



## 4. 2 结构化设计

### 1. 何谓软件设计

一种软件开发活动, 定义实现需求规约所需的软件结构。

**设计目标：**依据需求规约，在一个抽象层上建立系统软件模型，包括软件体系结构（数据和程序结构），以及详细的处理算法，产生设计规格说明书。

**即：**要回答如何解决问题 - 给出软件解决方案

结构化设计分为：

(1) 总体设计：确定系统的整体模块结构（即软件体系结构），即系统实现所需要的软件模块以及这些模块之间的调用关系。

(2) 详细设计：详细描述模块。



北京大学



## 2. 软件设计的原则

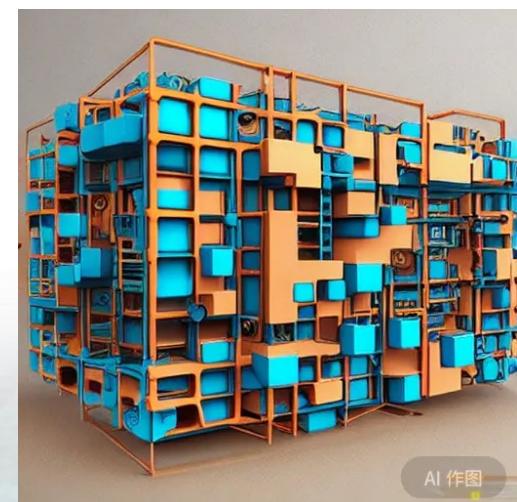
为提高软件开发的效率及软件产品的质量，人们在长期的软件开发实践中总结出一些软件设计的概念和原则。

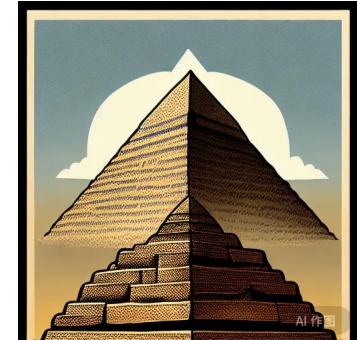
### (1) 关注点分离：

- 将复杂问题**分解为**可以独立解决或优化的若干**子问题**，以便更容易地解决。
- 举例：商业软件中通常根据功能和职责被划分为不同层次，如表示层、业务层和数据层等
- 关注点分离的思想也存在于模块化、切面、逐步求精等设计原则中。

### (2) 模块化：

- 关注点分离的重要表现。
- 将系统或程序划分为可以独立访问的**模块**（过程、函数、类、包等）。模块集成起来可以构成一个整体，满足整体的需求和功能。
- 模块化时需注意模块的**功能独立**和**信息隐蔽**。





## 2. 软件设计的原则

### (3) 抽象：

- 关注与某一特定目的相关的信息而忽略其余的信息，从事物中获取其本质属性。抽象可分为**过程抽象**和**数据抽象**。
- 举例：在购买商品中，将购买行为抽象为一系列过程（提交订单、金额结算、支付扣款等）；将商品抽象为一系列的属性（商品名称、价格、生产批次等）。

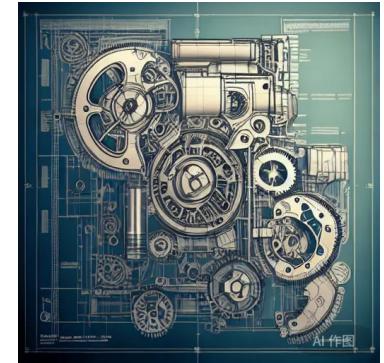
### (4) 逐步求精：

- 自顶向下的设计策略，连续细化、分解高抽象层次的宏观描述，最终到达程序设计语言级别的低抽象层次。
- 有助于在细化设计的过程中**揭示底层细节**。

**抽象和逐步求精抽象互为逆过程，两者互补有助于形成完整的设计模型**



北京大学



## 2. 软件设计的原则

### (5) 设计模式

- 设计模式是一套可复用、为人知晓、经过分类编目的**代码设计经验的总结**，帮助软件开发人员解决软件开发过程中面临的一般问题。
- 提高代码的可读性、可维护性和可扩展性。
- 常见的设计模式包括工厂模式、观察者模式、装饰者模式等。

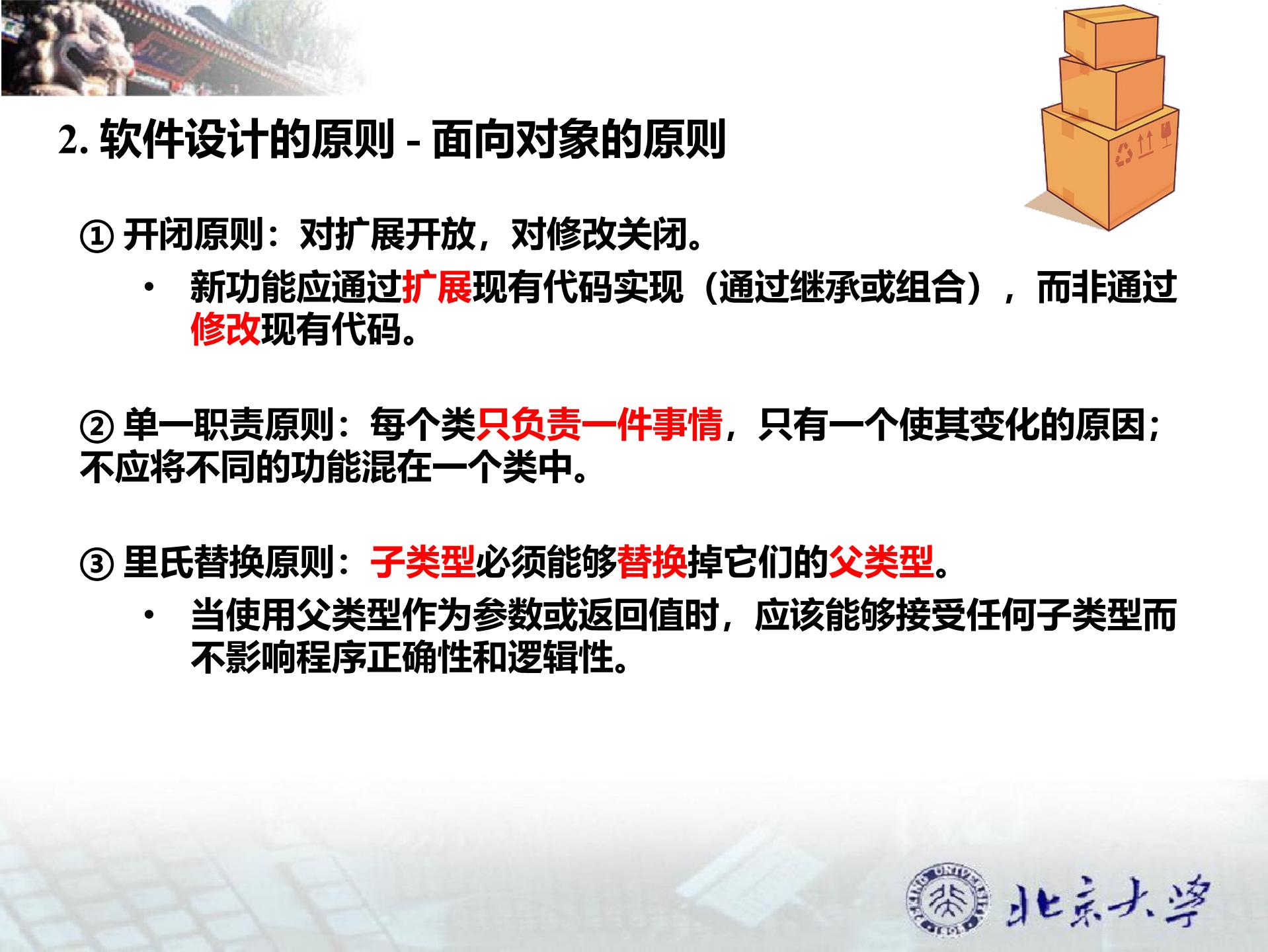
(详细内容将在后续课程中详细介绍)

### (6) 面向对象的原则

- 在面向对象设计时应该遵循的一些**指导性的原则**，以提高代码的可读性、可维护性、可扩展性和可复用性。
- 包括：开闭原则、单一职责原则、里氏替换原则、接口隔离原则、依赖倒置原则、迪米特法则、组合/聚合复用原则。



北京大学



## 2. 软件设计的原则 - 面向对象的原则

① 开闭原则：对扩展开放，对修改关闭。

- 新功能应通过**扩展**现有代码实现（通过继承或组合），而非通过**修改**现有代码。

② 单一职责原则：每个类**只负责一件事情**，只有一个使其变化的原因；不应将不同的功能混在一个类中。

③ 里氏替换原则：**子类型**必须能够**替换**掉它们的**父类型**。

- 当使用父类型作为参数或返回值时，应该能够接受任何子类型而不影响程序正确性和逻辑性。

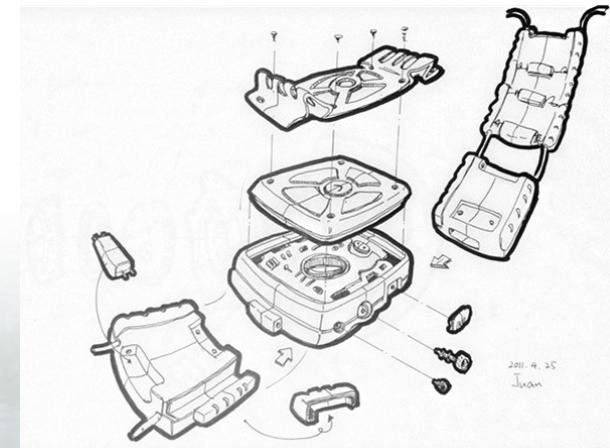


北京大学



## 2. 软件设计的原则 - 面向对象的原则

- ④ 接口隔离原则：要求客户端不应该被强迫依赖于它们不使用的接口。
  - 应当将臃肿的**接口拆分**成多个更小更专业的接口，以满足不同的客户端需求。
- ⑤ 依赖倒置原则：上层模块不应依赖于下层模块，而应该**依赖于它们的抽象**。
- ⑥ 迪米特法则：要求一个对象应当对其它对象有**尽可能少的了解**，以减少对象之间的交互和耦合，使对象只和与其直接相关的对象通信。
- ⑦ 组合/聚合复用原则：新功能优先使用**组合/聚合**关系而不是继承关系实现。

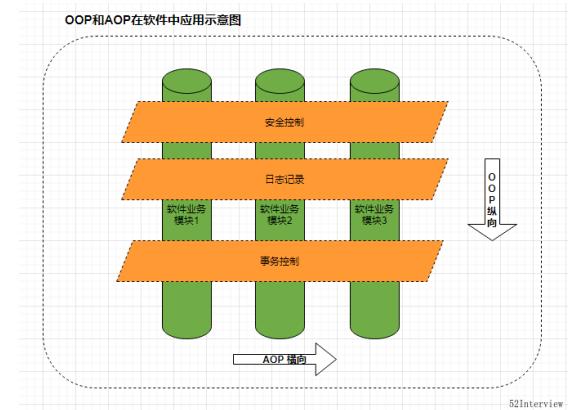




## 2. 软件设计的原则

### (7) 面向切面

- **面向切面是一种编程思想，它提供了一种在不修改原有业务逻辑的情况下对程序中的横切关注点（如日志、安全、事务等）进行统一管理和处理的方式。**
- **切面：对横切关注点的抽象，由通知（要做的动作）和切入点（在哪些地方做动作）组成。**
- 举例：为某类中的“添加用户”方法添加功能：调用该方法前后打印日志，并在执行时开启事务。
  - Bad：直接在方法中添加相关代码  $\Rightarrow$  增加代码的耦合度，且无法复用；
  - Good：定义日志切面类和事务切面类，类中包含通知和事务的逻辑，并指定切入点为“添加用户”方法。 $\Rightarrow$  统一管理关注点，只要配置切入点就可以复用到其他类和方法上。



北京大学



图1 设计阶段和设计内容



北京大学

## 4. 2. 1 结构化总体设计

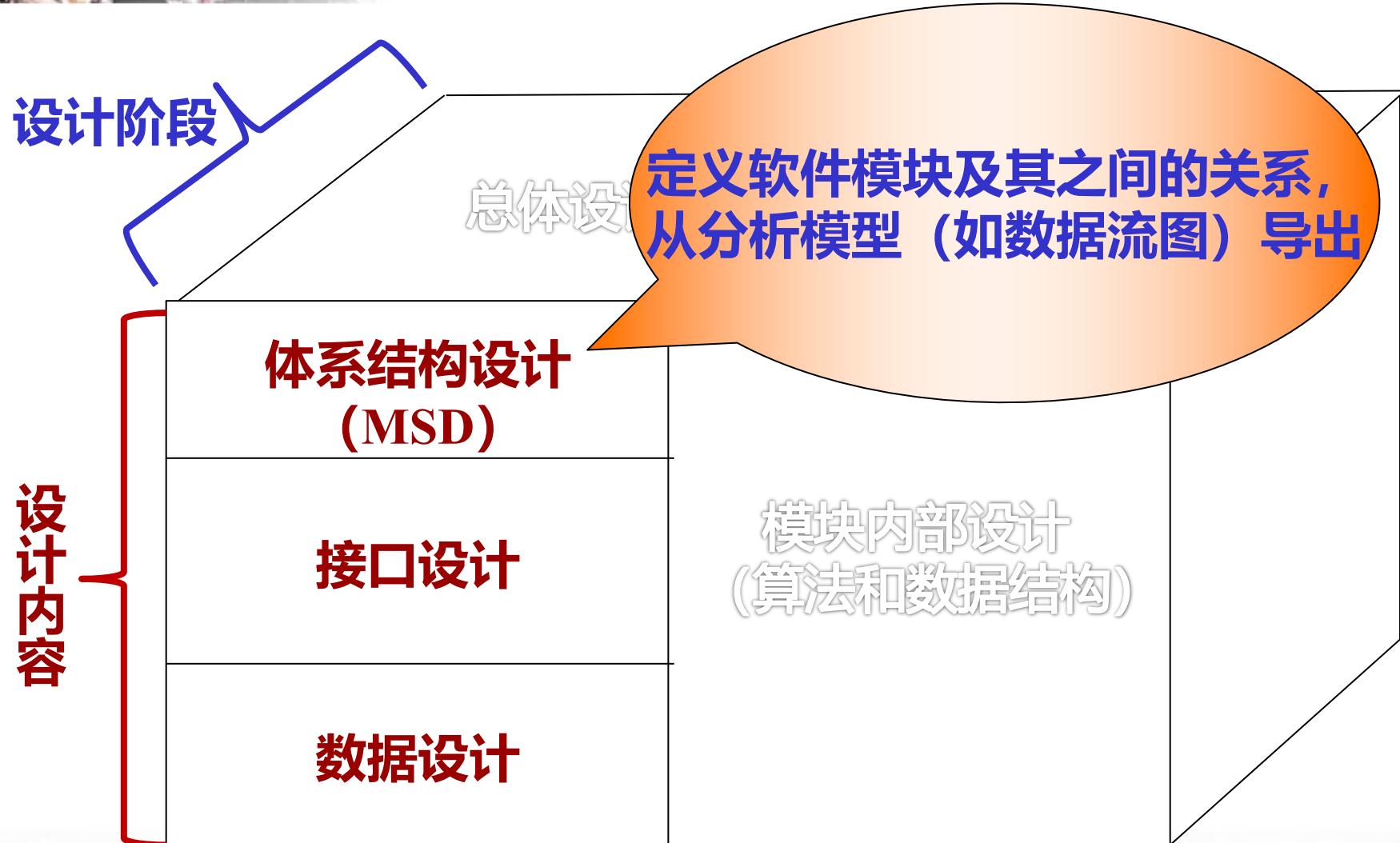


图1 设计阶段和设计内容



北京大学



## 4. 2. 1 结构化总体设计

### 1. 软件体系结构

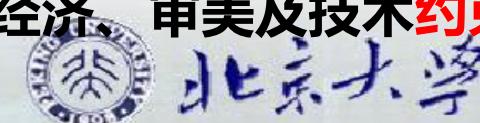
#### (1) 定义：

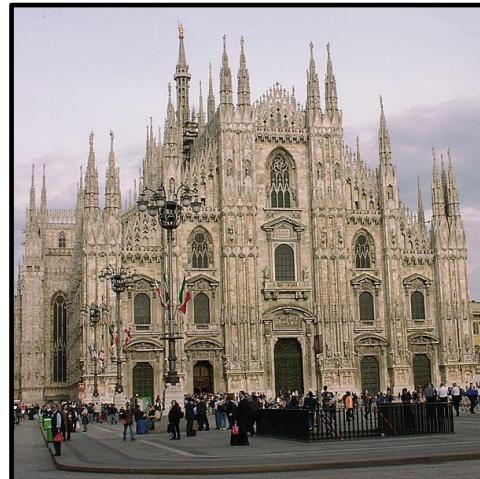
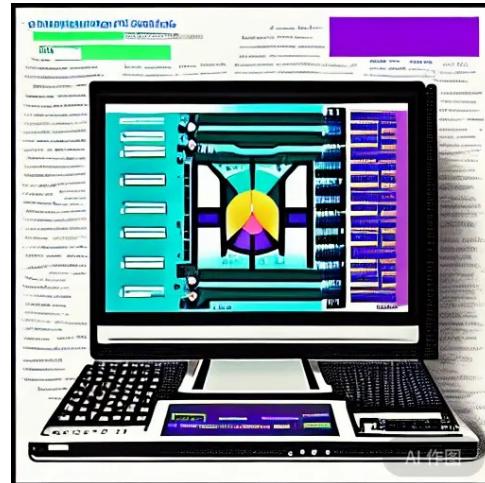
软件体系结构也称为软件架构，是一个**软件系统的高层设计结构**，定义了组成软件系统的**软件构件、构件的外部可见属性、以及构件之间的关系**。

- **软件构件**：计算或数据存储的单元（如客户、服务器、数据库、过滤器等）可封装为模块或类；
- **构件的外部可见属性**：其他构件对该构件所做的假定（提供的功能、性能、错误处理、共享资源的使用等）；
- **构件之间的关系**：构件之间的静态结构的依赖关系，也可以是动态行为的交互关系等。

#### (2) 软件体系结构体现了对如下设计决策的综合：

- 构成系统的**结构化元素**和它们**接口**的选择；
- 组织软件构件以及软件构件之间的交互关系的**体系结构风格**；
- 系统的使用、功能、性能、弹性、复用、可理解性、经济、审美及技术**约束**等





## 软件系统

### 软件体系结构

### 软件体系结构风格

该软件体系基于使用、功能、性能、弹性、复用、可理解性、经济和技术约束与折衷、审美考虑而做出的设计决策

## 建筑物

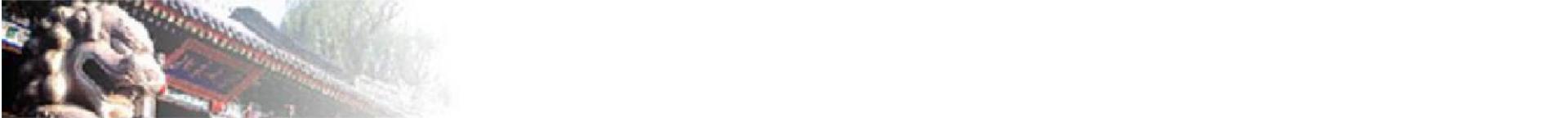
### 建筑的总体设计图

### 建筑风格（如拜占庭风格、中国宫殿式风格等）

该建筑物考虑使用用途、经济性、抗震性能、美观等因素而做出的组成部分和组成部分之间关系的选择



北京大学



### (3) 软件体系结构风格

**软件体系结构风格：**组成软件系统的**构件类型**、**构件之间关系类型**的一个描述，以及它们如何被组织使用的**约束**。

常用的软件体系结构风格包括：

- ① 管道/过滤器
- ② 层次体系结构
- ③ 仓库体系结构
- ④ 客户/服务器体系结构
- ⑤ 消息总线体系结构
- ⑥ 面向服务的体系结构



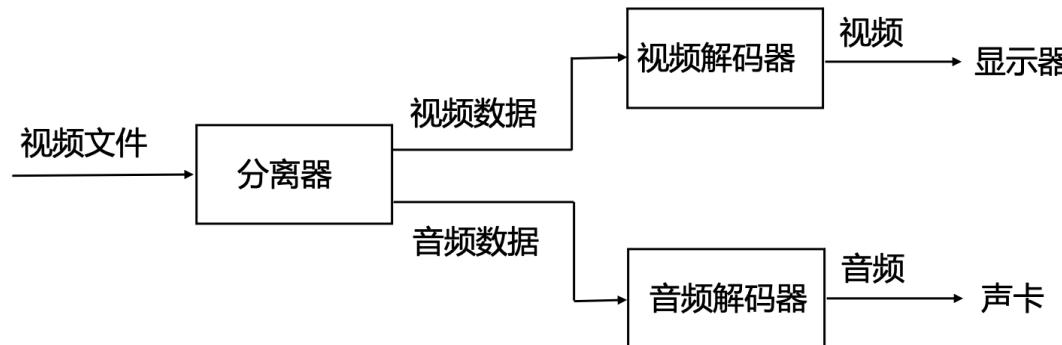
北京大学



# 软件体系结构风格的六种常见类型：

## ① 管道/过滤器风格体系结构

- 特点：由一组被称为过滤器 (filter) 的构件和一组被称为管道 (pipe) 的数据流组成，每个构件读取一组输入数据流，经过内部变换，产生一组输出数据流。
- 举例：媒体播放器



### 优点：

- 良好的封装性，高内聚低耦合；
- 方便合成、复用、系统的维护和演化；
- 支持任务的并行，允许对吞吐量、死锁等属性做分析。

### 不足：

- 性能低，编写困难：每个过滤器需要解析和合成数据；
- 不适合交互式系统的设计。



北京大学



## 软件体系结构风格的六种常见类型：

### ② 层次体系结构风格

- 将软件系统组织成一个层次结构，每层由抽象级别相同的若干构件组成，使用下层构件提供的功能，并为上层构件提供服务。内部层只对相邻层可见；
- 最外层与用户交互；最底层完成最基础公共的功能。
- 优点：支持系统增量开发，有助于系统的维护和演化；
- 不足：有些软件很难分解为层次结构，且层次间传递数据会带来性能问题。



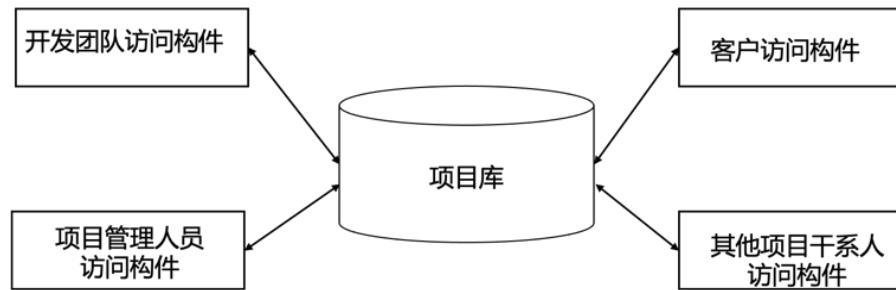
北京大学



## 软件体系结构风格的六种常见类型：

### ③ 仓库体系结构风格

- 将数据存储构件（文件、数据库、知识库等）作为核心，与其他构件以类似于星型结构的拓扑结构连接。分为传统仓库风格和黑板风格两种。
  - 传统仓库风格：数据由一个中心构件产生，由其他构件使用；
  - 黑板风格：“黑板”作为中心构件，负责协调信息在构件间传递。
- 举例：信息管理系统、指挥控制系统和交互式开发环境

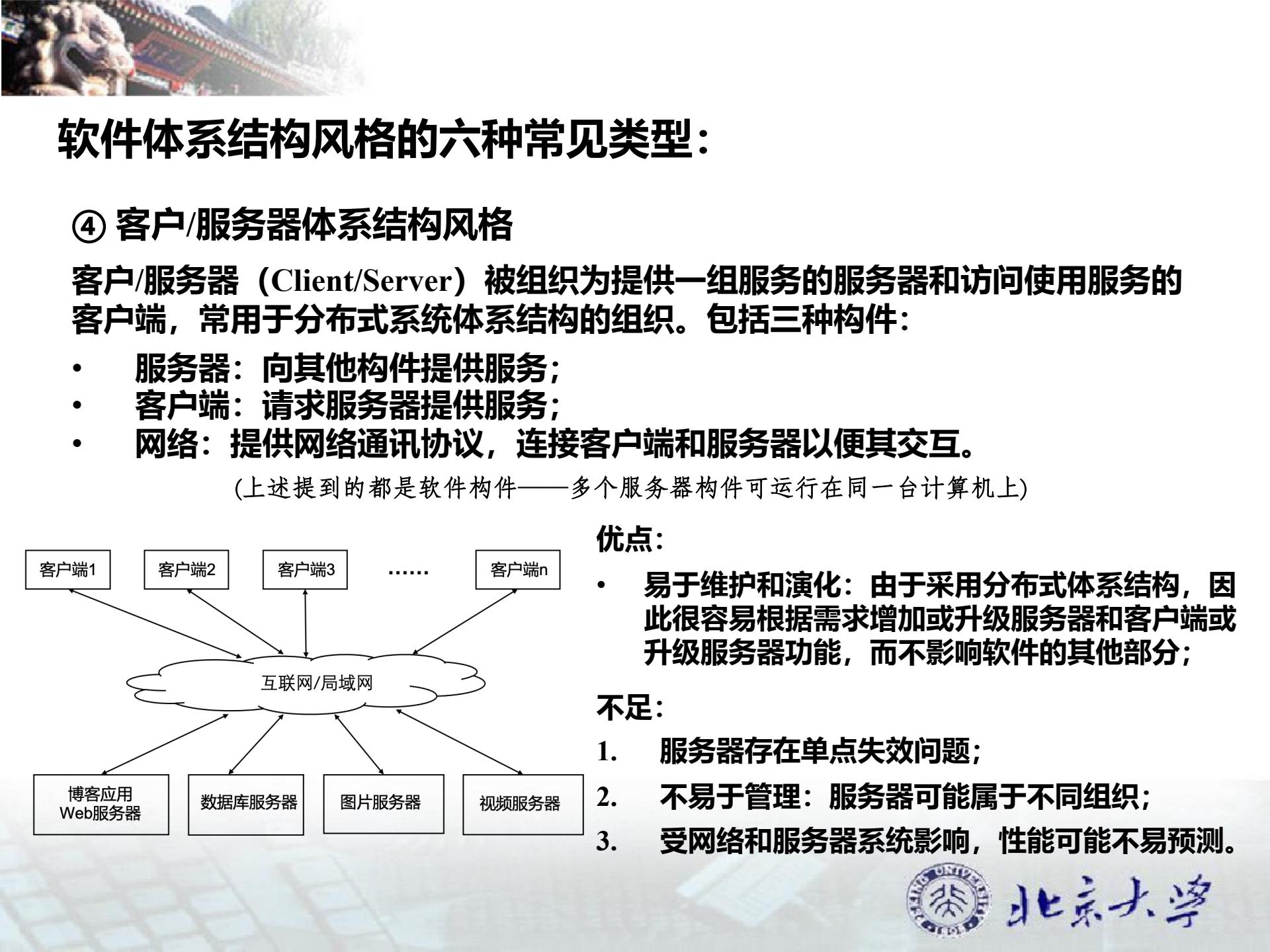


优点：对数据的修改可以传递到所有的构件中。

不足：中心构件的失效将影响整个系统。



北京大学



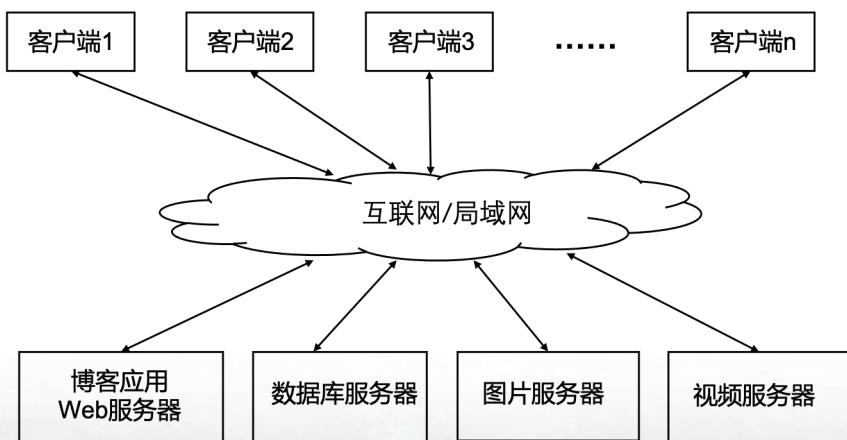
## 软件体系结构风格的六种常见类型：

### ④ 客户/服务器体系结构风格

**客户/服务器** (Client/Server) 被组织为提供一组服务的服务器和访问使用服务的客户端，常用于分布式系统体系结构的组织。包括三种构件：

- **服务器：向其他构件提供服务；**
- **客户端：请求服务器提供服务；**
- **网络：提供网络通讯协议，连接客户端和服务器以便其交互。**

(上述提到的都是软件构件——多个服务器构件可运行在同一台计算机上)



#### 优点：

- **易于维护和演化：**由于采用分布式体系结构，因此很容易根据需求增加或升级服务器和客户端或升级服务器功能，而不影响软件的其他部分；

#### 不足：

1. **服务器存在单点失效问题；**
2. **不易于管理：**服务器可能属于不同组织；
3. **受网络和服务器系统影响，性能可能不易预测。**



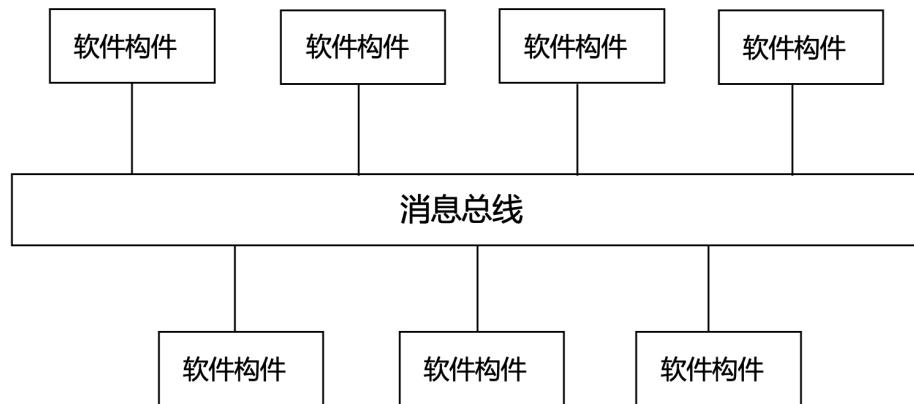
北京大学



## 软件体系结构风格的六种常见类型：

### ⑤ 消息总线体系统结构风格

- 一种分布式系统的体系结构风格。
- 包含一个消息总线构件，连接所有构件，提供规范化通信接口，负责构件之间的消息传递；其他构件通过总线发送和接收信息，实现构件间通信。



#### 优点：

1. 构件相互独立，易于更新维护；
2. 构件之间解耦，方便构件的动态增加删除。

#### 不足：

1. 通信异步，不适合实时软件系统；
2. 总线是中心化的，存在可靠性风险。



北京大学



## 软件体系结构风格的六种常见类型：

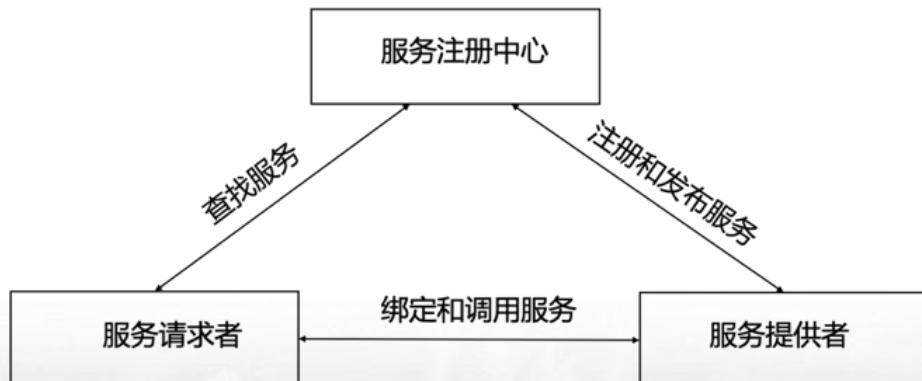
### ⑥ 面向服务的体系结构风格

将软件系统的不同功能单元封装为服务，服务之间通过定义良好的接口连接起来。  
包括三种类型的软件构件：

1. 服务提供者：定义和实现服务，并将服务的描述发布到服务注册中心；
2. 服务请求者：向服务注册中心查找所需服务，与该服务绑定，并调用该服务
3. 服务注册中心。

特点：

1. 模块化：每个服务封装了独立的功能；
2. 高内聚和低耦合，服务请求者只可见服务提供者的服务接口，而不可见其位置、实现技术、当前状态和私有数据等；
3. 有利于系统的维护和演化：服务请求者和服务提供者的有效分离。

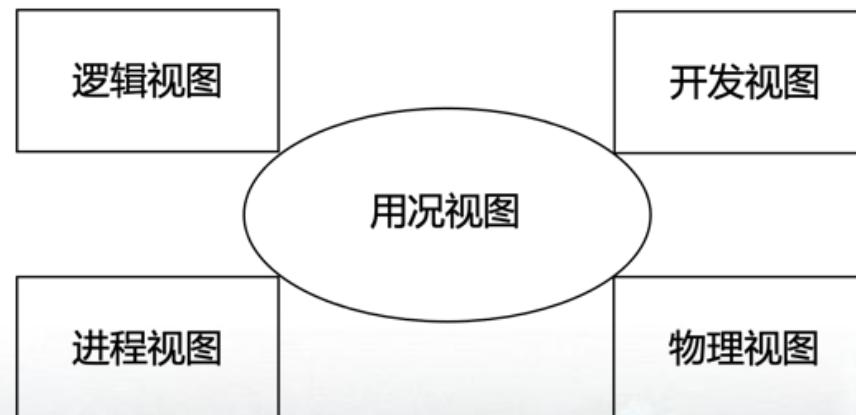


北京大学

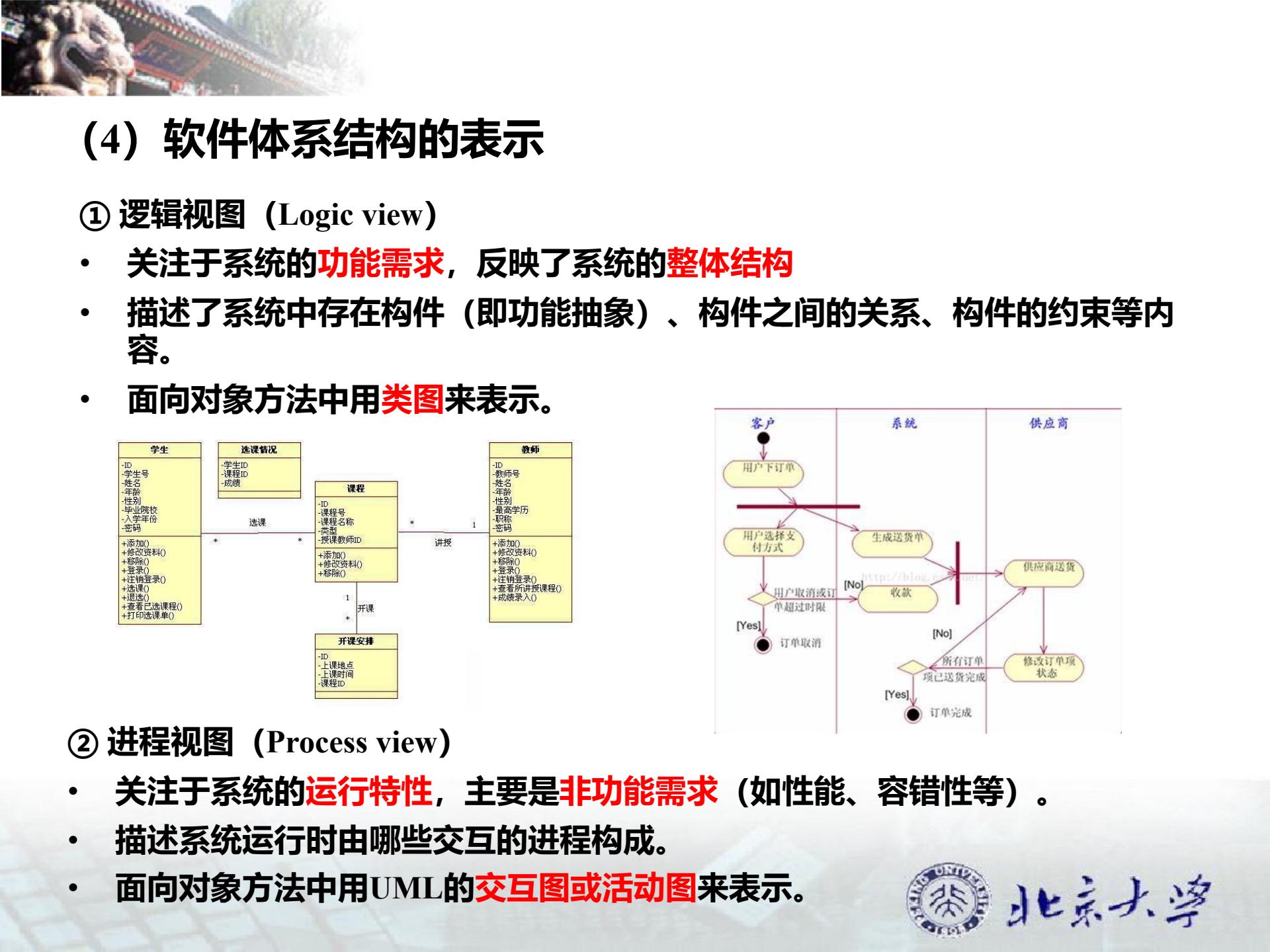


## (4) 软件体系结构的表示

- 软件体系结构模型是在系统设计阶段**描述设计方案**的表示工具，用于**设计的文档化**。
- 不同模型的意图、角度、抽象层次不同，单个体系结构模型很难描述软件的所有信息：我们需要从多个视角/视图描述软件体系结构。
- 1995年，Philippe Kruchten在《IEEE Software》上提出了著名的“4+1”视图来表示软件体系结构：4个主视图+1个用况图。



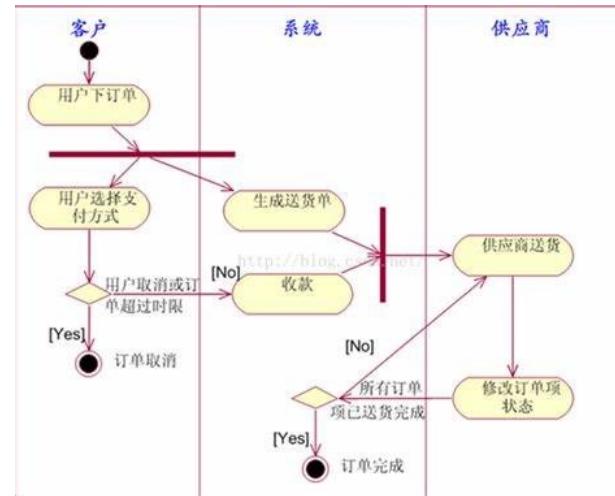
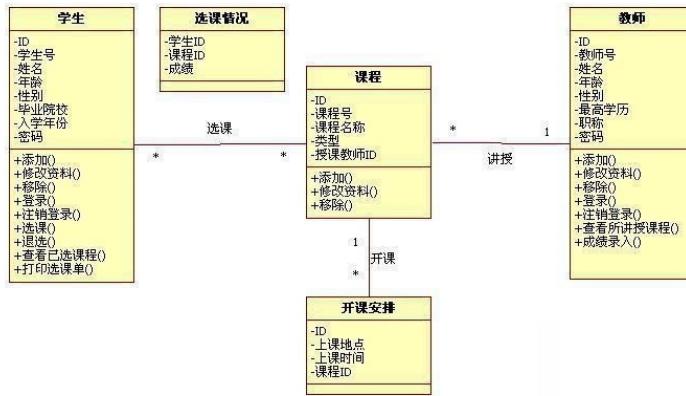
北京大学



## (4) 软件体系结构的表示

### ① 逻辑视图 (Logic view)

- 关注于系统的**功能需求**, 反映了系统的**整体结构**
- 描述了系统中存在构件 (即功能抽象) 、构件之间的关系、构件的约束等内容。
- 面向对象方法中用**类图**来表示。

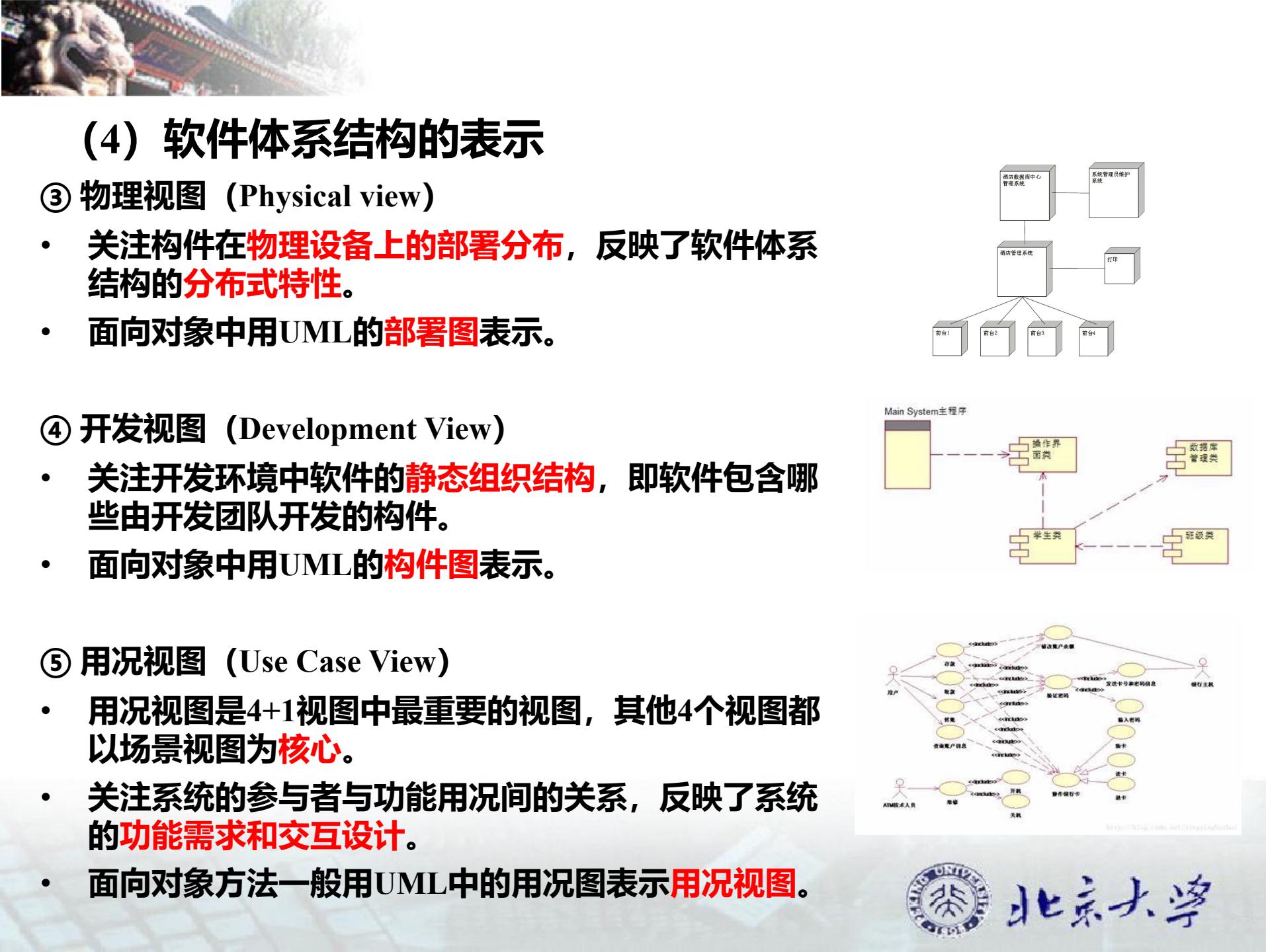


### ② 进程视图 (Process view)

- 关注于系统的**运行特性**, 主要是**非功能需求** (如性能、容错性等) 。
- 描述系统运行时由哪些交互的进程构成。
- 面向对象方法中用UML的**交互图或活动图**来表示。



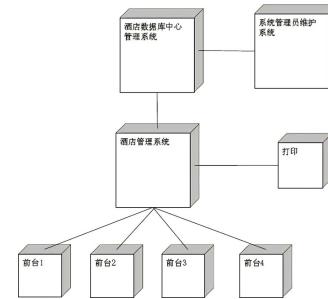
北京大学



## (4) 软件体系结构的表示

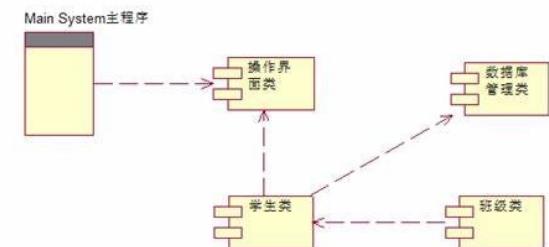
### ③ 物理视图 (Physical view)

- 关注构件在物理设备上的部署分布，反映了软件体系结构的分布式特性。
- 面向对象中用UML的部署图表示。



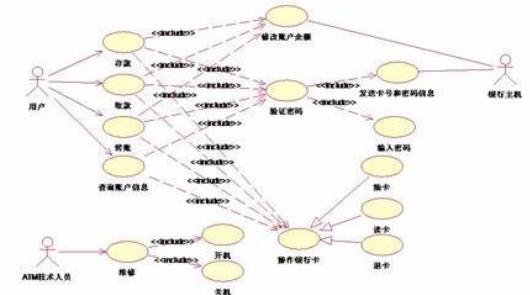
### ④ 开发视图 (Development View)

- 关注开发环境中软件的静态组织结构，即软件包含哪些由开发团队开发的构件。
- 面向对象中用UML的构件图表示。

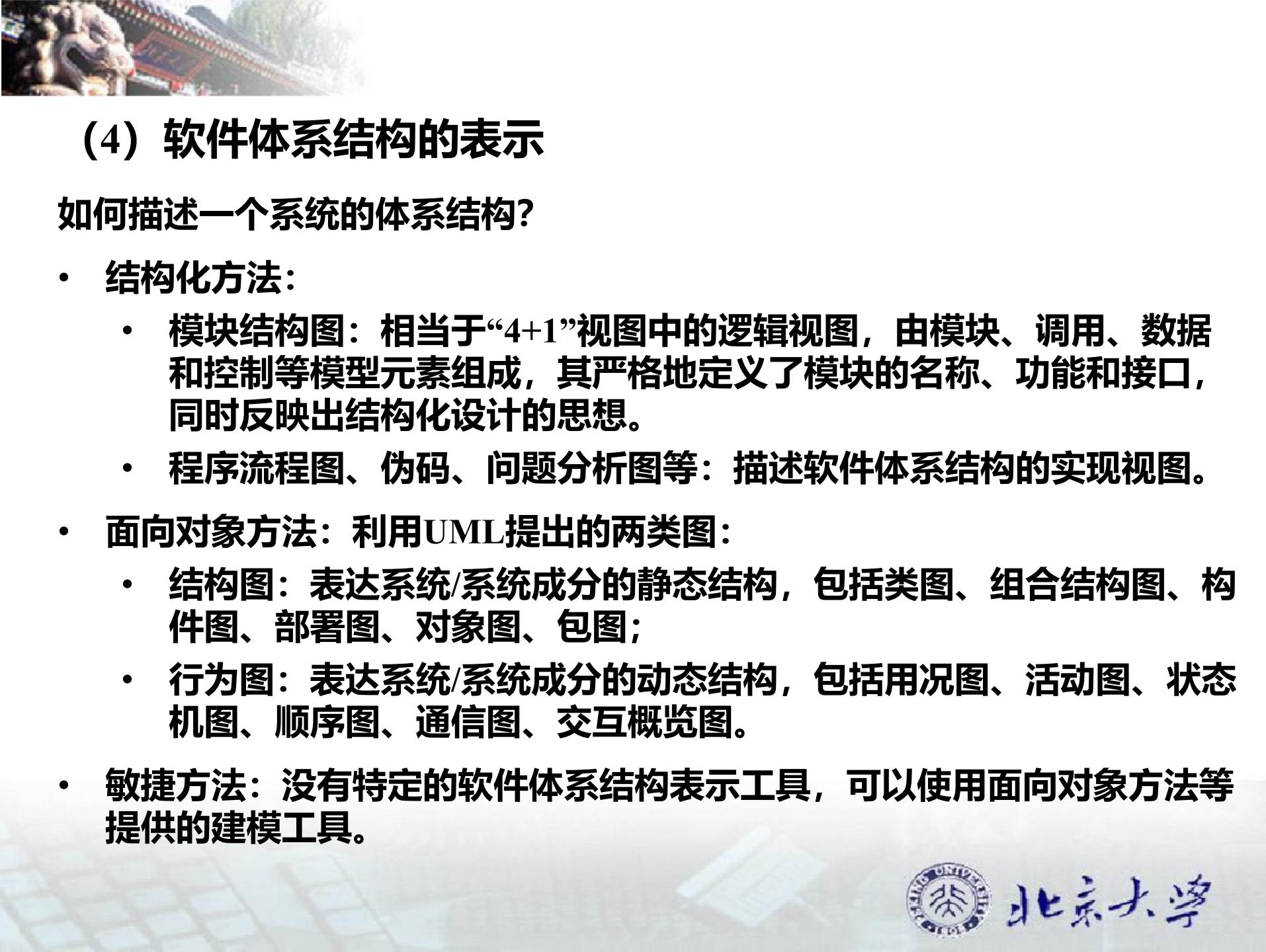


### ⑤ 用况视图 (Use Case View)

- 用况视图是4+1视图中最重要的视图，其他4个视图都以场景视图为核心。
- 关注系统的参与者与功能用况间的关系，反映了系统的功能需求和交互设计。
- 面向对象方法一般用UML中的用况图表示用况视图。



北京大学



## (4) 软件体系结构的表示

如何描述一个系统的体系结构？

- 结构化方法：
  - 模块结构图：相当于“4+1”视图中的逻辑视图，由模块、调用、数据和控制等模型元素组成，其严格地定义了模块的名称、功能和接口，同时反映出结构化设计的思想。
  - 程序流程图、伪码、问题分析图等：描述软件体系结构的实现视图。
- 面向对象方法：利用UML提出的两类图：
  - 结构图：表达系统/系统成分的静态结构，包括类图、组合结构图、构件图、部署图、对象图、包图；
  - 行为图：表达系统/系统成分的动态结构，包括用况图、活动图、状态机图、顺序图、通信图、交互概览图。
- 敏捷方法：没有特定的软件体系结构表示工具，可以使用面向对象方法等提供的建模工具。



北京大学



## (5) 软件体系结构在软件开发中的作用

软件体系结构对于软件开发（尤其是大型复杂系统）的开发具有重要的作用：

- **是系统相关人员之间相互交流的手段：**

软件体系结构代表了对系统中某些**共性的抽象**，与系统相关的大多数人员都可以以之作为**彼此理解、达成一致和相互交流**的基础。

- **是最初设计决策的体现：**

软件体系结构是系统开发中最早得到、对系统质量属性影响最大的制品，可利用它分析系统需求的优先级，并可以用来在开发成本和性能、安全性、可维护性、可靠性之间做出**权衡**；

- **是系统可复用、可传递的抽象：**

软件体系结构构成了相对较小又容易理解的**模型**，可说明系统构造及各部分之间相互联系。还可对其他有相似需求的系统起参考作用。

软件体系结构为构件的组装提供了**基础和上下文**，其本身也可以作为一种大粒度的、抽象级别较高的软件构件复用，有利于系统较高级别性质的描述和分析。



北京大学

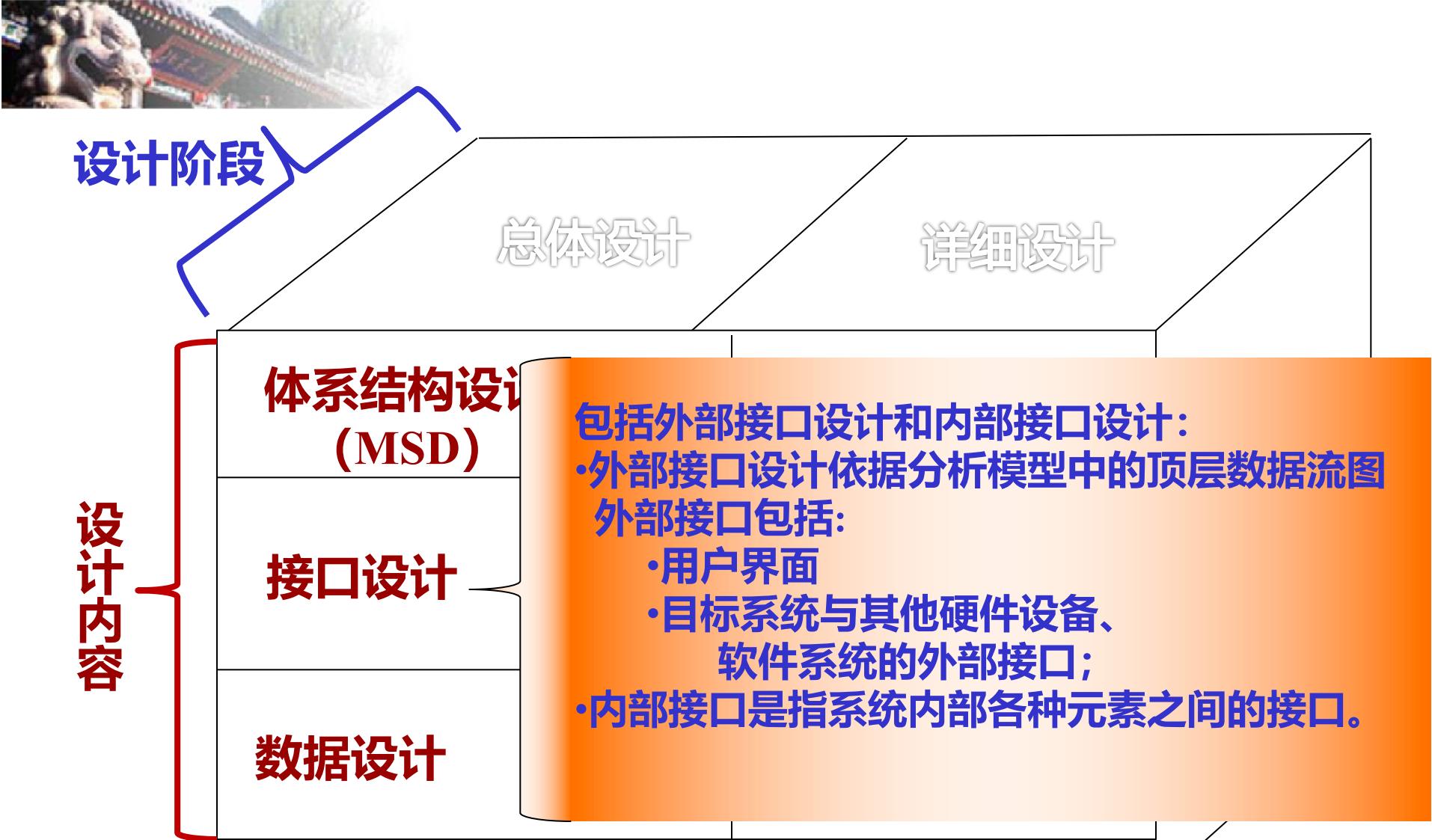


图1 设计阶段和设计内容



北京大学

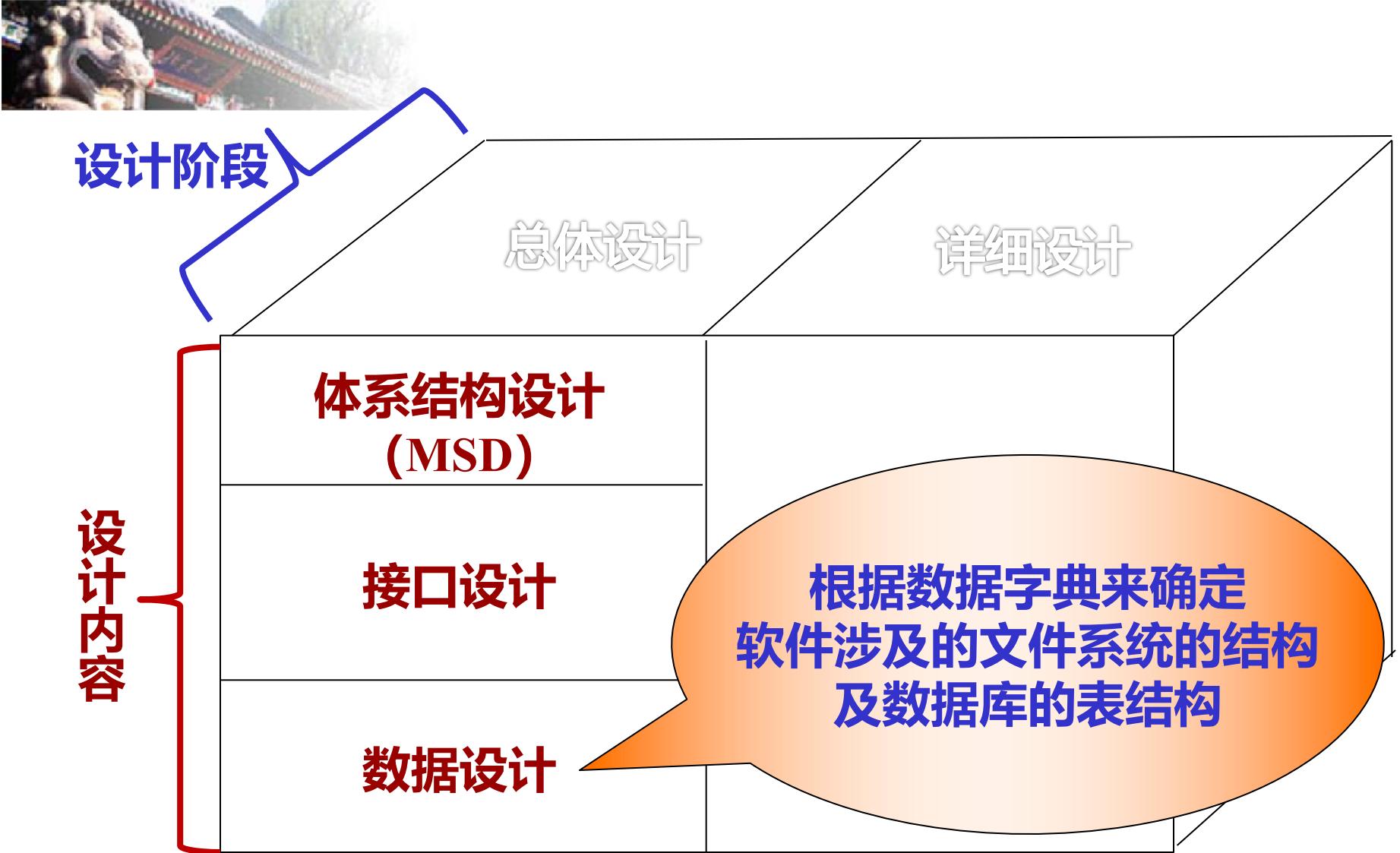


图1 设计阶段和设计内容



北京大学



## 2. 实现软件设计的目标对结构化设计方法的需求

- (1) 提供可体现“原理/原则”的一组术语(符号)，形成一个特定的抽象层，用于表达设计中所使用的部件。
- (2) 依据术语所形成的“空间”，给出表达软件模型工具。
- (3) 给出设计的过程指导。



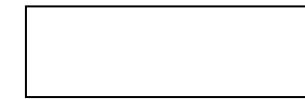
北京大学

### 3. 结构化设计方法

#### (1) 在总体设计层

##### ① 引入了两个术语/符号

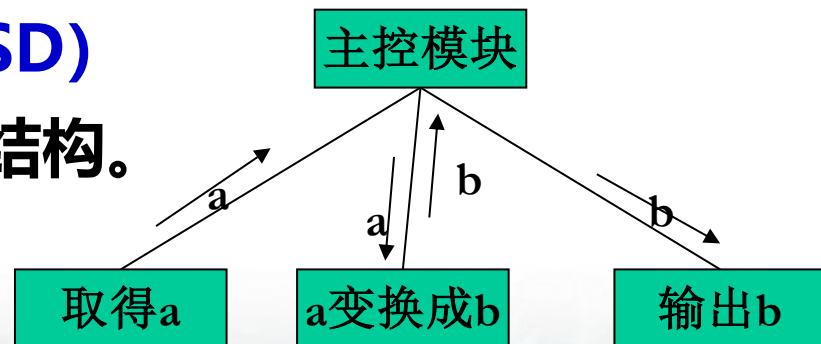
模块：一种可独立标识的软件成分。



调用：模块间的一种关系，模块A为了完成其任务必须  
依赖其他模块，表示为线段

##### ② 引入了模块结构图 (MSD)

用于表达软件系统的静态结构。



北京大学



### ③ 过程指导

为了实现设计目标, 总体设计的具体任务是:

**将DFD转化为MSD**

分二步实现:

**第一步: 如何将DFD转化为初始的MSD**

**分类: 变换型数据流图**

**事务型数据流图**

**变换设计**

**事务设计**

**第二步: 如何将初始的MSD转化为最终可供详细设计使用的  
MSD**



**北京大学**



总体设计分为三个阶段：

**第一阶段：初始设计。在对给定的数据流图进行复审和精化的基础上，将其转化为初始的模块结构图。根据穿越系统边界的数据流初步确定系统与外部的接口。**

**第二阶段：精化设计。依据模块“高内聚低耦合”的原则，精化初始的模块结构图，并设计其中的全局数据结构和每一模块的接口。**

**第三阶段：设计复审阶段，对前两个阶段得到的高层软件结构进行复审，必要时还可能需要对软件结构做一些精化工作。**



北京大学

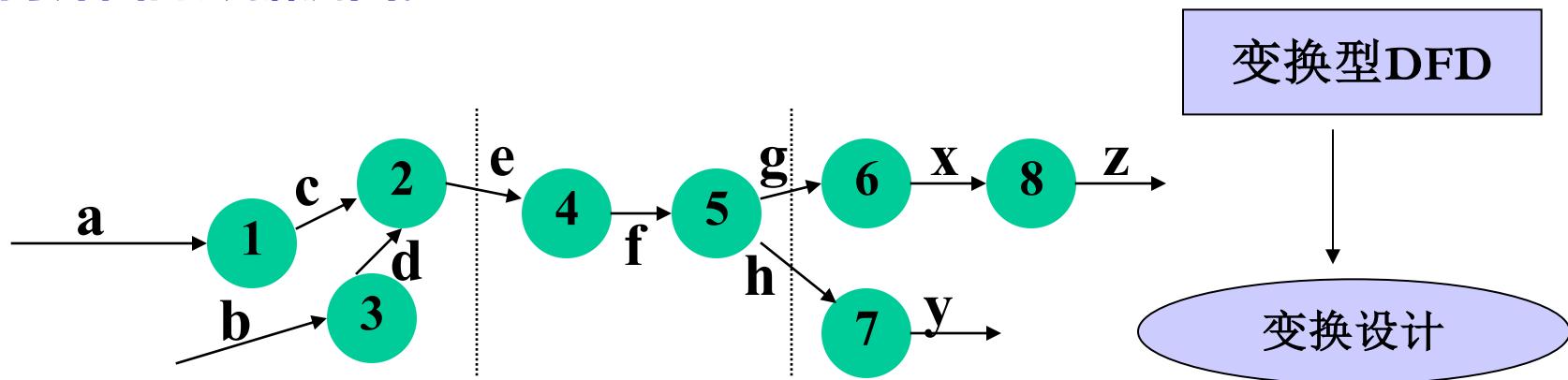


## 4. 总体设计第一步：DFD→初始的MSD

### (1) 数据流图分类

- 变换型数据流图

具有较明显的输入部分和变换部分之间的界面、变换部分和输出部分之间界面的数据流图。

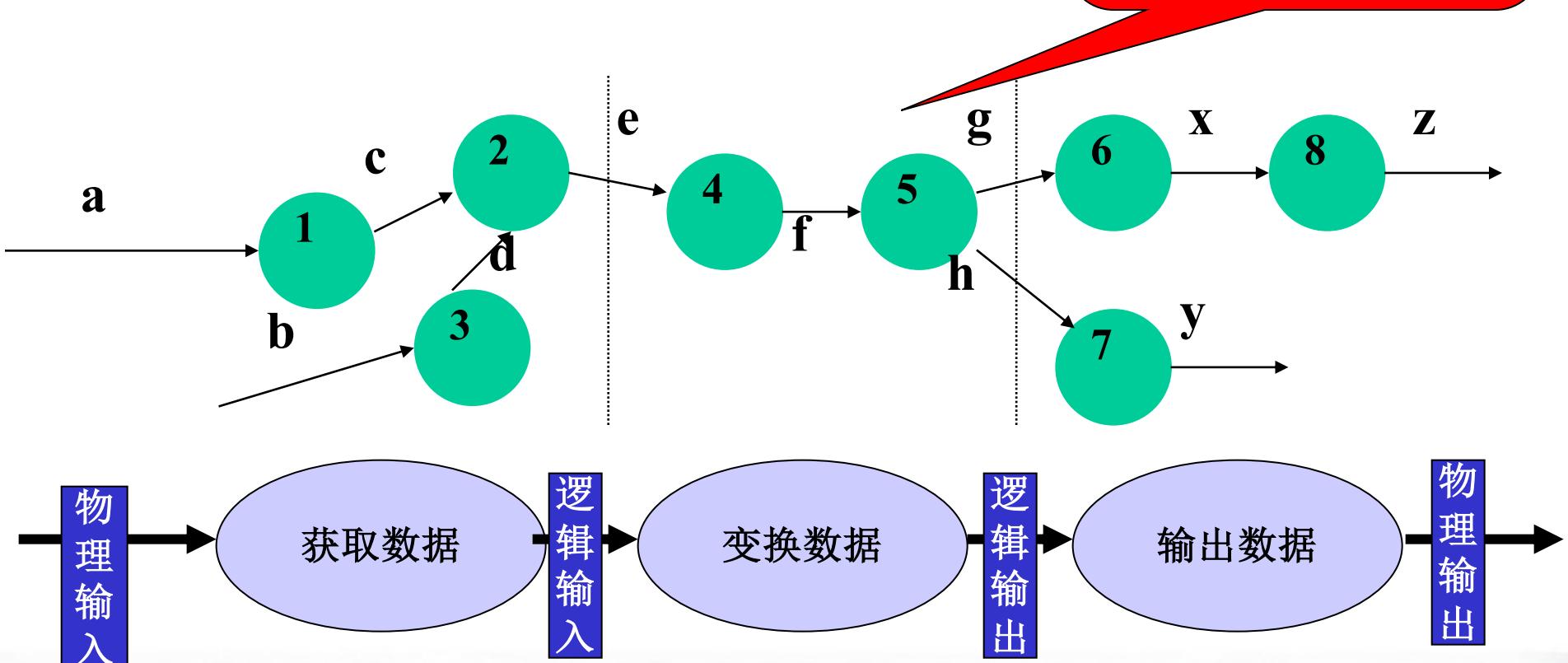




# 变换型 DFD

问题:

逻辑输入: e  
逻辑输出: g,h  
物理输入: a, b  
物理输出: z,y



- 逻辑输入: 离物理输入最远、仍被看成系统输入的数据流
- 逻辑输出: 离物理输出最远、仍被看成系统输出的数据流

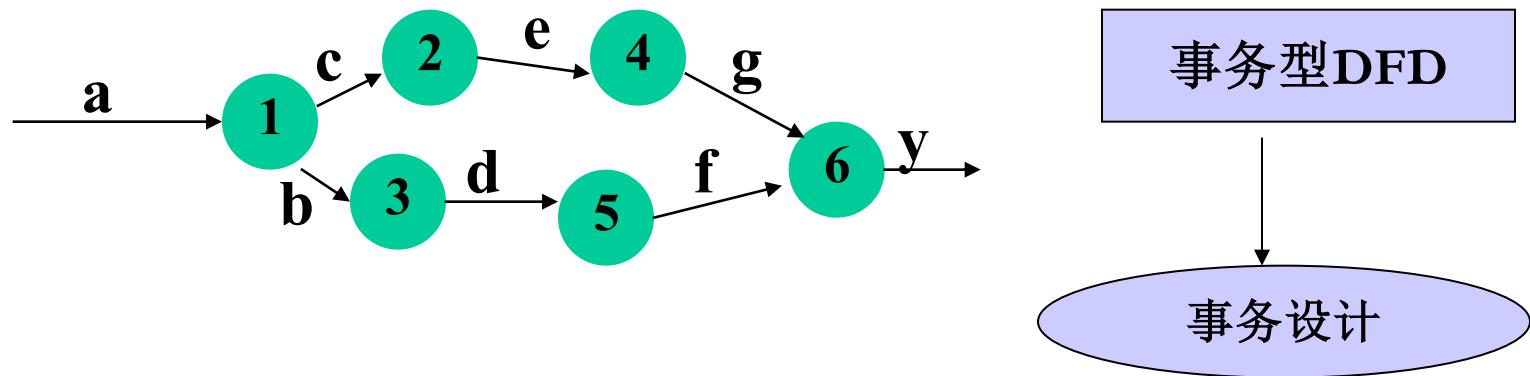


北京大学



- 事务型数据流图:

数据到达一个加工（例如下图1），该加工根据输入数据的值，在其后的若干动作序列（称为一个事务）中选出一个来执行，这类数据流图称为事务型数据流图。



北京大学



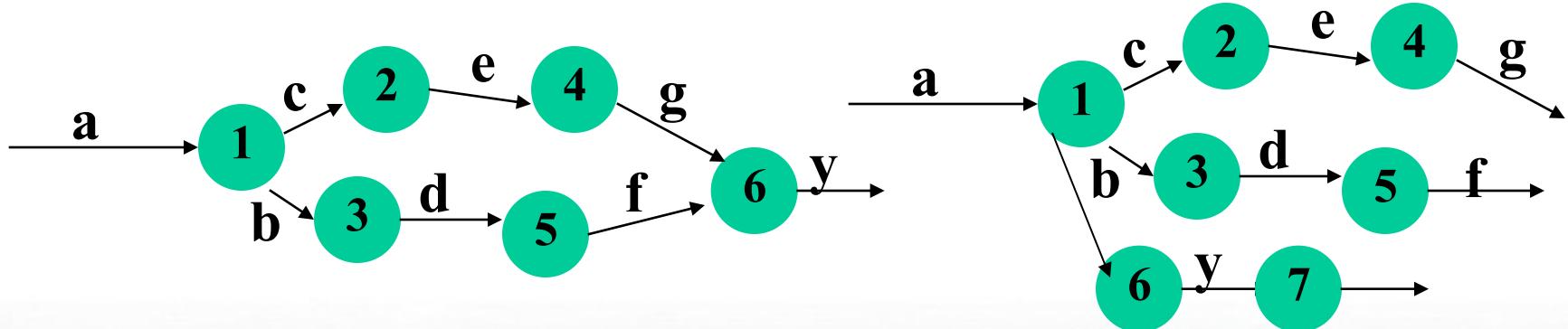
# 事务型 DFD

- 事务型DFD完成下述任务

- 1) 接受输入数据
- 2) 分析并确定对应的事务
- 3) 选取与该事务对应的一条活动路径

- 事务型DFD和变换型DFD的区别

- 原则上所有DFD都可以看成是变换型DFD
- 一般而言，接受1个输入数据，分成多条路径



北京大学



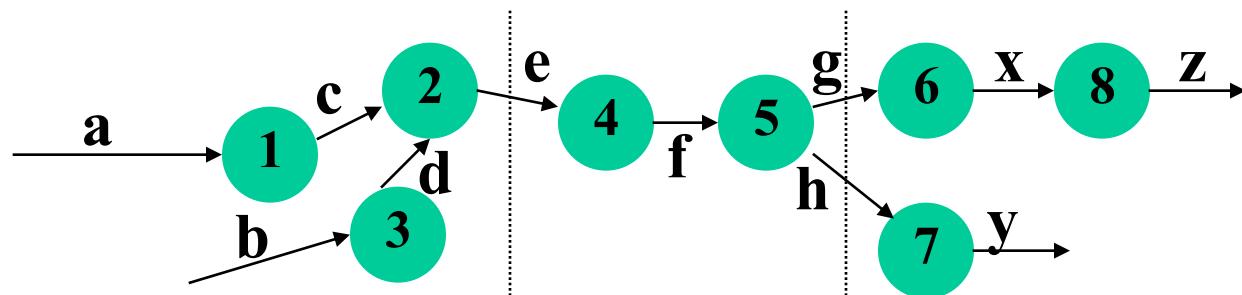
## (2) 变换设计的基本步骤

### ① 第1步：设计准备—复审并精化系统模型

- 为了确保系统的输入数据和输出数据符合实际情况而复审其语境
- 为了确保是否需要进一步精化系统的DFD图而复审其语境

### ② 第2步：确定输入、变换、输出这三部分之间的边界

- 根据加工的语义和相关的数据流，确定系统的逻辑输入和逻辑输出

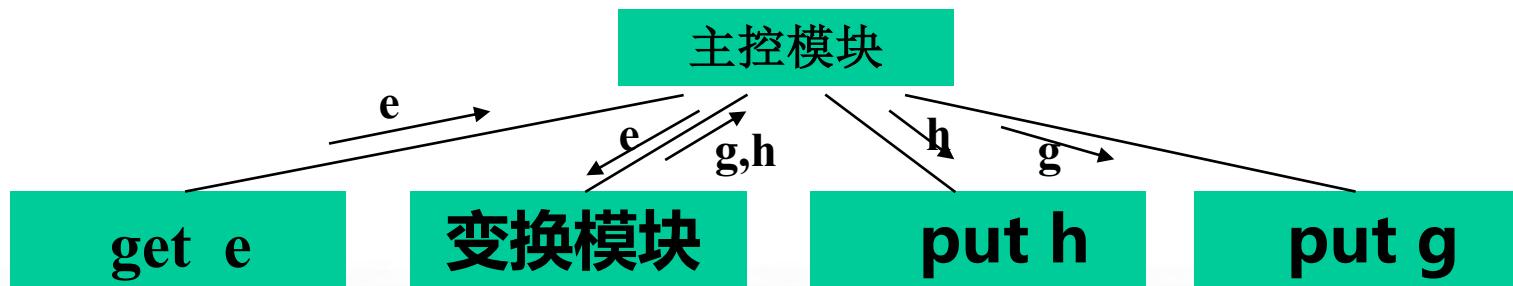


北京大学



### ③ 第3步：第一级分解—系统模块结构图顶层和第一层的设计

- **主模块：**位于最顶层，一般以所建系统的名字命名，其任务是协调控制第一层模块
- **输入模块部分：**为主模块提供加工数据，有几个逻辑输入就设计几个输入模块
- **变换模块部分：**接受输入模块部分的数据，并对内部形式的数据加工，产生系统所有的内部输出数据
- **输出模块部分：**将变换模块产生的输出数据，以用户可见的形式输出。有几个逻辑输出，就设计几个输出模块

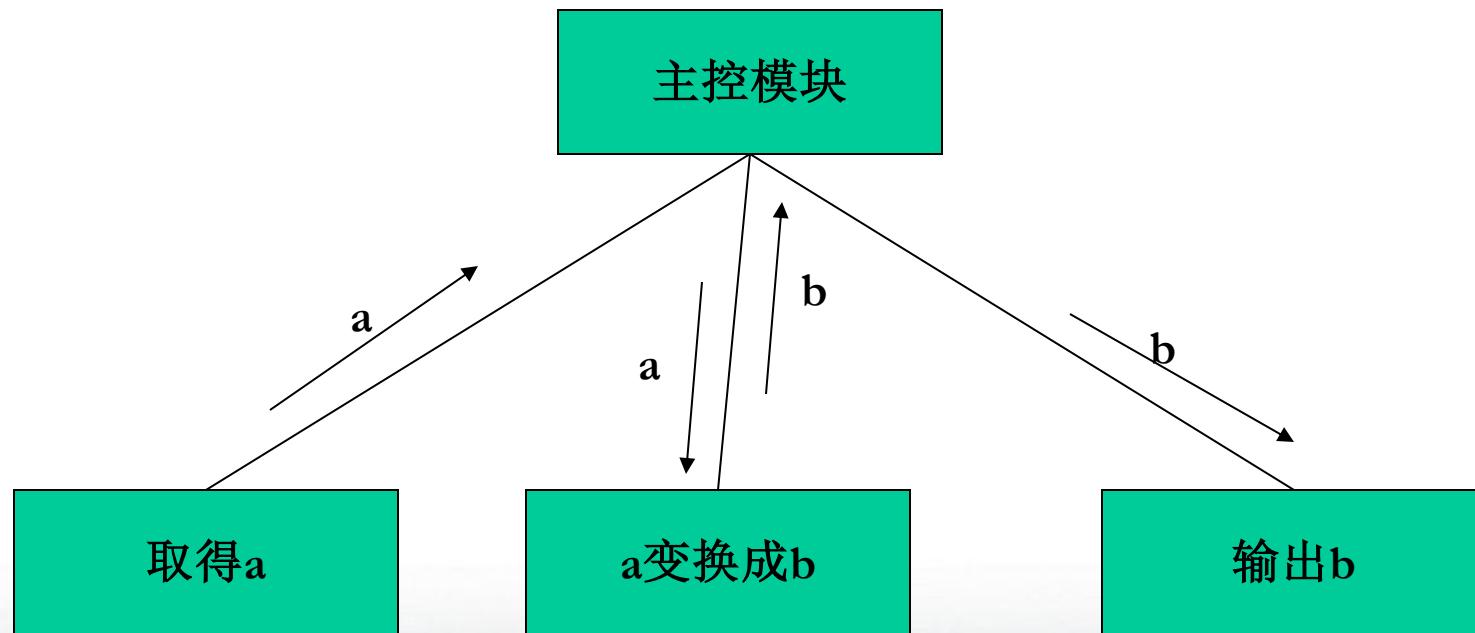




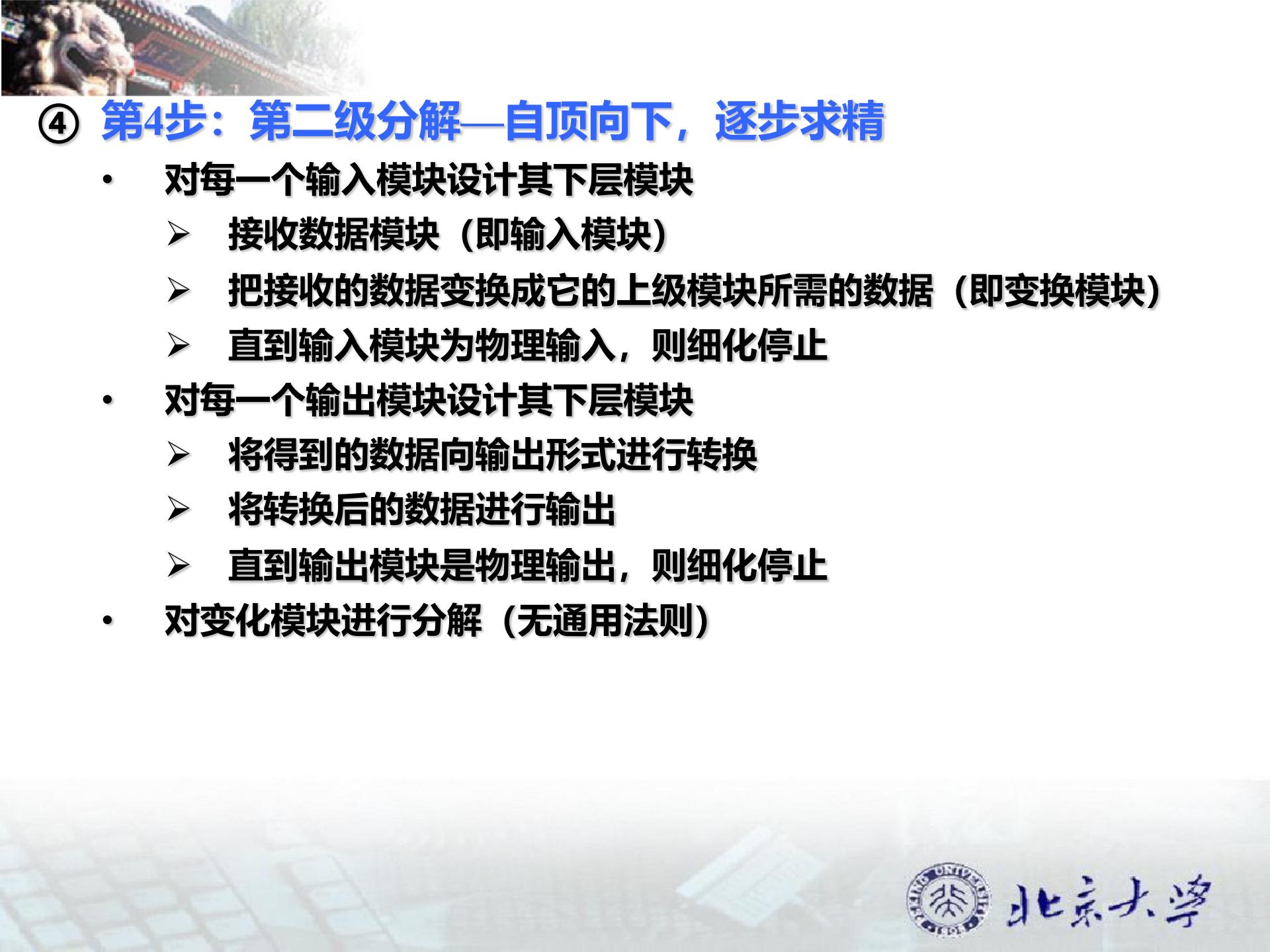
# 最简单的变换型DFD



逻辑输入=物理输入  
逻辑输出=物理输出



北京大学

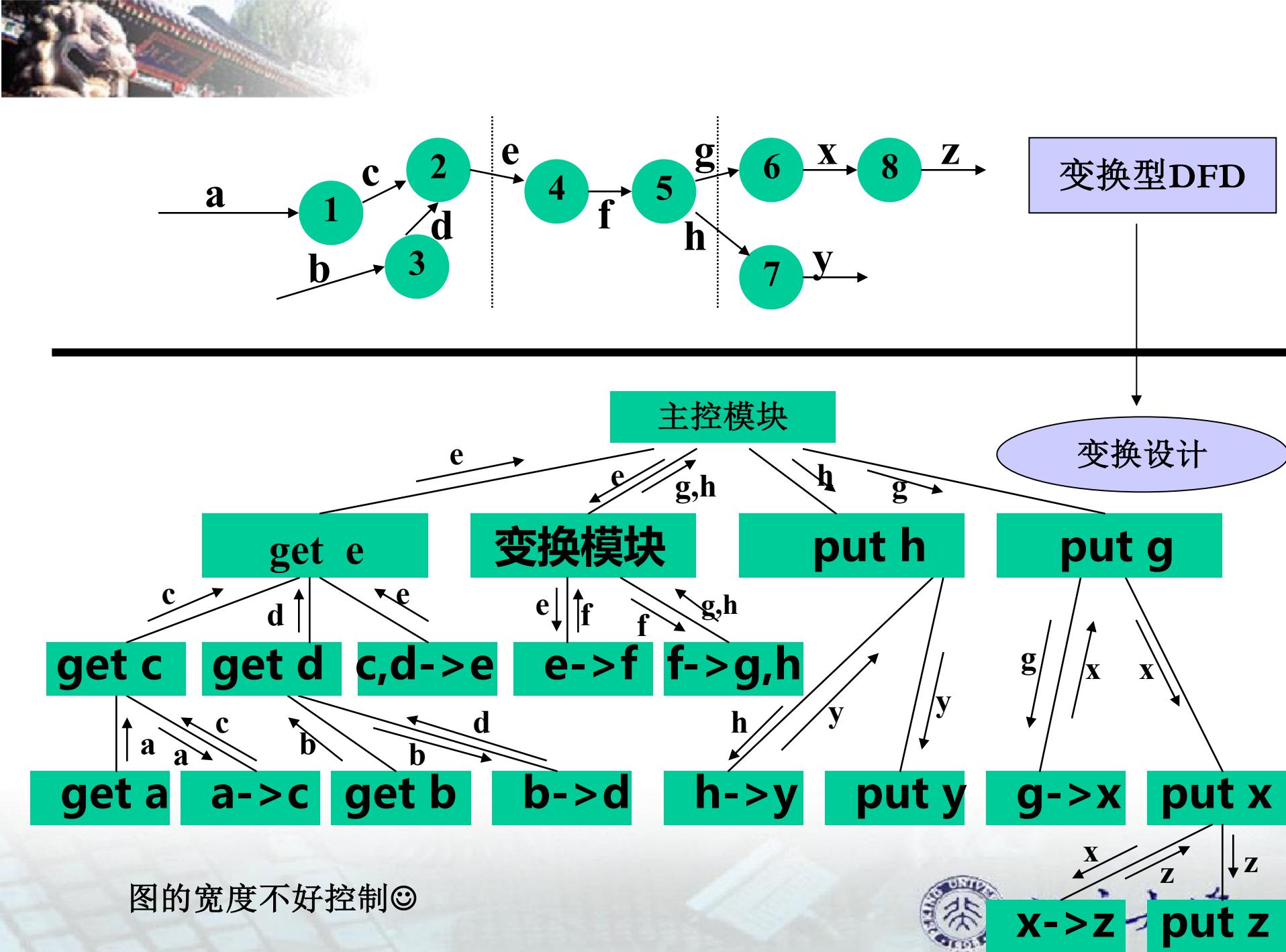


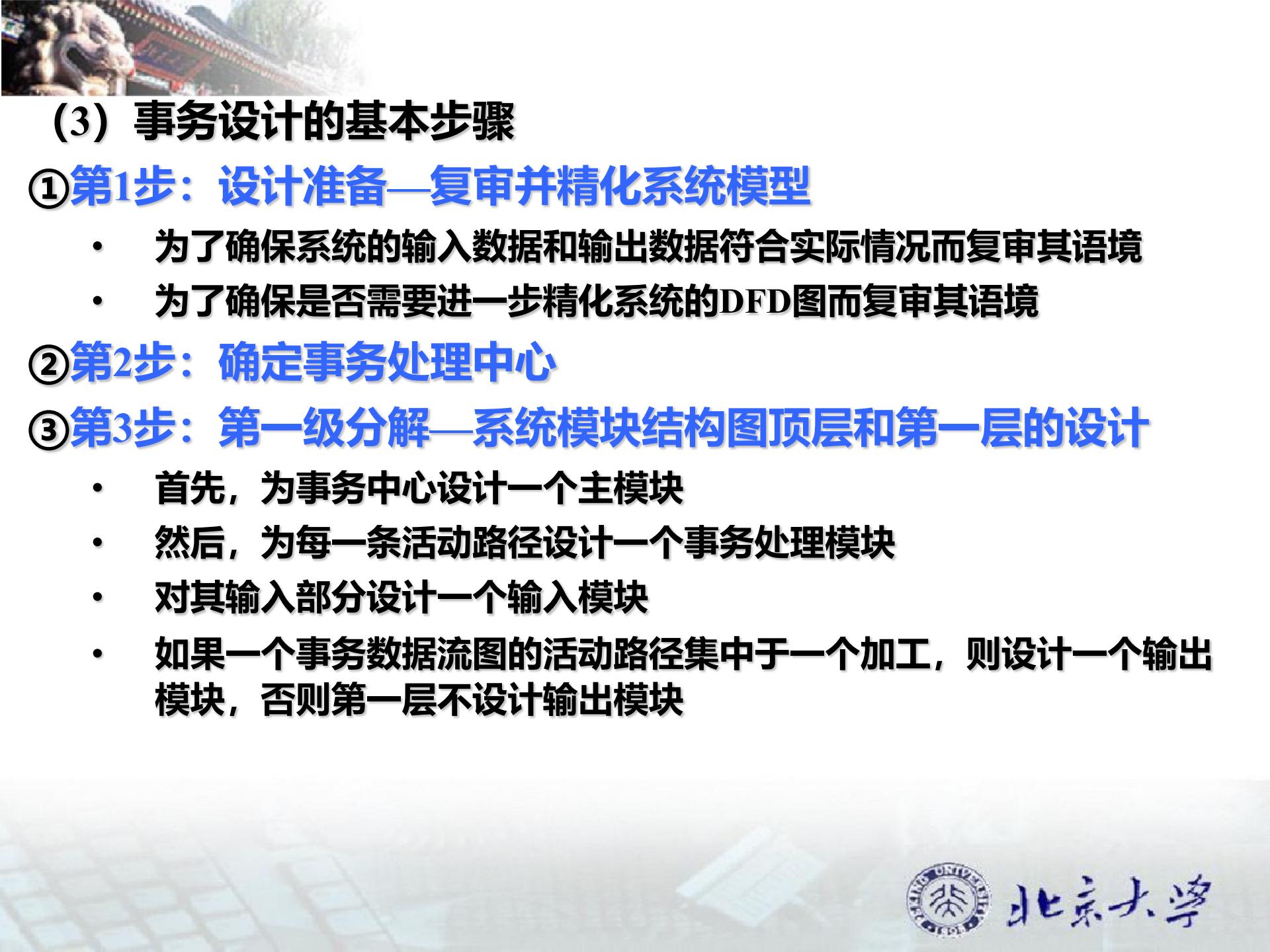
## ④ 第4步：第二级分解—自顶向下，逐步求精

- 对每一个输入模块设计其下层模块
  - 接收数据模块（即输入模块）
  - 把接收的数据转换成它的上级模块所需的数据（即变换模块）
  - 直到输入模块为物理输入，则细化停止
- 对每一个输出模块设计其下层模块
  - 将得到的数据向输出形式进行转换
  - 将转换后的数据进行输出
  - 直到输出模块是物理输出，则细化停止
- 对变化模块进行分解（无通用法则）



北京大学





### (3) 事务设计的基本步骤

#### ① 第1步：设计准备—复审并精化系统模型

- 为了确保系统的输入数据和输出数据符合实际情况而复审其语境
- 为了确保是否需要进一步精化系统的DFD图而复审其语境

#### ② 第2步：确定事务处理中心

#### ③ 第3步：第一级分解—系统模块结构图顶层和第一层的设计

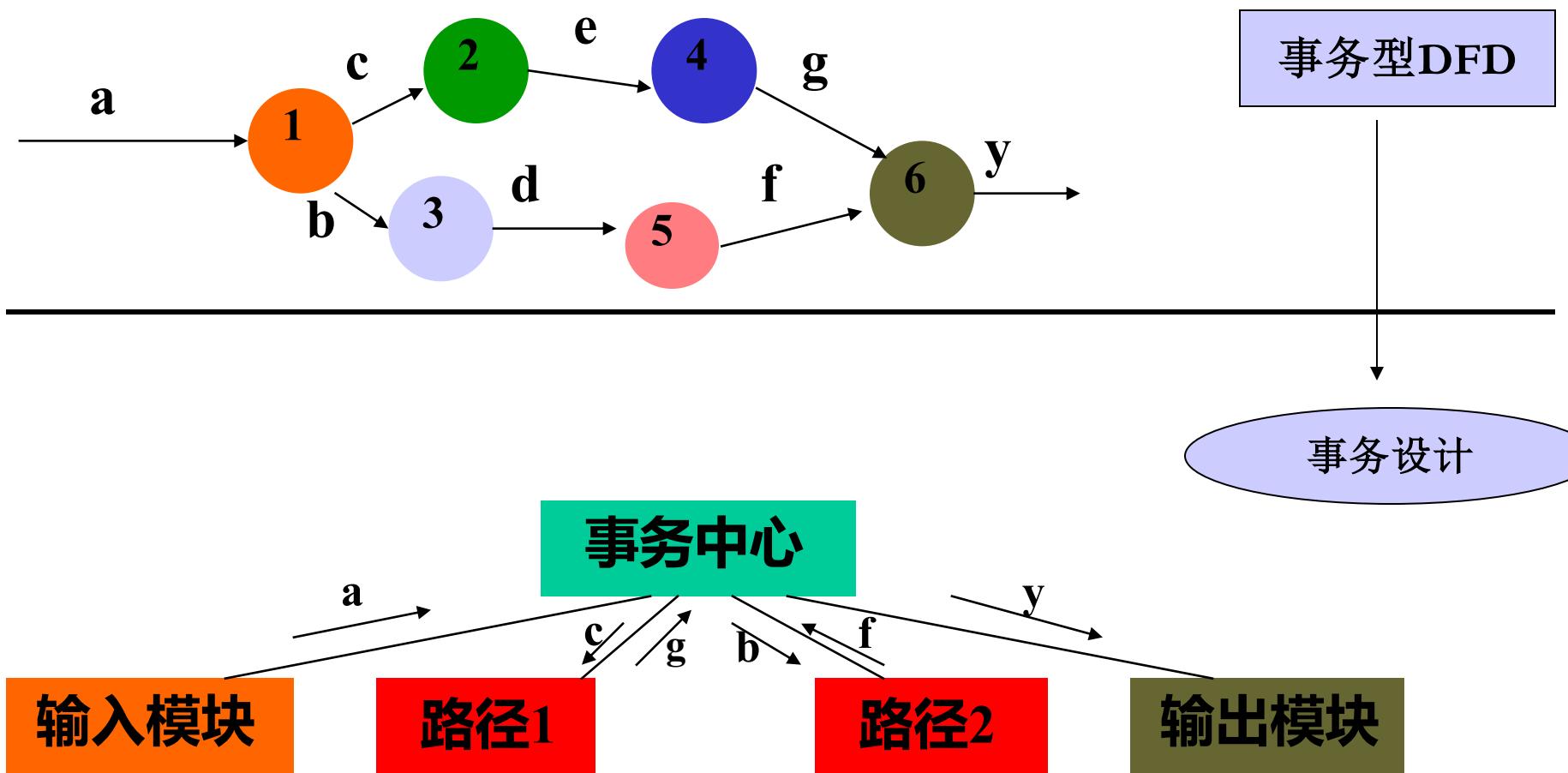
- 首先，为事务中心设计一个主模块
- 然后，为每一条活动路径设计一个事务处理模块
- 对其输入部分设计一个输入模块
- 如果一个事务数据流图的活动路径集中于一个加工，则设计一个输出模块，否则第一层不设计输出模块



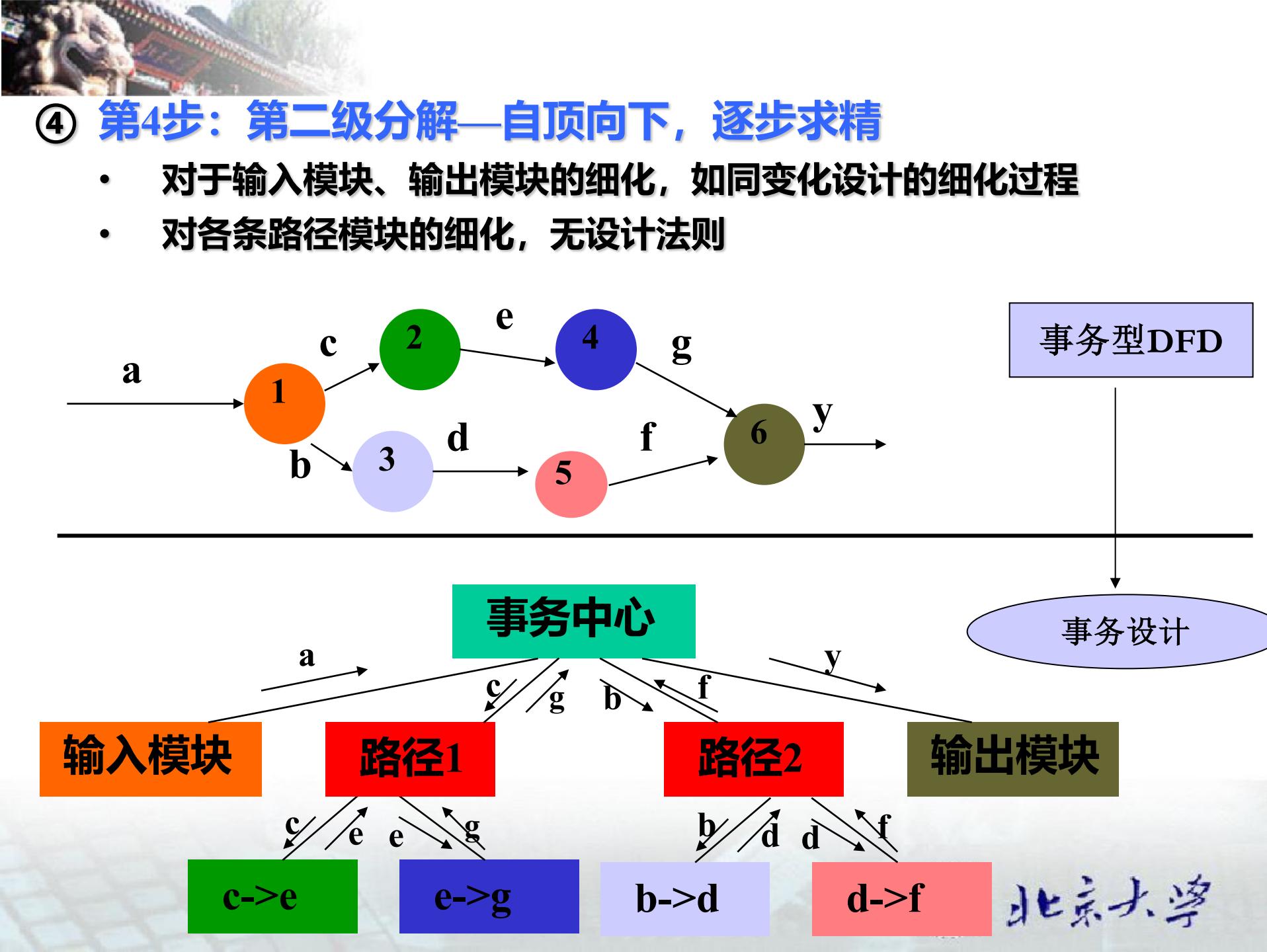
北京大学



## 第一级分解举例：



北京大学

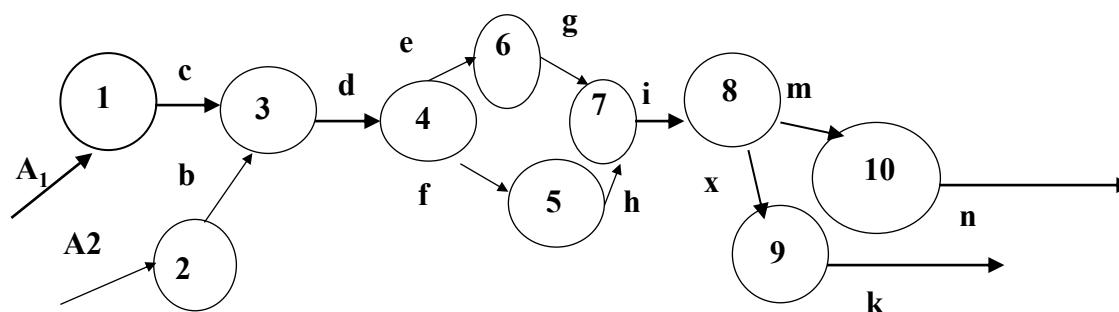




# 总体设计第一步

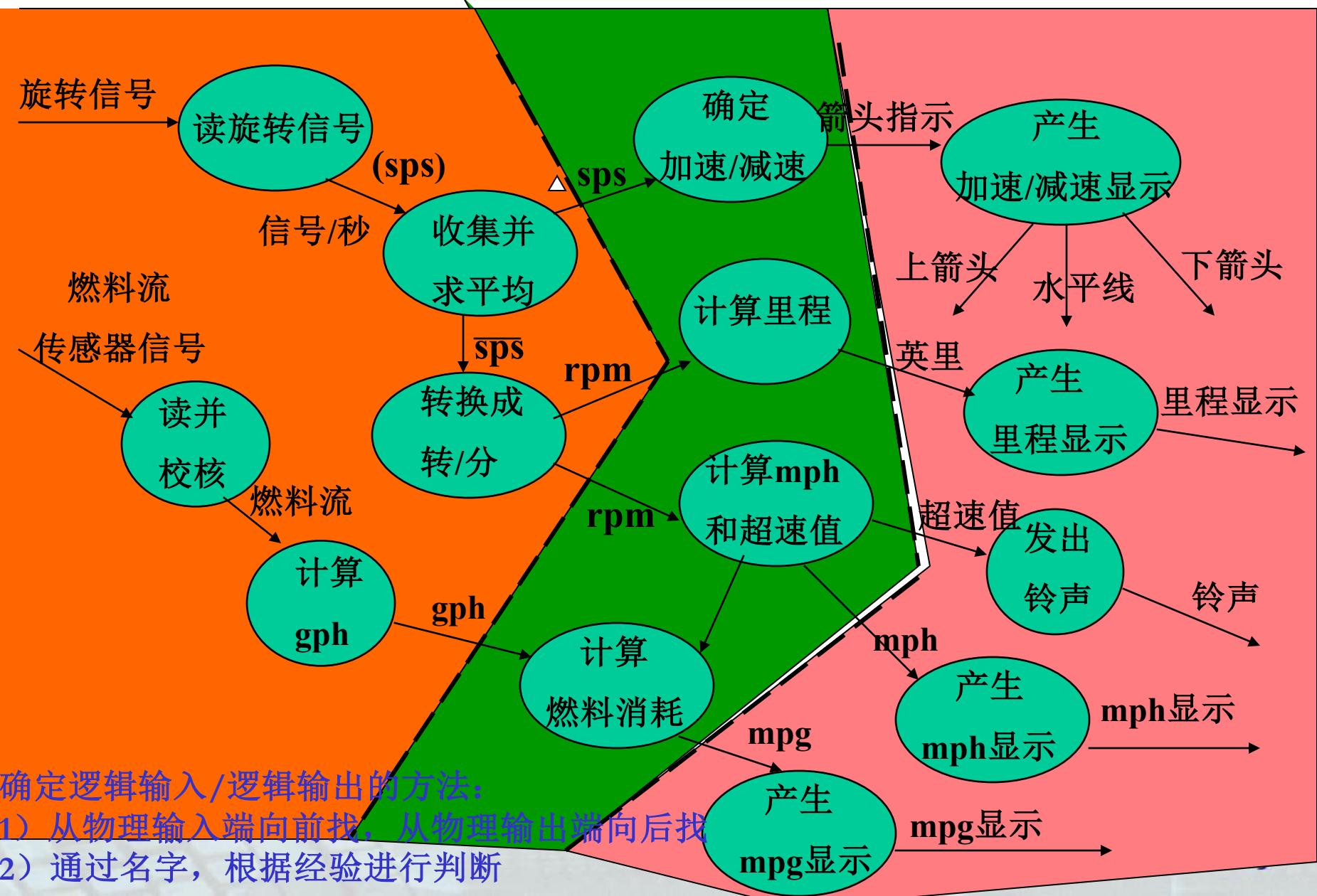
- DFD→初始的MSD

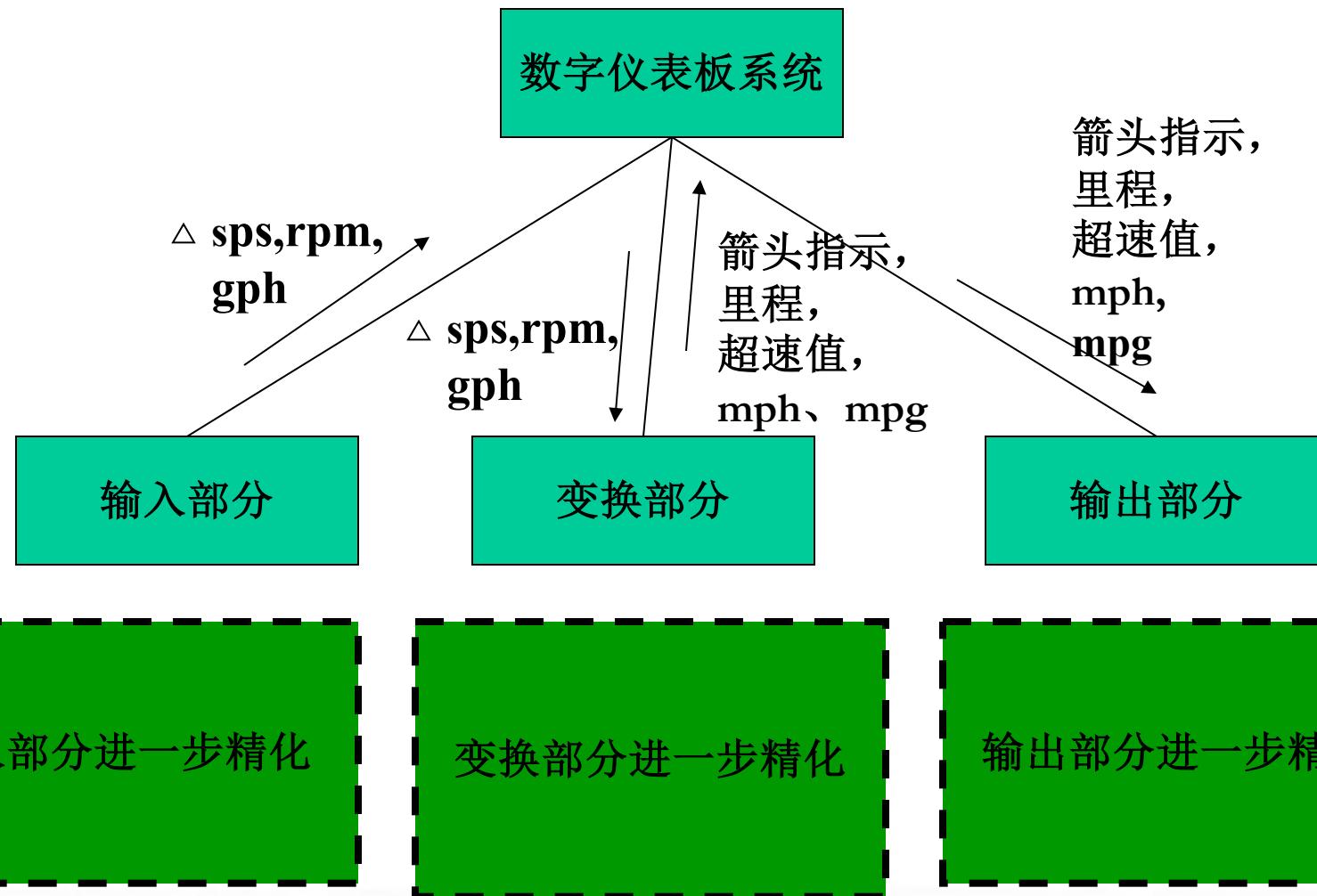
- 一个系统的DFD，通常是变换型数据流图和事务型数据流图的组合
- 自动的变换设计
- 自动的事务设计



北京大学

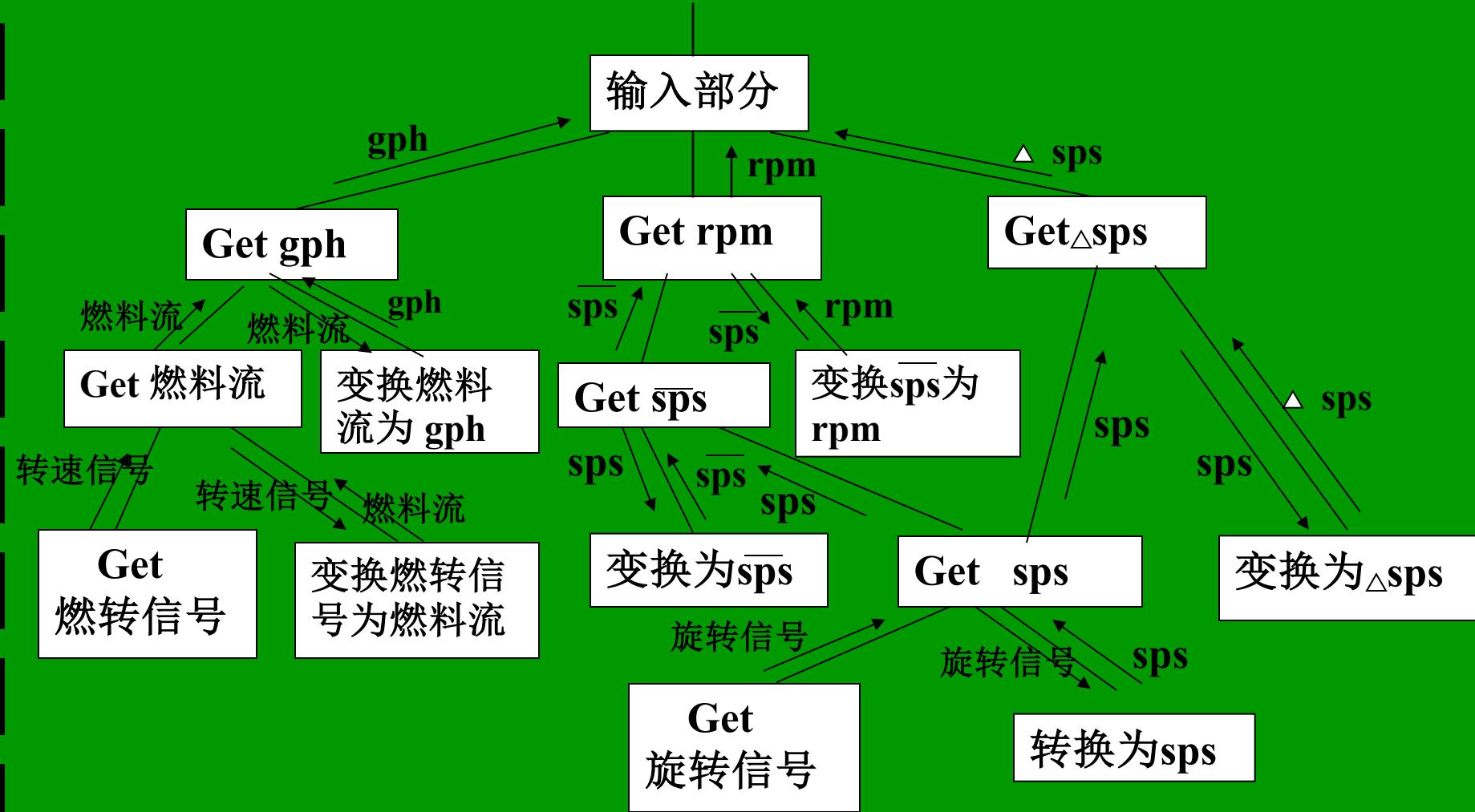
# 例子：数字仪表板系统



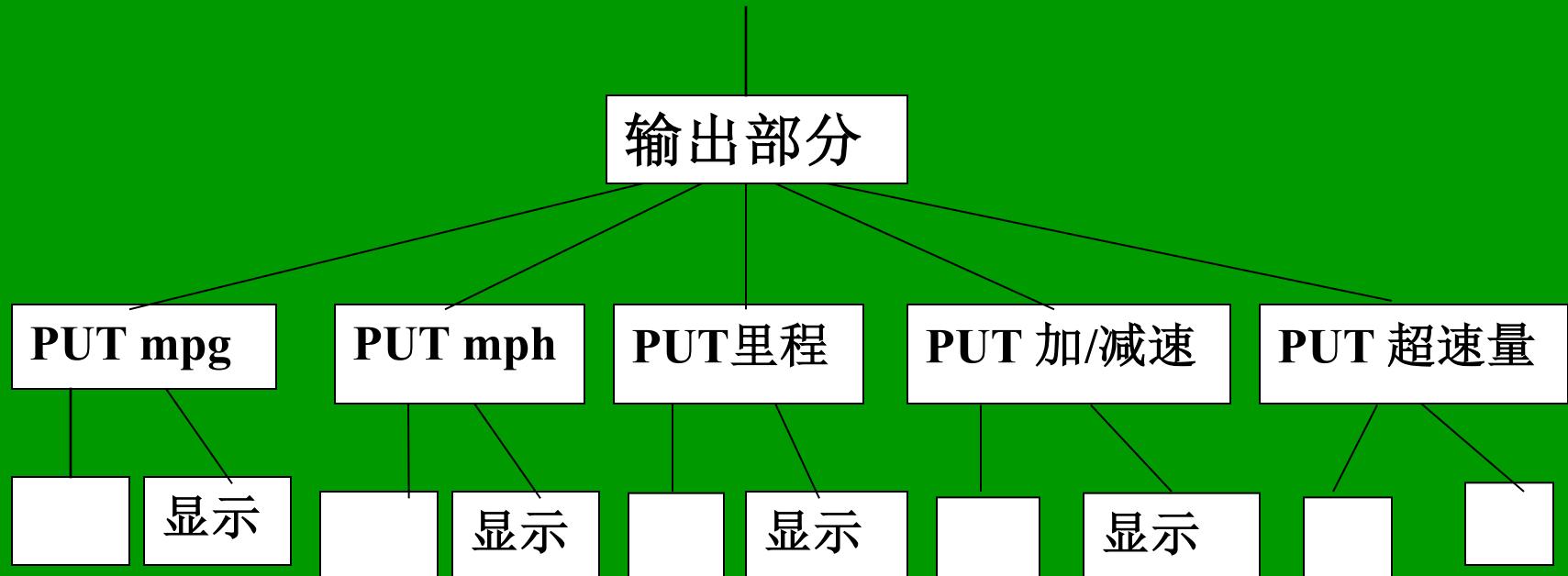


其中：sps为转速的每秒信号量； $\bar{sps}$  为sps的平均值； $\triangle sps$ 为sps的瞬时变化值；rpm为每分钟转速；mph为每小时英里数；gph为每小时燃烧的燃料加仑数；m为行进里程。

## 输入部分进一步精化



## 输出部分进一步精化





## 5. 总体设计第二步：将初始的MSD转化为最终可供详细设计使用的MSD

概念：模块，模块化

基于模块化原理-高内聚 低耦合，  
给出设计规则 - 经验规则-启发式规则  
用于精化初始的MSD  
- 体现设计人员的创造



北京大学



# 模块 和 模块化

- 模块：执行一个特殊任务的一组例程和数据结构
  - 接口：给出可由其他模块和例程访问的对象
    - 常量，变量，数据类型，函数
  - 实现：接口的实现（模块功能的执行机制）
    - 私有量，过程描述，源程序代码
- 模块化：把系统分解成若干模块的过程
  - 50多年的历史
  - 软件的单个属性，使得程序能够被理性的管理
    - Myers, G. *Composite Structured Design*, 1978



北京大学



# 为什么要模块化？（1）

- 设 $C(x)$ 是定义问题 $x$ 复杂性的函数， $E(x)$ 是定义解决问题 $x$ 所需要的工作量，那么，对于两个问题 $p_1$ 和 $p_2$ ，如果

$$C(p_1) > C(p_2)$$

- 那么

$$E(p_1) > E(p_2)$$

解释：解决困难问题需要花费更多的时间



北京大学



# 为什么要模块化？（2）

- 人们又发现了另外一个有趣的特征：

$$C(p_1+p_2) > C(p_1) + C(p_2)$$

- 由上页结论：

$$\text{If } C(g_1) > C(g_2) \text{ Then } E(g_1) > E(g_2)$$

- 所以：

$$E(p_1+p_2) > E(p_1) + E(p_2)$$



北京大学



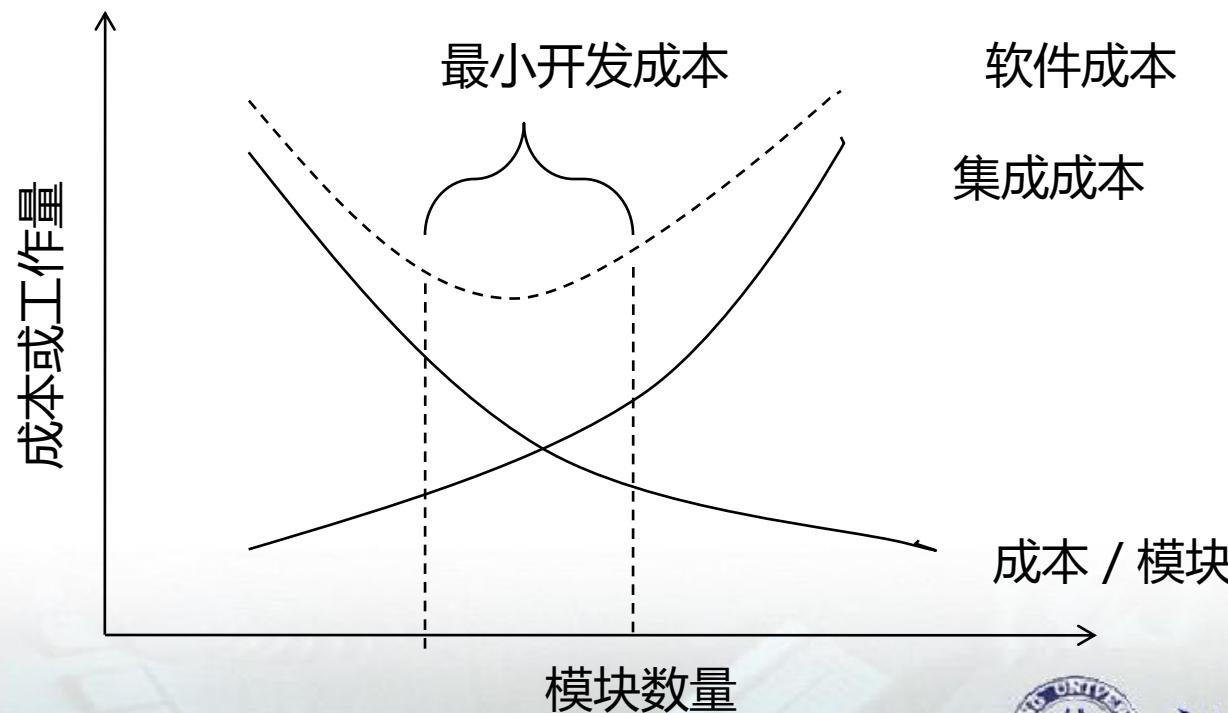
# 为什么要模块化？（3）

## ❖ 一个理想的情况

- 如果我们能够无限制地划分软件，那么开发它所需的工作量可以变得非常小，乃至可以忽略！

## ❖ 但是，这个结论是错误的

- 随着模块数量的增长，集成模块所需的工作量（成本）也在增长。



北京大学



# 模块化的评价

- 基本原则

高内聚，低耦合

- 概念和分类
  - 耦合
  - 内聚
- 启发式规则

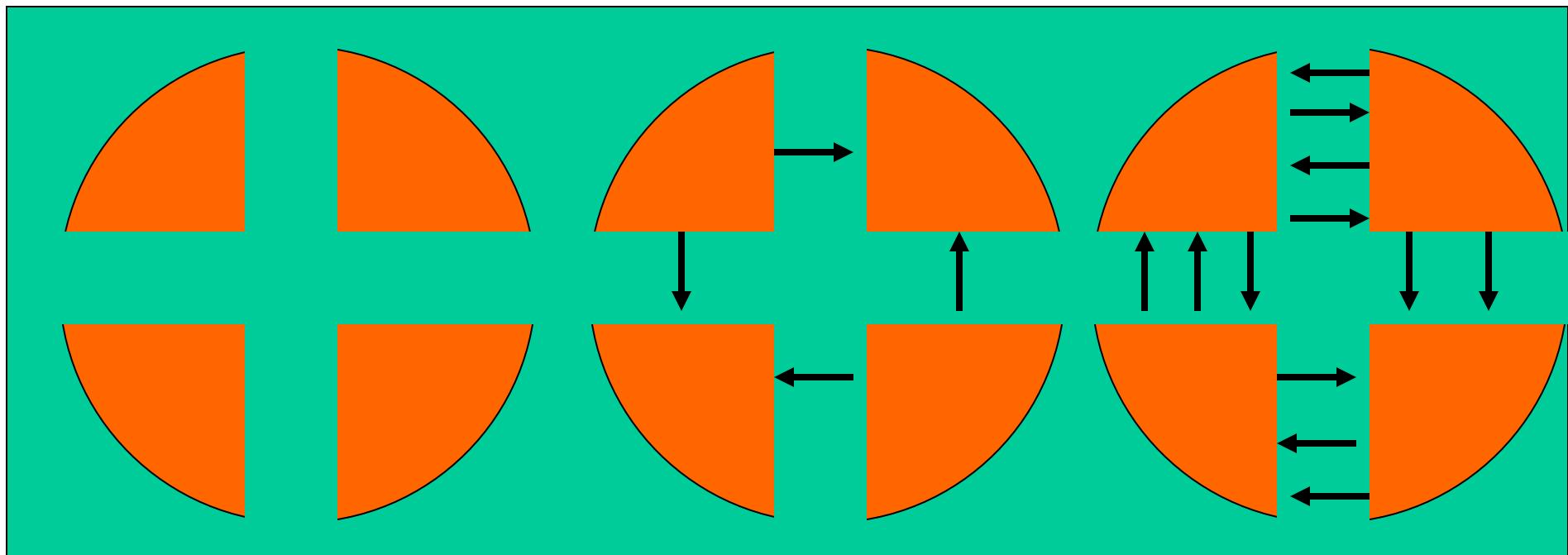


北京大学



# 耦合 (1)

- 定义：不同模块之间相互依赖程度的度量



无耦合

松散耦合

紧密耦合



北京大学



# 耦合 (2)

- 耦合的强度所依赖的因素：
  - 一个模块对另一个模块的引用
  - 一个模块向另一个模块传递的数据量
  - 一个模块施加到另一个模块的控制的数量
  - 模块之间接口的复杂程度
    - 整数，数组，控制信号.....



北京大学



# 耦合 (3)

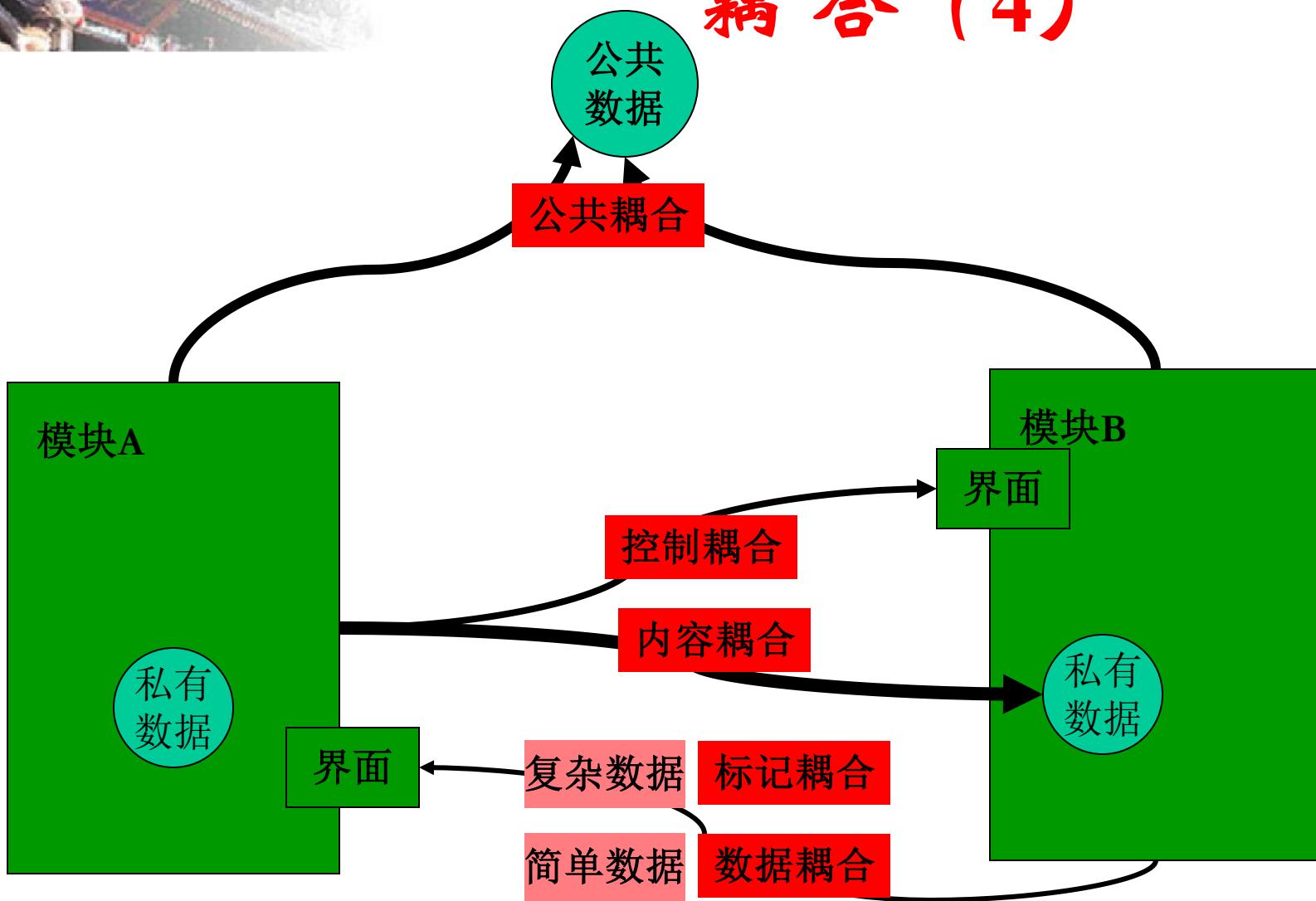
- 耦合类型：(由强到弱)

- 内容耦合：一个模块直接修改或操作另一个模块的数据。
- 公共耦合：两个以上的模块共同引用一个全局数据项。
- 控制耦合：一个模块向另一模块传递一个控制信号，接受信号的模块将依据该信号值进行必要的活动。
- 标记耦合：两个模块至少有一个通过界面传递的公共参数，包含内部结构，如数组、字符串等。
- 数据耦合：模块间通过参数传递基本类型的数据。



北京大学

# 耦合 (4)



北京大学



# 耦合 (5)

- 原则：

如果模块间必须存在耦合，就尽量使用数据耦合，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。



北京大学



# 内聚 (1)

- 定义：一个模块之内各成分之间相互依赖程度的度量。
- 好的设计满足：
  - 模块的功能单一
  - 模块的各部分都和模块的功能直接相关
  - 高内聚



北京大学



# 内聚 (2)

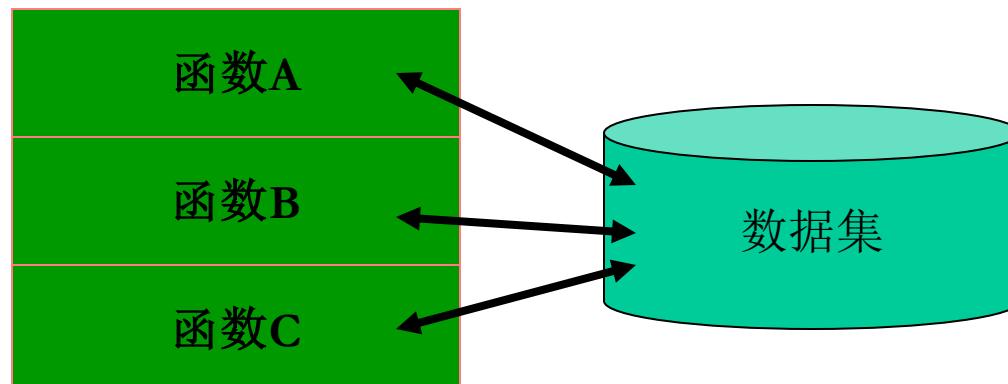
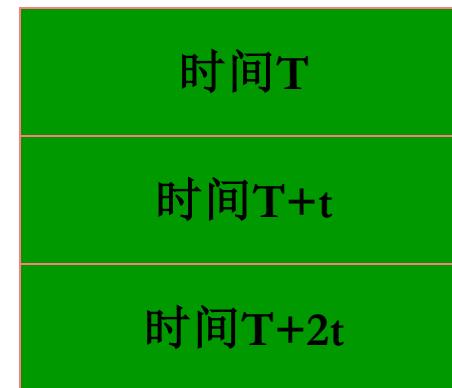
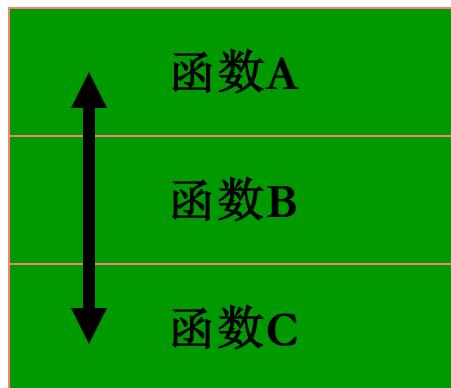
- 内聚类型：(由低到高)

- 偶然内聚：一个模块之内各成分之间没有任何关系。
- 逻辑内聚：几个逻辑上相关功能放在同一模块中。
- 时间内聚：一个模块完成的功能必须在同一时间内完成，而这些功能只是因为时间因素关联在一起。
- 过程内聚：处理成分必须以特定的次序执行。
- 通信内聚：各成分都操作在同一数据集或生成同一数据集。
- 顺序内聚：各成分与一个功能相关，且一个成分的输出作为另一成分的输入。
- 功能内聚：模块的所有成分对完成单一功能是最基本的，且该模块对完成这一功能而言是充分必要的。



北京大学

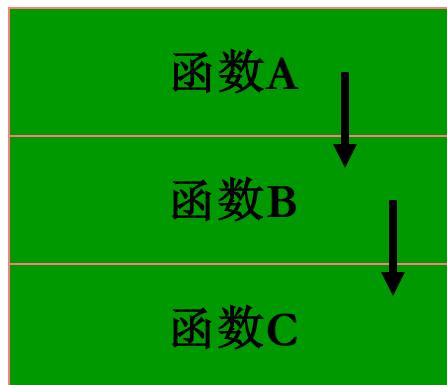
# 内聚 (3)



北京大学



# 内聚 (4)



函数A——处理1

函数B——处理2

函数C——处理3

顺序内聚：

一个部分的输出作为下一部分的输入

功能内聚：

充分而必要的功能



北京大学



# 启发式规则 (1)

- 什么叫做“启发式”?

- 根据设计准则，从长期的软件开发实践中，总结出来的规则。
- 既不是设计目标，也不是设计时应该普遍遵循的原理

- 常见的六种启发式规则

- 改进软件结构，提高模块独立性；
- 模块规模适中-每页60行语句；
- 深度、宽度、扇入和扇出适中；
- 模块的作用域力争在控制域之内；
- 降低模块接口的复杂性；
- 模块功能应该可以预测。



北京大学



## 改进软件结构，提高模块独立性

- 通过模块的分解和合并，力求降低耦合，提高内聚。
  - 例：多个模块公用的子功能可以独立形成一个模块，供这些模块调用。



北京大学



模块规模适中，每页60行语句

- 模块最好能够写在一页纸内（60行）
  - 心理学研究表明：模块语句>30之后，可理解性迅速下降。
- 方法
  - 进一步分解过大的模块
  - 将频繁调用的小模块合并到上级模块中



北京大学

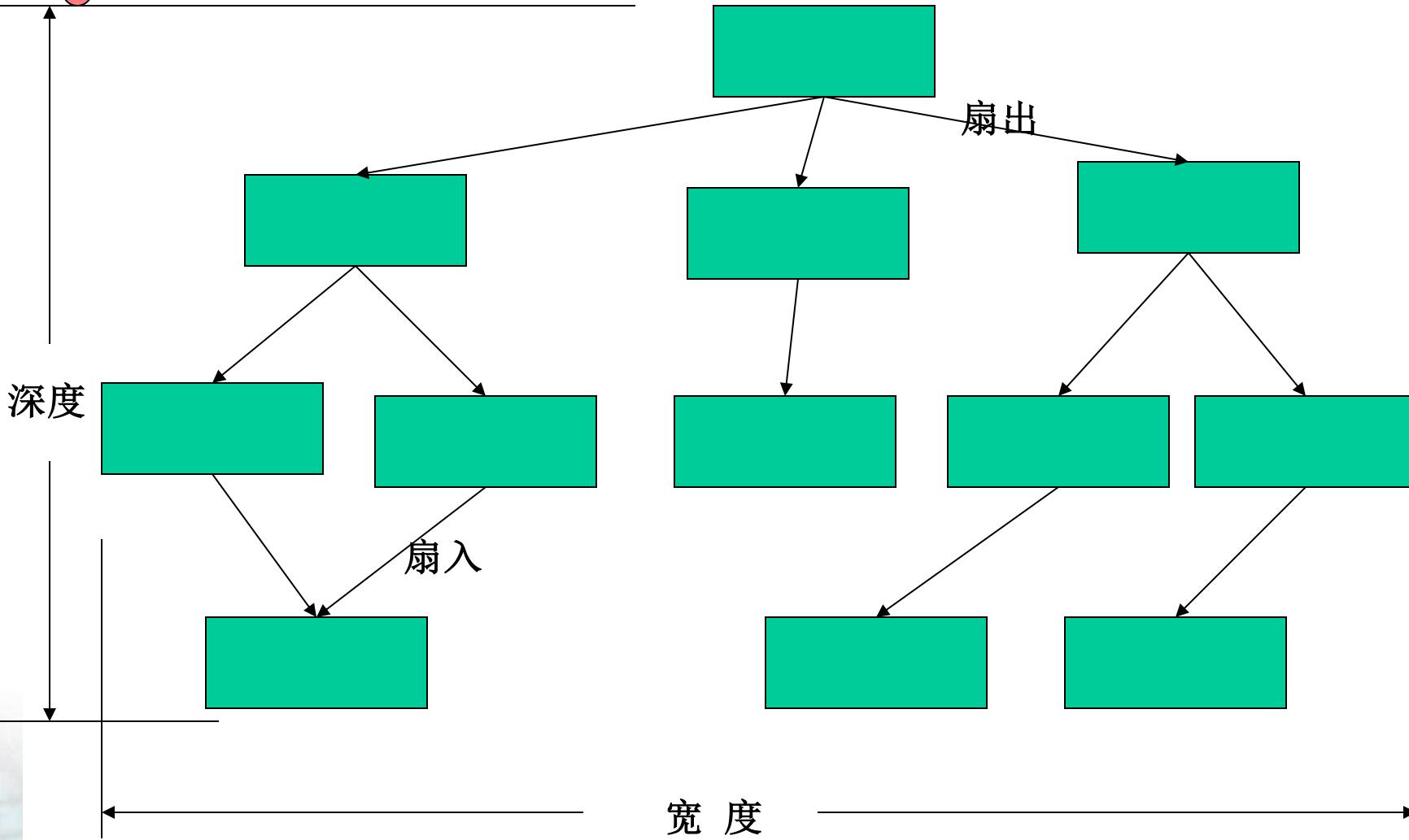
# 深度、宽度、扇入和扇出适中

- **深度：软件结构中的控制的层数**
  - 标示一个系统的大小和复杂程度
- **宽度：软件结构中同一个层次上的模块总数的最大值**
  - 宽度越大的系统越复杂
- **扇入：表示有多少个上级模块直接调用它**
  - 一般而言，扇入越大说明共享该模块的上级模块越多
  - 不违背模块独立性的条件下，扇入越大越好
- **扇出：一个模块直接控制（调用）的下级模块数目**
  - 扇出过大意味着模块过分复杂
  - 扇出过小意味功能过度集中
  - 典型的3或者4(上限5-9)
  - 好的系统：顶层扇出高，中层扇出少，底层扇入高，系统呈“葫芦”型



北京大学

# 深度、宽度、扇入和扇出适中



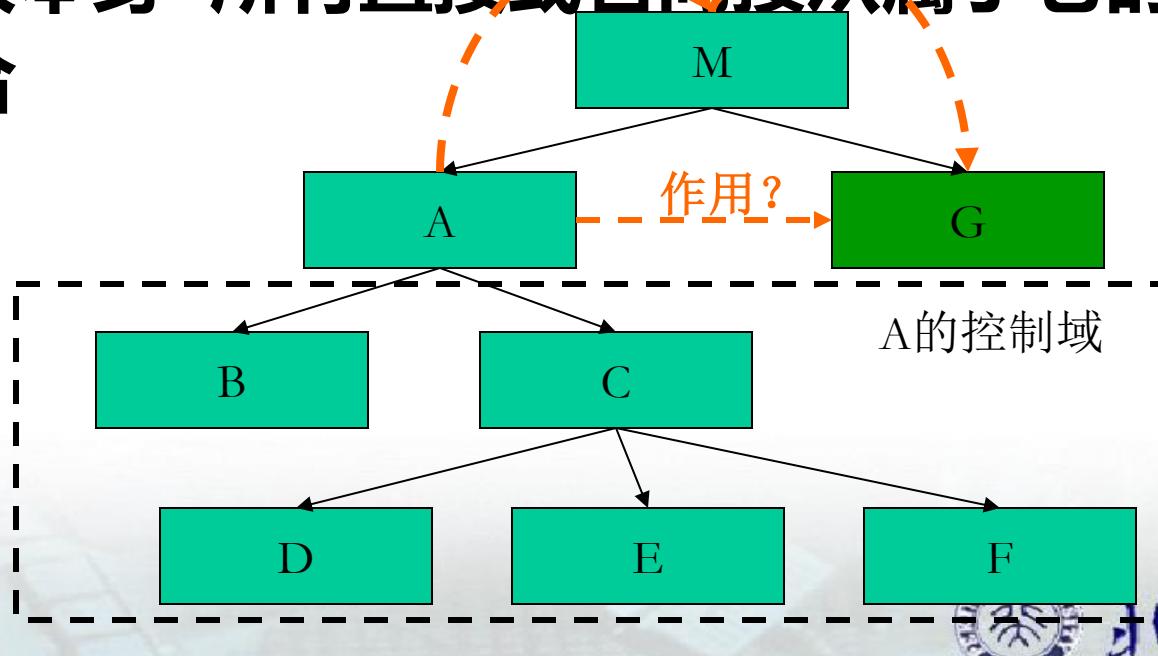
# 模块的作用域力争在控制域之内

- **作用域**

- 受该模块内一个判定影响的所有模块的集合

- **控制域**

- 模块本身+所有直接或者间接从属于它的模块的集合



# 降低模块接口的复杂性

- 使得信息传递简单并且和模块的功能一致
- 接口复杂或不一致往往导致紧耦合和低内聚
- 例子：求 $A x^2 + B x + C = 0$ 的根
  - QUAD-ROOT(TBL,X)
    - 数组TBL传送方程系数，数组X回送求得的根
  - QUAD-ROOT(A,B,C,Root1,Root2)



北京大学

# 模块功能应该可以预测

- 什么叫做“功能可以预测”？

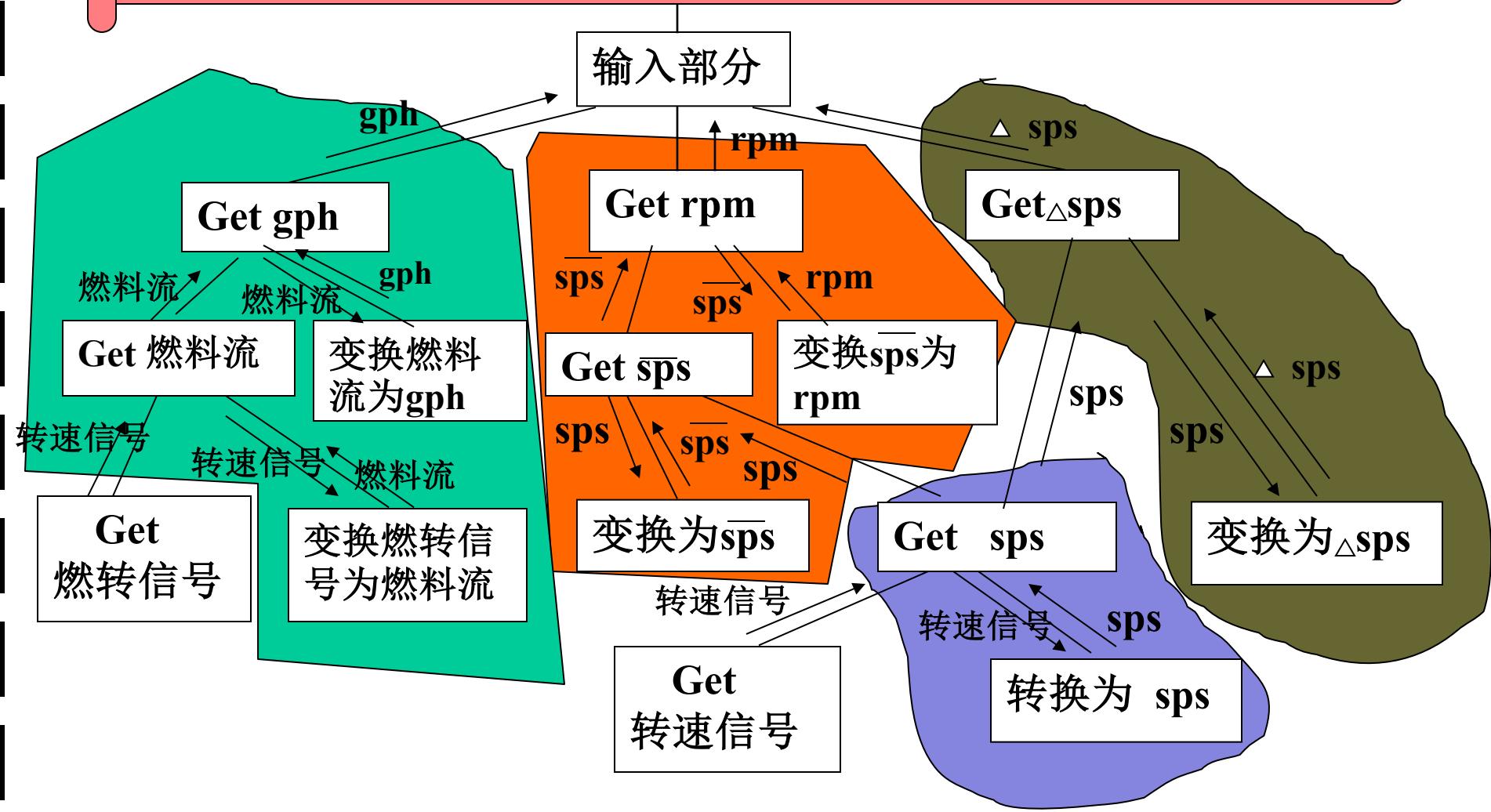


- 什么样的模块功能不可预测？
  - 模块带有内部状态→输出取决于该状态



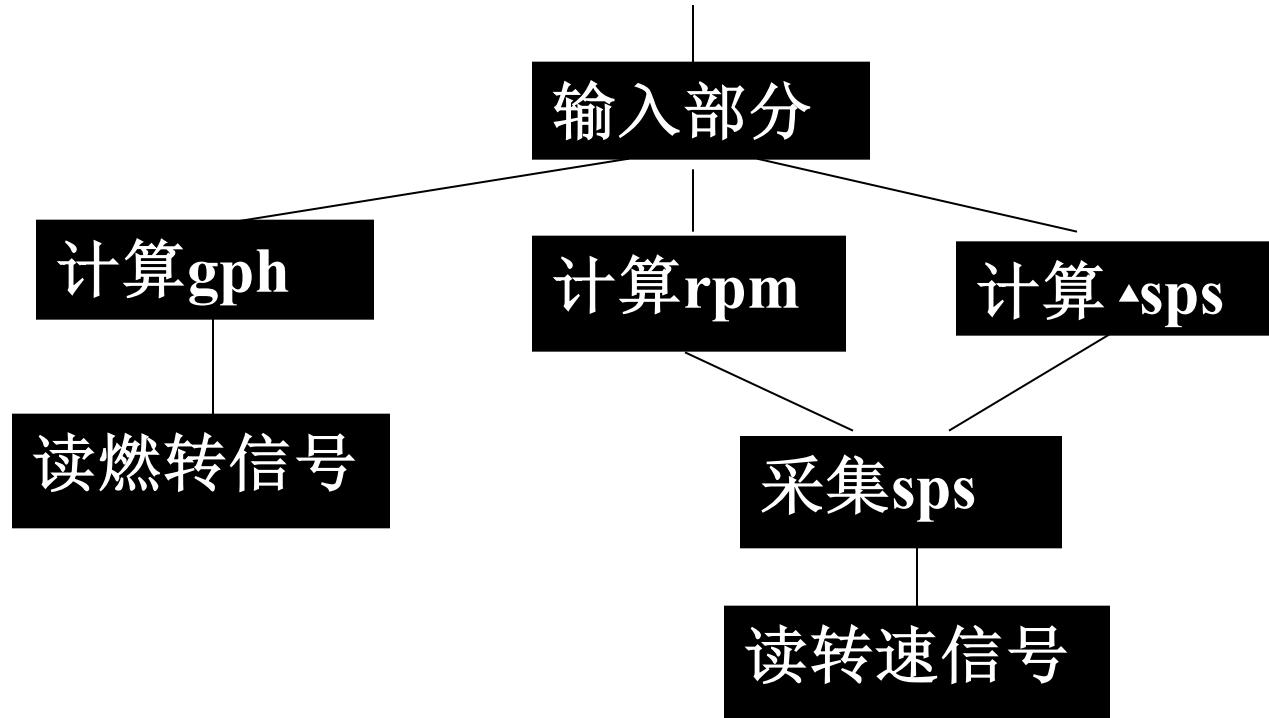
北京大学

# 改进软件结构，提高模块独立性





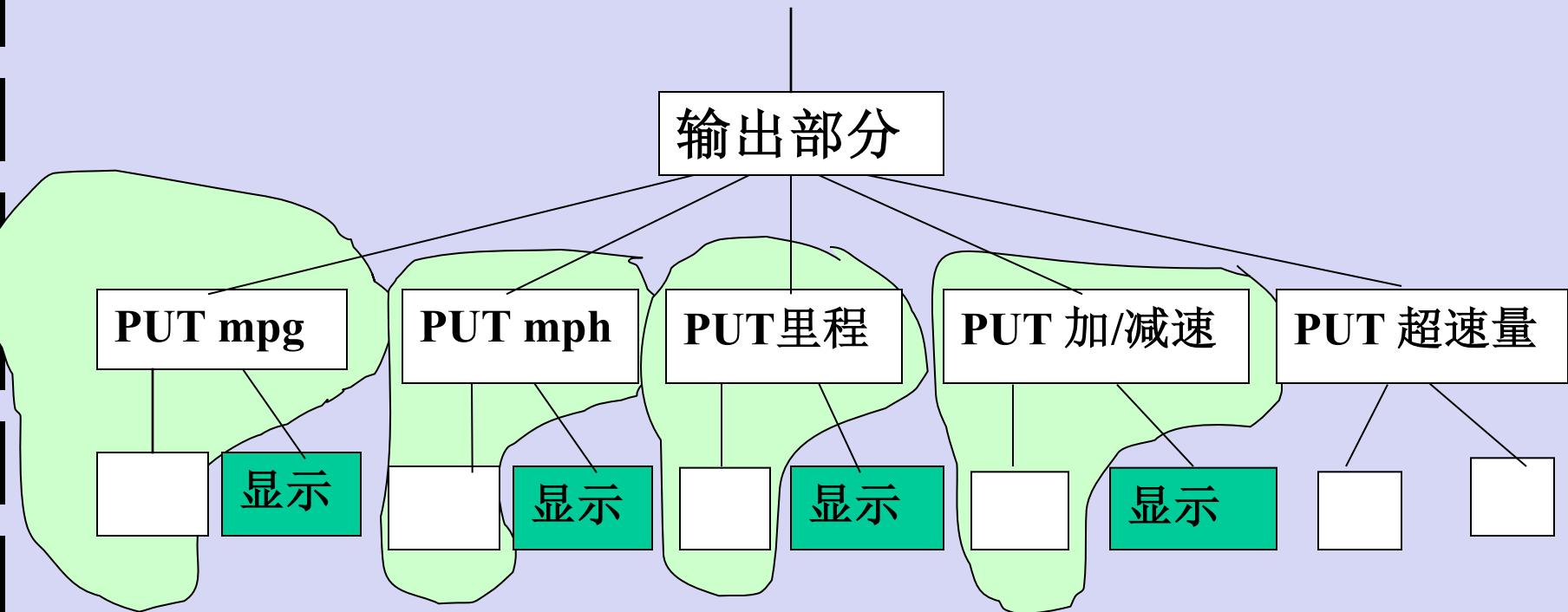
# 输入部分的精化



其中：sps为转速的每秒信号量； $\bar{sps}$ 为sps的平均值； $\Delta s_{\text{sp}}$ 为sps的瞬时变化值；rpm为每分钟转速；mph为每小时英里数；gph为每小时燃烧的燃料加仑数；rpm为行进里程。

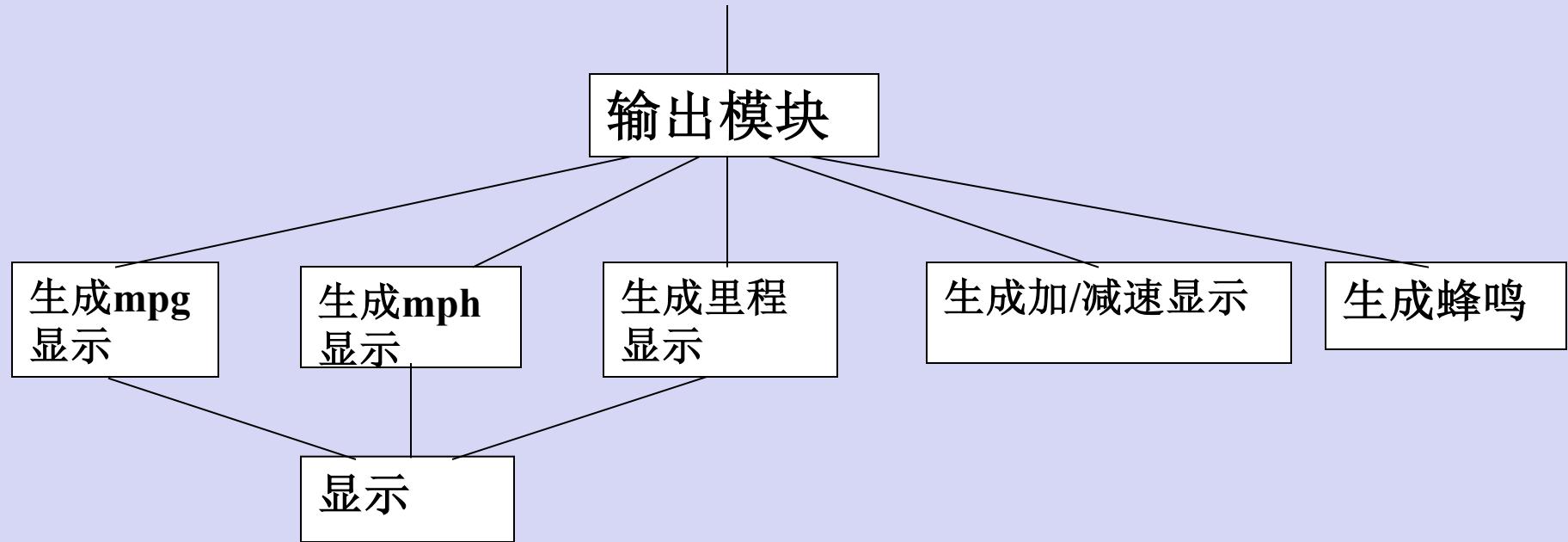
输出部分进一步精化

把相同或类似的物理输出合并为一个模块，  
以减少模块之间的关联。





# 输出部分的精化





# 变换部分的精化

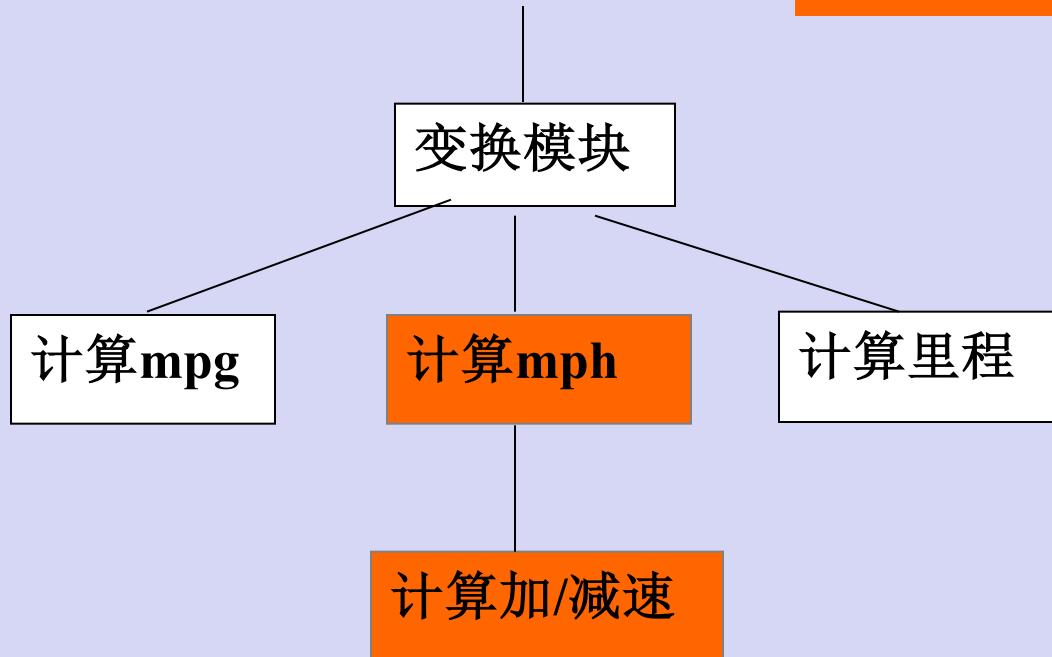
- 对于变换部分的求精，是一项具有挑战性的工作。其中主要是根据设计准则，并要通过实践，不断地总结**经验**，才能设计出合理的模块结构。



北京大学



# 变换部分的精化



把“确定加/减速”的模块放在  
“计算速度mph”模块下面，  
则可以减少模块之间的关联，  
提高模块的独立性。



# 总体设计总结

- 将一个给定的DFD转换为初始的模块结构图
  - 基本上是一个“机械”的过程
  - 一般体现不了设计人员的创造力
  
- 优化设计
  - 将一个初始的模块结构图转换为最终的模块结构图
  - 对设计人员将是一种挑战
  - 其结果将直接影响软件系统开发的质量。



北京大学