第3章 代码静态测试

3.1 代码静态测试

3.1.1 静态测试

静态测试(Static Testing)是指不运行被测试程序本身,仅通过分析或检查源程序的语法、结构、过程、接口等来检查程序的正确性。因为静态测试方法并不真正运行被测程序,只进行特性分析,所以,静态测试常常称为“静态分析”。静态测试是对被测程序进行特性分析方法的总称。

静态测试包括代码检查、静态结构分析、代码质量度量等。它可以由人工进行,充分发挥人的逻辑思维优势,也可以借助软件工具自动进行。代码检查包括代码走查、桌面检查、代码审查等。代码检查主要检查代码和设计的一致性,代码对标准的遵循,代码的可读性,代码的逻辑表达的正确性,代码结构的合理性等。

静态测试可以完成下列工作。

(1) 发现程序中的下列错误:错用局部变量和全局变量,未定义的变量,不匹配的参数,不适当的循环嵌套或分支嵌套,死循环,不允许的递归,调用不存在的子程序,遗漏标号或代码。

(2) 找出以下问题的根源:从未使用过的变量,不会执行到的代码,从未使用过的标号,潜在的死循环。

(3) 提供程序缺陷的间接信息:所用变量和常量的交叉应用表,是否违背编码规则,标识符的使用方法和过程的调用层次。

(4) 为进一步查找错误做好准备。

(5) 为测试用例选取提供指导。

(6) 进行符号测试。

静态测试成本低、效率较高,并且可以在软件开发早期阶段发现软件缺陷。因此静态测试是一种非常有效而重要的测试技术。在实际使用中,代码检查比动态测试更有效率,能快速找到缺陷,发现30%～70%的逻辑设计和编码缺陷,而且代码检查看到的是问题本身而非征兆。但是代码检查非常耗费时间,而且代码检查需要知识和经验的积累。

3.1.2 静态测试工具

静态测试工具直接对代码进行分析，不需要运行代码，也不需要对代码编译链接生成可执行文件。静态测试工具一般是对代码进行语法扫描，找出不符合编码规范的地方，根据某种质量模型评价代码的质量，生成系统的调用关系图等。

下面介绍几款常用的静态测试工具。

1. PC-lint

PC-lint 是 GIMPEL SOFTWARE 公司开发的 C/C++软件代码静态分析工具，它的全称是 PC-lint/FlexeLint for C/C++。PC-lint 能够在 Windows、MS-DOS 和 OS/2 平台上使用，以二进制可执行文件的形式发布，而 FlexeLint 运行于其他平台，以源代码的形式发布。PC-lint 在全球拥有广泛的客户群，许多大型的软件开发组织都把 PC-lint 检查作为代码走查的第一道工序。PC-lint 不仅能够对程序进行全局分析，识别没有被适当检验的数组下标，报告未被初始化的变量，警告使用空指针以及冗余的代码，还能够有效地提出许多程序在空间利用、运行效率上的改进点。

网站地址：http://www.gimpel.com/html/index.htm

2. Checkstyle

Checkstyle 是 SourceForge 下的一个项目，提供了一个帮助 Java 开发人员遵守某些编码规范的工具。它能够自动化代码规范检查过程，从而使得开发人员从枯燥的任务中解脱出来。Checkstyle 可以有效检视代码，以便更好地遵循代码编写标准，特别适用于小组开发时彼此间的编码规范和统一。Checkstyle 提供了高可配置性，以便适用于各种代码规范，除了使用它提供的几种常见标准之外，也可以定制自己的标准。

网站地址：http://checkstyle.sourceforge.net

3. Logiscope

Logiscope 是 IBM Rational(原 Telelogic)推出的专用于软件质量保证和软件测试的产品。其主要功能是对软件做质量分析和测试以保证软件的质量，并可做认证、反向工程和维护，特别是针对要求高可靠性和高安全性的软件项目和工程。Logiscope 支持 4 种源代码语言：C，C++，Java 和 Ada。

Logiscope 工具集包含以下三个功能组件。

Logiscope RuleChecker：根据工程中定义的编程规则自动检查软件代码错误，可直接定位错误。RuleChecker 包含大量标准规则，用户也可定制创建规则，自动生成测试报告。

Logiscope Audit：定位错误模块，可评估软件质量及复杂程度。Audit 提供代码的直观描述，并自动生成软件文档。

Logiscope TestChecker：测试覆盖分析，显示没有测试的代码路径，基于源码结构分析。TestChecker 直接反馈测试效率和测试进度，协助进行衰退测试。既可在主机上测试，也可在目标板上测试，支持不同的实时操作系统，并支持多线程。

4. Splint

Splint 是一个 GNU 免费授权的 Lint 程序，是一个动态检查 C 语言程序安全弱点和编写错误的程序。Splint 会进行多种常规检查，包括未使用的变量，类型不一致，使用未定义变量，无法执行的代码，忽略返回值，执行路径未返回，无限循环等错误。

网站地址：http://www.splint.org/

5. FindBugs

FindBugs 是由马里兰大学提供的一款开源 Java 静态代码分析工具。FindBugs 通过检查类文件或 JAR 文件，将字节码与一组缺陷模式进行对比从而发现代码缺陷，完成静态代码分析。FindBugs 既提供可视化 UI 界面，同时也可以作为 Eclipse 插件使用。

FindBugs 可以简单高效全面地发现程序代码中存在的 Bug，Bad Smell，以及潜在隐患。针对各种问题，FindBugs 提供了简单的修改意见供我们重构时进行参考。通过使用 FindBugs，可以一定程度上降低 Code Review 的工作量，并且会提高 Review 效率。

网站地址：http://findbugs.sourceforge.net/

3.2 Checkstyle

3.2.1 Checkstyle 简介

Checkstyle 是 SourceForge 下的一个项目，提供了一个帮助 Java 开发人员遵守某些编码规范的工具。Checkstyle 可以根据设置好的编码规则来检查代码，比如符合规范的变量命名，良好的程序风格等。它能够自动化代码规范检查过程，从而使开发人员从这项重要而枯燥的任务中解脱出来。

Checkstyle 是一款检查 Java 程序代码样式的工具，可以有效检视代码以便更好地遵循代码编写标准，特别适用于小组开发时彼此间的样式规范和统一。

Checkstyle 提供了高可配置性，以便适用于各种代码规范。可以只检查一种规则，也可以检查几十种规则，可以使用 Checkstyle 自带的规则，也可以自己增加检查规则。Checkstyle 支持几乎所有主流 IDE，包括 Eclipse、IntelliJ、NetBeans、JBuilder 等 11 种。

需要强调的是，Checkstyle 只能做检查，而不能修改代码。

Checkstyle 检验的主要内容如下。

(1) Annotations(注释)；

(2) Javadoc Comments(Javadoc 注释)；

(3) Naming Conventions(命名约定)；

(4) Headers(文件头检查)；

(5) Imports(导入检查)；

(6) Size Violations(检查大小)；

(7) Whitespace(空白)；

(8) Modifiers(修饰符)；

(9) Blocks(块);
(10) Coding Problems(代码问题);
(11) Class Design(类设计);
(12) Duplicates(重复);
(13) Metrics(代码质量度量);
(14) Miscellaneous(杂项)。

3.2.2 Checkstyle 规则文件

1. Checkstyle 原理

Checkstyle 配置是通过指定 modules 来应用到 Java 文件的。modules 是树状结构,以一个名为 Checker 的 module 作为 root 节点,一般的 checker 都会包括 TreeWalker 子 module。可以参照 Checkstyle 中的 sun_checks.xml,这是根据 Sun 的 Java 语言规范写的配置。

在 XML 配置文件中通过 module 的 name 属性来区分 module,module 的 properties 可以控制如何去执行这个 module,每个 property 都有一个默认值,所有的 check 都有一个 severity 属性,用它来指定 check 的 level。TreeWalker 为每个 Java 文件创建一个语法树,在节点之间调用 submodules 的 Checks。

2. Checkstyle 检查项

1) Annotations
(1) Annotation Use Style(注解使用风格)
这项检查可以控制要使用的注解的样式。
(2) Missing Deprecated(缺少 Deprecad)
检查 java.lang.Deprecated 注解或@deprecated 的 Javadoc 标记是否同时存在。
(3) Missing Override(缺少 Override)
当出现{@inheritDoc}的 Javadoc 标签时,验证 java.lang.Override 注解是否出现。
(4) Package Annotation(包注解)
这项检查可以确保所有包的注解都在 package-info.java 文件中。
(5) Suppress Warnings(抑制警告)
这项检查允许用户指定不允许 SuppressWarnings 抑制哪些警告信息,还可以指定一个 TokenTypes 列表,其中包含所有不能被抑制的警告信息。

2) Javadoc Comments
(1) Package Javadoc(包注释)
检查每个 Java 包是否都有 Javadoc 注释。
(2) Method Javadoc(方法注释)
检查方法或构造器的 Javadoc。
(3) Style Javadoc(风格注释)
验证 Javadoc 注释,以便于确保它们的格式。可以检查以下注释:接口声明、类声明、方法声明、构造器声明、变量声明。

(4) Type Javadoc(类型注释)
检查方法或构造器的 Javadoc。
(5) Variable Javadoc(变量注释)
检查变量是否具有 Javadoc 注释。
(6) Write Tag(输出标记)
将 Javadoc 作为信息输出。
3) Naming Conventions
(1) Abstract Class Name(抽象类名称)
检查抽象类的名称是否遵守命名规约。
(2) Class Type Parameter Name(类的类型参数名称)
检查类的类型参数名称是否遵守命名规约。
(3) Constant Names(常量名称)
检查常量(用 static final 修饰的字段)的名称是否遵守命名规约。
(4) Local Final Variable Names(局部 final 变量名称)
检查局部 final 变量的名称是否遵守命名规约。
(5) Local Variable Names(局部变量名称)
检查局部变量的名称是否遵守命名规约。
(6) Member Names(成员名称)
检查成员变量(非静态字段)的名称是否遵守命名规约。
(7) Method Names(方法名称)
检查方法名称是否遵守命名规约。
(8) Method Type Parameter Name(方法的类型参数名称)
检查方法的类型参数名称是否遵守命名规约。
(9) Package Names(包名称)
检查包名称是否遵守命名规约。
(10) Parameter Names(参数名称)
检查参数名称是否遵守命名规约。
(11) Static Variable Names(静态变量名称)
检查静态变量(用 static 修饰,但没用 final 修饰的字段)的名称是否遵守命名规约。
(12) Type Names(类型名称)
检查类的名称是否遵守命名规约。
4) Headers
(1) Header(文件头)
检查源码文件是否开始于一个指定的文件头。
(2) Regular Expression Header(正则表达式文件头)
检查 Java 源码文件头部的每行是否匹配指定的正则表达式。
5) Imports
(1) Avoid Star(Demand)Imports(避免通配符导入)
检查是否有 import 语句使用 * 符号。从一个包中导入所有的类会导致包之间的紧耦

合，当一个新版本的库引入了命名冲突时，这样就有可能导致问题发生。

(2) Avoid Static Imports(避免静态导入)

检查没有静态导入语句。

(3) Illegal Imports(非法导入)

检查是否导入了指定的非法包。

(4) Import Order Check(导入顺序检查)

检查导入包的顺序/分组。

(5) Redundant Imports(多余导入)

检查是否存在多余的导入语句。如果一条导入语句满足以下条件，那么就是多余的：①它是另一条导入语句的重复。也就是，一个类被导入了多次。②从 java. lang 包中导入类，例如，导入 java. lang. String。③从当前包中导入类。

(6) Unused Imports(未使用导入)

检查未使用的导入语句。

CheckStyle 使用一种简单可靠的算法来报告未使用的导入语句。如果一条导入语句满足以下条件，那么就是未使用的：①没有在文件中引用。②它是另一条导入语句的重复。③从 java. lang 包中导入类。④从当前包中导入类。⑤可选：在 Javadoc 注释中引用它。

(7) Import Control(导入控制)

控制允许导入每个包中的哪些类。可用于确保应用程序的分层规则不会违法，特别是在大型项目中。

6) Size Violations(尺寸超标)

(1) Anonymous inner classes lengths(匿名内部类长度)

检查匿名内部类的长度。

(2) Executable Statement Size(可执行语句数量)

将可执行语句的数量限制为一个指定的限值。

(3) Maximum File Length(最大文件长度)

检查源码文件的长度。

(4) Maximum Line Length(最大行长度)

检查源码每行的长度。

(5) Maximum Method Length(最大方法长度)

检查方法和构造器的长度。

(6) Maximum Parameters(最大参数数量)

检查一个方法或构造器的参数的数量。

(7) Outer Type Number(外层类型数量)

检查在一个文件的外层(或根层)中声明的类型的数量。

(8) Method Count(方法总数)

检查每个类型中声明的方法的数量。

7) Whitespace

(1) Generic Whitespace(范型标记空格)

检查范型标记＜和＞的周围的空格是否遵守标准规约。

(2) Empty For Initializer Pad(空白 for 初始化语句填充符)

检查空的 for 循环初始化语句的填充符,也就是空格是否可以作为 for 循环初始化语句空位置的填充符。如果代码自动换行,则不会进行检查。

(3) Empty For Iterator Pad(空白 for 迭代器填充符)

检查空的 for 循环迭代器的填充符,也就是空格是否可以作为 for 循环迭代器空位置的填充符。

(4) No Whitespace After(指定标记之后没有空格)

检查指定标记之后没有空格。若要禁用指定标记之后的换行符,将 allowLineBreaks 属性设为 false 即可。

(5) No Whitespace Before(指定标记之前没有空格)

检查指定标记之前没有空格。若要允许指定标记之前的换行符,将 allowLineBreaks 属性设为 true 即可。

(6) Operator Wrap(运算符换行)

检查代码自动换行时,运算符所处位置的策略。

(7) Method Parameter Pad(方法参数填充符)

检查方法定义、构造器定义、方法调用、构造器调用的标识符和参数列表的左圆括号之间的填充符。如果标识符和左圆括号位于同一行,那么就检查标识符之后是否需要紧跟一个空格。如果标识符和左圆括号不在同一行,那么就报错,除非将规则配置为允许使用换行符。想要在标识符之后使用换行符,将 allowLineBreaks 属性设置为 true 即可。

(8) Paren Pad(圆括号填充符)

检查圆括号的填充符策略,也就是在左圆括号之后和右圆括号之前是否需要有一个空格。

(9) Typecast Paren Pad(类型转换圆括号填充符)

检查类型转换的圆括号的填充符策略。也就是,在左圆括号之后和右圆括号之前是否需要有一个空格。

(10) File Tab Character(文件制表符)

检查源码中没有制表符('\t')。

(11) Whitespace After(指定标记之后有空格)

检查指定标记之后是否紧跟了空格。

(12) Whitespace Around(指定标记周围有空格)

检查指定标记的周围是否有空格。

8) Regexp

(1) RegexpSingleline(正则表达式单行匹配)

检查单行是否匹配一条给定的正则表达式。可以处理任何文件类型。

(2) RegexpMultiline(正则表达式多行匹配)

检查多行是否匹配一条给定的正则表达式。可以处理任何文件类型。

(3) RegexpSingleLineJava(正则表达式单行 Java 匹配)

用于检测 Java 文件中的单行是否匹配给定的正则表达式。它支持通过 Java 注释抑制匹配操作。

9) Modifiers

(1) Modifier Order(修饰符顺序)

检查代码中的标识符的顺序是否符合指定的顺序。正确的顺序应当如下：public、protected、private、abstract、static、final、transient、volatile、synchronized、native、strictfp。

(2) Redundant Modifier(多余修饰符)

在以下部分检查是否有多余的修饰符：①接口和注解的定义；②final 类的方法的 final 修饰符；③被声明为 static 的内部接口声明。接口中的变量和注解默认就是 public、static、final 的，因此，这些修饰符也是多余的。因为注解是接口的一种形式，所以它们的字段默认也是 public、static、final 的。定义为 final 的类是不能被继承的，因此，final 类的方法的 final 修饰符也是多余的。

10) Blocks

(1) Avoid Nested Blocks(避免嵌套代码块)

找到嵌套代码块，也就是在代码中无节制使用的代码块。

(2) Empty Block(空代码块)

检查空代码块。

(3) Left Curly Brace Placement(左花括号位置)

检查代码块的左花括号的放置位置。通过 property 选项指定验证策略。

(4) Need Braces(需要花括号)

检查代码块周围是否有大括号，可以检查 do、else、if、for、while 等关键字所控制的代码块。

(5) Right Curly Brace Placement(右花括号位置)

检查 else、try、catch 标记的代码块的右花括号的放置位置。通过 property 选项指定验证策略。

11) Coding Problems

(1) Avoid Inline Conditionals(避免内联条件语句)

检测内联条件语句。

(2) Covariant Equals(共变 equals 方法)

检查定义了共变 equals()方法的类中是否同样覆盖了 equals(java.lang.Object)方法。

(3) Default Comes Last(默认分支置于最后)

检查 switch 语句中的 default 是否在所有的 case 分支之后。

(4) Declaration Order Check(声明顺序检查)

根据 Java 编程语言的编码规约，一个类或接口的声明部分应当按照以下顺序出现：①类(静态)变量。首先应当是 public 类变量，然后是 protected 类变量，再是 package 类变量(没有访问标识符)，最后是 private 类变量。②实例变量。首先应当是 public 类变量，然后是 protected 类变量，接下来是 package 类变量(没有访问标识符)，最后是 private 类变量。③构造器。④方法。

(5) Empty Statement(空语句)

检测代码中是否有空语句(也就是单独的;符号)。

(6) Equals Avoid Null(避免调用空引用的 equals 方法)

检查 equals()比较方法中，任意组合的 String 常量是否位于左边。

(7) Equals and HashCode(equals 方法和 hashCode 方法)

检查覆盖了 equals()方法的类是否也覆盖了 hashCode()方法。

(8) Explicit Initialization(显式初始化)

检查类或对象的成员是否显式地初始化为成员所属类型的默认值(对象引用的默认值为 null,数值和字符类型的默认值为 0,布尔类型的默认值为 false)。

(9) Fall Through(跨越分支)

检查 switch 语句中是否存在跨越分支。如果一个 case 分支的代码中缺少 break、return、throw 或 continue 语句,那么就会导致跨越分支。

(10) Illegal Catch(非法异常捕捉)

从不允许捕捉 java. lang. Exception、java. lang. Error、java. lang. RuntimeException 的行为。

(11) Illegal Throws(非法异常抛出)

这项检查可以用来确保类型不能声明抛出指定的异常类型。从不允许声明抛出 java. lang. Error 或 java. lang. RuntimeException。

(12) Illegal Tokens(非法标记)

检查不合法的标记。

(13) Illegal Type(非法类型)

检查代码中是否有在变量声明、返回值、参数中都没有作为类型使用过的特定类。包括一种格式检查功能,默认情况下不允许抽象类。

(14) Inner Assignment(内部赋值)

检查子表达式中是否有赋值语句,例如 String s = Integer. toString(i = 2);。

(15) JUnit Test Case(JUnit 测试用例)

确保 setUp()、tearDown()方法的名称正确,没有任何参数,返回类型为 void,是 public 或 protected 的。同样确保 suite()方法的名称正确,没有参数,返回类型为 junit. framewotk. Test,并且是 public 和 static 的。

(16) Magic Number(幻数)

检查代码中是否含有“幻数”,幻数就是没有被定义为常量的数值文字。默认情况下,-1、0、1、2 不会被认为是幻数。

(17) Missing Constructor(缺少构造器)

检查类(除了抽象类)是否定义了一个构造器,而不是依赖于默认构造器。

(18) Missing Switch Default(缺少 switch 默认分支)

检查 switch 语句是否含有 default 子句。

(19) Modified Control Variable(修改控制变量)

检查确保 for 循环的控制变量没有在 for 代码块中被修改。

(20) Multiple String Literals(多重字符串常量)

检查在单个文件中,相同的字符串常量是否出现了多次。

(21) Multiple Variable Declaration(多重变量声明)

检查每个变量是否使用一行一条语句进行声明。

(22) Nested For Depth(for 嵌套深度)

限制 for 循环的嵌套层数(默认值为 1)。

(23) Nested If Depth(if 嵌套深度)

限制 if-else 代码块的嵌套层数(默认值为 1)。

(24) Nested Try Depth(try 嵌套深度)

限制 try 代码块的嵌套层数(默认值为 1)。

(25) No Clone(没有 clone 方法)

检查是否覆盖了 Object 类中的 clone()方法。

(26) No Finalizer(没有 finalize 方法)

验证类中是否定义了 finalize()方法。

(27) Package Declaration(包声明)

确保一个类具有一个包声明,并且(可选地)包名要与源代码文件所在的目录名相匹配。

(28) Parameter Assignment(参数赋值)

不允许对参数进行赋值。

(29) Redundant Throws(多余的 throws)

检查 throws 子句中是否声明了多余的异常,例如重复异常、未检查的异常或一个已声明抛出的异常的子类。

(30) Require This(需要 this)

检查代码中是否使用了 this.,也就是说,在默认情况下,引用当前对象的实例变量和方法时,应当显式地通过 this.varName 或 this.methodName(args)这种形式进行调用。

(31) Return Count(return 总数)

限制 return 语句的数量,默认值为 2。可以忽略检查指定的方法(默认忽略 equals()方法)。

(32) Simplify Boolean Expression(简化布尔表达式)

检查是否有过于复杂的布尔表达式。现在能够发现诸如 if (b == true)、b || true、!false 等类型的代码。

(33) Simplify Boolean Return(简化布尔返回值)

检查是否有过于复杂的布尔类型 return 语句。

(34) String Literal Equality(严格的常量等式比较)

检查字符串对象的比较是否使用了==或!=运算符。

(35) SuperClone(父类 clone 方法)

检查一个覆盖的 clone()方法是否调用了 super.clone()方法。

(36) SuperFinalize(父类 finalize 方法)

检查一个覆盖的 finalize()方法是否调用了 super.finalize()方法。参考:清理未使用对象。

(37) Trailing Array Comma(数组尾随逗号)

检查数组的初始化是否包含一个尾随逗号。例如:

```
int[] a = new int[] {
            1,   2,   3, };
```

如果左花括号和右花括号都位于同一行,那么这项检查允许不添加尾随逗号。如:

```
return new int[] { 0 };
```

(38) Unnecessary Parentheses(不必要的圆括号)

检查代码中是否使用了不必要的圆括号。

(39) One Statement Per Line(每行一条语句)

检查每行是否只有一条语句。下面的一行将会被标识为出错。

```
x = 1; y = 2; //一行中有两条语句.
```

12) Class Design

(1) Designed For Extension(设计扩展性)

检查类是否具有可扩展性。更准确地说,它强制使用一种编程风格,父类必须提供空的"句柄",以便子类实现它们。确切的规则是,类中可以由子类继承的非私有、非静态方法必须是:abstract 方法,或 final 方法,或有一个空的实现。

(2) Final Class(final 类)

检查一个只有私有构造器的类是否被声明为 final。

(3) Inner Type Last(最后声明内部类型)

检查嵌套/内部的类型是否在当前类的最底部声明(在所有的方法/字段的声明之后)。

(4) Hide Utility Class Constructor(隐藏工具类构造器)

确保工具类(在 API 中只有静态方法和字段的类)没有任何公有构造器。

(5) Interface Is Type(接口是类型)

Bloch 编写的 *Effective Java* 中提到,接口应当描述为一个类型。因此,定义一个只包含常量,但是没有包含任何方法的接口是不合适的。

(6) Mutable Exception(可变异常)

确保异常(异常类的名称必须匹配指定的正则表达式)是不可变的。

(7) Throws Count(抛出计数)

将异常抛出语句的数量配置为一个指定的限值(默认值为 1)。

(8) Visibility Modifier(可见性标识符)

检查类成员的可见性。只有 static final 的类成员可以是公有的,其他的类成员必须是私有的,除非设置了 protectedAllowed 属性或 packageAllowed 属性。

13) Duplicates

Strict Duplicate Code(严格重复代码)

逐行地比较所有的代码行,如果有若干行只有缩进有所不同,那么就报告存在重复代码。Java 代码中的所有的 import 语句都会被忽略,任何其他的行(包括 Javadoc、方法之间的空白行等)都会被检查。

14) Metrics

(1) Boolean Expression Complexity(布尔表达式复杂度)

限制一个表达式中的 &&、||、&、|、^等逻辑运算符的数量。

(2) Class Data Abstraction Coupling(类的数据抽象耦合)

这项度量会测量给定类中的其他类的实例化操作的次数。

(3) Class Fan Out Complexity(类的扇出复杂度)

一个给定类所依赖的其他类的数量。这个数量的平方还可以用于表示函数式程序(基

于文件)中需要维护总量的最小值。

(4) Cyclomatic Complexity(循环复杂度)

检查循环复杂度是否超出了指定的限值。该复杂度由构造器、方法、静态初始化程序、实例初始化程序中的 if、while、do、for、?:、catch、switch、case 等语句,以及 && 和||运算符的数量所测量。它是遍历代码的可能路径的一个最小数量测量,因此也是需要的测试用例的数量。通常 1～4 是很好的结果,5～7 较好,8～10 就需要考虑重构代码了,如果大于 11,则需要马上重构代码。

(5) Non Commenting Source Statements(非注释源码语句)

通过对非注释源码语句(NCSS)进行计数,确定方法、类、文件的复杂度。这项检查遵守 Chr. Clemens Lee 编写的 JavaNCSS-Tool 中的规范。

(6) NPath Complexity(NPath 复杂度)

NPath 度量会计算遍历一个函数时,所有可能的执行路径的数量。它会考虑嵌套的条件语句,以及由多部分组成的布尔表达式(例如,A && B,C || D,等等)。

解释:在 Nejmeh 的团队中,每个单独的例程都有一个取值为 200 的非正式的 NPath 限值;超过这个限值的函数可能会进行进一步的分解,或者至少一探究竟。

15) Miscellaneous

(1) Array Type Style(数组类型风格)

检查数组定义的风格。有的开发者使用 Java 风格:public static void main(String[] args);有的开发者使用 C 风格:public static void main(String args[])。

(2) Descendent Token Check(后续标记检查)

检查在其他标记之下的受限标记。警告:这是一项非常强大和灵活的检查,但是与此同时,它偏向于底层技术,并且非常依赖于具体实现,因为,它的结果依赖于我们用来构建抽象语法树的语法。

(3) Final Parameters(final 参数)

检查方法/构造器的参数是否是 final 的。

(4) Indentation(代码缩进)

检查 Java 代码的缩进是否正确。

(5) New Line At End Of File(文件末尾的新行)

检查文件是否以新行结束。

(6) Todo Comment(TODO 注释)

这项检查负责 TODO 注释的检查。

(7) Translation(语言转换)

这是一项 FileSetCheck 检查,通过检查关键字的一致性属性文件,它可以确保代码的语言转换的正确性。可以使用两个描述同一个上下文环境的属性文件来保证一致性,如果它们包含相同的关键字。

(8) Uncommented Main(未注释 main 方法)

检查源码中是否有未注释的 main()方法(调试的残留物)。

(9) Upper Ell(大写 L)

检查 long 类型的常量在定义时是否由大写的 L 开头。注意,是 L,不是 l。

(10) Regexp(正则表达式)

这项检查可以确保指定的格式串在文件中存在,或者允许出现几次,或者不存在。

(11) Outer Type File Name(外部类型文件名)

检查外部类型名称是否与文件名称匹配。例如,类 Foo 必须在文件 Foo.java 中。

16) Other

(1) Checker(检查器)

每个 Checkstyle 配置的根模块,不能被删除。

(2) TreeWalker(树遍历器)

FileSetCheck TreeWalker 会检查单个的 Java 源码文件,并且定义了适用于检查这种文件的属性。

17) Filters

(1) Severity Match Filter(严重度匹配过滤器)

Severity Match Filter 过滤器会根据事件的严重级别决定是否要接受审计事件。

(2) Suppression Filter(抑制过滤器)

在检查错误时,SuppressionFilter 过滤器会依照一个 XML 格式的策略抑制文件,选择性地拒绝一些审计事件。

(3) Suppression Comment Filter(抑制注释过滤器)

Suppression Comment Filter 过滤器使用配对的注释来抑制审计事件。

(4) Suppress With Nearby Comment Filter(抑制附近注释过滤器)

Suppress With Nearby Comment Filter 过滤器使用独立的注释来抑制审计事件。

3. Checkstyle 规则文件示例

Checkstyle 自带了几个配置文件,如 sun_checks.xml、sun_checks_eclispse.xml、google_checks.xml 等。sun_checks.xml 是严格符合 Sun 编码规范的。只是这些配置文件的检查太过严格,任何一个项目都会检查出上千个 Warning 来。

用户可以根据自己的需要来撰写配置文件。

下面是一个 Checkstyle 配置文件示例。

```
<?xml version="1.0" encoding="UTF-8"?>
 <!DOCTYPE module PUBLIC
           "-//Puppy Crawl//DTD Check Configuration 1.2//EN"
           "http://www.puppycrawl.com/dtds/configuration_1_2.dtd">
<module name="Checker">
          <property name="severity" value="warning"/>
     <module name="StrictDuplicateCode">
         <property name="charset" value="utf-8" />
     </module>

     <module name="TreeWalker">
         <!-- javadoc 的检查 -->
         <!-- 检查所有的 interface 和 class -->
         <module name="JavadocType" />
```

```
<!-- 命名方面的检查 -->
<!-- 局部的 final 变量,包括 catch 中的参数的检查 -->
<module name = "LocalFinalVariableName" />
<!-- 局部的非 final 型的变量,包括 catch 中的参数的检查 -->
<module name = "LocalVariableName" />
<!-- 包名的检查(只允许小写字母) -->
<module name = "PackageName">
    <property name = "format" value = "^[a - z] + ( \.[a - z][a - z0 - 9] * ) * $ " />
</module>
<!-- 仅仅是 static 型的变量(不包括 static final 型)的检查 -->
<module name = "StaticVariableName" />
<!-- 类型(Class 或 Interface)名的检查 -->
<module name = "TypeName" />
<!-- 非 static 型变量的检查 -->
<module name = "MemberName" />
<!-- 方法名的检查 -->
<module name = "MethodName" />
<!-- 方法的参数名 -->
<module name = "ParameterName " />
<!-- 常量名的检查 -->
<module name = "ConstantName" />
<!-- 没用的 import 检查 -->
<module name = "UnusedImports" />

<!-- 长度方面的检查 -->
<!-- 文件长度不超过 1500 行 -->
<module name = "FileLength">
    <property name = "max" value = "1500" />
</module>
<!-- 每行不超过 150 个字 -->
<module name = "LineLength">
    <property name = "max" value = "150" />
</module>
<!-- 方法不超过 150 行 -->
<module name = "MethodLength">
    <property name = "tokens" value = "METHOD_DEF" />
    <property name = "max" value = "150" />
</module>
<!-- 方法的参数个数不超过 5 个。并且不对构造方法进行检查 -->
<module name = "ParameterNumber">
    <property name = "max" value = "5" />
    <property name = "tokens" value = "METHOD_DEF" />
</module>

<!-- 空格检查 -->
<!-- 允许方法名后紧跟左边圆括号"(" -->
<module name = "MethodParamPad" />
<!-- 在类型转换时,不允许左圆括号右边有空格,也不允许与右圆括号左边有空格 -->
<module name = "TypecastParenPad" />
```

```
        <!-- 关键字 -->
        <!--
          每个关键字都有正确的出现顺序。
          比如 public static final XXX 是对一个常量的声明。如果使用 static public final
就是错误的。
         -->
        <module name="ModifierOrder" />
        <!-- 多余的关键字 -->
        <module name="RedundantModifier" />

        <!-- 对区域的检查 -->
        <!-- 不能出现空白区域 -->
        <module name="EmptyBlock" />
        <!-- 所有区域都要使用大括号 -->
        <module name="NeedBraces" />
        <!-- 多余的括号 -->
        <module name="AvoidNestedBlocks">
            <property name="allowInSwitchCase" value="true" />
        </module>

        <!-- 编码方面的检查 -->
        <!-- 不许出现空语句 -->
        <module name="EmptyStatement" />
        <!-- 不允许魔法数 -->
        <module name="MagicNumber">
            <property name="tokens" value="NUM_DOUBLE, NUM_INT" />
        </module>
        <!-- 多余的 throw -->
        <module name="RedundantThrows" />
        <!-- String 的比较不能用!= 和 == -->
        <module name="StringLiteralEquality" />
        <!-- if 最多嵌套三层 -->
        <module name="NestedIfDepth">
            <property name="max" value="3" />
        </module>
        <!-- try 最多被嵌套两层 -->
        <module name="NestedTryDepth">
            <property name="max" value="2" />
        </module>
        <!-- clone 方法必须调用了 super.clone() -->
        <module name="SuperClone" />
        <!-- finalize 必须调用了 super.finalize() -->
        <module name="SuperFinalize" />
        <!-- 不能 catch java.lang.Exception -->
        <module name="IllegalCatch">
            <property name="illegalClassNames" value="java.lang.Exception" />
        </module>
        <!-- 确保一个类有 package 声明 -->
        <module name="PackageDeclaration" />
        <!-- 一个方法中最多有三个 return -->
        <module name="ReturnCount">
            <property name="max" value="3" />
```

```
            <property name="format" value="^$" />
        </module>
        <!--
          根据 Sun 编码规范,class 或 interface 中的顺序如下:
          1.class 声明。2.变量声明。3.构造函数 4.方法
         -->
        <module name="DeclarationOrder" />
        <!-- 同一行不能有多个声明 -->
        <module name="MultipleVariableDeclarations" />
        <!-- 不必要的圆括号 -->
        <module name="UnnecessaryParentheses" />

        <!-- 杂项 -->
        <!-- 禁止使用 System.out.println -->
        <module name="GenericIllegalRegexp">
            <property name="format" value="System\.out\.println" />
            <property name="ignoreComments" value="true" />
        </module>
        <!-- 检查并确保所有的常量中的 L 都是大写的。因为小写的字母 l 跟数字 1 太像了 -->
        <module name="UpperEll" />
        <!-- 检查数组类型的定义是 String[] args,而不是 String args[] -->
        <module name="ArrayTypeStyle" />
        <!-- 检查 Java 代码的缩进 默认配置:
             基本缩进 4 个空格,新行的大括号: 0。新行的 case 4 个空格
         -->
        <module name="Indentation" />
    </module>
</module>
```

3.2.3 Checkstyle 的安装

Checkstyle 可以在 Eclipse 中直接通过网络更新,其安装步骤如下。

(1) 启动 Eclipse。

(2) 在菜单中单击 Help→Install New Software,将弹出 Install 对话框,单击 Add 按钮,将弹出 Add Repository 对话框。

(3) 在 Add Repository 对话框的 Name 文本框中输入 Checkstyle,在 Location 文本框中输入“http://eclipse-cs.sourceforge.net/update”,如图 3-1 所示。

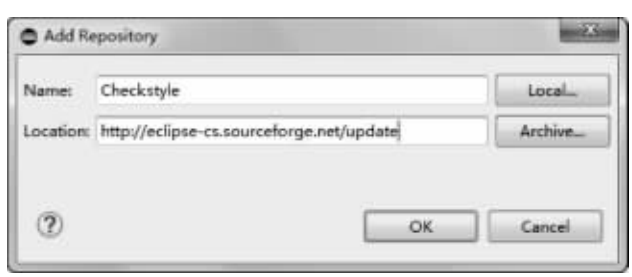

图 3-1 Add Repository 对话框

(4) 单击 OK 按钮,将打开 Available Software 窗口,如图 3-2 所示。

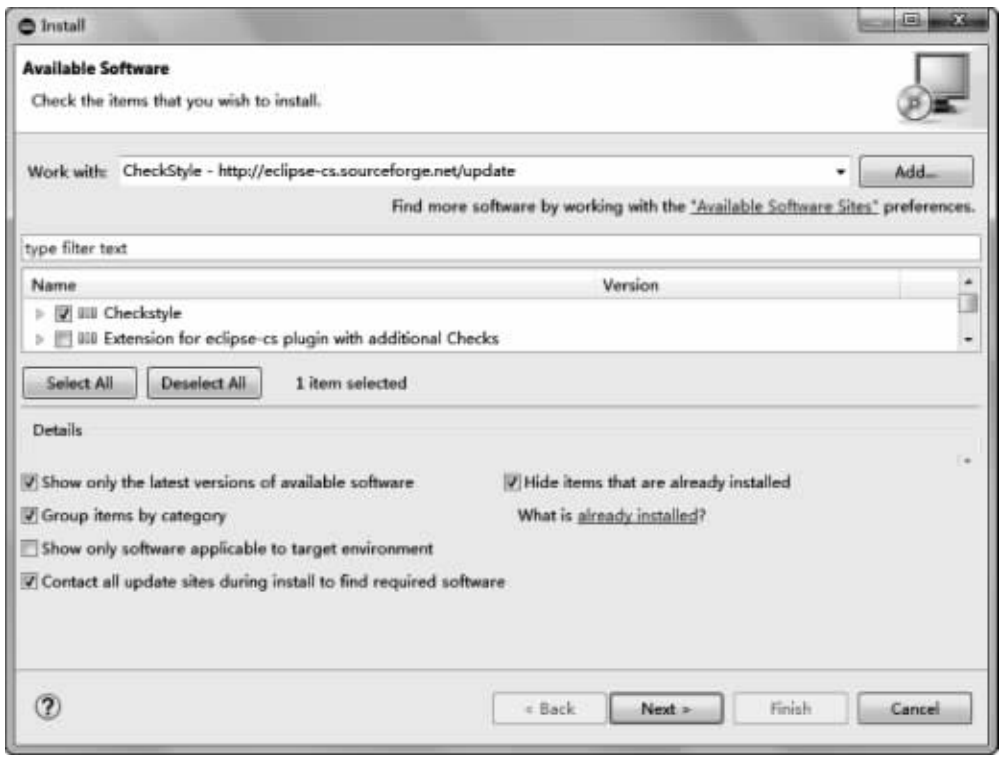

图 3-2 Available Software 窗口

选择 Checkstyle 复选框,然后单击 Next 按钮。按照提示依次完成后面的安装步骤。安装软件需要一定时间,请耐心等待。

(5) 安装完成后,重新启动 Eclipse,单击菜单栏中的 Windows→Preferences,在弹出的窗口中将看到 Checkstyle 选项。

Checkstyle 的下载地址为:http://sourceforge.net/projects/checkstyle/files/checkstyle/。

Checkstyle 的官方网址为:http://checkstyle.sourceforge.net。

3.2.4 Checkstyle 的应用

1. 设置规范文件

启动 Eclipse,单击菜单栏中的 Windows→Preferences,在弹出的窗口中将看到 Checkstyle 选项。单击 Checkstyle,在右侧窗格中将显示 Checkstyle 的相关信息,如图 3-3 所示。

单击 New 按钮,将打开 Check Configuration Properties 对话框,如图 3-4 所示。

首先选择文件类型,其中包括 Internal Configuration、External Configuration File、Remote Configuration 和 Project Relative Configuration 4 种类型。

Internal Configuration:内建于 Eclipse 的 workspace 中,位于 C:/Eclipse/plugins/net.sf.eclipsecs.core 目录下(本例中是将 Eclipse 放在 C 盘根目录下的),无法在项目目录

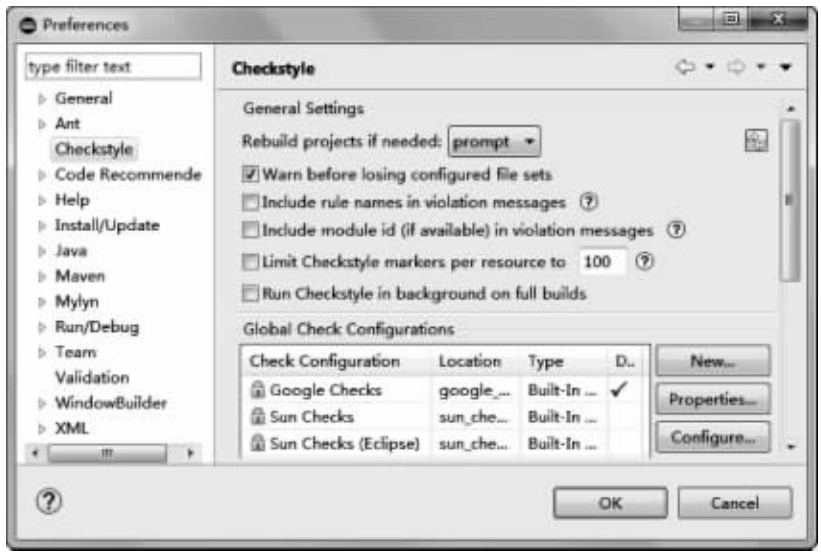

图 3-3 设置规范文件

图 3-4 Check Configuration Properties 对话框

中看到。文件内容可从已有的规范文件中导入。单击右下的 Import 按钮，找到相应的规范文件即可，此时是将外部配置文件复制到 C:/Eclipse/plugins/ plugins/ net. sf. eclipsecs. core 目录下，重新命名。

External Configuration File：直接在项目中引用外部代码规范文件，并可以通过对规范进行配置来修改外部文件，适合团队协作开发。选择 Protect Checkstyle Configuration File 选项，以防止源文件被改写。

Remote Configuration：连接到远程代码规范文件，需要提供地址，用户名和密码。选择 Cache Configuration File 选项，对远程文件进行缓存处理。此文件的配置不可修改，否

则经过配置后会修改原规范文件，删除掉原规范文件的所有注释。

Project Relative Configuration：当代码规范配置文件已经存在于 workspace 中的项目里时，适合于使用此选项。此处可以在确定配置类型后，直接从已有的规范文件中导入，单击 Import 按钮，找到相应的文件即可。

2. 代码规范配置选项

在代码规范配置中，不同的逻辑内容被划分为不同的 module，每个 module 下面有不同的子项目，如图 3-5 所示。

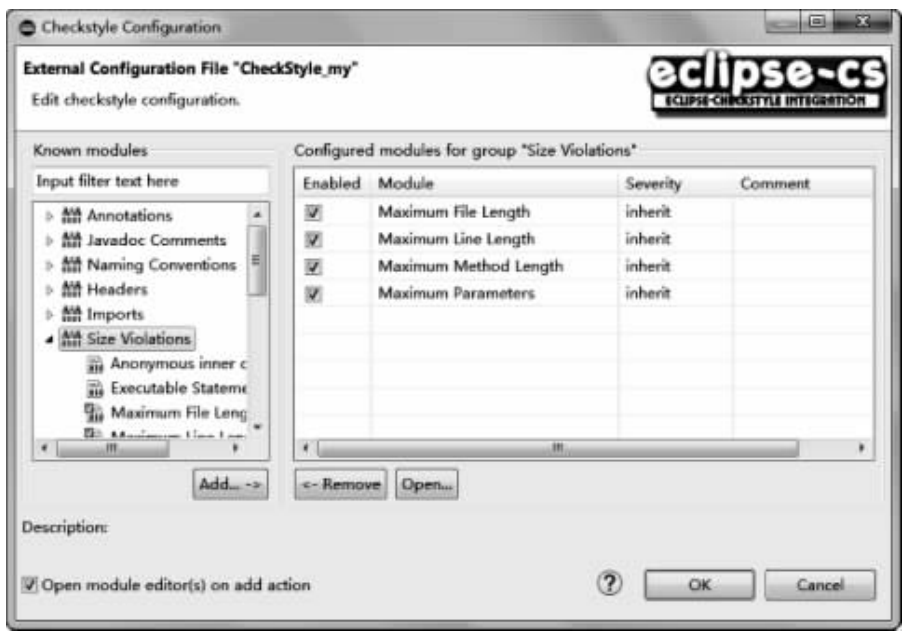

图 3-5 Checkstyle Configuration 窗口

从左侧条目数中，选择一个配置选项，单击 Add 按钮，弹出 New module 对话框，如图 3-6 所示。

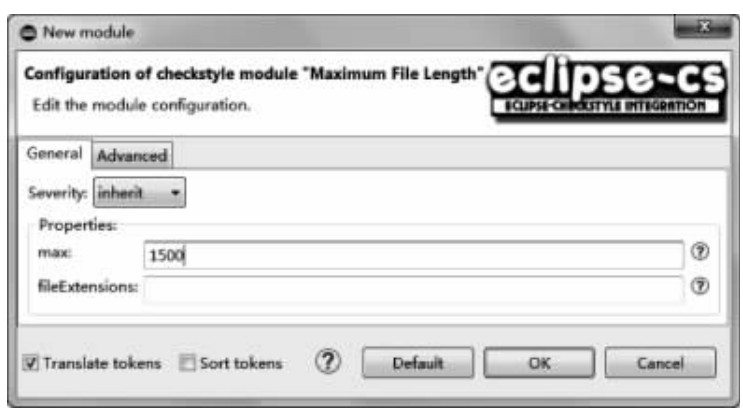

图 3-6 Module 配置示意图(a)

【注意】 系统中自带的规则文件不能配置，只有用户引入的或新建的才能配置。

在 New module 配置对话框中的 General 选项卡里，Severity 表示所出现的问题的严重性，选项值有 inherit、ignore、info、warning、error 5 个不同的等级。在下面的 Properties(属性)栏里，不同的 module 具有不同的属性。

Advanced 选项卡如图 3-7 所示。在 Advanced 选项卡中，Comment 中的内容为对该规范的说明信息。Id 属性用于定义同一个检查类型的不同实例，可以定义不同的检查条件。Custom check messages 是自定义的检查信息，即在发现代码不符合规范时，出现在 Problems 选项卡中 warnings 下的信息，以及将鼠标悬停在代码区域右侧的小放大镜上时出现的信息。底部的两个选项 Translate tokens 和 Sort tokens 默认选中。

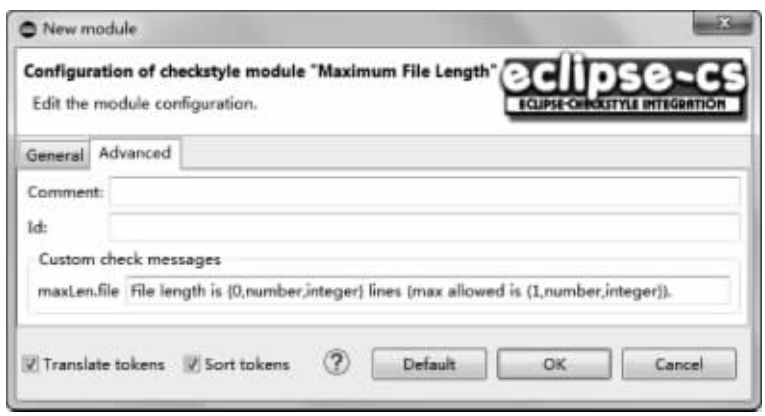

图 3-7 Module 配置示意图(b)

配置完毕后即可添加到右侧的 module 集合中，并且在配置的条目图标上会有小的对勾标识。

上面所有的配置，其实都可以导出为一个 XML 文件。该文件中保存了所有经过配置的 module 信息，方便配置文件的导入导出。

3. 执行规范检查

配置好代码检查规范后，即可使用 Checkstyle 进行检查。

右键单击要进行代码规范检查的项目，选择 Checkstyle 之后会出现子菜单，如图 3-8 所示。

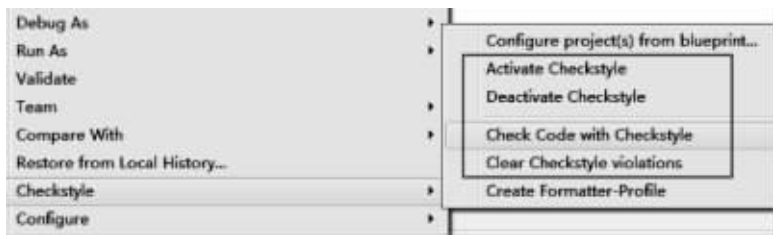

图 3-8 Checkstyle 弹出菜单

Activate Checkstyle：激活 Checkstyle。激活之后，就是开始动态检测。比如输入一行代码之后，这行代码如果不符合之前定义的规则，这行代码就会变成红色，并且会提示当前代码是什么问题。

Deactivate Checkstyle：不启用 Checkstyle 检查。

Check Code with Checkstyle：检测代码是否符合 Checkstyle 的规则。

Clear Checkstyle violations：清空当前所有的检测结果。

选择 Check Code with Checkstyle 菜单项，即可完成代码检查。检查后，Checkstyle 对有问题的代码会使用警告或错误标识，如图 3-9 所示。在编辑窗格下面的 Problems 选项卡中，可以看到问题的详细描述信息。

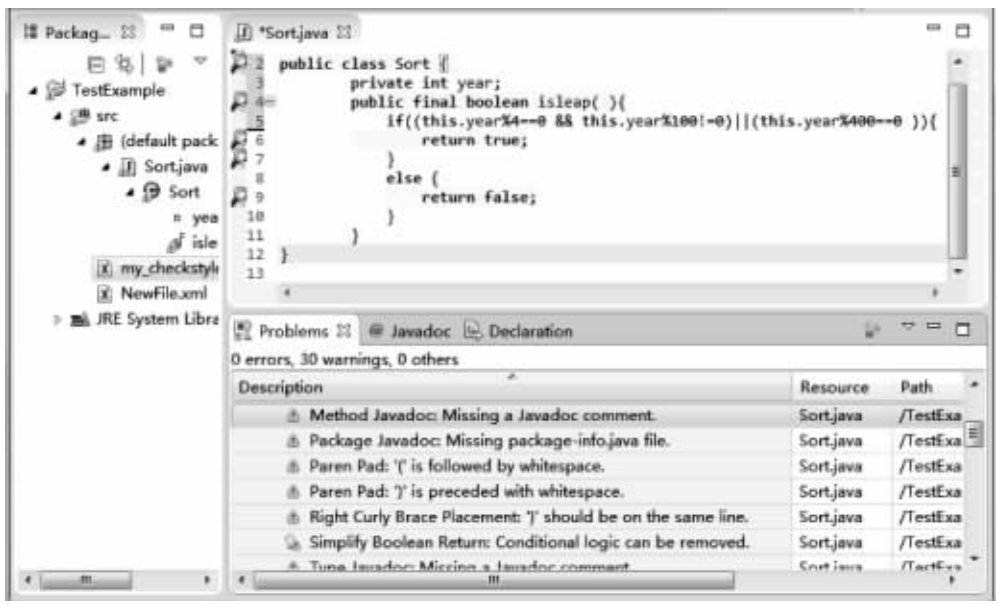

图 3-9　Checkstyle 检查信息

如果代码有问题，在左侧会显示小圆圈标记。将鼠标移动到小圆圈上面时将给出提示信息，如图 3-10 所示。

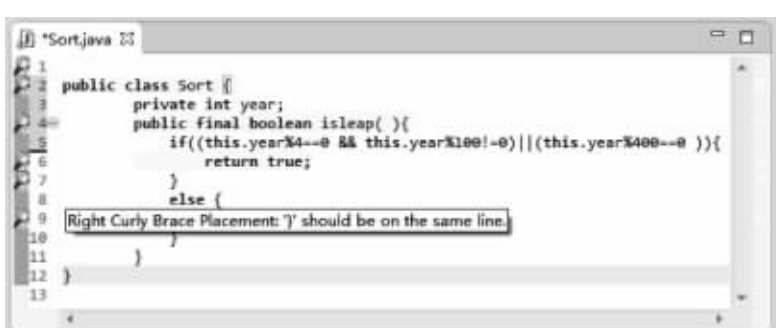

图 3-10　查看错误信息

4. Checkstyle 常见的错误提示

表 3-1 中列举了一些常见的错误提示及解决办法。

表 3-1 Checkstyle 错误提示

序号	Checkstyle 错误提示信息	说 明	解 决 办 法
1	Type is missing a javadoc commentClass	缺少类型说明	增加 javadoc 说明
2	“{” should be on the previous line	“{”应该位于前一行	把“{”放到上一行
3	Methods is missing a javadoc comment	方法前面缺少 javadoc 注释	添加 javadoc 注释
4	Expected @ throws tag for “Exception”	在注释中希望有@throws 的说明	在方法前的注释中添加这样一行：* @ throws Exception if has error(异常说明)
5	“.”is preceded with whitespace	“.”前面不能有空格	把“.”前面的空格去掉
6	“.”is followed by whitespace	“.”后面不能有空格	把“.”后面的空格去掉
7	“=” is not preceded with whitespace	“=”前面缺少空格	在“=”前面加个空格
8	“=” is not followed with whitespace	“=”后面缺少空格	在“=”后面加个空格
9	“}” should be on the same line	“}”应该与下条语句位于同一行	把“}”放到下一行的前面
10	Unused @param tag for “unused”	没有参数“unused”,不需注释	“* @param unused parameter additional(参数名称)”把这行 unused 参数的注释去掉“
11	Variable “CA” missing javadoc	变量“CA”缺少 javadoc 注释	在“CA“变量前添加 javadoc 注释：/** CA. */
12	Line longer than 80 characters	行长度超过 80	把它分成多行写
13	Line contains a tab character	行含有“tab”字符	删除 tab
14	Redundant “public” modifier	冗余的“public” modifier	删除冗余的 public
15	Final modifier out of order with the JSL suggestion	Final modifier 的顺序错误	调整其顺序
16	Avoid using the “.*” form of import	import 格式避免使用“.*”	
17	Redundant import from the same package	从同一个包中 import 内容	
18	Unused import-java.util.list	import 进来的 java.util.list 没有被使用	去掉导入的多余的类
19	Duplicate import to line 13	重复 import 同一个内容	去掉导入的多余的类
20	Import from illegal package	从非法包中 import 内容	
21	“while” construct must use “{}”	“while”语句缺少“{}”	给 while 循环体加上“{}”
22	Variable “ABC” must match pattern “^[a-z][a-zA-Z0-9]*$”	变量“ABC”不符合命名规则“^[a-z][a-zA-Z0-9]*$”	把这个命名改成符合规则的命名“aBC”
23	“(” is followed by whitespace	“(”后面不能有空格	把“(”后面的空格去掉

续表

序号	Checkstyle 错误提示信息	说　明	解 决 办 法
24	")" is proceeded by whitespace	")"前面不能有空格	把")"前面的空格去掉
25	Line matches the illegal pattern 'X'	含有非法字符	修改非法字符
26	Line has trailing spaces	多余的空行	删除这行空行
27	Must have at least one statement	至少有一个声明	try{}catch(){}中的异常捕捉里面不能为空，在异常里面加上语句
28	Switch without "default" clause	switch 语句判断没有 default 的情况处理	在 switch 中添加 default 语句
29	Redundant throws: 'NameNotFoundException' is subclass of 'NamingException'	'NameNotFoundException' 是'NamingException'的子类重复抛出异常	如果抛出两个异常，一个异常类是另一个的子类，那么只需要写父类
30	Parameter docType should be final	参数 docType 应该为 final 类型	在参数 docType 前面加个 final
31	Expected @param tag for 'dataManager'	缺少 dataManager 参数的注释	在注释中添加 @ param dataManager DataManager

3.3 FindBugs

3.3.1 FindBugs 简介

FindBugs 是由马里兰大学提供的一款开源 Java 静态代码分析工具。FindBugs 通过检查类文件或 JAR 文件，将字节码与一组缺陷模式进行对比从而发现代码缺陷，完成静态代码分析。FindBugs 既提供可视化 UI 界面，同时也可以作为插件使用。使用 FindBugs 有很多种方式，从 GUI、从命令行、使用 Ant、作为 Eclipse 插件程序和使用 Maven，甚至作为 Hudson 持续集成的插件。

FindBugs 可以简单高效全面地发现程序代码中存在的 Bug，Bad Smell，以及潜在隐患。针对各种问题，提供了简单的修改意见供我们重构时进行参考。通过使用 FindBugs，可以一定程度上降低 Code Review 的工作量，并且会提高 Review 效率。

FindBugs 自己定义了一系列的检测器，1.3.9 版本的检测器有 83 种 Bad practice(不好的习惯)，133 种 Correctness(正确性)，两种 Experimental(实验性问题)，一种 Internationalization(国际化问题)，12 种 Malicious code vulnerability(恶意的代码)，41 种 Multithreaded correctness(线程问题)，27 种 Performance(性能问题)，9 种 Security(安全性问题)，62 种 Dodgy(狡猾的问题)。

3.3.2 FindBugs 的安装

单击 Eclipse 菜单栏上的 Help→Eclipse Marketplace，将打开 Eclipse Marketplace 窗口，如图 3-11 所示。在 Find 输入框中输入"FindBug"，并按回车键。Eclipse 将搜索出：

FindBugs Eclipse Plugin 3.0.1。

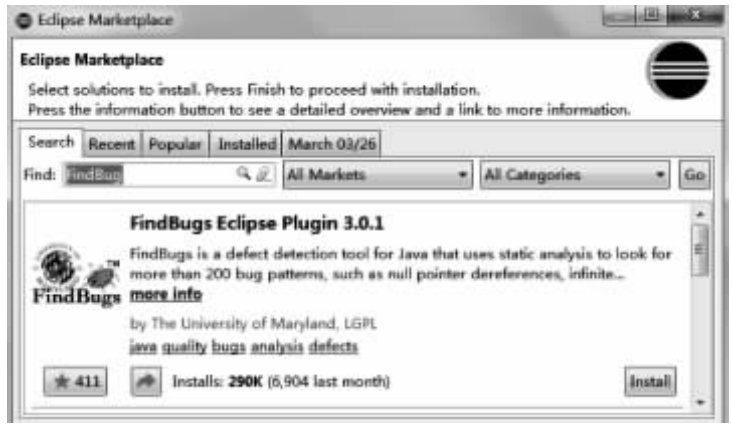

图 3-11 Eclipse Marketplace 窗口

单击 Install 按钮，Eclipse 将弹出 Confirm Selected Features 对话框，如图 3-12 所示。

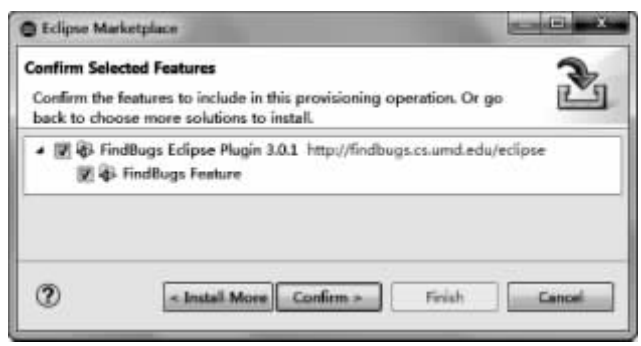

图 3-12 Confirm Selected Features 对话框

确认 FindBugs Eclipse Plugin 3.0.1 http://findbugs.cs.umd.edu/eclipse 复选框已经选中，然后单击 Confirm 按钮，Eclipse 将准备安装 FindBugs 插件。按照 Eclipse 的提示进行相应的操作，即可完成安装。

3.3.3 FindBugs 的使用

下面简要介绍 Eclipse 里面使用 FindBugs 进行简单测试的例子。

首先，创建练习工程 FindBugsTest，然后创建测试类 NextDateFrame。待测试代码如下。

```
import java.awt.*;
import java.awt.event.*;
import java.lang.Character;

public class NextDateFrame extends WindowAdapter implements ActionListener {
      Frame frame;
      Label lab0, lab1, lab2, lab3,lab4;
```

```
TextField text1, text2, text3,text4;
Button b1,b2;
Dialog dlg1 = new Dialog(frame, "输入的日期无效",true);
Dialog dlg2 = new Dialog(frame, "输入不能为空",true);
Dialog dlg3 = new Dialog(frame, "输入非数字的字符",true);
FlowLayout flayout;
NextDate today;
…
```

这个类里面有错误，以便测试用。代码写好之后，在类名上单击鼠标右键，将弹出右键菜单，如图 3-13 所示。

图 3-13　Find Bugs 右键菜单

选择 Find Bugs→Find Bugs 菜单项，FindBugs 将进行静态测试。如果代码中有缺陷，测试完成后，将在编辑框中有错误的代码行上显示 Bug 图标(臭虫标志)，如图 3-14 所示。

图 3-14　执行 FindBugs 检查

不同严重级别的 Bug，图标的颜色不同。Bug 图标的颜色有三种：黑色、红色和橘黄色。黑色的臭虫标志是分类。红色的臭虫表示严重 Bug，发现后必须修改代码。橘黄色的臭虫表示潜在警告性 Bug，应尽量修改。

用鼠标双击代码左侧的 Bug 图标，将在编辑窗口的下面显示 Bug 的详细信息，如图 3-15 所示。

图 3-15　Bug 详细信息

根据详细的信息，可以看到 FindBugs 对代码报告的错误信息，及相应的处理办法，根据它的提示，可以快速方便地进行代码修改。如果双击问题，系统会自动跳转到相对应的问题所在行。

打开 Bugs Explore，将看到所查出的 Bug 层次结构，如图 3-16 所示。

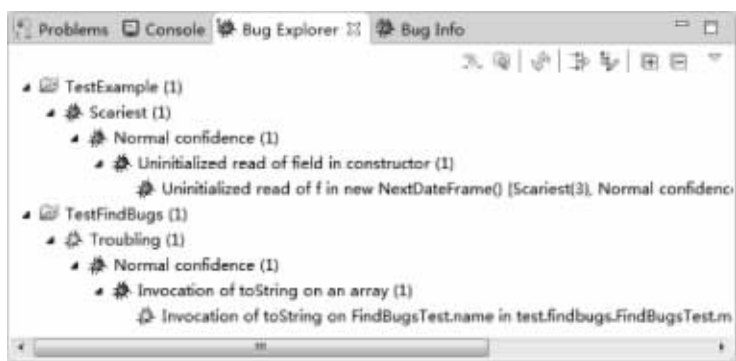

图 3-16　Bug Explorer

3.3.4　配置 FindBugs

如果执行 Find Bugs 菜单命令时，没有发现任何 Bug，可能是没有启动 FindBugs 检查。单击菜单 Project→Properties，将打开项目属性设置，如图 3-17 所示。选择 Enable project specific settings 和 Run automatically 复选框，然后单击 OK 按钮。重新执行 Find Bugs 菜单命令，即可启动 FindBugs 检查。

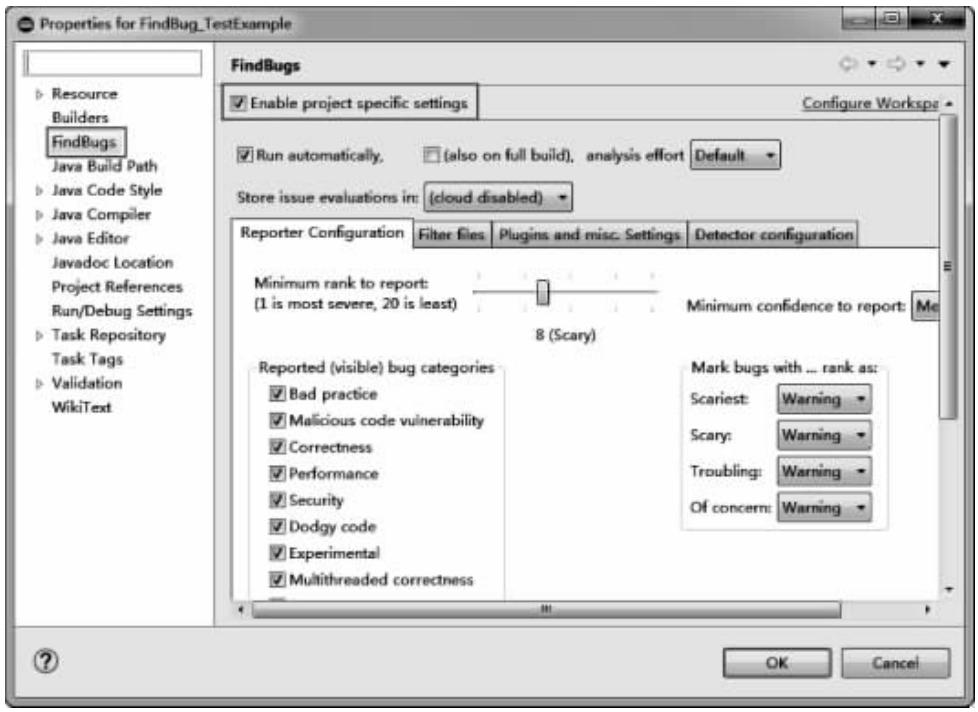

图 3-17 项目属性设置

下面介绍各设置项的内容。

1. Reported(visible) bug categories

Reporter Configuration 选项卡下的 Reported(visible) bug categories 有下列选项。

(1) Bad practice：关于代码实现中的一些不好的习惯。

(2) Malicious code vulnerability：关于恶意破坏代码相关方面的。

(3) Correctness：关于代码正确性相关方面的。

(4) Performance：关于代码性能相关方面的。

(5) Security：关于代码安全性防护的。

(6) Dodgy code：关于代码运行期安全方面的。

(7) Experimental：关于实验性问题的。

(8) Multithreaded correctness：关于代码多线程正确性相关方面的。

(9) Internationalization：关于代码国际化相关方面的。

例如，如果把 Performance 的检查项去掉(不选中它)，那么与 Performance 分类相关的警告信息就不会显示了。其他的与此类似。

2. Run automatically

当此项选中后，FindBugs 将会在用户修改 Java 类时自动运行。如果设置了 Eclipse 自动编译开关后，修改完 Java 文件并保存，FindBugs 就会运行，并将相应的信息显示出来。

当此项没有选中，只能每次在需要的时候自己去运行 FindBugs 来检查代码。

3. Minimum confidence to report

Reporter Configuration 选项卡下的 Minimum confidence to report 选择项是让用户选择哪个级别的信息进行显示，有 Low、Medium、High 三个选择项可以选择。

选择 High 选项时，只有 High 级别的提示信息才会被显示。

选择 Medium 选项时，只有 Medium 和 High 级别的提示信息才会被显示。默认情况下，选择的是 Medium。

选择 Low 选项时，所有级别的提示信息都会被显示。

4. Detector configuration

在这里可以选择所要进行检查的相关的 Bug Pattern 条目。

可以从 Bug codes、Detector name、Detector description 中看到相应的要检查的内容，可以根据需要选择或去掉相应的检查条件。

5. FindBugs 检测器

1) Bad practice 坏的实践

一些不好的实践，下面列举几个。

HE：类定义了 equals()，却没有 hashCode()；或类定义了 equals()，却使用 Object.hashCode()；或类定义了 hashCode()，却没有 equals()；或类定义了 hashCode()，却使用 Object.equals()；类继承了 equals()，却使用 Object.hashCode()。

SQL：Statement 的 execute 方法调用了非常量的字符串；或 Prepared Statement 是由一个非常量的字符串产生。

DE：方法终止或不处理异常，一般情况下，异常应该被处理或报告，或被方法抛出。

2) Correctness 一般的正确性问题

可能导致错误的代码，下面列举几个。

NP：空指针被引用；在方法的异常路径里，空指针被引用；方法没有检查参数是否 null；null 值产生并被引用；null 值产生并在方法的异常路径被引用；传给方法一个声明为@NonNull 的 null 参数；方法的返回值声明为@NonNull 而实际是 null。

Nm：类定义了 hashcode()方法，但实际上并未覆盖父类 Object 的 hashCode()；类定义了 tostring()方法，但实际上并未覆盖父类 Object 的 toString()；很明显的方法和构造器混淆；方法名容易混淆。

SQL：方法尝试访问一个 Prepared Statement 的 0 索引；方法尝试访问一个 ResultSet 的 0 索引。

UwF：所有的 write 都把属性置成 null，这样所有的读取都是 null，这样这个属性是否有必要存在；或属性从没有被 write。

Internationalization 国际化：当对字符串使用 upper 或 lowercase 方法，如果是国际的字符串，可能会不恰当的转换。

3) Malicious code vulnerability 可能受到的恶意攻击

如果代码公开，可能受到恶意攻击的代码，下面列举几个。

FI：一个类的 finalize()应该是 protected，而不是 public 的。

MS：属性是可变的数组；属性是可变的 Hashtable；属性应该是 package protected 的。

4) Multithreaded correctness 多线程的正确性

多线程编程时，可能导致错误的代码，下面列举几个。

ESync：空的同步块，很难被正确使用。

MWN：错误使用 notify()，可能导致 IllegalMonitorStateException 异常；或错误地使用 wait()。

No：使用 notify()而不是 notifyAll()，只是唤醒一个线程而不是所有等待的线程。

SC：构造器调用了 Thread.start()，当该类被继承时可能会导致错误。

5) Performance 性能问题

可能导致性能不佳的代码，下面列举几个。

DM：方法调用了低效的 Boolean 的构造器，而应该用 Boolean.valueOf(…)；用类似 Integer.toString(1)代替 new Integer(1).toString()；方法调用了低效的 float 的构造器，应该用静态的 valueOf 方法。

SIC：如果一个内部类想在更广泛的地方被引用，它应该声明为 static。

SS：如果一个实例属性不被读取，考虑声明为 static。

UrF：如果一个属性从没有被 read，考虑从类中去掉。

UuF：如果一个属性从没有被使用，考虑从类中去掉。

6) Dodgy 危险的

具有潜在危险的代码，可能运行期产生错误。下面列举几个。

CI：类声明为 final，但声明了 protected 的属性。

DLS：对一个本地变量赋值，但却没有读取该本地变量；本地变量赋值成 null，却没有读取该本地变量。

ICAST：整型数字相乘结果转化为长整型数字，应该将整型先转化为长整型数字再相乘。

INT：没必要的整型数字比较，如 X <= Integer.MAX_VALUE。

NP：对 readline()的直接引用，而没有判断是否为 null；对方法调用的直接引用，而方法可能返回 null。

REC：直接捕获 Exception，而实际上可能是 RuntimeException。

ST：从实例方法里直接修改类变量，即 static 属性。

3.4 Cppcheck

3.4.1 Cppcheck 简介

Cppcheck 是一个 C/C++代码缺陷静态检查工具，用来检查代码缺陷，如数组越界，内存泄漏等。不同于 C/C++编译器及其他分析工具，Cppcheck 只检查编译器检查不出来的 Bug，不检查语法错误。

Cppcheck 作为编译器的一种补充检查，对产品的源代码执行严格的逻辑检查。

Cppcheck 执行的检查包括以下几种。

(1) Out of bounds checking：边界检查，如数组越界检查。

(2) Memory leaks checking：内存泄漏检查。

(3) Detect possible null pointer dereferences：检查空指针引用。

(4) Check for uninitialized variables：检查未初始化的变量。

(5) Check for invalid usage of STL：异常 STL 函数使用检查。

(6) Checking exception safety：异常处理安全性检查。

(7) Warn if obsolete or unsafe functions are used：过期的函数或不安全的函数调用检查。

(8) Warn about unused or redundant code：未使用的或冗余的代码检查。

(9) Detect various suspicious code indicating bugs：检查代码中可能存在的各种 Bug。

(10) Check for auto variables：自动变量检查。

3.4.2 Cppcheck 的安装

1. 下载 Cppcheck

Cppcheck 是开源项目，可以从官网上获得其源代码，当前版本是 Cppcheck 1.69。

Cppcheck 官方地址：http://cppcheck.sourceforge.net/

2. 安装

双击已下载的 Cppcheck 源文件 cppcheck-1.69-x86-Setup.msi，进入安装界面，如图 3-18 所示。

图 3-18 Cppcheck 安装界面

单击 Next 按钮，按照提示信息进行操作，即可完成安装。

安装完后，双击 cppcheckgui.exe 启动其 GUI 程序，如图 3-19 所示。

工具栏第一个按钮 可以用于添加待检测的目录。

【注意】 Cppcheck 不支持中文路径。

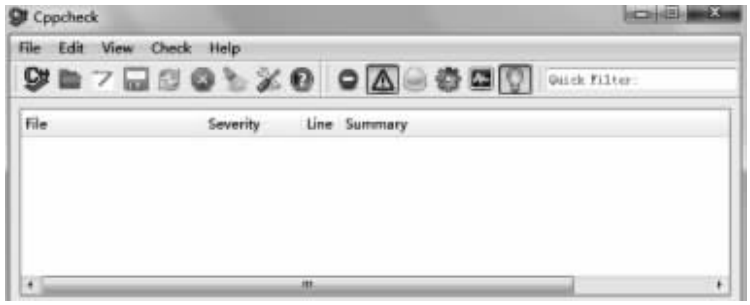

图 3-19 Cppcheck 主窗口

3.4.3 Cppcheck 的使用

1. 准备好待测试的程序(C/C++)

例如,在编译器中写一段程序代码,文件名为 file.c,代码如下所示。

```
int main()
{
    char a[10];
    a[10] = 0;
    return 0;
}
```

2. 新建测试项目

启动 Cppcheck,单击菜单栏中的 File→New Project File,将弹出 Select Project Filename 对话框,在“文件名”编辑框中输入项目文件名称,如 test1.cppcheck,然后单击“保存”按钮,如图 3-20 所示。

图 3-20 Select Project Filename 对话框

接下来，Cppcheck 将弹出项目文件配置对话框，如图 3-21 所示。

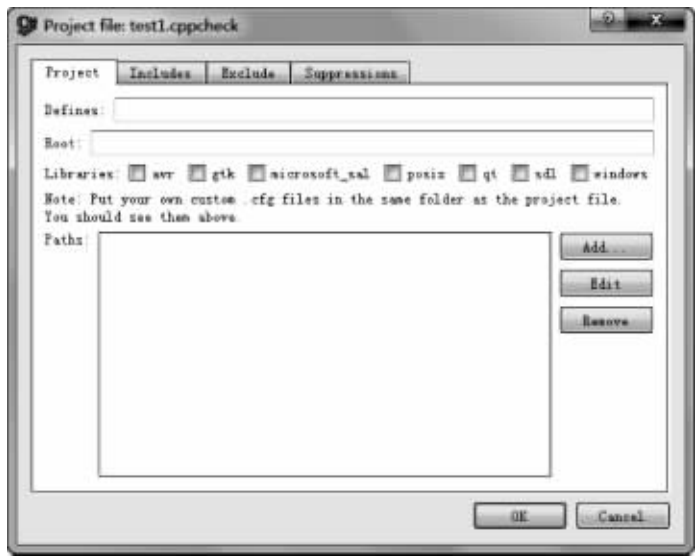

图 3-21　Project file 配置

在 Project 选项卡中，单击 Add 按钮，将弹出 Select a directory to check 对话框，选择待测试文件所在的文件夹。本例选择 file.c 所在的文件夹 CppCheck_test，然后单击“选择文件夹”按钮，此时在 Paths 文本框中，将出现所选择的文件夹，如图 3-22 所示。

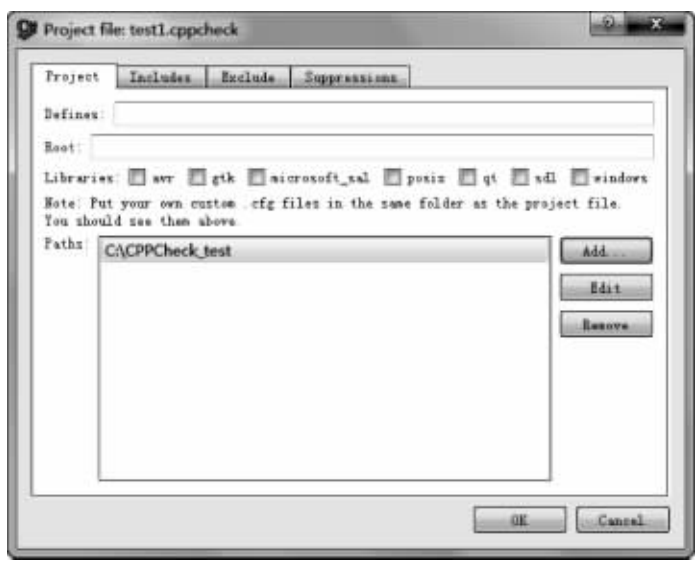

图 3-22　选择待测试文件夹

3. 执行测试

在图 3-22 中，单击 OK 按钮，Cppcheck 将对此文件夹中的所有 C/C++源文件进行测试。测试结果如图 3-23 所示。

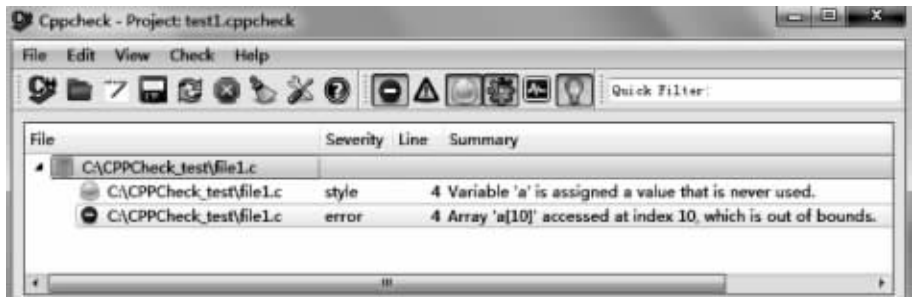

图 3-23 Cppcheck 测试结果

1）测试信息

在测试信息窗格中，各部分显示的内容如下。

File：被测试文件的文件名。

Severity：问题严重级别。

Line：出现的问题在文件中的行号。

Summary：问题描述的简要信息。

鼠标单击某项错误信息，将在窗口的下面显示错误的简要信息。如本例的信息为：

```
Variable 'a' is assigned a value that is never used.
Array 'a[10]' accessed at index 10, which is out of bounds.
```

2）问题严重级别

Cppcheck 中问题严重级别的定义如下。

error(错误)：used when bugs are found 出现的错误。

warning(警告)：suggestions about defensive programming to prevent bugs 为了预防 bug 出现的防御性编程建议。

style(风格)：stylistic issues related to code cleanup (unused functions, redundant code, constness, and such)编码格式问题(没有使用的函数，多余的代码、常量等)。

performance(性能)：Suggestions for making the code faster. These suggestions are only based on common knowledge. It is not certain you'll get any measurable difference in speed by fixing these messages. 建议优化该部分代码的性能。这些建议仅仅基于常识，并不能保证通过修复这些问题，代码运行速度能得到显著的提升。

portability(可移植性)：portability warnings, 64-bit portability, code might work different on different compilers, etc. 移植性警告，64 位可移植，代码可能在不同的编译器上运行情况不同。

information(信息)：Informational messages about checking problems. 关于问题检查的通知消息。

4. 保存测试结果

执行完测试后，可以将测试结果保存到文件。单击菜单栏 File→Save results to file，将弹出文件保存对话框。保存的文件格式为. xml。本例的测试文件内容如下。

```
<?xml version = "1.0" encoding = "UTF - 8"?>
< results version = "2">
    < cppcheck version = "1.69"/>
    < errors >
        < error id = "unreadVariable" severity = "style"
            msg = "Variable &amp;amp;#039;a&amp;amp;#039; is assigned a value that is never used." >
            < location file = "D:\Test\file1.c" line = "4"/>
        </error >
        < error id = "arrayIndexOutOfBounds" severity = "error"
            msg = "Array &amp;amp;#039;a[10]&amp;amp;#039; accessed at index 10, which is out of
bounds." >
            < location file = "D:\Test\file1.c" line = "4"/>
        </error >
    </errors >
```

3.5 PC-lint

3.5.1 PC-lint 简介

PC-lint 是 GIMPEL SOFTWARE 公司开发的 C/C++软件代码静态分析工具，它的全称是 PC-lint/Flexelint for C/C++。PC-lint 能够在 Windows、MS-DOS 和 OS/2 平台上使用，以二进制可执行文件的形式发布，而 Flexelint 运行于其他平台，以源代码的形式发布。PC-lint 在全球拥有广泛的客户群，许多大型的软件开发组织都把 PC-lint 检查作为代码走查的第一道工序。

PC-lint 是一个简单易用的代码静态检查工具。它可以帮助开发人员，检查出语法逻辑上的错误，还能够提出程序在空间利用、运行效率上的改进点。它也可以帮助测试人员检查源码是否符合 C/C++代码编写规范，是否有语法错误，如不匹配的参数、未使用过的变量、空指针的引用等；也可以找出代码逻辑性、合理性上的问题，如不适当的循环嵌套和分支嵌套、不允许的递归和可疑的计算等；还可以利用静态检查的结果做进一步的查错，且能为测试用例的编写提供些许的指导。使用 PC-lint 在代码走读和单元测试之前进行检查，可以提前发现程序隐藏错误，提高代码质量，节省测试时间。

PC-lint 具有下列特点。

(1) PC-lint 是一种静态代码检测工具，可以说，PC-lint 是一种更加严格的编译器，不仅可以像普通编译器那样检查出一般的语法错误，还可以检查出那些虽然完全合乎语法要求，但很可能是潜在的、不易发现的错误。

(2) PC-lint 不但可以检测单个文件，也可以从整个项目的角度来检测问题，PC-lint 在检查当前文件的同时还会检查所有与之相关的文件。

(3) PC-lint 支持几乎所有流行的编辑环境和编译器，比如 Borland C++从 1.×到 5.×各个版本、Borland C++Build、GCC、VC，VC.NET、Watcom C/C++、Source Insight、Intel C/C++等。

(4) 支持 Scott Meyes 的名著(*Effective C++/More Effective C++*)中所描述的各种

提高效率和防止错误的方法。

PC-lint 的官网站：http://www.gimpel.com/

3.5.2 PC-lint 的安装与配置

1. PC-lint 的安装

(1) 在 PC-lint 的官网下载安装包，解压后执行文件(当前版本是 pclint9setup.exe)，将进入安装页面，如图 3-24 所示。

图 3-24 PC-lint 安装

(2) 单击 Next 按钮，进入新的页面，并单击 Next 按钮，将进入 PC-lint 的配置页面，如图 3-25 所示。

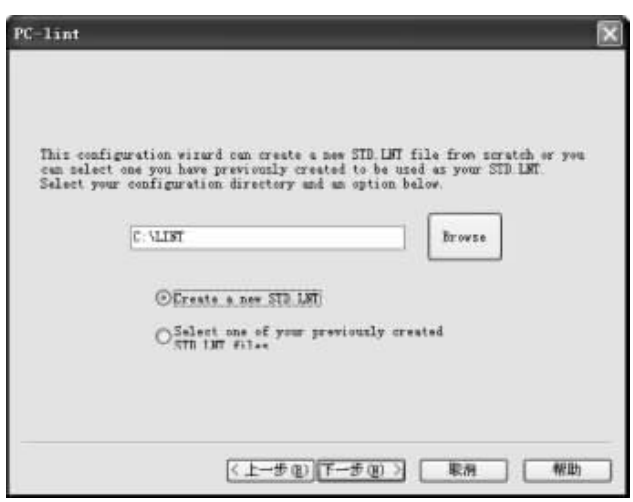

图 3-25 PC-lint 配置

Create a new STD.LNT 是创建或修改已有配置文件 STD.LNT 的选项。如果是第一次配置，则选择此选项。不修改配置路径(C:\LINT)，然后单击“下一步”按钮。

【说明】 界面中配置路径不修改的话就是 PC-lint 安装的路径“C:\ LINT”，新建的 STD. LNT 就存放在这个目录下，当然用户也可选择另外的配置路径存放生成的 STD. LNT。

(3) 接下来是选择编译器，在下拉框中选择自己使用的编程开发环境，即 PC-lint 要使用的地方。由于我们配置的是 VC++6.0 环境，因此选择 Microsoft Visual C++6.×(co-msc60.lnt)。然后单击“下一步”按钮。

(4) 在 Libraries 页面中，会看到一个库类型的列表，在这里选择一个或多个编译时使用的库，如图 3-26 所示。这一步就是依据读者的开发环境及 PC 的配置进行选择，对于 VC++ 6.0 的环境，建议选择 Microsoft Foundation Class Library、Windows NT、Windows 32-bit 和 Standard Template Library。设置好后，单击“下一步”按钮。

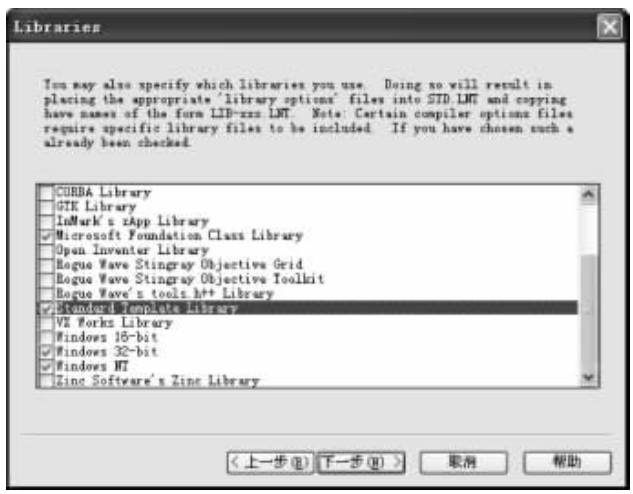

图 3-26 PC-lint Libraries 设置

【说明】 各种库的配置文件名为 lib-×××.lnt，配置向导会把选中的库的 lnt 配置文件复制到配置路径下。

(5) 接下来选择 PC-lint 进行检查时所依据的标准，如图 3-27 所示。其中列出了为 C/C++编程提出过重要建议的作者。选择某位作者后，其提出的编程建议方面的选项将被打开，作者建议的配置名为 AU-×××.LNT。一般要选择 MISRA 2004，这是目前高效编程中标准最好的。

【说明】 同样，选中作者建议的 AU-×××.LNT，也会被配置向导复制到配置路径下。

(6) 下面是选择用何种方式设置包含文件目录，如图 3-28 所示。这里选择“Create - i option”方式，然后单击“下一步”按钮。

【说明】 这里有两个选项：第一个选项是使用-i 选项协助设置。-i 选项体现在 STD. LNT 文件中，每个目录前以-i 引导，目录间以空格分隔。第二个选项是跳过这一步，手工设置。建议选择第一种。

(7) 如果第(6)步中选择使用-i 选项，安装程序会接着让用户选择开发环境的一些检查路径所对应的文件夹，主要是一些头文件。检查时会检查是否与这个头文件内容冲突了。

在文本框里，可以手工输入文件包含路径，用分号“；”或用 Ctrl+Enter 键换行来分隔多个包含路径。也可以单击 Browse 按钮，在目录树中直接选择，如图 3-29 所示。

图 3-27 PC-lint Author Recommendations 设置

图 3-28 PC-lint Header Files 设置

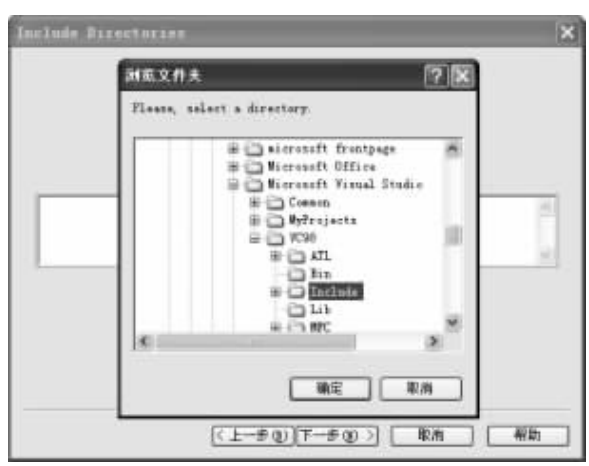

图 3-29 PC-lint Include Directories 设置

【说明】 如果不输入包含文件目录，直接选择下一步，安装完成后在 std.lnt 文件中手工添加。注意如果目录名中有长文件名，使用时要加上双引号，例如-i“E:\Program Files\MSVC\VC98\Include”。

添加完成后，将显示所添加的路径。VC++ 6.0 环境选择完毕后，如图 3-30 所示。

图 3-30 VC6.0 环境中添加的路径

(8) 接下来将会准备产生一个控制全局编译信息显示情况的选项文件 OPTIONS.LNT，这里选择 No，即不取消这些选项，如图 3-31 所示。然后单击“下一步”按钮，此时将弹出一个对话框，单击“确定”按钮。

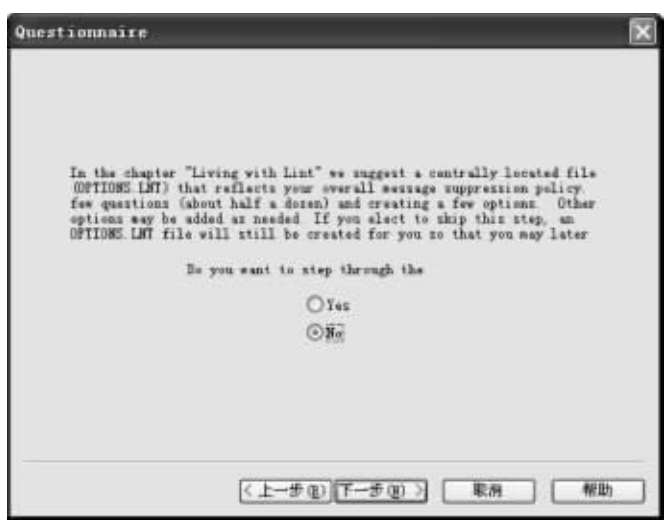

图 3-31 PC-lint Questionnaire 设置

【说明】 该文件的产生方式有两种，一种是安装程序对几个核心选项逐一解释并提问是否取消该选项，如果选择取消，则会体现在 OPTIONS.LNT 文件中，具体体现方式是在

该类信息编码前加-e，后面有一系列逐一选择核心选项的过程。如果选择第二种选择方式，安装文件会生成一个空的 OPTIONS. LNT 文件，等以后在实际应用时加入必要的选项。

(9) 接着选择所支持的集成开发环境选项，可选多个或一个也不选，PC-lint 提供了集成在多种开发环境中工作的功能，例如，可集成在 VC、BC、Source Insight 中。这里选择 MS VC++6，这样 env-v6. lnt 就会被复制到配置路径中，如图 3-32 所示。

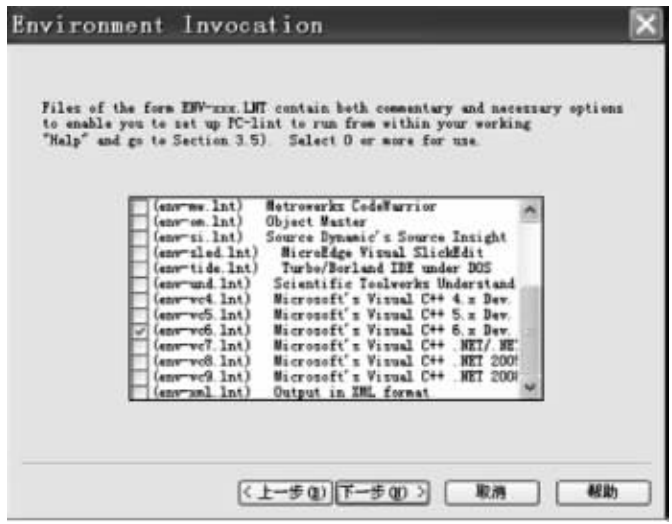

图 3-32 Environment Invocation 设置

(10) 安装程序会生成一个 LIN. BAT 文件，该文件是运行 PC-lint 的批处理文件，为了使该文件能在任何路径下运行，安装程序提供了两种方法供选择，如图 3-33 所示。第一种方法是选择把 LIN. BAT 复制到任何一个 PATH 目录下。第二种方法是生成一个 LSET. BAT 文件，在每次使用 PC-lint 前先运行它来设置路径，或者把 LSET. BAT 文件的内容复制到 AUTOEXEC. BAT 文件中。建议选择第一种方法，指定的目录为安装目录。

图 3-33 Batch Files 设置

【说明】 以上配置过程中在配置路径下产生的多个 *.lnt 文件，除了 std.lnt，std_a.lnt，std_b.lnt，option.lnt 为配置向导所生成，其他 co-xxx.lnt，lib-xxx.lnt，env-xxx.lnt 均是从 C:\lint9\lnt 中复制出来的，在这个目录下还有其他 PC-lint 所支持的编译器、库及集成开发环境的 lnt 配置文件，所有的 lnt 文件均为文本文件。

经过了上述步骤后，PC-lint 软件本身部分的配置就算完成，接下来只需要结合到用户自己的开发环境就行了。但也需要对用户自己的开发环境进行配置，至少对于 VC++ 6.0 或是 Source Insight 来说是需要的。

2. VC++ 6.0 中 PC-lint 的配置

PC-lint 与 VC 集成的方式就是在 VC 的集成开发环境中添加几个定制的命令，添加定制命令的方法是选择 VC 菜单栏的 Tools→Customize 菜单，在弹出的 Customize 窗口中选择 Tools 标签，在定制工具命令的标签页中添加定制命令。

1) PC-lint 检查当前文件的配置

使用 PC-lint 检查当前文件是否存在隐藏错误，需要进行相应的配置。配置内容如图 3-34 所示。Command 里的路径是 PC-lint 的安装路径。如果路径不同，仅需将路径替换。

配置 PClint Current File 命令如下。

```
Command: C:\lint\lint-nt.exe
Arguments: -i"C:\lint" -u std.lnt env-vc6.lnt "$(FileName)$(FileExt)"
```

其中，std.lnt 是为 VC 编译环境定制的配置文件，$(FileName)和 $(FileExt)是 VC 集成开发环境的环境变量，"$(FileName)$(FileExt)"表示当前文件的文件名。

同时需要选中 Use Output Window 选项。

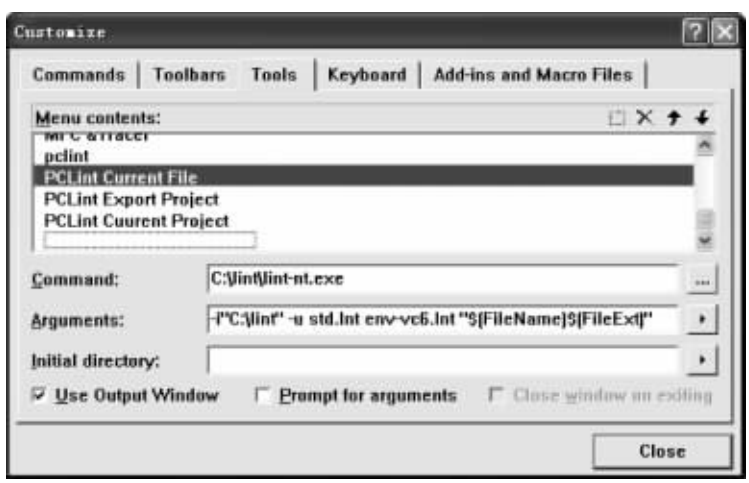

图 3-34 PC-lint Current File 配置

2) PC-lint 导出项目的配置

如果要检查当前的整个项目的内容，首先需要将这个项目的错误导入到某个文件。配置 PClint Export Project 命令如下。

```
Command: C:\lint\lint-nt.exe;
Arguments: +linebuf $(TargetName).dsp>$(TargetName).lnt;
```

参数+linebuf表示加倍行缓冲的大小,最初是600B,行缓冲用于存放当前行和读到的最长行的信息。$(TargetName)是VC集成开发环境的环境变量,表示当前激活的Project名字同时选中Use Output Window选项,如图3-35所示。

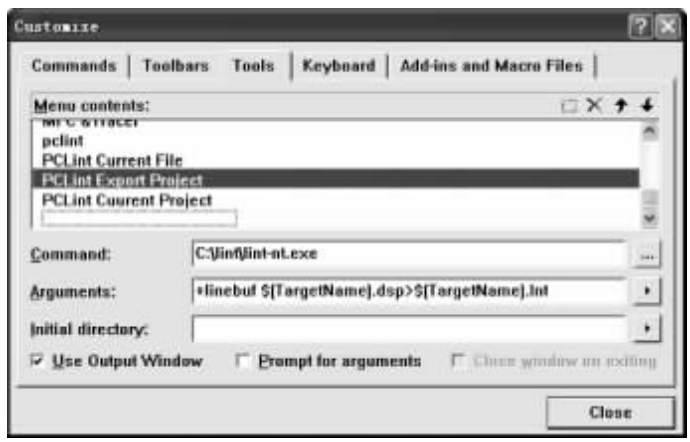

图3-35 PC-lint Export Project配置

3) PC-lint检查当前项目的配置

配置PC-lint Current Project命令如下。

```
Command: C:\lint\lint-nt.exe;
Arguments: +ffn -i"C:\lint" std.lnt env-vc6.lnt $(TargetName).lnt;
```

这个命令的结果就是将整个工程的检查结果输出到与工程同名的.chk文件中。参数中+ffn表示Full File Names,可被用于控制是否使用完整路径名称表示。同时选中Use Output Window选项,如图3-36所示。

配置完成后,在VC++6.0的Tools下将出现上述配置的命令,如图3-37所示。

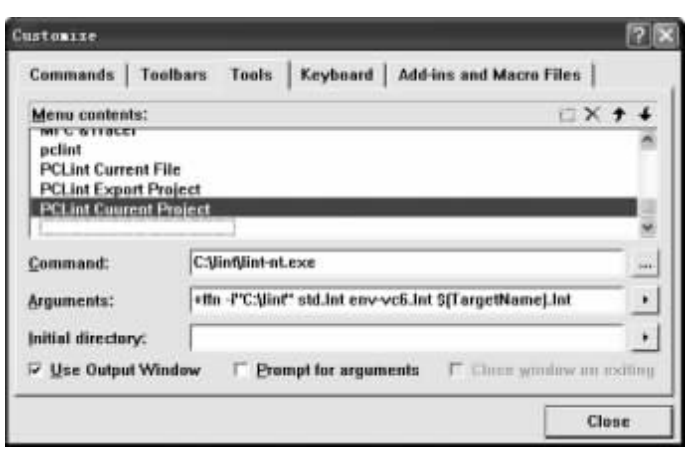

图3-36 PC-lint Current Project配置

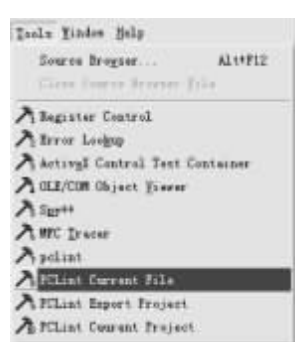

图3-37 VC++6.0集成PC-lint命令

PC-lint 关于 VC++6.0 的安装配置完成，接下来可以利用这些命令对整个项目进行安全性和错误性检查。

3.5.3 PC-lint 的代码检查功能

PC-lint 能够检查出很多语法错误和语法上正确的逻辑错误，PC-lint 为大部分错误消息都分配了一个错误号，编号小于 1000 的错误号是分配给 C 语言的，编号大于 1000 的错误号则用来说明 C++的错误消息。表 3-2 列出了 PC-lint 告警消息的详细分类。

表 3-2 PC-lint 告警消息分类

错误说明	C	C++	告警级别
语法错误	1～199	1001～1199	1
内部错误	200～299		
致命错误	300～399		
告警	400～699	1400～1699	2
消息	700～899	1700～1899	3
可选信息	900～999	1900～1999	4

以 C 语言为例，其中，编号 1～199 指的是一般编译器也会产生的语法错误。编号 200～299 是 PC-lint 程序内部的错误，这类错误不会出现在代码中。编号 300～399 指的是由于内存限制等导致的系统致命错误。编号 400～999 中出现的提示信息，是根据隐藏代码问题的可能性进行分类的。其中，编号 400～699 指的是被检查代码中很可能存在问题而产生的告警信息。编号 700～899 中出现的信息，产生错误的可能性相比告警信息来说级别要低，但仍然可能是因为代码问题导致的问题。编号 900～999 是可选信息，它们不会被默认检查，除非在选项中指定要检查它们。

PC-lint/Felexlint 提供了和许多编译器类似的告警级别设置选项-wLevel，它的告警级别分为以下几个级别，默认告警级别为三级。

w0 不产生信息(除了遇到致命的错误)。

w1 只生成错误信息，没有告警信息和其他提示信息。

w2 只有错误和告警信息。

w3 生成错误、告警和其他提示信息(这是默认设置)。

w4 生成所有信息。

PC-lint/Felexlint 还提供了用于处理函数库的头文件的告警级别设置选项-wlib(Level)，这个选项不会影响处理 C/C++源代码模块的告警级别。它有和-wLevel 相同的告警级别，默认告警级别为三级。

wlib(0)不生成任何库信息。

wlib(1)只生成错误信息(当处理库的源代码时)。

wlib(2)生成错误和告警信息。

wlib(3)生成错误、告警和其他信息(这是默认设置)。

wlib(4)产生所有信息。

PC-lint 的检查分为很多种类，有强类型检查、变量值跟踪、语义信息、赋值顺序检查、弱

定义检查、格式检查、缩进检查、const 变量检查和 volatile 变量检查等。对每一种检查类型，PC-lint 都有很多详细的选项，用以控制 PC-lint 的检查效果。

PC-lint 的选项有三百多种，这些选项可以放在注释中(以注释的形式插入代码中)，例如：

/ * lint option1 option2 … optional commentary * /选项可以有多行

//lint option1 option2 … optional commentary 选项仅为一行(适用于 C++)。

选项间要以空格分开，lint 命令一定要小写，并且紧跟在/ * 或//后面，不能有空格。如果选项由类似于操作符和操作数的部分组成，例如-esym(534, printf, scanf, operator new)，其中最后一个选项是 operator new，那么在 operator 和 new 中间只能有一个空格。

PC-lint 的选项还可以放在宏定义中，当宏被展开时选项才生效。例如：

#define DIVZERO(x) / * lint -save -e54 * / ((x) /0) / * lint -restore * /允许除数为 0 而不告警。

3.5.4 PC-lint 错误信息

1. C 语法错误

C 语法部分错误信息如表 3-3 所示。详细信息请查阅 PC-lint 的参考手册。安装好 PC-lint 后，可在菜单栏上打开参考手册。

表 3-3 C 语法错误信息

编号	错 误 信 息	说 明
1	Unclosed Comment (*Location*)	未关闭注释(位置)
2	Unclosed Quote	未关闭的引号(单引号或双引号)
3	#else without a #if	#else 没有一个#if(在一个区域内有一个#else，但是没有一个#if，#ifdef 或#ifndef)
4	Too many #if levels	太多的#if 嵌套层次
5	Too many #endif's	太多的#endif(出现一个#endif，但不是#if 或#ifdef 或#ifndef. 的)
6	Stack Overflow	堆栈溢出
7	Unable to open include file:*FileName*	不能打开引用的文件
8	Unclosed #if (*Location*)	未关闭的#if(一个#if，或#ifdef 或#ifndef 没有遇到相应的#endif)
9	Too many #else's in #if(*Location*)	#if(位置)包含过多的 else
10	Expecting*String*	期望的字符串(当一定的保留字没有被认出时，给出这条消息)
11	Excessive Size	超出大小范围，在#include l 行确定的文件名的长度超过了 FILENAME_MAX 字符
12	Need<or"	需要<或"
13	Bad type	错误类型
14	Symbol ' *Symbol* ' previously defined (*Location*)	符号以前定义过(符号被第二次定义)
15	Symbol '*Symbol*' redeclared(*TypeDiff*) (*Location*)	符号被以前声明过或在其他模块定义过(其他位置)的类型和在当前位置的声明的类型不同

续表

编号	错 误 信 息	说　　明
16	Unrecognized name — A # directive is not followed by a recognizable word.	不认识的名字
17	Unrecognized name — A non-parameter is being declared where only parameters should be	未被承认的名称
18	Symbol '*Symbol*' redeclared(*TypeDiff*)	符号重新声明(TypeDiff)和此位置冲突
19	Useless Declaration	无效的声明
20	Illegal use of =	非法使用=
21	Expected {	需要{
22	Illegal operator	非法的操作符
23	Expected colon	需要冒号
24	Expected an expression, found '*String*'	期望一个表达式,但是得到一个字符串
25	Illegal constant	非法的常量
26	Expected an expression, found '*String*'	期望一个表达式,但是得到一个字符串
27	Illegal character (0x*hh*)	非法的字符(0xff)。消息中提供十六进制代码
28	Redefinition of symbol '*Symbol*' *Location*	重定义一个符号(符号位置)
29	Expected a constant	期望一个常量,但是没有得到。可能是在 case 关键字后,数组维数、bit field 长度、枚举值、# if 表达式等
30	Redefinition of symbol '*Symbol*' conflicts with *Location*	重新定义一个符号。数据对象或函数在此模块中以前定义过又被定义
31	Field size (member '*Symbol*') should not be zero	Field 大小不能是 0
32	Illegal constant	非法常量。当一个八进制的常量包含数字 8 或 9 时,这是一个错误的形式
33	Non-constant initializer	非常量初始化。在一个 static 数据项中发现非常量初始化
34	Initializer has side-effects	初始化有副作用
35	Redefining the storage class of symbol '*Symbol*' conflicts with *Location*	重新定义存储类的符号'Symbol'和位置 Location 冲突
36	Value of enumerator '*Symbol*' inconsistent (conflicts with *Location*)	枚举值'Symbol'不一致
37	Offset of symbol '*Symbol*' inconsistent (*Location*)	符号'Symbol'的偏移量不一致(Location)
38	Redefinition of symbol '*Symbol*' conflicts with *Location*	重新定义符号'Symbol'和位置 Location 冲突
39	Undeclared identifier '*Name*'	没有声明标识符'Name'
40	Redefinition of symbol '*Symbol*'	重新定义符号'Symbol'
41	Expected a statement	需要一条语句
42	Vacuous type for variable '*Symbol*'	变量'Symbol'是虚类型的
43	Need a switch	需要一个 switch：在一个 switch 外出现 case 或 default 语句
44	Bad use of register	错误地使用 register

续表

编号	错误信息	说明
45	Field type should be int	域类型应该是 int
46	Bad type— Unary minus requires an arithmetic operand	错误的类型：一元减需要一个算术操作数
47	Bad type— Unary * or the left hand side of the ptr (－＞) operator requires a pointer operand	错误的类型：一元的 * 或左边的指针(－＞)操作符需要一个指针操作数
48	Expected a type — Only types are allowed within prototypes	期望一个类型：在原型内只有类型被允许
49	Attempted to take the address of a non-lvalue	试图取非左值的地址
50	Expected integral type— Unary ～ expects an integral type (signed or unsigned char,short,int,or long)	期望整型：一元运算符“～”需要一个整型(signed 或 unsigned char、short、int 或 long)
51	Expected an lvalue	期望一个左值：自动递减(--)和自动递增(＋＋)操作符需要一个左值(对分配操作符左手边合适的值)
52	Expected a scalar	期望一个标量：自动递减(--)和自动递增(＋＋)操作符可能只应用于标量(算术和指针)或这些操作符定义的对象
53	Division by 0	被 0 除
54	Bad type— The context requires a scalar, function, array, or struct(unless -fsa)	错误类型：上下文需要一个标量、函数或结构(除非-fsa)
55	Bad type— Add/subtract operator requires scalar types and pointers may not be added to pointers	错误类型：需要标量类型和指针的加/减操作符可能被加到指针中
56	Bad type— Bit operators (&, | and ^) require integral arguments	错误类型：Bit 操作符(&, |和^)需要 require 整型参数
57	Bad type— Bad arguments were given to a relational operator	错误类型：错误的参数给相关的操作符
58	Bad type— The amount by which an item can be shifted must be integral	错误类型：移位的数量必须是整数
59	Bad type— The value to be shifted must be integral	错误类型：被移位的值必须是整数
60	Bad type— The context requires a Boolean. Booleans must be some form of arithmetic or pointer	错误类型：上下文需要一个布尔值,布尔值必须是算术或指针形式
61	Incompatible types (*TypeDiff*) for operator ':'	与操作符':'矛盾的类型
62	Expected an lvalue Type mismatch (*Context*)(*TypeDiff*)	预计一个左值类型不匹配(上下文)
63	Type mismatch(*Context*)(*TypeDiff*)	上下文类型不匹配

续表

编号	错误信息	说明
64	Expected a member name— After a dot (.) or pointer(－＞) operator a member name should appear	期望一个成员名称。在一个(.)或(－＞)操作符后,需要一个成员名
65	Bad type— A void type was employed where it is not permitted	错误类型：不允许使用 void 类型
66	Can't cast from*Type* to *Type* — Attempt to cast a non-scalar to an integral	不能从 Type 到 Type 计算：试图非标量到整数计算
67	Can't cast from*Type* to *Type* — Attempt to cast a non-arithmetic to a float	不能从 Type 到 Type 计算：试图非标量到浮点数计算
68	Can't cast from*Type* to *Type* — Bad conversion involving incompatible structures or a structure and some other object	不能从 Type 到 Type 计算：涉及结构到结构或其他对象间的不匹配的转换
69	Can't cast from*Type* to *Type* — Attempt to cast to a pointer from an unusual type (non-integral)	不能从 Type 到 Type 计算：试图计算一个指针到一个非寻常的类型(非整数)间的计算
70	Can't cast from*Type* to *Type* — Attempt to cast to a type that does not allow conversions	不能从 Type 到 Type 计算：试图计算一个不允许转换的类型
71	Bad option '*String*' — Was not able to interpret an option	错误的选项'String'：不能解释一个选项
72	Bad left operand— The cursor is positioned at or just beyond either an －＞ or a . operator	错误的左操作数：指针位于－＞或.操作符的前面
73	Address of Register	Register 的地址：试图应用地址操作符(&)到一个存储类是一个 register 的变量
74	Too late to change sizes (option '*String*')	太晚改变大小：在所有的或部分的模块被处理后,给出大小选项。确保在第一个模块被处理时或在任何模块被处理前的命令行上对目标的大小重新设置
75	can't open file '*String*'	不能打开文件 String
76	Address of bit-field cannot be taken	位域的地址不能取
77	Symbol '*Symbol*' typedef'ed at *Location* used in expression	定义为类型的符号'Symbol'在 Location 处用作表达式：符号被定义在一个 typedef 语句中,因此被认为是一个类型,后来发现在上下文中期望一个表达式
78	Bad type for % operator	%操作符类型错误
79	This use of ellipsis is not strictly ANSI	使用省略号不是严格的 ANSI 标准
80	struct/union not permitted in equality comparison	结构体/联合体不允许在等式比较中。两个 struct 或 union 被用于比较操作,如＝＝或！＝。这在 ANSI 标准中是不允许的
81	return ＜exp＞; illegal with void function	返回＜exp＞;非法的 void 函数
82	Incompatible pointer types with subtraction	在减操作中不兼容的指针类型
83	sizeof object is zero or object is undefined	对象大小是零,或者对象未定义

续表

编号	错 误 信 息	说 明
84	Array '*Symbol*' has dimension 0	数组'Symbol'有 0 维。一个数组被声明在上下文中没有一个维数,需要一个非零的维数
85	Structure '*Symbol*' has no data elements	结构'Symbol'没有数据元素
86	Expression too complicated for ＃ifdef or ＃ifndef	＃ifdef 或＃ifndef 表达式太复杂
87	Symbol '*Symbol*' is an array of empty elements	符号'Symbol'是一个有空元素的数组
88	Argument or option too long ('*String*')	参数或选项太长(String)
89	Option '*Name*' is only appropriate within a lint comment	选项'Name'仅合适在一个 lint 注释中
90	Line exceeds*Integer* characters (use ＋linebuf)	行超过整型字符(使用＋linebuf)
91	Negative array dimension or bit field length (*Integer*)	数组维数或位域长度为负数
92	New-line is not permitted within string arguments to macros	在宏的字符串参数内不允许新的行
93	Expected a macro parameter but instead found '*Name*'	期望一个宏参数
94	Illegal parameter specification	非法的参数
95	Unexpected declaration	不期望的声明。在一个原型后,只能是一个逗号、分号、右括号或左括号
96	Conflicting types	冲突的类型
97	Conflicting modifiers	冲突的修饰符
98	Illegal constant	非法常量
99	Label '*Symbol*'(*Location*) not defined	标签'Symbol'(Location)没有定义
100	Invalid context	无效的上下文。遇到一个 continue 或 break 语句,没有合适的上下文
101	Attempt to assign to void	试图给一个 void 分配
102	Assignment to const object	分配给一个常量
103	Inconsistent enum declaration	不一致的枚举声明
104	Inconsistent structure declaration for tag '*Symbol*'	不一致的结构声明
105	Struct/union not defined	结构体/联合体未定义
106	Inappropriate storage class—A storage class other than register was given in a section of code this is dedicated to declaring parameters	不合适的存储类。一个不同于 register 的存储类在一个代码段中被给出,专注于声明参数
107	Inappropriate storage class—A storage class was provided outside any function that indicated either auto or register	不合适的存储类。一个存储类在函数外被给出,表示 auto 或 register。这个存储类仅适合于函数内
108	Too few arguments (*Integer*) for prototype '*Name*'	原型参数太少。一个函数提供的参数少于范围内原型指示的个数

续表

编号	错误信息	说明
109	Too many arguments (*Integer*) for prototype '*Name*'	原型参数太多。一个函数提供的参数多于范围内原型指示的个数
110	Digit (*Char*) too large for radix	数字(字符)对基数太大。例如,08 在一些编译器中被认为是 8,但是它应该是 010 或 8
111	Macro '*Symbol*' defined with arguments at *Location* this is just a warning	宏定义符号是一个标识符
112	Pointer to void not allowed	指针指向 void 是不允许的。这包括减、加和关系操作符(> >= < <=)
113	Too many storage class specifiers	太多的存储类定义符(如: static、extern, typedef, register 或 auto,只允许有一个)
114	Inconsistent structure definition '*Symbol*'	不一致的结构定义'Symbol'
115	Illegal constant— An empty character constant ('') was found	非法常量。一个空字符常量('')被发现
116	Pointer to function not allowed	指针不允许指向函数
117	declaration expected, identifier '*Symbol*' ignored	期望声明,标识符'Symbol'被忽略
118	Expected integral type	期望一个整型类型。在一个 switch 语句中的表达式,必须是 int 的一些变种(可能是 long 或 unsigned)或一个 enum
119	syntax error in call of macro '*Symbol*' at location *Location*	在位置 location 调用宏'Symbol'时语法错误
120	Expected function definition	期望函数定义
121	Too many initializers for aggregate	初始化太多
122	Missing initializer — An initializer was expected but only a comma was present	丢失初始化器
123	comma assumed in initializer	假定在初始化器中用逗号,在两个初始化器之间缺少逗号
124	Illegal macro name	非法的宏名称
125	constant '*Symbol*' used twice within switch	在 switch 内,常量'Symbol'被使用了两次
126	Can't add parent '*Symbol*' to strong type *String*; creates loop	不能增加父类型'Symbol'到强类型 String; 创建循环
127	Can't take sizeof function	不能对函数进行 sizeof 计算
128	Type appears after modifier	类型出现在一个修饰符后
129	The following option has too many elements: '*String*'	下列选项有太多的元素: 'String'
130	Non-existent return value for symbol '*Symbol*', compare with *Location*	符号'Symbol'不存在返回值
131	Type expected before operator, void assumed	在操作符前期望一个类型,假定是 void
132	Assuming a binary constant	假定一个二进制常量
133	sizeof takes just one argument	sizeof 只能有一个参数

续表

编号	错误信息	说明
134	member '*Symbol*' previously declared at *Location*	成员'Symbol'在 Location 以前声明过
135	C++construct '*String*' found in C code	C++构造'String'在代码中发现。一个非法的结构在 C 代码中发现,它看起来适合于 C++
136	Token '*String*' unexpected *String*	记号'String'不期望 String
137	Token '*Name*' inconsistent with abstract type	记号'Name'和抽象类型不一致
138	Lob base file '*file name*' missing	丢失 Lob 基础文件'file name'
139	Could not create temporary file	不能创建临时文件
140	Could not evaluate type '*String*', int assumed	不能确定'String'的类型,假定为 int
141	Ignoring {…} sequence within an expression, 0 assumed	在一个表达式内忽略{…}系列,假定为 0

2. 致命错误

致命错误信息如表 3-4 所示。

表 3-4 致命错误信息

编号	错误信息	说明	详细描述
301	Stack overflow	堆栈溢出	当处理声明时,有一个堆栈溢出
302	Exceeded Available Memory	超过可用的内存	内存被耗尽
303	String too long (try + macros)	字符串太长(尝试+macros)	一个单独的#define 定义或宏调用超过一个内部的限制(超过 409 字符)。诊断指出的问题可以被使用一个选项校正
304	Corrupt object file, code *Integer*, symbol=*String*	被破坏的目标文件,代码 Integer,符号=String	一个 PC-lint/FlexeLint 目标文件是明显的被破坏的
305	Unable to open module '*file name*'	不能打开模块'file name'	file name 这个模块不能被打开,可能是拼写错误名称
306	Previously encountered module '*FileName*'	以前遇到的模块'FileName'	FileName 这个模块以前遇到过,这可能是用户的一个失误
307	Can't open indirect file '*FileName*'	不能打开间接文件'FileName'	FileName 是间接文件的名称。这个名称的间接文件(结尾是.lnt)不能被打开
308	Can't write to standard out	不能写到标准输出	stdout 被发现等于 NULL
309	# error …— The # error directive was encountered.	遇到#error	遇到错误,省略号反映最初的行。通常在这点中断。如果设置 fce (连续#error)标志,处理将继续
310	Declaration too long	声明太长	发现一个单独的声明对于内部的缓冲太长(差不多 2000 个字符)
312	Lint Object Module has obsolete or foreign version id	荒废的或外来的版本号	PC-lint/FlexeLint 以前的或不同的版本产生。删除这个.lob 文件,使用新版本的 PC-lint/FlexeLint 重新创建它

续表

编号	错误信息	说明	详细描述
313	Too many files	太多文件	PC-lint/FlexeLint 能处理的文件的数量超过内部的限制。目前，文件的数量限制到 6400
314	Previously used .lnt file: FileName	以前使用的.lnt 文件：FileName	指定名称的间接文件以前遇到过。如果这不是一次事故，可以抑制这个信息
315	Exceeded message limit (see -limit)	超过信息限制	超过信息的最大量。通常没有限制，除非强加限制使用选项-limit(n)
316	Error while writing to file "*file name*"	写文件"file name"时错误	给定的文件不能输出打开
321	Declaration stack overflow	声明堆栈溢出	当处理一个声明时在堆栈使用于特定的数组、指针、函数或引用修饰符时发生堆栈溢出
322	Unable to open include file *FileName*	不能打开包含文件 FileName	FileName 是不能被打开的包含文件的名称。目录寻找通过选项：－i ＋fdi 和 INCLUDE 环境变量控制
323	Token '*String*' too long	记号 String 太长	试图为以后的重用存储一个记号，超过一个固定的大小缓冲(通过大小 M_TOKEN 来控制)
324	Too many symbols*Integer*	太多的符号	遇到太多的符号，打断内部的限制
325	Cannot re-open file ' *file name*'	不能重新打开文件'file name'	在大量嵌套的 include 的情况下，在外部的文件需要在一个新的文件被打开前被关闭。然后这些外部文件需要被重新打开。当试图重新打开这样的一个文件时，发生一个错误

3. C++语法错误

C++语法错误信息如表 3-5 所示。

表 3-5 C++语法错误信息

编号	错误信息	说明
1001	Scope ' *Name* ' must be a struct or class name	'Name'应该是一个结构体或一个类名
1002	' this ' must be used in class member function	'this'指针必须在类的成员函数中应用，在类成员函数外是无效的
1003	'this' may not be used in a static member function	'this'指针不可以在类静态成员函数中使用
1004	Expected a pointer to member after .* or －＞*	在.*或－＞*后需指向结构或类成员
1005	Destructor declaration requires class	析构函数要在类中声明
1006	Language feature '*String*' not supported	该特性目前版本不支持

续表

编号	错误信息	说明
1007	Pure specifier for function '*Symbol*' requires a virtual function	后有'='的函数声明应该是纯虚函数
1008	Expected '0' to follow '=', text ignored	'='后要跟'0'
1009	operator '*String*' not redefinable	操作符'String'不能重新定义。操作符如'.*','?','::','.'等不能重定义
1010	Expected a type or an operator	缺少类型或运算符。类型包括 new, delete, (), [],逗号等
1011	Conversion Type Name too long	转义类型名太长,限 50 字符
1012	Type not needed before 'operator type'	在运算符类型前不需要类型
1013	Symbol '*Name*' not a member of class '*Name*'	在'.'或'->'后的'Name'不是类(结构,联合)的成员
1014	Explicit storage class not needed for member function '*Symbol*'	对成员函数来说,不需要定义为显示存储类别
1015	Symbol '*Name*' not found in class	符号'Name'没有在类中发现
1016	Symbol '*Symbol*' is supposed to denote a class	'Symbol'可能是一个类
1017	conflicting access-specifier '*String*'	存取属性冲突,基类必在子类前声明
1018	Expected a type after 'new'	'new'后应跟类型
1019	Could not find match for function '*Symbol*(*String*)'	函数"Symbol(String)"不能找到匹配的
1022	Function: '*String*' must be a class member	函数'Sting'必须是类成员
1023	Call '*Name*' is ambiguous; candidates: '*String*'	调用'Name'是不确定的。调重载函数或操作符是不确定的
1024	No function has same argument count as '*Invocation*'	调用函数时,找不到具有相同参数的函数
1025	No function matches invocation '*Name*' on arg no. *Integer*	调用函数时,与声明函数的参数冲突
1026	Undominated function '*String*' does not dominate '*String*' on call to '*String*'	调用函数'string'时并不能找到优于其他'string'的函数
1027	Non-consecutive default arguments in function '*String*', assumed 0	默认参数应为连续的 例如: f(int i=0, int j, int k=0);中参数默认是非法的
1028	Last argument not default in first instance of function '*String*', assumed 0	函数后续变量没有默认值,例如 0
1029	Default argument repeated in function '*String*'	默认参数值重复(默认值只应给出一次)
1030	Not all arguments after arg no. *Integer* are default in function '*String*'	默认参数后的所有参数都应有默认值(一个具有默认值的参数要么其后所有参数都有默认值,要么为最后一个参数)
1031	Local variable '*Symbol*' used in default argument expression	局部变量应为参数默认值
1032	Member '*String*' cannot be called without object	成员'string'应通过对象调用

续表

编号	错 误 信 息	说　　明
1033	Static member functions cannot be virtual	静态成员函数不能为虚函数
1034	Static member '*Symbol*' is global and cannot be redefined	静态成员是全局的，不能被重定义
1035	Non-static member '*Symbol*' cannot initialize a default argument	非静态成员不能初始化默认参数
1036	ambiguous reference to constructor; candidates: '*String*'	构造函数的不确定引用
1037	ambiguous reference to conversion function; candidates: '*String*'	转化函数的不确定引用(除非类提供实例，否则类成员不能初始化默认变量)
1038	type*Name* not found, nested type '*Name*::*String*' assumed	类型'Name'没找到，假设为嵌套类型'Name::String'
1039	Symbol '*Symbol*' is not a member of class '*String*'	'Symbol'不是类'String'的成员
1040	Symbol '*Symbol*' is not a legal declaration within class '*String*'	'Symbol'在类'String'内不合法的声明
1041	Can't declare '*String*', assumed 'operator *String*'	不能声明'String'，假设为'operator String'
1042	At least one class-like operand is required with*Name*	定义操作符时需要至少一个类作为操作数
1043	Attempting to 'delete' a non-pointer	企图'delete'一个非指针
1046	member '*Symbol*', referenced in a static function, requires an object	成员在静态函数中需由对象来引用
1047	a template declaration must be made at file scope	模板声明需为全文件范围
1048	expected a constant expression	期望一个常量表达式
1049	Too many template arguments	太多模板参数，比初始模板声明中参数要多
1050	expected a template argument list '<…>' for template '*Symbol*'	模板缺少参数列表
1051	Symbol '*Name*' is both a function and a variable	符号'Name'既是函数又是变量
1052	a type was expected, 'class' assumed	缺少类型，如'class'类型
1053	'*String*' cannot be distinguished from '*String*'	'String'不能和'String'区分
1054	template variable declaration expects a type, int assumed	模板变量缺少类型，如 int 型
1055	Symbol '*Symbol*' undeclared, assumed to return int	符号'Symbol'未声明，假设返回 int
1056	assignment from void * is not allowed in C++	C++中从 void * 赋值是不允许的
1057	member '*Symbol*' cannot be used without an object	成员应通过对象引用

续表

编号	错 误 信 息	说 明
1058	Initializing a non-const reference '*Symbol*' with a non-lvalue	用 non-lvalue 初始化非常量引用
1059	Can't convert from*Type* to *Type*	不能从类型 TYPE 转变到类型 TYPE
1060	*String* member *Symbol* is not accessible to non-member non-friend functions	成员不能被非成员函数、非友元函数访问
1061	*String* member *Symbol* is not accessible through non-public inheritance	成员不能被非公有继承类访问
1062	template must be either a class or a function	模板必须为类或函数
1063	Argument to copy constructor for class '*Symbol*' should be a reference	复制构造函数中参数应为引用
1064	Template parameter list for template '*Symbol*' inconsistent with *Location*	模板参数列表声明定义不一致
1065	Symbol '*Symbol*' not declared as "C" conflicts with *Location*	'Symbol'没有声明为"C"导致与位置冲突
1066	Symbol '*Symbol*' declared as "C" conflicts with *Location*	'Symbol'声明为"C"导致与位置冲突
1067	invalid prototype for function '*Symbol*'	函数原型无效
1068	Symbol '*Symbol*' can not be overloaded	符号'Symbol'不能被重载。操作符 delete、[]可以重定义，但不能被重载
1069	Symbol '*Name*' is not a base class of class '*Name*'	符号'Name'不是基类
1070	No scope in which to find symbol '*Name*'	在查找'Name'时无效范围
1071	Constructors and destructors can not have return type	构造析构函数不能有返回类型
1072	Reference variable '*Symbol*' must be initialized	引用变量必须初始化
1073	Insufficient number of template parameters; '*String*' assumed	模板参数不足
1074	Expected a namespace identifier	期望一个命名空间标识符
1075	Ambiguous reference to symbol '*Symbol*' and symbol '*Symbol*'	不确定的引用，两个命名空间中有相同的名字
1076	Anonymous union assumed to be 'static'	匿名联合体必须声明为静态的(static)
1077	Could not evaluate default template parameter '*String*'	不能评估默认模板参数
1078	class '*Symbol*' should not have itself as a base class	不能把自己作为自己的基类
1079	Could not find '>' or ',' to terminate template parameter at *Location*	缺少'>'或','，不能终止模板参数列表
1080	Definition for class '*Name*' is not in scope	类定义不在范围内

3.5.5 PC-lint 的应用举例

1. 编写源程序

首先在 VC 6.0 中编写源程序,程序代码如下。

```
1:
2: char * report( short m, short n, char * p )
3: {
4:     int result;
5:     char * temp;
6:     long nm;
7:     int i, k, kk;
8:     char name[11] = "Joe Jakeson";
9:
10:    nm = n * m;
11:    temp = p == "" ? "null" : p;
12:    for( i = 0; i < m; i++)
13:      { k++; kk = i; }
14:    if( k == 1 ) result = nm;
15:      else if( kk > 0 ) result = 1;
16:      else if( kk < 0 ) result = -1;
17:    if( m == result ) return temp;
18:      else return name;
19: }
```

然后对其进行编译。编译时会提示第 8 行数组下标越界。

2. 使用 PC-lint 进行检查

如果 PC-lint 已经集成到 VC 6.0 中,可以直接单击菜单栏中的 Tools→PC-lint Current File 命令(如图 3-38 所示),即可对当前的程序文件进行 PC-lint 检查。

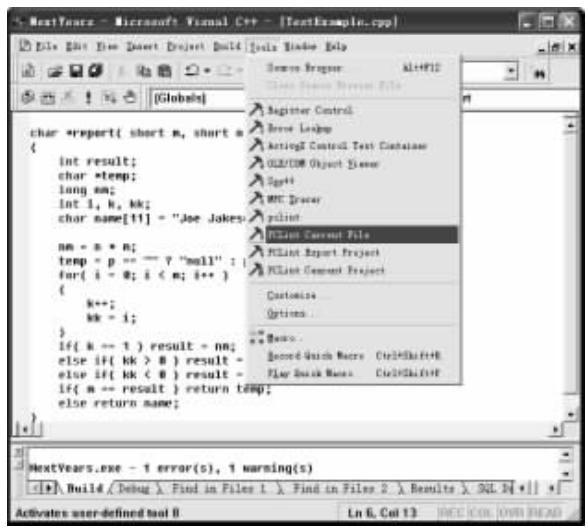

图 3-38 执行 PC-lint Current File 命令

执行完后，PC-lint 将显示详细的出错信息，如图 3-39 所示。

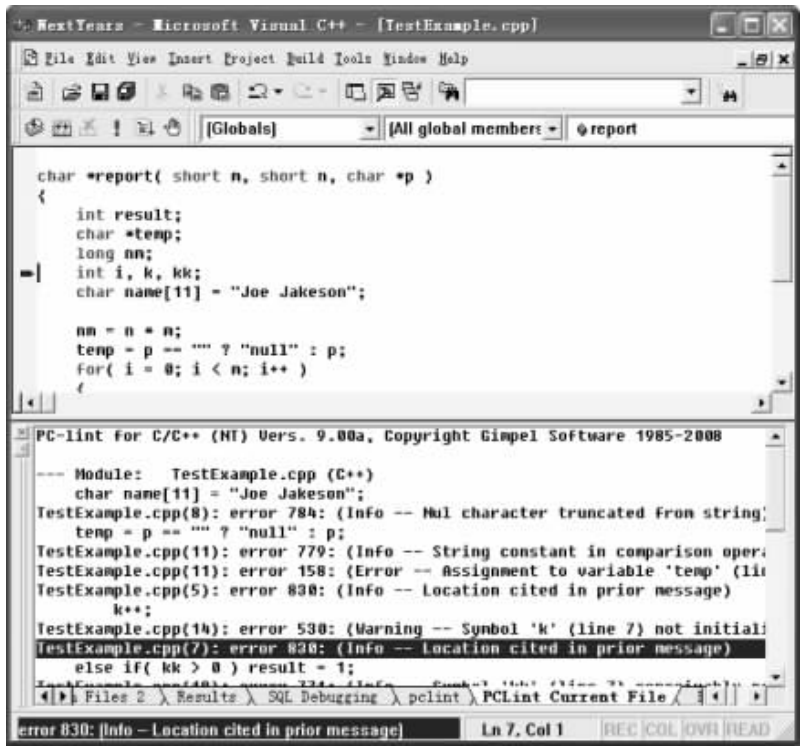

图 3-39 错误信息

其中，第 8 行向 name 数组赋值时丢掉了结尾的 nul 字符；第 11 行的比较有问题；第 14 行的变量 k 没有初始化，第 15 行的 kk 可能没有被初始化，第 22 行的 result 也有可能没有被初始化，第 23 行返回的是一个局部对象的地址。

详细的错误信息如下。

```
C-lint for C/C++(NT) Vers. 9.00a, Copyright Gimpel Software 1985-2008

-- - Module: TestExample.cpp (C++)
    char name[11] = "Joe Jakeson";
TestExample.cpp(8): error 784: (Info -- Nul character truncated from string)
    temp = p == "" ? "null" : p;
TestExample.cpp(11): error 779: (Info -- String constant in comparison operator '==')
TestExample.cpp(11): error 158: (Error -- Assignment to variable 'temp' (line 5) increases
capability)
TestExample.cpp(5): error 830: (Info -- Location cited in prior message)
        k++;
TestExample.cpp(14): error 530: (Warning -- Symbol 'k' (line 7) not initialized -- - Eff.
C++3rd Ed. item 4)
TestExample.cpp(7): error 830: (Info -- Location cited in prior message)
    else if( kk > 0 ) result = 1;
TestExample.cpp(18): error 771: (Info -- Symbol 'kk' (line 7) conceivably not initialized -
- - Eff. C++3rd Ed. item 4)
```

```
TestExample.cpp(7): error 830: (Info -- Location cited in prior message)
    if( m == result ) return temp;
TestExample.cpp(20): error 644: (Warning -- Variable 'result' (line 4) may not have been
initialized -- - Eff. C++3rd Ed. item 4)
TestExample.cpp(4): error 830: (Info -- Location cited in prior message)
    else return name;
TestExample.cpp(21): error 604: (Warning -- Returning address of auto variable 'name')
}

TestExample.cpp(22): error 783: (Info -- Line does not end with new - line)
TestExample.cpp(22): error 952: (Note -- Parameter 'm' (line 2) could be declared const -- -
Eff. C++3rd Ed. item 3)
TestExample.cpp(2): error 830: (Info -- Location cited in prior message)
}
TestExample.cpp(22): error 952: (Note -- Parameter 'n' (line 2) could be declared const -- -
Eff. C++3rd Ed. item 3)
TestExample.cpp(2): error 830: (Info -- Location cited in prior message)

 -- - Global Wrap - up

 error 900: (Note -- Successful completion, 16 messages produced)
Tool returned code: 16
```

3.6 代码静态测试实验

1. 实验目的

(1) 掌握静态代码分析技术；

(2) 使用静态测试工具进行代码静态检查。

2. 实验环境

Windows 环境，Checkstyle，Cppcheck 或其他静态测试工具，Office 办公软件。

3. 实验内容

1) 题目一：选择排序

设计一个选择排序算法，将输入的一组数据按从小到大的顺序进行排序。

2) 题目二：三角形问题

一个程序读入三个整数。把此三个数值看成是一个三角形的三个边。这个程序要打印出信息，说明这个三角形是三边不等的、是等腰的、还是等边的。

3) 题目三：日期问题

程序有三个输入变量 month、day、year(month、day 和 year 均为整数值，并且满足：1≤month≤12 和 1≤day≤31)，分别作为输入日期的月份、日、年份，通过程序可以输出该输入日期在日历上隔一天的日期。例如，输入为 2004 年 11 月 29 日，则该程序的输出为 2004 年

12月1日。

4. 实验步骤

(1) 根据题目要求,用Java或者C++语言实现各题目测试程序的编写,后续的实验将以这些程序作为测试对象用不同的测试方法来进行测试。

(2) 针对被测试代码选择一种静态测试工具,建立代码静态测试环境安装静态工具,如Checkstyle。

(3) 熟悉该测试工具的测试流程和业务功能。

(4) 针对待测试程序代码,实施静态测试。

(5) 针对待测试程序代码撰写静态测试报告。