常见的八种导致 APP 内存泄漏的问题

像 Java 这样具有垃圾回收功能的语言的好处之一，就是程序员无需手动管理内存分配。这减少了段错误（segmentation fault）导致的闪退，也减少了内存泄漏导致的堆空间膨胀，让编写的代码更加安全。然而，Java 中依然有可能发生内存泄漏。所以你的安卓 APP 依然有可能浪费了大量的内存，甚至由于内存耗尽（OOM）导致闪退。

传统的内存泄漏是由忘记释放分配的内存导致的，而逻辑上的内存泄漏则是由于忘记在对象不再被使用的时候释放对其的引用导致的。如果一个对象仍然存在强引用，垃圾回收器就无法对其进行垃圾回收。在安卓平台，泄漏 Context 对象问题尤其严重。这是因为像 Activity 这样的 Context 对象会引用大量很占用内存的对象，例如 View 层级，以及其他的资源。如果 Context 对象发生了内存泄漏，那它引用的所有对象都被泄漏了。安卓设备大多内存有限，如果发生了大量这样的内存泄漏，那内存将很快耗尽。

如果一个对象的合理生命周期没有清晰的定义，那判断逻辑上的内存泄漏将是一个见仁见智的问题。幸运的是，activity 有清晰的生命周期定义，使得我们可以很明确地判断 activity 对象是否被内存泄漏。onDestroy() 函数将在 activity 被销毁时调用，无论是程序员主动销毁 activity，还是系统为了回收内存而将其销毁。如果 onDestroy 执行完毕之后，activity 对象仍被 heap root 强引用，那垃圾回收器就无法将其回收。所以我们可以把生命周期结束之后仍被引用的 activity 定义为被泄漏的 activity。

Activity 是非常重量级的对象，所以我们应该极力避免妨碍系统对其进行回收。然而有多种方式会让我们无意间就泄露了 activity 对象。我们把可能导致 activity 泄漏的情况分为两类，一类是使用了进程全局（process-global）的静态变量，无论 APP 处于什么状态，都会一直存在，它们持有了对 activity 的强引用进而导致内存泄漏，另一类是生命周期长于 activity 的线程，它们忘记释放对 activity 的强引用进而导致内存泄漏。下面我们就来详细分析一下这些可能导致 activity 泄漏的情况。

1. 静态 Activity

泄漏 activity 最简单的方法就是在 activity 类中定义一个 static 变量，并且将其指向一个运行中的 activity 实例。 如果在 activity 的生命周期结束之前，没有清除这个引用，那它就会泄漏了。这是因为 activity（例如 MainActivity）的类对象是静态的，一旦加载，就会在 APP 运行时一直常驻内存，因此如果类对象不卸载，其静态成员就不会被垃圾回收。

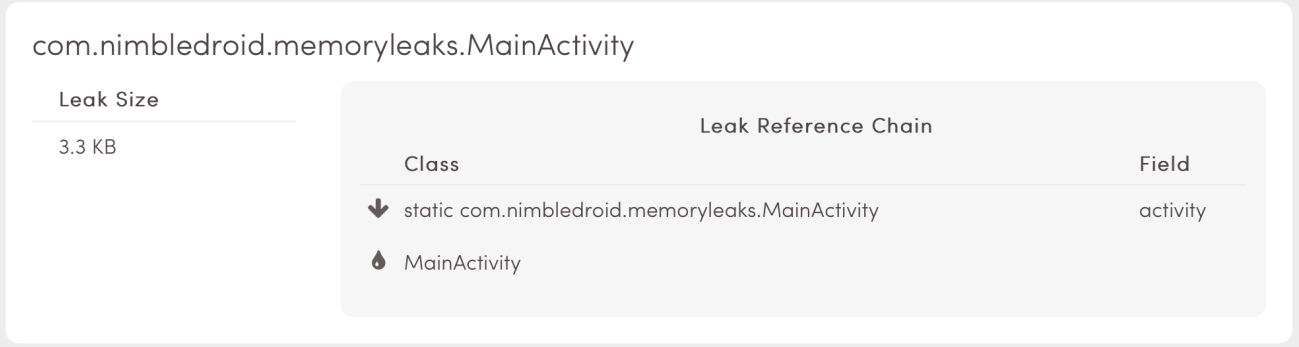
[代码]xml代码：

?

```
01
02
03
04
05
06
07
08
09
10
11
```

```
void setStaticActivity() {
  activity = this;
}

View saButton = findViewById(R.id.sa_button);
saButton.setOnClickListener(new View.OnClickListener() {
  @Override public void onClick(View v) {
    setStaticActivity();
    nextActivity();
  }
});
```



内存泄漏场景 1 - Static Activity

2. 静态 View

另一种类似的情况是对经常启动的 activity 实现一个单例模式，让其常驻内存可以使它能够快速恢复状态。然而，就像前文所述，不遵循系统定义的 activity 生命周期是非常危险的，也是没必要的，所以我们应该极力避免。

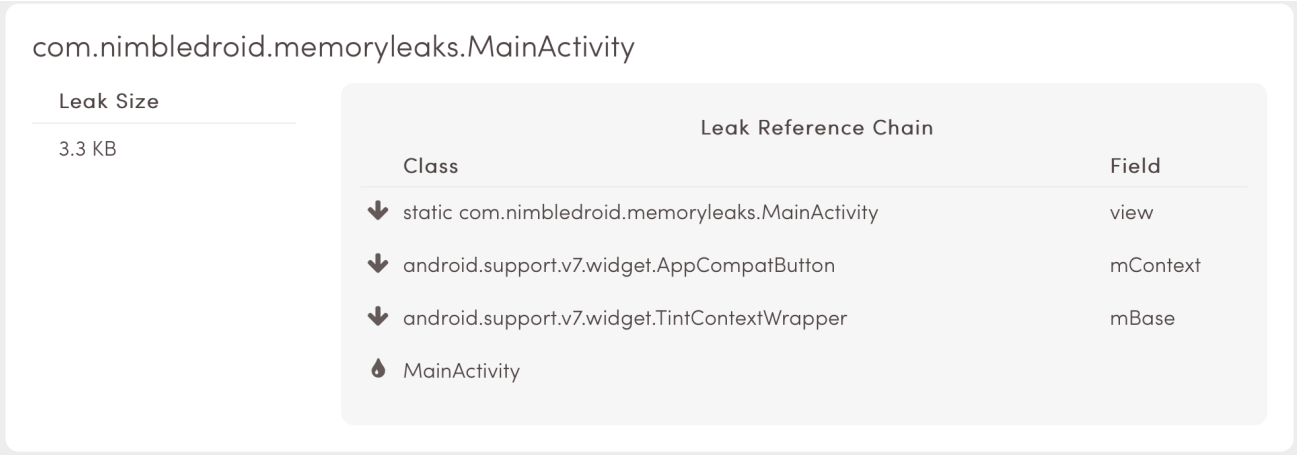
但是如果我们有一个创建起来非常耗时的 View，在同一个 activity 不同的生命周期中都保持不变呢？所以让我们为它实现一个单例模式，就像这段代码。现在一旦 activity 被销毁，那我们就应该释放大部分的内存了。

[代码]java代码：

?

```
void setStaticView() {
  view = findViewById(R.id.sv_button);
}

View svButton = findViewById(R.id.sv_button);
svButton.setOnClickListener(new View.OnClickListener() {
  @Override public void onClick(View v) {
    setStaticView();
    nextActivity();
  }
});
```



内存泄漏场景 2 - Static View

内存泄漏了！因为一旦 view 被加入到界面中，它就会持有 context 的强引用，也就是我们的 activity。由于我们通过一个静态成员引用了这个 view，所以我们也就引用了 activity，因此 activity 就发生了泄漏。所以一定不要把加载的 view 赋值给静态变量，如果你真的需要，那一定要确保在 activity 销毁之前将其从 view 层级中移除。

3. 内部类

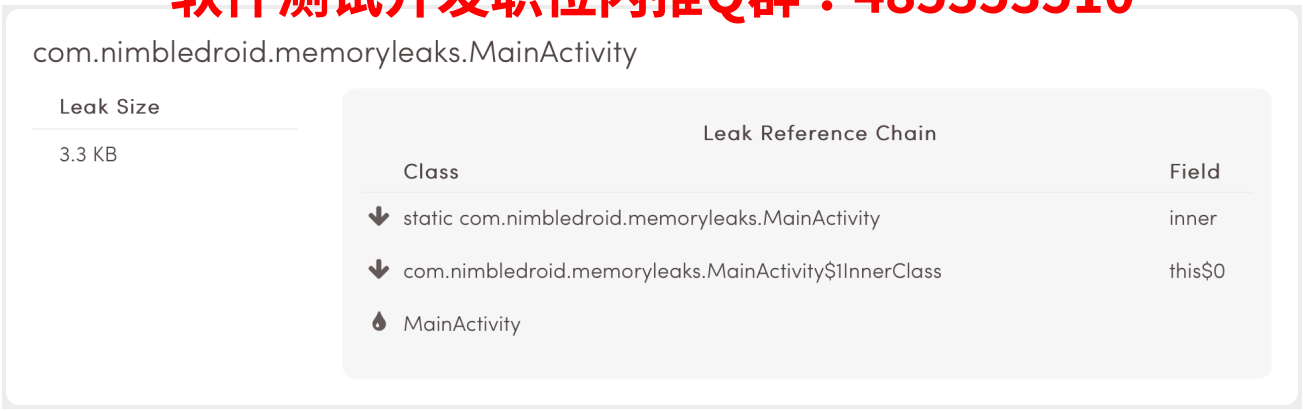
现在让我们在 activity 内部定义一个类，也就是内部类。这样做的原因有很多，比如增加封装性和可读性。如果我们创建了一个内部类的对象，并且通过静态变量持有了 activity 的引用，那也会发生 activity 泄漏。

[代码]java代码：

?

```
void createInnerClass() {
    class InnerClass {
    }
    inner = new InnerClass();
}

View icButton = findViewById(R.id.ic_button);
icButton.setOnClickListener(new View.OnClickListener() {
    @Override public void onClick(View v) {
        createInnerClass();
        nextActivity();
    }
});
```



内存泄漏场景 3 - Inner Class

不幸的是，内部类能够引用外部类的成员这一优势，就是通过持有外部类的引用来实现的，而这正是 activity 泄漏的原因。

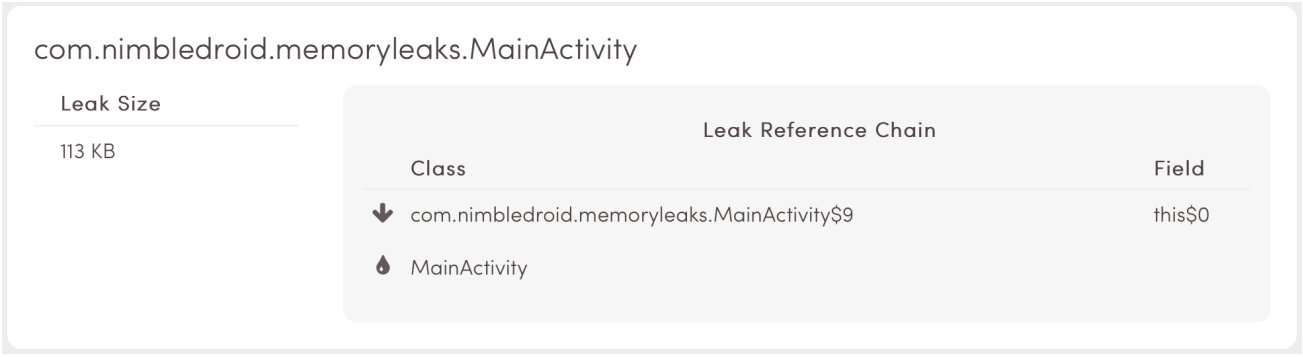
4. 匿名类

类似的，匿名类同样会持有定义它们的对象的引用。因此如果在 activity 内定义了一个匿名的 AsyncTask 对象，就有可能发生内存泄漏了。如果 activity 被销毁之后 AsyncTask 仍然在执行，那就会组织垃圾回收器回收 activity 对象，进而导致内存泄漏，直到执行结束才能回收 activity。

[代码]java代码：

?

```
01
02
03 void startAsyncTask() {
04     new AsyncTask<void, void,="" void="">() {
05         @Override protected Void doInBackground(Void... params) {
06             while(true);
07         }
08     }.execute();
09 }
10
11 super.onCreate(savedInstanceState);
12 setContentView(R.layout.activity_main);
13 View aicButton = findViewById(R.id.at_button);
14 aicButton.setOnClickListener(new View.OnClickListener() {
15     @Override public void onClick(View v) {
16         startAsyncTask();
17         nextActivity();
18     }
});
</void,>
```



内存泄漏场景 4 - AsyncTask

5. Handlers

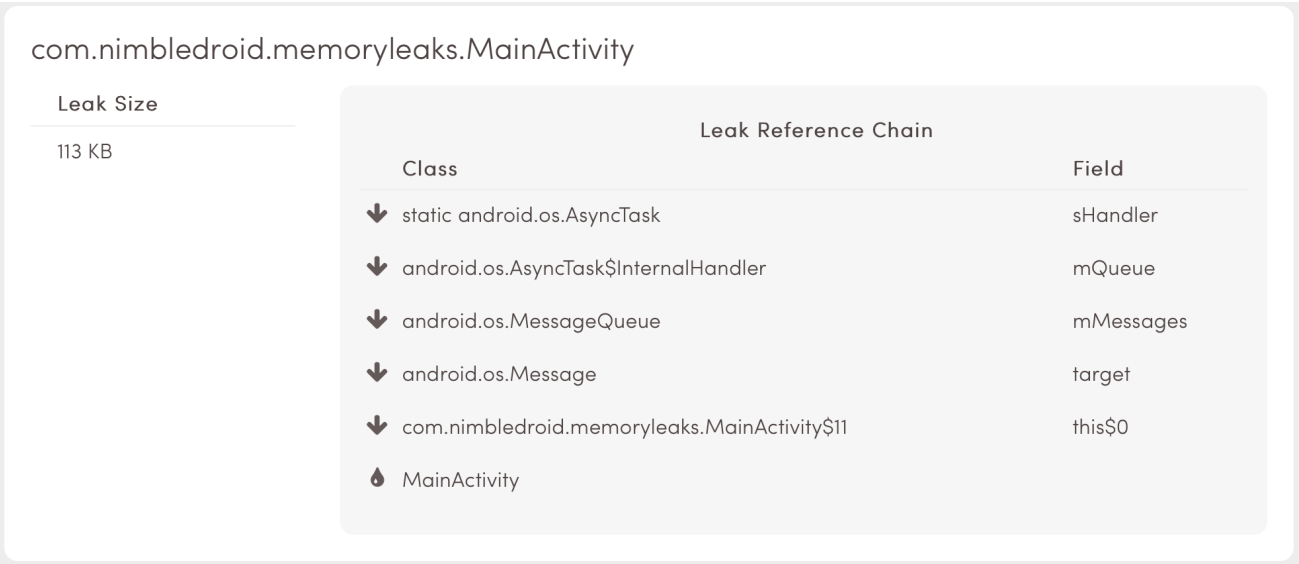
同 样的，定义一个匿名的 Runnable 对象并将其提交到 Handler 上也可能导致 activity 泄漏。Runnable 对象间接地引用了定义它的 activity 对象，而它会被提交到 Handler 的 MessageQueue 中，如果它在 activity 销毁时还没有被处理，那就会导致 activity 泄漏了。

[代码]java代码：

?

```
void createHandler() {
    new Handler() {
        @Override public void handleMessage(Message message) {
            super.handleMessage(message);
        }
    }.postDelayed(new Runnable() {
        @Override public void run() {
            while(true);
        }
    }, Long.MAX_VALUE >> 1);
}


View hButton = findViewById(R.id.h_button);
hButton.setOnClickListener(new View.OnClickListener() {
    @Override public void onClick(View v) {
        createHandler();
        nextActivity();
    }
});
```



内存泄漏场景 5 - Handler

6. Threads

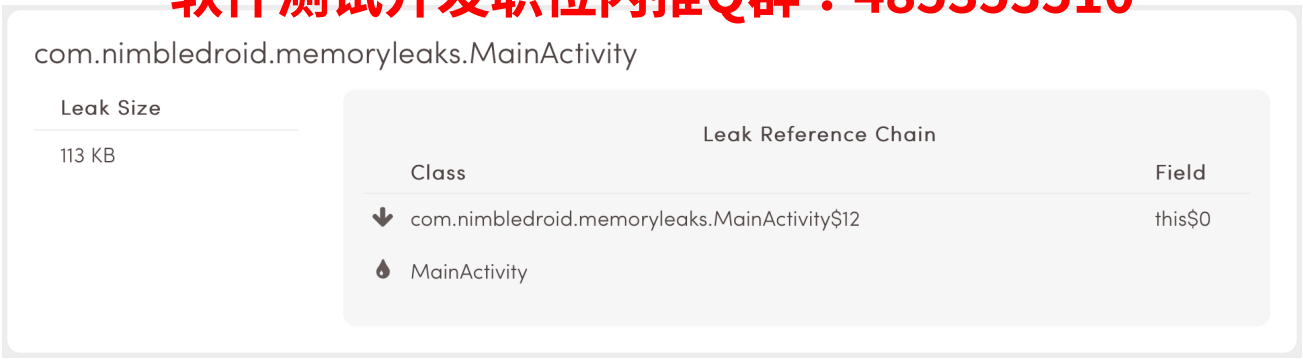
同样的，使用 Thread 和 TimerTask 也可能导致 activity 泄漏。

[代码]java代码：

?

```
void spawnThread() {
    new Thread() {
        @Override public void run() {
            while(true);
        }
    }.start();
}

View tButton = findViewById(R.id.t_button);
tButton.setOnClickListener(new View.OnClickListener() {
  @Override public void onClick(View v) {
      spawnThread();
      nextActivity();
  }
});
```



内存泄漏场景 6 - Thread

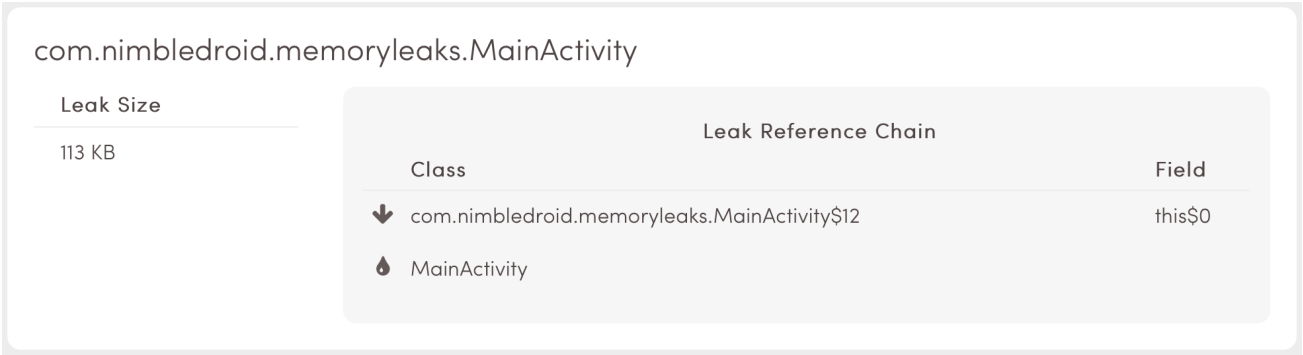
7. Timer Tasks

只要它们是通过匿名类创建的，尽管它们在单独的线程被执行，它们也会持有对 activity 的强引用，进而导致内存泄漏。

[代码]java代码：

?

```
01
02
03  void scheduleTimer() {
04      new Timer().schedule(new TimerTask() {
05          @Override
06          public void run() {
07              while(true);
08          }
09      }, Long.MAX_VALUE >> 1);
10  }
11
12  View ttButton = findViewById(R.id.tt_button);
13  ttButton.setOnClickListener(new View.OnClickListener() {
14      @Override public void onClick(View v) {
15          scheduleTimer();
16          nextActivity();
      }
  });
```



内存泄漏场景 7 - TimerTask

8. Sensor Manager

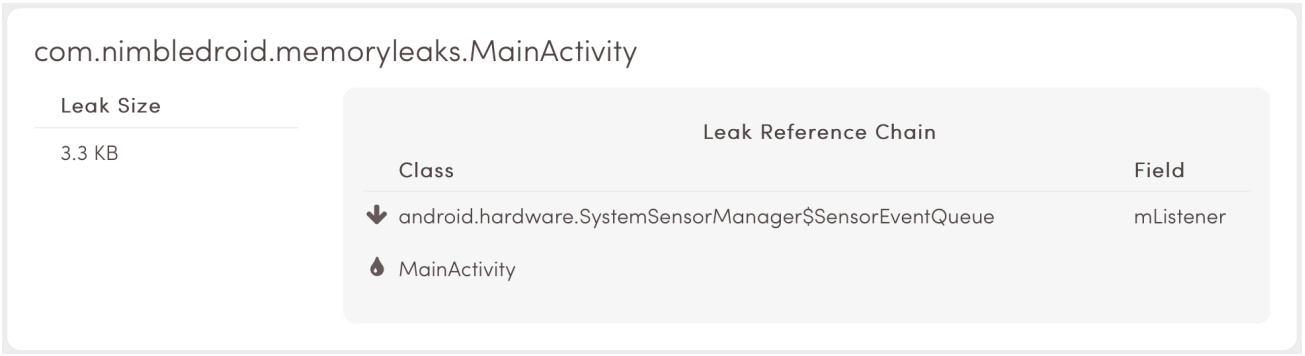
最后，系统服务可以通过 context.getSystemService 获取，它们负责执行某些后台任务，或者为硬件访问提供接口。如果 context 对象想要在服务内部的事件发生时被通知，那就需要把自己注册到服务的监听器中。然而，这会让服务持有 activity 的引用，如果程序员忘记在 activity 销毁时取消注册，那就会导致 activity 泄漏了。

[代码]java代码：

?

```
void registerListener() {
        SensorManager sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
        Sensor sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ALL);
        sensorManager.registerListener(this, sensor, SensorManager.SENSOR_DELAY_FASTEST);
}

View smButton = findViewById(R.id.sm_button);
smButton.setOnClickListener(new View.OnClickListener() {
    @Override public void onClick(View v) {
        registerListener();
        nextActivity();
    }
});
```



内存泄漏场景 8 - Sensor Manager

现在，我们展示了八种很容易不经意间就泄漏大量内存的情景。请记住，最坏的情况下，你的 APP 可能会由于大量的内存泄漏而内存耗尽，进而闪退，但它并不总是这样。相反，内存泄漏会消耗大量的内存，但却不至于内存耗尽，这时，APP 会由于内存不够分配而频繁进行垃圾回收。垃圾回收是非常耗时的操作，会导致严重的卡顿。在 activity 内部创建对象时，一定要格外小心，并且要经常测试是否存在内存泄漏。