第 27 章

27 Looper、Handler与HandlerThread

从Flickr下载并解析XML后，接下来的任务就是下载并显示图片。本章，为实现该任务，我们将学习如何使用Looper、Handler与HandlerThread。

27.1 设置 GridView 以显示图片

为协助GridView显示内容，当前PhotoGalleryFragment中的adapter仅提供了TextView。每个TextView显示一张图片的文字说明。

而要显示图片，我们需要的是能提供ImageView的定制adapter。然后，通过它，最终实现每个ImageView都显示一张下载自GalleryItem 的mUrl地址的图片。

首先，为gallery图片项创建一个名为gallery_item.xml的布局文件。该布局将包含一个ImageView组件，如图27-1所示。

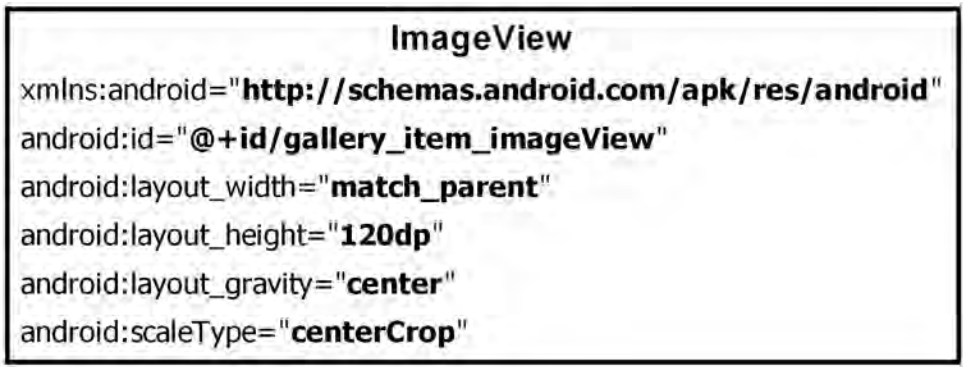

图27-1　Gallery图片项布局（res/layout/gallery_item.xml）

显示图片的ImageView由GridView负责管理，这意味着其宽度会变动，而高度会保持固定不变。为最大化利用ImageView的空间，我们已设置它的scaleType属性值为centerCrop。该设置居中放置图片，然后进行放大，也就是说放大较小图片，裁剪较大图片以匹配视图。

接下来，需为每个ImageView设置初始占位图片，等成功下载图片后再对其进行替换。在随书代码文件中找到brian_up_close.jpg，并复制到项目的res/drawable-hdpi目录中。

在PhotoGalleryFragment类中，以定制ArrayAdapter替换基本ArrayAdapter，该定制ArrayAdapter的getView(...)方法会返回一个显示初始占位图片的ImagerView，如代码清单27-1所示。

代码清单27-1　创建GalleryItemAdapter（PhotoGalleryFragment.java）

```
public class PhotoGalleryFragment extends Fragment {

    ...

    void setupAdapter() {
        if (getActivity() == null || mGridView == null) return;

        if (mItems != null) {
            mGridView.setAdapter(new ArrayAdapter<GalleryItem>(getActivity(),
                    android.R.layout.simple_gallery_item, mItems));
            mGridView.setAdapter(new GalleryItemAdapter(mItems));
        } else {
            mGridView.setAdapter(null);
        }
    }

    private class FetchItemsTask extends AsyncTask<Void,Void,ArrayList<GalleryItem>> {
        ...
    }

    private class GalleryItemAdapter extends ArrayAdapter<GalleryItem> {
        public GalleryItemAdapter(ArrayList<GalleryItem> items) {
            super(getActivity(), 0, items);
        }

        @Override
        public View getView(int position, View convertView, ViewGroup parent) {
            if (convertView == null) {
                convertView = getActivity().getLayoutInflater()
                        .inflate(R.layout.gallery_item, parent, false);
            }

            ImageView imageView = (ImageView)convertView
                    .findViewById(R.id.gallery_item_imageView);
            imageView.setImageResource(R.drawable.brian_up_close);

            return convertView;
        }
    }
}
```

记住，AdapterView（这里指GridView）会为每一个所需视图调用其adapter的getView(...)方法，如图27-2所示。

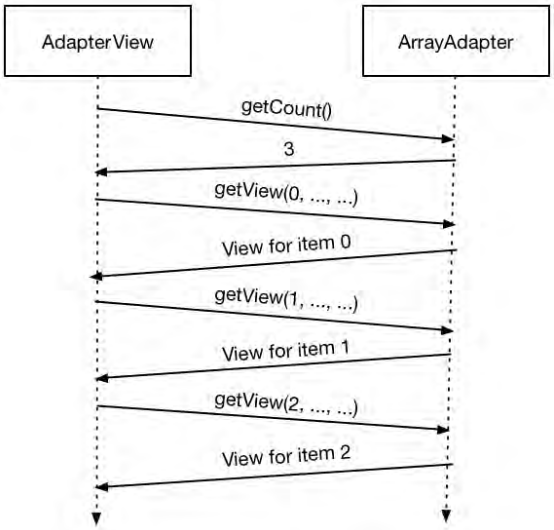

图27-2 AdapterView与ArrayAdapter的互动

运行PhotoGallery应用，欣赏Brian的一组大头照，如图27-3所示。

图27-3 满屏的Brian大头照

27.2　批量下载缩略图

当前，PhotoGallery应用的网络使用部分工作方式如下：PhotoGalleryFragment执行一个AsyncTask，该AsyncTask在后台线程上从Flickr获取XML数据，然后解析XML并将解析结果存放到GalleryItem数组中。最终每个GalleryItem都带有一个指向某张缩略图的URL。

接下来是下载那些URL指向的缩略图。是不是认为只要在FetchItemsTask的doInBackground()方法中添加一些网络下载相关代码就行了？GalleryItem数组含有100个URL下载链接。我们一次下载一张，直到完成全部100张的下载。最后，等onPostExecute(...)方法执行完毕，所有下载的图片一下全部显示在GridView视图中。

然而，一次性下载全部缩略图存在两个问题。首先，下载比较耗时，而且在下载完成前，UI都无法完成更新。这样，网速较慢时，用户就只能长时间盯着Brian的照片墙。

其次，缩略图的保存也是个问题。100张缩略图保存在内存中固然轻松，但如果是1000张呢？如果还需要实现无限滚动来显示图片呢？显然，这样会耗尽内存。

由于此类问题的存在，实际开发的应用通常会选择仅在需要显示图片时才去下载。显然，GridView及其adapter应负责实现按需下载。作为getView(...)方法实现代码的一部分，adapter将触发图片的下载。

AsyncTask是获得后台线程的最简单方式，但它基本上不适用于重复且长时间运行的任务。（可阅读本章末尾的深入学习部分，了解具体原因。）

代替AsyncTask的使用，接下来我们将创建一个专用的后台线程。这是实现按需下载的最常用方式。

27.3　与主线程通信

虽然我们准备采用专用线程负责下载图片，但在无法与主线程直接通信的情况下，它是如何协同GridView的adapter实现图片显示的呢？

再次回到闪电侠与鞋店的假想场景。后台工作的闪电侠已结束与分销商的电话沟通。他需要将库存已补足的消息通知给前台工作的闪电侠。如果前台闪电侠非常忙碌，则后台闪电侠可能无法立即与他取得联系。于是，他选择登记预约，等到前台闪电侠空闲时再联系。这虽然可行，但效率不高。

比较好的解决方案是为每个闪电侠提供一个收件箱。后台闪电侠写下库存补足的信息，并将其放置在前台闪电侠的收件箱顶部。而前台闪电侠如需告知后台闪电侠库存已空的信息，也可执行类似操作。

实践证明，收件箱的办法非常好用。有时，闪电侠可能需要及时完成某项任务，但当时并不方便去做。这种情况下，他也可以在自己的收件箱放上一条提醒消息，然后在空闲的时候去完成它。

Android系统中，线程使用的收件箱叫做*消息队列*（message queue）。使用消息队列的线程叫做*消息循环*（message loop）。*消息循环*会不断循环检查队列上是否有新消息，如图27-4所示。

消息循环由一个线程和一个looper组成。Looper对象管理着线程的消息队列。

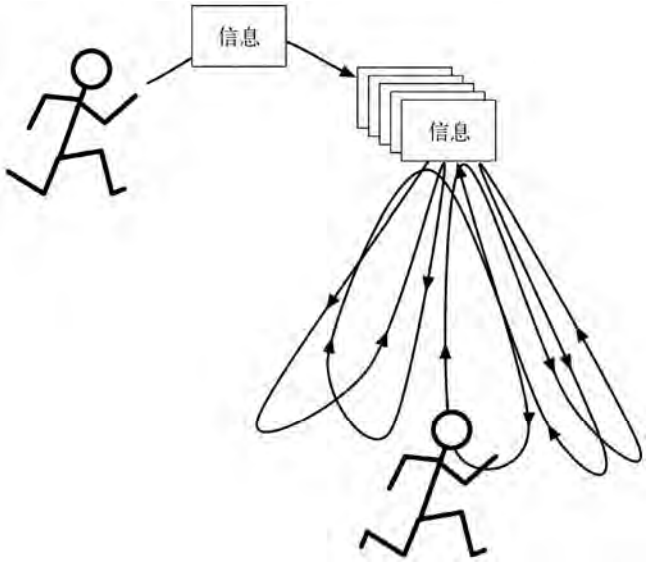

图27-4 闪电侠之舞

主线程也是一个消息循环，因此具有一个looper。主线程的所有工作都是由其looper完成的。looper不断从消息队列中抓取消息，然后完成消息指定的任务。

接下来，我们将创建一个同样是消息循环的后台线程。准备需要的looper时，我们会使用一个HandlerThread类。

27.4 创建并启动后台线程

继承HandlerThread类，创建一个名为ThumbnailDownloader的新类。ThumbnailDownloader类需要使用某些对象来标识每一次下载。因此，在类创建对话框，通过ThumbnailDownloader<Token>的命名，为其提供一个Token泛型参数。然后，再添加一个构造方法以及一个名为queueThumbnail()的存根方法。如代码清单27-2所示。

代码清单27-2 初始线程代码（ThumbnailDownloader.java）

```
public class ThumbnailDownloader<Token> extends HandlerThread {
    private static final String TAG = "ThumbnailDownloader";

    public ThumbnailDownloader() {
        super(TAG);
    }

    public void queueThumbnail(Token token, String url) {
        Log.i(TAG, "Got an URL: " + url);
    }
}
```

注意，queueThumbnail()方法需要一个Token和一个String参数。同时，它也是GalleryItemAdapter在其getView(...)实现方法中要调用的方法。

打开PhotoGalleryFragment.java文件，添加ThumbnailDownloader类型的成员变量到PhotoGalleryFragment。虽然可使用任何对象作为ThumbnailDownloader的token，但在这里，ImageView是最合适方便的token，因为该视图是下载图片最终要显示的地方。然后，在onCreate(...)方法中，创建并启动线程。最后，覆盖onDestroy()方法退出线程，如代码清单27-3所示。

代码清单27-3　创建ThumbnailDownloader（PhotoGalleryFragment.java）

```
public class PhotoGalleryFragment extends Fragment {
    private static final String TAG = "PhotoGalleryFragment";

    GridView mGridView;
    ArrayList<GalleryItem> mItems;
    ThumbnailDownloader<ImageView> mThumbnailThread;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setRetainInstance(true);
        new FetchItemsTask().execute();

        mThumbnailThread = new ThumbnailDownloader<ImageView>();
        mThumbnailThread.start();
        mThumbnailThread.getLooper();
        Log.i(TAG, "Background thread started");
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
        ...
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        mThumbnailThread.quit();
        Log.i(TAG, "Background thread destroyed");
    }
    ...
}
```

下面是两点安全注意事项。

- 在ThumbnailDownloader线程上，getLooper()方法是在start()方法之后调用的。这是一种保证线程就绪的处理方式（稍后，我们会学习到更多有关Looper的知识）。
- 结束线程的quit()方法是在onDestroy()方法内完成调用的。这非常关键。如不终止HandlerThread，它会一直运行下去。

最后，在GalleryItemAdapter.getView(...)方法中，使用position参数定位获取正确的

GalleryItem，然后调用线程的queueThumbnail()方法，并传入ImageView和gallery图片项的URL。如代码清单27-4所示。

代码清单27-4 关联使用ThumbnailDownloader（PhotoGalleryFragment.java）

```
    private class GalleryItemAdapter extends ArrayAdapter<GalleryItem> {
        ...

        @Override
        public View getView(int position, View convertView, ViewGroup parent) {
            ...

            ImageView imageView = (ImageView)convertView
                    .findViewById(R.id.gallery_item_imageView);
            imageView.setImageResource(R.drawable.brian_up_close);
            GalleryItem item = getItem(position);
            mThumbnailThread.queueThumbnail(imageView, item.getUrl());

            return convertView;
        }
    }
```

运行PhotoGallery应用并查看LogCat窗口。在GridView视图中滚动时，可看到ThumbnailDownloader正处理各个下载请求。

成功创建并运行HandlerThread线程后，接下来的任务是：使用传入queueThumbnail()方法的信息创建消息，并放置在ThubnailDownloader的消息队列中。

27.5 Message 与 message Handler

创建消息前，首先要理解什么是Message，以及它与Handler（或者说message handler）之间的关系。

27.5.1 消息的剖析

首先来看消息。闪电侠放入自己或另一闪电侠收件箱的消息并非鼓励性语句，如“你跑的真快，闪电侠。”，而是需要处理的各项任务。

消息是Message类的一个实例，包含有好几个实例变量。其中有三个需在实现时定义：

- what 用户定义的int型消息代码，用来描述消息；
- obj 随消息发送的用户指定对象；
- target 处理消息的Handler。

Message的目标是Handler类的一个实例。Handler可看作是“message handler”的简称。Message在创建时，会自动与一个Handler相关联。Message在准备处理状态下，Handler是负责让消息处理行为发生的对象。

27.5.2 Handler的剖析

要处理消息以及消息指定的任务，首先需要一个消息Handler实例。Handler不仅仅是处理

Message的目标（target），也是创建和发布Message的接口。

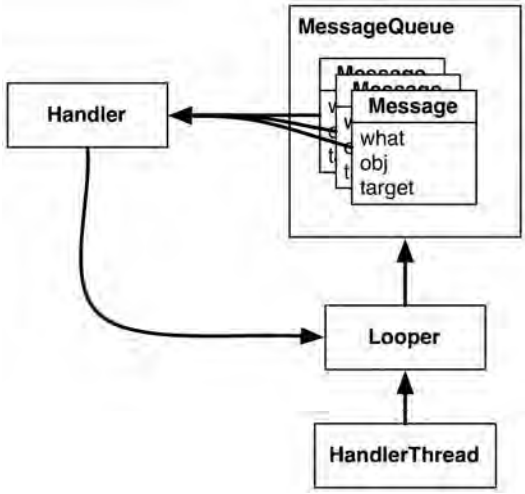

图27-5　Looper、Handler、HandlerThread与Message

Looper拥有Message对象的收件箱，所以Message必须在Looper上发布或读取。基于Looper和Message的这种关系，为与Looper协同工作，Handler总是引用着它。

一个Handler仅与一个Looper相关联，一个Message也仅与一个目标Handler（也称作Message目标）相关联。Looper拥有着整个Message队列。具体关系图如图27-5所示。

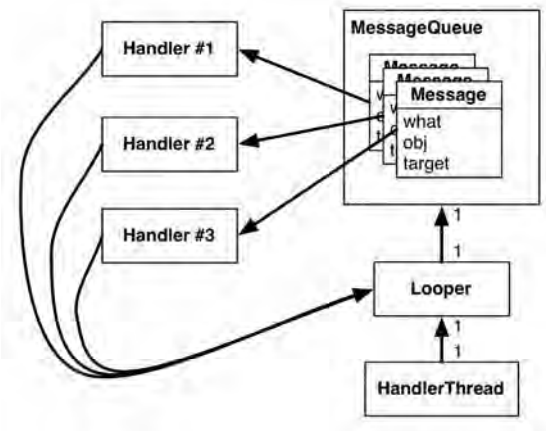

图27-6　多个Handler对应一个Looper

如图27-6所示，多个Handler可与一个Looper相关联。这意味着一个Handler的Message可能与另一个Handler的Message存放在同一消息队列中。

27.5.3 使用handler

消息的目标Handler通常不需要手动设置。一个比较理想的方式是，我们调用Handler.obtainMessage(...)方法创建信息并传入其他消息字段，然后该方法自动完成目标Handler的设置。

为避免创建新的Message对象，Handler.obtainMessage(...)方法会从公共循环池里获取消息。因此相比创建新实例，这样有效率多了。

一旦取得Message，我们就调用sendToTarget()方法将其发送给它的Handler。紧接着Handler会将Message放置在Looper消息队列的尾部。

PhotoGallery应用中，我们将在queueThumbnail()实现方法中获取并发送消息给它的目标。消息的what属性是一个定义为MESSAGE_DOWNLOAD的常量。消息的obj属性是一个Token，这里指由adapter传入queueThumbnail()方法的ImageView。

Looper取得消息队列中的特定消息后，会将它发送给消息目标去处理。消息一般是在目标的Handler.handleMessage(...)实现方法中进行处理的。

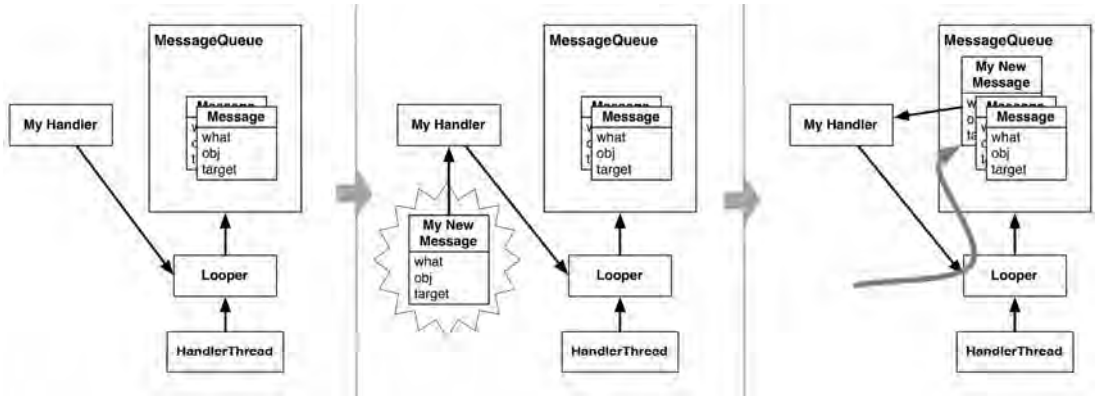

图27-7 创建并发送Message

这里，handleMessage(...)实现方法将使用FlickrFetchr从URL下载图片字节，然后再转换为位图。在ThumbnailDownloader.java中，添加代码清单27-5所示代码。

代码清单27-5 获取、发送以及处理消息（ThumbnailDownloader.java）

```
public class ThumbnailDownloader<Token> extends HandlerThread {
    private static final String TAG = "ThumbnailDownloader";
    private static final int MESSAGE_DOWNLOAD = 0;

    Handler mHandler;
    Map<Token, String> requestMap =
            Collections.synchronizedMap(new HashMap<Token, String>());

    public ThumbnailDownloader() {
        super(TAG);
    }
```

```
    @SuppressLint("HandlerLeak")
    @Override
    protected void onLooperPrepared() {
        mHandler = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                if (msg.what == MESSAGE_DOWNLOAD) {
                    @SuppressWarnings("unchecked")
                    Token token = (Token)msg.obj;
                    Log.i(TAG, "Got a request for url: " + requestMap.get(token));
                    handleRequest(token);
                }
            }
        };
    }

    public void queueThumbnail(Token token, String url) {
        Log.i(TAG, "Got a URL: " + url");
        requestMap.put(token, url);

        mHandler
            .obtainMessage(MESSAGE_DOWNLOAD, token)
            .sendToTarget();
    }

    private void handleRequest(final Token token) {
        try {
            final String url = requestMap.get(token);
            if (url == null)
                return;

            byte[] bitmapBytes = new FlickrFetchr().getUrlBytes(url);
            final Bitmap bitmap = BitmapFactory
                    .decodeByteArray(bitmapBytes, 0, bitmapBytes.length);
            Log.i(TAG, "Bitmap created");

        } catch (IOException ioe) {
            Log.e(TAG, "Error downloading image", ioe);
        }
    }
```

首先，解释一下onLooperPrepared()方法顶部的@SuppressLint("HandlerLeak")注解说明。这里，Android Lint将报出Handler类相关的警告信息。Looper控制着Handler的生死，因此如果Handler是匿名内部类，则隐式的对象引用很容易导致内存泄露。不过，所有对象都与HandlerThread绑定在一起，因此这里不用担心任何内存泄露问题。

另一个注解@SuppressWarnings("unchecked")，是常见的普通注解。这里必须使用该注解，因为Token是泛型类参数，而Message.obj是一个Object。由于类型擦除（type erasure），这里的强制类型转换应该是不可以的。如需进一步探究原因，请阅读类型擦除相关资料——本章仅关注Android本身的内容。

变量requestMap是一个同步HashMap。这里，使用Token作为key，可存储或获取与特定Token相关联的URL。

在queueThumbnail()方法中，将传入的Token-URL键值对放入map中。然后以Token为obj

获取一条消息，并发送出去以存放到消息队列中。

在onLooperPrepared()方法内，我们在Handler子类中实现了Handler.handleMessage(...)方法。因为HandlerThread.onLooperPrepared()方法的调用发生在Looper第一次检查消息队列之前，所以该方法成了我们创建Handler实现的好地方。

在Handler.handleMessage(...)方法中，检查消息类型，获取Token，然后将其传递给handleRequest(...)方法。

handleMessage(...)方法是下载动作发生的地方。这里，我们确认URL已存在后，将它传递给FlickrFetchr的新实例。确切地说，此处使用的是上一章中创建的FlickrFetchr.getUrlBytes(...)方法。

最后，使用BitmapFactory将getUrlBytes(...)返回的字节数组转换为位图。

运行PhotoGallery应用，并通过LogCat窗口的日志确认代码工作正常。

当然，在这里，将位图设置给ImageView视图（原始来自于GalleryItemAdapter）之前，请求并不会得到完全处理。不过这是UI的工作。因此，我们必须在主线程上完成它。

目前为止，我们全部的工作都是在线程上使用handler和消息，以及将消息放入自己的收件箱。下一小节，我们的学习内容是：ThumbnailDownloader是如何使用Handler访问主线程的。

27.5.4 传递handler

HandlerThread能在主线程上完成任务的一种方式是，让主线程将其自身的Handler传递给HandlerThread。

主线程是一个拥有handler和Looper的消息循环。主线程上创建的Handler会自动与它的Looper相关联。我们可以将主线程上创建的Handler传递给另一线程。传递出去的Handler与创建它的线程Looper始终保持着联系。因此，任何已传出Handler负责处理的消息都将在主线程的消息队列中处理。

这看上去就像我们在使用ThumbnailDownloader的Handler，实现在主线程上安排后台线程上的任务，如图27-8所示。

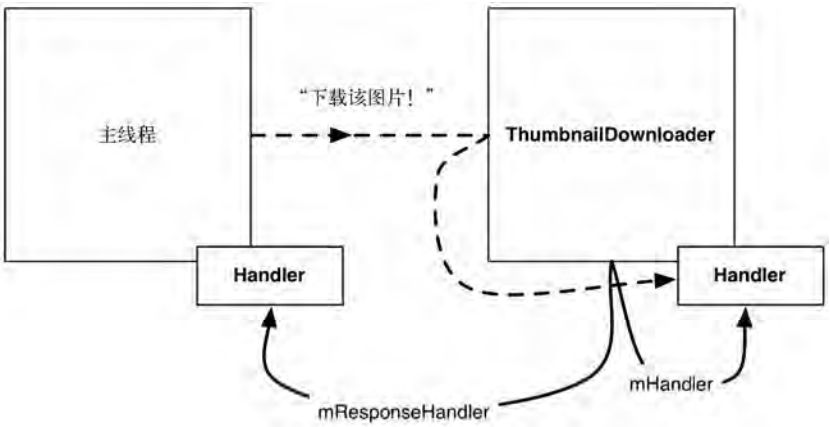

图27-8 从主线程安排ThumbnailDownloader上的任务

反过来，我们也可从后台线程使用与主线程关联的Handler，安排要在主线程上完成的任务，如图27-9所示。

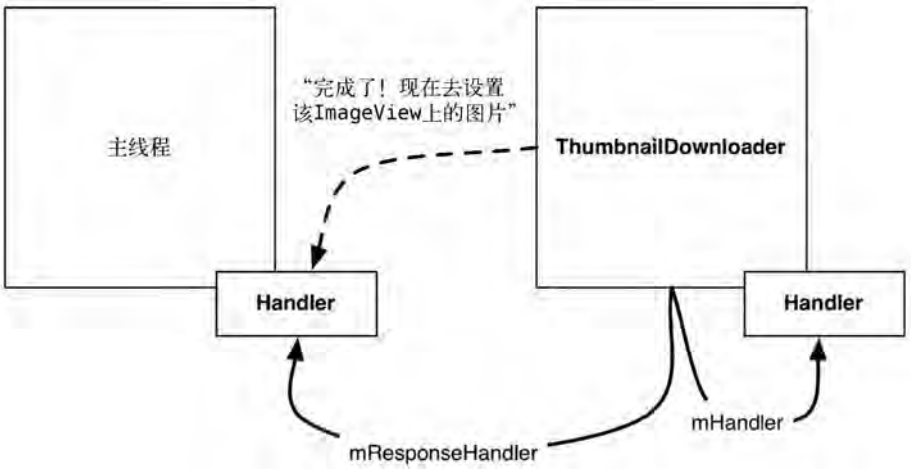

图27-9　从ThumbnailDownloader线程上规划主线程上执行的任务

在ThumbnailDownloader.java中，添加上述mResponseHandler变量，以存放来自于主线程的Handler。然后，以一个接受Handler的构造方法替换原有构造方法，并设置变量的值，最后新增一个用来通信的监听器接口。如代码清单27-6所示。

代码清单27-6　添加反馈Handler（ThumbnailDownloader.java）

```
Public class ThumbnailDownloader <Token> extends HandlerThread {
    private static final String TAG = "ThumbnailDownloader";
    private static final int MESSAGE_DOWNLOAD = 0;

    Handler mHandler;
    Map<Token,String> requestMap =
            Collections.synchronizedMap(new HashMap<Token,String>());
    Handler mResponseHandler;
    Listener<Token> mListener;

    public interface Listener<Token> {
        void onThumbnailDownloaded(Token token, Bitmap thumbnail);
    }

    public void setListener(Listener<Token> listener) {
        mListener = listener;
    }

    public ThumbnailDownloader() {
        super(TAG);
    public ThumbnailDownloader(Handler responseHandler) {
        super(TAG);
        mResponseHandler = responseHandler;
    }
```

修改PhotoGalleryFragment类，将Handler传递给ThumbnailDownloader，并设置Listener，将返回的BitMap设置给ImageView。记住，Handler默认与当前线程的Looper相关联。该Handler是在onCreate(...)方法中创建的，因此它将与主线程的Looper相关联。如代码清单27-7所示。

代码清单27-7 关联使用反馈Handler（PhotoGalleryFragment.java）

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

    setRetainInstance(true);
    new FetchItemsTask().execute();

    mThumbnailThread = new ThumbnailDownloader();
    mThumbnailThread = new ThumbnailDownloader<ImageView>(new Handler());
    mThumbnailThread.setListener(new ThumbnailDownloader.Listener<ImageView>() {
        public void onThumbnailDownloaded(ImageView imageView, Bitmap thumbnail) {
            if (isVisible()) {
                imageView.setImageBitmap(thumbnail);
            }
        }
    });
    mThumbnailThread.start();
    mThumbnailThread.getLooper();
    Log.i(TAG, "Background thread started");
}
```

现在，通过mResponseHandler，ThumbnailDownloader能够访问与主线程Looper绑定的Handler。同时，它的Listener会使用返回的Bitmap执行UI更新操作。注意，调用imageView.setImageBitmap(Bitmap)方法之前，应首先调用Fragment.isVisible()检查方法，以保证不会将图片设置到无效的ImageView视图上去。

我们也可返回定制Message给主线程。不过，这需要另一个Handler子类，以及一个handleMessage(...)覆盖方法。这里，我们使用的是另一种方便的Handler方法——post(Runnable)。

Handler.post(Runnable)是一个张贴Message的便利方法。具体使用如下：

```
Runnable myRunnable = new Runnable() {
    public void run() {
        /* Your code here */
    }
};
Message m = mHandler.obtainMessage();
m.callback = myRunnable;
```

Message具有回调方法时，使用回调方法中的Runnable，而非其Handler目标来实现运行。

在ThumbnailDownloader.handleRequest()方法中，添加代码清单27-8所示代码。

代码清单27-8 下载与显示（ThumbnailDownloader.java）

```
...

private void handleRequest(final Token token) {
    try {
        final String url = requestMap.get(token);
        if (url == null)
```

```
                return;

            byte[] bitmapBytes = new FlickrFetchr().getUrlBytes(url);
            final Bitmap bitmap = BitmapFactory
                    .decodeByteArray(bitmapBytes, 0, bitmapBytes.length);
            Log.i(TAG, "Bitmap created");

            mResponseHandler.post(new Runnable() {
                public void run() {
                    if (requestMap.get(token) != url)
                        return;

                    requestMap.remove(token);
                    mListener.onThumbnailDownloaded(token, bitmap);
                }
            });
        } catch (IOException ioe) {
            Log.e(TAG, "Error downloading image", ioe);
        }
    }
```

因为mResponseHandler与主线程的Looper相关联，所以UI更新代码也是在主线程中完成的。

那么这段代码有什么作用呢？首先，它再次检查了requestMap。这很有必要，因为GridView会循环使用它的视图。ThumbnailDownloader完成Bitmap下载后，GridView可能已经循环使用了ImageView，并继续请求一个不同的URL。该检查可保证每个Token都能获取到正确的图片，即使中间发生了其他请求也无妨。

最后，从requestMap中删除Token，然后将bitmap设置到Token上。

在运行应用并欣赏下载的图片前，还应考虑一个风险点。如果用户旋转屏幕，因ImageView视图的失效，ThumbnailDownloader则可能会挂起。如果点击这些ImageView，就可能发生异常。

新增下列方法清除队列外的所有请求，如代码清单27-9所示。

代码清单27-9　添加清理方法（ThumbnailDownloader.java）

```
public void clearQueue() {
    mHandler.removeMessages(MESSAGE_DOWNLOAD);
    requestMap.clear();
}
```

既然视图已销毁，别忘了在PhotoGalleryFragment.java中添加下载器清除代码，如代码清单27-10所示。

代码清单27-10　调用清理方法（PhotoGalleryFragment.java）

```
@Override
public void onDestroyView() {
    super.onDestroyView();
    mThumbnailThread.clearQueue();
}
```

至此，本章的所有任务都完成了。运行PhotoGallery应用。滚动屏幕查看图片的动态下载。

PhotoGallery应用已完成从Flickr下载并显示图片的基本目标。接下来的几章，我们将为应用增加更多功能，如搜索图片、在web视图中打开图片的Flickr网页。

27.6 深入学习：AsyncTask 与 Thread

理解了Handler和Looper后，会发现AsyncTask也没看上去那么神奇。不过就本章所做的线程相关工作来看，改用AsyncTask会省事些。那么为什么还是坚持使用HandlerThread，而不使用AsyncTask呢?

原因有很多。最基本的一个是因为AsyncTask的工作方式并不适用于本章的使用场景。它主要应用于那些短暂且较少重复的任务。上一章的实现代码才是AsyncTask大展身手的地方。如果创建了大量的AsyncTask，或者长时间运行了AsyncTask，那么很可能是做了错误的选择。

另一个更让人信服的技术层面理由是，在Android 3.2系统版本中，AsyncTask的内部实现发生了重大变化。自Android 3.2版本起，AsyncTask不再为每一个AsyncTask实例单独创建一个线程。相反，它使用一个Executor在单一的后台线程上运行所有AsyncTask的后台任务。这意味着每个AsyncTask都需要排队逐个运行。显然，长时间运行的AsyncTask会阻塞其他AsyncTask。

使用一个线程池executor虽然可安全地并发运行多个AsyncTask，但我们不推荐这么做。如果真的考虑这么做，最好自己处理线程相关的工作，必要时可使用Handler与主线程通信。

27.7 挑战练习：预加载以及缓存

应用中并非所有任务都能即时完成，对此，大多用户表示理解。不过，即使是这样，开发者们也一直在努力做到最好。

为接近完美的即时性，大多实际应用都通过以下两种方式增强自己的代码：

- 增加一个缓存层
- 预加载图片

缓存指存储一定数目Bitmap对象的地方。这样，即使不再使用这些对象，它们也依然存储在那里。缓存的存储空间有限，因此，在缓存空间用完的情况下，需要某种策略对保存的对象做一定的取舍。许多缓存机制使用一种叫做LRU（least recently used，最近最少使用）的存储策略。基于该种策略，当存储空间用尽时，缓存将清除最近最少使用的对象。

Android支持库中的LruCache类实现了LRU缓存策略。作为第一个挑战练习，请使用LruCache为ThumbnailDownloader增加简单的缓存功能。这样，每次完成下载Bitmap时，将其存入缓存中。然后，准备下载新图片时，首先查看缓存，确认它是否存在。

缓存实现完成后，即可使用它进行预加载。预加载是指在实际使用对象前，预先就将处理对象加载到缓存中。这样，在显示Bitmap时，就不会存在下载延迟。

实现完美的预加载比较棘手，但对用户来说，良好的预加载是一种截然不同的体验。作为第二个稍有难度的挑战练习，请在显示GalleryItem时，为前十个和后十个GalleryItem预加载Bitmap。