Android性能优化之常见的内存泄漏

前言

对于内存泄漏，我想大家在开发中肯定都遇到过，只不过内存泄漏对我们来说并不是可见的，因为它是在堆中活动，而要想检测程序中是否有内存泄漏的产生，通常我们可以借助LeakCanary、MAT等工具来检测应用程序是否存在内存泄漏，MAT是一款强大的内存分析工具，功能繁多而复杂，而LeakCanary则是由Square开源的一款轻量第三方内存泄漏检测工具，当它检测到程序中有内存泄漏的产生时，它将以最直观的方式告诉我们该内存泄漏是由谁产生的和该内存泄漏导致谁泄漏了而不能回收，供我们复查。

最近腾讯bugly也推出了三篇关于Android内存泄漏调优的文章：
1、内存泄露从入门到精通三部曲之基础知识篇
2、内存泄露从入门到精通三部曲之排查方法篇
3、内存泄露从入门到精通三部曲之常见原因与用户实践

关于性能优化的文章，出自Realm.io：
10 条提升 Android 性能的建议

内存泄漏

为什么会产生内存泄漏？

当一个对象已经不需要再使用了，本该被回收时，而有另外一个正在使用的对象持有它的引用从而导致它不能被回收，这导致本该被回收的对象不能被回收而停留在堆内存中，这就产生了内存泄漏。

内存泄漏对程序的影响？

内存泄漏是造成应用程序OOM的主要原因之一！我们知道Android系统为每个应用程序分配的内存有限，而当一个应用中产生的内存泄漏比较多时，这就难免会导致应用所需要的内存超过这个系统分配的内存限额，这就造成了内存溢出而导致应用Crash。

Android中常见的内存泄漏汇总

单例造成的内存泄漏

单例模式非常受开发者的喜爱，不过使用的不恰当的话也会造成内存泄漏，由于单例的静态特性使得单例的生命周期和应用的生命周期一样长，这就说明了如果一个对象已经不需要使用了，而单例对象还持有该对象的引用，那么这个对象将不能被正常回收，这就导致了内存泄漏。
如下这个典例：

```Java
public class AppManager {
private static AppManager instance;
private Context context;
private AppManager(Context context) {
this.context = context;
}
public static AppManager getInstance(Context context) {
if (instance != null) {
instance = new AppManager(context);
}
return instance;
}
}
```

这是一个普通的单例模式，当创建这个单例的时候，由于需要传入一个Context，所以这个Context的生命周期的长短至关重要：

1、传入的是Application的Context：这将没有任何问题，因为单例的生命周期和Application的一样长
2、传入的是Activity的Context：当这个Context所对应的Activity退出时，由于该Context和Activity的生命周期一样长（Activity间接继承于Context），所以当前Activity退出时它的内存并不会被回收，因为单例对象持有该Activity的引用。
所以正确的单例应该修改为下面这种方式：

```Java
public class AppManager {
private static AppManager instance;
private Context context;
private AppManager(Context context) {
this.context = context.getApplicationContext();
}
public static AppManager getInstance(Context context) {
if (instance != null) {
instance = new AppManager(context);
}
return instance;
}
}
```

这样不管传入什么Context最终将使用Application的Context，而单例的生命周期和应用的一样长，这样就防止了内存泄漏

非静态内部类创建静态实例造成的内存泄漏

有的时候我们可能会在启动频繁的Activity中，为了避免重复创建相同的数据资源，可能会出现这种写法：

```Java
public class MainActivity extends AppCompatActivity {
private static TestResource mResource = null;
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
if(mManager == null){
mManager = new TestResource();
}
//...
}
class TestResource {
//...
}
}
```

这样就在Activity内部创建了一个非静态内部类的单例，每次启动Activity时都会使用该单例的数据，这样虽然避免了资源的重复创建，不过这种写法却会造成内存泄漏，因为非静态内部类默认会持有外部类的引用，而又使用了该非静态内部类创建了一个静态的实例，该实例的生命周期和应用的一样长，这就导致了该静态实例一直会持有该Activity的引用，导致Activity的内存资源不能正常回收。正确的做法为：
将该内部类设为静态内部类或将该内部类抽取出来封装成一个单例，如果需要使用Context，请使用ApplicationContext

Handler造成的内存泄漏

Handler的使用造成的内存泄漏问题应该说最为常见了，平时在处理网络任务或者封装一些请求回调等api都应该会借助Handler来处理，对于Handler的使用代码编写一不规范即有可能造成内存泄漏，如下示例：

```Java
public class MainActivity extends AppCompatActivity {
private Handler mHandler = new Handler() {
@Override
public void handleMessage(Message msg) {
//...
}
};
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
loadData();
}
private void loadData(){
//...request
Message message = Message.obtain();
```

```
17 mHandler.sendMessage(message);
18 }
19 }
```

这种创建Handler的方式会造成内存泄漏，由于mHandler是Handler的非静态匿名内部类的实例，所以它持有外部类Activity的引用，我们知道消息队列是在一个Looper线程中不断轮询处理消息，那么当这个Activity退出时消息队列中还有未处理的消息或者正在处理消息，而消息队列中的Message持有mHandler实例的引用，mHandler又持有Activity的引用，所以导致该Activity的内存资源无法及时回收，引发内存泄漏，所以另外一种做法为：

Java

```
1  public class MainActivity extends AppCompatActivity {
2  private MyHandler mHandler = new MyHandler(this);
3  private TextView mTextView ;
4  private static class MyHandler extends Handler {
5  private WeakReference<Context> reference;
6  public MyHandler(Context context) {
7  reference = new WeakReference<>(context);
8  }
9  @Override
10 public void handleMessage(Message msg) {
11 MainActivity activity = (MainActivity) reference.get();
12 if(activity != null){
13 activity.mTextView.setText("");
14 }
15 }
16 }
17
18 @Override
19 protected void onCreate(Bundle savedInstanceState) {
20 super.onCreate(savedInstanceState);
21 setContentView(R.layout.activity_main);
22 mTextView = (TextView)findViewById(R.id.textview);
23 loadData();
24 }
25
26 private void loadData() {
27 //...request
28 Message message = Message.obtain();
29 mHandler.sendMessage(message);
30 }
31 }
```

创建一个静态Handler内部类，然后对Handler持有的对象使用弱引用，这样在回收时也可以回收Handler持有的对象，这样虽然避免了Activity泄漏，不过Looper线程的消息队列中还是可能会有待处理的消息，所以我们在Activity的Destroy时或者Stop时应该移除消息队列中的消息，更准确的做法如下：

Java

```
1  public class MainActivity extends AppCompatActivity {
2  private MyHandler mHandler = new MyHandler(this);
3  private TextView mTextView ;
4  private static class MyHandler extends Handler {
5  private WeakReference<Context> reference;
6  public MyHandler(Context context) {
7  reference = new WeakReference<>(context);
8  }
9  @Override
```

```
10 public void handleMessage(Message msg) {
11 MainActivity activity = (MainActivity) reference.get();
12 if(activity != null){
13 activity.mTextView.setText("");
14 }
15 }
16 }
17
18 @Override
19 protected void onCreate(Bundle savedInstanceState) {
20 super.onCreate(savedInstanceState);
21 setContentView(R.layout.activity_main);
22 mTextView = (TextView)findViewById(R.id.textview);
23 loadData();
24 }
25
26 private void loadData() {
27 //...request
28 Message message = Message.obtain();
29 mHandler.sendMessage(message);
30 }
31
32 @Override
33 protected void onDestroy() {
34 super.onDestroy();
35 mHandler.removeCallbacksAndMessages(null);
36 }
37 }
```

使用mHandler.removeCallbacksAndMessages(null);是移除消息队列中所有消息和所有的Runnable。当然也可以使用mHandler.removeCallbacks();或mHandler.removeMessages();来移除指定的Runnable和Message。

线程造成的内存泄漏

对于线程造成的内存泄漏，也是平时比较常见的，如下这两个示例可能每个人都这样写过：

Java

```
1  public class MainActivity extends AppCompatActivity {
2  private MyHandler mHandler = new MyHandler(this);
3  private TextView mTextView ;
4  private static class MyHandler extends Handler {
5  private WeakReference<Context> reference;
6  public MyHandler(Context context) {
7  reference = new WeakReference<>(context);
8  }
9  @Override
10 public void handleMessage(Message msg) {
11 MainActivity activity = (MainActivity) reference.get();
12 if(activity != null){
13 activity.mTextView.setText("");
14 }
15 }
16 }
17
18 @Override
19 protected void onCreate(Bundle savedInstanceState) {
20 super.onCreate(savedInstanceState);
21 setContentView(R.layout.activity_main);
22 mTextView = (TextView)findViewById(R.id.textview);
```

```
23 loadData();
24 }
25
26 private void loadData() {
27 //...request
28 Message message = Message.obtain();
29 mHandler.sendMessage(message);
30 }
31
32 @Override
33 protected void onDestroy() {
34 super.onDestroy();
35 mHandler.removeCallbacksAndMessages(null);
36 }
37 }
```

上面的异步任务和Runnable都是一个匿名内部类，因此它们对当前Activity都有一个隐式引用。如果Activity在销毁之前，任务还未完成，
那么将导致Activity的内存资源无法回收，造成内存泄漏。正确的做法还是使用静态内部类的方式，如下：

Java

```
1  static class MyAsyncTask extends AsyncTask<Void, Void, Voi
2  private WeakReference<Context> weakReference;
3
4  public MyAsyncTask(Context context) {
5  weakReference = new WeakReference<>(context);
6  }
7
8  @Override
9  protected Void doInBackground(Void... params) {
10 SystemClock.sleep(10000);
11 return null;
12 }
13
14 @Override
15 protected void onPostExecute(Void aVoid) {
16 super.onPostExecute(aVoid);
17 MainActivity activity = (MainActivity) weakReference.get()
18 if (activity != null) {
19 //...
20 }
21 }
22 }
23 static class MyRunnable implements Runnable{
24 @Override
25 public void run() {
26 SystemClock.sleep(10000);
27 }
28 }
29 //------
30 new Thread(new MyRunnable()).start();
31 new MyAsyncTask(this).execute();
```

这样就避免了Activity的内存资源泄漏，当然在Activity销毁时候也应该取消相应的任务AsyncTask::cancel()，避免任务在后台执行浪费资源。

资源未关闭造成的内存泄漏

对于使用了BraodcastReceiver，ContentObserver，File，Cursor，Stream，Bitmap等资源的使用，应该在Activity销毁时及时关闭或者注销，否则这些资源将不会被回收，造成内存泄漏。

一些建议

1、对于生命周期比Activity长的对象如果需要应该使用ApplicationContext
2、在涉及到Context时先考虑ApplicationContext，当然它并不是万能的，对于有些地方则必须使用Activity的Context，对于Application，Service，Activity三者的Context的应用场景如下：

功能	Application	Service	Activity
Start an Activity	NO1	NO1	YES
Show a Dialog	NO	NO	YES
Layout Inflation	YES	YES	YES
Start a Service	YES	YES	YES
Bind to a Service	YES	YES	YES
Send a Broadcast	YES	YES	YES
Register BroadcastReceiver	YES	YES	YES
Load Resource Values	YES	YES	YES

其中：NO1表示Application和Service可以启动一个Activity，不过需要创建一个新的task任务队列。而对于Dialog而言，只有在Activity中才能创建
3、对于需要在静态内部类中使用非静态外部成员变量（如：Context、View），可以在静态内部类中使用弱引用来引用外部类的变量来避免内存泄漏
4、对于生命周期比Activity长的内部类对象，并且内部类中使用了外部类的成员变量，可以这样做避免内存泄漏：

> 1. 将内部类改为静态内部类
> 2. 静态内部类中使用弱引用来引用外部类的成员变量

5、对于不再需要使用的对象，显示的将其赋值为null，比如使用完Bitmap后先调用recycle()，再赋为null
6、保持对对象生命周期的敏感，特别注意单例、静态对象、全局性集合等的生命周期