哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

软件工程
第四章　软件设计
12 结构化软件设计方法

徐汉川
xhc@hit.edu.cn

2013年10月8日

主要内容

12.1 结构化设计方法概述

12.2 概要设计与结构图**(SC)**

12.3 两种典型形态的**SC**

12.4* DFD→SC的映射方法

12.5* 结构图**(SC)**的优化原则

12.6* 结构化设计的核心原则：模块化

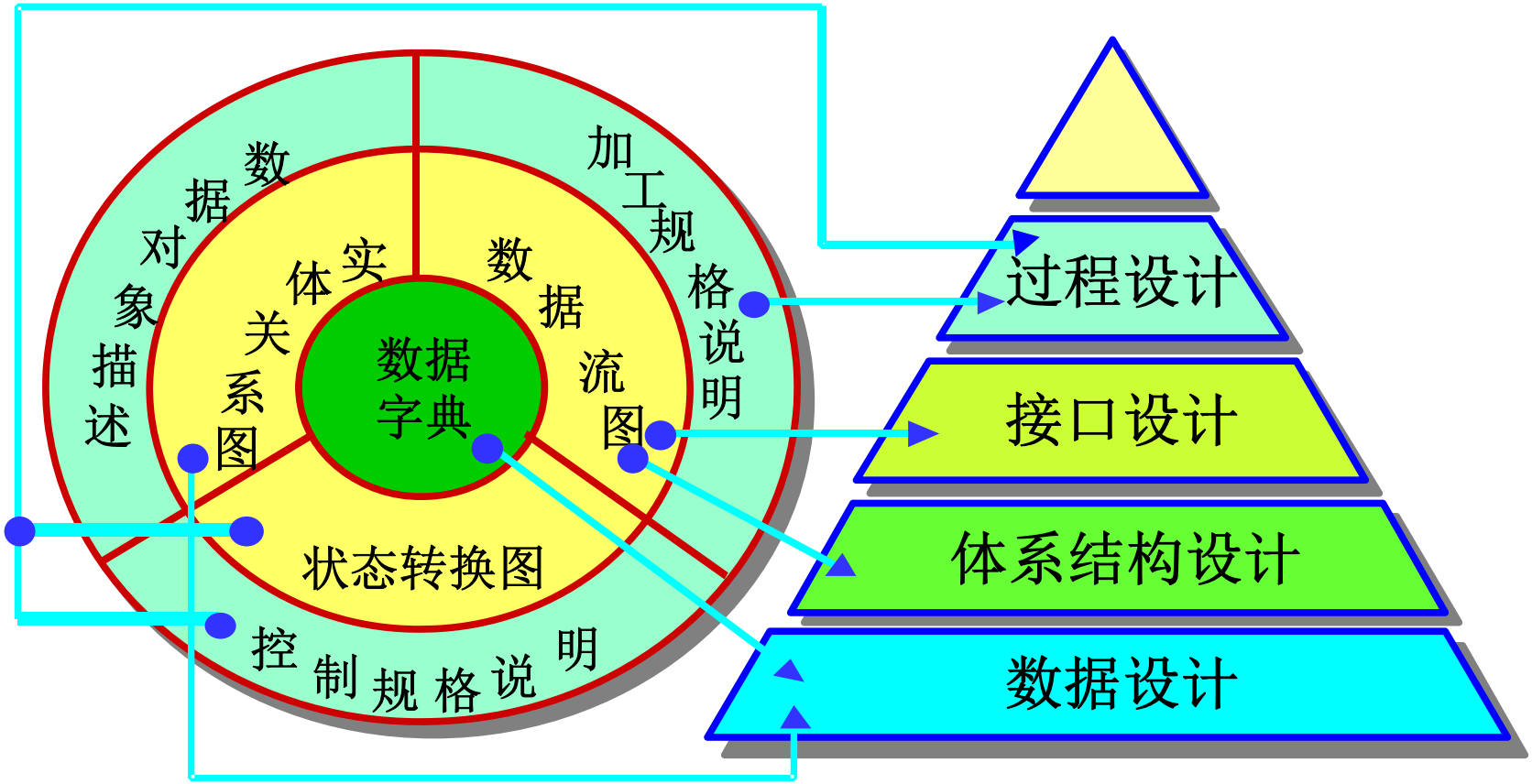
12.7* 数据设计

12.8 详细设计

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

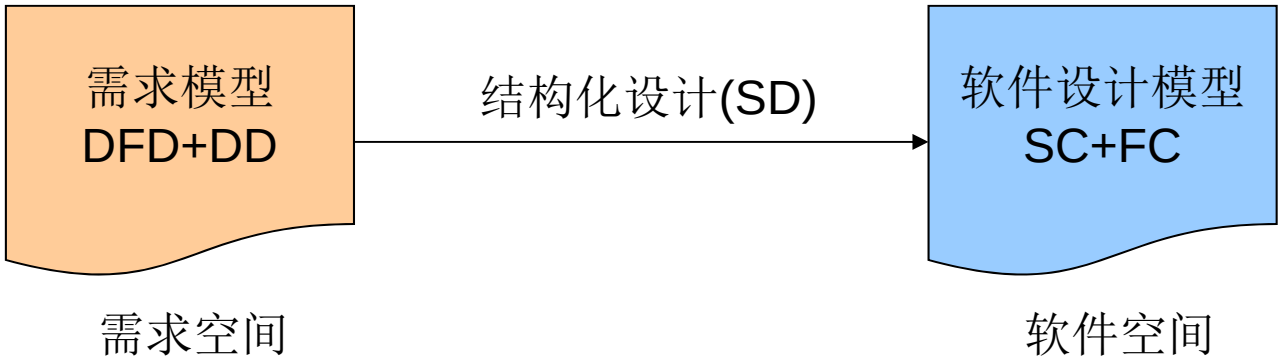
12.1 结构化设计方法概述

将结构化需求分析模型转换为设计模型



结构化设计方法

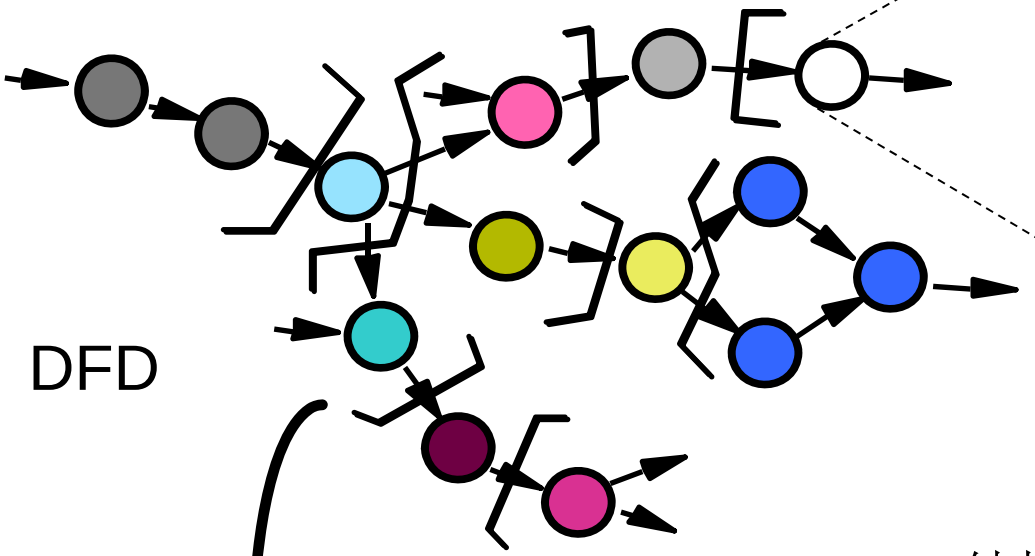
- **结构化设计(Structured Design, SD)方法：**
  - 1970年代中期提出；
  - 按照DFD的不同类型，将它们转换为相应的软件体系结构；
  - 目标：通过对模块的合理划分，得到软件的体系结构图，进而细化每个模块内部的流程；
  - 采用结构图(Structure Chart, SC)和程序流程图(Flow Chart, FC)作为描述工具。



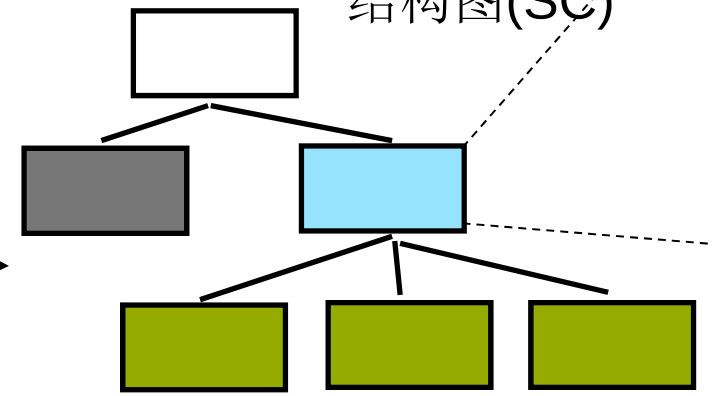
结构化设计方法

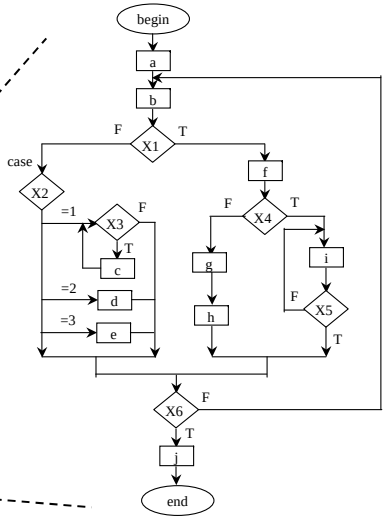
结构化语言

```
IF customer does more than $50,000 business
    THEN
                    IF the customer wasn't in debt to us the last 3 months
                    THEN discount = 15%
                ELSE (he was in debt to us)
                    IF customer has been with us for more than 20 years
            THEN discount =10%
            ELSE (20 year or less)
                    THEN discount = 5%
ELSE (customer does $50.000 OR Less)
                THEN discount = 0
```



结构图(SC)





程序流程图(FC)

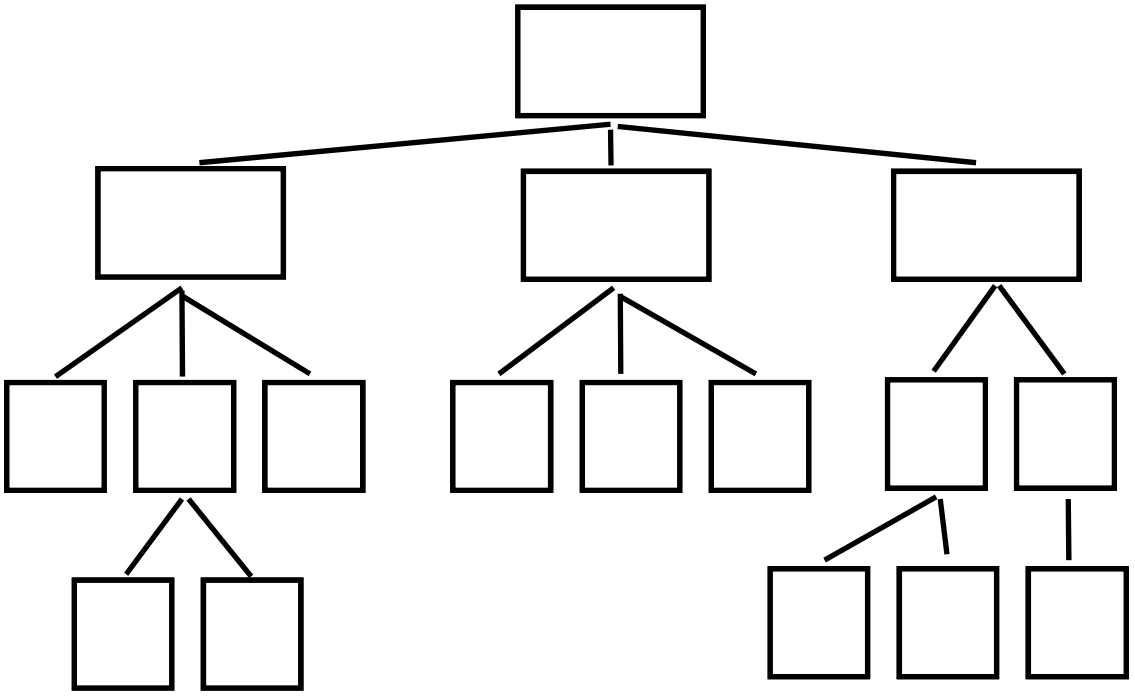
哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

12.2 概要设计与结构图(SC)

结构化概要设计

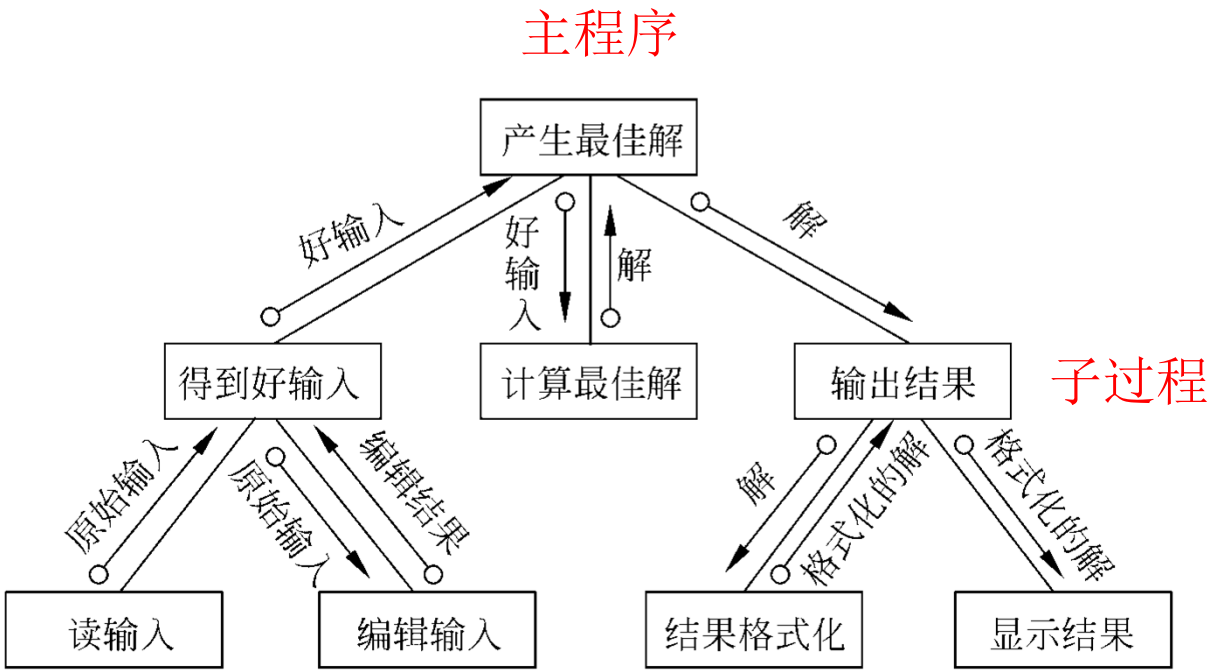
- **概要设计之一——体系结构设计**
  - 确定软件系统的体系结构：主程序-子过程风格；
  - 将系统划分为多个功能模块；
  - 确定各模块的功能与接口；
  - 确定各模块之间的调用关系；
  - 所用的模型：结构图(Structure Chart, SC)
- **概要设计之二——接口设计**
  - 确定系统与外部其他软硬件系统之间的接口；
  - 相对容易一些；
- **概要设计之三——数据设计**
  - 根据对“数据存储”的数据字典的定义，设计相应的数据库或文件。

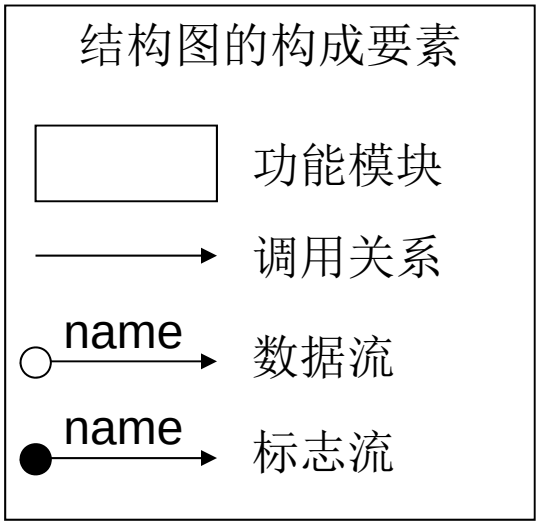
简化的结构图(SC)

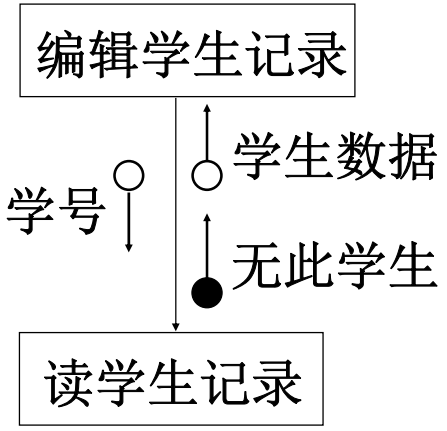


- 你对此应该很熟悉…
- 类似于“功能菜单树”
- **结构图(Structure Chart)：**用来表达软件的组成模块及其之间的调用关系。

结构图(SC)的直观印象

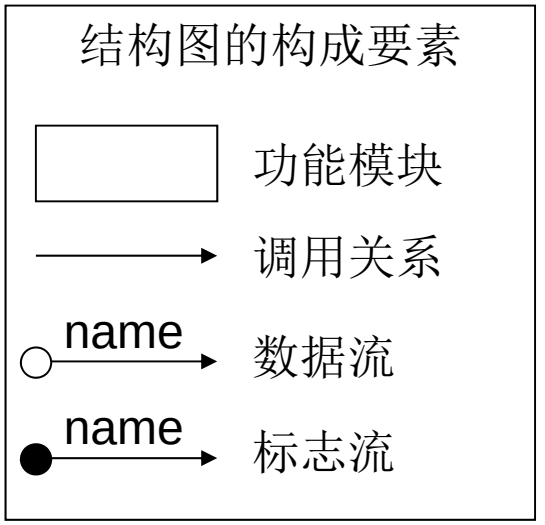

主程序
产生最佳解
好输入
好输入
解
解
得到好输入
计算最佳解
输出结果
子过程
原始输入
原始输入
编辑结果
解
格式化的解
格式化的解
读输入
编辑输入
结果格式化
显示结果
最底层函数


结构图的构成要素
功能模块
调用关系
name
数据流
name
标志流


编辑学生记录
学号
学生数据
无此学生
读学生记录

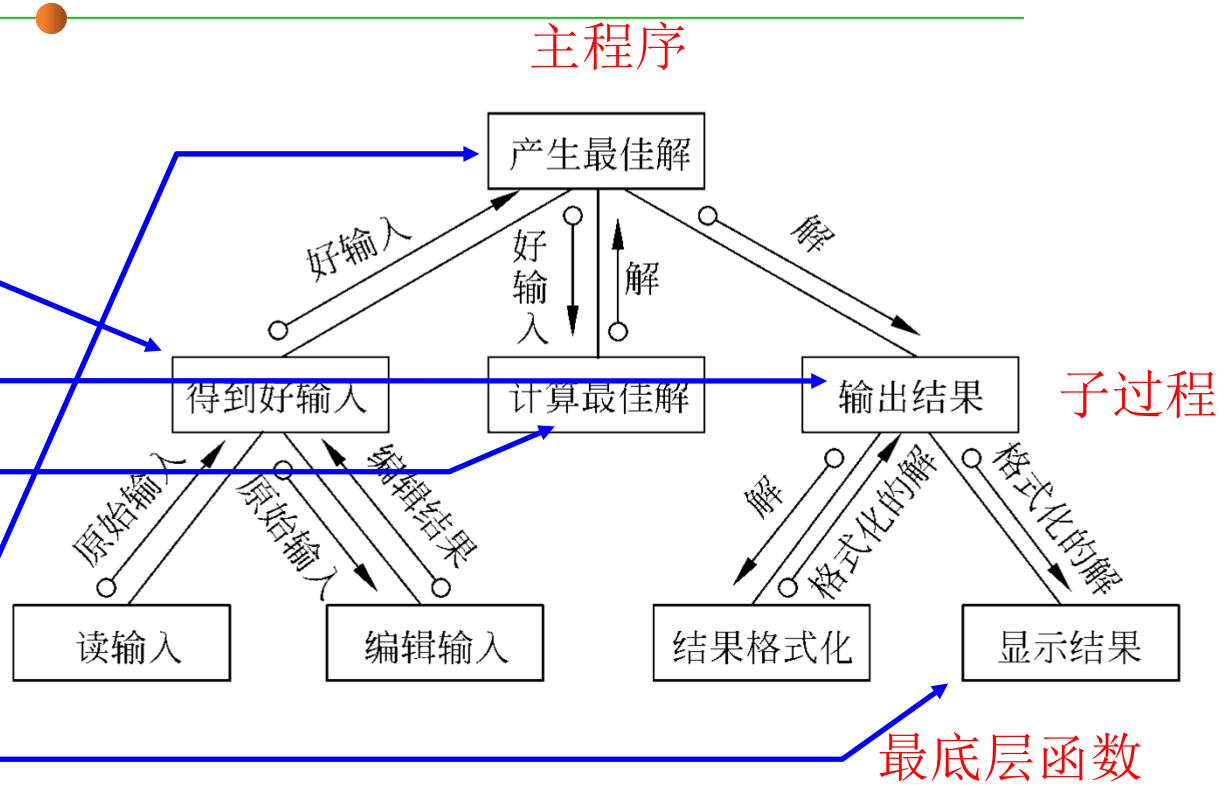
SC中的基本要素

- **模块(module)：是软件空间的术语，实现特定的功能，有明确的输入、内部处理逻辑、输出。**
  - 形式：返回值类型 函数名(参数类型1 参数名1,参数类型2 参数名2, …)
- **调用关系(invocation)：模块之间如何通过“函数调用”的方式建立连接。**
  - 形式：g() { rtn = f(p1,p2,…);}
- **数据流：模块之间相互调用时所传递的业务类数据，分为两类(输入数据-参数；输出数据-返回值)。**
  - 形式：简单数据类型、复杂数据类型(结构体)；
- **标志流：模块之间相互调用时所传递的标志性数据**
  - 形式：“Y”/“N”、 “T”/“F”、 状态标志、循环变量等；



12.2.1 结构图(SC)中的模块类型

- 传入模块
- 传出模块
- 变换模块
- 协调(控制)模块
- 源模块
- 漏模块



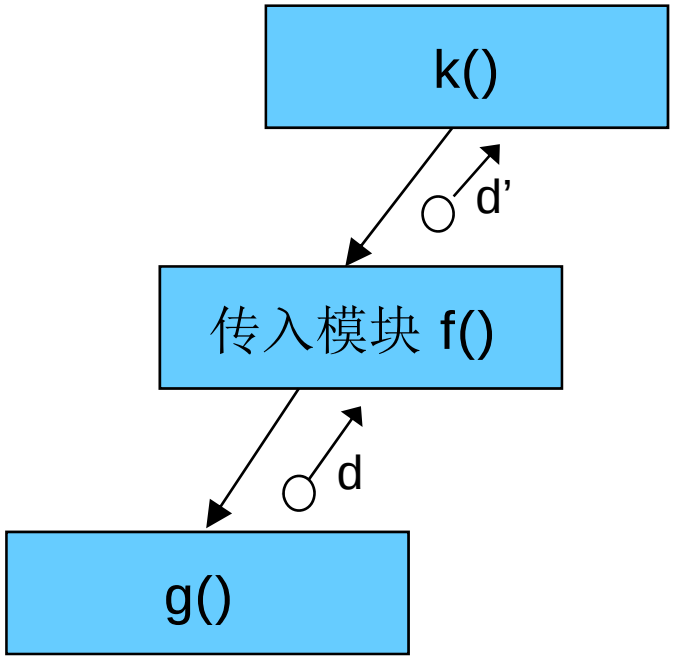
- 越底层的模块，越接近系统边界(软件使用者、其他软件、其他硬件、操作系统、文件、数据库、用户界面)，模块的粒度越小。
- 越高层的模块，越接近系统的核心处理逻辑，模块的粒度越大。

结构图(SC)中的模块类型(1)：传入模块

- **传入模块：**
  - 从下属模块取得数据，经过某些处理，再将其传送给上级模块。
  - 它传送的数据流叫做逻辑输入数据流。

```
void k () {
   ...
   int d' = f();
   ...
}

int f() {
   ...
   int d = g();
   int d' = HelloWorld (d);
   return d'
}
```

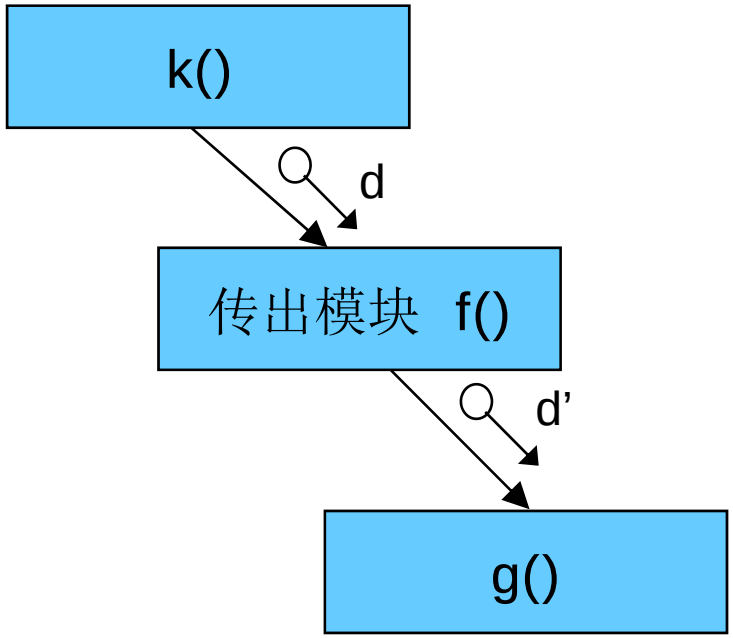


结构图(SC)中的模块类型(2)：传出模块

- **传出模块：**
  - 从上级模块获得数据，进行某些处理，再将其传送给下属模块。
  - 它传送的数据流叫做逻辑输出数据流。

```
void k () {
   ...
   int d = …;
   f(d);
   ...
}

void f(int d) {
   ...
   int d' = HelloWorld (d);
   g(d’);
   …
}
```

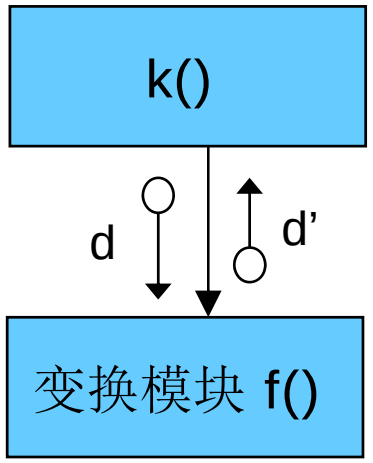


结构图(SC)中的模块类型(3)：变换模块

- 变换模块：
  - 从上级模块取得数据，进行特定的处理，转换成其它形式，再传送回上级模块。
  - 它加工的数据流叫做变换数据流。

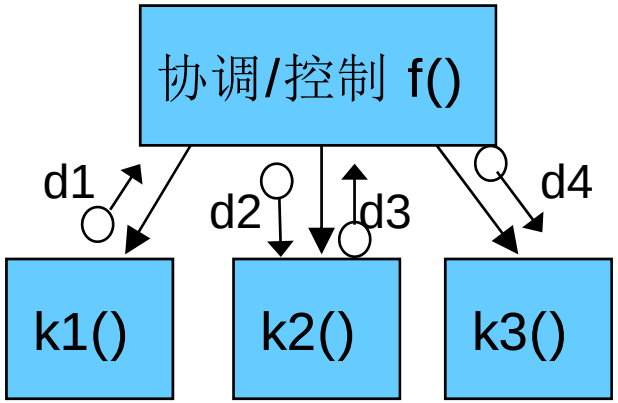
```
void k () {
   ...
   int d = …;
   int d’ = f(d);
   ...
}

int f(int d) {
   ...
   int d' = HelloWorld (d);
   return d’;
}
```



结构图(SC)中的模块类型(4)：协调(控制)模块

- **协调(控制)模块：**
  - 对所有下属模块进行协调和管理。

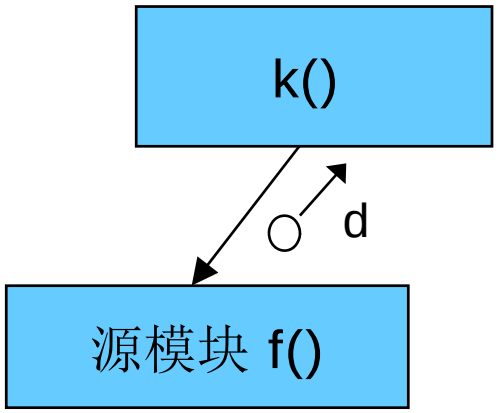



```
void f () {
   ...
   int d1 = k1();
   int d2 = HelloWorld1 (d1);
   int d3 = k2(d2);
   int d4 = HelloWorld2 (d2, d3);
   k3(d4);
   ...
}
```

结构图(SC)中的模块类型(5)：源模型和漏模块

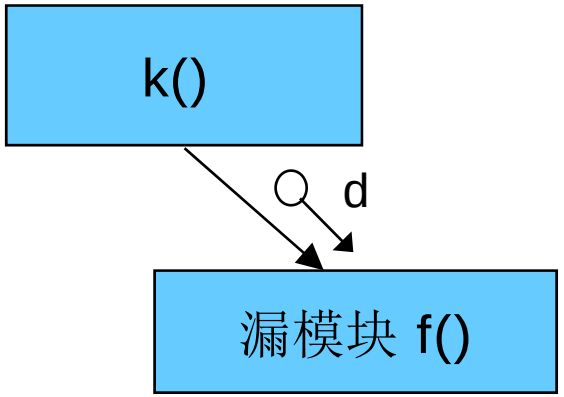
- **源模块：**
  - 不调用其他模块的传入模块，只适用于传入部分的开始端；

```
void k () {
   ...
   int d = f();
   ...
}
```



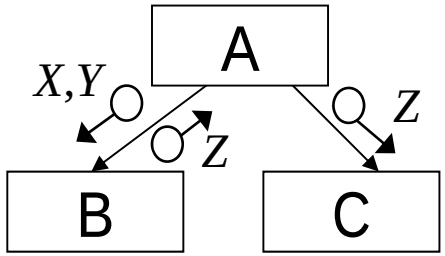
- **漏模块：**
  - 不调用其它模块的传出模块，只适用于传出部分的末端。

```
void k () {
   ...
   f(d);
   ...
}
```

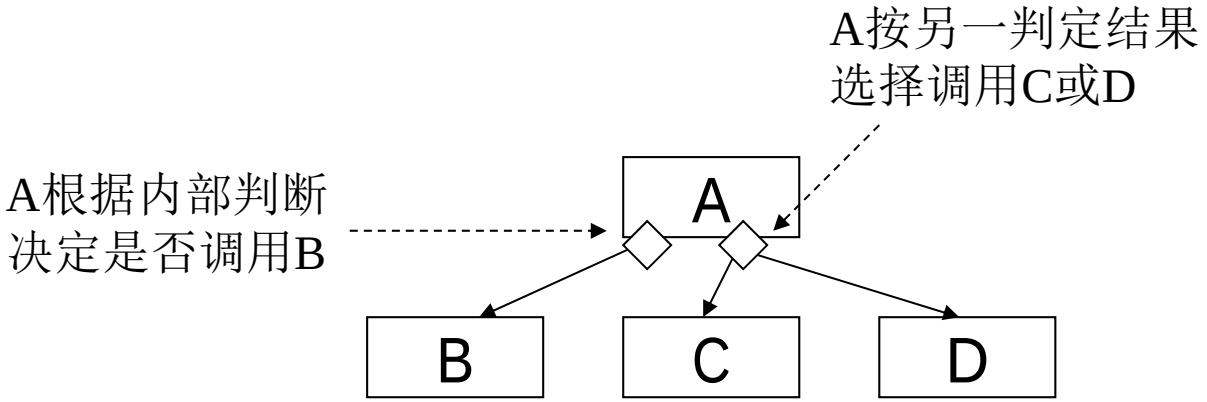


12.2.2 SC图中的模块调用

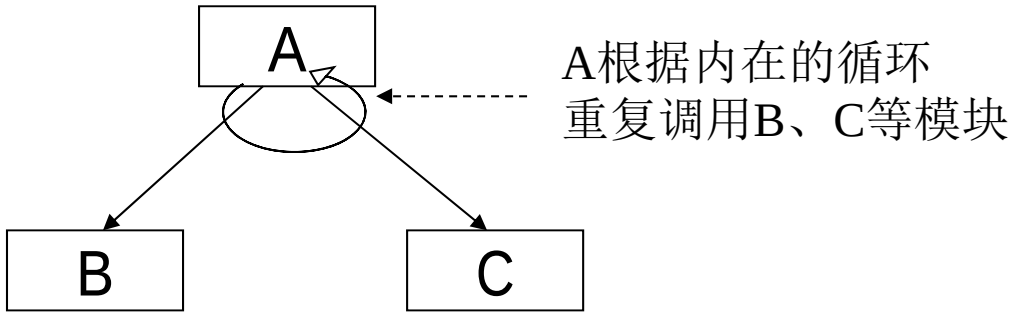
- 简单调用



- 选择调用



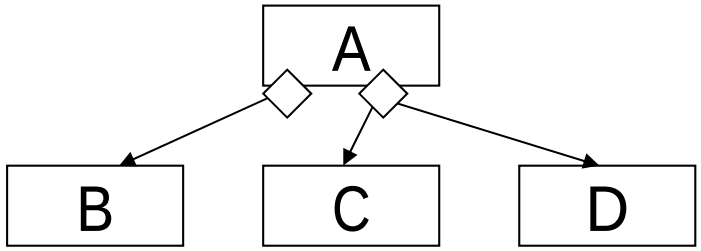
- 循环调用



SC图中的模块调用

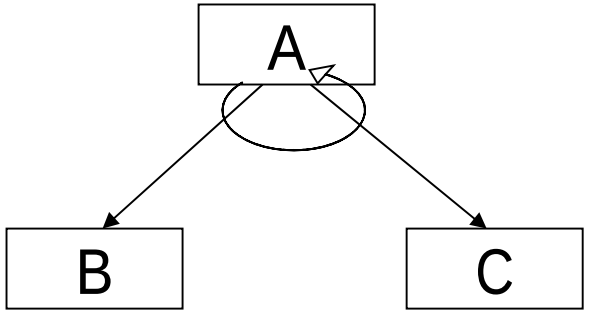
- 选择调用

```
void A (){
  if (condition1 == TRUE)
        B();
  if (condition2 == TRUE)
        C();
  else
        D();
}
```



- 循环调用

```
void A (){
  while (condition == TRUE) {
      B();
      C();
  }
}
```

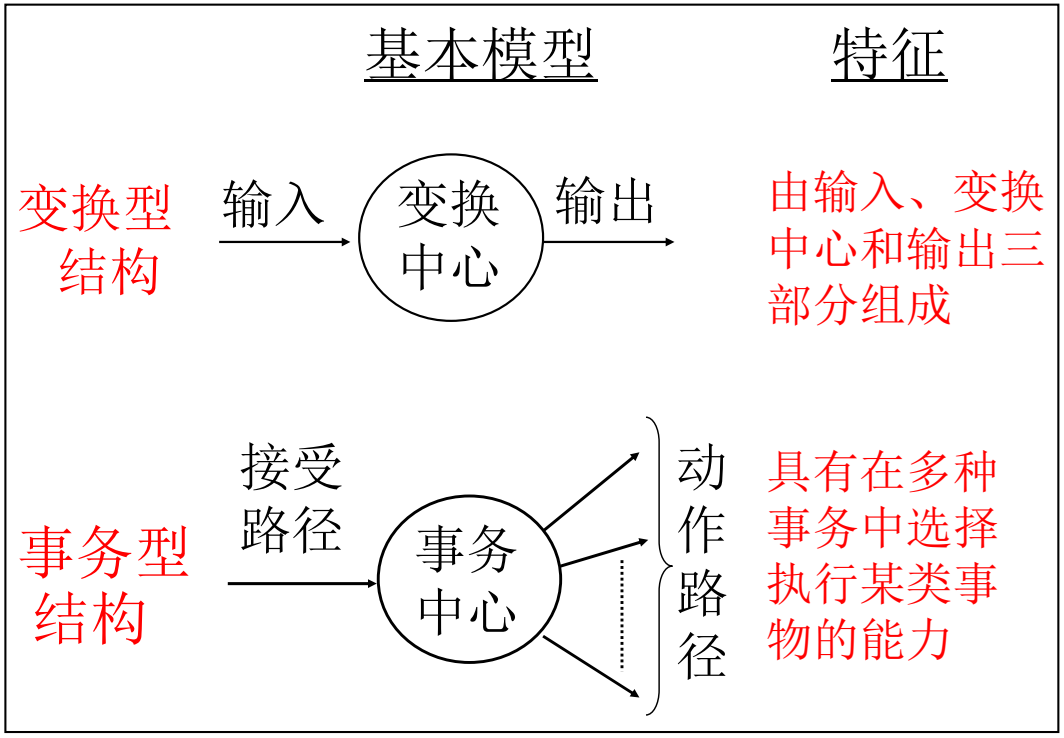


哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

12.3 两种典型形态的SC

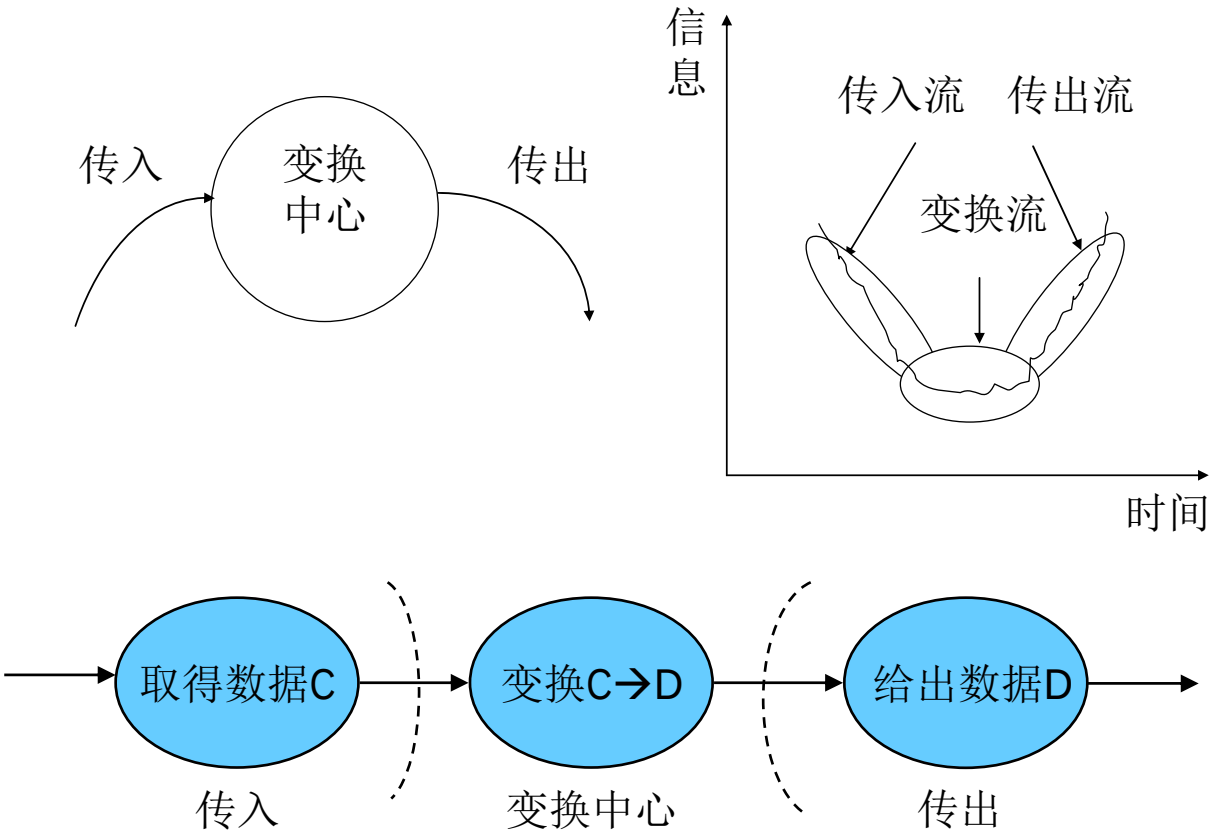
12.3.1 DFD的两种类型

- 为了实现从DFD到SC的映射(需求模型→设计模型)，需要仔细区分DFD中数据流的性质，并分别学习相应的映射方法。
- 两种类型的数据流：
  - 变换型
  - 事务型

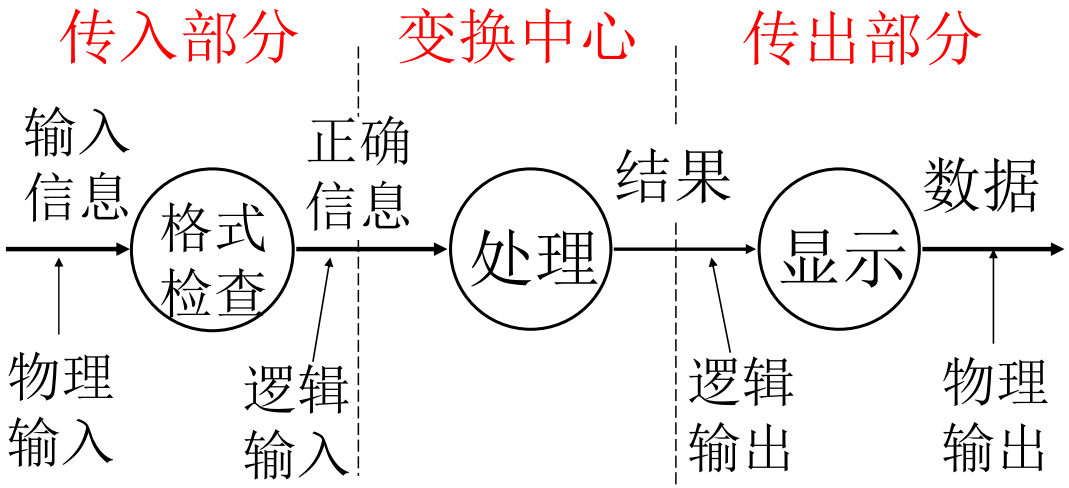


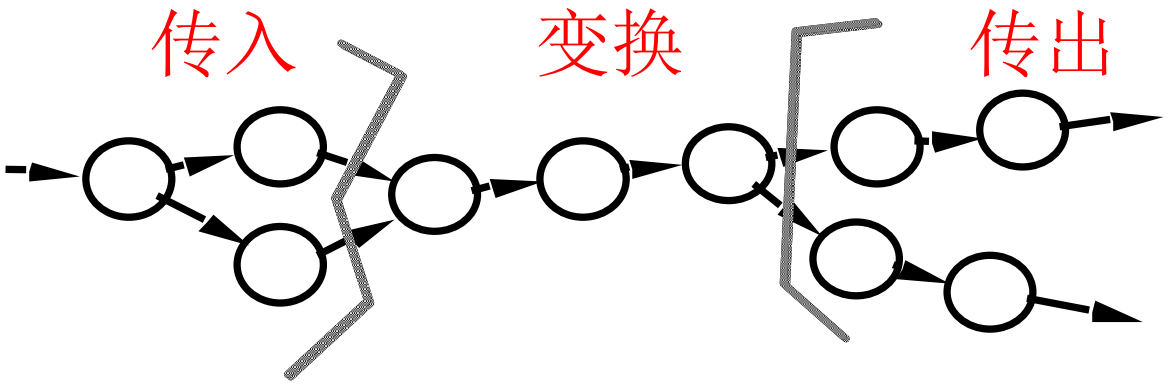
12.3.2 变换型结构

- 变换型结构：工作过程大致分为三步，即取得数据，变换数据和给出数据。

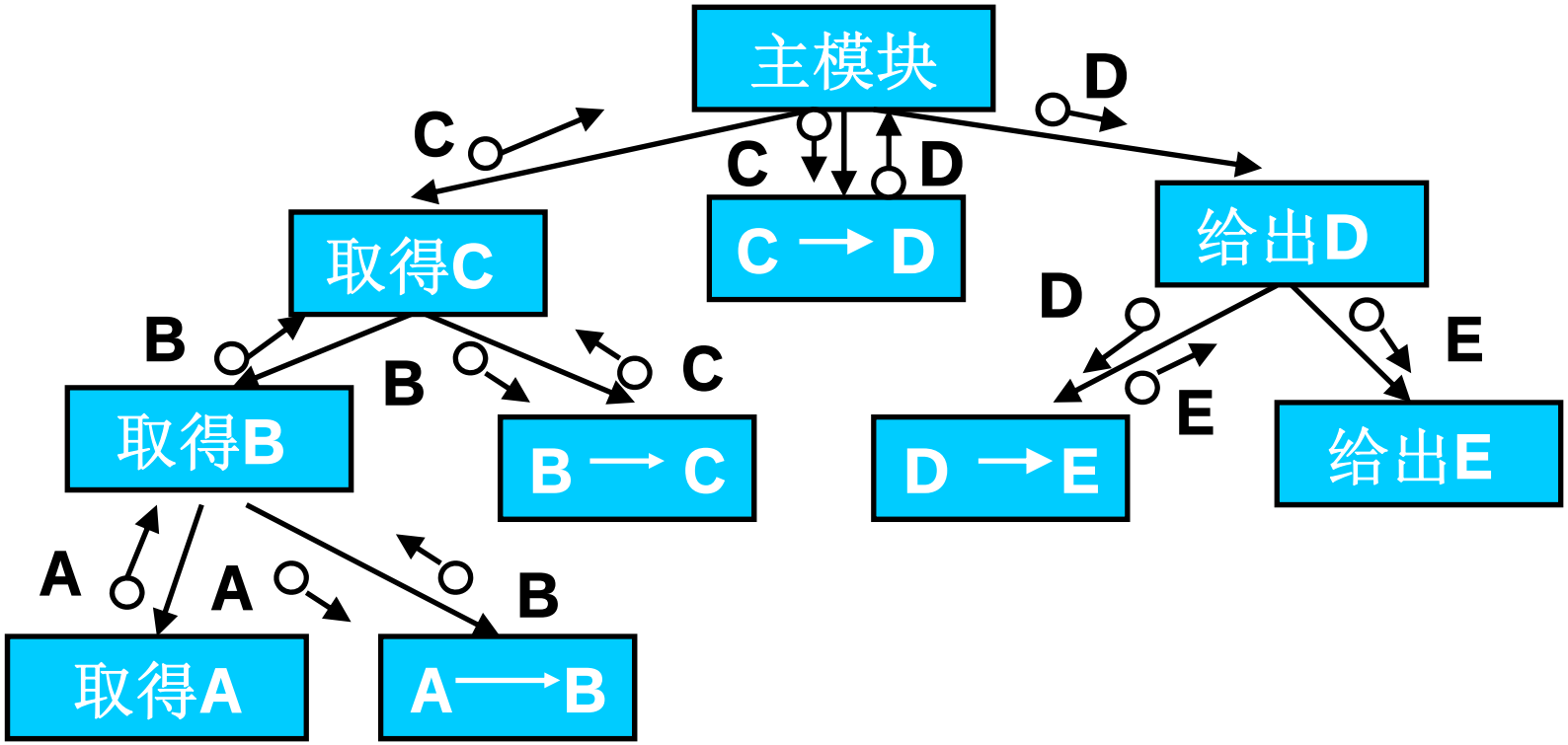


变换型结构


传入部分
变换中心
传出部分
输入
信息
格式
检查
正确
信息
处理
结果
显示
数据
物理
输入
逻辑
输入
逻辑
输出
物理
输出

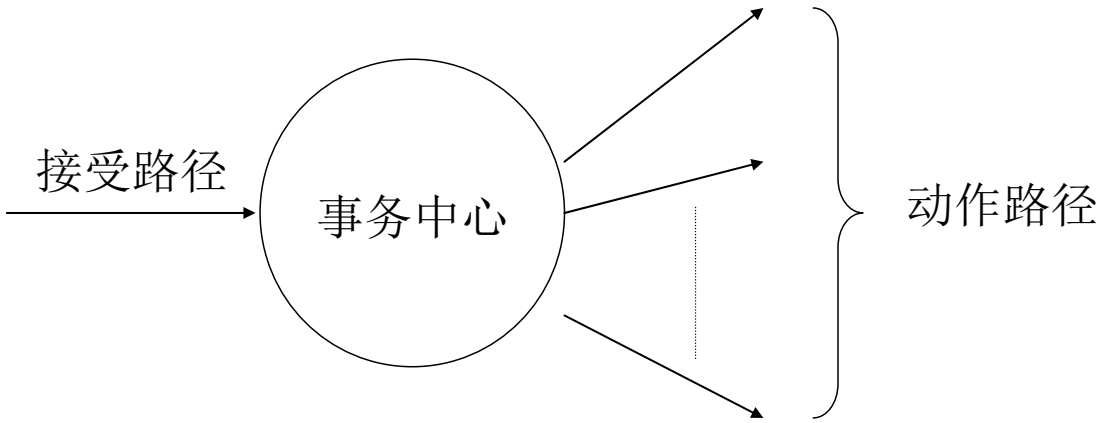

传入
变换
传出

变换型DFD所对应的SC

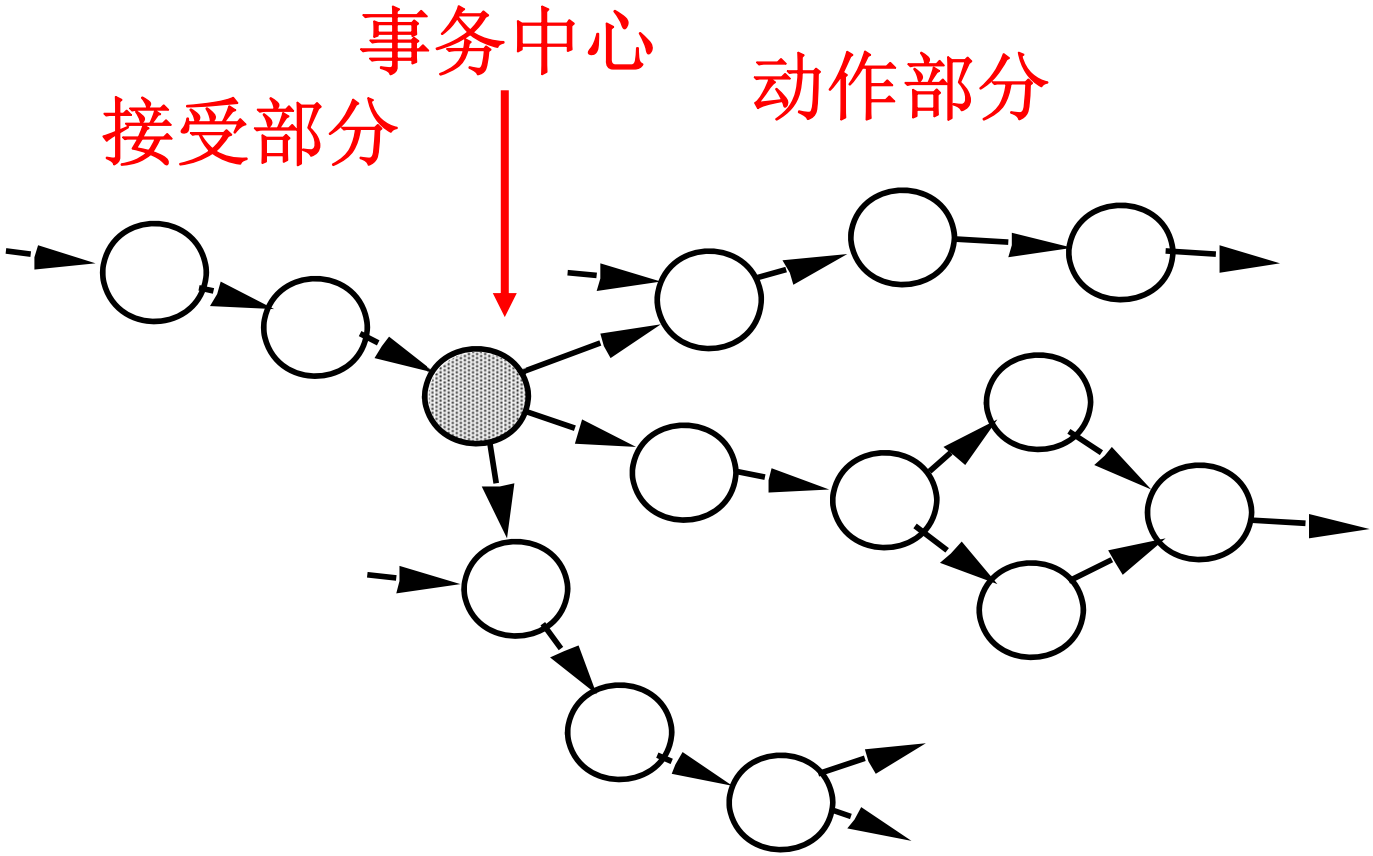


12.3.3 事务型结构

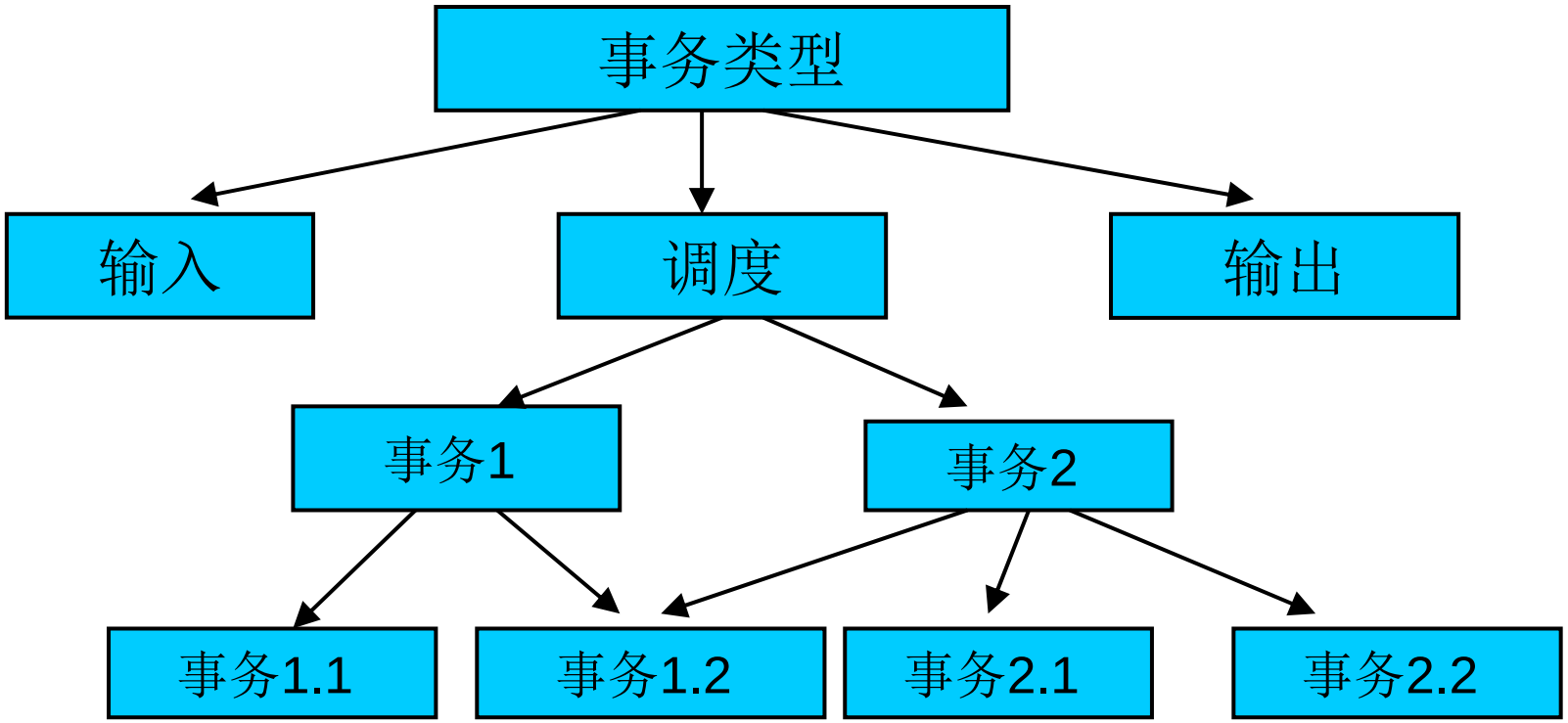
- **事务型结构：接受一项事务，根据事务处理的特点和性质，选择分派一个适当的处理单元，然后给出结果。**
  - 由至少一条接受路径(reception path)、一个事务中心(transaction center)和若干条动作路径(action path)组成。



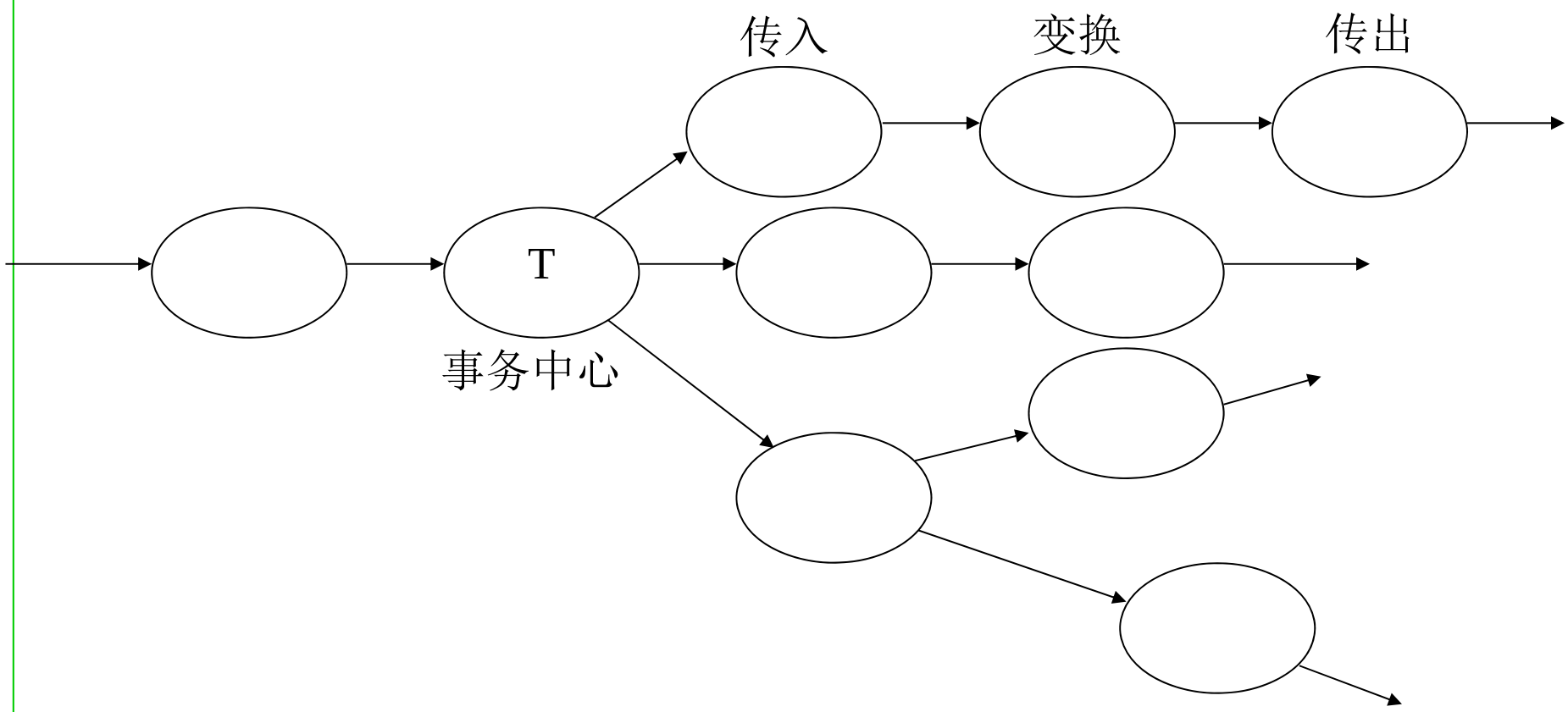
事务型结构



事务型DFD所对应的SC



12.3.4 事务型+变换型结构

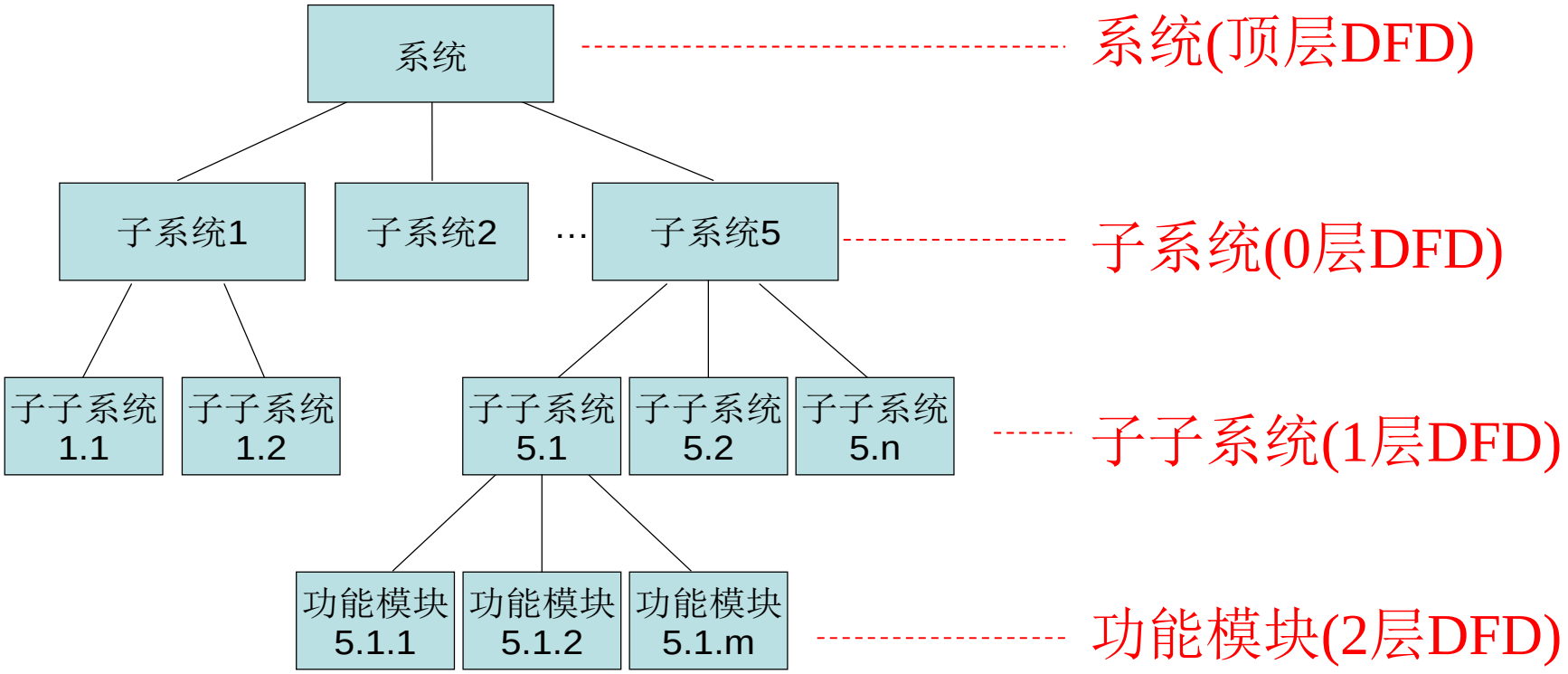


12.4* DFD→SC的映射方法

DFD→SC的映射方法

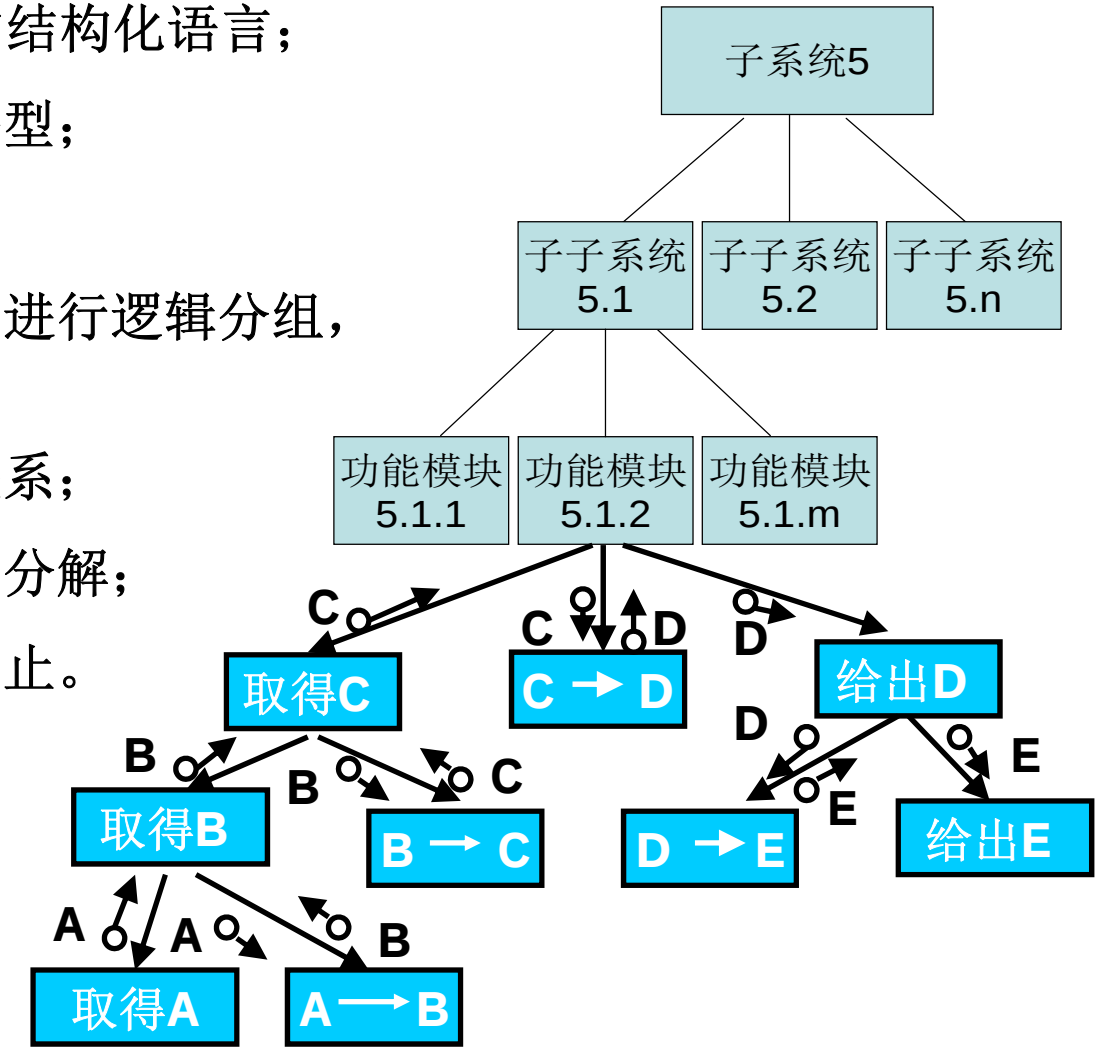
- 如何从需求分析阶段的**DFD**出发，进行**SC**的设计？
- 两大步骤：
  - Step 1：针对顶层DFD和各中间层DFD，将其中的“加工”识别出来，形成SC的上层结构(也称为“菜单树”)；
  - Step 2：针对最底层DFD，按照事务型或变换型的结构，设计SC的下层结构(面向程序实现的“函数”)。

Step 1：SC的上层结构(菜单树)



Step 2：SC的下层结构(函数)

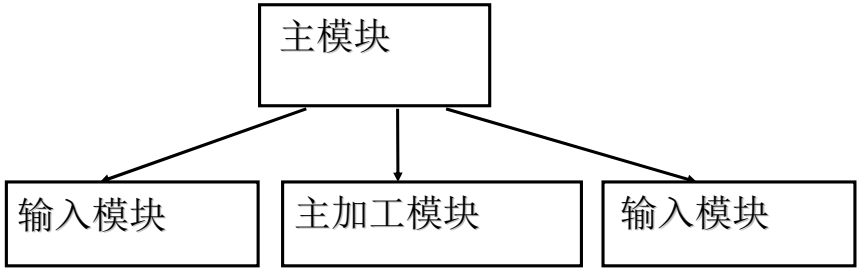
- 分析底层**DFD**中各个加工的结构化语言；
- 判断其类型：变换型、事务型；
- 将结构化语言中的各条语句进行逻辑分组，识别出一组“函数”；
- 描述这些函数之间的调用关系；
- 若函数仍嫌过大，对其继续分解；
- 一直分解到“系统边界”为止。



Step 2：SC的下层结构(函数)

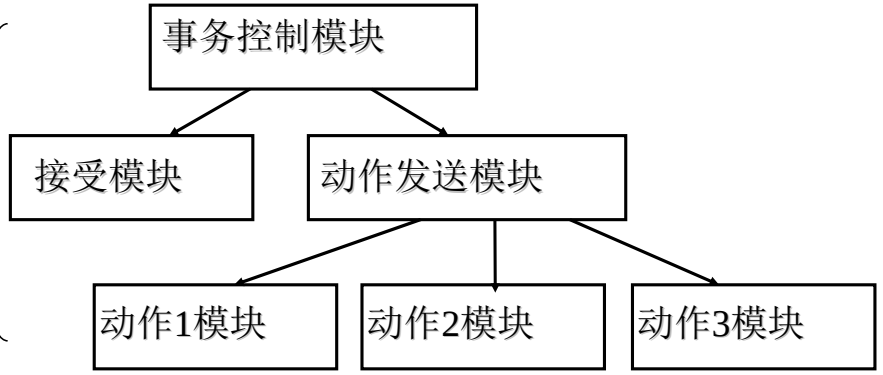
变换型DFD —变换映射→ 初始SC

由变换映射产生



事务型DFD —事务映射→ 初始SC

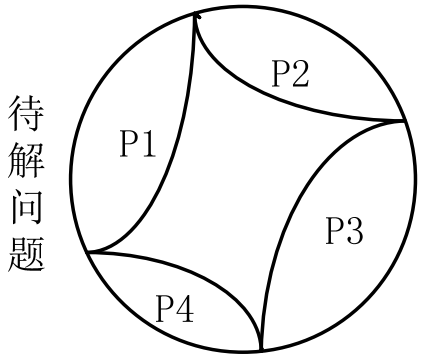
由事务映射产生

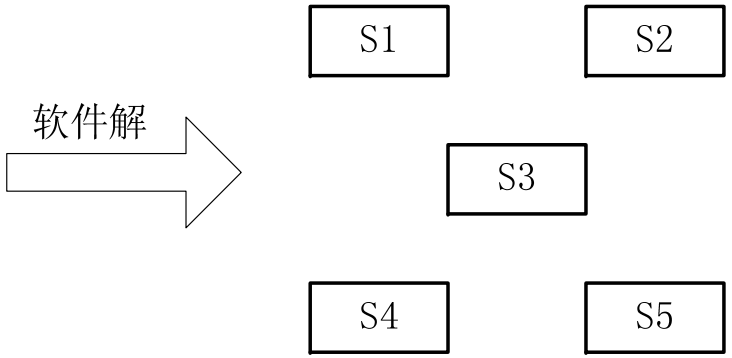


12.5* 结构图(SC)的优化原则

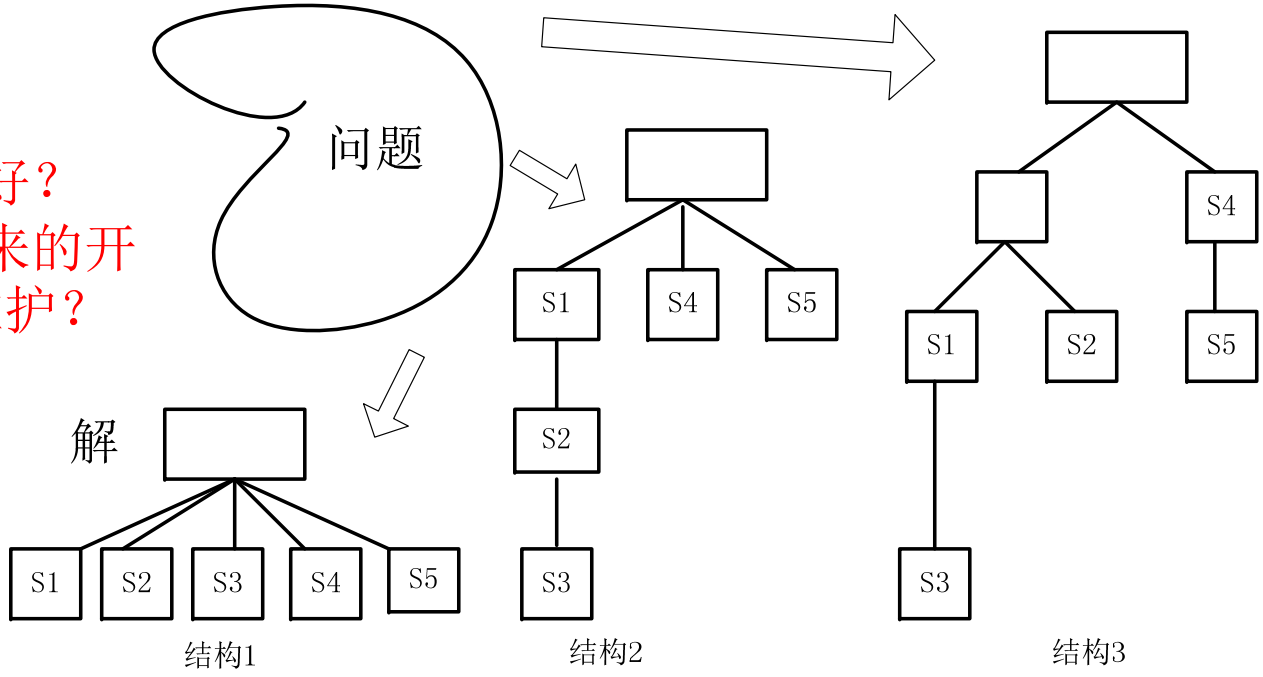
为什么要进行SC的优化？

同一个问题的各种软件结构





哪种方案更好？
哪个更适应将来的开发、测试、维护？



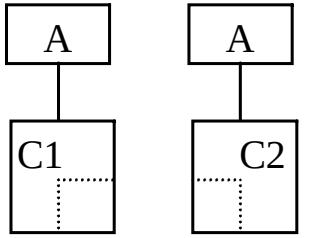
结构设计的优化原则

- **把初始SC变成最终设计文档中的SC，需要进一步的细化和改进。**

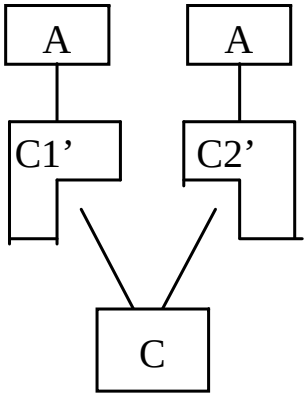
- **改进的依据：**
  - 对模块分割、合并和变动的指导规则
    - 提高内聚，降低耦合
    - 简化模块接口
    - 少用全局性数据和控制型信息
  - 保持高扇入/低扇出的原则
  - 作用域/控制域规则
    - 作用域不要超出控制域的范围
    - 位置离受它控制的模块越近越好

(1) 改进SC提高模块独立性

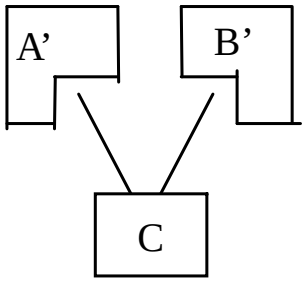
- 通过模块分解或合并，力求降低耦合提高内聚。



如 C1，C2 有类似功能，也有不同功能。可把功能类似的部分分离出来，增加一个公共下属模块



如果余下的 C1’，C2’比较简单，可分别与其上级模块合并，以减少控制的传递、全局数据的引用和接口的复杂性。



(1) 改进SC提高模块独立性

```
void withdraw (Account acc, double money) {
   if(acc.balance >= money) {
        double pre_balance = acc.balance;
        acc.balance = acc.balance – money;
        println(“账户信息：” + acc.info);
        println(“操作前余额：” + pre_balance);
        println(“操作后余额：” + acc.balance);
   }
  else
      打印“余额不足”;
}

void deposit (Account acc, double money) {
        double pre_balance = acc.balance;
        acc.balance = acc.balance + money;
        println(“账户信息：” + acc.info);
        println(“操作前余额：” + pre_balance);
        println(“操作后余额：” + acc.balance);
}
```

```
void withdraw (Account acc, double money) {
   if(acc.balance >= money) {
        double pre_balance = acc.balance;
        acc.balance = acc.balance – money;
        printAccountInfo (pre_balance, acc);
   }
  else
      打印“余额不足”;
}

void deposit (Account acc, double money) {
        double pre_balance = acc.balance;
        acc.balance = acc.balance + money;
        printAccountInfo (pre_balance, acc);
}

void printAccountInfo (double pre, Account acc)  {
        println(“账户信息：” + acc.info);
        println(“操作前余额：” + pre);
        println(“操作后余额：” + acc.balance);
}
```

(1) 改进SC提高模块独立性

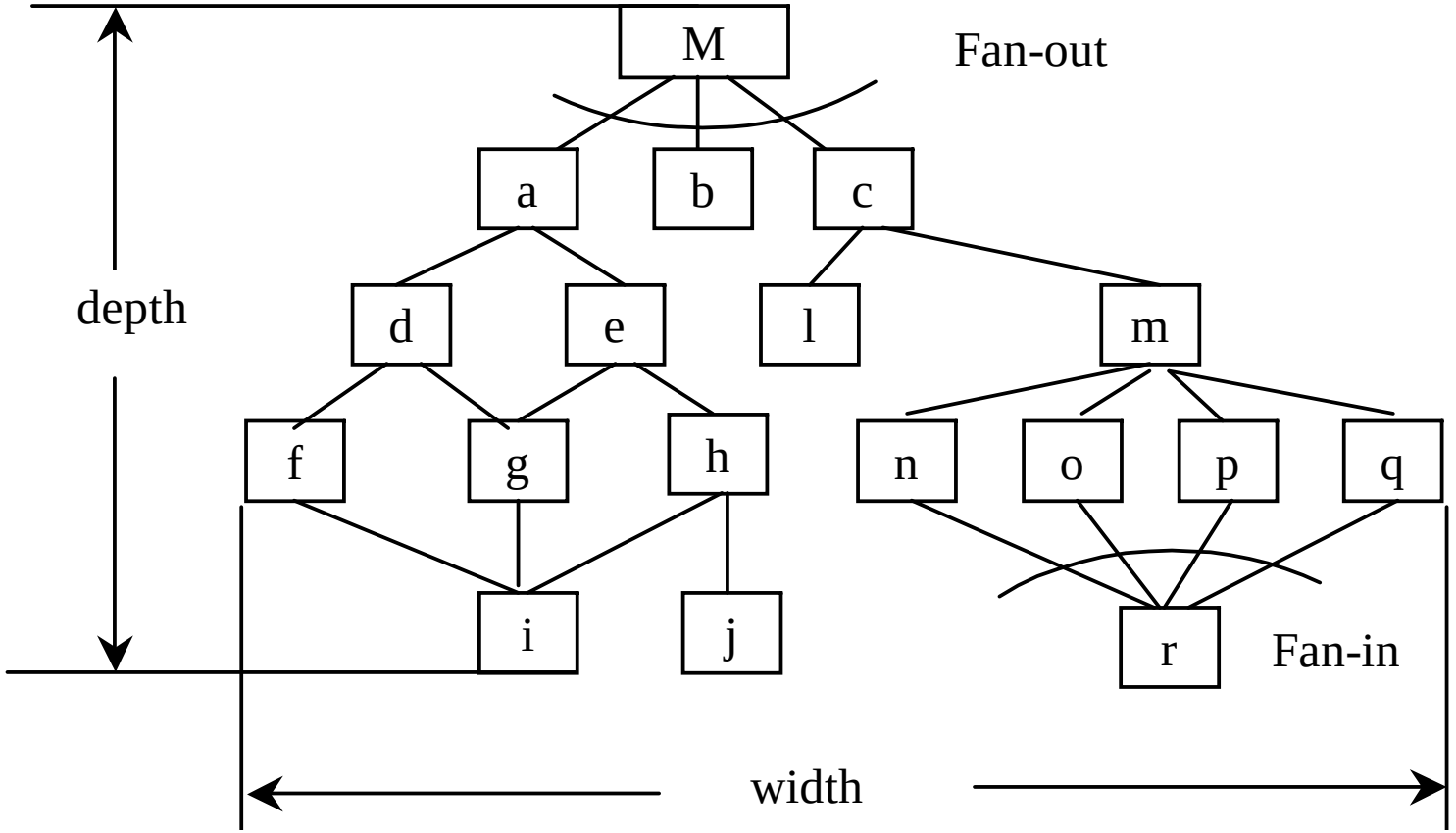
- 通过模块分解或合并，力求降低耦合提高内聚。

- 例如：
  - 对文件的公共操作：打开文件、读取数据、向文件写入数据、等；
  - 对数据库的公共操作：连接DB、执行嵌入式SQL语句、等；
  - 对数据结构的公共操作：排序、max/min/sum/average、等；
  - 打印操作：向屏幕输出、向打印机输出、等；
  - 绘图操作：在屏幕上绘制图形，对图形的各类公共操作(移动、删除等)；

(2) 模块规模要适中

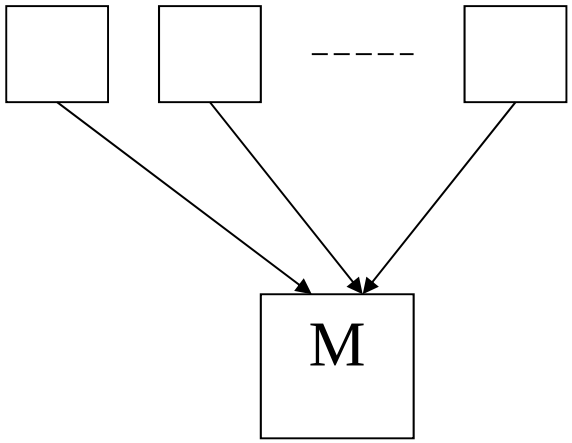
- **模块规模适中：**
  - 过大：分解不充分，不易理解——可以对功能进一步分解，生成一些下级模块或同层模块；
  - 太小：开销过大、接口复杂——把它合并到上级模块中去而不必单独存在
    - 但是如果体积小的模块是功能内聚性模块，或者它为多个模块所共享，或者调用它的上级模块很复杂，则不要把它合并到其它模块中去。
- **一个模块的规模最好能写在一页纸内，通常不超过60行语句。**

(3) 深度、宽度、扇出和扇入

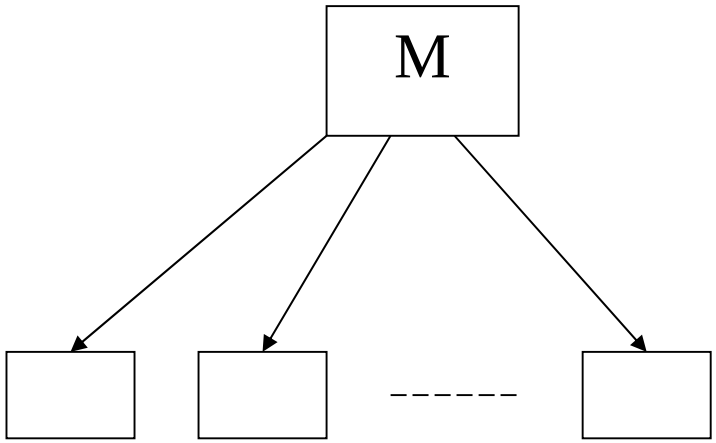


模块的扇入和扇出

- **扇入(fan-in)、扇出(fan-out)**
  - 扇入高表示上级模块多，可增加模块的利用率；
  - 扇出低表示下级模块少，可以减少模块调用和控制的复杂度。
  - 原则：减少高扇出结构；随着深度增大扇入；



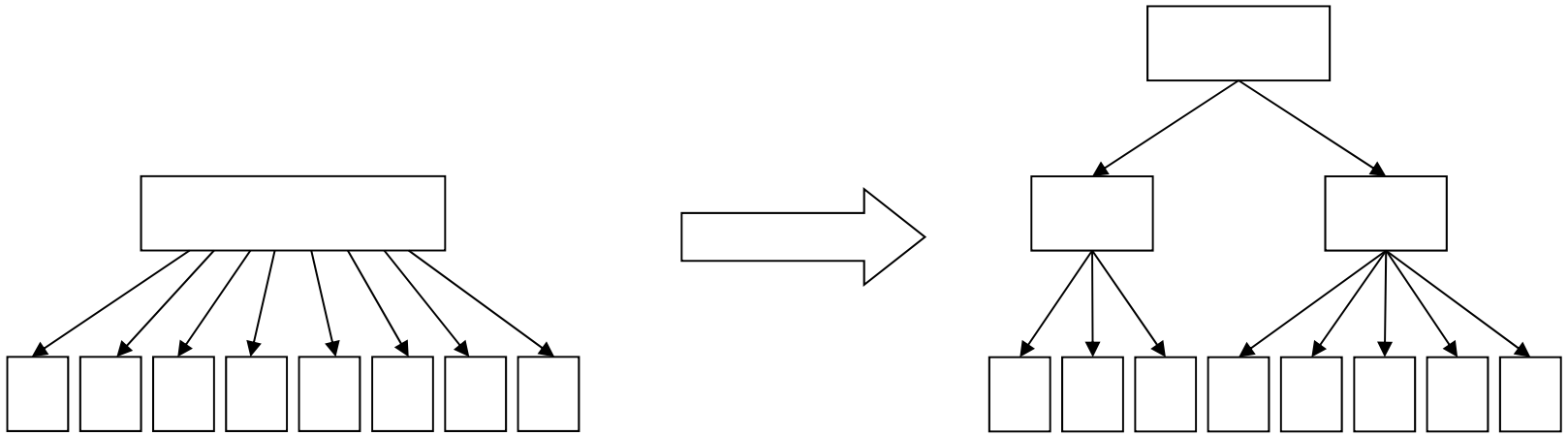
M的扇入



M的扇出

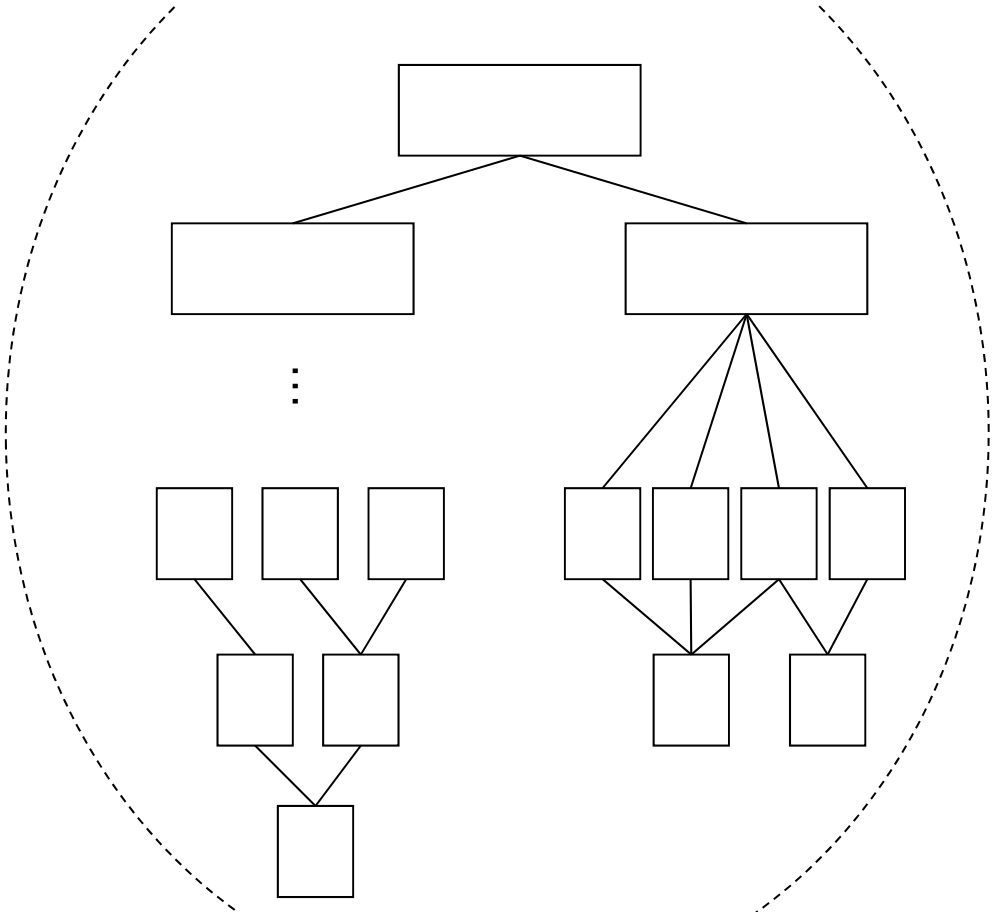
模块的扇入和扇出

- 通常，模块的扇出数以**3~4**为好，最多不超过**5~7**。
- 如果一个模块的扇出数过大，软件结构将呈煎饼形，就意味着该模块过分复杂，需要协调和控制过多的下属模块。应当适当增加中间层次的办法使扇出减少。
- 方法：对下层模块进行归类，合并为若干中间层模块；

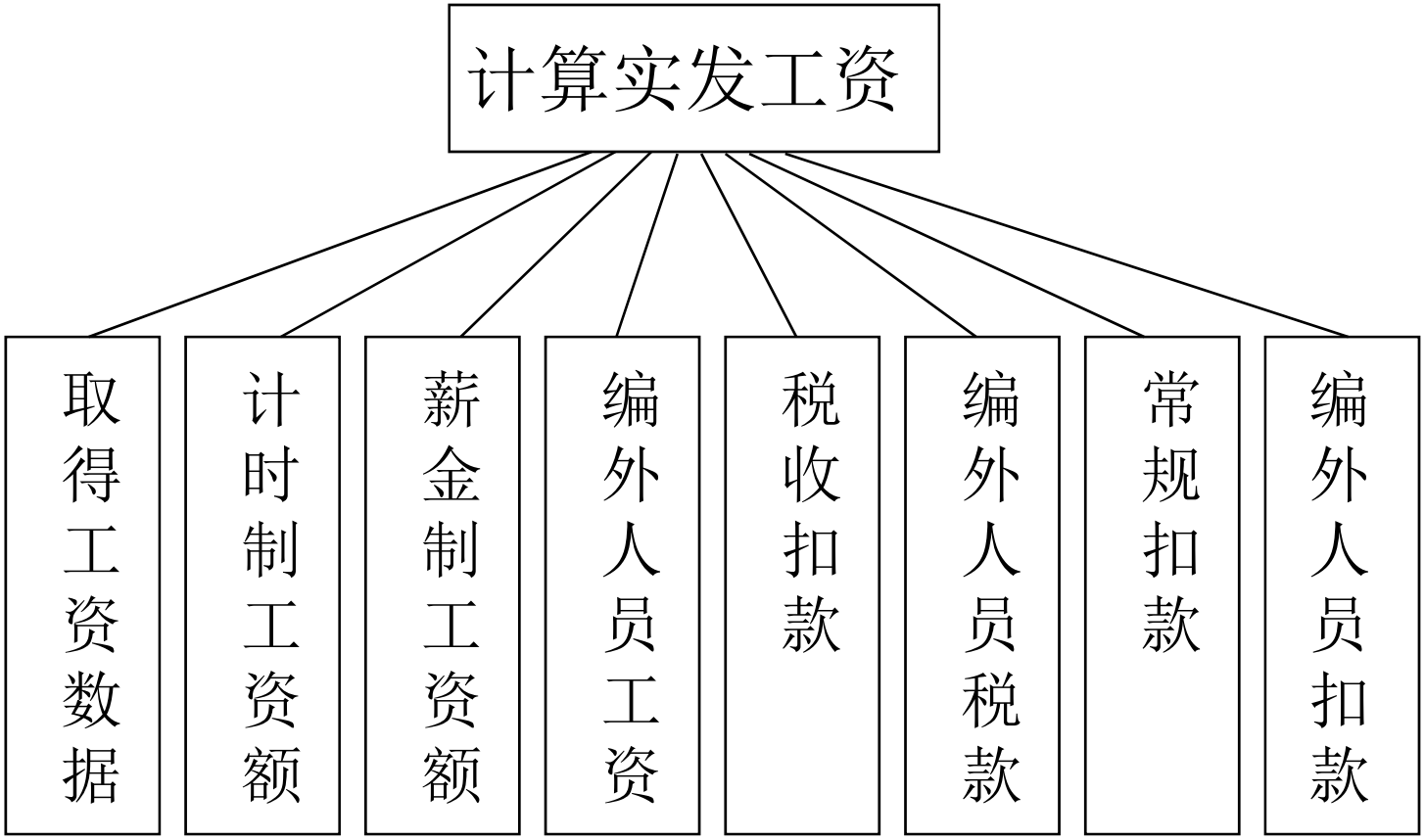


模块的扇入和扇出

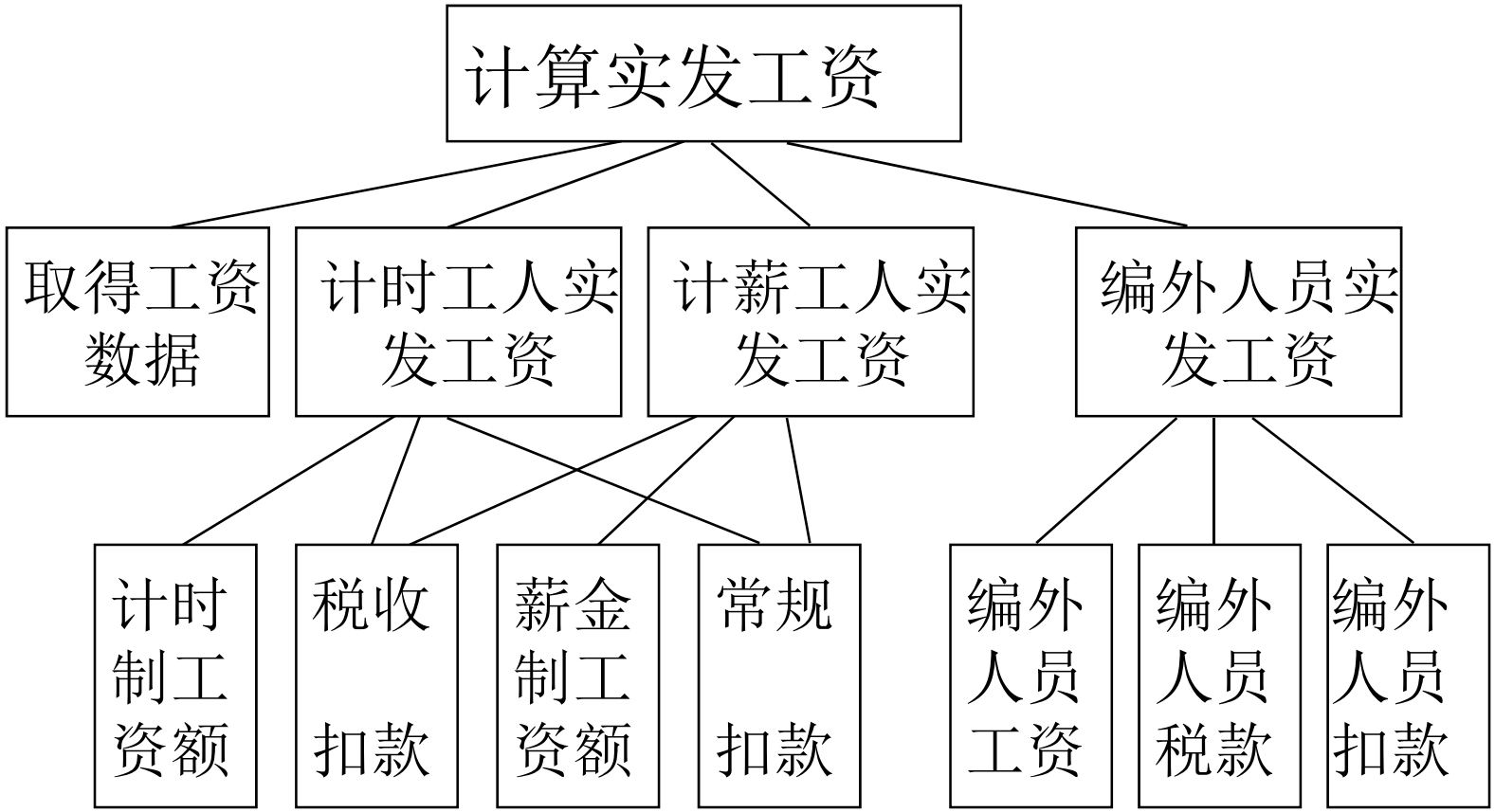
- 设计良好的软件通常具有瓮型结构，两头小，中间大；
- 该类系统的结构在下部收拢，表明它在底层模块中使用了较多高扇入的共享模块。
- 在不破坏独立性的前提下，扇入大的比较好。



示例：高扇出


计算实发工资
取得工资数据
计时制工资额
薪金制工资额
编外人员工资
税收扣款
编外人员税款
常规扣款
编外人员扣款

示例：修改后

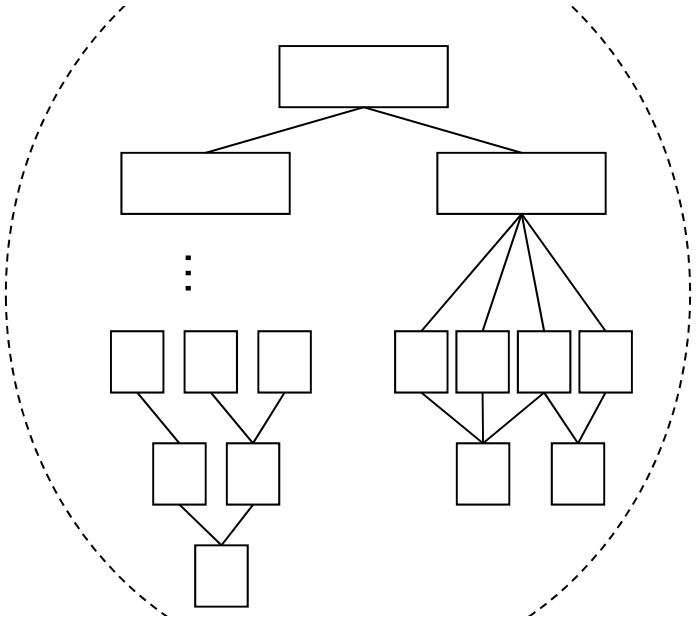

计算实发工资
取得工资数据
计时工人实发工资
计薪工人实发工资
编外人员实发工资
计时制工资额
税收扣款
薪金制工资额
常规扣款
编外人员工资
编外人员税款
编外人员扣款

深度和宽度

- **适当控制 ——**
  - 深度 = 分层的层数。

    深度过大表示分工过细。
  - 宽度 = 同一层上模块数的最大值。

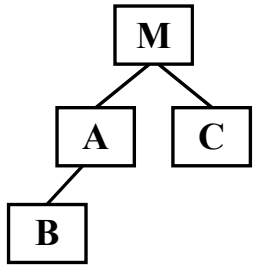
    宽度过大表示系统复杂度大。



(4) 作用域/控制域规则

- **一个模块的控制域(Scope of Control)，等于模块本身加上其下级模块(即可供它调用的模块)；**
- **一个模块的作用域(Scope of Effect)，是受这个模块中的判定(IF语句)所影响的模块。**

- **规则：**
  - 模块的作用范围保持在该模块的控制范围内；
  - 判定的位置离受它控制的模块越近越好。



M的控制域为 {M，A，B，C}

作用域/控制域规则

- **若F中有判定，影响到B...**

  不好，使模块件出现控制耦合

- **若M中有判定，影响到B和F...**

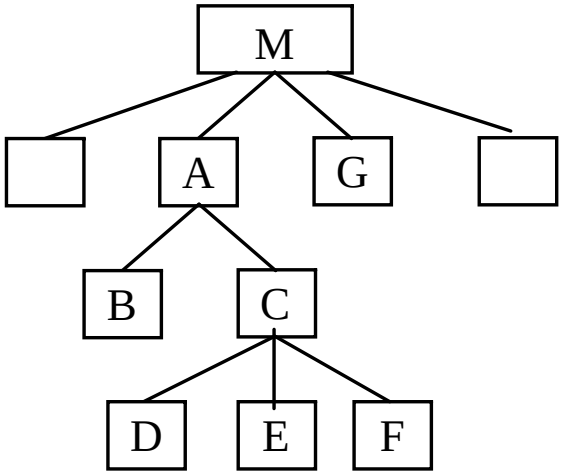
  可用但不好，判定在模块层次中的位置太高

- **若A中有判定，影响到B和F...**

  较好：控制范围正好包含了作用范围，距离也不太远

- **若C中有判定，影响到D和F...**

  理想的设计



原因？——GOTO语句造成的祸害。
消除方式？——在任何使用避免使用GOTO语句！

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

12.6* 模块化设计原则

12.6.1 模块化设计

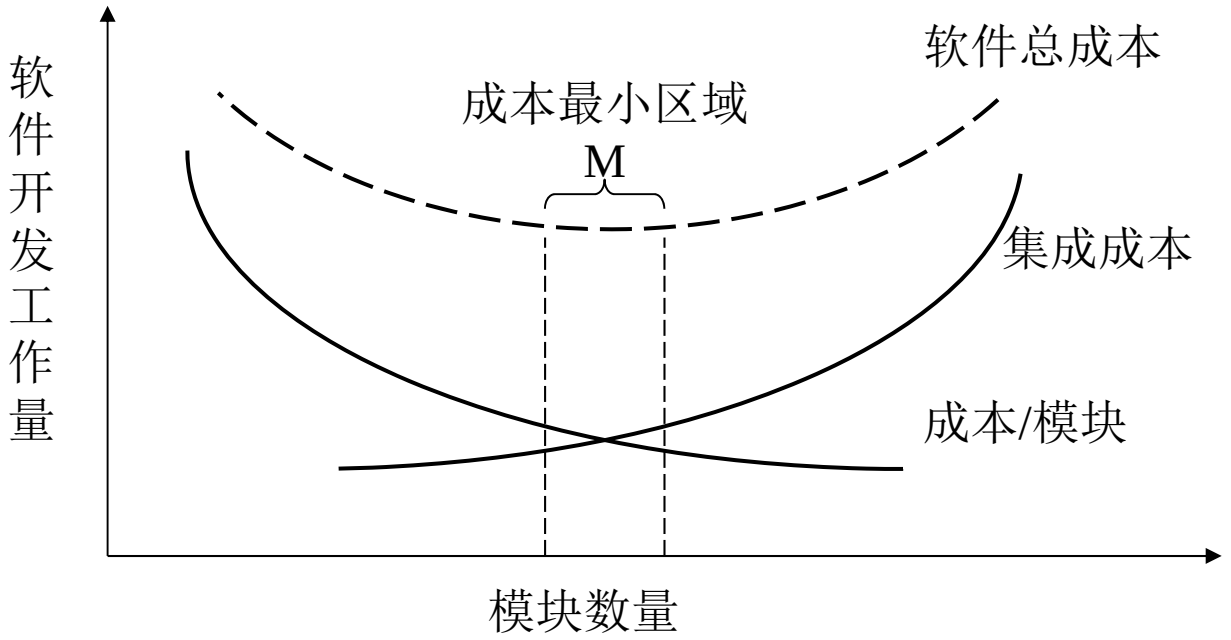
- **模块化：软件系统被分成若干个模块**
  - 每个模块完成一个相对独立的子功能
  - 每个模块定义了各自的输入和输出
  - 若干个模块集成在一起，形成整体，完成系统的全部功能。

- **基本思想：分而治之，将复杂大问题分解为小问题**
  - 在系统整个生命周期中，通过功能分解降低系统复杂度、提高复用度，以最小化开发和维护成本。
  - 选定问题→分解得到模块→描述模块的交互→重复分解

模块化设计的依据

$$C(P_1+P_2) > C(P_1) + C(P_2)$$
$$E(P_1+P_2) > E(P_1) + E(P_2)$$

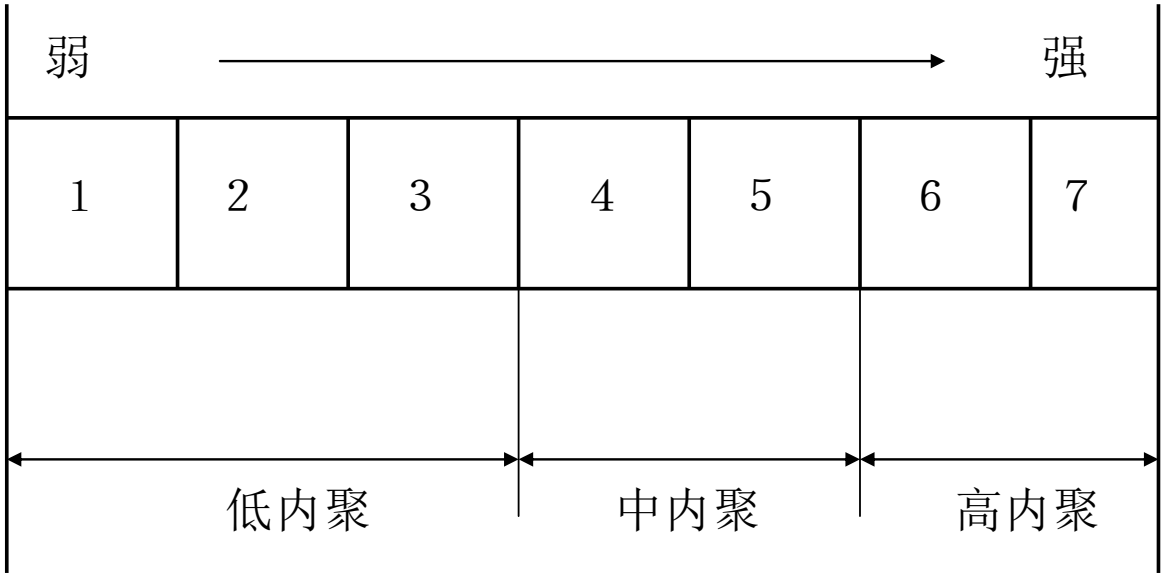
C为问题的复杂度，E为解题需要的工作量



12.6.2 模块独立性之内聚性

- **模块独立性：用来判断模块构造是否合理的标准。**
- **从两个方面来度量：**
  - 模块内部的聚合度(Cohesion)：某一模块内部包含的各功能之间相互关联的紧密程度；
  - 模块之间的耦合度(Coupling)：多个模块之间相互关联的紧密程度；
- **模块化设计的目标：**
  - 高内聚、低耦合(high cohesion and low coupling)

内聚强度的划分



1 .偶然性内聚 Coincidental Cohesion
2 .逻辑性内聚 Logical Cohesion
3 .时间性内聚 Temporal Cohesion
4 .过程性内聚 Procedural Cohesion
5 .通讯性内聚 Communicational Cohesion
6 .顺序性内聚 Sequential Cohesion
7 .功能性内聚 Functional Cohesion

(7) 功能性内聚

- **功能性内聚：模块中各个部分都是为完成一项单一的功能而协同工作**
  - 模块只执行单一的计算并返回结果
  - 无副作用(执行前后系统状态相同)，对其他模块无影响；
  - 这类模块通常粒度最小，且不可分解；
  - 通常以“动宾短语”来命名。

- **例如：**
  - 根据输入的角度，计算其正弦值；
  - 求一组数中的最大值；

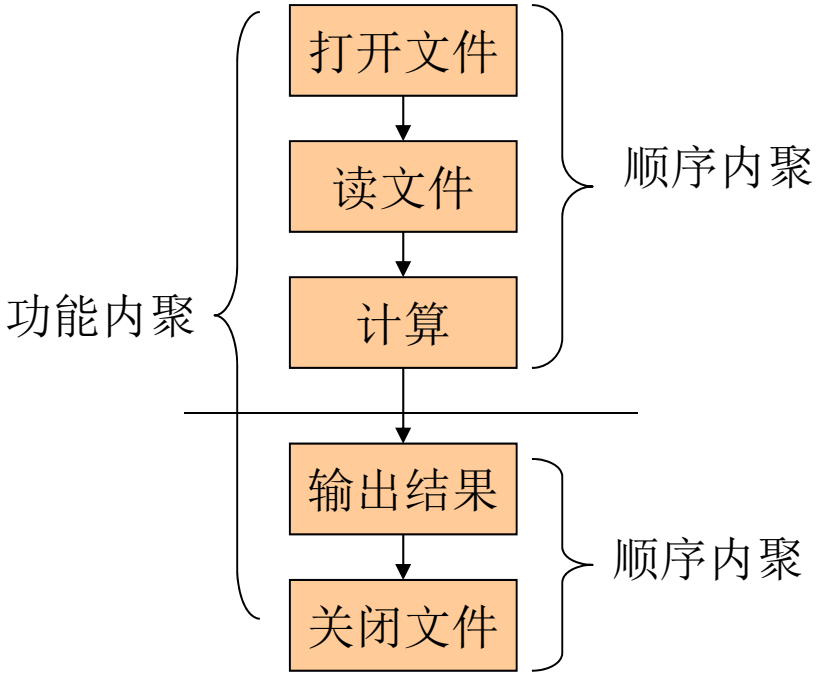
- **“集中精力做一件事情”**

优点：

- 模块的功能只是生成特定的输出而没有副作用，容易理解该模块。
- 因功能内聚的模块没有副作用，所以模块的可复用性高。
- 功能单一、易修改、易替换、易维护

(6) 顺序性内聚

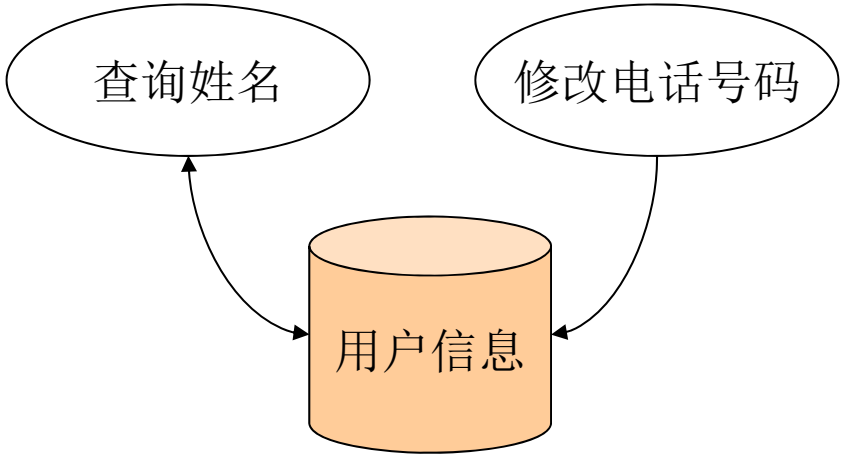
- **顺序性内聚：**
  - 模块完成多个功能，它们无法实现一个完整的功能；
  - 各功能之间按顺序执行，形成操作序列；
  - 上一个功能的输出是下一个功能的输入；
  - 各功能都在同一数据结构上操作。
  - 判断标准：能否用“动宾短语”命名？
- “先后次序，放置在一起”

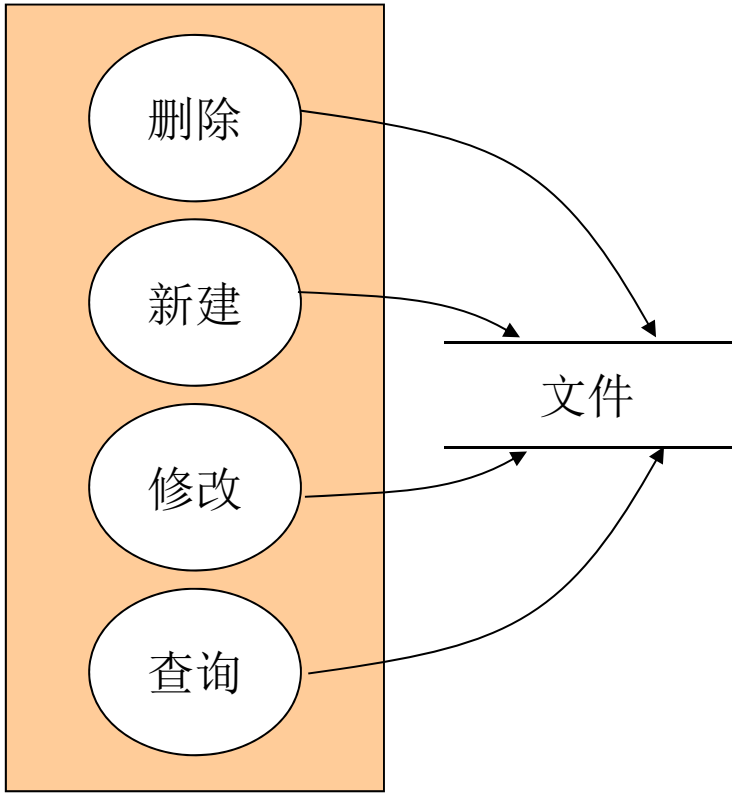


(5) 通讯性内聚

- **通讯性内聚：**
  - 模块内各部分操作访问相同的数据
  - 除此之外，再无任何关系
  - “无特定的次序，非单一功能”

- **“处理同样的数据”**



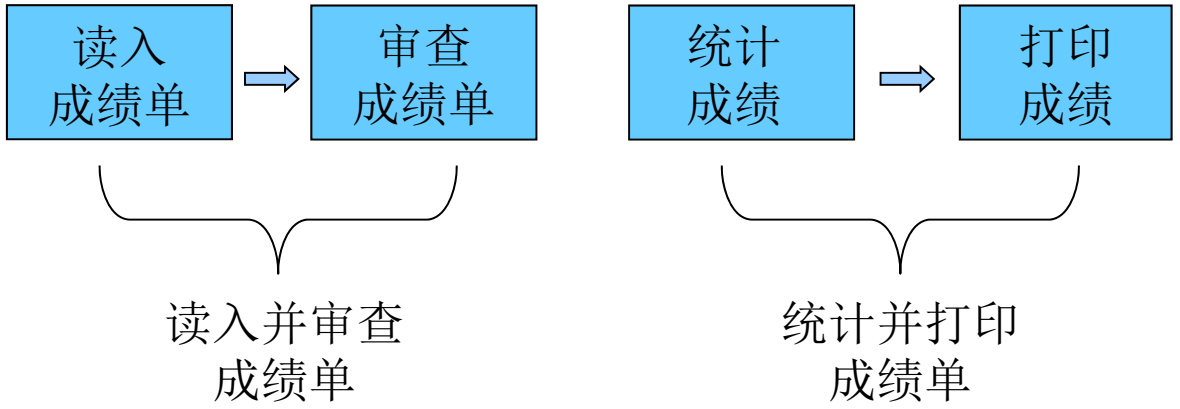


优点：容易维护

缺点：难以复用

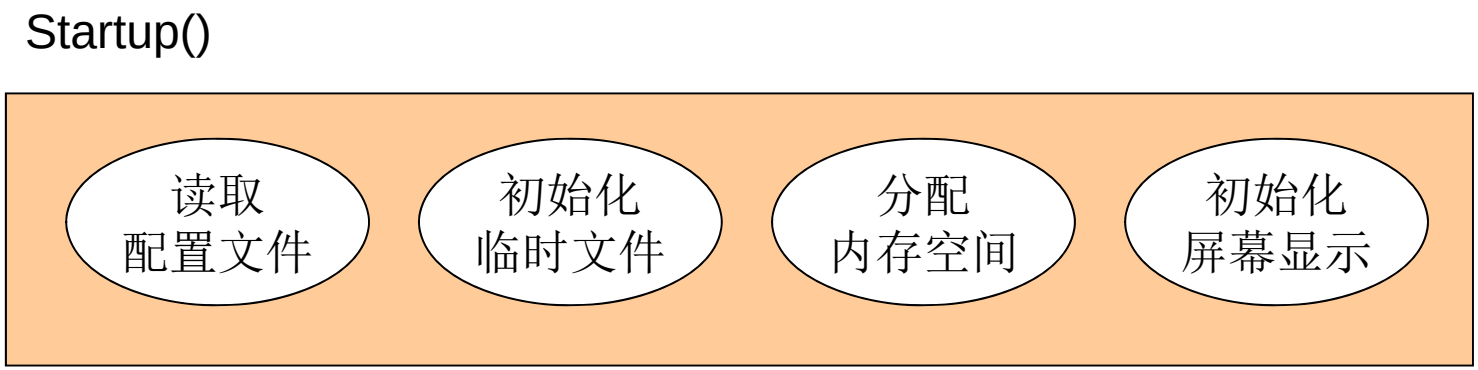
(4) 过程性内聚

- **过程性内聚：**
  - 模块中的操作遵循某一特定的顺序
  - 但这些顺序操作并不使用相同的数据
  - 给这类模块命名是非常困难的



(3) 时间性内聚

- **时间性内聚：**
  - 模块的各个成分必须在同一时间段执行，但各个成分之间无必然的联系

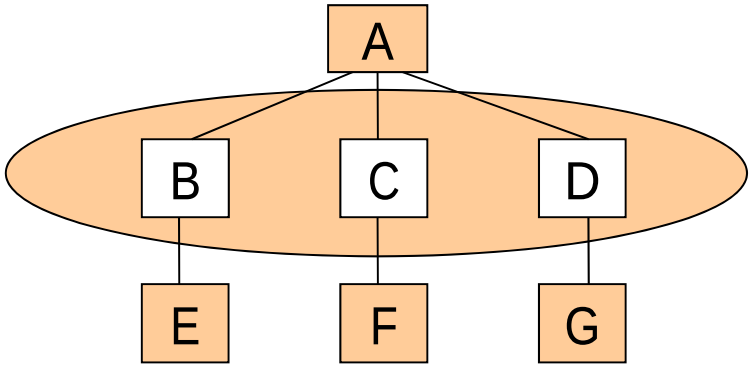


问题：不同的功能混在一个模块中，有时共用部分编码，
使局部功能的修改牵动全局。

(2) 逻辑性内聚

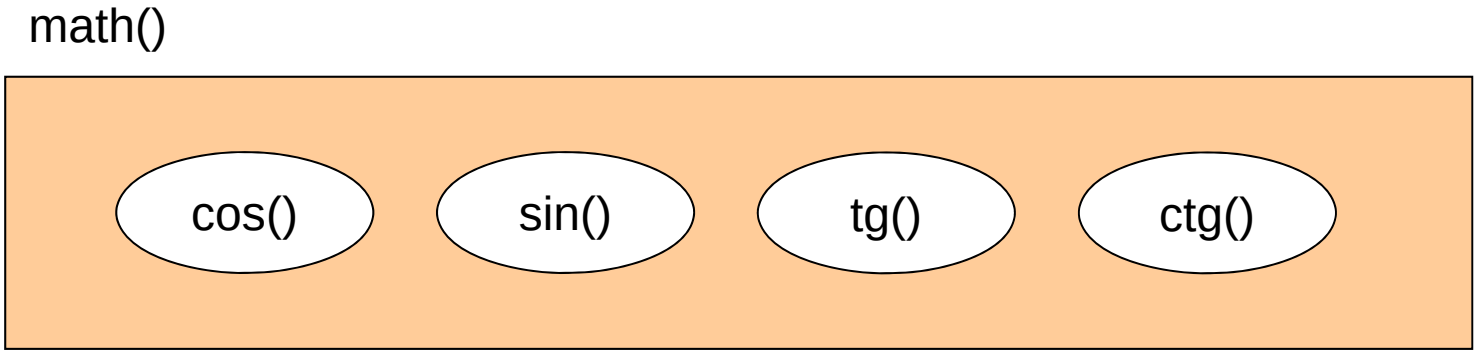
- **逻辑性内聚：**
  - 一个模块中同时含有几个操作，这些操作之间既无顺序关系，也无数据共享关系；
  - 它们的执行与否由外面传进来的控制标志所决定
  - 之所以称之为逻辑内聚性，是因为这些操作仅仅是因为控制流，或者说“逻辑”的原因才联系到一起的，它们都被包括在一个很大的if或者case语句中，彼此之间并没有任何其它逻辑上的联系。

问题：接口难于理解；完成多个操作的代码互相纠缠在一起，导致严重的维护问题。



(1) 偶然性内聚

- **偶然性内聚：**
  - 构成模块的各组成部分无任何关联。
  - 通常用于库函数管理，将多个相互无关但比较功能类似的模块放置在同一个模块内。

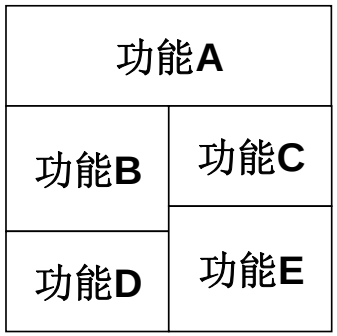


缺点：产品的可维护性退化；模块不可复用，增加软件成本。
解决：将模块分成更小的模块，每个小模块执行一个操作。

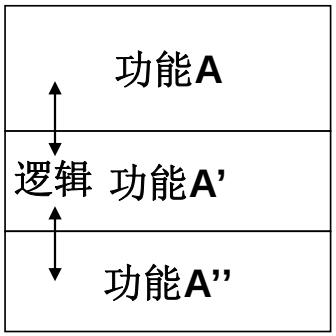
内聚强度的划分

内聚类型	说　明
偶然性内聚	各组成部分在功能上互不相关
逻辑性内聚	各组成部分逻辑功能相似
时间性内聚	各组成部分需要在同一时间内执行
过程性内聚	各组成部分必须按照某一特定的次序执行
通信性内聚	各组成部分处理公共的数据
顺序性内聚	各组成部分顺序执行，前一个的输出数据为后一个的输入数据
功能性内聚	内部所有活动均完成单一功能

内聚强度的划分



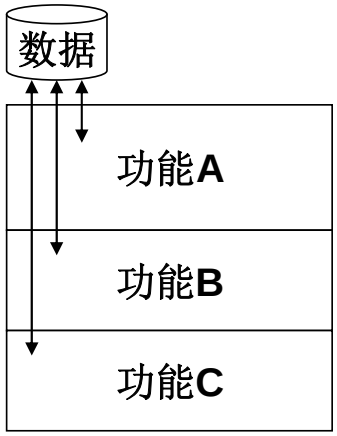
偶然内聚
各部分不相关



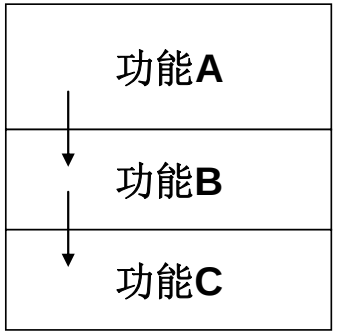
逻辑内聚
类似的功能

时间内聚
按时间相关

过程内聚
按功能的顺序相关



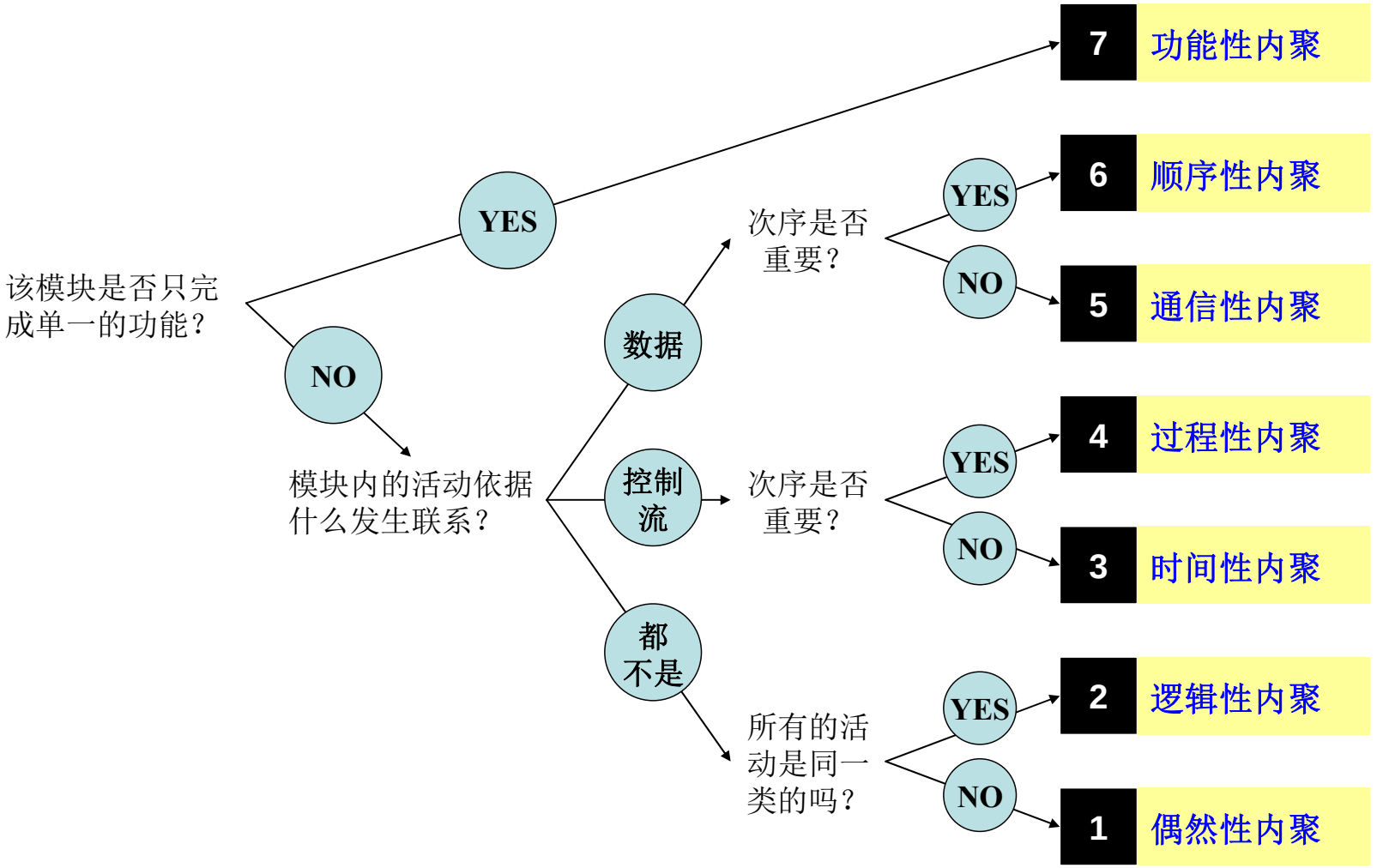
通信内聚
访问同样的数据

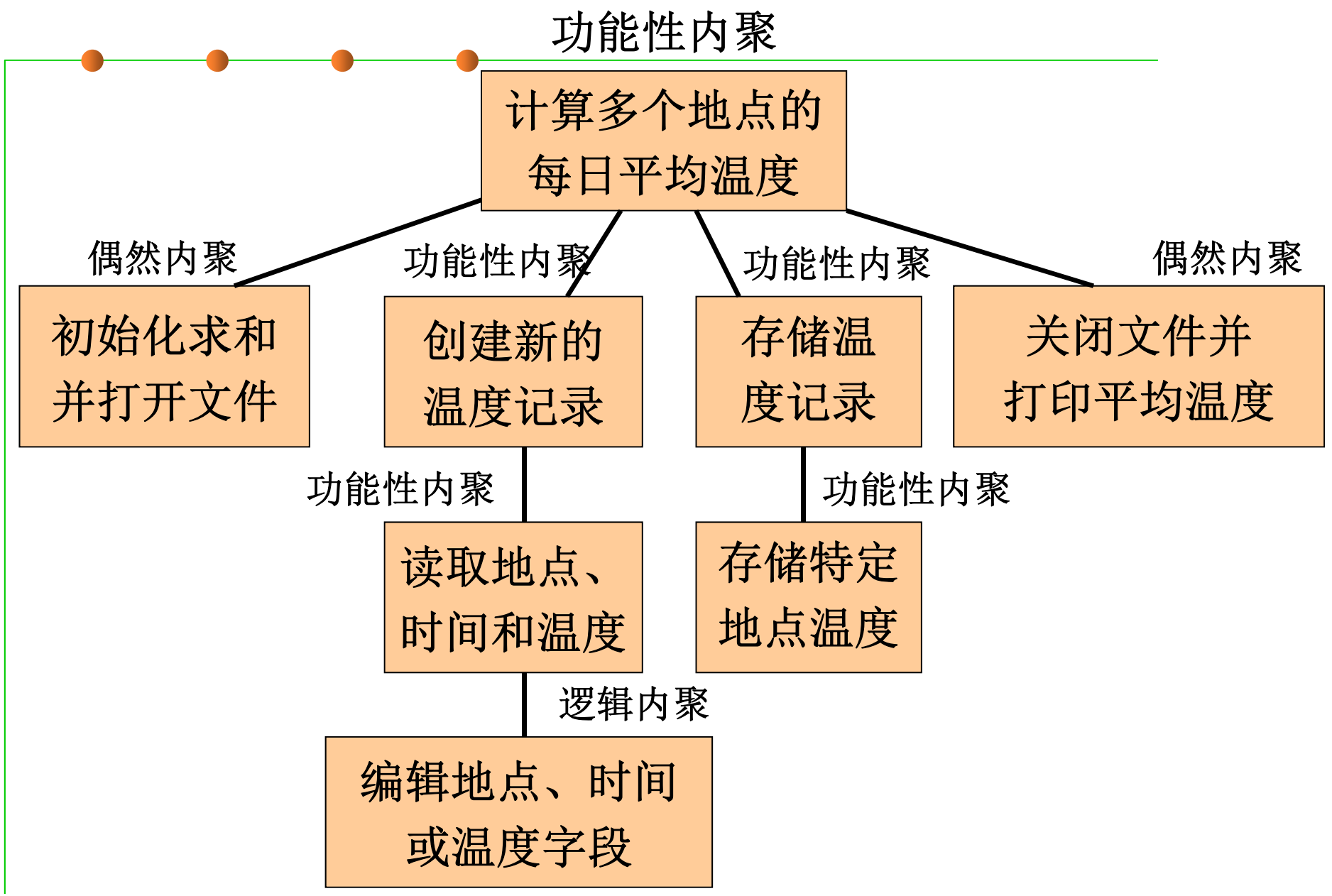


顺序内聚
某一部分的输出
是下一部分的输入

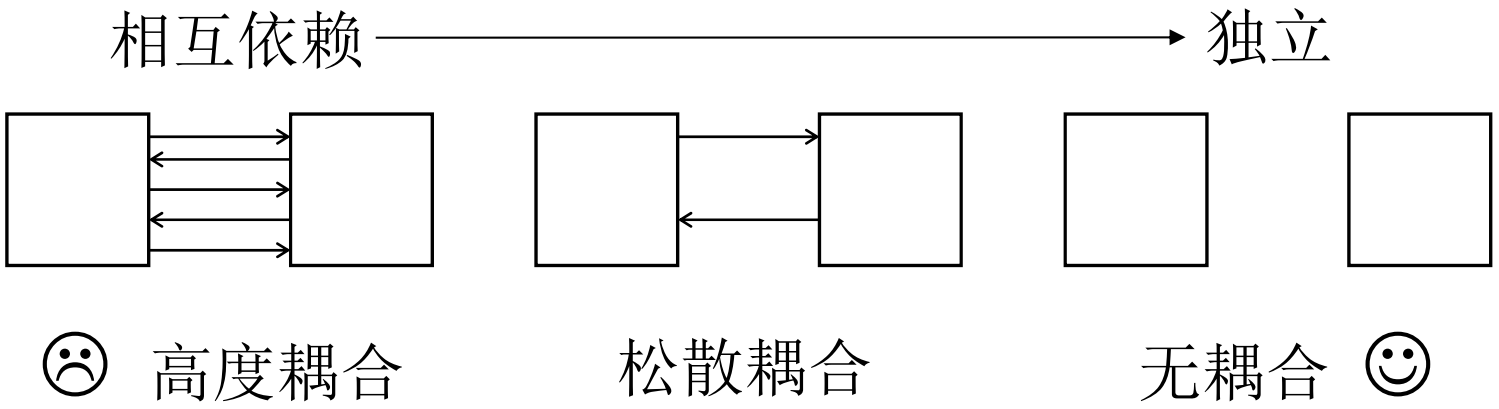
功能内聚
完备的、相关的功能

内聚强度的划分


该模块是否只完成单一的功能？
YES
7
功能性内聚
NO
模块内的活动依据什么发生联系？
数据
次序是否重要？
YES
6
顺序性内聚
NO
5
通信性内聚
控制流
次序是否重要？
YES
4
过程性内聚
NO
3
时间性内聚
都不是
所有的活动是同一类的吗？
YES
2
逻辑性内聚
NO
1
偶然性内聚


功能性内聚
计算多个地点的
每日平均温度
偶然内聚
初始化求和
并打开文件
功能性内聚
创建新的
温度记录
功能性内聚
存储温
度记录
偶然内聚
关闭文件并
打印平均温度
功能性内聚
读取地点、
时间和温度
功能性内聚
存储特定
地点温度
逻辑内聚
编辑地点、时间
或温度字段

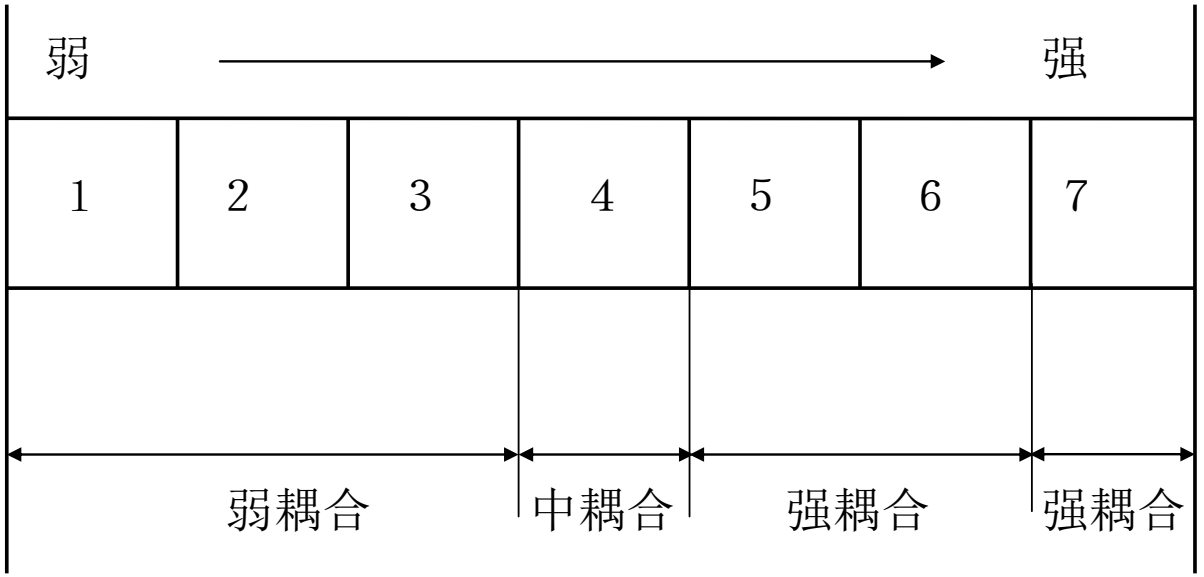
12.6.3 模块独立性之耦合度



影响模块之间发生耦合的因素

- 一个模块引用另一个模块；
- 一个模块传递给另一个模块的数据量；
- 某个模块控制其他模块的数量；
- 模块之间接口的复杂程度；

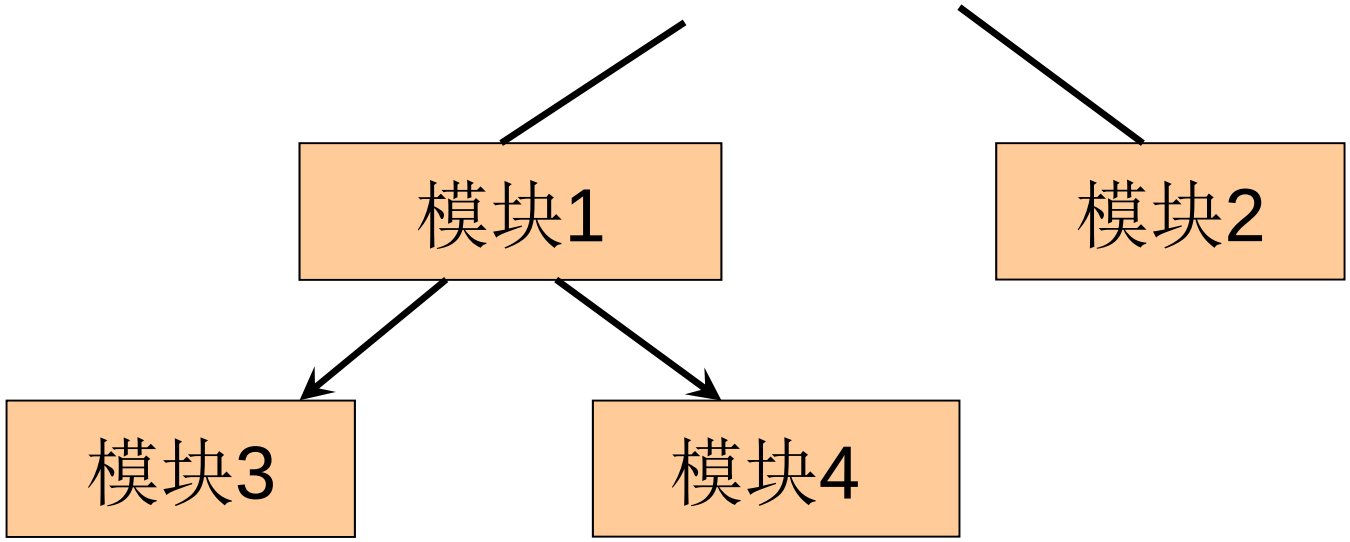
耦合强度的划分



1 非直接耦合　No Direct Coupling
2 数据耦合　Data Coupling
3 特征耦合　Stamp Coupling
4 控制耦合　Control Coupling
5 外部耦合　External Coupling
6 公共耦合　Common Coupling
7 内容耦合　Content Coupling

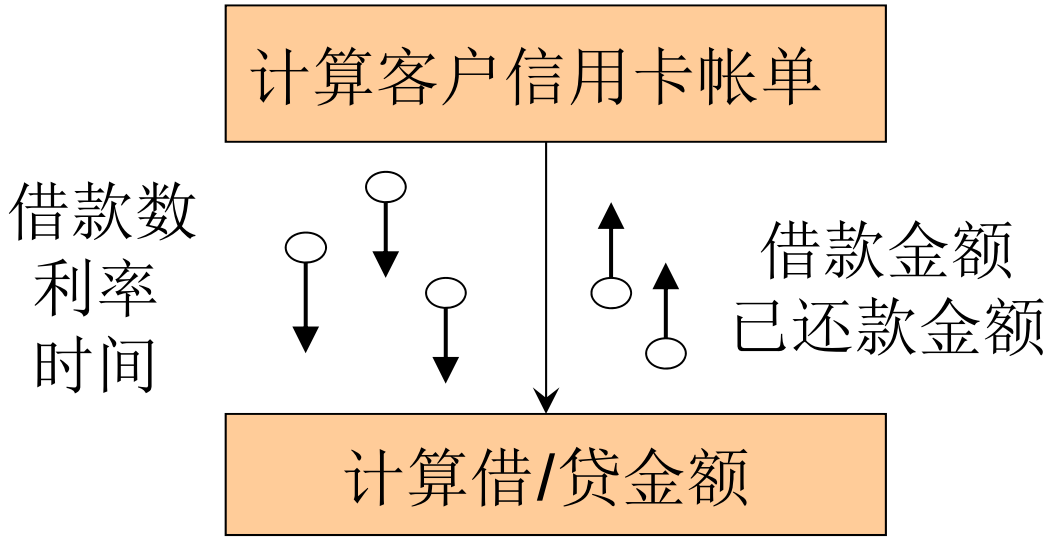
(1) 非直接耦合

- **非直接耦合：**
  - 两个模块之间没有直接关系，它们之间的联系完全是通过主模块的控制和调用来实现的。



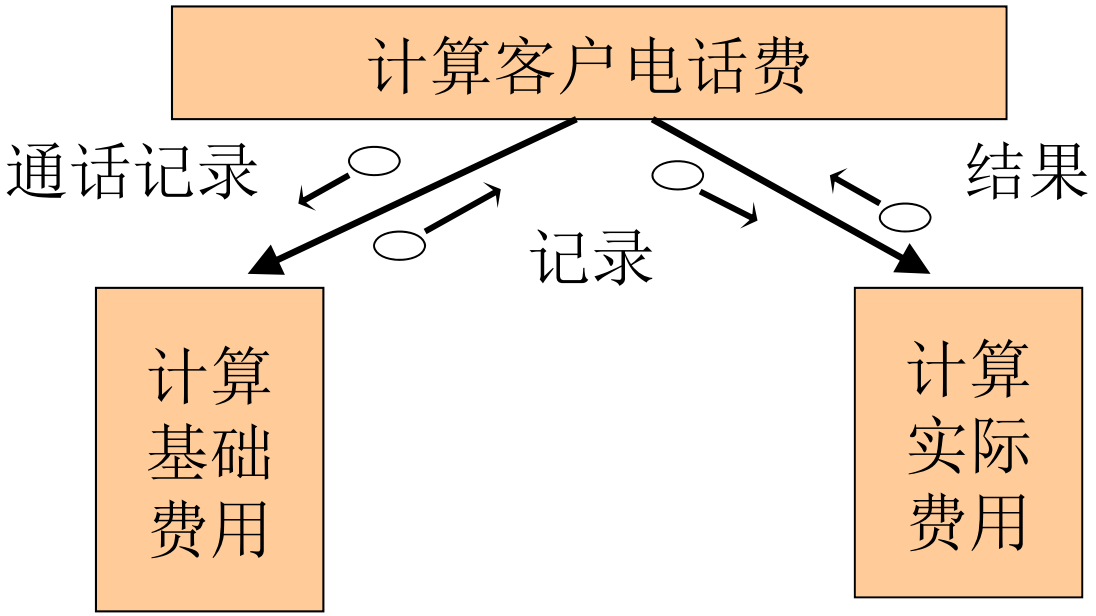
(2) 数据耦合

- **数据耦合：**
  - 一个模块访问另一个模块时，彼此之间是通过简单数据参数 (不是控制参数、公共数据结构或外部变量) 来交换输入、输出信息的。



(3) 特征耦合

- **特征耦合：**
  - 模块之间通过参数表传递记录信息。
  - 这个记录是复杂的数据结构(struct、class等)，而不是简单类型的变量。

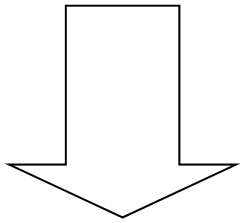


特征耦合的一个例子

- 一个简单的例子：

求一元二次方程的根的模块QUAD-ROOT(TBL, X)

其中用数组TBL传送方程的系数，用数组X回送求得的根

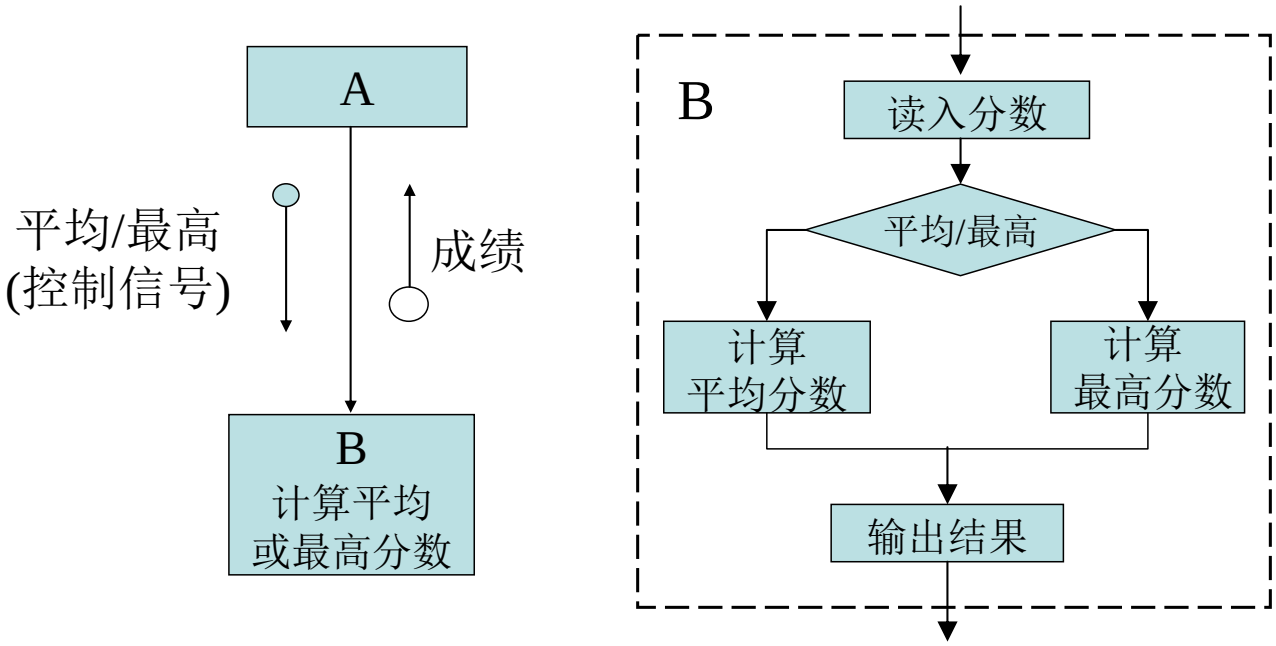


QUAD-ROOT(A, B, C, ROOT1, ROOT2)

其中A、B、C是方程的系数，ROOT1和ROOT2是算出的两个根。

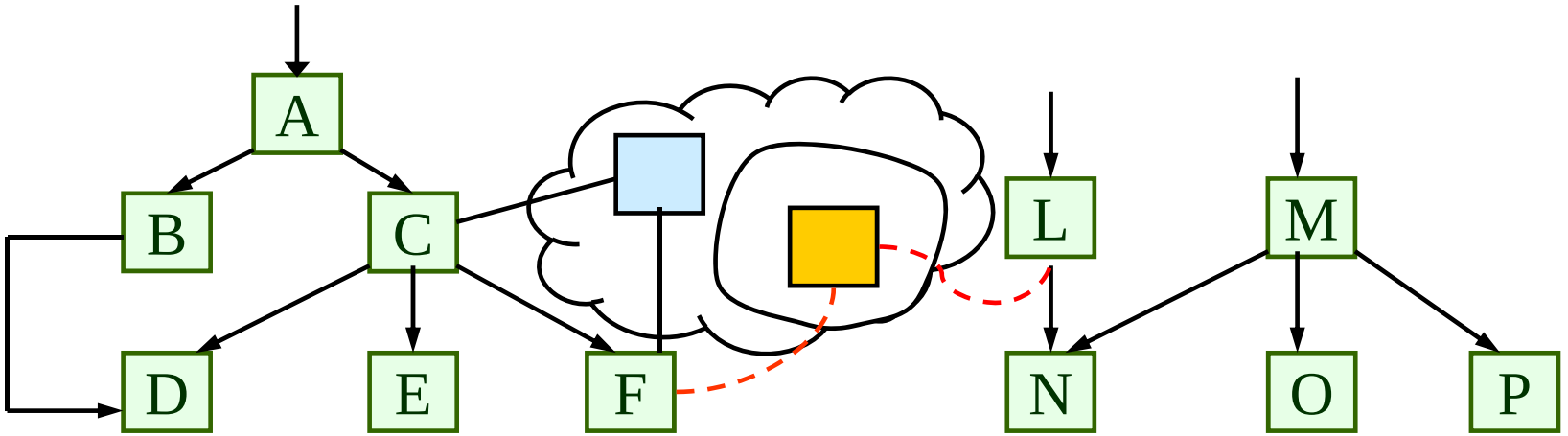
(4) 控制耦合

- **控制耦合：**
  - 如果一个模块通过传送开关、标志、名字等控制信息，以控制选择另一模块的功能，就是控制耦合。



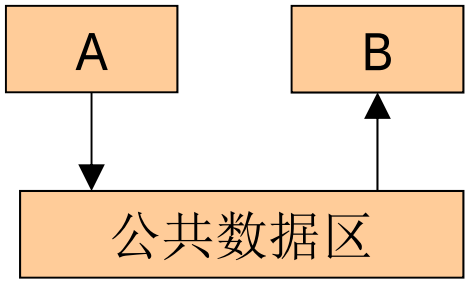
(5) 外部耦合

- **外部耦合：**
  - 一组模块都访问同一全局简单变量而不是同一全局数据结构，而且不是通过参数表传递该全局变量的信息，则称之为外部耦合。

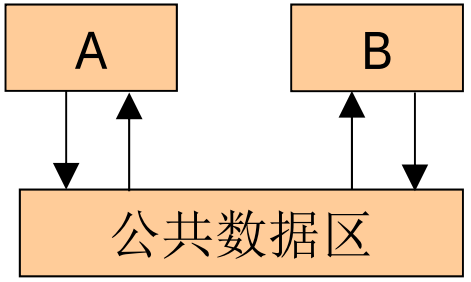


(6) 公共耦合

- **公共耦合：**
  - 若一组模块都访问同一个公共数据环境，则它们之间的耦合就称为公共耦合。
  - 公共的数据环境可以是全局数据结构、共享的通信区、内存的公共覆盖区等。



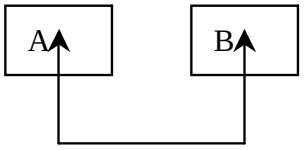
松散的公共耦合



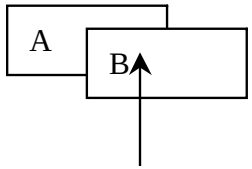
紧密的公共耦合

(7) 内容耦合

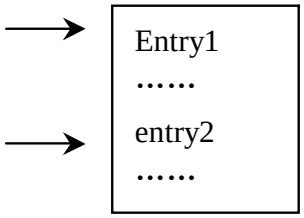
- **内容耦合：如果发生下列情形，两个模块之间就发生了内容耦合**
  - 一个模块直接访问或修改另一个模块的内部数据;
  - 一个模块不通过正常入口转到另一模块内部;
  - 两个模块有一部分程序代码重迭(只可能出现在汇编语言中);
  - 一个模块有多个入口。
- **内容耦合违反了“信息隐藏”的原则。**



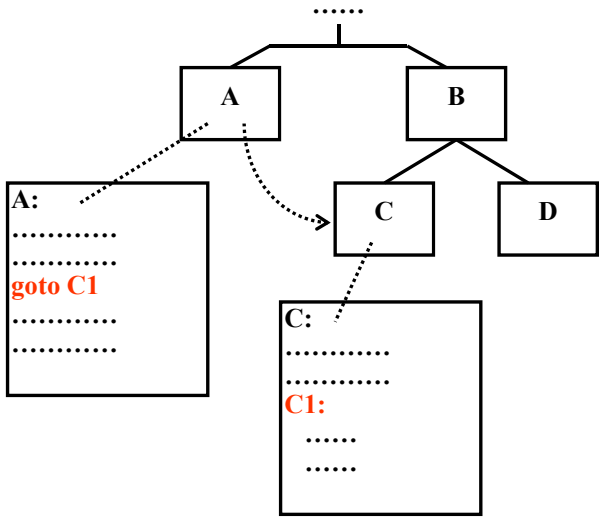
进入另一模块内部



模块代码重叠



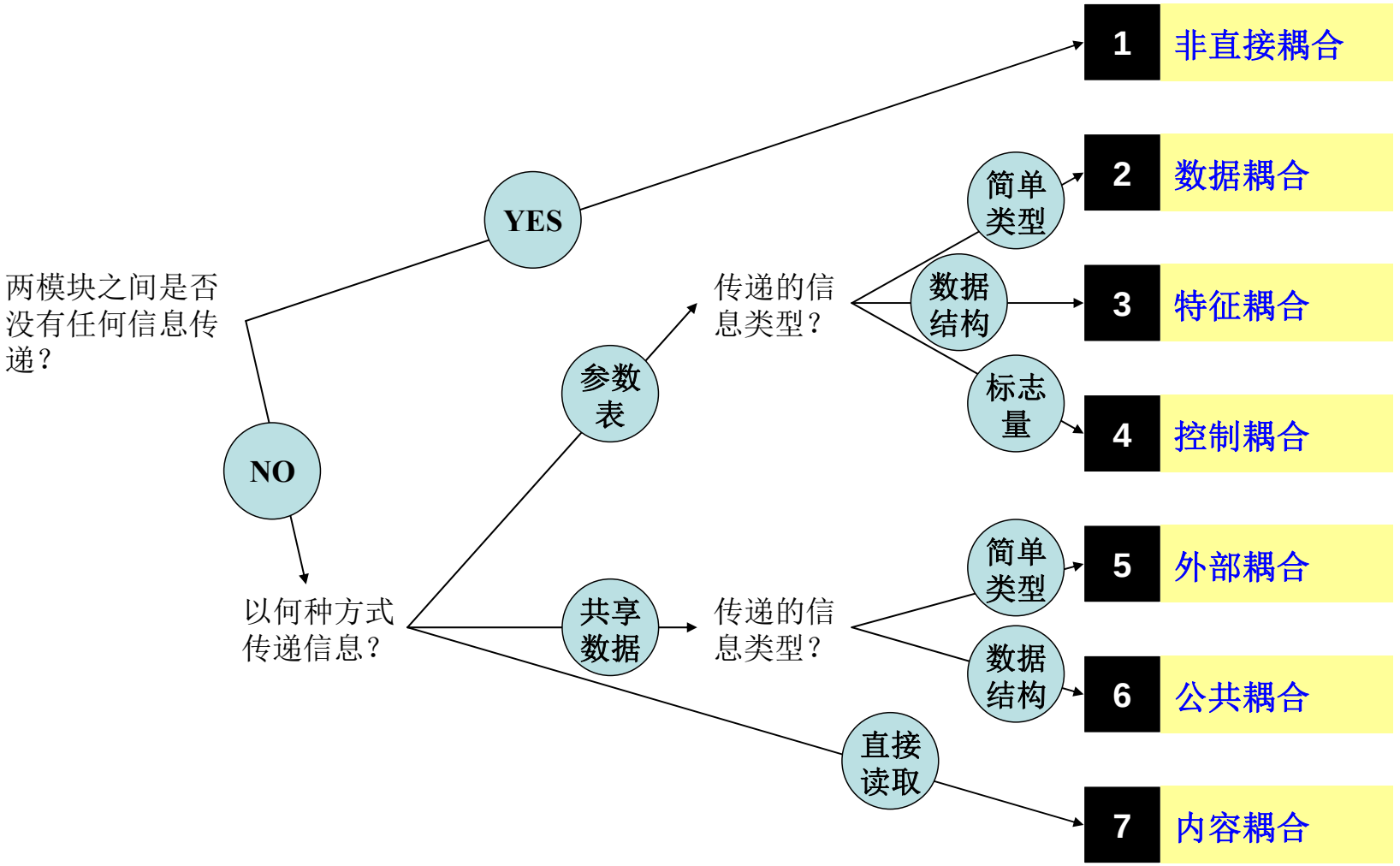
多入口模块



耦合强度的划分

内聚类型	说明
非直接耦合	两个模块之间没有直接关系
数据耦合	通过简单数据参数交换信息
特征耦合	通过复杂数据参数交换信息
控制耦合	传送开关、标志、名字等控制信息
外部耦合	访问同一全局简单变量
公共耦合	访问同一个公共数据环境
内容耦合	一个模块可直接访问另一模块

耦合强度的等级


两模块之间是否没有任何信息传递？
YES
NO
以何种方式传递信息？
参数表
共享数据
直接读取
传递的信息类型？
传递的信息类型？
简单类型
数据结构
标志量
简单类型
数据结构
1 非直接耦合
2 数据耦合
3 特征耦合
4 控制耦合
5 外部耦合
6 公共耦合
7 内容耦合

关于耦合度的小结

- **耦合是影响软件复杂程度和设计质量的重要因素，应建立模块间耦合度尽可能松散的系统；**

- **降低模块间耦合度：**
  - 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

  - 尽量少的交换数据
  - 尽量小的接口
  - 尽量简单的数据交换

12.7* 数据设计

数据设计

- **DFD=外部实体+加工+数据存储+数据流；**
  - 外部实体←→角色、外部软件/硬件系统；
  - 加工←→SC中的“模块”
  - 数据存储←→？
  - 数据流←→？
- **通过数据设计，将DFD中的数据存储和数据流转化为设计中的数据模型。**

数据存储的形态

- **数据文件(file)：**
  - 由操作系统提供的存储形式，应用系统将数据按字节顺序存储，并定义如何以及何时检索数据。
  - 当需要存储大容量数据、临时数据、低信息密度数据时，选用文件形态；
- **关系数据库(rational database)：**
  - 数据是以表的形式存储在预先定义好的成为Schema 的类型中；
  - 当并发访问要求高、系统跨平台、多个应用程序使用相同数据时，选用数据库形式。

12.5.1 文件设计

- 数据存储文件设计
  - 根据使用要求、处理方式、存储的信息量、数据的活动性，以及所能提供的设备条件等，来确定文件类别，选择文件媒体，决定文件组织方法，设计文件记录格式，并估算文件容量。

- 应用《数据结构》课程中学习的相关知识。

文件设计的三个方面(1)

- **1) 文件的逻辑设计**
  - 整理必须的数据元素
  - 分析数据间的关系
  - 确定文件的逻辑结构
- **2) 文件的物理设计**
- **3) 文件的组织方式设计**

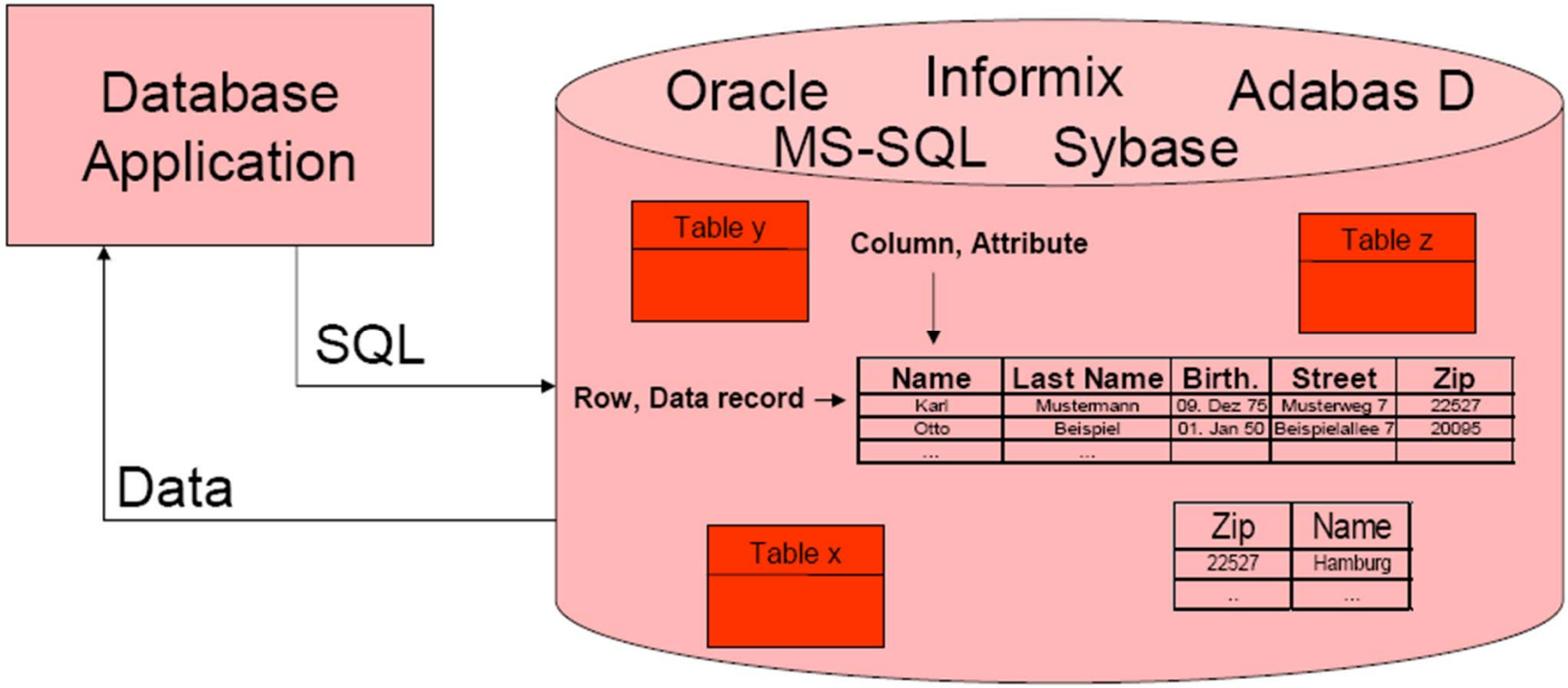
文件设计的三个方面(2)

- **1) 文件的逻辑设计**
- **2) 文件的物理设计**
  - 理解文件的特性
  - 确定文件的存储媒体(考虑数据量、处理方式、存取时间和处理时间、数据结构、操作要求、费用要求等因素)
- **3) 文件的组织方式设计**

文件设计的三个方面(3)

- **1) 文件的逻辑设计**
- **2) 文件的物理设计**
- **3) 文件的组织方式设计**
  - 顺序文件、直接存取文件、索引文件、分区文件、虚拟存储文件、倒排序文件等
  - 确定文件的记录格式
    - 记录的长度
    - 数据项的顺序
    - 数据项的属性
    - 预留空间
    - 子数据项
    - 估算存取时间和存储容量

12.5.2 关系数据库设计



关系数据库设计

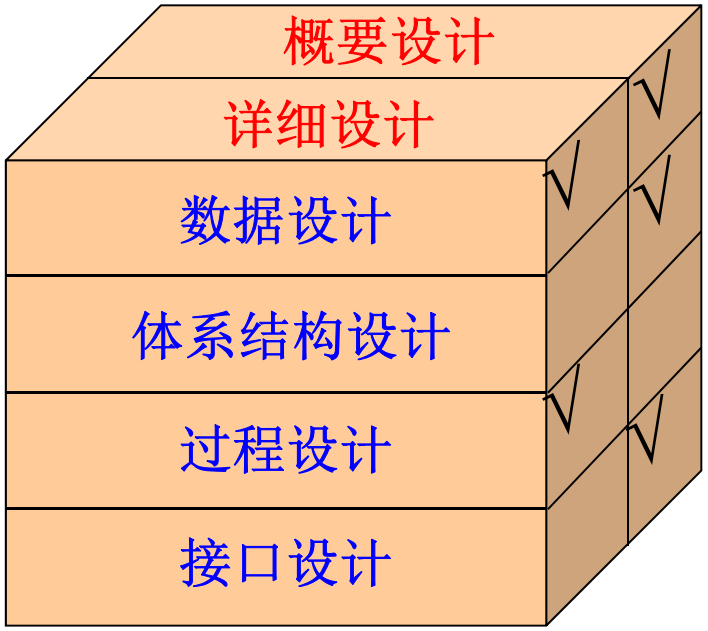
- 为每个实体建立一张表(table)；
- 为每个表确定主键(primary key)、外键(foreign key)；
- 确定表间关系：
  - 增加外部码表示一对多(1:n)关系
  - 建立新表，表示多对多(m:n)关系
- 检查数据字典；
- 对数据表进行规范化(3NF即可)
- 数据库完整性设计
- 数据库安全性设计

应用《数据库系统》课程中学习的相关知识

12.8 详细设计

12.8.1 过程设计

- 过程设计：即详细设计，是软件设计的第二步。
- 在此前的概要设计阶段，已经用SC确定了软件的体系结构，给出了系统中各个模块的功能和模块间的联系(接口)。
- 下一步的工作，就是对每个模块给出足够详细的过程性描述，并使用规范化的表达工具来表示。



过程设计的目的和任务

- 目的：
  - 在使用程序设计语言编制程序以前，需要对所采用算法的逻辑关系进行分析，设计出全部必要的过程细节，并给予清晰的表达。
- 任务：编写软件的“过程设计说明书”
  - 为每个模块确定采用的算法：选择某种适当的工具表达算法的过程，写出模块的详细过程性描述；
  - 确定每一模块使用的数据结构；
  - 确定模块接口的细节，包括对系统外部的接口和用户界面、对系统内部其他模块的接口、以及关于模块输入数据/输出数据/局部数据的全部细节。

12.8.2 过程设计的原则

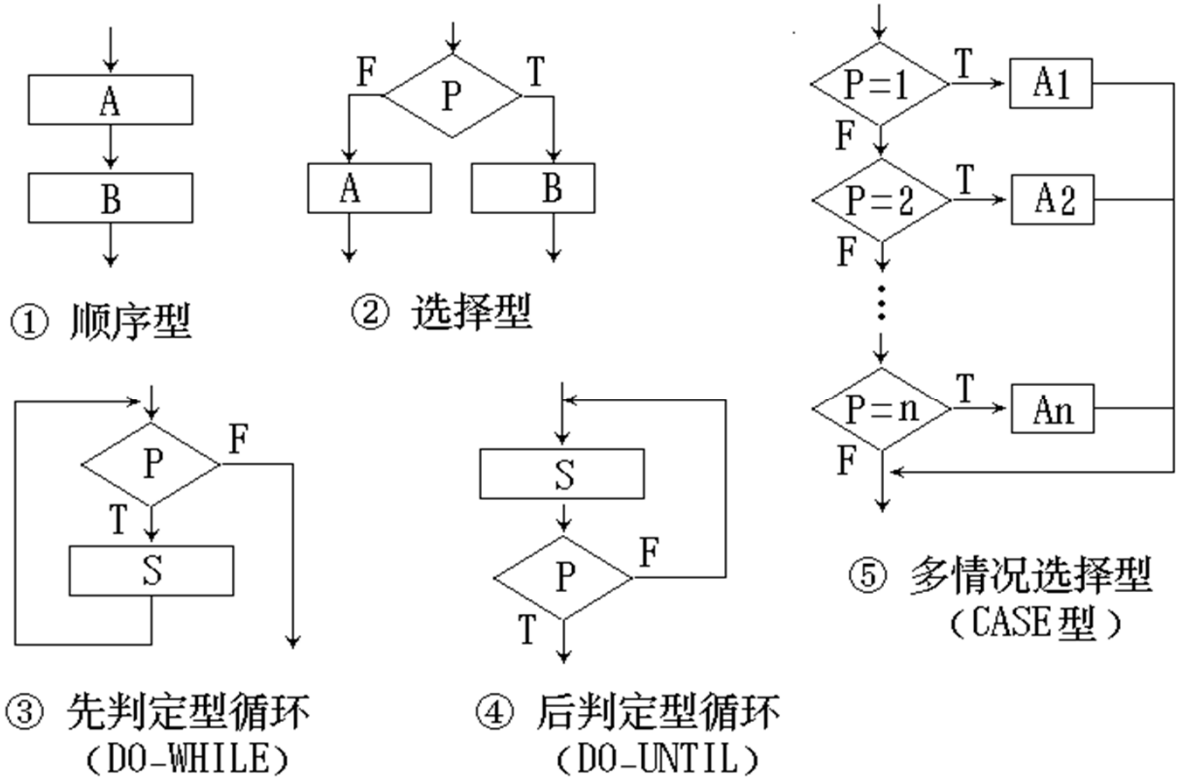
- **清晰第一的设计风格：**
  - 抛弃GOTO语句；
  - 在大多数情况下，应优先考虑程序的清晰度，把效率放在第二位。
- **结构化的控制结构**
  - 尽量只用顺序、选择和循环(DO-WHILE)三种控制接口来设计程序；
  - 每个控制结构尽可能只有单入口和单出口；

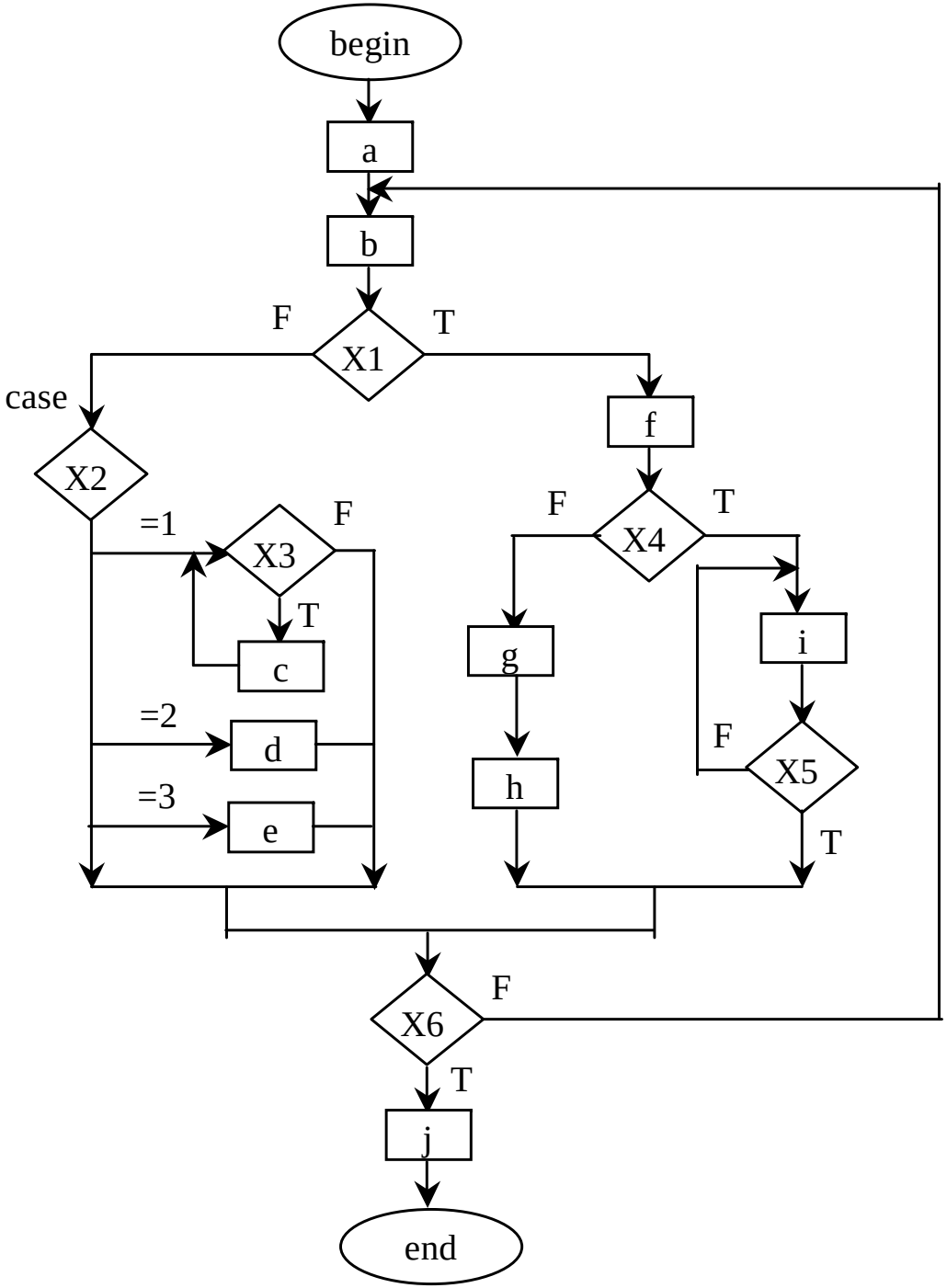
12.8.3 过程设计工具

- 程序流程图(Flow Chart)
- 盒图(N-S图)
- 伪代码与PDL
- HIPO图(Hierarchical Input-Process-Output)

程序流程图

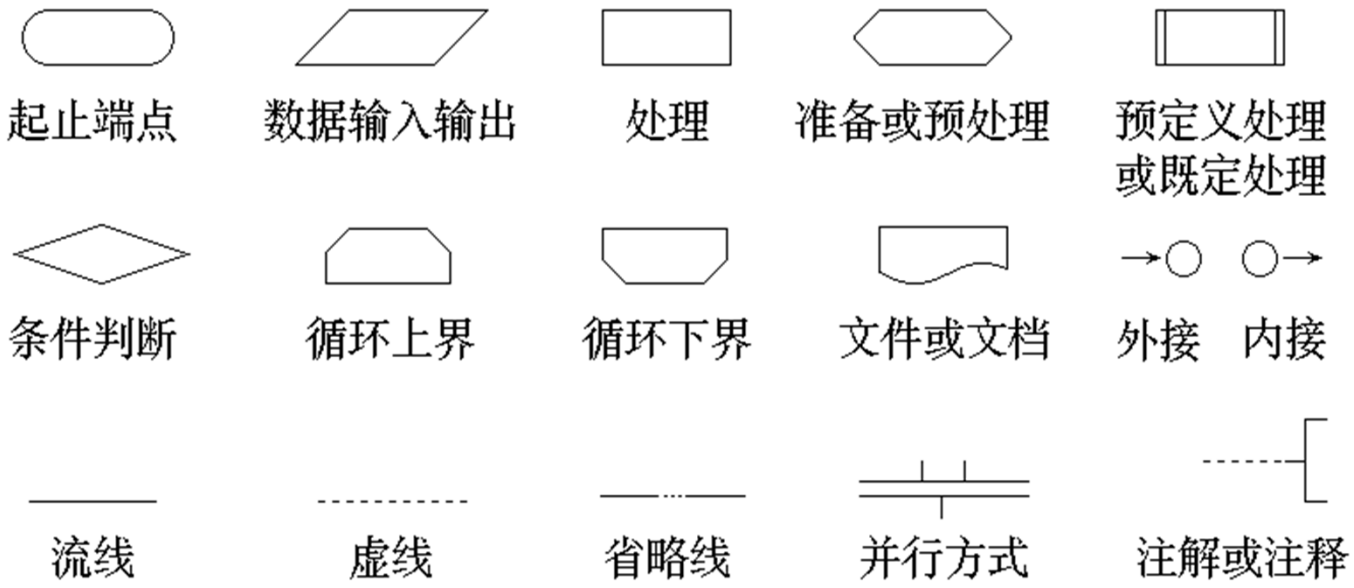
- 程序流程图也称为程序框图，具有五种基本控制结构：



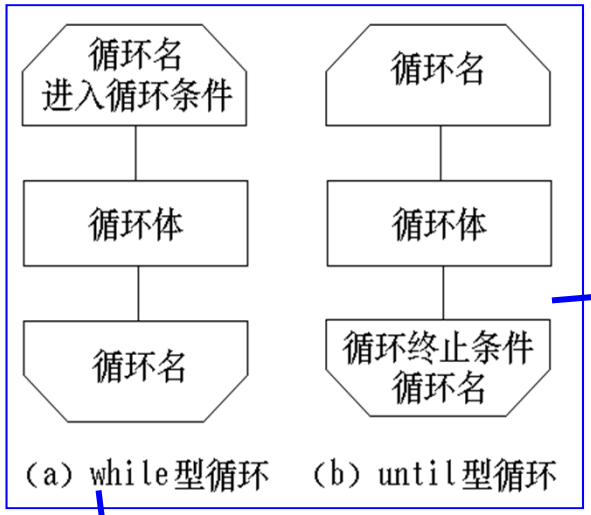


程序流程图

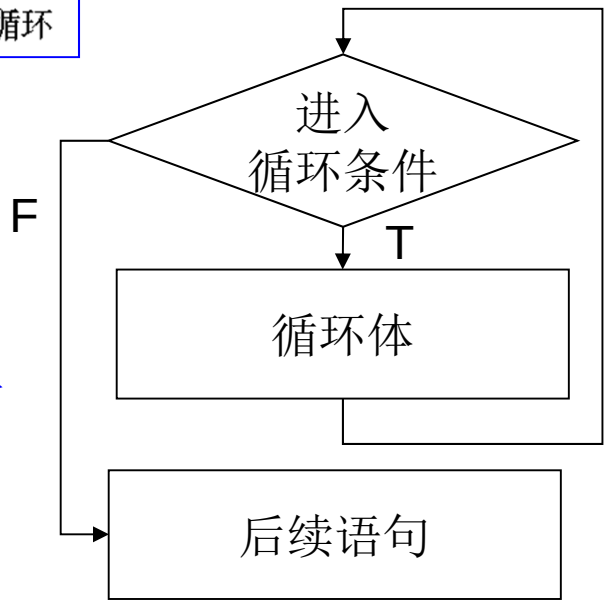
程序流程图的标准符号

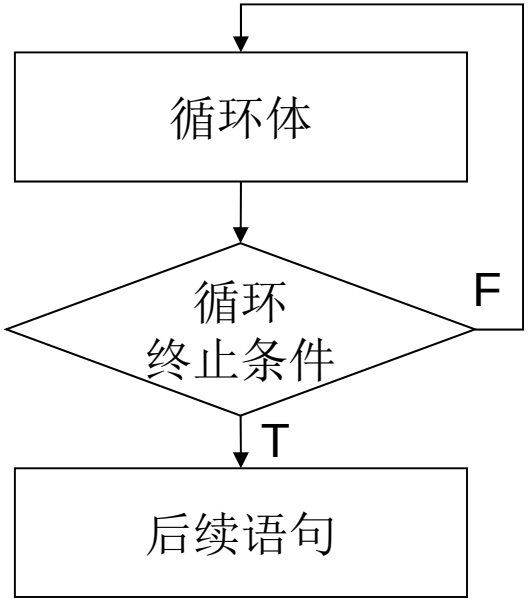

起止端点
数据输入输出
处理
准备或预处理
预定义处理
或既定处理
条件判断
循环上界
循环下界
文件或文档
外接
内接
流线
虚线
省略线
并行方式
注解或注释

程序流程图的标准符号

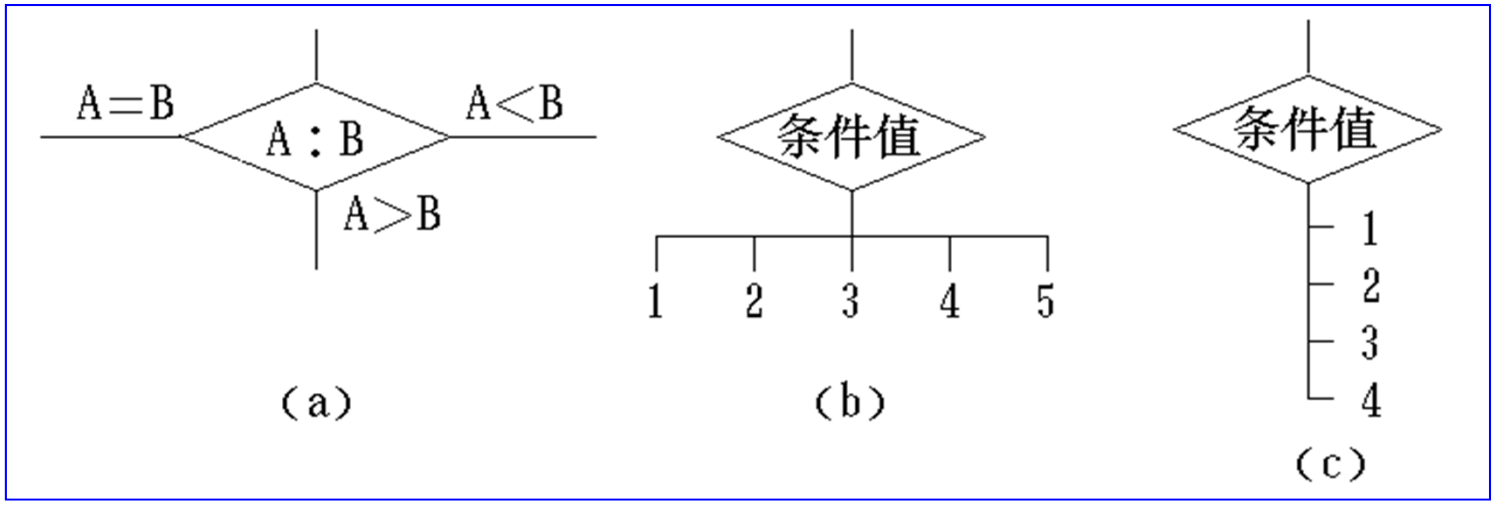


- 通常情况下，将左侧的循环表示方法转换为分支结构**(IF-ELSE-THEN)**。



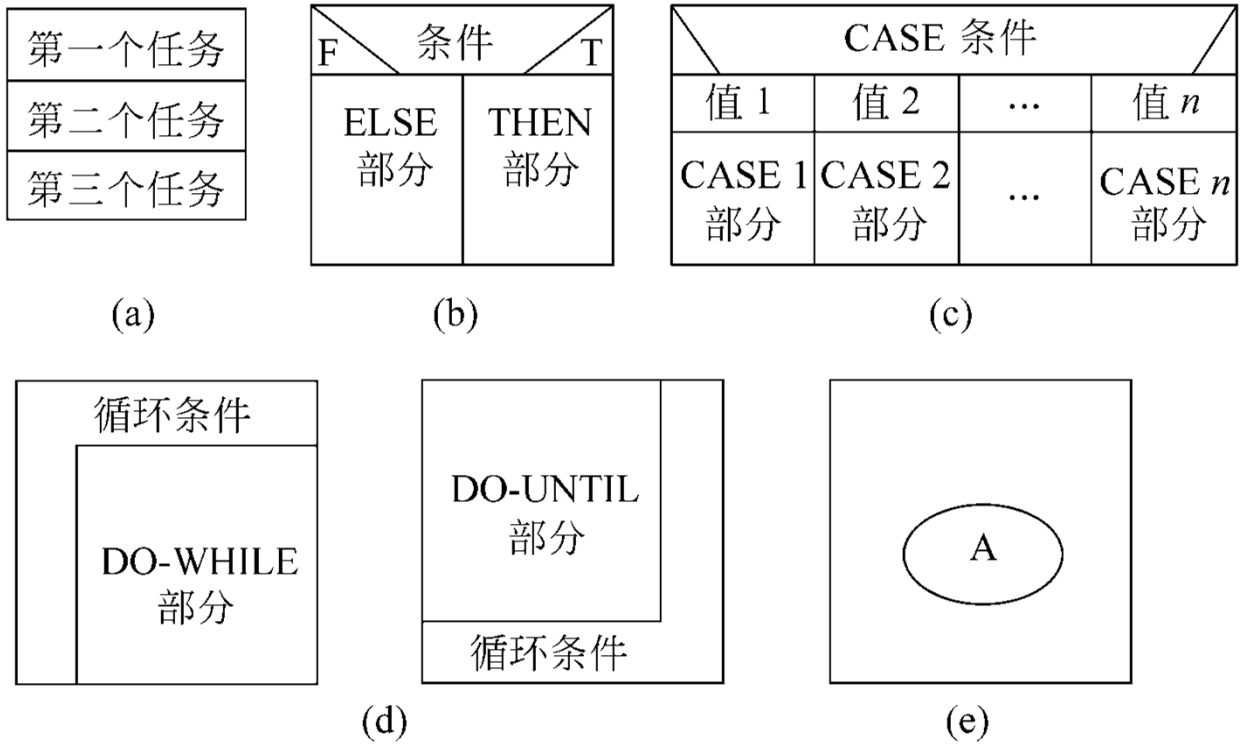


多出口判断

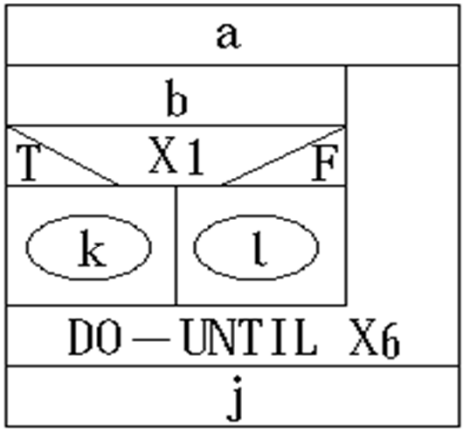


N-S图

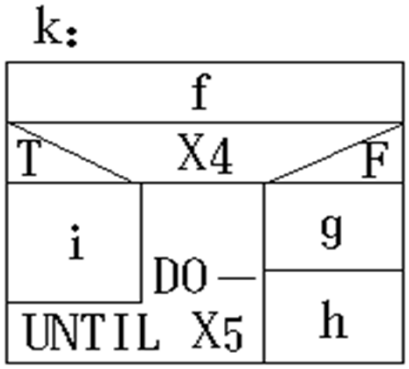
- **N-S**图也叫做盒图。五种基本控制结构由五种图形构件表示。



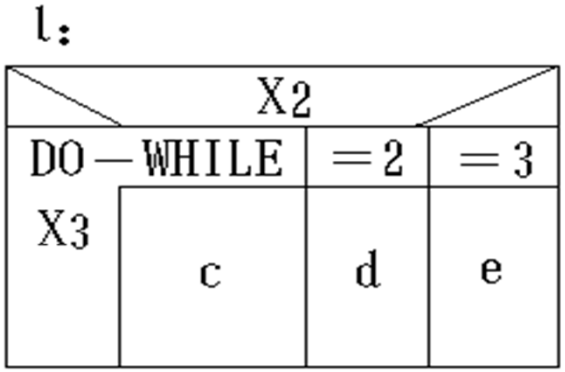
N-S图的嵌套定义形式



(a)

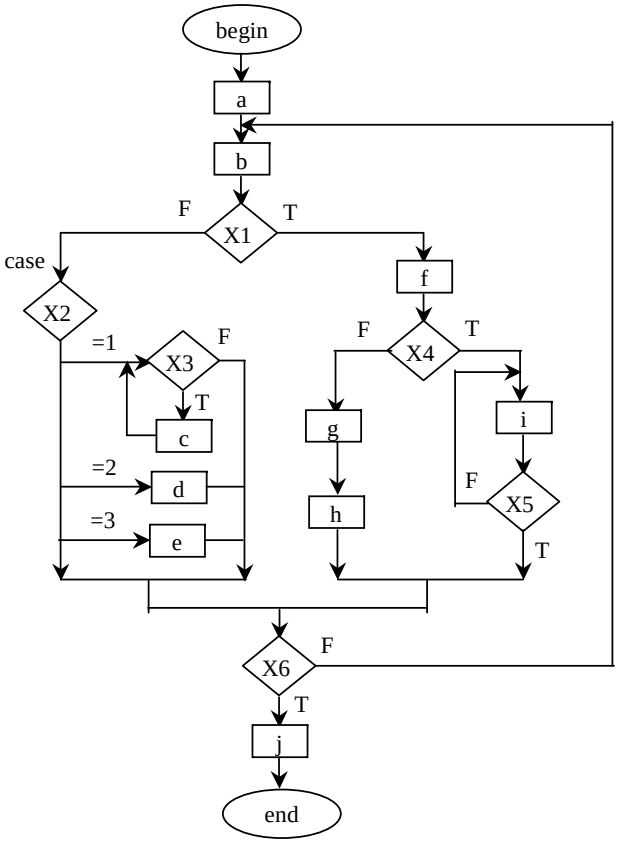


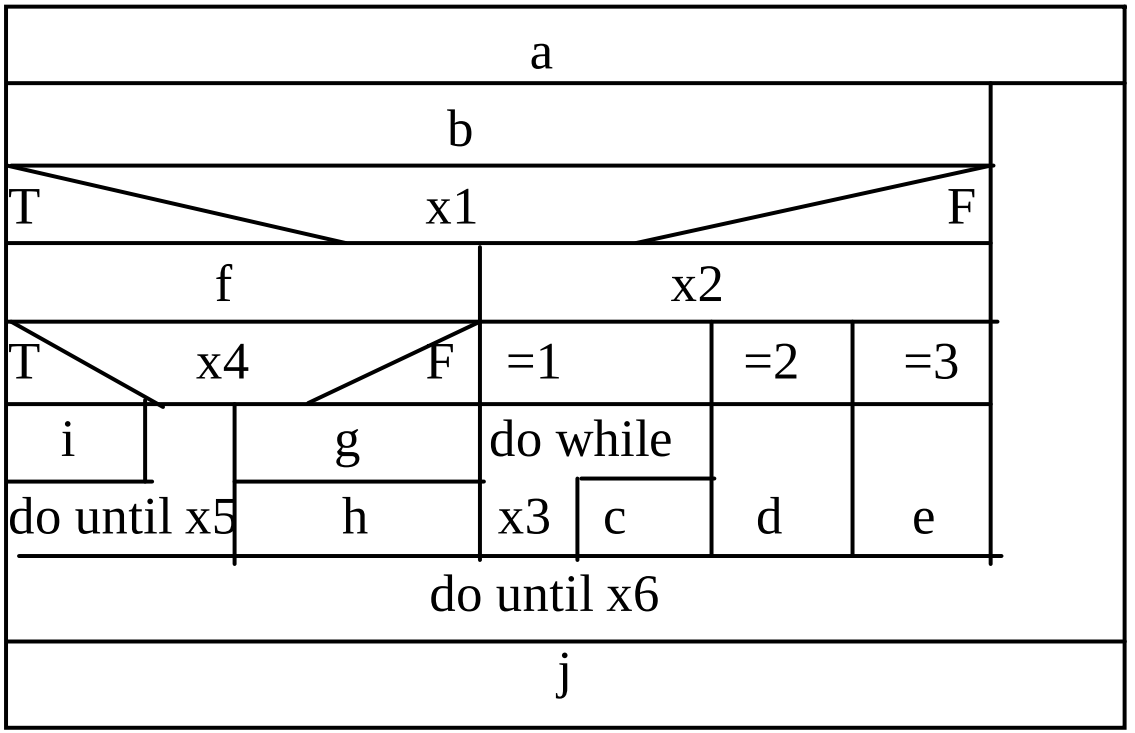
(b)



(c)

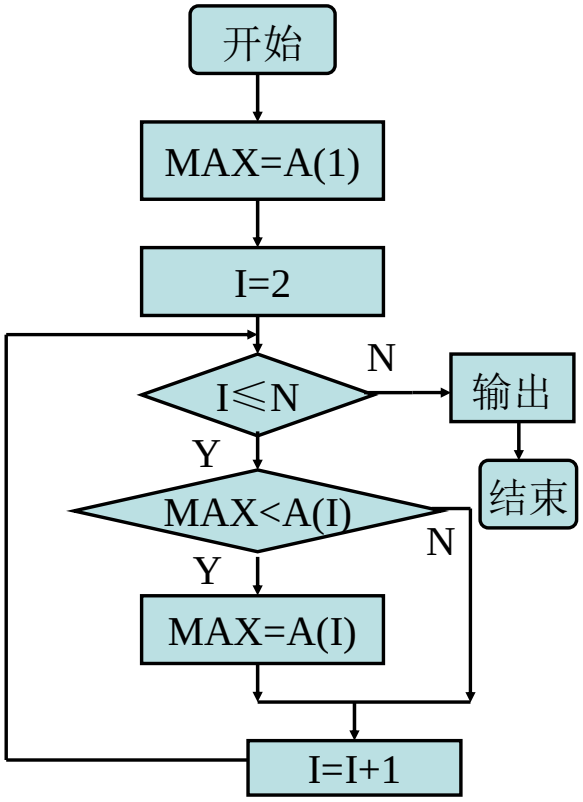
N-S图示例


begin
a
b
F
X1
T
case
X2
=1
X3
F
T
c
=2
d
=3
e
f
F
X4
T
g
h
i
F
X5
T
F
X6
T
j
end

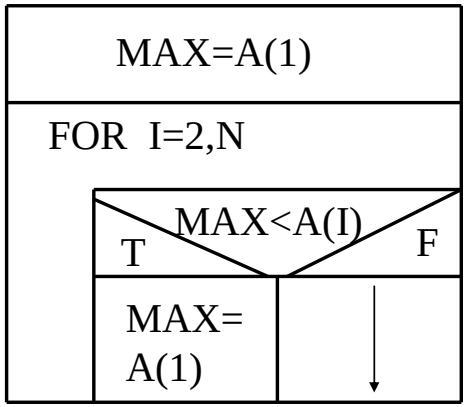

a
b
T
x1
F
f
x2
T
x4
F
=1
=2
=3
i
g
do while
do until x5
h
x3
c
d
e
do until x6
j

流程图与N-S图

- 某一功能模块的输入：数组**A (A[1], A[1],…, A[N])**，计算其最大值**MAX**并输出；
- 绘制其程序流程图与**N-S**图。



流程图



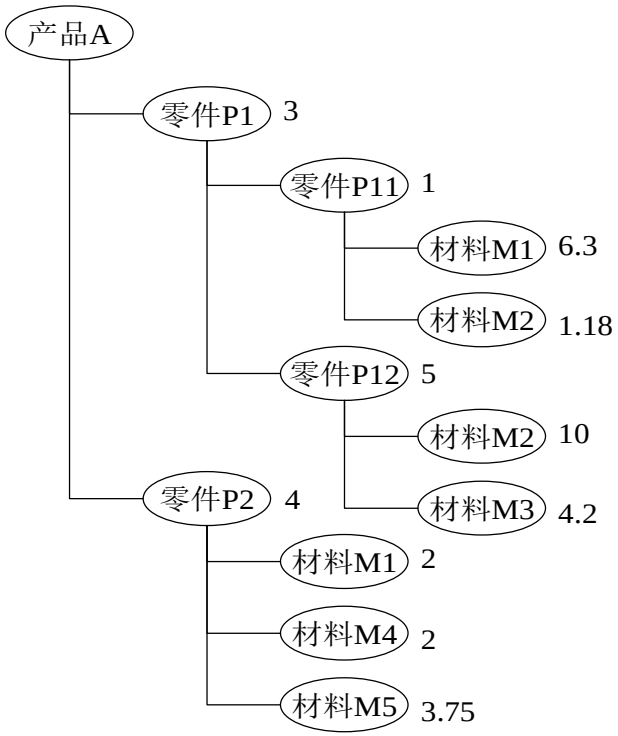
N-S图

12.8.4* 数据结构设计

- 程序流程图刻画了模块内部的算法流转过程；
- 程序=算法+数据结构；
- 因此，需要对程序流程图中所用到的数据结构进行详细设计；
- 使用《数据结构》课程中学到的相关知识加以设计。
  - 链表、队列、堆栈、二叉树、树、图、…

数据结构设计示例

- **产品数据结构(BOM)：**
  - BOM是一个树型结构，其根节点代表“产品”，叶结点代表“原材料”，其它中间层次的节点则代表“零件”。BOM中连接父子节点之间的边表示“装配关系”。每个节点具有一个字符型属性name，用来表示零件或材料的名称；每个非根节点均具有一个浮点型属性quantity，用来表示“生产1个产品(零件)所需的零件(原材料)的数量”。
  - 下图给出了一个简单的BOM实例，它的含义为：1个产品A由3个零件P1和4个零件P2装配而成；1个零件P1则由1个零件P11和5个零件P12装配而成；生产1个P11需要6.3数量的材料M1和1.18数量的材料M2。以此类推。
  - 需要注意的是，一个零件可能由其它零件装配而成(例如P1由P11和P12组成)，也可能由若干种原材料直接加工得到(例如P2由M1、M4和M5加工得到)。
  - 为降低复杂度，做如下限定：某个零件在一棵BOM树中不可能重复出现，但某种原材料却可能重复出现(例如M1既是P11的构成部分，同时也是P2的构成部分)。



数据结构设计示例

- 采用“孩子链表表示法”定义**BOM**的数据结构：

```
#define MaxTreeSize 100

typedef struct CNode {
    int child;
    struct CNode *next;
} CNode;

typedef struct {
    String name;
    float unitQuantity;
    float totalQuantity;
    CNode *firstchild;
} PTNode;

typedef struct {
    PTNode nodes[MaxTreeSize];
    int n, root;
} CTree;

CTree BOM;
```

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

结束

2013年10月8日