CHAPTER 5

第 5 章

单元测试与集成测试

按阶段进行测试是一种基本的测试策略，代码的静态测试和单元测试是测试执行过程中的第一个阶段，本章主要从代码静态测试和单元测试的定义、目标、过程、技术与方法、评估等方面进行介绍和讨论，并澄清在这一测试阶段存在的一些误区。然后介绍集成测试，确保各个单元能正常结合起来形成所要构成的系统。现在人们越来越强调持续构建、持续集成和持续测试，单元测试和集成测试往往交替进行、同步进行，但概念上还是先单元测试，后集成测试，所以本章内容还是这样安排的。

在测试过程中应该依据每一个阶段的不同特点，采用不同的测试方法和技术，制定不同的测试目标。在单元测试或集成测试阶段中主要采用白盒测试方法，包括对代码的评审、静态分析等静态测试和结合测试工具进行动态测试。

5.1 代码静态测试

静态测试技术是代码级测试中最重要的手段之一，适用于新开发的和重用的代码，通常在代码完成并无错误地通过编译或汇编后进行。代码静态测试分为人工和自动化两种方式，即代码评审和采用工具进行扫描分析。测试人员主要由软件开发人员及其开发小组成员组成。

5.1.1 编码的标准和规范

代码即使可以正常运行，但是不符合某种标准和规范，仍然会给将来程序维护带来隐患。标准是建立起来和必须遵守的规则——做什么和不做什么，而规范是建议如何去做，推荐更好的工作方式，如自定义变量和函数的命名。标准没有例外情况，是结构严谨的，规范就没有那么严格，相对松一些。在一些正规的项目中，经常有一些在测试中表现稳定的软件，因为不符合规范而被认为有问题，为什么呢？至少有以下三个重要原因可以说明要坚持标准和规范。

(1) 可靠性。事实证明按照某种标准或规范编写的代码比不这样做的代码更加可靠，软件缺陷更少。

(2) 可读性和维护性。符合设备标准和规范的代码易于阅读、理解

和维护。

(3) 移植性。代码经常需要在不同的硬件上运行,或者使用不同的编译器编译,如果代码符合标准,迁移到另一个平台就会相对容易,甚至完全没有障碍。

代码中最常用的是表达式,而表达式通常由变量、函数、常数和运算符组成,通过运算符将变量、函数、常数组合成合理、有效的表达式。变量通常分为系统变量和自定义变量,自定义变量又分为全局变量和局部变量,因此在检查代码时首先要检查变量定义的对不对,有没有对变量赋予初始值,变量的命名是否正确以及命名是否符合规范。除了变量和函数外,代码中还有谓语动词语句,如 C 语言中的 goto、do-while 和 if-else 语句就有它们的编程标准,而目前流行编程语言中,如 C++、Java 等都设立了使用它们的标准。例如,著名的 MISRA C Coding Standard,这一标准中包括了 127 条 C 语言编码标准。通常认为,如果能够完全遵守这些标准,则所写的 C 代码是易读、可靠、可移植和易于维护的,如:

(1) 不得使用类型 char,必须显式声明为 unsigned char 或者 signed char。

(2) 所有数字常数应当加上合适的后缀表示类型,如 51L, 42U, 34.12F 等。

(3) 不得定义与外部作用域中某个标识符同名的对象,以避免遮盖外部作用域中的标识符。

(4) 具有文件作用域的对象尽量声明为 static。

(5) 同一个编译单元中,同一个标识符不应该同时具有内部链接和外部链接的声明。

每个开发项目由于自身特点都必须符合一组标准,除必须符合计算机语言标准外还需要符合相应的行业标准,如金融系统、航天系统的软件都有各自严格的标准。如果想获得计算机软件和信息技术国家的相关国际标准,可以通过以下站点获得。

(1) 美国国家标准会(ANST): www.ansi.org

(2) 国际工程协议(IEC): www.iec.org

(3) 国际标准化组织: www.iso.ch

(4) 美国计算机械联合会(ACM): www.acm.org

(5) 国际电子电气工程学会(IEEE): www.ieee.org

在软件工程领域,源程序的风格统一标志着可维护性、可读性,是软件项目的一个重要组成部分。如果没有成文的编码风格文档,以至于很多时候,程序员没有一个共同的标准可以遵守,编码风格各异,程序可维护性、可读性差。通过建立代码编写规范,形成开发小组编码约定,提高程序的可靠性、可读性、可修改性、可维护性、可继承性和一致性,可以保证程序代码的质量,继承软件开发成果,充分利用资源,使开发人员之间的工作成果可以共享。

以下是一个实际项目小组曾参考使用过的 Java 代码的书写规范。由于篇幅较长,略去其中部分内容,以供参考。

Java 代码书写规范示例

一、目的(略)

二、整体编码风格

1. 缩进

缩进建议以 4 个空格为单位。建议在 Tools/Editor Options 中设置 Editor 页面的 Block ident 为 4,Tab Size 为 8。预处理语句、全局数据、标题、附加说明、函数说明、标号等均顶格书写。语句块的“{”、“}”配对对齐,并与其前一行对齐,语句块类的语句缩进建议每个“{”、“}”单独占一行,便于匹配。JBuilder 默认方式是开始的“{”不是单独一行,建议更改成上述格式(在 Project/Default Project Properties 的 Code Style 中选择 Braces 为 Next line)。

2. 空格

原则上变量、类、常量数据和函数在其类型和修饰名称之间适当空格并据情况对齐。关键字原则上空一格，如：if（…）等。运算符的空格规定如下："::"、"→"、"["、"]"、"++"、"--"、"~"、"!"、"+"(正号)、"-"(负号)、"&"(引用)等几个运算符两边不加空格(其中单目运算符指与操作数相连的一边)，其他运算符(包括大多数二目运算符和三目运算符"?:")两边均加一空格，在作函数定义时还可据情况多空或不空格来对齐，但在函数实现时可以不用。","运算符只在其后空一格，需对齐时也可不空或多空格。不论是否有括号，对语句行后加的注释应用适当空格与语句隔开并尽可能对齐。此项可以依照个人习惯决定遵循与否。

3. 对齐

原则上，关系密切的行应对齐，对齐包括类型、修饰、名称、参数等各部分对齐。每一行的长度不应超过屏幕太多，必要时适当换行，换行时尽可能在","处或运算符处，换行后最好以运算符打头，并且以下各行均以该语句首行缩进，但该语句仍以首行的缩进为准，即如其下一行为"{"应与首行对齐。

变量定义最好通过添加空格形成对齐，同一类型的变量最好放在一起。如下例所示：

```
int       Value;
int       Result;
int       Length;
Object    currentEntry;
```

此项可以依照个人习惯决定遵循与否。

4. 空行

不得存在无规则的空行，比如连续 10 个空行。程序文件结构各部分之间空两行，若不必要也可只空一行，各函数实现之间一般空两行，由于每个函数还要有函数说明注释，故通常只需空一行或不空，但对于没有函数说明的情况至少应再空一行。对自己写的函数，建议也加上"//------"做分隔。函数内部数据与代码之间应至少空一行，代码中适当处应以空行空开，建议在代码中出现变量声明时，在其前空一行。类中 4 个"p"之间至少空一行，在其中的数据与函数之间也应空行。

5. 注释

注释是软件可读性的具体体现。程序注释量一般占程序编码量的 20%，软件工程要求不少于 20%。程序注释不能用抽象的语言，类似于"处理"、"循环"这样的计算机抽象语言，要精确表达出程序的处理说明。例如，"计算净需求"、"计算第一道工序的加工工时"等。避免每行程序都使用注释，可以在一段程序的前面加一段注释，具有明确的处理逻辑。

注释必不可少，但也不应过多，不要被动地为写注释而写注释。以下是 4 种必要的注释。

(1) 标题、附加说明。

(2) 函数、类等的说明。对几乎每个函数都应有适当的说明，通常加在函数实现之前，在没有函数实现部分的情况下则加在函数原型前，其内容主要是函数的功能、目的、算法等说明，参数说明、返回值说明等，必要时还要有一些如特别的软硬件要求等说明。公用函数、公用类的声明必须由注解说明其使用方法和设计思路，当然选择恰当的命名格式能够帮助把事情解释得更清楚。

(3) 在代码不明晰或不可移植处必须有一定的说明。

(4) 少量的其他注释,如自定义变量的注释、代码书写时间等。

注释有块注释和行注释两种,分别是指:"/ ** /"和"//"建议对(1)用块注释,(4)用行注释,(2)、(3)则视情况而定,但应统一,至少在一个单元中(2)类注释形式应统一。具体对不同文件、结构的注释会在后面详细说明。

6. 代码长度

对于每个函数建议尽可能控制其代码长度不超过53行,超过53行的代码要重新考虑将其拆分为两个或两个以上的函数。函数拆分规则应该以不破坏原有算法为基础,同时拆分出来的部分应该是可以重复利用的。对于在多个模块或者窗体中都要用到的重复性代码,完全可以将其独立成为一个具备公用性质的函数,放置于一个公用模块中。

7. 页宽

页宽应该设置为80字符。源代码一般不会超过这个宽度,否则会导致无法完整显示,但这一设置也可以灵活调整。在任何情况下,超长的语句都应该在一个逗号或者一个操作符后折行。一条语句折行后,应该比原来的语句再缩进两个字符。

8. 行数

一般的集成编程环境下,每屏大概只能显示不超过50行的程序,所以这个函数大概要5或6屏显示,在某些环境下要8屏左右才能显示完。这样一来,无论是读程序还是修改程序,都会有困难。因此建议把完成比较独立功能的程序块抽出,单独成为一个函数;把完成相同或相近功能的程序块抽出,独立为一个子函数。可以发现,越是上层的函数越简单,就是调用几个子函数,越是底层的函数完成的越是具体的工作。这是好程序的一个标志。这样,就可以在较上层函数里更容易地控制整个程序的逻辑,而在底层的函数里专注于某方面的功能实现。

三、代码文件风格(略)

四、函数编写风格(略)

五、符号风格(略)

5.1.2 代码评审

代码审查(Code Review)也是一种有效的测试方法。据有关数据统计,代码中60%以上的缺陷可以通过代码审查(包括互查、走查、会议评审等形式)发现。代码审查不仅能有效地发现缺陷,而且为缺陷预防获取各种经验,为改善代码质量打下坚实的基础。即使没有时间完成所有代码的检查,也应该尽可能去做,哪怕是对其中一部分代码进行审查。人们也为代码审查进行了大量的探索,获得了一些最佳实践,例如:

(1) 一次检查200～400行代码,不宜超过60～90min。

(2) 合适的检查速度:每小时300～500行代码。

(3) 在审查前,代码作者应该对代码进行注释。

(4) 建立量化的目标并获得相关的指标数据,从而不断改进流程。

(5) 使用检查表(Checklist)肯定能提高评审效果。

1. 代码走查

代码互查是日常工作中使用最多的一种代码评审方式,比较容易开展,相对自由,而走

查(Walk Through)是一种相对比较正式的代码评审过程。在此过程中,设计者或程序员引导小组部分成员通读编码,其他成员提出问题并对有关技术、风格、可能的错误、是否有违背开发标准/规范的地方等进行评论。走查过程中,由测试成员提出一批测试实例,在会议上对每个测试实例用头脑来执行程序,在纸上或黑板上演变程序的状态。在这个过程中,测试实例并不起关键作用,它们仅作为怀疑程序逻辑与计算错误的参考。大多数走查中,在怀疑程序的过程中所发现的缺陷比通过测试实例本身发现的缺陷更多。编程者对照讲解设计框图和源码图,特别是对两者相异之处加以解释,有助于验证设计和实现之间的一致性。

2. 正式会议审查

会议审查(Inspection)是一种最为正式的检查和评估方法,最早是由 IBM 公司提出的,经实践证明,是一种有效的检查方法,从而得到软件工程界的普遍认同。它是用逐步检查源代码中有无逻辑或语法错误的办法来检测故障。可以认为它是拿代码与标准和规范对照的补充,因为它不但需要软件开发者自查,还要组织代码检查小组进行代码检查。代码检查小组通常由独立的主持人(协调员)、程序编写小组、其他组程序员和测试小组成员组成。代码检查程序如下:主持人提前把程序目录表和设计说明分配给小组各成员,小组成员在开会前先熟悉这些材料,然后开会。在会议上,主要的工作如下。

(1) 由程序编写小组成员逐句阐明程序的逻辑,在此过程中可由程序员或测试小组成员提出问题,追踪缺陷是否存在。

(2) 利用通用缺陷检查表来分析讨论。主持人负责讨论沿着建设性方向进行,而其他人则集中注意力发现缺陷。

(3) 记录所有已确定的缺陷,在会议之后形成《评审报告》。在《评审报告》中必须写明错误的位置、类型、影响范围和原因等,评审报告需交给程序编写者并同时存档。

(4) 审查小组根据代码审查的错误记录来评估该程序,决定是否需要重新进行审议。如发现太多缺陷,那么在改正缺陷之后,可能需要再开评审会议。

无论是走查还是正式的会议审查,都需要注意限时和避免现场修改。限时是为了避免跑题,不要针对某个技术问题进行无休止的讨论。发现问题时不要现场修改,适当地进行记录,会后再进行修改是必要的,否则会浪费大家的时间。会议主持人要牢记会议的宗旨和目标。检查的要点是代码编写是否符合标准和规范,是否存在逻辑错误。

在审查会前项目经理要制定或维护好代码缺陷检查表,检查表的内容主要是检查的要点,作为评审的检查依据、主要参考资料。在评审会上项目组的每一个人员都能看到自己和其他人员的编码问题,也是大家很好的学习机会,从而起到缺陷预防的作用。评审会中确定的所有缺陷都要被解决,并且解决的结果可能需要评审会主持人或项目经理的确认,如果需要,再上评审会确认。评审通过的准则如下。

(1) 充分审查了所规定的代码,并且全部编码准则被遵守。

(2) 审查中发现的错误已全部修改。

3. 走查与会议审查的对比

走查与会议审查的对比如表 5-1 所示。

表 5-1 走查与会议审查的对比

对比内容	走　查	会议审查
准备	通读设计和编码	应准备好需求描述文档、程序设计文档、程序的源代码清单、代码编码标准和代码缺陷检查表
形式	非正式会议	正式会议
参加人员	开发人员为主	项目组成员包括测试人员
主要技术方法	无	缺陷检查表
注意事项	限时、不要现场修改代码	限时、不要现场修改代码
生成文档	会议记录	静态分析错误报告
目标	代码标准规范,无逻辑错误	代码标准规范,无逻辑错误

4. 缺陷检查表

检查过程所采用的主要技术是设计与使用缺陷检查表。这个表通常是把程序设计中可能发生的各种缺陷进行分类,以每一类列举尽可能多的典型缺陷,然后把它们制成表格,以供在会议中使用,并且在每次审议会议之后,对新发现的缺陷也要进行分析和归类,不断充实缺陷检查表。缺陷检查表会因项目不同而不同,在实际工作中不断积累完善,使用缺陷检查表的目的是防止人为的疏漏。下面就是一个代码检查表的示例,这个示例只对结构化编程测试具有普遍和通用的意义。

代码评审的通用检查表

1. 格式

(1) 嵌套的 IF 是否正确地缩进?

(2) 注释是否准确并有意义?

(3) 是否使用有意义的标号?

(4) 代码是否基本上与开始时的模块模式一致?

(5) 是否遵循全套的编程标准?

2. 程序语言的使用

(1) 是否使用一个或一组最佳动词?

(2) 模块中是否使用完整定义的语言的有限子集?

(3) 是否使用了适当的转移语句?

3. 数据引用错误

(1) 是否引用了未初始化的变量?

(2) 数组和字符串的下标是整数值吗? 下标总是在数组和字符串大小范围内吗?

(3) 是否在应该使用常量的地方使用了变量,如在检查数组范围时?

(4) 变量是否被赋予了不同类型的值?

(5) 为引用的指针分配内存了吗?

(6) 一个数据结构是否在多个函数或者子程序中引用,是否在每一个引用中明确定义了结构?

4. 数据声明错误

(1) 所有变量都被赋予正确的长度、类型和存储类了吗? 例如,本应声明为字符串的变量声明为字符数组了。

(2) 变量是否在声明的同时进行了初始化？是否正确初始化并与其类型一致？

(3) 变量有相似的名称吗？是否自定义变量使用了系统变量名？

(4) 存在声明过，但从未引用或者只引用过一次的变量吗？

(5) 在特定模块中所有变量都显式声明了吗？如果没有，是否可以理解为该变量与更高级别的模块共享？

5. 计算错误

(1) 计算中是否使用了不同数据类型的变量？例如，将整数与浮点数相加。

(2) 计算中是否使用了不同数据类型的变量？例如，将字节与字相加。

(3) 计算时是否了解和考虑到编译器对类型和长度不一致的变量的转换规则？

(4) 赋值的目的变量是否小于赋值表达式的值？

(5) 在数值计算过程中是否可能出现溢出？

(6) 除数/模是否可能为零？

(7) 对于整型算术运算，特别是除法的代码处理是否会丢失精度？

(8) 变量的值是否超过有意义的范围？

(9) 对于包含多个操作数的表达式，求值的次序是否混乱，运算优先级对吗？

6. 比较错误

(1) 比较得正确吗？虽然听起来容易，但是比较中应该是小于还是小于或等于常常发生混淆。

(2) 存在分数或者浮点值之间的比较吗？如果有，精度问题会影响比较吗？

(3) 每个逻辑表达式都正确表达了吗？逻辑计算如期进行了吗？求值次序有疑问吗？

(4) 逻辑表达式的操作数是逻辑值吗？例如，是否包含整数值的整型变量用于逻辑计算中？

7. 入口和出口的连接

(1) 初始入口和最终出口是否正确？

(2) 对另一个模块的每次调用是否恰当？例如：全部所需的参数是否传送给每个被调用的模块？被传送的参数值是否正确地设置？对关键的被调用模块的意外情况(如丢失、混乱)是否处理？

(3) 每个模块的代码是否只有一个入口和一个出口？

8. 存储器的使用

(1) 每个域在其第一次被使用前是否正确初始化？

(2) 规定的域是否正确？

(3) 每个域是否有正确的变量类型声明？

9. 控制流程错误

(1) 如果程序包含 begin-end 和 do-while 等语句组，end 是否对应？

(2) 程序、模块、子程序和循环能否终止？如果不能，可以接受吗？

(3) 可能存在永远不停的循环吗？

(4) 存在循环从不执行吗？如果是这样，可以接受吗？

(5) 如果程序包含像 switch-case 语句这样的多个分支，索引变量能超出可能的分支数目吗？如果超出，该情况能正确处理吗？

(6) 是否存在“丢掉一个”错误,导致意外进入循环?
(7) 代码执行路径是否已全部覆盖?是否能保证每条源代码语句至少执行一次?

10. 子程序参数错误

(1) 子程序接收的参数类型和大小与调用代码发送的匹配吗?次序正确吗?
(2) 如果子程序有多个入口点,引用的参数是否与当前入口点没有关联?
(3) 常量是否当作形式参数传递,意外在子程序中改动?
(4) 子程序是否更改了仅作为输入值的参数?
(5) 每一个参数的单位是否与相应的形参匹配?
(6) 如果存在全局变量,在所有引用子程序中是否有相似的定义和属性?

11. 输入/输出错误

(1) 软件是否严格遵守外部设备读写数据的专用格式?
(2) 文件或者外设不存在或者未准备好的错误情况有处理吗?
(3) 软件是否处理外部设备未连接、不可用或者读写过程中存储空间占满等情况?
(4) 软件以预期方式处理预计的错误吗?
(5) 检查错误提示信息的准确性、正确性、语法和拼写了吗?

12. 逻辑和性能

(1) 是否已实现全部设计?
(2) 逻辑是否被最佳地编码?
(3) 是否提供正式的错误/例外子程序?
(4) 每一个循环是否执行正确的次数?

13. 维护性和可靠性

(1) 清单格式是否适于提高可读性?
(2) 标号和子程序是否符合代码的逻辑意义?
(3) 对从外部接口采集的数据是否有确认?
(4) 是否遵循可靠性编程要求?
(5) 是否存在内存泄露的问题?

5.2 代码评审案例分析

在代码评审中能够发现比较多的问题,有些问题是常见的,有些问题偶尔出现的,可以从中学习并整理成检查表,有助于未来更好地进行代码评审。下面通过一些常见的问题来建立代码评审的强烈意识和培养良好的基本能力。

5.2.1 空指针保护错误

空指针保护错误(Null Pointer Exception)应该是 Java 程序中最常见的一类错误,通过合理的编码规则、开发者对此类问题的理解程度和测试人员的 Case 覆盖率来避免此类错误。

1. 测试场景

某站点通过用户输入的用户名与密码来判断出现什么页面,是管理员页面、站点普通用户页面,还是匿名访问的用户页面。不同的人访问页面的权限与页面上的元素都是不尽相同的。

管理员有管理普通用户的功能，以及站点的其他管理类操作；站点普通用户可以进行站点的普通操作，比如某些文档程序只有注册登录的合法用户才能下载；匿名用户一般只能访问一些公共的资源，此权限一般是站点最小的。程序代码如下：

```
21  /**
22   * 通过用户UI界面输入的用户名，传递到Action层，进行用户角色识别操作
23   *
24   * @param request HttpServletRequest
25   *
26   * @return String 用户角色，像管理员/普通用户/...
27   */
28  public String getUserRole(HttpServletRequest request) {
29      String userRole = "";
30      String userName = request.getParameter("userName");
31      if (userName.equals("schadmin")) {
32          //这是系统初始化时默认的管理员账号，如果是，则做以下的验证操作.....
33      }
34      //非系统初始化的账号，做以下验证操作.....
35      return userRole;
36  }
```

2. 分析

程序第 31 行，在特定的 Case 下，有可能会出 NullPoint（空指针）错。比如匿名用户访问页面时，可能就没有输入用户名。这类问题看起来很简单，如果程序开发人员平时不注意，则可能会导致某些情况下页面无法正常工作。

3. 解决方案

按正确的规则来使用常量.equals（变量），这样就可以省去以后的很多麻烦，同时保证了函数的健壮性，也减轻了因为代码自身的错误给测试人员带来的工作量。正确的代码如下：

```
28  public String getUserRole(HttpServletRequest request) {
29      String userRole = "";
30      String userName = request.getParameter("userName");
31      if ("schadmin".equals(userName)) {
32          //这是系统初始化时默认的管理员账号，如果是，则做以下的验证操作.....
33      }
34      //非系统初始化的账号，做以下验证操作.....
35      return userRole;
36  }
```

4. 要求

开发人员在写代码时要遵循规则去做，同时要经常审查所做的代码，修改不符合规则的代码。测试人员也要积累相关的经验，比如在页面输入的地方不输入内容，检查是否有合理的保护；想想有没有其他的 Case 能绕过输入的页面而访问其后继的页面；在做 API 测试时，相应的参数值被置空，检查是否出错等。

5.2.2 数据类型转换错误

数据类型转换错误（Number Format Exception）也是平时测试过程中常见的问题，Java 自身具有的 Integer.parseInt()；Long.parseLong()方法在数据类型转换时没有对传入参数的合法性进行判断。如果在代码中没有对传入参数做合法性检查就直接调用 Java 中的方法，在某些 Case 下就会抛错，致使程序或页面无法正常进行。

1. 测试场景

用户注册时要输入年龄字段，用户输入的参数传入 Action 层，通过 request.getParameter()获得参数值时，返回的是字符型。而数据库中该字段为数值型，所以需要做相应的数据类型转换。

程序代码如下：

```
40    /**
41     * 通过用户输入的年龄，转换为数值型
42     *
43     * @param request HttpServletRequest
44     *
45     * @return Integer 用户年龄
46     */
47    public int getUserAge(HttpServletRequest request) {
48        int age = 0;
49        String userAge = request.getParameter("userAge");
50        if (userAge != null) {
51            age = Integer.parseInt(userAge);
52        }
53        return age;
54    }
```

2. 分析

程序第 51 行，虽然在 50 行已考虑到了 NullPoint 的保护，但对于传过来的不是数字的参数没有做必要的保护。

3. 解决方案

因为一个项目中进行数据类型转换的地方应该很多，所以建议写一个 Util 工具类，实现一些常用的数据转换的方法，以供调用。建议代码如下：

```
58    /**
59     * 对传入的字符型转换为整型值
60     *
61     * @param intStr String
62     * @return Integer
63     */
64    public static int getIntValue(String intStr) {
65        int pareseInt = 0;
66        if (isNumeric(intStr)) { //isNumeric是判断传入的变量值是否为数值类型
67            pareseInt = Integer.parseInt(intStr);
68        }
69        return pareseInt;
70    }
```

4. 要求

开发人员在写代码时，遇到数值转换要使用公共的安全方法(做过保护的)，同时要经常复审代码，修改不符合规则的代码。测试时，要关注类似的问题，如在应输入数字的地方输入非数字内容、边界值、特殊符号等，以验证是否有异常保护。

5.2.3 字符串或数组越界错误

字符串或数组越界错误(Out Of Bounds Exception)也是常见的问题之一。

1. 测试场景

按程序约定电话号码有如下 4 部分组成：国家编码，区位号码，电话号码，分机号，中间用逗号分隔，进行传输操作与数据库存取。假设系统想取出电话号码值或分机号值，类似这样的操作经常因保护不够而出越界错误。程序代码如下：

2. 分析

程序第 23 行，虽然从表面上看没有问题，但如果取出(传过)来的数据，本来就没有电话号码或没有分机号，则会出字符串或数组越界错误。

```
/**
 * 假设电话号码字串设计的标准格式为:国家编码，区位号码，电话号码，分机号
 * 举例如86,0551,2313222,8093
 *
 * @param strPhoneNumber String
 *
 * @return String 电话号码(如:例子中的2313222)
 */
public static String getPhoneNumber(String strPhoneNumber) {
    if ((strPhoneNumber == null) || "".equals(strPhoneNumber)) {
        return "";
    }
    String[] arrPhone = strPhoneNumber.split(",");
    return arrPhone[2];
}
```

3. 解决方案

需要做好字符串或数组越界错误保护，才能供调用。建议代码如下：

```
public static String getPhoneNumber(String strPhoneNumber) {
    if ((strPhoneNumber == null) || "".equals(strPhoneNumber)) {
        return "";
    }
    String[] arrPhone = strPhoneNumber.split(",");
    if (arrPhone.length > 2) {
        return arrPhone[2];
    }
    return "";
}
```

4. 要求

在遇到截取字符串或取数组指定下标值前一定要进行异常保护。另外，Java 的数组下标是从 0 开始的。在测试时，如果有分机号，可以保留其为空白，因为现实中就有电话号码不设分机号的；其他内容也可置为空白，以测试程序的健壮性。类似这样的测试案例有许多，值得关注。

5.2.4　资源不合理使用

1. 测试场景

上传/下载文件、向文件中写入内容、将文件中的内容读出等功能，如果在操作结束时忘记关闭流文件，则当频繁使用时会导致 Web 服务器的性能下降，甚至导致服务器崩溃。程序代码如下：

```
public static void writeStringFile(File file, String writeContent,
        String encoding) throws FileOperatorException {
        FileOutputStream fos = null;
        try {
            if(!file.exists()) {
                file.createNewFile();
            }
            fos = new FileOutputStream(file);
            fos.write(writeContent.getBytes(encoding));
        } catch (Exception ex) {
            throw new FileOperatorException (ex);
        } finally{  //如果没有 finally 下面的段
            if(fos != null) {
                try{
                    fos.close();
```

```
                } catch (IOException ioe) {
                    throw newFileOperatorException(ioe);
            }
        }
    }
}
```

2. 分析

这段代码在 finally 中最后做了关闭流操作，这是正确的并且安全的写法。如果没有 finally 这段代码，或是把 finally 中的关闭流方法写到了 try 中或 catch 中，那都是很危险的，迟早会出问题。Java 中的“try-catch-finally”结构可以这样理解：

(1) try 块中的内容在无异常发生时执行到结束。

(2) 在 try 块中内容发生 catch 所声明的异常时，跳转到 catch 块执行。

(3) 无论是否发生异常，都会执行 finally 块的内容。

所以，在代码逻辑中，如果存在“无论发生什么都必须执行的代码”，则应放在 finally 块中。最常见的就是把关闭连接、释放资源等类似的代码放在 finally 块中。

3. 要求

上述错误在测试中往往不容易发现，可能要等到服务器运行了一段时间以后，服务器在某个峰值上崩溃了才知道，而想复现它又很难。测试人员可以通过阅读源代码找出此类错误，或通过集成测试、压力测试来发现类似的问题，另外，让服务器运行一段时间后，通过查看错误日志(Error Log)也比较容易发现其中的问题。

5.2.5 不当使用 synchronized 导致系统性能下降

1. 测试场景

某网站专门组织各类活动，如演讲比赛、足球赛、舞会等，需要给所有用户发送 Email。程序代码如下：

```
public synchronized static void sendMail(String templateName,
        Map replaceMap, Event event, String sender, String replyTo,
Locale curlocale) {
        //组织每封信的需替换的内容
replaceMap.put("EventName", event.getEventName());
replaceMap.put("EventDesc", event.getEventDescription());
replaceMap.put("StartTime",
TimeUtil.formatDateAndTime(event.getStartTime(),curlocale));
replaceMap.put("EndTime",
TimeUtil.formatDateAndTime(event.getEndTime(), curlocale));
replaceMap.put("EventHost", event.getHost());
…

        //信的模板，收件人，发件人，收件人的语言等准备
MailTBO mailTBO = new MailTBO();
mailTBO.setTemplateName(templateName);
mailTBO.setSender(sender);
mailTBO.setReplyTo(replyTo);
mailTBO.setLocale(curlocale);
```

```
mailTBO.setReplaceMap(replaceMap);
...

        //最后将准备好的内容发送出去
MailBizFactory bizFactory = MailBizFactory.getInstance();
EmailManager emailManager = (EmailManager)
    bizFactory.getManager(EmailManager.class);
emailManager.sendMail(mailTBO);
}
```

2. 分析

这里发邮件时用了 synchronized 方法。如果邀请 5～10 人时，一般不会有性能问题，但如果邀请超过 100 人，可能页面就长时间不动或导致系统性能严重下降，甚至 Web 服务器崩溃。如果像这样大的方法声明为 synchronized，将会严重影响系统的效率。典型地，若将线程类的方法 run() 声明为 synchronized，由于在线程的整个生命期内它一直在运行，容易导致它对本类任何 synchronized 方法的调用都永远不会成功。

5.3 代码静态检测工具

上面介绍了代码评审这一人工代码静态测试方法。但是，如果靠编程人员自行检查代码，不仅工作量大，而且测试过程缺少一致性、准确性和可靠性。最好是采用代码评审和自动的静态方法相结合的方式，即借助静态检测工具来完成代码静态测试(也叫静态分析)。

静态检测工具虽然要引入一些新规则，但其维护工作量很低，越来越受到人们的关注。如果将静态测试工具集成到项目的持续集成(Continuous Integration)环境中，通过不断的检查与修改来减少软件缺陷可能存在的地方，效果更好。

代码的静态检测工具比较多，包括：

(1) 支持 Java 语言检测的 CheckStyle、FindBugs、PMD 等。

(2) 支持 C++语言的 Parasoft C++Test、Helix QAC 等。

(3) 支持 Python 语言的 PyCharm、Pyflakes、PEP 8 等。

另外还有支持多种语言的、功能更为强大的代码质量管理平台 SonarQube。下面以 FindBugs、PMD、CheckStyle、SonarQube 为例，介绍如何使用静态测试工具进行代码的静态测试。

5.3.1 FindBugs 检查代码缺陷

FindBugs 实际是扫描和分析 Java 字节码(.class 文件)，如果选中.class 文件对应的源文件(.java 文件)，可以定位到出问题的代码行。FindBugs 支持在 JRE 环境中独立运行，也可以在 IntelliJ IDEA、Eclipse 等开发环境中运行。这里讲解如何在 Eclipse 中使用 FindBugs。FindBugs 的 Eclipse 插件位置为：http://findbugs.sourceforge.net/downloads.html，安装后，可以在 Eclipse 的 Preferences 中 Java 选项看到 FindBugs，用户可以完成其报告选项、过滤文件和规则等浏览和设置，如图 5-1 所示。

FindBugs 检查的问题有以下几类。

(1) 恶意的代码安全漏洞；

(2) 可疑的或危险的代码；

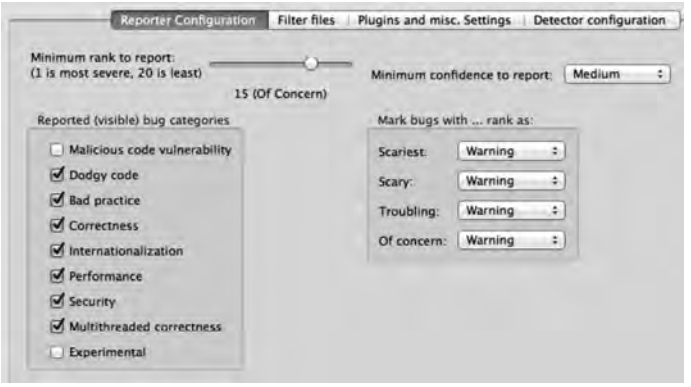

图 5-1 FindBugs 设置的主要界面

(3) 不良实践;

(4) 正确性;

(5) 国际化问题;

(6) 性能;

(7) 安全性;

(8) 多线程问题;

(9) 经验性的问题。

如果要了解上述各类问题的具体描述,可参见“探测器配置(Detector configuration)”选项卡,如图 5-2 所示。

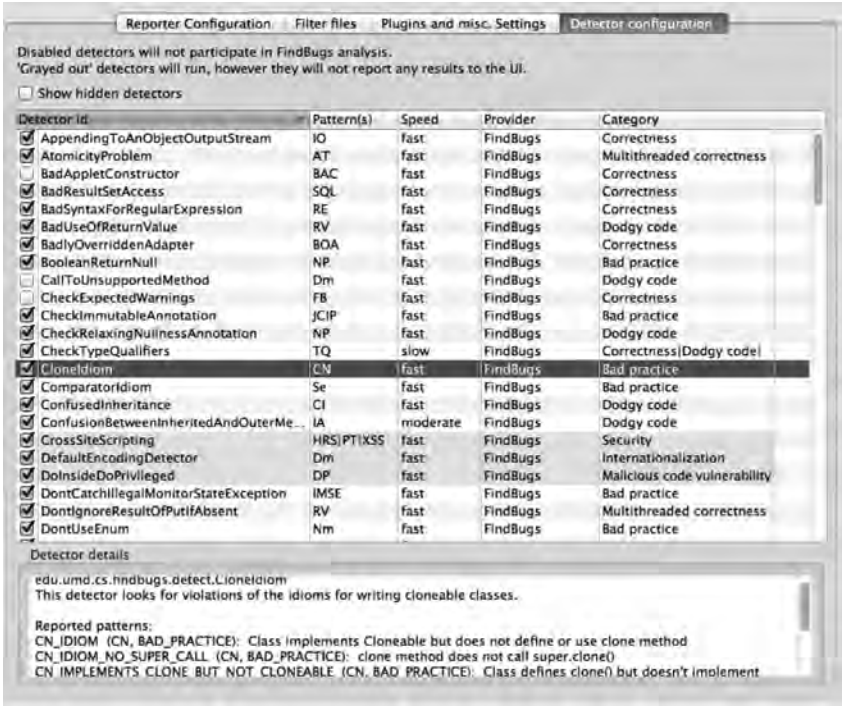

图 5-2 FindBugs 探测器的设置和说明

其运行很简单，选择要被测试的.class或源文件，右击选择FindBugs菜单，执行FindBugs，检查完成后生成报告。

5.3.2 PMD检查代码缺陷

PMD(http://pmd.sourceforge.net)是一款采用BSD协议发布的Java程序代码检查工具，其功能强、效率高，能检查Java代码中是否含有未使用的变量、空的抓取块、不必要的对象、过于复杂的表达式、冗余代码等。

PMD既支持单独安装运行，也可以在开发环境中运行。这里以PMD在IntelliJ IDEA中的使用为例。在IDEA中的Settings→Plugins→Marketplace中直接查找PMDPlugin进行安装。安装后，可以在Settings中以及IDEA界面下方看到PMD。PMD包含内置规则集，并支持用户编写自定义规则，可以在Settings中的PMD界面中添加自定义规则。

PMD为Java语言内置了以下7类规则集。

(1) 最佳实践(Best Practices)：检查代码是否遵循了公认最佳实践。

(2) 代码风格(Code Style)：强制遵守特定的代码风格。

(3) 设计(Design)：帮助用户发现设计问题的规则集。

(4) 文档(Documentation)：检查代码中的注释文档。

(5) 易错倾向(Error Prone)：用于检测损坏的、容易混淆或出现运行错误的代码结构。

(6) 多线程(Multithreading)：检查代码中处理多个执行线程的问题。

(7) 性能(Performance)：标记次优代码的规则集。

在IDEA中的代码编辑框或Project窗口的文件夹、包、文件右击选择Run PMD，再右击选择Pre Defined并选择规则集，就可以运行PMD，如图5-3所示。

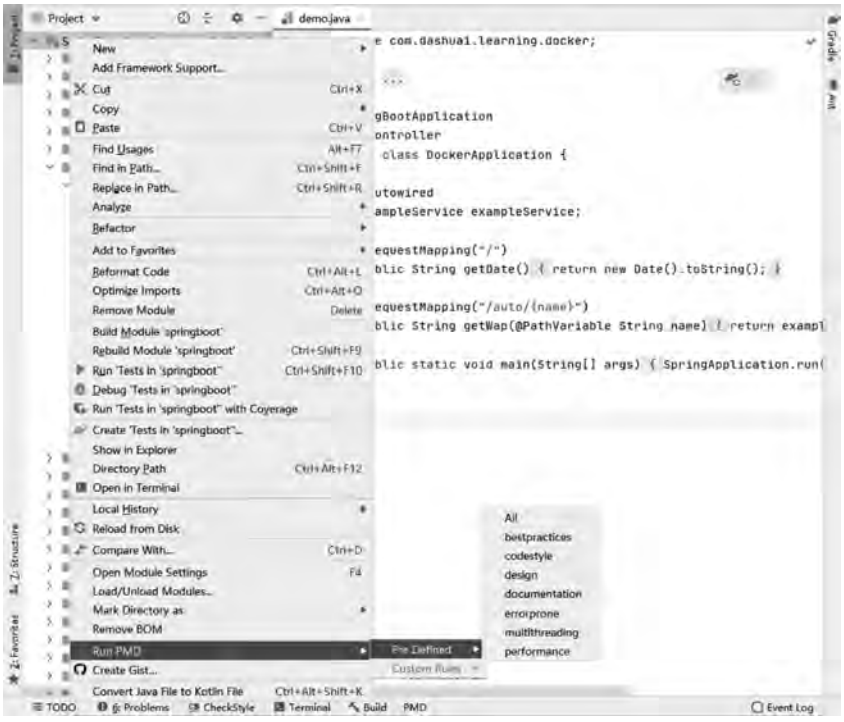

图 5-3 PMD的运行界面

5.3.3 CheckStyle 检查代码风格

CheckStyle(https://checkstyle.sourceforge.io)是 Java 代码风格检查工具。CheckStyle 能够根据代码规范自动地检查代码的风格,能够检查的主要内容包括以下几个方面。

(1) 注解;
(2) 代码块;
(3) 类设计;
(4) 编码问题;
(5) 文件头;
(6) Import 语句;
(7) Javadoc 注释;
(8) 复杂度;
(9) 混合检查;
(10) 修饰符;
(11) 命名约定;
(12) 正则表达式;
(13) 体积大小;
(14) 空白字符。

CheckStyle 也支持在开发环境中运行,可以在 IntelliJ IDEA 中选择安装 CheckStyle-IDEA 插件并对其进行配置。CheckStyle 插件默认支持 Oracle 和 Google 等的代码规范,用户也可以添加自定义的代码规范。

表 5-2 将三个静态分析工具 FindBugs、PMD、CheckStyle 进行了比较,以便进一步了解它们各自的特点。

表 5-2 FindBugs、PMD 与 CheckStyle 主要功能

工 具	目 的	检 查 项
FindBugs (检查.class)	基于 Bug Patterns 概念,查找 Java 源文件和 bytecode(.class 文件)中的潜在 Bug	bytecode 中的 Bug Patterns,如 code 性能、NullPoint 空指针检查、没有合理关闭资源、字符串相同判断错(==,而不是 equals)等
PMD (检查源文件)	检查 Java 源文件中的潜在问题	空 try/catch/finally/switch 语句块; 未使用的局部变量、参数和 private 方法; 空 if/while 语句; 过于复杂的表达式,如不必要的 if 语句等; 复杂类
CheckStyle (检查源文件, 主要关注格式)	检查 Java 源文件是否与代码规范相符	Javadoc 注释; 命名规范; 多余没用的 Imports; Size 度量,如过长的方法; 缺少必要的空格; 重复代码

5.3.4 SonarQube 构建自动的代码扫描

SonarQube(http://www.sonarqube.org)是一款基于 Web 的静态代码质量管理平台，支持 Java、C/C++、Python、JavaScript 等 27 种语言。SonarQube 通过可配置的代码规则，从代码的可靠性、安全性、可维护性、重复率、单元测试覆盖率 5 个方面分析项目的代码质量，将风险等级从 A 到 E 划分为 5 个等级。该系统支持的功能包括以下几个方面。

(1) 检测文件、类、方法中复杂度分布；

(2) 检测源代码中重复的代码；

(3) 检测代码中注释不足或过多的问题；

(4) 检测糟糕的设计，找出循环、包与包以及类与类之间的相互依赖关系，检测代码耦合问题；

(5) 检测单元测试的充分性，通过和 Jacoco 的集成可以方便地统计并展示单元测试覆盖率。

SonarQube 由以下 4 部分组成。

(1) SonarQube Server，提供 Web UI 供用户进行管理配置 SonarQube 实例，安装管理多种 SonarQube 插件，并查看代码分析结果。SonarQube 的管理界面非常友好，可以实现插件安装、质量与阈值管理、规则配置、结果展示等功能。图 5-4 展示了一个项目代码质量检测后的结果。另外，通过安装中文语言包 Chinese Pack，SonarQube 可以提供中文的 UI 界面。

(2) SonarQube 数据库，存储 SonarQube 实例的配置和代码分析结果等数据。SonarQube 自带 H2 数据库，但作为服务器在实际项目中使用，最好配置更强大稳定的数据库，目前支持 MS SQL Server、Oracle、PostgreSQL 三种数据库类型。

(3) SonarScanner，可在多台机器上安装 SonarScanner 并连接到 SonarQube Server 对本地项目进行静态代码分析，分析结果会展示到统一的 UI 界面。

(4) SonarLint，作为插件嵌入 Eclipse、IntelliJ IDEA、Visual Studio 等开发环境中使用，可以实时检测代码，给开发人员快速的反馈，排除代码缺陷。以 IntelliJ IDEA 为例，SonarLint 在代码下方显示检测结果，针对发现的问题，在右侧给出解释以及处理意见。

SonarQube 便于用户从整体上把握项目的代码质量，且无须耗费大量的人力和时间。区别于一般的代码静态测试工具，SonarQube 是一个代码质量的管理平台，可以集成不同的测试工具、代码分析工具，随时对整个项目进行代码质量分析，生成可视化的分析报告，也可以嵌入开发环境中，实时检测开发人员编写的代码。

另外，SonarQube 还提供各类构建工具的 SonarScanner 插件。例如，在 Jenkins 中可以安装此插件，在代码构建时自动进行代码分析并上传结果到 SonarQube Server 进行处理并展示。

下面以 Gradle 项目为例，介绍如何集成 SonarQube 并自动运行代码分析。首先需要在一个 Gradle 项目中的 build.gradle 文件中添加插件信息。

```
plugins {
  id "org.sonarqube" version "3.0"
 }

 apply plugin: 'org.sonarqube'
```

其次,需要在~/.gradle/gradle.properties 文件中配置 SonarQube Server 的认证信息。

```
systemProp.sonar.host.url = http://x.x.x.x:9000
systemProp.sonar.login = < username >
systemProp.sonar.password = < password >
```

接着,运行分析命令 gradle sonarqube。

最后,运行完毕,在 SonarQube 管理界面上浏览代码分析结果,如图 5-4 所示。

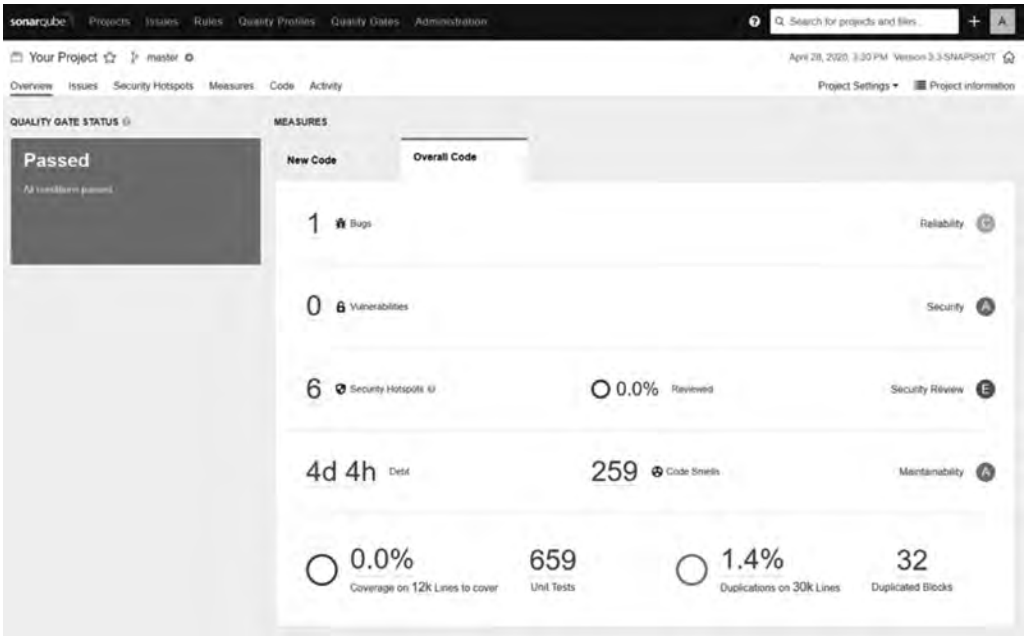

图 5-4 SonarQube 代码质量检测结果界面

5.4 单元测试的目标和任务

软件系统是由许多单元构成的,这些单元可能是一个对象或是一个类,可能是一个函数,也可能是一个更大的单元——组件或模块。要保证软件系统的质量,首先就要保证构成系统的单元的质量,也就是要开展单元测试活动。通过充分的单元测试,发现并修正单元中的问题,从而为系统的质量打下基础。单元测试属于代码级的动态测试。代码级的测试除了测试其功能性外,还需确保代码在结构上可靠、健全并且能够有良好的响应。因此只进行静态测试是不够的,必须要运行单元,进行动态测试,需要设计更充分的测试用例以验证业务逻辑合理性和单元的实际表现行为。

5.4.1 为何要进行单元测试

软件测试的目的之一就是尽可能早地发现软件中存在的错误,从而降低软件质量成本,测试越早进行越好,单元测试就显得更重要,也是系统的功能测试的基础。在实践中,单元测试的大部分工作由开发人员完成,开发人员更多的兴趣在编程上,而不愿在测试上花比较多的时间,对测试自己的代码总会存在心理障碍。一旦编码完成,开发人员总是迫切希望交给测试人

员，让测试人员去执行测试。如果没有执行好单元测试，软件在集成阶段及后续的测试阶段会发现更多的、各种各样的错误，甚至软件根本不能运行。大量的时间将被花费在跟踪那些包含在独立单元内的、简单的错误上面，所以表面上的进度取代不了实际进度，对于整个项目或系统反而会增加额外的工期，导致软件成本的提高。软件中存在的错误发现得越早，则修改和维护的费用就越低，而且难度越小，所以单元测试是早期抓住这些错误的最好时机。

另一方面，总有一些自认为很棒的程序员，对自己的程序充满了信心，对单元测试很漠然，认为代码没有什么小问题，而只会出现一些集成上的大问题，这些问题要依赖测试人员来发现。但是规模越大的系统，其系统集成的复杂性就越高。现在大多数软件系统的规模都很大，想完成各个单元之间的接口进行全面的测试，几乎不可能。其结果是测试将无法达到它所应该有的全面性，较多的缺陷将被遗漏。即使在后期测试中再被发现，也会造成严重的影响，代码的修改量会很大。所以在单元测试中实际也包含接口测试，相当于集成测试的一部分工作。从目前实践来看，软件单元测试和软件集成测试难以分离，往往是同时进行的，所以把单元测试和集成测试放在一章内进行讨论。

5.4.2 单元测试的目标和要求

单元测试是对软件基本组成单元进行的测试，而且软件单元是在与程序的其他部分相隔离的情况下进行独立的测试。单元测试的对象可以是软件设计的最小单位——一个具体函数或一个类的方法，也可以是一个功能模块、组件。一般情况下，被测试的单元能够实现一个特定的功能，具有一定的独立性，同时又通过明确的接口定义与其他单元联系起来。调试与单元测试在工作中常交织在一起，操作上有一定的相似性，但两者的目的完全不同。测试是为了找出代码中存在的缺陷，通过某种测试覆盖要求，检查代码或运行代码以验证是否符合规范、符合设计要求等；而调试是为了修正已发现的缺陷，即针对已发现的缺陷来寻找引起缺陷的原因，如通过设置断点跟踪程序来检查变量状态，以判断是不是某个变量取值不对而导致问题的出现。

检验各单元模块是否被正确地编码，即验证代码和软件系统设计的一致性是单元测试的主要目标，但是单元测试的目标不仅测试代码的功能性，还需确保代码在结构上可靠且健壮，能够在各种条件下（包括异常条件，如异常操作和异常数据）给予正确的响应。如果这些系统中的代码未被适当测试，则其弱点可被用于侵入代码，并导致安全性风险（如内存泄漏或被窃指针）以及性能问题。执行完全的单元测试可以比较彻底地消除各个单元中所存在的问题，避免将来功能测试和系统测试问题查找的困难，从而减少应用级别所需的测试工作量，并且彻底减少发生误差的可能性。概括起来，单元测试是对单元的代码规范性、正确性、安全性、性能等进行验证，通过单元测试，需要验证下列这些内容。

（1）数据或信息能否正确地流入和流出单元。

（2）在单元工作过程中，其内部数据能否保持其完整性，包括内部数据的形式、内容及相互关系不发生错误，也包括全局变量在单元中的处理和影响。

（3）在数据处理的边界处能否正确工作。

（4）单元的运行能否做到满足特定的逻辑覆盖。

（5）单元中发生了错误，其中的出错处理措施是否有效。

（6）指针是否被错误引用、资源是否及时被释放。

（7）有没有安全隐患？是否使用了不恰当的字符串处理函数等。

单元测试的主要依据是《软件需求规格说明书》《软件详细设计说明书》,同时要参考并符合软件的整体测试计划和集成方案。单元测试的一系列活动包括以下几方面。

(1) 建立单元测试环境,包括在开发集成环境(Integrated Development Environment,IDE)中安装和设置单元测试工具(插件);

(2) 测试脚本(测试代码)的开发和调试;

(3) 测试执行及其结果分析。

在单元测试活动中强调被测试对象的独立性,软件的独立单元将与程序的其他部分被隔离开,以避免其他单元对该单元的影响。这样,缩小了问题分析范围。在单元测试中,需要关注以下主要内容。

(1) 目标:确保模块被正确地编码;

(2) 依据:详细设计描述;

(3) 过程:经过设计、脚本开发、执行、调试和分析结果等环节;

(4) 执行者:由程序开发人员和测试人员共同完成;

(5) 采用哪些测试方法:包括代码控制流和数据流分析方法,并结合参数输入域的测试方法;

(6) 测试脚本的管理:可以按照产品代码管理的方法进行类似的配置管理(并入代码库),包括代码评审、版本分支、变更控制等;

(7) 如何进行评估:通过代码覆盖率分析工具来分析测试的代码覆盖率、分支或条件的覆盖率。

何时可以结束单元测试?测试是否充分足够?如何评估测试的结果?每个项目都有自己的特殊需求,但通常除了代码的标准和规范,单元测试中主要考虑的是对结构和数据测试覆盖率。下面给出是否通过单元测试的一般准则。

(1) 软件单元功能与设计需求一致。

(2) 软件单元接口与设计需求一致。

(3) 能够正确处理输入和运行中的错误。

(4) 在单元测试中发现的错误已经得到修改并且通过了测试。

(5) 达到了相关的覆盖率的要求。

(6) 完成软件单元测试报告。

5.4.3 单元测试的任务

为了实现上述目标,单元测试的主要任务包括对单元功能、逻辑控制、数据和安全性等各方面进行必要的测试。具体地说,包括单元中所有独立执行路径、数据结构、接口、边界条件、容错性等测试。

1. 单元独立执行路径的测试

在单元中应对每一条独立执行路径进行测试,这不仅检验单元中每条语句(代码行)至少能够正确执行,主要检查下列问题。

(1) 误解或用错了算符优先级;

(2) 混合类型运算;

(3) 变量初始化错误、赋值错误;

(4) 错误计算或精度不够;

(5) 表达式符号错等。

而且要检验所涉及的逻辑判断、逻辑运算是否正确,如是否存在不正确的比较和不适当的控制流造成的错误。此时判定覆盖、条件覆盖和基本路径覆盖等方法是最常用且最有效的测试技术。比较判断与控制流常常紧密相关,这方面常见的错误主要有以下几种。

(1) 不同数据类型的对象之间进行比较;

(2) 错误地使用逻辑运算符或优先级;

(3) 因变量取值的局限性,期望理论上相等而实际上不相等的两个变量的比较;

(4) 比较运算或变量出错;

(5) 循环终止条件错误,或形成死循环;

(6) 错误地修改了循环变量。

2. 单元局部数据结构的测试

检查局部数据结构是检查临时存储的数据在程序执行过程中是否正确、完整。局部数据结构往往是错误的根源,应仔细设计测试用例,力求发现下面几类错误。

(1) 不合适或不相容的类型说明;

(2) 变量无初值;

(3) 变量初始化或默认值有错;

(4) 不正确的变量名(拼错或不正确地截断);

(5) 出现上溢、下溢和地址异常。

3. 单元接口测试

只有在数据能正确输入(如函数参数调用)、输出(如函数返回值)的前提下,其他测试才有意义。对单元接口的检验,不仅是集成测试的重点,也是单元测试的不可忽视的部分。单元接口测试应该考虑下列主要因素。

(1) 输入的实际参数与形式参数的个数、类型等是否匹配、一致;

(2) 调用其他单元时所给实际参数与被调单元的形式参数个数、属性和量纲是否匹配;

(3) 调用预定义函数时所用参数的个数、属性和次序是否正确;

(4) 是否存在与当前入口点无关的参数引用;

(5) 是否修改了只读型参数;

(6) 对全程变量的定义各单元是否一致;

(7) 是否把某些约束作为参数传递。

如果单元内包括外部输入输出(如打开某文件、读入文件数据、向数据库写入等),还应该考虑下列因素。

(1) 文件属性是否正确;

(2) OPEN/CLOSE 语句是否正确;

(3) 格式说明与输入输出语句是否匹配;

(4) 缓冲区大小与记录长度是否匹配;

(5) 文件使用前是否已经打开;

(6) 是否处理了文件尾;

(7) 是否对异常的输入输出进行判断;

(8) 输出信息中是否有格式错误。

4. 单元边界条件的测试

众所周知,程序容易在边界上失效,采用边界值分析技术,针对边界值及其最靠近的左右两个值设计测试用例,很有可能发现新的错误。如果在单元测试中忽略边界条件的测试,在系统级测试中很难被发现,即使被发现后对其跟踪、寻其根源也是一件不容易的事。

5. 单元容错性测试

在软件构造中强调防御式编程,即要求在编写程序时能预见各种可能的出错条件,并针对这些出错进行正确处理,如给予出错提示或设置统一的出错处理函数。针对单元错误处理机制(容错性),着重检查下列问题。

(1) 输出的出错信息难以理解;

(2) 记录的错误与实际遇到的错误不相符;

(3) 在程序自定义的出错处理代码运行之前,系统已介入;

(4) 异常处理不当;

(5) 错误陈述中未能提供足够的定位出错信息。

6. 内存分析

内存泄漏会导致系统运行的崩溃,尤其对于嵌入式系统这种资源比较匮乏、应用非常广泛,而且往往又处于重要部位的,将可能导致无法预料的重大损失。通过检查内存使用情况,可以了解程序内存分配的真实情况,发现对内存的不正常使用,在问题出现前发现征兆,在系统崩溃前发现内存泄露错误;发现内存分配错误,并精确显示发生错误时的上下文情况,指出发生错误的缘由。

5.4.4 驱动程序和桩程序

运行被测试单元,为了隔离单元,根据被测试单元的接口,开发相应的驱动程序(Driver)和桩程序(Stub),如图 5-5 所示。

图 5-5 单元测试中驱动程序和桩程序

(1) 驱动程序(Driver),也称驱动模块,用以模拟被测模块的上级模块,能够调用被测模块。在测试过程中,驱动模块接收测试数据,调用被测模块并把相关的数据传送给被测模块。

(2) 桩程序(Stub),也称桩模块,用以模拟被测模块工作过程中所调用的下层模块。桩模块由被测模块调用,它们一般只进行很少的数据处理,如打印入口和返回,以便于检验被测模块与其下级模块的接口。

通过驱动程序和桩程序就可以隔离被测单元,而又能使测试继续下去。驱动程序作为入口,可以设置不同的数据参数,来完成各种测试用例。

具有驱动程序和桩程序作用的小程序示例

公司正在进行一项大型的网络服务系统的开发,项目组承担的是服务器端的软件开发。其中有个项目负责多台数据库服务器的数据复制。服务系统是实时的,对数据复制的性能要求很高。当开发人员完成了数据传输模块时(还未编制和数据库相关的模块),就主动要求对其性能进行单元测试。

进行这样的性能测试，不需要详细了解该单元的结构，但首先要掌握设计文档中相关的性能指标和运行的网络环境及服务器环境等指标，以便搭建相应的测试环境。

其次，要求开发人员提供相应的程序接口，测试人员根据接口定义来设计驱动程序和桩程序用来运行并测试该单元程序。为此，编写了一个功能简单的小程序，既作为驱动程序也是桩程序。该驱动程序在服务器端运行模拟数据库提供和接收需复制的数据，它能够随机产生可设置大小的数据包，按设置好的单位时间发包数量进行数据包的发送，同时它也是接收端，能对接收到的数据包的数量和大小进行简单的统计，以便实现简单的验证，如图 5-6 所示。

图 5-6 具有桩模块作用的驱动程序

接着要设计测试用例并实施测试。设计测试用例时：

(1) 根据指标考虑数据包的大小和频率，如大包低频或小包高频；

(2) 考虑两个驱动程序的数据对发；

(3) 从两个驱动程序变为多个驱动程序的数据对发；

(4) 从同一网段变为多个网段，验证代理服务器或网关造成的影响。

发现问题后，要先排除网络等环境因素，再报告开发人员进行调试。

这样会确保了该单元将不会是该项目的性能瓶颈，也避免了后续开发的盲目性。很多参考书中误导人们认为单元测试采用的是白盒测试技术，由开发人员完成。这很片面，在有些情况下是完全不对的。从另一方面来说，该案例的测试工作也可由开发者完成，但在开发的初期，测试人员并没有大的测试压力，而开发者面临着大量代码编写压力，浪费开发者的时间直接影响项目的进度，何况开发者与测试者的心理状态的不同，还可能直接影响测试结果的可靠性。

5.4.5 类测试

面向对象的单元测试通常是对一个基类或其子类进行测试，因为类是面向对象软件的基本单位。对于类的单元测试可以看作是对类的成员函数进行测试。一般不会对类的每个成员及方法进行测试，例如，一般不会针对成员变量的定义进行单元测试，一般也不需要对 get/set 方法进行单独测试，但对于核心或重要的方法需要进行全面的单元测试。对单个方法的测试类似于对传统软件的单个函数的测试，第 3 章所介绍的测试方法(如基于输入域的、基于逻辑覆盖的等测试方法)都可以应用在这里。例如，可以根据前置条件的输入条件(包括常见值和边界值)来设计单元测试用例，以检验输出结果的正确性和后置条件是否得到满足。

类测试要验证类的实现是否和该类的说明完全一致。如果类的实现正确，那么类的每个实例的行为也应该是正确的。下面通过一个具体的 Tester 类来说明类的测试。在具体的 Tester 类中，为每个测试用例定义了一个方法，被称为测试用例方法。测试用例方法的任务是为某个用例构建输入状态，生成事件序列并检查输出状态来执行测试用例。例如，通过将一个输出和作为参数传递的对象实例化，然后生成测试用例指定的事件。这些方法还为测试计

划提供了可跟踪性——每个测试用例或每一组紧密联系的测试用例都有一个方法。图 5-7 显示了一个满足这些需求的 Tester 类的模型。

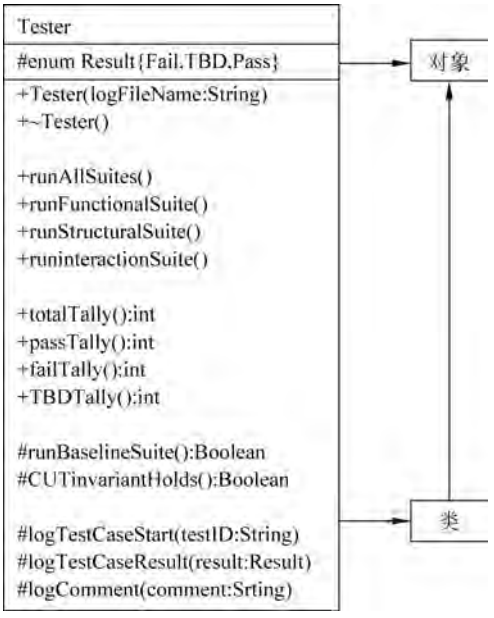

图 5-7 Tester 类需求的一个类模型

由于继承与多态的使用,对于类的测试通常不能限定在子类中定义的成员变量和成员方法上,还需要考虑父类对子类的影响。

一般而言,子类对父类中的多态方法的覆盖应该保持父类对该方法的定义说明。多态服务测试就是为了测试子类中多态方法的实现是否保持了父类对该方法的要求。假设已存在父类的一个测试用例集,在对子类的测试时,可以选取其中涉及相关多态方法的测试用例,并把子类的实例当作父类的实例来执行这些测试用例。

某个方法在主类中已有定义,但由于某种要求,需要重载父类中的方法,子类中这个重载的方法已做了定义。后来,由于要加入一个新功能,中间也要用到此方法,但新功能的开发人员发现父类中已有此方法,可能对子类中的重载方法为什么会使用的情景不了解或根本不知道,容易导致出错。

类似地,多态地引入同一个方法名,因为接口参数的不同,中间的操作结果与最终的返回结果也就不同。如果选择错了同名方法,那么无疑实际结果与最终要求是不一致的。这就要求代码中要有足够的、清晰的注释,特别是供大家调用的公用方法,建议对于各参数的要求、返回的结果、需不需要生成新的 Cookie 或 Session 等进行严格定义,并在方法有修改时,注释也应做相应的修改,这样可以大大减少错误出现的概率。

在最复杂的情况下,对于子类的测试可能只能采用展平测试的策略。所谓展平测试是指将子类自身定义的成员方法与成员变量以及从父类继承来的成员方法与成员变量全部放在一起组成一个新类(如果成员方法间存在覆盖关系,还需要确定哪些成员方法是子类真正拥有的),并对其进行测试。需要指出的是:展平后的类可能很大,测试的代价也较高,此时要尽可能地减少不必要的代价。

5.5 分层单元测试

目前应用程序都是分层构造的，如数据访问层、业务逻辑层、表示层等，那么在单元测试时也要分层进行。下面分别讨论如何对 Action 层、BIZ 业务逻辑层、Servlet 层等不同层次进行测试，从而完成对核心功能、数据库存取、页面跳转等功能的验证。

5.5.1 Action 层的单元测试

Action 层主要用于接收页面传来的参数，然后调用业务逻辑层的封装方法，最后负责跳转到相应的页面。所以对 Action 层的测试主要是对跳转的验证，也就是在相同的情况下，能不能跳到指定的页面。

对依赖于其他外部系统(如数据库或 EJB 等)的代码进行单元测试，这是一件很困难的工作。但在这种情况下，进行单元测试能有效地隔离测试对象和外部依赖，以便管理测试对象的状态和行为。使用 Mock 对象是隔离外部依赖的一个有效方法。

1. 什么是 Mock

简单地说，Mock 就是模拟，模拟测试时所需的对象及测试数据。比如，Struts 中的 Action 类的运行必须依靠服务器的支持，只有服务器可以提供 HttpServletRequest 对象。如果不启动服务器，那么就无法对 Action 类进行单元测试。即使当业务逻辑被限定在业务层，Struts Action 通常还会包含重要的数据验证、数据转换和数据流控制代码。依靠启动服务器运行程序来测试 Action 过于麻烦。如果让 Action 脱离容器，那么测试就变得极为简单。脱离了容器，Request 与 Response 对象如何获得？这时可以使用 Mock 来模拟 Request 与 Response 对象。

2. StrutsTestCase

StrutsTestCase 是 Junit TestCase 类的扩展，提供基于 Struts 框架的代码测试。可以通过设置请求参数，检查在 Action 被调用后的输出请求或 Session 状态这种方式完成 Struts Action 的测试。StrutsTestCase 提供了用框架模拟 Web 容器的模拟测试方法，也提供了真实的 Web 容器(如 Tomcat)下的测试方法。所有的 StrutsTestCase 单元测试类都源于模拟测试 MockStrutsTestCase 或容器内测试的 CactusStrutsTestCase。StrutsTestCase 不仅可以测试 Action 对象的实现，而且可以测试 mapping、frombeans 和 forwards 声明。

(1) MockStrutsTestCase：是对 JUnitTestCase 基类的扩展，从而实现对 Struts Action 对象的模拟测试。它借助 Mock 对象方法来模拟 Servlet 容器，为 ActionForm 子类提供相应方法设置请求路径、请求参数并能够验证 ActionForward 正确性和 ActionError 信息。

(2) CactusStrutsTestCase：是对 CactusServletTestCase 基类的扩展，从而实现对 Struts Action 对象的模拟测试。而 cactus 是容器内 Servlet、EJB 等组件的面向服务器端的测试框架(见 http://jakarta.apache.org/cactus)。

3. 更多 StrutsTestCase 资源与参考

(1) 通过 http://sourceforge.net/project/showfiles.php?group_id=39190 来下载它的最新版本。

(2) JavaDoc：http://strutstestcase.sourceforge.net/api/index.html。

(3) 常见问题：http://strutstestcase.sourceforge.net/faq.htm。

4. 用 MockStrutsTestCase 测试举例

模拟用户登录的例子，使用 MockStrutsTestCase 测试，因为它需要更少的启动和更快的运行。

```
public classLoginAction extends Action {
public ActionForward perform(ActionMapping mapping,
ActionForm form,
HttpServletRequest request,
HttpServletResponse response) {
   String username = ((LoginForm) form).getUsername();
   String password = ((LoginForm) form).getPassword();
ActionErrors errors = new ActionErrors();
   //如果用户名密码不是 SchAdmin/BJ@2008,则返回 Login 页面,并显示错误的信息
if ((!"SchAdmin".equals(username)) || (!"BJ@2008".equals(password)))
errors.add("password",new ActionError("error.password.mismatch"));

if (!errors.empty()) {
saveErrors(request,errors);
return mapping.findForward("login");
   }
   //用户名与密码正确,保存认证的信息到 Session 中,并跳转到成功页面
HttpSession session = request.getSession();
session.setAttribute("authentication", username);
return mapping.findForward("success");
}
```

编写成功的测试案例(用户名与密码都正确，测试其跳转)：

```
public class TestLoginAction extends MockStrutsTestCase {

public TestLoginAction(String testName) {
super(testName);
}
public void testSuccessfulLogin() {
setConfigFile("mymodule","/WEB-INF/struts-config-mymodule.xml");
setRequestPathInfo("/mymodule","/login.do");
addRequestParameter("username","SchAdmin");
addRequestParameter("password","BJ@2008");
actionPerform();
verifyForward("success");
assertEquals("SchAdmin",(String) getSession().getAttribute("authentication"));
verifyNoActionErrors();
  }
}
```

编写出错的测试案例(用户名正确但密码不正确，测试其跳转与出错信息)：

```
public void testFailedLogin() {
addRequestParameter("username","SchAdmin");
addRequestParameter("password","111111");
setRequestPathInfo("/login");
```

```
actionPerform();
verifyForward("login");
verifyActionErrors(new String[] {"error.password.mismatch"});
assertNull((String) getSession().getAttribute("authentication"));
}
```

5.5.2 数据访问层的单元测试

业务逻辑层，一般用于处理比较复杂的逻辑，也用于 DAO 层的数据操作。对于简单的业务逻辑，可以用 JUnit 测试，而对复杂的逻辑，可以用 Mock 对象来模拟测试。如果测试对象依赖于 DAO 的代码，可以采用 Mock Object 方法。但是，如果测试对象变成了 DAO 本身，又如何进行单元测试呢？开源的 DbUnit 项目就是为了解决这一问题。

DbUnit(http://dbunit.sourceforge.net/)是为数据库驱动的项目而对 JUnit 的扩展，可以控制测试数据库的状态。在 DAO 单元测试之前，DbUnit 为数据库准备好初始化数据；而在测试结束时，DbUnit 会把数据库状态恢复到测试前的状态。DbUnit 的主要功能是为数据库测试提供稳定且一致的数据。DbUnit 通过预先在 XML 文件设置数据值、使用 SQL 查询另外的表格为测试提供数据等方式来达到这个目的，而通常只需要使用 XML 文件预置数据的方法即可。

DbUnit 支持多种方式向数据库中插入数据，如 FlatXmlDataSet、DTDDataSet 等，而最常用的是 FlatXmlDataSet。顾名思义，这种方式就是用 XML 的方式准备数据，DbUnit 载入 XML 文件并完成插入数据库的操作。

首先需要准备一份 XML 的数据文件，格式如下(数据文件 dataset.xml)：

```
1 <?xmlversion = "1.0" encoding = "GB2312"?>
2 < dataset >
3 < TABLEid = "001" name = "mike" />
4 < TABLEid = "002" name = "jack" />
5 </dataset >
```

其中，第 2 行 dataset 标签是 XML 的根节点，对应于 DbUnit 中的一个 FlatXmlDataSet 对象。第 3 行表示要插入的一条记录，表名为 TABLE，插入的字段为 id、name，对应的值分别为“001”“mike”。整个 XML 文件一共插入两条记录。注意：XML 文件中的值必须用双引号，DbUnit 会根据实际的表结构进行类型转换。DbUnit 无法插入空值，只能跳过该字段。

接下来要做的就是载入这份数据文件(载入 XML 文件中的数据)。

```
1    public IDataSet getDataSet(String path) {
2    FlatXmlDataSet dataSet = null;
3    try {
4      dataSet = new FlatXmlDataSet(new FileInputStream(new File (path)));
5    } catch (Exception e) {
6      e.printStackTrace();
7    }
8    return dataSet;
9  }
```

其中：

(1) 第 2 行，声明一个 FlatXmlDataSet 对象用来装载测试数据。

(2) 第 4 行，读取 path 指定的文件来初始化 dataSet 对象。

(3) 第 8 行，返回载有数据的对象。

最后就是连接数据库，对数据库进行读写操作。

代 码 示 例

1. 连接数据库代码

```
    public DbUnit(String driver, String url, String user, String password) {
try {
          Class driverClass = Class.forName(driver);
          jdbcConnection = DriverManager.getConnection(url, user, password);
       } catch (Exception e) {
e.printStackTrace();
       }
    }
```

2. 添加数据库记录代码

```
public void insertData(IDataSet dataSet) {
try {
             //DatabaseOperation.DELETE.execute(connection, dataSet);
DatabaseOperation.INSERT.execute(connection, dataSet);
         } catch (Exception e) {
e.printStackTrace();
         }
}
```

说明：一般添加之前先删除数据库原有的数据，再把 XML 文件中的数据保存进去，以达到每次测试数据都相同的目的。本例中注释的一行删除就是为了这个目的。

3. 删除数据库记录代码

```
public void deleteData(IDataSet dataSet) {
try {
DatabaseOperation.DELETE.execute(connection, dataSet);
         } catch (Exception e) {
e.printStackTrace();
         }
    }
```

4. 修改数据库代码

```
public void updateData(IDataSet dataSet) {
try {
DatabaseOperation.UPDATE.execute(connection, dataSet);
         } catch (Exception e) {
e.printStackTrace();
         }
    }
```

5.5.3 Servlet 的单元测试

在开发复杂的 Servlet 时，需要对 Servlet 本身的代码块进行测试，可以选择 HttpUnit (http://httpunit.sourceforge.net/)，它提供了一个模拟的 Servlet 容器，让 Servlet 代码不需要发布到 Servlet 容器(如 Tomcat)就可以直接测试。

使用 HttpUnit 测试 Servlet 时，需要创建一个 ServletRunner 的实例，负责模拟 Servlet 容器环境。如果只是测试一个 Servlet，可直接使用 registerServlet 方法注册这个 Servlet。如果需要配置多个 Servlet，可以编写自己的 web.xml，然后在初始化 ServletRunner 的时候将它的位置作为参数传给 ServletRunner 的构造器。

在测试 Servlet 时，应该记得使用 ServletUnitClient 类作为客户端，它继承自 WebClient。要注意的差别是，在使用 ServletUnitClient 时，它会忽略 URL 中的主机地址信息，并直接指向它的 ServletRunner 实现的模拟环境。

通过对 HelloWorld 代码的测试展示 HttpUnit 来测试 Servlet 的方法。

```
import java.io.IOException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloWorld extends HttpServlet {
    public void saveToSession(HttpServletRequest request) {
        request.getSession().setAttribute("testAttribute",
            request.getParameter("testparam"));
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        String username = request.getParameter("username");
        response.getWriter().write(username + ":Hello World!");
    }
    public boolean authenticate() {
        return true;
    }
}
```

HttpUnit 测试代码示例

```
import com.meterware.httpunit.GetMethodWebRequest;
import com.meterware.httpunit.WebRequest;
import com.meterware.httpunit.WebResponse;
import com.meterware.servletunit.InvocationContext;
import com.meterware.servletunit.ServletRunner;
import com.meterware.servletunit.ServletUnitClient;
import junit.framework.Assert;
import junit.framework.TestCase;

public class HttpUnitTestHelloWorld extends TestCase {

protected void setUp() throws Exception {
super.setUp();
  }

protected void tearDown() throws Exception {
super.tearDown();
  }
```

```
 public void testHelloWorld() {

try {
//创建 Servlet 的运行环境
ServletRunner sr = new ServletRunner();
    // 向环境中注册 Servlet
sr.registerServlet("HelloWorld", HelloWorld.class.getName());
// 创建访问 Servlet 的客户端
ServletUnitClient sc = sr.newClient();
    // 发送请求
WebRequest request = new GetMethodWebRequest("http://localhost/HelloWorld");
request.setParameter("username", "testuser");
InvocationContext ic = sc.newInvocation(request);
    HelloWorld is = (HelloWorld) ic.getServlet();
// 测试 servlet 的某个方法
Assert.assertTrue(is.authenticate());
    // 获得模拟服务器的信息
WebResponse response = sc.getResponse(request);
    // 断言
Assert.assertTrue(response.getText().equals("testuser:Hello World!"));
  } catch (Exception e) {
e.printStackTrace();
  }
 }
}
```

5.6 单元测试工具

单元测试一般针对程序代码进行测试,这决定了其测试工具和特定的编程语言密切相关,所以单元测试工具基本是相对不同的编程语言而存在,多数集成开发环境(如 IntelliJ IDEA、Microsoft Visual Studio、Eclipse)会提供单元测试工具,甚至提供测试驱动开发方法所需要的环境。最典型的就是 xUnit 工具家族。

(1) JUnit 是针对 Java 的单元测试工具。

(2) CppUnit 是 C++单元测试工具。

(3) NUnit 是 C# (.Net)单元测试工具。

(4) HtmlUnit、JsUnit、PhpUnit、PerlUnit、XmlUnit 则分别是针对 HTML、Javascript、PHP、Perl、XML 的单元测试工具(框架)。

除了上述典型的 xUnit 单元测试框架之外,还有 GoogleTest 单元测试框架(http://code.google.com/p/googletest/),它是基于 xUnit 架构的测试框架,在不同平台上(Linux、Mac OS X、Windows、Cygwin、Windows CE 和 Symbian)为编写 C++测试而生成的,支持自动发现测试、丰富的断言集、用户定义的断言、death 测试、致命与非致命的失败、类型参数化测试、各类运行测试的选项和 XML 的测试报告等。

5.6.1 JUnit 介绍

JUnit 是一个开放源代码的 Java 测试框架，用在编写和运行可重复的测试脚本之上。它是单元测试框架体系 xUnit 的一个实例。JUnit 框架功能强大，目前已成为 Java 单元测试框架的业界标准。JUnit 的主要特性如下。

(1) 可以使测试代码与产品代码分开，这更有利于代码的打包发布和测试代码的管理。

(2) 针对某一个类的测试代码，以较少的改动便可以应用另一个类的测试，JUnit 提供了一个编写测试类的框架，使测试代码的编写更加方便。

(3) 易于集成到程序中的构建过程中，JUnit 和 Gradle、Maven、Ant 的结合还可以实施增量开发。

(4) JUnit 的源代码是公开的，故而可以进行二次开发。

(5) JUnit 具有很强的扩展性，可以方便地对 JUnit 进行扩展。

JUnit 的最新版本是 JUnit 5，支持 Java 8 及以上版本，并且由 JUnit platform、JUnit Jupiter、JUnit Vintage 等组成，如图 5-8 所示。

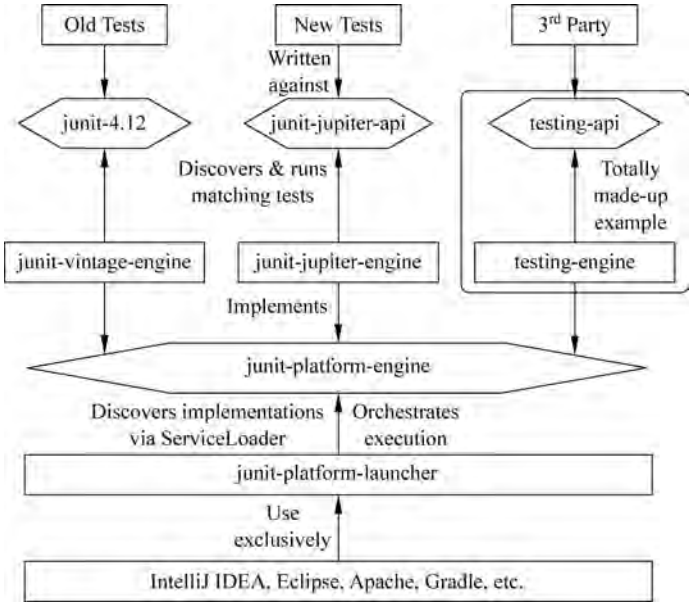

图 5-8 JUnit 5 架构示意图

(1) JUnit platform，其主要作用是在 JVM 上启动测试框架，包含一个内部的 JUnit 公共库以及用于测试引擎、配置和启动测试计划、配置测试套件的注释等公共 API，同时还支持通过控制台(Console Launcher)命令、IDE 、构建工具 Gradle 或 Maven(即借助 surefire-provider、gradle-plugin)等来启动测试。

(2) JUnit Jupiter，包含了 JUnit 5 最新的编程模型(注释、类、方法)、扩展机制的组合(Jupiter API)和一个测试引擎(Test Engine)，用于编写和执行 JUnit 5 的新测试，其中 junit-jupiter-params 为参数化测试提供支持。

(3) JUnit Vintage，一个测试引擎，允许在平台上运行老的 JUnit 3 和 JUnit 4 测试用例，从而确保必要的向下兼容性。

JUnit 提供了丰富的 Assert(断言)语句,用来对测试执行结果进行验证。在 JUnit 4 中,Assert 类提供了下列断言方法。

(1) assertEquals(),查看对象中存的值是否与期望的值相等,与字符串比较方法 equals()类似。

(2) assertFalse(),查看变量是否为 false 或 true,如果 assertFalse()查看的变量的值是 false,则测试成功,如果是 true 则失败,assertTrue()与之相反。

(3) assertSame()和 assertNotSame(),比较两个对象的引用是否相等和不相等,类似于通过"==="和"!="比较两个对象。

(4) assertNull()和 assertNotNull(),查看对象是否为空和不为空。

(5) fail(),意为失败,执行它会直接抛出错误。

(6) assertThat(),JUnit4.4 引入了 Hamcrest 框架,能提供一套匹配符 Matcher 使得 assertThat 断言的使用更接近自然语言。assertThat 可以用来代替 Assert 类中的各种方法,如 assertEquals、assertFalse 等。

JUnit 5 中的所有断言是 org.junit.jupiter.api.Assertions 类中的静态方法,保留了 JUnit 4 的许多断言方法,同时添加了以下新的断言方法。

(1) assertArrayEquals(),用来判断两个对象或原始类型的数组是否相等。

(2) assertAll(),用来创建分组断言,执行其中所有断言并一起报告失败。如果不采用 assertAll 断言方法,测试将会在第一次断言失败时就停止。assertAll 断言方法示例如下。

```
@Test
    void groupedAssertions() {
        // In a grouped assertion all assertions are executed, and all
        // failures will be reported together.
assertAll("person",
            () -> assertEquals("Jane", person.getFirstName()),
            () -> assertEquals("Doe", person.getLastName())
        );
    }
```

JUnit 5 提供了丰富的注解,在编写测试用例的时候对方法进行注解,常用的注解如下。

(1) @Test:表示被注解的方法是一个基本的测试方法。与 JUnit 4 的@Test 注解不同的是,它没有声明任何属性。

(2) @ParameterizedTest:表示参数化测试方法。

(3) @RepeatedTest:表示重复测试的模板/方法。

(4) @TestFactory:用于动态测试的测试工厂类方法。

(5) @BeforeEach:表示被注解的方法在当前类中的每一个测试方法(被@Test、@RepeatedTest、@ParameterizedTest 或者@TestFactory 注解的方法)之前都执行一次。

(6) @AfterEach:表示被注解的方法在每一个测试方法执行之后都执行一次,一般用于释放参数、释放空间等操作。

(7) @BeforeAll:表示被注解的方法在当前测试类中所有测试方法执行之前执行,每个测试类运行时只会执行一次。

(8) @AfterAll:表示被注解的方法在当前测试类中所有测试方法执行完毕后执行,每个测试类运行时只会执行一次。

(9) @TestTemplate：表示被注解的方法是被多次调用的测试用例的模板，这依赖于已注册的提供者返回的调用上下文的数量。

(10) @TestMethodOrder：用于定义被注解的测试类内部测试方法的执行顺序，类似于JUnit 4 的@FixMethodOrder。

(11) @ParameterizedTest：用于定义参数化测试方法，使用不同的数据重复运行测试脚本。而且还可以和其他注释组合使用，指定多个数据来源，包括@ValueSource、@CsvSource、@MethodSource、@ArgumentSource 等。代码示例如下。

```
@ParameterizedTest
@NullAndEmptySource
@ValueSource(strings = { " ", " ", "\t", "\n" } )
Void nullEmptyAndBlankStrings(String text) {
        assertTrue(text == null || text.trim().isEmpty());
}
```

5.6.2 IntelliJ IDEA 中的 JUnit 应用举例

在 IntelliJ IDEA 中通常已经默认安装了 JUnit，在 Settings-> Plugins-> Installed 界面可以找到 JUnit 插件。这里以 JUnit 5 为例讲解如何进行单元测试。

1. 建立一个被 JUnit 测试的类

为了便于讲解，这里以一个简单的 StringUtil.java 的工具类为被测试的类，它就是将两个传入的字符串连接在一起。Java 中工具类(Util)的功能相对简单，一般不涉及复杂的业务逻辑，如求一个数的最大公约数、数值转换、字符串简单操作、拼 URL 等。程序代码如下：

```
package utils;

public class StringUtil {
/**
    * 功能：对传入的两个字符串进行连接
*
    * @param str1 String 第一个传入的字符串
* @param str2 String 第二个传入的字符串
* 要求：传入的两个字符串都不能为 null
    * @return String 经过连接后的字符串
*/
public String addString(String str1, String str2) {
return str1 + str2;
    }
}
```

2. 建立其对应的 JUnit Test 类

选择要进行测试的类文件，在类文件中按下组合键 Ctrl+Shift+T 弹出创建测试类的窗口，选择“Create New Test…”，然后在弹出的如图 5-9 所示的对话框中进行如下设置。

(1) Testing library：JUnit5。

(2) Class name：用于设置新建的测试类名称，一般命名规则是：测试的类名+Test，即StringUtilTest。

(3) Destination package：类文件所在的包，本例为 utils。

(4) 勾选 setUp/@Before 和 tearDown/@After。

(5) 选择要生成测试的方法：addString(str1：String，str2：String)：String。

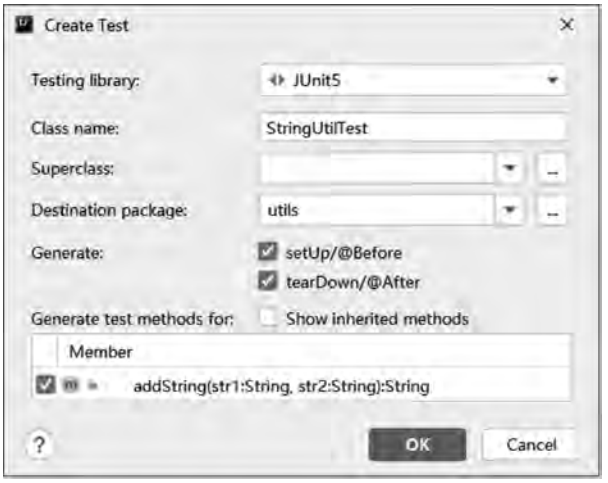

图 5-9 创建测试类的对话框

单击 OK 按钮，就会自动生成如下的代码。

```
package utils;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;

import static org.junit.jupiter.api.Assertions.*;

class StringUtilTest {

@BeforeEach
void setUp() {
    }

@AfterEach
void tearDown() {
    }
}
```

3. 针对自动生成的代码，进行补充修改，使其满足对特定功能的测试

```
package utils;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;
```

```
class StringUtilTest {

@BeforeEach
void setUp() {
    }

@AfterEach
void tearDown() {
    }

@Test
void addString() {
StringUtil a = new StringUtil();
assertEquals("aabb",a.addString("aa","bb"));
    }
}
```

4. 执行测试

单击 StringUtilTest 左边的运行按钮，开始运行单元测试，界面下方会出现测试通过的提示，如图 5-10 所示。

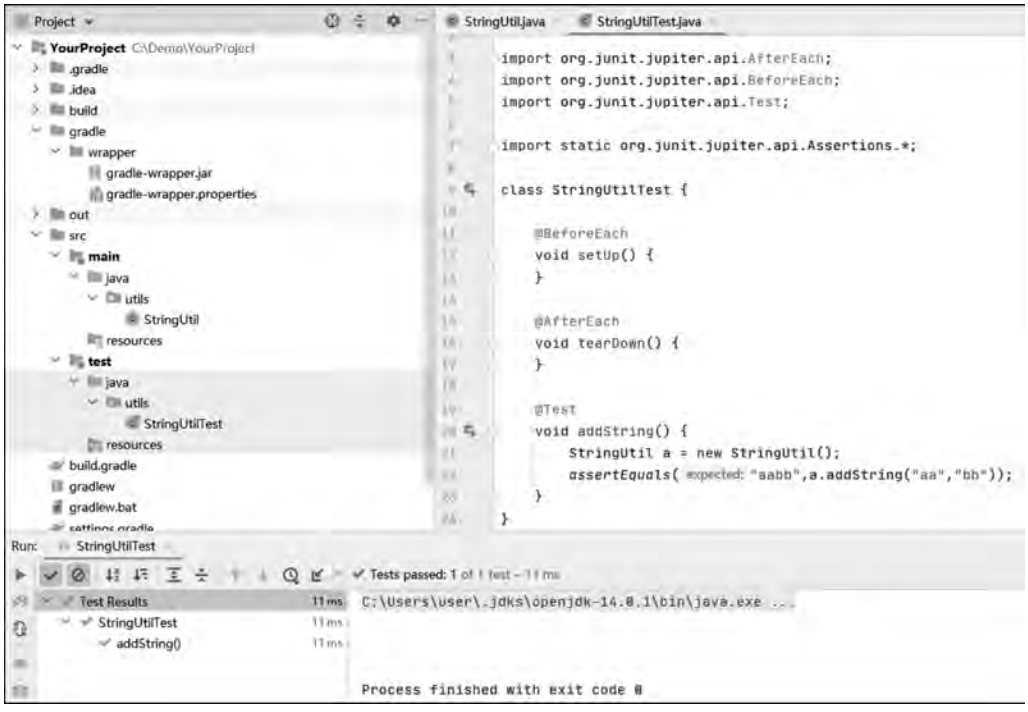

图 5-10 JUnit test case 测试正确示例

如果失败会出现错误的原因和数目。例如，将 assertEquals("aabb", a.addString("aa", "bb"))；语句改为：assertEquals("cc", a.addString("aa", "bb"))；语句。两个字符串 aa 与 bb 的连接不可能等于 cc，修改后再运行一下，会出现测试失败的提示，如图 5-11 所示。

从上面可以看出一个失效(Tests failed: 1)，测试人员还能进一步查看出错的具体结果，单击“Click to see difference”，会出现结果比较(Comparison Failure)的对话框，说明期望值 cc

与实际结果为 aabb 不符,测试没通过。

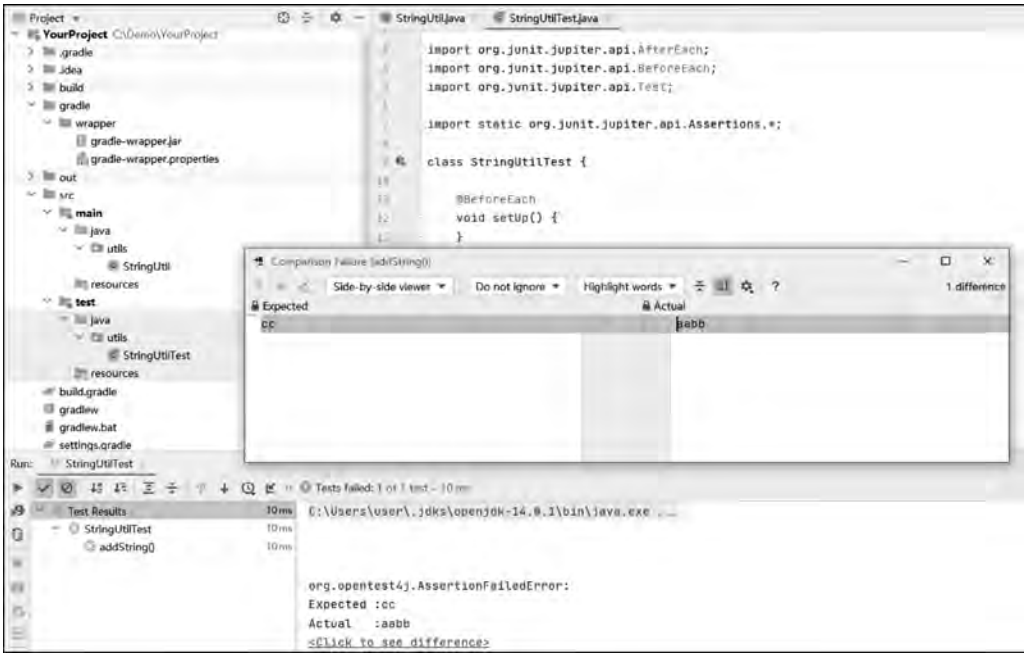

图 5-11 JUnit test case 测试出错示例

5.6.3 Mock 框架 Mockito

Mockito 是目前主流的支持 Java 语言的 Mock 框架,可以利用简单 API 快速创建测试替身(Mock 对象),可以很容易地编写测试来模拟被测试对象的依赖对象,如类和接口。以下是一个 Mockito 测试代码的示例。

```
import static org.mockito.Mockito.*;          //静态导入 Mockito

List mockedList = mock(List.class);           //在需要 Mock 的接口或者类上创建实例

mockedList.add("one");                        //像调用其他方法一样调用被 Mock 对象的方法
mockedList.clear();

verify(mockedList).add("one");                //验证测试执行时 add("one")确实被调用
verify(mockedList).clear();
```

Mockito 的 verify 方法提供了强大的验证功能。在上面的示例代码中,如果 add("one")方法调用成功,则程序正常运行,反之则会报告错误。此外,verify 方法还可以验证以下几个方面。

(1) 调用某个方法的次数,如 verify(mockedList,times(1)).add();语句。

(2) 验证没有调用任何方法,如 verifyZeroInteractions(mockedList);语句。

(3) 验证没有调用某个方法,如 verify(mockedList, never()).size();语句。

(4) 判断是否有未被验证的调用,如 verifyNoMoreInteractions(mockedList);语句。

(5) 验证调用的顺序,如 inOrder.verify(mockedList).add("one");语句。

Mockito 为 JUnit5 的扩展模式提供了一个实现：Mockito-junit-jupiter。开发人员可以通过将@ExtendWith(MockitoExtension.class)添加到测试类并使用@Mock 注释模拟字段来应用扩展。代码示例如下：

```
@ExtendWith(MockitoExtension.class)
public class ExampleTest {

    @Mock
    private List list;

    @Test
    public void shouldDoSomething() {
list.add(100);
    }

}
```

在一个 Gradle 项目中使用 Mockito 时，需要在 build.gradle 中添加 Mockito 的依赖。代码示例如下：

```
Dependencies {
    implementation 'org.mockito:mockito-core:3.7.7'
}
```

5.6.4 测试覆盖率工具 JaCoCo

单元测试用例执行结束后，通常还需要利用自动化工具自动统计测试覆盖率，通过计算单元测试用例对被测应用的代码覆盖率反映单元测试是否充分，为是否需要补充测试用例提供依据。常用的测试覆盖率统计工具包括支持 Java 语言的 JaCoCo，支持 Python 语言的 Coverage，支持多种语言的 Coverity 等。这里以 JaCoCo 为例进行介绍。

JaCoCo 是一个开源的 Java 测试覆盖率统计工具，使用插桩的方式来记录覆盖率，通过在被测应用的代码中插入探针(probe)采集覆盖信息。JaCoCo 提供了多种维度的覆盖率计数器，主要包括以下几种。

(1) 指令覆盖(Instructions，C0coverage)：JaCoCo 计数的最小单元是 Java 字节码指令，指令覆盖率提供有关已执行或遗漏的代码数量的信息。

(2) 分支覆盖(Branches，C1coverage)：度量 if 和 switch 语句的分支覆盖情况，计算一个方法中的所有判断的总分支数、已执行的和未执行的分支数量。

(3) 行覆盖(Lines)：当分配给该代码行的至少一条指令已经执行时，就认为源代码行已经执行了。

(4) 方法覆盖率：每个非抽象方法至少包含一条指令。当至少执行了一条指令时，方法就被认为已经执行了。

(5) 类覆盖(classes)：当一个类中至少有一个方法已执行，该类被认为已执行。

JaCoCo 可以嵌入 Ant、Maven、Gradle 中，并提供了 EclEmma Eclipse 插件，也可以使用 JavaAgent 技术监控 Java 程序。很多第三方的工具提供了对 JaCoCo 的集成，如 Sonar、Jenkins 等。如图 5-12 所示，是在 Eclipse 中安装了 JaCoCo 插件，执行完单元测试用例后显示的指令、分支、类、方法、代码行测试覆盖率报告。

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.jacoco.examples		58%		64%	24	53	97	193	19	38	6	12
org.jacoco.core		97%		93%	111	1,384	116	3,320	20	710	2	136
org.jacoco.agent.rt		77%		84%	31	121	62	310	21	74	7	20
jacoco-maven-plugin		90%		80%	37	186	46	412	8	111	0	19
org.jacoco.cli		97%		100%	4	109	10	275	4	74	0	20
org.jacoco.report		99%		99%	4	572	2	1,345	1	371	0	64
org.jacoco.ant		98%		99%	4	163	8	429	3	111	0	19
org.jacoco.agent		86%		75%	2	10	3	27	0	6	0	1
Total	1,358 of 27,250	95%	150 of 2,141	92%	217	2,598	344	6,311	76	1,495	15	291

图 5-12 Eclipse 中 JaCoCo 测试覆盖率报告

5.6.5 JUnit 5+Gradle 构建自动的单元测试

支持 Java 的主要有 3 个开源的构建工具：Ant、Maven 和 Gradle。Ant 最早出现，但目前 Gradle 和 Maven 已经超越 Ant 成为主流的 Java 构建工具。这 3 个构建工具都可以集成 JUnit，可以在软件版本构建过程中自动执行单元测试。下面以 JUnit 5 结合 Gradle 为例来讲解如何构建自动的单元测试。

Gradle 融合了 Ant 和 Maven 的功能，使用 Groovy 语言来声明项目设置，而不是 Maven 采用的 xml，让配置更加简洁。Gradle 4.6 及以上版本对 JUnit 提供了原生支持，要在构建过程中启动 JUnit 执行测试任务，只需在 Gradle 的配置文件 build.gradle 中添加 JUnit Jupiter 测试引擎的相关依赖，并在 test 任务声明中指定 useJunitPlatform。代码示例如下：

```
dependencies {
testImplementation("org.junit.jupiter:junit-jupiter-api:5.7.0")
testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:5.7.0")
}

test {
useJUnitPlatform()
}
```

然后运行 Gradle 进行构建任务，单元测试会在构建过程被执行，测试结果保存在工程项目目录下的 build-> reports-> tests-> test 中的 index.html 文件中，可以在浏览器中查看，如图 5-13 所示。

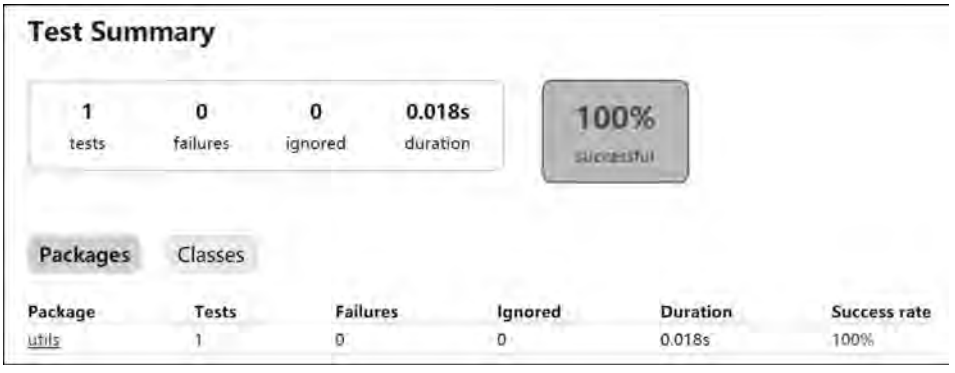

图 5-13 Gradle 中可视化的单元测试结果

Gradle 还支持和测试覆盖率统计工具 JaCoCo 的集成，实现在构建过程中单元测试用例的自动执行和测试覆盖率的自动统计。

首先，将 JaCoCo 插件添加到需要统计测试覆盖率的项目的 build.gradle 中。

```
plugins {
    id 'jacoco'
}
```

Gradle 将创建一个名为 jacocoTestReport 的新任务。默认情况下，HTML 报告是在 $ buildDir/reports/jacoco/test 中生成的。

还需要在 build.gradle 中定义测试任务执行完毕后生成测试覆盖率报告。

```
test {
finalizedBy jacocoTestReport        // report is always generated after tests run
}
jacocoTestReport {
dependsOn test                      // tests are required to run before generating the report
}
```

测试覆盖率报告的格式包括 xml、csv、html。在 build.gradle 中指定生成格式为 html 的报告。

```
jacocoTestReport {
    reports {
xml.enabled false
csv.enabled false
html.destination file(" $ {buildDir}/jacocoHtml")
    }
}
```

项目构建完毕后生成的测试覆盖率报告如图 5-14 所示。

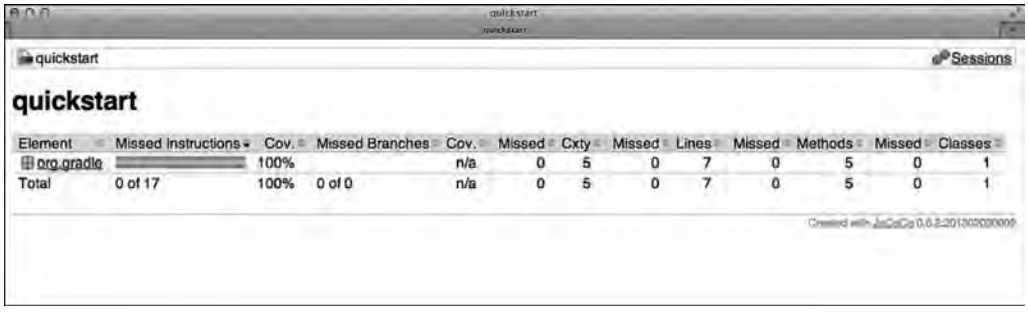

图 5-14 Gradle 中 JaCoCo 测试覆盖率报告

5.6.6 开源的单元测试工具

通过 JUnit 了解了单元测试工具的基本构成和功能。实际上，JUnit 只是开源的单元测试工具中的一个代表，还有许多开源的单元测试工具可以使用。例如，在 JUnit 基础之上扩展的一些工具，如 Boost、Cactus、CUTest、JellyUnit、Junitperf、JunitEE、Pisces 和 QtUnit 等。

在选择测试工具时，首先可考虑开源工具，毕竟开源工具投入成本低，而且有了源代码，能结合自己特定的需求进行修改、扩展，具有良好的定制性和适应性。如果开源工具不能满足要求，再考虑选用商业工具。

1. C/C++语言单元测试工具

(1) 适合各种操作系统：C Unit Test System，CppTest，CppUnit，CxxTest。

(2) Win32/Linux/Mac OS X：UnitTest++。

(3) Win32/Solaris/Linux：Splint。

(4) Mac OS X：ObjcUnit，OCUnit，TestKit。

(5) UNIX：cutee。

(6) Linux：GUnit。

(7) Windows：simplectest。

(8) 嵌入式系统：Embedded Unit。

(9) 其他：Cgreen 、POSIX Check。

2. Java 语言单元测试工具

(1) TestNG 的灵感来自 JUnit，消除了老框架的大多数限制，使开发人员可以编写更加灵活的测试代码，处理更复杂、量更大的测试。

(2) Surrogate Test framework 基于 AspectJ 技术，它是适合于大型、复杂 Java 系统的单元测试框架，并与 JUnit、MockEJB 和各种支持 Mock 对象的测试工具无缝结合。

(3) Mock Object 类工具：MockObjects、Xdoclet、EasyMock、MockCreator、MockEJB、ObjcUnit、jMock、JMockit 等。例如，EasyMock 通过简单的方法对于指定的接口或类生成 Mock 对象的类库，把测试与测试边界以外的对象隔离开，利用对接口或类的模拟来辅助单元测试。

(4) AssertJ 提供了丰富的断言方法帮助开发人员对复杂元素进行测试，相比 assertEquals()、assertTrue()等 JUnit、TestNG 自带的断言方法，AssertJ 提供的流式断言具有直观、易懂的特点，能够显著提高测试代码的可读性。

(5) Mockrunner 是 J2EE 环境中的单元测试工具，包括 JDBC、JMS 测试框架，支持 Struts、Servlet、EJB、过滤器和标签类。

(6) Dojo Objective Harness 是 Web 2.0(Ajax)UI 开发人员用于 JUnit 的工具。与已有的 JavaScript 单元测试框架(如 JSUnit)不同，DOH 不仅能够自动处理 JavaScript 函数，还可以通过命令行界面和基于浏览器的界面完成 UI 的单元测试。

(7) jWebUnit(http://jwebunit.sourceforge.net/)是基于 Java 的测试网络程序的框架，提供了一套测试见证和程序导航标准。以 HttpUnit 和 JUnit 单元测试框架为基础，提供了导航 Web 应用程序的高级 API，并通过一系列断言的组合来验证链接导航、表单输入项和提交、表格内容以及其他典型商务 Web 应用程序特性的正确性。jWebUnit 以 JAR 文件形式存在，很容易和大多数 IDE 集成起来。

(8) JSFUnit 测试框架构建在 HttpUnit 和 Apache Cactus 之上，对 JSF(Java Server Faces)应用和 JSF AJAX 组件实施单元测试，在同一个测试类里测试 JSF 产品的客户端和服务器端。它支持 RichFaces 和 Ajax4jsf 组件，还提供了 JSFTimer 组件来执行 JSF 生命周期的性能分析。通过 JSFUnit API，测试类方法可以提交表单数据，并且验证管理的 bean 是否被正确更新。借助 Shale 测试框架(Apache 项目)，可以对 Servlet 和 JSF 组件的 Mock 对象实现，也可以借助 Eclipse Web Tools Platform (WTP)和 JXInsight 协助对 JSF 应用进行更有效的测试。

(9) EvoSuite 是由英国谢菲尔德等大学联合开发的一款开源的智能化工具，用于自动生成测试用例集，生成的测试用例均符合 JUnit 的标准，可直接在 JUnit 中运行，并得到了 Google 和 Yourkit 的支持。通过使用此自动测试工具能够在保证代码覆盖率的前提下极大地提高测试人员的开发效率。但是只能辅助测试，并不能完全取代人工，测试用例的正确与否还需人工判断。

(10) Diffblue Cover 是另一款智能化的单元测试用例编写工具，通过分析 Java 应用程序编写反映当前行为的单元测试，提高测试覆盖率，并帮助开发人员在将来的代码更改中发现回归缺陷。

3. 其他语言单元测试工具

(1) HtmlUnit 是 JUnit 的扩展测试框架之一，使用如 table、form 等标识符将测试文档作为 HTML 来处理。

(2) NUnit 是类似于 JUnit、针对 C# 语言的单元测试工具。NUnit 利用了许多. NET 的特性，如反射机制。NUnitForms 是 NUnit 在 WinFrom 上的扩展。

(3) TestDriven. Net 是以插件的形式集成在 Visual Studio 中的单元测试工具，其前身是 NUnitAddIn。个人版可以免费下载使用，企业版是商业化的工具。

(4) PHPUnit 是针对 PHP 语言的单元测试工具。

(5) DUnit 是 xUnit 家族中的一员，用于 Delphi 的单元测试。

(6) SQLUnit 是 xUnit 家族的一员，以 XML 的方式来编写，用于对存储过程进行单元测试的工具，也可以用于针对数据库数据、性能的测试等。

(7) Easyb 是一个基于 Groovy 行为驱动开发的测试工具，为 Java 和 Groovy 测试。

(8) RSpec 是 Ruby 语言的新一代测试工具，与 Ruby 的核心库 Test::Unit 相比功能上和非常接近，RSpec 的优点是可以容易地编写领域特定语言(Domain Specific Language，DSL)，其目标是支持 BDD(Behaviour-Driven Development，行为驱动开发)，BDD 是一种融合了 TDDt、Acceptance Test Driven Planning 和 Domain Driven Design 的一种敏捷开发模型。

(9) Zentest 也是针对 Ruby 语言的单元测试工具，可以和 Autotest 一起使用。

5.7 系统集成的模式与方法

在软件开发中，经常会遇到这样的情况，单元测试时能确认每个模块都单独工作，但这些模块集成在一起之后会出现有些模块不能正常工作。这主要因为模块相互调用时接口会引入新的问题，包括接口参数不匹配、传递错误数据、全局数据结构出现错误等。这时，需要进行集成测试(Integration Test)。集成测试是将已分别通过测试的单元按设计要求集成起来再进行的测试，以检查这些单元之间的接口是否存在问题，包括接口参数的一致性引用、业务流程端到端的正确性等。

集成测试既要求参与的人熟悉单元的内部细节，又要求能够从足够高的层次上观察整个系统。一般由有经验的测试人员和软件开发者共同完成集成测试的计划和执行。

早期的软件应用采用单体架构，软件所有功能都放在一个工程里进行开发，各模块紧密耦合，相互依赖，将一个软件包整体部署到服务器上；而当前流行的软件架构是微服务架构，强调业务系统彻底的组件化和服务化，一个微服务完成一个特定的业务功能，服务之间通过轻量级的通信协议 HTTP、RPC 等进行交互。每个微服务可以独立开发并部署到不同的服务器

上。单体架构和微服务架构的区别如图 5-15 所示。

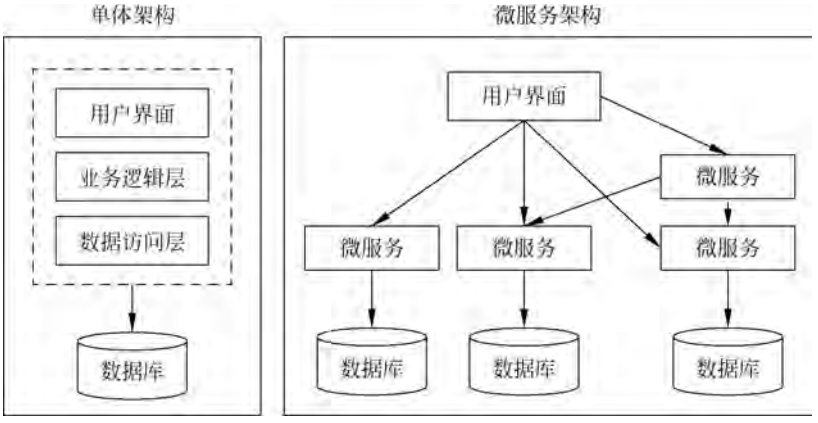

图 5-15 软件系统的单体架构和微服务架构对比

系统集成的模式在单体架构和微服务架构的软件系统中有很大不同,因此集成测试的方法也有很大不同,下面就分别介绍两种架构模式下的集成测试。

5.7.1 单体架构的集成测试

在开始集成测试时,首先需要选择何种集成模式。集成模式是软件集成测试中的策略体现,其重要性是明显的,直接关系到测试的效率、结果等,一般要根据具体的系统来决定采用哪种模式。集成测试基本可以概括为以下两种。

(1) 非渐增式测试模式:先分别测试每个模块,再把所有模块按设计要求放在一起结合成所要的程序,如大棒模式。

(2) 渐增式测试模式:把下一个要测试的模块同已经测试好的模块结合起来进行测试,测试是在模块一个一个的扩展下进行,其测试的范围逐步增大。

在非增量式集成中容易出现混乱,因为测试时可能发现一大堆错误,为每个错误定位和纠正非常困难,并且在改正一个错误的同时又可能引入新的错误,新旧错误混杂,更难断定出错的原因和位置。与之相反的是增量式集成模式,程序一段一段地扩展,每次测试的接口非常有限,错误易于定位和纠正,界面的测试也可做到完全彻底。

在实际工作中,一般采用渐增式测试模式,具体的实践有自顶向下、自底向上、混合策略等。

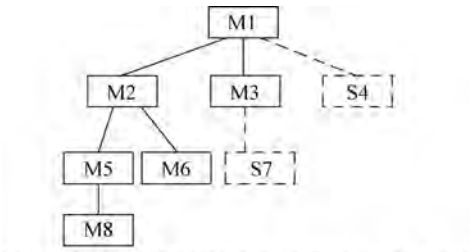

图 5-16 自顶向下集成方法示意图

1. 自顶向下法

自顶向下法(Top-down Integration)从主控模块("主程序")开始,沿着软件的控制层次向下移动,从而逐渐把各个模块结合起来。在集成过程中,可以使用深度优先的策略或宽度优先的策略,如图 5-16 所示,其具体步骤如下。

(1) 对主控模块进行测试,测试时用桩程序代替所有直接附属于主控模块的模块。

(2) 根据选定的结合策略(深度优先或宽度优先),每次用一个实际模块代替一个桩程序(新结合进来的模块往往又需要新的桩程序)。

(3) 在结合下一个模块的同时进行测试。

(4) 为了保证加入模块没有引进新的错误,可能需要进行回归测试(即全部或部分地重复以前做过的测试)。

从第(2)步开始不断地重复进行上述过程,直至完成。自顶向下法的主要优点是不需要测试驱动程序,能够在测试阶段的早期实现并验证系统的主要功能,而且能在早期发现上层模块的接口错误。其缺点是需要桩程序,可能遇到与此相联系的测试困难,低层关键模块中的错误发现较晚,而且用这种方法在早期不能充分展开人力。

2. 自底向上法

自底向上(Bottom-up Integration)测试从"原子"模块(即在软件结构最底层的模块)开始集成以进行测试,如图 5-17 所示,具体策略如下。

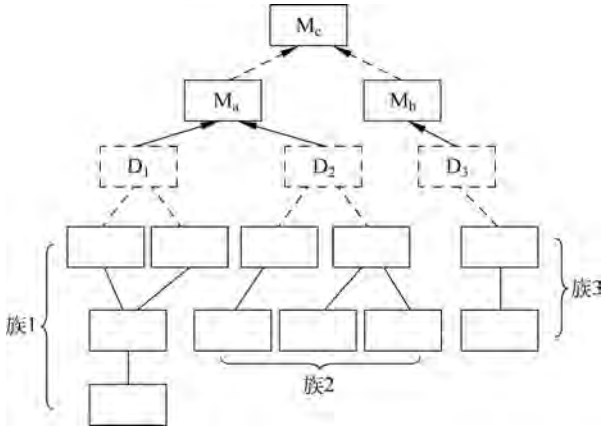

图 5-17　自底向上集成方法示意图

(1) 把底层模块组合成实现某个特定的软件子功能的族。

(2) 写一个驱动程序(用于测试的控制程序),协调测试数据的输入和输出。

(3) 对由模块组成的子功能族进行测试。

(4) 去掉驱动程序,沿软件结构自下向上移动,把子功能族组合起来形成更大的子功能族(Cluster)。

从第(2)步开始不断地重复进行上述过程,直至完成。

自底向上法的优缺点与自顶向下法刚好相反。

在具体测试中,可采用混合策略,即结合上述的两种方法——自顶向下法和自底向上法来逐步实施集成测试。

(1) 改进的自顶向下法:基本使用"自顶向下"法,但在测试早期,使用"自底向上"法测试少数的关键模块。

(2) 混合法:对软件结构中较上层,使用的是"自顶向下"法;对软件结构中较下层,使用的是"自底向上"法,两者相结合,如图 5-18 所示。

这种混合策略也有一些不同的组合方式,如三明治集成方法(Sandwich Integration),基本思想是一致的,自两头向中间集成,只是具体实现有些差异,如图 5-19 所示。

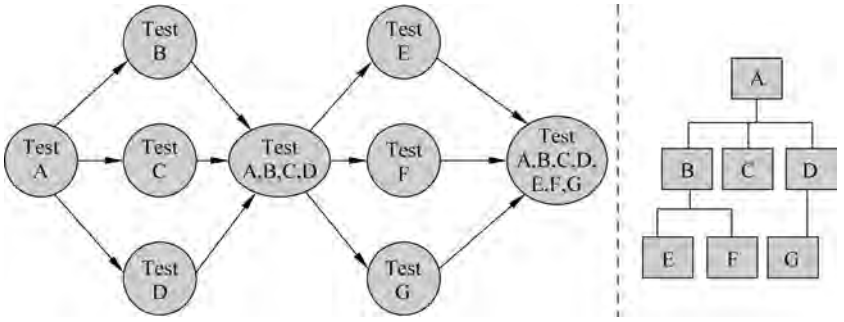

图 5-18　混合策略集成示意图

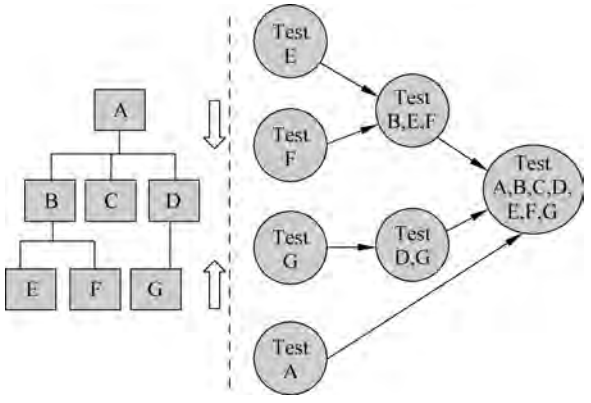

图 5-19　三明治集成方法示意图

采用三明治方法的优点是它将自顶向下和自底向上的集成方法有机地结合起来，不需要写桩程序，因为在测试初自底向上集成已经验证了底层模块的正确性。采用这种方法的主要缺点是在真正集成之前每一个独立的模块都没有完全测试过。

5.7.2　微服务架构的集成测试

在微服务架构下，传统的单体应用拆分为多个微服务，形成了多个松耦合的微服务组件模块。原来单个系统内部的 API 接口调用变成了微服务之间的接口调用，而且不同的微服务很可能是由不同的开发团队负责开发。一个微服务的具体结构包括以下 5 部分。

(1) 资源组件，负责将请求信息映射到业务逻辑，同时将业务逻辑组件产生的结果转换成预先定义好的数据格式响应信息。

(2) 服务层，负责协调多个领域之间的操作以及和其他子系统之间的交互。

(3) 领域层，负责表达业务概念、业务状态信息和业务规则，是软件应用的核心。

(4) 仓库层，作为数据源的网关，处理连接、查询，以及将输出和领域对象模型进行适配等工作。

(5) 数据映射器，负责在处理业务逻辑的领域对象和数据库之间传递数据。

(6) 网关和 HTTP 客户端，负责和外部服务进行通信。网关用来处理底层的通信协议，封装了所有需要连接外部服务的逻辑，通常使用一个 HTTP 客户端连接其他外部服务。

微服务下的集成测试是为了验证一个子系统或功能模块和外部组件之间的正常通信。外部组件包括其他的微服务或者外部数据存储系统。如图 5-20 所示，在微服务架构下集成测试需要验证微服务结构中的两部分(图中两个虚线框)，一部分是一个微服务对外通信的模块(网关和 HTTP 客户端)和外部组件的通信，集成测试负责验证一个微服务与外部服务的通信是否正常，包括二者之间的连接以及交互协议相关的问题；另一部分是微服务的数据库访问模块(仓库和数据映射器)和外部数据库的交互，集成测试需要负责验证微服务所使用的数据结构是否与数据源相符。

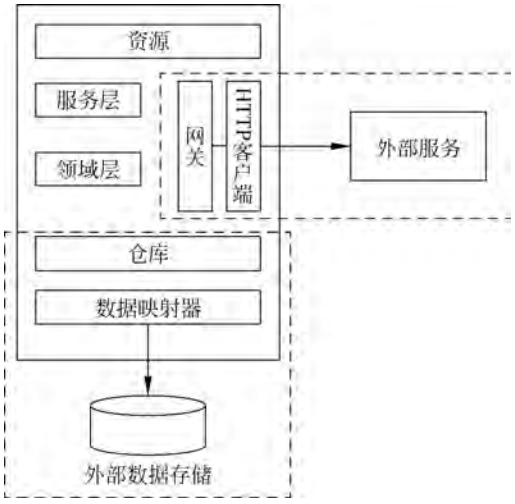

图 5-20 微服务架构下的集成测试

一个微服务和外部服务的集成测试的测试步骤如下。

(1) 启动被测微服务和外部服务的实例；

(2) 调用被测微服务，该服务会通过外部服务提供的 API 读取响应数据；

(3) 检查被测微服务是否能正确解析返回结果。

一个微服务和外部数据存储的集成测试的测试步骤如下。

(1) 启动外部数据库；

(2) 连接被测应用到数据库；

(3) 调用被测微服务，该服务会往数据库写数据；

(4) 读取数据库，查看期望的数据是否被写到了数据库里。

5.7.3 持续集成及其测试

通常系统集成都会采用持续集成(Continuous Integration，CI)的策略，软件开发中各个模块不是同时完成，根据进度将完成的模块尽可能早地进行集成，有助于尽早发现 Bug，避免集成中大量 Bug 涌现。

在没有采用 CI 策略的开发中，开发人员经常需要集中开会来分析软件究竟在什么地方出了错。因为某个程序员在写自己这个模块代码时，可能会影响其他模块的代码，造成与已有程序的变量冲突、接口错误，结果导致被影响的人还不知道发生了什么，Bug 就出现了。这种 Bug 是最难查的，因为问题不是出在某一个人的领域里，而是出在两个人的交流上面。随着时间的推移，问题会逐渐恶化。通常，在集成阶段出现的 Bug 早在几周甚至几个月之前就已经

存在了。结果,开发者需要在集成阶段耗费大量的时间和精力来寻找这些 Bug 的根源。

如果使用了 CI,这样的 Bug 绝大多数都可以在引入的第一天就被发现。而且,由于一天之中发生变动的部分并不多,所以可以很快找到出错的位置。如果找不到 Bug 究竟在哪里,也可以不把这些代码集成到产品中去。所以,持续集成可以减少集成阶段消灭 Bug 所消耗的时间,从而最终提高软件开发的质量与效率。

而在敏捷开发模式中,CI 已经成为核心实践之一。除了每日构建这种已经非常普遍的集成方式,开发人员每次提交代码就会触发持续集成,一天可以多达几百、上千次的持续集成。只要开发人员提交代码,就能触发对代码的快速检查,因为每次提交的代码都只包含小批量的变更,所以发现问题和解决问题的效率更高。

1. 持续集成中的测试活动

持续集成在 1996 年就被列入极限编程的核心实践之一。2006 年,Martin Fowler 提出了比较完善的方法与实践,并且给出了目前大家普遍认可的定义:持续集成是一种软件开发实践,即团队开发成员经常集成他们的工作,通常每个成员每天至少集成一次,也就意味着每天可能会发生多次集成。每次集成都通过自动化的构建(测试)来验证,从而尽快地发现集成错误。

代码的一次持续集成大概包括以下活动:编译、测试、打包、部署、结果反馈。开发人员从提交代码变更触发构建到结果反馈所需的时间最好在 10min 内,并且整个过程是自动化的,如图 5-21 所示。

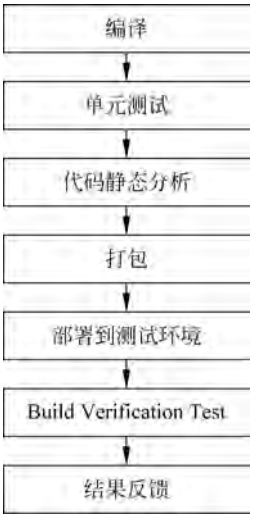

图 5-21 持续集成活动

持续集成中的自动化测试活动包括单元测试、代码静态测试和构建包的验证(Build Verification Test,BVT)。

(1) 单元测试是指对软件最小可测试单元(函数或类)进行验证,目的是发现代码底层的缺陷。单元测试对外部系统的依赖少,运行时间通常在秒级,发现代码缺陷的成本低、效率高,敏捷开发的研发团队需要对单元测试的代码覆盖率提出比较高的要求,比如 80%以上。

(2) 代码的静态测试,也叫静态分析,通过静态分析工具不需要运行应用程序就可以对软件代码进行检查,发现代码规范问题、结构问题,以及代码错误等。

(3) BVT 执行基本的功能和接口测试,用来验证软件的基本功能是否能正常工作。这种 BVT 可以看作是非严格意义上的集成测试,因为如果集成有问题,会在版本构建中出现问题,会被 BVT 发现。

2. CI 测试工具

目前业界通常采用构建工具 Maven、Gradle、Ant 在 CI/CD 调度工具中完成持续构建和集成,然后在此基础上自动触发自动化测试,完成 BVT。基于良好的基础设施,就可以做到持续构建、持续集成测试。良好的持续集成(CI)环境能够支持代码的自动构建、自动部署、自动验证和自动反馈,如图 5-22 所示。

因此,CI 环境需要强大的工具链的支持,包括代码管理工具、版本构建工具、CI 调度工具、代码静态分析工具、单元测试工具,以及用来支持 BVT 测试的 UI 或接口自动化测试工具

等。CI 调度工具包括 Jenkins、Bamboo、Travis、GitLab CI 等。关于测试工具，前面已经介绍过代码静态分析工具和单元测试工具，UI 自动化测试工具包括 Selenium、Appium 等，接口自动化工具包括 REST Assured、Jmeter 等。

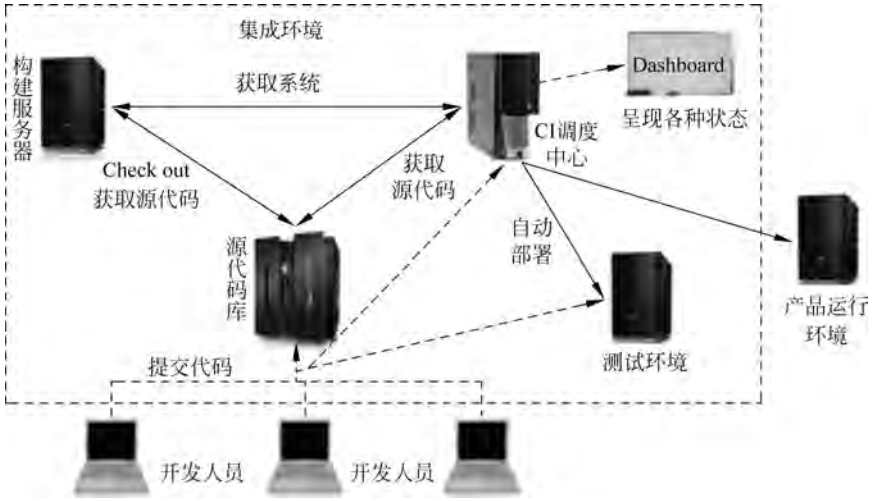

图 5-22 CI 环境基本构成示意图

Jenkins 是一个可扩展的自动化服务器，可以用作持续集成中的调度管理服务器。它通过 1000 多个插件集成持续集成环境中需要的几乎所有的工具。另外，Jenkins 支持分布式运行，支持跨多个平台的构建、测试和软件部署。

Jenkins 中提供对 Maven、Gradle、Ant 等构建工具的支持。开发人员将代码提交到源代码库，Jenkins 作为 CI 调度中心可以定时或周期性的触发构建，或定时检查是否有代码变更，有变更时就会触发构建。如 5.3 节和 5.6 节所讲解的，构建工具通过配置各类工具的插件支持代码静态测试和单元测试。当构建触发时，构建工具就会启动包括代码静态检测、单元测试在内的构建过程。

Jenkins 1.x 通过界面手动操作来配置各类任务，Jenkins 2.x 以代码的形式进行配置，用户可以选择创建一个 pipeline 项目，所有的逻辑写在 Jenkinsfile 中。下面举例进行说明。

(1) 执行持续集成时，在 Jenkins 中收集并展示 JUnit 测试报告：首先需要在 Gradle 项目中 build.gradle 文件添加 JUnit 的依赖，然后在 Jenkins 中安装 Jenkins JUnit 插件，并在 Jenkins 中加入下列 JUnit 步骤。

```
post {
    always{
        junit testResults: "**/target/surefire-reports/*.xml"

    }
}
```

(2) 在 Jenkins 中集成 SonarQube：首先在 Gradle 项目 build.gradle 文件中添加 sonarscanner 插件，然后在 Jenkins 中安装 SonarScanner 插件(https://plugins.jenkins.io/sonar/)，接着配置 SonarQube 服务器信息，并在 Jenkinsfile 中加入以下步骤。

```
    node {
```

```
        stage('SCM') {
          git 'https://github.com/foo/bar.git'
        }
      stage('SonarQube analysis') {
      withSonarQubeEnv() { // Will pick the global server connection you have configured
sh './gradlew sonarqube'
          }
        }
}
```

构建完毕后，登录 SonarQube 管理界面，就可以看到代码质量检查的结果了。

小结

借助代码静态测试，可以更好地清楚缺陷，提高代码的质量。静态测试主要体现在两个方面，一方面通过工具进行自动分析，另一方面可以通过人工评审，最常用的方法是互为评审(Peer review)，对一些关键代码或新人写的代码，主要采用走查(Walk Through)和会议审查(Inspection)等评审方式。此外，可以借助静态测试工具来完成对所有代码的扫描和分析，输出测试报告，这种方式的应用越来越多。

单元测试的对象是程序系统中的最小单元——模块或组件上，其目标不仅是测试代码的功能性，还需确保代码在结构上可靠且健全。单元测试采用动态测试技术，主要采用基于代码的逻辑覆盖方法，从程序的内部结构出发设计测试用例，检查程序模块或组件的已实现的功能与定义的功能是否一致，并结合基于输入域的测试方法、组合测试方法等，完成对调用参数、变量取值等测试，最终完成控制流和数据流的分析与测试。由于模块规模小、功能单一、逻辑简单，测试人员有可能通过模块说明书和源程序，清楚地了解该模块的 I/O 条件和模块的逻辑结构，采用结构测试的用例，尽可能达到彻底测试，使之对任何合理和不合理的输入都能鉴别和响应。

本章还介绍了代码静态测试和单元测试中常用的开源测试工具。静态检测工具重点介绍了 FindBugs、PMD、CheckStyle 和 SonarQube。单元测试工具重点介绍了单元测试框架 JUnit，以及如何用 JUnit 5＋Gradle 构建自动的单元测试。通过使用这些不同方面的单元测试工具，可以更容易实施单元测试，不断复用测试脚本，减少单元测试的工作量。

代码静态分析、单元测试和集成测试紧密相关，几乎同步进行，而且目前业界都提倡持续集成、持续测试。在持续测试中，更关注集成测试的基础设施，从而能够比较容易地完成持续构建和持续集成测试。

思考题

1. 为什么要进行单元测试？单元测试的主要任务有哪些？

2. 单元测试的对象不可能是一组函数或多个程序的组合，为什么？

3. 单元测试一般由开发人员完成，并采用动态测试技术，这样会获得更高的测试效率和更彻底的测试，谈谈其中的道理。

4. 代码评审有哪些方法？哪一种方法比较有效？为什么？

5. 如何做好单元测试各个阶段的管理工作？

6. 动手写一个类的 JUnit 单元测试方法，并让其运行成功，体验 JUnit 的使用方法。

7. CheckStyle/PMD 与 FindBugs 各自的主要功能是什么？试着在某个 Java EE 项目中使用 FindBugs 进行检查，并分析检查结果。

8. SonarQube 的主要功能是什么？试着对一个已有的项目用 SonarQube 进行代码质量分析。

9. 进一步了解持续集成和持续测试的知识，设法自己搭建这样的持续集成环境。

实验 2 单元测试实验

（共 2 学时）

使用 JUnit 工具，针对 Spring Unit Testing 控制器代码中 ItemController 类进行测试，编写对应的测试类以完成单元测试，最终提交测试代码。

```
package com.sprint.unittesting.unittesting.controller;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import com.sprint.unittesting.unittesting.business.ItemBusinessService;
import com.sprint.unittesting.unittesting.model.Item;

@RestController
public class ItemController {

    @Autowired
    private ItemBusinessService businessService;

    @GetMapping("/dummy-item")
    public Item dummyItem() {
        return new Item(1, "Ball", 10, 100);
    }

    @GetMapping("/item-from-business-service")
    public Item itemFromBusinessService() {
        Item item = businessService.retreiveHardcodedItem();

        return item;
    }

    @GetMapping("/all-items-from-database")
    public List<Item> retrieveAllItems() {
        return businessService.retrieveAllItems();
    }

}
```