软件测试与开发

软件测试基础

软件概述

1.1.1软件的生命周期

问题定义-->需求分析-->软件设计-->软件开发-->软件测试-->软件维护

1.1.2 软件开发模型

六款软件开发模型

1. **瀑布模型**：划分了清晰的检查点，一个阶段完成后直接专注于开发，有利于提高开发效率

强调文档的作用，线性过程太理想化，不适合现代软件工程

- 各个阶段的划分完全固定，阶段之间的大量文档极大增加工作量
- 开发是线性的，早期错误可能要等到后期测试才能发现，增加风险

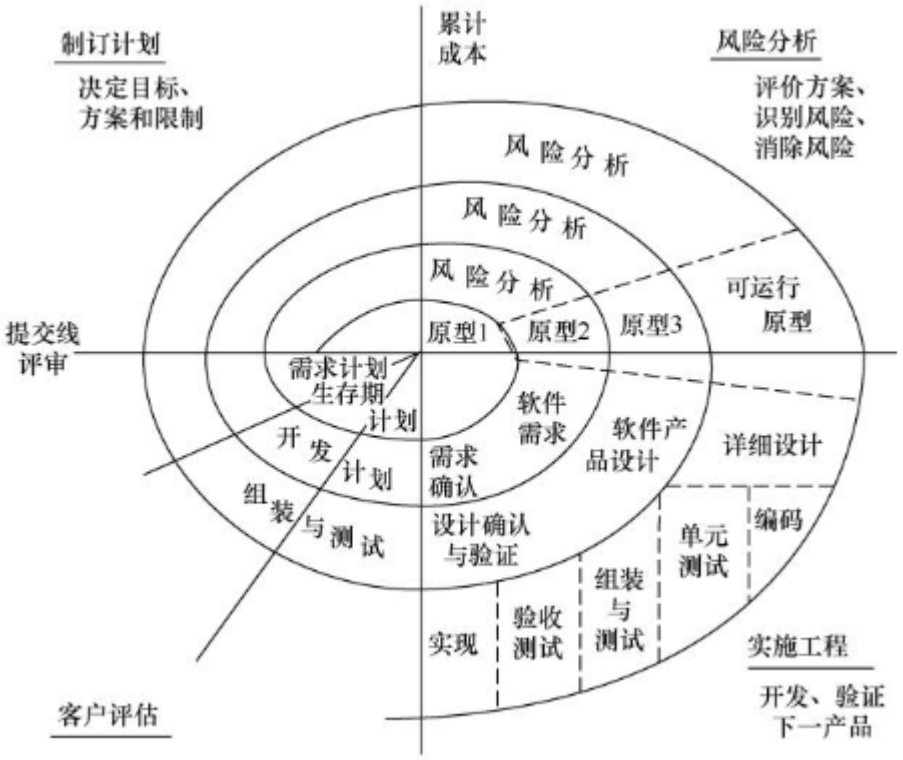
2. **快速原型模型**：建造一个快速原型，实现用户与系统交互，然后用户对原型评价，精细化开发需求

   - 规避了需求不明带来的风险，适合不能预先确定需求的软件项目
   - 快速原型模型准确设计出软件原型存在一定难度，这种模型也不利于开发人员对产品扩展

3. **迭代模型**：增量模型或者演化模型，软件被作为一系列增量构件来设计实现集成和测试，它将完整软件拆成不同组件，对每个组件进行开发测试

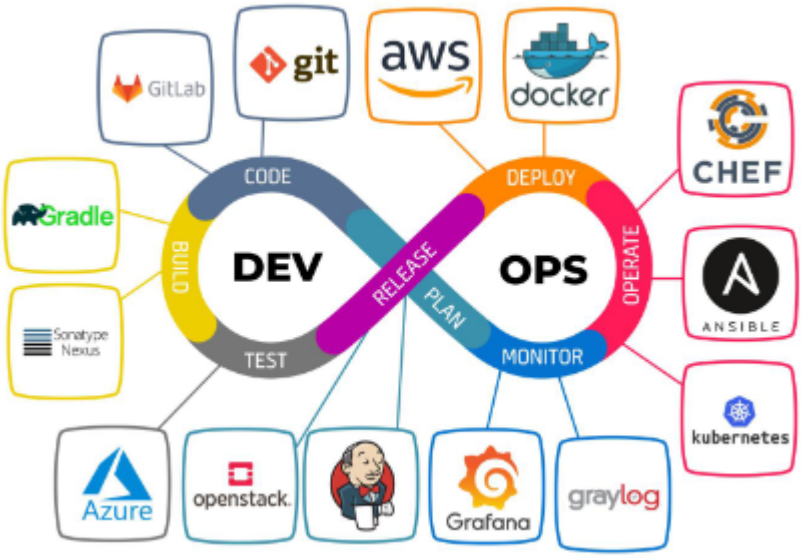
   - 能很好适应用户需求变更，以组件形式逐个交付，降低软件开发成本
   - 需要软件具备开放式体系结构
   - 容易退化为边做边改的开发形式，从而失去对软件开发过程的整体控制

4. **螺旋模型**：融合了瀑布模型和快速原型模型，引入风险分析，适合开发复杂的大型软件



- 制定计划：确定软件目标
- 风险分析：评价所制定的实施方案，识别并消除风险
- 用户评估：开发产品并验证
- 实施工程：用户对产品进行评估，提意见，制定下一步计划

5. **敏捷模型**：以用户需求进化为核心，循序渐进开发（迭代）
   - 开发不是线性，开发同时也进行测试，甚至能提前写好测试代码
   - 提出需求的用户可以全程参加到开发，以适应软件需求变更频繁
   - 由于项目越大开发人员越多，敏捷模型适合小型项目开发
     Scrum 是一个开发管理框架，一般会选一个产品负责人全面负责开发过程，Kanban 开发方式将工作细分成任务，显示在"看板"上，每个人能了解自己工作进度以及任务
6. **DevOps**：开发与运维组合词，是一组过程、方法、系统的统称，同时也是一种文化理念、实践和工具集



生命周期

- 规划（PLAN）：确定下一个版本软件新特性和功能需求
- 编码（CODE）
- 集成（BUILD）：新代码被集成到代码库，然后进行测试打包，为发布和部署准备
- 测试（TEST）：进行各类测试，包括单元、集成、系统、用户验收测试
- 发布（RELEASE）：正式可用状态交付给用户或投入生产
- 部署（DEPLOY）：经过测试的软件部署到生产环境
- 运维（OPEERATE）：对生产环境监控维护
- 监控反馈（MONITOR）：持续监控，收集用户数据、反馈，为下一轮开发提供依据形成闭环

  工具链
- 版本控制：git，提供分支管理、合并、重写存储历史记录等
- 构建工具：Jenkins，用于自动构建软件项目
- 配置管理工具：Ansible、Chef，用于自动化配置服务器和管理基础设施
- 容器化平台：Docker，用于创建、打包、分发应用程序即环境依赖
- 监控和日志：Prometheus（监控系统指标） 和 ELK Stack（收集存储分析日志数据）

  关键特性：
- 强调跨职能团队之间开发沟通和协作，打破开发、测试、运维等之间壁垒
- 持续集成：允许多个开发人员频繁将代码集成到共享存储库，合并时自动构建和测试
- 持续交付：确保软件随时可以部署到生产环境，无需大量人工干预
- 支持敏捷方法论，快速迭代开发，快速响应需求变化

软件质量概述

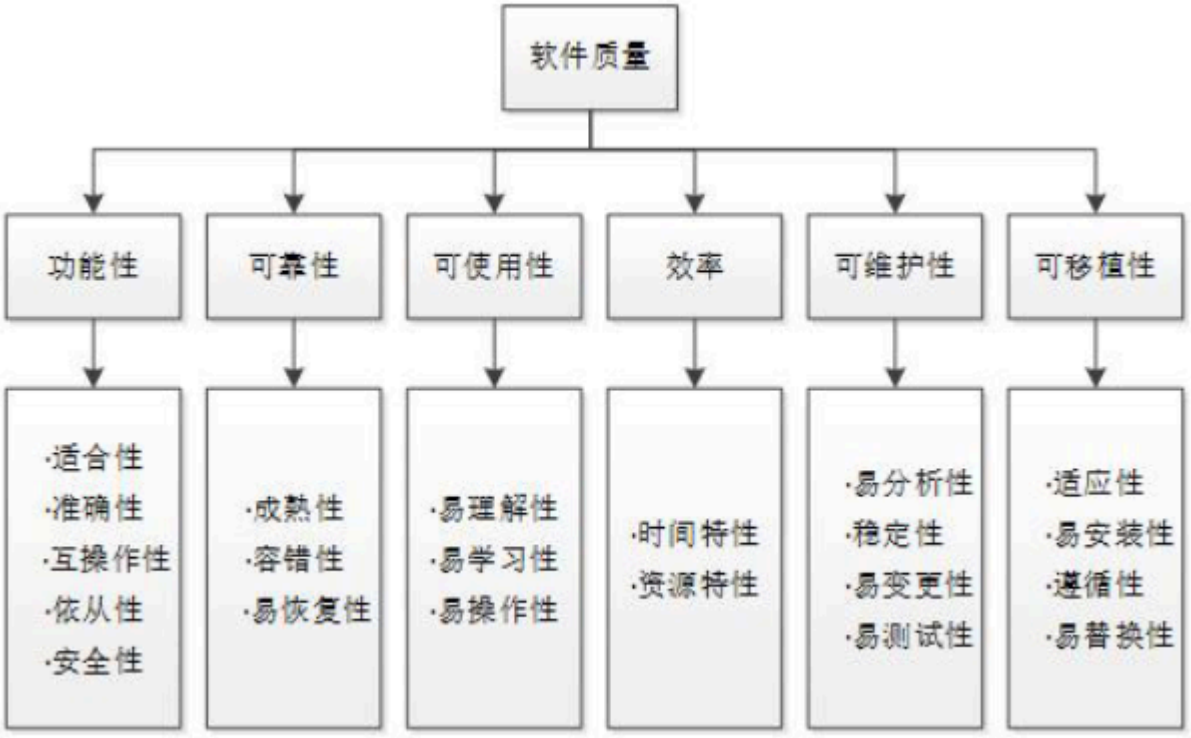
软件质量的概念：指软件产品满足基本需求和隐式需求的程度

- 满足需求规定：符合开发者明确给定的目标，并能可靠运行
- 满足用户基本需求：解决用户实际问题
- 满足用户隐式需求：能极大提升用户满意度

  软件质量模型

  SO/IEC 9126:1991 是一个通用的评价软件质量的国际标准，包括测试计划攥写、测试用

例设计等



- 兼容性：产品与其他系统或产品共存和协同工作能力
- 安全性：保护信息和资产免受未经授权的访问、使用、披露、中断、修改或销毁的能力

影响软件质量的因素

1. 需求模糊
2. 软件开发人员问题：人员流动性大导致软件开发前后不一致
3. 软件开发缺乏规范性文件指导
4. 缺乏软件质量的控制、管理

软件的缺陷管理

1.2.1 软件缺陷的概念

- 未达到说明书中标明的功能
- 出现了说明书中不该出现的功能
- 功能超出了说明书指明的范围
- 未达到说明书中应该达到的目的
- 软件难以理解和使用，运行速度慢、用户认为不好

原因：

- 需求不明确
- 软件结构复杂
- 编码问题（水平参差不齐）
- 项目期限短
- 使用新技术

1.2.2软件缺陷的分类

- 测试种类划分
  - 界面缺陷
  - 功能缺陷
  - 性能缺陷
  - 安全性缺陷
  - 兼容性缺陷
- 严重程度划分
  - 严重缺陷
  - 一般缺陷
  - 次要缺陷
  - 建议缺陷
- 优先等级划分
  - 立即解决缺陷
  - 高优先级缺陷
  - 正常排队缺陷
  - 低优先级缺陷
- 发生阶段划分
  - 需求阶段缺陷
  - 设计阶段缺陷
  - 编码阶段缺陷
  - 测试阶段缺陷

1.2.3缺陷处理流程

1. 提交
2. 分配
3. 确认
4. 拒绝/延期（发现不是一个真正的缺陷）
5. 处理
6. 复测
7. 关闭（正确修改后将缺陷关闭）
   软件缺陷报告：缺陷 ID、类型、严重程度、优先级以及测试环境

1.2.4 缺陷管理工具

1. Bugzilla：建立一个完整的缺陷跟踪体系，包括跟踪、记录、报告、解决情况等
2. 禅道：功能完备的项目管理软件
3. Jira：基于 Java 架构的管理工具

软件测试概述

1.3.1 软件测试定义

特定条件下运行系统或者构件，观察或记录结果，对系统某方面做出评价，分析某个软件发现现存的和要求的条件的差别并评价此软件的特性

1.3.2 软件测试目的

1. 从开发角度：帮助开发人员找到开发过程中存在的问题，预防缺陷产生
2. 从软件测试角度：使用最少的人力物力时间，找到隐藏缺陷
3. 从用户需求角度：检验软件是否符合用户需求

1.3.3 软件质量保证定义

贯穿软件项目整个生命周期，遵循对应的标准及规范要求，且产生了合适的文档和精确反映项目情况的报告
这个活动主要包括评审项目过程、审计软件产品、就项目是否真正遵循已经制定的计划、标准、规程等，给管理者提供可视性项目和产品可视化管理报告

1.3.4 软件质量保证和软测区别

质量保证：预防质量问题，关注过程正确，通过规范过程来保证结果质量
软件测试：发现质量问题，关注结果验证，通过检查产品来发现已有缺陷

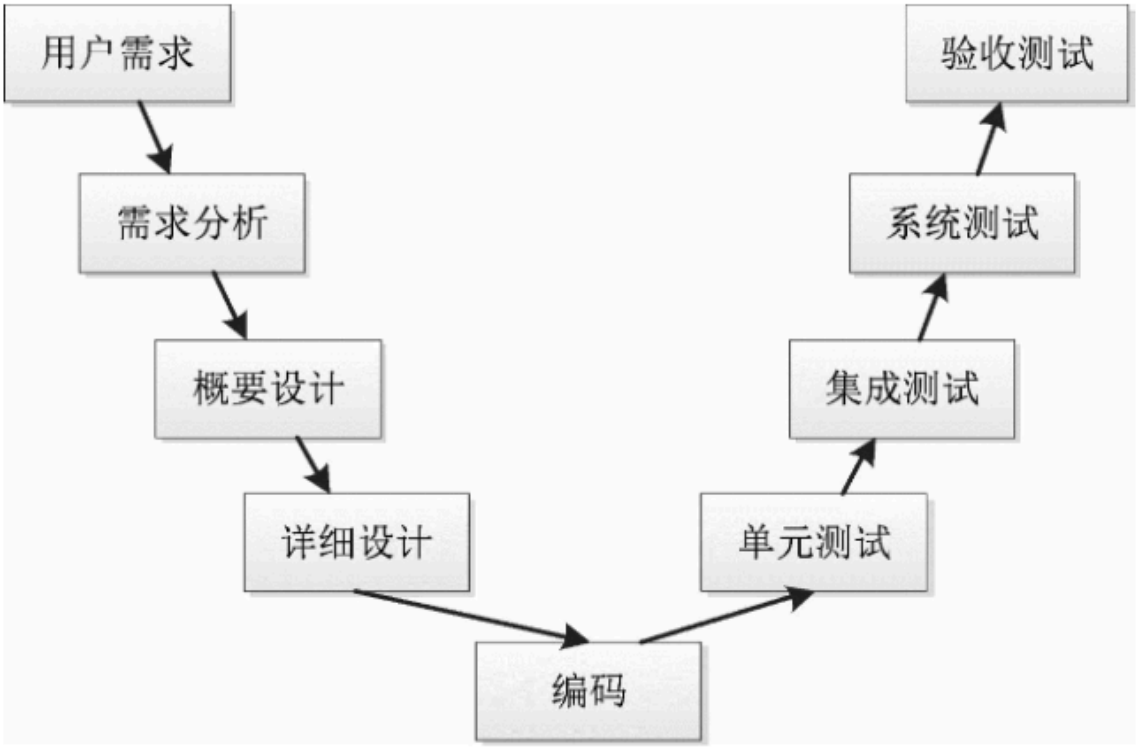
1.3.5 软件测试的分类

1. 按照测试阶段：
    1. 单元测试（开发人员自测）
    2. 冒烟测试（对系统基本功能简单测试）
    3. 集成测试（验证软件是否满足设计需求）
    4. 系统测试（与数据库或者硬件组合一起测试）
    5. 验收测试（按照说明书测试）
2. 测试技术分类
    1. 黑盒测试：又称功能测试、数据驱动测试，把软件当成黑盒子，只要输入数据能输出预计结果，侧重功能性需求
    2. 白盒测试：又称透明盒测试、结构测试，按照程序执行路径得到结果，测试人员清楚内部每一步
    3. 灰盒测试：介于两者之间，由方法和工具组成，不仅要关注输入输出还要关注程序内部
3. 软件质量特性分类
    1. 功能测试
    2. 性能测试
4. 自动化程度分类
    1. 手工测试
    2. 自动化测试
5. 测试项目分类
    1. 界面测试

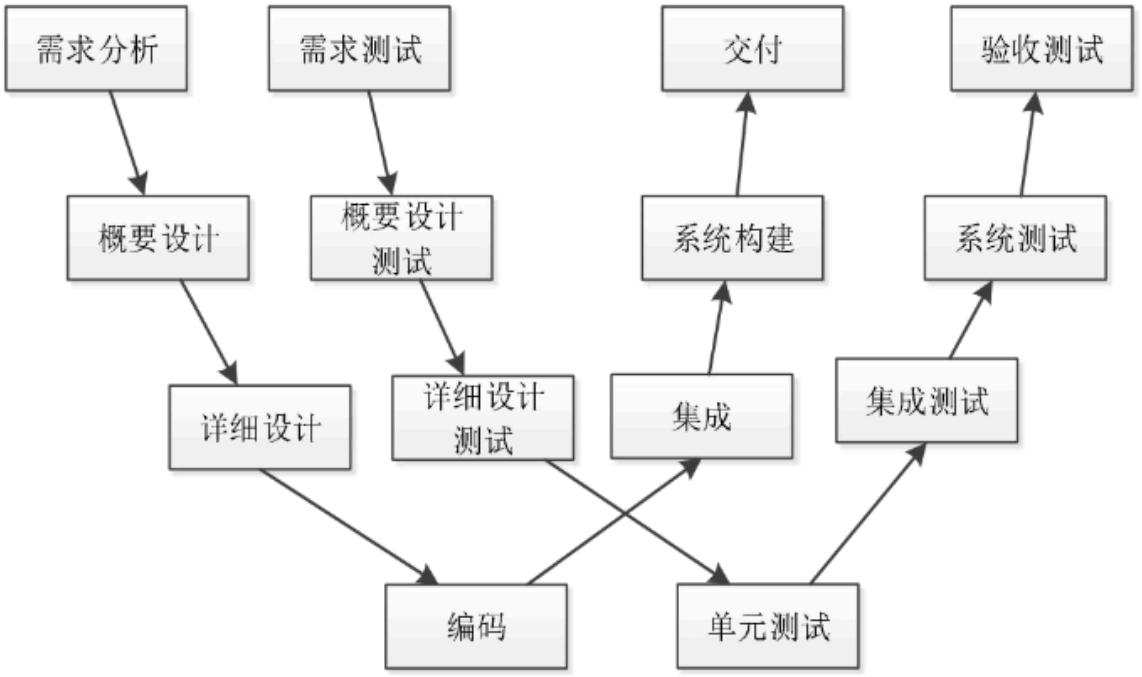
    2. 安全性测试
    3. 文档测试
6. 其他
    1. α测试：初版测试
    2. β测试：上线之后的版本测试
    3. 回归测试：确定原有缺陷已经消除并没用引入新的缺陷
    4. 随机测试：根据经验进行功能和性能抽查

软件测试过程

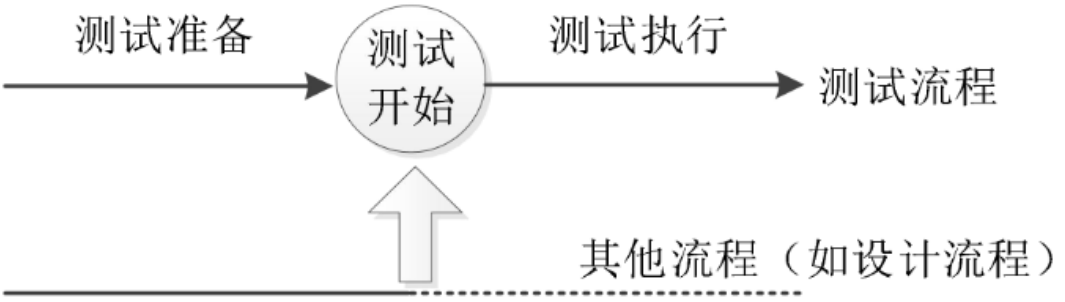
1. V 模型：每个测试阶段与开发阶段对应，但是不能发现前期阶段产生的错误



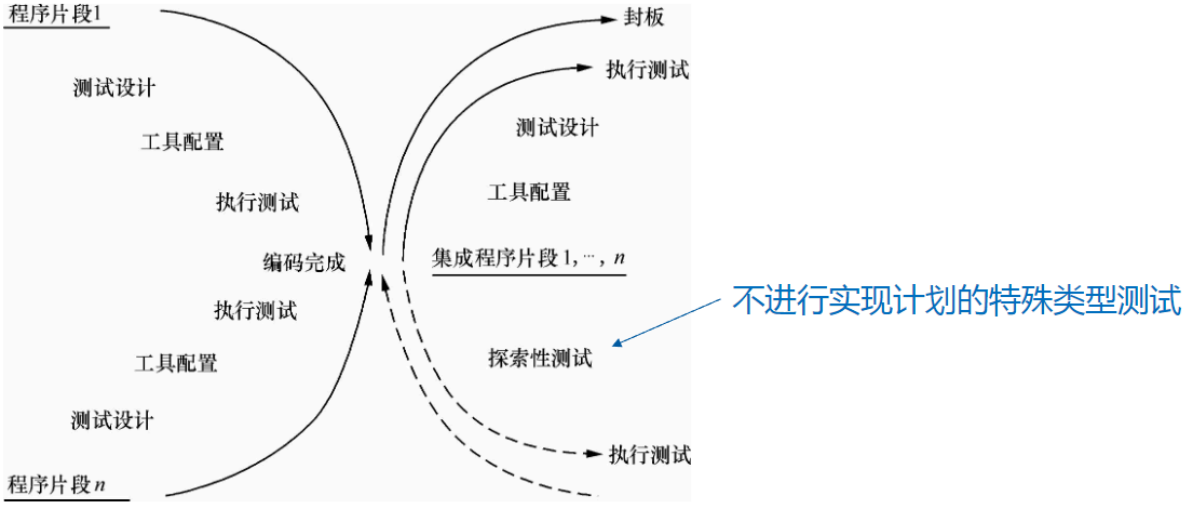
2. W 模型：测试开发同步进行，但是只有在上一步结束之后才能开始下一步行动，无法支持迭代、自发性以及变更调整的项目



3. H 模型：将活动完全独立出来，形成一个完全独立的流程



4. X 模型：将程序分成多个片段反复迭代测试，再将片段集成测试



一般结合 W 模型和 H 模型工作，软件各个方面测试以 W 模型为主，测试周期、测试计划、进度以 H 模型为指导，X 模型一般是最终测试

软件测试原则

1.5.1 六点原则

1. 测试基于用户需求
2. 测试要尽早进行
3. 不能做到穷尽测试（选有代表性的）
4. 遵循 GoodEnough 原则，只要足够好就行
5. 测试缺陷要符合"二八"定理（80%的缺陷集中在 20%的模块中，提高针对性）
6. 避免缺陷免疫（不能因为经常遇到某个缺陷就忽视它们）

1.5.2 软件测试停止原则

1. 测试超过了预定时间
2. 执行完所有测试发现没有故障
3. 用特定测试用例方法作为判断测试停止的基础
4. 正面指出测试完成要求，比如发现并修改 20 个软件故障
5. 根据单位时间查出来的故障数量决定

软件测试流程

1.6.1 测试流程

1. 分析测试需求：需求是否完整、准确、无歧义
2. 制定测试计划：测试范围、测试策略、测试资源安排、测试进度安排、预估测试风险
3. 设计测试用例
4. 执行测试
5. 编写测试报告

黑盒测试方法

黑盒测试检测范围：程序能否按照说明书正常使用，功能是否有遗漏，性能等特性要求是否满足；检测人机交互、数据结构等，程序能否正常接收输入数据并输出正确结果，并保持外部信息（数据库或文件）完整；检测初始化或终止是否有错

优点：

- 简单，不用了解代码
- 与内部实现无关
- 从用户角度出发能知道用户会用到哪些功能，遇到什么问题
- 基于软件开发文档，能知道实现了哪些功能
- 根据测试用例针对性寻找问题，定位更准确

缺点：

- 代码得不到测试，覆盖有盲区
- 如果需求规格说明设计有误，难发现错误
- 测试用例设计难度大

- 存在冗余性

等价类划分法

2.1.1 概述

把所有可能的输入数据划分成互不相交的子集，称为**等价类**。从每个子集中选少量有代表性的数据作为测试用例
意义：测试完备性，等价类互不相交保证测试无冗余性
等价类划分
有效等价类：有效值的集合
无效等价类：无效值的集合，不符合程序要求的数据
等价类划分原则

- 在输入条件规定取值范围情况下，可以确立一个有效等价类和两个无效等价类
- 输入条件规定了“必须如何”的情况下，可以确立一个有效等价类和一个无效等价类
- 在输入条件是一个布尔量的情况下，可确定一个有效等价类和无效等价类
- 输入数据是一组值（n 个），并且每个值分别处理情况下，确立 n 个有效等价类和一个无效等价类
- 规定了输入数据必须遵守规则情况下，可确定一个有效等价类和若干个无效等价类
  设计测试用例
  每个测试用例编号唯一
- 确定测试对象，保证非测试对象的正确性
- 为每一个等价类规定一个唯一编号
- 设计有效等价类的测试用例，尽可能覆盖多的
- 设计无效等价类的测试用例，尽可能覆盖多的

边界值分析法

2.2.1 概述

对软件的输入或者输出边界进行测试的一种方法，作为等价类划分法的一种补充测试，它的所有测试用例都是在**等价类的边界**上执行软件测试工作
边界值是在边界附近寻找值，三个点：上点（边界上的点）、离点（离边界最近的点）、内点（需求范围内的点）

边界值分析

- 标准边界值测试：只考虑有效数据范围内的边界值，对于一个有 n 个变量的程序，保留一个变量，其取值会有最小值、略高于最小值、正常值、略低于最大值、最大值，其余变量为正常值，会产生 4 n+1 个用例
- 健壮边界值测试：会考虑有效和无效数据范围内的边界值，保留一个变量取略低于最小值、最小值、略高于最小值、正常值、略低于最大值、最大值、略高于最大值，其余变量正常，会产生 6 n+1 个测试用例
  原则

- 如果输入规定值的范围，则取刚达到范围边界值和刚刚超越边界的值
- 如果规定了值得个数，则用 6 n+1
- 如果程序规格说明输入或者输出是有序集合，选第一个或者最后一个测试
- 如果程序用了内部结构，应取内部数据结构边界值作为测试
- 分析规格说明，找出其他可能边界

因果图法和决策表法

2.3.1 因果图法概述

利用图解法分析输入的各种组合情况的测试方法，考虑了输入条件的各种组合和互相制约关系，并考虑了输出，例如北京->昌平区，北京就把昌平限制了，因果图法解决多个输入之间的作用关系而产生的测试用例设计方法

1. 恒等：一个输入一个输出
2. 非：用~表示，有一个输入一个输出，输出取反
3. 或：用 V 表示，可以有多个输入，输入有一个为 1 输出则为 1
4. 与：用^表示，可以有多个输入，只有输入全为 1，输出才为 1
5. 异 E：ab 只能有一个为 1，且不能同时为 1
6. 或 I：abc 中至少有一个为 1，且 abc 不能同时为 1
7. 唯一 O：ab 有且仅有一个为 1
8. 要求 R：ab 必须保持一致
   设计步骤

- 确定程序输入（原因）和输出（结果）
- 分析输入输出之间关系，用因果图表示
- 使用符号标记它们之间限制或者约束关系
- 将因果图转为决策表
- 根据决策表设计测试用例

2.3.2 决策表法概述

决策表也称判定表，实质是逻辑表
决策表组成

- 条件桩：列出所有问题的所有条件
- 条件项：条件桩所有取值
- 动作桩：对问题可能采取的动作
- 动作项：在条件项的各组取值情况下应采取的动作
  决策表中每一列就是一条规则，每一列可以设计一个测试用例
  如果对于某两条规则，前两个问题取值一样导致第三个取值对结果无影响，可以合并前两个条件

构件决策表

1. 确定规则个数，n 个条件有 2^n 个规则
2. 列出所有条件桩和动作桩
3. 填入条件项
4. 填入动作项
5. 简化决策表

使用情况

1. 规格说明以决策表形式给出
2. 条件的排列顺序不会影响执行哪些操作
3. 规则的排列顺序不会影响执行哪些操作
4. 某一规则条件满足要执行操作时，不检验别的规则
5. 如果一个规则得到满足要执行多个操作，这些操作执行顺序无关紧要
   缺点时不能表达重复步骤，比如循环结构

正交试验设计法

概述

从大量实验点中挑出适量的、有代表性的点
三个关键因素

1. 指标：判断结果优劣
2. 因子：所有影响指标的条件
3. 因子状态：因子变量的取值

构造步骤

1. 提取因子，构造因子-状态表：比如某软件运行受到操作系统和数据库影响，所以因子有操作系统和数据库

因子	因子状态		
操作系统	Windows	Linux	MacOS
数据库	MySQL	MongoDB	Oracle

2. 加权筛选，简化因子-状态表：根据重要程度进行加权筛选，选出重要因子及状态
3. 构件正交表($L_n(t^c)$)，设计测试用例：L 表示正交表；n 表示正交表的行数，一行表示一个测试用例；c 表示正交实验的因子数目，即正交表的列数；t 称为水平数，表示每个因子能

取得最大值，即因子有多少个状态 $n=(t-1)*c+1$

例如$L_4(2^3)$是较为简单的正交表，它表示该实验有3个因子，每个因子有2个状态，可以做4次实验。如果用0和1表示每个因子的2种状态，则该正交表就是一个4行3列的表。$L_4(2^3)$正交表如下表所示。

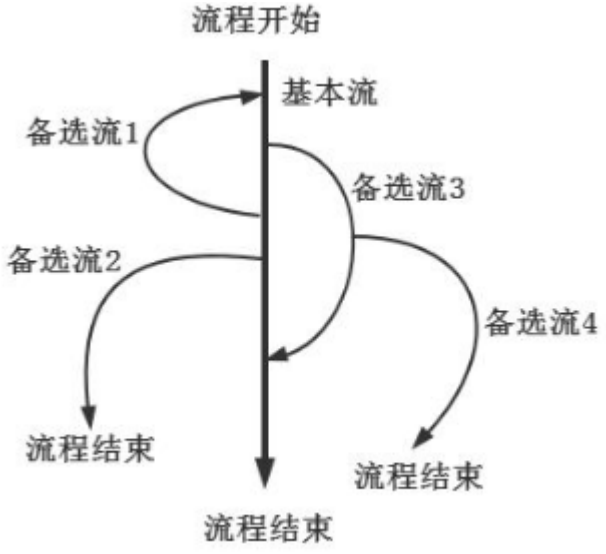
行	列		
	1	2	3
1	1	1	1
2	1	0	0
3	0	1	0
4	0	0	1

场景法

2.5.1 概述

也叫流程图法，指通过模拟用户操作软件的场景对系统进行测试，一般用于测试多个功能之间组合使用情况，以及用于集成测试、系统测试和验收

将用户操作流程分为基本流（有效流）和备选流（无效流、错误流），基本流模拟用户正确操作流程；备选流模拟用户错误操作流程



分析步骤

1. 分析需求规格说明书
2. 根据说明书绘制流程图
3. 根据流程图确定测试场景

4. 根据测试场景设置用例

符号	名称	说明
	椭圆	表示流程的开始或结束
	平行四边形	表示流程的输入或输出
	长方形	表示处理或执行
	菱形	表示对某个条件的判断
	箭头	表示流程进行的方向

流程图常用符号

状态迁移图法

2.6.1 概述

用来描述系统或者对象的状态以及导致系统或者对象状态改变的事件，通过分析被测系统的状态以及状态之间的转换条件来设计测试用例
状态迁移图通过触发的事件来完成各个状态之间的迁移

步骤

1. 绘制状态图：根据需求分析系统中有哪些状态以及迁移关系
2. 列出状态-事件表
3. 绘制状态转换树并推导测试路径：确定一个根节点，延申到所有状态都包含在状态转换树中
4. 设计测试用例：选取达到规定测试覆盖率的路径，每条路径设计一个或者多个测试用例

白盒测试方法

代码检查法

主要检查代码和程序设计的一致性，代码结构的合理性，代码编写的标准性、可读性

代码审查和走查
代码审查：由团队资深开发者对代码进行系统检查，重点关注代码正确性、安全性、性能、可读性，核心目的是发现代码中的缺陷，优化代码质量
代码走查：代码作者主动引导团队成员逐行模拟或逐模块模拟，重点关注代码逻辑流程、设计意图和思路，目的是让团队理解代码的工作原理，确认实现是否符合需求

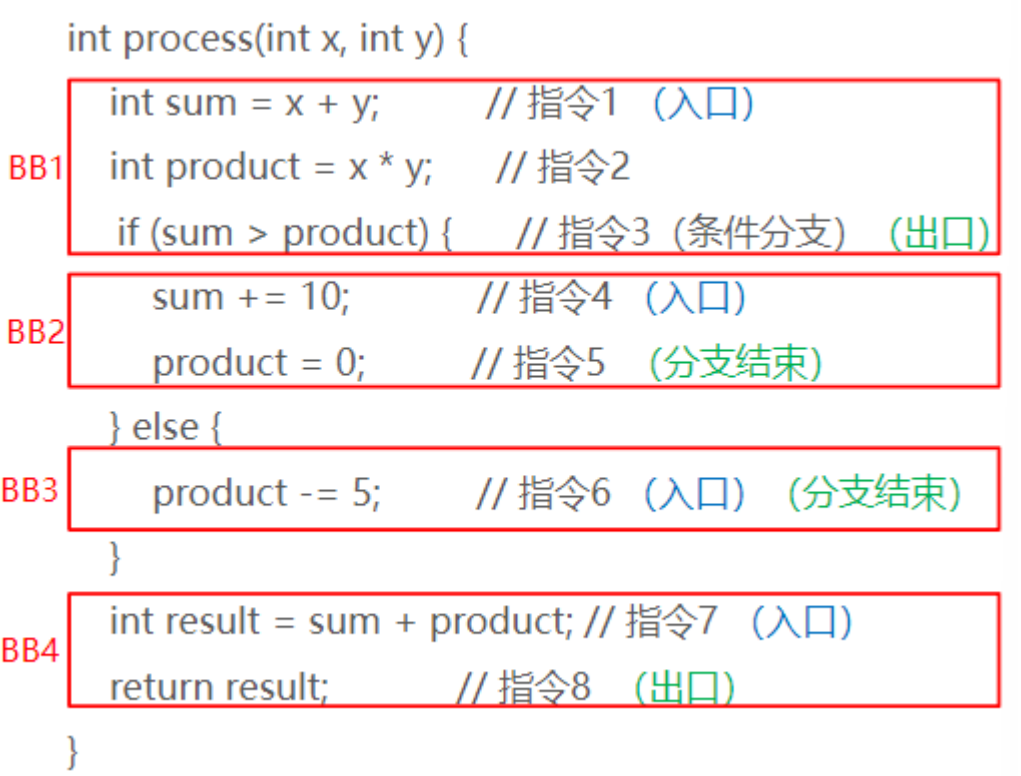
桌面检查
程序员检查自己编写的程序，对源码进行分析、检验、并补充文档

静态结构分析法
以图像的方式表现程序的内部结构，生成函数调用关系图、模块控制流程图、函数内部控制图等各种图表

白盒测试又称透明盒测试、结构测试，需要对程序的内部逻辑进行测试

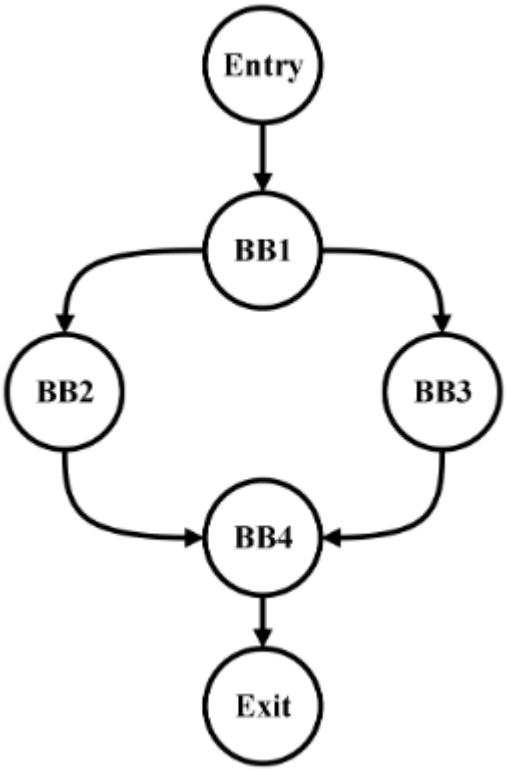
基本块的划分

1. 确定入口点（基本块的起始指令）：程序的第一条指令、被分支指令比如 if-else、跳转指令
2. 确定出口（基本块的结束指令）：条件分支、goto、返回指令、程序的最后一条指令
3. 划分



控制流图

程序执行地图，用有向图表示程序执行，以“基本块”为节点



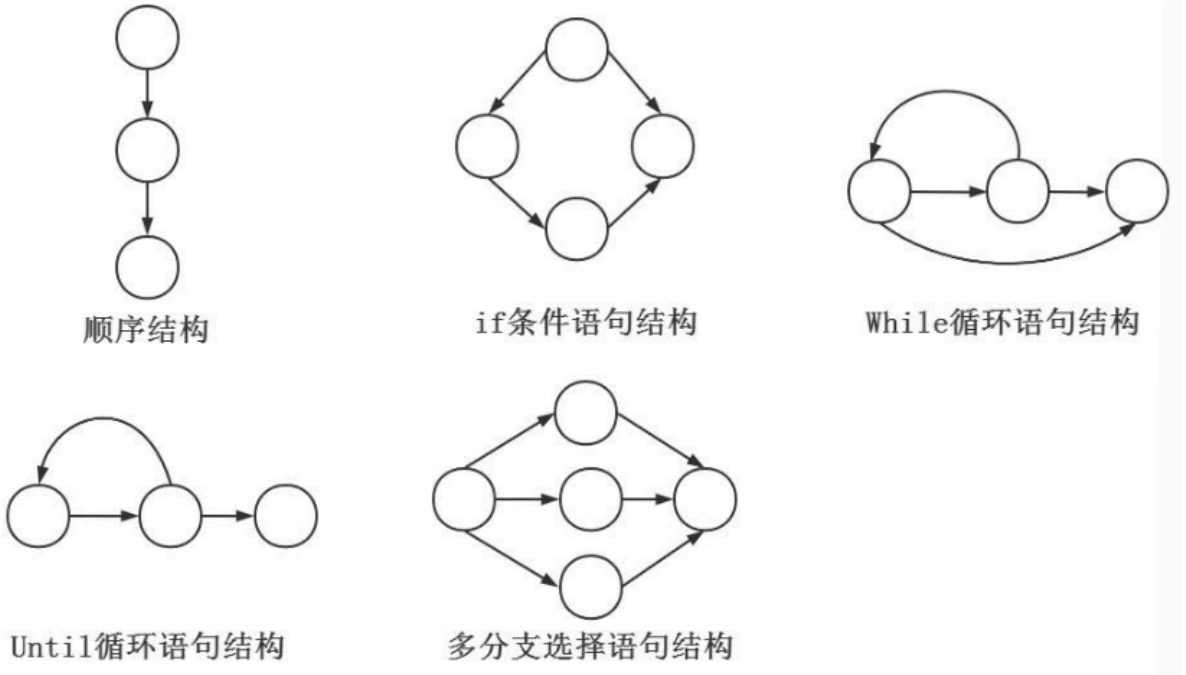
基本路径法

将程序流图转化为程序控制流图，需要**确保**程序中每个可执行语句**至少执行一次**

3.1.1 概述

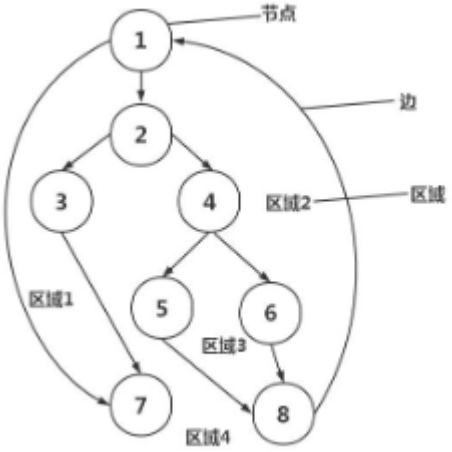
步骤

1. 分析源代码，画出程序的流程图
2. 画出控制流图：用圆圈表示一条或多条无分支的语句（程序块）



3. 计算程序圈复杂度：

1. $V(G)=E-N+2$ 其中 $V(G)$ 表示复杂度，E 表示边数量，N 表示节点数量
2. $V(G)=p+1$ P 表示控制流图中判定的节点数量
3. 程序复杂数等于控制流图中的区域数量



4. 设计测试用例：圈复杂数为 4，于是有 4 条路径（1-7；1-2-3-7；1-2-4-5-8-1-7...）

程序流图转化为控制流图

1. 识别入口
2. 识别出口
3. 划分基本块
4. 为基本块命名
5. 绘制结点与边
6. 添加入口和出口节点

逻辑覆盖法

测试覆盖率：确定测试所执行的覆盖项的百分比，一般测试覆盖率越高，效果越好

1. 基于需求的测试覆盖，生命周期的里程碑处提供覆盖标识（已计划的、已实施的等）
2. 基于代码的测试覆盖，已经执行代码的多少，可以建立在控制流（语句、分支、路径）基础上

3.2.1 语句覆盖

只测试代码中的**执行语句**，不包括注释、头文件等，只能覆盖某一条路径，路径上每个语句至少执行一次

```
test1:x=3 y=-1 z=2
```

但是仅执行一次不能全面覆盖，所以语句覆盖是弱覆盖方法

3.2.2 判定覆盖

又称为分支覆盖，设计足够多的测试用例，保证每个判定条件至少有一次真值，一次假值，但是任然具有单一性

3.2.3 条件覆盖

设计足够多的测试用例，使语句中每个逻辑条件取真值假值至少一次

3.2.4 判定-条件覆盖

使得判定语句中所有条件取值至少出现一次，同时，所有判定语句可能结果也至少出现一次（也存在条件遗漏）

3.2.5 条件组合覆盖

使得判定语句中每个条件所有可能情况至少出现一次

覆盖准则	覆盖目标（核心）	优点	缺点
语句覆盖	所有语句至少执行 1 次	简单易实现	不关心判定 / 条件，漏洞多
判定覆盖	每个判定的 T/F 至少执行 1 次	覆盖判定结果	不关心条件的真假状态
条件覆盖	每个条件的 T/F 至少执行 1 次	覆盖条件状态	可能遗漏判定结果（如本例未覆盖 T）
判定 - 条件覆盖	同时覆盖判定 T/F 和条件 T/F	兼顾判定和条件	未覆盖条件的所有组合
条件组合覆盖	所有条件的真假组合至少执行 1 次	覆盖最全面，漏洞最少	用例数多（2^n），测试成本高

- 覆盖程度从低到高：**语句覆盖 < 判定覆盖 < 判定 - 条件覆盖 < 条件组合覆盖**（条件覆盖与判定覆盖无绝对高低，需看场景）；
- 测试成本从低到高：与覆盖程度正相关（条件组合覆盖成本最高）；
- 实际测试中需平衡 “覆盖程度” 和 “成本”，并非越全面越好（如简单逻辑用判定 - 条件覆盖即可，复杂逻辑需考虑条件组合覆盖）。

程序插桩法

3.3.1 目标代码插桩

向目标代码中插入测试代码，获取程序的运行信息，对内存监控、指令跟踪、错误检测等，这个方法不用重新编译程序

原理

在原有逻辑不变的情况下，向目标代码中（二进制代码）插入特定的测试代码，获取运行时信息

1. 插桩位置确定：对目标代码进行**静态分析**，解析函数调用等，确定哪些位置能获得有用的信息
2. 桩代码插入：二进制形式的目标代码，需要专门的二进制编辑工具或者插桩工具，高级语言可以在编译阶段介入
3. 桩代码执行与信息获取

目标代码插桩的两种方式

1. 静态插桩：未运行之前对代码插桩，适用于实现完整系统或仿真，更容易系统和全面的规划

2. 动态插桩：运行时利用插桩工具，在不中断程序正常运行前提下, 插入代码

目标代码插桩的执行模式

1. 即时模式：先存副本，测试时进修改部分目标代码
2. 解释模式：插入测试代码作为目标代码指令的解释语言, 目标执行一条指令，就找一条相应替代指令
3. 探测模式：新指令覆盖旧指令

插桩工具

1. Pin
2. DynamoRIO

源代码插桩

对高级语言的源代码插桩，获取程序运行信息，辅助程序调试

插桩过程

1. 源文件
2. 预处理
3. 语法分析：构建语法树，明确逻辑结构（函数、分支、循环）
4. 插入探针代码
5. 编译器

测试运行过程

1. 收集测试信息
2. 信息分析
3. 测试覆盖分析

源代码插桩有效提高代码测试覆盖，但是会出现代码膨胀、执行效率低

对比维度	源代码插桩	目标代码插桩
操作对象	高级语言源代码（如 C/C++、Java、Python 代码）	二进制目标代码（可执行文件、库文件等）
是否需要源码	必须需要，无源码则无法操作	无需源码，直接处理二进制文件
编程语言依赖	强依赖：需适配不同语言的语法（如 Java 用 AspectJ，C 用宏定义）	弱依赖：与源码语言无关，仅依赖操作系统 / 指令集（如 x86、ARM）
插桩时机	程序编译前（源码阶段）	程序编译后（二进制阶段），可分静态（运行前）、动态（运行中）
编译依赖	插桩后需重新编译、链接生成新程序	无需重新编译，直接修改二进制或运行时插入
适用场景	开发阶段调试、单元测试、源码级性能分析	闭源软件测试、运行时监控、漏洞检测、跨语言程序分析
对程序影响	可能增加源码体积，编译后才能验证插桩效果	不改变原始源码，动态插桩可实时启停，影响较小

黑白盒测试对比

对比维度	黑盒测试（Black-Box Testing）	白盒测试（White-Box Testing）
测试核心依据	仅依赖软件的需求规格说明书 / 用户手册，不关注内部逻辑	依赖软件的源代码、数据结构、算法、控制流程等内部信息
测试视角	站在用户角度，验证软件 “输入→输出” 是否符合预期	站在开发 / 测试角度，验证代码逻辑执行是否正确
常用测试方法	等价类划分法、边界值分析法、因果图法、错误推测法	逻辑覆盖法（语句 / 判定 / 条件覆盖等）、基本路径测试法
人员能力要求	无需懂编程，需熟悉需求和用户场景	需具备编程能力，能读懂代码、分析逻辑结构
可发现的问题	功能缺失、输入输出异常、UI 交互问题、兼容性问题	代码逻辑错误（如死循环）、变量未初始化、内存泄漏、安全漏洞
适用测试阶段	集成测试、系统测试、验收测试（后期阶段）	单元测试、代码审查（前期阶段）
优点	1. 不依赖源码，适合闭源软件； 2. 贴近用户实际使用场景； 3. 测试用例设计简单	1. 能深入定位代码级问题； 2. 测试覆盖更全面（如分支、路径）； 3. 提前发现底层风险
缺点	1. 无法检测内部逻辑错误； 2. 需求不完整时测试不全面； 3. 可能遗漏复杂逻辑组合问题	1. 依赖源码，闭源软件无法使用； 2. 测试成本高（需懂代码）； 3. 代码变更后测试用例需频繁更新

接口测试

接口测试简介

模拟客户端向服务器发送请求

接口文档也叫 API 文档，由开发人员编写，描述接口信息，通常，接口测试用例包括用例 ID、用例名称、接口名称、前置条件、请求地址（URL）、请求方法、请求头、请求参数、预期结果、实际结果

HTTP

统一资源定位符（URL）格式 `protocol://hostname[:port]/path? query#fragment`
Protocol: 传输协议
Hostname: 域名、主机名
Port: 端口号（可省略）
Path: 资源路径
Query: 查询参数，参数名和参数值之间用=隔开，参数之间用&隔开
Fragment: 锚点，定位页面具体位置

HTTP 请求

请求行：请求方法（GET、POST、PUT、DELETE）、请求地址、协议（HTTP）
请求头：Host、UserAgent、ContentType、Authorization（身份验证）
请求体：text/html text/plain application/JSON image/jpeg 等

HTTP 响应

状态行：HTTP 响应的协议版本、状态码和状态码描述（1 xx 指示信息、2 xx 成功、3 xx 请求重定向、4 xx 客户端错误、5 xx 服务器错误）

响应头：Server（服务器所使用软件信息）、ContentType、Connection（客户端和服务器连接类型）、contentLength、contentLanguage
响应体：实际类容

Postman 入门

发送请求

Params：参数，可以设置请求地址参数
Authorization：授权
Headers：请求头
Body：请求体
Script：预请求脚本
Tests：表示测试（编写测试代码，比如断言代码、关联代码等）
Settings：设置

Postman 断言

断言是一种逻辑判断式，目的是验证软件开发的预期结果于实际结果是否一致
适用 js 编写，常见的有**响应状态码断言、包含制定字符串断言、JSON 数据断言**
响应状态码断言：检查 API 请求的响应状态码是否为 200，如果是则测试通过
包含指定字符串：检查 API 返回响应体中是否包含某个字符串
JSON 数据断言：

Postman 关联

接口测试中，关联是指两个或者两个以上的接口互相存在依赖关系，比如一个接口的请求参数是另一个接口的响应结果，实现接口关联的方法是**设置环境变量或全局变量**，可以设置多组环境变量和一组全局变量，且环境变量优先级高于全局变量

1. 获取第一个接口的响应结果
2. 保存到环境变量
3. 第二个接口的请求地址中引用环境变量

性能测试

性能测试是度量软件质量的重要方式，从软件的响应速度、稳定性、兼容性、可移植性等方面检测

性能测试概述

通过工具模拟正常、峰值、异常负载条件来对系统进行各项指标测试，能检验软件是否达到了用户期望性能需求，同时也能发现系统中可能存在的性能瓶颈和缺陷，从而优化系统性能

1. 基准测试：单用户测试（狭义），是一种测量和评估软件性能指标的测试（广义）
2. 负载测试：逐步增加系统负载（增加模拟用户数量），确定在满足性能指标情况下的最大负载

3. 压力测试：强度测试，让系统超负荷运行，使得系统某些资源达到饱和或接近系统崩溃边缘，确定系统能承受最大压力
4. 并发测试：模拟用户并发访问同一个应用等是否存在死锁或者响应慢等情况
5. 配置测试：调整软硬件环境，测试环境对系统性能影响，找到系统各项资源最优分配
6. 稳定性测试：在强负载情况下持续运行一段时间，比如 7* 24 ,可以检测系统是否存在内存泄露等
7. 容量测试：最大用户数、最大存储
   指标
   响应时间（请求做出响应的时间）、吞吐量（单位时间完成的工作量，请求数/s、访问人数/天）、并发用户数、QPS（每秒查询数）、TPS（每秒事务数）、点击率（每秒用户向服务器提交的 HTTP 请求）、错误率、资源利用率

JMeter 测试

添加线程组

1. SetUp 线程组：执行测试前初始化操作
2. TearDown 线程组：执行测试结束之后回收工作
3. 开放模型线程组：模拟不同场景下用户动态变化的访问情况
4. 线程组：一个线程组可以表示一个虚拟用户组

JMeter 核心组件

取样器：模拟用户操作
监听器：监听测试结果，将结果以表格或者图像形式展现，只能监听同级或者下级元件数据（最后查看结果树和聚合报告）

断言

断言持续时间：主要用于断言请求的响应时间是否满足要求

前置处理器

用于在请求发送前对请求进行一些特殊化处理，比如参数化、加密请求和替换请求字段等

后置处理器

用于对响应数据进行关联处理

1. 正则表达式提取器：用来检索、替换符合规定的字符串

元字符	含义
()	封装待返回的字符串
.	匹配除换行符以外的任意字符
+	匹配前面的字符串一次或多次
?	匹配前面的字符串0次、1次，在找到第一个匹配项后停止
*	匹配前面出现的字符0次或多次
^	匹配字符串的开始位置
$	匹配字符串的结束位置
|	模式选择符，从中任选一个匹配

```
左边界(正则表达式)右边界
<title>(.*)</title>
```

2. XPath
3. Json 提取器

逻辑控制器

1. 控制测试计划节点发送请求的逻辑顺序控制器，比如 if、循环
2. 用来对测试计划中的脚本分组，运行时控制，比如事务控制器、吞吐量控制器等

定时器

用于为请求设置等待时间，使得请求暂停一段时间再发送

1. 同步定时器：可以阻塞线程，使这些线程在同一时间发送请求
2. 常数吞吐量定时器：设置 QPS 限制，按照指定吞吐量发送请求，用于稳定性测试和混压测试
3. 固定定时器

APP 测试

App 特性

1. 设备多样性
2. 网络多样性（3 g、4 g、5 g）
3. 平台多样性（操作系统不同）

App 与 Pc 端软件测试方面区别

1. 页面布局不同
2. 使用场合不同（地点不固定，网络不稳定）
3. 输入方式不同

4. 操作方式不同

测试环节
接受测试版本->App 版本测试->UI 测试->功能测试->专项测试->正式环境测试->上线准备

测试要点

UI 测试

界面是否美观、文字表达是否简洁准确、界面是否美观、操作是否简便
三要点

1. 导航测试（风格、布局、准确与否）
2. 图形测试（质量显示、字体风格、搭配）
3. 内容测试（文字表达、是否简洁、长度限制）

功能测试

- 注册
- 登录
- 运行
- 切换
  - 后台切换
  - 删除进程
  - 锁屏
- 推送
- 更新

专项测试

- 安装测试
- 卸载测试
- 升级测试
- 交互性测试
- 弱网测试
- 耗电测试

性能指标

- 边界测试
- 压力测试
- 耗能测试
- 响应能力测试

兼容性测试

1. 系统兼容测试
2. 屏幕分辨率兼容测试
3. 屏幕尺寸兼容测试
4. 网络兼容测试
5. 品牌兼容性测试

Appium 基本应用

元素定位

1. 通过 resource-id 定位（该属性唯一）
2. 通过 class 定位，通常由多个，需要验证元素 class 是否唯一
3. 通过 content-desc 定位
4. 通过 xpath 定位：选取 XML 中的节点或节点集

手势操作

1. 轻敲
2. 长按操作
3. 滑动操作
4. 拖曳操作
5. 复杂路径手势