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。