Java中的类型（Type in Java）

您可以使用CodeQL查找有关Java代码中使用的数据类型的信息。这使您可以编写查询来识别与类型相关的特定问题。

关于使用Java类型的工作（About working with Java types）

标准的CodeQL库通过Type类及其各种子类表示Java类型。
特别是，类PrimitiveType表示Java语言中内置的基元类型（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。 这包括Java标准库中的类型（例如java.lang.Object）和非库代码定义的类型。
RefType类还可以对类层次结构进行建模：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，类A仅具有一个直接的父类型（java.lang.Object）和恰好具有一种直接子类型（B）； 接口I同样如此。另一方面，类B具有两个立即超类型（A和I），并且没有立即子类型。

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，我们可以使用以下查询：

```
import java

```

```
from Class B
where B.hasName("B")
select B.getASupertype+()
```

tips:
如果你想看到的位置`B`，以及`A`，你可以替换`B.getASupertype+()`使用`B.getASupertype*()`，并重新运行该查询。
除了类层次结构建模之外，RefType还提供成员谓词getAMember来访问在类型中声明的成员（即字段，构造函数和方法），并提供谓词Inherited（方法m）来检查类型是否声明或继承方法m。

```
tips:如果你想看到的位置B，以及A，你可以替换B.getASupertype+()使用B.getASupertype*()，并重新运行该查询。
```

示例：查找有问题的数组强制转换（Example: Finding problematic array casts）

作为如何使用类层次结构API的示例，我们可以编写一个查询来查找数组上的下转换，即将某些类型为A []的表达式e转换为类型B []的情况，例如B是 （不一定是立即数）A的子类型。

这种转换是有问题的，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向下转换也是如此。 例如，以下代码引发ClassCastException：

另一方面，如果表达式e恰好实际上是对B []数组求值，则强制转换将成功：

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

- 两个`source`和`target`是数组类型。
- 的元素类型`source`是的元素类型的传递超类型`target`。

将此方法翻译成查询不是很困难：

```
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."
```

请注意，通过将target.getElementType（）强制转换为RefType，可以消除元素类型为基本类型（即target为基本类型的数组）的所有情况：在这种情况下，我们所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：如果表达式无法强制转换为所需的类型，则将其从查询结果中排除，这正是我们想要的。

改进（improvement）

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Collection.toArray（T []）方法而产生的许多误报结果，该方法将集合转换为T []类型的数组。
在不使用泛型的代码中，通常按以下方式使用此方法

```
List l = new ArrayList();
// add some elements of type A to l
A[] as = (A[])l.toArray(new A[0]);
```

在这里，l具有原始类型List，因此l.toArray的返回类型Object []与其参数数组的类型无关。 因此，强制类型转换从Object []到A []并被我们的查询标记为有问题的，尽管在运行时此强制类型永远不会出错。

为了识别这些情况，我们可以创建两个CodeQL类，分别表示Collection.toArray方法，并调用此方法或任何重写该方法的方法：

```
/** 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()
    }
}
```

请注意，在CollectionToArrayCall的构造函数中使用了getSourceDeclaration和overridesOrInstantiates：我们要查找对Collection.toArray以及对其进行覆盖的任何方法的调用，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toArray方法。 它的源声明是通用类ArrayList <T>中的toArray，它重写了AbstractCollection <T> .toArray，后者又重写了Collection <T> .toArray，后者是Collection.toArray的实例化（因为 覆盖的方法属于ArrayList，并且是属于Collection的类型参数的实例。

使用这些新类，我们可以扩展查询，以排除对类型为A []的参数的toArray的调用，然后将其强制转换为A []：

```
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."
```

示例：查找不匹配包含检查（Example: Finding mismatched contains checks）

现在，我们将开发一个查询，以查找Collection的用途。该查询包含所查询元素的类型与该集合的元素类型无关的位置，从而保证测试始终返回false。

例如，Apache Zookeeper过去在QuorumPeerConfig类中具有类似于以下代码的代码段：

```
Map<Object, Object> zkProp;

// ...

if (zkProp.entrySet().contains("dynamicConfigFile")){
    // ...
}
```

由于zkProp是从Object到Object的映射，因此zkProp.entrySet返回类型为Set <Entry <Object, Object >>的集合。 这样的集合可能不能包含String类型的元素。 （此后，该代码已修复为使用zkProp.containsKey。）

通常，我们要查找对Collection.contains的调用（或其在Collection的任何参数化实例中的任何覆盖方法），以使collection元素的类型E与要包含的参数的类型A不相关，即，他们没有共同的亚型。

我们首先创建一个描述java.util.Collection的类：

```
class JavaUtilCollection extends GenericInterface {
    JavaUtilCollection() {
        this.hasQualifiedName("java.util", "Collection")
    }
}
```

为了确保我们没有输错任何东西，我们可以运行一个简单的测试查询：

```
from JavaUtilCollection juc
select juc
```

此查询应恰好返回一个结果。

接下来，我们可以创建一个描述java.util.Collection.contains的类：

```
class JavaUtilCollectionContains extends Method {
    JavaUtilCollectionContains() {
        this.getDeclaringType() instanceof JavaUtilCollection and
        this.hasStringSignature("contains(Object)")
    }
}
```

请注意，我们使用hasStringSignature来检查以下内容：

- 有问题的方法包含名称。
- 它只有一个论点。
- 参数的类型是对象。

或者，我们可以使用TypeObject的hasName，getNumberOfParameters和getParameter（0）.getType（）更加详细地实现这三个检查。
和以前一样，最好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。同样，应该只有一个结果。
现在，我们要确定对Collection.contains的所有调用，包括覆盖它的所有方法，并考虑Collection及其子类的所有参数化实例。 也就是说，我们正在寻找方法访问，其中被调用方法的源声明（反射地或传递地）覆盖Collection.contains。 我们将其编码在CodeQL类JavaUtilCollectionContainsCall中：

```
class JavaUtilCollectionContainsCall extends MethodAccess {
    JavaUtilCollectionContainsCall() {
        exists(JavaUtilCollectionContains jucc |
            this.getMethod().getSourceDeclaration().overrides*(jucc)
        )
    }
}
```

这个定义有些微妙，因此您应该运行一个简短的查询来测试JavaUtilCollectionContainsCall是否正确识别对Collection.contains的调用。

对于每个对contains的调用，我们都对两件事感兴趣：参数的类型以及对其进行调用的集合的元素类型。因此，我们需要在类JavaUtilCollectionContainsCall中添加两个成员谓词getArgumentType和getCollectionElementType来计算此信息。

前者很容易：

```
Type getArgumentType() {
    result = this.getArgument(0).getType()
}
```

对于后者，我们进行如下操作：

- 查找被调用的contains方法的声明类型D。
- 查找D的（自反或可传递）超类型S，它是java.util.Collection的参数化实例。
- 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)
    )
}
```

将这两个成员谓词添加到JavaUtilCollectionContainsCall之后，我们需要编写一个谓词，以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 wi
  th argument type " + argType
```

改进（Improvements）

对于许多程序，由于类型变量和通配符，此查询会产生大量的误报结果：例如，如果集合元素类型为某种类型变量E并且参数类型为String，则CodeQL会认为这两者没有 common子类型，我们的查询将标记该调用。 排除这种假阳性结果的一种简单方法是简单地要求collEltType和argType都不是TypeVariable的实例。

误报的另一个来源是原始类型的自动装箱：例如，如果集合的元素类型为Integer且参数的类型为int，则谓词haveCommonDescendant将失败，因为int不是RefType。 为了解决这个问题，我们的查询应检查collEltType不是argType的框式类型。

最后，null是特殊的，因为它的类型（在CodeQL库中称为<nulltype>）与每种引用类型都兼容，因此我们应将其排除在考虑范围之外。

添加这三个改进后，我们的最终查询变为：

```
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 insta
  nceof TypeVariable and
    not collEltType = argType.(PrimitiveType).getBoxedType() and
    not argType.hasName("<nulltype>")
select juccc, "Element type " + collEltType + " is incompatible w
  ith argument type " + argType
```