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。