链滴

Kotlin 实战 | 使用 DSL 构建结构化 API 去掉冗余的接口方法

作者：lzlyy

原文链接：https://ld246.com/article/1592988250158

来源网站：链滴

即使不需要 Java 接口中的某些方法，也必须将其implements，然后保持其为空实现，傻傻地处在那
利用 Kotlin 的 DSL 可以只实现自己感兴趣的方法。

（这篇将在上一篇的代码基础上新增功能，并利用自定义的 DSL 来简化代码。）

引子

上篇中利用 apply()语法来简化组合动画的构建过程，代码如下：

```
val span = 300
AnimatorSet().apply {
    playTogether(
            ObjectAnimator.ofPropertyValuesHolder(
                    tvTitle,
                    PropertyValuesHolder.ofFloat("alpha", 0f, 1.0f),
                    PropertyValuesHolder.ofFloat("translationY", 0f, 100f)).apply {
                interpolator = AccelerateInterpolator()
                duration = span
            },
            ObjectAnimator.ofPropertyValuesHolder(
                    ivAvatar,
                    PropertyValuesHolder.ofFloat("alpha", 1.0f, 0f),
                    PropertyValuesHolder.ofFloat("translationY", 0f,100f)).apply {
                interpolator = AccelerateInterpolator()
                duration = span
            }
    )
    start()
}
```

如果动画的时间被拉长，需要在其暂停时显示 toast 提示，并且在结束时展示视图A，代码需做如下改：

```
val span = 5000
AnimatorSet().apply {
    playTogether(
            ObjectAnimator.ofPropertyValuesHolder(
                    tvTitle,
                    PropertyValuesHolder.ofFloat("alpha", 0f, 1.0f),
                    PropertyValuesHolder.ofFloat("translationY", 0f, 100f)).apply {
                interpolator = AccelerateInterpolator()
                duration = span
            },
            ObjectAnimator.ofPropertyValuesHolder(
                    ivAvatar,
                    PropertyValuesHolder.ofFloat("alpha", 1.0f, 0f),
                    PropertyValuesHolder.ofFloat("translationY", 0f,100f)).apply {
                interpolator = AccelerateInterpolator()
                duration = span
            }
    )
    addPauseListener(object :Animator.AnimatorPauseListener{
        override fun onAnimationPause(animation: Animator?) {
```

```
            Toast.makeText(context,"pause",Toast.LENGTH_SHORT).show()
        }

        override fun onAnimationResume(animation: Animator?) {
        }

    })
    addListener(object : Animator.AnimatorListener{
        override fun onAnimationRepeat(animation: Animator?) {
        }

        override fun onAnimationEnd(animation: Animator?) {
            showA()
        }

        override fun onAnimationCancel(animation: Animator?) {
        }

        override fun onAnimationStart(animation: Animator?) {
        }
    })
    start()
}
```

这一段 apply()有点过长了，严重降低了它的可读性。罪魁祸首是 java 接口。虽然只用到接口中的一方法，但却必须将其余的方法保留空实现。

有没有什么办法只实现想要的方法，去掉不用的方法?

利用 kotlin 的自定义 DSL 就可以实现。

DSL

DSL = domain specific language，即“特定领域语言”，与它对应的一个概念叫“通用编程语言，通用编程语言有一系列完善的能力来解决几乎所有能被计算机解决的问题，像 Java 就属于这种类。而特定领域语言只专注于特定的任务，比如 SQL 只专注于操纵数据库，HTML 只专注于表述超文。

既然通用编程语言能够解决所有的问题，那为啥还需要特定领域语言？因为它可以使用比通用编程语中等价代码更紧凑的语法来表达特定领域的操作。比如当执行一条 SQL 语句时，不需要从声明一个及其方法开始。

更紧凑的语法意味着更简洁的 API。应用程序中每个类都提供了其他类与之交互的可能性，确保这些互易于理解并可以简洁地表达，对于软件的可维护性至关重要。

DSL 有一个普通API不具备特征：DSL 具有结构。而 带接收者的lambda使得构建结构化的 API 变得易。

带接收者的 lambda

它是一种特殊的 lambda，是 kotlin 中特有的。可以把它理解成“为接收者声明的一个匿名扩展函数。（扩展函数是一种在类体外为类添加功能的特性）

带接收者的lambda的函数体除了能访问其所在类的成员外，还能访问接收者的所有非私有成员，这特性是它能够轻松地构建结构。

当带接收者的 lambda 配合高阶函数时，构建结构化的 API 就变得易如反掌。

高阶函数

1. 它是一种特殊的函数，它的参数或者返回值是另一个函数。

比如集合的扩展函数 filter()就是一个高阶函数：

```
//'filter的参数是一个带接收的lambda'
public inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T> {
    return filterTo(ArrayList<T>(), predicate)
}
```

可以使用它来过滤集合中的元素：

```
students.filter { age > 18 }
```

这样就是一种结构化 API 的调用（在 java 中看不到），虽然这种结构得益于 kotlin 的一个约定（如函数只有一个参数且它是 lambda，则可以省略函数参数列表的括号）。但更关键的是 lambda 的内，得益于 带接收者的lambda，age > 18运行在一个和其调用方不同的 上下文中，在这个上下文中可以轻松的访问到 Student的成员 Student.ag （ 指向 age 时可以省略 this ）

让我们使用这样的技巧来解决“必须实现java所有接口”的问题。

构建 DSL 解决 java 接口问题

新建类用于存放接口中各个方法的实现

```
class AnimatorListenerImpl {
    var onRepeat: ((Animator) -> Unit)? = null
    var onEnd: ((Animator) -> Unit)? = null
    var onCancel: ((Animator) -> Unit)? = null
    var onStart: ((Animator) -> Unit)? = null
}
```

它包含四个成员，每个成员的类型都是 函数类型。看一下 Animator.AnimatorListener的定义就能解 AnimatorListenerImpl的用意：

```
public static interface AnimatorListener {
    void onAnimationStart(Animator animation);
    void onAnimationEnd(Animator animation);
    void onAnimationCancel(Animator animation);
    void onAnimationRepeat(Animator animation);
}
```

该接口中的每个方法都接收一个 Animator参数并返回空值，用 lambda 可以表达成 (Animator) -> nit。所以 AnimatorListenerImpl将接口中的四个方法的实现都保存在函数变量中，并且实现是可空。

2. 为 Animator 定义一个高阶扩展函数

```
fun AnimatorSet.addListener(action: AnimatorListenerImpl.() -> Unit) {
    AnimatorListenerImpl().apply { action }.let { builder ->
        //'将回调实现委托给AnimatorListenerImpl的函数类型变量'
        addListener(object : Animator.AnimatorListener {
            override fun onAnimationRepeat(animation: Animator?) {
                animation?.let { builder.onRepeat?.invoke(animation) }
            }

            override fun onAnimationEnd(animation: Animator?) {
                animation?.let { builder.onEnd?.invoke(animation) }
            }

            override fun onAnimationCancel(animation: Animator?) {
                animation?.let { builder.onCancel?.invoke(animation) }
            }

            override fun onAnimationStart(animation: Animator?) {
                animation?.let { builder.onStart?.invoke(animation) }
            }
        })
    }
}
```

为 Animator定义了扩展函数 addListener()，该函数接收一个带接收者的lambda action。

扩展函数体中构建了 AnimatorListenerImpl实例并紧接着应用了 action，最后为 Animator设置动监听器并将回调的实现委托给 AnimatorListenerImpl中的函数类型变量。

3. 使用自定义的 DSL 将本文开头的代码改写：

```
val span = 5000
AnimatorSet().apply {
    playTogether(
            ObjectAnimator.ofPropertyValuesHolder(
                    tvTitle,
                    PropertyValuesHolder.ofFloat("alpha", 0f, 1.0f),
                    PropertyValuesHolder.ofFloat("translationY", 0f, 100f)).apply {
                interpolator = AccelerateInterpolator()
                duration = span
            },
            ObjectAnimator.ofPropertyValuesHolder(
                    ivAvatar,
                    PropertyValuesHolder.ofFloat("alpha", 1.0f, 0f),
                    PropertyValuesHolder.ofFloat("translationY", 0f,100f)).apply {
                interpolator = AccelerateInterpolator()
                duration = span
            }
    )
    addPauseListener{
        onPause = { Toast.makeText(context,"pause",Toast.LENGTH_SHORT).show() }
    }
    addListener {
        onEnd = { showA() }
    }
```

```
    start()
}
```

（省略了扩展函数 addPauseListener()的定义，它和 addListener()是类似的。）

得益于 带接收者的lambda，可以轻松地为 AnimatorListenerImpl的成员 onEn赋值，这段逻辑会在画结束时被调用。

这段调用拥有自己独特的结构，它解决了“必须实现全部 java 接口”这个特定的问题，所以它可以得上是一个自定义 DSL 。（当然和 SQL 相比，它显得太简单了）。

下一篇会进一步使用 DSL 的思想将 Android 整套构建动画的接口重构成结构化的代码。到时候就可使用这样的代码来构建动画：

```
animSet {
    objectAnim {
        target = textView
        scaleX = floatArrayOf(1.0f,1.3f)
        scaleY = scaleX
        duration = 300L
        interpolator = LinearInterpolator()
    } with objectAnim {
        target = button
        translationX = floatArrayOf(0f,100f)
        duration = 300
        interpolator = LinearInterpolator()
    } before anim{
        values = intArrayOf(ivRight,screenWidth)
        action = { value -> imageView.right = value as Int }
        duration = 400
        interpolator = LinearInterpolator()
    }
    onEnd = Toast.makeText(activity,"animation end",Toast.LENGTH_SHORT).show()
    start()
}
```