**JDT核心**

JDT核心（Org.Eclipse，JD..Core）是定义核心Java元素和API的插件。在开发Java特定特性时，应始终将此插件列为先决条件。

JDT核心包使您可以访问Java模型对象和无头Java IDE基础结构。JDT核心包包括：

* **org.eclipse.jdt.core版**-定义描述Java模型的类。
* **org.eclipse.jdt.core.编译器**-为编译器基础结构定义API。
* **org.eclipse.jdt.core.dom（日蚀.jdt.core.dom）**-支持抽象语法树（ast），可用于检查编译单元的结构，直至语句级别。
* **org.eclipse.jdt.core.dom.rewrite（重写）**-支持重写抽象语法树（ast），该树可用于将编译单元的结构操作到语句级别。
* **org.eclipse.jdt.core.eval（日蚀.jdt.core.eval）**-支持对剪贴簿或调试器中的代码段进行计算。
* **org.eclipse.jdt.core.formatter格式设置工具**-支持编译单元、类型、语句、表达式等的格式设置。
* **org.eclipse.jdt.core.index索引**-支持生成JDT搜索引擎可以使用的索引。
* Orclipse .jdt.Cord.jDOM支持一个Java文档对象模型（DOM），它可以用于Java编译单元的结构。此包中的所有类型都已弃用。
* **org.eclipse.jdt.core.search搜索**-支持搜索与特定描述相匹配的Java元素的工作区的Java模型。
* **org.eclipse.jdt.core.util文件**-提供用于操作类文件和Java模型元素的实用程序类。

org.eclipse.jdt.core.jdom自3.0版以来已被弃用。编译单元结构的操作应该使用org.eclipse.jdt.core.dom来完成。

**java模型**

Java模型是建模与创建、编辑和构建Java程序相关联的对象的类集合。Java模型类是在Orclipse .jdtCar中定义的。这些类为资源实现Java特定的行为，并进一步将Java资源分解为模型元素。

**爪哇元素**

包Or.Eclipse .jdtCar定义了对组成Java程序的元素进行建模的类。JDT使用内存对象模型来表示Java程序的结构。这个结构是从项目的类路径派生的。模型是分层的。程序的元素可以分解为子元素。

操作Java元素类似于操作资源对象。当您使用Java元素时，实际上是在处理一些基础模型对象的句柄。必须使用exists（）协议确定元素是否实际存在于工作区中。

下表概述了不同类型的Java元素。

|  |  |
| --- | --- |
| **元素** | **描述** |
| IjavaMod | 表示与工作区对应的根Java元素。具有Java性质的所有项目的父级。它还允许您访问没有Java性质的项目。 |
| IjavaPayes | 表示工作区中的Java项目。（Ijavamodel的孩子） |
| I模块描述 | 表示Java模块描述符。（IjavaProject的孩子） |
| iPackageFragmentRoot（iPackageFragmentRoot） | 表示一组包片段，并将片段映射到文件夹、JAR或zip文件的基础资源。（IjavaProject的孩子） |
| iPackageFragment（iPackageFragment） | 表示与整个包或包的一部分相对应的工作区部分。（ipackagefragmentroot的子级） |
| 图标编译单元 | 表示Java源代码（.java）文件。（ipackagefragment的子级） |
| iPackage声明 | 表示编译单元中的包声明。（ICompilationUnit的子级） |
| IImportContainer公司 | 表示编译单元中包导入声明的集合。（ICompilationUnit的子级） |
| 进口货物声明 | 表示单个包导入声明。（IImportContainer的子级） |
| 伊米普 | 表示编译单元内的源类型，或类文件内的二进制类型。 |
| 菲尔德 | 表示类型内的字段。（IType的子级） |
| 方法 | 表示类型内的方法或构造函数。（IType的子级） |
| 初始化器 | 表示类型内的静态或实例初始值设定项。（IType的子级） |
| ICLASSFILE | 表示已编译（二进制）类型。（ipackagefragment的子级） |
| imodularclassfile（I模块化解压文件） | 表示模块描述的类文件（“module info.class”）。（ipackagefragment的子级） |
| 特征参数 | 表示类型参数。（不是任何Java元素的子元素，它是使用Iype .GETType参数（String）或IMEOD.GETType参数（String）获得的。 |
| 非变量 | 表示方法或初始值设定项中的局部变量。（不是Java元素的子元素，它是使用ICoDeSoop.CordSead（int，int）获得的。 |
| 注释 | 表示Java 5注释。（不是任何Java元素的子元素，它是使用InNabTabel.GeaNeNeType（String）或InNabTabel.GeNangNoDeNS（））获得的。 |
| 可注释的 | 表示类型、字段、方法、局部变量或包声明，这些声明可以用一个或多个IAnnotation进行注释。 |

所有Java元素都支持ijavaEngor接口。

一些元素显示在Packages视图中。这些元素实现IOpenable接口，因为在导航之前必须先打开它们。下图显示了这些元素在包视图中的表示方式。

实现IOPENTABLE的Java元素主要是从底层资源文件中发现的信息创建的。相同的元素一般在资源导航器视图中表示。

从3.3开始，iopEnables是ITypes实现ITyperoot的根。这些是ICompilationUnit和IClassFile。

其他元素对应于构成Java编译单元的项。下图显示了一个Java编译单元和一个内容外接程序，它显示编译单元中的源元素。

这些元素实现isourcereference接口，因为它们可以提供相应的源代码。（当这些元素在内容外层中被选择时，它们相应的源代码在Java编辑器中显示）。

**Java元素及其资源**

许多Java元素对应于工作空间中的通用资源。当您想从泛型资源创建Java元素时，类javaCor是最好的起点。下面的代码片段展示了如何从相应的资源中获取Java元素。



private void createjavaelementsFrom（iproject myproject，ifold myfolder，ifile myfile）{

ijavaProject myjavaProject=javacore.create（myProject）；

if（myJavaProject==空）

//项目未为Java配置（没有Java性质）

返回；

//获取包片段或包片段根

ijavaElement myPackagefragment=javacore.create（myfolder）；

//获取.java（编译单元）、类（类文件）或

//.jar（包碎片根）

ijavaelement myjavafile=javacore.create（myfile）；

}

一旦有了Java元素，就可以使用JDT API来遍历和查询模型。您还可以查询Java元素中包含的非Java资源。

private void createjavaelementsFrom（iproject myproject、ifolder myfolder、ifile myfile）抛出javaModelException{

        ...

/获取项目中包含的非Java资源。

对象[]nonJavaChildren=myJavaProject.getNonJavaResources（）；

        ...

}

**Java项目**

当您从一个简单的项目创建Java项目时，JavaCoor将检查该项目是否配置为Java属性。JDT插件使用项目性质将项目指定为具有Java行为。当“新Java项目”向导创建一个项目时，这个性质（org。eclipse。jdt.Corn.javaCorO.NATURILID）被分配给一个项目。如果没有在项目上配置Java属性，则javaCor在被要求创建项目时将返回NULL。

JavaCoor还用于维护Java类路径，包括查找源代码和库的位置，以及生成输出二进制（.class）文件的位置。

Java项目的独特特性是什么？他们在“.caspSPATH”文件中记录他们的类路径，并将Java增量项目生成器添加到项目的构建规范中。否则，它们只是常规项目，并且可以通过插件配置其他性质（和其他增量建造者）。除了使用自己的行为之外，还需要使用NejjavaProjkWiZad页面来配置Java特性，而除了它们自己的自定义属性或行为之外，插件也要使用Java行为来配置项目。

IjavaMod可以被认为是具有Java项目性质的工作空间中的所有项目的父代（因此可以被视为IJavaProject）。

**操作Java代码**

您的插件可以使用JDT API创建类或接口，向现有类型添加方法，或者更改类型的方法。

更改Java对象最简单的方法是使用Java元素API。可以使用更通用的技术来处理Java元素的原始源代码。

**使用Java元素进行代码修改**

**生成编译单元**

以编程方式生成编译单元的最简单方法是使用IPackageFragment.CreateCompilationUnit。指定编译单元的名称和内容。编译单元在包内创建，并返回新的ICompilationUnit。

编译单元可以通过创建扩展名为“.java”的文件资源，一般地在对应于包目录的适当文件夹中创建。使用泛型资源API是Java工具的后门，因此Java模型不会被更新，直到通用资源更改侦听器被通知，JDT侦听器用新编译单元更新Java模型。

**修改编译单元**

Java源代码的最简单修改可以使用Java元素API完成。

例如，可以从编译单元查询类型。一旦拥有了ITYPE，就可以使用诸如createField、createInitializer、createMethod或createType等协议将源代码成员添加到该类型中。这些方法提供了有关成员位置的源代码和信息。

IsReMeRANIOPACTION接口定义了Java元素的公共源操作。这包括重命名、移动、复制或删除类型成员的方法。

**工作拷贝**

可以通过操作编译单元来修改代码（从而修改底层的ifile），也可以修改编译单元的内存中副本（称为工作副本）。

使用getworkingcopy方法从编译单元获取工作副本。（注意，编译单元不需要存在于Java模型中，以便创建工作副本。）创建这样一个工作副本的人负责在不再需要使用DeCdWorkReCopy方法时丢弃它。

工作副本修改内存缓冲区。getWorkingCopy（）方法创建默认缓冲区，但客户端可以使用getWorkingCopy（WorkingCopyOwner，IProgressMonitor）方法提供自己的缓冲区实现。客户端可以直接操作此缓冲区的文本。如果执行此操作，则必须使用reconcile（int、boolean、workingcopyowner、iprogressmonitor）方法不时将工作副本与缓冲区同步。

最后，可以使用commitworkingcopy方法将工作副本保存到磁盘（替换原始编译单元）。

例如，以下代码段使用自定义工作副本所有者在编译单元上创建工作副本。代码段修改缓冲区，协调更改，将更改提交到磁盘，最后丢弃工作副本。

//获取原始编译单元

ICompilationUnit原始单元=…；

//获取工作副本所有者

WorkingCopyOwner所有者=…；

//创建工作副本

icompilationUnit workingcopy=originalUnit.getWorkingcopy（owner，空）；

//修改缓冲区和协调

ibuffer buffer=（iopenable）workingcopy.getbuffer（）；

buffer.append（“class x”）；

workingcopy.reconcile（icompilationUnit.no-ast，false，null，null）；

//提交更改

workingcopy.commitworkingcopy（false，空）；

//销毁工作副本

workingcopy.discardwingcopy（）；

也可以使用ICompilationUnit.ApplyTextEdit方法修改编译单元的缓冲区。

//获取原始编译单元

ICompilationUnit原始单元=…；

//获取工作副本所有者

WorkingCopyOwner所有者=…；

//创建工作副本

icompilationUnit workingcopy=originalUnit.getWorkingcopy（owner，空）；

//获取文本编辑

textedit编辑=…；

//修改缓冲区和协调

workingcopy.applytextedit（编辑，空）；

workingcopy.reconcile（icompilationUnit.no-ast，false，null，null）；

//提交更改

workingcopy.commitworkingcopy（false，空）；

//销毁工作副本

workingcopy.discardwingcopy（）；

工作副本也可以由使用工作副本所有者的多个客户端共享。以后可以使用findworkingcopy方法检索工作副本。因此，共享的工作副本在原始编译单元和工作副本所有者上设置了键。

下面显示客户端1如何创建共享工作副本，客户端2如何检索此工作副本，客户端1如何丢弃工作副本，客户端2如何尝试检索不再存在的共享工作副本通知：

//客户端1和2:获取原始编译单元

ICompilationUnit原始单元=…；

//客户端1和2:获取工作副本所有者

WorkingCopyOwner所有者=…；

//客户端1:创建共享工作副本

ICompilationUnit WorkingCopyForClient1=OriginalUnit.GetWorkingCopy（所有者，空）；

//客户端2:检索共享工作副本

ICompilationUnit WorkingCopyForClient2=originalUnit.findWorkingCopy（所有者）；

//这是相同的工作副本

断言WorkingCopyForClient1==WorkingCopyForClient2；

//客户端1:放弃共享工作副本

WorkingCopyForClient1.DiscardWorkingCopy（）；

//客户端2:尝试检索共享工作副本并发现它为空

workingCopyForClient2=originalUnit.findworkingCopy（所有者）；

断言WorkingCopyForClient2==空；

**使用dom/ast api修改代码**

有三种方法可以创建。第一个是使用。第二个是使用。第三种方法是从头开始使用factory方法on（抽象语法树）。**编译单元分析压缩单元协调（…）ast**

**从现有源代码创建AST**

必须使用创建的实例。源代码通过以下方法之一提供给astparser：**astparserastparser.newparser（int）**

* **设置源（char[]）**：从源代码创建AST
* **设置源（iclassfile）**：从类文件创建AST
* **设置源（ICompilationUnit）**：从编译单元创建AST

然后通过调用创建AST。结果是一个具有每个节点正确源位置的AST。在使用setResolveBindings（布尔值）创建树之前，必须请求解析绑定。解析绑定是一项代价高昂的操作，只有在必要时才能执行。一旦修改了树，所有位置和绑定都将丢失。请注意，在使用setbindingsrecovery（布尔值）进行解析的过程中，也可以进行一些绑定恢复。使用此恢复，某些绑定（通常是丢失的类型）将不再为空，从而提高了使用AST树的任何客户机的恢复能力。**createast（iprogressmonitor）**

**通过协调工作副本创建AST**

如果工作副本不一致（已修改），则可以通过调用该方法创建AST。若要请求创建ast，请使用作为第一个参数的方法调用。只有当问题请求程序处于活动状态或强制进行问题检测时，才会计算其绑定。解析绑定是一项代价高昂的操作，只有在必要时才能执行。一旦修改了树，所有位置和绑定都将丢失。注意，在这个解析过程中，还可以通过使用方法reconcile（int，int，workingcopyowner，iprogressmonitor）和在第二个参数上设置的标志enable\_bindings\_recovery来完成一些绑定恢复。使用此恢复，某些绑定（通常是丢失的类型）将不再为空，从而提高了使用AST树的任何客户机的恢复能力。**协调（int，boolean，workingcopyowner，iprogressmonitor）协调（…）ast.jls3**

**从头开始**

可以使用ast上的工厂方法从头创建compilationUnit。这些方法名以new….下面是一个创建HelloWorld类的示例。

第一个代码段是生成的输出：

包装示例；

导入Java.UTL.\*；

公共类地狱世界{

公共静态void main（string[]args）{

system.out.println（“hello”+“world”）；

}

}

以下代码段是生成输出的相应代码。



ast ast=ast.newast（ast.jls3）；

compilationUnit Unit=ast.NewCompilationUnit（）；

packageDeclaration packageDeclaration=ast.newPackageDeclaration（）；

packagedeclaration.setname（ast.newsimplename（“example”））；

Unit.SetPackage（包装声明）；

importDeclaration importDeclaration=ast.newimportDeclaration（）；

限定名称=

ast.新限定名（

AST.NealSimuleNeMe（“Java”）

ast.newsimplename（“util”））；

importDeclaration.setName（名称）；

importDeclaration.setOnDemand（真）；

Unit.Imports（）.Add（导入声明）；

typedeclaration type=ast.newtypedeclaration（）；

type.setinterface（错误）；

type.modifiers（）.add（ast.newmodifier（modifier.modifier关键字.public\_关键字））；

type.setname（ast.newsimplename（“helloworld”））；

methodDeclaration methodDeclaration=ast.newmethodDeclaration（）；

methodDeclaration.setConstructor（false）；

list modifiers=methodDeclaration.modifiers（）；

modifiers.add（ast.newmodifier（modifier.modifier keyword.public\_keyword））；

modifiers.add（ast.newmodifier（modifier.modifierKeyword.static\_关键字））；

methodDeclaration.setName（ast.newSimpleName（“main”））；

methodDeclaration.setReturnType2（ast.newPrimitiveType（primitiveType.void））；

singlevariableDeclaration variableDeclaration=ast.newsingLevariableDeclaration（）；

variableDeclaration.setType（ast.newArrayType（ast.newSimpleType（ast.newSimpleName（“string”））））；

variabledeclaration.setname（ast.newsimplename（“args”））；

methodDeclaration.parameters（）.add（variableDeclaration）；

org.eclipse.jdt.core.dom.block block=ast.newblock（）；

methodInvocation methodInvocation=ast.newmethodInvocation（）；

名字=

ast.新限定名（

ast.newsimplename（“系统”），

ast.newsimplename（“out”）；

methodInvocation.setExpression（名称）；

methodInvocation.setName（ast.newSimpleName（“println”））；

infixexpression infixexpression=ast.newinfixexpression（）；

infixexpression.setOperator（infixexpression.operator.plus）；

stringLiteral literal=ast.newStringGlitteral（）；

literal.setLiteralValue（“hello”）；

infixexpression.setleftoperand（文字）；

literal=ast.newsTringliteral（）；

literal.setLiteralValue（“world”）；

infixexpression.setRightOperand（文字）；

methodInvocation.arguments（）.add（infixexpression）；

expressionStatement expressionStatement=ast.newExpressionStatement（方法转换位置）；

block.statements（）.add（表达式状态）；

methodDeclaration.setBody（块）；

type.bodyDeclarations（）.add（methodDeclaration）；

unit.types（）.add（类型）；

**检索额外位置**

dom/ast节点只包含一对位置（节点的起始位置和长度）。这并不总是足够的。为了检索中间位置，应该使用iscanner API。例如，我们有一个InstanceOfExpression，我们想知道InstanceOf运算符的位置。我们可以编写以下方法来实现这一点：



private int[]getOperatorPosition（表达式，char[]source）{

if（InstanceOfExpression的表达式实例）{

iscanner scanner=toolfactory.createScanner（假、假、假、假）；

scanner.setsource（源）；

int start=expression.getStartPosition（）；

int end=start+expression.getLength（）；

scanner.resetto（开始，结束）；

int令牌；

尝试{

同时（（token=scanner.getNextToken（））！=iterminalsymbols.tokennameeof）{

开关（令牌）{

案例iterminalsymbols.tokennameinstanceof:

返回new int[]scanner.getcurrentTokenStartPosition（），scanner.getcurrentTokenEndPosition（）

}

}

}捕获（InvalidInputException E）{

}

}

返回空；

}

iscanner用于将输入源划分为令牌。每个令牌都有一个在ITerminalSymbols接口中定义的特定值。迭代和检索正确的令牌相当简单。我们还建议您使用扫描器，如果您想在一个超级职业中找到超级关键字的位置。

**源代码修改**

一些源代码修改不是通过Java元素API提供的。使用编译单元的原始源代码和dom/ast的rewrite api，可以更通用地编辑源代码（例如更改现有元素的源代码）。

要执行dom/ast重写，有两组API：描述性重写和修改重写。

描述性API不修改AST，但使用ASTRewrite API生成修改的描述。AST重写器收集对节点修改的描述，并将这些描述转换为文本编辑，然后可以应用于原始源。



//创建文档

ICompilationUnit CU=…；//内容为“公共类X \n”

字符串source=cu.getsource（）；

文件文件=新文件（来源）；

//从ICompilationUnit创建dom/ast

ast parser parser=astparser.newparser（ast.jls3）；

解析器.setsource（cu）；

compilationUnit astrot=（compilationUnit）parser.createast（空）；

//创建astrewrite

astrewrite rewrite=astrewrite.create（astrot.getast（））；

//变更说明

simplename oldname=（typedeclaration）astrot.types（）.get（0））.getname（）；

simplename newname=astrot.getast（）.newSimpleName（“y”）；

rewrite.replace（oldname，newname，空）；

//文本编辑的计算

textedit edits=rewrite.rewriteEast（document，cu.getJavaProject（）.getOptions（true））；

//计算新的源代码

编辑、应用（文件）；

string newsource=document.get（）；

//编译单元更新

cu.getBuffer（）.setContents（新闻源）；

修改API允许直接修改AST：

* 请求记录修改（compilationUnit.recordModifications（））。
* 在AST节点上执行修改。
* 修改完成后，生成文本编辑，然后将其应用于原始源（compilationUnit.rewrite（…）。

//创建文档

ICompilationUnit CU=…；//内容为“公共类X \n”

字符串source=cu.getsource（）；

文件文件=新文件（来源）；

//从ICompilationUnit创建dom/ast

ast parser parser=astparser.newparser（ast.jls3）；

解析器.setsource（cu）；

compilationUnit astrot=（compilationUnit）parser.createast（空）；

//修改的开始记录

astrot.recordmodifications（）；

//修改ast

typedclaration typedclaration=（typedclaration）astrot.types（）.get（0）；

simplename newname=astrot.getast（）.newSimpleName（“y”）；

typedeclaration.setname（newname）；

//文本编辑的计算

textedit edits=astrot.rewrite（document，cu.getJavaProject（）.getOptions（true））；

//计算新的源代码

编辑、应用（文件）；

string newsource=document.get（）；

//编译单元更新

cu.getBuffer（）.setContents（新闻源）；

**响应Java元素的更改**

如果您的插件需要知道在事实之后对Java元素的更改，您可以用JavaCor注册Java IeltCealEdTeListEnter。



javacore.addelementChangedListener（new myjavaelementChangeReporter（））；

您可以更具体地使用addElementChangedListener（IElementChangedListener，int）指定您感兴趣的事件类型。

例如，如果您只对在协调操作期间侦听事件感兴趣：



javacore.addelementChangedListener（new myjavaelementChangeReporter（），elementChangedEvent.post-concile）；

Javacore支持两种类型的事件：

* post-change：此类事件的侦听器将在相应的post-change资源更改通知期间得到通知。
* 协调后：此类事件的侦听器将在工作副本的协调操作结束时得到通知（请参见ICompilationUnit.Reconcile（int，boolean，WorkingCopyOwner，IProgressMonitor））。

Java元素更改侦听器在概念上类似于资源更改侦听器（在跟踪资源更改中描述）。下面的代码段实现了将元素Deltas打印到系统控制台的Java元素更改报告。

三

公共类MyJavalementChangeReporter实现IElementChangedListener{

public void elementChanged（elementChangedEvent事件）{

ijavaelementDelta delta=event.getDelta（）；

如果（三角洲）！{NULL）{

system.out.println（“接收的增量：”）；

系统输出打印（delta）；

}

}

}

ijavaelementDelta包含已更改的元素和描述发生的更改类型的标志。大多数时候，Delta树都植根于Java模型级别。然后，客户机必须使用getAffectedChildren导航此增量，以了解哪些项目发生了更改。

以下示例方法遍历delta并打印已添加、删除和更改的元素：



空隙横穿和倾斜（IjavaelementDelta Delta）{

开关（delta.getkind（））{

案例Ijavaelementdelta.added：

system.out.println（delta.getelement（）+“已添加”）；

断裂；

案例IjavaelementDelta.删除：

system.out.println（delta.getelement（）+“已删除”）；

断裂；

案例IjavaelementDelta.Changed：

system.out.println（delta.getelement（）+“已更改”）；

如果（（delta.getflags（）&ijavaelementdelta.f\_children）！= 0）{

system.out.println（“更改在其子项中”）；

}

如果（（delta.getflags（）&ijavaelementdelta.f\_内容）！= 0）{

system.out.println（“更改的内容”）；

}

/\*也可以检查其他标志\*/

断裂；

}

ijavaElementDelta[]children=delta.getAffectedChildren（）；

对于（int i=0；i<children.length；i++）{

traverseandprint（children[i]）；

}

}

由于IANN注释不是任何Java元素的子代，所以注释GeldTras是使用GeNoNoDealDeltAs（）获得的。

几种操作可以触发Java元素更改通知。以下是一些例子：

* 创建资源，例如ipackagefragment.createCompilationUnit（增量表示添加了编译单元）
* 修改资源，例如ICompilationUnit.CreateType（增量表示编译单元已更改，并将类型添加为此编译单元的子级）
* 修改项目的类路径，例如ijavaProject.setrawclaspath（增量表示包片段根已添加到类路径、从类路径中移除或在类路径上重新排序）
* 修改classpath变量值，例如javacore.setclasspathvariable（delta还表示包片段根受到影响）
* 将工作副本与其缓冲区协调，例如ICompilationUnit.Reconcile
* 修改以“.java”结尾的IFIL，并在项目的类路径上，例如使用IFIL.SETFILATE（delta表示编译单元被更改，但没有提供细粒度的信息，因为这不是通过Java模型操作完成的）。

类似于iRealEdSeltA，Java元素增量可以使用iWorkStudioNoNeNeld进行批量处理。由几个工作在IWorkStudioNunnIn中的Java模型操作导致的增量被合并并立即报告。

JavaCurror提供了一种批处理Java元素更改的运行方法。

例如，下面的代码片段将触发2个Java元素更改事件：



//获取软件包

ipackagefragment pkg=…；

//创建2个编译单元

icompilationUnit unita=pkg.createCompilationUnit（“a.java”，“public class A”，false，null）；

ICompilationUnit UnitB=pkg.createCompilationUnit（“b.java”，“public class B”，false，null）；

而下面的代码片段将触发1个Java元素更改事件：



//获取软件包

最终的ipackagefragment pkg=…；

//创建2个编译单元

JavaCoR.Run（

新建iWorkSpaceRunnable（）。{

public void run（iprogressmonitor monitor）引发coreException{

icompilationUnit unita=pkg.createCompilationUnit（“a.java”，“public class A”，false，null）；

ICompilationUnit UnitB=pkg.createCompilationUnit（“b.java”，“public class B”，false，null）；

}

}

零）；

**设置Java构建路径**

本节介绍如何设置Java构建路径。构建路径是用于构建Java项目（IjavaProject）的类路径。

类路径只是类路径条目（iclasspathentry）的数组，描述可用的类型。类型可以以源或二进制形式出现，并且路径上条目的顺序定义了在生成期间解析类型的查找顺序。

Java构建路径反映在Java项目元素的结构中。可以查询项目的包片段根（ipackagefragmentroot）。每个类路径条目映射到一个或多个包片段根，每个根进一步包含一组包片段。

构建路径的讨论不涉及Java运行时路径，它可以与构建路径分开定义。（参见运行Java代码讨论运行时类路径）。

**更改生成路径**

可以使用SeTracWrasSPATH在相应项目的Java元素上以编程方式更改项目的生成路径。以下代码设置项目资源的类路径：



IProject项目=…//获取一些项目资源

ijavaProject javaProject=javacore.create（项目）；

IClassPathenTry[]新类路径=…；

javaproject.setrawclaspath（newclasspath，someprogressmonitor）；

（注意：使用术语“原始”类路径是为了强调一个事实，即用于描述入口位置的任何变量都没有被解析。）

Java构建路径被保存到项目的文件结构中名为“.CaspScript”的文件中。此文件的目的是提供一种通过某些源代码库与其他人共享Java构建路径设置的方法。尤其是，不应该手动编辑此文件，因为它可能会损坏。

**类路径条目**

类路径条目可以使用Javacore上定义的工厂方法定义。类路径条目可以引用以下任一项：

* **源文件夹**-包含源编译单元的文件夹，在相应的包目录结构下组织。源文件夹用于更好地组织大型项目中的源文件，并且只能在包含源文件的项目中引用。相应的工厂方法是newsourceEntry。在给定的源文件夹中，每个编译单元都应该根据其package语句嵌套在适当的文件夹结构中。例如，包“p1”中的编译单元“x.java”必须位于源文件夹的子文件夹“p1”中。可以使用多个源文件夹，只要它们不重叠。源文件夹可以分配自己的输出位置，该位置决定生成的类文件应放置在何处。如果未指定任何文件，则类文件将放置在包含项目的输出位置（请参见ijavaProject.setOutputLocation）。

以下是指示项目“myproject”的源文件夹“src”的类路径条目示例：



iclasspathentry srcEntry=javacore.newsourceEntry（新路径（“/myproject/src”））；

* **二进制库**-类文件文件夹（包含在工作区内部或外部）或类文件存档文件（包含在工作区内部或外部）。存档库可以附加源存档，在向类文件元素请求其源（getsource）时提取源存档。库的工厂方法是NewLibraryEntry。

下面是一个classpath条目示例，它指示“myproject”的类文件文件夹“lib”：



iclasspathentry libentry=javacore.newlibraryentry（

新路径（“/myproject/lib”），

空，//无源

空，//无源

false）；//未导出

以下ClassPath条目具有源附件：



iclasspathentry libentry=javacore.newlibraryentry（

新路径（“d:/lib/foo.jar”），//库位置

新路径（“d:/lib/foo-src.zip”），//源存档位置

new path（“src”），//源存档根路径

true）；//导出

源存档根路径描述源存档中根的位置。如果设置为空，将动态推断存档的根目录。

* **先决条件项目**-另一个Java项目。先决条件项目总是将其源文件夹贡献给依赖项目。它还可以选择贡献标记为导出的任何类路径条目（请参见支持额外布尔参数“isexported”的工厂方法）。这意味着除了将其源贡献给其从属项之外，项目还将导出标记为此类的所有类路径条目。这允许先决条件项目更好地隐藏自己的结构更改。例如，给定的项目可以选择从使用源文件夹切换到导出库。这可以在不需要依赖项目更改其类路径的情况下完成。项目前提条件的工厂方法是NewProjectEntry。

以下ClassPath条目表示必备项目“MyFramework”。



iclasspathentry prjentry=javacore.newprojectentry（new path（“/myframework”），true）；//导出

* **使用某个类路径变量间接引用项目或库。**-项目或库的位置可以相对于类路径变量进行动态解析，类路径变量被指定为入口路径的第一段。然后将条目路径的其余部分附加到解析的变量路径。类路径变量的工厂方法是newvariableEntry。类路径变量是工作区的全局变量，可以通过javacore方法getClassSpahvariable和setClassPathvariable进行操作。

可以注册一个自动类路径变量初始值设定项，该初始值设定项在工作区启动时通过扩展点org.eclipse.jdt.core.ClassPathVariable初始值设定项调用。

以下classpath条目表示其位置保存在变量“home”中的库。源附件是使用变量“src\_home”和“src\_root”定义的：



iclasspathentry varentry=javacore.newvariableentry（

new path（“home/foo.jar”），//库位置

new path（“src\_home/foo\_src.zip”），//源存档位置

new path（“src\_root”），//源存档根路径

true）；//导出

javacore.setclasspathvariable（“home”，new path（“d:/myinstall”），null）；//无进度监视器

* **表示类路径容器的条目**-对一个或多个结构化项目集的间接引用。类路径容器用于引用一组描述复杂库结构的类路径条目。与类路径变量一样，类路径容器（IClassPathContainer）也是动态解析的。类路径容器可以由不同的项目使用，从而使其路径项解析为每个项目的不同值。它们还提供有关它们所表示的库的元信息（库的名称、类型、描述）。类路径变量的工厂方法是NewContainerEntry。类路径容器可以通过javacore方法getClassPathContainer和setClassPathContainer进行操作。

可以注册自动类路径容器初始值设定项，当需要绑定容器时，可以通过扩展点org.eclipse.jdt.core.ClassPathContainer初始值设定项惰性地调用该初始值设定项。

以下ClassPath条目表示系统类库容器：



iclasspathentry varentry=javacore.newContainerEntry（

new path（“jdclib/default”），//container'jdclib'+提示'default'

false）；//未导出

javacore.setClassPathContainer（

新路径（“jdklib/default”），

new ijavaproject[]myproject，/'myproject'的值

新建IClassPath容器[]{

新建IClassPathContainer（）。{

公共IClassPathEntry[]getClassPathEntries（）。{

返回新的iclasspathentry[]{

javacore.newlibraryentry（new path（“d:/rt.jar”），空、空、假）；

}；

}

public string getdescription（）返回“基本JDK库容器”；

public int getkind（）返回iclasspathcontainer.k\_system；

public ipath getpath（）返回新路径（“jdklib/basic”）；

}

}

零）；

**排除模式**

类路径源条目可以被分配一个排除模式，这将阻止源文件夹中的某些资源在类路径上可见。使用模式允许筛选出资源树的指定部分。每个排除模式路径都与类路径条目相关，并使用类似于Ant的模式机制。排除模式可用于指定嵌套的源文件夹，只要外部模式排除了内部模式。

有关排除模式的详细信息，请参阅GetExclusionPatterns。

Java项目API iNoCLSASTH在确定特定资源是否在类路径之前检查包含和排除模式。

评论：

* 排除模式比包含模式具有更高的优先级；换句话说，排除模式可以从要包含的模式中删除文件，而不是从另一个模式中删除。
* 从生成路径中排除的嵌套源文件夹可以设置为输出位置。下面是一个类路径条目示例，它表示项目“myproject”的源文件夹“src”，其中排除的嵌套源文件夹用作输出位置：



（）；

（）；

（

，源文件夹位置

新路径，排除的嵌套文件夹

）；输出位置



**包合模式**

还可以为类路径源条目分配包含模式，该模式显式定义在类路径上可见的资源。如果未指定包含模式，则源条目将包含资源树中根于此源条目路径的所有相关文件。指定一个或多个包含模式意味着只包含资源树的指定部分。指定的每个路径都必须是相对路径，并且将被解释为相对于此源条目的路径。文件模式区分大小写。与这些模式中的一个或多个匹配的文件包含在相应的包片段根目录中，除非它被此条目的一个或多个排除模式排除。

有关路径模式的语法和语义的讨论，请参阅getExclusionPatterns。没有任何包含模式在语义上等同于显式包含模式\*\*。

Java项目API iNoCLSASTH在确定特定资源是否在类路径之前检查包含和排除模式。

实例：

* 包含模式src/\*\*本身包含名为src的根文件夹下的所有文件。
* 包含模式src/\*\*和tests/\*\*包含名为src和tests的根文件夹下的所有文件。
* inclusion pattern src/\*\*和exclusion pattern src/\*\*/foo.java包含名为src的根文件夹下的所有文件，名为foo.java的除外。

**类路径分辨率**

由于类路径变量和容器允许您定义动态绑定的类路径条目，因此类路径API区分原始类路径和解析的类路径。原始类路径是Java项目中最初设置的一个，可以通过询问项目来进一步查询。可以使用查询已解析的类路径。此操作触发解析类路径所需的任何变量和容器的初始化。许多Java模型操作隐式地导致Java构建路径被解析。例如，计算项目的包片段根需要解析构建路径。**setrawclaspathgetrawclaspathgetresolvedclaspath**

**编译Java代码**

请参阅使用批处理编译器和使用Ant Javac适配器的Java开发用户指南。

JDT插件包括一个用于从源代码构建Java.class文件的增量和批处理Java编译器。编译器不提供直接API。它是作为Java项目的构建器安装的。编译是使用标准的平台构建机制触发的。

平台构建机制在增量项目构建器中进行了详细描述。

**编译码**

可以使用生成API编程地编译项目中的Java源文件。



我的项目；

i进程监视器myprogressmonitor；

myproject.build（incrementalprojectbuilder.incremental构建，myprogressmonitor）；

对于Java项目，它调用Java增量项目生成器（连同已经添加到项目的构建规范中的任何其他增量项目生成器）。生成的.class文件将写入指定的输出文件夹。其他资源文件也会复制到输出文件夹。

如果是完整的批处理生成，则输出文件夹中的所有.class文件都可能被“清除”，以确保没有找到过时的文件。这是使用JDT核心生成器选项（core\_java\_build\_clean\_output\_文件夹）控制的。此选项的默认设置是清除输出文件夹。除非重置此选项，否则必须确保将没有相应源文件的所有.class文件放在类路径上的单独的class file文件夹中，而不是输出文件夹中。

增量和批处理生成器可以配置其他选项，以控制将哪些资源复制到输出文件夹。以下示例显示如何设置资源筛选器，以便不将以“.ignore”结尾的文件和名为“meta-inf”的文件夹复制到输出文件夹：



hashTable选项=javacore.getOptions（）；

选项.put（java core.core\_java\_build\_resource\_copy\_filter，“\*.ignore，meta-inf/”）

javacore.setoptions（选项）；

如果文件名与所提供的模式之一匹配，则会将其过滤掉。如果整个文件夹的名称与提供的以路径分隔符结尾的文件夹名称之一匹配，则会筛选出这些文件夹。

增量和批处理生成器也可以配置为仅在.classpath文件有错误时生成单个错误。此选项默认设置，可以消除许多错误。有关与构建器相关的选项及其默认值的完整列表，请参见JDT核心构建器选项。

编译器也可以使用javacore选项进行配置。例如，您可以定义在编译期间发现的不同类型的问题应该使用的严重性。有关编译器相关选项及其默认值的完整列表，请参阅JDT核心编译器选项。

当以编程方式为生成器或编译器配置选项时，应指定选项的范围。例如，设置资源筛选器只能应用于特定项目：



hashtable options=myproject.getoptions（false）；//仅获取在此项目中设置的选项

选项.put（java core.core\_java\_build\_resource\_copy\_filter，“\*.ignore，meta-inf/”）

myproject.setoptions（选项）；

**问题确定**

JDT核心定义了一个专门的标记（标记类型“org.eclipse.jdt.core.problem”）来表示编译问题。要以编程方式发现编译器检测到的问题，应使用标准平台标记协议。有关使用标记的概述，请参见资源标记。

下面的代码段查找编译单元中的所有Java问题标记。



公用iMarker[]findjavaProblemMarkers（ICompilationUnit CU）

引发CoreException{

IResource javaSourceFile=cu.getUnderlyingResource（）；

iMarker[]标记=

javasourcefile.findmarkers（ijavamodelmarker.java\_model\_problem\_marker，

是的，iResource.Depth\_Infinite）；

}

Java问题标记由Java项目生成器维护，并在问题解决和Java源被重新编译时自动删除。

问题ID值被设置为IProblem中定义的常量之一。问题的ID是可靠的，但是消息是本地化的，因此可以根据默认的区域设置进行更改。IProblem中定义的常量是自描述性的。

应定义IQuestRestor的实现，以收集Java操作期间发现的问题。如果为工作副本创建提供了IProblemRequestor，则工作副本可以与问题检测协调。要实现这一点，可以使用协调方法。下面是一个例子：



ICompilationUnit Unit=.；//获取一些编译单元

//创建请求程序以累积发现的问题

ipProblemRequestor ProblemRequestor=new ipProblemRequestor（）。{

公共空接受问题（IProblem问题）{

system.out.println（problem.getid（）+“：”+problem.getmessage（））；

}

public void beginreporting（）

public void endreporting（）

public boolean isactive（）return true；//如果激活，将检测问题

}；

//使用工作副本保存有错误的源

ICompilationUnit WorkingCopy=Unit.GetWorkingCopy（New WorkingCopyOwner（），ProblemRequest，空）；

（（iopenable）workingcopy）.getbuffer（）.setcontents（“public class x extends zork”）；

//触发调节

WorkingCopy.Reconcile（不为空，为真，为空，为空）；

您可以在acceptProblem（iproblem）方法中对报告的问题添加操作。在本例中，报告的问题将是zork无法解决或不是有效的超类，其ID为iproblem.superclasssOutfound。

**使用Java搜索引擎**

您的插件可以使用JDT API在Java元素的工作空间中搜索Java项目，例如方法引用、字段声明、接口的实现者等。

Java搜索的入口点是SeaCHEngEngress类。您可以在Java元素内搜索特定模式，并将搜索范围应用于特定元素。可以使用CreatePattern创建搜索模式。模式的作用域使用CreateJavaSearchScope。一旦定义了模式和范围，就使用搜索方法来收集结果。

搜索结果将报告给搜索请求者，您必须对其进行扩展才能访问结果。

**正在准备搜索**

搜索操作将同时使用描述搜索性质的模式和限制调查范围的范围。

**创建Java搜索模式**

搜索模式定义如何找到搜索结果。您可以从Java元素创建一个搜索模式（参见CeaTePATTrnType（IjavaEngult元素，int LimITTO））或从一个字符串（见CeaTePATTALN（String，int，int，int））。最后一个方法支持通配符（即‘\*’），并且可以用来扩展搜索结果。

例如，创建搜索模式以搜索对给定方法的引用，如下所示：



//获取方法

IMethod方法=…；

//创建搜索模式

searchPattern pattern=searchPattern.createPattern（method，ijavaSearchConstants.references）；

或者创建搜索模式来搜索以“obj”开头的所有类型的声明：



//创建搜索模式

searchPattern pattern=searchPattern.createPattern（“obj\*”，ijavaSearchConstants.type，ijavaSearchConstants.declarations，searchPattern.r\_pattern\_match\_searchPattern.r\_区分大小写）；

支持以下搜索模式：

* 包声明
* 类型声明
* 字段声明
* 方法（和构造函数）声明
* 程序包引用
* 类型引用
* 接口实现者
* 字段引用
* 字段写入访问
* 模块声明
* 模块参考
* 现场读取访问
* 方法（和构造函数）引用
* 使用或模式组合上述模式（请参见CreatorPattern）

请注意，这些模式是使用以下可能的规则创建的：

* **R-精确匹配**
* **R-前缀匹配**
* **R-模式匹配**
* **R~ReXExpLeMatter匹配**
* **r\_camelcase\_匹配**
* **r\u camelcase \u相同的\u部分\u计数\u匹配**

也可以与以下标志之一组合：

* **区分大小写**
* **RelaRuluiX匹配**
* **R\_等价物匹配**

例如，



//请求不区分大小写的前缀匹配

searchPattern pattern1=searchPattern.createPattern（“hash”，ijavaSearchConstants.type，ijavaSearchConstants.declarations，searchPattern.r\_prefix\_match）；

//请求匹配camel case

searchPattern pattern2=searchPattern.createPattern（“hm”，ijavaSearchConstants.type，ijavaSearchConstants.declarations，searchPattern.r\_camel\_case\_match）；

//请求具有严格期望的部件匹配数的camel case

searchPattern pattern3=searchPattern.createPattern（“hm”，ijavaSearchConstants.type，ijavaSearchConstants.declarations，searchPattern.r\_camelcase\_same\_part\_count\_match）；

请注意，使用上面创建的模式可以减少可能结果的数量：

* 图案1:、、等。哈希表哈希图散列地图
* 图案2等哈希图散列地图
* 图案3等散列地图

**在Java搜索模式中使用细粒度标志**

一些引用模式可以通过向参数添加一个或多个细粒度标志来细化。限界

例如，只有在强制转换表达式中使用的类型引用才会匹配以下创建的模式：



/获取类型

IType类型=…；

//创建搜索模式

searchPattern pattern=searchPattern.createPattern（type，ijavaSearchConstants.references\_ijavaSearchConstants.cast\_type\_reference）；

请注意，细粒度标志可以组合在一起，但只能用于同一种搜索（例如，如果参数设置为类型，则只有类型引用标志的组合才有意义）。搜索

类型引用的标志：

* **字段\声明\类型\引用**：用作字段声明类型的类型引用。
* **本地变量声明类型引用**：用作局部变量声明类型的类型引用。
* **参数\声明\类型\引用**：用作方法参数类型的类型引用。
* **父类型\类型\引用**：用作超级类型或超级接口的类型引用。
* **throws子句类型引用**：throws子句中使用的类型引用。
* **铸件类型参考**：转换表达式中使用的类型引用。
* **catch\_type\_引用**：catch头中使用的类型引用。
* **类\实例\创建\类型\引用**：类实例创建中使用的类型引用。
* **返回类型参考**：用作方法返回类型的类型引用。
* **导入\声明\类型\引用**：导入声明中使用的类型引用。
* **注释\类型\参考**：用作批注的类型引用。
* **类型\参数\类型\引用**：用作参数化类型或参数化方法中的类型参数的类型引用。
* **类型变量绑定类型引用**：用作类型变量绑定的类型引用。
* **通配符绑定的类型引用**：用作通配符绑定的类型引用。
* **\_type\_引用的实例**：条件中使用的类型引用。实例

字段或方法引用的标志：

* **超参**：超级字段访问或超级方法调用（例如使用限定符）。超级的
* **合格参考**：限定字段访问或限定方法调用。
* **参考文献**：主字段访问或主方法调用（例如使用限定符）。这
* **隐式\此\引用**：字段访问或方法调用，无任何限定。

**创建Java搜索范围**

如果您对给定项目或甚至给定包中的搜索结果感兴趣，或者如果您知道搜索结果只能在给定类型的层次结构中找到，则可以使用CreateJavaSearchScope（iJavaElement[]）或CreateHierarchyScope（ITYPE）创建适当的搜索范围。

例如，在给定包上创建搜索范围的操作如下：



//获取包

ipackagefragment pkg=…；

//创建搜索范围

ijavaSearchScope Scope=SearchEngine.CreateJavaSearchScope（新ijavaElement[]pkg）；

或者在给定类型的层次结构上创建搜索范围是：



/获取类型

IType类型=…；

//创建搜索范围

iJavaSearchScope Scope=SearchEngine.CreateHierarchyScope（类型）；

最后，可以使用CreateWorkspaceScope创建包含整个工作区的搜索范围：



//创建搜索范围

ijavaSearchScope scope=searchEngine.createWorkspaceScope（）；

**搜索**

一旦创建了搜索模式和搜索范围，并且扩展了searchrequestor，就可以按如下方式启动搜索查询：



//获取搜索模式

搜索模式模式=…；

//获取搜索范围

iJavaSearchScope范围=…；

//获取搜索请求程序

搜索请求程序请求程序=…；

/搜索

searchengine searchengine=new searchengine（）；

searchengine.search（pattern，new searchparticipant[]searchengine.getdefaultsearchparticipant（），scope，requestor，空）；

使用beginreporting方法向搜索请求者发送搜索开始的通知。然后，使用AcceptSearchMatch方法报告每个搜索结果。最后，endreporting指示搜索已结束。

**收集搜索结果**

使用AcceptSearchMatch（SearchMatch）方法报告搜索结果。下面的段落突出显示了SearchMatch的一些功能。

**资源与Java元素**

搜索结果可以对应于Java元素（例如类型声明），或者可以包含在Java元素中（例如，对方法内的类型的引用）。搜索引擎总是试图找到与搜索结果对应或包含搜索结果的最内层Java元素。例如，搜索对方法的引用可以在初始值设定项中找到这样的引用。包含此方法引用的初始值设定项是搜索匹配项的元素。

搜索引擎还试图查找包含Java元素的资源。如果Java元素包含在编译单元或类文件中，则该资源是相应的IFILE。如果Java元素包含在.jar文件中，则返回的资源是.jar文件，如果它在工作区中，则为空。

**震源位置**

源位置getoffset和getlength是相对于包含搜索结果的编译单元给出的。如果搜索结果包含在.jar文件中，则源位置相对于附加的源。如果没有源附加到.jar文件，则它们是（-1，-1）。

**准确与不准确的搜索结果**

在大多数情况下，搜索结果是准确的，这意味着搜索引擎能够确定给定的匹配是什么要求。然而，在某些情况下，搜索引擎不能这样做，在这种情况下，匹配是不准确的。匹配可能不准确的一些可能原因是：

* 包含结果的项目上的类路径设置不正确。例如，它引用了一个不可访问的项目，类路径上的JAR需要另一个不在类路径上的JAR，等等。
* 用户代码无法编译。例如，它指的是尚未定义的类。

**Java搜索索引**

JDT搜索为所有源和依赖JAR创建索引。当jar被添加到类路径中时，它被索引。根据罐子的大小，此操作可能需要一些时间。为了提高性能，可以预先构建索引，并在将JAR添加到类路径时为其指定索引文件。这只支持jar。

**生成索引**

可以通过以下方式预先生成索引：

* 一个api javaindexer generateindexforjar。
* 命令行应用程序javaindexer
* 一个Ant任务eclipse.buildjarIndex

**指定索引**

可以将索引指定为要添加的库的classpath属性。索引文件必须采用有效的URL格式。如果找不到索引文件，JDT将生成一个索引文件并使用它进行搜索。

**对Java代码执行代码辅助**

JDT API允许其他插件在一些Java元素上执行代码辅助或代码选择。允许此操作的元素应实现ICodeassist。

有两种操作：

* 代码完成-计算Java令牌的完成。
* 代码选择-回答给定偏移和长度所选文本所指示的Java元素。

在Java模型中，有两个元素来实现这个接口：ICLASSFILE和ICOMIPICATUN单元。代码完成和代码选择只回答类文件的结果，如果它附加了源。

**代码完成**

**执行代码完成**

以编程方式执行代码完成的一种方法是调用ICodeassist.CodeComplete。在编译单元中指定代码完成所需的偏移量。还必须提供CompletionRequestor的实例才能接受可能的完成。

completionRequestor.accept（completionProposal）中的方法接受各种代码完成建议。completionProposal的方法提供了描述所建议元素（其名称、声明类型等）、在编译单元中插入的建议位置及其相关性的信息。

完成请求者可以接受许多不同类型的完成。此类型由CompletionProposal.GetKind提供。

一些可能的完成类型是（完成方案中可以看到可能完成类型的完整列表）：

* 注释属性-注释属性参考
* 匿名类型-匿名\类\声明
* 类型参考-类型参考
* 字段参考-字段参考
* 关键字-关键字
* 标签参考-标签参考
* 局部变量参考-局部变量参考
* 方法参考-方法参考
* 方法声明-方法声明
* 包导入或引用-包引用
* 变量名-变量声明

完成请求程序还必须能够接受编译错误。

**完成相关性**

因为可能有许多不同的完成方式，所以相关性的概念被用来比较建议完成与其他建议的相关性。相关性由正整数表示。该值没有隐含意义，只是相对于其他建议的值使用。代码完成候选项的相关性可能会受到表达式的预期类型的影响，因为它与周围代码中的类型相关，例如变量类型、强制转换类型、返回类型等。在完成中出现预期前缀或后缀也会影响其相关性。

**完成上下文**

CompletionRequestor的实例还可以接受完成上下文。此上下文由CompletionRequestor.AcceptContext（CompletionContext）方法提供，不依赖于特定的完成建议。completionContext方法提供描述一般上下文的信息，如完成的偏移量、完成的令牌、完成的令牌类型（名称或字符串文字）及其位置。

completionContext还可以提供与完成位置相关的元素（ijavaElement）的一些信息。这些元素基于完成的编译单元缓冲区的内容，不是最后一次协调操作的结果。

其中一些方法是：

* **获取EnclosingElement（）**-此方法返回包含完成位置的最内部封闭元素
* **GetVisibleElements（字符串）**-此方法返回从完成位置可见并可分配给给定类型的元素。

**代码完成选项**

JDT核心插件定义了控制代码完成行为的选项。这些选项可以由其他插件更改。

* 激活可见性敏感完成此选项处于活动状态时，代码完成将不会回答当前上下文中不可见的元素。（例如，它不会回答超级类的私有方法。）
* 自动限定隐式成员当此选项处于活动状态时，完成将自动限定隐式字段引用和消息表达式的完成。

其他选项允许您为字段、静态字段、局部变量和方法参数的建议完成名称指定前缀和后缀。

有关代码辅助选项及其默认值的更多信息，请参阅JDT核心代码辅助选项。

**代码选择**

**执行代码选择**

代码选择用于在编译单元中查找由一系列文本（通常是所选文本）表示的Java元素。要以编程方式执行代码选择，必须调用ICodeassist.CodeSelect。必须提供所选内容的起始索引位置及其长度。结果是一个Java元素数组。大多数情况下，数组中只有一个元素，但如果选择不明确，则返回所有可能的元素。

在下面的示例中，将为编译单元调用代码选择。

//获取编译单元

ICompilationUnit单元=…；

//获取偏移量和长度

int偏移量=…；

int长度=…；

//执行选择

ijavaelement[]elements=unit.codeselect（偏移量，长度）；

system.out.println（“所选元素是”+element[0].getElementName（））；

**光标位置的选择**

当选择长度指定为0时，将通过查找包含指定偏移量的完整标记来计算选择。考虑以下示例方法：

公共空方法（对象）{}

如果在的第一个字符后指定偏移量，并指定长度0，则将计算所选内容以包括整个标记。如果指定长度为5，则选择将被视为。测光法测光法奥莫特

**JDT核心选项**

JDT核心选项控制核心特性的行为，例如Java编译器、代码格式化程序、代码辅助和其他核心行为。用于访问选项的API在javacore中定义。可按如下方式分组访问选项：

* **javacore.getDefaultOptions（）。**-回答选项的默认值。
* **javacore.getoptions（）。**-回答选项的当前值。
* **javacore.setoptions（hashtable newoptions）**-用新值替换选项值。

还可以通过字符串名称单独访问选项。

* **javacore.getoption（字符串选项名）**-回答特定选项的值。

还可以找到配置问题严重性的选项。

* **javacore.getOptionForConfigurableSeverity（int problemID）**-回答用于配置由ProblemID标识的问题严重性的选项。

选项存储为所有已知可配置选项及其值的哈希表。已经为每个选项ID及其可能的常量值在javacore上定义了helper常量。

下面的代码片段将所有核心选项的值还原为其默认值，但有一个（编译器禁用），它是专门设置的。



//获取默认选项

hashTable选项=javacore.getDefaultOptions（）；

//更改选项的值

options.put（javacore.compiler\_pb\_deprecation，javacore.error）；

//设置新选项

javacore.setoptions（选项）；

以下代码片段保留当前选项的值，并且只修改一个（编译器\u pb\_deprecation）：



//获取当前选项

hashTable选项=javacore.getOptions（）；

//更改选项的值

options.put（javacore.compiler\_pb\_deprecation，javacore.error）；

//设置新选项

javacore.setoptions（选项）；

**项目特定选项**

可以使用ijavaProject中的协议覆盖每个项目的选项值。

以下代码片段以两种不同的方式检索特定项目的选项（编译器禁用）的值。布尔参数控制在查询中是否只应返回特定于项目的选项，或者项目的选项值是否应与javacore中的值合并。



//获取项目

IjavaProject项目=…；

//查看此项目中是否设置了选项的值

字符串值=project.getoption（javacore.compiler\_pb\_deprecation，false）；

如果（值=空）{

//没有在项目上设置特定选项

     ...

}

//从此项目中获取选项的值。使用来自的值

//如果没有为项目指定任何值，则返回javacore值

字符串值=project.getoption（javacore.compiler\_pb\_deprecation，true）；

**JDT核心选项描述**

下表描述了可用的JDT核心选项。选项ID显示在括号中，默认值显示为粗体斜体。

**选项类别**

* **编译器选项**
* **生成器选项**
* **Javacore选项**
* **格式化程序选项**
* **代码辅助选项**

**编译器选项**

|  |  |
| --- | --- |
| **描述** | **价值观** |
| **基于注释的空分析**（编译器注释\u空\u分析） | |
| 此选项控制编译器是否将使用空注释来改进（潜在）空引用的分析。  启用后，编译器将解释使用编译器\_nonnull\_annotation\_name和编译器\_nullable\_annotation\_name定义的注释类型，以指定给定类型是否包含值null。  这些分析的效果进一步受到以下选项的控制：编译器\u pb\_空\规范\冲突、编译器\u pb\_空\注释\推理\冲突、编译器\u pb\_空\未选中\转换、编译器\u pb\_冗余\u空\注释、编译器\u pb\_按默认\注释缺少\u非空\ u、编译器\u pb\_语法\ u空\分析\用于\u字段、编译器\u pb u非null\_参数\_注释\_已删除，编译器\_继承\_空\_注释。 | **启用** |
| **残疾人** |
| **内联JSR字节码指令**（编译器\_codegen\_inline\_jsr\_字节码） | |
| 当与一个小于或等于“1.4”的Java目标平台一起启用时，编译器将不再生成JSR指令，而是内联的相应子程序代码序列（主要对应于尝试最后块）。因此，生成的代码将变得更大，但在虚拟机上加载更快，因为验证过程要简单得多。该模式为Java规范请求202添加到“1.5”Java目标平台之前的支持。对于大于或等于“1.5”的Java目标平台，JSR字节码指令的内联是强制性的，并且该选项被忽略。 | **启用** |
| **残疾人** |
| **生成方法参数属性**（编译器代码生成方法参数属性） | |
| 生成后，此属性将允许从反射库、注释处理、代码编织以及在调试器中从平台目标级别1.8及更高级别访问有关方法的形参（例如它们的名称）的信息。 | **生成** |
| **dotNOT生成** |
| **设置目标Java平台**（编译器代码生成目标平台） | |
| 出于二进制兼容性的原因，.class文件用为引用规范的每个级别定义的VM版本进行标记。目标Java平台指定执行生成的类文件所需的最小运行时级别。遵从性、源和目标级别必须满足下表中汇总的一组约束。 | **版本1 1** |
| **版本1** |
| **版本1** |
| **版本1** |
| **版本11.5** |
| **版本1 16** |
| **版本1 17** |
| **版本1 18** |
| **版本\_cldc\_1\_1** |
| **保留未使用的局部变量**（编译器代码生成未使用的本地代码） | |
| 除非请求保留未使用的局部变量（即从不读取），否则编译器将优化它们，可能会改变调试。 | **保存** |
| **优化输出** |
| **设置符合性级别**（编译器符合性） | |
| 为编译器选择符合性级别，然后根据引用规范的所述级别进行操作。遵从性、源和目标级别必须满足下表中汇总的一组约束。 | **版本1** |
| **版本1** |
| **版本11.5** |
| **版本1 16** |
| **版本1 17** |
| **版本1 18** |
| **版本9** |
| **JavaDoc注释支持**（编译器注释支持） | |
| 禁用此支持后，编译器将忽略所有JavaDoc问题选项设置，并且不会报告任何JavaDoc问题。它也不会在javadoc注释中找到任何引用，dom ast javadoc节点将只是一个纯文本，而不是具有结构化标记元素。 | **启用** |
| **残疾人** |
| **空批注的继承**（编译器\u继承\u空\u注释） | |
| 启用后，编译器将检查没有任何显式空注释的每个方法（请参见编译器注释\u空\u分析）：如果它重写具有空注释的方法，它将把当前方法视为具有与被重写方法相同的注释。  注释继承将在传递性应用继承之后以及在重写方法的站点应用任何默认的空值之后（请参见编译器的“非空值”（按“默认值”），使用重写方法的有效空值。  如果不同的隐式空批注（来自非空默认和/或重写方法）适用于方法签名中的同一类型，则这将被标记为错误，并且必须使用显式空批注来消除歧义。 | **启用** |
| **残疾人** |
| **生成行号调试属性**（编译器行编号） | |
| 生成后，此属性将在调试器中启用源代码突出显示（类文件则更大）。 | **生成** |
| **dotNOT生成** |
| **生成局部变量调试属性**（编译器本地变量） | |
| 生成后，此属性将使本地变量名能够显示在调试器中，仅在明确指定变量的位置显示（类文件则更大）。 | **生成** |
| **dotNOT生成** |
| **非空类型的批注类型的名称**（编译器\非空\注释\名称） | |
| 此选项定义编译器可以用来执行特殊空分析的完全限定Java类型名称。  如果此选项指定的注释应用于方法签名或变量声明中的类型，则这将被解释为一个规范，即空值不是该位置的合法值。当前支持的位置有：方法参数、方法返回类型和局部变量。  对于用此注释声明的值，编译器将永远不会触发空引用诊断（由编译器\u pb\_potential \u null\_reference和编译器\u pb\_null\_reference控制），因为假定在这些位置的运行时不会出现空值。  编译器还可以进一步检查是否符合空规格，这是由编译器\u pb \u空\u规格冲突、编译器\u pb \u空\u注释\u推断\u冲突和编译器\u pb \u空\u未检查\u转换进一步控制的。  只有启用选项编译器注释\u空\分析时，此选项才有效。 | Java注解类型默认的限定名为：**org.eclipse.jdt.annotation.non空** |
| **非空类型的辅助注释类型的名称**（编译器\u非空\u注释\u辅助\u名称） | |
| 此选项定义了一个逗号分隔的完全限定Java类型名称列表，编译器可以使用它来执行特殊的null分析。  此列表中名称标识的注释类型的解释方式与@link compiler \_nonnull \_annotation name标识的注释相同。其目的是除了当前项目使用的注释外，还支持使用不同的空注释集的库。辅助空注释不应在项目自己的源代码中使用。  JDT将永远不会主动使用此列表中的任何辅助注释名称，即推断出的空注释和提到空注释的内容辅助建议始终使用编译器的主名称\nonnull \u annotation \u name呈现。  只有启用选项编译器注释\u空\分析时，此选项才有效。 | Java注解类型默认的限定名为：**“** |
| **可为空类型的批注类型的名称**（编译器可为空的注释名称） | |
| 此选项定义编译器可以用来执行特殊空分析的完全限定Java类型名称。  如果此选项指定的注释应用于方法签名或变量声明中的类型，则这将被解释为一个规范，即空值是该位置的合法值。当前支持的位置有：方法参数、方法返回类型和局部变量。  如果在不检查空值的情况下取消引用其类型已用此注释注释注释的值，则编译器将触发一个诊断，该诊断由编译器的“pb\_potential”空值引用进一步控制。  编译器还可以进一步检查是否符合空规格，这是由编译器\u pb \u空\u规格冲突、编译器\u pb \u空\u注释\u推断\u冲突和编译器\u pb \u空\u未检查\u转换进一步控制的。  只有启用选项编译器注释\u空\分析时，此选项才有效。 | Java注解类型默认的限定名为：**org.eclipse.jdt.annotation.nullable** |
| **可为空类型的辅助注释类型的名称**（编译器可为空的注释辅助名称） | |
| 此选项定义了一个逗号分隔的完全限定Java类型名称列表，编译器可以使用它来执行特殊的null分析。  此列表中名称标识的注释类型的解释方式与编译器可为空的注释名称标识的注释相同。其目的是除了当前项目使用的注释外，还支持使用不同的空注释集的库。辅助空注释不应在项目自己的源代码中使用。  JDT将永远不会主动使用此列表中的任何辅助注释名称，即推断出的空注释和提到空注释的内容辅助建议始终使用编译器的主名称\u nullable \u annotation \u name呈现。  只有启用选项编译器注释\u空\分析时，此选项才有效。 | Java注解类型默认的限定名为：**“** |
| **注释类型的名称，用于为未标记的类型指定空值默认值。**（编译器\u非空\u按默认\u注释\u名称） | |
| 此选项定义编译器可以用来执行特殊空分析的完全限定Java类型名称。  如果在没有参数的情况下应用注释，则注释元素中方法签名中所有未注释的类型都将被视为使用非空注释指定的类型（请参见编译器的非空注释名称）。  如果以常量false作为参数应用批注，则在外部作用域使用此批注指定的所有相应默认值都将为带批注的元素取消。  只有启用选项编译器注释\u空\分析时，此选项才有效。 | Java ANNOTTAN类型默认的限定名为：**默认为org.eclipse.jdt.annotation.nonNullBy** |
| **用于为未标记类型指定空值默认值的辅助注释类型的名称**（编译器\u非空\u默认\u注释\u辅助\u名称） | |
| 此选项定义了一个逗号分隔的完全限定Java类型名称列表，编译器可以使用它来执行特殊的null分析。  此列表中名称标识的注释类型的解释方式与编译器\_nonnull\_通过\_default\_annotation\_name标识的注释相同。其目的是除了当前项目使用的注释外，还支持使用不同的空注释集的库。辅助空注释不应在项目自己的源代码中使用。  只有启用选项编译器注释\u空\分析时，此选项才有效。 | Java ANNOTTAN类型默认的限定名为：**“** |
| **报告将注释类型用作超级接口的使用情况**（编译器\_pb\_注释\_super\_接口） | |
| 启用后，每当将注释类型用作超级接口时，编译器将发出错误或警告。尽管这是合法的，但这是不可取的。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告“assert”标识符的使用情况**（编译器\u pb \u断言\u标识符） | |
| 启用后，只要将“assert”用作标识符（1.4中的保留关键字），编译器就会发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告装箱/拆箱转换**（编译器自动氧化） | |
| 启用后，每当执行装箱或取消装箱转换时，编译器将发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告字符串串联中char[]表达式的使用情况**（编译器\_pb\_char\_array\_in\_string\_concatenation） | |
| 启用后，每当字符串连接中使用char[]表达式（例如，“hello”+new char[]'w'、'o'、'r'、'l'、'd'），编译器将发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告折旧**（编译程序弃预测） | |
| 启用后，编译器将以错误或警告的形式发出使用已弃用的API的信号。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告终端折旧**（编译程序\_pb\_terminal \_deprecation） | |
| 启用后，编译器将以错误或警告的形式发出使用最终弃用的API的信号。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告弃用代码中的弃用**（编译器\_pb\_deprecation\_in\_deprecated\_code） | |
| 启用后，编译器将在不推荐使用的代码中使用不推荐使用的API作为错误或警告。 | **启用** |
| **残疾人** |
| **覆盖弃用方法时报告弃用**（编译器\_pb\_deprecation\_当\_overriding\_deprecated\_方法时） | |
| 启用后，编译器将向重写已弃用方法的方法的声明发出信号。 | **启用** |
| **残疾人** |
| **报告对具有受限访问权限的类型的不鼓励引用**（不鼓励编译器引用） | |
| 如果启用，编译器在引用根据访问规则规范定义的具有不鼓励访问的类型时将发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告空语句和不必要的分号**（compiler\_pb\_empty\_语句） | |
| 启用后，如果遇到空语句或不必要的分号，编译器将发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告“枚举”标识符的使用情况**（编译器\u pb \u枚举\u标识符） | |
| 启用后，每当“enum”用作标识符（1.5中的保留关键字）时，编译器将发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告交换机故障案例**（编译器\_pb\_fallthrough\_case） | |
| 启用后，当通过前面的非空案例输入案例时，编译器将发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **将可选错误视为致命错误**（编译器\u pb\_致命\u可选\u错误） | |
| 启用后，可选错误（即严重性设置为“错误”的可选问题）将被视为标准编译错误，即致命错误。当检测到源代码中的致命错误时，编译器会将问题方法/类型生成到相应的类文件中，从而在解决所有问题之前，防止有问题的代码运行。禁用时，可选错误仅被视为用于代码生成的警告，但它们仍然带有错误指示器，使其比常规警告更严重。 | **启用** |
| **残疾人** |
| **报告字段声明隐藏另一个变量**（编译器字段隐藏） | |
| 启用后，每当字段声明隐藏某些字段或局部变量（本地、继承或在封闭类型中定义）时，编译器将发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告类型参数的最终绑定**（编译器\u pb \u final \u参数\u bound） | |
| 启用后，每当泛型类型参数与对应于最终类型的绑定相关联时，编译器将发出错误或警告；由于最终类型无法进一步扩展，因此该参数非常无用。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告最终阻止未正常完成**（编译器\u pb\_finally \u block \u not \u completing） | |
| 启用后，当finally块不能正常完成时，编译器将发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告对访问受限类型的禁止引用**（编译器\u pb \u禁止引用） | |
| 如果启用，编译器将在引用根据访问规则规范定义的不可访问类型时发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告缺少哈希代码方法**（编译器\_pb\_缺少\_hashcode\_方法） | |
| 启用后，如果类型重写object.equals（object），但不重写hashcode（），编译器将发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告隐藏的捕获块**（编译器\u pb \u hidden \u catch \u块） | |
| 在try语句的局部，某些catch块可能隐藏其他块，例如  尝试{  抛出新的java. Io.ChanCurrExchange异常（）；  }catch（java.io.charConversionException E）{  }catch（java.io.ioexception e）。  启用此选项时，编译器将对与选中的异常对应的隐藏catch块发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **作为断言语句的结果报告与空相关的问题**（编译器在空分析中包含断言） | |
| 启用后，编译器将为标记为可能或绝对为空或非空的变量引发与空相关的错误或警告，并且该变量的空状态未被断言后面的任何其他语句修改。此选项仅在编译器符合性为1.4或更高时有效。请注意，此选项仅与Assert语句导致的选项compiler\_pb\_redundant\_null\_check、compiler\_pb\_null\_reference和compiler\_pb\_potential\_null\_reference引发的警告上下文相关。 | **启用** |
| **残疾人** |
| **报告接口方法与非继承方法不兼容**（编译器\_-pb\_不兼容\u非继承\u接口\u方法） | |
| 启用后，每当接口定义与非继承对象不兼容的方法时，编译器将发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告不完整的枚举开关**（编译器\u pb\_uncomplete \u enum \u开关） | |
| 启用后，每当枚举开关语句缺少默认情况时，编译器将发出错误或警告。如果没有给出默认情况，则还会对缺少相应事例标签的每个枚举常量发出一个错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告对静态成员的间接引用**（编译器\u pb\_间接\u静态\u访问） | |
| 启用后，每当以间接方式访问静态字段或方法时，编译器将发出错误或警告。对静态成员的引用最好使用其声明类型名进行限定。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告无效的JavaDoc注释**（编译器\_pb\_无效\_javadoc） | |
| 这是JavaDoc问题严重性的通用控件。启用后，编译器将针对javadoc中的问题发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告无效的JavaDoc标记**（编译器\_pb\_无效的\_javadoc\_标记） | |
| 启用后，编译器将在javadoc中发出未绑定或意外引用标记的信号。引用未声明异常的“throws”标记将被视为意外。请注意，可以根据与javadoc关联的构造的可见性来启用此诊断；另请参见编译器的\u pb\_invalid\_javadoc\_tags\_visibility。 | **启用** |
| **残疾人** |
| **使用不推荐使用的引用报告JavaDoc标记**（编译器\u pb\_无效\u javadoc\_标记\u已弃用\u ref） | |
| 指定编译器是否将报告JavaDoc标记中使用的不推荐使用的引用。请注意，可以根据与javadoc关联的构造的可见性来启用此诊断；另请参见编译器的\u pb\_invalid\_javadoc\_tags\_visibility。 | **启用** |
| **残疾人** |
| **报告带有不可见引用的JavaDoc标记**（编译器\u pb\_无效\u javadoc\_标记\u不可见\u ref） | |
| 指定编译器是否报告JavaDoc标记中使用的不可见引用。请注意，可以根据与javadoc关联的构造的可见性来启用此诊断；另请参见编译器的\u pb\_invalid\_javadoc\_tags\_visibility。 | **启用** |
| **残疾人** |
| **无效JavaDoc标记的可见性级别**（编译器\_pb\_无效的\_javadoc\_标记可见性） | |
| 设置JavaDoc标记问题的最低可见性级别。低于此级别的问题将被忽略。 | **公众** |
| **受保护的** |
| **违约** |
| **私人的** |
| **报告隐藏另一个变量的局部变量声明**（编译器\u pb \u本地\u变量\u隐藏） | |
| 启用后，每当局部变量声明隐藏某些字段或局部变量（本地、继承或在封闭类型中定义）时，编译器将发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **每个编译单元报告的最大问题数**（每个单元的编译器最大值） | |
| 指定每个编译单元上报告的最大问题数（如果最大值为零，则报告所有问题）。 | 正整数或空整数。默认值为100。 |
| **报告限定为静态但未声明为静态的方法**（compiler\_pb\_missing\_static\_on\_方法） | |
| 启用后，如果某个方法未声明为静态方法，则编译器将发出错误或警告，即使该方法限定为静态方法。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告可以限定为静态但未声明为静态的方法**（编译器\u pb\_可能在\u方法上缺少\u static\_） | |
| 启用后，如果某个方法未声明为静态方法，则编译器将发出错误或警告，即使当另一个方法不重写该方法时，它可能被限定为静态方法。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **具有构造函数名称的报告方法**（编译器\u pb\u方法\具有\u构造函数\名称） | |
| 用构造函数名命名方法通常被认为是糟糕的编程风格。当启用此选项时，编译器将发出错误或警告的信号。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告缺少@deprecated注释**（编译器\_pb\_缺少\_已弃用的\_注释） | |
| 启用后，每当遇到带有@deprecated doc标记但没有对应的@deprecated注释的声明时，编译器将发出错误或警告。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告缺少JavaDoc注释**（编译器\_pb\_缺少\_javadoc\_注释） | |
| 这是针对缺少JavaDoc注释问题严重性的通用控件。启用后，如果缺少javadoc注释，编译器将发出错误或警告。请注意，可以根据与预期JavaDoc关联的构造的可见性来启用此诊断。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告重写方法缺少JavaDoc注释**（编译器\_pb\_缺少\_javadoc\_注释\_覆盖） | |
| 指定编译器是否将验证重写方法以报告缺少的JavaDoc注释问题。 | **启用** |
| **残疾人** |
| **缺少JavaDoc注释的可见性级别**（编译器\_pb\_缺少\_javadoc\_注释\_可见性） | |
| 为缺少的JavaDoc问题设置最低可见性级别。低于此级别的问题将被忽略。 | **公众** |
| **受保护的** |
| **违约** |
| **私人的** |
| **报告缺少标记说明**（编译器\_pb\_缺少\_javadoc\_标记描述） | |
| 启用后，编译器将报告任何缺少所需描述的javadoc标记的警告或错误。问题的严重性由选项编译器\_pb\_invalid\_javadoc控制。它不依赖于选项编译器的\_pb\_invalid\_javadoc\_标记。当此选项的值为compiler\_pb\_missing\_ujavadoc\_tag\_description\_u all\_standard\_标记时，将选中具有描述、文本或标签的标准javadoc标记的子集。虽然这一组可能会在将来增长，但请注意，用户定义的标记不会也不会被选中。 | **编译器\_pb\_缺少\_ujavadoc\_标记\_描述\_返回\_标记** |
| **编译器\_pb\_缺少\_ujavadoc\_标记\_描述\_所有\_标准\_标记** |
| **编译器\_pb\_缺少\_ujavadoc\_标记描述\_无\_标记** |
| **报告缺少JavaDoc标记**（编译器\_pb\_缺少\_javadoc\_标记） | |
| 这是JavaDoc缺少标记问题严重性的通用控件。启用后，当javadoc注释中缺少标记时，编译器将发出错误或警告。请注意，可以根据与JavaDoc关联的构造的可见性来启用此诊断。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告重写方法缺少JavaDoc标记**（编译器\_pb\_缺少\_javadoc\_标记\_覆盖） | |  |
| 指定编译器是否将验证重写方法以报告JavaDoc缺少标记问题。 | **启用** |  |
| **残疾人** |  |
| **报告方法类型参数缺少JavaDoc标记**（编译器\_pb\_缺少\_javadoc\_标记\_method\_type\_参数） | |  |
| 指定是否应报告方法声明中类型参数缺少的@param。启用后，编译器将为没有对应@param标记的类型参数发出缺少javadoc标记错误或警告。此选项仅在编译器符合性为1.5或更高时有效。 | **启用** |  |
| **残疾人** |  |
| **缺少JavaDoc标记的可见性级别**（编译器\_pb\_缺少\_javadoc\_标记\_可见性） | |  |
| 设置JavaDoc缺少标记问题的最低可见性级别。低于此级别的问题将被忽略。 | **公众** |  |
| **受保护的** |  |
| **违约** |  |
| **私人的** |  |
| **报告缺少@override注释**（编译器\_pb\_缺少\_override\_注释） | |  |
| 启用后，每当遇到重写超类方法但没有@override注释的方法声明时，编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告缺少接口方法实现的@override注释**（编译器\_pb\_缺少\_override\_annotation\_for\_interface\_method\_实现） | |  |
| 启用后，每当遇到重写或实现SuperInterface方法但没有@override注释的方法声明时，编译器将发出错误或警告。此选项仅在编译器符合性为1.6或更高时有效。问题的严重性由选项编译器“缺少”覆盖“注释”来控制。 | **启用** |  |
| **残疾人** |  |
| **报告可序列化类上缺少serialversionID字段的声明**（编译器\_pb\_缺少\_serial\_版本） | |  |
| 启用后，每当可序列化类缺少serialversionID字段的本地声明时，编译器将发出错误或警告。此字段必须声明为静态final，并且类型为long。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **无效果的报告分配**（编译器“不影响”赋值） | |  |
| 如果启用，编译器将在分配无效（例如“x=x”）时发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告非外部化字符串文本**（编译器\_pb\_non\_nls\_string\_literal） | |  |
| 启用后，编译器将对非外部化字符串文本（即未用//$non-nls-<n>$标记）发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告删除的非空参数批注**（编译器\_pb\_nonnull\_参数\_注释\_已删除） | |  |
| 启用后，如果以下所有条件都保持不变，编译器将针对重写继承方法的方法的参数发出错误或警告：   * 重写的方法将相应的参数声明为非空（请参见编译器的\非空\注释\名称）。 * 重写方法中的参数没有空批注。 * 重写方法不受nullness默认值的影响（请参见编译器的\u default \u annotation \u name）。 * 已禁用空批注的继承（请参见编译器继承空批注）。   这种特殊情况与任何未标记的方法参数具有相同的固有风险，因为编译器的空分析无法确定wither null是否是该参数的合法值。但是，重写方法中的注释建议还应将参数注释为非空。如果这不是有意的或不可能的，建议将参数注释为可以为空，以便明确合同的这一（合法）更改。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告类型变量的不安全非空解释**（编译器\_pb\_nonnull\_typevar\_来自\_legacy\_调用） | |  |
| 启用后，如果以下所有条件都保持不变，编译器将对方法调用发出错误或警告：   * 该方法声明的返回类型是不带任何空批注的类型变量。 * 对于给定的调用，此类型变量由非空类型替换。 * 声明方法的类型由第三方库提供。 * 此库类型不存在空批注，既不在其类文件中，也不使用外部批注。   这种特殊情况利用选项一致地用非空类型替换所有出现的类型变量，但它承担了库类型可能不知道空注释的风险，因此对于类型变量的特定出现缺少必要的@nullable注释。  只有启用了选项编译器\u注释\u空\分析，并且配置的一组空注释声明目标类型\使用时，此选项才有效。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告空取消引用**（编译器\u pb \u空\u引用） | |  |
| 启用后，每当使用静态已知包含空值的变量访问字段或方法时，编译器将发出错误或警告。由于断言而产生的这个选项引发的错误或警告可以由CopySerpPbInEclipse AsvsTysIn null分析（Java 1.4和更大）控制。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告从具有未知空值的类型到带空注释的类型的未选中转换**（编译器\_pb\_null\_unchecked\_conversion） | |  |
| 启用后，每当检测到以下情况之一时，编译器将发出错误或警告：   1. 使用非空批注声明的方法返回一个表达式，该表达式没有足够的空度信息可用于静态证明任何流在运行时都不会传递空值。 2. 对于没有足够的空信息可用于静态证明其在运行时永远不会计算为空值的表达式，将作为方法调用中的参数传递，在该方法调用中，被调用方法的相应参数用非空批注声明。 3. 如果表达式的空值信息不足，无法静态地证明其在运行时永远不会计算为空值，则将该表达式分配给使用非空批注声明的局部变量。   未选中的空转换通常是使用其他未标记变量或方法的结果。  编译器选项编译器\u nonnull\_annotation\_name和编译器\u nullable\_annotation\_name控制编译器应分别解释为非空或可空注释的注释。  只有启用选项编译器注释\u空\分析时，此选项才有效。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告通过对自由类型变量进行悲观的空分析检测到的问题**（编译器\_pb\_悲观\_空\_分析\_自由\_类型\_变量） | |  |
| 除非设置为“忽略”，否则不受任何显式空注释影响的类型变量将从两个方向进行悲观分析：当读取此类型的值时，假定它可以为空。当此类型显示为所需类型（即，在赋值或变量初始化的左侧，或作为正在检查返回语句的方法返回类型）时，该类型被认为需要非空属性。  由于这种悲观分析而报告的问题将按此选项中给出的级别报告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告违反空规格的情况**（编译器\u pb \u空\u规范\u冲突） | |  |
| 启用后，每当检测到以下情况之一时，编译器将发出错误或警告：   1. 使用非空批注声明的方法返回可为空的表达式。 2. 可以为空的表达式在方法调用中作为参数传递，在方法调用中，被调用方法的相应参数用非空批注声明。 3. 可以为空的表达式被分配给用非空批注声明的局部变量。 4. 重写用非空批注声明的继承方法的方法尝试通过指定一个可为空的批注来放宽该约定（禁止反变量返回）。 5. 重写继承方法（该继承方法至少有一个参数具有可为空的声明）的方法，尝试通过为其相应参数指定一个非空注释（禁止协变参数）来收紧该空协定。   在上面的例子中，如果一个表达式静态地计算为空值，或者如果它是用一个可以为空的注释声明的，那么它就被认为是可以为空的。  编译器选项编译器\u nonnull\_annotation\_name和编译器\u nullable\_annotation\_name控制编译器应分别解释为非空或可空注释的注释。  只有启用选项编译器注释\u空\分析时，此选项才有效。 | **误差** |  |
| **警告** |  |
| **报告不调用其超级方法调用的重写方法。**（编译器\_pb\_覆盖\_方法\_而不调用\_super\_） | |  |
| 启用后，如果某个方法在不调用超级调用的情况下重写另一个方法，编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告可能的空解引用**（编译器\_pb\_potential\_null\_reference） | |  |
| 启用后，每当使用以前针对空进行过测试但静态不知道（不再知道）持有非空值的变量访问字段或方法时，编译器将发出错误或警告。由于断言而产生的这个选项引发的错误或警告可以由CopySerpPbInEclipse AsvsTysIn null分析（Java 1.4和更大）控制。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告声明的空批注和推断的空值之间的冲突**（编译器\u pb \u空\u注释\u推理\u冲突） | |  |
| 启用后，每当检测到以下情况之一时，编译器将发出错误或警告：   1. 使用非空批注声明的方法返回一个静态已知的表达式，以在某些流上计算为空值。 2. 静态已知的表达式在某些流上计算为空值，在方法调用中作为参数传递，其中被调用方法的相应参数用非空批注声明。 3. 在某些流上静态计算为空值的表达式被分配给用非空注释声明的局部变量。   编译器选项编译器\u nonnull\_annotation\_name和编译器\u nullable\_annotation\_name控制编译器应分别解释为非空或可空注释的注释。  只有启用选项编译器注释\u空\分析时，此选项才有效。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告冗余空检查**（编译器\u pb\_冗余\u空\u检查） | |  |
| 启用后，每当静态已知的保存空值或非空值的变量针对空值进行测试时，编译器将发出错误或警告。由于断言而产生的这个选项引发的错误或警告可以由CopySerpPbInEclipse AsvsTysIn null分析（Java 1.4和更大）控制。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告多余的空注释。**（编译器\_pb\_冗余\_空\_注释） | |  |
| 启用后，当应用非空注释（请参见编译器非空注释名称）时，编译器将发出错误或警告，尽管在当前位置应用的默认值已达到相同的效果。通过使用选项编译器“非空”指定的注释和“默认注释”名称，这样的默认值可能有效。  编译器选项编译器\u nonnull\_annotation\_name和编译器\u nullable\_annotation\_name控制编译器应分别解释为非空或可空注释的注释。  只有启用选项编译器注释\u空\分析时，此选项才有效。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告缺少默认的空注释**（编译器\u pb\_缺少\u非空\u由\u默认\u注释） | |  |
| 启用后，编译器将在以下情况下发出错误或警告：   * 当包不包含默认的空值批注时，由于package-info.java中缺少package-info.java或package-info.java中缺少默认的空值批注。 * 当默认包中的类型不包含默认的空注释时。   只有启用选项编译器注释\u空\分析时，此选项才有效。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告用于创建泛型类实例的类型参数的冗余规范**（编译器\u pb\_冗余\u类型\u参数） | |  |
| 启用后，编译器将在泛型实例创建表达式中指定类型参数时发出错误或警告，尽管可以使用菱形运算符“<>”。（Java 1.7和更大） | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告试图重写包可见方法**（编译器重写包默认方法） | |  |
| 包可见方法（未显式声明为public、protected或private的任何方法）在其他包中不可见，因此不能从其他包中重写。试图重写另一个包的包可见方法时，引入了一个与原始方法无关的新方法。启用此选项时，编译器将发出错误或警告等情况的信号。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报表参数分配**（编译器\u pb \u参数\u分配） | |  |
| 启用后，如果参数分配给，编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告可能的意外布尔赋值**（编译器“可能的”意外“布尔”赋值） | |  |
| 启用后，如果布尔赋值充当控制语句的条件（可能是布尔比较），编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告原始类型引用**（编译器\_pb\_raw\_type\_reference） | |  |
| 启用后，编译器将在检测到对原始类型的引用时发出错误或警告。不鼓励使用原始类型，它们旨在帮助与遗留代码进行交互。将来，语言规范可能会拒绝对泛型类型的原始引用。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告冗余的上级**（编译器的冗余超级接口） | |  |
| 启用后，如果类型显式实现已由其任何父类型实现的接口，编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告隐藏其他字段的特殊参数**（编译器\u pb\_特殊\u参数\u隐藏\u字段） | |  |
| 启用后，编译器将指示构造函数或setter方法参数声明隐藏某些字段（本地、继承或在封闭类型中定义）的情况。 | **启用** |  |
| **残疾人** |  |
| **报告对静态成员的非静态引用**（编译器静态访问接收器） | |  |
| 启用后，每当使用表达式接收器访问静态字段或方法时，编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **进一步确定@suppress warnings的效果，如果也启用了编译器\_pb\_suppress\_warnings。**（编译器抑制可选错误） | |  |
| 启用后，@suppresswarnings注释还可用于抑制已配置为错误的可选编译器诊断。禁用时，所有@suppresswarnings注释只影响警告。 | **启用** |  |
| **残疾人** |  |
| **正在确定@suppresswarnings的效果**（编译器禁止显示警告） | |  |
| 启用后，@suppress warnings注释可用于禁止某些编译警告。禁用时，将忽略所有@supressWarnings注释；即，即使警告出现在带有@suppressWarnings注释的实体范围内，也会报告警告。 | **启用** |  |
| **残疾人** |  |
| **对字段执行语法空分析**（编译器\u pb\_语法\u空值\u分析\u字段） | |  |
| 启用后，编译器将检测某些语法星座，在这些星座中，通常会对字段引用发出与空相关的警告，但由于同一字段引用之前已知为非空，因此可以在低风险下抑制该警告。 | **启用** |  |
| **残疾人** |  |
| **报告合成访问仿真**（编译器\_-pb\_合成\_-access\_仿真） | |  |
| 启用后，编译器将在模拟对封闭类型的不可访问成员的访问时发出错误或警告。这种访问可能会影响性能。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告类型参数声明隐藏其他类型**（编译器\_pb\_type\_参数\_隐藏） | |  |
| 启用后，每当类型参数声明隐藏某个类型时，编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告由于原始API而不可避免的泛型类型问题**（编译器问题不可避免） | |  |
| 启用后，编译器将发出错误或警告，即使它检测到程序员无法避免的与泛型相关的类型问题，因为引用的API已经包含原始类型。例如，可能会强制类型在其方法签名和返回类型中使用原始类型，因为它从父类型重写的方法首先声明为使用原始类型。 | **启用** |  |
| **残疾人** |  |
| **报告未选中的类型操作**（编译器\_pb\_unchecked\_type\_operation） | |  |
| 如果启用，编译器将在操作涉及泛型类型时发出错误或警告，并且可能会使类型安全性失效，因为涉及原始类型（例如，使用参数（x）调用foo（x<string>）。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告未关闭关闭**（编译器未关闭可关闭） | |  |
| 启用后，如果局部变量包含java.lang.autocloseable（compliance>=1.7）类型的值或java.io.closeable（compliance<=1.6）类型的值，并且如果流分析显示该值没有在本地调用方法close（），则编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告潜在未关闭的可关闭**（编译器可能未关闭） | |  |
| 启用后，如果局部变量包含java.lang.autocloseable（compliance>=1.7）类型的值或java.io.closeable（compliance<=1.6）类型的值，并且如果流分析显示没有对所有执行路径的该值本地调用方法close（），则编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告显式关闭可关闭**（编译器\_pb\_显式关闭\_autocloseable） | |  |
| 启用后，如果局部变量包含java.lang.autoclosable类型的值，并且在该资源上显式调用了方法close（），但资源不是由try with resources块管理，则编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告提供不可能类型的参数的方法调用**（编译器“不太可能”集合“方法”参数“类型） | |  |
| 启用后，当某些采用“object”的已知集合方法（如map.get（object））与似乎与集合的相应类型参数无关的参数类型一起使用时，编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **对预期的收集方法类型执行严格的分析**（编译器“不太可能”集合“方法”参数“类型”严格） | |  |
| 这是编译器“不太可能”集合“方法”参数“类型”的子选项，它将用严格的兼容性检查替换启发式方法，即每个与预期类型不严格兼容的参数都将触发错误或警告。 | **启用** |  |
| **残疾人** |  |
| **向方法“equals”报告提供不太可能类型的参数的方法调用**（编译器“不太可能”等于“参数”类型） | |  |
| 当启用时，编译器将发出一个错误或警告，当java. Lang.Objist.Error（object）与一个似乎与接收器类型无关的参数类型使用时，或者相应地，当java. UTI.Objist.Abjes（object，object）的参数具有彼此不相关的类型时。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **公共API使用非API类型时报告**（编译器\_pb\_api\_泄漏） | |  |
| 启用后，当公共API提到客户端无法访问的类型时，编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告未记录的空块**（编译器\_pb\_undocumented\_empty\_块） | |  |
| 启用后，当检测到空块并且没有用任何注释记录时，编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **为@suppresswarnings报告未处理的警告标记**（编译器\u pb\_未处理的\u警告\u令牌） | |  |
| 启用后，编译器将在遇到@suppresswarnings注释中无法处理的标记时发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告不必要的其他**（编译器\_pb\_unnecessary\_else） | |  |
| 启用后，当语句不必要地嵌套在else子句中时（在then子句未正常完成的情况下），编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告不必要的类型检查**（编译器\u pb\_不必要的\u类型\u检查） | |  |
| 启用后，如果不需要强制转换或实例操作，编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告不合格的现场访问**（编译器\_pb\_unqualified\_field\_access） | |  |
| 启用后，当一个字段在没有任何限制的情况下被访问时，编译器将发出一个错误或警告。为了提高代码可读性，它应该是限定的，例如“x”应该写为“this.x”。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告未使用的已声明引发的异常**（编译器“未使用”已声明“引发”异常） | |  |
| 启用后，当方法或构造函数将选中的异常声明为引发时，编译器将发出错误或警告，但其主体实际上既不引发该异常，也不引发任何其他扩展该异常的异常。此诊断由选项compiler\_pb\_uuu unused\_declared\_through\_exception\_exception\_and\_throwable、compiler\_pb\_u unused\_declared\_through\_exception\_include\_doc\_comment\_reference和compiler\_pb\_u unused\_declared\_through\_exception\_u overriding进一步优化。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告未使用的已声明引发的异常将免除异常并可丢弃**（编译器\_pb\_unused\_声明了\_through\_exception\_exception\_exception\_和\_throwable） | |  |
| 启用后，当方法或构造函数声明抛出java.lang.throwable或java.lang.exception之外的已检查异常时，编译器将发出错误或警告，但其主体实际上既没有引发该异常，也没有任何其他扩展该异常的异常。当禁用时，当一个方法或构造函数声明一个被检查的异常（包括java. Lang.Lable和Java.Lang.ExtExchange）被抛出时，编译器将发出错误或警告，但它的主体实际上既不引发该异常，也不引发任何其他异常。未使用声明的引发异常问题的严重性由选项编译器“未使用声明的引发异常”控制。此诊断由选项compiler\_pb\_uuu unused\_declared\_through\_exception\_包括\_doc\_comment\_reference和compiler\_pb\_u unused\_declared\_through\_exception\_u overriding进一步优化。 | **启用** |  |
| **残疾人** |  |
| **考虑在文档注释中引用未使用的已声明的引发异常检查**（编译器\_pb\_unused\_声明了\_through\_exception\_include\_doc\_comment\_reference） | |  |
| 启用后，编译器将考虑对未使用的已声明引发异常检查的异常（即@throws子句）的文档注释引用。因此，记录在案的例外情况将被视为根据DOC合同强制执行。未使用声明的引发异常问题的严重性由选项编译器“未使用声明的引发异常”控制。注意：在根据选项编译器注释支持启用文档注释支持之前，此选项无效。此诊断将通过选项编译器“未使用的”声明的“抛出的”异常的“豁免的”异常的“和”可丢弃的“以及编译器“未使用的”声明的“抛出的”异常的“覆盖时”进一步优化。 | **启用** |  |
| **残疾人** |  |
| **报告重写方法中未使用的已声明引发的异常**（编译器\_pb\_unused\_声明\_through\_exception\_when\_overriding） | |  |
| 禁用时，编译器将报告未使用的声明的抛出异常，无论是在重写方法上还是在实现方法上。未使用声明的引发异常问题的严重性由选项编译器“未使用声明的引发异常”控制。此诊断通过选项compiler\_pb\_uu unused\_declared\_through\_exception\_exception\_and\_throwable和compiler\_pb\_u unused\_declared\_through\_exception\_include\_doc\_comment\_reference进一步优化。 | **启用** |  |
| **残疾人** |  |
| **报告未引用标签**（编译器“未使用”标签） | |  |
| 启用后，编译器将在遇到从未显式引用标签的带标签语句时发出错误或警告。如果标签的名称显式出现在break或continue语句中，则认为该标签被引用；例如，以下标签将被视为未引用：label:break； | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告未使用的类型参数**（编译器\_pb\_unused\_type\_参数） | |  |
| 启用后，编译器将对未使用的类型参数发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告未使用的导入**（编译器未使用的导入） | |  |
| 启用后，编译器将对未使用的导入引用发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告未使用的本地**（编译器\_pb\_unused\_local） | |  |
| 启用后，编译器将对未使用的局部变量（即从未读取的变量）发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告未使用对象的分配**（编译器未使用的对象分配） | |  |
| 启用后，如果分配了一个对象但从未使用过，编译器将发出一个错误或警告，既不通过保持引用，也不通过调用对象的一个方法。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告未使用的参数**（编译器\_pb\_unused\_参数） | |  |
| 启用后，编译器将对未使用的方法参数（即从未读取的参数）发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **如果实现抽象方法，则报告未使用的参数**（在执行摘要时编译器未使用的参数） | |  |
| 启用后，编译器将在抽象方法实现中发出未使用参数的信号。 | **启用** |  |
| **残疾人** |  |
| **如果覆盖具体方法，则报告未使用的参数**（编译器\_pb\_unused\_参数\_覆盖\_concrete时） | |  |
| 启用后，编译器将在重写具体参数的方法中发出未使用参数的信号。 | **启用** |  |
| **残疾人** |  |
| **在文档注释中考虑未使用参数检查的引用**（编译器\_pb\_unused\_参数\_include\_doc\_comment\_reference） | |  |
| 启用后，编译器将考虑对参数（即@param子句）的文档注释引用，以进行未使用的参数检查。因此，文件化参数将被视为根据DOC合同规定的强制性参数。未使用参数问题的严重性由选项“org.eclipse.jdt.core.compiler.problem.unused parameter”控制。注意：在根据选项“org.eclipse.jdt.core.compiler.doc.comment.support”启用文档注释支持之前，此选项无效。 | **启用** |  |
| **残疾人** |  |
| **报告未使用的异常参数**（编译器\_pb\_unused\_exception\_参数） | |  |
| 启用后，编译器将对未使用的异常参数发出错误或警告（即，从不从中读取引发的异常）。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告未使用的私人成员**（编译器未使用的私有成员） | |  |
| 启用后，每当声明私有方法或字段但从未在同一单元中使用时，编译器将发出错误或警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告存在非泛型方法调用的类型参数**（编译器\u pb\_未使用的\u类型\u参数\u用于\u方法\u调用） | |  |
| 启用后，每当遇到非泛型方法调用的类型参数时，编译器将发出错误或警告。请注意，在符合性级别“1.7”之前，这种情况将自动导致错误。从Java 7上，未使用的类型参数被容忍，并且可选地警告。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告不必要的@suppresswarnings**（编译器\_pb\_unused\_warning\_token） | |  |
| 启用后，编译器将在遇到@SuppressWarnings注释时发出错误或警告，代码中未检测到相应的警告。此诊断可帮助开发人员消除不再需要的瞬时@suppresswarnings。请注意，@suppresswarnings（“all”）仍然会使不必要的@suppresswarnings的警告静音，因为它是主开关，用于使所有警告静音。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **报告需要强制转换方法/构造函数inv的varargs参数**（编译器变量参数需要转换） | |  |
| 启用后，每当将varargs参数传递给方法/构造函数调用时，编译器将发出错误或警告。（例如class.getmethod（字符串名称，类…args）使用参数调用（“foo”，null））。 | **误差** |  |
| **警告** |  |
| **信息** |  |
| **忽略** |  |
| **设置源兼容模式**（编译器来源） | |  |
| 指定Java源代码的兼容性级别。源级别“1.4”启用断言。从“1.4”开始，“assert”是保留关键字。源代码级别“1.5”支持对循环、静态导入和varargs增强的泛型、自动修改、协方差、注释、枚举。从“1.5”开始，“enum”是保留关键字。遵从性、源和目标级别必须满足下表中汇总的一组约束。 | **版本1** |  |
| **版本1** |  |
| **版本11.5** |  |
| **版本1 16** |  |
| **版本1 17** |  |
| **版本1 18** |  |
| **生成源调试属性**（编译器源文件） | |  |
| 生成时，此属性将使调试器能够显示相应的源代码。 | **生成** |  |
| **dotNOT生成** |  |
| **确定任务标记是否区分大小写**（编译器任务区分大小写） | |  |
| 启用后，任务标记将以区分大小写的方式考虑。 | **启用** |  |
| **残疾人** |  |
| **定义自动任务优先级**（编译器任务优先级） | |  |
| 与自动任务标记并行，此列表定义编译器发布的任务标记的优先级（高、正常或低）。可能的优先级是“高”、“正常”或“低”。 | <优先级>[，<优先级>]\*。默认值为“正常、高、正常” |  |
| **定义自动任务标记**（编译器任务标记） | |  |
| 当标签列表不是空的时候，编译器将在它遇到Java源代码中任何注释中的一个对应标签时发出任务标记。生成的任务消息将以标记和范围开始，直到下一行分隔符、注释结束或标记。当一行代码带有多个标记时，每个标记将分别报告。此外，使用行的下一个非空标记（如果有的话）的内容，一个标记紧接着另一个标记将被报告。请注意，任务消息已被剪裁。如果标记以字母或数字开头，则它不能由另一个要识别的字母或数字引导（“foo todo”不会被识别为标记“todo”的任务，但“foo todo”将被检测为标记“todo”或“todo”）。分别地，以字母或数字结尾的标记不能后跟要识别的字母或数字（“todo foo”将不被识别为标记“todo”的任务，但“todo:foo”将被检测为标记“todo”或“todo:”）。 | <tag>[，<tag>]\*。默认值为“todo，fixme，xxx” |  |

下表总结了Java遵从性、目标和源级别的兼容组合（粗体值是每个遵从级别的默认值）。

|  |  |  |
| --- | --- | --- |
| **顺从** | **靶标** | **来源** |
| **1.7** | **1.7** | **1.7**, 1.6, 1.5, 1.4, 1.3 |
| 1.6 | 1.6, 1.5, 1.4, 1.3 |
| 1.5 | 1.5, 1.4, 1.3 |
| 1.4 | 1.4, 1.3 |
| 1.3, 1.2, 1.1 | 1.3 |
| **1.6** | **1.6** | **1.6**, 1.5, 1.4, 1.3 |
| 1.5 | 1.5, 1.4, 1.3 |
| 1.4 | 1.4, 1.3 |
| 1.3, 1.2, 1.1 | 1.3 |
| **1.5** | **1.5** | **1.5**, 1.4, 1.3 |
| 1.4 | 1.4, 1.3 |
| 1.3, 1.2, 1.1 | 1.3 |
| **1.4** | 1.4 | 1.4, 1.3 |
| 1.3、1.2、1.1、CLDC1.1 | **1.3** |
| **1.3** | 1.3、1.2、1.1、CLDC1.1 | **1.3** |

**生成器选项**

|  |  |
| --- | --- |
| **描述** | **价值观** |
| **正在清理输出文件夹**（core\_java\_build\_clean\_output\_文件夹） | |
| 指示在执行完全生成操作时是否允许JavaBuilder清理输出文件夹。 | **干净的** |
| **忽略** |
| **报告重复资源**（core\_java\_build\_duplicate\_resource） | |
| 指示在将给定资源的多个事件复制到输出位置时报告的问题的严重性。 | **误差** |
| **警告** |
| **类路径无效时中止**（core\_java\_build\_无效的\_classpath） | |
| 如果类路径无效，指示生成器中止。 | **中止** |
| **忽略** |
| **计算项目生成顺序**（核心Java构建订单） | |
| 指示Javacore是否应根据类路径先决条件链强制执行项目构建顺序。当请求计算时，这将接管平台默认顺序（基于项目引用）。 | **计算** |
| **忽略** |
| **在输出文件夹中重新创建修改后的类文件**（core\_java\_build\_在\_output\_文件夹中重新创建修改过的\_class\_文件） | |
| 指示JavaBuilder在执行增量生成操作时是否应检查输出文件夹中.class文件的任何更改。如果检测到对托管.class文件的更改，则执行完整的生成，否则将保留更改。进一步改变生成的.class文件的工具，比如优化器，应该确保这个选项保持设置为默认的忽略状态。 | **使能** |
| **忽略** |
| **指定资源复制控件的筛选器**（核心Java构建资源复制过滤器） | |
| 指定筛选器以控制资源复制过程。（<name>是文件名模式（仅允许\*通配符）或以“/”结尾的文件夹的名称；名称与这些模式中的一个或多个模式匹配的任何资源都不会复制到输出文件夹。） | <name>[，<name>]\*。默认值为“”。 |

**Javacore选项**

|  |  |
| --- | --- |
| **描述** | **价值观** |
| **报告类路径周期**（核心\圆形\类路径） | |
| 指示当项目涉及到一个周期时报告的问题的严重性。 | **误差** |
| **警告** |
| **启用类路径排除模式的使用**（核心\启用\类路径\排除\模式） | |
| 当设置为“禁用”时，项目类路径上的任何条目都不能与排除或包含模式相关联。 | **启用** |
| **残疾人** |
| **允许使用类路径多个输出位置**（核心\启用\类路径\多个\输出\位置） | |
| 当设置为“禁用”时，项目类路径上的任何条目都不能与特定的输出位置相关联。特别是，这可以防止为单个项目使用多个输出位置。 | **启用** |
| **残疾人** |
| **报告一个输出位置与另一个源位置重叠**（核心输出位置重叠另一个来源） | |
| 指示源项的输出位置与另一个源项重叠时报告的问题的严重性。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **默认源编码格式**（核心编码） | |
| 获取源文件的默认编码格式。此值是不可变的，并预设为resourcesPlugin.getEncoding（）的结果。它仅作为方便快捷方式提供。 | **不可变，预设为平台默认值。** |
| **报告所需二进制文件的不兼容JDK级别**（核心不兼容级别） | |
| 指示当项目先决条件是目标JDK级别不兼容的另一个项目或库（例如，目标为1.1 vm，但根据1.4库编译的项目）时报告的问题的严重性。 | **误差** |
| **警告** |
| **信息** |
| **忽略** |
| **报告不完整的类路径**（核心不完整类路径） | |
| 指示当类路径上的条目不存在、不合法或不可见（例如，引用的项目已关闭）时报告的问题的严重性。 | **误差** |
| **警告** |
| **设置从JavaDoc检索方法参数名的超时值**（来自附加的javadoc的参数名称超时） | |
| 从JavaDoc检索方法参数名的超时（以毫秒计）。如果值为0，则不提取参数名，并返回原始名称。 | 正整数或空整数。默认值为“50”。 |
| **设置Java格式化程序名称**（Java\_格式化程序） | |
| 用于格式化Java代码的Java格式化程序的名称。 | 一个非空的sring值。默认值为“org.eclipse.jdt.core.defaultJavaFormatter”。 |

**格式化程序选项**

|  |  |
| --- | --- |
| **描述** | **价值观** |
| **用于对齐列上类型声明的类型成员的选项**（格式化程序在列上对齐\u类型的\u成员） | |
| 可能值 | **真的** |
| **错误的** |
| **用于在分配表达式中对齐参数的选项**（格式化程序在分配表达式中对参数进行对齐） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **用于在枚举常量中对齐参数的选项**（格式化程序“对齐”用于“枚举”常量中的“参数” | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **注释中参数对齐的选项**（格式化程序\_-Alignment\_for\_-Arguments\_in\_-Annotation） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_no\_split，indent\_default） |
| **用于在显式构造函数调用中对齐参数的选项**（格式化程序\_-Alignment\_for\_-Arguments\_in\_-Explicit\_-Constructor\_调用） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **方法调用中参数对齐的选项**（格式化程序在方法调用中对参数进行对齐） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **用于在限定的分配表达式中对齐参数的选项**（格式化程序\_-Alignment\_for\_-Arguments\_in\_-Qualified\_-Allocation\_表达式） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **Try with Resources语句中参数对齐的选项**（格式化程序对资源进行对齐） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（假，每行换行，缩进默认值） |
| **多catch表达式中并集类型的对齐选项**（格式化程序在multicatch中为union-type-u对齐） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **调整任务的选项**（格式化程序“对齐”用于“分配”） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（假，0，缩进默认值） |
| **二进制表达式的对齐选项**（格式化程序对二进制表达式进行对齐） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **紧凑型中频校准选项**（格式化程序\_-Alignment\_for\_-Compact\_-if） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（错误，每行换行一个，缩进一个） |
| **条件表达式的对齐选项**（格式化程序“对齐”用于“条件”表达式） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（假，每行换行一个，默认缩进） |
| **用于对齐枚举常量的选项**（格式化程序“对齐”用于“枚举”常量） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_no\_split，indent\_default） |
| **数组初始值设定项中表达式的对齐选项**（格式化程序“数组初始值设定项”中“表达式”的“对齐方式”） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **方法声明的对齐选项**（格式化程序\_-Alignment\_for\_-method\_声明） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **多个字段对齐选项**（格式化程序对多个字段进行对齐） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **用于在构造函数声明中对齐参数的选项**（格式化程序\u在\u构造函数\u声明中为\u参数\u对齐） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **方法声明中参数对齐的选项**（格式化程序在方法声明中对参数进行对齐） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **方法调用中选择器对齐的选项**（格式化程序\_-Alignment\_for\_-Selector\_in\_-method\_invocation） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **用于在类型声明中对齐超类的选项**（格式化程序\_-type\_声明中\_超类\_的对齐方式） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（错误，换行后移位，缩进默认值） |
| **用于在枚举声明中对齐超级接口的选项**（格式化程序“对齐”用于“枚举”声明中的“超级接口”） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **用于在类型声明中对齐超级接口的选项**（格式化程序在“类型”声明中为“超级接口”对齐） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **构造函数声明中throws子句的对齐选项**（格式化程序在构造函数声明中对子句进行对齐） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **方法声明中throws子句的对齐选项**（格式化程序在方法声明中对子句进行对齐） | |
| 可能值 | createAligmentValue（boolean、int、int）调用返回的值 |
| 默认值 | CreateAlignmentValue（false，wrap\_compact，indent\_default） |
| **在导入声明后添加空行的选项**（格式化程序\_导入后空白\_行） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “0” |
| **在包声明后添加空行的选项**（格式化程序\u空白\u行在\u包之后） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “0” |
| **在方法体的开头添加空行的选项**（格式化程序\u空白\u行在\u方法\u体的\u开始处） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “0” |
| **在字段声明前添加空行的选项**（格式化程序\_空白\_行\_前\_字段） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “0” |
| **在第一个类主体声明之前添加空行的选项**（格式化程序\_空白\_行\_第一类\_正文\_声明之前） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “0” |
| **在导入声明前添加空行的选项**（格式化程序\_导入前空白\_行） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “0” |
| **在成员类型声明前添加空行的选项**（格式化程序在成员类型之前\u空白\u行） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “0” |
| **在方法声明前添加空行的选项**（格式化程序\_blank\_lines\_before\_method） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “0” |
| **在新块之前添加空行的选项**（格式化程序\u新块前空白\u行） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “0” |
| **在包声明前添加空行的选项**（格式化程序\u空白\u行在\u包之前） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “0” |
| **在导入组之间添加空行的选项**（格式化程序\_空白\_行\_导入\_组之间） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “1” |
| **在类型声明之间添加空行的选项**（格式化程序\_空白\_行\_类型\_声明之间） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “0” |
| **用于定位批注类型声明的大括号的选项**（格式化程序\_brace\_position\_for\_annotation\_type\_声明） | |
| 可能值 | **尾线** |
| **下一条线** |
| **下一条线移动了** |
| **包装上的下一行** |
| **用于定位匿名类型声明的大括号的选项**（格式化程序\_brace\_position\_for\_anonymous\_type\_声明） | |
| 可能值 | **尾线** |
| **下一条线** |
| **下一条线移动了** |
| **包装上的下一行** |
| **用于定位数组初始值设定项的大括号的选项**（格式化程序\_brace\_position\_for\_array\_initializer） | |
| 可能值 | **尾线** |
| **下一条线** |
| **下一条线移动了** |
| **包装上的下一行** |
| **放置块大括号的选项**（格式化程序\_brace\_position\_for\_block） | |
| 可能值 | **尾线** |
| **下一条线** |
| **下一条线移动了** |
| **包装上的下一行** |
| **当块是后面的第一条语句时，在case语句中放置块大括号的选项**（格式化程序\_brace\_position\_for\_block\_in\_case） | |
| 可能值 | **尾线** |
| **下一条线** |
| **下一条线移动了** |
| **包装上的下一行** |
| **用于定位构造函数声明的大括号的选项**（格式化程序\_brace\_position\_for\_constructor\_声明） | |
| 可能值 | **尾线** |
| **下一条线** |
| **下一条线移动了** |
| **包装上的下一行** |
| **用于定位枚举常量大括号的选项**（格式化程序\_brace\_position\_for\_enum\_constant） | |
| 可能值 | **尾线** |
| **下一条线** |
| **下一条线移动了** |
| **包装上的下一行** |
| **用于定位枚举声明大括号的选项**（格式化程序\_brace\_position\_for\_enum\_声明） | |
| 可能值 | **尾线** |
| **下一条线** |
| **下一条线移动了** |
| **包装上的下一行** |
| **用于定位方法声明大括号的选项**（格式化程序\_brace\_position\_for\_method\_声明） | |
| 可能值 | **尾线** |
| **下一条线** |
| **下一条线移动了** |
| **包装上的下一行** |
| **用于定位switch语句大括号的选项**（格式化程序\_brace\_position\_for\_switch） | |
| 可能值 | **尾线** |
| **下一条线** |
| **下一条线移动了** |
| **包装上的下一行** |
| **用于定位类型声明的大括号的选项**（格式化程序\_brace\_position\_for\_type\_声明） | |
| 可能值 | **尾线** |
| **下一条线** |
| **下一条线移动了** |
| **包装上的下一行** |
| **控制是否在块注释内清除空行的选项**（格式化程序\_comment\_clear\_blank\_lines\_in\_block\_comment） | |
| 可能值 | **真的** |
| **错误的** |
| **控制JavaDoc注释中是否清除空白行的选项**（格式化程序\_comment\_clear\_blank\_lines\_in\_javadoc\_comment） | |
| 可能值 | **真的** |
| **错误的** |
| **用于控制是否格式化多行注释的选项**（格式化程序\_comment\_format\_block\_comment） | |
| 可能值 | **真的** |
| **错误的** |
| **用于控制是否格式化JavaDoc注释的选项**（格式化程序\_comment\_format\_javadoc\_comment） | |
| 可能值 | **真的** |
| **错误的** |
| **用于控制是否格式化单行注释的选项**（格式化程序注释\u格式化\u行注释） | |
| 可能值 | **真的** |
| **错误的** |
| **设置从第一列开始的行注释格式的选项**（格式化程序\_comment\_format\_line\_comment\_starting\_on\_first\_column） | |
| 可能值 | **真的** |
| **错误的** |
| **选项来控制Java源文件的头注释是否已格式化。**（格式化程序注释格式标题） | |
| 可能值 | **真的** |
| **错误的** |
| **选项控制是否格式化HTML标记。**（格式化程序注释格式HTML） | |
| 可能值 | **真的** |
| **错误的** |
| **用于控制代码段是否在注释中格式化的选项**（格式化程序注释格式源） | |
| 可能值 | **真的** |
| **错误的** |
| **用于控制是否缩进JavaDoc参数描述的选项**（格式化程序注释缩进参数描述） | |
| 可能值 | **真的** |
| **错误的** |
| **选项控制JavaDoc根标记是否缩进。**（格式化程序注释缩进根标记） | |
| 可能值 | **真的** |
| **错误的** |
| **在javadoc根标记块之前插入空行的选项**（格式化程序\_comment\_在\_root\_标记前插入\_empty\_line\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在javadoc根标记参数后插入新行的选项**（格式化程序\_comment\_insert\_new\_line\_for\_参数） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **选项指定注释的行长度。**（格式化程序注释行长度） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “80” |
| **控制块注释是否在边界处有新行的选项**（格式化程序注释\u块边界处的新行） | |
| 可能值 | **真的** |
| **错误的** |
| **控制javadoc注释是否在边界处有新行的选项**（格式设置工具\_comment\_new\_lines\_at\_javadoc\_bounders） | |
| 可能值 | **真的** |
| **错误的** |
| **保留代码和行注释之间现有空白的选项**（格式化程序\_comment\_保留\_code\_和\_line\_comment之间的\u空格） | |
| 可能值 | **真的** |
| **错误的** |
| **压缩else/if的选项**（格式化程序压缩其他格式） | |
| 可能值 | **真的** |
| **错误的** |
| **用于设置连续缩进的选项**（格式化程序继续缩进） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “2” |
| **在数组初始值设定项内设置连续缩进的选项**（格式化程序\_continue\_indentation\_for\_array\_initializer） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “2” |
| **使用禁用和启用标记的选项**（格式化程序使用“开/关”标签） | |
| 可能值 | **真的** |
| **错误的** |
| **用于定义要放入注释中的标记以禁用格式设置的选项**（格式化程序禁用标记） | |
| 可能值 | 弦 |
| 默认值 | “@格式：关闭” |
| **选项定义要放入注释中的标记，以便在禁用后重新启用格式设置**（格式化程序启用标记） | |
| 可能值 | 弦 |
| 默认值 | “@格式：on” |
| **用于将正文声明与其封闭的批注声明头进行比较的缩进选项**（格式化程序\_indent\_body\_declarations\_compare\_to\_annotation\_declaration\_header） | |
| 可能值 | **真的** |
| **错误的** |
| **用于与封闭枚举常量头比较的缩进正文声明的选项**（格式化程序\_indent\_body\_declarations\_compare\_to\_enum\_constant\_header） | |
| 可能值 | **真的** |
| **错误的** |
| **用于与封闭枚举声明头比较的缩进正文声明的选项**（格式化程序\_indent\_body\_declarations\_compare\_to\_enum\_declaration\_header） | |
| 可能值 | **真的** |
| **错误的** |
| **用于与封闭类型头比较的缩进正文声明的选项**（格式化程序\_indent\_body\_declarations\_compare\_to\_type\_header） | |
| 可能值 | **真的** |
| **错误的** |
| **与事例相比缩进分隔符的选项**（格式化程序缩进分隔符比较大小写） | |
| 可能值 | **真的** |
| **错误的** |
| **空行缩进选项**（格式化程序\_缩进\_空行） | |
| 可能值 | **真的** |
| **错误的** |
| **在块内缩进语句的选项**（格式化程序\_indent\_语句\_compare\_to\_块） | |
| 可能值 | **真的** |
| **错误的** |
| **在方法或构造函数的主体内缩进语句的选项**（格式化程序缩进语句与正文比较） | |
| 可能值 | **真的** |
| **错误的** |
| **用于缩进switch语句与cases的选项**（格式化程序\_indent\_switchstatements\_compare\_to\_cases） | |
| 可能值 | **真的** |
| **错误的** |
| **用于缩进switch语句的选项与switch进行比较**（格式化程序\_indent\_switchstatements\_compare\_to\_switch） | |
| 可能值 | **真的** |
| **错误的** |
| **用于指定表示一个缩进的等量空格数的选项**（格式化程序缩进大小） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “4” |
| **在参数的批注后插入新行的选项**（格式化程序在参数上的注释后插入新行） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在字段声明的批注后插入新行的选项**（格式化程序在域上的注释后插入新行） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法声明的批注后插入新行的选项**（格式化程序在方法上的注释后插入新行） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在包声明的批注后插入新行的选项**（格式化程序在\u包上的\u注释\u后插入\u新行） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在类型声明的批注后插入新行的选项**（格式化程序在类型上的注释后插入新行） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在局部变量的注释后插入新行的选项**（格式化程序在本地变量上的注释后插入新行） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在标签后插入新行的选项**（格式化程序在标签后插入新行） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组初始值设定项中的左大括号后插入新行的选项**（格式化程序在数组初始值设定项中打开大括号后插入新行） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **如果缺少，则在当前文件结尾插入新行的选项**（格式化程序在文件的结尾插入新的行，如果丢失） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在try语句中catch关键字之前插入新行的选项**（格式化程序在“try”语句中的“catch”之前插入“new”行） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组初始值设定项中的右大括号之前插入新行的选项**（格式化程序在数组初始值设定项中关闭大括号之前插入新行） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在if语句中的else关键字前面插入新行的选项**（格式化程序在if语句中的\u else之前插入\u新行） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在try语句的finally关键字前插入新行的选项**（格式化程序在“try”语句中的“finally”之前插入“new”行） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在do语句中的while之前插入新行的选项**（格式化程序\_在\_do\_语句中的\_之前插入\_新行\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在空批注声明中插入新行的选项**（格式化程序\_在\_empty\_annotation\_声明中插入\_new\_line\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在空匿名类型声明中插入新行的选项**（格式化程序\_在\_empty\_anonymous\_type\_声明中插入\_new\_line\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在空块中插入新行的选项**（格式化程序在空块中插入新行） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在空枚举常量中插入新行的选项**（格式化程序在空枚举常量中插入新行） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在空枚举声明中插入新行的选项**（格式化程序\_在\_empty\_enum\_声明中插入\_new\_line\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在空方法体中插入新行的选项**（格式化程序\_在\_empty\_method\_body中插入\_new\_line\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在空类型声明中插入新行的选项**（格式化程序\_在\_empty\_type\_声明中插入\_new\_line\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在wilcard后和中插入空格的选项**（格式化程序\_在\_type\_参数中的\_和\_后面插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在赋值运算符后插入空格的选项**（格式化程序在分配运算符后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在“In”批注中的处后插入空格的选项**（格式化程序在注释中的“at”后插入“space”） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在注释类型声明中的at后面插入空格的选项**（格式化程序\_在\_注释\_类型\_声明中的\_at\_后面插入\_空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在二元运算符后插入空格的选项**（格式化程序在二进制运算符后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在类型参数中，在右尖括号后插入空格的选项**（格式化程序\_在\_type\_参数中的\_closing\_angle\_bracket\_后面插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在类型参数中，在闭合角括号后插入空格的选项**（格式化程序\_在关闭\_type\_参数中的\_angle\_bracket\_后插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在块的右大括号后插入空格的选项**（格式化程序在关闭块中的大括号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在转换表达式的右括号后插入空格的选项**（格式化程序在关闭后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在assert语句中在冒号后面插入空格的选项**（格式化程序在断言中的冒号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在case语句中，当左大括号跟在冒号后面时，在冒号后面插入空格的选项**（格式化程序在冒号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在条件表达式的冒号后面插入空格的选项**（格式化程序在条件中的冒号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在for语句中冒号后插入空格的选项**（格式化程序在冒号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在带标签的语句中，在冒号后插入空格的选项**（格式化程序在带标记的语句中的冒号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **选项在分配表达式中逗号后插入空格**（格式化程序在分配表达式中的逗号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在注释中逗号后插入空格的选项**（格式化程序在注释中的逗号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组初始值设定项中的逗号后插入空格的选项**（格式化程序在数组初始值设定项中的逗号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在构造函数声明的参数中逗号后插入空格的选项**（格式化程序在\u构造函数\u声明\u参数中的\u逗号\u后插入\u空格\u） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在构造函数声明的throws子句中的异常名称中，在逗号后插入空格的选项**（格式化程序在\u构造函数\u声明\u throws中的\u逗号\u后插入\u空格\u） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **选项在枚举常量的参数中逗号后插入空格**（格式化程序\_在\_enum\_constant\_参数中的\_comma\_后面插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在枚举声明中在逗号后插入空格的选项**（格式化程序在枚举声明中的逗号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在显式构造函数调用的参数中，在逗号后插入空格的选项**（格式化程序在\u显式\u构造函数\u调用\u参数中的\u逗号\后插入\u空格\u） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **选项以for语句的增量在逗号后插入空格**（格式化程序在逗号后插入空格，以\u为增量） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在for语句的初始化过程中，在逗号后插入空格的选项**（格式化程序在逗号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法声明的参数中的逗号后插入空格的选项**（格式化程序在方法声明参数中的逗号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法声明的throws子句的异常名称中，在逗号后插入空格的选项**（格式化程序在方法中的逗号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法调用的参数中，在逗号后插入空格的选项**（格式化程序在方法调用参数中的逗号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在多字段声明中在逗号后插入空格的选项**（格式化程序在多个字段声明中的逗号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在多个本地声明中在逗号后插入空格的选项**（格式化程序在多个本地声明中的逗号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在参数化类型引用中，在逗号后插入空格的选项**（格式化程序在参数化类型参考中的逗号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **选项在类型头的超级接口名称中的逗号后插入空格**（格式化程序在超级接口中的逗号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在类型参数的逗号后插入空格的选项**（格式化程序在类型参数中的逗号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在类型参数中的逗号后插入空格的选项**（格式化程序\_在\_type\_参数中的\_comma\_后面插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在省略号后插入空格的选项**（格式化程序在省略号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在参数化类型引用中，在开口角括号后插入空格的选项**（格式化程序在“参数化”类型“参考”中的“打开”角度“括号”后插入“空格”） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在类型参数的左尖括号后插入空格的选项**（格式化程序在“类型”参数中的“打开角度”括号后插入“空格”） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在类型参数中，在开口角括号后插入空格的选项**（格式化程序在“类型”参数中的“打开角度”括号后插入“空格”） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组初始值设定项中的左大括号后插入空格的选项**（格式化程序在数组初始值设定项中打开大括号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组分配表达式的左括号后插入空格的选项**（格式化程序在数组分配表达式中打开括号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组引用中的左括号后插入空格的选项**（格式化程序在数组引用中打开括号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在注释中的左括号后插入空格的选项**（格式化程序在打开注释后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在转换表达式中的左括号后插入空格的选项**（格式化程序在打开后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在catch中的左括号后插入空格的选项**（格式化程序在“捕获”中打开“paren”后插入“空格”） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在构造函数声明中的左括号后插入空格的选项**（格式化程序在构造函数声明中打开paren后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在枚举常量的左括号后插入空格的选项**（格式化程序\_在打开\_enum\_常量中的\_paren\_后插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在for语句的左括号后插入空格的选项**（格式化程序在打开后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在if语句的左括号后插入空格的选项**（格式化程序在打开后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法声明中的左括号后插入空格的选项**（格式化程序在方法声明中打开paren后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法调用中的左括号后插入空格的选项**（格式化程序在方法调用中打开paren后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在带圆括号的表达式中的左括号后插入空格的选项**（格式化程序在用括号括起来的表达式中打开后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在switch语句中的左括号后插入空格的选项**（格式化程序在打开开关后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在同步语句的左括号后插入空格的选项**（格式化程序在“同步”中打开“paren”后插入“空格”）。 | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在try with resources语句的左括号后插入空格的选项**（格式化程序在打开后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在while语句的左括号后插入空格的选项**（格式化程序在打开后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在后缀运算符后插入空格的选项**（格式化程序在后缀运算符后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在前缀运算符后插入空格的选项**（格式化程序在前缀运算符后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在条件表达式中问号后插入空格的选项**（格式化程序在条件中的问题后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在通配符中问号后插入空格的选项**（格式化程序在通配符中的问题后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在for语句中分号后插入空格的选项**（格式化程序在分号后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在try with resources语句的每个分号后插入空格的选项**（格式化程序在“尝试资源”中的“分号”后插入“空格”） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在一元运算符后插入空格的选项**（格式化程序在一元运算符后插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在通配符之前和中插入空格的选项**（格式化程序\_在\_type\_参数的\_和\_之前插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在赋值运算符前插入空格的选项**（格式化程序在分配运算符前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在批注类型声明中的at之前插入空格的选项**（格式化程序\_在\_注释\_类型\_声明中的\_处插入\_空格\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在二元运算符前插入空格的选项**（格式化程序在二进制运算符前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在参数化类型引用中的右尖括号前插入空格的选项**（格式化程序\_在\_参数化\_类型\_参考中关闭\_角度\_括号\_之前插入\_空格\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在类型参数中的右尖括号前插入空格的选项**（格式化程序\_在\_type\_参数中的\_closing\_angle\_bracket\_之前插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在类型参数中，在右尖括号前插入空格的选项**（格式化程序\_在关闭\_type\_参数中的\_angle\_bracket\_之前插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组初始值设定项中的右大括号之前插入空格的选项**（格式化程序在数组初始值设定项中关闭大括号之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组分配表达式中的右括号前插入空格的选项**（格式化程序\_在\_array\_allocation\_表达式中的\_closing\_bracket\_之前插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组引用中的右括号前插入空格的选项**（格式化程序在关闭数组引用中的括号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在注释中的右括号前插入空格的选项**（格式化程序在关闭注释前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在转换表达式中的右括号前插入空格的选项**（格式化程序在关闭演员表前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在catch中的右括号前插入空格的选项**（格式化程序在关闭捕获中的paren之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在构造函数声明中的右括号前插入空格的选项**（格式化程序在关闭构造函数声明中的paren之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在枚举常量的右括号前插入空格的选项**（格式化程序\_在关闭\_enum\_常量中的\_paren\_之前插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在for语句的右括号前插入空格的选项**（格式化程序在关闭前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在if语句的右括号前插入空格的选项**（格式化程序在关闭前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法声明中的右括号前插入空格的选项**（格式化程序在关闭方法声明中的paren之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法调用中的右括号前插入空格的选项**（格式化程序在关闭方法调用中的paren之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在带圆括号的表达式中的右括号前插入空格的选项**（格式化程序在用括号括起来的表达式中关闭paren之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在switch语句中的右括号前插入空格的选项**（格式化程序在关闭开关前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **选项在同步语句的右括号前插入空格**（格式化程序在“已同步”中关闭“paren”之前插入“空格”） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在try with resources语句中关闭paranthesis之前插入空格的选项**（格式化程序在关闭前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在while语句的右括号前插入空格的选项**（格式化程序在关闭前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在assert语句中在冒号前插入空格的选项**（格式化程序在断言中的冒号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在case语句中在冒号前插入空格的选项**（格式化程序在小写的冒号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在条件表达式中的冒号前插入空格的选项**（格式化程序在条件中的冒号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在默认语句中在冒号前插入空格的选项**（格式化程序在“默认”中的“冒号”前插入“空格”） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在for语句中冒号前插入空格的选项**（格式化程序在为中的冒号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在带标签的语句中，在冒号前插入空格的选项**（格式化程序在带标记的语句中的冒号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在分配表达式中，在逗号之前插入空格的选项**（格式化程序在分配表达式中的逗号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在批注中逗号前插入空格的选项**（格式化程序在注释中的逗号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组初始值设定项中，在逗号之前插入空格的选项**（格式化程序在数组初始值设定项中的逗号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在构造函数声明的参数中，在逗号之前插入空格的选项**（格式化程序在\u构造函数\u声明\u参数中的\u逗号\u之前插入\u空格\u） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在构造函数声明的throws子句的异常名称中，在逗号之前插入空格的选项**（格式化程序在\u构造函数\u声明\u throws中的\u逗号\u之前插入\u空格\u） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在枚举常量的参数中，在逗号之前插入空格的选项**（格式化程序在\u枚举\u常量\u参数中的\u逗号\前插入\u空格\ u） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在枚举声明中，在逗号之前插入空格的选项**（格式化程序在枚举声明中的逗号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在显式构造函数调用的参数中，在逗号之前插入空格的选项**（格式化程序在\u显式\u构造函数\u调用\u参数中的\u逗号\u之前插入\u空格\u） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **选项以for语句的增量在逗号前插入空格**（格式化程序在逗号前插入空格，以\u为增量） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在for语句的初始化中，在逗号之前插入空格的选项**（格式化程序在逗号前插入空格 | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法声明的参数中，在逗号之前插入空格的选项**（格式化程序在方法声明参数中的逗号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法声明的throws子句的异常名称中，在逗号之前插入空格的选项**（格式化程序在方法声明中的逗号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法调用的参数中，在逗号之前插入空格的选项**（格式化程序在方法调用参数中的逗号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在多字段声明中在逗号前插入空格的选项**（格式化程序在多个字段声明中的逗号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在多个本地声明中，在逗号之前插入空格的选项**（格式化程序在多个本地声明中的逗号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在参数化类型引用中，在逗号之前插入空格的选项**（格式化程序在参数化类型参考中的逗号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **选项在类型标题的超级接口名称中逗号前插入空格**（格式化程序在超级接口中的逗号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在类型参数中，在逗号之前插入空格的选项**（格式化程序在类型参数中的逗号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在类型参数中，在逗号之前插入空格的选项**（格式化程序\_在\_type\_参数中的\_comma\_之前插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在省略号前插入空格的选项**（格式化程序在省略号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在参数化类型引用中，在开口角括号之前插入空格的选项**（格式化程序在“参数化”类型“参考”中的“打开”角度“括号”前插入“空格” | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在类型参数中的左尖括号前插入空格的选项**（格式化程序在“u-type”参数中的“u-opening”角“bracket”前插入“u-space” | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在类型参数中，在开口角度括号之前插入空格的选项**（格式化程序在“类型”参数中的“打开角度”括号前插入“空格”） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在批注类型声明中的左大括号之前插入空格的选项**（格式化程序在注释类型声明中打开大括号之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在匿名类型声明中的左大括号前插入空格的选项**（格式化程序在匿名类型声明中打开大括号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组初始值设定项中的左大括号之前插入空格的选项**（格式化程序在数组初始值设定项中打开大括号之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在块中的左大括号之前插入空格的选项**（格式化程序在打开块中的大括号之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在构造函数声明中的左大括号前插入空格的选项**（格式化程序在“构造函数”声明中打开大括号之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在枚举常量的左大括号前插入空格的选项**（格式化程序在“枚举”常量中打开大括号之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在枚举声明中的左大括号之前插入空格的选项**（格式化程序在枚举声明中打开大括号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法声明中的左大括号之前插入空格的选项**（格式化程序在方法声明中打开大括号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在switch语句中的左大括号前插入空格的选项**（格式化程序在打开开关中的大括号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在类型声明中的左大括号前插入空格的选项**（格式化程序在“u type”声明中的“u opening”大括号之前插入“u space”） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组分配表达式的左括号前插入空格的选项**（格式化程序\_在\_array\_allocation\_表达式中的\_opening\_bracket\_之前插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组引用中的左括号前插入空格的选项**（格式化程序在打开数组引用中的括号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组类型引用中的左括号前插入空格的选项**（格式化程序\_在\_array\_type\_reference中的\_opening\_bracket\_之前插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在注释中的左括号前插入空格的选项**（格式化程序在打开注释前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在批注类型成员声明中的左括号前插入空格的选项**（格式化程序在“注释”类型“成员”声明中打开“paren”之前插入“空格” | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在catch中的左括号前插入空格的选项**（格式化程序在打开\u catch中的\u paren\_之前插入\u space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在构造函数声明中的左括号前插入空格的选项**（格式化程序在“构造函数”声明中打开“paren”之前插入“空格”）。 | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在枚举常量的左括号前插入空格的选项**（格式化程序\_在打开\_enum\_常量中的\_paren\_之前插入\_space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在for语句的左括号前插入空格的选项**（格式化程序在打开之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在if语句的左括号前插入空格的选项**（格式化程序在打开“if”中的“paren”之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法声明中的左括号前插入空格的选项**（格式化程序在方法声明中打开paren之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法调用中的左括号前插入空格的选项**（格式化程序在调用方法之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在带圆括号的表达式中的左括号前插入空格的选项**（格式化程序在用括号括起来的表达式中打开paren之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在switch语句中的左括号前插入空格的选项**（格式化程序在打开开关前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在同步语句的左括号前插入空格的选项**（格式化程序在打开“同步”中的“paren”之前插入“空格”）。 | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在try with resources语句的左括号前插入空格的选项**（格式化程序在打开之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在while语句的左括号前插入空格的选项**（格式化程序在打开之前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在RETURN语句中的带括号表达式之前插入空格的选项**（格式化程序在返回时用括号括起来的表达式前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在throw语句中的括号表达式之前插入空格的选项**（格式化程序在\u括号前插入\u空格\u在\u throw中插入\u表达式） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在后缀运算符之前插入空格的选项**（格式化程序在后缀运算符前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在前缀运算符前插入空格的选项**（格式化程序在前缀运算符前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在条件表达式中的问号前插入空格的选项**（格式化程序在条件中的问题前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在通配符中问号之前插入空格的选项**（格式化程序在通配符中的问题前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在分号前插入空格的选项**（格式化程序在分号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在for语句中的分号前插入空格的选项**（格式化程序在分号前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **选项在try with resources语句中的每个分号前插入一个空格**（格式化程序在“尝试资源”中的“分号”前插入“空格”） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在一元运算符前插入空格的选项**（格式化程序在一元运算符前插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组类型引用中的括号之间插入空格的选项**（格式化程序在数组类型参考中的括号之间插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组初始值设定项中的空大括号之间插入空格的选项**（格式化程序在数组初始值设定项中的空大括号之间插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在数组分配表达式中的空括号之间插入空格的选项**（格式化程序在数组分配表达式中的空括号之间插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在批注类型成员声明中的空括号之间插入空格的选项**（格式化程序在注释类型成员声明中的空参数之间插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在构造函数声明中的空括号之间插入空格的选项**（格式化程序在\u构造函数\u声明中的\u空\u parens \u之间插入\u空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在枚举常量中的空括号之间插入空格的选项**（格式化程序在\u enum\_常量中的\u empty\_parens\_之间插入\u space\_） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法声明中的空括号之间插入空格的选项**（格式化程序在方法声明中的空参数之间插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **在方法调用中在空括号之间插入空格的选项**（格式化程序在方法调用中的空参数之间插入空格） | |
| 可能值 | **插入** |
| **多诺特插入** |
| **将else语句保持在同一行上的选项**（格式化程序将其他语句放在同一行） | |
| 可能值 | **真的** |
| **错误的** |
| **将空数组初始值设定项保留一行的选项**（格式化程序在一行上保留空数组初始值设定项） | |
| 可能值 | **真的** |
| **错误的** |
| **在一行上保留监护人条款的选项**（格式化程序将“监护人”条款放在“一行”上） | |
| 可能值 | **真的** |
| **错误的** |
| **在一行上保留简单if语句的选项**（格式化程序在一行保持简单） | |
| 可能值 | **真的** |
| **错误的** |
| **将THEN语句保持在同一行的选项**（格式化程序将语句放在同一行） | |
| 可能值 | **真的** |
| **错误的** |
| **用于指定页面长度的选项。超过此长度后，格式化程序将尝试拆分代码**（格式化程序\_line\_split） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “80” |
| **用于缩进从第一列开始的块注释的选项**（格式化程序“从不”在“第一列”上缩进“块”注释） | |
| 可能值 | **真的** |
| **错误的** |
| **用于缩进从第一列开始的行注释的选项**（格式化程序“从不”在“第一列”上缩进“行”注释） | |
| 可能值 | **真的** |
| **错误的** |
| **指定要保留的空行数的选项**（格式化程序\u空\u行的编号\u保存） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “0” |
| **用于指定空语句是否应在新行上的选项**（格式化程序将空语句放在新行上） | |
| 可能值 | **真的** |
| **错误的** |
| **指定表格大小的选项**（格式化程序\_tab\_char） | |
| 可能值 | **桌棋类游戏** |
| **空间** |
| **混合的** |
| **用于指定表示一个表格的等效空格数的选项**（格式化程序标签大小） | |
| 可能值 | “<n>”，其中n是零或正整数 |
| 默认值 | “4” |
| **仅对前导缩进使用表格的选项**（格式化程序\u使用\u制表符\u仅用于\u前导\u缩进） | |
| 可能值 | **真的** |
| **错误的** |
| **在二进制运算符之前换行的选项**（格式化程序在二进制运算符之前包装） | |
| 可能值 | **真的** |
| **错误的** |
| **在多catch表达式中的运算符或运算符之前换行的选项**（格式化程序\_在\_或\_操作员\_multicatch之前包装\_） | |
| 可能值 | **真的** |
| **错误的** |
| **在嵌套表达式中包装外部表达式的选项**（格式化程序在嵌套时包装外部表达式） | |
| 可能值 | **真的** |
| **错误的** |

**代码辅助选项**

|  |  |
| --- | --- |
| **描述** | **价值观** |
| **定义参数名称的前缀**（codeassist参数前缀） | |
| 当前缀不为空时，参数名称的完成将以建议的一个前缀开始。 | <前缀>[，<前缀>]\*。默认值为“”。 |
| **定义参数名称的后缀**（codeassist\_参数\_后缀） | |
| 当后缀非空时，参数名的完成将以建议的后缀之一结束。 | <后缀>[，<后缀>]\*。默认值为“”。 |
| **激活camel区分大小写完成**（codeassist\_camel\_case\_match） | |
| 启用后，完成显示名称与camelcase模式匹配的建议。 | **启用** |
| **残疾人** |
| **激活子字符串代码完成**（代码辅助\子字符串\匹配） | |
| 启用后，完成将以不区分大小写的方式显示模式可以作为子字符串的建议。 | **启用** |
| **残疾人** |
| **启用折旧敏感完成**（codeassist\_deprecation\_check） | |
| 启用后，完成不会建议不推荐使用的成员和类型。 | **启用** |
| **残疾人** |
| **激活不鼓励的参考敏感完成**（codeassist不鼓励参考检查） | |
| 启用后，完成操作不会建议与不鼓励的引用规则匹配的元素。 | **启用** |
| **残疾人** |
| **定义字段名的前缀**（codeassist\_field\_前缀） | |
| 当前缀不为空时，字段名的完成将以一个建议的前缀开始。 | <前缀>[，<前缀>]\*。默认值为“”。 |
| **定义字段名的后缀**（codeassist\_field\_后缀） | |
| 当后缀非空时，字段名的完成将以建议的后缀之一结束。 | <后缀>[，<后缀>]\*。默认值为“”。 |
| **激活禁止的参考敏感完成**（codeassist\_禁止参考\_检查） | |
| 启用后，完成不会建议与禁止引用规则匹配的元素。 | **启用** |
| **残疾人** |
| **隐式成员的自动限定**（codeassist\_隐式\_限定） | |
| 启用后，完成将自动限定隐式字段引用和消息表达式的完成。 | **启用** |
| **残疾人** |
| **定义局部变量名的前缀**（codeassist\_local\_前缀） | |
| 当前缀不为空时，本地变量名的完成将以一个建议的前缀开始。 | <前缀>[，<前缀>]\*。默认值为“”。 |
| **定义局部变量名的后缀**（codeassist\_本地\_后缀） | |
| 当后缀非空时，局部变量名的完成将以建议的后缀之一结束。 | <后缀>[，<后缀>]\*。默认值为“”。 |
| **定义静态字段名的前缀**（codeassist\_static\_field\_前缀） | |
| 当前缀不为空时，静态字段名的完成将以一个建议的前缀开始。 | <前缀>[，<前缀>]\*。默认值为“”。 |
| **定义静态字段名的后缀**（codeassist\_static\_field\_后缀） | |
| 当后缀非空时，静态字段名的完成将以建议的后缀之一结束。 | <后缀>[，<后缀>]\*。默认值为“”。 |
| **激活静态导入建议**（codeassist\_建议静态\_导入） | |
| 启用后，完成建议可以包含静态导入模式。 | **启用** |
| **残疾人** |
| **激活可见性敏感完成**（codeassist\_可见性检查） | |
| 启用后，完成不会根据Java可见性规则提出插入点不可见的元素（例如，不建议使用超级类的私有方法）。 | **启用** |
| **残疾人** |

**格式化Java代码**

**使用代码格式化程序**

JDT API允许其他插件使用默认的代码格式化程序来格式化源代码。要考虑的两种方法是：

* **toolfactory.createCodeFormatter（映射）**
* **toolfactory.createCodeFormatter（映射，int）**

注意：CodeFormatter类不打算被客户机子类化。

**获取代码格式化程序实例**

可以调用toolfactory上的工厂方法来创建默认代码格式化程序的新实例。在调用其中一个之前，需要定义一个包含代码格式化程序选项的映射。为了创建这样的映射，可以使用类DefaultCodeFormatterConstants中定义的方法，如DefaultCodeFormatterConstants.GetEclipseDefaultSettings（）。

注意：这些预定义的映射只包含代码格式化程序特定的选项。为了调用代码格式化程序，还需要指定代码格式化程序将格式化的源类型。为此，请指定三个选项：

* **javacore.compiler\_codegen\_target\_平台**
* **javacore.compiler\_源代码**
* **javacore.compiler\_符合性**

这些选项的可能值由常数给出：

* **javacore.version\_1\_1版**
* **javacore.version\_1\_2**
* **javacore.version\_1\_3版**
* **javacore.version\_1\_4**
* **javacore.version\_1\_5**
* **javacore.version\_1\_6**
* **javacore.version\_1\_7**
* **javacore.version\_1\_8**
* **javacore.version\_9**

如果要修改默认映射，建议使用在DefaultCodeFormatterConstants上定义的方法来创建相应选项的值。这对于与代码包装相关的选项尤其适用。

**调用代码格式化程序**

使用新创建的代码格式化程序来格式化代码段。默认的代码格式化程序允许您格式化不同类型的代码段。这些类型在格式化方法的文档中指定。此方法的返回值是文本编辑。然后需要将此文本编辑应用于IDocument实例，以便获得格式化的结果。

**例子**

//采用默认Eclipse格式选项

map options=defaultcodeFormatterConstants.getEclipseDefaultSettings（）；

//初始化编译器设置以能够格式化1.5代码

期权.put（javacore.compiler\_compliance，javacore.version\_1\_5）；

选项.put（javacore.compiler\_codegen\_target\_platform，javacore.version\_1\_5）；

options.put（javacore.compiler\_source，javacore.version\_1\_5）；

//更改选项，将每个枚举常量换行

选项

defaultcodeformatterConstants.formatter\_-Alignment\_用于\_-Enum\_常量，

默认代码格式terConstants.CreateAlignmentValue（

真的，

defaultcodeformatterconstants.wrap\_one\_per\_line，

defaultcodeformatterconstants.indent\_on\_column））；

//用给定的选项实例化默认代码格式化程序

final codeformatter codeformatter=toolfactory.createCodeformatter（选项）；

//检索要格式化的源

字符串源=空；

尝试{

source=..；//检索源

}捕获（IOException E）{

system.err.println（“无法检索源”）；/$non-nls-1$

e.printstacktrace（）；

返回；

}

final textedit edit=代码格式化程序.format（

codeformatter.k\_compilation\_unity，//格式化编译单元

source，//要格式化的源

0，//起始位置

source.length（），//长度

0，//初始缩进

system.getproperty（“line.separator”）//行分隔符

；

idocument document=新文档（源）；

尝试{

编辑、应用（文件）；

}catch（异常TreeException E）{

e.printstacktrace（）；

}catch（BadLocationException E）{

e.printstacktrace（）；

}

//在系统上显示格式化字符串

system.out.println（document.get（））；

在这个例子中，



公共枚举x a、b、c、d、e、f

结果是：



公共枚举x{

A

B

C

D

E

f

}

**格式化一组区域**

默认代码格式化程序允许格式化给定源文件的一组区域。这可以通过调用代码格式化程序的格式（int，string，iregion[]，int，string）方法来实现，方法是使用给定的源类型和区域数组。

* 要格式化的每个区域都必须在源范围内，并且不应与另一个区域重叠。
* 要设置格式的区域数组必须至少包含一个区域。区域应按其偏移量排序，首先是较小的偏移量。

**注释格式化程序API**

默认的代码格式化程序API提供了在处理代码段期间格式化注释的可能性。这可以通过将适当的标志f\_include\_comments与k\_compilation\_unity和k\_unknown flags组合来实现。只有在调用format（int，string，int，int，int，string）或format（int，string，iregion[]，int，string）方法时启用了相应的格式化选项，此标志才有效：

* **格式设置工具\_comment\_format\_javadoc\_comment**它控制JavaDoc注释的格式
* **格式设置工具\u注释\u格式\u块\u注释**它控制多行注释的格式
* **格式设置工具\u注释\u格式\u行\u注释**它控制单行注释的格式

**注释格式设置工具选项**

可以使用各种格式选项来设置注释的格式：

* 常规选项启用或禁用特定注释的格式设置（javadoc、多行或单行注释），设置注释的最大行宽。
* JavaDoc注释选项，用于启用JavaDoc注释、缩进标记描述等中代码段或HTML节的格式设置。
* 阻止注释选项以保留或删除此类注释中的空行

有关这些设置的详细信息，请参阅defaultcodeformatterconstants

**使用独立格式化程序格式化注释**

默认代码格式化程序可用于格式化注释（javadoc、多行或单行）。在这种情况下，传递给格式方法的源只应包含特定类型的注释，并应使用相应类型的k\_java\_doc、k\_multi\_line\_comment或k\_single\_line\_comment。

**格式化JavaDoc注释**

以下未格式化的javadoc：



/\*\*

\*这只是一个简单的示例，演示如何格式化JavaDoc注释。

\*@param str输入字符串

\*@返回结果字符串

\*/

格式如下：

/\*\*

\*这只是一个简单的示例，演示如何格式化JavaDoc注释。

\*

\*@ PARAM STR

\*输入字符串

\*@返回结果字符串

\*/

使用命令：



（…）

final textedit edit=代码格式化程序.format（

codeformatter.k\_java\_doc，//指定类型：javadoc

source，//要格式化的源（按照上面的示例）

0，//起始位置

source.length（），//长度

0，//初始缩进

system.getproperty（“line.separator”）//行分隔符

；

（…）

**设置多行批注格式**

以下未格式化的多行注释：



/\*

\*这只是一个多行注释的示例，旨在演示默认的代码格式化程序格式化多行注释的能力。

\*

\*这些可能性包括：javadoc注释的格式化，

\*多行批注格式

\*/

格式如下：

/\*

\*这只是一个多行注释示例，旨在演示

\*设置多行注释格式的默认代码格式化程序功能。

\*

\*这些可能性包括：javadoc注释的格式化、格式化

\*多行注释

\*/

使用命令：

（…）

final textedit edit=代码格式化程序.format（

codeformatter.k\_multi\_line\_comment，//指定类型：multi line comments

source，//要格式化的源（按照上面的示例）

0，//起始位置

source.length（），//长度

0，//初始缩进

system.getproperty（“line.separator”）//行分隔符

；

（…）

**设置单行批注格式**

以下未格式化的单行注释：

//这是一个长注释，如果启用了行注释格式，则应拆分为多行注释。

格式如下：

//这是一个长注释，应在

//如果启用了行注释格式

使用命令：

（…）

final textedit edit=代码格式化程序.format（

codeformatter.k\_single\_line\_comment，//指定类型：single line comments

source，//要格式化的源（按照上面的示例）

0，//起始位置

source.length（），//长度

0，//初始缩进

system.getproperty（“line.separator”）//行分隔符

；

（…）