导航调用图（Navigating the call graph）

CodeQL具有用于标识调用其他代码的代码以及可以从其他位置调用的代码的类。例如，这使您可以找到从未使用过的方法。

调用图类（Call graph classes）

Java的CodeQL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的通用超类，后者是MethodAccess，ClassInstanceExpression，ThisConstructorInvocationStmt和SuperConstructorInvocationStmt的通用超类。 简而言之，Callable是可以调用的事物，而Call是调用Callable的事物。
例如，在下面的程序中，所有可调用项和调用都带有注释注释：

```
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
    }
}
```

类调用提供了两个调用图导航谓词：

- getCallee返回此调用（静态）解析为的Callable； 请注意，对于实例方法（即非静态方法）的调用，在运行时调用的实际方法可能是覆盖该方法的其他方法。
- getCaller返回此调用在语法上属于其的Callable。

例如，在我们的示例中，Client.main中第二个调用的getCallee将返回Super.getX。 但是，在运行时，此调用实际上将调用Sub.getX。

Callable类定义了大量的成员谓词； 就我们的目的而言，两个最重要的是：

- calls（Callable target），如果此可调用对象包含被叫方为目标的调用，则calls（可呼叫目标）成功。
- ployCalls（Callable target）如果此callable可以在运行时调用目标，则polyCalls（Callable target）成功。 如果它包含被调用方为目标或目标覆盖的方法的调用，则为这种情况。

在我们的示例中，Client.main调用构造函数Sub（int）和方法Super.getX;。 此外，它还使用polyCalls方法Sub.getX。

示例：寻找未被调用的方法（Example: Finding unused methods）

我们可以使用Callable类来编写查询，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
```

```
我们必须在这里使用polyCalls而不是调用：我们要合理地确保没有直接或通过覆盖调用被调用方。
```

在典型的Java项目上运行此查询会导致Java标准库中出现很多匹配项。 这是有道理的，因为没有单个客户端程序使用标准库的每种方法。 更一般而言，我们可能要从编译的库中排除方法和构造函数。 我们可以使用谓词fromSource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."
```

我们可能还会注意到一些未使用的名称为<clinit>的方法：这些是类初始化器；它们是类初始化器。 尽管在代码中的任何地方都没有显式调用它们，但是在加载周围的类时会隐式调用它们。 因此，将它们从我们的查询中排除是有意义的。 在此过程中，我们还可以排除finalize类型的成员及变量：

```
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."
```

我们可能还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
```

```
2
3 from Callable callee
4 where not exists(Callable caller | caller.polyCalls(callee)) and
5     callee.getCompilationUnit().fromSource() and
6     not callee.hasName("<clinit>") and not callee.hasName("finaliz
  e") and
7     not callee.isPublic()
8 select callee, "Not called."
```

另一个特殊情况是非公共默认构造函数：例如，在单例模式中，为类提供了私有的空默认构造函数，以防止实例化它。 由于此类构造函数的真正目的是不调用它们，因此不应对其进行标记：

```
1 import java
2
3 from Callable callee
4 where not exists(Callable caller | caller.polyCalls(callee)) and
5     callee.getCompilationUnit().fromSource() and
6     not callee.hasName("<clinit>") and not callee.hasName("finaliz
  e") and
7     not callee.isPublic() and
8     not callee.(Constructor).getNumberOfParameters() = 0
9 select callee, "Not called."
```

此更改对某些项目的结果影响很大，而对其他项目的结果影响很小。 在不同项目之间，此模式的使用差异很大。

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，尽管没有调用这些方法的调用，但实际上它们是被使用的。 通常很难识别这种方法。 但是，一个非常常见的特殊情况是JUnit测试方法，由测试运行器反射地调用。 Java的CodeQL库支持识别JUnit和其他测试框架的测试类，我们可以使用它们来过滤掉在此类中定义的方法：

```
1 import java
2
3 from Callable callee
4 where not exists(Callable caller | caller.polyCalls(callee)) and
5     callee.getCompilationUnit().fromSource() and
6     not callee.hasName("<clinit>") and not callee.hasName("finali
  ze") and
```

```
    not callee.isPublic() and
    not callee.(Constructor).getNumberOfParameters() = 0 and
    not callee.getDeclaringType() instanceof TestClass
select callee, "Not called."
```