Android 内存优化——常见内存泄露及优化方案

如果一个无用对象（不需要再使用的对象）仍然被其他对象持有引用，造成该对象无法被系统回收，以致该对象在堆中所占用的内存单元无法被释放而造成内存空间浪费，这中情况就是内存泄露。

在 Android 开发中，一些不好的编程习惯会导致我们的开发的 app 存在内存泄露的情况。下面介绍一些在 Android 开发中常见的内存泄露场景及优化方案。

单例导致内存泄露

单例模式在 Android 开发中会经常用到，但是如果使用不当就会导致内存泄露。因为单例的静态特性使得它的生命周期同应用的生命周期一样长，如果一个对象已经没有用处了，但是单例还持有它的引用，那么在整个应用程序的生命周期它都不能正常被回收，从而导致内存泄露。

```
public class AppSettings {



    private static AppSettings sInstance;

    private Context mContext;



    private AppSettings(Context context) {

        this.mContext = context;

    }



    public static AppSettings getInstance(Context context) {

        if (sInstance == null) {

            sInstance = new AppSettings(context);

        }

        return sInstance;

    }

}
```

像上面代码中这样的单例，如果我们在调用 `getInstance(Context context)`方法的时候传入的 `context` 参数是 `Activity`、`Service` 等上下文，就会导致内存泄露。

以 Activity 为例，当我们启动一个 Activity，并调用 getInstance(Context context)方法去获取 AppSettings 的单例，传入 Activity.this 作为 context，这样 AppSettings 类的单例 sInstance 就持有了 Activity 的引用，当我们退出 Activity 时，该 Activity 就没有用了，但是因为 sIntance 作为静态单例（在应用程序的整个生命周期中存在）会继续持有这个 Activity 的引用，导致这个 Activity 对象无法被回收释放，这就造成了内存泄露。

为了避免这样单例导致内存泄露，我们可以将 context 参数改为全局的上下文：

```
private AppSettings(Context context) {

    this.mContext = context.getApplicationContext();

}
```

全局的上下文 Application Context 就是应用程序的上下文，和单例的生命周期一样长，这样就避免了内存泄漏。

单例模式对应应用程序的生命周期，所以我们在构造单例的时候尽量避免使用 Activity 的上下文，而是使用 Application 的上下文。

静态变量导致内存泄露

静态变量存储在方法区，它的生命周期从类加载开始，到整个进程结束。一旦静态变量初始化后，它所持有的引用只有等到进程结束才会释放。

比如下面这样的情况，在 Activity 中为了避免重复的创建 info，将 sInfo 作为静态变量：

```
public class MainActivity extends AppCompatActivity {



    private static Info sInfo;



    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        if (sInfo != null) {

            sInfo = new Info(this);

        }

    }

}
```

```
class Info {

    public Info(Activity activity) {

    }

}
```

Info 作为 Activity 的静态成员，并且持有 Activity 的引用，但是 sInfo 作为静态变量，生命周期肯定比 Activity 长。所以当 Activity 退出后，sInfo 仍然引用了 Activity，Activity 不能被回收，这就导致了内存泄露。

在 Android 开发中，静态持有很多时候都有可能因为其使用的生命周期不一致而导致内存泄露，所以我们在新建静态持有的变量的时候需要多考虑一下各个成员之间的引用关系，并且尽量少地使用静态持有的变量，以避免发生内存泄露。当然，我们也可以在适当的时候讲静态量重置为 null，使其不再持有引用，这样也可以避免内存泄露。

非静态内部类导致内存泄露

非静态内部类（包括匿名内部类）默认就会持有外部类的引用，当非静态内部类对象的生命周期比外部类对象的生命周期长时，就会导致内存泄露。

非静态内部类导致的内存泄露在 Android 开发中有一种典型的场景就是使用 Handler，很多开发者在使用 Handler 是这样写的：

```
public class MainActivity extends AppCompatActivity {



    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        start();

    }



    private void start() {

        Message msg = Message.obtain();

        msg.what = 1;

        mHandler.sendMessage(msg);

    }
```

```
    private Handler mHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            if (msg.what == 1) {
                // 做相应逻辑
            }
        }
    };
}
```

也许有人会说，mHandler 并未作为静态变量持有 Activity 引用，生命周期可能不会比 Activity 长，应该不一定会导致内存泄露呢，显然不是这样的！

熟悉 Handler 消息机制的都知道，mHandler 会作为成员变量保存在发送的消息 msg 中，即 msg 持有 mHandler 的引用，而 mHandler 是 Activity 的非静态内部类实例，即 mHandler 持有 Activity 的引用，那么我们就可以理解为 msg 间接持有 Activity 的引用。msg 被发送后先放到消息队列 MessageQueue 中，然后等待 Looper 的轮询处理（MessageQueue 和 Looper 都是与线程相关联的，MessageQueue 是 Looper 引用的成员变量，而 Looper 是保存在 ThreadLocal 中的）。那么当 Activity 退出后，msg 可能仍然存在于消息对列 MessageQueue 中未处理或者正在处理，那么这样就会导致 Activity 无法被回收，以致发生 Activity 的内存泄露。

通常在 Android 开发中如果要使用内部类，但又要规避内存泄露，一般都会采用*静态内部类+弱引用*的方式。

```
public class MainActivity extends AppCompatActivity {

    private Handler mHandler;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mHandler = new MyHandler(this);
        start();
```

```
    }

    private void start() {
        Message msg = Message.obtain();
        msg.what = 1;
        mHandler.sendMessage(msg);
    }

    private static class MyHandler extends Handler {

        private WeakReference<MainActivity> activityWeakReference;

        public MyHandler(MainActivity activity) {
            activityWeakReference = new WeakReference<>(activity);
        }

        @Override
        public void handleMessage(Message msg) {
            MainActivity activity = activityWeakReference.get();
            if (activity != null) {
                if (msg.what == 1) {
                    // 做相应逻辑
                }
            }
        }
    }
}
```

mHandler 通过弱引用的方式持有 Activity，当 GC 执行垃圾回收时，遇到 Activity 就会回收并释放所占据的内存单元。这样就不会发生内存泄露了。

上面的做法确实避免了 Activity 导致的内存泄露，发送的 msg 不再已经没有持有 Activity 的引用了，但是 msg 还是有可能存在消息队列 MessageQueue 中，所以更好的是在 Activity 销毁时就将 mHandler 的回调和发送的消息给移除掉。

```
@Overrideprotected void onDestroy() {

    super.onDestroy();

    mHandler.removeCallbacksAndMessages(null);

}
```

非静态内部类造成内存泄露还有一种情况就是使用 Thread 或者 AsyncTask。

比如在 Activity 中直接 new 一个子线程 Thread：

```
public class MainActivity extends AppCompatActivity {



    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        new Thread(new Runnable() {

            @Override

            public void run() {

                // 模拟相应耗时逻辑

                try {

                    Thread.sleep(2000);

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

        }).start();

    }
```

```
}
```

或者直接新建 AsyncTask 异步任务：

```
public class MainActivity extends AppCompatActivity {



    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        new AsyncTask<Void, Void, Void>() {

            @Override

            protected Void doInBackground(Void... params) {

                // 模拟相应耗时逻辑

                try {

                    Thread.sleep(2000);

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

                return null;

            }

        }.execute();

    }

}
```

很多初学者都会像上面这样新建线程和异步任务，殊不知这样的写法非常地不友好，这种方式新建的子线程 Thread 和 AsyncTask 都是匿名内部类对象，默认就隐式的持有外部 Activity 的引用，导致 Activity 内存泄露。要避免内存泄露的话还是需要像上面 Handler 一样使用*静态内部类+弱应用*的方式（代码就不列了，参考上面 Hanlder 的正确写法）。

未取消注册或回调导致内存泄露

比如我们在 Activity 中注册广播，如果在 Activity 销毁后不取消注册，那么这个刚播会一直存在系统中，同上面所说的非静态内部类一样持有 Activity 引用，导致内存泄露。因此注册广播后在 Activity 销毁后一定要取消注册。

```
public class MainActivity extends AppCompatActivity {



    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        this.registerReceiver(mReceiver, new IntentFilter());

    }



    private BroadcastReceiver mReceiver = new BroadcastReceiver() {

        @Override

        public void onReceive(Context context, Intent intent) {

            // 接收到广播需要做的逻辑

        }

    };



    @Override

    protected void onDestroy() {

        super.onDestroy();

        this.unregisterReceiver(mReceiver);

    }

}
```

在注册观察则模式的时候，如果不及时取消也会造成内存泄露。比如使用 Retrofit+RxJava 注册网络请求的观察者回调，同样作为匿名内部类持有外部引用，所以需要记得在不用或者销毁的时候取消注册。

Timer 和 TimerTask 导致内存泄露

Timer 和 TimerTask 在 Android 中通常会被用来做一些计时或循环任务，比如实现无限轮播的 ViewPager：

```
public class MainActivity extends AppCompatActivity {



    private ViewPager mViewPager;

    private PagerAdapter mAdapter;

    private Timer mTimer;

    private TimerTask mTimerTask;



    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        init();

        mTimer.schedule(mTimerTask, 3000, 3000);

    }



    private void init() {

        mViewPager = (ViewPager) findViewById(R.id.view_pager);

        mAdapter = new ViewPagerAdapter();

        mViewPager.setAdapter(mAdapter);



        mTimer = new Timer();

        mTimerTask = new TimerTask() {

            @Override

            public void run() {

                MainActivity.this.runOnUiThread(new Runnable() {

                    @Override
```

```
                public void run() {

                    loopViewpager();

                }

            });

        }

    };

}



private void loopViewpager() {

    if (mAdapter.getCount() > 0) {

        int curPos = mViewPager.getCurrentItem();

        curPos = (++curPos) % mAdapter.getCount();

        mViewPager.setCurrentItem(curPos);

    }

}



private void stopLoopViewPager() {

    if (mTimer != null) {

        mTimer.cancel();

        mTimer.purge();

        mTimer = null;

    }

    if (mTimerTask != null) {

        mTimerTask.cancel();

        mTimerTask = null;

    }

}
```

```
    @Override
    protected void onDestroy() {
        super.onDestroy();
        stopLoopViewPager();
    }
}
```

当我们 Activity 销毁的时，有可能 Timer 还在继续等待执行 TimerTask，它持有 Activity 的引用不能被回收，因此当我们 Activity 销毁的时候要立即 cancel 掉 Timer 和 TimerTask，以避免发生内存泄漏。

集合中的对象未清理造成内存泄露

这个比较好理解，如果一个对象放入到 ArrayList、HashMap 等集合中，这个集合就会持有该对象的引用。当我们不再需要这个对象时，也并没有将它从集合中移除，这样只要集合还在使用（而此对象已经无用了），这个对象就造成了内存泄露。并且如果集合被静态引用的话，集合里面那些没有用的对象更会造成内存泄露了。所以在使用集合时要及时将不用的对象从集合 remove，或者 clear 集合，以避免内存泄漏。

资源未关闭或释放导致内存泄露

在使用 IO、File 流或者 Sqlite、Cursor 等资源时要及时关闭。这些资源在进行读写操作时通常都使用了缓冲，如果及时不关闭，这些缓冲对象就会一直被占用而得不到释放，以致发生内存泄露。因此我们在不需要使用它们的时候就及时关闭，以便缓冲能及时得到释放，从而避免内存泄露。

属性动画造成内存泄露

动画同样是一个耗时任务，比如在 Activity 中启动了属性动画（ObjectAnimator），但是在销毁的时候，没有调用 cancle 方法，虽然我们看不到动画了，但是这个动画依然会不断地播放下去，动画引用所在的控件，所在的控件引用 Activity，这就造成 Activity 无法正常释放。因此同样要在 Activity 销毁的时候 cancel 掉属性动画，避免发生内存泄漏。

```
@Overrideprotected void onDestroy() {
    super.onDestroy();
    mAnimator.cancel();
}
```

WebView 造成内存泄露

关于 WebView 的内存泄露，因为 WebView 在加载网页后会长期占用内存而不能被释放，因此我们在 Activity 销毁后要调用它的 destory()方法来销毁它以释放内存。

另外在查阅 WebView 内存泄露相关资料时看到这种情况：

Webview 下面的 Callback 持有 Activity 引用，造成 Webview 内存无法释放，即使是调用了 Webview.destory()等方法都无法解决问题（Android5.1 之后）。

最终的解决方案是：在销毁 WebView 之前需要先将 WebView 从父容器中移除，然后在销毁 WebView。详细分析过程请参考这篇文章：[](http://blog.csdn.net/xygy8860/article/details/53334476?utm_source=itdadao&utm_medium=referral)(http://blog.csdn.net/xygy8860/article/details/53334476)[WebView 内存泄漏解决方法]。

```
@Overrideprotected void onDestroy() {

    super.onDestroy();

    // 先从父控件中移除 WebView

    mWebViewContainer.removeView(mWebView);

    mWebView.stopLoading();

    mWebView.getSettings().setJavaScriptEnabled(false);

    mWebView.clearHistory();

    mWebView.removeAllViews();

    mWebView.destroy();

}
```

总结

内存泄露在 Android 内存优化是一个比较重要的一个方面，很多时候程序中发生了内存泄露我们不一定就能注意到，所有在编码的过程要养成良好的习惯。总结下来只要做到以下这几点就能避免大多数情况的内存泄漏：

构造单例的时候尽量别用 Activity 的引用；
静态引用时注意应用对象的置空或者少用静态引用；
使用静态内部类+软引用代替非静态内部类；
及时取消广播或者观察者注册；
耗时任务、属性动画在 Activity 销毁时记得 cancel；
文件流、Cursor 等资源及时关闭；
Activity 销毁时 WebView 的移除和销毁。