

Software Engineering Practice

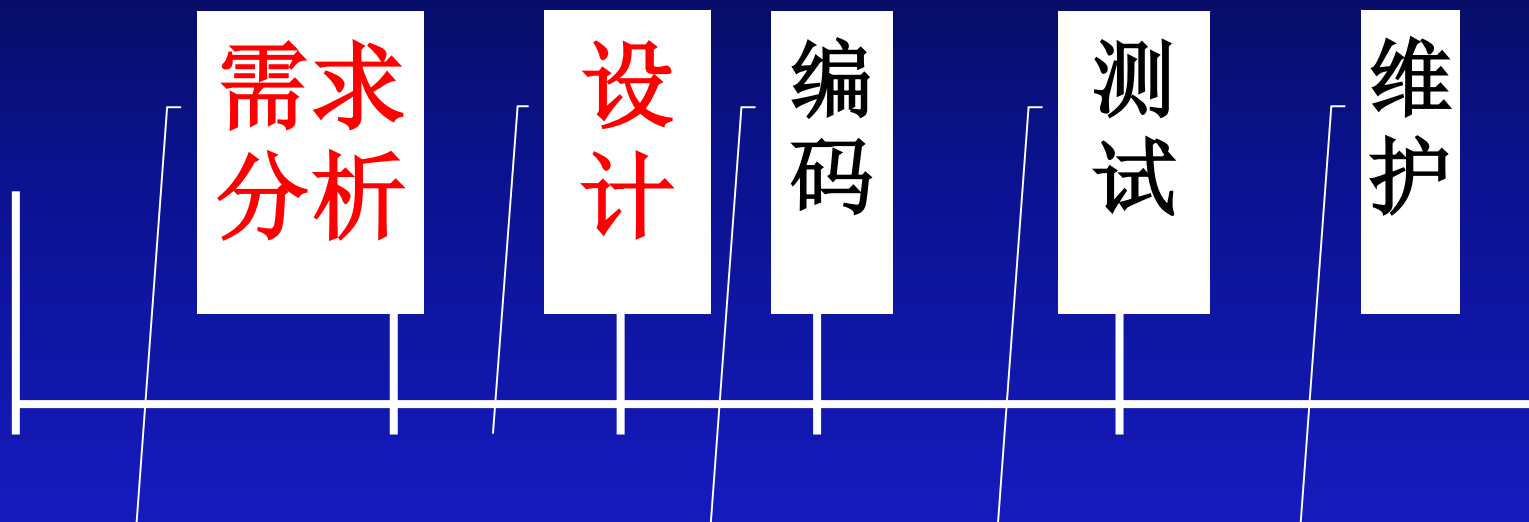
# 软件工程综合训练

## 软件需求与设计

计算机学院 闫波

yanbo@bit.edu.cn

# 软件需求分析与设计



# 软件需求分析与设计

## ➤ 软件需求

- （正在构建的）系统必须符合的条件或具备的功能和能力。【用户需解决某一问题或达到某一目标所需的软件功能；系统或系统构件为了满足合同、规约标准或其他正式实行的文档而必须满足或具备的软件功能】
- 需求分析的任务是借助于当前系统的物理模型导出目标系统的逻辑模型，解决目标系统做什么的问题。

# 软件需求

- 站在用户的角度上思考问题
- 业务-业务-还是业务



# 软件需求

## ➤快贷系统的合规性检查

- 前端还是后端？

- 业务的完整性保证？

- 如何平衡？

# 软件需求

- 大学教务管理系统-排课、学生选课？
- 多个教师上同一门课？单双周？实验？卓越计划学生？教室的安排（人数 位置）？教室调整？热门课选不上？小学期？上课时间调整？放假调课？休学、退学？短期出国？学生选课的冲突？教师上课的冲突？Curriculum-course-class-lassen？集中选课的系統压力？

国家级火炬计划项目



北京理工大学

# 正方现代教学管理信息系统

## —— WEB服务管理子系统



用户名：



密码：

☐ 部门 ☐ 教师 ☒ 学生 ☐ 访客

登录

关闭



点击查看教务处最新通知

说明：1、2012级学生初始密码为本人身份证号的后6位（字母均为大写），请登录及时更改密码。

2、对于2011年前入校的教师：用户名为：t+工号，如工号为：12345，则用户名为：t12345；  
对于2011年（含）后入校的教师，用户名为：工号的后6位，如工号为：6120110001，则用户名为：110001。



# 软件需求

## 需求错误的代价



# 软件需求

- 需求工程：用经过证实的有效原理、方法，通过合适的工具和记号，系统地描述待开发产品或系统的行为特征及相关约束。
- 需求工程过程：包括：需求获取、需求分析、编写软件需求规格说明（Software Requirement Specification, SRS）、一致性验证。
- 需求与模型：模型是描述现实问题的一种手段，它能抽象和简化地反映实际产品或系统的外部特征和本质。因此，它是需求工程的主要方法。

## ➤需求工程内容

过程包括:

- ◆初步沟通
- ◆导出需求
- ◆分析和精化
- ◆可行性研究
- ◆协商与沟通
- ◆规格说明
- ◆需求验证
- ◆变更管理

从需求到满足要求的软件产品。

通过不断揭示和判断的过程;

软件的功能和性能;

需求; 非功能性需求;

## ➤ 目标性功能需求

某系统需求这样写道：

根据详细调研和需求分析，系统功能必须满足：

- 1) 编制计划、工程拨款管理， .....工程必复、进度统计；
- 2) 工程项目管理
- 3) 计划拨款、征费收缴信息及其他收拨款信息查询统计；
- 4) 路产信息，违章建筑信息，工程材料、进度信息管理；
- 5) 养征信息管理，收费站信息管理；
- 6) 养护信息管理，桥梁维护预警；

.....

**目标性需求不具体，缺少必要的条件限定和执行过程。**

## ➤无二义性需求

通常给出的需求，都会给出必要的条件限定和执行描述，但对于它们的理解，并不一定只有一种。

例如：

下面的问题具有二义性：

“发现不友好的并且带有未知任务的或者有可能在5分钟内飞入禁区的飞机时鸣响警报。……”

第一种解释：

不友好的、带有未知任务的飞机要鸣响警报；或者5分钟内飞入禁区的飞机要鸣响警报（无论是否友好？无论是否带有未知任务？）。

第二种解释：

不友好的飞机，如果带有未知任务或者5分钟内会飞入禁区，则要鸣响警报（友好的飞机无论怎样都不鸣响警报？）

第三种解释：

.....

无二义性需求-----陈述的问题有且仅有一种解释

## ➤二义性缺陷的原因

- 对问题的理解不深入
- 表述不具体和严谨
- 自然语言本身的丰富语义表现力

## ➤解决二义性的途径----模型

- 系统模型可以抽象表示问题，使问题简单清晰
- 模型可以提高对问题的准确性理解
- 容易沟通并达成一致的理解
- 容易检查宏观上的完整性

随着问题的复杂，模型也会变得复杂。

## ➤非功能性需求

反映软件系统质量和特性的额外要求:

过程需求: 交付需求  
实现方法需求  
标准需求

外部需求: 法规需求  
费用需求  
互操作性需求  
.....

产品需求: 可用性  
性能需求  
可靠性需求  
可移植性需求  
可重用性需求  
安全性需求  
.....

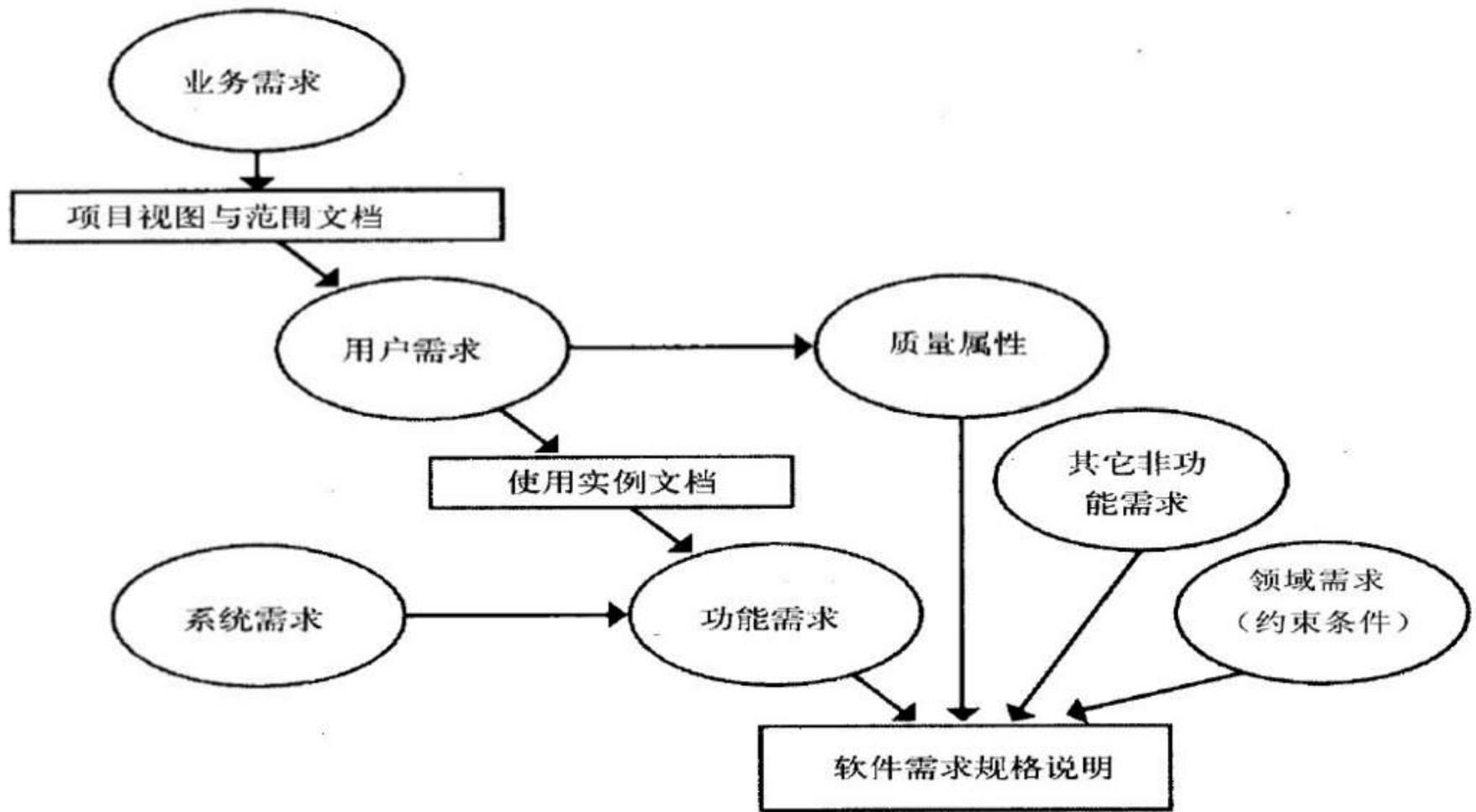


## ➤非功能性需求针对的问题

- 可移植性：软件在不同操作系统环境下可运行的程度；
- 可靠性：软件的错误保障能力，错误的识别和处理能力；
- 性能需求：软件的资源占用、客户容量、平均及峰值速率；
- 可用性：软件的人机界面友好性、可配置性、可理解性；
- 安全性：软件系统的访问、操作权限；抗攻击能力；
- 可重用性：软件的结构及构件可以再次被使用的程度；

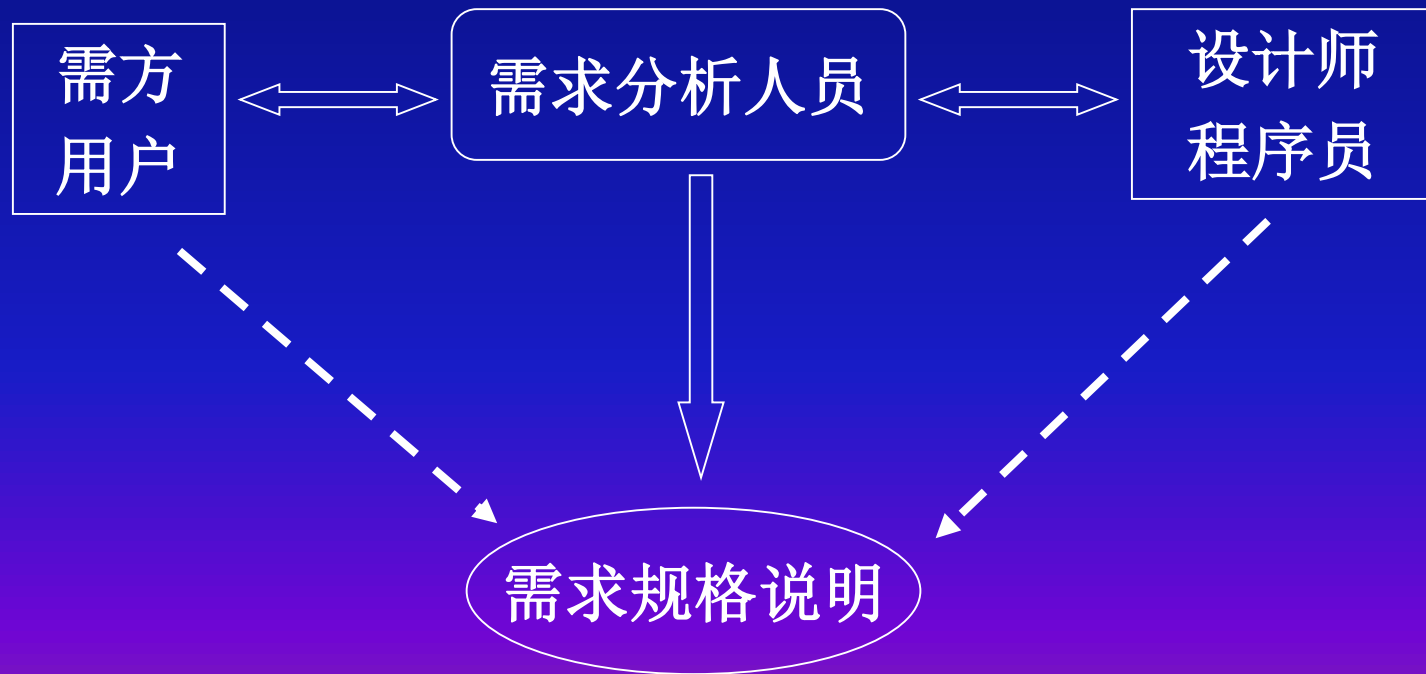
# 软件需求

需求工程文档



# 软件需求

- 需求工程所涉及的角色



# 软件需求

## ■ 需求分析能力

具有总体和全局的观点

能力：具有抽象观察和考虑问题的能力；

过程：能够把握需求过程和进度；

交流：善于表达、交流和与人沟通；

技术：具有全面的专业技术；

# 软件需求

## 需求工程过程

包括：需求获取

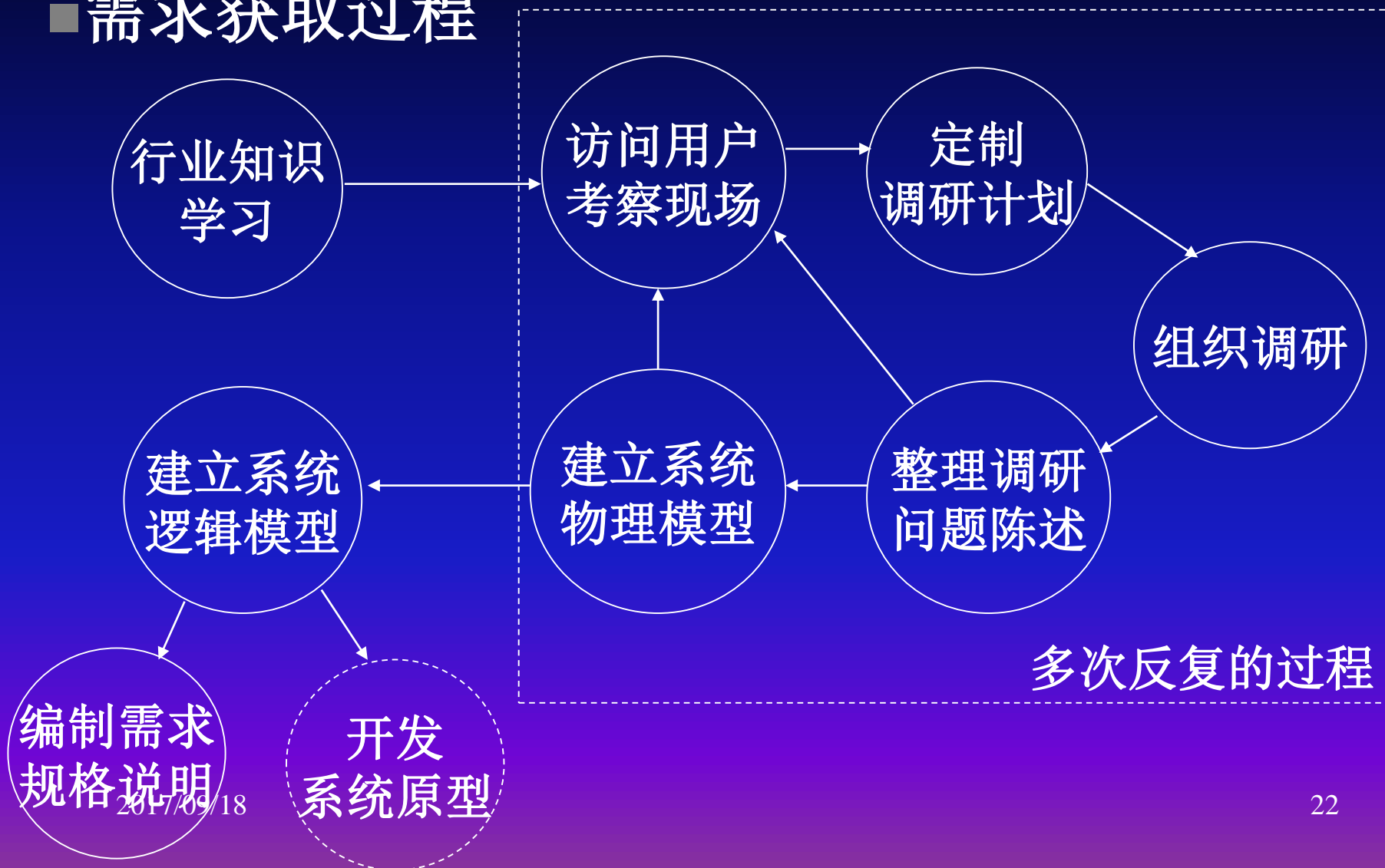
需求分析

编写软件需求规格说明SRS

需求验证

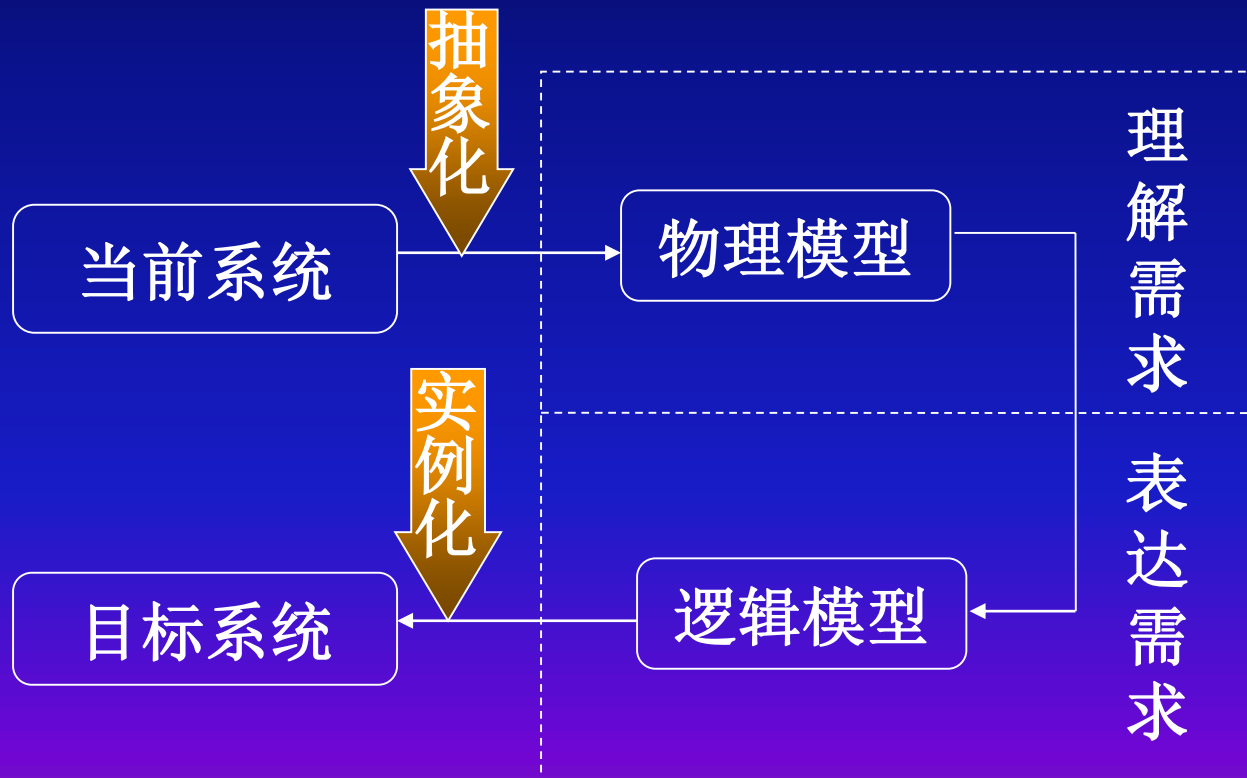
# 软件需求

## ■需求获取过程



# 软件需求

## 从物理模型导出逻辑模型



# 软件需求

	逻辑模型 (本质模型、概念模型)	物理模型 (实施模型、技术模型)
现行系统	描述重要的业务功能，无论系统是如何实施的。	描述现实系统是如何在物理上实现的。
目标系统	描述新系统的主要业务功能和用户新的需求，无论系统应如何实施。	描述新系统是如何实施的（包括技术）。

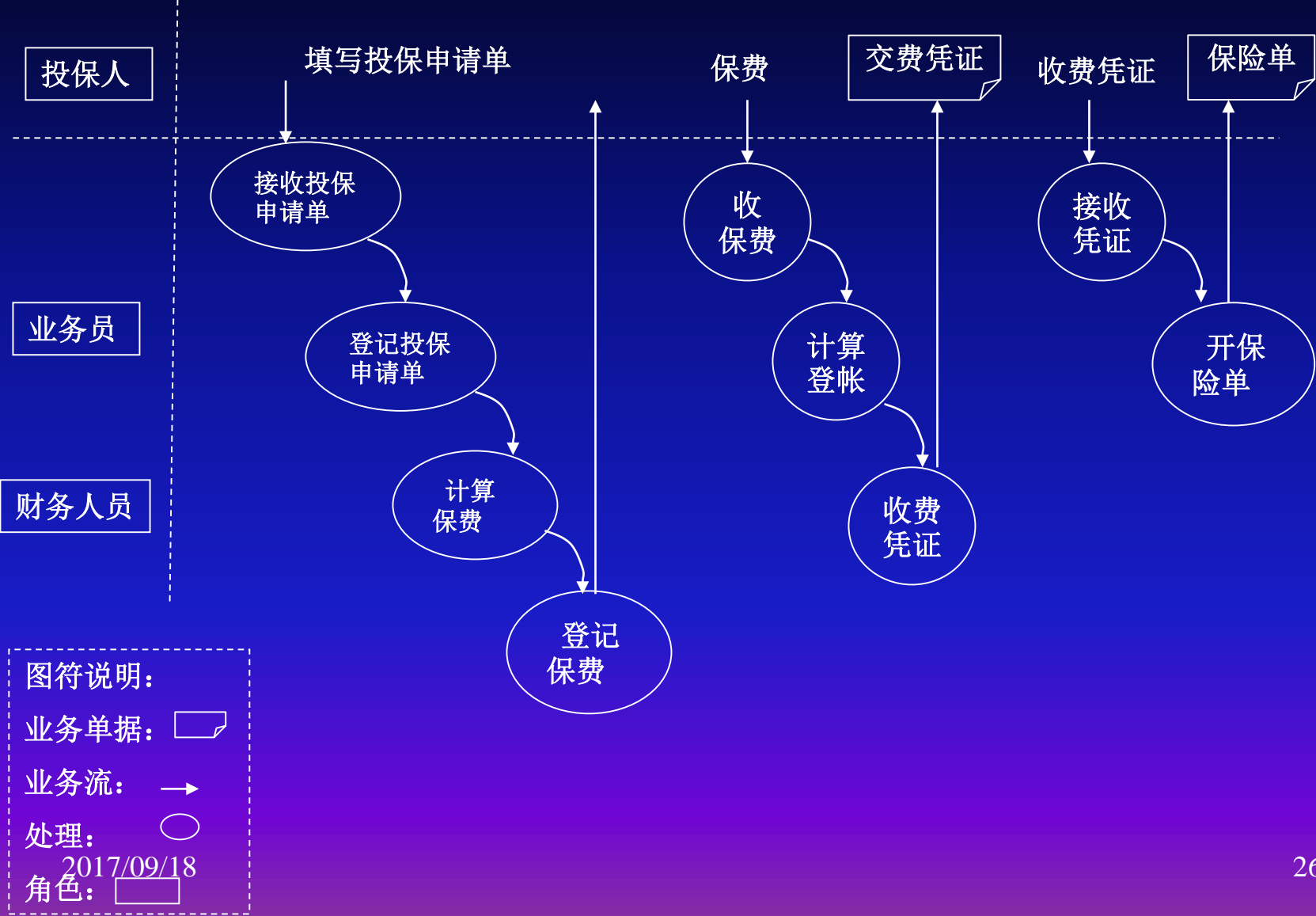


# 物理模型导出逻辑模型

## 业务陈述：

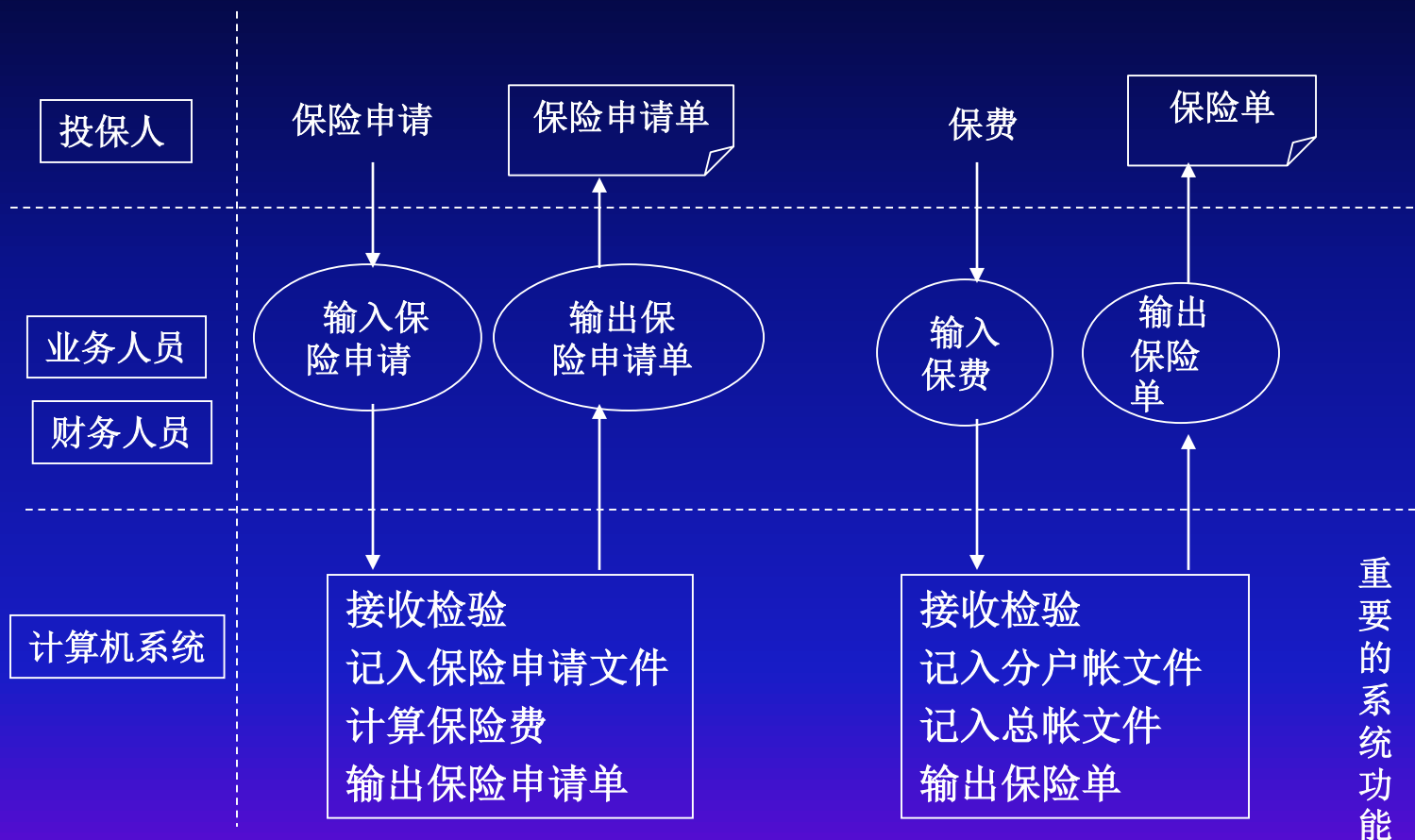
保险公司柜台业务，保险业务人员受理投保人的投保申请（包括：保险人，保险受益人、保险种类、保险年限等信息）；业务人员需要录入申请，根据投保申请计算保费；投保人若同意计算得到的保费，业务人员则将保费登记在投保申请单上；投保人持投保申请到交费处办理交费业务；待交费业务完成，需持交费票据，再由业务员办理保险单。保险单将作为保险公司与投保人理赔时的法律合同依据。

# 保险公司柜台业务过程物理模型：



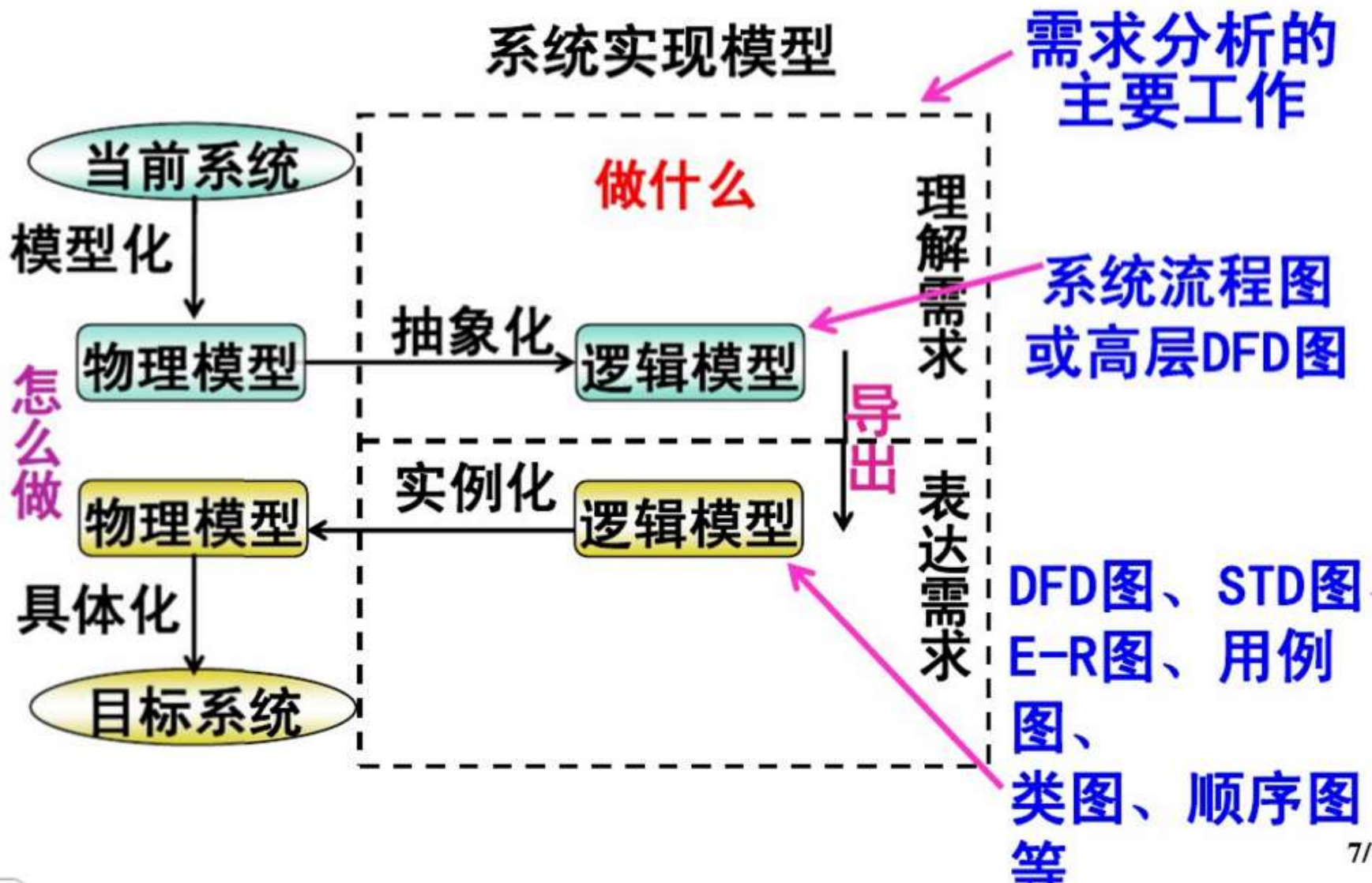
# 逻辑模型

分析保险公司柜台业务逻辑过程，得到系统的主要功能：



可根据系统的主要功能画出系统逻辑模型。

# 软件需求



# 软件需求

## ■编写需求规格说明

IEEE97年提出的需求规格说明的8条原则：

- 1) 从实现中分离功能，即描述“做什么”而不是“怎么做”；
- 2) 有面向处理的规格说明，以描述系统级的动态行为；
- 3) 以软件为元素，描述系统中各元素的关系；
- 4) 对系统的运行环境说明，以保持系统接口描述的一致；
- 5) 以用户能接受和理解的形式描述；
- 6) 规格说明必须可操作；
- 7) 可容忍不完备性和可修改性；
- 8) 说明应局部化和松散耦合，以适应信息变化的易修改性。

## ■不属于SRS的内容

反映产品开发队伍的特性要求不属于SRS的内容  
例如：

### 1) 项目需求

人员、进度、费用、阶段性成果等

### 2) 产品保证计划

配置管理计划、测试计划、质量保证计划等

### 3) 软件设计

## ■需求验证

评审需求规格说明的质量一般具有如下性质：

- 1) 正确性
- 2) 无二义性
- 3) 完整性
- 4) 可验证性
- 5) 一致性
- 6) 非计算机人员理解性
- 7) 可修改性
- 8) 可跟踪性

- 正确性

相对于用户的应用需求，SRS描述的内容是正确地表示了未来软件的功能及性能要求。

- 无二义性

SRS中使用标准化术语，并对术语的语义进行显式的、统一的、一致的说明。



- 完整性

SRS达到完整性的4个指标:

- 1) 包含软件要做的全部事情，包括输入和输出、功能和性能；
- 2) 包含软件所有交互数据的有效和无效说明解释；
- 3) SRS中文字上的完整性，包括图表编号、页码、名字，等；
- 4) 如果遗留待解决的问题（**to be determined TBD**），需要注明解决的责任人和期限。

- 可验证性

存在技术上和经济上的手段，能够对软件的需求给与验证。所有需求应该对应有有限次可验证过程。

不可验证的性能描述，例如：

“当按下按钮时，通常应该红灯亮...”

“软件有一个容易使用的用户界面...”

## •一致性

SRS中的各项需求不能相互矛盾，主要表现是术语上的冲突、行为上的冲突和时序上的冲突。

相互矛盾的例如：

“提醒用户输入....”， “要求用户输入....”

“输入必须通过菜单完成...”，“输入命令如下....”

“按钮A之后B起作用....”， “A按钮在B之后起作用”

- 非计算机人员可理解性

SRS是面向用户和软件设计人员两方面的文档，尽量使用用户行业熟悉的词汇，避免太早地使用计算机专业术语。

例如：

“账号”，不要使用 “帐户标识”

“提交表单”，不要使用 “写文件”

## •可修改性

SRS的格式和组织方式，应能保证后续的修改容易进行。

- 1) 有必要的索引；
- 2) 各个事务的描述保持独立性（独立用纸）；
- 3) 在独立性基础上，有足够的上下文描述；

## •可跟踪性

SRS必须提供能与用户需求联系，并为后续设计文档引用的必要手段。

后向跟踪----与用户需求的联系

前向跟踪----与设计文档的联系

要求：

- 1) 每项需求都要有独立的名字或标识；
- 2) 建立必要的需求跟踪对照表；

(Requirements Traceability Matrix, RTM)

# 原型化方法

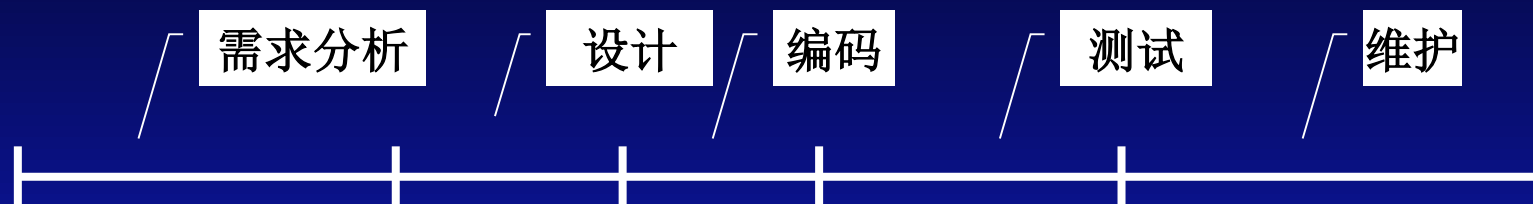
- 原型是制造业普遍使用的产品开发方法;
- 原型能表现出目标产品的功能和行为特性;
- 软件原型与一般产品的原型区别:
  - 1) 对于整个开发过程而言, 相对时间短;
  - 2) 对于整体软件费用而言, 相对费用低;
  - 3) 一个原型只对应软件产品的一个版本, 而不是一个原型下对应多个型号。

- 软件原型的类型

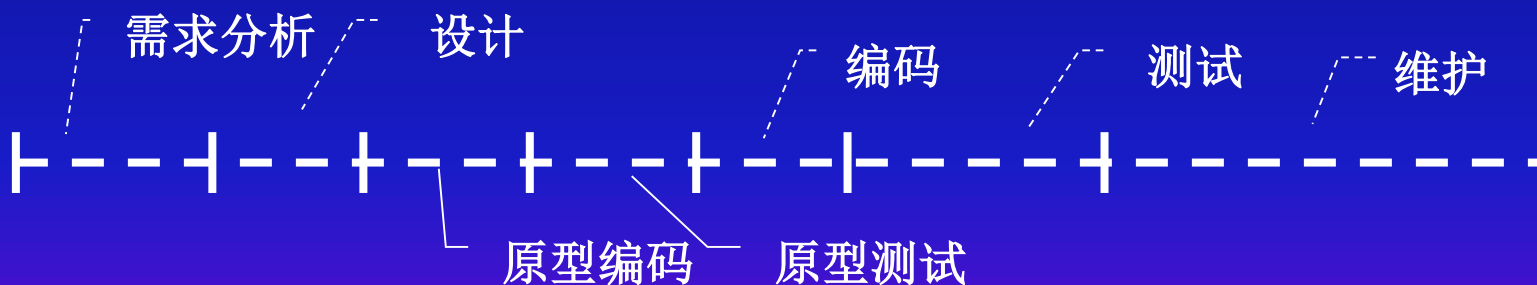
- 1) 全功能展示
- 2) 人机交互模拟
- 3) 程序实现框架
- 4) 探索性开发技术研究
- 5) 作为开发演进的基础



# • 原型化方法的软件过程



非原型化方法软件生命周期



原型化方法软件生命周期

- 软件原型方法的过程

## 1) 功能选择

包括最终产品的部分功能，为探索系统需求和实现技术。

包括可选择的多个方案，为通过原型演示，进行方案比较。

## 2) 构造

着眼于评估，而不是长期使用。

性能问题先不与考虑（不包括主要为解决性能问题的原型）。

### 3) 评估

评估是原型化方法中决定性的工作；

评估要有人力物力的保证；

评估一定有用户的参与；

### 4) 使用

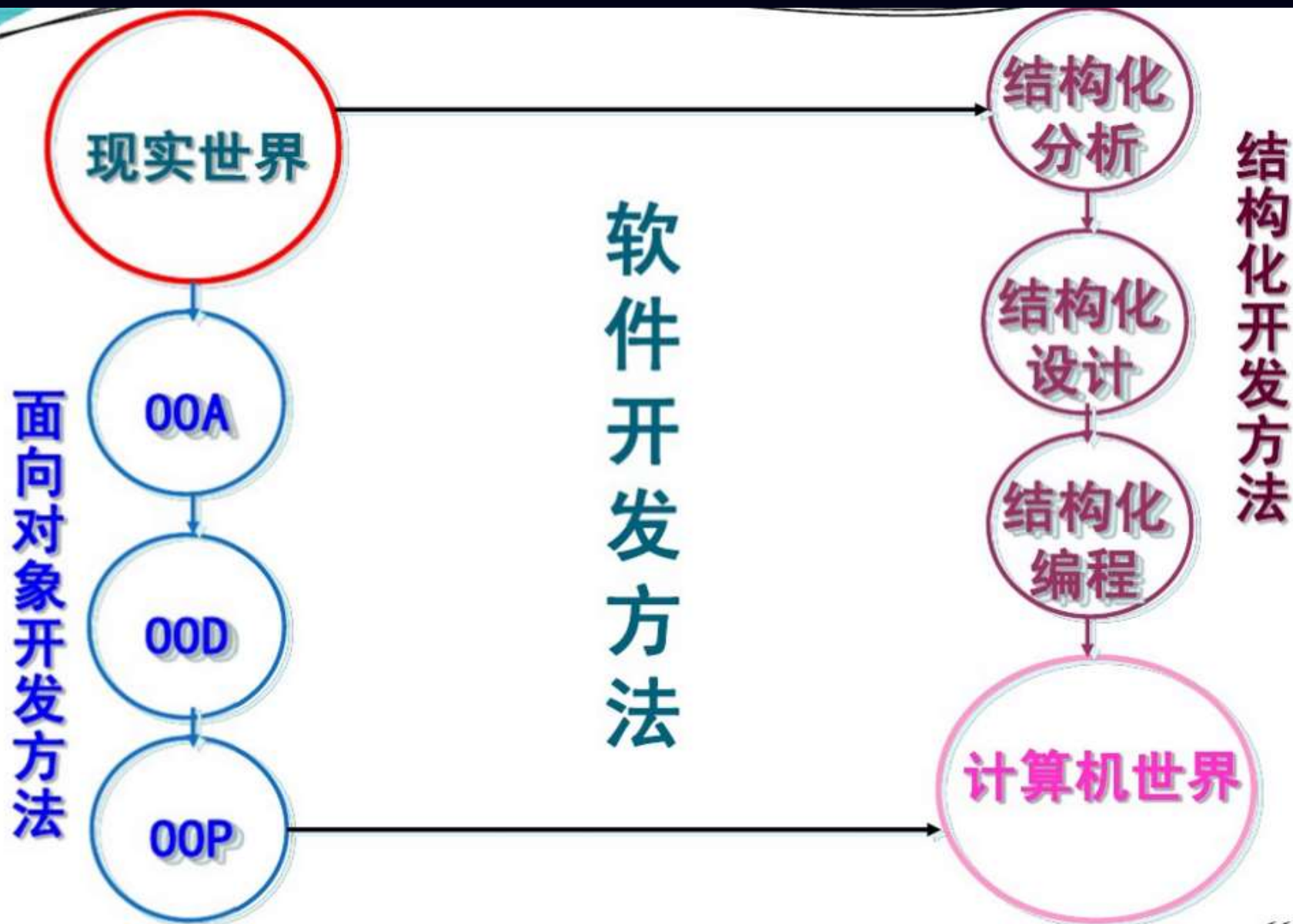
几种对原型可能的使用：

用于未来系统的某一部分；

用于后期开发实现技术的使用；

用于未来系统的熟悉环境性培训；

完全不使用；



# 结构化分析与建模 (Structured Analysis)

# 主要思想与基本内容

## ➤ 结构化方法的主要思想：（Structured Analysis）

- **结构化分析方法**是一种传统的系统建模技术，其过程是创建描述信息内容和数据流模型，依据功能和行为对系统进行划分，并描述过必须建立的系统要素。
- **结构化分析**：使用数据流程图、数据字典、结构化英语、判定表和判定树等工具，来建立一种新的、称为结构化说明书的目标文档—需求规格说明书。
- 结构化体现在将软件系统抽象为一系列的逻辑加工单元，各单元之间以**数据流**发生关联。
- 该方法的要点是：面对数据流的分解和抽象；把复杂问题自顶向下逐层分解

## ➤结构化方法基本内容包括:

- 数据建模
- 功能建模
- 行为建模

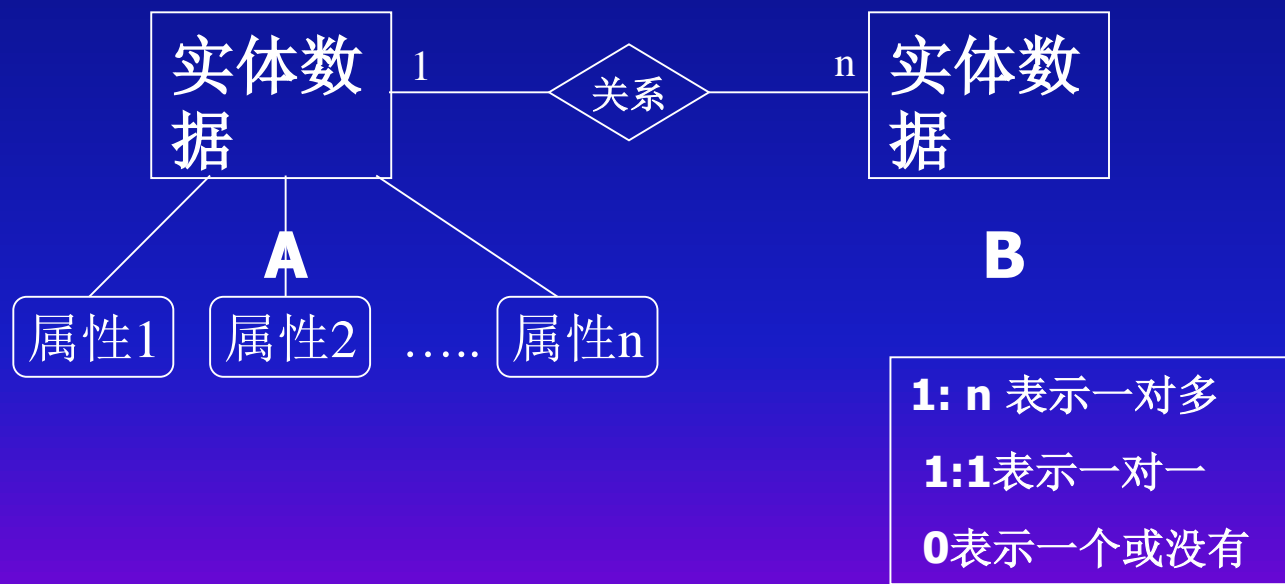


# 数据建模E-R图

实体-关系图（Entity Relation Diagram）

实体：系统需要记录及处理的数据集（DB中的表）

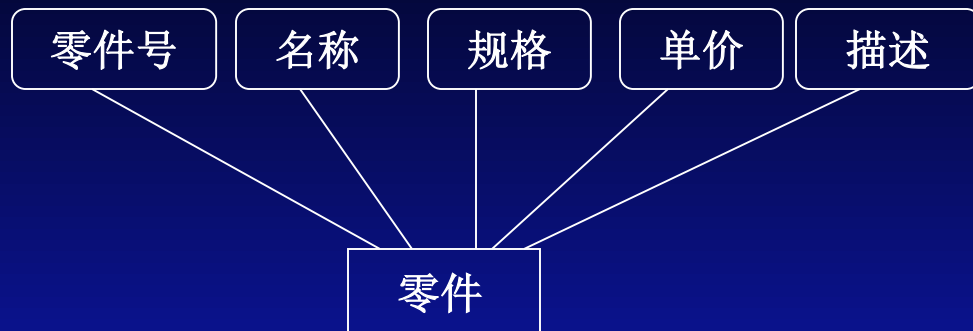
关系：实体间在系统中需要的必然联系（DB中的主键和外键）



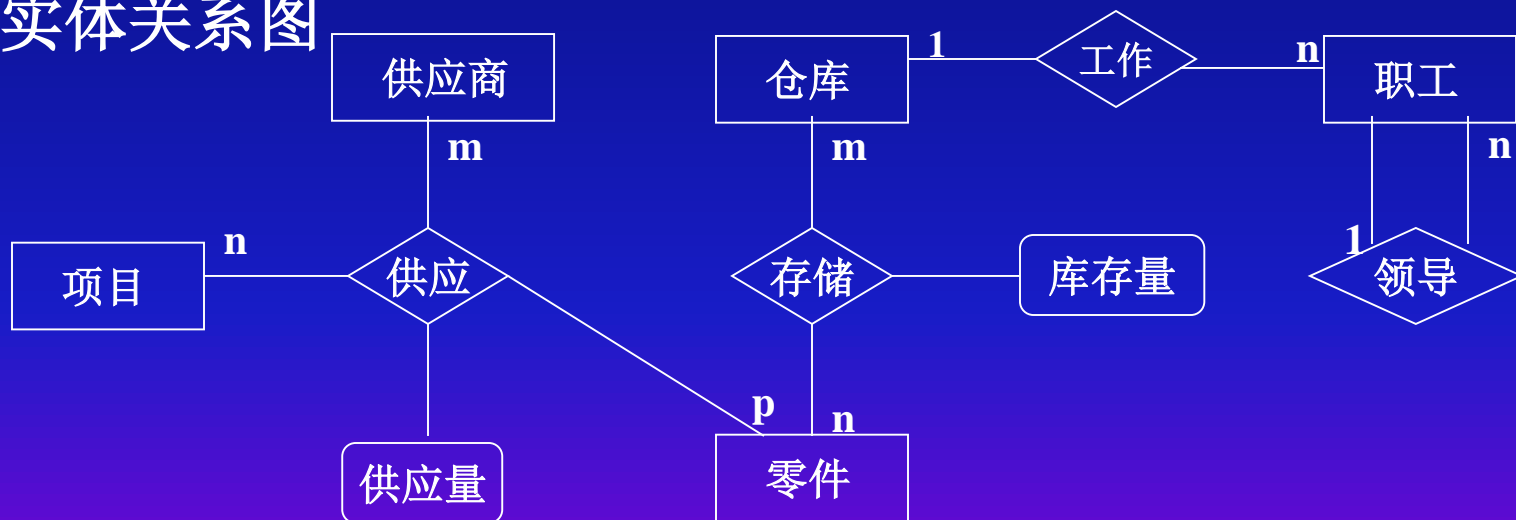


# 实体-关系图示例

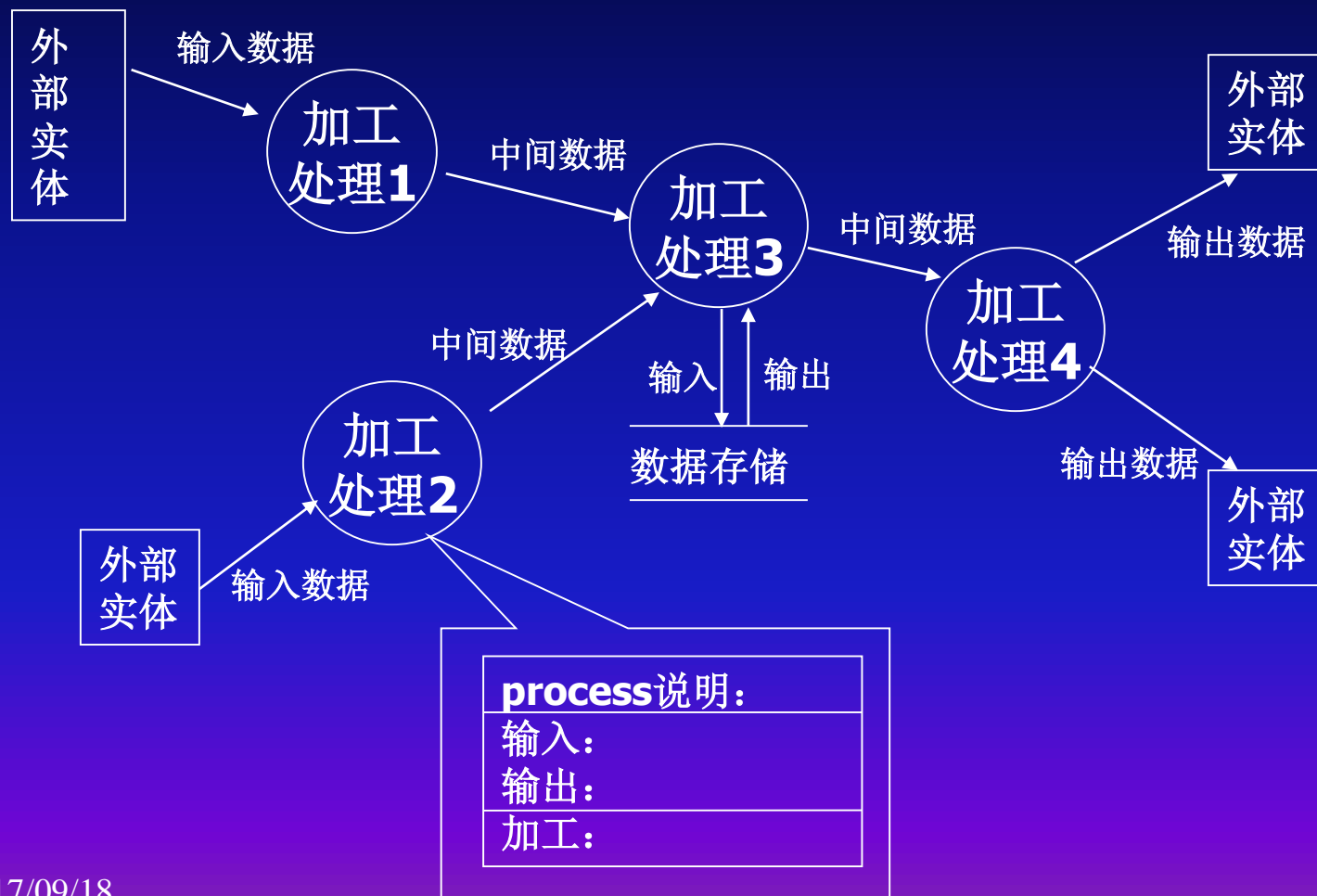
## 实体属性图



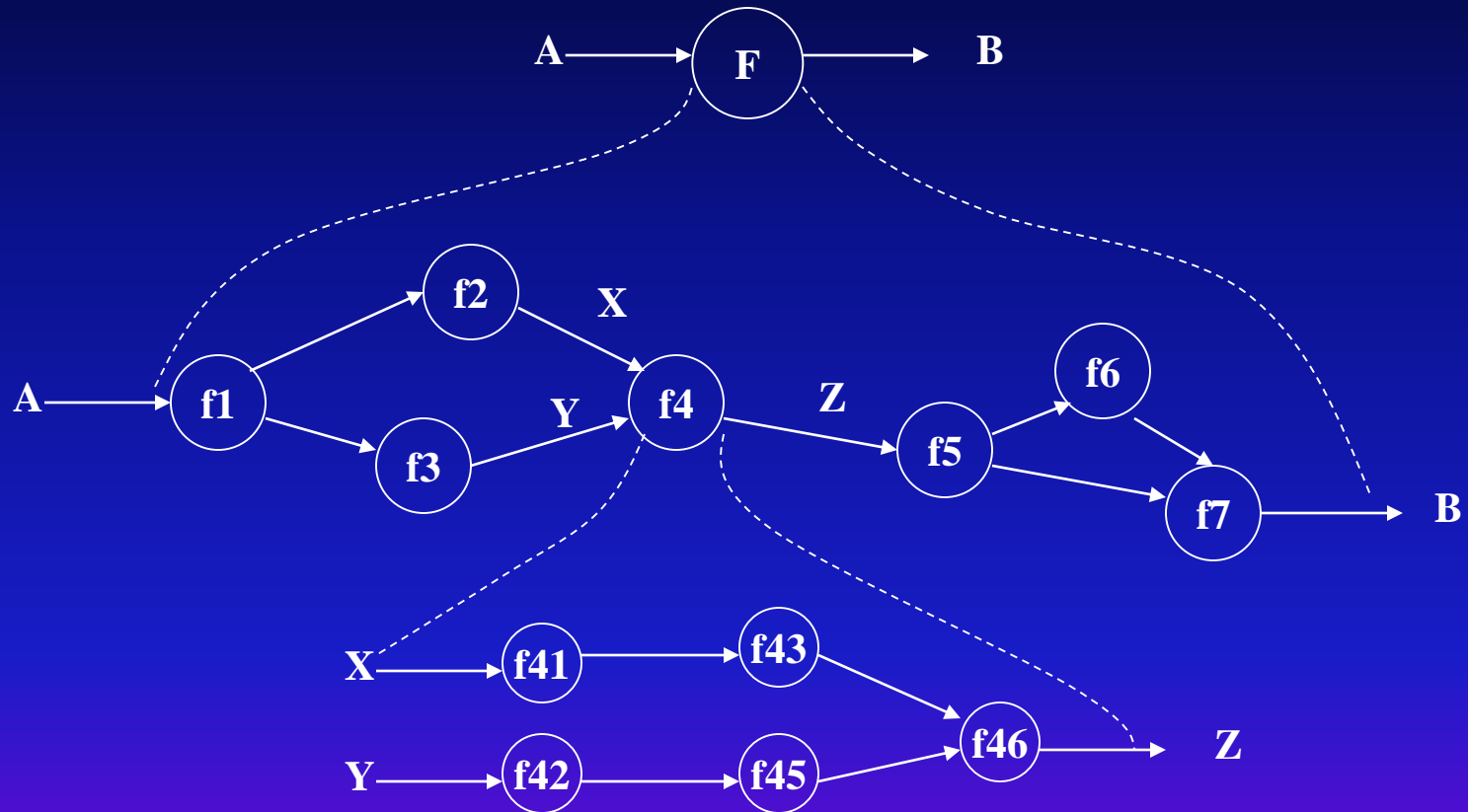
## 实体关系图



# 功能建模DFD 数据流图(Data Flow Diagram DFD)



## ➤数据流结构化



## ➤数据字典

- 作用：对数据流图中的数据流、文件、数据项和加工的约束性描述

- 数据流定义：列出数据流组成的数据项

数据项集合={数据项1+数据项2+数据项3…}即：文件

数据项=数据子项1+数据子项2+数据子项3…

数据子项=[范围] | [限值] | [类型] | [精度]…

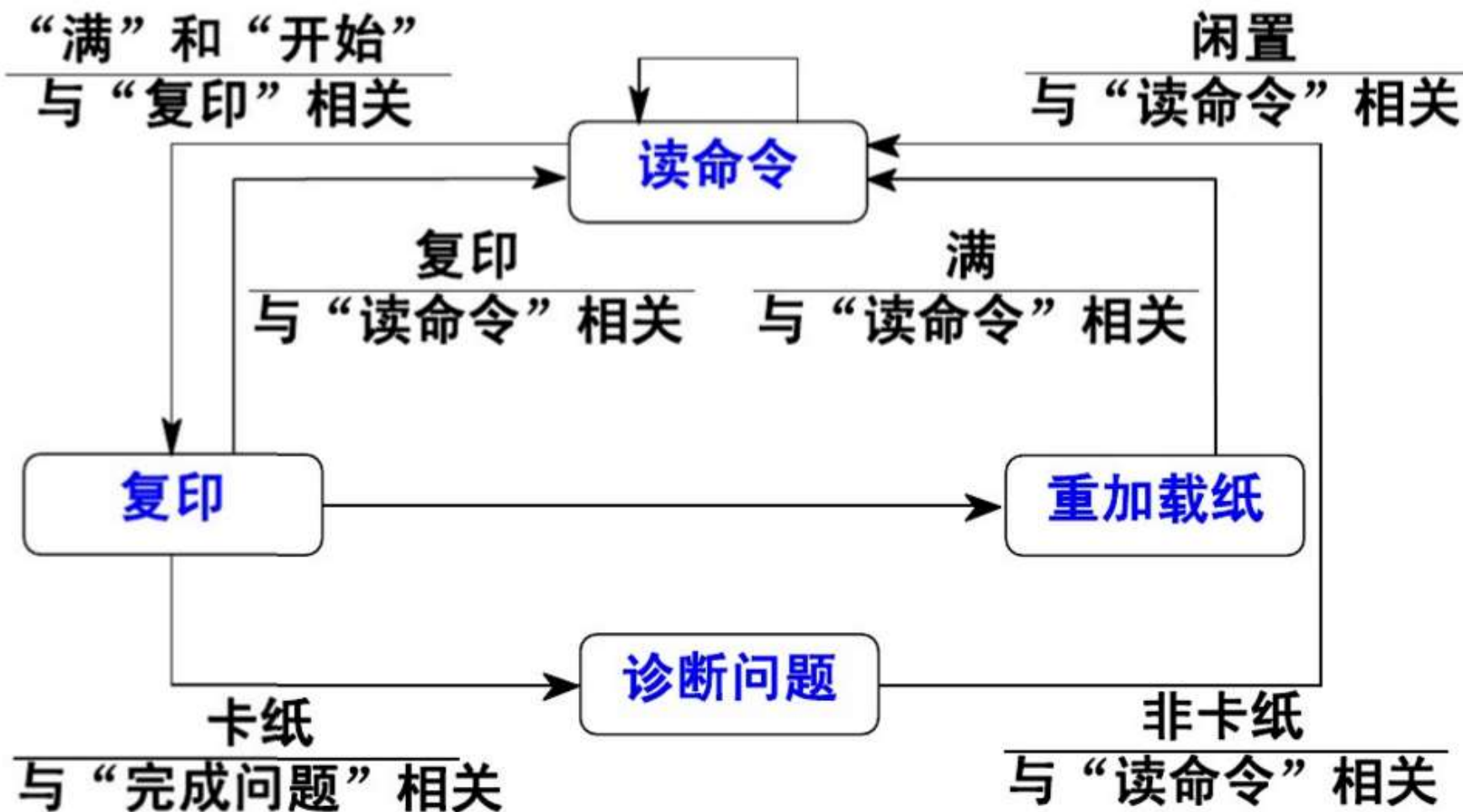
- 加工说明：

简单处理逻辑、激发条件、优先级、执行频率、最大数据量、峰值等

## ➤数据流图的绘制原则

- 图上仅限于规定的图符及线段；
- 图中的每个元素必须有名字；
- 顶层图必须包括数据源或数据终点（外部实体完整）；
- 每个加工至少有一个输入流和输出流（数据守恒或封闭）；
- 数据流图分层，加工的输入输出必须保持与分层图的一致（子父图平衡）；
- 文件不可独立出现，要与其连接成份一同出现（合理文件表示）；
- 每个加工的分解要自然合理，原则上不超过7。

## 举例：复印机控制软件状态图



# 建模过程与示例

对问题陈述做语法分析：

- ①区分所有动词和名词
- ②动词作为“变换处理”
- ③名词作为外部实体、控制对象和数据存储
- ④画 DFD
- ⑤对“变换”做进一步的陈述精化
- ⑥重复做①到④步
- ⑦用状态变迁图协助分析控制事件和系统状态，画 STD
- ⑧可以写必要的规约变换加工小说明
- ⑨对所有系统的加工和产生数据，以及控制事件和转换状态定义数据字典DD，既明确做出解释。

# 图书馆系统实体关系图

## 图书馆系统数据字典

**名称:** 标题

**别名:** 抽象的图书

**描述:** 描述一个抽象的图书的信息

**定义:** 标题=ISBN+书名+作者+出版社+出版日期版次+价格  
+目录+内容简介+馆藏数+可借数+预约数

**位置:** 图书查询、借书、还书、预约

**名称:** 书目

**别名:** 具体的书

**描述:** 对应标题的具体的一本书

**定义:** 书目=条码号+分类号+ISBN

**位置:** 借书、还书、更新



# 面向对象分析与建模

# 面向对象方法基本概念

## ➤ 思想方法

- 从现实世界中客观存在的事物出发，直接以问题域中的事物为中心，思考和认识问题。
- 将事物的本质特征和系统责任，抽象表示为系统的对象，作为系统的基本构成单位，建立软件系统。
- 强调运用日常逻辑思维经常采用的思想方法和原则，例如：抽象、分类、继承、聚合、封装和关联，并以易懂方式表达。

## ➤ 类和对象的概念

- 类使用抽象的方式，抽象是人们分析问题常用的方法，抽象即是忽略事物非本质的东西，而突出事物本质的特征和行为；
- 在外部世界中，相同属性和操作的对象属于一个类，对象是类的一个实例；在机器空间中，类是一个可复用的模板，对象是复用出来的独立可执行程序块；
- 抽象包括数据抽象和过程抽象，数据抽象是根据施加在数据上的操作来定义的数据类型；过程抽象是将确定的处理功能定义为独立的过程实体；
- 对象包括实体对象和无形对象；

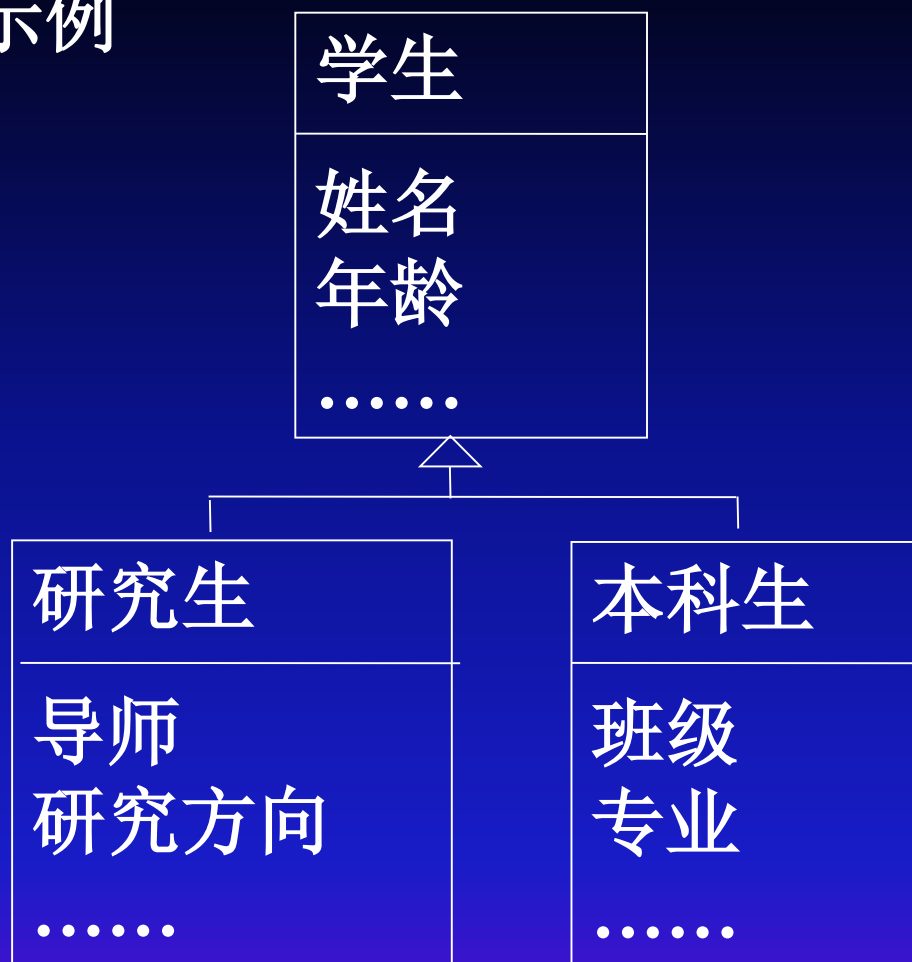
## ➤ 封装和消息的概念

- 把类的内部属性和一些操作隐藏起来，只将公共的操作对外可见。避免外界错误和内部修改带来的影响；
- 对象只通过消息来请求其他的对象执行自身的操作；
- 消息必须直接发给指定的对象，消息中包括请求执行操作的必要信息；
- 一个对象是消息的接收执行者，也可以是消息的请求发送者。

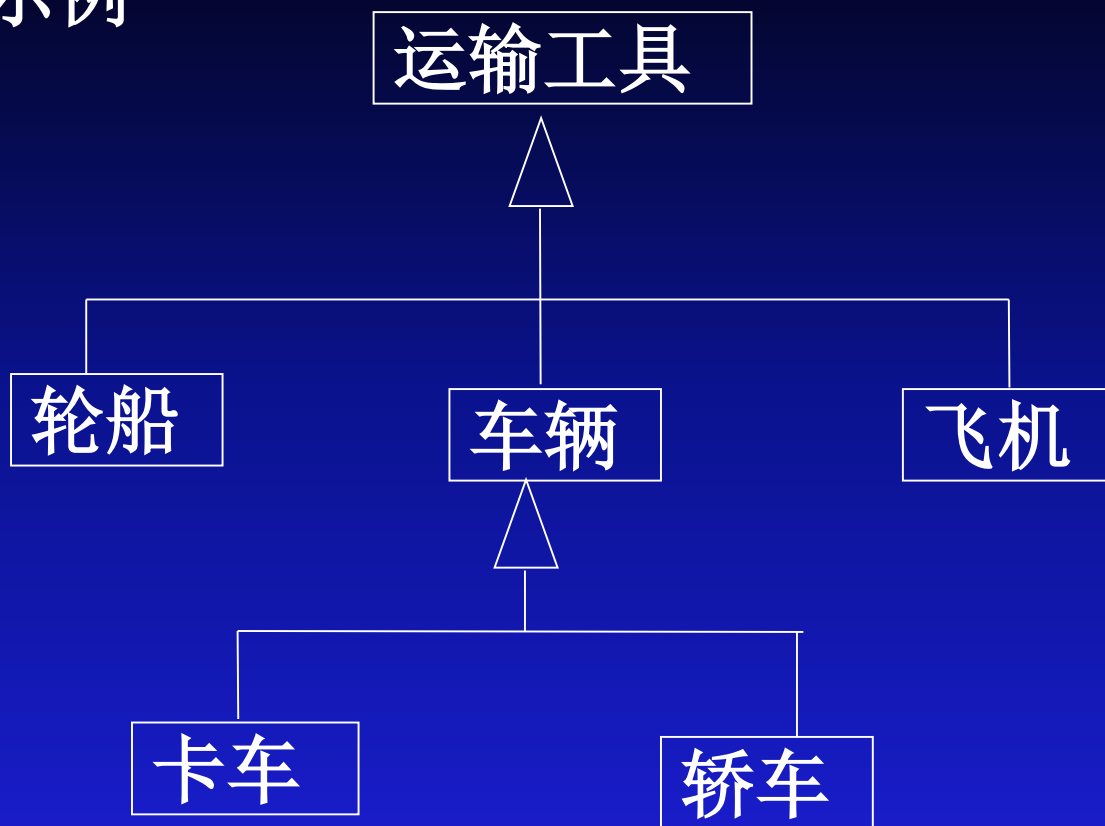
## ➤ 继承的概念

- 类可以有子类，子类继承父类的全部属性和操作，并允许添加自己的属性和操作；
- 继承是泛化和特殊的关系，父类具有某事物的一般共性，子类描述的事物在父类的基础上并比父类更特殊；
- 继承可以有双重性，允许多于一个父类；
- 继承允许多层，各层之间具有传递性；
- 继承的重要作用在于源代码的复用。

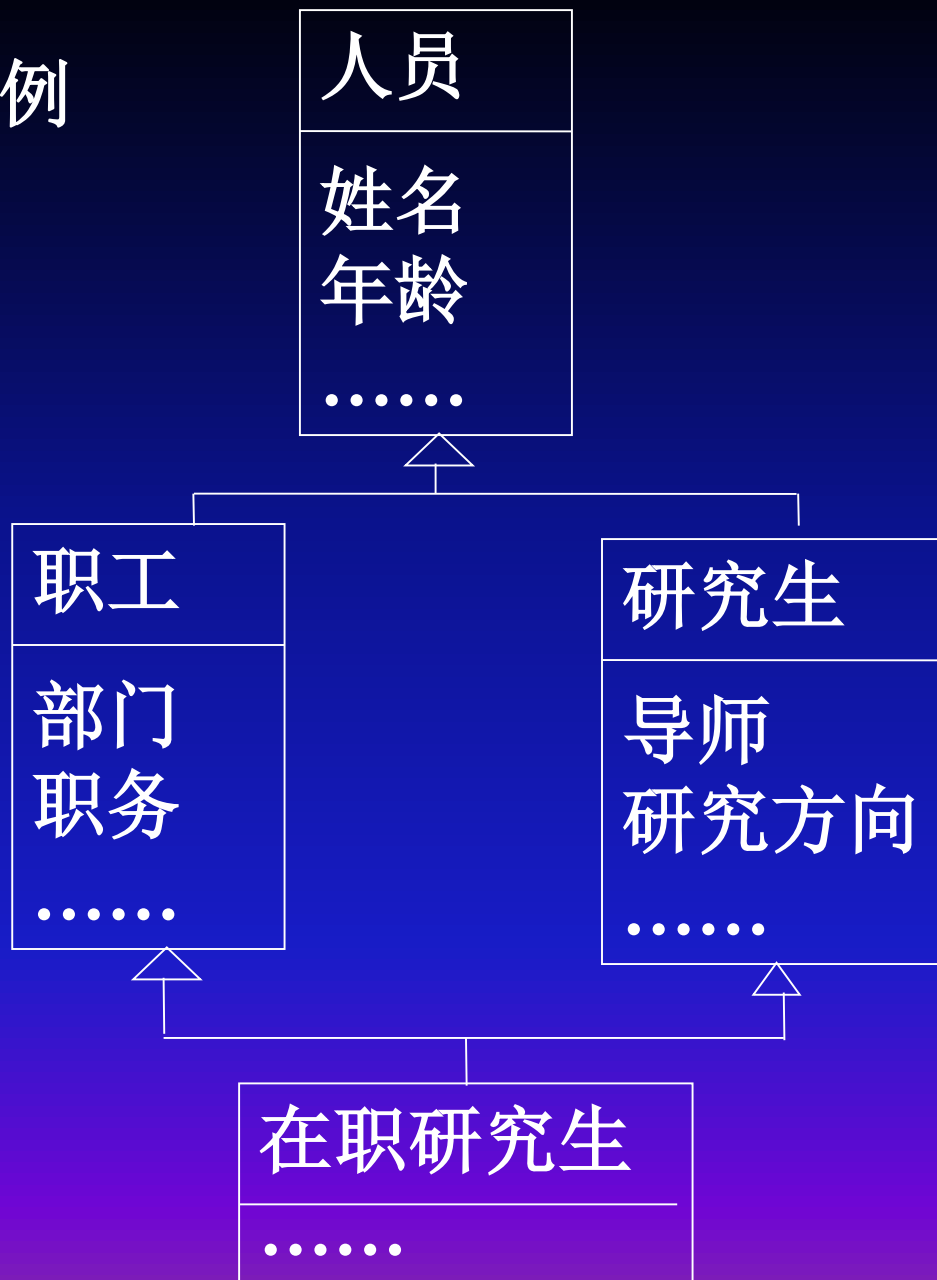
## 继承示例



## 多层继承示例



## 多重继承示例





## ➤ 多态性

- 在继承类结构中允许定义同名操作，同一个消息的响应可以执行不同的行为，即同一操作的多种形态。
- 多态性更好地体现了操作语义的一致性，实现接口封装独立性和信息隐蔽的原则。

## •多态性的实现机制

静态联编：

编译时确定所访问对象的操作地址；

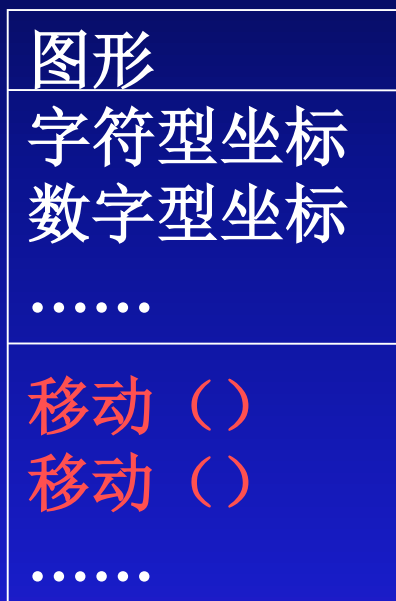
动态联编：（滞后联编 或 动态绑定）

编译时不确定所访问对象的操作地址，在运行时根据操作对象的不同再确定；

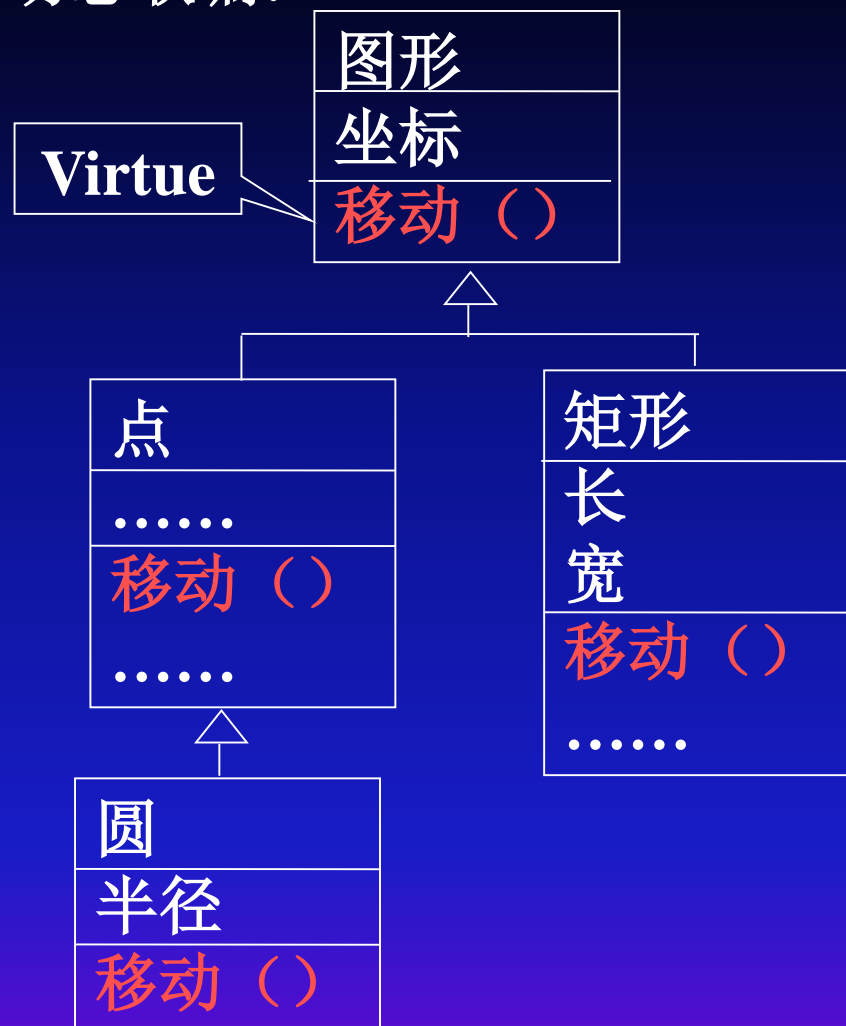
动态联编使软件应用分布式发展有了重要突破，成为网络信息化、软件体系结构研究的主要支撑技术。

多态性:

静态联编:



动态联编:



可使用统一图形移动接口, 采用移动 () 的动态联编

## ➤面向对象方法的目标及实现机制

- 高可维性 ----对象类机制
- 可复用性 ----继承机制
- 程序无关性----多态性动态绑定机制

## ●高可维性:

寻找可构造的元素（类）作为系统构造的基础，回避在不稳定基础上建造系统，使系统成为可构造和高可维

- 1) 类是封装了操作的一个“**代码级复用**”程序模板，类的对象是系统的可构造元素；
- 2) 采用**消息机制**执行对象的操作，回避了功能调用的过程性；

## ●可复用性:

1) 对象语义一致性: 功能的复用依赖于对功能的理解, 对功能的描述是复杂和多义的, 相比对象语义来说不容易复用。

2) 全方位复用: 功能复用是代码级的, 软件复用不但需要代码级的, 还应该有序程序级的复用。继承机制是源程序级的复用。

## ●程序无关性（机器无关性）

在任何机器环境下，使用任意程序语言，所编的程序都应该是逻辑通用的。这样，与机器相关的成分应该与处理逻辑无关。

类的多态性以及动态绑定技术，提供了独立接口的实现技术，这样可以将与机器相关的成分独立出来，为程序无关性奠定了基础。

## ➤ 面向对象方法的发展

从80年代到现在20多年的发展过程，面向对象方法学经历了百家争鸣的过程，逐渐走向成熟和统一。主要表现在：

### 1) 横向求同

多种的面向对象方法之间的求同。例如：基于方法观点的Booch方法，引入OMT方法的建模概念，以及赞同使用CRC卡技术(Use Case)，基于模型的方法中也添加了基于方法的抽取类的策略途径。



## 2) 纵向求同

与非面向对象方法的求同。例如：结构化方法很多优秀的表示和策略被面向对象方法肯定和继承。同时，结构化方法仍然在发展，并加入了面向对象方法论的主要成分。如“用对向分割信息，用事件分割功能”，等等。

## 3) 标准化

96年由Booch、Jacobson、Rumbaugh三位面向对象方法的专家，在国际对象管理组织OMG建议下，在Rational公司研制了（Unified Modeling Language UML），从97年的1.0版到现在的1.3版，经历逐步完善的过程。UML是业界共识的标准。

# 统一建模语言UML

## ➤ UML (Unified Modeling Language) 定义:

UML是对软件密集型系统中的制品进行可视化、详述、构造和文档化的语言。 [Booch, The Unified Modeling Language User Guide]

其中：制品 (artifact) 是指软件开发过程中产生的各种产物，如：模型、测试用例、源代码等等。

# ➤UML的构成



## ➤ UML模型

目标：

用抽象表示的图表和文字，简化现实问题描述以理解所建造的软件系统。

需求模型 use case diagram	基本模型 class diagram object diagram	行为模型 Sequence diagram Collaboration diagram state chart diagram
workflow模型 Activity diagram	组织模型 Packet diagram	实现模型 Component diagram Deployment diagram

## 用例图（use Case diagram）

描述使用系统功能的角色和系统相关的功能，是需求建模的重要工具。

## 活动图（activity diagram）

用来描述任务流程或算法过程，可用来分析系统并发事务流程。

注意：

用例图和活动图的建立，并不包含面向对象思想。

## 类图 (class diagram)

类图是系统模型的基础，描述系统的静态结构。

- 对象层：描述系统实体以及承载的系统责任
- 特征层：描述实体抽象的特征
- 关系层：实体类的固有关系

## 对象图 (object diagram)

描述类图的实例关系，为表示系统特定时刻的对象情况。**注：**在UML中使用较少

## 包图 (package diagram)

描述系统的组织模型，为控制表示的复杂性。

**时序图**（sequence diagram）

描述按时间顺序排列的对象交互。

**协作图**（collaboration diagram）

表示交互对象的行为组织结构。

**状态转换图**（state chart diagram）

描述对象在生命周期内，响应事件的状态转换过程，以及响应事件后所做的反映。

## 部署图（deployment diagram）

用来描述系统中计算结点的拓扑结构，一个系统只有一个部署图，可用来分析分布式系统。

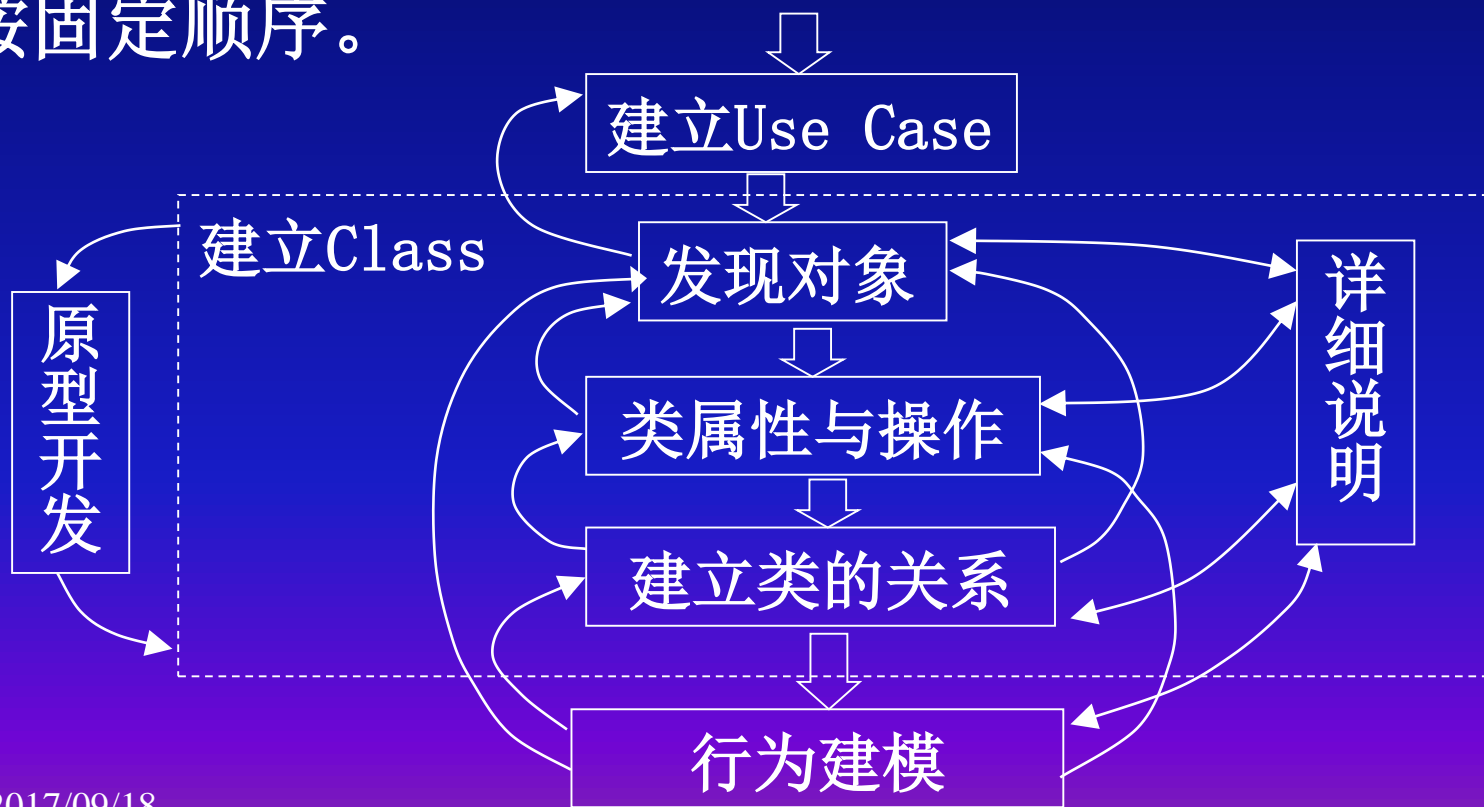
## 构件图（component diagram）

描述一组构件以及相互间的关系，是系统实现的物理建模。



## UML建模过程

面向对象的建模过程，目前没有统一标准，各种建模方法的建模过程存在着较大差别，但基本原则是统一的。强调过程中各个步骤的相对独立，不要求按固定顺序。



## ➤ 建模过程的建议

- 1) 首先进行需求确定;
- 2) 建立类图的过程可随时切换到其他活动;
- 3) 行为建模与类图交错进行;
- 4) 详细说明分散在各活动中;
- 5) 对详细说明要审查和补充;
- 5) 原型可以反复地进行;
- 6) 规模较小系统可以省略包图;

# 1. 前言

## UML简介

- UML的定义包括UML语义和UML表示法两个部分。
- (1) UML语义: UML对语义的描述使开发者能在语义上取得一致认识, 消除了因人而异的表达方法所造成的影响。
- (2) UML表示法: UML表示法定义UML符号的表示法, 为开发者或开发工具使用这些图形符号和文本语法为系统建模提供了标准。

## UML模型图的构成

- 事物(Things): UML模型中最基本的构成元素, 是具有代表性的成分的抽象。
- 关系(Relationships): 关系把事物紧密联系在一起。
- 图(Diagrams): 图是事物和关系的可视化表示。

## • UML事物

- UML包含4种事物：构件事物、行为事物、分组事物 注释事物；

- **构件事物**： UML模型的静态部分，描述概念或物理元素
- 它包括以下几种：

**类**：具有相同属性相同操作 相同关系相同语义的对象的描述

**接口**：描述元素的外部可见行为，即服务集合的定义说明

**协作**：描述了一组事物间的相互作用的集合

**用例**：代表一个系统或系统的一部分行为，是一组动作序列的集合

**构件**：系统中物理存在，可替换的部件

**节点**：运行时存在的物理元素

- 另外，参与者、信号应用、文档库、页表等都是上述基本事物的变体。

## • UML事物

- **行为事物**：UML模型图的动态部分，描述跨越空间和时间的行为。
  - 交互**：实现某功能的一组构件事物之间的消息的集合，涉及消息、动作序列、链接。
  - 状态机**：描述事物或交互在生命周期内响应事件所经历的状态序列。
- **分组事物**：UML模型图的组织部分，描述事物的组织结构。
  - 包**：把元素组织成组的机制。
- **注释事物**：UML模型的解释部分，用来对模型中的元素进行说明，解释。
  - 注解**：对元素进行约束或解释的简单符号。

- UML关系

## 1. 前言

- 依赖

- 依赖(dependency)是两个事物之间的语义关系，其中一个事物(独立事物)发生变化，会影响到另一个事物(依赖事物)的语义。

- 关联

- 关联(association)是一种结构关系，它指明一个事物的对象与另一个事物的对象间的联系。

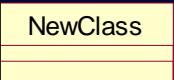
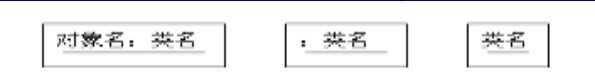

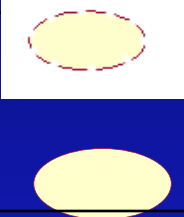
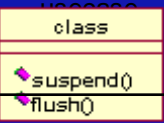
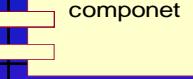


- 泛化



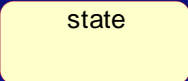
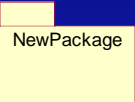





- 泛化(generalization)是一种特殊/一般的关系。也可以看作是常说的继承关系。

- 实现

- 实现(realization)是类元之间的语义关系，其中的一个类元指定了由另一个类元保证执行的契约。

•UML语法描述

类	是对一组具有相同属性、相同操作、相同关系和相同语义的对象的描述	
对象		
接口	是描述了一个类或构件的一个服务的操作集	
协作	定义了一个交互，它是由一组共同工作以提供某种协作行为的角色和其他元素构成的一个群体	
用例	是对一组动作序列的描述	
主动类	对象至少拥有一个进程或线程的类	
构件	是系统中物理的、可替代的部件	
参与者	在系统外部与系统直接交互的人或事物	

节点	是在运行时存在的物理元素	
交互	它由在特定语境中共同完成一定任务的一组对象间交换的消息组成	
状态机	它描述了一个对象或一个交互在生命期内响应事件所经历的状态序列	
包	把元素组织成组的机制	
注释事物	是UML模型的解释部分	
依赖	一条可能有方向的虚线	
关联	一条实线，可能有方向	
泛化	一条带有空心箭头的实线	
实现	一条带有空心箭头的虚线	

# 需求建模

## Use Case 用例图

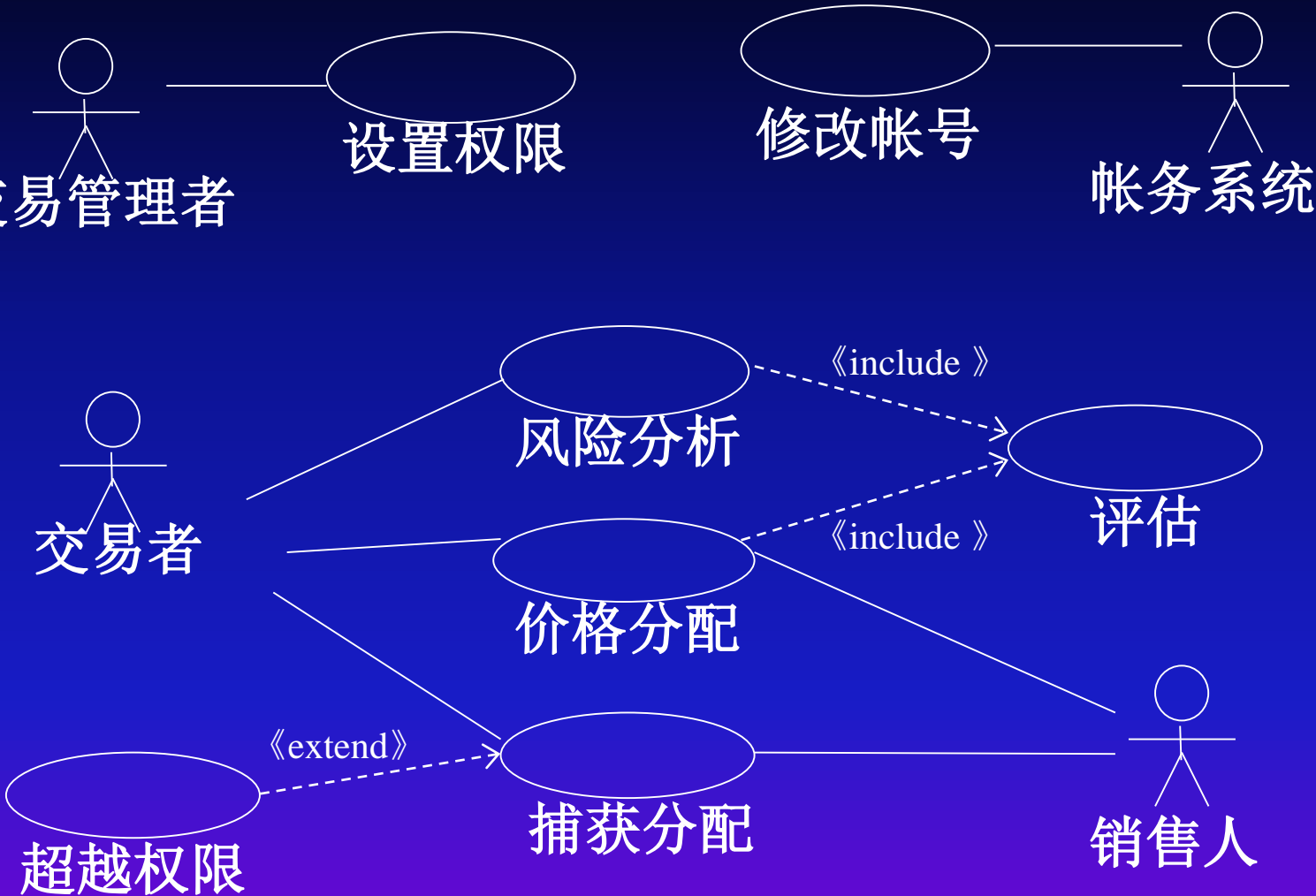
作用：

**Use Case** 用于对系统的功能及与系统进行交互的外部事物建模。

目的：

通过寻找与系统交互的外部事物，说明他们与系统如何交互，可以使用户和开发者，对系统的理解达成共识。





## ■ use case 图元素



参与者：使用系统相关功能的角色



用例：与参与者有交互的一个功能



参与者与用例的关系：表示双向，没有箭头



用例之间的关系：表示包含其中必须的功能



用例之间的关系：表示扩展可选的功能



参与者之间或用例之间的关系：表示继承

## ■ 目标

从系统边界入手，对未知空间描述。

系统边界---系统与外界事物的分界；

未知空间---待建造的未来系统；

## ■ 建立use case的切入点

切入点---与未来系统进行交互的事物；

包括：人员、设备、系统，以及操作功能

## ■参与者

使用一组密切相关功能的角色。当用这组功能与系统交互来完成某项事务时，该参与者就扮演了这个的角色。

例：

检查商品

验证顾客信用卡

收银



收款员角色

例：

查询机票信息

输入旅客信息

出票

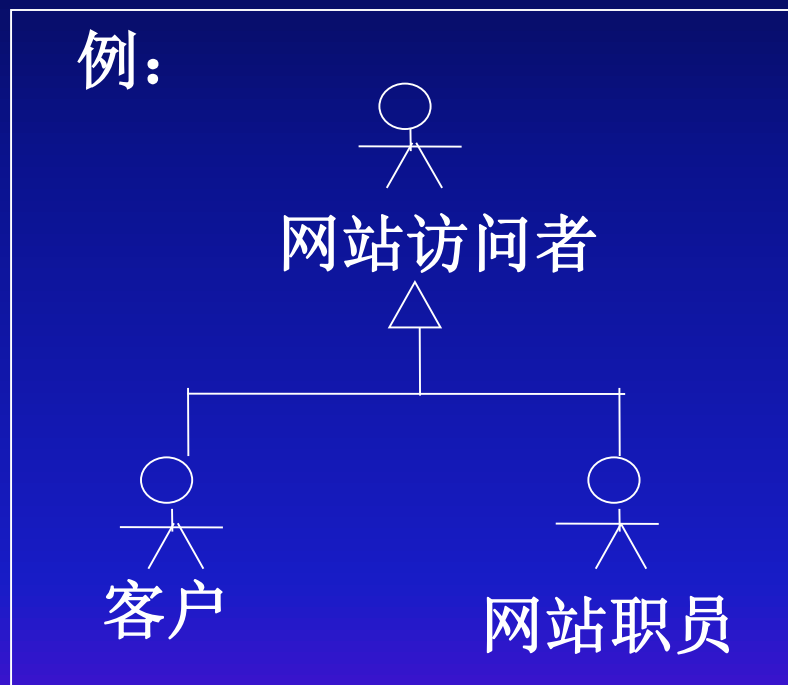


售票员角色

- 参与者可以向系统提出服务请求
- 参与者也必须接受系统的要求并做出响应
- 模型中的参与者属于系统之外的事物

## ■参与者泛化关系

某参与者具有其他参与者的共同性质，则该参与者与其他参与者构成泛化关系，其共性可以被其他参与者继承。



表示：客户和网站职员都是网站访问者，即都有网站访问者的性质。

## ■ 识别参与者

从系统的人员、设备和外部系统三个方面考虑。

**注意：**设备不包括显示器、键盘、鼠标这类标准接口设备，而是指计算机系统之外的系统使用设备，例如：传感器、受控马达等。

识别参与者的指导性策略：

- 谁是系统的操作者？
- 怎样使用系统？
- 系统的责任有哪些？
- 哪些参与者具有共同的行为？

## ■用例（case）

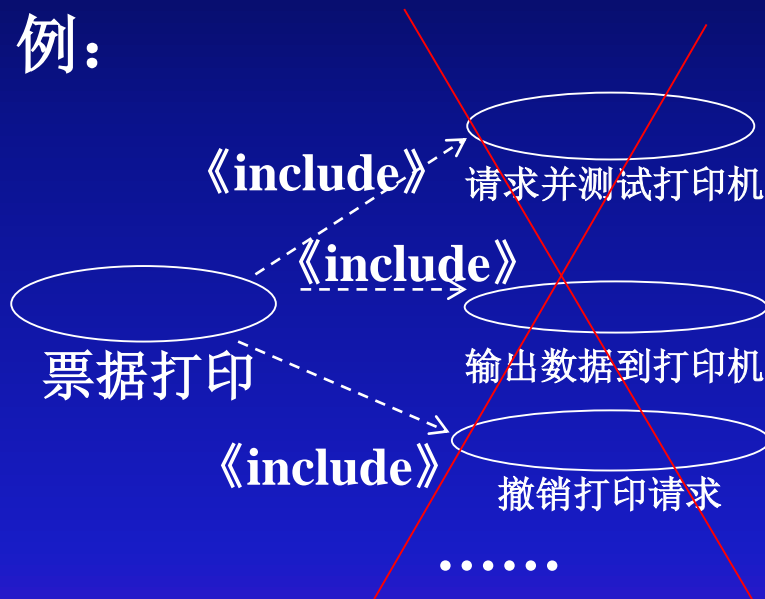
一个用例描述系统的一项**功能**，该项功能可被描述为参与者可视的一组操作，其中的每个操作表示参与者与系统的一个交互过程

用例要点：

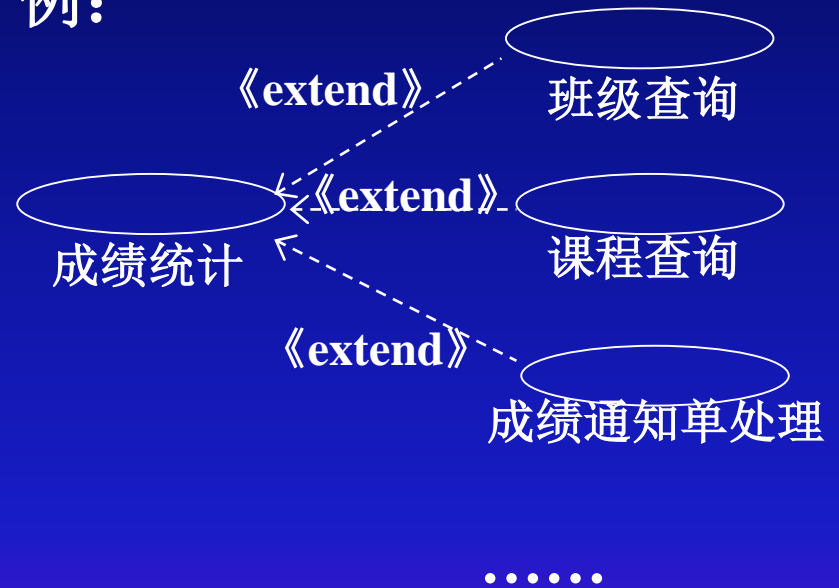
- 用例描述系统外部可见的功能需求；
- 只描述做什么，不描述怎么做；
- 多数是由参与者发起的动作；
- 也允许系统发起的动作，例如：异常情况处理；

- 用例不描述功能实现的各项处理
- 用例仅描述外部可视的交互操作

例：



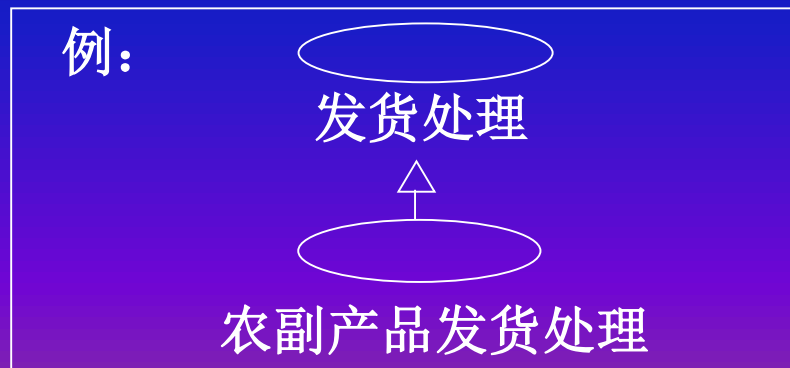
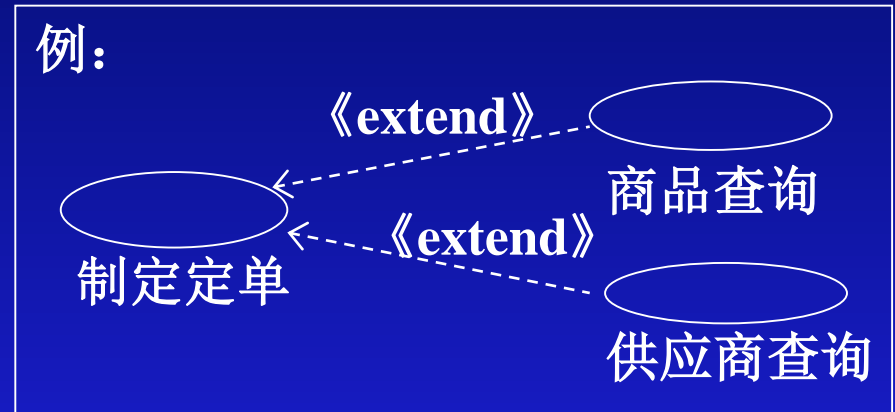
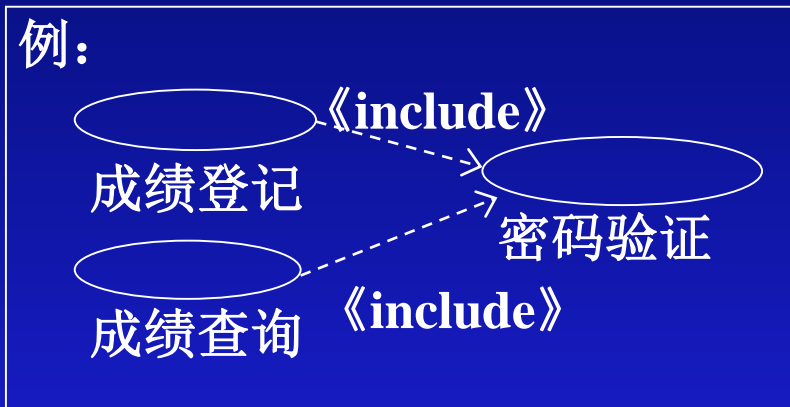
例：





## ■用例关系

- 包含关系 《include》：描述用例间具有的公用行为
- 扩展关系 《extend》：描述用例间可选的独立行为
- 泛化关系 generalization：用例之间的继承关系 ↑



## ■用例说明

对有必要说明算法的用例，可以给出详细的说明。

例：

收款

```
收款
输入本次收款的命令；
for 顾客选则商品
输入商品号；
if 选择商品多于一件
商品数量+1
end if
检索商品名称及单价
减商品存量
if 商品存量低于下限
告警商品存量不足
end if
.....
```

## ■捕获用例

参与者的责任是基础，用例是由参与者操作的。

从参与者角度分析以下问题：

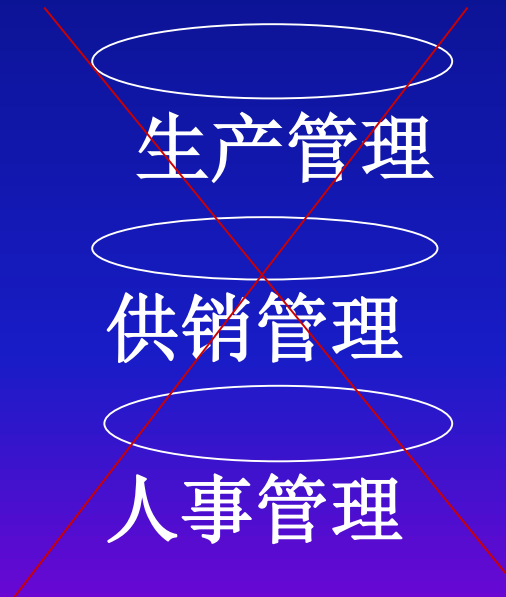
- 参与者使用系统的主要目的是什么？
- 参与者使用系统所进行的各项独立事务？
- 参与者怎样使用系统的服务？
- 本质上不同的各项活动过程有哪些？
- 除参与者外引起与系统交互有哪些？

# 捕获用例原则

1) 一个用例描述一个功能，但用例的功能不能太笼统，不应该在一个用例中针对了多个功能角色。一个用例的参与者如果为多个功能角色，应该考虑细化。

例：太笼统的系统功能划分

企业信息系统



生产计划人员  
车间管理人员

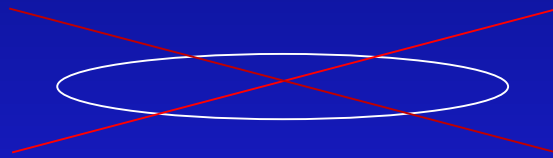
采购、计划人员  
仓库管理人员  
质量检验人员

工资考勤管理人员  
人才招聘管理人员  
离退休管理人员  
临时合同人员管理

## 捕获用例原则

2) 一个用例是在一个相对完整的时间段中发生的，应尽量避免一个用例涉及多个时间段。

不在一个时间段的用例

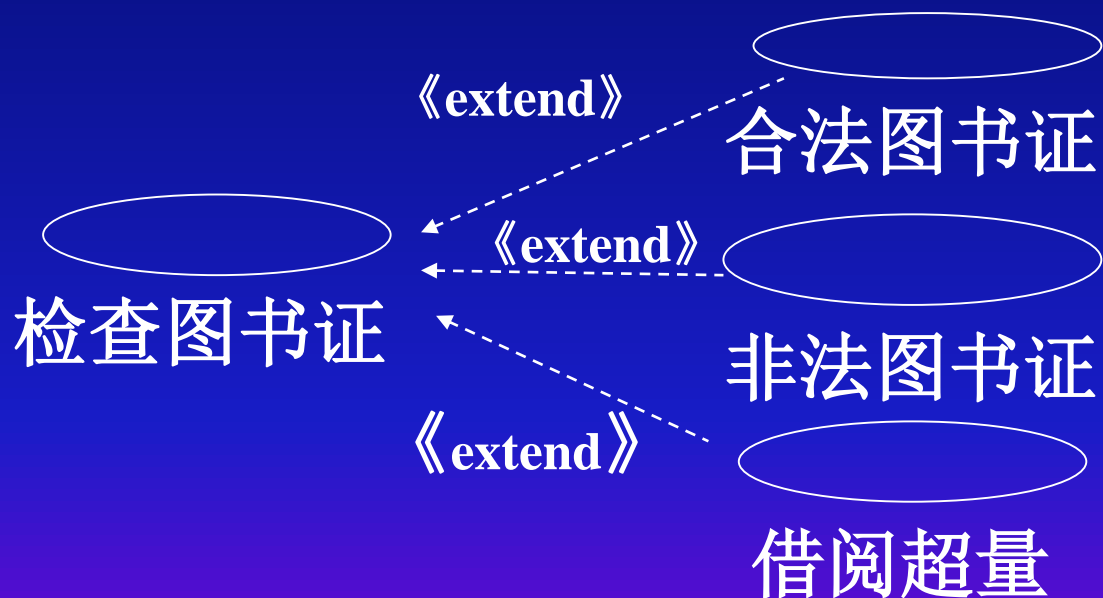


订货与退货管理

## 捕获用例原则

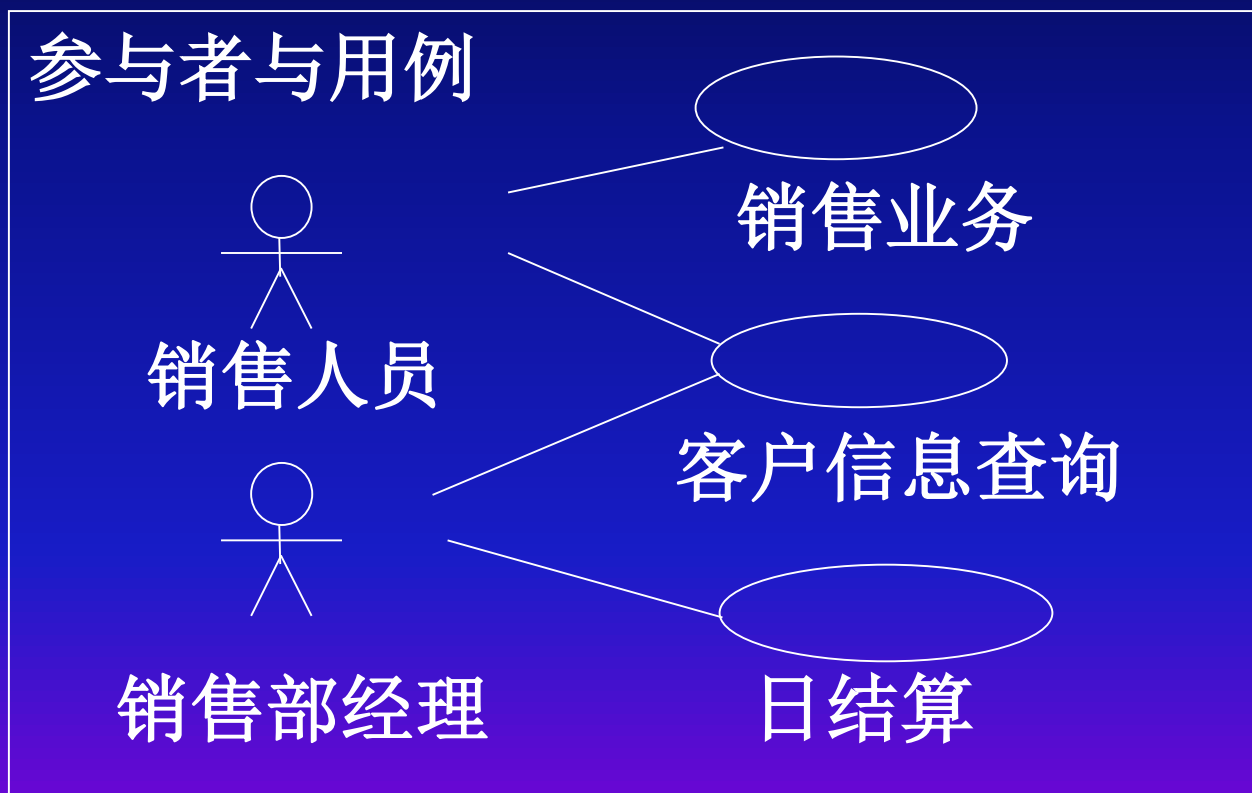
3) 用例分为主要事件流与可选事件流,

### 用例的主要和可选事件流



## 捕获用例原则

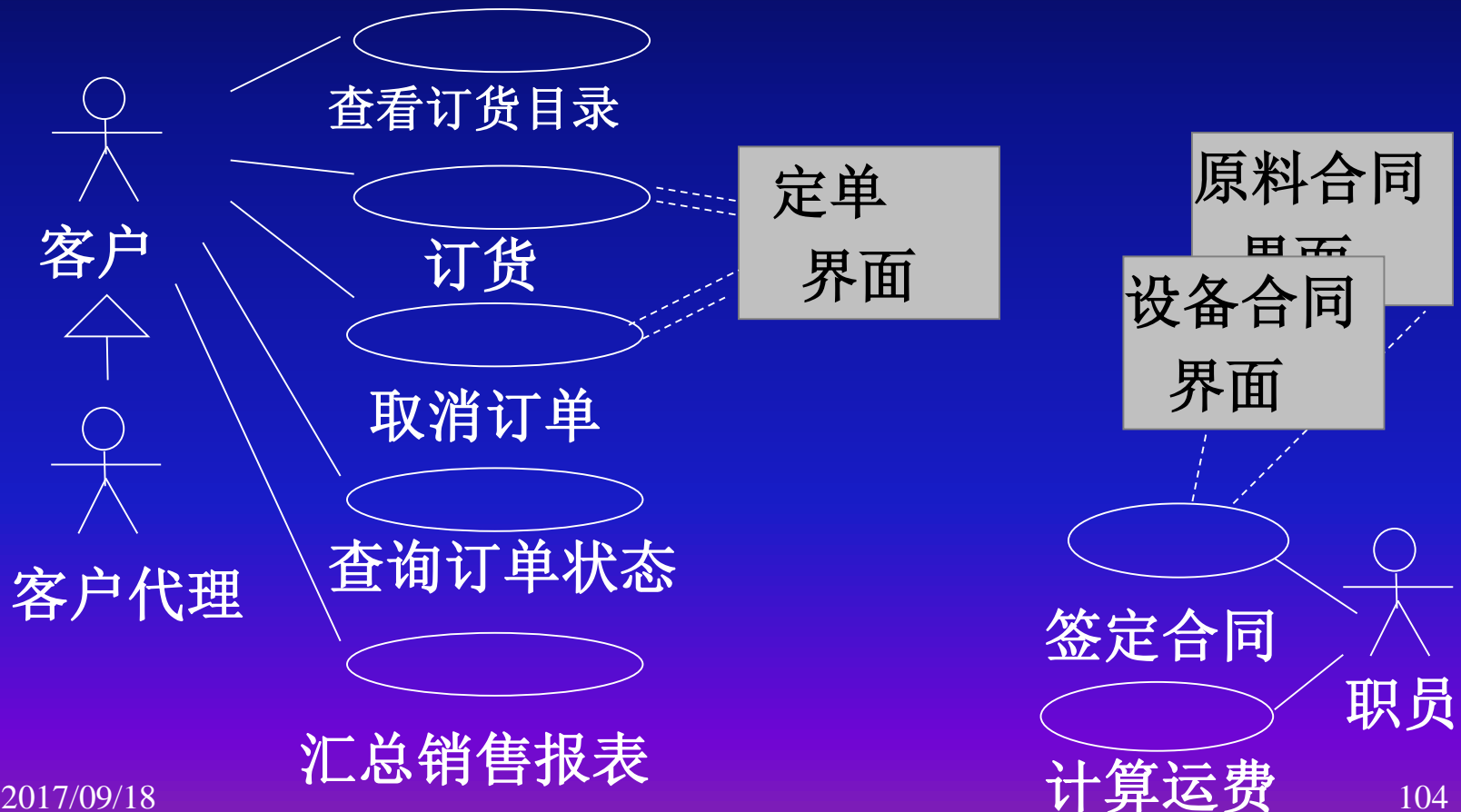
4) 参与者可以对应多个用例，用例也可以对应多个参与者。



# 捕获用例原则

5) 用例不是界面，界面也不是用例。一个用例可以对应多个界面，一个界面也可能由多个用例使用。

## 订单处理系统用例



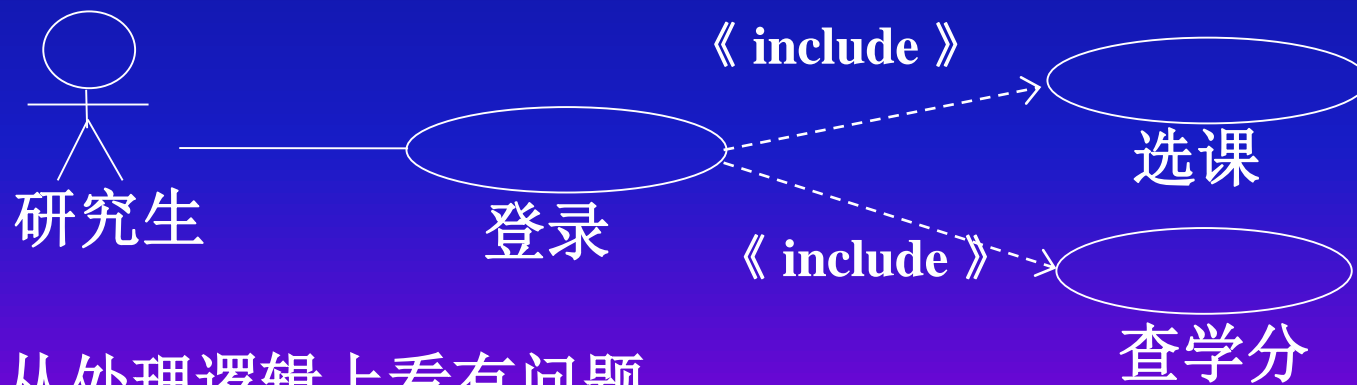


## 用例表示举例：

研究生教务系统：对登录、选课，以及查学分功能用例描述的四种表示，说明了四种不同的工作方式。

表示一：

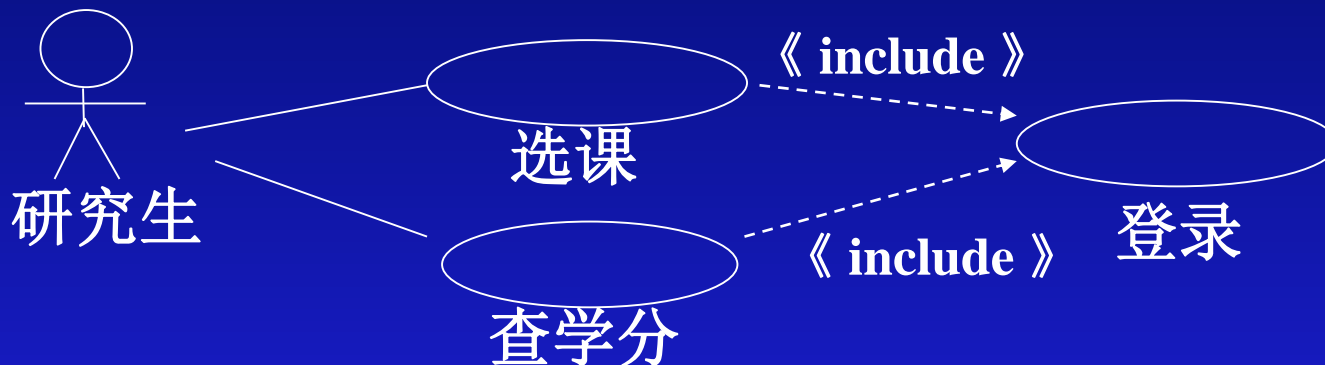
说明研究生在登录后，有两个功能是被反复使用的。两个功能作为登录主程序从属的，并且都是必须要执行的功能。



从处理逻辑上看有问题。

表示二：

研究生可选择两个功能，都包含有登录事务的需要。  
每个功能的进入都必须登录。



当有这种严格的限定时，才需要这样的考虑。

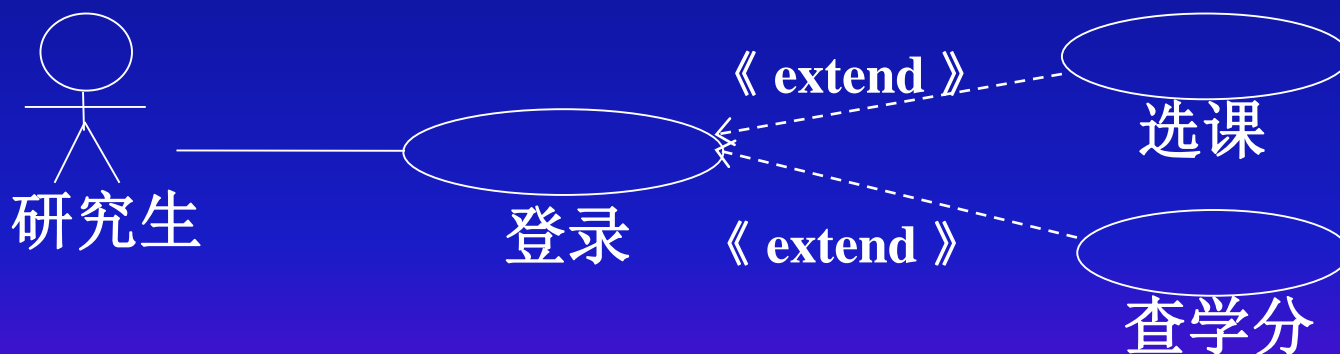
表示三:

研究生在登录后, 选择“选课”或“查学分”  
在登录中根据必要的条件, 选择是哪个功能。

条件:

if 选择“选课” then 选课

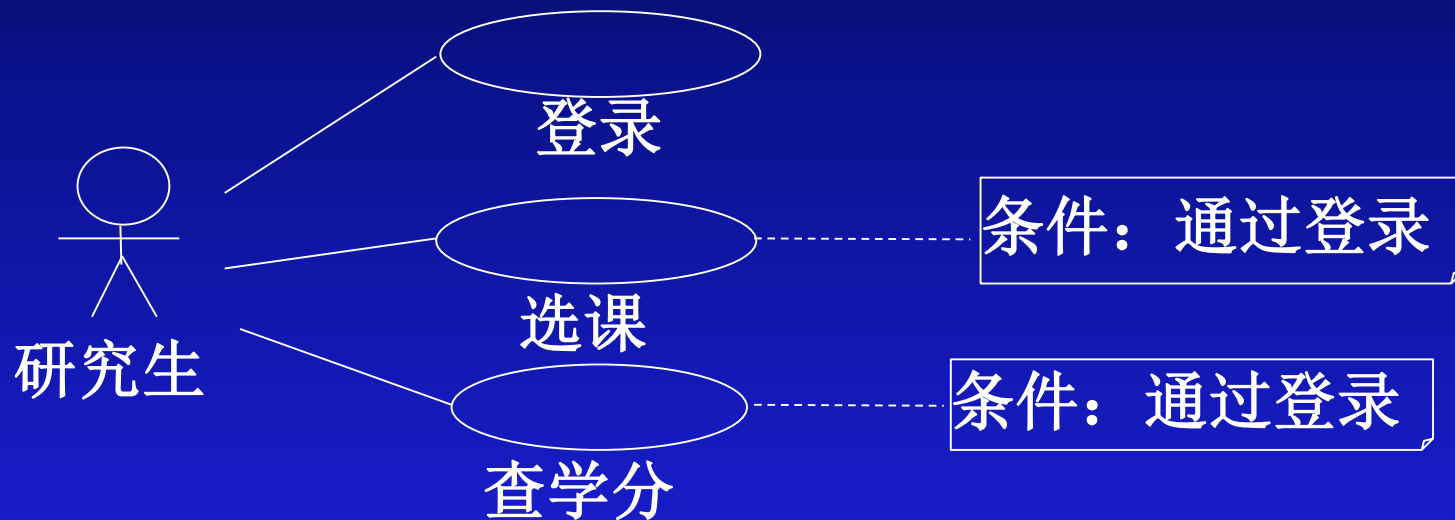
If 选择“查学分” then 查学分



若增加功能, 必须修改登录的条件和判断逻辑。

表示四：

研究生有三个独立的功能，它们之间没有直接的关系，修改或添加功能，对其它的功能没有直接影响。



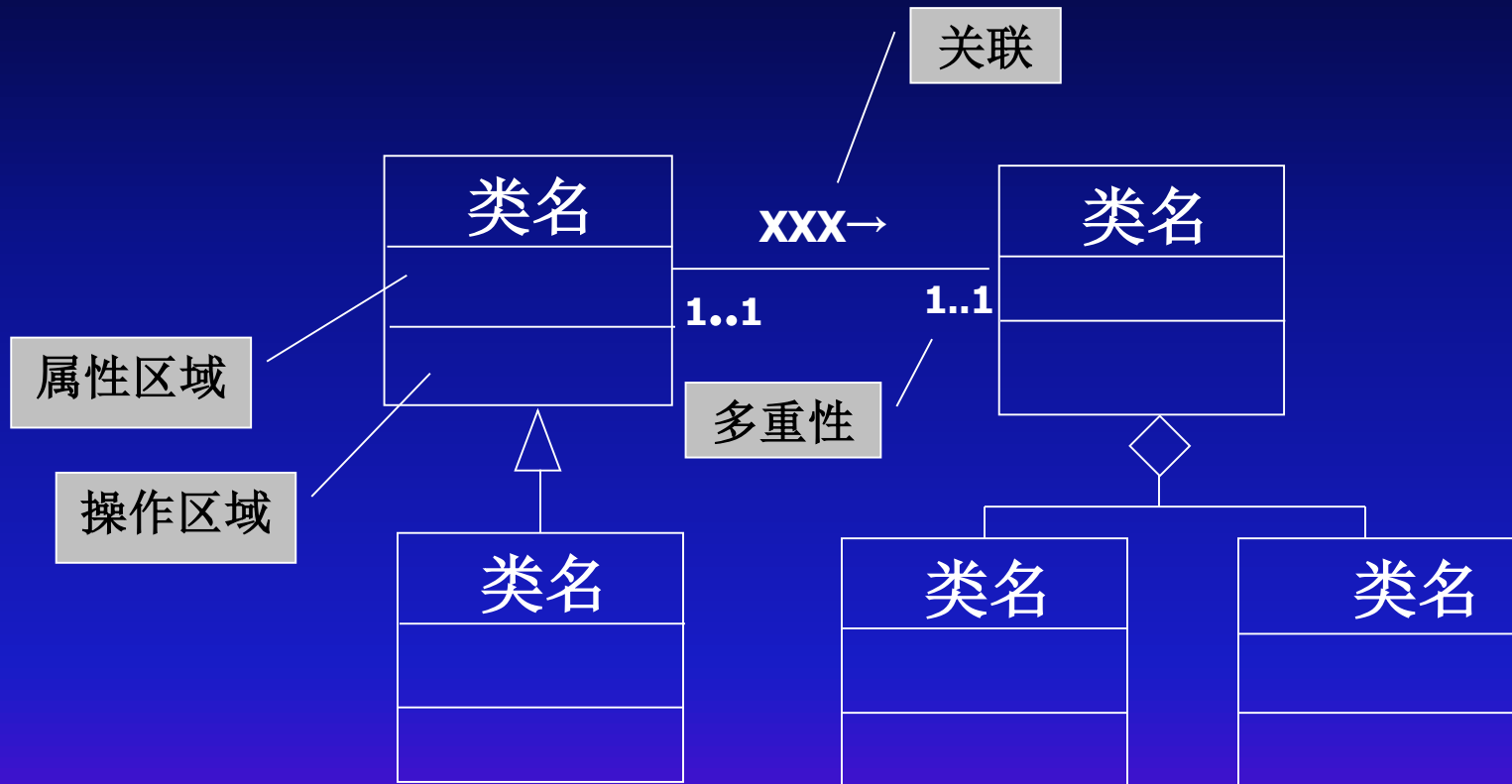
# 静态建模

# 类图 and 对象建模

类图，是系统建模过程中最重要的部分，也是花费精力最大的活动。

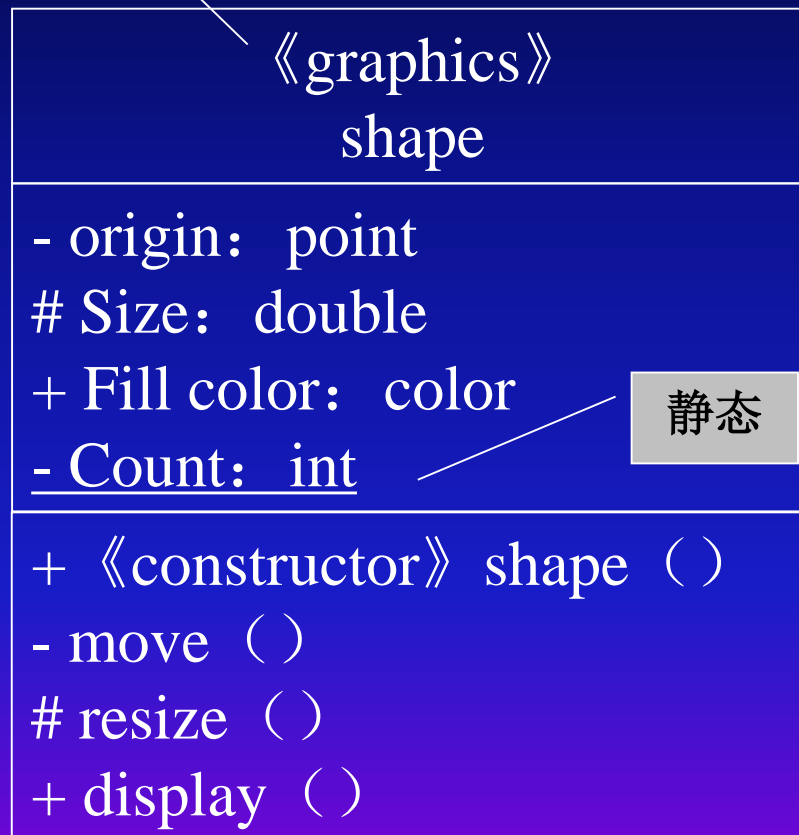
类图描述系统中各个对象之间存在的关系，表达系统的静态结构，也叫做“对象建模”。

# 类图的基本图元素：



# 类的属性和操作表示:

版型



说明:

1) 可见性(visibility):

**private -**

**public +**

**protected #**

2) 版型 (stereotype) :  
UML构造块包括: 事物、  
关系和图, 版型是在三  
种构造块基础上可定义  
的构造块。



## 类图中的事物及解释

接口--一组操作的集合, 3 只有操作的声明而没有实现;

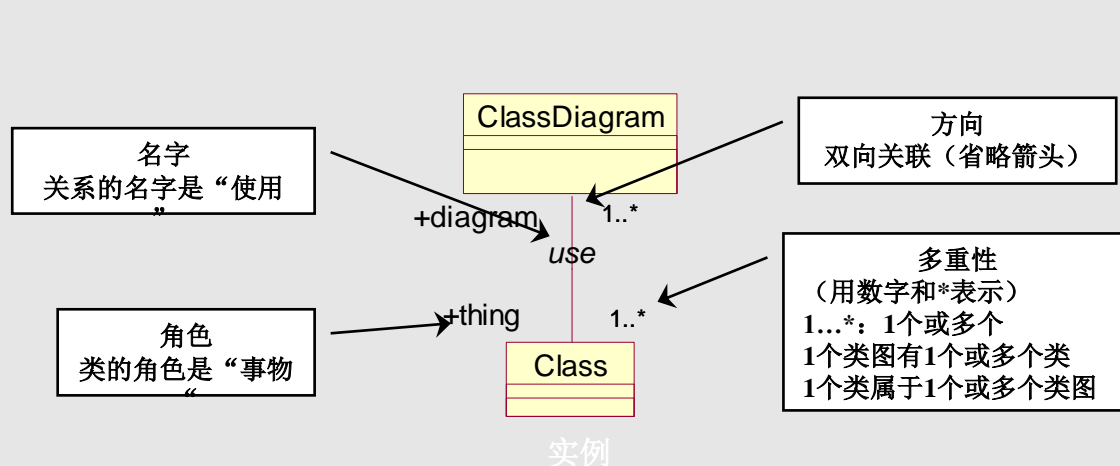
抽象类--不能被实例化的类, 一般至少包含一个抽象操作;

模版类--一种参数化的类, 在编译时把模版参数绑定到不同的数据类型, 从而产生不同的类;



# 类图中的关系及解释

关联关系--描述了类的结构之间的关系。具有方向、名字、角色和多重性等信息。一般的关联关系语义较弱。也有两种语义较强，分别是聚合与组合



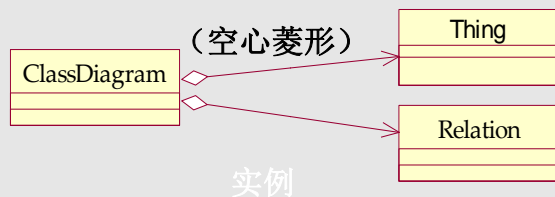
## 聚合关系

➤ 特殊关联关系，指明一个聚集（整体）和组成部分之间的关系

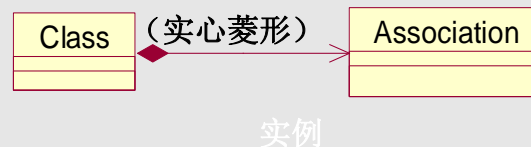


## 组合关系

➤ 语义更强的聚合，部分和整体具有相同的生命周期



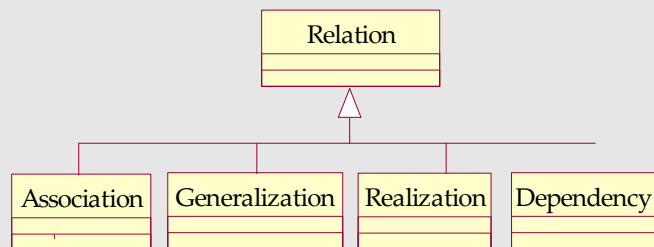
类图包含有事物和关系，类图不存在了，事物和关系还可用于其它的类图



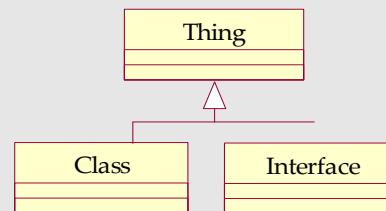
类与关联关系之间有组合关系，类不存在了，则相应的关联关系也不存在

- 泛化关系--在面向对象中一般称为继承关系，存在于父类与子类、父接口与子接口之间

UML表示法



关联、泛化、实现、依赖都是一种关系

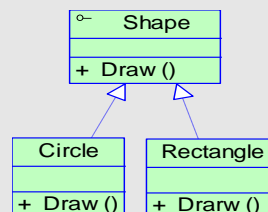


类、接口都是一种事物

## 实现关系

※ 对应于类和接口之间的关系

UML表示法

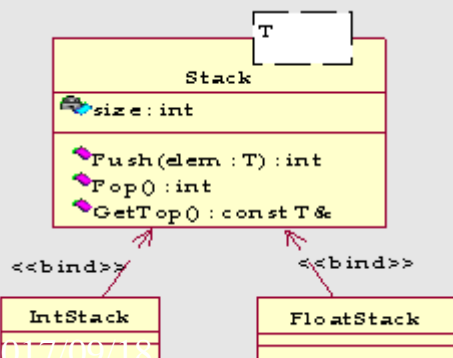


类Circle、Rectangle实现了接口Shape的操作

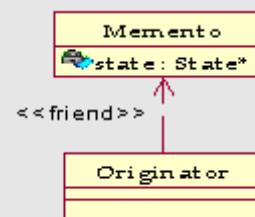
## 依赖关系

※描述了一个类的变化对依赖于它的类产生影响的情况。有多种表现形式，例如绑定(bind)、友元(friend)等

UML表示法



模板类Stack<T>定义了栈相关的操作；IntStack将参数T与实际类型int绑定，使得所有操作都针对int类型的数据



类Memento和类Originator建立了友元依赖关系，以便Originator使用Memento的私有变量state

# 类定义

类是对具有相同属性、操作、关系和语义的对象集合的描述。

# 对象定义

对象是具有明确语义边界并封装了状态和行为的实体，即它是系统中用来描述客观事物的一个实体，是构成系统的一个基本单位，由一组属性和作用在这组属性上的一组操作构成。

# 识别对象和筛选的策略

## 对象建模的基础：

在用例图完成捕获需求后，将问题域和系统责任作为基础，分析系统中的对象和类。

## 识别对象

### 方法1：

利用需求得到的问题陈述，从中挑选名词或代词，以及名词短语来识别对象和类。

## 方法2:

直接考虑现实问题中的对象，对应为系统中的对象。

## 方法3:

从系统边界外发现与系统进行交互的参与者，寻找系统处理对外接口的对象类。

## 方法4:

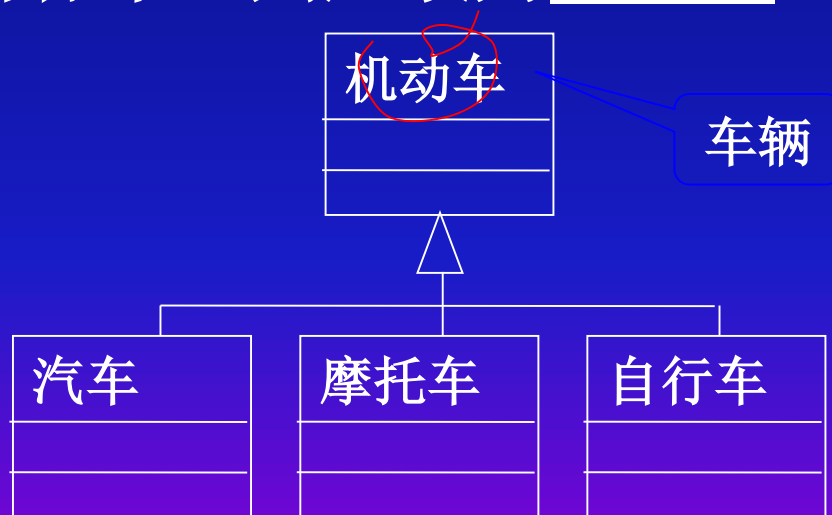
对照系统责任所要求的每项功能，确定能完成这些功能对象。

# 类的命名

类的命名应遵循的原则：

1) 类的名字应适合类所包含的每一个对象，包括它的子类对象。

例如：机动车类，适合于摩托车和汽车；如果子类还有自行车，则应改为车辆类。



2) 类的名字反映的是每个对象个体，而不是整个群体。

3) 类名应采用名词或带定语的名词，并应注意行业规范的用语，不能使用无实际意义的数字或字符作为类名。

例如：定语名词“线装书”；“出租车”类名，不要用“面的”命名类；在某化学分析软件中，用“碳酸钙沉积岩”做类名，不要用“大理石”来命名。

4) 使用适当的语言文字命名类，无论哪种文字，从编程的角度，都应该标注英文符号对照。



# 类属性及识别筛选策略

## 类的实例属性和类属性

**实例属性定义：**类的构成元素，用于描述类所对应的事物的一个性质。

- 实例属性与问题域和系统责任紧密相关；它并不是对象具有的全部属性；
- 每个实例属性有唯一的名字，其值属于给定的类型；
- 实例属性具有可见性；

**类属性定义：**类属性是描述一个类的所有对象的共同性质的一个数据项，对于该类的任何对象，它的属性值是相同的。

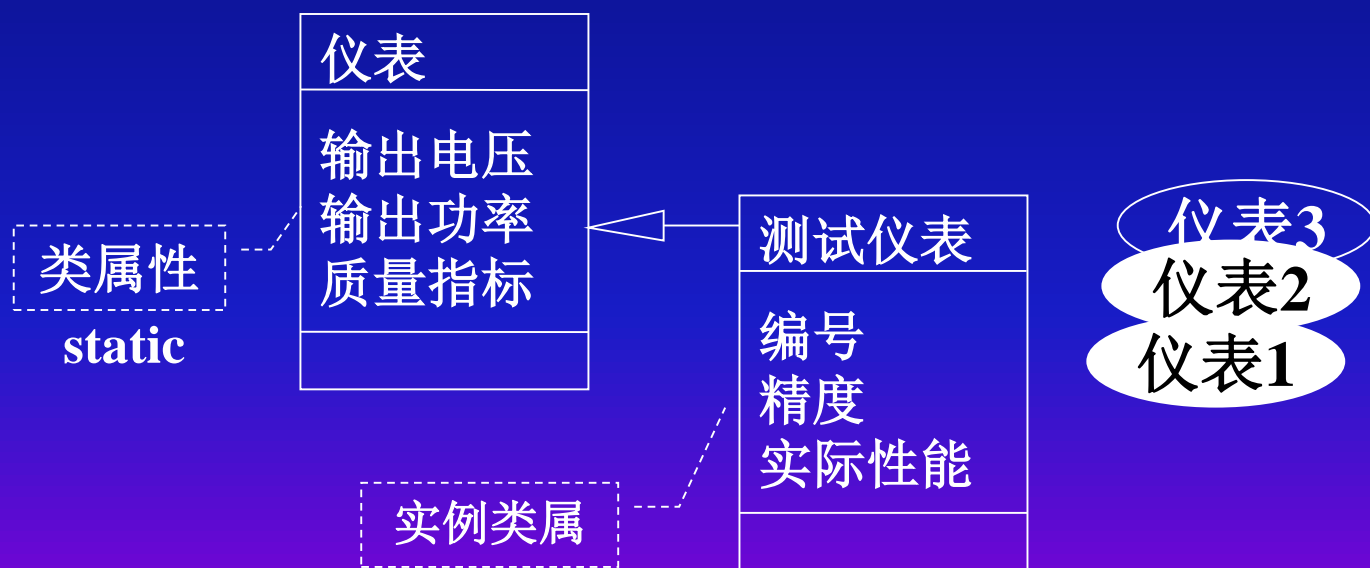
类属性，对一个类的全部对象只有一份共同的数据空间；在C++中可以用static静态成员说明符标出。

实例属性和类属性的区别和作用：

- 实例属性----用于不同性质的同种个体。
- 类属性-----用于相同性质的同种个体。

注意：在特定的问题中，有些个体同时具备某些不同的性质和共同的性质。

例如：一个仪表产品测试系统中，每台仪表的输入电压、功率及质量指标都是相同的，但每台仪表的编号、精度和实际达到的性能值却不同。



## ■识别属性的启发性策略

1) 根据常识确定对象应用的基本属性;

例如:

传感器属性, 包括编号、类型、临界值、.....

2) 根据问题域, 确定对象应该有的属性

例如:

传感器属性, 在Safehome系统中,需要安装地点、性质。同样传感器属性, 在设备管理系统中, 需要数量、购置时间。

# 识别属性的启发性策略

3) 根据系统责任, 确定对象应该有的属性

例如:

报警器的属性, 在Safehome系统中, 有自动拨打电话的责任, 因此需要延时时间属性。

4) 根据对象需要状态, 确定对象可能的属性

例如:

传感器属性, 在Safehome系统中, 有被设置在线和撤消的状态, 则需要有在线状态属性。

# 识别属性的启发性策略

5) 问题陈述中定语的词汇，可帮助确定类的属性。

例如：

“绿色的按钮”，可以确定按钮类有颜色属性。

6) 类间的关联与聚合是类的特殊属性，它表示类的一个性质值是另一个类的实例。**注意：不用属性名表示**

例如：

- 某学生的指导教师，用学生类和教师类的关联。
- 控制台的输出显示窗口，用控制台类与输出显示窗口类的聚合关系。

## ■筛选类的属性

对于初步确定的属性进行筛选的策略：

1) 对象的属性要描述对象本身固有的性质，否则即使它在系统中提供有用的信息，也不应作为对象的属性。

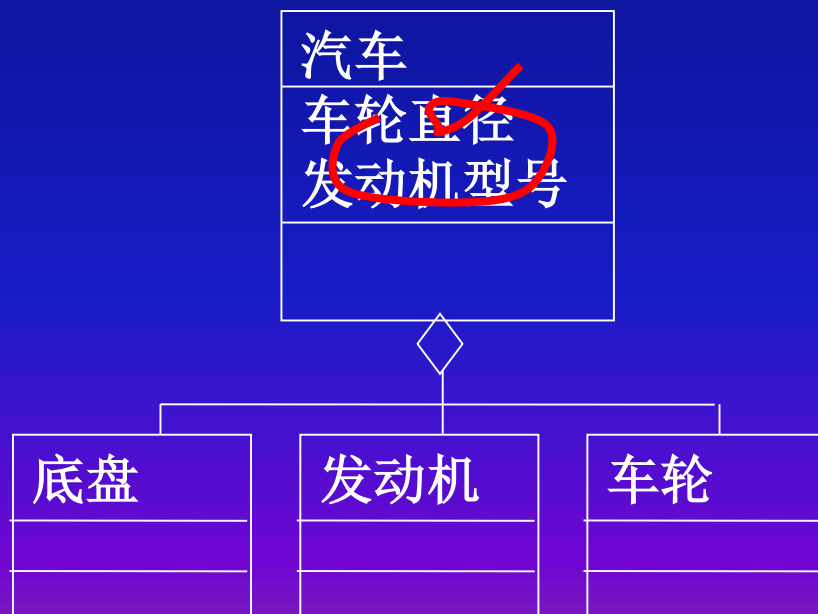
例如：课程类有主讲教师属性，把主讲教师的电话、住址等作为课程的属性会造成概念上的混乱。



## ■筛选类的属性

2) 在类间存在有聚合关系时，整体对象类的属性不要包括部分对象类的属性。

例如：汽车由底盘、发动机、车轮和驾驶仓等组成，汽车的属性与组成各部分的属性要分开，不要包含在汽车中。





## ■筛选类的属性

3) 属性应按一般的思维习惯，采用原子的（即不可分的）概念。

例如：

房间的属性门窗，应该具体分为门和窗；犯罪嫌疑人的属性服装，应该具体分为上衣和裤子。

4) 一般类定义的属性，在特殊类中不重复出现。

## ■筛选类的属性

5) 一个属性值明显可以从另一个属性值直接导出，则应该去掉。但是如果需要较复杂的计算才能导出，则可以考虑保留。

例如：有出生年月属性，不必保留年龄属性；但是，有各分项税额属性，也可以有总税率属性。

# 类操作及识别调整策略

**操作定义** 操作是类的构成元素，是类的对象被要求执行的服务。

- 内部操作和外部操作：

**内部操作**---只供对象内部的其他操作使用。

**外部操作**---响应其他对象请求时提供的服务。

- 系统行为和自身行为：

**系统行为**---系统施加于对象的行为。

例如：对象创建、复制、存储、删除

**自身行为**---简单算法行为：简单读写属性的值。

复杂算法行为：完成对象固有的行为算法。

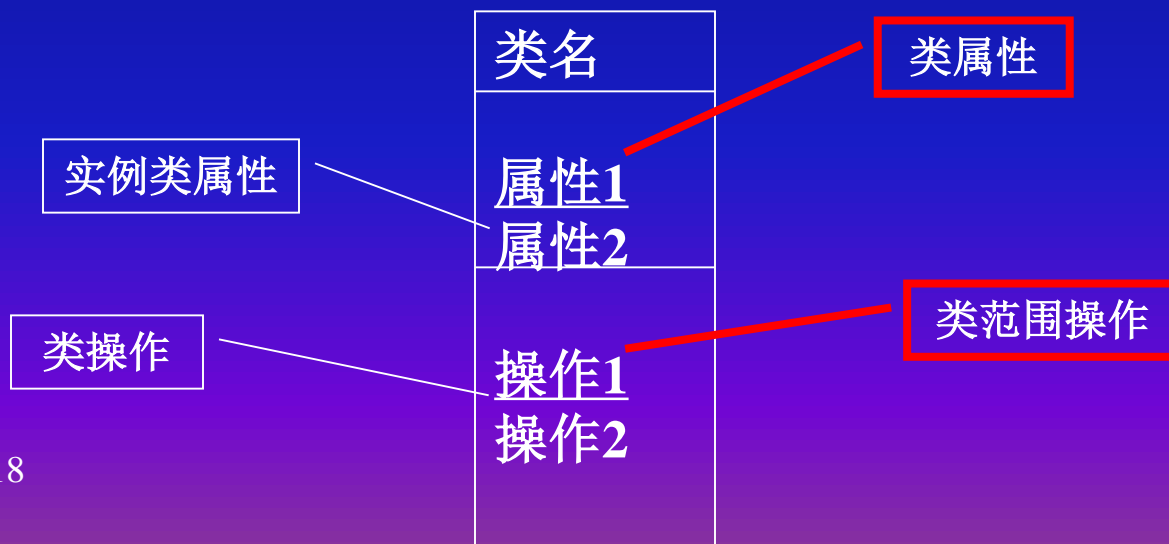
- 分析模型应该以描述复杂算法操作为主。

## 类范围操作定义

仅用于操纵类属性的操作叫做类范围操作，其余的操作叫做实例范围的操作。

类属性，对于一个类的全部对象只有一份共同的数据空间；类范围操作，仅对这样的共同空间数据进行处理。

类属性和类范围操作表示：



## 对象操作的发现策略：

- ① 问题陈述中的动词或动词短语
- ② 考虑系统责任----从需求中的每项功能，寻找可对应对象的操作。
- ③ 考虑问题域----从对象在问题域中的行为，寻找与系统责任有关的对应的操作。
- ④ 从属性值的设置----对象的属性值必须由对应的具体操作来设置。
- ⑤ 追踪操作轨迹----从特定场景的对象间交互，寻找对象必须提供的服务响应行为。
- ⑥ 分析对象状态----从对象生命周期呈现的每种状态，寻找对象保持状态的必然行为。

## ■类操作的调整原则

- 1) 如果一个操作没有系统或其他部分的请求（包括外部系统和参与者的请求），则是无用的操作，应该丢弃。
- 2) 操作应该是高内聚的。如果一个操作不仅是完成一项明确定义的、完整的、单一功能，则应该分解该操作。

## ■ 类操作的调整原则

3) 对象的操作，应该反映问题域中实际事物的行为，所以必须放在相应的类中。

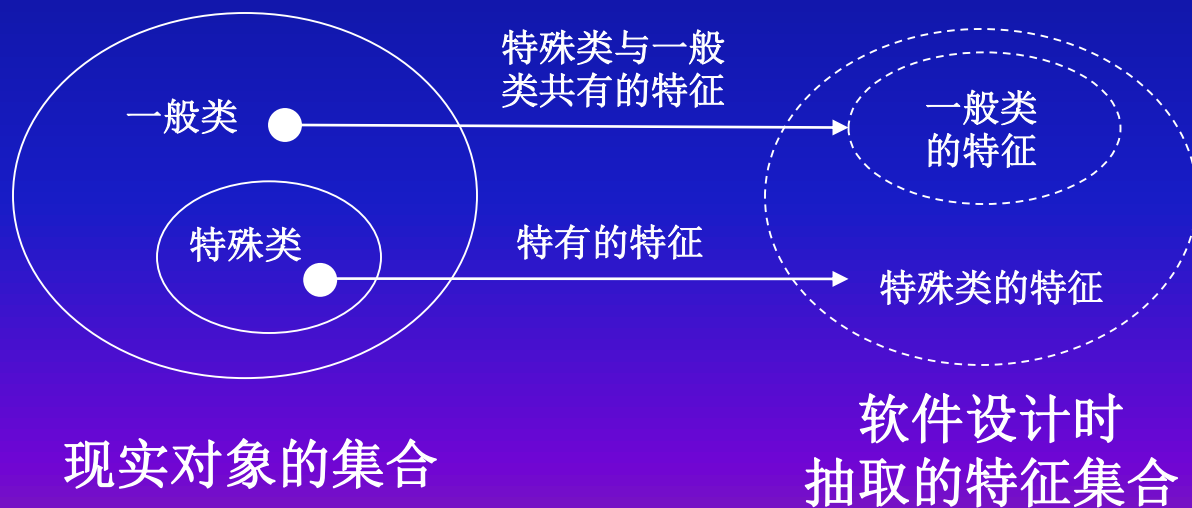
例如：售货操作，不是“货物”上的操作，而是售货员的操作

4) 操作的命名，应该是动词或动宾结构，应尽量清楚地反映操作的内容。

# 类的继承关系

**继承关系定义** 如果类A具有类B的全部属性和全部操作，而且还具有自己特有的一些属性或操作，则A叫做B的特殊类，B叫做A的一般类，A与B之间的关系称为继承关系。

## 一般类与特殊类



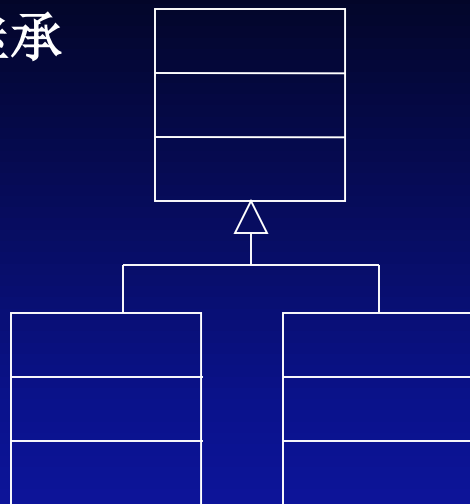


## 继承的性质：

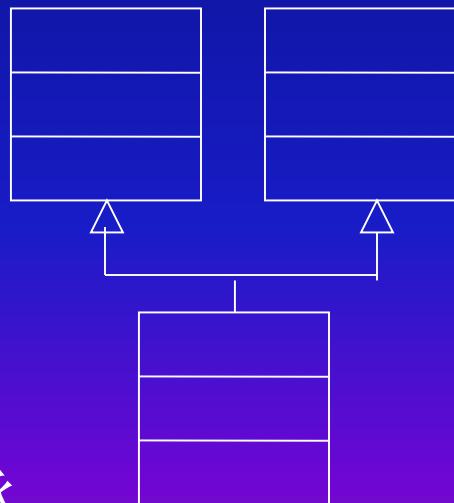
- 非对称性----继承关系语义为“is a”关系，如果类A是类B的后代，则类B不能是类A的后代，因为，B将不会与类A有“is a”关系。
- 传递性----如果类A继承类B，类B继承类C，则类A继承类C。
- 单继承----特殊类只直接地继承一个一般类。
- 多继承----允许特殊类可直接地继承两个以上的一般类。

# 继承关系表示

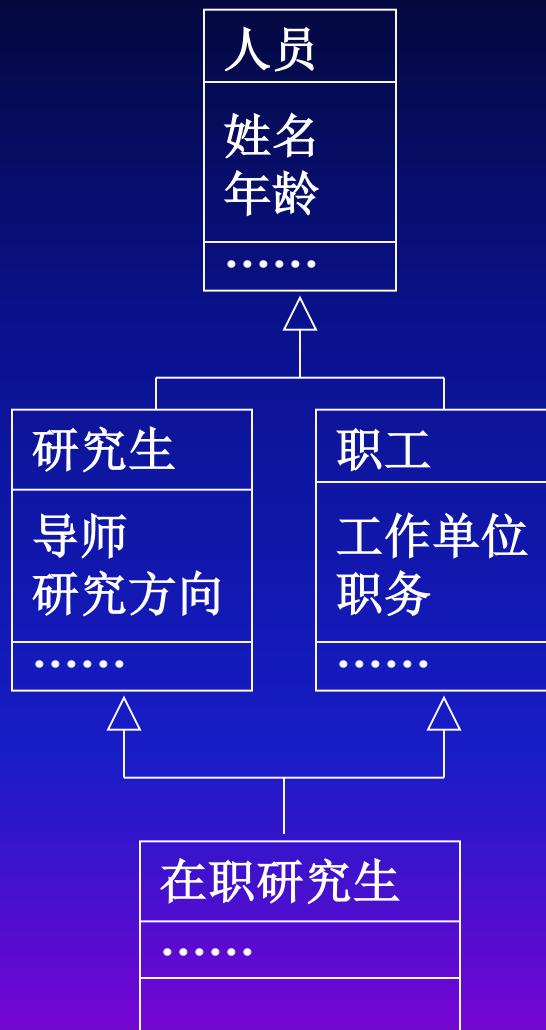
单继承



多继承



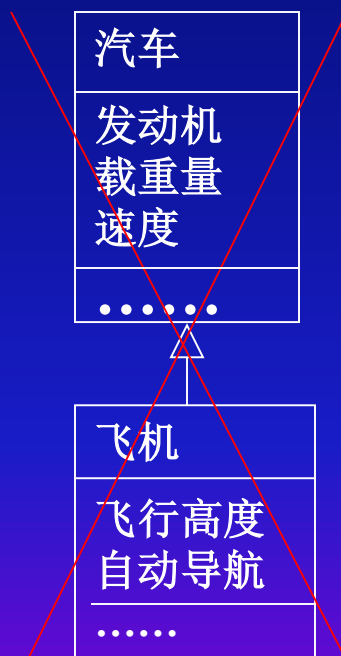
多继承示例



## ■确定继承的策略和原则

1) 按照问题领域的分类知识以及事物的分类常识, 寻找相应的继承关系。

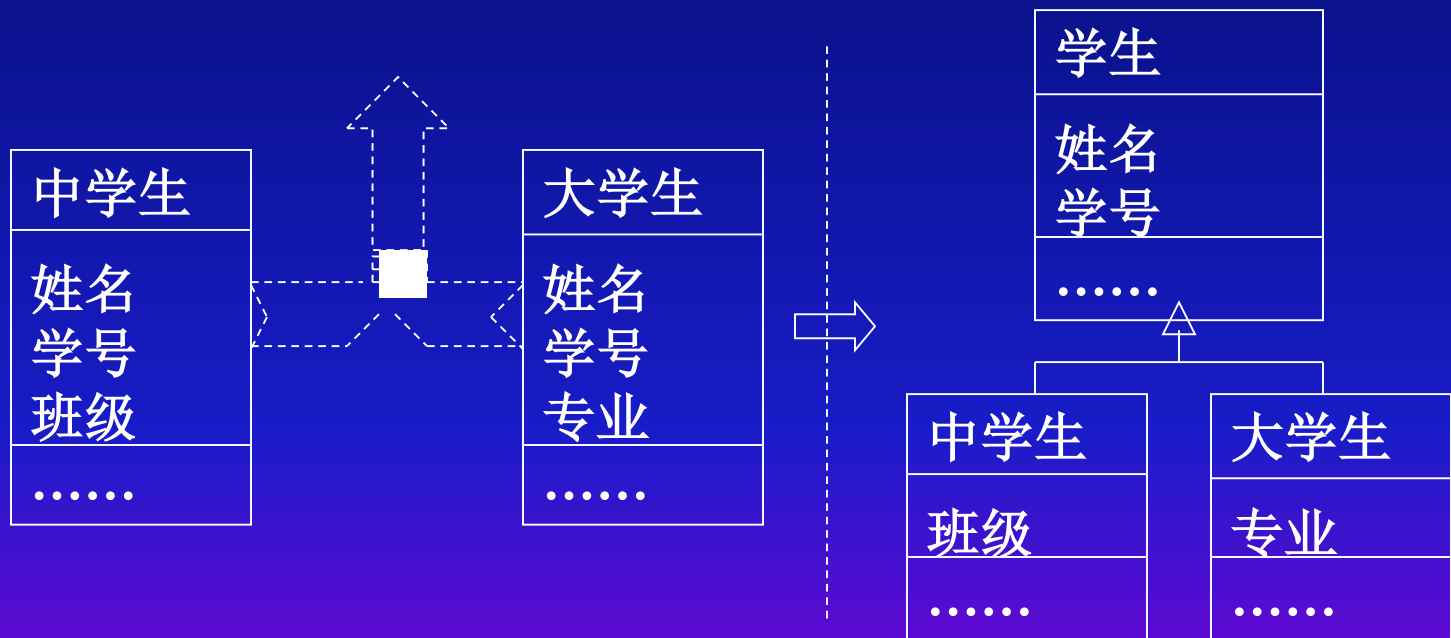
违反常识的继承关系



# 确定继承的策略和原则

2) 寻找类之间的包含关系。如果一个类是另一个类的子集，并且，类之间的语义有“is a”的关系，则它们是继承关系。

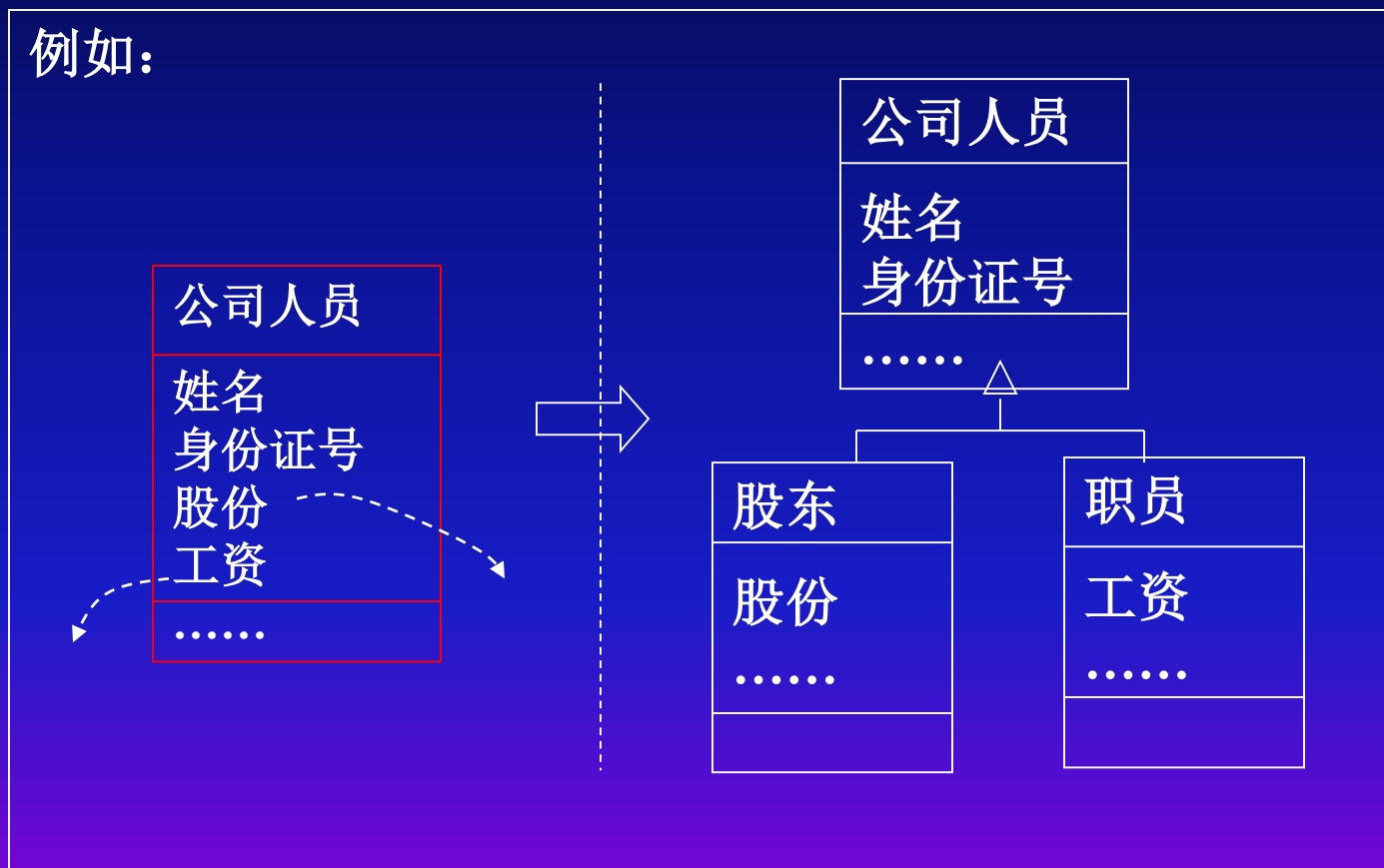
例如：



## 确定继承的策略和原则

3) 判断类的属性与操作是否适合于所有的对象，从中发现特殊类，建立继承关系。

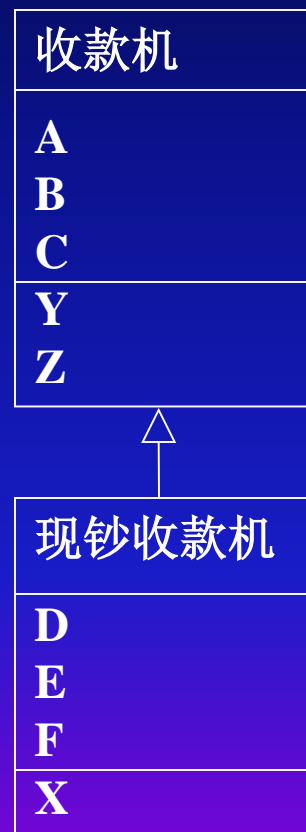
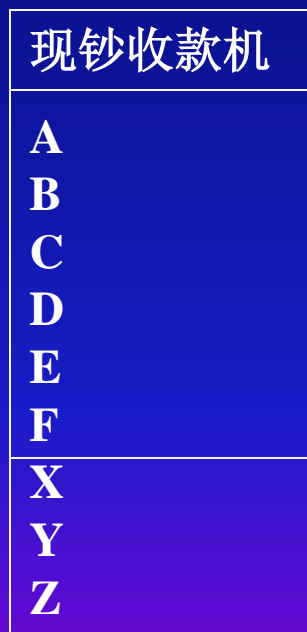
例如：



# 确定继承的策略和原则

4) 为适应领域范围内更多的复用，分解类为继承关系。

为复用建立继承关系：



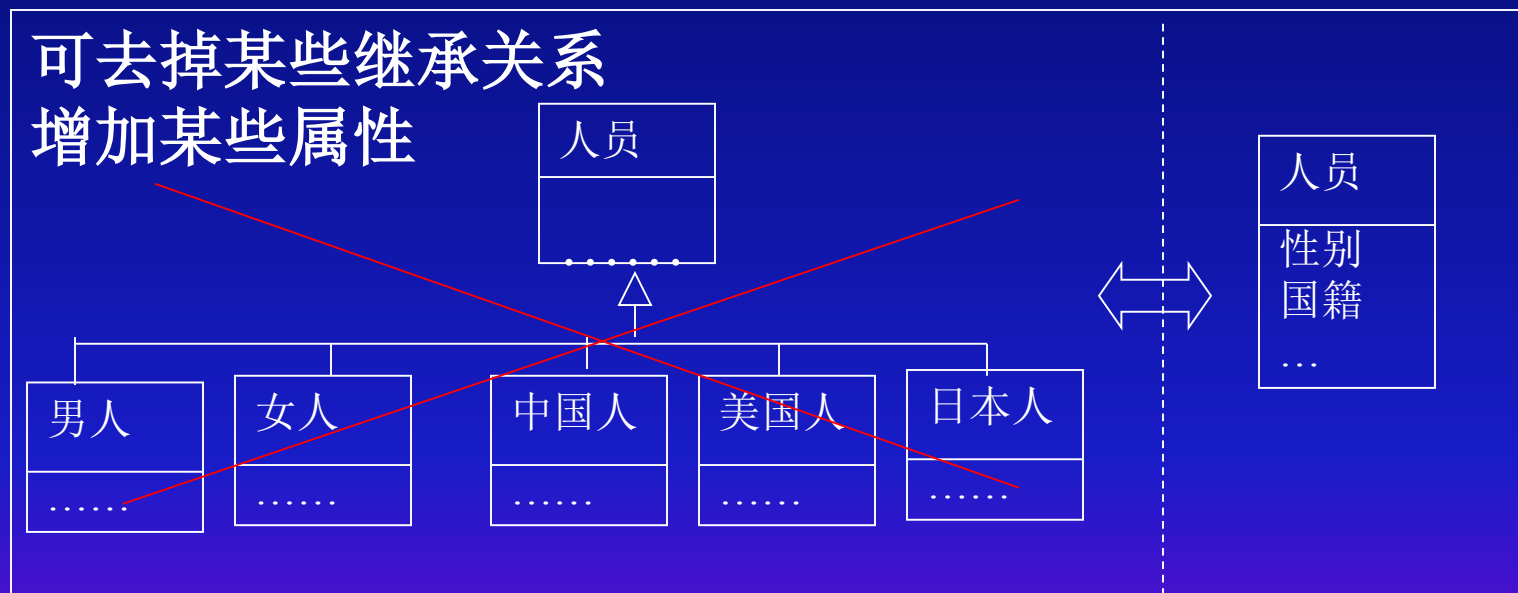
## 确定继承的策略和原则

5) 若特殊类不具有自己特殊的属性和操作，则可以去掉该特殊类。



## 确定继承的策略和原则

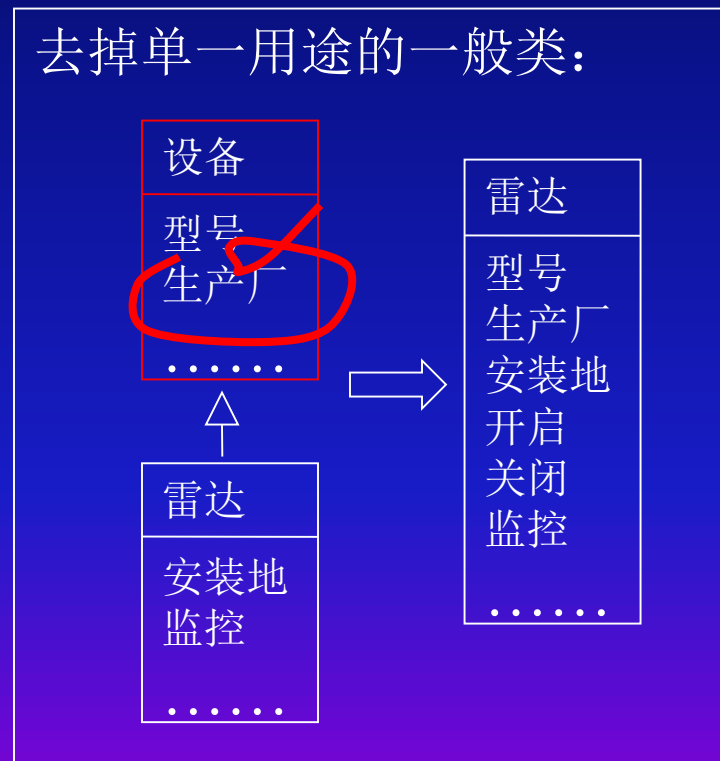
6) 如果某些特殊类之间的差别，可以通过某属性值来体现，则可以去掉这些特殊类。





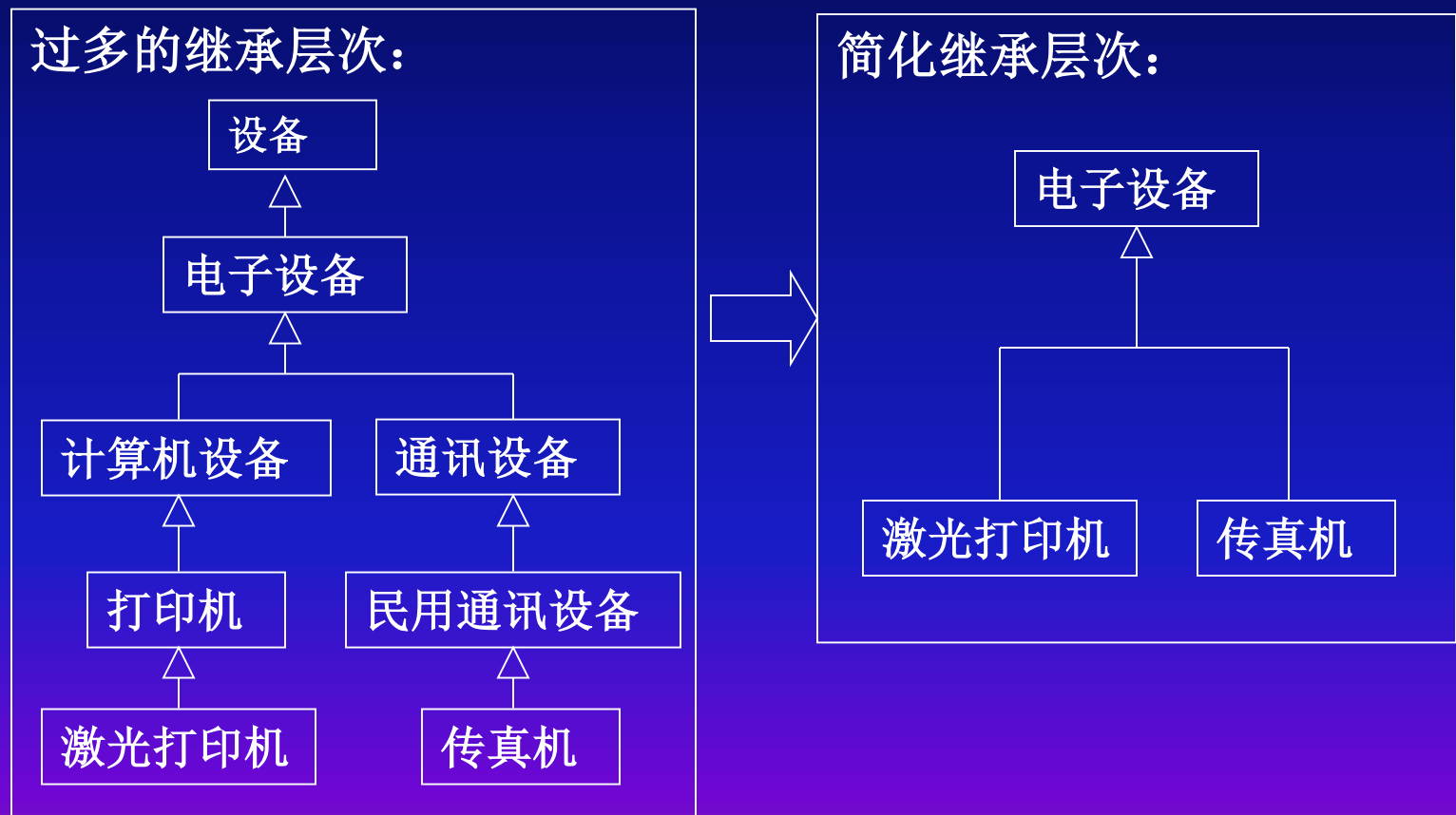
## 确定继承的策略和原则

7) 如果只有唯一特殊类，且该一般类不用于创建对象，也不用于复用，则取消该一般类。



# 确定继承的策略和原则

8) 应该尽量避免类的层次过深，使系统结构过于复杂。



# 建立类的关联

- 1) 类的关联
- 2) 关联的重数
- 3) 对象链
- 4) 关联角色
- 5) 关联类
- 6) N元关联
- 7) 关联的限定符
- 8) 聚合关联
- 9) 类的依赖关系表示

# 1) 类的关联

关联的基本概念：

关联-----表示对象类之间的静态关系。

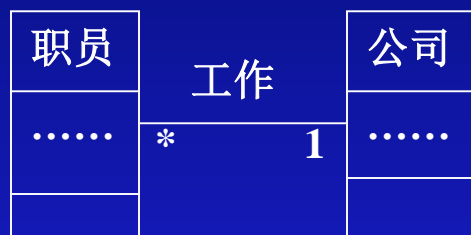
静态关系----表示对象之间固有的联系。

要点：

- 两个类之间的关联，实质上是通过属性来表示对象之间的联系，即一个类的对象属性值是另一个类的对象实例（用指针实现）。
- 静态关联与系统责任相关，如果这些关系是系统责任目标的必要信息，则需要表达它们。

**关联定义** 如果一个类的对象与另一个类的对象之间有语义连接关系，则这两个类之间的语义关联就是关联。

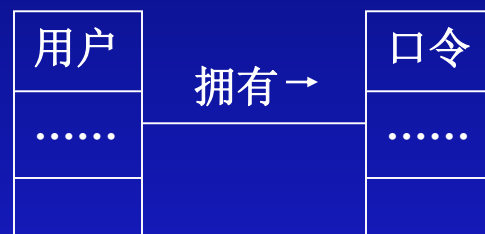
双向关联例：



自身关联例：



单向关联例：



关联一般是双向的，若有单向限制用箭头表示。

## 2) 关联的重数

**重数定义** 关联端上的重数，表示一端的对象需要另一端对象的个数。

重数的表示：A端的一个对象需要B端Mb个对象  
B端的一个对象需要A端Ma个对象



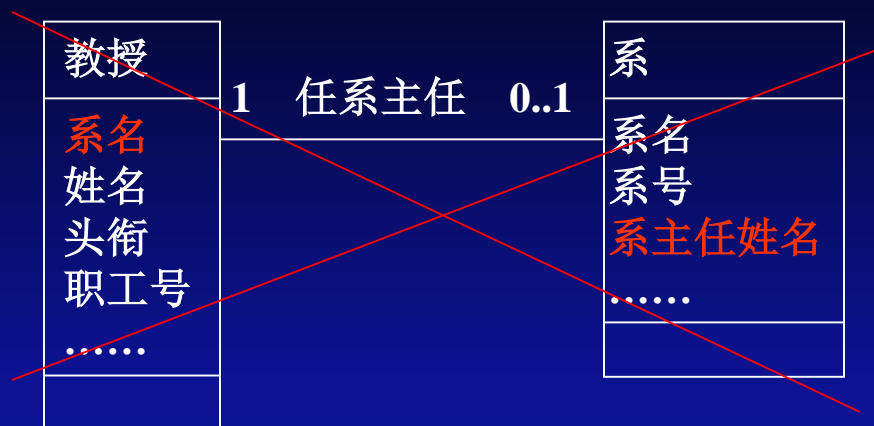
重数值的表示：

值	表 示
n ..m	表示与类的n至m个对象关联

例：

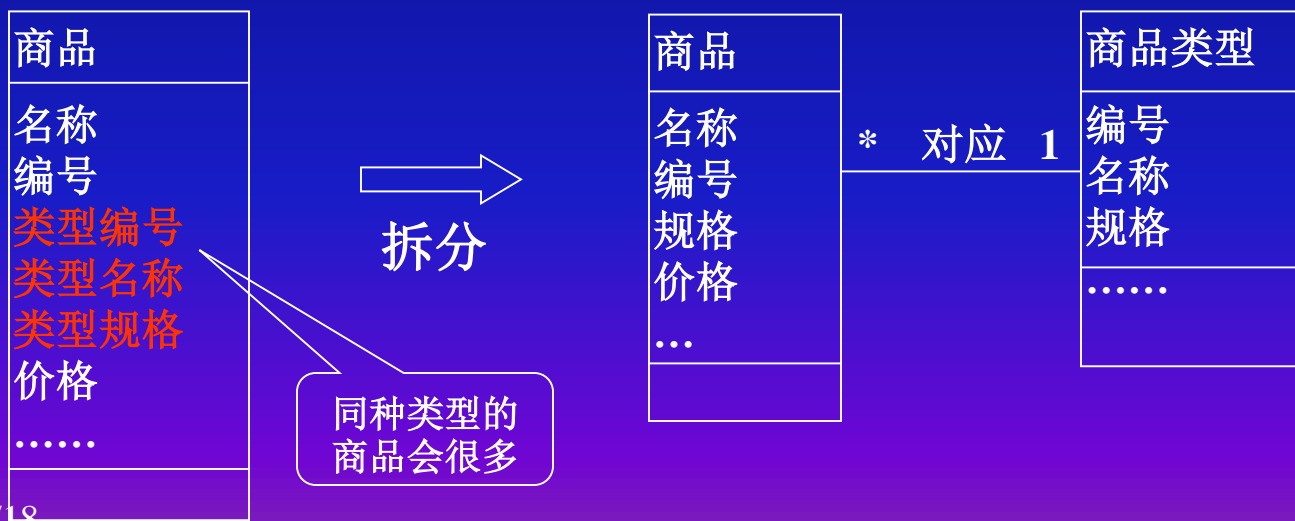


在类中不用表示关联的具体项（外键）。

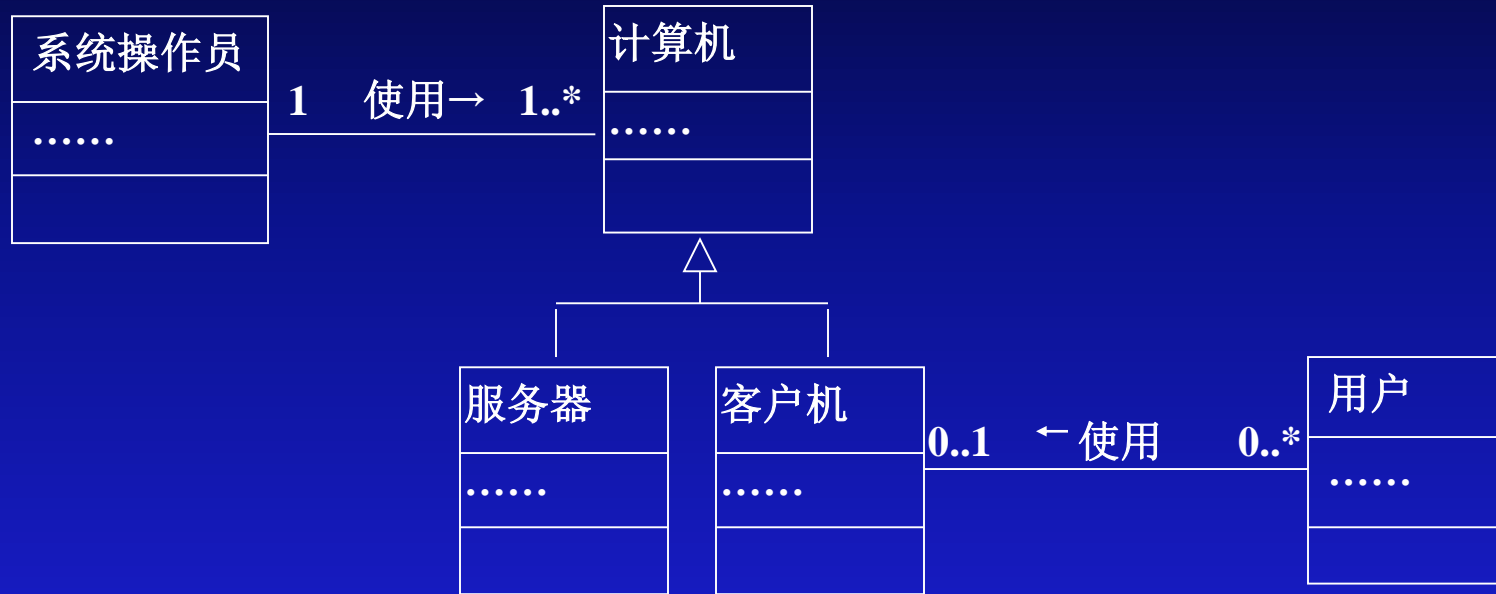


有数据冗余的类，可拆分类并关联它们。

例：

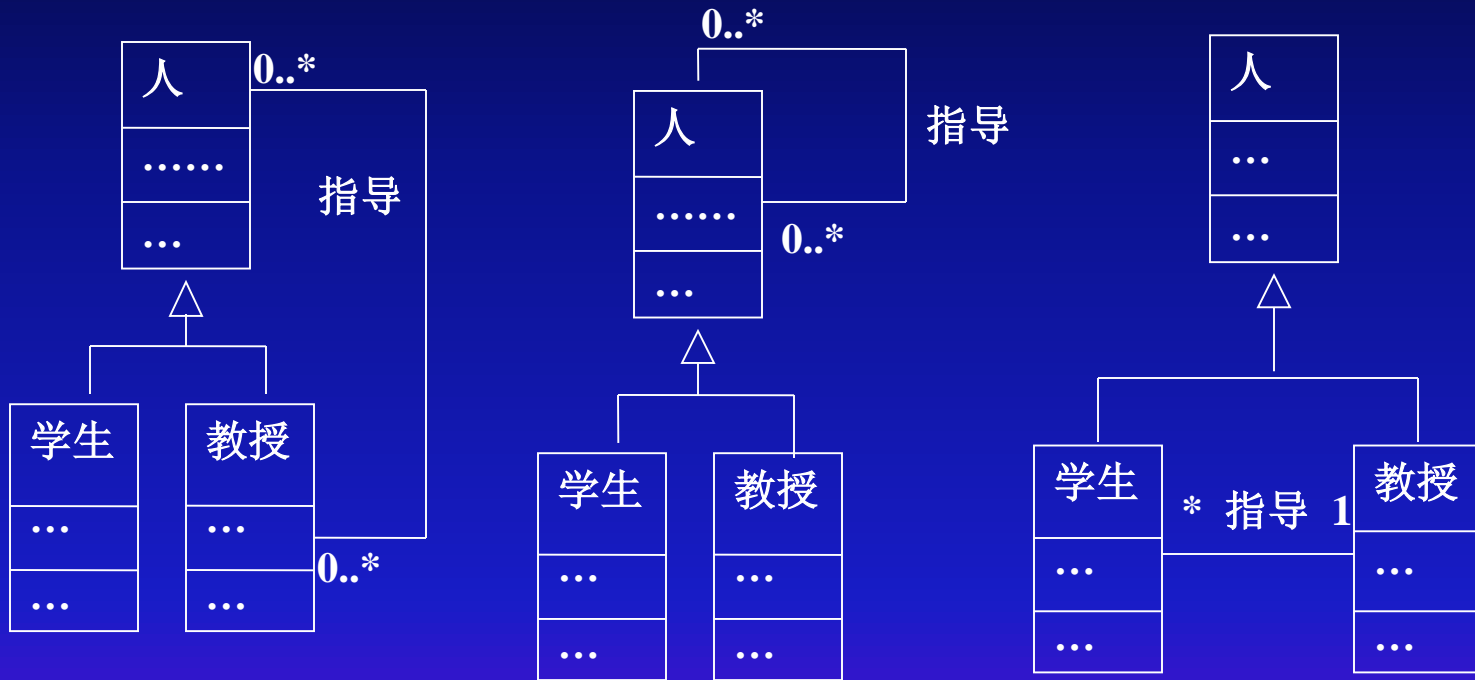


## 特殊类直接继承一般类的关联:



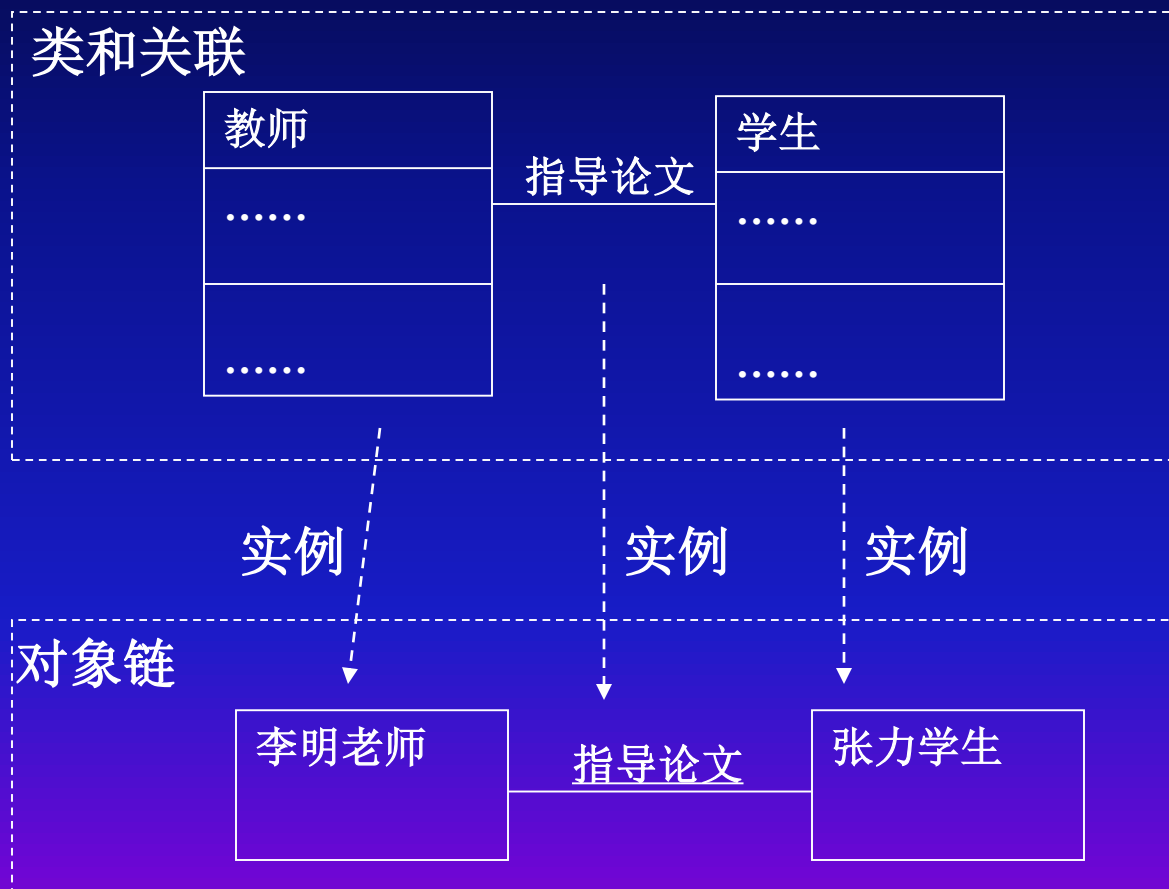


同样的类，由于关联不同，可以有不同的表示：



### 3) 对象链

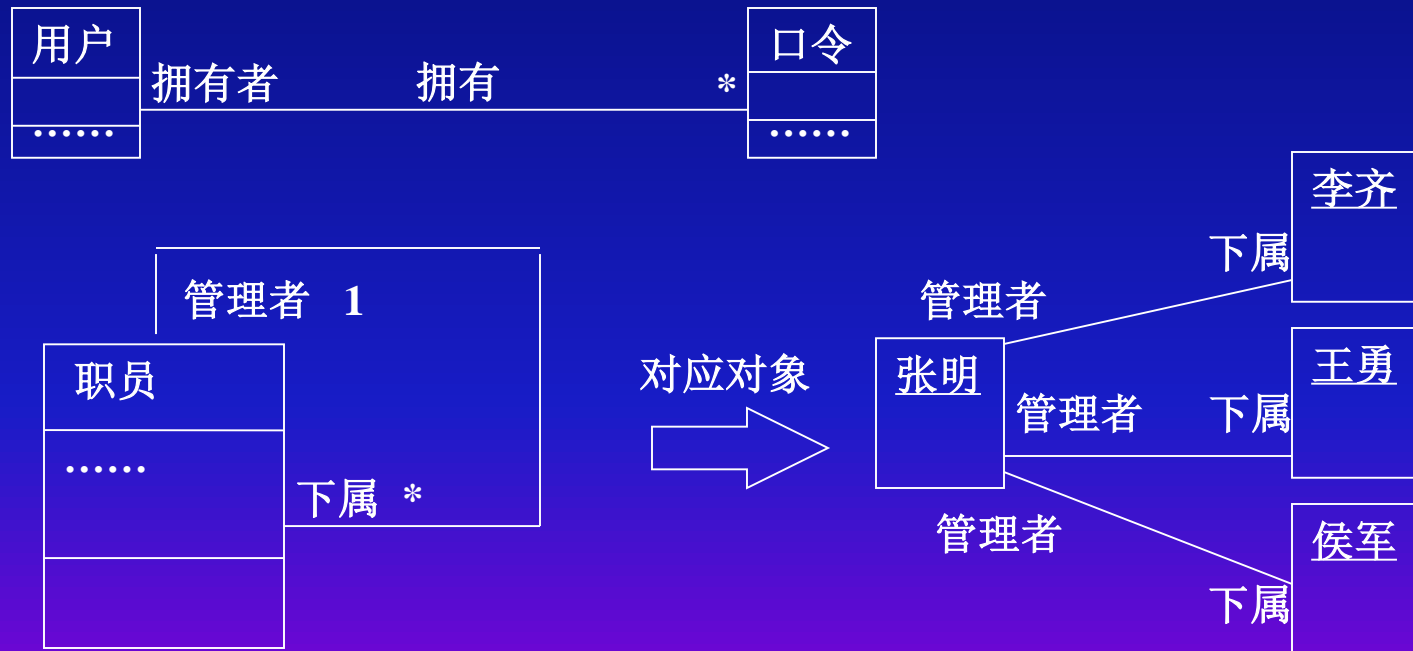
**链定义** 链是关联的实例，是对象间的语义连接。



## 4) 关联角色

关联角色----表示需要明确的一个角色属性。在关联的端点，可以表示相连接类所扮演的角色，关联角色的名字称为“关联角色名”。

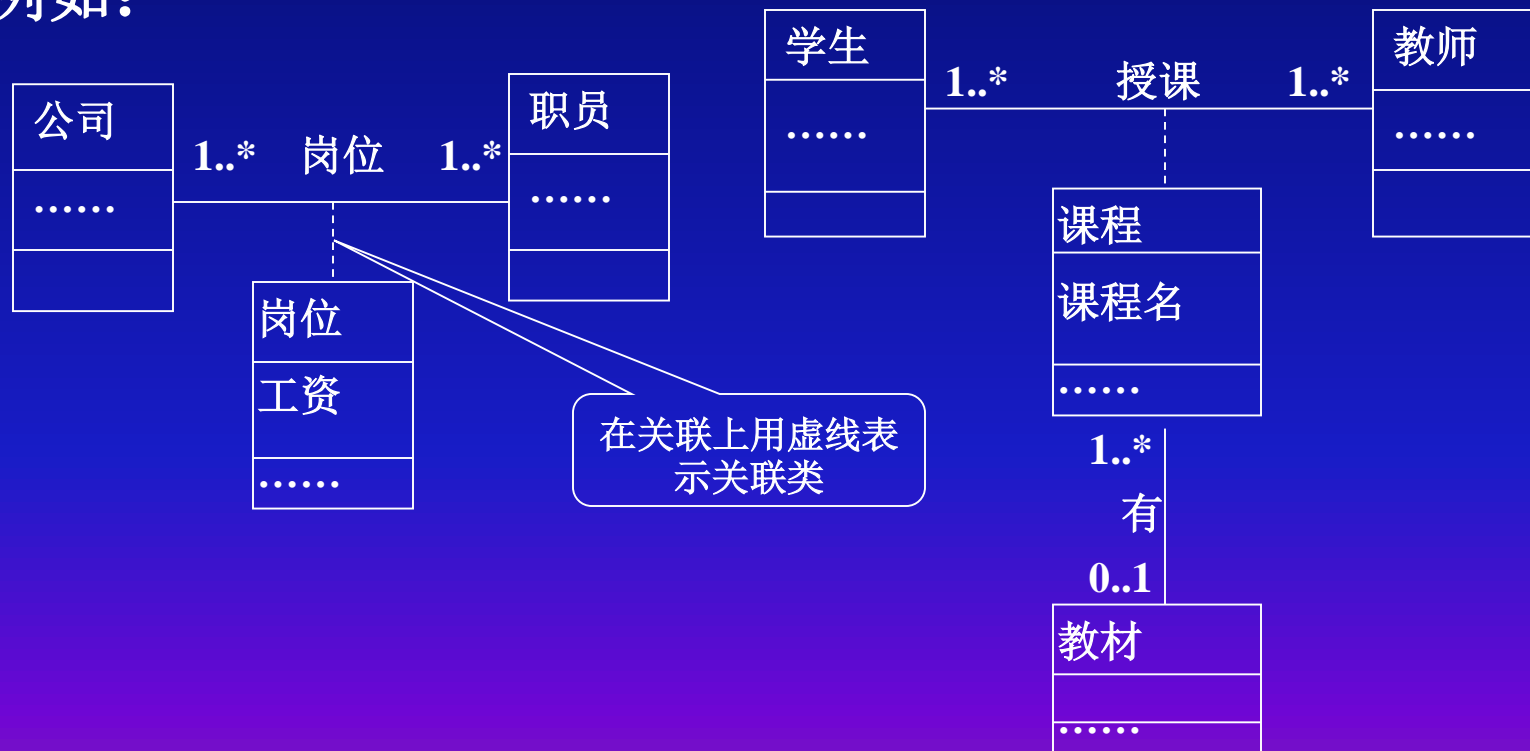
带有角色名的关联：



## 5) 关联类

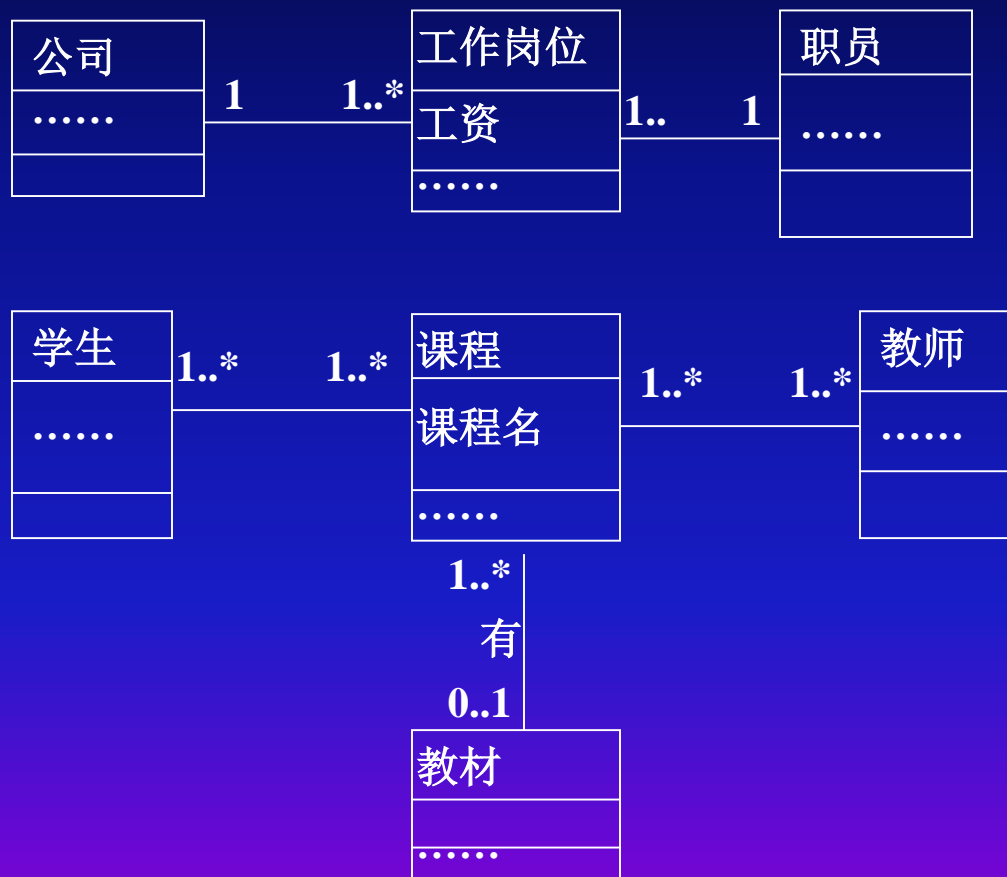
关联类是一种建模元素，表示关联本身也是一个类。关联类兼有关联和类的双重特征，可把它看作具有类性质的关联，也可看作是有关联性质的类。

例如：



关联类的概念在建模中不是必不可少的，可以通过增设类把关联类表示为普通类。

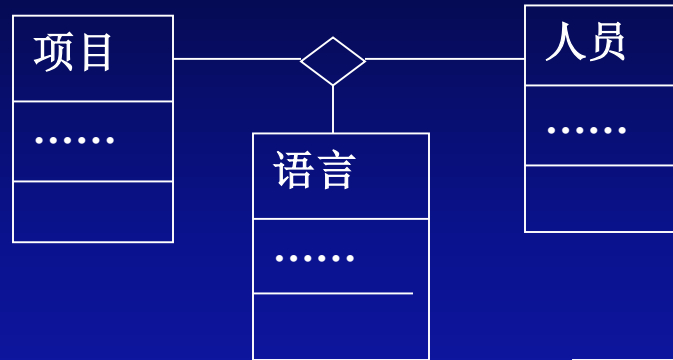
例如：



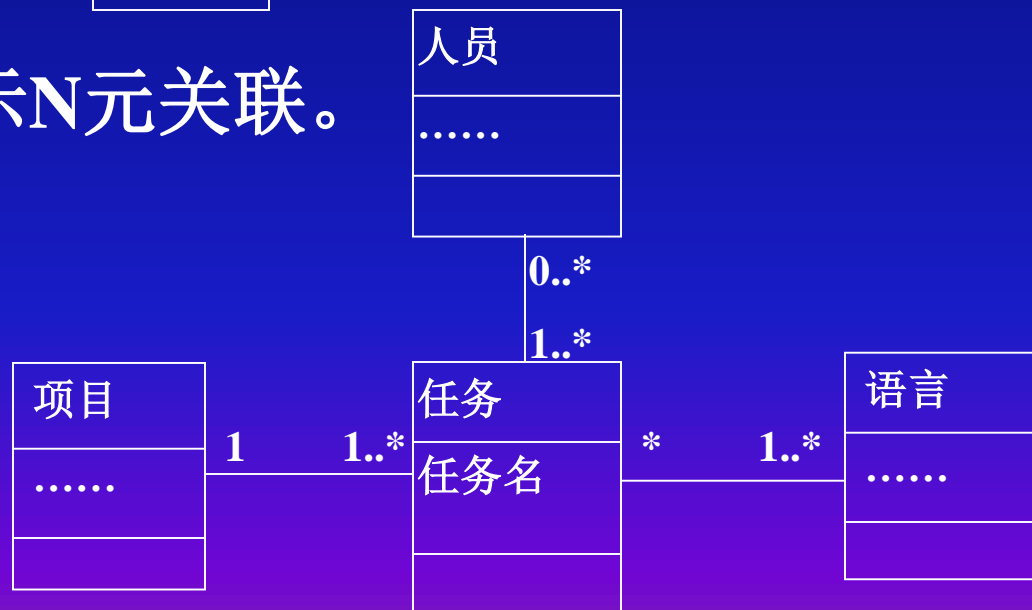
## 6) N元关联

N元关联是三个或多个类之间的关联。

N元关联



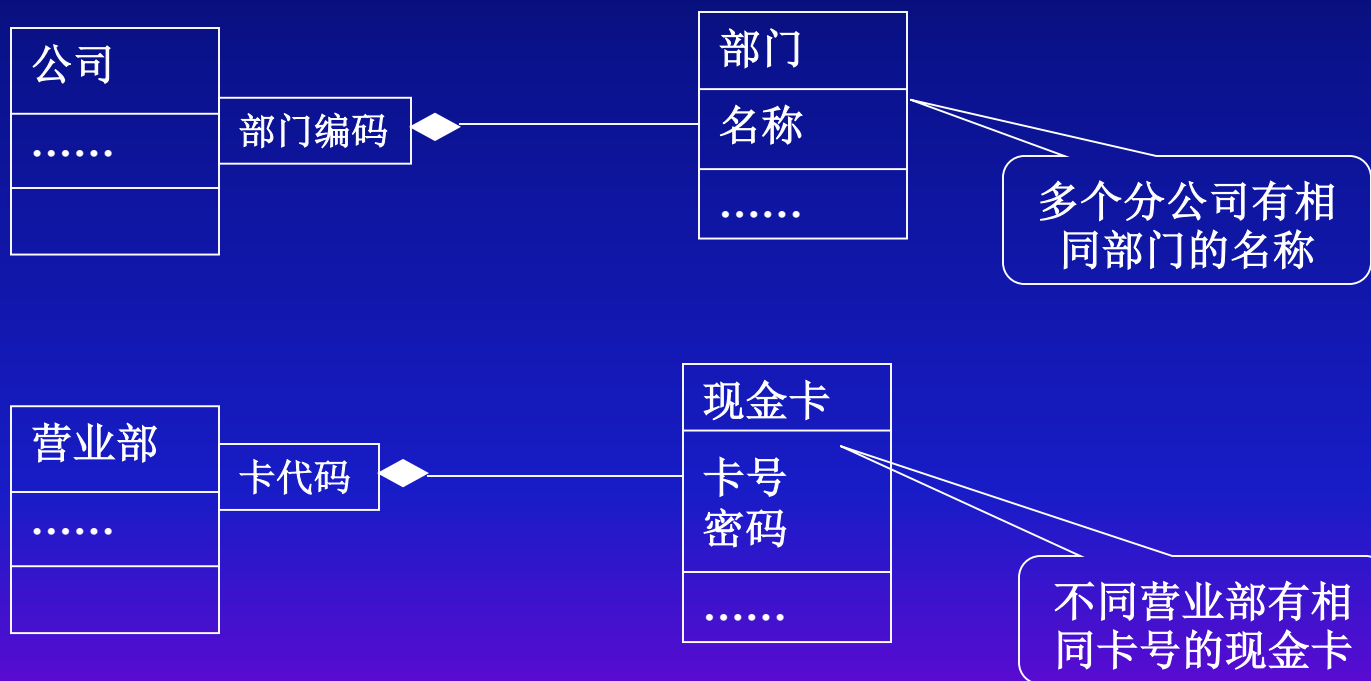
增设类可以表示N元关联。



## 7) 关联限定

关联限定----表示作为查找另一端对象的特征属性，在特定的约束下提供快捷的搜索路径。

例：



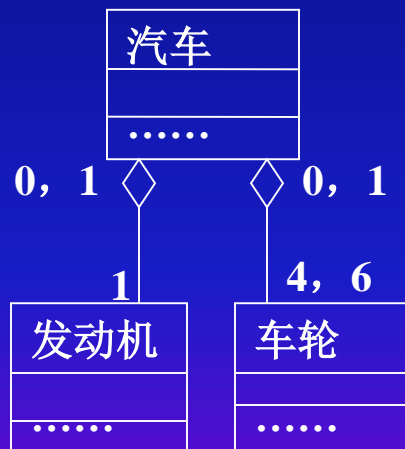
## 8) 聚合关联(Aggregation)

聚合关链表示一种特殊的关联。

**聚合定义** 聚合表示整体类和部分类之间的“整体—部分”关系。

聚合----部分对象和整体对象可以独立生存

例:



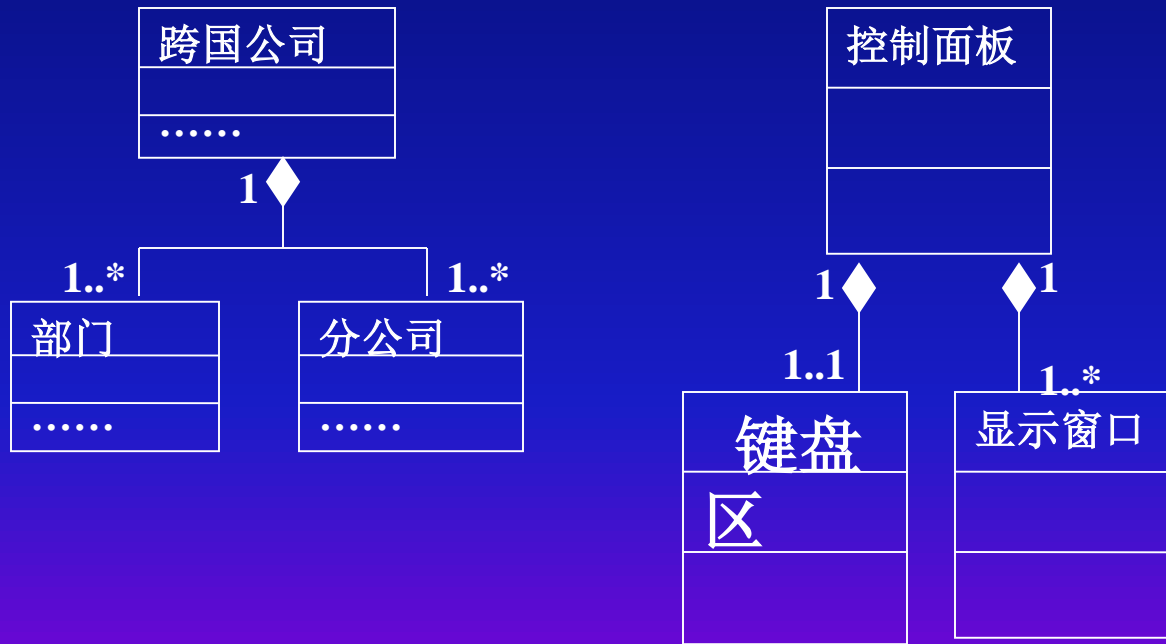


## 组合关联(Composition)

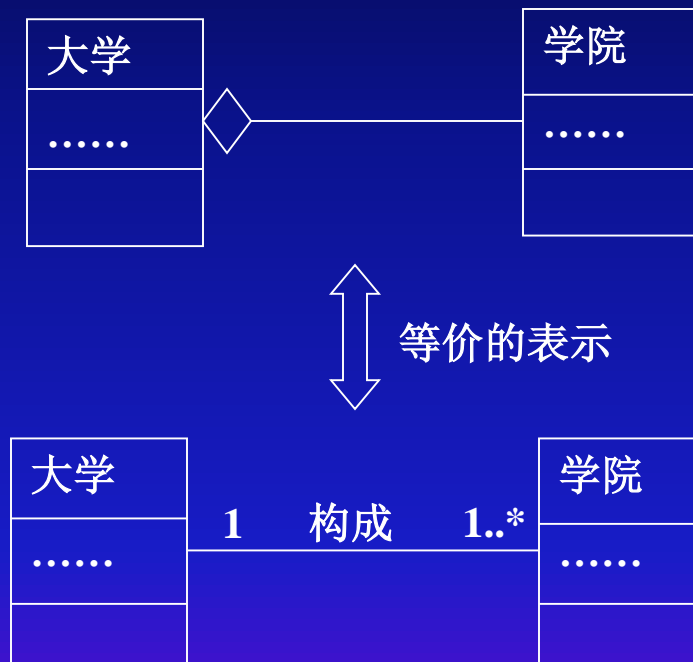
**组合定义** 组合是聚合的一种形式，其中部分和整体之间具有很强的“属于”关系，且它们的生存期是一致的。

组合----部分对象和整体对象生存期一致

例：



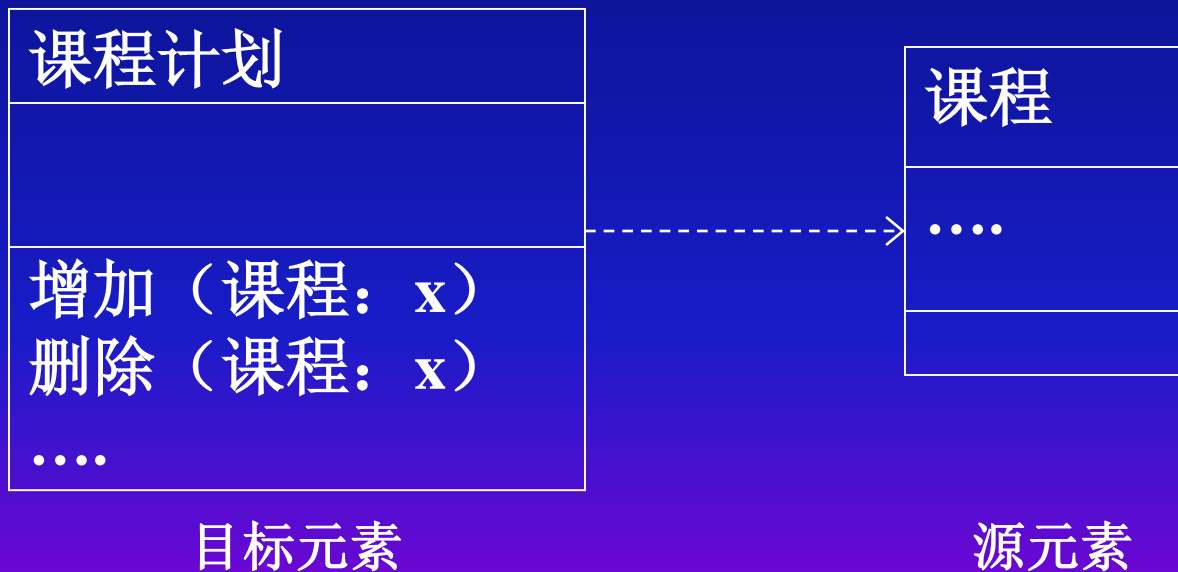
聚合也是一种关联，所以，可以表示成一般的关联的形式。



## 9) 类的依赖关系表示

**依赖关系定义** 模型元素(或模型元素集合)之间的一种语义关系, 目标元素的改变需要根据源元素的改变而变化。

例: 课程计划依赖课程的变化而变化



# 接口类和信号

接口类——把类的公共可见性操作组织在一起，提供的服务集合。

- 接口类作为类之间交互操作的契约。
- 接口类两端的类可独立变更，但操作契约不变。
- 多个类可使用一个接口类呈现整体服务。

接口类的表示：

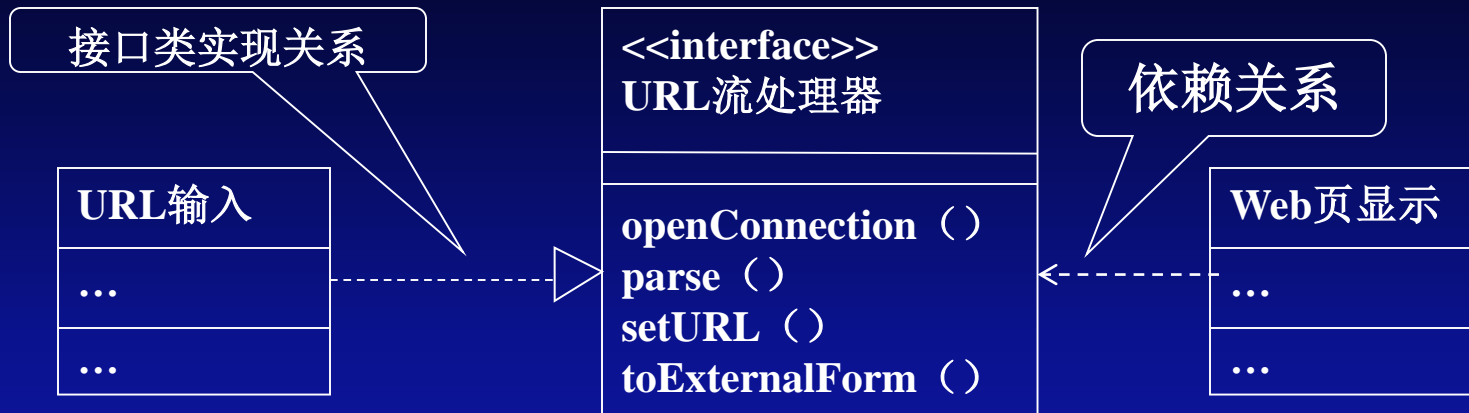


或



接口名

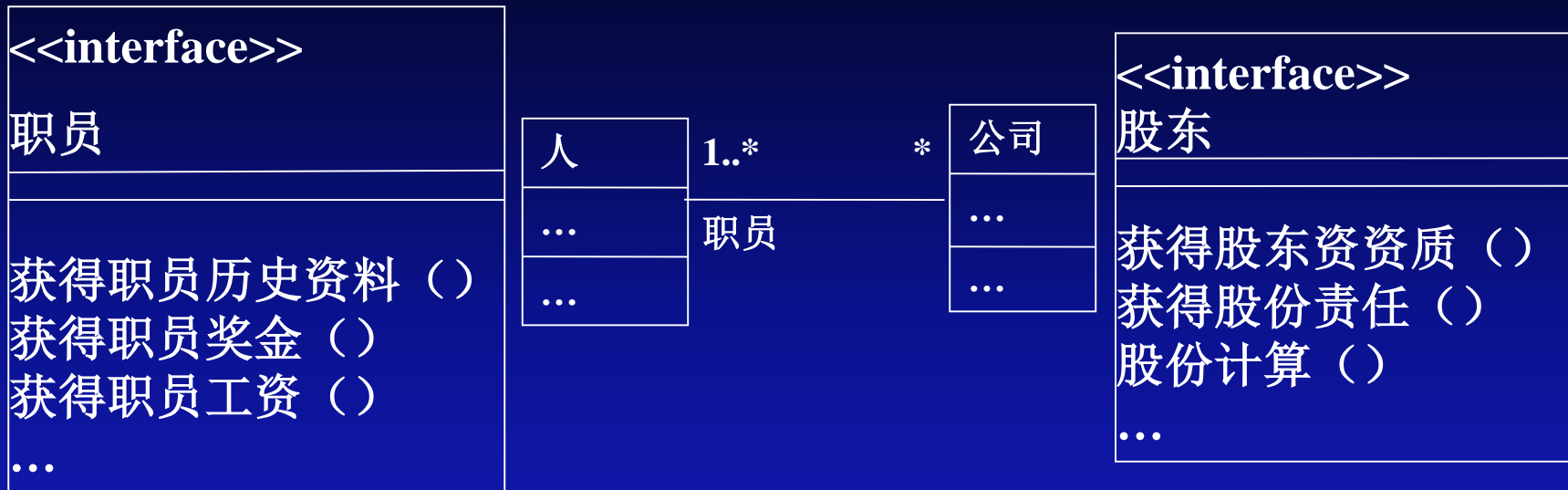
## 接口类的表示例：



## 简化形式的类接口表示：

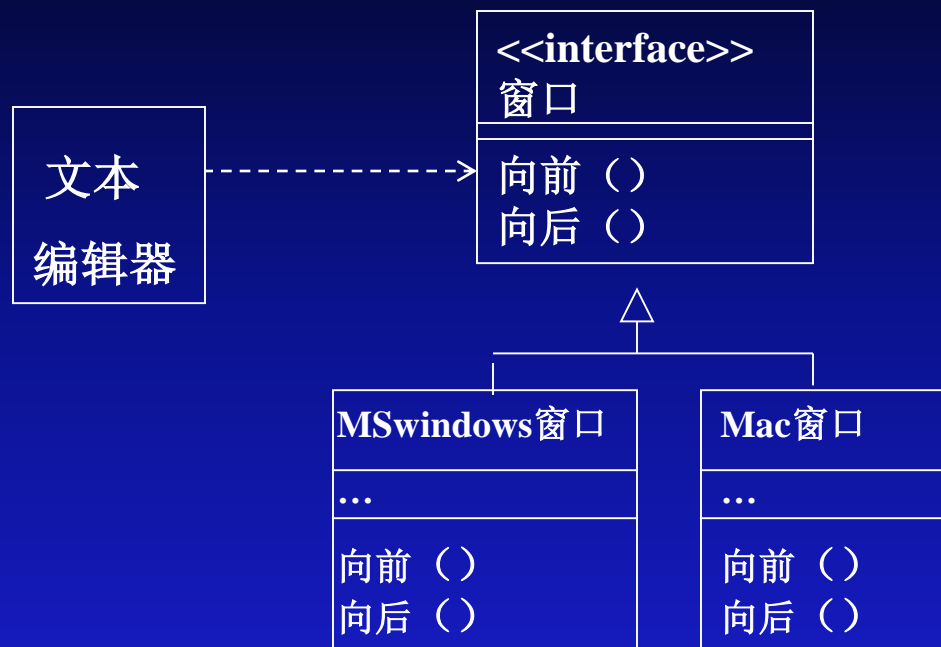


## 表示关联上的接口：



在人和公司之间定义了职员角色的接口类，其中提供了角色的相关操作。类似地，可以提供股东角色的接口。

## 公共接口的抽象类:



可以把继承中的抽象类作为接口类，用特殊类对接口中的抽象操作进行具体的操作执行。

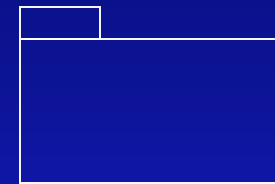
# 包图 (Packages)

## ■包图概念与表示

**包定义：包是对模型成分分组的机制**

要点：

- 把模型成分组织成为包；
- 模型成分包括类或用例；
- 包有唯一的命名，可以被独立引用。



包的表示：

目的：

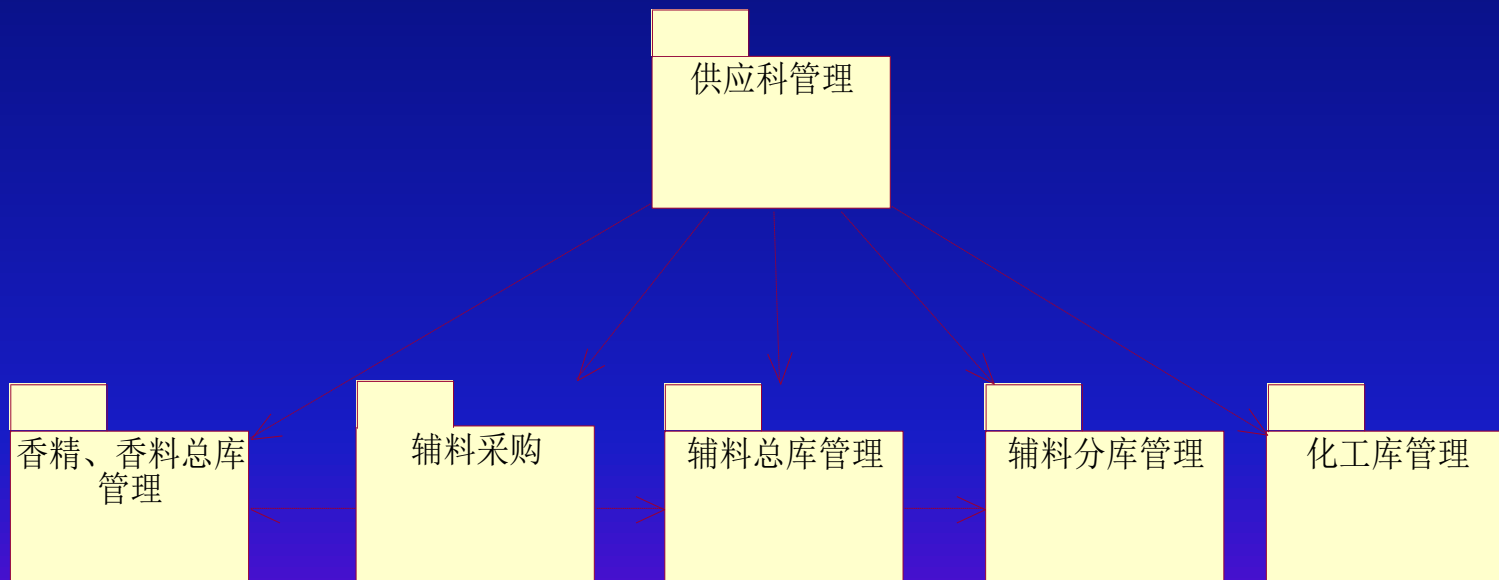
- 用包组织系统成分，使系统形成整体组织结构。
- 包作为独立系统成分，可被整体利用。



## 包的层次性:

多个包可以形成严格的树形层次结构，用于描述系统的组织结构。

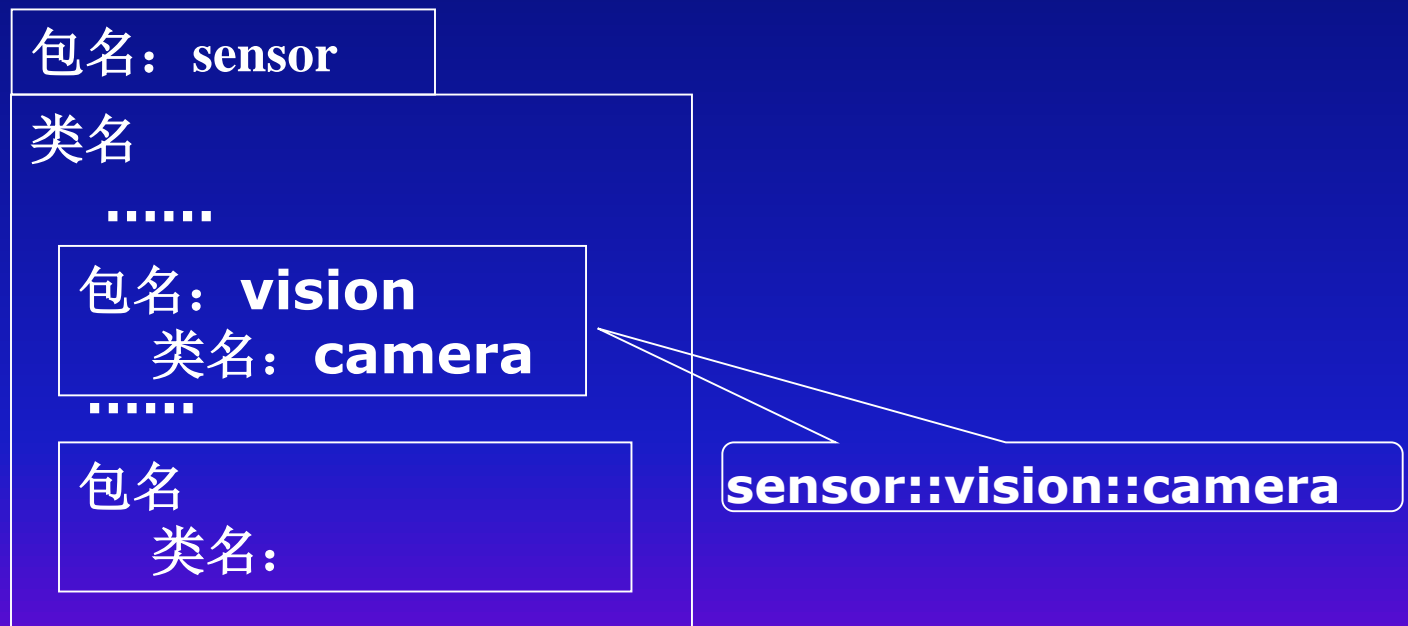
### 树状包结构例:



## 包的嵌套性:

一个包可以嵌套在另一个包内，内层的包成分，同时属于内层和外层两个包。

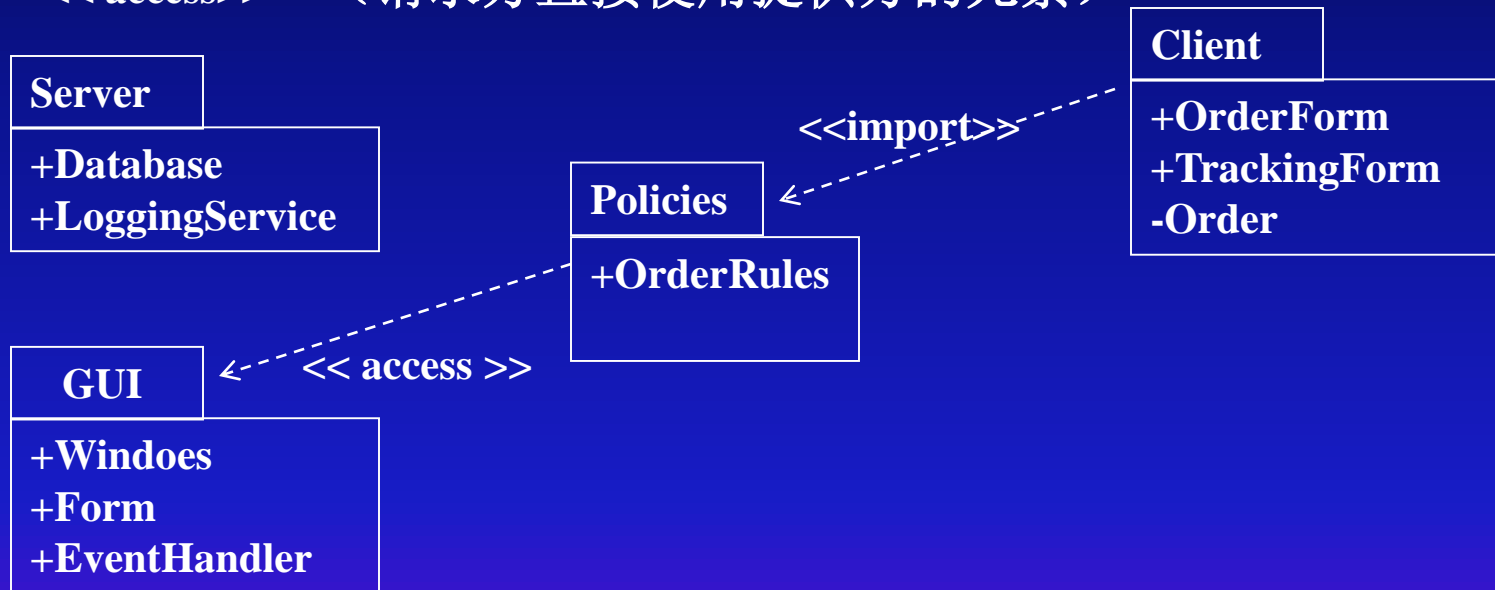
## 包的嵌套结构例:



## 包之间依赖关系:

引入依赖---包中可见的元素可以被另一个包引用  
<<import>> (即提供方的元素直接附加到请求方)

访问依赖---包中可见的元素可以被另一个包使用  
<<access>> (请求方直接使用提供方的元素)



包中元素的可见性分为:

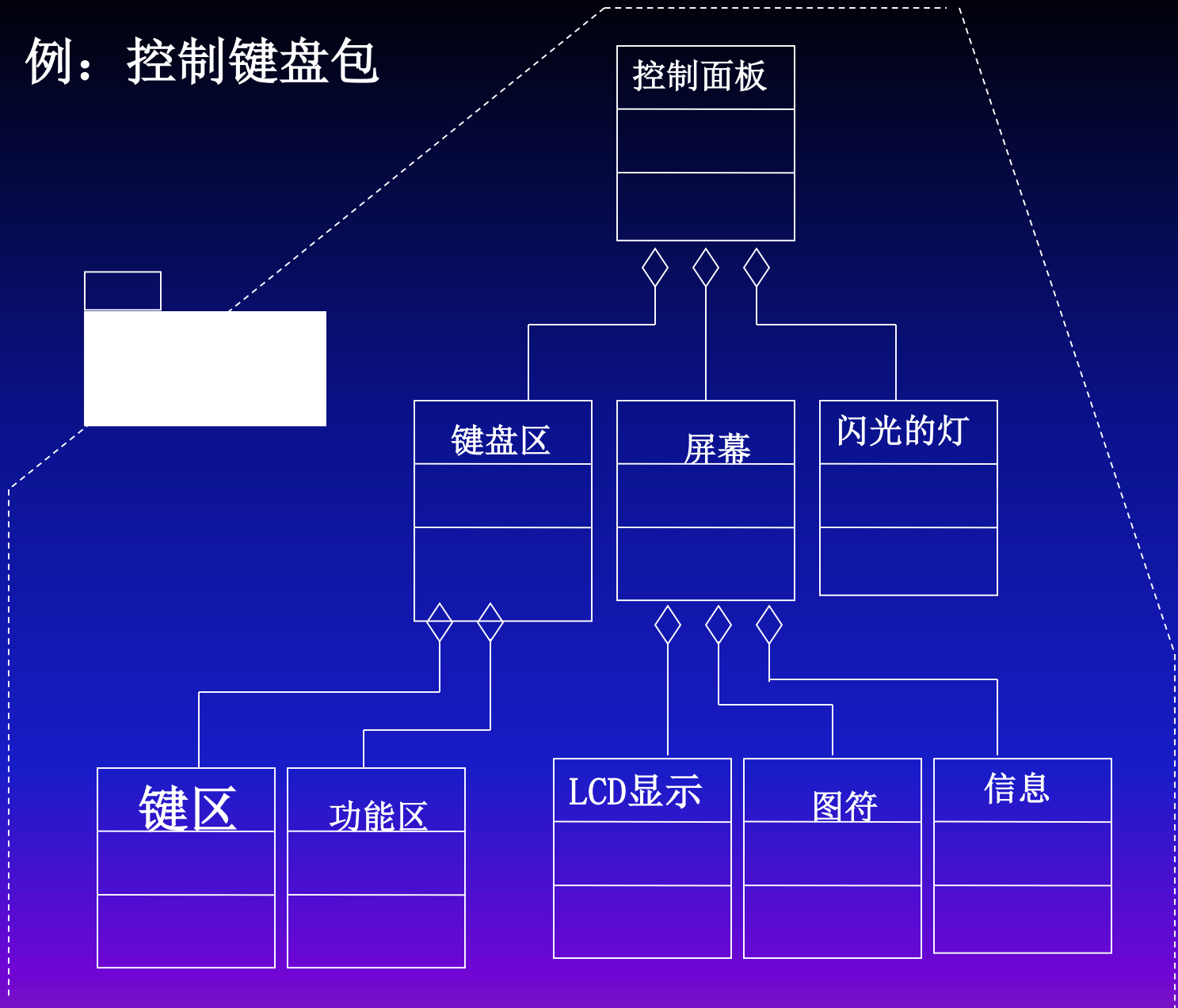
公共的 (+)、私有的 (-) 和受保护的 (#)

## ■包的组织和划分

划分包的策略：

- 1) 把在语义上接近并需要一起变化的成分组织成包；
- 2) 可以组织合并包，形成包的嵌套结构，每个包的内层成分最多5-7个；
- 3) 组织包形成树层次结构；
- 4) 标识包中模型成分的可见性；
- 5) 标识包之间的依赖关系。

## 例：控制键盘包



# 动态模型

## ■交互图 Interaction Diagram

描述对象间以及对象与参与者间动态交互过程的次序，通常对应一个用例。

包括：{ 顺序图---强调消息交互的时序；  
协作图---强调对象收发消息的组织结构

## ■状态图 Statechart Diagram

描述一个对象在其生存期中的状态序列，表现引起状态变换的事件和状态变化的动作。

## ■活动图 Activity Diagram

描述系统的工作流程和并发行为。

# 顺序图概念与表示

## 顺序图Sequence Diagram 作用:

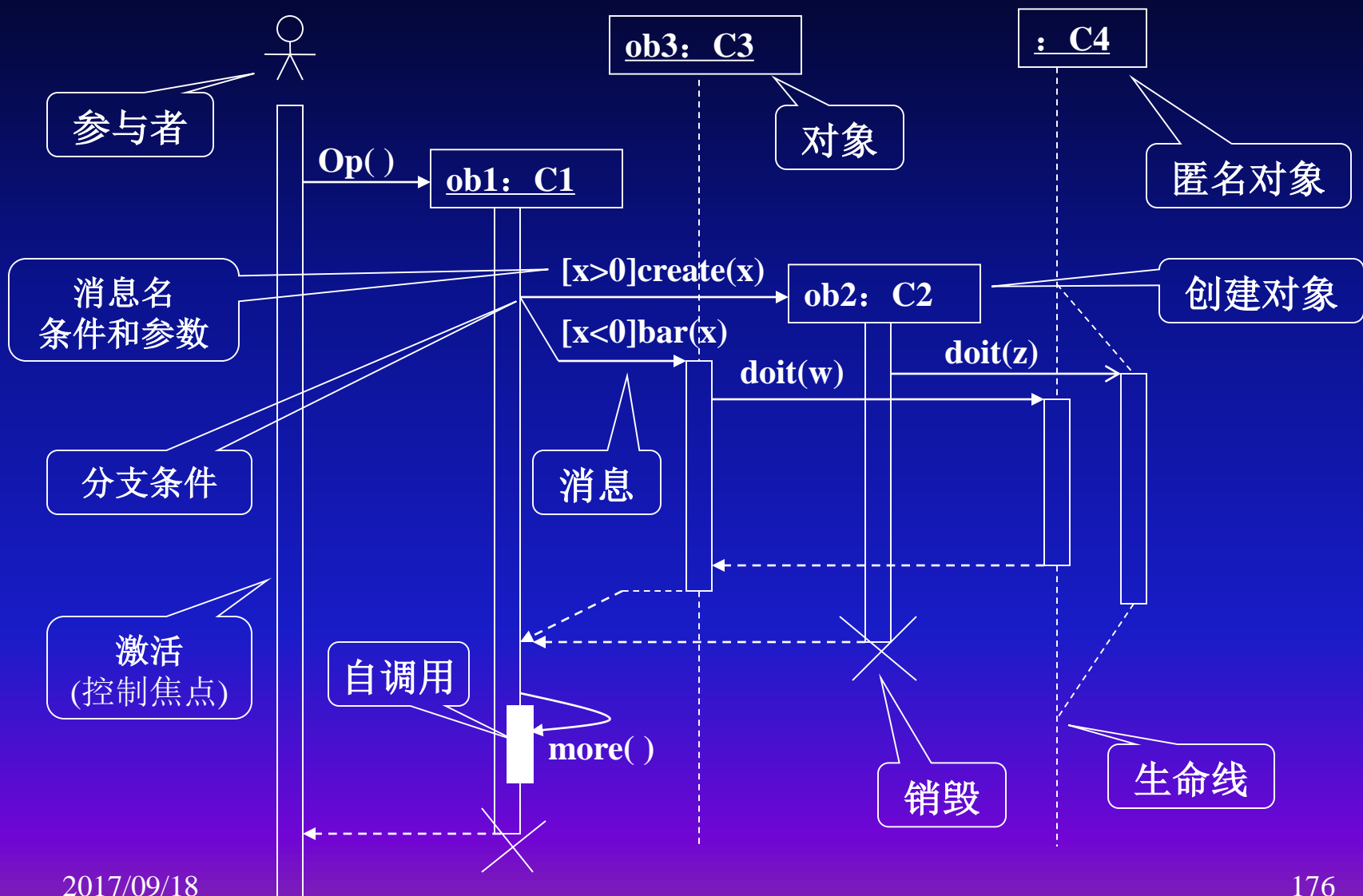
- 在给定的语境中，通过对象之间的消息通信，展现对象的行为；用以发现对象的操作。
- 顺序图可协助发现主动对象；

表示：对象(与参与者)之间的交互；

元素：一组协作的对象(与参与者)和之间发送的消息；


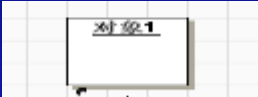

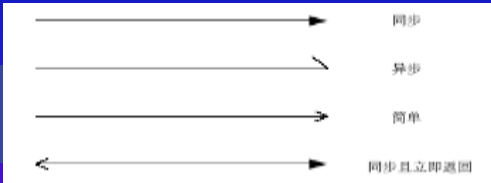
强调：消息之间的顺序；

# 顺序图示例：





# 顺序图示说明：

事物名称	解释	图
参与者	与系统、子系统或类发生交互作用的外部用户(参见用例图定义)。	
对象	顺序图的横轴上是与序列有关的对象。对象的表示方法是：矩形框中写有对象或类名，且名字下面有下划线。	
生命线	坐标轴纵向的虚线表示对象在序列中的执行情况(即发送和接收的消息，对象的活动)这条虚线称为对象的“生命线”。	
消息符号	消息用从一个对象的生命线到另一个对象生命线的箭头表示。箭头以时间顺序在图中从上到下排列。	

## 顺序图说明：

对象---类创建的对象实体名

匿名对象---只表示类名

创建对象---通过消息发送创建的对象

生命线---表示对象存在的时间段，

激活（控制焦点）---表示对象行为的操作历程，即活动的持续时间

销毁---对象被结束生命（杀死）

自调用---对象的操作递归调用自己，或本对象的其他操作

消息--- 包括：

实箭线 “ ” 同步过程，需要等待回应，会有嵌套控制

枝状箭线 “ ” 异步通信，发出消息后不必等待，继续执行

行虚的枝状箭线 “ ” 显式表示回应返回（一般

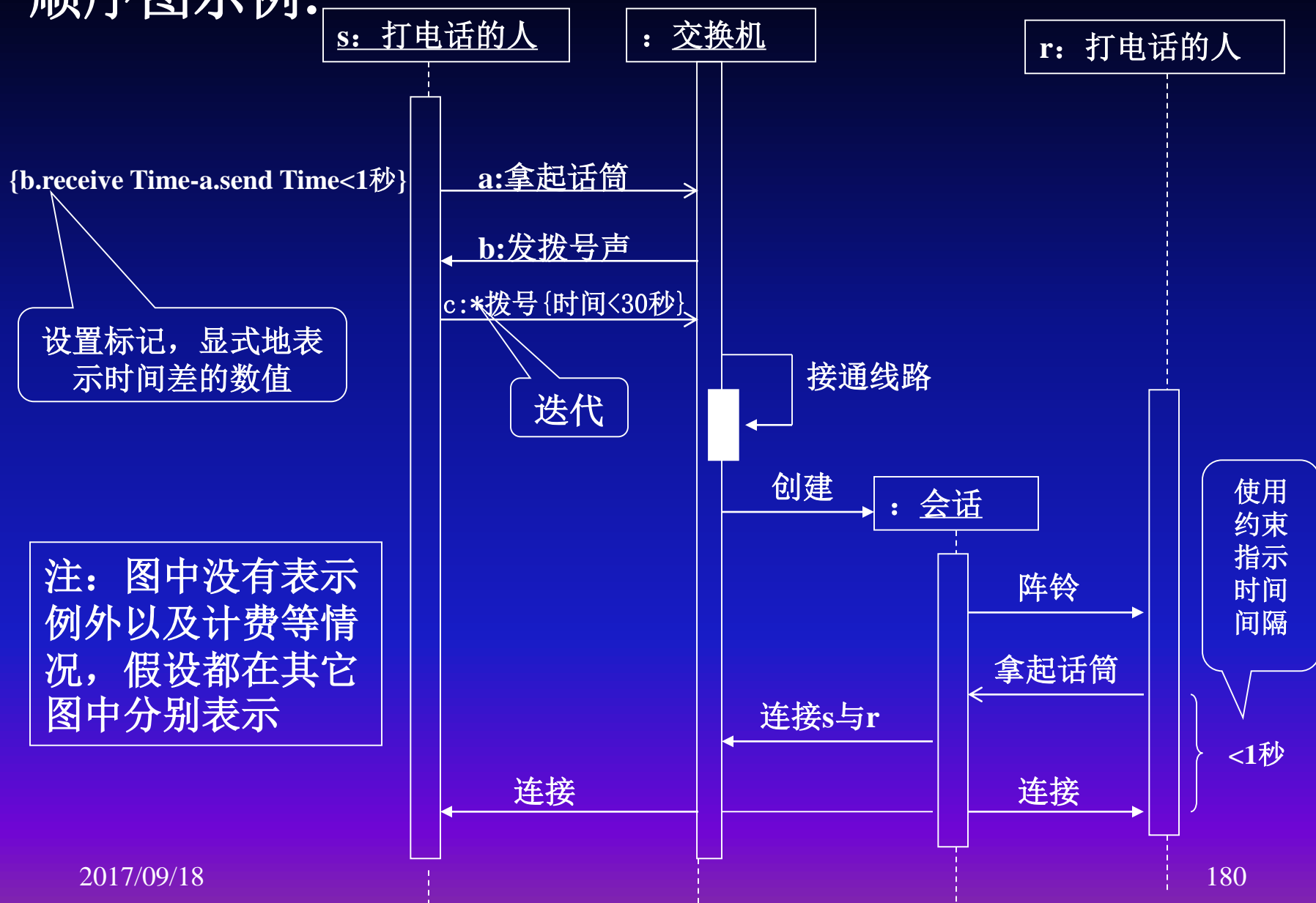
的控制流省略返回消息，表明每个请求都有返回）

# 顺序图的建立

过程和步骤：

- 1) 根据具体用况中的对象或参与者的交互语境，设置交互；
- 2) 在顺序图上部列出所选的一组对象或参与者；
- 3) 为每个对象和参与者设置生命线。
- 4) 用消息箭线显式地标出交互中将被创建和撤消的对象；
- 5) 在对象之间，标出消息进行传递的序列；
- 6) 在对象生命线上，按对象操作的次序，排列各操作的激活区间，若两个对象的操作执行属于同一控制线程，则接收者操作的执行应在发送者发出消息之后开始，并在发送者结束之前结束；
- 7) 描述对象执行的操作功能、时间、范围约束；
- 8) 可显式地表示消息的迭代或分支。

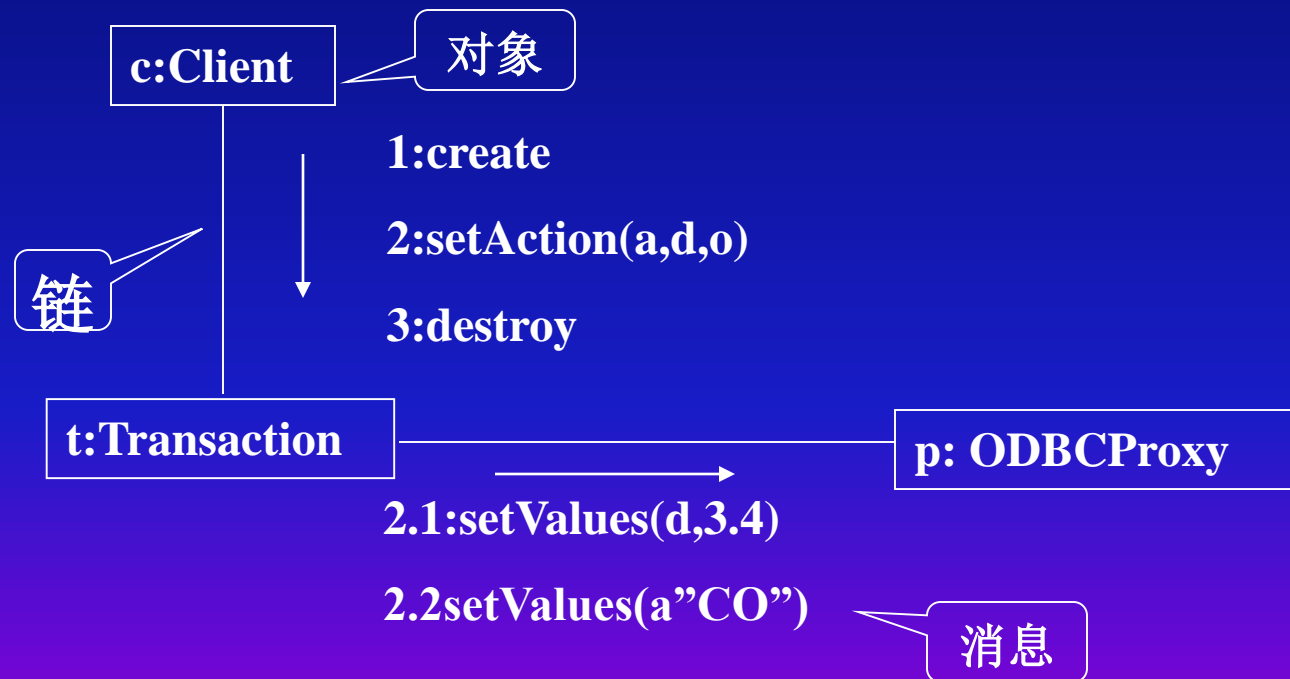
# 顺序图示例:



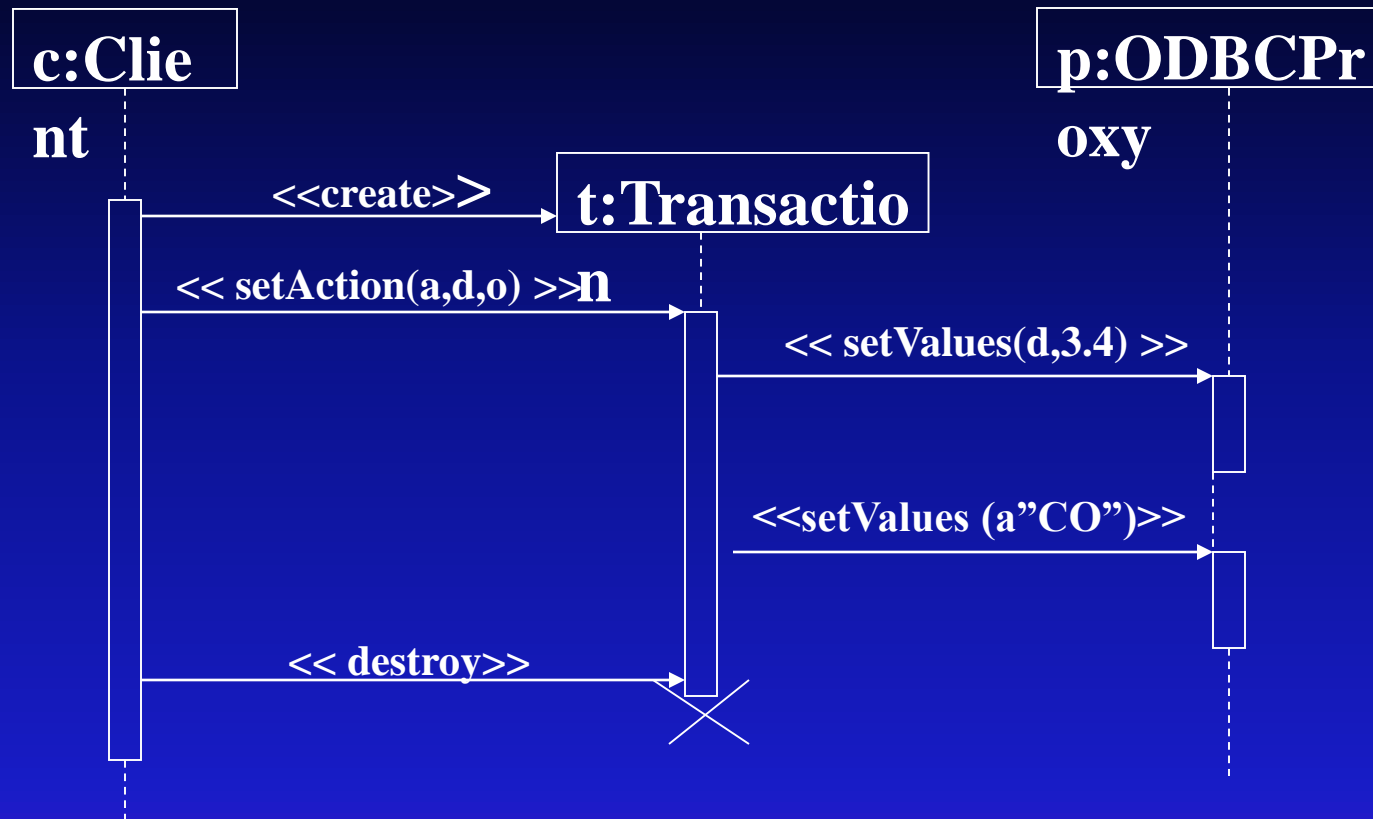
# 协作图概念与表示

**协作图** Collaboration Diagram 表示协同完成某行为对象之间的交互，强调对象的消息结构，而忽略时间顺序。

协作图示例：



与前页协作图在语义上等价的顺序图：



顺序图和协作图表示相同的模型语义，可以把顺序图和协作图从一种形式转换为另一种形式。但它们都可以表示对方不能表示的某些内容。

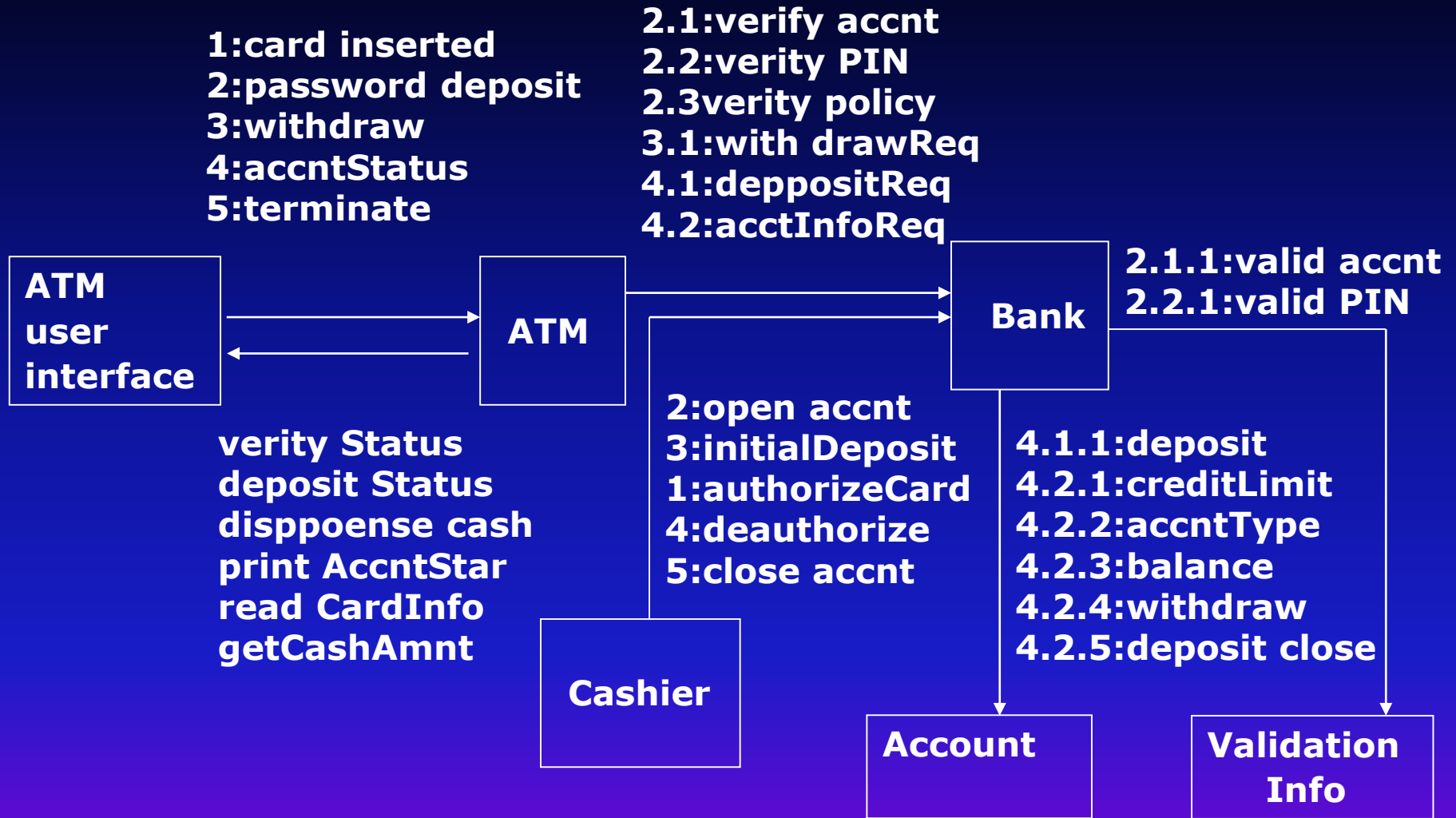
# 协作图的用途

可用于分析对象行为的控制线程，规划系统分布及测试用例

例：用协作图规划储蓄应用的行为：



## 例：用协作图规划储蓄应用的行为：



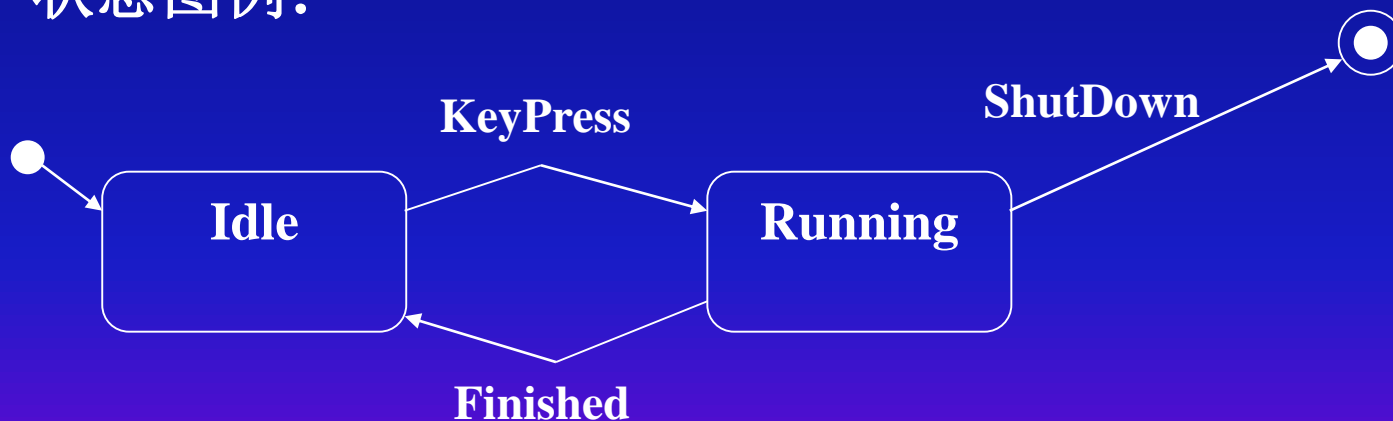


# 状态图概念与表示

## 状态图State Transition:

表示对象事物的行为，描述了一个对象在其生命期内响应事件所经历的状态序列以及对这些事件所做出的反应。

### 状态图例:



## 状态图遵循事物转换的规律：

- 1) 事物在生命期中都经历不同的状态；
- 2) 事物在特定时刻总处于某确定的状态；
- 3) 事物的状态变化总是由事件引起的；
- 4) 事物状态的转化是即时的；
- 5) 事件发生时事物需要采取相应的动作；
- 6) 事物的状态转换总会遵照一定的规律的。

状态图的作用：

分析对象上的所有状态，确定状态值（行为属性）的范围，以及对应的操作，包括：

- 对象具有的稳定状态
- 保持稳定状态的动作行为
- 状态转换的触发事件
- 状态改变时发生的动作
- 对象的创建和消亡（必要时）

要点：

针对有明确生命阶段和复杂行为的对象，才有必要建立状态图，用以分析行为属性和值域范围。

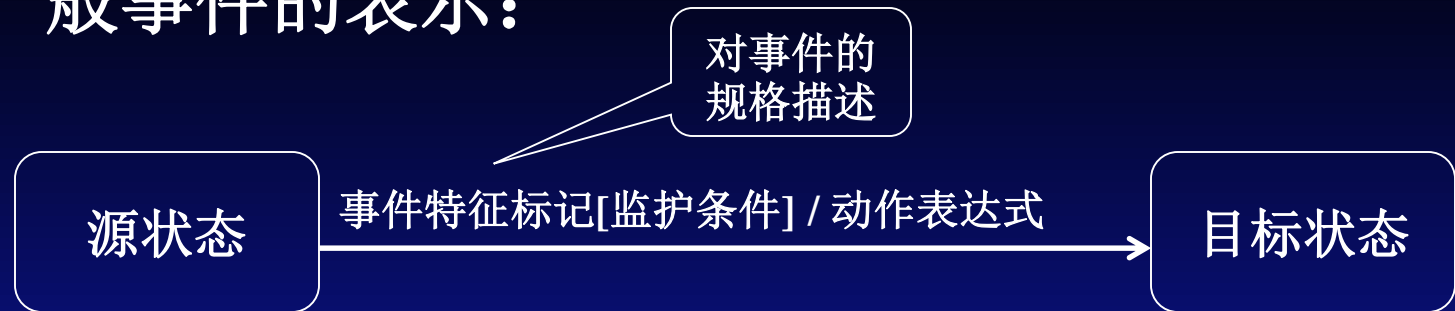
## 状态图元素：

**事件定义：** 事件是对一个可视的事情的描述，这种事情的发生可以引发状态的转换。

### 事件的种类：

- 1) 信号事件---显式地接收，直接响应的事件
- 2) 调用事件---操作调用间接引发
- 3) 时间事件---某时间段之后  
由“after”和时间表达式表示
- 4) 改变事件---满足某条件  
用“when”和布尔表达式表示

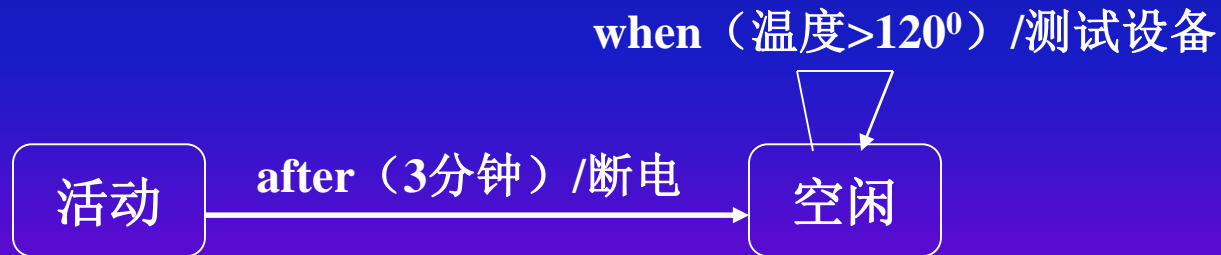
## 一般事件的表示:



当满足一定的条件出现特定事件时，引发对象状态转换，并执行一定的动作。

## 时间事件和改变事件表示:

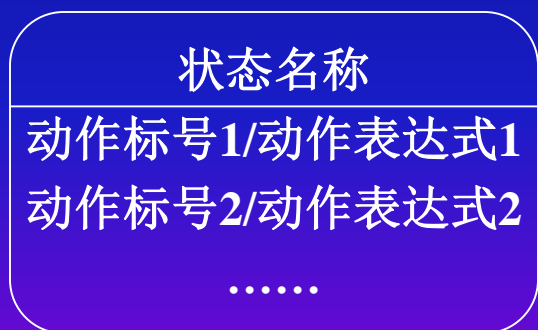
例:



**状态：**状态是对象生命期的一个阶段，在该阶段中，该对象满足一些特定的条件、从事特定的动作或等待某个（些）事件。

**动作：**动作是在状态内或在状态转化时所做的操作，是原子的和即时的。

状态表示：



伪状态表示：



初始状态



终止状态

## 与状态有关的几个概念：

- 1) 内部动作---不发生状态转换的动作
- 2) 延迟事件---保留在状态队列中，适当时及执行的事件  
表示为：事件/defer
- 3) 伪状态 ---初始和终止状态

## 特定动作标号含义：

- 1) entry/动作表达式---进入状态时执行该动作
- 2) exit/动作表达式---退出状态时执行该动作
- 3) do/活动---在状态整个阶段执行的动作集合

## 状态示例:

“打印服务器” 输入口令的状态

### EnterPassword

```
entry/password.reset()  
exit/password.test()  
digit/handle character()  
clear/password.reset()  
help/display help  
print/defer  
do/suppress echo
```

## 事件例:

**right-mouse-down(location)**      事件特征标记

**[location in window]/**      监护条件

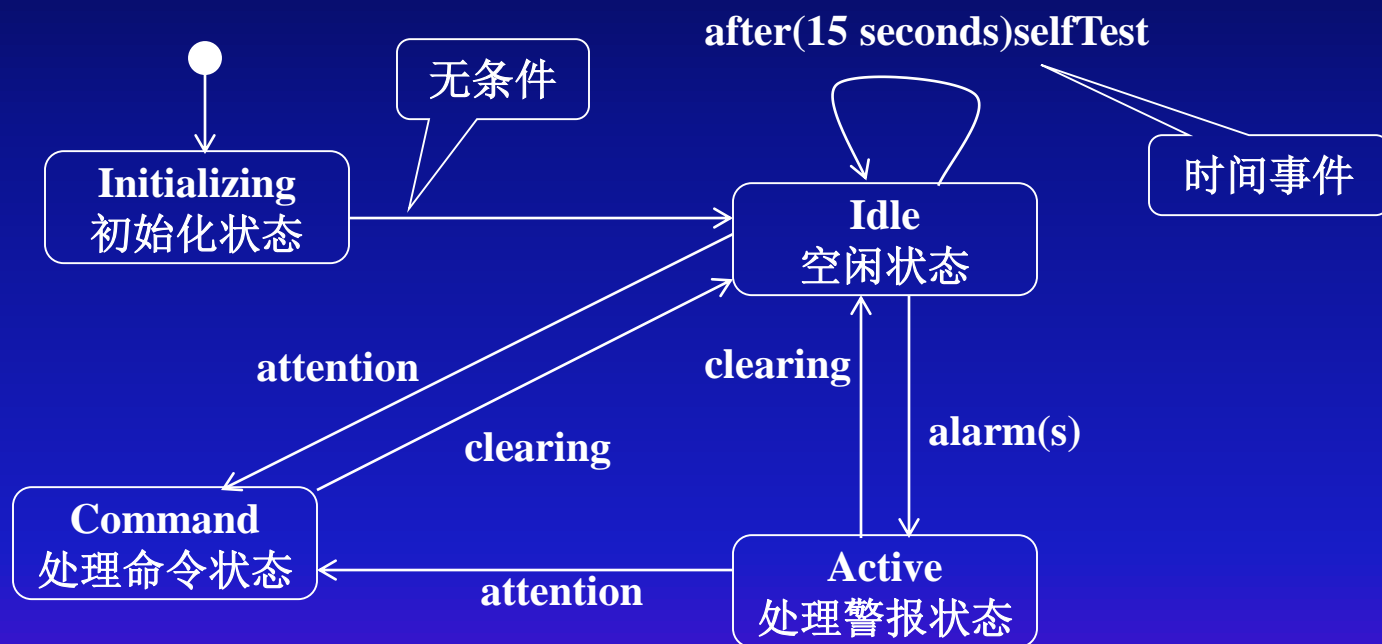
**object:=pick-object(location);** } 动作表达式

**Object.highlight();**



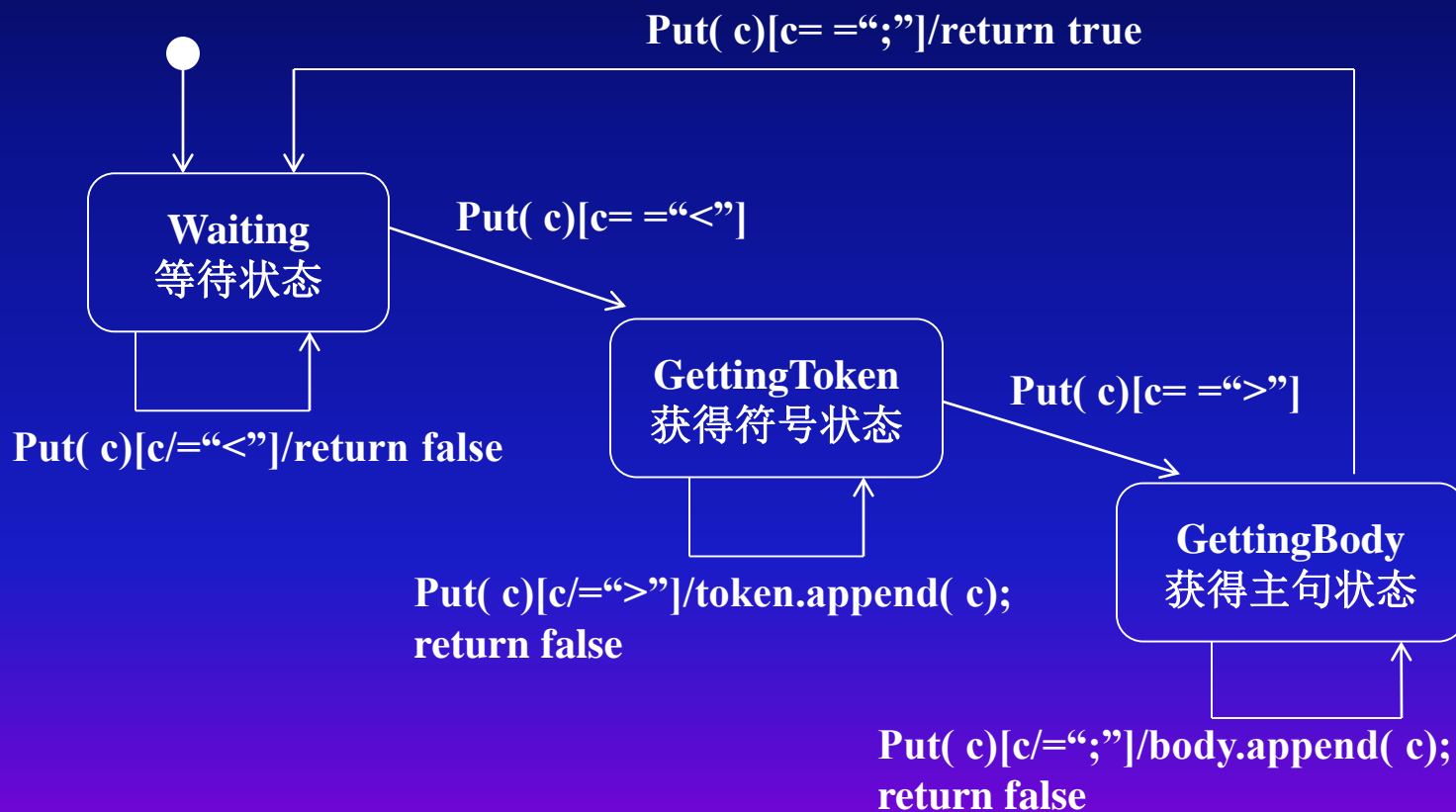
# 状态图例：

## 监视传感器类的状态图：



# 状态图例:

字符流分析器类状态图: { 字符流形式为:  $x...x <xx...x> xx..x$  ;  
要求分析出<>中的符号,以及;号前的主句

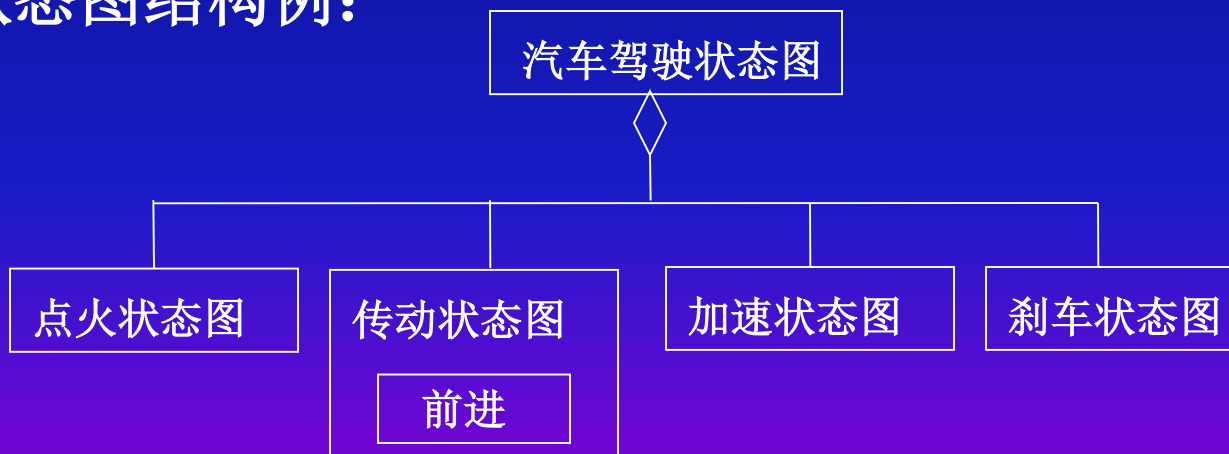


## 状态图的结构表示:

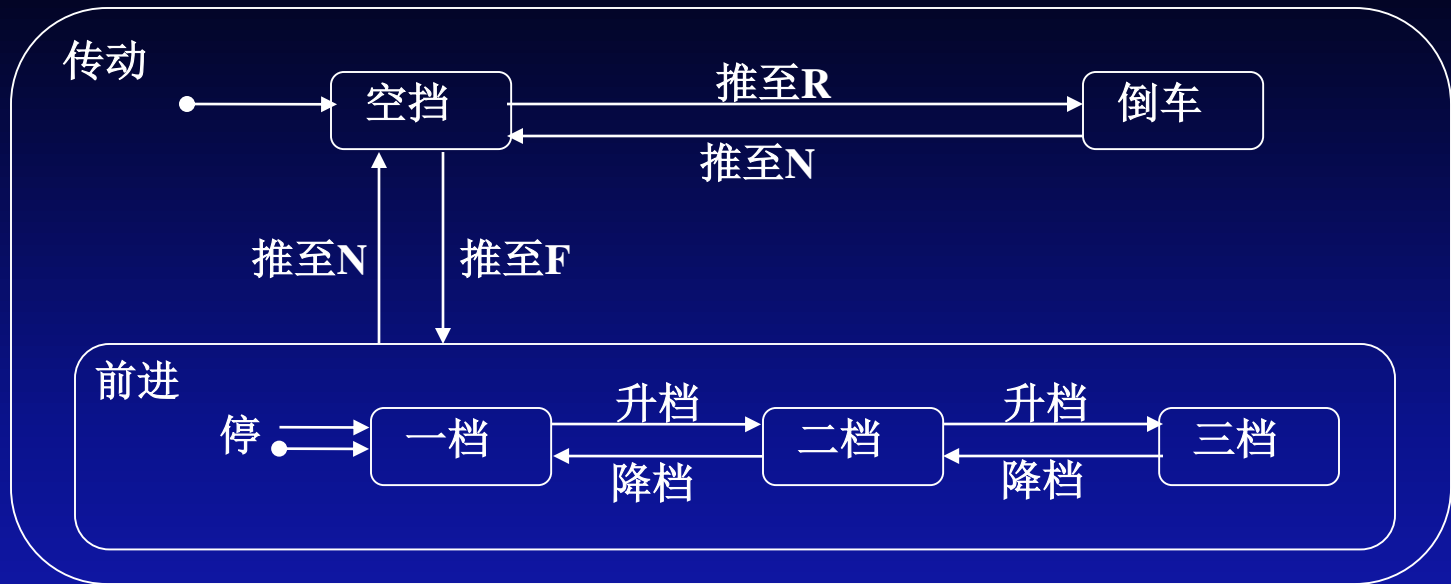
状态图可以用组合状态表示其组织结构, 表明由两个或多个子状态构成的状态。

子状态包括: { 顺序子状态  
并发子状态  
嵌套子状态

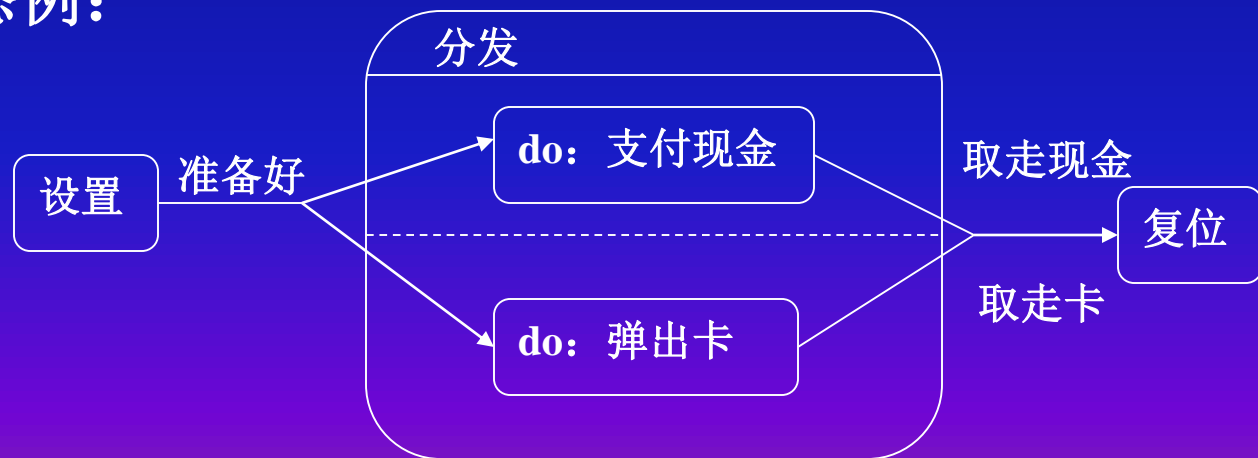
### 状态图结构例:



## 顺序子状态例：



## 并发子状态例：



# 状态图的建立

## 建立策略：

- 1) 考虑某对象在特定语境中的交互行为；
- 2) 建立初始状态和终止状态；
- 3) 从属性值的范围和条件考虑对象所在的稳定状态；
- 4) 从对象的生命期开始，确定状态转换；
- 5) 决定对象可能响应的事件；
- 6) 用事件连接状态，给出事件名称、条件和动作；
- 7) 描绘各状态进入或退出的动作及保持状态的动作；
- 8) 从对象高层状态描绘可能的子状态（必要的时候）。

# 活动图概念与表示

## 活动图Activity Diagram:

描述系统的工作流程和并发行为，与状态图相比它强调流程的控制而不是状态在事件下的变化。

### 活动图的作用：

- 描述业务过程，特别能较好地表示并发流程，帮助理解涉及多个用例的工作流程。
  - 描述具体操作算法，与程序框图有相同的作用
- 活动图的概念接近结构化方法的流程图思想。

## 活动图元素：

包括：

- 1) 活动---流程中的任务执行单元
- 2) 泳道---活动的区域划分
- 3) 分支---活动转向的分支
- 4) 分叉--- 并发控制流的分支
- 5) 汇合---分叉后的合并
- 6) 对象流---活动图中的控制流

## ■活动图中的“活动” Activity

包括：动作状态 Action State

Calculate Total Cost

活动状态 Activity State

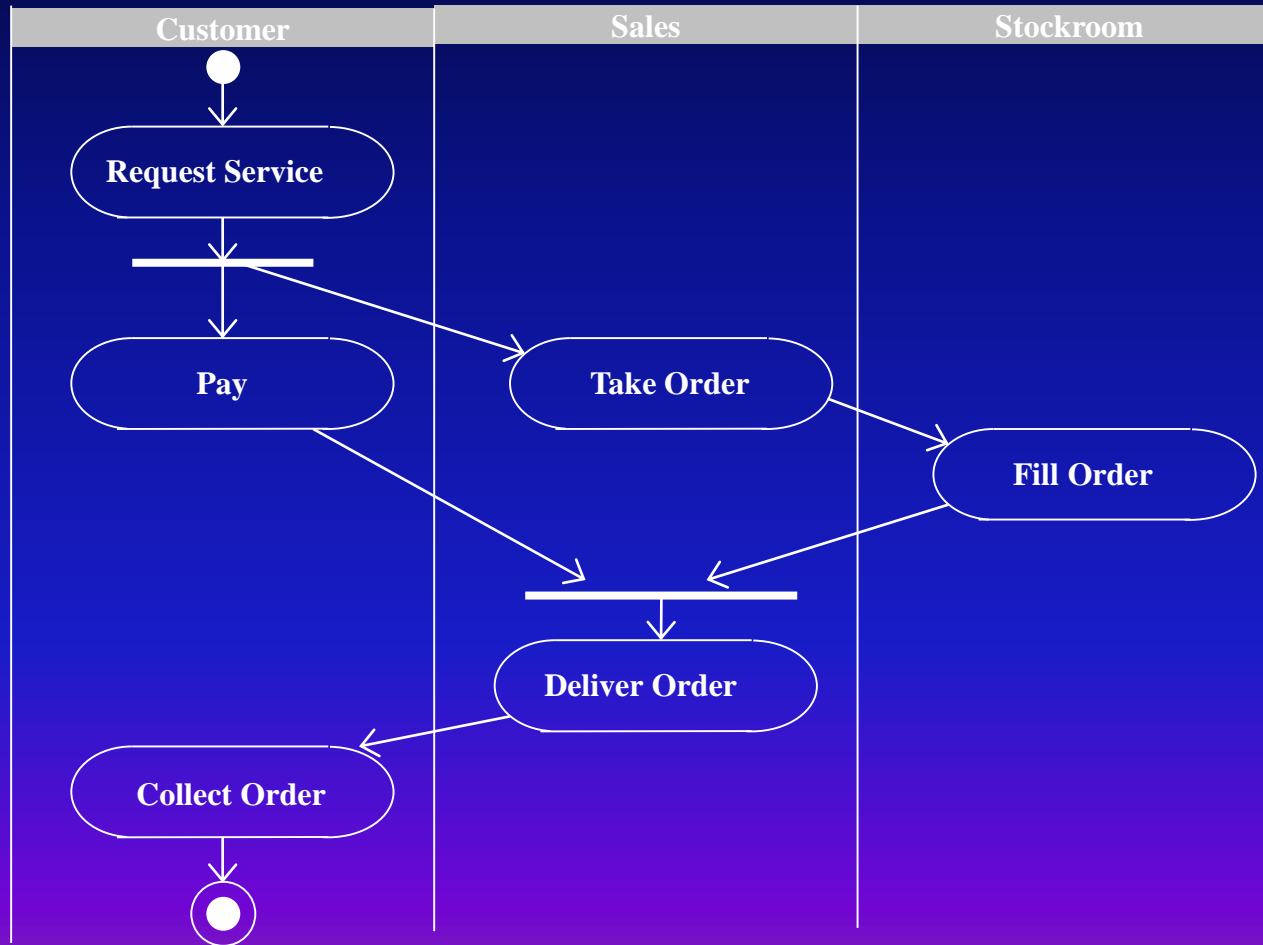
- 动作状态，没有内部转移和内部活动，是原子的最小执行单元，作用是表示执行进入动作后转向的状态，执行时间可以被忽略。
- 活动状态，可分解的、非原子的，并有一定持续时间的执行单元。



## ■活动图中的“泳道” swimlane

根据每项活动的职责，划分所有活动的责任区域

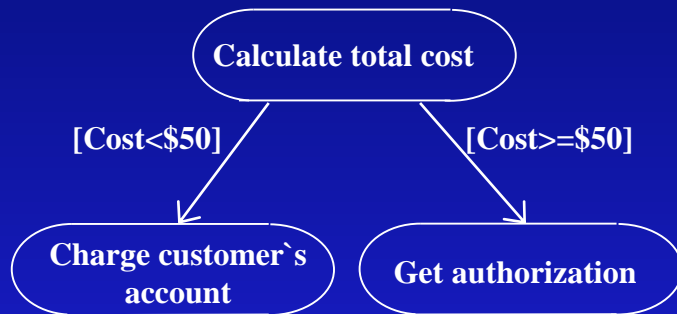
例：



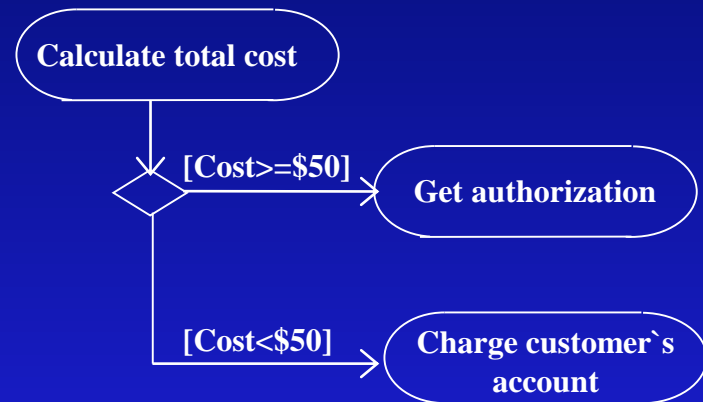
## ■活动图中的“分支” branch

在活动图中同一出触发事件，可根据警戒条件转向不同的活动，有两种表示方法：

例：



表示法1

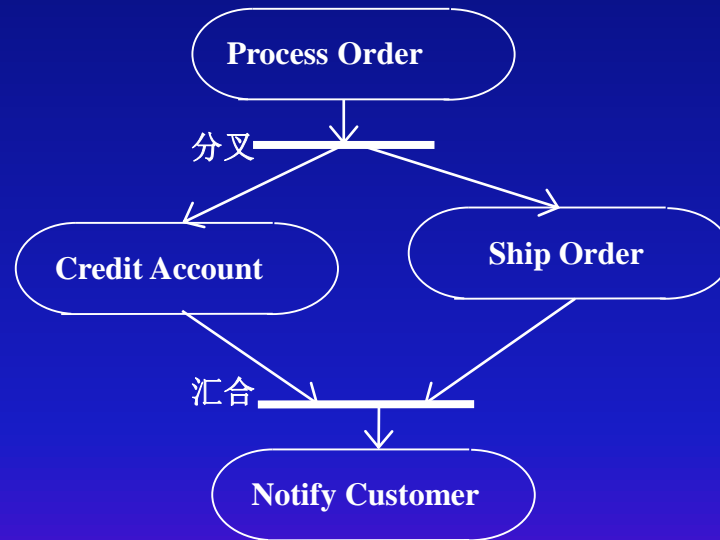


表示法2

## ■活动图中的“分叉和汇合” fork and join

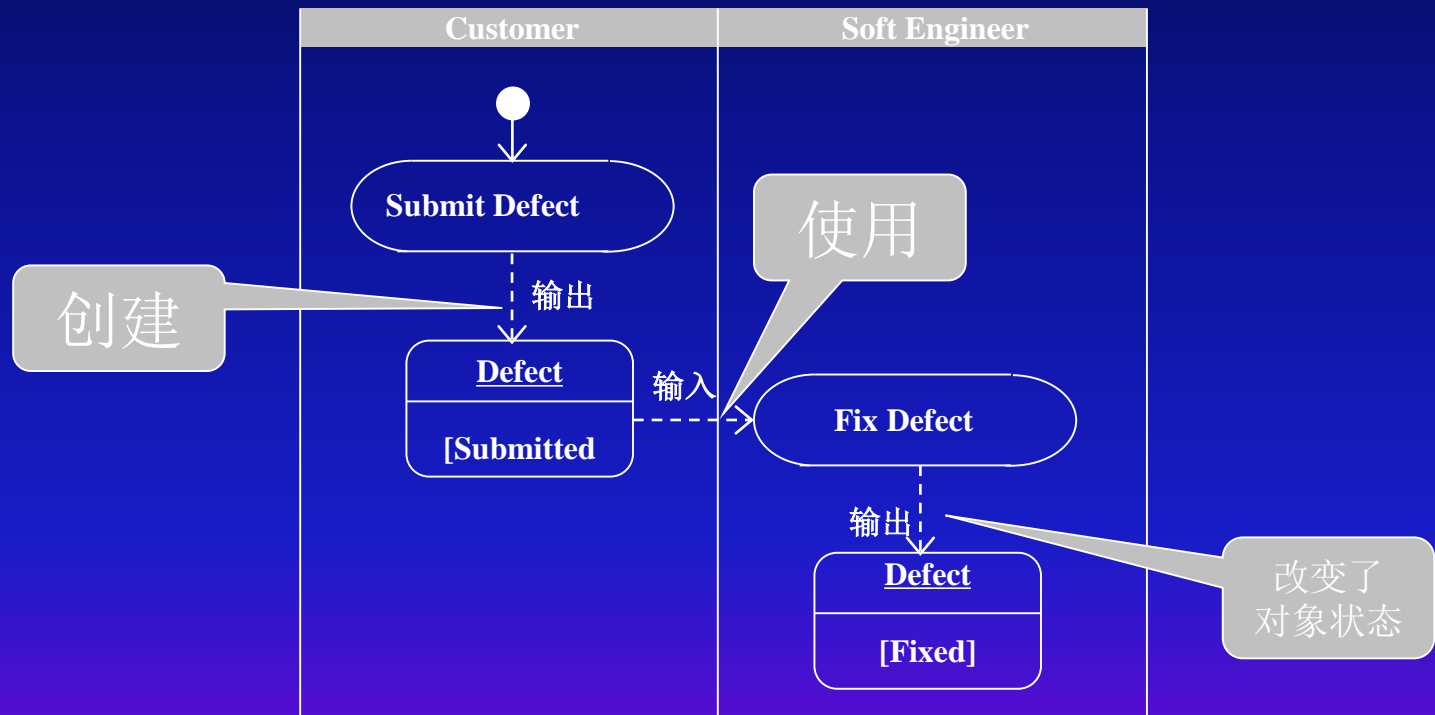
在活动图中同一控制流被两个或多个控制流替代且并发，其转换点为分叉；汇合则与此过程相反。

例：



## ■活动图中的“对象流”

对象作为活动的输入或输出，对象流可表示对象与活动之间的行为，对象流属于控制流。



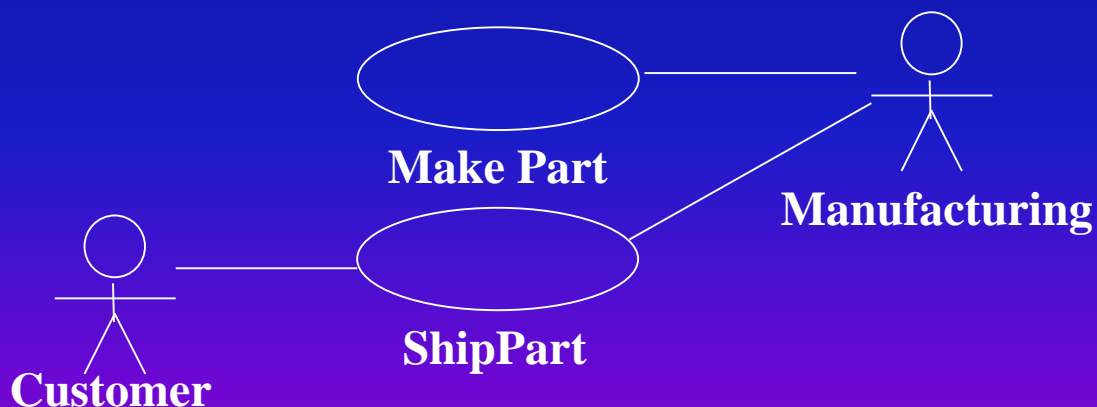
# 活动图应用

## ■用活动图对 workflow 建模

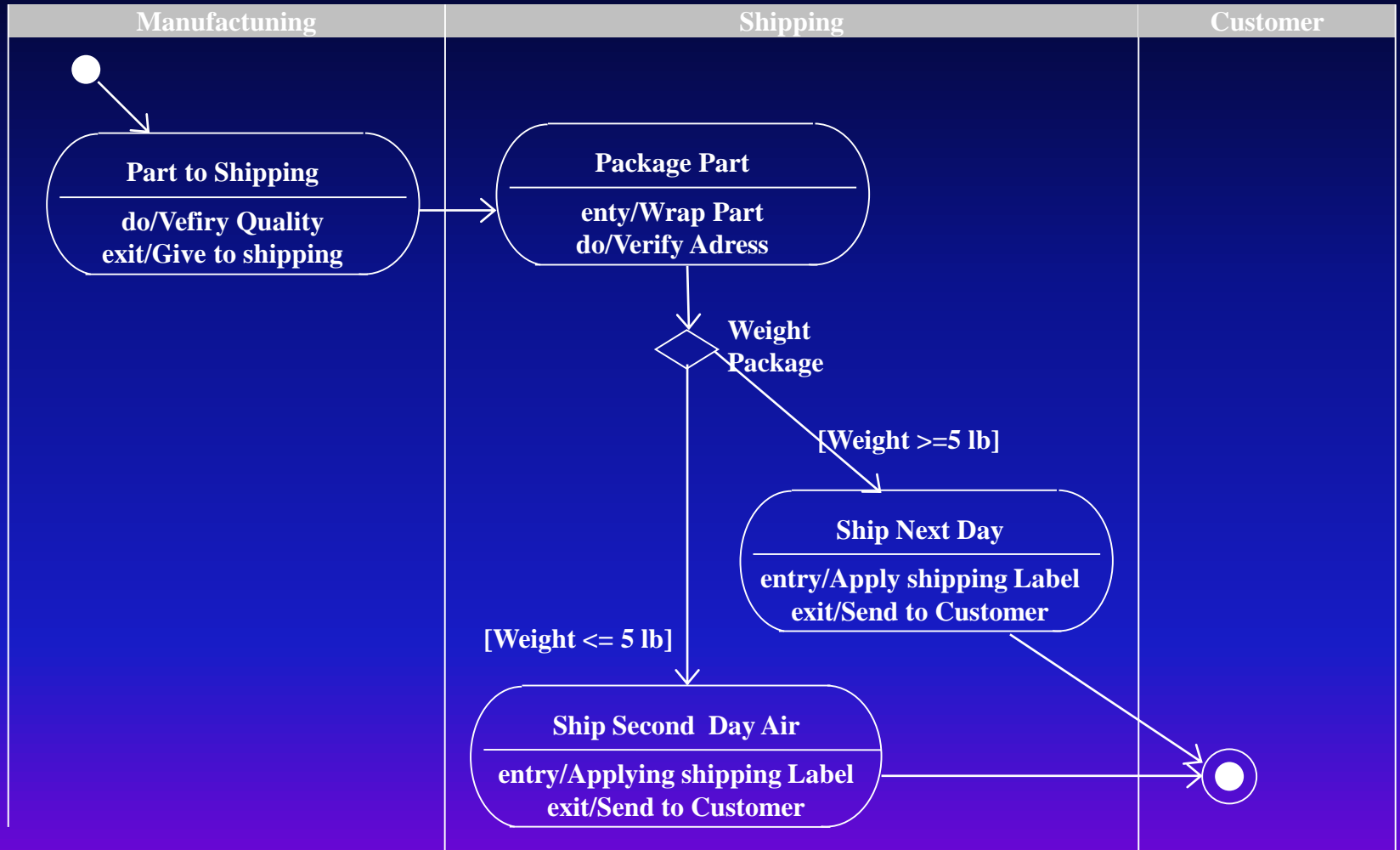
- workflow 建模：系统的业务过程的描述
- 用例图的局限性：

用例图是以系统的各独立功能为单位描述的，并不涉及到整体业务过程，有时需要对业务过程进行必要的描述。

例：产品制造和发货过程在用例图中无法表示



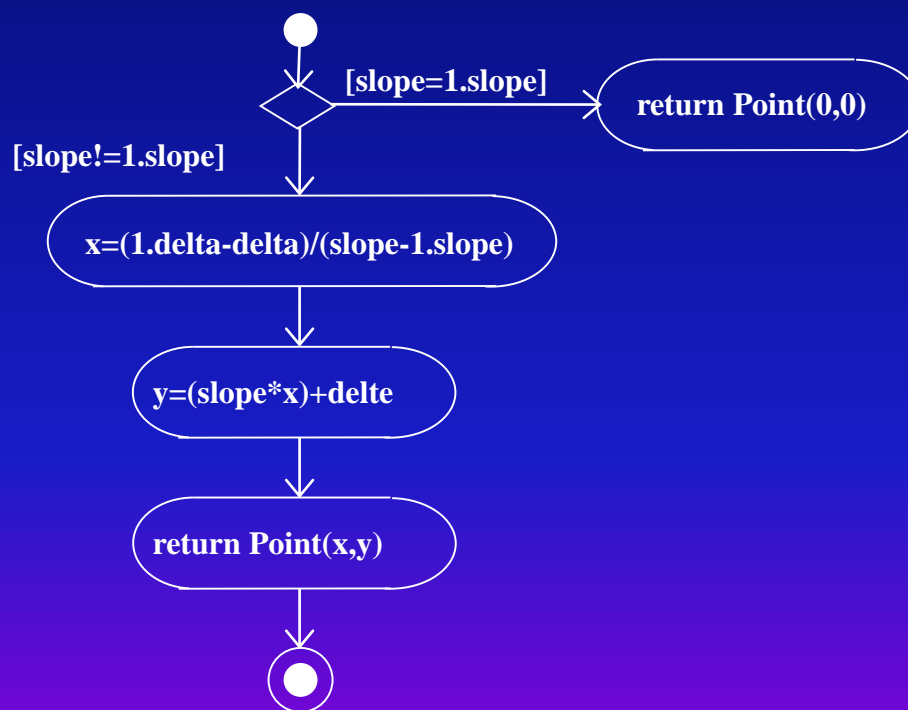
## 例：用活动图描述产品制造和发货过程



## ■用活动图对具体操作建模

用活动图描述具体算法，类似于结构化分析时的流程图

例：Line类的求直线焦点的算法



# 实现建模

构件和部署基本概念

构件图



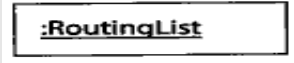
部署图



# 构件图概要



构件图用于静态建模，是表示构件类型的组织以及各种构件之间依赖关系的图。  
构件图通过对构件间依赖关系的描述来估计对系统构件的修改给系统可能带来的影响。

## 构件图中的事物及解释

事物名称	含义	图例
构件	指系统中可替换的物理部分，构件名字(如图中的Dictionary)标在矩形中，提供了一组接口的实现。	
接口	外部可访问到的服务 (如图中的Spell-check)。	
构件实例	节点实例上的构件的一个实例，冒号后是该构件实例的名字(如图中的RoutingList)。	

可替换的物理部分包括软件代码、脚本或命令行文件，也可以表示运行时的对象，文档，数据库等。  
节点(node)是运行时的物理对象，代表一个计算机资源。具体请参见教程“部署图(deployment diagram)”部分。

## 构件图中的关系及解释

关系名称	含义	图例
实现关系	构件向外提供的服务。	
依赖关系	构件依赖外部提供的服务(由构件到接口)。	

# 构件和部署基本概念

## 系统设计工具

构件图Component Diagram 与部署图Deployment Diagram 是在系统设计时，用来表示系统软件成分以及之间关系结构的工具。

## 物理事物建模：

分析构件及其间的关系，并对它们在运行节点上的成分给与描述，也叫“物理事物建模”。

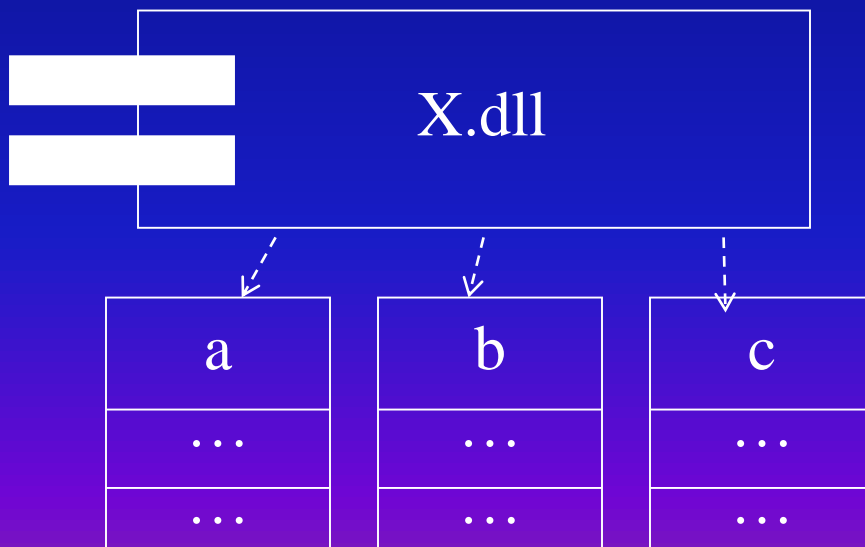
# 构件图Component Diagram

## ■UML中的构件:

提供单个或组接口，物理上可替换的软件实现单元



## ■构件与类之间的关系:



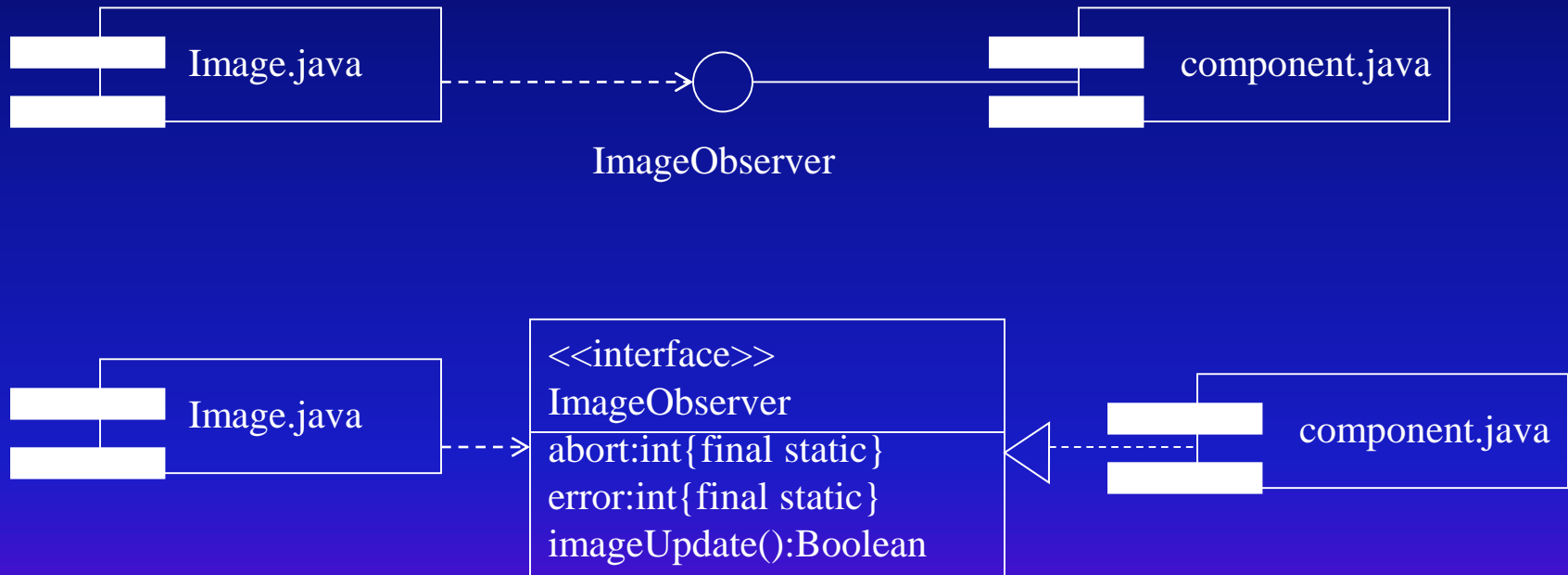
## ■构件与类差别:

- 类表示逻辑抽象，是逻辑模块；
- 构件表示机器空间中的物理模块，是逻辑元素及协作关系的物理实现；
- 类有属性和操作；
- 构件仅通过接口向外提供可请求的操作。

## ■ 构件的接口

构件接口是构件提供的操作集合；

构件之间接口的表示：



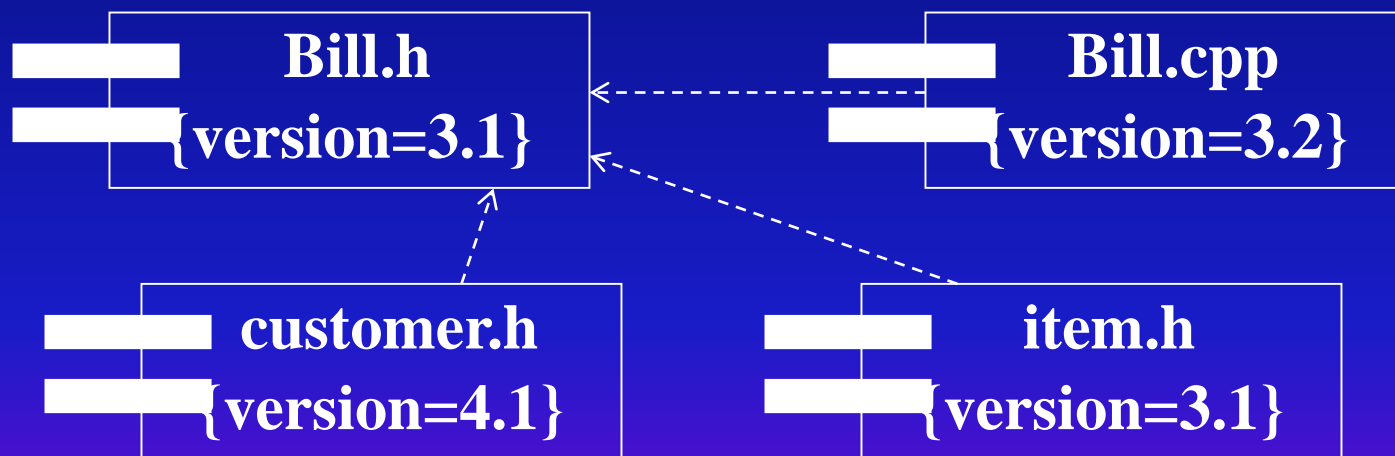
## ■构件的种类

- 部署构件：  
可用于构造的执行系统，如：动态链接库（**DLL**）和可执行程序（**EXE**）；
- 产品构件：  
开发过程的产物，包括创建部署构件的源代码文件及数据文件等；
- 可执行构件：  
由执行系统创建的构件；

## ■产品构件建模

- 用《file》标识一组相关源代码文件的集合
- 给出源代码文件的版本号、作者名、修改日期等
- 用依赖关系标出源代码文件之间的编译依赖关系

产品构件建模示例：

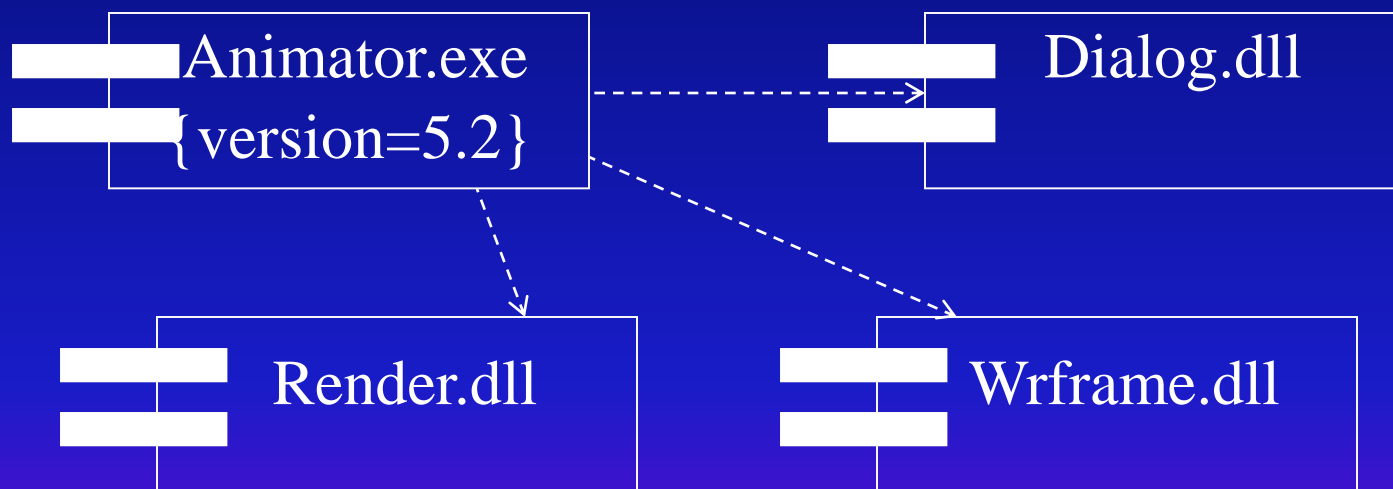


## ■部署构件建模

①表示可执行程序 and 动态连接库的构件；

②表示可执行程序与动态连接库及接口之间的关系；

对可执行体建模示例：

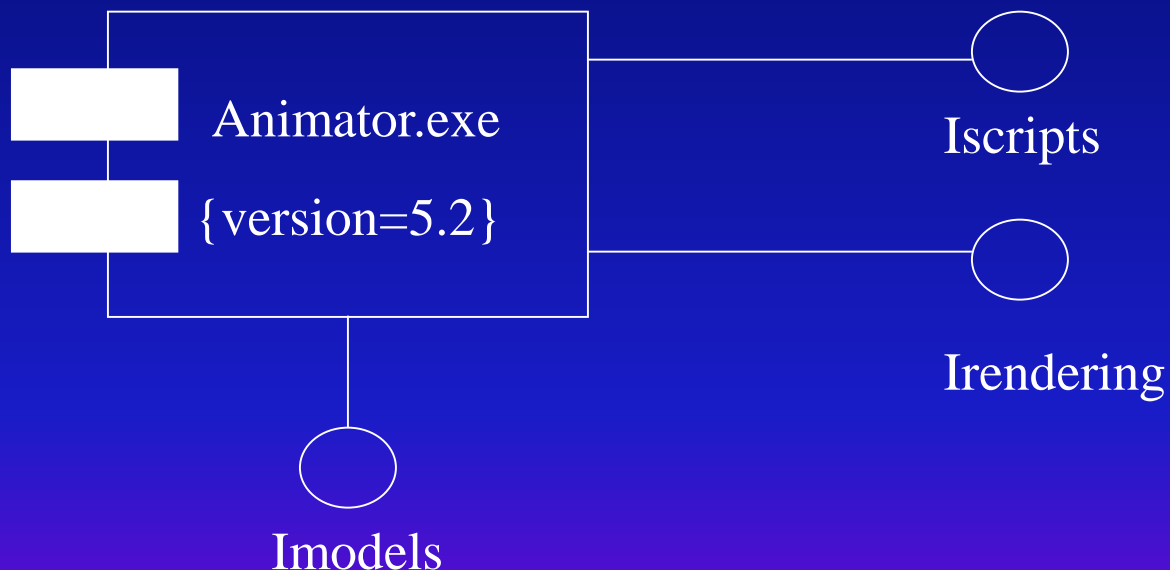




## ■应用接口API建模

API表示系统可编程的界面，可用接口和构件表示

对API建模示例

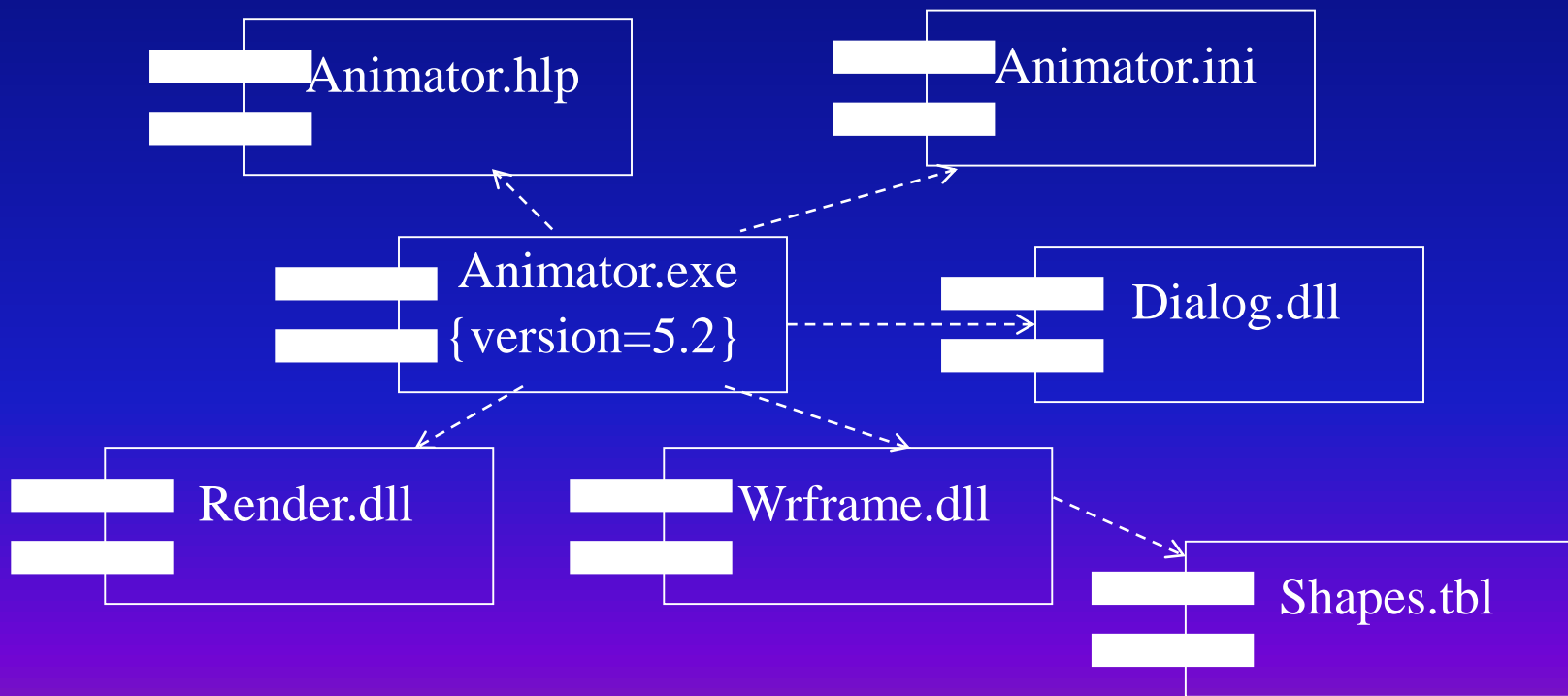


## ■可执行构件建模

①标识系统物理实现部分的附属构件

②构件与可执行程序、动态连接库及接口的关系

对表、文件和文档建模示例：



# 部署图 Deployment Diagram

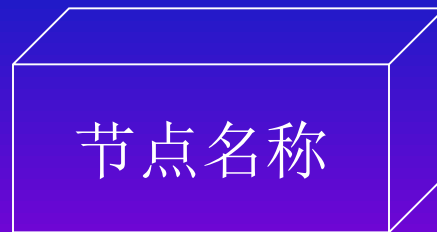
## ■部署图

表示系统在一个或多个物理节点上运行的拓扑结构。物理节点是可部署构件的处理器或设备。

## ■节点

是具有独立存储空间，运行时存在，并代表一项计算资源的物理元素和执行能力。

节点表示法：

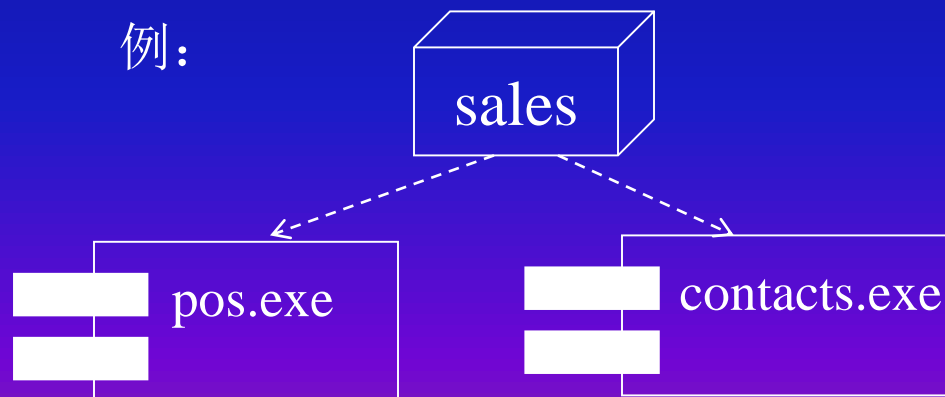


## ■节点和构件

- ①构件是系统执行的事物，节点是执行构件的事物。
- ②构件代表逻辑元素的物理打包，节点可表示构件的物理部署。

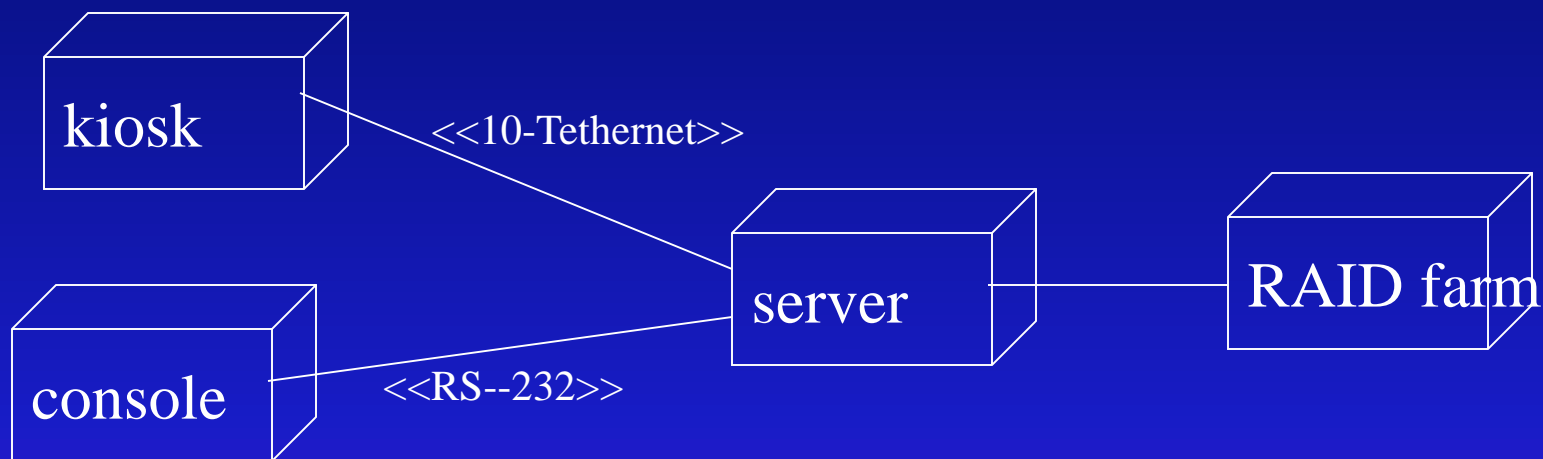
## ■节点和构件的关系：

节点上可以有一个或多个构件，一个构件也可以部署在一个或多个节点上。



■节点之间的关联关系：用来表示节点之间的物理连接。

节点之间的连接示例：



## ■建立部署图

- 部署图通包含节点、节点间的关联关系、构件以及构件和节点间的依赖关系。
- 部署图中部署的构件和可执行构件都必须存在于某些节点上。

使用部署图对以下两种系统建模：

### ①嵌入式系统建模

用部署图描述嵌入式系统的处理器、设备以及构件在其上的分布情况。

### ②分布式系统建模

用部署图描述分布式系统的网络拓扑结构以及构件在其上的分布情况。

# 软件设计概述

软件设计是软件开发的关键步骤，直接影响软件质量。

软件设计阶段要解决“**如何做**”的问题。

## 一、软件设计阶段的任务与目标

设计任务：将需求阶段获得的需求说明（模型）转换为计算机中可实现的系统。

设计阶段主要任务是：

- 软件体系结构设计
- 数据结构设计
- 界面设计
- 过程设计

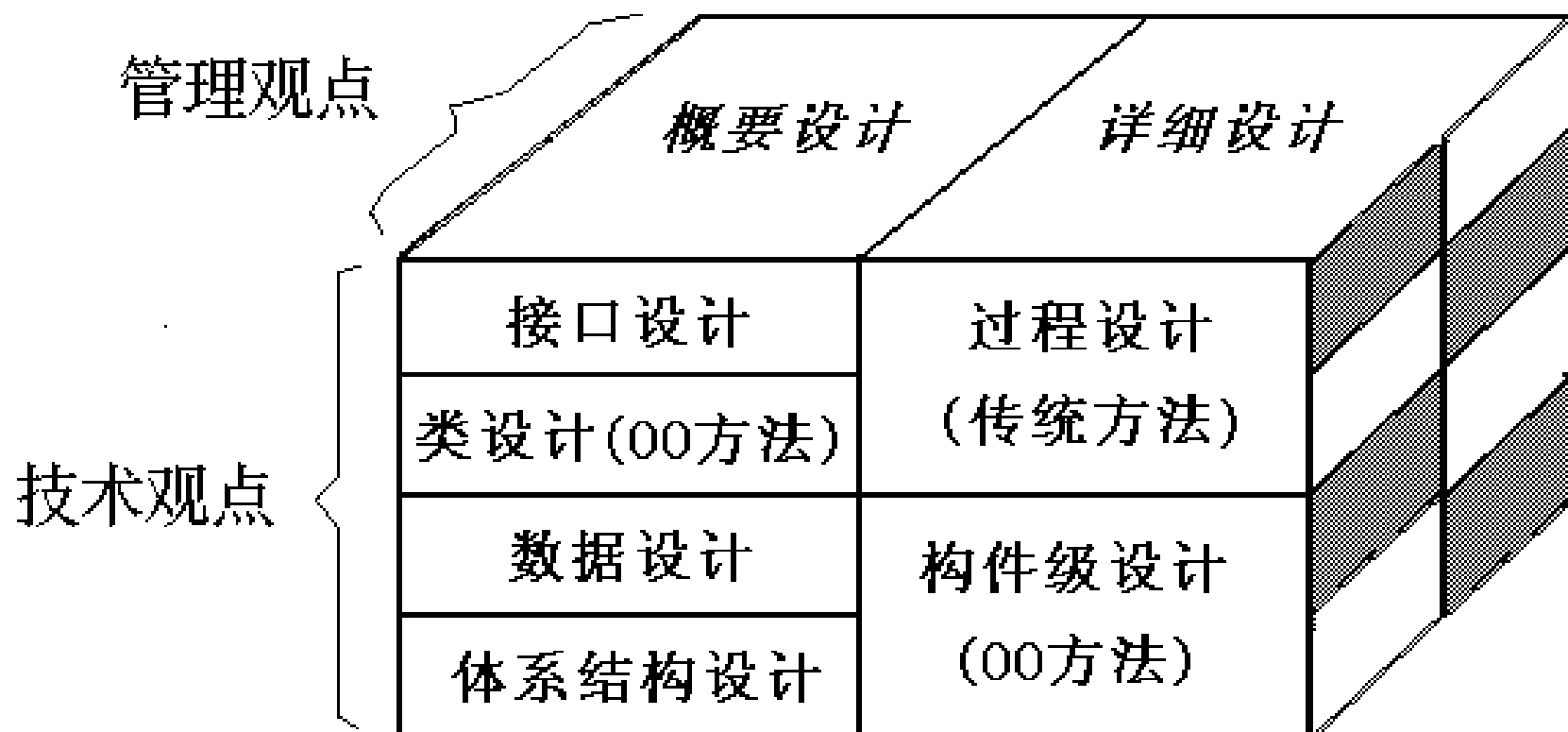
# 软件设计概述

## 设计阶段：

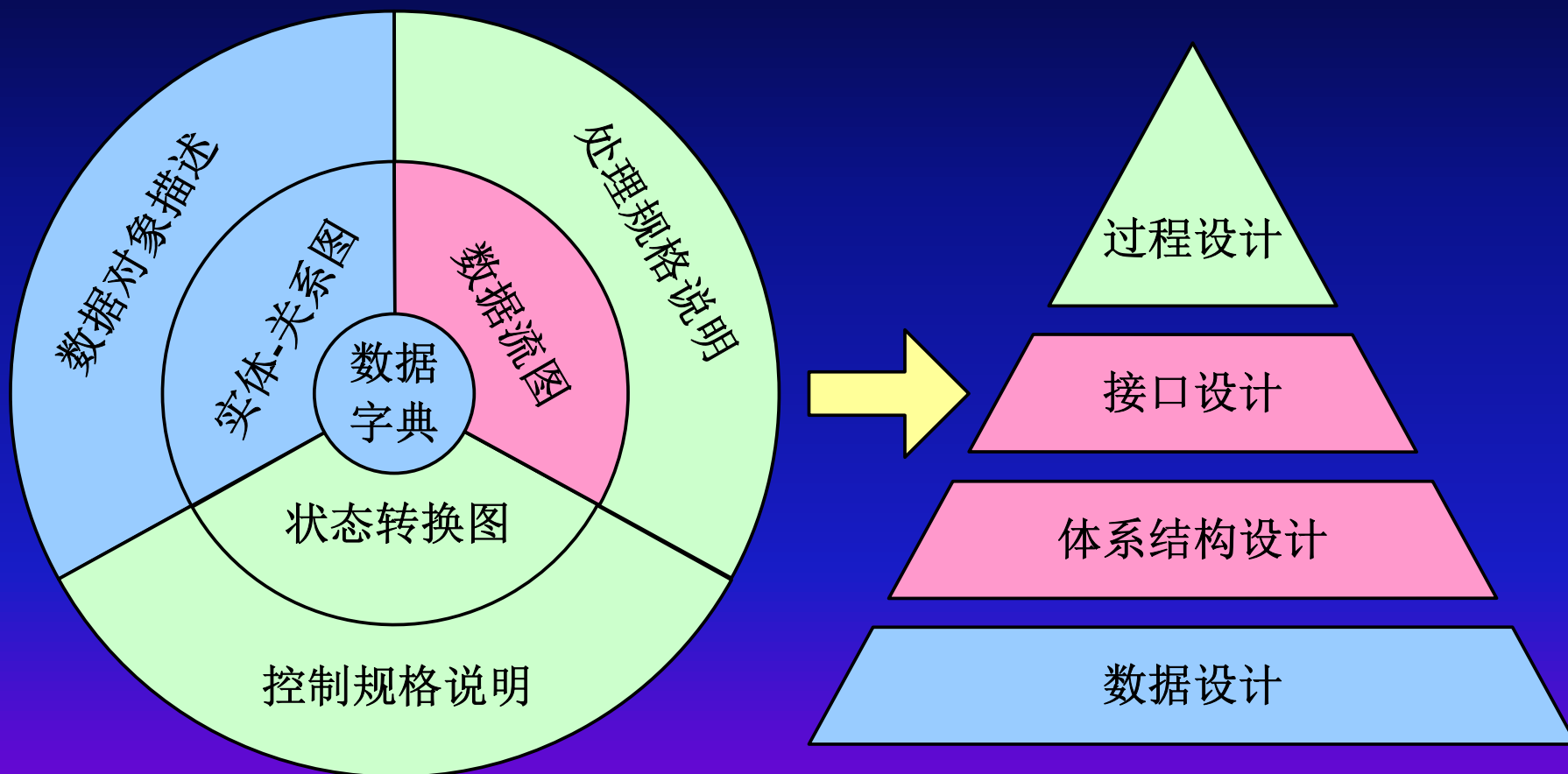
- 从工程管理的角度，可以将软件设计分为概要设计阶段和详细设计阶段。
- 从技术的角度，传统的结构化方法将软件设计划分为体系结构设计、数据设计、接口设计和过程设计4部分。
- 面向对象方法则将软件设计划分为体系结构设计、类设计/数据设计、接口设计和构件级设计4部分。



# 软件设计概述



# 结构化设计和结构化分析的关系：



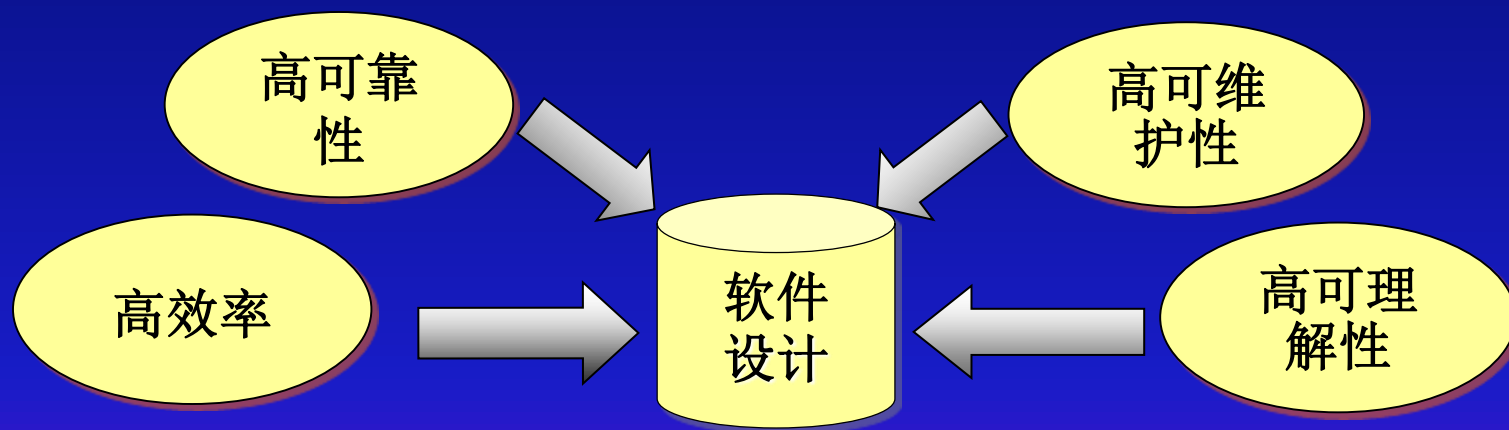
# 软件设计概述

软件设计任务涉及多方面，可分为“总体设计”和“详细设计”。



# 软件设计概述

软件设计的目标 就是构造一个**高内聚低耦合**的软件模型。



软件设计的目标

# 软件设计

## 软件设计的划分：

### (1) 总体设计（概要设计）

确定软件的结构以及各组成成分(子系统或模块)之间的相互关系。

### (2) 技术设计（详细设计）

确定模块内部的算法和数据结构，产生描述各模块程序过程的详细文档。

# 软件设计

- **总体设计过程：**首先寻找实现目标系统的各种不同的方案；然后分析员从这些供选择的方案中选取若干个合理的方案，从中选出一个最佳方案向用户和使用部门负责人推荐；分析员应该进一步为这个最佳方案设计软件结构，进行必要的数据库设计，确定测试要求并且制定测试计划。
- **必要性：**总体设计可以站在全局高度上，花较少成本，从较抽象的层次上分析对比多种可能的系统实现方案和软件结构，从中选出最佳方案和最合理的软件结构，从而用较低成本开发出较高质量的软件系统。

# 软件设计

典型的总体设计过程包括下述9个步骤：

## 1. 设想供选择的方案

- 根据需求分析阶段得出的数据流图考虑各种可能的实现方案，力求从中选出最佳方案。

## 2. 选取合理的方案

- 从前一步得到的一系列供选择的方案中选取若干个合理的方案。对每个合理的方案分析员都应该准备下列4份资料：
  - 系统流程图；
  - 组成系统的物理元素清单；
  - 成本/效益分析；
  - 实现这个系统的进度计划。

# 软件设计

## 3. 推荐最佳方案

- 分析员应该综合分析对比各种合理方案的利弊，推荐一个最佳的方案，并且为推荐的方案制定详细的实现计划。

## 4. 功能分解

- 首先进行结构设计，然后进行过程设计。
- 结构设计确定程序由哪些模块组成，以及这些模块之间的关系；过程设计确定每个模块的处理过程。
- 结构设计是总体设计阶段的任务，过程设计是详细设计阶段的任务。



# 软件设计

## 5. 设计软件结构

- 通常程序中的一个模块完成一个适当的子功能。应该把模块组织成良好的层次系统。软件结构可以用层次图或结构图来描绘。
- 如果数据流图已经细化到适当的层次，则可以直接从数据流图映射出软件结构，这就是面向数据流的设计方法。

## 6. 设计数据库

- 对于需要使用数据库的那些应用系统，软件工程师应该在需求分析阶段所确定的系统数据需求的基础上，进一步设计数据库。

# 软件设计

## 7. 制定测试计划

- 在软件开发的早期阶段考虑测试问题，能促使软件设计人员在设计时注意提高软件的可测试性。

## 8. 书写文档

- 应该用正式的文档记录总体设计的结果，在这个阶段应该完成的文档通常有下述几种：
  - (1) 系统说明；
  - (2) 用户手册；
  - (3) 测试计划；
  - (4) 详细的实现计划；
  - (5) 数据库设计结果。

## 9. 审查和复审

- 最后应该对总体设计的结果进行严格的技术审查和管理复审。

# 软件设计

- 设计 应该展示系统的层次结构。
- 设计 应该模块化与信息隐藏。
- 设计 应该遵循分解与求精的原则。
- 设计 应该包括数据、体系结构、接口和模块的清楚表示。
- 设计过程是迭代过程，基于需求所获取的信息量。
- 设计应该有一致性和集成性。

# 软件体系结构设计

软件体系结构确定了系统的组织结构和拓扑结构，显示了系统需求和构成系统的元素之间的对应关系，提供了一些设计决策的基本原理。

体系结构的设计过程的主要活动：

1. 系统分解—将系统分解为若干相互作用的子系统。
2. 控制建模—建立系统各部分间控制关系的一般模型。
3. 模块分解 — 将子系统进一步划分为模块。

# 软件体系结构设计

体系结构设计是软件设计的第一个阶段，该阶段侧重于系统“**宏观**”结构的设计，而不关心模块的内部算法。

体系结构的分类：

## 一、仓库模型 (The repository model)

也称“容器模型”，是一种集中式的模型。各子系统可以直接访问中央数据仓库存储的共享数据。子系统之间紧密耦合。



仓库结构

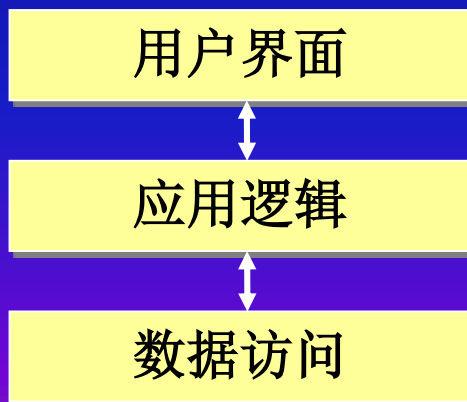
# 软件体系结构设计

## 二、 客户机/服务器模型 (Client/Server Architectural Model)

C/S结构是一种分布式模型，采用发请求、得结果的模式：

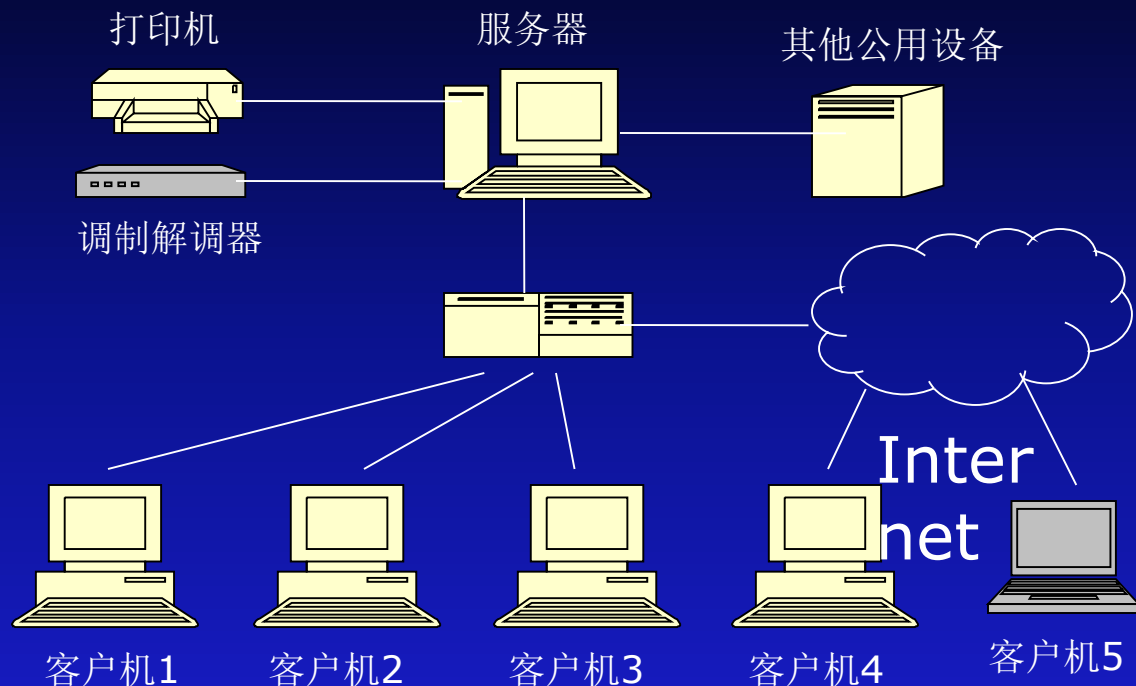
- 客户机 向服务器发出请求(数据请求、网页请求、文件传输请求等等)，
- 服务器 响应请求，进行相应的操作，将结果回传给客户机，客户机再将格式化后的结果呈现给用户。

C/S结构的应用都由三个相对独立的逻辑部分组成：



三种逻辑之间的关系

## C/S体系结构硬件示意图



客户机是具有较强性能的微机系统，在自身操作系统的控制下，执行着运行在其内部的应用系统，并且向服务器发送消息，以完成文件存取或数据库访问等服务。

C/S结构可实现分布在多个营业点的大型分布式业务系统。

# 浏览器/服务器 ( Browser/Server)

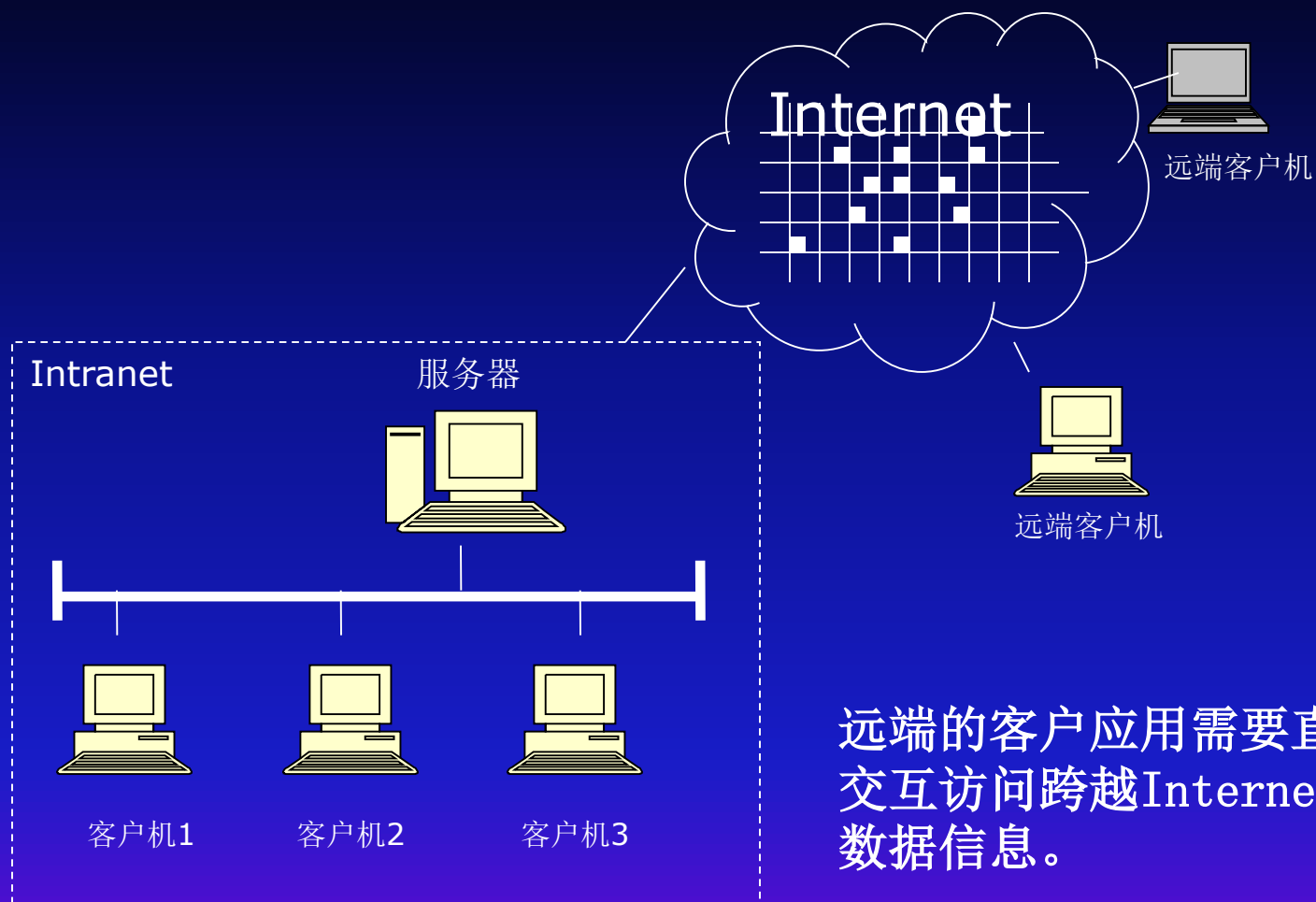
随着Internet技术的发展，大量的应用需要在广域网络上运行，而C/S体系结构不适合广域网应用需求：

- 以特定服务器局域网为中心的频繁的交互操作方式，不能适应多节点传输的广域网应用。
- 客户应用系统的操作直接对应数据库信息格式，广域网络的应用系统不能要求必须针对具体的数据库信息格式。

B/S体系结构是在成熟的WWW(World Wide Web)技术基础上，并且，继承了C/S结构的基本形式和内容，从两层结构扩展为三层体系结构。

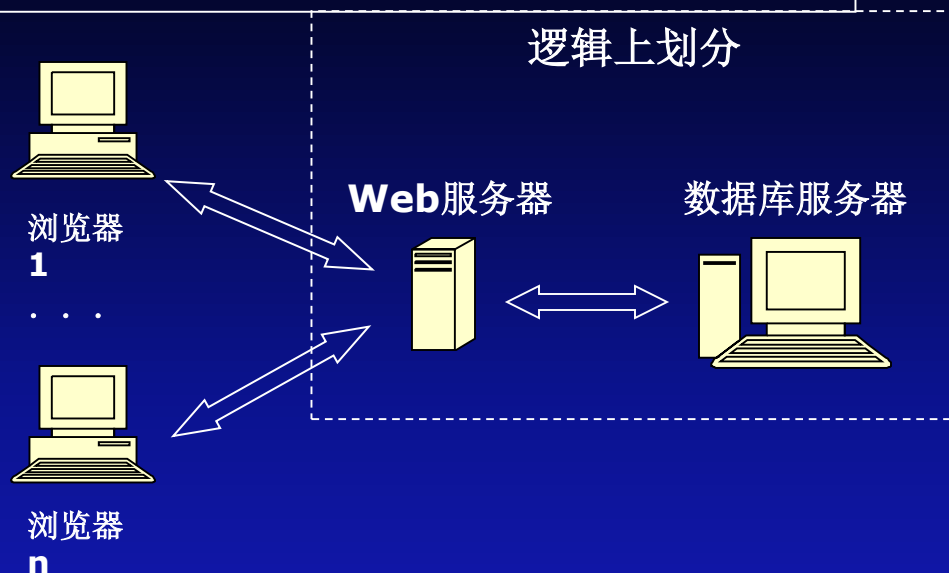


## 更广泛的应用结构形式



远端的客户应用需要直接交互访问跨越Internet的数据信息。

# B/S体系结构逻辑示意



客户浏览器:

浏览器程序  
(0客户端)

Web服务器:

应用界面处理  
应用业务处理  
数据库信息转换

数据库服务器:

数据读写  
数据检索和更新

## B/S体系结构存在的问题

- 操作交互处理和业务处理逻辑紧密相关，不能单独修改或复用。
- 业务处理的输出信息是以全体共识的方式直接发往客户端的，所以外界可直接了解系统的数据结构，安全性差。
- Web服务器权力过于集中，处理业务加上处理传输，使负载过大，系统处理中回避风险的能力降低。

## B/S多层体系结构

将**Web**服务器分为两层或多层的结构。

**Web**服务器:

界面规范化  
数据校验

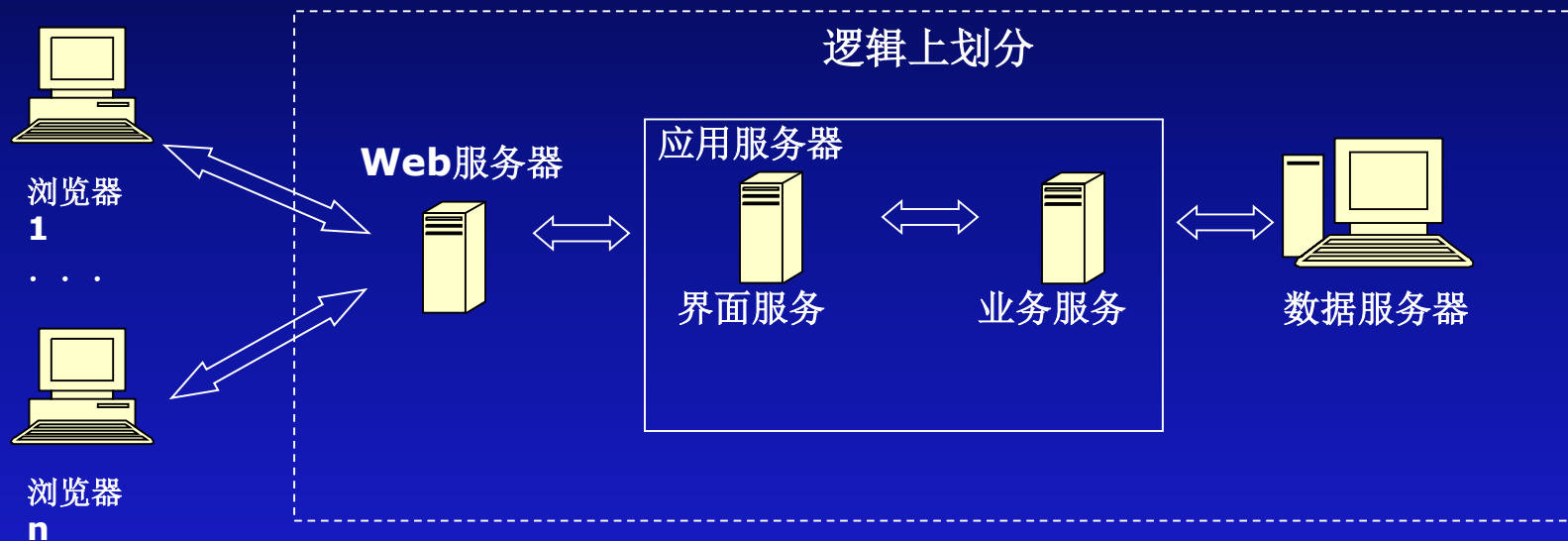
应用服务器:

业务处理  
数据访问

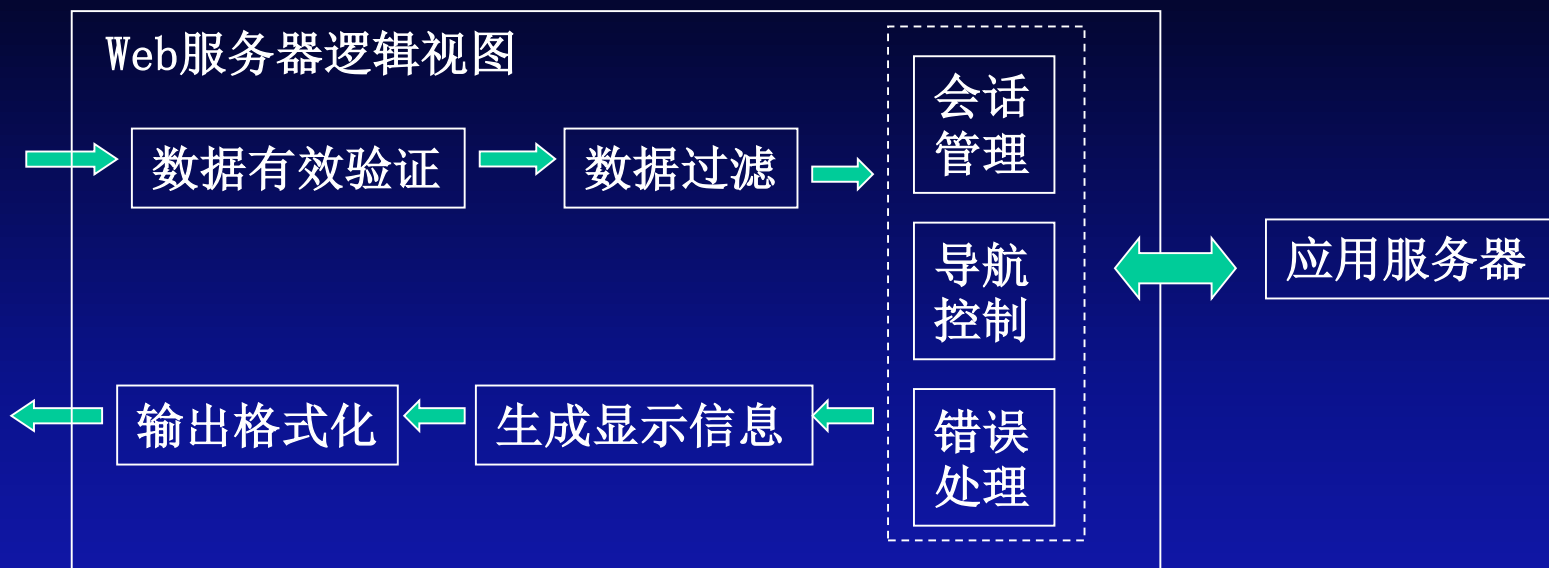
数据服务器:

数据访问

# B/S多层体系逻辑结构



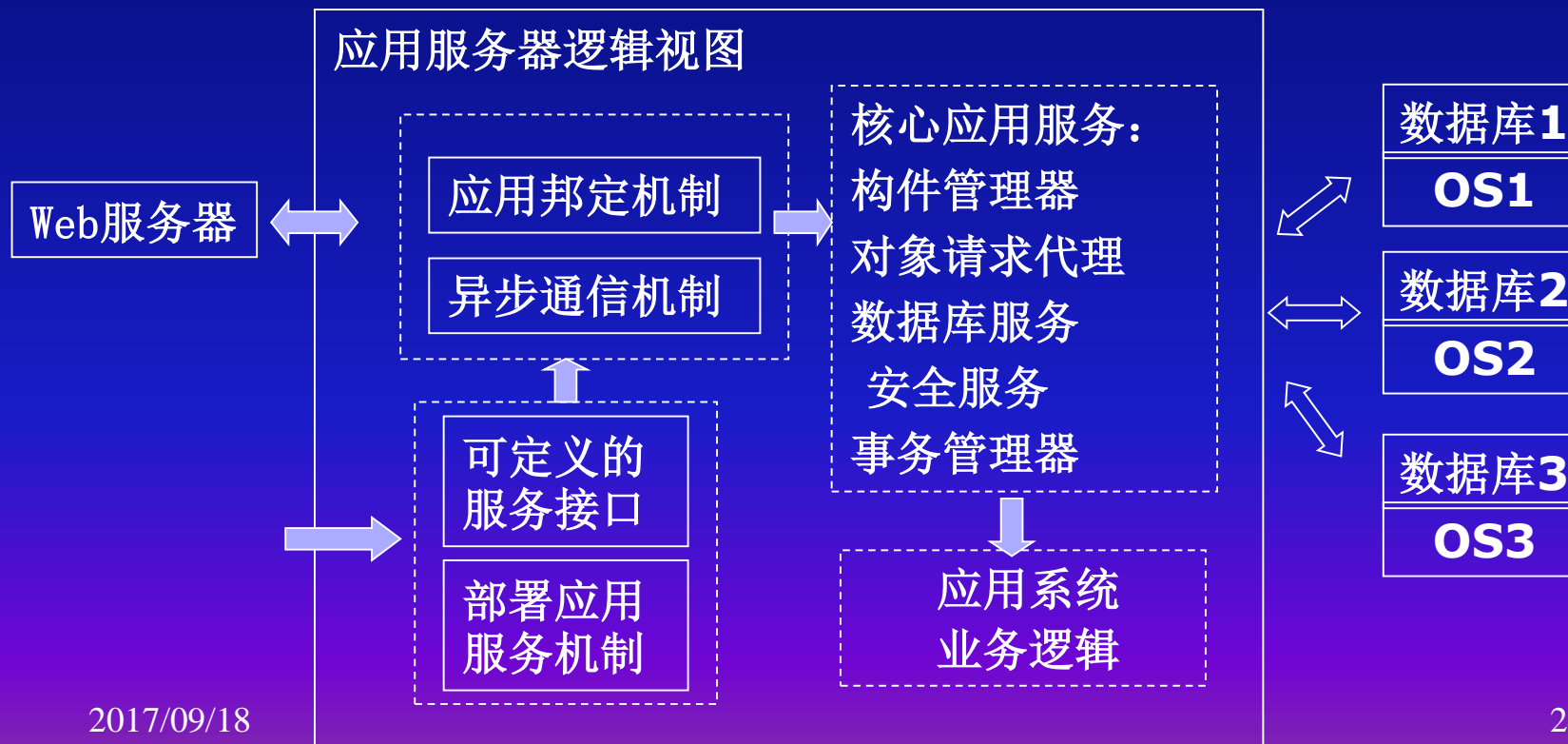
# Web服务器上的工作



- 数据验证：确保数据在提交前的正确有效性。
- 数据过滤：输入数据转换、过滤空格、大小写转换等。
- 会话管理：保证操作在各页面间正确传递，转发必要的信息。
- 生成显示信息：根据任务请求动作，生成显示信息。
- 输出格式化：转换为浏览器显示格式。
- 导航控制：保证应用程序的页面导航控制流。
- 错误处理

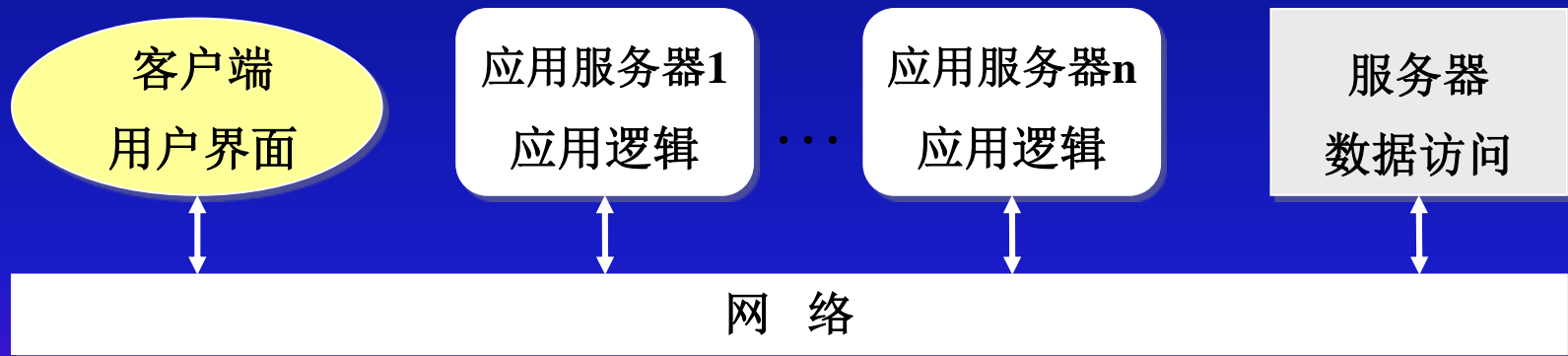
## 应用服务器的工作

- 支持将后端的应用程序绑定到多种不同的客户机
- 处理事务在分布式系统的各部分异步通信的机制
- 提供核心应用服务
- 提供描述服务以及定义服务接口
- 安装应用部署机制



# 软件体系结构设计

在多层模型中，中间层会用到应用服务，包括事务服务、消息服务等等。



多层应用模型

# 软件体系结构设计

## 三、分布式对象结构(Distributed Objects Architecture)

在C/S模型中，客户和服务器的服务/请求上的差别，在一定程度上限制了系统的灵活性和可扩展性。

采用分布式对象结构：

- “对象(Object)”——提供服务的系统组件(System component)。
- 每个对象在逻辑上是平等的，它们可以互相为对方提供所需的服务。
- 提供服务的对象就是服务器，而提出服务请求的对象就是客户。



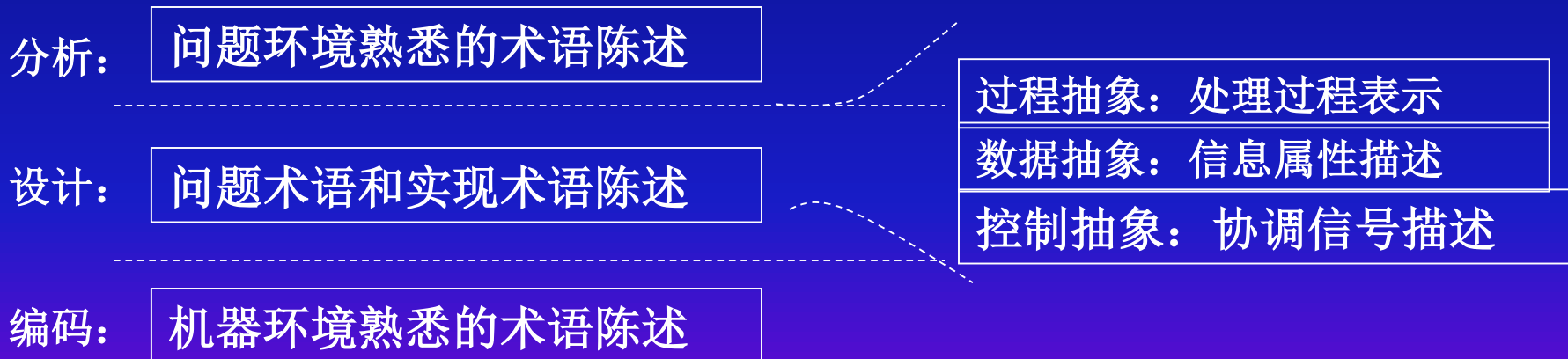
# 软件体系结构设计

## 1) 抽象:

集中于问题的一般性的概念，而忽略无关的细节。

## 2) 逐步求精:

进一步详细描述的过程，与抽象可达到互补的作用。



# 抽 象

- **抽象**：现实世界中一定事物、状态或过程之间总存在着某些相似的方面(共性)。把这些相似的方面集中和概括起来，暂时忽略它们之间的差异，这就是抽象。
- 抽象就是抽出事物本质特性而暂时不考虑细节。
- “抽象是人类处理复杂问题的基本方法之一。”  
——Grady Boach

# 抽象

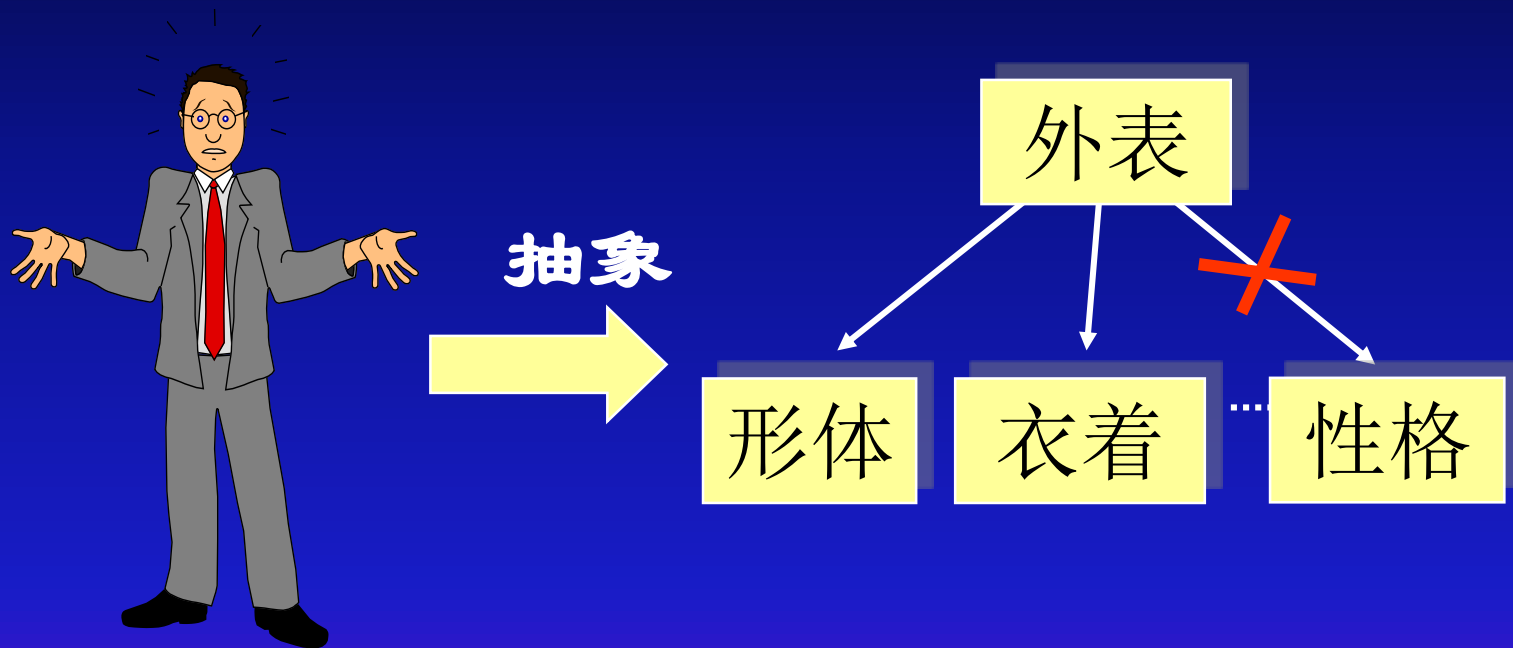
## 一般抽象过程:

- 处理复杂系统的惟一有效的方法是用层次的方式构造和分析它。
- 一个复杂的动态系统首先可以用一些高级的抽象概念构造和理解，这些高级概念又可以用一些较低级的概念构造和理解，如此进行下去，直至最低层次的具体元素。
- 例：过程抽象、数据抽象

开(行为抽象) + 门(数据抽象)

# 抽象

## 抽象例子



# 抽象

## 软件工程抽象过程：

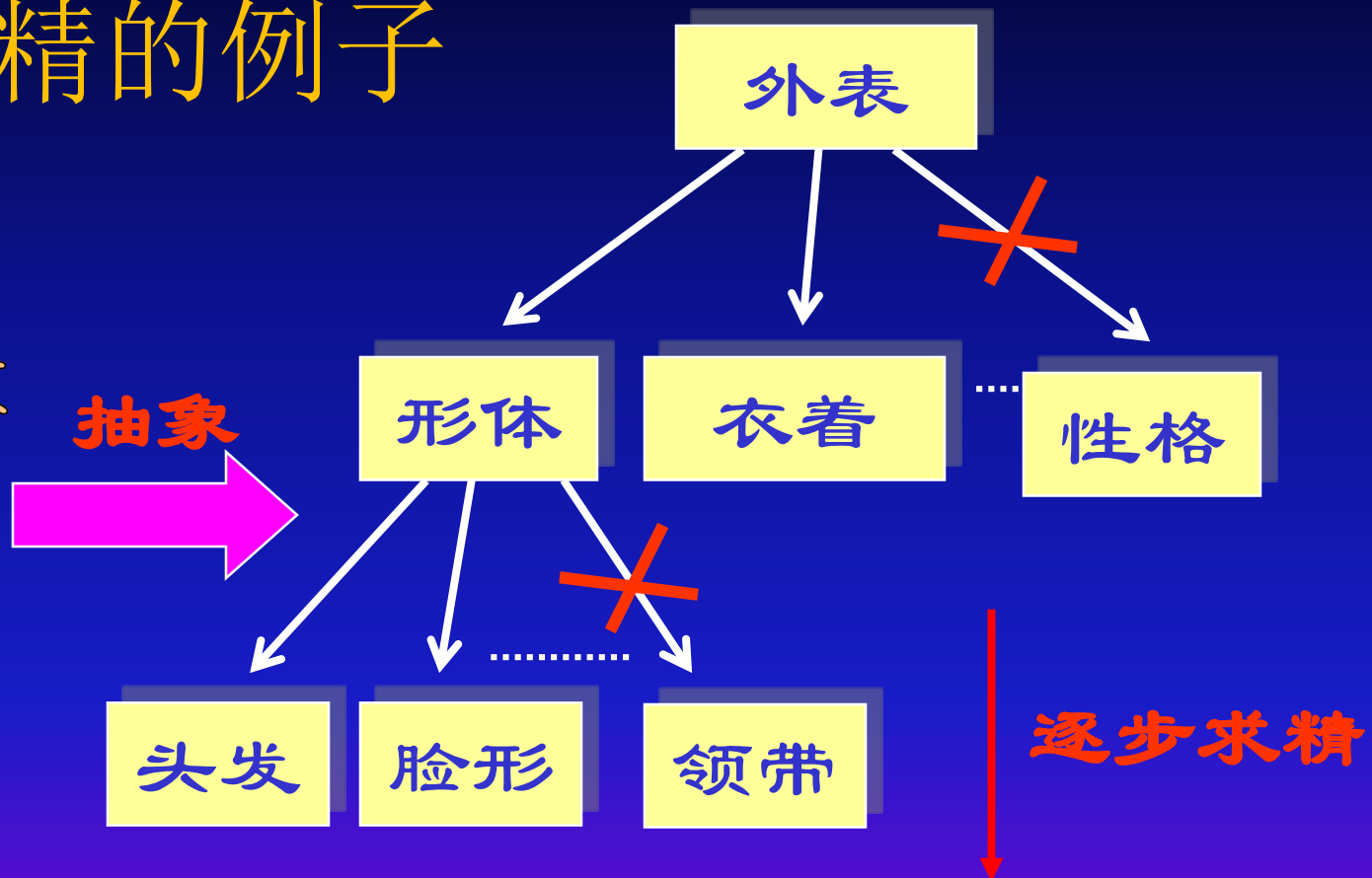
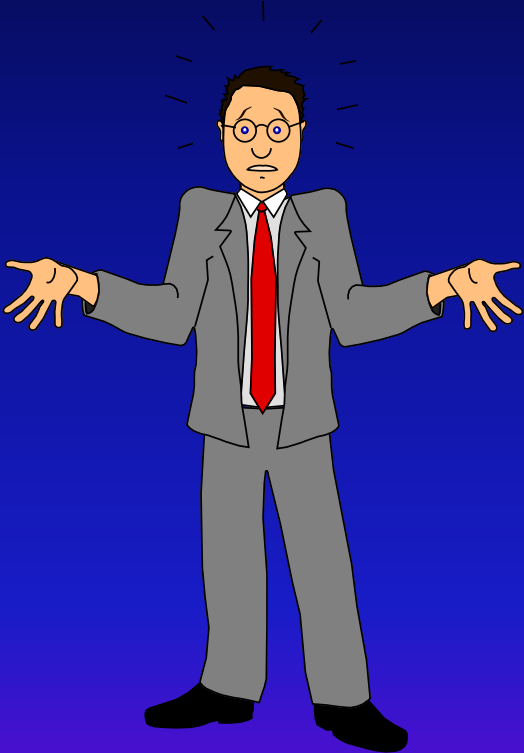
- 软件工程过程的每一步都是对软件解法的抽象层次的一次精化。
- 在可行性研究阶段，软件作为系统的一个完整部件；
- 在需求分析期间，软件解法是使用在问题环境内熟悉的方式描述的；
- 当由总体设计向详细设计过渡时，抽象的程度也就随之减少了；
- 最后，当源程序写出来以后，也就达到了抽象的最低层。

# 逐步求精

- **逐步求精**：为了能集中精力解决主要问题而尽量推迟对问题细节的考虑。逐步求精是人类解决复杂问题时采用的基本方法，也是许多软件工程技术的基础。
- **Miller法则**：一个人在任何时候都只能把注意力集中在  $(7 \pm 2)$  个知识块上。

# 逐步求精

## 逐步求精的例子



# 逐步求精

用筛选法求100以内的素数。所谓的筛选法，就是从2到100中去掉2，3，5，7的倍数，剩下的就是100以内的素数。

- 首先按程序功能写出一个框架

```
main()
```

```
{
```

```
    建立2到100的数组A[ ]，其中A[i]=i; .....1
```

```
    建立2到10的素数表B[ ]，存放2到10以内的素数; .....2
```

```
    若A[i]=i是B[ ]中任一数的倍数，则剔除A[i]; .....3
```

```
    输出A[ ]中所有没有被剔除的数; .....4
```

```
}
```



# 逐步求精

## 逐步求精的作用：

- 它能帮助软件工程师把精力集中在与当前开发阶段最相关的那些方面上，而忽略那些对整体解决方案来说虽然是必要的，然而目前还不需要考虑的细节。
- 逐步求精方法确保每个问题都将被解决，而且每个问题都将在适当的时候被解决，但是，在任何时候一个人都不需要同时处理7个以上知识块。

# 信息隐藏

## 信息隐藏(Information Hiding)

信息隐藏的含义：有效的模块化可以通过定义一组独立模块来实现，这些模块相互之间只交流软件功能必需的信息。

隐藏实现信息局部化：

把关系密切的软件元素物理地放得彼此靠近。

优点：可维护性好、可靠性好、可理解性好。

信息隐藏的目的：

提高模块的独立性，减少修改或维护时的影响面。

# 信息隐藏和局部化

**信息隐藏：**应该这样设计和确定模块，使得一个模块内包含的信息(过程和数据)对于不需要这些信息的模块来说，是不能访问的。

**局部化：**局部化的概念和信息隐藏概念是密切相关的。所谓局部化是指把一些关系密切的软件元素物理地放得彼此靠近。显然，局部化有助于实现信息隐藏。

# 信息隐藏和局部化

## 信息隐藏和局部化的作用：

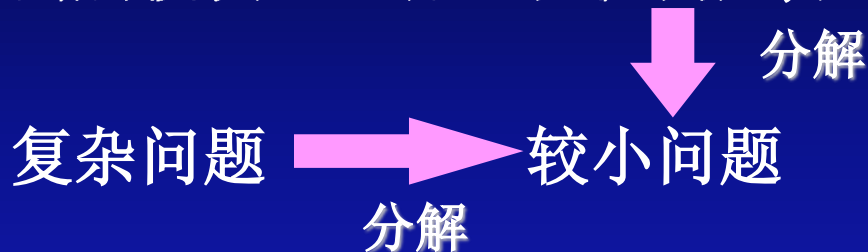
“隐藏”意味着有效的模块化可以通过定义一组独立的模块而实现，这些独立的模块彼此间仅仅交换那些为了完成系统功能而必须交换的信息。

使用信息隐藏原理作为模块化系统设计的标准就会带来极大好处。因为绝大多数数据和过程对于软件的其他部分而言是隐藏的，在修改期间由于疏忽而引入的错误就很少可能传播到软件的其他部分。

# 模块分解

模块化 (Modularity): 好的软件设计的一个基本准则。

高层模块 整体上把握问题, 隐藏细节



可减小解题所需的总的工作

模块独立性含义:

- 模块完成独立的功能;
- 符合信息隐蔽和信息局部化原则;
- 模块间关连和依赖程度尽量小。

# 模块化

- **模块：**是由边界元素限定的相邻程序元素的序列，而且有一个总体标识符代表它。
- **模块化：**就是把程序划分成独立命名且可独立访问的模块，每个模块完成一个子功能，把这些模块集成起来构成一个整体，可以完成指定的功能满足用户的需求。

# 模块化

## 为什么要模块化？

- 模块化是为了使一个复杂的大型程序能被人的智力所管理，软件应该具备的惟一属性。
- 如果一个大型程序仅由一个模块组成，它将很难被人所理解。

# 模块化

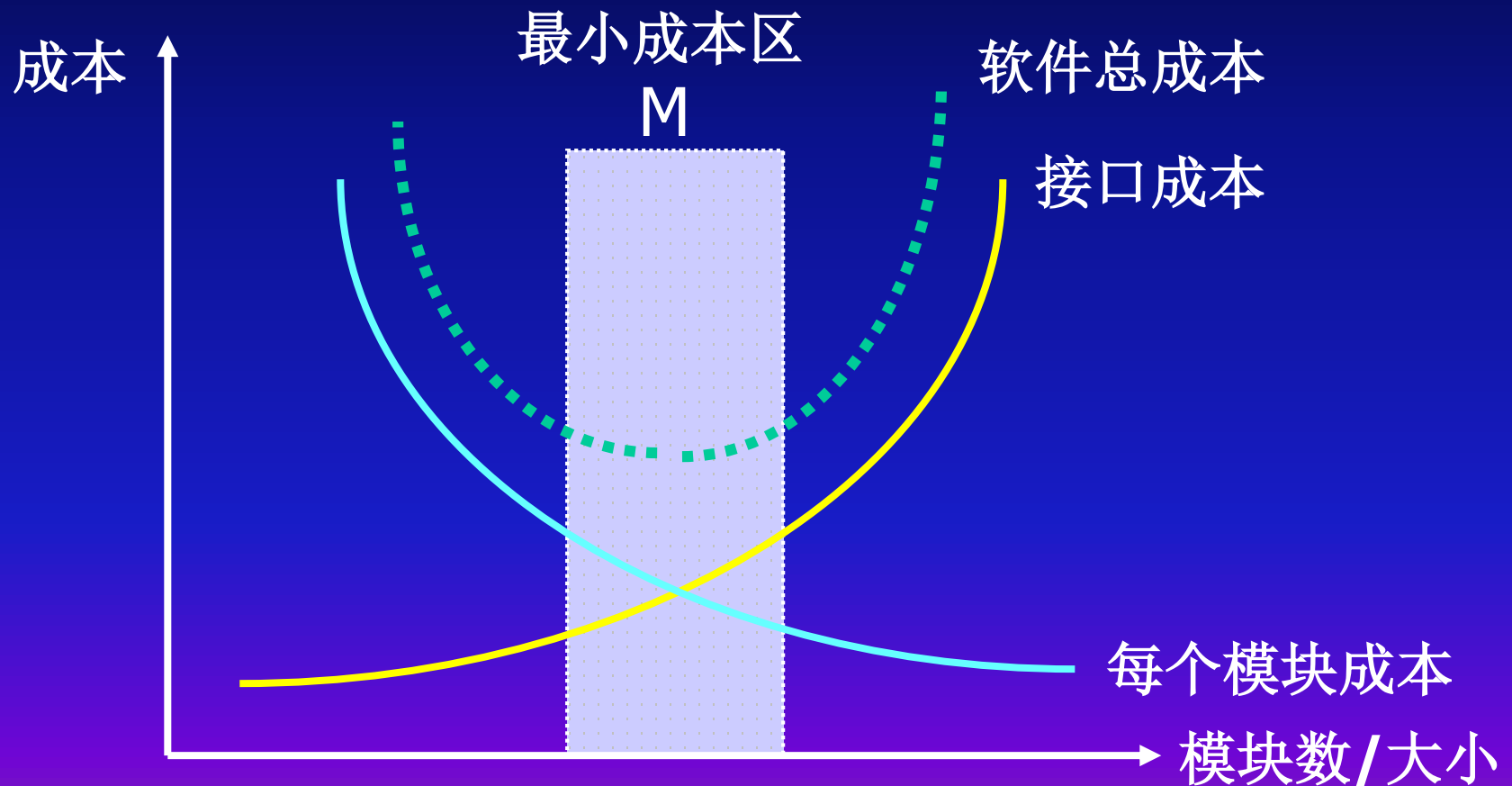
## 模块化的作用：

- 采用模块化原理可以使软件结构清晰，不仅容易设计也容易阅读和理解。
- 模块化使软件容易测试和调试，因而有助于提高软件的可靠性。
- 模块化能够提高软件的可修改性。
- 模块化也有助于软件开发工程的组织管理。



# 模块分解

模块数目、大小与软件成本关系



# 模块分解

模块独立性的度量——独立性取决于模块的内部和外部特征。

*SD方法提出的定性的度量标准:*

- 👉 模块之间的耦合性
- 👉 模块自身的内聚性

模块独立性的度量——耦合性

- 耦合性是指软件结构中模块质检相互连接的紧密程度，是模块间相互连接性的度量。
- 耦合性越高，模块独立性越弱。

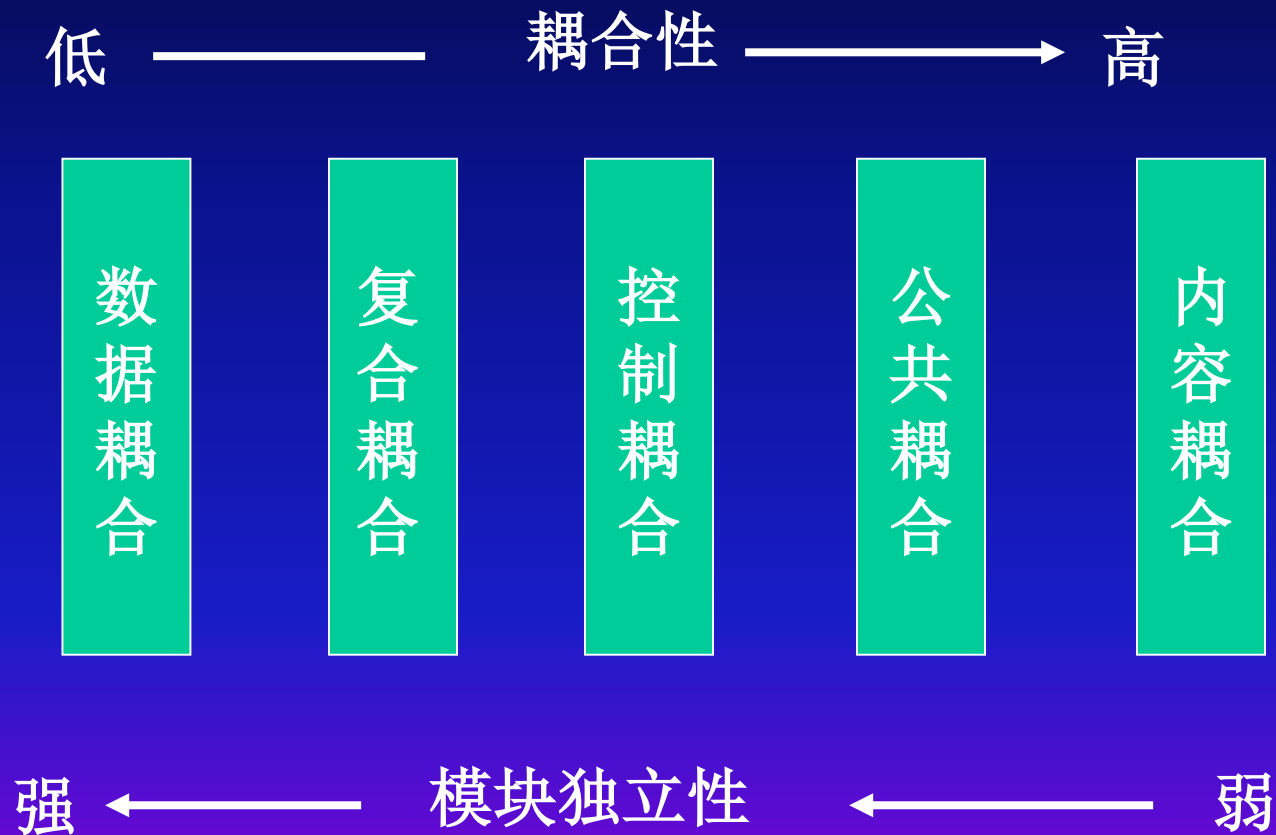
# 模块分解

耦合强度依赖的因素：

- 一模块对另一模块的引用；
- 一模块向另一模块传递的数据量；
- 一模块施加到另一模块的控制的数量；
- 模块间接口的复杂程度；

# 模块分解

## 耦合类型与关系



# 模块分解

耦合是影响软件复杂程度和设计质量的重要因素。

模块化设计目标：建立模块间耦合度尽可能松散的系统。

*原则：*

尽量使用数据耦合

少用控制耦合

限制公共耦合的范围

坚决避免使用内容耦合

# 模块分解

模块独立性的度量——内聚性

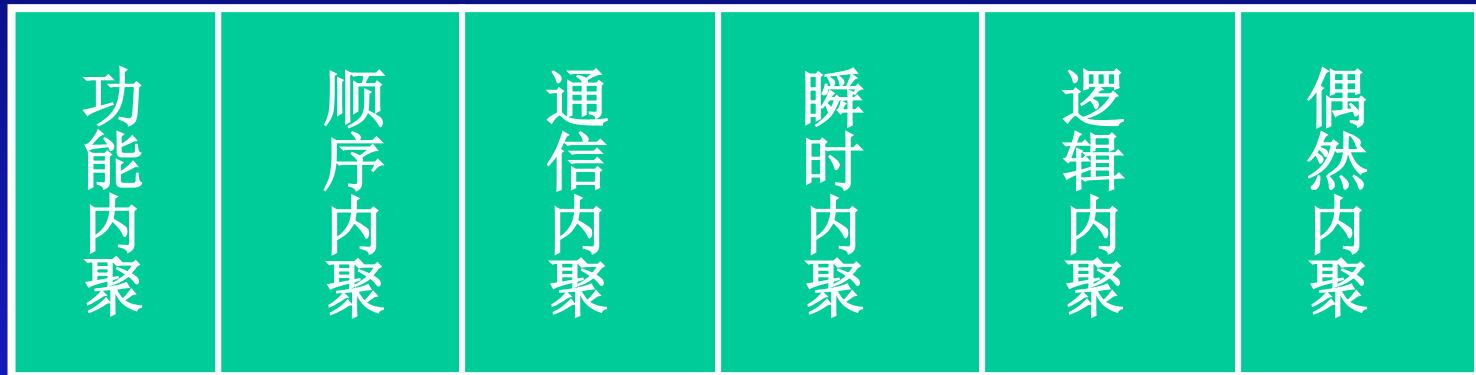
内聚性：一个模块内部各成分之间相互关联的强度

设计目标：高内聚

# 模块分解

## 内聚类型与关系

高 ←—— 内聚性 ——→ 低



强 ←—— 模块独立性 ——→ 弱

# 模块分解

## 内聚与耦合关系：

内聚与耦合密切相关，同其它模块强耦合的模块意味着弱内聚，强内聚模块意味着与其它模块间松散耦合。

## 设计目标：力争强内聚、弱耦合

内聚、耦合是密切相关的，模块内的高内聚往往意味着模块间的松耦合。内聚和耦合都是进行模块化设计的有力工具，但是实践表明内聚更重要，应该把更多注意力集中到提高模块的内聚程度上。



# 模块独立

## 模块独立:

- 模块独立的概念是模块化、抽象、信息隐藏和局部化概念的直接结果。
- 希望这样设计软件结构，使得每个模块完成一个相对独立的特定子功能，并且和其他模块之间的关系很简单。

## 模块独立的重要性:

- 有效的模块化(即具有独立的模块)的软件比较容易开发出来。这是由于能够分割功能而且接口可以简化, 当许多人分工合作开发同一个软件时, 这个优点尤其重要。
- 独立的模块比较容易测试和维护。这是因为相对说来, 修改设计和程序需要的工作量比较小, 错误传播范围小, 需要扩充功能时能够“插入”模块。

## 模块独立程度的两个定性标准度量：

- **耦合**衡量不同模块彼此间互相依赖(连接)的紧密程度。耦合要低，即每个模块和其他模块之间的关系要简单；
- **内聚**衡量一个模块内部各个元素彼此结合的紧密程度。内聚要高，每个模块完成一个相对独立的特定子功能。

# 耦合

- **耦合**：是对一个软件结构内不同模块之间互连程度的度量。
- **要求**：在软件设计中应该追求尽可能松散耦合的系统。
- 可以研究、测试或维护任何一个模块，而不需要对系统的其他模块有很多了解；
- 模块间联系简单，发生在一处的错误传播到整个系统的可能性就很小；
- 模块间的耦合程度强烈影响系统的可理解性、可测试性、可靠性和可维护性。

# 内聚

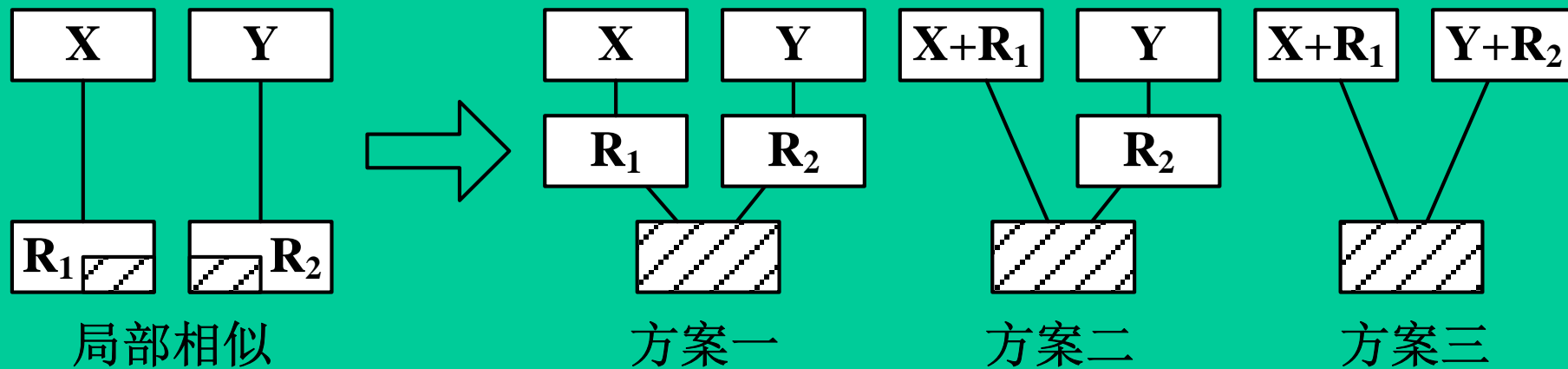
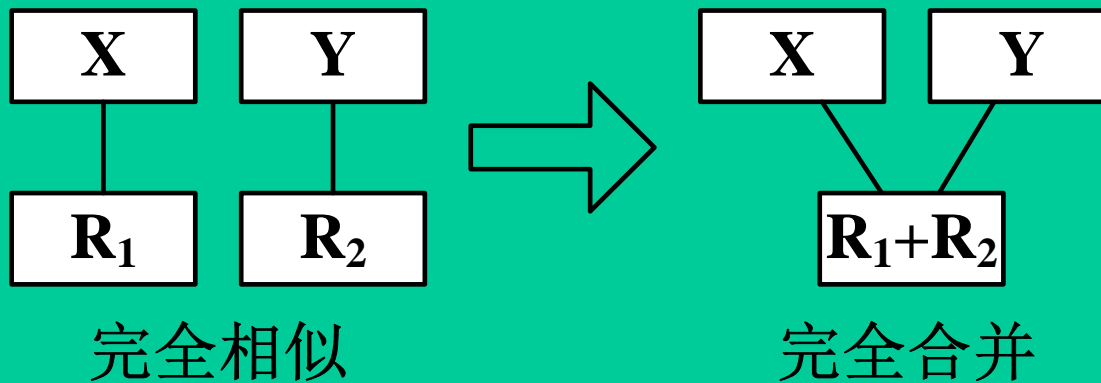
- **内聚：**标志一个模块内各个元素彼此结合的紧密程度，它是信息隐藏和局部化概念的自然扩展。简单地说，理想内聚的模块只做一件事情。
- **要求：**设计时应该力求做到高内聚，通常中等程度的内聚也是可以采用的，而且效果和高内聚相差不多；但是，低内聚不要使用。
- 内聚和耦合是密切相关的，模块内的高内聚往往意味着模块间的松耦合。实践表明内聚更重要，应该把更多注意力集中到提高模块的内聚程度上。

## 模块设计的几个经验

- 模块的作用域应该在控制域之内
- 应该尽量降低模块接口的复杂性
- 模块功能应该可以预测
- 尽量将相同功能的模块提取为公共模块
- 分解模块可以减少控制信息传递
- 合并模块可以减少全局数据的引用

# 1. 改进软件结构提高模块独立性

- 通过模块分解或合并，降低耦合提高内聚。
- 两个方面：
  - 模块功能完善化。一个完整的模块包含：
    - 执行规定的功能的部分
    - 出错处理的部分
    - 返回一个“结束标志”
  - 消除重复功能，改善软件结构。
    - 完全相似
    - 局部相似



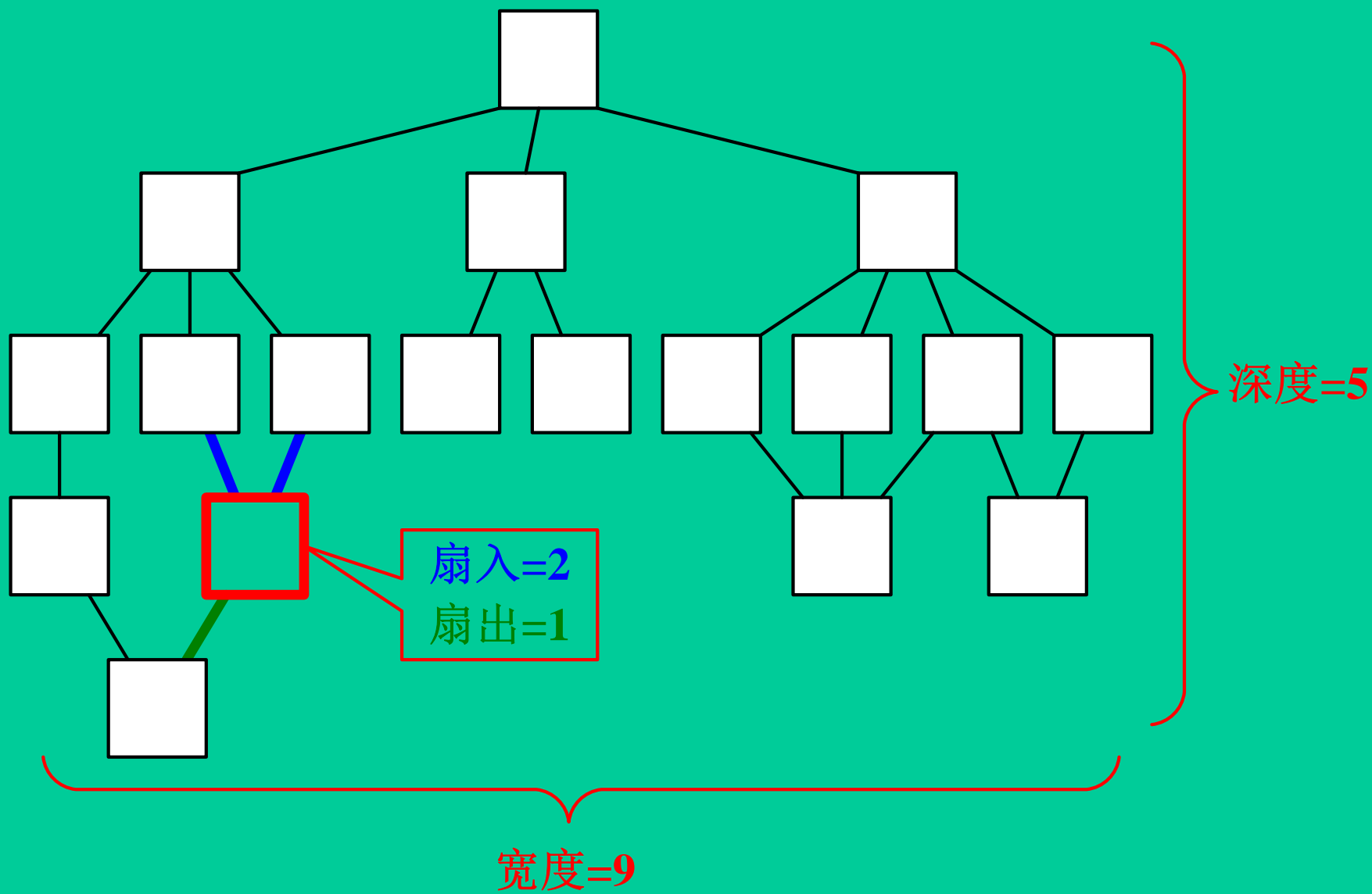


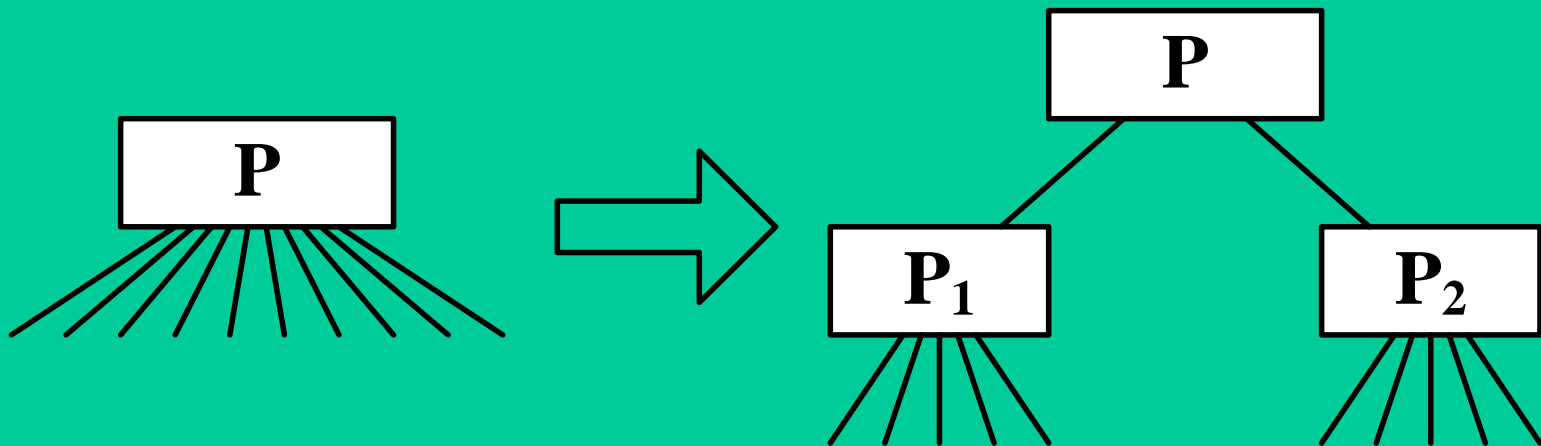
## 2. 模块规模应该适中

- 经验表明，一个模块的规模不应过大，最好能写在一页纸内。通常规定50~100行语句，最多不超过500行。数字只能作为参考，根本问题是要保证模块的独立性。
- 过大的模块往往是由于分解不充分，但是进一步分解必须符合问题结构，一般说来，分解后不应该降低模块独立性。
- 过小的模块开销大于有效操作，而且模块数目过多将使系统接口复杂。

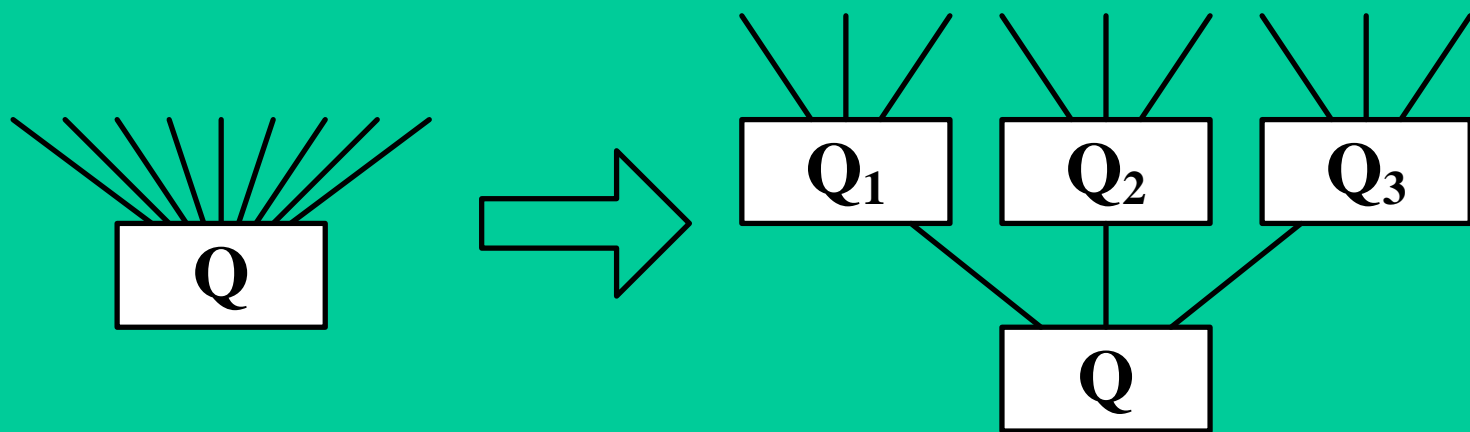
### 3. 深度、宽度、扇出和扇入都应适当

- **深度**：软件结构中控制的层数，它往往能粗略地标志一个系统的大小和复杂程度。
- **宽度**：软件结构内同一个层次上的模块总数的最大值。
- **扇出**：一个模块直接控制(调用)的模块数目。
- **扇入**：有多少个上级模块直接调用它。





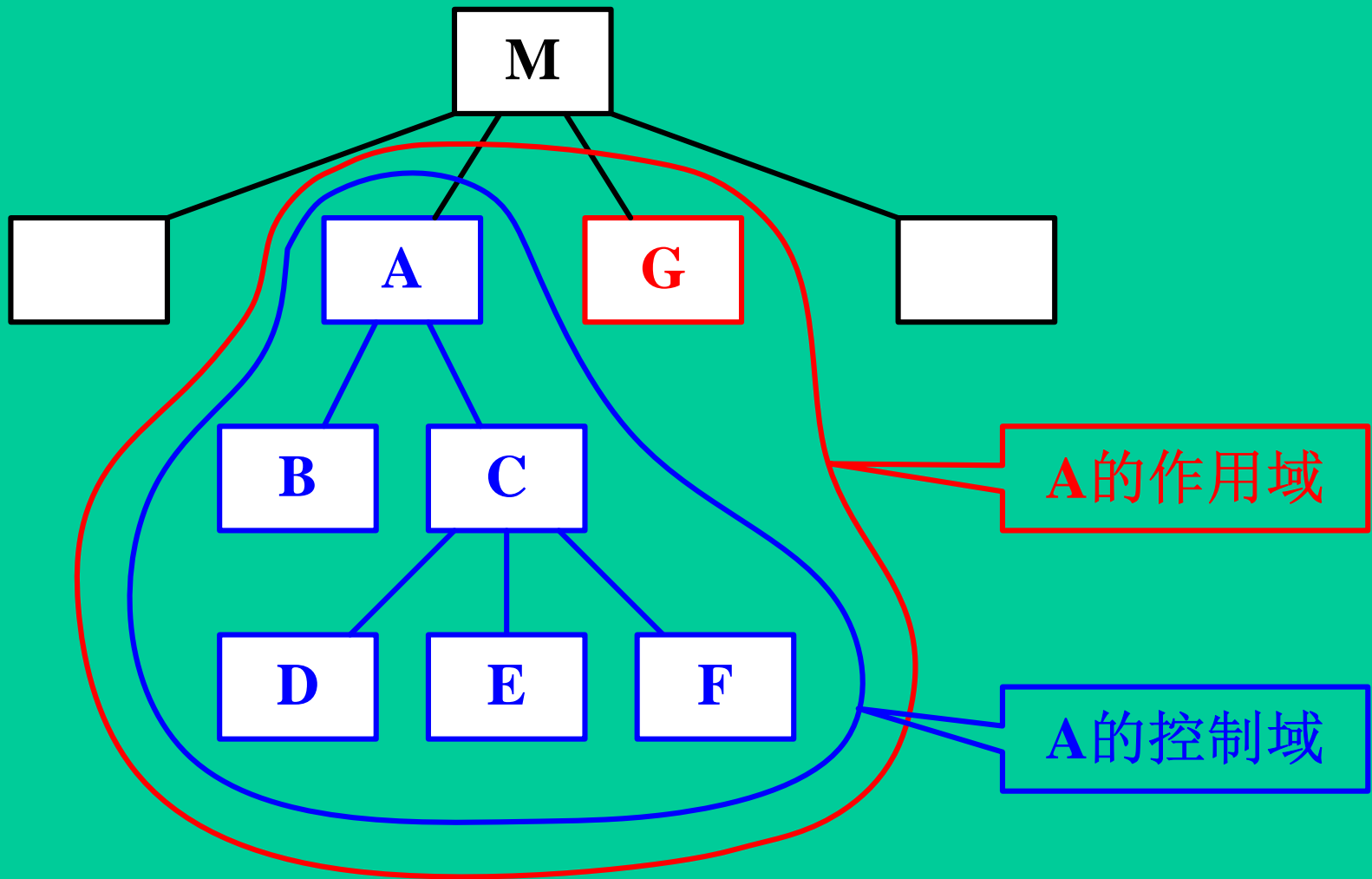
扇出过大



非公共模块扇入过大

## 4. 模块的作用域应该在控制域之内

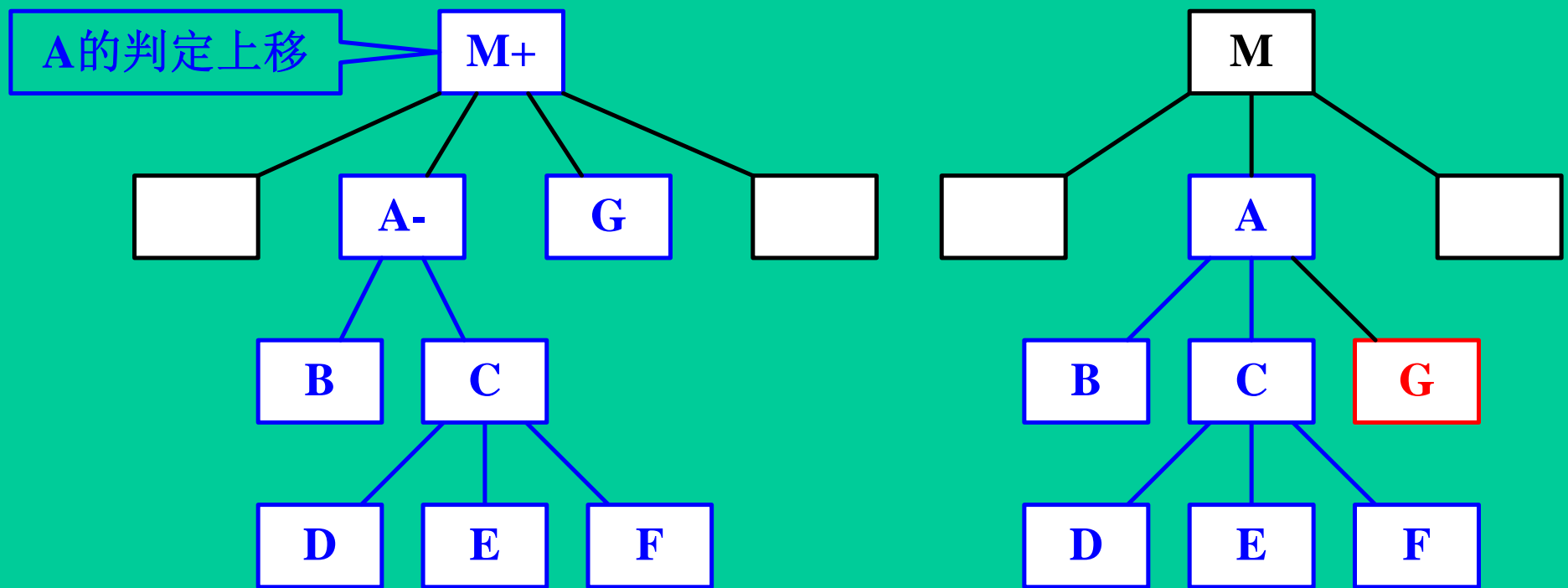
- **模块的作用域**：定义为受该模块内一个判定影响的所有模块的集合。
- **模块的控制域**：是这个模块本身以及所有直接或间接从属于它的模块的集合。
- 在一个设计得很好的系统中，所有受判定影响的模块应该都从属于做出判定的那个模块，最好局限于做出判定的那个模块本身及它的直属下级模块。



违反规则的情况

## 解决方案:

- 把模块A中的判定移到模块M中;
- 把模块G移到模块A下面, 作为他的下级模块。



## 5. 力争降低模块接口的复杂程度

- 模块接口复杂是软件发生错误的一个主要原因。应该仔细设计模块接口，使得信息传递简单并且和模块的功能一致。

例：解一元二次方程的函数

- `QUAD_ROOT(TBL,X)` 
  - 其中数组TBL传送方程的系数
  - 数组X送回求得的根
- `QUAD_ROOT(A,B,C,ROOT1,ROOT2)` 



## 6. 设计单入口单出口的模块

- 警告软件工程师不要使模块间出现内容耦合。当从顶部进入模块并且从底部退出来时，软件是比較容易理解的，因此也是比較容易维护的。

## 7. 模块功能应该可以预测

- 模块的功能应该能够预测，但也要防止模块功能过分局限。
- **功能可预测：**如果一个模块可以当做一个黑盒子，只要输入的数据相同就产生同样的输出，这个模块的功能就是可以预测的。

# 软件设计评审

## 软件设计文档评审

目的：尽早发现设计中的错误、缺陷，经过修改完善，  
形成SCM设计阶段的基线；

评审内容：

- ❖ 软件总体结构；
- ❖ 数据结构；
- ❖ 人机界面；
- ❖ 接口；
- ❖ 出错处理功能；

# 设计模式

# 设计模式概念与组成

- 模式化的过程就是把问题抽象化，在忽略掉不重要的细节后，发现问题的一般性本质，并找到普遍适用的解决方案的过程。
- 原则
  - 开-闭原则；里氏代换原则；依赖倒转原则；接口隔离原则；组合/聚合复用原则；迪米特法则；

# 设计模式概念与组成

- 软件设计模式是软件高效和成熟的程序设计模板，模板包含故有问题的解决方案和逻辑，强调复用程序结构。
- 设计模式是对软件开发经验总结。一个设计模式有系统的命名、解释和评价，是某个重要的可重现的设计方案。

## 设计模式的基本成分

- 1) 模式名称：用来描述一个设计问题，能表达和交流设计思想
- 2) 问题：能反映使用模式的必要性，必要条件，希望的特性
- 3) 解决方案：描述设计要素，要素的组织结构，相互关系
- 4) 后果：描述使用的优势和代价，与相关模式的区别

# 开-闭原则

- 一个软件实体应该对扩展开放，对修改关闭。软件系统中包含的各种组件，例如模块（Modules）、类（Classes）以及功能（Functions）等等，应该在不修改现有代码的基础上，引入新功能。
- 实现开闭原则的关键就在于“**抽象**”。把系统的所有可能的行为抽象成一个抽象底层，这个抽象底层规定出所有的具体实现必须提供的方法的特征。作为系统设计的抽象层，要预见所有可能的扩展，从而使得在任何扩展情况下，系统的抽象底层不需修改；同时，由于可以从抽象底层导出一个或多个新的具体实现，可以改变系统的行为，因此系统设计对扩展是开放的。（Java抽象类 接口）

# 开-闭原则

- （1）对于扩展是开放的（**Open for extension**）。这意味着模块的行为是可以扩展的。当应用的需求改变时，我们可以对模块进行扩展，使其具有满足那些改变的新行为。也就是说，我们可以改变模块的功能。
- （2）对于修改是关闭的（**Closed for modification**）。对模块行为进行扩展时，不必改动模块的源代码或者二进制代码。模块的二进制可执行版本，无论是可链接的库、**DLL**或者**.EXE**文件，都无需改动。

# 里氏代换原则

- 里氏代换原则，任何基类可以出现的地方，子类一定可以出现。LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。
- 所有引用基类的地方必须能透明地使用其子类的对象。



# 里氏代换原则

- 问题由来：有一功能P1，由类A完成。现需要将功能P1进行扩展，扩展后的功能为P，其中P由原有功能P1与新功能P2组成。新功能P由类A的子类B来完成，则子类B在完成新功能P2的同时，有可能会導致原有功能P1发生故障。
- 解决方案：当使用继承时，遵循里氏替换原则。类B继承类A时，除添加新的方法完成新增功能P2外，尽量不要重写父类A的方法，也尽量不要重载父类A的方法。
- 继承包含这样一层含义：父类中凡是已经实现好的方法（相对于抽象方法而言），实际上是在设定一系列的规范和契约，虽然它不强制要求所有的子类必须遵从这些契约，但是如果子类对这些非抽象方法任意修改，就会对整个继承体系造成破坏。而里氏替换原则就是表达了这一层含义。

# 依赖倒转原则（DIP）

- 要依赖于抽象，不要依赖于具体。要针对接口编程，不要针对实现编程。
- 依赖倒置原则（**Dependence Inversion Principle**）是程序要依赖于抽象接口，不要依赖于具体实现。简单的说就是要求对抽象进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合。
- 传统设计中，上层调用下层，上层依赖于下层（高层模块依赖于低层，即抽象依赖于具体），当下层剧烈变动时上层也要跟着变动，这就会导致模块的复用性降低而且大大提高了开发的成本。

# 接口隔离原则（ISP）

- 使用多个专用的接口比使用单一的总接口要好。
- 客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。每一个接口应该承担一种相对独立的角色，不干不该干的事，该干的事都要干。
- 在面向对象编程语言中，实现一个接口就需要实现该接口中定义的所有方法，因此大的总接口使用起来不一定很方便，为了使接口的职责单一，需要将大接口中的方法根据其职责不同分别放在不同的小接口中，以确保每个接口使用起来都较为方便，并都承担某一单一角色。接口应该尽量细化，同时接口中的方法应该尽量少，每个接口中只包含一个客户端（如子模块或业务逻辑类）所需的方法即可，这种机制也称为“定制服务”，即为不同的客户端提供宽窄不同的接口。

# 组合/聚合复用原则（CARP）

- 组合/聚合复用原则（**Composite/Aggregate Reuse Principle CARP**），组合和聚合都是对象建模中关联（**Association**）关系的一种，聚合表示整体与部分的关系，表示“含有”，整体由部分组合而成，部分可以脱离整体作为一个独立的个体存在。组合则是一种更强的聚合，部分组成整体，而且不可分割，部分不能脱离整体而单独存在。
- 合成复用原则(**Composite Reuse Principle, CRP**): 尽量使用对象组合，而不是继承来达到复用的目的。

# 组合/聚合复用原则（CARP）

- 在面向对象设计中，可以通过两种方法在不同的环境中复用已有的设计和实现，即通过组合/聚合关系或通过继承，但首先应该考虑使用组合/聚合，组合/聚合可以使系统更加灵活，降低类与类之间的耦合度，一个类的变化对其他类造成的影响相对较少；其次才考虑继承，在使用继承时，需要严格遵循里氏代换原则，有效使用继承会有助于对问题的理解，降低复杂度，而滥用继承反而会增加系统构建和维护的难度以及系统的复杂度，因此需要慎重使用继承复用。

# 组合/聚合复用原则（CARP）

- 通过继承来进行复用的主要问题在于继承复用会破坏系统的封装性，因为继承会将基类的实现细节暴露给子类，由于基类的内部细节通常对子类来说是可见的，所以这种复用又称“白箱”复用，如果基类发生改变，那么子类的实现也不得不发生改变；从基类继承而来的实现是静态的，不可能在运行时发生改变，没有足够的灵活性；而且继承只能在有限的环境中使用（如类没有声明为不能被继承）。
- 由于组合或聚合关系可以将已有的对象（也可称为成员对象）纳入到新对象中，使之成为新对象的一部分，因此新对象可以调用已有对象的功能，这样做可以使得成员对象的内部实现细节对于新对象不可见，所以这种复用又称为“黑箱”复用，相对继承关系而言，其耦合度相对较低，成员对象的变化对新对象的影响不大，可以在新对象中根据实际需要有针对性地调用成员对象的操作；合成复用可以在运行时动态进行，新对象可以动态地引用与成员对象类型相同的其他对象。



# 迪米特法则（LoD）

- 也称为最少知识原则（Least Knowledge Principle, LKP）。
- 一个对象应该对其他对象有最少的了解。通俗地讲，一个类应该对自己需要耦合或调用的类知道得最少，你（被耦合或调用的类）的内部是如何复杂都和我没关系，那是你的事情，我就知道你提供的public方法，我就调用这么多，其他的一概不关心。
- 如果一个系统符合迪米特法则，那么当其中某一个模块发生修改时，就会尽量少地影响其他模块，扩展会相对容易，这是对软件实体之间通信的限制，迪米特法则要求限制软件实体之间通信的宽度和深度。迪米特法则可降低系统的耦合度，使类与类之间保持松散的耦合关系。

# 迪米特法则（LoD）

- 在将迪米特法则运用到系统设计中时，要注意下面的几点：在类的划分上，应当尽量创建松耦合的类，类之间的耦合度越低，就越有利于复用，一个处在松耦合中的类一旦被修改，不会对关联的类造成太大波及；在类的结构设计上，每一个类都应当尽量降低其成员变量和成员函数的访问权限；在类的设计上，只要有可能，一个类型应当设计成不变类；在对其他类的引用上，一个对象对其他对象的引用应当降到最低。



# 设计模式的类型

## 1. 构造式：对象创建的过程处理

设计模式	简要说明
抽象工厂 (Abstract Factory)	提供创建相关的一组对象的接口，不需要指定它们的具体实现。
构造器 (Builder)	将对象的结构与它的表示分离，可用同样的结构得到不同的表示。
工厂方法 (Factory Method)	定义创建对象的接口，但由子类来决定实例化。
原型 (Prototype)	用原形实例指定创建对象的种类，并通过拷贝原形来创建新的对象。
单例 (Singleton)	保证一个类仅有一个实例，提供访问它的全局访问点。

# 设计模式的类型

## 2. 结构式：处理对象/类的组合

设计模式	简要说明
适配器(Adapter)	把类的接口转换成另外一个接口，解决接口不兼容问题。
桥接(Bridge)	将对象的抽象和实现部分进行分离，使它们能独立变化。
合成(Composite)	把对象组织成树形结构来表示层次关系，使对单个和复合对象的使用具有一致性。
装饰器(Decorator)	动态地为对象添加新的操作功能。
外观(Facade)	把子系统各个接口统一为一致的接口。
享元(Flyweight)	运用共享技术支持细粒度的对象集合。
代理(Proxy) 2017/09/18	以对象的名义对另一个对象进行访问。

### 3. 行为式：处理对象间的交互方式和任务分布

设计模式	简要说明
责任链 (Chain of Responsibility)	将不同对象对请求的处理形成一个链，使它们都有机会检查请求，从而解除请求和处理者之间的耦合（类似异常的抛出）。
命令 (Command Processor)	把请求封装为一个对象，可用不同的请求对客户进行参数化控制。
解释器 (Interpreter)	在给定语言和文法的情况下，建立一个该语言的解释器。
迭代器 (Iterator)	在不暴露对象内部结构的情况下，循环访问一个对象的集合。
协调器 (Mediator)	使用一个中介，在对象不需要显式引用的情况下维持交互关系，从而达到维持对象松散耦合的作用。
备忘录 (Memento)	在不破坏封装的前提下，获得对象内部状态的访问和控制权。
观察者 (Observer)	建立对象与依赖它的对象间的变化-通知机制（MVC模式）。
状态 (state)	使对象在其内部状态发生变化时发生类似类行为的改变。
策略 (Strategy/Policy)	定义可以替换的算法，使得算法的变化可以独立于用户。
模板方法 (Template Method)	定义某个算法的计算过程框架，具体计算步骤到子类中加以实现，在不改变算法构架的同时改变某些计算步骤的实现。
2017/09/18 中介者 (Mediator)	用中介对象来封装一系列交互对象，它使对象之间不需要显式地相互引用，构成松散耦合，可独立地改变它们之间的交互。 307

# Singleton模式

**意义：** 保证一个类仅有一个实例，并提供一个访问它的全局访问点。

**协作：** 应用只能通过Instance操作访问唯一的Singleton对象实例。

**效果：**

- 控制访问唯一实例。因为Singleton 类封装它的唯一实例，所以它可以严格的控制应用的访问。
- Singleton 模式使用静态数据成员和成员函数，得到全局变量应用，可保证唯一存储，避免混乱的对象空间。
- 可以在需要时，用相同的方法改变Singleton类的实例数量。

# Singleton模式

- 查看 Singleton设计模式用例.cpp

Singleton	
+void* operator new(unsigned int)	•
+static GetInstance()	
-singleton static *s	

Return  
uniqueInstance

# Singleton模式

- 适用场景:

- 1.某个软件组件需要访问数据库。为此，可能希望整个组件只创建一个数据库连接；
- 2.整个系统的主控类，一般只能创建一个。

# Factory模式

- 抽象工厂和生成器模式都定义一个创建系列具体产品的类。
- 抽象工厂模式，用这个类来负责产生多个不同产品对象，但它们具有相同部件；
- 生成器模式，用这个类构造不同部件成分的系列产品。

# Factory模式

- 查看 Factory设计模式1.cpp

Real
+Real(int a = 0, b = 0) +Print()
-int X, Y

```
void main()
{
    Real R(2, 5);
    R.Print();
}
```

分析:

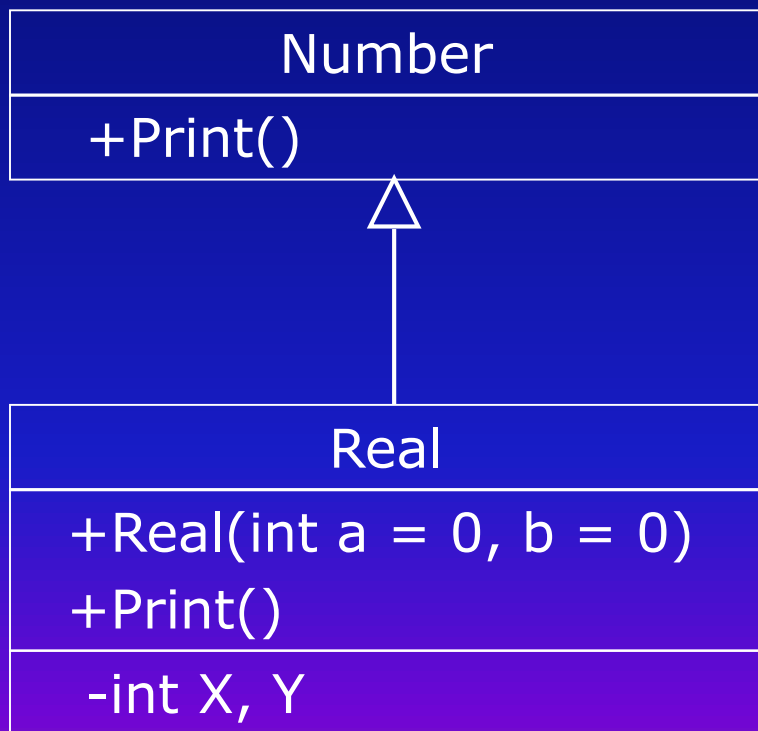
**main**函数直接依赖于类**Real**，如果改为显示其他数据类型，则必须修改两句代码。



# Factory模式

## ■ 改进一：查看 Factory设计模式2.cpp

- 使用抽象类



- 更改main函数代码

```
void main()
{
    Number *N=new Real(2, 5);
    N->Print();
}
```

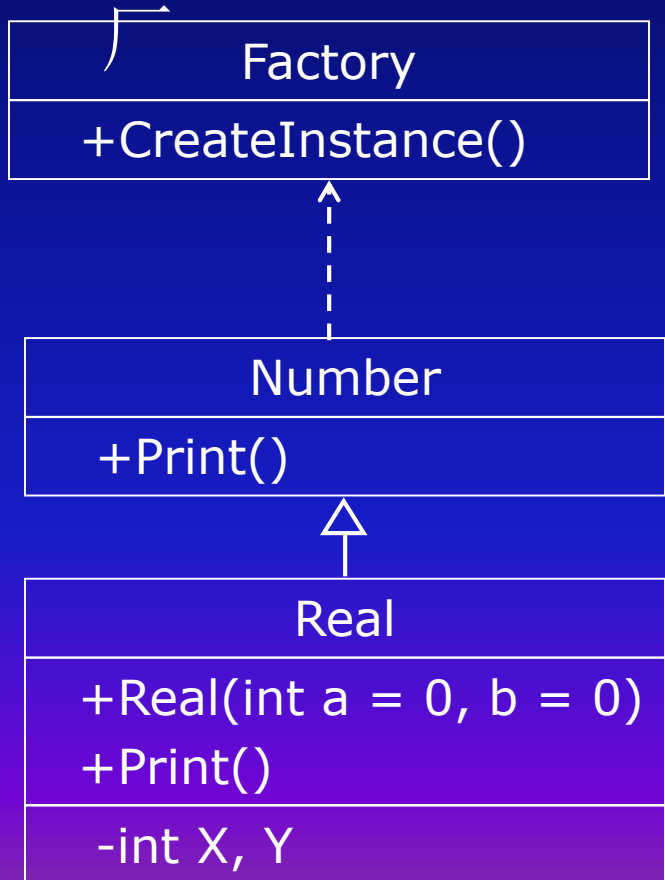
分析：

现在只有一句用到了类**Real**，耦合度降低了！但仍然与**Real**有关。

# Factory模式

## ■ 改进二：查看 Factory设计模式3.cpp

- 定义数据类型工



```
void main()
{
    Factory F;
    Number * N = F.CreateInstance();
    N->Print();
}
```

分析：

抽象化程度提高，完全与数据类型 **Real** 无关。

# 中介者模式

**定义** 用一个中介对象来封装一系列的对象交互。

**目的** 设计模式中的多对多模型。

**协作** 同事向中介者对象发送和接收请求。中介者在各个同事之间转发请求完成协作行为。

**效果**

- 将分布于多个对象间的行为形成统一的行为；
- 各同事对象实现解耦；
- 使多对多协议简化为一对多协议，关系易于理解和维护；
- 交互的复杂性集中于中介者，使中介者难于维护。

# 中介者模式

- 一个供应方可以有多种产品
- 要进行以货易货必须有两个供应方
- 具体的供应商可能有多个

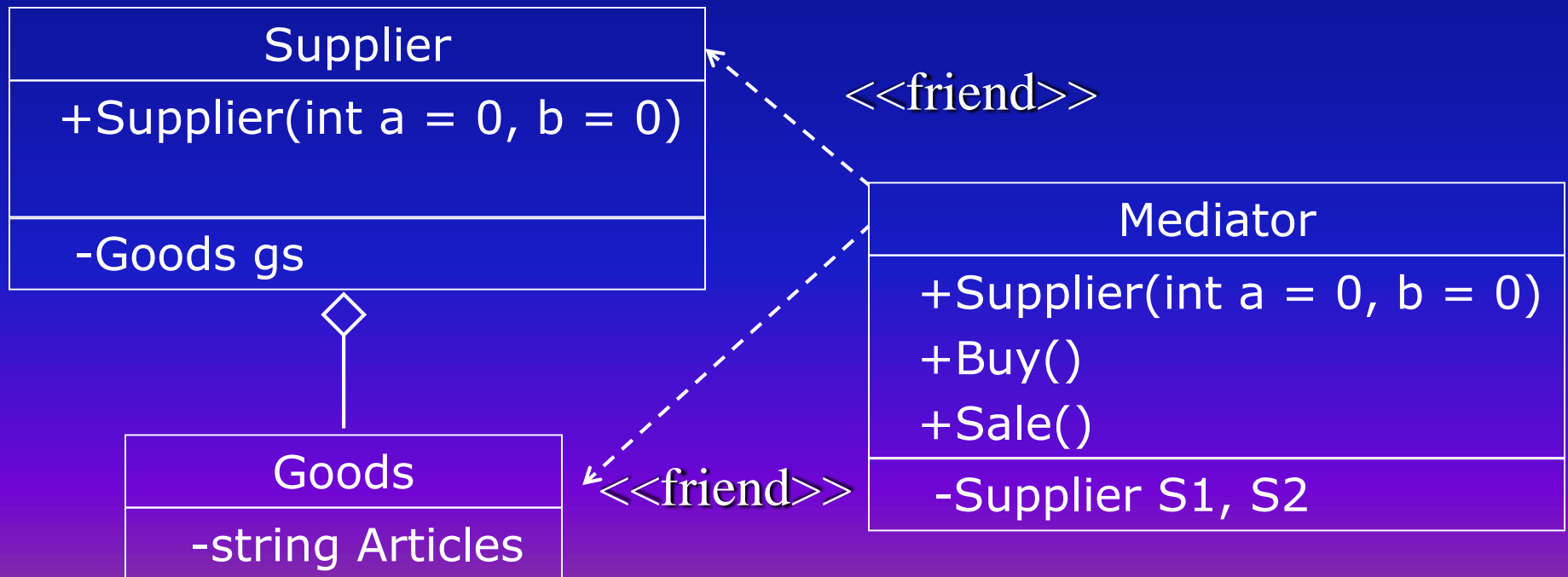
以货易货这一基本流程所涉及的双方是 多对多的关系

- 系统一定会扩充：
  1. 只买，只卖，预订，期货交易等等
  2. 每个交易一般都需要满足一定的条件：比如以货易货要求价值相等

系统必须支持功能不断地扩充。

# 中介者模式

- 创建一个供应商基类：
  - 使用Goods这一数据结构来表达以货易货交易信息
- 每个供应商就是这个类的一个实例。
  - 利用中介者集中处理交易



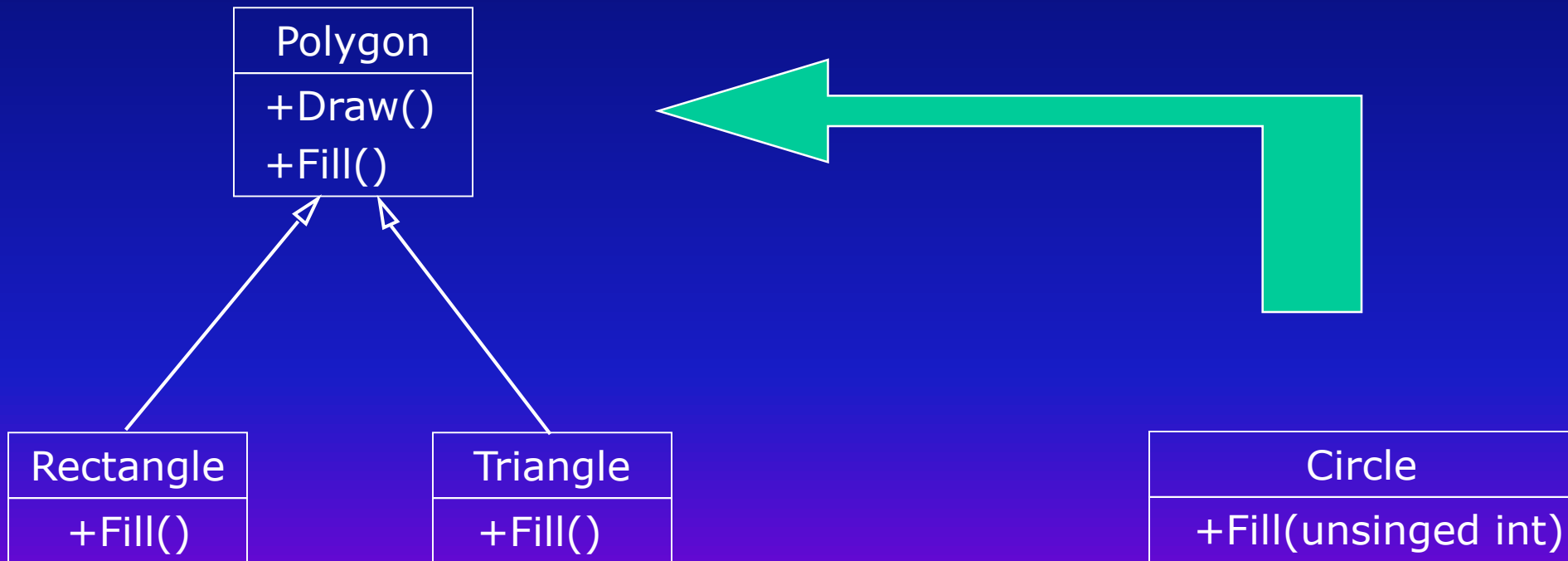
# 中介者模式

## 为何要用中介者模式？

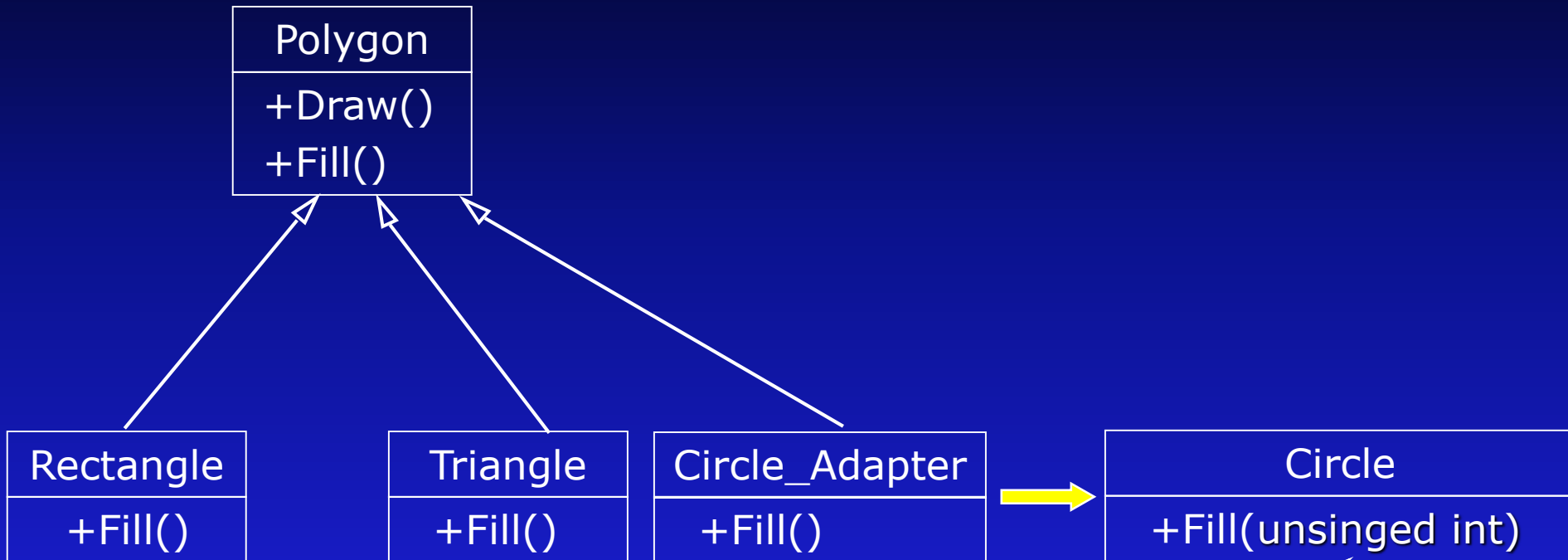
- 1. 可以很容易地**扩充系统功能**：加入只买，只卖等功能，要做的就是给Mediator类增加一个公有方法
- 2. 可以在公有方法中**封装业务逻辑**，比如以货易货要求价值相等
- 3. 现在供应商对象之间不再需要了解到底各自有哪些商品，于是，就将供应商对象之前的**多对多**关系变成了两个供应商与经纪人的两个**一对多**关系。这就简化了系统。
- 4. 主要用于简化以下类型的系统：  
有多个对象以及这多个对象之间有着复杂的联系

# Adapter模式

适配器（Adapter）模式是用于将类的一个接口转换为另一个类所需接口，以使由于接口不兼容的类能够一起协作。



# Adapter模式



问题：增加适配器后，原Cirlcle的参数如何传递给Circle?