Kotlin 学习之object关键字

天蝎_少 关注

0.248 2018.12.06 13:50:16 字数 914 阅读 8,421

1.什么是object关键字

Kotlobject关键字在多种情况下出现，但是它都遵循同样的核理念,这个关键字定义一个类并同时创建个实例(也就是一个类对象)

2.object关键字在kotlin中的用例

2.1对象声明

java中使用单例模式,需要三步: 私有化构造方法,创建一个该类的实例,提供一个获取该实例的方法.而在kotlin中只需要object一个关键字即可.对,就是这么简单!用object关键字进行对象声明后,就可以用类名.方法 的方式调用方法.这也印证了定义中说明的声明一个类,同时声明该类的一个实例.

示例代码如下:

```
object ObjectKeyTest {

    override fun toString(): String {

        return "我是一个ObjectKeyTest"

       }

}

    fun main(args: Array) {

         println(ObjectKeyTest.toString())

    }
```

对象声明一样可以继承类和接口,这个一般在你的实现并不包含任何实现的时候很有用.例如你实现一个比较器的接口用于比较:

```
object FilePathComparator : Comparator<File> {

    override fun compare(file1 : File, file2: File): Int {
        return file1.path!!.compareTo(file2.path)
    }
}

fun main(args: Array<String>) {
    println(FilePathComparator.compare(File("/User"),File("/user")))
}
```

同样,可以在类中使用声明对象,这样的对象在类中也是单一实例存在的,kotlin中可以理解成一个类中的单一实例,不随宿主类对象的不同而变化.这种类中嵌套一个类,用object关键字声明时一样可以用类名.对象的方法,用法如下:

```
class FileUtils {

    object FilePathComparator : Comparator<File> {
        override fun compare(file1: File, file2: File): Int {
            return file1.path!!.compareTo(file2.path)
        }
```

```
    }

}

fun main(args: Array<String>) {
    val  fileUtils : FileUtils = FileUtils()
    FileUtils.FilePathComparator
    println(FileUtils.FilePathComparator.compare(File("/User"),File("/user")))
}
```

2.2使用object声明伴生对象

在kotlin没有static关键字的概念,假如要使用静态方法怎么办呢?那么可以使用伴生对象,或者使用顶层函数,不过,顶层函数不能访问类中的私有成员,但伴生对象可以,用法如下:

```
class CompanionTest {

    private val string = "伴生对象可以访问我"

    companion object {

        fun printStr() {
            println(CompanionTest().string)
        }

    }

}

fun main(args: Array<String>) {
    CompanionTest.printStr()
}
```

如上所示,string作为一个私有变量,伴生对象是有访问权限的.但是这里要注意,直接访问string这个类成员是不可以的,因为伴生对象的方法相当于一个static静态方法,而string是非静态的,所以要先创建对象才能访问.
伴生对象在工厂方法中使用是其一个很好的实践,可以替代多构造方法的构造函数,例如有如下示例:

```
class UserBean {

    var nickName : String

    constructor(id : Int) {
        nickName = "$id"
    }

    constructor(email : String) {
        nickName = email.plus("*")
    }

}
```

然后使用伴生对象创建工厂方法,并且把构造方法变成私有,示例如下:

```

//构造方法标记为私有,防止外部调用
class UserBean1 private constructor(val userName : String){

    companion object {

        fun newIdUser(id : Int) = UserBean1("$id")

        fun newEmailUser(email : String) = UserBean1(email.plus("abc"))

    }

}

fun main(args: Array<String>) {
    var User1 = UserBean1.newIdUser(1)
    var User2 = UserBean1.newEmailUser("abc@sina.com")
}
```

如果需要创建不同的对象,可以使用上面的工厂方法创建,但是这里要注意,伴生对象是不可以重写的,如果需要扩展,还是采用多构造方法比较好.
同时伴生对象还可以有名字:

```
class UserBean2 {

    companion object Loader {

        fun getData() : String {
            return "userbean2"
        }

    }

}

fun main(args: Array<String>) {
    println(UserBean2.getData())
}
```

伴生对象还可以实现接口:

```
interface Loader {

    fun getData() : String

}

class UserBean3 {

    companion object MyLoader : Loader {

        override fun getData() : String {
            return "userbean3"
        }
    }

}

fun main(args: Array<String>) {
    println(UserBean3.getData())
}
```

还可以直接将类名当昨对象当做该接口的实现对象进行传递

```
interface Loader {

    fun getData() : String

}

class UserBean3 {

    companion object MyLoader : Loader {

        override fun getData() : String {
            return "userbean3"
        }
    }

}

fun loadUserData(loader: Loader) {
    println(loader.getData())
}

fun main(args: Array<String>) {
    loadUserData(UserBean3)
}
```

你还可以为伴生对象定义扩展函数,因为有时你想将某些数据处理与原类的核心逻辑分离,这个时候可以使用扩展函数,而伴生对象也支持扩展函数.

先定义一个伴生对象:

```
class UserBean4 {

    //声明一个空的伴生对象,为定义扩展函数做准备
    companion object {

    }

}
```

再为该对象定义扩展函数:

```

//注意这里没有定义伴生对象名称,直接用Companion引用
fun UserBean4.Companion.getType() : String {
    return "UserName4"
}

fun main(args: Array<String>) {
    val data = UserBean4.getType()
    println(data)
}
```

object关键字还可以用来声明匿名内部类,kotlin中用匿名对象的方式代替了java中的匿名内部类的使用方式,并且,该匿名内部类不像java只能实现一个接口或继承一个对象,它可以实现多个接口,用法如下:

```
interface MyInterface1 {

    fun getSomething()

}

interface MyInterface2 {

    fun doSomething()

}

fun doing(myInterface1: MyInterface1) {
    myInterface1.getSomething()
}

fun main(args: Array<String>) {
    doing(object : MyInterface1,MyInterface2 {

        override fun getSomething() {

        }

        override fun doSomething() {

        }

    })
}
```

同时,它还可以有名字,也就是像Java一样定义成一个成员:

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

val MyInterfaceImpl = object : MyInterface1 {

    override fun getSomething() {

    }

}
```

它还可以访问外部函数创建的变量,并且不用标识为final类型:

```
class MyTest {
    fun main(args: Array<String>) {
        var count : Int = 0
        doing(object : MyInterface1 {

            override fun getSomething() {
                val arg0 = args[0]
                count ++
            }

        })
    }
}
```