


Fragment 的过去、现在和将来



谷歌开发者 
已认证的官方帐号

已关注

41 人赞同了该文章

Fragment 是 Android 中历史十分悠久的一个组件，它在 API 11 被加入，时至今日已成为 Android 开发中最常用的组件之一。Fragment 有了哪些新特性、修复了哪些问题，都是开发者们十分关心的话题。下面我们就来重新说一说 Fragment —— 不仅仅是说现在的 Fragment，还会回顾它的发展，并让您一瞥它未来的样子。

Fragment 的诞生与发展

不知道您是否还记得 "上古时期"，在那些还没有 Fragment 的日子，几乎所有逻辑都被放在了 Activity 中，使得 Activity 臃肿而又混乱。此时，Fragment 作为一个微型 Activity 而诞生，迈出了缩减 Activity 之路上的一小步。

不过 "欲戴王冠，必承其重"，Fragment 由此继承了诸多本来是为 Activity 设计的 API 和组件。其中有些组件，其实应该被设计为独立的 View，比如当年的 Action Bar，这个组件现在已经被 Toolbar 代替了；又比如现今已经基本没人使用的 Context Menus。API 这部分就更复杂一些，所有以前要发送到 Activity 的信息，现在也要发送到 Fragment，我们处理权限时很常用的 `onActivityResult` 就是这种情况下的产物；当 Android 加入运行时权限时，Fragment 理所当然的也要支持，因为 Activity 已经支持了。类似的 API 还有 `onMultiWindowModeChanged` 以及 `onPictureInPictureModeChanged`。

遵循着成为一个微型 Activity 的设计初衷，Fragment 自然而然的就得到了这些功能。但是回过头来看，这些功能其实并不是专门为 Fragment 设计的 —— 随便一个什么东西，有了这些回调，似乎都能胜任 Fragment 的功能。这种状况使得我们开始转变思路，并尝试抛弃让 Fragment 去做微型 Activity 的想法，如今的 Fragment 正是由此而来。

Fragment 的现状

我们的想法产生改变，还是 2011 年的事，到今天已经经历了很长的时间。这期间我们花费了很多的精力去重新构思 Fragment 的定位。我们希望 Fragment 成为一个真正的核心组件，它应该拥有可预测的、合理的行为，不应该出现随机错误，也不应该破坏现有的功能。

其实我们希望挑个时间发布 Fragment 的 2.0 版，它将只包含那些新的、好用的 API。但在时机成熟之前，我们会在现有的 Fragment 中逐步加入新的并弃用旧的 API，并为旧功能提供更好的替代方案。当没人再使用已弃用的 API 时，迁移到 Fragment 2.0 就会变得很容易。接下来我就来讲讲，我们为此所做的一些工作。

FragmentScenario

首先我要说，合理的 API 应当是可测试的。不能被测试的代码不是好代码，现在已经 2020 年了，我们也希望 Fragment 能在这方面做得更好。于是，通过与 AndroidX 团队紧密合作，我们开发出了测试工具 `FragmentScenario`，它可以用来单独对 Fragment 进行测试。`FragmentScenario` 基于 `ActivityScenario` 实现，这也意味着它同样适用于 Instrumentation 和 Robolectric 测试。同时它的 API 十分简洁，它最主要的方法就是 `onFragment`，这个方法接收一个 Lambda 表达式，而 Lambda 表达式则在其中返回已存在的 Fragment 实例。同时 `FragmentScenario` 也提供了方便测试生命周期和重建 Fragment 的 `Hook` 方法。

如下示例代码来说，首先使用 `launchFragmentInContainer<MyFragment>()` 创建 `FragmentScenario` 对象，这一步操作将会帮您完成创建 Fragment 的整个流程。接下来就可以进



成。最后只要在 onFragment 中检查 Fragment 的状态，就可以确认 Fragment 是否有正确处理点击事件。

```
@Test
fun testEventFragment() {
    val scenario = launchFragmentInContainer<MyFragment>()
    onView(withId(R.id.refresh)).perform(click())
    scenario.onFragment { fragment ->
        // 检查 Fragment 有没有正确处理点击事件
    }
}
```

如果需要测试一些更加复杂的情况，比如 Fragment 的生命周期切换，您可以调用 Scenario 的 moveToState() 方法，来让 Fragment 触发各种生命周期。测试 Fragment 的重建也是类似操作，假如您想要测试是否正确存储和恢复了 Fragment 的状态信息，只需要调用 recreate() 方法，就可以检查 Fragment 重建前后状态信息的保存情况，就是这么简单。

```
@Test
fun testEventMoveToCreatedFragment() {
    val scenario = launchFragmentInContainer<MyFragment>()
    scenario.moveToState(State.CREATED)
}
```

```
@Test
fun testEventFragment() {
    val scenario = launchFragmentInContainer<MyFragment>()
    scenario.recreate()
}
```

FragmentFactory

讲到 Fragment 的重建，就联想到 Fragment 的实例化。Fragment 已经有很多种实例化方式了，后来又有了 FragmentScenario。我们希望能统一这些方法，而解决方案便是 FragmentFactory，它让我们可以注入 Fragment 的构造方法，也顺带解除了 Fragment 必须有一个无参构造方法的限制。

下面是一个简单的 FragmentFactory，它只有一个方法 —— instantiate，您只需要在这个方法中传入 Fragment 的类名，随后 super.instantiate() 方法就会使用反射调用对应 Fragment 的无参构造方法。正如我们在《Android 依赖注入指南》这场演讲中提到的，我们很乐意通过这种模式来减少使用者的重复工作。而如果您需要传入参数，则可以将参数传入 FragmentFactory 并通过构造方法注入将参数传入 Fragment。

接下来，您需要将 FragmentManager 的 FragmentFactory 设置为您的 FragmentFactory。这一步最好放在 super.onCreate() 之前，因为它是重新实例化 Fragment 的地方。

```
private class MyFactory() : FragmentFactory() {
    override fun instantiate(
        classLoader: ClassLoader,
        className: String
    ) = when (className) {
        MyFragment::class.java.name -> MyFragment()
        else -> super.instantiate(classLoader, className)
    }
}

override fun onCreate(savedInstanceState: Bundle?) {
    supportFragmentManager.fragmentFactory = MyFactory()
    super.onCreate(savedInstanceState)
}
```



为了保证 API 的一致性，我们还准备通过下面的方式统一其他地方创建 Fragment 的方式。比如 Commit 操作，我们代理了您的 FragmentFactory，现在您只需要使用 Fragment 的类名，通过一行简单的代码，便能完成 Fragment 的创建、添加和初始化。

```
// 通过类名来添加 Fragment
supportFragmentManager.commit {
    add<MyFragment>(R.id.container)
}
```

类似的，使用 FragmentScenario 时，只需要传入您的 FragmentFactory 即可。这个 FragmentFactory 既可以是只用来模拟依赖的虚拟 Factory，也可以是用于更多测试的真实 FragmentFactory。

```
// 使用自定义 FragmentFactory 创建 FragmentScenario
val scenario =
    launchFragmentInContainer<MyFragment>(factory = MockFactory())
```

FragmentContainerView

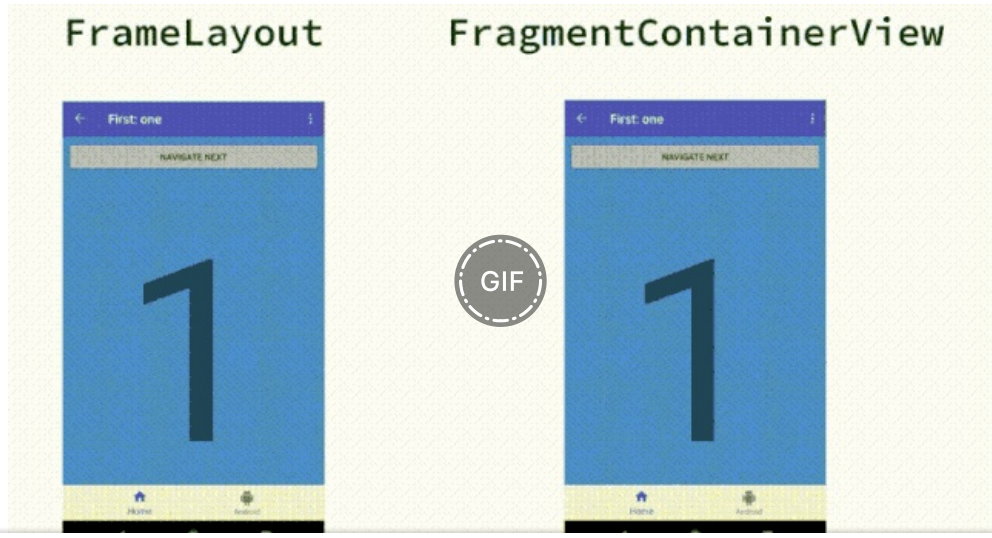
关于 API 的一致性，我们也尝试解决了 Fragment 的另一个一致性问题。

我们发现在添加 Fragment 时，通过 <Fragment> 标签添加与通过 FragmentTransaction 使用的是完全不同的两套系统。为了提供行为一致的 API，我们创建了 FragmentContainerView，并把它作为 Fragment 专属的容器。

FragmentContainerView 继承于 FrameLayout，但它只允许填充 FragmentView。它同时也替代了 <Fragment> 标签，只要在 class 属性中传入类名即可。由于 FragmentContainerView 内部使用的是 FragmentTransaction，所以无需担心，稍后在替换这个 Fragment 时也不会出现问题。

```
<!-- 与在 onCreate 中调用 add() 方法效果相同 -->
<androidx.fragment.app.FragmentContainerView
    class="com.example.MyFragment"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

FragmentContainerView 也让我们有机会解决一些动画问题。例如 Fragment 在 Z 轴的层级问题。如下图所示，我们可以看到在 FrameLayout 中，Fragment 切换时没有显示动画，而是整个跳出到了屏幕上。这种问题是由于切入的 Fragment 和它的动画位于之前的 Fragment 的层级之下导致的。而 FragmentContainerView 会确保 Fragment 间的层级关系处于正确的状态，我们就可以看到切换动画了。



另一个长期困扰我们的问题，是在 Fragment 中处理系统回退事件。为了解决这个问题，我们加入了 onBackPressedDispatcher。我们没有选择在 Fragment 中添加这个 API，而是将其加入了 Activity 中。现在任何组件都可以通过依赖 Activity 来处理回退事件。

下面是一段使用 onBackPressedDispatcher 的示例代码。您可以看到，首先 Fragment 从调用它的 Activity 中获取 onBackPressedDispatcher 对象，然后通过 addCallback() 方法创建了一个 onBackPressedCallback，由于 Fragment 是 LifecycleOwner，所以这里可以传入 "this"。在此示例中，如果用户触发了回退操作，就会弹出一个确认窗口，而如果用户随后表示无论如何都想要退出的话，您可以先使回调失效，然后就可以执行默认的回退操作。

```
val dispatcher by lazy { requireActivity().onBackPressedDispatcher }
lateinit var callback: OnBackPressedCallback

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    callback = dispatcher.addCallback(this) {
        showConfirmDialog()
    }
}

private fun onConfirm() {
    callback.enabled = false
    dispatcher.onBackPressed()
}
```

所以这里其实并没有新的 API，只是整合了 Fragment 和架构组件现有功能。而我们接下来也打算进一步加深与架构组件的整合。举个例子，在 Fragment 中理应可以方便地获得 ViewModel 实例，但现实状况却稍微有些麻烦。为了解决这个问题，我们创建了一些 Kotlin 属性代理。如下面的代码所示，利用这些属性代理，您可以轻松获得不同作用域的 ViewModel。

```
// 让获取 ViewModel 实例变得简单
val viewModel : MyViewModel by viewModels()
val navGraphViewModel: MyViewModel by navGraphViewModels(R.id.main)
val activityViewModel: MyViewModel by activityViewModels()
```

我们也从 Lifecycle 组件中受益良多。比如，我们不再使用自定义的生命周期方法 setUserVisibleHint，取而代之的是在添加 Fragment 到 ViewPager 或 Adapter 时调用统一的生命周期。这便是 ViewPager2 目前的工作机制，只有当前页面的 Fragment 会调用 onResume 方法。

```
// 设置只让当前展示的 Fragment 调用 onResume() 方法
class MyAdapter : FragmentPagerAdapter(BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT)
```

Fragment 的未来

前面讲过的功能大多在 Fragment 1.1 中已经提供，与此同时，我们强烈建议使用 FragmentContainerView 容器来存储动态添加的 Fragment，而不要使用 FrameLayout 或其他布局。

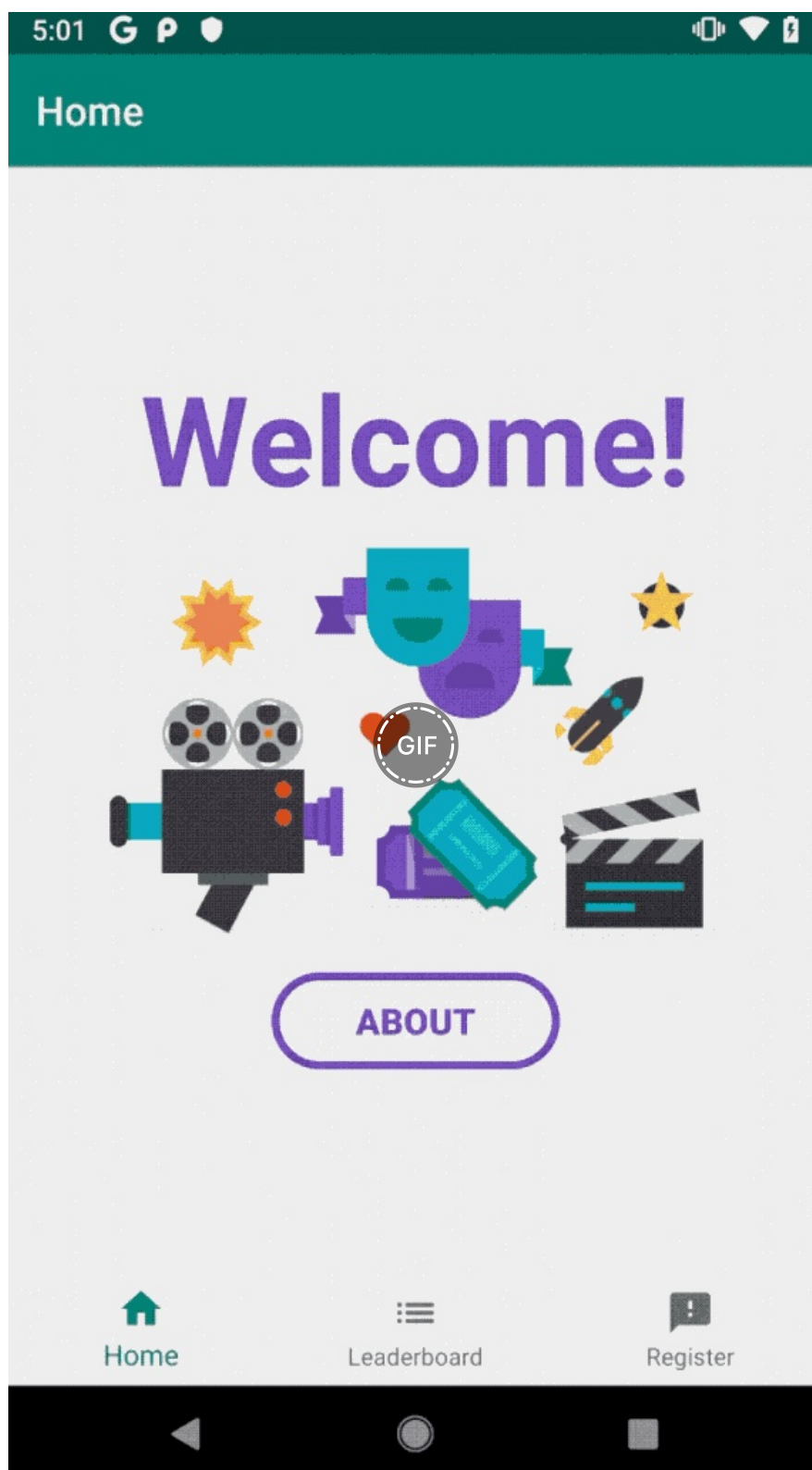
当然，未来我们还将对 Fragment 做出许许多多的改进，下面我就来介绍几个我们当前正在进行的长期规划。不过要注意的是，接下来部分内容目前还没有正式推出，所以一些细节可能会有改变。

多重回退栈 (Multiple Back Stack)

首先再讲的是多重回退栈 (Multiple Back Stack)。我们知道在 Android 中，总是会有一个 Activity，

持单一回退栈和多重回退栈的模型，好让屏幕上不可见的 Fragment 也能保存自己的状态，从而避免状态的丢失。与此相关的使用场景，比较典型的就底部导航一类的导航视图。

下面是一个我们的示例应用。我们想要做的事情就是让应用中每个底部标签页都拥有自己的栈，这样它们就能保存各自的状态。而当您在这些标签页间切换时，我们也将帮您处理好从一个栈到另一个栈时状态的保存和恢复。



Fragment 间的通讯问题

我们想要解决的另一个问题与返回结果有关。

一直以来，诸如如何在 Fragment 间通讯，或者说如何在 Android 的各种组件间通讯的这类问题都深深困扰着我们。想要在 Fragment 间通讯，方法有很多，它们有好有坏。而这正体现出 Fragment 在这方面的 API 设计不佳。我们可以设计一些用于 Fragment 间通讯的 API，并且让它们在其互互相持有依赖的前提下工作，但是这样的话，当前的 Fragment 将无法感知

其它 Fragment 的生命周期。如果通讯的 Fragment 处在不活跃的生命周期中，那么通讯也将失败。



还有一个选项，是使用类似 onActivityResult 的 API。但我们所考虑的，不只是在 Fragment 之间通讯，而是希望能设计出一套公用的 API。它应当同时兼容 Activity、Fragment 等可能的导航组件，这样就算不知道对方的类型，也能建立通讯。

简化 Fragment 的生命周期

最后要说的问题，是 Fragment 的生命周期。当前 Fragment 的生命周期十分复杂，它包含了两套不同的生命周期。Fragment 自己的生命周期从它被添加到 FragmentManager 的时候开始，一直持续到它被 FragmentManager 移除并销毁为止；而 Fragment 所包含的视图，则有一个完全分离的生命周期。当您的 Fragment 进入回退栈时，视图将会被销毁。但 Fragment 则会继续存活。

于是我们产生了一个大胆的想法：将两者合二为一会怎么样？在 Fragment 视图销毁时便销毁 Fragment，想要重建视图时就直接重建 Fragment，这样的话将大大减少 Fragment 的复杂度。而诸如 FragmentFactory 和状态保存一类，以往在 onConfigurationChange、进程的死亡和恢复时使用的方法，在这种情况下将会成为默认选项。

当然，这个改动将会是十分的巨大。我们目前处理的方式，是将它作为一个可选 API 加入到了 FragmentActivity 中。使用了这个新的 API，就可以开启生命周期简化过的新世界。

总结

我们前面讲了 Fragment 一些历史问题的由来，以及我们刚刚为它加入的一些特性，包括：

- FragmentScenario，Fragment 的测试框架
- FragmentFactory，统一的 Fragment 实例化组件
- FragmentContainerView，Fragment 专属视图容器
- OnBackPressedDispatcher，帮助您在 Fragment 或其他组件中处理返回按钮事件

最后还介绍了几个我们仍在开发中的功能：

- 多重回退栈
- 使 Fragment 以及其他导航组件间可以优雅的通讯
- 简化 Fragment 的生命周期

希望这些内容可以帮助您更好地使用和理解 Fragment。

我们正努力将文中提到的新特性带给各位开发者，而在此之前，如果您在使用 Fragment 时有任何问题和疑惑，可以使用 issuetracker.google.com 向我们提交反馈或功能请求，谢谢！

您也可以通过视频回顾 2019 Android 开发者峰会演讲 —— Fragment 的过去、现在和将来：

<https://v.qq.com/x/page/z3027g2zz5x.html>
v.qq.com



[点击这里](#)了解更多 Fragment 相关内容



发布于 06-22

Android 开发 Android

推荐阅读

死磕Android_App 启动过程 (含 Activity 启动过程)

. 前言Activity是日常开发中最常用的组件,系统给我们做了很多很多的封装,让我们平时用起来特别简单,很顺畅.但是你有没有想过,系统内部是如何启动一个Activity的呢?Activity对象是如何创建...

萧风寒月6... 发表于Andro...



#Android# 使用 Fragment 构建 Presenter

nekoc... 发表于『Andr...

Android面试必备26题（阿里腾讯总结）

1.Activity的启动过程（不要回答生命周期）
http://blog.csdn.net/luoshengyang
的启动模式以及使用场景 （1）manifest设置， （2）startActivity flag ht...
[已重置]

Android 性能优化 - 常见内存泄露分析

本文来源于微信公众号『玉刚说』中的一篇文章，原文太长，懒得全帖出来，大家去原地址看吧：
Android 性能优化 - 详解内存优化的来龙去脉1. 永远的单例

夫人不吃鱼

8 条评论

切换为时间排序

写下你的评论...

 时代在召唤

06-22

确实应该简化fragment的生命周期，当前生命周期实在是太多了

👍 赞

 思慕的人 回复 时代在召唤

06-22

建议还是自己封装

👍 赞

 宝哥

06-22

太长不看。

👍 1

 宝哥

06-22



Michael Lee

06-22

说实话，fragment一整套东西都做得挺烂的

👍 5



Madara 回复 Michael Lee

06-23

至少Google终于意识到这点并且已经开始改正了

👍 1



java

06-23

以后flutter吧

👍 赞



WhiteLau

07-04

flutter中所有界面都是widget的理念更扁平更先进。

👍 赞