ANR OOM

ANR

概念

ANR(Application Not responding)，是指应用程序未响应，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

场景

- Service Timeout
- BroadcastQueue Timeout
- ContentProvider Timeout
- InputDispatching Timeout

Timeout时长

- 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
- 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;
- ContentProvider超时为CONTENT_PROVIDER_PUBLISH_TIMEOUT = 10s;
- InputDispatching Timeout: 输入事件分发超时5s，包括按键和触摸事件。

> *注意事项: Input的超时机制与其他的不同，对于input来说即便某次事件执行时间超过timeout时长，只要用户后续在没有再生成输入事件，则不会触发ANR*

超时检测机制

1. Service超时检测机制：
    - 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;
2. BroadcastReceiver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;
3. 另外:
    - 对于Service, Broadcast, Input发生ANR之后,最终都会调用AMS.appNotResponding;

- 对于provider,在其进程启动时publish过程可能会出现ANR,则会直接杀进程以及清理相应信息,而不会弹出ANR的对话框

前台与后台ANR【了解下就行】

- 前台ANR：用户能感知，比如拥有前台可见的activity的进程，或者拥有前台通知的fg-service的进程，此时发生ANR对用户体验影响比较大，需要弹框让用户决定是否退出还是等待
- 后台ANR：，只抓取发生无响应进程的trace，也不会收集CPU信息，并且会在后台直接杀掉该无响应的进程，不会弹框提示用户

ANR分析

1. 前台ANR发生后，系统会马上去抓取现场的信息，用于调试分析，收集的信息如下:
    - 将am_anr信息输出到EventLog，也就是说ANR触发的时间点最接近的就是EventLog中输出的am_anr信息
    - 收集以下重要进程的各个线程调用栈trace信息，保存在data/anr/traces.txt文件
        - 当前发生ANR的进程，system_server进程以及所有persistent进程
        - audioserver, cameraserver, mediaserver, surfaceflinger等重要的native进程
        - CPU使用率排名前5的进程
    - 将发生ANR的reason以及CPU使用情况信息输出到main log
    - 将traces文件和CPU使用情况信息保存到dropbox，即data/system/dropbox目录
    - 对用户可感知的进程则弹出ANR对话框告知用户，对用户不可感知的进程发生ANR则直接杀掉
2. 分析步骤
    a. 定位发生ANR时间点
    b. 查看trace信息
    c. 分析是否有耗时的message,binder调用，锁的竞争，CPU资源的抢占
    d. 结合具体的业务场景的上下文来分析

如何避免ANR发生

1. 主线程尽量只做UI相关的操作,避免耗时操作，比如过度复杂的UI绘制，网络操作，文件IO操作；
2. 避免主线程跟工作线程发生锁的竞争，减少系统耗时binder的调用，谨慎使用sharePreference，注意主线程执行provider query操作

> *总之,尽可能减少主线程的负载，让其空闲待命，以期可随时响应用户的操作*

trace.txt文件解读 【重点要看的】

1. 人为的收集trace.txt的命令 `adb shell kill -3 888 //可指定进程pid` 执行完该命令后traces信息的结果保存到文件/data/anr/traces.txt
2. trace文件解读

```
----- pid 888 at 2016-11-11 22:22:22 -----
Cmd line: system_server
ABI: arm
Build type: optimized
Zygote loaded classes=4113 post zygote classes=3239
Intern table: 57550 strong; 9315 weak
JNI: CheckJNI is off; globals=2418 (plus 115 weak)
Libraries: /system/lib/libandroid.so
/system/lib/libandroid_servers.so
/system/lib/libaudioeffect_jni.so
/system/lib/libcompiler_rt.so /system/lib/libjavacrypto.so
/system/lib/libjnigraphics.so /system/lib/libmedia_jni.so
/system/lib/librs_jni.so /system/lib/libsechook.so
/system/lib/libshell_jni.so /system/lib/libsoundpool.so
/system/lib/libwebviewchromium_loader.so
/system/lib/libwifi-service.so
/vendor/lib/libalarmservice_jni.so
/vendor/lib/liblocationservice.so libjavacore.so (16)
//已分配堆内存大小40MB，其中29M已用，总分配207772个对象
Heap: 27% free, 29MB/40MB; 307772 objects
... //省略GC相关信息

//当前进程总99个线程
DALVIK THREADS (99):
//主线程调用栈
"main" prio=5 tid=1 Native
  | group="main" sCount=1 dsCount=0 obj=0x75bd9fb0
self=0x5573d4f770
  | sysTid=12078 nice=-2 cgrp=default sched=0/0
handle=0x7fa75fafe8
  | state=S schedstat=( 5907843636 827600677 5112 )
utm=453 stm=137 core=0 HZ=100
  | stack=0x7fd64ef000-0x7fd64f1000 stackSize=8MB
  | held mutexes=
  //内核栈
  kernel: __switch_to+0x70/0x7c
  kernel: SyS_epoll_wait+0x2a0/0x324
  kernel: SyS_epoll_pwait+0xa4/0x120
  kernel: cpu_switch_to+0x48/0x4c
  native: #00 pc 0000000000069be4 /system/lib64/libc.so
(__epoll_pwait+8)
  native: #01 pc 000000000001cca4 /system/lib64/libc.so
(epoll_pwait+32)
  native: #02 pc 000000000001ad74
/system/lib64/libutils.so
(_ZN7android6Looper9pollInnerEi+144)
```

```
  native: #03 pc 0000000000001b154
/system/lib64/libutils.so
(_ZN7android6Looper8pollOnceEiPiS1_PPv+80)
  native: #04 pc 00000000000d4bc0
/system/lib64/libandroid_runtime.so
(_ZN7android18NativeMessageQueue8pollOnceEP7_JNIEnvP8_jobj
ecti+48)
  native: #05 pc 000000000000082c /data/dalvik-
cache/arm64/system@framework@boot.oat
(Java_android_os_MessageQueue_nativePollOnce__JI+144)
  at android.os.MessageQueue.nativePollOnce(Native method)
  at android.os.MessageQueue.next(MessageQueue.java:323)
  at android.os.Looper.loop(Looper.java:135)
  at
com.android.server.SystemServer.run(SystemServer.java:290)
  at
com.android.server.SystemServer.main(SystemServer.java:175
)
  at java.lang.reflect.Method.invoke!(Native method)
  at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run
(ZygoteInit.java:738)
  at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:62
8)

"Binder_1" prio=5 tid=8 Native
  | group="main" sCount=1 dsCount=0 obj=0x12c610a0
self=0x5573e5c750
  | sysTid=12092 nice=0 cgrp=default sched=0/0
handle=0x7fa2743450
  | state=S schedstat=( 796240075 863170759 3586 ) utm=50
stm=29 core=1 HZ=100
  | stack=0x7fa2647000-0x7fa2649000 stackSize=1013KB
  | held mutexes=
  kernel: __switch_to+0x70/0x7c
  kernel: binder_thread_read+0xd78/0xeb0
  kernel: binder_ioctl_write_read+0x178/0x24c
  kernel: binder_ioctl+0x2b0/0x5e0
  kernel: do_vfs_ioctl+0x4a4/0x578
  kernel: SyS_ioctl+0x5c/0x88
  kernel: cpu_switch_to+0x48/0x4c
  native: #00 pc 0000000000069cd0 /system/lib64/libc.so
(__ioctl+4)
  native: #01 pc 0000000000073cf4 /system/lib64/libc.so
(ioctl+100)
  native: #02 pc 000000000002d6e8
/system/lib64/libbinder.so
(_ZN7android14IPCThreadState14talkWithDriverEb+164)
  native: #03 pc 000000000002df3c
/system/lib64/libbinder.so
(_ZN7android14IPCThreadState20getAndExecuteCommandEv+24)
```

```
  native: #04 pc 000000000002e114
/system/lib64/libbinder.so
(_ZN7android14IPCThreadState14joinThreadPoolEb+124)
  native: #05 pc 0000000000036c38
/system/lib64/libbinder.so (???)
  native: #06 pc 000000000001579c
/system/lib64/libutils.so
(_ZN7android6Thread11_threadLoopEPv+208)
  native: #07 pc 0000000000090598
/system/lib64/libandroid_runtime.so
(_ZN7android14AndroidRuntime15javaThreadShellEPv+96)
  native: #08 pc 0000000000014fec
/system/lib64/libutils.so (???)
  native: #09 pc 0000000000067754 /system/lib64/libc.so
(_ZL15__pthread_startPv+52)
  native: #10 pc 000000000001c644 /system/lib64/libc.so
(__start_thread+16)
  (no managed stack frames)
... //此处省略剩余的N个线程.
```

3. trace参数解读

```
"Binder_1" prio=5 tid=8 Native
  | group="main" sCount=1 dsCount=0 obj=0x12c610a0
self=0x5573e5c750
  | sysTid=12092 nice=0 cgrp=default sched=0/0
handle=0x7fa2743450
  | state=S schedstat=( 796240075 863170759 3586 ) utm=50
stm=29 core=1 HZ=100
  | stack=0x7fa2647000-0x7fa2649000 stackSize=1013KB
  | held mutexes=
```

- 第0行:
    - 线程名: Binder_1（如有daemon则代表守护线程)
    - prio: 线程优先级
    - tid: 线程内部id
    - 线程状态: NATIVE
- 第1行:
    - group: 线程所属的线程组
    - sCount: 线程挂起次数
    - dsCount: 用于调试的线程挂起次数
    - obj: 当前线程关联的java线程对象
    - self: 当前线程地址
- 第2行：
    - sysTid：线程真正意义上的tid
    - nice: 调度有优先级
    - cgrp: 进程所属的进程调度组
    - sched: 调度策略
    - handle: 函数处理地址
- 第3行：

        - state: 线程状态
        - schedstat: CPU调度时间统计
        - utm/stm: 用户态/内核态的CPU时间(单位是jiffies)
        - core: 该线程的最后运行所在核
        - HZ: 时钟频率
    - 第4行：
        - stack：线程栈的地址区间
        - stackSize：栈的大小
    - 第5行：
        - mutex: 所持有mutex类型，有独占锁exclusive和共享锁shared两类
    - schedstat含义说明：
        - nice值越小则优先级越高。此处nice=-2, 可见优先级还是比较高的;
        - schedstat括号中的3个数字依次是Running、Runable、Switch，紧接着的是utm和stm
            - Running时间：CPU运行的时间，单位ns
            - Runable时间：RQ队列的等待时间，单位ns
            - Switch次数：CPU调度切换次数
            - utm: 该线程在用户态所执行的时间，单位是jiffies，jiffies定义为sysconf(_SC_CLK_TCK)，默认等于10ms
            - stm: 该线程在内核态所执行的时间，单位是jiffies，默认等于10ms
    - 可见，该线程Running=186667489018ns,也约等于186667ms。在CPU运行时间包括用户态(utm)和内核态(stm)。 utm + stm = (12112 + 6554) ×10 ms = 186666ms。
    - 结论：utm + stm = schedstat第一个参数值。

OOM

概述

OOM 就是传说中的OutOfMemory，内存占用超出了app的最大允许范围

> *内存溢出与内存泄露要区分清楚，内存溢出了不一定是内存泄露造成的，内存泄露了最终一定会导致内存溢出*

产生原因

- Android中的大部分内存问题归根结底都是Bitmap的问题
- 数组 List Map等集合没及时释放，或者使用的时候没关注当前App剩余可用内存量
- 数据库游标问题
- 构造Adapter时，没有使用缓存的convertView

必须掌握的工具和分析命令

- MAT

- Lint
- Android Studio的Memory Monitor
- ```adb shell dumpsys meminfo -a packageName ````

bitmap的使用 【重点部分】

高效加载大图

1. 读取位图的尺寸与类型

```
BitmapFactory.Options options = new
BitmapFactory.Options();
options.inJustDecodeBounds = true;
BitmapFactory.decodeResource(getResources(), R.id.myimage,
options);
int imageHeight = options.outHeight;
int imageWidth = options.outWidth;
String imageType = options.outMimeType;
```

为了避免java.lang.OutOfMemory 的异常，我们需要在真正解析图片之前检查它的尺寸（除非你能确定这个数据源提供了准确无误的图片且不会导致占用过多的内存）

2. 加载一个按比例缩小的版本到内存中 通过上面的步骤我们已经获取到了图片的尺寸，这些数据可以用来帮助我们决定应该加载整个图片到内存中还是加载一个缩小的版本。有下面一些因素需要考虑：

评估加载完整图片所需要耗费的内存。 程序在加载这张图片时可能涉及到的其他内存需求。 呈现这张图片的控件的尺寸大小。 屏幕大小与当前设备的屏幕密度。 例如，如果把一个大小为1024x768像素的图片显示到大小为128x96像素的ImageView上吗，就没有必要把整张原图都加载到内存中。

为了告诉解码器去加载一个缩小版本的图片到内存中，需要在BitmapFactory.Options 中设置 inSampleSize 的值。例如, 一个分辨率为2048x1536的图片，如果设置 inSampleSize 为4，那么会产出一个大约512x384大小的Bitmap。加载这张缩小的图片仅仅使用大概0.75MB的内存，如果是加载完整尺寸的图片，那么大概需要花费12MB（前提都是Bitmap的配置是ARGB_8888）。下面有一段根据目标图片大小来计算Sample图片大小的代码示例：

```
public static int calculateInSampleSize(
            BitmapFactory.Options options, int reqWidth,
int reqHeight) {
    // Raw height and width of image
    final int height = options.outHeight;
    final int width = options.outWidth;
    int inSampleSize = 1;

    if (height > reqHeight || width > reqWidth) {
```

```
        final int halfHeight = height / 2;
        final int halfWidth = width / 2;

        // Calculate the largest inSampleSize value that
is a power of 2 and keeps both
        // height and width larger than the requested
height and width.
        while ((halfHeight / inSampleSize) > reqHeight
                && (halfWidth / inSampleSize) > reqWidth)
{
            inSampleSize *= 2;
        }
    }

    return inSampleSize;
}
```

为了使用该方法，首先需要设置 inJustDecodeBounds 为 true, 把options的值传递过来，然后设置 inSampleSize 的值并设置 inJustDecodeBounds 为 false，之后重新调用相关的解码方法。

```
public static Bitmap
decodeSampledBitmapFromResource(Resources res, int resId,
        int reqWidth, int reqHeight) {

    // First decode with inJustDecodeBounds=true to check
dimensions
    final BitmapFactory.Options options = new
BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    BitmapFactory.decodeResource(res, resId, options);

    // Calculate inSampleSize
    options.inSampleSize = calculateInSampleSize(options,
reqWidth, reqHeight);

    // Decode bitmap with inSampleSize set
    options.inJustDecodeBounds = false;
    return BitmapFactory.decodeResource(res, resId,
options);
}
```

使用上面这个方法可以简单地加载一张任意大小的图片。如下面的代码样例显示了一个接近 100x100像素的缩略图：

mImageView.setImageBitmap(decodeSampledBitmapFromResource(getResources(), R.id.myimage, 100, 100)); 我们可以通过替换合适的BitmapFactory.decode* 方法来实现一个类似的方法，从其他的数据源解析Bitmap。

非UI线程处理Bitmap

1. 使用AsyncTask

```
class BitmapWorkerTask extends AsyncTask {
    private final WeakReference imageViewReference;
    private int data = 0;

    public BitmapWorkerTask(ImageView imageView) {
        // Use a WeakReference to ensure the ImageView can
be garbage collected
        imageViewReference = new WeakReference(imageView);
    }

    // Decode image in background.
    @Override
    protected Bitmap doInBackground(Integer... params) {
        data = params[0];
        return
decodeSampledBitmapFromResource(getResources(), data, 100,
100));
    }

    // Once complete, see if ImageView is still around and
set bitmap.
    @Override
    protected void onPostExecute(Bitmap bitmap) {
        if (imageViewReference != null && bitmap != null)
{
            final ImageView imageView =
imageViewReference.get();
            if (imageView != null) {
                imageView.setImageBitmap(bitmap);
            }
        }
    }
}
```

为ImageView使用WeakReference确保了AsyncTask所引用的资源可以被垃圾回收器回收。由于当任务结束时不能确保ImageView仍然存在，因此我们必须在onPostExecute()里面对引用进行检查。该ImageView在有些情况下可能已经不存在了，例如，在任务结束之前用户使用了回退操作，或者是配置发生了改变（如旋转屏幕等）。

2. 处理并发问题 通常类似ListView与GridView等视图控件在使用上面演示的AsyncTask 方法时，会同时带来并发的问题。首先为了更高的效率，ListView与GridView的子Item视图会在用户滑动屏幕时被循环使用。如果每一个子视图都触发一个AsyncTask，那么就无法确保关联的视图在结束任务时，分配的视图已经进入循环队列中，给另外一个子视图进行重用。而且， 无法确保所有的异步任务的完成顺序和他们本身的启动顺序保持一致

- 解决方法: mageView保存最近使用的AsyncTask的引用，这个引用可以在任务完成的时候再次读取检查。使用这种方式, 就可以对前面提到的AsyncTask进行扩展

创建一个专用的Drawable的子类来储存任务的引用。在这种情况下，我们使用了一个BitmapDrawable，在任务执行的过程中，一个占位图片会显示在ImageView中:

```
static class AsyncDrawable extends BitmapDrawable {
    private final WeakReference bitmapWorkerTaskReference;

    public AsyncDrawable(Resources res, Bitmap bitmap,
            BitmapWorkerTask bitmapWorkerTask) {
        super(res, bitmap);
        bitmapWorkerTaskReference =
            new WeakReference(bitmapWorkerTask);
    }

    public BitmapWorkerTask getBitmapWorkerTask() {
        return bitmapWorkerTaskReference.get();
    }
}
```

在执行BitmapWorkerTask之前，你需要创建一个AsyncDrawable并且将它绑定到目标控件ImageView中：

```
public void loadBitmap(int resId, ImageView imageView) {
    if (cancelPotentialWork(resId, imageView)) {
        final BitmapWorkerTask task = new
BitmapWorkerTask(imageView);
        final AsyncDrawable asyncDrawable =
                new AsyncDrawable(getResources(),
mPlaceHolderBitmap, task);
        imageView.setImageDrawable(asyncDrawable);
        task.execute(resId);
    }
}
```

在上面的代码示例中，`cancelPotentialWork` 方法检查是否有另一个正在执行的任务与该ImageView关联了起来，如果的确是这样，它通过执行`cancel()`方法来取消另一个任务。在少数情况下, 新创建的任务数据可能会与已经存在的任务相吻合，这样的话就不需要进行下一步动作了。下面是`cancelPotentialWork`方法的实现 。

```
public static boolean cancelPotentialWork(int data,
ImageView imageView) {
    final BitmapWorkerTask bitmapWorkerTask =
getBitmapWorkerTask(imageView);

    if (bitmapWorkerTask != null) {
        final int bitmapData = bitmapWorkerTask.data;
```

```
        if (bitmapData == 0 || bitmapData != data) {
            // Cancel previous task
            bitmapWorkerTask.cancel(true);
        } else {
            // The same work is already in progress
            return false;
        }
    }
    // No task associated with the ImageView, or an
existing task was cancelled
    return true;
}
```

在上面的代码中有一个辅助方法：`getBitmapWorkerTask()`，它被用作检索AsyncTask是否已经被分配到指定的ImageView:

```
private static BitmapWorkerTask
getBitmapWorkerTask(ImageView imageView) {
   if (imageView != null) {
       final Drawable drawable = imageView.getDrawable();
       if (drawable instanceof AsyncDrawable) {
           final AsyncDrawable asyncDrawable =
(AsyncDrawable) drawable;
           return asyncDrawable.getBitmapWorkerTask();
       }
    }
    return null;
}
```

最后一步是在BitmapWorkerTask的`onPostExecute()`方法里面做更新操作:

```
class BitmapWorkerTask extends AsyncTask {
    ...

    @Override
    protected void onPostExecute(Bitmap bitmap) {
        if (isCancelled()) {
            bitmap = null;
        }

        if (imageViewReference != null && bitmap != null)
{
            final ImageView imageView =
imageViewReference.get();
            final BitmapWorkerTask bitmapWorkerTask =
                    getBitmapWorkerTask(imageView);
            if (this == bitmapWorkerTask && imageView !=
null) {
                imageView.setImageBitmap(bitmap);
            }
```

```
        }
    }
}
```

这个方法不仅仅适用于ListView与GridView控件，在那些需要循环利用子视图的控件中同样适用：只需要在设置图片到ImageView的地方调用 `loadBitmap` 方法。例如，在GridView 中实现这个方法可以在 getView()中调用。

缓存Bitmap

将单个Bitmap加载到UI是简单直接的，但是如果我们需要一次性加载大量的图片，事情则会变得复杂起来。在大多数情况下（例如在使用ListView，GridView或ViewPager时），屏幕上的图片和因滑动将要显示的图片的数量通常是没有限制的。

通过循环利用子视图可以缓解内存的使用，垃圾回收器也会释放那些不再需要使用的Bitmap。这些机制都非常好，但是为了保证一个流畅的用户体验，我们希望避免在每次屏幕滑动回来时，都要重复处理那些图片。内存与磁盘缓存通常可以起到辅助作用，允许控件可以快速地重新加载那些处理过的图片。

这一课会介绍在加载多张Bitmap时使用内存缓存与磁盘缓存来提高响应速度与UI流畅度。

- 使用内存缓存(Use a Memory Cache)

内存缓存以花费宝贵的程序内存为前提来快速访问位图。LruCache类（在API Level 4的Support Library中也可以找到）特别适合用来缓存Bitmaps，它使用一个强引用（strong referenced）的LinkedHashMap保存最近引用的对象，并且在缓存超出设置大小的时候剔除（evict）最近最少使用到的对象。

> ***Note:*** *在过去，一种比较流行的内存缓存实现方法是使用软引用（SoftReference）或弱引用（WeakReference）对Bitmap进行缓存，然而我们并不推荐这样的做法。从Android 2.3 (API Level 9)开始，垃圾回收机制变得更加频繁，这使得释放软（弱）引用的频率也随之增高，导致使用引用的效率降低很多。而且在Android 3.0 (API Level 11)之前，备份的Bitmap会存放在Native Memory中，它不是以可预知的方式被释放的，这样可能导致程序超出它的内存限制而崩溃。*

为了给LruCache选择一个合适的大小，需要考虑到下面一些因素：

- 应用剩下了多少可用的内存?
- 多少张图片会同时呈现到屏幕上？有多少图片需要准备好以便马上显示到屏幕？
- 设备的屏幕大小与密度是多少？一个具有特别高密度屏幕（xhdpi）的设备，像Galaxy Nexus会比Nexus S（hdpi）需要一个更大的缓存空间来缓存同样数量的图片。
- Bitmap的尺寸与配置是多少，会花费多少内存？
- 图片被访问的频率如何？是其中一些比另外的访问更加频繁吗？如果是，那么我们可能希望在内存中保存那些最常访问的图片，或者根据访问频率给Bitmap分组，为不同的Bitmap组设置多个LruCache对象。

- 是否可以在缓存图片的质量与数量之间寻找平衡点？某些时候保存大量低质量的Bitmap会非常有用，加载更高质量图片的任务可以交给另外一个后台线程。

通常没有指定的大小或者公式能够适用于所有的情形，我们需要分析实际的使用情况后，提出一个合适的解决方案。缓存太小会导致额外的花销却没有明显的好处，缓存太大同样会导致java.lang.OutOfMemory的异常，并且使得你的程序只留下小部分的内存用来工作（缓存占用太多内存，导致其他操作会因为内存不够而抛出异常）。

下面是一个为Bitmap建立LruCache的示例：

```
private LruCache<String, Bitmap> mMemoryCache;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Get max available VM memory, exceeding this amount
will throw an
    // OutOfMemory exception. Stored in kilobytes as
LruCache takes an
    // int in its constructor.
    final int maxMemory = (int)
(Runtime.getRuntime().maxMemory() / 1024);

    // Use 1/8th of the available memory for this memory
cache.
    final int cacheSize = maxMemory / 8;

    mMemoryCache = new LruCache<String, Bitmap>(cacheSize)
{
        @Override
        protected int sizeOf(String key, Bitmap bitmap) {
            // The cache size will be measured in
kilobytes rather than
            // number of items.
            return bitmap.getByteCount() / 1024;
        }
    };
    ...
}

public void addBitmapToMemoryCache(String key, Bitmap
bitmap) {
    if (getBitmapFromMemCache(key) == null) {
        mMemoryCache.put(key, bitmap);
    }
}

public Bitmap getBitmapFromMemCache(String key) {
    return mMemoryCache.get(key);
}
```

> ***Note:***在上面的例子中, 有1/8的内存空间被用作缓存。 这意味着在常见的设备上（hdpi），最少大概有4MB的缓存空间（32/8）。如果一个填满图片的GridView控件放置在800x480像素的手机屏幕上，大概会花费1.5MB的缓存空间（800x480x4 bytes），因此缓存的容量大概可以缓存2.5页的图片内容。

当加载Bitmap显示到ImageView 之前，会先从LruCache 中检查是否存在这个Bitmap。如果确实存在，它会立即被用来显示到ImageView上，如果没有找到，会触发一个后台线程去处理显示该Bitmap任务。

```
public void loadBitmap(int resId, ImageView imageView) {
    final String imageKey = String.valueOf(resId);

    final Bitmap bitmap = getBitmapFromMemCache(imageKey);
    if (bitmap != null) {
        mImageView.setImageBitmap(bitmap);
    } else {

 mImageView.setImageResource(R.drawable.image_placeholder)
;
        BitmapWorkerTask task = new
BitmapWorkerTask(mImageView);
        task.execute(resId);
    }
}
```

上面的程序中 `BitmapWorkerTask` 需要把解析好的Bitmap添加到内存缓存中：

```
class BitmapWorkerTask extends AsyncTask<Integer, Void,
Bitmap> {
    ...
    // Decode image in background.
    @Override
    protected Bitmap doInBackground(Integer... params) {
        final Bitmap bitmap =
decodeSampledBitmapFromResource(
                getResources(), params[0], 100, 100));
        addBitmapToMemoryCache(String.valueOf(params[0]),
bitmap);
        return bitmap;
    }
    ...
}
```

- 使用磁盘缓存(Use a Disk Cache)

内存缓存能够提高访问最近用过的Bitmap的速度，但是我们无法保证最近访问过的Bitmap都能够保存在缓存中。像类似GridView等需要大量数据填充的控件很容易就会用尽整个内存缓存。另外，我们的应用可能会被类似打电话等行为而暂停并退到后台，因为后台应用可能会被杀死，那么内存缓存就会被销毁，里面

的Bitmap也就不存在了。一旦用户恢复应用的状态，那么应用就需要重新处理那些图片。

磁盘缓存可以用来保存那些已经处理过的Bitmap，它还可以减少那些不再内存缓存中的Bitmap的加载次数。当然从磁盘读取图片会比从内存要慢，而且由于磁盘读取操作时间是不可预期的，读取操作需要在后台线程中处理。

> ***Note:*** *如果图片会被更频繁的访问，使用ContentProvider或许会更加合适，比如在图库应用中。*

这一节的范例代码中使用了一个从Android源码中剥离出来的`DiskLruCache`。改进过的范例代码在已有内存缓存的基础上增加磁盘缓存的功能。

```
private DiskLruCache mDiskLruCache;
private final Object mDiskCacheLock = new Object();
private boolean mDiskCacheStarting = true;
private static final int DISK_CACHE_SIZE = 1024 * 1024 *
10; // 10MB
private static final String DISK_CACHE_SUBDIR =
"thumbnails";

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Initialize memory cache
    ...
    // Initialize disk cache on background thread
    File cacheDir = getDiskCacheDir(this,
DISK_CACHE_SUBDIR);
    new InitDiskCacheTask().execute(cacheDir);
    ...
}

class InitDiskCacheTask extends AsyncTask<File, Void,
Void> {
    @Override
    protected Void doInBackground(File... params) {
        synchronized (mDiskCacheLock) {
            File cacheDir = params[0];
            mDiskLruCache = DiskLruCache.open(cacheDir,
DISK_CACHE_SIZE);
            mDiskCacheStarting = false; // Finished
initialization
            mDiskCacheLock.notifyAll(); // Wake any
waiting threads
        }
        return null;
    }
}

class BitmapWorkerTask extends AsyncTask<Integer, Void,
Bitmap> {
```

```
    ...
    // Decode image in background.
    @Override
    protected Bitmap doInBackground(Integer... params) {
        final String imageKey = String.valueOf(params[0]);

        // Check disk cache in background thread
        Bitmap bitmap = getBitmapFromDiskCache(imageKey);

        if (bitmap == null) { // Not found in disk cache
            // Process as normal
            final Bitmap bitmap =
decodeSampledBitmapFromResource(
                    getResources(), params[0], 100, 100));
        }

        // Add final bitmap to caches
        addBitmapToCache(imageKey, bitmap);

        return bitmap;
    }
    ...
}

public void addBitmapToCache(String key, Bitmap bitmap) {
    // Add to memory cache as before
    if (getBitmapFromMemCache(key) == null) {
        mMemoryCache.put(key, bitmap);
    }

    // Also add to disk cache
    synchronized (mDiskCacheLock) {
        if (mDiskLruCache != null &&
mDiskLruCache.get(key) == null) {
            mDiskLruCache.put(key, bitmap);
        }
    }
}

public Bitmap getBitmapFromDiskCache(String key) {
    synchronized (mDiskCacheLock) {
        // Wait while disk cache is started from
background thread
        while (mDiskCacheStarting) {
            try {
                mDiskCacheLock.wait();
            } catch (InterruptedException e) {}
        }
        if (mDiskLruCache != null) {
            return mDiskLruCache.get(key);
        }
    }
```

```
    return null;
}

// Creates a unique subdirectory of the designated app
cache directory. Tries to use external
// but if not mounted, falls back on internal storage.
public static File getDiskCacheDir(Context context, String
uniqueName) {
    // Check if media is mounted or storage is built-in,
if so, try and use external cache dir
    // otherwise use internal cache dir
    final String cachePath =

 Environment.MEDIA_MOUNTED.equals(Environment.getExternalS
torageState()) ||
                    !isExternalStorageRemovable() ?
getExternalCacheDir(context).getPath() :

 context.getCacheDir().getPath();

    return new File(cachePath + File.separator +
uniqueName);
}
```

> ***Note***:*因为初始化磁盘缓存涉及到I/O操作，所以它不应该在主线程中进行。但是这也意味着在初始化完成之前缓存可以被访问。为了解决这个问题，在上面的实现中，有一个锁对象（lock object）来确保在磁盘缓存完成初始化之前，应用无法对它进行读取。*

内存缓存的检查是可以在UI线程中进行的，磁盘缓存的检查需要在后台线程中处理。磁盘操作永远都不应该在UI线程中发生。当图片处理完成后，Bitmap需要添加到内存缓存与磁盘缓存中，方便之后的使用。

- 处理配置改变(Handle Configuration Changes)

如果运行时设备配置信息发生改变，例如屏幕方向的改变会导致Android中当前显示的Activity先被销毁然后重启。我们需要在配置改变时避免重新处理所有的图片，这样才能提供给用户一个良好的平滑过度的体验。

幸运的是，在前面介绍使用内存缓存的部分，我们已经知道了如何建立内存缓存。这个缓存可以通过调用setRetainInstance(true)保留一个Fragment实例的方法把缓存传递给新的Activity。在这个Activity被重新创建之后，这个保留的Fragment会被重新附着上。这样你就可以访问缓存对象了，从缓存中获取到图片信息并快速的重新显示到ImageView上。

下面是配置改变时使用Fragment来保留LruCache的代码示例：

```
private LruCache<String, Bitmap> mMemoryCache;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
```

```
    RetainFragment retainFragment =

 RetainFragment.findOrCreateRetainFragment(getFragmentMana
ger());
    mMemoryCache = retainFragment.mRetainedCache;
    if (mMemoryCache == null) {
        mMemoryCache = new LruCache<String, Bitmap>
(cacheSize) {
            ... // Initialize cache here as usual
        }
        retainFragment.mRetainedCache = mMemoryCache;
    }
    ...
}

class RetainFragment extends Fragment {
    private static final String TAG = "RetainFragment";
    public LruCache<String, Bitmap> mRetainedCache;

    public RetainFragment() {}

    public static RetainFragment
findOrCreateRetainFragment(FragmentManager fm) {
        RetainFragment fragment = (RetainFragment)
fm.findFragmentByTag(TAG);
        if (fragment == null) {
            fragment = new RetainFragment();
            fm.beginTransaction().add(fragment,
TAG).commit();
        }
        return fragment;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);
    }
}
```

为了测试上面的效果，可以尝试在保留Fragment与没有这样做的情况下旋转屏幕。我们会发现当保留缓存时，从内存缓存中重新绘制几乎没有延迟的现象。内存缓存中没有的图片可能存储在磁盘缓存中。如果两个缓存中都没有，则图像会像平时正常流程一样被处理。

管理Bitmap的内存使用

为了优化垃圾回收机制与Bitmap的重用，我们还有一些特定的事情可以做。 同时根据Android的不同版本，推荐的策略会有所差异。DisplayingBitmaps的示例程序会演示如何设计我们的程序，使得它能够在不同的Android平台上高效地运行.

为了给这节课奠定基础，我们首先要知道Android管理Bitmap内存使用的演变进程:

- 在Android 2.2 (API level 8)以及之前，当垃圾回收发生时，应用的线程是会被暂停的，这会导致一个延迟滞后，并降低系统效率。 **从Android 2.3开始，添加了并发垃圾回收的机制， 这意味着在一个Bitmap不再被引用之后，它所占用的内存会被立即回收。**
- 在Android 2.3.3 (API level 10)以及之前, 一个Bitmap的像素级数据（pixel data）是存放在Native内存空间中的。 这些数据与Bitmap本身是隔离的，Bitmap本身被存放在Dalvik堆中。我们无法预测在Native内存中的像素级数据何时会被释放，这意味着程序容易超过它的内存限制并且崩溃。 **自Android 3.0 (API Level 11)开始， 像素级数据则是与Bitmap本身一起存放在Dalvik堆中。**

下面会介绍如何在不同的Android版本上优化Bitmap内存使用。

- 管理Android 2.3.3及以下版本的内存使用

在Android 2.3.3 (API level 10) 以及更低版本上，推荐使用recycle()方法。 如果在应用中显示了大量的Bitmap数据，我们很可能会遇到OutOfMemoryError的错误。 recycle()方法可以使得程序更快的释放内存。

> ***Caution：*** *只有当我们确定这个Bitmap不再需要用到的时候才应该使用recycle()。在执行recycle()方法之后，如果尝试绘制这个Bitmap， 我们将得到`"Canvas: trying to use a recycled bitmap"`的错误提示。*

下面的代码片段演示了使用`recycle()`的例子。它使用了引用计数的方法（`mDisplayRefCount` 与 `mCacheRefCount`）来追踪一个Bitmap目前是否有被显示或者是在缓存中。并且在下面列举的条件满足时，回收Bitmap：

- `mDisplayRefCount` 与 `mCacheRefCount` 的引用计数均为 0；
- bitmap不为`null`, 并且它还没有被回收。

```
private int mCacheRefCount = 0;
private int mDisplayRefCount = 0;
...
// Notify the drawable that the displayed state has
changed.
// Keep a count to determine when the drawable is no
longer displayed.
public void setIsDisplayed(boolean isDisplayed) {
    synchronized (this) {
        if (isDisplayed) {
            mDisplayRefCount++;
            mHasBeenDisplayed = true;
        } else {
            mDisplayRefCount--;
        }
```

```
    }
    // Check to see if recycle() can be called.
    checkState();
}

// Notify the drawable that the cache state has changed.
// Keep a count to determine when the drawable is no
longer being cached.
public void setIsCached(boolean isCached) {
    synchronized (this) {
        if (isCached) {
            mCacheRefCount++;
        } else {
            mCacheRefCount--;
        }
    }
    // Check to see if recycle() can be called.
    checkState();
}

private synchronized void checkState() {
    // If the drawable cache and display ref counts = 0,
and this drawable
    // has been displayed, then recycle.
    if (mCacheRefCount <= 0 && mDisplayRefCount <= 0 &&
mHasBeenDisplayed
            && hasValidBitmap()) {
        getBitmap().recycle();
    }
}

private synchronized boolean hasValidBitmap() {
    Bitmap bitmap = getBitmap();
    return bitmap != null && !bitmap.isRecycled();
}
```

- 管理Android 3.0及其以上版本的内存

从Android 3.0 (API Level 11)开始，引进了BitmapFactory.Options.inBitmap字段。 如果使用了这个设置字段，decode方法会在加载Bitmap数据的时候去重用已经存在的Bitmap。这意味着Bitmap的内存是被重新利用的，这样可以提升性能，并且减少了内存的分配与回收。然而，使用inBitmap有一些限制，特别是在Android 4.4 (API level 19)之前，只有同等大小的位图才可以被重用。详情请查看inBitmap文档。

- 保存Bitmap供以后使用

下面演示了如何将一个已经存在的Bitmap存放起来以便后续使用。当一个应用运行在Android 3.0或者更高的平台上并且Bitmap从LruCache中移除时，Bitmap的一个软引用会被存放在Hashset中，这样便于之后可能被inBitmap重用：

```
Set<SoftReference<Bitmap>> mReusableBitmaps;
```

```
private LruCache<String, BitmapDrawable> mMemoryCache;

// If you're running on Honeycomb or newer, create a
// synchronized HashSet of references to reusable bitmaps.
if (Utils.hasHoneycomb()) {
    mReusableBitmaps =
            Collections.synchronizedSet(new
HashSet<SoftReference<Bitmap>>());
}

mMemoryCache = new LruCache<String, BitmapDrawable>
(mCacheParams.memCacheSize) {

    // Notify the removed entry that is no longer being
cached.
    @Override
    protected void entryRemoved(boolean evicted, String
key,
            BitmapDrawable oldValue, BitmapDrawable
newValue) {
        if
(RecyclingBitmapDrawable.class.isInstance(oldValue)) {
            // The removed entry is a recycling drawable,
so notify it
            // that it has been removed from the memory
cache.
            ((RecyclingBitmapDrawable)
oldValue).setIsCached(false);
        } else {
            // The removed entry is a standard
BitmapDrawable.
            if (Utils.hasHoneycomb()) {
                // We're running on Honeycomb or later, so
add the bitmap
                // to a SoftReference set for possible use
with inBitmap later.
                mReusableBitmaps.add
                        (new SoftReference<Bitmap>
(oldValue.getBitmap()));
            }
        }
    }
....
}
```

- 使用已经存在的Bitmap

在运行的程序中，decode方法会检查看是否存在可重用的Bitmap。 例如:

```
public static Bitmap decodeSampledBitmapFromFile(String
filename,
```

```
        int reqWidth, int reqHeight, ImageCache cache) {

    final BitmapFactory.Options options = new
BitmapFactory.Options();
    ...
    BitmapFactory.decodeFile(filename, options);
    ...

    // If we're running on Honeycomb or newer, try to use
inBitmap.
    if (Utils.hasHoneycomb()) {
        addInBitmapOptions(options, cache);
    }
    ...
    return BitmapFactory.decodeFile(filename, options);
}
```

下面的代码是上述代码片段中，`addInBitmapOptions()`方法的具体实现。 它会为inBitmap查找一个已经存在的Bitmap，并将它设置为inBitmap的值。 注意这个方法只有在找到合适且可重用的Bitmap时才会赋值给inBitmap（我们需要在赋值之前进行检查）：

```
private static void
addInBitmapOptions(BitmapFactory.Options options,
        ImageCache cache) {
    // inBitmap only works with mutable bitmaps, so force
the decoder to
    // return mutable bitmaps.
    options.inMutable = true;

    if (cache != null) {
        // Try to find a bitmap to use for inBitmap.
        Bitmap inBitmap =
cache.getBitmapFromReusableSet(options);

        if (inBitmap != null) {
            // If a suitable bitmap has been found, set it
as the value of
            // inBitmap.
            options.inBitmap = inBitmap;
        }
    }
}

// This method iterates through the reusable bitmaps,
looking for one
// to use for inBitmap:
protected Bitmap
getBitmapFromReusableSet(BitmapFactory.Options options) {
        Bitmap bitmap = null;
```

```
    if (mReusableBitmaps != null &&
!mReusableBitmaps.isEmpty()) {
        synchronized (mReusableBitmaps) {
            final Iterator<SoftReference<Bitmap>> iterator
                    = mReusableBitmaps.iterator();
            Bitmap item;

            while (iterator.hasNext()) {
                item = iterator.next().get();

                if (null != item && item.isMutable()) {
                    // Check to see it the item can be
used for inBitmap.
                    if (canUseForInBitmap(item, options))
{
                        bitmap = item;

                        // Remove from reusable set so it
can't be used again.
                        iterator.remove();
                        break;
                    }
                } else {
                    // Remove from the set if the
reference has been cleared.
                    iterator.remove();
                }
            }
        }
    }
    return bitmap;
}
```

最后，下面这个方法判断候选Bitmap是否满足inBitmap的大小条件:

```
static boolean canUseForInBitmap(
        Bitmap candidate, BitmapFactory.Options
targetOptions) {

    if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.KITKAT) {
        // From Android 4.4 (KitKat) onward we can re-use
if the byte size of
        // the new bitmap is smaller than the reusable
bitmap candidate
        // allocation byte count.
        int width = targetOptions.outWidth /
targetOptions.inSampleSize;
        int height = targetOptions.outHeight /
targetOptions.inSampleSize;
```

```
        int byteCount = width * height *
getBytesPerPixel(candidate.getConfig());
        return byteCount <=
candidate.getAllocationByteCount();
    }

    // On earlier versions, the dimensions must match
exactly and the inSampleSize must be 1
    return candidate.getWidth() == targetOptions.outWidth
            && candidate.getHeight() ==
targetOptions.outHeight
            && targetOptions.inSampleSize == 1;
}

/**
 * A helper function to return the byte usage per pixel of
a bitmap based on its configuration.
 */
static int getBytesPerPixel(Config config) {
    if (config == Config.ARGB_8888) {
        return 4;
    } else if (config == Config.RGB_565) {
        return 2;
    } else if (config == Config.ARGB_4444) {
        return 2;
    } else if (config == Config.ALPHA_8) {
        return 1;
    }
    return 1;
}
```

在UI上显示Bitmap

- 实现加载图片到ViewPager

Swipe View Pattern是一个使用滑动来切换显示不同详情页面的设计模型。（关于这种效果请先参看Android Design: Swipe Views）。我们可以通过PagerAdapter与ViewPager控件来实现这个效果。 不过，一个更加合适的Adapter是PagerAdapter的一个子类，叫做FragmentStatePagerAdapter：它可以在某个ViewPager中的子视图切换出屏幕时自动销毁与保存Fragments的状态。这样能够保持更少的内存消耗。

> ***Note:*** *如果只有为数不多的图片并且确保不会超出程序内存限制，那么使用PagerAdapter或FragmentPagerAdapter会更加合适。*

下面是一个使用ViewPager与ImageView作为子视图的示例。主Activity包含有ViewPager和Adapter。

```
public class ImageDetailActivity extends FragmentActivity
{
```

```
    public static final String EXTRA_IMAGE =
"extra_image";

    private ImagePagerAdapter mAdapter;
    private ViewPager mPager;

    // A static dataset to back the ViewPager adapter
    public final static Integer[] imageResIds = new
Integer[] {
            R.drawable.sample_image_1,
R.drawable.sample_image_2, R.drawable.sample_image_3,
            R.drawable.sample_image_4,
R.drawable.sample_image_5, R.drawable.sample_image_6,
            R.drawable.sample_image_7,
R.drawable.sample_image_8, R.drawable.sample_image_9};

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.image_detail_pager); //
Contains just a ViewPager

        mAdapter = new
ImagePagerAdapter(getSupportFragmentManager(),
imageResIds.length);
        mPager = (ViewPager) findViewById(R.id.pager);
        mPager.setAdapter(mAdapter);
    }

    public static class ImagePagerAdapter extends
FragmentStatePagerAdapter {
        private final int mSize;

        public ImagePagerAdapter(FragmentManager fm, int
size) {
            super(fm);
            mSize = size;
        }

        @Override
        public int getCount() {
            return mSize;
        }

        @Override
        public Fragment getItem(int position) {
            return
ImageDetailFragment.newInstance(position);
        }
    }
}
```

Fragment里面包含了ImageView控件:

```
public class ImageDetailFragment extends Fragment {
    private static final String IMAGE_DATA_EXTRA =
"resId";
    private int mImageNum;
    private ImageView mImageView;

    static ImageDetailFragment newInstance(int imageNum) {
        final ImageDetailFragment f = new
ImageDetailFragment();
        final Bundle args = new Bundle();
        args.putInt(IMAGE_DATA_EXTRA, imageNum);
        f.setArguments(args);
        return f;
    }

    // Empty constructor, required as per Fragment docs
    public ImageDetailFragment() {}

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mImageNum = getArguments() != null ?
getArguments().getInt(IMAGE_DATA_EXTRA) : -1;
    }

    @Override
    public View onCreateView(LayoutInflater inflater,
ViewGroup container,
            Bundle savedInstanceState) {
        // image_detail_fragment.xml contains just an
ImageView
        final View v =
inflater.inflate(R.layout.image_detail_fragment,
container, false);
        mImageView = (ImageView)
v.findViewById(R.id.imageView);
        return v;
    }

    @Override
    public void onActivityCreated(Bundle
savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        final int resId =
ImageDetailActivity.imageResIds[mImageNum];
        mImageView.setImageResource(resId); // Load image
into ImageView
    }
}
```

希望你有发现上面示例存在的问题：在UI线程中读取图片可能会导致应用无响应。因此使用在第二课中学习的AsyncTask会更好。

```
public class ImageDetailActivity extends FragmentActivity
{
    ...

    public void loadBitmap(int resId, ImageView imageView)
{

 mImageView.setImageResource(R.drawable.image_placeholder)
;
        BitmapWorkerTask task = new
BitmapWorkerTask(mImageView);
        task.execute(resId);
    }

    ... // include BitmapWorkerTask class
}

public class ImageDetailFragment extends Fragment {
    ...

    @Override
    public void onActivityCreated(Bundle
savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        if
(ImageDetailActivity.class.isInstance(getActivity())) {
            final int resId =
ImageDetailActivity.imageResIds[mImageNum];
            // Call out to ImageDetailActivity to load the
bitmap in a background thread
            ((ImageDetailActivity)
getActivity()).loadBitmap(resId, mImageView);
        }
    }
}
```

在BitmapWorkerTask中做一些例如重设图片大小，从网络拉取图片的任务，可以确保不会阻塞UI线程。如果后台线程不仅仅是一个简单的加载操作，增加一个内存缓存或者磁盘缓存会比较好（请参考第三课：缓存Bitmap），下面是一些为了内存缓存而附加的内容：

```
public class ImageDetailActivity extends FragmentActivity
{
    ...
    private LruCache mMemoryCache;

    @Override
    public void onCreate(Bundle savedInstanceState) {
```

```
        ...
        // initialize LruCache as per Use a Memory Cache
section
    }

    public void loadBitmap(int resId, ImageView imageView)
{
        final String imageKey = String.valueOf(resId);

        final Bitmap bitmap = mMemoryCache.get(imageKey);
        if (bitmap != null) {
            mImageView.setImageBitmap(bitmap);
        } else {

 mImageView.setImageResource(R.drawable.image_placeholder)
;
            BitmapWorkerTask task = new
BitmapWorkerTask(mImageView);
            task.execute(resId);
        }
    }

    ... // include updated BitmapWorkerTask from Use a
Memory Cache section
}
```

把前面学习到的所有技巧合并起来，我们将得到一个响应性良好的ViewPager实现：它拥有最小的加载延迟，同时可以根据实际需求执行不同的后台处理任务。

- 实现加载图片到GridView

Grid List Building Block是一种有效显示大量图片的方式。它能够一次显示许多图片，同时即将被显示的图片会处于准备显示的状态。如果我们想要实现这种效果，必须确保UI是流畅的，能够控制内存使用，并且正确处理并发问题（因为GridView会循环使用子视图)。

下面是一个典型的使用场景，在Fragment里面内置GridView，其中GridView的子视图是ImageView：

```
public class ImageGridFragment extends Fragment implements
AdapterView.OnItemClickListener {
    private ImageAdapter mAdapter;

    // A static dataset to back the GridView adapter
    public final static Integer[] imageResIds = new
Integer[] {
            R.drawable.sample_image_1,
R.drawable.sample_image_2, R.drawable.sample_image_3,
            R.drawable.sample_image_4,
R.drawable.sample_image_5, R.drawable.sample_image_6,
            R.drawable.sample_image_7,
R.drawable.sample_image_8, R.drawable.sample_image_9};
```

```
    // Empty constructor as per Fragment docs
    public ImageGridFragment() {}

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mAdapter = new ImageAdapter(getActivity());
    }

    @Override
    public View onCreateView(
            LayoutInflater inflater, ViewGroup container,
Bundle savedInstanceState) {
        final View v =
inflater.inflate(R.layout.image_grid_fragment, container,
false);
        final GridView mGridView = (GridView)
v.findViewById(R.id.gridView);
        mGridView.setAdapter(mAdapter);
        mGridView.setOnItemClickListener(this);
        return v;
    }

    @Override
    public void onItemClick(AdapterView parent, View v,
int position, long id) {
        final Intent i = new Intent(getActivity(),
ImageDetailActivity.class);
        i.putExtra(ImageDetailActivity.EXTRA_IMAGE,
position);
        startActivity(i);
    }

    private class ImageAdapter extends BaseAdapter {
        private final Context mContext;

        public ImageAdapter(Context context) {
            super();
            mContext = context;
        }

        @Override
        public int getCount() {
            return imageResIds.length;
        }

        @Override
        public Object getItem(int position) {
            return imageResIds[position];
        }
```

```
        @Override
        public long getItemId(int position) {
            return position;
        }

        @Override
        public View getView(int position, View
convertView, ViewGroup container) {
            ImageView imageView;
            if (convertView == null) { // if it's not
recycled, initialize some attributes
                imageView = new ImageView(mContext);

 imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
                imageView.setLayoutParams(new
GridView.LayoutParams(
                        LayoutParams.MATCH_PARENT,
LayoutParams.MATCH_PARENT));
            } else {
                imageView = (ImageView) convertView;
            }
        //请注意下面的代码
        imageView.setImageResource(imageResIds[position]);
// Load image into ImageView
        return imageView;
    }
}
```

这里同样有一个问题，上面的代码实现中，犯了把图片加载放在UI线程进行处理的错误。如果只是加载一些很小的图片，或者是经过Android系统缩放并缓存过的图片，上面的代码在运行时不会有太大问题，但是如果加载的图片稍微复杂耗时一点，这都会导致你的UI卡顿甚至应用无响应。

与前面加载图片到ViewPager一样，如果`setImageResource`的操作会比较耗时，也有可能会阻塞UI线程。不过我们可以使用类似前面异步处理图片与增加缓存的方法来解决这个问题。然而，我们还需要考虑GridView的循环机制所带来的并发问题。为了处理这个问题，可以参考前面的课程 。下面是一个更新过后的解决方案：

```
public class ImageGridFragment extends Fragment implements
AdapterView.OnItemClickListener {
    ...

    private class ImageAdapter extends BaseAdapter {
        ...

        @Override
        public View getView(int position, View
convertView, ViewGroup container) {
            ...
            loadBitmap(imageResIds[position], imageView)
```

```
            return imageView;
        }
    }

    public void loadBitmap(int resId, ImageView imageView)
{
        if (cancelPotentialWork(resId, imageView)) {
            final BitmapWorkerTask task = new
BitmapWorkerTask(imageView);
            final AsyncDrawable asyncDrawable =
                    new AsyncDrawable(getResources(),
mPlaceHolderBitmap, task);
            imageView.setImageDrawable(asyncDrawable);
            task.execute(resId);
        }
    }

    static class AsyncDrawable extends BitmapDrawable {
        private final WeakReference
bitmapWorkerTaskReference;

        public AsyncDrawable(Resources res, Bitmap bitmap,
                BitmapWorkerTask bitmapWorkerTask) {
            super(res, bitmap);
            bitmapWorkerTaskReference =
                new WeakReference(bitmapWorkerTask);
        }

        public BitmapWorkerTask getBitmapWorkerTask() {
            return bitmapWorkerTaskReference.get();
        }
    }

    public static boolean cancelPotentialWork(int data,
ImageView imageView) {
        final BitmapWorkerTask bitmapWorkerTask =
getBitmapWorkerTask(imageView);

        if (bitmapWorkerTask != null) {
            final int bitmapData = bitmapWorkerTask.data;
            if (bitmapData != data) {
                // Cancel previous task
                bitmapWorkerTask.cancel(true);
            } else {
                // The same work is already in progress
                return false;
            }
        }
        // No task associated with the ImageView, or an
existing task was cancelled
        return true;
    }
```

```
    private static BitmapWorkerTask
getBitmapWorkerTask(ImageView imageView) {
       if (imageView != null) {
           final Drawable drawable =
imageView.getDrawable();
           if (drawable instanceof AsyncDrawable) {
               final AsyncDrawable asyncDrawable =
(AsyncDrawable) drawable;
               return asyncDrawable.getBitmapWorkerTask();
           }
        }
        return null;
    }

    ... // include updated BitmapWorkerTask class
```

> ***Note:***对于ListView同样可以套用上面的方法。

上面的方法提供了足够的弹性，使得我们可以做从网络下载图片，并对大尺寸大的数码照片做缩放等操作而不至于阻塞UI线程。