深圳黑马 Android 高薪直通车

前言：

当前的 Android 市场面向的大部分都是三年及以上的中高级开发工程师，我们如果想快速找到满意的工作、想高薪入职那就必须得掌握一手装逼的绝技、回答出一些较为深入且有新意的答案。维哥经过对往期学生面试情况的总结和分析，以及自身的工作和面试经验，给大家总结出了《深圳黑马 Android 高薪直通车》包含：暴走篇、大杀特杀篇、超神篇、项目篇、人事面试篇。估计能够涵盖 Android 开发面试中的百分之八十以上的问题，而且每个问题的答案都是经过反复思考、推敲后纯手打出来的。由于维哥自身水平有限如果答案中有一些不妥的地方，我只想说：那是我故意的，你来打我啊！！！（哈哈，开个玩笑，如有不正确之处希望大家能指出来通过 liweisz@itcast.cn 跟我联系）**PS：本书借鉴《上海黑马程序员面试宝典》优秀的地方，并对很多问题进行更加深入的分析和解答，从而能更好地适应当前的 Android 市场。此文不包含 Java 基础，需要掌握 Java 基础的请参考《上海黑马程序员面试宝典》。**

目录

暴走篇

一、 四大组件、UI 布局及数据存储。

1、 请描述 Activity 的生命周期？

首先 Activity 创建到与用户交互过程会依次走：

- onCreat(表示 Activity 创建了，这里面可以做一些初始化数据的操作，但是不宜做耗时操作。)为了装逼：我们可以说该方法有一个参数是 Bundle 用于恢复 Activity 意外销毁时保存的数据。
- onStart(表示 Activity 已经由不可见变成了可见状态)，为了装逼：我们可以说通过看源码的注释，onStart 方法可以在 onCreat 方法后调用也会在 onRestart 方法后调用。
- onResume(表示 Activity 已经由不可与用户交互变成了可与用户交互，即获取到了焦点)，通过看源码的注释我们可以知道，在这个方法中我们适合开启动画或者打开相机等操作。

下面再说一下 Activity 由失去焦点到销毁的过程。

- onPause(表示 Activity 失去焦点，由可与用户交互变成不可与用户交互)，当一个 Activity 不在栈顶的时候，就算它还是可见的（例如栈顶的 Activity 透明）它也不能与用户交互。我们通过观察源码可以发现,这个方法中主要是用于保存 Acticity 正在编辑的一些数据和状态，提出一个“edit in place”的模式给用户确保在内存不足的情况下不至于丢失数据。(这段话是装逼用的，万一理解不了也没关系)，在这个方法中我们还是和做一些如停止动画、关闭资源访问等操作。
- onStop(表示 Activity 由可见变成不可见)，接下来可能会走 onDestroy 方法或者 onRestart 方法。
- onRestart(onStop 之后没有走 onDestroy,而是再次变成用户可见，onRestart 方法之后是 onStart 方法。)

2、 横竖屏切换时 Activity 的生命周期？

默认情况下横竖屏切换，无论是横屏切换到竖屏还是竖屏切换到横屏，都会重新走一次生命周期方法不会出现网络上说的横屏切换回竖屏生命周期会走两遍的情况（即 Activity 会重新创建一遍。）

- 在 2.3 的机器上，所走的生命周期方法流程如下：
  onSaveInstanceState->onPause->onStop->onCreate->onStart->onRestoreInstanceState->onResume
- 在 4.0 的机器上，搜走的生命周期方法流程如下：
  onPause->onSaveInstanceState->onStop->onCreate->onStart->onRestoreInstanceState->onResume

怎么样解决横竖屏切换 Activity 重新创建的问题呢?

有两种方法：

(1)、直接写死屏幕朝向为竖直或者横向。

(2)在 manifest 中设置

android:configChange=" orientation|keyboardHidden|scr eenSize"

如果只设置 android:configChange=" orietation" 是不起作用的。

如果只设置 android:configChange=" orientation|keyboardHidden" ，

在 2.3 的机器上是不会重新创建 Activity。但是在 4.0 的机器上就有所不同了：

a).如果 targetVersion<=12，则不重新创建 Activity。

b).如果 targetVersion>12,则重新创建 Activity。

3、 如将 Activity 设置成窗口模式？

在 manifest 中给该 Activity 设置如下属性：

android:theme=" @android:style/Theme.Dialog"

4、 Actvity 的 4 种启动模式？

Activity 的启动模式一共有四种，分别是 standard、singleTop、singleTask、singleInstance。

(1)standard：默认的启动模式，每开启一个 Activity 都会直接在 Activity 任务栈累加。如依次启动 A、B、C、A、A、A 六个 Activity，在任务栈中我们将看到有这六个 Activity。

(2)singleTop：单一栈顶模式，意思就是说如果某个 Activity 已经在栈顶的时

候，如果再启动该 Activity 不会重新创建一个 Activity 而是会使用已经在栈顶的这个 Activity。如依次 A、B、C、D、D 五个 Activity，其实在任务栈中只有 A、B、C、D 四个 Activity 实例，因为 D 已经在栈顶了不会重新创建 D。

(3)singleTask：单一任务栈模式，意思就是说该 Activity 在此任务栈中只能有一个实例。如果某个 Activity 已经在任务栈中了，在此启动该 Activity 时会将该 Activity 上方的所有 Activity 全部弹出栈将该 Activity 至于栈顶。例如：任务栈中已经有 A、B、C、D、E、F 这六个 Activity 了，如果我现在启动 C，那么就会将 D、E、F 弹出栈，此时任务栈中就只有 A、B、C 三个 Activity 了。

(4)singleInstance：单一实例模式，设置这中启动模式的 Activity 在启动的时候会给其单独开一个任务栈。

怎么样设置 Activity 的启动模式？

有两种方法设置 Activity 的启动模式：1、在 manifest 中进行配置 launchMode。2、在代码中配置，通过 intent.addFlags()方法进行设置。通过看官方文档我们可以了解到：第一种方法是 Activity 对其自身的要求以何种模式启动。第二种方法是上一个 Activity 对它的要求以何种模式启动。

关于 addFlags 中的一些 Flag 的解释？

(1)FLAG_ACTIVITY_CLEAR_TOP:顾名思义清空其顶部所有的 Activity 就是以 singleTask 的模式启动 Activity。

(2)FLAG_ACTIVITY_SINGLE_TOP:顾名思义就是以 singleTop 的模式启动 Activity。

(3)FLAG_ACTIVITY_NEW_TASK :系统会寻找或创建一个新的任务栈来放置目标 Activity。

一个启动模式是 singleTop 的 Activity，当它在栈顶时再次试图启动它会发生什么？

不会重新创建该 Activity 的实例，但是通过看 Activity 的源码我们会发现它会回调 Activity 的 onNewIntent(Intent intent)方法，但是此时 getIntent()方法返回的还是原来的 intent 对象，你可以通过 setIntent(intent)方法来更新 intent。

四种启动模式的应用场景？

a).standard:默认的都是用这种模式。

b).singleTop:例如接收消息后显示的界面，QQ 接收消息后立马弹出界面，当一次性接收 100 个消息，不可能弹出一百个界面吧。

c).singleTask:适合作为程序的入口点，例如浏览器的主界面，无论从多少个应用程序打开浏览器，都只会启动一次主界面不会重复创建。

d).singleInstance:适合做与应用程序分离的界面，例如：来电界面，它的特点是会创建一个单独的且与用户正在交互的界面的任务栈，它会一直在前端直到所有 Activity 全部退出。

5、 怎么保存 Activity 的状态？

首先，只有当 Activity 被意外回收时才需要去保存 Activity 的状态和某些数据。（**意外回收就是指没有调用 finish 方法，但是 Activity 被销毁了**）一般情况下 Activity 的状态会自动保存，我们这里说的保存状态是说的一些额外的需要保存的状态和数据。

我们可以通过重写 onSaveInstanceState()方法来进行数据和状态的保存。该方法接收一个 Bundle 类型的参数，我们可以将要保存的数据存储到该 Bundle 对象中，在 onCreat(Bundle saveInstanceState)方法中从 Bundle 对象中将保存的数据取出并恢复。从源码的注释我们可以看到，该方法是在 onStop 之前被调用的，但是没法确定是在 onPause 之前或之后调用。**一定要注意：只有当 Activity 意外销毁时才会调用该方法，如果 Activity 是调用 finish 方法销毁的则不会调用该方法。**

6、 描述一下 BroadcastReceiver 的生命周期？

广播接收者的生命周期非常短暂，在接收到广播的时候创建，当 onReceiver()方法执行完就结束了。正因为其生命周期很短暂，所以最好不要在广播接收者中创建子线程和服务之类的，因为广播接收者被销毁后就成了空进程很容易被系统回收。

7、 BroadcastReceiver 的注册方法？

一共有两种注册方式，静态注册和动态注册。

静态注册：在 manifest 中进行如下配置：

```
<receiver android:name=".BroadcastReceiverTest">
    <intent-filter>
<actionandroid:name="android.intent.action.BATTERY_LOW">
```

```
        </action>
    </intent-filter>
</receiver>
```

静态注册的广播接收者只要 app 在运行则一直能接收到广播。

动态注册：在代码中进行，代码例子如下：

```
    mReceiver = new BroadcastReceiverTest();
IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction(Intent.ACTION_BATTERY_LOW);
registerReceiver(mReceiver, intentFilter);
```

动态注册一般是在 Activity 或者 Service 中进行注册，当 Activity 或 Service 被销毁的时候，就接收不到广播了。

8、 你在项目中使用过广播接收者吗？

广播接收者的作用主要是通信。

a).我们在项目中如果要监控系统的变化：如开机、SD 卡挂载状态、电量状态、收到短信、WIFI 状态切换等等。

b).使用广播接收者可以很好地进行多线程之间的频繁通信，因为广播本来就是双向的（A 线程可以向 B 线程发广播，同理 B 线程也可以向 A 线程发）。这样的话比用 Handler 就要好，因为 Handler 是单向的。

c).广播接收者还能用于 Fragment 和 Activity 之间或者不同组件之间通信。但是由于用广播接收者的话代码比较冗余，现在涉及到这方面的需求一般都使用 EventBus 来实现。

9、 默认情况下，Service 中是否能做耗时操作？

默认情况下 service 也是运行中 app 所在进程的主线程中，肯定是不能做耗时操作的。但是我们可以通过在 manifest 中指定它的 process 属性，让他在另外一个进程中运行，这样的话就可以做耗时操作了。

10、 描述一下两种启动 Service 的方式及其各自的生命周期？

启动 Service 的方式有两种：startService 和 bindService。

startService：用这种方式启动的 service 由于没有和 Activity 进行绑定，所以可

以长期在后台运行但不能调用服务里的方法。用这种方式启动的 service 的生命周期如下 onCreat()、onStartCommand()、onDestroy()。第一次启动之后如果没有调用 stopService()方法，再次调用 startService()方法，不会重新走 onCreat()但是会重新走 onStartCommand()。

bindService：用这种方式启动的 service 会和 Activity 进行绑定，所以不能长期在后台运行但是能调用服务里面的方法。它的生命周期方法如下：onCreat()、onBind()、onUnbind()、onDestroy()。一个服务可以被多个客户（一般指 Activity 绑定），只有当所有被绑定的对象都调用了 onUnbind()方法，该 service 才算销毁。

11、 Service 和 IntentService 的对比。

Service 默认是运行在 app 运行的进程的主线程中，我们不能在里面做耗时操作。

IntentService 是 Service 的子类，它会创建一个工作线程来处理所有的 Intent 请求。执行完一个 Intent 对象的请求后，如果没有新的 Intent 请求到达，则会自动停止 service 不用你去调用 stopservice 方法。

IntentService 处理事务时是采用的 handler 方式，创建了一个名为 ServiceHandler 的内部 Handler，并把它直接绑定 HandlerThread。ServiceHandler 把处理 Intent 所对应的事务都封装到 onHandleIntent 方法中，我们可以直接实现 onHandleIntent 方法再在里面根据 Intent 的不同进行不同的事务处理。

12、 如何确保一个 Service 不会在系统内存不足的情况下被杀死？

由于服务是在在后台运行的，是不可见的，属于后台进程，所以在系统内存不足的情况下可能会被清理掉。

a).众所周知，前台进程是不会在系统资源不足的情况下被自动清理掉的，所以要确保 Service 不会被杀死我们可以想办法让它变成一个前台进程。要让 Service 变成前台进程我们可以在启动服务的时候使用 startForeground () 方法将 Service 就变成前台进程。经过实测，使用该方法能降低被 kill 的概率但是不能确保不被 kill。

b).除了让他变成前台进程之外，我们还能通过在 onDestroy()方法中发送广播，在广播接收者中重启服务。但是当使用第三方应用强制 Kill 服务的话，连 onDestroy()方法都进不来，所以还是无法确保。

c).使用双 Service 守护，两个 Service 不断相互判断对方是否存活，如果发现对方被杀死则立马将其重新开启。

13、 你的项目中使用过 Service 吗？

根据 Service 的定义，我们可以知道需要长期在后台进行的工作我们需要将其放在 Service 中去做。说得再通熟易懂一点，就是不能放在 Activity 中来执行的工作就必须得放到 Service 中去做。如：音乐播放、下载、上传大文件、定时关闭应用等功能。这些功能如果放到 Activity 中做的话，那 Activity 退出被销毁了的话，那这些功能也就停止了，这显然是不符合我们的设计逻辑的，所以要将他们放在 Service 中去执行。

14、 简单描述一下 ContentProvider。

顾名思义，ContentProvider 是内容提供者，为存储和获取数据提供统一的接口，可以在不同应用程序之间实现数据共享。它提供了 query、insert、delete、update、getType、onCreate 等方法，使用 ContentProvider 首先要定义一个类继承 ContentProvider，然后再覆写 query、insert、delete、update 等方法，当然作为四大组件之一必须在 manifest 中注册。外部应用可以通过 ContentResolver 来访问该 Provider。

15、 写一条创建表的 sql 语句。（关于数据库较难的问题在提升篇会有介绍）

字段说明 系别编号 系名称 系主任 字段名称 depNo depName depMan。

create table Depart(depNo int primary key,depName varchar(50) not null,depMan varchar(50) not null)

二、 其他基础部分。

1、说说ListView的优化？

ListView的优化可以从ListView和Adapter、ViewHolder三方面入手：

1)、从Adapter入手。

(1).在adapter中的getView方法中尽量少使用逻辑

(2).尽最大可能避免GC

(3).滑动的时候不加载图片

(4).将ListView的scrollingCache和animateCache设置为false

(5).item的布局层级越少越好

(6).使用ViewHolder

2)、从ListView入手。

如果多层嵌套无法避免的情况下，建议把listview的高和宽设置为match_parent. 如果是代码继承的listview，那么也请你别忘记为你的继承类添加上LayoutPrams，注意高和宽都是match_parent的。

3)、从ViewHolder入手。

将缓存类ViewHolder设置为静态类也就是static, 静态类只有在首次加载的时候比较耗时，后面就可以直接使用了，同时保证了，内存中只会有一个ViewHolder，节省了内存的开销。

2、 常见的Java设计模式。

- **单例模式。**

定义：一个类有且仅有一个实例，并且自行实例化向整个系统提供。

实现步骤：(1).私有化构造函数。(2).自行实例化对象。(3).提供一个公有方法向整个系统提供该实例化对象。

分类：按创建时机可以分成两种：饿汉式和懒汉式。

饿汉式：类加载时创建，线程安全。

懒汉式：在使用时判断，如果需要再创建。线程不安全，在多线程并发访问的情况下，可能会产生多个实例对象。

怎么来解决懒汉式线程不安全的问题呢？

使用双重检测锁的机制可以解决，具体实现是：先检测是否为空，如果为空再对该单例类.class加锁，再判断一次是否为空。这样既解决了线程安全问题，也避免了每次使用单例都要加锁带来的系统资源消耗。具体代码如下：

```
public class Singleton {

    private static volatile Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

另外介绍两种非常牛逼的单例模式的写法：

(1).使用静态内部类。

a).步骤：

1).私有化构造函数。

2).声明一个私有的静态内部类,内部类里面声明一个静态变量instance，直接将单例对象赋予该变量。

3).声明一个公有的静态方法获取该实例。

具体代码如下：

```
public class Singleton {

    private Singleton() {}

    private static class SingletonInstance {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonInstance.INSTANCE;
    }
}
```

b).好处：

1).实例在静态内部类加载的时候创建，一定是单例的。

2).不会过早创建，只有在内部类被使用的时候会创建。

(2).使用枚举创建单例。

这是最牛逼的实现单例的写法，不仅避免多线程并发访问的问题，还能防止反序列化重新创建新的对象。但由于枚举是在JDK1.5之后才添加所以

际开发中很少有人应用。具体代码如下：

```
public enum Singleton {
    INSTANCE;
    public void whateverMethod() {
    }
}
```

单例模式在项目中的应用？

例如使用工厂模式时的工厂类、登录成功后的用户类我们都可以使用单例模式来实现。

- **工厂模式。**

简单工厂模式

定义：只有一个工厂类，通过想创建方法中传入不同的参数，来生产不同的产品。代码如下

```
public class FoodFactory {

   public static Food getFood(String type) throws InstantiationException,
IllegalAccessException, ClassNotFoundException {
      if(type.equalsIgnoreCase("mcchicken")) {
         return McChicken.class.newInstance();

      } else if(type.equalsIgnoreCase("chips")) {
         return Chips.class.newInstance();
      } else {
         System.out.println("哎呀！找不到相应的实例化类啦！");
         return null;
      }


   }
}
```

优点：能够根据外界给定的信息，决定应该创建哪个具体类对象。用户在使用时可以无需了解这些对象是如何创建的，有利于整个软件体系结构的优化。
缺点：违背面向对象设计的“开闭原则”和“单一职责原则”。因为我们每增加一个产品就要去修改工厂类的代码，而且一个工厂生产各式各样的产品显然职责不单一。

工厂方法

定义：定义一个创建对象的接口（抽象工厂类），让其子类（具体工厂类）去决定实例化哪个类（具体产品类）。一个工厂实现一种产品，一对一的关系。
例子：定义一个抽象的手机厂类：MobileFactory，然后让苹果手机厂、诺基亚手机厂、三星手机厂、小米手机厂分别继承MobileFactory。然后这四个手机厂分别生产苹果手机、诺基亚手机、三星手机、小米手机。功能明确、职责单一，如果要生产另外一种如华为手机，就只要在声明一个华为手机厂就行了。
优点：符合面向对象设计的“开闭原则”和“单一职责原则”。
缺点：每增加一个产品都要增加相对应的具体产品类和具体工厂类，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。

抽象工厂模式

定义：提供一个创建一系列相关或者相互依赖产品的接口，而无需制定他们的具体类，一对多的关系。抽象工厂模式是工厂方法模式的升级版。
例子：
定义一个抽象工厂接口：西式快餐工厂，定义两个具体工厂类实现抽象工厂类：KFC和麦当劳。
定义两个抽象产品接口：KFC的食物和麦当劳的食物，在分别定义新奥尔良鸡腿堡、老北京鸡肉卷、吮指原味鸡实现KFC的食物接口；定义板烧鸡腿堡、麦辣鸡、麦乐鸡实现麦当劳的食物接口。
然后KFC就能够生产新奥尔良鸡腿堡、老北京鸡肉卷、吮指原味鸡；麦当劳就能生产板烧鸡腿堡、麦辣鸡、麦乐鸡。

优点：即符合面向对象设计的“开闭原则”和“单一职责原则”。又减少了系统中的类的数量，不用像工厂方法一样每生产一个产品都要有一个具体的工厂类。

观察者模式。

定义：又名发布—订阅模式，对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

组成部分：

1).抽象目标角色(Subject)：目标角色知道它的观察者，可以有任意多个观察者观察同一个目标。并且提供注册和删除观察者对象、通知观察者的方法。目标角色往往由抽象类或者接口来实现。

2).抽象观察者角色(Observer)：为那些在目标发生改变时需要获得通知的对象定义一个更新接口，里面有一个更新数据的方法。抽象观察者角色主要由抽象类或者接口来实现。

3).具体目标角色(Concrete Subject)：实现Subject，实现添加、删除观察者、通知观察者的方法，当它的状态发生改变时，向它的各个观察者发出通知。

4).具体观察者角色(Concrete Observer)：实现Observer的更新接口以使自身状态与目标的状态保持一致，获取通知进行更新。

Subject接口

```
1  public interface Subject {
2      public void registerObserver(Observer o);
3      public void removeObserver(Observer o);
4      public void notifyAllObservers();
5  }
```

Observer接口

```
1  public interface Observer {
2      public void update(Subject s);
3  }
```

适配器模式。

定义：适配器模式就是将一个类的接口，转换成客户期望的另一个接口。适配器让原本接口不兼容的类可以合作无间。

分类：单接口适配器模式和对象适配器模式。

对象适配器模式：我们来看这么一个例子：生产一种"A型螺母适配器"，这种A型螺母适配器的前端符合A型螺母标准要求，可以拧在A型螺母上，后端又焊接了一个B型螺母。这样用户就可以借助A型螺母适配器在A型螺母上使用B型的螺丝了。**这样就实现了将一个接口转换成客户期望的另一个接口。**

单接口适配器模式：一个A接口中抽象方法很多，B类需要A接口中的某个方法，如果由B类直接实现A接口的话，就要实现A接口中所有的方法，这样会产生很多无用代码造成接口浪费。所以我们可以定义一个Adapter类实现A接口，所有方法都是空实现，然后让B类继承Adapter类，选择性实现自己需要的方法。

3、 描述一下Fragment的生命周期。

Fragment创建过程：

1).onAttach(Context context):当Fragment第一次依附到context上时调用。在早期版本SDK上时onAttach(Activity activity)。

2).onCreate:初始化Fragment。

3).onCreatView:初始化Fragment的视图。

4).onActivityCreated():当fragment所依附的Activity创建和Fragment的视图实例化完成后调用。

5).onStart():Fragment由不可见变为可见时调用。

6).onResume():Fragment获取到焦点是调用。

Fragment的销毁过程：

1).onPause():Fragment失去焦点时调用。

2).onStop():Fragment变为不可见时调用。

3).onDestroyView():Fragment的视图被销毁时调用。

4).onDestroy():Fragment不再使用了，被销毁时调用。

5).onDetach():Fragment不再依附Activity时调用。

4、 Fragment和Activity之间、Fragment和Fragment之间如何进行数据传递？

目前最流行的数据传递方式就是使用EventBus，自定义一个事件类，在要发送消息的地方用EventBus对象发送该事件，在要接收消息的地方使用`eventBus.register(this)`方法，然后再接收的地方实现(onEvent、onEventMainThread、onEventBackGroundThread、onEventAsync)这四个方法中的一个来接收发送过来的事件。

5、 Fragment如何实现类似Activity任务栈的压栈和出栈效果？

往Actvity中添加Fragment默认是不会添加到任务栈的，不会有Activity那种压栈和出栈的效果。要实现这种效果我们可以通过如下步骤实现：

1).添加`Fragment`的时候通过`fragmentTransaction.addToBackStack(String)`方法，来将`Fragment`添加到任务栈中。

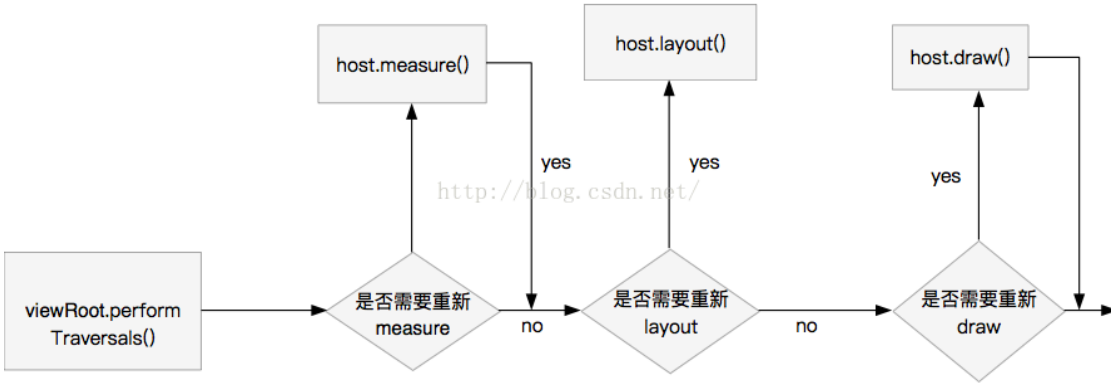
2).在要退出Fragment的时候，先通过fragmentManager.getBackStackEntryCount()方法获取栈中Fragment的数量。

3).如果当前任务栈中的Fragment只有一个的时候，我们按后退键就直接退出Activity。如果数量大于1，则调用fragmentManager.popBackStack(String name, int flags)或者fragmentManager. popBackStack(int id, int flags)方法来将Fragment弹出栈。

大杀特杀篇

1、 View 的绘制流程。

View 的绘制是从 ViewRoot 的 performTraversals()方法开始的，经过一下流程。



从上图我们可以看出，View 的绘制分为三个步骤：measure、layout、draw

measure 过程：

作用：measure 过程对整个 view 树的所有控件的宽高进行计算。

源头：measure 是从 ViewRoot 类中的 host.measure 开始的，内部调的是 View 的 measure(int widthMeasureSpec, int heightMeasureSpec)方法，measure 方法里面调用了 onMeasure(int widthMeasureSpec, int heightMeasureSpec)方法。

MeasureSpec:

1).定义：measure 方法中的两个参数的类型就是 MeasureSpec，而 MeasureSpec 是父控件对子控件宽高的期望。它是一个 32 位的 int 类型的数，前两位代表测量模式 SpecModel,后 30 位代表测量大小 SpecSize。

2).SpecModel:一共有三种测量模式——EXACTLY、AT_MOST、UNSPECIFIDE。

a). EXACTLY 精确地测量模式，xml 文件中写 200px、match_parent 等代表使用 EXACTLY 测量模式。

b). AT_MOST 最大模式。xml 文件中写 wrap_content 表示使用 AT_MOST 测量模式

c). UNSPECIFIED，它表示的测量模式是无限大，就是你想要多大就多大，我们只在绘制特定情况的自定义 view 才用得到此模式。

结束：真正代表测量结束的方法是 setMeasuredDimension 方法，该方法传入的两个参数是宽、高的 SpecSize。测量结束后我们能够通过 getMeasureHeight 和 getMeasureWidth 来获取测量宽高。

自定义 ViewGoup 一定要重写 onMeasure 方法，用于测量子 View 的宽高，不

重写的话子 View 就没有宽高。

自定义 View 如果在 xml 中使用需要 wrap_content 属性的话就需要重写 onMeasure 方法来设置 wrap_content 时的默认大小，要不然的话就会显示为 match_parent 的效果。

layout 过程：

作用：ViewGroup 用来将子 View 放在合适的位置上。

源头：layout 是从 ViewRoot 类中的 host.layout 开始的，内部调的是 ViewGroup 的 layout 方法。

在 ViewGroup 的 layout 方法中，先调用 setFrame 方法来确定自己的左上右下的位置，再调用 onLayout 方法来确定子 View 的位置。

自定义 ViewGroup 一定要重写 onLayout 方法来确定子 View 的位置，而自定义 View 一般不需要重写 onLayout 方法，因为它的位置是由父控件来确定的。

draw 过程：

作用：真正将内容展示在屏幕上让我们能够看到。

源头：draw 是从 ViewRoot 类中的 host.draw 开始的，内部调的是 View 的 draw 方法。

draw 的步骤：

1).绘制背景。

2).绘制内容，也就是调用 onDraw 方法。

3).绘制子 View，也就是调用 dispatchDraw 方法。

4).绘制装饰品，如 listview 的滚动条等。

对于 draw 过程，可以说是最简单也可以说是最复杂的，简单的在于 Google 已经帮我们吧 draw 框架写好了，所以我们在自定义 ViewGroup 的时候不用管 draw 过程，只需要实现 measure 和 layout 过程就可。复杂的在于，我们写继承 View 的自定义控件的时候是一定要重写 onDraw 方法的，这样才能绘制出你自定义的 View 的内容。而 onDraw(Canvas canvas)方法中最重要的两个东西则是 Paint（画笔）和 Canvas（画布），可以记住一些常用的 API 方法。

Paint 中常见的 API：

1).setColor，设置颜色。

2).setColorFilter,设置颜色过滤。

3).setAlpha,设置透明度。

4).set(Paint paint)，直接将另外一支画笔的属性设置给这支画笔。

5).setStyle，设置是填充还是描边。

6).setTextSize,设置文字大小。等等

Canvas 常见的 API：

1).drawArc，画弧形。

2).drawCircle，画圆形。

3).drawBitmap，画 Bitmap。

4).drawLine,画直线。

5).drawOval，画椭圆。

6).drawRect，画矩形。

7).drawPath，画路径，该方法可以画出很多奇形怪状的形状。

8).rotate,旋转画布。

9).translate,平移画布。

10).save,保存画布状态，一般是在旋转或者平移之前可以保存画布状态，便于还原。

11).restore,还原画布状态，调用这个方法之前肯定调用 save 来保存了状态。

关于 View 的绘制更详细的内容请看维哥的博客

http://blog.csdn.net/itheimaleevi/article/details/52389273

2、 Touch 事件的传递机制。

一个完整的 touch 事件，由一个 down 事件、n 个 move 事件，一个 up 事件组成。

Touch 事件一般的传递流程 Activity------>window(唯一实现类是 PhoneWindow)------>顶级 View（DecorView）-------->ViewGroup------->View。

监听 Touch 事件有两种方式：setOnTouchListener 和直接重写三个方法

(dispatchTouchEvent、onInterceptTouchEvent、onTouchEvent)。

setOnTouchListener：

该方式监听 Touch 事件，优先级较高，如果在 onTouchListener 的 onTouch 方法中 return true 的话，那么 onTouchEvent 方法是接收不到该 Touch 事件的。而且因为 onClickListener 中的 onClick 方法实际上是在 onTouchEvent 中被调用的，所以如果 Touch 事件走不到 onTouchEvent 方法的话，点击事件也不会生效。

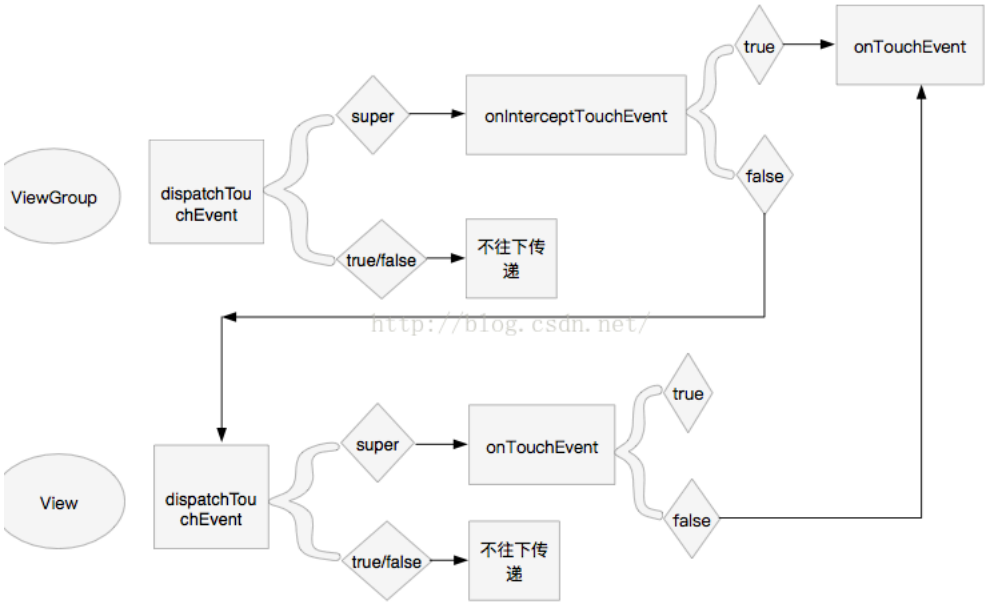
直接重写三个方法：

dispatchTouchEvent：该方法表示对事件进行分发，在这个方法中我们一般 return super.dispatchTouchEvent，将该事件分发下去。

onInterceptTouchEvent：该方法表示对 Touch 事件进行拦截，该方法是 ViewGroup 特有的，View 没有。在 ViewGroup 中如果 onInterceptTouchEvent 返回 true，表示将该事件拦截，那么事件将传递给该 ViewGroup 的 onTouchEvent 方法来处理。如果 onInterceptTouchEvent 返回 false 表示不拦截，那么该事件将传递给子 View 的 dispatchTouchEvent 来进行分发。

onTouchEvent：该方法表示对 Touch 事件进行消费，返回 true 表示消费，返回 false 表示不消费那么该事件将传递给父控件的 onTouchEvent 进行处理。

具体的传递流程如下图所示：



关于 Touch 事件的分发机制更详细的内容请看维哥的博客 http://blog.csdn.net/itheimaleevi/article/details/52422839

3、 说说对 Http 协议的理解。

定义：Http 协议是一个基于请求和响应模式的、无状态的、应用层的超文本传输协议。

请求：请求由请求行、消息报头、请求正文组成。

1). 请求行：包含请求方法、请求 URI 和协议版本。

请求方法一共有八种，但是我们 Android 中常用的只有 GET 和 POST 两种。(只需要弄懂 GET 和 POST 及其区别，需要装逼的请记住八种)

(1).GET：请求获取 Request-URI 所标识的资源。

(2).POST：在 Request-URI 所标识的资源后附加新的数据。

(3).HEAD：请求获取由 Request-URI 所标识的资源的响应消息报头。

(4).PUT：请求服务器存储一个资源，并用 Request-URI 作为其标识。

(5).DELETE：请求服务器删除 Request-URI 所标识的资源。

(6).TRAC：请求服务器回送收到的请求信息，主要用于测试或诊断。

(7).CONNECT：保留将来使用。

(8).OPTIONS：请求查询服务器的性能，或者查询与资源相关的选项和需求。

GET 和 POST 的区别：(1).用 GET 提交表单数据只经过了简单的编码，并且附在 URL 地址后面发送给服务器。POST 请求提交数据更加安全。(2).各种浏览器对 URL 的长度有限制，所以 GET 请求提交的数据量较小。POST 请求提交的数据量较大。(3). 服务器取值方式不一样。GET 方式取值，如 php 可以使用$_GET 来取得变量的值，而 POST 方式通过$_POST 来获取变量的值。

2). 请求报头：请求报头允许客户端向服务器端传递请求的附加信息以及客户端自身的信息。

常见的请求报头：(可以不掌握，想装逼的请掌握)

a).Accept 用于指定客户端希望接收哪些类型的信息。例如：Accept：image/gif 表示希望接收 gif 图。

b).Accept-Charset: 用于指定客户端接受的字符集。例如：Accept-Charset:gb2312

c). Accept-Encodin: 用于指定可接受的内容编码,例如：Accept-Encoding:gzip.deflate。没设置则表示任何内容编码都能接受。

d). Accept-Language: 用于指定一种可接收的自然语言,例如: Accept-Language:zh-cn, 没设置则表示任何语言都能接受。

e).Host: 用于指定被请求资源的 Internet 主机和端口号，发送请求时该报头是必须的。

f).User-Agent: 允许客户端将它的操作系统、浏览器和其它属性告诉服务器。

g). Cookie: 非常重要，作用是将 cookie 的值发送给 HTTP 服务器

h).Content-Length:请求消息正文的长度。

i).Content-Type:发送的内容的类型。

响应：响应包括状态行、状态报头、响应正文。

1).状态行：包含协议版本号、响应状态码、状态码文本描述。

常见状态码：

a).1XX——指示信息，表示请求已接受，继续处理。

b).200——请求成功。

c).3XX——重定向。

d).403——服务器收到请求，但拒绝进行处理。404——资源不存在。

e).500——服务器发生不可预期的错误。

2).响应报头：允许服务器传递不能放在状态行中的附加响应信息，以及关于服务器的信息和对 Request-URI 所标识的资源进行下一步访问的信息。

常见的响应报头：(可以不掌握，想装逼的请掌握)

Location:用于重定向接受者到一个新的位置。常用在更换域名

Server:包含了服务器用来处理请求的软件信息。

Date: 生成消息的具体时间和日期，即当前的 GMT 时间。

Set-Cookie: 非常重要，用于把 cookie 发送到客户端浏览器， 每一个写入 cookie 都会生成一个 Set-Cookie.

Content-Type:告诉客户端，响应的内容类型。

4、 MVC 和 MVP 模式。

MVC 模式:

定义：MVC 是一种经典的三层架构模式，目的是将数据和视图分离，使他们不相互依赖，符合面向对象设计的单一职责原则。

组成：MVC 模式有三个组成部分，分别是： Modle、View、Controller。

在 Android 工程中，这三部份的分工如下：

Modle：业务逻辑和实体模型，包含网络请求、数据库读取、文件读取、逻辑运算、业务 Bean 类等等。

View：布局文件。

Controller：Activity 或者 Fragment。

缺点：MVC 虽然将界面呈现和逻辑代码分离了,但是在实际的 Android 开发中并没有完全起到想要的作用。View 对应的 XML 文件实际能做的事情很少，很多界面显示由 Controllor 对应的 Activity 给做了，这样使得 Activity 变成了一个类似 View 和 Controllor 之间的一个东西。造成了 Controller 层非常臃肿。

MVP 模式：

定义:基于 MVC 模式改进得到的三层架构模式, 旨在解决 MVC 的造成 Controller 层臃肿的问题。

组成:MVP 模式有三个组成部分，分别是 Model、View、Presenter。在 Android 工程中，这三部分的分工如下：

Modle：业务逻辑和实体模型，包含网络请求、数据库读取、文件读取、逻辑运算、业务 Bean 类等等。

View :Actviity 和 Fragment。先写一个 View 接口, 再让相应的 Activity 和 Fragment 来实现该接口。

Presenter：自己写 Presenter 类来进行 Model 和 View 的交互。

优点：提高代码复用性、增加可拓展性、降低耦合度、代码逻辑更加清晰。

缺点：增加了很多的接口和实现类。代码逻辑虽然清晰，但是代码量要庞大一些。

5、 屏幕适配。

屏幕适配分为两部分：图片适配和尺寸适配。

图片适配：目前流行的是切一套最大的图——720*1280，在低分辨率的手机上可以通过设置 imageView 的大小和 scaleType 来将图片压缩。（这样做的目的是减小 APK 体积）针对那些常用的、比较重要的 icon 会切多套图。

尺寸适配：

1)较为通俗的方法就是根据不同的手机分辨率在 res 文件夹下创建不同的 values 文件夹，在每个都有一个 dimens 文件，dimens 文件里就写着所有的尺寸。每种分辨率的手机就到不同的 values 文件夹里面的 dimens 文件去进行尺寸匹配。

2)使用谷歌官方推出的 percent library 来进行适配。（详情请百度）

3)使用国内某大神写的 AutoLayout 来进行适配。（详情请百度）

一些概念：

1)屏幕尺寸：屏幕对角线的长度。

2)屏幕分辨率：横纵向上的像素点数。

3) dpi:是 dot per inch 的缩写，表示每英寸上的像素点数。与屏幕尺寸和屏幕分辨率有关。

4) dip/dp: 是 Density Independent Pixels 的缩写，即密度无关像素

5)px:像素。

6)sp: 即 scale-independent pixels，与 dp 类似，但是可以根据文字大小首选项进行放缩，是设置字体大小的御用单位。

mdpi、hdpi、xdpi、xxdpi：用来修饰 Android 中的 drawable 文件夹及 values 文件夹，用来区分不同像素密度下的图片和 dimen 值。

1).mdpi:480*320,

2).hdpi:800*480,854*480,960*540

3).xhdpi:1280*720

4).xxhdpi:1920*1080

6、 Handler 消息机制。

作用：由于主线程中做耗时操作会导致 ANR 异常，所以需要将网络请求等耗时操作放到子线程中进行，但是由于在子线程中不能操作 UI，所以需要将子线程中获取到的数据传递到 UI 线程中进行 UI 更新。这样 Handler 机制就应运而生了。（当然 Handler 机制不仅仅能完成子线程和主线程的通讯，任何线程之间的通讯

都能用 Handler。）

使用：

1).在主线程创建 Handler 对象，重写 handleMessage 方法。用该 handler 对象在子线程发送消息，然后在主线程的 handleMessage 方法中处理消息。

2).handler.post(Runnable runnable)。

组成：Handler 消息机制有四大组成部分，分别是：

1).Handler:用于发送消息和处理消息。

2).Message:用于携带数据和通知。

3).MessageQueue:用于存储消息。（单链表）

4).Looper:无限循环的轮训器，用于从 MessageQueue 中取出消息，并交由 Handler 处理。

源码解析：完全进行源码解析的话篇幅太大，这里就从几个问题出发来说说源码。

1). 为什么我们在使用 Handler 的时候没看到 Looper 和 MessageQueue？

主线程默认通过 Looper.prepareMainLooper 方法准备了一个 Looper 对象，prepareMainLooper 方法的具体实现如下:

```
    public static void prepareMainLooper() {
    prepare(false);
    synchronized (Looper.class) {
        if (sMainLooper != null) {
            throw new IllegalStateException("The main
Looper has already been prepared.");
        }
        sMainLooper = myLooper();
    }
}
```

实际上是先调用了 prepare 方法，然后通过 myLooper 方法获取 Looper 对象赋值给 sMainLooper。prepare 方法的具体实现如下：

```
    private static void prepare(boolean quitAllowed) {
    if (sThreadLocal.get() != null) {
```

```
        throw new RuntimeException("Only one Looper may be
created per thread");
    }
    sThreadLocal.set(new Looper(quitAllowed));
}
```

实际上就是 new 一个 Looper 对象，然后通过 set 方法将其存储到 ThreadLocal 中，那我们猜测 myLooper 方法应该是通过 get 方法从 ThreadLocal 中获取 Looper 对象。我们看看吧

```
    public static @Nullable Looper myLooper() {
    return sThreadLocal.get();
}
```

果然，这样的话我们就获取到了 Looper，对象。但是 MessageQueue 对象呢？我们看 Looper 的构造函数就知道了

```
    private Looper(boolean quitAllowed) {
    mQueue = new MessageQueue(quitAllowed);
    mThread = Thread.currentThread();
}
```

在 Looper 的构造函数中，new 了一个 MessageQueue 对象，并赋值给 Looper 类的成员变量 mQueue。

2).怎么实现由主线程向子线程传递数据？

要实现主线程向子线程中传递数据，那么我们必须在子线程中 new 一个 Handler 对象，然后用该 handler 对象在主线程发送消息到子线程。

但是在子线程 new Handler 对象的话会报错：没有 Looper,不能 new Handler。

那么我们就必须先调用 Looper.prepare 方法准备一个 Looper 对象，然后调用 Looper.loop 方法开始轮询。

prepare 方法我们在第一个问题中分析了,会 new 一个 Looper 对象，然后通过 set 方法将其存储到 ThreadLocal 中。

那么我们来看看 loop 方法：

```
    public static void loop() {
    final Looper me = myLooper();
```

```
    if (me == null) {
        throw new RuntimeException("No Looper;
Looper.prepare() wasn't called on this thread.");
    }
    final MessageQueue queue = me.mQueue;

    // Make sure the identity of this thread is that of
the local process,
    // and keep track of what that identity token actually
is.
    Binder.clearCallingIdentity();
    final long ident = Binder.clearCallingIdentity();

    for (;;) {
        Message msg = queue.next(); // might block
        if (msg == null) {
            // 下面省略很多代码。。。。。。。
}
```

该方法我们省略了很多代码，但是以上这些足够我们来了解原理了。

首先第一行就通过 myLooper()方法（该方法在问题 1 中分析过），从 ThreadLocal 中取出 Looper 对象，并赋值给局部变量 me。

然后通过 me.mQueue 得到 MessageQueue 对象，接着立马通过 for(;;)开启一个无限循环，通过 MessageQueue 对象 queue.next()取出消息。

所以通过调用 Looper.prepare 和 Looper.loop 方法之后，我们就能够通过 Handler 来实现主线程向子线程传递数据。

3).Looper 是死循环为什么不会引起 ANR？

我们首先要找到调用 Looper.prepareMainLooper 和 Looper.loop 的地方，就是 ActivityThread.Java 类的 main 方法。这也是应用程序的入口

```
public static void main(String[] args) {
        。。。。。。。。。。
    //创建 Looper 和 MessageQueue
```

```
    Looper.prepareMainLooper();
    。。。。。。。。。。
   //开始无限循环轮询。
    Looper.loop();

    throw new RuntimeException("Main thread loop
unexpectedly exited");
}
```

发现了什么问题吗？如果 main 方法中没有 looper 进行循环，那么主线程一运行完毕就会退出。这还玩个蛋啊！

所以我们可以得出结论，**ActivityThread 的 main 方法主要就是做消息循环，一旦退出消息循环，那么你的应用也就退出了。**

既然我们知道了死循环的重要性，那么为什么死循环不会造成主线程阻塞呢？

首先我们要了解造成 ANR 的原因 :(1).当前事件没机会得到处理。(2).当前事件正在处理而没有及时完成。

因为 Android 是由事件驱动的，Looper.loop 不断接收事件、处理事件，每一个点击、触摸和 Activity 的生命周期都运行在 Looper.loop 的控制下，如果它停止了，应用程序也停了。如果某一个消息的处理时间过长，那么下一次用户的交互就得不到及时的处理，就产生了卡顿，时间长了就会 ANR。**所以只能说一个消息或者对消息的处理阻塞了 Looper.loop，不能说 Looper.loop 阻塞了主线程。**

而且，主线程 Looper 从消息队列中读取消息，当所有消息读完，主线睡眠，当子线程再次往消息队列中发消息时，主线程将被唤醒。主线程被唤醒只是为了读取消息，读取完毕再次睡眠因此 loop 循环并不会对 CPU 有过多的消耗。

总结 :只要 Looper.loop 的消息循环和处理没有被阻塞，那么久不会引起 ANR。

4).Looper 在取出消息后怎么能准确无误地将消息交给发送该消息的 Handler 对象？

这里从 Handler 的 obtainMessage 开始分析，该方法实际上调用的 Message.obtain(this)方法。看看 Message.obtain 方法的源码

```
    public static Message obtain(Handler h) {
    Message m = obtain();
    m.target = h;
```

```
    return m;
}
```

源码的实现很简单，就是先获取一个消息对象，再将 Handler 对象本身赋值给消息对象的 target 成员变量。也就是说，而 Looper 是通过 MessageQueue 的 next 方法取出消息的，通过取出的这个消息的 target 属性我们可以得到发送该消息的 Handler 对象，然后 looper 对象就能将消息交给发送给消息的 Handler 对象处理了。

7、 动画。

分类：View 动画、帧动画、属性动画。

View 动画:一共有四种 View 动画，ransationAnimation、RotateAnimation、ScaleAnimation、AlphaAnimation。实现 View 动画的方式有两种：xml 中配置和代码实现。可以通过 Animation 的 setAnimationListener 方法给 View 动画添加过程监听，可以监听 start、end、repeat 状态。Activity 和 Fragment 的转场动画也是通过 View 动画来实现的。**这里告诉大家一个用来装逼的动画 LayoutAnimation。用于为一个 layout 里面的控件，或者是一个 ViewGroup 里面的控件设置动画效果，可以在 XML 文件中设置，亦可以在 Java 代码中设置。**

帧动画：通过 XML 来定义一个 AnimationDrawable,然后将该 xml 文件设置成 View 的背景即可播放

属性动画：属性动画是 API11 之后加入的新特性，若要支持 API11 之前则使用 nineoldandroids 兼容包，该兼容库在 API11 之前实际上是通过代理 View 动画来实现的。通过 XML 文件来配置属性动画的时候存放在 res/animator 目录下

常见的属性动画类：

(1).ObjectAnimator。

使用objectAnimator要使动画生效的话需要满足两个条件：

1、执行动画的对象的属性必须提供setXXX方法，如果动画没有传递初始值，那么还要提供getXXX方法，因为系统要去取该属性的初始值（不满足则程序直接Crash）。

2、执行动画的对象的通过setXXX对XXX属性做的改变必须能够通过某种方法反映出来。（不满足则动画无效）

之所以要求提供set方法，是因为属性动画根据你传递的该属性的初始值和最终值，以动画的效果多次调用set方法，每次传递给set方法的值都不一样，最终接近最终值。

如果该对象的属性没有 setXXX 和 getXXX 方法怎么来实现属性动画呢？

(1).给你的对象的属性加上 get 和 set 方法。（源码一般改不了）

(2).用一个类包装原始对象，间接为其提供 get 和 set 方法。

(3).采用 ValueAnimator。监听动画过程，自己实现属性的改变。

(2).ValueAnimator：需要配合 AnimatorUpdateListener 来使用。在 onUpdate 方法中通过 animator.getAnimatedFraction()获取当前进度占整个动画过程的比例，然后再用估值器计算当前的属性值。

插值器InterPolator:常见的动画插值器有LinearInterpolator匀速插值器、DecelerateInterpolator减速插值器等等很多种。它的作用是根据时间流逝的百分比计算出当前属性改变的百分比。

估值器Evaluator：常见的估值器有IntEvaluator、FloatEvaluator、ArgbEvaluator等等。它的作用是根据当前属性改变的百分比来计算改变后的属性。

属性动画的监听器：主要有两个接口1).AnimatorListener。2).AnimatorUpdateListener。

矢量动画：不要求大家掌握，能了解最好，参考博客地址
http://blog.csdn.net/u013478336/article/details/52277875

使用动画的注意事项:

1).OOM 问题，尽量避免使用帧动画。

2).内存泄漏问题，在属性动画中有一类无限循环动画，在 activity 销毁的时候一定要停止动画。

3).版本兼容性问题，主要是属性动画。

4).View 动画只是对 View 的影像做动画。

5).在进行动画的过程中尽量使用 dp。

8、 强引用、软引用、弱引用、虚引用。

强引用：大部分情况下，我们使用的引用都是强引用，这也是最普遍的，强引用的对象就算内存不足 Java 虚拟机宁可抛出 OOM 来终止程序也不会回收强引用的对象。

软引用：在内存足够的情况下，垃圾回收器不会回收软引用对象，但是当内存不足的情况下就会回收。软引用的代码写法如下：

```
   Dog d = new Dog();

SoftReference<Dog> dog = new SoftReference<Dog>(d);
```

弱引用：弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。

虚引用：（可以不用了解太清楚）顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。虚引用主要用来跟踪对象被垃圾回收的活动。虚引用必须和引用队列（ReferenceQueue）联合使用。回收时，在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

9、 Volley 自定义请求。

自定义请求可以参考 StringRequest, 一共有三步, 下面就以自定义一个包含 Gson 解析的 Request——GsonRequest：

1).写一个 GsonRequest 继承 Request 类。

2).重写 parseNetWorkResponse 方法来解析服务器响应的数据。

```
    @Override
protected Response<T>
parseNetworkResponse(NetworkResponse response) {
    try
    {
        /**
         * 得到返回的数据
         */
        String jsonStr = new String(response.data);
        /**
         * 转化成对象
         */
        return Response.success(gson.fromJson(jsonStr,
clazz), HttpHeaderParser.parseCacheHeaders(response));
    } catch (JsonSyntaxException e)
    {
        return Response.error(new ParseError(e));
    }
}
```

3).重写 deliverRespone 方法，将解析的响应数据传递出去给主线程更新 UI。

```

@Override
protected void deliverResponse(T response) {
    //将数据传递出去。s
    listener.onResponse(response);
}
```

10、IPC 机制。

概述：IPC是Inter-Process Communication的缩写，含义为进程间通信或者跨进程通信，是指两个进程之间进行数据交换的过程。

多进程使用的方法：在manifist文件中指定四大组件的process属性。

Android中实现IPC机制的途径：Bundle、文件共享、AIDL、Messenger、ContentProvider、Socket。

IPC机制的应用场景：1).两个应用程序之间数据交换。2). 一个应用程序开启两个进程，这两个进程之间进行通信。（一个应用程序一般对应一个进程，也就是一台虚拟机，开辟的内存空间是16M。有的应用程序需要的内存空间较大，或者有的功能需要防止被杀死要在单独的进程中做，那我们就可以重新开辟一个进程。）我们应用程序中用到IPC机制基本上是指第二种情况。

应用程序使用多进程的隐患：

1).全局变量和单例会失效。

2). SharedPreferences 可靠性下降(SharedPreferences不支持多个进程同时写，会有一定的几率丢失数据)

3).Application会被创建多次，所以我们不能直接将一些数据保存在Application中。

4).线程同步机制完全失效，内存都不是同一块了，所以对象也不是同一个，还同步个鬼。

但是我们不能因为它有问题就不去使用了。那么我们怎么使用AIDL呢？

通过看官方文档我们能够很清楚的看到实现AIDL的步骤：

1).创建aidl文件，一般是先建一个interface注意，接口名和方法名前面都不能有修饰词，然后将该文件的后缀名改成.aidl。

2).eclipse或者AS自动将aidl文件编译成.java文件，放在gen目录下。

3).在服务端Service中写aidl文件中的抽象类Stub的实现对象，代码如下：

```
    private IMyAidlInterface.Stub mStub = new
IMyAidlInterface.Stub(){
    @Override
    public void registerTestCall() throws RemoteException
{
        //写具体逻辑实现。
```

```
    }
    @Override
    public void invokCallBack() throws RemoteException {
        //写具体方法实现。
    }
};
```

4).在客户端绑定绑定服务，在ServiceConnection的onServiceConnected方法中通过Stub.asInterface方法获取Stub的实现类对象。然后就可以通过该对象调用远程服务里的方法了。

```
    ServiceConnection conn = new ServiceConnection() {

    @Override
    public void onServiceDisconnected(ComponentName name)
{
    }

    @Override
    public void onServiceConnected(ComponentName name,
IBinder service) {
        remoteService =
IMyAidlInterface.Stub.asInterface(service);
        try {
            remoteService.invokCallBack();
            remoteService.registerTestCall();
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
};
```

那么什么类型的数据可以跨进程读写呢？

要进行跨进程读写的数据类型必须是可以被序列化的，所以基本类型和实现了 Serializable 接口或 Parcelable 接口的对象类型数据都可以跨进程读写。

实现 Parcelable 接口进行序列化比实现 Serializable 接口进行序列化效率要高得多，后者会产生很多中间变量导致频繁 GC，开销很大。所以在 Android 中我们一般都采用 Parcelable。

那么这两者具体怎么使用呢？

Serializable：类实现 Serializable 接口，在类的声明中指定 serializableUID，就可自动实现序列化。（制定该 UID 主要是为了反序列化），具体代码如下：

```
    public class UserA implements Serializable{
    private static final long serialVersionUID =
7247714666080613254L;
    public String name;
    public int age;
    public String school;
}
```

序列化过程：

```
    //序列化过程
UserA user = new UserA("leevi",18,"heima");
ObjectOutputStream out = null;
try {
    out = new ObjectOutputStream(new
FileOutputStream("cache.txt"));
    out.writeObject(user);
    out.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

反序列化过程：

```
    //反序列化过程
ObjectInputStream in = null;
try {
```

```
    in = new ObjectInputStream(new
FileInputStream("cache.txt"));
    UserA userA = (UserA) in.readObject();
    in.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Parcelable：(AS 有插件能直接生成)

1).写一个类实现 Parcelable 接口。

2).重写 writeToParcel 方法，将你的对象序列化为一个 Parcel 对象，即：将类的数据写入外部提供的 Parcel 中，打包需要传递的数据到 Parcel 容器保存，以便从 Parcel 容器获取数据。

3).重写 describeContents 方法，内容接口描述，默认返回 0 就行。

4).实例化静态内部类CREATOR实现接口Parcelable.Creator

public static final Parcelable.Creator<T> CREATOR

注意：其中 public static final 一个都不能少，内部对象 CREATOR 的名称也不能改变，全部要大写。需要重写本接口的两个方法：createFromParcel(Parcel in)实现从 Parcel 容器中读取传递数据值，封装成 Parcelable 对象返回逻辑层。newArray(int size)创建一个类型为 T，长度为 size 的数组，仅一句话即可（return new T[size]），供外部类反序列化本类数组使用。

简而言之：通关过 writeToParcel 将你的对象映射成 Parcel 对象，再通过 createFromParcel 将 Parcel 对象映射成你的对象。也可以将 Parcel 看成一个流，他能够过 writeToParcel 把对象写到流里面，再通过 createFromParcel 从流里读取对象，只不过这个过程需要你来实现，因此写的顺序与度的顺读必须一致。

```
    /**
 * Created by leevi on 16/9/23.
 */
public class Book implements Parcelable{
    public String bookName;
```

```
    public String author;
    public int publishDate;
    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(bookName);
        dest.writeString(author);
        dest.writeInt(publishDate);
    }

    public Book(String bookName,String author,int
publishDate) {
        this.bookName = bookName;
        this.author = author;
        this.publishDate = publishDate;
    }

    protected Book(Parcel in) {
        bookName = in.readString();
        author = in.readString();
        publishDate = in.readInt();
    }

    public static final Creator<Book> CREATOR = new
Creator<Book>() {
        @Override
        public Book createFromParcel(Parcel source) {
```

```
            return new Book(source);
        }

        @Override
        public Book[] newArray(int size) {
            return new Book[size];
        }
    };
}
```

超神篇

1、 版本迭代中的数据库更新问题。

应用程序如何知道数据库需要升级：SQLiteOpenHelper类的构造函数有一个参数是**int version**，它的意思就是指数据库版本号。比如在应用1.0版本中，我们使用SQLiteOpenHelper访问数据库时，该参数为1，那么数据库版本号1就会写在我们的数据库中。到了1.1版本，我们的数据库需要发生变化，那么我们1.1版本的程序中就要使用一个大于1的整数来构造SQLiteOpenHelper类，用于访问新的数据库，比如2。当我们的1.1新程序读取1.0版本的老数据库时，就发现老数据库里存储的数据库版本是1，而我们新程序访问它时填的版本号为2，系统就知道数据库需要升级。

何时触发数据库升级：当系统在构造 SQLiteOpenHelper 类的对象时，如果发现版本号不一样，就会自动调用 **onUpgrade** 函数，让你在这里对数据库进行升级。

升级时应该考虑的问题：数据库中的原数据不能丢失。

如何升级：在 onUpgrade 方法中执行相应的更新数据库的 sql 语句就行了。下面列举一些常见的更新的 sql 语句。

增加新表：CREATE TABLE table_name(_id integer primary key autoincrement, region varchar, code varchar)。

增加或者删除列：SQLite数据库对ALTER TABLE命令支持非常有限，只能在表末尾添加列，不能修改列定义，不能删除已有的列。那么如果要修改表呢？我们可以采用临时表的办法。具体来说有四步：

1).将表明改为临时表：ALTER TABLE table_name RENAME TO table_name _temp

2).创建新表。

3).将临时表的数据导入新表（不能出现values关键字）：**insert into** table_name (_id, region, code, country) **select** _id, region, code, \"CHINA\" **from** table_name _temp

4).删除临时表：**DROP TABLE** table_name _temp

跨版本的数据库升级问题：应用程序发布了多个版本，以致出现了三个及以上数据库版本。如何确保所有的用户升级应用后数据库都能用呢？

方法一：确定相邻版本的差别，从版本1开始依次迭代更新，先执行v1到v2，再v2到v3这样依次下去。

优点：每次更新数据库的时候只需要在onUpgrade方法的末尾加一段从上个版本升级到新版本的代码，易于理解和维护。

缺点：是当版本变多之后，多次迭代升级可能需要花费不少时间，增加用户等待时间；

方式二：为每个版本确定与现在数据库的差别，为每个case撰写专门的升级代码。

优点：则是可以保证每个版本的用户都可以在消耗最少的时间升级到最新的数据库而无需做无用的数据多次转存。

缺点：是强迫开发者记忆所有版本数据库的完整结构，且每次升级时onUpgrade方法都必须全部重写。

```
   @Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
int newVersion) {
   switch (newVersion) {
```

```
        case 2:
            db.execSQL(CREATE_TEMP_BOOK);
            db.execSQL(CREATE_BOOK);
            db.execSQL(INSERT_DATA);
            db.execSQL(DROP_BOOK);
            break;
        case 3:
            .....
            break;
    }
}
```

2、 数据库优化的方法。

(1).使用事务。大量数据增加或删除时如果不使用事务，而是循环调用 INSERT 和 DELETE 的话，会因为每一笔操作都需要打开、写入最后关闭 journal 文件（这个文件是临时用来保存数据操作的中间结果）而使得开销非常大。

使用方法：

1).db.beginTransaction()。

2).开始进行循环添加或者删除操作。

3). db.setTransactionSuccessful()。

4).db.endTransaction()。

(2).使用索引。索引维护着一个表中某一列或某几列的顺序，这样就可以快速定位到一组值，而不用扫遍全表。所有的索引信息会被保存在一个独立的索引表中，所以会产生额外的空间占用，不过绝对物超所值，特别是当你会在数据库中进行大量的读及搜索操作时。

创建语句：CREATE INDEX name_index **ON** username(firstname, lastname)

索引分类：（大概清楚就行）

a).普通索引和唯一性索引。

b).单个索引：索引建立语句中仅包含单个字段。

复合索引：在索引建立语句中同时包含多个字段。

c).聚簇索引：物理索引，与基表的物理顺序相同，数据值的顺序总是按照顺序排列。

非聚簇索。

使用场景：

(1).当某字段数据更新频率较低，查询频率较高，经常有范围查询(>, <, =, >=, <=)或 order by、group by 发生时建议使用索引。

(2).经常同时存取多列，且每列都含有重复值可考虑建立复合索引。

使用注意项（较难）：

(1). 对于复合索引，把使用最频繁的列做为前导列(索引中第一个字段)。

(2).避免对索引列进行计算，对 where 子句列的任何计算如果不能被编译优化，都会导致查询时索引失效。

(3).比较值避免使用 NULL。

(4).多表查询时要注意是选择合适的表做为内表。内外表的选择可由公式：外层表中的匹配行数*内层表中每一次查找的次数确定，乘积最小为最佳方案。

(5).把过滤记录数最多的条件放在最前面。

(3).sql 语句拼接时可以使用 StringBuffer 代替 String。

(4).在写表时调用 sqliteOpenHelper..getWritableDatabase()，在读表时候调用 sqliteOpenHelper..getReadableDatabase()，getReadableDatabase 性能更优。

(5).开启子线程进行数据库读写。

3、 性能优化。

布局优化：尽量减少布局文件的层级。

1).删除无用的控件和层级。使 HierarchyViewer 来查看布局层级。

2).**在布局层级相同的情况下**，使用 LinearLayout 的效率比使用 RelativeLayout 的性能要高，因为 RelativeLayout 的功能比较复杂，它的 CPU 渲染时间较长，而 LinearLayout 和 FramLayout 都是简单高效的 ViewGroup。

3).include 标签配合 merge 标签使用来重用布局、减少布局层数。

4).使用 ViewStub。它是个非常轻量级的 View，宽高都为 0，因此它本身不参与任何的布局和绘制过程。它的意义在于按需加载所需的布局，使用的时候再加载，

不需要的时候就不加载进来，提高程序初始化时的性能。注意 ViewStub 与设置 visibility 之间的差异：设置 visibility 只是设置可不可见，就算设置 INBISIBLE 或者 GONE，布局还是会加载。

绘制优化：

1).onDraw 方法里不要做初始化对象的操作，因为 onDraw 方法会被频繁调用。

2).onDraw 方法不要做耗时操作，这样会造成 View 的绘制过程不流畅。

内存泄漏优化：主要是分析会引起内存泄漏的原因。

1).资源未关闭。使用了 BraodcastReceiver, ContentObserver, File, Cursor, Stream, Bitmap 等资源的使用，应该在 Activity 销毁时及时关闭或者注销，否则这些资源将不会被回收，造成内存泄漏。

2).静态成员变量持有类的引用。代码如下：

```
    public class MainActivity extends AppCompatActivity {
    private static Context mContext;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mContext = this;
    }
}
```

由于 mContext 是静态成员变量，它的生命周期与应用程序的生命周期相同，而它又持有了 MainActivity 的引用，所以 Activity 无法正常销毁。

3).非静态内部类持有外部类的引用，使用非静态内部类创建静态变量。代码如下：

```
    public class MainActivity extends AppCompatActivity {
    private static ViewHolder mHolder = null;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        if(mHolder == null){
            mHolder = new ViewHolder();
        }
        //...
    }
    class ViewHolder {
        //...
    }
}
```

ViewHolder 类是非静态内部类，它默认持有外部类 MainActivity 的引用，又在 MainActiivty 中声明了 ViewHolder 为静态成员变量，所以它的生命周期和应用程序一样长。所以导致 MainActivity 无法正常销毁。

4).单例引起内存泄漏。因为单例的生命和应用程序的生命周期一样，如果单例对象持有了某个不再需要的对象的引用时，会导致该对象无法被正常回收，导致内存泄漏。

5).Handler 造成内存泄漏。由于 mHandler 是 Handler 的非静态匿名内部类的实例，所以它持有外部类 Activity 的引用，我们知道消息队列是在一个 Looper 线程中不断轮询处理消息，那么当这个 Activity 退出时消息队列中还有未处理的消息或者正在处理消息，而消息队列中的 Message 持有 mHandler 实例的引用，mHandler 又持有 Activity 的引用，所以导致该 Activity 的内存资源无法及时回收，引发内存泄漏。

解决办法：创建一个静态 Handler 内部类，然后对 Handler 持有的对象使用弱引用，这样在回收时也可以回收 Handler 持有的对象，避免了 Activity 泄漏，在 Activity 的 Destroy 时或者 Stop 时应该移除消息队列中的消息。

代码如下：

```
public class MainActivity extends AppCompatActivity {
    private MyHandler mHandler = new MyHandler(this);
    private TextView mTextView ;
    private static class MyHandler extends Handler {
        private WeakReference<Context> reference;
        public MyHandler(Context context) {
```

```
            reference = new WeakReference<>(context);
        }
        @Override
        public void handleMessage(Message msg) {
            MainActivity activity = (MainActivity)
reference.get();
            if(activity != null){
                activity.mTextView.setText("");
            }
        }
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mTextView = (TextView)findViewById(R.id.textview);
        loadData();
    }

    private void loadData() {
        //...request
        Message message = Message.obtain();
        mHandler.sendMessage(message);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        mHandler.removeCallbacksAndMessages(null);
    }
}
```

6).线程生命周期不可控。使用线程是如果不将 Runnable 内部类、AsyncTask 内部类生命成静态的话，那它就会持有外部类（如 Activity 的引用），一旦当 Activity 要退出的时候线程任务还没执行完的话将会导致 Activity 无法正常销毁。代码如下：

```
    new Thread(new Runnable() {
    @Override
    public void run() {
        SystemClock.sleep(10000);
    }
}).start();
```

或者：

```
    new AsyncTask<Void, Void, Void>() {
    @Override
    protected Void doInBackground(Void... params) {
        SystemClock.sleep(10000);
        return null;
    }
}.execute();
```

7).无限循环的属性动画引起内存泄漏。由于 Activity 中的 View 会被动画持有，而 Activity 又持有 Activity，最终会导致 Activity 无法被正常销毁。解决办法是：在 Activity 的 onStop 方法或者 onDestroy 方法中停止动画。最好是在 onStop 中停止。

数据库优化、图片优化、大数据量优化、线程池优化等，这些都单独列出来了。

优化之工具篇：HierarchyViewer(检查布局层级)、TraceView(对应用中方法耗时进行统计)、APT(对进程的 CPU 和内存进行监视很分析)、MAT(进行内存泄露分析的主要工具)。

4、 大图片优化。

(1).通过 BitmapFactory.Options 来缩放图片，主要用到了它的 inSampleSize 参数即采样率。当 inSampleSize 为 1 时，采样后的大小为图片的原

始大小。当 inSampleSize 为 2 时，采样后图片的宽高均为原图的 1/2，像素数为原图的 1/4，占内存大小也为原图的 1/4。

获取合适采样率的方法如下：

1).将 BitmapFactory.Options 的 inJustDecodeBounds 参数设置为 true 并加载图片，当设置为 true 时 BitmapFactory 只会去解析原图的宽高，并不会去真正加载图片。

2).从 BitmapFactory.Options 中取出图片的原始宽高信息。

3).根据采样率的规则，并结合目标 View 的所需大小计算采样率 inSampleSize。

4).将 BitmapFactory.Options 的 inJustDecodeBouns 参数设为 false，然后重新加载图片。

(**2).采用缓存策略。**目前常用的缓存算法是 LRU(Least Recently Used),近期最少使用算法。它的核心思想就死当缓存满时，会优先淘汰那些近期最少使用的缓存对象。采用 LRU 算法的缓存有两种：LruCache 和 DiskLruCache。

LruCache 用于实现内存缓存，DiskLruCache 用于实现存储设备缓存。

LruCache:是一个泛型类，它内部采用 LinkedHashMap 以强引用的方式存储外界的缓存对象，并采用 get 和 put 方法来完成缓存的获取和添加操作，当缓存满时 LruCache 会移除较早使用的缓存对象，然后添加新的缓存对象。

DisLruCache:用于磁盘缓存。它通过将缓存对象写入文件系统从而实现缓存效果。

三级缓存：实际上是内存和磁盘两级缓存。首先尝试从内存缓存中读取图片，如果内存中没有该图片缓存接着尝试从磁盘缓存中读取图片，如果磁盘缓存中也没有最后才考虑从网络加载图片。

5、 数据量大的优化。

一般涉及到的都是 ListView 加载大量数据时的优化问题，可参考 ListView 的优化。

1).动态加载，也是分页加载数据。仅在需要加载的时候加载数据项。ListView 有这个回调（onScroll 和 onScrollStateChanged），我们结合这二个重载的函数，检查当滚动状态改变。

2).对 ListView 中的图片进行处理。参考**图片优化。**

6、 四大图片加载框架的对比。

这里主要列出四大图片加载框架各自的优点。

UniversalImageLoader:

1).支持下载进度监听。

2).支持本地缓存文件名规则定义。

3).可以在 View 的滚动中停止加载图片。

4). 默认实现多种内存缓存算法,这几个图片缓存都可以配置缓存算法,不过 ImageLoader 默认实现了 较多缓存算法,如 Size 最大先删除、使用最少先删除、最近最少使用、先进先删除、时间最长先删除 等。

Picasso:

1).自带统计监控功能。支持图片缓存使用的监控,包括缓存命中率、已使用内存大小、节省的流量等。

2).支持优先级处理。

3). 支持飞行模式、并发线程数根据网络类型而变 手机切换到飞行模式或网络类型变换时会自动调整线程池最大并发数,比如 wifi 最大并发为 4,4g 为 3,3g 为 2。

4).支持延迟到图片尺寸计算完成时加载。

Glide:

1).支持gif、webp、缩略图等。

2).支持优先级处理。

3).与Activity/fragment生命周期保持一致，支持trimMemory。

4). Glide 的内存缓存有个 active 的设计，从内存缓存中取数据时,不像一般的实现用 get,而是用 remove,再将这个缓存数据放到一个 value 为软引用的 activeResources map 中,并计数引用数,在图片加载完成后进行判断,如果引用计数为空 则回收掉。

5). 内存缓存更小图片,Glide 以 url、view_width、view_height、屏幕的分辨率等做为联合 key,将处理后的图片缓存在内 存缓存中,而不是原始图片以节省大小

Fressco：可以说是综合了之前图片加载库的优点，其在5.0以下的内存优化非常好，但它的不足是体积太大。Fresco 在图片较多的应用中更能凸显其价值，如果应用没有太多图片需求，不推荐使用 Fresco。

7、 Volley 的源码分析。

对 Volley 创建 Request 的时候做了什么事：

```
    public Request(int method, String url,
Response.ErrorListener listener) {
    mMethod = method;
    mUrl = url;
    mErrorListener = listener;
    setRetryPolicy(new DefaultRetryPolicy());

    mDefaultTrafficStatsTag =
findDefaultTrafficStatsTag(url);
}
```

保存请求的 url、请求方法、成功的回调、失败的回调。

创建 RequestQueue 的时候做了什么事：

我们先找到 Volley 类中的 newRequestQueue 方法,通过看它的源码我们能发现。实际上是先创建一个 Network 对象传入 RequestQueue 的构造函数中来创建一个 RequestQueue 对象最后再调用 queue 的 start 方法。

```
    Network network = new BasicNetwork(stack);

RequestQueue queue = new RequestQueue(new
DiskBasedCache(cacheDir), network);
queue.start();
```

那么我们就要看看 Network 是个什么东西，照我们现在看来 Network 应该就是真正实现网络请求的地方。

我们可以看到在创建 Network 对象的时候传入了一个 stack，我们先找到 stack 被赋值的代码：

```
    if (stack == null) {
    if (Build.VERSION.SDK_INT >= 9) {
        stack = new HurlStack();
    } else {
                stack = new
HttpClientStack(AndroidHttpClient.newInstance(userAgent))
;
    }
}
```

可以看到，当 SDK 版本大于 9 的时候是 new HurlStack()来给 stack 赋值，小于 9 时是 new HttpClientStack()来给 stack 赋值。

通过查看 HurlStack 和 HttpClientStack 我们能够发现在这两个类中都有个 performRequest 方法，用于处理网络请求。HurlStack 使用 HttpURLConnection 来请求网络，HttpClientStack 中使用 HttpClient 来请求网络。

所以我们可以得出结论真正发起网络请求的是实现了 HttpStack 接口的 HurlStack 和 HttpClientStack 类，SDK 版本小于 9 使用 HttpClient 否则使用 HttpURLConnection。

接下来看看在将 request 添加到 requestQueue 中去的时候做了些啥：

首先我们能看到

```
    synchronized (mCurrentRequests) {
    mCurrentRequests.add(request);
}
```

有一个 **mCuttentRequests** 成员变量，调用它的 add 方法添加 request。mCurrentRequestQueue 是一个 HashSet 类型，用来存储所有还没完成的 request。相对应的，在请求完成后在 finish 方法中会调用调用 mCurrentRequestQueue 的 remove 方法移除 request。

```
    void finish(Request<T> request) {
    // Remove from the set of requests currently being
processed.
    synchronized (mCurrentRequests) {
        mCurrentRequests.remove(request);
    }
    ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦
    }
```

接下来我们看到通过 request.shouldCache()判断是否需要缓存，默认是需要缓存的，将所有请求存放到 mCacheQueue 中。

```
    mCacheQueue.addAll(waitingRequests);
```

mCacheQueue：用于存放需要先从缓存获取数据的请求，默认情况下所有请求先加入 mCacheQueue 中。

看到上述代码 mCacheQueue 中添加的是 waitingRequests，我们首先要弄清楚 mWaitingRequests 是什么：

mWaitingRequests：是一个 map，key 是 cacheKey——>请求的 uri，存放同一个 uri 下第一次请求后发起的多个请求（就是第 2、3、4 等次请求），这些请求不用直接去访问网络，而是直接从缓存中获取数据。

在 add 方法中有如下代码：

```
    String cacheKey = request.getCacheKey();
    if (mWaitingRequests.containsKey(cacheKey)) {
    // There is already a request in flight. Queue up.
    Queue<Request<?>> stagedRequests =
mWaitingRequests.get(cacheKey);
    if (stagedRequests == null) {
        stagedRequests = new LinkedList<Request<?>>();
    }
    stagedRequests.add(request);
    mWaitingRequests.put(cacheKey, stagedRequests);
```

意思就是说，如果 **mWaitingRequests 包含了 request 的 uri 的话（表明不是第一次发起该请求了），就直接将该请求放入 mWaitingRequests 中。**

这样的话我们就很好理解上面的，

```
    mCacheQueue.addAll(waitingRequests);
```

添加到 mCacheQueue 中的是第一次以后发起的请求。

如果不需要缓存就直接将 request 添加到 mNetWorkQueue。那么我们又看到了另外一个成员变量 mNetWorkQueue，这个队列用来存放所有直接从网络获取数据的 request。

```
    if (!request.shouldCache()) {
    mNetworkQueue.add(request);
    return request;
}
```

添加 request 到 requestQueue 中之后，我们再看看 RequestQueue 的 start 方法里面实现了什么：

```
    public void start() {
    stop();  // Make sure any currently running
    mCacheDispatcher = new CacheDispatcher(mCacheQueue,
mNetworkQueue, mCache, mDelivery);
    mCacheDispatcher.start();
    for (int i = 0; i < mDispatchers.length; i++) {
        NetworkDispatcher networkDispatcher = new
NetworkDispatcher(mNetworkQueue, mNetwork,
                mCache, mDelivery);
        mDispatchers[i] = networkDispatcher;
        networkDispatcher.start();
    }
}
```

在这里我们能看到两个很重要的类 **CacheDispatcher** 和 **NetworkDispatcher**。分别调用它们的 start 方法

CacheDispatcher：是一个线程，里面有个死循环，不断到 mCachQueue 中取缓存数据，如果没有缓存或者缓存过期则将请求添加到 mNetworkQueue 中。

NetworkDispatcher：也是线程，总共有四个这样的线程，也是一个死循环，不断从 mNetworkQueue 中拿取请求，判断如果没有被 cancel 的话就发起网络请

求，由 mNetwork.performRequest(request)来发起网络请求，里面真正实现的是 HttpStack 的 performRequest 来请求网络。

然后调用 parseNetworkResponse 方法解析 response 响应数据，并写入缓存。

最后在 **CacheDispatcher 和 NetworkDispatcher 的 run 方法里面，**由 ResponseDelivery 接口的实现 ExecutorDelivery 来发布结果。

ExecutorDelivery：构造函数中传入一个 Handler 对象，内部有一个 Executor 类型的成员变量。

```
    public ExecutorDelivery(final Handler handler) {
    mResponsePoster = new Executor() {
        @Override
        public void execute(Runnable command) {
            handler.post(command);
        }
    };
}
```

Executor 类型的成员变量内部有个 execute 方法，具体如上：调用了 handler 的 post 方法，很明显使用 post 将结果发布到主线程执行。

```
    mResponsePoster.execute(new
ResponseDeliveryRunnable(request, response, runnable));
```

我们可以看到 execute 方法中传入的 Runnable 类型的数据是，ResponseDeliveryRunnable。

ResponseDeliveryRunnable：我们看看它里面的 run 方法就知道具体是怎么来发布结果的。

```
    public void run() {
     if (mRequest.isCanceled()) {
         mRequest.finish("canceled-at-delivery");
         return;
     }
     if (mResponse.isSuccess()) {
         mRequest.deliverResponse(mResponse.result);
     } else {
```

```
            mRequest.deliverError(mResponse.error);
        }
        if (mResponse.intermediate) {
            mRequest.addMarker("intermediate-response");
        } else {
            mRequest.finish("done");
        }
        if (mRunnable != null) {
            mRunnable.run();
        }
}
```

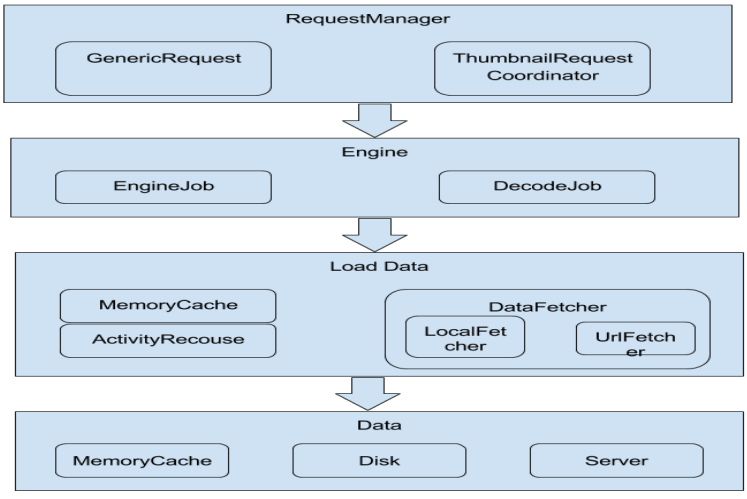
可以看到，如果成功则调用 deliverResponse 方法（该方法在自定义 Request 中讲过）将结果传递给外部回调。如果失败则调用 deliverError 方法将错误信息传递给失败的回调。

8、 OkHttp 的源码分析。

由于维哥目前还没有对 OkHttp 的源码有较为深入的研究，故提供一篇个人觉得不错的博客供大家学习。[OkHttp 源码解析]

9、 Glide 的源码分析。

Glide 是谷歌在 2014 年 IO 大会上提出来的，属于谷歌的亲儿子，深受广大开发者喜爱。



针对上图，我们再了解一些基本概念：

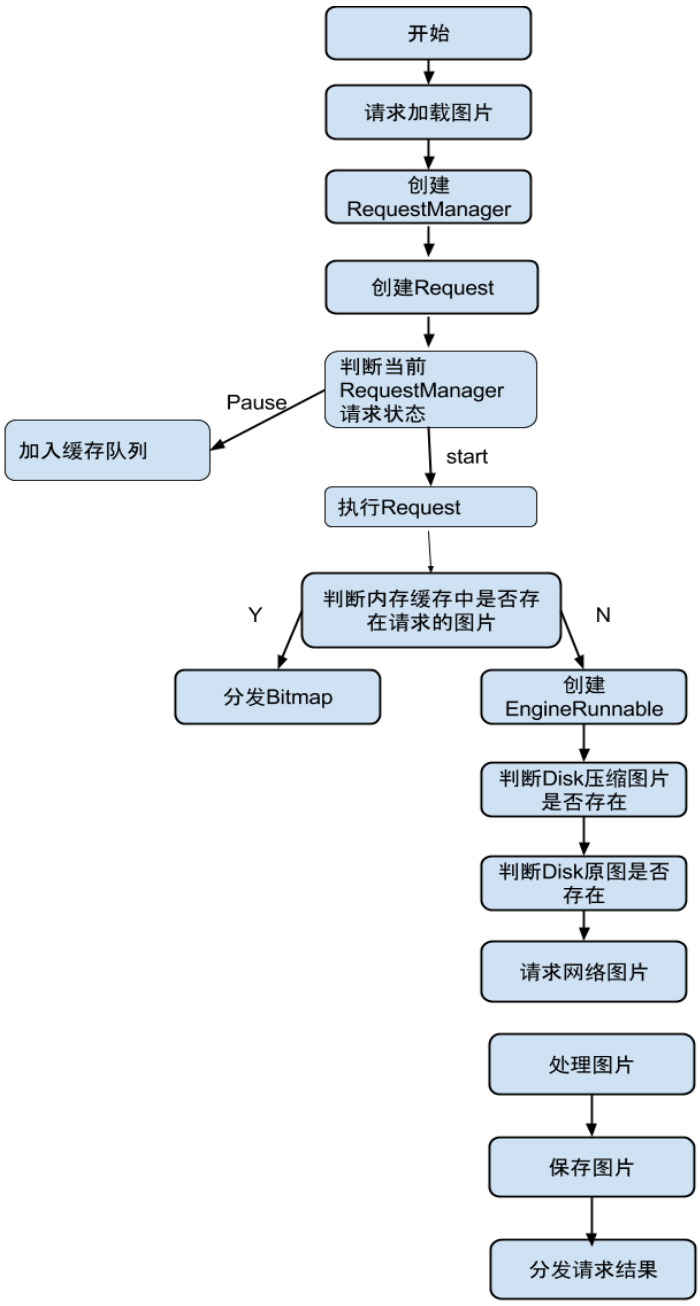
1). RequestManager :请求管理, 每一个 Activity 都会创建一个 RequestManager, 根据对应 Activity 的生命周期管理该 Activity 上所以的图片请求。

2). Engine：加载图片的引擎，根据 Request 创建 EngineJob 和 DecodeJob。

3).EngineJob：图片加载。

4).DecodeJob：图片处理。

下面给出大致的内部运行流程图：



下面介绍一些重要的类：

1).Glide:用于保存整个框架中的配置。

重要的方法是：**with**

```
    public static RequestManager with(FragmentActivity
activity) {
    RequestManagerRetriever retriever =
RequestManagerRetriever.get();
    return retriever.get(activity);
}
```

用于创建RequestManager，这里是Glide通过Activity/Fragment生命周期管理Request原理所在，这个类很关键、很关键、很关键，重要的事情我只说三遍。主要原理是创建一个自定义 Fragment，然后通过自定义 Fragment 生命周期操作RequestManager，从而达到管理 Request。

2).RequestManagerRetriever：这个类中包含一系列的静态方法去创建RequestManager 和检索现有的 activity 和 fragment。

重要的方法：supportFragmentGet 方法

```
    RequestManager supportFragmentGet(Context context,
FragmentManager fm) {
    SupportRequestManagerFragment current =
getSupportRequestManagerFragment(fm);
    RequestManager requestManager =
current.getRequestManager();
    if (requestManager == null) {
        requestManager = new RequestManager(context,
current.getLifecycle(),
current.getRequestManagerTreeNode());
        current.setRequestManager(requestManager);
    }
    return requestManager;
}
```

这里判断是否只当前 RequestManagerFragment 是否存在 RequestManager，保

证一个 Activity 对应一个 RequestManager， 这样有利于管理一个 Activity 上所有的 Request。创建 RequestManager 的时候会将 RequestManagerFragment 中的回调接口赋值给 RequestManager，达到 RequestManager 监听 RequestManagerFragment 的生命周期。

3).RequestManager:用于管理和开启 Request。

成员变量：

(1)Lifecycle lifecycle,用于监听RequestManagerFragment生命周期。

(2)RequestTracker requestTracker，用于保存当前 RequestManager 所有的请求和带处理的请求。

重要方法：

```
    @Override
public void onStart() {//开始暂停的请求
    resumeRequests();
}
@Override
public void onStop() {//停止所有的请求
    pauseRequests();
}
@Override
public void onDestroy() {//关闭所有的请求。
    requestTracker.clearRequests();
}
    //创建 RequestBuild
    public DrawableTypeRequest<String> load(String string)
{
    return (DrawableTypeRequest<String>)
fromString().load(string);
}
```

4).DrawableRequestBuilder:用于创建Request。 这里面包括很多方法，主要是配置加载图片的url、大小、动画、ImageView对象、自定义图片处理接口等。

5). Request :它是一个接口，主要实现类是GenericRequest，主要是操作请求，加载资源到一个目标对象。

6).Engine：请求引擎，主要做请求的开始的初始化。

主要方法：

(1).load方法。

该方法主要有以下功能。

a).获取MemoryCache中缓存。首先创建当前Request的缓存key，通过key值从MemoryCache中获取缓存，判断缓存是否存在。

b).获取activeResources中缓存

activeResources通过弱引用保存recouse ，也是通过key获取缓存。

c).判断当前的请求任务是否已经存在,如果任务请求已经存在,直接将回调事件传递给已经存在的EngineJob，用于请求成功后触发回调。

d).执行请求任务。

7). EngineRunnable

请求执行Runnable，主要功能请求资源、处理资源、缓存资源。

重要方法有两个：

a).decodeFromCache:加载DiskCache。

b).decodeFromSource:加载网络资源。

8).DecodeJob:负责处理从缓存中的资源或者网络上的资源。

9).ResourceDecoder: 用于将文件、IO流转化为Resource。

10).BitmapPool:用于存放从LruCache中remove的Bitmap， 用于后面创建Bitmap时候的重复利用。

10、 对线程池的理解。

使用的原因：

1).减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

2).可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约 1MB 内存，线程开的越多，消耗的内存也就越大，最后死机)。

线程池的分类：

线程池都是通过 Executors 来创建的。

1).newCachedThreadPool 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

2). newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

3). newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。

4). newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序执行。

参数解析：

1).corePoolSize： 线程池维护线程的最少数量

2).maximumPoolSize：线程池维护线程的最大数量

3).keepAliveTime： 线程池维护线程所允许的空闲时间

4).workQueue： 线程池所使用的缓冲队列

5).handler： 线程池对拒绝任务的处理策略

创建规则：

当一个任务通过execute(Runnable)方法欲添加到线程池时：

1).如果此时线程池中的数量小于corePoolSize，即使线程池中的线程都处于空闲状态，也要创建新的线程来处理被添加的任务。

2).如果此时线程池中的数量等于 corePoolSize，但是缓冲队列 workQueue未满，那么任务被放入缓冲队列。

3).如果此时线程池中的数量大于corePoolSize，缓冲队列workQueue满，并且线程池中的数量小于maximumPoolSize，建新的线程来处理被添加的任务。

4).如果此时线程池中的数量大于corePoolSize，缓冲队列workQueue满，并且线程池中的数量等于maximumPoolSize，那么通过 handler所指定的策略来处理此任务。也就是：处理任务的优先级为：核心线程corePoolSize、任务队列workQueue、最大线程maximumPoolSize，如果三者都满了，使用handler处理被拒绝的任务。

5).当线程池中的线程数量大于 corePoolSize 时，如果某线程空闲时间超过keepAliveTime，线程将被终止。这样，线程池可以动态的调整池中的线程数。

停止和关闭线程池：

ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：
shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务
shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

11、优化 JNI 的调用。（感谢林锐波老师提供）

思路:
在 Java 中声明一个 native 方法，然后生成本地接口的函数原型声明，再用 C/C++ 实现这些函数，并生成对应平台的动态共享库放到 Java 程序的类路径下，最后在 Java 程序中调用声明的 native 方法就间接的调用到了 C/C++ 编写的函数了，**在 C/C++ 中写的程序可以避开 JVM 的内存开销过大的限制、处理高性能的计算、调用系统服务等功能。
遇到的问题:
其实在JNI中,与java最常接触的无非就是查找 class 和 ID (属性和方法 ID),但是这个查找的过程是十分消耗时间的.
解决方法:
因此在 native 里保存 class 和 member id 是很有必要的,但是class 和 member id 在一定范围内是稳定的**，但在动态加载的 class loader 下，保存全局的 class 要么可能失效，要么可能造成无法卸载classloader,在诸如OSGI(j2e的东西,自己百度) 框架下的 JNI 应用还要特别注意这方面的问题。
总结:
所以在 JNI 开发中，合理的使用缓存技术能给程序提高极大的性能。缓存有两种，分别为使用时缓存和类静态初始化时缓存，区别主要在于缓存发生的时刻。
使用时缓存:
字段 ID、方法 ID 和 Class 引用在函数当中使用的同时就缓存起来.
判断字段 ID 是否已经缓存，如果没有先取出来存到fid_str中，下次再调用的时候该变量已经有值了，不用再去JVM中获取，起到了缓存的作用。
遇到的坑:

但是请注意：cls_string是一个局部引用，与方法和字段 ID 不一样，局部引用在函数结束后会被 JVM 自动释放掉，这时cls_string成为了一个野针对（指向的内存空间已被释放，但变量的值仍然是被释放后的内存地址，不为NULL），当下次再调用 Java_com_xxxx_newString 这个函数的时候，会试图访问一个无效的局部引用，从而导致非法的内存访问造成程序崩溃。所以在函数内用 static 缓存局部引用这种方式是错误的。

类静态初始化时缓存:

在调用一个类的方法或属性之前，Java 虚拟机会先检查该类是否已经加载到内存当中，如果没有则会先加载，然后紧接着会调用该类的静态初始化代码块，所以在静态初始化该类的过程当中计算并缓存该类当中的字段 ID 和方法 ID 也是个不错的选择。

两种缓存方式比较

如果在写 JNI 接口时，不能控制方法和字段所在类的源码的话，用使用时缓存比较合理。但比起类静态初始化时缓存来说，用使用时缓存有一些缺点：

使用前，每次都需要检查是否已经缓存该 ID 或 Class 引用

如果在用使用时缓存的 ID，要注意只要本地代码依赖于这个 ID 的值，那么这个类就不会被 unload。另外一方面，如果缓存发生在静态初始化时，当类被 unload 或 reload 时，ID 会被重新计算。因为，尽量在类静态初始化时就缓存字段 ID、方法 ID 和类的 Class 引用。

12、蓝牙开发。

市面上的无线传输方案：WIFI、NFC、蓝牙。

应用场景：

1).WIFI——>智能家居、无人机。

2).NFC——>手机支付、门禁、信用卡、AndroidBeam(两台手机碰一下就能传输数据)。

3).蓝牙——>蓝牙耳机、音响、无线键鼠、可穿戴设备、车载设备。

蓝牙开发初步了解：

传输距离是 10 米以内，理论传输速度是 24Mbps，最新标准是蓝牙 4.0。

关键 API：

1).添加权限：普通权限 android.permission.BLUETOOTH，高级权限（进行配对时

需要）android.permission.BLUETOOTH_ADMIN

2). BluetoothAdapter adapter= BluetoothAdapter.getDefaultAdapter()； 如果mBluetoothAdapter为空，表示设备不支持蓝牙。

3).adapter.isEnable(),判断蓝牙是否打开。

4). Intent intent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
activity.startActivityForResult(intent,requestCode)；通过意图打开蓝牙。

5).adapter.disable(),关闭蓝牙。

6).通过广播接收者来监听蓝牙打开的状态，注册广播接收者的代码如下：

```
IntentFilter filter = new IntentFilter(BluetoothAdapter.ACTION_STATE_CHANGED);
registerReceiver(receiver, filter);
```

在广播接收者中通过intent. getIntExtra(BluetoothAdapter.EXTRA_STATE,-1)获取蓝牙状态，一共有四种状态：BluetoothAdapter.STATE_OFF、BluetoothAdapter.STATE_ON、BluetoothAdapter.STATE_TURNING_ON、BluetoothAdapter.STATE_TURNING_OFF。

7). mAdapter.startDiscovery，开始查找设备。

8).打开设备可见性。

```
Intent intent= new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
intent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 300);
context.startActivity(discoverableIntent);
```

9).通过广播接收者可以获取查找到的设备，注册广播接收者的代码如下：

```
IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND)
registReciver(mReciver,filter);
```

在广播接收者中通过intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);获取查找到的设备。

10). BlueToothDevice类中createBond方法能绑定设备,该方法在API19后才生效。

蓝牙连接和传输数据：

服务端创建过程：

1).通过mAdapter.listenUsingRfcommWithServiceRecord创建一个BluetoothServerSocket。

2).监听网络accept，即mBluetoothServerSocket.accept()获取到客户端的socket。

3).处理网络socket。

4).关闭连接。

客户端创建过程：

1).通过mAdapter.createRfcommWithServiceRecord创建一个BluetoothSocket。

2).连接服务器connect，mBluetoothSocket.connect()。

3).处理数据。

4).关闭连接。

13、 EventBus 的源码分析。

通过 EventBus 的使用，我们可以知道 EventBus 类是单例类，所以我们就先看看创建出该实例的时候做了什么事情：

```
    EventBus(EventBusBuilder builder) {
    this.currentPostingThreadState = new ThreadLocal() {
        protected EventBus.PostingThreadState initialValue() {
            return new EventBus.PostingThreadState();
        }
    };
    this.subscriptionsByEventType = new HashMap();
    this.typesBySubscriber = new HashMap();
    this.stickyEvents = new ConcurrentHashMap();
    this.mainThreadPoster = new HandlerPoster(this,
Looper.getMainLooper(), 10);
    this.backgroundPoster = new BackgroundPoster(this);
    this.asyncPoster = new AsyncPoster(this);
    this.subscriberMethodFinder = new
SubscriberMethodFinder(builder.skipMethodVerificationForClasses);
    this.logSubscriberExceptions = builder.logSubscriberExceptions;
    this.logNoSubscriberMessages = builder.logNoSubscriberMessages;
    this.sendSubscriberExceptionEvent =
builder.sendSubscriberExceptionEvent;
    this.sendNoSubscriberEvent = builder.sendNoSubscriberEvent;
```

```
    this.throwSubscriberException = builder.throwSubscriberException;
    this.eventInheritance = builder.eventInheritance;
    this.executorService = builder.executorService;
}
```

通过以上代码我们可以看到，在创建实例的时候做了很多初始化操作，如数据结构，handler，Runnble、线程池等初始化。

然后，我们在跟着 EventBus 的使用来看看 register 方法:

EventBus 类中提供给我们的一共有四个公开的 register 方法：

```
    public void register(Object subscriber) {
    this.register(subscriber, false, 0);
}

public void register(Object subscriber, int priority) {
    this.register(subscriber, false, priority);
}

public void registerSticky(Object subscriber) {
    this.register(subscriber, true, 0);
}

public void registerSticky(Object subscriber, int priority) {
    this.register(subscriber, true, priority);
}
```

这四个方法中又殊途同归地调用了同一个方法来实现具体逻辑：

```
    private synchronized void register(Object subscriber,
boolean sticky, int priority) {
    //根据订阅者字节码文件找出订阅者所有的订阅方法
    List subscriberMethods =
this.subscriberMethodFinder.findSubscriberMethods(subscriber.g
etClass());
    Iterator i$ = subscriberMethods.iterator();
```

```
    while(i$.hasNext()) {
        SubscriberMethod subscriberMethod =
(SubscriberMethod)i$.next();
        this.subscribe(subscriber, subscriberMethod, sticky,
priority);
    }
}
```

在上述方法中，先是通过 subscriberMethodFinder.findSubscriberMethods 方法，根据订阅者的字节码查找订阅者所有的订阅方法。

那我们就进入到 findSubscriberMethods 方法内部看看它到底是怎么来找出订阅者所有的订阅方法的：

这个方法代码比较多，我先给大家在关键的地方加上注释（在文档上可能不好看，大家将注释按照我的加到源码中去分析），然后我们来一步步分析：

```
    List<SubscriberMethod> findSubscriberMethods(Class<?>
subscriberClass) {
    //先根据字节码文件获取类名
    String key = subscriberClass.getName();
    Map clazz = methodCache;
    //定义一个 List 来保存所有的订阅方法。
    List subscriberMethods;
    //先从缓存中查找订阅方法。
    synchronized(methodCache) {
        subscriberMethods = (List)methodCache.get(key);
    }
    //如果缓存中有，那么直接将其 return 出去
    if(subscriberMethods != null) {
        return subscriberMethods;
    } else {
     //如果缓存中没有就创建一个 ArrayList 来存储订阅方法
        ArrayList var23 = new ArrayList();
        Class var24 = subscriberClass;
```

```
        HashSet eventTypesFound = new HashSet();
    //这里是先循环找出订阅者的所有父类
        for(StringBuilder methodKeyBuilder = new
StringBuilder(); var24 != null; var24 = var24.getSuperclass())
{
            String name = var24.getName();
    //排除系统定义的类
            if(name.startsWith("java.") ||
name.startsWith("javax.") || name.startsWith("android.")) {
                break;
            }
    //通过反射找出所有的订阅方法
            Method[] methods = var24.getDeclaredMethods();
            Method[] arr$ = methods;
            int len$ = methods.length;
    //遍历出每一个方法
            for(int i$ = 0; i$ < len$; ++i$) {
                Method method = arr$[i$];
    //通过反射得到方法名
                String methodName = method.getName();
    //判断方法名是不是以 onEvent 开头，如果是的那才是我们需要的方法
                if(methodName.startsWith("onEvent")) {
    //通过反射获取方法的修饰符
                    int modifiers = method.getModifiers();
    //如果修饰符是 public 的，才是我们需要的方法
                    if((modifiers & 1) != 0 && (modifiers & 5192)
== 0) {
    //通过反射获取方法的参数
                        Class[] parameterTypes =
method.getParameterTypes();
```

```
    //一般情况下都只有一个参数，我们也只处理一个参数的
                    if(parameterTypes.length == 1) {
    //截取方法名”onEvent”后面的部分，用来判断是四个方法中的哪一种，确定
对应的是哪种线程模型。
                     String modifierString =
methodName.substring("onEvent".length());
                        ThreadMode threadMode;
                        if(modifierString.length() == 0) {
                            threadMode = ThreadMode.PostThread;
                        } else
if(modifierString.equals("MainThread")) {
                            threadMode = ThreadMode.MainThread;
                        } else
if(modifierString.equals("BackgroundThread")) {
                            threadMode =
ThreadMode.BackgroundThread;
                        } else {
                            if(!modifierString.equals("Async"))
{

if(!this.skipMethodVerificationForClasses.containsKey(var24))
{
                                    throw new
EventBusException("Illegal onEvent method, check for typos: "
+ method);
                                }
                                continue;
                            }

                            threadMode = ThreadMode.Async;
                        }
```

```
                    Class eventType = parameterTypes[0];
                    methodKeyBuilder.setLength(0);
                    methodKeyBuilder.append(methodName);

methodKeyBuilder.append('>').append(eventType.getName());
                    String methodKey =
methodKeyBuilder.toString();
                    if(eventTypesFound.add(methodKey)) {
    //将 method、threadMode、eventType 封装成 SubscriberMethod 并添
加到存储它的集合中
                        var23.add(new
SubscriberMethod(method, threadMode, eventType));
                    }
                }
            } else
if(!this.skipMethodVerificationForClasses.containsKey(var24))
{
                Log.d(EventBus.TAG, "Skipping method (not
public, static or abstract): " + var24 + "." + methodName);
            }
        }
    }
}

    if(var23.isEmpty()) {
        throw new EventBusException("Subscriber " +
subscriberClass + " has no public methods called " +
"onEvent");
    } else {
        Map var25 = methodCache;
```

```
            synchronized(methodCache) {
                methodCache.put(key, var23);
    //最后将该集合 return 出去。
                return var23;
            }
        }
    }
}
```

参考上述代码和注释我们对 **findSubscriberMethods** 方法进行一下总结，在该方法中，首先从缓存 methodCache 中查找订阅方法，如果能查找到则直接将这些订阅方法 return 出去。

如果缓存中没有，则先创建一个 ArrayList 来存储所有订阅方法，然后循环通过反射找出订阅类所有的父类，并通过类名判断排除那些系统定义的类。

然后通过 getDeclaredMethods 方法反射出所有的方法，接着 for 循环遍历每一个方法，通过反射得到方法名，并判断是否以 onEvent 开头，是否是 public 且非 static 和 abstract 方法，是否是一个参数。如果都复合，才进入封装的部分。然后在截取方法名 “onEvent”后面的部分，来确定是四个方法中的哪一个，并给其确定相对应的线程模型。然后通过 new SubscriberMethod(method, threadMode, eventType)封装成 subscriberMethod 对象，并添加到集合中，最后将集合 return 出去当然也要添加到缓存中。

然后我们接着回到 register 方法，

```
    private synchronized void register(Object subscriber,
boolean sticky, int priority) {
    List subscriberMethods =
this.subscriberMethodFinder.findSubscriberMethods(subscriber.g
etClass());
    Iterator i$ = subscriberMethods.iterator();
//遍历所有的 SubscriberMethod
    while(i$.hasNext()) {
        SubscriberMethod subscriberMethod =
(SubscriberMethod)i$.next();
```

```
    //调用 subscribe 方法。
        this.subscribe(subscriber, subscriberMethod, sticky,
priority);
    }
}
```

接下来我们就只需要看看 subscribe 方法到底是怎么实现的就可以了：

```
    private void subscribe(Object subscriber, SubscriberMethod
subscriberMethod, boolean sticky, int priority) {
    Class eventType = subscriberMethod.eventType;
    //根据 eventType 到 subscriptionsByEventType（是个 Map，
EventBus 存放方法的地方）中查找 CopyOnWriteArrayList
    CopyOnWriteArrayList subscriptions =
(CopyOnWriteArrayList)this.subscriptionsByEventType.get(eventT
ype);
    //封装 Subscription 对象
    Subscription newSubscription = new Subscription(subscriber,
subscriberMethod, priority);
    if(subscriptions == null) {
    //如果没有找到，则创建一个
        subscriptions = new CopyOnWriteArrayList();
        this.subscriptionsByEventType.put(eventType,
subscriptions);
    } else if(subscriptions.contains(newSubscription)) {
        throw new EventBusException("Subscriber " +
subscriber.getClass() + " already registered to event " +
eventType);
    }

    int size = subscriptions.size();

    for(int subscribedEvents = 0; subscribedEvents <= size;
```

```
++subscribedEvents) {
       if(subscribedEvents == size ||
newSubscription.priority >
((Subscription)subscriptions.get(subscribedEvents)).priority)
{
    //往 subscriptions 中添加 Subscription 对象。按
          subscriptions.add(subscribedEvents,
newSubscription);
          break;
       }
    }

    Object var14 =
(List)this.typesBySubscriber.get(subscriber);
    if(var14 == null) {
       var14 = new ArrayList();
       this.typesBySubscriber.put(subscriber, var14);
    }
    //根据 subscriber 存储它所有的 eventType
    ((List)var14).add(eventType);
    //判断 sticky，如果为 true 则根据 eventType 查找有没有
stickyEvent，如果有则立即发布去执行。
    if(sticky) {
       Map var11 = this.stickyEvents;
       Object stickyEvent;
       synchronized(this.stickyEvents) {
          stickyEvent = this.stickyEvents.get(eventType);
       }

       if(stickyEvent != null) {
```

```
    // postToSubscription 在讲 post 的时候详细介绍
            this.postToSubscription(newSubscription,
stickyEvent, Looper.getMainLooper() == Looper.myLooper());
        }
    }

}
```

subscribe方法实际上是把匹配的方法最终保存在subscriptionsByEventType（Map，key：eventType ； value：CopyOnWriteArrayList<Subscription> ）中；

eventType 是我们方法参数的 Class， Subscription 中则保存着 subscriber、subscriberMethod（method, threadMode, eventType）、priority，包含了执行该方法所需的一切。

Register 分析完毕，我们知道了 EventBus 如何来存储方法，那 post 又是怎么来调用方法的呢？

同样的，我们先来看看 post 的源码吧。

```
    public void post(Object event) {
    //获取 PostingThreadState 对象
    EventBus.PostingThreadState postingState =
(EventBus.PostingThreadState)this.currentPostingThreadState.ge
t();
    //获取当前线程的 EventQueue 对象
    List eventQueue = postingState.eventQueue;
    //将 event 添加到 eventQueue 中
    eventQueue.add(event);
    //判断是否分发
    if(!postingState.isPosting) {
    //若没有分发，则先判断当前线程是否是主线程
        postingState.isMainThread = Looper.getMainLooper() ==
Looper.myLooper();
        postingState.isPosting = true;
```

```
        if(postingState.canceled) {
            throw new EventBusException("Internal error. Abort
state was not reset");
        }
        try {
    //遍历出所有 event
            while(!eventQueue.isEmpty()) {
    //在这个方法里分发。
                this.postSingleEvent(eventQueue.remove(0),
postingState);
            }
        } finally {
            postingState.isPosting = false;
            postingState.isMainThread = false;
        }
    }
}
```

结合 post 方法的源码和我加上的注释，我们可以得出以下总结，通过 currentPostingThreadState.get()获取 PostingThreadState 对象，再通过该对象的 eventQueue 属性获取当前线程的 EventQueue 对象，目的是将当前的 event 添加到 eventQueue，通过 postThreadState.isPosting 判断当前是否已经在分发事件。 如果为 false 表示没有分发，先判断当前线程是否是主线程，接着遍历 eventQueue 中所有的 event，最后调用 postSingleEvent()来分发所有方法。接下来我们看看 postSingleEvent()的具体实现

```
        private void postSingleEvent(Object event,
EventBus.PostingThreadState postingState) throws Error {
    Class eventClass = event.getClass();
    boolean subscriptionFound = false;
    //判断是否支持继承
    if(this.eventInheritance) {
```

```
    //如果支持继承则找出该订阅者的所有父类和接口
       List eventTypes = this.lookupAllEventTypes(eventClass);
       int countTypes = eventTypes.size();
       for(int h = 0; h < countTypes; ++h) {
          Class clazz = (Class)eventTypes.get(h);
    //这里比较重要下面分析该方法。
     subscriptionFound |=
this.postSingleEventForEventType(event, postingState, clazz);
       }
    } else {
    //如果不支持继承则直接调用 postSingleEventForEventType。
       subscriptionFound =
this.postSingleEventForEventType(event, postingState,
eventClass);
    }
    if(!subscriptionFound) {
    //这是 postSingleEventForEventType 返回 false 的情况，post 一个
NoSubscriberEvent 并结束。
       if(this.logNoSubscriberMessages) {
          Log.d(TAG, "No subscribers registered for event " +
eventClass);
       }
       if(this.sendNoSubscriberEvent && eventClass !=
NoSubscriberEvent.class && eventClass !=
SubscriberExceptionEvent.class) {
          this.post(new NoSubscriberEvent(this, event));
       }
    }
}
```

下面我们先看 postSingleEventForEventTypes，然后再一起总结：

```
    private boolean postSingleEventForEventType(Object event,
EventBus.PostingThreadState postingState, Class<?> eventClass)
{
    CopyOnWriteArrayList subscriptions;
    synchronized(this) {
    //先到 subscriptionsByEventType 中找到 subscriptions 这里存放了所
有的订阅方法。
        subscriptions =
(CopyOnWriteArrayList)this.subscriptionsByEventType.get(eventC
lass);
    }
   // subscriptions 不为空
    if(subscriptions != null && !subscriptions.isEmpty()) {
        Iterator i$ = subscriptions.iterator();

        while(i$.hasNext()) {
    //遍历出所有的 subscription
            Subscription subscription = (Subscription)i$.next();
            postingState.event = event;
            postingState.subscription = subscription;
            boolean aborted = false;
            try {
    //在这里面做订阅方法的调用
                this.postToSubscription(subscription, event,
postingState.isMainThread);
                aborted = postingState.canceled;
            } finally {
                postingState.event = null;
                postingState.subscription = null;
                postingState.canceled = false;
            }
```

```
            if(aborted) {
                break;
            }
        }
        return true;
    } else {
        return false;
    }
}
```

在postSingleEvent方法中首先判断是否支持继承，支持的话则调用lookupAllEventTypes方法找出所有父类和接口，全部添加到eventTypes集合当中。然后调用postSingleEventForEventType方法 根据subscriptionByEventType集合获取所有订阅者。（不支持继承的话效率更高，就可以直接调用postSingleEventForEventType方法。）

接下来，遍历所有订阅者 ，并判断subscription是否为空。 若为空：post一个NosubscriberMethod，结束。

不为空：遍历subscriptions,后调用postTosubscription()方法。那咱们最后看看postTosubscription方法的具体实现：

```
    private void postToSubscription(Subscription
subscription, Object event, boolean isMainThread)
{ switch(EventBus.SyntheticClass_1.$SwitchMap$de$greenrob
ot$event$ThreadMode[subscription.subscriberMethod.threadM
ode.ordinal()]) {
    case 1:
    //直接调用 invokeSubscriber 来反射调用订阅方法。
        this.invokeSubscriber(subscription, event);
        break;
    //首先判断如果是 UI 线程则直接调用，不是的话则将当前方法加入队列
然后通过 handler 发送一个消息然后再 handleMessage 方法中执行
    case 2:
        if(isMainThread) {
```

```
            this.invokeSubscriber(subscription, event);
        } else {
            this.mainThreadPoster.enqueue(subscription,
event);
        }
        break;
    //如果不是 UI 线程则直接反射调用，是的话则将任务加入后台队列，最
终由 EventBus 的一个线程池去调用。
    case 3:
        if(isMainThread) {
            this.backgroundPoster.enqueue(subscription,
event);
        } else {
            this.invokeSubscriber(subscription, event);
        }
        break;
    //将任务加入后台队列，最终由 EventBus 的一个线程池去调用。
    case 4:
        this.asyncPoster.enqueue(subscription, event);
        break;
    default:
        throw new IllegalStateException("Unknown thread
mode: " + subscription.subscriberMethod.threadMode);
    }

}
```

其实就是判断subscription的线程模型，再通过反射调用订阅者的订阅方法，就结束了。

case PostThread: 直接反射调用；也就是说在当前的线程直接调用该方法；

case MainThread:首先去判断当前如果是UI线程，则直接调用；否则： mainThreadPoster.enqueue(subscription, event);把当前的方法加入到队

列，然后直接通过handler去发送一个消息，在handler的handleMessage中，去执行我们的方法。说白了就是通过Handler去发送消息，然后执行的。

case BackgroundThread:如果当前非UI线程，则直接调用；如果是UI线程，则将任务加入到后台的一个队列，最终由Eventbus中的一个线程池去调用executorService = Executors.newCachedThreadPool();。

case Async:将任务加入到后台的一个队列，最终由 Eventbus 中的一个线程池去调用；线程池与 BackgroundThread 用的是同一个。

至此，EventBus 存储和调用方法我们就都摸清楚了。

14、 Java 和 Js 的互调。

实现Java和Js的交互十分便捷，通常只需要一下几步：

1).WebView开启JavaScript脚本执行

2).WebView设置供JavaScript调用的交互接口。

3).客户端和网页端编写调用对方的代码。

Java调Js：

webView调用js的基本格式为：

webView.loadUrl(“javascript:methodName(parameterValues)”)。

调用js无参无返回值函数：

```
    String call = "javascript:sayHello()";
    webView.loadUrl(call);
```

调用js有参无返回值函数：

```
    String call = "javascript:alertMessage(\"" + "content"
+ "\")";
    webView.loadUrl(call);
```

调用js有参有返回值函数：

Android在4.4之前并没有提供直接调用js函数并获取值的方法，所以在此之前，常用的思路是 java调用js方法，js方法执行完毕，再次调用java代码将值返回。

1).Java调用js

```
String call = "javascript:sumToJava(1,2)";
webView.loadUrl(call);
```

2).js函数处理，并将结果通过Java方法返回。

```
    function sumToJava(number1, number2){
    window.control.onSumResult(number1 + number2)
}
```

3).Java在回调方法中获取js函数的返回值

```
    @JavascriptInterface
public void onSumResult(int result) {
    Log.i(LOGTAG, "onSumResult result=" + result);
}
```

但是在4.4之后，使用evaluateJavascript即可。例子如下：

js的函数：

```
function getGreetings() {
    return 1;
}
```

java代码时用evaluateJavascript方法调用：

```
    private void testEvaluateJavascript(WebView webView) {
    webView.evaluateJavascript("getGreetings()", new
ValueCallback<String>() {
        @Override
        public void onReceiveValue(String value) {
            Log.i(LOGTAG, "onReceiveValue value=" + value);
        }});
}
```

注意事项：

1).上面限定了结果返回结果为String，对于简单的类型会尝试转换成字符串返回，对于复杂的数据类型，建议以字符串形式的json返回。

2).evaluateJavascript方法必须在UI线程（主线程）调用，因此onReceiveValue也执行在主线程。

Js调用Java:

调用格式为window.jsInterfaceName.methodName(parameterValues) 此例中我们使用的是control作为注入接口名称。

```
    function toastMessage(message) {
    window.control.toastMessage(message)
}
```

项目篇

1、 项目中遇到的问题。

这个问题是同学们遇到过的最恐怖的问题，因为在大家的印象中貌似根本不知道项目中会遇到什么问题，顶多也就是一大堆的空指针。那维哥就教大家从以下这些方面去思考：

1).登录 cookie 的存取问题。

有的公司是使用 cookie 来登录和获取数据等相关操作。此时，我们在切换账户时应该将 cookie 清除掉，如果继续使用原账号的 cookie 并结合新账号和密码来登录的话，服务器会接收但是返回数据有误。原因是：

我们第一次请求服务器的时候，服务器会把 cookie 放在 Head 信息中返回给我们，字段为“Set-Cookie”，我们在接收后对它进行存储，之后的请求服务器还会把 cookie 放在 Head 信息中返回给我们，字段为“Cookie”，也就是说 Cookie 只需要设置一次，之后的网络请求都不需要设置了，如果我们切换账户不清楚 Cookie 的话，服务器返回的字段为“Cookie”，表示已经设置过了，但是是之前用户的，所以服务器返回的数据有误。

2).在 APP 退出登录后，还会收到极光推送针对该用户的相关推送。

一般我们退出登录时，都会清除极光的 tag 和别名，为的就是在退出登录之后，不会收到该用户的相关推送。但是在实战中，即使退出登录了，有时还是会收到与该用户相关的推送。清除极光的 tag 和别名用的是：

`JPushInterface.setAliasAndTags(SettingActivity.this, "", tagList, new TagAliasCallback() {}`里面第一个参数为当前对象，第二个为别名（我们在这置为空），第三个为 tag，（我们在这置为空），第四个是回调，如果回调成功我们才做一些清理工作，如清除 Activity 等

```
Set<String> tagList = new HashSet<>();
JPushInterface.setAliasAndTags(SettingActivity.this,  "",
tagList, new TagAliasCallback() {
 @Override
 public  void  gotResult(int  responseCode,  String  arg1,
Set<String> set) {
    if (responseCode == 0) {
     customProgressDialog.dismiss();

JPushInterface.clearAllNotifications(SettingActivity.this
);//清掉极光通知栏信息
     SPUtil.clear();//退出清除所有缓存
       finishActivities();  //清除所有 Activity
     } else {
       customProgressDialog.dismiss();
 UIUtils.showToastSafe("当前网络信号差，请重试");
   }
  }});
```

3).时间戳转换问题。

在开发中，我们经常要将服务器返回的时间戳转换成年月日时分秒的格式，但是在转换的时间会发现时间和服务器的时间好像对应不起来，原因是因为我们没有设置时区，sdf.setTimeZone(TimeZone.getTimeZone("GMT+08:00"));这句代码很重要.

4).类型转换时的安全问题。

基本类型数据转换时经常会遇到空指针的情况，在这个时候我们会自定义一个安全的数据转换方法，如果为空时会给默认值。

5).应用程序退出到后台再次进入时，会出现一块块的空白。（具体原因在 Fragment 那一节有讲到）

6).滑动冲突问题。

7).嵌套问题,包括 ListView 嵌套 GridView、ScroolView 嵌套 ListView 等等。

2、 项目的下载量、日活、留存率的问题。

这里主要对这些概念做个介绍。

下载量：所有应用市场上的下载总量。

日活：每天的活跃用户量，指该统计日登录或使用了我们 APP 的用户量（除去重复登录）。

留存率：用户在某段时间内开始使用应用，经过一段时间后，仍然继续使用该应用的用户，被认作是留存用户，这部分用户占当时新增用户的比例即是留存率，会按照每隔 1 单位时间（例日、周、月）来进行统计。

3、 敏捷开发中有哪些必须了解的会议？

1).需求评估会，由项目经理组织召开，全体人员参与，由产品经理讲解每一个需求。目的是确定此次迭代必须要完成的需求，将一些不是必须在该版本完成的需求推迟到后面的版本或者或者直接干掉。然后再由项目经理在会后跟开发人员、测试人员来分配需求及评估出需求的工时和工期。

2).测试用例评审会，测试人员、开发人员、产品经理会一起在会上确定需要测试的功能点及通过测试的条件。

3).冲刺会,相当于动员大会,在项目开动当天或者前一天由项目经理组织召开,时间很短。主要是项目经理告诉大家本次迭代正式开始，计划什么时候出版本,给大家打打鸡血之类的。

4).每日例会，也叫站会，每天早上上班的时候召开。由项目经理组织，团队所有人员参与，主要目的是总结昨天的工作完成度、计划今天的工作内容。

5).总结会，在每次版本迭代完成之后由项目经理组织召开，主要是总结此次版本迭代过程中做的好的和不好的地方，好的继续保持，不好的集思广益寻找解决办法。

4、 项目经理的职责。

1).项目经理不需要关心太多的业务逻辑，只需要关心项目进度即可。

2).在需求评估会后，和开发人员、测试人员一起确定 Task 的分配、工时和工期，并制定项目推进时序图。
3).主持每天的例会，并发送会议纪要。
4).推动 bug 修复工作。
5).每天在项目管理工具（维哥上家公司用的是 redmine）上给开发、测试人员分配工作任务。
6).每次迭代结束后组织召开总结会。

5、 产品经理的职责。

1).提需求、优化产品的用户体验。
2).参与每天的例会，这样才能知道开发人员在哪些需求上遇到了逻辑问题，从而及时做出调整。
3).测试团队提的 bug，经过开发人员分析后，发现是产品需求的问题，会将 bug 转给产品经理重新定义业务逻辑。
4).验收需求，在开发结束、测试工作接近尾声的时候，产品经理会验证实际开发出来的功能是否与他的需求一致。

6、 项目团队人员构成。

对于这个问题，不同的团队有不同的人员构成，这里就以维哥曾经待过的一个开发团队为例：1 个产品经理、1 个项目经理、2 个 Android 开发人员、2 个 IOS 开发人员、2 个后台开发人员（1 个是 CTO）、1 个测试、1 个美工。

7、 实际开发流程。

这里以没个月进行一次版本迭代的敏捷开发流程为例子讲解。

需求准备：

1).修复上次迭代来不及处理的 bug。
2).做一些代码上的重构工作，永远以产品需求为最高优先级的 Task，在完成产品需求后，再利用空余时间来做代码优化、项目重构。
3).产品经理提出新需求，并在需求评估会上确定新版本的需求及必须解决的 bug。

排期：由产品经理和开发人员、测试人员一起确定 Task 的分配、工时和工期，并制定项目推进时序图。

开发：每天早上开例会（例会的主要内容上面有说到），项目经理在项目管理工具上分配工作任务，开发人员完成后提交给测试、测试通过后再返给开发人员，开发人员将任务标注为“已完成、测试通过”再提交给项目经理。如果测试不通过，测试人员会将任务标注为“测试不通过”然后返回给开发人员。

测试：测试人员的工作如下

1).编写测试用例，并对着测试用例上的功能点来进行测试。

2).每天测试开发人员完成的功能。

3).Monkey 测试，项目每天下班前都要跑 Monkey，然后会有专人分析 Crash 日志。

4).版本发布前一周主要是用来做测试。在这五天里面，前三天是集中测试：集中所有测试人力对所有新功能进行一次测试。开发人员必须保证 bug 日清。

后两天主要是全功能回归测试，这是最后一轮测试，这期间发现任何 bug 我们都要评估，如果不是很严重，本期迭代就不修复了。

版本发布：

现在一般要用到多渠道打包技术，一次性打好多个渠道包，再分别发布到各个应用市场。

8、 面试中怎么介绍项目。

这里写一套通用的项目讲解套路，然后再由大家各自发挥自己所长。

1).一两句话简单介绍 APP 面向的群体和功能。如：这个项目是一个商城类的项目，主要是做婴幼儿产品的零售。

2).介绍自己在项目开发过程中所负责的模块。如：我在项目开发中负责框架搭建、网络协议二次封装、基类的抽取、一些必要的工具类的封装、首页模块、个人中心模块。

3).框架搭建怎么讲：项目整体采用 MVP 开发模式，整体 UI 框架采用 TabHost 来进行 Fragment 的切换。侧滑菜单用的是 DrawerLayout 实现的，左侧菜单是用 NavigationView 完成的。

项目包含 XXX 哪些包，每个包的放的是与哪些业务逻辑相关的类。

采用抽象工厂模式来设计 FragmentFactory 类，采用双重检查锁来设计单例的用户类 User，另外如果觉得哪里还能用到设计模式都可以讲出来。

4).网络请求二次封装怎么讲：为什么要对网络请求进行二次封装怎样实现网络协议的二次封装？

原因：(1)进行二次封装能减少代码量,如果不进行二次封装的话在需要进行网络请求的地方都需要写很多重复的请求和解析数据的代码。(2)进行二次封装便于在后续版本中进行开源框架的替换。如果不封装的话，假如我现在是用的 volley，一年以后 volley 出问题了需要换乘 OkHttp，那就得去所有写了网络请求的地方去修改，不符合面向对象设计的“开放-封闭原则”。

怎么封装：以 OkHttp 为例，定义一个 HttpUtils 类，在该类中定义 get、post 等请求方法，然后在这些方法中封装 OkHttp 的 get 和 post 请求的代码和 Json 解析的代码，返回解析出来的 Bean 对象即可。

5).基类抽取怎么讲：讲到在基类中封装了什么功能，暴露什么方法让子类实现。具体参考谷歌市场。

6).一些必要工具类：如检查网络状态、安全地进行数据类型转换、dp 和 px 的互转、网络请求封装的工具类、时间戳转换工具类等等。

7).针对自己具体所负责的模块：如首页、个人中心等等。可以这样说：首页采用 RecyclerView + CardView 来展示数据，如果有涉及到数据库的地方可以说到一下方面：数据库是怎么设计的——有哪些字段、数据库在版本更新时是怎么来进行升级的、如果数据库比较复杂还可以讲讲数据库的优化。

8).举两个例子来说说网络请求，比如说在请求中携带什么参数、服务器返回的是什么数据？（主要是为了增加可信度）

9).个人中心修改头像、个人资料是 post 上传文件、上传 Json 数据，可以针对性地讲一下 post 请求。

10).项目中如果有较炫的动画或者自定义 View，可以详细讲一下该动画或自定义 View 是怎么实现的。

11).集成了哪些第三方开源框架、第三方 SDK。（主要目的是让面试官了解你做过什么东西），针对某些第三方 SDK 的集成，可以讲一下大概的集成步骤。

12).比如你说你图片加载用的是 glide、网络请求用的是 OkHttp，那讲一下你为什么选用它们，它们有什么优势。

13).最后就是你们自己根据自己的项目发挥，讲一下你项目中较难实现的功能。

讲项目的时候千万别总是停留在界面上讲，一定要深入代码中，让面试官相信这个项目确实是你做的。

人事篇

1、 为什么从上家公司离职？

①最重要的是 :应聘者要使找招聘单位相信, 应聘者在过往的单位的“离职原因”在此家招聘单位里不存在。②避免把“离职原因”说得太详细、太具体。③不能掺杂主观的负面感受，如“太辛苦”、“人际关系复杂”、“管理太混乱”、“公司不重视人才”、“公司排斥我们某某的员工”等。④但也不能躲闪、回避，如“想换换环境”、“个人原因”等。⑤不能涉及自己负面的人格特征，如不诚实、懒惰、缺乏责任感、不随和等。⑥尽量使解释的理由为应聘者个人形象添彩。⑦相关例子：如“我离职是因为这家公司倒闭；我在公司工作了三年多，有较深的感情；从去年始，由于市场形势突变，公司的局面急转直下；到眼下这一步我觉得很遗憾，但还要面对显示，重新寻找能发挥我能力的舞台。”同一个面试问题并非只有一个答案，而同一个答案并不是在任何面试场合都有效，关键在应聘者掌握了规律后, 对面试的具体情况进行把握, 有意识地揣摩面试官提出问题的心理背景, 然后投其所好。

分析：除非是薪资太低，或者是最初的工作，否则不要用薪资作为理由。“求发展”也被考官听得太多，离职理由要根据每个人的真实离职理由来设计，但是在回答时一定要表现得真诚。实在想不出来的时候，家在外地可以说是因为家中有事，须请假几个月，公司又不可能准假，所以辞职，这个答案一般面试官还能接受。

2、 怎么体现自己学习能力强？

研究源码、framework 层、前端和后台技术、RxAndroid、Kotlin 等，主要表现出自己在空余时间研究较难、较深、较新的技术或者其它语言。

3、 你的职业规划。

这是每一个应聘者都不希望被问到的问题，但是几乎每个人都会被问到，我们作为技术人员，最好的说法就是：自己比较喜欢专研技术，想一直在技术这条路上走下去，学习前端、后台的技术成为一名全栈式开发工程师或者往 framework 层研究等等。

4、 能不能接受加班？

如果是工作需要我会义不容辞加班，我现在单身，没有任何家庭负担，可以全身心的投入工作。但同时，我也会提高工作效率，减少不必要的加班。

5、 目前你最明显的缺点是什么？

这个问题企业问的概率很大，通常不希望听到直接回答的缺点是什么等，如果求职者说自己小心眼、爱忌妒人、非常懒、脾气大、工作效率低，企业肯定不会录用你。绝对不要自作聪明地回答“我最大的缺点是过于追求完美”，有的人以为这样回答会显得自己比较出色，但事实上，他已经岌岌可危了。业喜欢求职者从自己的优点说起，中间加一些小缺点，最后再把问题转回到优点上，突出优点的部分，企业喜欢聪明的求职者。

6、 对于我们公司还有什么需要了解的吗？

企业的这个问题看上去可有可无，其实很关键，企业不喜欢说“没问题”的人，因为其很注重员工的个性和创新能力。企业不喜欢求职者问个人福利之类的问题，如果有人这样问：贵公司对新入公司的员工有没有什么培训项目，我可以参加吗？或者说贵公司的晋升机制是什么样的？企业将很欢迎，因为体现出你对学习的热情和对公司的忠诚度以及你的上进心。

7、 和上级意见不一致的时候，你会怎么办？

①一般可以这样回答“我会给上级以必要的解释和提醒，在这种情况下，我会服从上级的意见。”②如果面试你的是总经理，而你所应聘的职位另有一位经理，且这位经理当时不在场，可以这样回答：“对于非原则性问题，我会服从上级的意见，对于涉及公司利益的重大问题，我希望能向更高层领导反映。”

分析：这个问题的标准答案是思路①，如果用②的回答，必死无疑。你没有摸清楚改公司的内部情况，先想打小报告，这样的人没有人敢要。

8、 为什么选择我们公司？

去面试之前最好是了解一下这家公司，回答这个问题的时候，最好是说公司的优势，哪些方面非常适合自己，而自己所擅长的方面也能为公司提供很大的帮助。

9、 期望薪资是多少？

如果你对薪酬的要求太低，那显然贬低自己的能力；如果你对薪酬的要求太高，那又会显得你分量过重，公司受用不起。一些雇主通常都事先对求聘的职位定下开支预算，因而他们第一次提出的价钱往往是他们所能给予的最高价钱，他们问

你只不过想证实一下这笔钱是否足以引起你对该工作的兴趣。如果你自己必须说出具体数目，请不要说一个宽泛的范围，那样你将只能得到最低限度的数字。最好给出一个具体的数字，这样表明你已经对当今的人才市场作了调查，知道像自己这样学历的雇员有什么样的价值，给出自己能接受的价格，**不要期望着对方会跟你还价，对方只会对比众多面试者，从中选取性价比最高的。如果他跟你还价了，那就说明只要你答应他那个薪资就一定要你。**

回答样本一：我对工资没有硬性要求，我相信贵公司在处理我的问题上会友善合理。我注重的是找对工作机会，所以只要条件公平，我则不会计较太多。

回答样本二：我受过系统的软件编程的训练，不需要进行大量的培训，而且我本人也对编程特别感兴趣。因此，我希望公司能根据我的情况和市场标准的水平，给我合理的薪水。

10、 你这项目很简单嘛，我们这边的程序员一年能做出十几个。

这个问题有时候会被那种很苛刻的面试官问到，这个问题很好回答，老板和产品经理要求我做成这个样子，项目简单与否不是程序员能决定的。如果有很多项目给我做的话，像这类的项目我一年也能做好几个，其实做了几年应用层开发的人都知道，应用层开发没有多少高深的东西，所以我现在也在往 framework 层学习。

11、 自我介绍。

一般人回答这个问题过于平常，只说姓名、年龄、爱好、工作经验，这些在简历上都有。其实，企业最希望知道的是求职者能否胜任工作，包括：最强的技能、最深入研究的知识领域、个性中最积极的部分、做过的最成功的事，主要的成就等，这些都可以和学习无关，也可以和学习有关，但要突出积极的个性和做事的能力，说得合情合理企业才会相信。企业很重视一个人的礼貌，求职者要尊重考官，在回答每个问题之后都说一句“谢谢”，企业喜欢有礼貌的求职者。