00. 来源

- 我们在推荐页中进行视频的播放，因此需要在里面初始化视频播放器。具体是在OnViewCreated阶段调用initVideoPlayer方法。
- 该方法如下：
  - 创建VideoPlayer实例。VideoPlayer是我们自定义的类，它继承自FrameLayout。
  - 设置它布局参数为MATCH_PARENT
  - 将其添加为生命周期观察者，使得它可以响应RecommendFragment的生命周期事件。
  - ```
    private fun initVideoPlayer() {
        var params = LayoutParams(LayoutParams.MATCH_PARENT,
    LayoutParams.MATCH_PARENT)
        videoView = VideoPlayer(requireActivity())
        /**
         * 设置布局参数
         */
        videoView.layoutParams = params
        /**
         * 将 videoView 添加为生命周期观察者，以便它能响应生命周期事件（如 onResume、
    onPause 等）。
         */
        lifecycle.addObserver(videoView)
    }
    ```

01.VideoPlayer介绍

- 这是它的来源，那么我们来具体看看VideoPlayer吧。
- 视图逻辑：
  - VideoPlayer继承自FrameLayout。可是FrameLayout只是布局类，无法显示具体的播放器啊？
  - 具体的视频播放器其实是它的子视图playView的布局。这个子视图playView会通过视图绑定，通过inflate，指定parent为this，attachToParent为true，将自身挂载到VideoPlayer的Framelayout中。
  - 子视图playView是一个FrameLayout，其内部包裹的一个exoplayer2.ui.PlayerView。他们都是match_parent的。

> - 我们这里有了VideoPlayer的视图了，可是，我们遇到一个问题，就是它并没有在RecommendFragment的视图xml中声明啊？它是如何出现在里面的呢？
>   - 这里其实采用的是动态添加。具体来说，我们首先调用videoView.parent.removeView，将其从父视图中移除（如果有的话），然后调用推荐页视图rootView.addView将其动态加载进里面。这里我们指定它的父视图层级为0.
    ```

```
/**
 * 将videoView从原来的父视图中移除，并添加到新的rootView中,我之前还很困惑为什么xml中没有videoView的布局定义呢，原来在这里啊。
 */
private fun dettachParentView(rootView: ViewGroup) {
    //1.添加videoView到当前需要播放的item中,添加进item之前，保证videoView没有父view
    videoView?.parent?.let {
        (it as ViewGroup).removeView(videoView)
    }

    rootView.addView(videoView, 0)
}
```

- 代码逻辑：
  - 首先，VideoPlayer对象创建时，执行实例代码块init中的initPlayer方法，该方法具体如下：
    - 将 SimpleExoPlayer 类型的播放器实例与 playerview 播放器视图绑定。
    - 禁用 PlayerView 的控制器。默认情况下，PlayerView 会显示播放、暂停、进度条等控制按钮，禁用后就不显示了，保证用户的沉浸式体验。
    - 设置了mplayer的playWhenReady自动播放和repeatMode循环模式。
    - 这样，我们既有播放器视图，又有播放器实例，还有对应的播放器视图和实例的设置。
    - ```
      /**
       * 初始化播放器：
       * 将 mPlayer 这个 SimpleExoPlayer 实例与 playerview 绑定。
       * 禁用 PlayerView 的控制器。默认情况下，PlayerView 会显示播放、暂停、进度条等控制按钮。设置 useController 为 false 会隐藏这些控制器。
       * 设置播放器在准备好后立即播放视频。playWhenReady 是一个布尔值，表示播放器是否应该在所有资源准备就绪后自动开始播放。
       * 设置播放器的重复模式。REPEAT_MODE_ALL 表示所有播放列表中的媒体项播放完毕后，重新从第一个媒体项开始播放。
       */
      private fun initPlayer() {
          binding.playerview.player = mPlayer
          binding.playerview.useController = false
          mPlayer.playWhenReady = true
          mPlayer.repeatMode = Player.REPEAT_MODE_ALL
      }
      ```
  - 这个 SimpleExoPlayer类型播放器实例mPlayer是怎么来的呢?
    - VideoPlayer有几个关键的实例成员变量，如：trackSelector，轨道选择器，用于管理音视频轨道的选择。loadControl：用于设置加载和缓存策略。 mPlayer：ExoPlayer库的核心播放器实例，负责处理媒体的播放和控制。
    - 我们在初始化阶段会进行实例变量的初始化。
    - 先介绍trackSelector，我们这里使用DefaultTrackSelector，这是ExoPlayer的默认轨道选择器，用于管理音视频轨道的选择，通过setParameters设置优先中文音频和720p
      ```

        - ```
          private val trackSelector: TrackSelector =
          DefaultTrackSelector(context).apply {
              setParameters(buildUponParameters().apply {
                  setPreferredAudioLanguage("zh") // 优先中文音频
                  setMaxVideoSize(1280, 720) // 限制720P分辨率
              })
          }
          ```
    - 接着是loadControl：用于设置加载和缓存策略，通过DefaultLoadControl.Builder()去构建。缓存达到(700ms)即可开始播放，加载会持续到填满MAX_BUFFER_MS(7000ms)的缓冲区，当剩余缓冲 < REBUFFER_MS(1000ms)时触发加载。
        - ```
          /**设置加载和缓存策略：
           *
           * 缓冲达到PLAYBACK_BUFFER_MS(700ms)即可开始播放
           * 加载会持续到填满MAX_BUFFER_MS(7000ms)的缓冲区
           * 当剩余缓冲 < REBUFFER_MS(1000ms)时触发加载
           */
          val MIN_BUFFER_MS = 5_000 // 初始缓冲的最低要求。
          val MAX_BUFFER_MS = 7_000 // 缓冲的最大容量。
          val PLAYBACK_BUFFER_MS = 700 // 允许开始播放的最小缓冲。
          val REBUFFER_MS = 1_000 // 重新缓冲的阈值。
          val loadControl = DefaultLoadControl.Builder()
              .setPrioritizeTimeOverSizeThresholds(true)//缓冲时时间优先级高于大小
              .setBufferDurationsMs(MIN_BUFFER_MS, MAX_BUFFER_MS, PLAYBACK_BUFFER_MS, REBUFFER_MS)
              .build()
          ```
    - 然后是mPlayer，它是ExoPlayer库的核心播放器实例。
        - 我们使用SimpleExoPlayer.Builder构建也就是ExoPlayer的构建器模式，我们指定了它的轨道和缓冲控制为上面的trackSelector 和 loadControl，最后调用build进行构建。
        - ```
          private val mPlayer : SimpleExoPlayer by lazy {
                      SimpleExoPlayer.Builder(context)
                          .setTrackSelector(trackSelector)
                          .setLoadControl(loadControl) //构建内存缓冲
                          .build()
              }
          ```
- 然后是被外部调用的它的核心方法：playVideo播放视频。它的流程就是：设置媒体源，准备播放，开始播放。但是根据数据来源有两种播放视频的方式，分别是指定媒体源播放，url构建媒体源播放。（后者是需要根据url我们自己构建媒体源）。
    - 假如我们传递的参数是MediaSource：
        - 设置媒体源，准备播放，开始播放
        - ```
          fun playVideo(mediaSource: BaseMediaSource) {
              mPlayer.setMediaSource(mediaSource)
              mPlayer.prepare()
              mPlayer.play()
          }
          ```
    - 假如我们传递的参数是url，那么我们需要先构建媒体源。

- 构建媒体源
  - 使用MediaItem.fromUri(url)创建一个MediaItem对象。其是ExoPlayer中对要播放的内容的抽象描述。
    - `val mediaItem = MediaItem.fromUri(url)`
  - 配置缓存数据源，也就是CacheDataSource，这个缓存策略会允许ExoPlayer在播放时缓存数据，这有什么用呢？ 这里的setCache(cache)指定了本地缓存文件，而setUpstreamDataSourceFactory设置了默认的数据源工厂，用于当缓存中没有数据时从原始URL获取数据
    - ```
      val dataSourceFactory =
      CacheDataSource.Factory().setCache(cache).setUpstreamDa
      taSourceFactory(
                DefaultDataSource.Factory(context))
      ```

> 这里的cache本地缓存文件是什么？
>
> 它也是一个实例成员变量，我们通过以下为其构造：
>
> - 我们指定缓存路径为tiktok_cache_file
> - 设置缓存策略为LRU
> - 通过SimpleCache构造方法传递上面两个参数去构建它。
> - ```
>   /**
>    * 缓存部分使用SimpleCache懒加载，路径为应用缓存目录下的
>   tiktok_cache_file。缓存策略为LRU，最近最少使用
>    */
>   val cache: SimpleCache by lazy {
>       val cacheFile =
>   context.cacheDir.resolve("tiktok_cache_file")
>       SimpleCache(cacheFile,
>   LeastRecentlyUsedCacheEvictor(MAX_CACHE_BYTE),
>   StandaloneDatabaseProvider(context))
>   }
>   ```

  - 使用上面的mediaItem和CacheDataSource 来创建媒体源。ProgressiveMediaSource.Factory指定数据源工程，然后`createMediaSource(mediaItem)`根据视频条目创建媒体源。
    - `val mediaSource = ProgressiveMediaSource.Factory(dataSourceFactory).createMediaSource(mediaItem)`
  - 将mediaSource设置给mPlayer（ExoPlayer实例），至此，mediaSource我们构建完成。接下来就是：准备播放，`mPlayer.prepare()`和开始播放，`mPlayer.play()`
    - `mPlayer.setMediaSource(mediaSource)`
- 准备播放，`mPlayer.prepare()`
- 开始播放，`mPlayer.play()`

02.生命周期方法

- onResume：当 Fragment恢复时，如果当前页面在推荐页，则继续播放视频。
      ```

```
override fun onResume(owner: LifecycleOwner) {
    super.onResume(owner)

    //返回时，推荐页面可见，则继续播放视频
    if (MainActivity.curMainPage == 0 && MainFragment.Companion.curPage
== 1) {
        play()
    }
}
```

- onPause：当 Fragment暂停时，暂停视频播放。`pause()`
- onStop：当 Fragment停止时，停止视频播放。`stop()`
- onDestroy：当 Fragment销毁时，释放视频播放器资源。`release()`