java中的注释（Javadoc）

您可以使用CodeQL在Java代码中的Javadoc注释中查找错误。

关于Javadoc（About analyzing Javadoc）

要访问与程序元素关联的Javadoc，我们使用Element类的成员谓词getDoc，它返回一个Documentable。 反过来，Documentable类提供成员谓词getJavadoc来检索附加到所讨论元素的Javadoc（如果有）。

Javadoc注释由Javadoc类表示，该类以JavadocElement节点树的形式提供了注释视图。 每个JavadocElement可以是代表标签的JavadocTag，也可以是代表自由格式文本的JavadocText。

Javadoc类最重要的成员谓词是：

- getAChild–在树表示形式中检索顶级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，并返回此信息：

```
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定义成员谓词getAChild和getParent来在元素树上上下移动。 它还提供谓词getTagName以返回标签的名称，并提供谓词getText以访问与标签关联的文本。

我们可以重写上面的查询以使用此API代替getAuthor和getVersion

```
import java

from Class c, Javadoc jdoc, JavadocTag authorTag, JavadocTag versi
onTag
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具有几个代表特定种类的Javadoc标签的子类：

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

示例：查找虚假的@param标签（Example: Finding spurious @param tags）

作为使用CodeQL Javadoc API的示例，让我们编写一个查询来查找@param标记，该标记引用了不存在的参数。
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);
    }
}
```

此处，A.get上的@param标记将参数列表的名称误拼为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，查找所有可调用对象（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() and
    not c.getAParameter().hasName(pt.getParamName())
select pt, "Spurious @param tag."
```

示例：查找虚假的@throws标签(Example: Finding spurious @throws tags)

一个相关的但更复杂的问题是找到@throws标记，该标记引用了所讨论的方法无法实际抛出的异常。 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 wron
g
    */
    public void foo() {
        // ...
    }
}
```

注意，A.foo的Javadoc注释记录了两个引发的异常：IOException和RuntimeException。 前者显然是虚假的：A.foo没有throws IOException子句，因此不能抛出这种异常。 另一方面，RuntimeException是未经检查的异常，因此即使没有明确的throws子句将其列出，也可以将其抛出。 因此，我们的查询应将IOException标记为@throws标记，而不是RuntimeException标记为@throws标记。
请记住，CodeQL库使用ThrowsTag类表示@throws标签。 此类未提供用于确定要记录的异常类型的成员谓词，因此我们首先需要实现自己的版本。 一个简单的版本可能看起来像这样：

```
RefType getDocumentedException(ThrowsTag tt) {
    result.hasName(tt.getExceptionName())
}
```

同样，Callable并不带有用于查询方法或构造函数可能抛出的所有异常的成员谓词。 但是，我们可以使用getAnException来查找可调用对象的所有throws子句，然后使用getType来解析相应的异常类型，从而自己实现此目的：

```
predicate mayThrow(Callable c, RefType exn) {
    exn.getASupertype*() = c.getAnException().getType()
}
```

注意使用getASupertype *来查找throws子句中声明的异常及其子类型。 例如，如果某个方法具有throws IOException子句，则它可能会抛出MalformedURLException，它是IOException的子类型。
现在，我们可以编写一个查询来查找所有可调用对象c和@throws标签tt，这样：

- tt属于c附带的Javadoc注释。
- c不能抛出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."
```

优化（Improvements）

当前，此查询存在两个问题：
1、getDocumentedException过于宽松：即使其位于其他程序包中并且在当前编译单元中实际上不可见，它也会返回具有正确名称的任何引用类型。
2、mayThrow的限制过于严格：它无法说明无需声明的未经检查的异常。
要了解为什么前一个问题，请考虑以下程序：

```
class IOException extends Exception {}

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

该程序定义了自己的类IOException，该类与标准库中的java.io.IOException类无关：它们位于不同的程序包中。 但是，我们的getDocumentedException谓词不会检查包，因此它将考虑@throws子句引用这两个IOException类，并因此将@param标记标记为虚假的，因为B.bar实际上无法抛出java.io.IOException 。

作为第二个问题的示例，我们前面示例中的方法A.foo带有@throws RuntimeException标记。 但是，我们当前版本的mayThrow会认为A.foo不会抛出RuntimeException，因此将标记标记为虚假。

我们可以通过引入一个新类来表示未检查的异常来减少mayThrow的限制，这些异常只是java.lang.RuntimeException和java.lang.Error的子类型：

```
class UncheckedException extends RefType {
    UncheckedException() {
        this.getASupertype*().hasQualifiedName("java.lang", "RuntimeException") or
        this.getASupertype*().hasQualifiedName("java.lang", "Error")
    }
}
```

现在，我们将此新类合并到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，但是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称。
2、@throws标记引用同一包中的类型。
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将getDocumentedException更改为使用@throws标记的限定名称来解决。 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，该谓词通过属于同一包或被显式导入来检查引用类型在编译单元中是否可见。 然后，我们将getDocumentedException重写为：

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

```
    cu.getPackage() = tp.getPackage()
    or
    exists(ImportType it | it.getCompilationUnit() = cu | it.getI
mportedType() = tp)
}

RefType getDocumentedException(ThrowsTag tt) {
    result.getQualifiedName() = tt.getExceptionName()
    or
    (result.hasName(tt.getExceptionName()) and visibleIn(tt.getFi
le(), result))
}
```