Java 类型

可以使用 CodeQL 来查找有关 Java 代码中使用的数据类型的信息。这允许您编写查询来识别与类型相关的特定问题。

关于使用 Java 类型

标准 CodeQL 库通过类型类及其各种子类来表示 Java 类型。
特别是，类 PrimitiveType 表示构建在 Java 语言中的基元类型（例如 boolean 和 int），而 RefType 及其子类表示引用类型，即类、接口、数组类型等等。这包括 Java 标准库中的两种类型（如 java.lang.Object）以及由非库代码定义的类型。
类 RefType 还为类层次结构建模：成员谓词 getASupertype 和 getASubtype 允许您查找引用类型的直接超类型和子类型。例如，考虑以下 Java 程序：

```
class A {}


interface I {}


class B extends A implements I {}
```

在这里，A 类正好有一个直接超类型（java.lang.Object）而且只有一个立即子类型（B）；接口 I 也是如此。另一方面，类 B 有两个立即超类型（A 和 I），没有立即子类型。
为了确定祖先类型（包括直接超类型，以及它们的超类型等），我们可以使用传递闭包。例如，要在上面的示例中查找 B 的所有祖先，可以使用以下查询：

```
import java


from Class B

where B.hasName("B")

select B.getASupertype+()
```

➤请在 LGTM.com 网站. 如果在上面的示例片段上运行此查询，则查询将返回

A、I 和 java.lang.Object.

小费

如果要查看 B 和 A 的位置，可以将 B.getASupertype+（）替换为 B.getASupertype*（），然后重新运行查询。
除了类层次结构建模之外，RefType 还提供成员谓词 getAMember 以访问在类型中声明的成员（即字段、构造函数和方法），谓词继承（方法 m）用于检查类型是否声明或继承方法 m。

示例：查找有问题的数组转换

作为如何使用类层次结构 API 的一个示例，我们可以编写一个查询来查找数组上的向下转换，也就是说，在某种类型 a[]的表达式 e 转换为类型 B[]的情况下，B 是 a 的（不一定是立即的）子类型。
这种类型的转换是有问题的，因为向下转换数组会导致运行时异常，即使每个单独的数组元素都可以被下推。例如，以下代码引发 ClassCastException：

```
Object[] o = new Object[] { "Hello", "world" };

String[] s = (String[])o;
```

另一方面，如果表达式 e 碰巧实际计算为 B[]数组，则转换将成功：

```
Object[] o = new String[] { "Hello", "world" };

String[] s = (String[])o;
```

在本教程中，我们不尝试区分这两种情况。我们的查询只需查找从某个类型源强制转换到另一个类型目标的转换表达式 ce，例如：

- 源和目标都是数组类型。
- source 的元素类型是 target 元素类型的可传递超类型。

这个配方不难翻译成一个查询：

```
import java


from CastExpr ce, Array source, Array target

where source = ce.getExpr().getType() and

    target = ce.getType() and

    target.getElementType().(RefType).getASupertype+() =
source.getElementType()

select ce, "Potentially problematic array downcast."
```

➤请在 LGTM.com 网站. 许多项目返回此查询的结果。

注意，通过铸造 target.getElementType（）对于 RefType，我们消除了元素类型是基元类型的所有情况，也就是说，目标是基元类型的数组：在这种情况下，我们要寻找的问题不会出现。与 Java 不同，QL 中的强制转换永远不会失败：如果一个表达式不能转换为所需的类型，它将被简单地从查询结果中排除，这正是我们想要的。

改进

在版本 5 之前的旧 Java 代码上运行此查询，通常会返回由于使用该方法而产生的许多误报结果集合.toArray（T[]），它将集合转换为 T[]类型的数组。
在不使用泛型的代码中，此方法通常按以下方式使用：

```
List l = new ArrayList();

// add some elements of type A to l

A[] as = (A[])l.toArray(new A[0]);
```

这里，l 有原始类型列表，因此 l.toArray 有返回类型 Object[]，独立于其参数数组的类型。因此，转换从 Object[]变成了一个[]，并将被我们的查询标记为有问题，尽管在运行时这个转换永远不会出错。
为了识别这些情况，我们可以创建两个 CodeQL 类，分别表示集合.toArray 类，并调用此方法或重写它的任何方法：

```
/** class representing java.util.Collection.toArray(T[]) */

class CollectionToArray extends Method {

    CollectionToArray() {

        this.getDeclaringType().hasQualifiedName("java.util",
"Collection") and

        this.hasName("toArray") and

        this.getNumberOfParameters() = 1

    }

}


/** class representing calls to java.util.Collection.toArray(T[]) */

class CollectionToArrayCall extends MethodAccess {

    CollectionToArrayCall() {
```

```
        exists(CollectionToArray m |

this.getMethod().getSourceDeclaration().overridesOrInstantiates*(m)

        )

    }


    /** the call's actual return type, as determined from its
argument */

    Array getActualReturnType() {

        result = this.getArgument(0).getType()

    }

}
```

注意在 CollectionToArrayCall 的构造函数中使用了 getSourceDeclaration 和 overridesOrInstantiates:我们希望找到集合.toArray 以及重写它的任何方法，以及这些方法的任何参数化实例。例如，在上面的示例中，调用 l.toArray 解析为原始类 ArrayList 中的方法 toArray。它的源声明是泛型类 ArrayList<T>中的 toArray，它重写 AbstractCollection<T>.toArray，后者又重写 Collection<T>.toArray，它是集合.toArray（因为重写方法中的类型参数 T 属于 ArrayList，是属于集合的类型参数的实例化）。
使用这些新类，我们可以扩展查询以排除对类型 A[]的参数的 toArray 调用，然后将其转换为[]：

```
import java


// Insert the class definitions from above


from CastExpr ce, Array source, Array target

where source = ce.getExpr().getType() and

    target = ce.getType() and

    target.getElementType().(RefType).getASupertype+() =
source.getElementType() and
```

```
    not ce.getExpr().(CollectionToArrayCall).getActualReturnType() =
target

select ce, "Potentially problematic array downcast."
```

➤请在 LGTM.com 网站. 请注意，通过这个改进的查询可以找到更少的结果。

示例：查找不匹配的包含检查

现在，我们将开发一个查询来查找集合. 包含其中被查询元素的类型与集合的元素类型无关，这保证测试始终返回 false。
例如，Apache Zookeeper 曾经在 QuorumPeerConfig 类中有一段类似于以下内容的代码片段：

```
Map<Object, Object> zkProp;


// ...


if (zkProp.entrySet().contains("dynamicConfigFile")){

    // ...

}
```

因为 zkProp 是从一个对象到另一个对象的映射，zkProp. entrySet 公司返回 Set<Entry<Object，Object>>类型的集合。这样的集合不能包含 String 类型的元素。（该代码后来被固定使用 zkProp. containsKey 公司.）
一般来说，我们希望找到集合. 包含（或其在集合的任何参数化实例中的任何重写方法），使得集合元素的类型 E 和要包含的参数的类型 A 不相关，即它们没有公共子类型。
我们首先创建一个类来描述 java. util. Collection:

```
class JavaUtilCollection extends GenericInterface {

    JavaUtilCollection() {

        this.hasQualifiedName("java.util", "Collection")

    }

}
```

为了确保没有输入错误，我们可以运行一个简单的测试查询：

```
from JavaUtilCollection juc

select juc
```

这个查询应该只返回一个结果。
接下来，我们可以创建一个类来描述 java.util.Collection。包含：

```
class JavaUtilCollectionContains extends Method {

    JavaUtilCollectionContains() {

        this.getDeclaringType() instanceof JavaUtilCollection and

        this.hasStringSignature("contains(Object)")

    }

}
```

请注意，我们使用 hasStringSignature 来检查：

- 有问题的方法的名称包含。
- 它只有一个论点。
- 参数的类型是 Object。

或者，我们可以使用 TypeObject 的 hasName、getNumberOfParameters 和 getParameter（0）.getType（） instanceof TypeObject 更详细地实现这三个检查。
如前所述，通过运行一个简单的查询来选择 JavaUtilCollectionContains 的所有实例来测试新类是一个好主意；同样应该只有一个结果。
现在我们要确定所有呼叫集合.包含，包括重写它的任何方法，并考虑集合及其子类的所有参数化实例。也就是说，我们正在寻找被调用方法的源声明（反射性或传递性）重写的方法访问集合.包含. 我们在 CodeQL 类 JavaUtilCollectionContainsCall 中对其进行编码：

```
class JavaUtilCollectionContainsCall extends MethodAccess {

    JavaUtilCollectionContainsCall() {

        exists(JavaUtilCollectionContains jucc |

            this.getMethod().getSourceDeclaration().overrides*(jucc)

        )

    }

}
```

这个定义有点微妙，所以您应该运行一个简短的查询来测试 JavaUtilCollectionContainsCall 是否正确地识别了对集合.包含.
对于对 contains 的每次调用，我们对两件事感兴趣：参数的类型和调用它的集合的元素类型。因此，我们需要将两个成员谓词 getArgumentType 和 getCollectionElementType 添加到类 JavaUtilCollectionContainsCall 中来计算这些信息。
前者很简单：

```
Type getArgumentType() {

    result = this.getArgument(0).getType()

}
```

对于后者，我们的工作如下：

- 查找正在调用的 contains 方法的声明类型 D。
- 找到 D 的（自反或传递）超类型 S，它是的参数化实例 java.util.Collection.
- 返回 S 的（唯一）类型参数。

我们将其编码如下：

```
Type getCollectionElementType() {

    exists(RefType D, ParameterizedInterface S |

        D = this.getMethod().getDeclaringType() and

        D.hasSupertype*(S) and S.getSourceDeclaration() instanceof
JavaUtilCollection and

        result = S.getTypeArgument(0)

    )

}
```

在将这两个成员谓词添加到 JavaUtilCollectionContainsCall 之后，我们需要编写一个谓词来检查两个给定的引用类型是否具有公共子类型：

```
predicate haveCommonDescendant(RefType tp1, RefType tp2) {

    exists(RefType commondesc | commondesc.hasSupertype*(tp1) and
commondesc.hasSupertype*(tp2))

}
```

现在我们准备编写查询的第一个版本：

```
import java


// Insert the class definitions from above


from JavaUtilCollectionContainsCall juccc, Type collEltType, Type
argType

where collEltType = juccc.getCollectionElementType() and argType =
juccc.getArgumentType() and

    not haveCommonDescendant(collEltType, argType)

select juccc, "Element type " + collEltType + " is incompatible with
argument type " + argType
```

➤请在 LGTM.com 网站.

改进

对于许多程序，由于类型变量和通配符，此查询会产生大量误报结果：例如，如果集合元素类型是某个类型变量 E，而参数类型是 String，则 CodeQL 将认为这两个元素没有公共子类型，我们的查询将标记调用。排除这种误报结果的一个简单方法是简单地要求 collEltType 和 argType 都不是 TypeVariable 的实例。
另一个误报的来源是原语类型的自动装箱：例如，如果集合的元素类型是 Integer 而参数是 int 类型，则谓词 haveCommonDescendant 将失败，因为 int 不是 RefType。为了说明这一点，我们的查询应该检查 collEltType 不是 argType 的装箱类型。
最后，null 是特殊的，因为它的类型（在 CodeQL 库中称为<nulltype>）与每个引用类型都兼容，因此我们应该将其排除在考虑范围之外。
加上这三个改进，我们的最后一个问题是：

```
import java


// Insert the class definitions from above


```

```
from JavaUtilCollectionContainsCall juccc, Type collEltType, Type
argType

where collEltType = juccc.getCollectionElementType() and argType =
juccc.getArgumentType() and

    not haveCommonDescendant(collEltType, argType) and

    not collEltType instanceof TypeVariable and not argType
instanceof TypeVariable and

    not collEltType = argType.(PrimitiveType).getBoxedType() and

    not argType.hasName("<nulltype>")

select juccc, "Element type " + collEltType + " is incompatible with
argument type " + argType
```

➤在上的查询控制台中查看完整查询 LGTM.com 网站.