Table of Contents *generated with DocToc*

Android车载多媒体开发MediaSession框架理解（建议收藏）

背景图来源：<Googel.Connect your phone. Now hit the road(点我跳转)>

争取每一篇文章都是精华，每一篇文章都能做到后期维护

写在前面

大家最熟悉的Android系统应该是手机和平板设备上的，大部分人可能没想过Android系统和汽车有什么关系。实际上谷歌15年前就在布局汽车这个平台，发布了《**automotive(点我跳转)**》。今年进入车载行业以来，我最近对相关内容做了一些了解。下面将我所了解到的信息分享给大家

车载多媒体开发MediaSession框架，本篇.应该是.目前.国内网上关于MediaSession的最全面的一篇了，建议收藏。本文的目的之一就是通过梳理总结MediaSession框架相关的知识点，来完成即将升级的 **#*Media3** 知识点。同时顺便分享给各位同学

⚠ **注意**

> - media3称为下一代媒体框架，基于androidx可以替换掉一些compact库
> - media3包含了exoplayer，有track的能力，不用过度关注trackSession了
> - android auto 和android automotive是两个不同的概念

1、关键词

Android、MediaSession、MediaSession框架、车载多媒体开发、Android历史发布版本、media3、ExoPlayer、Automotive

> **《谷歌官网：媒体应用架构(点我跳转)》、《谷歌官网：Build media apps for cars(点我跳转)》**

2、目录结构

一、MediaSession概述

Android 5.0开始（当前最新是 Android14 U34 ）引入的媒体应用框架，分为媒体控制器MediaController（用于**界面UI**）和媒体会话MediaSession（用于**播放器Player**）。MediaSession框架，使用一套接口，减少了很多流程的繁琐和service的通信等，实现多个设备或者UI的统一调用，其代码可读性、结构耦合度（解耦UI和播放器：MediaPlayer、ExoPlayer等）方面都控制得非常好

MediaSession主要应用于播放音频或视频的多媒体应用中

⚠ **注意**

> 可以对比一下视频和音频的区别：还有传统播放音乐的APP架构的区别。在下面↓↓↓< *一.3、多媒体应用架构* 会进行补充说明

1、历史

2014年Google I/O会上发布了Android5.0，这个版本上引入了的媒体应用框架，也就是MediaSession。Android Auto也是首次亮相、于2015年3月19日发布，2015年Hyundai Sonata是第一个支持Android Auto的汽车

后面陆续发布了：N O P Q R S T U 各版本，下表为我整理的一个Android历史版本映射表

1.1、Android历史发布版本和API对照

下面记录Android版本号、发布日期、版本名称代号和API级别对照表

Android版本	发布日期	代号和API Level
Android 15	2024年10月15日	V→35
Android 14	2023年10月04日	U→34
Android 13	2022年08月15日	T→33
Android 12L	2022年2月，Android 12L Beta 3 版本发布，首次支持了 Pixel 6 和 Pixel 6 Pro	S→32
Android 12	2021年10月04日	S→31
Android 11	2020年09月08日	R→30
Android 10	2019年09月03日	Q→29
Android 9	2018年08月06日	Pie→28
Android 8.1	2017年12月5日	Orea→27
Android 8	2017年08月21日	Orea→26
Android 7.1.1	2016年12月05日	Nougat→25
Android 7	2016年08月22日	Nougat→24
Android 6	2015年10月05日	MarshMallow→23

(1)、统一各个AA中引用Android版本的代号

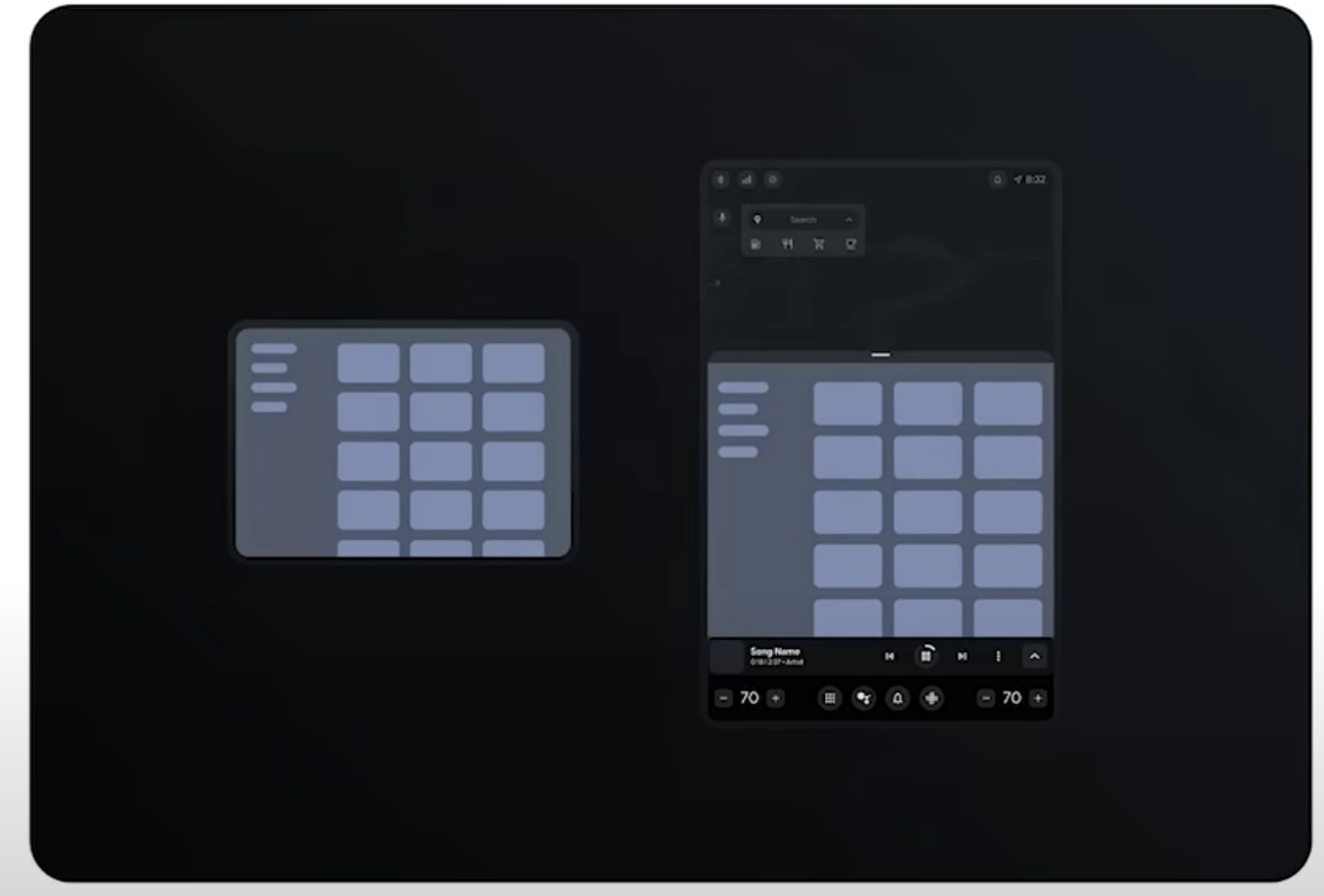
前面的数字是Android的版本号、中间的字母是Android的代号、后面的数字为Android的API版本号

⚠ **注意：**这是第4次统一说明标准了、未来在AA中涉及到**Android版本**的内容，都采用上面↑↑↑的格式做版本说明

2、概念类

2.1、Android Automotive OS

Automotive指车载系统。视频应用可在运行Android Automotive OS的停放车辆上运行，这一点通过使用与驱动你的移动应用代码几乎相同，这样针对可折叠设备和平板电脑的大屏幕优化，就改善了汽车内置屏幕的用户体验

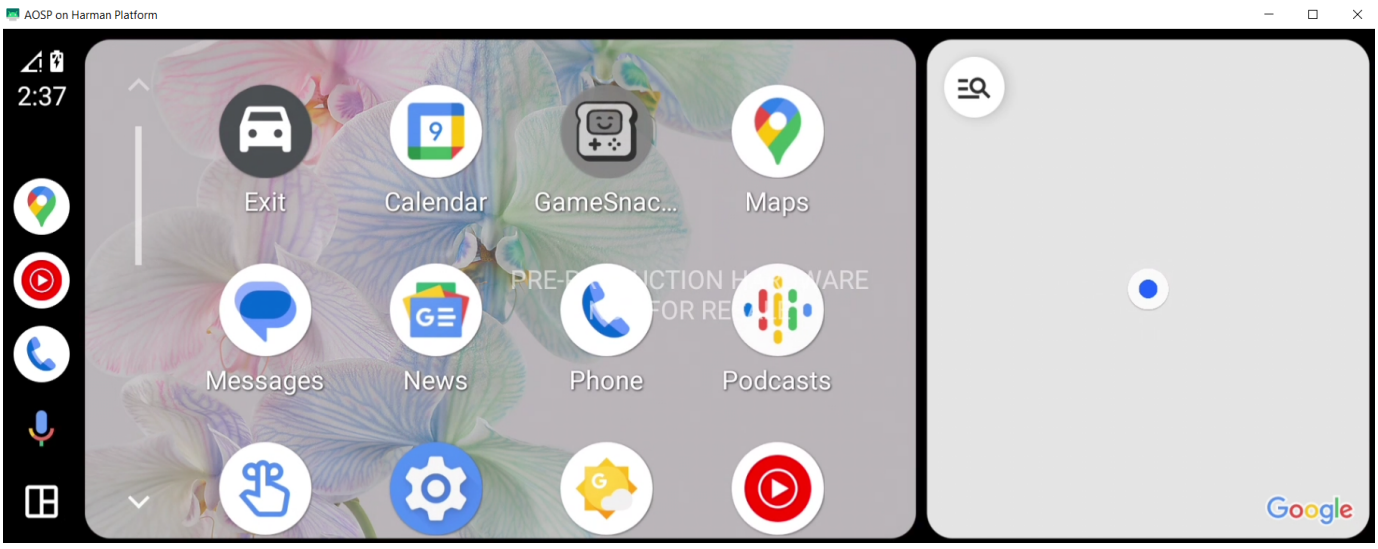


⚠ **注意**

> - MediaSession也就是运行在Automotive系统之上的API服务
> - 最新的Android14 U34、发布的Automotive OS可以很方便的驱动移动应用代码适配到车机上

2.2、Android Auto

Android Auto（我们一般简称AA）指车载投屏，是一个Android端的App，是专门为驾驶环境而设计的。类似还有CarLife投屏，CarPlay投屏、HiCar。当AA接到汽车屏幕上其界面看起来是下面这个样子



⚠ **注意**

> - 运行AA需要Android 5.0或更高版本的系统，并且还需要Google地图和Google Play音乐应用。国内一般用不了

> - AA是谷歌也就是Android自己的标准，CarLife是百度的标准，CarPlay是iOS(苹果）的投屏方案。**这三个都是必须掌握和学习的技术**
> - 用不了AA，国内很多车厂替代的方案就是，CarLife和CarPlay，HiCar是华为搞的。CarLife可以在Android和iOS上使用，CarPlay则只能使用iPhone手机才能使用投屏功能
> - HiCar是华为2019年9月，它是人-车-家全场景智慧互联解决方案。像问界M9也是搭载了HiCar的

用AA，CarLife，CarPlay播放音乐，他们都会用到MediaSession，这就是我为什么要在这里引出他们的原因

2.3、车载专业术语

在车载开发中有很多的专业术语，你需要有一定的了解。这里是网上关于 ****车联网常见缩写**，已经很详细了

否则，你看余承东怼行业内说连AEB都不知道

> AEB，全称Advanced Emergency Braking System，即高级紧急制动系统。这是一种主动安全技术，旨在通过车辆传感器和算法，自动检测前方障碍物和潜在危险，并在必要时自动刹车以避免或减少碰撞

2.4、开发播放操作词汇

Do Play, Pause and Next, Previous and shuffle, repeat Operations

(1)、播放&暂停

- PlaybackState.ACTION_PLAY
- PlaybackState.ACTION_PLAY_FROM_MEDIA_ID
- PlaybackState.ACTION_PAUSE

(2)、下一曲&上一曲

点按

- PlaybackState.ACTION_SKIP_TO_NEXT
- PlaybackState.ACTION_SKIP_TO_PREVIOUS

(3)、快进&快退/快倒.播放操作

长按

❶、快进播放操作

一般指长按下一曲键，视频或者音频会以2x倍速，逐渐向16x倍速的速率，快进播放，然后松开手或者鼠标，视频或者音频会恢复播放（继续以正常的1x倍速）

- PlaybackState.ACTION_FAST_FORWARD
- PlaybackState.STATE_FAST_FORWARDING 4处于快进状态

代码中有的命名：

- fastforward快进
- fast forwarding处于快进状态

❷、快退播放操作

一般指长按上一曲键，视频或者音频会以2x倍速快退播放，然后松开手或者鼠标，视频或者音频会恢复播放（继续以正常的1x倍速）

- PlaybackState.ACTION_REWIND

- PlaybackState.STATE_REWINDING 5处于快退状态
- Fast Reverse Playback快退播放（Fast Reverse快退）

代码中有的命名：

- fastrewind快退/快倒
- rewinding处于快退状态

(4)、倍速播放模式

❶、倍速播放

一般指设置倍速为1.5x、2x、2.5x、3x，视频或者音频会以对应的倍速播放

❷、慢进播放

一般指设置倍速为0.5x，视频或者音频会以0.5x的倍速播放

- Slow Forward Playback慢进播放

(5)、收藏或喜欢一首歌曲

- PlaybackState.ACTION_SET_RATING

(6)、拖动进度条

- PlaybackState.ACTION_SEEK_TO

(7)、播放模式

主要就是这三种，别搞太多，容易让用户迷惑

- 单曲循环--- 重复播放一首歌1_repeat : PlaybackState.ACTION_SET_REPEAT_MODE
- 顺序播放--- 按列表顺序播放
- 随机播放--- x_shuffle：PlaybackState.ACTION_SET_SHUFFLE_MODE

`shuffle`通常是指音响系统或播放器的一种功能。 它表示随机播放（Random Play）或者乱序播放（Shuffle Play），即不按原来的顺序，而是随机地播放歌曲、音频或其他媒体文件。 这种播放方式可以让驾驶者在长途驾驶中听到更多不同的音乐组合，减轻听力的疲劳感

(8)、停止播放

一般用的比较少，和Pause进行区别

- ACTION_STOP = 1L

⚠ **注意_跑题了（没关系）**

> - rewind本身是“倒带”，“倒回”。指的是将录音带、录像带或其他录像设备上的录像拉回到播放前的位置，以重新播放。rewinding以直觉来判断它即使rewind的动名词/现在分词的形式
> - 除了录像设备上的行为，rewind还有其他用法。例如，rewind也可以用来描述某件事情重新发生的意思，就像将时间倒转一样，将事情重新发生
> - 此外，rewind也可以用来描述某种情绪的回溯，比如当某人想起某个过去的经历时，他可能会说“我被rewind了”，表示他回想起了过去的经历

3、多媒体应用架构

刚刚提及过，那么大致知道，播放音频、视频，还有图片（图片也是可以播的哈）的多媒体应用通常是由两部分组成即**界面UI**和**播放器Player**两部分组成。而基于MediaSession的媒体应用框架就是解耦，多了MediaController和MediaSession（可以通过下面架构图一览无余）

3.1、播放器Player

用于接收数字媒体并将其呈现为视频/音频。可以是MediaPlayer、ExoPlayer或其他Player

- MediaPlayer ：提供准系统播放器的基本功能，支持最常见的音频/视频格式和数据源
- ExoPlayer ：一个提供低层级Android音频API的开放源代码库。ExoPlayer支持DASH和HLS流等高性能功能，这些功能在MediaPlayer中未提供

⚠ **注意_2023年IO大会中的原话**

> - 现在Exoplayer位于Media3中，在Media3中，你还可以找到熟悉API的最新版本，比如向后兼容的ExoPlayer和MediaSession，它们可自定义且易于使用，这些API的更新便于你更轻松地构建丰富的媒体体验
> - 更新后的MeidaSession API，可以更轻松地让播放状态和元数据保持最新状态，因此你可以实现与Android Auto、Wear OS Android TV等平台的高质量集成

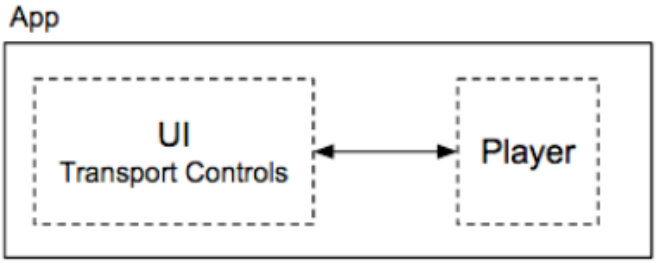
3.2、界面UI

用于显示、控制播放器状态界面

⚠ **注意**

> 2025-01-06修复：这张图原本的地址是：https://developer.android.google.cn/static/Image/training/cars/metadata_indicators.png。但现在打不开了，不过还好我有备份，后面有一些图片也是一样，不再赘述

3.3、传统应用架构



我们先来看看如何设计一款音乐播放App的架构，假如要求音频可以后台继续播放。传统的做法是这样的：

- 注册一个Service，用于异步获取音乐库数据、音乐控制等，在Service中我们可能还需要自定义一些状态值和回调接口用于流程控制
- 把Player放置在这个Service中，Service提供一个Binder或者广播（其他方式如接口、Messenger都可以）实现Activity和Service之间的通信，使得用户可以通过界面上的组件控制音乐的播放、暂停、拖动进度条等操作

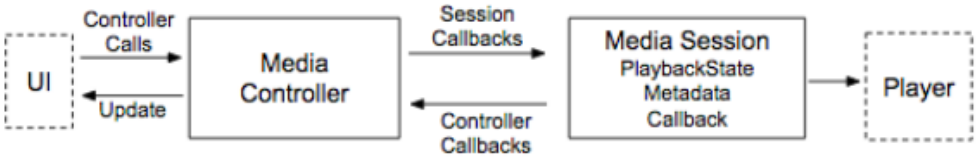
如果我们的音乐播放器还需要支持通知栏快捷控制音乐播放的功能，那么又得新增一套广播和相应的接口去响应通知栏按钮的事件

如果遇到锁屏时，要与Service之间进行通信就不得不用到AIDL接口/广播/ContentProvider来完成与其它应用之间的通信，这些通信手段既增加了应用开发者之间的沟通成本，也增加了应用之间的耦合度

如果还需要支持多端（电视、手表、耳机、车机等）控制同一个播放器，那么整个系统架构可能会变得非常复杂，我们要花费大量的时间和精力去设计、优化代码的结构。那么有什么方法可以节省这些工作，提高我们的效率，然后还可以优雅地实现上述这些功能呢？

它就是MediaSession框架

3.4、MediaSession架构



MediaSession框架专门用来解决媒体播放时界面和Service通讯的问题，意在规范上述这些功能的流程。MediaSession框架规范了音视频应用中**界面**与**播放器**之间的通信接口，**属于典型的 C/S 架构**，实现界面与播放器之间的完全解耦。框架定义了两个重要的类**媒体会话**和**媒体控制器**，它们为构建多媒体播放器应用提供了一个完善的结构。

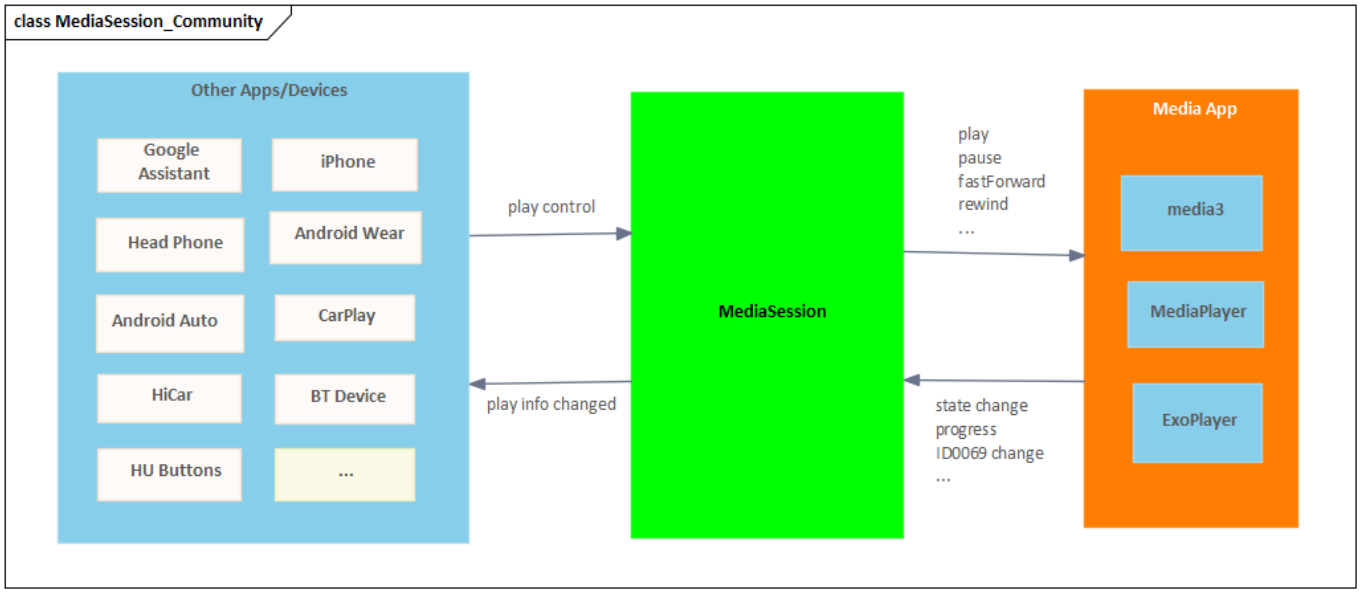
媒体会话和媒体控制器通过以下方式相互通信：使用与标准播放器操作（播放、暂停、停止等）相对应的预定义回调，以及用于定义应用独有的特殊行为的可扩展自定义调用

⚠ **注意**

> - support-v4中提供了MediaSession相应的兼容包，相关的类是以Compat结尾，API完全一致。若文中有提到的类似MediaBrowserCompat就得明白它和MediaBrowser是指同一个类
> - 我们公司计划2023/11要替换成androidx，建议大家还是升级适配androidx吧

优点

总结一下优点就是，使用一套接口，减少了很多流程的繁琐和service的通信等，实现**多个设备或者UI的统一调用**。。这个图得自己画



3.5、媒体会话MediaSession

媒体会话负责与播放器的所有通信。它会对应用的其他部分隐藏播放器的 API。只能从控制播放器的媒体会话中调用播放器。

会话会维护播放器状态（播放/暂停）的表示形式以及播放内容的相关信息。会话可以接收来自一个或多个媒体控制器的回调。这样一来，应用的界面以及搭载 Wear OS 和 Android Auto 的配套设备便可以控制您的播放器。响应回调的逻辑必须保持一致。无论由哪个客户端应用发起回调，对 MediaSession 回调的响应都应该相同。

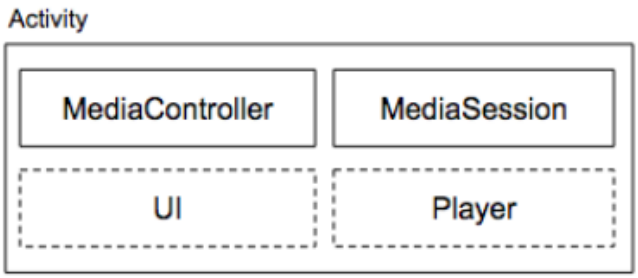
3.6、媒体控制器MediaController

媒体控制器会隔离您的界面。界面代码只与媒体控制器（而非播放器本身）通信。媒体控制器会将传输控制操作转换为对媒体会话的回调。每当会话状态发生变化时，它也会接收来自媒体会话的回调。这提供了一种自动更新关联界面的机制。一个媒体控制器一次只能连接到一个媒体会话。

当您使用媒体控制器和媒体会话时，您可以在运行时部署不同的接口和/或播放器。您可以根据运行应用的设备的功能单独更改该应用的外观和/或性能。

3.7、视频APP中MediaSession的不同

ui + player，不能在后台播放，必然是暂停或者退出的，那么他可以是单Activity的完成，呈现视频的屏幕是Activity的一部分，如下



视频也是媒体media，那么同样可以使用MediaSession，其它的还有图片也是一样。由于视频中不需要MediaBrowserService的后台服务，所以视频app中MediaSession和MediaController连接，看下图相信你就明白了，MediaController提供了两个连接MediaSession的方式

```
private final HashSet<Callback> mRegisteredCallbacks = new HashSet<>();

    Creates a media controller from a session.
    Params: session – The session to be controlled.
public MediaControllerCompat(Context context, @NonNull MediaSessionCompat session) {...}

    Creates a media controller from a session token which may have been obtained from another process.
    Params: sessionToken – The token of the session to be controlled.
    Throws: RemoteException – if the session is not accessible.
public MediaControllerCompat(Context context, @NonNull MediaSessionCompat.Token sessionToken)
        throws RemoteException {...}
```

视频应用

音频应用，可以是其它进程APP

⚠ **注意**

> 两个构造方法的在最新的源码里面移除了，但不影响构建媒体类应用视频，图片，音频使用MediaSession

下面开始介绍MediaSession框架的核心成员和使用流程

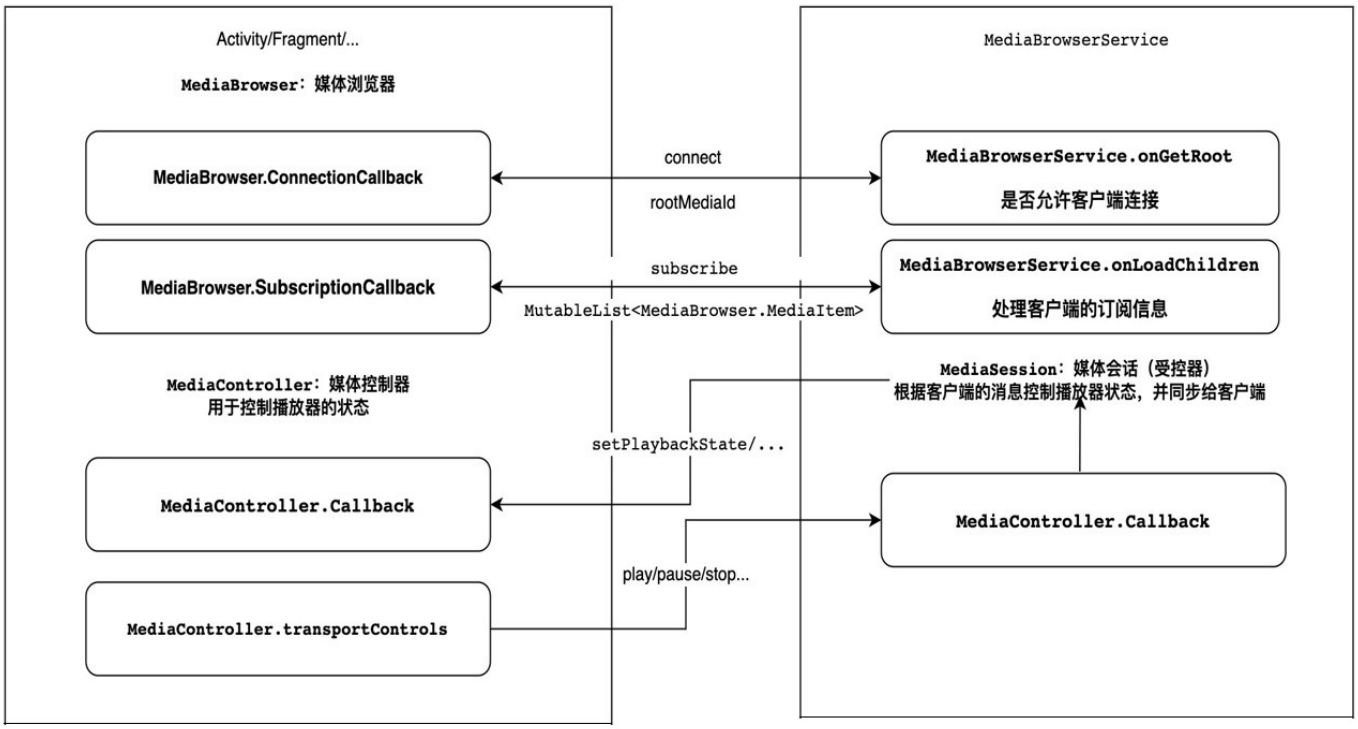
二、MediaSession接口

MediaSession框架相当于C/S架构

```
1、核心类
1.1 MediaBrowser
2023-12-28:新增《非主线程创建MediaBrowser》
⑴、MediaBrowser相关API列表（可选）
⑵、MediaBrowser.ConnectionCallback
⑶、MediaBrowser.ItemCallback
⑷、MediaBrowser.MediaItem
⑸、MediaBrowser.SubscriptionCallback

1.2、MediaBrowserService
⑴、MediaBrowserService相关API列表（可选）
⑵、MediaBrowserService.BrowserRoot
⑶、MediaBrowserService.Result

1.3、MediaSession
⑴、MediaSession相关API列表（可选）
⑵、MediaSession.Callback
⑶、MediaSession.QueueItem
⑷、MediaSession.Token

1.4、MediaController
⑴、MediaController相关组件API列表（可选）
⑵、MediaController.Callback
⑶、MediaController.PlaybackInfo
⑷、MediaController.TransportControls

2、其它相关API
2.1、播放器状态 - PlaybackState
⑴、PlaybackState相关组件API列表（可选）
⑵、PlaybackState.Builder
⑶、PlaybackState.CustomAction

2.2、元数据类 - MediaMetadata
```

1、核心类

MediaSession框架中有四个常用的成员类，MediaBrowser、MediaBrowserService、MediaSession、MediaController，它们是MediaSession整个流程控制的核心



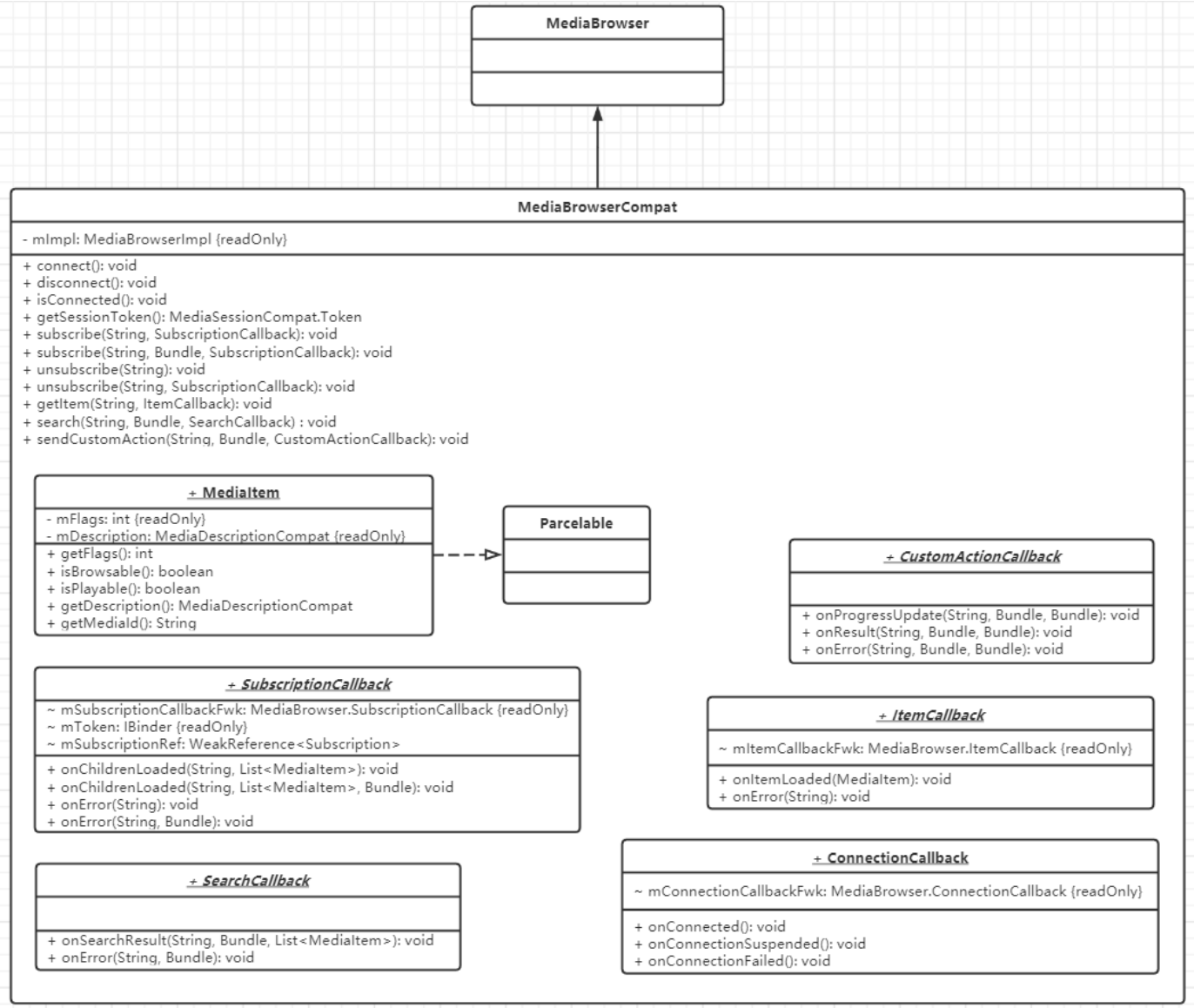
1.1、MediaBrowser

媒体浏览器、就是Client，用来**连接**MediaBrowserService**服务端**（*Server*）、调用它的onGetRoot()方法，在连接成功的结果回调后，获取token（配对令牌），并以此**获得MediaController媒体控制器**，在它的回调接口中可以获取和Service的连接状态以及获取在Service中异步获取的音乐库数据。然后可以有订阅并设置订阅信息回调功能来**订阅数据**

媒体浏览器一般创建于**客户端**（*Client APP*）（可以理解为各个终端负责控制音乐播放的界面）中，**不是线程安全的，所有调用都应在构造MediaBrowser的线程上进行**

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    //MediaService继承于MediaBrowserService
    val component = ComponentName(this, MediaService::class.java)
    //新建MediaBrowser,第一个参数是context
    //第二个参数是CompoentName,有多种构造方法,指向要连接的服务
```

```
    //第三个参数是连接结果的回调connectionCallback，第四个参数为Bundle
    mMediaBrowser = MediaBrowser(this, component, connectionCallback, null);
    mMediaBrowser.connect()
}
```



2023-12-28：新增《非主线程创建MediaBrowser》

非主线程创建MediaBrowser并connect的时候会报错。这是因为连接时底层代码会使用Handler，并且采用Handler handler = new Handler()的创建方式，如此使用必然会报错。解决办法：

```
Looper.prepare()
mMediaBrowser = MediaBrowser(BaseApplication.getInstance(),
        //绑定服务，这里绑定的是系统蓝牙音乐的核心服务，a2dp为蓝牙的一个协议
        new ComponentName("com.android.bluetooth",
        "com.android.bluetooth.a2dpsink.mbs.A2dpMediaBrowserService"),
        connectionCallback,//关联连接回调
        null)
mMediaBrowser.connect()
Looper.loop()
```

⑴、MediaBrowser相关API列表（可选）

除了上面刚用到的connect()以外，其它MediaBrowser的API如下所示：

方法名	说明
void connect()	连接到媒体浏览器服务
void disconnect()	断开与媒体浏览器服务的连接
Bundle getExtras()	获取介质服务的任何附加信息
void getItem(String mediaId, MediaBrowser.ItemCallback cb)	从连接的服务中检索特定的MediaItem
String getRoot()	获取根ID
ComponentName getServiceComponent()	获取媒体浏览器连接到的服务组件
MediaSession.Token getSessionToken()	获取与媒体浏览器关联的媒体会话Token
boolean isConnected()	返回浏览器是否连接到服务
void subscribe(String parentId,Bundle options, MediaBrowser.SubscriptionCallback callback)	使用特定于服务的参数进行查询，以获取有关指定 ID 中包含的媒体项的信息，并订阅以在更新更改时接收更新
void subscribe(String parentId, MediaBrowser.SubscriptionCallback callback)	询有关包含在指定 ID 中的媒体项的信息，并订阅以在更改时接收更新
void unsubscribe(String parentId)	取消订阅指定媒体 ID
void unsubscribe(String parentId, MediaBrowser.SubscriptionCallback callback)	通过回调取消订阅对指定媒体 ID

⑵、MediaBrowser.ConnectionCallback

接收与MediaBrowserService连接状态的回调，在创建MediaBrowser时传入，当MediaBrowser向service发起连接请求后，请求结果将在这个ConnectionCallback中返回，获取到的meidaId对应服务端在onGetRoot()函数中设置的mediaId，如果连接成功那么就可以做创建媒体控制器之类的操作了

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val component = ComponentName(this, MediaService::class.java)//MediaService继承于
MediaBrowserService
    mMediaBrowser = MediaBrowser(this, component, connectionCallback, null);
    mMediaBrowser.connect()
}
//连接结果的回调
private val connectionCallback = object : MediaBrowser.ConnectionCallback() {
    override fun onConnected() {
      //与MediaBrowserService连接成功。在调用MediaBrowser.connect()后才会有回调。
        super.onConnected()
    }

    override fun onConnectionFailed() {
      //与MediaBrowserService连接失败。比如onGetRoot返回null
        super.onConnectionFailed()
    }

    override fun onConnectionSuspended() {
      //与MediaBrowserService连接断开。进程死掉
        super.onConnectionSuspended()
```

```
    }
}
```

⑶、MediaBrowser.ItemCallback

用于接受调用`MediaBrowser.getItem()`后，MediaService返回的结果。媒体控制器MediaController负责向service中session发送例如播放暂停之类的指令的，这些指令的执行结果将在这个ItemCallback回调中返回，可重写的函数有很多，比如播放状态的改变，音乐信息的改变等

```
private val connectionCallback = object : MediaBrowser.ConnectionCallback() {
    override fun onConnected() {
        super.onConnected()
        // ...
        if(mMediaBrowser.isConnected) {
            val mediaId = mMediaBrowser.root
            mMediaBrowser.getItem(mediaId, itemCallback)
        }
    }
}

@RequiresApi(Build.VERSION_CODES.M)
private val itemCallback = object : MediaBrowser.ItemCallback(){
    override fun onItemLoaded(item: MediaBrowser.MediaItem?) {
      //返回Item时调用
        super.onItemLoaded(item)
    }

    override fun onError(mediaId: String) {
      //检索时出错，或者连接的服务不支持时回调
        super.onError(mediaId)
    }
}
```

⑷、MediaBrowser.MediaItem

包含有关单个媒体项的信息，用于浏览/搜索媒体。MediaItem依赖于服务端提供，因此框架本身无法保证它包含的值都是正确的

方法名	说明
int describeContents()	描述此可打包实例的封送处理表示中包含的特殊对象的种类
MediaDescription getDescription()	获取介质的说明。包含媒体的基础信息如：标题、封面等等
int getFlags()	获取项的标志。FLAG_BROWSABLE：表示Item具有自己的子项（是一个文件夹）。FLAG_PLAYABLE：表示Item可播放
String getMediaId()	返回此项的媒体 ID
boolean isBrowsable()	返回此项目是否可浏览
boolean isPlayable()	返回此项是否可播放

⚠ **重要**

> FLAG_BROWSABLE：表示Item具有自己的子项（是一个文件夹）FLAG_PLAYABLE：表示Item可播放。这对于MediaBrowserTree理解很有帮助

p.s. 注意区别：媒体信息对象 MediaMetadata、MediaSession.QueueItem、MediaBrowser.MediaItem、MediaDescription（后面提到的都在这里）

> - MediaSession.QueueItem比MediaMetadata《2、其它API》多了一个唯一的id
> - MediaBrowser.MediaItem跟MediaSession.QueueItem很相似，不同的是唯一的id，变成了flags

相互转换的代码：

```
//构建，传入MediaDescription 和id
MediaDescription description = new MediaDescription.Builder()
                .setMediaId(song.mediaId)
                .setTitle(song.title)
                .setSubtitle(song.subtitle)
                .setExtras(bundle)
                .build();
QueueItem queueItem = new QueueItem(description, song.queueId);
//MediaMetadata转化为QueueItem
QueueItem queueItem = new QueueItem(mediaMetadata.getDescription(), id);
//解析跟MediaMetadata一样，获取MediaDescription
MediaDescription description = queueItem.getDescription();
//获取标题
String title = description.getTitle().toString();

//.....................分割线.........................

//MediaMetadata转化为MediaItem,构造方法第一个都是MediaDescription,第二个是flags
MediaBrowser.MediaItem mediaItem = new MediaBrowser.MediaItem(metadata.getDescription(),
MediaBrowser.MediaItem.FLAG_PLAYABLE);
//解析一样用MediaDescription
MediaDescription description = queueItem.getDescription();
//获取标题
String title = description.getTitle().toString();
```

⑸、MediaBrowser.SubscriptionCallback

连接成功后，首先客户端调用subscribe()订阅MediaBrowserService服务，同样还需要注册订阅回调，订阅成功的话服务端可以返回一个音乐信息的序列，可以在客户端展示获取的音乐列表数据MediaBrowser.MediaItem

下面这就是订阅MediaBrowserService中MediaBrowser.MediaItem列表变化的回调

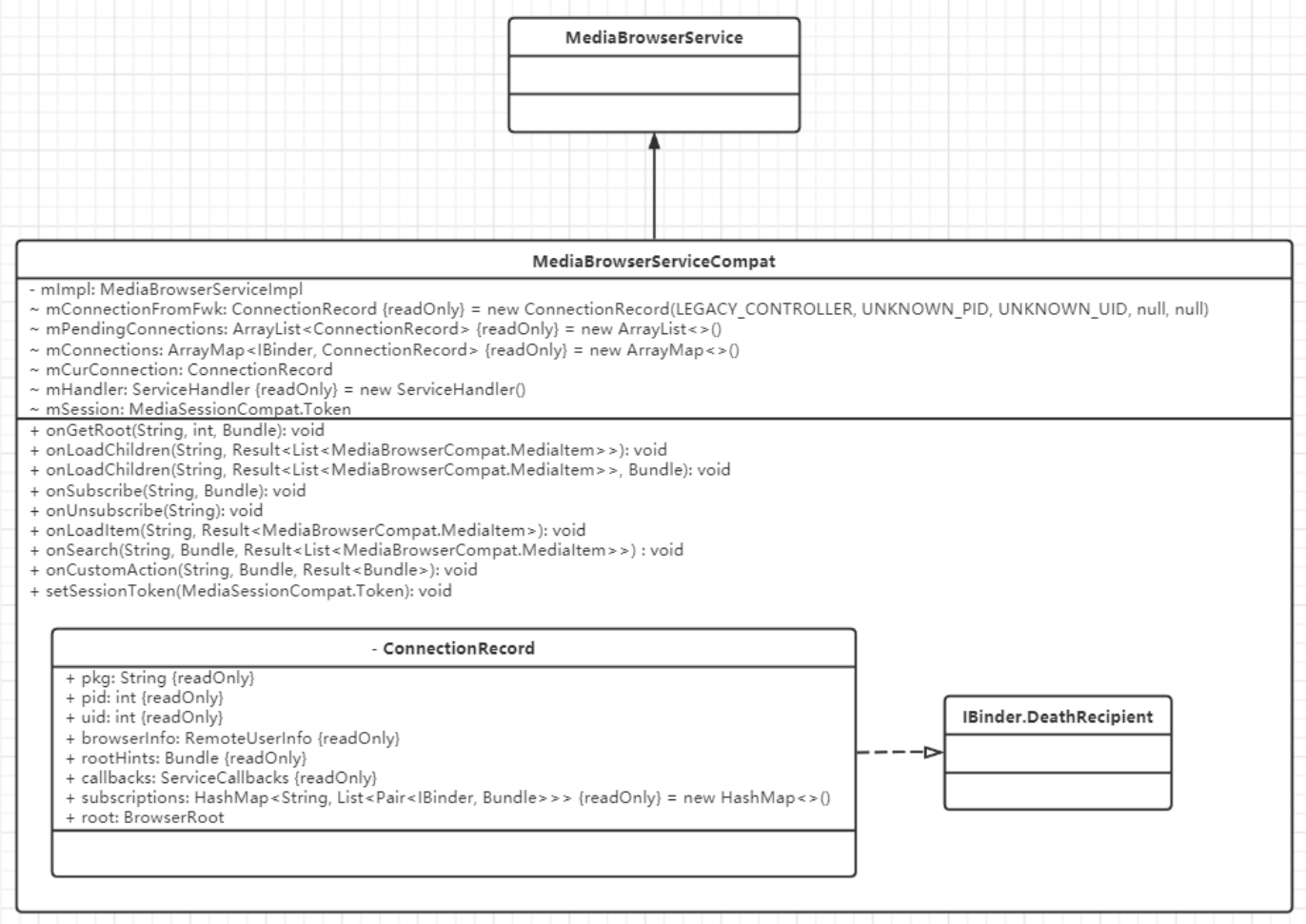
```
private val connectionCallback = object : MediaBrowser.ConnectionCallback() {
    override fun onConnected() {
        super.onConnected()
        // ...
        if(mMediaBrowser.isConnected) {
            val mediaId = mMediaBrowser.root
            // 重复订阅会报错，所以先解除订阅  这样可以进行异步数据回调
            mMediaBrowser.unsubscribe(mediaId)
            //第一个参数是String类型的parentId(标识)
            //第二个参数为订阅的回调MediaBrowser.SubscriptionCallback
            // 服务端会调用onLoadChildren
            mMediaBrowser.subscribe(mediaId, subscribeCallback)
        }
    }
}

private val subscribeCallback = object : MediaBrowser.SubscriptionCallback(){
    override fun onChildrenLoaded(parentId: String,children: MutableList<MediaBrowser.MediaItem>)
```

```
{
      //在客户端调用mMediaBrowser.subscribe()，服务端MediaBrowserService会调用onLoadChildren()，服务端会去browse()浏览数据。
      //浏览完后，加载或更新子项列表时 服务端调用result.sendResult()方法（详见二.MediaSession接口1.2MediaBrowserService.(二)、MediaBrowserService.Result<T>中），会回调到客户端onChildrenLoaded这里。下同
        super.onChildrenLoaded(parentId, children)
    }

    override fun onChildrenLoaded(parentId: String,children:
MutableList<MediaBrowser.MediaItem>,options: Bundle) {
        super.onChildrenLoaded(parentId, children, options)
    }

    override fun onError(parentId: String) {
      //当 ID 不存在或订阅时出现其他错误时回调。下同
        super.onError(parentId)
    }

    override fun onError(parentId: String, options: Bundle) {
        super.onError(parentId, options)
    }
}
```

⚠ **重要**

> - onChildrenLoaded() 是MediaBrowser客户端的方法
> - onLoadChildren() 是MediaBrowserService服务端的方法
> - 不能重复订阅相同parentId的，会报错，建议订阅时都先做解除订阅操作
> - 在 mMediaBrowser.subscribe(...)方法中，可以添加第三个Bundle参数，此时回调到同存在Bundle参数的onChildrenLoaded(...)方法中，注意别弄错了回调方法

1.2、MediaBrowserService



媒体浏览器服务MediaBrowserService继承自Service，MediaBrowserService属于服务端。提供onGetRoot（接受客户端**媒体浏览器MediaBrowser**的**连接请求**，通过返回值决定是否允许该客户端连接服务）和onLoadChildren（**媒体浏览器MediaBrowser**向Service发送数据订阅时调用，*一般在这执行异步获取数据的操作，最后将数据发送至媒体浏览器的回调接口`onChildrenLoaded()`中*）这两个抽象方法

> - 一般在onCreate()中用setSessionToken(...)来设置token。在重写的onGetRoot(...)中判断是否允许连接，在onLoadChildren(...)中处理**订阅信息**
> - 同时MediaBrowserService还作为承载媒体播放器（如MediaPlayer、ExoPlayer等）和MediaSession的容器。也就是可以在这里创建Player

客户端调用MediaBrowser.subscribe时会触发onLoadChildren方法

```
const val FOLDERS_ID = "__FOLDERS__"
const val ARTISTS_ID = "__ARTISTS__"
const val ALBUMS_ID = "__ALBUMS__"
const val GENRES_ID = "__GENRES__"
const val ROOT_ID = "__ROOT__"

class MediaService : MediaBrowserService() {
    // 获取供特定客户端浏览的根信息，控制是否允许客户端连接，并返回root media id给客户端
    override fun onGetRoot(clientPackageName: String,clientUid: Int,rootHints: Bundle?):
BrowserRoot? {
        //由MediaBrowser.connect触发，可以通过返回null拒绝客户端的连接
        return BrowserRoot(ROOT_ID, null)
    }
    // 处理客户端的订阅信息，由MediaBrowser.subscribe触发
    override fun onLoadChildren(parentId: String,result:
Result<MutableList<MediaBrowser.MediaItem>>) {
      //获取有关媒体项的子项的信息。由MediaBrowser.subscribe触发
      //一般在这执行**异步获取数据**的操作，最后将数据通过sendResult()发送至
```

```
MediaBrowser.SubscriptionCallback(如上面↑↑↑<1.1、MediaBrowser#⑷、
MediaBrowser.SubscriptionCallback>)的回调接口中
        val mediaItems = arrayList<MediaBrowser.MediaItem>()
        when (parentId) {
            ROOT_ID -> {
                // 查询本地媒体库
                // ...
                // 将此消息与当前线程分离，并允许稍后进行sendResult调用
                result.detach()
                //发送数据，他会回调客户端的onChildrenLoaded()方法
                result.sendResult()
            }
            FOLDERS_ID -> {

            }
            ALBUMS_ID -> {

            }
            ARTISTS_ID -> {

            }
            GENRES_ID -> {

            }
            MEDIA_ID_ROOT -> {

            }
            PARENT_ID_1 -> {
                //模拟数据
                val metadata =
MediaMeadata.Builder().putString(MediaMetadata.METADATA_KEY_MEDIA_ID, "101")
                .putString(MediaMetadata.METADATA_KEY_TITLE, "一首歌").build()
                mediaItems.add(MediaBrowser.MediaItem(metadata.getDescription(),
                                        MediaBrowser.MediaItem.FLAG_PLAYABLE))
            }
            else -> {

            }
        }
    }
    //获取有关特定媒体项的信息。由MediaBrowser.getItem触发。
    override fun onLoadItem(itemId: String?, result: Result<MediaBrowser.MediaItem>?) {
        super.onLoadItem(itemId, result)
        Log.e("TAG", "onLoadItem: $itemId")
        // 根据itemId，返回对用MediaItem
        result?.detach()
        result?.sendResult(null)
    }
}
```

然后还需要在manifest中注册这个Service

```
<service
    android:name=".MediaService"
    android:label="@string/service_name">
    <intent-filter>
        <action android:name="android.media.browse.MediaBrowserService" />
    </intent-filter>
</service>
```

2024-04-18：新增《MediaBrowserService类中方法在APP中ContentProvider和Application的执行的先后顺序》

MediaBrowserService其实是一个Service，除了上面↑↑↑的onGetRoot()、onLoadChildren()和下面↓↓↓即将分析的方法以外，它还有自己的onCreate()等生命周期的方法，要分析**MediaBrowserService类中方法在APP中ContentProvider和Application的执行的先后顺序**，我们肯定是要了解一点Android启动的过程的，大致是这样子

> - 启动过程过度到这里说：bootloader=RAM，init进程（初始化），zygote进程，Application初始化
> - 这里有一个Binder机制，通过反射拉起我们的AMS，拉起Entertainment(娱乐)应用

然后我索性直接抓一个我们现有的项目的Log直接分析：

```
01-01 00:00:21.847   952   997 I SystemConfig: Reading permissions from
/system/etc/permissions/privapp_permissions_mediacoreservice.xml
12-13 04:08:22.004  2343  2343 I MediaService: MediaContentProvider__onCreate
12-13 04:08:23.844  2957  2957 I MediaService: MediaContentProvider__onCreate
12-13 04:08:23.869  2957  2957 D MediaService: MediaManager__loadMediaXmlConfig
12-13 04:08:23.874  2957  2957 D MediaService: ConfigParser__loadXmlConfigByPath : configPath =
/vendor/etc/media/media_config.xml
12-13 04:08:23.875  2957  2957 D MediaService: ConfigParser__loadXmlConfigByPath : Whether config
file exists = true
12-13 04:08:23.875  2957  2957 D MediaService: ConfigParser__loadXmlConfigByPath : Xml Config
does exist, start to init XmlPullParser.
12-13 04:08:23.876  2957  2957 D MediaService: ConfigParser__loadXmlConfigByPath : Xml Config
does exist, start to load config values.
12-13 04:08:24.687  2957  2957 D MediaService: MyMediaBrowserService__onCreate_LocalPlayer3
12-13 04:08:24.867  2957  2957 D MediaService: ServiceSessionBase__getRoot(LocalPlayer3) :
clientPackageName  = com.sunst0069.mediamodeservice, client uid = 1001000, rootHints = null
```

可以看到2957的进程被系统AMS拉起来后，ContentProvider先执行，接着到MCS的's，Application再到，（然后在MCS的静态方法）、MediaBrowserService的onCreate()，再到getRoot()，所以顺序应该是，下面这样

```
点火
init进程
zygote进程
AMS进程

ContentProvider
App's Application
App's 'Application ->onCreate()
MCS's MediaManager
MCS's Static Method
MediaBrowserService's ->onCreate()
MediaBrowserService's ->getRoot()
```

⚠ **注意**

> UsbDeviceManager or MtpDeviceManager的静态代码快先于MediaBrowserService's ->onCreate()执行，但是如果静态代码块的方法，比较耗时，那么也可能静态代码块没有执行完，onCreate()就已经执行了

2025-01-03再次补充：为什么本神要分析它呢？试想一下如果你的音乐，它有收藏功能，有最近播放功能，这些数据要么是存在SP中，要么是存在Sqlite中，如果你在下一次要去恢复播放它的时候，恢复它的播放列表，恢复CarPlay，恢复CarLife,AndroidAuto等等，那么你可能会想，我要在什么地方去拿我保存下来的数据呢，肯定不是你在扫描数据的时候吧，所以你应该知道，那必定在你的服务（APK）启动后的ContentProvider或者是Application中异步拿到你上次保存的数据

当然绝大部分人，接着会这样处理，在数据扫描的时候，检查是否拿到了数据，如果拿到了数据，就进一步的更新（如收藏，最近播放列表）数据，如果没有拿到，就等待Application中异步去拿数据库的数据的回调结果，如果回调回来了，就更新（如收藏，最近播放列表）数据。这种逻辑其实就是if else看起来没问题，实际上还是有问题，

你就思考一点就能明白我说的：为什么我在数据扫描或者其它操作的时候，我还拿不到数据？为什么，我期待的行为是什么，多一个else的代价是什么，就千万不要被代码本身困住了。包含我们公司的代码也是如此，我就漏一小点，我们公司的代码：

```
private void loadLastModeData(MediaBrowseTree tree) {
    if (LastModeManager.getInstance().isLoadLastModeDBSuccess()) {
        tree.openLastPlayingNode();
    } else {
        LastModeManager.getInstance().registerLoadLastModeDBListener(tree::openLastPlayingNode);
    }
}
```

⑴、MediaBrowserService相关API列表（可选）

MediaBrowserService除了上面onGetRoot、onLoadChildren、onLoadItem方法，其它相关组件API如下所示：

方法名	说明
final Bundle getBrowserRootHints()	获取从当前连接MediaBrowser的发送的根提示
final MediaSessionManager.RemoteUserInfo getCurrentBrowserInfo()	获取发送当前请求的浏览器信息
MediaSession.Token getSessionToken()	获取会话令牌，如果尚未创建会话令牌或已销毁会话令牌，则获取 null
void notifyChildrenChanged(String parentId)	通知所有连接的媒体浏览器指定父 ID 的子级已经更改
void notifyChildrenChanged(String parentId, Bundle options)	通知所有连接的媒体浏览器指定父 ID 的子级已经更改
void onLoadChildren(String parentId, Result> result,Bundle options)	获取有关媒体项的子项的信息。由MediaBrowser.subscribe触发
void setSessionToken(MediaSession.Token token)	设置媒体会话
void onSearch(String query, Bundle extras, Result> result)	客户端调用搜索相关

⚠ **注意**
有两个方法比较类似：

> - mMediaBrowser.getItem(rootMediaId,itemCallback)，会触发MediaBrowserService.onLoadItem方法来获取**根mediaId的item列表**
> - 订阅之前需要先unsubscribe

```
MediaBrowser.unsubscribe(rootMediaId)
// media item的改变，会触发服务端MediaBrowserService.onLoadChildren方法
mMediaBrowser.subscribe(rootMediaId, subscribeCallback)
```

> - **服务端重写的onLoadChildren(…)用作订阅不同parentId返回不同的媒体数据。此外进行订阅后，服务端可以通过notifyChildrenChanged(String parentId)发送消息来进行回调自己的onLocadChildren()**
> - 服务端MediaBrowserService可以直接使用notifyChildrenChanged(String)，内部会触发MediaBrowserService自己的onLocadChildren()方法，并回调数据。如果客户端订阅了对应parentId,那么在MediaBrowser.SubscriptionCallback中就能收到媒体数据

```
notifyChildrenChanged("parentId_1");
```

p.s. 重要：所以这里得出一个结论：onLocadChildren()有两种方式可以触发

> - 1、当客户端调用MediaBrowser.subscribe(rootMediaId, subscribeCallback)时会触发`onLoadChildren()`方法。一般来说是客户端点击子项目就会触发subscribe()
> - 2、服务端MediaBrowserService自己调用notifyChildrenChanged(String)也会触发`onLoadChildren()`。如果onLocadChildren在浏览数据，那么这种情况会接着一级一级的browse数据

⑵、MediaBrowserService.BrowserRoot

返回包含浏览器服务首次连接时需要发送给客户端的信息。构造函数

```
MediaBrowserService.BrowserRoot(String rootId, Bundle extras)
```

它有两个方法API：

- getExtras()：获取有关浏览器服务的附加信息
- getRootId()：获取用于浏览的根 ID

⑶、MediaBrowserService.Result

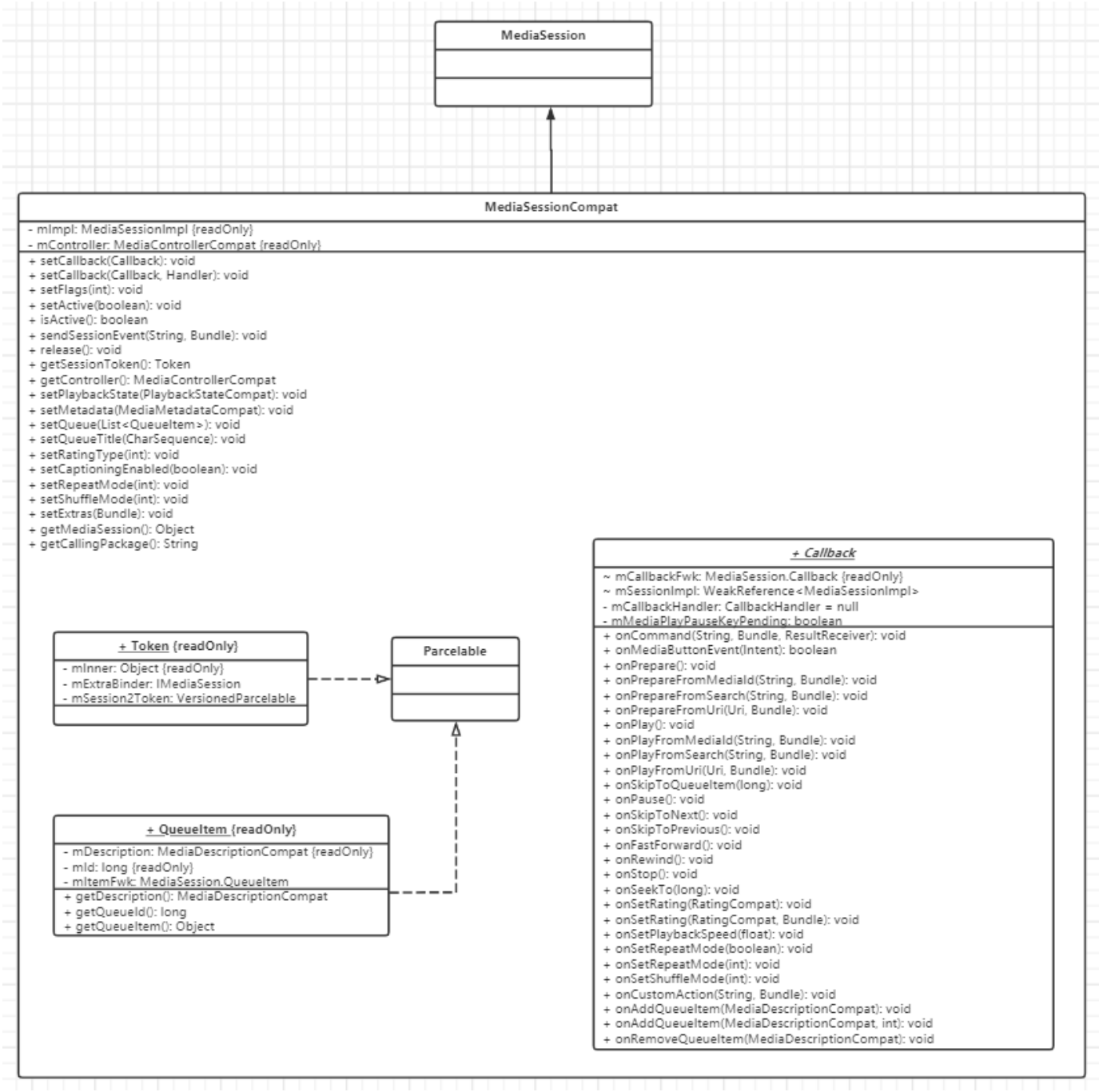
包含浏览器服务返回给客户端的结果集。通过调用`sendResult()`将结果返回给调用方，但是在此之前需要调用detach()

- MediaBrowserService.Result API 列表

方法名	说明
void detach()	0将此消息与当前线程分离，并允许稍后进行调用sendResult(T)
void sendResult(T result)	将结果发送回调用方

1.3、MediaSession

媒体会话，即**受控端**也可以理解为服务端，通过设置MediaSession.Callback回调，来接收**媒体控制器MediaController**发送的指令，当收到指令时会触发Callback中各个指令对应的回调方法（回调方法中会执行播放器相应的操作，如播放、暂停等）



Session一般在Service.onCreate方法中创建，最后需要调用上面↑↑↑*<1.2、MediaBrowserService>*中**setSessionToken()方法**设置用于和**控制器**配对的令牌。当媒体信息或状态改变后，可以使用形如`mediaSession.setMetadata(mediaMetadata)`来通知客户端

```
const val FOLDERS_ID = "__FOLDERS__"
const val ARTISTS_ID = "__ARTISTS__"
const val ALBUMS_ID = "__ALBUMS__"
const val GENRES_ID = "__GENRES__"
const val ROOT_ID = "__ROOT__"

class MediaService : MediaBrowserService() {
    private lateinit var mediaSession: MediaSession;

    override fun onCreate() {
        super.onCreate()
        //初始化，第一个参数为context，第二个参数为String类型tag，这里可以设置为类名
        mediaSession = MediaSession(this, "TAG")
        //设置callback，这里的callback就是客户端对服务指令到达处
        mediaSession.setCallback(callback)
```

```
        //设置token
        sessionToken = mediaSession.sessionToken
    }

    // 与MediaController.transportControls中的大部分方法都是一一对应的
    // 在该方法中实现对 播放器 的控制·
    private val callback = object : MediaSession.Callback() {
        override fun onPlay() {
            super.onPlay()
            //客户端mMediaController.getTransportControls().play()就会调用到这里·以下类推
            // 处理 播放器 的播放逻辑。
            // 车载应用·别忘了处理音频焦点requestAudoFocus()·因为有VPA·电话·抢占音频焦点
        }

        override fun onPause() {
            //暂停
            super.onPause()
        }

    }
    // 控制是否允许客户端连接·并返回root media id给客户端
    // 第一个参数为客户端的packageName·第二个参数为Uid
    // 第三个参数是从客户端传递过来的Bundle
    override fun onGetRoot(clientPackageName: String,clientUid: Int,rootHints: Bundle?):
BrowserRoot? {
        Log.e("TAG", "onGetRoot: $rootHints")
        // 通过以上参数来进行判断·若同意连接·则返回BrowserRoot对象·否则返回null

        // 构造BrowserRoot的第一个参数为rootId(自定义)·第二个参数为Bundle;
        return BrowserRoot(ROOT_ID, null)
    }
    // 处理客户端的订阅信息
    override fun onLoadChildren(parentId: String,result:
Result<MutableList<MediaBrowser.MediaItem>>) {
      // 由MediaBrowser.subscribe触发
        Log.e("TAG", "onLoadChildren: $parentId")
        result.detach()
        when (parentId) {
            ROOT_ID -> {
                result.sendResult(null)
            }
            FOLDERS_ID -> {

            }
            ALBUMS_ID -> {

            }
            ARTISTS_ID -> {

            }
            GENRES_ID -> {

            }
            else -> {

            }
        }
    }

    override fun onLoadItem(itemId: String?, result: Result<MediaBrowser.MediaItem>?) {
        super.onLoadItem(itemId, result)
        Log.e("TAG", "onLoadItem: $itemId")
```

```
    }
}
```

⑴、MediaSession相关API列表（可选）

部分方法如`setExtras()`与MediaController.Callback（详见下面↓↓↓<*1.4、MediaController#⑵、MediaController.Callback*>）中，如`onExtrasChanged()`，`onAudioInfoChanged()`一一对应

方法名	说明
MediaController getController()	获取此会话的控制器
MediaSessionManager.RemoteUserInfo getCurrentControllerInfo()	获取发送当前请求的控制器信息
MediaSession.Token getSessionToken()	获取此会话令牌对象
boolean isActive()	获取此会话的当前活动状态
void release()	当应用完成播放时，必须调用此项
void sendSessionEvent (String event, Bundle extras)	将专有事件发送给监听此会话的所有MediaController。会触发MediaController.Callback.onSessionEvent
void setActive(boolean active)	设置此会话当前是否处于活动状态并准备好接收命令
void setCallback (MediaSession.Callback callback)	设置回调以接收媒体会话的更新
void setCallback (MediaSession.Callback callback,Handler handler)	设置回调以接收媒体会话的更新
void setExtras(Bundle extras)	设置一些可与MediaSession关联的附加功能
void setFlags(int flags)	为会话设置标志
void setMediaButtonBroadcastReceiver(ComponentName broadcastReceiver)	设置应接收媒体按钮的清单声明类的组件名称
void setMediaButtonReceiver(PendingIntent mbr)	此方法在 API 级别 31 中已弃用。改用setMediaButtonBroadcastReceiver（android.content.ComponentName）
void setMetadata(MediaMetadata metadata)	更新当前MediaMetadata
void setPlaybackState(PlaybackState state)	更新当前播放状态
void setPlaybackToLocal(AudioAttributes attributes)	设置此会话音频的属性
void setPlaybackToRemote(VolumeProvider volumeProvider)	将此会话配置为使用远程音量处理
void setQueue(List queue)	更新播放队列中的项目列表
void setQueueTitle(CharSequence title)	设置播放队列的标题
void setRatingType(int type)	设置此会话使用的评级样式
void setSessionActivity(PendingIntent pi)	设置启动此会话的Activity的Intent

⑵、MediaSession.Callback

接收来自**控制器MediaController**和**系统的媒体按钮**（像方向盘上面的按钮）、传输控件和命令，如【上一曲】、【下一曲】。也是与下面↓↓↓<*1.4、MediaController#⑷、MediaController.TransportControls*>中方法一一对应

```
override fun onCreate() {
    super.onCreate()
    mediaSession = MediaSession(this, "TAG")
    mediaSession.setCallback(callback)
    sessionToken = mediaSession.sessionToken
}

// 与MediaController.transportControls中的方法是一一对应的。
// 在该方法中实现对 播放器 的控制·
private val callback = object : MediaSession.Callback() {
    override fun onPlay() {
        super.onPlay()
        // 处理 播放器 的播放逻辑。
        // 车载应用的话，别忘了处理音频焦点
        // ...
        if (!mediaSession.isActive) {
            mediaSession.isActive = true
        }
        // 更新播放状态.
        val state = PlaybackState.Builder()
            .setState(
                PlaybackState.STATE_PLAYING,1,1f
            )
            .build()
        // 此时MediaController.Callback.onPlaybackStateChanged会回调
        mediaSession.setPlaybackState(state)
    }

    override fun onPause() {
      //处理暂停播放的请求
        super.onPause()
    }

    override fun onStop() {
      //处理停止播放的请求
        super.onStop()
    }
    override fun onSkipToNext(){
        super.onSkipToNext();
        //下一首
        .....
        //通知媒体信息改变
        mediaSession.setMetadata(mediaMetadata)
    }
}
```

❶、MediaSession.Callback相关组件API列表（可选）

除了上面提到的`onPlay()`，`onPause()`，`onStop()`以外，其它**MediaController.transportControls**回调到**MediaSession.Callback**的API如下所示：

方法名	说明
void onCommand(String command,Bundle args,ResultReceiver cb)	当控制器已向此会话发送命令时调用
void onCustomAction(String action, Bundle extras)	当要执行MediaController.PlaybackState.CustomAction时调用。对应客户端mMediaController.getTransportControls().sendCustomAction(...)

void onFastForward()	处理快进请求
boolean onMediaButtonEvent(Intent mediaButtonIntent)	当按下媒体按钮并且此会话具有最高优先级或控制器向会话发送媒体按钮事件时调用
void onPlayFromMediaId(String mediaId, Bundle extras)	处理播放应用提供的特定mediaId的播放请求
void onPlayFromSearch(String query, Bundle extras)	处理从搜索查询开始播放的请求
void onPlayFromUri(Uri uri, Bundle extras)	处理播放由URI表示的特定媒体项的请求
void onPrepare()	处理准备播放的请求
void onPrepareFromMediaId(String mediaId, Bundle extras)	处理应用提供的特定mediaId的准备播放请求
void onPrepareFromSearch(String query, Bundle extras)	处理准备从搜索查询播放的请求
void onPrepareFromUri(Uri uri, Bundle extras)	处理由URI表示的特定媒体项的准备请求
void onRewind()	处理倒带请求
void onSeekTo(long pos)	处理跳转到特定位置的请求
void onSetPlaybackSpeed(float speed)	处理修改播放速度的请求
void onSetRating(Rating rating)	处理设定评级的请求
void onSetRating(RatingCompat rating, Bundle extras)	处理设定评级的请求。可以用extras接受如mediaId等参数
void onSkipToNext()	处理要跳到下一个媒体项的请求
void onSkipToPrevious()	处理要跳到上一个媒体项的请求
void onSkipToQueueItem(long id)	处理跳转到播放队列中具有给定 ID 的项目的请求

⑶、MediaSession.QueueItem

作为播放队列一部分的单个项目。相比MediaMetadata多了一个ID属性

❶、MediaSession.QueueItem相关组件API列表（可选）

方法名	说明
MediaDescription getDescription()	返回介质的说明。包含媒体的基础信息如：标题、封面等等
long getQueueId()	获取此项目的队列 ID

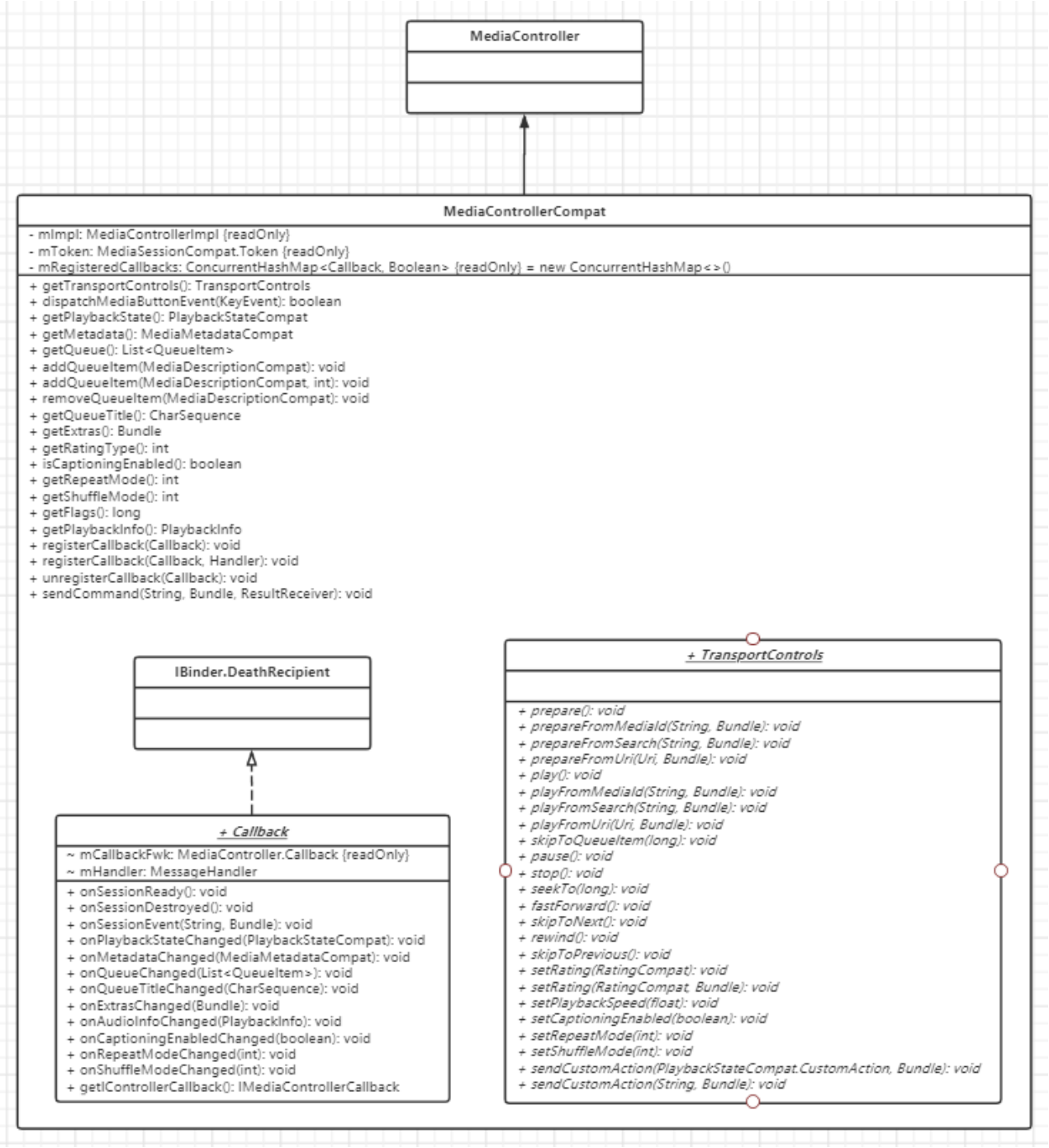
⑷、MediaSession.Token

表示正在进行的会话。可以通过会话所有者传递给客户端，以允许客户端与服务端之间建立通信

1.4、MediaController

媒体控制器，在客户端中开发者不仅可以使用控制器向Service中的受控端发送指令（播放、暂停），还可以通过设置MediaController.Callback回调方法接收**MediaSession受控端**的状态，从而根据相应的状态刷新界面UI。MediaController的创建需要受控端的配对令牌，因此需在**MediaBrowser成功连接服务的回调执行创建的操作，媒体控制器是线程安全的**

> 如用mMediaController.getTransportControls().skipToNext()来发送播放下一曲的控制指令；也可以用mMediaController.getMetadata()等方法来主动获取媒体信息



MediaController还有一个关联的权限`android.permission.MEDIA_CONTENT_CONTROL`（不是必须加的权限）必须是系统级应用才可以获取，幸运的是车载应用一般都是系统级应用。（但是我们公司项目中在Android14 U34上面谷歌对于OEM系统的权限收紧后，我们的MCS服务不能放在/system/priv-app/下面，后来统一修改到vendor后就不能工作了，这种情况就需要加上相应的权限）。**MediaController必须在MediaBrowser连接成功后才可以创建**

```
private val connectionCallback = object : MediaBrowser.ConnectionCallback() {

    override fun onConnected() {
        super.onConnected()
        // ...
        if(mMediaBrowser.isConnected) {
            //获取配对令牌
            val sessionToken = mMediaBrowser.sessionToken
            //通过token，获取MediaController,第一个参数是context，第二个参数为token
```

```
            mMediaController = MediaController(applicationContext,sessionToken)
        }
    }
}
```

⑴、MediaController相关组件API列表（可选）

方法名	说明
void adjustVolume (int direction, int flags)	调整此会话正在播放的输出的音量
boolean dispatchMediaButtonEvent (KeyEvent keyEvent)	将指定的媒体按钮事件发送到会话
Bundle getExtras()	获取此会话的附加内容
long getFlags()	获取此会话的标志
MediaMetadata getMetadata()	获取此会话的当前Metadata
String getPackageName()	获取会话所有者的程序包名称
MediaController.PlaybackInfo getPlaybackInfo()	获取此会话的当前播放信息
PlaybackState getPlaybackState()	获取此会话的当前播放状态
List getQueue()	获取此会话的当前播放队列（如果已设置）
CharSequence getQueueTitle()	获取此会话的队列标题
int getRatingType()	获取会话支持的评级类型
PendingIntent getSessionActivity()	获取启动与此会话关联的 UI 的意图（如果存在）
Bundle getSessionInfo()	获取创建会话时设置的其他会话信息
MediaSession.Token getSessionToken()	获取连接到的会话的令牌
String getTag()	获取会话的标记以进行调试
MediaController.TransportControls getTransportControls()	获取TransportControls实例以将控制操作发送到关联的会话
void registerCallback (MediaController.Callback callback, Handler handler)	注册回调以从会话接收更新
void registerCallback (MediaController.Callback callback)	注册回调以从会话接收更新
void sendCommand (String command, Bundle args, ResultReceiver cb)	向会话发送通用命令
void setVolumeTo (int value, int flags)	设置此会话正在播放的输出的音量
void unregisterCallback (MediaController.Callback callback)	注销指定的回调

⑵、MediaController.Callback

用于从MediaSession接收回调，它也是与上面↑↑↑<*1.3、MediaSession#⑴、MediaSession相关组件API列表*>的接口一一对应

```
private val connectionCallback = object : MediaBrowser.ConnectionCallback() {
    override fun onConnected() {
        super.onConnected()
        // ...
        if(mMediaBrowser.isConnected) {
            val sessionToken = mMediaBrowser.sessionToken
```

```
            mMediaController = MediaController(applicationContext,sessionToken)
            //mediaController注册回调，callback就是媒体信息改变后，服务给客户端的回调
            mMediaController.registerCallback(controllerCallback)
        }
    }
}
//服务对客户端的信息回调
private val controllerCallback = object : MediaController.Callback() {
    //音频信息，音量
    override fun onAudioInfoChanged(info: MediaController.PlaybackInfo?) {
      //当前音频信息发生改变。
        super.onAudioInfoChanged(info)
        val currentVolume = info?.currentVolume
        // 显示在 UI 上
    }

    override fun onExtrasChanged(extras: Bundle?) {
      //当前附加内容发生改变。
        super.onExtrasChanged(extras)
        val artUri = metadata?.getString(MediaMetadata.METADATA_KEY_ALBUM_ART_URI)
        // 显示 UI 上
    }
    // ...
}
```

这里我们取得了mMediaController，并且注册了一个回调，用于知晓服务端通知的媒体信息变更。在后面的代码中，就可以用mMediaController**为所欲**了（内容太多了，皮一下(^_^)，没打错）

```
//在需要的地方使用以下代码
//控制媒体服务的一些方法，播放、暂停、上下首、跳转某个时间点...
// 更多参考下面↓↓↓<1.4、MediaController#⑷、MediaController.TransportControls>的内容
mMediaController.transportControls.play()
mMediaController.transportControls.pause()
mMediaController.transportControls.skipToPrevious()
mMediaController.transportControls.skipToNext()
mMediaController.transportControls.seekTo(...)
....
//主动获取媒体信息的一些操作，获取媒体信息，播放状态...
// 下面↓↓↓<2、其它API#⑷、PlaybackState>的内容
val metadata = mMediaController.metadata
val playbackState = mMediaController.playbackState
....
```

❶、MediaController.Callback相关组件API列表（可选）

除了上面onAudioInfoChanged()、onExtrasChanged两个方法，其它相关API列表

方法名	说明
void onMetadataChanged (MediaMetadata metadata)	当前Metadata发生改变。服务端运行mediaSession.setMetadata(mediaMetadata)就会到达此处，以下类推
void onPlaybackStateChanged(PlaybackState state)	当前播放状态发生改变。客户端通过该回调来显示界面上音视频的播放状态
void onQueueChanged (List queue)	当前队列中项目发生改变
void onQueueTitleChanged	当前队列标题发生改变

(CharSequence title)

void onSessionDestroyed()	会话销毁
void onSessionEvent (String event, Bundle extras)	MediaSession所有者发送的自定义事件

⚠ **注意：这里解释一下**

> - onMetadataChange(MediaMetadataCompat mediaMetadata)比如收藏状态，歌曲的歌名改变，播放切歌，（switch play list）等都会触发该方法
> - onSessionEvent(String event, Bundle extras)切源，比如正在播放CarPlay的歌曲，然后插上了U盘，切放播放源（source list）这种场景会触发该方法
> - 项目中有时候会将这个方法起名为：onActiveSourceChange()//表示活动的源变化了；同样的看到还有SessionChange()类似这样的方法也是指会话发生了变化
> - 在谷歌MediaSessionManager中有：addOnActiveSessionsChangedListener(listener)

❷、其它逻辑处理

另外比如要处理一些逻辑上的问题

> 当两个源切换的时候，肯定会触发SessionChange相关的回调。如果先前的源存在，那么就会将先前的源的Controller设置为false，再将当前源的Controller设置为true，因为播放的时候，每个源都会有一个Controller

具体的案例可以本公司的：ExternalMediaServer中MediaSessionController.java。Trigger Session Change false for previous Source if previous source exists. 都会调用notifySessionChange(MediaController mc, boolean flag)

⑶、MediaController.PlaybackInfo

保存有关当前播放以及如何处理此会话的音频的信息，也可以获取当前播放的音频信息，包含播放的进度、时长等

```
// 获取当前回话播放的音频信息
val playbackInfo = mMediaController.playbackInfo
```

❶、MediaController.PlaybackInfo相关组件API列表（可选）

除了上面onAudioInfoChanged()、onExtrasChanged两个方法，其它相关API列表

方法名	说明
AudioAttributes getAudioAttributes()	获取此会话的音频属性
int getCurrentVolume()	获取此会话的当前音量
int getMaxVolume()	获取可为此会话设置的最大音量
int getPlaybackType()	获取影响音量处理的播放类型
int getVolumeControl()	获取可以使用的音量控件的类型
String getVolumeControlId()	获取此会话的音量控制 ID

⑷、MediaController.TransportControls

用于控制**MediaSession会话**中媒体播放的接口。这允许客户端使用**控制器MediaController**，来发送如**系统的媒体按钮**（像方向盘上面的按钮）、命令（如【上一曲】、【下一曲】）和传输控件到MediaSession。它也与MediaSession.Callback（上面↑↑↑<*二.1.3、MediaSession#MediaSession.Callback*>）中方法一对应

```
private val connectionCallback = object : MediaBrowser.ConnectionCallback() {

    override fun onConnected() {
        super.onConnected()
        // ...
        if(mMediaBrowser.isConnected) {
            val sessionToken = mMediaBrowser.sessionToken
            mMediaController = MediaController(applicationContext,sessionToken)
            // 请求播放器在其当前位置开始播放。
            mMediaController.transportControls.play()
            // 请求播放器暂停播放并保持在当前位置。
            mMediaController.transportControls.pause()
        }
    }
}
```

❶、MediaController.TransportControls相关组件API列表（可选）

除了上面play()和pause()两个方法以外，其它API如下所示

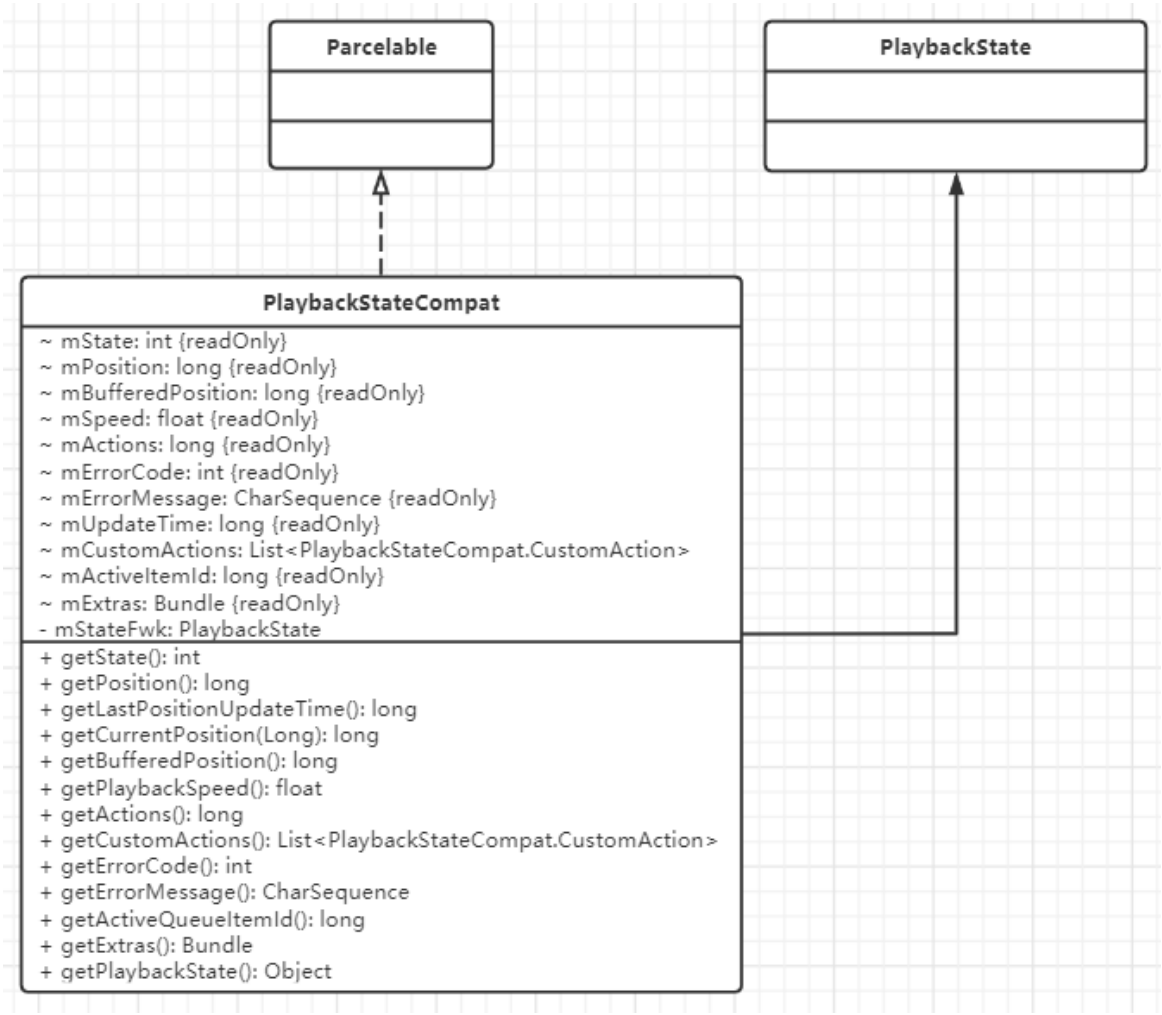
方法名	说明
void fastForward()	开始快进
void playFromMediaId (String mediaId, Bundle extras)	请求播放器开始播放特定媒体 ID
void playFromSearch (String query, Bundle extras)	请求播放器开始播放特定的搜索查询
void playFromUri (Uri uri, Bundle extras)	请求播放器开始播放特定Uri
void prepare()	请求播放器准备播放
void prepareFromMediaId (String mediaId, Bundle extras)	请求播放器为特定媒体 ID 准备播放
void prepareFromSearch (String query, Bundle extras)	请求播放器为特定搜索查询准备播放
void prepareFromUri (Uri uri, Bundle extras)	请求播放器为特定Uri
void rewind()	开始倒退
void seekTo(long pos)	移动到媒体流中的新位置
void sendCustomAction (PlaybackState.CustomAction customAction, Bundle args)	发送自定义操作以供MediaSession执行
void sendCustomAction (String action,Bundle args)	将自定义操作中的 id 和 args 发送回去，以便MediaSession执行
void setPlaybackSpeed (float speed)	设置播放速度
void setRating(Rating rating)	对当前内容进行评级
void setRating(RatingCompat rating, Bundle extras)	对当前内容进行评级，可以用extras传递如mediaId等参数
void skipToNext()	跳到下一项
void skipToPrevious()	跳到上一项
void skipToQueueItem(long id)	在播放队列中播放具有特定 ID 的项目
void stop()	请求播放器停止播放;它可以以任何适当的方式清除其状态

2、其它相关API

MediaSession框架中还有一些同样重要的类

- 封装了各种**播放状态**的**PlaybackState**，**PlaybackState**类为我们定义了各种状态的规范
- 与Map相似通过**键值对**保存**媒体信息**的**MediaMetadata**
- 在**MediaBrowser**和**MediaBrowserService**之间进行数据交互的**MediaItem**（见上面↑↑↑<二、*1.1、MediaBrowser#⑷、MediaBrowser.MediaItem*>的内容）

2.1、播放器状态 - PlaybackState



用于承载播放状态的类。如当前播放位置和当前控制功能。在MediaSession.Callback更改状态后需要调用MediaSession.setPlaybackState把状态同步给客户端，回调客户端的`MediaController.Callback`的`onPlaybackStateChanged()`

```
private val callback = object : MediaSession.Callback() {
    override fun onPlay() {
        super.onPlay()
        // ...
        // 更新状态
        val state = PlaybackState.Builder()
            .setState(
                PlaybackState.STATE_PLAYING,1,1f
            )
            .build()
        mediaSession.setPlaybackState(state)
    }
}
```

⑴、PlaybackState相关组件API列表（可选）

方法名	说明
long getActions()	获取此会话上可用的当前操作
long getActiveQueueItemId()	获取队列中当前活动项的 ID
long getBufferedPosition()	获取当前缓冲位置（以毫秒为单位）
List getCustomActions()	获取自定义操作的列表
CharSequence getErrorMessage()	获取用户可读的错误消息
Bundle getExtras()	获取在此播放状态下设置的任何自定义附加内容
getLastPositionUpdateTime	获取上次更新位置的经过的实时时间
float getPlaybackSpeed()	获取当前播放速度作为正常播放的倍数
long getPosition()	获取当前播放位置（以毫秒为单位）
int getState()	获取当前播放状态
boolean isActive()	返回是否将其视为活动播放状态

⑵、PlaybackState.Builder

PlaybackState.Builder 主要用来创建 PlaybackState 对象，创建它使用的是建造者模式

```
//PlaybackState的构建
PlaybackState state = new PlaybackState.Builder()
        //三个参数分别是，状态，位置，播放速度
        .setState(PlaybackState.STATE_PLAYING,
                mMediaPlayer.getCurrentPosition(), PLAYBACK_SPEED)
        .setActions(PLAYING_ACTIONS)
        .addCustomAction(mShuffle)
        .setActiveQueueItemId(mQueue.get(mCurrentQueueIdx).getQueueId())
        .build();
```

❶、PlaybackState.Builder相关组件API列表（可选）

方法名	说明
PlaybackState.Builder addCustomAction(String action, String name, int icon)	将自定义操作添加到播放状态
PlaybackState.Builder addCustomAction (PlaybackState.CustomAction customAction)	将自定义操作添加到播放状态
PlaybackState.Builder setActions(long actions)	设置此会话上可用的当前操作
PlaybackState.Builder setActiveQueueItemId(long id)	通过指定活动项目的 id 来设置播放队列中的活动项目
PlaybackState.Builder setBufferedPosition(long bufferedPosition)	设置当前缓冲位置（以毫秒为单位）
PlaybackState.Builder setErrorMessage(CharSequence error)	设置用户可读的错误消息
PlaybackState.Builder setExtras(Bundle extras)	设置要包含在播放状态中的任何自定义附加内容

PlaybackState.Builder setState(int state, long position, float playbackSpeed)	设置当前播放状态，三个参数分别是：状态，位置，播放速度，他会调用下面这个同名方法默认更新时间为开机时间
PlaybackState.Builder setState(int state, long position, float playbackSpeed, long updateTime)	设置当前播放状态，四个参数分别是：状态，位置，播放速度，更新时间
PlaybackState build()	生成并返回具有这些值的PlaybackState实例

❷、MediaController.Callback PlaybackState的解析

```
//PlaybackState的解析
private MediaController.Callback mCallBack = new MediaController.Callback() {
     ....
     @Override
     public void onPlaybackStateChanged(PlaybackState playbackState) {
        super.onPlaybackStateChanged(state);
        //获得进度时长
        long position = playbackState.getPosition();

        //获得当前状态
        switch(playbackState.getState()){
            case PlaybackState.STATE_PLAYING:
                //正在播放
                ...
                break;
            case PlaybackState.STATE_PAUSED:
                //暂停
                ...
                break;
            case PlaybackState.ACTION_SKIP_TO_NEXT:
                //跳到下一首
                ...
                break;
            ...//还有很多状态标志,按需求添加
        }
    }
}
```

⚠ **注意**

> - 播放进度的获取需要具体逻辑进行计算，客户端和服务端逻辑统一就可以了。 简单的直接通过position表示播放进度也是ok的

⑶、PlaybackState.CustomAction

CustomActions可用于通过将特定于应用程序的操作发送给MediaControllers，这样就可以扩展标准传输控件的功能

```
CustomAction action = new CustomAction
        .Builder("android.car.media.localmediaplayer.shuffle",
        mContext.getString(R.string.shuffle),
        R.drawable.shuffle)
        .build();

PlaybackState state = new PlaybackState.Builder()
        .setState(PlaybackState.STATE_PLAYING,
                mMediaPlayer.getCurrentPosition(), PLAYBACK_SPEED)
        .setActions(PLAYING_ACTIONS)
        .addCustomAction(action)
```

```
            .setActiveQueueItemId(mQueue.get(mCurrentQueueIdx).getQueueId())
            .build();
```

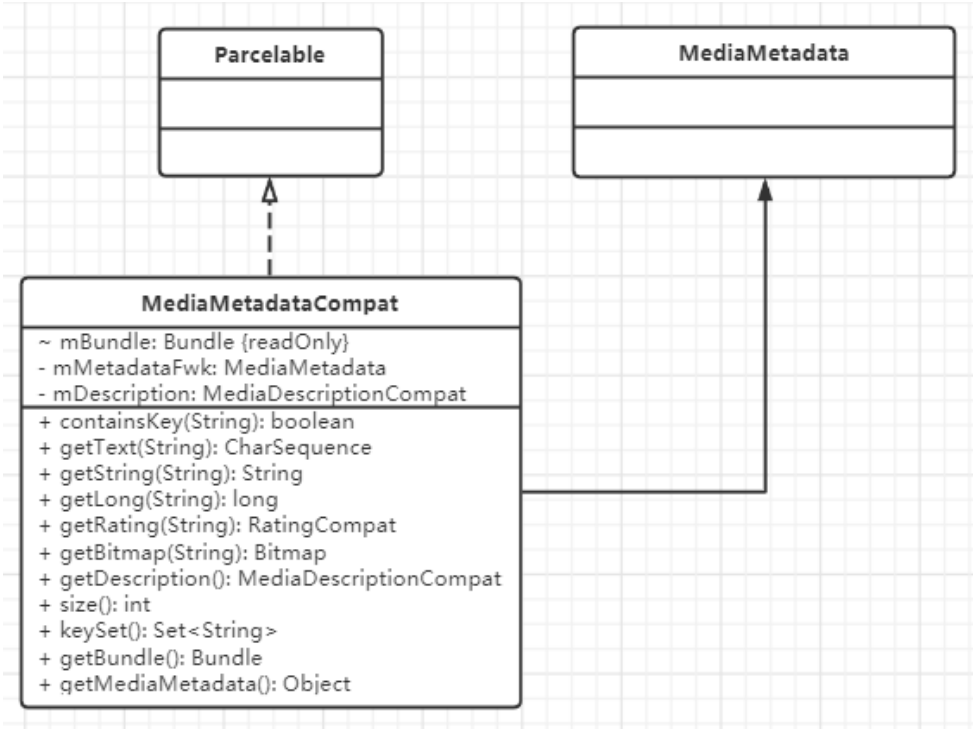
- PlaybackState.CustomAction API 说明

方法名	说明
String getAction()	返回CustomAction的action
Bundle getExtras()	返回附加项，这些附加项提供有关操作的其他特定于应用程序的信息，如果没有，则返回 null
int getIcon()	返回package中图标的资源 ID
CharSequence getName()	返回此操作的显示名称

2.2、元数据类 - MediaMetadata

和Map相似通过键值对保存媒体信息，包含有关项目的基础数据，例如标题、艺术家、专辑名、总时长等。一般需要服务端从本地数据库或远端查询出原始数据在封装成MediaMetadata再通过MediaSession.setMetadata(metadata)返回到客户端的MediaController.Callback.onMetadataChanged中

注意与MediaSession.QueueItem、MediaBrowser.MediaItem之间的差异



⑴、MediaMetadata API 说明

方法名	说明
boolean containsKey(String key)	如果给定的key包含在元数据中，则返回 true
int describeContents()	描述此可打包实例的封送处理表示中包含的特殊对象的种类
Bitmap getBitmap(String key)	返回给定的key的Bitmap;如果给定key不存在位图，则返回 null
int getBitmapDimensionLimit()	获取创建此元数据时位图的宽度/高度限制（以像素为单位）

MediaDescription getDescription()	获取此元数据的简单说明以进行显示
long getLong(String key)	返回与给定key关联的值，如果给定key不再存在，则返回 0L
Rating getRating(String key)	对于给定的key返回Rating;如果给定key不存在Rating，则返回 null
String getString(String key)	以String格式返回与给定key关联的文本值，如果给定key不存在所需类型的映射，或者null值显式与该key关联，则返回 null
CharSequence getText(String key)	返回与给定键关联的值，如果给定键不存在所需类型的映射，或者与该键显式关联 null 值，则返回 null
Set keySet()	返回一个 Set，其中包含在此元数据中用作key的字符串
int size()	返回此元数据中的字段数

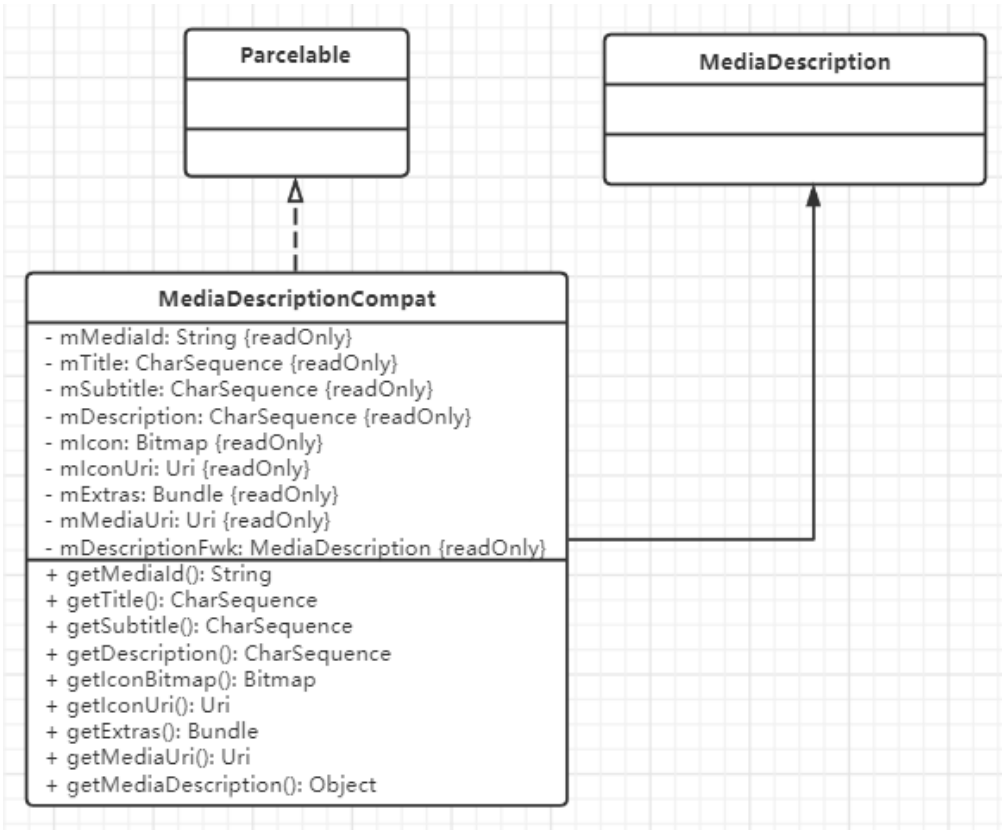
⑵、MediaMetadata 常用Key

方法名	说明
METADATA_KEY_ALBUM	媒体的唱片集标题
METADATA_KEY_ALBUM_ART	媒体原始来源的相册的插图，Bitmap格式
METADATA_KEY_ALBUM_ARTIST	媒体原始来源的专辑的艺术家
METADATA_KEY_ALBUM_ART_URI	媒体原始源的相册的图稿，Uri格式（推荐使用）
METADATA_KEY_ART	媒体封面，Bitmap格式
METADATA_KEY_ART_URI	媒体的封面，Uri格式
METADATA_KEY_ARTIST	媒体的艺术家
METADATA_KEY_AUTHOR	媒体的作者
METADATA_KEY_BT_FOLDER_TYPE	蓝牙 AVRCP 1.5 的 6.10.2.2 节中指定的媒体的蓝牙文件夹类型
METADATA_KEY_COMPILATION	媒体的编译状态
METADATA_KEY_COMPOSER	媒体的作曲家
METADATA_KEY_DATE	媒体的创建或发布日期
METADATA_KEY_DISC_NUMBER	介质原始来源的光盘编号
METADATA_KEY_DISPLAY_DESCRIPTION	适合向用户显示的说明
METADATA_KEY_DISPLAY_ICON	适合向用户显示的图标或缩略图
METADATA_KEY_DISPLAY_ICON_URI	适合向用户显示的图标或缩略图， Uri格式
METADATA_KEY_DISPLAY_SUBTITLE	适合向用户显示的副标题
METADATA_KEY_DISPLAY_TITLE	适合向用户显示的标题
METADATA_KEY_DURATION	媒体的持续时间（以毫秒为单位）
METADATA_KEY_GENRE	媒体的流派
METADATA_KEY_MEDIA_ID	用于标识内容的字符串Key
METADATA_KEY_MEDIA_URI	媒体内容，Uri格式
METADATA_KEY_NUM_TRACKS	媒体原始源中的曲目数
METADATA_KEY_RATING	媒体的总体评分

METADATA_KEY_TITLE	媒体的标题
METADATA_KEY_TRACK_NUMBER	媒体的磁道编号
METADATA_KEY_USER_RATING	用户对媒体的分级
METADATA_KEY_WRITER	媒体作家
METADATA_KEY_YEAR	媒体创建或发布为长的年份

2.3、MediaDescription

解析媒体信息类。与MediaMetadata的作用相对应




```
Bundle bundle = new Bundle();
bundle.putLong(Constants.EXTRA_MEDIA_NODE_ID, node.getNodeId());
MediaDescriptionCompat desc = new MediaDescriptionCompat.Builder()
        .setMediaId(mediaId)
        .setIconBitmap(iconBitmap)
        .setMediaUri(createUri(mediaUri, mediaPath))
        .setTitle(mediaTitle)
        .setExtras(bundle)
        .setIconUri(thumbnail != Uri.EMPTY ? thumbnail : Uri.parse(""))
        .setSubtitle(getSubtitle(node, isSearching))
        .build();
```

- 其中setIconBitmap()可以用于传递一些图片的位图，比如我们在MtpDocumentsProvider中获取到视频或者音频的缩略图，thumbli等可以放在这里面
- setMediaUri用于传递媒体（图片，视频，音频的）ContentUri

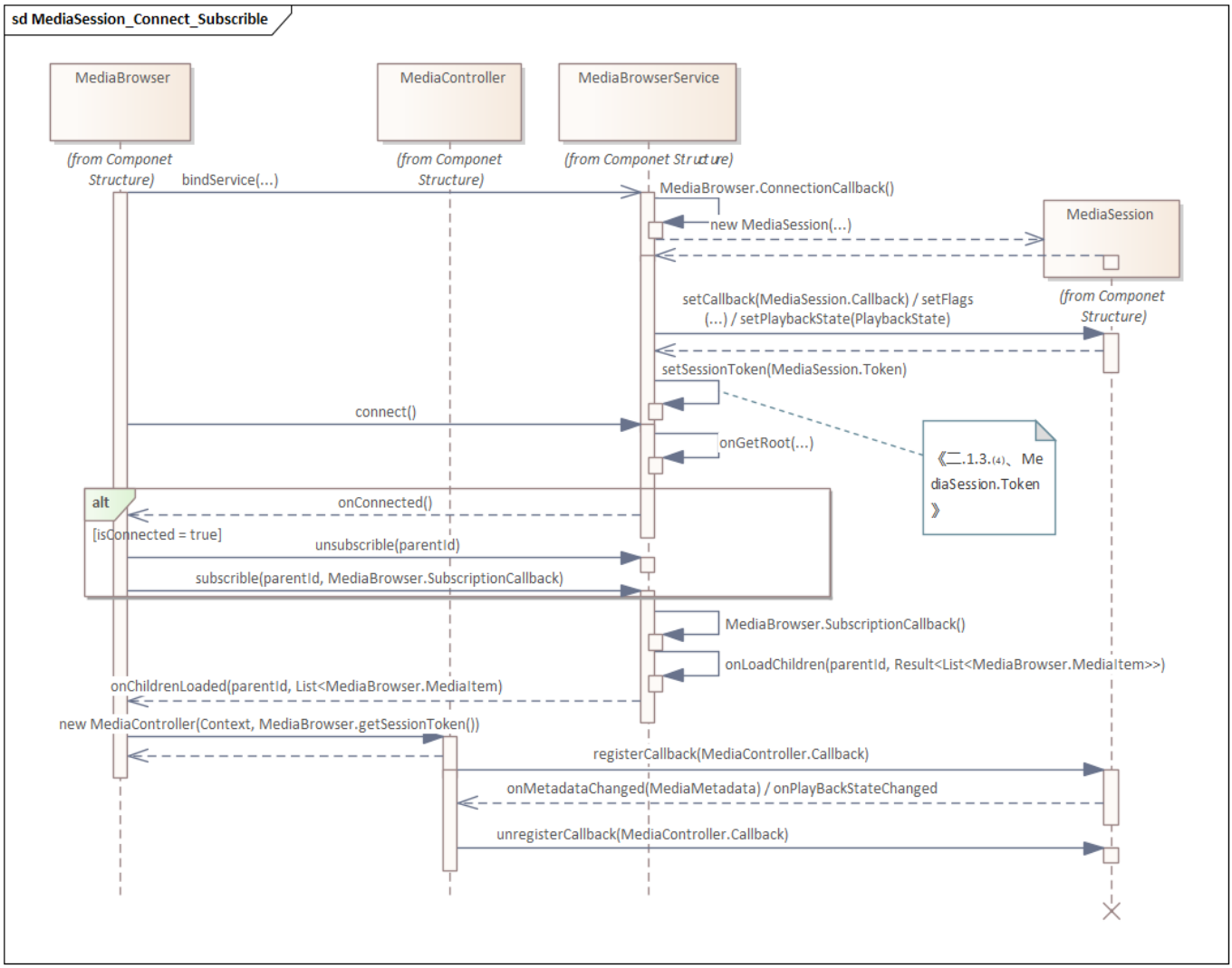
⚠ **注意**

> 通过subscribe的客户端就可以通过拿到的MediaDescription来解析数据，做显示UI的操作

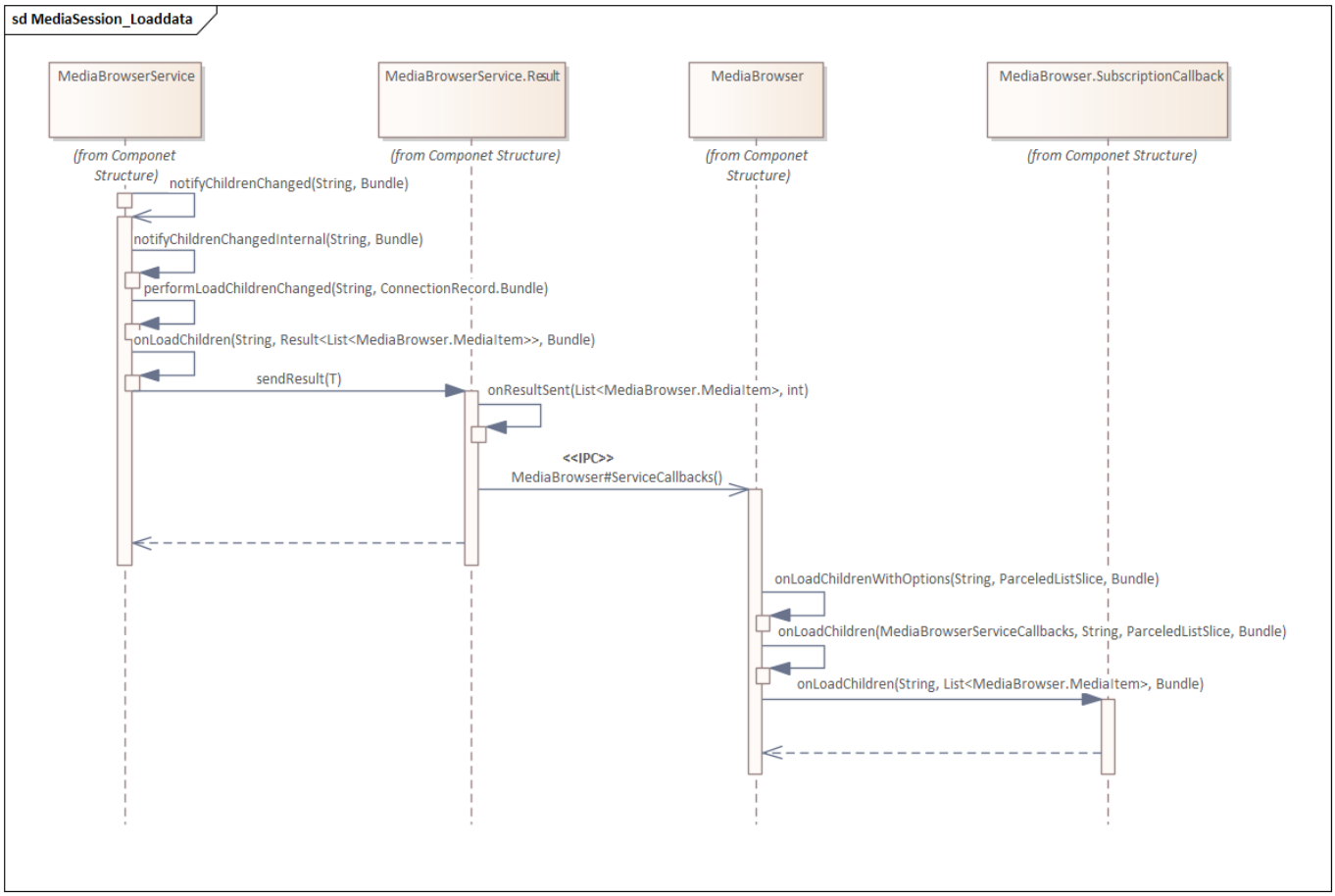
3、连接订阅/数据加载/媒体控制的流程

上面↑↑↑的内容基本上把**MediaSession框架**的所有API都理清楚和讲解清楚了，俗话说兵马未动，粮草先行。在实战之前（在下面↓↓↓<*三、MediaSession构建简单的播放器*>），先来画一下连接订阅、数据加载，媒体控制的大致流程，通过这个图也能帮助大家理解，和了解到开发一个媒体类别的框架设计思想思路

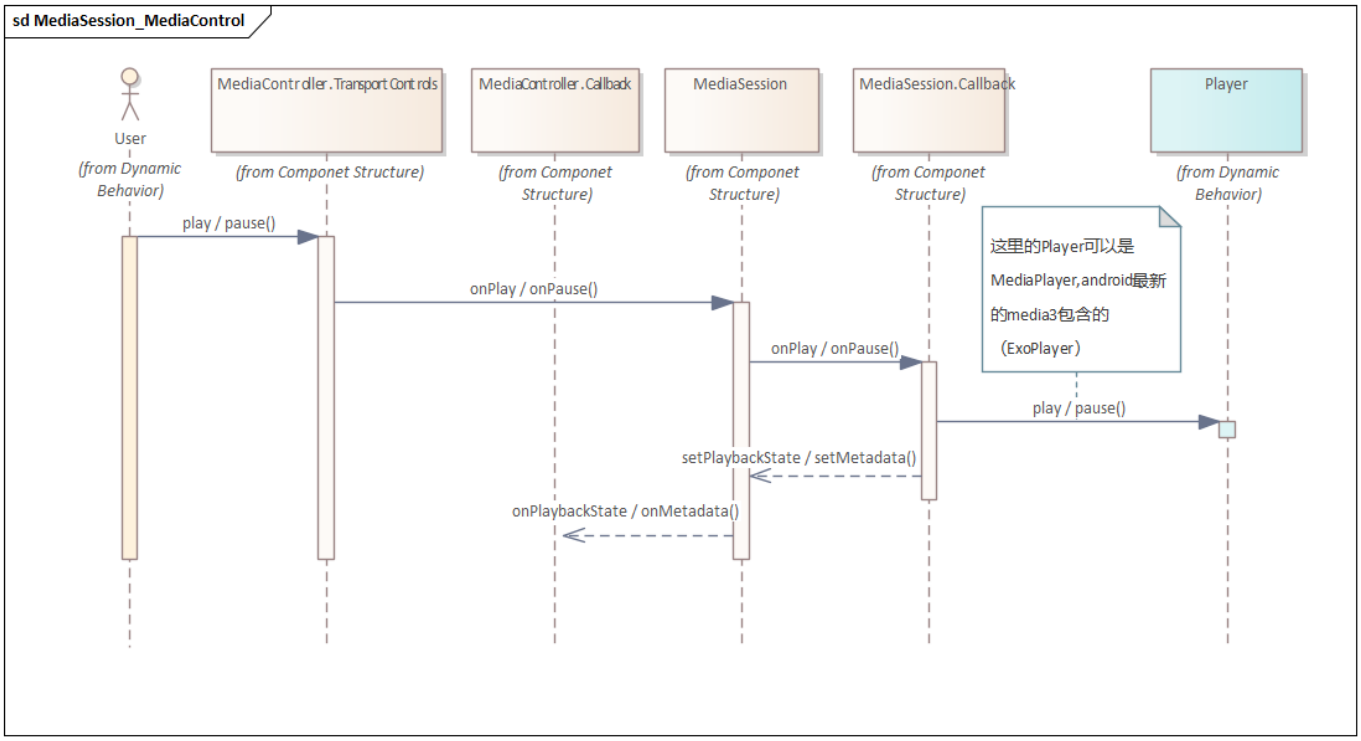
3.1、连接订阅



3.2、数据加载



3.3、媒体控制



本文有点重，就不拆了，如果是查API的，看到这儿下面↓↓↓的就属于可选内容了

@版权 声明CopyRight

> 本内容作者：sunst0069，转载与引用需标明出处，违者追究法律责任！！！

4、MediaSession实战项目接口对照表

在上面↑↑↑提到了很多一一对应的接口关系，本文内容比较重，一个个不方便，故整理一版MediaSession客户端和服务端调用方法的对照关系。在你学习一个新的东西的时候，这种整理习惯是很有用处的，比如画画图

4.1、MediaSession API对照关系

这里也可以看到MediaSession框架中，各个类的包名信息

(1)、服务端类

方法名	说明
android.media.session.MediaSession	受控端
android.media.session.MediaSession.Token	配对密钥
android.media.session.MediaSession.Callback	受控端回调，可以接受到控制端的指令

(2)、客户端类

方法名	说明
android.media.session.MediaController	控制端
android.media.session.MediaController.TransportControls	控制端的控制器，用于发送指令
android.media.session.MediaController.Callback	控制端回调，可以接受到受控端的状态
android.media.browse.MediaBrowser.SubscriptionCallback	订阅信息回调

(3)、客户端调用服务端

TransportControls	MediaSession.Callback	说明
play()	onPlay()	播放
pause()	onPause()	暂停
seekTo(long pos)	onSeekTo(long)	指定播放位置
fastForward()	onFastForward()	快进
rewind()	onRewind()	回倒
skipToNext()	onSkipToNext()	下一首
skipToPrevious()	onSkipToPrevious()	上一首
skipToQueueItem(long)	onSkipToQueueItem(long)	指定id播放。指定的是Queue的id
playFromMediaId(String,Bundle)	onPlayFromMediaId(String,Bundle)	指定id播放。指定的是MediaMetadata的id
playFromSearch(String,Bundle)	onPlayFromSearch(String,Bundle)	搜索播放。比如像globalSearch APP中可以用
playFromUri(Uri,Bundle)	onPlayFromUri(Uri,Bundle)	指定uri播放
sendCustomAction(String,Bundle)	onCustomAction(String,Bundle)	发送自定义动作。可用来更换播放模式、重新加载音乐列表等
setRating(Rating rating)	onSetRating(Rating)	打分。内置的评分系统有星级、红心、赞/踩、百分比
setRating(RatingCompat rating, Bundle extras)	onSetRating(RatingCompat rating, Bundle extras)	打分。内置的评分系统有星级、红心、赞/踩、百分比。可以用extras接受如mediaId等参数

(4)、服务端回调至客户端

MediaSession	MediaController.Callback	说明
setMetadata(MediaMetadata)	onMetadataChanged(MediaMetadata)	当前播放音乐
setPlaybackState(PlaybackState)	onPlaybackStateChanged(PlaybackState)	播放状态
setQueue(List MediaSession.QueueItem>)	onQueueChanged(List MediaSession.QueueItem>)	播放队列
setQueueTitle(CharSequence)	onQueueTitleChanged(CharSequence)	播放队列标题
setExtras(Bundle)	onExtrasChanged(Bundle)	额外信息。可以记录播放模式等信息
sendSessionEvent(String,Bundle)	onSessionEvent(String, Bundle)	自定义事件

4.2、实战项目ExternalService对照表

这是公司项目ExternalService学习中整理的MediaSession接口对照ExternalService。当时还在用support.v4

ExternalMediaServer	support.v4.media	MediaCoreService	third mode
MediaAdapterService	mediasession	HarmanXXXPlayer	
sourceprovider			
一、公共部分SourceProvider			
SourceProvider.onActiveSessionChange	MediaSessionManager.OnActiveSessionsChangedListener{onActiveSessionsChanged}		
二、AlexaSourceProvider		HarmanStreamingPlayer	OnlinePlayer
#1、连接订阅			
connectTo	MediaBrowserCompat.connect	onGetRoot	
subscribe	MediaBrowserCompat.subscribe	onLoadChildren	
onChildrenLoaded(String parentId, List<MediaBrowserCompat.MediaItem> mediaItems)	(MediaBrowserServiceCompat.Result<List<MediaBrowserCompat.MediaItem>>).sendResult(List<MediaBrowserCompat.MediaItem>)	Utils.sendResult(result, itemList)	
#2、播放控制SessionPlayer			
SessionPlayer.play()	MediaControllerCompat..getTransportControls().play()	OnlineStreamingPlayer.play()	OnlineRadioPlayer.playFromUri()
pause()	MediaControllerCompat..getTransportControls().pause()	OnlineStreamingPlayer.pause()	OnlineRadioPlayer.pause()
stop	MediaControllerCompat..getTransportControls().stop()	OnlineStreamingPlayer.stop()	OnlineRadioPlayer.stop()
快进fastForward	MediaControllerCompat..getTransportControls().fastForward()	OnlineStreamingPlayer.seekTo(true, (long) msg.obj)	OnlineRadioPlayer.seekTo(long positionMs)
快退fastRewind	MediaControllerCompat..getTransportControls().rewind()	OnlineStreamingPlayer.seekTo(false, (long) msg.obj)	OnlineRadioPlayer.seekTo(long positionMs)
skipToNext	MediaControllerCompat..getTransportControls().skipToNext()	OnlineStreamingPlayer.next()	IStreamingSessionCallback.onNextTrack
skipToPrevious	MediaControllerCompat..getTransportControls().skipToPrevious()	OnlineStreamingPlayer.previous()	IStreamingSessionCallback.onPreviousTrack
seekToTime	MediaControllerCompat..getTransportControls().seekTo()	OnlineStreamingPlayer.seekTo()	OnlineRadioPlayer.seekTo(long positionMs)
setRepeatMode	MediaControllerCompat.getTransportControls().setRepeatMode()	OnlineStreamingPlayer.setRepeat()	IStreamingSessionCallback.updateRepeatMode
setShuffleMode	MediaControllerCompat..getTransportControls().setShuffleMode()	OnlineStreamingPlayer.setRandom(int randomMode)	IStreamingSessionCallback.updateShuffleMode
playFromMediaId	MediaControllerCompat..getTransportControls().playFromMediaId()	OnlineStreamingPlayer.play()	OnlineRadioPlayer.playFromUri()
playFromUri	MediaControllerCompat..getTransportControls().playFromUri()	OnlineStreamingPlayer.playFromUri()	OnlineRadioPlayer.playFromUri()
sendCustomCommand	MediaControllerCompat.sendCommand(String, Bundle)		
sendCustomAction	MediaControllerCompat..getTransportControls().sendCustomAction(String, Bundle)	HarmanStreamingPlayerAdapter.customAction	NA
#3、事件回调SessionPlayer			
onPlaybackStateChanged	MediaSessionCompat.setPlaybackState(PlaybackStateCompat)	OnlineStreamingPlayer.updatePlaybackStatus()	
onMetadataChanged	MediaSessionCompat.setMetadata(MediaMetadataCompat)	OnlineStreamingPlayer.onMetaDataChanged()	
onRepeatModeChanged	MediaSessionCompat.setRepeatMode(repeatMode)	OnlineStreamingPlayer.notifyShuffleModeChanged()	
onShuffleModeChanged	MediaSessionCompat.setShuffleMode(shuffleMode)	OnlineStreamingPlayer.notifyRepeatModeChanged()	
onQueueChanged (未处理)	NA	NA	
onSessionDestroyed (未处理)	NA	NA	
onSessionEvent	MediaSessionCompat.sendSessionEvent(deviceMediaId)		
setActive	MediaSessionCompat.setActive(boolean)		

三、MediaSession构建一个简单的播放器

demo效果图如下，只提供简单的播放暂停操作，音乐数据源从raw资源文件夹中获取

1、客户端类DemoActivity.java

首先界面上方添加一个RecyclerView来展示获取的音乐列表

1.1、RecyclerView的初始化

```
public class DemoActivity extends AppCompatActivity {
    private RecyclerView recyclerView;
    private List<MediaBrowserCompat.MediaItem> list;
    private DemoAdapter demoAdapter;
```

```
    private LinearLayoutManager layoutManager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_demo);

        list = new ArrayList<>();
        layoutManager = new LinearLayoutManager(this);
        layoutManager.setOrientation(LinearLayoutManager.VERTICAL);
        demoAdapter = new DemoAdapter(this,list);

        recyclerView = (RecyclerView) findViewById(R.id.recyclerView);
        recyclerView.setLayoutManager(layoutManager);
        recyclerView.setAdapter(demoAdapter);
    }
}
```

⚠ **注意**

> 注意List元素的类型为MediaBrowserCompat.MediaItem，因为MediaBrowser从服务中获取的每一首音乐都会封装成MediaItem对象

1.2、创建MediaBrowser，并执行连接服务端和订阅数据的操作

```
public class DemoActivity extends AppCompatActivity {
    ...
    private MediaBrowserCompat mBrowser;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        mBrowser = new MediaBrowserCompat(
                this,
                new ComponentName(this, MusicService.class),//绑定浏览器服务
                BrowserConnectionCallback,//设置连接回调
                null
        );
    }

    @Override
    protected void onStart() {
        super.onStart();
        //Browser发送连接请求
        mBrowser.connect();
    }

    @Override
    protected void onStop() {
        super.onStop();
        mBrowser.disconnect();
    }

    /**
     * 连接状态的回调接口，连接成功时会调用onConnected()方法
     */
    private MediaBrowserCompat.ConnectionCallback BrowserConnectionCallback =
            new MediaBrowserCompat.ConnectionCallback(){
                @Override
                public void onConnected() {
```

```
                    Log.e(TAG,"onConnected------");
                    //必须在确保连接成功的前提下执行订阅的操作
                    if (mBrowser.isConnected()) {
                        //mediaId即为MediaBrowserService.onGetRoot的返回值
                        //若Service允许客户端连接，则返回结果不为null，其值为数据内容层次结构的根ID
                        //若拒绝连接，则返回null
                        String mediaId = mBrowser.getRoot();

                        //Browser通过订阅的方式向Service请求数据，发起订阅请求需要两个参数，其一为
mediaId
                        //而如果该mediaId已经被其他Browser实例订阅，则需要在订阅之前取消mediaId的订阅
者
                        //虽然订阅一个 已被订阅的mediaId 时会取代原Browser的订阅回调，但却无法触发
onChildrenLoaded回调

                        //ps：虽然基本的概念是这样的，但是Google在官方demo中有这么一段注释...
                        // This is temporary: A bug is being fixed that will make subscribe
                        // consistently call onChildrenLoaded initially, no matter if it is
replacing an existing
                        // subscriber or not. Currently this only happens if the mediaID has no
previous
                        // subscriber or if the media content changes on the service side, so we
need to
                        // unsubscribe first.
                        //大概的意思就是现在这里还有BUG，即只要发送订阅请求就会触发onChildrenLoaded回
调
                        //所以无论怎样我们发起订阅请求之前都需要先取消订阅
                        mBrowser.unsubscribe(mediaId);
                        //之前说到订阅的方法还需要一个参数，即设置订阅回调SubscriptionCallback
                        //当Service获取数据后会将数据发送回来，此时会触发
SubscriptionCallback.onChildrenLoaded回调
                        mBrowser.subscribe(mediaId, BrowserSubscriptionCallback);
                    }
                }

                @Override
                public void onConnectionFailed() {
                    Log.e(TAG,"连接失败！");
                }
            };
    /**
     * 向媒体浏览器服务(MediaBrowserService)发起数据订阅请求的回调接口
     */
    private final MediaBrowserCompat.SubscriptionCallback BrowserSubscriptionCallback =
            new MediaBrowserCompat.SubscriptionCallback(){
                @Override
                public void onChildrenLoaded(@NonNull String parentId,
                                             @NonNull List<MediaBrowserCompat.MediaItem>
children) {
                    Log.e(TAG,"onChildrenLoaded------");
                    //children 即为Service发送回来的媒体数据集合
                    for (MediaBrowserCompat.MediaItem item:children){
                        Log.e(TAG,item.getDescription().getTitle().toString());
                        list.add(item);
                    }
                    //在onChildrenLoaded可以执行刷新列表UI的操作
                    demoAdapter.notifyDataSetChanged();
                }
            };
}
```

通过上面代码和注释可以知道：MediaBrowser从连接服务，到向MediaBrowserService订阅数据的流程是这样子

```
connect → onConnected → subscribe → onChildrenLoaded
```

2、Service端MusicService

2.1、配置AndroidManifest.xml

```
<service
    android:name=".demo.MusicService">
    <intent-filter>
        <action android:name="android.media.browse.MediaBrowserService" />
    </intent-filter>
</service>
```

2.2、MusicService实现

通过继承MediaBrowserService（这里使用了support-v4包的类）创建MusicService类。**MediaBrowserService继承自Service**

```
public class MusicService extends MediaBrowserServiceCompat {
    private MediaSessionCompat mSession;
    private PlaybackStateCompat mPlaybackState;

    @Override
    public void onCreate() {
        super.onCreate();
        mPlaybackState = new PlaybackStateCompat.Builder()
                .setState(PlaybackStateCompat.STATE_NONE,0,1.0f)
                .build();

        mSession = new MediaSessionCompat(this,"MusicService");
        mSession.setCallback(SessionCallback);//设置回调
        mSession.setFlags(MediaSessionCompat.FLAG_HANDLES_TRANSPORT_CONTROLS);
        mSession.setPlaybackState(mPlaybackState);

        //设置token后会触发MediaBrowserCompat.ConnectionCallback的回调方法
        //表示MediaBrowser与MediaBrowserService连接成功
        setSessionToken(mSession.getSessionToken());
    }
}
```

调用MediaSession.setFlag为Session设置标志位，以便Session接收控制器的指令。然后是播放状态的设置，需调用MediaSession.setPlaybackState，那么PlaybackState又是什么呢？之前我们简单介绍过它是封装了各种播放状态的类，我们可以通过判断当前播放状态来控制各个成员的行为，而PlaybackState类为我们定义了各种状态的规范。此外我们还需要设置SessionCallback回调，当客户端使用控制器发送指令时，就会触发这些回调方法，从而达到控制播放器的目的

```
public class MusicService extends MediaBrowserServiceCompat {
    ...
    private MediaPlayer mMediaPlayer;

    @Override
    public void onCreate() {
        ...
        mMediaPlayer = new MediaPlayer();
```

```
        mMediaPlayer.setOnPreparedListener(PreparedListener);
        mMediaPlayer.setOnCompletionListener(CompletionListener);
    }

    /**
     * 响应控制器指令的回调
     */
    private android.support.v4.media.session.MediaSessionCompat.Callback SessionCallback = new
MediaSessionCompat.Callback(){
        /**
         * 响应MediaController.getTransportControls().play
         */
        @Override
        public void onPlay() {
            Log.e(TAG,"onPlay");
            if(mPlaybackState.getState() == PlaybackStateCompat.STATE_PAUSED){
                mMediaPlayer.start();
                mPlaybackState = new PlaybackStateCompat.Builder()
                        .setState(PlaybackStateCompat.STATE_PLAYING,0,1.0f)
                        .build();
                mSession.setPlaybackState(mPlaybackState);
            }
        }

        /**
         * 响应MediaController.getTransportControls().onPause
         */
        @Override
        public void onPause() {
            Log.e(TAG,"onPause");
            if(mPlaybackState.getState() == PlaybackStateCompat.STATE_PLAYING){
                mMediaPlayer.pause();
                mPlaybackState = new PlaybackStateCompat.Builder()
                        .setState(PlaybackStateCompat.STATE_PAUSED,0,1.0f)
                        .build();
                mSession.setPlaybackState(mPlaybackState);
            }
        }

        /**
         * 响应MediaController.getTransportControls().playFromUri
         * @param uri
         * @param extras
         */
        @Override
        public void onPlayFromUri(Uri uri, Bundle extras) {
            Log.e(TAG,"onPlayFromUri");
            try {
                switch (mPlaybackState.getState()){
                    case PlaybackStateCompat.STATE_PLAYING:
                    case PlaybackStateCompat.STATE_PAUSED:
                    case PlaybackStateCompat.STATE_NONE:
                        mMediaPlayer.reset();
                        mMediaPlayer.setDataSource(MusicService.this,uri);
                        mMediaPlayer.prepare();//准备同步
                        mPlaybackState = new PlaybackStateCompat.Builder()
                                .setState(PlaybackStateCompat.STATE_CONNECTING,0,1.0f)
                                .build();
                        mSession.setPlaybackState(mPlaybackState);
                        //我们可以保存当前播放音乐的信息，以便客户端刷新UI
                        mSession.setMetadata(new MediaMetadataCompat.Builder()
```

```
.putString(MediaMetadataCompat.METADATA_KEY_TITLE,extras.getString("title"))
                                .build()
                        );
                        break;
                }
            }catch (IOException e){
                e.printStackTrace();
            }
        }

        @Override
        public void onPlayFromSearch(String query, Bundle extras) {
        }
    };

    /**
     * 监听MediaPlayer.prepare()
     */
    private MediaPlayer.OnPreparedListener PreparedListener = new
MediaPlayer.OnPreparedListener() {
        @Override
        public void onPrepared(MediaPlayer mediaPlayer) {
            mMediaPlayer.start();
            mPlaybackState = new PlaybackStateCompat.Builder()
                    .setState(PlaybackStateCompat.STATE_PLAYING,0,1.0f)
                    .build();
            mSession.setPlaybackState(mPlaybackState);
        }
    } ;

    /**
     * 监听播放结束的事件
     */
    private MediaPlayer.OnCompletionListener CompletionListener = new
MediaPlayer.OnCompletionListener() {
        @Override
        public void onCompletion(MediaPlayer mediaPlayer) {
            mPlaybackState = new PlaybackStateCompat.Builder()
                    .setState(PlaybackStateCompat.STATE_NONE,0,1.0f)
                    .build();
            mSession.setPlaybackState(mPlaybackState);
            mMediaPlayer.reset();
        }
    };
}
```

在上面↑↑↑<*二.1.3、MediaSession#⑵、MediaSession.Callback*>已经知道MediaSession.Callback**中还有许多回调方法**，可以按需覆盖重写即可

⚠ **注意再次提醒**

> 构建好**MediaSession后记得调用setSessionToken保存Session的配对令牌**，同时调用此方法也会**回调客户端MediaBrowser.ConnectionCallback的onConnected方法**，告知客户端Browser与BrowserService连接成功了，我们也就完成了MediaSession的创建和初始化

2.3、MediaBrowser与MediaBrowserService的订阅关系

MediaBrowserService中我们需要重写onGetRoot和onLoadChildren方法，其作用之前已经讲过就不多赘述了

```
public class MusicService extends MediaBrowserServiceCompat {
    @Nullable
    @Override
    public BrowserRoot onGetRoot(@NonNull String clientPackageName, int clientUid, @Nullable
Bundle rootHints) {
        Log.e(TAG,"onGetRoot-----------");
        return new BrowserRoot(MEDIA_ID_ROOT, null);
    }

    @Override
    public void onLoadChildren(@NonNull String parentId, @NonNull final
Result<List<MediaBrowserCompat.MediaItem>> result) {
        Log.e(TAG,"onLoadChildren--------");
        //将信息从当前线程中移除，允许后续调用sendResult方法
        result.detach();

        //我们模拟获取数据的过程，真实情况应该是异步从网络或本地读取数据
        MediaMetadataCompat metadata = new MediaMetadataCompat.Builder()
                .putString(MediaMetadataCompat.METADATA_KEY_MEDIA_ID, ""+R.raw.jinglebells)
                .putString(MediaMetadataCompat.METADATA_KEY_TITLE, "圣诞歌")
                .build();
        ArrayList<MediaBrowserCompat.MediaItem> mediaItems = new ArrayList<>();
        mediaItems.add(createMediaItem(metadata));

        //向Browser发送数据
        result.sendResult(mediaItems);
    }

    private MediaBrowserCompat.MediaItem createMediaItem(MediaMetadataCompat metadata){
        return new MediaBrowserCompat.MediaItem(
                metadata.getDescription(),
                MediaBrowserCompat.MediaItem.FLAG_PLAYABLE
        );
    }
}
```

3、客户端控制器MediaController

回到客户端这边，四大核心类中还剩下控制器MediaController没讲。**MediaController的创建依赖于Session的配对令牌，当Browser和BrowserService连接成功就可以通过**`Browser拿到这个令牌`。控制器创建后，我们就可以通过`MediaController.getTransportControls`发送播放指令，同时也可以注册MediaControllerCompat.Callback回调接收播放状态，用以刷新界面UI

```
public class DemoActivity extends AppCompatActivity {
    ...
    private Button btnPlay;
    private TextView textTitle;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        btnPlay = (Button) findViewById(R.id.btn_play);
        textTitle = (TextView) findViewById(R.id.text_title);
    }

    public void clickEvent(View view) {
        switch (view.getId()) {
            case R.id.btn_play:
                if(mController!=null){
```

```
                    handlerPlayEvent();
                }
                break;
        }
    }

    /**
     * 处理播放按钮事件
     */
    private void handlerPlayEvent(){
        switch (mController.getPlaybackState().getState()){
            case PlaybackStateCompat.STATE_PLAYING:
                mController.getTransportControls().pause();
                break;
            case PlaybackStateCompat.STATE_PAUSED:
                mController.getTransportControls().play();
                break;
            default:
                mController.getTransportControls().playFromSearch("", null);
                break;
        }
    }

    /**
     * 连接状态的回调接口，连接成功时会调用onConnected()方法
     */
    private MediaBrowserCompat.ConnectionCallback BrowserConnectionCallback =
            new MediaBrowserCompat.ConnectionCallback(){
                @Override
                public void onConnected() {
                    Log.e(TAG,"onConnected------");
                    if (mBrowser.isConnected()) {
                        ...
                        try{
                            mController = new
MediaControllerCompat(DemoActivity.this,mBrowser.getSessionToken());
                            //注册回调
                            mController.registerCallback(ControllerCallback);
                        }catch (RemoteException e){
                            e.printStackTrace();
                        }
                    }
                }

                @Override
                public void onConnectionFailed() {
                    Log.e(TAG,"连接失败！");
                }
            };

    /**
     * 媒体控制器控制播放过程中的回调接口，可以用来根据播放状态更新UI
     */
    private final MediaControllerCompat.Callback ControllerCallback =
            new MediaControllerCompat.Callback() {
                /***
                 * 音乐播放状态改变的回调
                 * @param state
                 */
                @Override
                public void onPlaybackStateChanged(PlaybackStateCompat state) {
                    switch (state.getState()){
```

```
                        case PlaybackStateCompat.STATE_NONE://无任何状态
                            textTitle.setText("");
                            btnPlay.setText("开始");
                            break;
                        case PlaybackStateCompat.STATE_PAUSED:
                            btnPlay.setText("开始");
                            break;
                        case PlaybackStateCompat.STATE_PLAYING:
                            btnPlay.setText("暂停");
                            break;
                    }
                }

                /**
                 * 播放音乐改变的回调
                 * @param metadata
                 */
                @Override
                public void onMetadataChanged(MediaMetadataCompat metadata) {
                    textTitle.setText(metadata.getDescription().getTitle());
                }
            };

    private Uri rawToUri(int id){
        String uriStr = "android.resource://" + getPackageName() + "/" + id;
        return Uri.parse(uriStr);
    }
}
```

四、附录（可选）

这里是官方Demo和一些MediaSession的控制命令

1、官方Demo

- UAMP — 通用音乐播放器使用 ExoPlayer 来播放本地音频。
- ExoPlayer 演示应用 — 官方代码库中包含一个演示应用，该应用展示了该库的许多高级功能。
- 通过 MediaSession 控制媒体
- Google官方UniversalMusicPlayer源码 可以通过Google官方demo，看看播放进度条、播放队列控制、通知栏上的快捷操作等功能是如何结合MediaSession框架实现的

2、其他媒体APP

通过MediaBrowser和MediaControl连接MusicDemo的Service，就可以远程控制MusicDemo中的音乐播放。 比如：可以只实现UI上的MediaBrowser和MediaControl连接到官方的UAMP上可以支持操作。

3、adb shell 提供的media控制

```
adb shell media dispatch pause
adb shell media dispatch play
adb shell media dispatch play-pause
adb shell media dispatch fast-forward
adb shell media dispatch rewind
```

4、支持物理按键的控制-MediaSession的回调方法onMediaButtonEvent中

```
adb shell input keyevent 87 // next
adb shell input keyevent 88 // previous
adb shell input keyevent 126 // play
adb shell input keyevent 127 // pause
```

5、语音助手

我们的语音助手，可以控制播放的音乐： 如VPA， Google Assistant， HiCar，你好小安，你好小度，你好小德可以发送，指令"播放周杰伦的歌"， "暂停"， "继续"， "下一首"

"播放我收藏的歌曲"，有了MediaSession框架，类似这样语音的功能，就不需要每个音频都去实现了它的功能了

致谢（引用和推荐）

本文参考借鉴了以下文章部分内容，非常感谢各位前辈的开源精神，当代互联网的发展离不开你们的分享，再次感谢🙏.同时以下↓↓↓，也是本神推荐阅读系列

- ☑ **#*媒体应用架构概览 | Android 开发者 | Android Developers**
- ☑ **#*MediaSession | Android Developers**
- ☑ ****MediaSession框架全解析_qzns木雨的博客-CSDN博客_mediasession**
- ☑ ****车载多媒体（二）- 多媒体应用架构与MediaSession框架**
- ☑ ****Android车载多媒体开发——MediaSession框架**
- ☑ ****Android车载多媒体开发MediaSession框架示例详解**
- ☑ ****Android 媒体播放框架MediaSession分析与实践**
- ☑ ****Android MediaSession框架简析**
- ☑ **#*Audio and video overview**
- ☑ ****MediaSession框架的介绍和使用**
- ☑ ****MediaSession框架的源码分析**
- ☑ ****Android车载开发之Android Automotive**

@©LICENSE（版权和更新记录）

请尊重劳动成果，注意文中版权声明， Android专栏不定时更新，◯欢迎关注我的知乎Bgwan**(点我跳转)**。也可以同时关注人工智能专栏， 文艺语录专栏，技术上沟通可以在qyddai@gmail.com或知乎上留言

> 2023-10-08：首次发布《Android车载多媒体开发MediaSession框架理解（建议收藏）》
> 2023-12-28：维护更新——新增《二.1.1⑴、MediaBrowser相关API列表》↑↑↑→《非主线程创建MediaBrowser》
> 2024-04-18：维护更新——新增《MediaBrowserService类中方法在APP中ContentProvider和Application的执行的先后顺序》
> 2025-01-06：维护更新——编辑本文CD排版、布局、结构、用词、补充内容等

知乎是一个不错的平台，对于技术类的内容，不像CSDN需要付费阅读；当然整理本内容我也会花费了不少时间和精力，尤其是现在有了小baby要照顾；本着技术类分享精神，如果你觉得本文对你所帮助，你也可以分享给更多的同学，或支持一下#(香🔗)

 。授人以渔，不如授人以渔，开源和技术分享才能促进时代的发展

作者：sunst0069 Author sunst0069 初始日期⏲：2023-06-09；发布日期⏲：2023-10-08 14:38；更新日期⏲：2025-01-06；维护次数：4次