Checkstyle4.3 中文手册

加入OpenDoc前的 预览版。

申 思维

1.0

本文根据Checkstyle4.3 英文官方文档而来。我打算发布成OpenDoc, 欢迎大家给我来信，多提意见！谢谢

目录

插图清单

表格清单

前言

Preface

Checkstyle是非常优秀的代码规范检查软件，可以大幅的提高代码质量， 当项目的开发人员比较多时，用它来统一代码风格是很有必要的。

本文的写作，是由于公司的质量管理部门对代码格式进行了要求。 在网上也没有发现有比较详细全面的中文文档。所以参考Checkstyle4.3的官方文档写就。

有个比较神奇的20%-80%规律是这样说的：一本书，用的最多的只是20%的内容，它的出现几率是80%； 而剩下的80%内容，被使用的不到20%。这个规律也同样适用在其他东东上。只是数据上稍有差异。 所以我特意安排了*第 4 章 用的最多的20%功能*，作为典型的使用方法。

对于赶时间的朋友，也可以直接看*第 2 章 N分钟入门*，可以让你在最快的时间内入门。 对于时间充沛的朋友，建议多看看文档。因为作者一再的强调“it is worth reading the documentation”。

*第 5 章 在Ant中使用Checkstyle*说明了在ant下的用法。*第 6 章 在Eclipse中使用Checkstyle*说明了Eclipse的插件Checkclipse的用法。

对于初次接触代码规范的朋友，我安排了*第 3 章 常用的检查*，里面是个人以为满足大多数公司要求的检查，包括一个配置文件。

*第 7 章 各种检查*是各种检查的详细用法，读起来比较枯燥，建议象查字典那样有需要时翻阅，所以放在最后。

> **欢迎意见**
>
> 为了加入OpenDoc , 欢迎各位朋友指出文档中的任何错误和不足，也可以给我任何意见。请Email给我：shensiwei(at)sina.com

希望本文对您有用。谢谢！

第 1 章 介绍

Introduction

目录

1.1. 概述

Checksytle 是一款代码格式检查工具。它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，良好的程序风格等等。如果你的项目经理开会时说，“我希望我们写出来的代码就象一个人写的！”时，用Checkstyle绝对是正确选择。:)

本文档就是在4.3的基础上完成。截止到2008-02-22，最新的版本是4.4。

需要强调的是，Checkstyle只能做检查，而不能做修改代码。

提醒

想修改代码格式，请使用Jalopy. 它和Checkstyle配合使用非常合适。

1.2. 特点

Checkstyle的配置性极强，你可以只检查一种规则，也可以检查三十，四十种规则。可以使用Checkstyle自带的规则， 也可以自己增加检查规则。（这点跟Ant自定义target比较象）

支持几乎所有主流IDE，包括 Eclipse , IntelliJ, NetBeans, JBuilder 等11种。

1.3. 下载

最新的发布版本在： 这里

4.4使用了SVN，关闭了CVS。SVN在 这里

各种插件下载，见 Checkstyle主页 中的列表。

第 2 章 N分钟入门

extreme learning

让您在几分钟之内了解Checkstyle的大致用法。适合赶时间的朋友。假设您已经安装好了Checkstyle的Eclipse插件。

测试

欢迎在阅读下一行之前，记录下当前的时间，然后在读完本节之后，算算您用了多少时间。 如果方便，请将您用的时间告诉我，谢谢:)

1. 首先，我们建立一个eclipse的项目：test_checkstyle。包含一个源文件夹：src，一个目标生成文件夹 eclipse_build. 如图 2.1 “测试如何使用checkstyle的项目”所示。

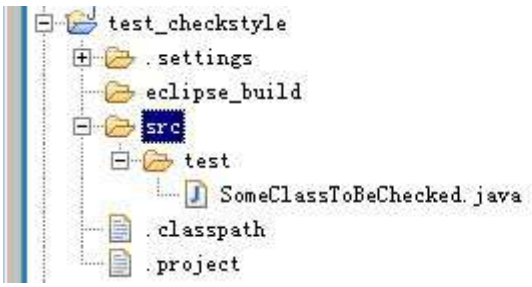

图 2.1. 测试如何使用checkstyle的项目

2. 在项目中开启Checkstyle: 打开该project的属性，点中左侧的Checkclipse后，将"Enable Checkstyle"前面打上勾。 如图 2.2 “开启Checkstyle”所示。

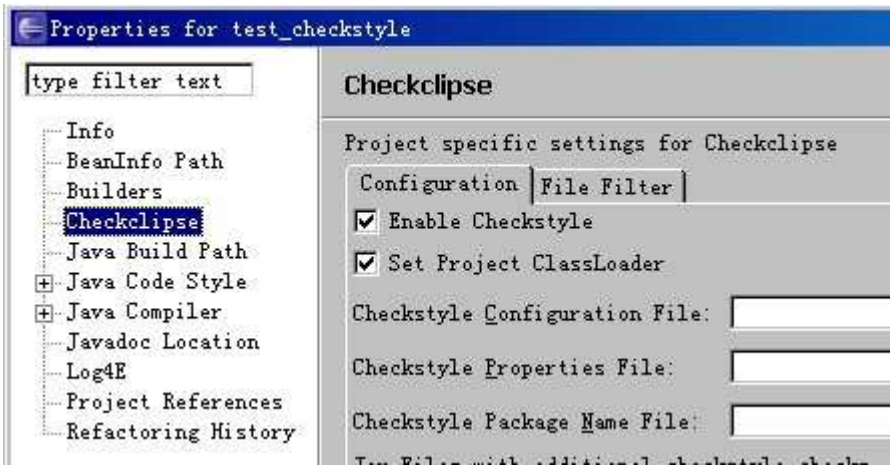

图 2.2. 开启Checkstyle

3. 建立一个测试用的Class: 比如SomeClassToBeChecked，内容如下：

```
/*
 * Copyright (c) 2001-2008 Beijing BidLink Info-Tech Co., Ltd.
 * All rights reserved.
 * Created on 2008-2-22
 *
 * $Id: learn_in_5_min.xml,v 1.3 2008/03/03 03:43:44 Administrator Exp $
 */
package test;

public class SomeClassToBeChecked {

}
```

只有一个头部注释，没有方法，啥啥都没有。

4. 用Checkstyle检查它：右键点项目名，选择"Build Project"，会把src文件夹进行编译，把class文件放到eclpise_build中。 结束之后，我们可以看到：图 2.3 “代码窗口中的错误提示”中的代码第10行处，有一个叹号，把鼠标移上去就会出现 提示"Missing a Javadoc comment."

图 2.3. 代码窗口中的错误提示

同时，在Problems窗口中也有提示，如图 2.4 “Problems窗口中的错误提示”所示。

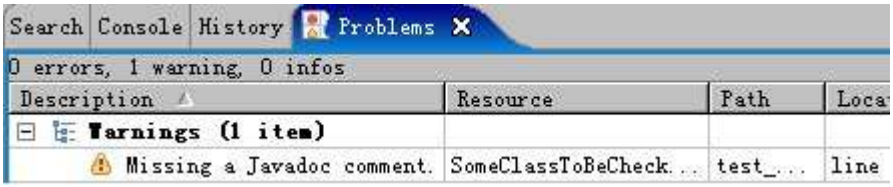

图 2.4. Problems窗口中的错误提示

5. 修改代码：既然提示说缺少了Javadoc注释，我们就把它加上。 如图 2.5 “增加了class的注释后的效果图”所示。

图 2.5. 增加了class的注释后的效果图

然后重新编译，可以看出，Warning没有了。检查通过。

> **注意**
>
> 上面的很简单是吧？恩，我也这么觉得。但别走！如果以为这样就可以用Checkstyle， 那你就错了。你可以试一下用同样的方式来编译一个项目，会发现根本是Warning满天飞。为什么？因为Checkstyle自带的检查非常变态， 随便一个项目都可以弄出几千个Warning。所以，想用它，一定要使用自己的定制检查。:)

6. 定制检查：Checkstyle没有图形化的定制器，所以需要手工修改配置文件。比如，我们的代码需要符合下列规则：

   - 长度方面：文件长度不超过1500行，每行不超过120个字，方法不超过60行.
   - 命名方面：类名不能小写开头，方法名不能大写开头，常量不能有小写字母。
   - 编码方面：不能用魔法数(Magic Number），if最多嵌套3层。

   那么，我们的检查配置文件（如命名成 my_check.xml ） 应该是这样的：

```
<?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">
```

```
    <module name="TreeWalker">

        <!-- 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120个字-->
        <module name="LineLength">
                    <property name="max" value="120"/>
        </module>
        <!-- 方法不超过60行 -->
        <module name="MethodLength">
            <property name="tokens" value="METHOD_DEF"/>
            <property name="max" value="60"/>
        </module>

        <!-- 命名方面的检查，它们都使用了Checkstyle默认的规则。 -->
        <!-- 类名(class 或interface) 的检查 -->
        <module name="TypeName"/>
        <!-- 方法名的检查 -->
        <module name="MethodName"/>
        <!-- 常量名的检查 -->
        <module name="ConstantName"/>

        <!-- 编码方面的检查 -->
        <!-- 不能用魔法数 -->
        <module name="MagicNumber"/>
        <!-- if最多嵌套3层 -->
        <module name="NestedIfDepth">
            <property name="max" value="3"/>
        </module>

    </module>
</module>
```

可以看出，想增加一个检查，就是增加一个<module/>结点。具体的结点内容在后面的文档都会写明。

7. 让Checkstyle使用指定的检查配置文件：打开项目属性，在Checkclipse中的"Checkstyle Configuration File" 一栏中 选定我们的配置文件，然后确定。如图 2.6 “使用自定义的Checkstyle配置文件”所示。

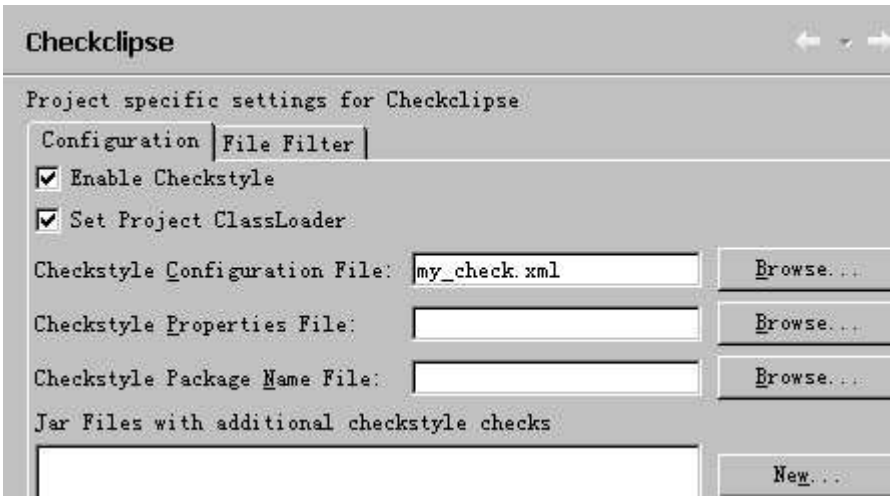

图 2.6. 使用自定义的Checkstyle配置文件

然后重新编译项目，就会发现，Checkstyle的规则如我们所愿：只检查我们在文件中配置的几项。并且它们是以"Error"级别进行提示，而不是默认检查时出现的"Warning"级别。 比如，我们把一个方法中，增加4层嵌套（共5个if），并将方法名大写，就会出现图 2.7 “定制

配置的检查结果”:

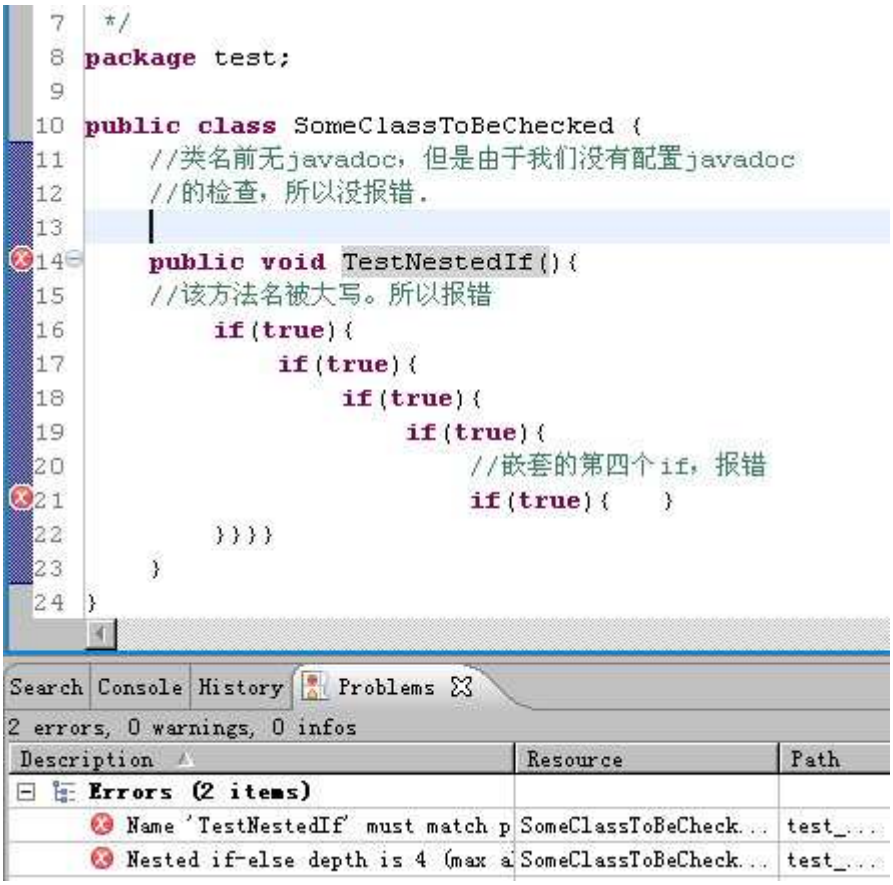

图 2.7. 定制配置的检查结果

可以看到，出现了两个Error: 方法名的"Name xx must match pattern..." 和if嵌套的"Nested if-else depth is 4..."。 把它们都改过来，程序就好了。

8. 代码的修正：依照上面的例子，把方法名小写，if循环嵌套3层，然后重新编译，OK。如：图 2.8 “修正后的定制配置的检查结果”

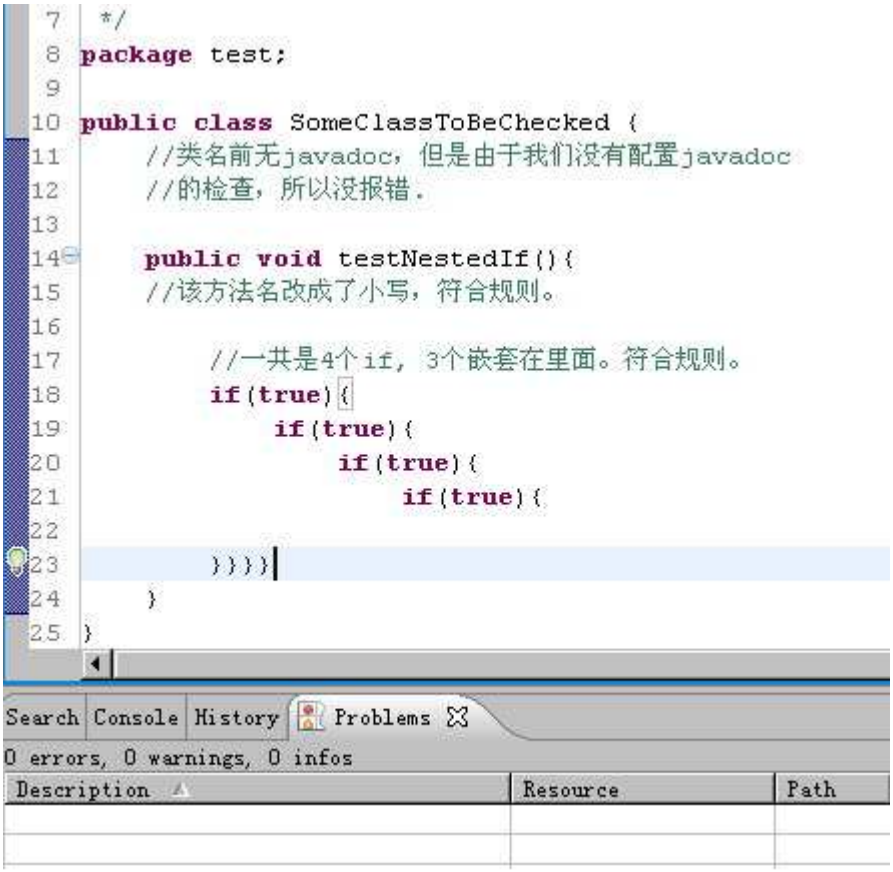

图 2.8. 修正后的定制配置的检查结果

到这里就结束了。看明白了吗？欢迎给我意见，以及看到这里所用的时间。:)

想在N分钟内了解ant下checkstyle的使用方法，请看第 5.1 节 “N分钟极速入门”

第 3 章 常用的检查

目录

Checkstyle自带了两个配置文件：checkstyle_checks.xml 和 sun_checks.xml。前者是checkstyle作者定义的，后者是严格符合Sun编码规范的。 可惜它们的检查太过严格，任何一个项目都会搞出上千个Warning来。

所以这里提供了一个配置文件，包含了比较常用的检查，去掉了对空格位置，大括号位置等国内不注重的内容， 建议个人使用。

如果是公司使用，建议把它再精简些。因为对公司来说，这个配置文件还是比较苛刻的。

3.1. 典型的配置文件

下面是一个典型的checkstyle配置文件，应该适合于大多数情况的要求。每个检查前的注释说明了它的作用。本文的PDF文档 无法正常显示其中的汉字，建议本节使用HTML版查看。:)

```
<!DOCTYPE module PUBLIC
    "-//Puppy Crawl//DTD Check Configuration 1.2//EN"
    "http://www.puppycrawl.com/dtds/configuration_1_2.dtd">
<module name="Checker">
```

```
    <!-- 重复代码的检查，超过8行就认为重复，UTF-8格式
            本检查一定要放在"TreeWalker"节点前，否则在
            Checkclipse中会无法使用。(在ant下可以)
            -->
    <module name="StrictDuplicateCode">
        <property name="min" value="8"/>
        <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"/>

            <!-- 检查所有方法的javadoc，可以不声明RuntimeException -->
            <module name="JavadocMethod">
                <property name="allowUndeclaredRTE" value="true"/>
            </module>
            <!-- 检查某个变量的javadoc -->
            <module name="JavadocVariable"/>

            <!-- 命名方面的检查，它们都使用了Sun官方定的规则。 -->
            <!-- 类名(class 或interface) 的检查 -->
            <module name="TypeName"/>
            <!-- 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"/>


            <!-- 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120个字-->
            <module name="LineLength">
                <property name="max" value="120"/>
            </module>
            <!-- 方法不超过30行 -->
            <module name="MethodLength">
               <property name="tokens" value="METHOD_DEF"/>
               <property name="max" value="30"/>
            </module>
            <!-- 方法的参数个数不超过3个。 -->
            <module name="ParameterNumber">
               <property name="max" value="3"/>
            </module>


            <!-- 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"/>
<!-- 每个类都实现了equals()和hashCode() -->
<module name="EqualsHashCode"/>
<!-- 不许使用switch -->
<module name="IllegalToken">
        <property name="tokens"
        value="LITERAL_SWITCH"/>
</module>
<!-- 不许内部赋值 -->
<module name="InnerAssignment"/>
<!-- 绝对不能容忍魔法数 -->
<module name="MagicNumber"/>
<!-- 循环控制变量不能被修改 -->
<module name="ModifiedControlVariable"/>
<!-- 多余的throw -->
<module name="RedundantThrows"/>
<!-- 不许使用未被简化的条件表达式 -->
<module name="SimplifyBooleanExpression"/>
<!-- 不许使用未被简化的布尔返回值 -->
<module name="SimplifyBooleanReturn"/>
<!-- String的比较不能用!= 和 == -->
<module name="StringLiteralEquality"/>
<!-- if最多嵌套3层 -->
<module name="NestedIfDepth">
    <property name="max" value="3"/>
</module>
<!-- try最多被嵌套1层 -->
<module name="NestedTryDepth"/>
<!-- 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>
<!-- JUnitTestCase 的核心方法存在。 -->
<module name="JUnitTestCase"/>
<!-- 一个方法中最多有3个return -->
<module name="ReturnCount">
    <property name="max" value="3"/>
</module>
<!-- 不许对方法的参数赋值 -->
<module name="ParameterAssignment"/>
<!-- 不许有同样内容的String -->
<module name="MultipleStringLiterals"/>
<!-- 同一行不能有多个声明 -->
<module name="MultipleVariableDeclarations"/>


<!-- 各种量度 -->
<!-- 布尔表达式的复杂度，不超过3 -->
<module name="BooleanExpressionComplexity"/>
<!-- 类数据的抽象耦合，不超过7 -->
<module name="ClassDataAbstractionCoupling"/>
<!-- 类的分散复杂度，不超过20 -->
<module name="ClassFanOutComplexity"/>
<!-- 函数的分支复杂度，不超过10 -->
<module name="CyclomaticComplexity"/>
<!-- NPath复杂度，不超过200 -->
<module name="NPathComplexity"/>


<!-- 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>

                <!-- 不许使用与代码同行的注释 -->
                <module name="TrailingComment"/>

    </module>



        <!-- 检查翻译文件      -->
        <module name="Translation"/>

</module>
```

第 4 章 用的最多的20%功能

Core 20%

这里列出了个人以为的20%的最常用功能。欢迎您把自己认为的20%发过来，我会进行一个统计，及时更新本节。

- 命名符合规范

  类，方法，成员变量等等的命名，一般要遵循Sun编码规范。

- 编码中的长度问题

  类，方法的长度不应该过大。

- 编码习惯的检查

  检查代码中是否有过多的if嵌套，魔法数，switch丢失的default, 复杂的条件表达式等。

- 重复的代码

  如果两个代码段出现了一定行数的严格重复，就判定它们已经重复。

第 5 章 在Ant中使用Checkstyle

目录

如果您赶时间，请看第 5.1 节 “N分钟极速入门”。:)

5.1. N分钟极速入门

（请打开一个秒表记时，谢谢）

需要"checkstyle-all-4.3.jar"，该jar文件包含了checkstyle所用的几乎所有类。

需要有一个指定的配置文件，比如当前项目中的"./config/my_check.xml"。

另外，把checkstyle-all.jar放在一个合适的目录中，比如"./lib"。

最后，设置Ant的输出文件夹为"./ant_build" 如图 5.1 “在Ant环境下Checkstyle的所须文件”所示。

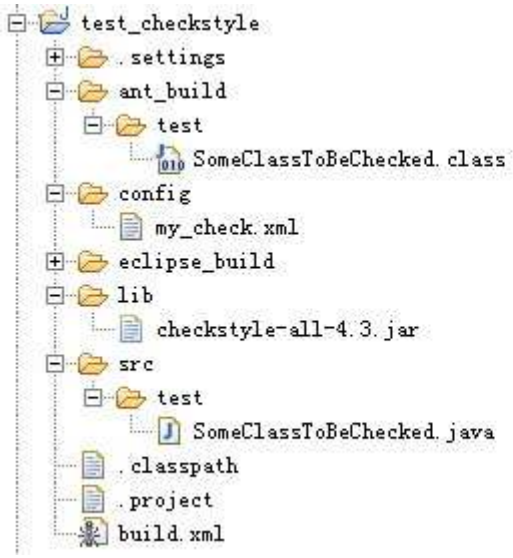

图 5.1. 在Ant环境下Checkstyle的所须文件

ant的配置文件中需要指定一个taskdef来定义checkstyle任务, 然后在checkstyle任务中指定配置文件(config属性)， 和需要被检查的文件夹(fileset)。本例使用的ant文件是这样的：

```
<?xml version="1.0" ?>
<project>
    <taskdef resource="checkstyletask.properties"
         classpath="./lib/checkstyle-all-4.3.jar"/>
    <target name="my_check">
        <checkstyle config="config/my_check.xml">
            <fileset dir="src" includes="**/*.java"/>
        </checkstyle>
    </target>
</project>
```

如果checkstyle检查通过，则会看到"BUILD SUCCESSFUL"， 如图 5.2 “Ant下Checkstyle检查正确的结果”所示。

图 5.2. Ant下Checkstyle检查正确的结果

如果checkstyle检查到出错，就会输出错误信息，包括所有检查到的错误。比如，我们把一个方法的长度超过60，方法名大写，就会看到出错信息，并且"BUILD FAILED" 如图 5.3 “Ant下Checkstyle检查错误的结果”所示。

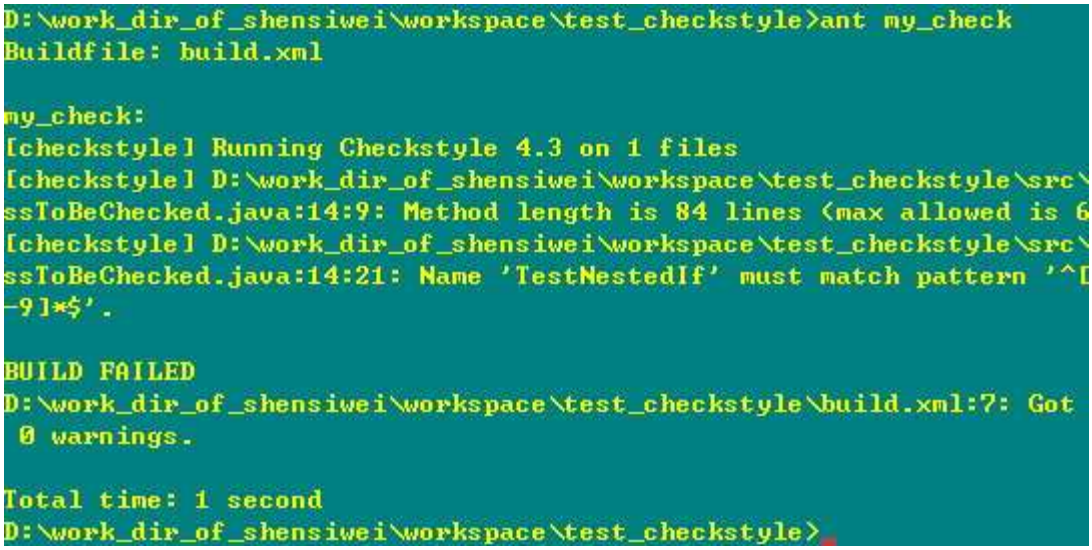

图 5.3. Ant下Checkstyle检查错误的结果

本节结束，欢迎您把看明白本节所用的时间通过Email告诉我。因为我想知道标题中的 "N" 是多少。 :)谢谢。

5.2. 安装与配置

其实本节内容已经被包含在了第 5.1 节 “N分钟极速入门”中，就是拥有checkstyle-all-4.3.jar 这个文件，并且在 ant的配置文件中进行task声明，如：

```
<taskdef resource="checkstyletask.properties"
     classpath="./lib/checkstyle-all-4.3.jar"/>
```

然后通过<checkstyle/>任务来运行。

5.3. 典型例子

本节假定使用checkstyle任务的前提条件都已满足。

使用my_check.xml文件，对src目录下的所有java文件进行检查:

```
<checkstyle config="config/my_check.xml">
    <fileset dir="src" includes="**/*.java"/>
</checkstyle>
```

使用my_check.xml文件，对src目录下的所有java文件进行检查，并且把出错的信息分别写到两个文件中:ant_build目录下的 checkstyle_errors.txt 和checkstyle_errors.xml中:

```
<checkstyle config="config/my_check.xml">
    <fileset dir="src" includes="**/*.java"/>
    <formatter type="plain" toFile="ant_build/checkstyle_errors.txt"/>
    <formatter type="xml" toFile="ant_build/checkstyle_errors.xml"/>
</checkstyle>
```

值得一提的是，xml格式的文件里面把错误的信息格式弄的很清晰，调试起来比较方便。如果需要的话可以比较方便的扩展到诸如cruisecontrol里。

使用定义了package的文件：

```
<checkstyle config="config/my_check.xml"
    packageNamesFile="myPackageNames.xml"
    file="src/test/SomeClass.java"/>
```

5.4. checkstyle任务的参数

checkstyle任务的参数如表 5.1 “checkstyle任务属性表”所示。

表 5.1. checkstyle任务属性表

名字	描述	是否必须
file	被检查的文件。	一个file或者fileset
config	Checkstyle的配置文件。配置文件的使用方法见：第 7.1 节 “如何配置检查”	config或configURL必具其一
configURL	指定了Checkstyle配置文件的URL。用法	config或configURL必具其一
properties	定义了ant使用到的属性的文件。	否
packageNamesFile	定义了Checkstyle配置文件中检查的package name的文件。	否
failOnViolation	有violation时是否继续检查，默认是"true"	否
failureProperty	The name of a property to set in the event of a violation.	否
maxErrors	停止build前允许出现的"Error"的最大数目。默认是"0"	否
maxWarnings	停止build前允许出现的"Warning"的最大数目。默认是"2147483647"，也就是Integer.MAX_VALUE.	否
classpath	类路径，默认是当前使用的classpath。	否
classpathref	类路径的引用。	否

5.5. 可以嵌套的ant元素

checkstyle任务中可以前台以下元素: <filese>, <classpath>, <formatter> 以及<property>

formatter元素的属性如表 5.2 “formatter元素的属性”所示。

表 5.2. formatter元素的属性

名字	描述	是否必须
type	结果的输出方式，可用的值是： plain :使用了 DefaultLogger xml : 使用了XMLLogger 默认是 plain.	否
toFile	输出到的文件。默认是标准输出（控制台）	否
useFile	是否把结果输出到文件中。 true或false. 默认是"true"	否

第 6 章 在Eclipse中使用Checkstyle

目录

6.1. 下载和安装

下载在Checkstyle的Checkstyle主页，有一个插件列表。 截止到2008-02-26，Eclipse插件的可用的地址是：EclipseCS 和 Checkclipse

本节讲解的是后者，也就是Checkclipse。

Checkclipse安装的方法同其他的Eclipse插件一样。不清楚的朋友可以看这篇文章：Eclipse基础——使用links方式安装Eclipse插件。links方式 适用于Eclipse3.0, 3.1, 3.2 和 Europa

6.2. 配置方法

当Checkclipse成功安装后，在"window->Preferences"中可以看到checkclipse的选项， 如图 6.1 “成功安装Checkclipse后的Preferences窗口”所示。

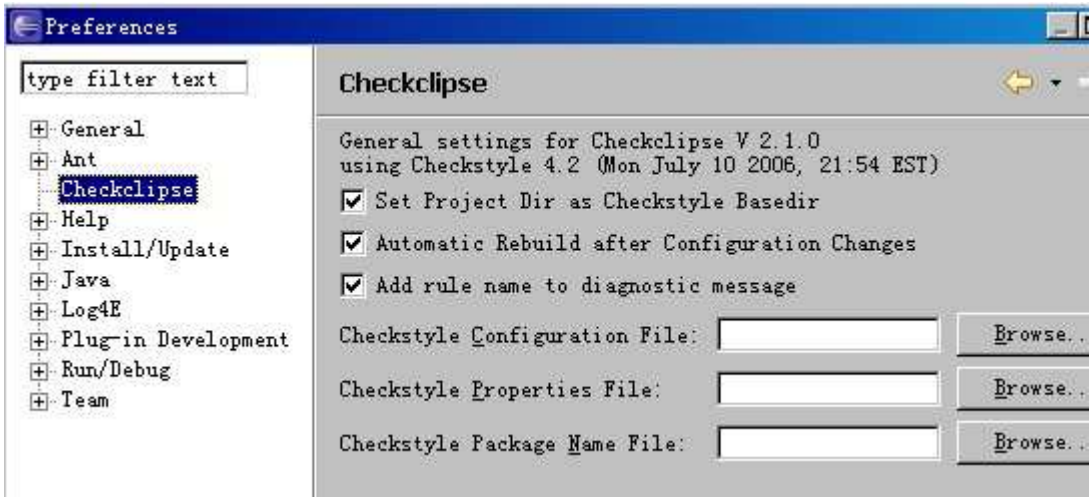

图 6.1. 成功安装Checkclipse后的Preferences窗口

Checkclipse有两个渠道可以进行配置，一个是全局的，一个是单个项目(Project)的。全局的可以在整个Eclipse的workbench中生效， 而单个项目的配置可以在指定的项目中生效，它优先于全局的配置。

对于单个项目：右键点某个项目，然后选择"Properties"就可以看到Checkclipse的窗口。在"Configuration"标签中，"Enable Checkstyle"一行前面打勾， 然后在"Checkstyle Configuration File:" 一行中选择你的Checkstyle配置文件就可以了。

如图 6.2 “配置单个项目的Checkclipse”所示。

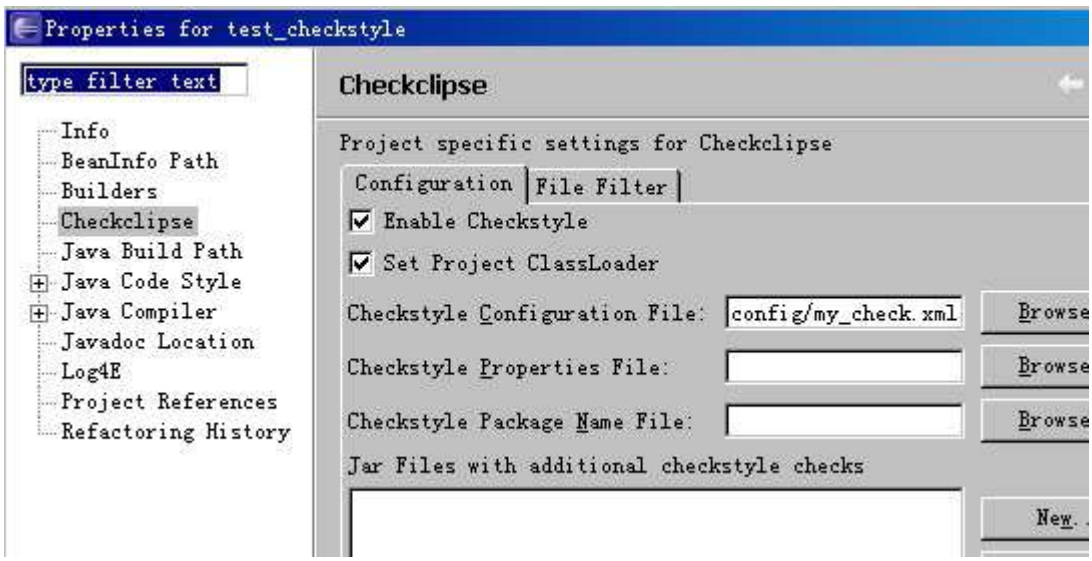

图 6.2. 配置单个项目的Checkclipse

对于全局的设置："window->preferences"就可以看到。设置方法跟单个项目的设置是一样的。

经过上面的设置，Checkclipse就可以使用了。如果你想设置需要被检查的文件名，那么就在"File Filter"标签中修改被包含的文件。可以使用"Add","Remove","Change"等按钮进行编辑。 Included Resources 中显示了被检查的文件清单。

如图 6.3 “设置单个项目的Checkclipse的文件过滤器(file filter)”所示。

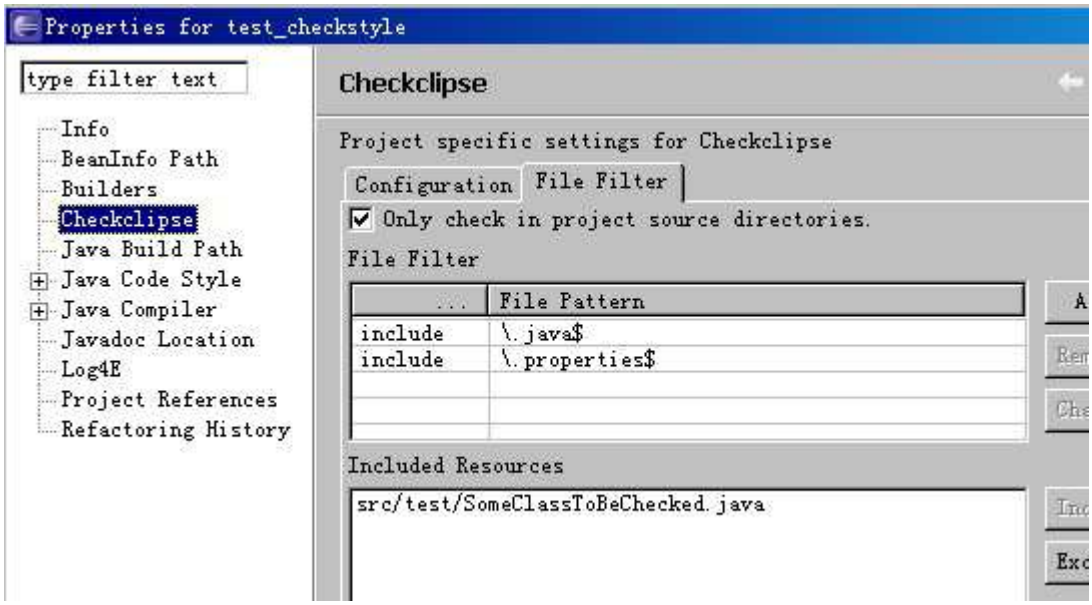

图 6.3. 设置单个项目的Checkclipse的文件过滤器(file filter)

注意

除非很有必要，否则不要改FileFilter。使用默认的就满足95%的情况。

官方帮助可以在Eclipse的"Help-> HelpeContents" 中的"Checkclipse - Checkstyle Plugin" 中找到。内容还是很详尽的。

6.3. 使用

Checkclipse的使用非常简单，它的每次检查是跟随Eclipse的"Build Project"一起进行的。如果有错误，会显示在"Problems"子窗口中。 另外，对于默认的检查，错误是Warning级的，对于设置了配置文件后的检查，错误都是Error级的。 看一下第 2 章 *N分钟入门*就肯定明白了：）

查看是哪个检查发现的错误：在出错信息提示的结尾的"[]"中有显示。比如，程序中有两个相同字符串的变量"hehe"，检查时就会 出现“重复的字符串串”报错，显示"...[MultipleStringLiteralsCheck]"， 如：图 6.4 “出错信息中的检查名”所示。

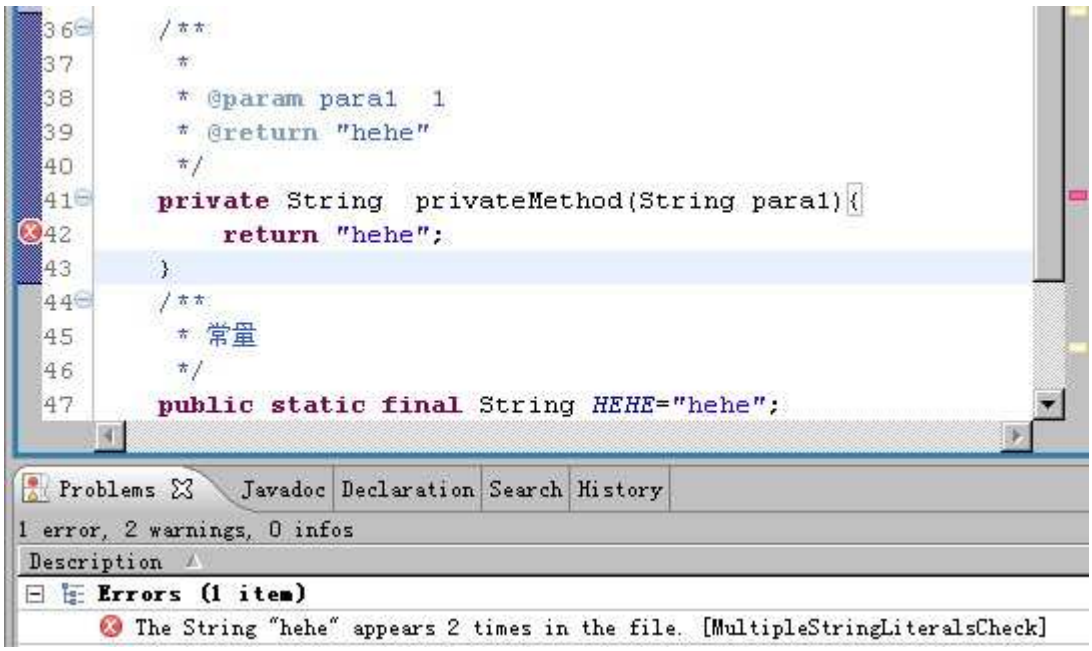

图 6.4. 出错信息中的检查名

如果在"Problems"窗口中没有提示"Checkstyle"错误，而在编辑器中又确实存在该错误，那么很有可能是"Problems"窗口的过滤器被修改了。 请进入到配置"Problems"过滤器窗口，在右侧的"Show items of type"列表中的"Checkclipse Problem Marker"前打勾。 如：图 6.5 “Problems的过滤器中配置Checkclipse。”所示。

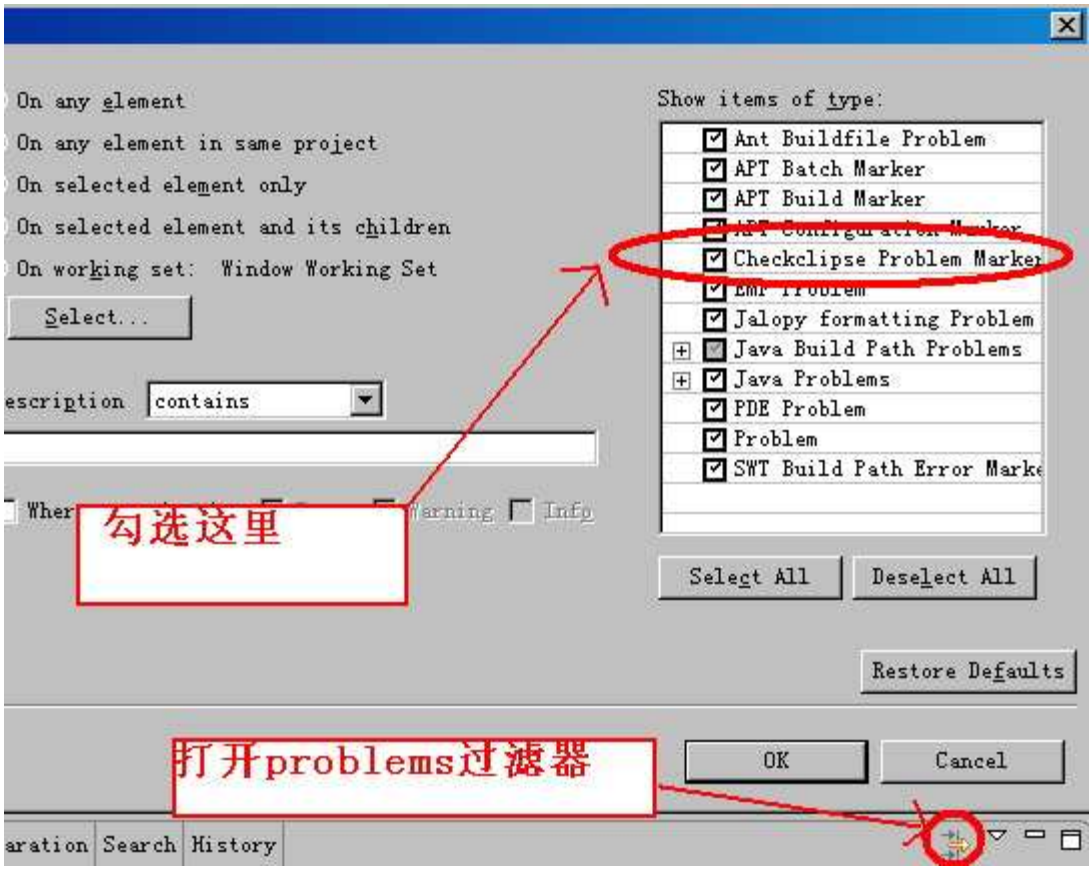

图 6.5. Problems的过滤器中配置Checkclipse。

右键菜单中的checkstyle检查：对任何打开的java文件，点右键，就会弹出一个菜单， 在"Run As"那一组中，就有Checkstyle的选项， 如图 6.6 “右键菜单中的checkstyle选项”所示。

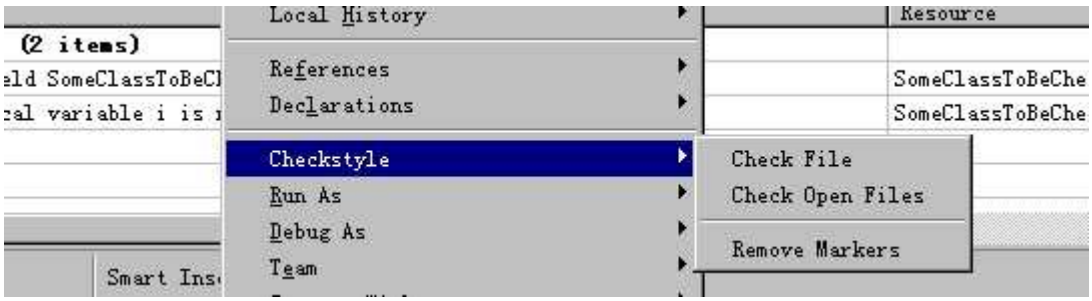

图 6.6. 右键菜单中的checkstyle选项

可以看出，"Checkstyle"选项有三个子项：

- Check File: 只检查当前文件。
- Check Open Files: 检查当前Eclipse中打开的文件。
- Remove Markers : 当前文件如果存在Checkstyle检查出的错误，就清除这些错误标记。

6.4. 常见问题

请EMAIL告诉我。：）

第 7 章 各种检查

checks

目录

本节看起来比较烦冗，建议使用哪些东东，就看哪个小节。就象字典那样用。

我挑的是看起来比较常用的检查。一些另外的，比如空格的约定，J2EE，EJB的检查，没有包括在这里。有需要的朋友请看官方文档。

注意

除非特别说明，否则检查代码的位置都要放在 TreeWalker 这个结点下面。

7.1. 如何配置检查

Checkstyle中的各种检查的增加，都是以增加配置文件的内容来实现的。比如想检查方法的长度，就需要在<module name="TreeWalker">节点下增加：

```
<module name="MethodLength">
    <property name="tokens" value="METHOD_DEF"/>
    <property name="max" value="60"/>
</module>
```

如果没有特殊说明，一般是增加在<module name="TreeWalker">结点下。

如果一个module节点的属性可以有多个值备选，那么这些值之间一般用","连接。比如"tokens = LITERAL_STRING, LITERAL_INT"

是否需要我按照表示的形式来说明这些属性？也就是按照官方文档的格式？

7.2. JavaDoc注释

Javadoc Comments

7.2.1. 类和接口的javadoc

检查所有的类和接口：

```
<module name="JavadocType"/>
```

也可以只检查protected 和 public 的类：

```
<module name="JavadocType">
        <property name="scope" value="public"/>
        <property name="tokens" value="CLASS_DEF"/>
</module>
```

其中的"scope"属性指定了检查可见的范围，也就是说，如果是"private"，那么就可以见"private, protected, public"三种， 如果赋值为"protected"那么就只检查"protected,public"两种，如果赋值为"public"，就只检查"public"的类或接口。默认是"private"。

"tokens"属性指定了是检查类还是接口还是都检查。默认是都检查。可选的值是"CLASS_DEF"（检查类）和"INTERFACE_DEF"。（检查接口）

7.2.2. 方法的javadoc

```
```

7.2.3. 方法的javadoc

检查所有方法的javadoc：

```
<module name="JavadocMethod"/>
```

只检查public方法的javadoc：

```
<module name="JavadocMethod">
        <property name="scope" value="public"/>
</module>
```

其中的"scope"属性指定了检查可见的范围，也就是说，如果是"private"，那么就可以见"private, protected, public"三种， 如果赋值为"protected"那么就只检查"protected,public"两种，如果赋值为"public"，就只检查"public"的类或接口。默认是"private"。

注意

如果被检查的方法是某个interface方法的实现，那么该方法的javadoc是：

```
/**
 * @inheritDoc
 */
public void someMethod(){
        //implementation ...
}
```

也就是说，不用把接口的javadoc COPY过来，一样可以通过本检查。（也一样能生成正确的javadoc）

javadoc中的参数等必须有说明

比如，这样的参数说明会被检查认为是没有：

```
/**
 * @param somePara
 */
public void someMethod(String somePara){
...
```

因为它没有对该参数的说明，应该加上。如：

```
/**
 * @param somePara This is the description...
 */
public void someMethod(String somePara){
...
```

7.2.4. 变量的javadoc

检查类或接口的变量的javadoc. 这样：

```
<module name="JavadocVariable"/>
```

7.3. 命名约定

Naming Conventions

这里可以对java中的各种命名进行检查，比如方法名，变量名，常量名等等。

7.3.1. 模块一览

由于相互之间很类似，所以放在一个表中说明。这些检查使用的<module> 的"name"属性见下表， 它们具有相同的"format"属性，需要赋予正则表达式，默认值已经被设置好了，是与Sun编码规范相同的。 如表 7.1 “命名约定检查模块一览表”所示。

表 7.1. 命名约定检查模块一览表

模块的"name"属性	检查范围	"format"属性的默认值
AbstractClassName	抽象类	^Abstract.*$\|^.*Factory$
ConstantName	常量(static , final 字段)	^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$
LocalFinalVariableName	局部的final变量，包括catch中的参数	^[a-z][a-zA-Z0-9]*$
LocalVariableName	局部的非final型的变量，包括catch中的参数	^[a-z][a-zA-Z0-9]*$
MemberName	非static型变量	^[a-z][a-zA-Z0-9]*$
MethodName	方法名	^[a-z][a-zA-Z0-9]*$
PackageName	package名	^[a-z]+(\.[a-zA-Z_][a-zA-Z0-9_]*)*$
ParameterName	方法中的参数名	^[a-z][a-zA-Z0-9]*$
StaticVariableName	仅仅是static型的变量（不包括static final型） ^[a-z][a-zA-Z0-9]*$	

模块的"name"属性	检查范围	"format"属性的默认值
TypeName	类型(Class或Interface)名	^[A-Z][a-zA-Z0-9]*$

7.3.2. 注意

PackageName检查的format默认值是允许大写的包名的，建议设置成只能小写，如：

```
<module name="PackageName">
    <property name="format"
        value="^[a-z]+(\.[a-z][a-z0-9]*)*$"/>
</module>
```

TypeName的检查是可以通过修改"tokens"属性来指定检查Class还是Interface，默认是两者都检查。如果只想检查Interface，可以这样：

```
<module name="TypeName">
    <property name="tokens"
              value="INTERFACE_DEF"/>
</module>
```

其中，tokens属性的值是：CLASS_DEF 和 INTERFACE_DEF.

7.4. 文件头

Header

检查文件头区域是否包含指定的内容。可以在headerFile属性中指定一个外部文件的位置，该文件保存了文件头信息。也可以使用另外一种 方法：将文件头信息直接写在 header属性中。这样就可以不用外部文件。

属性 ignoreLines指定了头文件中可以不被检查的行数。这个属性对于包含版权日期信息的文件头很有用。比如下面的文件头：

```
/*
 * Copyright (c) 2001-2008 Some Company Co., Ltd.
 * All rights reserved.
 *
 * Created on 2007-10-8
 * $Id: checks.xml,v 1.8 2008/03/03 03:37:19 Administrator Exp $
 */
```

其中第2,6行包含的日期信息会发生变化。所以在检查时，最好通过设置 ignoreLines 属性把这两行忽略掉。如：

```
<module name="Header">
    <property name="ignoreLines" value="2, 6"/>
</module>
```

想严格对文件头进行检查，请把文件头定义在一个外部文件中，如：java_header_file.txt 然后使用"headerFile"属性：

```
<module name="Header">
    <property name="headerFile" value="java_header_file.txt"/>
    <property name="ignoreLines" value="2, 6"/>
</module>
```

7.5. Imports

Imports

7.5.1. import中避免星号"*"

AvoidStarImport

如果不希望有任何 import a.b.*; 这样的语句，请在配置文件中，增加：

```
<module name="AvoidStarImport"/>
```

"excluds"属性定义了被忽略的import。默认是空。比如：

```
<module name="AvoidStarImport">
   <property name="excludes" value="a.b.c,c.d"/>
</module>
```

这样就会允许"import a.b.c.*; import c.d.*; "。

7.5.2. 没用的import

以下几种import是没用/没意义的：

- 没有被用到。
- 重复的。
- import java.lang的。
- import 与该类在同一个package的。

进行检查：

```
<module name="UnusedImports"/>
```

7.6. 长度限制

FileLength

7.6.1. 文件长度

检查一个文件的行数不超过1500，如下：

```
        <module name="FileLength">
              <property name="max" value="1500"/>
        </module>
```

7.6.2. 每行长度

想规定一行的长度不朝过120个字母，如下：

```
        <module name="LineLength">
            <property name="max" value="120"/>
        </module>
```

7.6.3. 方法长度

使用"max"属性可以设置最多的行数，默认150，"countEmpty"是否对空白或"//"注释行记数，默认是"true"，"tokens"可以指定 检查的是一般方法还是构造方法，默认都检查，可用的值是"METHOD_DEF, CTOR_DEF"。

想规定一个方法的行数不超过30行，如下：

```
        <module name="MethodLength">
           <property name="tokens" value="METHOD_DEF"/>
           <property name="max" value="30"/>
        </module>
```

7.6.4. 方法的参数个数

"max"属性可以设置最多的参数个数，默认7，"tokens"属性指定检查的是一般方法还是构造方法，默认都检查。可用的值是"METHOD_DEF, CTOR_DEF"。

想规定一个方法的参数不超过5个，并不检查构造函数，请增加：

```
<module name="ParameterNumber">
   <property name="max" value="5"/>
   <property name="tokens" value="METHOD_DEF"/>
</module>
```

7.7. 空格

Whitespace

个人以为不太重要。

7.7.1. 方法名与左边圆括号之间

本检查约定方法名与左边圆括号之间不许出现空格。例如，下列的代码格式是错误的，在方法名"wrongStyleMethod"后面紧跟的是个空格，而不是"()"：

```
...
public void wrongStyleMethod (){
...
```

方法名与圆括号不是一行默认也是不允许的，如：

```
...
public void wrongStyleMethod
                        (String para1, String para2){
...
```

本检查只允许方法名后紧跟左边圆括号"("。

```
<module name="MethodParamPad"/>
```

7.7.2. 圆括号附近的空格

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。例如下面的代码会报错，

```
...
public void wrongStyleMethod( String para1, String para2 ){
...
```

把对应的空格去掉才是正确的：

```
...
public void correctStyleMethod(String para1, String para2){
...
```

这样检查：

```
<module name="ParenPad"/>
```

默认检查范围：构造函数的调用，方法的调用，左，右圆括号，父类构造函数的调用。

7.7.3. 类型转换中圆括号附近的空格

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。例如下面的代码就会报错：

```
Object myObject = ( Object )other;
```

正确的应该是：

```
Object myObject = (Object)other;
```

使用本检查：

```
<module name="TypecastParenPad"/>
```

7.7.4. 对"Tab"的检查

代码中不允许使用"tab"键。('\t') 因为在不同的编辑器下有的tab占8个空格（如windows记事本），有的占4个（如当前大多数IDE）。

使用本检查：

```
<module name="TabCharacter"/>
```

强烈建议

使用本检查前一定要先用代码格式化工具把"tab"转换成空格。否则满屏幕的"Error"会吓人一跳。:)

7.7.5. 特定符号后的空格

检查某些特定符号后是否有空格。如果没有，就报错。

详细属性如表 7.2 “WhitespaceAfter 属性列表”所示。

表 7.2. WhitespaceAfter 属性列表

名称	说明	类型	默认值
tokens	被检查的范围	"COMMA, SEMI, TYPECAST" 中的一个或多个。 COMMA:逗号，SEMI：分号，TYPECAST：类型转换圆括号。	COMMA, SEMI, TYPECAST

例如，规定逗号","后必须有空格：

```
<module name="WhitespaceAfter">
        <property name="tokens" value="COMMA"/>
</module>
```

7.8. 关键字

key word

7.8.1. 关键字的出现顺序

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。 这样：

```
<module name="ModifierOrder"/>
```

7.8.2. 多余的关键字

"tokens"属性定义了可以被检查的范围，包括："METHOD_DEF,VARIABLE_DEF, ANNOTATION_FIELD_DEF"。默认都检查，

```
<module name="RedundantModifier"/>
```

如果只想对一般方法做检查，这样：

```
<module name="RedundantModifier">
    <property name="tokens" value="METHOD_DEF"/>
</module>
```

7.9. 对区域(empty block)的检查

block

也就是大括号包含起来的区域：）

7.9.1. 空白区域

只要是{} 包含起来的，都是区域。比如if , else, catch 后面紧跟的，都是区域。可以对它们进行检查。默认是检查所有区域：

```
<module name="EmptyBlock"/>
```

7.9.2. 对左侧括号{ 的检查（略）

个人认为此检查无意义……欢迎听到你的意见。

7.9.3. 需要括号的区域

比如，只有一条语句的 if ，对于大括号是可用可不用的。也就是说，

```
if(condition)    a= 0;

and

if(condition)     {
      a = 0 ;
}
```

它们是都可以通过编译的。 但是良好的代码规范，都推荐使用后者的风格。所以，想对此进行检

查，请在配置文件中，加上：

```
<module name="NeedBraces">
    <property name="tokens" value="LITERAL_IF, LITERAL_ELSE"/>
</module>
```

其中"tokens"属性指定了被检查的区域，它的值可以是："LITERAL_DO, LITERAL_ELSE, LITERAL_IF, LITERAL_FOR, LITERAL_WHILE"， 默认值是全部这些值。个人建议使用默认的配置：

```
<module name="NeedBraces"/>
```

7.9.4. 对右侧括号} 的检查（略）

需要的话请EMAIL我。我把它加上。

7.9.5. 不必要的括号

比如，这样的代码是可以通过编译，但是风格很差的：

```
public void guessTheOutput()
{
    int whichIsWich = 0;
    {
        int whichIsWhich = 2;
    }
    System.out.println("value = " + whichIsWhich);
}
```

因为它增加了一对很不必要的括号。对它进行检查可以这样：

```
<module name="AvoidNestedBlocks">
        <property name= "allowInSwitchCase" value="true"/>
</module>
```

其中"allowInSwitchCase"属性指定了在switch区域中是否允许不必要的括号。默认值是"false"。

或者直接：

```
<module name="AvoidNestedBlocks"/>
```

7.10. 编码的检查

coding

编码的习惯检查

7.10.1. 数组尾巴的逗号

如果一个数组定义的右括号 与最后一个元素不在同一行，就需要有一个逗号。比如：

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

我对这种编程风格貌似没见过。如果希望检查数组的最后一个元素后面是有逗号的，这样：

```
<module name="ArrayTrailingComma"/>
```

据说每个元素后面都带个逗号，有利于重新安排元素的各个顺序。仁者见仁，智者见智了：）

7.10.2. 避免内联(inline)条件判断

也就是三目运算符"？:" 。有的内联条件让代码难以理解，比如：

```
String b = (a==null || a.length<1) ? null : a.substring(1);
```

想避免它出现，可以进行检查：

```
<module name="AvoidInlineConditionals"/>
```

注意

使用了本检查，也就是彻底禁止了"?:"的使用。就连最简单的也不行，比如：

```
return true? true : false;
```

只能使用 if-else 来代替。

7.10.3. override的equals方法

想检查一个类覆写(override)了euqals()方法，这样 ：

```
<module name="CovariantEquals"/>
```

7.10.4. 空语句(statement)

如果两个分号";"之前只有空格，那么后一个分号所定义的语句就是空的。比如下面代码的第二行就是空语句：

```
  int a = 0;  //正常
   ;   // 这里就是一个空的语句
  int b = 0;  //正常
```

想对它进行检查，请在配置文件中增加：

```
<module name="EmptyStatement"/>
```

7.10.5. equals和hashCode方法

可以检查一个类是否覆写(override)了equals和hashCode方法，这样：

```
<module name="EqualsHashCode"/>
```

7.10.6. 应该声明成final的局部变量

如果某个变量一直都没变，那么它就应该是final型的。想检查它，请增加：

```
<module name="FinalLocalVariable"/>
```

7.10.7. 不合适的初始化

通过检查来确保程序员使用了工厂方法(factory method)，而不是构造函数。 比如，规定不能通过new 来构造一个java.lang.Boolean，对应的检查应该是：

```
<module name="IllegalInstantiation">
    <property name="classes" value="java.lang.Boolean"/>
```

```
</module>
```

"classes"属性指定了不能使用构造函数的类的范围。多个类之间用分号表示。每个类名前都要有完整的package名字。

7.10.8. 不合适的token

很多大牛都建议在java中不要使用switch；另外，使用c++ 或 c-- 这样的后缀也会让可读性变差。想进行这样的检查，使用：

```
<module name="IllegalToken"/>
```

就可以检查程序中是否使用了"swtich, ++后缀， --后缀"这三个不合适的符号。如果想指定只检查"switch"，可以指定"tokens"属性：

```
<module name="IllegalToken">
        <property name="tokens" value="LITERAL_SWITCH"/>
</module>
```

想检查更多，请参考com.puppycrawl.tools.checkstyle.api.TokenTypes中定义的常量。

7.10.9. 内部赋值语句

如果有人这样写：String s = Integer.toString(i = 2); 是不是很想扁他？这样检查：

```
<module name="InnerAssignment"/>
```

7.10.10. 魔法数

也叫 MagicNumber，非常让程序不可读。比如：

```
sex = 0
```

表示的什么意思？大多数时候，就连作者本人都要皱眉头想半天，汗……所以，这里的"0" 就是一个魔法数。如果这样写就好的多：

```
public static final int MALE= 0;
sex = MALE;
```

这样我们就知道，原来是把 sex 变量的值设成 “男性”。想检查它，请增加：

```
<module name="MagicNumber"/>
```

如果只想对int型和double型的数做检查，而且忽略数字"0"和"0.05"，可以这样：

```
<module name="MagicNumber">
    <property name="tokens" value="NUM_DOUBLE, NUM_INT"/>
    <property name="ignoreNumbers" value="0, 0.05"/>
</module>
```

该检查有两个属性："tokens"属性可以指定检查的范围，可用的值是"NUMBER_DOUBLE,

NUM_FLOAT, NUM_INT, NUM_LONG"中的子集。默认是全部。 "ignoreNumbers"属性指定了可以忽略的数字，默认是"-1, 0, 1, 2"。

魔法数的检查很有必要，可以让你的程序的可读性大大增加。强烈建议使用。:)

7.10.11. 丢了default分支的switch

switch最好别用。因为用了必须要有default分支才能逻辑正确。否则的话，编译不报错。但是运行起来会报错（把所有的case依次执行一遍）。想检查它，请增加：

```
<module name="MissingSwitchDefault"/>
```

7.10.12. 被更改的循环控制变量

比如，一个for循环的 循环数是只应该在 最后的 i++ 中更改的，如果出现以下代码：

```
        for (int i = 0; i < 1; i++) {
            i++;    // 这里是极可能是程序员大意写出来的。
        }
```

则说明，这个循环节有90%是大意，写错了。因为其中的循环控制变量 i 在一个循环中 ++ 了两次。这样检查：

```
<module name="ModifiedControlVariable"/>
```

7.10.13. 多余的throw

为异常声明中的多余的Excpetion（比如duplicate, unchecked异常，或者它们的子类）做检查。默认检查，增加：

```
<module name="RedundantThrows"/>
```

如果允许 unchecked exception ¸请增加：

```
<module name="RedundantThrows">
    <property name="allowUnchecked" value="true"/>
</module>
```

7.10.14. 未被简化的条件表达式

检查过度复杂的条件表达式，比如： (b == true), b || true, !false, 难读且容易出错。

```
<module name="SimplifyBooleanExpression"/>
```

7.10.15. 未被简化的布尔返回值

检查未被简化的boolean返回语句，比如，

```
if (valid())
    return false;
else
    return true;
```

就可以写成：

```
return !valid();
```

Checkstyle作者自我调侃说：这个想法是从PMD的类似规则中剽窃过来的。哈哈。这样检查：

```
<module name="SimplifyBooleanReturn"/>
```

7.10.16. 字符串(String)的比较

检查字符串的比较时没有使用 == 或!=。

Java新手通常使用这样的代码：

```
if (x == "something")
```

而他实际的意思应该是：

```
if ("something".equals(x))
```

想检查，请增加：

```
<module name="StringLiteralEquality"/>
```

7.10.17. 嵌套的if 层次

通过指定来限制 if-else 的嵌套的层次。所谓的“嵌套的if”，是指一个if被包含在另一个if中。下面是嵌套数是"1"的代码:

```
if(true){
        if(true){
        }
}
```

比如，允许的嵌套深度是3的检查：

```
<module name="NestedIfDepth">
    <property name="max" value="3"/>
</module>
```

其中"max"属性定义了最多的嵌套次数。默认是1.

这个检查请大家宽容一点。见过IBM的朋友写的代码里if嵌套的也比较多。一般的项目就更是……代码的改善是个持续的过程，耐心些。

7.10.18. 嵌套的try 层次

通过指定来限制 try-catch 的嵌套的层次。

对最大的嵌套深度是3的检查：

```
<module name="NestedTryDepth">
    <property name="max" value="3"/>
</module>
```

其中"max"指定了最大的嵌套数，默认是1。

7.10.19. 调用父类的clone

检查并确认一个类的clone()方法调用了父类的clone()。 这样：

```
<module name="SuperClone"/>
```

7.10.20. 父类的finalize

检查并确认一个类的finalize()调用了父类的finalize() 这样：

```
<module name="SuperFinalize"/>
```

7.10.21. 不合理的catch

捕获 java.lang.Exception , java.lang.Error, java.lang.RuntimeError 是不合理的。

新手经常把本应捕获多种异常的格式，简化写成：catch Exception 这一种。（汗，我有时也是 -_-!），这样会导致无法正常捕获NPE，OutOfMemoryErrors这样的异常。 而且也不能针对不同的异常进行不同的处理。默认检查：

```
<module name="IllegalCatch"/>
```

想检查 java.lang.Exception 和 your.package.Exception 是不合理的，可以这样：

```
<module name="IllegalCatch">
        <property name="illegalClassNames" value="java.lang.Exception, your.package.Ex
</module>
```

其中"illegalClassNames"属性指定了不合理的异常名称，多个名称之间用","号。

7.10.22. 不合理的throws

确保某些类型（老外叫type，我们叫class, interface ==）不被throw. 比如说，声明抛出 java.lang.Error, java.lang.RuntimeException 就是不可以的。默认检查：

```
<module name="IllegalThrows"/>
```

想检查 java.lang.Exception 和 your.package.Exception 是不合理的，可以这样：

```
<module name="IllegalThrows">
        <property name="illegalClassNames" value="java.lang.Exception, your.package.Ex
</module>
```

其中"illegalClassNames"属性指定了不合理的异常名称，多个名称之间用","号。

7.10.23. package 声明

确保一个类有package声明。如果一个 class 没有package声明，那么它就无法被引用，作者说很多新手都懒的写这个东东。不过我还没见过。呵呵。

```
<module name="PackageDeclaration"/>
```

7.10.24. JUnitTestCase

确保setUp(), tearDown() 方法被正确的命名，没有参数，void返回类型，而且是public 或者protected的 。确保suite() 被正确的命名，没有参数，返回 junit,framework.Test, 而且是 public static 型。注意：上面这几点经常被搞错的后果是写错的方法编译肯定通过，但是不会被junit执行。

```
<module name="JUnitTestCase"/>
```

7.10.25. return 语句的数量

限制一个方法中return语句的数量。默认是2。忽略特定的方法（默认是equals()）如果return 语句太多，说明某个方法需要实现的功能太多，而且很难阅读。（这个时候就需要重构，建议看看《重构》的Extract Method 和 Simplifying Conditional Expressions. 两章。 ）想确保每个方法最多有3个return 语句，忽略equals()，这样做：

```
 <module name="ReturnCount">
    <property name="max" value="3"/>
</module>
```

想确保每个方法最多有3个return语句，检查所有方法，这样做：

```
<module name="ReturnCount">
    <property name="max" value="3"/>
    <property name="format" value="^$"/>
</module>
```

其中"format"属性是一个 regular expression（正则表达式）。"max"属性指定了可允许的最大数。

7.10.26. 声明的顺序

根据 Sun编码规范， class 或 interface 中的顺序如下：

1. class 声明。首先是 public, 然后是protected , 然后是 package level （不包括access modifier ）最后是private . (多个class放在一个java文件中的情况）
2. 变量声明。 首先是 public, 然后是protected然后是 package level （不包括access modifier ）最后是private . (多个class放在一个java文件中的情况）
3. 构造函数
4. 方法

这样：

```
<module name="DeclarationOrder"/>
```

7.10.27. 参数被赋值

禁止对参数赋值。

某个方法传递进来的参数，是不允许在该方法中改变值的。比如：

```
public someMethod(String para1){…}
```

这个方法中，就不允许出现

```
para1 = “new value”;
```

这样的语句。

官方文档：对方法的参数赋值一般说来，是不好的编程技巧。而强制开发人员声明一个final 参数也不太合适的。所以让Checkstyle做检查正合适。

想进行检查，这样：

```
<module name="ParameterAssignment"/>
```

7.10.28. 详尽的变量初始化

确保某个class 在被使用时都已经被初始化成默认值(对象是null , 数字和字符是0 , boolean 变量是false.)了。 这样检查：

```
<module name="ExplicitInitialization"/>
```

7.10.29. switch语句的default位置排在最后

确保switch语句的default一定在最后出现。虽然java语法允许default可以放在switch中的任何位置，但是把它放在各个case的最下面是可读性最强的。

p.s. 如果不写default , 就会产生逻辑错误，却没有提示。非常不建议使用switch，请使用if…else代替。当然了，如果没使用default,貌似本检查也能检查的出来。 这样使用：

```
<module name="DefaultComesLast"/>
```

7.10.30. 丢失的构造函数

确保某个类定义了一个默认之外的构造函数。这样：

```
<module name="MissingCtor"/>
```

7.10.31. switch中错误分支。

检查每个switch只执行一个分支。也就是说，检查某个case中是否有 break, return ,throw 或continue语句。

```
<module name="FallThrough"/>
```

《重构》中建议，不使用switch。（貌似在 简化条件表达式一章。）

7.10.32. 多个内容相同的字符串变量

确保一个文件中的同样内容的字符串不出现多次。重复的代码难以维护。所以建议使用常量来代替多个相同字符串。 默认只出现一次的检查：

```
<module name="MultipleStringLiterals"/>
```

最多重复两次的检查：

```
<module name="MultipleStringLiterals">
    <property name="allowedDuplicates" value="2"/>
</module>
```

忽略逗号”,” 和空格” “的检查：

```
<module name="MultipleStringLiterals">
    <property name="ignoreStringsRegexp" value='^(("")|(", "))$'/>
</module>
```

上面例子中可以看出，"allowedDuplicates"属性指定了可以重复的最大数，默认1。"ignoreStringsRegexp"属性指定了被忽略的字符的正则表达式格式，默认是忽略空的字符串(^""$） 。

7.10.33. 同一行禁止声明多个变量

确保每行只声明一个变量。Sun编码规范 6.1 节就是这样约定的。这样：

```
<module name="MultipleVariableDeclarations"/>
```

7.10.34. 不使用this

确保代码不使用this.关键字。默认配置： （检查方法和变量）

```
<module name="RequireThis"/>
```

只检查变量，不检查方法：

```
<module name="RequireThis">
    <property name="checkMethods" value="false"/>
</module>
```

"checkFields"属性指定了是否检查field（类的变量），默认是true, "checkMethods"属性指定了是否检查方法，默认是true.

7.10.35. 不必要的圆括号

检查不必要的圆括号”(,)”。比如， if(((((true))))) 这样。是不是很个性？这样：

```
<module name="UnnecessaryParentheses"/>
```

不过很多时候圆括号的正确使用会提高代码的清晰度。比如：

```
return ( SomeClassName.CONSTANT1 == variable )? result1 : result2;
```

是否使用请酌情选择。

7.11. Class的设计

Class Design

7.11.1. 可见的修改方法

检查类成员的可见性。只有static final 成员是public的，其他的类成员都是private的，除非在本检查的protectedAllowed和packagedAllowed属性中进行了设置。默认检查：

```
<module name="VisibilityModifier"/>
```

允许包可见成员的检查：

```
<module name="VisibilityModifier">
    <property name="packageAllowed" value="true"/>
</module>
```

不允许公共成员的检查：

```
<module name="VisibilityModifier">
    <property name="publicMemberPattern" value="^$"/>
</module>
```

7.11.2. Final class

确保只有private 构造函数的class声明成final.

```
<module name="FinalClass"/>
```

7.11.3. Interfacels Type

本检查就是实现了：

只在定义type的时候使用interface

根据Bloch 的说法，一个interface用来描述一个type, 所以，在interface中不定义方法却只定义常量的做法是不合适的。

本检查也可以配置成不允许marker interface，比如 java.io.Serializable这样的不包括方法和常量的类。

```
<module name="InterfaceIsType"/>
```

7.11.4. 隐藏工具类的构造方法

确保只有static方法的工具类没有public的构造方法。

原理：它的构造函数只应该是private 或 protected（如果它有子类的话）的。 常见的错误是，忘记了隐藏默认的构造函数（它是public的）。

如果想把一个构造函数定义成protected，你也许会考虑下面的构造函数的方法：例子．一个工具类。

```
// not final to allow subclassing
public class StringUtils
{
    protected StringUtils() {
        // prevents calls from subclass
        throw new UnsupportedOperationException();
    }

    public static int count(char c, String s) {
        // ...
    }
}
```

这样检查：

```
<module name="HideUtilityClassConstructor"/>
```

7.11.5. 方便继承(extention)而进行的设计

确保某个class是为了方便继承而设计的。本检查可以强制实现这样的编程风格：在父类中使用空的方法，然后在子类中进行实现。

规则是：在子类中overrided的非private ,非static方法，需要符合下列情况之一：

1. abstract
2. final
3. 有空语句。

原理（借用官方文档）：这个API设计风格保护了父类不被子类所破坏。 缺点是子类的灵活性降低，子类无法阻止父类的代码被执行。同时子类却也不会由于忘记调用父类的方法而破坏父类的状态。

这样检查：

```
<module name="DesignForExtension"/>
```

7.11.6. throws的数量

异常是methods interface的一部分。一个方法如果声明抛出太多不同的异常，会导致异常处理(catch)混乱， 本检查强制规定一个方法的throws的次数。

想配置每个方法最多抛出两个异常，使用：

```
<module name="ThrowsCount">
    <property name="max" value="2"/>
</module>
```

其中"max"属性指定了允许的最大数。默认是1。

7.12. 重复的代码

Duplicated code

7.12.1. StrictDuplicateCode 严格的重复代码检查

本检查应该放在<module name = "Checker">结点下。

重复代码的检查可以让你找到经过COPY/PAST出来的代码。重复代码的典型缺点是增加维护代价， 因为需要你把一个BUG修改很多次，测试更多次。 Checkstyle在实现该方面的检查过程中，考虑了以下的目标：

- 快速
- ?多内存占用
- 避免无效的，错误的警告
- 支持多种语言（java,JSP, c++ ...）
- 支持模糊匹配（注释，空格，换行，变量重命名等）

checkstyle提供的是StrictDuplicateCode, 它速度很快，消耗的内存很少，不会出现错误的警告。当检查支持多种语言时，不支持模糊匹配（这也是它被称为strict（严格）检查的原因）。 也就是说，如果一段重复代码，既在JSP中出现，又在JAVA中出现，就无法支持模糊查询。

> 请注意，其实有很多相当出彩的商业性的重复代码检查工具。最具有代表性的就是RedHill咨询公司的Simian 它很好的实现了以上需求的折中。它在各个方面都优于checkstyle包中的其他类，价格很有诱惑力（对非商业用途免费），因而被checkstyle用作一个插件（plugin）
>
> 我们推荐所有checkstyle的用户试用Simian。
>
> --官方文档

表 7.3 “重复代码插件的特性摘要：”是checkstyle中的重复代码检查的插件的特性摘要：

表 7.3. 重复代码插件的特性摘要：

名字	速度	内存占	错的警告	支持的语言	模糊匹配

		用			
StrictDuplicateCode	快	非常低	肯定不会	任何语言	无
Simian	非常快	低	非常小的几率会出现	多种语言，包括java, C/C++/C#	有限度的支持

7.12.1. StrictDuplicateCode 严格的重复代码检查

当代码只是在缩进上有所不同时，就把它们逐行的进行比较，并报告是否重复。忽略import , 其他的内容，包括javadoc, 方法间隔的空白行，等都会被包含进来，进行比较。

使用默认检查：

```
<module name="StrictDuplicateCode"/>
```

相同代码行超过15行，并且文件编码是UTF-8的检查：

```
<module name="StrictDuplicateCode">
    <property name="min" value="15"/>
    <property name="charset" value="UTF-8"/>
</module>
```

说明

本检查会把空白行算在检查范围里。如果代码中存在一段多于"min"属性所定义的空白行时，也会报错。例如，"min"=8　时，　如果代码中存在9个连续的空白行，就会报错。

注意

本检查一定要放在"TreeWalker"节点前面，否则在Checkclipse中无法使用（Eclipse 3.2, Europa都是）。 但是在Ant中却正常。

7.13. 各种量度

Metrics

编码时的各种量度，很理论化。我还没在国内的其他文档中看到此类内容，所以把内容基本照搬来了，大家共同学习吧。:)

7.13.1. 布尔表达式的复杂度

限制一个布尔表达式中的 &&, || 和^的个数。可以让代码看起来更清晰，容易排错和维护。默认检

查：

```
<module name="BooleanExpressionComplexity"/>
```

最多允许7个操作符：

```
<module name="BooleanExpressionComplexity">
    <property name="max" value="7"/>
</module>
```

其中的"max"指定了允许的操作符的最大数，默认是3。

7.13.2. 类数据的抽象耦合

ClassDataAbstractionCoupling

检查一个类中建立的其他类的实例的个数。这种类型的耦合不是由继承或面向对象的聚合而引起的。 一般而言，任何一个抽象数据类型，如果将其他抽象数据类型作为自己的成员，那么它都会产生数据抽象耦合(DAC)； 换句话说，如果一个类使用了其他类的实例作为自己的成员变量，那么就会产生数据抽象耦合。 耦合度越高，这个系统的数据结构就越复杂。

原文：This metric measures the number of instantiations of other classes within the given class. This type of coupling is not caused by inheritance or the object oriented paradigm. Generally speaking, any abstract data type with other abstract data types as members has data abstraction coupling; therefore, if a class has a local variable that is an instantiation (object) of another class, there is data abstraction coupling. The higher the DAC, the more complex the data structure (classes) of the system.

申思维注：也就是说，一个类中如果使用太多其他的类，是肯定会增加系统复杂度的。使用其他的类越少，耦合度就越少。

耦合度最多是5的检查：

```
<module name="ClassDataAbstractionCoupling">
    <property name="max" value="5"/>
</module>
```

其中"max"属性指定了最大数，默认是7。

7.13.3. 类的分散复杂度

ClassFanOutComplexity

一个类依靠的其他类的个数。这个数字的平方也会被显示出来，以表示最少需要的修改个数。

原文：The number of other classes a given class relies on. Also the square of this has been shown to indicate the amount of maintenence required in functional programs (on a file basis) at least.

分散复杂度最多是10：

```
<module name="ClassFanOutComplexity">
    <property name="max" value="10"/>
</module>
```

其中"max"属性指定了最大数，默认是20。

7.13.4. 函数的分支复杂度

CyclomaticComplexity

确保函数的分支复杂度没有超出限制。该复杂度是通过考察大部分函数（构造函数，一般方法，静态初始函数，实例初始化函数） 中的 if, while, do, for, ? : , catch, switch, case 语句 和 &&, || 的操作符的数目来计算得到的。 它表示了通过一个函数的最少分支数，也因此体现了需要进行的测试数目。 一般而言1-4 是优秀，5-7是合格。8-10看情况重构。11个以上一定要马上重构！

分支复杂度最多是7的检查：

```
<module name="CyclomaticComplexity">
    <property name="max" value="7"/>
</module>
```

其中"max"属性指定了最大数，默认是10。

7.13.5. Npath复杂度

NpathComplexity

NPATH 量度 计算了一个函数的可执行的分支个数。它很重视条件语句的嵌套和多个部分组成的布尔表达式 （比如， A&&B, C||D, 等等）。

Nejmeh说，他的小组有一个非正式的NPATH约定：不超过200。NPATH超过这个数字的函数就一定要被分解。

原文：The NPATH metric computes the number of possible execution paths through a function. It takes into account the nesting of conditional statements and multi-part boolean expressions (e.g., A && B, C || D, etc.).

Rationale: Nejmeh says that his group had an informal NPATH limit of 200 on individual routines; functions that exceeded this value were candidates for further decomposition - or at least a closer look.

复杂度不超过40：

```
<module name="NPathComplexity">
    <property name="max" value="40"/>
</module>
```

其中"max"属性指定了最大数，默认是200。

7.14. 杂项

Miscellaneous

很难归类的杂项

7.14.1. 禁止使用的表达式

为代码问题而做的检查。 用户可以通过本检查来控制某些代码的使用。比如，禁止在代码中使用"System.out.println"的检查：

```
<module name="GenericIllegalRegexp">
    <property name="format" value="System\.out\.println"/>
</module>
```

其中"format"属性指定了非法的字符的正则表达式的格式。（"."用"\."表示），完整的属性表如 表 7.4 “GenericIllegalRegexp的属性列表”所示。

表 7.4. GenericIllegalRegexp的属性列表

名称	说明	类型	默认值
format	禁止使用的表达式	正则表达式	^$（空）
ignoreCase	是否忽略大小写	Boolean	false
ignoreComments	是否忽略注释	Boolean	false
message	检查到错误时显示的信息	String	""（空）

7.14.2. 文件结尾的回车

检查文件末尾是否有一个回车。

原理：源文件和文本文件一般应该以一个新行（回车）作为结尾。特别是在使用源代码控制系统比如CVS的时候。如果没有以新行作为文件结尾，CVS甚至会显示警告。（。。。 貌似我没遇到过，也许用的是WINCVS？）

本检查应该放在<module name = "Checker">结点下。

默认检查：

```
<module name="NewlineAtEndOfFile"/>
```

检查使用LINUX风格的换行符：

```
<module name="NewlineAtEndOfFile">
    <property name="lineSeparator" value="lf"/>
</module>
```

只检查java, xml 或python文件：

```
 <module name="NewlineAtEndOfFile">
    <property name="fileExtensions" value="java, xml, py"/>
</module>
```

该详细属性如表 7.5 “NewlineAtEndOfFile的属性列表”所示。

表 7.5. NewlineAtEndOfFile的属性列表

名称	说明	类型	默认值
lineSeparator	行分隔符号	下列值之一："system", "crlf"(windows), "cr"(Mac) 和 "lf"(Unix)	system
fileExtensions	被检查的文件的扩展名	String Set	所有

7.14.3. Todo注释

对于//TODO： 的检查。事实上就是用一个正则表达式来匹配JAVA文件的注释内容，

如，检查程序中没有包含“WARNING”的注释：

```
<module name="TodoComment">
    <property name="format" value="WARNING"/>
</module>
```

其中"format"属性指定了被检查的字符串的内容，默认是"TODO:"，格式是正则表达式。

7.14.4. 翻译属性文件

为了保证翻译的正确性而检查所有的属性文件(propertiy files)的键(key)是否一致。 如果两个属性文件描述的是同一个东东的不同语言版本，那么它们应该包含相同的键。

本检查应该放在<module name = "Checker">结点下。

假如有同一个目录下的两个属性文件：

```
#messages.properties
hello=Hello
cancel=Cancel

#messages_de.properties
hell=Hallo
ok=OK
```

使用本检查就会找出错误：默认文件 messages.properties中，缺少了hell, ok两个键，而在德文

文件 messages_de.properties中，缺少了hello, cancel键。所以下面就是提示：

```
messages_de.properties: Key 'hello' missing.
messages_de.properties: Key 'cancel' missing.
messages.properties: Key 'hell' missing.
messages.properties: Key 'ok' missing.
```

本检查的算法，就是把若干语言包文件中的key合并到一个集合中，去掉重复项，然后再依次对比每个文件缺少的key。做提示。很适合对支持多语言的项目做检查。

这样就可以：

```
<module name="Translation"/>
```

如果你的翻译文件扩展名是".zh-CN" 和".jp"，就这样：

```
<module name="Translation">
    <property name="fileExtensions" value="zh-CN, jp"/>
</module>
```

其中"fileExtensions"指定了被检查文件的扩展名。默认是"properties"。

7.14.5. 没有被注释掉的Main函数

检查没有被注释掉的main函数

原理：Main()函数通常被用于调试，在JUnit出现之后Main()有所减少，但是也有人喜欢用。我们调试完后很容易就忘掉删了它。 它修改了API，增加了class或者jar文件的体积。所以，除了正常的程序入口之外，所有的main方法都应该被删掉或注释掉。

默认检查：

```
<module name="UncommentedMain"/>
```

可以设置"excludedClasses"属性来不检查某个类，类型是正则表达试，默认值是"^$"。如果不想检查名称中以"Main"结尾的类，可以这样：

```
<module name="UncommentedMain">
        <property name="excludedClasses" value="^Main&"/>
</module>
```

7.14.6. 大写的L

检查并确保所有的常量中的L都是大写的。也就是在常量中没有使用”l”而是用的”L”，比如long型常量，都要以”L”结尾。这与Sun编码规范 3.10.1 相一致。小写的字母l 跟数字1 太象了。

```
<module name="UpperEll"/>
```

7.14.7. 声明数组的风格

检查数组类型的定义。有两种风格：java风格的： String[] args 或者 C/C++ 风格的：String args[]

检查是否使用了java风格：

```
<module name="ArrayTypeStyle"/>
```

检查是否使用了C风格：

```
<module name="ArrayTypeStyle">
    <property name="javaStyle" value="false"/>
</module>
```

7.14.8. final型的参数

检查并确保方法，构造函数，和catch区域中的参数是final的。 Interface 和 抽象类不会被检查，因为它们中没有实现代码，所以它们中的也就没有被修改的可能。

原理：在函数中的参数会很让人感到迷惑，也是个很差的习惯。避免该情况的好办法是把参数声明成 final型。

默认：在一般方法和构造函数中检查：

```
<module name="FinalParameters"/>
```

只在构造函数中检查：

```
<module name="FinalParameters">
    <property name="tokens" value="CTOR_DEF"/>
</module>
```

7.14.9. 缩进

检查java代码的缩进

因为"tab"在不同的文本编辑器下的缩进是不同的，所以最好都使用空格。

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"/>
```

详细属性如表 7.6 “Indentation的属性列表”所示。

表 7.6. Indentation的属性列表

名称	说明	类型	默认值
basicOffset	一个缩进所使用的空格	Integer	4
braceAdjustment	代码段结束标志的大括号前使用的空格数	Integer	0
caseIndent	switch中的case缩进所使用的空格	Integer	4

7.14.10. 与代码同行的注释

本检查为了确保每个注释都是单独的一行。对于以 //开头的注释，它所在行的前面只能是空格。但是，如果一段注释不是一行的结尾，比如

```
Thread.sleep( 10 <some comment here> );
```

那么该行就不会被检查。

原理：Steve McConnel在《Code Complete》（代码大全）中认为与代码同行的注释是个不好的习惯。与代码同行的注释也就是说，该行前面是代码，后面是注释。比如：

```
a = b + c;      // Some insightful comment
d = e / f;        // Another comment for this line
```

引用《代码大全》作为佐证：

- "The comments have to be aligned so that they do not interfere with the visual structure of the code. If you don't align them neatly, they'll make your listing look like it's been through a washing machine."

  注释一定要排列整齐，这样你的程序结构就不会受到视觉上的扰乱。如果不把它们排列整齐，那么杂乱的注释会让你的代码看起来象被洗衣机搅拌过一样。

- "Endline comments tend to be hard to format...It takes time to align them. Such time is not spent learning more about the code; it's dedicated solely to the tedious task of pressing the spacebar or tab key."

  每行结尾的注释难于被格式化……排列它们很需要时间。可这些时间却不是被用在学习代码上，而是被浪费在枯燥的按空格和tab键上。

- "Endline comments are also hard to maintain. If the code on any line containing an endline comment grows, it bumps the comment farther out, and all the other endline comments will have to bumped out to match. Styles that are hard to maintain aren't maintained...."

  每行结尾的注释也很难被维护。如果任何包含结尾注释的行发生了增长，那么该行的结尾注释的开端就会相应的右移，那么其他行的结尾注释的开端同样需要右移，来与之对应。难于被维护的风格基本不会被维护的……

- "Endline comments also tend to be cryptic. The right side of the line doesn't offer much room and the desire to keep the comment on one line means the comment must be short. Work then goes into making the line as short as possible instead of as clear as possible. The comment usually ends up as cryptic as possible...."

  每行结尾的注释也很难全部显示出来。每行的右边没多少闲余的空间，而且如果一定要把注释与代码放在同一行，造成了注释肯定长不了。于是程序员的工作就变成了尽力将每行变的越短越好，而不是越干净越好。这个结尾的注释也通常越来越难被看到而失去意义。

- "A systemic problem with endline comments is that it's hard to write a meaningful comment for one line of code. Most endline comments just repeat the line of code, which hurts more than it helps."

  每行结尾的注释的影响到系统的问题是，它很难为一行代码写出有意义的内容。大部分情况下，它只是把代码的字面意思重复一下，而这个代码也是弊大于利。

他的关于 当行改变时代码难于维护的阐述，随着可以自动化重构的时代的到来变的更重要了。

His comments on being hard to maintain when the size of the line changes are even more important in the age of automated refactorings.

默认这样检查：

```
<module name="TrailingComment"/>
```

想严格遵循 每个 //注释都占一行，这样：

```
<module name="TrailingComment">
    <property name="format" value="^\\s*$"/>
</module>
```

"format"属性指定了 与代码同行的注释 前面部分的代码，是一个正则表达式，默认是"^[\\s\\}\\);]*$"。

7.14.11. 必须出现的字符串

确保代码中存在某个字符串的检查。该检查不关心它出现的位置在哪儿，只要它出现了就可以。比如，检查代码中必须出现了版权的字样, “This code is copyrighted”：

```
<module name="RequiredRegexp">
   <property name="format" value="This code is copyrighted"/>
</module>
```

"format"属性指定了 必须出现的字符串的 正则表达式。默认是空。"^$"

术语表

第三方工具

Ant

java的自动化执行工具。一定一定一定要多用啊。 http://ant.apache.org/

Checkstyle

代码检查工具。可以让对代码进行某个标准的检查。 http://checkstyle.sourceforge.net/

Japloy

代码美化工具。可以让杂乱无章的代码变的干净漂亮。 http://jalopy.sourceforge.net/

Sun编码规范

Sun公司提倡的编码规范。
http://java.sun.com/docs/books/jls/second_edition/html/packages.doc.html#40169

OpenDoc

开源文档。随时有好东东等着你。 ：） http://www.redsaga.com

参考书目

[1] *Checkstyle 4.3 Documentation*. http://checkstyle.sourceforge.net. Oliver Burn.