CodeQL的Java类库(CodeQL library for Java)

CodeQL的Java类库（CodeQL library for Java）

分析Java程序时，可以利用CodeQL Java库中的大量类。

Java的CodeQL类库（About the CodeQL library for Java）

有一个广泛的库，用于分析从Java项目中提取的CodeQL数据库。该库中的类以面向对象的形式显示来自数据库的数据，并提供抽象和谓词来帮助您完成常见的分析任务。
该库以一组QL模块（即扩展名为.qll的文件）的形式实现。模块java.qll导入所有核心Java库模块，因此您可以通过以以下内容开始查询来包括完整的库：

```
1 import java
```

本文的其余部分简要总结了此库提供的最重要的类和谓词。

类库摘要（Summary of the library classes）

标准Java库中最重要的类可以分为五个主要类别：

1. 表示程序元素的类（例如类和方法）
2. 表示AST节点的类（例如语句和表达式）

3. 表示元数据的类（例如注释和注释）
4. 用于计算指标的类（例如圈复杂度和耦合）
5. 用于浏览程序调用图的类

我们将依次讨论每个类别，简要描述每个类别的最重要类别。

程序要素（Program elements）

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）。

它们的公共超类是Element，它提供通用成员谓词，用于确定程序元素的名称并检查两个元素是否相互嵌套。

引用可能是方法或构造函数的元素通常很方便； Callable类是Method和Constructor的常见超类，可以用于此目的。

类型（Types）

类`Type`具有许多子类，用于表示不同类型的类型：

- `PrimitiveType`表示基本类型，也就是之一`boolean`，`byte`，`char`，`double`，`float`，`int`，`long`，`short`；QL也将`void`和`<nulltype>`（`null`文字的类型）分类为原始类型。
- `RefType`表示引用（即非原始）类型；它又具有几个子类：
  - `Class` 表示一个Java类。
  - `Interface` 表示一个Java接口。
  - `EnumType`表示Java`enum`类型。
  - `Array` 表示Java数组类型。

例如，以下查询查找程序中所有类型的变量`int`：

```
import java

from Variable v, PrimitiveType pt
where pt = v.getType() and
    pt.hasName("int")
select v
```

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` 是在另一种类型内声明的类型。

例如，此查询查找名称与其编译单元名称不同的所有顶级类型：

```
import java
```

```
2
3 from TopLevelType tl
4 where tl.getName() != tl.getCompilationUnit().getName()
5 select tl
```

更多的实例

- `TopLevelClass` 表示在编译单元的顶层声明的类。
- `NestedClass`表示在另一种类型内声明的类，例如：
  - A `LocalClass`，它是在方法或构造函数内部声明的类。
  - An `AnonymousClass`，这是一个匿名类。

最后，图书馆也有一些单身类中的包装常用的Java标准库类：`TypeObject`，`TypeCloneable`，`TypeRuntime`，`TypeSerializable`，`TypeString`，`TypeSystem`和`TypeClass`。每个CodeQL类代表其名称建议的标准Java类。
例如，我们可以编写一个查询来查找直接扩展的所有嵌套类`Object`：

```
1 import java
2
3 from NestedClass nc
4 where nc.getASupertype() instanceof TypeObject
5 select nc
```

泛型（Generics）

还有一些Type的子类用于处理泛型类型。
GenericType可以是GenericInterface或GenericClass。它表示通用类型声明，例如Java标准库中的接口java.util.Map：

```
1 package java.util.;
2
3 public interface Map<K, V> {
4     int size();
5
6     // ...
7 }
```

类型参数（例如本示例中的K和V）由TypeVariable类表示。

泛型类型的参数化实例提供了一种具体类型，用于实例化类型参数，如Map <String，File>。此类类型由ParameterizedType表示，该类型不同于表示其实例化类型的GenericType的GenericType。要从ParameterizedType转到其对应的GenericType，可以使用谓词getSourceDeclaration。
例如，我们可以使用以下查询找到java.util.Map的所有参数化实例：

```
import java

from GenericInterface map, ParameterizedType pt
where map.hasQualifiedName("java.util", "Map") and
    pt.getSourceDeclaration() = map
select pt
```

通常，通用类型可能会限制类型参数可以绑定到哪些类型。例如，可以声明从字符串到数字的映射类型，如下所示：

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

这意味着StringToNumberMap的参数化实例只能实例化具有Number类型或其子类型之一的类型参数N，而不能实例化File。我们说N是一个有界类型参数，以Number为上限。在QL中，可以使用谓词getATypeBound来查询类型变量的类型绑定。类型范围本身由类TypeBound表示，该类具有成员谓词getType来检索变量所限制的类型。
例如，以下查询查找所有具有类型绑定编号的类型变量：

```
import java

from TypeVariable tv, TypeBound tb
where tb = tv.getATypeBound() and
    tb.getType().hasQualifiedName("java.lang", "Number")
select tv
```

为了处理不了解泛型的遗留代码，每个泛型类型都有一个“原始”版本，没有任何类型参数。在CodeQL库中，原始类型使用RawType类表示，该类具有预期的子类RawClass和RawInterface。同样，有一个谓词getSourceDeclaration用于获取相应的泛型类型。例如，我们可以找到（原始）类型Map的变量：

```
import java
```

```
2
3 from Variable v, RawType rt
4 where rt = v.getType() and
5     rt.getSourceDeclaration().hasQualifiedName("java.util", "Map")
6 select v
```

例如，在以下代码片段中，此查询将找到m1，但找不到m2：

```
1 Map m1 = new HashMap();
2 Map<String, String> m2 = new HashMap<String, String>();
```

最后，可以将变量声明为通配符类型：

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

通配符？扩展Number和？超级浮点数由类WildcardTypeAccess表示。像类型参数一样，通配符也可能具有类型界限。与类型参数不同，通配符可以具有上限（如？extended Number中的值），也可以具有下限（如？super Float中的值）。 WildcardTypeAccess类提供成员谓词getUpperBound和getLowerBound分别检索上限和下限。
为了处理泛型方法，有GenericMethod，ParameterizedMethod和RawMethod类，它们与用于表示泛型类型的同名类完全相似。

变量（Variables）

类变量表示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），或者是局部变量或参数。因此，有三个子类可以满足这些特殊情况：

- Field表示一个Java字段。
- LocalVariableDecl表示局部变量。
- Parameter表示方法或构造函数的参数。

抽象语法树（Abstract syntax tree）

此类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，请参见“用于Java程序的抽象语法树类”。
Expr和Stmt都提供成员谓词，用于探索程序的抽象语法树：

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

例如，以下查询查找其父代为return语句的所有表达式：

```
import java

from Expr e
where e.getParent() instanceof ReturnStmt
select e
```

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

```
import java

from Stmt s
where s.getParent() instanceof IfStmt
select s
```

该查询将找到程序中所有if语句的then分支和else分支。
最后，这是一个查找方法主体的查询：

```
import java

from Stmt s
where s.getParent() instanceof Method
select s
```

如这些示例所示，表达式的父节点并不总是一个表达式：它也可以是一条语句，例如IfStmt。同样，一条语句的父节点并不总是一条语句：它也可以是方法或构造函数。为了捕获这一点，QL Java库提供了两个抽象类ExprParent和StmtParent，前者表示可能是表达式的父节点的任何节点，而后者则可能是语句的父节点的任何节点。

元数据（Metadata）

Java程序除了适当的程序代码外，还具有几种元数据。特别是，有注释和Javadoc注释。由于此元数据对于增强代码分析和作为本身的分析主题都很有趣，因此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。对于每个此类元素，其谓词getAnAnnotation允许您检索该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注释：

```
import java

from Constructor c
select c.getAnAnnotation()
```

这些注释由类Annotation表示。注释只是其类型为AnnotationType的表达式。例如，您可以修改此查询，以使其仅报告已弃用的构造函数：

```
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
```

对于Javadoc，类Element具有成员谓词getDoc，该成员谓词返回委托的Documentable对象，然后可以查询该对象的附加Javadoc注释。例如，以下查询在私有字段中查找Javadoc注释：

```
import java

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

Javadoc类将整个Javadoc注释表示为JavadocElement节点树，可以使用成员谓词getAChild和getParent遍历该树。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
```

指标（Metrics）

标准的QL Java库为Java程序元素上的计算指标提供了广泛的支持。为避免用过多与度量计算相关的成员谓词来使代表那些元素的类负担过多，请改为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`。每个对应的元素类都提供一个成员谓词`getMetrics`，该成员谓词可用于获取委托类的实例，然后可以在其上执行度量计算。
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
```

调用图（Call graph）

从Java代码库生成的CodeQL数据库包含有关程序调用图的预先计算的信息，即，给定调用可以在运行时分派给哪些方法或构造函数。
上面介绍的Callable类包括方法和构造函数。调用表达式使用Call类来抽象，其中包括方法调用，新表达式以及使用this或super的显式构造函数调用。
我们可以使用谓词`Call.getCallee`来找出特定调用表达式所引用的方法或构造函数。例如，以下查询查找对称为的方法的所有调用`println`：

```
import java

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

相反，`Callable.getAReference`返回`Call`引用它的a。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
```