用于处理 Java 程序的抽象语法树类

CodeQL 有很多类可以用来表示 Java 程序的抽象语法树。
抽象语法树（AST）表示程序的语法结构。AST 上的节点表示语句和表达式等元素。

语句类

此表列出了 Stmt 的所有子类。

Statement syntax	CodeQL class	Superclasses	Remarks
`;`	EmptyStmt		
Expr `;`	ExprStmt		
`{` Stmt `...` `}`	Block		
`if (` Expr `)` Stmt `else` Stmt	IfStmt	ConditionalStmt	
`if (` Expr `)` Stmt			
`while (` Expr `)` Stmt	WhileStmt	ConditionalStmt, LoopStmt	
`do` Stmt `while (` Expr `)`	DoStmt	ConditionalStmt, LoopStmt	
`for (` Expr `;` Expr `;` Expr `)` Stmt	ForStmt	ConditionalStmt, LoopStmt	
`for (` VarAccess `:` Expr `)` Stmt	EnhancedForStmt	LoopStmt	
`switch (` Expr `){` SwitchCase `...}`	SwitchStmt		
`try {` Stmt `... }finally {` Stmt `... }`	TryStmt		
`return` Expr `;`	ReturnStmt		
`return ;`			
`throw` Expr `;`	ThrowStmt		
`break ;`	BreakStmt	JumpStmt	
`break` label `;`			
`continue ;`		JumpStmt	

Statement syntax	CodeQL class	Superclasses	Remarks
`continue label;`	ContinueStmt		
`label : Stmt`	LabeledStmt		
`synchronized (Expr ) Stmt`	SynchronizedStmt		
`assert Expr :Expr ;`	AssertStmt		
`assert Expr ;`			
`TypeAccess name;`	LocalVariableDeclStmt		
`class name {Member ... } ;`	LocalClassDeclStmt		
`this ( Expr ,... ) ;`	ThisConstructorInvocationStmt		
`super ( Expr ,... ) ;`	SuperConstructorInvocationStmt		
`catch (TypeAccess name) { Stmt ... }`	CatchClause		can only occur as child of a `TryStmt`
`case Literal :Stmt ...`	ConstCase		can only occur as child of a `SwitchStmt`
`default : Stmt...`	DefaultCase		can only occur as child

Statement syntax	CodeQL class	Superclasses	Remarks
			of a `SwitchStmt`

有很多表达式类，所以我们按类别来呈现它们。本节中的所有类都是 Expr 的子类。

直接常量

本小节中的所有类都是 Literal 的子类。

Expression syntax example	CodeQL class
`true`	`BooleanLiteral`
`23`	`IntegerLiteral`
`23l`	`LongLiteral`
`4.2f`	`FloatingPointLiteral`
`4.2`	`DoubleLiteral`
`'a'`	`CharacterLiteral`
`"Hello"`	`StringLiteral`
`null`	`NullLiteral`

一元表达式

本小节中的所有类都是 UnaryExpr 的子类。

Expression syntax example	CodeQL class	Superclasses	Remarks
`x++`	`PostIncExpr`	`UnaryAssignExpr`	
`x--`	`PostDecExpr`	`UnaryAssignExpr`	
`++x`	`PreIncExpr`	`UnaryAssignExpr`	
`--x`	`PreDecExpr`	`UnaryAssignExpr`	
`~x`	`BitNotExpr`	`BitwiseExpr`	see below for other subclasses of `BitwiseExpr`
`-x`	`MinusExpr`		
`+x`	`PlusExpr`		

Expression syntax example	CodeQL class	Superclasses	Remarks
`!x`	`LogNotExpr`	`LogicExpr`	see below for other subclasses of `LogicExpr`

本小节中的所有类都是 BinaryExpr 的子类。

Expression syntax example	CodeQL class	Superclasses
`x * y`	`MulExpr`	
`x / y`	`DivExpr`	
`x % y`	`RemExpr`	
`x + y`	`AddExpr`	
`x - y`	`SubExpr`	
`x << y`	`LShiftExpr`	
`x >> y`	`RShiftExpr`	
`x >>> y`	`URShiftExpr`	
`x && y`	`AndLogicalExpr`	`LogicExpr`
`x || y`	`OrLogicalExpr`	`LogicExpr`
`x < y`	`LTExpr`	`ComparisonExpr`
`x > y`	`GTExpr`	`ComparisonExpr`
`x <= y`	`LEExpr`	`ComparisonExpr`
`x >= y`	`GEExpr`	`ComparisonExpr`
`x == y`	`EQExpr`	`EqualityTest`
`x != y`	`NEExpr`	`EqualityTest`
`x & y`	`AndBitwiseExpr`	`BitwiseExpr`
`x | y`	`OrBitwiseExpr`	`BitwiseExpr`
`x ^ y`	`XorBitwiseExpr`	`BitwiseExpr`

赋值表达式

此表中的所有类都是赋值的子类。

Expression syntax example	CodeQL class	Superclasses
`x = y`	`AssignExpr`	
`x += y`	`AssignAddExpr`	`AssignOp`

Expression syntax example	CodeQL class	Superclasses
`x -= y`	`AssignSubExpr`	`AssignOp`
`x *= y`	`AssignMulExpr`	`AssignOp`
`x /= y`	`AssignDivExpr`	`AssignOp`
`x %= y`	`AssignRemExpr`	`AssignOp`
`x &= y`	`AssignAndExpr`	`AssignOp`
`x |= y`	`AssignOrExpr`	`AssignOp`
`x ^= y`	`AssignXorExpr`	`AssignOp`
`x <<= y`	`AssignLShiftExpr`	`AssignOp`
`x >>= y`	`AssignRShiftExpr`	`AssignOp`
`x >>>= y`	`AssignURShiftExpr`	`AssignOp`

访问

Expression syntax examples	CodeQL class
`this`	ThisAccess
`Outer.this`	
`super`	SuperAccess
`Outer.super`	
`x`	VarAccess
`e.f`	
`a[i]`	ArrayAccess
`f(...)`	MethodAccess
`e.m(...)`	
`String`	TypeAccess
`java.lang.String`	
`? extends Number`	WildcardTypeAccess
`? super Double`	

引用字段的 VarAccess 是 FieldAccess。

其他

Expression syntax examples	CodeQL class	Remarks
`(int) f`	CastExpr	
`(23 + 42)`	ParExpr	
`o instanceof String`	InstanceOfExpr	

Expression syntax examples	CodeQL class	Remarks
Expr ? Expr : Expr	ConditionalExpr	
String. class	TypeLiteral	
new A()	ClassInstanceExpr	
new String[3][2]	ArrayCreationExpr	
new int[] { 23, 42}		
{ 23, 42 }	ArrayInit	can only appear as an initializer or as a child of an ArrayCreationExpr
@Annot(key=val)	Annotation	

进一步阅读

- Java 的 CodeQL 查询
- Java 查询示例
- Java 代码库参考
- QL 语言参考
- CodeQL 工具

Java 数据流分析

您可以使用 CodeQL 来跟踪通过 Java 程序使用的数据流。

关于这篇文章

本文描述了如何在 CodeQL 库中实现数据流分析，并提供了一些示例来帮助您编写自己的数据流查询。以下部分描述如何使用库进行本地数据流、全局数据流和污点跟踪。
有关数据流建模的更一般性介绍，请参阅关于数据流分析。

本地数据流

本地数据流是单个方法或可调用的数据流。本地数据流通常比全局数据流更容易、更快、更精确，并且足以满足许多查询。

使用本地数据流

本地数据流库位于模块 DataFlow 中，它定义了表示数据可以通过的任何元素的类节点。节点分为表达式节点（ExprNode）和参数节点（ParameterNode）。可以使用成员谓词 asExpr 和 ASAMETER 在数据流节点和表达式/参数之间进行映射：

```
class Node {

  /** Gets the expression corresponding to this node, if any. */

  Expr asExpr() { ... }



  /** Gets the parameter corresponding to this node, if any. */

  Parameter asParameter() { ... }



  ...

}
```

或者使用谓词 exprNode 和 parameterNode：

```
/**

 * Gets the node corresponding to expression `e`.

 */

ExprNode exprNode(Expr e) { ... }



/**

 * Gets the node corresponding to the value of parameter `p` at
function entry.

 */

ParameterNode parameterNode(Parameter p) { ... }
```

谓词 localFlowStep（Node nodeFrom，Node nodeTo）保持从 nodeFrom 到 nodeTo 的即时数据流边缘。您可以通过使用+和*运算符递归地应用谓词，或者使用预定义的递归谓词 localFlow，它相当于 localFlowStep*。
例如，可以在零个或多个本地步骤中找到从参数源到表达式接收器的流：

```
DataFlow::localFlow(DataFlow::parameterNode(source),
DataFlow::exprNode(sink))
```

使用本地污点跟踪

局部污点跟踪通过包含非值保持流步骤来扩展本地数据流。例如：

```
String temp = x;

String y = temp + ", " + temp;
```

如果 x 是一个受污染的字符串，那么 y 也是受污染的。
本地污点跟踪库位于污点跟踪模块中。与本地数据流一样，谓词 localTaintStep（DataFlow::Node nodeFrom，DataFlow::nodeTo）保持从 nodeFrom 到 nodeTo 的即时污染传播边缘。可以通过使用+和*运算符递归地应用谓词，也可以使用预定义的递归谓词 localTaint，它相当于 localTaintStep*。
例如，可以在零个或多个本地步骤中找到从参数源到表达式接收器的污点传播：

```
TaintTracking::localTaint(DataFlow::parameterNode(source),
DataFlow::exprNode(sink))
```

示例

此查询查找传递给新 FileReader（..）的文件名。

```
import java


from Constructor fileReader, Call call

where

  fileReader.getDeclaringType().hasQualifiedName("java.io",
"FileReader") and

  call.getCallee() = fileReader

select call.getArgument(0)
```

不幸的是，这只给出参数中的表达式，而不是可以传递给它的值。因此，我们使用本地数据流来查找流入参数的所有表达式：

```
import java

import semmle.code.java.dataflow.DataFlow


from Constructor fileReader, Call call, Expr src

where

  fileReader.getDeclaringType().hasQualifiedName("java.io",
"FileReader") and

  call.getCallee() = fileReader and

  DataFlow::localFlow(DataFlow::exprNode(src),
DataFlow::exprNode(call.getArgument(0)))

select src
```

然后我们可以使源更具体，例如访问公共参数。查找传递给新的文件的读取器（

```
import java

import semmle.code.java.dataflow.DataFlow


from Constructor fileReader, Call call, Parameter p

where

  fileReader.getDeclaringType().hasQualifiedName("java.io",
"FileReader") and

  call.getCallee() = fileReader and

  DataFlow::localFlow(DataFlow::parameterNode(p),
DataFlow::exprNode(call.getArgument(0)))

select p
```

调用不包含此格式的字符串的函数。

```
import java

import semmle.code.java.dataflow.DataFlow
```

```
import semmle.code.java.StringFormat


from StringFormatMethod format, MethodAccess call, Expr formatString
where
  call.getMethod() = format and
  call.getArgument(format.getFormatStringIndex()) = formatString and
  not exists(DataFlow::Node source, DataFlow::Node sink |
    DataFlow::localFlow(source, sink) and
    source.asExpr() instanceof StringLiteral and
    sink.asExpr() = formatString
  )
select call, "Argument to String format method isn't hard-coded."
```

练习

练习 1：编写一个查询，查找用于创建 java.net.URL，使用本地数据流。（回答）

全局数据流

全局数据流跟踪整个程序中的数据流，因此比本地数据流更强大。然而，全局数据流不如本地数据流精确，分析通常需要更多的时间和内存来执行。

注意

可以通过创建路径查询在 CodeQL 中对数据流路径进行建模。要查看由 codeqlforvs 代码中的路径查询生成的数据流路径，需要确保它具有正确的元数据和 select 子句。有关详细信息，请参见创建路径查询。

使用全局数据流

您可以通过扩展 DataFlow::Configuration 类来使用全局数据流库：

```
import semmle.code.java.dataflow.DataFlow


class MyDataFlowConfiguration extends DataFlow::Configuration {
  MyDataFlowConfiguration() { this = "MyDataFlowConfiguration" }
```

```
  override predicate isSource(DataFlow::Node source) {
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

这些谓词在配置中定义：

- isSource 定义数据的来源
- ISink 定义数据可能流向的位置
- isBarrier 可选，限制数据流
- isAdditionalFlowStep 可选，添加其他流步骤

特征谓词 MyDataFlowConfiguration（）定义配置的名称，因此“MyDataFlowConfiguration”应该是唯一的名称，例如，类的名称。

使用谓词 hasFlow（DataFlow:：Node source，DataFlow:：Node sink）执行数据流分析：

```
from MyDataFlowConfiguration dataflow, DataFlow::Node source,
DataFlow::Node sink

where dataflow.hasFlow(source, sink)

select source, "Data flow to $@.", sink, sink.toString()
```

使用全局污染跟踪

全局污点跟踪针对全局数据流，正如局部污点跟踪针对本地数据流一样。也就是说，全局污点跟踪通过附加的非保值步骤扩展了全局数据流。您可以通过扩展 TaintTracking:：Configuration 类来使用全局污染跟踪库：

```
import semmle.code.java.dataflow.TaintTracking


class MyTaintTrackingConfiguration extends
TaintTracking::Configuration {
```

```
  MyTaintTrackingConfiguration() { this =
"MyTaintTrackingConfiguration" }


  override predicate isSource(DataFlow::Node source) {
    ...
  }


  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

这些谓词在配置中定义：

- isSource 定义了污点的来源
- isSink 定义了污点可能流向何处
- isSanitizer 可选，限制污染流
- isAdditionalTaintStep 可选，添加其他污染步骤

与全局数据流类似，特征谓词 MyTaintTrackingConfiguration（）定义配置的唯一名称。
污点跟踪分析使用谓词 hasFlow（DataFlow::Node source，DataFlow::Node sink）执行。

流动源

数据流库包含一些预定义的流源。类 RemoteFlowSource（在中定义 semmle.code.java.数据流.FlowSources)表示可能由远程用户控制的数据流源，这对于查找安全问题很有用。

示例

此查询显示使用远程用户输入作为数据源的污点跟踪配置。

```
import java

import semmle.code.java.dataflow.FlowSources


class MyTaintTrackingConfiguration extends
TaintTracking::Configuration {
```

```
  MyTaintTrackingConfiguration() {
    this = "..."
  }

  override predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }

  ...
}
```

练习

练习 2：编写一个查询，查找用于创建 java.net.URL，使用全局数据流。（回答）
练习 3：编写一个表示流源的类 java.lang.System.getenv（…）。（回答）
练习 4：使用 2 和 3 中的答案，编写一个查询，查找从 getenv 到 java.net.URL. （回答）

答案

练习 1

```
import semmle.code.java.dataflow.DataFlow

from Constructor url, Call call, StringLiteral src
where
  url.getDeclaringType().hasQualifiedName("java.net", "URL") and
  call.getCallee() = url and
  DataFlow::localFlow(DataFlow::exprNode(src),
DataFlow::exprNode(call.getArgument(0)))
select src
```

练习 2

```
import semmle.code.java.dataflow.DataFlow


class Configuration extends DataFlow::Configuration {
  Configuration() {
    this = "LiteralToURL Configuration"
  }

  override predicate isSource(DataFlow::Node source) {
    source.asExpr() instanceof StringLiteral
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(Call call |
      sink.asExpr() = call.getArgument(0) and
call.getCallee().(Constructor).getDeclaringType().hasQualifiedName("
java.net", "URL")
    )
  }
}


from DataFlow::Node src, DataFlow::Node sink, Configuration config
where config.hasFlow(src, sink)
select src, "This string constructs a URL $@.", sink, "here"
```

练习 3

```
import java


class GetenvSource extends MethodAccess {
  GetenvSource() {
    exists(Method m | m = this.getMethod() |
      m.hasName("getenv") and
      m.getDeclaringType() instanceof TypeSystem
    )
  }
}
```

练习 4

```
import semmle.code.java.dataflow.DataFlow


class GetenvSource extends DataFlow::ExprNode {
  GetenvSource() {
    exists(Method m | m = this.asExpr().(MethodAccess).getMethod() |
      m.hasName("getenv") and
      m.getDeclaringType() instanceof TypeSystem
    )
  }
}


class GetenvToURLConfiguration extends DataFlow::Configuration {
```

```
  GetenvToURLConfiguration() {
    this = "GetenvToURLConfiguration"
  }

  override predicate isSource(DataFlow::Node source) {
    source instanceof GetenvSource
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(Call call |
      sink.asExpr() = call.getArgument(0) and
call.getCallee().(Constructor).getDeclaringType().hasQualifiedName("
java.net", "URL")
    )
  }
}

from DataFlow::Node src, DataFlow::Node sink,
GetenvToURLConfiguration config
where config.hasFlow(src, sink)
select src, "This environment variable constructs a URL $@.", sink,
"here"
```

进一步阅读

- 用路径查询探索数据流
- Java 的 CodeQL 查询
- Java 查询示例
- Java 代码库参考
- QL 语言参考

Java 中的注释

Java 项目的 CodeQL 数据库包含关于附加到程序元素的所有注释的信息。

关于使用注释

注释由以下 CodeQL 类表示：

- Annotatable 类表示可能附加了注释的所有实体（即包、引用类型、字段、方法和局部变量）。
- AnnotationType 类表示 Java 注释类型，例如 java.lang.Override；注释类型是接口。
- AnnotationElement 类表示注释元素，即注释类型的成员。
- 类注释表示一个注释，如@Override；可以通过成员谓词 getValue 访问注释值。

例如，Java 标准库定义了一个注释 SuppressWarnings，指示编译器不要发出某些类型的警告：

```
package java.lang;


public @interface SuppressWarnings {

    String[] value;

}
```

SuppressWarnings 表示为 AnnotationType，value 是其唯一的 AnnotationElement。
SuppressWarnings 的典型用法是此注释，用于防止有关使用原始类型的警告：

```
class A {

    @SuppressWarnings("rawtypes")

    public A(java.util.List rawlist) {

    }

}
```

表达式@SuppressWarnings（“rawtypes”）表示为注释。字符串文本“rawtypes”用于初始化注释元素值，其值可以通过 getValue 谓词从注释中提取。
然后，我们可以编写此查询来查找附加到构造函数的所有@SuppressWarnings 注释，并返回注释本身及其 value 元素的值：

```
import java


from Constructor c, Annotation ann, AnnotationType anntp
where ann = c.getAnAnnotation() and
    anntp = ann.getType() and
    anntp.hasQualifiedName("java.lang", "SuppressWarnings")
select ann, ann.getValue("value")
```

➤在上的查询控制台中查看完整查询 LGTM.com 网站. 一些 LGTM.com 网站演示项目使用@SuppressWarnings 注释。查看查询返回的 annotation 元素的值，我们可以看到 apache/activemq 项目使用了上面描述的“rawtypes”值。

另一个示例是，此查询查找只有一个注释元素的所有注释类型，该元素具有名称值：

```
import java


from AnnotationType anntp
where forex(AnnotationElement elt |
    elt = anntp.getAnAnnotationElement() |
    elt.getName() = "value"
)
select anntp
```

➤在上的查询控制台中查看完整查询 LGTM.com 网站.

示例：查找缺少的@Override 注释

在 Java 的较新版本中，建议（尽管不是必需的）用@override 注释对重写另一个方法的方法进行注释。这些由编译器检查的注释作为文档，还可以帮助您避免在打算重写的地方意外重载。
例如，考虑以下示例程序：

```
class Super {

    public void m() {}

}


class Sub1 extends Super {

    @Override public void m() {}

}


class Sub2 extends Super {

    public void m() {}

}
```

在这里，Sub1.m 和 Sub2.m 都覆盖 Super.m，但是只有 Sub1.m 被@override 注释。
现在，我们将开发一个查询来查找 Sub2.m 之类的方法，这些方法应该用@Override 注释，但是没有。
作为第一步，让我们编写一个查询来查找所有@Override 注释。注释是表达式，因此可以使用 getType 访问它们的类型。另一方面，注释类型是接口，因此可以使用 hasQualifiedName 查询它们的限定名。因此我们可以这样实现查询：

```
import java


from Annotation ann

where ann.getType().hasQualifiedName("java.lang", "Override")

select ann
```

作为一个 Java 项目，它总是能产生一个好的查询结果。在前面的示例中，它应该在 Sub1.m 上找到注释。接下来，我们将@Override 注释的概念封装为 CodeQL

```
class OverrideAnnotation extends Annotation {

    OverrideAnnotation() {

        this.getType().hasQualifiedName("java.lang", "Override")

    }

}
```

这使得编写查询来查找重写另一个方法的方法变得非常容易，但是没有 @override 注释：我们使用谓词覆盖来确定一个方法是否覆盖另一个方法，并使用谓词 getAnAnnotation（可用于任何注释性表）来检索某些注释。

```
import java


from Method overriding, Method overridden

where overriding.overrides(overridden) and

    not overriding.getAnAnnotation() instanceof OverrideAnnotation

select overriding, "Method overrides another method, but does not
have an @Override annotation."
```

➤请在 LGTM.com 网站. 在实践中，这个查询可能会从编译的库代码中得到许

多结果，这些结果不是很有趣。因此，添加另一个联合词是个好主意重

写.fromSource（）将结果限制为只有源代码可用的报表方法。

示例：查找对不推荐的方法的调用

作为另一个例子，我们可以编写一个查询来查找对用@Deprecated 注释标记的方法的调用。
例如，考虑以下示例程序：

```
class A {

    @Deprecated void m() {}
```

```
    @Deprecated void n() {
        m();
    }

    void r() {
        m();
    }
}
```

在这里，A.m 和 A.n 都标记为已弃用。方法 n 和 r 都调用 m，但请注意，n 本身已被弃用，因此我们可能不应对此调用发出警告。
与上一个示例一样，我们将首先定义一个类来表示@Deprecated 注释：

```
class DeprecatedAnnotation extends Annotation {
    DeprecatedAnnotation() {
        this.getType().hasQualifiedName("java.lang", "Deprecated")
    }
}
```

现在我们可以定义一个类来表示不推荐使用的方法：

```
class DeprecatedMethod extends Method {
    DeprecatedMethod() {
        this.getAnAnnotation() instanceof DeprecatedAnnotation
    }
}
```

最后，我们使用这些类来查找对不推荐方法的调用，不包括本身出现在不推荐方法中的调用：

```
import java
```

```
from Call call
where call.getCallee() instanceof DeprecatedMethod
    and not call.getCaller() instanceof DeprecatedMethod
select call, "This call invokes a deprecated method."
```

在我们的示例中，此查询标记 A.r 中对 A.m 的调用，但不标记 A.n 中的调用。有关类调用的更多信息，请参见导航调用图。

改进

Java 标准库提供了另一种注释类型 java.lang.superssWarnings 可用于抑制某些类别的警告。特别是，它可以用来关闭关于调用不推荐的方法的警告。因此，有必要改进我们的查询，以忽略来自用@SuppressWarnings（“deprecated”）标记的方法内部对已弃用方法的调用。
例如，考虑这个稍微更新的示例：

```
class A {
@Deprecated void m() {}

@Deprecated void n() {
    m();
}

@SuppressWarnings("deprecated")
    void r() {
        m();
    }
}
```

在这里，程序员显式地禁止了关于 A.r 中不推荐使用的调用的警告，因此我们的查询不应该再标记对 A.m 的调用。
为此，我们首先引入一个类来表示所有@SuppressWarnings 注释，其中不推荐使用的字符串出现在要禁止显示的警告列表中：

```
class SuppressDeprecationWarningAnnotation extends Annotation {

    SuppressDeprecationWarningAnnotation() {

        this.getType().hasQualifiedName("java.lang",
"SuppressWarnings") and


this.getAValue().(Literal).getLiteral().regexpMatch(".*deprecation.*
")

    }

}
```

这里，我们使用 getAValue（）来检索任何注释值：事实上，注释类型 SuppressWarnings 只有一个注释元素，因此每个@SuppressWarnings 注释只有一个注释值。然后，我们确保它是一个文本，使用 getLiteral 获取它的字符串值，并使用正则表达式匹配检查它是否包含字符串弃用。
对于真实世界的使用，这个检查必须稍微概括一下：例如，openjdkjava 编译器允许@SuppressWarnings（“all”）注释来抑制所有警告。我们还可能希望通过将正则表达式更改为“.*\\bdeprection\\b.*”，确保 deprecision 作为整个单词匹配，而不是作为另一个单词的一部分匹配。
现在，我们可以扩展查询以筛选出带有 suppressDeprectionWarningAnnotation 的方法中的调用：

```
import java


// Insert the class definitions from above


from Call call

where call.getCallee() instanceof DeprecatedMethod

    and not call.getCaller() instanceof DeprecatedMethod

    and not call.getCaller().getAnAnnotation() instanceof
SuppressDeprecationWarningAnnotation

select call, "This call invokes a deprecated method."
```

➤请在 LGTM.com 网站. 项目包含对似乎已弃用的方法的调用是相当常见的。

Java 代码库

在分析 Java 程序时，可以利用 CodeQL 库中的大量类集合。

关于 Java 的 CodeQL 库

有一个用于分析从 Java 项目中提取的 CodeQL 数据库的扩展库。这个库中的类以面向对象的形式显示数据库中的数据，并提供抽象和谓词来帮助您完成常见的分析任务。

该库实现为一组 QL 模块，即扩展名为.qll 的文件。模块 java.qll 文件导入所有核心 Java 库模块，因此您可以通过以下方式开始查询来包含完整的库：

导入 java

本文的其余部分简要总结了这个库提供的最重要的类和谓词。

注意

本文中的示例查询说明了不同库类返回的结果类型。结果本身并不有趣，但可以作为开发更复杂查询的基础。本节帮助中的其他文章将介绍如何使用一个简单的查询并对其进行微调，以精确地找到您感兴趣的结果。

Summary of the library classes¶

标准 Java 库中最重要的类可以分为五大类：

1. 表示程序元素的类（如类和方法）
2. 表示 AST 节点的类（如语句和表达式）
3. 用于表示元数据的类（如注释和注释）
4. 计算度量的类（例如圈复杂度和耦合）
5. 用于导航程序调用图的类

我们将依次讨论其中的每一个，简要描述每个类别中最重要的类。

程序元素

这些类表示命名的程序元素：包（Package）、编译单元（CompilationUnit）、类型（Type）、方法（Method）、构造函数（Constructor）和变量（Variable）。

它们共同的超类是 Element，它提供了通用成员谓词，用于确定程序元素的名称并检查两个元素是否嵌套在彼此内部。

引用可能是方法或构造函数的元素通常很方便；可调用类（method 和 constructor 的通用超类）可用于此目的。

类型

类类型有许多子类用于表示不同类型的类型：

- PrimitiveType 表示一个基元类型，即 boolean、byte、char、double、float、int、long、short 之一；QL 还将 void 和<nulltype>（空文本的类型）划分为基元类型。

- RefType 表示引用（即非基元）类型；它又有几个子类：
  - 类表示 Java 类。
  - 接口表示 Java 接口。
  - EnumType 表示 Java 枚举类型。
  - 数组表示 Java 数组类型。

例如，以下查询查找程序中所有 int 类型的变量：

```
import java


from Variable v, PrimitiveType pt

where pt = v.getType() and

    pt.hasName("int")

select v
```

➤请在 LGTM.com 网站. 运行此查询时可能会得到许多结果，因为大多数项目

包含许多 int 类型的变量。

引用类型也根据其声明范围进行分类：

- TopLevelType 表示在编译单元的顶层声明的引用类型。
- NestedType 是在另一个类型内声明的类型。

例如，此查询查找与其编译单元的名称不同的所有顶级类型：

```
import java


from TopLevelType tl

where tl.getName() != tl.getCompilationUnit().getName()

select tl
```

➤请在 LGTM.com 网站. 这种模式在许多项目中都可以看到。当我们在

LGTM.com 网站演示项目中，大多数项目的源代码中至少有一个此问题的实

例。源代码引用的文件中还有许多实例。

还有几个更专业的课程：

- TopLevelClass 表示在编译单元的顶层声明的类。

- NestedClass 表示在另一个类型中声明的类，例如：
  - LocalClass，它是在方法或构造函数中声明的类。
  - 一个匿名类，它是一个匿名类。

最后，这个库还有许多封装常用 Java 标准库类的单例类：TypeObject、TypeCloneable、TypeRuntime、TypeSerializable、TypeString、TypeSystem 和 TypeClass。每个 CodeQL 类代表由其名称建议的标准 Java 类。
例如，我们可以编写一个查询来查找所有直接扩展对象的嵌套类：

```
import java


from NestedClass nc

where nc.getASupertype() instanceof TypeObject

select nc
```

➤请在 LGTM.com 网站.运行此查询时可能会得到许多结果，因为许多项目都包含直接扩展对象的嵌套类。

仿制药

还有几个类型的子类用于处理泛型类型。
GenericType 可以是 GenericInterface 或 GenericClass。它表示一个泛型类型声明，例如 interfacejava.util.Map 从 Java 标准库：

```
package java.util.;


public interface Map<K, V> {

    int size();


    // ...

}
```

类型参数（如本例中的 K 和 V）由类 TypeVariable 表示。
泛型类型的参数化实例提供一个具体的类型来实例化类型参数，如 Map<String，File>中所示。泛型与表示它的 ictype 不同。要从 parameteredType 转换到其对应的 GenericType，可以使用谓词 getSourceDeclaration。
例如，我们可以使用以下查询来查找 java.util.Map:

```
import java


from GenericInterface map, ParameterizedType pt

where map.hasQualifiedName("java.util", "Map") and

    pt.getSourceDeclaration() = map

select pt
```

➤请在 LGTM.com 网站. 没有 LGTM.com 网站演示项目包含的参数化实例

java.util.Map 在它们的源代码中，但它们都在引用文件中有结果。

一般来说，泛型类型可能会限制类型参数可以绑定到哪些类型。例如，从字符串到数字的映射类型可以声明如下：

```
class StringToNumMap<N extends Number> implements Map<String, N> {

    // ...

}
```

这意味着 StringToNumberMap 的参数化实例只能用类型号或其子类型之一实例化类型参数 N，但不能实例化 File。我们说 N 是一个有界类型参数，其上界是数字。在 QL 中，可以使用谓词 getATypeBound 查询类型变量的类型绑定。类型边界本身由 TypeBound 类表示，它有一个成员谓词 getType 来检索变量所绑定的类型。
例如，以下查询将查找具有类型绑定数字的所有类型变量：

```
import java


from TypeVariable tv, TypeBound tb

where tb = tv.getATypeBound() and

    tb.getType().hasQualifiedName("java.lang", "Number")

select tv
```

➤请在 LGTM.com 网站. 当我们在 LGTM.com 网站演示项目、neo4j/neo4j、gradle/gradle 和 hibernate/hibernate orm 项目都包含这种模式的示例。

为了处理不知道泛型的遗留代码，每个泛型类型都有一个没有任何类型参数的“原始”版本。在 CodeQL 库中，原始类型使用类 RawType 表示，该类具有预期的子类 RawClass 和 RawInterface。同样，还有一个用于获取相应泛型类型的谓词 getSourceDeclaration。例如，我们可以找到（原始）类型映射的变量：

```
import java


from Variable v, RawType rt
where rt = v.getType() and
    rt.getSourceDeclaration().hasQualifiedName("java.util", "Map")
select v
```

➤请在 LGTM.com 网站. 许多项目都有原始类型的 Map 变量。

例如，在下面的代码片段中，此查询将找到 m1，而不是 m2：

```
Map m1 = new HashMap();

Map<String, String> m2 = new HashMap<String, String>();
```

最后，变量可以声明为通配符类型：

```
Map<? extends Number, ? super Float> m;
```

通配符？扩展数字和？super Float 由类通配符 typeaccess 表示。与类型参数一样，通配符也可能有类型边界。与类型参数不同，通配符可以有上界（如？扩展数字），以及下限（如？超级浮动）。类通配符 typeaccess 提供成员谓词 getUpperBound 和 getLowerBound 分别检索上下界。
对于泛型方法，有 GenericMethod、ParameterizedMethod 和 RawMethod 类，它们完全类似于表示泛型类型的类似命名类。
有关使用类型的更多信息，请参阅关于 Java 类型的文章。

变量

Class Variable 代表 Java 意义上的变量，它可以是类的成员字段（无论是静态的还是非静态的）、局部变量或参数。因此，有三个子类满足这些特殊情况：

- 字段表示 Java 字段。
- LocalVariableDecl 表示局部变量。
- 参数表示方法或构造函数的参数。

抽象语法树

此类别中的类表示抽象语法树（AST）节点，即语句（类 Stmt）和表达式（类 Expr）。有关标准 QL 库中可用的表达式和语句类型的完整列表，请参阅使用 Java 程序的抽象语法树类。
Expr 和 Stmt 都为探索程序的抽象语法树提供了成员谓词：

- Expr.getAChildExpr 返回给定表达式的子表达式。
- Stmt.getAChild 公司返回直接嵌套在给定语句中的语句或表达式。
- Expr.getParent 以及 Stmt.getParent 返回 AST 节点的父节点。

例如，以下查询查找其父级为 return 语句的所有表达式：

```
import java


from Expr e

where e.getParent() instanceof ReturnStmt

select e
```

➤请在 LGTM.com 网站.许多项目都有带有子语句的返回语句的示例。

因此，如果程序包含返回语句 return x+y; ，则此查询将返回 x+y。
作为另一个示例，以下查询查找其父级为 if 语句的语句：

```
import java


from Stmt s

where s.getParent() instanceof IfStmt

select s
```

➤请在 LGTM.com 网站.许多项目都有带有子语句的 if 语句的示例。

此查询将在程序中找到所有 if 语句的 then 分支和 else 分支。
最后，下面是一个查找方法体的查询：

```
import java
```

```
from Stmt s

where s.getParent() instanceof Method

select s
```

➤请在 LGTM.com 网站. 大多数项目都有许多方法体。

正如这些示例所示，表达式的父节点并不总是表达式：它也可能是语句，例如 IfStmt。类似地，语句的父节点并不总是语句：它也可以是方法或构造函数。为了捕捉这一点，qljava 库提供了两个抽象类 ExprParent 和 StmtParent，前者表示可能是表达式父节点的任何节点，后者表示可能是语句父节点的任何节点。
有关如何使用容易溢出的类的更多信息，请参阅文章中的“使用易于溢出的类进行比较”。

元数据

除了程序代码之外，Java 程序还有几种元数据。尤其是注释和 Javadoc 注释。由于此元数据对于增强代码分析和本身作为一个分析主题都很有趣，所以 QL 库定义了用于访问它的类。
对于注释，Annotatable 类是所有可以注释的程序元素的超类。这包括包、引用类型、字段、方法、构造函数和局部变量声明。对于每个这样的元素，其谓词 getAnAnnotation 允许您检索元素可能具有的任何注释。例如，以下查询查找构造函数上的所有批注：

```
import java



from Constructor c

select c.getAnAnnotation()
```

➤请在 LGTM.com 网站. 这个 LGTM.com 网站演示项目都使用注释，您可以看

到它们用于抑制警告和将代码标记为不推荐使用的示例。

这些注释由类注释表示。注释只是一个类型为 AnnotationType 的表达式。例如，可以修改此查询，使其只报告不推荐的构造函数：

```
import java
```

```
from Constructor c, Annotation ann, AnnotationType anntp
where ann = c.getAnAnnotation() and
    anntp = ann.getType() and
    anntp.hasQualifiedName("java.lang", "Deprecated")
select ann
```

➤请在 LGTM.com 网站. 这次只报告带有@deprecated 注释的构造函数。

有关使用注释的更多信息，请参阅使用注释的更多信息。
对于 Javadoc，class 元素有一个成员谓词 getDoc，该谓词返回一个委托 Documentable 对象，然后可以查询它附加的 Javadoc 注释。例如，以下查询在私有字段上查找 Javadoc 注释：

```
import java


from Field f, Javadoc jdoc
where f.isPrivate() and
    jdoc = f.getDoc().getJavadoc()
select jdoc
```

➤请在 LGTM.com 网站. 您可以在许多项目中看到这种模式。

类 Javadoc 将整个 Javadoc 注释表示为 JavadocElement 节点的树，可以使用成员谓词 getAChild 和 getParent 遍历这些节点。例如，您可以编辑查询，以便在私有字段的 Javadoc 注释中找到所有@author 标记：

```
import java


from Field f, Javadoc jdoc, AuthorTag at
where f.isPrivate() and
    jdoc = f.getDoc().getJavadoc() and
    at.getParent+() = jdoc
select at
```

➤请在 LGTM.com 网站. 没有 LGTM.com 网站演示项目在私有字段上使用

@author 标记。

注意
在第 5 行中，我们使用 getParent+来捕获嵌套在 Javadoc 注释中任何深度的标记。
有关使用 Javadoc 的更多信息，请参阅关于 Javadoc 的文章。

韵律学

标准 qljava 库为计算 Java 程序元素上的度量提供了广泛的支持。为了避免用太多与度量计算相关的成员谓词来表示那些元素的类负担过重，可以在委托类上使用这些谓词。
总共有六个这样的类：MetricElement、MetricPackage、MetricRefType、MetricField、MetricCallable 和 MetricStmt。相应的元素类各自提供一个成员谓词 getMetrics，该谓词可用于获取委托类的实例，然后可以在该实例上执行度量计算。
例如，以下查询查找圈复杂度大于 40 的方法：

```
import java



from Method m, MetricCallable mc

where mc = m.getMetrics() and

    mc.getCyclomaticComplexity() > 40

select m
```

➤请在 LGTM.com 网站. 大多数大型项目包括一些圈复杂度非常高的方法。这些方法可能很难理解和测试。

调用图

从 Java 代码库生成的 CodeQL 数据库包括有关程序调用图的预计算信息，即给定调用在运行时可以向哪些方法或构造函数分派。
上面介绍的类 Callable 包括方法和构造函数。调用表达式是使用类调用抽象的，类调用包括方法调用、新表达式和使用 this 或 super 的显式构造函数调用。
我们可以用谓语调用. getCallee 找出特定调用表达式引用的方法或构造函数。
例如，以下查询查找对 println 方法的所有调用：

```
import java


from Call c, Method m
where m = c.getCallee() and
    m.hasName("println")
select c
```

➤请在 LGTM.com 网站. 这个 LGTM.com 网站演示项目都包括对这个名称的方法的许多调用。

相反，Callable.getAReference 返回引用它的调用。因此，我们可以找到从未使用此查询调用的方法和构造函数：

```
import java


from Callable c
where not exists(c.getAReference())
select c
```

➤请在 LGTM.com 网站. 这个 LGTM.com 网站演示项目似乎都有许多不直接调用的方法，但这不可能是全部。要进一步研究此区域，请参阅导航调用图。

有关可调用项和调用的更多信息，请参阅调用图上的文章。

Java 类型

可以使用 CodeQL 来查找有关 Java 代码中使用的数据类型的信息。这允许您编写查询来识别与类型相关的特定问题。

关于使用 Java 类型

标准 CodeQL 库通过类型类及其各种子类来表示 Java 类型。
特别是，类 PrimitiveType 表示构建在 Java 语言中的基元类型（例如 boolean 和 int），而 RefType 及其子类表示引用类型，即类、接口、数组类型等等。
这包括 Java 标准库中的两种类型（如 java.lang.Object）以及由非库代码定义的类型。

类 RefType 还为类层次结构建模：成员谓词 getASupertype 和 getASubtype 允许您查找引用类型的直接超类型和子类型。例如，考虑以下 Java 程序：

```
class A {}


interface I {}


class B extends A implements I {}
```

在这里，A 类正好有一个直接超类型(java.lang.Object)而且只有一个立即子类型（B）；接口 I 也是如此。另一方面，类 B 有两个立即超类型（A 和 I），没有立即子类型。
为了确定祖先类型（包括直接超类型，以及它们的超类型等），我们可以使用传递闭包。例如，要在上面的示例中查找 B 的所有祖先，可以使用以下查询：

```
import java


from Class B

where B.hasName("B")

select B.getASupertype+()
```

➤请在 LGTM.com 网站. 如果在上面的示例片段上运行此查询，则查询将返回

A、I 和 java.lang.Object.

小费
如果要查看 B 和 A 的位置，可以将 B.getASupertype+（）替换为 B.getASupertype*（），然后重新运行查询。
除了类层次结构建模之外，RefType 还提供成员谓词 getAMember 以访问在类型中声明的成员（即字段、构造函数和方法），谓词继承（方法 m）用于检查类型是否声明或继承方法 m。

示例：查找有问题的数组转换

作为如何使用类层次结构 API 的一个示例，我们可以编写一个查询来查找数组上的向下转换，也就是说，在某种类型 a[]的表达式 e 转换为类型 B[]的情况下，B 是 a 的（不一定是立即的）子类型。
这种类型的转换是有问题的，因为向下转换数组会导致运行时异常，即使每个单独的数组元素都可以被下推。例如，以下代码引发 ClassCastException：

```
Object[] o = new Object[] { "Hello", "world" };

String[] s = (String[])o;
```

另一方面，如果表达式 e 碰巧实际计算为 B[]数组，则转换将成功：

```
Object[] o = new String[] { "Hello", "world" };

String[] s = (String[])o;
```

在本教程中，我们不尝试区分这两种情况。我们的查询只需查找从某个类型源强制转换到另一个类型目标的转换表达式 ce，例如：

- 源和目标都是数组类型。
- source 的元素类型是 target 元素类型的可传递超类型。

这个配方不难翻译成一个查询：

```
import java



from CastExpr ce, Array source, Array target

where source = ce.getExpr().getType() and

    target = ce.getType() and

    target.getElementType().(RefType).getASupertype+() =
source.getElementType()

select ce, "Potentially problematic array downcast."
```

➤请在 LGTM.com 网站. 许多项目返回此查询的结果。

注意，通过铸造 target.getElementType（）对于 RefType，我们消除了元素类型是基元类型的所有情况，也就是说，目标是基元类型的数组：在这种情况下，我们要寻找的问题不会出现。与 Java 不同，QL 中的强制转换永远不会失败：如果一个表达式不能转换为所需的类型，它将被简单地从查询结果中排除，这正是我们想要的。

改进

在版本 5 之前的旧 Java 代码上运行此查询，通常会返回由于使用该方法而产生的许多误报结果集合.toArray（T[]），它将集合转换为 T[]类型的数组。
在不使用泛型的代码中，此方法通常按以下方式使用：

```
List l = new ArrayList();
```

```
// add some elements of type A to l

A[] as = (A[])l.toArray(new A[0]);
```

这里，l 有原始类型列表，因此 l.toArray 有返回类型 Object[]，独立于其参数数组的类型。因此，转换从 Object[]变成了一个[]，并将被我们的查询标记为有问题，尽管在运行时这个转换永远不会出错。
为了识别这些情况，我们可以创建两个 CodeQL 类，分别表示集合.toArray 类，并调用此方法或重写它的任何方法：

```
/** class representing java.util.Collection.toArray(T[]) */

class CollectionToArray extends Method {

    CollectionToArray() {

        this.getDeclaringType().hasQualifiedName("java.util",
"Collection") and

        this.hasName("toArray") and

        this.getNumberOfParameters() = 1

    }

}


/** class representing calls to java.util.Collection.toArray(T[]) */

class CollectionToArrayCall extends MethodAccess {

    CollectionToArrayCall() {

        exists(CollectionToArray m |


this.getMethod().getSourceDeclaration().overridesOrInstantiates*(m)

        )

    }


    /** the call's actual return type, as determined from its
argument */
```

```
    Array getActualReturnType() {

        result = this.getArgument(0).getType()

    }

}
```

注意在 CollectionToArrayCall 的构造函数中使用了 getSourceDeclaration 和 overridesOrInstantiates:我们希望找到集合.toArray 以及重写它的任何方法，以及这些方法的任何参数化实例。例如，在上面的示例中，调用 l.toArray 解析为原始类 ArrayList 中的方法 toArray。它的源声明是泛型类 ArrayList<T>中的 toArray，它重写 AbstractCollection<T>.toArray，后者又重写 Collection<T>.toArray，它是集合.toArray（因为重写方法中的类型参数 T 属于 ArrayList，是属于集合的类型参数的实例化）。
使用这些新类，我们可以扩展查询以排除对类型 A[]的参数的 toArray 调用，然后将其转换为[]：

```
import java


// Insert the class definitions from above


from CastExpr ce, Array source, Array target

where source = ce.getExpr().getType() and

    target = ce.getType() and

    target.getElementType().(RefType).getASupertype+() =
source.getElementType() and

    not ce.getExpr().(CollectionToArrayCall).getActualReturnType() =
target

select ce, "Potentially problematic array downcast."
```

➤请在 LGTM.com 网站.请注意，通过这个改进的查询可以找到更少的结果。

示例：查找不匹配的包含检查

现在，我们将开发一个查询来查找集合.包含其中被查询元素的类型与集合的元素类型无关，这保证测试始终返回 false。

例如，Apache Zookeeper 曾经在 QuorumPeerConfig 类中有一段类似于以下内容的代码片段：

```
Map<Object, Object> zkProp;


// ...


if (zkProp.entrySet().contains("dynamicConfigFile")){
    // ...
}
```

因为 zkProp 是从一个对象到另一个对象的映射，zkProp.entrySet 公司返回 Set<Entry<Object，Object>>类型的集合。这样的集合不能包含 String 类型的元素。（该代码后来被固定使用 zkProp.containsKey 公司.）
一般来说，我们希望找到集合.包含（或其在集合的任何参数化实例中的任何重写方法），使得集合元素的类型 E 和要包含的参数的类型 A 不相关，即它们没有公共子类型。
我们首先创建一个类来描述 java.util.Collection:

```
class JavaUtilCollection extends GenericInterface {
    JavaUtilCollection() {
        this.hasQualifiedName("java.util", "Collection")
    }
}
```

为了确保没有输入错误，我们可以运行一个简单的测试查询：

```
from JavaUtilCollection juc
select juc
```

这个查询应该只返回一个结果。
接下来，我们可以创建一个类来描述 java.util.Collection。包含：

```
class JavaUtilCollectionContains extends Method {
    JavaUtilCollectionContains() {
```

```
        this.getDeclaringType() instanceof JavaUtilCollection and

        this.hasStringSignature("contains(Object)")

    }

}
```

请注意，我们使用 hasStringSignature 来检查：

- 有问题的方法的名称包含。
- 它只有一个论点。
- 参数的类型是 Object。

或者，我们可以使用 TypeObject 的 hasName、getNumberOfParameters 和 getParameter（0）.getType（） instanceof TypeObject 更详细地实现这三个检查。

如前所述，通过运行一个简单的查询来选择 JavaUtilCollectionContains 的所有实例来测试新类是一个好主意；同样应该只有一个结果。

现在我们要确定所有呼叫集合.包含，包括重写它的任何方法，并考虑集合及其子类的所有参数化实例。也就是说，我们正在寻找被调用方法的源声明（反射性或传递性）重写的方法访问集合.包含. 我们在 CodeQL 类 JavaUtilCollectionContainsCall 中对其进行编码：

```
class JavaUtilCollectionContainsCall extends MethodAccess {

    JavaUtilCollectionContainsCall() {

        exists(JavaUtilCollectionContains jucc |

            this.getMethod().getSourceDeclaration().overrides*(jucc)

        )

    }

}
```

这个定义有点微妙，所以您应该运行一个简短的查询来测试 JavaUtilCollectionContainsCall 是否正确地识别了对集合.包含.

对于对 contains 的每次调用，我们对两件事感兴趣：参数的类型和调用它的集合的元素类型。因此，我们需要将两个成员谓词 getArgumentType 和 getCollectionElementType 添加到类 JavaUtilCollectionContainsCall 中来计算这些信息。

前者很简单：

```
Type getArgumentType() {

    result = this.getArgument(0).getType()
```

```
}
```

对于后者，我们的工作如下：

- 查找正在调用的 contains 方法的声明类型 D。
- 找到 D 的（自反或传递）超类型 S，它是的参数化实例 java.util.Collection.
- 返回 S 的（唯一）类型参数。

我们将其编码如下：

```
Type getCollectionElementType() {

    exists(RefType D, ParameterizedInterface S |

        D = this.getMethod().getDeclaringType() and

        D.hasSupertype*(S) and S.getSourceDeclaration() instanceof
JavaUtilCollection and

        result = S.getTypeArgument(0)

    )

}
```

在将这两个成员谓词添加到 JavaUtilCollectionContainsCall 之后，我们需要编写一个谓词来检查两个给定的引用类型是否具有公共子类型：

```
predicate haveCommonDescendant(RefType tp1, RefType tp2) {

    exists(RefType commondesc | commondesc.hasSupertype*(tp1) and
commondesc.hasSupertype*(tp2))

}
```

现在我们准备编写查询的第一个版本：

```
import java


// Insert the class definitions from above


from JavaUtilCollectionContainsCall juccc, Type collEltType, Type
argType
```

```
where collEltType = juccc.getCollectionElementType() and argType =
juccc.getArgumentType() and

    not haveCommonDescendant(collEltType, argType)

select juccc, "Element type " + collEltType + " is incompatible with
argument type " + argType
```

➤请在 LGTM.com 网站.

改进

对于许多程序，由于类型变量和通配符，此查询会产生大量误报结果：例如，如果集合元素类型是某个类型变量 E，而参数类型是 String，则 CodeQL 将认为这两个元素没有公共子类型，我们的查询将标记调用。排除这种误报结果的一个简单方法是简单地要求 collEltType 和 argType 都不是 TypeVariable 的实例。
另一个误报的来源是原语类型的自动装箱：例如，如果集合的元素类型是 Integer 而参数是 int 类型，则谓词 haveCommonDescendant 将失败，因为 int 不是 RefType。为了说明这一点，我们的查询应该检查 collEltType 不是 argType 的装箱类型。
最后，null 是特殊的，因为它的类型（在 CodeQL 库中称为<nulltype>）与每个引用类型都兼容，因此我们应该将其排除在考虑范围之外。
加上这三个改进，我们的最后一个问题是：

```
import java


// Insert the class definitions from above


from JavaUtilCollectionContainsCall juccc, Type collEltType, Type
argType

where collEltType = juccc.getCollectionElementType() and argType =
juccc.getArgumentType() and

    not haveCommonDescendant(collEltType, argType) and

    not collEltType instanceof TypeVariable and not argType
instanceof TypeVariable and

    not collEltType = argType.(PrimitiveType).getBoxedType() and
```

```
    not argType.hasName("<nulltype>")

select juccc, "Element type " + collEltType + " is incompatible with
argument type " + argType
```

➤在上的查询控制台中查看完整查询 LGTM.com 网站.

爪哇文档

可以使用 CodeQL 在 Java 代码中查找 Javadoc 注释中的错误。

关于分析 Javadoc

为了访问与程序元素相关联的 Javadoc，我们使用 class 元素的成员谓词 getDoc，它返回一个 Documentable。类 Documentable 反过来提供一个成员谓词 getJavadoc 来检索附加到相关元素的 Javadoc（如果有的话）。
Javadoc 注释由 Javadoc 类表示，该类将注释作为 JavadocElement 节点的树提供视图。每个 JavadocElement 要么是表示标记的 JavadocTag，要么是表示一段自由格式文本的 javadocext。
Javadoc 类最重要的成员谓词是：

- getAChild-检索树表示中的顶级 JavadocElement 节点。
- getVersion-返回@version 标记的值（如果有）。
- getAuthor-返回@author 标记的值（如果有）。

例如，以下查询查找同时具有@author 标记和@version 标记的所有类，并返回以下信息：

```
import java



from Class c, Javadoc jdoc, string author, string version

where jdoc = c.getDoc().getJavadoc() and

    author = jdoc.getAuthor() and

    version = jdoc.getVersion()

select c, author, version
```

JavadocElement 定义了成员谓词 getAChild 和 getParent，以在元素树中上下导航。它还提供一个谓词 getTagName 来返回标记的名称，并提供一个谓词 getText 来访问与标记相关联的文本。
我们可以重写上面的查询以使用此 API 而不是 getAuthor 和 getVersion：

```
import java
```

```
from Class c, Javadoc jdoc, JavadocTag authorTag, JavadocTag
versionTag

where jdoc = c.getDoc().getJavadoc() and

    authorTag.getTagName() = "@author" and authorTag.getParent() =
jdoc and

    versionTag.getTagName() = "@version" and versionTag.getParent()
= jdoc

select c, authorTag.getText(), versionTag.getText()
```

JavadocTag 有几个代表特定类型 Javadoc 标记的子类：

- ParamTag 表示@param tags；成员谓词 getParamName 返回正在记录的参数的名称。
- ThrowsTag 表示@throws 标记；成员谓词 getExceptionName 返回正在记录的异常的名称。
- AuthorTag 表示@author 标记；成员谓词 getAuthorName 返回作者的名称。

示例：查找虚假的@param 标记

作为使用 codeqljavadocapi 的一个示例，让我们编写一个查询来查找引用不存在的参数的@param 标记。
例如，考虑以下程序：

```
class A {

    /**

    * @param lst a list of strings

    */

    public String get(List<String> list) {

        return list.get(0);

    }

}
```

在这里，A.get 上的@param 标记将参数列表的名称拼错为 lst。我们的查询应该能够找到这种情况。

首先，我们编写一个查询来查找所有可调用项（即方法或构造函数）及其@param 标记：

```
import java


from Callable c, ParamTag pt

where c.getDoc().getJavadoc() = pt.getParent()

select c, pt
```

现在可以很容易地向 where 子句添加另一个联合词，将查询限制为引用不存在的参数的@param 标记：我们只需要要求 c 的任何参数都没有名称 pt.getParamName().

```
import java


from Callable c, ParamTag pt

where c.getDoc().getJavadoc() = pt.getParent() and

    not c.getAParameter().hasName(pt.getParamName())

select pt, "Spurious @param tag."
```

示例：查找虚假的@throws 标记

一个相关的，但更复杂的问题是找到@throws 标记，这些标记引用了所讨论的方法实际上无法引发的异常。
例如，考虑以下 Java 程序：

```
import java.io.IOException;


class A {

    /**

    * @throws IOException thrown if some IO operation fails

    * @throws RuntimeException thrown if something else goes wrong

    */
```

```
    public void foo() {

        // ...

    }

}
```

注意，A.foo 的 Javadoc 注释记录了两个抛出的异常：IOException 和 RuntimeException。前者显然是假的：A.foo 没有 throws-IOException 子句，因此不能抛出这种异常。另一方面，RuntimeException 是一个未检查的异常，因此即使没有显式的 throws 子句列出它，也可以抛出它。所以我们的查询应该标记 IOException 的@throws 标记，而不是 RuntimeException 的标记。
请记住，CodeQL 库使用 ThrowsTag 类表示@throws 标记。这个类没有提供一个成员谓词来确定正在记录的异常类型，因此我们首先需要实现自己的版本。简单的版本可能如下所示：

```
RefType getDocumentedException(ThrowsTag tt) {

    result.hasName(tt.getExceptionName())

}
```

类似地，Callable 也没有提供成员谓词来查询方法或构造函数可能抛出的所有异常。但是，我们可以通过使用 getAnException 来查找可调用的所有 throws 子句，然后使用 getType 来解析相应的异常类型：

```
predicate mayThrow(Callable c, RefType exn) {

    exn.getASupertype*() = c.getAnException().getType()

}
```

注意使用 getASupertype*查找 throws 子句中声明的异常及其子类型。例如，如果一个方法有一个 throws IOException 子句，它可能抛出 malformeduralException，这是 IOException 的一个子类型。
现在我们可以编写一个查询来查找所有可调用的 c 和@throws 标记 tt，以便：

- tt 属于附加到 c 的 Javadoc 注释。
- c 不能抛出由 tt 记录的异常。

```
import java
```

```
// Insert the definitions from above


from Callable c, ThrowsTag tt, RefType exn
where c.getDoc().getJavadoc() = tt.getParent+() and
    exn = getDocumentedException(tt) and
    not mayThrow(c, exn)
select tt, "Spurious @throws tag."
```

➤请在 LGTM.com 网站. 这在 LGTM.com 网站演示项目。

改进

当前，此查询存在两个问题：

1. getDocumentException 太自由了：它将返回具有正确名称的任何引用类型，即使它位于不同的包中，并且在当前编译单元中实际上不可见。
2. mayThrow 限制性太强：它不考虑不需要声明的未检查异常。

要了解为什么前者是个问题，请考虑以下程序：

```
class IOException extends Exception {}


class B {
    /** @throws IOException an IO exception */
    void bar() throws IOException {}
}
```

这个程序定义了自己的类 IOException，它与类无关 java.io.IOException 异常在标准库中：它们在不同的包中。但是，我们的 getDocumentedException 谓词不检查包，因此它将考虑@throws 子句引用两个 IOException 类，并因此将@param 标记标记标记为伪，因为 B.bar 实际上不能抛出 java.io.IOException 异常.
作为第二个问题的一个例子，我们前一个例子中的方法 A.foo 用@throws RuntimeException 标记进行了注释。然而，我们当前版本的 mayThrow 会认为 A.foo 不能抛出 RuntimeException，因此将标记标记标记为虚假的。
我们可以通过引入一个新的类来表示未经检查的异常，这只是 java.lang.RuntimeException 以及 java.lang.Error:

```
class UncheckedException extends RefType {

    UncheckedException() {

        this.getASupertype*().hasQualifiedName("java.lang",
"RuntimeException") or

        this.getASupertype*().hasQualifiedName("java.lang", "Error")

    }

}
```

现在我们将这个新类合并到 mayThrow 谓词中：

```
predicate mayThrow(Callable c, RefType exn) {

    exn instanceof UncheckedException or

    exn.getASupertype*() = c.getAnException().getType()

}
```

修复 getDocumentedException 更为复杂，但我们可以轻松涵盖三种常见情况：

1. @throws 标记指定异常的完全限定名。
2. @throws 标记引用同一个包中的一个类型。
3. @throws 标记引用由当前编译单元导入的类型。

第一种情况可以通过更改 getDocumentedException 来使用@throws 标记的限定名来解决。为了处理第二种和第三种情况，我们可以引入一个新的谓词 visibleIn 来检查引用类型是否在编译单元中可见，无论是由于属于同一个包还是显式导入。然后将 getDocumentedException 重写为：

```
predicate visibleIn(CompilationUnit cu, RefType tp) {

    cu.getPackage() = tp.getPackage()

    or

    exists(ImportType it | it.getCompilationUnit() = cu |
it.getImportedType() = tp)

}



RefType getDocumentedException(ThrowsTag tt) {

    result.getQualifiedName() = tt.getExceptionName()
```

```
    or

    (result.hasName(tt.getExceptionName()) and
visibleIn(tt.getFile(), result))

}
```

➤请在 LGTM.com 网站. 这在 LGTM.com 网站演示项目。

目前，visibleIn 只考虑单一类型的导入，但是您可以扩展它以支持其他类型的导入。

浏览调用图

CodeQL 有用于标识调用其他代码的代码和可以从其他地方调用的代码的类。例如，这允许您找到从未使用过的方法。

调用图类

CodeQL Java 库提供了两个抽象类来表示程序的调用图：Callable 和 call。前者只是 Method 和 Constructor 的公共超类，后者是 MethodAccess、ClassInstanceExpression、ThisConstructorInvocationStmt 和超结构 InvocationsTM 的通用超类。简单地说，可调用是可以调用的，而调用是调用可调用的。
例如，在以下程序中，所有可调用项和调用都已用注释进行了注释：

```
class Super {

    int x;


    // callable

    public Super() {

        this(23);       // call

    }


    // callable

    public Super(int x) {

        this.x = x;

    }
```

```
    // callable
    public int getX() {
        return x;
    }
}

    class Sub extends Super {
    // callable
    public Sub(int x) {
        super(x+19);    // call
    }

    // callable
    public int getX() {
        return x-19;
    }
}

class Client {
    // callable
    public static void main(String[] args) {
        Super s = new Sub(42);  // call
        s.getX();               // call
    }
}
```

类调用提供两个调用图导航谓词：

- getCallee 返回此调用（静态）解析到的可调用；请注意，对于对实例（即非静态）方法的调用，在运行时调用的实际方法可能是重写此方法的其他方法。
- getCaller 返回此调用在语法上是其一部分的可调用项。

例如，在我们的示例中，第二个调用的 getCallee 客户端.main 会回来的超级.getX. 不过，在运行时，这个调用实际上会调用 Sub.getX 公司.
类 Callable 定义了大量成员谓词；就我们的目的而言，两个最重要的谓词是：

- 如果此 Callable 包含一个被调用方是 target 的调用，则 calls（Callable target）将成功。
- 如果 polyCalls（Callable target）可以在运行时调用 target，则 polyCalls（Callable target）将成功；如果它包含的调用的被调用方是 target 或 target 重写的方法，则是这种情况。

在我们的例子中，客户端.main 调用构造函数 Sub（int）和方法超级.getX；此外，它还采用了聚合方法 Sub.getX 公司.

示例：查找未使用的方法

我们可以使用 Callable 类编写一个查询，该查询查找未被任何其他方法调用的方法：

```
import java



from Callable callee

where not exists(Callable caller | caller.polyCalls(callee))

select callee
```

➤请在 LGTM.com 网站. 这个简单的查询通常返回大量结果。

注意
我们必须在这里使用 polycall 而不是调用：我们希望合理地确保被调用方没有被调用，无论是直接调用还是通过重写。
在一个典型的 Java 项目上运行这个查询会导致 Java 标准库中的大量命中。这是有意义的，因为没有一个客户端程序使用标准库的所有方法。更一般地说，我们可能希望从编译的库中排除方法和构造函数。我们可以使用 fromSource 谓词来检查编译单元是否是源文件，并优化查询：

```
import java



from Callable callee
```

```
where not exists(Callable caller | caller.polyCalls(callee)) and

    callee.getCompilationUnit().fromSource()

select callee, "Not called."
```

➤请在 LGTM.com 网站. 此更改减少了大多数项目返回的结果数。

我们还可能注意到一些未使用的方法，它们的名称有些奇怪：它们是类初始值设定项；虽然它们不是在代码中的任何地方显式调用的，但只要周围的类被加载，它们就会被隐式调用。因此，从我们的查询中排除它们是有意义的。在进行此操作时，我们还可以排除终结器，它们同样被隐式调用：

```
import java


from Callable callee

where not exists(Callable caller | caller.polyCalls(callee)) and

    callee.getCompilationUnit().fromSource() and

    not callee.hasName("<clinit>") and not
callee.hasName("finalize")

select callee, "Not called."
```

➤请在 LGTM.com 网站. 这也减少了大多数项目返回的结果数量。

我们还可能希望从查询中排除公共方法，因为它们可能是外部 API 入口点：

```
import java


from Callable callee

where not exists(Callable caller | caller.polyCalls(callee)) and

    callee.getCompilationUnit().fromSource() and

    not callee.hasName("<clinit>") and not
callee.hasName("finalize") and

    not callee.isPublic()

select callee, "Not called."
```

➤请在 LGTM.com 网站. 这对返回的结果数量应该有更明显的影响。

另一个特殊情况是非公共的默认构造函数：例如，在 singleton 模式中，为类提供私有的空默认构造函数，以防止它被实例化。由于此类构造函数的目的是不被调用，因此不应标记它们：

```
import java


from Callable callee

where not exists(Callable caller | caller.polyCalls(callee)) and

    callee.getCompilationUnit().fromSource() and

    not callee.hasName("<clinit>") and not
callee.hasName("finalize") and

    not callee.isPublic() and

    not callee.(Constructor).getNumberOfParameters() = 0

select callee, "Not called."
```

➤请在 LGTM.com 网站. 这种变化对某些项目的结果有很大影响，但对其他项

目的结果影响不大。这种模式的使用在不同的项目中差别很大。

最后，在许多 Java 项目中，有一些方法是通过反射间接调用的。因此，虽然没有调用调用这些方法的调用，但它们实际上是被使用的。一般来说，很难确定这种方法。然而，一个非常常见的特殊情况是 JUnit 测试方法，它由测试运行程序反射性地调用。QL-Java 库支持识别 JUnit 和其他测试框架的测试类，我们可以用它来过滤这些类中定义的方法：

```
import java


from Callable callee

where not exists(Callable caller | caller.polyCalls(callee)) and

    callee.getCompilationUnit().fromSource() and

    not callee.hasName("<clinit>") and not
callee.hasName("finalize") and
```

```
    not callee.isPublic() and

    not callee.(Constructor).getNumberOfParameters() = 0 and

    not callee.getDeclaringType() instanceof TestClass

select callee, "Not called."
```

➤请在 LGTM.com 网站. 这将进一步减少返回的结果数。

Java 中易溢出的比较

您可以使用 CodeQL 检查 Java 代码中的比较，其中比较的一侧容易溢出。

关于这篇文章

在本教程文章中，您将编写一个查询，用于查找循环中整数和长整数之间的比较，这些循环可能会导致溢出导致不终止。
首先，考虑以下代码片段：

```
void foo(long l) {

    for(int i=0; i<l; i++) {

        // do something

    }

}
```

如果 l 大于 231-1（int 类型的最大正值），那么这个循环将永远不会终止：我将从零开始，一直递增到 231-1，这仍然小于 l。当它再次递增时，发生算术溢出，我变成-231，它也小于 l！最终，我会再次达到零，循环重复。
有关溢出的详细信息
所有基元数值类型都有一个最大值，超过这个值，它们将返回到其可能的最小值（称为“溢出”）。对于 int，这个最大值是 231-1。long 类型可以容纳更大的值，最大值为 263-1。在这个例子中，这意味着 l 可以接受一个大于 int 类型的最大值的值；我永远无法达到这个值，而是溢出并返回到一个较低的值。
我们将开发一个查询，它可以找到看起来可能表现出这种行为的代码。我们将使用几个标准库类来表示语句和函数。有关完整列表，请参阅使用 Java 程序的抽象语法树类。

初始查询

我们将首先编写一个查询，该查询查找小于表达式（CodeQL class LTExpr），其中左操作数的类型为 int，右操作数的类型为 long：

```
import java


from LTExpr expr
where expr.getLeftOperand().getType().hasName("int") and
    expr.getRightOperand().getType().hasName("long")
select expr
```

➤请在 LGTM.com 网站. 此查询通常查找大多数项目的结果。

注意，我们使用谓词 getType（可用于 Expr 的所有子类）来确定操作数的类型。反过来，类型定义 hasName 谓词，它允许我们标识 int 和 long 基元类型。目前，这个查询会查找比较 int 和 long 的所有小于表达式，但实际上我们只对循环条件中的比较感兴趣。此外，我们还希望过滤掉任何一个操作数都是常量的比较，因为它们不太可能是真正的 bug。修改后的查询如下所示：

```
import java


from LTExpr expr
where expr.getLeftOperand().getType().hasName("int") and
    expr.getRightOperand().getType().hasName("long") and
    exists(LoopStmt l | l.getCondition().getAChildExpr*() = expr)
and
    not expr.getAnOperand().isCompileTimeConstant()
select expr
```

➤请在 LGTM.com 网站. 请注意，找到的结果较少。

类 LoopStmt 是所有循环的一个公共超类，尤其包括上面例子中的 for 循环。虽然不同类型的循环有不同的语法，但它们都有一个循环条件，可以通过谓词 getCondition 访问。我们使用自反传递闭包运算符*来表示 expr 应该嵌套在循环条件中的要求。特别是，它可以是循环条件本身。
where 子句中的最后一个连接词利用了这样一个事实：谓词可以返回多个值（它们实际上是关系）。特别是，getAnOperand 可能返回 expr 的任意一个操作数，因此 expr.getAnOperand 公司（）.isCompileTimeConstant（）在至少

一个操作数为常量时保持。否定此条件意味着查询将只查找两个操作数都不是常量的表达式。

泛化查询

当然，int 和 long 之间的比较并不是唯一有问题的情况：窄类型和宽类型之间的任何小于比较都可能是可疑的，小于或等于、大于或大于或等于的比较与小于比较一样有问题。
为了比较类型的范围，我们定义了一个谓词，该谓词返回给定整数类型的宽度（以位为单位）：

```
int width(PrimitiveType pt) {

    (pt.hasName("byte") and result=8) or

    (pt.hasName("short") and result=16) or

    (pt.hasName("char") and result=16) or

    (pt.hasName("int") and result=32) or

    (pt.hasName("long") and result=64)

}
```

现在，我们希望将查询推广到任何比较中，其中比较小端的类型宽度小于较大一端类型的宽度。让我们称这种比较容易溢出，并引入一个抽象类来建模：

```
abstract class OverflowProneComparison extends ComparisonExpr {

    Expr getLesserOperand() { none() }

    Expr getGreaterOperand() { none() }

}
```

这个类有两个具体的子类：一个用于<=或<比较，另一个用于>=或>比较。在这两种情况下，我们都以这样一种方式实现构造函数：它只匹配我们想要的表达式：

```
class LTOverflowProneComparison extends OverflowProneComparison {

    LTOverflowProneComparison() {

        (this instanceof LEExpr or this instanceof LTExpr) and

        width(this.getLeftOperand().getType()) <
width(this.getRightOperand().getType())
```

```
    }
}


class GTOverflowProneComparison extends OverflowProneComparison {
    GTOverflowProneComparison() {
        (this instanceof GEExpr or this instanceof GTExpr) and
        width(this.getRightOperand().getType()) <
width(this.getLeftOperand().getType())
    }
}
```

现在我们重写查询以使用这些新类：

```
import Java


// Insert the class definitions from above


from OverflowProneComparison expr
where exists(LoopStmt l | l.getCondition().getAChildExpr*() = expr)
and
not expr.getAnOperand().isCompileTimeConstant()
select expr
```

使用源位置

您可以使用 Java 代码中实体的位置来查找潜在的错误。位置允许您推断是否存在空白，在某些情况下，这可能表示存在问题。

关于源位置

Java 提供了一组具有复杂优先规则的丰富运算符，这些规则有时会使开发人员感到困惑。例如，openjdkjava 编译器中的 ByteBufferCache 类（它是

com.sun.tools 网站.javac.util.BaseFileManager)包含用于分配缓冲区的以下代码：

```
ByteBuffer.allocate(capacity + capacity>>1)
```

据推测，作者打算分配一个缓冲区，其大小是可变容量所示大小的 1.5 倍。然而，事实上，operator+绑定比 operator>>更紧密，因此表达式 capacity+capacity>>1 被解析为（capacity+capacity）>>1，这等于 capacity（除非存在算术溢出）。
请注意，源代码布局给出了一个相当清晰的指示：在+周围比在>>周围有更多的空白，这表明后者是为了更紧密地绑定。
我们将开发一个查询来发现这种可疑的嵌套，其中内部表达式的运算符比外部表达式的运算符周围有更多的空白。这种模式可能不一定表示 bug，但至少它使代码难以阅读并容易被误解。
空白在 CodeQL 数据库中没有直接表示，但是我们可以从与程序元素和 AST 节点相关的位置信息推断出它的存在。因此，在编写查询之前，我们需要了解 Java 标准库中的源位置管理。

位置 API

对于在 Java 源代码中具有表示形式的每个实体（尤其包括程序元素和 AST 节点），标准 CodeQL 库为访问源位置信息提供了以下谓词：

- getLocation 返回一个描述实体的起始位置和结束位置的 Location 对象。
- getFile 返回一个 File 对象，该对象表示包含实体的文件。
- getTotalNumberOfLines 返回实体源代码跨越的行数。
- getNumberOfCommentLines 返回注释行数。
- getNumberOfLinesOfCode 返回非注释行数。

例如，假设这个 Java 类是在编译单元中定义的 SayHello.java 网站：

```
package pkg;


class SayHello {

    public static void main(String[] args) {

        System.out.println(

            // Display personalized message

            "Hello, " + args[0];

        );

    }
```

```
}
```

对 main 主体中的 expression 语句调用 getFile 将返回一个表示该文件的 File 对象 SayHello.java 网站. 该语句总共跨越四行（getTotalNumberOfLines），其中一行是注释行（getNumberOfCommentLines），而三行包含代码（getNumberOfLinesOfCode）。
类位置定义了成员谓词 getstartine、getEndLine、getStartColumn 和 getEndColumn，分别检索实体开始和结束的行号和列号。行和列都从 1 开始计数（不是 0），并且结束位置是包含的，也就是说，它是属于实体源代码的最后一个字符的位置。
在我们的示例中，expression 语句从第 5 行第 3 列开始（行中的前两个字符是制表符，每一个字符计为一个字符），最后在第 8 行第 4 列结束。
类文件定义了以下成员谓词：

- getFullName 返回文件的完全限定名。
- getRelativePath 返回文件相对于源代码基目录的路径。
- getExtension 返回文件的扩展名。
- getShortName 返回文件的基名称，不带扩展名。

在我们的示例中，假设文件 A.java 位于目录/home/testuser/code/pkg，其中/home/testuser/code 是所分析程序的基本目录。然后，a.java 的 File 对象返回：

- getFullName 是/home/testuser/code/pkg/A.java。
- getRelativePath 是 pkg/A.java。
- getExtension 是 java。
- getShortName 是一个。

确定运算符周围的空白

让我们从考虑如何编写一个谓词来计算给定二进制表达式的运算符周围的空白总量。如果 rcol 是表达式右操作数的起始列，而 lcol 是其左操作数的结束列，那么 rcol-（lcol+1）将给出两个操作数之间的字符总数（注意，我们必须使用 lcol+1 而不是 lcol，因为结束位置是包含的）。
这个数字包括运算符本身的长度，我们需要减去它。为此，我们可以使用谓词 getOp，它返回运算符字符串，两边各有一个空格。总体而言，用于计算二进制表达式表达式的运算符周围的空白量的表达式为：

```
rcol - (lcol+1) - (expr.getOp().length()-2)
```

但是，显然，只有当整个表达式在一行上时，这才有效，我们可以使用上面介绍的谓词 getTotalNumberOfLines 来检查这一行。我们现在可以定义谓词来计算运算符周围的空白：

```
int operatorWS(BinaryExpr expr) {

    exists(int lcol, int rcol |
```

```
        expr.getNumberOfLinesOfCode() = 1 and

        lcol = expr.getLeftOperand().getLocation().getEndColumn()
and

        rcol = expr.getRightOperand().getLocation().getStartColumn()
and

        result = rcol - (lcol+1) - (expr.getOp().length()-2)

    )

}
```

注意，我们使用 exists 来引入临时变量 lcol 和 rcol。只需将 lcol 和 rcol 内联到它们的使用中，就可以编写不带它们的谓词，但在可读性方面会有所损失。

找到可疑的巢穴

以下是我们查询的第一个版本：

```
import java


// Insert predicate defined above


from BinaryExpr outer, BinaryExpr inner,

    int wsouter, int wsinner

where inner = outer.getAChildExpr() and

    wsinner = operatorWS(inner) and wsouter = operatorWS(outer) and

    wsinner > wsouter

select outer, "Whitespace around nested operators contradicts
precedence."
```

➤请在 LGTM.com 网站. 此查询可能会找到大多数项目的结果。

where 子句的第一个连接词将 inner 限制为 outer 的操作数，第二个连接词绑定 wsinner 和 wsouter，而最后一个连接词选择可疑案例。
一开始，我们可能会想写 inner=外部。getAnOperand（）在第一个连接处。然而，这并不完全正确：getAnOperand 从其结果中去掉了任何周围的括号，这通

常很有用，但不是我们这里想要的：如果内部表达式周围有圆括号，那么程序员可能知道他们在做什么，查询不应该标记这个表达式。

改进查询

如果我们运行这个初始查询，我们可能会注意到一些由不对称空白引起的误报。例如，以下表达式被标记为可疑，但在实践中不太可能引起混淆：

```
i< start + 100
```

注意，我们的谓词运算符 ws 计算运算符周围的空白总量，在本例中，<为 1，+为 2。理想情况下，我们希望排除运算符前后空白量不同的情况。目前，CodeQL 数据库没有记录足够的信息来解决这个问题，但是作为一个近似值，我们可以要求空白字符的总数是偶数：

```
import java


// Insert predicate definition from above


from BinaryExpr outer, BinaryExpr inner,
    int wsouter, int wsinner
where inner = outer.getAChildExpr() and
    wsinner = operatorWS(inner) and wsouter = operatorWS(outer) and
    wsinner % 2 = 0 and wsouter % 2 = 0 and
    wsinner > wsouter
select outer, "Whitespace around nested operators contradicts
precedence."
```

➤请在 LGTM.com 网站. 任何结果都将通过我们对查询的更改进行优化。

另一个误报的来源是关联运算符：在 x+y+z 形式的表达式中，第一个加号在语法上嵌套在第二个加号中，因为 Java 中的+与左边相关联；因此该表达式被标记为可疑。但是由于+一开始是关联的，所以操作符的嵌套方式并不重要，所以这是错误的肯定。到排除这些情况，让我们定义一个新类，用关联运算符标识二进制表达式：

```
class AssociativeOperator extends BinaryExpr {
```

```
    AssociativeOperator() {

        this instanceof AddExpr or

        this instanceof MulExpr or

        this instanceof BitwiseExpr or

        this instanceof AndLogicalExpr or

        this instanceof OrLogicalExpr

    }

}
```

现在，我们可以扩展查询以丢弃外部表达式和内部表达式具有相同的关联运算符的结果：

```
import java


// Insert predicate and class definitions from above


from BinaryExpr inner, BinaryExpr outer, int wsouter, int wsinner

where inner = outer.getAChildExpr() and

    not (inner.getOp() = outer.getOp() and outer instanceof
AssociativeOperator) and

    wsinner = operatorWS(inner) and wsouter = operatorWS(outer) and

    wsinner % 2 = 0 and wsouter % 2 = 0 and

    wsinner > wsouter

select outer, "Whitespace around nested operators contradicts
precedence."
```

➤请在 LGTM.com 网站.

注意，我们再次使用 getOp，这次是为了确定两个二进制表达式是否具有相同的运算符。运行我们改进的查询，现在可以找到概述中描述的 Java 标准库错误。它还会在 Hadoop HBase 中标记以下可疑代码：

```
KEY_SLAVE = tmp[ i+1 % 2 ];
```

空白表示程序员打算在 0 和 1 之间切换 i，但实际上表达式被解析为 i+（1%2），这与 i+1 相同，所以 i 只是递增的。