Kotlin 数据类与密封类 Kotlin 枚举类

Kotlin 泛型

泛型，即 "参数化类型"，将类型参数化，可以用在类，接口，方法上。

与 Java 一样，Kotlin 也提供泛型，为类型安全提供保证，消除类型强转的烦恼。

声明一个泛型类:

```
class Box<T>(t: T) {
    var value = t
}
```

创建类的实例时我们需要指定类型参数:

```
val box: Box<Int> = Box<Int>(1)
// 或者
val box = Box(1) // 编译器会进行类型推断，1 类型 Int，所以编译器知道我们说的是 Box<Int>。
```

以下实例向泛型类 Box 传入整型数据和字符串：

```
class Box<T>(t : T) {
    var value = t
}

fun main(args: Array<String>) {
    var boxInt = Box<Int>(10)
    var boxString = Box<String>("Runoob")

    println(boxInt.value)
    println(boxString.value)
}
```

输出结果为：

```
10
Runoob
```

定义泛型类型变量，可以完整地写明类型参数，如果编译器可以自动推定类型参数，也可以省略类型参数。

Kotlin 泛型函数的声明与 Java 相同，类型参数要放在函数名的前面：

```
fun <T> boxIn(value: T) = Box(value)
```

```
// 以下都是合法语句
val box4 = boxIn<Int>(1)
val box5 = boxIn(1)     // 编译器会进行类型推断
```

在调用泛型函数时，如果可以推断出类型参数，可以省略泛型参数。

以下实例创建了泛型函数 doPrintln，函数根据传入的不同类型做相应处理：

```
fun main(args: Array<String>) {
    val age = 23
    val name = "runoob"
    val bool = true

    doPrintln(age)    // 整型
    doPrintln(name)   // 字符串
    doPrintln(bool)   // 布尔型
}

fun <T> doPrintln(content: T) {

    when (content) {
        is Int -> println("整型数字为 $content")
        is String -> println("字符串转换为大写：${content.toUpperCase()}")
        else -> println("T 不是整型，也不是字符串")
    }
}
```

输出结果为：

```
整型数字为 23
字符串转换为大写：RUNOOB
T 不是整型，也不是字符串
```

泛型约束

我们可以使用泛型约束来设定一个给定参数允许使用的类型。

Kotlin 中使用 : 对泛型的的类型上限进行约束。

最常见的约束是上界(upper bound)：

```
fun <T : Comparable<T>> sort(list: List<T>) {
    // ……
}
```

Comparable 的子类型可以替代 T。 例如:

```
sort(listOf(1, 2, 3)) // OK。Int 是 Comparable<Int> 的子类型
sort(listOf(HashMap<Int, String>())) // 错误：HashMap<Int, String> 不是 Comparable<HashMap<Int, String>>
 的子类型
```

默认的上界是 Any?。

对于多个上界约束条件，可以用 where 子句：

```
fun <T> copyWhenGreater(list: List<T>, threshold: T): List<String>
    where T : CharSequence,
          T : Comparable<T> {
    return list.filter { it > threshold }.map { it.toString() }
}
```

型变

Kotlin 中没有通配符类型，它有两个其他的东西：声明处型变（declaration-site variance）与类型投影（type projections）。

声明处型变

声明处的类型变异使用协变注解修饰符：in、out，消费者 in, 生产者 out。

使用 out 使得一个类型参数协变，协变类型参数只能用作输出，可以作为返回值类型但是无法作为入参的类型：

```
// 定义一个支持协变的类
class Runoob<out A>(val a: A) {
    fun foo(): A {
        return a
    }
}

fun main(args: Array<String>) {
    var strCo: Runoob<String> = Runoob("a")
    var anyCo: Runoob<Any> = Runoob<Any>("b")
    anyCo = strCo
    println(anyCo.foo())   // 输出 a
}
```

in 使得一个类型参数逆变，逆变类型参数只能用作输入，可以作为入参的类型但是无法作为返回值的类型：

```
// 定义一个支持逆变的类
class Runoob<in A>(a: A) {
    fun foo(a: A) {
    }
}

fun main(args: Array<String>) {
```

```
    var strDCo = Runoob("a")
    var anyDCo = Runoob<Any>("b")
    strDCo = anyDCo
}
```

星号投射

有些时候, 你可能想表示你并不知道类型参数的任何信息, 但是仍然希望能够安全地使用它. 这里所谓"安全地使用"是指, 对泛型类型定义一个类型投射, 要求这个泛型类型的所有的实体实例, 都是这个投射的子类型。

对于这个问题, Kotlin 提供了一种语法, 称为 星号投射(star-projection):

- 假如类型定义为 Foo<out T> , 其中 T 是一个协变的类型参数, 上界(upper bound)为 TUpper ,Foo<> *等价于 Foo<out TUpper> . 它表示, 当 T 未知时, 你可以安全地从 Foo<>* 中 读取TUpper 类型的值.
- 假如类型定义为 Foo<in T> , 其中 T 是一个反向协变的类型参数, Foo<> *等价于 Foo<inNothing> . 它表示, 当 T 未知时, 你不能安全地向 Foo<>* 写入 任何东西.
- 假如类型定义为 Foo<T> , 其中 T 是一个协变的类型参数, 上界(upper bound)为 TUpper , 对于读取值的场合, Foo<*> 等价于 Foo<out TUpper> , 对于写入值的场合, 等价于 Foo<in Nothing> .

如果一个泛型类型中存在多个类型参数, 那么每个类型参数都可以单独的投射. 比如, 如果类型定义为interface Function<in T, out U> , 那么可以出现以下几种星号投射:

1. Function<*, String> , 代表 Function<in Nothing, String> ;
2. Function<Int, *> , 代表 Function<Int, out Any?> ;
3. Function<*, *> , 代表 Function<in Nothing, out Any?> .

注意: 星号投射与 Java 的原生类型(raw type)非常类似, 但可以安全使用

← Kotlin 数据类与密封类 Kotlin 枚举类 →

1 篇笔记 写笔记

关于星号投射，其实就是*代指了所有类型，相当于Any?
给文中补个例子方便理解：

```
class A<T>(val t: T, val t2 : T, val t3 : T)
class Apple(var name : String)
fun main(args: Array<String>) {
    //使用类
    val a1: A<*> = A(12, "String", Apple("苹果"))
    val a2: A<Any?> = A(12, "String", Apple("苹果"))   //和a1是一样的
    val apple = a1.t3    //参数类型为Any
    println(apple)
    val apple2 = apple as Apple   //强转成Apple类
```

```
    println(apple2.name)
    //使用数组
    val l:ArrayList<*> = arrayListOf("String",1,1.2f,Apple("苹果"))
    for (item in l){
        println(item)
    }
}
```

coding 5个月前 (10-24)