类（Class）

类的定义

除了基础的数据类型，你也可以在QL中定义自己的类型。一种实现方法是定义一个类（Class），类提供了一种简单的方法来重用和构造代码

例如：

- 将相关联的值分组在一起
- 根据这些值定义成员谓词
- 定义覆盖成员谓词的子类

QL中的类不会“创建”新对象，而只是表示逻辑属性。如果值满足该逻辑属性，则该值属于特定类。

定义一个类

类要素

要定义一个类，你需要遵守以下几条规则

- 关键字 class
- 类的名称，这是一个以字母开头的标识符
- 要扩展的类型
- 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
  }
}
```

这里定义了一个类OneTwoThree 继承自基本的数据类型int，特征谓词是约束变量需要满足值为1、2、3中的一个，QL并不会创建一个实例对象，而是通过逻辑条件约束变量所满足的条件。QL中的类必须始终继承至少一个现有的类。类的值包含在基本类型（int、float、boolean、string、date）的交集内。一个类从其基本类型继承所有的成员谓词

一个类可以继承自多种类型，如果需要更多的信息，请查看 多重继承 章节

一个有效的类必须包含以下条件：

- 不能继承自这个类本身
- 不能继承使用final修饰的类
- 不能继承不兼容的类型，更多相关信息请查看“类型兼容章节”

你也可以使用注释修饰一个class，详情可查看注释列表的章节

类体（Class bodies）

类的主体（大括号括起的部分）可以包含：

- 一个特征谓词的声明
- 任意数量的成员谓词声明
- 任意数量的字段声明

当定义一个类时，该类还从其超类型继承所有非私有（非private）成员谓词和字段。您可以重写（override）这些谓词和字段以为其提供更具体的定义。

特征谓词（Characteristic predicates）

这些是在类主体内部定义的谓词。它们是使用变量`this`来限制类中可能值的逻辑属性。

成员谓词（Member predicates）

这些谓词仅适用于特定类别的成员。您可以根据值调用成员谓词。例如，您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"`
该表达式`(OneTwoThree)`是强制转换。它确保`1`具有类型 `OneTwoThree`而不是`int`。因此，它可以访问成员谓词 `getAString()`。
成员谓词特别有用，因为您可以将它们链接在一起。例如，您可以使用`toUpperCase()`为定义的内置函数`string`：

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

tips: 特征谓词和成员谓词经常使用变量`this`。该变量始终引用该类的成员–在这种情况下，该值属于该类`OneTwoThree`。在特征谓词中，变量`this`约束类中的值。在成员谓词中，其`this`作用与该谓词的任何其他参数相同。

字段（Field）

这些是在类主体中声明的变量。一个类在其主体内可以具有任意数量的字段声明（即变量声明）。您可以在类内部的谓词声明中使用这些变量。就像变量一样 `this`，字段必须限制在特征谓词中。

```
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()
```

在此示例中，声明引入了一个字段，将其约束在特征谓词中，然后在成员谓词的声明中使用它 。这类似于在部分中通过在select语句中声明变量来引入变量 。`SmallInt divisor` `divisor` `getADivisor` `from`
你还可以注释谓词和字段，可参阅注释列表

具体的类（Concrete classes）

上面示例中的类都是**具体的**类。通过将值限制为更大的类型来定义它们。具体类中的值恰好是也满足该类特征谓词的基本类型交集中的那些值 。

抽象类（Abstract classes）

一类注解用`abstract`，被称为**抽象**类，也是值的更大的类型的限制。但是，抽象类被定义为其子类的并集。特别地，对于一个值是在一个抽象类，它必须满足类本身的特性谓词和一个子类的特征谓词。
如果要将多个现有类以通用名称组合在一起，则抽象类很有用。然后，您可以在所有这些类上定义成员谓词。您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，则可以向其添加更多子类。
例子
如果要编写安全查询，则可能有兴趣识别所有可以解释为SQL查询的表达式。您可以使用以下抽象类来描述这些表达式：

```
abstract class SqlExpr extends Expr {
  ...
}
```

现在定义各种子类–每种数据库管理系统一个。例如，您可以定义一个子类，其中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。`class PostgresSqlExpr extends SqlExpr`
抽象类`SqlExpr`引用所有这些不同的表达式。如果以后要添加对另一个数据库系统的支持，则只需将一个新的子类添加到`SqlExpr`；即可。无需更新依赖它的查询。

需要注意的是：将新的子类添加到现有的抽象类时，必须小心。添加子类不是一个孤立的更改，它还扩展了抽象类，因为那是其子类的并集。

重写成员谓词（Overriding member predicates）

如果类从超类型继承成员谓词，则可以**覆盖**继承的定义。为此，您可以定义一个成员谓词，该成员谓词的名称和别名与继承的谓词相同，并添加`override`注解。如果要优化谓词以为子类中的值提供更具体的结果，这将很有用。

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

  override string getAString() {
    result = "One or two: " + this.toString()
  }
}
```

成员谓词将`getAString()`覆盖`getAString()` from的原始定义`OneTwoThree`。
可以使用以下查询

```
from OneTwoThree o
select o, o.getAString()
```

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

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

在QL中，与其他面向对象的语言不同，相同类型的不同子类型不需要脱节。例如，您可以定义的另一个子类`OneTwoThree`，该子类与重叠`OneTwo`

```
class TwoThree extends OneTwoThree {
  TwoThree() {
    this = 2 or this = 3
  }

  override string getAString() {
    result = "Two or three: " + this.toString()
  }
}
```

现在，值2包含在类类型`OneTwo`和中`TwoThree`。这两个类均覆盖的原始定义`getAString()`。有两个新的“最具体的”定义，因此运行上述查询将得到以下结果：

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

一个类可以扩展多种类型。在这种情况下，它从所有这些类型继承。
例如，使用上一节中的定义：

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

在类的任意值`Two`必须满足由下式表示的逻辑属性`OneTwo`，和逻辑属性表示通过`TwoThree`。这里的类`Two`包含一个值，即2。
它从`OneTwo`和继承成员谓词`TwoThree`。它还（间接）继承自`OneTwoThree`和`int`。
tips: 如果子类为同一个谓词名称继承多个定义，则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。（这些是比较微妙的概念，在实际的查询编写中很少出现。）
QL类的**字符类型**是满足该类特征谓词的一组值。它是域类型的子集。对于具体类，当且仅当值属于字符类型时，该值才属于该类。对于抽象类，除字符类型外，值还必须至少属于一个子类。
QL类的**域类型**是其所有超类型的字符类型的交集，即，如果值属于每个超类型，则该值属于域类型。它作为类`this`的特征谓词中的类型出现。

代数数据类型

```
tips:代数数据类型的语法被认为是实验性的，并且可能会发生变化。但是，它们出现在标准QL库中， 因此以下各节应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，用关键字声明`newtype`。
代数数据类型用于创建既不是原始值也不是数据库实体的新值。一个示例是在分析通过程序的数据流时对流节点建模。
代数数据类型由多个互不相交的*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。分支可以具有参数和主体。对于满足参数类型和主体的每组值，都会产生一个分支类型的新值。
这样做的好处是每个分支可以具有不同的结构。例如，如果要定义一个“选项类型”以保存一个值（例如`Call`）或为空，则可以这样编写：

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

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>
```

分支定义具有以下形式：

```
<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，它们以开头`T`。
- 代数数据类型的不同分支之间用分隔`or`。
- 分支的参数（如果有）是 用逗号分隔的变量声明。
- 分支的主体是谓词主体。您可以省略分支主体，在这种情况下，默认为`any()`。请注意，分支实体已完全评估，因此它们必须是有限的。它们应保持较小尺寸以获得良好的性能。

例如：以下代数数据类型具有三个分支

```
newtype T =
  Type1(A a, B b) { body(a, b) }
or
  Type2(C c)
or
```

```
6   Type3()
```

使用代数数据类型的标准模式

代数数据类型与类不同。特别是，代数数据类型没有 `toString()` 成员谓词，因此您不能在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QL语言库中，通常按以下步骤完成：

- 定义一个 `A` 扩展代数数据类型的类，并可选地声明抽象 谓词。
- 对于每种分支类型，定义一个 `B` 既扩展 `A` 了分支类型又扩展分支类型的类，并为中的任何抽象谓词提供定义 `A` 。
- 用private注释代数数据类型，并将类保留为public。

例如，C＃的CodeQL数据流库中的以下代码片段定义了用于处理有污染或无污染值的类。在这种情况下， `TaintType` 扩展数据库类型没有任何意义 。它是污点分析的一部分，而不是基础程序，因此扩展新类型（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 {
}
```

类型联合

类型联合是用关键字声明的用户定义类型`class`。语法类似于类型别名，但在右侧具有两个或多个类型表达式。

通过显式选择该数据类型的分支的子集并将其绑定到新的类型，类型联合用于创建现有代数数据类型的受限子集。还支持数据库类型的类型联合。

您可以使用类型联合为来自代数数据类型的分支的子集命名。在某些情况下，在整个代数数据类型上使用类型并集可以避免谓词中的虚假 递归。例如，以下构造是合法的：

```
newtype InitialValueSource =
  ExplicitInitialization(VarDecl v) { exists(v.getInitializer()) } or
  ParameterPassing(Call c, int pos) { exists(c.getParameter(pos)) } or
  UnknownInitialGarbage(VarDecl v) { not exists(DefiniteInitialization di | v = target(di)) }

class DefiniteInitialization = ParameterPassing or ExplicitInitialization;

VarDecl target(DefiniteInitialization di) {
  di = ExplicitInitialization(result) or
  exists(Call c, int pos | di = ParameterPassing(c, pos) and
                           result = c.getCallee().getFormalArg(pos))
}
```

但是，限制InitialValueSource类扩展的类似实现无效。如果我们DefiniteInitialization改为将其实现为类扩展，则会触发的类型测试InitialValueSource。这导致了一个非法递归，因为依赖于：但是，限制`InitialValueSource`类扩展的类似实现无效。如果我们`DefiniteInitialization`改为将其实现为类扩展，则会触发的类型测试`InitialValueSource`。这导致了一个非法递归，因为依赖于：`DefiniteInitialization -> InitialValueSource -> UnknownInitialGarbage -> ¬DefiniteInitialization` `UnknownInitialGarbage` `DefiniteInitialization`

```
// THIS WON'T WORK: The implicit type check for InitialValueSource involves an illegal recursion
// DefiniteInitialization -> InitialValueSource -> UnknownInitialGarbage -> ¬DefiniteInitialization!
```

```
class DefiniteInitialization extends InitialValueSource {
  DefiniteInitialization() {
    this instanceof ParameterPassing or this instanceof ExplicitInitialization
  }
  // ...
}
```

从CodeQL CLI的2.2.0版本开始支持类型联合。

数据库类型

数据库类型在数据库模式中定义。这意味着它们取决于您要查询的数据库，并根据要分析的数据而有所不同。

例如，如果您要查询CodeQL数据库中的Java项目，则数据库类型可以包括`@ifstmt`，表示Java代码中的if语句和`@variable`表示变量。

类型兼容性

并非所有类型都兼容。例如，这是没有道理的，因为您无法将和进行比较。`4 < "five"` `int` `string`

为了确定类型何时兼容，QL中有许多不同的“类型Universe”。

QL中的宇宙是：

- 每个原始类型一个（除了`int`和`float`，它们在相同的“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。

例如，在定义类时，这导致以下限制：

- 一个类不能扩展多个基本类型。
- 一个类不能扩展多个不同的数据库类型。
- 一个类不能扩展代数数据类型的多个不同分支。

小结

本节我们学习了QL语言里的Class对象的定义，与传统语言不同 QL并不会实例化这些对象，而是通过对象对数据进行集合划分以及提供一些对数据划分的方法，这里需要读者理解一下QL语言这样的设计，从本质上来说，QL语言是一门查询语言