# 第一讲 软件测试概述

## 软件测试背景

### 软件测试史

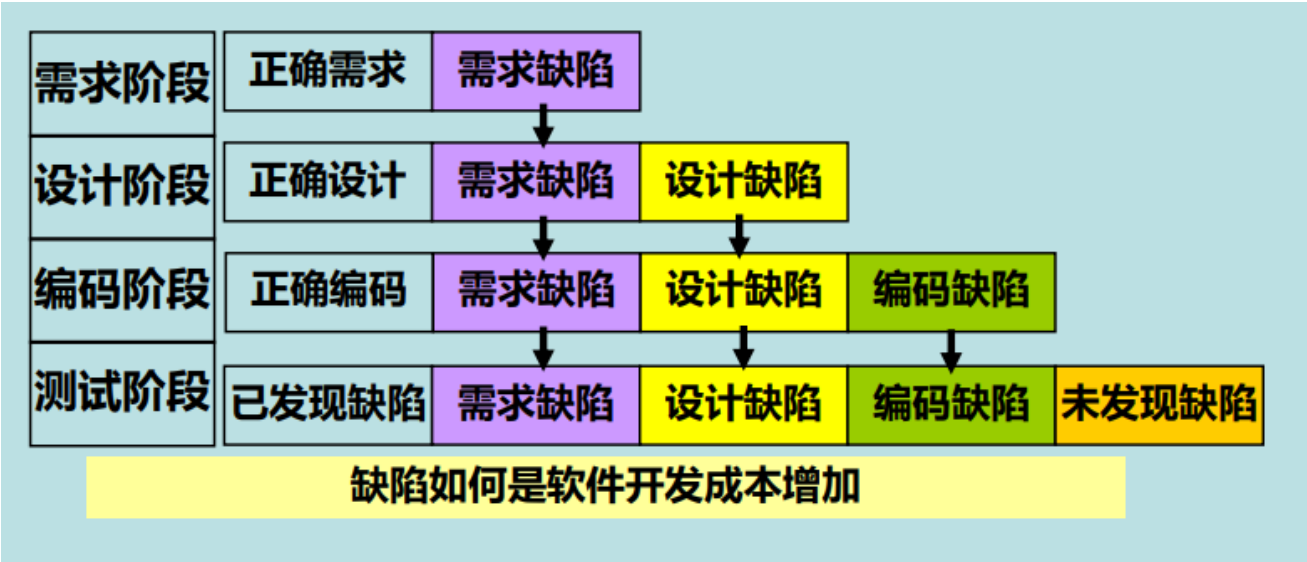
- 面向调试的时代（软件是否实现用户预期的功能要求，投入少，介入晚）
- 面向可运行的时代
- 面向缺陷发现的时代
- 面向评估的时代
- 面向预防的时代

### 软件危机

落后的软件生产方式无法满足日趋复杂大型软件系统的开发需求（项目延期、经费超支、产品无法维护），原因在于缺乏工程化的开发缺陷的不断积累和放大效应。缺陷修复代价随开发时间快速增加

#### 缺陷积累

在软件开发的不同阶段会产生不同的缺陷——需求阶段（需求缺陷）=>设计阶段（设计缺陷）=>编码阶段（编码缺陷）=>测试阶段（未发现缺陷）。随之而来的是缺陷的修复成本快速增加（可能面临大规模重构的风险）。




#### 有关测试观点的正确理解

- 面向可运行的时代
  - 测试就是建立一种信心，认为程序能够按照预期设想运行
- 面向缺陷发现的时代
  - 测试是尽可能多地发现软件错误
  - Myers的软件测试定义：测试是为发现错误而执行一个程序或者系统的过程
    - 测试是为了证明程序有错，而不是证明程序无错误

        - 一个好的测试用例是在于它能发现至今没有发现的错误
        - 一个成功的测试是发现了至今没有发现的错误的测试
- 测试的意义---软件缺陷难以避免
    - 集成测试必要性
    - 可靠性测试必要性
    - 不要企图掩盖缺陷
    - 性能测试在互联网应用中的重要性
    - 软件测试是保证软件质量的重要手段，软件测试深入软件开发过程的每个阶段，在有限的开发条件下，最大程度地抱枕最终软件产品符合用户需要。

# 软件测试基本概念

## 测试定义

分析某个软件项以发现现存和要求的条件之间的差别并评价此软件项 的特性。

测试与调试

## 测试目的

- 确保软件质量——找出软件的错误和缺陷，降低软件发布后潜在错误和缺陷造成的损失；验证软件是否能满足用户需求，树立对软件的信心
- 确保软件开发过程方向的正确性——通过测试报告提供的数据
- 1. 确保软件质量——找出软件的错误和缺陷，降低软件发布后潜在错误和缺陷造成的损失；验证软件是否能满足用户需求，树立对软件的信心
  2. 确保软件开发过程方向的正确性——通过测试报告提供的数据
- **测试原理/原则：**
- 1. 确保软件质量——找出软件的错误和缺陷，降低软件发布后潜在错误和缺陷造成的损失；验证软件是否能满足用户需求，树立对软件的信心
  2. 确保软件开发过程方向的正确性——通过测试报告提供的数据
- **测试原理/原则：**

测试原理

- 用户需求至上，任何错误应当追溯到用户需求，最严重的错误是导致软件无法满足需求
- 测试是有计划的活动
- 缺陷出现的集群性（缺陷集中出现，大部分错误可能来源于小部分内容）
- 测试应该从小规模走向大规模
- 穷尽测试不可能
- 有效的测试要由第三方独立进行（专门的测试人员）
- 测试无法揭示所有缺陷
- 测试的杀虫剂悖论（潜在缺陷对已进行的测试有免疫力，一次测不出来，再用同样的方法反复测试很难发现）
- 测试是有风险的行为（测试量增大的过程中会出现“性价比”越来越低的情况，这也导致了缺陷的积累）

## 测试过程

- 拟定测试计划
- 编制测试大纲
- 设计测试用例
- 实施测试
- 分析测试结果

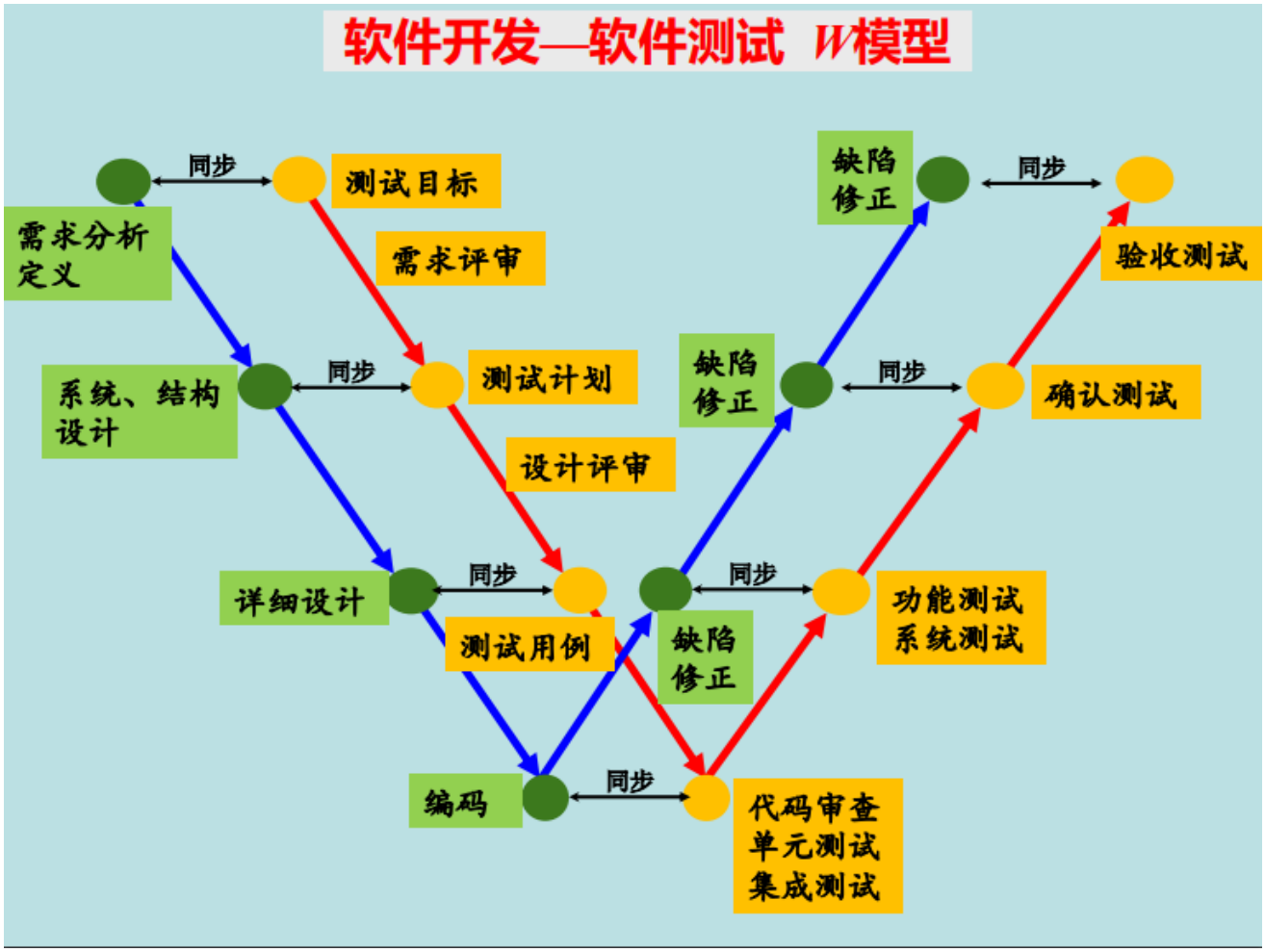
# 测试用例（三要素）

- 定义
  - 一个文档，详细说明**输入、期望输出**和为一测试项所准备的一组的**执行条件**
- 设计准则
  - 代表性（代表合法非法，边界越界，极限条件，操作环境）；
  - 可判定性（测试结果正确性可以判定）；
  - 可再现性（对同样测试用例结果应该一样的，个人理解这样可以避免偶然记录错误之类的情况）
- 三要素
  - 输入
  - 期望输出
  - 执行条件

# 软件测试类型

- 测试技术
  - 黑盒测试：基于需求和功能，不了解内部实现
  - 白盒测试：基于软件内部逻辑，覆盖率是退出条件
  - 灰盒测试：黑白之间，既要关注外部表现也要关注程序内部逻辑结构
- 开发阶段
  - 单元测试：对最小的软件设计单元模块的验证测试
  - 集成测试：模块间的接口是否正确，模块协调性
  - 系统测试：整个系统的行为和错误属性
  - 性能测试：响应时间、吞吐率，并与不同版本或同类产品比较
  - 确认测试：验证软件是否符合用户的期望工作方式
  - 验收测试：保证客户对需求满意
  - 回归测试：保证软件的修改正常运行，不影响个现有功能（增量开发？）
- 执行状态
  - 静态测试：**不运行程序**，对程序和文档进行检查
  - 动态测试：通过人工或利用工具**运行程序**进行检查
- 执行主体
  - 开发方测试（$\alpha$测试）；用户测试（$\beta$测试）；第三方测试
- 特殊测试
  - 国际化测试;即兴测试;兼容性测试;安全性测试;可用性与易获得性测试;面向对象系统测试;Web测试

# 软件测试过程$W$模型




## 软件测试现状和趋势

### 软件测试的地位（工作量百分比）

- 国际现状
  - 测试在软件开发中占有不可获取的重要地位（53%~87%）
  - 开发人员：测试人员 = 1:1（微软1:2）
- 国内现状
  - 软件测试逐渐受到重视；测试人才缺口；高素质测试人才紧缺
  - 专业集中于计算机和相关专业
  - 学历集中于本科
  - 初级测试工程师比重较大

# 第二讲 白盒测试

## 白盒测试基本概念

### 定义

一种基于源程序或代码结构与逻辑，生成测试用例以尽可能多地发现并修改源程序错误的测试方法

白盒测试分静态和动态

其他称谓：结构测试，逻辑驱动测试，基于程序的测试

## 意义

主要的单元测试方法

保证软件质量的基础

## 实施者

单元测试阶段：一般由开发人员进行

集成测试阶段：一般由测试人员和开发人员共同完成

## 步骤

- 动态：
  - 程序逻辑分析
  - 生成测试用例
  - 执行测试
  - 分析覆盖标准
  - 判定测试结果
- 静态：
  - 桌面检查
  - 代码走查
  - 代码审查

# 静态白盒测试

## 定义

在不执行软件 的条件下有条理地仔细审查软件设计 、体系结构和代码 体系结构和代码，从而找到软件缺陷的过程，有时称为结构化分析

## 静态白盒测试方法

桌面检查->代码检查/代码走查->代码审查 【规范化程度由弱变强】

### 桌面检查

- 实施者
  - 通常是代码编写者
  - 多数程序员在编译执行前都会做桌面检查
- 特点
  - 无结构化或者形式化方法保证
  - 不维护记录或者检查单

- 记录格式
  - 行号
  - 变量
  - 条件
  - 输入输出
- 优点
  - 编码者容易理解和阅读自己的代码
  - 开销小，没有指定的进度
  - 尽早发现缺陷
- 缺点
  - 开发人员不是实施最佳人选
  - 依靠个人勤奋和技能，有效性难移保证

## 代码走查

都是以组为单位进行代码阅读的测试方式

- 特点
  - 由特定人员组成的团队通过会议完成
  - 与会者充当计算机执行测试用例
  - 开发人员及时回答与会者提出的问题

## 代码评审

代码审查是有利的质量技术

- 代码审查步骤
  - 计划
  - 概述
  - 准备
  - 审查会议
  - 审查报告
  - 返工
  - 跟进
- 代码审查作用
  - 发现代码缺陷
  - 提高代码质量
  - 及早定位缺陷群集位置
- 代码审查挑战
  - 费时间
  - 涉及多人参加
  - 不能保证参与者全部理解程序

## 动态白盒测试

不但要提供软件源代码，还要提供可执行程序，测试过程需要在计算机上执行程序

- 优点
  - 检测代码中的判断和错误
  - 揭示隐藏在代码中的错误
  - 对代码的测试比较彻底
- 缺点
  - 无法检测代码中中不可达路径
  - 不验证需求规格
- 动态白盒测试流程
  - 设计测试用例
  - 执行测试
  - 收集测试结果
  - 测试覆盖分析
    - 测试足够->测试结果文档
    - 测试不够->设计测试用例【返回第一步】
- 判定准则
  - 意义：对测试执行到何时才是足够的定量回答
  - 作用：测试软件的一种度量标准，描述程序源码被测试的程度
- 测试方法
  - 控制流测试方法
  - 数据流测试方法

# 控制流测试

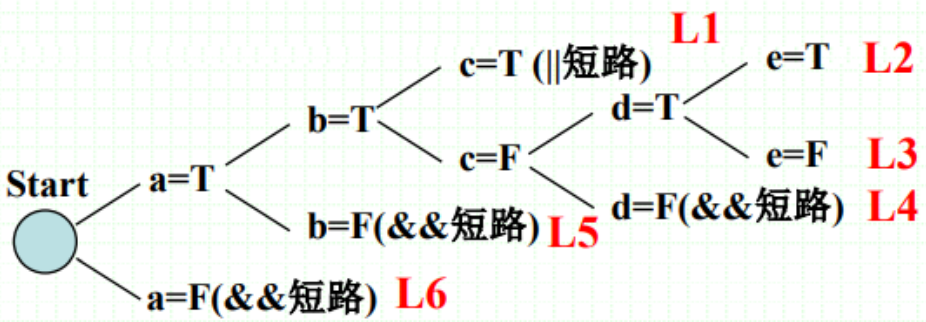
## 六种覆盖标准

- 语句覆盖：保证程序中每条语句都执行一次。
  - 本质：控制流程图的节点覆盖
  - 缺陷：100%覆盖困难（错误处理，不可达代码）可能会漏边
- 判定覆盖：保证程序中每个判断去T、F至少各一次。
  - 本质：控制流程图的边覆盖
  - 缺陷：不能发现每个条件的错误
- 条件覆盖：保证每个判断的每个条件取值至少满足一次
  - 缺陷：错误屏蔽问题；可能会有较多测试用例；不能覆盖所有路径
- 判定条件覆盖：保证每个条件和由条件组成的判断的取值
  - 缺陷：可能会有故障屏蔽和错误屏蔽；可能会有较多测试用例；不能覆盖所有路径
- 条件组合覆盖：保证每个条件的取值组合至少出现一次
  - 缺陷：错误屏蔽问题；代价昂贵 $2^n$ 个组合数字；不能覆盖所有路径

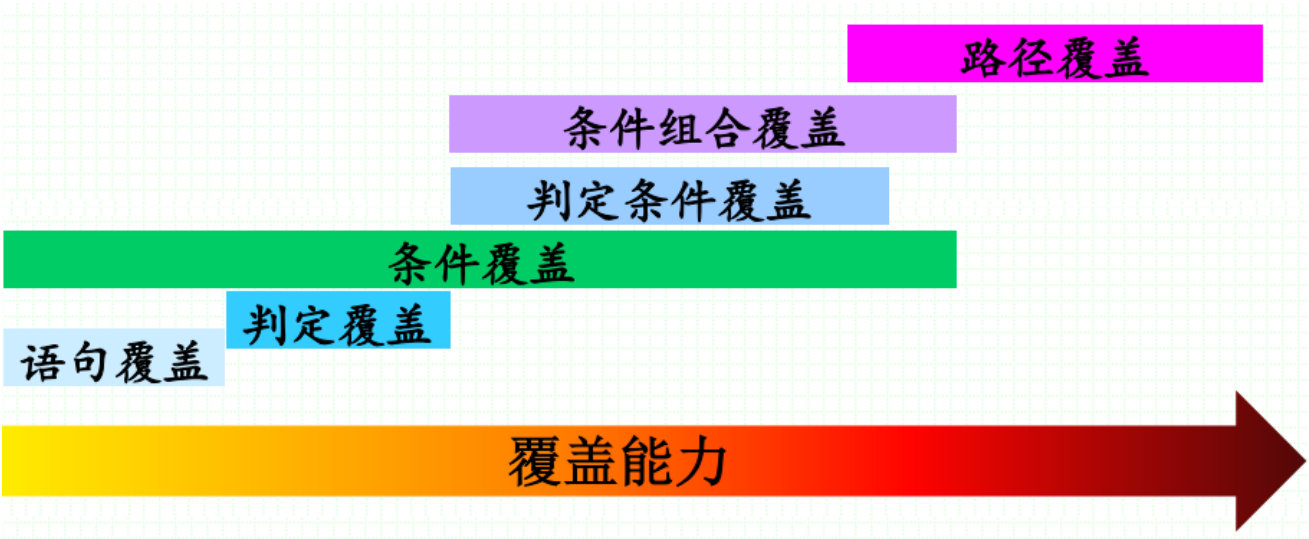
# 条件组合测试用例的约简

## 利用逻辑短路机制寻找最小测试用例集

**a && b && (c || (d && e))**




- 路径覆盖：覆盖程序中所有可能的路径
  - 本质：控制流程图的路径覆盖
  - 缺陷：不能替代涉及条件的测试；路径数可能较多
- 总结
  - 在考虑逻辑短路的情况下
    - 条件覆盖 = 判定条件覆盖
    - 条件覆盖 != 条件组合覆盖
  - 六个测试覆盖能力比较



# 基本路径测试

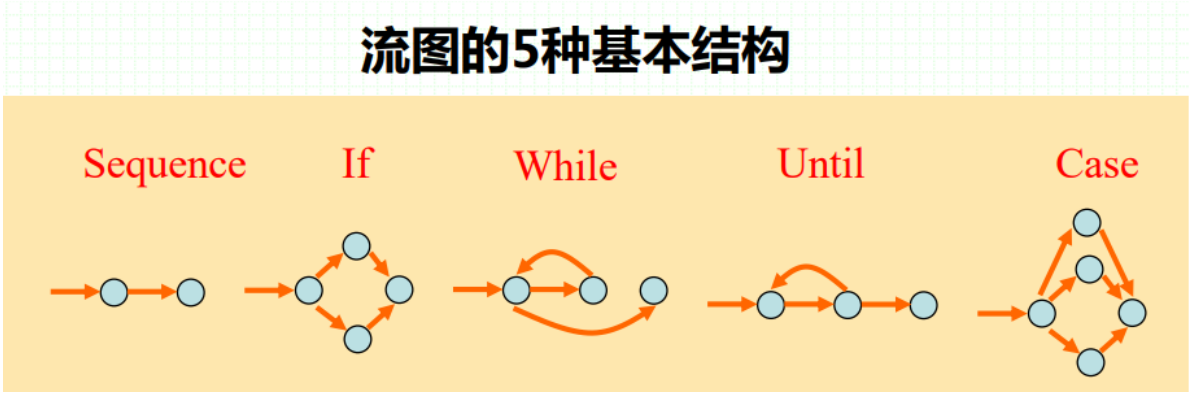
- 概念
  - 选择足够的测试用例，保证每条可能执行到的路径都至少经过一次（如果包含环路，则要求每条环路至少经过一次）
  - 如果判定是串联的，路径覆盖本质就是判定的组合
  - 如果判定不是串联改的，则需要根据程序流程图具体分析
- 优点
  - 较为彻底的一种测试
- 局限
  - 路径分支指数级增加
  - 不可达路径存在
  - 不能替代涉及条件测试的方法
- 分析
  - 考虑了各种判定结果的所有可能组合，但未必能覆盖判定中条件结果的各种可能情况
  - 他是一种较强的覆盖标准，但是不能替代条件覆盖，判定条件覆盖和条件组合覆盖标准

# $MC/DC$覆盖

- 定义
  - 每个判定都要取TF
  - 每个条件都要取TF
  - 判断中每个条件对判断取值是独立的 （在条件X以外条件不变条件下，X取值改变，判定取值改变就说X对判断取值独立）
- 覆盖测试用例数(n为条件数)
  - 最大：$2n$
  - 最小：$n+1$

# 流冬

- 基本路径测试
  - 思想：寻找基本路径，根据基本路径构造测试用例，保证每条路径至少执行一次
  - 优势
    - 更细的控制流描述---程序流图：同时刻画判定和条件对程序控制流的影响
    - 更少的覆盖路径---基本路径：近似的路径覆盖；数学和算法理论支撑
- 流图
  - 流图有叫程序图，用来描述程序中的逻辑控制流
  - 组成
    - 节点：一个或者多个过程语句
    - 边：控制流
    - 域：边和节点限定的区间
  - 基本结构




- 环复杂度---基本路径的度量
  - 说明
    - 环复杂度度量基本路径数，是所有语句被执行一次所需测试用例的上限
    - 具有最高环复杂度的模块蕴含错误的可能性最大，是测试中关注的焦点
  - 计算方法
    - 域的数量
    - $V(G)=P+1$ 判断节点+1
      （不适合存在多分支情形 一一一这种是按照出度为2计算的特别情况，所以多分支不在适用）
    - $V(G)=E-N+2$ E为边数，N为节点数
    - 新路径特点：贯穿路径；引入新的处理语句或者新判断
    - 说明：基本路径集合不唯一，但是路径数是确定的
- 基本路径测试分析
  - 基本路径的本质：每一条路径都是可以表示为边的向量
  - 基本路径的定义：是一组路径集合，任何其他的路径都是可以通过这些基本路径对应的向量线性运算得出

# 循环测试

- 类型

  循环处理
  - 简单循环的测试用例构造
    - 跳过整个循环
    - 只执行一次
    - 执行两次
    - 执行m（m<n）次
    - 执行n-1，n，n+1次（黑盒测试思想）
  - 嵌套循环的测试用例构造
    - 测最内层循环：所有外层循环变量置位最小值，最内层按简单循环处理
    - 由内向外，测试上一层循环：此层以外的层，循环变量设置为最小值；以内的循环变量设置为经典值；改成为简单循环测试
    - 重复上述，直到测试完成
    - 对各层循环同时取最小或者最大循环次数
  - 串接循环的测试用例构造
    - 独立：分别简单循环
    - 不独立：第一个为外循环，第二个为内循环

- 循环内部测试
  - 特点：循环取值影响循环内部控制流
  - 缺点：对循环体内部的测试可能会造成混淆
  - 推荐：单独测试内部模块

# 数据流测试

考虑从变量定义到其后变量使用之间的子路径

# 定义使用路径

- 注意
  - 这里的路径不一定贯穿程序
  - 定义节点中不一定出现相对应的变量（指针）
  - $dc-path$集是$du-path$集的子集
- 定义
  - 定义节点DEF 变量声明，初始化，分配内存空间；值改变
  - 使用节点USE 从内存中读取变量值并使用
  - 定义使用路径（du-path） $DEF$为开头，$USE$为结尾（路径中可有多个定义节点）
  - 定义清除路径（dc-path） $DEF$开头，$USE$结尾（路径中只有开头是定义节点）

—

# 白盒测试工具

## 测试工具分类

- 功能
  - 静态测试功能
    - 软件度量：复杂度分析
    - 代码分析：编程规范，代码优化，不可达路径，内存泄漏，数组越界
  - 动态测试功能
    - 运行时细节：模块运行时间
    - 覆盖率分析：动态白盒测试方法
    - 测试用例：自动生成测试用例
- IBM Rational
  - LogiScope
    - 改进代码，定位易出错模块
    - 提供图形化软件度量信息
    - 代码优化重构，降低维护成本
  - Purify
    - 检查内存泄漏
    - 检查数组越界
  - Quantify

        - 性能瓶颈分析：记录执行时间细节
        - 性能瓶颈解决：定位修改性能问题
        - 性能比较：比较不同修改版本性能
    - PureCoverage
        - 覆盖率统计
        - 自动程序插桩
- ParaSoft
- Xunit

# 第三讲 黑盒测试

## 黑盒测试基本概念

### 定义

- 黑盒测试：一种基于规格说明，不要求考察代码，以用户视角进行的测试。
- 功能测试，基于规格说明的测试

### 意义

- 检查明确需求和隐含需求
- 采用有效输出/输出和无效输入/输出
- 包含用户视角

### 目的

### 实施者

### 步骤

- 阅读规格说明书
- 设计测试用例
- 执行测试
- 分析测试结果

### 进入退出条件

## 黑盒测试方法基础

### 基于需求的测试（RTM）

### 目的

确认软件需求规格说明书列出的需求

#### 前提

- 需求规格已经过仔细评审
- 隐含需求明确化

### 需求跟踪矩阵（RTM）

- 可跟踪每个需求的测试状态而不会遗漏任何需求
- 优先执行优先级高的测试用例，尽早发现高优先级区域内缺陷
- 可导出特定需求对应的测试用例清单
- 评估测试工作量和测试进度的重要数据

## 正面测试和负面测试

### 正面测试

- 测试用例通过一组预期输入输出验证产品的需求
- 目的：证明软件对于每条规格说明和期望都能通过
- 正面测试用于验证已知测试条件，证明软件可以完成所期望的工作

### 负面测试

- 展示当输入非预期输入时产品没有失败
- 目的：使用产品没有设计和预想到的场景，尝试使系统垮掉
- 负面测试用于通过未知条件把产品搞垮

# 黑盒测试方法

## 等价划分

原理：

- 将软件输入和输出的值域进行划分，选择少数样本作为测试用例
- 实现对无穷输入量的科学规划，极大减少测试用例数目，降低测试工作量

等价类：如果软件行为对一组输入输出值来说是相同的，则称这组值为等价类

- 有效等价类，有意义的输入输出数据构成的集合
- 无效等价类：无意义的输入输出数据构成的集合

| 输入数据类型 | 划分等价类规则 | |
|---|---|---|
| 布尔值 | 1个有效等价类：TRUE | 1个无效等价类：FALSE |
| 连续取值范围 | 1个有效等价类：正确取值范围 | 2个无效等价类：大于和小于取值范围 |
| 数据个数 | 1个有效等价类：正确数据个数 | 2个无效等价类：大于和小于数据个数 |
| 集合 | 1个有效等价类：正确的集合取值 | 1个或多个无效等价类 |
| 需分别处理的输入数据 | 多个有效等价类：每个输入数据为1个等价类 | 1个无效等价类 |
| 符合某些规则的输入 | 多个有效等价类：符合某个规则的输入数据为1个等价类 | 若干个无效等价类 |

非冗余性：尽可能使等价类间没有交叉，保证测试用例不存在冗余

完整性：等价类的并集等于输入输出域

局限：忽略边界位置的测试用例

## 边界值分析

边界值分析原理：

- 软件的两个主要缺陷源：条件（变量取值决定控制流走向），边界（变量值的极限）
- 边界值分析

  能有效捕获出现在边界处的缺陷的一种测试方法，利用了缺陷更容易出现在边界的事实
- 缺陷易出现在边界处的原因

  使用比较操作符时未仔细检查

  循环和条件检查方法引起的困惑

  对边界附近需求的理解不够

独立参数的变量的边界值：

- 基本边界测试（4n+1）
- 健壮性边界测试（6n+1）

非独立参数的多变量的边界值

- 边界条件测试（3m）

## 因果分析法

因果图：将导致问题的结果划分成多种因素，并描述这些因素之间的关系，从而找出问题根源的复杂问题的分析工具

软件输入和输出之间存在逻辑关系

C表示原因，E表示结果

- 恒等（-）
- 非（~）
- 或（v）
- 与（^）

4种输入约束

- 互斥（E）：多个原因不能同时成立，最多有一个能成立
- 包含（I）：多个原因中至少有一个必须成立
- 唯一（O）：多个原因必须有且仅有一个成立
- 要求（R）：当$c_1$成立，则$c_2$必须成立

一种输出约束

- 屏蔽

总结：

- 适用于输入输出存在因果关系的测试
- 适用于输入输出参数取值为2值的测试
- 是对输入的组合测试
- 因果图保证各种组合不被遗漏，能消除违反约束的组合

## 决策表

适用情形：输入输出较多且输入之间和输出之间相互制约的条件较多

决策表：把作为条件的所有输入的组合以及对应输出都罗列出来形成表格

特点：能将复杂问题按照各种可能情况全部列出，表示简明，避免遗漏

4种成分：

- 条件桩：列出所有可能问题
- 条件项：列出条件所有可能取值
- 动作桩：列出可能采取的操作
- 动作项：指出在条件项的各种取值情况下应采取的动作

决策表化简：合并相似规则

- 有两条或以上规则具有相同动作
- 且在条件项之间存在极大相似性

## *正交数组*

从大量测试用例中挑选适量的，有代表性的用例，从而合理地进行测试的方法

适用于输入域相对较少而详尽测试的次数又过大的问题

特点：输入域有限，参数取值有限，需要将参数取值进行排列组合

正交表：

- 因子：影响结果的条件因素（输入参数）
- 因子数：正交表中的列数，既要测试的功能点数

- 水平：因子的取值
- 水平数：单因子的取值个数
- 行数：正交表行数，即测试用例的个数

正交表的正交性：

- 整齐可比性：每个因子的每个水平出现的次数是完全相同的
- 任意两列（两个因子）的水平搭配（横向形成的数字对）是完全相同的

正交表选择原则：

- 正交表因子数>=实际因子数
- 正交表每个水平数>=实际每个水平数
- 如果出现2个或者2个以上正交表符合以上条件，则选择行数最少的正交表

# 黑盒测试工具

## 测试工具原理

黑盒测试工具==功能测试工具

功能：

- 录制（测试输入的自动化），回放（重新执行测试用例）
- 检验：设置检测点，对用例执行的正确性进行检查
- 可编程性：脚本更加灵活，功能更强大

原理：运行被测试软件的同时，捕获过程中的键盘，鼠标操作，生成脚本文件，这个脚本文件可以进行修改和回放

## 作用

- 创建功能和回归测试
- 自动捕获，验证和重放用户的交互行为
- 为每一个重要软件应用和环境提供功能和回归测试自动化的行业最佳解决方案

# 第四讲 单元测试与集成测试

# 单元测试

## 基本概念（软件单元、定义、意义、目标、实施者、关注点）

- 软件单元：一个应用程序中的最小可测部分，面向过程中的单元如独立的程序、函数、过程、网页、菜单，面向对象中的单元如类（基类、抽象类、子类）
- 单元测试/模块测试定义：对最小的软件设计单元（模块/源程序单元）的验证工作
- 意义：
    - 消除软件单元本身的不确定性
    - 其他测试阶段的必要的基础环节

- 实施者：软件开发人员
- 目标：
  - 单元体现了预期的功能（黑盒 白盒）
  - 单元的运行能覆盖预订的各种逻辑（白盒）
  - 单元工作中内部数据能够保持完整性（白盒）
  - 可接受正确数据，也能处理非法数据（黑盒）
  - 在数据边界条件上，单元能正确工作（黑盒）
  - 单元的算法合理，性能良好（黑盒）
  - 扫描单元代码未发现任何安全性问题（白盒）
- 关注点：
  - 包括模块功能、内部逻辑处理、数据结构、接口、边界条件、独立路径、错误处理、性能、安全等
  - 模块或构建接口，目标：进出模块/构建的数据流正确，关注点：（1）接口名称、参数个数、类型、匹配顺序（2）输出或返回值及其类型是否正确
  - 局部数据结构，目标：数据在模块执行过程中维持完整性和正确性，关注点：（1）数据结构定义和使用过程的正确性（2）局部数据结构对全局数据结构的影响
  - 边界条件,目标:保证模块在边界条件上能够正确执行，关注点:(1)数据结构中的边界（如数字a[n])(2)控制流中的边界（如循环次数、判断条件）
  - 独立路径,目标：保证模块中的每条独立路径（基本路径）都被覆盖，使得所有语句都被执行一次，关注点：对路径的选择性测试（基本路经测试+循环测试)
  - 处理错误的路径，目标：保证错误处理的正确性，软件的健壮性，关注点：（1）异常条件的正确性（2）异常的可发生性（3）异常处理的逻辑正确性

## 单元测试规程（驱动器和程序桩)

- 单元测试通常与编码工作结合起来进行，模块本身不是一个独立的程序，在测试模块时，必须为每个被测模块开发一个驱动器（driver）和若干个程序桩（stub）
- 驱动器（Driver）：对底层或子层模块进行测试时所编制的调试被测模块的程序，用以模拟被测模块的上级模块
- 桩程序（Stub）：对上层模块进行测试时，所编制的替代下层模块的程序，用以模拟被测模块工作过程中所调用的模块
- 驱动器的一般结构：数据说明;初始化；输入测试数据；调用被测模块；输出测试结果；停止;
- 桩程序的一般结构：数据说明；初始化；输出提示信息（表示进入了哪个桩模块）；验证调用参数；打印验证结果；讲模拟结果送回被测程序；返回
- 手工测试：按照需求规格设计测试用例完成测试，静态测试
- 自动测试：自动化方法的效果：有效验证较为独立单元的正确性

# 集成测试

## 概念

- 集成测试：把单独的软件模块结合在一起作为整体接受测试
- 意义：
  - 验证软件单元间能否协调工作
  - 验证单元结合的功能、性能和可靠性需求
- 技术：黑盒测试技术为主，白盒测试技术为辅
- 步骤：与集成测试策略相关

- 目标：集成测试用来构造程序并实施测试以发现于接口连接有关的错误，它把经过单元测试的模块拿来，构造一个在设计中所描述的场景

## 接口

- 内部接口：产品内部两模块之间的通信接口
- 外部接口：产品之外第三方可以看到的接口
- 接口提供方式：API、SDK
- 显示接口：写入文档的明确化的接口
- 隐式接口：未写入文档，只有开发人员知道，如Windows的隐藏API

## 瞬时集成测试

又称为大爆炸测试策略，其特点是：当所有构建都通过单元测试，就把他们组合成一个最终系统，并观察它是否正常运转，适用于小型软件开发

- 缺陷：
  - 无休止的错误
  - 模块一次性结合，难以找出错误原因
  - 接口错误和其他错误容易混淆

## 增量集成测试

- 特点：讲程序分成小的部分进行构造和测试
- 优点：
  - 错误容易分离和修正
  - 接口容易进行彻底测试
- 缺点：会有额外开销，但能大大减少发现和修正错误的时间
- 三种增量集成测试：
  - 自顶向下集成
    - 顺序：先集成主模块（主程序)，然后按照控制层次向下进行集成
    - 集成方式：深度优先、广度优先两种
    - 步骤：1.主程序作为测试驱动器 2. 根据集成顺序，程序桩逐渐被各模块替换 3.每个模块集成时都需要进行 测试 4.每完成一次测试之后，将新模块替换程序桩 5.利用回归测试来保证没有引进新的错误
    - 优点：1.尽早发现高层控制和决策错误2.只需要一个驱动器3.每步只增加一个模块4.支持深度优先和广度优先
    - 难点：高层测试需要在低层测试完成后才可进行的情形
    - 难点解决方法：1.推迟测试，直到低层模块完成测试 2.设计程序桩，模拟低层模块 3.自底向上测试
    - 缺点：1.对低层模块行为的验证比较晚 2.需要编写额外程序模拟未测试的模块 3.测试用例的输入和输出很难明确表示
  - 自底向上集成
    - 顺序：从原子模块，即程序最底层模块开始就行，构造并进行集成测试。原子模块->造件（build）->应用软件系统
    - 步骤：1.低层模块组成实现特定子功能的构件 2.编写驱动器程序协调输入输出 3.测试特定子功能的构件 4.盐层次向上对构件进行组合

        - 优点：1. 今早确认低层行为 2.无需编写程序桩 3.对实现特定功能的书容易表示输入输出
        - 缺点：1.推迟确认高层行为 2.需编写驱动器 3.组合子树时，有许多元素要进行集成
    - 混合式集成
        - 顺序：综合自顶向下和自底向上，是实际测试中的使用集成测试策略
        - 特点：开发小组对各自的低层模块向上集成；专门的集成小组进行自顶向下集成
        - 步骤：1.用程序桩独立测试上层模块 2.用驱动器独立测试低层模块 3.集成式对中间层进行测试
    - 比较：

| | 优点 | 缺点 |
|---|---|---|
| 自顶向下 | 1.如果主要的缺陷发生在程序的顶层将非常有利<br>2.一旦引入I/O功能，提交测试用例会更容易<br>3.早期的程序框架可以进行演示，并可激发积极性 | 1.必须开发桩模块<br>2.桩模块要比最初表现的更复杂<br>3.在引入I/O功能之前，向装模块引入测试用例比较困难<br>4.创建测试环境可能很难，甚至无法实现<br>5.观察测试输出很困难<br>6.会导致特定模块测试的完成延后 |
| 自底向上 | 1.如果主要的缺陷发生在程序的底层将非常有利<br>2.测试环境比较容易建立<br>3.观察测试输出比较容易 | 1.必须开发驱动模块<br>2.直到最后一个模块添加进去，程序才形成一个整体 |

- 集成测试方法的选取

| 因素 | 建议使用集成方法 |
|---|---|
| 设计和需求清晰 | 自顶向下 |
| 需求、设计和体系结构不断变化 | 自底向上 |
| 体系结构改变、设计稳定 | 混合 |
| 体系结构较小，规模小 | 瞬时集成 |
| 体系结构复杂，软件规模大 | 混合、端-端 |

# 测试插桩

测试插桩原理：在程序特定部位附加一些操作或功能，用来检验程序运行结果以及执行特性，以便支持测试

## 黑盒插桩

- 作用：
    - 测试预言
    - 随机数据生成

- 测试预言插桩
  - 定义：一种独立于被测形同的程序或机制，用于确认对于给定的输入，系统是否有一个给定输出
  - 特点：不能获得源代码，但可以调用API等信息，不属于纯黑盒测试，而是一种灰盒测试
  - 作用：检查函数正确性、系统安全性
- 随机数据生成器（随机测试技术）
  - 定义：一种在可能输入集中选取一个任意子集进行测试的技术
  - 作用：避免只测试所知道的将奏效的场景
  - 有效测试用例生成方法：1.软件输入域进行等价划分 2.各子域边界进行边界值选取 3.各子域内进行随机测试选择输入样本

## 白盒插桩

- 作用：
  - 生成特定状态，检验状态的可达性
  - 显示或读取内部数据或私有数据
  - 监测不变数据
  - 监测前提条件
  - 认为触发事件
  - 监测事件事件
- 插桩方式一：目标代码插桩
  - 优点：目标代码的格式主要和操作系统相关，和具体的编程语言和版本无关，应用范围广；特别适合需要对内存进行监控的软件中
  - 缺点：目标代码中语法、语义信息不完整
- 插桩方式二：源代码插桩
  - 优点：词法分析和语法分析的基础上进行的，这就保证对源文件的插桩能够达到很高的准确度和针对性
  - 缺点：工作量较大，而且随着编码语言和版本的不同需要做一定的修改
- 常用插桩策略：
  - 语句覆盖插桩（基本块插桩）：在基本块的入口和出口处植入探针
  - 分支覆盖插桩：在每一个分支开始处植入探针
  - 条件覆盖插桩：在每个条件表达式的布尔表达式处植入探针

## 插桩作用(见上)

# 第五讲 JUnit测试工具（LL重点上没有，WP课件上没有）

## JUnit基础

### 概念

Java 最著名的测试框架之一

## JUnit使用

## JUnit4.X（元数据、断言、失败与错误、测试固件等）

元数据：@Test @Before @After @Paramenters等

断言：assertEquals 、assertFalse、assertTrue 、assertSame 等断言方法

失败与错误：Faliure一般时有单元测试所使用的断言方法判断失败所引起的，Error通常是由代码本身的错误引起

# 第六讲 系统测试、确认测试和回归测试

## 系统测试

### *概念*

#### 定义

对完整集成后的产品和解决方案进行测试，用来评价系统对具体需求规格说明的功能和非功能的符合性的测试

在集成测试之后进行

#### 意义

是既测试产品功能也测试产品非功能的唯一测试阶段

#### 目的

- 发现可能难以直接与模块或接口关联的缺陷
- 发现产品设计，体系和代码的基础问题（产品级缺陷）

#### 实施者

独立测试团队

## 功能测试

- 设计/体系结构测试

  对照设计和体系结构开发和检查测试用例，从而整理出产品级测试用例

- 业务垂直测试

  针对不同业务纵深的产品，根据业务定制测试用例，验证业务运作和使用

  方法：

  - 模拟和复制
  - 场景测试

- 部署测试

  验证新的系统是否能有效替代老的系统

    - 目的：特定产品版本短期内是否能成功使用
    - 阶段：
        - 采集实际系统真实数据，建立镜像测试环境，重新执行用户操作
        - 引入新产品，重新执行用户操作
- Alpha测试和Beta测试

  Alpha测试：用户在开发环境下进行的受控测试。其特点是不由程序员或测试员完成，但开发者会在现场在Alpha测试达到一定程度后进行Beta测试

  Beta测试：用户在实际使用环境下进行测试，一种可以把待测产品交给客户收集反馈意见的机制。其特点是开发者通常不在现场。挑战是客户数量以及客户充分了解产品

- 符合性的认证，标准和测试
    - 主流基础设施：操作系统，硬件，数据库
    - 约定和法律要求：质量行业标准，法规

## 非功能性测试

最大挑战：设值配置

- 性能测试
- 互操作性测试/兼容性测试

  目的：保证两个或多个产品可以交换和使用信息，并恰当地在一起运行。如兼容操作系统，数据库，浏览器等

  后向兼容：老版本中的数据能否在当前版本中使用

  前向兼容：当前版本中的数据能否在未来版本中使用

- 可使用性与易获得性测

  可用性：以用户视角确认产品的易用性，速度和美感的测试

  可用性质量因素：可理解性，一致性，导航，响应

  易获得性：检查产品对于行动不便的用户也可使用的测试

  基本易获得性：粘贴键，声响键，多种模拟鼠标功能，朗读器，触摸

- 国际化测试

  目的：确保软件支持不同国家语言的测试手段

  国家化问题：语言的消息显示，语言和习俗的消息处理，日期格式，货币格式

# 确认测试

- 定义：检测产品是否满足在项目的需求阶段定义的确认准则，或者说是否具备在真实环境中使用的条件
- 实施者：客户或客户代表
- 引入时机：系统测试之后
- 确认准则：
    - 产品确认

      对现有测试用例进行分类形成确认准则
    - 规程确认

文档，使用培训

- 服务等级约定

# 回归测试基础

## 概念

回归测试是对之前已经测试过，经过修改的程序进行重新测试，以保证该修改没有引入新的错误或者由于更改而发现之前未发现的错误

回归测试要保证增强型或者改正型修改使软件正常进行并且不影响已有的功能

## 组测试

- 原理：一个模块可能单独工作正常，多个模块集成工作时却出现错误；组测试是寻找此类错误的有效方法

## 波及效应

波及效应是为了发现所有受影响部分和发现潜在的受影响部分，以保证软件发生改变后仍然保持一致性与完整性

- 需求的波及效应
- 设计的波及效应
- 代码的波及效应
- 测试用例的波及效应

# 第七讲 性能测试

# 性能测试基础

## 概念

- 狭义性能测试：通过模拟生产运行的业务压力或用户使用场景来测试系统的性能是否满足生产性能要求。
- 广义性能测试：压力测试、负载测试、强度测试、并发测试、大量数据测试、配置测试、可靠性测试等性能相关测试的统称。

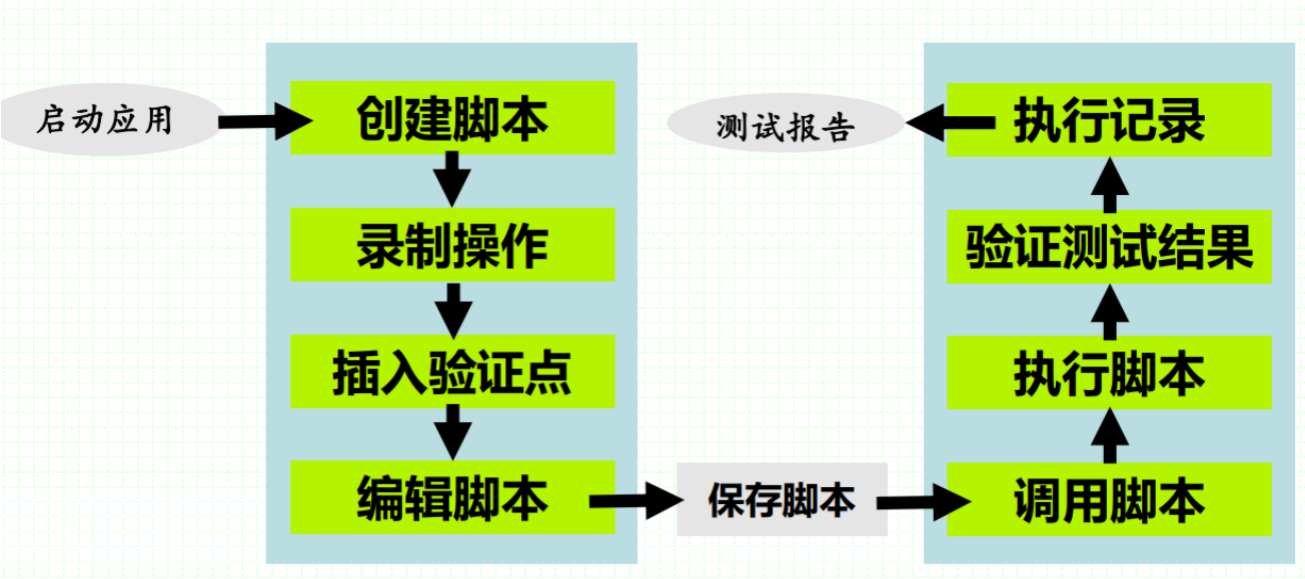
## 各种广义性能测试的理解

- 压力测试：通过对程序施加越来越大的负载，知道发现程序性能下降的拐点——强调压力大小变化。
- 负载测试：不断增加压力或者增加一定压力下持续一段时间，知道系统性能指标达到极限——强调压力增加和持续时间。
- 强度测试：迫使系统在异常资源下运行，检查系统对异常情况的抵抗力——强调极端的运行条件。
- 并发测试：多用户同时访问或操作数据时是否存在死锁或其他性能问题——强调对多用户操作的承受能力。
- 大数据量测试：在存储、传输、统计等业务中结合并发操作测试系统的数据处理极限——钱掉数据处理能力。
- 配置测试：通过测试找打系统各项资源的最优分配原则——钱掉资源优化配置
- 可靠性测试：在给系统加载一定压力的情况下，使系统运行一段时间，以此检验系统是否稳定——强调系统稳定性。

## 性能测试的指标

- 并发用户数量
  - 狭义并发：多有用户在同一时刻做同样的操作
  - 广义并发：多个用户同时进行操作，但这些操作可以相同也可以不同
  - 虚拟用户：模拟真实用户操作来使用软件
- 响应时间：
  - 场景（Scenario）：在一个性能测试起见一侧发生并实现特定业务流程的事件总和
  - 事务（Transaction）：用户业务流程
  - 思考时间（Think Time）用户操作过程中两个请求的间隔时间
- 吞吐量：
  - 一次测试过程中网络上传输的数据总和
  - 吞吐率=吞吐量/传输时间
- TPS（Transaction Per Second）：每秒处理事务数
- 点击率（Hit Per Second）：每秒用户提交的HTTP请求数
- 资源利用率：CPU、磁盘、网络、数据库利用率

## 自动化测试原理




- GUI对象识别：
  - 获得用户界面上对象的类别、名称、属性值
  - 常见的GUI对象：窗口、按钮、菜单、列表、编辑区域等
- DOM对象识别：DOM（Document Object Model）文档对象模型可以捕获Web也买拿上的各种对象
- 其他对象技术：C#/Java语言中的反射机制
- 自动比较技术：
  - 简单的数字、文字比较
  - 复杂的图像比较
- 脚本技术：
  - 脚本通过录制测试的操作而产生：击键、移动、录入数据
  - 录制得到的脚本可被进一步编程

# LoadRunner

## 架构（四个组成部分）

- 虚拟用户发生器
- 压力调度和监控中心
- 压力产生器
- 压力结果分析工具

## 工作原理

- 控制台控制和加载所有虚拟用户
- 模仿大量真实用户对系统进行测试
- 实时监控系统把系统的性能、服务器状态和网络信息反馈给控制台
- 测试结果被生成报表并保存，留待对系统性能进行分析时使用

## 测试步骤

- 制定性能测试计划
- 录制并修改Vuser脚本（事务、参数化、关联）
- 方案场景描述测试活动中发生的各种事件（集合点）
- 运行场景并实时监测
- 分析数据，生成性能·评估报告

## 脚本录制与开发（事务、参数化、集合点、检查点、关联）

- 参数化
    - 目的：模拟真是的用户操作和创建显示的结果
    - 作用：虚拟用户脚本更接近真实用户行为，减少脚本大小与数量
- 集合点：控制各个Vuser在统一时刻执行任务，模拟真实意义上的并发
- 检查点：检查网页结果是否正确（是否存在制定Text或Image）
- 关联
    - 意义：
        - 可以将脚本中某些写死的代码数据转变为动态的每次都不一样的数据
        - 将一条语句的结果作为另一条语句的输入
    - 作用：
        - 简化或者优化代码
        - 用于动态数据
        - 容纳唯一数据记录
    - 方法：
        - 手动关联
        - 录制后自动关联
        - 录制过程自动关联

## 场景创建与执行

## 结果分析

- 内存分析方法

- 处理器分析方法
- 磁盘I/O分析方法
- 网络分析方法