扩展

Kotlin 能够扩展一个类的新功能而无需继承该类或者使用像装饰者这样的设计模式。 这通过叫做 *扩展* 的特殊声明完成。 例如，你可以为一个你不能修改的、来自第三方库中的类编写一个新的函数。 这个新增的函数就像那个原始类本来就有的函数一样，可以用普通的方法调用。 这种机制称为 *扩展函数* 。此外，也有 *扩展属性* ， 允许你为一个已经存在的类添加新的属性。

扩展函数

声明一个扩展函数，我们需要用一个 *接收者类型* 也就是被扩展的类型来作为他的前缀。 下面代码为 `MutableList<Int>` 添加一个 `swap` 函数：

```
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
}
```

这个 `this` 关键字在扩展函数内部对应到接收者对象（传过来的在点符号前的对象）现在，我们对任意 `MutableList<Int>` 调用该函数了：

```
val list = mutableListOf(1, 2, 3)
list.swap(0, 2) // “swap()”内部的“this”会保存“list”的值
```

当然，这个函数对任何 `MutableList<T>` 起作用，我们可以泛化它：

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // “this”对应该列表
    this[index1] = this[index2]
    this[index2] = tmp
}
```

为了在接收者类型表达式中使用泛型，我们要在函数名前声明泛型参数。 参见泛型函数。

扩展是静态解析的

扩展不能真正的修改他们所扩展的类。通过定义一个扩展，你并没有在一个类中插入新成员， 仅仅是可以通过该类型的变量用点表达式去调用这个新函数。

我们想强调的是扩展函数是静态分发的，即他们不是根据接收者类型的虚方法。 这意味着调用的扩展函数是由函数调用所在的表达式的类型来决定的， 而不是由表达式运行时求值结果决定的。例如：

```
open class Shape

class Rectangle: Shape()

fun Shape.getName() = "Shape"

fun Rectangle.getName() = "Rectangle"

fun printClassName(s: Shape) {
    println(s.getName())
}

printClassName(Rectangle())
```

Target platform: JVM Running on kotlin v. 1.7.10

这个例子会输出 "*Shape*"，因为调用的扩展函数只取决于参数 `s` 的声明类型，该类型是 `Shape` 类。

如果一个类定义有一个成员函数与一个扩展函数，而这两个函数又有相同的接收者类型、 相同的名字，并且都适用给定的参数，这种情况**总是取成员函数**。 例如：

```
class Example {
    fun printFunctionType() { println("Class method") }
}

fun Example.printFunctionType() { println("Extension function") }

Example().printFunctionType()
```

Target platform: JVM Running on kotlin v. 1.7.10

这段代码输出“*Class method*”。

当然，扩展函数重载同样名字但不同签名成员函数也完全可以：

```
class Example {
    fun printFunctionType() { println("Class method") }
}

fun Example.printFunctionType(i: Int) { println("Extension function")

Example().printFunctionType(1)
```

Target platform: JVM Running on kotlin v. 1.7.10

可空接收者

注意可以为可空的接收者类型定义扩展。这样的扩展可以在对象变量上调用， 即使其值为 null，并且可以在函数体内检测 `this == null` ，这能让你在没有检测 null 的时候调用 Kotlin 中的toString()：检测发生在扩展函数的内部。

```
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()
}
```

扩展属性

与函数类似，Kotlin 支持扩展属性：

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

注意：由于扩展没有实际的将成员插入类中，因此对扩展属性来说幕后字段是无效的。这就是为什么**扩展属性不能有初始化器**。他们的行为只能由显式提供的 getters/setters 定义。

例如:

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

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与属性。就像伴生对象的常规成员一样， 可以只使用类名作为限定符来调用伴生对象的扩展成员：

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

fun MyClass.Companion.printCompanion() { println("companion") }

fun main() {
    MyClass.printCompanion()
}
```

Target platform: JVM Running on kotlin v. 1.7.10

扩展的作用域

大多数时候我们在顶层定义扩展——直接在包里：

```
package org.example.declarations

fun List<String>.getLongestString() { /*……*/}
```

要使用所定义包之外的一个扩展，我们需要在调用方导入它：

```
package org.example.usage

import org.example.declarations.getLongestString

fun main() {
    val list = listOf("red", "green", "blue")
    list.getLongestString()
}
```

更多信息参见导入

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。在这样的扩展内部，有多个 *隐式接收者* —— 其中的对象成员可以无需通过限定符访问。扩展声明所在的类的实例称为 *分发接收者*，扩展方法调用所在的接收者类型的实例称为 *扩展接收者* 。

```
class Host(val hostname: String) {
    fun printHostname() { print(hostname) }
}

class Connection(val host: Host, val port: Int) {
     fun printPort() { print(port) }

     fun Host.printConnectionString() {
         printHostname()   // 调用 Host.printHostname()
         print(":")
         printPort()   // 调用 Connection.printPort()
     }

     fun connect() {
         /*……*/
         host.printConnectionString()   // 调用扩展函数
     }
}

fun main() {
    Connection(Host("kotl.in"), 443).connect()
    //Host("kotl.in").printConnectionString(443)  // 错误，该扩展函数在
}
```

Target platform: JVM Running on kotlin v. 1.7.10

对于分发接收者与扩展接收者的成员名字冲突的情况，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 Connection {
    fun Host.getConnectionString() {
        toString()         // 调用 Host.toString()
        this@Connection.toString()  // 调用 Connection.toString()
    }
}
```

声明为成员的扩展可以声明为 `open` 并在子类中覆盖。这意味着这些函数的分发对于分发接收者类型是虚拟的，但对于扩展接收者类型是静态的。

```
open class Base { }

class Derived : Base() { }

open class BaseCaller {
    open fun Base.printFunctionInfo() {
        println("Base extension function in BaseCaller")
    }
```

```
    open fun Derived.printFunctionInfo() {
        println("Derived extension function in BaseCaller")
    }

    fun call(b: Base) {
        b.printFunctionInfo()   // 调用扩展函数
    }
}

class DerivedCaller: BaseCaller() {
    override fun Base.printFunctionInfo() {
        println("Base extension function in DerivedCaller")
    }

    override fun Derived.printFunctionInfo() {
        println("Derived extension function in DerivedCaller")
    }
}

fun main() {
    BaseCaller().call(Base())   // “Base extension function in BaseCa
    DerivedCaller().call(Base())  // “Base extension function in Deri
    DerivedCaller().call(Derived())  // “Base extension function in D
}
```

Target platform: JVM Running on kotlin v. 1.7.10

关于可见性的说明

扩展的可见性与相同作用域内声明的其他实体的可见性相同。例如：

— 在文件顶层声明的扩展可以访问同一文件中的其他 `private` 顶层声明；

— 如果扩展是在其接收者类型外部声明的，那么该扩展不能访问接收者的 `private` 成员。