1、目标

> Learn how to use common techniques to process and load `Bitmap` objects in a way that keeps your user interface (UI) components responsive and avoids exceeding your application memory limit. If you're not careful, bitmaps can quickly consume your available memory budget leading to an application crash due to the dreaded exception: `java.lang.OutofMemoryError: bitmap size exceeds VM budget` .

- Loading Large Bitmaps Efficiently

> Given that you are working with limited memory, ideally you only want to load a lower resolution version in memory. The lower resolution version should match the size of the UI component that displays it. An image with a higher resolution does not provide any visible benefit, but still takes up precious memory and incurs additional performance overhead due to additional on the fly scaling.

Read Bitmap Dimensions and Type

```
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;
```

Load a Scaled Down Version into Memory

- Estimated memory usage of loading the full image in memory.
- Amount of memory you are willing to commit to loading this image given any other memory requirements of your application.
- Dimensions of the target `ImageView` or UI component that the image is to be loaded into.
- Screen size and density of the current device.

To tell the decoder to subsample the image, loading a smaller version into memory, set `inSampleSize` to `true` in your `BitmapFactory.Options` object. For example, an image with resolution 2048x1536 that is decoded with an `inSampleSize` of 4 produces a bitmap of approximately 512x384. Loading this into memory uses 0.75MB rather than 12MB for the full image (assuming a bitmap configuration of `ARGB_8888`). Here's a method to calculate a sample size value that is a power of two based on a target width and height:

```
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);
}
```

```
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;
}
```

- **Processing Bitmaps Off the UI Thread**

The `BitmapFactory.decode*` methods, discussed in the Load Large Bitmaps Efficiently lesson, should not be executed on the main UI thread if the source data is read from disk or a network location (or really any source other than memory).

AsyncTask

The `AsyncTask` class provides an easy way to execute some work in a background thread and publish the results back on the UI thread. To use it, create a subclass and override the provided methods. Here's an example of loading a large image into an `ImageView` using `AsyncTask` and `decodeSampledBitmapFromResource()`:

才有异步任务类来进行引用 到时候，为何要用弱引用呢？因为很可能到时候，这个ImageView的本体已经消失

了，这样在设置的时候，就没有必要，也不可以执行下去了！

```
class BitmapWorkerTask extends AsyncTask<Integer, Void, Bitmap> {
    private final WeakReference<ImageView> imageViewReference;
    private int data = 0;

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

Concurrency

Common view components such as `ListView` and `GridView` introduce another issue when used in conjunction with the `AsyncTask` as demonstrated in the previous section. In order to be efficient with memory, these components recycle child views as the user scrolls. If each child view triggers an `AsyncTask`, there is no guarantee that when it completes, the associated view has not already been recycled for use in another child view. Furthermore, there is no guarantee that the order in which asynchronous tasks are started is the order that they complete.

The blog post Multithreading for Performance further discusses dealing with concurrency, and offers a solution where the `ImageView` stores a reference to the most recent `AsyncTask` which can later be checked when the task completes. Using a similar method, the `AsyncTask` from the previous section can be extended to follow a similar pattern.

- **Caching Bitmaps**

Use a Memory Cache

A memory cache offers fast access to bitmaps at the cost of taking up valuable application memory. The `LruCache` class (also available in the Support Library for use back to API Level 4) is particularly well suited to the task of caching bitmaps, keeping recently referenced objects in a strong referenced `LinkedHashMap` and evicting the least recently used member before the cache exceeds its designated size.

ConcurrentHashMap不同于`LinkedHashMap` ConcurrentHashMap实现了并发的访问限制。

Note: In the past, a popular memory cache implementation was a `SoftReference` or `WeakReference` bitmap cache, however this is not recommended. Starting from Android 2.3 (API Level 9) the garbage collector is more aggressive with collecting soft/weak references which makes them fairly ineffective. In addition, prior to Android 3.0 (API Level 11), the backing data of a bitmap was stored in native memory which is not released in a predictable manner, potentially causing an application to briefly exceed its memory limits and crash.

SIZE

- How memory intensive is the rest of your activity and/or application?
- How many images will be on-screen at once? How many need to be available ready to come on-screen?
- What is the screen size and density of the device? An extra high density screen (xhdpi) device like Galaxy Nexus will need a larger cache to hold the same number of images in memory compared to a device likeNexus S (hdpi).
- What dimensions and configuration are the bitmaps and therefore how much memory will each take up?
- How frequently will the images be accessed? Will some be accessed more frequently than others? If so, perhaps you may want to keep certain items always in memory or even have multiple `LruCache` objects for different groups of bitmaps.
- Can you balance quality against quantity? Sometimes it can be more useful to store a larger number of lower quality bitmaps, potentially loading a higher quality version in another background task.

Load Bitmaps into a ViewPager Implementation

The swipe view pattern is an excellent way to navigate the detail view of an image gallery. You can implement this pattern using a `ViewPager` component backed by a `PagerAdapter`. However, a more suitable backing adapter is the subclass `FragmentStatePagerAdapter` which automatically destroys and saves state of the `Fragments` in the `ViewPager` as they disappear off-screen, keeping memory usage down.

Note: If you have a smaller number of images and are confident they all fit within the application memory limit, then using a regular `PagerAdapter` or `FragmentPagerAdapter` might be more appropriate.

Load Bitmaps into a GridView Implementation

The grid list building block is useful for showing image data sets and can be implemented using a `GridView` component in which many images can be on-screen at any one time and many more need to be ready to appear if the user scrolls up or down. When implementing this type of control, you must ensure the UI remains fluid, memory usage remains under control and concurrency is handled correctly (due to the way `GridView` recycles its children views).