Android 0x03

0x00 Activity

> 在扯后续内容之前，先讲讲老生常谈的生命周期，内容参考自 Google开发文档: 了解Activity的生命周期

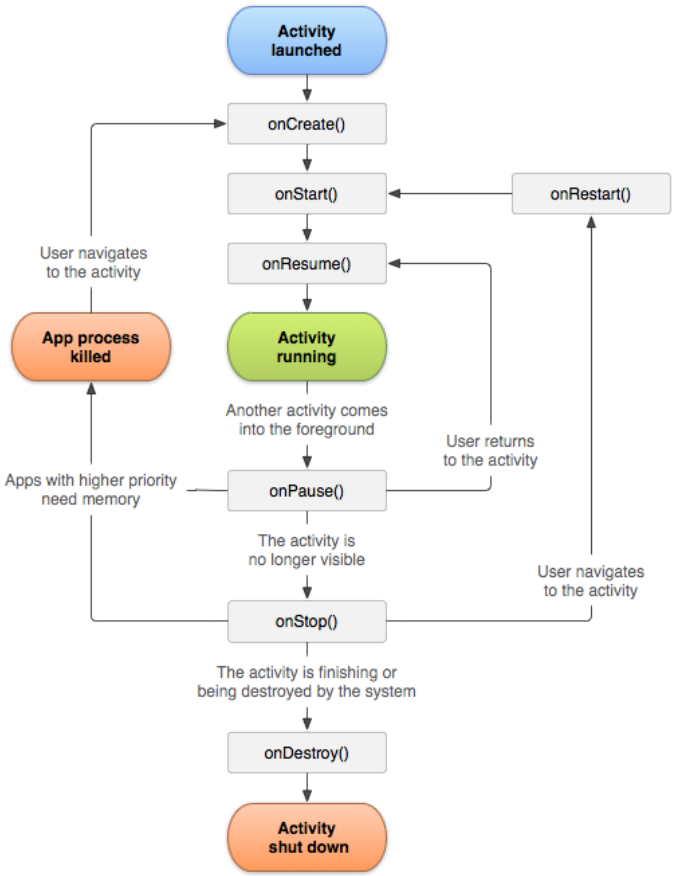
Activityの生命周期

什么是生命周期？（好🐶的一个问题），看看Google 爸爸怎么说

> 当用户浏览、退出和返回到您的应用时，您应用中的 `Activity` 实例会在其生命周期的不同状态间转换。`Activity` 类会提供许多回调，这些回调会让 Activity 知晓某个状态已经更改：系统正在创建、停止或恢复某个 `Activity`，或者正在销毁该 `Activity` 所在的进程。

在我的理解中，生命周期的本质目的是为了更好的用户体验，通过不同的生命周期阶段的回调和切换，使得开发者可以更好的处理用户的操作，在对应的生命周期去执行对应的事情，就可以给予用户更流畅的使用体验。

来复习一下Activity的各个生命周期吧，先来一张经典老图



在一切的一切之前，先明确一些概念

前台与后台🤲（foreground/background）

用户可以交互到的界面，或者说就是那块手机屏幕**当前**呈现的东西，一般而言，一个Activity处于前台代表这个Activity可以交互，处于后台则不可交互，甚至不可见

> 处于后台分为两个层次，一个是失去焦点，即不可交互，一个是不可见

可见与不可见👁（visible/invisible）

明确一下，这篇文档里面的可见与不可见指的是是否绘制并进入DecorView(稍后会介绍)，也可以简单的理解为用户是否可以用眼睛见到

可交互与不可交互👆

指的是Activity是否可以与用户交互，一般与是否进入前台相同步

焦点

这个和生活中焦点的概念很相似，“班级里的**焦点**人物”，“**焦点**访谈”中的焦点都具有类似的含义，在Android中可以浅显的认为，焦点就是当前用户所关注的点，丢失焦点可以理解为丢失用户的关注，获得焦点可以理解为得到了用户的关注。

由于一些Android设备是没有触控屏的（比如Android系统的电视），说控件获取焦点的方式是通过用户的点击是不太严谨的。

这里继续扯两句焦点的有关内容

焦点与点击事件

看一下view有关点击事件的源码

```
if ((mPrivateFlags & PFLAG_PRESSED) != 0 || prepressed) {
    // take focus if we don't have it already and we should in
    // touch mode.
    boolean focusTaken = false;
    if (isFocusable() && isFocusableInTouchMode() && !isFocused()) {
        focusTaken = requestFocus();
    }

    if (prepressed) {
        // The button is being released before we actually
        // showed it as pressed.  Make it show the pressed
        // state now (before scheduling the click) to ensure
        // the user sees it.
        setPressed(true, x, y);
    }

    if (!mHasPerformedLongPress && !mIgnoreNextUpEvent) {
        // This is a tap, so remove the longpress check
        removeLongPressCallback();

        // Only perform take click actions if we were in the pressed state
        if (!focusTaken) {
            // Use a Runnable and post this rather than calling
            // performClick directly. This lets other visual state
            // of the view update before click actions start.
            if (mPerformClick == null) {
                mPerformClick = new PerformClick();
            }
```

```
            if (!post(mPerformClick)) {
                performClickInternal();
            }
        }
    }
```

在view接收到点击事件之后，首先会判断这里的view是否需要焦点，如果需要焦点（`focusable = "true"`），则首先给view申请焦点，这一次的点击事件也就不会响应点击事件，反之如果设置为false，才会在第一次点击响应点击事件

onCreate()

回调时间

> 这是一个必须执行的回调，当Activity`第一次`被创建时，将会回调这个方法，在一个活动的生命周期中只会执行一次
>
> Tip: Android界面的横竖屏切换将会导致Activity被销毁并重新创建

需要做的事情

> 进行一些初始化操作，比如实例化对象，绑定`ViewModel`这种

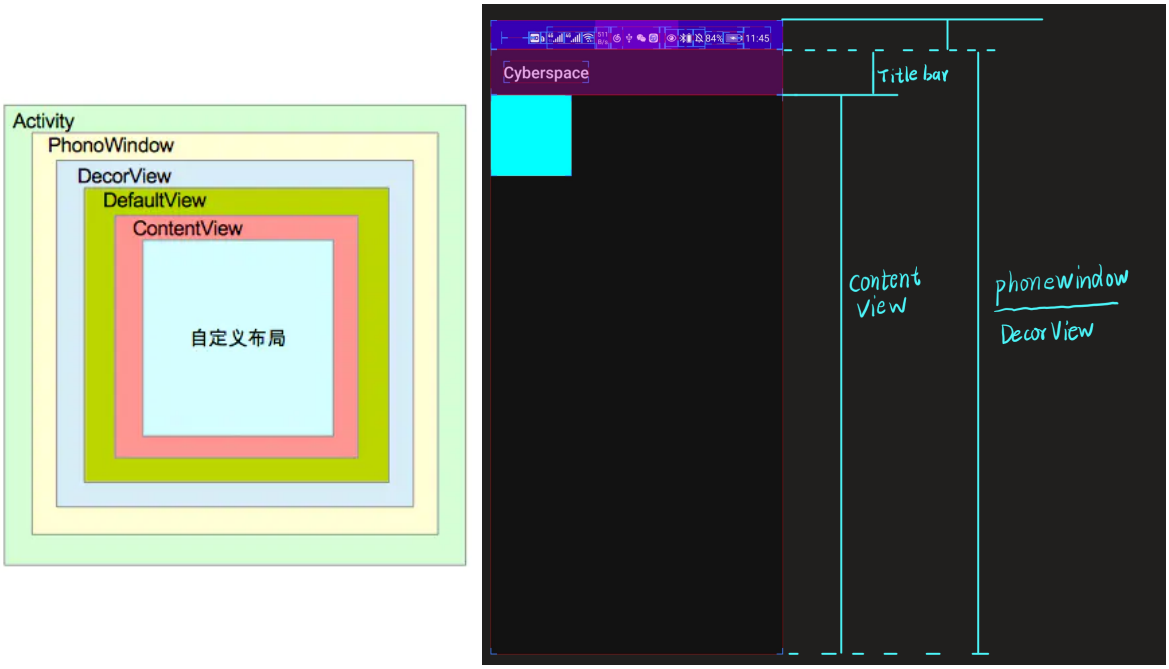
setContentView()

```
@Override
public void setContentView(@LayoutRes int layoutResID) {
    getDelegate().setContentView(layoutResID);
}

@Override
public void setContentView(View view) {
    getDelegate().setContentView(view);
}

@Override
public void setContentView(View view, ViewGroup.LayoutParams params) {
    getDelegate().setContentView(view, params);
}
```

`setContentView`的目的在于将所指的layout id或者view加载到Activity之中，比如常见用法

```
setContentView(R.layout.activity_main);
```

如果想要说清楚这个方法具体做了什么，需要先理解一下Activity的view层级结构



活动的view并不是只有简单的一层，或者说，我们在xml文件里面写的布局并不是Activity布局的全部，它只是其中`ContentView`层级中的内容，而`setContentView`就是要加载`contentView`之中的view

> 这里稍微提一句题外话，`findViewById()`是如何找view的呢？为什么用`findViewById()`查询与本活动没有建立关系的id会出错呢？在activity中直接调用的`findViewById()`会最终委托到`DecorView`去最终执行，`DecorView`本质上是一个`ViewGroup`，之后会调用这个`ViewGroup`的`findViewTraversal`方法来遍历自己或者自己的子view去寻找到对应id的view，如果找到就将返回，所以`DecorView`之下的view（`ToolBar`/`ContentView`及其子View）都可以通过`findViewById`进行加载。

onStart()

回调时间

> 当活动进入**已开始**的状态的时候将会调用

此时的Activity不可交互，**也不可见**，尚未来到**前台**（有关前台后台的内容会在后面提及）

尽量不要在`onStart()`函数回调的时候做一些耗时操作，如果活动长期停滞在这个不可交互的状态，体验将会很离谱

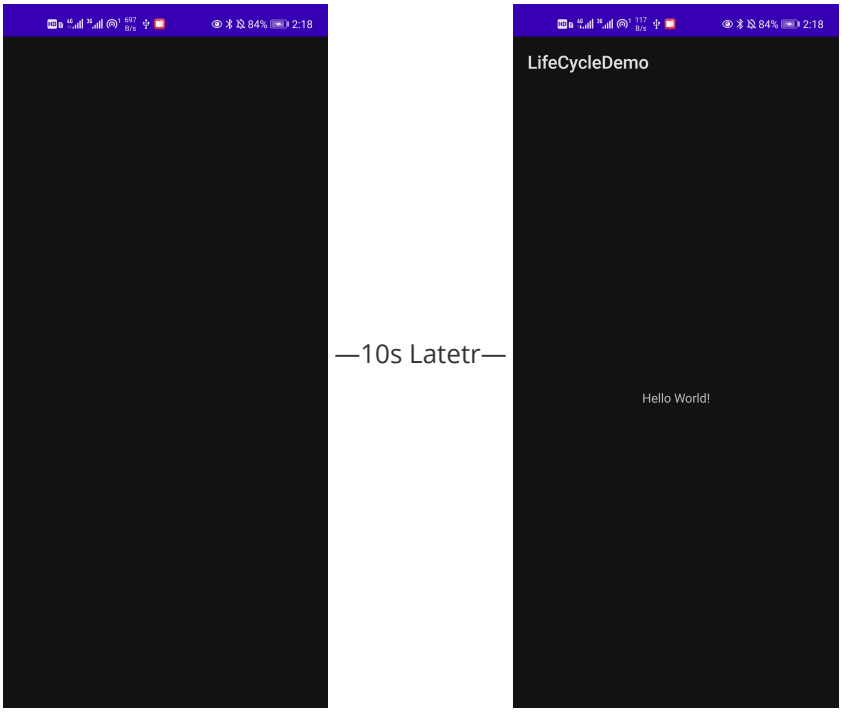
Tip：一些博客中可能会提到`onStart()`会使Activity**可见**，个人怀疑这是对google开发文档这句话的误解

> 当 Activity 进入“已开始”状态时，系统会调用此回调。`onStart()` 调用使 Activity 对用户可见，因为应用会为 Activity 进入前台并支持互动做准备。例如，应用通过此方法来初始化维护界面的代码。

需要明确的是，在执行`onStart()`的瞬间，活动还是**不可见**的，可以写一个小的demo证明一下

```
package com.example.lifecycledemo;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.util.Log;
import android.view.View;

public class MainActivity extends AppCompatActivity {
    final String TAG = getClass().getName();
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        Log.d(TAG, "onCreate");
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    protected void onStart() {
        super.onStart();
        try {
            Thread.sleep(10000);//强行给onStart函数一个10s的昏迷
        } catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}
```



结果显然可知

onResume()

回调时间

> Activity 会在进入“已恢复”状态时来到前台，然后系统调用 `onResume()` 回调，从最开始的那一张Activity生命周期图可以看到触发从后台来到前台的途径有：
>
> - `onCreate()` 路线：Activity本身不存在或者已经被销毁（`onFinish()`）导致活动需要被重新构建，之后回调到`onResume`，具体的例子可以是：
>   - 活动第一次被创建
>   - 在调用栈中存在，但是更高优先级的活动需要内存，栈中处于后台的Activity就会被销毁，而再次调用到就需要重新绘制
>
> More: 什么样的栈中活动需要被销毁？详情参考后面有关前台、后台、内存回收的介绍

> - `onStart()`路线：因活动本身不可见（`onStop()`）而导致需要被重新变为可见状态。除了上面`onCreate()`中的情景以外，还有以下情况可以引起：
>   - 被新的Activity覆盖，没有从活动栈中弹出，之后再返回活动
>   - 按下home键返回Launcher界面（应用菜单界面），再从Launcher热启动App
> - `onPause()`路线：因活动本身不再处于前台（`onPause()`）而导致需要重新回到前台，比如活动中出现了一个弹窗，弹窗被清除之后将会回调到这里
>
> 如果当前Activity丢失**焦点**，就会离开这个状态，有以下两点需要注意
>
> - 在 Android 7.0（API 级别 24）或更高版本中，有多个应用在多窗口模式下运行。无论何时，都只有一个应用（窗口）可以拥有焦点，因此系统会暂停所有其他应用。（Tip: 说是这么说，但是我在Huawei上测试了一下，并不会丢失焦点）
> - 有新的半透明 Activity（例如对话框）处于开启状态。只要 Activity 仍然部分可见但并未处于焦点之中，它便会一直暂停。

就像前面提到的一样，`onResume()`的时候应用已经来到了前台，而`onStart()`时应用还在后台

可以做的事情

主要是一些恢复工作，在pause状态（比如说Activity被切换到了后台）时为了缓解内存压力，可以释放掉一些对象，这些被释放掉的对象就可以在`onResume()`函数中恢复（后面会具体提到）

Tip：此时的Activity依然处于不可见状态，可以使用`onStart()`测试时的硬核阻断线程方法测试一下，Activity真正可见的时间是在`onResume()`回调执行完之后

onPause()

回调时间

> 如上文所说，如果当前应用丢失焦点，离开前台，就会导致处于暂停状态

需要做的事情

如果活动已经被暂停，证明活动已经处于后台，这时候应该释放一些暂时用不到的资源，比如一些系统资源，来减缓系统的内存压力，需要注意以下几点：

- 释放掉的资源记得在`onResume()`里面恢复过来
- 不要啥都释放掉，比如你写的是一个视频软件，我想分屏看视频，结果你在软件的`onPause()`中将视频给释放掉了，我还看个🔨，更推荐在`onStop()`函数中大量释放资源（换句话说，onPause的时候应用还是可见的，不要吧还有必要需求的视觉元素释放掉）
- **不**推荐在`onPause()`函数中进行耗时操作，比如保存或者存储用户数据，或者进行网络请求等操作，因为新的活动的`onResume()`必须要等到前一个活动的`onPause()`执行完毕之后才能调用，将当前活动阻塞在`onPause()`将会影响用户体验（长时间没有可交互的应用在前台）

onStop()

回调时间

> 当Activity已经对用户不可见的时候调用，这个时候Activity已经处于后台，被新的Activity完全遮盖或者说App被切到了后台

需要做到事情

`onStop()`回调中可以进行一些阶段性的"总结"工作，更新本地数据，释放无用资源

- 阶段性更新数据：如果你无法找到更合适的时机来将信息保存到数据库，可以在`onStop()`期间执行此操作

- 释放无用资源：应释放或调整在应用对用户不可见时的无用资源。例如，应用可以暂停动画效果，或从精确位置更新切换到粗略位置更新。反正都不可见了，也没这么多要求

当应用进入已停止状态时，Activity对象仍然存在，其内持有的一些成员变量依旧存在，Activity恢复之后，Activity会重新调用这些信息，除非你将这些资源释放掉了

总而言之，如果你有一个无敌大的内存，啥都可以不做，系统会帮你存着，但是如果要进行一些内存管理，建议通过`onStop()`函数处理掉一些在后台用户用不到的对象来降低内存压力，但代价是要在`onResume()`中恢复被释放掉的数据。

onDestory()

当当前Activity被finish()掉，用户完全关闭Activity，都将引起最终回调到这里

除了以上理由之外，屏幕旋转（系统配置改变）也会导致`onDestory`函数被调用并触发重绘

> 换个角度思考，屏幕旋转导致Activity被销毁的原因是包裹Activity的外部容器的宽高发生了变化，最顶层viewGroup的宽高改变之后，其他的子view也需要重新layout和重新绘制，这个过程看起来很顺其自然。也许我们可以得出结论：Activity外部容器的宽高改变会引起整个Activity的重绘。到这里为止都是自然而言的，但是Activity的重绘一定会引起Activity的销毁和重新生成吗?
>
> 答案是否定的，通过在manifest中配置`android:configChanges`属性，可以在Activity中通过`onConfigurationChanged()`来监听屏幕的状态，并且不会引起Activity的重绘
>
> 那么除了屏幕旋转之外，还有没有会导致外部容器大小变化的操作呢？其实还有，比如分屏模式下改变上下Activity的屏占比

回调时间

> 销毁 Ativity 之前，系统会先调用 `onDestroy()`

Activity被系统回收与恢复

现实是残酷的，一般的开发环境可供使用的内存是有限的，当内存告急，需要GC（Garbage Collection，垃圾回收）的时候，就需要释放掉一些Activity来解放内存。

GC何时触发

Android触发gc的条件与java类似，涉及一些dvm的相关知识，如果拎出来也能讲一节课．有兴趣的同学可以参考:

Android内存管理分析总结

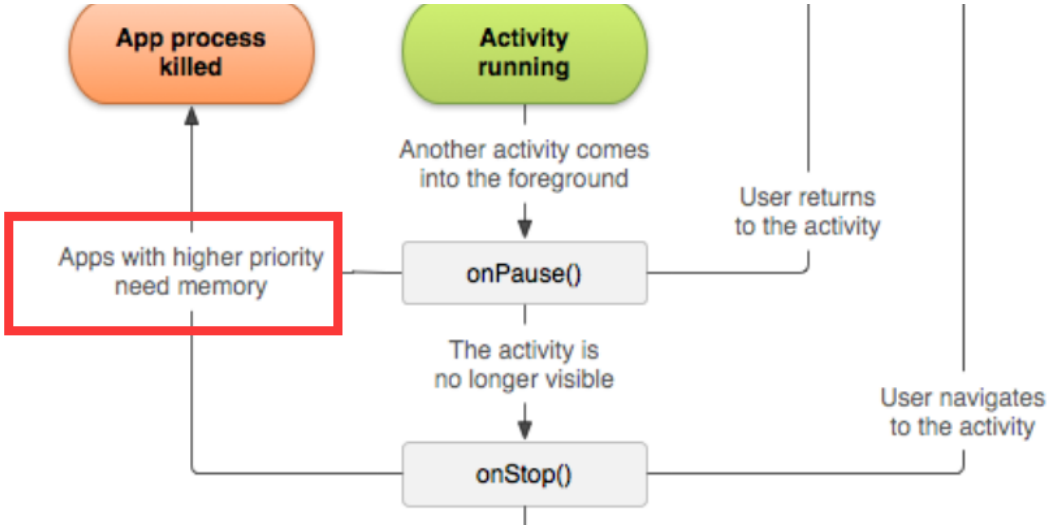
要销毁谁

首先来看Google开发文档上的一句话

> 系统永远不会直接终止 Activity 以释放内存，而是会终止 Activity 所在的进程。系统不仅会销毁 Activity，还会销毁在该进程中运行的所有其他内容。如需了解如何在系统启动的进程被终止时保留和恢复 Activity 的界面状态，请参阅保存和恢复 Activity 状态。

这句话极度精髓，曾一度看到怀疑人生。

还记得最上面的那张生命周期图吗，如果你仔细看，可以看到这个内容



如果你在第一层：

> 秒懂，原来当内存不足的时候已经pause掉或者stop掉的活动会被系统杀掉

如果你在第二层：

> 这里提到的process，应该是进程，加上上文提到的系统只会杀死进程来回收Activity，所以说这里的回收应该上升到App的层面，毕竟是进程级别的概念

如果你在第三层：

> 同一个App内的Activity可能会处于不同进程，所以这个回收既可以是App层面的概念，也可以是App之中的概念

总结一下，当内存告急的时候，系统需要通过杀死进程来终结活动以释放内存，一般而言，这些被杀死的进程是和当前前台App独立的进程，也就是其他后台进程，但是如果一个App内有多个进程，就也可能会出现销毁同一个App内的不同进程的情况

这样讨论的层级就上升了，什么样的进程（可暂且等同于App）才会被销毁呢？下面是Google给的一张表格

系统终止进程的可能性	进程状态	Activity 状态
较小	前台（拥有或即将获得焦点）	已创建 已开始 已恢复
较大	后台（失去焦点）	已暂停
最大	后台（不可见）	已停止
空	已销毁	

需要注意的是，进程杀死释`Activity`不会回调任何生命周期，但是会调用`onSaveInstance()`方法，这个方法也正是我们拯救世界的希望。

恢复

onSaveInstanceState() & onRestoreInstanceState()

只有在Activity**可能被系统回收**的时候才有可能调用`onSaveInstanceState()`方法，如果是直接finish()或者按下BACK键，显然**不属于**会被判定为可能被系统回收的情况，也就不会调用。这里有一个例外，就是屏幕旋转，从概念上看屏幕旋转似乎并不属于Activity被回收，但系统还是会调用`onSaveInstanceState()`和`onRestoreInstanceState()`来缓存Activity的状态

相反，如果是被新的Activity覆盖，息屏或者按下Home键，以及上文中提到的Activity的旋转，都将调用到`onSaveInstance()`

常规的调用顺序为`onPause()`—>`onSaveInstanceState()`—>`onStop()`，（硬说其实只能确定调用在`onStop()`之前，并不能确定与`onPause()`的前后关系）

在`onSaveInstanceState()`方法中，系统可以自动帮我们做一些Activity的状态存储，并在`onRestoreInstanceState()`中协助恢复一些Activity的相关信息，比如：`EditText`的文本信息，`ListView`的滚动位置，每一个view都有`onSaveInstanceState()`和`onRestoreInstanceState()`两个方法，可以查看源码来了解系统帮助我们做了什么。

`onRestoreInstanceState()`的调用时间则是只有在系统将Activity异常回收了之后才会调用，调用时间在`onCreate()`之后。比如一个Activity切换到后台，如果它没有被系统回收，就不会执行`onRestoreInstanceState()`方法，而如果被回收了就会执行

优雅的使用生命周期：androidx.lifecycle

`lifecycle`是jetpack推出的生命周期感知组件，可以用于响应另一个组件的生命周期（Activity或Fragment）

0x01 Service

寒假里面不少同学做了网抑云音乐播放器🐻🌧，应该提前对Service有过了解，这里俺还是简述一下。

先来看看Google对Service的描述

> `Service` 是一种可在后台执行长时间运行操作而不提供界面的应用组件。服务可由其他应用组件启动，而且即使用户切换到其他应用，服务仍将在后台继续运行。此外，组件可通过绑定到服务与之进行交互，甚至是执行进程间通信 (IPC)。例如，服务可在后台处理网络事务、播放音乐，执行文件 I/O 或与内容提供程序进行交互。

提取一下关键词：Service可以在后台**长时间**运行，即使用户切换到其他应用，服务仍然可在**后台**运行。服务是可以同时和**多个组件**绑定的（这里的组件可以理解为Activity BroadcastReceiver等）

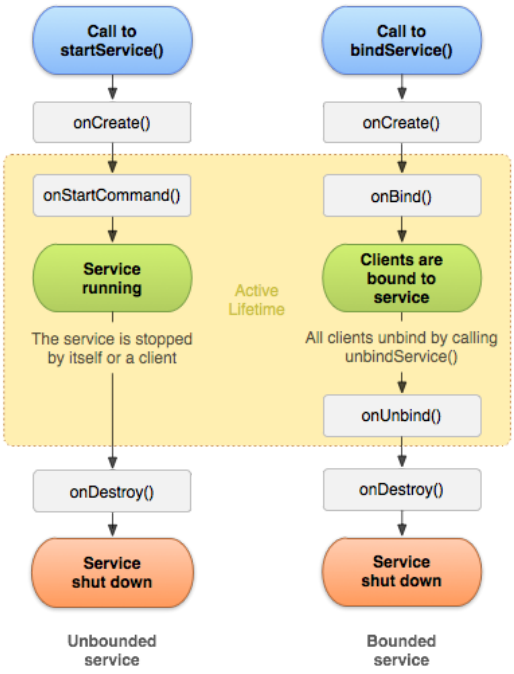
线程进程特性

服务既**不会**创建自己的线程，也**不会**创建属于自己的进程（当然可以用标签令行指定），也就是说建立的Service仍然会运行在当前App进程的主线程中，如果要执行运算密集操作或者延迟操作，需要另行开辟子线程。

> 既然运行在主线程里面，那这个东西有啥用呢？如果我想要用它去网络请求或者进行I/O操作，还得另开线程，为啥不直接在Activity里面执行呢？
>
> 其实如果单纯的去看上面说的下载功能，的确差异不是很大。举个例子，比如我在Activity里面开辟了一个线程去执行下载数据的任务，实现Activity与线程之间通信的桥梁是Handler，就算是Activity被切到了后台，Handler的引用依旧在，Activity的实例也在，线程也还在，依旧可以实现功能。如果用Service去做的话优势在于更好的保活性，进程中有Service在的话不容易被系统回收掉，更容易在后台活下来。

Service的生命周期

还是先看看Google官方文档的图



这里可以看到，通过`startService()`和`bindService()`两种方法启动的服务的生命周期是有些不同的下面分开理解一下

- `startService()`：通过这种方法开启的的服务姑且称之为**启动服务**，一般尝试启动服务的是Activity或者Fragment等组件，当然，后面会提到的BroadcastReceiver也可以启动服务。
    - `onCreate()`：通过**启动**方式开启的服务将会先执行回调，这个回调可以类比Activity的`onCreate()`方法，除非Service已经被destroy掉，不然的话`onCreate()`方法只会执行一次，如果要执行一些一次性的配置操作，可以在此方法中执行，因为只会执行一次，即只会在第一次启动时回调到这个方法。
    - `onStartCommand()`：当其他组件想要启动服务的时候，会调用`onStartCommand()`方法，与`onCreate()`不同，`onStartCommand()`会在每一次的启动之后调用，1w次启动将会调用1w次`onStartCommand()`，但是只要Service没有被销毁，就只会调用一次`onCreate()`
    - `onDestory()`：当某一个组件调用`stopService()`或者服务自己调用`stopSelf()`的时候，系统就会尽快销毁服务，并回调到`onDestory()`方法，因为一个服务可以被多个组件启动，所以`stopSelf()`方法可以支持传递一个int进来，即`stopSelf(int id)`，就可以与`onStartCommand()`方法传递进来的startId对照，如果stopSelf传递进来的id与最后一个启动服务的startId相同，则可以终止服务。
- `bindService()`：通过这种方式开启的服务姑且称之为**绑定服务**，尝试绑定服务的组件和上面的类似，其生命周期状态如下
    - `onCreate()`：同上，只有创建服务时才会调用，即只会在第一次尝试绑定服务时才会回调到这里，除非服务被销毁
    - `onBind()`：当有组件尝试通过`bindService()`的方法启动Service时会调用到这里，这同时也是Service抽象类里面唯一的抽象方法，也就是继承Service必须要实现的方法，需要返回一个IBinder对象，这里大家先了解一下调用时间就好，具体用法后面会详细介绍
    - `onUnbind()`：当**所有**与该Service绑定的组件都与该Service解除了绑定，则会回调到此方法。**需要注意的是，如果与该服务绑定的组件被销毁了，将会自动执行解绑顶操作。**但是如果是通过startService的方式启动，就算启动者被销毁了，服务还是可以正常运作下去。
    - `onDestory()`：当`onUnbind()`执行过后，几乎紧接着就会执行调用到这里，**需要注意的是，如果使用绑定的方法启动Service，如果尚未解绑，通过`stopService()`无法将其停止**

接下来举一个例子来说明如何创建一个Service用以播放音乐

如何创建Service

首先，你得有一个类，让它去继承`Service`这个抽象类，并实现其中的抽象方法`onBind()`

```
public class MusicService extends Service {

    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
}
```

作为四大组件之一的Service要想正式使用还需要再AndroidManifest里面注册一下，这点不要忘记了

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.servicedemo">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.ServiceDemo">
        <activity android:name=".ui.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <!--    在这里注册了Service-->
        <service android:name=".service.MusicService"/>
    </application>

</manifest>
```

这里就来介绍一下`onBind()`方法的作用

熟悉IPC（跨进程通信）的同学可能会比较了解Binder这个东西，如上文所述，在**绑定服务**时会回调到`onBind()`方法，而`onBind()`方法就是服务与请求绑定的组件之间进行通信的核心。

比如我可以这样写

```
public class MusicService extends Service {

    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return new MyBinder();//将内部类实例返回出去
    }

    //service的方法
    private void serviceFun(){
```

```
    }

    class MyBinder extends Binder{
        public MusicService getService(){
            //可以获取到外部类Service的实例
            return MusicService.this;
        }

        public void outFun(){
            //通过Binder的公共方法将外部类Service的私有方法暴露出去
            serviceFun();
        }
    }
}
```

这样的话就解决了组件以及与组件绑定的Service之间的通信问题，为了方便描述，这里就假设是Activity A绑定了Service B，如果要实现A与B之间的通信，最好的方法就是让A持有B的引用，B可以执行A的回调，Binder就是做到了这件事情，你可以利用Binder让Activity A拿到Service B的引用，进而可以在A中调用B的方法，并在A中实现B中给定的回调，实现二者之间的通信。这里充分运用了内部类的美妙之处，虽然Activity A拿到的是Service B的内部类Binder，但是因为内部类持有了外部类Service的引用，所以和拿到Service B本体没啥区别，并且也可以在Binder中直接拿到Service B的实例。

> 为了直观一些，这里还是举一个例子。
>
> 林潼姐姐（Activity）最喜欢上了一个小姐姐（Service），但是两人都不认识，没有办法“通信”，但是又不能直接要到小姐姐的联系方式，林潼姐姐就很痛苦，但是后来她发现张涛姐姐和这个小姐姐是室友，就想到了让张涛姐姐代替自己完成和小姐姐通信的主意，比如想要知道小姐姐的生日，就去找涛姐姐去帮自己问（Activity A获取Service B的信息），如果小姐姐要去上自习，涛姐姐就回调给林潼姐姐让林潼姐姐去Gank（Service B对Activity A的回调），等时机成熟了涛姐姐就直接介绍两个人认识（利用Binder获取Service的实例），两者就可以随便“通信”了。

接下来我们想一下，这个服务需要实现的功能是播放音乐，对外暴露的方法应该有那些呢？这里简单起见就提供播放和暂停两个方法吧，于是我们稍微完善一下代码

```
/**
 * Author: RayleighZ
 * Time: 2021-04-08 19:35
 */
public class MusicService extends Service {

    private MediaPlayer mediaPlayer;

    @Override
    public void onCreate() {
        super.onCreate();
        //初始化音乐播放器
        mediaPlayer = MediaPlayer.create(this, R.raw.pink_floyd);
    }

    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return new MyBinder();
    }

    public void playMusic() {
```

```
        mediaPlayer.start();
    }

    public void pauseMusic() {
        mediaPlayer.pause();
    }

    public class MyBinder extends Binder {
        public MusicService getService() {
            return MusicService.this;
        }
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        //释放mediaPlayer
        mediaPlayer.release();
    }
}
```

再Service中，我提供了开始和暂停音乐两个方法，这两个方法将会暴露给Activity，用于控制音乐的播放。

接下来看看如何实现Service的启动吧

如何启动Service

从上面我们已经知道，Service的启动分为启动（`startService()`）和绑定（`bindService()`）两种方法，其中，`startService()`是无法做到对Service的控制的，启动之后，二者基本上就没啥关系了，除了能再操控销毁。但是`bindService()`则可以通过Binder机制，实现绑定组件和Service的实时通信，考虑到音乐播放需要做到对Service的控制，所以Activity启动Service的方法应该选择为`bindService()`。

bindService(Intent service, ServiceConnection conn, int flags)

活动中绑定Service的核心代码如下

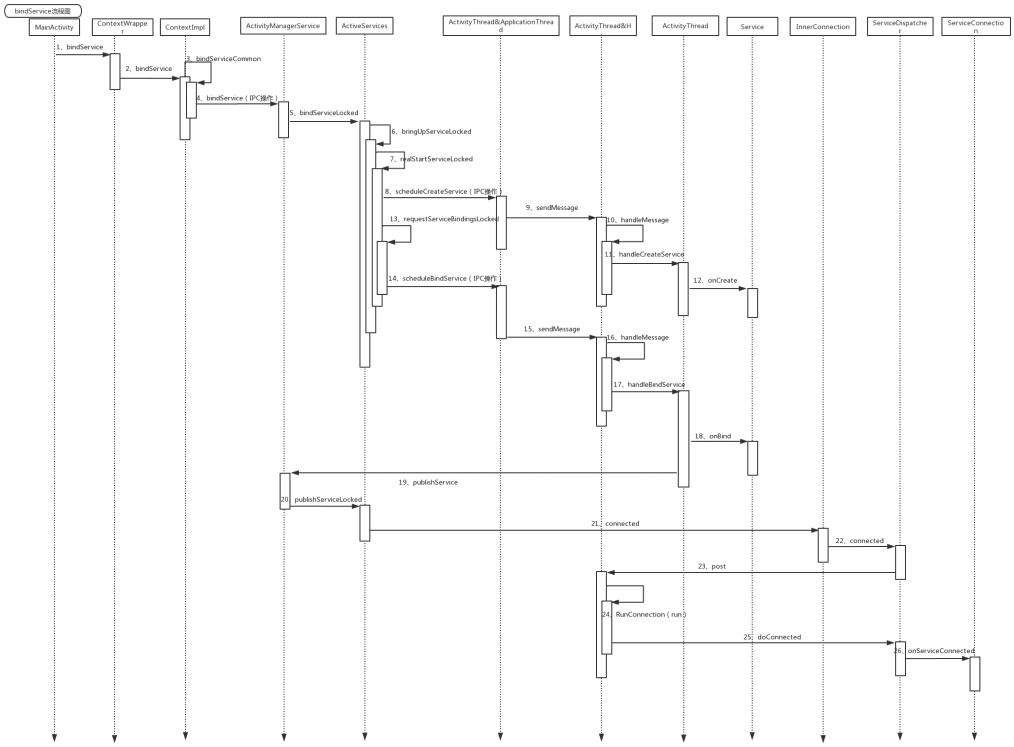
```
bindService(
    new Intent(this, MusicService.class),
    connection,
    Context.BIND_AUTO_CREATE
);
```

`bindService()`方法总共接收三个参数

- Intent service：指向目标Service的Intent，这里可以类比启动Activity的Intent的写法
- ServiceConnection conn：`ServiceConnection`接口的具体实现类，方便Service在绑定成功和**意外断开绑定**之后通知回调，也正是在这个接口中可以拿到Binder，我这里用了一个匿名内部类的思路写了一下

```
private final ServiceConnection connection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        MusicService.MyBinder myBinder = (MusicService.MyBinder) service;
        musicService = myBinder.getService();
    }

    @Override
    public void onServiceDisconnected(ComponentName name) {

    }
};
```

就可以轻松的在`onServiceConnected`接口中拿到Binder，进而拿到Service

> 这个接口具体在何时被回调的呢?
>
> 原本准备在这里带大家康康源码，但是感觉还是总结一下看源码得出的结论重要些，关于这里的源码解析，这篇博客还是蛮不错的，推荐一下：Android bindService源码解析
>
> 下面是整体的调用栈
>
> 
>
>
> 这里用自然语言简述一下bindService的执行过程
>
> Activity中`bindService()`方法的执行者其实是contextWrapper，在contextWrapper中则调用到了Context的具体实现类ContextImpl的bindService()方法，也是从这里开始，bindService才正式开始执行起来
>
> 在ContextImpl的`bindService()`方法中，首先会检查是否是系统进程尝试绑定服务，如果是系统进程绑定服务，则会抛出警告
>
> > 为啥，我理解的是因为系统进程将长期存在，而如果Service一直被一个生命周期过长的进程绑定着，正如前文所说，因为仍然处在绑定状态，所以Service将很难被回收，会容易造成内存泄漏

从宏观上说，接下来的要做的事情是将ServiceConnection转变为ServiceDispatcher.InnerConnection，为啥呢？首先明确一下ServiceConnection的作用，他是用来在绑定完成之后通知Activity去做事情的，而Service的开辟需要涉及到AMS（ActivityManagerService），AMS与应用处在不同下进程，需要进行跨进程通信，那就不能简单的使用ServiceConnection了，ServiceDispatcher.InnerConnection是一个Binder的具体实现类，可以实现跨进程通信，所以才需要进行这一次绑定通信。

接下来就是要通知AMS去新建Service了，AMS会通过ActiveServices去通知Service的启动（ActiveServices是辅助AMS管理Service的工具类），在通过新一轮的IPC操作通知ApplicationThread去真正启动Service，并用Message机制通知ActivityThread去回调声明周期以及回调ServiceConnection。这里逻辑简单，可以带着看一下源码

```
public final void scheduleCreateService(IBinder token,
                                        ServiceInfo info,
CompatibilityInfo compatInfo, int processState) {
    updateProcessState(processState, false);
    CreateServiceData s = new CreateServiceData();
    s.token = token;
    s.info = info;
    s.compatInfo = compatInfo;
    //这里在通知启动Service，康康咋handle的
    sendMessage(H.CREATE_SERVICE, s);
}
```

```
public void handleMessage(Message msg) {
    if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " +
codeToString(msg.what));
    switch (msg.what) {
        .......
        case CREATE_SERVICE:
            if (Trace.isTagEnabled(Trace.TRACE_TAG_ACTIVITY_MANAGER)) {
                Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER,
                        ("serviceCreate: " + String.valueOf(msg.obj)));
            }
            handleCreateService((CreateServiceData)msg.obj);
            Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
            break;
        .......
}
```

`handleCreateService()`是下一个我们锁定的方法，来看看

```
@UnsupportedAppUsage
private void handleCreateService(CreateServiceData data) {
    // If we are getting ready to gc after going to the background,
well
    // we are back active so skip it.
    unscheduleGcIdler();

    LoadedApk packageInfo = getPackageInfoNoCheck(
            data.info.applicationInfo, data.compatInfo);
    Service service = null;
    try {

        ........
```

```
        //正式生成Service
        service = packageInfo.getAppFactory()
                .instantiateService(cl, data.info.name, data.intent);

        ........

        //通知生命周期回调
        service.onCreate();
    } catch (Exception e) {
        if (!mInstrumentation.onException(service, e)) {
            throw new RuntimeException(
                    "Unable to create service " + data.info.name
                            + ": " + e.toString(), e);
        }
    }
}
```

同理再康康bindService是如何handle的

```
private void handleBindService(BindServiceData data) {
    Service s = mServices.get(data.token);
    if (DEBUG_SERVICE)
        Slog.v(TAG, "handleBindService s=" + s + " rebind=" +
data.rebind);
    if (s != null) {
        try {
            data.intent.setExtrasClassLoader(s.getClassLoader());
            data.intent.prepareToEnterProcess();
            try {
                if (!data.rebind) {
                    //如果不是二次绑定，则回调onBind
                    IBinder binder = s.onBind(data.intent);
                    ActivityManager.getService().publishService(
                            data.token, data.intent, binder);
                } else {
                    //如果已经绑定过，则回调reBind
                    s.onRebind(data.intent);
                    ActivityManager.getService().serviceDoneExecuting(
                            data.token, SERVICE_DONE_EXECUTING_ANON, 0,
0);
                }
            } catch (RemoteException ex) {
                throw ex.rethrowFromSystemServer();
            }
        } catch (Exception e) {
            if (!mInstrumentation.onException(s, e)) {
                throw new RuntimeException(
                        "Unable to bind to service " + s
                                + " with " + data.intent + ": " +
e.toString(), e);
            }
        }
    }
}
```

> 总结一下，再回调onBind或者onRebind之后将会通知AMS生命周期回调完成，接下来AMS就会回调ServiceConnection的诸多生命周期了。
>
> 即onCreate -> onBind/onRebind -> onServiceConnected

- flags：声明了绑定Service的一些配置，具体配置如下

> BIND_ABOVE_CLIENT
> 如果当绑定服务期间遇到OOM需要杀死进程，客户进程会先于服务进程被杀死。
>
> BIND_ADJUST_WITH_ACTIVITY
> 允许客户进程提升被绑定服务进程的优先级
>
> BIND_ALLOW_OOM_MANAGEMENT
> 如果绑定服务期间遇到OOM需要杀死进程，被绑定的服务进程会被OOM列入猎杀对象中。
>
> BIND_AUTO_CREATE
> 若绑定服务时服务未启动，则会自动启动服务。 **注意，这种情况下服务的onStartCommand仍然未被调用（它只会在显式调用startService时才会被调用）。**
>
> BIND_DEBUG_UNBIND
> 使用此标志绑定服务之后的unBindService方法会无效。 **这种方法会引起内存泄露，只能在调试时使用。**
>
> BIND_IMPORTANT
> 被绑定的服务进程优先级会被提到FOREGROUND级别
>
> BIND_NO_FOREGROUND
> 被绑定的服务进程优先级不允许被提到FOREGROUND级别
>
> BIND_WAIVE_PRIORITY
> 被绑定的服务进程不会被OOM列入猎杀对象中。

看一下完整的启动Service的Activity的代码吧

```
public class MainActivity extends AppCompatActivity {

    private MusicService musicService;
    private boolean isPlaying = false;

    private final ServiceConnection connection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            MusicService.MyBinder myBinder = (MusicService.MyBinder) service;
            musicService = myBinder.getService();
        }

        @Override
        public void onServiceDisconnected(ComponentName name) {

        }
    };

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

```
                new Intent(this, MusicService.class),
                connection,
                Context.BIND_AUTO_CREATE
        );
        findViewById(R.id.btn_play).setOnClickListener(
                v -> {
                    if (musicService != null){
                        if (isPlaying){
                            musicService.pauseMusic();
                            v.setBackgroundResource(R.drawable.ic_play);
                        } else {
                            musicService.playMusic();
                            v.setBackgroundResource(R.drawable.ic_pause);
                        }
                        isPlaying = !isPlaying;
                    }
                }
        );
    }
}
```

如果不需要和Service通信的话，可以使用startService方法启动Service

startService(Intent service)

因为完全不需要通信，只需要单方面通知启动Service就好了，所以这里只用单纯的传递一个intent进来。在Activity中无法做到和Service的及时通信。

两者配合使用

我们总结一下上面两种启动方法的特点

- bind：Service可以和组件通信，方便实现对Service的控制，但是组件一旦被销毁，就会解除和Service的绑定，进而导致Service被终结。

  > 试想，如果你写的是一个音乐播放器，Activity 1绑定了Service并启动了音乐播放，结果跳到Activity 2并关闭了Activity 1 Service就被终结了，这样何其可笑。

- start：Service的生命周期不直接和启动者挂钩，就算前者没了，Service依旧可以在后台运行，可惜不能通信

为了实现Service既不和启动者的生命周期直接挂钩，又可以实现对Service的控制，就可以两者同时使用，先start再bind，进而实现以上效果。

如何终结Service

stopService(Intent name)

最常用的外部通知终止服务的方式，用于外部通知结束Service，但是要注意，一个Service是可以被多个组件启动的，当对应到多个onStartCommand的时候，需要在意一下其他组件的感受，确保其他组件的需要已经被完成了之后再终止Service。

> 正如前文所说，如果当前Service仍然与其他组件绑定着，通过外部调用stopService将无法达到终结Service的作用，需要先将所有与Service绑定的组件解除与Service的绑定，之后才能调用stopService。

stopSelf(int startId)

服务自我终结的方式，有无参数和含参数两种形式，后者适用于上文中提到的存在多个组件启动Service的情况，需要传递一个onStartCommand时候传递进来的startId进去，如果在调用stopSelf之前有新的onStartCommand到达，则startId就会不匹配，就不能终止Service，防止出现其他Service的任务还未结束就将Service终结的情况

前台服务

前台服务是Android 8之后的新概念，Android 9之后启动前台服务需要申请FOREGROUND_SERVICE权限

何谓前台服务

前台服务是用户主动意识到的一种服务，因此在内存不足时，系统也不会考虑将其终止。前台服务必须为状态栏提供通知，将其放在运行中的标题下方。这意味着除非将服务停止或从前台移除，否则不能清除该通知。使用前台服务可以做到Service的有效保活

如何启动前台服务

需要使用startForeground(int id, Notification notification)方法来启动前台服务，这里需要接收两个参数

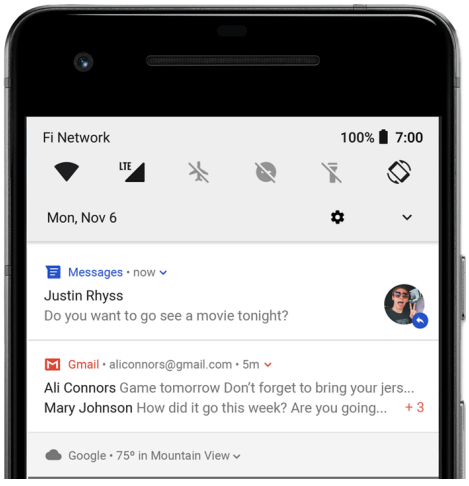
- notification：前台服务必须要绑定一个notification才能启动，因为必须要建立到状态栏的通知，具体方法会在Notification部分介绍
- id：notification的标识，不可为0，后面也会介绍。

这里算是一个承上启下，接下来就来看看Notification吧。

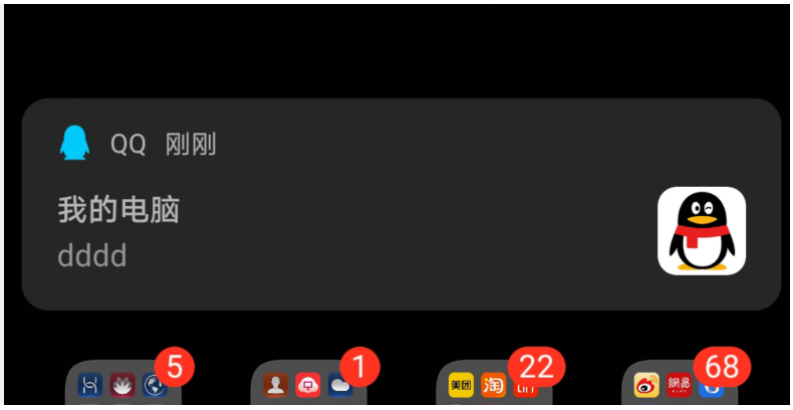
Notification

何谓通知？下面就是几种典型的通知

- 状态栏和抽屉式通知栏

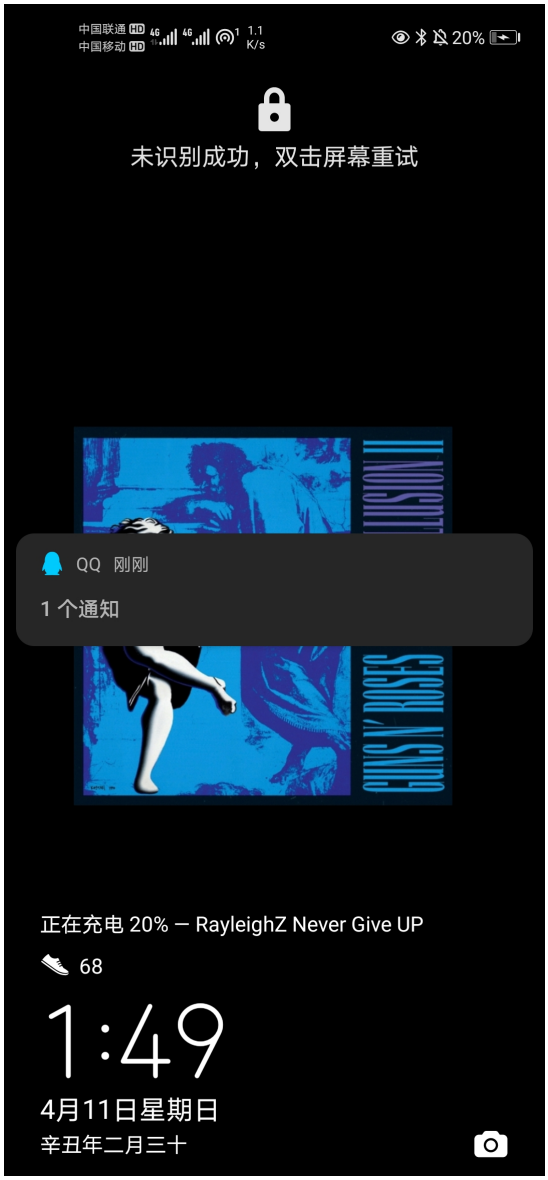


- 提醒式通知



提醒式通知自Android 5.0之后开始出现，提醒式通知会在应用发出通知后立即出现，稍后便会消失，但仍照常显示在抽屉式通知栏中。如果用户处于全屏模式下或者是通知（或者渠道）的重要程度比较高，就会采取这种提醒式通知

- 锁定屏幕通知



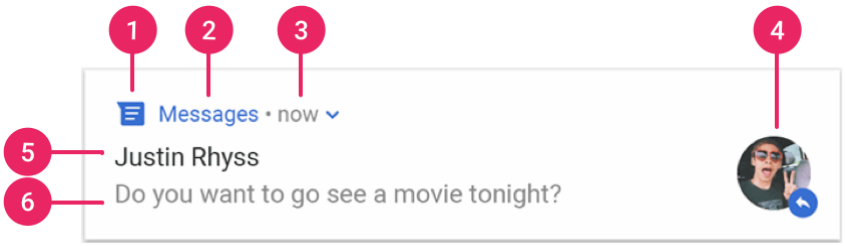
Android 5.0之后的机制，允许在锁屏界面展示一些通知

通知算是系统提供给用户以不点击就可以获取到App提供的信息的一种方式（除此之外还有桌面小工具和长按菜单栏），通过通知可以实现简单的数据呈现和交互，接下来看看咋实现一个简单通知。

因为我想和上面Service的内容联系起来，这里就以实现一个简单的通知栏控制音乐播放和暂停作为例子来实现一个Notification

一个简单通知的构成

以下内容参考自Google官方开发文档



1. 小图标：**必须提供**，通过 `setSmallIcon()` 进行设置。也就是在状态栏上面展示的通知图标
2. 应用名称：由系统提供。
3. 时间戳：由系统提供，但您可以使用 `setWhen()` 替换它或者使用 `setShowWhen(false)` 隐藏它。
4. 大图标：可选内容（通常仅用于联系人照片，请勿将其用于应用图标），通过 `setLargeIcon()` 进行设置。
5. 标题：可选内容，通过 `setContentTitle()` 进行设置。
6. 文本：可选内容，通过 `setContentText()` 进行设置。

版本变化

抄自简书（🐶 is 😁）

> - Android 4.1（API 级别 16）
>
> 引入了展开式通知模板（称为通知样式），可以提供较大的通知内容区域来显示信息。用户可以使用单指向上/向下滑动的手势来展开通知。
>
> - Android 5.0（API 级别 21）
>
> 引入了锁定屏幕和浮动通知。
> 向 API 集添加了通知是否在锁定屏幕上显示的方法 (setVisibility())，以及指定通知文本的“公开”版本的方法。
> 添加了 setPriority() 方法，告知系统该通知应具有的“干扰性”（例如，将其设置为“高”，可使该通知以浮动通知的形式显示）。
>
> - Android 7.0（API 级别 24）
>
> 用户可以使用内联回复直接在通知内回复（用户可以输入文本，然后将其发送给通知的父级应用）。
>
> - Android 8.0（API 级别 26）
>
> **现在必须将单个通知放入特定渠道中。**
> 用户现在可以按渠道关闭通知，而不是关闭应用的所有通知。
> 包含活动通知的应用会在应用图标上方显示通知“标志”。（小圆点或数字）
> 用户可以暂停抽屉式通知栏中的通知。您可以为通知设置自动超时。
> 可以设置通知的背景颜色。

构建渠道

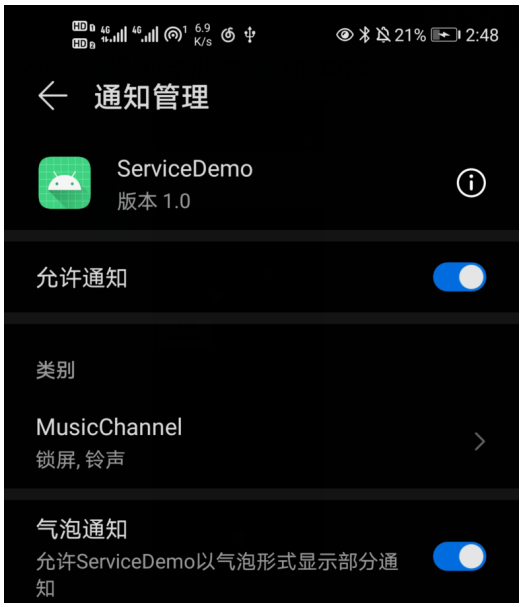
假如目前Android版本高于8，我们就要建立上文中提到的渠道了，来看看咋写

```
NotificationManager manager = (NotificationManager)
getSystemService(NOTIFICATION_SERVICE);
NotificationChannel channel = new NotificationChannel(id, name, level);
channel.setDescription("这是音乐播放控制的渠道");
manager.createNotificationChannel(channel);
```

这里的NotificationManager相当于是系统发送、清除通知的管理者，channel的构建自然也需要manager，这里构建Notification需要三个参数，这里分别解析一下

```
NotificationChannel(String id, CharSequence name, @Importance int importance)
```

- String id：渠道的id，必须是应用内唯一的。
- CharSequence name：渠道的名称，可以在设置中看到



- int importance：渠道的重要程度，具体对应关系如下（摘自官方文档）

用户可见的重要性级别	重要性（Android 8.0 及更高版本）	优先级（Android 7.1 及更低版本）
紧急：发出提示音，并以浮动通知的形式显示	`IMPORTANCE_HIGH`	`PRIORITY_HIGH` 或 `PRIORITY_MAX`
高：发出提示音	`IMPORTANCE_DEFAULT`	`PRIORITY_DEFAULT`
中：无提示音	`IMPORTANCE_LOW`	`PRIORITY_LOW`
低：无提示音，且不会在状态栏中显示。	`IMPORTANCE_MIN`	`PRIORITY_MIN`

此外还可以设置描述，这个是可选项。

如果要删除渠道的话可以采用notificationManager.deleteNotificationChannel(id)函数。

构建通知

通知的构建需要使用 `NotificationCompat.Builder` 对象，接下来看看咋用这个东西写个通知

```
NotificationCompat.Builder builder = new NotificationCompat.Builder(this,
channelId)
    .setSmallIcon(R.drawable.ic_rayleighz_logo_wite)     //设置通知图标
    .setContentText("Wish you were here")    //设置通知详细内容
    .setContentTitle("Pink Floyd")  //设置通知标题
    .setPriority(NotificationCompat.PRIORITY_DEFAULT);  //设置通知优先级
```

简单的链式调用，进行了一些常用配置。

> 8.0以上版本渠道重要程度将决定该渠道的重要程度，在没有渠道的低版本中，通知的重要程度完全取决于通知自己设定的重要程度

发送一条通知

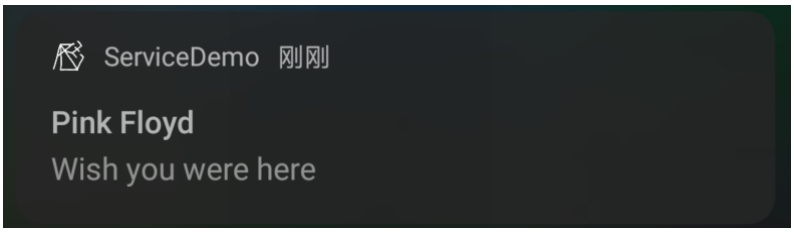
发送的话，只需要notificationManager.notify(notification)就可以了

这里贴一下完整的Activity代码

```
public class MainActivity extends AppCompatActivity {

    private MusicService musicService;
    private boolean isPlaying = false;
    private NotificationCompat.Builder builder;
    private NotificationChannel channel;
    private NotificationManager manager;

    private final ServiceConnection connection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            MusicService.MyBinder myBinder = (MusicService.MyBinder) service;
            musicService = myBinder.getService();
        }

        @Override
        public void onServiceDisconnected(ComponentName name) {

        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        String channelId = "1";
        int notificationId = 1;
        initView();
        initService();
        initNotificationBuilder(channelId);
        initNotificationChanel(channelId, "MusicChannel",
NotificationCompat.PRIORITY_DEFAULT);
        manager.notify(notificationId, builder.build());
    }

    private void initView(){
        findViewById(R.id.btn_play).setOnClickListener(
                v -> {
                    if (musicService != null){
                        if (isPlaying){
                            musicService.pauseMusic();
                            v.setBackgroundResource(R.drawable.ic_play);
                        } else {
                            musicService.playMusic();
                            v.setBackgroundResource(R.drawable.ic_pause);
                        }
                        isPlaying = !isPlaying;
                    }
                }
```

```
        );
    }

    private void initService(){
        bindService(
                new Intent(this, MusicService.class),
                connection,
                Context.BIND_AUTO_CREATE
        );
    }

    private void initNotificationBuilder(String channelId){
        builder = new NotificationCompat.Builder(this, channelId)
                .setSmallIcon(R.drawable.ic_rayleighz_logo_wite)
                .setContentText("Wish you were here")
                .setContentTitle("Pink Floyd")
                .setPriority(NotificationCompat.PRIORITY_DEFAULT);
    }

    private void initNotificationChanel(String id, String name, int level){
        manager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O ){
            channel = new NotificationChannel(id, name, level);
            manager.createNotificationChannel(channel);
        }
    }
}
```

效果如下



接下来玩点有意思的，同时也引出下一个重要组件：BroadcastReceiver

配合RemoteView以及点击事件的Notification

这种不可交互的通知不是我们想要的那种效果，前面提到了，我们要的效果是通过通知控制音乐播放，要想达到上述效果，就需要实现以下两点，1、拥有一个自定义的界面。2、设置点击事件，接下来就介绍一下这种通知咋写。

先解决视图的问题

RemoteView的强大

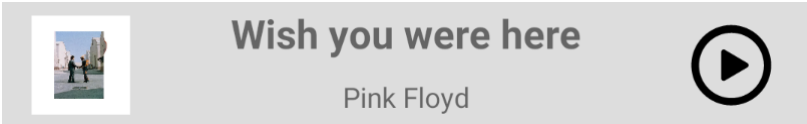
RemoteView从名字看即为远程的View，RemoteViews表示一个View结构，它可以在其他进程中显示，为了能够更新它的界面，RemoteViews提供了一组基础的操作用于**跨进程**更新它的界面。这种感觉就像是开疆拓土者，离开了故土（本身的App），前往了其他进程去展示（nrt🐱），是不是听起来有一种很炫酷的感觉？掌邮的课表小部件，网易云的状态栏音乐控制都是RemoteView所为

从xml开始

RemoteView只支持几种特定的View/ViewGroup，并不是啥都能往上面堆，并且收起后的视图布局的高度上限为 64 dp，展开后的视图布局的高度上限为 256 dp。这些是不能超过的

> 下面是支持的类型
>
> - Layout
>   FrameLayout LineraLayout RelativeLayout GridLayout
> - View
>   AnalogClock Button Chronometer ImageButton ImageView ProgressBar TextView ViewFlipper ListView GridView StackView AdapterViewFlipper ViewStub

手撸一个简单的页面



接下来就是要配置给通知并发送出去了，但是呢，目前的知识栈还不太够，需要补充一下BroadcastReceiver的相关知识。

0x02 BroadcastReceiver

学通信的同学对广播二字一定充满热忱，可能会回调起学信号或者单片机的痛苦时光，如果还没学，希望下次老师讲到广播的时候你可以回忆起Android的四大组件里面也有一个类似的玩意儿。

何为BroadcastReceiver？跟他的名字一样，广播的接收者，啥是广播呢？其实和现实生活中的广播相当类似，比如我在群里面发一条消息，就相当于发送了一条广播，这条广播的受众是全体群员，每一个群员都可以理解为是一个BroadcastReceiver，你可以选择对我发送的广播进行处理，或者当作没看见，广播和广播接收者这种通信方式酷似现实生活中的通信，茫茫电磁波中，总有那么一个波型是你想要的。（扯远了）

看看咋写吧

静态注册一个BroadcastReceiver

四大组件的东西，不在AndroidManifest里面注册一下都对不起他的身份

```
        <receiver android:name=".receiver.MusicOperateReceiver">
            <intent-filter>
                <action android:name="com.servicedemo.operate_music"/>
            </intent-filter>
        </receiver>
```

这里的MusicOperateReceiver为我写的一个继承了BroadcastReceiver的工具类，此外还需要配置的是一个intent过滤器，只有action为com.servicedemo.operate_music的intent才会被这个BroadcastReceiver拦截，这样就不会导致啥广播都拦截，只有有用的广播才会触发回调

> 在实际通信中也是这样，总不能广播中的所有信号都接收吧，很多信息对自己而言是没有作用的

剩下的操作需要在MusicOperateReceiver里面执行，看看代码

```
public class MusicOperateReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        switch (intent.getAction()){
```

```
            case "com.servicedemo.play_music": {
                //播放音乐
                break;
            }
            case "com.servicedemo.pause_music": {
                //暂停音乐
                break;
            }
        }
    }
}
```

在onReceive中可以接收到拦截的Intent，这里就对action进行判断，并执行对应的操作就行了

静态注册的BroadcastReceiver不需要App启动就可以运行，因而可以实现一些开机启动功能。

动态注册一个BroadcastReceiver

Q：为啥还需要动态注册?

A：不谈啥动态增加可以接收的广播，动态广播接收者可以是一个类的内部类，可以更方便的实现通信，这就已经很诱人了

代码如下，可以随便调用Service的内部方法

```
IntentFilter intentFilter = new IntentFilter("com.servicedemo.operate_music");
    registerReceiver(new BroadcastReceiver() {
        @Override
            public void onReceive(Context context, Intent intent) {
            if (isPlaying){
                pauseMusic();
            } else {
                playMusic();
            }
    }
}, intentFilter);
```

显然如果要和Service交互的话，还是动态注册更合适。

现在接收者有了，如何让它投入使用呢？我们继续上面的通知栏的例子。

给RemoteView添加点击事件

PendingIntent

设置点击事件需要使用到PendingIntent，啥是PendingIntent呢?

PendingIntent是一个不会立刻执行的intent，它只会在特定条件达成后执行

PendingIntent获取有三种方式:通过Activity,Service,BroadcastReceiver获取.

> 1. 可以通过getActivity(Context context, int requestCode, Intent intent, int flags)系列方法从系统取得一个**用于启动一个Activity**的PendingIntent对象.
> 2. 可以通过getService(Context context, int requestCode, Intent intent, int flags)方法从系统取得一个 **用于启动一个Service**的 PendingIntent对象.
> 3. 可以通过getBroadcast(Context context, int requestCode, Intent intent, int flags)方法从系统取得一个**用于向BroadcastReceiver的发送广播**的PendingIntent对象

我们要用的显然是第三个，先简单介绍一下所需参数

- int requestCode：请求码，也是判断PendingIntent是否相同的标志（存疑）
- Intent intent：传递的Intent，也就是广播接收者接收到的那条Intent
- int flags：一些关键配置，具体常量如下

> **FLAG_CANCEL_CURRENT**:如果当前系统中已经存在一个相同的PendingIntent对象，那么就将先将已有的PendingIntent取消，然后重新生成一个PendingIntent对象。
>
> **FLAG_NO_CREATE**:如果当前系统中不存在相同的PendingIntent对象，系统将不会创建该PendingIntent对象而是直接返回null。
>
> **FLAG_ONE_SHOT**:该PendingIntent只作用一次。在该PendingIntent对象通过send()方法触发过后，PendingIntent将自动调用cancel()进行销毁，那么如果你再调用send()方法的话，系统将会返回一个SendIntentException。
>
> **FLAG_UPDATE_CURRENT**:如果系统中有一个和你描述的PendingIntent对等的PendingInent，那么系统将使用该PendingIntent对象，但是会使用新的Intent来更新之前PendingIntent中的Intent对象数据，例如更新Intent中的Extras。

好，接收者的逻辑代码就可以如下去写。

```
    @Nullable
    @Override
    public IBinder onBind(Intent intent) {

        IntentFilter intentFilter = new
IntentFilter("com.servicedemo.operate_music");
        registerReceiver(new BroadcastReceiver() {
            @Override
            public void onReceive(Context context, Intent intent) {
                if (isPlaying){
                    pauseMusic();
                } else {
                    playMusic();
                }
            }
        }, intentFilter);
        return new MyBinder();
    }
```

remoteView这边就这样配置

```
//设置RemoteView里的控件监听
int notificationId = 1;
Intent pauseIntent = new Intent();
pauseIntent.setAction("com.servicedemo.operate_music");
PendingIntent pendingIntent = PendingIntent.getBroadcast(MainActivity.this, 0,
pauseIntent, PendingIntent.FLAG_CANCEL_CURRENT);
remoteViews.setOnClickPendingIntent(R.id.btn_remote_view_operate,
pendingIntent);
builder.setContent(remoteViews);
Notification notification = builder.build();
manager.notify(notificationId, notification);
```

就能实现简单的statusBar控制音乐播放了

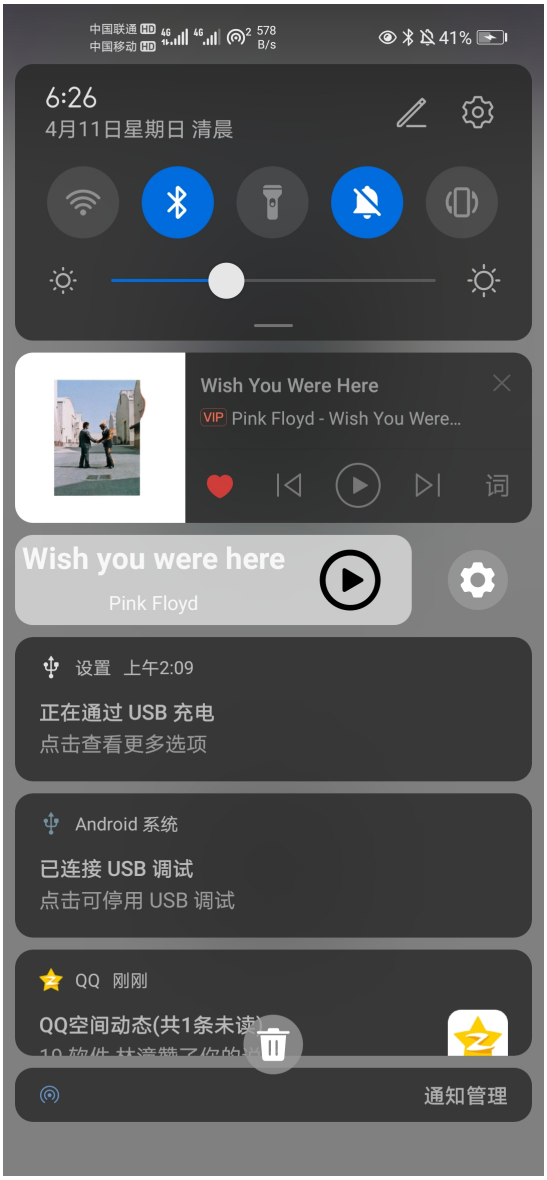
回归前台服务

现在大家就已经具备了启动前台服务的技术栈，回归到刚才去看看如何启动前台服务吧

启动前台服务可以理解为系统帮助我们实现了Channel，只需要将对应的Notification传递进去就可以了

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    startForeground(notificationId, notification);
    return super.onStartCommand(intent, flags, startId);
}
```

如上，你就启动了一个前台服务。这个与它绑定的Notification是不会被简单删除掉的，除非Service本身被杀掉



0x03 ContentProvider

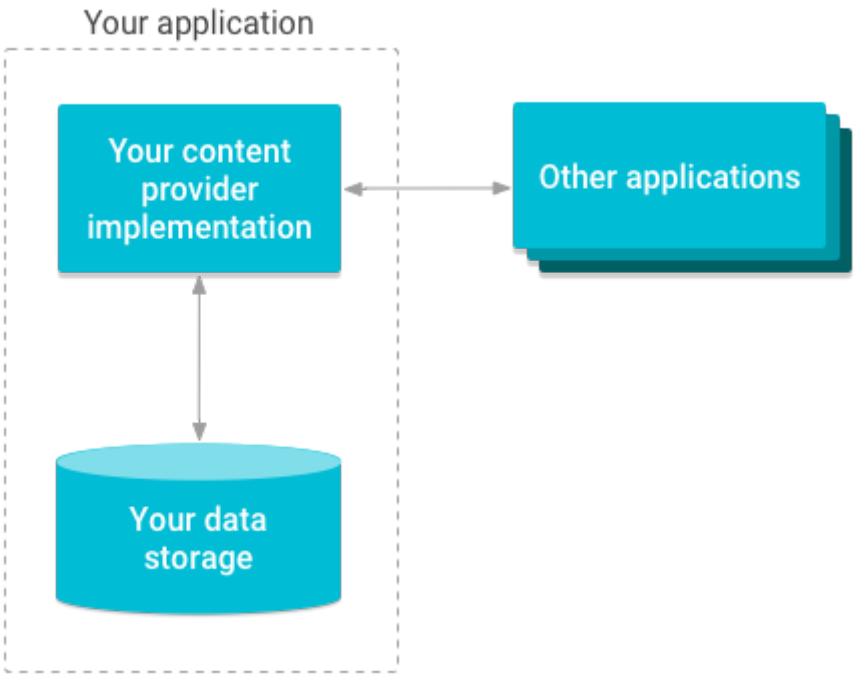
看看郭神书里面是咋对ContentProvider进行描述的

> ContentProvider主要用于在不同的应用程序之间实现数据共享的功能，它提供了一套完整的机制，允许一个程序访问另一个程序中的数据，同时还能保证被访问数据的安全性。目前，使用ContentProvider是Android实现跨程序共享数据的标准方式。

说白了，内容提供器主要在两方面发挥作用

- 获取其他进程的数据
- 将自己的数据提供给其他进程

ContentProvider其中也蕴含了接口的思想，它规范了不同进程之间的数据读取规范，可以安全的访问修改他人的数据以及被访问修改数据


Your application
Your content provider implementation
Other applications
Your data storage