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。