

第六章

信息系统实施

(Implement of the Information System)

[返回总目录](#)

教学目的

- 系统实施是在系统设计的基础上将新系统方案在计算机上实现，要求学生功能按系统设计方案购置和安装设备
- 能够建立数据库系统
- 能够编制程序
- 进行系统的测试与调试

教学要求

- 掌握系统**实施**的任务
- 了解系统实施计划包括的内容
- 掌握程序设计的**基本方法**
- 掌握系统**测试**的步骤与方法
- 了解**系统测试报告**包括的内容；具备进行系统测试的能力
- 掌握系统**转换**的方式和优缺点

信息系统实施

- ❑ 系统实施概述
- ❑ 程序设计
- ❑ 信息系统的测试
- ❑ 系统调试
- ❑ 系统转换
- ❑ 小结



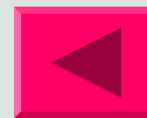
信息系统实施

第一节 系统实施概述



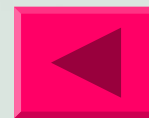
系统实施概述

- ❖ 系统实施的任务
- ❖ 系统实施的内容和流程
- ❖ 系统实施的计划安排



系统实施的任务

- 设备的购置与安装
- 程序的编制与测试
- 数据的录入
- 人员的培训
- 系统的测试、调试与转换



系统硬件平台的实施

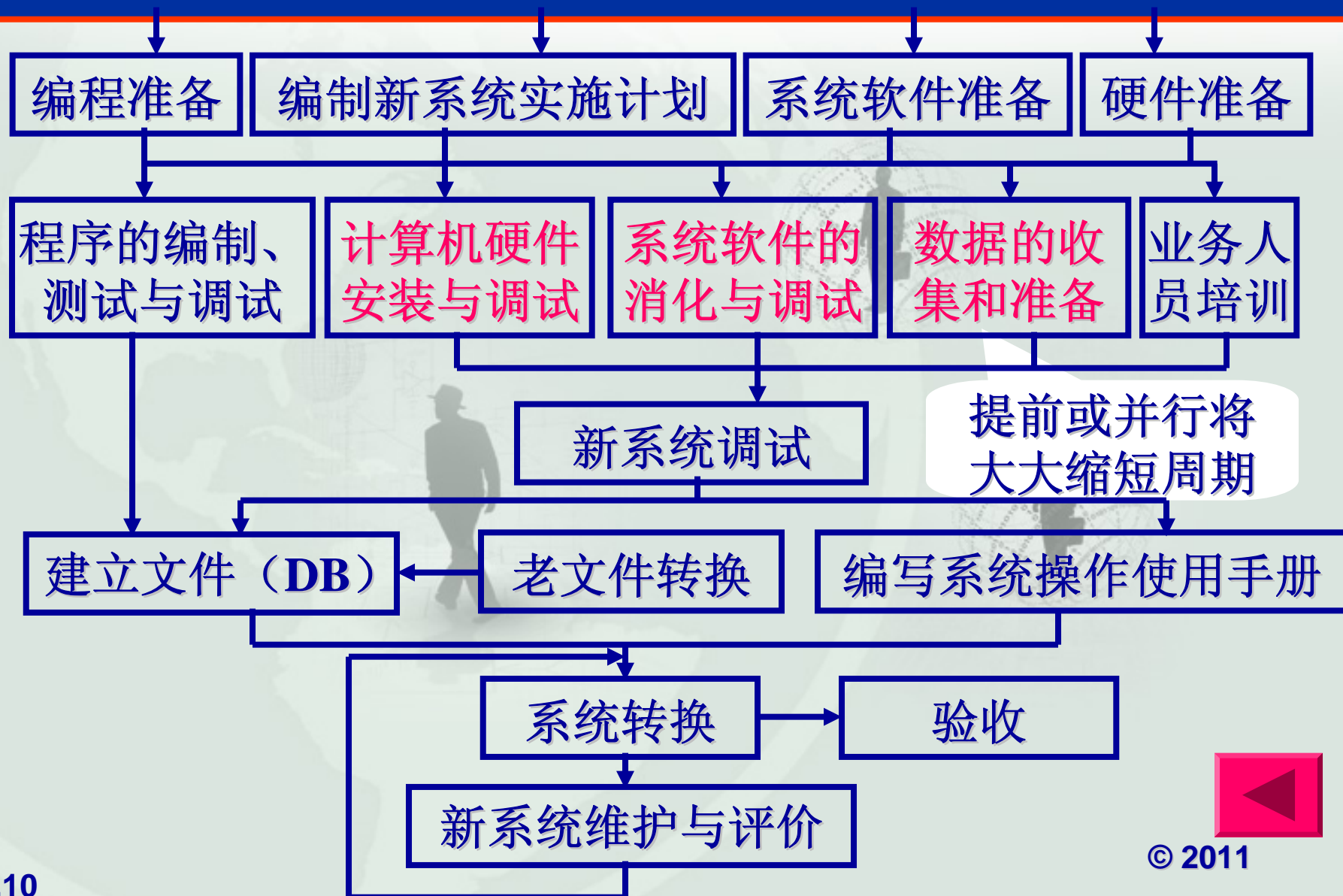
- 计算机系统采购
 - 性价比
 - 系统可扩充性
 - 售后服务和技术支持
- 网络系统
 - 网络通信设备采购安装
 - 电缆线的铺设
 - 网络性能调试

应用程序的购买

- 功能范围及功能实现质量
 - 输入部分
 - 输出部分
 - 处理方法
- 效率
- 灵活性
- 可靠性
- 使用方便性
- 技术支持

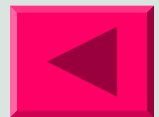


系统实施的内容及流程



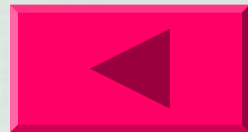
系统实施的计划安排

- 工作量估计
- 实施进度安排
- 系统人员的配备和培训计划
- 系统实施的资金筹措和投入计划



系统实施

第二节 程序设计



程序设计

- ❖ 程序语言的选择
- ❖ 程序设计的基本要求
- ❖ 程序设计的基本方法
- ❖ 结构化程序设计的基本特点



程序设计语言的选择

- 应用领域--选择语言的关键因素
- 算法和计算的复杂性
- 软件的运行环境
- 各种性能的考虑
- 数据结构的复杂性
- 程序设计人员的知识水平



应用程序编制语言的选择

- 应用领域
- 算法和计算的复杂性
- 软件的执行环境
- 性能
- 数据结构的复杂性
- 程序员的技术水平

第一代语言

二进制机器语言

第二代语言

符号汇编语言

第三代语言

高级语言 COBOL,FORTRAN等

第四代语言

4GL: 突破冯·诺依曼结构

第五代语言

专家系统、知识库、推理、自然语言处理。Lisp, PROLOG

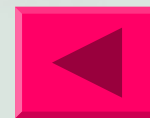
应用程序设计目标

- 可维护性
- 可靠性
- 可理解性（逻辑清晰、易读易懂）
- 效率



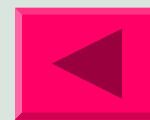
程序设计的基本要求

- 程序内部文档化的要求
- 数据说明要求
- 语句构造要求
- 输入输出要求
- 程序运行要求



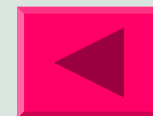
程序设计语句构造要求

- 不要为了节省空间而把多个语句写在同一行上
- 尽量避免复杂的条件判断测试
- 尽量减少对“非”条件的测试
- 尽量少使用循环嵌套和条件嵌套
- 尽量利用括号，可使逻辑表达式或算术表达式的运算次序清晰直观



输入输出要求

- 输入数据要有完善的检验措施
- 输入格式设计有简单、直观、布局合理
- 明确提示交互输入请求，详细说明可用的选择及边界数据
- 输出报表要易读、易懂，符合使用者的要求的习惯



程序运行要求

- 编程前要优化算法
- 仔细研究循环条件及嵌套循环，检查是否语句从内向外移
- 尽量避免使用多维数组
- 尽量避免使用指针和复杂的数据结构
- 不要混合使用不同的数据类型
- 对I/O效率, 存储器运行效率等应考虑

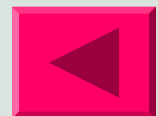


程序设计的基本方法

➤ 自顶向下的模块化设计

➤ 逐步求精

把一个模块的功能一步步地分解成一组子功能，而这组子功能可以通过执行若干个程序步来完成该模块的全部功能



程序设计的基本方法——自顶向下的模块化设计

- ✓ 自顶向下的扩展原则在不同阶段的用法和含义
- ✓ 自顶向下的程序设计原则
- ✓ 层次模块图



自顶向下的扩展原则在不同阶段的用法和含义

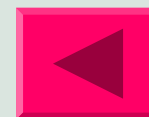
- 系统分析阶段
- 系统设计阶段
- 程序设计阶段



自顶向下的扩展原则在不同阶段的用法和含义

在系统分析阶段

- 在画数据流程图时，先画高层的数据流程图
- 对高层数据流程图中的处理逻辑进行逐层向下扩展
- 在同一张数据流程图中所有的处理逻辑都处于平等的地位



自顶向下的扩展原则在不同阶段的用法和含义

系统设计阶段

- 在画结构图时，先画高层的结构图
- 对高层结构图中的模块进行逐层向下扩展
- 在同一张结构图中高层模块调用下层模块，存在调用和被调用关系



自顶向下的扩展原则在不同阶段的用法和含义

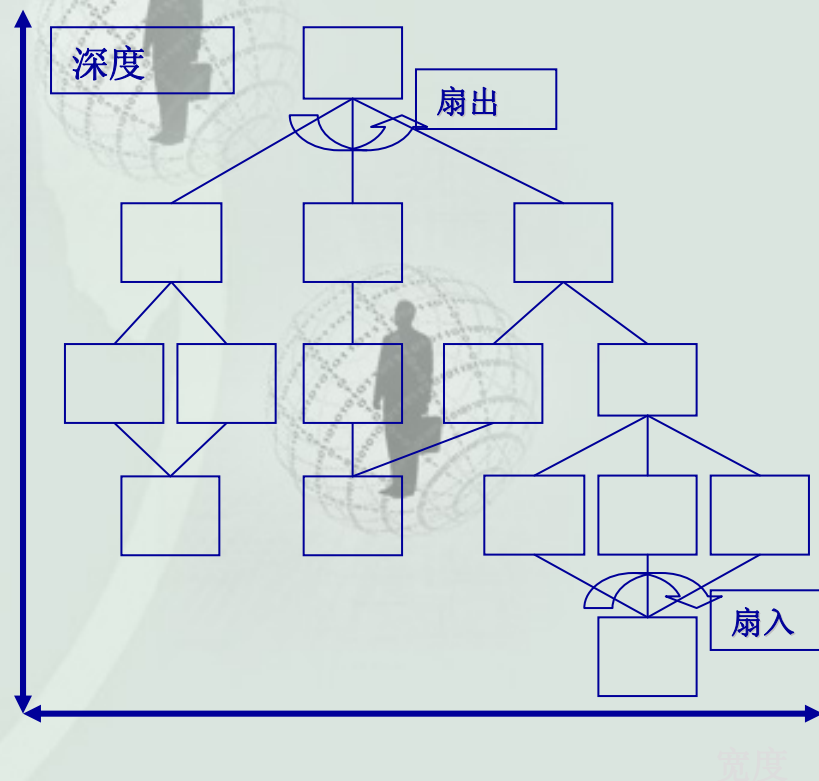
- 先把程序高度概括
- 对下层模块进行逐层向下扩展
- 对扩展出来的下层模块，反复进行修改
- 层次模块图反映程序的功能以及在这个程序中各个模块之间的关系



结构化程序设计方法（一）

1. 自顶向下的模块化设计（ TOP-DOWN）

- 模块的独立性
- 模块大小的划分
- 模块功能简单化
- 模块的集中调用



模块间联接形式的比较

联接形式	对联接反应的影响	可修改性	可读性	通用性
数据联接	弱	好	好	好
控制联接	中	中	中	中
公共联接	强	不好	不好	不好
内容联接	最强	最坏	最坏	最坏

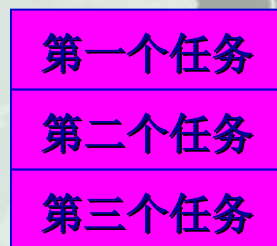
结构化程序设计方法（二）

2. 结构化程序设计方法

A. 顺序结构

B. 循环结构

C. 选择结构



A



B

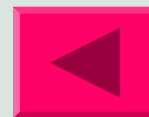


C



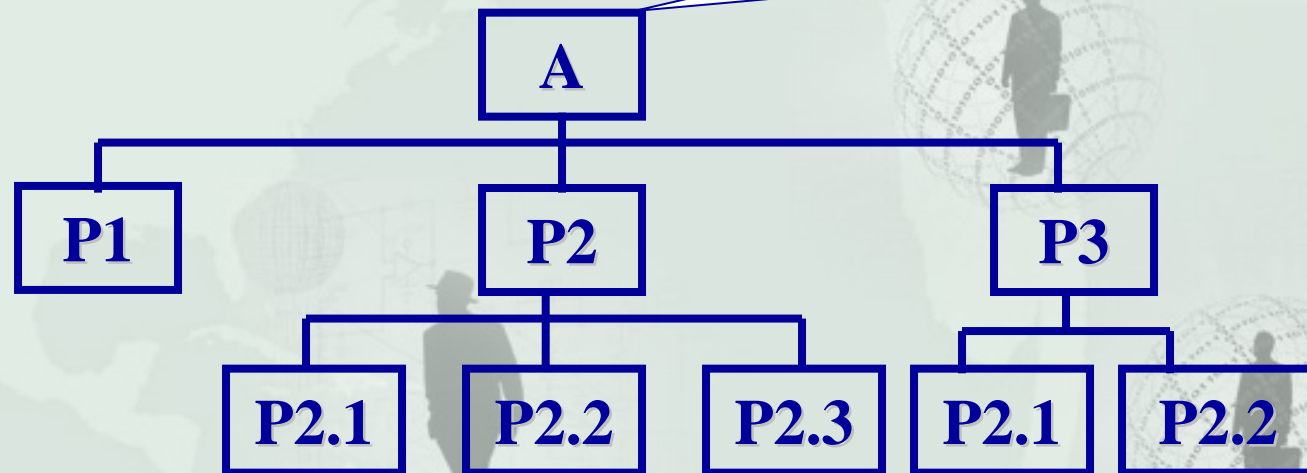
自顶向下的模块设计原则

- 先把程序高度概括，看作是一个最简单的控制结构，即
- 为了完成这个功能，需要进一步分解成若干个较低一层的模块，每一个下层模块都有一个名称，表达了一个较小的功能对扩展出来的每一个下层模块
- 反复运用自顶向下程序设计中的第二条原则，逐层扩展，直到最低一层每一个模块都非常简单、功能很小，能够很容易地用程序语句实现为止。



层次模块图

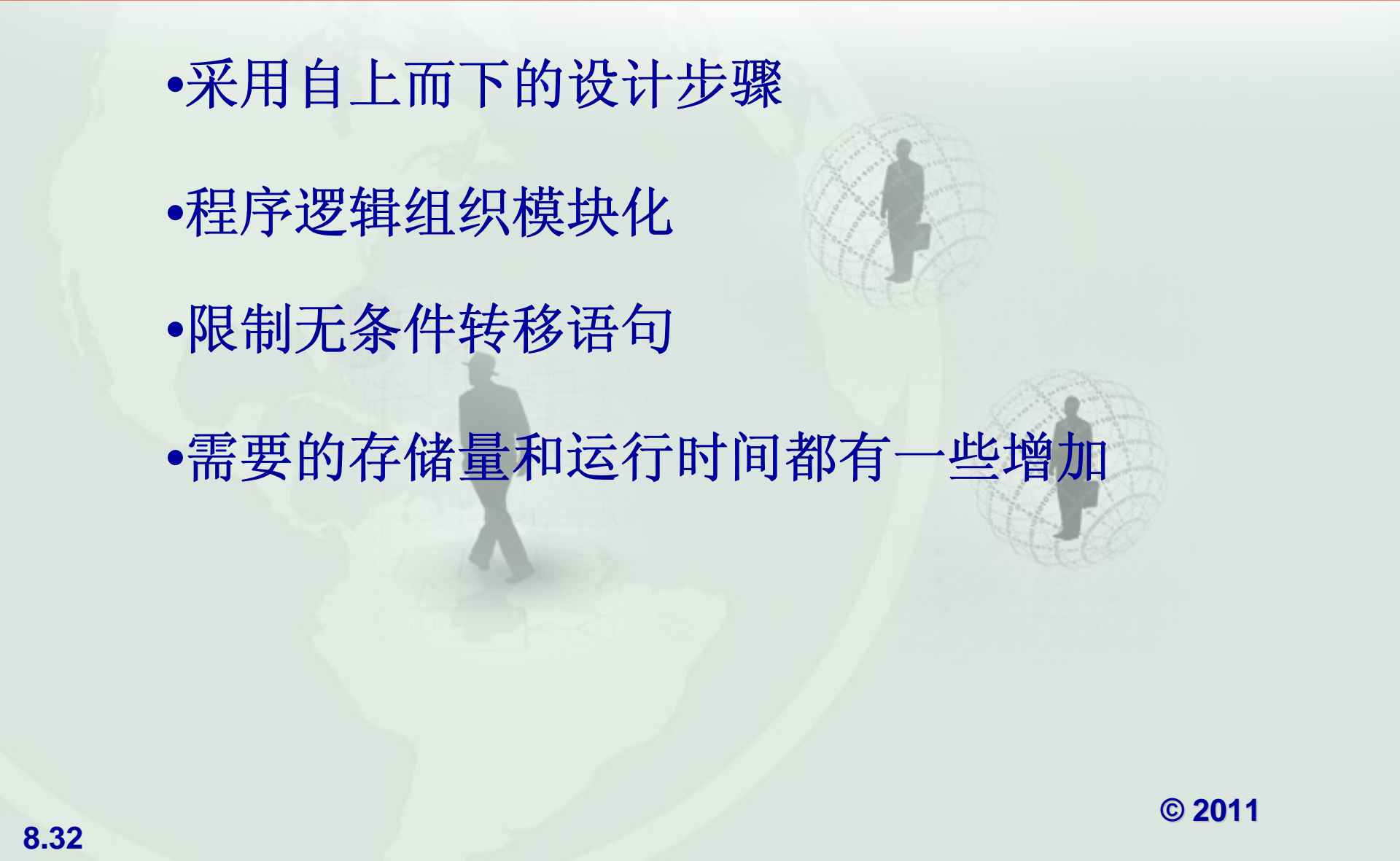
模块相当于一个基本控制结构
是一个子程序或一个程序段



结构图中的模块至少是一个程序或一组程序



程序设计的基本特点

- 采用自上而下的设计步骤
 - 程序逻辑组织模块化
 - 限制无条件转移语句
 - 需要的存储量和运行时间都有一些增加
- 

软件开发工具

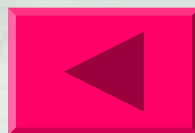
- 电子表格软件
- 设计库管理软件
- 套装软件
- 可视化编程工具
- 计算机辅助软件工程 (**CASE**)



信息系统实施

第三节

信息系统的测试



教学目的

- 学习系统测试的意义、目的和基本原则
- 系统测试的方法和过程
- 系统测试的步骤
- 系统测试方案的设计
- 系统调试

教学要求

1. 熟练掌握：

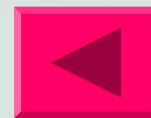
- ◆ 系统测试的基本原则；
- ◆ 系统测试的步骤包括单元测试、集成测试和系统测试的相关知识
- ◆ 系统测试方案的设计包括白盒（重点在逻辑覆盖和基本路径法）和黑盒（重点在等价类、边界值法和错误推测法）测试方法
- ◆ 学会综合使用各种测试技术

2. 一般掌握：

- ◆ 系统测试的意义、目的；
- ◆ 系统测试的方法和过程；
- ◆ 系统调试的相关内容

信息系统的测试

- ❖ 系统测试概述
- ❖ 软件测试方法
- ❖ 动态测试用例的设计
- ❖ 动态测试策略
- ❖ 软件测试步骤
- ❖ 软件正确性证明



信息系统测试概述

- 系统测试概述
- 软件测试的时间
- 软件测试的原则
- 软件错误
- 软件测试任务
- 软件测试的基本手段



系统测试概述

测试：就是为了发现程序中的错误而执行程序的过程

- 测试应该把查出新错误的测试看作是成功的测试
- 没有发现错误的测试是失败的测试

系统测试概述

Grenford J.Myers就系统测试的目的提出下列观点

- (1) 测试是程序的执行过程，目的在于发现错误；
 - (2) 一个好的测试用例在于能发现至今未发现的错误；
 - (3) 一个成功的测试是发现了至今未发现的错误的测试
- ✓ 设计测试的目标是想以最少的时间和人力系统地找出系统中潜在的各种错误和缺陷

系统测试概述

- 发现错误不是目的，目的是开发出高质量的完全符合用户需要的软件
 - 测试发现的错误还必须诊断并改正错误
- 

信息系统测试概述

- 测试至少占据了制作过程的一半工作量

软件测试的工作量往往占软件总工作量的40%以上。
在极端的情况下，测试关系人的生命安全的软件所花的成本可能相当于软件工程其他步骤总成本的 3~5倍

- 一般程序员很少喜欢测试，更不喜欢进行测试设计

如果测试设计和测试工作量比程序设计和编程调试的工作量大，则更少有程序员喜欢

- 测试是系统开发中的一个重要环节，是成功开发信息系统的重要保证。



软件测试的时间

软件测试在软件生命周期中横跨两个阶段

(1) 单元测试

- 单元测试和编码属于软件生命的同一个阶段，通常在写出每个模块之后，就对它做必要的测试
- 模块的编写者和测试者是同一个人

(2) 综合测试

- 综合测试在程序全部完成之后进行
- 由专门的测试人员承担

软件测试原则

- 确定预期输出（或结果）是测试情况必不可少的一部分
- 程序员应避免测试自己的程序
- 程序设计机构不应该测试自己的程序



软件测试原则

- 测试用例的设计和选择、预期结果的定义要有利于错误的检测
- 要严格执行测试计划、排除测试的随意性

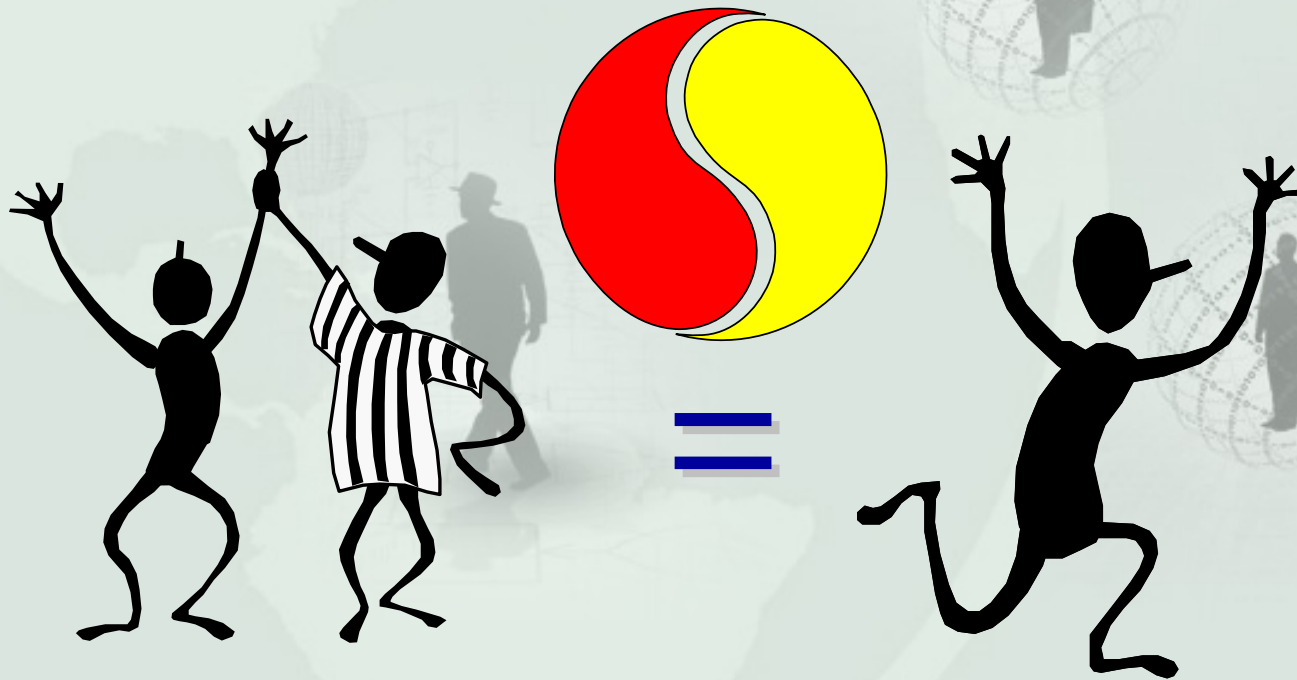
- 测试的目的
- 环境、机时
- 系统集成方式
- 排错规程
- 测试用例标准，工具
- 完成标准，进度，岗位责任
- 跟踪规程
- 回归测试的规定等

软件测试原则

- 要将软件测试贯穿于软件开发的整个过程，以便尽可能地发现错误，从而减少由于错误带来的损失
- 软件测试不仅要检查程序是否做了应该做的事情，还要检查它是否做了不应该做的事情

软件测试原则

- 经验表明：程序中尚未发现的错误的数量与该程序段已发现的错误数量往往成正比



软件错误

Neson将错误和缺陷概括为七个方面：

1. 编程时的语法错误

- 保留字拼写错误
- 循环体不匹配
- 参数与变元不匹配
- 程序员发现在用某些解释性程序设计语言（如VB，VFP等）编程时检查这类错误容易而且及时

软件错误

2. 程序员对语言结果误解所造成的错误
 - 对循环体结构的误解
3. 算法或逻辑上的错误
4. 近似算法会使某些输入变量得不到精确的甚至错误的结果
5. 由于错误的输入导致程序的错误

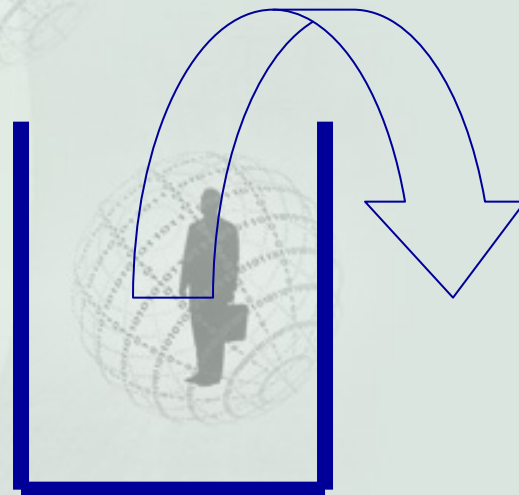


软件错误

6. 数据结构说明不当或实现中的缺陷所造成的错误

过小的栈容量造成

- 栈操作的上溢
- 栈操作的下溢



软件错误

7. 由于系统(或模块)说明书的缺陷所造成的错误

此类为最严重的错误



测试任务

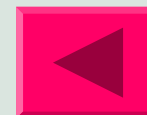
- 预防软件发生错误
- 发现并改正程序错误
- 提供错误诊断信息



软件测试的基本手段

人工测试

计算机测试



人工测试——静态测试

人工测试指被测程序不在机器上运行

可以由编写者本人非正式地进行，也可以由
审查小组正式地进行。

人工测试技术有：

- ◆ 程序审查 (Code Inspections)
- ◆ 人工运行(Walkthroughs)
- ◆ 静态检查(Desk Checking)



计算机测试——动态测试

准备一些测试程序在计算机上运行，
以此来查找程序错误

计算机测试要遵循的步骤：

- (1) 设计测试情况
- (2) 进行模块测试
- (3) 进行高级测试



软件测试方法

动态测试法

静态测试法



动态测试方法



黑盒测试

白盒测试

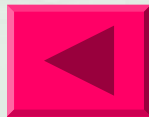
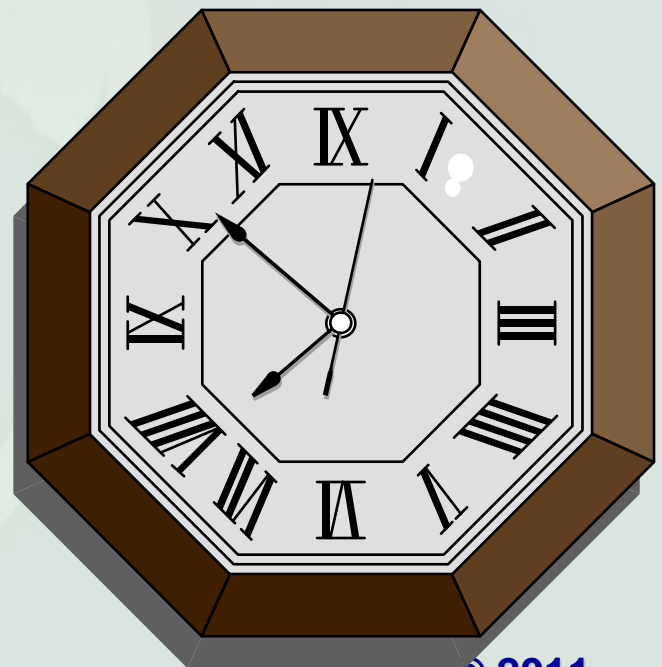
黑盒测试(功能测试)方法(Block-box Testing)

- 黑盒测试方法
- 黑盒测试方法的原理
- 穷举测试
- 黑盒测试使用的数据



黑盒测试方法

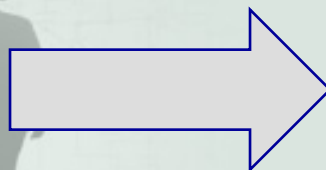
- 已经知道了产品应该具有的功能
- 通过测试检验是否每个功能都能正常使用



黑盒测试方法工作原理

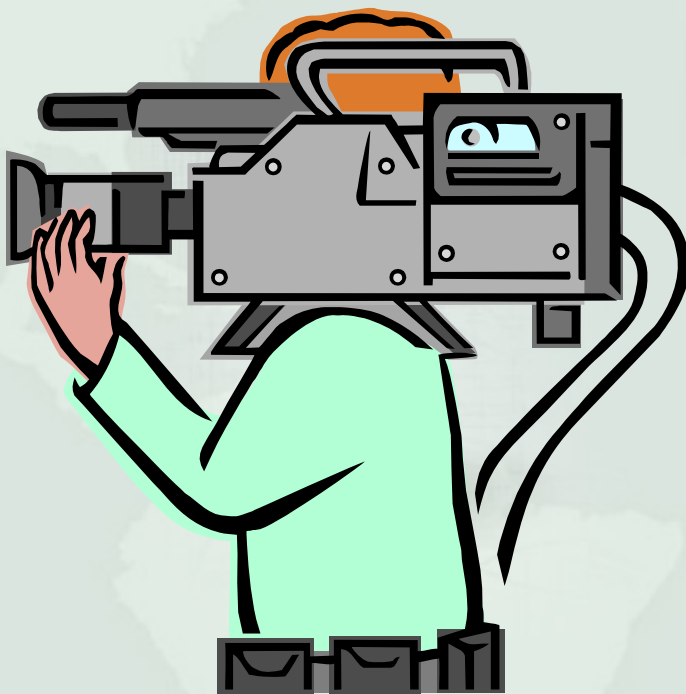
- 把程序看成一个黑盒子

程序



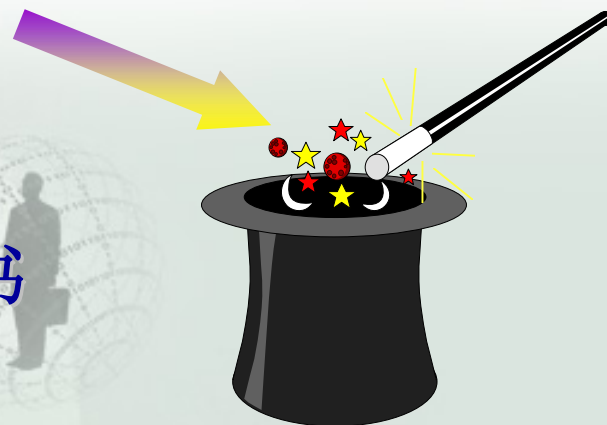
黑盒测试方法工作原理

- 完全不考虑程序的内部结构和处理过程



黑盒测试方法工作原理

- 在接口进行测试
- 检查程序功能是否按规格说明书的规定正常使用



正常使用



黑盒测试方法工作原理

- 程序是否适当地接收输入数据产生正确的输出数据
- 保持外部信息的完整性



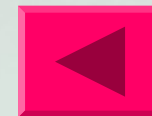
黑盒测试的穷尽输入测试

- 至少必须对所有输入数据的各种可能值的排列组合都进行测试
- 例一个程序需要两个整数型的输入数据
- 如果计算机字长是32位，则每个整数可能取的值有 2^{32} 个
- 二个数的可能排列组合是 $2^{32} * 2^{32}$ (2^{64} 种)
- 假设每执行一次程序需要1毫秒，则需5亿年



黑盒测试使用的数据

- 程序有效的输入数据
- 程序无效的输入数据
- 极端的数据元素
- 正常的数据元素
- 特殊的数据元素



白盒测试(结构测试)方法(White-box Testing)

- 白盒测试方法
- 白盒测试方法的原理
- 穷尽测试

白盒测试



白盒测试(结构测试或逻辑覆盖法)方法

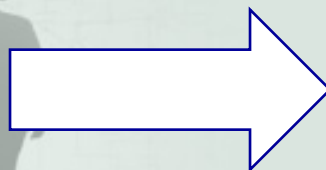
- 把测试对象看作一个打开的盒子
- 测试人员须了解程序的内部结构和处理过程，以检查处理过程的细节为基础
- 对程序中尽可能多的逻辑路径进行测试
- 检验内部控制结构和数据结构是否有错，实际的运行状态与预期的状态是否一致
- 通过测试检验来检验产品内部动作是否按照规格说明书的规定正常进行



白盒测试方法工作原理

- 把程序看成装在一个透明的白盒子里

程序

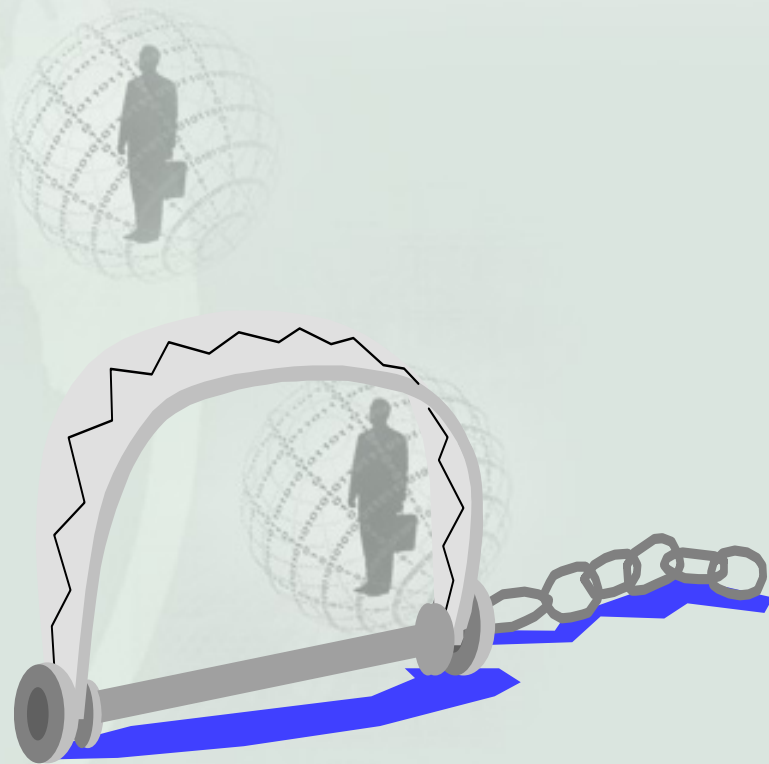
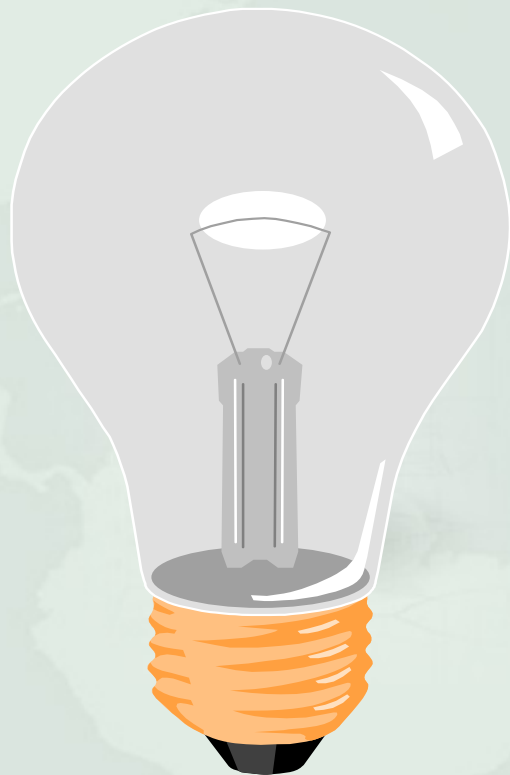


白盒



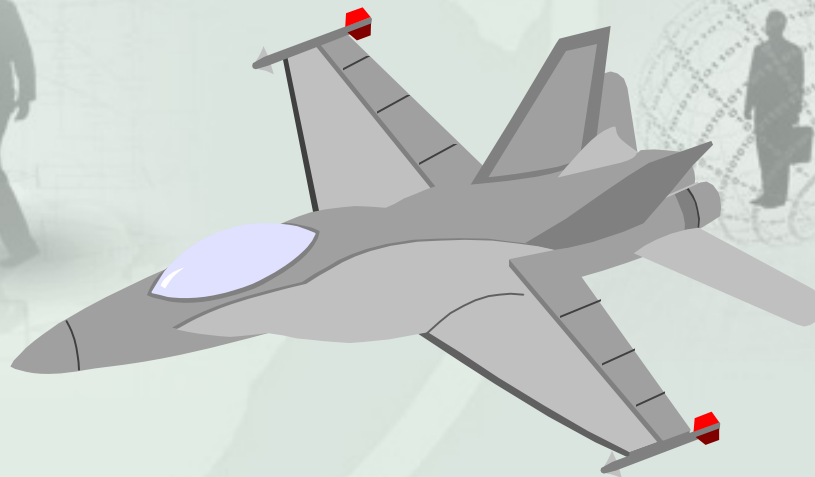
白盒测试方法工作原理

- 完全了解程序的内部结构和处理过程



白盒测试方法工作原理

- 按照程序内部的逻辑测试程序
- 检验程序中的每一条通路是否都能按预定的要求正常工作

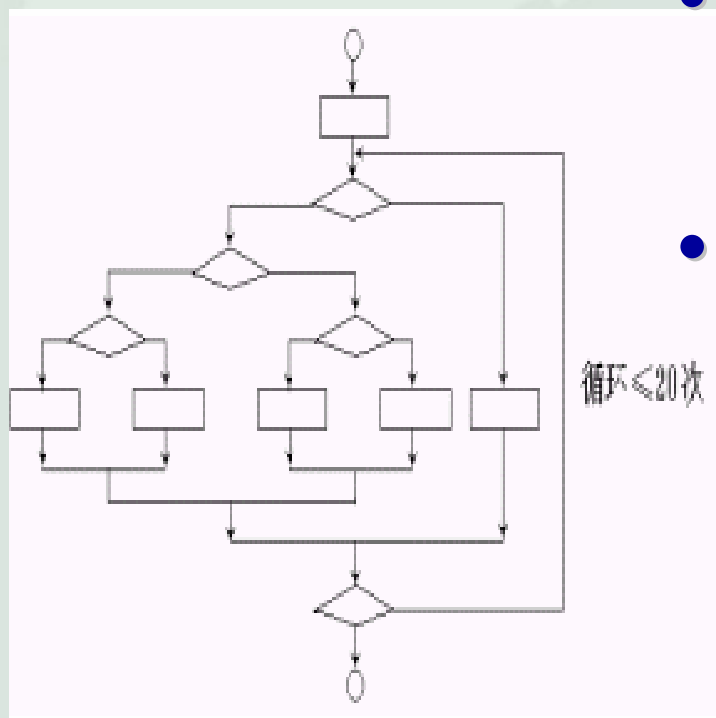


白盒测试的穷尽路径测试

- 至少必须对所有路径都进行测试
- 即使可以穷举出所有的路径，但是若程序少写了一个路径，则查不出错误

白盒测试的穷尽路径测试

例：一段具有多重选择和循环嵌套的程序，循环次数为**20**次。



- 它包含的不同执行路径数达 5^{20} ($\approx 9.54 \times 10^{13}$) 条
- 假设测试一条路径需要**1ms**，假定一天工作**24h**，一年工作**365**天，那么想把所有路径测试完，需**3024**年。

静态测试方法

➤ 静态测试法概述

➤ 静态测试法种类



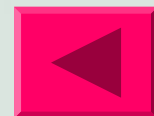
静态测试法

- 不涉及程序的实际执行
- 以人工的，非形式化的方法对程序进行分析和测试
- 可检出大约30%~70%的逻辑设计错误
- 该方法的成本较低



静态测试法——人工测试

- 程序审查会——代码会审
- 桌前检查 (Desk Checking)--静态检查
- 人工运行



程序审查会

- 由一组人员通过阅读、讨论和争议，对程序进行静态分析的过程
- 需要的材料：
 - ☆ 待审程序文档
 - ☆ 控制流程图
 - ☆ 有关要求规范

程序审查会工作过程

- 会议前把要审查的程序清单和设计规范分发给小组的其他成员
- 请程序员讲述程序的逻辑结构
- 根据常见程序错误检验单分析程序

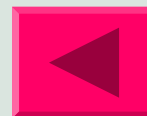
错误检验单中的项目

- ☆数据引用错误
- ☆数据说明错误
- ☆计算错误
- ☆比较错误
- ☆控制流程错误
- ☆接口错误
- ☆输入/输出错误
- ☆其他检验



桌前检查——静态检查

- 由程序员反复阅读编码和流程图，对照模块功能说明、算法、语法规则检查程序的语法错误和逻辑错误
- 可设计少量测试实例，由人工来模拟计算机单步执行并观察执行过程的结果



动态测试用例设计



白盒测试
用例设计



黑盒测试
用例设计



白盒测试--逻辑覆盖测试的种类

- 语句覆盖
- 判定覆盖
- 条件覆盖
- 判定/条件覆盖
- 多重条件覆盖
- 实例



控制流程图及被测试程序

PROCEDURE prol

parameter A,B,X

if(A>1).AND.(B=0)

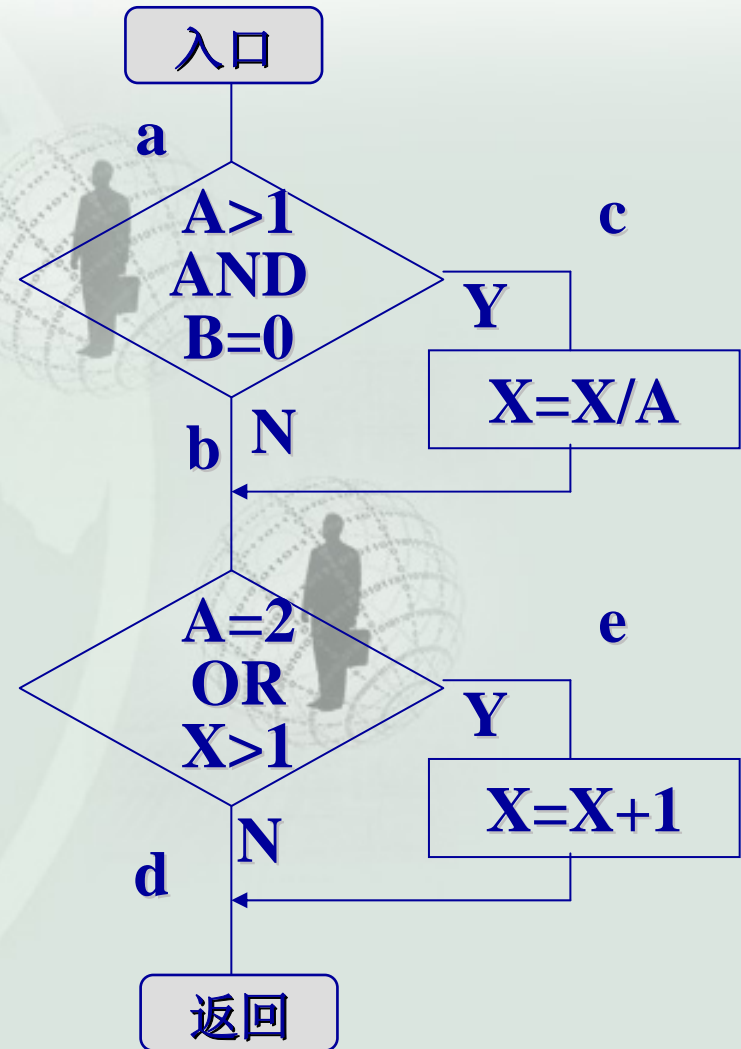
X=X/A

endif

if(A=2).OR.(X>1)

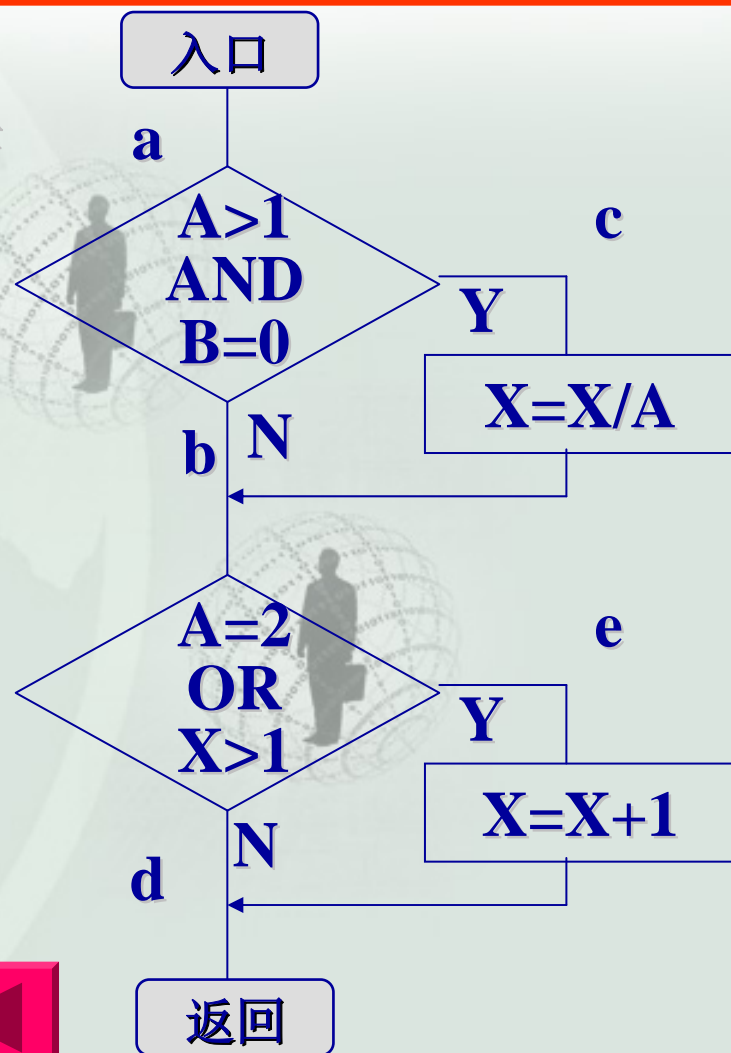
X=X+1

endif



语句覆盖

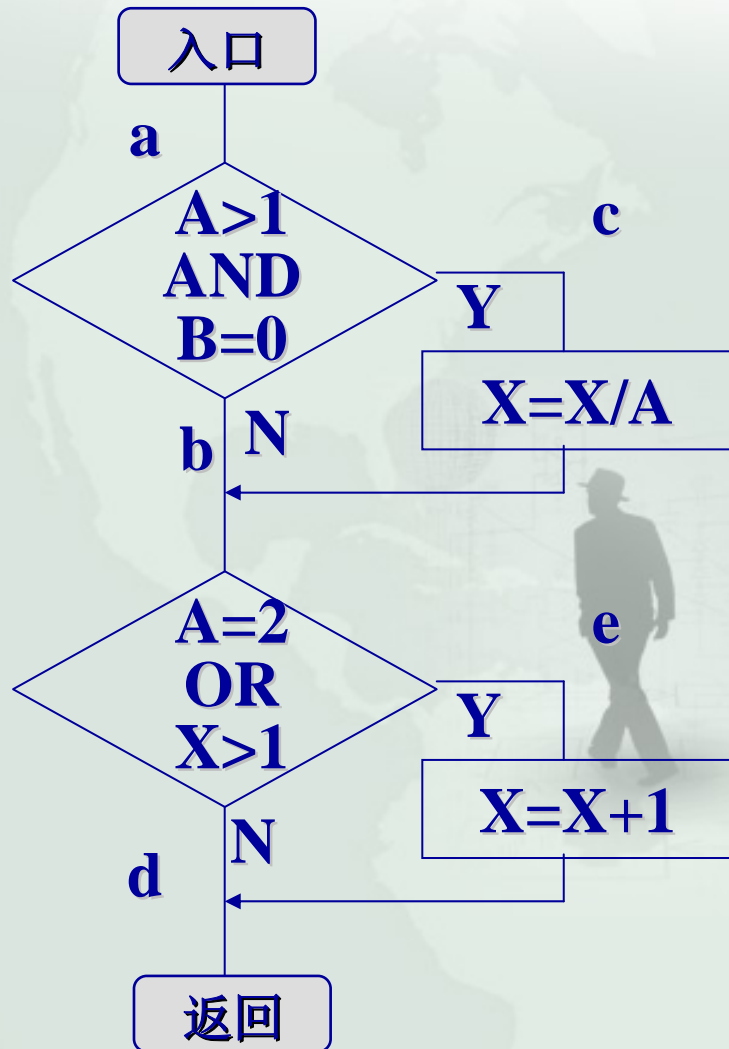
- 编写足够的测试情况，使得每条语句至少执行一次
- 编写一个通过路径ace的单个测试情况
- 在a点 $A=2$; $B=0$ 和 $X=3$



判定覆盖——分支覆盖

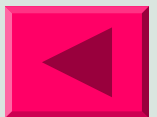
- 编写足够的测试情况，使得每个判定至少有一次“真”和一次“假”的结果
- 每个分支方向都必须至少经过一次
- 要在程序或子程序的每个入口点至少进入一次

判定覆盖



例：ace和abd或acd和abe都可满足判定覆盖

如果选择路径acd及abe，则
 $A=3, B=0, X=1$ (acd) 和
 $A=2, B=1, X=1$ (abe)



条件覆盖

- 编写足够的测试情况，使得判定中每个条件的所有可能结果至少出现一次
- 要在程序或子程序的每个入口点至少进入一次

条件覆盖

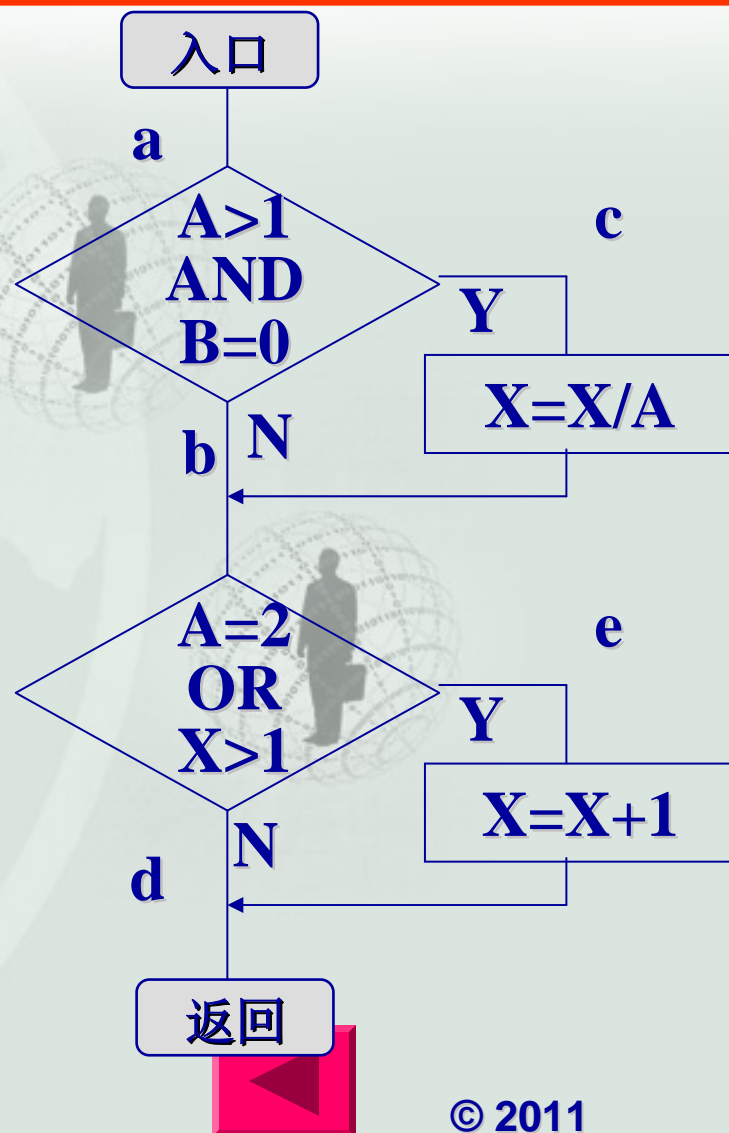
- 有4个条件: $A > 1, B = 0, A = 2$ 和 $X > 1$

- 需要有足够的测试情况以形成:
在a点出现 $A > 1, A \leq 1, B = 0, B \neq 0$




- 在b点出现 $A = 2, A \neq 2, X > 1, X \leq 1$

- $A = 2, B = 0, X = 1$, 路径ace

- $A = 1, B = 1, X = 2$, 路径abe



判定/条件覆盖

- 编写足够的测试情况，使得判定中每个条件的所有可能结果至少出现一次
 - 每个判定本身所有可能结果也至少出现一次
 - 同时每个入口点至少要进入一次
- 
- 
- 

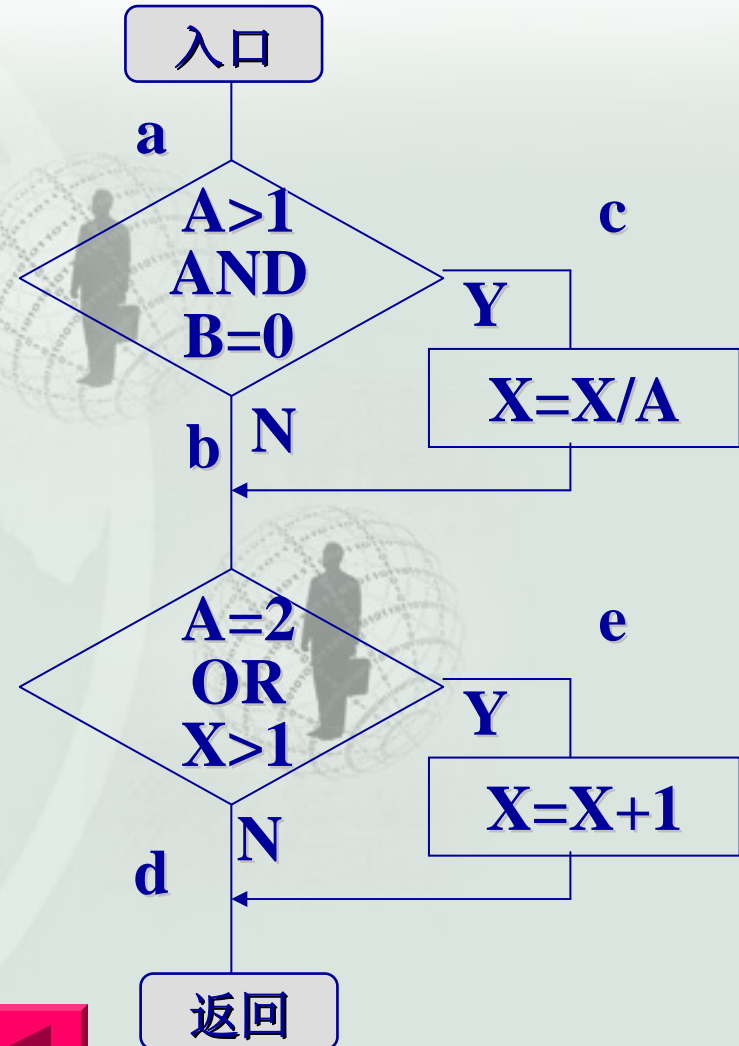
判定/条件覆盖

•有4个条件： $A > 1$, $B = 0$, $A = 2$ 和 $X > 1$

•ace和abd或acd和abe都可满足判定覆盖

• $A = 2, B = 0, x = 4$

• $A = 1, B = 1, X = 1$



多重条件覆盖

- 要写出足够的测试情况，以使判定的每条分支至少通过一次
- 编写足够的测试情况，使得每个判定判定中条件结果的所有可能组合至少出现一次
- 所有的入口点都至少进入一次

多重条件覆盖

•有8个条件:

① $A > 1, B = 0$

② $A > 1, B \neq 0$

③ $A \leq 1, B = 0$

④ $A \leq 1, B \neq 0$

⑤ $A = 2, X > 1$

⑥ $A = 2, X \leq 1$

⑦ $A \neq 2, X > 1$

⑧ $A \neq 2, X \leq 1$

• $A = 2, B = 0, X = 4$

覆盖①, ⑤

• $A = 2, B = 1, X = 1,$

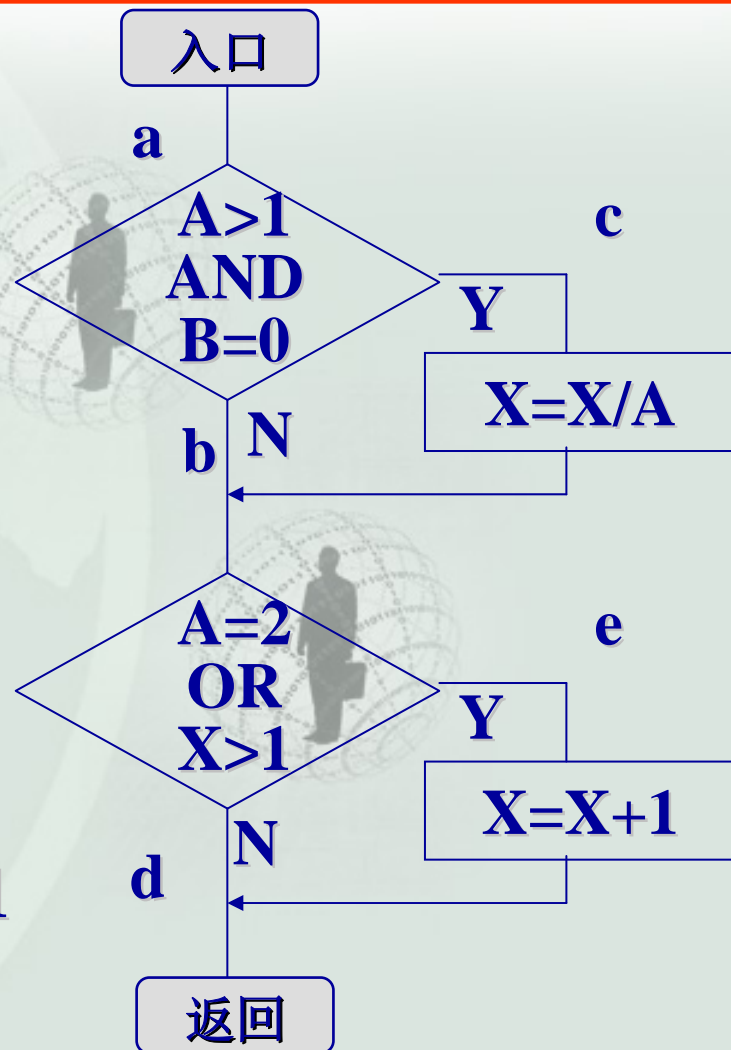
覆盖②, ⑥

• $A = 1, B = 0, X = 2$

覆盖③, ⑦

• $A = 1, B = 1, X = 1$

覆盖④, ⑧



实例

假如有如下一条语句

IF (X>0)AND(Y≠0)

THEN S1

ELSE S2

ENDIF

实例

- 满足判定覆盖标准，但不满足条件覆盖标准

$$S = \{(1,1), (1,0)\}$$

- 满足条件覆盖标准，但不满足判定覆盖标准

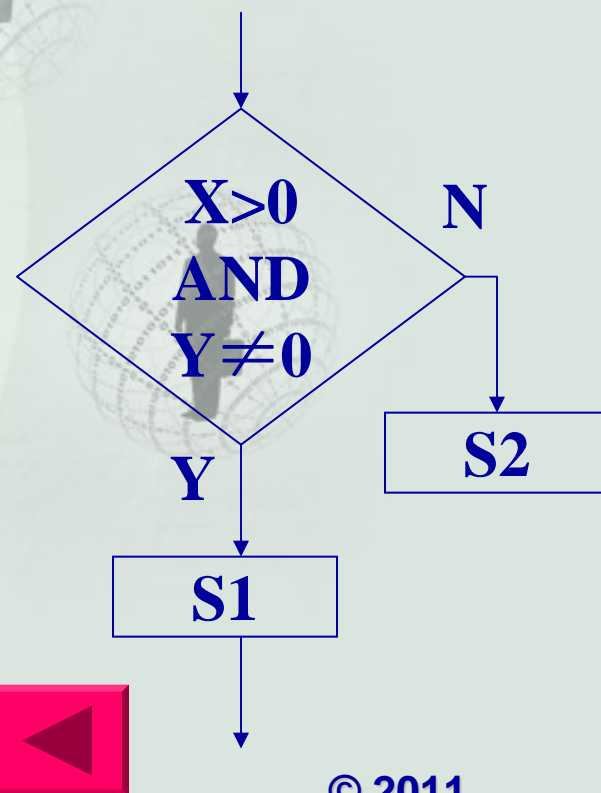
$$S = \{(0,1), (1,0)\}$$

- 满足判定/条件覆盖标准

$$S = \{(0,0), (1,1)\}$$

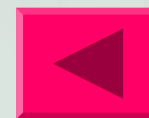
- 满足组合条件覆盖标准

$$S = \{(1,1), (1,0), (0,1), (0,0)\}$$



黑盒测试的种类

- 等价类法
- 边值分析法
- 因果图法
- 错误推测法



等价类划分

- 等价类划分的原理
- 等价类划分进行测试情况设计的步骤
- 实例（教材P214）



等价类划分的原理

- 根据程序的输入/输出特性，将程序的输入划分为有限个等价区段
- 对每一个输入条件存在着程序有效的有效等价类
- 对每个输入条件存在着对程序错误输入的无效等价类
- 从每个区段内抽取的代表性数据进行的测试等价于该区段内任何数据的测试



等价类划分测试情况设计步骤

• 确定等价类

• 确定测试情况



确定等价类 identifying the equivalence classes

• 确定等价类的方法

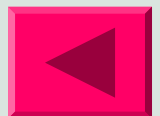
• 确定等价类的原则



确定等价类的方法

- 先取出每一个输入条件
- 把每一个输入条件化为成两组或更多组
- 列出等价类表

外部条件	有效等价类	无效等价类



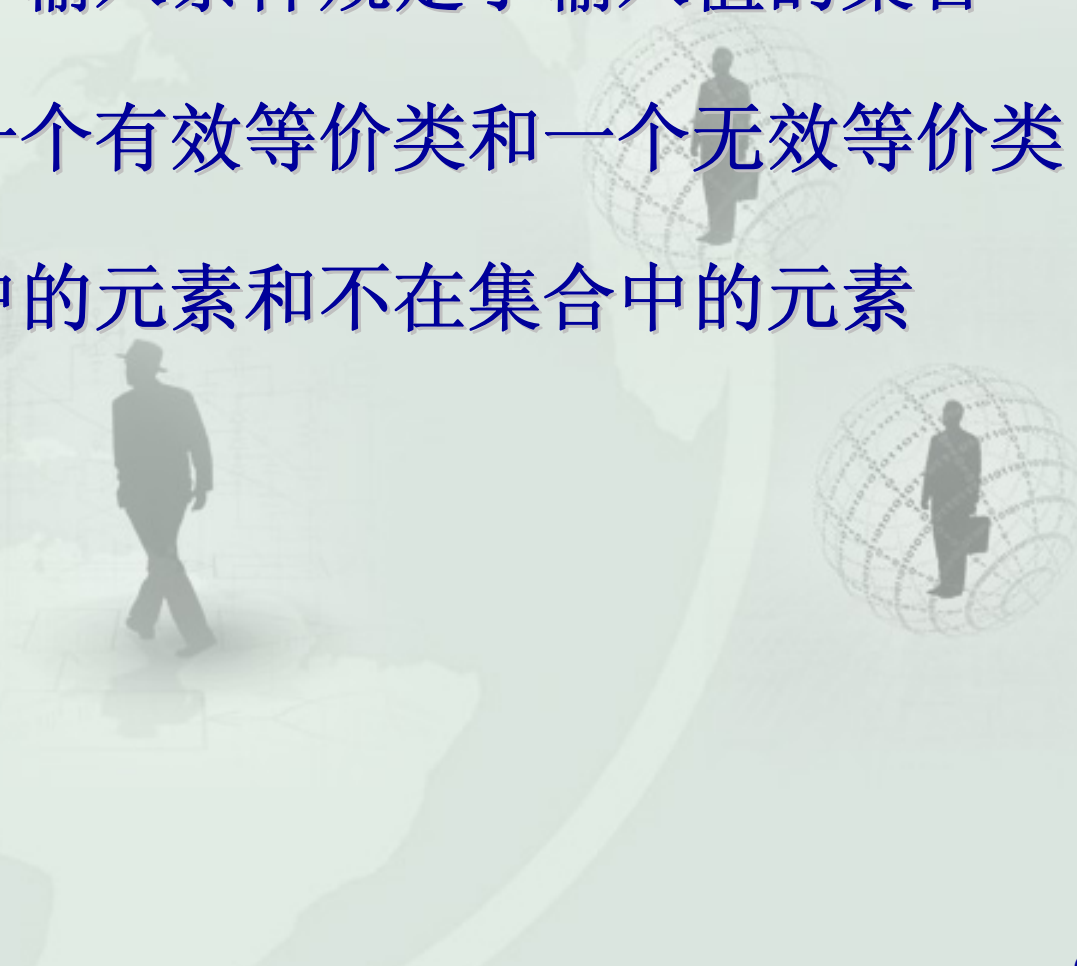
确定等价类的原则——范围

- 如果某个输入条件规定了值的范围
- 可确定一个有效等价类和两个无效等价类
- 某实数 X 的取值范围为 $1\sim 999$
- 则有效等价类为 $1\leq X\leq 999$
- 无效类为 $X<1, X>999$

确定等价类的原则——个数

- 如果一个输入条件规定了值的个数
- 可确定一个有效等价类和两个无效等价类
- 每班人数不超过40人
- 则有效等价类为 $1 \leq \text{学生人数} \leq 40$
- 无效类为学生人数=0, 学生人数>40

确定等价类的原则——集合

- 如果一个输入条件规定了输入值的集合
 - 可确定一个有效等价类和一个无效等价类
 - 在集合中的元素和不在集合中的元素
- 

确定等价类的原则——条件

- 如果一个输入条件规定“必须如何”的条件
- 可确定一个有效等价类和一个无效等价类
- 例：有效等价类是字母，无效等价类不是字母

确定等价类的原则——分解

- 如果有理由确信某一个等价类中的各元素在程序中的处理方式是有区别的
- 把这个等价类分成更小的等价类



确定测试情况 (identifying the test cases)

- 给每个等价类规定一个唯一的编号
- 设计一个新的测试情况，使其尽可能多地覆盖未被覆盖的有效等价类，直到所有有效等价类都被覆盖为止
- 设计出一个测试情况，使其仅仅覆盖一个未被覆盖的无效等价类，直到覆盖了全部无效等价类



边值分析

- 边值分析
- 边值分析与等价类划分的区别
- 边值分析的总原则



边值分析

- 相对于输入与输出等价类直接在其边缘上，稍高于其边界和低于其边界的这些状态条件
- 利用边值条件进行测试就是边值分析



边值分析与等价类法的区别

- 边值分析选取的测试数据应该刚好等于、刚刚小于和刚刚大于边界值
- 边值分析要考虑输入条件（输入空间）
- 边值分析还要考虑结果空间（考虑输出等价类）



边值分析的总原则

- 确定边界情况
- 边值分析选取的测试数据应该刚好等于、刚刚小于和刚刚大于边界值

具体如下：

边值分析的原则——范围

(1) 如果输入条件规定了值的范围

- 写出这个范围的边界测试情况
- 写出刚刚超出范围的无效测试情况

例：输入范围是-1.0到1.0

测试情况为-1.0,1.0,-1.001和1.001

边值分析的原则——个数

(2) 如果输入条件规定了值的个数

- 写出这个范围的最大个数和最小个数
- 写出稍小于最小个数和稍大于最大个数的状态

例：学生数是1~40

测试情况为1,0,40和41

边值分析的原则——输出

(3) 对输出条件使用第1条

例：有个程序计算每月的保险金额，若最小额是0元，最大额是1000元，写测试情况

- 写出导致扣除0元和1000元的测试情况
- 设计扣除一个负额或大于1000元的测试情况

边值分析的原则——输出

(4) 对输出条件使用第2条

例：一个情报检索系统根据某一输入请求，显示有关几个摘要，但不能多于4条，写出测试情况

- 写出使程序显示0，1和4个摘要的情况
- 写出使得程序错误地显示5个摘要的情况

边值分析的原则——集合、条件

(5) 程序的输入或输出是个有序集

- 测试集合的第一个
- 最后一个元素

(6) 另外可以找出其他的边界条件



因果图法 (case-effect graphing)

- 因果图法的原理
- 因果图使用符号
- 因果图实例

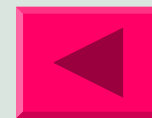
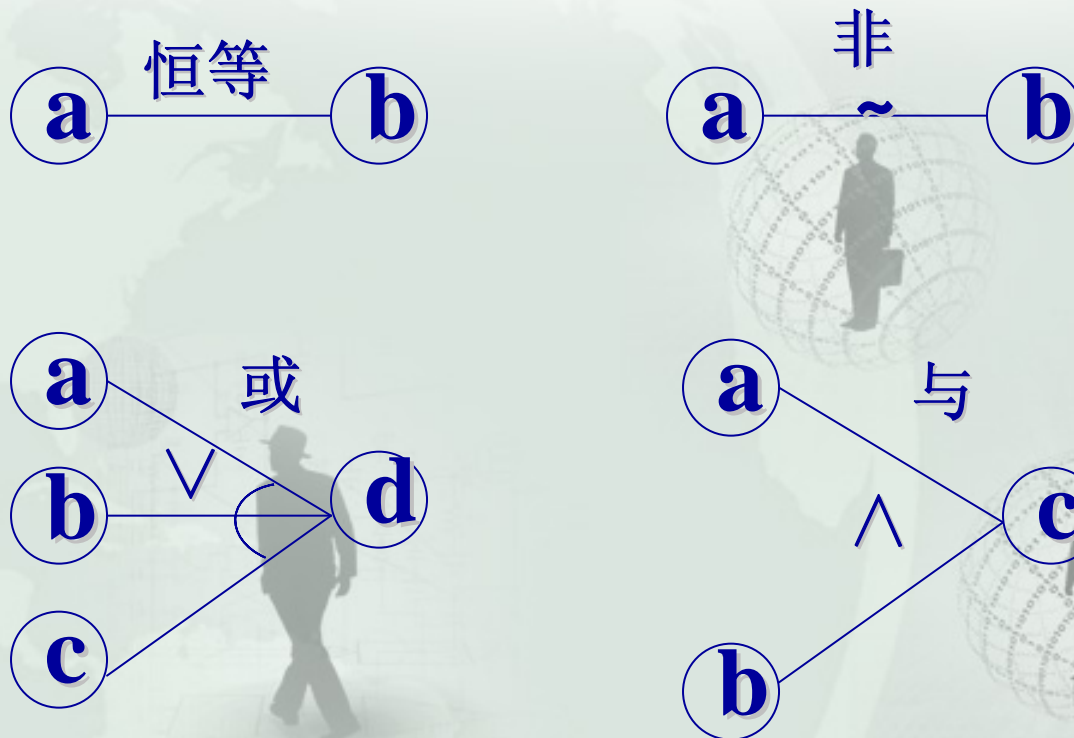


因果图法原理

- 从用自然语言书写的功能说明表中找出因--输入条件--输出结果
- 通过因果图将功能说明转换成一张判断表
- 为每种输出条件的组合设计测试用例



因果图使用的符号



因果图法实例

第一列字符必须是A或B，第二列字符必须是一个数字。在这种情况下，修改文件。
如果第一个字符不正确，则发出信息X12。如果第二列字符不是数字，则发出信息X13

实例分析

因

1--第1列字符是A

2--第1列字符是B

3--第2列字符是一个数字

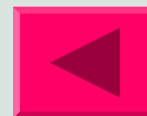
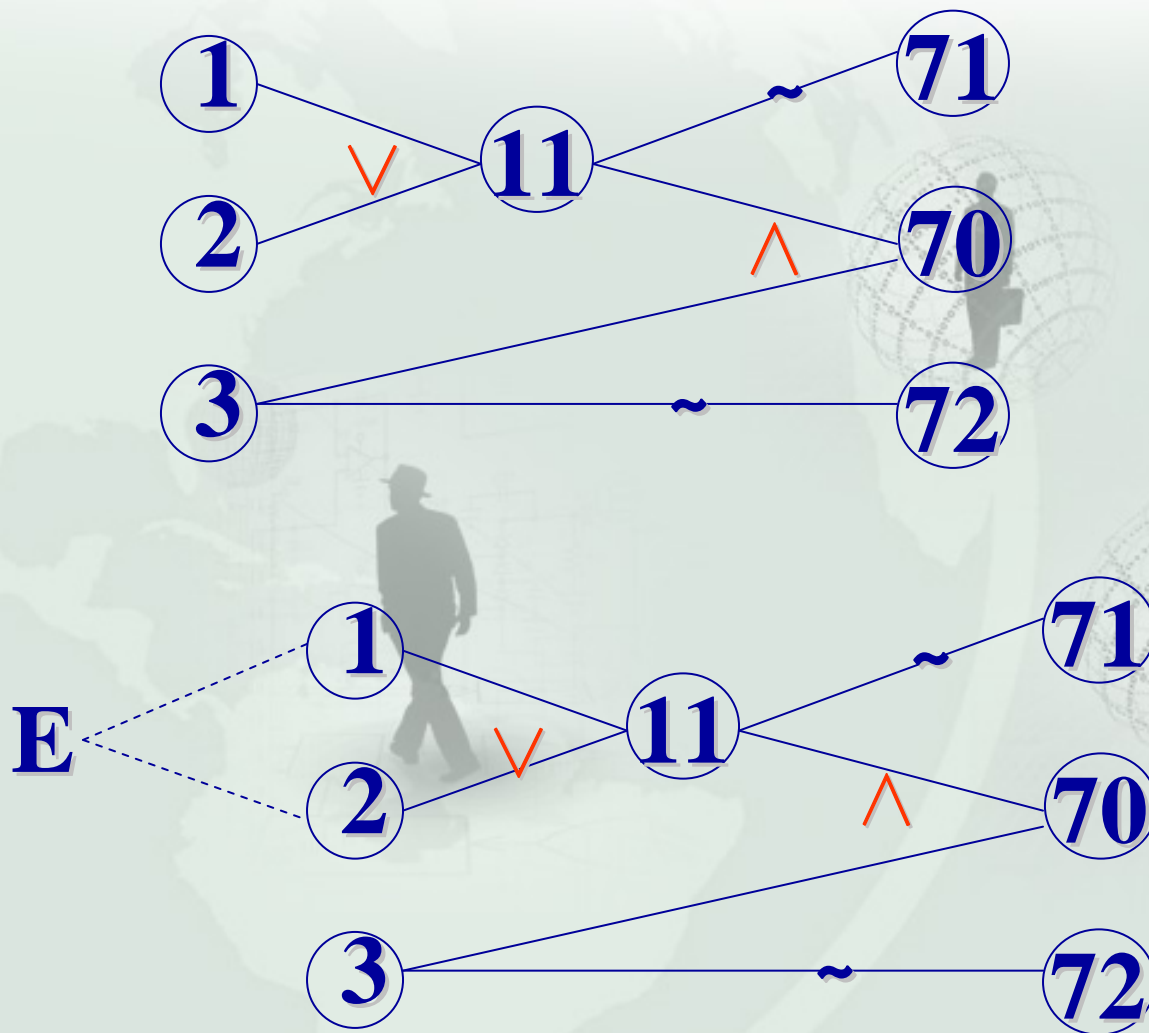
果

70--修改文件

71--发出信息X12

72--发出信息X13

实例分析



错误推错法（猜错）(error guessing)

- 很大程度上依靠直觉和经验进行
- 列举出程序中可能有的错误和容易发生错误的特殊情况
- 选择测试方案



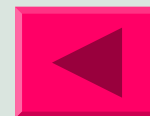
软件测试策略

- 用黑盒法设计基本的测试方案
- 用白盒法补充一些必要的测试方案

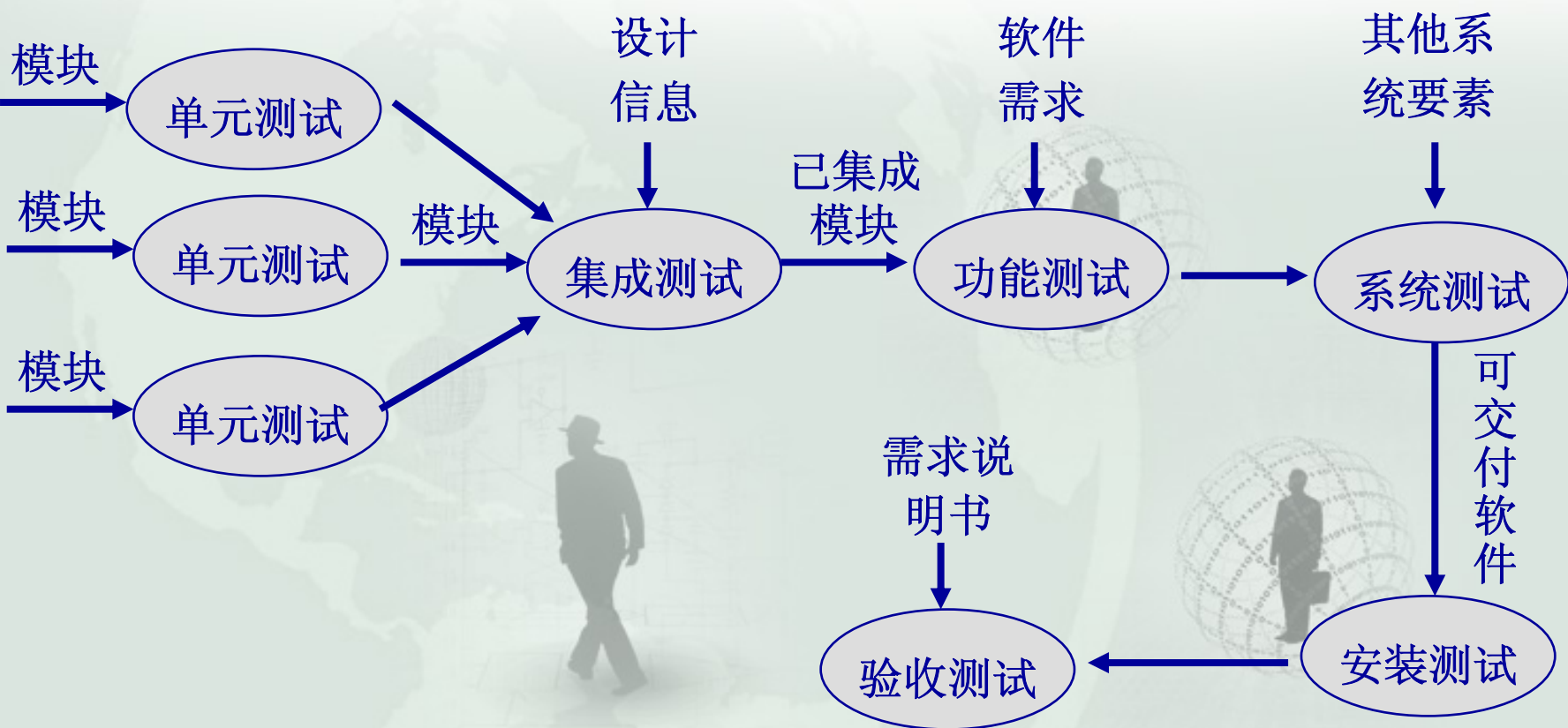
具体策略如下：

软件测试策略

- ☆如果规范含有输入条件的组合，便从因果图开始
- ☆在任何情况下都应该使用边界值分析的方法
- ☆必要时用等价划分法补充测试方案
- ☆必要时再用错误推测法补充测试方案
- ☆对照程序逻辑，检查已经设计出的测试方案



软件测试步骤



软件测试步骤

- 单元测试 (Unit Testing)
- 集成测试 (Integration Testing)
- 功能测试 (Function Testing)
- 系统测试 (System Testing)
- 安装测试 (Acceptance Testing)
- 验收测试 (Acceptance Testing)
- 软件测试框架



单元测试—模块测试 (Module Testing)

测试单个程序模块，确定模块的逻辑功能是否正确

单元测试的目的：

对模块的功能与定义模块的性能规范或接口规范进行比较

单元测试的依据

- 模块的规范--模块说明书

详细地说明了模块的输入、输出参数以及模块的功能（模块的外部属性）

- 模块的源程序

显示了模块内部所使用的数据和模块的功能实现方式（模块的内部属性）

单元测试—模块测试 (Module Testing)

- 单元测试的原则
- 单元测试的内容
- 单元测试数据的选择
- 单元测试情况的设计



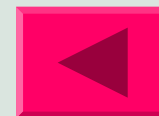
单元测试的基本原则

- (1) 至少一次测试所有的语句
- (2) 测试所有可能的执行或逻辑路径的组合
- (3) 在索引或下标的全域中测试所有的重复
- (4) 测试每个模块的所有入口和出口



单元测试内容

- 模块的接口
- 数据结构
- 重要执行通路
- 边界条件

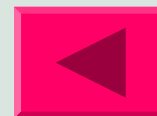


模块接口错误类型

- 一个模块向其子模块传递和接收数据元素的个数不相等;
- 传递的参数的属性和变元的属性不匹配;
- 传递给内部函数的变元数据类型和次序不匹配;
- 只修改了做输入用的变元;
- 全程变量的定义和用法在各个模块中定义不一致。

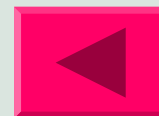
输入/输出错误

- 没有正确地打开或关闭文件;
- 文件记录、数据域的定义不正确;
- 键的存取不正确;
- 缓冲区的大小与记录长度不匹配;
- 输出信息中有文字书写错误;
- 文件终止条件没有正确处理。



数据结构

- 数据库的大小和属性没有正确定义;
- 搜索下标和索引的定义和使用不正确;
- 数据名称和使用不一致;
- 常数、累加器和计数器的初始化不正确;
- 数据项的格式和属性的定义不正确;
- 上溢、下溢或地址异常



重要路径

算法错误

- 中间结果数据项的大小、类型和精度等特性不正确；
- 算法操作顺序不正确；
- 对除数为0的除法的处理不正确；
- 精度不够。

重要路径

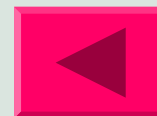
比较错误

- 所比较的数据项的属性不匹配;
- 在AND, OR等关系运算次序不正确;
- “差1”错误 (多循环一次或少循环一次);
- 错误的或不存在的循环终止条件;
- 当遇到发散的迭代时不能终止循环;
- 错误地修改循环变量。

重要路径

控制逻辑错误

- 没有经历所有选择结果的路径;
- 对所有选择路径的共同出口点的规定不正确;
- 循环下标不正确;
- 初始化和步长不正确;
- 对循环出口的规定不正确。



边界条件

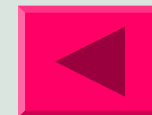
- 重点测试数据流和控制流在刚好等于、大于或小于最大值或最小值的情况



单元测试数据的选择

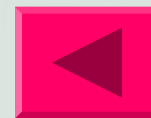
单元测试的数据集有：

- 值域
- 值类
- 离散值
- 值的次序集（用来测试顺序文件和表）



单元测试情况的设计

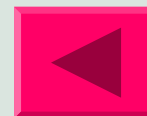
- 利用一种或多种白盒测试法对模块的逻辑结构（内部属性）进行分析，得到一些测试情况
- 根据模块说明书（外部属性）再用黑盒测试方法对原有的测试情况加以补充。



子系统测试——组装测试、集成测试

把多个模块组合在一起进行的测试
测试模块之间的接口，即模块之间的数据和控制传递

- 模块集成时可能出现的问题
- 模块集成测试的方式
- 增殖测试方式和非增殖测试方式的区别



模块集成时可能出现的问题

- 经过模块接口的数据是否丢失
- 一个模块是否破坏另一个模块的功能
- 子功能的组合是否达到了预期要求的主功能
- 全程数据结构是否有问题
- 单个模块的误差集成放大是否会达到不能接受的程度



模块集成测试的方式

● 非增值测试方式

● 增值测试方式



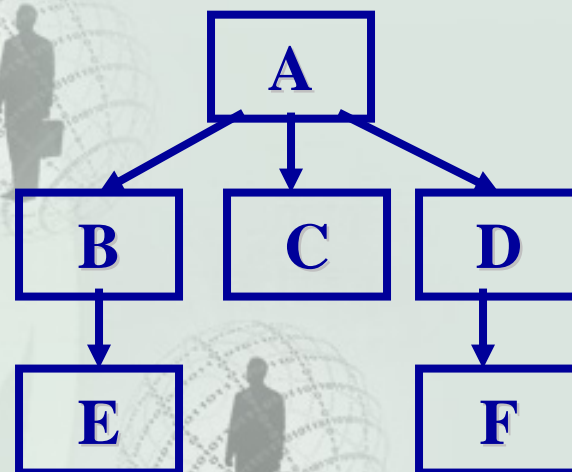
非增值测试方式

独立地测试程序的每个模块，再把它们组合成整个程序

驱动模块
(driver module)

用来驱动或传送测试情况给被测模块

桩模块
(stub module)



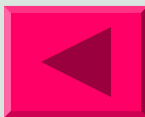
用来体现接受该调用时的控制，用来模拟模块E的功能

非增式测试的特点

- 为每一个模块准备相应的驱动模块(driver module)和桩(Stub module)模块，测试成本较高

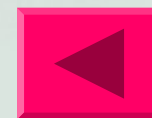
Stub模块的作用是为正在测试的上级模块提供调用的目标，以及为上级模块传递预期的数据和控制标识

- 集成后包含多种错误，难以对错误定位和纠正
- 1999年美国航天局火星基地登陆——多小组测试脚落地。前一小组测试着陆过程，不考虑数据位，后一个小组测试时对数据重置。



增式测试方式

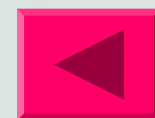
- 增式测试方式的种类
- 增式测试方式的特点
 - ☆ 自顶向下测试的特点
 - ☆ 自底向上测试的特点



增式测试的方式

• 自顶向下测试

• 自底向上测试



自顶向下测试 (top-down testing)

从顶端模块开始测试，下一次测试的模块至少有一个调用它的模块已经测试过

- 深度测试

- A, B, E, C, D, F

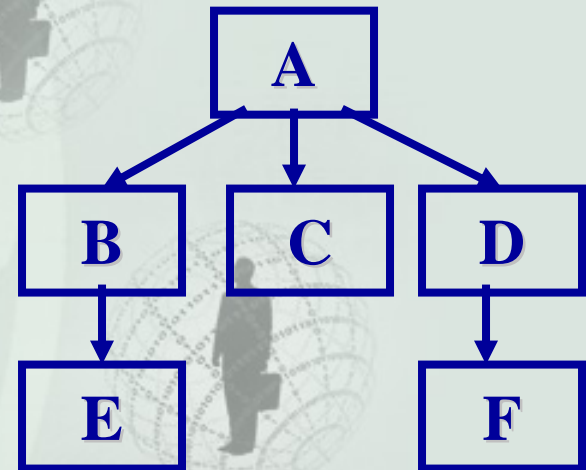
- A, D, F, C, B, E

- 宽度测试

- A, B, C, D, E, F

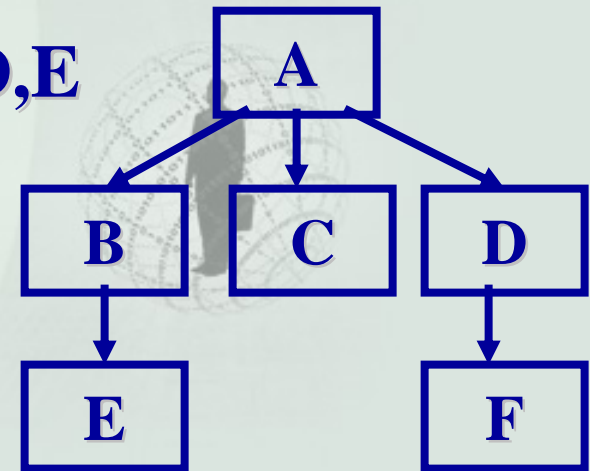
- A, C, B, E, D, F

桩模块
(stub module)



自顶向下测试 (top-down testing)

- (1) 测试模块A，设计桩模块 B,C,D
- (2) 根据选定的结合策略，用一个实际的模块代替桩模块
- (3) 在结合进模块B时，用桩模块C,D,E来测试模块A,B
- (4) 进行回归测试
- (5) 重复测试



自底向上测试 (bottom-up testing)

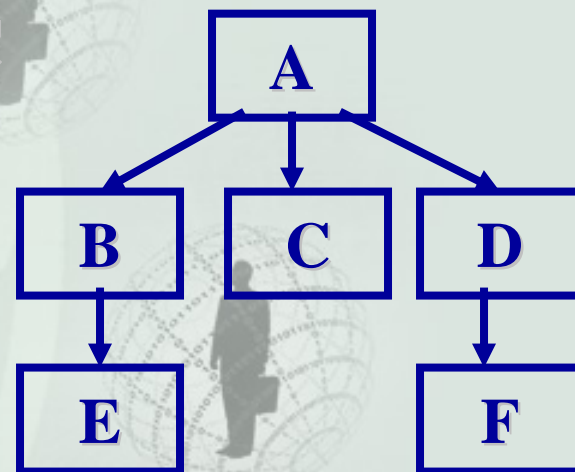
从程序的**末端模块**开始测试，下一次测试的模块的所有下层模块必须事先都被测试过

(1) 顺序或并行地测试E, F, C

(设计专门的驱动模块)

(2) 对B, D模块进行测试

(3) 对A模块进行测试



测试初期不能形成程序总体的概念



自顶向下测试的特点

优点:

- (1) 如果主要的错误趋向于发生在程序的顶端时，有利于查处错误
- (2) 一旦加入了I/O功能，测试情况很容易描述
- (3) 初期的程序设计轮廓可以让人们看到程序的功能，并使人们增强工作信心

自顶向下测试的特点

缺点:

- (1) 需要考虑桩模块
- (2) 桩模块比想象的更复杂
- (3) 在I/O功能加入之前, 桩中很难描述测试情况
- (4) 不可能或很难产生测试条件
- (5) 很难观察测试输出
- (6) 使人想到设计和测试同时进行
- (7) 会使人想推迟完成某些模块的测试



自底向上测试的特点

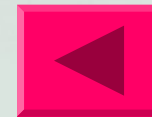
优点:

- (1) 如果主要的错误发生在程序的底端时, 有利于查处错误
- (2) 容易产生测试条件
- (3) 容易观察测试结果

自底向上测试的特点

缺点:

- (1) 必须给出驱动模块
- (2) 在加入最后一个模块之前，程序不能作为一个整体存在

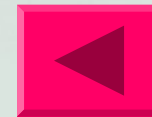


增式和非增式的区别

- 非增式测试方式需要更多的工作量
- 增式测试中，模块之间的接口的错误或是关于模块错误假定能够被较早地检查出来
- 利用增式测试，改错比较容易
- 增式测试可能导致彻底地对程序进行测试
- 非增式测试方法只需要用较少的机器时间，增式测试需要执行更多的机器指令。非增式测试需要更多的驱动模块和桩模块
- 用非增式测试方式在模块测试阶段的开始就有可能进行并行的工作

功能测试—确认测试Function Testing

- 功能测试概述
- 功能测试原理
- 功能测试存在的问题



功能测试概述

- 是继集成测试之后进行的测试
- 目的是找出程序和其外部规范之间的不一致
- 外部规范是对模块外部属性的描述，一般采用黑盒方式



功能测试—有效性测试

找出程序和其外部规范之间的一致

- 使用黑盒法进行测试
- 对外部规范进行分析
- 得出一组测试情况



在进行功能测试时注意的问题

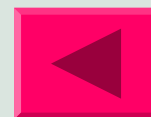
- (1) 要考虑到那些不合理的和意想不到的输入条件
- (2) 要将预期结果的定义做为测试情况的重要部分
- (3) 目的是暴露错误，不是证明程序符合外部规范



系统测试

• 系统测试

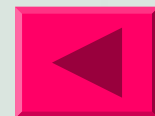
• 系统测试类型



系统测试

将系统或程序与其原定目标相比较

- 发现设计和编码的错误
- 验证系统可以提供需求说明书中指定的功能
- 验证系统的动态特性符合预定要求



系统测试的类型

- 机能测试 (facility testing)
- 批量测试 (volume testing)
- 强度测试 (stress testing)
- 便利性测试 (usability testing)
- 安全性测试 (security testing)
- 性能测试 (performance testing)
- 存储量测试 (storage testing)

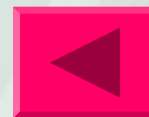
系统测试的类型

- 配置测试 (facility testing)
- 兼容/变换测试 (volume testing)
- 可靠性测试 (stress testing)
- 恢复测试 (usability testing)
- 可安装性测试 (security testing)
- 可用性测试 (performance testing)
- 文件资料测试 (storage testing)
- 工序测试 (storage testing)



机能测试 (facility testing)

- 判断系统是否把目标提到的每个机能真正地实现了
- 方法：逐句扫描目标（人工进行即可）



批量测试 (volume testing)

- 让程序处理大量的数据
- 企图证明程序不能处理目标中指出的大批量数据
- 代价很大

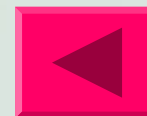


强度测试 (stress testing)

- 让程序在高负荷即高紧张的情况下运行

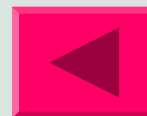
在很短的时间内遇到最多的数据

- 用于测试负载不定的程序，或交互式的、实时的，以及过程控制的程序



便利性测试 (usability testing)

- 便利性测试力图确定系统的人为因素
- 分析程序中的人为因素，主要靠主观臆断
- 如每个与用户交换的信息是否与使用者的智力水平、文化程度及所处的环境相一致？
- 程序的所有输出结果是否有意义、语句通顺？
- 对错误的诊断是否简明易懂？程序的操作是否方便？



性能测试 (performance testing)

- 说明在一定工作负荷和格局分配条件下，响应时间及处理速度等特性
- 美国爱国者导弹：一个很小的系统时钟错误，累计起来可能拖延14小时。在多哈袭击中，系统拖延100多个小时，击毙28名美军士兵



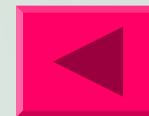
存储量测试 (storage testing)

- 说明诸如程序所用的内存和外存容量，临时文件和溢出文件的大小等
- 要设计测试情况以证明没有达到其存储量的目标
- 1974千年虫——工资系统年设计为2位，程序员95年退休，程序被沿用。



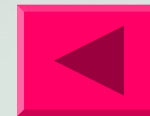
配置测试 (configuration testing)

- 至少每一个类型的硬件的最大最小的配置都要测试到
- 如果程序本身能重新配置，那么就要测试可能出现的各种配置



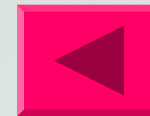
兼容/变换测试 (compatibility/conversion testing)

- 说明与现行系统的兼容性
- 说明如何由现有系统转换而来。
- 设法证明未能满足兼容性的目标，以及不能实现两系统间相互转换
- 1994年迪斯尼的狮子王——不兼容



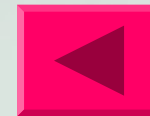
可靠性测试 (reliability testing)

- 可靠目标很难测试
- 可以考虑用一套数学模型来估计该目标的有效性
- 英特尔奔腾浮点除法
- $(4195835/3145727) * 3145727 - 4195835$



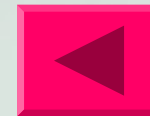
恢复测试 (recovery testing)

- 像操作系统、数据库管理系统和远程处理程序这类程序往往有系统恢复的目标，说明在出现程序错误、硬件失效及数据错误之后，整个系统应该怎样恢复
- 要证明恢复功能不正常工作。



可用性测试 (serviceability testing)

- 这类目标定义了供系统使用的公用子程序
- 定义了纠正明显错误的平均时间
- 定义了维护过程及内部逻辑文件资料的质量等



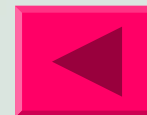
文件资料测试 (documentation testing)

- 检查文件资料是否准确
- 分析资料中的每个例子都要编成测试情况送给程序运行



安装测试

- 1994年迪斯尼的狮子王
- 检验用户选择的一套任选方案是否相容
- 检验系统的每一部分是否齐全
- 所有文件是否产生并确实有需要的内容
- 硬件配置是否合理



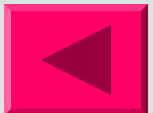
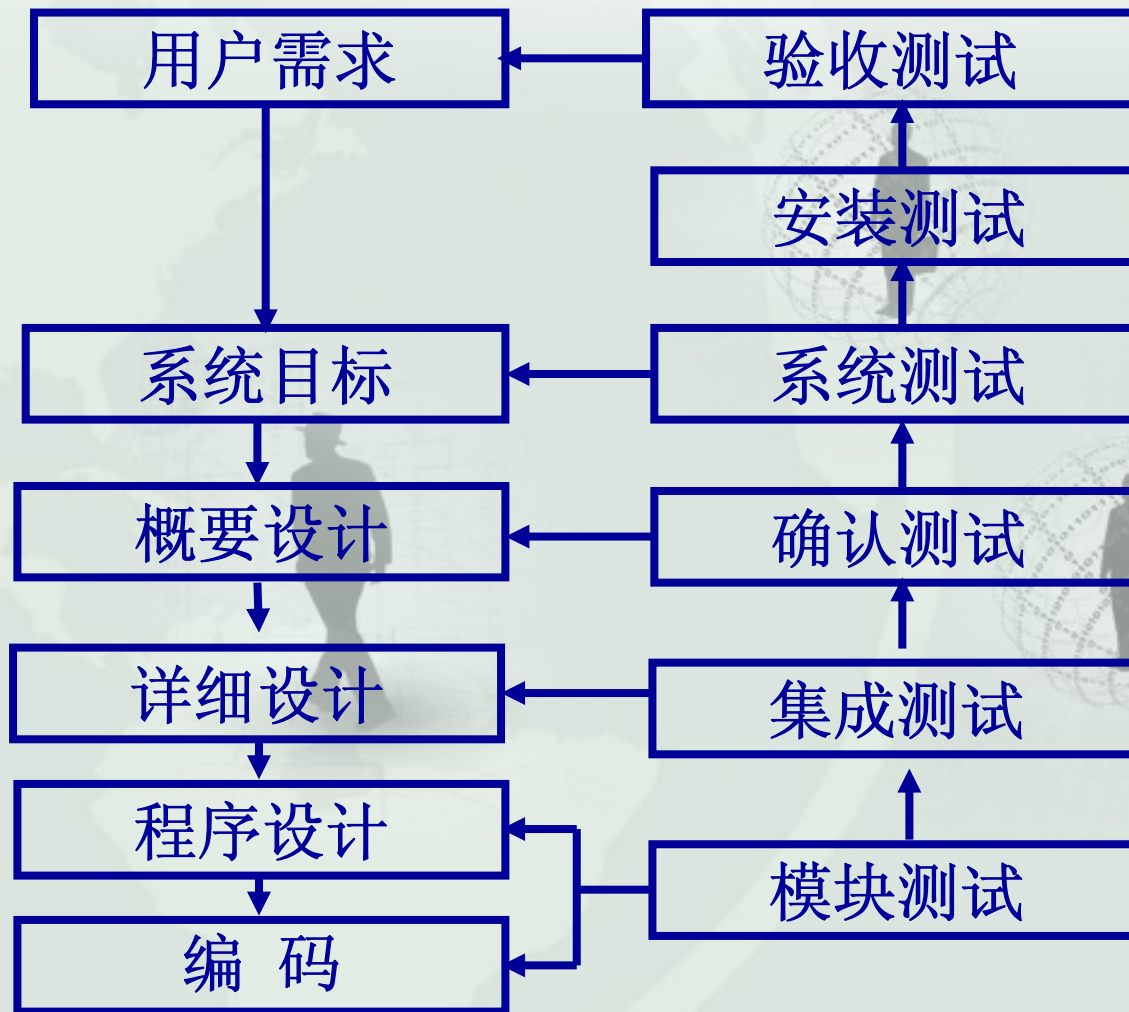
验收测试 (Acceptance testing)

把程序和用户的需要性比较

- 测试内容与系统测试基本类似
- 可以使用实际数据
- 验证系统能满足用户的需要
- 往往发现系统需求说明书中的错误



软件测试框架



软件正确性证明

程序能够准确无误地完成编写者所期望赋予它的功能，或
对任何一组允许的输入信息得到一组相应的正确的输出信息

•程序正确性的种类

☆程序的部分正确性

☆程序的终止性

☆程序的完全正确性

为了证明一个程序的完全正确性，通常采用
证明该程序的部分正确性和终止性

软件正确性证明的方法

- 部分正确性的证明方法

- ☆ A.Floyd的不变式断言法

- ☆ B.Manna的子目标断言法

- ☆ C.Hoare的公理化方法

- 终止性的证明方法

- ☆ A.Floyd的良序集方法

- ☆ B.Knuth的计数器方法

- ☆ C.Manna的不动点方法



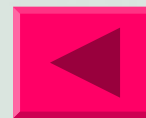
信息系统实施

第四节 系统调试



系统调试

- 系统调试概述
- 系统调试的步骤
- 系统调试的技术
- 系统调试的方法



系统调试概述

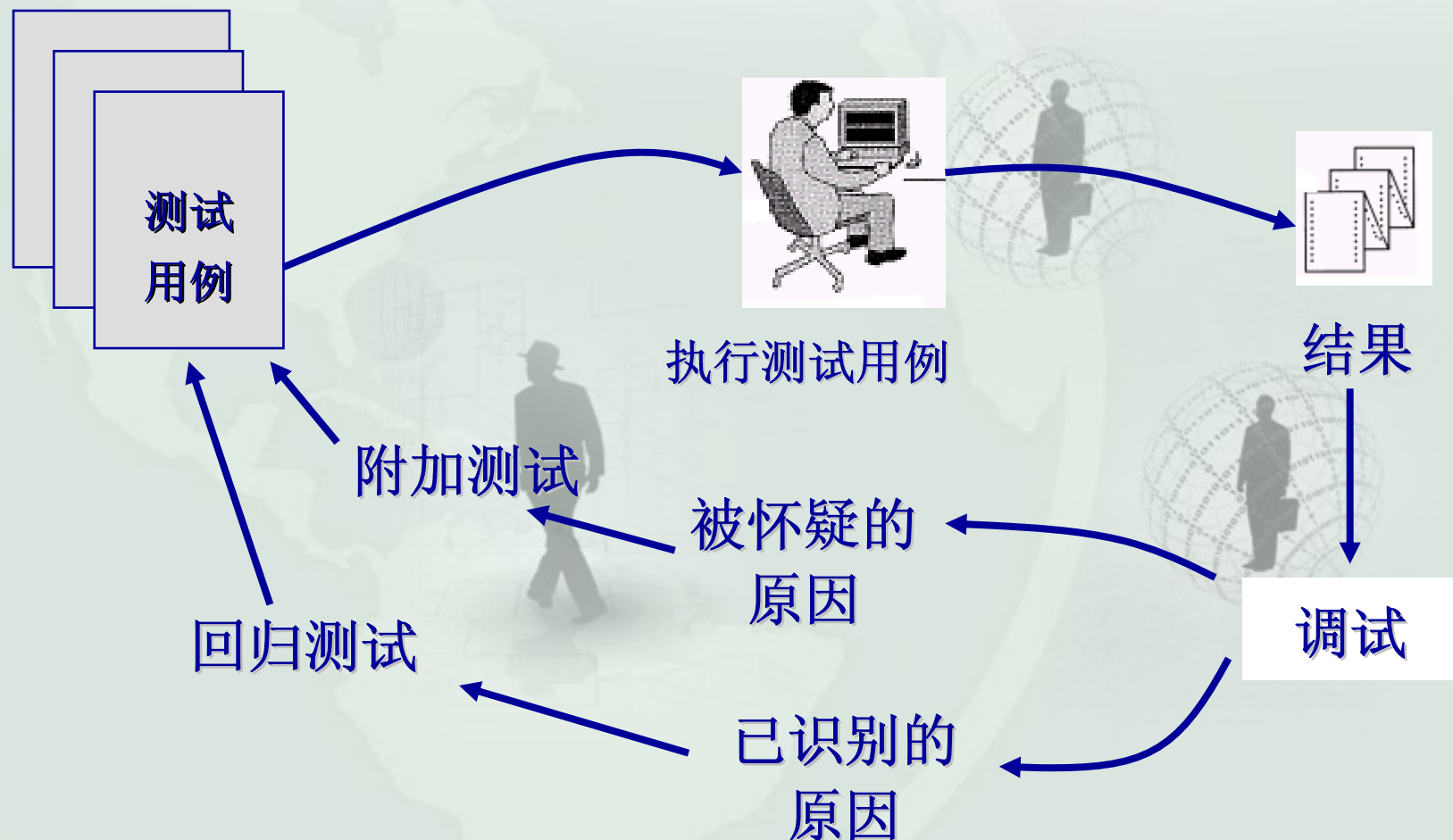
- 调试是指在成功地进行了测试之后，进一步诊断和改进程序中存在的错误过程
- 它由两部分工作组成：
 - 确定存在在程序中发生错误的确切的性质和位置
 - 对程序进行修改和排除。

系统调试概述

- 可以保证新系统运行的正确性和有效性
- 将一切可能发生的问题和错误尽量消灭在正式运行之前
- 系统调试事先应拟订一份方案
- 确定调试步骤，可提高效率，缩短周期，降低费用



系统调试的步骤



系统调试的步骤

1. 错误的诊断

从测试程序中存在错误的某些迹象出发，确定错误的准确位置，也就是找出是哪个模块或哪些接口引起的错误。错误诊断是非常难的，它是调试过程的关键

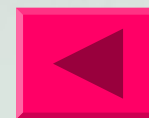
2. 错误排除

诊断出错误的准确位置以后，仔细研究这段代码以确定出现问题的真正原因，并设法改正错误。



系统调试的技术

- 输出存储器内容
- 打印语句
- 自动工具



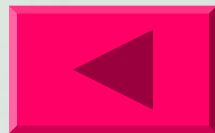
系统调试的方法

- 试探法 (Tentative)
- 回溯法 (Backtracking)
- 折半查找法 (Bisearch)
- 归纳法 (Induction)
- 演绎法 (Deductive Method)



信息系统实施

第五节 系统切换



系统切换

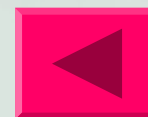
□ 系统切换前的准备

□ 系统切换



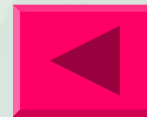
系统切换前的准备

- 数据准备
- 文档准备
- 用户培训
- 系统设备



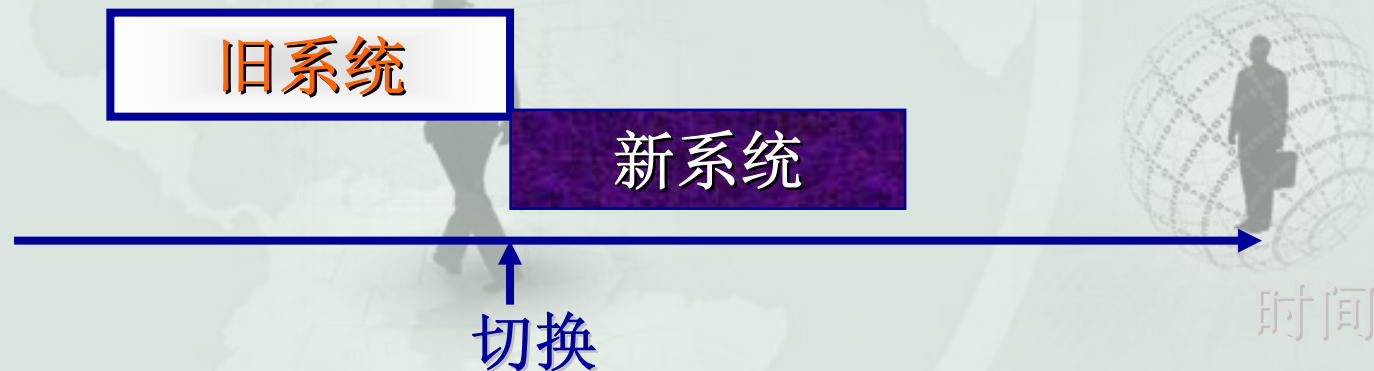
系统切换

- ❖ 直接转换方式
- ❖ 平行转换方式
- ❖ 逐步转换方式



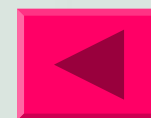
直接转换方式

- 在某一特定的时刻，旧系统停止使用，同时新系统立即投入使用



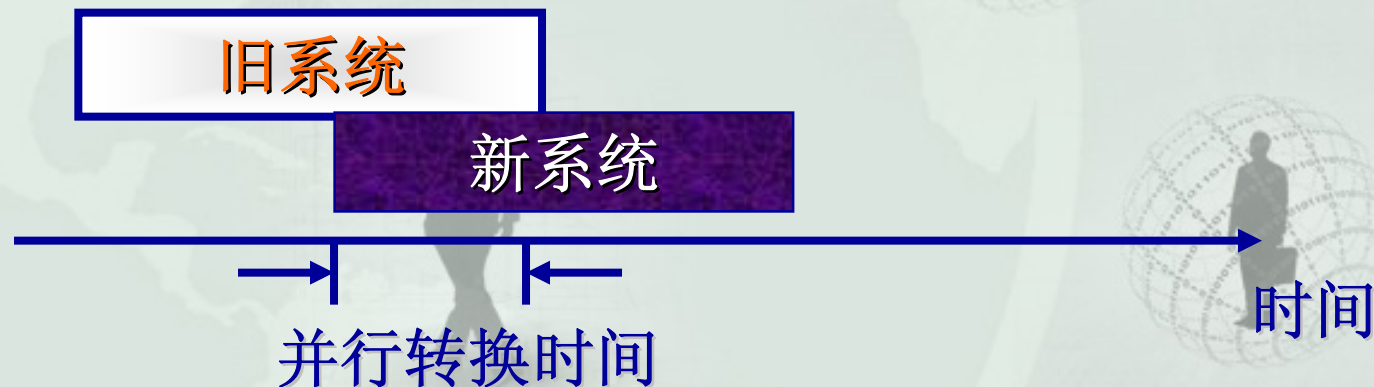
直接转换方式的特点

- 在某一特定的时刻，旧系统停止使用，同时新系统立即投入使用
- 转换简单
- 人员和设备费用节省
- 预先要经过详细的测试和模拟运行
- 风险大



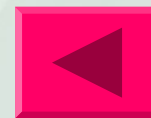
平行（并行）转换方式

- 在一段时间内新旧系统并存，各自运行完成相应的工作，并相互对比、审核



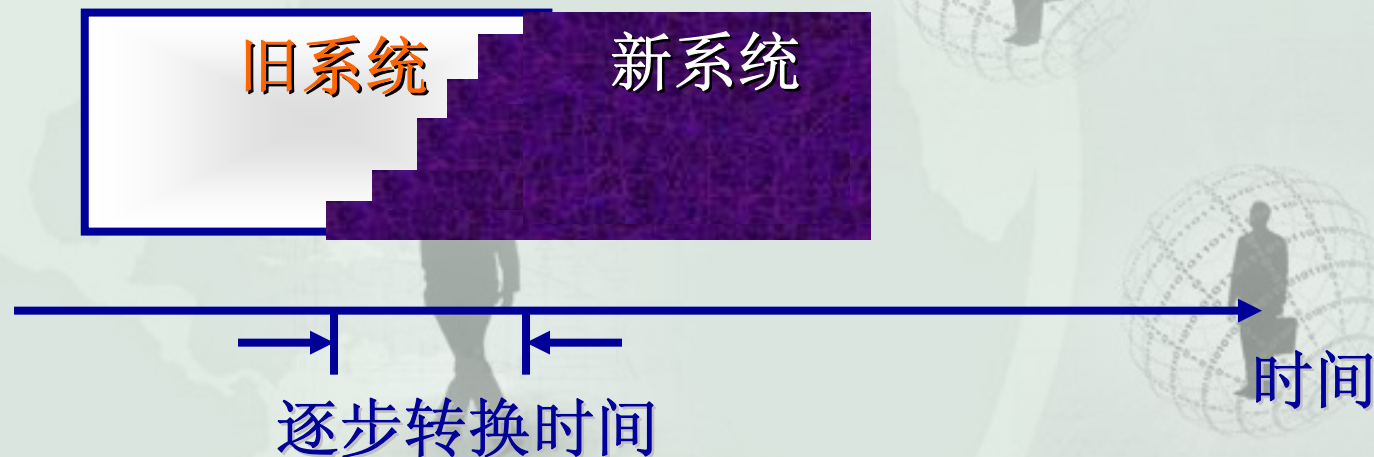
平行（并行）转换方式的特点

- 新旧系统并存一段时间
- 人员和设备费用增加
- 系统的可靠性高
- 风险较少，新系统的运行成功率高



逐步（分段）转换方式

- 分阶段、按部分地完成新旧系统的交替过程，开发完一部分则在某一时间段内转换一部分



逐步（分段）转换方式的特点

- 开发完一部分就转换一部分
- 避免直接转换方式的风险
- 避免平行方式的双倍费用
- 逐步转换方式的接口多



小 结

- 系统实施的任务
- 系统测试的方法
 - ◆ 静态测试
 - 程序审查会
 - 桌前检查
 - 人工运行
 - ◆ 动态测试
 - 等价类法
 - 边值分析法
 - 因果图法
 - 错误推测法

小 结

- ◆ 非增值测试
- ◆ 增值测试
 - 自顶向下
 - 自此向上
- 系统转换方式
 - ◆ 直接转换
 - ◆ 并行转换
 - ◆ 逐步转换