

软件编程

1 良好的编程实践

软件编程是一个复杂而迭代的过程，它不仅仅是编写代码，还应该包括代码审查、单元测试、代码优化、集成调试等一系列工作。

软件编码是一个复杂而迭代的过程，包括程序设计 (program design) 和程序实现 (program implementation)。

软件编码要求：

- 正确的理解用户需求和软件设计思想
- 正确的根据设计模型进行程序设计
- 正确而高效率的编写和测试源代码

软件编码是设计的继续，会影响软件质量和可维护性。

软件编码规范是与特定语言相关的描写如何编写代码的规则集合。

良好的编程实践：

1. 不要编写需要外部文档支持的代码，这样的代码是脆弱的，要确保你的代码本身读起来就很清晰。
编写自文档化的代码。
2. 学会只编写够用的注释，过犹不及，重视质量而不是数量。应该把时间花在编写不需要大量注释支持的代码上，即让代码自文档化。
3. 函数编写的第一条规则是短小，第二条规则是更短小。函数应该做一件事，做好这件事，并且只做这件事。
4. 浮点运算——准确度、精度

代码分析工具：

- (Python) Pylint
- (Java) Checkstyle
- (Java) FindBugs
- (Java) PMD
- (Java) Jtest
- (C) Lint
- (Java Script) JSHint
- (CSS) CSSLint
- (HTML) HTMLHint

2 代码审查

代码审查 (Code Review) 是一种用来确认方案设计和代码实现的质量保证机制，它通过阅读代码来检查源代码与编码规范的符合性以及代码的质量。

代码审查的作用：

- 检查设计的合理性
- 互为 Backup
- 分享知识、设计、技术
- 增加代码可读性
- 处理代码中的“地雷区”

缺陷检查表：见 ppt。

3 代码重构

重构 (Refactoring) 是对软件内部结构的一种调整，其目的是在不改变软件功能和外部行为的前提下，提高其可理解性、可扩展性和可重用性。

什么时候不适合重构：

- 代码太混乱，设计完全错误
与其重构不如重新开始。
- 明天是 Deadline
永远不要做 Last-Minute-Change；应推迟重构但不可忽略，即使进入 Production 的代码都正确地运行。
- 重构的工作量显著地影响估算
一个任务的估算时间是 3 天，如果为了重构，就需要更多的时间。
推迟重构但不忽略，可以把重构作为一个新任务，或者安排在重构的迭代周期中完成。

对比——重构与添加新功能：重构不再添加功能，只管改进程序结构；添加新功能时不应该修改既有代码，只管添加新功能；重构与添加新功能可交替进行。

代码的坏味道：

- 重复的代码
症状：两个代码段几乎相同/拥有几乎相同的作用
措施：
 - 抽取方法
 - 抽取类
 - 替换算法
- 过长的函数
症状：一个方法超过 N (如 10) 行代码（长度是一个警告信号，并不表示一定有问题）

措施：

- 抽取方法
- 每当需要用注释说明点什么时，就把需要说明的东西写进一个独立方法中
- 以其用途（而非实现方法）对抽取的方法命名

说明：对于现代开发工具，方法调用增多不会影响性能，但长方法难以理解。

- 发散式变化

症状：某个类因为不同的原因在不同的方向上发生变化

措施：

- 如果类既要找到对象，又要对其做某些处理，则让调用者查找对象，并将该对象传入，或者让类返回调用者所用的值
- 采用抽取类的方法，为不同的决策抽取不同的类
- 如果多个类共享相同类型的决策，则可以合并这些新类，至少这些类可以构成一层

说明：发散式变化指的是“一个类受多个外界变化的影响”，其基本思想是把相对不变的和相对变化相隔离，即封装变化。

- 霰弹式修改

症状：发生一次改变时，需要修改多个类的多个地方

措施：

- 找出一个应对这些修改负责的类，这可能是一个现有的类，也可能需要通过抽取类来创建一个新类
- 使用移动字段和移动方法等措施将功能置于所选的类中，如果未选中类足够简单，则可以使用内联类将该类除去

说明：霰弹式修改指的是“一种变化引发多个类的修改”，其基本思想是将变化率和变化内容相似的状态和行为放在同一个类中，即集中变化。

- 数据泥团

症状：

- 同样的两至三项数据频繁地一起出现在类和参数表中
- 代码声明了某些字段，并声明了处理这些字段的方法，然后又声明了更多的字段和更多的方法，如此继续
- 各组字段名以类似的子串开头或结束

措施：

- 如果项是类中的字段，则使用抽取类将其取至一个新类中
- 如果值共同出现在方法的签名中，则使用引入参数对象的重构方法以抽取新对象
- 查看这些项的使用：通常可以利用移动方法等重构技术，从而将这些使用移至新的对象中

- 纯稚的数据类

症状：类仅由字段构成，或者只有简单的赋值方法和取值方法构成。

措施：

- 采用封装字段阻止对字段直接访问(仅允许通过赋值方法和取值方法)
- 对可能的方法尽量采用移除设置方法进行重构
- 采用封装集合方法去除对所有集合类型字段的直接访问

- 查看对象的各个客户，如果客户试图对数据做同样的工作，则对客户采用抽取方法，然后将方法移到该类中
- 在完成上述工作后，可能发现类中存在多处相似的方法，使用相应的重构技术，以协调签名并消除重复
- 对字段的大多数访问都不再需要，因为所移动的方法涵盖了其实际应用，因此可以使用隐藏方法来消除对赋值方法和取值方法的访问

软件测试

1 软件测试概述

1.1 软件缺陷术语

错误 (Error)： 在软件生存期内的不希望或不可接受的人为错误，其结果是导致 软件缺陷的产生。

缺陷 (Defect)： 软件缺陷是存在于软件产品之中的那些不希望或不可接受的偏差，其结果是软件运行于某一特定条件时出现故障。

故障 (Fault)： 软件运行过程中出现的一种不希望或不可接受的内部状态，若无 适当措施(容错)加以及时处理，便产生软件失效。

失效 (Failure)： 软件运行时产生的一种不希望或不可接受的外部行为结果。

1.2 软件测试概念

测试是使用人工和自动手段来运行或检测某个系统的过程，其目的在于检验系统是否满足规定的需求或弄清预期结果与实际结果之间的差别。

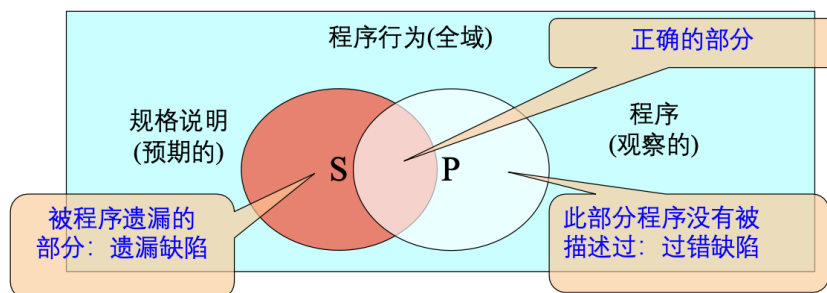
软件测试的目的：

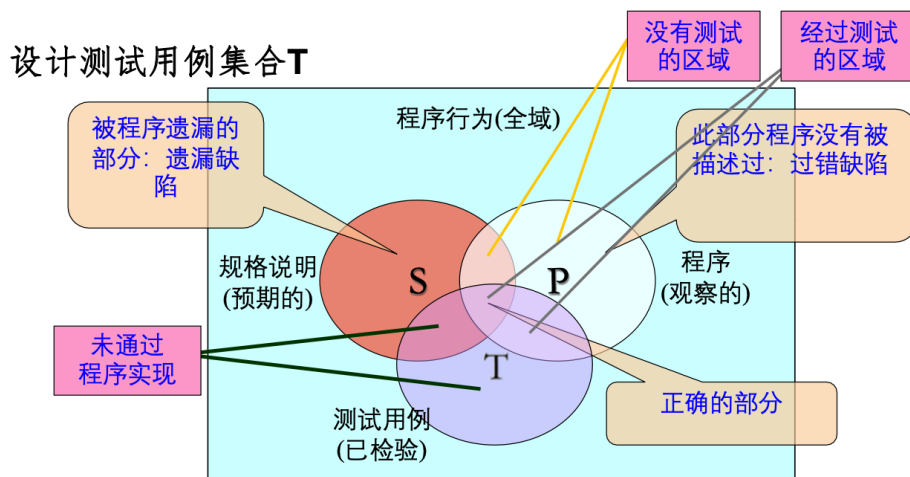
- 直接目标：发现软件错误
- 期望目标：检查系统是否满足需求
- 附带目标：改进软件过程

测试的局限性：

- 测试的不彻底性
- 测试的不完备性
- 测试作用的间接性

Venn 图：





1.3 软件测试基本原则

1. 测试用例中一个必需部分是对预期输出或结果的定义。
2. 编程人员应当避免测试自己编写的程序。
3. 编写软件组织不应测试自己编写的软件。
4. 应当彻底检查每个测试的执行结果。
5. 测试用例不仅根据有效的和预期的输入情况，也应根据无效的和未预料到的输入情况。
6. 检查程序是否“未做应该做的”仅是测试的一半，另一半是检查是否“做了不应该做的”。
7. 应避免测试用例用后即弃，除非是一次性软件。
8. 计划测试工作时不应默许假定不会发现错误。
9. 程序某部分存在更多错误的可能性，与该部分已发现错误的数量成正比。
10. 软件测试是一项极富创造性、极具智力挑战性的工作。

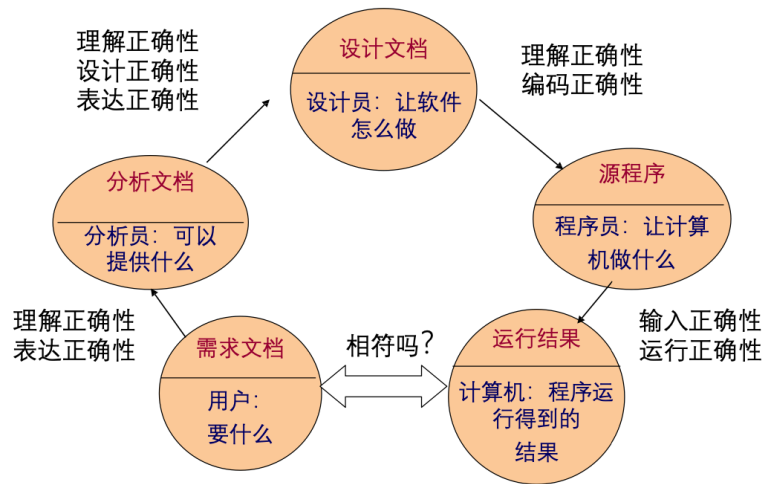
1.4 软件测试团队

角色：

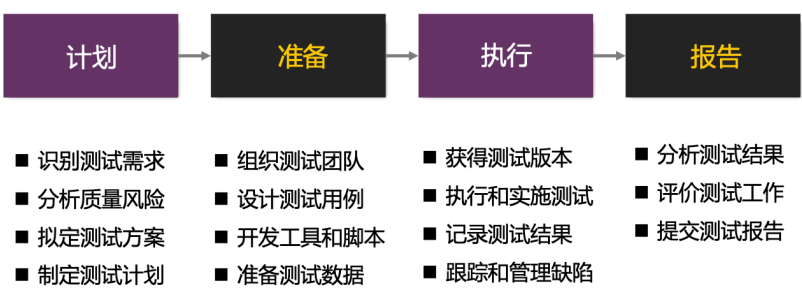
- 测试经理：建立和完善测试流程以及部门管理体制，审核测试项目并分配资源，监控和协调各项目的测试工作，负责与其他部门的协调和沟通工作。
- 测试组长：制定测试项目计划（包括人员、进度、软硬件环境和流程等），实施软件测试，跟踪和报告计划执行情况，负责测试用例质量，管理测试小组并提供技术指导。
- 测试工程师：理解软件产品的要求，对其进行测试以便发现软件中的错误，验证软件是否满足规格说明所描述的需求，编写相应的测试方案和测试用例。
- 测试工具工程师：编写软件测试工具，并利用所编写的测试工具对软件进行测试，或者为测试工程师开发测试工具。

2 软件测试策略

2.1 软件测试对象



2.2 软件测试过程



2.3 软件测试类型

测试对象角度:

- 单元测试: 软件基本组成单元进行的测试, 有时也称“组件测试”。

单元测试环境:

- 驱动模块: 模拟被测模块的上一级模块, 接收测试数据, 把测试数据传送给所测模块, 最后再输出实际测试结果
- 桩模块: 模拟被测单元需调用的其他函数接口, 模拟实现子函数的某些功能

- 集成测试: 在单元测试的基础上, 将所有模块按照总体设计的要求组装成子系统或系统进行的测试。

集成测试的对象是模块间的接口, 其目的是找出在模块接口上, 包括系统体系结构上的问题

方式:

- 整体集成方式 (非增量式集成)

优点: 效率高, 所需人力资源少; 测试用例数目少, 工作量低; 简单, 易行

缺点：可能发现大量错误，难以进行错误定位和修改；即使测试通过，也会遗漏很多错误；测试和修改过程中，新旧错误混杂，带来调试困难

— 自顶向下增量式集成

优点：能尽早地对程序的主要控制和决策机制进行检验，因此较早地发现错误；较少需要驱动模块

缺点：所需的桩模块数量巨大；在测试较高层模块时，低层处理采用桩模块替代，不能反映真实情况，重要数据不能及时回送到上层模块，因此测试并不充分

— 自底向上增量式集成

优点：不用桩模块，测试用例的设计亦相对简单

缺点：程序最后一个模块加入时才具有整体形象，难以尽早建立信心

- 系统测试：在实际运行环境或模拟实际运行环境下，针对系统的非功能特性所进行的测试，包括负载测试、性能测试、压力测试、恢复测试、安全测试和可靠性测试等。
- 验收测试：在软件产品完成了功能测试和系统测试之后、产品发布之前所进行的软件测试活动，目的是验证软件的功能和性能是否能够满足用户所期望的要求。
- 回归测试：在软件生命周期的任何一个阶段，只要软件发生了改变，就可能带来问题，为了验证修改的正确性及其影响就需要进行回归测试。

测试技术角度：

- 黑盒测试（功能测试）：将测试对象看作黑盒子，完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它的功能说明。
- 白盒测试（结构测试）：把测试对象看做一个透明的盒子，允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。

程序执行角度：

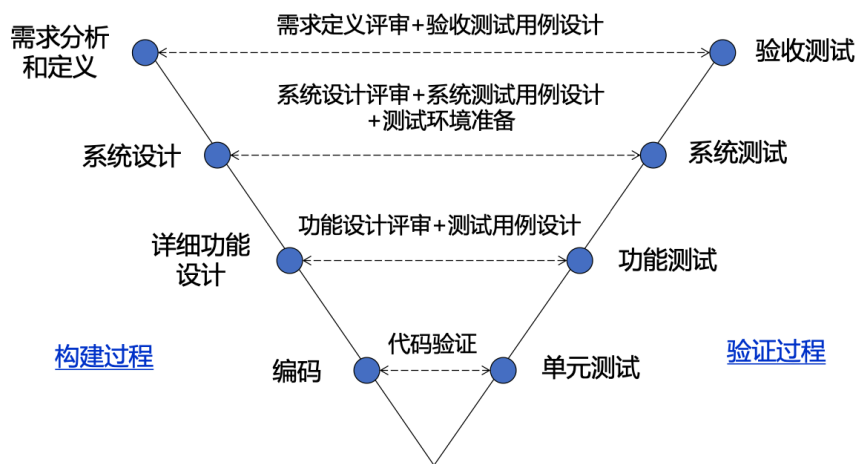
- 静态测试：通过人工分析或程序正确性证明的方式来确认程序正确性。
- 动态测试：通过动态分析和程序测试等方法来检查程序执行状态，以确认程序是否有问题。

人工干预角度：

- 手工测试：测试人员根据测试大纲中所描述的测试步骤和方法，手工地输入测试数据并记录测试结果。
- 自动化测试：相对于手工测试而言，主要是通过所开发的软件测试工具、脚本等手段，按照测试工程师的预定计划对软件产品进行的自动测试。

自动化测试只是对手工测试的一种补充，但绝不能代替手工测试，二者有各自的特点。

测试的 V 模型：



2.4 测试用例

测试用例是为特定的目的而设计的一组测试输入、执行条件和预期的结果。

测试用例是执行的最小测试实体。

测试用例就是设计一个场景，使软件程序在这种场景下，必须能够正常运行并且达到程序所设计的执行结果。

重要性：

- 指导人们系统地进行测试
- 有效发现缺陷，提高测试效率
- 作为评估和检验的度量标准
- 积累和传递测试的经验与知识

要求：

- 具有代表性和典型性
- 寻求系统设计和功能设计的弱点
- 既有正确输入也有错误或异常输入
- 考虑用户实际的诸多使用场景

白盒测试

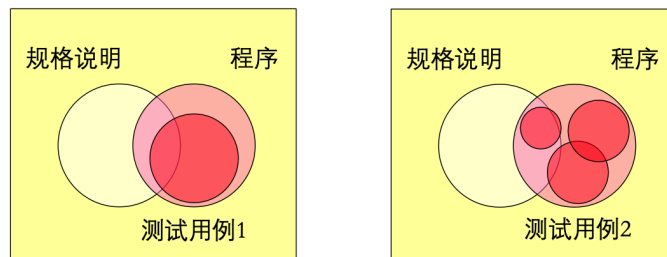
1 白盒测试概述

白盒测试（又称为“结构测试”或“逻辑驱动测试”）：把测试对象看做一个透明的盒子，它允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。

白盒测试主要对程序模块进行如下的检查：

- 对模块的每一个独立的执行路径至少测试一次；
- 对所有的逻辑判定的每一个分支（真与假）都至少测试一次；
- 在循环的边界和运行界限内执行循环体；
- 测试内部数据结构的有效性。

白盒测试的 Venn 图：



白盒测试的特点：

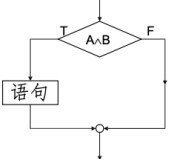
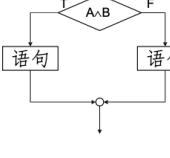
- 以程序的内部逻辑为基础设计测试用例，又称逻辑覆盖法。
- 应用白盒法时，手头必须有程序的规格说明以及程序清单。

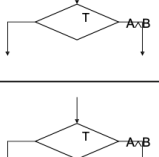
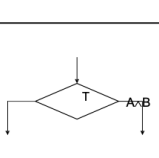

白盒测试考虑测试用例对程序内部逻辑的覆盖程度：最彻底的白盒法是覆盖程序中的每一条路径，但是由于程序中一般含有循环，所以路径的数目极大，要执行每一条路径是不可能的，只能希望覆盖的程度尽可能高些。

2 白盒测试方案

2.1 逻辑覆盖

- 语句覆盖：使得被测试程序中的每条可执行语句至少被执行一次。
- 判定覆盖（分支覆盖）：不仅每个语句必须至少执行一次，而且每个判定的每个分支都至少执行一次。
- 条件覆盖：不仅每个语句至少执行一次，而且使判定表达式中每个条件的可能取值至少各**执行**一次。
- 判定/条件覆盖：同时满足判定覆盖和条件覆盖的要求。
- 条件组合覆盖：每个判定表达式中条件的各种可能组合都至少**出现**一次。

覆盖标准	程序结构举例	测试用例应满足的条件
语句覆盖		$A \wedge B = T$ 使得被测试程序中的每条可执行语句至少被执行一次。
判定覆盖		$A \wedge B = T$ $A \wedge B = F$ 每一判定的每个分支至少被执行一次。

覆盖标准	程序结构举例	测试用例应满足的条件
条件覆盖		$A=T, A=F$ $B=T, B=F$ 每一判定中的每个条件，分别按“真”、“假”至少各执行一次。
判定/条件覆盖		$A \wedge B = T, A \wedge B = F$ $A=T, A=F$ $B=T, B=F$ 同时满足判定覆盖和条件覆盖的要求。
条件组合覆盖		$A=T \wedge B=T$ $A=T \wedge B=F$ $A=F \wedge B=T$ $A=F \wedge B=F$ 求出判定中所有条件的各种可能组合值，每一可能的条件组合至少执行一次。

2.2 控制结构测试

2.2.1 基本路径测试

基本路径测试是一种白盒测试技术。使用这种技术设计测试用例时，首先计算过程设计结果的逻辑复杂度，并以该复杂度为指南定义执行路径的基本集合。

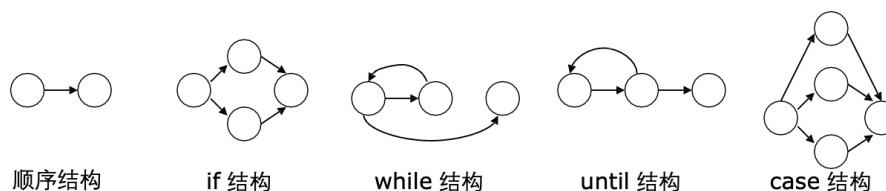
- 在程序控制图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例。
- 设计出的测试用例要保证在测试中程序的每条可执行语句至少执行一次。
- 程序中的每个条件至少被测试一次（True 和 False 都要测试到）。

前提条件：测试进入的前提条件是在测试人员已经对被测试对象有了一定的了解，基本上明确了被测试软件的逻辑结构。

测试过程：过程是通过针对程序逻辑结构设计和加载测试用例，驱动程序执行，以对程序路径进行测试。测试结果是分析实际的测试结果与预期的结果是否一致。

步骤：

1. 根据设计或源代码画出相应流程图



注意：如果判断中的条件表达式是由一个或多个逻辑运算符连接的复合条件表达式，则需要改成一系列只有单一条件的嵌套的判断。

2. 确定环复杂度

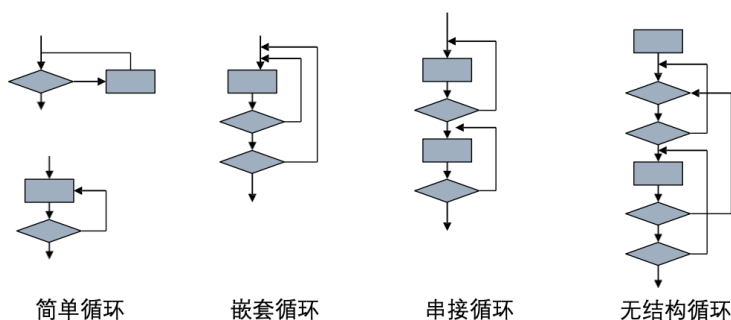
三种等价的计算方法：流图中区域的数量； $E - N + 2$ ； $P + 1$ （ P 是判定节点数量）

3. 确定独立路径的基本集合

独立路径：一条路径，至少包含一条在定义该路径之前不曾用过的边（至少引入程序的一个新处理语句集合或一个新条件）

4. 准备测试用例，执行基本集合中每条路径

2.2.2 循环测试



● 简单循环：测试应该包括以下几种：

- 零次循环：从循环入口直接跳到循环出口。
- 一次循环：查找循环初始值方面的错误。
- 二次循环：检查在多次循环时才能暴露的错误。
- m 次循环：此时的 $m < n$ ，也是检查在多次循环时才能暴露的错误。
- n （最大）次数循环、 $n + 1$ （比最大次数多一）次的循环、 $n - 1$ （比最大次数少一）次的循环。

● 嵌套循环：

- 从最内层循环开始，设置所有其他层的循环为最小值；
- 对最内层循环做简单循环的全部测试。测试时保持所有外层循环的循环变量为最小值。另外，对越界值和非法值做类似的测试。
- 逐步外推，对其外面一层循环进行测试。测试时保持所有外层循环的循环变量取最小值，所有其它嵌套内层循环的循环变量取“典型”值。
- 反复进行，直到所有各层循环测试完毕。
- 对全部各层循环同时取最小循环次数，或者同时取最大循环次数。对于后一种测试，由于测试量太大，需人为指定最大循环次数。

● 串接循环：区别两种情况

- 如果各个循环互相独立，则用与简单循环相同的方式进行测试。
- 如果两个循环处于串接状态且前一个循环的循环变量值是后一个循环的初值，则这几个循环不是互相独立的，需要使用测试嵌套循环的办法来处理。
- 非结构循环：不能测试，应该重新设计循环结构，使之成为其他循环方式。

黑盒测试

1 黑盒测试概述

黑盒测试又称“功能测试”、“数据驱动测试”或“基于规格说明书的测试”，是一种从用户观点出发的测试。

将测试对象看做一个黑盒子，测试人员完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它的功能说明。

通常在软件接口处进行。

黑盒测试能发现的错误：

- 是否有不正确或遗漏的功能
- 接口错误
- 数据结构错误或外部信息访问
- 行为或性能错误
- 初始化或终止错误

因为穷举测试数量太大，无法完成，所以测试人员只能在大量可能的数据中，选取其中一部分作为测试用例。

测试方法的评价标准：在最短时间内，以最少的人力，发现最多的，以及最严重的缺陷。

黑盒测试并不是白盒测试的替代品，而是用于辅助白盒测试发现其他类型错误。通常由独立测试人员根据用户需求文档来进行，但不一定要求用户参与。

2 黑盒测试方法

2.1 等价类测试

将程序的输入划分为若干个数据类，从中生成测试用例。并合理地假定“测试某等价类的代表值就等于对这一类其它值的测试”。

等价类的划分要求：分而不交、合而不变、类内等价。

等价类划分的基本原则：

- 每个可能的输入属于某一个等价类
- 任何输入都不会属于多个等价类
- 用等价类的某个成员作为输入时，如果证明执行存在误差，那么用该类的任何其他成员作为输入，也能检查到同样的误差
- 同一输入域的等价类划分可能不唯一
- 对于相同的等价类划分，不同测试人员选取的测试用例集可能是不同的，测试用例集的故障检测效率取决于人员经验

等价类类型：

- 有效等价类

输入域中一组有意义的数据的集合

有效等价类被用于检验系统指定功能和性能是否正确实现

- 无效等价类

输入域中一组无意义的数据的集合

无效等价类被用于检验系统的容错性

无效等价类至少应有一个，也可能有多个

确定等价类的一些建议：

1. 在输入条件规定了取值范围的情况下，可以确定一个有效等价类和两个无效等价类
2. 在规定了输入数据必须遵守的规则情况下，可确定一个有效等价类（符合规则）和若干个无效等价类（从不同角度违反规则）
3. 若规定输入数据是一组值（假定 N 个），并且程序要对每一个输入值分别处理，可确定 N 个有效等价类和一个无效等价类
4. 如果某个输入条件指定了一组特定取值，则可以定义一个有效等价类和一个无效等价类

等价类组合：

- 所有有效等价类的代表值都集成到测试用例中，即覆盖有效等价类的所有组合。任何一个组合都将设计成一个有效的测试用例，也称正面测试用例。
- 无效等价类的代表值只能和其他有效等价类的代表值(随意)进行组合。因此，每个无效等价类将产生一个额外的无效测试用例，也称负面测试用例。

设计测试用例：

测试用例 = {测试数据+期望结果}

测试结果 = {测试数据+期望结果+实际结果}

1. 为每一个等价类规定一个唯一的编号
2. 设计一个新的测试用例，使其**尽可能多的覆盖**尚未被覆盖的有效等价类；重复这一步，直到所有的有效等价类都被覆盖为止
3. 设计一个新的测试用例，使其**仅覆盖一个**尚未被覆盖的无效等价类；重复这一步，直到所有的无效等价类都被覆盖为止

注意：对无效等价类设计测试用例时采用**单缺陷原则**。

例子：见 ppt。

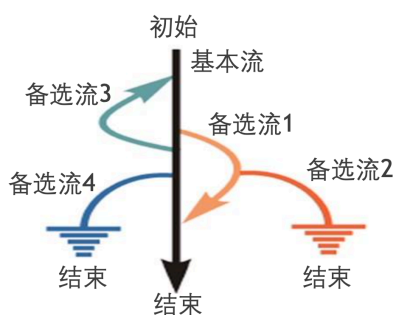
2.2 边界值测试

边界值分析是对输入或输出的边界值进行测试的一种方法，它通常作为等价类划分法的补充，这种情况下的测试用例来自等价类的边界。

- 首先确定边界情况。通常输入或输出等价类的边界就是应该着重测试的边界情况。
- 选取正好等于、刚刚大于或刚刚小于边界的值作为测试数据，而不是选取等价类中的典型值或任意值。

2.3 场景法测试

场景法是通过运用场景来对系统的功能点或业务流程的描述，从而提高测试效果的一种方法。



基本流：系统从初始态到终止态的最主要的业务流程。测试中至少要确保系统基本流的执行是完全正确的。

备选流：备选事件流，以基本流为基础，在基本流所经过的每个判定节点处满足的不同触发条件而导致的其他事件流。

场景：可以看做是基本流与备选流的有序集合。一个备选流场景中至少包含一条基本流。

例子：见 ppt。

变异测试

变异测试是一种对**测试集的充分性**进行评估的技术，以创建更有效的测试集。

变异测试应该与传统的测试技术结合，而不是取代它们。

基本思想：给定一个程序 P 和一个测试数据集 T ，通过变异算子为 P 产生一组变异体 M_i （合乎语法的变更），对 P 和 M 都使用 T 进行测试运行：

- 如果某 M_i 在某个测试输入 t 上与 P 产生不同的结果，则该 M_i 被杀死；
- 如果某 M_i 在所有的测试数据集上都与 P 产生相同的结果，则称其为活的变异体；
- 接下来对活的变异体进行分析，检查其是否等价于 P ：对不等价于 P 的变异体 M 进行进一步的测试，直到充分性度量达到满意的程度。

概念：

- P' 称为 P 的变异体
- 如果对于 T 中的测试 t ，有 $P(t) \neq P'(t)$ ，称作 P' 与 P 有区别，或者 t 杀死 P' 。
- 如果 T 中所有的测试 t 使得 $P(t) = P'(t)$ ，称 T 不能区别 P 和 P' ，那么称在测试过程中 P' 是活的。
- 如果在程序 P 的输入域中不存在任何测试用例 t 使得 P 与 P' 区别，则称 P' 等价于 P 。
- 如果 P' 不等价于 P ，且 T 中没有测试能够将 P' 与 P 区别，则认为 T 是不充分的。
- 不等价而且是活的变异体为测试人员提供了一个生成新测试用例的机会，进而增强测试 T 。

变异分数的计算： T 的变异分数记为 $MS(T)$ ，则：

$$MS(T) = \frac{|D|}{|L| + |D|} = \frac{|D|}{|M| - |E|}$$

其中， $|D|$ 表示杀死的变异体数， $|L|$ 表示活着的变异体数， $|E|$ 表示等价的变异体数， $|M|$ 表示生成的所有变异体数。

一阶和高阶变异体：通过一次改变获得的变异体被认为是一阶突变；通过两次改变获得的突变体为二阶突变体。同样，可以定义高阶突变体。在实践中仅产生一级突变体，原因有两个：(a) 降低测试成本；(b) 大多数高阶突变体通过与一阶突变体相关的测试被杀死。

变异算子：

Mutant operator	In P	In mutant
Variable replacement	$z=x*y+1;$	$x=x*y+1;$ $z=x*x+1;$
Relational operator replacement	$if\ (x<y)$	$if(x>y)$ $if(x<=y)$
Off-by-1	$z=x*y+1;$	$z=x*(y+1)+1;$ $z=(x+1)*y+1;$
Replacement by 0	$z=x*y+1;$	$z=0*y+1;$ $z=0;$
Arithmetic operator replacement	$z=x*y+1;$	$z=x*y-1;$ $z=x+y-1;$

优点:

- 排错能力强
发现错误的能力较强——分析评估的结果
- 自动化程度高
测试工具自动产生变异体，自动运行 P 和 M ，自动发现被杀死的变异体
- 灵活性高
通过与测试提供工具的交互，有选择地使用功能变异算子
- 变异体与被测试程序的差别信息可以较容易地发现软件的错误
- 可以完成语句覆盖和分支覆盖
将每条语句或每个条件进行突变

缺点:

- 需要大量的计算机资源完成充分性分析
 n 行程序产生 $O(n^2)$ 变异体
存储变异体的开销大
变异体与被测试程序的等价判断需人工判定（判断两个程序是否等价是不可判定的命题）

性能测试

1 性能测试的关注点

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的要求。

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。

- 压力测试：通过对程序施加越来越大的负载，直到发现程序性能下降的拐点。——强调压力大小变化
- 负载测试：不断增加压力或增加一定压力下持续一段时间，直到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。——强调数据处理能力
- 配置测试：通过测试找到系统各项资源的最优分配原则。——强调资源优化配置
- 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

性能测试的应用：

- 系统性能瓶颈定位：最常见的应用
- 系统参数配置：寻找让系统表现最优的软、硬件配置参数
- 发现算法性能缺陷：针对多线程、同步并发算法
- 系统验收测试：验证预期性能指标
- 系统容量规划：测试不同硬件环境下的性能表现
- 产品评估/选型：全面评估产品，选择适合自己的产品类型

2 性能测试的指标

- 并发用户数量

狭义并发：所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(Vuser)：模拟真实用户操作来使用软件

- 响应时间

场景(Scenario)：在一个性能测试期间依次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

3 性能测试的模型

模型/策略:

- 预期指标性能测试: 针对需求和设计阶段提出的性能指标而进行的测试 (性能测试基本要求)
- 独立业务性能测试: 核心业务模块是性能测试重点
- 组合业务性能测试: 模拟多个用户, 对多项业务进行组合性能测试 (接近用户使用系统的真实情形)
- 疲劳强度性能测试: 以一定负载长时间运行系统, 测试系统长时间处理大量业务的能力和稳定性
- 大数据量性能测试: 测试运行时数据量较大或历史数据量较大时的性能情况
- 网络性能测试: 确认带宽、延迟、负载和端口变化对用户响应时间的影响
- 服务器性能测试: 确认数据库、Web服务器等瓶颈

原则:

- 成本最优原则: 投入的测试成本能否使系统满足预先确定的性能目标
- 策略为中心原则: 测试策略决定测试用例设计和测试实施
- 适当裁剪原则: 对测试用例包含的内容进行合理裁剪, 既满足测试需求, 又节约测试成本
- 模型完善原则: 具体测试中对模型不断调整、补充、完善, 使之满足测试目的

4 测试自动化的原理

自动化测试优点:

- 测试速度快
- 测试结果准确
- 高复用性
- 永不疲劳
- 可靠: 计算机不会弄虚作假
- 能力: 有些手工测试做不到的地方, 自动化测试可以做到。比如, 在性能测试中模拟1000个用户同时访问网站

正确认识自动化测试:

- 自动化测试有助于提高软件开发效率、缩短开发周期、节省人力资源等
- 机器执行测试用例按部就班, 没有变通和创造力

- 人具有创造力，可以举一反三，从一个测试想到其他测试场景
- 资料显示，良好的自动化测试只能发现 30% 的问题，70% 的问题还是 由人工发现
- 对于复杂的逻辑判断、界面是否友好人工测试具有优势
- 自动化测试更适合于负载测试、性能测试和回归测试