操作符重载

Kotlin 允许我们为自己的类型提供预定义的一组操作符的实现。这些操作符具有固定的符号表示 （如 `+` 或 `*`）和固定的优先级。为实现这样的操作符，我们为相应的类型（即二元操作符左侧的类型和一元操作符的参数类型）提供了一个固定名字的成员函数或扩展函数。 重载操作符的函数需要用 `operator` 修饰符标记。

另外，我们描述为不同操作符规范操作符重载的约定。

一元操作

一元前缀操作符

表达式	翻译为
`+a`	`a.unaryPlus()`
`-a`	`a.unaryMinus()`
`!a`	`a.not()`

这个表是说，当编译器处理例如表达式 `+a` 时，它执行以下步骤：

— 确定 `a` 的类型，令其为 `T`；

— 为接收者 `T` 查找一个带有 `operator` 修饰符的无参函数 `unaryPlus()`，即成员函数或扩展函数；

— 如果函数不存在或不明确，则导致编译错误；

— 如果函数存在且其返回类型为 `R`，那就表达式 `+a` 具有类型 `R`；

注意 这些操作以及所有其他操作都针对基本类型做了优化，不会为它们引入函数调用的开销。

以下是如何重载一元减运算符的示例：

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

val point = Point(10, 20)

fun main() {
   println(-point)  // 输出“Point(x=-10, y=-20)”
}
```

Target platform: JVM Running on kotlin v. 1.7.10

递增与递减

表达式	翻译为

表达式	翻译为
a++	a.inc() + 见下文
a--	a.dec() + 见下文

inc() 和 dec() 函数必须返回一个值，它用于赋值给使用 ++ 或 -- 操作的变量。它们不应该改变在其上调用 inc() 或 dec() 的对象。

编译器执行以下步骤来解析*后缀*形式的操作符，例如 a++：

— 确定 a 的类型，令其为 T；

— 查找一个适用于类型为 T 的接收者的、带有 operator 修饰符的无参数函数 inc()；

— 检测函数的返回类型是 T 的子类型。

计算表达式的步骤是：

— 把 a 的初始值存储到临时存储 a0 中；

— 把 a0.inc() 结果赋值给 a；

— 把 a0 作为表达式的结果返回。

对于 a--，步骤是完全类似的。

对于*前缀*形式 ++a 和 --a 以相同方式解析，其步骤是：

— 把 a.inc() 结果赋值给 a；

— 把 a 的新值作为表达式结果返回。

二元操作

算术运算符

表达式	翻译为
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)、a.mod(b)（已弃用）
a..b	a.rangeTo(b)

对于此表中的操作，编译器只是解析成*翻译为*列中的表达式。

请注意，自 Kotlin 1.1 起支持 rem 运算符。Kotlin 1.0 使用 mod 运算符，它在

Kotlin 1.1 中被弃用。

示例

下面是一个从给定值起始的 Counter 类的示例，它可以使用重载的 `+` 运算符来增加计数：

```
data class Counter(val dayIndex: Int) {
    operator fun plus(increment: Int): Counter {
        return Counter(dayIndex + increment)
    }
}
```

“In”操作符

表达式	翻译为
`a in b`	`b.contains(a)`
`a !in b`	`!b.contains(a)`

对于 `in` 和 `!in`，过程是相同的，但是参数的顺序是相反的。

索引访问操作符

表达式	翻译为
`a[i]`	`a.get(i)`
`a[i, j]`	`a.get(i, j)`
`a[i_1, ……, i_n]`	`a.get(i_1, ……, i_n)`
`a[i] = b`	`a.set(i, b)`
`a[i, j] = b`	`a.set(i, j, b)`
`a[i_1, ……, i_n] = b`	`a.set(i_1, ……, i_n, b)`

方括号转换为调用带有适当数量参数的 `get` 和 `set`。

调用操作符

表达式	翻译为
`a()`	`a.invoke()`
`a(i)`	`a.invoke(i)`
`a(i, j)`	`a.invoke(i, j)`
表达式	翻译为

`a(i_1, ……, i_n)`	`a.invoke(i_1, ……, i_n)`

圆括号转换为调用带有适当数量参数的 `invoke` 。

广义赋值

表达式	翻译为
`a += b`	`a.plusAssign(b)`
`a -= b`	`a.minusAssign(b)`
`a *= b`	`a.timesAssign(b)`
`a /= b`	`a.divAssign(b)`
`a %= b`	`a.remAssign(b)`, `a.modAssign(b)`（已弃用）

对于赋值操作，例如 `a += b` ，编译器执行以下步骤：

- 如果右列的函数可用
  - 如果相应的二元函数（即 `plusAssign()` 对应于 `plus()` ）也可用，那么报告错误（模糊），
  - 确保其返回类型是 `Unit` ，否则报告错误，
  - 生成 `a.plusAssign(b)` 的代码；
- 否则试着生成 `a = a + b` 的代码（这里包含类型检测： `a + b` 的类型必须是 `a` 的子类型）。

注意：赋值在 Kotlin 中*不是*表达式。

相等与不等操作符

表达式	翻译为
`a == b`	`a?.equals(b) ?: (b === null)`
`a != b`	`!(a?.equals(b) ?: (b === null))`

这些操作符只使用函数 `equals(other: Any?): Boolean`，可以覆盖它来提供自定义的相等性检测实现。不会调用任何其他同名函数（如 `equals(other: Foo)` ）。

注意： `===` 和 `!==` （同一性检测）不可重载，因此不存在对他们的约定。

这个 `==` 操作符有些特殊：它被翻译成一个复杂的表达式，用于筛选 `null` 值。 `null == null` 总是 true，对于非空的 `x` ， `x == null` 总是 false 而不会调用 `x.equals()` 。

比较操作符

表达式	翻译为
`a > b`	`a.compareTo(b) > 0`
`a < b`	`a.compareTo(b) < 0`
`a >= b`	`a.compareTo(b) >= 0`
`a <= b`	`a.compareTo(b) <= 0`

所有的比较都转换为对 `compareTo` 的调用，这个函数需要返回 `Int` 值

属性委托操作符

`provideDelegate` 、 `getValue` 以及 `setValue` 操作符函数已在委托属性中描述。

具名函数的中缀调用

我们可以通过中缀函数的调用 来模拟自定义中缀操作符。