root@zs600b:/ # top -m 10

-t表示查看线程

CTRL+C结束统计，

# 稳定性优化

Android应用的稳定性定义很宽泛，影响稳定性的原因也很多，比如内存使用不合理、代码异常场景考虑不周全、代码逻辑不合理等，都会对应用的稳定性造成影响。本章讨论的是稳定性中的两个常见场景：Crash和ANR，这两种错误将使程序无法使用。因为引发这两种情况的原因无法系统归类，因此本章不会介绍如何修改某个Crash或者ANR，而是介绍一系列的方法和工具，探索如何发现问题和规避问题，把风险降低。

本章从如何提高代码的质量以减少这些情况的发生，到通过对Crash和ANR的监控，以对应用的稳定性有一个全局的掌握，再到采取有效的上报机制逐个解决。

## 提高代码质量

代码质量是一个很大的话题，不同的编程语言都有优秀的书籍介绍如何编写高质量的代码，因此这里不介绍如何编写。然而实际的应用开发团队都会面临以下两个问题：

1）程序员的水平差异：如果程序员基础扎实并且开发经验丰富，毫无疑问，可以写出更高质量的代码，然而在开发团队中，难免会有一些经验较少、代码编写质量并不好的程序员存在。

2）开发人员变动：人员变动是不可避免的，一个模块的开发人员变更后，新的开发者可能对以前的设计理解不全面，从而产生偏差。而一般前期也不会出现问题，并且满足完成需求，但随着时间的推移会发现开发越来越吃力，不停地打补丁修复问题。

因此，要解决这两个问题可以从两个方向入手：代码编写阶段的优化和完成后借助工具检查发现问题。

本节介绍在实际项目开发中，非常重要的提高代码质量的两种方法：代码审查与重构。

### 代码审查

代码审查也就是我们常说的Code Review，是指程序员在完成自己的代码后，由其他人来复查他写的代码，目的是发现应用开发初期未发现的错误并修改，同时提升软件质量及开发者的技术。

在开发阶段增加Code Review环节，可以大大提高代码的质量，不但能发现缺陷，也能提高代码的可读性和可维护性。特别是较大的开发团队，开发人员的变动会很频繁，代码审查可以督促开发者思考如何提高编码质量。做代码审查的人也是受益者，通过读别人的代码，可以更深刻地理解什么是代码的可读性。代码审查还可以促进开发团队人员之间的技术交流。

虽然代码审查有如此大的好处，但使用不当带来的坏处也不能轻视。因为代码审查的代价不小，需要从时间、人力、代码质量方面综合考虑，形成有效的代码审查机制需要解决以下三个问题：

·何时代码审查？

确定什么时候需要代码审查。

·谁来代码审查？

确定代码审查的对象是谁。

·代码审查的内容是什么？

有目的地代码审查。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　代码审查不仅仅是找Bug，更多是使代码更易读，有更高的维护性，同时可以达到知识共享。

#### 何时代码审查

何时代码审查分两个方面：一是这个模块是否需要审查，明确代码审查的必要性，二是在开发阶段的哪个时间点代码审查，确定代码审查合理的时间点。

在实际的应用开发过程中，一个主要的矛盾是需求开发量大，但开发人力有限，而代码审查增加了不少的时间。权衡上线时间压力与稳定性需求，在确定模块是否要代码审查和何时代码审查时可以从以下几个方面考虑。

（1）底层公共模块

比如网络模块、图片管理模块这类底层公共模块，因为整个应用都使用这些模块，并且因为要提供给上层模块的开发者使用，如果有问题影响面大。这类模块因为不涉及具体的业务，不可避免地会对设计考虑不全，通过代码审查可以及早发现模块的缺陷并修复，同时让接口做得更友好。因此，这类模块不仅在开发完成后需要代码审查，以后的维护、修改都需要代码审查。

2）重大特性业务

由于重大特性业务的复杂性，更容易出现问题，并且大的特性维护的时间也会比较长，通过代码审查及早发现问题，达到更好的可维护性。重大特性业务尽量不要出现重大问题，否则影响应用的用户口碑。

（3）与其他模块有耦合

设计一个新的业务模块时，因为产品特性的原因，或多或少会有些耦合，通过代码审查发现两个（或者多个）模块之间是否对业务理解一致，避免业务或者体验有矛盾。

（4）新手

新来的开发者，特别是刚毕业的学生，也许他们做的功能比较简单，并不是前面两种模块中的一种，但通过代码审查可以发现一些代码规范方面的不足和一些明显的缺陷，使他们更快地向有经验的开发者学习。

（5）应用即将发布前

全量测试已经完成，在准备上线的情况下，有些紧急Bug的修复需要代码审查，主要是发现真正的问题以修复，并且没有带来新问题，避免上线后出现严重问题再重新发版本。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　代码审查是一个耗时耗力的过程，过度的代码审查不但不能达到预期的效果，并且容易引起一些经验丰富的程序员反感，因此要避免没有必要的代码审查，影响团队开发的效率。

#### 2.谁来代码审查

如何选择合适的人去做代码审查非常重要，由于应用开发需要敏捷思维，所以也不能固定一种审查方式，而是根据不同的业务/模块选择审查者和审查的方式，代码审查分为三种方式：团队审查、模块负责人审查、结对审查。可以根据不同的情况选择。

（1）团队审查

俗话说“三个臭皮匠，顶个诸葛亮”。即使是经验欠缺的团队，如果能群策群力，也能交付出高质量的应用。这里先不论代码审查对团队的成长的帮助，单单是每一次修改都让其他人审查的行为，就能提高初级程序员的信心、对产品的参与度和责任心。这种审查方式适用于底层公用的模块，通过审查让大家都熟悉底层的实现原理和接口调用方式，在以后使用该模块时可以很快地理解如何使用，并且根据上层业务特点查看模块实现是否合理，对上层业务是否有影响。

但这种审查方式，要整个项目组全体成员去审查，毫无疑问付出的代价太大，因此只适用于底层通用的大模块，如网络引擎、图片管理模块等。而其他的模块可以考虑使用后面两种审查方式。

（2）模块负责人审查

一个应用的开发，可以根据功能模块划分成多个开发小组，每个开发小组的负责人无疑是开发经验最丰富的人员，也是对业务最了解的人员，由他负责组织和指导其他成员是非常合适的。而这位负责人，有责任有效地完成代码审查工作，一般重大业务模块可以使用这个方案，因为重大业务比较复杂，需要一个经验丰富和业务熟悉的人去代码审查，避免风险，同时因为业务模块和底层公用模块不同，所以没有必要团队集体审查。

（3）结对审查

结对代码审查和结对编程（pair-programming）一样，两个开发者，一个人是Driver角色，另一个人是Observer。Driver编写完代码后，由Observer查看Driver的代码，两人一起讨论代码的实现方案，而这两个角色是可以互换的。结对代码审查除了提高代码质量外，还因为一个模块有两个人熟悉，在碰到紧急事情，原开发者没有时间跟进时，原来的Observer可以很快代替原Driver来工作，减少熟悉代码的过程。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　结对编程也有一些弊端，对于不同习惯的程序员，有时候会对问题坚持自己的看法，容易产生争吵，一定层面上有些内耗，因此是否使用结对编程需要根据具体问题考虑，同时需要很好地处理过程中的矛盾。

#### 代码审查内容

在代码审查时，需要确定好审查的基本方向，否则漫无目的地查看代码，没有针对性地分析代码作用不大，这样也容易使审查流于形式。审查的流程为：先审查设计实现思路，然后审查设计模式，接着审查成形的骨干代码，最后审查完成的代码。一般来说审查代码时可以从以下几个方面查看代码。

（1）实现思路与设计思想

·单一职责原则：一个模块有且只能有一个职责。如果一个模块或者一个类提供了不同类型的功能，或者一个功能需要几个模块共同完成，这就有可能在抽象层上设计不合理。

·开闭原则，在面向对象的语言中，对象对可扩展开放、对修改关闭。所以需要考虑添加/扩展另外的内容时是否会带来新的问题。

1. 代码设计

代码复用：根据“三振法”，如果代码被复制一次，通常没什么问题。但如果复制超过三次，就应该通过提取公共的部分来重构它。

·更合理的代码：是否有更好的实现。

·潜在的缺陷：异常情况考虑是否全面，错误的传参是否会引起的其他错误？循环是否以我们期望的方式终止？

严重缺陷有两种类型：业务实现是否有缺陷或者有矛盾的地方。代码缺陷，如溢出、空指针等，这些审查非常重要，但简单而且机械，比较耗时，建议通过代码静态扫描工具发现，在5.12节介绍常用的静态代码扫描工具。

·错误处理：要考虑错误确定后是否被优雅地修改，是否会导致其他错误，修改是否有用。

·效率：如果代码中包含算法，这种算法是否高效？

（3）设计逻辑

设计逻辑和设计模式是复杂并且有深度的审查，需要有一定理论基础和开发经验的人才能有效发现设计逻辑方面的错误。

（4）代码风格

每个人都有一些不同的编码习惯，不用规范得太严格，只需要关注以下几点：

·命名：在计算机科学中，命名是一个难题。一个不准确的函数命名会误导使用者的预期。数据结构尽量写得详细些，在别人查看或自己重新查看代码时，那些见名知意的命名可以让人更容易理解它们的定义。

·方法长度：对于一个方法的长度，一般的经验值是小于100行，如果一个函数在100行以上，最好是分成多个方法。但在Android系统上存在不同的情况，因为在Android应用中有个最大方法数的问题，所以一个方法的长度多少倒不是很重要，需要根据应用自身的特点去考虑。

·函数参数个数：建议越少越好，如果需要参数很多，建议封装成一个新的数据结构。

（5）需求理解是否到位

这需要对业务特别熟悉，特别是复杂的业务需求，需要审查实现后的结果是否和预期一致。

#### 小结

一直强调代码审查是一个比较重的环节，合理地代码审查，不要流于形式，也不要成为开发者的负担。

然而再好的代码审查也不能发现所有的问题，特别是一些代码上的缺陷，再仔细认真的人也会有疏漏，所以需要借助一些代码审查工具，代替部分人工审查，提高开发效率。

### 代码静态扫描工具

静态代码分析是指通过分析或检查源代码的语法、结构、过程、接口等来检查程序的正确性，找出代码隐藏的错误和缺陷，比如参数类型不匹配、错误的递归、线程安全、非法计算，以及可能出现的空指针引用等。

在Android开发过程中，开发人员不可避免地需要花费大量的时间去发现并修改代码缺陷，而静态代码分析工具可以作为开发辅助工具，在构建过程中帮助开发人员快速发现问题，然后快速修改，这样可以在一定程度上提高软件的稳定性，并且节省开发和测试成本。目前在Android应用开发中，大部分是Java代码，Java静态代码分析工具种类繁多，且各有优缺点，本节介绍4种常用的Java代码分析工具：Android Lint、Findbugs、Checkstyle以及PMD，如表5-1所示。



表5-1　4种静态代码扫描工具对比

（1）Checkstyle

Checkstyle是SourceForge的开源项目，通过对代码编码格式、命名约定、Javadoc、类设计等方面进行代码规范和风格的检查，从而有效约束开发人员更好地遵循代码编写规范。

（2）FindBugs

FindBugs是由马里兰大学提供的一款开源Java静态代码分析工具。FindBugs通过检查类文件或JAR文件，将字节码与一组缺陷模式进行对比，从而发现代码缺陷，完成静态代码分析。FindBugs既提供可视化UI界面，也可以作为Eclipse插件使用。文本主要将FindBugs作为Eclipse插件。在安装成功后会在Eclipse中增加FindBugs perspective，用户可以对指定Java类或JAR文件运行FindBugs，此时FindBugs会遍历指定文件，进行静态代码分析，并将代码分析结果显示在FindBugs perspective的bugs explorer中。

1. PMD

PMD是由DARPA在SourceForge上发布的开源Java代码静态分析工具。PMD通过其内置的编码规则对Java代码进行静态检查，主要检查潜在的bug、未使用的代码、重复的代码、循环体创建新对象等问题。PMD提供了和多种Java IDE的集成，如Android Studio、Eclipse，IDEA、NetBean等。

（4）Android Lint

在前面的章节中介绍过Android Lint部分功能的使用，这里不再过多介绍。Android Lint是一个非常强大的Android应用程序代码检查利器，除了代码缺陷外，还能检查代码布局的合理性等。

由于这4种工具内置的编程规范各有不同，因此它们发现不同种类的代码问题的发现能力也有所不同。其中Checkstyle更加偏重于检查代码编写格式，而FindBugs、PMD更偏重于发现代码缺陷，Android Lint则非常明显地除了检查代码外，也包括了针对Android平台特点的更丰富的检查。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　利用Gradle，在Android平台使用代码质量检查工具是非常容易。比在自己的计算机上使用质量工具局部检查更全面、效率更高，Gradle可以用于自动构建如Jenkins/Hudson这样的平台，让你自动进行质量检查，并自动建立过程。

## Crash监控

Crash（应用崩溃）是由于代码异常而导致App非正常退出，导致应用程序无法继续使用，所有工作都停止的现象。发生Crash后需要重新启动应用（有些情况会自动重启），而且不管应用在开发阶段做得多么优秀，也无法避免Crash发生，特别是在Android系统中，系统碎片化严重、各ROM之间的差异，使Android在稳定性方面需要付出更多的代价。

在应用中发生Crash具有以下两个特点。

·非必现：一般来说Crash不是必现的，可能是一个很少使用的场景触发，在开发和测试阶段都没有考虑到，并且是有概率地触发。

·原因多，无法系统性解决：导致Crash的原因有很多，有代码逻辑缺陷、系统兼容问题、硬件兼容问题，而在应用开发时较少关注到，特别是硬件和ROM的兼容性问题，需要特定的机型/ROM和特定的场景才触发。

因此，要降低Crash发生的概率，需要Crash监控和发生Crash时的堆栈信息，开发者拿到这些信息后分析导致Crash的原因并修复。

在Android应用中发生的Crash有两种类型，Java层的Crash和Native层Crash。这两种Crash的监控和获取堆栈信息有所不同。接下来介绍发生Crash时，如何监控和获取日志。

### Java层Crash监控

在Android中，Java虚拟机为每个进程都设置了一个默认UncaughtExceptionHandler，用于处理本进程中未被try catch的Exception。因此只要实现UncaughtExceptionHandler接口，并在进程启动时调用Thread.setDefaultUncaughtExceptionHandler（...）传入自定义的Uncaught ExceptionHandler，当出现没被catch的异常时，就会回调uncaughtException（Thread thread，Throwable ex）方法，可以在这里记录crash日志，并上报给服务器，也可以执行一些个性化的异常处理操作。

下面是一个实现Thread.UncaughtExceptionHandler接口的类：CrashHandler，代码如代码清单5-1所示。

|  |
| --- |
| package com.android.androidtech;  import android.content.Context;  import android.content.pm.PackageInfo;  import android.content.pm.PackageManager;  import android.os.Build;  import android.os.Environment;  import com.android.androidtech.utils.GLog;  import java.io.BufferedWriter;  import java.io.File;  import java.io.FileWriter;  import java.io.IOException;  import java.io.PrintWriter;  import java.text.SimpleDateFormat;  import java.util.Date;  import android.os.Process;  public class CrashHandler implements Thread.UncaughtExceptionHandler {  private static final String TAG = "CrashHandler";  private static final String CRASH\_FILE\_NAME = "crash";  private static final String CRASH\_FILE\_PATH = Environment.getExternalStorageDirectory().getPath() + "/log/";  //log文件的后缀名  private static final String CRASH\_FILE\_NAME\_SUFFIX = ".txt";  //系统默认的异常处理（默认情况下，系统会终止当前的异常程序）  private Thread.UncaughtExceptionHandler mDefaultCrashHandler;  private Context mContext;  //这里主要完成初始化工作  public void init(Context context) {  //获取系统默认的异常处理器  mDefaultCrashHandler = Thread.getDefaultUncaughtExceptionHandler();  //将当前实例设为系统默认的异常处理器  Thread.setDefaultUncaughtExceptionHandler(this);  //获取Context，方便内部使用  mContext = context;  //可以在初始化后在异步线程中上报上次保存的Crash信息  }  /\*\*  \* 这个是最关键的方法，当程序中有未被捕获的异常，系统将会自动调用#uncaughtException方法  \* thread为出现未捕获异常的线程，ex为未捕获的异常，有了这个ex，我们就可以得到异常信息。  \*/  @Override  public void uncaughtException(Thread thread, Throwable ex) {  try {  //导出异常信息到SD卡中  dumpExceptionToSDCard(ex);  //这里可以通过网络上传异常信息到服务器，便于开发人员分析日志从而解决bug  } catch (IOException e) {  e.printStackTrace();  }  //打印出当前调用栈信息  ex.printStackTrace();  //如果系统提供了默认的异常处理器，则交给系统去结束我们的程序，否则就由我们自己结束自己  if (mDefaultCrashHandler != null) {  mDefaultCrashHandler.uncaughtException(thread, ex);  } else {  Process.killProcess(Process.myPid());  }  }  private void dumpExceptionToSDCard(Throwable ex) throws IOException {  //如果SD卡不存在或无法使用，则无法把异常信息写入SD卡  if (!Environment.getExternalStorageState().equals(Environment.MEDIA\_MOUNTED)) {  GLog.e(TAG, "sdcard unmounted,skip dump exception");  return;  }  File dir = new File(CRASH\_FILE\_PATH);  if (!dir.exists()) {  dir.mkdirs();  }  long current = System.currentTimeMillis();  String time = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date(current));  //以当前时间创建log文件  File file = new File(CRASH\_FILE\_PATH + CRASH\_FILE\_NAME + time + CRASH\_FILE\_NAME\_SUFFIX);  try {  PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter(file)));  //导出发生异常的时间  pw.println(time);  //获取手机信息  getPhoneInfo(pw);  pw.println();  //导出异常的调用栈信息  ex.printStackTrace(pw);  pw.close();  } catch (Exception e) {  GLog.e(TAG, "dump crash info failed");  }  }  private void getPhoneInfo(PrintWriter pw) throws PackageManager.NameNotFoundException {  //TODO 上报一些辅助信息，如应用版本号，系统版本号，手机型号等，方便数据分析和归类  }  } |

|  |
| --- |
| /\*\*  \* 读取当前设备信息 并追加到异常日志中  \*/  private void dumpPhoneInfo(PrintWriter pw) throws PackageManager.NameNotFoundException {  // 获得应用包管理者 并获取存储当前应用的信息对象  PackageManager pm = mContext.getPackageManager();  PackageInfo pi = pm.getPackageInfo(mContext.getPackageName(), PackageManager.GET\_ACTIVITIES);  pw.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*设备信息\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  // 开始写入 应用信息  pw.println("App Version ");  pw.print(" VersionName: ");  pw.println(pi.versionName);  pw.print(" VersionCode: ");  pw.println(pi.versionCode);  // 开始写入 Android版本信息  pw.println("OS Version ");  pw.println(" SDK\_NAME: " + Build.VERSION.RELEASE);  pw.println(" SDK\_INT: " + Build.VERSION.SDK\_INT);  // 开始写入 手机制造商  pw.println("Vendor ");  pw.println(" " + Build.MANUFACTURER);  // 开始写入 手机型号  pw.println("Model ");  pw.println(" " + Build.MODEL);  // 开始写入 CPU架构  pw.println("CPU ABI ");  String[] supportedAbis = SUPPORTED\_ABIS;  for (int i = 0; i < supportedAbis.length; i++) {  pw.println(" " + supportedAbis[i]);  }  pw.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*end\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  }  } |

当发生Crash后，CrashHandler类通过回调方法uncaughtException，在这里可以拿到错误的信息，也可以上报更多的辅助数据去分析。而CrashHandler最好是在AppApplication中注册

CrashHandler crashHandler = new CrashHandler();

crashHandler.init(this);

这样就可以监听Java层的Crash，并保存发生Crash的错误堆栈，或者根据需求增加一些辅助信息。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　有些文章指出Error不能通过try/catch捕获异常，实际上因为Exception和Error都是Throwable的子类，所以，Error也是可以被try/catch的。

### ？Native层Crash监控：写NDK代码

5.2.1节讲述了在Java层发生Crash后，如何捕获并保存Crash日志，Native层和Java层不一样，Native层发生Crash后，虽然可以通过Logcat查看日志，但在应用运行过程中不能通过5.2.1节的方式捕获到异常并上报。

我们知道，Android系统是Linux内核。在Linux上，当程序发生Crash时，也会生成core dump文件，然后分析dump文件查找崩溃的原因。和Linux一样，在Android系统的Linux内核中，一种重要的进程间通信方式就是Linux的信号机制。Linux信号除了用于正常的进程间通信和同步外，还负责监控系统异常和中断，当应用程序发生异常时，Linux内核会生成错误信号并通知当前进程。应用进程接收到错误信号后，可以捕获该信号并执行对应的信号处理函数。而在应用运行过程中发生严重错误时会发生Crash。Linux有一类专门用于描述Crash的信号：SIGALRM、SIGHUP、SIGINT、SIGKILL、SIGPIPE、SIGPOLL、SIGPROF、SIGSYS、SIGTERM、SIGUSR1、SIGUSR2、SIGVTALRM，在Android系统上，Native层最常见的导致Crash的信号量如表5-2所示。

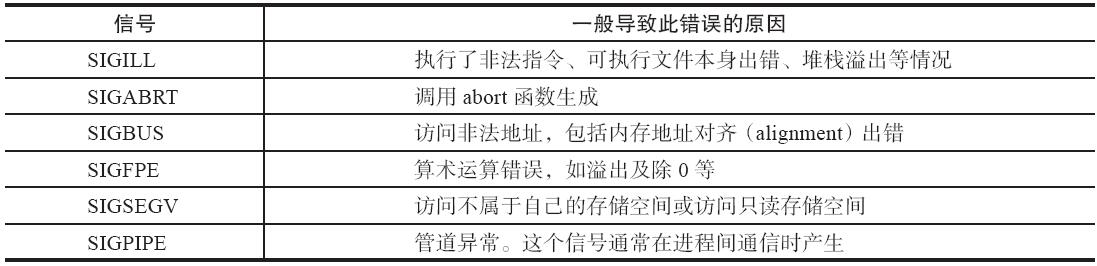


表5-2　常见Android Native Crash信号

只要在应用程序中注册了这些信号的处理函数，当JNI crash时，我们的处理函数就会被调用到，然后获取dump文件再上传，后续的工作就和Java层异常逻辑一致了。Android系统除了保留Linux对信号的默认处理方式外，还定义了一些额外的行为，例如SIGQUIT对SIGQUIT信号的默认行为是“终止+CORE”，即产生core dump文件后，立即终止运行。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　可以通过adb输入kill-l，获取当前手机支持的kill信号列表。

和Java层捕获Crash一样，捕获Native层Crash分两个步骤，捕获Native Crash和获取堆栈信息。

前面讲到，要捕获Native层的Crash，需要先在应用程序中注册导致应用Crash的信号量处理函数，在Android Native层注册信号量函数的代码如下：

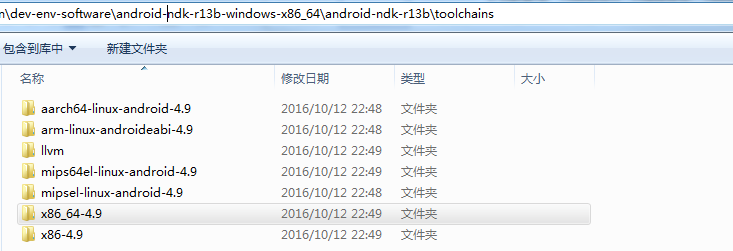
|  |
| --- |
|  |

在Android应用程序中，native层代码通过Dalvik执行System.loadLibrary（），把so库加载进来，然后执行so中的jint JNI\_OnLoad（JavaVM\*vm，void\*reserved）函数。

。。。。。，ndk单独研究，不在此处展开

#### [NDK开发Crash错误定位](http://blog.csdn.net/xyang81/article/details/42319789)

 NDK安装包中提供了三个调试工具：addr2line、objdump和ndk-stack，其中ndk-stack放在$NDK\_HOME目录下，与ndk-build同级目录。addr2line和objdump在ndk的交叉编译器工具链目录下，下面是我本机NDK交叉编译器工具链的目录结构：



### Crash上报机制

不管是Java层，还是Navtice导致了Crash，因为应用已经异常了，所以发生Crash后，尽量先保存日志到本地，在下一次网络正常时再上传日志。Crash日志上报机制如图5-1所示。

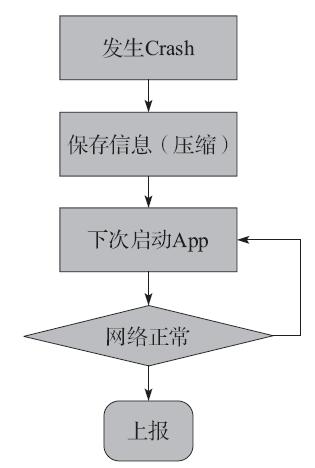


图5-1　Crash日志上报机制

## ANR剖析

Application Not Responding, Android系统对于一些事件需要在一定的时间范围内完成，如果超过预定时间能未能得到有效响应或者响应时间过长，都会造成ANR。一般地，这时往往会弹出一个提示框，告知用户当前xxx未响应，用户可选择继续等待或者Force Close。.

和发生Crash一样，ANR也是Android应用程序无法继续运行的一种异常，和Crash的区别是，它不一定是由于程序的异常错误导致的，一般是应用处理长时间没有结果响应导致主进程不能处理下一件事情。

本节介绍ANR的定义和发生ANR的原因，掌握如何快速定位导致ANR的原因以及如何在开发过程中尽量避免发生ANR。

### ANR介绍

ANR（Application Not Responding）即应用无响应，在Android系统中，应用发生的ANR有以下几种类型：

1.KeyDispatchTimeout(InputDispatching Timeout)

最常见的ANR类型是对输入事件**5s**内无响应，比如**按键或触摸事件**在此时间内无响应。

2.BroadcastTimeout(BroadcastQueue)

BroadcastReceiver在指定时间（原生系统默认是**10s**）内无法处理完成，并且没有结束执行onReceive。对于前台广播，则超时为BROADCAST\_FG\_TIMEOUT = 10s；对于后台广播，则超时为BROADCAST\_BG\_TIMEOUT = 60s; 超时检测机制为超过一定时间没有执行完相应操作来触发移除延时消息，则会触发anr;

3. ContentProvider Timeout：内容提供者, CONTENT\_PROVIDER\_PUBLISH\_TIMEOUT = 10s;

4. ServiceTimeout：对于前台服务，则超时为SERVICE\_TIMEOUT = 20s；对于后台服务，则超时为SERVICE\_BACKGROUND\_TIMEOUT = 200s；超时检测机制：有序广播的总执行时间超过 2\* receiver个数 \* timeout时长，则会触发anr;有序广播的某一个receiver执行过程超过 timeout时长，则会触发anr;这种类型在Android应用中出现的概率很小

引起ANR的根本原因总体来说有以下两种：

1）应用程序自身逻辑有缺陷，或者在某些异常场景触发了此缺陷，如主线程堵塞、死循环等导致。

2）由于Android设备**其他进程的CPU**占用高、事件没有得到及时的响应、死锁，导致当前应用进程无法抢占到CPU时间片。

所有类型的ANR的过程可分为三个步骤: 埋炸弹, 拆炸弹, 引爆炸弹，下面以Service Timeout为例。

### Service Timeout

Service Timeout是位于”ActivityManager”线程中的AMS.MainHandler收到SERVICE\_TIMEOUT\_MSG消息时触发。

对于Service有两类:对于前台服务，则超时为SERVICE\_TIMEOUT = 20s；

对于后台服务，则超时为SERVICE\_BACKGROUND\_TIMEOUT = 200s

由变量ProcessRecord.execServicesFg来决定是否前台启动

[理解Android ANR的触发原理](http://gityuan.com/2016/07/02/android-anr/)

#### 埋炸弹

文章[startService流程分析](http://gityuan.com/2016/03/06/start-service/)详细介绍Service启动流程. 其中在Service进程attach到system\_server进程的过程中会调用realStartServiceLocked()方法来埋下炸弹. 通过发送delay消息(SERVICE\_TIMEOUT\_MSG). 炸弹已埋下, 我们并不希望炸弹被引爆, 那么就需要在炸弹爆炸之前拆除炸弹.

|  |
| --- |
| ActiveServices.realStartServiceLocked(ServiceRecord r,  ProcessRecord app, boolean execInFg) throws RemoteException {  //发送delay消息(SERVICE\_TIMEOUT\_MSG)，  bumpServiceExecutingLocked(r, execInFg, "create");  try {  //最终执行服务的onCreate()方法  app.thread.scheduleCreateService(r, r.serviceInfo,  mAm.compatibilityInfoForPackageLocked(r.serviceInfo.applicationInfo),  app.repProcState);  } catch (DeadObjectException e) {  mAm.appDiedLocked(app);  throw e;  }  }  ActiveServices.bumpServiceExecutingLocked(ServiceRecord r, boolean fg, String why) {  scheduleServiceTimeoutLocked(r.app);  }  ActiveServices.scheduleServiceTimeoutLocked(ProcessRecord proc) {  long now = SystemClock.uptimeMillis();  Message msg = mAm.mHandler.obtainMessage(  ActivityManagerService.SERVICE\_TIMEOUT\_MSG);  msg.obj = proc;  //当超时后仍没有remove该SERVICE\_TIMEOUT\_MSG消息，则执行service Timeout流程【引爆】  **mAm.mHandler.sendMessageAtTime(msg,**  **proc.execServicesFg ? (now+SERVICE\_TIMEOUT) : (now+ SERVICE\_BACKGROUND\_TIMEOUT));**  } |

#### 拆炸弹

在system\_server进程AS.realStartServiceLocked()调用的过程会埋下一颗炸弹, 超时没有启动完成则会爆炸. 那么什么时候会拆除这颗炸弹的引线呢? 经过Binder等层层调用进入目标进程的主线程handleCreateService()的过程.

|  |
| --- |
| ActivityThread.handleCreateService(CreateServiceData data) {  java.lang.ClassLoader cl = packageInfo.getClassLoader();  Service service = (Service) cl.loadClass(data.info.name).newInstance();  try {  //创建ContextImpl对象  ContextImpl context = ContextImpl.createAppContext(this, packageInfo);  context.setOuterContext(service);  //创建Application对象  Application app = packageInfo.makeApplication(false, mInstrumentation);  service.attach(context, this, data.info.name, data.token, app,  ActivityManagerNative.getDefault());  //调用服务onCreate()方法  service.onCreate();    **//拆除炸弹引线**  ActivityManagerNative.getDefault().serviceDoneExecuting(  data.token, SERVICE\_DONE\_EXECUTING\_ANON, 0, 0);  }  }  AS.serviceDoneExecutingLocked(ServiceRecord r, boolean inDestroying,  boolean finishing) {  if (r.executeNesting <= 0) {  if (r.app != null) {  r.app.execServicesFg = false;  r.app.executingServices.remove(r);  if (r.app.executingServices.size() == 0) {  //当前服务所在进程中没有正在执行的service,则移除服务超时消息SERVICE\_TIMEOUT\_MSG  **mAm.mHandler.removeMessages(ActivityManagerService.SERVICE\_TIMEOUT\_MSG, r.app**);  }  } |

#### 引爆炸弹

前面介绍了埋炸弹和拆炸弹的过程, 如果在炸弹倒计时结束之前成功拆卸炸弹,那么就没有爆炸的机会, 但是世事难料. 总有些极端情况下无法即时拆除炸弹,导致炸弹爆炸, 其结果就是App发生ANR. 接下来,带大家来看看炸弹爆炸的现场:

在system\_server进程中有一个Handler线程, 名叫”ActivityManager”.当倒计时结束便会向该Handler线程发送 一条信息SERVICE\_TIMEOUT\_MSG,

|  |
| --- |
| ActivityManagerService.MainHandler **extends** Handler{  **public void** handleMessage(Message msg){  **switch**(msg.what){  **case** SERVICE\_TIMEOUT\_MSG:{  mServices.serviceTimeout((ProcessRecord)msg.obj);  }**break**;  }  }  } AS.serviceTimeout(ProcessRecord proc){  String anrMessage=**null**;  **synchronized**(mAm){  **if**(proc.executingServices.size()==0||proc.thread==**null**) **return**;  **final long** now=SystemClock.uptimeMillis();  **final long** maxTime=now-  (proc.execServicesFg?SERVICE\_TIMEOUT:SERVICE\_BACKGROUND\_TIMEOUT);  ServiceRecord timeout=**null**;  **long** nextTime=0;  **for**(**int** i=proc.executingServices.size()-1;i>=0;i--){  ServiceRecord sr=proc.executingServices.valueAt(i);  **if**(sr.executingStart<maxTime){  timeout=sr;  **break**;  }  **if**(sr.executingStart>nextTime){  nextTime=sr.executingStart;  }  }  **if**(timeout!=**null**&&mAm.mLruProcesses.contains(proc)){  Slog.w(TAG,**"Timeout executing service: "**+timeout);  StringWriter sw=**new** StringWriter();  PrintWriter pw=**new** FastPrintWriter(sw,**false**,1024);  pw.println(timeout);  timeout.dump(pw,**" "**);  pw.close();  mLastAnrDump=sw.toString();  mAm.mHandler.removeCallbacks(mLastAnrDumpClearer);  mAm.mHandler.postDelayed(mLastAnrDumpClearer,LAST\_ANR\_LIFETIME\_DURATION\_MSECS);  anrMessage=**"executing service "**+timeout.shortName;  }  }   **if**(anrMessage!=**null**){   ***//当存在timeout的service，则执行appNotResponding***mAm.appNotResponding(proc,**null**,**null**,**false**,anrMessage);  } } |

### KeyDispatchTimeout具体的原因

我们需要关注的是第一种情况导致的ANR。**KeyDispatchTimeout具体的原因有**：

(1). 当前的事件没有机会得到处理（即UI线程正在处理前一个事件，没有及时的完成或者looper被某种原因阻塞住了）

(2). 当前的事件正在处理，但没有及时完成

首先要知道事件发生的线程，一般来说大多数可能是ui线程操作超时，那么ui线程都有哪些呢：

(1). Activity 生命周期

(2). View post 的runnable方法 、 handler（MainLooper） 的 handleMessage()

(3). Asycktask 的 onPreExecute(), onPostExecute() , onProgressUpdate()方法

(4). Broadcast 的onReceive()

(5). Service

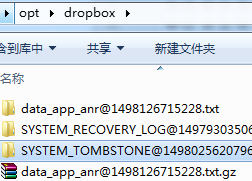
其他原因造成的anr：

(1). 线程死锁

(2). cpu 饥饿

### ANR分析

在Android系统上，如果发生ANR，Logcat会产生对应的日志和一个**/data/anr/traces.txt**文件，只保留最后一次发生ANR时的信息，“/data/system/dropbox则是保存最近3d的anr。”是DB指定的文件存放位置.分析ANR原因主要是分析这两个信息。但是Logcat不一定保证在发生ANR时可以拿到，特别是在非开发人员手上发生ANR情况时，再去拿logcat已经晚了，并且ANR的原因非常复杂。



#### Traces系统源码

SystemServer在启动时， 会创建并添加DROPBOX\_SERVICE.

|  |
| --- |
| //SystemServer.java  private void startOtherServices() {  try {  Slog.i(TAG, "DropBox Service");  ServiceManager.addService(Context.DROPBOX\_SERVICE,  new DropBoxManagerService(context, new File("/data/system/dropbox")));  } catch (Throwable e) {  reportWtf("starting DropBoxManagerService", e);  }  } |

framework是如何输出ANR信息到traces.txt文件中去的？

ActivityManagerService.java中的appNotResponding()方法

|  |
| --- |
| final void appNotResponding(ProcessRecord app, ActivityRecord activity,  ActivityRecord parent, boolean aboveSystem, final String annotation) {  //firstPids和lastPids两个集合存放那些将会在traces中输出信息的进程的进程号  ArrayList<Integer> firstPids = new ArrayList<Integer>(5);  SparseArray<Boolean> lastPids = new SparseArray<Boolean>(20);  //mController是IActivityController接口的实例，是为Monkey测试程序预留的，默认为null  if (mController != null) {  try {  // 0 == continue, -1 = kill process immediately  int res = mController.appEarlyNotResponding(app.processName, app.pid, annotation);  if (res < 0 && app.pid != MY\_PID) {  app.kill("anr", true);  }  } catch (RemoteException e) {  mController = null;  Watchdog.getInstance().setActivityController(null);  }  }  long anrTime = SystemClock.uptimeMillis();  if (MONITOR\_CPU\_USAGE) {  updateCpuStatsNow(); //更新CPU使用率  }  synchronized (this) {  //一些特定条件下会忽略ANR  // PowerManager.reboot() can block for a long time, so ignore ANRs while shutting down.  if (mShuttingDown) {  Slog.i(TAG, "During shutdown skipping ANR: " + app + " " + annotation);  return;  } else if (app.notResponding) {  Slog.i(TAG, "Skipping duplicate ANR: " + app + " " + annotation);  return;  } else if (app.crashing) {  Slog.i(TAG, "Crashing app skipping ANR: " + app + " " + annotation);  return;  }  // In case we come through here for the same app before completing  // this one, mark as anring now so we will bail out.  app.notResponding = true;  // Log the ANR to the event log.  EventLog.writeEvent(EventLogTags.AM\_ANR, app.userId, app.pid,  app.processName, app.info.flags, annotation);  //当前发生ANR的应用进程被第一个添加进firstPids集合中.  //**所以会第一个向traces文件中写入信息**。反过来说，traces文件中出现的第一个进程正常情况下就是发生ANR的那个进程。  // Dump thread traces as quickly as we can, starting with "interesting" processes.  firstPids.add(app.pid);  int parentPid = app.pid;  if (parent != null && parent.app != null && parent.app.pid > 0) parentPid = parent.app.pid;  if (parentPid != app.pid) firstPids.add(parentPid);  if (MY\_PID != app.pid && MY\_PID != parentPid) firstPids.add(MY\_PID);  for (int i = mLruProcesses.size() - 1; i >= 0; i--) {  ProcessRecord r = mLruProcesses.get(i);  if (r != null && r.thread != null) {  int pid = r.pid;  if (pid > 0 && pid != app.pid && pid != parentPid && pid != MY\_PID) {  if (r.persistent) {  firstPids.add(pid);  } else {  lastPids.put(pid, Boolean.TRUE);  }  }  }  }  }  // Log the ANR to the main log.  StringBuilder info = new StringBuilder();  info.setLength(0);  info.append("ANR in ").append(app.processName);  if (activity != null && activity.shortComponentName != null) {  info.append(" (").append(activity.shortComponentName).append(")");  }  info.append("\n");  info.append("PID: ").append(app.pid).append("\n");  if (annotation != null) {  info.append("Reason: ").append(annotation).append("\n");  }  if (parent != null && parent != activity) {  info.append("Parent: ").append(parent.shortComponentName).append("\n");  }  final ProcessCpuTracker processCpuTracker = new ProcessCpuTracker(true);  //dumpStackTraces()是输出ANR信息到traces文件的函数  File tracesFile = dumpStackTraces(true, firstPids, processCpuTracker, lastPids,  NATIVE\_STACKS\_OF\_INTEREST);  String cpuInfo = null;  if (MONITOR\_CPU\_USAGE) {  updateCpuStatsNow();  synchronized (mProcessCpuTracker) {  cpuInfo = mProcessCpuTracker.printCurrentState(anrTime);  }  //输出ANR发生前一段时间内的CPU使用率  info.append(processCpuTracker.printCurrentLoad());  info.append(cpuInfo);  }  //输出ANR发生后一段时间内的CPU使用率  info.append(processCpuTracker.printCurrentState(anrTime));  //此处， 把ANR的信息输出到main log中.  Slog.e(TAG, info.toString());  if (tracesFile == null) {  // There is no trace file, so dump (only) the alleged culprit's threads to the log  Process.sendSignal(app.pid, Process.SIGNAL\_QUIT);  }  //将ANR信息同时输出到DropBox中  addErrorToDropBox("anr", app, app.processName, activity, parent, annotation,  cpuInfo, tracesFile, null);  //在>=Android4.0中可以设置是否不显示ANR提示对话框，如果设置的话就不会显示对话框，并且会杀掉ANR进程  // Unless configured otherwise, swallow ANRs in background processes & kill the process.  boolean showBackground = **Settings.Secure.getInt(mContext.getContentResolver(),**  **Settings.Secure.ANR\_SHOW\_BACKGROUND, 0) != 0;**  synchronized (this) {  mBatteryStatsService.noteProcessAnr(app.processName, app.uid);  if (!showBackground && !app.isInterestingToUserLocked() && app.pid != MY\_PID) {  app.kill("bg anr", true);  return;  }  // Set the app's notResponding state, and look up the errorReportReceiver  makeAppNotRespondingLocked(app,  activity != null ? activity.shortComponentName : null,  annotation != null ? "ANR " + annotation : "ANR",  info.toString());  // 显示ANR提示对话框  // Bring up the infamous App Not Responding dialog  Message msg = Message.obtain();  HashMap<String, Object> map = new HashMap<String, Object>();  msg.what = SHOW\_NOT\_RESPONDING\_MSG;  msg.obj = map;  msg.arg1 = aboveSystem ? 1 : 0;  map.put("app", app);  if (activity != null) {  map.put("activity", activity);  }  mUiHandler.sendMessage(msg);  }  }  public static File dumpStackTraces(boolean clearTraces, ArrayList<Integer> firstPids,  ProcessCpuTracker processCpuTracker, SparseArray<Boolean> lastPids, String[] nativeProcs) {  //系统属性“dalvik.vm.stack-trace-file”用来配置trace信息输出文件  //之所以trace信息会输出到“/data/anr/traces.txt”文件中，就是系统属性“dalvik.vm.stack-trace-file”设置的  //adb shell下， 可以使用setprop和getprop对系统属性进行设置和读取  //getprop dalvik.vm.stack-trace-file  //setprop dalvik.vm.stack-trace-file /tmp/stack-traces.txt  String tracesPath = SystemProperties.get("dalvik.vm.stack-trace-file", null);  if (tracesPath == null || tracesPath.length() == 0) {  return null;  }  File tracesFile = new File(tracesPath);  try {  File tracesDir = tracesFile.getParentFile();  if (!tracesDir.exists()) {  tracesDir.mkdirs();  if (!SELinux.restorecon(tracesDir)) {  return null;  }  }  //FileUtils.setPermissions是个很有用的函数，设置文件属性时经常会用到  FileUtils.setPermissions(tracesDir.getPath(), 0775, -1, -1); // drwxrwxr-x  //clearTraces为true，会删除旧文件，创建新文件  if (clearTraces && tracesFile.exists()) tracesFile.delete();  tracesFile.createNewFile();  FileUtils.setPermissions(tracesFile.getPath(), 0666, -1, -1); // -rw-rw-rw-  } catch (IOException e) {  Slog.w(TAG, "Unable to prepare ANR traces file: " + tracesPath, e);  return null;  }  dumpStackTraces(tracesPath, firstPids, processCpuTracker, lastPids, nativeProcs);  return tracesFile;  }  private static void dumpStackTraces(String tracesPath, ArrayList<Integer> firstPids,  ProcessCpuTracker processCpuTracker, SparseArray<Boolean> lastPids, String[] nativeProcs) {  // Use a FileObserver to detect when traces finish writing.  // The order of traces is considered important to maintain for legibility.  //使用FileObserver监听SignalCatcher线程是否已经完成写入traces文件的操作  FileObserver observer = new FileObserver(tracesPath, FileObserver.CLOSE\_WRITE) {  @Override  public synchronized void onEvent(int event, String path) {  notify(); //让其他阻塞在observer.wait()这行代码的线程继续往下执行.  }  };  try {  observer.startWatching();  //首先输出firstPids集合中指定的进程，这些也是对ANR问题来说最重要的进程  // First collect all of the stacks of the most important pids.  if (firstPids != null) {  try {  int num = firstPids.size();  for (int i = 0; i < num; i++) {  synchronized (observer) {  //前面提到的SIGNAL\_QUIT, 由SignalCatcher线程处理Process.SIGNAL\_QUIT信号， 并把进程中的各个线程输出到traces.txt文件中.  //具体代码可以查看/dalvik/vm/目录下的SignalCatcher.cpp::logThreadStacks函数和Thread.cpp:: dvmDumpAllThreadsEx函数。  Process.sendSignal(firstPids.get(i), Process.SIGNAL\_QUIT);  observer.wait(200); // Wait for write-close, give up after 200msec  //block here, 直到另一个线程调用了observer.notify() or observer.notifyAll()方法， 再或是等待了200毫秒超时后， 才会继续往下执行代码.  }  }  } catch (InterruptedException e) {  Slog.wtf(TAG, e);  }  }  } finally {  observer.stopWatching();  }  }  Object.java  final void notify()  //Causes a thread which is waiting on this object's monitor (by means of calling one of the wait() methods) to be woken up.  final void notifyAll()  //Causes all threads which are waiting on this object's monitor (by means of calling one of the wait() methods) to be woken up.  final void wait()  //Causes the calling thread to wait until another thread calls the notify() or notifyAll() method of this object.  final void wait(long millis)  //Causes the calling thread to wait until another thread calls the notify() or notifyAll() method  //of this object or until the specified timeout expires. |

getprop dalvik.vm.stack-trace-file

#### 分析工具Analyze Stacktrace

在Android Studio上提供了一个分析trace文件的工具：**Analyze Stacktrace**。Analyze Stacktrace可以更直观地分析导致ANR的原因。

Analyze Stacktrace使用方法如下：

1）在Android Studio的工具栏中，选择Analyze→Analyze Stacktrace，打开Analyze Stacktrace工具窗口。

2）将traces.txt中的内容复制到窗口，单击Normalize按钮，生成Thread Dump列表，如图5-2所示，左边为所有线程列表，右边为选中线程的具体信息。注意有可能不显示没有上锁的线程。值得注意的是， 当前运行的其他进程也会把当前的函数堆栈信息输出到traces.txt文件中, 但发生ANR的进程正常情况下会第一个输出，所以一般情况下我们只看traces.txt的开头就行了。

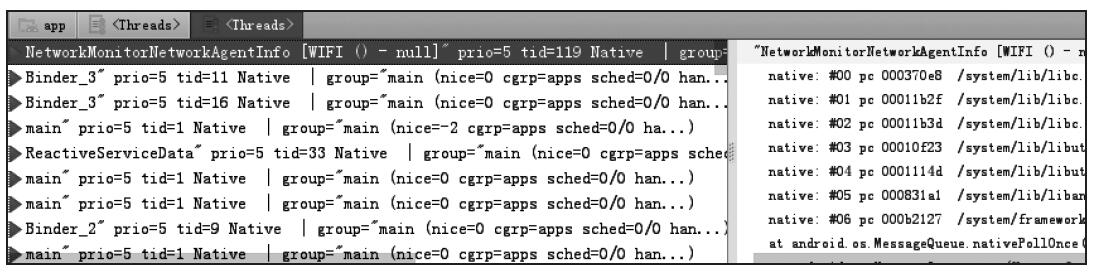


图5-2　Thread Dump列表

3）如果某个线程被标红，说明此线程被堵塞了，然后在右边的详细信息中查看堵塞的具体原因。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　部分ANR是由于多个线程之间的同步锁导致，在使用Analyze Stacktrace分析时，可以在详情页查看是在等待哪个线程，再定位到对应的线程。

从以上的分析能够知道， 发生ANR时Android为我们提供了两种“利器”：traces文件和CPU使用率。**线程有很多状态，了解这些状态的意义对分析ANR的原因是有帮助的.**



开发中定位ANR问题日志有个很简单的规律，就是直接找到我们自己开发App所使用的包名（包括第三方Library库）信息开始定位找就可以了

[浅析android 线程状态](http://www.jianshu.com/p/3fb50ddc36d0)，java的6种线程状态定义在/java/lang/Thread.java中:在VMThread.java中, 可以看到下面的代码， native thread有10种状态, 对应着java thread的6种状态.

//

static final Thread.State[] STATE\_MAP = new Thread.State[] {

Thread.State.TERMINATED, // ZOMBIE

Thread.State.RUNNABLE, // RUNNING

Thread.State.TIMED\_WAITING, // TIMED\_WAIT

Thread.State.BLOCKED, // MONITOR

Thread.State.WAITING, // WAIT

Thread.State.NEW, // INITIALIZING

Thread.State.NEW, // STARTING

**Thread.State.RUNNABLE, // NATIVE**

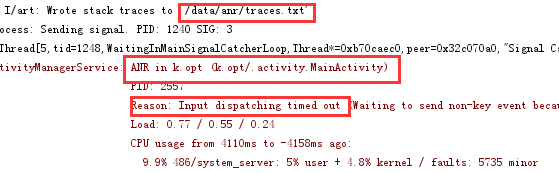
Thread.State.WAITING, // VMWAIT

**Thread.State.RUNNABLE // SUSPENDED**

};

#### KeyDispatchTimeout分析

这里简单构造一个ANR，在HomePageActivity中的单击处理函数中等待100s（Thread，sleep），然后在屏幕上滑动，投递触屏事件，结果出现ANR。发生ANR后，首先看Logcat的信息，**Logcat**信息如下：

从以上Logcat信息中，关注标粗的几个值的信息，它们的意义如下：

·ANR IN：发生ANR的具体类。

·PID：发生ANR的进程，系统在此时会生成trace文件，当前的时间点也是发生ANR的具体时间，以及生成trace文件的时间。

·Reason：当前ARN的类型以及导致ANR的原因。

·CPU usage：CPU的使用情况，在日志中CPU usage有两个时间点，第一个是发生ANR前的CPU使用情况，第二个是发生ANR后的CPU使用情况。

此时用户如果在屏幕上滑动，会产生InputDispatcher事件得不到处理

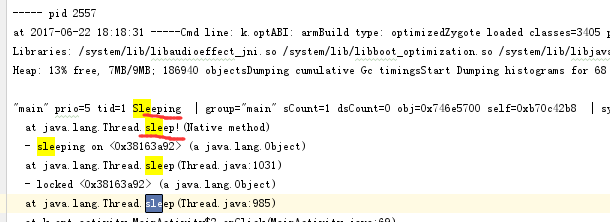
从Logcat中除了能看出在哪个类发生ANR以及ANR的类型，具体的原因主要还是看CPU的使用情况，如果**CPU使用量很少，说明主线程可能阻塞**，如果IOwait很高，说明ANR有可能是由于主线程进行耗时的I/O操作造成的。

接下来分析这个文件**/data/anr/traces.txt**。

可以直接使用adb工具获取traces.txt文件：adb pull/data/anr/traces.txt.

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　trace文件在非root手机上也可以读取，虽然在非root手机上没有权限查看data目录下的内容，**但anr目录可以读取**。

用工具使用，读取出来的traces.txt内容如下：



因此当前这个例子故意写了一个ANR，所以很容易定位到就是在onClick事件中超时导致，在这个日志中需要关注的是线程的状态。

#### [死锁ANR](http://blog.csdn.net/oujunli/article/details/9102101)分析

参考：[通过Android trace文件分析死锁ANR](http://blog.csdn.net/oujunli/article/details/9102101)

|  |
| --- |
| 对应的部分trace文件内容如下：  "PowerManagerService" prio=5 tid=24 MONITOR   | group="main" sCount=1 dsCount=0 obj=0x41dd0eb0 self=0x5241b218   | sysTid=567 nice=0 sched=0/0 cgrp=apps handle=1380038664   | state=S schedstat=( 6682116007 11324451214 33313 ) utm=450 stm=219 core=1   at com.[android](http://lib.csdn.net/base/android).server.am.ActivityManagerService.broadcastIntent(ActivityManagerService.[Java](http://lib.csdn.net/base/java):~13045)   - waiting to lock <0x41a874a0> (a com.android.server.am.ActivityManagerService) held by tid=12 (android.server.ServerThread)   at android.app.ContextImpl.sendBroadcast(ContextImpl.java:1144)   at com.android.server.power.PowerManagerService$DisplayBlankerImpl.unblankAllDisplays(PowerManagerService.java:3442)   at com.android.server.power.DisplayPowerState$PhotonicModulator$1.run(DisplayPowerState.java:456)   at android.os.Handler.handleCallback(Handler.java:800)   at android.os.Handler.dispatchMessage(Handler.java:100)   at android.os.Looper.loop(Looper.java:194)   at android.os.HandlerThread.run(HandlerThread.java:60)      "Binder\_B" prio=5 tid=85 MONITOR   | group="main" sCount=1 dsCount=0 obj=0x42744770 self=0x58329e88   | sysTid=3700 nice=-20 sched=0/0 cgrp=apps handle=1471424616   | state=S schedstat=( 1663727513 2044643318 6806 ) utm=132 stm=34 core=1   at com.android.server.power.PowerManagerService$DisplayBlankerImpl.toString(PowerManagerService.java:~3449)   - waiting to lock <0x41a7e420> (a com.android.server.power.PowerManagerService$DisplayBlankerImpl) held by tid=24 (PowerManagerService)   at java.lang.StringBuilder.append(StringBuilder.java:202)   at com.android.server.power.PowerManagerService.dump(PowerManagerService.java:3052)   at android.os.Binder.dump(Binder.java:264)   at android.os.Binder.onTransact(Binder.java:236)   at android.os.IPowerManager$Stub.onTransact(IPowerManager.java:373)   at android.os.Binder.execTransact(Binder.java:351)   at dalvik.system.NativeStart.run(Native Method)    "android.server.ServerThread" prio=5 tid=12 MONITOR   | group="main" sCount=1 dsCount=0 obj=0x41a76178 self=0x507837a8   | sysTid=545 nice=-2 sched=0/0 cgrp=apps handle=1349936616   | state=S schedstat=( 15368096286 21707846934 69485 ) utm=1226 stm=310 core=0   at com.android.server.power.PowerManagerService.isScreenOnInternal(PowerManagerService.java:~2529)   - waiting to lock <0x41a7e2e8> (a java.lang.Object) held by tid=85 (Binder\_B)   at com.android.server.power.PowerManagerService.isScreenOn(PowerManagerService.java:2522)   at com.android.server.wm.WindowManagerService.sendScreenStatusToClientsLocked(WindowManagerService.java:7749)   at com.android.server.wm.WindowManagerService.setEventDispatching(WindowManagerService.java:7628)   at com.android.server.am.ActivityManagerService.updateEventDispatchingLocked(ActivityManagerService.java:8083)   at com.android.server.am.ActivityManagerService.wakingUp(ActivityManagerService.java:8077)   at com.android.server.power.Notifier.sendWakeUpBroadcast(Notifier.java:474)   at com.android.server.power.Notifier.sendNextBroadcast(Notifier.java:455)   at com.android.server.power.Notifier.access$700(Notifier.java:62)   at com.android.server.power.Notifier$NotifierHandler.handleMessage(Notifier.java:600)   at android.os.Handler.dispatchMessage(Handler.java:107)   at android.os.Looper.loop(Looper.java:194)   at com.android.server.ServerThread.run(SystemServer.java:1328) |

从trace文件看，是因为TID为24的线程等待一个TID为12的线程持有的锁，TID为12的线程等待一个TID为85的线程持有的锁，而TID为85的线程确等待一个TID为24的线程持有的锁，导致了循环等待的现象，对应的trace文件的语句如下：

TID 24：- waiting to lock <0x41a874a0> (a com.android.server.am.ActivityManagerService) held by tid=12 (android.server.ServerThread)

TID 12： - waiting to lock <0x41a7e2e8> (a java.lang.Object) held by tid=85 (Binder\_B)

TID 85：- waiting to lock <0x41a7e420> (a com.android.server.power.PowerManagerService$DisplayBlankerImpl) held by tid=24 (PowerManagerService)

    既然是死锁，那么先看各线程都有那些锁。

先看TID=24的线程的栈顶，ActivityManagerService的broadcastIntent函数代码如下:

public final int broadcastIntent(IApplicationThread caller,

Intent intent, String resolvedType, IIntentReceiver resultTo,

int resultCode, String resultData, Bundle map,

String requiredPermission, boolean serialized, boolean sticky, int userId) {

enforceNotIsolatedCaller("broadcastIntent");

synchronized(this) {

intent = verifyBroadcastLocked(intent);

final ProcessRecord callerApp = getRecordForAppLocked(caller);

final int callingPid = Binder.getCallingPid();

final int callingUid = Binder.getCallingUid();

final long origId = Binder.clearCallingIdentity();

int res = broadcastIntentLocked(callerApp,

callerApp != null ? callerApp.info.packageName : null,

intent, resolvedType, resultTo,

resultCode, resultData, map, requiredPermission, serialized, sticky,

callingPid, callingUid, userId);

Binder.restoreCallingIdentity(origId);

return res;

}

可以看到TID=24需要ActivityManagerService这个锁。再看TID=12线程的栈顶，PowerManagerService的isScreenOnInternal函数代码如下：

private boolean isScreenOnInternal() {

synchronized (mLock) {

return !mSystemReady

|| mDisplayPowerRequest.screenState != DisplayPowerRequest.SCREEN\_STATE\_OFF;

}

}

可以看到需要PowerManagerService的mlock这个锁。最后看TID=85线程的栈顶，同样在PowerManagerService里面，内部类DisplayBlankerImpl的toString函数：

public String toString() {

synchronized (this) {

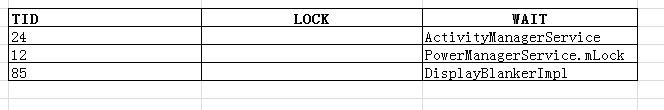
return "blanked=" + mBlanked;

}

}

这是在内部类DisplayBlankerImpl里面实现的，所以需要DisplayBlankerImpl这个锁。

对应的表格如下：



各线程等待的锁情况

死锁的原因是持有锁，又等待锁，下面我们要填充LOCK列，死锁原因就一目了然了。

跟踪TID=24线程的堆栈，在PowerManagerService内部类**DisplayBlankerImp**l的unblankAllDisplays函数中持有锁，持有了this就是持有了**DisplayBlankerImpl**

public void unblankAllDisplays() {

**synchronized (this)** {

mDisplayManagerService.unblankAllDisplaysFromPowerManager()}

}

跟踪TID=12线程的堆栈，在ActivityManagerService的wake\_up函数中持有锁，持有了this就是持有了**ActivityManagerService**

public void wakingUp() {

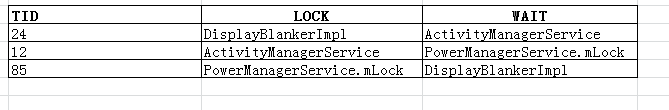
synchronized(this) { } }

跟踪TID=85线程的堆栈，在PowerManagerService的dump有持有锁**mLock**的操作

protected void dump(FileDescriptor fd, PrintWriter pw, String[] args) {

synchronized (mLock) {}

因此得到以下的变量lock和wait图，可以清晰的看出死锁了，资源循环等待了。



表二 各线程锁的情况

分析方法：先填充wait列的同步变量，然后在每个线程执行的堆栈的方法中，找出这些同步变量，添加到lock里面就好！

[非常经典的一个案例：Android ANR问题原因分析](http://blog.csdn.net/sinat_22657459/article/details/52780759)

[ANR问题的分析流程](http://blog.csdn.net/jiangguohu1/article/details/52636470)

### 如何避免

* 避免在主线程进行复杂耗时的操作，特别是文件读取或者数据库操作；
* 避免频繁实时更新UI；

long currTime = SystemClock.uptimeMillis();

if (currTime - lastUpdateTime >= MIN\_RATE&&!progressModel.isDone()) {

lastUpdateTime = currTime;

* BroadCastReceiver 要进行复杂操作的的时候，可以在onReceive()方法中启动一个Service来处理；
* 避免在IntentReceiver里启动一个Activity，因为它会创建一个新的画面，并从当前用户正在运行的程序上抢夺焦点。如果你的应用程序在响应Intent广 播时需要向用户展示什么，你应该使用Notification Manager来实现。
* 在设计及代码编写阶段避免出现出现同步/死锁或者错误处理不恰当等情况。

### ANR监控

发生ANR的体验非常糟糕，因此有必要掌握外网发生ANR的情况。ANR的监控原理和卡顿监控的原理类似，我们知道，卡顿监控可以通过一个子线程向主线程发消息，通过时间差来判断是否发生卡顿，如图5-3所示。卡顿监控详见2.8.2节。

从图5-3中可以看到，是否卡顿取决于T2-T1的时间，发生ANR后，只有T1的时间，没有T2的时间，发生消息后，由于主线程卡住导致消息一直没有执行，这时就不会再回到打印T2的时间戳了，因此，可以使用这个原理监控和上报ANR。

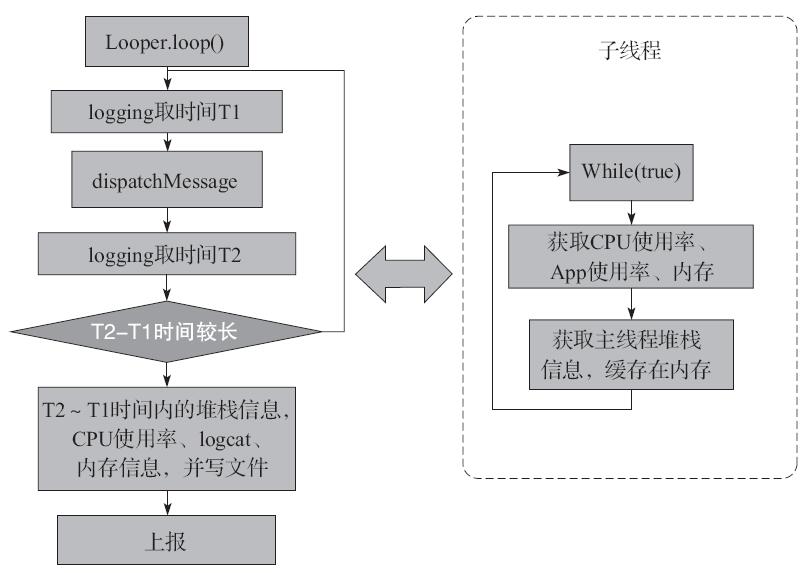
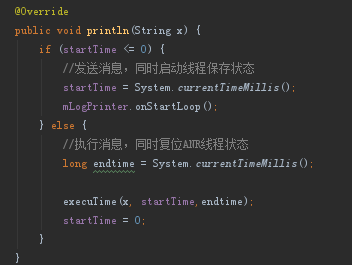


图5-3　卡顿监控原理

在代码清单2-10的基础上，为卡顿监控代码增加ANR的线程监控，在发送消息时，在ANR线程中保存一个状态，主线程消息执行完后再Reset标志位。如果在ANR线程中收到发送消息后，超过一定时间（也就是超时时间），没有复位，就可以认为发生了ANR，这时可以上报相关trace数据给开发人员分析，代码如下：



这个方案有个缺点就是它是通过在主线程监控执行时间的，但不同手机系统的ANR时间不一致，阈值的设定不好掌握，在这个阈值内，主线程还没有处理完，则认为发生了ANR；或者主线程已经发生ANR了，但还要等线程执行完后才能记录该ANR。会有一定概率的误报。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　和卡顿监控一样，数据的上报和Dump信息最好不要在全量版本中打开，否则会影响性能。

## 提高后台进程存活率

在Android系统中，应用进程停止运行有以下几个原因。

* + ·用户主动退出。
  + ·Crash异常退出。
  + ·系统通过杀掉进程回收内存。

其中用户主动退出是合理行为，Crash在前面分析过，这里不再重复分析。而系统通过杀掉应用进程回收内存，理论上这也是合理的形为。因为Android有一个淘汰机制，这个机制会根据应用的运行状态设置一个进程的优先级，然后根据系统整体内存使用状态进行调整，当内存占用达到一定阈值时，就需要把一些优先级低的进程杀掉，以保证其他进程能够有足够的内存使用。

但是杀掉应用进程对体验会有一些影响，比如杀掉后，用户再打开应用耗时会更长（冷启动），如果能从最大程序上提高应用进程的存活率，可以提高热启动的概率，减少冷启动的概率，让用户更快进入应用中，并且提高进程存活率在一些有后台服务进程的应用意义更加明显，比如播放器，如果在后台听歌过程中应用进程被杀，会中断歌曲播放。本节介绍Android系统进程优先级，学习提高应用进程存活的方法。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　降低应用内存开销是减小进程被系统回收概率的一个非常有效的方法，降低应用内存详见第3章。

### 应用进程优先级

在Android系统上，应用进程的优先级（oom\_adj值）分为以下等级，不同定义的进程系统采用了不同的管理机制。

·NATIVE\_ADJ=-17

系统创建的Native进程。

·SYSTEM\_ADJ=-16

系统进程，在运行的过程中永远不会杀掉，如果杀掉可能会导致严重问题。

·PERSISTENT\_PROC\_ADJ=-12

核心进程，系统不会杀掉这类进程，但即使杀掉，影响面也没有SYSTEM\_ADJ进程那么严重。

·PERSISTENT\_SERVICE\_ADJ=-11

正在运行的服务进程，一般不会被杀掉。

·FOREGROUND\_APP\_ADJ=0

前台进程，是指正在前台运行的应用，被杀概率不大。

·VISIBLE\_APP\_ADJ=1

可见进程，用户正在使用，或者有界面在显示，除非出现异常，否则系统不会杀掉这类进程。

·PERCEPTIBLE\_APP\_ADJ=2

可感知的进程，虽然不在前台，但应用进程还在状态，系统除非到内存非常紧张才会杀掉这类进程，比如播放音乐的应用。

·BACKUP\_APP\_ADJ=3

正在备份的进程。

·HEAVY\_WEIGHT\_APP\_ADJ=4

高权重进程。

·SERVICE\_ADJ=5

有Service的进程。

·HOME\_APP\_ADJ=6

与Home有交互的进程，比如有桌面部件和应用正在通信，widget小挂件之类，一般尽量避免杀掉此类进程。

·PREVIOUS\_APP\_ADJ=7

切换进程，可以理解为从可见进程切换过来的进程的状态。

·CACHED\_APP\_MIN\_ADJ=8

缓存进程，也就是空进程。

·SERVICE\_B\_ADJ=9

不活跃的进程。

·HIDDEN\_APP\_MIN\_ADJ=15

缓存进程，空进程，在内存不足的情况下会被优先杀掉。

·UNKNOWN\_ADJ=16

最低级别进程，只有缓存的进程，才有可能设置成这个级别。

当内存不足时，进程优先级低的（oom\_adj越大的）、占内存大的App进程将会被优先杀掉，系统杀进程的规则如下：

·进程优先级设置为PERSISTENT\_PROC\_ADJ（-12）被杀概率较低。

·进程优先级HEAVY\_WEIGHT\_APP\_ADJ（4），这种是Activity仅次于主进程，系统认为是高权重进程。

·前台进程FOREGROUND\_APP\_ADJ（0）不会被杀掉。

·当Activity、Service的生命周期发生变化时都会调整进程的优先级（oo\_adj）。

·进程中没有任何Activity存在会优先被杀。

·空进程最容易被杀。

Android系统进程管理机制的出发点是很好的，将一些空进程和挂起进程杀掉，从用户体验上来讲不会有太大影响，但是在部分应用的特殊场景下，如消息推送和后台播放，需要应用在后台时也尽量不要杀掉，因此有时需要适当提高或维持进程的高优先级，以免被系统所杀。

### 利用SyncAdapter提高进程优先级

可以提高进程优先级使应用在后台的存活时间更长，一般使用以下几种方法实现。

（1）网络连接

通过长连接心跳和进程保持通信，使进程保持活动状态，但是如果系统内存非常紧张，也有可能被杀。

（2）利用系统现有机制

一般可以注册系统消息（AlarmReceiver、BootReceiver等），通过系统消息响应挂起进程。

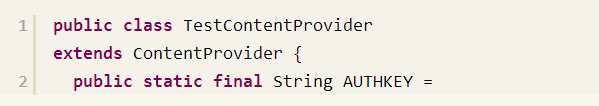
（3）SyncAdapter

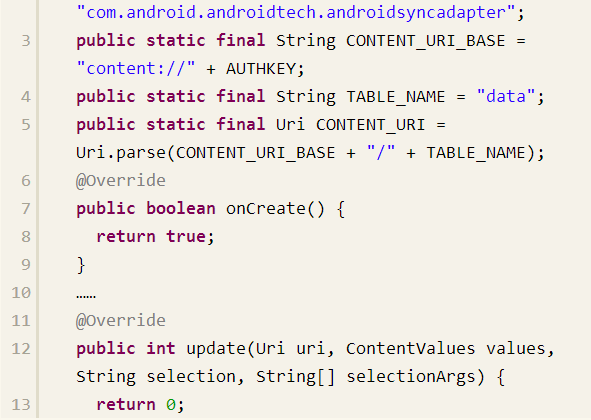
利用Android系统提供的账号同步机制实现进程优先级提高。

前两种使用比较广泛，这里主要介绍第三种：SyncAdapter的使用方法。SyncAdapter是一个系统服务，通过系统的定时器更新应用程序数据ContentProvide r，因为Sync服务工作在独立进程，并且由操作系统调度，进程属于核心进程级别，系统不会杀掉，而使用了SyncAdatper的进程优先级本身也会提高，从而降低应用进程被杀的概率，下面通过具体代码实现这个机制。

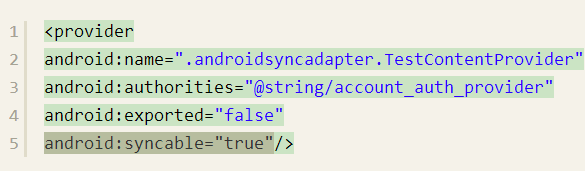
1.创建ContentProvider实现数据同步

首先通过一个ContentProvider来实现数据同步，这个示例不会有具体的数据同步，只建立一个空的ContentProvider，代码如下：





同时在AndroidMainfest.xml中注册，代码如下：



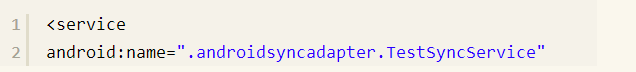
2.关联SyncAdapter通信服务

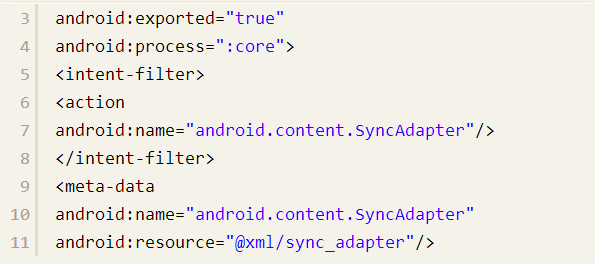
因为SyncAdapter是独立进程，所以要应用实现自己的Binder接口，不过Android系统提供了很简单的抽象类AbstractThreadedSyncAdapter来实现这个AIDL接口，实现进程间通信，代码如下：

|  |
| --- |
| public class TestSyncService extends Service {  private static final Object sSyncAdapterLock = new Object();  private static TestSyncAdapter mSyncAdapter = null;  @Override  public void onCreate() {  synchronized (sSyncAdapterLock) {  if (mSyncAdapter == null) {  mSyncAdapter = new TestSyncAdapter(getApplicationContext(), true);  }  }  }  @Override  public IBinder onBind(Intent intent) {  return mSyncAdapter.getSyncAdapterBinder();  }  static class TestSyncAdapter extends AbstractThreadedSyncAdapter {  public TestSyncAdapter(Context context, boolean autoInitialize) {  super(context, autoInitialize);  }  @Override  public void onPerformSync(Account account, Bundle extras, String authority, ContentProviderClient provider, SyncResult syncResult) {  //具体的同步操作，这里主要是为了提高进程优先级  //getContext().getContentResolver().notifyChange(XXAccountProvider.CONTENT\_URI, null, false);  }  }  } |

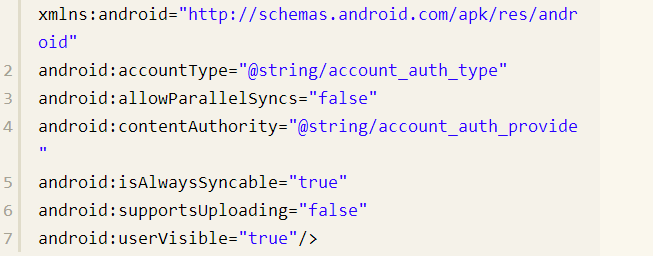
关联SyncAdapter进程后，只需重写AbstractThreadedSyncAdapter中的onPerformSync方法。

当然需要在AndroidMainfest.xml中声明这个服务，需要注意的是，这个服务必须exported，并且必须指定android.content.SyncAdapter为过滤器，这样系统才可以在启动时注册这个服务，代码如下。



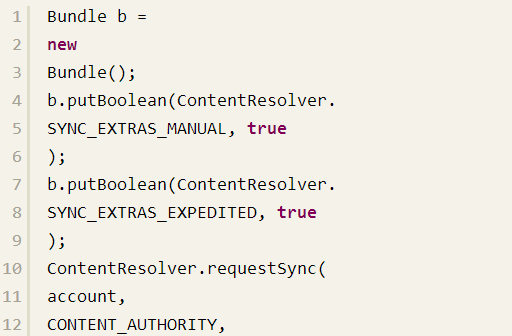






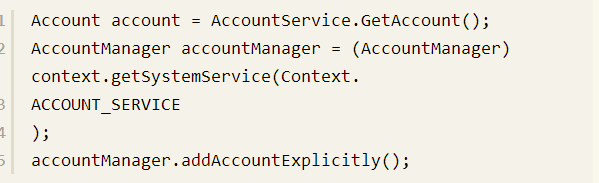
3.使用Sync服务

1）通过ContentProvider实现手动更新，代码如下：



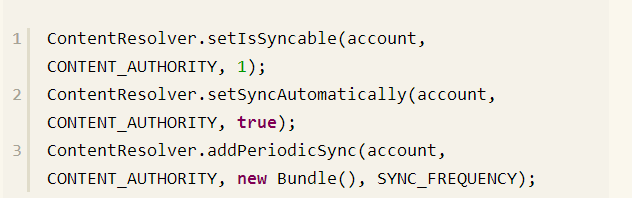


2）添加账号。



建立账号系统可参考官方文档：[https://developer.android.com/training/s ync-adapters/creating-authenticator.html](https://developer.android.com/training/s%20ync-adapters/creating-authenticator.html)

3）设置同步周期。



关联SyncAdapter后，进程的优先级变为1，仅仅低于前台正在运行的进程，因此可以降低应用被系统杀掉的概率。

## 本章小结

提高应用进程存活率最好不要为了提高而提高，如果不遵循系统的管理机制，采用一些“黑科技”的手段提高进程的存活率，会消耗移动设备更多的电量，因此需要谨慎考虑。减少应用内存消耗，合理使用系统管理机制，才是避免进程被系统杀掉最好的方法。

而Crash与ANR，出现的大部分场景不可预知，也不全是必现的问题，特别有些概率性的Crash在测试阶段不一定能发现，所以需要完善的监控机制记录发生问题的点以及有效的日志信息，再修复这类问题。同时在开发阶段提高开发者的技术水平，合理的代码监控与审查机制，将这类问题发生的概率降到最低。

做了稳定性优化，用户不一定能明显地感受到。但如果不做，在用户使用过程中碰到此类问题对体验来说是灾难性的，另外类似的一种优化也有这种情况，做了用户不一定能感受得到，但不做会或多或少带来一定的影响，这个优化就是耗电的优化。下一章，通过分析Android系统最耗电的几大模块，以及相应的优化方法，逐步了解和掌握优化耗电的方案。

## QA

为啥有多个main线程？

超时机制实现原理？跟看门狗有用么？

网上没有找到一个Android死锁了例子？难道有什么保护？

自己的代码加入了主线程

为啥我的死锁代码，分析不出死锁原因？

ui耗时分析还是不是很明确

## REF

[实现ANR原理源码分析](http://www.jianshu.com/p/8964812972be)

[理解Android ANR的触发原理](http://gityuan.com/2016/07/02/android-anr/)