# 操作符重载

## 回顾：

回想上节内容的循环语句，为什么我们能够在for循环中使用使用..语法？ 看上去是这样的：

```
for (i in 1..100 step 20) {
      print("$i ")
  }
```

实际上是这样的：

```
for (i in 1.rangeTo(100) step 20) {
      print("$i ")
   }
```

内部调用的是rangeTo方法：

```
public operator fun rangeTo(other: Int): IntRange
```

再回想下我们的解构声明：

```
operator fun component1() = age;
```

它们都是借助operator关键字来帮我们实现一些功能，那它到底是什么呢？

### operator:

将一个函数标记为重载一个操作符或者实现一个约定。 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。比如解构也是实现了一个约定。

**常用类型介绍：**

一元操作：

| 表达式 | 识别为 |
| --- | --- |
| +a | a.unaryPlus() |
| -a | a.unaryMinus() |
| !a | a.not() |
| a++ | a.inc() |
| a-- | a.dec() |

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

下面以!a示例：

```
data class Human(val x: String)

operator fun Human.not() = "非人哉"

fun main() {
    print(!Human("人类"))
}
```

我定义的Human类用重载运算符指定了not方法，并且让它返回值为一个"非人哉"的String类型，这样，我就能直接对Human对象使用！符号了。

递增与递减：

| 表达式 | 识别为 |
| --- | --- |
| a++ | a.inc() |
| a-- | a.dec() |

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()
- 检查函数的返回类型是 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 a.inc() 结果赋值给 a；
- 把 a0 作为表达式的结果返回。

举个例子

```
open class SilverKnight {

    fun commonRight(): String {
```

```
        return "接单权限 2单 "
    }

    override fun toString(): String {
        return commonRight()
    }
}

open class GoldKnight : SilverKnight() {

    fun moreRight(): String {
        return "节假日福利 商城优惠券"
    }

    override fun toString(): String {
        return commonRight() + moreRight();
    }
}

operator fun SilverKnight.inc() = GoldKnight()
```

我给白银骑士增加一重载运算符重载inc()方法，可能因为一些突出表现，我给他++操作，他就能升级为黄金骑士，从而享受更多福利了。

```
var knight = SilverKnight()
    //因为骑士人很好，所以给他++
    knight++
    print(knight)
```

二元操作：

| 表达式 | 识别为 |
| --- | --- |
| a + b | a.plus(b) |

| | |
|---|---|
| a - b | a.minus(b) |
| a * b | a.times(b) |
| a / b | a.div(b) |
| a..b | a.rangeTo(b) |

看到这里，我们也就明白什么我们在for循环中可以用".."替代1~10的区间了，我们再看看+的一些实现：

```
var str = str + " test"
```

而这个加号也是因为String内部用重载运算符重载了上图的plus方法。

```
public operator fun plus(other: Any?): String
```

更多操作符可参考官网

**注意：**

都是事先约定好的（目前120多个），不能凭空出现，所以支持的操作符依赖于kotlin自己的支持

**那如果操作符不够了怎么办？：**

回到我们那个for循环的例子，我们看看step的实现：

```
public infix fun IntProgression.step(step: Int): IntProgression {
    checkStepIsPositive(step > 0, step)
    return IntProgression.fromClosedRange(first, last, if (this.step > 0)
step else -step)
}
```

这个infix 关键字可以理解为对 operator关键字的拓展—中缀表达式。

## infix: 中缀表达式：

**用infix 方式修饰函数让函数可以用中缀的形式进行调用。**

用infix 为我修饰函数让函数可以用中缀的形式进行调用。

如上面的示例代码所述：表示只要是IntProgression 这个类型的对象，都能使用step这个关键字，也就是说，在fun和函数名之间加“.”再指定类型，则表示该函数接受者只能是这个类型，Int最终也就是继承自IntProgression

**自定义：**

比如我想来定义一个 “vs”的中缀表达式来比价两个int值的大小： 我们先定义一个密闭枚举类来清晰的表示比较结果：

```
sealed class CompareResult {
    object LESS : CompareResult() {
        override fun toString(): String {
            return "小于"
        }
    }

    object MORE : CompareResult() {
        override fun toString(): String {
            return "大于"
        }
    }

    object EQUAL : CompareResult() {
        override fun toString(): String {
            return "等于"
        }
    }
}
```

为Int 类型定义vs的中缀表达式：

```
infix fun Int.vs(num: Int): CompareResult =
        if (this - num > 0) {
            CompareResult.MORE
```

```
        } else if (this - num < 0) {
            CompareResult.LESS
        } else {
            CompareResult.EQUAL
        }
```

调用看看：

```
print(5 vs 6)
//输出小于
```

如果我们想让其他类型也支持，就类似的这么定义即可。

**注意点：**

- 它们必须是成员函数或扩展函数；
- 它们必须只有一个参数；
- 其参数不得接受可变数量的参数且不能有默认值。
- 一个函数只有勇于两个角色类似的对象时才将其声明为中缀函数。（推荐：and、to、zip，反例：add）
- 如果一个方法会改动接受者，那么不要声明为中缀形式。