关于 QL 语言

QL 是 CodeQL 的强大查询语言，CodeQL 用于分析代码。

关于查询语言和数据库

本节面向具有通用编程和数据库背景的用户。有关如何入门的基本介绍和信息，请参阅学习 CodeQL。
QL 是一种声明性的、面向对象的查询语言，经过优化，可以有效地分析分层数据结构，特别是表示软件构件的数据库。
数据库是有组织的数据集合。关系数据库中最常用的查询模型是关系数据库(SQL)中最常用的查询语言。
查询语言的目的是提供一个编程平台，在这个平台上，您可以询问有关存储在数据库中的信息的问题。数据库管理系统管理数据的存储和管理，并提供查询机制。查询通常引用相关的数据库实体，并指定结果必须满足的各种条件(称为谓词)。查询求值包括检查这些谓词并生成结果。好的查询语言及其实现的一些理想特性包括：

- 声明性规范–声明性规范描述结果必须满足的属性，而不是提供计算结果的过程。在数据库查询语言的上下文中，声明性规范抽象了底层数据库管理系统和查询处理技术的细节。这大大简化了查询的编写。
- 表达能力–强大的查询语言允许您编写复杂的查询。这使得这种语言具有广泛的适用性。
- 高效执行–查询可能很复杂，数据库可能非常大，因此高效地处理和执行查询对于查询语言实现至关重要。

QL 的属性

QL 的语法与 SQL 相似，但是 QL 的语义基于 Datalog，一种通常用作查询语言的声明性逻辑编程语言。这使得 QL 主要是一种逻辑语言，QL 中的所有操作都是逻辑操作。此外，QL 从 Datalog 继承递归谓词，并添加对聚合的支持，使得即使是复杂的查询也变得简洁和简单。例如，考虑一个包含人的父子关系的数据库。如果我们想找到一个人的后代数量，通常我们会：

1. 找到某个人的后代，即孩子或孩子的后代。
2. 计算使用上一步找到的子体数。

当您在 QL 中编写这个过程时，它与上面的结构非常相似。请注意，我们使用递归来查找给定人员的所有后代，并使用聚合来计算后代的数量。由于语言的声明性质，可以在不添加任何过程细节的情况下将这些步骤转换为最终查询。
QL 代码如下所示：

```
Person getADescendant(Person p) {

  resultor =p.getAChild()

  result =getADescendant(p.getAChild())

}



Int getNumberOfDescendants(Person p) {

  resultcount = (getADescendant(p))

}
```

有关 QL 的重要概念和语法结构的更多信息，请参阅各个参考主题，如表达式和递归。这些解释和示例可以帮助您理解该语言是如何工作的，以及如何编写更高级的 QL 代码。
有关 QL 语言和 QLDoc 注解的正式规范，请参见 QL 语言规范和 QLDoc 注解规范。

QL 和面向对象

面向对象是 QL 的一个重要特性。面向对象的好处是众所周知的——它增加了模块化，支持信息隐藏，并允许代码重用。QL 提供了所有这些好处而不损害其逻辑基础。这是通过定义一个简单的对象模型来实现的，其中类被建模为谓词，继承被建模为隐含。为所有支持的语言提供的库广泛使用了类和继承。

QL 和通用程序设计语言

以下是通用编程语言和 QL 在概念和功能上的一些显著区别：

- • QL 没有任何必需的特性，比如对变量的赋值或文件系统操作。
- • QL 对一组元组进行操作，查询可以看作是定义查询结果的集合操作的复杂序列。
- • QL 基于集合的语义使得处理值集合变得非常自然，而不必担心有效地存储、索引和遍历它们。
- • 在面向对象编程语言中，实例化类涉及到通过分配物理内存来保存类实例的状态来创建对象。在 QL 中，类只是描述一组已经存在的值的逻辑属性。

进一步阅读

学术参考资料还提供了 QL 及其语义的概述。有关数据库查询语言和数据日志的其他有用参考：

- 数据库理论：查询语言
- 逻辑编程与数据库书籍–亚马逊页面
- 数据库基础
- 数据日志

谓词

谓词用于描述构成 QL 程序的逻辑关系。
严格地说，谓词的计算结果是一组元组。例如，考虑以下两个谓词定义：

```
predicate isCountry(string country) {
  country = "Germany"
  or
  country = "Belgium"
  or
  country = "France"
}

predicate hasCapital(string country, string capital) {
  country = "Belgium" and capital = "Brussels"
  or
  country = "Germany" and capital = "Berlin"
  or
  country = "France" and capital = "Paris"
}
```

谓词 isCountry 是一个元组{("Belgium"),("Germany"),("法国")}的集合，而 hasCapital 是两个元组 {("Belgium","Brussels"),("Germany","Berlin"),("France","Paris")}。这些谓词的数量分别是 1 和 2。
一般来说，谓词中的所有元组都有相同数量的元素。谓词的 **arity** 是指元素的数量，不包括可能的结果变量(请参见带结果的谓词)。
QL 中有许多内置谓词。您可以在任何查询中使用这些谓词，而无需导入任何其他模块。除了这些内置谓词外，您还可以定义自己的谓词：

定义谓词

定义谓词时，应指定：

1. 关键字谓词(对于没有结果的谓词)或结果的类型(对于有结果的谓词)。
2. 谓词的名称。这是一个以小写字母开头的标识符。
3. 谓词的参数(如果有)用逗号分隔。对于每个参数，指定参数类型和参数变量的标识符。
4. 谓词体本身。这是一个用大括号括起来的逻辑公式。

注意

抽象谓词或外部谓词没有正文。要定义这样的谓词，请改用分号(；)结束谓词定义。

没有结果的谓词

这些谓词定义以关键字 predicate 开始。如果一个值满足主体中的逻辑属性，则谓词将保留该值。
例如：

```
predicate isSmall(int i) {

  i in [1 .. 9]

}
```

如果 i 是整数，那么如果 i 是小于 10 的正整数，则 `isSmall(i)`成立。

带结果的谓词

通过用结果类型替换关键字谓词，可以用 result 定义谓词。这将引入特殊变量结果。
例如：

```
int getSuccessor(int i) {

  result = i + 1 and

  i in [1 .. 9]

}
```

如果 i 是一个小于 10 的正整数，那么谓词的结果就是 i 的后继。
请注意，您可以使用与谓词的任何其他参数相同的方式使用 result。你可以用你喜欢的任何方式来表达结果和其他变量之间的关系。例如，给定一个返回 x 父级的谓词 getAParentOf(Person x)，您可以定义一个“reverse”谓词，如下所示：

```
Person getAChildOf(Person p) {

  p = getAParentOf(result)

}
```

一个谓词的每个参数值也可能有多个结果(或者根本没有结果)。例如：

```
string getANeighbor(string country) {

  country = "France" and result = "Belgium"

  or

  country = "France" and result = "Germany"

  or

  country = "Germany" and result = "Austria"

  or

  country = "Germany" and result = "Belgium"

}
```

在这种情况下：
• 谓词调用 `getANeighbor("Germany")` 返回两个结果：“`Austria`”和“`Belgium`”。
• 谓词调用 `getANeighbor("Belgium")`不返回任何结果，因为 getANeighbor 没有为“`Belgium`”定义结果。

递归谓词

QL 中的谓词可以是**递归**的。这意味着它直接或间接地依赖于自身。
例如，您可以使用递归来优化上面的示例。现在，getANeighbor 中定义的关系是不对称的，它不能捕捉到这样一个事实：如果 x 是 y 的邻居，那么 y 是 x 的邻居。捕捉这一点的一个简单方法是递归调用该谓词，如下所示：

```
string getANeighbor(string country) {
  country = "France" and result = "Belgium"
  or
  country = "France" and result = "Germany"
  or
  country = "Germany" and result = "Austria"
  or
  country = "Germany" and result = "Belgium"
  or
  country = getANeighbor(result)
}
```

现在 `getANeighbor("Belgium")` 也返回结果，即`"France"` 和 `"Germany"`。
有关递归谓词和查询的更一般的讨论，请参阅递归。

谓词的种类

谓词有三种，即非成员谓词、成员谓词和特征谓词。
非成员谓词是在类之外定义的，也就是说，它们不是任何类的成员。
有关其他类型谓词的更多信息，请参阅类主题中的特征谓词和成员谓词。
下面是一个示例，显示了每种谓词：

```
int getSuccessor(int i) {  // 1. Non-member predicate
  result = i + 1 and
  i in [1 .. 9]
}
```

```
class FavoriteNumbers extends int {

  FavoriteNumbers() {  // 2. Characteristic predicate

    this = 1 or

    this = 4 or

    this = 9

  }


  string getName() {   // 3. Member predicate for the class
`FavoriteNumbers`

    this = 1 and result = "one"

    or

    this = 4 and result = "four"

    or

    this = 9 and result = "nine"

  }

}
```

您还可以对这些谓词中的每一个进行注解。请参阅每种谓词可用的注解列表。

约束行为

谓词必须能够在有限的时间内求值，因此它所描述的集合通常不允许是无限的。换句话说，谓词只能包含有限数量的元组。
当 QL 编译器可以证明谓词包含的变量不受限制为有限个值时，它会报告一个错误。有关详细信息，请参见绑定。
以下是一些无限谓词的示例：

```
/*

  Compilation errors:

  ERROR: "i" is not bound to a value.
```

```
  ERROR: "result" is not bound to a value.
*/
int multiplyBy4(int i) {
  result = i * 4
}

/*
  Compilation error:
  ERROR: "str" is not bound to a value.
*/
predicate shortString(string str) {
  str.length() < 10
}
```

在 `multiplyBy4` 中，参数 i 声明为 int，它是一个无限类型。它用于二进制运算 *，它不绑定其操作数。result 从一开始是未绑定的，并且仍然是未绑定的，因为它用于对 i*4 进行相等检查，这也是未绑定的。
在 shortString 中，str 保持未绑定状态，因为它是用无限类型字符串声明的，而内置函数 `length()` 不绑定它。

绑定集

有时您可能想定义一个“无限谓词”，因为您只打算在一组受限的参数上使用它。在这种情况下，可以使用 `bindingset` 注解指定显式绑定集。此注解对任何类型的谓词都有效。
例如：

```
bindingset[i]
int multiplyBy4(int i) {
  result = i * 4
}
```

```
from int i

where i in [1 .. 10]

select multiplyBy4(i)
```

虽然 `multiplyBy4` 是一个无限谓词，但是上面的 QL 查询是合法的。它首先使用 bindingset 注解声明谓词 multiply4 将是有限的，前提是 i 绑定到有限数量的值。然后它在上下文中使用谓词，其中 i 被限制在范围`[1 .. 10]`.
也可以为一个谓词声明多个绑定集。这可以通过添加多个绑定集批注来完成，例如：

```
bindingset[x] bindingset[y]

predicate plusOne(int x, int y) {

  x + 1 = y

}



from int x, int y

where y = 42 and plusOne(x, y)

select x, y
```

以这种方式指定的多个绑定集彼此独立。上述示例意味着：
•如果 x 是有界的，那么 x 和 y 是有界的。
•如果 y 是有界的，那么 x 和 y 是有界的。
也就是说，`bindingset[x] bindingset[y]`与`bindingset[x, y]`不同，`bindingset[x, y]`声明 x 和 y 都必须绑定。
当您想要声明一个带有带多个输入参数的结果的谓词时，后者可能很有用。例如，以下谓词接受 string str 并将其截断为 len 个字符的最大长度：

```
bindingset[str, len]

string truncate(string str, int len) {

  if str.length() > len

  then result = str.prefix(len)

  else result = str
```

```
}
```

然后可以在 select 子句中使用此选项，例如：

```
select truncate("hello world", 5)
```

数据库谓词

查询的每个数据库都包含表示值之间关系的表。这些表(“数据库谓词”)的处理方式与 QL 中的其他谓词相同。
例如，如果数据库包含 persons 表，则可以编写 `persons(x, firstName, _, age)`将 `x`, `firstName`, 和 age 约束为该表中行的第一列、第二列和第四列。
唯一的区别是不能在 QL 中定义数据库谓词，它们是由底层数据库定义的。因此，可用的数据库谓词因查询的数据库而异。

查询

查询是 QL 程序的输出。他们对结果进行评估。
有两种查询。对于给定的查询模块，该模块中的查询包括：

- 在该模块中定义的 select 子句(如果有)。
- 该模块的谓词命名空间中的任何查询谓词。也就是说，它们可以在模块本身中定义，也可以从其他模块导入。

我们还经常将整个 QL 程序称为查询。

选择子句

在编写查询模块时，可以包含一个 select 子句(通常在文件末尾)，格式如下：

```
from /* ... variable declarations ... */

where /* ... logical formula ... */

select /* ... expressions ... */
```

from 和 where 部分是可选的。

除了表达式中描述的表达式外，还可以包括：
• as 关键字，后跟一个名称。这为结果列提供了一个“标签”，并允许您在后续的 select 表达式中使用它们。
• 按 order by 关键字，后跟结果列的名称，以及可选的关键字 asc 或者 desc。这决定了显示结果的顺序。
例如：

```
from int x, int y

where x = 3 and y in [0 .. 2]

select x, y, x * y as product, "product: " + product
```

此 select 子句返回以下结果：

x	y	product	
3	0	0	product: 0
3	1	3	product: 3
3	2	6	product: 6

您还可以在 select 子句的末尾添加 `order by y desc` 。现在，将根据 y 列中的值按降序对结果进行排序：

x	y	product	
3	2	6	product: 6
3	1	3	product: 3
3	0	0	product: 0

查询谓词

查询谓词是带有查询注解的非成员谓词。它返回谓词计算到的所有元组。
例如：

```
query int getProduct(int x, int y) {

  x = 3 and

  y in [0 .. 2] and

  result = x * y

}
```

此谓词返回以下结果：

x	y	result
3	0	0
3	1	3
3	2	6

编写查询谓词而不是 select 子句的好处是可以在代码的其他部分调用谓词。例如，可以在类的主体内调用 getProduct：

```
class MultipleOfThree extends int {

  MultipleOfThree() { this = getProduct(_, _) }

}
```

相反，select 子句类似于一个匿名谓词，因此以后不能调用它。
在调试代码时，向谓词添加 query 注解也很有帮助。这样就可以显式地看到谓词计算到的元组集。

类型

QL 是一种静态类型语言，因此每个变量都必须有一个声明的类型。
类型是一组值。例如，int 类型是一组整数。请注意，一个值可以属于这些集合中的多个，这意味着它可以有多个类型。
QL 中的类型有 primitive types, classes, character types, class domain types, algebraic datatypes, 和 database types.

基本类型

这些类型内置于 QL 中，并且在全局命名空间中始终可用，与查询的数据库无关。

1. **boolean**：此类型包含值是的和假.
2. **float**：此类型包含 64 位浮点数字，例如 `6.28` 和 `-0.618`.
3. **int**：此类型包含 32 位二补整数，例如 `-1` 和 `42`.
4. **string**：此类型包含 16 位字符的有限字符串。

5. **date**：此类型包含日期(和可选的时间)。

QL 在基元类型上定义了一系列内置操作。通过对适当类型的表达式使用 dispatch 可以获得这些属性。例如，`1.toString()` 是整数常量 1 的字符串表示。有关 QL 中可用的内置操作的完整列表，请参阅 QL 语言规范中有关内置操作的部分。

类

您可以在 QL 中定义自己的类型。一种方法是定义一个类。

类提供了重用和构造代码的简单方法。例如，您可以：

- 将相关值分组。
- 根据这些值定义成员谓词。
- 定义重写成员谓词的子类。

QL 中的类不会“创建”一个新对象，它只是表示一个逻辑属性。如果某个值满足某个逻辑属性，则该值在该类中。

定义类

要定义类，请编写：

1. 关键字 class。
2. 类的名称。这是一个以大写字母开头的标识符。
3. 要扩展的类型。
4. 类的主体，用大括号括起来。

例如：

```
class OneTwoThree extends int {

  OneTwoThree() { // characteristic predicate

    this = 1 or this = 2 or this = 3

  }


  string getAString() { // member predicate

    result = "One, two or three: " + this.toString()

  }


  predicate isEven() { // member predicate
```

```
    this = 2

  }

}
```

这定义了一个类 OneTwoThree，它包含值 1、2 和 3。特征谓词捕获“是整数 1、2 或 3 之一”的逻辑属性

`OneTwoThree` 继承 `int`，也就是说，它是 `int` 的子类型。QL 中的类必须始终至少扩展一个现有类型。这些类型称为**类的基类型**。类的值包含在基类型的交集中(即，它们在类域类型中)。类从其基类型继承所有成员谓词。

一个类可以扩展多个类型。参见下面的多重继承。

为了有效，一个类：

- 不得自我继承。
- 不能继承 final 类。
- 不能继承不兼容的类型。(请参见类型兼容性。)

您还可以为类添加注解。请参阅可用于类的注解列表。

类主体

类的主体可以包含：

- 特征谓词声明。
- 任意数量的成员谓词声明。
- 任何数量的字段声明。

当定义一个类时，该类还从其超类型继承所有非私有成员谓词和字段。您可以重写这些谓词和字段，为它们提供更具体的定义。

特征谓词

它们是在类的主体中定义的谓词。它们是使用变量 this 限制类中可能的值的逻辑属性。

成员谓词

这些谓词只适用于特定类的成员。可以对值调用成员谓词。例如，可以使用上述类中的成员谓词：

```
1.(OneTwoThree).getAString()
```

此调用返回结果`"One, two or three: 1"`。
表达式`(OneTwoThree)` 是一个类型转换(cast)。它确保 1 的类型是 OneTwoThree，而不仅仅是 int。因此，它可以访问成员谓词 `getAString()`.
成员谓词特别有用，因为可以将它们链接在一起。例如，可以使用 `toUpperCase()`，它是为字符串定义的内置函数：

```
1.(OneTwoThree).getAString().toUpperCase()
```

此调用返回 `"ONE, TWO OR THREE: 1"`.

注意

特征谓词和成员谓词通常使用变量 this。这个变量总是引用类的一个成员，在这种情况下是属于类 OneTwoThree 的值。类中的特征值是该类中的特征值。在成员谓词中，这与谓词的任何其他参数的作用方式相同。

Fields

这些是在类的主体中声明的变量。一个类的主体中可以有任意数量的字段声明(即变量声明)。可以在类内部的谓词声明中使用这些变量。与变量 this 非常相似，字段必须在特征谓词中受到约束。
例如：

```
class SmallInt extends int {

  SmallInt() { this = [1 .. 10] }

}


class DivisibleInt extends SmallInt {

  SmallInt divisor;   // declaration of the field `divisor`

  DivisibleInt() { this % divisor = 0 }


  SmallInt getADivisor() { result = divisor }

}
```

```
from DivisibleInt i

select i, i.getADivisor()
```

在本例中，声明 SmallInt divisor 引入一个字段 `divisor`，将其约束在特征谓词中，然后在成员谓词 getADivisor 的声明中使用它。这类似于通过在 from 部分声明变量来在 select 子句中引入变量。
您还可以注解谓词和字段。请参见可用注解的列表。

体类(实体类,具类)

上面例子中的类都是具体类。它们是通过限制较大类型中的值来定义的。具体类中的值正是基类型交集中的值，这些值也满足类的特征谓词。

抽象类

用 abstract 注解的类，称为抽象类，也是对较大类型中的值的限制。然而，抽象类被定义为其子类的并集。特别是，对于一个抽象类中的值，它必须满足类本身的特征谓词和子类的特征谓词。
如果要将多个现有类组合在一个公共名称下，则抽象类非常有用。然后可以在所有这些类上定义成员谓词。还可以扩展预定义的抽象类：例如，如果导入包含抽象类的库，则可以向其中添加更多子类。

例子
如果您正在编写一个安全查询，您可能会对标识所有可以解释为 SQL 查询的表达式感兴趣。您可以使用以下抽象类来描述这些表达式：

```
abstract class SqlExpr extends Expr {

  ...

}
```

现在为每种数据库管理系统定义不同的子类。例如，可以定义子类 `class postgresqlexpr extends SqlExpr`，它包含传递给执行数据库查询的某些 postgres api 的表达式。您可以为 MySQL 和其他数据库管理系统定义类似的子类。
抽象类 SqlExpr 引用所有这些不同的表达式。如果以后想添加对另一个数据库系统的支持，只需向 SqlExpr 添加一个新的子类；不需要更新依赖它的查询。
重要的

向现有抽象类添加新子类时必须小心。添加子类不是孤立的更改，它还扩展了抽象类，因为抽象类是其子类的联合。

重写成员谓词(override)

如果类从父类型继承成员谓词，则可以重写(override)继承的定义。为此，可以定义一个与继承谓词具有相同名称和 arity 的成员谓词，并添加 override 注解。如果您想优化谓词，为子类中的值提供更具体的结果，这很有用。
例如，从第一个示例扩展类：

```
class OneTwo extends OneTwoThree {

  OneTwo() {

    this = 1 or this = 2

  }



  override string getAString() {

    result = "One or two: " + this.toString()

  }

}
```

成员谓词 `getAString()` 重写 OneTwoThree 中 `getAString()`的原始定义。
现在，考虑以下查询：

```
from OneTwoThree o

select o, o.getAString()
```

查询使用谓词 `getAString()`,的“最具体”定义，因此结果如下所示：

o	`getAString()` result
1	One or two: 1

o	`getAString()` result
2	One or two: 2
3	One, two or three: 3

在 QL 中，与其他面向对象语言不同，相同类型的不同子类型不需要分离。例如，您可以定义 OneTwoThree 的另一个子类，它与 OneTwo 重叠：

```
class TwoThree extends OneTwoThree {

  TwoThree() {

    this = 2 or this = 3

  }



  override string getAString() {

    result = "Two or three: " + this.toString()

  }

}
```

现在值 2 包含在 OneTwo 和 TwoThree 两个类类型中。这两个类都重写了`getAString()`的原始定义。有两个新的“最具体”定义，因此运行上述查询会得到以下结果：

o	`getAString()` result
1	One or two: 1
2	One or two: 2
2	Two or three: 2
3	Two or three: 3

多重继承

一个类可以扩展多个类型。在这种情况下，它继承了所有这些类型。
例如，使用上述部分的定义：

```
class Two extends OneTwo, TwoThree {}
```

类 `Two` 中的任何值都必须满足 OneTwo 表示的逻辑属性和 TwoThree 表示的逻辑属性。这里，类 `Two` 包含一个值，即 2。
它从 OneTwo 和 `TwoThree` 继承成员谓词。它还(间接)继承了 `OneTwoThree` 和 int。

注意

如果一个子类继承了同一谓词名称的多个定义，那么它必须重写这些定义以避免歧义。在这种情况下，超级表达式通常很有用。

字符类型和类域类型

不能直接引用这些类型，但是 QL 中的每个类都隐式定义了一个字符类型和一个类域类型。(这些都是比较微妙的概念，在实际的查询写作中并不经常出现。)
QL 类的字符类型是满足类的特征谓词的一组值。它是域类型的子集。对于具体的类，值只有在字符类型中才属于该类。对于抽象类，值除了属于字符类型外，还必须至少属于其中一个子类。
QL 类的域类型是其所有超类型的字符类型的交集，也就是说，如果一个值属于每个超类型，则该值属于该域类型。它以 this 类型出现在类的特征谓词中。

代数数据类型

注意

代数数据类型的语法被认为是实验性的，可能会发生变化。但是，它们出现在标准的 QL 库中，因此下面的部分将帮助您理解这些示例。

代数数据类型是用户定义类型的另一种形式，用关键字 newtype 声明。
代数数据类型用于从数据库中创建既不是原始值也不是实体的新值。一个例子是在分析程序中的数据流时对流节点进行建模。
代数数据类型由许多相互不相交的分支组成，每个分支定义一个分支类型。代数数据类型本身是所有分支类型的并集。分支可以有参数和主体。对于满足参数类型和主体的每一组值，将生成分支类型的新值。
这样做的好处是每个分支可以有不同的结构。例如，如果您想定义一个“option type”，它要么包含一个值(比如一个调用(Call))，要么是空的，可以这样写：

```
newtype OptionCall = SomeCall(Call c) or NoCall()
```

这意味着对于程序中的每个调用(call)，都会生成一个不同的 SomeCall 值。这也意味着产生了一个独特的 NoCall 值。

定义代数数据类型

要定义代数数据类型，请使用以下常规语法：

```
newtype <TypeName> = <branches>
```

分支定义的形式如下：

```
<BranchName>(<arguments>){<body>}
```

- 类型名称和分支名称必须是以大写字母开头的标识符。按照惯例，它们从T开始。
- 代数数据类型的不同分支用 `or` 分隔。
- 分支的参数(如果有)是用逗号分隔的变量声明。
- 分支的主体是谓词体。可以省略分支体，在这种情况下，它默认为 `any()`.请注意，分支体是完全求值的，因此它们必须是有限的。为了有好的表现，它们应该少一点(keep small)。

例如，以下代数数据类型有三个分支：

```
newtype T =

  Type1(A a, B b) { body(a, b) }

  or

  Type2(C c)

  or

  Type3()
```

使用代数数据类型的标准模式

代数数据类型不同于类。特别是，代数数据类型没有 `toString()`成员谓词，因此不能在 select 子句中使用它们。
类通常用于扩展代数数据类型(并提供 `toString()`谓词)。在标准的 QL 语言库中，这通常是按如下方式完成的：

- 定义一个扩展代数数据类型并可选地声明抽象谓词的类 a。
- 对于每个分支类型，定义一个扩展 a 和分支类型的类 B，并为来自 a 的任何抽象谓词提供定义。
- 用 private 注解代数数据类型，并将类保留为 public。

例如，以下来自 C#的 CodeQL 数据流库的代码片段定义了用于处理受污染或未着色值的类。在这种情况下，`TaintType` 扩展数据库类型是没有意义的。它是污点分析的一部分，而不是底层程序，因此扩展一个新类型(即 `TTaintType`)很有帮助：

```
private newtype TTaintType =
  TExactValue()
  or
  TTaintedValue()

/** Describes how data is tainted. */
class TaintType extends TTaintType {
  string toString() {
    this = TExactValue() and result = "exact"
    or
    this = TTaintedValue() and result = "tainted"
  }
}

/** A taint type where the data is untainted. */
class Untainted extends TaintType, TExactValue {
}

/** A taint type where the data is tainted. */
class Tainted extends TaintType, TTaintedValue {
}
```

数据库类型

数据库类型在数据库架构中定义。这意味着它们依赖于您正在查询的数据库，并且根据您正在分析的数据而变化。

例如，如果要查询 Java 项目的 CodeQL 数据库，则数据库类型可能包括 @ifstmt(表示 Java 代码中的 if 语句)和@variable(表示变量)。

类型的兼容性

并非所有类型都兼容。例如，`4 < "five"`没有意义，因为您无法将 int 与 string 进行比较。
为了确定类型何时兼容，QL 中有许多不同的“类型通用体”。
QL 中的宇宙是：

- 每个基元类型一个(除了 int 和 float，它们在同一个“数字”世界中)。
- 每个数据库类型一个。
- 一个代数数据类型的每个分支一个。

例如，在定义类时，这会导致以下限制：

- 一个类不能扩展多个基元类型。
- 一个类不能扩展多个不同的数据库类型。
- 类不能扩展一个代数数据类型的多个不同分支。

模块

模块提供了一种通过将相关类型、谓词和其他模块组合在一起来组织 QL 代码的方法。
您可以将模块导入到其他文件中，这样可以避免重复，并有助于将代码组织成更易于管理的部分。

定义模块

定义模块有多种方法这里是一个最简单的方法示例，它声明一个名为 example 的显式模块，其中包含一个 `OneTwoThree` 类：

```
module Example {

  class OneTwoThree extends int {

    OneTwoThree() {
```

```
      this = 1 or this = 2 or this = 3

    }

  }

}
```

模块的名称可以是任何以大写或小写字母开头的标识符。
`.ql` 或 `.qll` 文件也隐式定义模块。阅读更多有关以下不同类型模块的信息。
您还可以对模块进行注解。请参阅可用于模块的注解列表。
请注意，您只能注解显式模块。无法对文件模块进行批注。

模块种类

文件模块

每个查询文件(.ql)和库文件(.qll)都隐式定义一个模块。模块的名称与文件相同，但文件名中的任何空格都将替换为下划线(_)文件的内容构成模块的主体。

库模块

库模块由.qll 文件定义。除了 select 子句外，它可以包含下面模块主体中列出的任何元素。
例如，考虑以下 QL 库：

OneTwoThreeLib.qll

```
class OneTwoThree extends int {

  OneTwoThree() {

    this = 1 or this = 2 or this = 3

  }

}
```

这个文件定义了一个名为 `OneTwoThreeLib` 的库模块。这个模块的主体定义了 OneTwoThree 类。

查询模块

查询模块由.ql 文件定义。它可以包含下面模块主体中列出的任何元素。
查询模块与其他模块略有不同：

·无法导入查询模块。

·查询模块的命名空间中必须至少有一个查询。这通常是一个 select 子句，但也可以是一个查询谓词。

例如：

OneTwoQuery.ql

```
import OneTwoThreeLib


from OneTwoThree ott

where ott = 1 or ott = 2

select ott
```

这个文件定义了一个名为 OneTwoQuery 的查询模块。这个模块的主体由 import 语句和 select 子句组成。

显式模块

也可以在另一个模块中定义模块。这是一个明确的模块定义。
显式模块定义为关键字 module 后跟模块名，然后将模块主体括在大括号中。
除了 select 子句外，它可以包含下面模块主体中列出的任何元素。
例如，可以将以下 QL 片段添加到库文件中 OneTwoThreeLib.qll 上述定义：

```
...

module M {

  class OneTwo extends OneTwoThree {

    OneTwo() {

      this = 1 or this = 2

    }

  }
```

```
}
```

这定义了一个名为 M 的显式模块。该模块的主体定义了类 OneTwo。

模块主体

模块的主体是模块定义中的代码，例如显式模块 M 中的 OneTwo 类。
通常，模块主体可以包含以下结构：

- 导入语句
- 谓词
- 类型(包括用户定义的类)
- 别名
- 显式模块
- Select 子句(仅在查询模块中可用)

导入模块

在模块中存储代码的主要好处是可以在其他模块中重用它。要访问外部模块的内容，可以使用 import 语句导入该模块。
当您导入一个模块时，这会将其命名空间中的所有名称(私有名称除外)引入当前模块的命名空间中。

导入语句

导入语句用于导入模块。它们的形式是：

```
import <module_expression1> as <name>

import <module_expression2>
```

导入语句通常列在模块的开头。每个 import 语句导入一个模块。您可以通过包含多个 import 语句来导入多个模块(要导入的每个模块对应一个语句)。import 语句也可以用 private 进行注解。
可以使用 as 关键字以其他名称导入模块，例如 `import javascriptas js` `<module_expression>` 本身可以是模块名、选择或限定引用。有关详细信息，请参见名称解析。
有关如何查找导入语句的信息，请参阅 QL 语言规范中的模块解析。

别名

别名是现有 QL 实体的另一个名称。
一旦定义了别名，就可以使用该新名称来引用当前模块命名空间中的实体。

定义别名

您可以在任何模块的主体中定义别名。为此，应指定：

1. 关键字 module、class 或 predicate 分别为 module、type 或 non-member 谓词定义别名。
2. 别名的名称。这应该是此类实体的有效名称。例如，有效的谓词别名以小写字母开头。
3. 对 QL 实体的引用。这包括实体的原始名称，对于谓词，还包括谓词的 arity。

也可以为别名添加注解。请参见别名可用的注解列表。
请注意，这些注解适用于别名引入的名称(而不是底层 QL 实体本身)。例如，别名可以对其别名的名称具有不同的可见性。

模块别名

使用以下语法为模块定义别名：

```
module ModAlias = ModuleName;
```

例如，如果创建的新模块 NewVersion 是 OldVersion 的更新版本，则可以使用名称 OldVersion，如下所示：

```
deprecated module OldVersion = NewVersion;
```

这样，两个名称都会解析为同一个模块，但如果使用名称 OldVersion，则会显示一个弃用警告。

类型别名

用于定义别名的类型语法：

```
class TypeAlias = TypeName;
```

注意 class 只是一个关键字。您可以为任何类型(即基元类型、数据库类型和用户定义类)定义别名。
例如，可以使用别名将基本类型 boolean 的名称缩写为 bool：

```
class bool = boolean;
```

或者，使用模块 M 中定义的一个类 OneTwo `OneTwoThreeLib.qll`，您可以创建一个别名来使用较短的名称 `OT` 替代：

```
import OneTwoThreeLib

class OT = M::OneTwo;

...

from OT ot

select ot
```

谓词别名

使用以下语法为非成员谓词定义别名：

```
predicate PredAlias = PredicateName/Arity;
```

这适用于有或无结果的谓词。
例如，假设您经常使用以下谓词，它计算小于 10 的正整数的后继数：

```
int getSuccessor(int i) {

  result = i + 1 and

  i in [1 .. 9]

}
```

您可以使用别名将名称缩写为 `succ`：

```
predicate succ = getSuccessor/1;
```

作为一个没有结果的谓词的例子，假设您有一个谓词，它可以容纳任何小于 10 的正整数：

```
predicate isSmall(int i) {

  i in [1 .. 9]

}
```

可以为谓词指定一个更具描述性的名称，如下所示：

```
predicate lessThanTen = isSmall/1;
```

变量

QL 中变量的使用方式与代数或逻辑中的变量类似。它们表示一组值，这些值通常受公式的限制。
这与其他一些编程语言中的变量不同，变量表示可能包含数据的内存位置。这些数据也会随着时间的推移而改变。例如，在 QL 中，n=n+1 是一个等式公式，它只在 n 等于 n+1 时成立(所以实际上它不适用于任何数值)。在 Java 中，n=n+1 不是一个等式，而是通过在当前值上加 1 来改变 n 的值的赋值。

声明变量

所有变量声明都由变量的类型和名称组成。名称可以是以大写或小写字母开头的任何标识符。
例如，`int i`, `SsaDefinitionNode` 节点和 `LocalScopeVariable lsv` 分别声明类型为 `int`, `SsaDefinitionNode`，和 `LocalScopeVariable` 的变量 i、node 和 lsv。
变量声明出现在不同的上下文中，例如在 select 子句中，在量化公式中，作为谓词的参数，等等。
从概念上讲，可以将变量视为包含其类型允许的所有值，并受任何进一步的约束。
例如，考虑下面的 select 子句：

```
from int i

where i in [0 .. 9]

select i
```

根据它的类型，变量 i 可以包含所有整数。但是，它受到公式 `i in [0 .. 9]` 。因此，select 子句的结果是 0 到 9 之间的 10 个数字(包括 0 和 9)。
另外，请注意，以下查询会导致编译时错误：

```
from int i

select i
```

理论上，它会有无穷多的结果，因为变量 i 不受有限个可能值的约束。有关详细信息，请参见绑定。

自由变量和约束变量

变量可以有不同的角色。有些变量是自由的，它们的值直接影响使用它们的表达式的值，或者使用它们的公式是否成立。其他变量，称为绑定变量，仅限于特定的值集。
在一个例子中可能最容易理解这种区别。请看以下表达式：

```
"hello".indexOf("l")

min(float f | f in [-3 .. 3])

(i + 7) * 3

x.sqrt()
```

第一个表达式没有任何变量。它查找字符串“hello”中“l”出现的位置(从零开始)的索引，因此计算结果为 2 和 3。
第二个表达式的计算结果为-3，即范围`[-3 .. 3]` .尽管此表达式使用变量 f，但它只是一个占位符或“伪”变量，不能为其赋值。您可以在不改变表达式含义的情况下用其他变量替换 f。例如，`min(float f | f in [-3 .. 3])` 始终等于 `min(float other | other in [-3 ..3])`.这是一个**绑定变量的**例子。
表达式`(i + 7) * 3` and `x.sqrt()`? 在这两种情况下，表达式的值取决于分别分配给变量 i 和 x 的值。换句话说，变量的值对表达式的值有影响。这些是**自由变量**的例子。
类似地，如果一个公式包含自由变量，那么公式可以保留还是不保留取决于分配给这些变量的值[1]。例如：

```
"hello".indexOf("l") = 1

min(float f | f in [-3 .. 3]) = -3

(i + 7) * 3 instanceof int
```

```
exists(float y | x.sqrt() = y)
```

第一个公式不包含任何变量，并且它永远不会保持不变(因为`"hello".indexOf("l")`的值是 2 和 3，而不是 1)。
第二个公式只包含一个绑定变量，因此不受该变量更改的影响。从`min(float f | f in [-3 .. 3])`等于-3，此公式始终成立。

第三个公式包含一个自由变量 i。该公式是否成立取决于为 i 赋值的值。例如，如果 i 被赋值为 1 或 2(或任何其他 int)，则该公式成立。另一方面，如果我被分配到 3.5，那么它就不成立了。
最后一个公式包含一个自由变量 x 和一个约束变量 y。如果 x 被赋予一个非负数，则最后一个公式成立。另一方面，如果 x 被赋值为-9，那么公式就不成立了。变量 y 不影响公式是否成立。
有关如何计算对自由变量的赋值的更多信息，请参见 QL 程序的求值。

脚注

[1]	这只是一个小小的简化。有些公式总是正确的或总是错误的，不管它们的自由变量的赋值是什么。编写 QL 时通常不会使用这些参数，例如，a=a 始终为真(称为重言式)，而 x 和 not x 始终为 fals

表达式

表达式的计算结果为一组值并具有类型。
例如，表达式 1+2 的计算结果为整数 3，表达式“QL”的计算结果为字符串`"QL"`
1+2 有 int 类型，`"QL"` 有 string 类型。
以下部分描述了 QL 中可用的表达式。

变量引用

变量引用是声明变量的名称。这种表达式与它所引用的变量具有相同的类型。
例如，如果您声明了变量 int i 和 LocalScopeVariable lsv，那么表达式 i 和 lsv 分别具有 int 和 LocalScopeVariable 类型。
你也可以引用变量 this 和 result。它们在谓词定义中使用，其作用方式与其他变量引用相同。

直接常量

您可以直接在 QL 中表示某些值，例如数字、布尔值和字符串。

·布尔值：这些值是 true 和 false。

·整数字面量：这些是十进制数字序列(0 到 9)，可能以减号(-)开头。例如：

```
0
42
-2048
```

•浮点字面值：由点(.)分隔的十进制数字序列，可能以减号(–)开头。例如：

```
2.0
123.456
-100.5
```

•字符串文字：这些是 16 位字符的有限字符串。您可以通过在引号("...").中包含字符来定义字符串文本。大多数字符表示自己，但也有一些字符需要用反斜杠“转义”。以下是字符串文字的示例：

```
"hello"
"They said, \"Please escape quotation marks!\""
```

有关详细信息，请参阅 QL 语言规范中的字符串文本。

注意：QL 中没有“date literal”。相反，要指定日期，应该使用 `toDate()` 谓词将字符串转换为它所表示的日期。例如，“2016-04-03”. `toDate()` 是 2016 年 4 月 3 日的日期，`"2000-01-01 00:00:01".toDate()`是 2000 年新年后一秒钟的时间点。

以下字符串格式被识别为日期：

- o，如“2016-04-03 17:00:24”。秒部分是可选的(如果丢失，则假定为“00”)，整个时间部分也可能丢失(在这种情况下，假定为“00:00:00”)。**ISO 日期**
- o，如“20160403”。**简写 ISO 日期**
- o，如“03/04/2016”。**英式日期**
- o，如`"03 April 2016"`.。**冗长的日期**

带圆括号的表达式

带圆括号的表达式是由圆括号(和)包围的表达式。此表达式与原始表达式具有完全相同的类型和值。圆括号用于将表达式分组在一起，以消除歧义并提高可读性。

范围

范围表达式表示在两个表达式之间排序的值的范围。它由两个用..分隔的表达式组成并用括号([和]).括起来。例如，`[3 .. 7]` 是有效的范围表达式。它的值是 3 到 7 之间的任何整数(包括 3 和 7 本身)。
在有效范围内，开始和结束表达式是整数、浮点或日期。如果其中一个是日期，那么两个都必须是日期。如果其中一个是整数，另一个是浮点，则两者都被视为浮点。

设置文本表达式

set-literal 表达式允许显式列出多个值之间的选择。它由逗号分隔的表达式集合组成，这些表达式用括号([和])括起来。例如，
`[2, 3,5, 7, 11, 13, 17, 19, 23, 29]` 是有效的 set-literal 表达式。它的值是前十个质数。
对于有效的 set-literal 表达式，包含的表达式的值必须是兼容类型。此外，至少有一个集合元素的类型必须是所有其他包含表达式类型的超类型。
codeQL cli 的 2.1.0 版和 LGTM Enterprise 的 1.24 版都支持 Set-literals。

超级表达式

QL 中的超级表达式类似于其他编程语言(如 Java)中的超级表达式。当您想使用超类型中的谓词定义时，可以在谓词调用中使用它们。实际上，当谓词从其超类型继承两个定义时，这是很有用的。在这种情况下，谓词必须重写这些定义以避免歧义。但是，如果要使用特定超类型的定义而不是编写新定义，则可以使用超级表达式。
在下面的示例中，类 C 继承谓词 getANumber()的两个定义-一个来自 A，另一个来自 B。它不重写这两个定义，而是使用来自 B 的定义。

```
class A extends int {

  A() { this = 1 }

  int getANumber() { result = 2 }

}



class B extends int {

  B() { this = 1 }
```

```
  int getANumber() { result = 3 }
}

class C extends A, B {
  // Need to define `int getANumber()`; otherwise it would be
ambiguous
  int getANumber() {
    result = B.super.getANumber()
  }
}

from C c
select c, c.getANumber()
```

此查询的结果是 1，3。

调用谓词(带结果)

对具有结果的谓词的调用本身是表达式，而对没有结果的谓词的调用是公式。
有关调用的更多一般信息，请参阅“公式”主题中的“调用谓词”。
对带有 result 的谓词的调用将计算被调用谓词的 result 变量的值。
例如，a.getAChild()是对变量 a 的谓词 getAChild()的调用。此调用的计算结果是 a 的一组子集。

聚合

聚合是一种映射，它根据由公式指定的一组输入值计算结果值。
一般语法是：

```
<aggregate>(<variable declarations><formula><expression>)
```

<variable declarations>中声明的变量称为聚合变量。

默认情况下，有序聚合(即 min、max、rank、concat 和 strictconcat)按其<expression>值排序。顺序可以是数字(对于整数和浮点数)或字典(对于字符串)。词典排序基于每个字符的 Unicode 值。
要指定不同的顺序，请在<expression>后面输入关键字 order by，然后是指定顺序的表达式，还可以选择关键字 asc 或 desc(以确定是按升序还是降序对表达式排序)。如果不指定排序，则默认为 asc。
QL 中提供了以下聚合：

- count：此聚合确定每个可能的聚合变量赋值的<expression>不同值的数量。

  例如，以下聚合返回包含 500 行以上的文件数：

```
count(File f | f.getTotalNumberOfLines() > 500 | f)
```

  如果没有满足公式的聚合变量的可能赋值，如 incount(inti | i=1 和 i=2 | i)，则 count 默认为值 0。

- 最小值和最大值：这些聚合确定聚合变量的可能赋值中<expression>的最小值(min)或最大值(max)。在这种情况下，<或字符串类型必须是数值表达式。

  例如，以下聚合返回具有最大行数的.js 文件的名称：

```
max(File f | f.getExtension() = "js" | f.getBaseName() order by
f.getTotalNumberOfLines())
```

下面的聚合返回下面提到的三个字符串中的最小字符串 s，也就是说，所有可能的 s 值的字典顺序中排在第一位的字符串(在本例中，它返回"De Morgan".)

```
min(string s | s = "Tarski" or s = "Dedekind" or s = "De Morgan"
| s)
```

- 确定该表达式中所有变量的平均值。<expression>的类型必须是数字。如果没有满足公式的聚合变量的可能赋值，则聚合失败且不返回值。换句话说，它的计算结果是空集。

  例如，以下聚合返回整数 0、1、2 和 3 的平均值：

```
avg(int i | i = [0 .. 3] | i)
```

- sum：此聚合确定<expression>值在所有可能的聚合变量赋值上的总和。<expression>的类型必须是数字。如果没有满足公式的聚合变量的可能赋值，则总和为 0。

  例如，以下聚合返回 i 和 j 的所有可能值的 i*j 之和：

```
sum(int i, int j | i = [0 .. 2] and j = [3 .. 5] | i * j)
```

- concat：此聚合将<expression>的值连接到聚合变量的所有可能赋值上。注意，<expression>必须是 string 类型。如果没有满足公式的聚合变量的可能赋值，则 concat 默认为空字符串。

例如，以下聚合返回字符串“3210”，即字符串“0”、“1”、“2”和“3”按降序连接：

```
concat(int i | i = [0 .. 3] | i.toString() order by i desc)
```

concat 聚合还可以采用第二个表达式，该表达式与第一个表达式之间用逗号分隔。第二个表达式作为每个串联值之间的分隔符插入。
例如，以下聚合返回"0|1|2|3"：

```
concat(int i | i = [0 .. 3] | i.toString(), "|")
```

• rank：这个聚合接受<expression>的可能值并对它们进行排序。在这种情况下，<expression>必须是数字类型或 string 类型。聚合返回排名表达式指定位置的值。必须在关键字 rank 后的括号中包含此排名表达式。
例如，下面的聚合返回在所有可能值中排名第四的值。在这种情况下，8 是 5 到 15 之间的第四个整数：

```
rank[4](int i | i = [5 .. 15] | i)
```

注意，rank 索引从 1 开始，所以 rank[0](...)不返回任何结果。
·strictconcat、strictcount 和 strictsum：这些聚合的工作方式分别与 concat、count 和 sum 类似，只是它们是 strict。也就是说，如果没有满足公式的聚合变量的可能赋值，则整个聚合将失败并计算为空集(而不是默认为 0 或空字符串)。如果您只对聚合体非常重要的结果感兴趣，那么这很有用。
·唯一：此聚合依赖于所有可能分配给聚合变量的<expression>值。如果聚合变量上存在唯一的<expression>值，则聚合计算为该值。否则，聚合没有值。
例如，下面的查询返回正整数 1、2、3、4、5。对于负整数 x，表达式 x 和 x.abs()具有不同的值，因此聚合表达式中 y 的值不是唯一确定的。

```
from int x

where x in [-5 .. 5] and x != 0

select unique(int y | y = x or y = x.abs() | y)
```

唯一聚合从 CodeQL cli 的 2.1.0 版和 LGTM Enterprise 的 1.24 版都支持。

聚合合计(Evaluation of aggregates)

一般来说，综合评价包括以下步骤：

1. 确定输入变量：这些变量是在<variabledeclarations>中声明的聚合变量，也是在聚合的某些组件中使用的聚合外部声明的变量。
2. 生成输入变量值的所有可能的不同元组(组合)，使<formula>为真。请注意，聚合变量的相同值可能出现在多个不同的元组中。在处理元组时，相同值的所有此类出现都被视为不同的出现。
3. 对每个元组应用<expression>并收集生成的(不同的)值。在元组上应用<expression>可能会导致生成多个值。
4. 对步骤 3 中生成的值应用聚合函数以计算最终结果。

让我们将这些步骤应用于以下查询中的 sum 合计：

```
select sum(int i, int j |

    exists(string s | s = "hello".charAt(i)) and exists(string s | s
= "world!".charAt(j)) | i)
```

1. 输入变量：i，j。
2. 满足给定条件的所有可能的元组(<value of i>，<value of j>)：(0，0)，(0，1)，(0，2)，(0，3)，(0，4)，(0，5)，(1，0)，(1，1)，…，(4，5)。在这个步骤中生成 30 个元组。
3. 对所有元组应用<expression>i。这意味着从所有元组中选择 i 的所有值：0，0，0，0，1，1，1，1，1，2，2，2，2，3，3，3，3，3，3，4，4，4，4，4。
4. 对上述值应用聚合函数 sum，得到最终结果 60。

如果我们在上面的查询中将<expression>改为 i+j，那么查询结果是 135，因为对所有元组应用 i+j 会得到以下值：0、1、2、3、4、1、2、3、3、4、5、6、2、3、4、3、4、5、7、4、6、7、8、4、5、6、7、8、9。

接下来，考虑以下查询：

```
select count(string s | s = "hello" | s.charAt(_))
```

1. `s` 是聚合的输入变量。
2. 在这个步骤中生成一个元组“hello”。
3. `<expression> charAt(_)`应用于此元组。`charAt(_)` 中的`_`下划线是一个不关心的表达式(就是统配符,`_`就和`*`一样)，它表示任何值。`s.charAt(_)` 生成四个不同的值 h、e、l、o。
4. 最后，对这些值应用 count，查询返回 4。

省略聚合的一部分

聚合的三个部分并不总是必需的，因此通常可以用更简单的形式编写聚合：

1. 如果要编写形式为<aggregate>(<type>v |<expression>=v | v)的聚合，则可以省略<variable declarations>和<formula>部分，并按如下方式编写：

```
<aggregate>(<expression>)
```

例如，下面的聚合决定了字母 l 在字符串“hello”中出现的次数。这些形式相当于：

```
count(int i | i = "hello".indexOf("l") | i)

count("hello".indexOf("l"))
```

2. 如果只有一个聚合变量，则可以省略<expression>部分。在这种情况下，表达式被认为是聚合变量本身。例如，以下聚合是等效的：

```
avg(int i | i = [0 .. 3] | i)

avg(int i | i = [0 .. 3])
```

作为一种特殊情况，即使存在多个聚合变量，也可以从 count 中省略<expression>部分。在这种情况下，它计算满足公式的聚合变量的不同元组的数量。换句话说，表达式部分被认为是常数 1。例如，以下聚合是等效的：

```
count(int i, int j | i in [1 .. 3] and j in [1 .. 3] | 1)

count(int i, int j | i in [1 .. 3] and j in [1 .. 3])
```

您可以省略<formula>部分，但在这种情况下，您应该包括两个竖线：

```
<aggregate>(<variable declarations>| |<expression>)
```

如果您不想进一步限制聚合变量，这很有用。例如，以下聚合返回所有文件的最大行数：

```
max(File f | | f.getTotalNumberOfLines())
```

最后，还可以省略<formula>和<expression>部分。例如，以下聚合是计算数据库中文件数的等效方法：

```
count(File f | any() | 1)

count(File f | | 1)

count(File f)
```

单调聚集体

除了标准聚合之外，QL 还支持单调聚合。单调聚合与标准聚集的不同之处在于，它们处理由公式的<expression>部分生成的值：

·标准聚合为每个<formula>值获取<expression>值并将其展平到一个列表中。单个聚合函数应用于所有值。
• 单调聚合对<formula>给出的每个值取一个<expression>，并创建所有可能值的组合。聚合函数应用于每个结果组合。

一般来说，如果<expression>是全函数的，那么单调聚集体与标准聚集体是等价的。当由<公式>生成的每个值没有精确的<expression>值时，结果会有所不同：

• 如果缺少<expression>值(即对于由<formula>生成的值没有<expression>值)，单调聚合将不会计算结果，因为您无法为由<formula>生成的每个值创建包含一个<expression>值的值的组合。
• 如果每个<formula>结果有多个<expression>，则可以为<formula>生成的每个值创建多个值组合，其中只包含一个<expression>值。这里，聚合函数应用于每个结果组合。

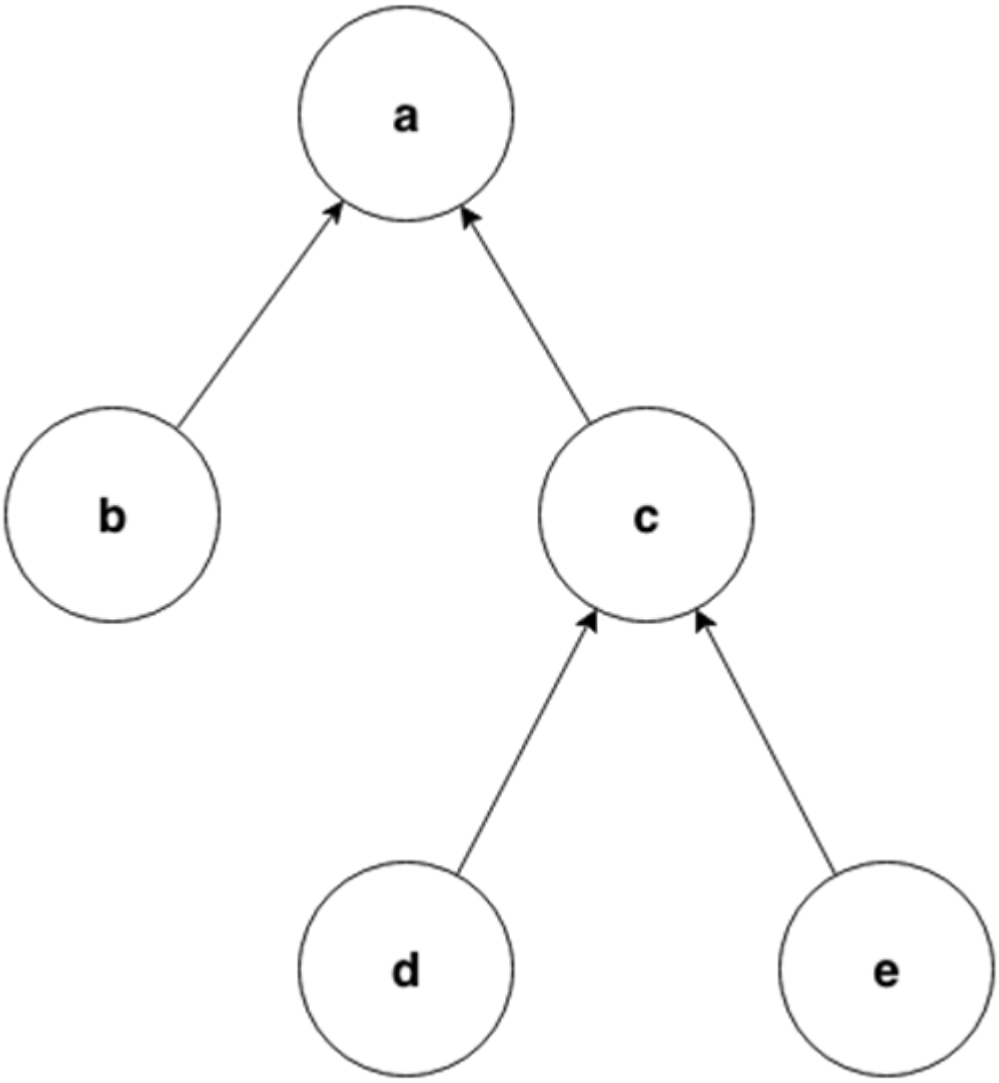
递归单调集

单调聚合可以递归使用，但递归调用只能出现在表达式中，而不能出现在范围中。聚合的递归语义与 QL 其余部分的递归语义相同。例如，我们可以定义一个谓词来计算图中节点与叶之间的距离，如下所示：

```
int depth(Node n) {

  if not exists(n.getAChild())

  then result = 0

  else result = 1 + max(Node child | child = n.getAChild() |
depth(child))

}
```

这里递归调用在表达式中，这是合法的。聚合的递归语义与 QL 其余部分的递归语义相同。如果您了解聚合在非递归情况下的工作原理，那么您应该不会发现递归使用它们很困难。但是，值得一看递归聚合的计算是如何进行的。

考虑我们刚才看到的深度示例，下面的图形作为输入(箭头从子对象指向父对象)：



然后，深度谓词的计算过程如下：

舞台	深度	评论
0		我们总是从空集开始。
1	`(0, b), (0, d), (0, e)`	没有子节点的深度为 0。a 和 c 的递归步骤无法生成值，因为它们的某些子级没有深度值。
2	`(0, b), (0, d), (0, e), (1, c)`	c 的递归步骤成功了，因为 depth 现在对它的所有子级(d 和 e)都有一个值。a 的递归步骤仍然失败。
3	`(0, b), (0, d), (0, e), (1, c), (2, a)`	a 的递归步骤成功，因为 depth 现在对它的所有子级(b 和 c)都有一个值。

在这里，我们可以看到，在中间阶段，如果某些子元素缺少值，则聚合失败是非常重要的，这可以防止错误的值被添加。

Any

any 表达式的一般语法类似于聚合的语法，即：

```
any(<variable declarations> | <formula> | <expression>)
```

应始终包含变量声明，但公式和表达式部分是可选的。
any 表达式表示具有特定形式且满足特定条件的任何值。更确切地说，any 表达式：

1. 引入临时变量。
2. 将它们的值限制为满足<formula>部分的值(如果存在)。
3. 为每个变量返回<expression>。如果没有<expression>部分，则返回变量本身。

下表列出了任何表达式的不同形式的一些示例：

Expression	Values
`any(File f)`	all `File`s in the database
`any(Element e | e.getName())`	the names of all `Element`s in the database
`any(int i | i = [0 .. 3])`	the integers `0`, `1`, `2`, and `3`
`any(int i | i = [0 .. 3] | i * i)`	the integers `0`, `1`, `4`, and `9`

注意

还有一个内置谓词 any()。这是一个始终有效的谓词。

一元操作

一元运算是减号(-)或加号(+)，后跟 int 或 float 类型的表达式。例如：

```
-6.28

+(10 - 4)

+avg(float f | f = 3.4 or f = -9.8)

-sum(int i | i in [0 .. 9] | i * i)
```

加号使表达式的值保持不变，而负号则对值进行算术求反。

二进制运算

二进制运算由一个表达式、一个二进制运算符和另一个表达式组成。例如：

```
5 % 2

(9 + 1) / (-2)

"Q" + "L"

2 * min(float f | f in [-3 .. 3])
```

可以在 QL 中使用以下二进制运算符：

姓名	符号
加法/串联	`+`
乘法	`*`
分部	`/`
减法	`-`
模	`%`

如果两个表达式都是数字，则这些运算符充当标准算术运算符。例如，10.6-3.2 的值为 7.4，123.456*0 的值为 0，9%4 的值为 1(9 除以 4 后的余数)。如果两个操作数都是整数，则结果为整数。否则，结果为浮点数。
也可以使用+作为字符串连接运算符。在这种情况下，至少一个表达式必须是字符串，另一个表达式使用 toString()谓词隐式转换为字符串。这两个表达式连接在一起，结果是一个字符串。例如，表达式 221+“B”的值为“221B”。

类型转换

强制转换允许您约束表达式的类型。这与其他语言中的强制转换类似，例如在 Java 中。
您可以用两种方式编写演员阵容：
• 作为“后缀”转换：一个点后跟一个类型的名称在括号中。例如，`x.(Foo)` 将 x 的类型限制为 Foo。
• 作为“prefix”类型转换：括号中的一种类型，后跟另一个表达式。例如，`((Foo)x` 也将 x 的类型限制为 Foo。
注意，后缀转换相当于用圆括号括起来的前缀转换-`x.(Foo)`完全等同`((Foo)x)`.
如果要调用只为更特定类型定义的成员谓词，则强制转换非常有用。例如，下面的查询选择具有直接超类型“List”的 Java 类：

```
import java



from Type t

where t.(Class).getASupertype().hasName("List")

select t
```

由于谓词 getASupertype()是为类定义的，而不是为 Type 定义的，所以不能直接调用 t.getASupertype()。强制转换 t.(Class)确保 t 是 Class 类型，因此它可以访问所需的谓词。
如果希望使用前缀转换，可以将 where 部分重写为：

```
where ((Class)t).getASupertype().hasName("List")
```

不用在乎的表达式

这是一个以单下划线(_)形式编写的表达式。它代表任何价值。(你“不在乎”值是多少。)
与其他表达式不同，don-not-care 表达式没有类型。实际上，这意味着_没有任何成员谓词，因此不能调用`_.somePredicate()`.
例如，以下查询选择字符串“hello”中的所有字符：

```
from string s

where s = "hello".charAt(_)

select s
```

`charAt(int i)` 谓词是在字符串上定义的，通常使用 int 参数。这里，don’t care 表达式用于告诉查询在每个可能的索引处选择字符。查询返回值 h、e、l 和 o。

公式

公式定义表达式中使用的自由变量之间的逻辑关系。

根据分配给这些自由变量的值，公式可以是真或假。当一个公式是真的，我们常说这个公式成立。例如，如果将值 9 赋给 x，则公式 x=4+5 成立，但对于其他赋值给 x 则不成立。有些公式没有任何自由变量。例如 1<2 始终有效，而 1>2 从不保持。
通常在类、谓词和 select 子句的主体中使用公式来约束它们引用的值集。例如，您可以定义一个包含所有整数 i 的类，其中公式 `i in [0 .. 9]` 保持。
下面几节描述了 QL 中可用的各种公式。

比较

比较公式的形式如下：

```
<expression><operator><expression>
```

有关可用比较运算符的概述，请参阅下表。

运算符

要使用这些顺序运算符之一比较两个表达式，每个表达式必须有一个类型，并且这些类型必须兼容且可排序。

姓名	符号
大于	`>`
大于或等于	`>=`
少于	`<`
小于或等于	`<=`

例如，公式`"Ann" < "Anne"` 和 `5 + 6 >= 11` 都成立。

平等

若要使用=比较两个表达式，其中至少有一个表达式必须具有类型。如果两个表达式都有一个类型，则它们的类型必须兼容。
比较两个表达式使用！=，两个表达式都必须有一个类型。这些类型也必须兼容。

姓名	符号
等于	`=`
不等于	`!=`

例如，如果 x 是 4，那么 `x.sqrt() = 2`
就成立了！=5 始终有效。
对于表达式 A 和 B，如果有一对值 A 和 B 相同，则公式 A=B 成立。换句话说，A 和 B 至少有一个共同的值。例如，`[1 .. 2] =[2 .. 5]` 保留，因为两个表达式的值都为 2。
结果，`A != B` 与 `A != B` 的否定有着截然不同的含义：

·A！=B 如果有一对值(一个来自 a，一个来自 B)不同，则保持不变。

·如果不存在一对相同的值，则 A=B 不成立。换句话说，a 和 B 没有共同的值。

示例

1. 如果两个表达式都有一个值(例如 1 和 0)，则比较很简单：
    - 1！=0 保持。
    - 1=0 不成立。
    - 不是 1=0 保持不变。
2. 现在比较 1 和[1..2] 公司名称：
    - 1！=[1..2] 坚持住，因为 1！=2。
    - 1=[1..2] 保持，因为 1=1。

    不是 1=[1..2] 不成立，因为有一个公共值(1)。
3. 比较 1 和 none()(“空集”)：
    - 1！=none()不成立，因为 none()中没有值，因此没有不等于 1 的值。
    - 1=none()也不成立，因为 none()中没有值，所以没有等于 1 的值。
    - 不是 1=none()保持，因为没有公共值。

类型检查

类型检查是一个如下公式：

```
<expression> instanceof <type>
```

可以使用类型检查公式检查表达式是否具有特定类型。例如，如果变量 x 的类型为 Person，则 xinstanceof Person 保持不变。

范围检查

范围检查是一个如下公式：

```
<expression> in <range>
```

可以使用范围检查公式检查数值表达式是否在给定范围内。例如，x in[2.1 .. 10.5]如果变量 x 在值 2.1 和 10.5 之间(包括 2.1 和 10.5 本身)，则保持不变。

注意<range>中的<expression>相当于<expression>=<range>。两个公式都会检查由<expression>表示的值集是否与由<range>表示的值集相同。

调用谓词

调用是一个公式或表达式，由对谓词的引用和多个参数组成。
例如，isThree(x)可能是对一个谓词的调用，如果参数 x 是 3，那么 x.isEven()可能是对成员谓词的调用，如果 x 是偶数，则该谓词保持不变。
对谓词的调用也可以包含闭包运算符，即*或+。例如，a.isChildOf+(b)是对isChildOf()的传递闭包的调用，因此如果 a 是 b 的子代，则它成立。
谓词引用必须精确解析为一个谓词。有关如何解析谓词引用的详细信息，请参见名称解析。
如果调用解析为一个没有结果的谓词，则该调用是一个公式。
也可以用 result 调用谓词。这种调用是 QL 中的表达式，而不是公式。有关相应主题，请参见调用谓词(带结果)。

带圆括号的公式

带圆括号的公式是由括号(和)包围的任何公式。这个公式和所附公式的含义完全相同。括号通常有助于提高可读性并将某些公式组合在一起。

量化公式

一个量化的公式引入了临时变量，并在其主体的公式中使用它们。一种从现有公式中创建新公式的方法。

显式量词

下面的显式“量词”与数理逻辑中常见的存在量词和普遍量词相同。

exits

此量词具有以下语法：

```
exists(<variable declarations> | <formula>)
```

您也可以编写 exists(<variable declarations><formula 1>|<formula 2>)。这相当于 exists(<variable declarations>和<formula 2>)。

这个量化公式引入了一些新的变量。它适用于至少有一组变量可以采用的值来使主体中的公式为真。
例如，exists(inti | i instanceof OneTwoThree)引入了一个 into 类型的临时变量，如果该变量的任何值具有 OneTwoThree 类型，则保留该变量。

forall

此量词具有以下语法：

```
forall(<variable declarations> | <formula 1> | <formula 2>)
```

forall 引入了一些新的变量，通常在它的主体中有两个公式。如果<formula 2>对所有<formula 1>的值都有效，则它成立。
例如，forall(inti | i instanceof OneTwoThree | i<5)在类 OneTwoThree 中的所有整数都小于 5 时成立。换句话说，如果 onetwo3 中有一个值大于或等于 5，那么这个公式就不成立了。
注意，forall(<vars><formula 1><formula 2>)在逻辑上与 not exists 相同(<vars><formula 1>| not<formula 2>)。

forex

此量词具有以下语法：

```
forex(<variable declarations> | <formula 1> | <formula 2>)
```

这个量词是以下意思的简写：

```
forall(<vars> | <formula 1> | <formula 2>) and

exists(<vars> | <formula 1> | <formula 2>)
```

换言之，外汇的工作方式与 forall 相似，只是它确保至少有一个值符合<formula 1>的要求。要了解这为什么有用，请注意 forall 量词可以微不足道地容纳。例如，forall(int i | i=1 and i=2 | i=3)成立：没有一个整数 i 同时等于 1 和 2，因此主体的第二部分(i=3)对于第一部分包含的每个整数都有效。
由于这通常不是您希望在查询中使用的行为，所以外汇量词是一种有用的速记。

隐式量词

隐式量化变量可以使用“不关心表达式”来引入。当你需要引入一个变量作为谓词调用的参数，但不关心它的值时，可以使用这些表达式。有关详细信息，请参见不关心表达式。

逻辑连接词

你可以在 QL 中的公式之间使用许多逻辑连接词，它们允许你将现有的公式组合成更长、更复杂的公式。
若要指示公式的哪些部分应优先，可以使用括号。否则，从高到低的优先顺序如下：

1. 否定(not)
2. 条件公式(if…then…else)
3. 连接(and)
4. 分离(or)
5. 暗示(implies)

例如，`A and B implies C or D`，相当于`(A and B) implies (C or D)`.。
类似地，`A and not if B then C else D` 等同于
`A and (not (if B then C else D))`。
请注意，以上示例中的括号不是必需的，因为它们突出显示了默认优先级。通常只添加圆括号来覆盖默认优先级，但也可以添加圆括号以使代码更易于阅读(即使不需要)。
QL 中的逻辑连接词的工作方式与其他编程语言中的布尔连接词类似。以下是简要概述：

Not

可以在公式之前使用关键字 not。由此得到的公式称为否定。
当 A 不存在时，A 就不成立了。
例子
下面的查询选择不是 HTML 文件的文件。

```
from File f

where not f.getFileType().isHtml()

select f
```

注意

在递归定义中使用 not 时应该小心，因为这可能导致非单调递归。有关详细信息，请参阅非单调递归部分。

if ... then ... else

可以使用这些关键字编写条件公式。这是另一种简化记法的方法：
`if Athen B else C` 等于写`(A and B) or ((not A) and C)`.
例子
根据以下定义，如果 x 是公共类，visibility(c)返回“public”，否则返回“private”：

```
string visibility(Class c){

  if c.isPublic()

  then result = "public"

  else result = "private"

}
```

And

可以在两个公式之间使用关键字 and。由此产生的公式称为连词。
当且仅当 A 和 B 都成立时，A 和 B 成立。
例子
以下查询选择具有 js 扩展名且包含少于 200 行代码的文件：

```
from File f

where f.getExtension() = "js" and

  f.getNumberOfLinesOfCode() < 200

select f
```

or

可以在两个公式之间使用关键字或。由此产生的公式称为析取。
如果 A 或 B 中至少有一个成立，则 A 或 B 成立。
例子
在下面的定义中，如果整数等于 1、2 或 3，则该整数属于 `OneTwoThree` 类：

```
class OneTwoThree extends int {

  OneTwoThree() {

    this = 1 or this = 2 or this = 3

  }

  ...

}
```

implies

可以在两个公式之间使用关键字 implies。由此得到的公式称为蕴涵式。这只是一个简化的符号：`A implies B` 与书写`(not A) or B`.相同。

例子

下面的查询选择任何奇数的 SmallInt 或 4 的倍数。

```
class SmallInt extends int {

  SmallInt() { this = [1 .. 10] }

}

from SmallInt x

where x % 2 = 0 implies x % 4 = 0

select x
```

脚注

	A 之间的区别！=B 而不是 A=B 是由底层量词引起的。如果你认为 A 和 B 是一组值，那么 A！=B 表示
	`exists( a, b | a in A and b in B | a != b )`
[1]	另一方面，不是 A=B 意味着：
	`not exists( a, b | a in A and b in B | a = b )`
	这相当于 a 中的 a，b | a 和 b | a 中的 b！=b)，这与第一个公式非常不同。

注解

注解是可以直接放在 QL 实体或名称声明之前的字符串。
例如，要将模块 M 声明为 private，可以使用：

```
private module M {

    ...

}
```

请注意，一些注解作用于实体本身，而其他注解作用于实体的特定名称：
•对实体执行操作：abstract、cached、external、transient、final、override、pragma、language 和 bindingset
• 对名称执行操作：deprecated、library、private 和 query
例如，如果用 private 注解一个实体，那么只有这个特定的名称是 private。您仍然可以使用其他名称(使用别名)访问该实体。另一方面，如果用 cached 注解一个实体，那么实体本身也会被缓存。

下面是一个明确的例子：

```
module M {

  private int foo() { result = 1 }

  predicate bar = foo/0;

}
```

在本例中，查询 `select M::foo()`将给出一个编译器错误，因为 foo 的名称是私有的。查询 `select M::bar()`是有效的(给出结果 1)，因为名称 bar 是可见的，并且是谓词 foo 的别名。
可以将缓存应用于 foo，但不能应用于 bar，因为 foo 是实体的声明。

注解概述

注解，以及在这一节中可以使用的不同之处。您还可以在 QL 语言规范的 Annotations 部分找到摘要表。

摘要

可用于：类，成员谓词
抽象注解用于定义抽象实体。
有关抽象类的信息，请参见类。
抽象谓词是没有主体的成员谓词。它们可以在任何类上定义，并且应该在非抽象子类型中重写。
下面是一个使用抽象谓词的示例。在 QL 中编写数据流分析时，一个常见的模式是定义一个配置类。这种配置必须描述它所跟踪的数据源等。所有这些配置的超类型可能如下所示：

```
abstract class Configuration extends string {

  ...

  /** Holds if `source` is a relevant data flow source. */

  abstract predicate isSource(Node source);

  ...

}
```

然后可以定义 `Configuration` 的子类型，这些子类型继承谓词 isSource，以描述特定的配置。任何非抽象子类型都必须重写它(直接或间接)以描述它们各自跟踪的数据源。
换句话说，所有扩展配置的非抽象类必须在其自身的主体中重写 isSource，或者必须从重写 isSource 的另一个类继承：

```
class ConfigA extends Configuration {

  ...

  // provides a concrete definition of `isSource`

  override predicate isSource(Node source) { ... }

}

class ConfigB extends ConfigA {

  ...

  // doesn't need to override `isSource`, because it inherits it
from ConfigA
```

```
}
```

cached

可用于：类、代数数据类型、特征谓词、成员谓词、非成员谓词、模块

cached 注解指示应该对实体进行整体求值，并将其存储在求值缓存中。以后对该实体的所有引用都将使用已计算的数据。这将影响来自其他查询以及当前查询的引用。
例如，缓存一个需要很长时间计算并在许多地方重用的谓词可能会很有帮助。
您应该小心地使用 cached，因为它可能会产生意想不到的后果。例如，缓存的谓词可能会占用大量存储空间，并且可能会阻止 QL 编译器根据每个使用位置的上下文优化谓词。然而，对于只需计算一次谓词，这可能是一个合理的折衷。
如果用 cached 注解类或模块，那么它的主体中的所有非私有实体也必须用 cached 进行注解，否则将报告编译器错误。

已弃用 deprecated

可用于：类、代数数据类型、成员谓词、非成员谓词、字段、模块、别名
已弃用的批注将应用于过期的名称，并计划在将来的 QL 版本中删除这些名称。如果任何 QL 文件使用不推荐使用的名称，则应考虑重写它们以使用较新的替代项。通常，不推荐使用的名称有一个 QLDoc 注解，它告诉用户应该使用哪个更新的元素。
例如，名称 DataFlowNode 已弃用，并具有以下 QLDoc 注解：

```
/**
 * DEPRECATED: Use `DataFlow::Node` instead.
 *
 * An expression or function/class declaration,
 * viewed as a node in a data flow graph.
 */
deprecated class DataFlowNode extends @dataflownode {
  ...
}
```

在 QL 编辑器中使用 DataFlowNode 名称时，将出现此 QLDoc 注解。

external

可用于：非成员谓词
外部注解用于谓词，用于定义外部“模板”谓词。这类似于数据库谓词。

transient

可用于：非成员谓词
临时注解应用于也用 external 注解的非成员谓词，以指示在计算期间不应将它们缓存到磁盘。注意，如果您尝试在没有外部的情况下应用 transient，编译器将报告错误。

final

可用于：类、成员谓词、字段
最后的注解将应用于无法重写或扩展的实体。换句话说，final 类不能作为任何其他类型的基类型，final 谓词或字段也不能在子类中重写。
如果您不希望子类更改特定实体的含义，这将非常有用。
例如，谓词 hasName(string name)在元素具有名称名称时有效。它使用谓词 getName()来检查这一点，而子类更改此定义是没有意义的。在这种情况下，hasName 应该是 final：

```
class Element ... {

  string getName() { result = ... }

  final predicate hasName(string name) { name = this.getName() }

}
```

library

可用于：类
重要的

此批注已弃用。不要用库来注解名称，而是将其放入私有(或私有导入)模块中。

库注解应用于只能从.qll 文件中引用的名称。如果试图从不具有.qll 扩展名的文件中引用该名称，则 QL 编译器将返回错误。

override

可用于：成员谓词，字段
override 注解用于指示定义重写基类型中的成员谓词或字段。
如果重写了谓词或字段而没有对其进行注解，则 QL 编译器将发出警告。

private

可用于：类、代数数据类型、成员谓词、非成员谓词、导入、字段、模块、别名
私有注解用于阻止导出名称。
如果名称的注解是私有的，或者通过带有 private 注解的 import 语句访问它，那么只能从当前模块的命名空间中引用该名称。

query

可用于：非成员谓词，别名
查询注解用于将谓词(或谓词别名)转换为查询。这意味着它是 QL 程序输出的一部分。

Compiler pragmas

可用于：特征谓词、成员谓词、非成员谓词
以下编译器 pragma 会影响查询的编译和优化。除非遇到严重的性能问题，否则应避免使用这些注解。
在向代码中添加 pragma 之前，请联系 GitHub 以描述性能问题。这样我们就可以为您的问题提出最佳解决方案，并在改进 QL 优化器时将其考虑在内。

Inlining

对于简单谓词，QL 优化器有时会用谓词体本身替换对谓词的调用。这称为内联。
例如，假设您有一个定义谓词 `predicate one(int i) { i = 1 }`并调用了该谓词`... one(y) ...`. 优化器可以将谓词内联到`... y = 1 ...`
您可以使用以下编译器 pragma 注解来控制 QL 优化器内联谓词的方式。

pragma[inline]

pragma[inline]注解告诉 QL 优化器始终将带注解的谓词内联到调用它的位置。当谓词体的整个计算开销非常大时，这一点非常有用，因为它可以确保谓词在调用它的位置使用其他上下文信息进行求值。

pragma[noinline]

pragma[noinline]注解用于防止谓词内联到调用它的位置。实际上，当您已经在“helper”谓词中将某些变量组合在一起时，此注解非常有用，以确保在一个片段中计算关系。这有助于提高性能。QL 优化器的内联可能会撤消 helper 谓词的工作，因此最好使用 pragma[noinline]对其进行注解。

pragma[nomagic]

pragma[nomagic]注解用于阻止 QL 优化器对谓词执行“magic sets”优化。
这种优化包括从谓词调用的上下文中获取信息并将其推送到谓词体中。这通常是有益的，因此除非 GitHub 建议使用 pragma[nomagic]注解，否则不应使用 pragma[nomagic]注解。
注意，nomagic 意味着没有线。

pragma[noopt]

pragma[noopt]注解用于防止 QL 优化器优化谓词，除非编译和求值是绝对必要的。
这很少是必需的，除非 GitHub 建议使用 pragma[noopt]注解来帮助解决性能问题，否则不应使用 pragma[noopt]注解。
使用此批注时，请注意以下问题：

1. QL 优化器以一种有效的方式自动对复杂公式的连接进行排序。在 noopt 谓词中，连接词的计算顺序与编写它们的顺序完全相同。
2. QL 优化器自动创建中间连接，将某些公式“转换”为更简单的公式的连接。在 noopt 谓词中，必须显式地编写这些连词。尤其是，不能在强制转换上链接谓词调用或调用谓词。你必须把它们写成多个连接词并显式地排序。
   例如，假设您有以下定义：

```
class Small extends int {

  Small() { this in [1 .. 10] }

  Small getSucc() { result = this + 1}
```

```
}


predicate p(int i) {
  i.(Small).getSsucc() = 2
}


predicate q(Small s) {
  s.getSsucc().getSsucc() = 3
}
```

如果添加 noopt pragma，则必须重写谓词。例如：

```
pragma[noopt]
predicate p(int i) {
  exists(Small s | s = i and s.getSucc() = 2)
}


pragma[noopt]
predicate q(Small s) {
  exists(Small succ |
    succ = s.getSucc() and
    succ.getSucc() = 3
  )
}
```

Language pragmas

可用于：类、特征谓词、成员谓词、非成员谓词

`language[monotonicAggregates]`

此注解允许您使用单调聚合，而不是标准的 QL 聚合。
有关详细信息，请参见单调聚合。

Binding sets

可用于：特征谓词、成员谓词、非成员谓词

`bindingset[...]`

可以使用此注解显式地声明谓词的绑定集。绑定集是谓词参数的子集，因此，如果这些参数被约束到一组有限的值，那么谓词本身是有限的(也就是说，它的计算结果是有限的元组集)。
bindingset 注解采用逗号分隔的变量列表。每个变量必须是谓词的一个参数，可能包括 this(对于特征谓词和成员谓词)和 result(对于返回结果的谓词)。
有关示例和更多信息，请参见谓词主题中的绑定行为。

递归

QL 为递归提供了强大的支持。如果 QL 中的谓词直接或间接依赖于自身，则称其为递归谓词。
为了计算递归谓词，QL 编译器查找递归的最小不动点。特别是，它从空值集开始，通过反复应用谓词直到值集不再更改来查找新值。此集合是最小不动点，因此是求值的结果。类似地，QL 查询的结果是查询中引用的谓词的最小固定点。
在某些情况下，还可以递归地使用聚合。有关详细信息，请参见单调聚合。

递归谓词示例

以下是 QL 中递归谓词的几个示例：

从 0 数到 100

以下查询使用谓词 getANumber()列出从 0 到 100(包括 0)的所有整数：

```
int getANumber() {

  result = 0

  or

  result <= 100 and result = getANumber() + 1

}


select getANumber()
```

谓词 getANumber()的计算结果是包含 0 的集合以及比集合中已有的数字多 1 个的任何整数(最多 100 个)。

相互递归

谓词可以是相互递归的，也就是说，可以有一个相互依赖的谓词循环。例如，下面是一个使用偶数计数为 100 的 QL 查询：

```
int getAnEven() {

  result = 0

  or

  result <= 100 and result = getAnOdd() + 1

}


int getAnOdd() {

  result = getAnEven() + 1

}


select getAnEven()
```

此查询的结果是 0 到 100 之间的偶数。可以用 select getAnEven()替换 select getAnEven()，以列出从 1 到 101 的奇数。

传递闭包

谓词的传递闭包是一个递归谓词，其结果是通过重复应用原始谓词得到的。特别是，原始谓词必须有两个参数(可能包括 this 或 result 值)，并且这些参数必须具有兼容的类型。
由于传递闭包是递归的常见形式，因此 QL 有两个有用的缩写：

1. **传递闭包 `+`**
   要将一个谓词应用一次或多次，请将+附加到谓词名称。
   例如，假设您有一个 Person 类，其成员谓词为 getAParent()。然后 p.getAParent()返回 p 的任何父项，传递闭包 p.getAParent+()返回 p 的父项，p 的父项，依此类推。
   使用这个+符号通常比显式定义递归谓词简单。在这种情况下，一个明确的定义可以如下所示：

```
Person getAnAncestor() {
  result = this.getAParent()
  or
  result = this.getAParent().getAnAncestor()
}
```

   谓词 getAnAncestor()相当于 getAParent+()。
2. **自反传递闭包 `*`**
   这与上述传递闭包运算符类似，只是可以使用它将谓词应用于自身零次或多次。
   例如，p.getAParent*()的结果是 p(如上所述)或 p 本身的祖先。
   在这种情况下，显式定义如下所示：

```
Person getAnAncestor2() {
  result = this
  or
  result = this.getAParent().getAnAncestor2()
}
```

谓词 getAnAncestor2()相当于 getAParent*()。

限制和常见错误

虽然 QL 是为查询递归数据而设计的，但递归定义有时很难得到正确的定义。
如果递归定义包含错误，则通常不会得到结果，或者编译器错误。
以下示例说明了导致无效递归的常见错误：

空递归

首先，一个有效的递归定义必须有一个起点或基本情况。如果递归谓词的计算结果是空值集，则通常会出现错误。
例如，您可以尝试定义谓词 getAnAncestor()(来自上例)，如下所示：

```
Person getAnAncestor() {

  result = this.getAParent().getAnAncestor()

}
```

在这种情况下，QL 编译器会给出一个错误，指出这是一个空的递归调用。
由于 getAnAncestor()最初假定为空，因此无法添加新值。谓词需要递归的起点，例如：

```
Person getAnAncestor() {

  result = this.getAParent()

  or

  result = this.getAParent().getAnAncestor()

}
```

非单调递归

有效的递归谓词也必须是单调的。这意味着(相互)递归只允许在偶数个否定的情况下进行。
直观地说，这防止了“说谎者悖论”的情况，即没有递归的解决方案。例如：

```
predicate isParadox() {
```

```
  not isParadox()

}
```

根据这个定义，谓词 isParadox()在不存在时精确地保持不变。这是不可能的，所以递归没有不动点解。
如果递归出现在偶数个否定下，那么这不是问题。例如，考虑 Person 类的以下成员谓词(有点恐怖)：

```
predicate isExtinct() {

  this.isDead() and

  not exists(Person descendant | descendant.getAParent+() = this |

    not descendant.isExtinct()

  )

}
```

如果 p 和 p 的所有后代都死了，`p.isExtinct()`就成立。
对 isExtinct()的递归调用嵌套在偶数个否定中，因此这是一个有效的定义。实际上，您可以将定义的第二部分重写如下：

```
forall(Person descendant | descendant.getAParent+() = this |

  descendant.isExtinct()

)
```

词汇句法

QL 语法包括不同种类的关键字、标识符和注释。
有关词法语法的概述，请参阅 QL 语言规范中的词法语法。

评论

QL 语言规范中描述的所有标准单行和多行注释都被 QL 编译器忽略，并且只在源代码中可见。您还可以编写另一种注释，即 QLDoc 注释。这些注释描述 QL 实体，并在 QL 编辑器中显示为弹出信息。有关 QLDoc 注释的信息，请参阅 QLDoc 注释规范。
以下示例使用这三种不同类型的注释：

```
/**
 * A QLDoc comment that describes the class `Digit`.
 */
class Digit extends int {  // A short one-line comment
  Digit() {
    this in [0 .. 9]
  }
}


/*
  A standard multiline comment, perhaps to provide
  additional details, or to write a TODO comment.
*/
```

名称解析

QL 编译器将名称解析为程序元素。
与其他编程语言一样，在 QL 代码中使用的名称和它们所引用的底层 QL 实体之间是有区别的。
QL 中的不同实体可能具有相同的名称，例如，如果它们是在不同的模块中定义的。因此，重要的是 QL 编译器可以将名称解析为正确的实体。
当您编写自己的 QL 时，可以使用不同类型的表达式来引用实体。然后将这些表达式解析为相应命名空间中的 QL 实体。
总而言之，表达式的种类有：

- **模块表达式**
  - 这些是指模块。
  - 它们可以是简单名称、限定引用(在导入语句中)或选择。
- **类型表达式**
  - 这些是指类型。
  - 它们可以是简单的名称或选择。
- **谓词表达式**
  - 这些是指谓词。
  - 它们可以是简单的名称或带 `arity` 的名称(例如在别名定义中)，也可以是选择。

名称

要解析一个简单名称(使用 arity)，编译器将在当前模块的名称空间中查找该名称(和 arity)。
在 import 语句中，名称解析稍微复杂一些。例如，假设您定义了一个查询模块示例.ql 使用以下 import 语句：

```
import javascript
```

编译器首先检查库模块 javascript.qll 文件，使用下面描述的步骤获取合格引用。如果失败，它将检查在的模块命名空间中定义的名为 javascript 的显式模块 Example.ql

范围限定(Qualified references)

限定引用是使用的模块表达式。作为文件路径分隔符。只能在 import 语句中使用这样的表达式来导入由相对路径定义的库模块。
例如，假设您定义了一个查询模块示例.ql 使用以下 import 语句：

```
import examples.security.MyLibrary
```

为了找到这个库模块的精确位置，QL 编译器处理 import 语句如下：

1. 限定引用中的.s 对应于文件路径分隔符，因此它首先查找 `examples/security/MyLibrary.qll` 从包含 `Example.ql`.的目录里.

2. 如果失败，它将查找 `examples/security/MyLibrary.qll` 相对于封闭查询目录(如果有)。此查询目录是包含 query.xml 文件，以及该文件的内容与当前数据库架构兼容的位置。(例如，如果要查询 JavaScript 数据库，则 query.xml 文件应包含<queries language="javascript"/>.)

3. 如果使用以上两个检查没有找到文件，它将查找 `examples/security/MyLibrary.qll` 相关到每个库路径条目。库路径取决于运行查询的环境，以及是否指定了任何额外的设置。
如果编译器无法解析 import 语句，则会给出编译错误。
这个过程在 QL 语言规范中关于模块解析的部分中有更详细的描述。

选择

可以使用选择引用特定模块内的模块、类型或谓词。选择的形式如下：

```
<module_expression>::<name>
```

编译器首先解析模块表达式，然后在名称空间中查找该模块的名称。

例子

考虑以下库模块：
CountriesLib.qll

```
class Countries extends string {

  Countries() {

    this = "Belgium"

    or

    this = "France"

    or

    this = "India"

  }

}


module M {

  class EuropeanCountries extends Countries {

    EuropeanCountries() {
```

```
      this = "Belgium"
      or
      this = "France"
    }
  }
}
```

您可以编写一个查询，导入 CountriesLib，然后使用 M::EuropeanCountries 引用类 EuropeanCountries:

```
import CountriesLib


from M::EuropeanCountries ec
select ec
```

或者，也可以使用 import 语句中的选择 CountriesLib::M 直接导入 M 的内容：

```
import CountriesLib::M


from EuropeanCountries ec
select ec
```

这使查询可以访问 M 中的所有内容，但 CountriesLib 中不在 M 中的任何内容。

命名空间

在编写 QL 时，了解名称空间(也称为环境)的工作方式非常有用。
与许多其他编程语言一样，命名空间是从键到实体的映射。键是一种标识符，例如名称，QL 实体是模块、类型或谓词。
QL 中的每个模块都有三个名称空间：

- 模块名称空间，其中键是模块名，实体是模块。
- 类型命名空间，其中键是类型名，实体是类型。
- 谓词名称空间，其中键是成对的谓词名称和 arity，实体是谓词。

知道不同名称空间中的名称之间没有关系是很重要的。例如，两个不同的模块可以定义一个谓词 getLocation()，而不会混淆。只要清楚您所在的名称空间，QL 编译器就会将名称解析为正确的谓词。

全局命名空间

包含所有内置实体的名称空间称为全局名称空间，并自动在任何模块中可用。特别地：

- 全局模块命名空间为空。
- 全局类型名称空间包含基本类型 int、float、string、boolean 和 date 的条目，以及在数据库模式中定义的任何数据库类型。
- 全局谓词名称空间包括所有内置谓词以及任何数据库谓词。

实际上，这意味着您可以直接在 QL 模块中使用内置类型和谓词(无需导入任何库)。您还可以直接使用任何数据库谓词和类型—这些依赖于您正在查询的底层数据库。

本地命名空间

除了全局模块、类型和谓词命名空间之外，每个模块还定义了许多本地模块、类型和谓词命名空间。
对于模块 M，区分其声明的、导出的和导入的名称空间是很有用的。(这些都是一般性描述的，但是请记住，对于每个模块、类型和谓词，总是有一个对应的。)

- 声明的命名空间包含声明的任何名称，即在 M 中定义的名称。
- 导入的名称空间包含由导入到 Musing import 语句的模块导出的任何名称。
- 导出的命名空间包含在 M 中声明的任何名称，或从导入到 M 中的模块导出的名称，但带有 private 注释的名称除外。这包括导入的命名空间中未由私有导入引入的所有内容。

这是一个最容易理解的例子：
OneTwoThreeLib.qll

```
import MyFavoriteNumbers


class OneTwoThree extends int {

  OneTwoThree() {

    this = 1 or this = 2 or this = 3

  }
```

```
}


private module P {
  class OneTwo extends OneTwoThree {
    OneTwo() {
      this = 1 or this = 2
    }
  }
}
```

`OneTwoThreeLib` 模块导入模块 `MyFavoriteNumbers` 导出的任何内容。
它声明了类 `OneTwoThree` 和模块 P。
它导出类 `OneTwoThree` 和 MyFavoriteNumbers 导出的任何内容。它不导出 P，因为它被注释为 private。

例子

一般 QL 模块的名称空间是本地名称空间、任何封闭模块的名称空间和全局名称空间的联合。(可以将全局命名空间视为顶级模块的封闭命名空间。)
让我们在一个具体的示例中看看模块、类型和谓词名称空间是什么样子：
例如，可以定义一个库模块 villagen，其中包含在 QL 教程中定义的一些类和谓词：
Villagers.qll

```
import tutorial


predicate isBald(Person p) {
  not exists(string c | p.getHairColor() = c)
}


class Child extends Person {
  Child() {
```

```
    this.getAge() < 10
  }
}

module S {
  predicate isSouthern(Person p) {
    p.getLocation() = "south"
  }

  class Southerner extends Person {
    Southerner() {
      isSouthern(this)
    }
  }
}
```

模块命名空间

村民的模块命名空间包含以下条目：

·模块 S。

·tutorial 导出的任何模块。

S 的模块名称空间还包含模块 S 本身以及 tutorial 导出的任何模块的条目。

类型命名空间

村民的类型命名空间包含以下项：

·Child 类。

·模块 tutorial 导出的类型。

·内置类型，即 int、float、string、date 和 boolean。

S 的类型命名空间包含以下项：

•以上所有类型。

•Southerner 类。

谓词命名空间

村民的谓词名称空间包含以下条目：

- 谓词 `isBald`，arity 为 1。
- 教程导出的任何谓词(及其 arity)。

• 内置谓词。
S 的谓词命名空间包含以下项：
• 以上所有谓词。
• 谓词是 southern，arity 为 1。

QL 程序的评估

QL 程序的计算分为几个不同的步骤。

过程

当对数据库运行 QL 程序时，它被编译成逻辑编程语言 Datalog 的一个变体。对其性能进行优化，然后对其进行评估以产生结果。
这些结果是有序元组的集合。有序元组是有限的有序值序列。例如，`(1, 2, "three")`是一个有两个整数和一个字符串的有序元组。在评估程序时可能会产生中间结果：这些也是元组的集合。
QL 程序是自下而上计算的，因此通常只有在计算了它所依赖的所有谓词之后才对谓词进行求值。
数据库包括内置谓词和外部谓词的有序元组集。每次求值都从这些元组开始。程序中剩余的谓词和类型根据它们之间的依赖关系被组织到许多层中。通过找到每个谓词的最小不动点，对这些层进行求值以生成它们自己的元组集。(例如，请参见递归。)
程序的查询决定了这些元组中的哪一组构成了程序的最终结果。根据查询中的任何排序指令(`order by`)对结果进行排序。
有关评估过程的每个步骤的更多详细信息，请参阅 QL 语言规范。

程序有效性

查询的结果必须始终是一组有限的值，否则无法对其求值。如果您的 QL 代码包含一个无限谓词或查询，QL 编译器通常会给出一条错误消息，以便您更容易地识别错误。
下面是一些定义无限谓词的常用方法。这些都会产生编译错误：
·下面的查询从概念上选择 int 类型的所有值，而不限制它们。QL 编译器返回错误“i”未绑定到值：

```
from int i
```

```
select i
```

·以下谓词生成两个错误：“n”未绑定到值，“result”未绑定到值：

```
int timesTwo(int n) {
  result = n * 2
}
```

·下面的类 Person 包含所有以“Peter”开头的字符串。有无限多这样的字符串，所以这是另一个无效的定义。QL 编译器给出错误消息“this”未绑定到值：

```
class Person extends string {
  Person() {
    this.matches("Peter%")
  }
}
```

要修复这些错误，考虑范围限制是很有用的：如果谓词或查询的每个变量至少有一个绑定出现，则谓词或查询就是范围限制的。没有绑定的变量称为未绑定。因此，要执行范围限制检查，QL 编译器将验证是否没有未绑定的变量。

结合

要避免查询中的无限关系，必须确保没有未绑定的变量。为此，可以使用以下机制：

1. **有限类型**：有限变量类型被束缚。特别是，任何类型的原始的是有限的。要给变量指定一个有限类型，可以声明对于有限类型，使用铸造，或使用型式检查.
2. **谓词调用**：有效谓语通常都有射程限制，所以绑定所有的论据。因此，如果你呼叫变量上的谓词，变量变为绑定。

   重要的

   如果谓词使用非标准绑定集，则它并不总是绑定所有参数。在这种情况下，谓词调用是否绑定特定参数取决于绑定了哪些其他参数，以及绑定集对相关参数的说明。有关详细信息，请参见绑定集。
3. **绑定运算符**：大多数运算符，例如算术运算符，要求绑定所有操作数。例如，不能在 QL 中添加两个变量，除非对这两个变量都有一个有限的可能值集。

但是，有些内置运算符可以绑定它们的参数。例如，如果相等性检查(使用=)的一侧被绑定，而另一侧是变量，则该变量也将变为绑定。示例见下表。

直观地说，绑定事件将变量限制为有限的一组值，而非绑定事件则不会。下面是一些示例来阐明变量的绑定和非绑定出现之间的区别：

Variable occurrence	Details
`x = 1`	Binding: restricts `x` to the value `1`
`x != 1`, `not x = 1`	Not binding
`x = 2 + 3`, `x + 1 = 3`	Binding
`x in [0 .. 3]`	Binding
`p(x, _)`	Binding, since `p()` is a call to a predicate.
`x = y`, `x = y + 1`	Binding for `x` if and only if the variable `y` is bound. Binding for `y` if and only if the variable `x` is bound.
`x = y * 2`	Binding for `x` if the variable `y` is bound. Not binding for `y`.
`x > y`	Not binding for `x` or `y`
`"string".matches(x)`	Not binding for `x`
`x.matches(y)`	Not binding for `x` or `y`
`not (... x ...)`	Generally non-binding for `x`, since negating a binding occurrence typically makes it non-binding. There are certain exceptions: `not not x = 1` is correctly recognized as binding for `x`.
`sum(int y | y = 1 andx = y | y)`	Not binding for `x`. `strictsum(int y | y = 1 and x = y | y)` would be binding for `x`. Expressions in the body of an aggregate are only binding outside of the body if the aggregate is *strict*.
`x = 1 or y = 1`	Not binding for `x` or for `y`. The first subexpression, `x = 1`, is binding for `x`, and the second subexpression, `y = 1`, is binding for `y`. However, combining them with disjunction is only binding for

Variable occurrence	Details
	variables for which **all**disjuncts are binding—in this case, that's no variable.

虽然变量的出现可以是绑定的也可以是非绑定的，但是变量的“绑定”或“未绑定”属性是一个全局概念—一次绑定就足以使变量绑定。
因此，您可以通过提供绑定实例来修复上面的“无限”示例。例如，您可以写下以下内容，而不是 `int timesTwo(int n) { result = n * 2 }`:

```
int timesTwo(int n) {

  n in [0 .. 10] and

  result = n * 2

}
```

谓词现在绑定 n，变量结果自动被计算 `result = n * 2`.绑定。