软件工程导论 –

第5章 总体设计

2021.05.

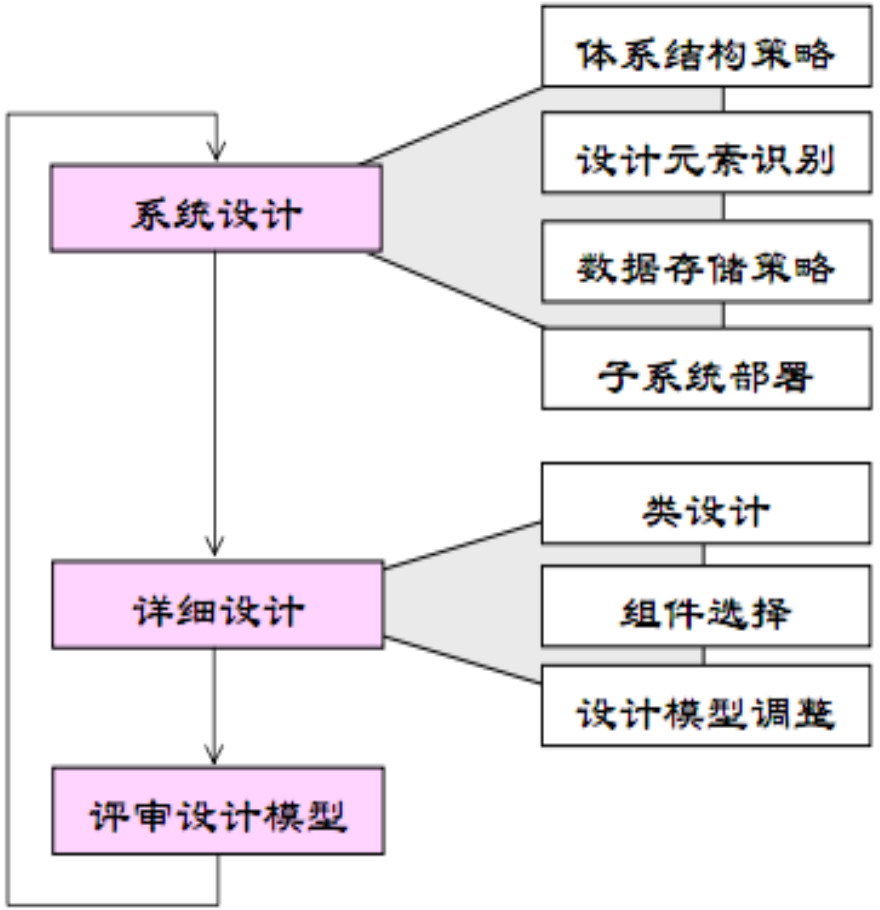
第5章 总体设计

- 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

第五章 总体设计

- 设计是研究系统的软件实现问题，即在分析模型的基础上形成实现环境下的设计模型；
- 设计主要涉及以下几个方面：

体系结构设计、详细设计、用户界面设计、数据库设计等方面。

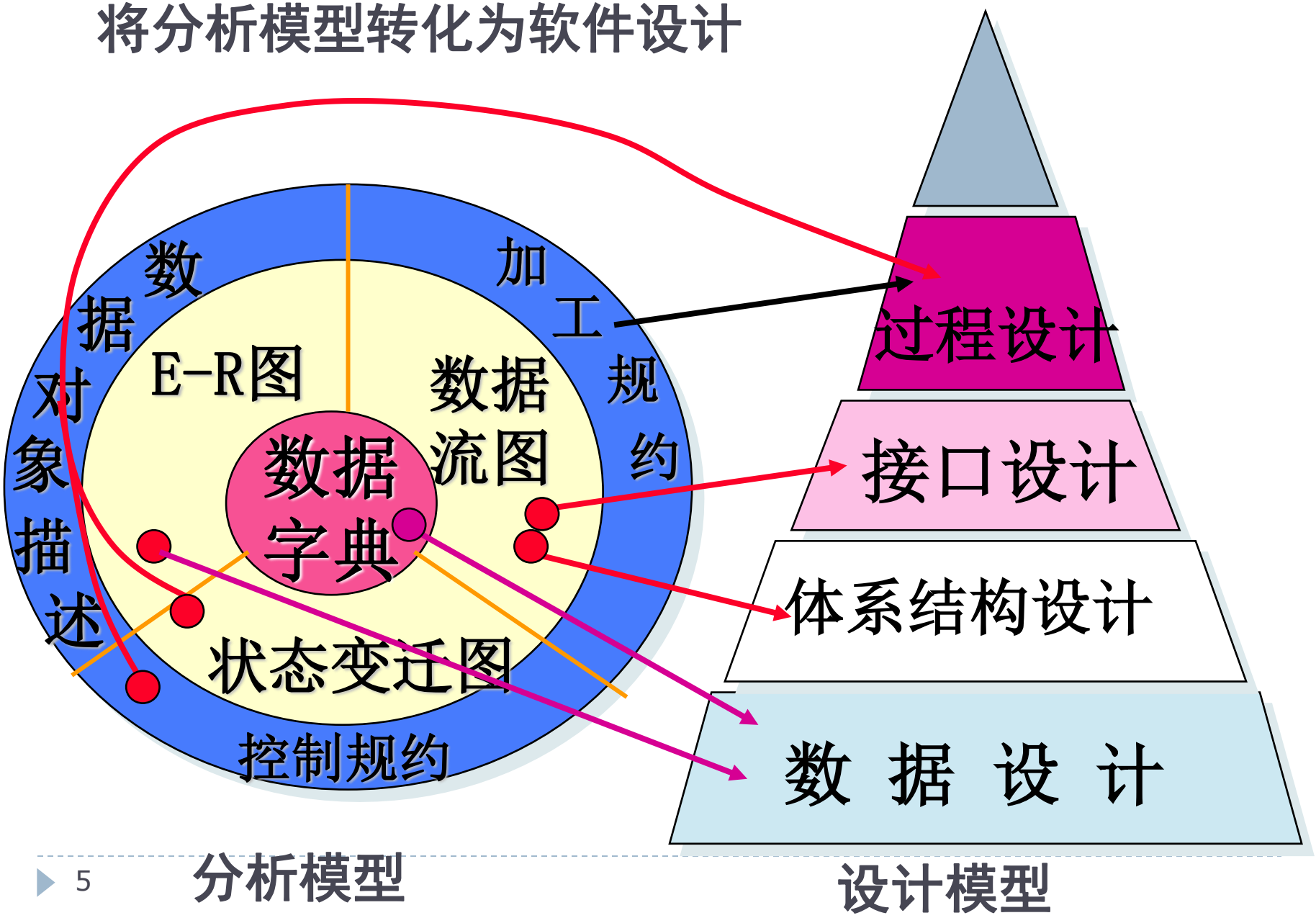


设计者的目标：

生成一个随后要构造的实体的模型或表示。

- 数据流图 → 接口设计，体系结构设计
- 数据字典 → 数据设计
- 加工 → 过程设计

将分析模型转化为软件设计

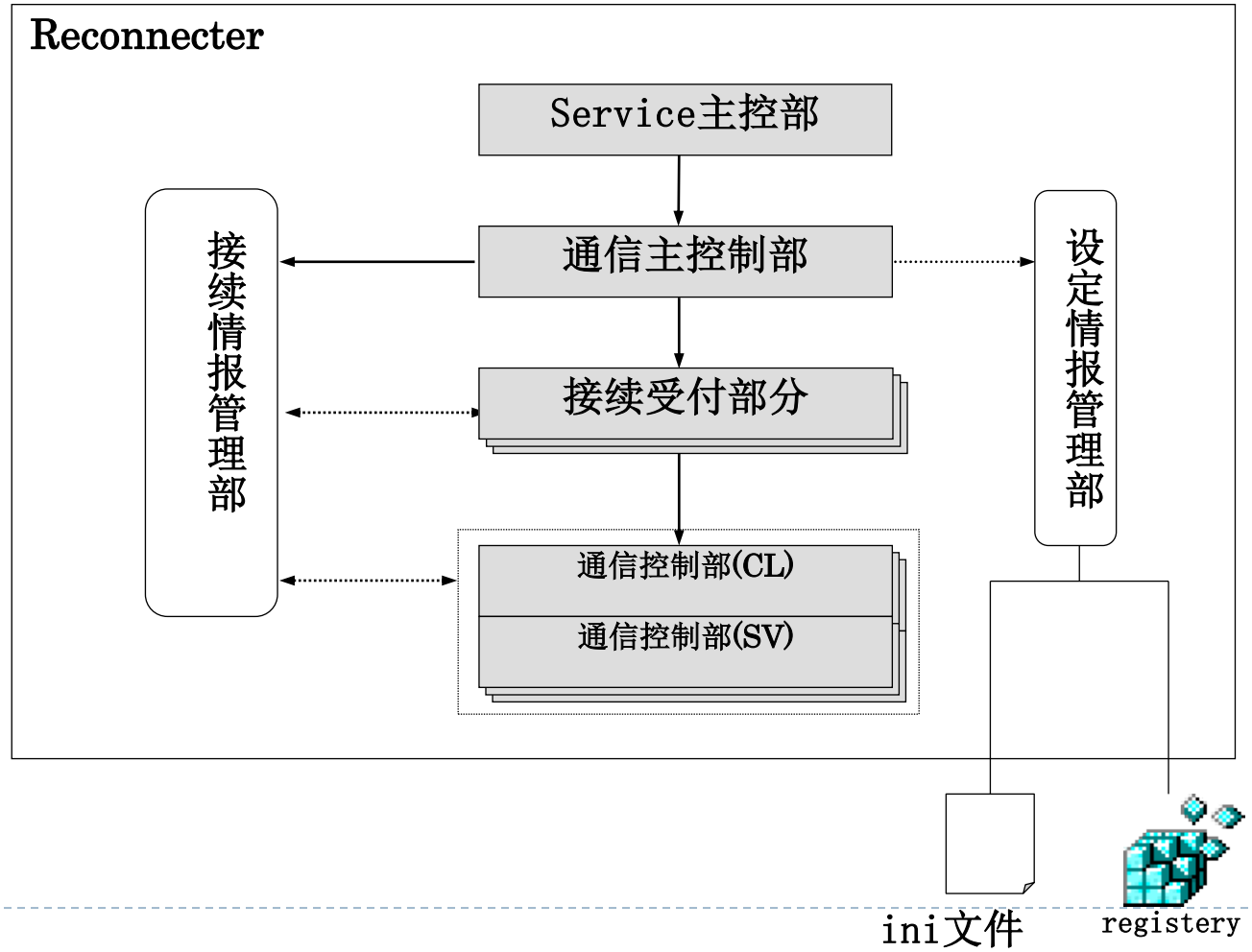


- **数据设计：**
  数据结构

- **体系结构设计：**
  软件的主要结构性元素

- **接口设计：**
  软件内部，软件和协作系统之间
  以及软件同人之间如何通信。

体系结构图例


Reconnecter
Service主控部
通信主控制部
接续受付部分
通信控制部(CL)
通信控制部(SV)
接续情报管理部
设定情报管理部
ini文件
registery

- 设计不是编码，编码也不是设计。

5.1 总体设计的过程

总体设计通常由系统设计和结构设计两个阶段组成。

系统设计阶段确定系统的具体实现方案，
结构设计阶段确定软件的结构。

实施总体设计的过程

1. 设想供选择的方案
2. 选取合理的方案
3. 推荐最佳方案
4. 功能分解
5. 设计软件结构
6. 数据库设计
7. 制定测试计划
8. 书写文档
9. 审查和复审

设计原则

- 模块化（*Modularity*）
  - 将一个复杂的大系统分解成若干个相对简单的较小部分，称为子系统（*Subsystem*）。
- 常见的模块层次

5.2 设计原理

- 将系统划分成模块
- 决定每个模块的功能
- 决定模块的调用关系
- 决定模块的界面，即模块间传递的数据

5.2.1 模块化

- **模块化论据：**

  C(x)定义为问题x的感知复杂性

  E(x)定义为解决问题x所需要的工作量

  对p1和p2两个问题，

  若 C(p1) > C(p2)，则 E(p1) > E(p2)

  C(p1 + p2) > C(p1) + C(p2)

  E(p1 + p2) > E(p1) + E(p2)

- 不要过度模块化！每个模块的简单性将被集成的复杂性所掩盖。

模块

- 用一个名字可以调用的一段可执行程序语句。
- 具有**输入和输出**，**功能**，**内部数据**，**程序代码**等四个特性。
- 完成一个独立功能的程序单元。

模块

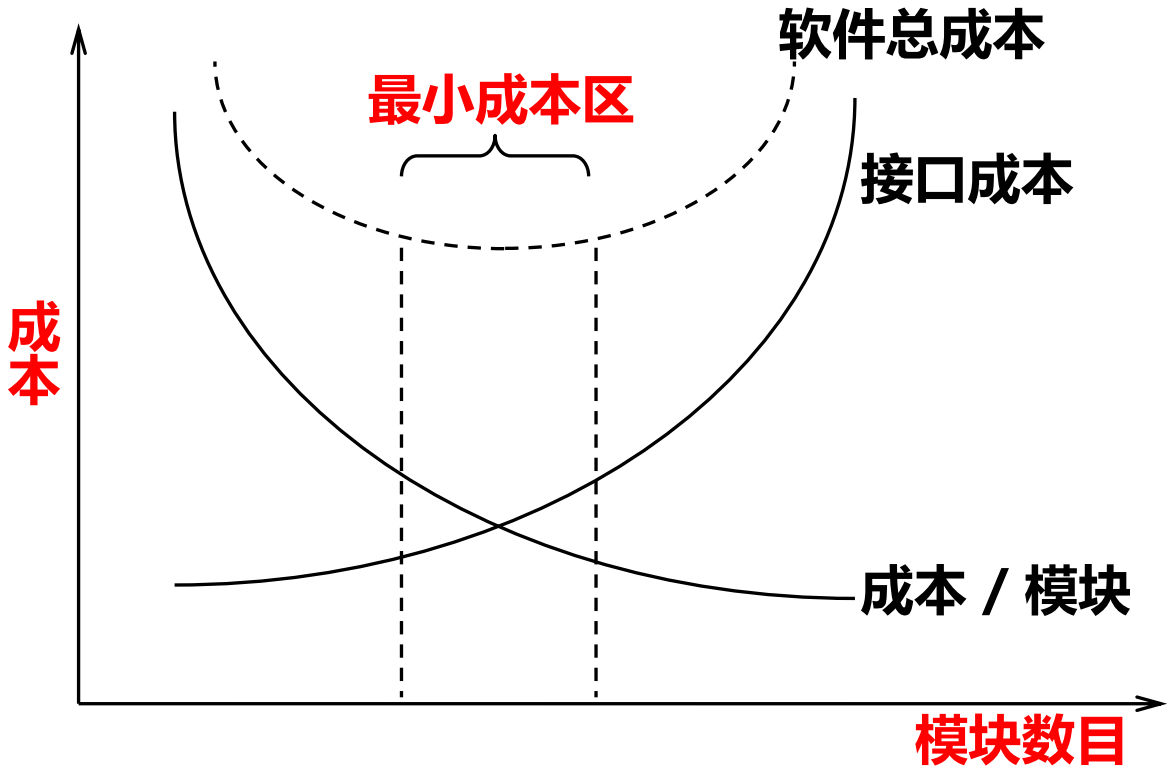
把大型软件按照规定的原则划分为一个个较小的、相对独立但又相关的模块的设计方法，叫做模块化设计（modular design）。

实现模块化设计的重要指导思想是分解、信息隐藏和模块独立性。

降低系统的开发难度，增加系统的可维护性。

模块化和软件成本

如何确定地预测最小成本区？



如何确定模块的大小：

- 模块可分解性
- 模块可组装性
- 模块可理解性
- 模块连续性
- 模块保护

信息隐藏

模块内部的数据与过程，应该对不需要了解这些数据与过程的模块隐藏起来。只有那些为了完成软件的总体功能而必需在模块间交换的信息，才允许在模块间进行传递。

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

这一指导思想的目的是为了提高模块的独立性，即当修改或维护模块时减少把一个模块的错误扩散到其他模块中去的机会。

自顶向下逐步细化

- **自顶向下设计的特点：**
  - a、易于修改和扩展
  - b、整体测试较易通过
  - c、需要进行详细的可行性论证
- **由底向上设计的特点：**
  - a、可能导致较大的重新设计
  - b、整体测试中可能在模块接口间发现不一致等问题
  - c、如果在可行性上出现问题，可以较早发现

模块独立性

模块独立性（module independence）概括了把软件划分为模块时要遵守的准则，也是**判断模块构造是否合理的标准**。

模块的独立性可以由两个定性标准度量，这两个标准分别称为**内聚**和**耦合**。

耦合用于衡量不同模块彼此间互相依赖（连接）的紧密程度；

内聚用于衡量一个模块内部各个元素间彼此结合的紧密程度。

一般地，坚持模块的独立性是获得良好设计的关键。

块间联系（耦合度）

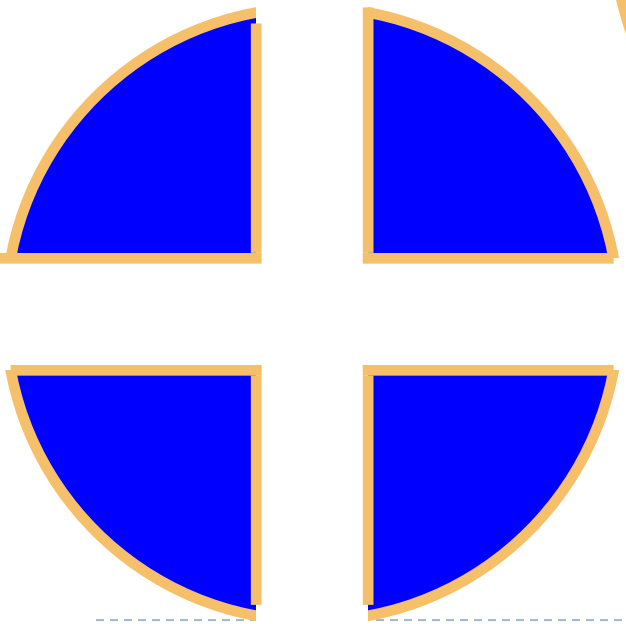
- 耦合：模块之间的联系。

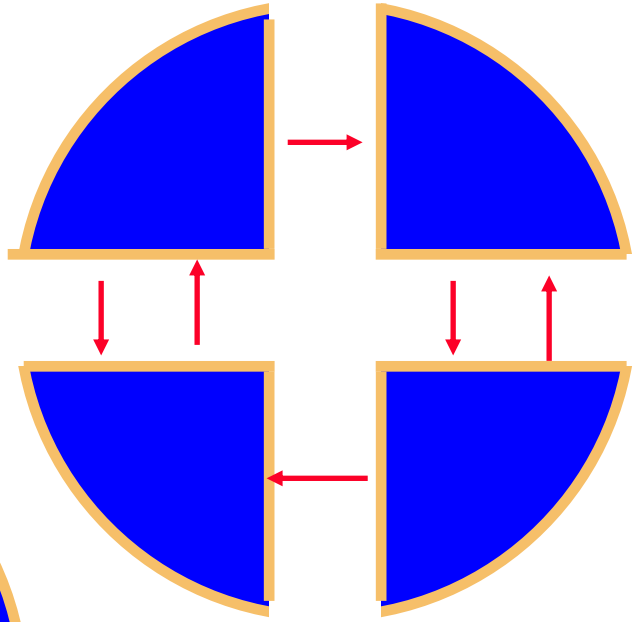
- 耦合是对一个软件结构内不同模块之间互联程度的度量。耦合强弱取决于模块间接口的复杂程度、进入或访问一个模块的点以及通过接口的数据。

- 当一个模块（子系统）发生变化时，对另一个模块（子系统）的影响很小，则称他们是松散耦合的；反之是紧密耦合的。

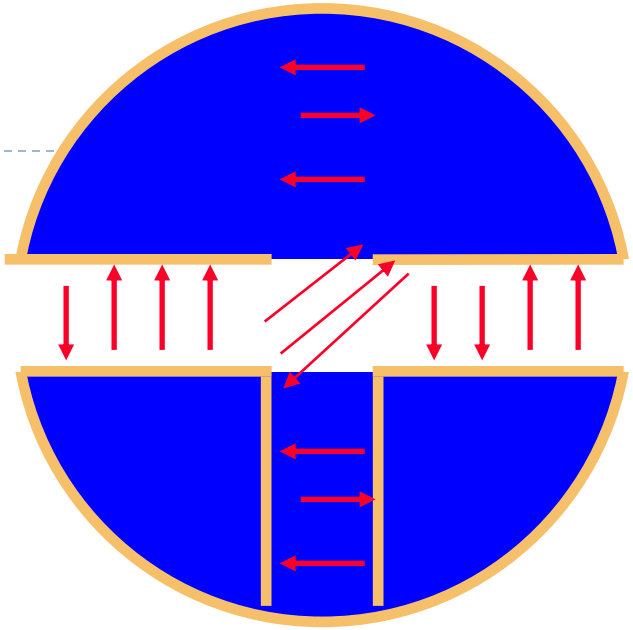
模块耦合度

无耦合 - 没有依赖关系





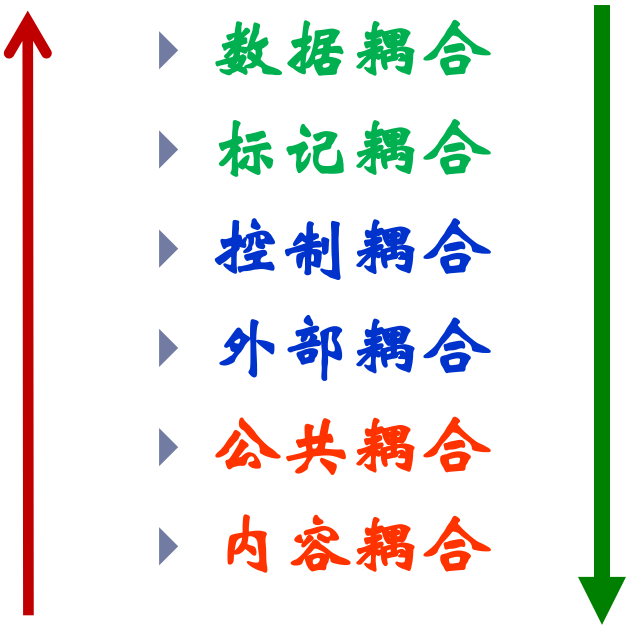
松散耦合 - 有少量依赖关系



紧密耦合 - 有很多依赖关系

耦合的七种类型

模块独立性自下而上递增



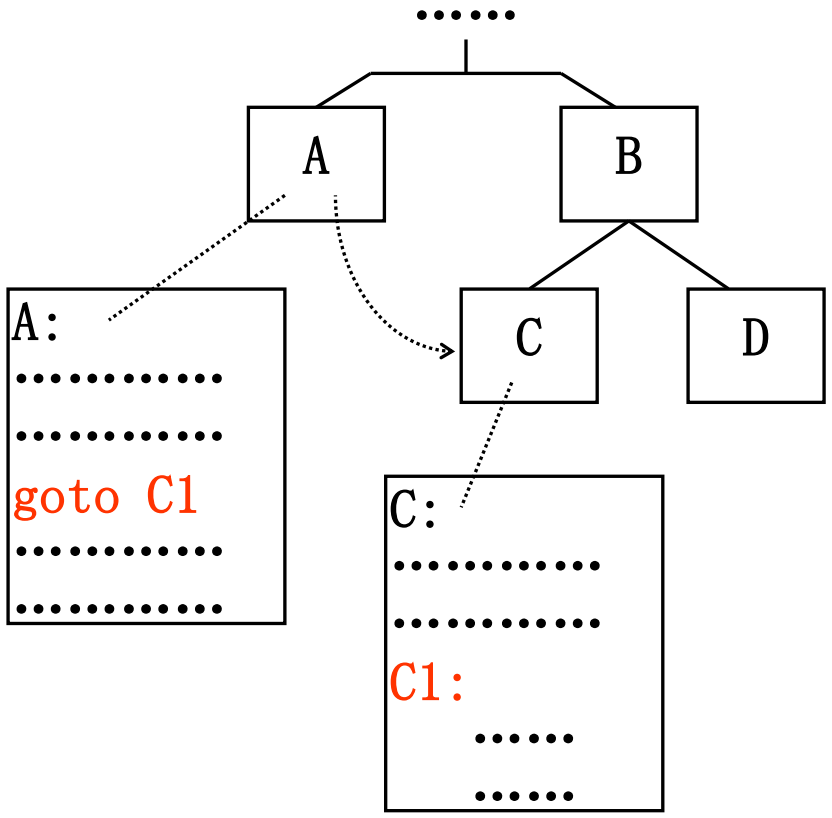
- 数据耦合
- 标记耦合
- 控制耦合
- 外部耦合
- 公共耦合
- 内容耦合

耦合度自上而下递增

1. 内容耦合 content coupling

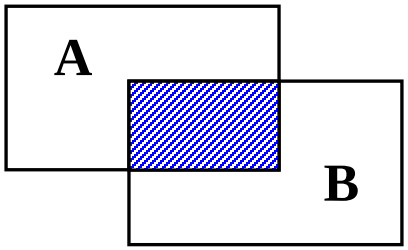
- 如果两个模块中的一个直接引用了另一个模块的内容，则它们之间是内容耦合。

例1：A访问C的内部数据或不通过正常入口而转入C的内部。

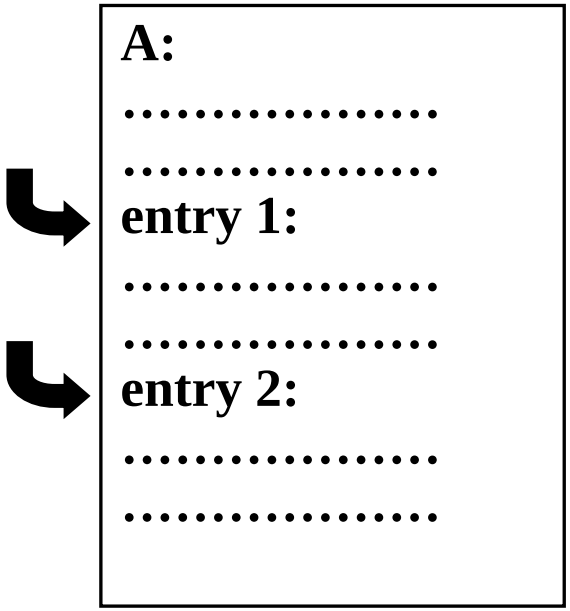


内容耦合 content coupling

例2：部分代码重叠

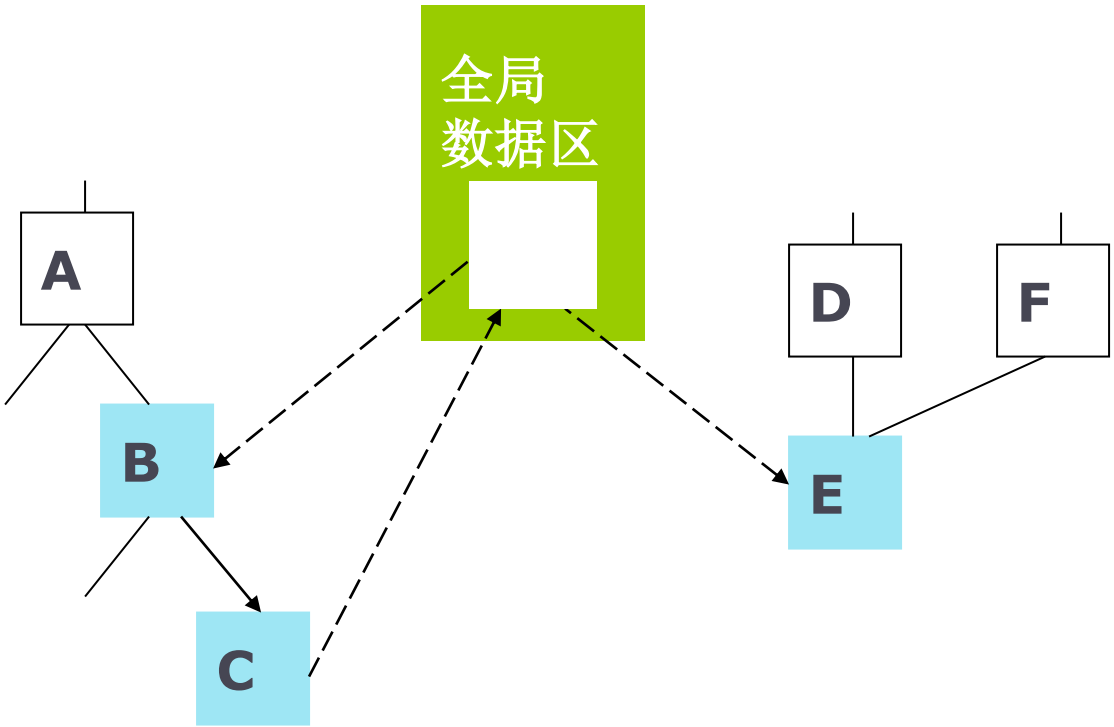


例3：一个模块有多个入口（功能）



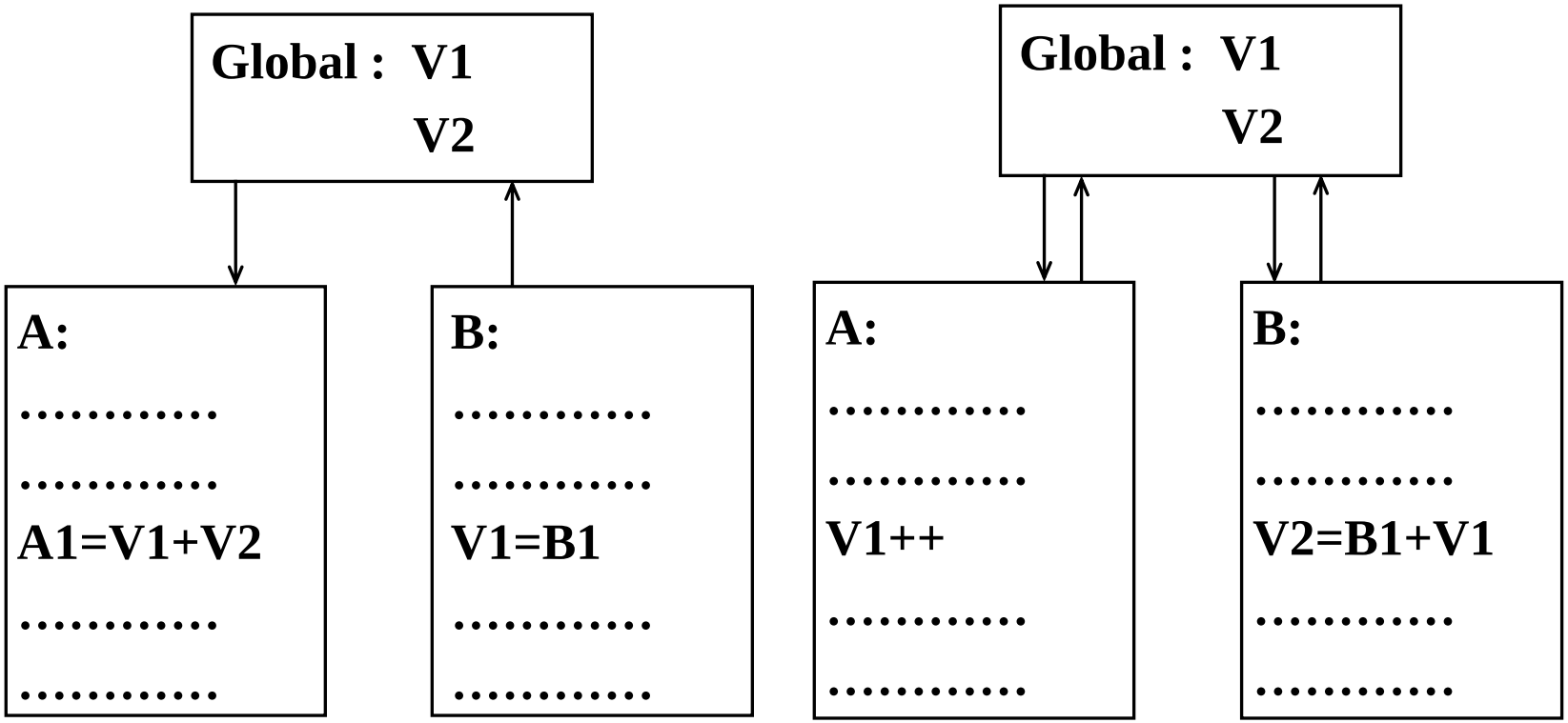
2. 公共耦合 common coupling

- 如果两个模块都可以存取相同的全局数据，则它们之间是公共耦合。



B、C、E 为公共耦合

公共耦合的例子，使用了全局变量

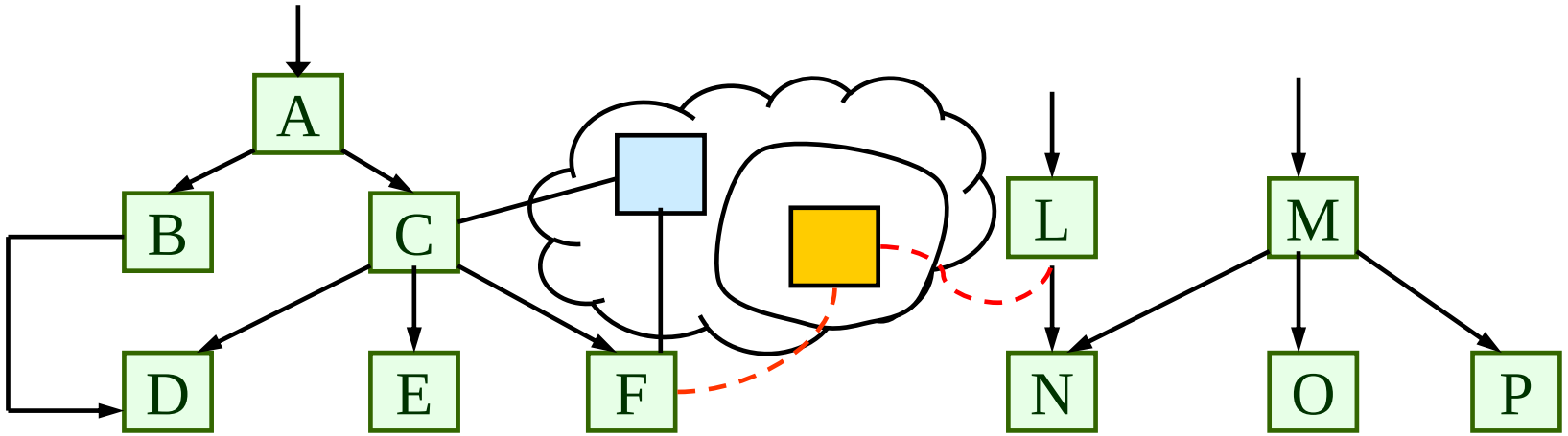


公共耦合 common coupling

- 公共耦合存在的问题：
  - 公共部分的改动将影响所有调用它的模块；
  - 公共部分的数据存取无法控制；
  - 复杂程度随耦合模块的个数增加而增加。
- 解决方法：
  - 通过使用信息隐藏来避免公共耦合。

3. 外部耦合 external coupling

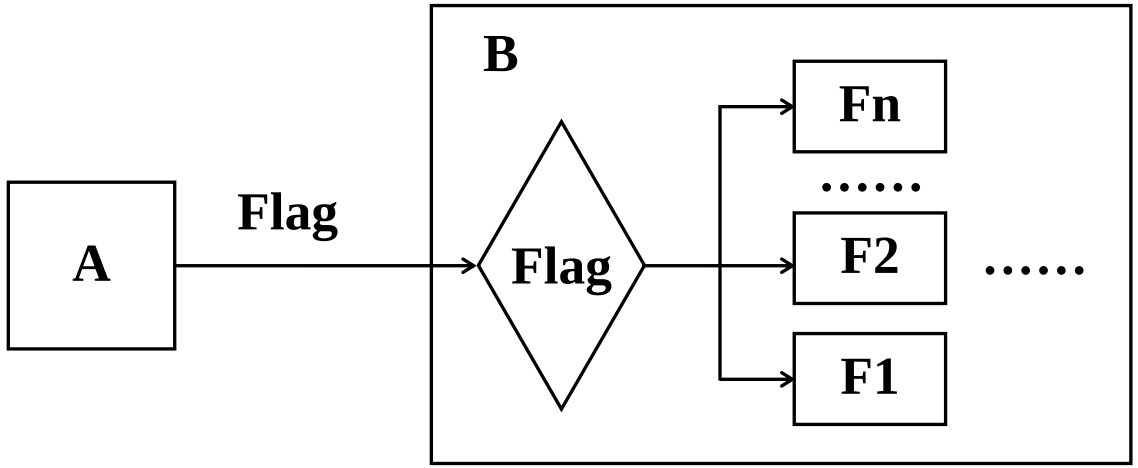
- 当模块与软件的外部环境联结在一起，并受到约束时就出现较高程度的耦合，则它们之间为外部耦合。



F和L为外部模块；C、F为公共模块；B和D为内容模块

4. 控制耦合 control coupling

- 如果两个模块中的一个模块给另一个模块传递控制信息，则它们具有控制耦合。

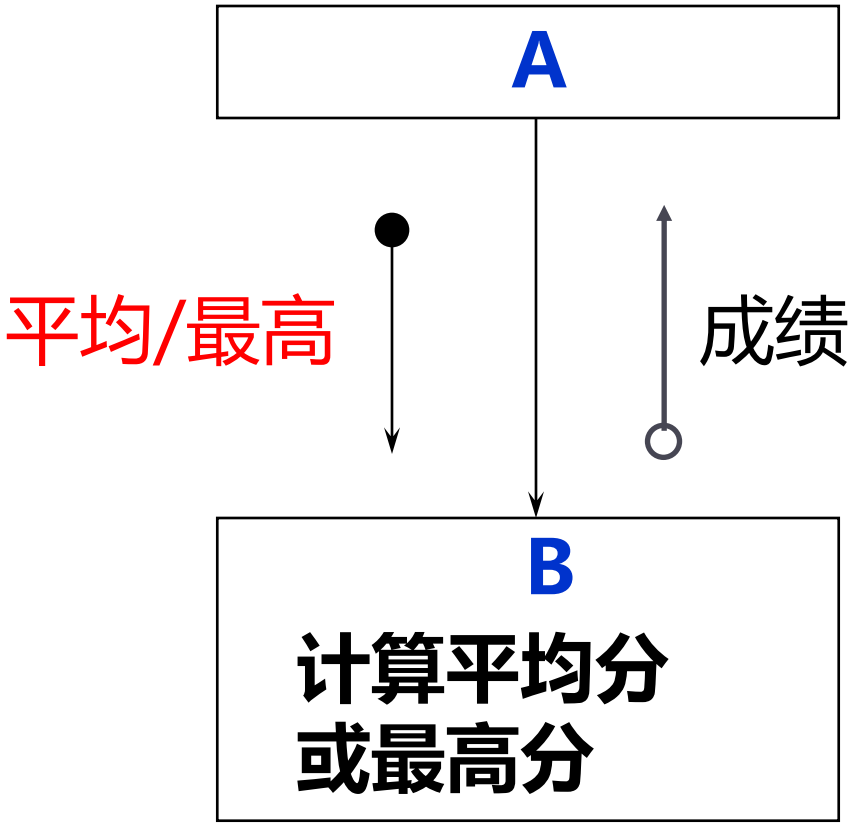


特点：接口单一，但仍然影响被控模块的内部逻辑。

控制耦合举例

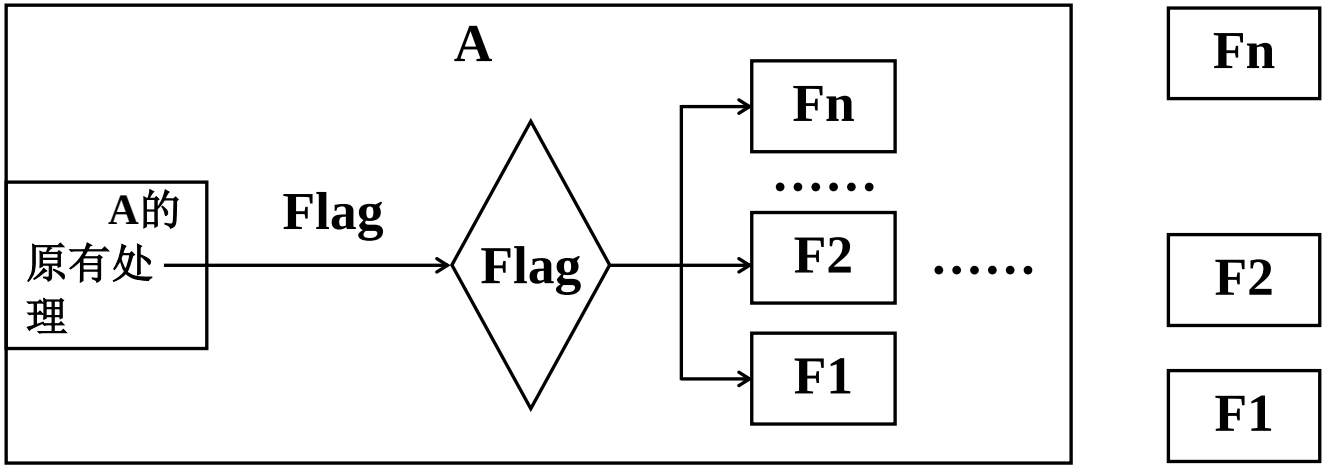
调用逻辑性模块B时，须先传递控制信号(平均分/最高分)，以选择所需的操作。

控制模块必须知道被控模块的内部逻辑，增强了相互依赖.

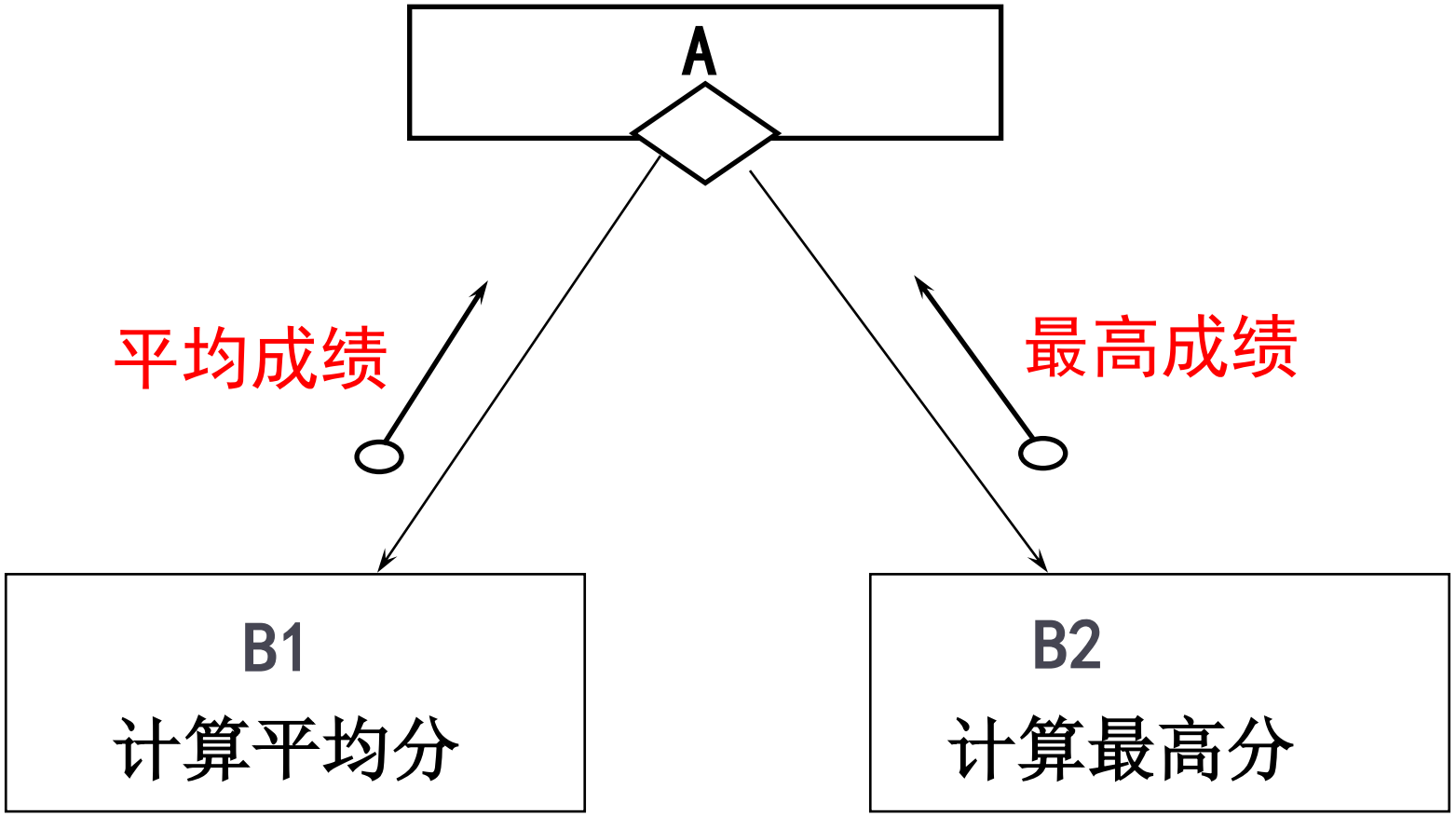


参考，解决方法

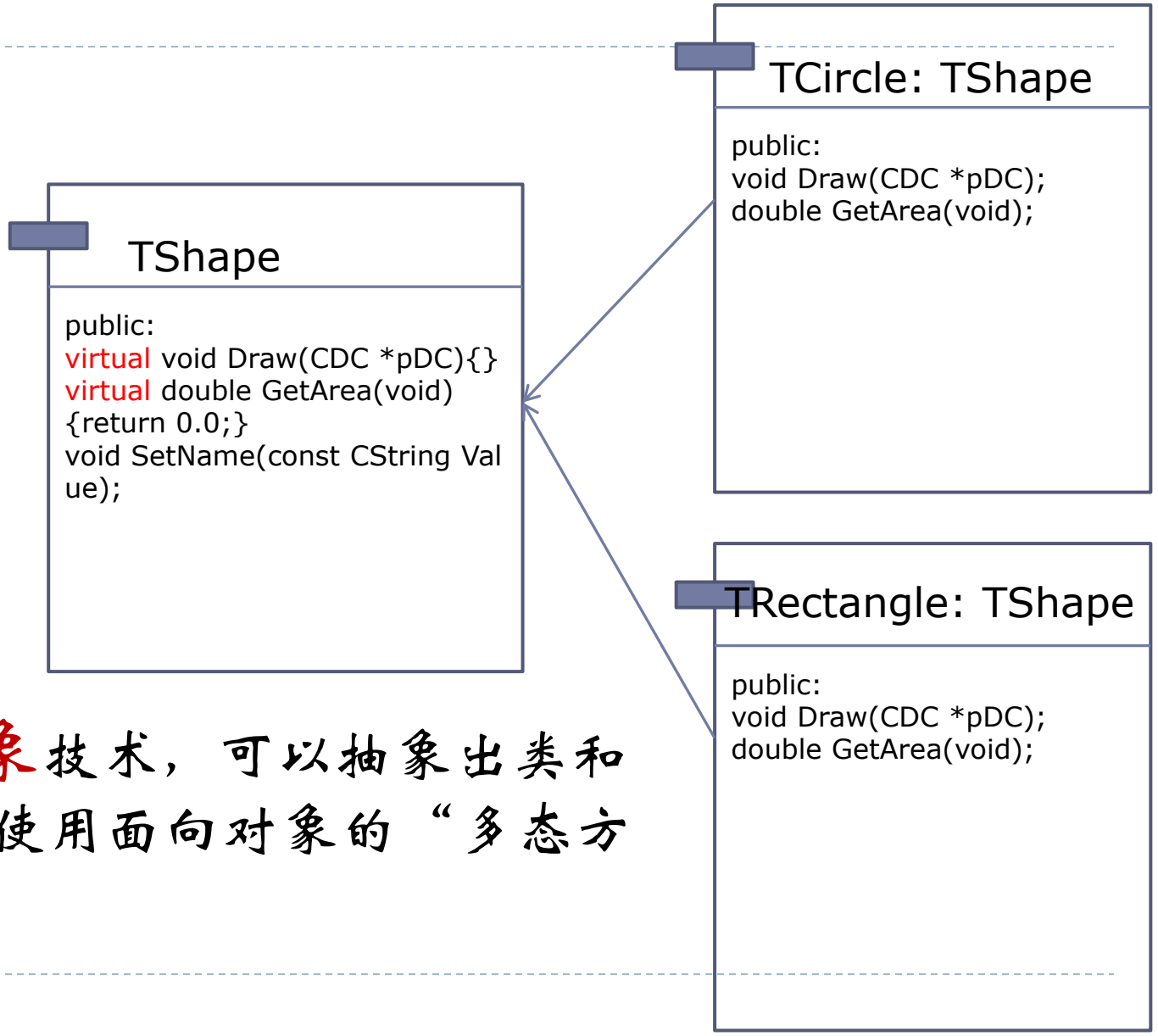
- 面向过程的修改方法：
  - 在调用点进行条件判断；
  - 将B函数按照子功能拆分：



改控制耦合为数据耦合举例



参考，使用面向对象技术的解决方法



- 采用**面向对象**技术，可以抽象出类和子类的情况，使用面向对象的“多态方法”解决。

调用点代码示例 – 画圆：

```
++++++++++++画圆++++++++++++++++++++
void CCircleTestDlg::OnCircle()
{
  // TODO: Add your control notification handler code here
  TShape *AShape=new TCircle(CPoint(rand()%480,rand()%300),
                             10+rand()%100);
   char buffer[20];
   CDC *pDC= new CClientDC(this);
   _itoa(rand()%200,buffer,10);
   CString TName=(CString)"我的名字是Circle"+buffer+";";
   AShape->SetName(TName);
   AShape->Draw(pDC);
   。。。

}
```

调用点代码示例– 画方形：

```
++++++++++++画方形++++++++++++++++++++
void CCircleTestDlg::OnTRectangle()
{// TODO: Add your control notification handler code here
  int a,b,offset,offset1;
  a=rand()%450;
  b=rand()%250;
  offset=rand()%100+20;
  offset1=offset+rand()%100+10;
  TShape *AShape=new TRectangle(CPoint(a,b),
                                CPoint(a+offset1,b+offset));
  char buffer[20];
  CDC *pDC= new CClientDC(this);
  _itoa(rand()%200,buffer,10);
  CString TName=(CString)"我的名字是TRectangle"+buffer+";";
  AShape->SetName(TName);
  AShape->Draw(pDC);
    。 。 。
}
```

Java的实现

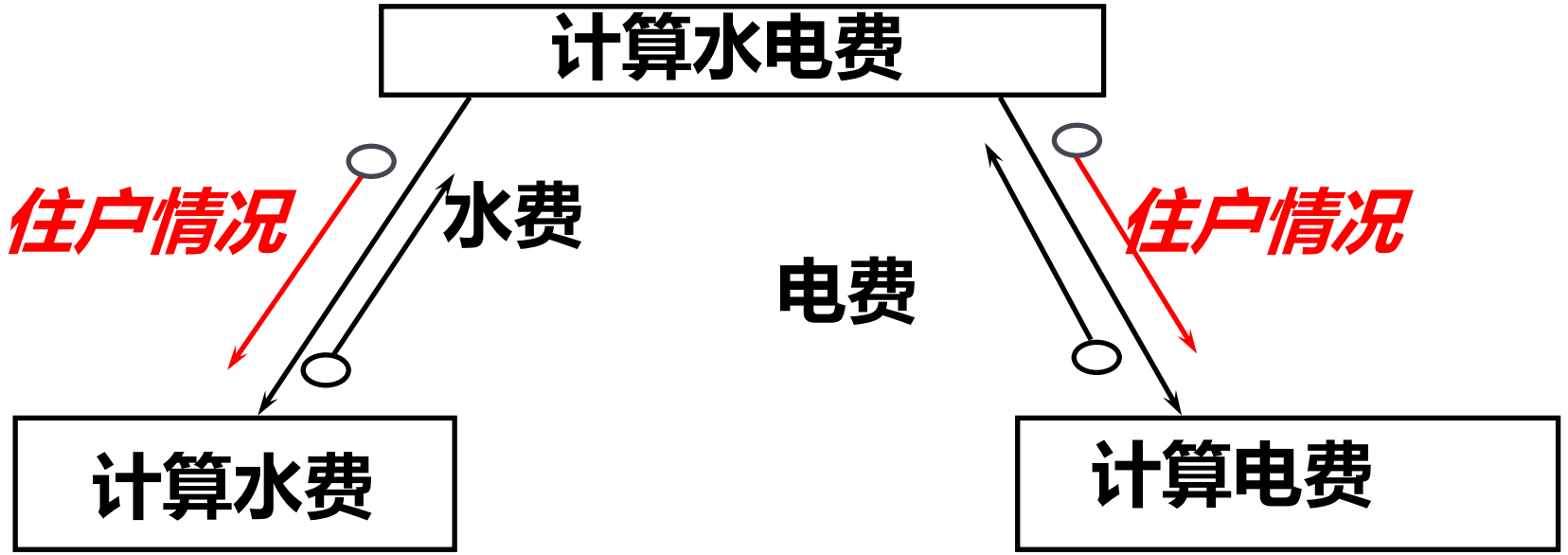
```
Inteface 形状{
      public void 画图();
}

class 圆形 implements 形状{
       public void 画图(){System.out.prinltn("圆形");}
}

class 三角形 implements 形状{
      public void 画图(){System.out.prinltn("三角形");}
}

class Test {
   public static void 画图(形状 graphic){graphic.画图();}

    public static void main(String []args){
         形状 a = new 圆形();
         形状 b = new 三角形();
         Test.画图(a);//打印出 圆形
         Test.画图(b);//打印出 三角形
    }
}
```

5. 标记耦合 stamp coupling

- 如果两个模块都要使用同一数据结构的一部分，不是采用全局公共数据区共享，而是通过模块结构传递数据结构的一部分，则它们之间为标记耦合。

标记耦合 stamp coupling（课本的定义）

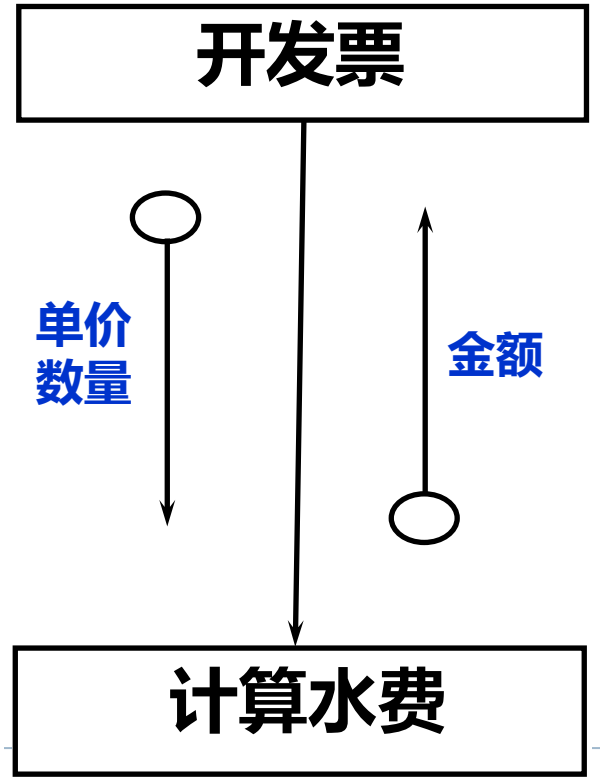
- 两个模块通过传递数据结构(不是简单数据，而是记录、数组等)加以联系，而被调用模块只需要部分数据项，则称这两个模块之间存在标记耦合或特征耦合。



“*住户情况*”包含水费和电费、煤气费、电话费等。
“*住户情况*”是一个数据结构，图中模块都与此数据结构有关.
“计算水费”和“计算电费”本无关，由于引用了此数据结构产生依赖关系，它们之间也是标记偶合.

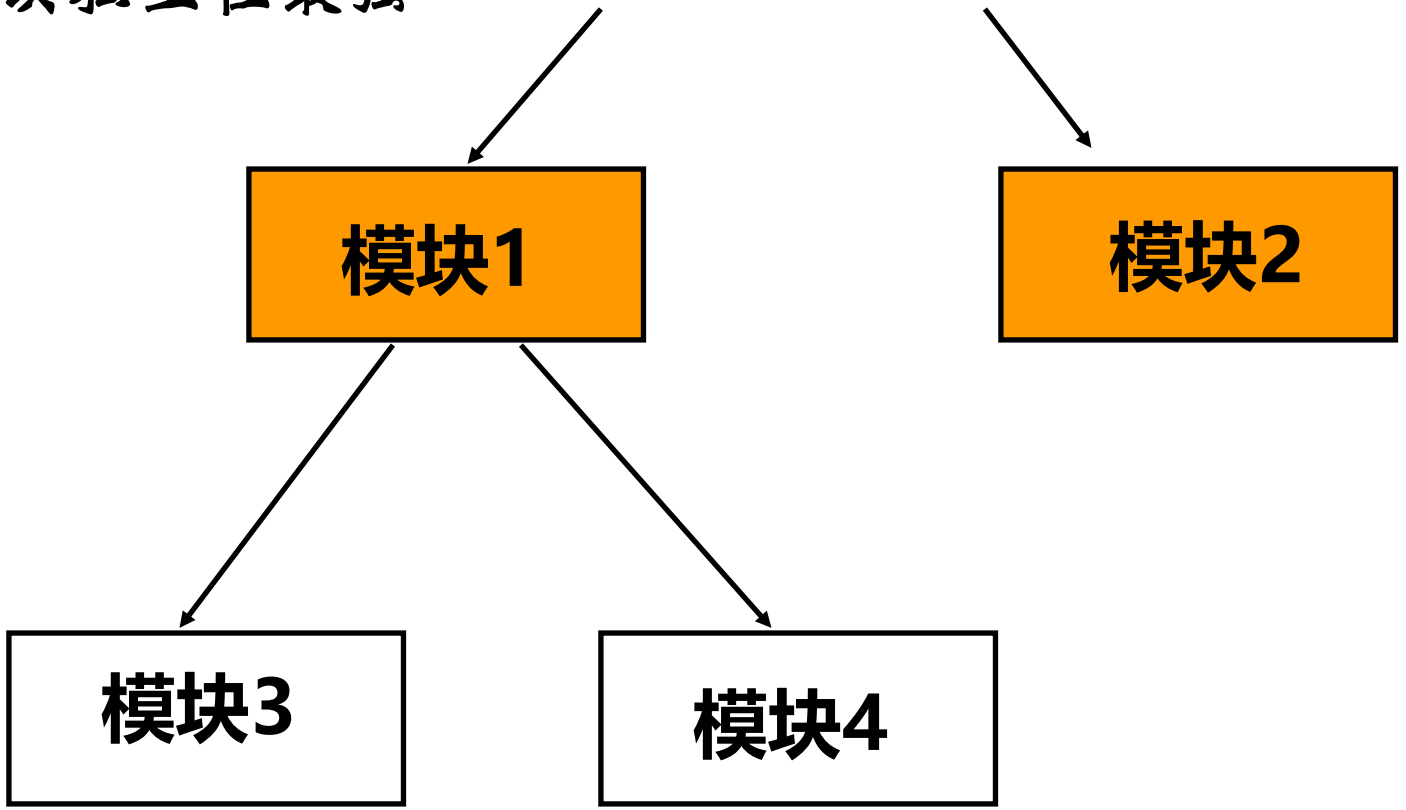
6. 数据耦合 data coupling

- 被调用模块的输入与输出是简单的参数或者是数据结构（该数据结构中的所有元素为被调用的模块使用），则它们之间为数据耦合。

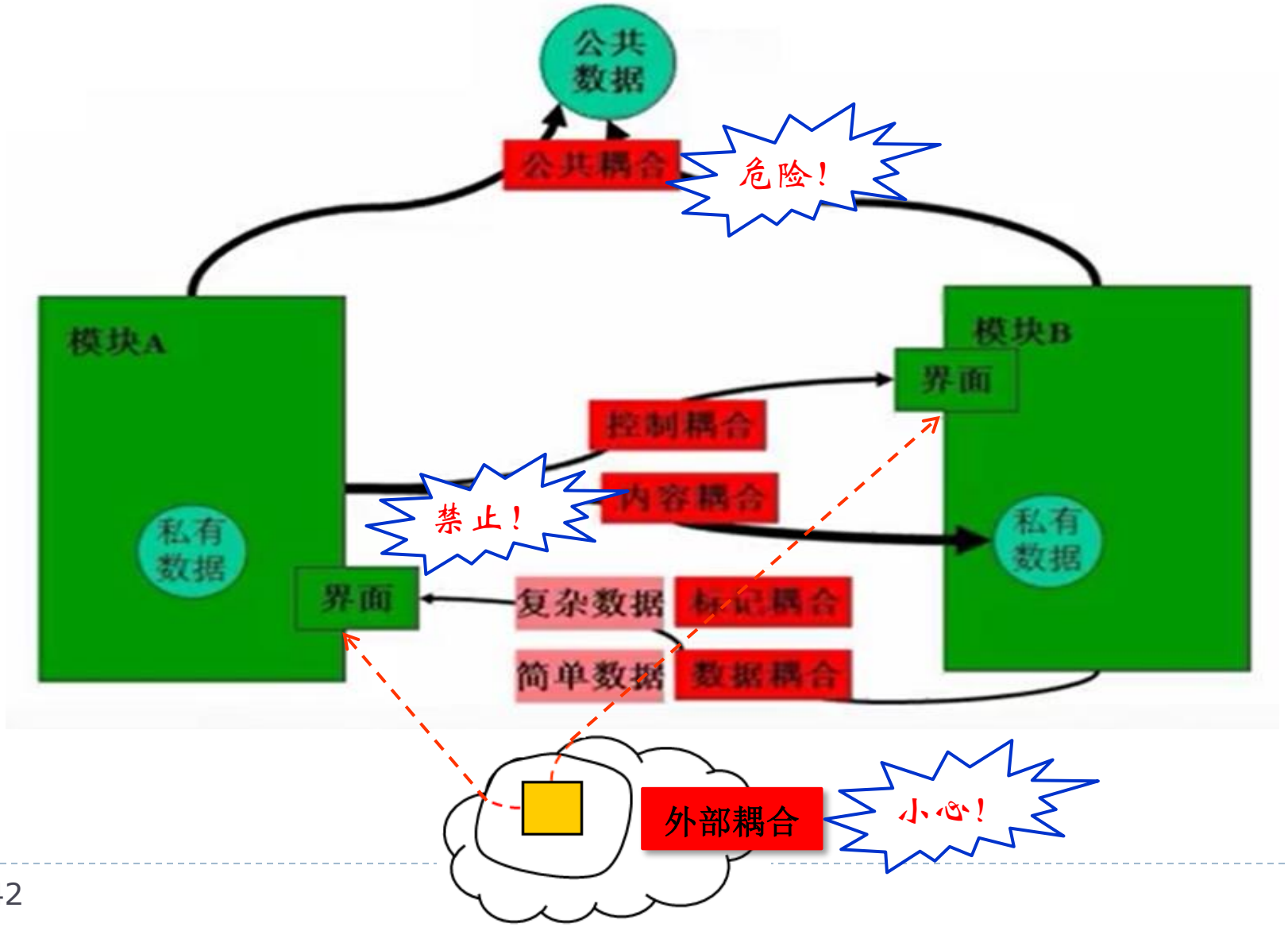


7. 非直接耦合 no direct coupling

- 两个模块之间没有联系，则它们之间为非直接耦合。
- 模块之间的联系是通过主模块的控制和调用实现的，模块独立性最强



几种耦合的图例表示


公共数据
公共耦合
危险！
模块A
私有数据
界面
控制耦合
内容耦合
禁止！
复杂数据
标记耦合
简单数据
数据耦合
模块B
界面
私有数据
外部耦合
小心！

块间联系的原则

- 实现低耦合，采取下列措施：
  - 耦合方式
    - 采用非直接耦合。
  - 传递信息类型
    - 尽量使用数据耦合，少采用控制耦合，外部耦合和公共耦合限制使用。
  - 耦合数量
    - 模块间相互调用时，传递参数最好只有一个。

- 原则：尽量使用数据耦合，少用控制耦合，限制公共耦合的范围，完全不用内容耦合。

块间联系的总结

- 用过程语句调用
- 参数作数据用
- 参数量尽量少

- 消除开关量
- 减少共用信息量

块内联系（聚合度）

- 内聚：模块内部各成分之间的关系。

- 内聚标志着一个模块内部各个元素间彼此结合的紧密程度。简单地说，理想内聚的模块只做一件事情。设计时应该力求做到高内聚，通常中等程度的内聚也是可以采用的，而且效果和高内聚相差不多。但是，坚决不要使用低内聚。

内聚和耦合

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低耦合。内聚和耦合都是进行模块化设计的有力工具。实践表明，内聚更重要，应该把更多注意力集中到提高模块的内聚程度上。

内聚的七种类型

- 偶然内聚（0分）
- 逻辑内聚（1分）
- 时间内聚（3分）
- 过程内聚（5分）
- 通信内聚（7分）
- 顺序内聚（9分）
- 功能内聚（10分）

差

好

1. 偶然内聚(Coincidental cohesion)

- 无关的函数、过程或者数据出现在同一个模块里。
- 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。

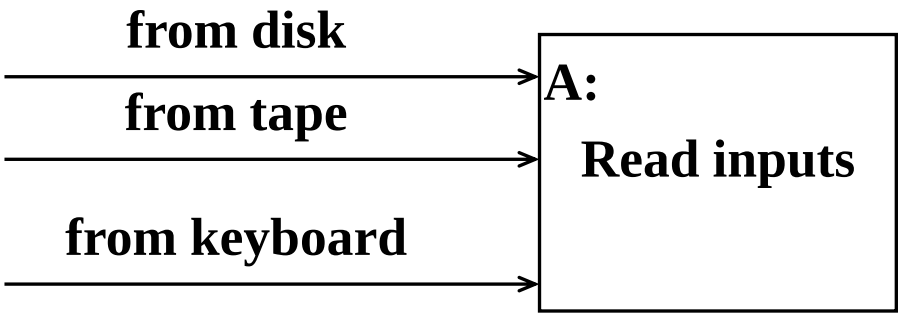
例：read disk file;
calculate current values;
produce user output; …

严重的缺点：产品的可维护性退化；模块是不可复用的，增加软件成本。

解决途径：将模块分成更小的模块，每个小模块执行一个操作。

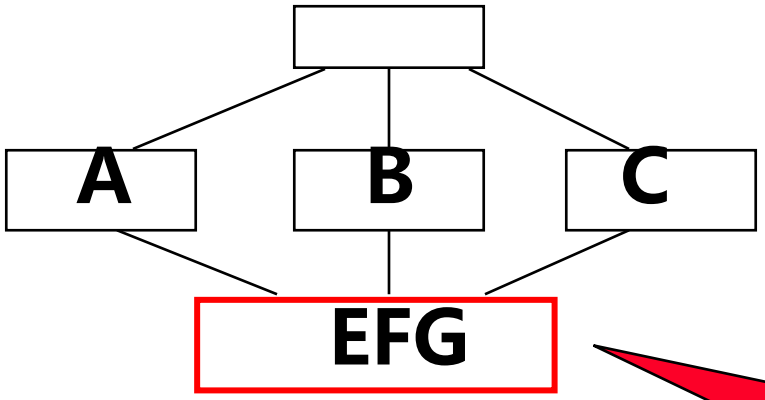
2. 逻辑内聚（Logical cohesion）

- 逻辑相关的函数或数据出现在同一个模块里。（逻辑上相似），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。



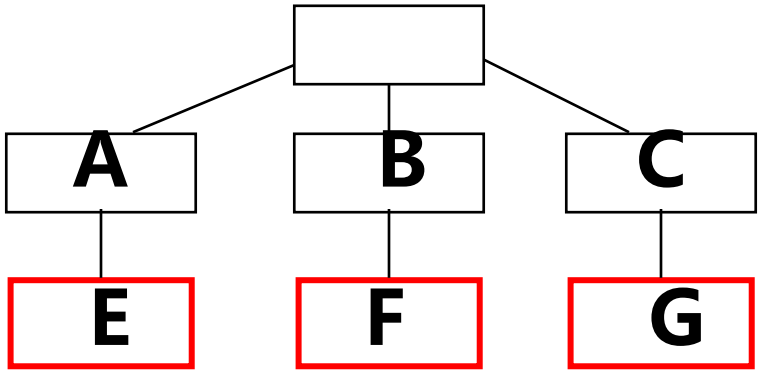
- 问题：接口难于理解；完成多个操作的代码互相纠缠在一起，导致严重的维护问题。

逻辑内聚例子



E、F、G逻辑功能相似，组成新模块EFG

缺点：增强了耦合程度(控制耦合)不易修改，效率低



3. 时间内聚(Temporal cohesion)

- 出现在同一个模块里是由于时间是相同的。（需要同时执行）

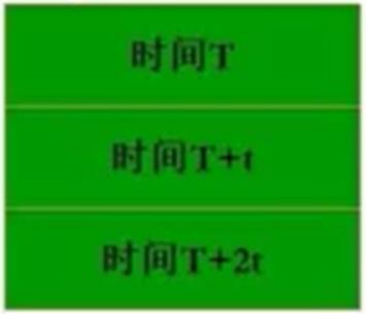
例如：系统的初始化

open old master file;

new master file, transaction file and print file;

initialize sales region table;

read first transaction record and first old master file record;



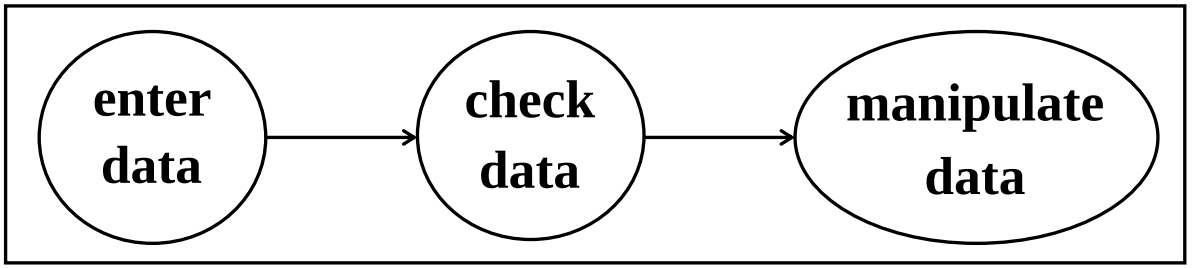
问题：不同的功能混在一个模块中，有时共用部分编码，使局部功能的修改牵动全局。

4. 过程内聚（Procedural cohesion）

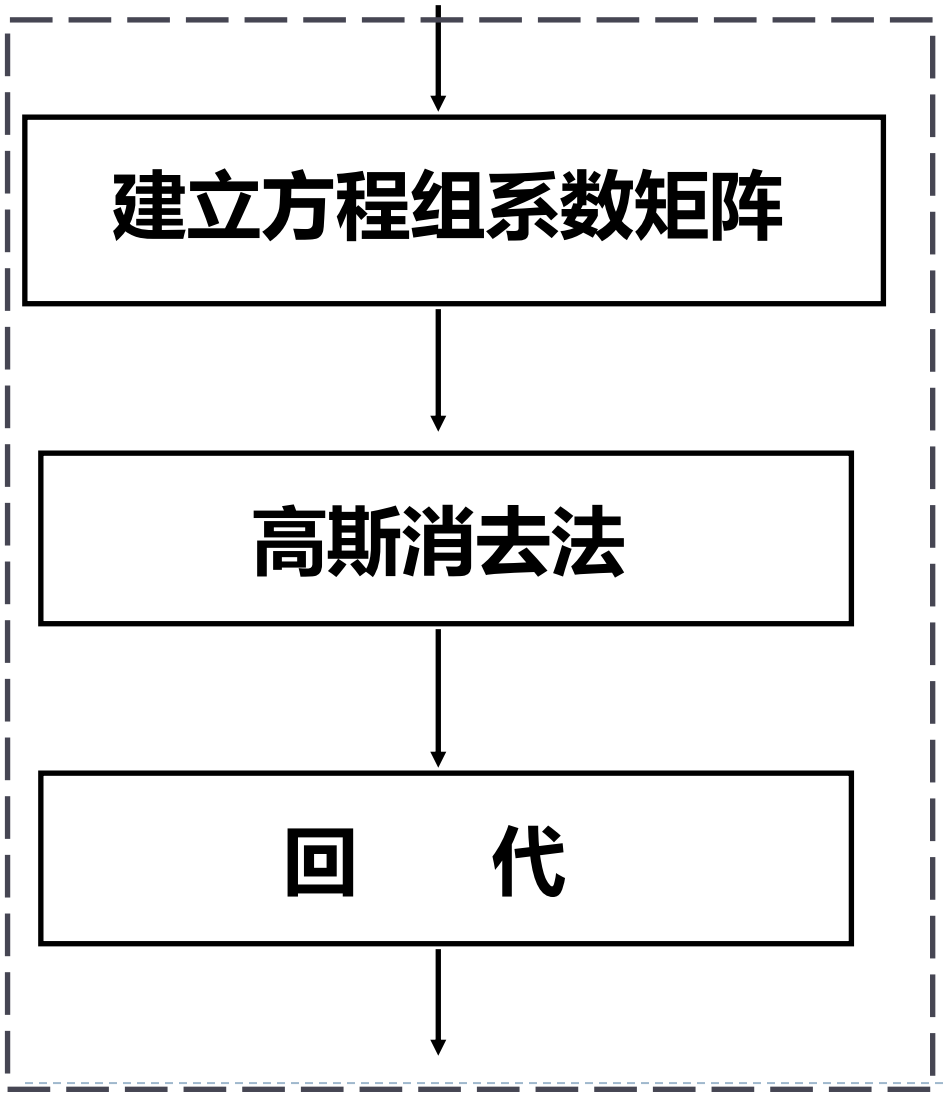
- 出现在同一个模块里是由于处理的顺序。

例子：

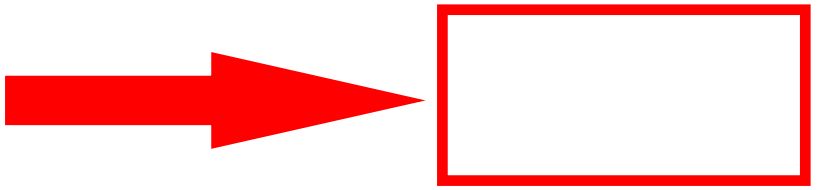
Read part number from database and update repair record on maintenance file.



过程内聚例子：



高斯消去法解题流程

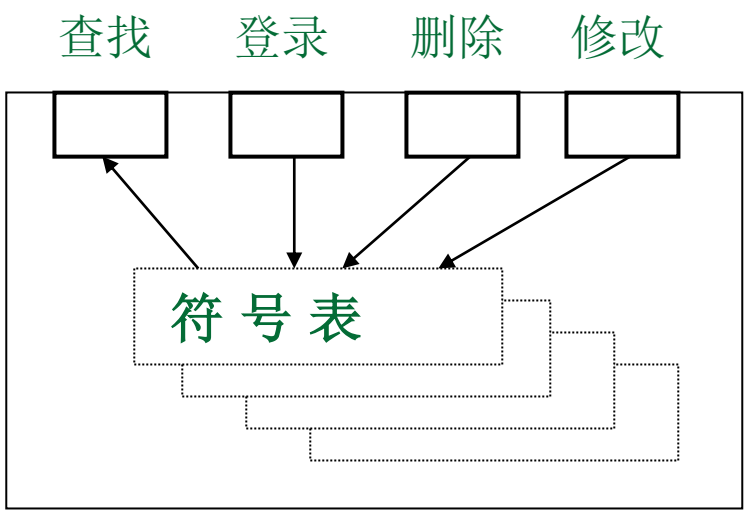


全部任务纳入一个模块，得到一过程性模块

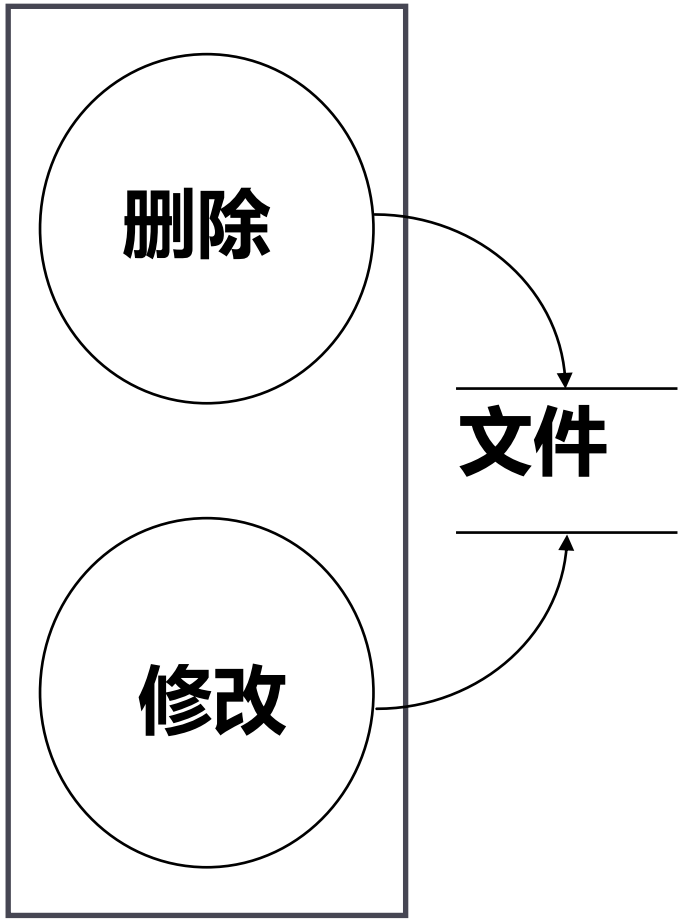
5. 通信内聚（Communicational cohesion）

- 出现在同一个模块里是由于操作或者生成同一个数据集。（引用共同的数据）

例如：从同一磁带上读取不相干的数据 —— 可能破坏独立性。

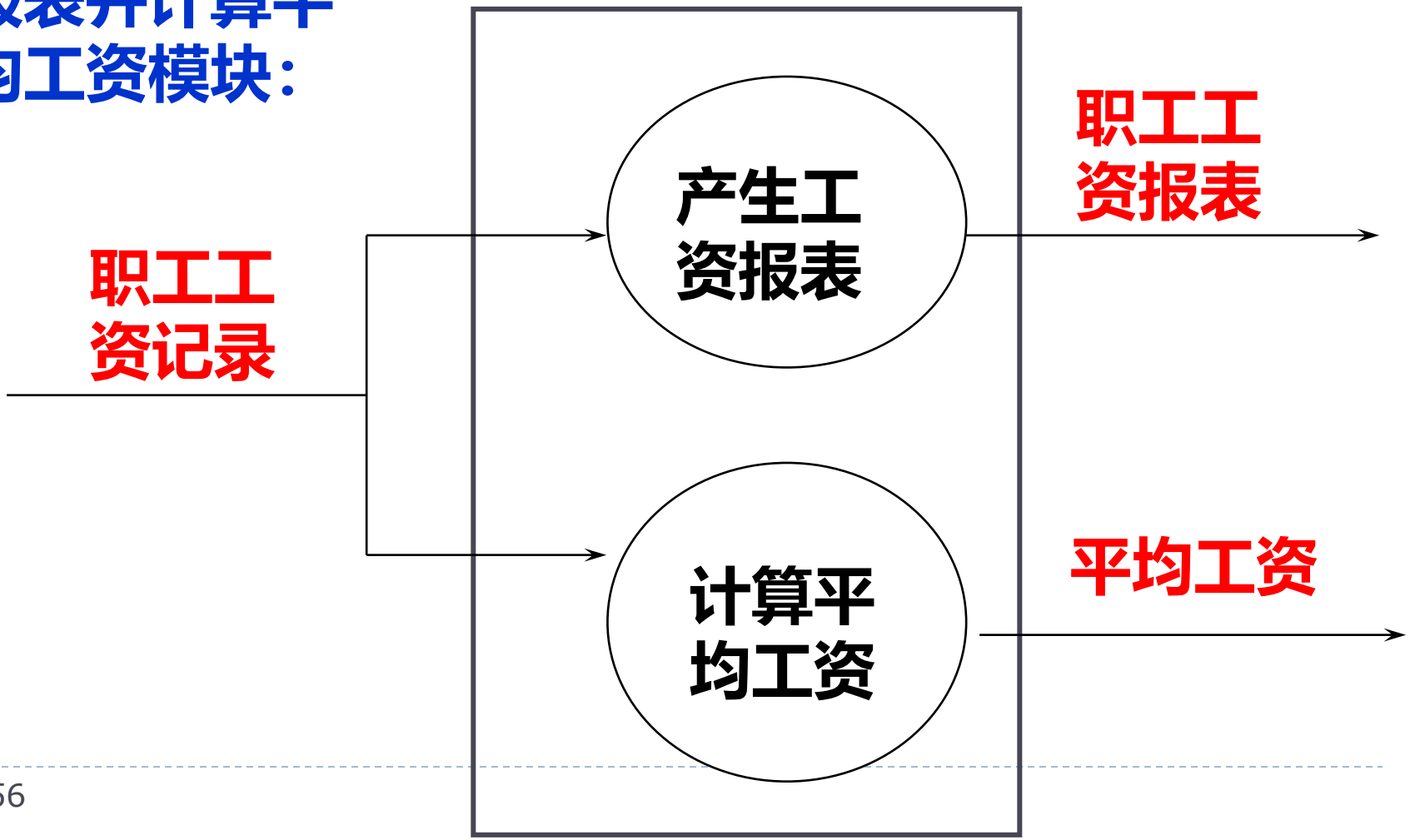


通信内聚例子1：操作或者生成同一个数据集



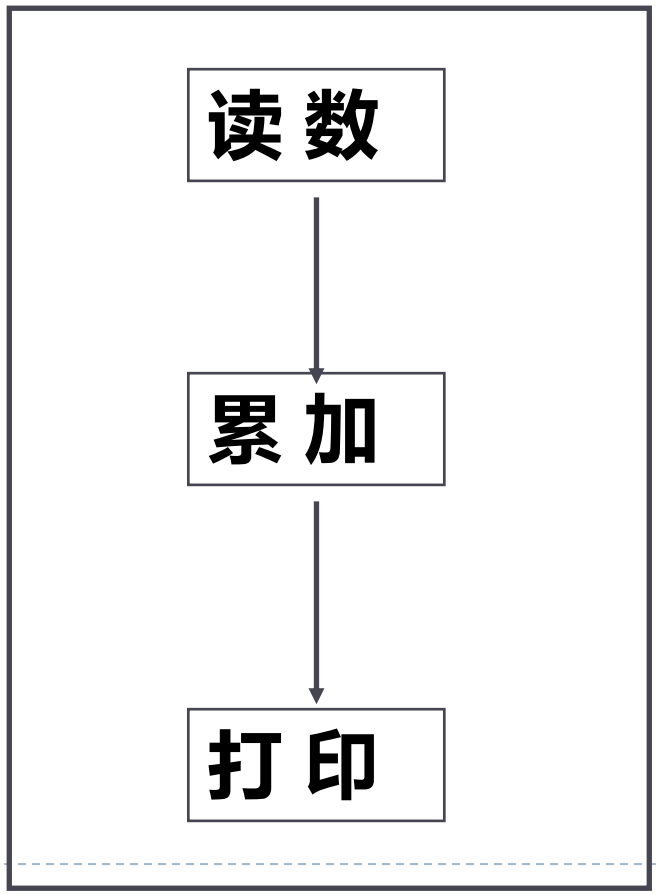
通信内聚例子2：操作或者生成同一个数据集

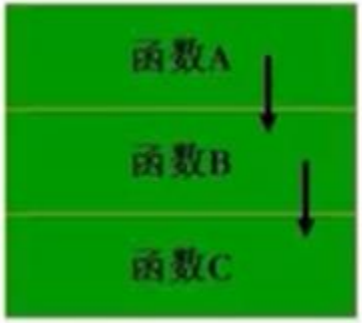
产生职工工资报表并计算平均工资模块：



6. 顺序内聚(Sequential cohesion)

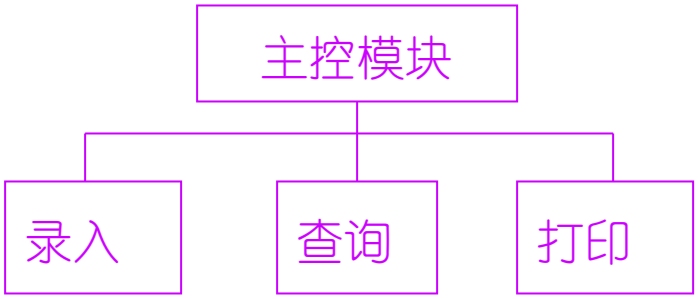
- 一个处理元素的输出数据作为下一个处理元素的输入数据。





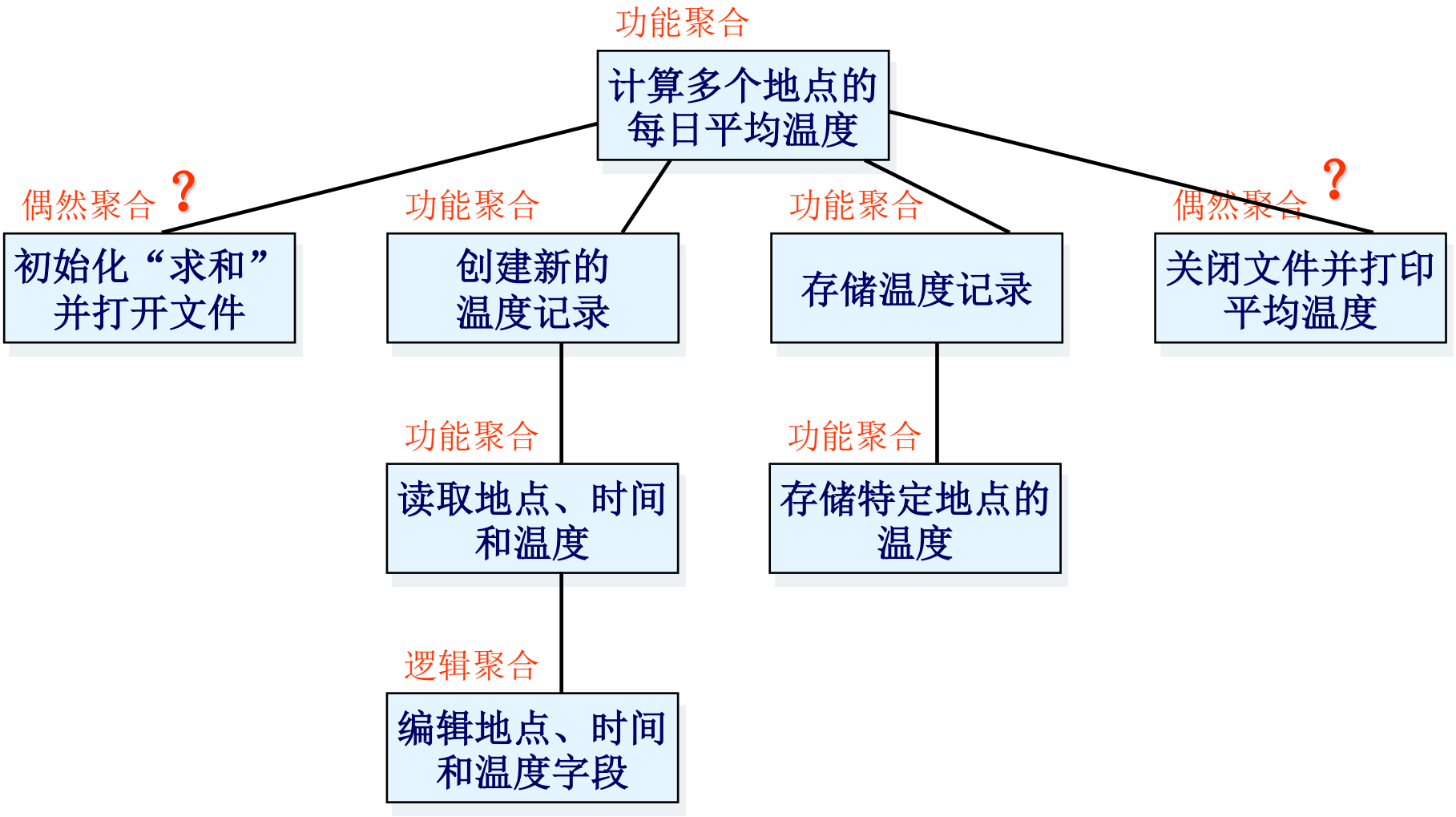
7. 功能内聚（Functional cohesion）

- 模块内所有处理元素属于一个整体，完成一个单一的功能。



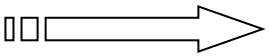
- 原则：在实际工作中，确定内聚的精确级别是不必要的，重要的是力争高内聚和识别低内聚，可以使得设计的软件具有较高的功能独立性。

内聚示例


功能聚合
计算多个地点的每日平均温度
偶然聚合
?
初始化“求和”并打开文件
功能聚合
创建新的温度记录
功能聚合
存储温度记录
偶然聚合
?
关闭文件并打印平均温度
功能聚合
读取地点、时间和温度
功能聚合
存储特定地点的温度
逻辑聚合
编辑地点、时间和温度字段

模块设计的步骤

建立初始结构图



进行改进

功能型模块的标准

- 用一个短句简略地描述这个模块“做什么”。
- 只完成一件事。

块间联系和块内联系总结

设计总则

- 使每个模块执行一个功能
- 模块间传送数据型参数
- 模块间共用信息尽量少

模块的评价方法

- 可分解性：模块是可分解的
- 可组装性：模块是可组装的
- 可理解性：模块可以被独立理解
- 连续性：　某个模块的修改不会导致整个系统的修改
- 保护：　异常限制在模块内

- 不要过度模块化！
  每个模块的简单性将被集成的复杂性所掩盖。

5.3 启发式规则

(1) 改进软件结构提高模块独立性

(2) 深度、宽度、扇出和扇入应适中

(3) 模块的作用域应该在控制域之内

(4) 力争降低模块接口的复杂程度

(5) 设计单入口、单出口的模块

(6) 模块功能应该可以预测

注：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。

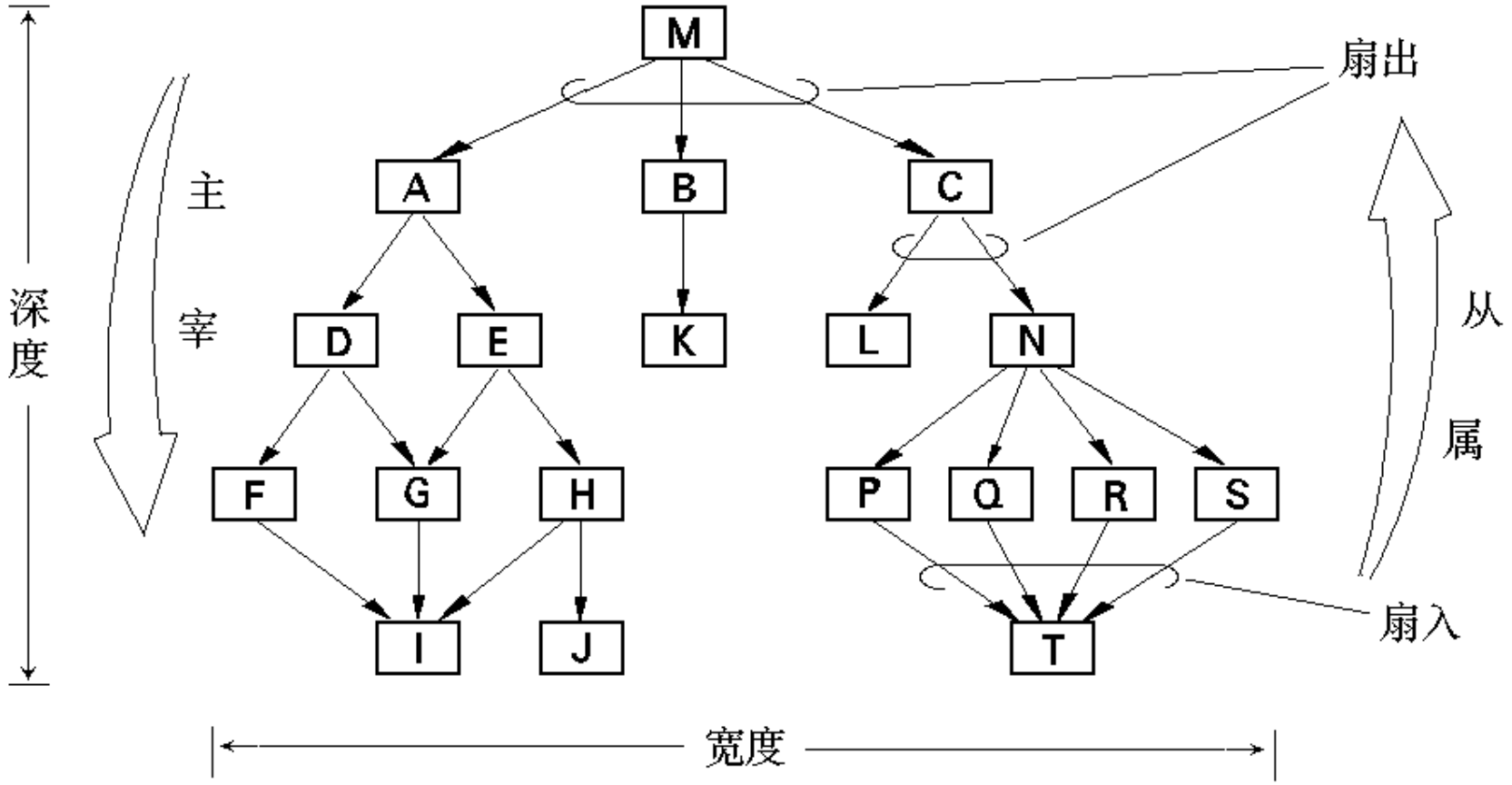
启发式规则1 – 模块独立性

1. 争取低耦合、高内聚（增加内聚 > 减少耦合）
2. 模块规模适中：过大分解不充分不易理解；太小则开销过大、接口复杂。注意分解后不应降低模块的独立性。
3. 适当控制 ——

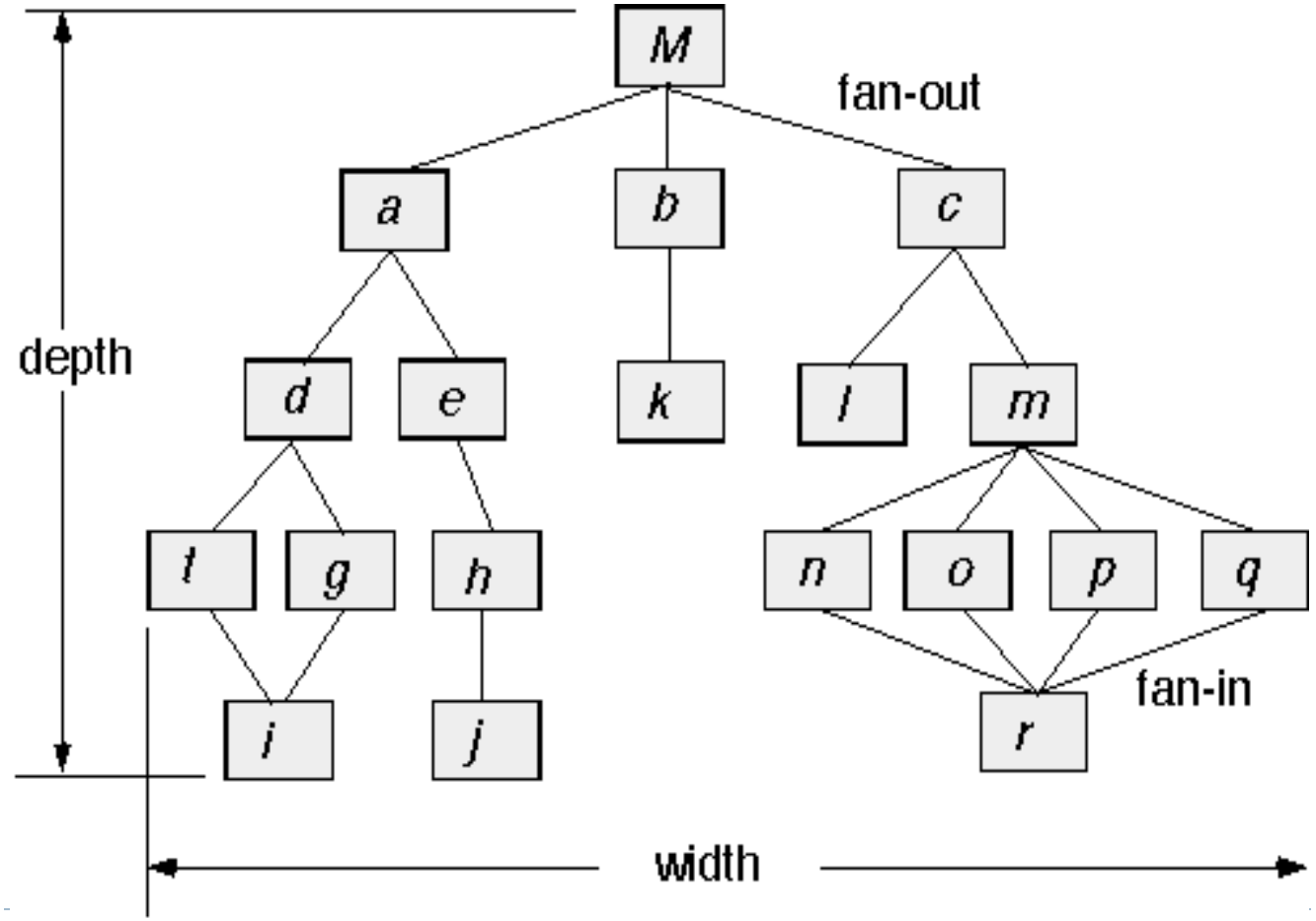
   深度 = 分层的层数。过大表示分工过细。

   宽度 = 同一层上模块数的最大值。过大表示系统复杂度大。

启发性规则2 – 深度和宽度

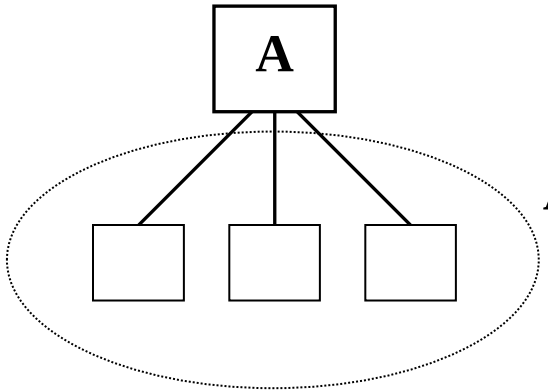


Call and Return Architecture

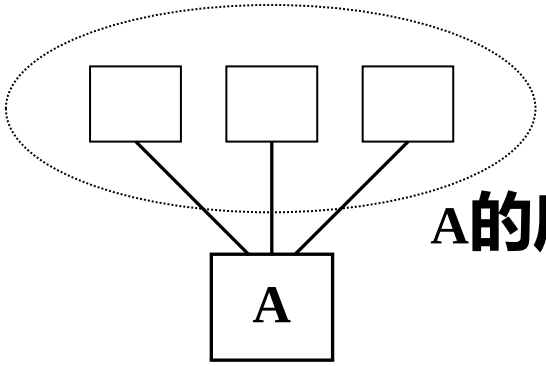


启发性规则3 – 扇出和扇入

♠ 扇出 = 一个模块直接调用\控制的模块数。 3 ≤ fan-out ≤ 9



A的扇出



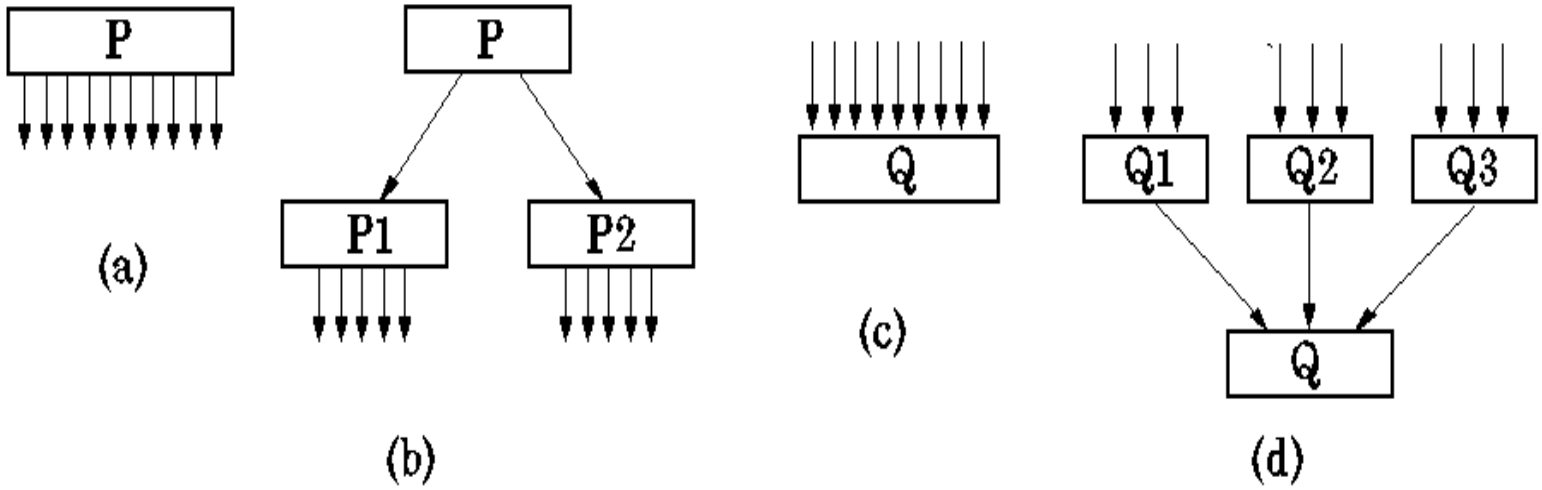
A的扇入

♠扇入 = 直接调用该模块的模块数

在不破坏独立性的前提下，fan-in 大的比较好。

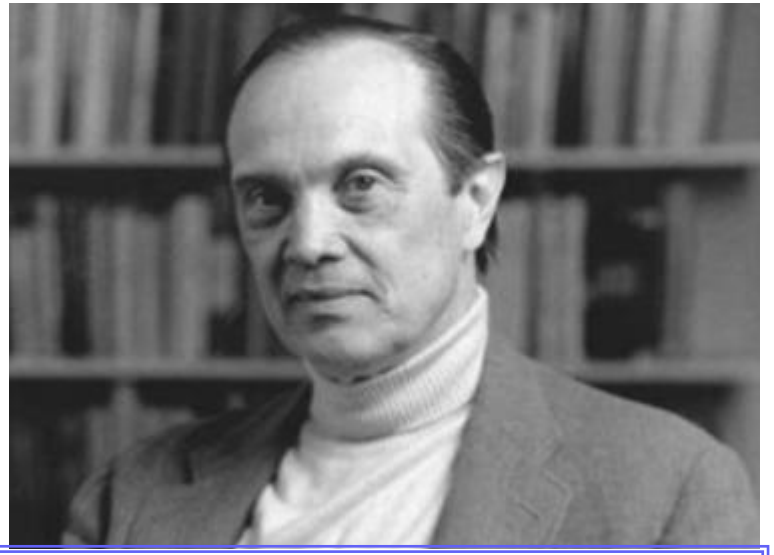
启发性规则4 – 扇出和扇入

- 尽可能减少高扇出结构，随着深度增大扇入。
  如果一个模块的扇出数过大，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控制模块。
- 一般来说，顶层扇出高，中间扇出少，低层高扇入。



扇出的控制：3或4，上限为5~9

扇入越大，说明复用性越好



奇妙的数字 **7±2**，人类信息处理能力的限度

G.A. Miller

Magical Number Seven,Plus or Minus Two,Some Limits on Our Capacity for Processing Information

The Psychological Review,1956

瓮型结构原则

好的软件系统具有两头小、中间大的结构。

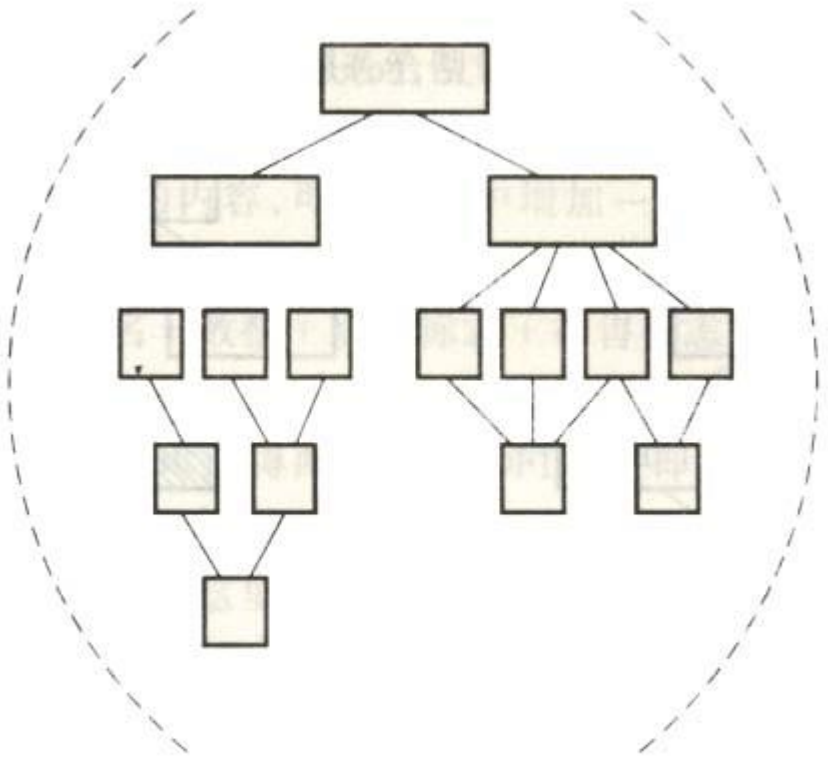


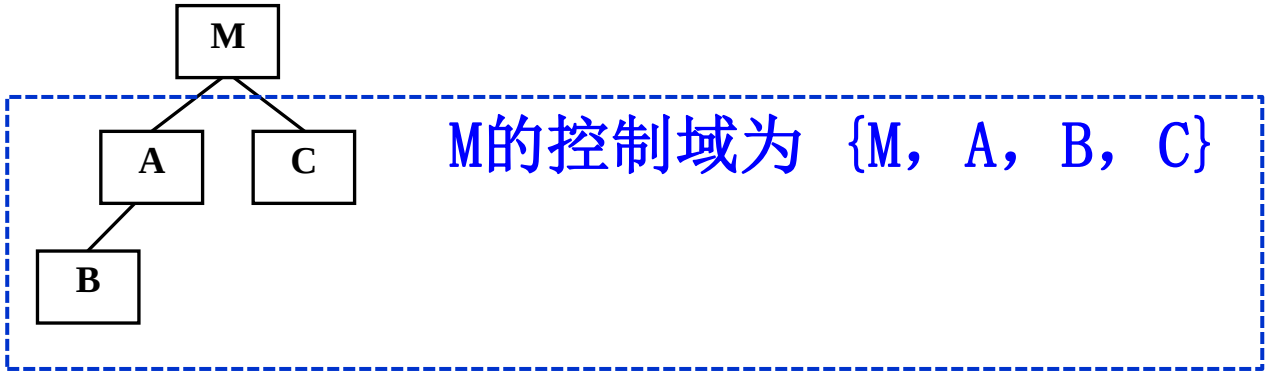
图 5.24　瓮形结构

启发性规则5 – 模块的作用域

模块的作用范围保持在该模块的控制范围内：

- 控制域指该模块本身以及所有直接或间接从属于它的模块。
- 作用域是指该模块中一个判断所影响的所有其它模块；

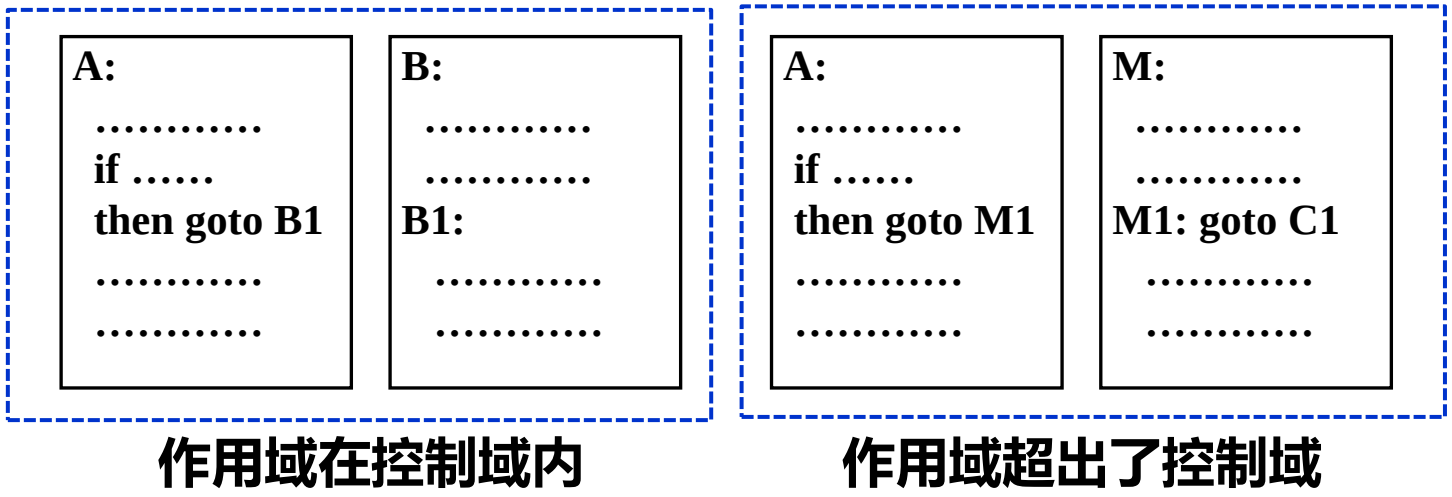
◆ 控制域



◆ 作用域：M中的一个判定所影响的模块。

启发性规则6 – 模块的作用域

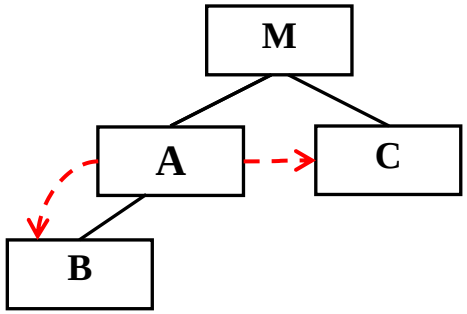
例：



上右例中，A的作用超出了控制域。

改进方法之一，可以把A中的 if 移到M中；

改进方法之二，可以把C移到A下面。



启发性规则7

4. 降低接口的复杂程度：模块接口的复杂性是引起软件错误的一个主要原因。接口设计应该使得信息传递简单并且与模块的功能一致。

5. 单出单入，避免内容耦合，易于理解和维护。

6. 模块功能可预测 —— 相同输入必产生相同输出。反例：模块中使用全局变量或静态变量，则可能导致不可预测。

5.4 描绘软件结构的图形工具

- **层次图和HIPO图**

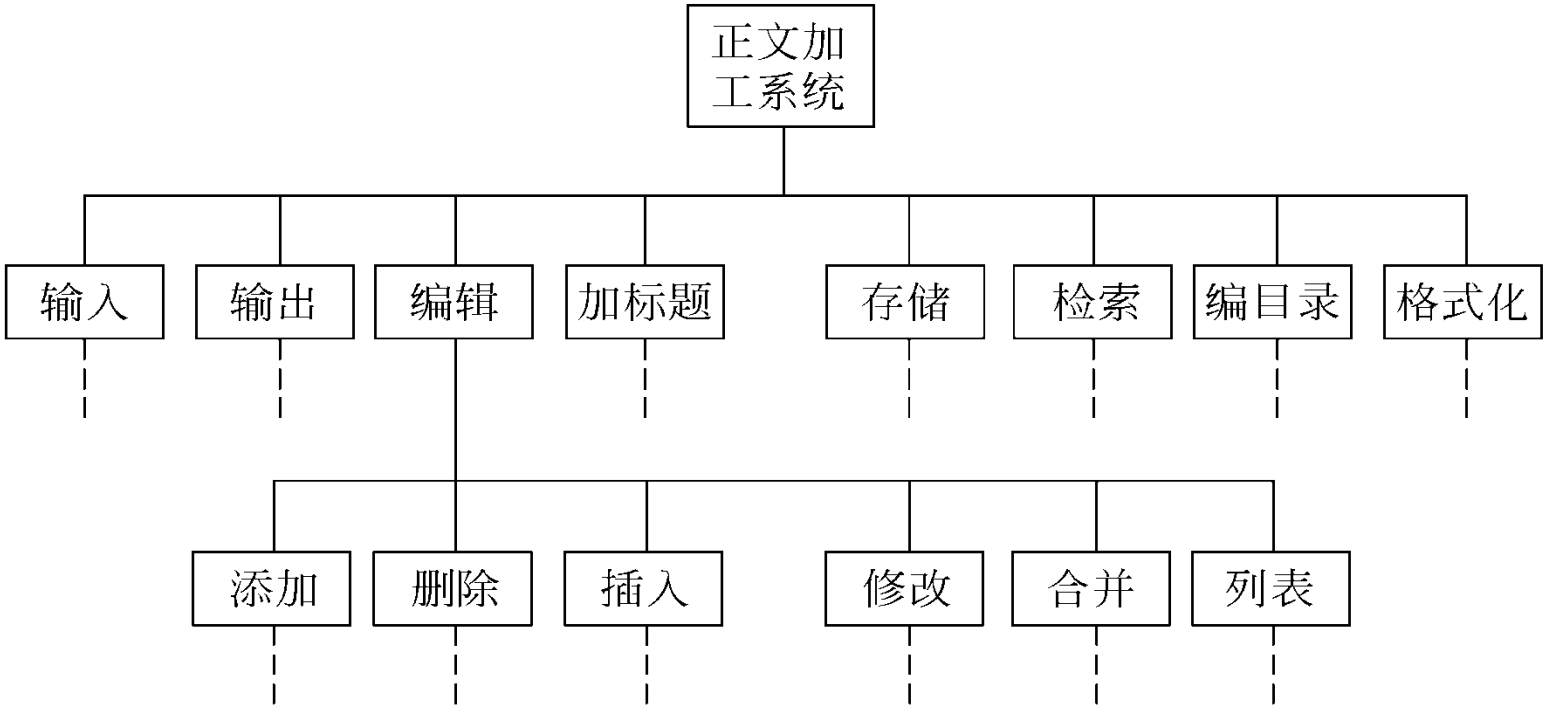


图5.3 正文加工系统的层次图

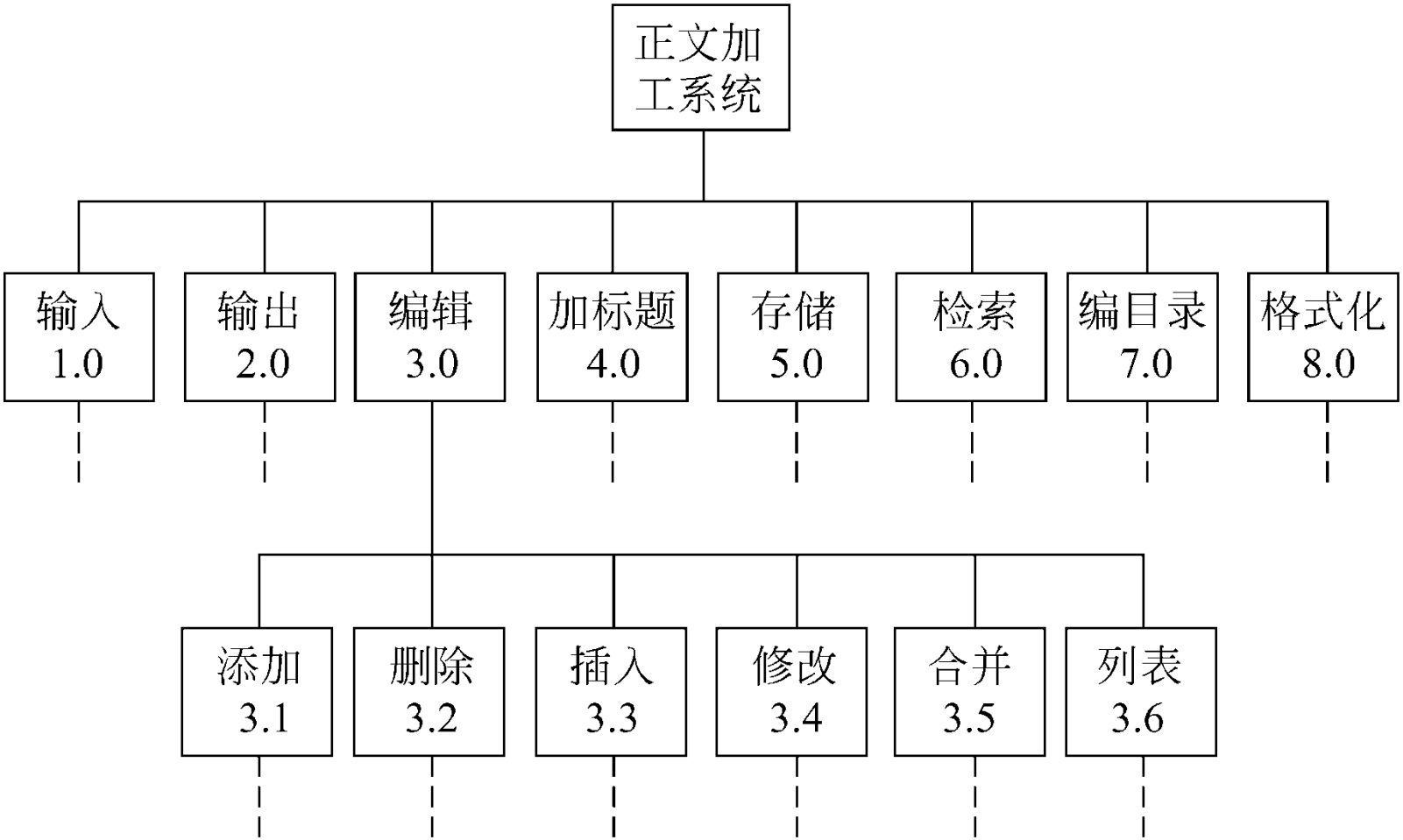
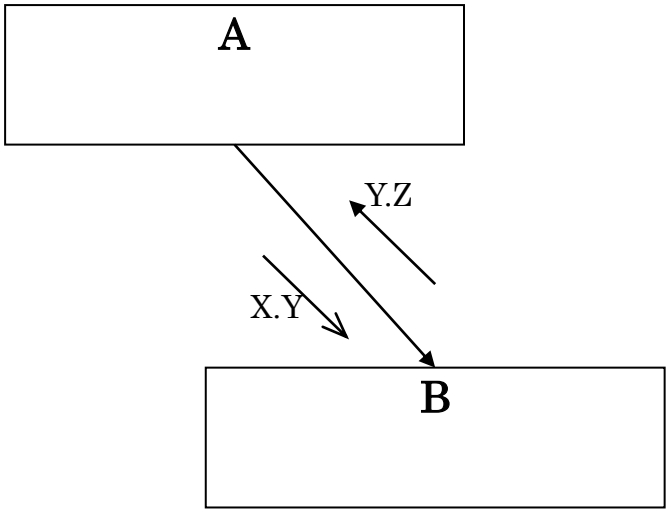


图5.4 带编号的层次图(H图)

结构图



← 调用模块/调用者

← 被调模块/下层模块

注意点

- **一个模块在结构图中只能出现一次。**
  - 一般习惯：输入模块在左，计算模块在中间，输出模块在右。
  - 结构图不同于流程图（框图）。

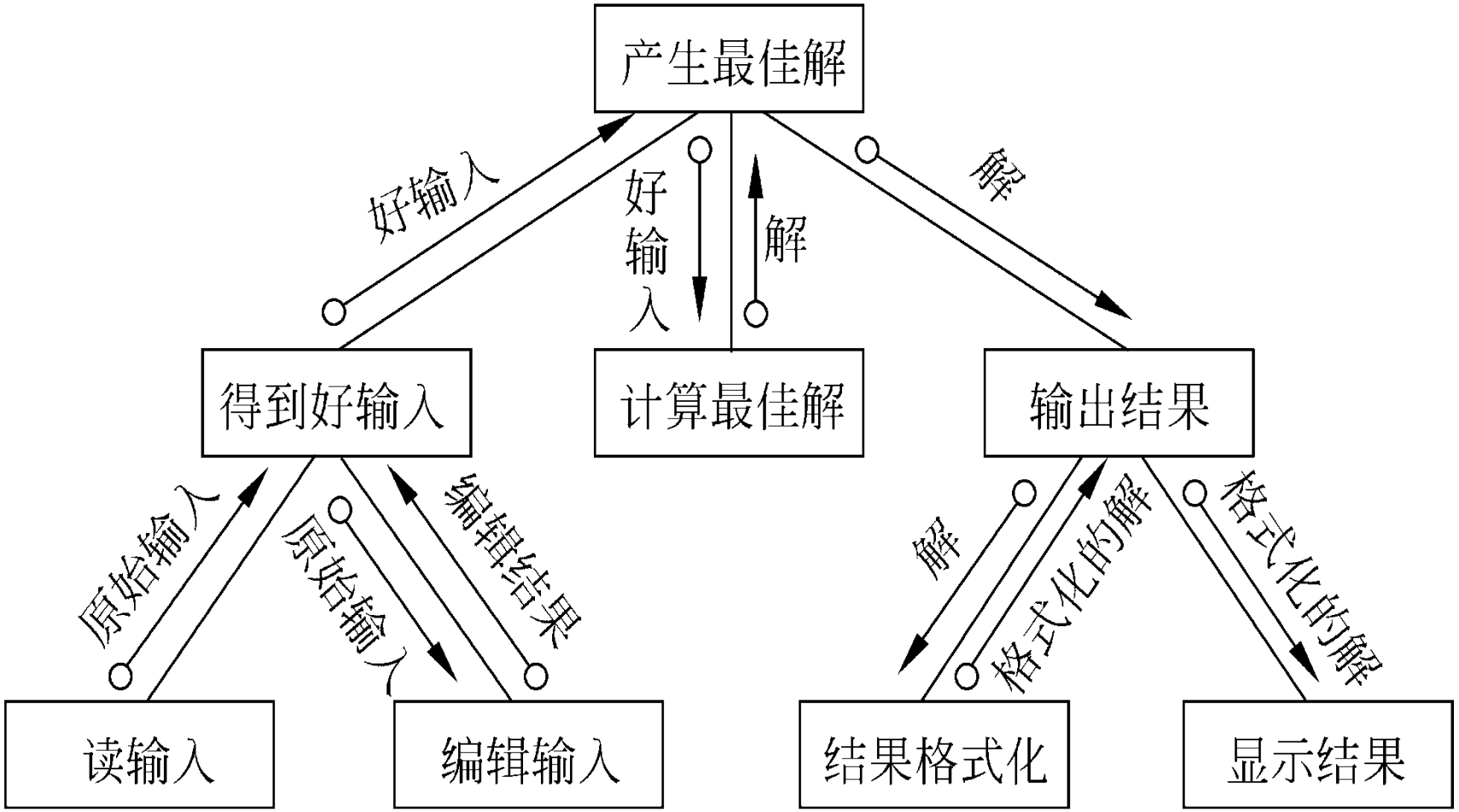


图5.5 结构图的例子——产生最佳解的一般结构

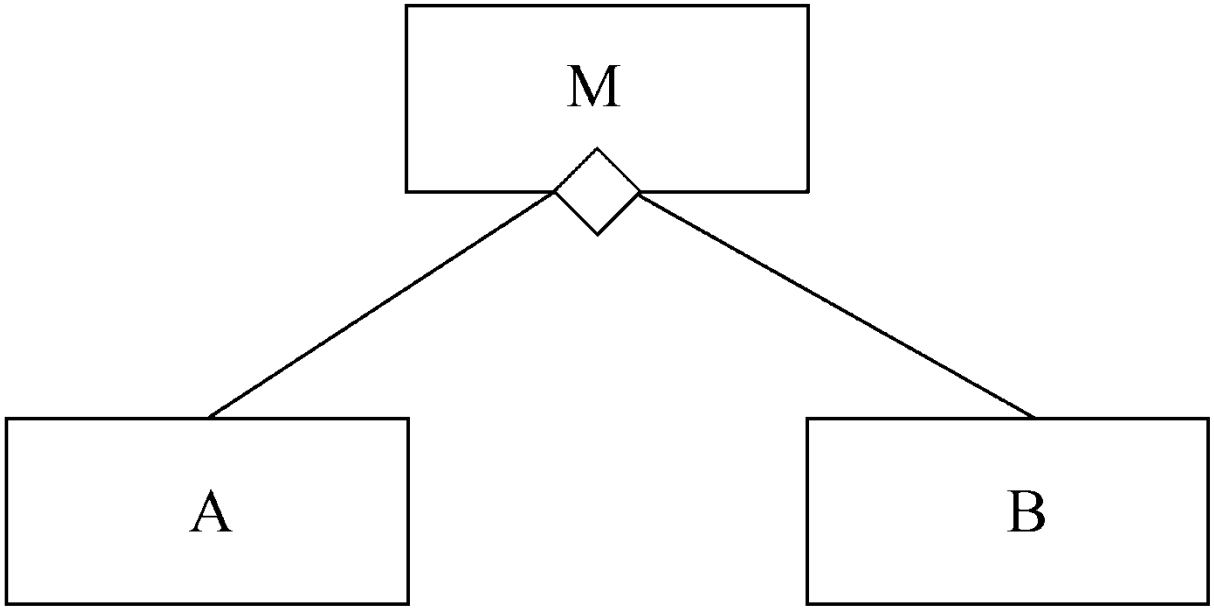


图5.6 判定为真时调用A，为假时调用B

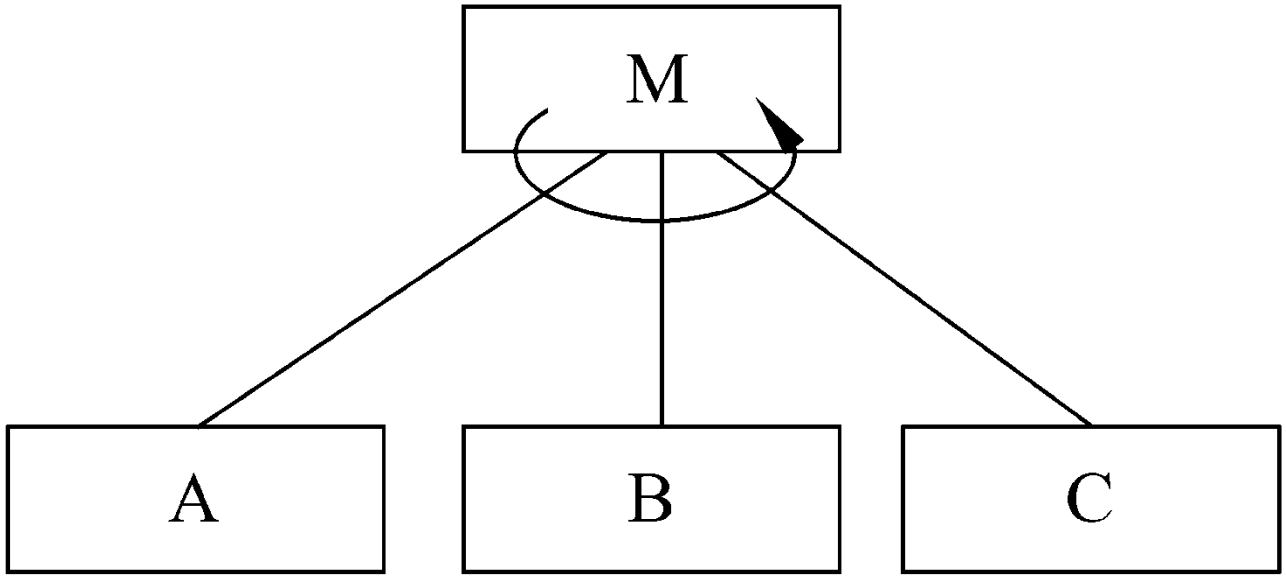


图5.7 模块M循环调用模块A、B、C

5.5 设计的方法　一

面向数据流的设计方法

一、基本概念

1．变换流（Transform flow）

根据基本系统模型，信息通常以“外部世界”的形式进入软件系统，经过处理以后再以“外部世界”的形式离开系统。信息沿输入通路进入系统，同时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，这种信息流就叫做变换流。

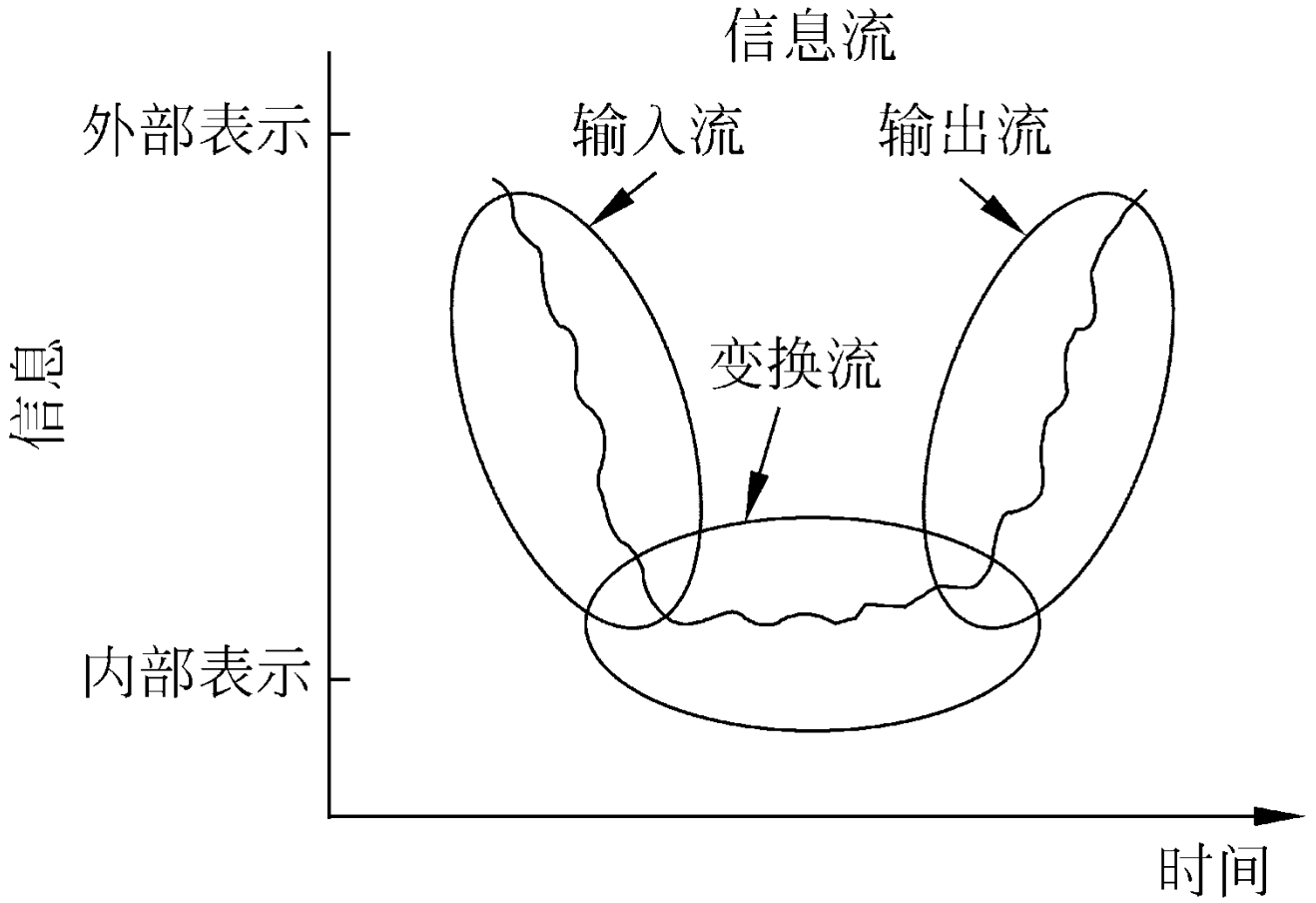
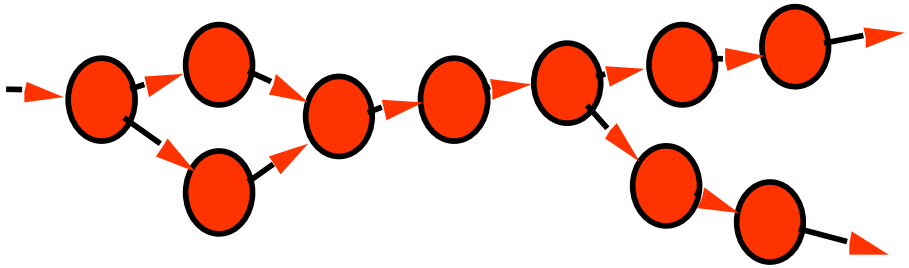
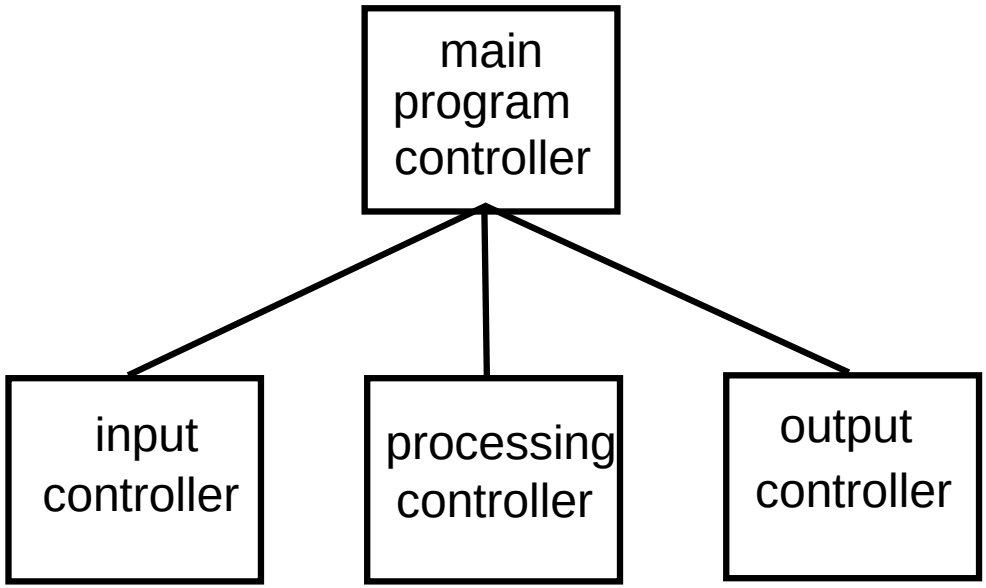


图5.8 变换流

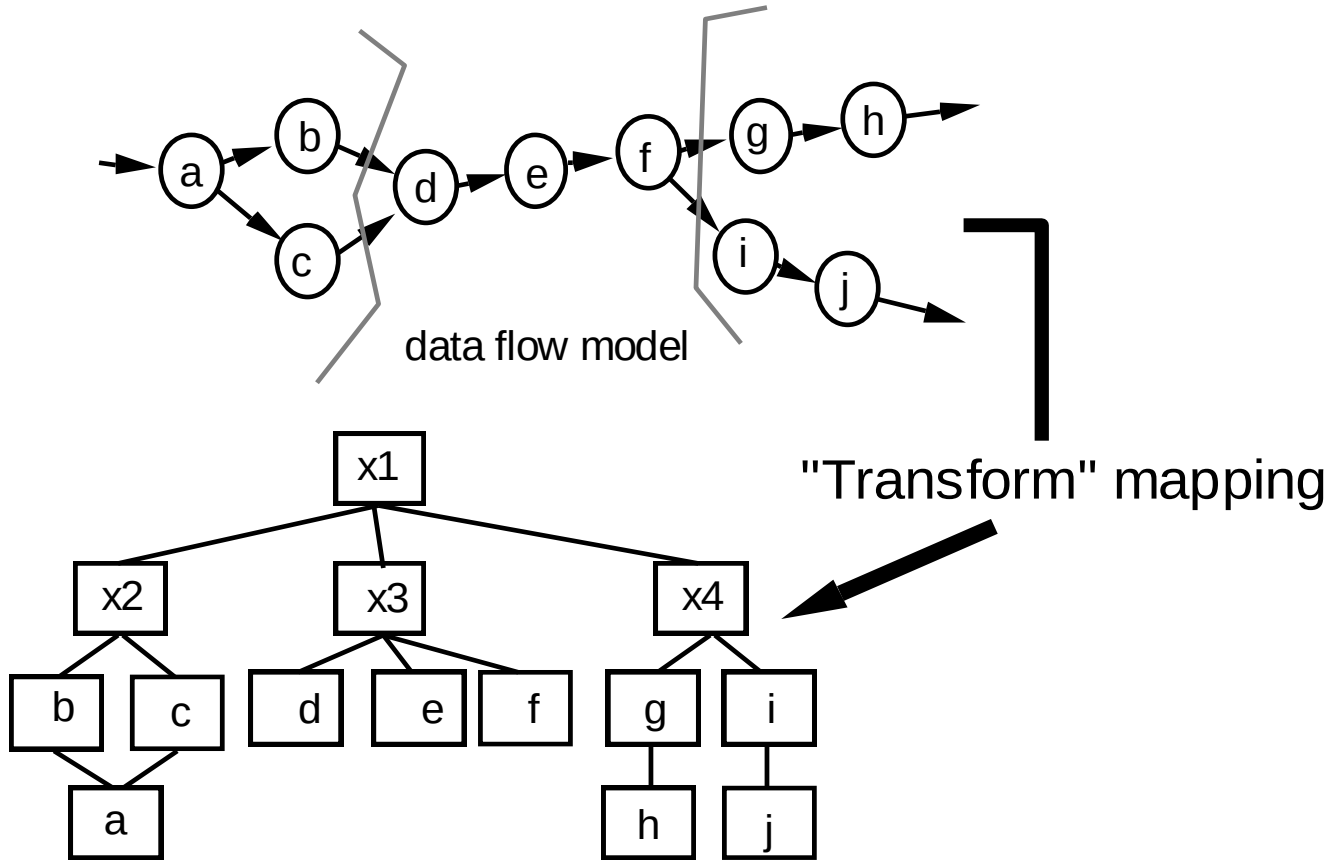
变换中心的First Level Factoring



Transform flow(变换中心)



Transform Mapping, example



2．事务流（Transaction Flow）

原则上所有信息流都可以归结为变换流，但是

当数据流图“以事务为中心”，也就是说，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。

这类系统的特征，是具有在多种事务中选择执行某种事务的能力。事务型结构由至少一条接受路径、一个事务中心和若干条动作路径组成。这类数据流应该划为一类特殊的数据流，称为事务流。

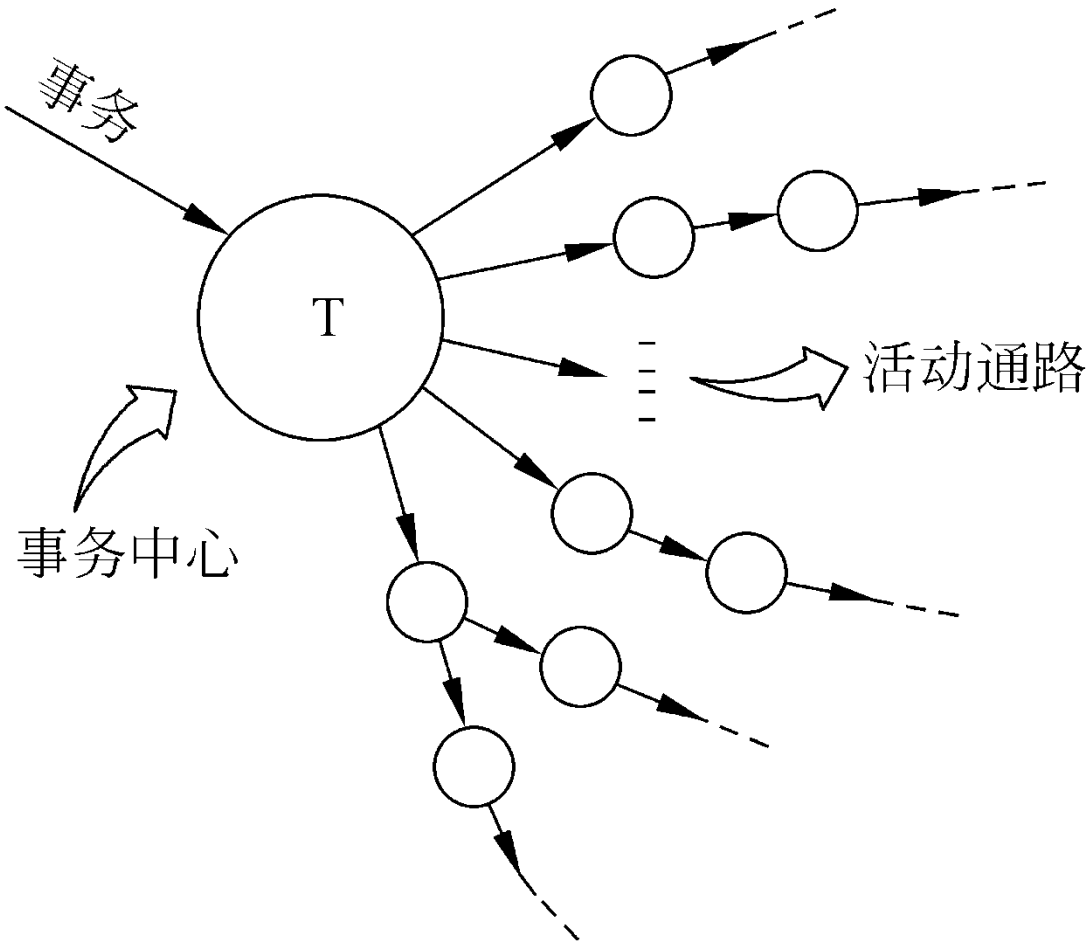
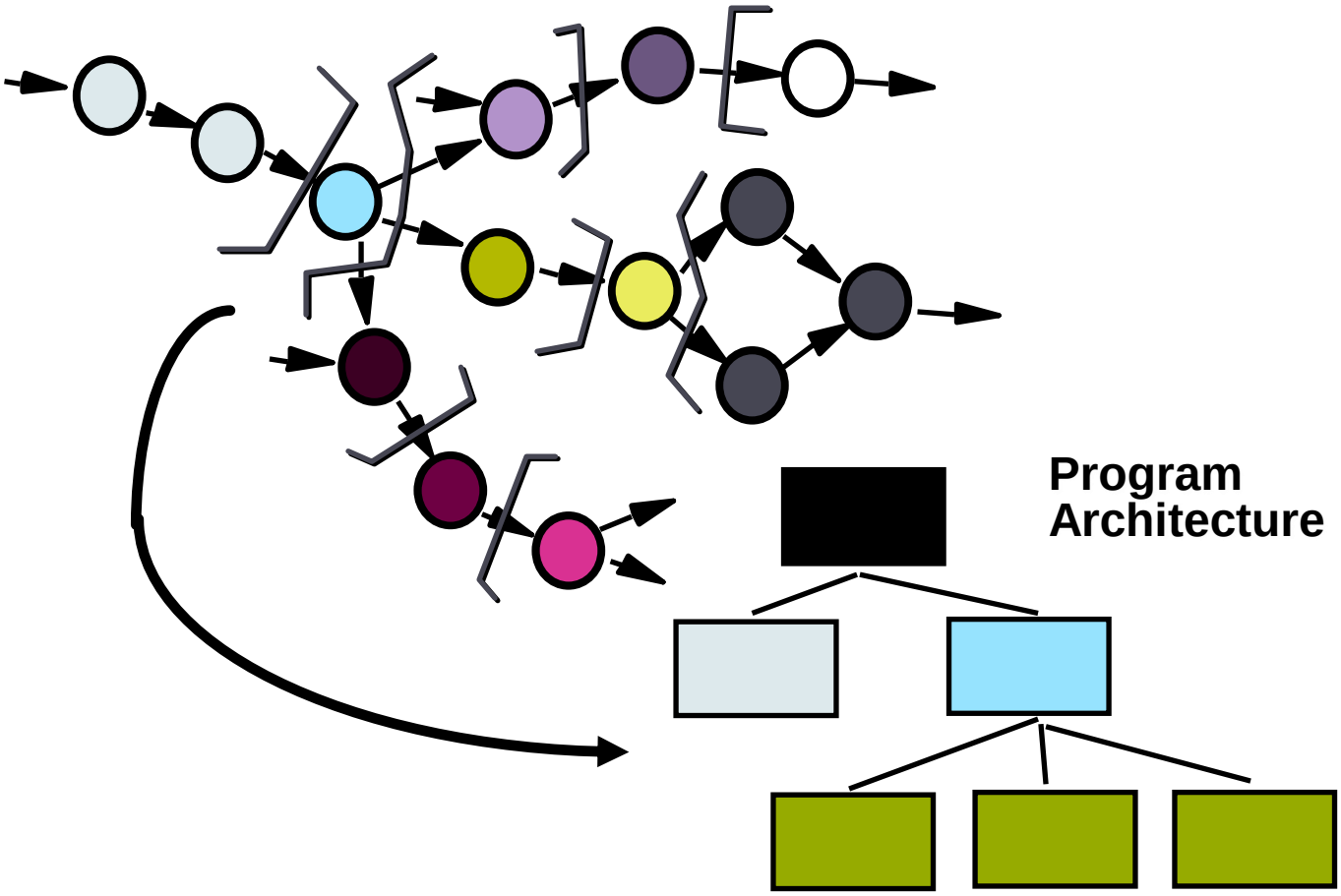
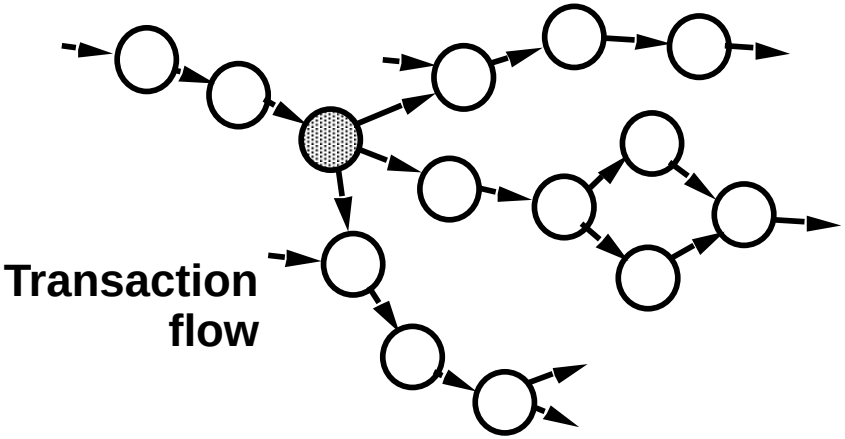


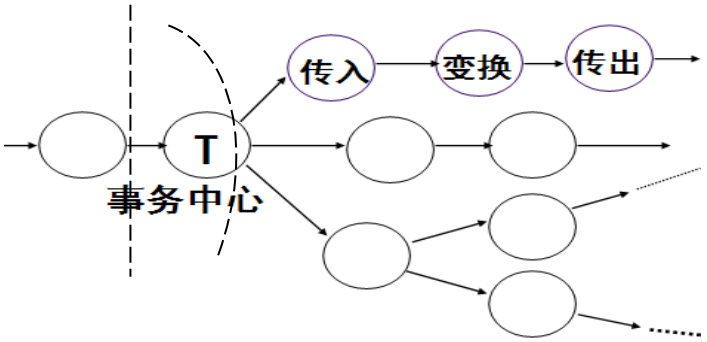
图5.9 事务流

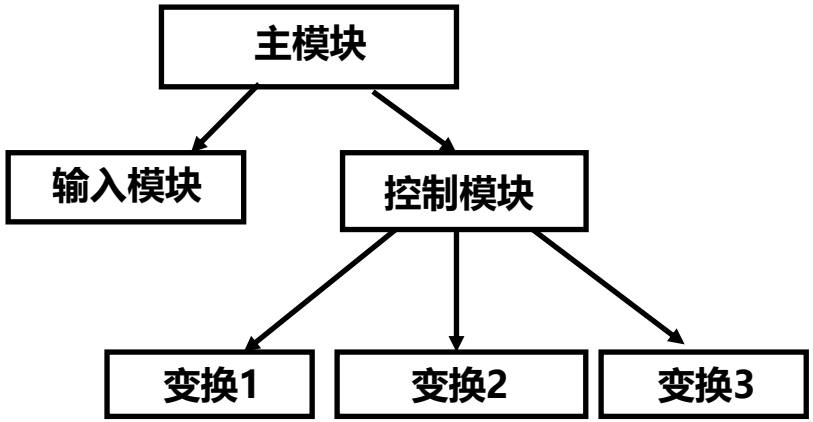
Deriving Program Architecture



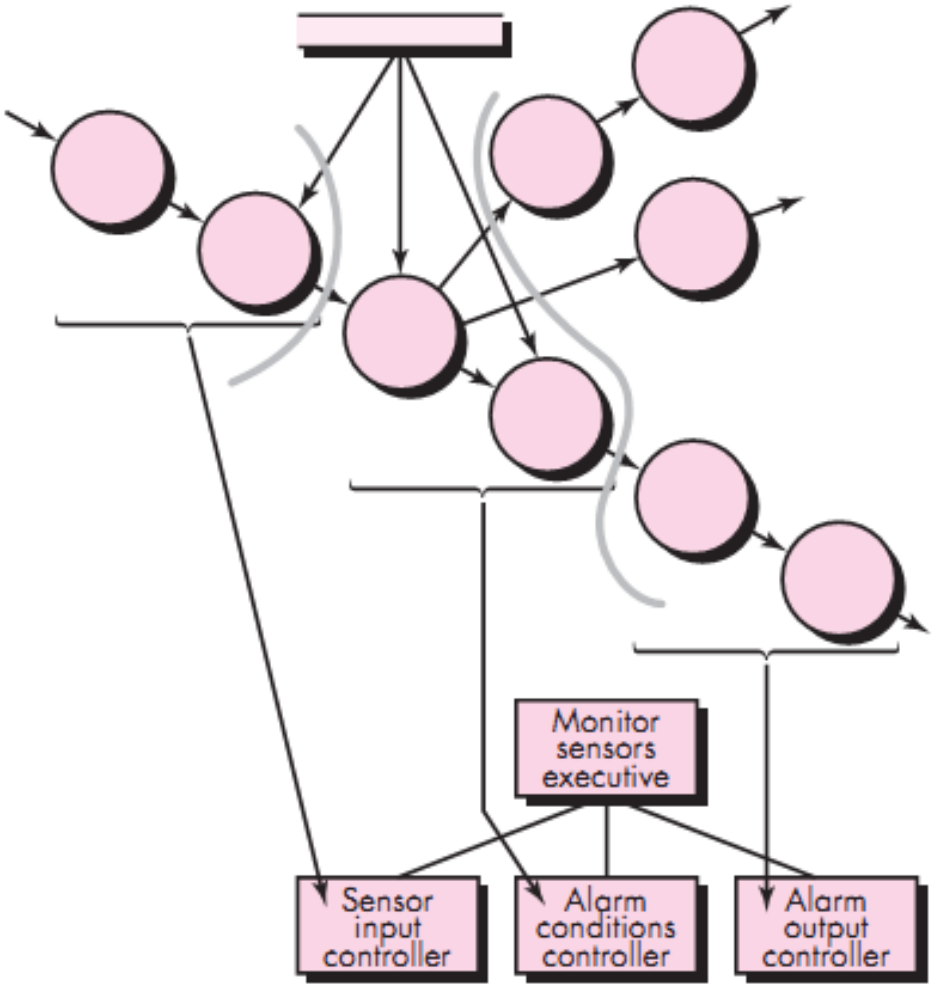
First Level Factoring-事务中心导出





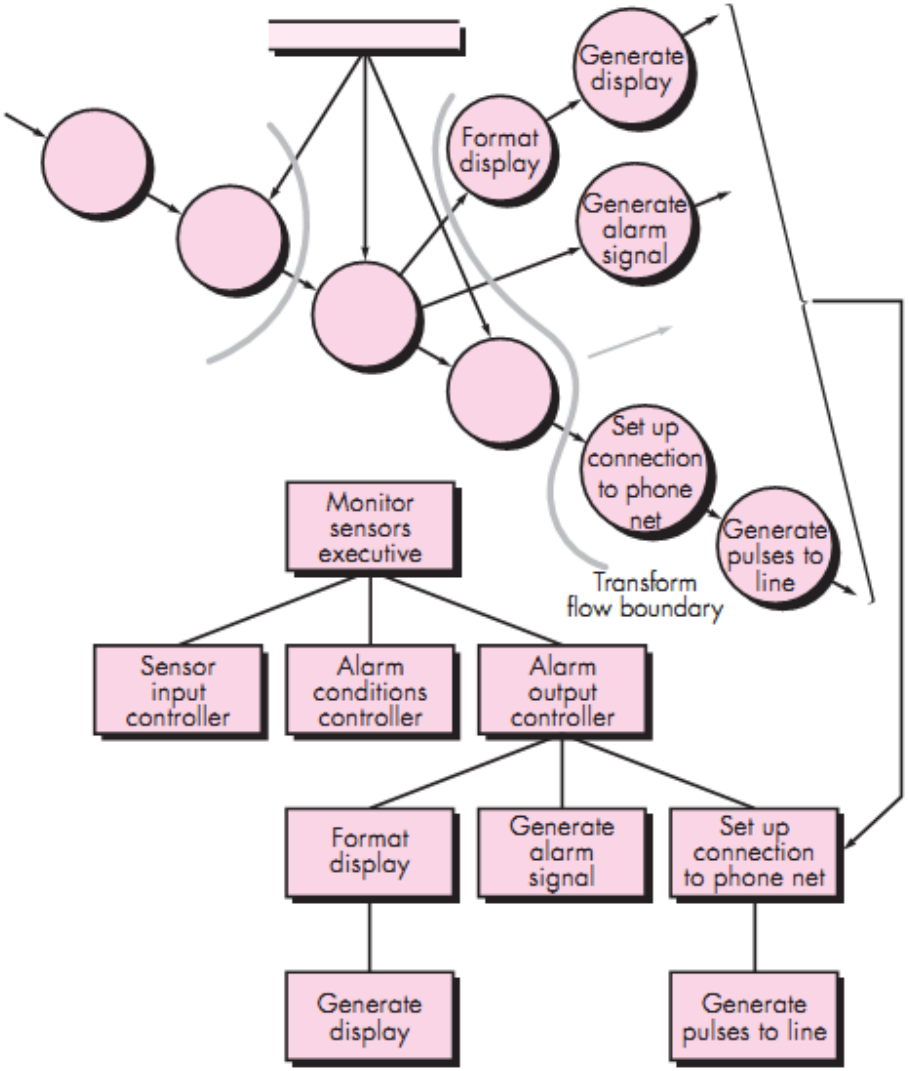


例子： First Level Factoring

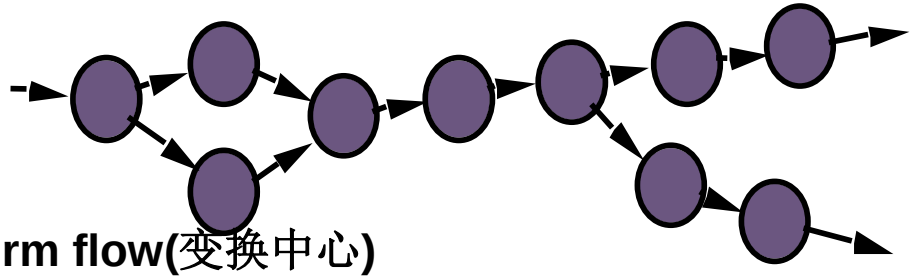


监控传感器

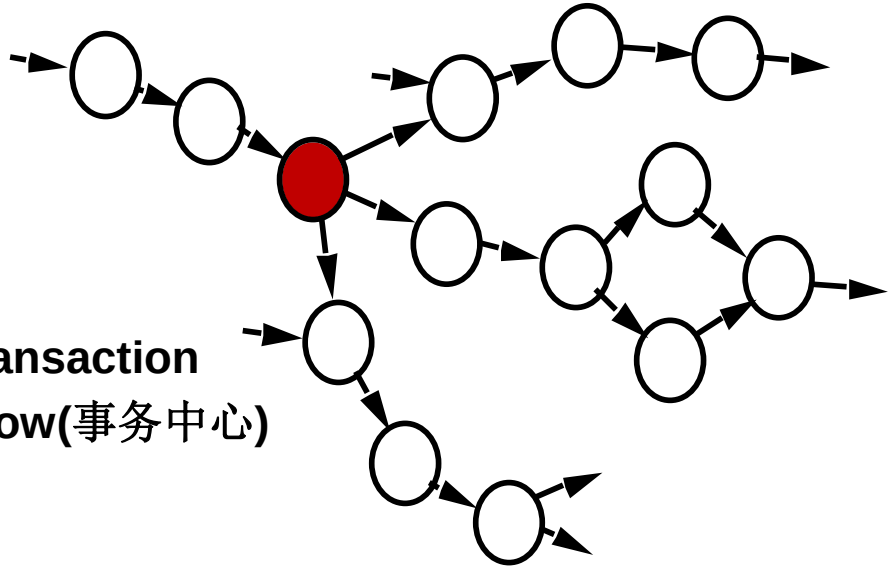
例子： Second Level Mapping



Flow Characteristics



Transform flow(变换中心)



Transaction Flow(事务中心)

二、SD方法概述

（1）首先研究、分析和审查数据流图，从软件的需求规格说明中弄清数据流加工的过程。

（2）然后根据数据流图决定问题的类型，即确定是变换型还是事务型。针对两种不同的类型分别进行分析处理。

（3）由数据流图推导出系统的初始结构图。

（4）利用一些试探性原则来改进系统的初始结构图，直到得到符合要求的结构图为止。

（5）修改和补充数据词典。

（6）制定测试计划。

三、SD方法的步骤

- 第一步是建立符合需求规格说明书要求的初始结构图（一般由数据流图导出初始结构图）；

- 第二步再用块间联系和块内联系等概念对初始结构图做进一步改进。

SD方法小结

- 根据问题的结构导出解答的结构，即根据数据流图导出结构图
- 为控制大型软件的复杂性，将系统分解，组织成适合于计算机实现的层次结构
- 用块间联系和块内联系作为评价软件结构质量的标准
- 给出一组设计技巧，如扇入，扇出，模块大小的掌握，作用范围和控制范围等
- 用结构图直观地描述软件结构，因此易于理解，分析和复查

概要设计的其他工作

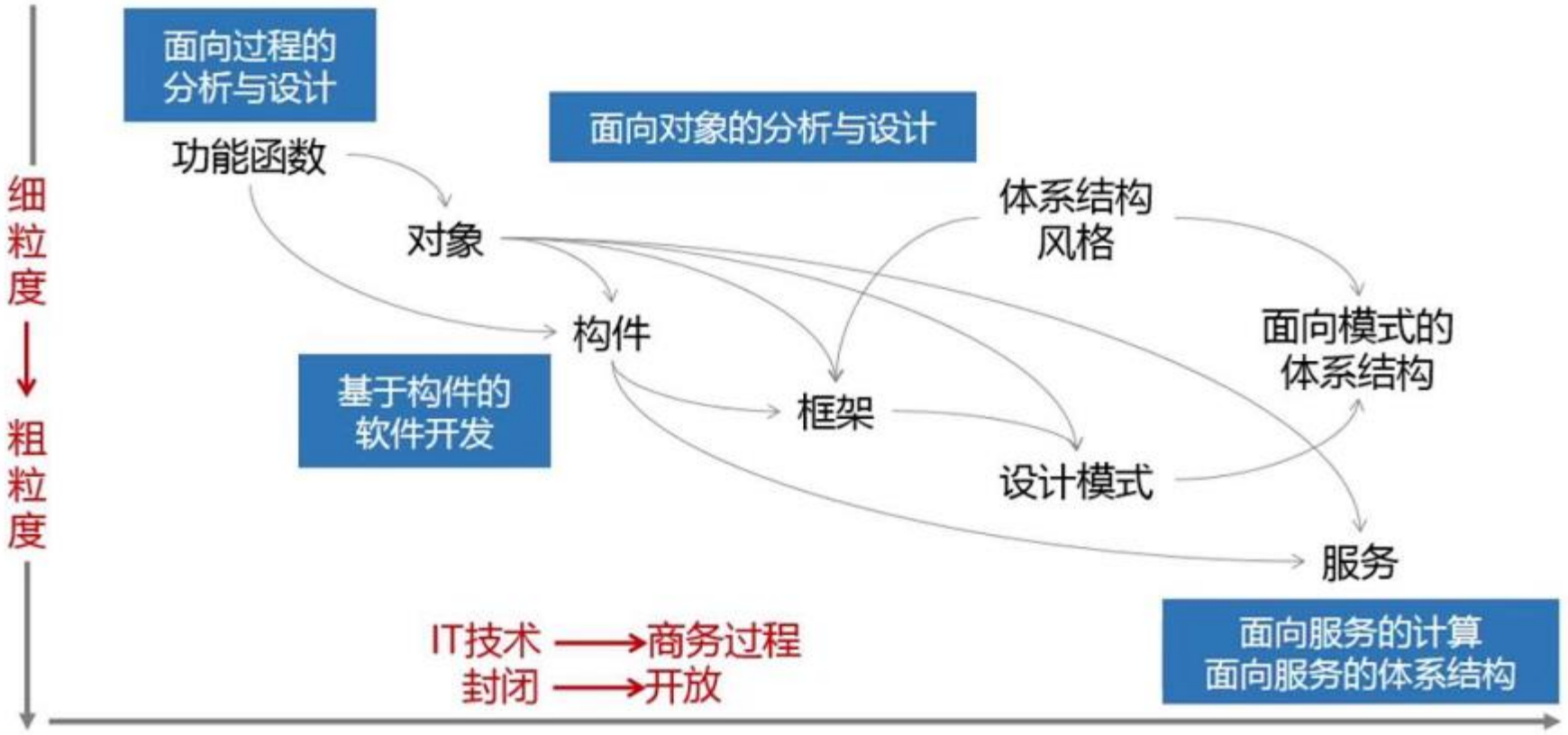
- **设计文档**
  - ✓ 结构图
  - ✓ 每个模块的描述，
  - ✓ 数据库文件结构和全程数据的描述
  - ✓ 需求/设计交叉表
  - ✓ 测试计划
- 概要设计复查

本章小结

- 模块的概念
- 内聚和耦合
- 扇入和扇出
- 作用域和控制域
- 设计的启发规则
- 面向数据流的设计方法(变化流、事务流)

以下为补充知识内容

软件体系结构的发展


面向过程的
分析与设计
功能函数
面向对象的分析与设计
对象
体系结构
风格
构件
面向模式的
体系结构
基于构件的
软件开发
框架
设计模式
服务
细粒度
粗粒度
IT技术 ⟶ 商务过程
封闭 ⟶ 开放
面向服务的计算
面向服务的体系结构

构件的基本概念

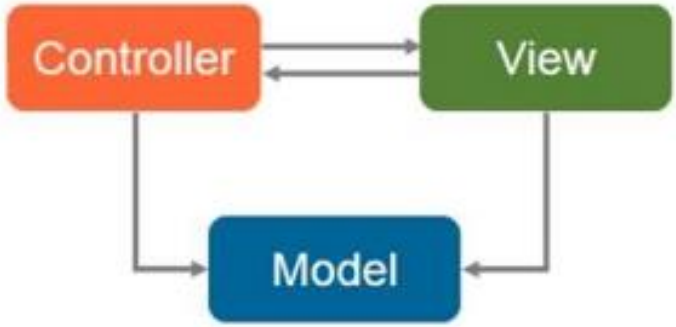
- 构件是为组装服务的!
- 软件构件是指可以独立生产、获取和部署的、可以被组装到一个功能性系统中去的可执行单元。
- 软构件是**标准的**、可以**互换的**、经过**装配**可随时使用的软件模块。
- 例如COM组件。

构件的基本形式

- 构件是可被用来构造其他软件的软件成分，基本形式可以是：
  - 被封装的对象类
  - 类树
  - 功能模块
  - 软件框架（framework）
  - 软件架构（或体系结构Architecture）
  - 文档
  - 分析件
  - 设计模式等

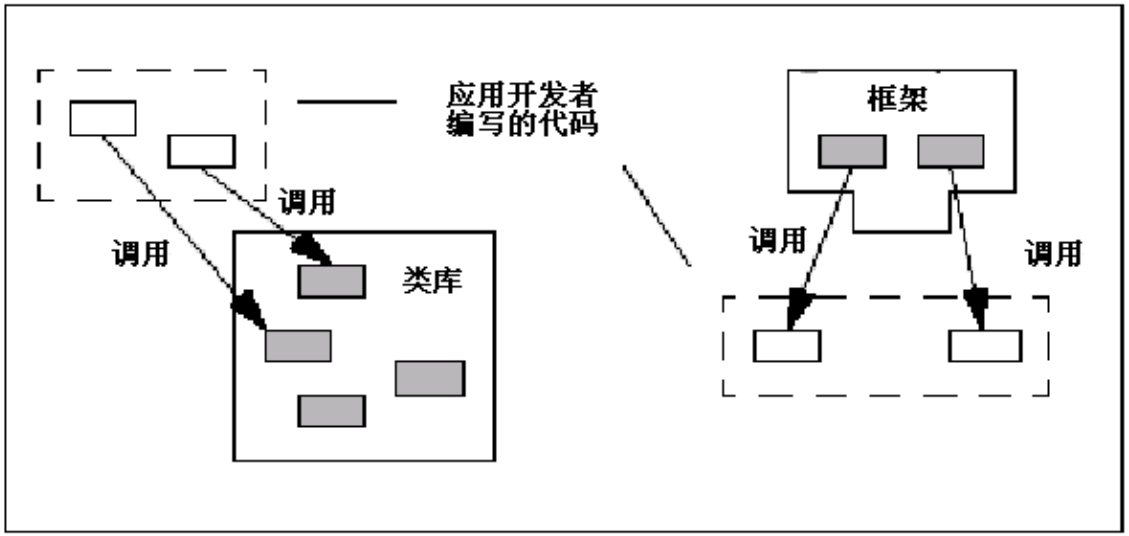
软件构架/软件体系结构（software Architecture ）

- 体系结构是一个软件系统从整体到部分的最高层次的划分。一般包括三个部分:构件，用于描述计算； 连接器，用于描述构件的连接部分；部署，将构件和连接器组成一个有机整体。
- 例如MVC架构、J2EE架构。



软件框架（Software Framework）

- 软件框架是指面向某领域（包括业务领域，如ERP，和计算领域，如GUI）的、可复用的“半成品”软件，它实现了该领域的共性部分，并提供一系列定义良好的可变点以保证灵活性和可扩展性。可以说，软件框架是领域分析结果的软件化，是领域内最终应用系统的模板。
- 例如：Spring框架、Struts2框架、Hibernate框架、MyBatis 框架。



Hollywood Principle: *Don’t call us. We’ll call you.*

软件构架（体系结构）与框架的区别

- 呈现形式不同。体系结构的呈现形式是一个设计规约，而框架则是物理实现。
- 目的不同。体系结构的首要目的大多是指导一个软件系统的实施与开发；而框架的首要目的是为复用．因此，一个框架可有其体系结构，用于指导该框架的开发，反之不然。

  - 体系结构的呈现形式是一个设计规约，而框架则是“半成品”的软件；
  - 体系结构的目的是指导软件系统的开发，而框架的目的是设计复用。

软件设计原则

- 设计原则是系统分解和模块设计的基本标准，应用这些设计原则可以使得代码更加灵活、易于维护和扩展。

扩展阅读：软件程序设计原则
https://blinkfox.github.io/2018/11/24/ruan-jian-she-ji/ruan-jian-cheng-xu-she-ji-yuan-ze/
或者：https://www.cnblogs.com/Lunais/p/11791011.html

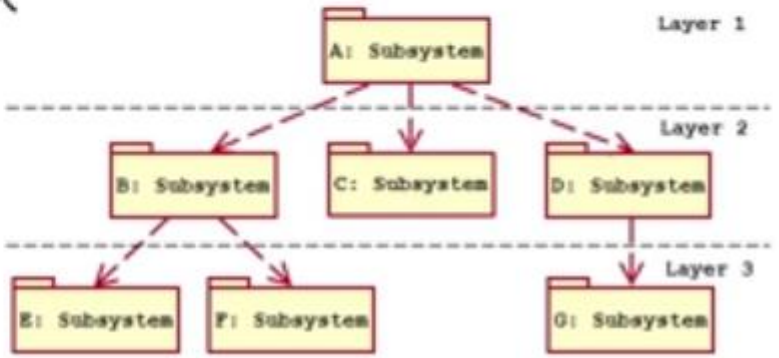
层次化的含义

分层（Layering）

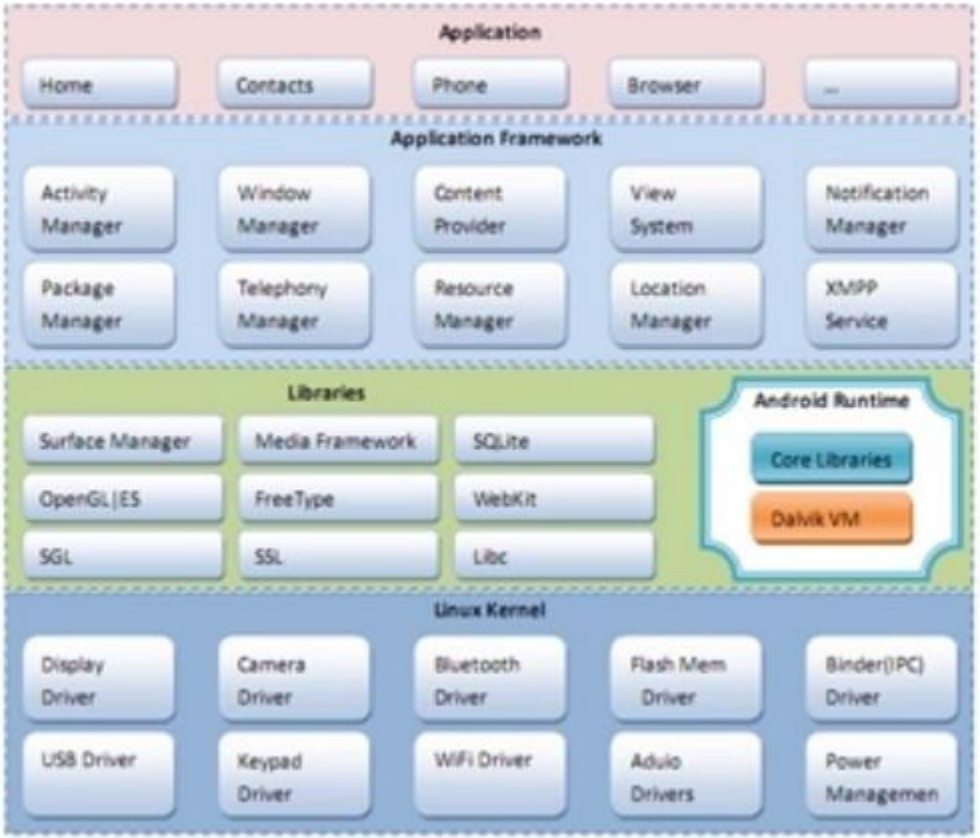
- 每一层可以访问下层，不能访问上层
- 封闭式结构：每一层只能访问与其相邻的下一层
- 开放式结构：每一层还可以访问下面更低的层次
- 层次数目不应超过 7±2 层

划分（Partitioning）

- 系统被分解成相互对等的若干模块单元
- 每个模块之间依赖较少，可以独立运行



实例：android操作系统层次结构



应用层：运行在虚拟机上的Java应用程序。

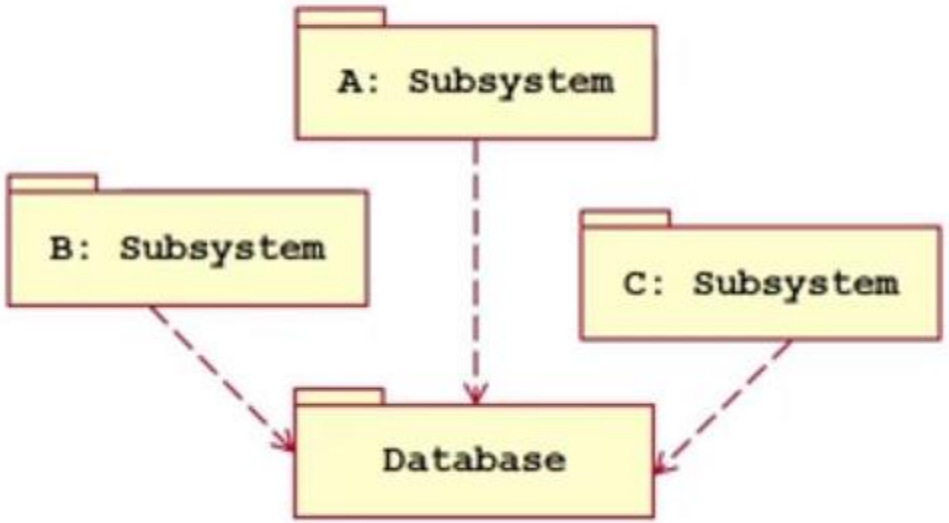
应用框架层：支持第三方开发者之间的交互，使其能够通过抽象方式访问所开发的应用程序需要的关键资源。

系统运行库层：为开发者和类似终端设备拥有者提供需要的核心功能。

Linux内核层：提供启动和管理硬件以及Android应用程序的最基本的软件。

为什么需要层次结构

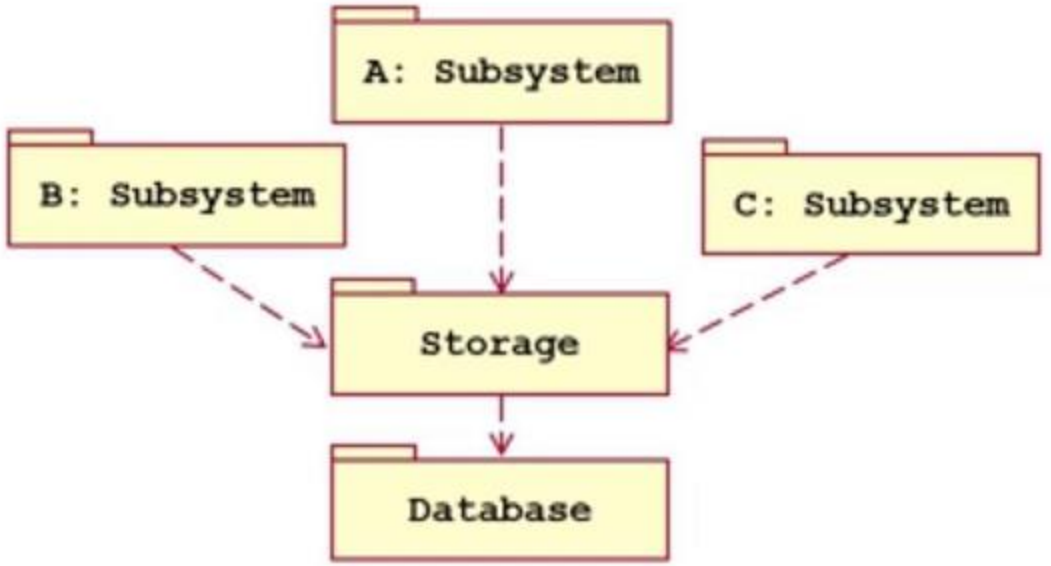
举例：在一个软件系统中，有三个子系统A、B、C都要访问一个关系数据库。



这个方案有什么问题？

为什么需要层次结构

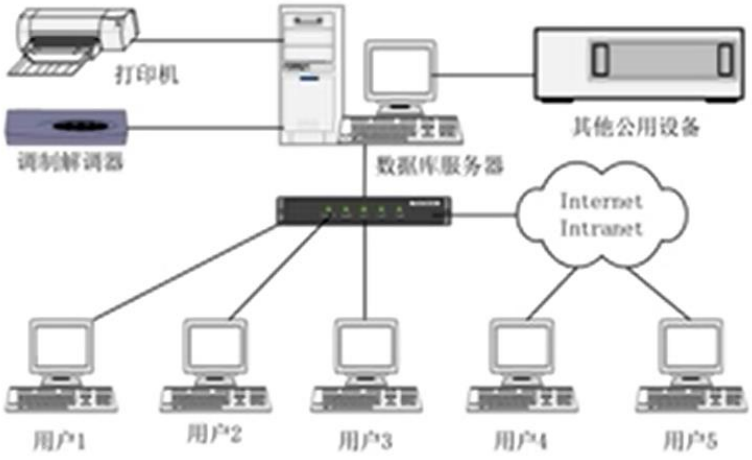
改进：在子系统和数据库之间增加一个存储子系统Storage，从而屏蔽了底层数据库的变化对上层子系统的影响。

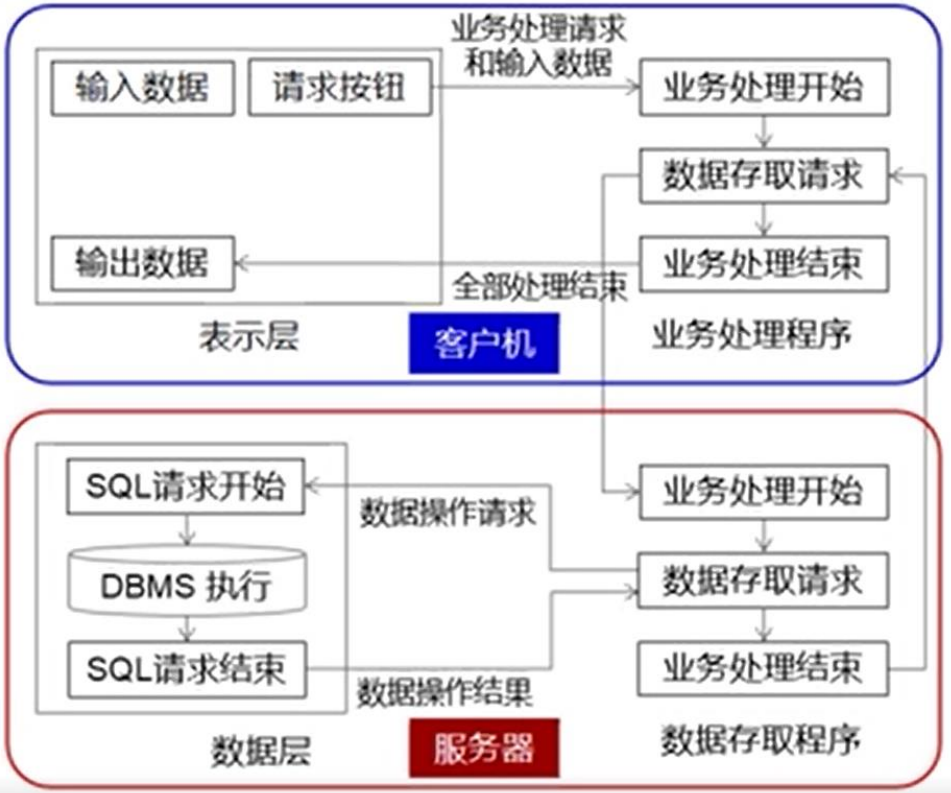


两层的C/S架构

胖客户端模型：

- 服务器只负责数据的管理
- 客户机实现应用逻辑和用户的交互

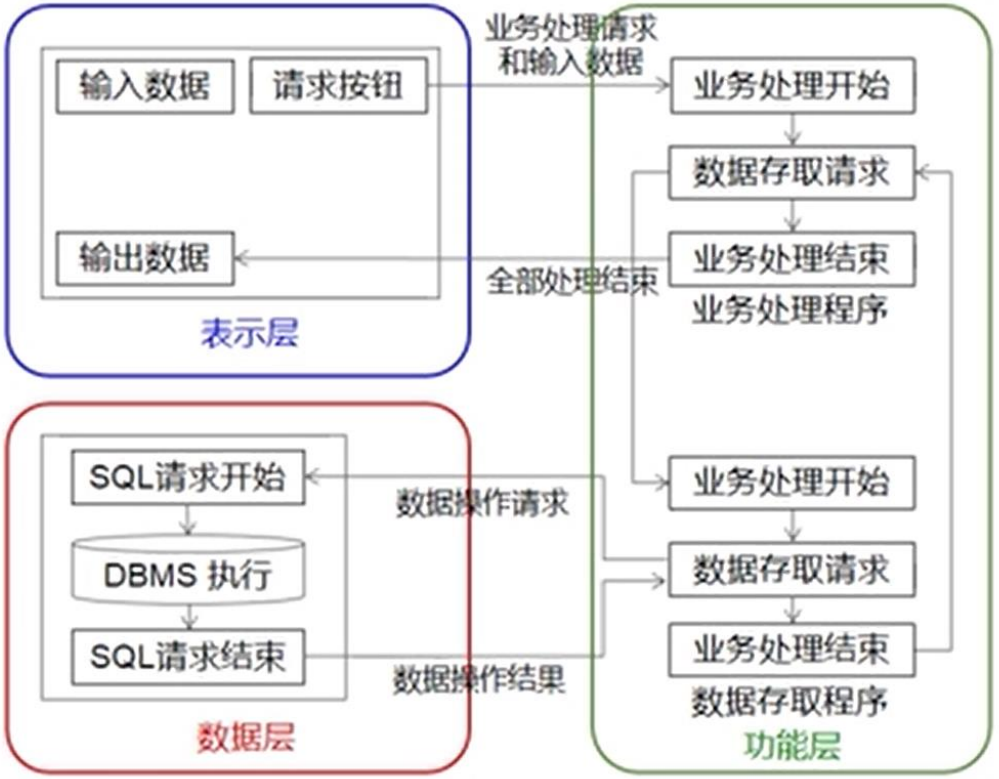




内容来源：
https://next.xuetangx.com/learn/THU08091000367/THU08091000367/1516221/video/1385976

三层的C/S架构

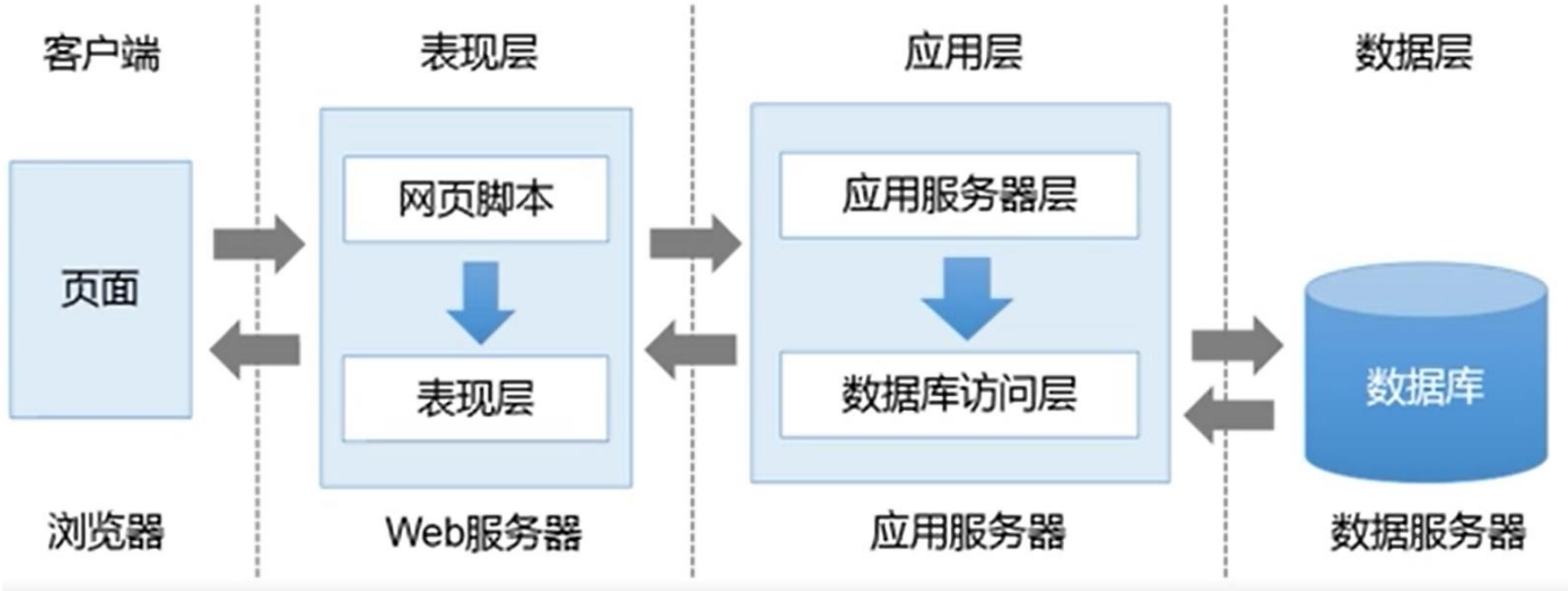
- 表示层：包括所有与客户机交互的边界对象，如窗口、表单、网页等。
- 功能层（业务逻辑层）：包括所有的控制和实体对象，实现应用程序的处理逻辑和规则。
- 数据层：实现对数据库的存储、查询和更新。



内容来源：
https://next.xuetangx.com/learn/THU08091000367/THU08091000367/1516221/video/1385976

B/S架构

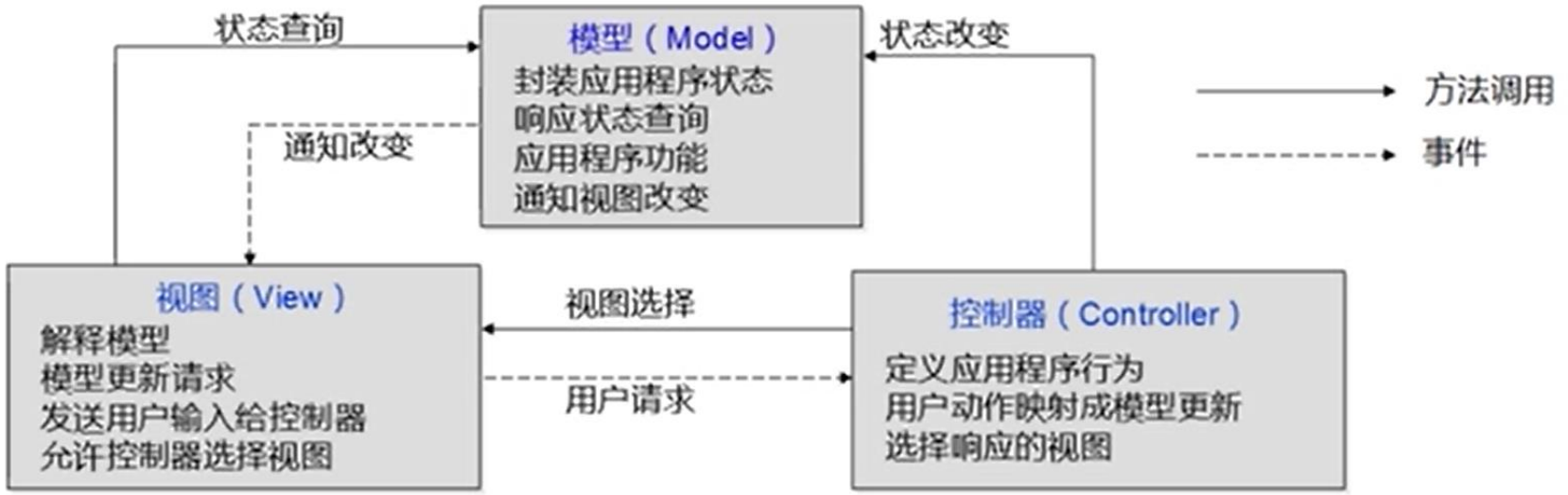
浏览器/服务器（Browser/Server）结构是三层C/S风格的一种实现方式。



内容来源：
https://next.xuetangx.com/learn/THU08091000367/THU08091000367/1516221/video/1385976

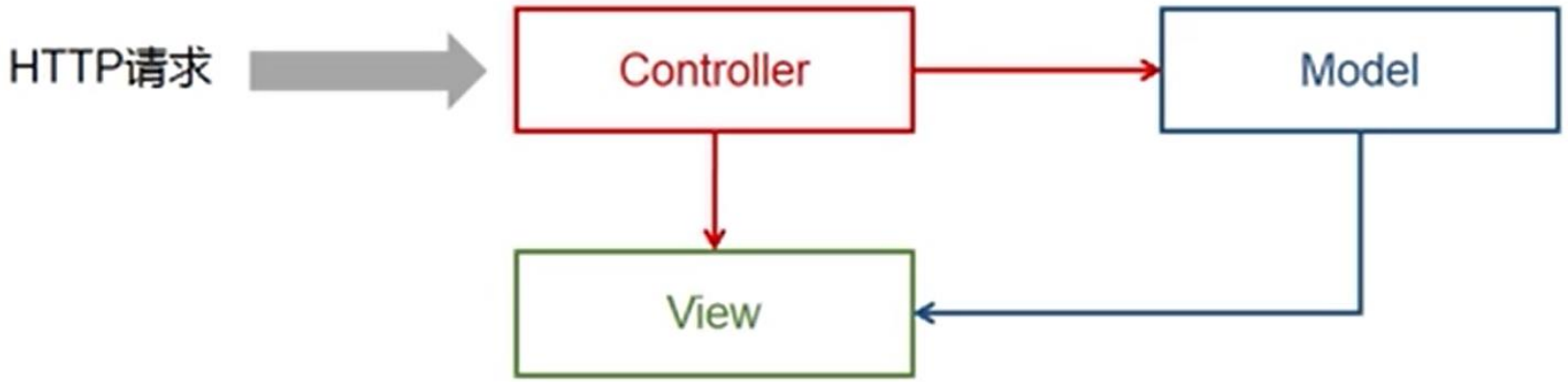
MVC三层结构

模型-视图-控制器（MVC）结构将应用程序的数据模型、业务逻辑和用户界面分别放在独立构件中，这样对用户界面的修改不会对数据模型/业务逻辑造成太大影响。



内容来源：
https://next.xuetangx.com/learn/THU08091000367/THU08091000367/1516221/video/1385976

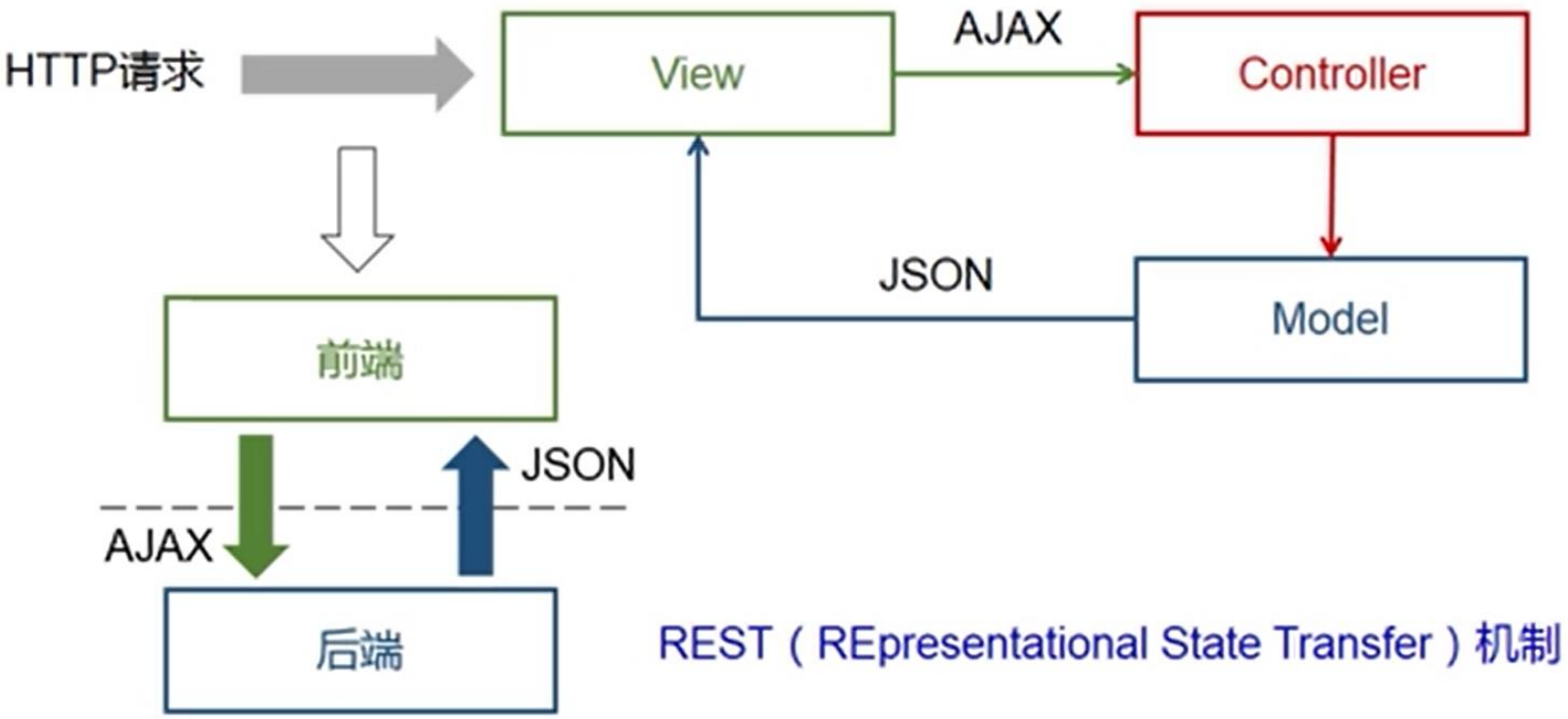
MVC结构的请求访问路径



- 每次请求必须经过“控制器->模型->视图”过程，才能看到最终展现的界面
- 视图是依赖于模型的
- 渲染视图在服务端完成，呈现给浏览器的是带有模型的视图页面，性能难优化

内容来源：
https://next.xuetangx.com/learn/THU08091000367/THU08091000367/1516221/video/1385976

前后端分离的MVC三层模式



内容来源：
https://next.xuetangx.com/learn/THU08091000367/THU08091000367/1516221/video/1385976

设计模式

- GOF的《设计模式--可复用面向对象软件的基础》一书总结了一套被反复使用的、经过分类编目的、代码设计经验的总结。
- GOF不是一个人，而是指四个人。它的原意是Gangs Of Four,就是“四人帮”，就是指此书的四个作者：Erich Gamma,Richard Helm, Ralph Johnson,John Vlissides。
- 这本书讲了23种主要的模式，包括：
  - 工厂模式（Factory Pattern）
  - 抽象工厂模式（Abstract Factory Pattern）
  - 单例模式（Singleton Pattern）
  - 建造者模式（Builder Pattern）

补充练习

单选题 1分

设置

总体设计不包括_____

A 体系结构设计

B 接口设计

C 数据设计

D 数据结构设计

提交

填空题 0分

设置

此题未设置答案，请点击右侧设置按钮

模块化的基本原则是_____

高内聚低耦合

作答

正常使用填空题需3.0以上版本雨课堂

单选题 1分

设置

通信内聚是指_____

A 把需要同时执行的动作组合在一起形成的模块

B 各处理使用相同的输入数据集或产生相同的输出数据集

C 一个模块内各个元素都密切相关于同一功能且必须顺序执行

D 模块内所有元素共同完成一个功能，缺一不可

提交

单选题 1分

设置

关于模块的扇入扇出，以下说法正确的是____

A 扇入表示有多少个上层模块直接或间接调用它

B 模块扇入高时应当重新分解，以消除控制耦合的情况

C 一个模块的扇出太多，说明该模块过分复杂，缺少中间层

D 一个模块的扇入太多，说明该模块过分复杂，缺少中间层

提交

单选题 1分

设置

划分模块时，一个模块的_____

A 作用范围应在其控制范围内

B 控制范围应在其作用范围内

C 作用范围与控制范围互不包含

D 作用范围与控制范围不受任何限制

提交

单选题 1分

设置

在对初始的SD精化过程中，将多个模块公用的子功能独立出来，形成一个新的模块，这利用了哪一条启发式规则？

A 改进软件结构，提高模块独立性

B 模块规模适中，每页60行语句

C 模块的作用域力争在控制域之内

D 降低模块接口的复杂性

提交

多选题 1分

设置

以下说法错误的是_____

A 启发式规则是人们从长期的软件开发实践中总结出来的规则，在设计中应当普遍严格遵循

B 扇入扇出应当适中，尽量满足7+2原则

C 好的设计控制域应当包含作用域

D 为了降低模块接口的复杂性，必须将多个同类型的参数合并为一个数组进行传递

提交

单选题 1分

设置

接口设计的主要内容是_____

A 模块或软件构件间的接口设计

B 软件与其他软硬件系统之间的接口设计

C 软件与用户之间的交互设计

D 以上都是

提交

单选题 1分

设置

用户界面应具备的特性中，最重要的是_____

A 可使用性

B 灵活性

C 可靠性

D 可扩展性

提交

单选题 1分

设置

设计人机交互的界面时，应当遵循一定的设计原则，不包括_____

A 操作步骤少

B 提供undo功能

C 减少人脑的记忆负担

D 增加复杂的功能

提交

end