

## 34 | Kotlin与Jetpack简直是天生一对！

2022-04-11 朱涛

《朱涛·Kotlin编程第一课》

课程介绍 >



讲述：朱涛

时长 11:22 大小 10.41M



你好，我是朱涛。今天，我们来聊聊 Android 的 Jetpack。

在我看来，Kotlin 和 Jetpack，它们两个简直就是天生一对。作为 Android 开发者，如果只用 Kotlin 不用 Jetpack，我们其实很难在 Android 平台充分发挥 Kotlin 的语言优势。而如果我们只用 Jetpack 而不用 Kotlin，那么，我们将只能用到 Jetpack 的小部分功能。毕竟，Jetpack 当中有很多 API 和库，是专门为 Kotlin 提供的。

经过前面课程内容的学习，相信现在你已经对 Kotlin 十分熟悉了，那么，接下来就让我们来看看 Jetpack 吧！这节课里，我会为你介绍 Jetpack 核心库的基本概念、简单用法，以及它跟 Kotlin 之间的关系，从而也为我们下节课的实战项目打下基础。

### Jetpack 简介

**Jetpack**，它有“喷气式背包”的意思。对于我们开发者来说，它其实就是 **Google** 官方为我们提供的一套开发套件，专门用来帮助 **Android** 开发者提升开发效率、提升应用稳定性的。



**Android Jetpack**，最初的宣传图标，就是“穿着喷气式背包的 **Android** 机器人”。大概意思就是：有了 **Jetpack**，**Android** 就能“起飞了”。这当然只是一种夸张的比喻，不过，从我实际的开发体验来说，**Jetpack** 确实可以给 **Android** 开发者带来极大的好处，尤其是当 **Jetpack** 与 **Kotlin** 结合到一起的情况下。

我们先来了解下 **KTX**。

## **KTX**

**KTX** 是 **Jetpack** 当中最特殊的一类库，它是由 **Kotlin** 编写的，同时也仅为 **Kotlin** 开发者服务，使用 **Java** 语言的 **Android** 开发者是用不了的。**KTX**，它的作用其实是对当前 **Android** 生

态当中的 **API** 进行额外补充。它依托 **Kotlin** 的扩展能力，为 **Android** 原有 **API** 增加新的：扩展函数、扩展属性、高阶函数、命名参数、参数默认值、协程支持。

如果我们想要使用 **KTX** 的核心功能，我们需要单独进行依赖：

 复制代码

```
1 // 代码段1
2
3 dependencies {
4     implementation "androidx.core:core-ktx:1.7.0"
5 }
```

让我们来看一个关于 **SharedPreferences** 的简单例子，如果我们使用 **Java**，我们大概率是需要写一堆模板代码的，类似这样：

 复制代码

```
1 // 代码段2
2
3 SharedPreferences sharedPreferences= getSharedPreferences("data",Context.MODE_P
4 SharedPreferences.Editor editor = sharedPreferences.edit();
5 editor.putString(SP_KEY_RESPONSE, response);
6
7 editor.commit();
8 editor.apply();
```

不过，如果我们有了 **KTX**，那么代码就会变得极其简单：

 复制代码

```
1 // 代码段3
2
3 preference.edit { putBoolean("key", value) }
```

上面的这个 **edit()** 方法，其实是一个高阶函数，它是由 **KTX** 提供的，如果你去看它的源代码，会发现，它其实就是一个扩展出来的高阶函数：

 复制代码

```
1 // 代码段4
2
```

```

3 inline fun SharedPreferences.edit(
4     commit: Boolean = false,
5     action: SharedPreferences.Editor.() -> Unit
6 ) {
7     val editor = edit()
8     action(editor)
9     if (commit) {
10         editor.commit()
11     } else {
12         editor.apply()
13     }
14 }

```

可以看到，KTX 其实就是将一些常见的模板代码封装了起来，然后以扩展函数的形式提供给开发者。虽然它自身的原理很简单，但是却可以大大提升开发者的效率。

KTX 除了能够扩展 Android SDK 的 API 以外，它还可以扩展 Jetpack 当中其他的库，比如说 LiveData、Room 等等。接下来，我们就来看看 Jetpack 当中比较核心的库：Lifecycle。

## Lifecycle

Lifecycle，其实就是 Android 的生命周期组件。在整个 Jetpack 组件当中的地位非常特殊，是必学的组件。举个例子，其他的组件比如 WorkManager，如果我们实际工作中用不上，那么我们不去学它是不会有什么问题的。Lifecycle 不一样，只要我们是做 Android 开发的，我们就绕不开 Lifecycle。Activity 里面有 Lifecycle；Fragment 里面也有；LiveData 里面也有；

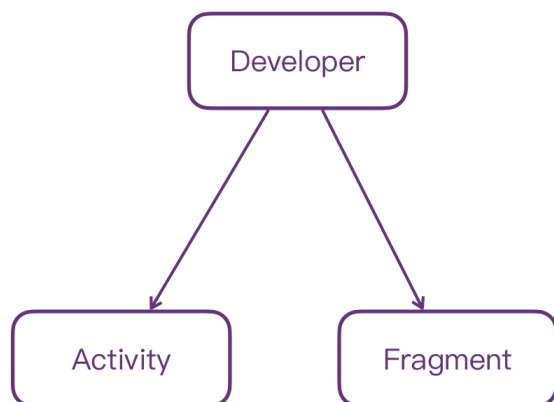
ViewModel 底层也用到了 Lifecycle；使用协程也离不开 Lifecycle。

那么，Lifecycle 到底是什么呢？我们平时提到生命周期，往往都是说的 Activity、Fragment，而它们两者之间却有一个很大的问题，**生命周期函数不一致**。

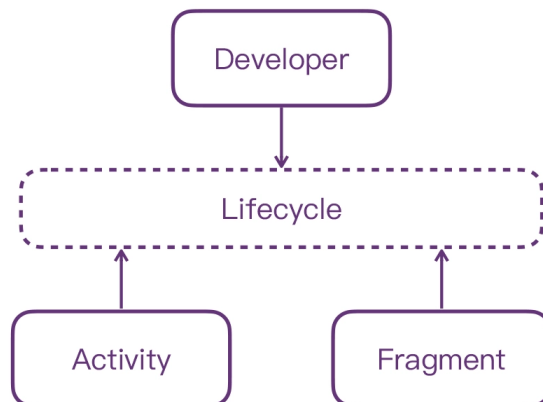
Activity 的生命周期我们肯定心里有数，不过 Fragment 生命周期函数比 Activity 多了几个：onCreateView、onViewCreated、onViewStateRestore、onDestoryView。最重要的是，Fragment 生命周期、回调函数、Fragment 内部 View 的生命周期，它们三者之间还有很复杂的对应关系。换句话说，Fragment 的生命周期函数要比 Activity 复杂一些。

加之，Activity 和 Fragment 结合的情况下，它们的生命周期行为在不同版本的 Android 系统上行为可能还会不一致。这在某些边界条件下，还会引发一些难以排查的 bug，进一步增加我们 Android 程序员的维护成本。

在计算机世界里，大部分问题都可以通过增加一个抽象层来解决。Android 团队的做法就是推出了 Lifecycle 这个架构组件，用它来统一 Activity、Fragment 的生命周期行为。



同时维护Activity、Fragment



增加Lifecycle作为抽象层



有了 Lifecycle 以后，我们开发者就可以面向 Lifecycle 编程。比如说，我们希望实现一个通用的地理位置监听的 Manager，就可以这样做：

复制代码

```
1 // 代码段5
2
3 // 不关心调用方是Activity还是Fragment
4 class LocationManager(
5     private val context: Context,
6     private val callback: (Location) -> Unit
7 ): DefaultLifecycleObserver {
8
9     override fun onStart(owner: LifecycleOwner) {
10         start()
11     }
12
13     override fun onStop(owner: LifecycleOwner) {
14         stop()
15     }
16
17     private fun start() {
18         // 使用高德之类的 SDK 请求地理位置
19     }
20
21     private fun stop() {
22         // 停止
23     }
24 }
```

```
25 class LifecycleExampleActivity : AppCompatActivity() {
26
27     override fun onCreate(savedInstanceState: Bundle?) {
28         super.onCreate(savedInstanceState)
29         setContentView(R.layout.activity_life_cycle_example)
30
31         val locationManager = LocationManager(this) {
32             // 展示地理位置信息
33         }
34         lifecycle.addObserver(locationManager)
35     }
36 }
37
```

上面代码的 `LocationManager` 只需要实现 `DefaultLifecycleObserver` 这个接口即可，外部是在 `Activity` 还是在 `Fragment` 当中使用，根本不必关心。

## Lifecycle 与协程

通过前面课程的学习，我们知道，协程其实也是有生命周期的。也就是说，`Android` 和 `Kotlin` 协程都是有生命周期的。这就意味着，当我们在 `Android` 当中使用协程的时候，就要格外小心。

作为 `Android` 开发者，你一定知道内存泄漏的概念：当内存变量的生命周期大于 `Android` 生命周期的时候，我们就认为内存发生泄漏了。类似的，当协程的生命周期大于 `Android` 生命周期的时候，**协程也就发生泄漏了**。

这一点，`Android` 官方早就帮我们考虑到了。`Lifecycle` 还可以跟我们前面提到的 `KTX` 结合到一起，进一步为 `Kotlin` 协程提供支持。

## lifecycleScope.launch

```
androidx.lifecycle.LifecycleCoroutineScope
public final fun launchWhenResumed(
    block: suspend CoroutineScope.() -> Unit
): Job
```

Launches and runs the given block when the `Lifecycle` controlling this `LifecycleCoroutineScope` is at least in `Lifecycle.State.RESUMED` state.

The returned `Job` will be cancelled when the `Lifecycle` is destroyed.

Caution: This API is not recommended to use as it can lead to wasted resources in some cases. Please, use the `Lifecycle.repeatOnLifecycle` API instead. This API will be removed in a future release.

See Also: `Lifecycle.whenResumed`, `Lifecycle.coroutineScope`

Gradle: androidx.lifecycle:lifecycle-runtime-ktx:2.4.0@aar

```
launchWhenResumed {...} (bl
launch(context: Coro... Job
launch {...} (... , b... Job
launchWhenCreated {...} Job
launchWhenStarted {...} Job
^↓ and ^↑ will move caret down and up in the editor Next Tip
```

在 `Activity`、`Fragment` 当中，KTX 还提供了对应的 `lifecycleScope`，它本质上就是一个：与生命周期绑定的协程作用域。

复制代码

```
1 // 代码段6
2
3 // 1
4 public val LifecycleOwner.lifecycleScope: LifecycleCoroutineScope
5     // 2
6     get() = lifecycle.coroutineScope
7
8 public abstract class LifecycleCoroutineScope internal constructor() : Coroutine
9     internal abstract val lifecycle: Lifecycle
10
11 public fun launchWhenCreated(block: suspend CoroutineScope.() -> Unit): Job
12     lifecycle.whenCreated(block)
13 }
14
15 public fun launchWhenStarted(block: suspend CoroutineScope.() -> Unit): Job
16     lifecycle.whenStarted(block)
17 }
18
19 public fun launchWhenResumed(block: suspend CoroutineScope.() -> Unit): Job
20     lifecycle.whenResumed(block)
21 }
22 }
```

在 Android 当中，Activity 和 Fragment 都会实现 LifecycleOwner 这个接口，代表它们都是拥有生命周期的组件。注释 1 处，这里使用了 Kotlin 的扩展属性，为 LifecycleOwner 扩展了 lifecycleScope。它的类型是 LifecycleCoroutineScope，而它其实就是 CoroutineScope 的实现类。

lifecycleScope 这个属性的具体实现，其实是通过注释 2 处的自定义 getter() 实现的，也就是：Lifecycle.coroutineScope。

 复制代码

```
1 // 代码段7
2
3 public val Lifecycle.coroutineScope: LifecycleCoroutineScope
4     get() {
5         while (true) {
6             // 1
7             val existing = mInternalScopeRef.get() as LifecycleCoroutineScopeIr
8             if (existing != null) {
9                 return existing
10            }
11            // 2
12            val newScope = LifecycleCoroutineScopeImpl(
13                this,
14                SupervisorJob() + Dispatchers.Main.immediate
15            )
16            //3
17            if (mInternalScopeRef.compareAndSet(null, newScope)) {
18                newScope.register()
19                return newScope
20            }
21        }
22    }
```

可以看到，Lifecycle.coroutineScope 仍然是一个扩展属性。它的逻辑其实也很简单，主要是分为了三个步骤：

- 第一步，检查是否存在缓存的 CoroutineScope，如果存在，那就直接返回即可。
- 第二步，如果不存在缓存，那就创建一个新的协程作用域。在创建的作用域的时候，用到了两个我们熟悉的概念：SupervisorJob、Dispatchers.Main，它们都是协程上下文的元素，前者是用来隔离协程异常传播的，后者是指定协程执行线程的。
- 第三步，更新缓存，并且调用 register() 绑定 scope 与 Lifecycle 的关系，最后返回。



接下来，我们打破砂锅问到底，看看 `register()` 的具体逻辑是什么：

 复制代码

```
1 // 代码段8
2
3 internal class LifecycleCoroutineScopeImpl(
4     override val lifecycle: Lifecycle,
5     override val coroutineContext: CoroutineContext
6     // 2
7 ) : LifecycleCoroutineScope(), LifecycleEventObserver {
8     init {
9         if (lifecycle.currentState == Lifecycle.State.DESTROYED) {
10             coroutineContext.cancel()
11         }
12     }
13
14     // 1
15     fun register() {
16         launch(Dispatchers.Main.immediate) {
17             if (lifecycle.currentState >= Lifecycle.State.INITIALIZED) {
18                 lifecycle.addObserver(this@LifecycleCoroutineScopeImpl)
19             } else {
20                 coroutineContext.cancel()
21             }
22         }
23     }
24
25     // 3
26     override fun onStateChanged(source: LifecycleOwner, event: Lifecycle.Event)
27         if (lifecycle.currentState <= Lifecycle.State.DESTROYED) {
28             lifecycle.removeObserver(this)
29             coroutineContext.cancel()
30         }
31     }
32 }
33
34 public interface LifecycleEventObserver extends LifecycleObserver {
35     void onStateChanged(@NonNull LifecycleOwner source, @NonNull Lifecycle.Event
36 }
```

上面的代码一共有三个注释，我们一个个来看：

- 注释 1，`register()`，可以看到，它的逻辑其实很简单，主要就是调用了 `addObserver()`，将自身作为观察者传了进去。之所以可以这么做，还是因为注释 2 处的 `LifecycleEventObserver`。

- 注释 2，LifecycleEventObserver，它其实就是一个 SAM 接口，每当 LifecycleOwner 的生命周期发生变化的时候，这个 onStateChanged() 方法就会被调用。而这个方法的具体实现则在注释 3 处。
- 注释 3，这里的逻辑也很简单，当 LifecycleOwner 对应的 Activity、Fragment 被销毁以后，就会调用 removeObserver(this) 移除观察者，最后，就是最关键的 coroutineContext.cancel()，取消整个作用域里所有的协程任务。

这样一来，就能保证 Lifecycle 与协程的生命周期完全一致了，也就不会出现协程泄漏的问题了。

## 小结

这节课，我们主要了解了 Android 当中的 Jetpack，它是 Android 官方提供给开发者的一个开发套件，可以帮助我们开发者提升开发效率。Jetpack 当中其实有 [几十个库](#)，在这节课里，我们是着重讲解了其中的 KTX 与 Lifecycle。

- KTX，主要是依托 Kotlin 的扩展能力，为 Android 原有 API 增加新的：扩展函数、扩展属性、高阶函数、命名参数、参数默认值、协程支持。
- Lifecycle，其实就是 Android 的生命周期组件。它统一封装了 Activity、Fragment 等 Android 生命周期的组件。让我们开发者可以只关注 Lifecycle 的生命周期，而不用在意其他细节。
- KTX 还为 Lifecycle 增加了协程支持，也就是 lifecycleScope。在它的底层，这个协程作用域和宿主的生命周期进行了绑定。当宿主被销毁以后，它可以确保 lifecycleScope 当中的协程任务，也跟着被取消。

所以，对于 Android 开发者来说，Kotlin 和 Jetpack 是一个“你中有我，我中有你”的关系，我们把它们称为“天生一对”一点儿也不为过。

## 思考题

在 Android 中使用协程的时候，除了 lifecycleScope 以外，我们还经常会使用 ViewModel 的 viewModelScope。你能结合前面协程篇、源码篇的知识点，分析出 viewModelScope 的实现原理吗？

```
1 class MyViewModel : ViewModel() {
2     private val _persons: MutableLiveData<List<Person>> = MutableLiveData()
3     val persons: LiveData<List<Person>> = _persons
4
5     fun loadPersons() {
6         viewModelScope.launch {
7             delay(500L)
8             _persons.value = listOf(Person("Tom"), Person("Jack"))
9         }
10    }
11 }
12
13 // 扩展属性
14 public val ViewModel.viewModelScope: CoroutineScope
15     get() {
16         val scope: CoroutineScope? = this.getTag(JOB_KEY)
17         if (scope != null) {
18             return scope
19         }
20         return setTagIfAbsent(
21             JOB_KEY,
22             // 实现类
23             CloseableCoroutineScope(SupervisorJob() + Dispatchers.Main.immediat
24         )
25     }
26
27 // 实现了Closeable的CoroutineScope
28 internal class CloseableCoroutineScope(context: CoroutineContext) : Closeable,
29     override val coroutineContext: CoroutineContext = context
30
31     override fun close() {
32         // 取消
33         coroutineContext.cancel()
34     }
35 }
36
37 public abstract class ViewModel {
38
39     @Nullable
40     private final Map<String, Object> mBagOfTags = new HashMap<>();
41     private volatile boolean mCleared = false;
42
43
44     @SuppressWarnings("WeakerAccess")
45     protected void onCleared() {
46     }
47
48     @MainThread
49     final void clear() {
50         mCleared = true;
51
52         if (mBagOfTags != null) {
```


```

53         synchronized (mBagOfTags) {
54             for (Object value : mBagOfTags.values()) {
55                 // 调用scope的close()
56                 closeWithRuntimeException(value);
57             }
58         }
59     }
60     onCleared();
61 }
62
63 // scope暂存起来
64 @SuppressWarnings("unchecked")
65 <T> T setTagIfAbsent(String key, T newValue) {
66     T previous;
67     synchronized (mBagOfTags) {
68         previous = (T) mBagOfTags.get(key);
69         if (previous == null) {
70             mBagOfTags.put(key, newValue);
71         }
72     }
73     T result = previous == null ? newValue : previous;
74     if (mCleared) {
75
76         closeWithRuntimeException(result);
77     }
78     return result;
79 }
80
81 private static void closeWithRuntimeException(Object obj) {
82     if (obj instanceof Closeable) {
83         try {
84             ((Closeable) obj).close();
85         } catch (IOException e) {
86             throw new RuntimeException(e);
87         }
88     }
89 }
90 }

```

分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 20 元

 生成海报并分享

上一篇 33 | Java Android开发者还会有未来吗？

下一篇 35 | 用Kotlin写一个GitHub Trending App

## 精选留言 (1)

写留言



Paul Shan

2022-04-11

思考题:viewModelScope是一个CloseableCoroutineScope，这个对象是懒加载的，第一次使用的时候才会创建，一旦创建以后，这个对象有一个close函数，会在ViewModel clear的时候调用，确保了viewModelScope的Coroutine scope和viewModel生命周期一致。了viewModelScope总体上和lifecycle的scope实现类似。区别是创建的时候，lifecycle用的是无锁+不断循环+compareAndSet方式，而viewModelScope实现的是synchronized带锁的方式，请问老师Android为什么会在两种类似的情况下采用不同的线程同步策略？

作者回复: Lifecycle只是简单的单个状态变更，所以用CAS能够最大程度保证效率。ViewModel使用synchronized，主要还是因为其中的mBagOfTags，它是一个Map，Android官方因为一些旧系统的限制，导致无法使用ConcurrentHashMap，所以才出此下策。

共 2 条评论 >

👍 1