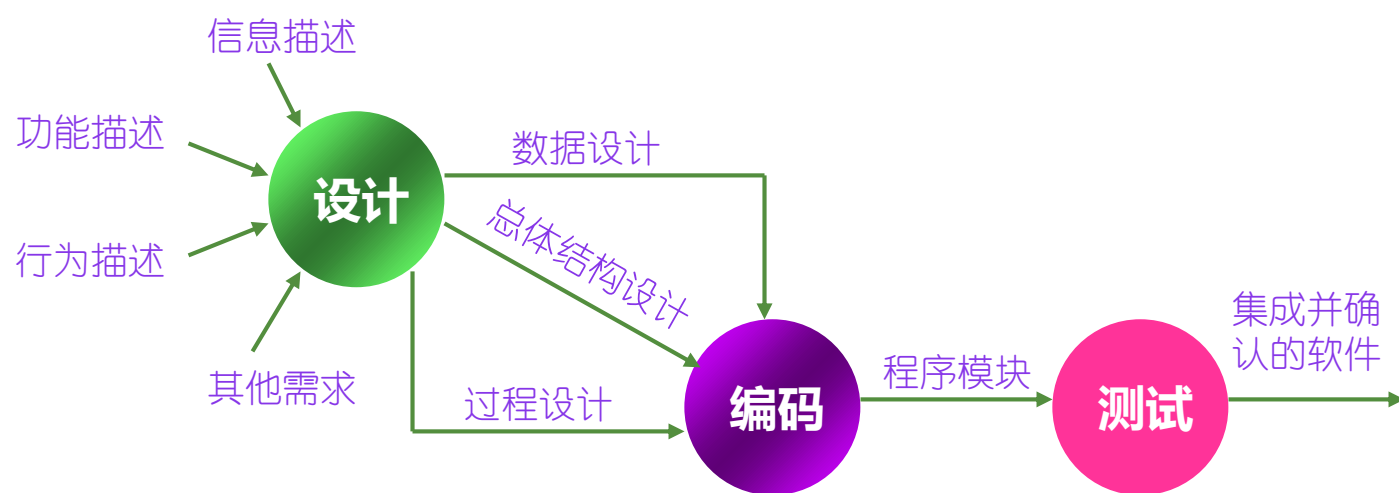


- 1996年前后, 微软的掌上电脑操作系统 Win CE 1.0的失败说明, 如何开发有用的软件非常重要



软件设计任务

制定规范

软件系统  
结构的总体设计

处理方式设计

数据结构设计

可靠性设计

# 第5章 总体设计

总体设计的基本目的就是回答“**概括**地说，系统应该**如何实现**”这个问题，因此，总体设计又称为概要设计或初步设计。

总体设计阶段的另一项重要任务是设计软件的结构，也就是要确定系统中每个程序是由哪些模块组成的，以及这些模块相互间的关系。



# 第5章 总体设计

总体设计的基本目的就是回答“**概括**地说，系统应该**如何实现**”这个问题，因此，总体设计又称为概要设计或初步设计。

总体设计阶段的另一项重要任务是设计软件的结构，也就是要确定系统中每个程序是由哪些模块组成的，以及这些模块相互间的关系。

# 主要内容

- 5. 1 设计过程
- 5. 2 设计原理
- 5. 3 启发规则
- 5. 4 描绘软件结构的图形工具
- 5. 5 面向数据流的设计方法

# 主要内容



5.1 设计过程

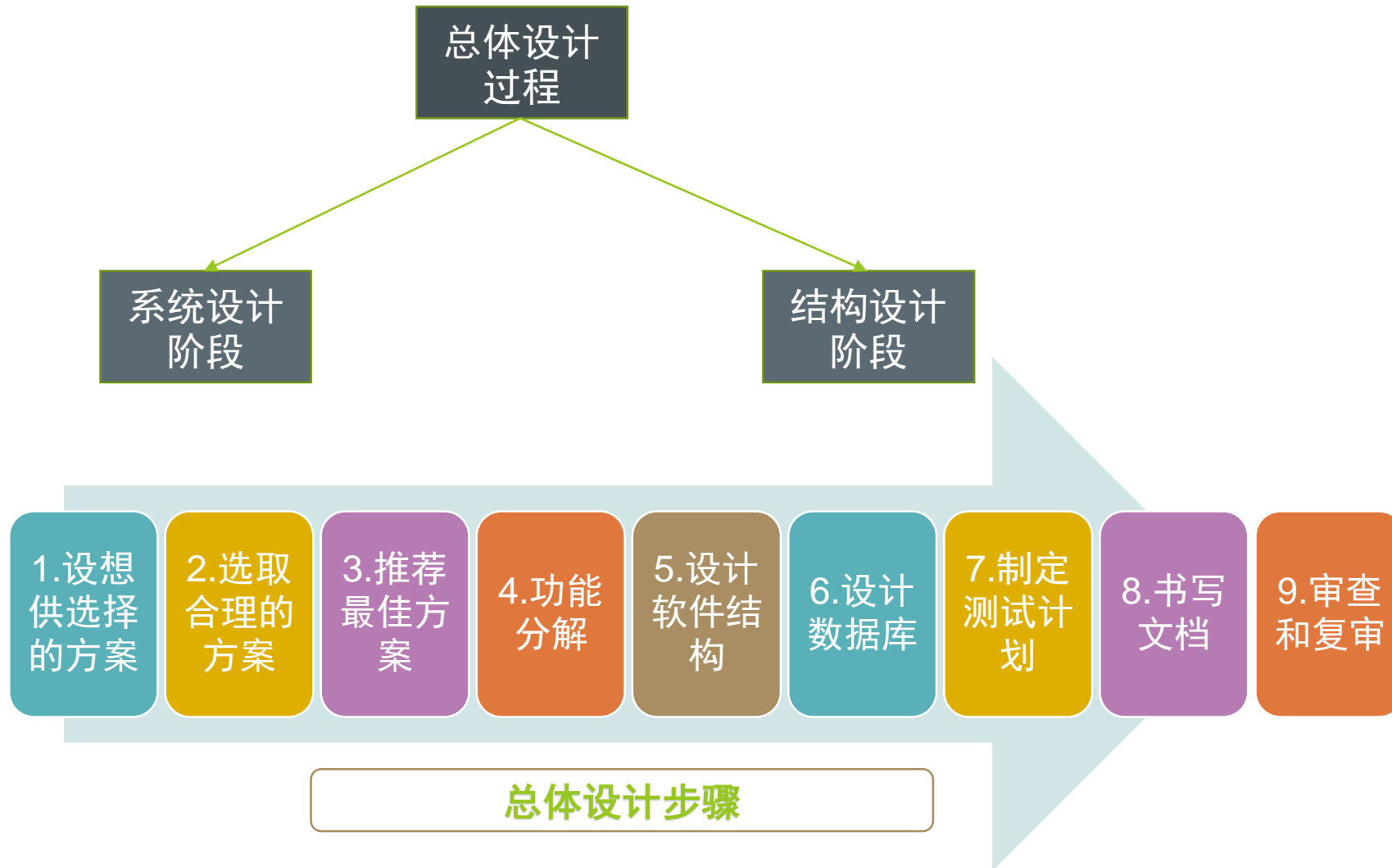
5.2 设计原理

5.3 启发规则

5.4 描绘软件结构的图形工具

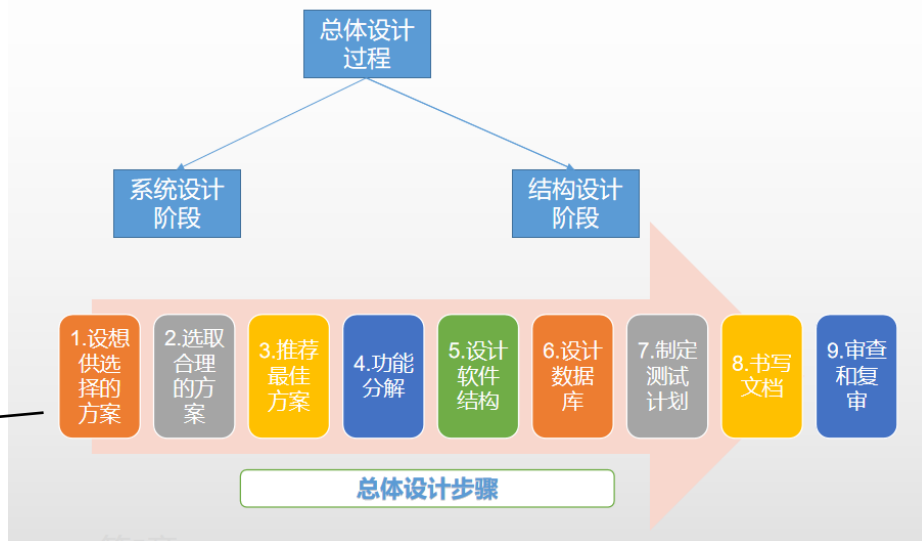
5.5 面向数据流的设计方法

# 5.1 设计过程



# 5.1 设计过程

## 典型的总体设计步骤



### 1. 设想供选择的方案

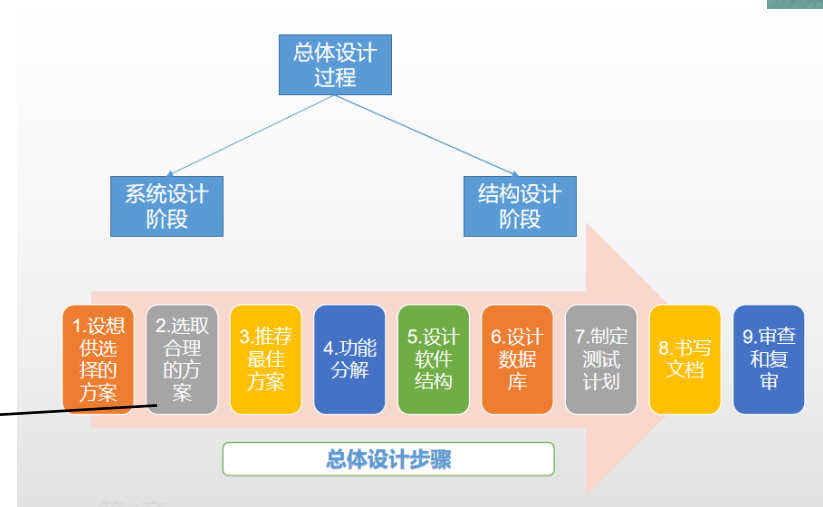
在总体设计阶段分析员应该考虑各种可能的实现方案，并且力求从中选出最佳方案。

需求分析阶段得出的数据流图是总体设计的极好的出发点。

设想供选择的方案的一种常用的方法是，设想把数据流图中的处理分组的各种可能的方法，抛弃在技术上行不通的分组方法(例如，组内不同处理的执行时间不相容)，余下的分组方法代表可能的实现策略，并且可以启示供选择的物理系统。



# 5.1 设计过程



## 2.选取合理的方案

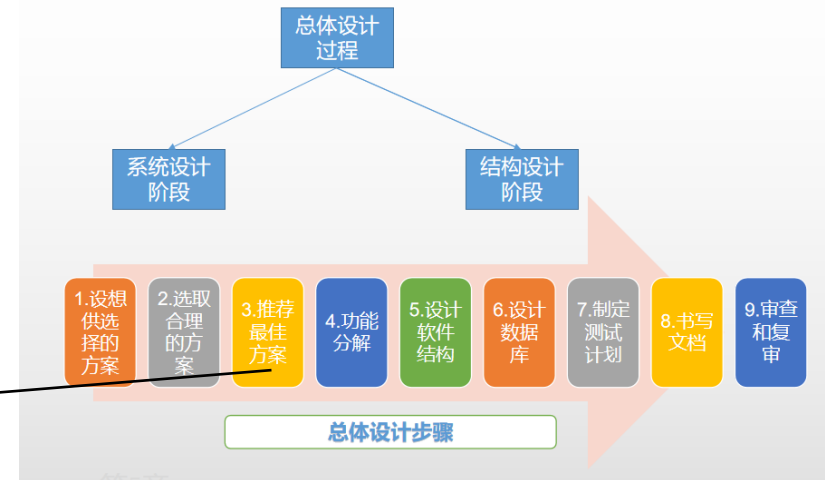
应该从前一步得到的一系列供选择的方案中选取若干个合理的方案，通常至少选取低成本、中等成本和高成本的3种方案。在判断哪些方案合理时应该考虑在问题定义和可行性研究阶段确定的工程规模和目标，有时可能还需要进一步征求用户的意见。

对每个合理的方案，分析员都应该准备下列4份资料。

- (1) 系统流程图。
- (2) 组成系统的物理元素清单。
- (3) 成本/效益分析。
- (4) 实现这个系统的进度计划。



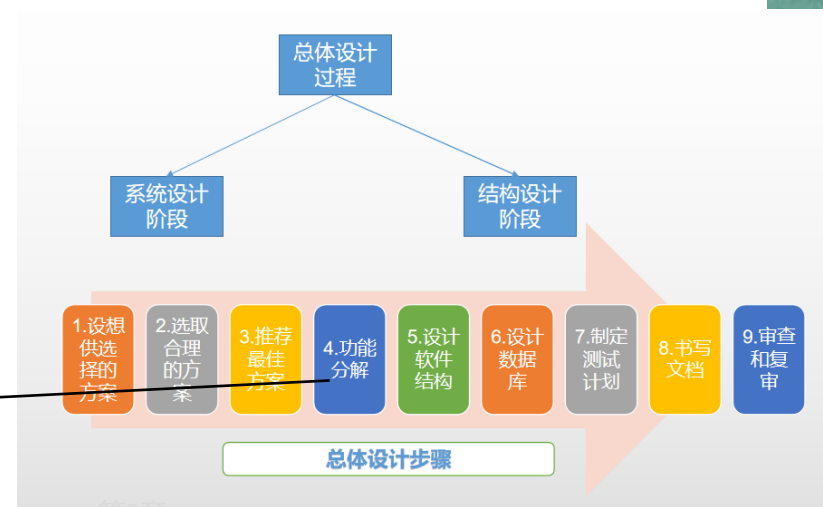
# 5.1 设计过程



## 3.推荐最佳方案

用户和有关的技术专家应该认真审查分析员所推荐的**最佳系统**，如果该系统确实符合用户的需要，并且是在现有条件下完全能够实现的，则应该提请使用**部门负责人**进一步**审批**。在使用部门的负责人也接受了分析员所推荐的方案之后，将进入总体设计过程的下一个重要阶段——结构设计。

# 5.1 设计过程

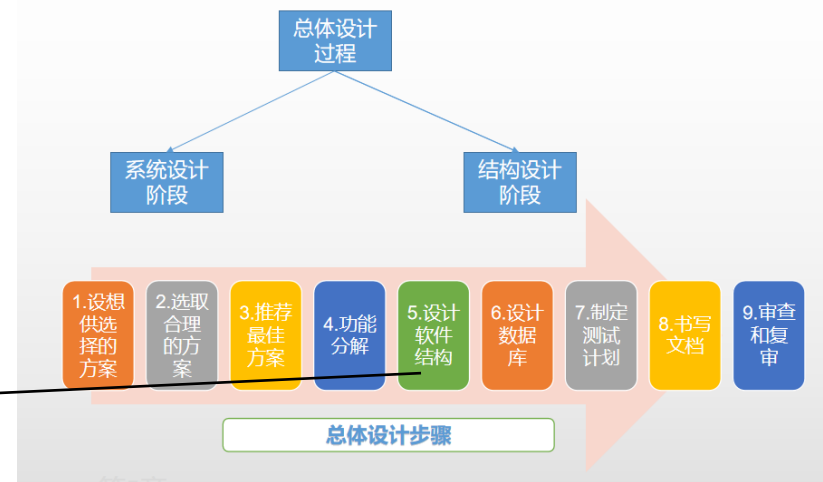


## 4.功能分解

为了最终实现目标系统，必须设计出组成这个系统的所有程序和文件(或数据库)。对程序(特别是复杂的大型程序)的设计，通常分为两个阶段完成：首先进行结构设计，然后进行过程设计。

为确定软件结构，首先需要从实现角度把复杂的功能进一步分解。分析员结合算法描述仔细分析数据流图中的每个处理，如果一个处理的功能过分复杂，必须把它的功能适当地分解成一系列比较简单的功能。

# 5.1 设计过程



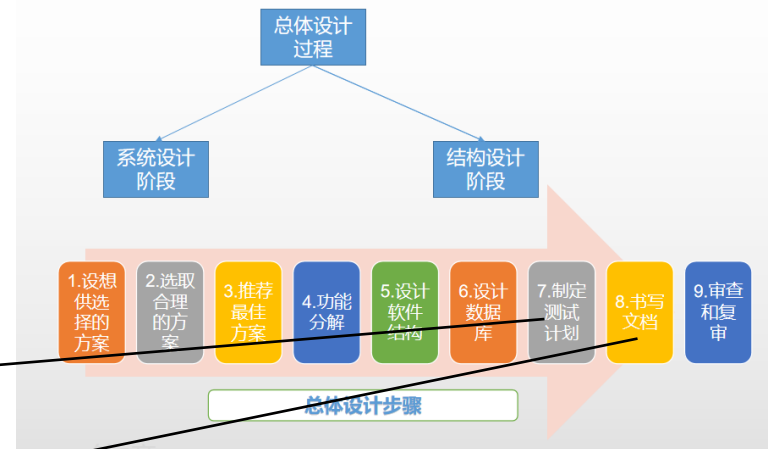
## 5. 设计软件结构

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**，顶层模块调用它的下层模块以实现程序的完整功能，每个下层模块再调用更下层的模块，完成程序的一个子功能，最下层的模块完成最具体的功能。

## 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。

# 5.1 设计过程



## 7.制定测试计划

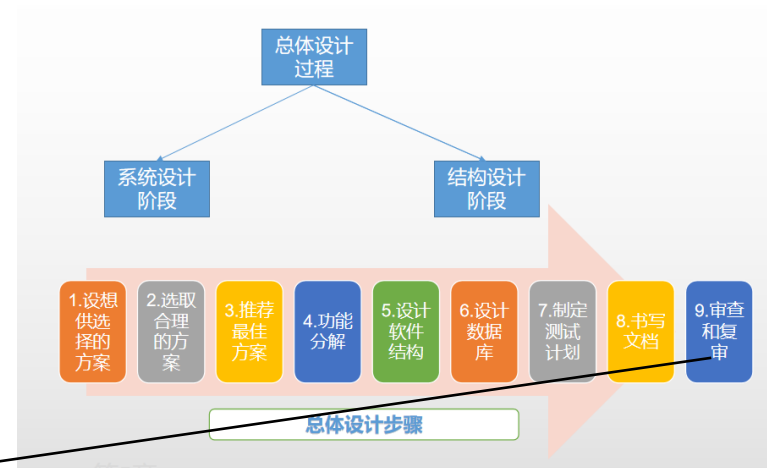
在软件开发的早期阶段考虑测试问题，能促使软件设计人员在设计时注意提高软件的可测试性。

## 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) 测试计划包括测试策略，测试方案，预期的测试结果，测试进度计划等
- (4) 详细的实现计划
- (5) 数据库设计结果

# 5.1 设计过程



## 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，在技术审查通过之后再由客户从管理角度进行复审。

# 主要内容

5.1 设计过程



5.2 设计原理

5.3 启发规则

5.4 描绘软件结构的图形工具

5.5 面向数据流的设计方法

# 5.2 设计原理

## 5.2.1 模块化

一般指用一个名字  
调用的一段程序

**模块**是由边界元素限定的相邻程序元素（例如，数据说明，可执行的语句）的序列，而且有一个总体标识符代表它。**模块**是构成程序的基本构件。

**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。

逻辑

功能

状态



## 5.2 设计原理

- 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，是软件应该具备的唯一属性。

- 如果一个大型程序仅由一个模块组成，很难被人理解。

设函数  $C(x)$  定义问题  $x$  的复杂程度，函数  $E(x)$  定义解决问题  $x$  需要的工作量（时间）。对于两个问题  $P1$  和  $P2$ ，如果：

$$C(P1) > C(P2)$$

那么  $E(P1) > E(P2)$

- 根据解决问题的经验，有一个规律是：

$$C(P1+P2) > C(P1) + C(P2)$$

于是有  $E(P1+P2) > E(P1) + E(P2)$

这个不等式导致“各个击破”的结论——把复杂的问题分解成许多容易解决的小问题，原来的问题也就容易解决了。这就是模块化的根据。

“软件工程基本定理”

## 5.2 设计原理

G.A. Miller

人类信息处理的能力是有限的，分辨或记忆同类信息的数量一般不能超过5---9项（即 $7 \pm 2$ ）

这说明，把一个复杂的问题分解为若干个较易管理的小片，总比对问题作“一揽子解决”更容易。

奇妙的数字  $7 \pm 2$ ，人类信息处理能力的限度

- 分解：是人们处理复杂问题常用的方法。如，在需求分析阶段中靠分解来画分层DFD图的；
- 在设计阶段中，可以用它来实现模块化设计。所以说，分解是模块化设计的重要的指导思想。
- **问题：**如果无限地分割软件，最后为了开发软件而需要的工作量也就小得可以忽略了？

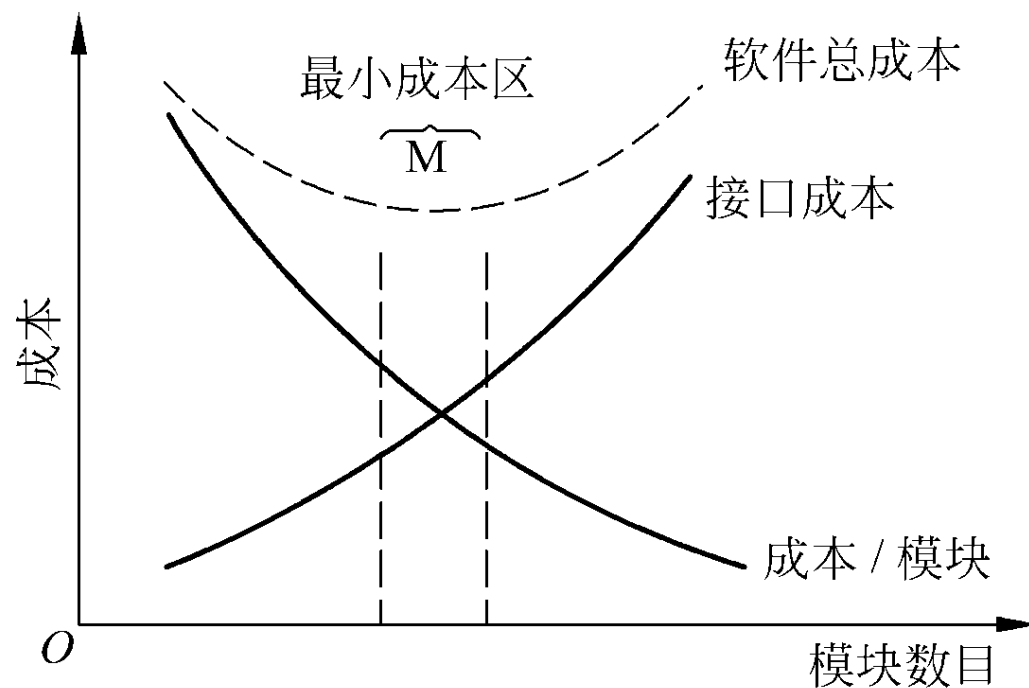
## 5.2 设计原理

事实上，还有另一个因素在起作用，从而使得上述结论不能成立。

如图，当模块数目增加时每个模块的规模将减小，开发单个模块需要的成本(工作量)确实减少了；但是，随着模块数目增加，设计模块间**接口**所需要的工作量也将增加。

根据这两个因素，得出了图中的总成本曲线。每个程序都相应地有一个最适当的模块数目 $M$ ，使得系统的开发成本最小。

**问题：** $M$ 的值如何确认？



## 5.2 设计原理

- 目前还不能精确地决定M的数值，但是在考虑模块化的时候总成本曲线确实是有用的指南。
- 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。
- 模块化也有助于软件开发工程的组织管理，一个复杂的大型程序可以由许多程序员分工编写不同的模块，并且可以进一步分配技术熟练的程序员编写困难的模块。

# 5.2 设计原理

## 5.2.2 抽象

- 人们在实践中认识到，在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。或者说抽象就是抽出事物的**本质特性**而暂时不考虑它们的细节。
- 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**。



# 5.2 设计原理

## 5.2.3 逐步求精

逐步求精定义为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。

逐步求精最初是由Niklaus Wirth提出的一种自顶向下的设计策略。按照这种设计策略，程序的体系结构是通过逐步精化处理过程的层次而设计出来的。通过逐步分解对功能的宏观陈述而开发出层次结构，直至最终得出用程序设计语言表达的程序。

- 求精实际上是细化过程。
- 抽象与求精是一对互补的概念。

# 5.2 设计原理

## 5.2.4 信息隐藏和局部化

**信息隐藏**原理：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**是指把一些关系密切的软件元素物理地放得彼此靠近。

**思考：为什么要进行信息隐藏？**

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(也就是“看”不见的)，在**在测试期间和需要修改软件的时候**，由于疏忽而引入的错误就很少有可能传播到软件的其他部分。



## 5.2 设计原理

### 5.2.5 模块独立

模块的独立性很重要，因为：

- ① 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。
- ② 独立的模块比较容易测试和维护。

**模块的独立性是指软件系统中每个模块只涉及软件要求的具体的子功能，而和软件系统中其他模块的接口是简单的。**

**耦合**

模块之间的  
相对独立性的  
度量。

**内聚**

模块功能强  
度的度量。

## 5.2 设计原理

### 1 耦合

- 耦合是软件结构内不同模块彼此之间相互依赖（连接）的紧密程度。耦合强弱取决于模块间接口的复杂程度，进入或访问一个模块的点，以及通过接口的数据。
- 模块耦合主要有数据耦合、控制耦合、特征耦合、公共环境耦合和内容耦合

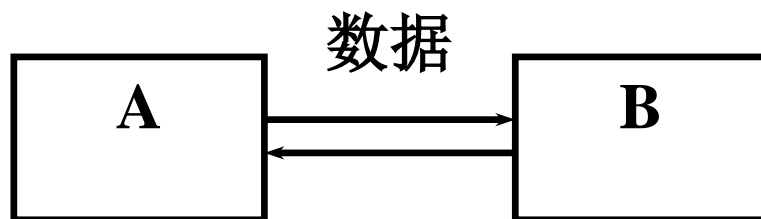


## 5.2 设计原理

### ① 数据耦合

两个模块彼此间通过参数交换信息，而且交换的信息仅仅是数据，那么这种耦合称为数据耦合。

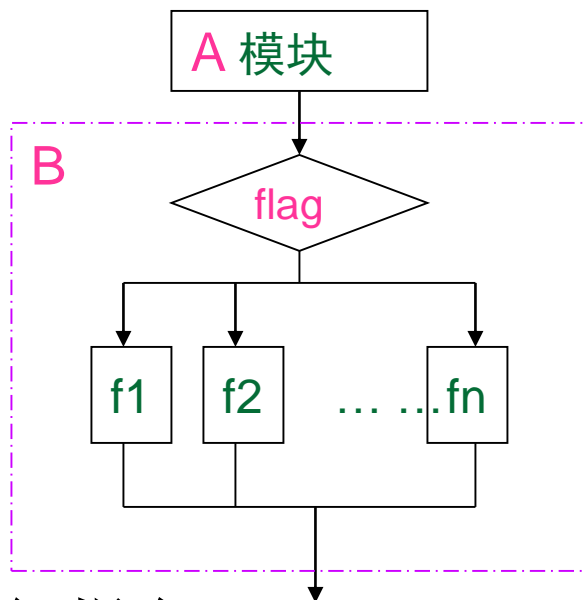
**数据耦合是最低程度的耦合。**



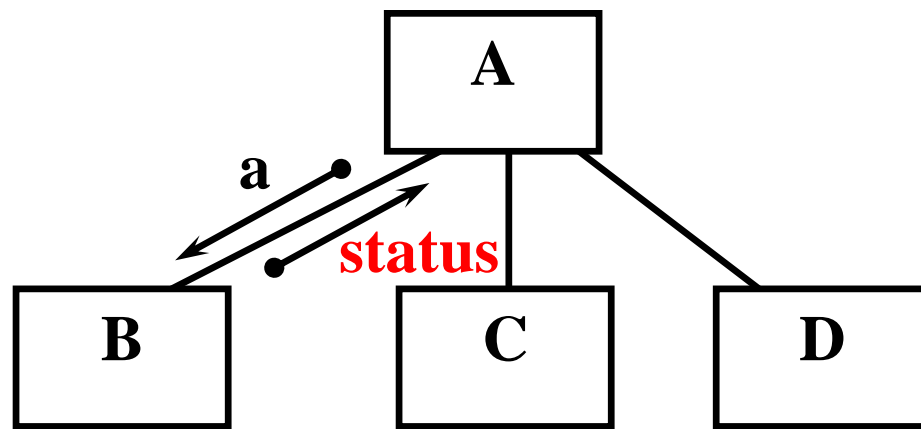
(1) 数据耦合

## ② 控制耦合

传递的信息中有控制信息(尽管有时这种控制信息以数据的形式出现), 则这种耦合称为控制耦合。控制耦合是中等程度的耦合。



图中模块A的内部处理程序判断是执行C还是执行D, 要取决于模块B传来的信息状态 (**Status**)。



(2) 控制耦合

## ③ 特征耦合

当把整个数据结构作为参数传递而被调用的模块只需要使用其中一部分数据元素时, 就出现了特征耦合。

#### ④ 公共环境耦合

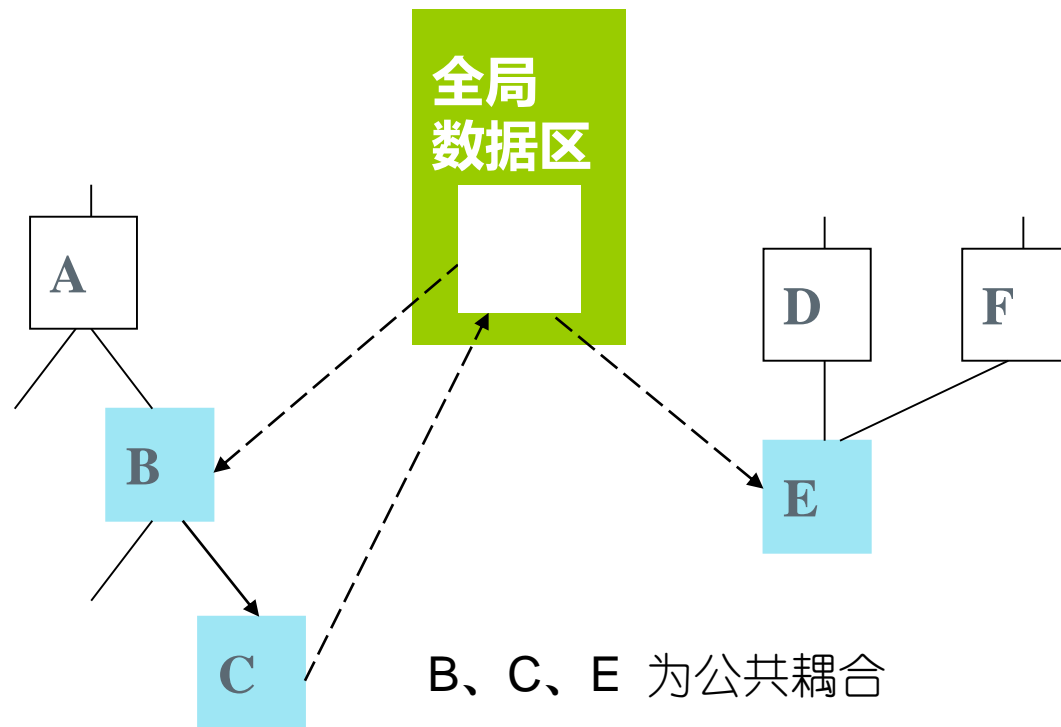
- 当两个或多个模块通过一个公共数据环境相互作用时，它们之间的耦合称为公共环境耦合。
- 公共环境可以是全程变量、共享的通信区、内存的公共覆盖区、任何存储介质上的文件、物理设备等。公共环境耦合的复杂程度随耦合的模块个数而变化，当耦合的模块个数增加时复杂程度显著增加。

图中存在公用耦合，假设模块A、C、E都存取全程数据区（如公用一个磁盘文件）中的一个数据项。

如果A模块读取该项数据，然后调用C模块对该项重新计算，并进行数据更新。

如果此时C模块错误地更新了该项数据，在往下的处理中模块E读该数据项时出现错误。

表面上看，问题由模块E产生，实际上由模块C引起。



## 两个模块的公共环境耦合：

只有两个模块有公共环境，耦合有下面两种可能。

(1) 一个模块往公共环境送数据，另一个模块从公共环境取数据。这是数据耦合的一种形式，是比较松散的耦合。

(2) 两个模块都既往公共环境送数据又从里面取数据，这种耦合比较紧密，介于数据耦合和控制耦合之间。

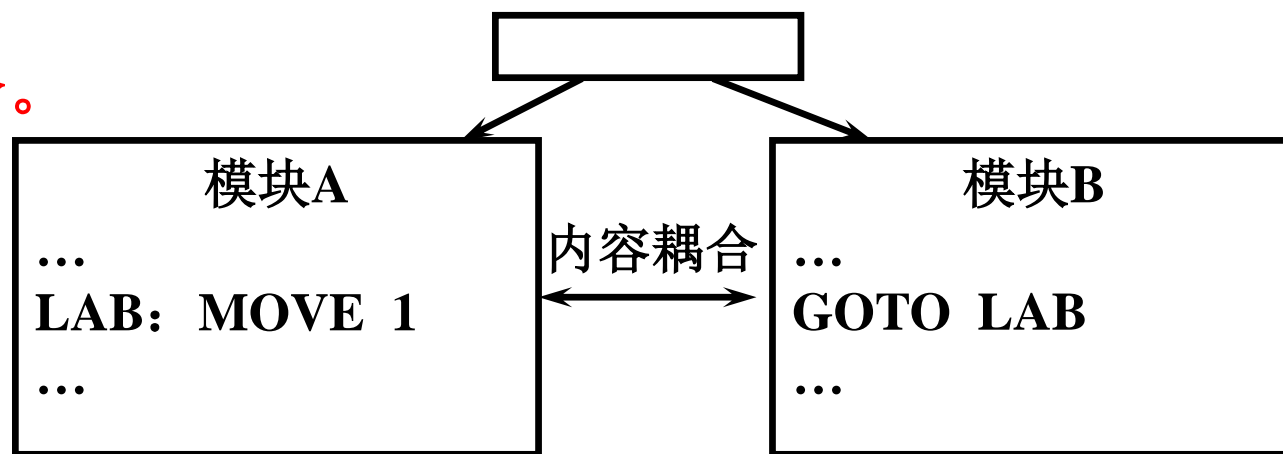


## ⑤ 内容耦合

■ 最高程度的耦合是内容耦合。如果出现下列情况之一，两个模块间就发生了内容耦合。

- 一个模块访问另一个模块的内部数据。
- 一个模块不通过正常入口而转到另一个模块的内部。
- 两个模块有一部分程序代码重叠(只可能出现在汇编程序中)。
- 一个模块有多个入口(这意味着一个模块有几种功能)。

应该坚决避免使用内容耦合。



(4) 内容耦合



- 软件设计应追求尽可能松散耦合，避免强耦合，这样模块间的联系就越小，模块的独立性就越强，对模块的测试、维护就越容易。
- 因此建议：尽量使用数据耦合，少用控制耦合，限制公用耦合，完全不用内容耦合。

## 5.2 设计原理

### 2) 内聚

- 内聚标志着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**。
- 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。
- 内聚分为三大类低内聚、中内聚和高内聚



## ① 低内聚

一个模块完成一组任务，这些任务彼此间即使有关系，关系也是很松散的，就叫做**偶然内聚**。

一个模块完成的任务在逻辑上属于相同或相似的一类，则称为**逻辑内聚**。

一个模块包含的任务必须在同一段时间内执行，就叫**时间内聚**。

## ② 中内聚

一个模块内的处理元素是相关的，而且必须以特定次序执行，则称为过程内聚。

模块中所有元素都使用同一个输入数据和(或)产生同一个输出数据，则称为**通信内聚**。

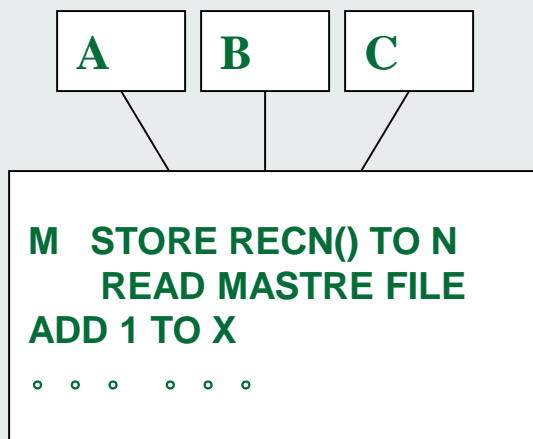
## ③ 高内聚

一个模块内的处理元素和同一个功能密切相关，而且这些处理必须顺序执行(通常一个处理元素的输出数据作为下一个处理元素的输入数据)，则称为**顺序内聚**。

模块内所有处理元素属于一个整体，完成一个单一的功能，则称为**功能内聚**。  
功能内聚是最高程度的内聚。

# 1. 偶然性内聚

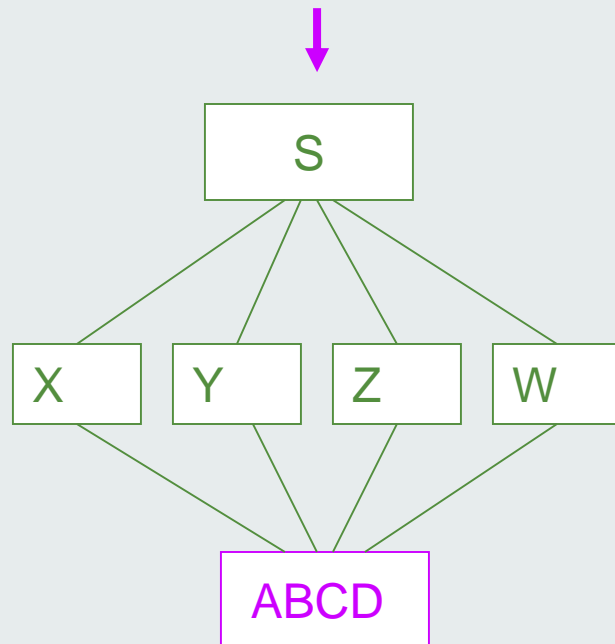
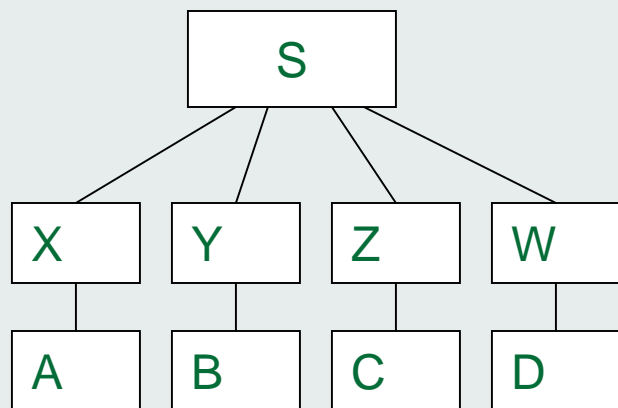
当模块内各部之间没有联系，或者即使有联系，这种联系也很松散。则称这种模块为巧合内聚模块。



例如，一些没有任何联系的语句可能在许多模块中重复使用，程序员为了节省存储，则把它们抽出来组成一个新的模块。这个模块就是偶然性内聚模块

偶然内聚是最差的一种内聚。

## 2. 逻辑性内聚



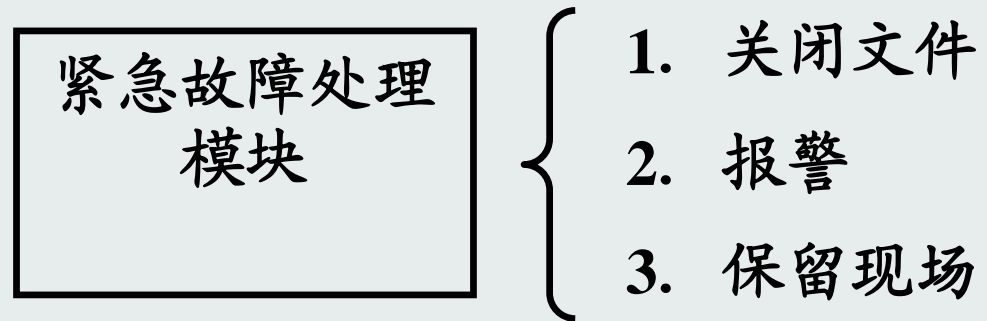
这种模块是把几种功能组合在一起，每次调用时，则由传递给模块的**判定参数**来确定该模块应执行哪一种功能。

- 例如，图中A,B,C,D是用于计算全班学生平均分,最高分,最低分,方差的模块，无论是计算那种分数，都要经过：  
**读入全班学生分数**----->**进行计算**---->**输出计算结果**等步骤
- 除了中间的一步须按不同的方法计算外，**前、后这两步都是相同的**
- 把这些在逻辑相似的功能放在一个模块ABCD中，就可省去程序中的重复部分，这种模块是单入口多功能模块。
- 具有逻辑性内聚的模块，通常是由若干个逻辑功能相似的成分组成。
- 逻辑内聚模块比偶然内聚模块的内聚程度要高。因为它表明了各部分之间在功能上的想相关关系。但它所执行的不是一种功能，而是执行若干功能中的一种。因此它**不易修改**。
- 另外，当调用时需要进行**控制参数的传递**，在多个关联性功能中选择执行某一个功能，这就增加了模块间的耦合程度。

### 3. 时间内聚 (Temporal Cohesion)

如果一个模块包含的任务必须在同一段时间内执行，称为**时间内聚**。也称为**瞬时内聚**。

- 例如，完成各种初始化工作的模块，或者处理故障的模块都存在时间内聚。
- 如图，在“紧急故障处理模块”中，“关闭文件”、“报警”、“保留现场”等任务都必须无中断地同时处理。



时间内聚示例

## 4. 过程内聚 (Procedural Cohesion)

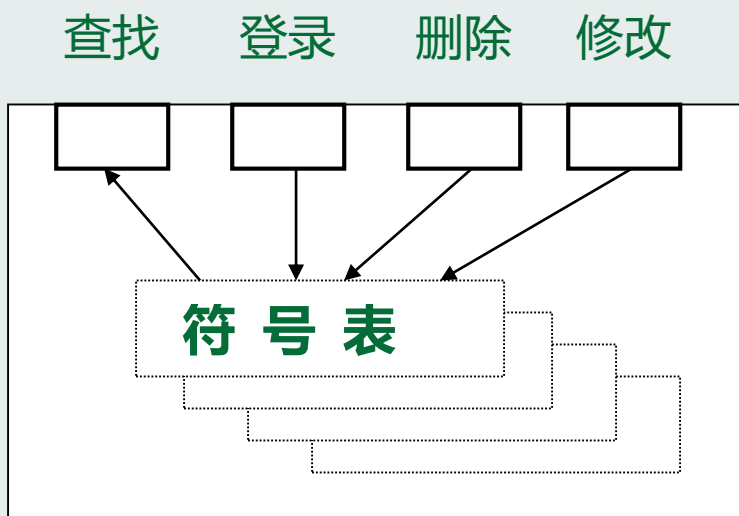
- 如果一个模块内的处理元素是相关的，而且必须以**特定的次序**执行，称为过程内聚。
- 过程内聚与顺序内聚的区别是：
  - 顺序内聚中是数据流从一个处理单元流到另一个处理单元，而过程内聚是控制流从一个动作流向另一个动作。



## 5. 通信内聚

如果一个模块中所有处理元素都使用同一个输入数据和（或）产生同一个输出数据，称为通信内聚。

这种模块能完成多个功能，各个功能都在同一数据结构上操作，每一项功能有一个唯一的入口点。



如图，模块A的处理单元将根据同一个数据文件FILE的数据产生不同的表格，因此它存在通信内聚。  
通信内聚有时也称为数据内聚。

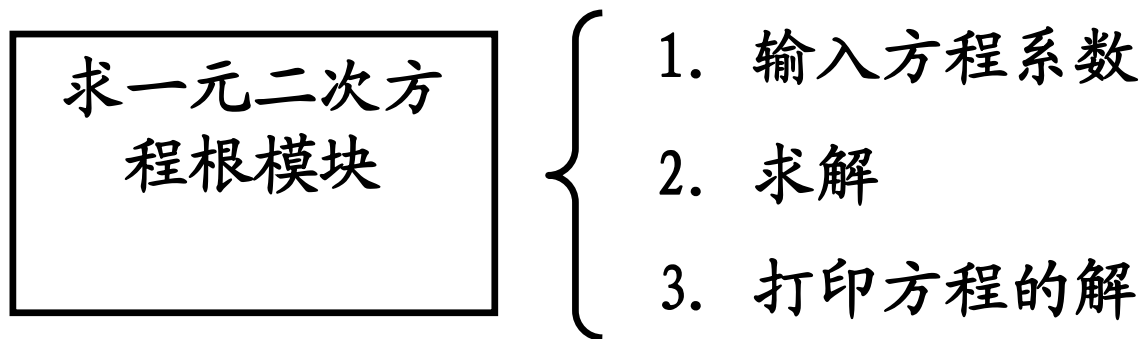


通信内聚示例

## 6. 顺序内聚

• **Sequential Cohesion**, 如果一个模块内处理元素和同一个功能密切相关, 而且这些处理元素必须顺序执行, 则称为顺序内聚。

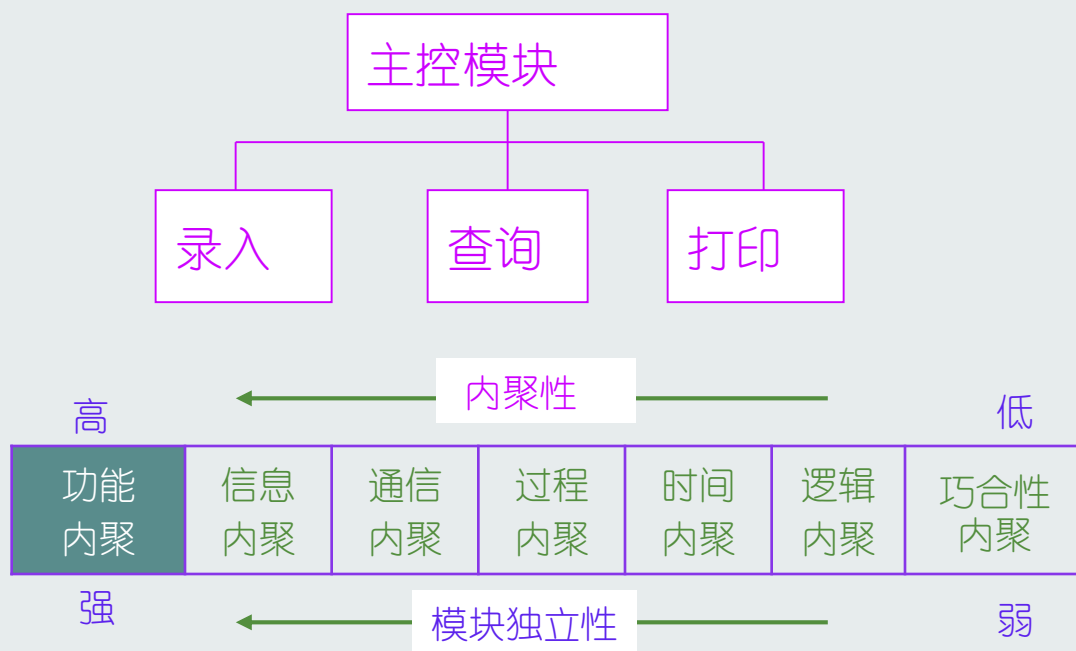
- 如图, 一个求一元二次方程根的模块由三个处理元素组成, 该模块中存在顺序内聚。
- 通常, 顺序内聚中一个处理元素的输出是另一个处理元素的输入。



顺序内聚示例

## 7. 功能性内聚

如果一个模块内所有成分都完成一个功能，而且仅完成一个功能，则称这样的模块为功能模块。



功能性内聚是最高程度的内聚，它的优点是功能明确，模块间耦合简单。

## 示例1 以下程序属于哪种耦合？

例：在宠物商店的例子中，假设有一个的产品类Product，该类有一个用来记录宠物单价的实例变量unitPrice，如果该变量是Public的，那么其它类（如订单类）就可以轻易的修改该变量，甚至将单价改为一个负数。代码如下所示

```
public class Product {  
    public float unitPrice;  
    ...  
}  
public class Order {  
    private Product myProduct=new Product();  
    public void setItem() {  
        myProduct.unitPrice = -100;  
    }  
}
```

**// 出现内容耦合**

- Public类和Order类之间构成了内容耦合。
- 为了避免这种耦合，Java的做法是将其变为私有变量，并提供get和set方法
- .Net则将该变量变为属性，在属性内部提供get和set方法。
- 这两种方法大同小异，使用get方法可以访问变量，而恰当地使用set方法能够保证“合法地”修改变量。
- 本例中，可以将unitPrice变量改为属性，并提供get和set方法。

## 示例2 以下程序属于哪种耦合？

例2 根据年龄判断是否大于18岁，然后根据是否满18岁判断是否到达法定饮酒年龄

```
#include <stdio.h>

static bool Signal;

void AdultOrNot(int age){
    if (age > 18)
        Signal = 1;

    else    // 出现控制耦合
        Signal = 0;
}
```

```
void WineOrNot(){
    if (Signal == 1)
        printf("%s\n", "您已到达法定饮酒年龄! ");
    else
        printf("%s\n", "您未到达法定饮酒年龄! ");
}

int main(){
    int Age = 0; printf("%s", "请输入您的年龄: ");
    scanf("%d", &Age);
    AdultOrNot(Age);
    WineOrNot();
}
```

这种形式耦合的主要问题在于B中的一个不相干变更，往往能够导致A所传递控制标记的意义必须发生变更。改善方法就是把B构件调用的函数直接写入A构件中，然后删除B构件。

### 示例3 标记耦合

例：在宠物商店的例子中，订单类（Order）有一个方法(getTotalMoney)是计算用户订单的总金额。该方法要根据用户的级别（如1-钻石级、2-白金级和3-黄金级等）和消费总积分而采用不同的折扣策略。

为了获得用户级别和消费总积分，在下面的程序中，User类的一个实例作为一个参数传入了Order类的calcTotalMoney方法中。这样，Order类和User类构成了标记耦合。

```
public class Order {  
    public float calcTotalMoney(User user) {  
        int userLevel = user.getLevel();  
        int userConsumeScore = user.getConsumeScore();  
        //计算订单总金额  
    }  
    ...  
}
```






上面主要问题在于：

1) calcTotalMoney方法只需要使用user的getLevel方法和getConsumeScore方法，然而user作为一个User类的实例传入calcTotalMoney中，它的所有公共方法和变量都暴露给了calcTotalMoney方法，显然授予了calcTotalMoney更大的、不必要的访问权限。这容易产生副作用。

2) 当开发人员修改User类时，必须得检查是否也要修改Order类，因为后者用到了前者的方法。

3) Order类可复用性较低，因为它依赖于User类。必须将二者“捆绑”在一起才能被复用到其它环境。





订单类（Order）的calcTotalMoney方法可以用两个简单变量类型的参数代替User类的实例，代码如下：

```
public class Order
{
    public float getTotalMoney(int userLevel, int
consumeScore)
    {
        //...
        return 0;
    }
}
```

显然，这种方法限制了Order类的访问权限，降低了Order类与User类的耦合。



## 5.2 设计原理

耦合和内聚的概念是Constantine, Yourdon, Myers和Stevens等人提出来的。上述7种内聚的优劣评分，将得到如下结果：



事实上，没有必要精确确定内聚的级别。重要的是设计时高内聚，并且能够**辨认出**低内聚的模块，有能力通过修改设计**提高模块的内聚**程度并且**降低模块间的耦合**程度，从而获得较高的模块独立性。

# 主要内容

5.1 设计过程

5.2 设计原理



5.3 启发规则

5.4 描绘软件结构的图形工具

5.5 面向数据流的设计方法

## 5.3 启发规则

### 1.改进软件结构提高模块独立性

设计出软件的初步结构以后，应该审查分析这个结构，通过模块分解或合并，力求降低耦合提高内聚。

### 2. 模块规模应该适中

一个模块的规模不应过大，最好能写在一页纸内(通常不超过60行语句)

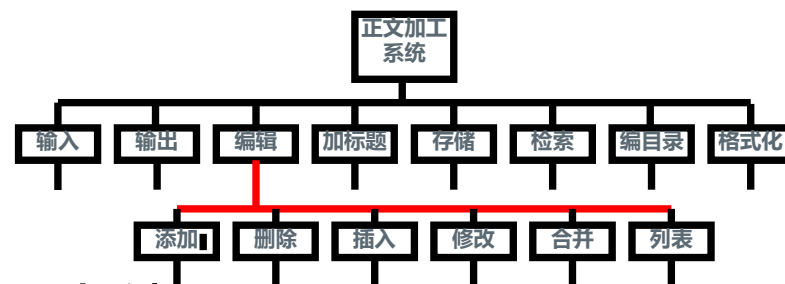
### 3.深度、宽度、扇出和扇入都应适当

深度：软件结构中控制的层数

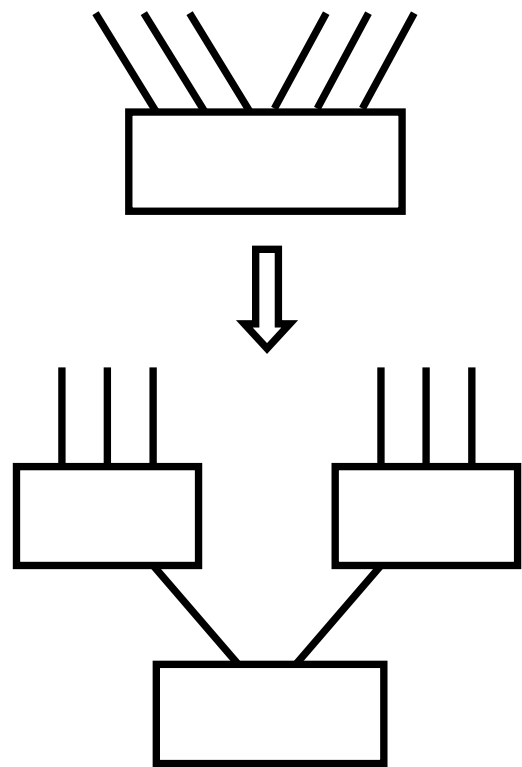
宽度：软件结构内同一个层次上的模块总数的最大值

扇出：一个模块直接控制(调用)的模块数目

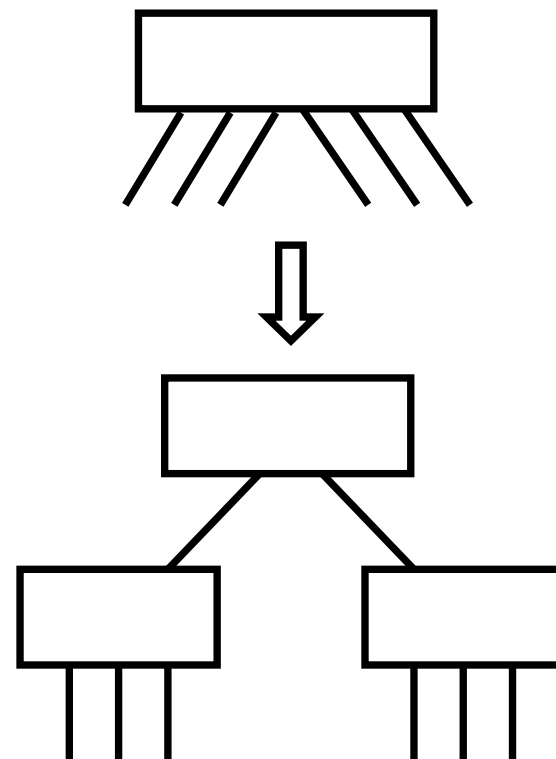
扇入：一个模块被多少个上级模块直接调用的数目



- 对扇出、扇入过大的改进：



**(a) 对扇入过大的改进**



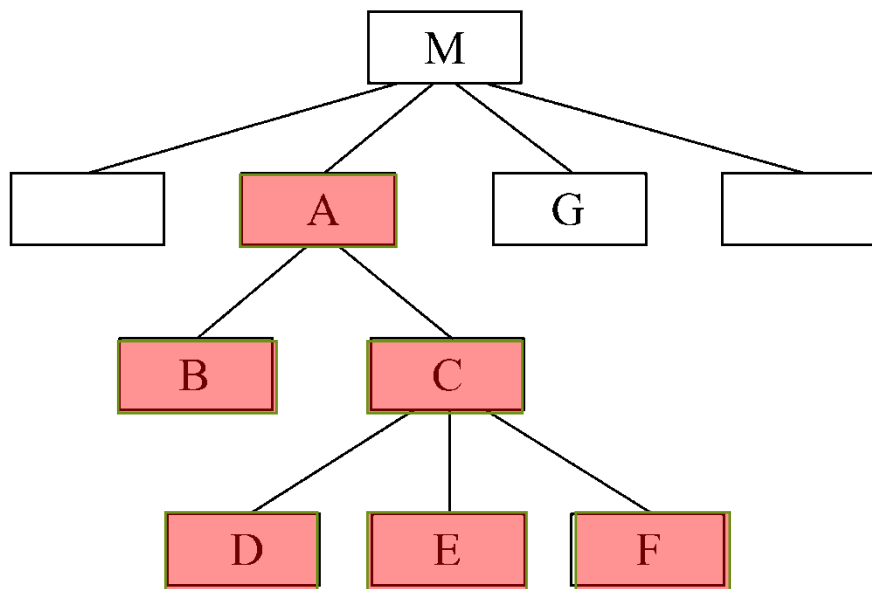
**(b) 对扇出过大的改进**

## 5.3 启发规则

### 4. 模块的作用域应该在控制域之内

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：模块本身以及所有直接或间接从属于它的模块的集合。



在图中模块A的控制域是A、B、C、D、E、F等模块的集合。



## 5.3 启发规则

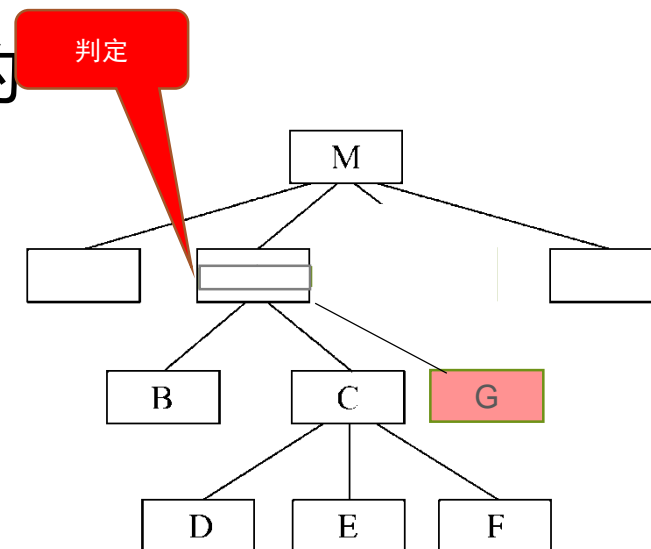
在一个设计得很好的系统中，所有受判定影响的模块应该都从属于做出判定的那个模块，最好局限于做出判定的那个模块及它的直属下级模块

怎样修改软件结构才能使作用域是控制域的子集呢？

(右图中，模块G受A的判定影响)

方法1是把做判定的点往上移（例如，把判定从模块A中移到模块M中）

方法2是把那些在作用域内但不在控制域内的模块移到控制域内（例如，把模块G移到模块A的下面，成为它的直属下级模块。）



## 5.3 启发规则

### 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。

如：QUAD\_ROOT(TBL,X)

求一元二次方程的根的模块，其中用数组TBL传送方程的系数，用数组X回送求得的根。

使用接口可能是比较简单，如：

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。

## 5.3 启发规则

### 6.设计单入口单出口的模块

这条启发式规则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，软件是比较容易理解的，因此也是比较容易维护的。

### 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。

# 主要内容

5.1 设计过程

5.2 设计原理

5.3 启发规则



5.4 描绘软件结构的图形工具

5.5 面向数据流的设计方法

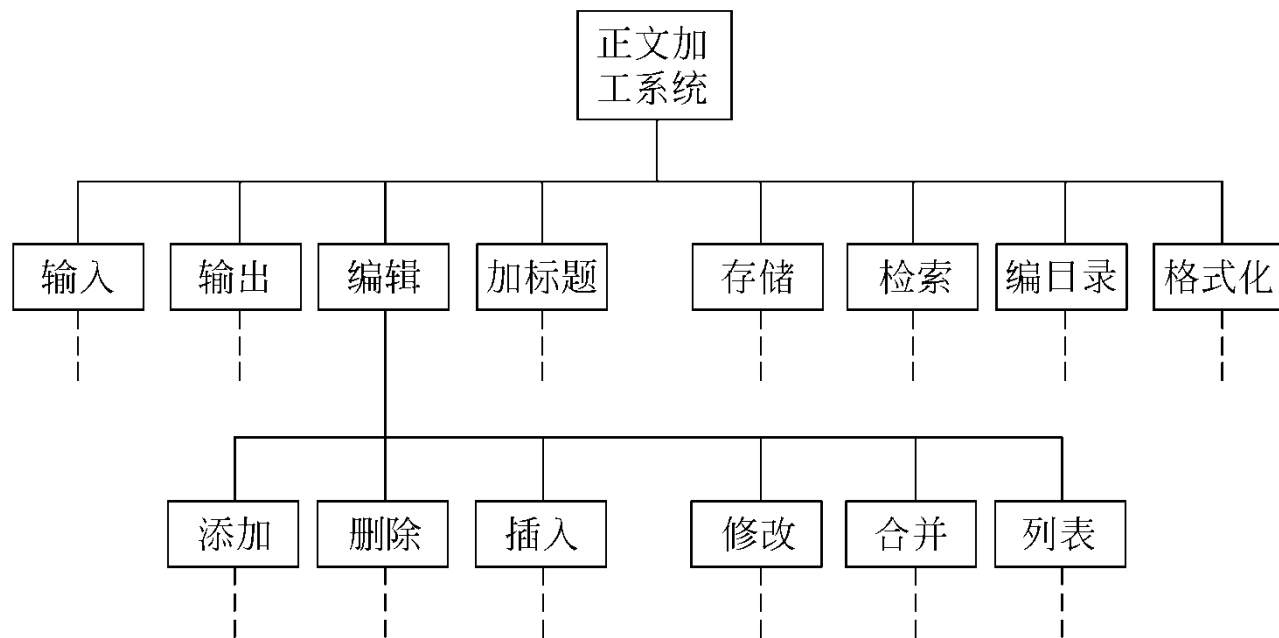


# 5.4 描绘软件结构的图形工具

## 5.4.1 层次图和HIPO图

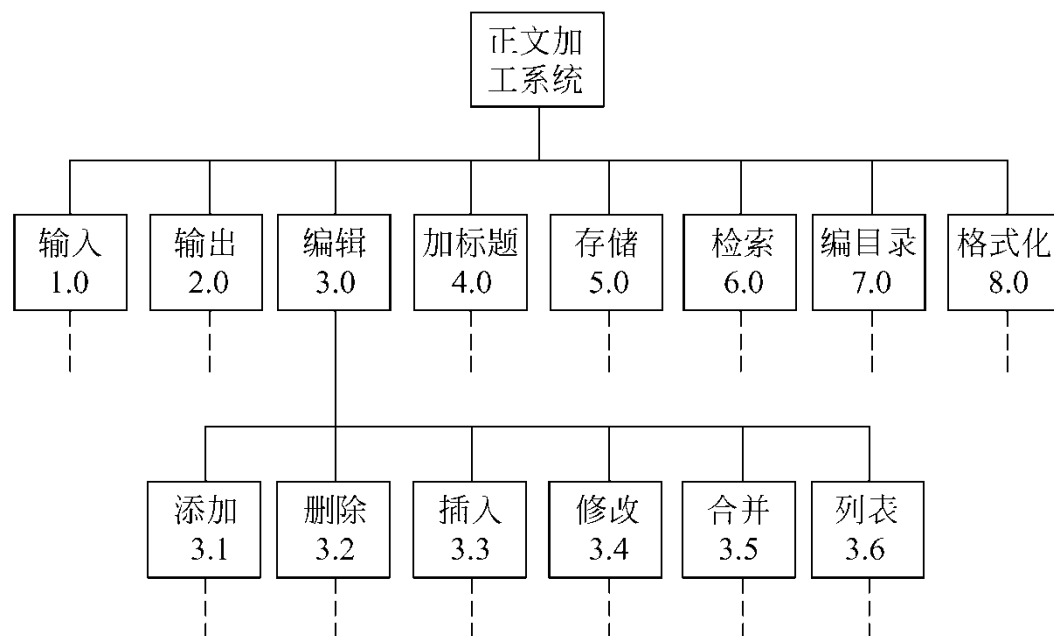
层次图用来描绘软件的层次结构。数据结构的层次方框图相同，但是表现的内容却完全不同。层次图很适于在自顶向下设计软件的过程中使用。

最顶层的方框代表正文加工系统的主控模块，它调用下层模块完成正文加工的全部功能；第二层的每个模块控制完成正文加工的一个主要功能，例如“编辑”模块通过调用它的下属模块可以完成6种编辑功能中的任何一种。



## 5.4 描绘软件结构的图形工具

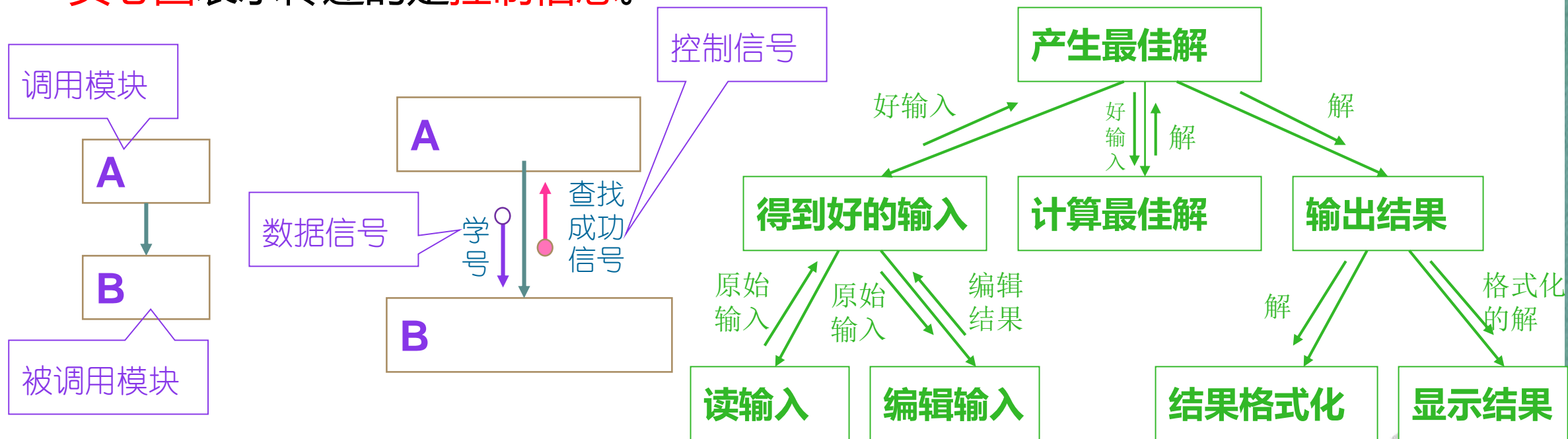
HIPO图是美国IBM公司发明的“层次图加输入/处理/输出图”的英文缩写。为了能使HIPO图具有可追踪性，在H图(层次图)里除了最顶层的方框之外，每个方框都加了**编号**。



# 5.4 描绘软件结构的图形工具

## 5.4.2 结构图

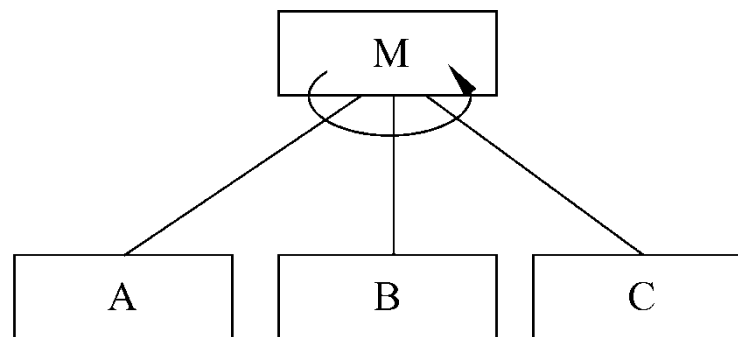
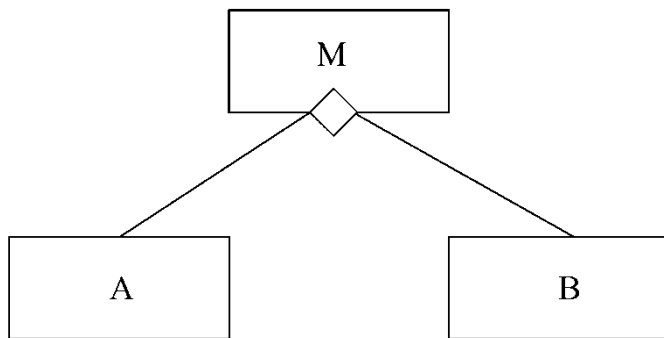
- Yourdon提出的结构图是进行软件结构设计的工具。图中一个方框代表一个模块，框内注明模块的名字或主要功能；
- 方框之间的箭头(或直线)表示模块的调用关系。尾部是**空心圆**表示传递的是**数据**，**实心圆**表示传递的是**控制信息**。





## 5.4 描绘软件结构的图形工具

- 一些附加的符号，可以表示模块的选择调用或循环调用；
- 左图表示当模块M中某个判定为真时调用模块A，为假时调用模块B；
- 右图表示模块M循环调用模块A、B和C。



# 主要内容

5.1 设计过程

5.2 设计原理

5.3 启发规则

5.4 描绘软件结构的图形工具



5.5 面向数据流的设计方法



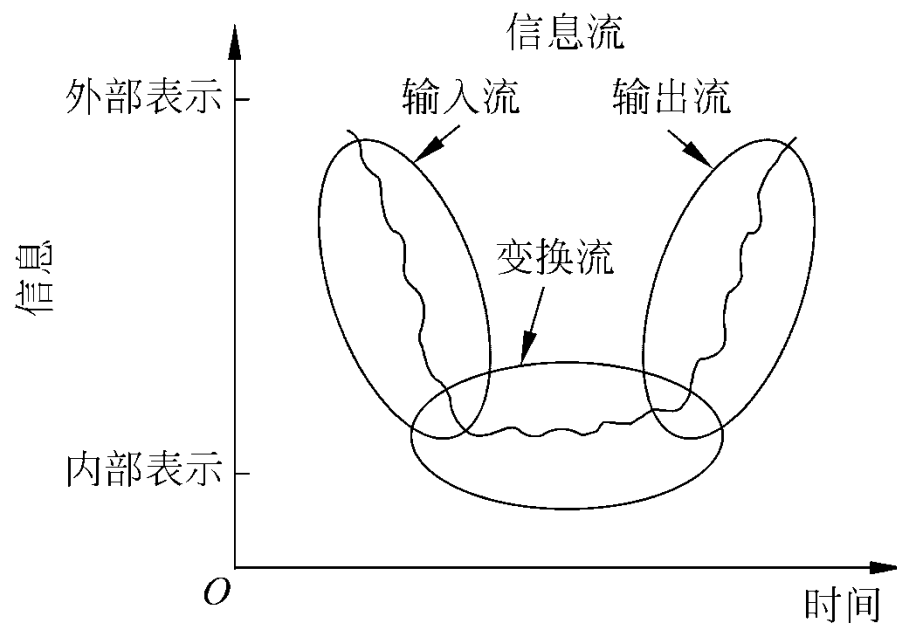
# 5.5 面向数据流的设计方法

## 5.5.1 概念

面向数据流的设计方法把信息流映射成**软件结构**（即：以**数据流图为基础的**软件模块结构图****），信息流的类型决定了映射的方法。**信息流**有下述两种类型（**变换流，事务流**）。

### 1) 变换流

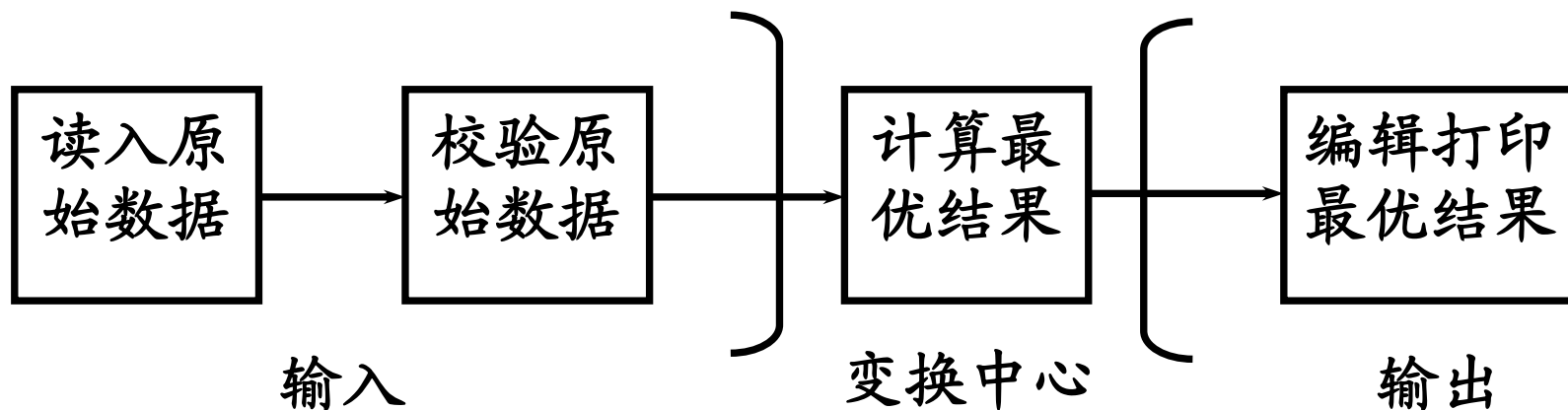
信息沿输入通路进入系统，由外部形式**变换**成内部形式，进入系统的信息通过**变换中心**，经加工处理以后再沿输出通路**变换**成外部形式离开软件系统。当数据流图具有这些特征时，这种信息流就叫作变换流。



## 变换型数据流图

具有较明确的输入、变换（或称主加工）和输出界面的数据流图称为变换型数据流图。

如图所示，该变换中心可以理解为数据的加工和处理程序。



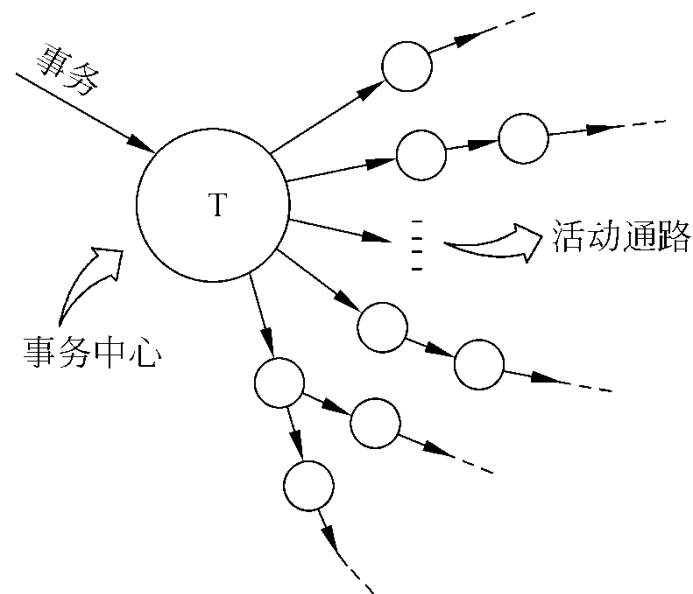
## 5.5 面向数据流的设计方法

### 2) 事务流

数据沿输入通路到达一个处理T，这个处理根据输入数据的类型在若干个动作序列中**选出**一个来执行。这类数据流应该划为一类特殊的数据流，称为事务流。

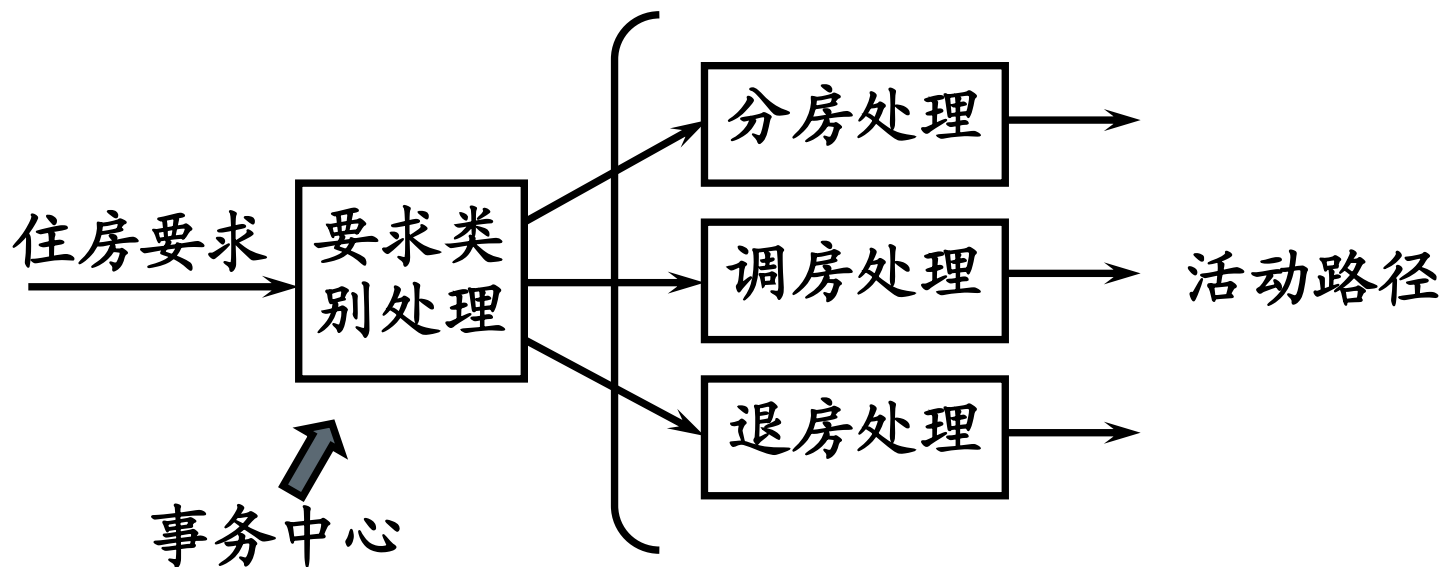
图中的处理T称为事务中心，它完成下述任务。

- (1) 接收输入数据(输入数据又称为事务)。
- (2) 分析每个事务以确定它的类型。
- (3) 根据事务类型选取一条活动通路。

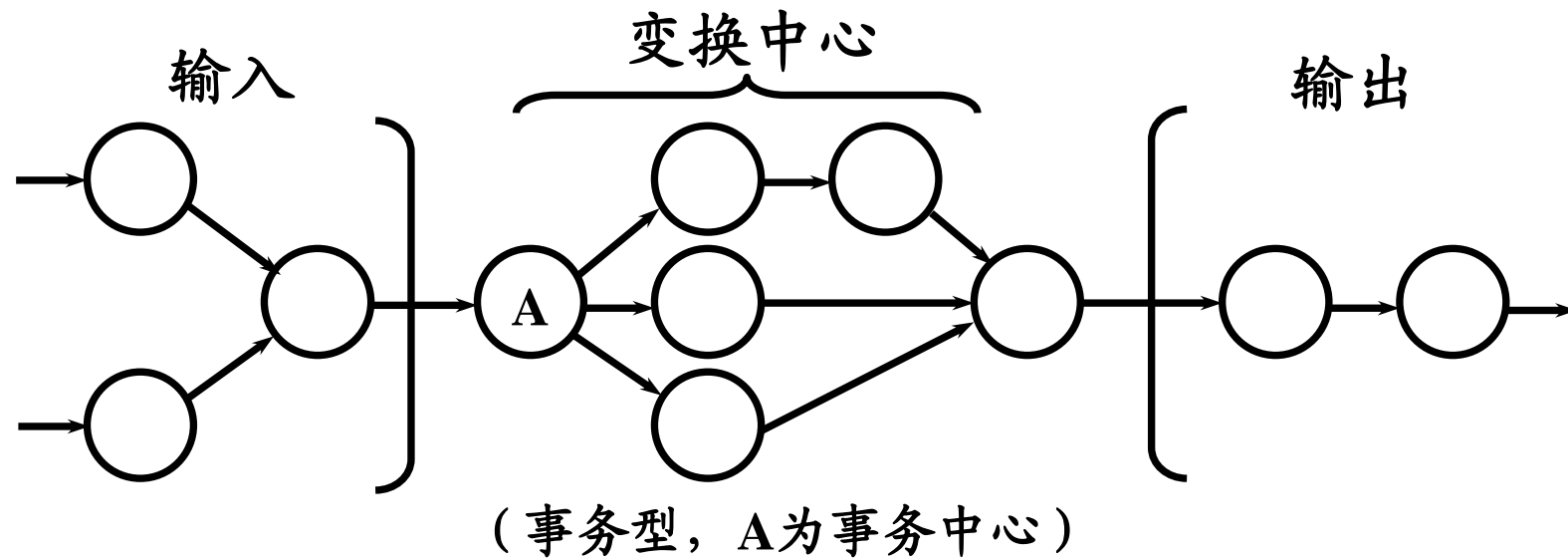


## 事务型数据流图

- 事务型数据流图中存在一个事务中心（也就是数据处理、加工中心），它将输入分离成若干个发散的数据流，形成许多活动路径，并根据输入值选择其中一条路径。



- 通常，一个实际系统的数据流图是变换型和事务型两种类型的混合体。
- 如图所示，中间子块属事务型数据流，如果把中间子块视为一个处理整体的话，整个程序属变换型程序。

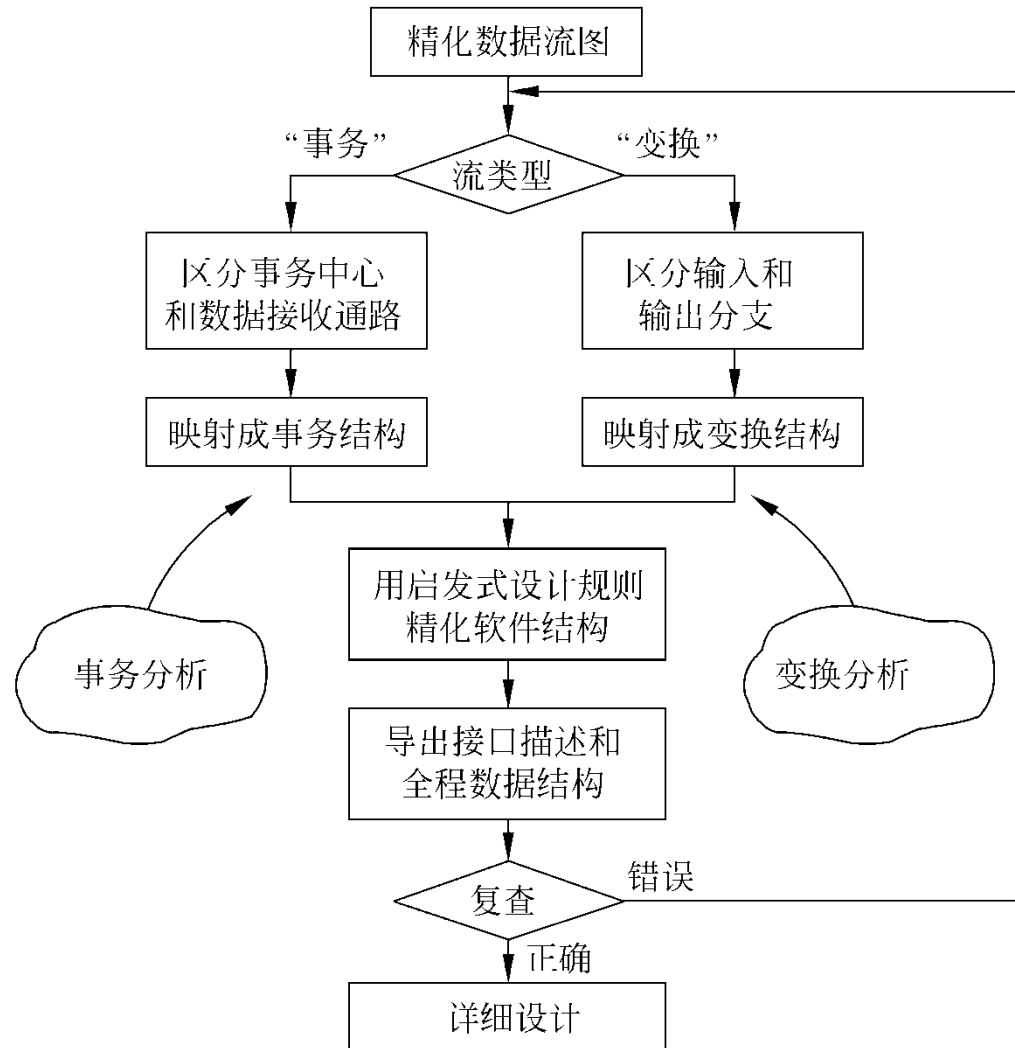


混合型数据流图



## 5.5 面向数据流的设计方法

### 3) 设计过程



在讨论如何转换之前，我们首先讨论一下典型的系统结构类型。

- 在系统结构图中把不能再分割的底层模块称之为-----**原子模块**。

### 完全因子分解系统

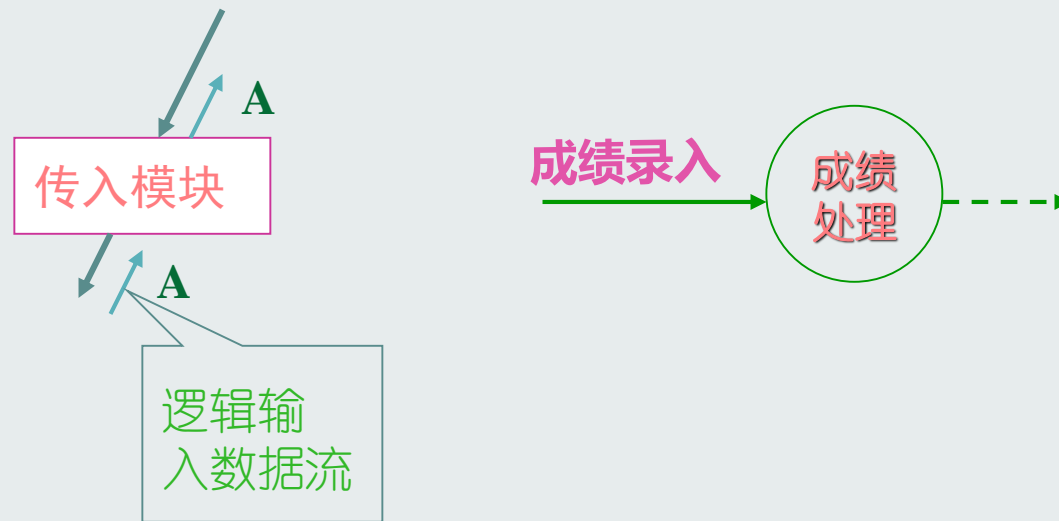
如果一个软件系统，它的全部实际加工（即数据计算或处理）都是由**底层的原子模块**来完成，而其它所有非原子模块仅仅执行**控制或协调**功能。

- 实际上，这只是力图达到的目标，大多数系统是做不到完全因子分解。

# 一般在系统结构图中有四种类型的模块：

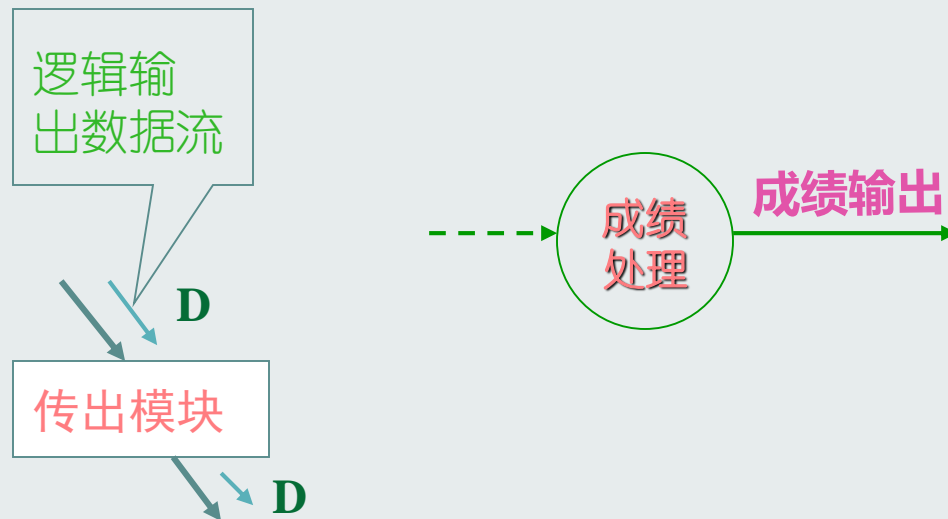
## 1. 传入模块

从下属模块取得数据，进行某些处理，再将其结果传给上级模块。在此，将它传送的数据流称为**逻辑输入数据流**。



## 2. 传出模块

从上级模块获得数据，进行某些处理，再将其结果传给下属模块。在此，将它传送的数据流称为**逻辑输出数据流**。



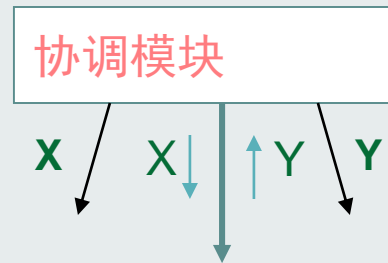
### 3. 变换模块

也叫加工模块。它是从上级模块获得数据，进行特定的处理，将其转换为其他形式，再传回上级模块。它所加工的数据流叫做**变换**数据流。



## 4. 协调模块

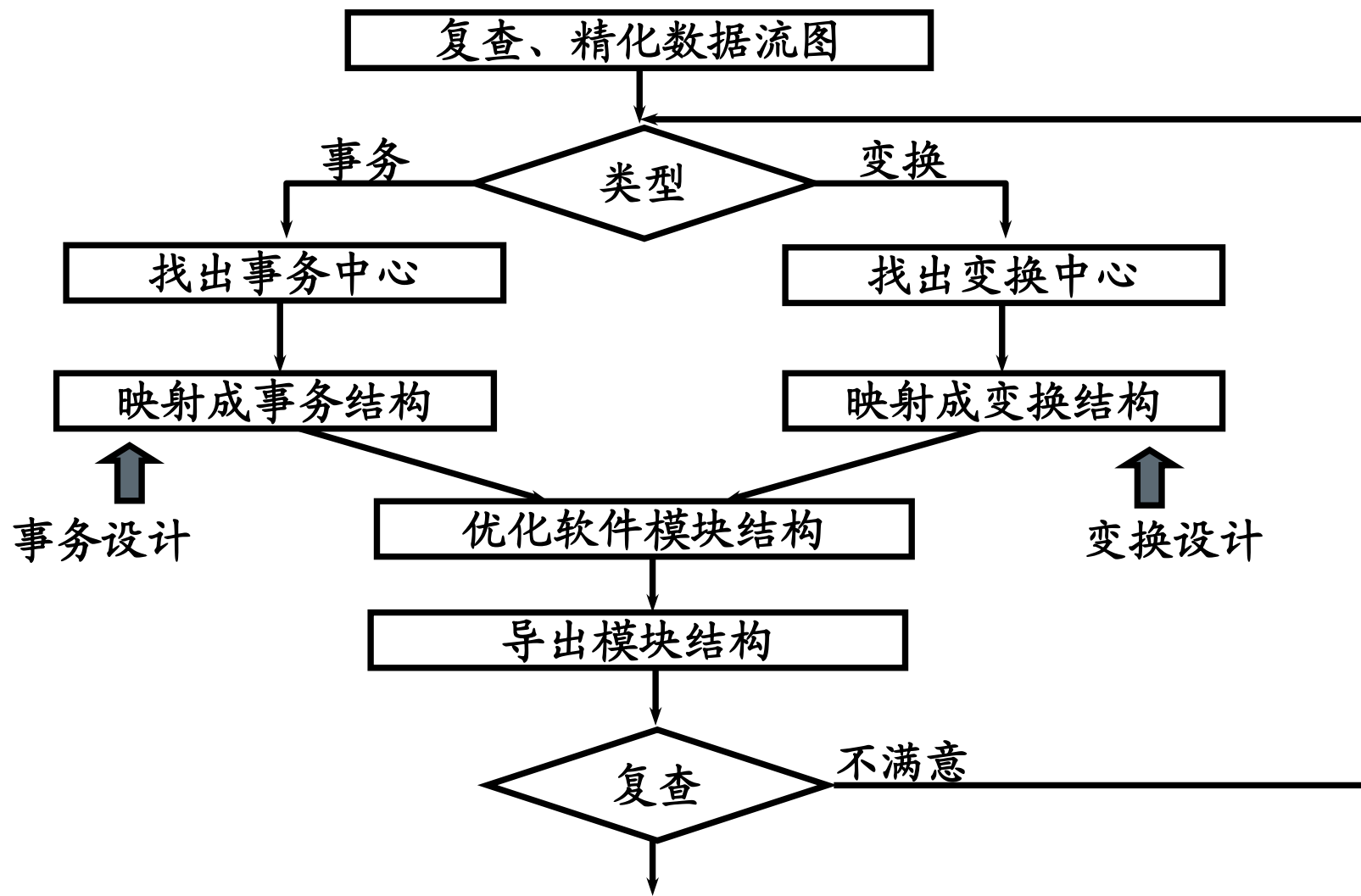
对所有下属模块进行**协调和管理**的模块。在一个好的系统结构图中，协调模块应在较高层出现。（如在系统的输入/输出部分或数据加工部分可以找到这样的模块。）



问题：如何从数据流图中获得拥有以上四种模块的软件结构？

- 面向数据流设计软件结构的基本步骤有七步：
  - 1) 复审并精化数据流图；
  - 2) 确定数据处理流图的类型；
  - 3) 确定变换中心或事务中心；
  - 4) 将数据流图映射成软件模块结构图，设计出该数据流图对应的第一层模块结构；
  - 5) 基于数据流图逐步分解，设计下层模块；
  - 6) 运用模块设计和优化准则优化软件结构；
  - 7) 描述模块的接口。





## 面向数据流的设计步骤

## 5.5.2 变换设计

- **变换设计**就是从变换型数据流图映射出软件模块结构的过程，也称以变换为中心的设计。



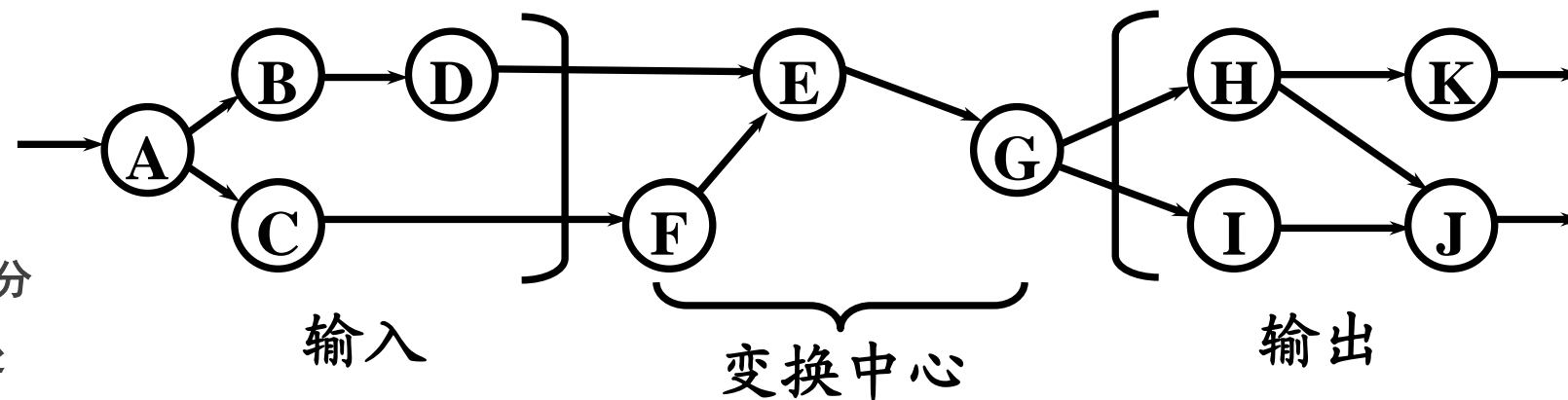
变换设计的基本方法有两步：

1) 分解第一层模块结构

就是把整个变换分解成输入控制模块 $C_i$ 、输出控制模块 $C_o$ 和变换中心控制模块 $C_t$ ，由主控模块控制。

• 2) 分别设计输入、输出和处理的下层模块结构

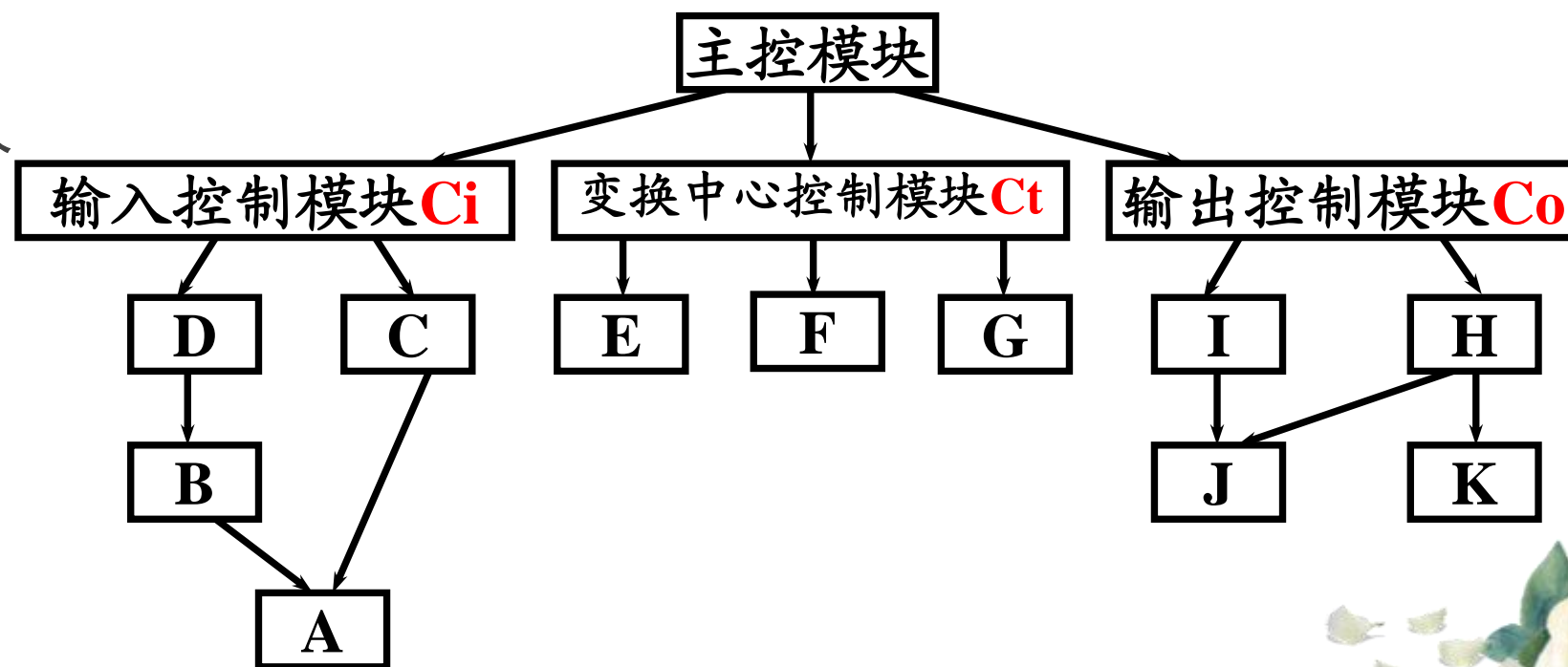
## 简图:



- 方法是:

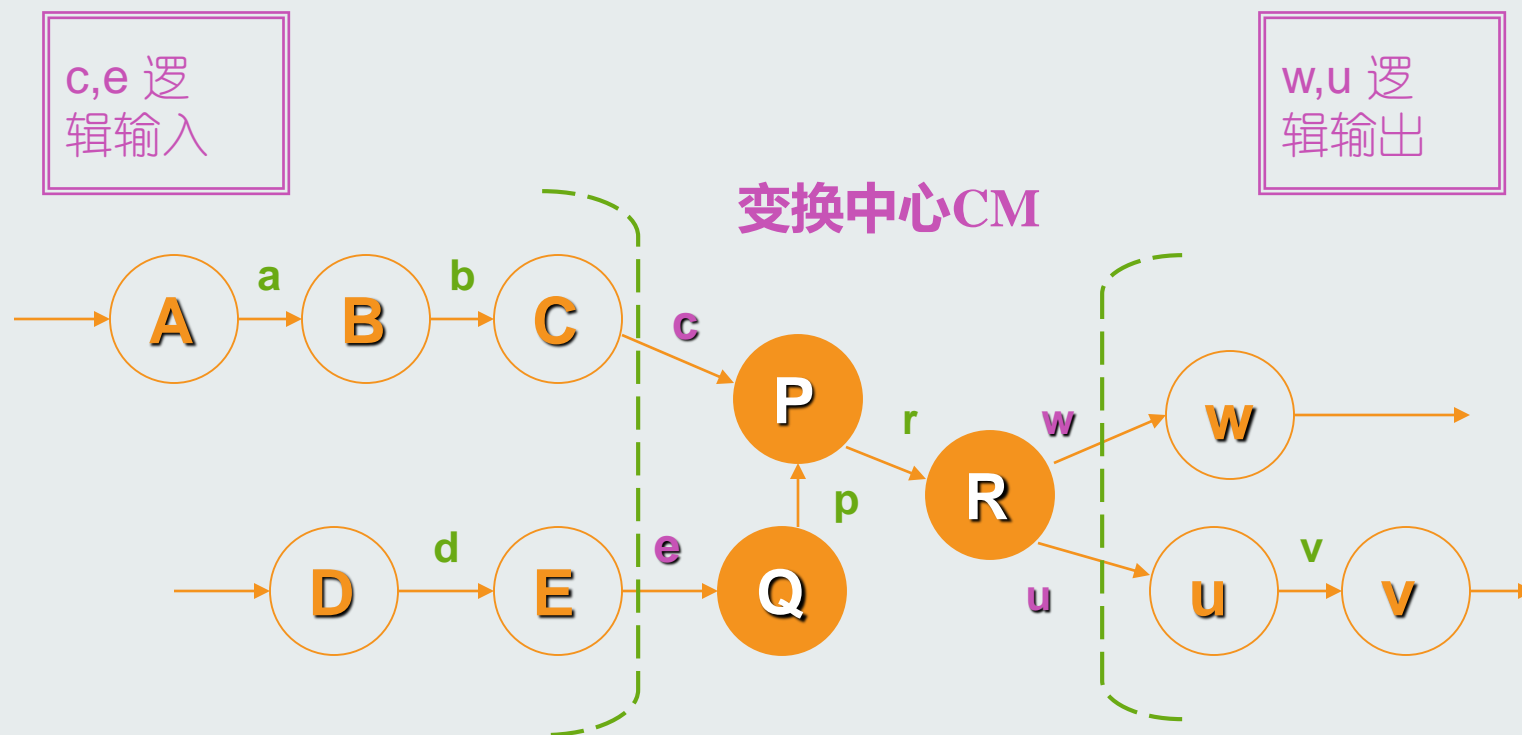
- 从变换中心边界向两侧移动, 分别把输入通路和输出通路的每个处理映射成输入控制模块Ci和输出控制模块Co的下属模块。

- 变换中心的下层模块, 是把每个处理映射成变换中心控制模块Ct的一个直接下属模块。



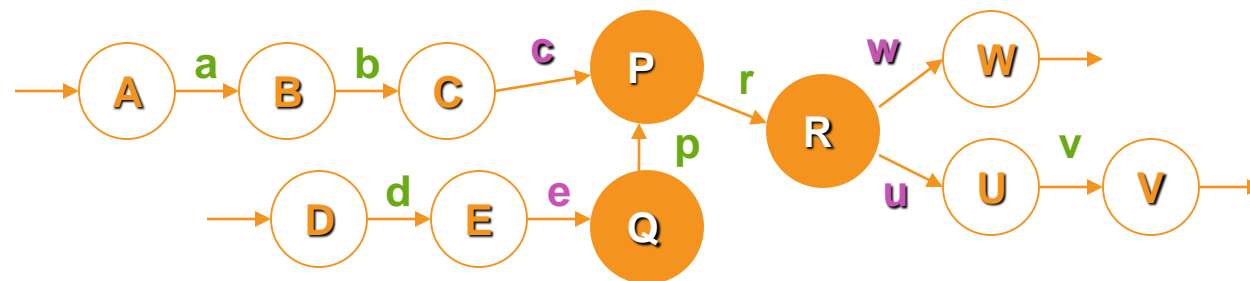
在 DFD 图上标出逻辑输入、逻辑输出和变换中心的分界

- 所谓**逻辑输入**，是指离物理输入端最远，但仍可以被看作系统的输入的那些数据流。
- 而**逻辑输出**则是离系统的物理输出端最远，但仍可视为系统的输出的数据流。



## (2) 完成第第一级分解

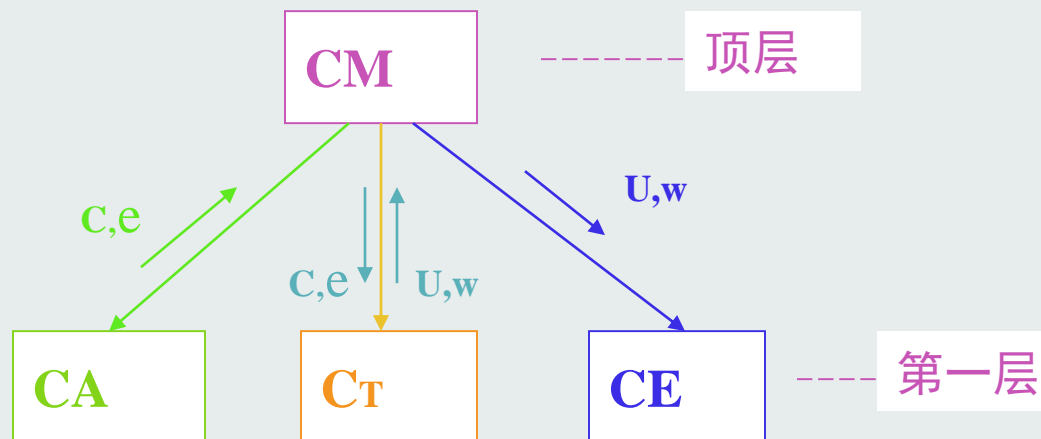
## 变换中心CM



CA

C<sub>T</sub>

CE



C<sub>T</sub>

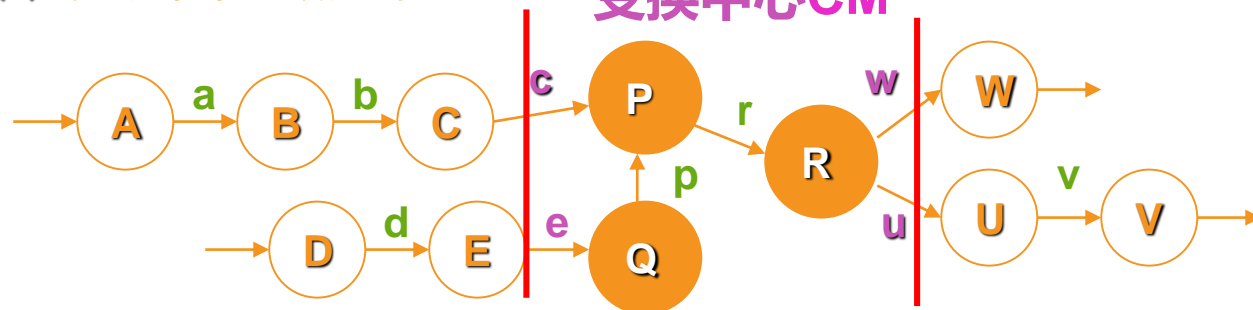
协调模块



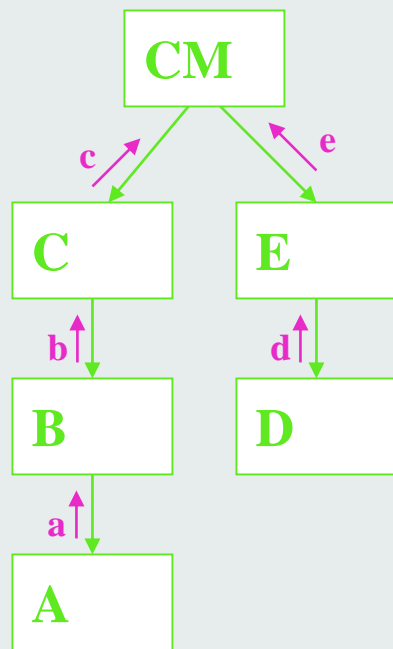
第一级分解后的 SC 图

### (3) 完成第第二级分解

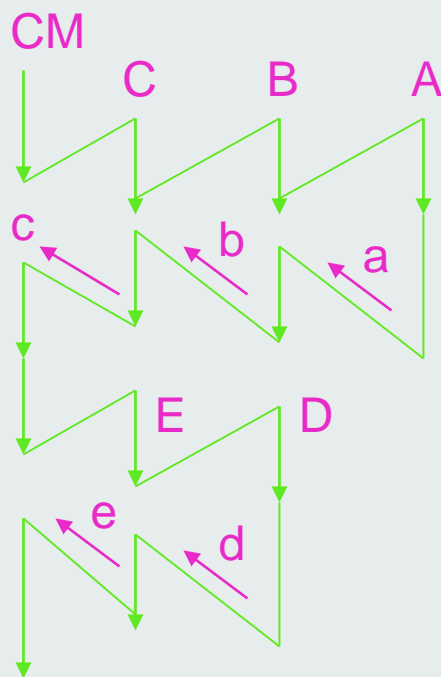
### 变换中心CM



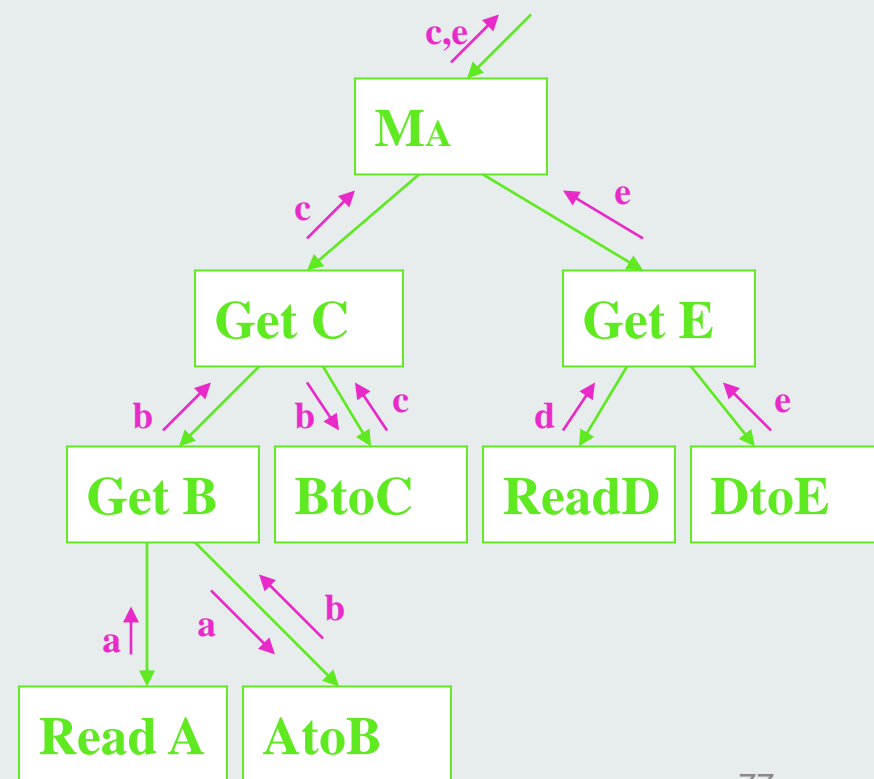
从图中可以看出，模块的调用顺序正好与加工的顺序相反



对逻辑输入的分解

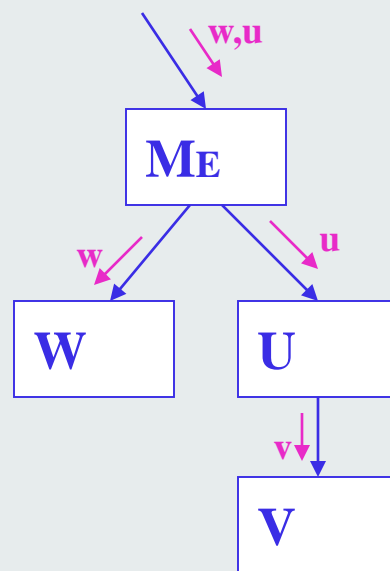
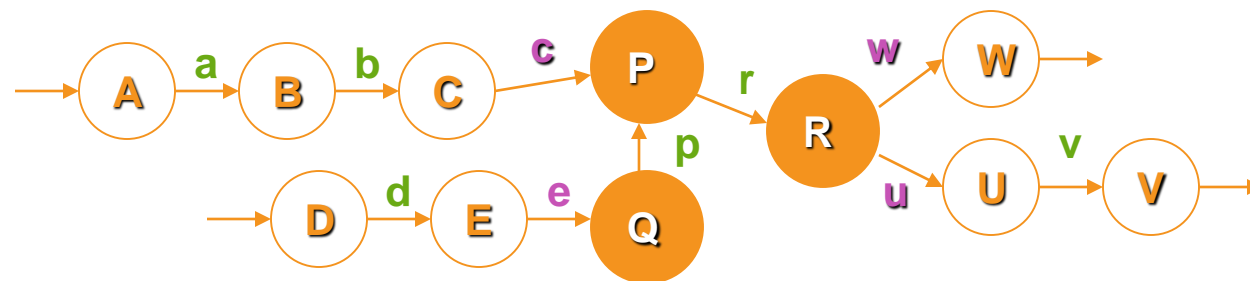


逻辑输入模块的调用与执行过程

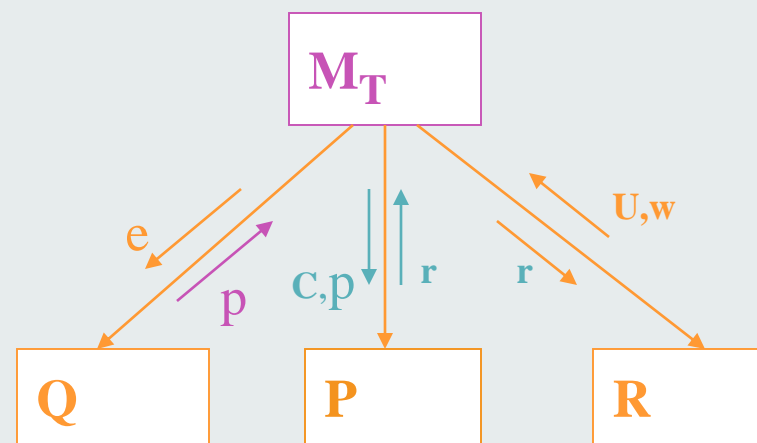




## 变换中心CM

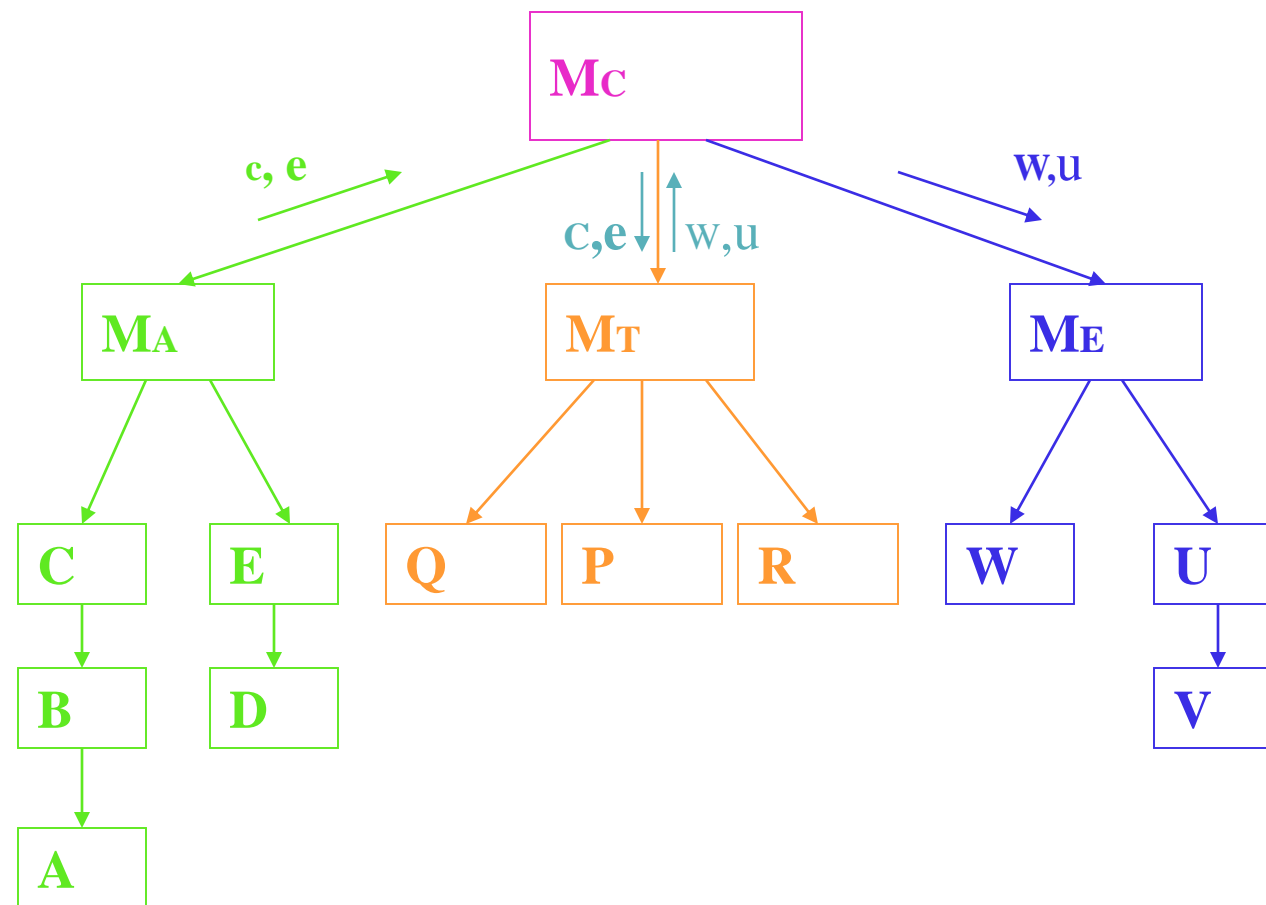


对输出的分解



对变换中心加工的分解

(4) 获得完整的 SC 图



从变换分析导出的初始 SC 图

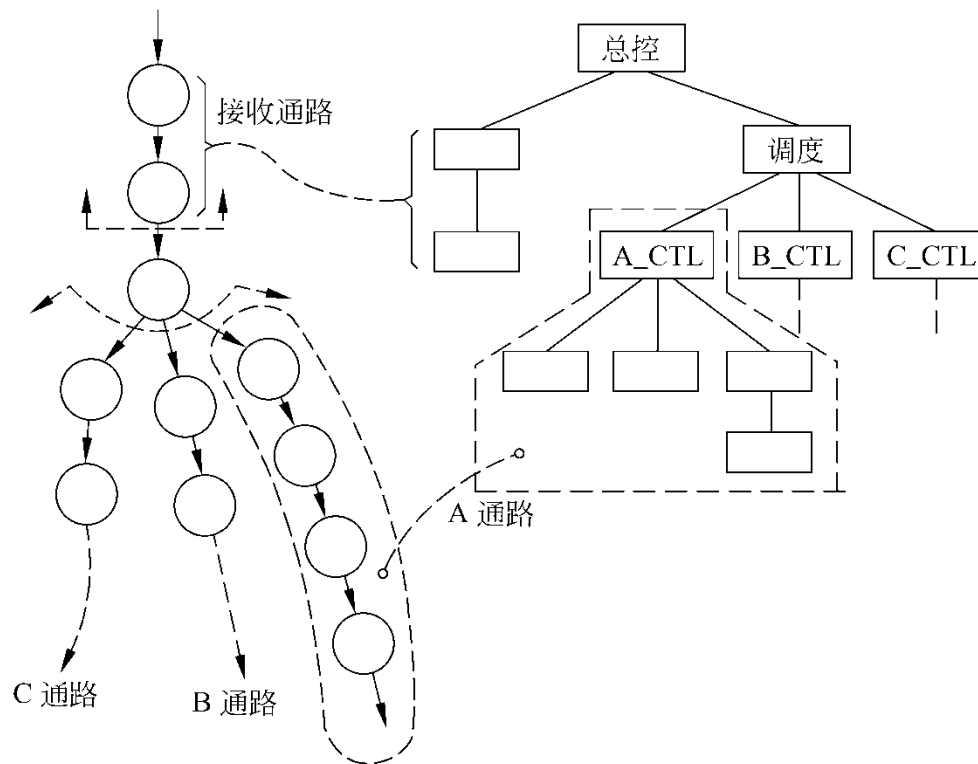
# 5.5 面向数据流的设计方法

## 5.5.3 事务分析

**事务设计**就是从事务型数据流图映射出软件模块结构的过程，也称为以事务为中心的设计。

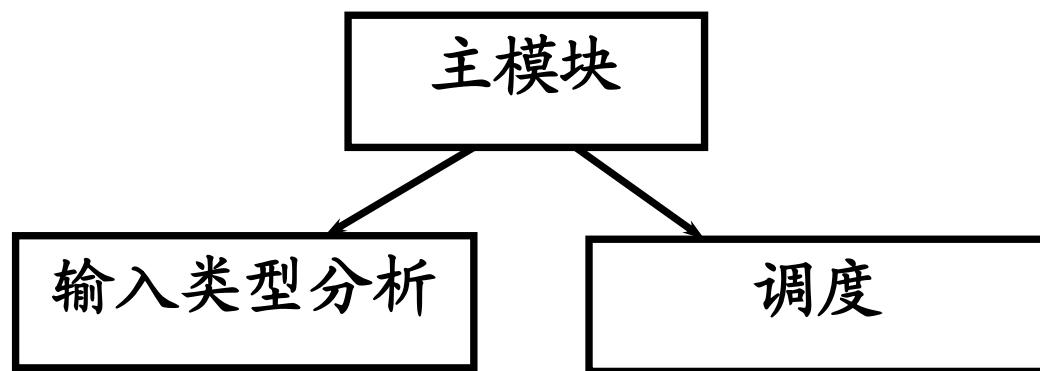
数据流具有明显的**事务特点**时采用事务分析方法。事务分析的设计步骤和变换分析的设计步骤大部分相同或类似，主要差别仅在于由数据流图到软件结构的**映射方法不同**。

由事务流映射成的软件结构包括一个接收分支和一个发送分支。



•事务设计的基本方法有两步：

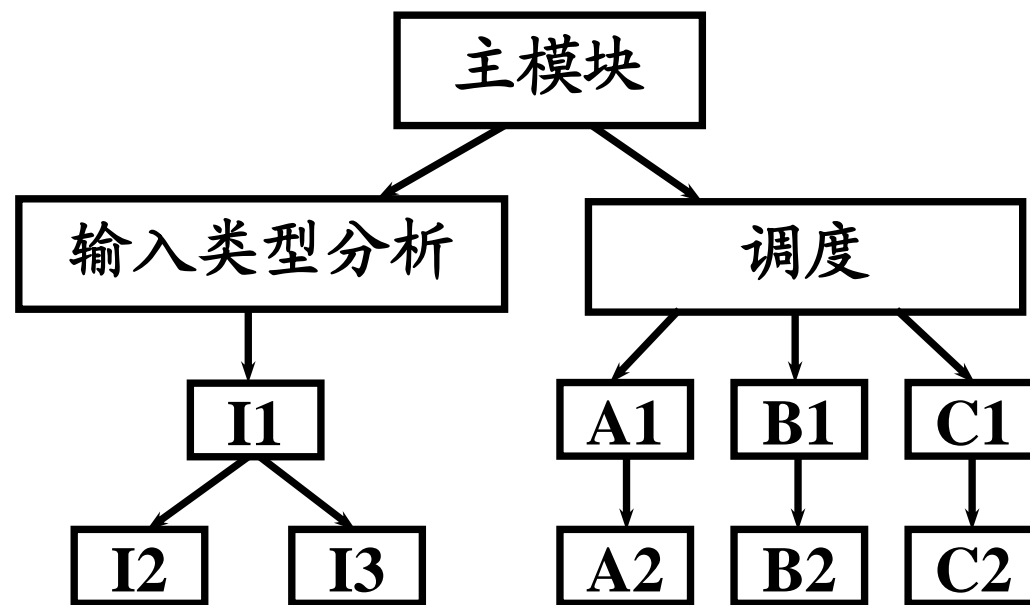
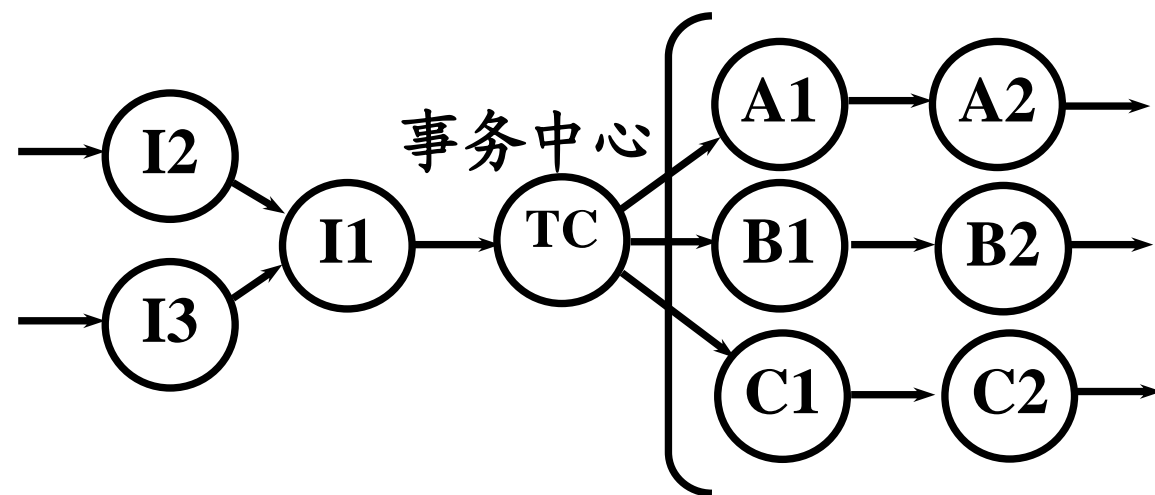
- 1) 建立主控模块、接收输入类型分析模块和事务调度模块；



•2) 分别设计输入类型分析模块和调度模块的下层模块结构。

- 方法是：将输出的每条通路作为调度模块的一个判断分支，而输入类型分析模块的下层模块与变换设计类似。

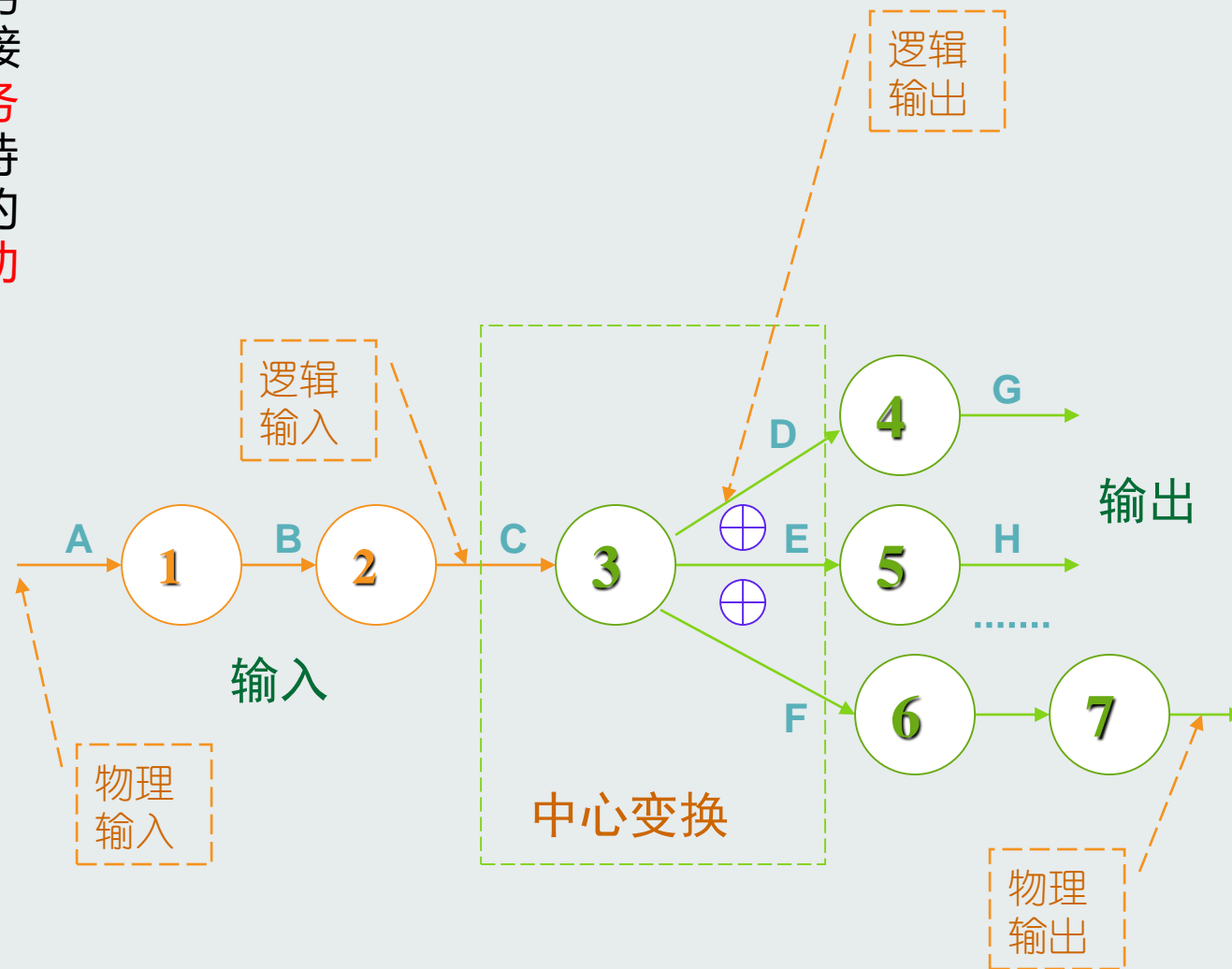
简图：



如图所示的是最典型的事务型的DFD图，当外部信息沿接受路径进入系统后，经过**事务中心的识别和分析**获得某一特定值，就可以根据这个特定的值来启动与该特定值相应的**动作路径**。

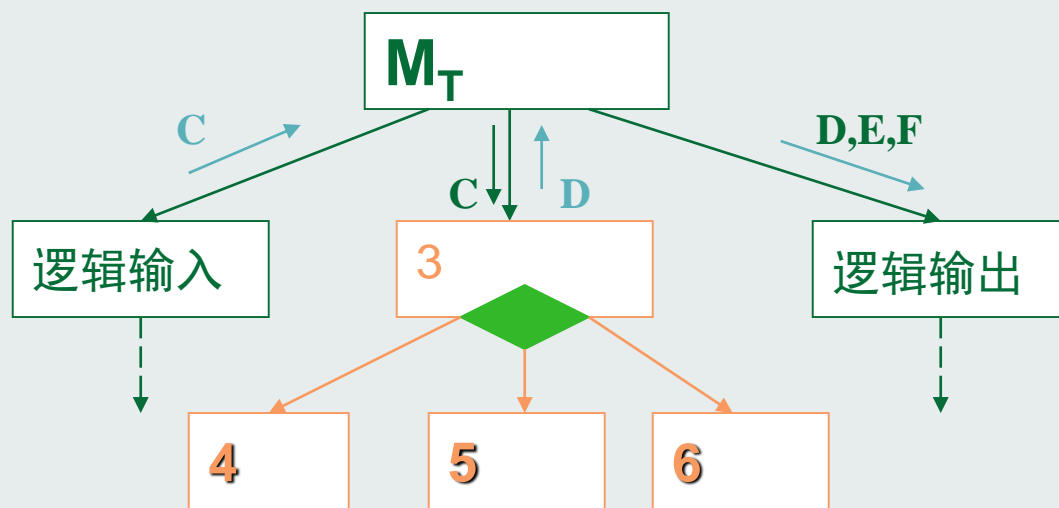
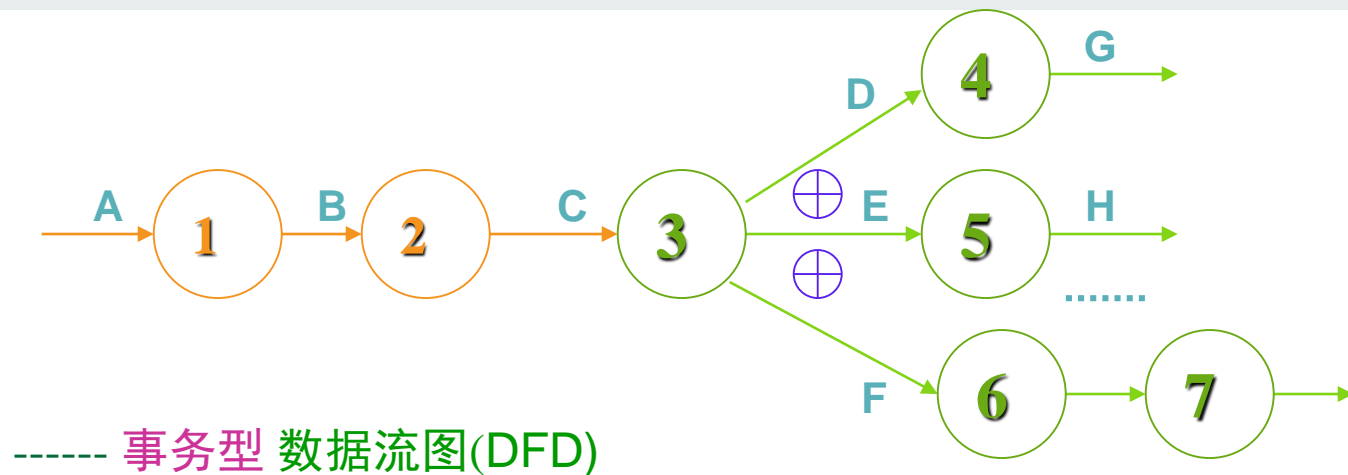
这类结构的特征，是具有能在**多种事务中选择执行某一事务的能力**。

如：在现代软件中常见的**菜单选择**，就是事务型结构的一个典型的实例。

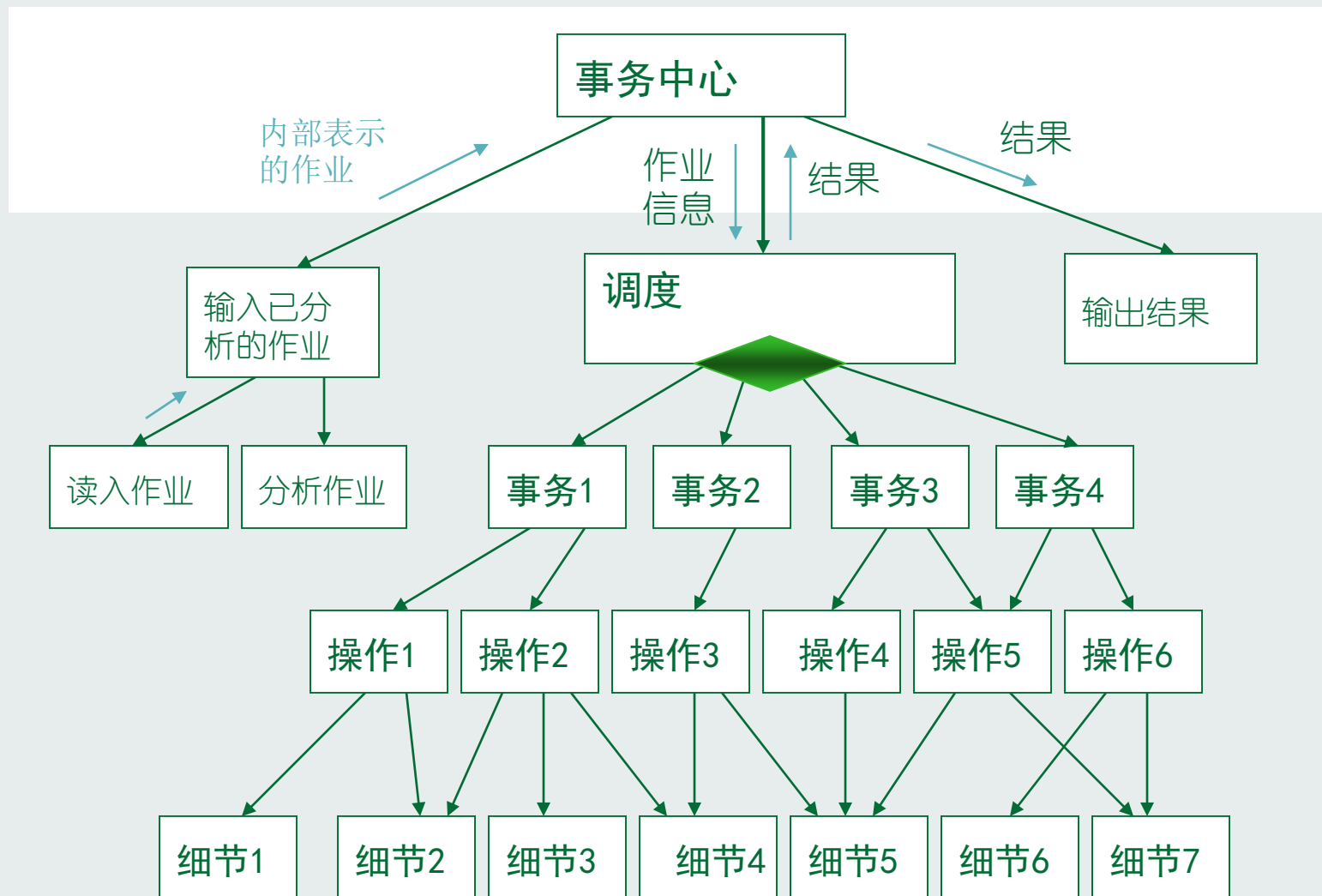


----- 事务型 数据流图

- 在事务型系统结构图中，事务中心模块按所接受的事务的类型，**选择**某一个事务处理模块执行。
- 各个事务处理模块是**并列**的，依赖于一定的选择条件，分别完成不同的事务处理工作。
- 每个事务处理模块可能要调用若干个操作模块，而操作模块又可能调用若干个细节模块。
- 由于不同的事务处理模块可能有共同的操作，所以某些事务处理模块可能共享一些操作模块。
- 同样不同的操作模块可以有相同的细节，所以某些操作模块又可以共享一些细节模块。







----- 事务型系统结构(层次)图

# 5.5 面向数据流的设计方法

## 5.5.4 变换分析

例子：

### ① 汽车数字仪表板的设计

假设的仪表板将完成下述功能。

- (1) 通过模数转换实现传感器和微处理机接口。
- (2) 在发光二极管面板上显示数据。
- (3) 指示每小时英里数(mph)，行驶的里程每加仑
- (4) 指示加速或减速。
- (5) 超速警告：如果车速超过55英里/小时，则发出超速警告铃声。



在软件需求分析阶段应该对上述每条要求以及系统的其他特点进行全面的分析评价，建立必要的文档资料，特别是数据流图。

# 5.5 面向数据流的设计方法

## ② 设计步骤

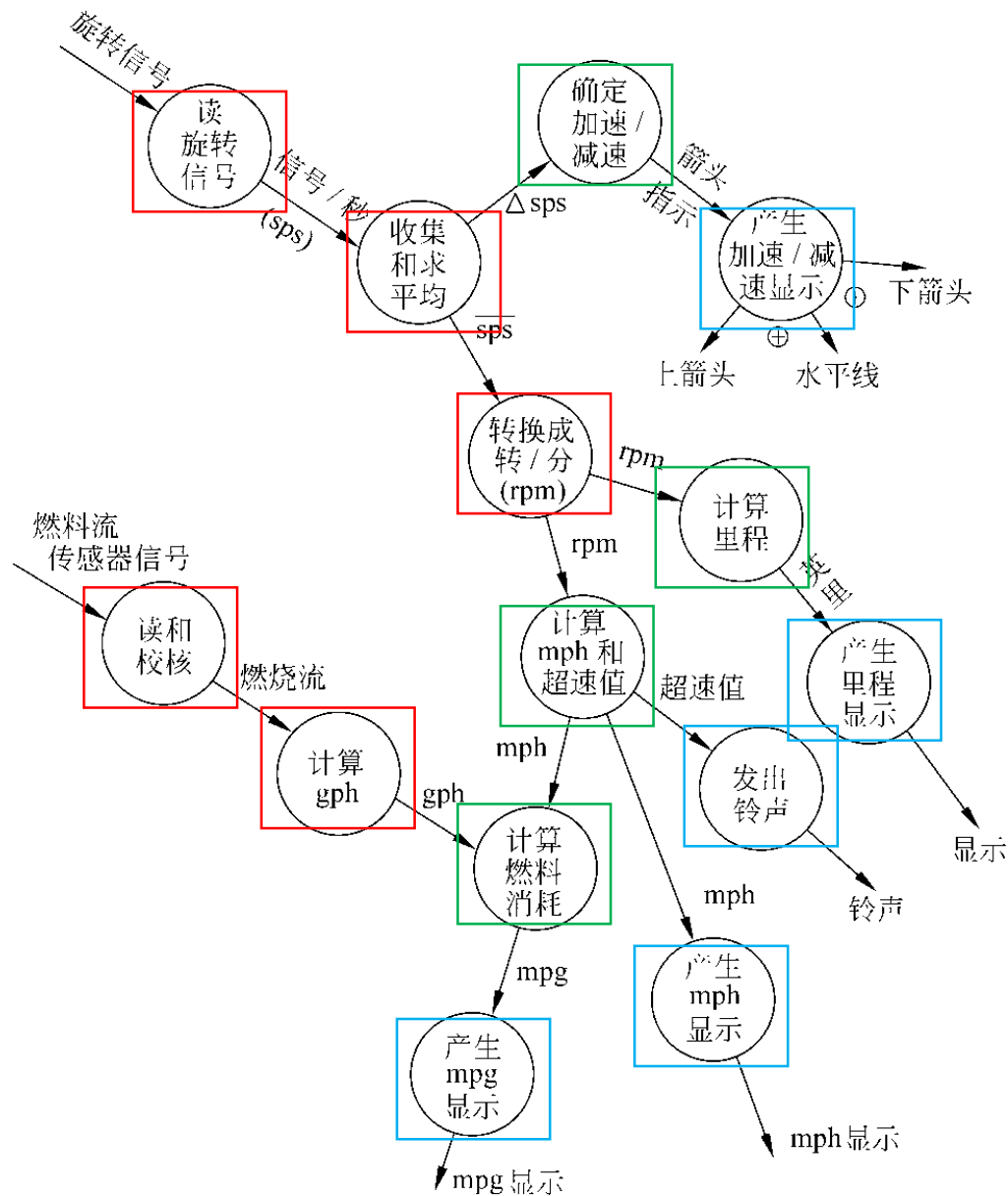
第1步复查基本系统模型。

第2步复查并精化数据流图。

假设在需求分析阶段产生的数字仪表盘系统的**数据流图**如图所示。

第3步确定数据流图具有**变换特性**还是**事务特性**。

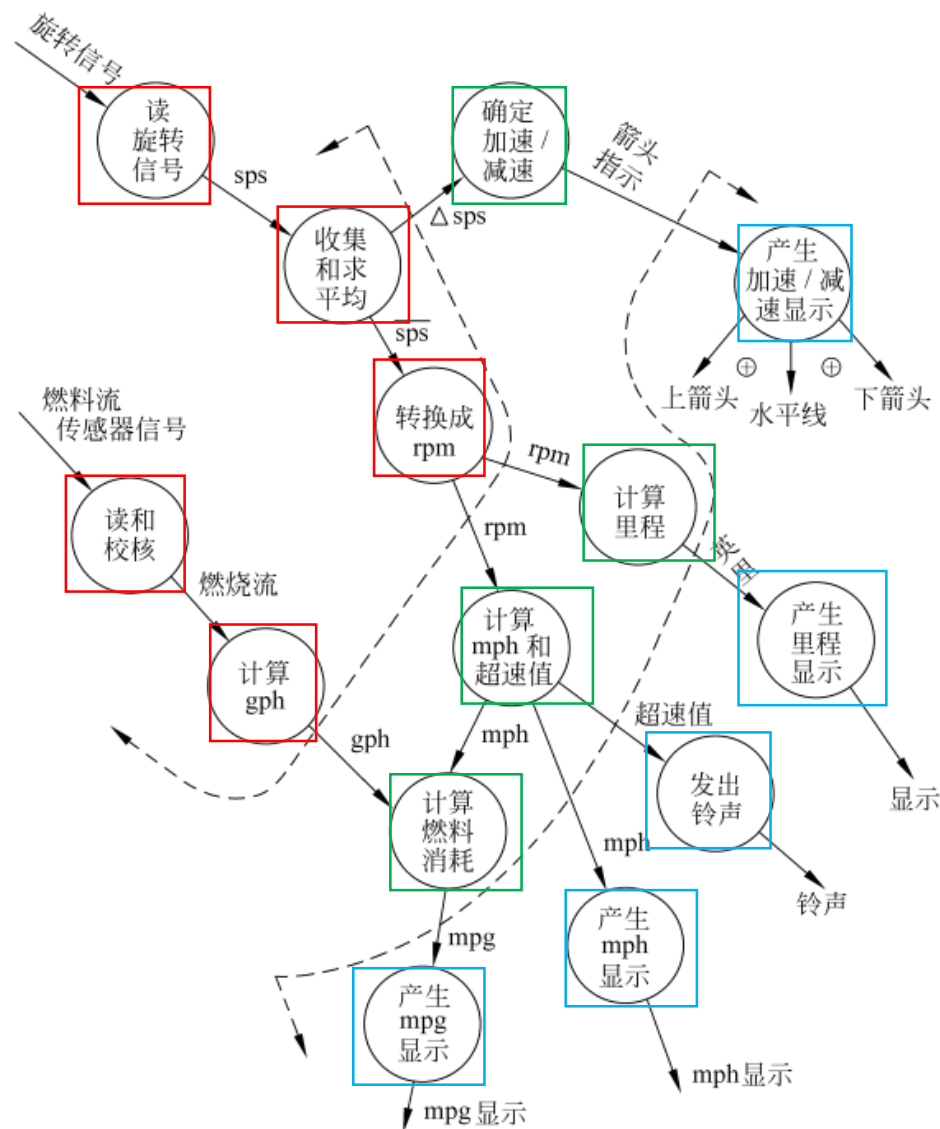
第4步确定输入流和输出流的边界，从而孤立出**变换中心**。



## 5.5 面向数据流的设计方法

**第3步**确定数据流图具有变换特性还是事务特性。

**第4步**确定输入流和输出流的边界，从而孤立出变换中心。



## 5.5 面向数据流的设计方法

### 第5步完成“第一级分解”

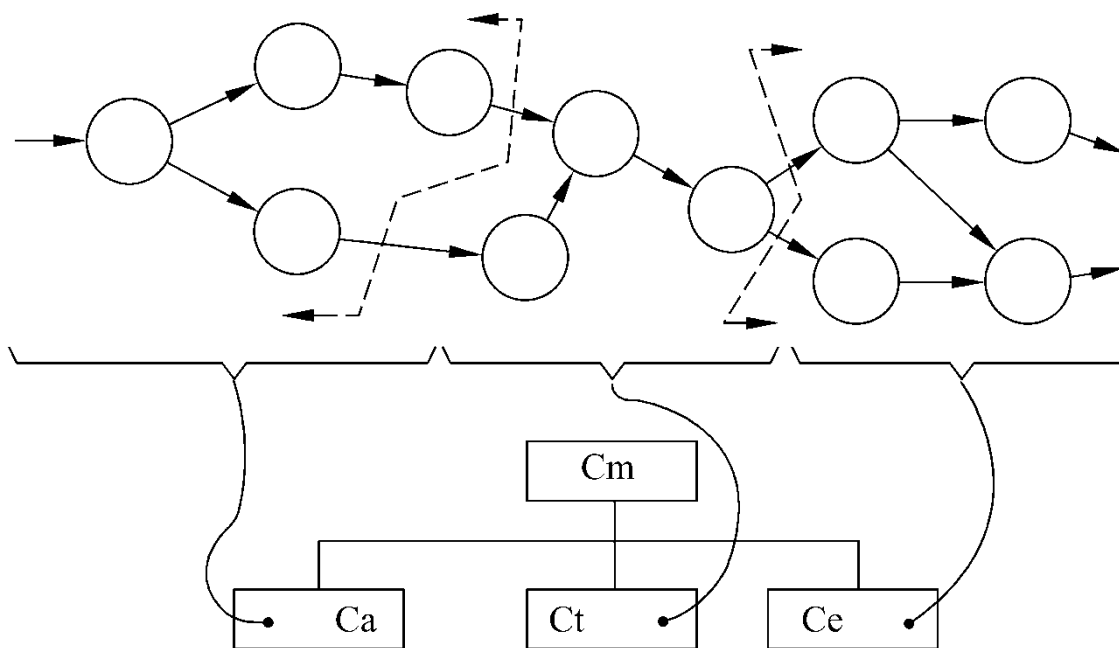
软件结构代表对控制的自顶向下的分配，所谓分解就是分配控制的过程。对于变换流的情况，数据流图被映射成一个特殊的软件结构，这个结构控制输入、变换和输出等信息处理过程。

位于软件结构**最顶层**的控制模块**Cm**协调下述从属的控制功能。

- **输入**信息处理控制模块**Ca**,协调对所有输入数据的接收。
- **变换中心**控制模块**Ct**,管理对内部形式的数据的所有操作。
- **输出**信息处理控制模块**Ce**，协调输出信息的产生过程。

## 5.5 面向数据流的设计方法

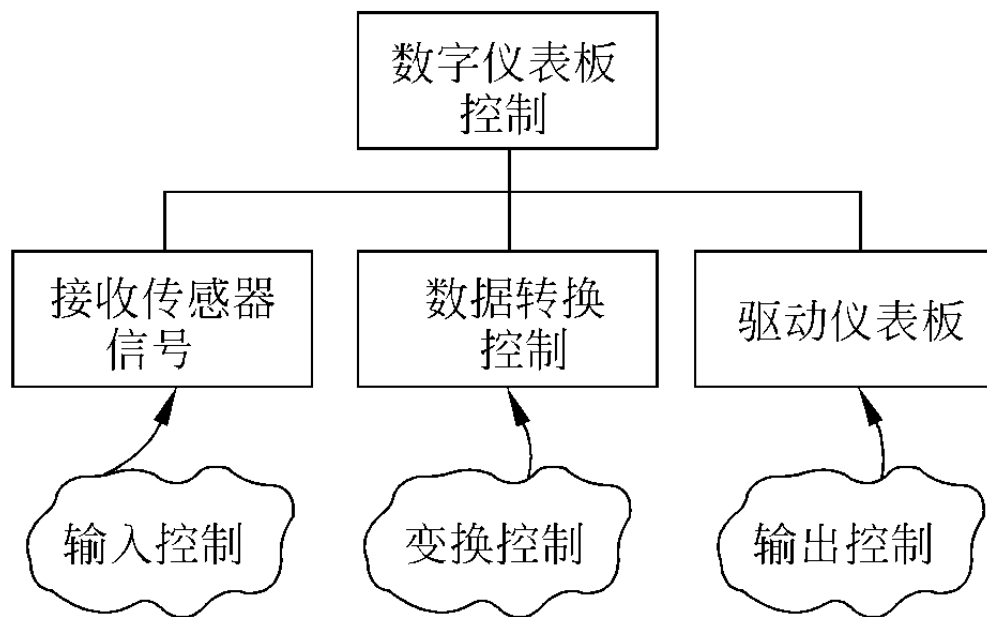
图5.13说明了第一级分解的方法。





## 5.5 面向数据流的设计方法

对于数字仪表板的例子，第一级分解得出的结构如图所示。每个控制模块的名字表明了为它所控制的那些模块的功能。





## 5.5 面向数据流的设计方法

### ② 设计步骤

#### 第6步完成“第二级分解”

第二级分解就是把数据流图中的每个处理映射成软件结构中一个适当的模块。

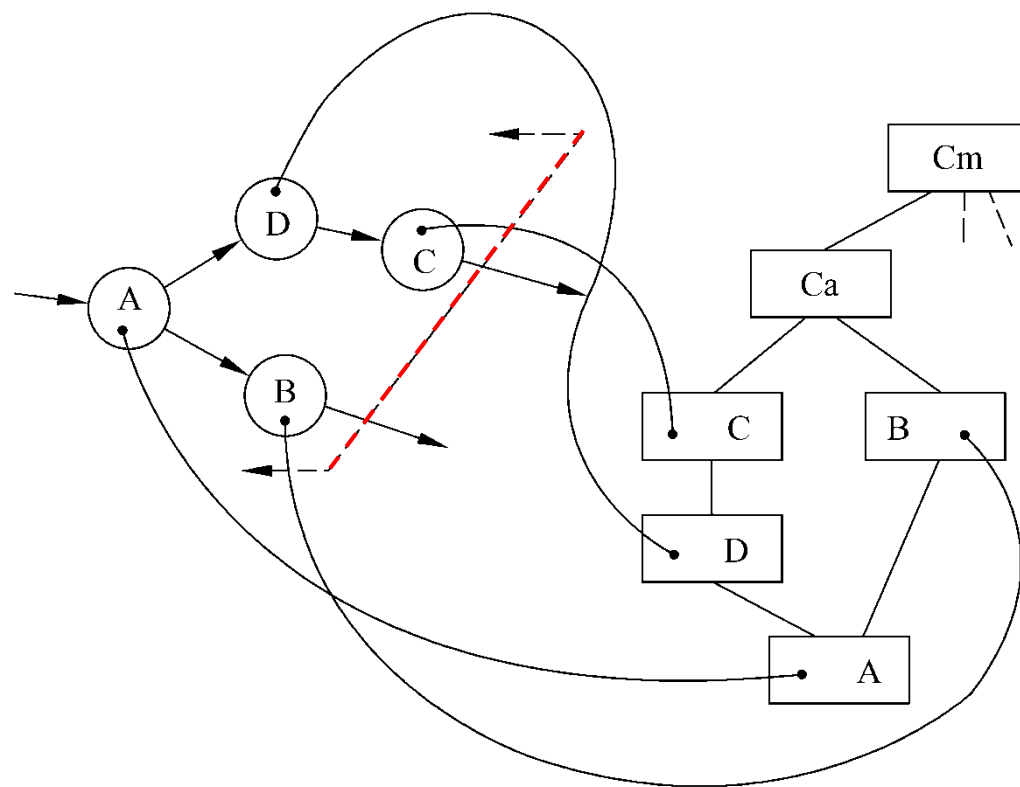
完成第二级分解的方法是，从变换中心的边界开始逆着输入通路向外移动，把输入通路中每个处理映射成软件结构中Ca控制下的一个低层模块；然后沿输出通路向外移动，把输出通路中每个处理映射成直接或间接受模块Ce控制的一个低层模块；最后把变换中心内的每个处理映射成受Ct控制的一个模块。

## 5.5 面向数据流的设计方法

### ② 设计步骤

第6步完成“第二级分解”

右图表示进行第二级分解的普遍途径。

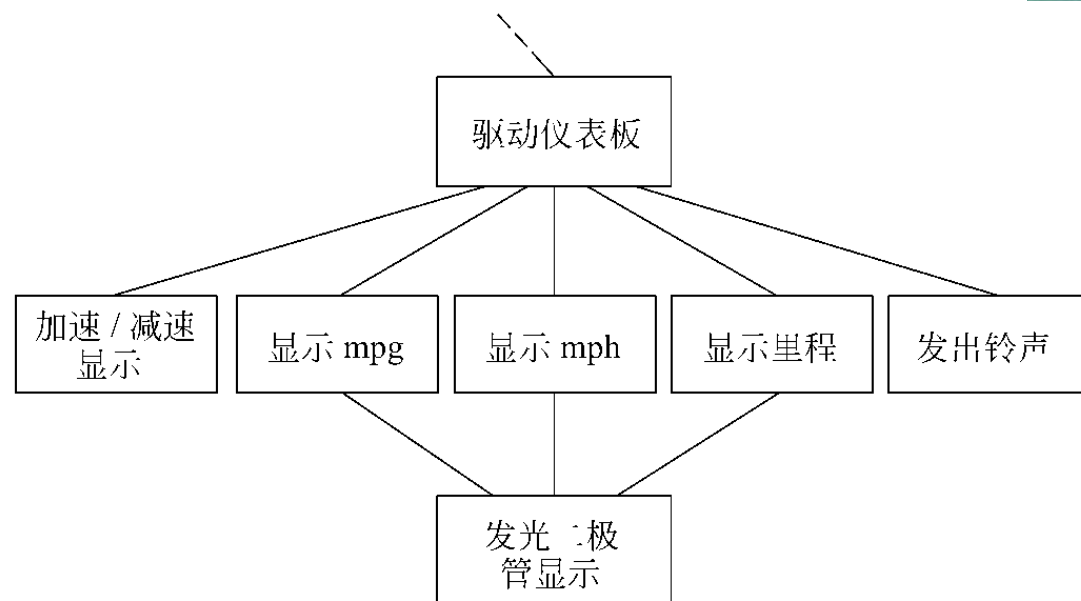
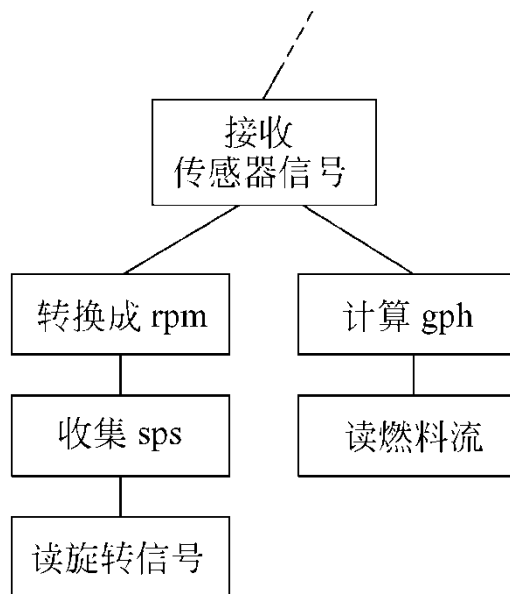
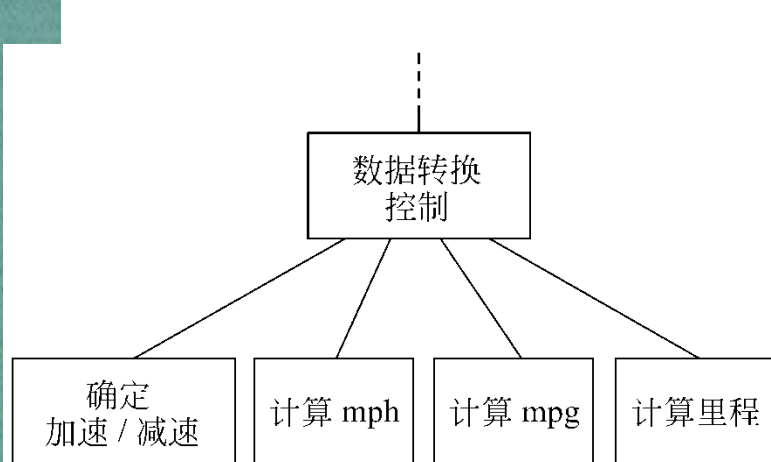


## 5.5 面向数据流的设计方法

### ② 设计步骤

#### 第6步完成“第二级分解”

- 应该根据实际情况以及“好”设计的标准，进行实际的第二级分解。
- 对于数字仪表盘系统的例子，第二级分解的结果分别用下面三张图描绘：



# 5.5 面向数据流的设计方法

## ② 设计步骤

### 第6步完成“第二级分解”

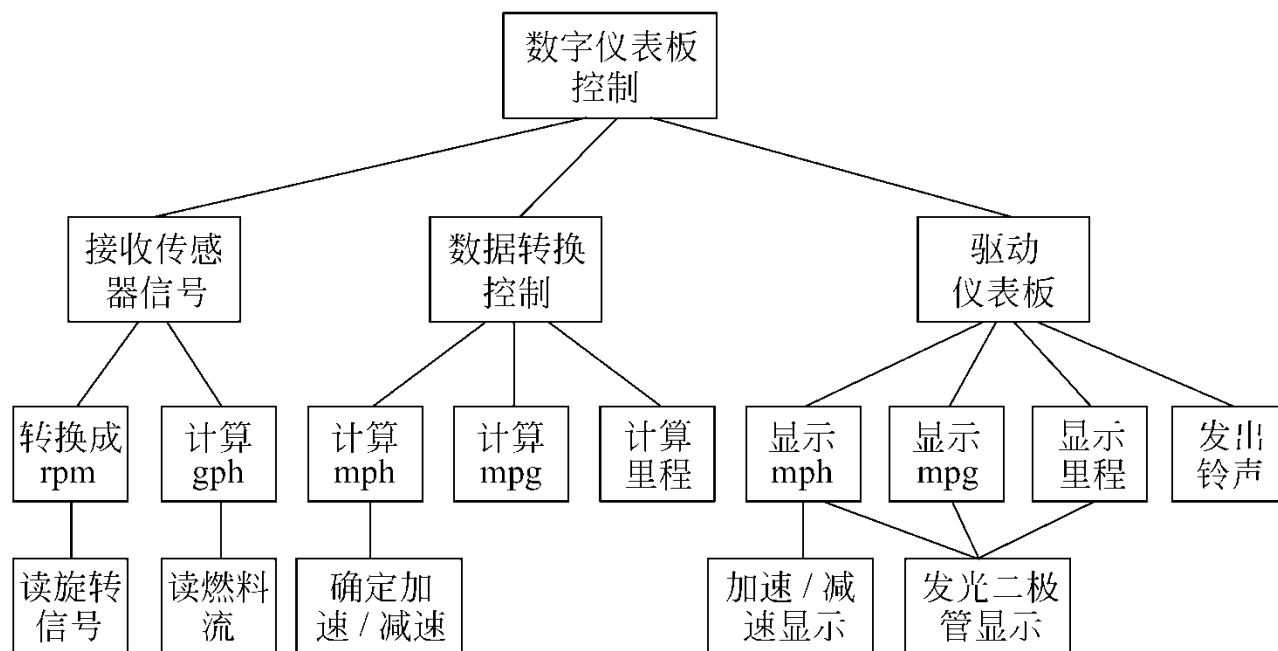
为每个模块写一个简要说明，描述以下内容：

- 进出该模块的信息(接口描述)。
- 模块内部的信息。
- 过程陈述，包括主要判定点及任务等。
- 对约束和特殊特点的简短讨论。

## 5.5 面向数据流的设计方法

### ② 设计步骤

第7步使用设计度量和启发式规则对第一次分割得到的软件结构进一步精化。



# 5.5 面向数据流的设计方法

## 5.5.5 设计优化

- 设计人员应该致力于开发能够满足所有功能和性能要求，而且按照**设计原理**和**启发式设计规则**衡量是值得的软件。
- 设计的早期阶段尽量对软件结构进行精化。
- 对时间起决定性作用的软件进行优化是合理的。
  - (1) 在不考虑时间因素的前提下**开发并精化**软件结构。
  - (2) 在详细设计阶段选出最耗费时间的那些模块，仔细地设计它们的处理过程(**算法**)，以求**提高效率**。
  - (3) 使用**高级程序设计语言**编写程序。
  - (4) 在软件中孤立出那些大量**占用**处理机资源的模块。
  - (5) 必要时重新设计或用依赖于机器的语言**重写**上述大量占用资源的模块的代码，以求提高效率。





## 本章小结

- 1.总体设计阶段主要由系统设计和结构设计两阶段组成。
- 2.进行软件结构设计时应该遵循的最主要的原理是模块独立原理。
- 3.在软件开发过程中既要充分重视和利用这些启发式规则，又要从实际情况出发避免生搬硬套。
- 4.层次图和结构图是描绘软件结构的常用工具。
- 5.用形式化的方法由数据流图映射出软件结构。

以下做课堂练习。