浏览调用图

CodeQL 有用于标识调用其他代码的代码和可以从其他地方调用的代码的类。例如，这允许您找到从未使用过的方法。

调用图类

CodeQL Java 库提供了两个抽象类来表示程序的调用图：Callable 和 call。前者只是 Method 和 Constructor 的公共超类，后者是 MethodAccess、ClassInstanceExpression、ThisConstructorInvocationStmt 和超结构 InvocationsTM 的通用超类。简单地说，可调用是可以调用的，而调用是调用可调用的。

例如，在以下程序中，所有可调用项和调用都已用注释进行了注释：

```
class Super {

    int x;


    // callable

    public Super() {

        this(23);       // call

    }


    // callable

    public Super(int x) {

        this.x = x;

    }


    // callable

    public int getX() {

        return x;

    }

}
```

```
    class Sub extends Super {
    // callable
    public Sub(int x) {
        super(x+19);    // call
    }

    // callable
    public int getX() {
        return x-19;
    }
}

class Client {
    // callable
    public static void main(String[] args) {
        Super s = new Sub(42);  // call
        s.getX();               // call
    }
}
```

类调用提供两个调用图导航谓词：

- getCallee 返回此调用（静态）解析到的可调用；请注意，对于对实例（即非静态）方法的调用，在运行时调用的实际方法可能是重写此方法的其他方法。
- getCaller 返回此调用在语法上是其一部分的可调用项。

例如，在我们的示例中，第二个调用的 getCallee 客户端.main 会回来的超级.getX. 不过，在运行时，这个调用实际上会调用 Sub.getX 公司.
类 Callable 定义了大量成员谓词；就我们的目的而言，两个最重要的谓词是：

- 如果此 Callable 包含一个被调用方是 target 的调用，则 calls（Callable target）将成功。
- 如果 polyCalls（Callable target）可以在运行时调用 target，则 polyCalls（Callable target）将成功；如果它包含的调用的被调用方是 target 或 target 重写的方法，则是这种情况。

在我们的例子中，客户端.main 调用构造函数 Sub（int）和方法超级.getX；此外，它还采用了聚合方法 Sub.getX 公司.

示例：查找未使用的方法

我们可以使用 Callable 类编写一个查询，该查询查找未被任何其他方法调用的方法：

```
import java



from Callable callee

where not exists(Callable caller | caller.polyCalls(callee))

select callee
```

➤请在 LGTM.com 网站.这个简单的查询通常返回大量结果。

注意
我们必须在这里使用 polycall 而不是调用：我们希望合理地确保被调用方没有被调用，无论是直接调用还是通过重写。
在一个典型的 Java 项目上运行这个查询会导致 Java 标准库中的大量命中。这是有意义的，因为没有一个客户端程序使用标准库的所有方法。更一般地说，我们可能希望从编译的库中排除方法和构造函数。我们可以使用 fromSource 谓词来检查编译单元是否是源文件，并优化查询：

```
import java



from Callable callee

where not exists(Callable caller | caller.polyCalls(callee)) and

    callee.getCompilationUnit().fromSource()

select callee, "Not called."
```

➤请在 LGTM.com 网站.此更改减少了大多数项目返回的结果数。

我们还可能注意到一些未使用的方法，它们的名称有些奇怪：它们是类初始值设定项；虽然它们不是在代码中的任何地方显式调用的，但只要周围的类被加载，它们就会被隐式调用。因此，从我们的查询中排除它们是有意义的。在进行此操作时，我们还可以排除终结器，它们同样被隐式调用：

```
import java


from Callable callee
where not exists(Callable caller | caller.polyCalls(callee)) and
    callee.getCompilationUnit().fromSource() and
    not callee.hasName("<clinit>") and not
callee.hasName("finalize")
select callee, "Not called."
```

➤请在 LGTM.com 网站. 这也减少了大多数项目返回的结果数量。

我们还可能希望从查询中排除公共方法，因为它们可能是外部 API 入口点：

```
import java


from Callable callee
where not exists(Callable caller | caller.polyCalls(callee)) and
    callee.getCompilationUnit().fromSource() and
    not callee.hasName("<clinit>") and not
callee.hasName("finalize") and
    not callee.isPublic()
select callee, "Not called."
```

➤请在 LGTM.com 网站. 这对返回的结果数量应该有更明显的影响。

另一个特殊情况是非公共的默认构造函数：例如，在 singleton 模式中，为类提供私有的空默认构造函数，以防止它被实例化。由于此类构造函数的目的是不被调用，因此不应标记它们：

```
import java
```

```
from Callable callee

where not exists(Callable caller | caller.polyCalls(callee)) and

    callee.getCompilationUnit().fromSource() and

    not callee.hasName("<clinit>") and not
callee.hasName("finalize") and

    not callee.isPublic() and

    not callee.(Constructor).getNumberOfParameters() = 0

select callee, "Not called."
```

➤请在 LGTM.com 网站. 这种变化对某些项目的结果有很大影响，但对其他项

目的结果影响不大。这种模式的使用在不同的项目中差别很大。

最后，在许多 Java 项目中，有一些方法是通过反射间接调用的。因此，虽然没有调用调用这些方法的调用，但它们实际上是被使用的。一般来说，很难确定这种方法。然而，一个非常常见的特殊情况是 JUnit 测试方法，它由测试运行程序反射性地调用。QL-Java 库支持识别 JUnit 和其他测试框架的测试类，我们可以用它来过滤这些类中定义的方法：

```
import java


from Callable callee

where not exists(Callable caller | caller.polyCalls(callee)) and

    callee.getCompilationUnit().fromSource() and

    not callee.hasName("<clinit>") and not
callee.hasName("finalize") and

    not callee.isPublic() and

    not callee.(Constructor).getNumberOfParameters() = 0 and

    not callee.getDeclaringType() instanceof TestClass

select callee, "Not called."
```

➤请在 LGTM.com 网站. 这将进一步减少返回的结果数。