本章主要内容

结构化设计方法，模块分解，耦合和内聚

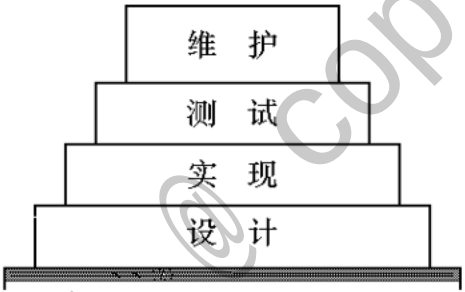
分解的启发式规则

第四章 总体设计

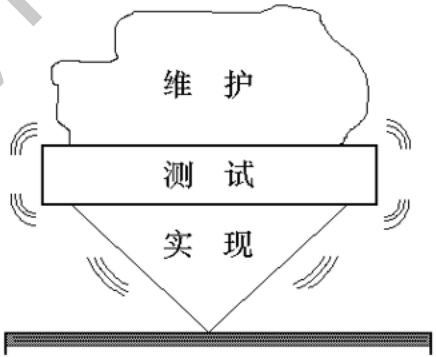
1. 总体设计概要
2. 模块化设计
3. 启发性规则
4. 结构化设计方法
5. 总体设计的图形工具

软件设计的重要性

软件设计是后续开发步骤及软件维护工作的基础。如果没有设计，只能建立一个不稳定的系统结构。



有软件设计



没有软件设计

4.1 总体设计概要 - 概念

总体设计又称为**概要设计**或**系统设计**，它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。

- 有无可复用的元素
- 系统蓝图（架构、关键技术）
- 考虑演变过程（性能、成本、原型开发）

设计过程中划分出组成系统的**物理元素**——**程序**、**文件**、**数据库**、**人工过程**和**文档**等，并确定系统中每个程序由哪些模块组成以及这些模块相互间的关系。

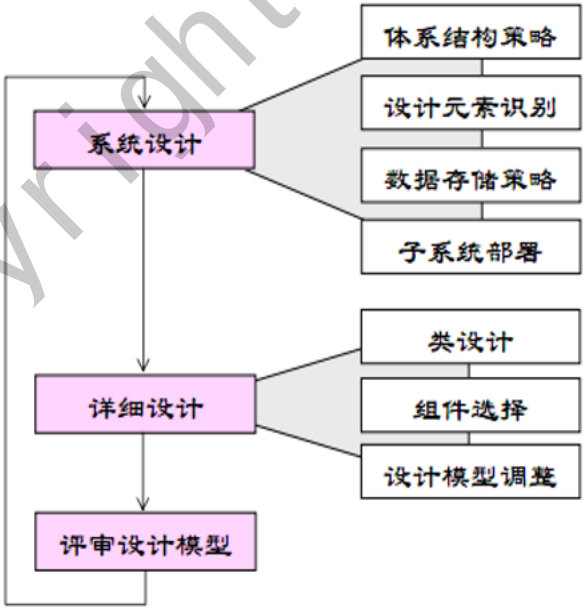
- **从回答“做什么”到回答“怎样做”**

软件设计过程概要

- 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**；

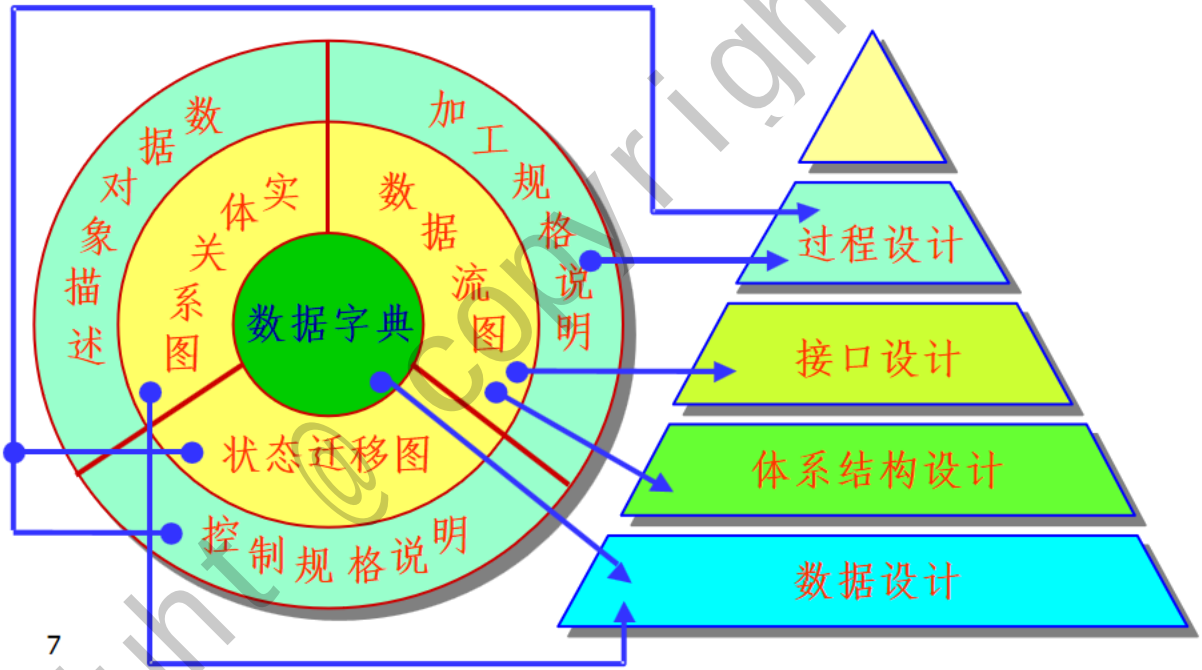
- 设计主要涉及以下几个方面：

体系结构设计、用户界面设计、数据库设计等方面。

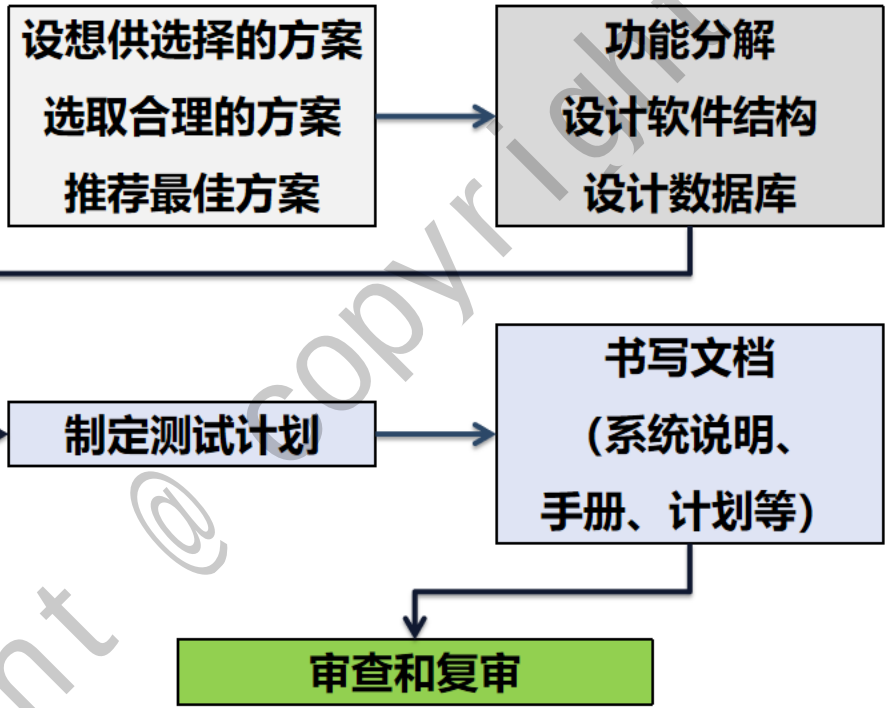


总体设计建模

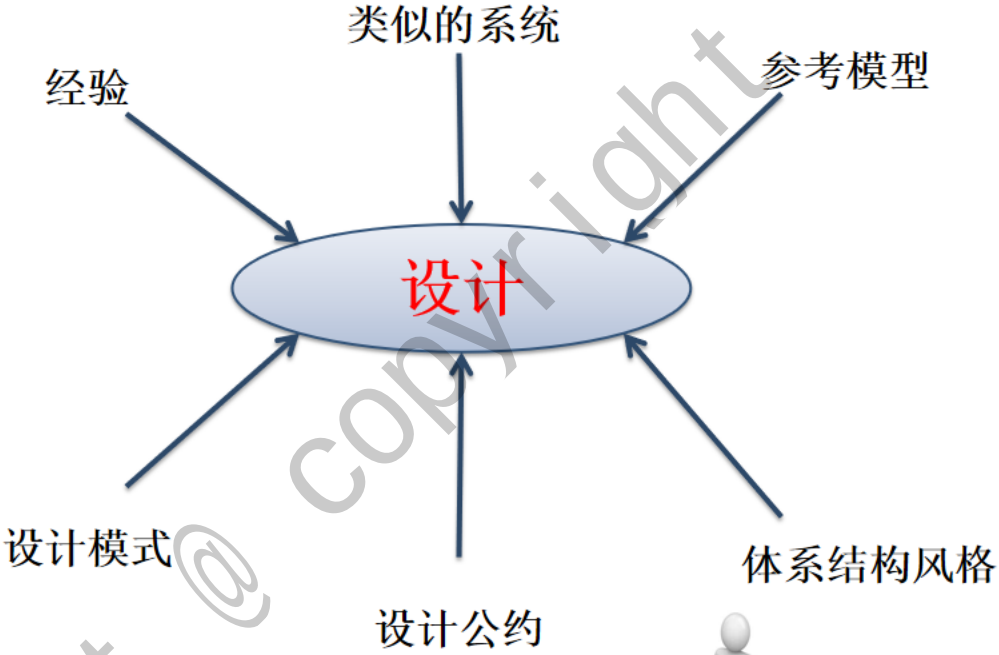
需求模型 --》 设计建模



总体设计步骤



设计的要素



创造性过程！

一切的开端？ 软件架构设计

常见软件体系结构/架构

- 管道、过滤器
- 客户-服务器
- P2P
- 信息库/黑板模式
- MVC
- SOA

克隆？

整合？

已有框架？

约束？

...

复杂的系统可以是这些体系的组合！

设计原则

- **抽象**

  首先用高级的抽象概念构造系统，抽出事物的本质而暂不考虑他们的细节。

- **逐步求精**

  求精是细化的过程。

  对抽象程序逐步分解，直到程序能被计算机接受为止。

- **模块化**

4.2 模块化设计

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)。**模块**(module)是**数据说明**和**可执行语句**等程序对象的集合，每个模块单独命名并且可以通过名字对模块进行访问。

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**。

一、分解

设函数C(x)定义问题x的复杂程度，函数E(x)确定解决问题x所需要的工作量（时间）。对于两个问题P_1和P_2，如果 $C(P_1)>C(P_2)$，

显然 $E(P_1)>E(P_2)$

根据人类解决一般问题的经验，如果一个问题由P_1和P_2两个问题组合而成，那么它的复杂程序大于分别考虑每个问题时的复杂程度之和，即 $C(P_1+P_2)>C(P_1)+C(P_2)$

综上所述，可得到下面的不等式

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

模块化设计 - 分解

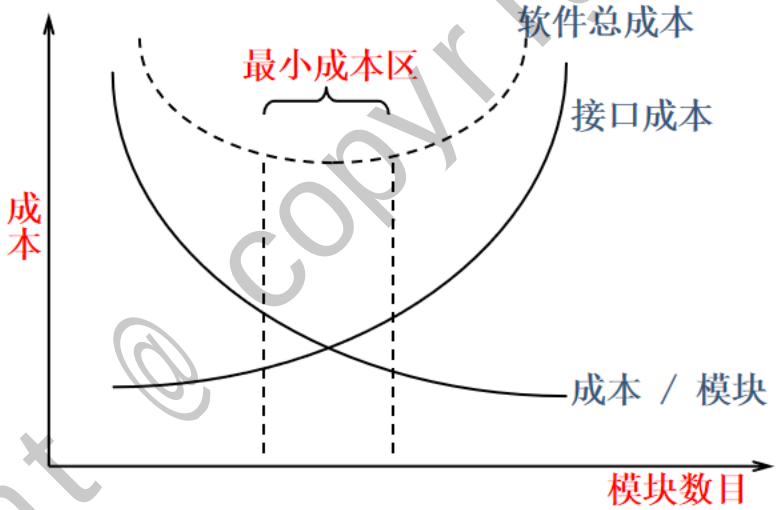
分解论据：

- 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。

模块分解和软件成本

- 如何确定地预测最小成本区？



模块设计原则 - 信息隐藏

二、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。

三、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。一般地，坚持模块的独立性是获得良好设计的关键。

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。

1、耦合

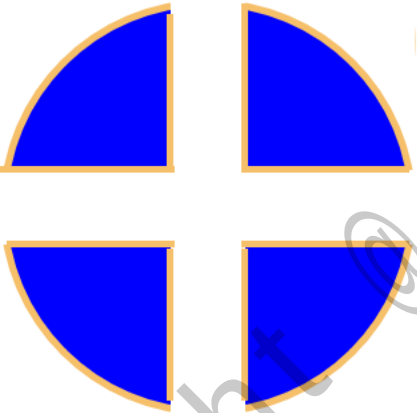
耦合是对一个软件结构内不同模块之间互联程度的度量。耦合强弱取决于模块间接口的复杂程度、进入或访问一个模块的点以及通过接口的数据。

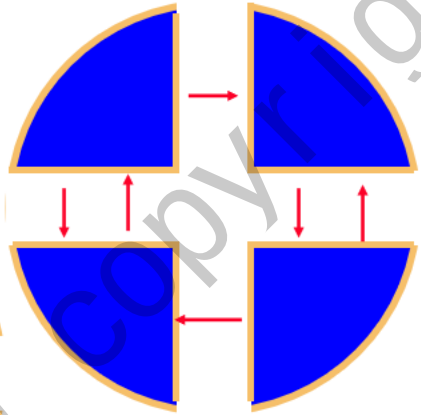
在软件设计中应该追求模块间尽可能松散耦合的系统。在这样的系统中可以研究、测试或维护任何一个模块，而不需要对系统中的其他模块有很多的了解。此外，由于模块间联系简单，发生在一处的错误传播到整个系统的可能性就很小。

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。

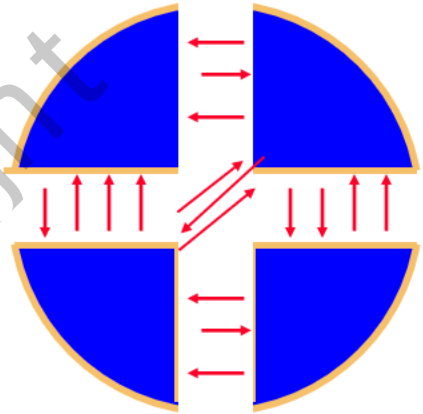
模块耦合度

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

耦合性原则

结构图设计的主要工作

- 把系统分解成若干个暗盒模块
- 设法将这些模块组织成合理的结构

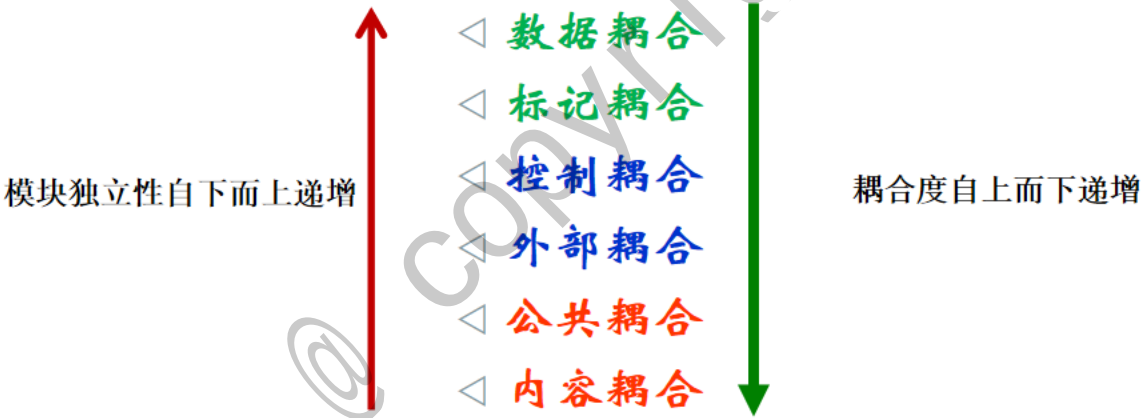
模块之间的必要联系是不可避免的，用模块的耦合性，来表示一个模块与其它模块之间联系的紧密程度

耦合性越高，模块之间联系就越多，相互影响就越大

应遵循“高内聚、低耦合”的原则

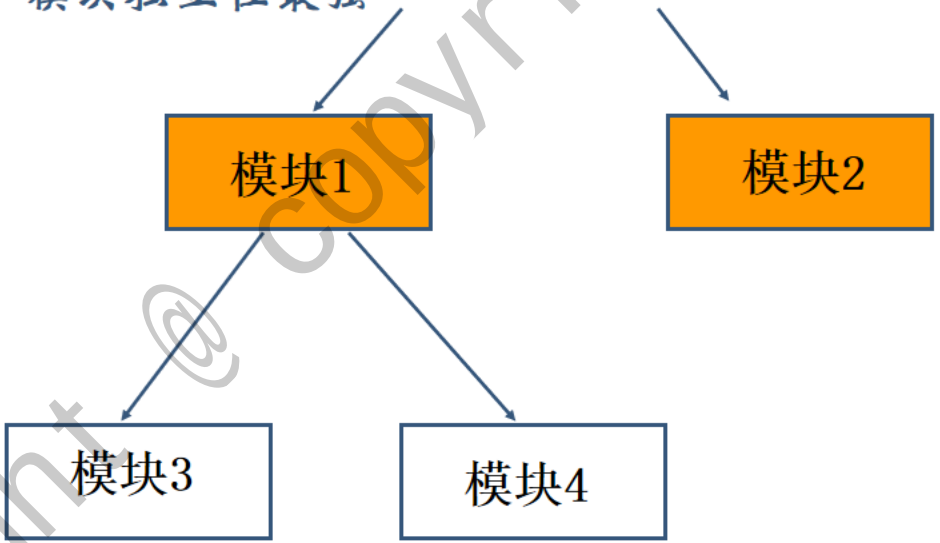
常见耦合类型

- 耦合的七种类型



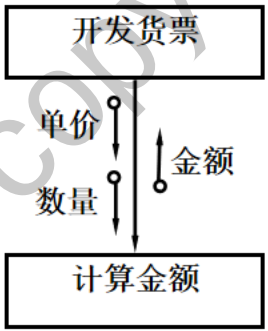
1. 非直接耦合（no direct coupling）

- 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



2.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。



模块间的数据耦合

3. 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。

“住户情况”包含水费和电费、煤气费、电话费等。

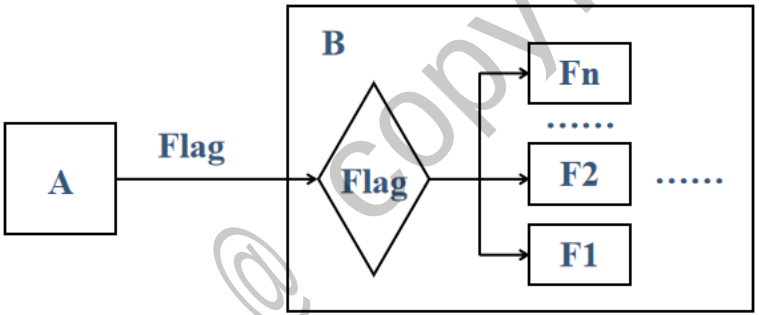
“住户情况”是一个数据结构，图中模块都与此数据结构有关.

“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.

4. 控制耦合 （control coupling）

- 两个模块之间不仅存在调用与被调用关系，而且模块A向模块B传递的信息控制了模块B的内部处理过程。

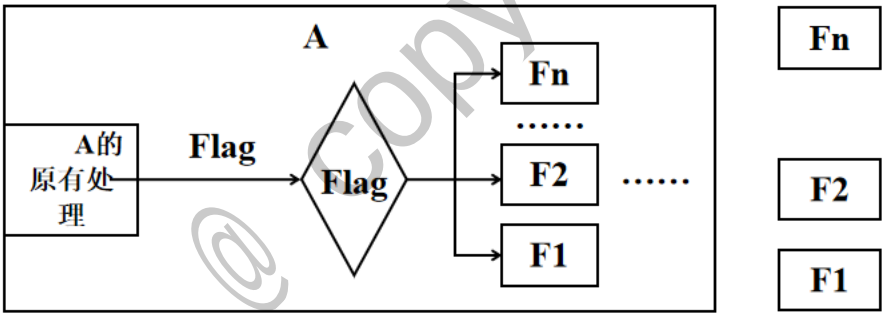
控制信息实质上相当于一个“标识”、或一个“开关”



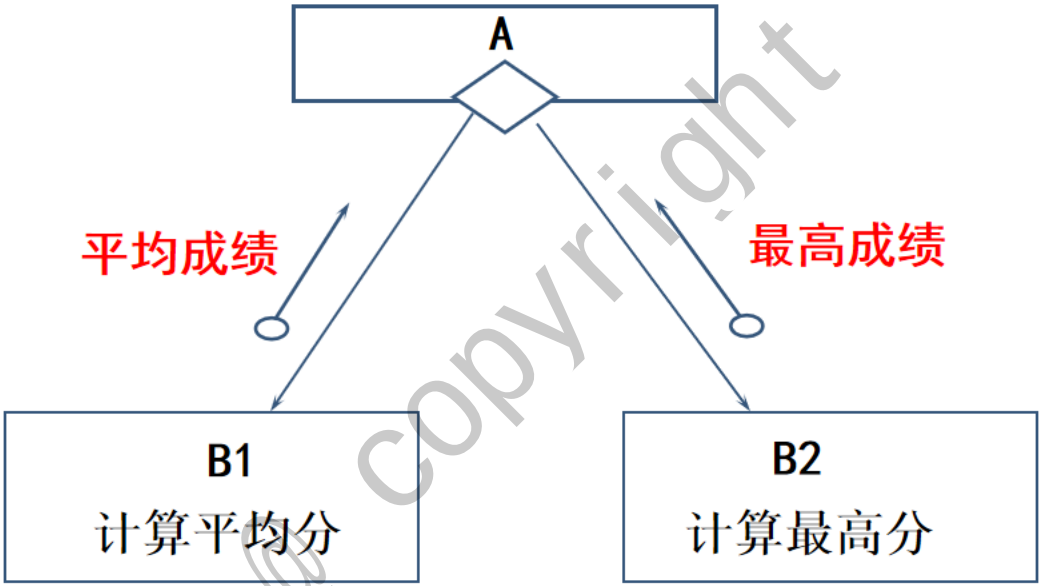
特点：接口单一，但仍然影响被控模块的内部逻辑。

控制耦合解耦方法

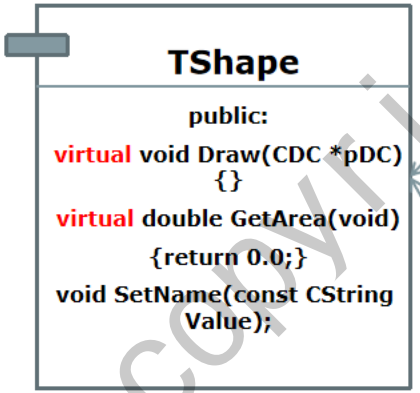
- 面向**过程**的修改方法：
  - 在调用点进行条件判断；
  - 将B函数按照子功能拆分：

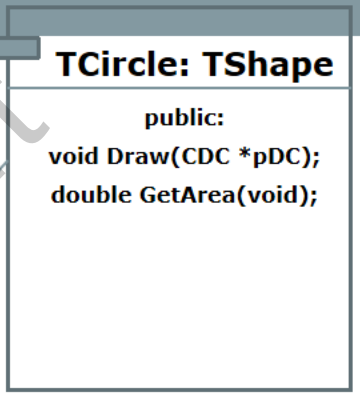


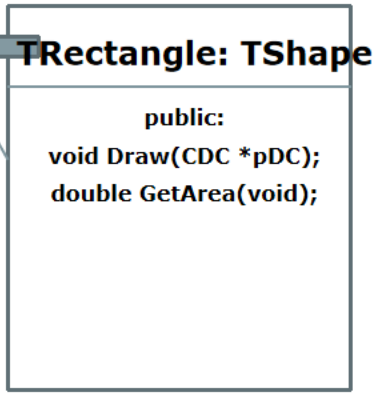
改控制耦合为数据耦合举例



参考，面向对象的解决方法







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

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);
    。。。

}
```

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);
    。。。
}
```

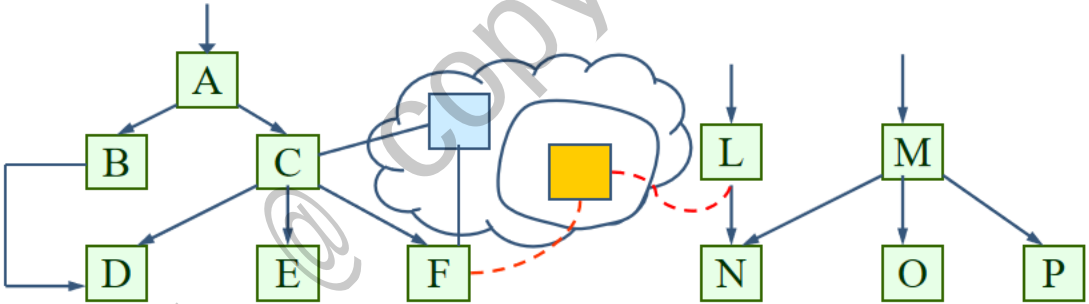
java实现

```
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. 外部耦合(external coupling)

- 共用全局变量，不用参数表传递消息

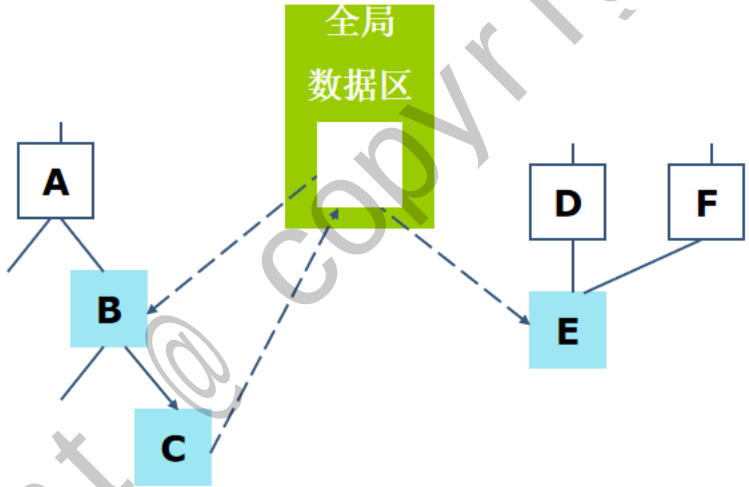
*类似于公共耦合，区别在外部耦合是简单的全局变量，而不是全局数据结构。



6. 公共耦合(common coupling)

- 两个模块都和同一个公用数据域有关

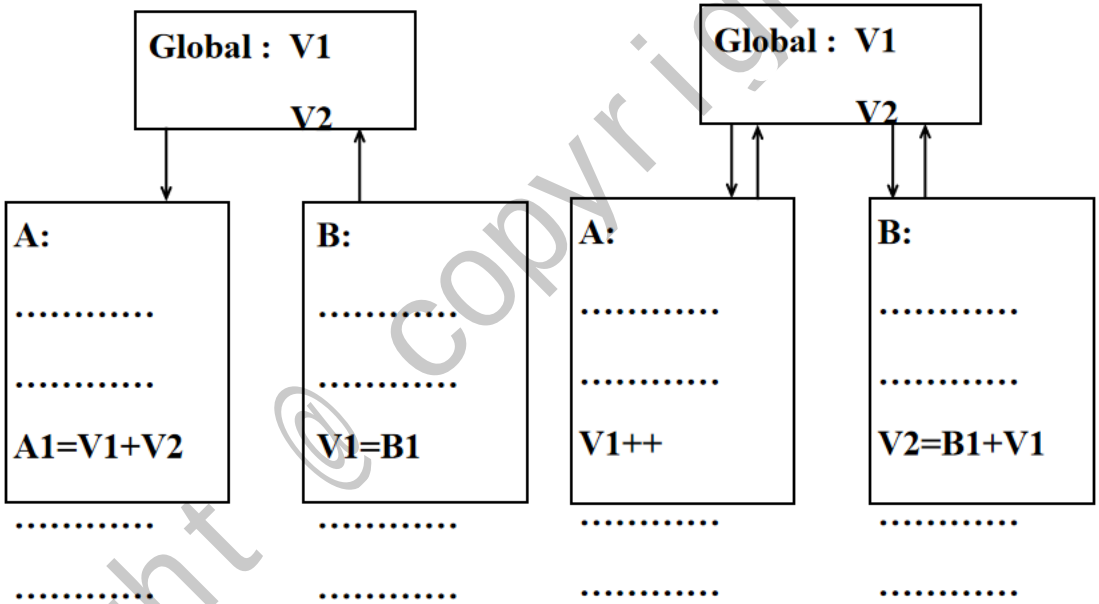
公共数据域可以是全局变量，共享的公共数据块，内存的公共覆盖区等



B、C、E 为公共耦合

公共耦合例子

使用了全局变量



避免公共耦合的原因

1

- 公用数据域是共享的，任何时间、任何模块都可以修改这些数据

2

- 当一个公共数据发生变化时，与之有关的模块都应随之而修改，而事实上很难判定

3

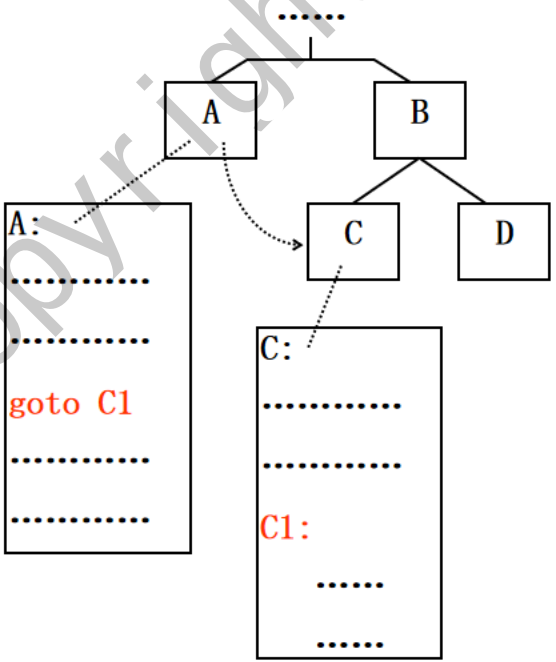
- 公共耦合是隐式(即没采用模块调用格式中明确表示出来)

模块共享数据较多，参数传递复杂，可适当利用公共耦合，但是要注意严格限制数据更新和引用的逻辑顺序。

7. 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。



块间联系的设计原则

- 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。

练习

- 什么耦合应该完全避免？ 内容耦合
- 使用全局变量会产生什么耦合？ 公共耦合
- 举例说明什么是控制耦合？ 传控制型参数

模块独立性 - 内聚

2、内聚

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。

内聚的种类

模块的内聚性—也称紧凑性，是指模块内部各组成部分为了执行处理功能而组合在一起的相关程度

提高系统的内聚性就是要讨论把什么样的内容放入同一个模块，才能使该模块尽量保持单一功能的问题

1. 功能内聚

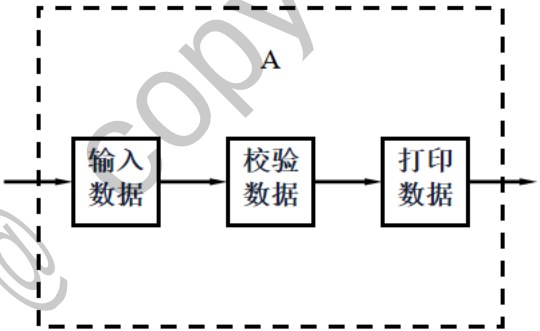
模块内所有处理元素属于一个整体，完成一个单一的功能。

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。

2. 顺序内聚(Sequential cohesion)

2. 顺序内聚

一个模块内部各个组成部分的处理动作是按顺序执行的，且前一个处理动作所产生的输出数据是下一个处理动作的输入数据

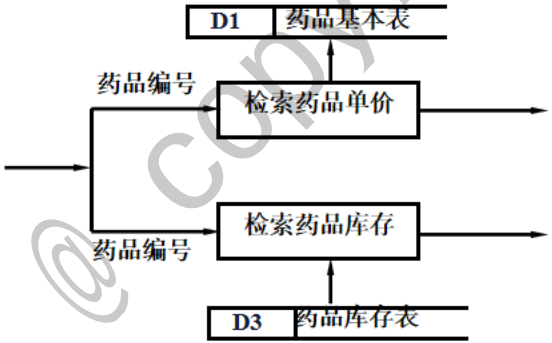


顺序内聚

3. 通信内聚(Communicational cohesion)

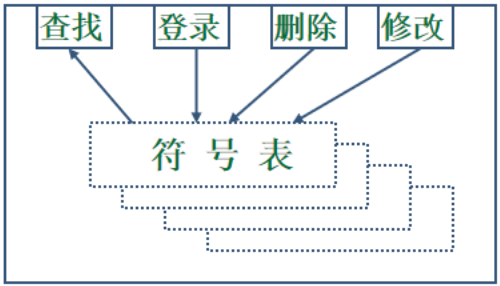
3. 通信内聚

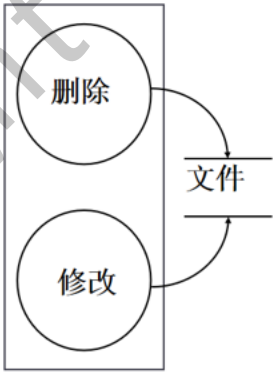
- 一个模块内部的各个组成部分的处理动作都引用相同的输入数据或产生相同的输出数据

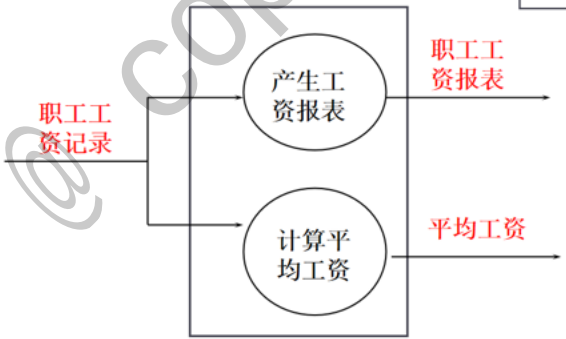


通信内聚

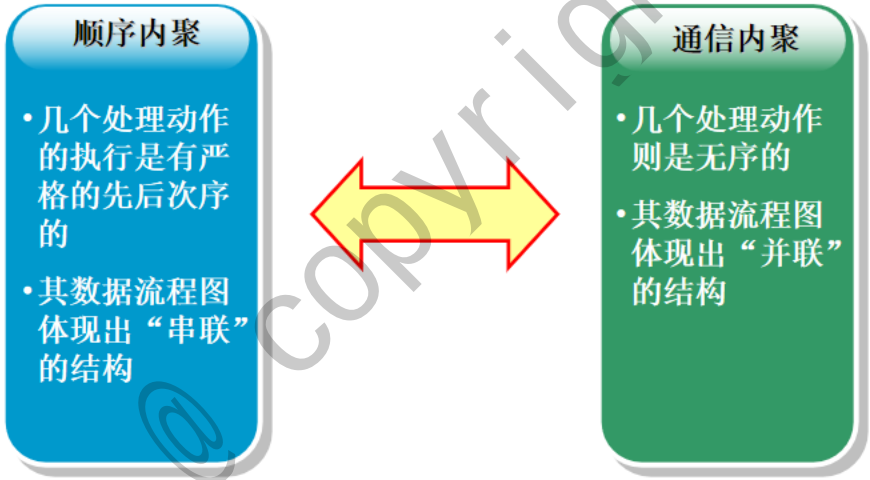
通信内聚例子







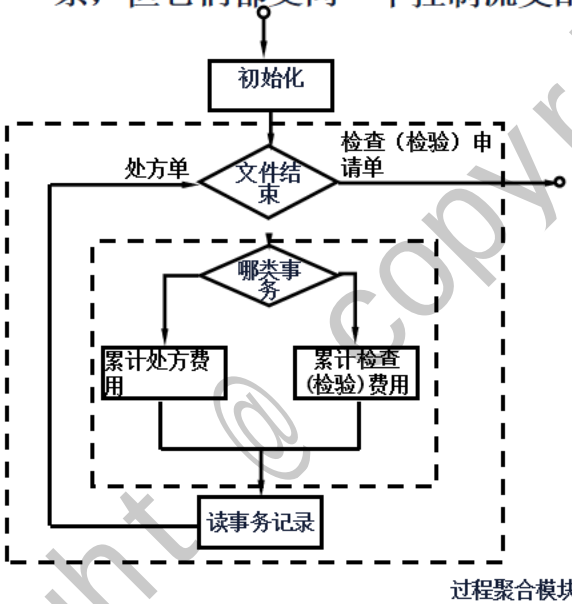
通信内聚与顺序内聚的区别



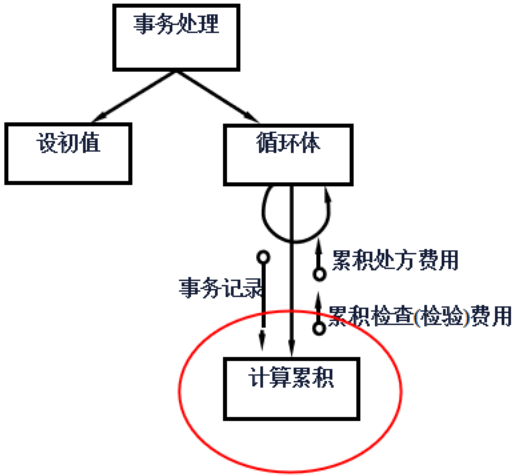
4.过程内聚（Procedural cohesion）

4. 过程内聚

一个模块内部各组成部分的处理动作各不相同，彼此也没有什么关系，但它们都受同一个控制流支配，决定它们的执行顺序



过程聚合模块

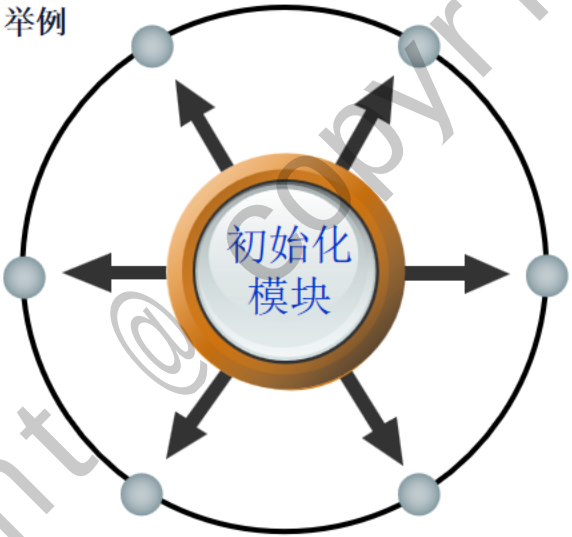


5. 时间内聚（Temporal cohesion）

5. 时间内聚

一个模块的各组成部分，它们的处理动作和时间有关

举例

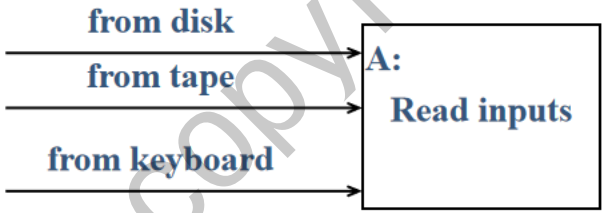


不同的功能混在一个模块中，有时共用部分编码，使局部功能的修改牵动全局。

6.逻辑内聚（Logical cohesion）

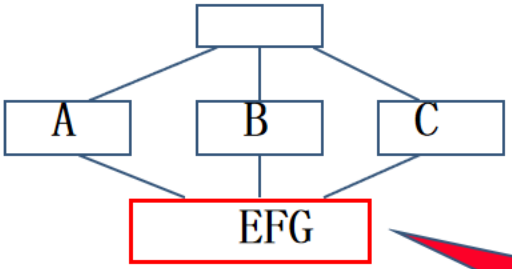
6. 逻辑内聚

- 逻辑相关的函数或数据出现在同一个模块里。（逻辑上相似）



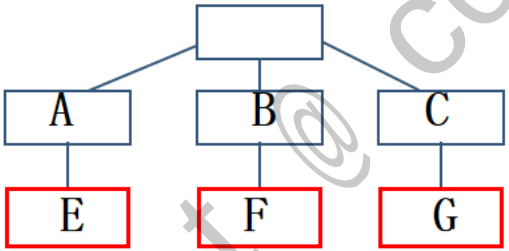
- 问题：接口难于理解；完成多个操作的代码互相纠缠在一起，导致严重的维护问题。

逻辑内聚变换例



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



7.偶然内聚(Coincidental cohesion)

7. 偶然内聚

无关的函数、过程或者数据出现在同一个模块里。

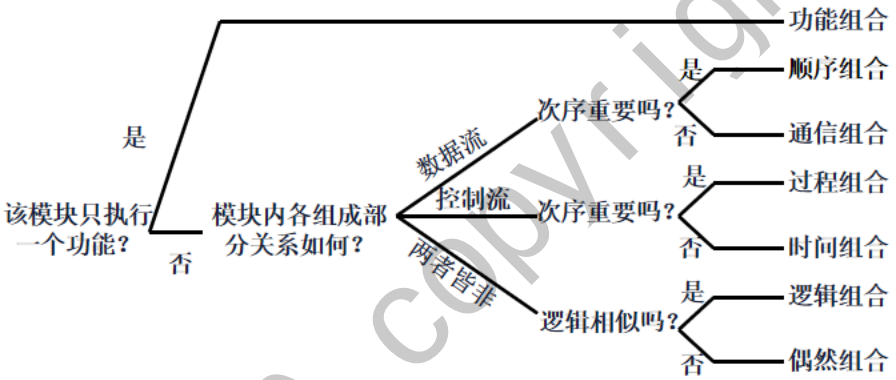
例：read disk file;
calculate current values;
produce user output; …

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。

内聚性判断方法

判定一个模块的内聚程度，可借助判断树来分析

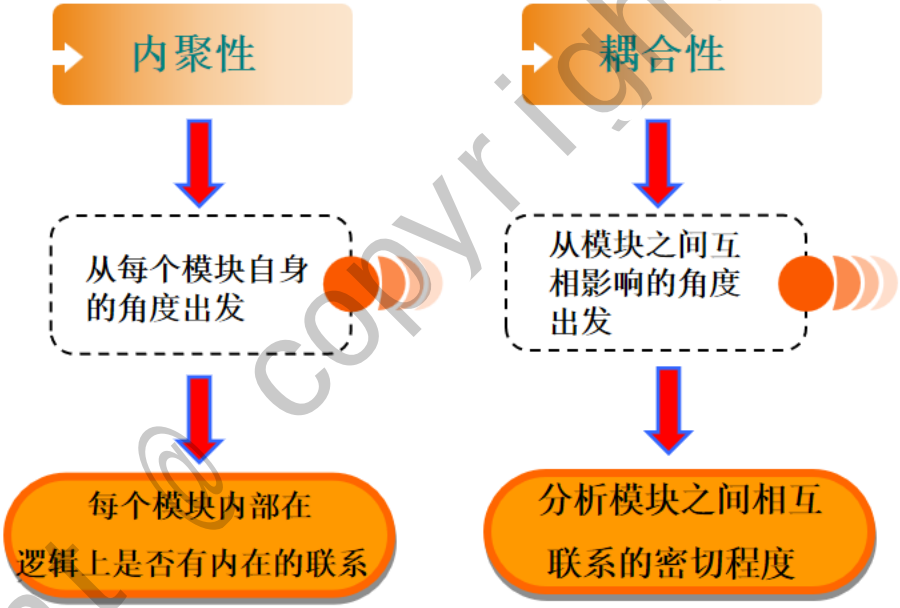


模块组合形式判断树

为了提高模块的独立性，应该多采用功能内聚、顺序内聚和通信内聚；
少采用过程内聚、时间内聚；
尽可能地避免逻辑内聚、偶然内聚

模块化设计总结

模块的内聚性和模块间的耦合性



相辅相成：通常，提高模块内部的内聚性，必然会降低模块间的耦合性

模块化标准和评价角度

- 可分解性：将问题分解成子问题的系统化机制，能降低整个系统的复杂性
- 可组装性：现存的设计模块能够被组装成新系统
- 可理解性：模块可以作为一个独立的单位被理解，那么它就易于构造和修改
- 连续性：　如果对系统需求的微小修改只导致对单个模块而不是对整个系统的修改，则修改引起的副作用就会被最小化
- 保护：　异常限制在模块内

高内聚低耦合为基本原则

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

此题未设置答案，请点击右侧设置按钮

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

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

4.3 启发性规则

(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。

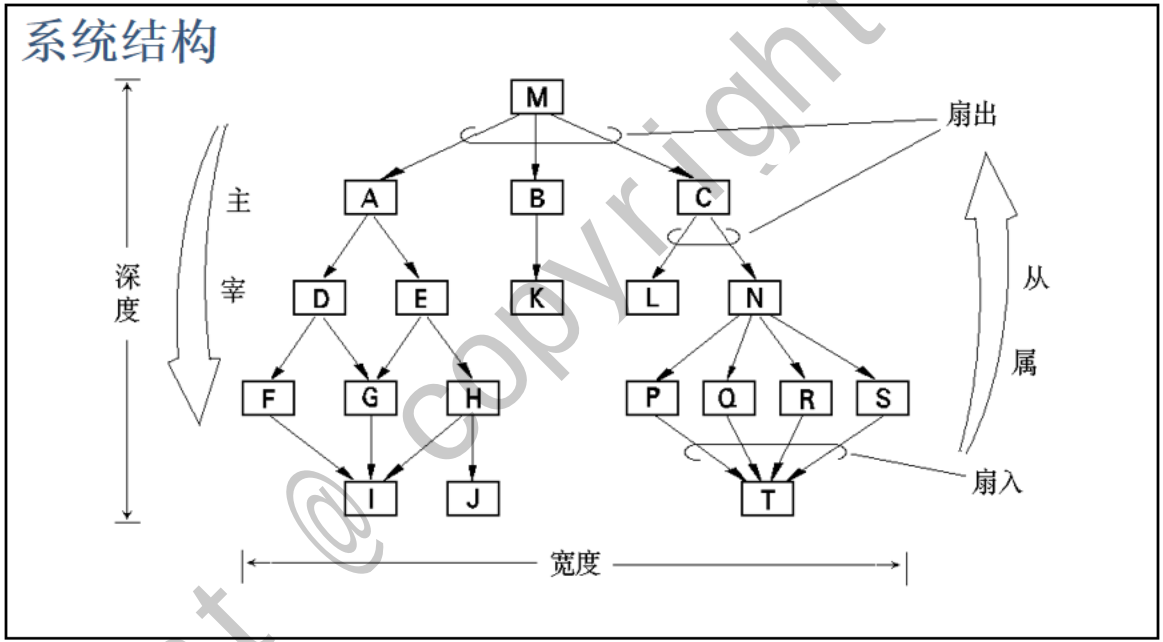
1. 模块独立性

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。

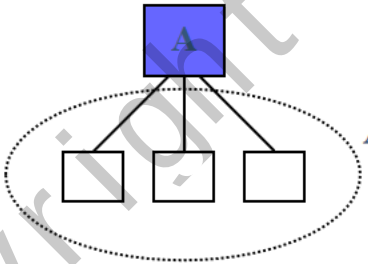
❖ 宽度 = 同一层上模块数的最大值。过大表示系统复杂度大。

2. 深度和宽度

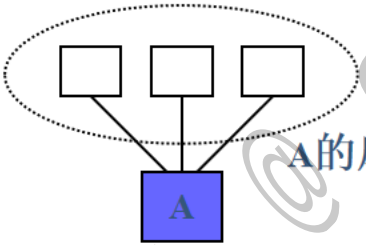


3. 扇入和扇出

- 扇出 = 一个模块直接调用/控制的模块数。 $3 \le \text{fan-out} \le 9$



A的扇出

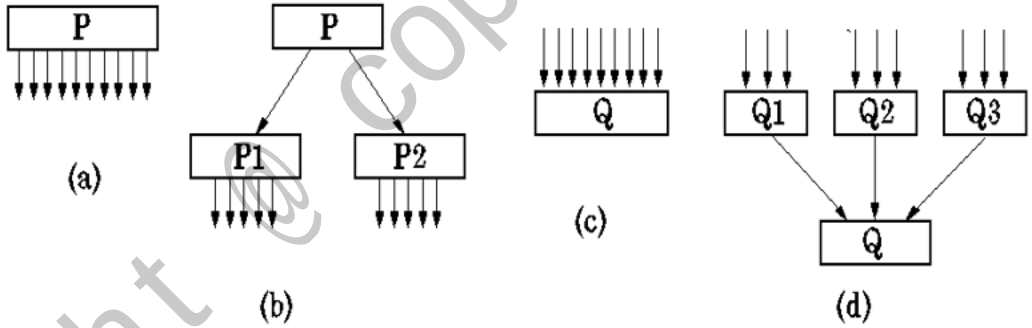


A的扇入

- 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。

扇入和扇出

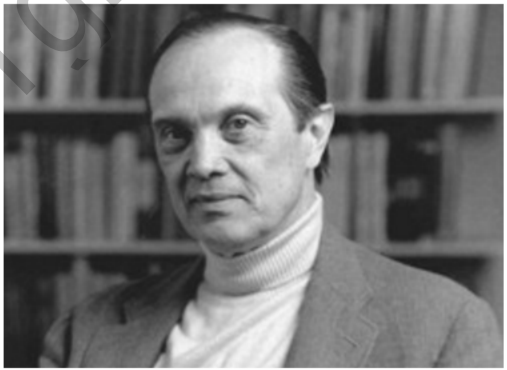
- 尽可能减少高扇出结构，随着深度增大扇入。
  如果一个模块的扇出数过大，就意味着该模块过分复杂，需要协调和控制过多的下属模块。应当适当增加中间层次的控制模块。
- 一般来说，顶层扇出高，中间扇出少，低层高扇入。



扇入和扇出

扇出的控制：3或4，上限为5~9

扇入越大，说明复用性越好。（注意前提是模块的独立性好）



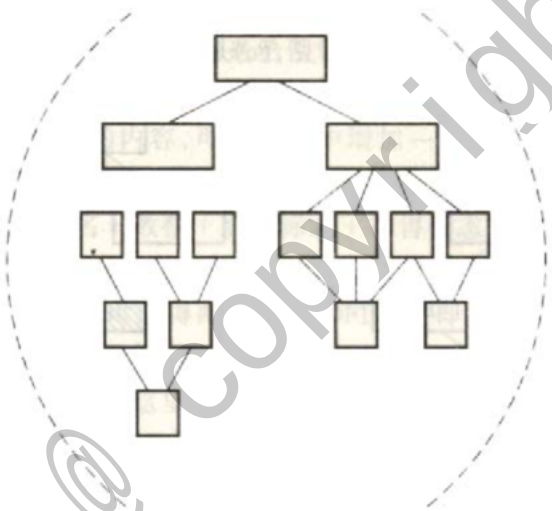
奇妙的数字 7±2，人类信息处理能力的限度

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



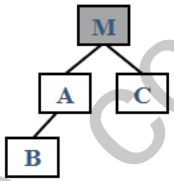
表明底层模块扇入高、共享程度好！

4. 模块的作用域

4. 模块的作用范围保持在该模块的控制范围内

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

◆ 控制域

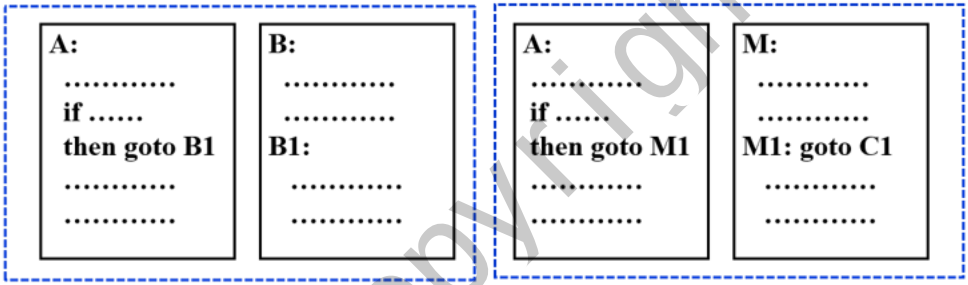


M的控制域为 {M，A，B，C}

◆ 作用域：M中的一个判定所影响的模块。

模块作用域的例子

例：

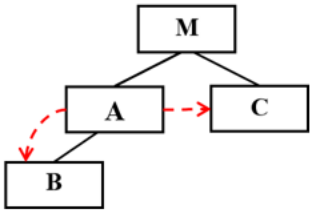


作用域在控制域内　　作用域超出了控制域

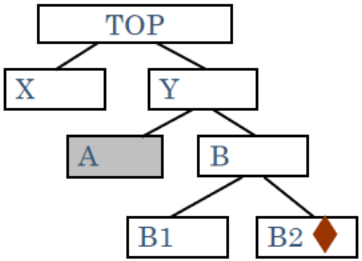
上右例中，A的作用超出了控制域。

改进方法之一，可以把A中的 if 移到M中；

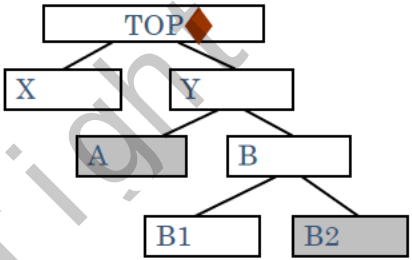
改进方法之二，可以把C移到A下面。



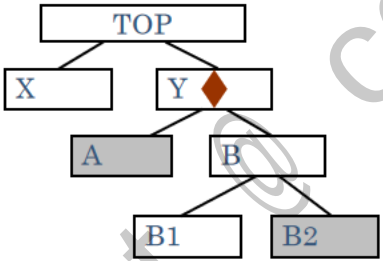
模块作用域的变换



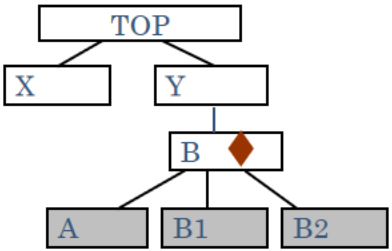
(a)



(b)



(c)



(d)

5、 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。

6、 单出单入，易于理解和维护。

7、 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。

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

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

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

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

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

4.3 启发性规则

怎样才能使所设计的系统具有合理的结构和良好的可维护性？

一句话：遵循“高内聚，低耦合，精分解，高扇入，低扇出”的原则

- 基于模块化、自顶向下细化、结构化程序设计等程序设计技术基础上发展起来的。

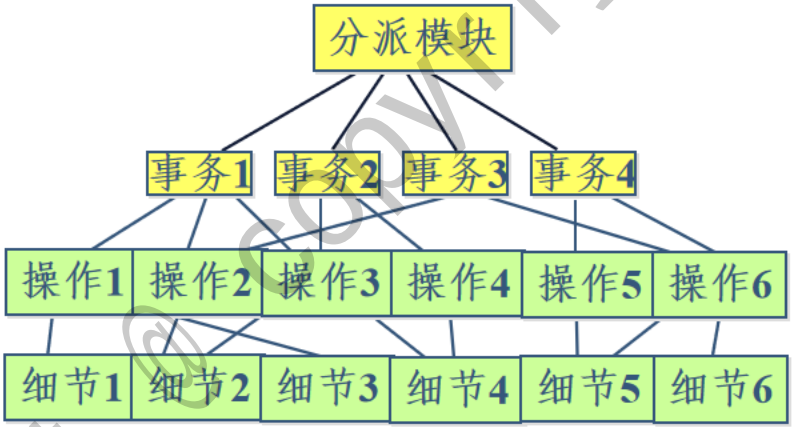
- 该方法实施的步骤是：
  1) 建立符合需求规格说明书要求的初始结构图（一般由数据流图导出初始结构图）；

  2) 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。

4.5 总体设计的图形工具

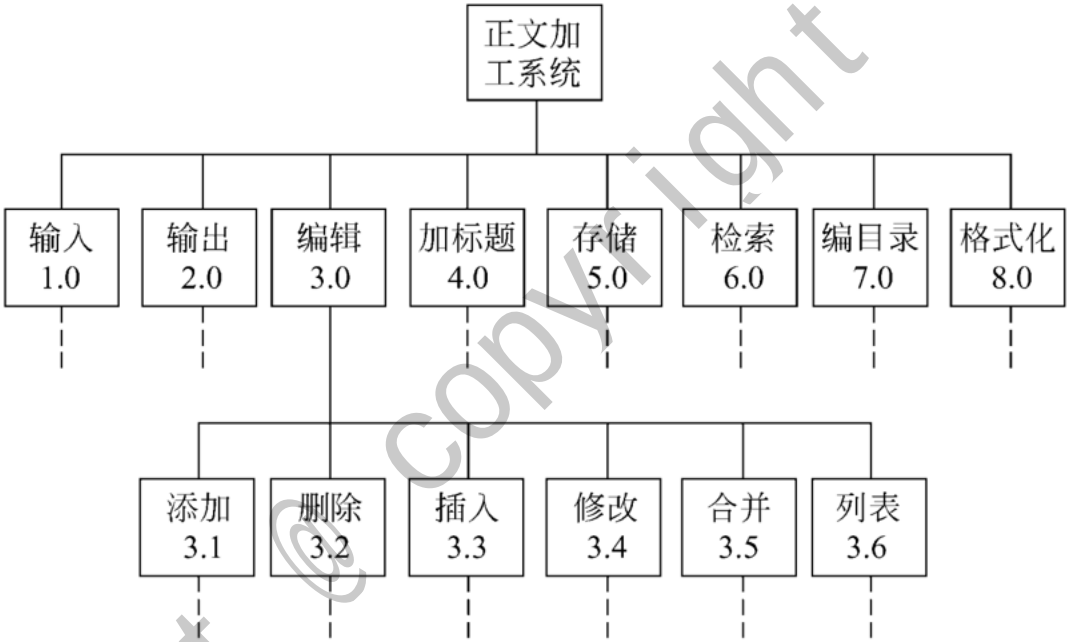
一、层次图

层次图（也称H图）是在总体设计阶段最常使用的图形工具之一，它常用于描绘软件的层次结构。层次图中的每个方框代表一个模块，方框间的连线表示模块间的调用关系。



？与层次方框图一样吗？

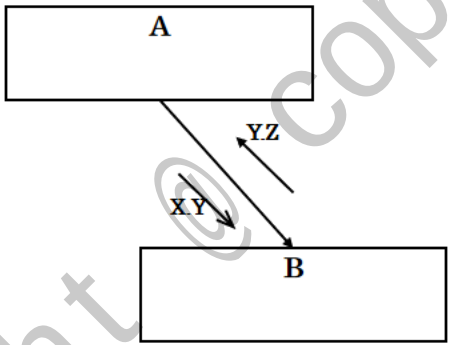
带编号的层次图（HIPO）



二、结构图

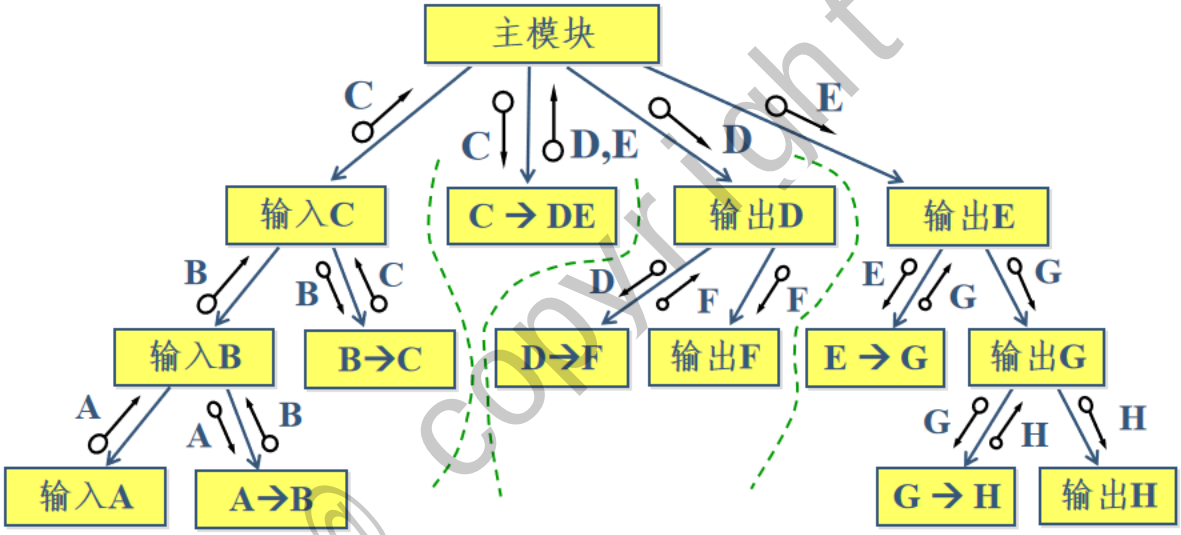
结构图中的每个方框代表一个模块，框内注明模块的名字或主要功能；方框之间的箭头（或直线）表示模块间的调用关系。

在结构图中通常还用带注释的箭头表示模块调用过程中模块之间传递的信息。可以利用注释箭头尾部的不同形状来区分：尾部是空心圆表示传递的是数据，尾部是实心圆则表示传递的是控制信息。



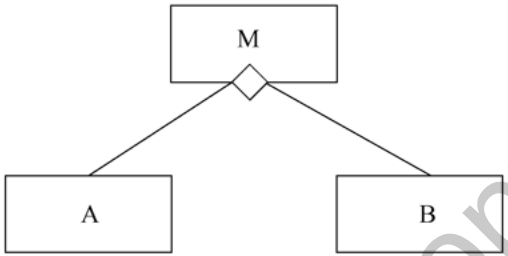
← 调用模块/调用者

← 被调模块/下层模块

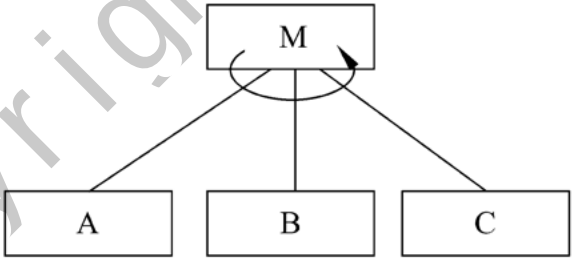

主模块
C
C
D,E
D
E
输入C
C → DE
输出D
输出E
B
B
C
D
F
F
E
G
G
输入B
B→C
D→F
输出F
E → G
输出G
A
A
B
G
H
H
输入A
A→B
G → H
输出H

结构图

结构图



选择调用



循环调用

注意点

- 层次图和结构图并不严格表示调用次序；
  （虽然画图习惯上是按调用序从左到右）
- 层次图和结构图仅表明一个模块调用哪些模块，至于内部其他成分则完全没有表示；

- 层次图导出结构图的过程，是检查设计正确性和模块独立性的好方法。

传送的数据是必需的吗？

传送的数据足够吗？

传送的数据和功能相关吗？

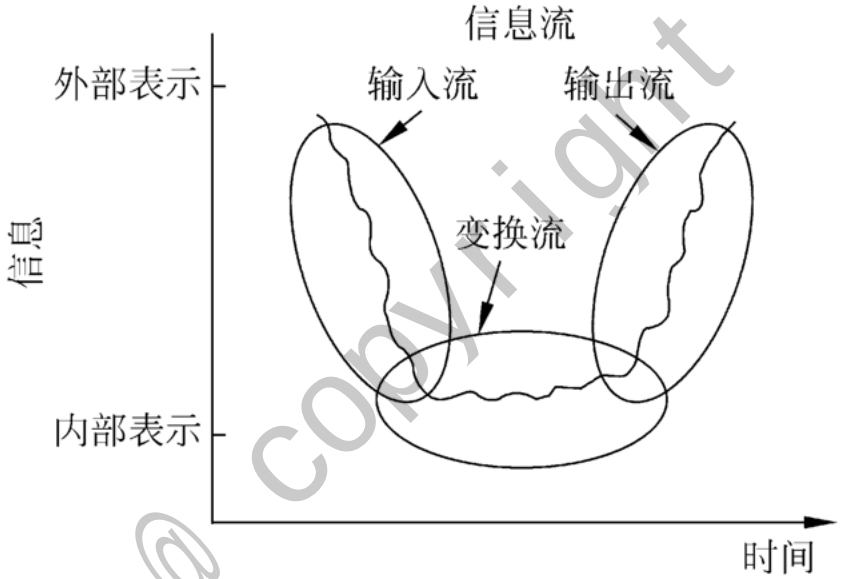
4.6 面向数据流的设计方法

目标：给出设计软件结构的一种系统化途径。

概念：把信息流映射成软件结构。利用这些映射将数据流图转换为软件结构，就是通常所说的结构化设计方法。

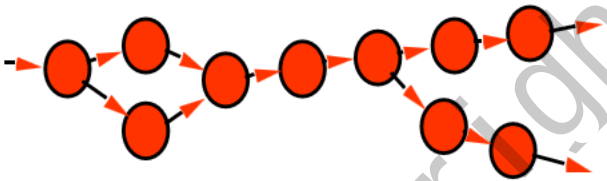
信息流有变换流和事务流两种形式：

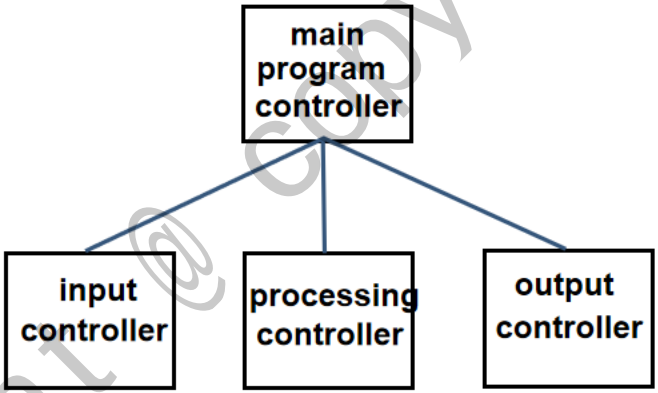
4.6.1 变换流（Transform flow）



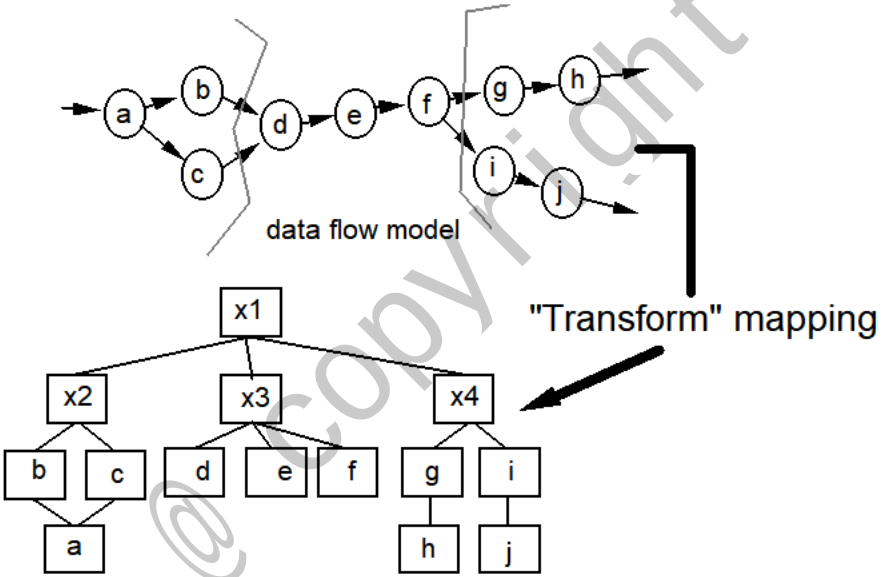
信息沿输入通路进入系统，由外部形式变换成内部表示，进入系统的信息通过变换中心，经加工处理以后沿输出通路变换成外部形式离开软件系统。

DFD的顶层变换





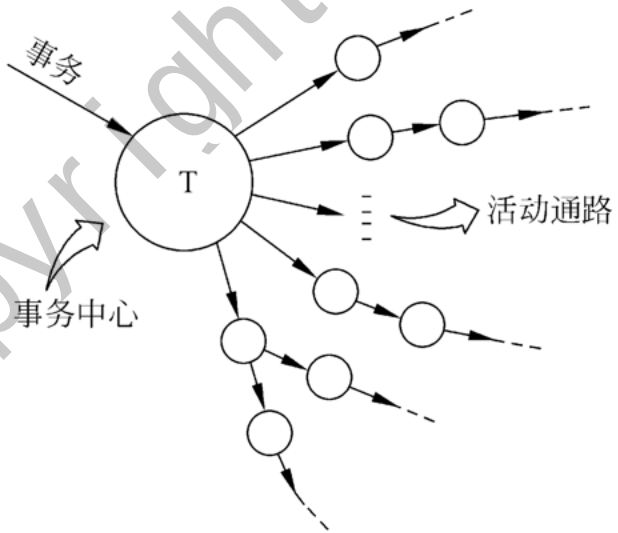
变换分析例子



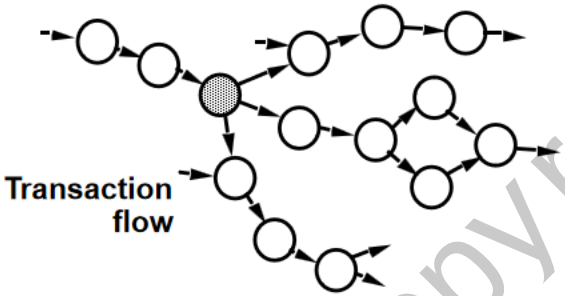
4.6.2 事务流（Transaction flow）

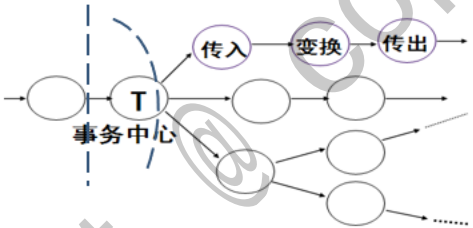
原则上所有信息流都可以归结为变换流。

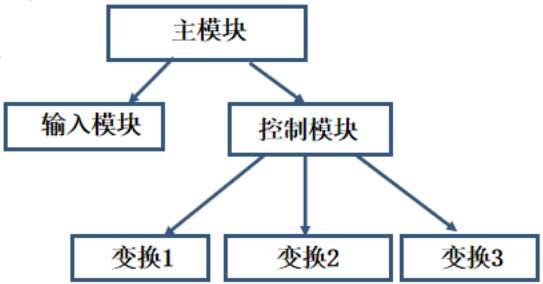
但是当数据流图“以事务为中心”，也就是说，数据沿输入通路到达一个处理T，这个处理根据输入数据的类型在若干个动作序列中选出一个来执行。这类数据流应该划为一类特殊的数据流，称为事务流。



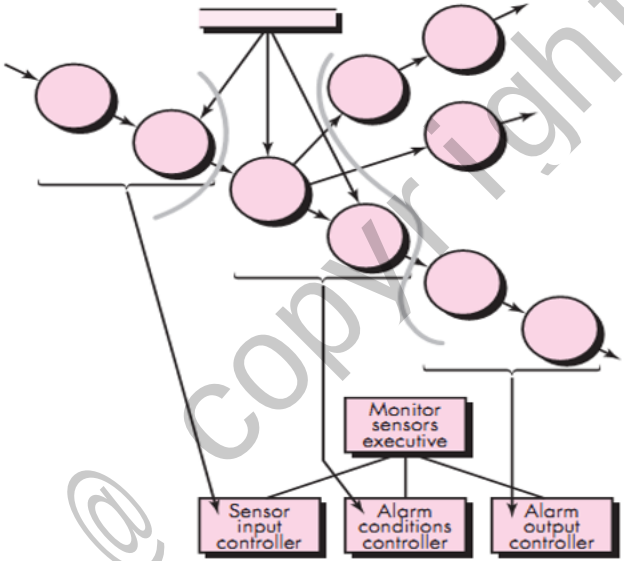
事务中心导出





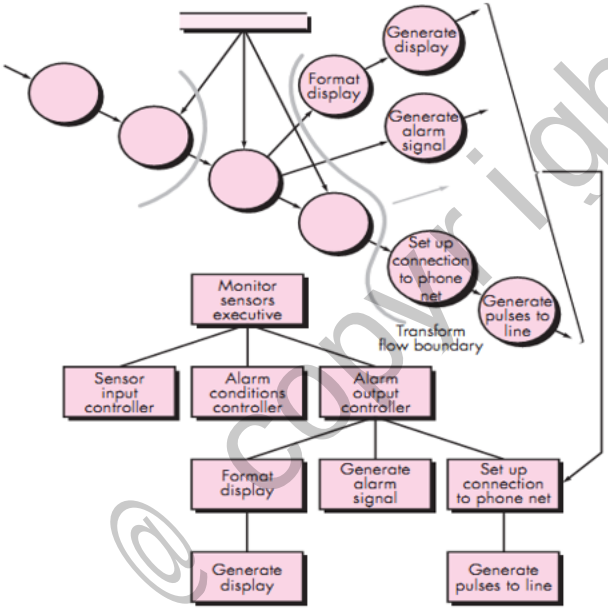


例子 -- 监控传感器



第一层映射

例子 -- 监控传感器



第二层映射

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。

SD方法步骤

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方法总结

- 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

小结

1) 模块化设计原则

2) 耦合和内聚

3) 设计技巧：扇入、扇出，深度、宽度，作用范围、控制范围。

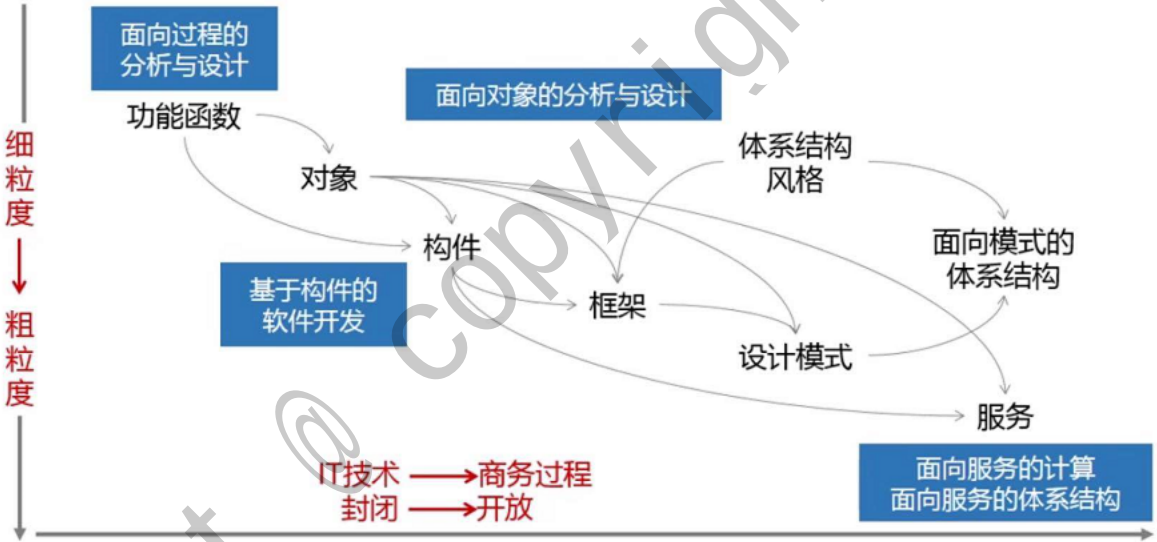
4) 层次图，结构图。

补充知识

软件设计的架构选择

- 软件体系结构发展轨迹
- 构件、框架
- 软件设计原则及层次化
- 常见模式/架构介绍

软件体系结构的发展



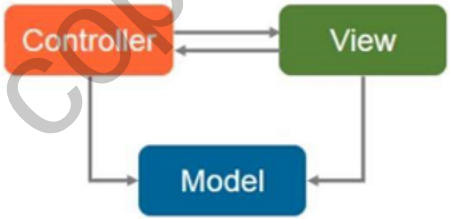
构件的基本概念

- 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。
- 软件构件是标准的、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

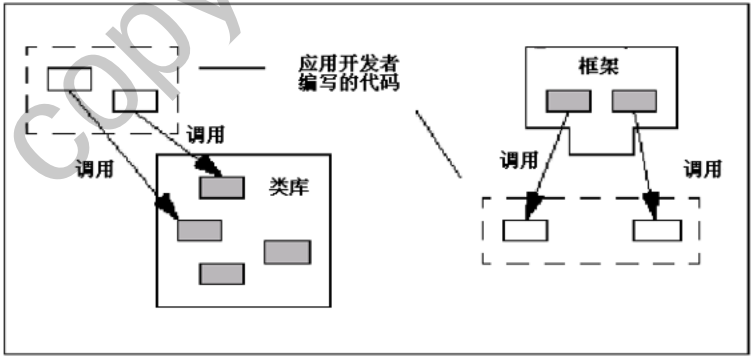
- 构件是可被用来构造其他软件的软件成分，基本形式可以是：
  - 被封装的对象类
  - 类树
  - 功能模块
  - 软件框架（framework）
  - 软件架构（或体系结构Architecture）
  - 文档
  - 分析件
  - 设计模式等

- 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.*

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

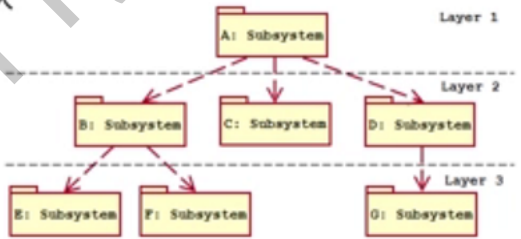
层次化的含义

分层（Layering）

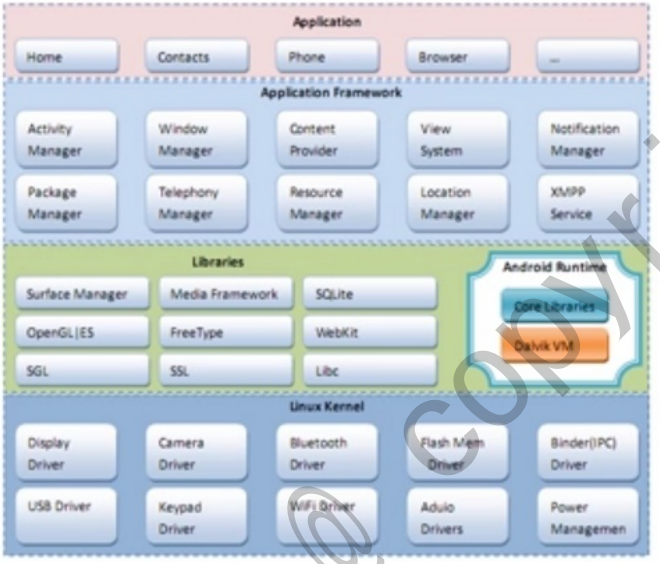
- 每一层可以访问下层，不能访问上层
- 封闭式结构：每一层只能访问与其相邻的下一层
- 开放式结构：每一层还可以访问下面更低的层次
- 层次数目不应超过 7±2 层

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。

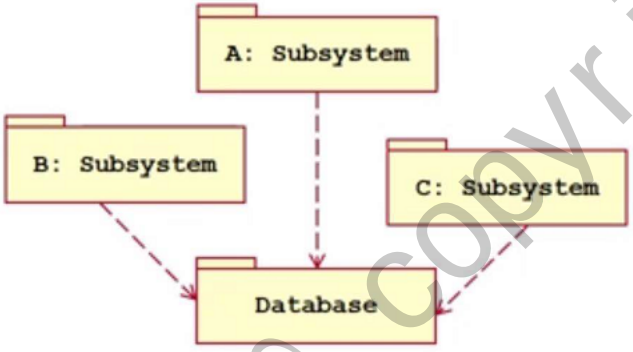
应用框架层：支持第三方开发者之间的交互，使其能够通过抽象方式访问所开发的应用程序需要的关键资源。

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

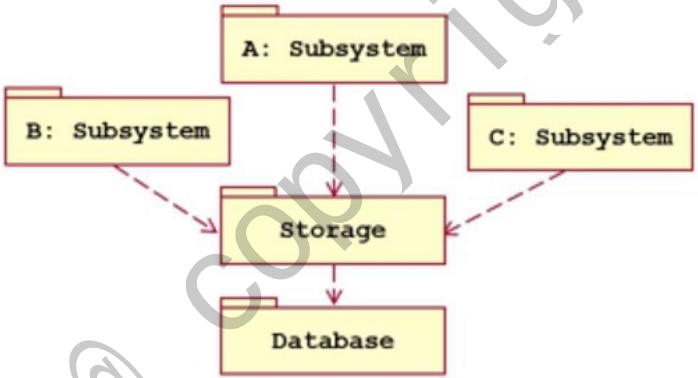
举例：在一个软件系统中，有三个子系统A、B、C都要访问一个关系数据库。



这个方案有什么问题？

为什么需要层次结构

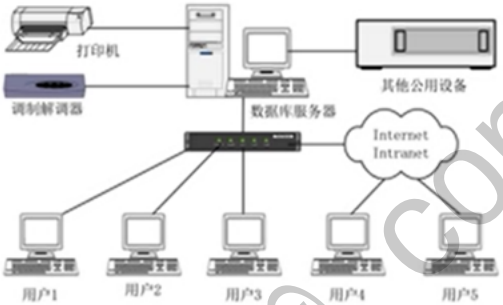
改进：在子系统和数据库之间增加一个存储子系统Storage，从而屏蔽了底层数据库的变化对上层子系统的影响。

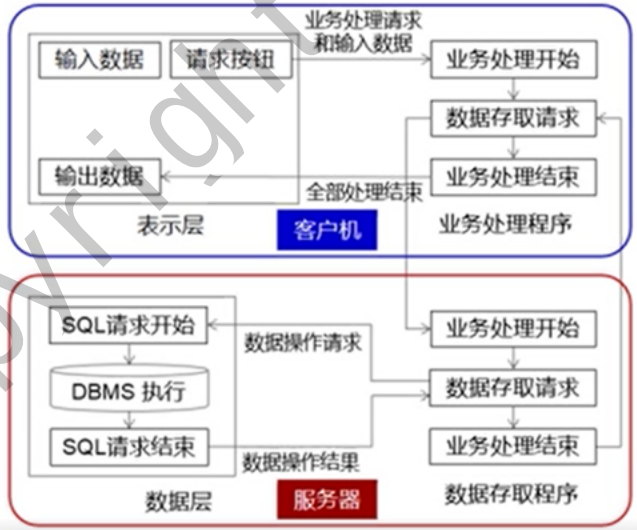


两层的C/S架构

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



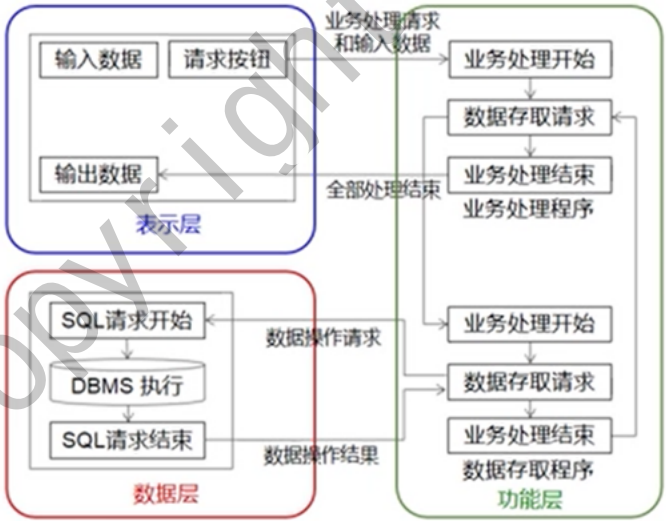


内容来源：

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。

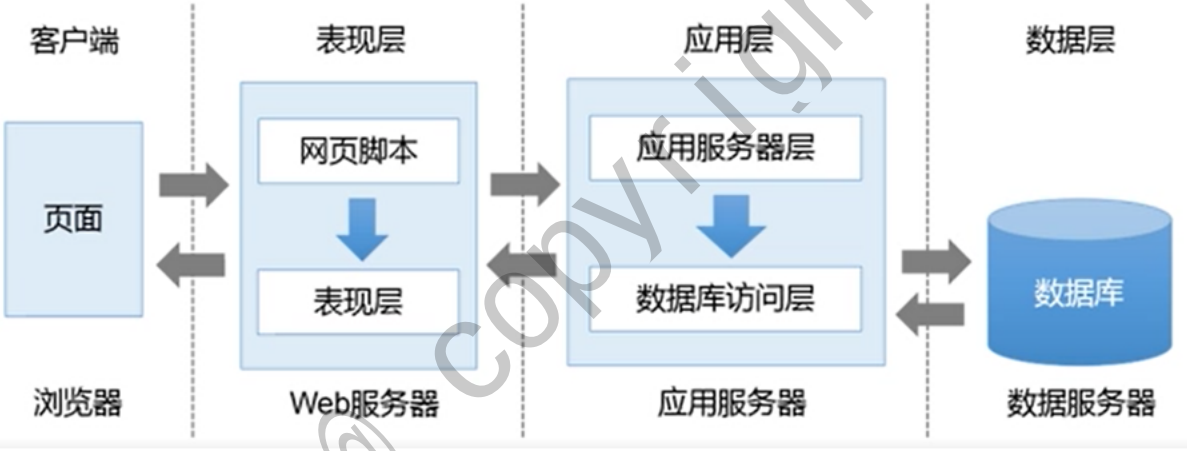


内容来源：

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。

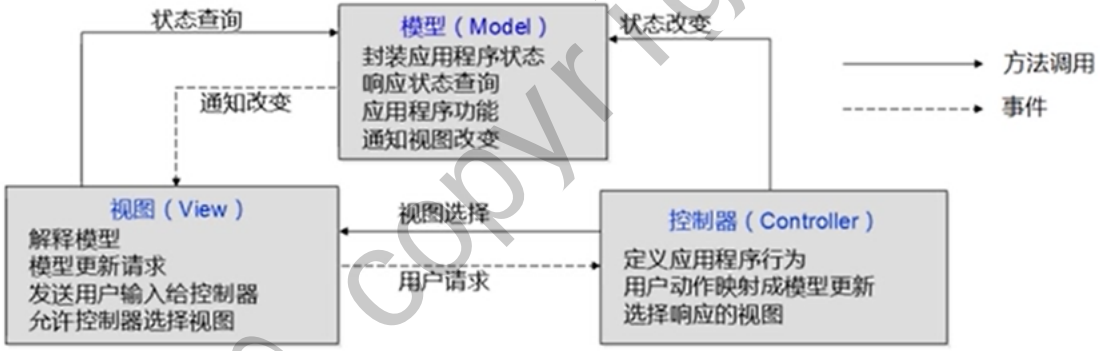


内容来源：

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

MVC三层结构

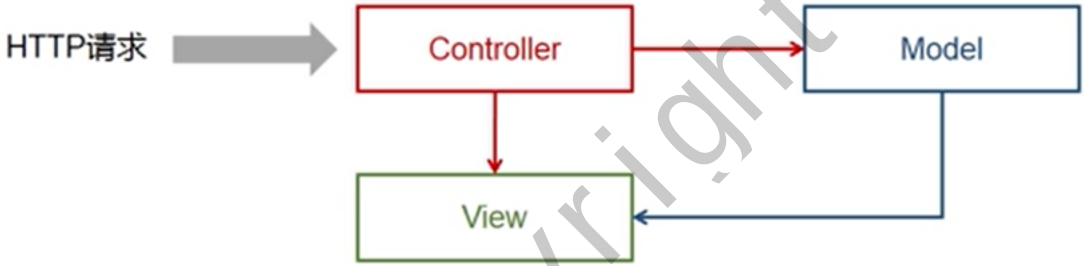
模型-视图-控制器（MVC）结构将应用程序的数据模型、业务逻辑和用户界面分别放在独立构件中，这样对用户界面的修改不会对数据模型/业务逻辑造成太大影响。



内容来源：

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

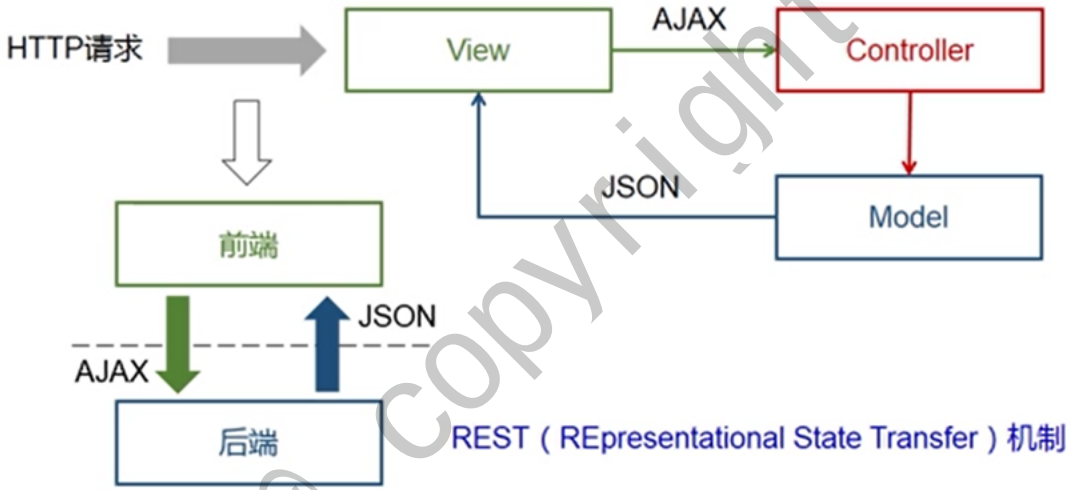
MVC结构的请求访问路径



- 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）

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

课后思考题

如何识别高耦合和低内聚？