对象表达式与对象声明

有时候，我们需要创建一个对某个类做了轻微改动的类的对象，而不用为之显式声明新的子类。 Kotlin 用*对象表达式*和*对象声明*处理这种情况。

对象表达式

要创建一个继承自某个（或某些）类型的匿名类的对象，我们会这么写：

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { /*……*/ }

    override fun mouseEntered(e: MouseEvent) { /*……*/ }
})
```

如果超类型有一个构造函数，则必须传递适当的构造函数参数给它。 多个超类型可以由跟在冒号后面的逗号分隔的列表指定：

```
open class A(x: Int) {
    public open val y: Int = x
}

interface B { /*……*/ }

val ab: A = object : A(1), B {
    override val y = 15
}
```

任何时候，如果我们只需要“一个对象而已”，并不需要特殊超类型，那么我们可以简单地写：

```
fun foo() {
    val adHoc = object {
        var x: Int = 0
        var y: Int = 0
    }
    print(adHoc.x + adHoc.y)
}
```

请注意，匿名对象可以用作只在本地和私有作用域中声明的类型。如果你使用匿名对象作为公有函数的返回类型或者用作公有属性的类型，那么该函数或属性的实际类型会是匿名对象声明的超类型，如果你没有声明任何超类型，就会是 Any 。在匿名对象中添加的成员将无法访问。

```
class C {
    // 私有函数，所以其返回类型是匿名对象类型
    private fun foo() = object {
        val x: String = "x"
    }

    // 公有函数，所以其返回类型是 Any
    fun publicFoo() = object {
        val x: String = "x"
    }

    fun bar() {
        val x1 = foo().x        // 没问题
        val x2 = publicFoo().x  // 错误：未能解析的引用“x”
    }
}
```

对象表达式中的代码可以访问来自包含它的作用域的变量。

```
fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0

    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // ……
}
```

对象声明

对象声明

单例模式在一些场景中很有用， 而 Kotlin（继 Scala 之后）使单例声明变得很容易：

```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ……
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ……
}
```

这称为*对象声明*。并且它总是在 `object` 关键字后跟一个名称。 就像变量声明一样，对象声明不是一个表达式，不能用在赋值语句的右边。

对象声明的初始化过程是线程安全的并且在首次访问时进行。

如需引用该对象，我们直接使用其名称即可：

```
DataProviderManager.registerDataProvider(……)
```

这些对象可以有超类型：

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { …… }

    override fun mouseEntered(e: MouseEvent) { …… }
}
```

注意：对象声明不能在局部作用域（即直接嵌套在函数内部），但是它们可以嵌套到其他对象声明或非内部类中。

伴生对象

类内部的对象声明可以用 `companion` 关键字标记：

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

该伴生对象的成员可通过只使用类名作为限定符来调用：

```
val instance = MyClass.create()
```

可以省略伴生对象的名称，在这种情况下将使用名称 `Companion`：

```
class MyClass {
    companion object { }
}

val x = MyClass.Companion
```

其自身所用的类的名称（不是另一个名称的限定符）可用作对该类的伴生对象 （无论是否具名）的引用：

```
class MyClass1 {
    companion object Named { }
}

val x = MyClass1

class MyClass2 {
    companion object { }
}

val y = MyClass2
```

请注意，即使伴生对象的成员看起来像其他语言的静态成员，在运行时他们仍然是真实对象的实例成员，而且，例如还可以实现接口：

```
interface Factory<T> {
    fun create(): T
}
```

```
class MyClass {
    companion object : Factory<MyClass> {
        override fun create(): MyClass = MyClass()
    }
}

val f: Factory<MyClass> = MyClass
```

当然，在 JVM 平台，如果使用 `@JvmStatic` 注解，你可以将伴生对象的成员生成为真正的静态方法和字段。更详细信息请参见Java 互操作性一节 。

对象表达式和对象声明之间的语义差异

对象表达式和对象声明之间有一个重要的语义差别：

— 对象表达式是在使用他们的地方**立即**执行（及初始化）的；

— 对象声明是在第一次被访问到时**延迟**初始化的；

— 伴生对象的初始化是在相应的类被加载（解析）时，与 Java 静态初始化器的语义相匹配。