Kotlin 接口 Kotlin 数据类与密封类

Kotlin 扩展

Kotlin 可以对一个类的属性和方法进行扩展，且不需要继承或使用 Decorator 模式。

扩展是一种静态行为，对被扩展的类代码本身不会造成任何影响。

扩展函数

扩展函数可以在已有类中添加新的方法，不会对原类做修改，扩展函数定义形式：

```
fun receiverType.functionName(params){
    body
}
```

- receiverType：表示函数的接收者，也就是函数扩展的对象
- functionName：扩展函数的名称
- params：扩展函数的参数，可以为NULL

以下实例扩展 User 类 ：

```
class User(var name:String)

/**扩展函数**/
fun User.Print(){
    print("用户名 $name")
}

fun main(arg:Array<String>){
    var user = User("Runoob")
    user.Print()
}
```

实例执行输出结果为：

```
用户名 Runoob
```

下面代码为 MutableList 添加一个swap 函数：

```
// 扩展函数 swap,调换不同位置的值
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1]     //  this 对应该列表
    this[index1] = this[index2]
```

```
    this[index2] = tmp
}

fun main(args: Array<String>) {

    val l = mutableListOf(1, 2, 3)
    // 位置 0 和 2 的值做了互换
    l.swap(0, 2) // 'swap()' 函数内的 'this' 将指向 'l' 的值

    println(l.toString())
}
```

实例执行输出结果为：

```
[3, 2, 1]
```

this关键字指代接收者对象(receiver object)(也就是调用扩展函数时, 在点号之前指定的对象实例)。

扩展函数是静态解析的

扩展函数是静态解析的，并不是接收者类型的虚拟成员，在调用扩展函数时，具体被调用的的是哪一个函数，由调用函数的的对象表达式来决定的，而不是动态的类型决定的:

```
open class C

class D: C()

fun C.foo() = "c"   // 扩展函数 foo

fun D.foo() = "d"   // 扩展函数 foo

fun printFoo(c: C) {
    println(c.foo())  // 类型是 C 类
}

fun main(arg:Array<String>){
    printFoo(D())
}
```

实例执行输出结果为：

```
c
```

若扩展函数和成员函数一致，则使用该函数时，会优先使用成员函数。

```
class C {
    fun foo() { println("成员函数") }
}

fun C.foo() { println("扩展函数") }

fun main(arg:Array<String>){
    var c = C()
    c.foo()
}
```

实例执行输出结果为：

```
成员函数
```

扩展一个空对象

在扩展函数内， 可以通过 this 来判断接收者是否为 NULL,这样，即使接收者为 NULL,也可以调用扩展函数。例如:

```
fun Any?.toString(): String {
    if (this == null) return "null"
    // 空检测之后，“this”会自动转换为非空类型，所以下面的 toString()
    // 解析为 Any 类的成员函数
    return toString()
}
fun main(arg:Array<String>){
    var t = null
    println(t.toString())
}
```

实例执行输出结果为：

```
null
```

>扩展属性
除了函数，Kotlin 也支持属性对属性进行扩展:

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

扩展属性允许定义在类或者kotlin文件中，不允许定义在函数中。初始化属性因为属性没有后端字段（backing field），所以不允许被初始化，只能由显式提供的 getter/setter 定义。

```
val Foo.bar = 1 // 错误: 扩展属性不能有初始化器
```

扩展属性只能被声明为 val。

伴生对象的扩展

如果一个类定义有一个伴生对象 ，你也可以为伴生对象定义扩展函数和属性。

伴生对象通过"类名."形式调用伴生对象，伴生对象声明的扩展函数，通过用类名限定符来调用：

```
class MyClass {
    companion object { }  // 将被称为 "Companion"
}

fun MyClass.Companion.foo() {
    println("伴随对象的扩展函数")
}

val MyClass.Companion.no: Int
    get() = 10

fun main(args: Array<String>) {
    println("no:${MyClass.no}")
    MyClass.foo()
}
```

实例执行输出结果为：

```
no:10
伴随对象的扩展函数
```

扩展的作用域

通常扩展函数或属性定义在顶级包下:

```
package foo.bar

fun Baz.goo() { …… }
```

要使用所定义包之外的一个扩展, 通过import导入扩展的函数名进行使用:

```
package com.example.usage

import foo.bar.goo // 导入所有名为 goo 的扩展
                   // 或者
```

```
import foo.bar.*   // 从 foo.bar 导入一切

fun usage(baz: Baz) {
    baz.goo()
}
```

扩展声明为成员

在一个类内部你可以为另一个类声明扩展。

在这个扩展中，有个多个隐含的接受者，其中扩展方法定义所在类的实例称为分发接受者，而扩展方法的目标类型的实例称为扩展接受者。

```
class D {
    fun bar() { println("D bar") }
}

class C {
    fun baz() { println("C baz") }

    fun D.foo() {
        bar()   // 调用 D.bar
        baz()   // 调用 C.baz
    }

    fun caller(d: D) {
        d.foo()   // 调用扩展函数
    }
}

fun main(args: Array<String>) {
    val c: C = C()
    val d: D = D()
    c.caller(d)

}
```

实例执行输出结果为：

```
D bar
C baz
```

在 C 类内，创建了 D 类的扩展。此时，C 被成为分发接受者，而 D 为扩展接受者。从上例中，可以清楚的看到，在扩展函数中，可以调用派发接收者的成员函数。

假如在调用某一个函数，而该函数在分发接受者和扩展接受者均存在，则以扩展接收者优先，要引用分发接收者的成员你可以使用限定的 this 语法。

```
class D {
    fun bar() { println("D bar") }
}

class C {
    fun bar() { println("C bar") }  // 与 D 类 的 bar 同名

    fun D.foo() {
        bar()         // 调用 D.bar()，扩展接收者优先
        this@C.bar()  // 调用 C.bar()
    }

    fun caller(d: D) {
        d.foo()   // 调用扩展函数
    }
}

fun main(args: Array<String>) {
    val c: C = C()
    val d: D = D()
    c.caller(d)

}
```

实例执行输出结果为：

```
D bar
C bar
```

以成员的形式定义的扩展函数, 可以声明为 open , 而且可以在子类中覆盖. 也就是说, 在这类扩展函数的派 发过程中, 针对分发接受者是虚拟的(virtual), 但针对扩展接受者仍然是静态的。

```
open class D {
}

class D1 : D() {
}

open class C {
    open fun D.foo() {
        println("D.foo in C")
    }
```

```
    open fun D1.foo() {
        println("D1.foo in C")
    }

    fun caller(d: D) {
        d.foo()   // 调用扩展函数
    }
}

class C1 : C() {
    override fun D.foo() {
        println("D.foo in C1")
    }

    override fun D1.foo() {
        println("D1.foo in C1")
    }
}


fun main(args: Array<String>) {
    C().caller(D())   // 输出 "D.foo in C"
    C1().caller(D())  // 输出 "D.foo in C1" —— 分发接收者虚拟解析
    C().caller(D1())  // 输出 "D.foo in C" —— 扩展接收者静态解析

}
```

实例执行输出结果为：

```
D.foo in C
D.foo in C1
D.foo in C
```

← Kotlin 接口　　Kotlin 数据类与密封类 →

- （1）类内扩展的伴随对象函数和类外扩展的伴随对象可以同名，它们是两个独立的函数，互不影响；
- （2）当类内扩展的伴随对象函数和类外扩展的伴随对象同名时，类内的其它函数优先引用类内扩展的伴随对象函数，即对于类内其它成员函数来说，类内扩展屏蔽类外扩展；
- （3）类内扩展的伴随对象函数只能被类内的函数引用，不能被类外的函数和伴随对象内的函数引用；
- （4）类外扩展的伴随对象函数可以被伴随对象内的函数引用，；

  例如以下代码：

```
class MyClass {
    companion object {
        val myClassField1: Int = 1
        var myClassField2 = "this is myClassField2"
        fun companionFun1() {
            println("this is 1st companion function.")
            foo()
        }
        fun companionFun2() {
            println("this is 2st companion function.")
            companionFun1()
        }
    }
    fun MyClass.Companion.foo() {
        println("伴随对象的扩展函数（内部）")
    }
    fun test2() {
        MyClass.foo()
    }
    init {
        test2()
    }
}
val MyClass.Companion.no: Int
    get() = 10
fun MyClass.Companion.foo() {
    println("foo 伴随对象外部扩展函数")
}
fun main(args: Array<String>) {
    println("no:${MyClass.no}")
    println("field1:${MyClass.myClassField1}")
    println("field2:${MyClass.myClassField2}")
    MyClass.foo()
    MyClass.companionFun2()
}
```

  运行结果：

```
no:10
field1:1
field2:this is myClassField2
foo 伴随对象外部扩展函数
this is 2st companion function.
this is 1st companion function.
foo 伴随对象外部扩展函数
```

applixy 8个月前 (07-27)