Android知识总结

**Self Introduction**

I'm a graduate student from Lehigh University in Pennsylvania and focus on mobile developement. I did my internship at touchpal which provides a softkeyboard app for mobile devices especially for android. The downloads in google play is about one hundred million. It’s widely used in all over the world. I had experience working closely with different teams and very familiar with the full life cycle of developing an App.

­

I worked in a team where we have about ten people. We have a team leader, product manager, QA, UI designer and sereral engineers. We start our work with a small meeting in the morning. An agile platform will help us track everyone’s mission. Everyone is clear about what we are doing and what we will do. And after that, we will start work. The pace of our team is fast, we released a new version of our app once a week. It’s really quick I think it inherit the culture of our early-stage company. Most of us get off work at 10pm, Sometime 12pm before the day we release a new version. But it’s not mandantory, you know, most of the engineers are bachelors, We got nothing to do after work. So we choose to stay at company to learn and improve ourselves. Some engineers who get married leave off work on time every day. But to be honest, I really learned a lot of things at that time and know learned to collerbrate with other team members. It also gave me some experience working in a professional environment, so I’m confident that I can quickly adapt any other start up company.

I want to talk about my recent project: document classification system. It’s a machine learning project and we need to classify the types of document. You know some files involves with some operations of customer like borrowing money or investment which represent the intention of them and the objective of these operations can be called the types. and The interesting thing is we need to deploy the model for online prediction. So, we use python to build our server and provide API for user’s query. Then we design a website and an android app to provide user interface. This project involves many skills like front-end, back-end, machine learning and mobile development. So, I think that is an interesting project.

Why yelp:

Well, the postion I applied for is Android Engineer. Actually, I’m fan for android. You can see from my resume; all of my project is about android and I want to be an android engineer in the future. And I do really like the Groupon APP. When I buy something I always use Groupon to find if there has discount. The design of the app is consice and easy to use. I really like the style of groupon. So I hope to join an android team.

**Challenge**

Most of the challenge is about technical challenges, but I don’t want to talk too much about that because as an engineer, it’s your duty to solve all the technical problems. You can google it or get help from senior engineer. What I want to say is beyond the technical things. I can hardly say that is a challenge, but it’s the thing I want to share.

I want to talk about one of my experience when I was an intern. One day, my product manager told me that some users complain about they don’t have enough emojia to use. You know as a keyboard app, We provide some emojis for users. And I say: what? Is there anyone who really need to so many emoji to use. I mean I like use emojis too, but I just use some of them. And our app have already provided nearly hundreds of emojis. So, I think it’s hard to understand. Is there anyone who really needs so many emoji? And my product manager took me to see the comments in google play. And I said: yeah, there are really some guys who enjoying using much emojis. So, I add more emojis for users. We add emojis according to the Unicode version in the past. And I changed it from Unicode version to android version. If you want to use more emojis, you need to update your android version. So, some conflicts like that often happened in our team. I can not understand the requirement of pm to let me add more emojis because I don’t often use emoji. Maybe I don’t like the UI because it doesn’t look great. Maybe Product manger ask me to implement some features which hard to understand. However, I want to say: every members of team have their own advantages, product manager knows what users think and what they like. UI designer know how to make a beautiful User interface. However, every people in our team have their own roles. Product manage know what people thinks of our app because he may spend lots of time to study what on earth users like. UI designer spends several years studing on designing perfect UI. When you have conficts, you need to learn to collobrate with others, follow the opinions from the people who is really good at what he is doing. We need to respect others’ job. Of cource, you can also give your own suggestions. For example, Product manger want you to add some small functions, but actually, as an engineer, you know sometime, it’s really hard to implement some functions that may looks very easy. In this case, you can tell the pm what you think. So I believe every member in a team need to learn how to trust others and fulfill their own duties. You can complain but you also need to respect the people who are really good at his job. Anyway, if you really insist on what you think. You can tell the team leader and he will make decisions because that is his responsibility and make our team united.

**Android Project**

As you can see, this is skin store for making your key board looks cool. I know most of people don’t care about the keyboard and just use default keyboard, but some people really like it.

* Structured the code with ***MVP*** and injected fragments and network API by ***Dagger*** to decouple the dependency.

The reason we use MVP here is because we want to separate the project into different functional parts. According to the Java design princinple SOLID. The first pricinple is Single responsibility, one class should have one and only one reasonablility. That’s why we use MVP here. And we also inject fragments and network API to decouple the dependency. And actually I only use dagger to inject some simple component because dagger is much more complicated than other library. It uses lots of annotations and hide some concrete creatation of some components like fragment. So when you read the code with dagger, you will find it’s hard to understand it if you are a fresh man or it’s the first time to read it. And annotation is implemented by reflection. So we use dagger at a cost of performance.

* Implemented themes search page with ***RxJava*** & ***Retrofit*** and increased the page load speed by 17%.

The search page is my favourite part I finished during my internship period. (the search page is you type some words, it will give you some hints,and when you click it, it will show you some results). In this part I used Rxjava to offload search work to background and uedd retrofit to send request to our sever; Rxjava is one of my favorite libraries in android. I’ve been study rxJava for a long time. It’s an implementation of reactive programming. Reactive programing is program with aysncronous data stream. You are able to create data streams of anything, not just from click. anything can be a stream: variables, user inputs, properties. There has a data source which will emit events. We capture these emitted events only asynchronously, by defining a function that will execute when event is emitted. We did the same things in Rxjava, we create Obsereble to be a data souce send a onSubscribe, onSubscribe is actually the event, it is a interface and defines what we will do when we capture the event. Rxjava also use Observer pattern and builder pattern to make the code more elegant. The most powerful part in Rxjava is the transformer, it will capture the event and do somethings and return a new event. So we can do lots of things with these transoformer. And another advantage is the switch between different threads. So we can easily do asycronous work. I learn a lot of things from reading the source code of Rxjava. Retrofit is great. Actually, it’s not the one who do the network job. You can take retrofit as a wrapper, It uses delegate pattern and hold another library called OkHttp. This library solves many network problem and use Chain of Responsibility Pattern to deal with the network things like add header. We mainly used there two library to implement this part. But I also did something to optimize this part. The first thing is you know, we have to offload the heavy seach job to the background or it will run on main thread and cause ANR problem. The second things is reducing the layer of layout. In android studio we have a tool called layout inspector and I used it analize the layout. Actually, most of the layout problem result from using too much nest layout like a liner…. And here I just simply use constaint layout. You don’t have to create too many nested layout. There still have another techinic I don’t use at that time. I learned in this year. It’s called AsyncLayoutInflater. When we get the response from user, we render this info to the view. So, only after we got the response we can create the view. So why don’t we create the view before we get the resonse?

* Sent and received messages between components by ***EventBus*** to show hot search keys and update Json info.

Well, I use Eventbus because I don’t like callback. There many drawbacks of callback like it may lead to memory leak. Callback is hard to understand and trace. If the callback contrain another, the code will become a mess. Callback also make your code not elegant. When you read code which use rxjava to dosome ayschronous work, you must be able to find the code is really long.

* Built unit test by ***Junit*** and ***Mockito*** for themes search page and increased the test speed.
* Implemented UI and designed a ***custom view*** to show checked effect.

I think it is a basic for every android engineer to know how to draw a view. I desined many views by myselt. In this page I implement a check effect. And on the second project of my resume I implement a siderbar view for quick navigation. And on the third project android new APP. I implement a PIE view to show download process. Basically, there have three steps for drawing a view: measure, layout, draw. But in most cases, we only need to overide onDraw method. When I learn recyclerview, I custom my own layoutmanager and create a different layout. It’s kind of like a table layout. You just need to put every item one by one. By the way, I really meet a problem with recycler view here. The check effect is when you click one of the item, I will label it with a tick to show that this image is choosed.There are only one image can be choosed at a time, when we click another one, we need to show this effect at new one and cancel the effect of the old one. There have two ways to implement it. When you click one image like the orange one, then we find the parent view of this image and enumerate its children to label new one and cancel old one. Another way is to refresh the dataset and set the tick again. Actually, the first one is wrong because we can not enumerate all the children, some of children are out of the screen. And we also need to be very careful about the cache of recycler view. One of the bug I met is when I click one of the image and scroll down until we can not see the choosen one, then we may see an image with a tick even we don’t choose it. The reason of that is the recycler view automatically recycle the view with tick and use this view to hold a new item. I solve this problem first by set a indicator to indicate which item is choosed. And my mentor told me this method is not elegant. The best way to solve it is creating a new class with the image and a variable to indicate weather this image is choosed or not. I think it’s right because if it allow more than one image to choose, The first method will create lots of indicator and that will hard to manage.

* Overrode ***Dispatch Event*** to avoid wrong touch event between ViewPager and ViewGroup*.*

You can see the seekbar at the right of the small eye. It’ used to adjust the transparency of these image. It’s a little complicated here. For example, if you have several fragments and you can put them in a viewpager. You can scroll right or left to go to another fragment. But one of the fragment contains another viewpager which contains some images and used to show a carousel effect. So when you scroll at the image carousel. According to the touch event principles in android. The touch event will be passed from the top to bottom and recieved the event from bottom to top. In the example, the image carousel will receive the touch event. So you can only go to another image. What if you want to go to another page when you scroll at the image carousel, you can write a interceptor in the fragment view to intercept event and avoid it to pass the event to carousel. In my page, this more complicated, when you scroll from the margin of the seek bar, this page will also be scrolled. I used a linear layout to contain the eye and seekbar. When we touch the margin of seekbar that means we touch the linear layout and the event will not be received until it meet the viewpage.

* Added ***State Recovery*** to avoid crash result from low system memory.

Yes, we really need it. Here is the scence I meet, In a activity, I open a fragment. And in this fragment I open another activity. And when I go back, I got a crash. The reason of that is because I want to get somethings from activity and I use getActivity in the fragment, When the activity is killed by system, the fragment can not get the activity, so crash happens. Actually, we don’t have to use getActivity every time. We can get all the things we need when the fragment created. And if crash happens, we store them and restore it when it restart. I mock the low system memory by using strick mode. You know, in your android phones, you can open strict mode in the deloper options. The system will kill activity as soon as you leave it.

**Android Resuming Download Demo**

**1. What is MVP and why?**

MVP stands for Model-View-Presenter which is a UI design pattern. MVP pattern promotes separation of concern, each component just does its own thing. In Android, we have a problem arising from the fact that Android activities are closely coupled to both UI and data access mechanisms.  For an application to be easily extensible and maintainable, we need to define well-separated layers.

**2. What is Context, When not to use getApplicationContext?**

As the name suggests, it’s the context of the current state of the application/object. It lets newly-created objects understand what has been going on. Typically you call it to get information regarding another part of your program. And also, Context is a handle to the system, it provides services like resolving resources, obtaining access to databases and preferences, and so on.

It’s not a complete Context, supporting everything that Activity does. Various things you will try to do with this Context will fail, mostly related to the GUI.

It can create memory leaks if the Context from getApplicationContext() holds onto something created by your calls on it that you don’t clean up. With an Activity, if it holds onto something, once the Activity gets garbage collected, everything else flushes out too. The Applicationobject remains for the lifetime of your process.

**3.Four Components**

An *activity* is the entry point for interacting with the user. It represents a single screen with a user interface.

A *service* is a general-purpose entry point for keeping an app running in the background for all kinds of reasons.

A *broadcast receiver* is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements.

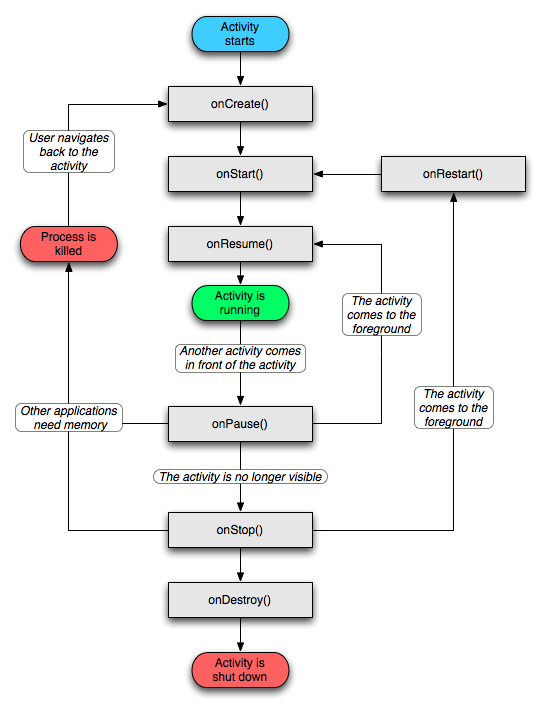
A *content provider* manages a shared set of app data that you can store in the file system, in a SQLite database, on the web, or on any other persistent storage location that your app can access. Through the content provider, other apps can query or modify the data if the content provider allows it.

# 常用命令

查看当前运行的activity： adb shell dumpsys activity activities

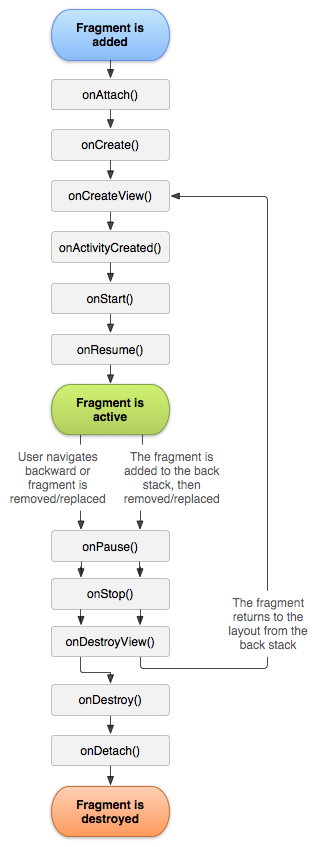
**Activity生命周期**

**先来看张官方说明图**



**Fragment**

## 生命周期



超详细版：https://github.com/xxv/android-lifecycle

**fragment生命周期解析**

* 当一个fragment被创建的时候：  
  onAttach()  
  onCreate()  
  onCreateView()  
  onActivityCreated()
* 当这个fragment对用户可见的时候，它会经历以下状态。  
  onStart()  
  onResume()

1.2可以理解为从创建到显示（或切换）

* 当这个fragment进入“后台模式”的时候，它会经历以下状态。  
  onPause()  
  onStop()
* 当这个fragment被销毁了（或者持有它的activity被销毁了）：  
  onPause()  
  onStop()  
  onDestroyView()  
  onDestroy()  
  onDetach()
* 就像Activity一样，在以下的状态中，可以使用Bundle对象保存一个fragment的对象。  
  onCreate()  
  onCreateView()  
  onActivityCreated()

**其他场景的调用**

* 屏幕灭掉  
  onPause() onSaveInstanceState() onStop()
* 屏幕解锁  
  onStart() onResume()
* 切换到其他Fragment  
  onPause() onStop() onDestroyView()
* 切换回本身的Fragment  
  onCreateView() onActivityCreated() onStart() onResume()
* 回到桌面  
  onPause() onSaveInstanceState() onStop()
* 回到应用  
  onStart() onResume()
* 退出应用  
  onPause() onStop() onDestroyView() onDestroy() onDetach()

## 关于Fragment的一些常见问题

### 在创建Fragment的时候传递参数

**介绍：**

Fragment在被由内存导致的销毁后重新创建时会调用默认的不含参数的构造函数，另外fragment必须创建默认构造函数，如果只包含带参数的，则会报no default construtor的错。

**方法：**

通过bundle储存

**public static** ContentFragment newInstance(String argument)  
{  
 Bundle bundle = **new** Bundle();  
 bundle.putString(ARGUMENT, argument);  
 ContentFragment contentFragment = **new** ContentFragment();  
 contentFragment.setArguments(bundle);  
 **return** contentFragment;  
}

### Fragment的通信方式(GetActivity)

**介绍：**

fragment获取activity的内容

可能你遇到过getActivity()返回null，或者平时运行完好的代码，在“内存重启”之后，调用getActivity()的地方却返回null，报了空指针异常。

大多数情况下的原因：你在调用了getActivity()时，当前的Fragment已经onDetach()了宿主Activity。  
比如：你在pop了Fragment之后，该Fragment的异步任务仍然在执行，并且在执行完成后调用了getActivity()方法，这样就会空指针。

解决办法：  
更"安全"的方法：(对于Fragment已经onDetach这种情况，我们应该避免在这之后再去调用宿主Activity对象，比如取消这些异步任务，但我们的团队可能会有粗心大意的情况，所以下面给出的这个方案会保证安全)

在Fragment基类里设置一个Activity mActivity的全局变量，在onAttach(Activity activity)里赋值，使用mActivity代替getActivity()，保证Fragment即使在onDetach后，仍持有Activity的引用（有引起内存泄露的风险，但是异步任务没停止的情况下，本身就可能已内存泄漏，相比Crash，这种做法“安全”些），即：

**方法：**

创建接口，利用监听

### Fragment 重叠问题

**介绍：**

前面我们介绍 Fragment 初始化时提到 Activity 销毁重建的问题，试想一下，当 Activity 重新执行 onCreate() 方法时，是不是会再次执行 Fragment 的创建和显示等操作呢？而之前已经存在的 Fragment 实例也会销毁再次创建，这不就与 Activity 中 onCreate() 方法里面第二次创建的 Fragment 同时显示从而发生 UI 重叠的问题了吗？

**方法：**

第一种方式，在 Activity 提供的 onAttachFragment() 方法中处理：

@Override  
**public void** onAttachFragment(Fragment fragment) {  
 **super**.onAttachFragment(fragment);  
 **if** (fragment **instanceof** OneFragment){  
 oneFragment = (OneFragment) fragment;  
 }  
}

第二种方式，在创建 Fragment 前添加判断，判断是否已经存在：

Fragment **tempFragment** = getSupportFragmentManager().findFragmentByTag(**"OneFragment"**);  
**if** (**tempFragment**==**null**) {  
 oneFragment = OneFragment.newInstance();  
 ft.add(R.id.fl\_content, oneFragment, **"OneFragment"**);  
} **else** {  
 oneFragment = (OneFragment) **tempFragment**;  
}

第三种方式，更为简单，直接利用 savedInstanceState 判断即可：  
  
**if** (savedInstanceState==**null**) {  
 oneFragment = OneFragment.newInstance();  
 ft.add(R.id.fl\_content, oneFragment, **"OneFragment"**);  
} **else** {  
 oneFragment = (OneFragment) getSupportFragmentManager().findFragmentByTag(**"OneFragment"**);  
}

### onActivityResult问题

Fragment 类提供有 startActivityForResult() 方法用于 Activity 间的页面跳转和数据回传，其实内部也是调用 Activity 的对应方法。但是在页面返回时需要注意 Fragment 没有提供 setResult() 方法，可以通过宿主 Activity 实现。

举个例子，在 ActivityA 中的 FragmentA 里面调用 startActivityForResult() 跳转至 ActivityB 中，并在 ActivityB 中的 FragmentB 里面返回到 ActivityA，返回代码如下：

Intent **intent** = **new** Intent();  
getActivity().setResult(Activity.RESULT\_OK, **intent**);  
getActivity().finish();

### Acitivity被杀后fragment的一些反应

在系统要把app回收之前，系统会把Activity的状态保存下来，Activity的FragmentManager负责把Activity中的Fragment保存起来。在“内存重启”后，Activity的恢复是从栈顶逐步恢复，Fragment会在宿主Activity的onCreate方法调用后紧接着恢复（从onAttach生命周期开始）。

## ViewPager+Fragment生命周期

<https://www.jianshu.com/p/150f92fc39b0>

# View

## View的绘制流程

1.Activity里setcontentview{getwindow.setcontentview}

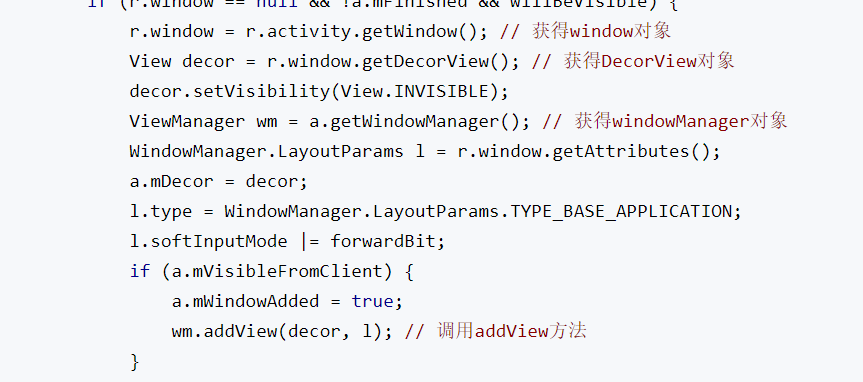
2. Window 本身是一个抽象类，Activity里是持有一个window的，在attach里初始化为phonewindow

3. phonewindow里持有一个decorview(还持有一些其他的，比如actionbar之类的)

4. DecorView 是Phonewindow的一个内部类，继承自framelayout

5. 在generatedecor里生成actionbar和mcontentparent（main.xml的父，id是**content**并在generatelayout添加到mDocor里

首先，在**ActivityThread#handleLaunchActivity**中启动Activity，在这里面会调用到Activity#onCreate方法，从而完成上面所述的DecorView创建动作，当onCreate()方法执行完毕，在handleLaunchActivity方法会继续调用到**ActivityThread#handleResumeActivity**方法，我们看看这个方法的源码：



在该方法内部，获取该activity所关联的window对象，DecorView对象，以及windowManager对象，而WindowManager是抽象类，它的实现类是WindowManagerImpl，所以后面调用的是**WindowManagerImpl#addView**方法，我们看看源码：

接着调用了mGlobal的成员函数，而mGlobal则是WindowManagerGlobal的一个实例，那么我们接着看**WindowManagerGlobal#addView**方法：



先看①号代码处，实例化了ViewRootImpl类，接着，在②号代码处，调用**ViewRootImpl#setView**方法，并把DecorView作为参数传递进去







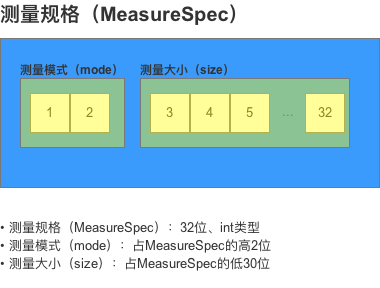


之后会调用**ViewRootImpl#** **PerformTraveals**

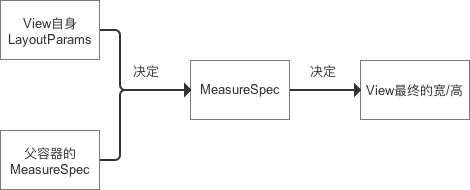
performMeasure、performLayout、performDraw

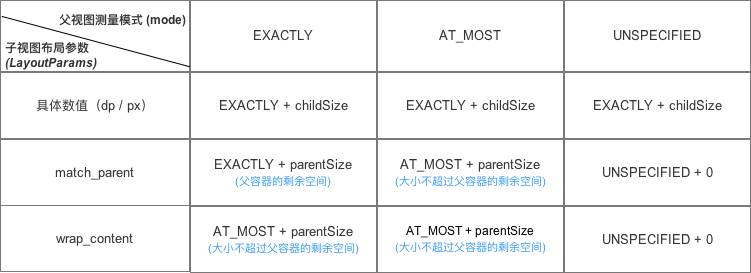
两个很重要的参数：

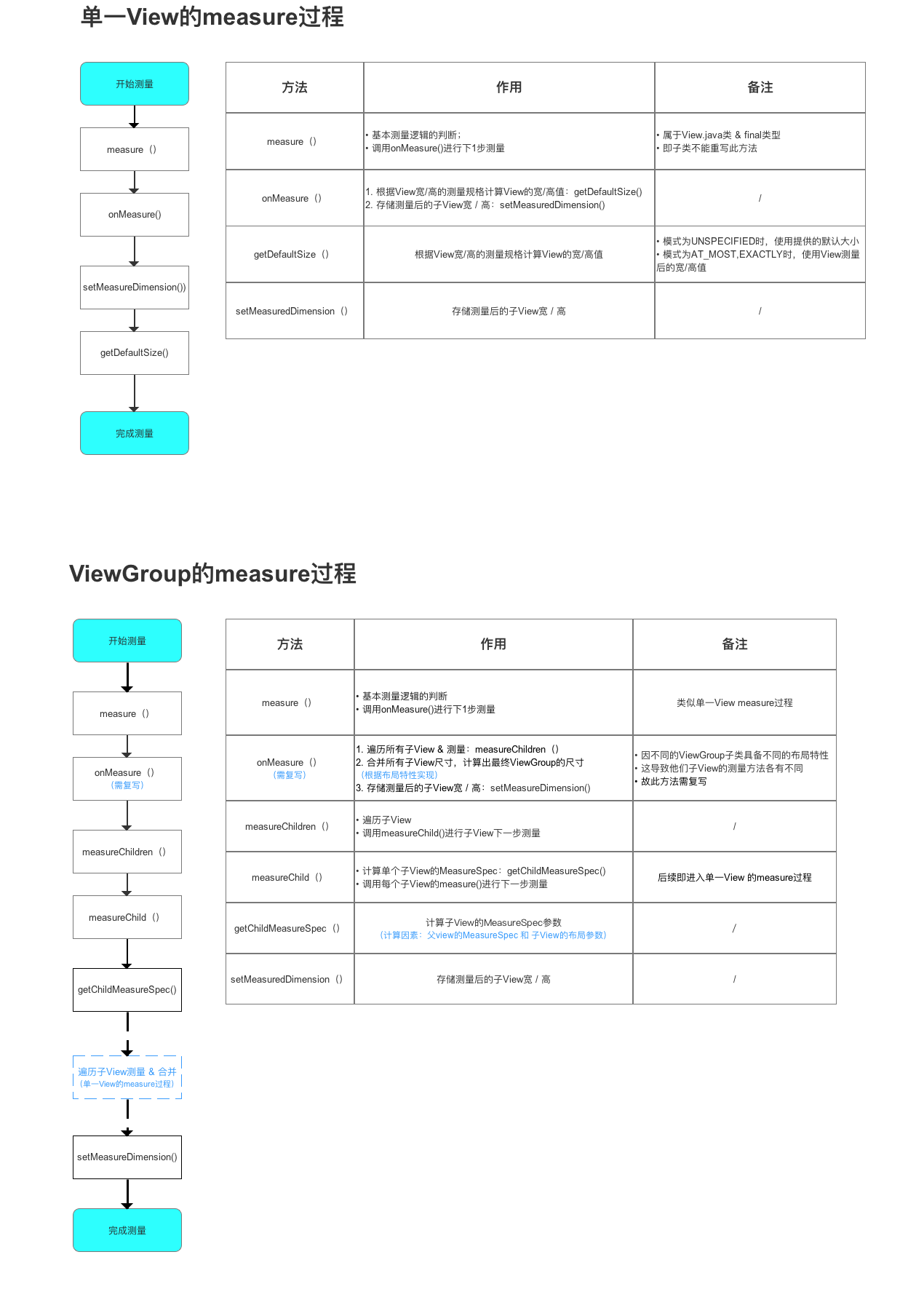
LayoutParam = warp\_content









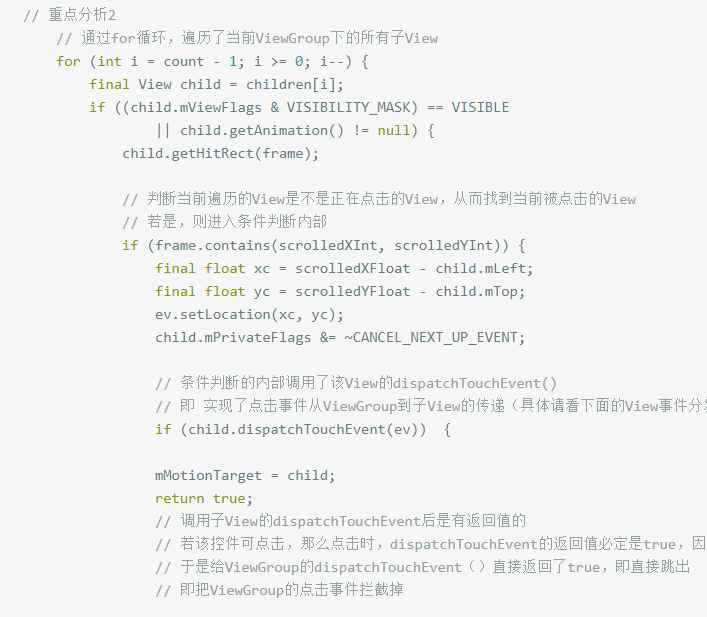


## 事件分发

**从顶层的viewgroup开始, 首先判断拦截，如果被拦截则不允许向子view传递，之后会调用viewgroup父类dispatchtouchevent，自己处理该事件，另外，点击空白地方也是如此**



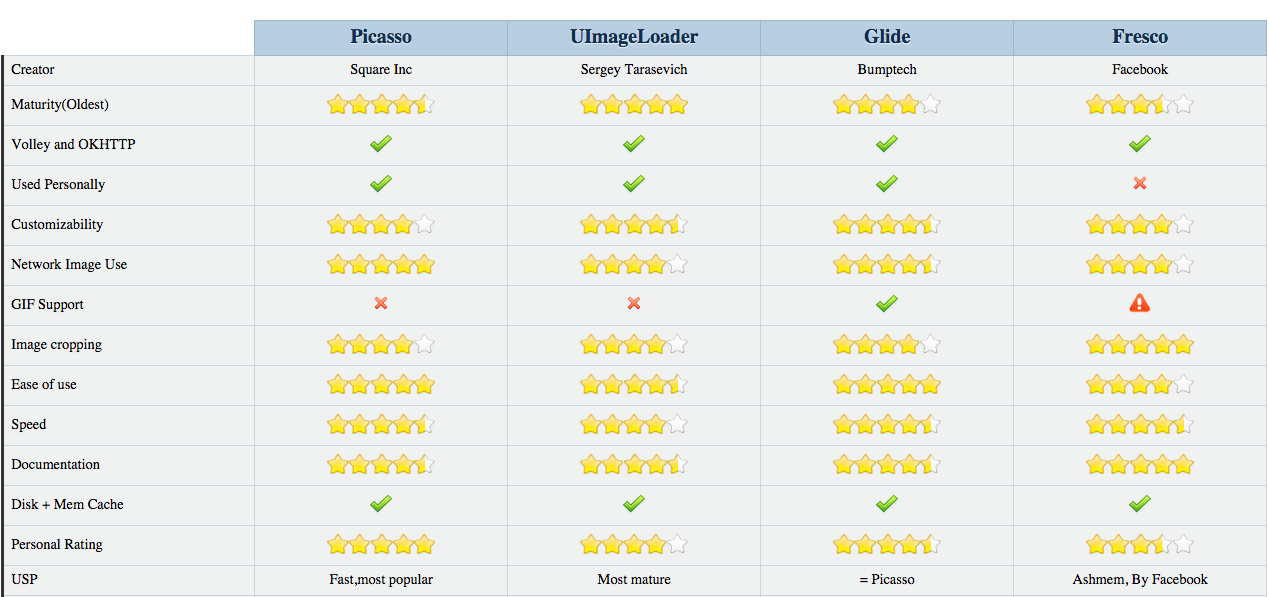
**然后for循环遍历子view，调用子view 的dispatchevent事件，如果返回ture则证明事件被消费，然后return true跳出循环。**



1. 触摸控件（View）首先执行dispatchTouchEvent方法。
2. 在dispatchTouchEvent方法中先执行onTouch方法，后执行onClick方法（onClick方法在onTouchEvent中执行，下面会分析）。
3. 如果控件（View）的onTouch返回false或者mOnTouchListener为null（控件没有设置setOnTouchListener方法）或者控件不是enable的情况下会调运onTouchEvent，dispatchTouchEvent返回值与onTouchEvent返回一样。
4. 如果控件不是enable的设置了onTouch方法也不会执行，只能通过重写控件的onTouchEvent方法处理（上面已经处理分析了），dispatchTouchEvent返回值与onTouchEvent返回一样。
5. 如果控件（View）是enable且onTouch返回true情况下，dispatchTouchEvent直接返回true，不会调用onTouchEvent方法。
6. onTouchEvent方法中会在ACTION\_UP分支中触发onClick的监听。
7. 当dispatchTouchEvent在进行事件分发的时候，只有前一个action返回true，才会触发下一个action。
8. 触摸控件（View）首先执行dispatchTouchEvent方法。
9. 在dispatchTouchEvent方法中先执行onTouch方法，后执行onClick方法（onClick方法在onTouchEvent中执行，下面会分析）。
10. 如果控件（View）的onTouch返回false或者mOnTouchListener为null（控件没有设置setOnTouchListener方法）或者控件不是enable的情况下会调运onTouchEvent，dispatchTouchEvent返回值与onTouchEvent返回一样。
11. 如果控件不是enable的设置了onTouch方法也不会执行，只能通过重写控件的onTouchEvent方法处理（上面已经处理分析了），dispatchTouchEvent返回值与onTouchEvent返回一样。
12. 如果控件（View）是enable且onTouch返回true情况下，dispatchTouchEvent直接返回true，不会调用onTouchEvent方法。
13. 当dispatchTouchEvent在进行事件分发的时候，只有前一个action返回true，才会触发下一个action（也就是说dispatchTouchEvent返回true才会进行下一次action派发）。

# 图片

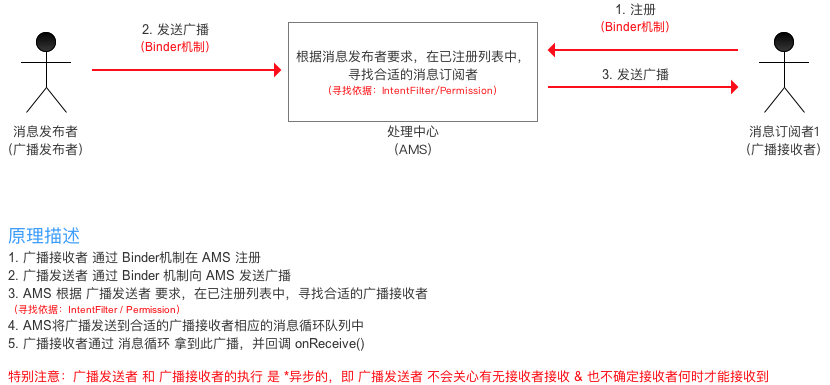




# APP之间通信

* Broadcast
* AIDL
* Fileshare
* Contentprovider

# BroadcastReceiver



**注意：在8.0之后，已经不允许通过manifest 发送implicit，动态注册可以**

**注册放在oncreate中执行，防止多次注册**

**在onResume()注册、onPause()注销是因为onPause()在App死亡前一定会被执行，从而保证广播在App死亡前一定会被注销，从而防止内存泄露。**要回收Activity占用的资源时，Activity在执行完onPause()方法后就会被销毁，有些生命周期方法onStop()，onDestory()就不会执行。当再回到此Activity时，是从onCreate方法开始执行。

**1、概念介绍及两种注册方式的区别**

BroadcastReceiver作为Android四大组件之一，不像Activity，没有可显示的界面。BroadcastReceiver包括两个概念，广播发送者和广播接收者(Receiver)，这里的广播实际就是指Intent，程序可以自己发送广播自己接收，也可以接受系统或其他应用的广播或是发送广播给其他应用程序。

发送者可以通过类似[Context.sendBroadcast](http://developer.android.com/reference/android/content/Context.html" \l "sendBroadcast(android.content.Intent))接口发送广播，接收者通过[Context.registerReceiver()](http://developer.android.com/reference/android/content/Context.html" \l "registerReceiver(android.content.BroadcastReceiver,%20android.content.IntentFilter))动态注册或在AndroidManifest.xml文件中通过<receiver>标签静态注册，注册完成后，当发送者发送某个广播时系统会将发送的广播(Intent)与系统中所有注册的符合条件的接收者(Receiver) 的IntentFilter进行匹配，若匹配成功则执行相应接收者的onReceive函数，匹配规则见Intent和IntentFilter的匹配规则。

关于registerReceiver动态注册和通过<receiver>标签静态注册广播的区别如下：

a.对bindService的调用，<receiver>注册的广播，在onReceive结束后广播即不存在，所以不能在其中给自己异步传递结果，如bindService而只能使用startService，如果想跟service交互可使用peekService。

b. 手动控制。registerReceiver为动态注册，自己可以手动注册或是取消注册；<receiver>标签为静态注册，由系统开机时自动扫描注册，所以无法手动控制，开机一直运行中。

c. 资源消耗不同。registerReceiver可以手动控制，所以适当的注册和取消注册能节省系统资源，<receiver>标签系统开机后一直有效。

d. 有效期不同。通过registerReceiver注册的BroadcastReceiver在对其进行注册的Context对象"销毁"了或者调用了unregisterReceiver方法时也就失效了，而通过<receiver>标签注册的BroadcastReceiver只要应用程序没有被删除就一直有效。

e. 对registerReceiver函数的调用许可不同。通过registerReceiver注册的BroadcastReceiver在其onReceive函数中可以再次调用某个Context的registerReceiver函数，而通过<receiver>标签注册的BroadcastReceiver不允许再调用某个Context的registerReceiver函数 。

f. 使用情况不同。对于自己发送和接受的广播可以通过registerReceiver注册，对于系统常用广播的接收通常用<receiver>标签注册。

**3、生命周期**

BroadcastReceiver在onReceive函数执行结束后即表示生命周期结束，所以不适合在onReceive中做绑定服务操作，结束后若某个进程只含有该BroadcastReceiver，则优先级将降低可能被系统回收，所以BroadcastReceiver中不适合做一些异步操作，如新建线程下载数据，BroadcastReceiver结束后可能在异步操作完成前进程已经被系统kill。

同时由于ANR限制BroadcastReceiver的onReceive函数必须在10秒内完成，而且onReceive默认会在主线程中执行，所以BroadcastReceiver中不适合做一些耗时操作，对于耗时操作需要交给service处理，比如网络或数据库耗时操作、对话框的显示(因为现实时间可能超时，用Notification代替)。

**4、安全性**

BroadcastReceiver的设计初衷就是从全局考虑的，可以方便应用程序和系统、应用程序之间、应用程序内的通信，所以对单个应用程序而言BroadcastReceiver是存在安全性问题的，相应问题及解决如下：

a、当应用程序发送某个广播时系统会将发送的Intent与系统中所有注册的BroadcastReceiver的IntentFilter进行匹配，若匹配成功则执行相应的onReceive函数。可以通过类似sendBroadcast(Intent, String)的接口在发送广播时指定接收者必须具备的permission。或通过Intent.setPackage设置广播仅对某个程序有效。

b. 当应用程序注册了某个广播时，即便设置了IntentFilter还是会接收到来自其他应用程序的广播进行匹配判断。对于动态注册的广播可以通过类似registerReceiver(BroadcastReceiver, IntentFilter, String, android.os.Handler)的接口指定发送者必须具备的permission，对于静态注册的广播可以通过android:exported="false"属性表示接收者对外部应用程序不可用，即不接受来自外部的广播。

c.上面两个问题其实都可以通过LocalBroadcastManager来解决，LocalBroadcastManager只会将广播限定在当前应用程序中，具体见下面6特殊的BroadcastReceiver中的介绍

d.使用[android:protectionLevel](https://developer.android.com/guide/topics/manifest/permission-element.html" \l "plevel" \t "_blank)

**5、分类**

BroadcastReceiver可以分为普通和有序两种，下面6特殊BroadcastReceiver中介绍了其他一些种类。

通过Context.sendBroadcast发送的广播即为普通广播，对于普通广播接收者接收到它的顺序是不定的，所以接收者接收到后无法使用其他接收者对它的处理结果也无法停止它。

通过Context.sendOrderedBroadcast发送的广播即为有序广播，与普通广播的不同在于，接收者是有序接收到广播的并且可以对广播进行修改或是取消广播向下传递。系统根据接收者定义的优先级顺序决定哪个接收者先接收到它，接收者处理完后可以将结果传递给优先级低的接收者也可以停止广播使得其他优先级低的接收者无法接收到该广播。优先级通过android:priority属性定义，数值越大优先级别越高，取值范围：-1000到1000，虽然API文档介绍对sendBroadcast发送的广播无效，不过本人测试同样有效，相同优先级的接收者接收到广播的顺序随机。Android系统收到短信、接到电话后发送的广播都是有序广播，所以可以进行短信或电话的拦截，即取消广播。

PS：有序广播可以在onReceive函数中通过BroadcastReceiver的[abortBroadcast](http://developer.android.com/reference/android/content/BroadcastReceiver.html" \l "abortBroadcast())接口(这个接口对[sendBroadcast](http://developer.android.com/reference/android/content/Context.html" \l "sendBroadcast(android.content.Intent))发送广播无效)取消广播，通过接口sendOrderedBroadcast(Intent, String, BroadcastReceiver, android.os.Handler, int, String, Bundle)发送的广播即便优先级高的广播取消了广播，接口参数中指定的BroadcastReceiver依然可以在其他接收者处理完后接收到广播。通过BroadcastReceiver的[getResultExtras](http://developer.android.com/reference/android/content/BroadcastReceiver.html" \l "getResultExtras(boolean))接口获得结果的Bundle再通过Bundle的putString和getString方法修改或获取数据，可以见本文最后的实例代码举例。

**6、特殊的BroadcastReceiver**

a. LocalBroadcastManager本地广播

全局广播：发出去的广播可以被任何应用程序接收到  
本地广播：发出去的广播只能被本应用程序接收到，一般使用LocalBroadcastManager对广播进行管理。

如果你之前有使用过普通的广播，你会发现在方法调用上，LocalBroadcastManager和普通的广播是一模一样，不同的LocalBroadCastManager的调用方不再是context，而是LocalBroadCastManager的实例，所以所有的逻辑都是在LocalBroadCastManager的掌控之内本地广播通过

LocalBroadcastManager.getInstance(context).sendBroadcast(intent)发送广播，LocalBroadcastManager.getInstance(context).registerReceiver注册服务，通过LocalBroadcastManager.getInstance(context).unregisterReceiver取消注册服务，其他同普通广播.

b. Sticky Broadcast粘性广播

如果发送者发送了某个广播，而接收者在这个广播发送后才注册自己的Receiver，这时接收者便无法接收到刚才的广播，为此Android引入了StickyBroadcast，在广播发送结束后会保存刚刚发送的广播（Intent），这样当接收者注册完Receiver后就可以继续使用刚才的广播。如果在接收者注册完成前发送了多条相同Action的粘性广播，注册完成后只会收到一条该Action的广播，并且消息内容是最后一次广播内容。系统网络状态的改变发送的广播就是粘性广播。

粘性广播通过Context的[sendStickyBroadcast(Intent)](http://developer.android.com/reference/android/content/Context.html" \l "sendStickyBroadcast(android.content.Intent))接口发送，需要添加权限<uses-permission android:name="android.permission.BROADCAST\_STICKY"/>

也可以通过Context的[removeStickyBroadcast](http://developer.android.com/reference/android/content/Context.html" \l "removeStickyBroadcast(android.content.Intent))([Intent](http://developer.android.com/reference/android/content/Intent.html) intent)接口移除缓存的粘性广播。

c. OrderedBroadcastReceiver有序广播

这个在5分类中已经介绍，接收者有序接收广播并可以修改广播结果或是取消广播，通过Context的[sendOrderedBroadcast](http://developer.android.com/reference/android/content/Context.html" \l "sendOrderedBroadcast(android.content.Intent,%20java.lang.String))接口发送

d. StickyOrderedBroadcast粘性有序广播

这个就是粘性广播和有序广播的结合了，通过Context的[sendStickyOrderedBroadcast](http://developer.android.com/reference/android/content/Context.html" \l "sendStickyOrderedBroadcast(android.content.Intent,%20android.content.BroadcastReceiver,%20android.os.Handler,%20int,%20java.lang.String,%20android.os.Bundle))接口发送。

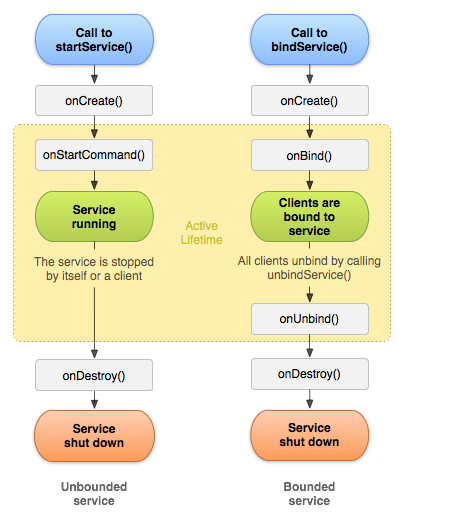
# Service

## 1. 简介

* 定义：服务，是Android四大组件之一， 属于 **计算型组件**
* 作用：提供 需在后台长期运行的服务

如：复杂计算、音乐播放、下载等

* 特点：无用户界面、在后台运行、生命周期长
* 8.0之后不允许长时间驻留在后台，可以用job scheduler代替
* Service默认是在主线程执行的，如果需要耗时操作，请重新创建线程运行
* If the system kills your service, it restarts it as soon as resources become available

2. 生命周期

## 2.具体使用解析

### 2.1 本地Service

这是最普通、最常用的后台服务Service。

### 2.1.1 使用步骤

* 步骤1：新建子类继承Service类

需重写父类的onCreate()、onStartCommand()、onDestroy()和onBind()方法

* 步骤2：构建用于启动Service的Intent对象
* 步骤3：调用startService()启动Service、调用stopService()停止服务
* 步骤4：在AndroidManifest.xml里注册Service

## 2.2 可通信的服务Service

* 上面介绍的Service是最基础的，但只能单机使用，即无法与Activity通信
* 接下来将在上面的基础用法上，增设“与Activity通信”的功能，即使用绑定Service服务（Binder类、bindService()、onBind(）、unbindService()、onUnbind()）

## 2.3 前台Service

前台Service和后台Service（普通）最大的区别就在于：

* 前台Service在下拉通知栏有显示通知（如下图），但后台Service没有；
* 前台Service优先级较高，不会由于系统内存不足而被回收；后台Service优先级较低，当系统出现内存不足情况时，很有可能会被回收

**通信方法：**

[Extending the Binder class](https://developer.android.com/guide/components/bound-services.html#Binder) （进程内）

Here's how to set it up:

1. In your service, create an instance of [Binder](https://developer.android.com/reference/android/os/Binder.html) that does one of the following:
   * Contains public methods that the client can call.
   * Returns the current [Service](https://developer.android.com/reference/android/app/Service.html) instance, which has public methods the client can call.
   * Returns an instance of another class hosted by the service with public methods the client can call.
2. Return this instance of [Binder](https://developer.android.com/reference/android/os/Binder.html) from the [onBind()](https://developer.android.com/reference/android/app/Service.html" \l "onBind(android.content.Intent)) callback method.
3. In the client, receive the [Binder](https://developer.android.com/reference/android/os/Binder.html) from the [onServiceConnected()](https://developer.android.com/reference/android/content/ServiceConnection.html" \l "onServiceConnected(android.content.ComponentName,%20android.os.IBinder)) callback method and make calls to the bound service using the methods provided.

[Using a Messenger](https://developer.android.com/guide/components/bound-services.html#Messenger) （IPC）单个进程间通信

Here's a summary of how to use a [Messenger](https://developer.android.com/reference/android/os/Messenger.html):

1. The service implements a [Handler](https://developer.android.com/reference/android/os/Handler.html) that receives a callback for each call from a client.
2. The service uses the [Handler](https://developer.android.com/reference/android/os/Handler.html) to create a [Messenger](https://developer.android.com/reference/android/os/Messenger.html) object (which is a reference to the [Handler](https://developer.android.com/reference/android/os/Handler.html)).
3. The [Messenger](https://developer.android.com/reference/android/os/Messenger.html) creates an [IBinder](https://developer.android.com/reference/android/os/IBinder.html) that the service returns to clients from [onBind()](https://developer.android.com/reference/android/app/Service.html" \l "onBind(android.content.Intent)).
4. Clients use the [IBinder](https://developer.android.com/reference/android/os/IBinder.html) to instantiate the [Messenger](https://developer.android.com/reference/android/os/Messenger.html) (that references the service's [Handler](https://developer.android.com/reference/android/os/Handler.html)), which the client uses to send [Message](https://developer.android.com/reference/android/os/Message.html) objects to the service.
5. The service receives each [Message](https://developer.android.com/reference/android/os/Message.html) in its [Handler](https://developer.android.com/reference/android/os/Handler.html)—specifically, in the [handleMessage()](https://developer.android.com/reference/android/os/Handler.html" \l "handleMessage(android.os.Message)) method.

[Using AIDL](https://developer.android.com/guide/components/aidl.html) 多个进程访问一个service，解决多线程

主要是通过下面构建回调函数，拿到service里的东西

private ServiceConnection mConnection = new ServiceConnection() {

### IntentService

IntentService 是继承于 Service 并处理异步请求的一个类，在 IntentService 内有一个工作线程来处理耗时操作，启动 IntentService 的方式和启动传统 Service 一样，同时，当任务执行完后，IntentService 会自动停止，而不需要我们去手动控制。另外，可以启动 IntentService 多次，而每一个耗时操作会以工作队列的方式在IntentService 的 onHandleIntent 回调方法中执行，并且，每次只会执行一个工作线程，执行完第一个再执行第二个，以此类推。

而且，所有请求都在一个单线程中，不会阻塞应用程序的主线程（UI Thread），同一时间只处理一个请求。 那么，用 IntentService 有什么好处呢？首先，我们省去了在 Service 中手动开线程的麻烦，第二，当操作完成时，我们不用手动停止 Service。

* It creates a default worker thread that executes all of the intents that are delivered to [onStartCommand()](https://developer.android.com/reference/android/app/Service.html" \l "onStartCommand(android.content.Intent,%20int,%20int)), separate from your application's main thread.
* Creates a work queue that passes one intent at a time to your [onHandleIntent()](https://developer.android.com/reference/android/app/IntentService.html" \l "onHandleIntent(android.content.Intent)) implementation, so you never have to worry about multi-threading.
* Stops the service after all of the start requests are handled, so you never have to call [stopSelf()](https://developer.android.com/reference/android/app/Service.html" \l "stopSelf()).
* Provides a default implementation of [onBind()](https://developer.android.com/reference/android/app/IntentService.html" \l "onBind(android.content.Intent)) that returns null.
* Provides a default implementation of [onStartCommand()](https://developer.android.com/reference/android/app/IntentService.html" \l "onStartCommand(android.content.Intent,%20int,%20int)) that sends the intent to the work queue and then to your [onHandleIntent()](https://developer.android.com/reference/android/app/IntentService.html" \l "onHandleIntent(android.content.Intent)) implementation.

# Content Provider

**他默认也是在主线程中使用，因此需要CursorLoader 即在loader中实现的来进行异步操作**

**使用情景**

* Sharing access to your application data with other applications
* Sending data to a widget
* Returning custom search suggestions for your application through the search framework using [SearchRecentSuggestionsProvider](https://developer.android.com/reference/android/content/SearchRecentSuggestionsProvider.html)
* Synchronizing application data with your server using an implementation of[AbstractThreadedSyncAdapter](https://developer.android.com/reference/android/content/AbstractThreadedSyncAdapter.html)
* Loading data in your UI using a [CursorLoader](https://developer.android.com/reference/android/content/CursorLoader.html)

# Android 数据存储https://upload-images.jianshu.io/upload_images/944365-f46f7030ad445462.png?imageMogr2/auto-orient/

Dalvik 经过优化，允许在有限的内存中同时运行多个虚拟机的实例，并且每一个Dalvik 应用作为一个独立的Linux 进程执行。独立的进程可以防止在虚拟机崩溃的时候所有程序都被关闭。

即Android Runtime  
ART 的机制与 Dalvik 不同。在Dalvik下，应用每次运行的时候，字节码都需要通过即时编译器（just in time ，JIT）转换为机器码，这会拖慢应用的运行效率，而在ART 环境中，应用在第一次安装的时候，字节码就会预先编译成机器码，使其成为真正的本地应用。这个过程叫做预编译（AOT,Ahead-Of-Time）。这样的话，应用的启动(首次)和执行都会变得更加快速。占用的存储空间也会相应的变大

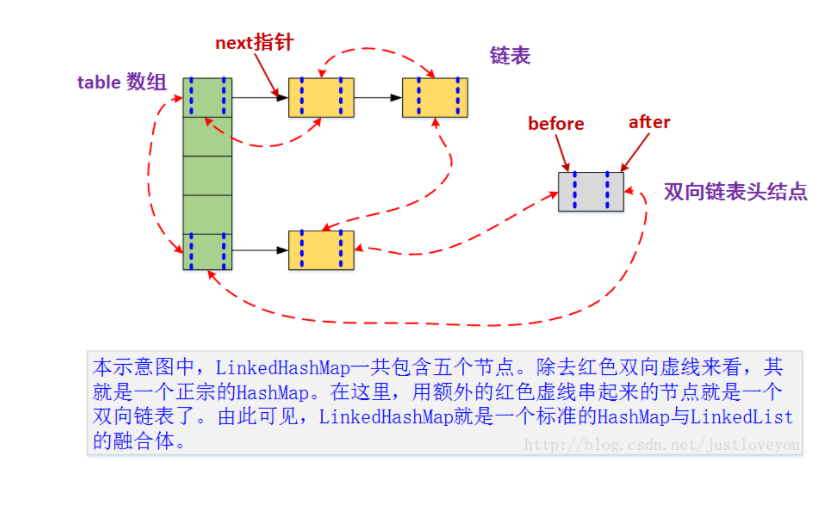
# LRU

  LRU，全称Least Rencetly Used，即最近最少使用，是一种非常常用的置换算法，也即淘汰最长时间未使用的对象。LRU在操作系统中的页面置换算法中广泛使用，我们的内存或缓存空间是有限的，当新加入一个对象时，造成我们的缓存空间不足了，此时就需要根据某种算法对缓存中原有数据进行淘汰货删除，而LRU选择的是将最长时间未使用的对象进行淘汰。

    根据LRU算法的思想，要实现LRU最核心的是要有一种数据结构能够基于访问顺序来保存缓存中的对象，这样我们就能够很方便的知道哪个对象是最近访问的，哪个对象是最长时间未访问的。LruCache选择的是LinkedHashMap这个数据结构，LinkedHashMap是一个双向循环链表，

显然，在LruCache中选择的是accessOrder = true；此时，当accessOrder 设置为 true时，每当我们更新（即调用put方法）或访问（即调用get方法）map中的结点时，LinkedHashMap内部都会将这个结点移动到链表的尾部，因此，在链表的尾部是最近刚刚使用的结点，在链表的头部是是最近最少使用的结点，当我们的缓存空间不足时，就应该持续把链表头部结点移除掉，直到有剩余空间放置新结点。

可以看到，LinkedHashMap完成了LruCache中的核心功能，那LruCache中剩下要做的就是定义缓存空间总容量，当前保存数据已使用的容量，对外提供put、get方法。



# 动态加载与静态加载的区别

1、动态加载是一种优化，降低资源的耗费。偶尔，在布局中会有一些比较复杂但是又很少用到的控件。不管它是消息详情，进度条还是未完成的提示信息，你都可以直到真正需要的时候再加载他们，以降低你的内存消耗，提升渲染效率。

2、动态布局，也就是可以根据业务的需求改变界面。实际上就是用代码写出界面，代码量比较大。而且维护起来十分的繁琐。特别是一些界面空间比较多的时候。静态的布局，是通过xml来实现的，适用于页面比较固定的情况。但是维护起来比较方便。

# 异步

## Handler and asynctask

## 什么是handler？

A Handler allows you to **send and process** Message and Runnable objects associated with a thread's MessageQueue.

## handler的作用

There are two main uses for a Handler:

(1) to schedule messages and runnables to be executed as some point in the future;

(2) to enqueue an action to be performed on a different thread than your own.

## 什么是Looper

Class used to run a message loop for a thread.

## 什么是ThreadLocal

The TheadLocal construct allows us to store data that will be accessible only by a specific thread.

其实就是每个Threadlocal把每个线程当做key，这个value就是对应线程的map，所以比如在A线程里A=1，但是在B线程里就不一定了

所以Looper里包含一个Threadlocal<Looper>，就是为了获取不同线程下的Looper

内部有俩线程池，一个是异步，一个是串行的

     在Android开发中，AsyncTask可以使得用户避免直接使用Thread类和Handler 来处理后台操作，适用于需要异步处理数据并将数据更新到界面上的情况。AsyncTask适用于后台操作只有几秒的短时操作。但是AsyncTask本身存在很多糟糕的问题，如果使用中不注意，将会影响程序的健壮性。

1、生命周期

       很多开发者会认为一个在Activity中创建的AsyncTask会随着Activity的销毁而销毁。然而事实并非如此。AsyncTask会一直执行, 直到doInBackground()方法执行完毕。然后，如果 cancel(boolean)被调用, 那么onCancelled(Result result) 方法会被执行；否则，执行onPostExecute(Result result) 方法。如果我们的Activity销毁之前，没有取消 AsyncTask，这有可能让我们的AsyncTask崩溃(crash)。因为它想要处理的view已经不存在了。所以，我们总是必须确保在销毁活动之前取消任务。总之，我们使用AsyncTask需要确保AsyncTask正确地取消。

       另外，即使我们正确地调用了cancle() 也未必能真正地取消任务。因为如果在doInBackgroud里有一个不可中断的操作，比如BitmapFactory.decodeStream()，那么这个操作会继续下去。

2、内存泄漏

        如果AsyncTask被声明为Activity的非静态的内部类，那么AsyncTask会保留一个对创建了AsyncTask的Activity的引用。如果Activity已经被销毁，AsyncTask的后台线程还在执行，它将继续在内存里保留这个引用，导致Activity无法被回收，引起内存泄露。

3、结果丢失

       屏幕旋转或Activity在后台被系统杀掉等情况会导致Activity的重新创建，之前运行的AsyncTask会持有一个之前Activity的引用，这个引用已经无效，这时调用onPostExecute()再去更新界面将不再生效。

4、并行还是串行

      在Android 1.6之前的版本，AsyncTask是串行的，在1.6至2.3的版本，改成了并行的。在2.3之后的版本又做了修改，可以支持并行和串行，当想要串行执行时，直接执行execute()方法，如果需要并行执行，则要执行executeOnExecutor(Executor)。

## Loader：

The Loader API lets you load data from a [content provider](https://developer.android.com/guide/topics/providers/content-providers.html) or other data source for display in an [FragmentActivity](https://developer.android.com/reference/android/support/v4/app/FragmentActivity.html) or [Fragment](https://developer.android.com/reference/android/support/v4/app/Fragment.html). If you don't understand why you need the Loader API to perform this seemingly trivial operation, then first consider some of the problems you might encounter without loaders:

* If you fetch the data directly in the activity or fragment, your users will suffer from lack of responsiveness due to performing potentially slow queries from the UI thread.
* If you fetch the data from another thread, perhaps with [AsyncTask](https://developer.android.com/reference/android/os/AsyncTask.html), then you're responsible for managing both the thread and the UI thread through various activity or fragment lifecycle events, such as[onDestroy()](https://developer.android.com/reference/android/app/Activity.html#onDestroy()) and configurations changes.

Loaders solve these problems and includes other benefits. For example:

* Loaders run on separate threads to prevent janky or unresponsive UI.
* Loaders simplify thread management by providing callback methods when events occur.
* Loaders persist and cache results across configuration changes to prevent duplicate queries.
* Loaders can implement an observer to monitor for changes in the underlying data source. For example, [CursorLoader](https://developer.android.com/reference/android/support/v4/content/CursorLoader.html) automatically registers a [ContentObserver](https://developer.android.com/reference/android/database/ContentObserver.html) to trigger a reload when data changes.
* Loaders persist and cache results across configuration changes to prevent duplicate queries.

能缓存，能检查数据源的变化 For example, [CursorLoader](https://developer.android.com/reference/android/support/v4/content/CursorLoader.html) automatically registers a [ContentObserver](https://developer.android.com/reference/android/database/ContentObserver.html) to trigger a reload when data changes

内部是通过asynctask实现

一次完整的数据加载流程为Activity调用LoaderManager的doStart()方法，然后LoaderManager调用Loader的startLoading()方法，然后Loader调运AsyncTaskLoader的doingBackground()方法进行耗时数据加载，然后AsyncTaskLoader回调LoaderManager的complete数据加载完成方法，接着LoaderManager回调我们在Activity中实现的callback中的onLoadFinish()方法。

Acivity和Fragment的生命周期主动管理了LoaderManager，每个Activity用一个ArrayMap的mAllLoaderManager来保存当前Activity及其附属Frament的唯一LoaderManager；在Activity配置发生变化时，Activity在destory前会保存mAllLoaderManager，当Activity再重新创建时，会在Activity的onAttcach()、onCreate()、performStart()方法中恢复mAllLoaderManager。

LoaderManager给Activity提供了管理自己的一些方法；同时主动管理了对应的Loader，它把每一个Loader封装为LoadInfo对象，同时它负责主动调运管理Loader的startLoading()、stopLoading()、,forceLoad()等方法。

由于整个Activity和Fragment主动管理了Loader，所以关于Loader的释放（譬如CursorLoader的Cursor关闭等）不需要我们人为处理，Loader框架会帮我们很好的处理的；同时特别注意，对于CursorLoader，当我们数据源发生变化时Loader框架会通过ContentObserver调用onContentChanged的forceLoad方法重新请求数据进行回调刷新。

## IntentService

内部其实用的是handlerThread，没有UI to serve(asynctask), handler thread没事的时候 会一直运行下去，省去了手动开线程的麻烦，service结束后会自动关闭，不容易被杀死

注意：intentservice在onstartcommand，onstart中实现了消息的传递，因此不能使用bind来开启service服务

## JobScheduler and GcmNetworkManager

## [CountDownTimer](https://developer.android.com/reference/android/os/CountDownTimer.html)

## [Java Threads](https://developer.android.com/reference/java/lang/Thread.html) or [Android HandlerThread](https://developer.android.com/reference/android/os/HandlerThread.html)

## [FutureTask](https://developer.android.com/reference/java/util/concurrent/FutureTask.html)

我们可以对它进行控制，比如终止任务，查询任务是否完成，获取结果等。Java [Timer](https://developer.android.com/reference/java/util/Timer.html) / [ScheduledThreadPoolExecutor](https://developer.android.com/reference/java/util/concurrent/ScheduledThreadPoolExecutor.html" \t "_blank)

# Android 图片

现在做android开发，在加载图片时，都是用现成的框架，简单粗暴，但是如果没有这些框架你是怎么处理的，你会有什么样的疑问。  
关于图片加载时的疑问：  
**1，对于一些大图片，特别加载很多大图片的时候，出现OOM怎么办？**  
压缩处理，官方给出的压缩算法： }

**2，对于listview中的图片，如果每从网络获取一张图片都要开启一个线程，势必会浪费掉很多资源，服务器也承受不起这么多的连接，如何解决？**  
使用线程池，可以避免了过多线程频繁创建和销毁，有的童鞋每次总是new一个线程去执行这是非常不可取的，  
好一点的用的AsyncTask类，其实内部也是用到了线程池。  
  
**3，如果每显示一张图片都要去请求一次网络，势必会浪费掉很多流量，怎样才能节省流量，这又如何解决？**  
使用缓存，内存缓存，本地缓存  
  
**4，如果listview在滑动的过程中都要去加载图片，势必是不科学的，如何解决？**  
监听listview的滑动状态，Listview滑动时不加载数据，停下来时再去加载数据

**5，有时由于listview中convertView的复用，引发图片显示错位，改如何解决？**

图片错位问题的本质源于我们的listview使用了缓存convertView，假设一种场景，一个listview一屏显示九个item，那么在拉出第十个item的时候，事实上该item是重复使用了第一个item，  
也就是说在第一个item从网络中下载图片并最终要显示的时候其实该item已经不在当前显示区域内了，此时显示的后果将是在可能在第十个item上输出图像，  
这就导致了图片错位的问题。所以解决之道在于可见则显示，不可见则不显示。在ImageLoader里有个imageViews的map对象，就是用于保存当前显示区域图像对应的url集，在显示前判断处理一下即可。  
  
最简单的解决方法就是网上说的，给 ImageView 设置一个 tag, 并预设一个图片。  
  
图片处理考虑点：  
1，图片压缩  
2，图片缓存：   
a,内存缓存  
1，软引用：  
map里面的键是用来放图片地址的，既可以是网络上的图片地址，也可以SDcard上的图片地址，map里面的值里面放的是持有软引用的Bitmap，  
当然如果你要放Drawable，那也是可以的。  
使用：private Map<String, SoftReference<Bitmap>> imageMap = new HashMap<String, SoftReference<Bitmap>>();  
2，LruCache缓存(最近最少使用算法)：  
当缓存的图片达到了预先设定的值的时候，那么近期使用次数最少的图片就会被回收掉。  
  
b,本地缓存  
1，DiskLruCache  
3，listview图片加载错位

图片三级缓存所用知识点：  
1，内存缓存，使用LRUCache(最近最少使用)算法  
2，使用了线程池来管理下载任务，提升多个图片加载效率  
3，如何防止图片错位，设置给imageView设置tag

# 进程间通信

在android开发中一个App很少情况下是只具备一个进程的，因此多个进程存活在同一个App中非常正常，这对减少app的pss也很有帮助。出现多个进程，也不得不谈及跨进程通信，目前主要有以下几种跨进程通信方式：

**Activity-Activity**

A\*B: Intent(Bundle)

B-A: startActivityForResult onActivityResult

Activity-Fragment

A-F通常A都是持有F的实例的，因此直接在F中设置getinfo函数就行；如果Activity中没有Fragment实例，每个Fragment都有唯一的TAG或者ID,可以通过getFragmentManager.findFragmentByTag()或者findFragmentById()获得任何Fragment实例；如果Activity向fragment中传递信息的话，用F.setArguments ;回调监听；设置广播

F-A 直接通过getActivity

Activity-Service

1.Bundle方式，该方式一般用于Service，Activity，BroadcastReceiver组件间通信，当进程A的ActivityA，想传个东西给进程B的ActivityB，此时就可以采用Bundle方式传递，但是有个前提就是Bundle支持的数据类型有限，有时候需要对数据进行Parcelable或Serializable；但有时候你觉得序列化很麻烦，IntentService策略也是一种选择，将获取ActivityB所要用到的数据处理逻辑让IntentService来处理，而IntentService也设置在进程B中，这样在ActivityA中启动下IntentService，结果马上就出来了。

2.文件共享，该方法简单，但是并不适合高并发场景，尽管在android中多个进程对文件进行读写没有报错，但是获取数据可能存在问题，对并发访问实时性不是很高的，可以考虑下这种方法；

3.Messenger方式，注意不是Message，该方式尽管底层逻辑也是AIDL模式，功能一般，对高并发情景处理也不是很乐观，当你需要服务器和客户端交换些数据时，可以考虑下这种方法；

4.ContentProvider，这本就是android的组件，效率也不要考虑，不过也可以看成是约束性AIDL，主要提供CRUD操作，支持一对多进程间通信；

5.Socket，有个网络经验的自然不会陌生，但其实现细节有点繁琐，也可以适用于网络数据交换；

6.SharedPreference，这个东西尽管大家经常用，但他不适合跨进程传递数据，如果喜欢它了，用ContentProvider改造下即可,重点实现getType方法。

7.AIDL，

# Activity 启动流程

       整个应用程序的启动过程要执行很多步骤，但是整体来看，主要分为以下五个阶段：

       一. Step1 - Step 11：Launcher通过Binder进程间通信机制通知ActivityManagerService，它要启动一个Activity；

       二. Step 12 - Step 16：ActivityManagerService通过Binder进程间通信机制通知Launcher进入Paused状态；

       三. Step 17 - Step 24：Launcher通过Binder进程间通信机制通知ActivityManagerService，它已经准备就绪进入Paused状态，于是ActivityManagerService就创建一个新的进程，用来启动一个ActivityThread实例，即将要启动的Activity就是在这个ActivityThread实例中运行；

       四. Step 25 - Step 27：ActivityThread通过Binder进程间通信机制将一个ApplicationThread类型的Binder对象传递给ActivityManagerService，以便以后ActivityManagerService能够通过这个Binder对象和它进行通信；

       五. Step 28 - Step 35：ActivityManagerService通过Binder进程间通信机制通知ActivityThread，现在一切准备就绪，它可以真正执行Activity的启动操作了。

     (1)：首先，系统刚刚启动的时候加载的是Linux内核，在Linux内核加载完成之后会创建出来一个init进程，启动init进程解析init.rc文件首先会fork出一个叫ServerManager的子进程，该进程主要用于管理我们的系统服务，他内部存在一个server服务列表，这个列表中存储的就是那些已经注册的系统服务，随后init进程会fork出一个Zygote的子进程，Zygote进程是一个守护进程，之后，当我们的应用程序想要启动的时候，实际上是由Zygote来fork出子进程给我们的应用程序的，这就保证了我们的应用程序运行在单独的子进程中，即使一个应用程序奔溃也不会影响另一个应用程序的执行；  
        (2)：在Zygote进程启动之后，会通过ZygoteInit的main方法fork出一个SystemServer子进程，SystemServer进程在启动的时候会创建ActivityThread对象以及系统上下文，注意一点就是虽然ActivityThread里面带有Thread，但他仅仅是一个普通的final类而已，并没有继承Thread或者实现Runnable接口，在ActivityThread里面存在一个内部类ApplicationThread，虽然ApplicationThread里面带有Thread，同样他也没有继承Thread或者实现Runnable接口，他实际上是一个Binder类，接着在SystemServer的main方法里面会创建ServerThread对象(同样虽然ServerThread里面带有Thread，但他只是一个普通类)，并且调用他的initAndLoop方法，在initAndLoop方法里面会初始化诸如ActivityManagerService等一系列系统级Service对象，并且调用ServerManager的addService方法，将这些Service服务注册到ServerManager里面，同时启动这些服务；到这里，系统进程的启动工作就结束了，之后，会开启系统Launcher程序来完成系统界面的加载与显示；  
        (3)：在我们点击应用程序的桌面图标的时候，App就由Launcher开始启动了，Launcher实际上是继承了Activity的，并且实现了点击、长点击等等监听器，在点击桌面图标的时候，实际上执行的是Launcher的onClick方法，在onClick里面会执行我们Activity的startActivitySafely方法，因为Launcher是继承自Activity的嘛，所以它里面的大多数方法都会辗转到Activity中去执行的，在startActivitySafely方法里面实际上执行的就是startActivity方法了，这个方法就是我们通常跳转Activity的时候调用的方法，这个方法里面会执行startActivityForResult方法，紧接着执行的是Instrumentation的execStartActivity方法，Instrumentation可以认为是一个大管家，每个Activity都持有一个Instrumentation对象的一个引用，但是整个进程中是只有一个Instrumentation的，AMS与ActivityThread之间诸如Activity的创建、暂停等的交互工作实际上是由Instrumentation具体操作的，在这里我们应该明白一点就是我们的ActivityManagerService是属于系统级服务的，而ActivityThread是用于管理应用程序的主线程执行的，而系统级服务和应用程序的ActivityThread是属于不同进程的，因此他们两者之间的交互是需要用到Binder通信的，而ActivityThread工作的真正执行者是Instrumentation，因此Instrumentation与AMS的交互是涉及到Binder通信的，也就是说Instrumentation的execStartActivity实际上是通过Binder与AMS进行通信的，在execStartActivity里面通过获得AMS在客户端的代理   ActivityManagerProxy，调用他的startActivity方法实际上最终执行到的就是AMS的startActivity方法，在AMS的startActivity方法中实际上通过ActivityStackSupervision与ActivityStack来交互操作完成Activity启动的，最终会执行到ApplicationThread的scheduleLaunchActivity方法来启动Activity，scheduleLaunchActivity会通过Handler来发送创建Activity的消息给主线程，也就是ActivityThread，而ActivityThread在接收到这个创建消息之后会调用他自己的handleLaunchActivity方法，这个方法会执行performLaunchActivity方法，在performLaunchActivity方法里面就会通过Instrumentation通过反射来创建一个Activity对象出来了，在创建完Activity之后就会调用Instrumentation的callActivityOnCreate方法来启动Activity，callActivityOnCreate实际上执行的是Activity的onCreate方法，进行一些必要的初始化操作；

1. 通过 Launcher 启动应用时，点击应用图标后，Launcher 调用 startActivity 启动应用。
2. Launcher Activity 最终调用 Instrumentation 的 execStartActivity 来启动应用。
3. Instrumentation 调用 ActivityManagerProxy (ActivityManagerService 在应用进程的一个代理对象) 对象的 startActivity 方法启动 Activity。
4. 到目前为止所有过程都在 Launcher 进程里面执行，接下来 ActivityManagerProxy 对象跨进程调用 ActivityManagerService (运行在 system\_server 进程)的 startActivity 方法启动应用。
5. ActivityManagerService 的 startActivity 方法经过一系列调用，最后调用 zygoteSendArgsAndGetResult 通过 socket 发送给 zygote 进程，zygote 进程会孵化出新的应用进程。
6. zygote 进程孵化出新的应用进程后，会执行 ActivityThread 类的 main 方法。在该方法里会先准备好 Looper 和消息队列，然后调用 attach 方法将应用进程绑定到 ActivityManagerService，然后进入 loop 循环，不断地读取消息队列里的消息，并分发消息。
7. ActivityManagerService 保存应用进程的一个代理对象，然后 ActivityManagerService 通过代理对象通知应用进程创建入口 Activity 的实例，并执行它的生命周期函数。

# Android应用程序编译过程简述

* 首先，资源文件会被资源打包工具（aapt:Android Asset Packaging Tool）编译压缩；
* 然后Java代码包括src以及自动生成的R.java文件会被Java编译器编译为Java字节码文件；
* 接着通过交叉编译为适合在Android虚拟机上运行的.dex文件；
* 最后根据.dex文件和已编译的资源文件创建APK文件。

# Intent

**一.Intent介绍：**

      Intent的中文意思是“意图，意向”，在Android中提供了Intent机制来协助应用间的交互与通讯，Intent负责对应用中一次操作的动 作、动作涉及数据、附加数据进行描述，Android则根据此Intent的描述，负责找到对应的组件，将 Intent传递给调用的组件，并完成组件的调用。Intent不仅可用于应用程序之间，也可用于应用程序内部的Activity/Service之间的 交互。因此，可以将Intent理解为不同组件之间通信的“媒介”专门提供组件互相调用的相关信息。

**二、Intent作用(3种)：**

      Intent 是一个将要执行的动作的抽象的描述，一般来说是作为参数来使用，由Intent来协助完成android各个组件之间的通讯。比如说调用 startActivity()来启动一个activity,或者由broadcaseIntent()来传递给所有感兴趣的 BroadcaseReceiver, 再或者由startService()/bindservice()来启动一个后台的service.所以可以看出来，intent主要是用来启动其他的 activity 或者service，所以可以将intent理解成activity之间的粘合剂。

**四.Intent的几个重要属性，下面进行详解：**

      动作(Action),数据(Data),分类(Category),类型(Type),组件(Compent)以及扩展信(Extra)。其中最常用的是Action属性和Data属性。 Flag是用来决定activity在 Activity栈中的存放方式

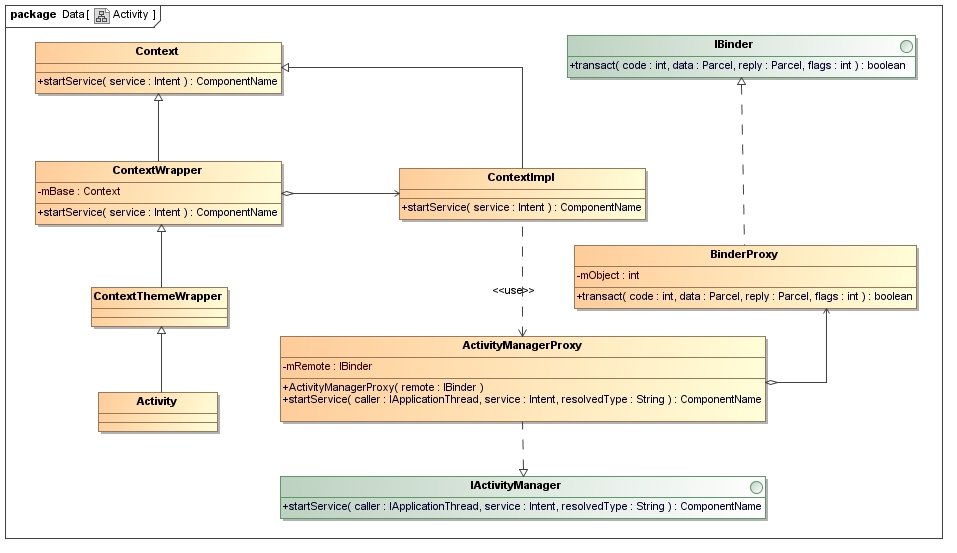
### 1.显式Intent

指定了component属性的Intent(调用setComponent(ComponentName)或者setClass(Context, Class)来指定)。通过指定具体的组件类，通知应用启动对应的组件。通过指定目标组件名字，寻找目标组件，一般应用于调用程序和目标组件在同一个应用中的情况；   
显式Intent通过指定组件来直接调用的，Intent中可以通过下面的方式实现显式Intent

### 2.隐式Intent

隐式Intent对象是没有指定mComponent的Intent，也就是说，在构建Intent的过程中，不会通过Intent intent=new Intent(MainActivity.this,OtherActivity.class);， setComponent(ComponentName name)``setClassName(String packageName,String classNameInThatPackage)``setClass(Context context,Class classObjectInThatContext)等方式来指定ComponentName对象，注意是不会，不会，不会，重要的事情说3遍。   
当没有指定mComponent对象时，需要足够的信息才能够匹配到目标组件，这些信息包括**Action,Category,Data**。这3个信息需要当前组件的Intent和目标组件的Intent都配置，只要当两个组件的3个信息匹配时，才会启动目标组件，当然也可能会同时匹配到多个目标组件，这时就需要用户自己去选择，如下边的栗子，就是一个Intent匹配到了多个目标组件。这种方式不仅可以启动本应用中的相关组件，也可以启动其他应用的相关组件，举个栗子，当我们的手机中安装了多个浏览器时，比如UC,百度等，当我们在某个应用中点击某个链接时，是不是会跳出一个选择框来让你选择使用哪个浏览器来打开链接（如果没有弹出选择框，有两个可能：1.你的手机只有一个浏览器。2.你之前勾选了默认使用该浏览器打开链接）。好了，栗子举完了。来说正事，这种方式就是通过隐式Intent来设置的。那我们要怎么设置来启动另外一个组件呢？继续看。

## Intent 启动一个activity

1、由主进程通过Binder进入AMS进程   
2、 在AMS进程获取和管理要启动的服务的相关信息   
3、 由AMS进0程回到该主进程启动服务

# ListView 和 recyclerview

## Android中ListView的几种常见的优化方法

优化一：

也是最普通的优化，就在MyAdapter类中的getView方法中，我们注意到，上面的写法每次需要一个View对象时，都是去重新inflate一个View出来返回去，没有实现View对象的复用，而实际上对于ListView而言，只需要保留能够显示的最大个数的view即可，其他新的view可以通过复用的方式使用消失的条目的view，而getView方法里也提供了一个参数：convertView，这个就代表着可以复用的view对象，当然这个对象也可能为空，当它为空的时候，表示该条目view第一次创建，所以我们需要inflate一个view出来,所以在这里，我们使用下面这种方式来重写getView方法：

优化二：

上面是对view对象的复用做的优化，我们经过上面的优化之后，我们不需要每一个view都重新生成了。下面我们来解决下一个每一次都需要做的工作，那就是view中组件的查找：

**TextView tv\_item = (TextView) view.findViewById(R.id.tv\_item);**

实际上，findViewById是到xml文件中去查找对应的id，可以想象如果组件多的话也是挺费事的，如果我们可以让view内的组件也随着view的复用而复用，那该是多美好的一件事啊。。实际上谷歌也推荐了一种优化方法来做应对，那就是重新建一个内部静态类，里面的成员变量跟view中所包含的组件个数类型相同，我们这里的view只包含了一个TextView，所以我们的这个静态类如下：

那么这个viewHolder类我们要如何使用才可以达到复用效果呢？基本思路就是在convertView为null的时候，我们不仅重新inflate出来一个view，并且还需要进行findviewbyId的查找工作，但是同时我们还需要获取一个ViewHolder类的对象，并将findviewById的结果赋值给ViewHolder中对应的成员变量。最后将holder对象与该view对象“绑”在一块。

当convertView不为null时，我们让view=converView，同时取出这个view对应的holder对象，就获得了这个view对象中的TextView组件，它就是holder中的成员变量，这样在复用的时候，我们就不需要再去findViewById了，只需要在最开始的时候进行数次查找工作就可以了。这里的关键在于如何将view与holder对象进行绑定，那么就需要用到两个方法：setTag和getTag方法了：

优化三：

上面的两个例子中ListView都是显示的本地的List集合中的内容，List的长度也只有100个，我们可以毫不费力一次性加载完这100个数据；但是实际应用中，我们往往会需要使用Listview来显示网络上的内容，比如说我们拿使用ListView显示新闻为例：

其一：假如网络情况很好，我们使用的手机也许能够一下子加载完所有新闻数据，然后显示在ListView中，用户可能感觉还好，假如说在网络不太顺畅的情况下，用户加载完所有网络的数据，可能这个list是1000条新闻，那么用户可能需要面对一个空白的Activity好几分钟，这个显然是不合适的

其二：我们知道Android虚拟机给每个应用分配的运行时内存是一定的，一般性能不太好的机器只有16M，好一点的可能也就是64M的样子，假如说我们现在要浏览的新闻总数为一万条，即便是网络很好的情况下，我们可以很快的加载完毕，但是多数情况下也会出现内存溢出从而导致应用崩溃的情况。

缓存机制：[链接](https://mp.weixin.qq.com/s?__biz=MzA3NTYzODYzMg==&mid=2653578065&idx=2&sn=25e64a8bb7b5934cf0ce2e49549a80d6&chksm=84b3b156b3c43840061c28869671da915a25cf3be54891f040a3532e1bb17f9d32e244b79e3f&scene=21" \l "wechat_redirect)

[链接2](https://blog.csdn.net/qq_17093685/article/details/78617913?locationNum=3&fps=1)

<https://kymjs.com/code/2016/07/10/01/>

**优化：**

图片上可以通过滑动监听，设置flag，当滑动时，加载默认的view，当滑动停止时，通过getChildCount()，和getChildAt(i)来重新加载当前页面的值。

对于使用了Picasso的来说，只要设置picasso的pauseTag和resumeTag即可

分批加载，Recyclerview里有state来记录当前滑动状态，可以判断是否滑动到底部idle

This [RecyclerView](https://developer.android.com/reference/android/support/v7/widget/RecyclerView.html) model does a lot of optimization work so you don't have to:

* When the list is first populated, it creates and binds some view holders on either side of the list. For example, if the view is displaying list positions 0 through 9, the [RecyclerView](https://developer.android.com/reference/android/support/v7/widget/RecyclerView.html) creates and binds those view holders, and might also create and bind the view holder for position 10. That way, if the user scrolls the list, the next element is ready to display.
* As the user scrolls the list, the [RecyclerView](https://developer.android.com/reference/android/support/v7/widget/RecyclerView.html) creates new view holders as necessary. It also saves the view holders which have scrolled off-screen, so they can be reused. If the user switches the direction they were scrolling, the view holders which were scrolled off the screen can be brought right back. On the other hand, if the user keeps scrolling in the same direction, the view holders which have been off-screen the longest can be re-bound to new data. The view holder does not need to be created or have its view inflated; instead, the app just updates the view's contents to match the new item it was bound to.
* When the displayed items change, you can notify the adapter by calling an appropriate [RecyclerView.Adapter.notify…()](https://developer.android.com/reference/android/support/v7/widget/RecyclerView.Adapter.html" \l "notifyItemChanged(int)) method. The adapter's built-in code then rebinds just the affected items.

## scrollListener

## 加速的方法

RecyclerView: notifyDataSetChanged [DiffUtil](https://developer.android.com/reference/android/support/v7/util/DiffUtil.html)

Sortedlist：有改变的回调函数，能进行批量修改，即定义完之后一起修改

RecyclerView: Nested RecyclerViews

RecyclerView: Too much inflation / Create taking too lon

# android进程优先级，内存不足时杀进程顺序

Android操作系统尝试尽可能长时间的保持应用的进程，但当可用内存很低时最终要移走一部分进程。怎样确定那些程序可以运行，那些要被销毁，Android让每一个进程在一个重要级的基础上运行，重要级低的进程最有可能被淘汰，**一共有5级**，下面这个列表就是按照重要性排列的：

1 一个前台进程显示的是用户此时需要处理和显示的。下列的条件有任何一个成立，这个进程都被认为是在前台运行的。  
        a 与用户正发生交互的（activity显示在最前台）。  
        b 有前台service(和通知栏相关，调用过startForeground）。  
        c有活跃的service(有一个正在调用生命周期的回调函数的service（如onCreate()、onStar()、onDestroy()）)  
        d有活跃的广播接受者（ 它有一个正在运行onReceive()方法的广播接收对象）。  
只有少数的前台进程可以在任何给定的时间内运行，销毁他们是系统万不得已的、最后的选择——当内存不够系统继续运行下去时。通常，在这一点上，设备已经达到了内存分页状态，所以杀掉一些前台进程来保证能够响应用户的需求。  
  
2 一个可用进程没有任何前台组件，但它仍然可以影响到用户的界面。下面两种情况发生时，可以称该进程为可用进程。  
        它是一个非前台的activity，但对用户仍然可用（onPause()方法已经被调用）这是可能发生的，例如：前台的activity是一个允许上一个activity可见的对话框，即当前activity半透明，能看到前一个activity的界面，它是一个服务于可用activity的服务。  
  
3 一个服务进程是一个通过调用startService()方法启动的服务，并且不属于前两种情况。尽管服务进程没有直接被用户看到，但他们确实是用户所关心的，比如后台播放音乐或网络下载数据。所以系统保证他们的运行，直到不能保证所有的前台可见程序都正常运行时才会终止他们。  
  
4 一个后台进程就是一个非当前正在运行的activity（activity的onStop()方法已经被调用），他们不会对用户体验造成直接的影响，当没有足够内存来运行前台可见程序时，他们将会被终止。通常，后台进程会有很多个在运行,所以他们维护一个LRU最近使用程序列表来保证经常运行的activity能最后一个被终止。如果一个activity正确的实现了生命周期的方法，并且保存它当前状态，杀死这些进程将不会影响到用户体验。  
  
5 一个空线程没有运行任何可用应用程序组，保留他们的唯一原因是为了设立一个缓存机制，来加快组件启动的时间。系统经常杀死这些内存来平衡系统的整个系统的资源，进程缓存和基本核心缓存之间的资源。  
Android把进程里优先级最高的activity或服务，作为这个进程的优先级。例如，一个进程拥有一个服务和一个可见的activity，那么这个进程将会被定义为可见进程，而不是服务进程。  
  
此外，如果别的进程依赖某一个进程的话，那么被依赖的进程会提高优先级。一个进程服务于另一个进程，那么提供服务的进程不会低于获得服务的进程。例如，如果进程A的一个内容提供商服务于进程B的一个客户端，或者进程A的一个service被进程B的一个组件绑定，那么进程A至少拥有和进程B一样的优先级，或者更高。

因为一个运行服务的进程的优先级高于运行后台activity的进程，一个activity会准备一个长时间运行的操作来启动一个服务，而不是启动一个线程–尤其是这个操作可能会拖垮这个activity。例如后台播放音乐的同时，通过照相机向服务器发送一张照片，启动一个服务会保证这个操作至少运行在service 进程的优先级下。

所以广播的onReceive里如果要做耗时处理，有2种方法

1.直接启线程，在线程里做耗时操作

2.启service，在service里启线程，在线程里做耗时操作

第二种方法至少可以让进程保持服务进程的优先级，不容易被杀死，所以第二种方法好一些

# OOM

一、什么是OOM

OOM（out of memory）即内存泄露。一个程序中，已经不需要使用某个对象，但是因为仍然有引用指向它垃圾回收器就无法回收它，当该对象占用的内存无法被回收时，就容易造成内存泄露。

Android的一个应用程序的内存泄露对别的应用程序影响不大，因为为了能够使得Android应用程序安全且快速的运行，Android的每个应用程序都会使用一个专有的Dalvik虚拟机实例来运行，也就是说每个应用程序都是在属于自己的进程中运行的。如果程序内存溢出，Android系统只会kill掉该进程，而不会影响其他进程的使用（如果是system\_process等系统进程出问题的话，则会引起系统重启）。

二、出现内存泄露原因

1.资源对象没关闭造成的内存泄露，try catch finally中将资源回收放到finally语句可以有效避免OOM。资源性对象比如：

1-1，Cursor

1-2，调用registerReceiver后未调用unregisterReceiver()

1-3，未关闭InputStream/OutputStream

1-4，Bitmap使用后未调用recycle()

2.作用域不一样，导致对象不能被垃圾回收器回收，比如：

2-1，非静态内部类会隐式地持有外部类的引用，

2-2，Context泄露

概括一下，避免Context相关的内存泄露，记住以下事情：

1、 不要保留对Context-Activity长时间的引用（对Activity的引用的时候，必须确保拥有和Activity一样的生命周期）

2、尝试使用Context-Application来替代Context-Activity 3、如果你不想控制内部类的生命周期，应避免在Activity中使用非静态的内部类，而应该使用静态的内部类，并在其中创建一个对Activity的弱引用。

这种情况的解决办法是使用一个静态的内部类，其中拥有对外部类的WeakReference。

2-3，Thread 引用其他对象也容易出现对象泄露。

2-4，onReceive方法里执行了太多的操作

3.内存压力过大

3-1，图片资源加载过多，超过内存使用空间，例如Bitmap 的使用

3-2，重复创建view，listview应该使用convertview和viewholder

三、如何避免内存泄露

1.使用缓存技术，比如LruCache、DiskLruCache、对象重复并且频繁调用可以考虑对象池

2.对于引用生命周期不一样的对象，可以用软引用或弱引用SoftReferner WeakReferner

3.对于资源对象 使用finally 强制关闭

4.内存压力过大就要统一的管理内存

# ANR

An ANR will be triggered for your app when one of the following conditions occur:

* While your activity is in the foreground, your app has not responded to an input event or [BroadcastReceiver](https://developer.android.com/reference/android/content/BroadcastReceiver.html) (such as key press or screen touch events) within 5 seconds.
* While you do not have an activity in the foreground, your [BroadcastReceiver](https://developer.android.com/reference/android/content/BroadcastReceiver.html) hasn't finished executing within a considerable amount of time.

There are some common patterns to look for when diagnosing ANRs:

1. The app is doing slow operations involving I/O on the main thread.
2. The app is doing a long calculation on the main thread.
3. The main thread is doing a synchronous binder call to another process, and that other process is taking a long time to return.
4. The main thread is blocked waiting for a synchronized block for a long operation that is happening on another thread.
5. The main thread is in a deadlock with another thread, either in your process or via a binder call. The main thread is not just waiting for a long operation to finish, but is in a deadlock situation. For more information, see [Deadlock](https://en.wikipedia.org/wiki/Deadlock) on Wikipedia.

The following techniques can help you find out which of these causes is causing your ANRs.

#### **Strict mode**

Using [StrictMode](https://developer.android.com/reference/android/os/StrictMode.html) helps you find accidental I/O operations on the main thread while you’re developing your app. You can use [StrictMode](https://developer.android.com/reference/android/os/StrictMode.html) at the application or activity level.

#### **Enable background ANR dialogs**

Android shows ANR dialogs for apps that take too long to process the broadcast message only if **Show all ANRs** is enabled in the device’s **Developer options**. For this reason, background ANR dialogs are not always displayed to the user, but the app could still be experiencing performance issues.

#### **Traceview**

You can use Traceview to get a trace of your running app while going through the use cases and identify the places where the main thread is busy. For information about how to use Traceview, see [Profiling with Traceview and dmtracedump](https://developer.android.com/studio/profile/traceview.html).

#### **Pull a traces file**

Android stores some trace information when it experiences an ANR in the /data/anr/traces.txt file on the device. You can get the file from an emulator by starting a shell session on the device as the root user, as shown in the following command-line example:

# Jank

Pinpointing the code in your app that is causing jank can be difficult. This section describes three methods for identifying jank:

* [Visual inspection](https://developer.android.com/topic/performance/vitals/render.html#visual-inspection)
* [Systrace](https://developer.android.com/topic/performance/vitals/render.html#systrace)
* [Custom performance monitoring](https://developer.android.com/topic/performance/vitals/render.html#custom-monitoring)

**Visual inspection**lets you quickly run through all the use-cases in your app in a few minutes, but it doesn't provide as much detail as Systrace. **Systrace**provides more details, but if you ran Systrace for all the use-cases in your app, you'd be flooded with so much data that it'd be difficult to analyze. Both visual inspection and systrace detect jank on your local device. If your jank cannot be reproduced on local devices, you can build **custom performance monitoring** to measure specific parts of your app on devices running in the field.

# 优化

## 搜索优化

**Offloading work onto background threads**

**Asynchronous view inflation(asynclayoutinflater)**

One example of duplicate work is painting white pixels on top of white pixels. If the background of the application is white by default, and each of our list items also has a white background, then we’re making the GPU do work that it doesn’t have to do.

**Reducing overdraw and layout hierarchy depth**

## 图片优化：

**使用webP格式**

## 使用异步

* 保持APP的高度响应，不要在UI线程做耗时操作，多使用异步任务
* 使用线程时要做好线程控制；使用队列、线程池
* 谨慎使用糟糕的AysncTask、Timer
* 警惕异步任务引起的内存泄露
* 应该异步任务分类，比如HTTP，图片下载，文件读写，每一类的异步任务维护一个任务队列，而不是每一个任务都开一个线程（Volley表示我一个可以搞定这些全部 \_(:з」∠)\_）
* 这些常用的任务应该做好优先级处理（一般JSON数据优先于图片等静态数据的请求）
* 一般异步任务应该开启一个SingleAsyncTask，保证一时只有一个线程在工作
* HTTP和图片下载尽量使用同一套网络请求
* 使用MVP模式规范大型Activity类的行为，避免异步任务造成的内存泄露

## 避免内存泄露

* 了解虚拟机内存回收机制
* 频繁GC也会造成卡顿，避免不必要的内存开销
* 错误的引用姿♂势造成的内存泄露（啊~要泄了~）
* 常见的Activity泄露（单例、Application、后台线程、无限动画、静态引用）
* Bitmap泄露（HoneyComb这个问题之前压力好大）
* 尽量使用IntentService代替Service，前者会自动StopItself
* 排查内存泄露问题的方法（我一直以来都是简单暴力的人肉dump检查大法）
* 使用LeakCanary自动检查Activity泄露问题
* 对内存负载要保持敏感（Sharp）

## 视图优化

* 布局优化、减少层次，Include Merge
* 使用ViewStub避免不必要的LayoutInflate，使用GONE代替重复LayoutInflate同一个布局
* 避免过度绘制，应该减少不必要的布局背景；布局层次太深会造成过度绘制以及Measure、Layout等方法时间复杂度的指数增长
* 使用过渡动画，比如给图片的呈现加一个轻量的淡入效果会让视觉上变得流畅许多
* 避免过度的动画，不要让一个界面同时出现多出动画，比如List滚动时Item项要停止播放动画或者GIF
* 复杂动画使用SurfaceView或TextureView
* 尽量提供多套分辨率的图片，使用矢量图

## Adapter优化

* 复用convertView，用ViewHolder代替频繁findViewById
* 不要重复setListener，要使用v.getId来复用Listener，不然会创建一堆Listener导致频繁GC
* 多布局要采用MutilItemView，而不是使用一个大布局然后动态控制需要现实的部分
* 不要在getView方法做做耗时的操作
* 快速滚动列表的时候，可以停止加载列表项的图片，停止列表项的动画，不要在这时候改变列表项的布局
* 尽量用RecyclerView（增量Notify和RecycledViewPool带你飞）

## 代码优化

* 算法优化，减少时间复杂度，参考一些经典的优化算法
* 尽量使用int，而不是float或者double
* 尽量采用基本类型，避免无必要的自动装箱和拆箱，浪费时间和空间
* 选用合适的集合类（尽量以空间换时间）、选用Android家的SparseArray,SparseBooleanArray和LongSparseArray
* 避免创建额外的对象（StringBuilder）
* 使用SO库完成一些比较独立的功能（高斯模糊）
* 预处理（提前操作）一些比较耗时的初始化工作统一放到启动图处理
* 懒加载（延迟处理）规避Activity的敏感生命周期
* Log工具类，要在编译时删掉调试代码，而不是在运行时通过判断条件规避
* 优先使用静态方法、公有方法还是私有方法？速度区别很大哦
* 类内部直接对成员变量进行操作，不要使用getter/setter方法，调用方法耗额外的时间
* 给内部类访问的外部类成员变量要声明称包内可访问，而不是私有，不然编译的时候还是会自动创建用于访问外部类成员变量的方法
* 遍历集合时，使用i++代替Iterator，后者需要额外的对象操作，应在循环体内避免这种情况
* 如果一个基本类型或者String的值不会改变，尽量用final static，编译时会直接用变量的值替换变量，也就不需要在查询变量的值了

## 其他优化

* 数据库优化：使用索引、使用异步线程
* 网络优化 …… 一堆优秀的轮子
* 避免过度使用依赖注入框架，大量的反射
* 不过过度设计/抽象，多态看起来很有设计感，代价就是额外的代码、空间、时间
* 尽量不要开启多进程，进程的开销很大

## APK瘦身

* 开启混淆
* 使用zipalign工具优化APK
* 适当有损图片压缩、使用矢量图
* 删除项目中冗余的资源，之前写过一些删除没有res资源的脚本
* 动态加载模块化，项目拆分啊！

## 性能问题的排查方法

* GPU条形图，没事开来看看淘宝
* 过度绘制颜色，嗯，不要一篇姨妈红就好
* LeakCanary，自动检测Activity泄露，挺好用的
* TraceView（Device Monitor），Systrace，分析哪些代码占用的CPU时间太大，屡试不爽
* Lint，检查不合理的res资源
* layoutopt（还是optlayout？），对当前布局提出优化建议，已被lint替代，但是还能用
* HierarchyViewer，查看手机当前界面的布局层次，布局优化时常用（只用于模拟器，真机上用要ROOT，不想ROOT加得使用ViewServer）
* StrictMode，UI操作、网络操作等容易出现性能问题的地方，如果出现异常情况StrictMode会报警

## 自定义View 优化

<一>降低刷新频率

为了提高view的运行速度，减少来自于频繁调用的程序的不必要的代码。从onDraw()方法开始调用，这会给你带来最好的回报。特别地，在onDraw()方法中你应该减少冗余代码，冗余代码会带来使你view不连贯的垃圾回收。初始化的冗余对象，或者动画之间的，在动画运行时，永远都不会有所贡献。

加之为了使onDraw()方法更有依赖性，你应该尽可能的不要频繁的调用它。大部分时候调用 onDraw()方法就是调用invalidate()的结果，所以**减少不必要的调用invalidate()方法。**有可能的，调用四种参数不同类型的invalidate()，而不是调用无参的版本。无参变量需要刷新整个view，而四种参数类型的变量只需刷新指定部分的view.这种高效的调用更加接近需求，也能减少落在矩形屏幕外的不必 要刷新的页面。

另外一个非常耗时的操作是请求layout。**任何时候执行requestLayout()，会使得Android UI系统去遍历整个View的层级来计算出每一个view的大小**。如果找到有冲突的值，它会需要重新计算好几次。另外需要尽量保持View的层级是扁平化的，这样对提高效率很有帮助。  如果你有一个复杂的UI，你应该写一个自定义的ViewGroup类来表现它的布局。不同于内置的 view类，你的自定义view能关于尺寸和它子控件的形状做出应用特定的假想，同时避免通过它子类来 计算尺寸。这圆图例子显示怎样作为自定义view一部分来继承ViewGroup类，圆图有它子view类， 但是重来都不测量他们。相反地，它直接地通过自己自定义的布局算法来设定他们的尺寸大小。

  如果你有一个复杂的UI，你应该考虑写一个自定义的ViewGroup来执行他的layout操作。与内置的view不同，自定义的view可以使得程序仅仅测量这一部分，这避免了遍历整个view的层级结构来计算大小。这个PieChart 例子展示了如何继承ViewGroup作为自定义view的一部分。PieChart 有子views，但是它从来不测量它们。而是根据他自身的layout法则，直接设置它们的大小。

<二>使用硬件加速

作为Android3.0，Android2D图表系统可以通过大部分新的Android装置自带GPU（图表处理单元）来增加，对于许多应用程序 来说，GPU硬件加速度能带来巨大的性能增加，但是对于每一个应用来讲，并不都是正确的选择。Android框架层更好地为你提供了控制应用程序部分硬件 是否增加的能力。

怎样在你的应用，活动，或者窗体级别中使用加速度类，请查阅Android开发者指南中的Hardware Acceleration类。注意到在开发者指南中的附加说明，你必须在你的AndroidManifest.xml 文件中的<uses-sdk android:targetSdkVersion="11"/>中将应用目标API设置到11或者更高的级别。

一旦你使用硬件加速度类，你可能没有看到性能的增长，手机GPUs非常擅长某些任务，例如测量，翻转，和平移位图类的图片。特别地，他们不擅长其他的任务，例如画直线和曲线。为了利用GPU加速度类，你应该增加GPU擅长的操作数量，和减少GPU不擅长的操作数量。

在PieChart 例子中，例如，画一个饼图相对来说是比较麻烦的。每次重新画饼图，它的翻转都会给UI界面带来不流畅的感觉。解决方案就是在子View类中放置饼图和将View的布局类型设置为 LAYER\_TYPE\_HARDWARE，**这样GPU能将它作为静态图片缓存。样例将子view作为PieChart的内部类来定义，这样减少了来自于需要实现方案的大量代码的改动。**

<三>初始化时创建对象；不要在onDraw方法内创建绘制对象，一般都在构造函数里面初始化对象；

《四》状态的存储与恢复：如果内存不足，而恰好我们的Activity置于后台，不幸被重启，或者用户旋转屏幕造成Activity重启，我们的View应该也能尽可能的去保存自己的属性。

# [px 与 dp, sp换算公式？](http://www.cnblogs.com/bluestorm/p/3640786.html)

# JVM和ART

#### https://upload-images.jianshu.io/upload_images/2466095-37c05f531dbb640d.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/700即时（JIT）

通过 Dalvik 即时编译器，每次app运行的时候，它动态的将一部分 Dalvik 字节码 解释为机器码。随着执行过程的进行，更多的字节码被编译和缓存。因此 JIT 只编译了一部分代码，它具有更小的内存占用和更少的设备物理空间占用。

#### 提前（AOT）

ART 使用的是提前型编译器。在app的安装时期，它静态的将 DEX 字节码转化为机器码并存储在设备存储空间中。这是发生在安装过程中的一次性事件。由于不需要 JIT 编译，代码执行的更快了。

由于 ART 直接本地执行机器码，它不会像 Dalvik 一样频繁的访问 Cpu，更少的 Cpu 使用也降低了电量消耗。

**执行的字节码不同**   
在Java SE程序中，Java类会被编译成一个或多个.class文件，打包成jar文件，而后JVM会通过相应的.class文件和jar文件获取相应的字节码。执行顺序为： .java文件 -> .class文件 -> .jar文件   
而DVM会用dx工具将所有的.class文件转换为一个.dex文件，然后DVM会从该.dex文件读取指令和数据。执行顺序为：   
.java文件 –>.class文件-> .dex文件

如上图所示，.jar文件里面包含多个.class文件，每个.class文件里面包含了该类的常量池、类信息、属性等等。当JVM加载该.jar文件的时候，会加载里面的所有的.class文件，JVM的这种加载方式很慢，对于内存有限的移动设备并不合适。   
而在.apk文件中只包含了一个.dex文件，这个.dex文件里面将所有的.class里面所包含的信息全部整合在一起了，这样再加载就提高了速度。.class文件存在很多的冗余信息，dex工具会去除冗余信息，并把所有的.class文件整合到.dex文件中，减少了I/O操作，提高了类的查找速度。

# RxJava

Rxjava源码解析：

链接：<https://www.jianshu.com/p/a488318dec0e>

## 事件产生

|  |
| --- |
| **private** void doSomeWork() {  Observable<String> observable= Observable.create(**new** ObservableOnSubscribe<String>() {  **@Override**  **public** void subscribe(ObservableEmitter<String> e) **throws** Exception {  e.onNext("a");  e.onComplete();  }  });  Observer observer = **new** Observer<String>() {  **@Override**  **public** void onSubscribe(Disposable d) {  Log.i("lx", " onSubscribe : " + d.isDisposed());  }  **@Override**  **public** void onNext(String str) {  Log.i("lx", " onNext : " + str);  }  **@Override**  **public** void onError(Throwable e) {  Log.i("lx", " onError : " + e.getMessage());  }  **@Override**  **public** void onComplete() {  Log.i("lx", " onComplete");  }  };  observable.subscribe(observer);  } |

**@SchedulerSupport**(SchedulerSupport.NONE)

**public** **static** <T> Observable<T> create(ObservableOnSubscribe<T> source) { *// ObservableOnSubscribe 是个接口，只包含subscribe方法，是事件生产的源头。*

ObjectHelper.requireNonNull(source, "source is null"); *// 判空*

**return** RxJavaPlugins.onAssembly(**new** ObservableCreate<T>(source));

}

这里关键的一点是create方法实际上创建了ObservableCreate对象(OnAssembly实际是一个钩子对象)

**public** **final** void subscribe(Observer<? **super** T> observer) {

ObjectHelper.requireNonNull(observer, "observer is null");

......

observer = RxJavaPlugins.onSubscribe(**this**, observer); *// hook ，默认直接返回observer*

......

subscribeActual(observer); *// 这个才是真正实现订阅的方法。*

......

}

*// subscribeActual 是抽象方法，所以需要到实现类中去看具体实现，也就是说实现是在上文中提到的ObservableCreate中*

**protected** **abstract** void subscribeActual(Observer<? **super** T> observer);

**public** ObservableCreate(ObservableOnSubscribe<T> source) {

**this**.source = source; *// 事件源，生产事件的接口，由我们自己实现*

}

**@Override**

**protected** void subscribeActual(Observer<? **super** T> observer) {

CreateEmitter<T> parent = **new** CreateEmitter<T>(observer); *// 发射器*

observer.onSubscribe(parent); *//直接回调了观察者的onSubscribe*

**try** {

*// 调用了事件源subscribe方法生产事件，同时将发射器传给事件源。*

*// 现在我们明白了，数据源生产事件的subscribe方法只有在observable.subscribe(observer)被执行后才执行的。 换言之，事件流是在订阅后才产生的。*

*//而observable被创建出来时并不生产事件，同时也不发射事件。*

source.subscribe(parent);

} **catch** (Throwable ex) {

Exceptions.throwIfFatal(ex);

parent.onError(ex);

}

}

**public** void onNext(T t) {

**if** (!isDisposed()) { *// 判断事件是否需要被丢弃*

observer.onNext(t); *// 调用Emitter的onNext，它会直接调用observer的onNext*

}

}

**public** void onError(Throwable t) {

**if** (!isDisposed()) {

**try** {

observer.onError(t); *// 调用Emitter的onError，它会直接调用observer的onError*

} **finally** {

dispose(); *// 当onError被触发时，执行dispose(), 后续onNext，onError， onComplete就不会继续发射事件了*

}

}

}

**@Override**

**public** void onComplete() {

**if** (!isDisposed()) {

**try** {

observer.onComplete(); *// 调用Emitter的onComplete，它会直接调用observer的onComplete*

} **finally** {

dispose(); *// 当onComplete被触发时，也会执行dispose(), 后续onNext，onError， onComplete同样不会继续发射事件了*

}

}

}

## 线程调度：

Schedulers.io() ew IoScheduler()后Rxjava会创建CachedWorkerPool的线程池，同时也创建并运行了一个名为RxCachedWorkerPoolEvictor的清除线程，主要作用是清除不再使用的一些线程。

但目前只创建了线程池并没有实际的thread，所以Schedulers.io()相当于只做了线程调度的前期准备。

**@Override**

**public** void subscribeActual(**final** Observer<? **super** T> s) {

**final** SubscribeOnObserver<T> parent = **new** SubscribeOnObserver<T>(s);

*// 没有任何线程调度，直接调用的，所以下游的onSubscribe方法没有切换线程，*

*//本文demo中下游就是观察者，所以我们明白了为什么只有onSubscribe还运行在main线程*

s.onSubscribe(parent);

parent.setDisposable(scheduler.scheduleDirect(**new** SubscribeTask(parent)));

}

*// 这个类很简单，就是一个Runnable，最终运行上游的subscribe方法*

**final** **class** **SubscribeTask** **implements** Runnable {

**private** **final** SubscribeOnObserver<T> parent;

SubscribeTask(SubscribeOnObserver<T> parent) {

**this**.parent = parent;

}

**@Override**

**public** void run() {

source.subscribe(parent);

}

}

**@NonNull**

**public** Disposable scheduleDirect(**@NonNull** Runnable run, long delay, **@NonNull** TimeUnit unit) {

*// IoSchedular 中的createWorker()*

**final** Worker w = createWorker();

*// hook decoratedRun=run;*

**final** Runnable decoratedRun = RxJavaPlugins.onSchedule(run);

*// decoratedRun的wrap，增加了Dispose的管理*

DisposeTask task = **new** DisposeTask(decoratedRun, w);

*// 线程调度*

w.schedule(task, delay, unit);

**return** task;

}

**public** Worker createWorker() {

*// 工作线程是在此时创建的*

**return** **new** EventLoopWorker(pool.get());

}

**public** Disposable schedule(**@NonNull** Runnable action, long delayTime, **@NonNull** TimeUnit unit) {

**if** (tasks.isDisposed()) {

*// don't schedule, we are unsubscribed*

**return** EmptyDisposable.INSTANCE;

}

*// action 中就包含上游subscribe的runnable*

**return** threadWorker.scheduleActual(action, delayTime, unit, tasks);

}

 Schedulers.io()等价于 new IoScheduler()

 new IoScheduler() Rxjava创建了线程池，为后续创建线程做准备，同时创建并运行了一个清理线程RxCachedWorkerPoolEvictor，定期执行清理任务。

 subscribeOn()返回一个ObservableSubscribeOn对象，它是Observable的一个装饰类，增加了scheduler。

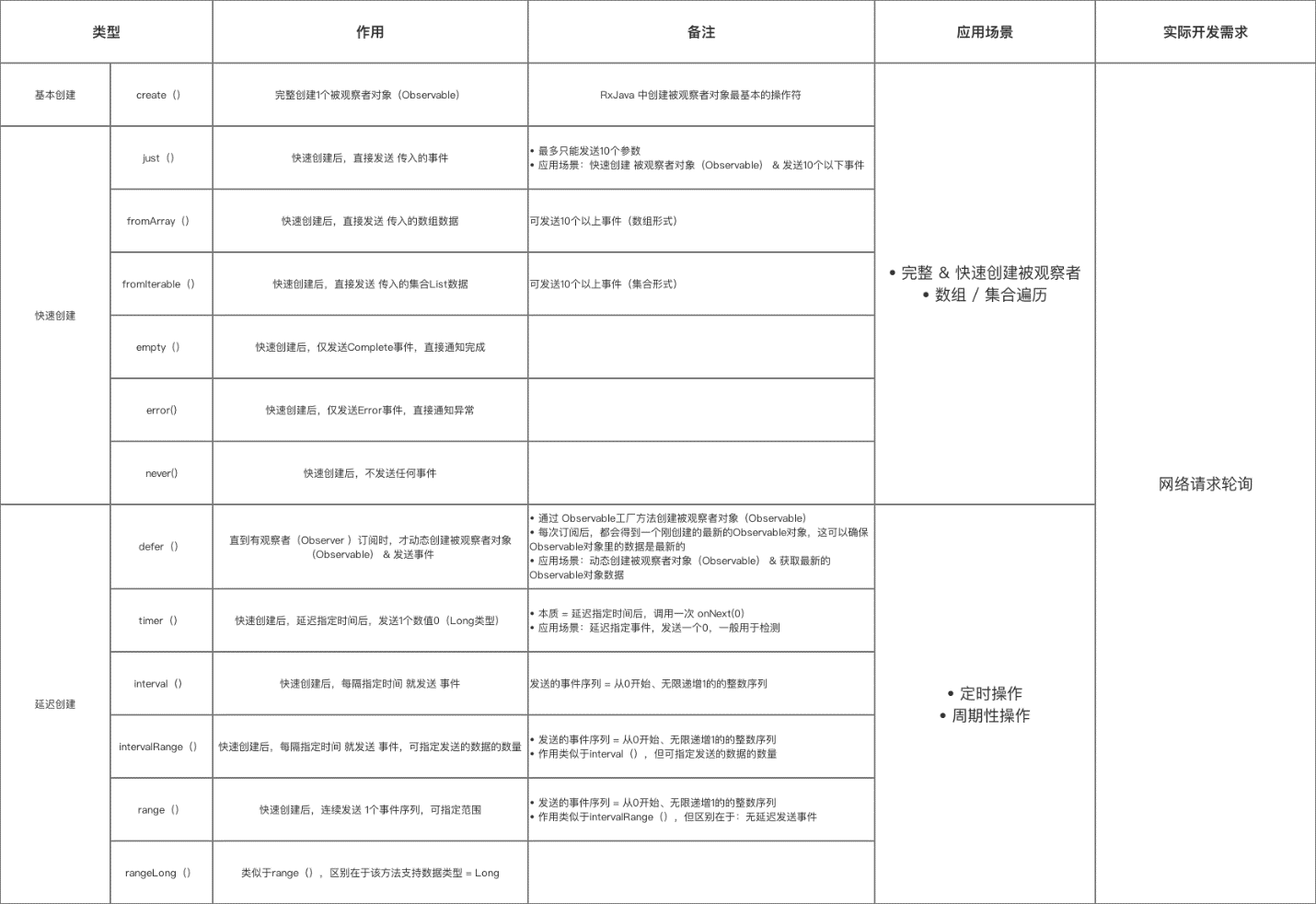
 调用subscribe()方法，在这个方法调用后，subscribeActual()被调用，才真正执行了IoSchduler中的createWorker()创建线程并运行，最终将上游Observable的subscribe()方法调度到新创建的线程中运行。

**订阅**的过程，是从**下游到上游依次订阅**的。

1. 即**终点** Observer **订阅了** map 返回的ObservableMap。
2. 然后map的Observable(ObservableMap)在被订阅时，会**订阅其内部保存上游Observable**，用于订阅上游的Observer是一个装饰者(MapObserver)，**内部保存了下游（本例是终点）Observer**，**以便上游发送数据过来时，能传递给下游**。
3. 以此类推，**直到源头Observable被订阅**，根据上节课内容，它**开始向Observer发送数据**。

**数据传递**的过程，当然是**从上游push到下游**的，

1. **源头**Observable**传递数据给下游**Observer（本例就是MapObserver）
2. 然后MapObserver接收到数据，**对其变换操作后**(实际的function在这一步执行)，再调用内部保存的下游Observer的onNext()**发送数据给下游**
3. 以此类推，直到**终点**Observer。



**3.1 Map（）**

* 作用  
  对 被观察者发送的每1个事件都通过 **指定的函数** 处理，从而变换成另外一种事件

即， **将被观察者发送的事件转换为任意的类型事件。**

**3.2 FlatMap（）**

* 作用：将被观察者发送的事件序列进行 **拆分 & 单独转换**，再合并成一个新的事件序列，最后再进行发送
* 原理

1. 为事件序列中每个事件都创建一个 Observable 对象；
2. 将对每个 原始事件 转换后的 新事件 都放入到对应 Observable对象；
3. 将新建的每个Observable 都合并到一个 新建的、总的Observable 对象；
4. 新建的、总的Observable 对象 将 新合并的事件序列 发送给观察者（Observer）

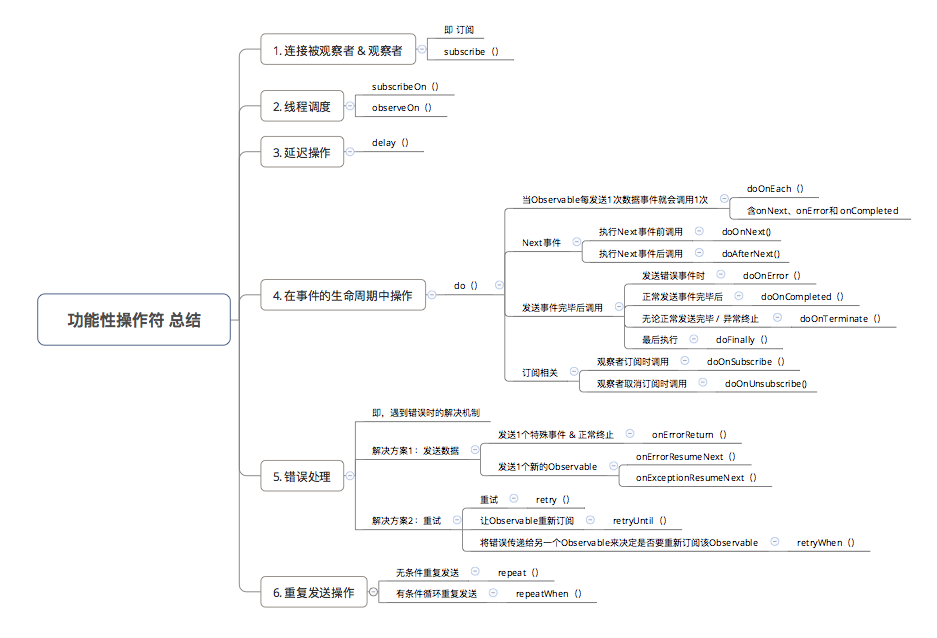
### 3.3 ConcatMap（）

* 作用：类似FlatMap（）操作符
* 与FlatMap（）的 区别在于：**拆分 & 重新合并生成的事件序列 的顺序 = 被观察者旧序列生产的顺序**

### 3.4 Buffer（）

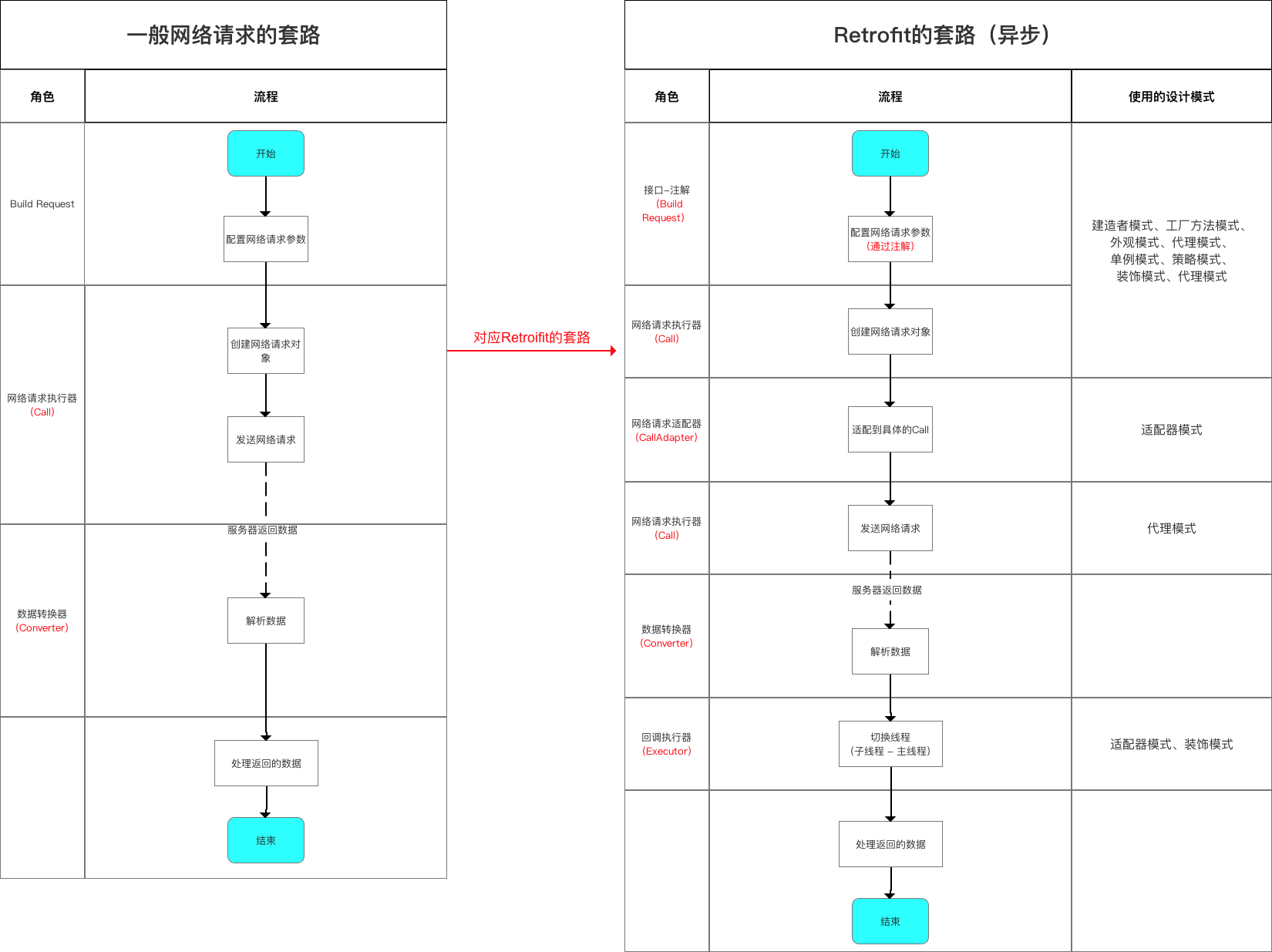
* 作用  
  定期从 被观察者（Obervable）需要发送的事件中 获取一定数量的事件 & 放到缓存区中，最终发送





# Retrofit2

 App应用程序通过 Retrofit 请求网络，实际上是使用 Retrofit 接口层封装请求参数、Header、Url 等信息，之后由 OkHttp 完成后续的请求操作

 在服务端返回数据之后，OkHttp 将原始的结果交给 Retrofit，Retrofit根据用户的需求对结果进行解析

Retrofit的本质

具体过程解释如下：

1. 通过解析 网络请求接口的注解 配置 网络请求参数
2. 通过 动态代理 生成 网络请求对象
3. 通过 网络请求适配器 将 网络请求对象 进行平台适配

平台包括：Android、Rxjava、Guava和java8

1. 通过 网络请求执行器 发送网络请求
2. 通过 数据转换器 解析服务器返回的数据
3. 通过 回调执行器 切换线程（子线程 ->>主线程）
4. 用户在主线程处理返回结果

## 步骤1

Retrofit**使用建造者模式通过Builder类**建立了一个Retrofit实例，具体创建细节是配置了：

* 平台类型对象（Platform - Android）
* 网络请求的url地址（baseUrl）
* 网络请求工厂（callFactory）

默认使用OkHttpCall

* 网络请求适配器工厂的集合（adapterFactories）

本质是配置了网络请求适配器工厂- 默认是ExecutorCallAdapterFactory

* 数据转换器工厂的集合（converterFactories）

本质是配置了数据转换器工厂

* 回调方法执行器（callbackExecutor）

默认回调方法执行器作用是：切换线程（子线程 - 主线程）

由于使用了建造者模式，所以开发者并不需要关心配置细节就可以创建好Retrofit实例，建造者模式get。

## 步骤2





Retrofit是**通过外观模式 & 代理模式 使用create（）方法**创建网络请求接口的实例（同时，通过网络请求接口里设置的注解进行了网络请求参数的配置）

 动态创建网络请求接口的实例**（代理模式 - 动态代理）**

 创建 serviceMethod 对象**（建造者模式 & 单例模式（缓存机制））**

 对 serviceMethod 对象进行网络请求参数配置：通过解析网络请求接口方法的参数、返回值和注解类型，从Retrofit对象中获取对应的网络请求的url地址、网络请求执行器、网络请求适配器 & 数据转换器。**（策略模式）**

 对 serviceMethod 对象加入线程切换的操作，便于接收数据后通过Handler从子线程切换到主线程从而对返回数据结果进行处理**（装饰模式）**

 最终创建并返回一个OkHttpCall类型的网络请求对象

## 步骤3

 **步骤1：**对网络请求接口的方法中的每个参数利用对应ParameterHandler进行解析，再根据ServiceMethod对象创建一个OkHttp的Request对象

 **步骤2：**使用OkHttp的Request发送网络请求；

 **步骤3：**对返回的数据使用之前设置的数据转换器（GsonConverterFactory）解析返回的数据，最终得到一个Response<T>对象

 **步骤4：**进行线程切换从而在主线程处理返回的数据结果

异步请求的过程跟同步请求类似，**唯一不同之处在于：异步请求会将回调方法交给回调执行器在指定的线程中执行。**



## OKHTTP

## GestureDetectoer

<https://www.cnblogs.com/ldq2016/p/7000300.html>

# Fragment的一些坑

<https://www.jianshu.com/p/d9143a92ad94>

# Dagger2

<https://zhuanlan.zhihu.com/p/24454466>