Managing Bitmap Memory

In addition to the steps described in Caching Bitmaps (cache-bitmap.html), there are specific things you can do to facilitate garbage collection and bitmap reuse. The recommended strategy depends on which version(s) of Android you are targeting. The `BitmapFun` sample app included with this class shows you how to design your app to work efficiently across different versions of Android.

To set the stage for this lesson, here is how Android's management of bitmap memory has evolved:

- On Android Android 2.2 (API level 8) and lower, when garbage collection occurs, your app's threads get stopped. This causes a lag that can degrade performance. **Android 2.3 adds concurrent garbage collection, which means that the memory is reclaimed soon after a bitmap is no longer referenced.**
- On Android 2.3.3 (API level 10) and lower, the backing pixel data for a bitmap is stored in native memory. It is separate from the bitmap itself, which is stored in the Dalvik heap. The pixel data in native memory is not released in a predictable manner, potentially causing an application to briefly exceed its memory limits and crash. **As of Android 3.0 (API level 11), the pixel data is stored on the Dalvik heap along with the associated bitmap.**

The following sections describe how to optimize bitmap memory management for different Android versions.

THIS LESSON TEACHES YOU TO

1. Manage Memory on Android 2.3.3 and Lower
2. Manage Memory on Android 3.0 and Higher

YOU SHOULD ALSO READ

- Memory Analysis for Android Applications blog post
- Memory management for Android Apps Google I/O presentation
- Android Design: Swipe Views
- Android Design: Grid Lists

TRY IT OUT

Download the sample

DisplayingBitmaps.zip

Manage Memory on Android 2.3.3 and Lower

On Android 2.3.3 (API level 10) and lower, using `recycle()` (../../reference/android/graphics/Bitmap.html#recycle()) is recommended. If you're displaying large amounts of bitmap data in your app, you're likely to run into `OutOfMemoryError` (../../reference/java/lang/OutOfMemoryError.html) errors. The `recycle()` (../../reference/android/graphics/Bitmap.html#recycle()) method allows an app to reclaim memory as soon as possible.

> **Caution:** You should use `recycle()` (../../reference/android/graphics/Bitmap.html#recycle()) only when you are sure that the bitmap is no longer being used. If you call `recycle()` (../../reference/android/graphics/Bitmap.html#recycle()) and later attempt to draw the bitmap, you will get the error: `"Canvas: trying to use a recycled bitmap"`.

The following code snippet gives an example of calling `recycle()` (../../reference/android/graphics/Bitmap.html#recycle()). It uses reference counting (in the variables `mDisplayRefCount` and `mCacheRefCount`) to track whether a bitmap is currently being displayed or in the cache. The code recycles the bitmap when these conditions are met:

- The reference count for both `mDisplayRefCount` and `mCacheRefCount` is 0.
- The bitmap is not `null`, and it hasn't been recycled yet.

```
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();
}
```

Manage Memory on Android 3.0 and Higher

Android 3.0 (API level 11) introduces the `BitmapFactory.Options.inBitmap` (../../reference/android/graphics/BitmapFactory.Options.html#inBitmap) field. If this option is set, decode methods that take the `Options` (../../reference/android/graphics/BitmapFactory.Options.html) object will attempt to reuse an existing bitmap when loading content. This means that the bitmap's memory is reused, resulting in improved performance, and removing both memory allocation and de-allocation. However, there are

certain restrictions with how `inBitmap` can be used. In particular, before Android 4.4 (API level 19), only equal sized bitmaps are supported. For details, please see the `inBitmap` documentation.

Save a bitmap for later use

The following snippet demonstrates how an existing bitmap is stored for possible later use in the sample app. When an app is running on Android 3.0 or higher and a bitmap is evicted from the `LruCache`, a soft reference to the bitmap is placed in a `HashSet`, for possible reuse later with `inBitmap`:

```
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()));
            }
        }
    }
....
}
```

Use an existing bitmap

In the running app, decoder methods check to see if there is an existing bitmap they can use. For example:

```
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);
}
```

The next snippet shows the `addInBitmapOptions()` method that is called in the above snippet. It looks for an existing bitmap to set as the value for `inBitmap`. Note that this method only sets a value for `inBitmap` if it finds a suitable match (your code should never assume that a match will be found):

```
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;
}
```

Finally, this method determines whether a candidate bitmap satisfies the size criteria to be used for `inBitmap` (see reference/android/graphics/BitmapFactory.Options.html#inBitmap):

```
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
    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
 */
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;
}
```