链滴

Kotlin 实战 | 语法糖，总有一颗甜到你

作者：lzlyy

原文链接：https://ld246.com/article/1600053307874

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

这是该系列的第九篇，系列文章目录如下：

1. Kotlin基础 | 白话文转文言文般的Kotlin常识
2. Kotlin基础 | 望文生义的Kotlin集合操作
3. Kotlin实战 | 用实战代码更深入地理解预定义扩展函数
4. Kotlin实战 | 使用DSL构建结构化API去掉冗余的接口方法
5. Kotlin基础 | 抽象属性的应用场景
6. Kotlin进阶 | 动画代码太丑，用DSL动画库拯救
7. Kotlin基础 | 用约定简化相亲
8. Kotlin基础 | 2 = 12？泛型、类委托、重载运算符综合应用
9. Kotlin实战 | 语法糖，总有一颗甜到你(持续更新)

学习了 Kotlin 后，写代码时常有一种“闹革命”的冲动，总是希望运用语法糖推翻“旧世界”（这不好，项目会 delay 的~）。本文归纳了 Kotlin 语法糖在项目实战中的综合运用，以实际问题为索引在分析解决方案的同时介绍相关语法知识。

脱离父控件

```
fun View?.detachParent() = this?.parent?.let { it as? ViewGroup }?.also { it.removeView(this) }
```

当控件还隶属于一个容器时，就无法将它再次 addView()到一个新的容器中。所以为 View 扩展一个法，主动将它从父控件中移出。

判断 View 是否在屏幕上

```
val View.inScreen: Boolean
    get() {
        // 获取屏幕宽度
        val screenWidth = context?.resources?.displayMetrics?.widthPixels ?: 0
        // 获取屏幕高度
        val screenHeight = context?.resources?.displayMetrics?.heightPixels ?: 0
        // 构建屏幕矩形
        val screenRect = Rect(0, 0, screenWidth, screenHeight)
        val array = IntArray(2)
        // 获取视图矩形
        getLocationOnScreen(array)
        val viewRect = Rect(array[0], array[1], array[0] + width, array[1] + height)
        // 判断屏幕和视图矩形是否有交集
        return screenRect.intersect(viewRect)
    }
```

为 View增加一个扩展方法，返回布尔值，在其中获取视图和屏幕矩形区域，然后判断是否有交集，有则表示视图出现在屏幕上。

协程和视图生命周期相绑定

若协程用于异步加载一张图片，这张图片显示在 ImageView 上。当 ImageView 所在界面已被销毁，得及时取消协程的加载任务，以释放资源：

```
fun Job.autoDispose(view: View) {
    val isAttached = Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT && view.isAttachedToWindow || view.windowToken != null
    if (!isAttached){
        cancel()
    }

    // 监听视图生命周期
    val listener = object : View.OnAttachStateChangeListener {
        // 在视图生命周期结束时取消协程
        override fun onViewDetachedFromWindow(v: View?) {
            cancel()
            v?.removeOnAttachStateChangeListener(this)
        }

        override fun onViewAttachedToWindow(v: View?) = Unit
    }

    view.addOnAttachStateChangeListener(listener)
    invokeOnCompletion {
        view.removeOnAttachStateChangeListener(listener)
    }
}
```

为 Job扩展方式，传入 View 类型的参数，表示该 Job和该 View 的生命周期绑定，当 View 生命周结束时，自动取消协程。

获取 DecorView

```
val Activity.decorView: FrameLayout?
    get() = (takeIf { !isFinishing && !isDestroyed }?.window?.decorView) as? FrameLayout
```

为 Activity扩展属性 decorView，它的类型是 FrameLayout。在 get()方法中定义如何获取该属性。

当 Activity生命周期没有结束的时候，获取它的 Window，再从从 Window中获取 DecorView，强为 FrameLayout。

用同样的思路可以获取 Activity的 Content View:

```
val Activity.contentView: FrameLayout?
    get() = takeIf { !isFinishing && !isDestroyed }?.window?.decorView?.findViewById<FrameLayout>(android.R.id.content)
```

将 px 值转换成 dp 值

在非 xml 环境下构建布局，需要将 px 转换为 dp 来进行多屏幕适配。Java 的做法是在 Util类中新增个静态函数。利用 Kotlin 的扩展属性可以更简洁地实现：

```
val Int.dp: Int
    get() {
        return TypedValue.applyDimension(
            TypedValue.COMPLEX_UNIT_DIP,
            this.toFloat(),
```

```
        Resources.getSystem().displayMetrics
    ).toInt()
}
```

为 Int 扩展一个属性 dp，它的类型是 Int。在 get()中定义该属性的取值算法。

然后就可以像这样动态地将 Int 值 dp 化：

```
viewGroup.addView( textView, LayoutParam( 40.dp, 50.dp ) )
```

弃用Builder模式

当构造复杂对象时，需要很多参数，如果将所有参数都通过一个构造函数来传递，缺乏灵活性，但如重载若干个带有不同参数的构造函数，代码就变得臃肿。Builder 模式可以简化构建过程。

在 Java 中 Builder模式 代码如下：

```
public class Person {
    //'必选参数'
    private String name;
    //'以下都是可选参数'
    private int gender;
    private int age;
    private int height;
    private int weight;

    //'私有构造函数，限制必须通过构造者构建对象'
    private Person(Builder builder) {
        this.name = builder.name;
        this.gender = builder.gender;
        this.age = builder.age;
        this.height = builder.height;
        this.weight = builder.weight;
    }

    //'构造者'
    public static class Builder {
        private String name;
        private int gender;
        private int age;
        private int height;
        private int weight;

        //'必选参数必须在构造函数中传入'
        public Builder(String name) {
            this.name = name;
        }

        //'以下是每个非必要属性的设值函数，它返回构造者本身用于链式调用'
        public Builder age(int age) {
            this.age = age;
            return this;
        }
```

```
        public Builder gender(int gender) {
            this.gender = gender;
            return this;
        }

        public Builder height(int height) {
            this.height = height;
            return this;
        }

        public Builder weight(int weight) {
            this.weight = weight;
            return this;
        }

        //'构建对象'
        public Person build() {
            return new Person(this);
        }
    }
```

然后就可以像这样构建 Person实例：

```
//'使用 Builder模式'
Person p = new Person.Builder("taylor")
            .age(50)
            .gender(1)
            .weight(43)
            .build();

//'使用构造函数'
Person p2 = new Person("taylor", 50, 1, 0, 43);
```

对比之下，Builder模式 有两个优势：

1. 为参数标注语义：在Builder模式中，每个属性的赋值都是一个函数，函数名标注了属性语义。而接使用构造函数时，很难分辨50，43哪个是年龄，哪个是体重。

2. 可选参数：Builder模式中，除了必选参数，其他参数是可选的。但直接使用构造函数必须为所有数赋值，比如上例中第四个参数身高被赋值为0。

但 Builder模式 也有代价，新增了一个中间类 Builder。

使用 Kotlin 的 命名参数+参数默认值+数据类语法，在没有任何副作用的情况下就能实现 Builder模：

```
//'将Person定义为数据类'
data class Person(
    var name: String,
    //'为以下可选参数设置默认值'
    var gender: Int = 1,
    var age: Int= 0,
    var height: Int = 0,
    var weight: Int = 0
)
```

```
//'使用命名参数构建Person实例'
val p  = Person(name =  “taylor” ,gender = 1,weight = 43)
```

关于 数据类、参数默认值、命名参数更详细的介绍可以点击这里

如果想增加参数约束条件可以调用 require()方法：

```
data class Person(
    var name: String,
    var gender: Int = 1,
    var age: Int= 0,
    var height: Int = 0,
    var weight: Int = 0
){
    //'在构造函数被调用的时候执行参数合法检查'
    init {
        require(name.isNotEmpty()){” name cant be empty “}
    }
}
```

此时如果像下面这样构造 Person，则会抛出异常：

```
val p = Person(name="",gender = 1)
java.lang.IllegalArgumentException: name cant be empty
```

打印列表、map

调试程序时，经常需要打印列表内容，通常会这样打印：

```
for (String str:list) {
    Log.v("test", "str="+str);
}
```

不同业务界面的数据类型不同，为了调试，这样的 for 循环就会散落在各处，而且列表内容会分若干条 log 输出，中间极有可能被别的log打断。

有没有一个函数可以打印包含任意数据类型的列表，并将列表内容组织成更具可读性的字符串？

用 Kotlin 的 扩展函数+泛型+高阶函数就能优雅地做到：

```
fun <T> Collection<T>.print(map: (T) -> String) =
    StringBuilder("\n[").also { sb ->
        //'遍历集合元素，通过 map 表达式将元素转换成感兴趣的字串，并独占一行'
        this.forEach { e -> sb.append("\n\t${map(e)},") }
        sb.append("\n]")
    }.toString()
```

为集合的基类 Collection新增一个扩展函数，它是一个高阶函数，因为它的参数是另一个函数，该函用 lambda 表示。再把集合元素抽象成泛型。通过 StringBuilder将所有集合内容拼接成一个自动换的字符串。

写段测试代码看下效果：

```
data class Person(var name: String, var age: Int)

val persons = listOf(
    Person("Peter", 16),
    Person("Anna", 28),
    Person("Anna", 23),
    Person("Sonya", 39)
)

persons.print { "${it.name}_${it.age}" }.let { Log.v("test",it) }
```

打印结果如下：

```
V/test: [
        Peter_16,
        Anna_28,
        Anna_23,
        Sonya_39,
    ]
```

同样地，可以如法炮制一个打印 map 的扩展函数：

```
fun <K, V> Map<K, V?>.print(map: (V?) -> String): String =
    StringBuilder("\n{").also { sb ->
        this.iterator().forEach { entry ->
            sb.append("\n\t[${entry.key}] = ${map(entry.value)}")
        }
        sb.append("\n}")
    }.toString()
```

将 data 类转换成 map

有些数据类字段比较多，调试时，想把它们通通打印出来，在 Java 中，借助于 AndroidStudio 的 toring功能倒是可以方便地生成可读性很高的字串：

```
public class Person {
    private String name;
    private int age;

    @Override
    public String toString() {
        return " Person{ " +
                " name= ' " + name + ' \" +
                " , age= " + age +
                '}' ;
    }
}
```

但是每新建一个数据类都要手动生成一个 toString()方法也挺麻烦。

利用 Kotlin 的 data class可以省去这一步，但打印效果是所有字段都在同一行中：

```
data class Person(var name: String, var age: Int)
```

```
Log.v(“test”, “person=${Person("Peter", 16)}”)

//输出如下:
V/test: person=Person(name=Peter, age=16)
```

如果字段很多，把它们都打印在一行中可读性很差。

有没有一种方法，可以读取一个类中所有的字段信息？ 这样我们就可以将他们组织成想要的形状。请下面这个方法：

```
fun Any.ofMap() =
    //'过滤掉除data class以外的其他类'
    this::class.takeIf { it.isData }
        // 遍历类的所有成员 过滤掉成员方法 只考虑成员属性
        ?.members?.filterIsInstance<KProperty<Any>>()
        //'将成员属性名和值存储在Pair中'
        ?.map { it.name to it.call(this) }
        //'将Pair转换成map'
        ?.toMap()
```

为任意 Kotlin 中的类添加一个 扩展函数，它的功能是将 data class中所有的字段名及其对应值存在个 map 中。其中用到的 Kotlin 语法糖如下：

- isData是KClass中的一个属性，用于判断该类是不是一个data class。KClass是 Kotlin 中用来描述**的类型**，KClass可以通过对象::class语法获得。
- members也是KClass中的一个属性，它包含了所有类的方法和属性。
- filterIsInstance()是Iterable接口的扩展函数，用于过滤出集合中指定的类型。
- to是一个infix扩展函数，它的定义如下：

```
public infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

- 带有 infix标识的函数只允许带有一个参数，并且在调用时可以省略包裹参数的括号。这种语法叫缀表达式

写段测试代码，结合上一节的打印 map 函数看下效果：

```
data class Person(var name: String, var age: Int)

Person("Peter", 16).ofMap()?.print { it.toString() }.let { Log.v("test","$it") }
```

测试代码先将 Person实例转换成 map，然后打印 map。输出结果如下：

```
V/test:
    {
        [age] = 16
        [name] = Peter
    }
```

若 data class嵌套会发生什么？

```
//'位置，嵌套在Person类中'
data class Location(var x: Int, var y: Int)
data class Person(var name: String, var age: Int, var locaton: Location? = null)
```

```
Person("Peter", 16, Location(20, 30)).ofMap()?.print { it.toString() }.let { Log.v("test", "$it") }

//'打印结果如下'
   {
      [age] = 16
      [locaton] = Location(x=20, y=30)
      [name] = Peter
   }
```

期望得到类似 Json 的打印效果，但输出结果还差一点。是因为将 Person转化成 Map时并没有将嵌的 Location也转化成键值对。

需要将 ofMap()方法重构成递归调用：

```
fun Any.ofMap(): Map<String, Any?>? {
   return this::class.takeIf { it.isData }
      ?.members?.filterIsInstance<KProperty<Any>>()
      ?.map { member ->
         val value = member.call(this)?.let { v->
            //'若成员变量是data class，则递归调用ofMap()，将其转化成键值对，否则直接返回值'
            if (v::class.isData) v.ofMap()
            else v
         }
         member.name to value
      }
      ?.toMap()
}
```

为了让打印结果也有嵌套缩进效果，打印 Map 的函数也需要相应地重构：

```
/**
 * 打印 Map，生成结构化键值对子串
 * @param space 行缩进量
 */
fun <K, V> Map<K, V?>.print(space: Int = 0): String {
   //'生成当前层次的行缩进，用space个空格表示，当前层次每一行内容都需要带上缩进'
   val indent = StringBuilder().apply {
      repeat(space) { append(" ") }
   }.toString()
   return StringBuilder("\n${indent}{").also { sb ->
      this.iterator().forEach { entry ->
         //'如果值是 Map 类型，则递归调用print()生成其结构化键值对子串，否则返回值本身'
         val value = entry.value.let { v ->
            (v as? Map<*, *>)?.print("${indent}${entry.key} = ".length) ?: v.toString()
         }
         sb.append("\n\t${indent}[${entry.key}] = $value,")
      }
      sb.append("\n${indent}}")
   }.toString()
}
```

写段测试代码，看看效果：

```
//'坐标类，嵌套在Location类中'
```

```
data class Coordinate(var x: Int, var y: Int)
//'位置类，嵌套在Person类中'
data class Location(var country: String, var city: String, var coordinate: Coordinate)
data class Person(var name: String, var age: Int, var locaton: Location? = null)

Person("Peter", 16, Location("china", "shanghai", Coordinate(10, 20))).ofMap()?.print().let { Log
v("test", "$it") }

//'打印如下'
   {
      [age] = 16,
      [locaton] =
          {
             [city] = shanghai,
             [coordinate] =
                    {
                       [x] = 10,
                       [y] = 20,
                    },
             [country] = china,
          },
      [name] = Peter,
   }
```

获取当前周一和周日

Java 系统默认一周的第一天是周日，最后一天是周六。

若需要做类似“一周一次提醒”的功能，就可以本地化提醒时间，然后在每次触发提醒时把它和当前一作比较，大于则说明本周已提醒。

```
fun thisMondayInMillis() = Calendar.getInstance().let { c ->
   //'如果当前是周天，则减一天，计算周六所在周的周一'
   if (c.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY) c.add(Calendar.DATE, -1)
   c.set(Calendar.DAY_OF_WEEK, Calendar.MONDAY)
   c.set(Calendar.HOUR_OF_DAY, 0)
   c.set(Calendar.MINUTE, 0)
   c.set(Calendar.SECOND, 0)
   c.set(Calendar.MILLISECOND, 0)
   c.timeInMillis
}

fun thisSundayInMillis() = Calendar.getInstance().let { c ->
   //'如果不是周天，则将日期调整到当前周的周六，然后在加一天'
   if (c.get(Calendar.DAY_OF_WEEK) != Calendar.SUNDAY) {
      c.set(Calendar.DAY_OF_WEEK, Calendar.SATURDAY)
      c.add(Calendar.DATE, 1)
   }
   c.set(Calendar.HOUR_OF_DAY, 0)
   c.set(Calendar.MINUTE, 0)
   c.set(Calendar.SECOND, 0)
   c.set(Calendar.MILLISECOND, 0)
   c.timeInMillis
}
```

RecyclerView表项点击监听器

RecyclerView没有子控件点击事件监听器，那就用 Kotlin扩展方法扩展一个：

```
//'为 RecyclerView 扩展表项点击监听器'
fun RecyclerView.setOnItemClickListener(listener: (View, Int) -> Unit) {
    //'为 RecyclerView 子控件设置触摸监听器'
    addOnItemTouchListener(object : RecyclerView.OnItemTouchListener {
        //'构造手势探测器，用于解析单击事件'
        val gestureDetector = GestureDetector(context, object : GestureDetector.OnGestureListe
er {
            override fun onShowPress(e: MotionEvent?) {
            }

            override fun onSingleTapUp(e: MotionEvent?): Boolean {
                //'当单击事件发生时，寻找单击坐标下的子控件，并回调监听器'
                e?.let {
                    findChildViewUnder(it.x, it.y)?.let { child ->
                        listener(child, getChildAdapterPosition(child))
                    }
                }
                return false
            }

            override fun onDown(e: MotionEvent?): Boolean {
                return false
            }

            override fun onFling(e1: MotionEvent?, e2: MotionEvent?, velocityX: Float, velocityY: Fl
at): Boolean {
                return false
            }

            override fun onScroll(e1: MotionEvent?, e2: MotionEvent?, distanceX: Float, distanceY:
loat): Boolean {
                return false
            }

            override fun onLongPress(e: MotionEvent?) {
            }
        })

        override fun onTouchEvent(rv: RecyclerView, e: MotionEvent) {

        }

        //'在拦截触摸事件时，解析触摸事件'
        override fun onInterceptTouchEvent(rv: RecyclerView, e: MotionEvent): Boolean {
            gestureDetector.onTouchEvent(e)
            return false
        }

        override fun onRequestDisallowInterceptTouchEvent(disallowIntercept: Boolean) {
        }
```

```
    })
}
```