Kotlin 对象表达式和对象声明

kotlin 委托

委托模式是软件设计模式中的一项基本技巧。在委托模式中，有两个对象参与处理同一个请求，接受请求的对象将请求委托给另一个对象来处理。

Kotlin 直接支持委托模式，更加优雅，简洁。Kotlin 通过关键字 by 实现委托。

类委托

类的委托即一个类中定义的方法实际是调用另一个类的对象的方法来实现的。

以下实例中派生类 Derived 继承了接口 Base 所有方法，并且委托一个传入的 Base 类的对象来执行这些方法。

```
// 创建接口
interface Base {
    fun print()
}

// 实现此接口的被委托的类
class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

// 通过关键字 by 建立委托类
class Derived(b: Base) : Base by b

fun main(args: Array<String>) {
    val b = BaseImpl(10)
    Derived(b).print() // 输出 10
}
```

在 Derived 声明中，by 子句表示，将 b 保存在 Derived 的对象实例内部，而且编译器将会生成继承自 Base 接口的所有方法，并将调用转发给 b。

属性委托

属性委托指的是一个类的某个属性值不是在类中直接进行定义，而是将其托付给一个代理类，从而实现对该类的属性统一管理。

属性委托语法格式：

```
val/var <属性名>: <类型> by <表达式>
```

- var/val：属性类型(可变/只读)
- 属性名：属性名称

- 类型：属性的数据类型
- 表达式：委托代理类

by 关键字之后的表达式就是委托, 属性的 get() 方法(以及set() 方法)将被委托给这个对象的 getValue() 和 setValue() 方法。属性委托不必实现任何接口, 但必须提供 getValue() 函数(对于 var属性,还需要 setValue() 函数)。

定义一个被委托的类

该类需要包含 getValue() 方法和 setValue() 方法，且参数 thisRef 为进行委托的类的对象，prop 为进行委托的属性的对象。

```
import kotlin.reflect.KProperty
// 定义包含属性委托的类
class Example {
    var p: String by Delegate()
}

// 委托的类
class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, 这里委托了 ${property.name} 属性"
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
        println("$thisRef 的 ${property.name} 属性赋值为 $value")
    }
}
fun main(args: Array<String>) {
    val e = Example()
    println(e.p)     // 访问该属性，调用 getValue() 函数

    e.p = "Runoob"   // 调用 setValue() 函数
    println(e.p)
}
```

输出结果为：

```
Example@433c675d, 这里委托了 p 属性
Example@433c675d 的 p 属性赋值为 Runoob
Example@433c675d, 这里委托了 p 属性
```

标准委托

Kotlin 的标准库中已经内置了很多工厂方法来实现属性的委托。

延迟属性 Lazy

lazy() 是一个函数, 接受一个 Lambda 表达式作为参数, 返回一个 Lazy <T> 实例的函数，返回的实例可以作为实现延迟属性的委托： 第一次调用 get() 会执行已传递给 lazy() 的 lamda 表达式并记录结果， 后续调用 get() 只是返回记录的结果。

```
val lazyValue: String by lazy {
    println("computed!")     // 第一次调用输出，第二次调用不执行
    "Hello"
}

fun main(args: Array<String>) {
    println(lazyValue)   // 第一次执行，执行两次输出表达式
    println(lazyValue)   // 第二次执行，只输出返回值
}
```

执行输出结果：

```
computed!
Hello
Hello
```

可观察属性 Observable

observable 可以用于实现观察者模式。

Delegates.observable() 函数接受两个参数: 第一个是初始化值, 第二个是属性值变化事件的响应器(handler)。

在属性赋值后会执行事件的响应器(handler)，它有三个参数：被赋值的属性、旧值和新值：

```
import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("初始值") {
        prop, old, new ->
        println("旧值：$old -> 新值：$new")
    }
}

fun main(args: Array<String>) {
    val user = User()
    user.name = "第一次赋值"
    user.name = "第二次赋值"
}
```

执行输出结果：

```
旧值：初始值 -> 新值：第一次赋值
旧值：第一次赋值 -> 新值：第二次赋值
```

把属性储存在映射中

一个常见的用例是在一个映射（map）里存储属性的值。 这经常出现在像解析 JSON 或者做其他"动态"事情的应用中。 在这种情况下，你可以使用映射实例自身作为委托来实现委托属性。

```
class Site(val map: Map<String, Any?>) {
    val name: String by map
    val url: String  by map
}

fun main(args: Array<String>) {
    // 构造函数接受一个映射参数
    val site = Site(mapOf(
        "name" to "菜鸟教程",
        "url"  to "www.runoob.com"
    ))

    // 读取映射值
    println(site.name)
    println(site.url)
}
```

执行输出结果：

```
菜鸟教程
www.runoob.com
```

如果使用 var 属性，需要把 Map 换成 MutableMap：

```
class Site(val map: MutableMap<String, Any?>) {
    val name: String by map
    val url: String by map
}

fun main(args: Array<String>) {

    var map:MutableMap<String, Any?> = mutableMapOf(
            "name" to "菜鸟教程",
            "url" to "www.runoob.com"
    )

    val site = Site(map)

    println(site.name)
    println(site.url)
```

```
    println("--------------")
    map.put("name", "Google")
    map.put("url", "www.google.com")

    println(site.name)
    println(site.url)

}
```

执行输出结果：

```
菜鸟教程
www.runoob.com
--------------
Google
www.google.com
```

Not Null

notNull 适用于那些无法在初始化阶段就确定属性值的场合。

```
class Foo {
    var notNullBar: String by Delegates.notNull<String>()
}

foo.notNullBar = "bar"
println(foo.notNullBar)
```

需要注意，如果属性在赋值前就被访问的话则会抛出异常。

局部委托属性

你可以将局部变量声明为委托属性。 例如，你可以使一个局部变量惰性初始化：

```
fun example(computeFoo: () -> Foo) {
    val memoizedFoo by lazy(computeFoo)

    if (someCondition && memoizedFoo.isValid()) {
        memoizedFoo.doSomething()
    }
}
```

memoizedFoo 变量只会在第一次访问时计算。 如果 someCondition 失败，那么该变量根本不会计算。

属性委托要求

对于只读属性(也就是说val属性), 它的委托必须提供一个名为getValue()的函数。该函数接受以下参数：

- thisRef —— 必须与属性所有者类型（对于扩展属性——指被扩展的类型）相同或者是它的超类型
- property —— 必须是类型 KProperty<*> 或其超类型

这个函数必须返回与属性相同的类型（或其子类型）。
对于一个值可变(mutable)属性(也就是说,var 属性),除 getValue()函数之外,它的委托还必须 另外再提供一个名为setValue()的函数, 这个函数接受以下参数:

- property —— 必须是类型 KProperty<*> 或其超类型
- new value —— 必须和属性同类型或者是它的超类型。

翻译规则

在每个委托属性的实现的背后，Kotlin 编译器都会生成辅助属性并委托给它。 例如，对于属性 prop，生成隐藏属性 prop$delegate，而访问器的代码只是简单地委托给这个附加属性：

```
class C {
    var prop: Type by MyDelegate()
}

// 这段是由编译器生成的相应代码：
class C {
    private val prop$delegate = MyDelegate()
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

Kotlin 编译器在参数中提供了关于 prop 的所有必要信息：第一个参数 this 引用到外部类 C 的实例而 this::prop 是 KProperty 类型的反射对象，该对象描述 prop 自身。

提供委托

通过定义 provideDelegate 操作符，可以扩展创建属性实现所委托对象的逻辑。 如果 by 右侧所使用的对象将 provideDelegate 定义为成员或扩展函数，那么会调用该函数来 创建属性委托实例。
provideDelegate 的一个可能的使用场景是在创建属性时（而不仅在其 getter 或 setter 中）检查属性一致性。
例如，如果要在绑定之前检查属性名称，可以这样写：

```
class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(
            thisRef: MyUI,
```

```
            prop: KProperty<*>
    ): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        // 创建委托
    }

    private fun checkProperty(thisRef: MyUI, name: String) { …… }
}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { …… }

class MyUI {
    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

provideDelegate 的参数与 getValue 相同：

- thisRef —— 必须与 属性所有者 类型（对于扩展属性——指被扩展的类型）相同或者是它的超类型
- property —— 必须是类型 KProperty<*> 或其超类型。

在创建 MyUI 实例期间，为每个属性调用 provideDelegate 方法，并立即执行必要的验证。
如果没有这种拦截属性与其委托之间的绑定的能力，为了实现相同的功能， 你必须显式传递属性名，这不是很方便：

```
// 检查属性名称而不使用“provideDelegate”功能
class MyUI {
    val image by bindResource(ResourceID.image_id, "image")
    val text by bindResource(ResourceID.text_id, "text")
}

fun <T> MyUI.bindResource(
        id: ResourceID<T>,
        propertyName: String
): ReadOnlyProperty<MyUI, T> {
   checkProperty(this, propertyName)
   // 创建委托
}
```

在生成的代码中，会调用 provideDelegate 方法来初始化辅助的 prop$delegate 属性。 比较对于属性声明 val prop: Type by MyDelegate() 生成的代码与 上面（当 provideDelegate 方法不存在时）生成的代码：

```
class C {
    var prop: Type by MyDelegate()
}

// 这段代码是当“provideDelegate”功能可用时
```

```
// 由编译器生成的代码：
class C {
    // 调用“provideDelegate”来创建额外的“delegate”属性
    private val prop$delegate = MyDelegate().provideDelegate(this, this::prop)
    val prop: Type
        get() = prop$delegate.getValue(this, this::prop)
}
```

请注意，provideDelegate 方法只影响辅助属性的创建，并不会影响为 getter 或 setter 生成的代码。

← Kotlin 对象表达式和对象声明