使用源位置

您可以使用 Java 代码中实体的位置来查找潜在的错误。位置允许您推断是否存在空白，在某些情况下，这可能表示存在问题。

关于源位置

Java 提供了一组具有复杂优先规则的丰富运算符，这些规则有时会使开发人员感到困惑。例如，openjdkjava 编译器中的 ByteBufferCache 类（它是 com.sun.tools 网站.javac.util.BaseFileManager)包含用于分配缓冲区的以下代码：

```
ByteBuffer.allocate(capacity + capacity>>1)
```

据推测，作者打算分配一个缓冲区，其大小是可变容量所示大小的 1.5 倍。然而，事实上，operator+绑定比 operator>>更紧密，因此表达式 capacity+capacity>>1 被解析为（capacity+capacity）>>1，这等于 capacity（除非存在算术溢出）。
请注意，源代码布局给出了一个相当清晰的指示：在+周围比在>>周围有更多的空白，这表明后者是为了更紧密地绑定。
我们将开发一个查询来发现这种可疑的嵌套，其中内部表达式的运算符比外部表达式的运算符周围有更多的空白。这种模式可能不一定表示 bug，但至少它使代码难以阅读并容易被误解。
空白在 CodeQL 数据库中没有直接表示，但是我们可以从与程序元素和 AST 节点相关的位置信息推断出它的存在。因此，在编写查询之前，我们需要了解 Java 标准库中的源位置管理。

位置 API

对于在 Java 源代码中具有表示形式的每个实体（尤其包括程序元素和 AST 节点），标准 CodeQL 库为访问源位置信息提供了以下谓词：

- getLocation 返回一个描述实体的起始位置和结束位置的 Location 对象。
- getFile 返回一个 File 对象，该对象表示包含实体的文件。
- getTotalNumberOfLines 返回实体源代码跨越的行数。
- getNumberOfCommentLines 返回注释行数。
- getNumberOfLinesOfCode 返回非注释行数。

例如，假设这个 Java 类是在编译单元中定义的 SayHello.java 网站：

```
package pkg;


class SayHello {

    public static void main(String[] args) {
```

```
            System.out.println(
                // Display personalized message
                "Hello, " + args[0];
            );
        }
    }
```

对 main 主体中的 expression 语句调用 getFile 将返回一个表示该文件的 File 对象 SayHello.java 网站. 该语句总共跨越四行（getTotalNumberOfLines），其中一行是注释行（getNumberOfCommentLines），而三行包含代码（getNumberOfLinesOfCode）。
类位置定义了成员谓词 getstartine、getEndLine、getStartColumn 和 getEndColumn，分别检索实体开始和结束的行号和列号。行和列都从 1 开始计数（不是 0），并且结束位置是包含的，也就是说，它是属于实体源代码的最后一个字符的位置。
在我们的示例中，expression 语句从第 5 行第 3 列开始（行中的前两个字符是制表符，每一个字符计为一个字符），最后在第 8 行第 4 列结束。
类文件定义了以下成员谓词：

- getFullName 返回文件的完全限定名。
- getRelativePath 返回文件相对于源代码基目录的路径。
- getExtension 返回文件的扩展名。
- getShortName 返回文件的基名称，不带扩展名。

在我们的示例中，假设文件 A.java 位于目录/home/testuser/code/pkg，其中/home/testuser/code 是所分析程序的基本目录。然后，a.java 的 File 对象返回：

- getFullName 是/home/testuser/code/pkg/A.java。
- getRelativePath 是 pkg/A.java。
- getExtension 是 java。
- getShortName 是一个。

确定运算符周围的空白

让我们从考虑如何编写一个谓词来计算给定二进制表达式的运算符周围的空白总量。如果 rcol 是表达式右操作数的起始列，而 lcol 是其左操作数的结束列，那么 rcol-（lcol+1）将给出两个操作数之间的字符总数（注意，我们必须使用 lcol+1 而不是 lcol，因为结束位置是包含的）。
这个数字包括运算符本身的长度，我们需要减去它。为此，我们可以使用谓词 getOp，它返回运算符字符串，两边各有一个空格。总体而言，用于计算二进制表达式表达式的运算符周围的空白量的表达式为：

```
rcol - (lcol+1) - (expr.getOp().length()-2)
```

但是，显然，只有当整个表达式在一行上时，这才有效，我们可以使用上面介绍的谓词 getTotalNumberOfLines 来检查这一行。我们现在可以定义谓词来计算运算符周围的空白：

```
int operatorWS(BinaryExpr expr) {

    exists(int lcol, int rcol |

        expr.getNumberOfLinesOfCode() = 1 and

        lcol = expr.getLeftOperand().getLocation().getEndColumn()
and

        rcol = expr.getRightOperand().getLocation().getStartColumn()
and

        result = rcol - (lcol+1) - (expr.getOp().length()-2)

    )

}
```

注意，我们使用 exists 来引入临时变量 lcol 和 rcol。只需将 lcol 和 rcol 内联到它们的使用中，就可以编写不带它们的谓词，但在可读性方面会有所损失。

找到可疑的巢穴

以下是我们查询的第一个版本：

```
import java


// Insert predicate defined above


from BinaryExpr outer, BinaryExpr inner,

    int wsouter, int wsinner

where inner = outer.getAChildExpr() and

    wsinner = operatorWS(inner) and wsouter = operatorWS(outer) and

    wsinner > wsouter
```

```
select outer, "Whitespace around nested operators contradicts
precedence."
```

➤请在 LGTM.com 网站. 此查询可能会找到大多数项目的结果。

where 子句的第一个连接词将 inner 限制为 outer 的操作数，第二个连接词绑定 wsinner 和 wsouter，而最后一个连接词选择可疑案例。
一开始，我们可能会想写 inner=外部。getAnOperand（）在第一个连接处。然而，这并不完全正确：getAnOperand 从其结果中去掉了任何周围的括号，这通常很有用，但不是我们这里想要的：如果内部表达式周围有圆括号，那么程序员可能知道他们在做什么，查询不应该标记这个表达式。

改进查询

如果我们运行这个初始查询，我们可能会注意到一些由不对称空白引起的误报。例如，以下表达式被标记为可疑，但在实践中不太可能引起混淆：

```
i< start + 100
```

注意，我们的谓词运算符 ws 计算运算符周围的空白总量，在本例中，<为 1，+为 2。理想情况下，我们希望排除运算符前后空白量不同的情况。目前，CodeQL 数据库没有记录足够的信息来解决这个问题，但是作为一个近似值，我们可以要求空白字符的总数是偶数：

```
import java


// Insert predicate definition from above


from BinaryExpr outer, BinaryExpr inner,
    int wsouter, int wsinner
where inner = outer.getAChildExpr() and
    wsinner = operatorWS(inner) and wsouter = operatorWS(outer) and
    wsinner % 2 = 0 and wsouter % 2 = 0 and
    wsinner > wsouter
select outer, "Whitespace around nested operators contradicts
precedence."
```

➤请在 LGTM.com 网站. 任何结果都将通过我们对查询的更改进行优化。

另一个误报的来源是关联运算符：在 x+y+z 形式的表达式中，第一个加号在语法上嵌套在第二个加号中，因为 Java 中的+与左边相关联；因此该表达式被标记为可疑。但是由于+一开始是关联的，所以操作符的嵌套方式并不重要，所以这是错误的肯定。到排除这些情况，让我们定义一个新类，用关联运算符标识二进制表达式：

```
class AssociativeOperator extends BinaryExpr {

    AssociativeOperator() {

        this instanceof AddExpr or

        this instanceof MulExpr or

        this instanceof BitwiseExpr or

        this instanceof AndLogicalExpr or

        this instanceof OrLogicalExpr

    }

}
```

现在，我们可以扩展查询以丢弃外部表达式和内部表达式具有相同的关联运算符的结果：

```
import java


// Insert predicate and class definitions from above


from BinaryExpr inner, BinaryExpr outer, int wsouter, int wsinner

where inner = outer.getAChildExpr() and

    not (inner.getOp() = outer.getOp() and outer instanceof
AssociativeOperator) and

    wsinner = operatorWS(inner) and wsouter = operatorWS(outer) and

    wsinner % 2 = 0 and wsouter % 2 = 0 and
```

```
    wsinner > wsouter

select outer, "Whitespace around nested operators contradicts
precedence."
```

➤请在 LGTM.com 网站.

注意，我们再次使用 getOp，这次是为了确定两个二进制表达式是否具有相同的运算符。运行我们改进的查询，现在可以找到概述中描述的 Java 标准库错误。它还会在 Hadoop HBase 中标记以下可疑代码：

```
KEY_SLAVE = tmp[ i+1 % 2 ];
```

空白表示程序员打算在 0 和 1 之间切换 i，但实际上表达式被解析为 i+（1%2），这与 i+1 相同，所以 i 只是递增的。