

像 Compose 那样写代码：Kotlin DSL 原理与实战

原创

fundroid

已于 2022-02-28 20:19:48 修改

2623

收藏 13

版权

分类专栏：

Kotlin

 文章标签：

kotlin

DSL

DslMarker

Compose



Kotlin 专栏收录该内容

24 订阅

102 篇文章

订阅专栏

1. 前言

Kotlin 是一门对 DSL 友好的语言，它的许多语法特性有助于 DSL 的打造，提升特定场景下代码的可读性和安全性。本文将带你了解 Kotlin DSL 的一般实现步骤，以及如何通过 **@DslMarker**，**Context Receivers** 等特性提升 **DSL** 的易用性。

2. 什么是 DSL?

DSL 全称是 **Domain Specific Language**，即领域特定语言。顾名思义 DSL 是用来专门解决某一特定问题的语言，比如我们常见的 SQL 或者正则表达式等，DSL 没有通用编程语言（Java、Kotlin等）那么万能，但是在特定问题的解决上更高效。

创作一套全新新语言的成本很高，所以很多时候我们可以基于已有的通用编程语言打造自己的 DSL，比如日常开发中我们将常见到 **gradle** 脚本，其本质就是来自 Groovy 的一套 DSL：

```
1 android {
2     compileSdkVersion 28
3     defaultConfig {
4         applicationId "com.my.app"
5         minSdkVersion 24
6         targetSdkVersion 30
7         versionCode 1
8         versionName "1.0"
9         testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
10    }
11    buildTypes {
12        release {
13            minifyEnabled false
14            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
15        }
16    }
17 }
```

build.gradle 中我们可以用大括号表现层级结构，使用键值对的形式设置参数，没有多余的程序符号，非常直观。如果将其还原成标准的 Groovy 语法则变成下面这样，是下面这样，在可读性上的好坏立判：

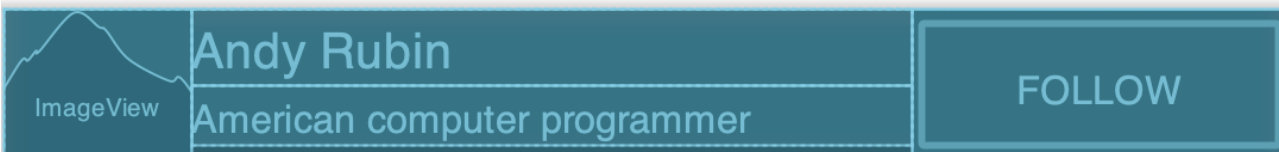
```
1 Android(30,
2     DefaultConfig("com.my.app",
3         24,
4         30,
5         1,
6         "1.0",
7         "android.support.test.runner.AndroidJUnitRunner"
8     )
9 ),
10 BuildTypes(
11     Release(false,
12         getDefaultProguardFile('proguard-android-optimize.txt'),
13         'proguard-rules.pro'
14     )
15 )
```

除了 Groovy，Kotlin 也非常适合 DSL 的书写，正因如此 Gradle 开始推荐使用 **kts** 替代 **gradle**，其实就是利用了 Kotlin 优秀的 DSL 特性。

3. Kotlin DSL 及其优势

Kotlin 是 Android 的主要编程语言，因此我们可以在 Android 开发中发挥其 DSL 优势，提升特定场景下的开发效率。例如 Compose 的 UI 代码就是一个很好的示范，它借助 DSL 让 Kotlin 代码具有了不输于 XML 的表现力，同时还兼顾了类型安全，提升了 UI 开发效率。

普通的 Android View 也可以使用 DSL 进行描述。下面是一个简单的 UI 布局，左边是其对应的 XML 代码，右边是我们为其设计的 Kotlin DSL 代码

	
XML	DSL
<pre><?xml version="1.0" encoding="utf-8"?> <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android" xmlns:tools="http://schemas.android.com/tools" android:layout_width="match_parent" android:layout_height="wrap_content" android:orientation="horizontal"> <ImageView android:layout_width="60dp" android:layout_height="match_parent" android:src="@drawable/avatar" /> <LinearLayout android:layout_width="0dp" android:layout_height="match_parent" android:layout_gravity="center" android:layout_weight="1" android:orientation="vertical"> <TextView android:layout_width="match_parent" android:layout_height="wrap_content" android:textSize="18dp" tool:text="Andy Rubin" /> <TextView android:layout_width="match_parent" android:layout_height="wrap_content" android:textSize="14dp" tool:text="American computer programmer" /> </LinearLayout> <Button android:layout_width="120dp" android:layout_height="match_parent" android:onClick="onClick" tool:text="Follow" /> </LinearLayout></pre>	<pre>val linearLayoutParams = LinearLayout.LayoutParams(context, null).apply { width = MATCH_PARENT height = WRAP_CONTENT } HorizontalLayout { ImageView { image = drawable(R.drawable.avatar) layoutParams = LinearLayout.LayoutParams(context, null).apply { width = dip(60) height = MATCH_PARENT } } VerticalLayout { Text { text = "Andy Rubin" textSize = dip(18) layoutParams = linearLayoutParams } Text { text = "American computer programmer" textSize = dip(14) layoutParams = linearLayoutParams } layoutParams = LinearLayout.LayoutParams(context, null).apply { width = dip(0) height = MATCH_PARENT weight = 1f gravity = Gravity.CENTER } } Button { text = "Follow" onClick { //... } layoutParams = LinearLayout.LayoutParams(context, null).apply { width = dip(120) height = MATCH_PARENT } } layoutParams = linearLayoutParams }</pre>

通过对比可以看到 Kotlin DSL 有诸多好处：

- 有着近似 XML 的结构化表现力
- 较少的字符串，更多的强类型，更安全
- layoutParams 这样的对象可以多次复用
- 可以在定义布局的同时实现 onClick 等
- 如果需要，还可以嵌入 if，for 这样的控制语句

倘若没有 DSL，我们想借助 Kotlin 达到上述好处，代码可能是下面这样的：

```
1 | LinearLayout(context).apply {
2 |     addView(ImageView(context).apply {
3 |         image = context.getDrawable(R.drawable.avatar)
4 |     })
5 | }
```

```
5 | }, LinearLayout.LayoutParams(context, null).apply {...})
6 |
7 |     addView(LinearLayout(context).apply {
8 |         ...
9 |     }, LinearLayout.LayoutParams(context, null).apply {...})
10 |
11 |     addView(Button(context).apply {
12 |         setOnClickListener {
13 |             ...
14 |         }
15 |     }, LinearLayout.LayoutParams(0,0).apply {...})
   | }

```

虽然代码已经借助 `apply` 等作用域函数进行了优化，但写起来仍然很繁琐，这样的代码是完全无法替代 XML 的。

接下来，本文带大家看看上述 DSL 是如何实现的，以及更进一步的优化技巧

4. Kotlin 如何实现 DSL

4.1 高阶函数实现大括号调用

常见的 DSL 都会用大括号来表现层级。Kotlin 的高阶函数允许指定一个 lambda 类型的参数，且当 lambda 位于参数列表的最后位置时可以脱离圆括号，满足 DSL 中的大括号语法要求。

我们知道了实现大括号语法的核心就是将对象创建及初始化逻辑封装成带有尾 lambda 的高阶函数中，我们按照这个思路改造下面代码

```
1 | LinearLayout(context).apply {
2 |     orientation = LinearLayout.HORIZONTAL
3 |     addView(ImageView(context))
4 | }
```

我们为 `LinearLayout` 的创建定义一个高阶函数，根据预设的 `orientation` 命名为 `HorizontalLayout` 以提高可读性。另外我们模仿 Compose 的风格使用首字母大写，让 DSL 节点更具辨识度

```
1 | fun HorizontalLayout(context: Context, init: (LinearLayout) -> Unit) : LinearLayout {
2 |     return LinearLayout(context).apply {
3 |         orientation = LinearLayout.HORIZONTAL
4 |         init(this)
5 |     }
6 | }
```

参数 `init` 是一个尾 lambda，传入刚创建的 `LinearLayout` 对象，便于我们在大括号中为其进行初始化。我们为 `ImageView` 也定义类似的高阶函数后，调用效果如下：

```
1 | HorizontalLayout(context) {
2 |     ...
3 |     it.addView(ImageView(context)) {
4 |         ...
5 |     }
6 | }
```

虽然避免了 `apply` 的出现，但是效果仍然差强人意。

4.2 通过 Receiver 传递上下文

前面经高阶函数转化后的 DSL 中大括号内必须借助 `it` 进行初始化，而且 `addView` 的出现也难言优雅。

首先，我们可以将 lambda 的参数改为 Receiver，大括号中对 `it` 的引用可以变为 `this` 并直接省略：

```
1 | fun HorizontalLayout(context: Context, init: LinearLayout.() -> Unit) : LinearLayout {
2 |     return LinearLayout(context).apply {
3 |         orientation = LinearLayout.HORIZONTAL
4 |         init()
5 |     }
6 | }
```

其次，我们如果能将 `addView` 隐藏到 `ImageView` 内部代码会更加简洁，这需要 `ImageView` 持有它的父 `View` 的引用，我们可以将参数 `context` 换成 `ViewGroup`

```
1 fun ImageView(parent: ViewGroup, init: ImageView.() -> Unit) {
2     parent.addView(ImageView(parent.context).apply(init))
3 }
```

由于不再需要返回实例给父 `View`，返回值也可以改为 `Unit` 了。

按照前面参数转 Receiver 的思路，我们可以进一步上 `ImageView` 的 `parent` 参数提到 Receiver 的位置，实际就是改成 `ViewGroup` 的扩展函数：

```
1 fun ViewGroup.ImageView(init: ImageView.() -> Unit) {
2     addView(ImageView(context).apply(init))
3 }
```

经过上面优化，DSL 中写 `ImageView` 时无需再传递参数 `context`，而且大括号中也不会出现 `it`

```
1 HorizontalLayout {
2     ...
3     ImageView {
4         ...
5     }
6 }
```

4.3 扩展函数优化代码风格

`View` 的固有方法签名都是为命令式语句设计的，不符合 DSL 的代码风格，此时可以借助 Kotlin 的扩展函数进行重新定义。

那么什么是 DSL 应该有的代码风格？虽然不同功能的 DSL 不能一概而论，但是它们大都是偏向于对结构的静态描述，所以应该避免出现命令式的命名风格。

```
1 fun View.onClick(l: (v: View) -> Unit) {
2     setOnClickListener(l)
3 }
```

比如上面这样，通过扩展函数使用 `onClick` 优化 `setOnClickListener` 命名，而且参数中使用函数类型替代了原有的 `OnClickListener` 接口类型，在 DSL 写起来更简单。由于 `OnClickListener` 是一个 SAM 接口，所以优势不够明显。下面的例子可能更能说明问题。

如果想在 DSL 中调用 `TextView` 的 `addTextChangedListener` 方法，写法上将非常冗余：

```
1 TextView {
2     addTextChangedListener( object : TextWatcher {
3         override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {
4             ...
5         }
6
7         override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
8             ...
9         }
10
11         override fun afterTextChanged(s: Editable?) {
12             ...
13         }
14     })
```

为 `TextView` 新增适合 DSL 的扩展函数：

```
1 fun TextView.textChangeListener(init: _TextWatcher.() -> Unit) {
2     val listener = _TextWatcher()
3     listener.init()
4     addTextChangedListener(listener)
5 }
6
7 class _TextWatcher : android.text.TextWatcher {
8
9 }
```

```

10 private var _onTextChanged: ((CharSequence?, Int, Int, Int) -> Unit)? = null
11 override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
12     _onTextChanged?.invoke(s, start, before, count)
13 }
14 fun onTextChanged(listener: (CharSequence?, Int, Int, Int) -> Unit) {
15     _onTextChanged = listener
16 }
17
18 // beforeTextChanged 和 afterTextChanged 的相关代码省略
19 }

```

DSL 中使用的效果如下，清爽了不少

```

1 Text {
2     textChangeListener {
3         beforeTextChanged { charSequence, i, i2, i3 ->
4             //...
5         }
6
7         onTextChanged { charSequence, i, i2, i3 ->
8             //...
9         }
10
11         afterTextChanged {
12             //...
13         }
14     }
15 }

```

5. 进一步优化你的 DSL

经过前面的优化我们的 DSL 基本达到了预期效果，接下来通过更多 Kotlin 的特性让这套 DSL 更加好用。

5.1 infix 增强可读性

Kotlin 的中缀函数可以让函数省略圆点以及圆括号等程序符号，让语句更自然，进一步提升可读性。

比如所有的 View 都有 `setTag` 方法，正常使用如下：

```

1 HorizontalLayout {
2     setTag(1,"a")
3     setTag(2,"b")
4 }

```

我们使用中缀函数来优化 `setTag` 的调用如下：

```

1 class _Tag(val view: View) {
2     infix fun <B> Int.to(that: B) = view.setTag(this, that)
3 }
4
5 fun View.tag(block: _Tag.() -> Unit) {
6     _Tag(this).apply(block)
7 }

```

DSL 中调用的效果如下：

```

1 HorizontalLayout {
2     tag {
3         1 to "a"
4         2 to "b"
5     }
6 }

```

5.2 @DslMarker 限制作用域

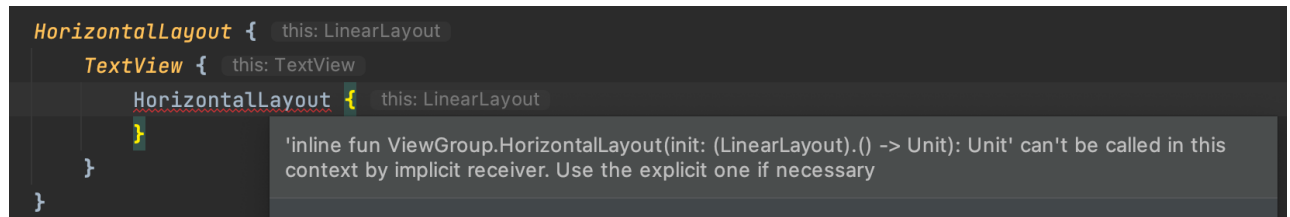
```

1 | HorizontalLayout { // this: LinearLayout
2 |     ...
3 |     TextView { // this: TextView
4 |         // 此处仍然可以调用 HorizontalLayout
5 |         HorizontalLayout {
6 |             ...
7 |         }
8 |     }
9 | }
10| }

```

上述 DSL 代码，我们发现在 `TextView {...}` 可以调用 `HorizontalLayout {...}`，这显示是不合逻辑的。由于 `Text` 的作用域同时处于父 `HorizontalLayout` 的作用域中，所以上面代码中，编译器会认为其内部的 `HorizontalLayout {...}` 是调用在 `this@LinearLayout` 中不会报错。缺少了编译器的提醒，会增大出现 Bug 的几率

Kotlin 为 DSL 的使用场景提供了 `@DslMarker` 注解，可以对方法的作用域进行限制。添加注解的 lambda 中在省略 `this` 的隐式调用时只能访问到最近的 Receiver 类型，当调用更外层的 Receiver 的方法会报错如下：



`@DslMarker` 是一个元注解，我们需要基于它定义自己的注解

```

1 | @DslMarker
2 | @Target(AnnotationTarget.TYPE)
3 | annotation class ViewDslMarker

```

接着，在尾 lambda 的 Receiver 添加此注解，如下：

```

1 | fun ViewGroup.TextView(init: (@ViewDslMarker TextView.()) -> Unit) {
2 |     addView(TextView(context).apply { init })
3 | }

```

`TextView {...}` 中如果不写 `this`，则只能调用 `TextView` 的方法，如果想调用外层 Receiver 的方法，必须显示的使用 `this@xxx` 调用

5.3 Context Receivers 传递多个上下文

Context Receivers 是刚刚在 Kotlin 1.6.20-M1 中发布的新语法，它使函数定义时拥有多个 Receiver 成为可能。

```

1 | context(View)
2 | val Float.dp
3 |     get() = this * this@View.resources.displayMetrics.density
4 |
5 | class SomeView : View {
6 |     val someDimension = 4f.dp
7 | }

```

上面代码是使用 Context Receivers 定义函数的例子，`dp` 是 `Float` 的扩展函数，所以已经有了一个 Receiver，在此基础上，通过 `context(View)` 又增加了 `View` 作为 Receiver，可以通过 `this@xxx` 引用不同 Receiver 完成运算。

`context` 的新特性乍看起来好像没啥用，但其实它对于 DSL 场景有很重要的意义，可以让我们的代码变得更智能。比如下面的例子

```

1 | fun View.dp(value: Int): Int = (value * context.resources.displayMetrics.density).toInt()
2 |
3 | HorizontalLayout {
4 |     TextView {
5 |         layoutParams = LinearLayout.LayoutParams(context, null).apply {
6 |             width = dp(60)
7 |             height = 0
8 |             weight = 1.0
9 |         }
10| }

```

```

10 |     }
11 | }
12 |
13 | RelativeLayout {
14 |     TextView {
15 |         layoutParams = RelativeLayout.LayoutParams(context, null).apply {
16 |             width = dp(60)
17 |             height = ViewGroup.LayoutParams.WRAP_CONTENT
18 |         }
19 |     }
20 | }

```

上面的代码中有几点可以使用 `context` 帮助改善。

首先，代码中使用带参数的 `dp(60)` 进行 `dip` 转换。我们可以通过前面介绍的 `context` 语法替换为 `60f.dp` 这样的写法，避免括号的出现，写起来更加舒适。

此外，我们知道 `View` 的 `LayoutParams` 的类型由其父 `View` 类型决定，上面代码中，我们在创建 `LayoutParams` 时必须时刻留意类型是否正确，心理负担很大。

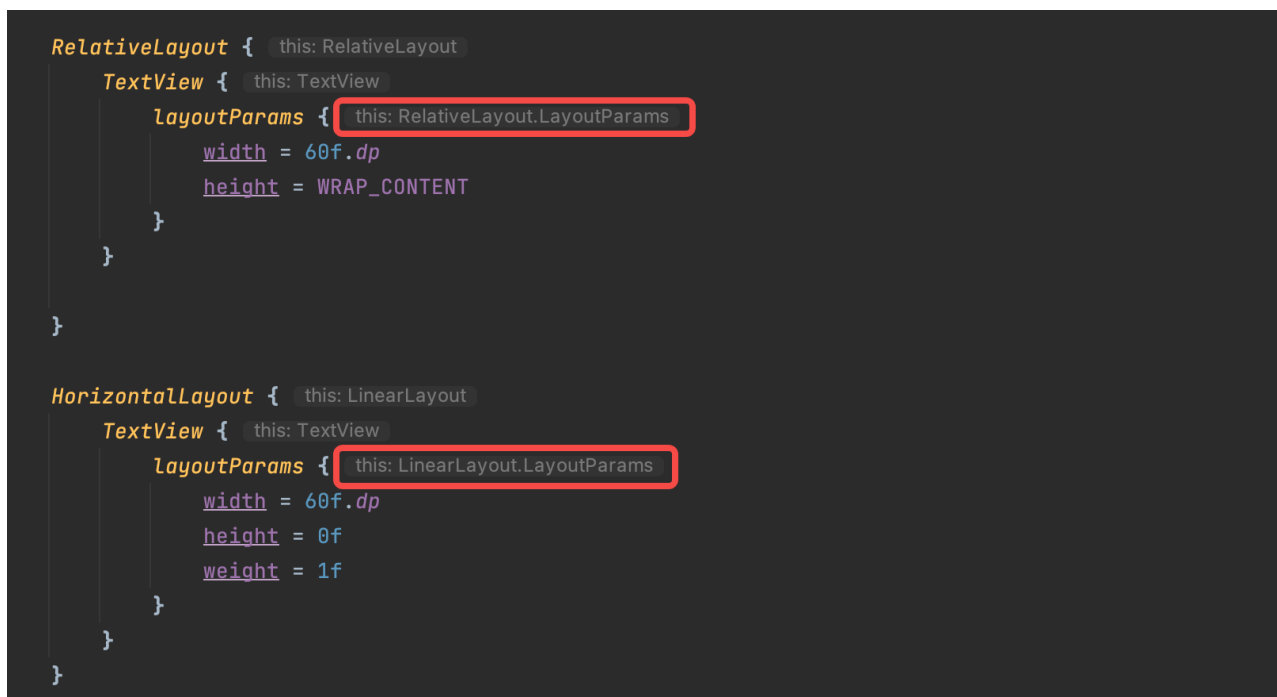
这个问题也可以用 `context` 很好的解决，如下我们为 `TextView` 针对不同的 `context` 定义 `layoutParams` 扩展函数：

```

1 | context(RelativeLayout)
2 | fun TextView.layoutParams(block: RelativeLayout.LayoutParams.() -> Unit) {
3 |     layoutParams = RelativeLayout.LayoutParams(context, null).apply(block)
4 | }
5 |
6 | context(LinearLayout)
7 | fun TextView.layoutParams(block: LinearLayout.LayoutParams.() -> Unit) {
8 |     layoutParams = LinearLayout.LayoutParams(context, null).apply(block)
9 | }

```

在 DSL 中使用效果如下：



```

RelativeLayout { this: RelativeLayout
    TextView { this: TextView
        layoutParams { this: RelativeLayout.LayoutParams
            width = 60f.dp
            height = WRAP_CONTENT
        }
    }
}

LinearLayout { this: LinearLayout
    TextView { this: TextView
        layoutParams { this: LinearLayout.LayoutParams
            width = 60f.dp
            height = 0f
            weight = 1f
        }
    }
}

```

`TextView` 的 `layoutParams {...}` 会根据父容器类型自动返回不同的 `this` 类型，便于后续配置。

5.4 使用 `inline` 和 `@PublishedApi` 提高性能

DSL 的实现使用了大量高阶函数，过多的 `lambda` 会产生过多的匿名类，同时也会增加运行时对象创建的开销，不少 DSL 选择使用 `inline` 操作符，减少匿名类的产生，提高运行时性能。

比如为 `ImageView` 的定义添加 `inline`：

```

1 | inline fun ViewGroup.ImageView(init: ImageView.() -> Unit) {
2 |     addView(ImageView(context).apply(init))
3 | }

```

`inline` 函数内部调用的函数必须是 `public` 的，这会造成一些不必要的代码暴露，此时可以借助 `@PublishedApi` 化解。

```

1 //resInt 指定图片
2 inline fun ViewGroup.ImageView(resId: Int, init: ImageView.() -> Unit) {
3     _ImageView(init).apply { setImageResource(resId) }
4 }
5
6 //drawable 指定图片
7 inline fun ViewGroup.ImageView(drawable: Drawable, init: ImageView.() -> Unit) {
8     _ImageView(init).apply { setImageDrawable(drawable) }
9 }
10
11 @PublishedApi
12 internal inline fun ViewGroup._ImageView(init: ImageView.() -> Unit) =
13     ImageView(context).apply {
14         this._ImageView.addView(this)
15         init()
16     }

```

如上，为了方便 DSL 中使用，我们定义了两个 `ImageView` 方法，分别用于 `resId` 和 `drawable` 的图片设置。由于大部分代码可以复用，我们抽出了一个 `_ImageView` 方法。但是由于要在 `inline` 方法中使用，所以编译器要求 `_ImageView` 必须是 `public` 类型。`_ImageView` 只需在库的内部服务，所以可以添加为 `internal` 的同时加 `@PublishedApi` 注解，它允许一个模块内部方法在 `inline` 中使用，且编译器不会报错。

6. 总结

经过上述几个步骤，我们的 DSL 终于成型了，而且还经过了优化，看看最终的样子：

```

1 val linearLayoutParams = LinearLayout.LayoutParams(context, null).apply {
2     width = MATCH_PARENT
3     height = WRAP_CONTENT
4 }
5
6 HorizontalLayout {
7     ImageView(R.drawable.avatar) {
8         layoutParams {
9             width = 60f.dp
10            height = MATCH_PARENT
11        }
12    }
13
14    VerticalLayout {
15
16        Text("Andy Rubin") {
17            textSize = 18.dp
18            layoutParams = linearLayoutParams
19        }
20        Text("American computer programmer") {
21            textSize = 14f.dp
22            layoutParams = linearLayoutParams
23        }
24
25        layoutParams {
26            width = dip(0)
27            height = MATCH_PARENT
28            weight = 1f
29            gravity = Gravity.CENTER
30        }
31    }
32
33    Button("Follow") {
34        onClick {
35            //...
36        }
37        layoutParams {
38            width = 120f.dp
39            height = MATCH_PARENT
40        }

```



```
41         }  
42     }  
43     layoutParams = linearLayoutParams  
44 }
```

当然 Android 中 DSL 远不止 UI 这一种使用场景，但是实现思路都是相近的，最后再来一起回顾一下基本步骤：

1. 使用带有尾 lambda 的高阶函数实现大括号的层级调用
2. 为 lambda 添加 Receiver，通过 this 传递上下文
3. 通过扩展函数优化代码风格，DSL 中避免出现命令式的语义
4. 使用 infix 减少点号圆括号等符号的出现，提高可读性
5. 使用 @DslMarker 限制 DSL 作用域，避免出错
6. 使用 Context Receivers 传递多个上下文，DSL 更聪明（非正式语法，未来有变动的可能）
7. 使用 inline 提升性能，同时使用 @PublishedApi 避免不必要的代码暴露



AndroidPub
咱们只聊技术不扯淡

微信公众号 >