始位置的偏移量与它们在一个**B**类对象里的偏移量相同。这样，**D**类对象就可以作为**B**类对象使用，**B**类里的方法能正确操作，它们只看属于**B**对象的那部分
- **D**类里的方法既可以使用对象中的**B**类数据成员，也可以使用对象里的**D**类数据成员

用**D**类对象给**B**类对象“赋值”（值 **copy**，或者值语义时）会产生“切割”现象，**D**类数据成员不能拷贝

B类的对象

B类的
数据成员

D类的对象

B类的
数据成员

D类新增的
数据成员

初始化和终结处理

对象可能具有任意复杂的内部结构

- 要求创建对象的程序段做对象初始化，需反复描述，很麻烦，易弄错
- 对象可能要求特殊的初始化方式和顺序，对象的使用者难以贯彻始终
- 继承使对象的初始化更复杂化，因为需要正确初始化继承来的数据成员
- 为更容易处理对象初始化的问题，**OO** 语言通常都提供了专门的机制，在对象创建时自动调用
- 初始化操作保证新创建对象具有合法的状态。自动调用非常有意义，可以避免未正确初始化造成的程序错误
- 现在常把对象初始化看作调用一个称为构造函数（**constructor**）的初始化子程序，它（们）在对象的存储块里构造出所需要的对象
- 语言通常支持参数化的初始化操作，以满足不同对象的需要。对象创建可能有多种需要，为此 **C++/Java** 等都支持一个类有多个不同的构造函数

初始化和终结处理

- 如果变量采用引用语义，所有（值）对象都需要显式创建，有明确的创建动作。这样很容易保证在存储分配之后调用构造函数
- 如果变量是值，为保证初始化，语言需要对变量创建提供特殊语义，要求变量创建包含隐式的构造函数调用
- 对象初始化必须按一定的顺序进行
 - 对象内部的基类部分必须在派生类部分之前完成初始化，因为派生类新增的数据成员完全可能依赖于基类成员的值
 - 数据成员本身也可能是某个类的对象，在执行整体对象的构造函数的过程中，就需要执行这些对象成员的构造函数
 - 这种构造规则是递归的，语言必须严格定义对象的构造顺序
- 如果变量采用值语义（例如 C++），在进入一个作用域的过程中，就可能出现许多构造函数调用
 - 进入作用域可能是代价很大的动作

初始化和终结处理

- 在对象销毁之前，可能需要做一些最后动作（终结处理），例如释放对象所占用的各种资源，维护有关状态等
- 忘记终结处理，就可能导致资源流失，或者状态破坏
- 有些 **OO** 语言提供终结动作定义机制，销毁对象前自动执行所定义动作
- **C++** 采用值语义，终结动作以类的析构函数的形式定义：
 - 类变量是堆栈上的对象，在其作用域退出时，自动调用它们的终结动作
 - 堆对象需要显式释放，释放之前恰好应该执行终结动作，易于处理
- 采用引用语义的语言（如 **Java**），通常并不提供销毁对象的显式操作（以防悬空引用），对象销毁由 **GC** 自动进行
 - 有了 **GC**，对终结动作的需求大大减少，终结动作由 **GC** 自动进行
 - 执行终结动作的时间不可预计，出现了（时间和顺序的）不确定性
 - 对象关联和 **GC** 顺序的不确定性使终结动作很难描述

类层次结构和强制转换

```
class foo { ...  
class bar : public foo { ...  
...  
foo F;  
bar B;  
foo* q;  
bar* s;  
...
```

```
q = &B;           // ok; references through q will use prefixes  
                  // of B's data space and vtable  
s = &F;           // static semantic error; F lacks the additional  
                  // data and vtable entries of a bar
```

- C++ 代码:

- 基类指针可以安全地引用派生类的对象，这时的（非变换）自动类型转换称为“向上强制”，**upcasting**

- 但子类指针不能引用基类对象

- 向上强制总是安全的，不会出问题，总可以自动进行。因为派生类包含基类所有数据成分，因此可以支持基类所有操作
- 后一个赋值是编译时错误，派生类指针不能引用基类对象

类层次结构和强制转换

- 如果用 **foo** 类的指针 **q** 传递一个对象，可保证该对象一定是 **foo** 的或它的某个派生类的对象
- 如果由 **foo** 类指针 **q** 传递的实际上是一个 **bar** 对象，我们有时需要把它作为 **bar** 对象使用，例如想对它调用 **foo** 里没有的方法
 - **q->s(..)** 是静态类型错误（**q** 的指向类型是 **foo**，**foo** 无方法 **s**）
 - **s = q** 也是静态类型错（不能保证 **q** 指向的是 **bar**，赋值不安全）

能不能用 **s = (bar*)q** ？

- 如果 **q** 指向的确实是一个 **bar** 对象，当前情况下恰好可以，因为
 - **(bar*)** 对指针是“非变换转换”，导致把 **foo** 指针当做 **bar** 指针
 - 恰好 **bar** 对象的起始位置和各成分的偏移量与 **foo** 一样

这些条件有时不成立（下面会看到，在存在多重继承时）

这种转换不安全，它要求 **q** 指向的确实是 **bar**。动态怎么检查类型？

类层次结构和强制转换

- C++ 为安全的向下强制转换提供了专门运算符 **dynamic_cast**。上述转换的正确写法：

```
bar *x = dynamic_cast<bar*>(q);
```

- 如果 **q** 指向的确实是 **bar** 类的对象，转换将成功，**x** 指向该 **bar** 类对象
- 如果 **q** 指向的不是 **bar** 类的对象，转换失败，**x** 被赋空指针值 **0**
- 通过检查 **x** 的值，可以判断转换是否成功

实现 **dynamic_cast**，就要求在运行中能判断对象的类型和类型间关系
这就是运行时类型识辨（**Run Time Type Identification, RTTI**）

要像支持安全的向下转换，C++ 的实现需要在 虚表里增加一个类描述符

- 常放在虚表最前。一些 C++ 编译器要求用户指明需要用 **RTTI**，在这种情况下才按这种方式创建虚表（虚表的形式与没有类描述符时不同）
- **dynamic_cast** 检查类型关系，确定能否转换，在能转换就完成转换

类层次结构和强制转换

多数 **OO** 语言（如 **Java** 等）默认支持 **RTTI**，虚表里总保存类描述符

- 如何描述类型是编译器的具体实现问题，不必关心
- **RTTI** 机制可保证类型安全的转换

虽然 **Java** 的类型转换采用 **C** 语言类型转换的描述形式，但功能不同

- 在牵涉到基本类型时，可能需要做值的转换
- 在牵涉到类类型时，需要做动态的类型转换合法性检查
 - 如果发现错误，就抛出异常 **ClassCastException**
 - 否则做“非变换类型转换”，把相应引用直接当作所需的类型的引用
- 从基本类型值到类类型的合法转换，还需要自动构造对象（**boxing**）；从类对象到基本类型值的转换需要提取对象内的值（**unboxing**）

运行时类型描述机制还被用于支持“自反”（**reflection**）功能

静态和动态束定的方法

- **OO** 语言里的方法调用通常采用 **x.m(...)** 的形式，其中
 - **x** 是一个指向或者引用对象的变量
 - **m** 是 **x** 的定义类型（类，假定为 **B**）的一个方法
- 问题：**x.m(...)** 所调用的方法何时/根据什么确定？两种可能性：
- 根据变量 **x** 的类型（在程序里静态定义）确定（静态束定）
- 根据方法调用时（被 **x** 引用/指向）的当前对象的类型确定（动态束定）
 - 由于 **x** 可能引用（指向）**B** 类或其任何子类的对象，因此同为这个方法调用，不同的执行中实际调用的完全可能是不同的方法

所有 **OO** 语言都支持动态方法束定（否则就不是 **OO** 语言），多数以它作为默认方式。少数语言同时也支持静态束定的方法，如 **C++**、**Ada** 等

- **C++** 把动态束定的方法称为虚方法（**virtual** 方法），而且以静态束定作为默认方式。这种设计与它的 **C** 基础有关

静态束定的实现

- 调用静态束定的方法，实现方式就像是调用普通子程序（过程/函数），唯一不同之处就是需要传入一个指向调用对象的指针
- 在符号表里，每个类的记录项都包含了一个基类索引，依靠这个索引形成的基类链就可以静态（编译时）完成静态束定的方法的查找：
 - 1. 首先在变量所属的类（静态已知）里查找（查找符号表）。如果在这里找到了所需要的方法，就生成对它的调用；如果不存在就反复做下一步
 - 2. 转到当前类的基类里去查找相应方法，如果找到就生成对它的调用；如果找不到就继续沿着基类链上溯查找
 - 3. 如果已无上层基类，查找失败。报告“调用了无定义的方法”错误
- 所有对静态束定的方法的调用都可以静态（编译时，一次）处理
 - 运行时的动作与一般子程序调用完全一样，没有任何额外运行开销
 - 如果语言允许静态束定的方法，采用静态束定可以提高效率。静态束定的方法还可以做 **inline** 处理

多态问题

- **polymorphism**: 对于给定的方法，获得**不同**但可**归一化的**对象行为设计与实现机制
 - 通过overload机制而获得的可归一化方法（重载）
 - 通过override机制而获得的可归一化方法（重写）
- 多态机制为表达复杂设计逻辑提供了简化手段
 - **Overload**: 区分不同场景来重载方法，避免一个方法处理多种场景而导致其逻辑复杂
 - **Override**: 让每个类根据其功能要求来重写所需要的方法，使用者无需区别对待通过统一方式即可获得相应的功能

基于重载的静态多态

- 从设计角度看，一个方法规范了类的一种能力
- 随着功能的扩展或演化，这种能力也需要演化来处理多种不同的场景，即处理不同的输入情况
- 这时需要对原方法进行overload扩展，从而使得相应能力得到演化
 - 求和(add)是一个类math的方法，`int add(int, int)`
 - 现在希望扩展该方法的能力，能够支持整数、实数的综合加法：`int add(int, double)`, `int add(double, double)`
 - 更进一步扩展以支持对不定长集合元素求和：`int add(int[])`, `int add(int[], int[])`
- 重载发生在一个类内部，编译时解析方法的多态性（静态）

方法的动态绑定

```
class B {  
    ... ..  
    T0 tem(...) { ... sp(...) ... }  
    virtual T1 sp(...) { ... .. }  
}  
  
void fun(B &x) { ... x.tem(...) ... }  
  
class D : public B {  
    T1 sp(...) { ... .. }  
}  
  
B b;  
D d;  
  
fun(b) ;  
fun(d) ;
```

- **B** 类里定义了一个一般性操作 **tem**，对所有 **B**类对象都可使用
- **tem** 中调用了一个特殊操作 **sp**，该操作可能因子类不同而不同
- 子类 **D** 覆盖操作 **sp**后，仍能正常地调用操作**tem**，而且其中对 **sp** 的调用能调用到 **D** 类里新的操作定义

这是 **OO** 程序设计里最重要的东西：

这一特征使新类给出的行为扩充（或修改）可以自然地融合到已有功能里，包括放入已有的操作框架里。

动态束定的实现：一般模型

- 对最一般的对象模型，运行中调用动态束定的方法时要做一次与编译时处理静态束定方法一样的查找，这种查找可能非常耗时
- 为完成这种方法查找：
 - 每个类需要有一个运行时表示（把类也作为程序对象），类表示中需要有一个成分是基类索引，还有一个成分是方法表
 - 每个对象里必须保存所属类的信息（一个类指针，指向其类）
 - 每个动态方法调用都启动一次方法查找。如果找到就调用，找不到就发出一个“**message is not understood**”（**Smalltalk**）动态错误
- 这种方式普遍有效，可以处理具有任何动态性质的对象模型，如动态类层次结构构造和动态方法更新（修改、添加和删除）、动态类属关系等
 - 查找的时间开销依赖于继承链的长度和继承链上各个类中方法的个数
 - 这种方法的缺点是效率太低。如果所采用的对象模型在动态特性方面有所限制，就可能开发出效率更高的方法

动态束定的实现：受限模型

- 早期 **OO** 语言（包括 **Smalltalk**）都采用功能强大灵活的对象模型
 - 在提供了极大灵活性的同时，也带来效率上的巨大开销
 - 这也是早期 **OO** 语言及其概念难被实际软件工作者接受的最关键原因
- 提高算法效率的最基本途径是限制要解决的问题（对更特殊一些的问题，可能找到效率更高的算法），并设计优化的实现模型
- 对于 **OO** 语言，就是要找到一个受限的对象模型，它能比较高效地实现，同时又能满足绝大部分实际 **OO** 程序开发的需要
- 常规 **OO** 语言中的对象模型有如下特性（足以支持常见程序设计工作）：
 - 类层次结构是静态确定的
 - 每个类里的动态束定方法的个数和顺序都静态确定
- 在这种模型里就可以避免动态方法查找，使方法调用的执行效率接近普通的子程序调用的执行效率（**C++** 和 **Stroustrup** 的贡献）

➤ 强类型语言的动态多态问题

C++的无法执行的多态函数

设父类ellipse子类circle均有求周长、求面积
area()函数，有以下主程序：

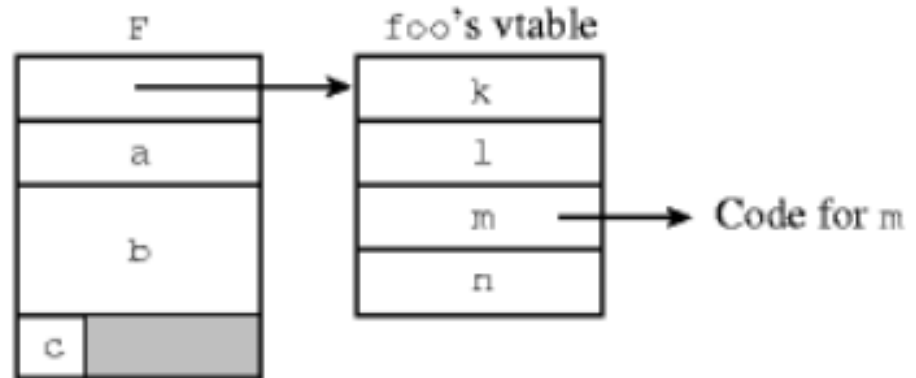
```
main( )  
{  
    ellipse *pe, e(8.0, 4.0);  
    circle c (5.0);  
    pe=& c;  
    cout <<pe -> area( );  
}
```

```
virtual float area( );
```

动态束定的实现

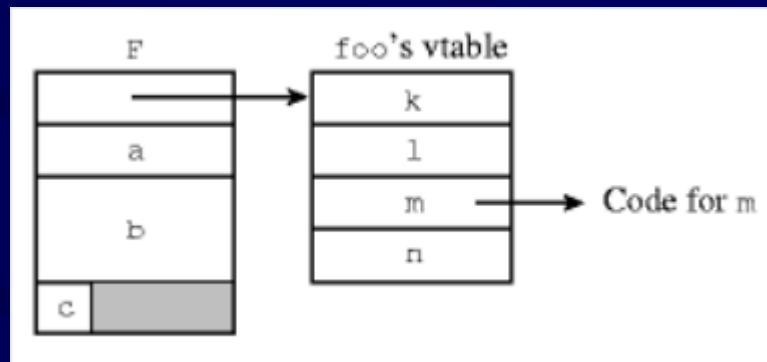
- 优化实现模型，其中绝大部分工作都能静态完成：
- 每个类的运行时体现是一个动态束定方法的表（称为虚表，**vtable**），这是一个指针表，指针指向本类的对象需要调用的方法的代码体
- 虚表的指针按方法在类里的顺序排列，每个方法对应于一个顺序下标

```
class foo {  
    int a;  
    double b;  
    char c;  
public:  
    virtual void k ( ...  
    virtual int l ( ...  
    virtual void m ();  
    virtual double n( ...  
    ...  
} F;
```



动态束定的实现

- 在每个对象开头（数据域之前）增加一个指针 **vt**
- 创建对象时，设置其 **vt** 指向其所属的类的虚表（运行中始终不变）
- **f** 是指向 **F** 的指针（或引用）
调用 **f->m(...)**的实现



to call **f->m()**:

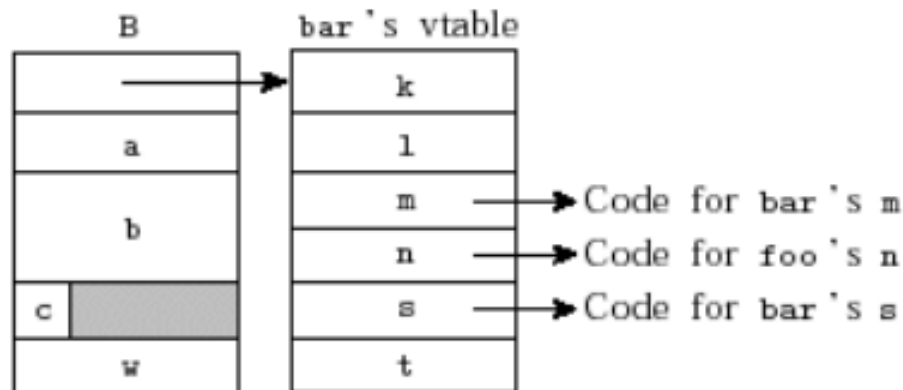
```
r1 := f
r2 := *r1                -- vtable address
r2 := *(r2 + (3-1) × 4)  -- assuming 4 = sizeof(address)
call *r2
```

- 比调用静态束定的方法多了中间的两条指令，它们都需要访问内存

动态束定的实现

- 虚表的创建:
- 如果类 **B** 没有基类, 就将其定义里的动态束定方法的代码体指针依次填入它的虚表 (下标从 **0** 或者 **1** 开始算)
- 若类 **D** 的基类是 **B**, 建立 **D** 的虚表时先复制 **B** 的虚表。如 **D** 覆盖了 **B** 的某个 (某些) 动态束定方法, 就用新方法的指针覆盖虚表里对应的已有指针。若 **D** 定义了新的动态束定方法, 就将它们顺次加入虚表, 放在后面

```
class bar : public foo {  
    int w;  
public:  
    void m (); //override  
    virtual double s ( ...  
    virtual char *t ( ...  
    ...  
} B;
```



- 如果 **f** 指向的对象是 **B**, 那么 **f->m(...)** 也会调用正确的方法

若前此已设计card, pay_data类, 部分程序是:

```
class employee {
    char name [ ], soc_sec [13], * dept_code, * job_code
public:
    employee * Link;
    employee (int); //构造函数; 变元是名字个数
    void print_paycheck;
    virtual pay_data compute_pay;
    virtual void print_list;
    void employee :: print_empl ( )
    { cout << "Name: " << name << "\n\t" << soc_sec
        << "\t Dept: " << dept_code << "\t Job: "
        << job_code << "\n";
    }
};

class salaried : public employee {
    int annual_salary, vocation_used;
public:
    pay_data comput_pay( );
    void take_vacation( );
};
```

```
class manager : salaried {
    employee * staff; public:
    void add_employee( );
    void print_list( );
    void do_payroll( );
    manager( );           //构造函数
};

class hourly : public employee {
    float pay_rate, hours_worked, overtime;
    int vacation_used;
public:
    pay_data compute_pay( );
    void record_time_card ( card *);
    hourly ( );           //构造函数
}

class officer : salaried { ...};
class professional : salaried {...};
```


其中employee :: print_list()定义为虚函数,
manager :: print_list()也是虚函数。它们的各自定义是:

```
void employee :: print_list( ) {  
    employee * scan;  
    for (scan = link; scan != NULL; scan = scan → link)  
}  
void manager :: print_list( ) {  
    cout << "\n\n Manager: ";  
    print_empl ( );  
    cout << "\n Employees Supervised: \n";  
    employee::print_list( );  
}
```

例子分析

```
public class Foo {
    public void method1() {
        System.out.println("foo 1");
    }

    public void method2() {
        System.out.println("foo 2");
    }

    public String toString() {
        return "foo";
    }
}

public class Bar extends Foo {
    public void method2() {
        System.out.println("bar 2");
    }
}
```

```
Foo[] pity = {new Baz(), new
    Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length;
    i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

```
public class Baz extends Foo {
    public void method1() {
        System.out.println("baz
1");
    }

    public String toString() {
        return "baz";
    }
}

public class Mumble extends Baz
{
    public void method2() {
        System.out.println("mumble
2");
    }
}
```

用5分钟时间给出执行结果，注意先梳理类的层次关系和方法调用

重写方法执行分析表

method	Foo	Bar	Baz	Mumble
method1				
method2				
toString				

重写方法执行分析表

method	Foo	Bar	Baz	Mumble
method1	foo 1		baz 1	
method2	foo 2	bar 2		mumble 2
toString	foo		baz	

重写方法执行分析表

	Foo	Bar	Baz	Mumble
method1	foo 1	<i>foo 1</i>	baz 1	<i>baz 1</i>
method2	foo 2	bar 2	<i>foo 2</i>	mumble 2
toString	foo	<i>foo</i>	baz	<i>baz</i>

- 一个方法会调用另一个方法，且混杂着方法的重写

```
Ham[] food = {new Lamb(), new Ham(), new Spam(), new Yam()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    System.out.println();
    food[i].b();
    System.out.println();
    System.out.println();
}
```

```
public class Ham {
    public void a() {
        System.out.print("Ham a ");
        b();
    }
    public void b() {
        System.out.print("Ham b ");
    }
    public String toString() {
        return "Ham";
    }
}

public class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b ");
    }
}
```

```
public class Yam extends Lamb {
    public void a() {
        System.out.print("Yam a ");
        super.a();
    }
    public String toString() {
        return "Yam";
    }
}

public class Spam extends Yam {
    public void b() {
        System.out.print("Spam b");
    }
}
```

更复杂的例子

- Lamb继承了Ham的方法a，a调用b，但是Lamb重写了方法b...

```
public class Ham {  
    public void a() {  
        System.out.print("Ham a    ");  
        b();  
    }  
  
    public void b() {  
        System.out.print("Ham b    ");  
    }  
  
    public String toString() {  
        return "Ham";  
    }  
}  
  
public class Lamb extends Ham {  
    public void b() {  
        System.out.print("Lamb b    ");  
    }  
}
```

动态束定的实现

重温受限的对象模型（对一般程序设计已经足够强大）：

- 类层次结构是静态确定的
- 每个类里的动态束定方法的个数和顺序都静态确定

优化实现的效果：

- 构造方法表的工作在编译时完成
 - 每个对象里需要增加一个指向其类的方法表的指针
 - 每次方法调用需要多执行两条指令（典型情况），多访问内存两次
- 对这种受限对象模型，动态束定方法调用的额外开销不大，一般软件系统（包括系统软件的绝大部分情况）都可以接受
- Stroustrup** 在设计和实现 C++ 语言时特别希望能够得到高效的实现，最后选择了这种对象模型，并设计了这种高效的实现方法

动态束定的实现

- 对数据抽象和面向对象技术的支持，以及高效的实现，使实际软件工作者看到了 **C++**（和 **OO**）的潜力，最终导致了面向对象的革命
- 以后的主流面向对象语言也都采用了这种技术。当然，采用这种选择，也就对它们可能采用的对象模型提出了严格的限制
- **Pragmatics** 里还讨论了其他高效实现方法，《**C++ 语言的设计与演化**》里也有讨论（通过几条指令构成的一段“蹦床代码”，将控制转到实际应该调用的方法，主要是要解决多重继承问题）
- 虚方法（动态束定方法）的一个重要缺点是不能做 **inline** 处理（在线展开要求静态确定被调用的方法），使编译器难以进行跨过程的代码优化

C++ 希望支持高效的系统程序设计，认为虚方法带来的效率损失有时也是不能容忍的，因此它同时支持静态方法束定

注意：如果一个类里只有静态束定的方法，该类编译之后就不会生成方法表，该类的对象也没有一个指针的额外存储开销

接口

- 有些类虽然没有层次关系，但可以有共性行为
 - Circle, Rectangle和Triangle
 - 它们需要的共性行为包括
 - 计算几何形状的外周长(perimeter)
 - 计算几何形状的面积(area)
- 对于不同的几何类型，这些共性行为的内涵解释(即计算方式)可能都不同
 - 使用接口，而不是公共父类(因为没什么具体计算行为可复用)

接口

- Rectangle (宽 w , 高 h):

$$\text{area} = w h$$

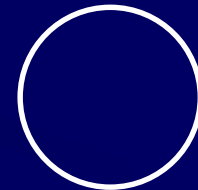
$$\text{perimeter} = 2w + 2h$$



- Circle (半径 r):

$$\text{area} = \pi r^2$$

$$\text{perimeter} = 2 \pi r$$

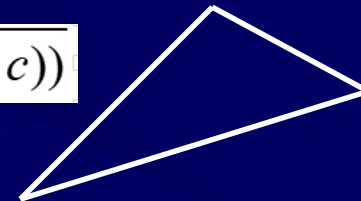


- Triangle (边长 a, b, c)

$$\text{area} = \sqrt{(s(s-a)(s-b)(s-c))}$$

其中 $s = \frac{1}{2}(a + b + c)$

$$\text{perimeter} = a + b + c$$



接口

- 为不同几何类型实现相应的方法 `perimeter` 和 `area`.
- 目标：客户代码无需区分不同的几何类型
 - 直接获得任意几何形状的面积和周长
 - 能够创建数组来管理各种可能的几何对象
 - 能够在屏幕上画出几何形状
- **interface**: 对一组类共性行为的抽取结果，使得设计规格和实现相分离
 - 继承是一种层次抽象和代码复用机制
 - 接口是一种行为层次抽象，无关代码复用

接口

```
public interface name {  
    public type name(type name, ..., type name);  
    public type name(type name, ..., type name);  
    ...  
}
```

Example:

```
public interface Vehicle {  
    public double speed();  
    public void setDirection(int  
direction);  
}
```

- 接口中的方法是抽象方法(**abstract method**), 即不提供实现的方法
 - 允许/要求相应的类来实现相应的接口

接口的实现

```
public class name implements interface {  
    ...  
}
```

– Example:

```
public class Bicycle implements  
Vehicle {  
    ...  
}
```

- 如果一个类实现(*implements*)一个接口
 - 这个类必须包括接口中规定的所有方法，否则无法通过编译

接口及其实现举例

```
// Represents circles.
```

```
public class Circle implements ClosedShape {  
    private double radius;
```

```
// Constructs a new circle with the given radius.
```

```
public Circle(double radius) {  
    this.radius = radius;  
}
```

```
// Returns the area of this circle.
```

```
public double area() {  
    return Math.PI * radius * radius;  
}
```

```
// Returns the perimeter of this circle.
```

```
public double perimeter() {  
    return 2.0 * Math.PI * radius;  
}
```

```
}
```

```
public interface ClosedShape {  
    public double area();  
    public double perimeter();  
}
```

4.4 Ada 的面向对象机制

• 定义类和实例对象

Ada95以抽象数据类型实现类。类的封装性由包实现，类的继承性则扩充了标签（tag）类型和抽象类型，标签类型只限记录类型。类的继承性利用了Ada8的类型派生机制实现子类。

```
package Object is
  type Object is tagged    --此类型的数据即对象的属性
    record                  --无tagged即一般的ADT，有它是为了类继承
      X_Coord: Float: =0;
      Y_Coord: Float: =0 --初值为缺省时用
    end record;
  function Distance (O: Object) return Float; --Object对象的行为
  function Area (O: Object) return Float;
end Object
```



```
with Object, use Object;
package Shapes is
    type Point is new Object with null record;
    type Circle is new Object with
        record
            Radius: Float;
        end record;
    function Area (C: Circle) return Float;
    type Triangle is new Object with
        record
            A, B, C: Float;
        end record;
    function Area (T: Triangle) return Float;
end Shape;
```

--这个包封装了三个子类（型）
--只继承不扩充的子类
--继承并扩充此属性
--覆盖Object中的Area
--继承并
--扩充三个属性
--覆盖

这些类（型）包外可见（可输出），在主子程序中声明实例，如同类型声明变量，以初值表达式作值构造子：

子类的实例

```
P: Point;    --声明实例对象  
C: Circle: = (0.0, 0.0, 34.7);  
T: Triangle: = (3.0, 4.0, 5.0);
```

也是父类的实例

```
O: Object:  
P: (O with null record);  
C: = (O with 34.7);  
T: = (O with 3.0, 4.0, 5.0);
```

如果动态生成实例，可将此声明放在类的方法（过程/函数）中，调用时生成。

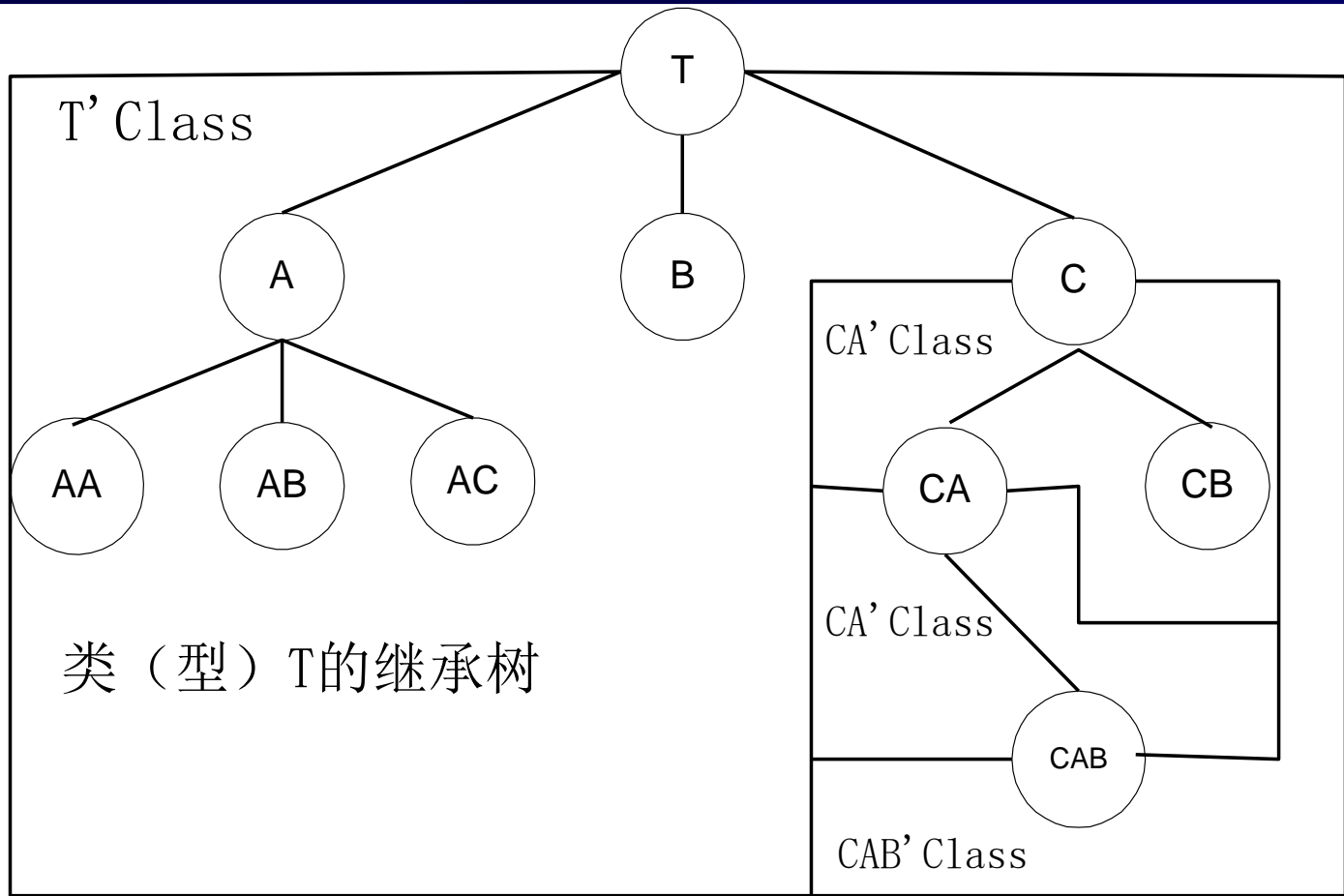
• 以类宽类型实现多态

Ada95的每个标签类型都有一个与之对应的类类型属性T' Class, 并把它叫做类宽类型 (Class Wide Type)

设已声明T类型, 及T' Class的变量V, 则

```
Y: T;           --一般声明, 正确
Y: T' Class;    --不可以
Y: T' Class: =V; --可以, T' Class束定为V的类型
```

类宽类型的范围示意如下:



- 民航订票系统：乘客向售票员提出订票要求，订票系统处理这个要求，并通知售票员能否满足这个要求。有3种座位可供选择：经济舱、公务舱、头等舱。

```
package Reservation-System is
  type Position is (Aisle, Window) ;
  type Meal_Type is (Fish, Chicken, Beef, Pork) ;
  type Reservation is tagged
    ...
  procedure Make (R: in out Reservation) ;
  procedure Select_Seat (R: in out Reservation) ;
  type Economic_Reservation is new Restrvation with .....
  .....
```

```
procedure Process_Reservation
  (Rc: in out Reservation'Class) is
    --形参可以是类宽类型，不必最初束定某特定类型
begin
    ...
    Make (Rc) ;    --它按相结合的Rc的具体类型出票
    ...
end Process_Reservation;
```

• 扩充程序包机制实现继承的类体系

Ada 95增设了子辈单元 (child unit) 和私有子辈单元。

子辈单元

```
package Reservation_System.Supersonic is    -- ``后是子辈单元名
    type Supersonic_Reservation is new Reservation with private;
private
    type Supersonic_Reservation is new Reservation with
        record
            Champagne: Vintage;
            .....
        end record;
procedure Make (Sr: in out Supersonic_Reservation) ;
procedure Select_Seat (Sr: in out Supersonic_Reservation) ;
    .....
end Reservation_System.Supersonic;
```

•私有子辈单元

package OS is

--父包OS

--OS 的可见成份

```
type File Descriptor is private;
```

...

private

```
type File Descriptor is new Integer;
```

```
end OS;
```

```
package OS.Exceptions is
```

--OS的子辈程序包

File Descriptor Error,

File Name Error,

Permission Error: **exception;**

--所定义异常OS各子辈包

均可用

```
end OS.Exceptions;
```

—公有，但不涉及泄露

```
with OS.Exceptions;
```

```

with OS.Exceptions;
package OS.File_Manager is      --OS的子辈程序包
    type File_Mode is (Read_Only, Write_Only, Read_Write);
    function Open (File_Name: String; Mode: File_Mode) return
                                                File_Descriptor;
    procedure Close (File: in File_Descriptor);
    ...
end OS.File_Manager;           --公有，只用私有类型。也无泄露
with OS.Exceptions;
procedure OS.Interpret (Command: String);
                                --命令解释过程, 等同子包
private package OS.Internals is
                                --私有子辈程序包，不用with
    ...
end OS.Internals;
private package OS.Internals_Debug is
                                --OS的私有子辈程序包
    ...
end OS.Internals_Debug;

```


Ada的多继承

```
with Abstract_Sets;
package Linked_Sets is
type Linked_Set is new Abstract_Sets with private;
--再定义Linked_Set的各种操作

private
type Cell;
type Cell_Ptr is access Cell;
type Cell is
    record
        E: Element;
        next: Cell_Ptr;
    end record
function Copy (P: Cell_Ptr) return Cell_Ptr;
type Inner is new Controlled with
    record
        The_Set: Cell_Ptr;
    end record;
procedure Adjuse (Obj : in out Inner);
type Linked_Set is new Abstract_Sets with --继承Abstraet_sets
    record
        Component: Inner;
    end record;
end Linked_Sets;
```

--其扩展成分又继承了Controlled

4.4 Ada 95语言的面向对象特征

Ada 95中的类是一种被称为标志类型（tagged type）的新类型，它们可以是记录类型也可以是私有类型。它们被定义在包之中，这就允许它们被分别编译。

Ada 95中不具有构造器或析构子程序的隐式调用。尽管也可以编写这样的子程序，但必须由程序人员显示地调用。

4.4 Ada 95语言的面向对象特征

封装：

Ada中的封装结构被称为包（package），包可以有两个部分，每个部分也被称为包：其一为说明包，它提供封装的接口；另外一个为体包，提供在说明中所命名的实体的实现。

定义一种数据类型时，可选择是使得这种类型对于客户完全可见，还是仅仅提供接口信息。

4.4 Ada 95语言的面向对象特征

继承：

- Ada 95中的派生类是以标志类型为基础的。通过包括一种记录定义，可以将新的实体增加到被继承的实体之中。这种继承机制没有办法阻止父类的实体被包括在派生类之中。其后果是，派生类只能扩充父类，称为子类型；
- 要派生一个不包括所有父类的实体的类，需要使用子库包。子库包是一个使用父库包名字作为其名字的前缀的包，也可以将子库包用于C++中友元定义的位置；
- Ada 95 不提供多继承。

4.4 Ada 95语言的面向对象特征

动态绑定:

- Ada 95在标志类型中提供了过程调用与函数定义的静态绑定与动态绑定。每一个标志类型都隐含地有一个类范围标志。对于标志类型T，这种类范围类型被使用T'class 来说明；
- 通过在类型定义和子程序定义中使用保留字abstract，可以定义纯抽象基类型。

4. Ada 95 语言的面向对象特征

子程序包：

- 可以将一个程序包直接嵌套在其他的程序包中，这时就称之为子程序包（**child package**）；
- 允许子程序包成为单独的单位，并且可以被分开编译，可以防止整个嵌套程序包过大；
- 子程序包可以是公有的（默认）或私用的。

4.5 C++语言的面向对象特征

C++的一个主要设计考虑是与C语言向后兼容。这使得C++成为了一种混合式语言，它既支持过程式程序设计，也支持面向对象程序设计。

4.5 C++语言的面向对象特征

封装:

– (1) 访问控制机制:

– C++提供完善的访问控制机制，分别是： `public`, `protected`和`private`。

表1 `private`, `public`, `protected` 访问标号的访问范围

private, public, protected 访问标号的访问范围		
public	可访问	1.该类中的函数； 2.子类的函数； 3.其友元函数访问； 4.该类的对象访问；
	不可访问	
protected	可访问	1.该类中的函数； 2.子类的函数； 3.其友元函数访问；
	不可访问	1.该类的对象访问；
private	可访问	1.该类中的函数； 2.其友元函数访问；
	不可访问	1.子类的函数； 2.该类的对象访问；

4.5 C++语言的面向对象特征

封装：

- (2) 对象的独立性：
 - C++中对象本身不具有独立性，也就是对象必须属于某一个类；
 - C++中类本身不是对象，对象是类的实例化；
 - C++的对象可以是静态的、栈动态的和堆动态的。

4.5 C++语言的面向对象特征

继承:

- 是基于继承的模型;
- 采用任意的类层次结构;
- C++的一个类可以是独立的, 它不具有超类;
- 允许单继承、多继承、多重继承和虚继承;
- 采用三种继承方式, 分别是public继承, protected继承, private继承;

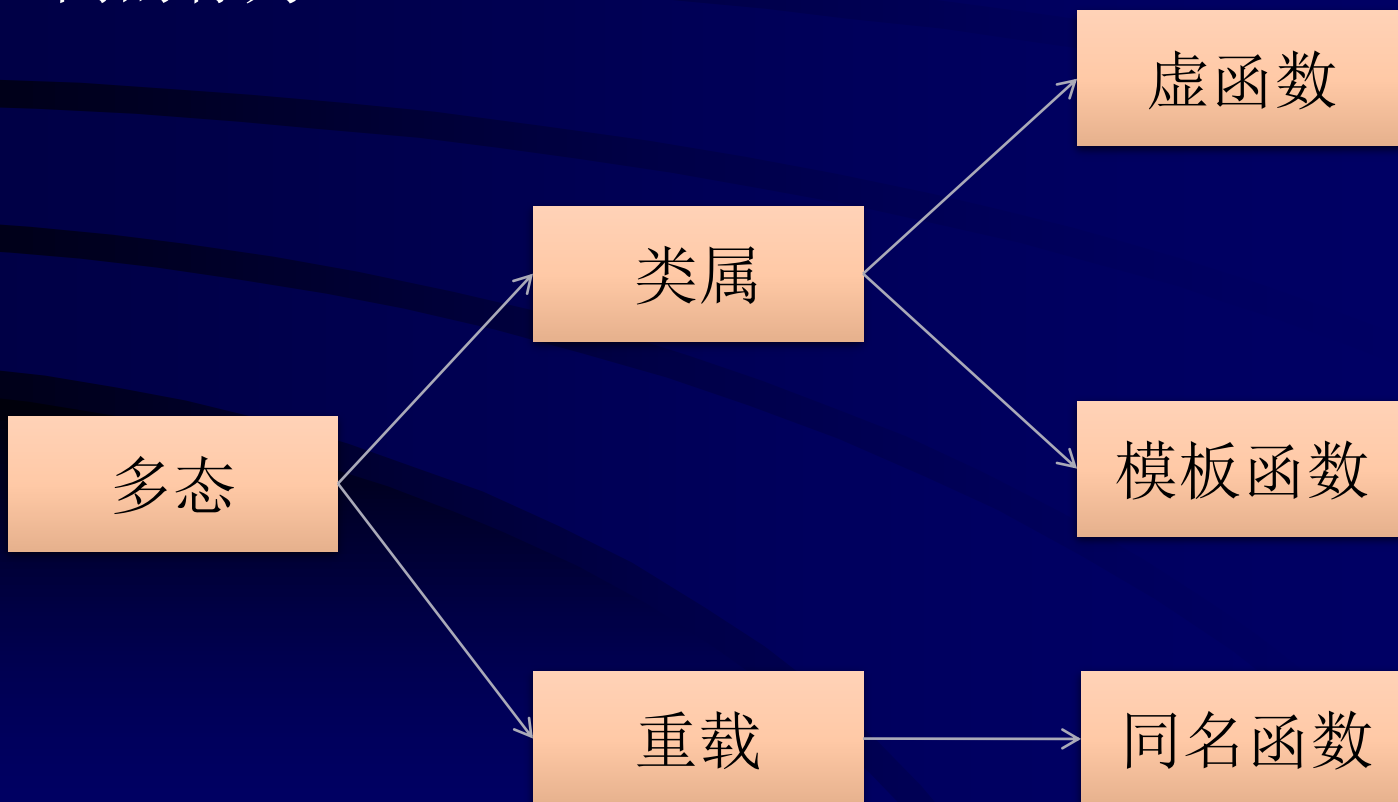
表2 类的继承方式

	public	protected	private
public继承	public	protected	不可用
protected继承	protected	protected	不可用
private继承	private	private	不可用

4.5 C++语言的面向对象特征

多态:

- 多态: 是指同样的消息被不同类型的对象接收时导致不同的行为。



4.5 C++语言的面向对象特征

多态:

(1) 类属：虚函数

用父类的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数，这种技术可以让父类的指针有“多种形态”；

(2) 类属：模板函数

模板是C++支持参数化多态的工具，使用模板可以使用户为类或者函数声明一种一般模式，使得类中的某些数据成员或者成员函数的参数、返回值取得任意类型。

4.5 C++语言的面向对象特征

多态:

(3) 重载: 同名函数

有两个或多个函数名相同的函数，但是函数的形参列表不同。在调用相同函数名的函数时，根据形参列表确定到底该调用哪一个函数。

4.6 Java语言的面向对象特征

在java中，只有原始数据类型（布尔类型、字符类型以及数值类型）的值不是对象。java中所有类必须是根类object的子类，或者是object的后代类的子类。

所有的java对象都是显式堆动态的。大多数是用new操作符来分配，但是没有显式解除分配操作符。垃圾收集被用于存储空间的回收。当垃圾收集器要收集对象占用的存储空间时，finalize方法被隐式调用。

4.6 Java语言的面向对象特征

封装:

- 访问控制机制:

- Java提供完善的访问控制机制，分别是：public，protected、friendly和private。

表3 Java类中成员修饰符的访问权限

Java类中成员修饰符的访问权限				
	public	protected	default	private
本类	√	√	√	√
本包	√	√	√	×
子类	√	√	×	×
其他	√	×	×	×

4.6 Java语言的面向对象特征

封装：

- Java是基于类的模型；
- 类本身不是对象，对象是类的实例化；
- 对象本身不具有独立性，也就是对象必须属于某一个类。

4.6 Java语言的面向对象特征

继承：

- Java采用单根的分类层次结构。当创建一个类时，总是在继承，如果没有明确指出要继承的类，就总是隐式地从根类Object进行继承。
- 在java中，一个方法可以被定义为final，意味着该方法不能够被覆盖于任何后裔类中。当使用final来对一个类的定义进行说明时，意味着该类不能够是任何子类的父类。

4.6 Java语言的面向对象特征

继承：

- Java包括了一种称为**接口**的虚拟类，它提供了多继承的版本。接口的定义与类的定义相似，只是接口仅仅能够包括命名常数以及方法声明，不包含构造器或非抽象方法。
- 接口并不是多继承的替代。因为多继承提供代码重用，而接口却不能提供，这是它们间重大的不同处，代码重用是继承最大的优点。

4.6 Java语言的面向对象特征

多态:

- 多态: 是指同样的消息被不同类型的对象接收时导致不同的行为。



4.6 Java语言的面向对象特征

多态:

- 方法覆盖实现多态性：通过子类对父类的重定义来实现。方法的参数个数，类型，顺序要完全相同。
- 重载实现多态性：通过在一个类中定义多个同名的方法来实现。方法的参数个数，类型，顺序要有所不同。
- 在java中，一个方法被定义为final，或是static，或private，此时使用静态绑定，该方法不能被覆盖。除此之外，java所有的方法调用都是动态绑定的。

4.7 C#语言的面向对象特征

C#对于面向对象程序设计的支持与java中的相类似。C#包括了类和结构(struct)，类与java中的类十分类似，其中的struct为能力稍差的栈动态结构。

4.7 C#语言的面向对象特征

封装：

- 访问控制机制：

访问修饰符有public、private、protected、internal和protected internal。它们是修饰在类型（类、接口、委托、结构和枚举）和类型成员（字段、属性、方法、构造函数等等）上控制其访问权限的关键字。

4.7 C#语言的面向对象特征

继承：

- C#使用C++的文法来定义类。
- 从父类中继承的方法，可以在派生的子类中被替代，需要在子类的替代方法前加new。对于一般的访问，带有new的方法将父类中相同名字的方法隐藏起来。通过在方法名称前加前缀base，可以继续访问父类中的方法。
- 与java相同，C#支持接口。

4.7 C#语言的面向对象特征

多态:

- C#中，要将方法调用动态的绑定于方法，必须将基方法（即原定义的方法）和它在派生类中的方法都给以标记。将基方法标上`virtual`，派生类中的对应方法都标上`override`。
- C#中包括类似于C++中的抽象方法，但使用不同的文法来说明。
- 与java相同，C#中的类都从根类Object派生而来。根类Object定义了一组方法，包括ToString, Finalize和Equals.

4.8 Ruby语言的面向对象特征

Ruby是一种纯面向对象程序设计语言，它对面向对象程序设计的支持是彻底的。它虚拟化所有一切为对象，并且通过消息传递来完成所有计算。

Ruby的所有变量都是对对象的引用并且都是无类型的，所有实例变量名都以标记@开头。

4.8 Ruby 语言的面向对象特征

封装：

- Ruby的访问控制在访问数据和访问方法上不同。
- 所有实例数据默认下都是私有访问的，并且不能改变。
- 若需要对实例变量进行外部访问，必须定义访问方法。创建获取方法和设置方法的函数都是指定的，因为它们为类的对象提供了协议（在Ruby中称为属性），因此类的属性是对类对象的数据接口（共有数据）；

Ruby方法的访问控制是动态的，所以访问违例只能在执行时检测到。默认的方法访问是公有的，也可设计成保护的或私有的。

4.8 Ruby 语言的面向对象特征

继承：

Ruby 也提供了子类化的概念，子类化即继承；

Ruby类作为命名空间封装，它还有一个额外的命名封装，称为模组（module），模组定义方法和常量的集合。包含模组的效果是使类获得了指向模组的指针，并且有效地继承了模组中定义的函数。当类包含了模组时，模组变成了类的一个代理超类，这种模组称为混入（mixin）。混入提供了多继承的优点，而且不会出现如果模组在它们函数中不需要模组名时可能出现的命令冲突。

4.8 Ruby 语言的面向对象特征

多态:

Ruby 变量不能类型化，它们都是对任何类的对象的引用。因此，所有变量都是多态的，所有对方法调用绑定都是动态的。

4.9 JavaScript语言的面向对象特征

Java 是一种静态类型的面向对象语言，JavaScript 是动态类型的并且不是面向对象的。JavaScript 没有类，不能够支持继承，它的对象既是对象又是对象模型。

JavaScript 具有两类变量，引用对象的变量和直接存储原始类型值的变量。变量可以被声明，但声明不蕴涵类型。当每次给一个变量赋值时都能够改变它的类型，而新的类型就是所赋的值的类型。

4.9 JavaScript语言的面向对象特征

4.9 JavaScript语言的面向对象特征

4.10 Python语言的面向对象特征

1. 对象模型：封装

– (1)访问控制机制：

- Python提供的访问控制机制，分别是： public和 private；

Python中，如果函数、类方法和属性如果以两个下划线开头，但是不以两个下划线结束，它就是private的，其他的一切都是public的。

```
class people:
    __name = 'jack'           #private
    age=12                    #public
    __school__='buaa'        #public
    #定义了一个方法
    def printName(self):
        print(self.name)
```


4.10 Python语言的面向对象特征

1. 对象模型：封装

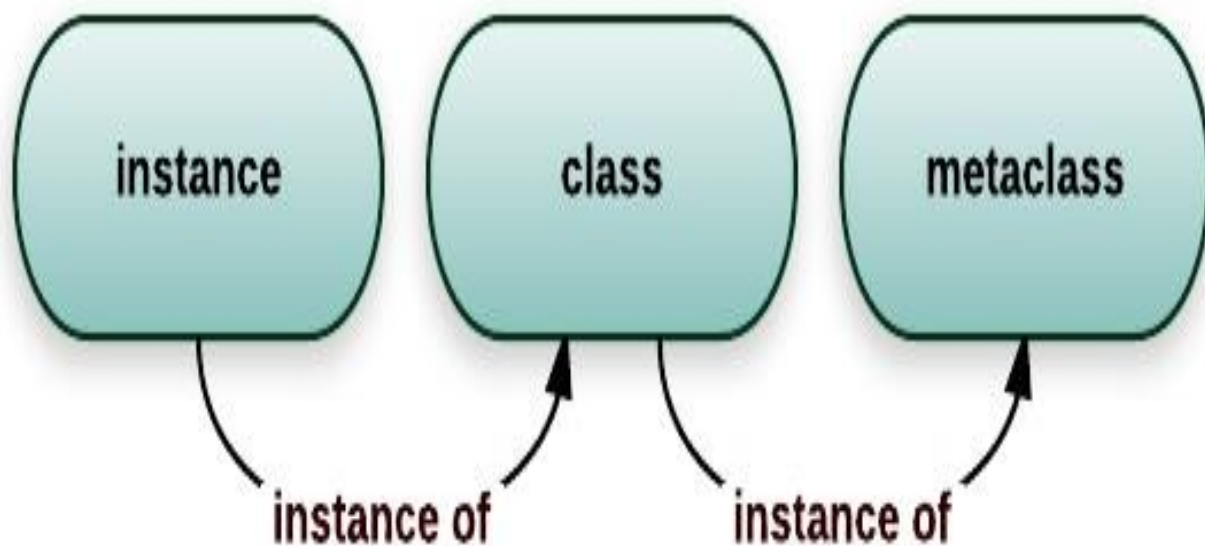
- (2) 对象的独立性：
 - Python中对象本身不具有独立性，也就是对象一定属于某一个类；

4.10 Python语言的面向对象特征

1. 对象模型：封装

- (3) 类本身是不是对象？
 - Python中类本身是(元类的)对象；
- (4) 基于类的模型，还是基于对象或原型的模型？
 - Python是基于对象的模型；

4.10 Python语言的面向对象特征

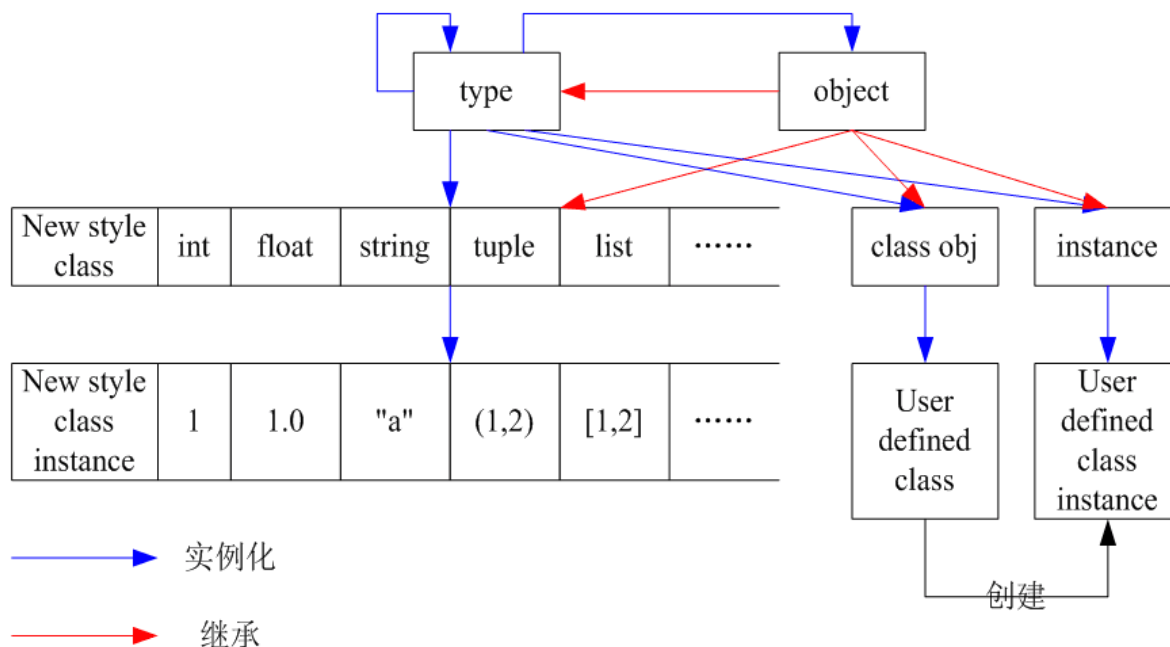


对象是类的实例，类又是元类的实例
python中默认的元素是type

Python中元类、类、对象的关系

4.10 Python语言的面向对象特征

类对象之间的关系(2.x)



Python中元类、类、对象的关系

4.10 Python语言的面向对象特征

1. 对象模型：继承

- (1) 类层次结构：

- 采用单根类层次结构,所有类都是Object类的子类
- 允许单继承、多继承和多重继承

4.10 Python语言的面向对象特征

1. 对象模型：继承

- (2) 继承方式：提供哪些继承方式？
 - Python采用两种继承方式，分别是：
 - 普通继承方式；
 - super继承方式；

4.10 Python语言的面向对象特征

普通继承方式

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')

class B(Base):
    def __init__(self):
        Base.__init__(self)
        print('B.__init__')

class C(A,B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        # super().__init__()
        print('C.__init__')

if __name__ == "__main__":
    c = C()
    print(C.__mro__)
```



Base.__init__
A.__init__
Base.__init__
B.__init__
C.__init__


4.10 Python语言的面向对象特征

super继承方式

```
5 class Base:
6     def __init__(self):
7         print('Base.__init__')
8
9 class A(Base):
10     def __init__(self):
11         super().__init__()
12         print('A.__init__')
13
14 class B(Base):
15     def __init__(self):
16         super().__init__()
17         print('B.__init__')
18
19 class C(A,B):
20     def __init__(self):
21         super().__init__() # Only one call to super() here
22         # super().__init__()
23         print('C.__init__')
24
25 if __name__ == "__main__":
26     c = C()
27     print(C.__mro__)
28
```

在super机制里可以保证公共父类仅被执行一次。

Base.__init__
B.__init__
A.__init__
C.__init__



(<class '__main__.C'>,
<class '__main__.A'>, <class '__main__.B'>,
<class '__main__.Base'>, <class 'object'>)

4.10 Python语言的面向对象特征

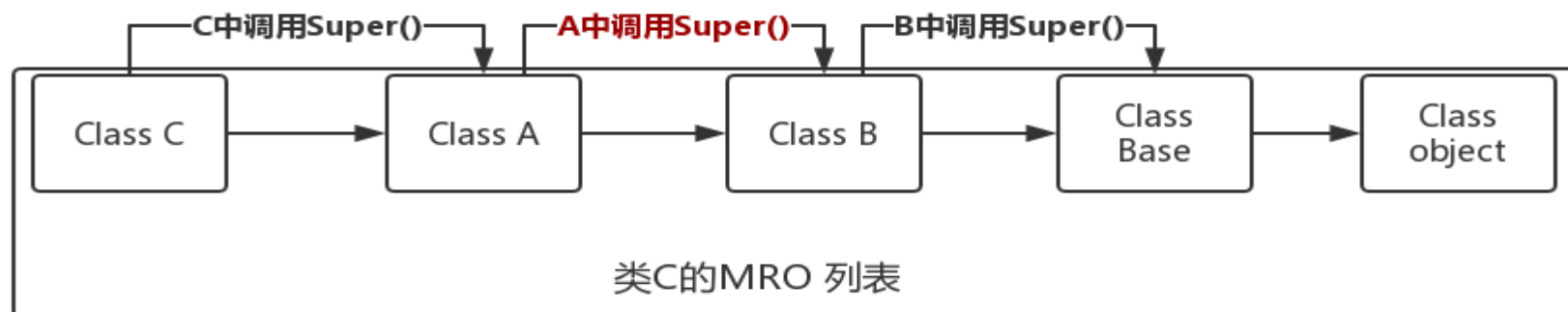
继承的实现

- 对于你定义的每一个类，Python会计算出一个所谓的方法解析顺序(MRO)列表，从左到右开始查找基类，直到找到第一个匹配这个属性的类为止。
- 这个MRO列表的构造实际上就是合并所有父类的MRO列表并遵循如下三条准则（C3算法）
 1. 子类会先于父类被检查
 2. 多个父类会根据它们在列表中的顺序被检查
 3. 如果对下一个类存在两个合法的选择，选择第一个父类

4.10 Python语言的面向对象特征

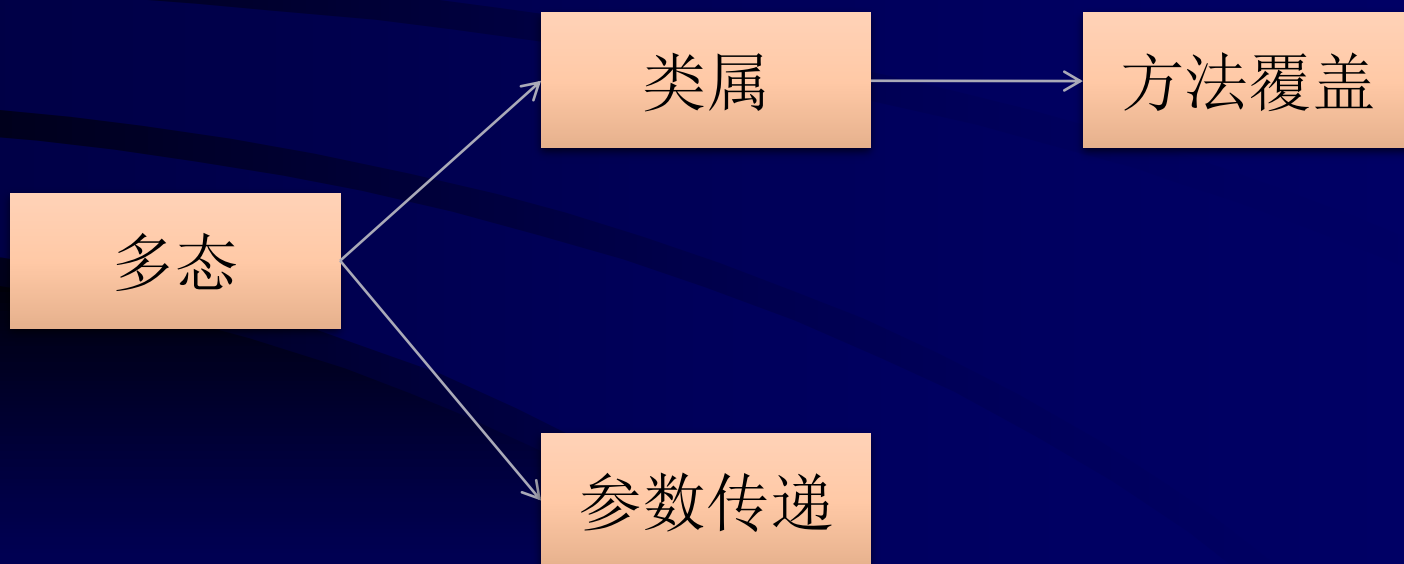
Super作用

- 当你使用 `super()` 函数时，Python会在MRO列表上继续搜索下一个类(并不一定是当前调用对象的父类，比如红色A中调Super实际执行的B)。



4.10 Python语言的面向对象特征

1. 对象模型：多态



4.10 Python语言的面向对象特征

1. 对象模型：多态

- 参数传递

```
class Bear(object):  
    def sound(self):  
        print("Groarrr")  
  
class Dog(object):  
    def sound(self):  
        print("Woof woof!")  
  
def makeSound(animalType):  
    animalType.sound()  
  
bearObj = Bear()  
dogObj = Dog()  
  
makeSound(bearObj)  
makeSound(dogObj)
```

通过动态绑定机制

```
__init__  
"D:\Program Files\Python\python.exe" D:/Source/python/20160918/src/__init__  
Groarrr  
Woof woof!  
  
Process finished with exit code 0
```

4.10 Python语言的面向对象特征

1. 对象模型：多态

- 类属

```
class Animal:
    def __init__(self, name=""):
        self.name = name
    def talk(self):
        pass

class Cat(Animal):
    def talk(self):
        print("Meow!")

class Dog(Animal):
    def talk(self):
        print("Woof!")

a = Animal()
a.talk()
c = Cat("Missy")
c.talk()
d = Dog("Rocky")
d.talk()
```

通过Self指代对象本身

```
__init__
"D:\Program Files\Python\python.exe" D:/Source/python/2016
Meow!
Woof!

Process finished with exit code 0
>>
```

4.10 Python语言的面向对象特征

2. 其他问题

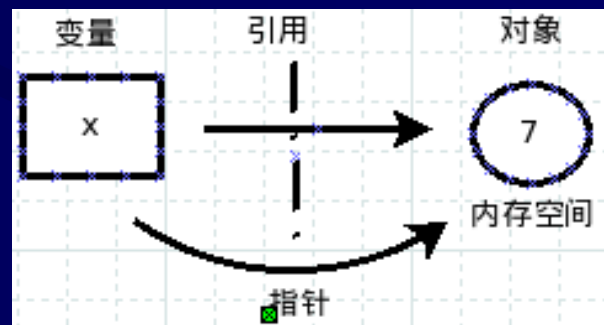
- (1) 是不是追求“纯粹”的面向对象语言？
- Python语言不是“纯粹”的面向对象语言；
 - 1) 存在全局变量和全局函数；
 - 2) main函数不是类的方法；
 - 3) 但Python一切皆对象。

4.10 Python语言的面向对象特征

2. 其他问题

- (2) 变量的语义模型：采用值模型还是引用模型？
 - 在变量的值模型中，值保存在变量的存储区里；
 - 而在变量的引用模型中，变量的值需要用另一个值对象表示，变量的存储区里存放的是对值对象的引用；

Python 采用的是引用模型；
变量通过引用建立与对象的联系；



4.10 Python语言的面向对象特征

2. 其他问题

- (3) 是否允许静态对象或者堆栈对象(自动对象)?
 - Python支持静态对象和自动对象;
 - 静态对象
 - 所有语言的全局变量都是静态对象;
 - 在Python语言中:
 - 使用global声明全局变量;
 - 用同样的global语句可以指定多个全局变量, 比如: `global x, y, z`。

4.10 Python语言的面向对象特征

堆栈对象(自动对象)

```
k=4  
def main():
```

```
    list1=[]  
    def add():  
        for x in  
            range(1,k):  
                list1.append(x)  
        print(list1)  
    add()
```

```
main()
```

Run-time Stack

add Activation Record

main Activation Record

list1

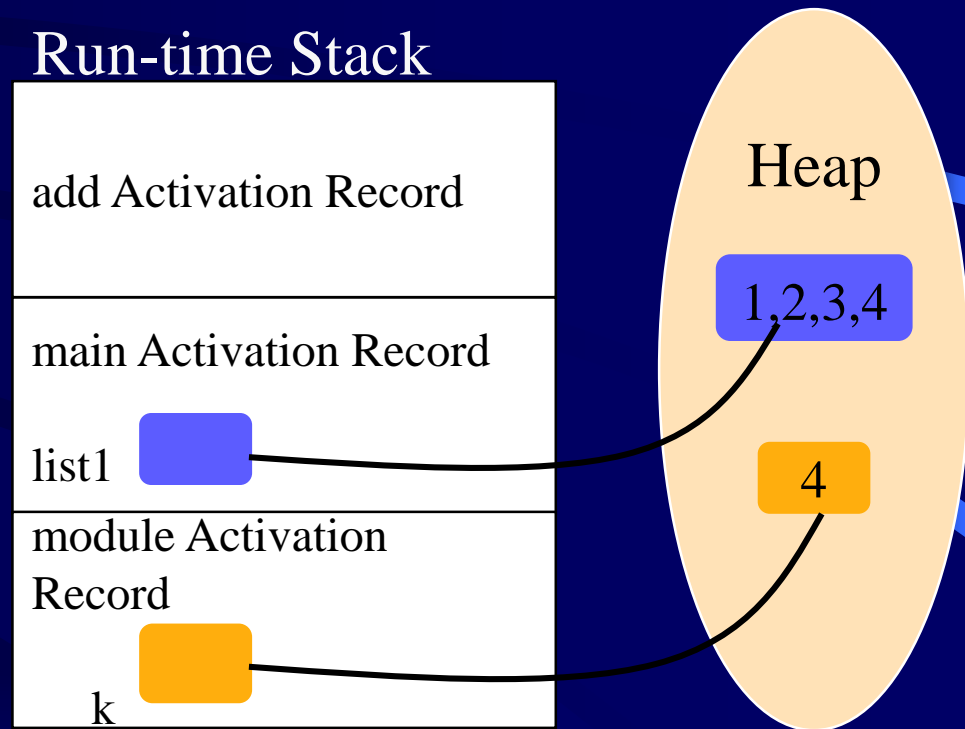
module Activation
Record

k

Heap

1,2,3,4

4



4.10 Python语言的面向对象特征

2. 其他问题

- (4) 是否依赖自动垃圾收集(GC)?
 - Python依赖自动垃圾回收(GC);
 - Python默认采用引用计数来管理对象的内存回收。
 - 当引用计数为0时，将立即回收该对象内存，要么将对应的block标记为空闲，要么返还给操作系统。

4.10 Python语言的面向对象特征

2. 其他问题

- (5) 是否所有方法都采用动态绑定？
 - Python中所有方法的调用都是根据对象所指向对象的类型来动态的确定(Python变量的语义模型：引用模型)。因此Python所有的方法采用的是动态绑定的方式。

4.10 Python语言的面向对象特征

2. 其他问题

- (5) 是否所有方法都采用动态绑定？
- Python 所有的方法采用动态绑定的方式；

```
class Fruit:
    def __init__(self, color):
        self.color = color
        print("fruit's color: %s" % self.color)
    def grow(self):
        print("grow...")
class Apple(Fruit):
    def __init__(self, color):
        Fruit.__init__(self, color)
        print("apple's color: %s" % self.color)
class Banana(Fruit):
    def __init__(self, color):
        Fruit.__init__(self, color)
        print("banana's color:%s" % self.color)
    def grow(self):
        print("banana grow...")

apple = Apple("red")
apple.grow()
banana = Banana("yellow")
banana.grow()
```

```
__init__
"D:\Program Files\Python\python.exe" D:/Source/python/20160918/
fruit's color: red
apple's color: red
grow...
fruit's color: yellow
banana's color:yellow
banana grow...
```

4.10 Python语言的面向对象特征

2. 其他问题

- (6) 类类型对象的动态转换机制
 - Python的类类型对象之间不提供动态转换机制;
 - 而是采用名-值(对象)动态绑定机制。