LeakCanary的工作过程以及原理

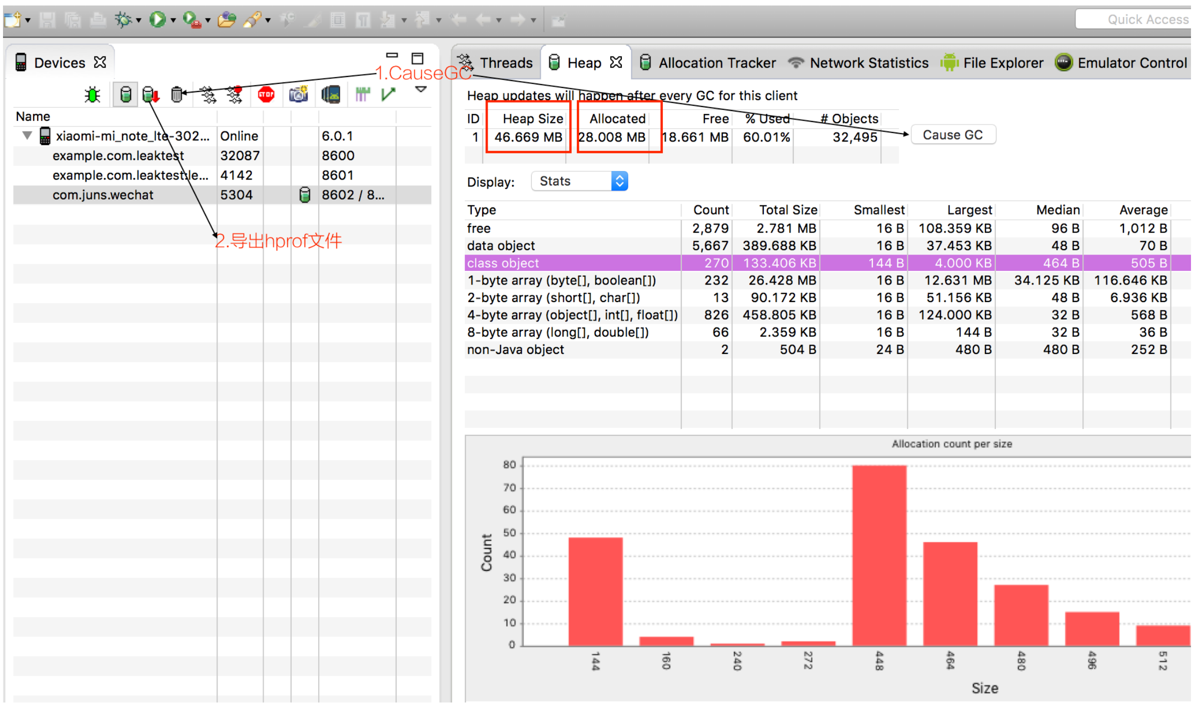
曾经检测内存泄露的方式

让我们来看看在没有LeakCanary之前，我们怎么来检测内存泄露

1. Bug收集通过Bugly、友盟这样的统计平台，统计Bug，了解OutOfMemaryError的情况。
2. 重现问题对Bug进行筛选，归类，排除干扰项。然后为了重现问题，有时候你必须找到出现问题的机型，因为有些问题只会在特定的设备上才会出现。为了找到特定的机型，可能会想尽一切办法，去买、去借、去求人（14年的时候，上家公司专门派了一个商务去广州找了一家租赁手机的公司，借了50台手机回来，600块钱一天）。然后，为了重现问题，一遍一遍的尝试，去还原当时OutOfMemaryError出现的原因，用最原始、最粗暴的方式。
3. Dump导出hprof文件使用Eclipse ADT的DDMS，观察Heap，然后点击手动GC按钮(Cause GC)，观察内存增长情况，导出hprof文件。 主要观测的两项数据：

3.1 Heap Size的大小，当资源增加到堆空余空间不够的时候，系统会增加堆空间的大小，但是超过可分配的最大值（比如手机给App分配的最大堆空间为128M）就会发生OutOfMemaryError,这个时候进程就会被杀死。这个最大堆空间，不同手机会有不同的值，跟手机内存大小和厂商定制过后的系统存在关联。

3.2 Allocated堆中已分配的大小，这是应用程序实际占用的大小，资源回收后，这项数据会变小。 查看操作前后的堆数据，看是否存在内存泄露，比如反复打开、关闭一个页面，看看堆空间是否会一直增大。



4. 然后使用MAT内存分析工具打开，反复查看找到那些原本应该被回收掉的对象。
5. 计算这个对象到GC roots的最短强引用路径。
6. 确定那个路径中那个引用不该有，然后修复问题。

很麻烦，不是吗。现在有一个类库可以直接解决这个问题

LeakCanary

使用方式

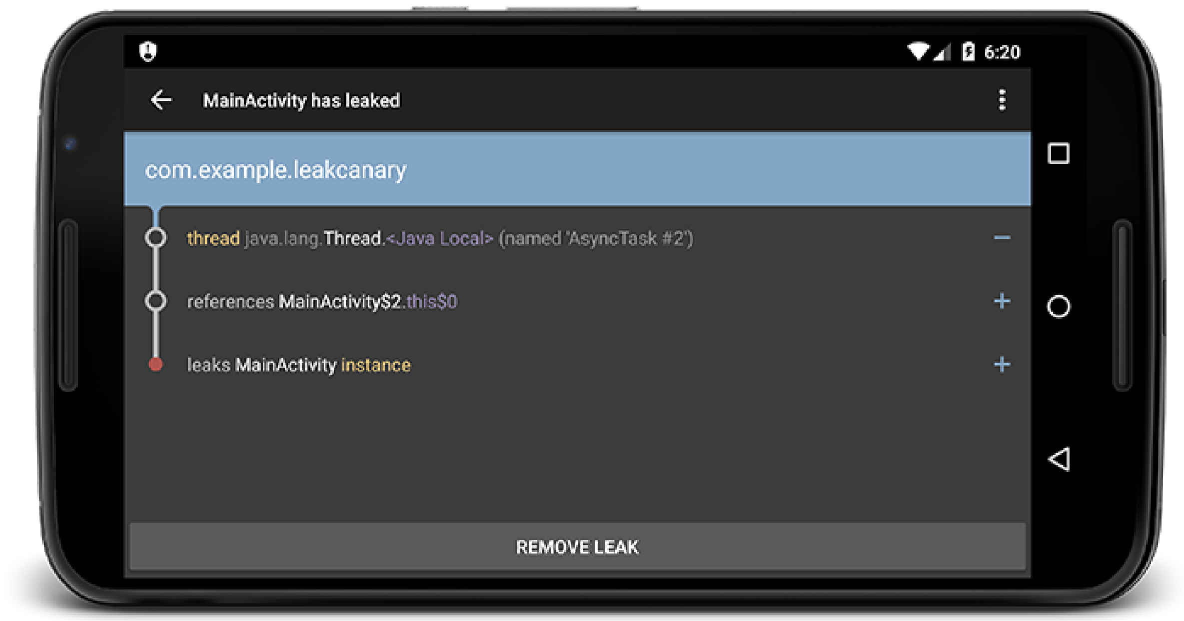
使用AndroidStudio，在Module.app的build.gradle中引入

```
dependencies {
   debugCompile 'com.squareup.leakcanary:leakcanary-android:1.4-beta2'
   releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.4-beta2'
   testCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.4-beta2'
 }
```

然后在Application中重写onCreate()方法

```
public class ExampleApplication extends Application {
  @Override public void onCreate() {
    super.onCreate();
    LeakCanary.install(this);
  }
}
```

在Activity中写一些导致内存泄露的代码，当发生内存泄露了，会在通知栏弹出消息，点击跳转到泄露页面



LeakCanary 可以做到非常简单方便、低侵入性地捕获内存泄漏代码，甚至很多时候你可以捕捉到 Android 系统组件的内存泄漏代码，最关键是不用再进行（捕获错误+Bug归档+场景重现+Dump+Mat分析） 这一系列复杂操作，6得不行。

原理分析

如果我们自己实现

首先，设想如果让我们自己来实现一个LeakCanary，我们怎么来实现。
按照前面说的曾经检测内存的方式，我想，大概需要以下几个步骤：

1. 检测一个对象，查看他是否被回收了。
2. 如果没有被回收，使用DDMS的dump导出.hprof文件，确定是否内存泄露，如果泄露了导出最短引用路径
3. 把最短引用路径封装到一个对象中，用Intent发送给Notification，然后点击跳转到展示页，页面展示

检测对象，是否被回收

我们来看看，LeakCanary是不是按照这种方式实现的。除了刚才说的只需要在Application中的onCreate方法注册LeakCanary.install(this);这种方式。 查看源码，使用官方给的Demo示例代码中，我们发现有一个RefWatcher对象，也可以用来监测，看看它是如何使用的。 MainActivity.class

```
@Override
    protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    RefWatcher refWatcher = LeakCanary.androidWatcher(getApplicationContext(),
                    new ServiceHeapDumpListener(getApplicationContext(),
DisplayLeakService.class),
                    AndroidExcludedRefs.createAppDefaults().build());
    refWatcher.watch(this);
}
```

就是把MainActivity作为一个对象监测起来，查看`refWatcher.watch(this)`的实现

```
public void watch(Object watchedReference) {
    watch(watchedReference, "");
  }

  /**
   * Watches the provided references and checks if it can be GCed. This method is
non blocking,
   * the check is done on the {@link Executor} this {@link RefWatcher} has been
constructed with.
   *
   * @param referenceName An logical identifier for the watched object.
   */
  public void watch(Object watchedReference, String referenceName) {
    Preconditions.checkNotNull(watchedReference, "watchedReference");
    Preconditions.checkNotNull(referenceName, "referenceName");
    if (debuggerControl.isDebuggerAttached()) {
      return;
    }
    final long watchStartNanoTime = System.nanoTime();
    String key = UUID.randomUUID().toString();
    retainedKeys.add(key);
    final KeyedWeakReference reference =
        new KeyedWeakReference(watchedReference, key, referenceName, queue);

    watchExecutor.execute(new Runnable() {
      @Override public void run() {
        ensureGone(reference, watchStartNanoTime);
      }
    });
  }
```

可以总结出他的实现步骤如下：

1. 先检查监测对象是否为空，为空抛出异常

2. 如果是在调试Debugger过程中允许内存泄露出现，不再监测。因为这个时候监测的对象是不准确的，而且会干扰我们调试代码。
3. 给监测对象生成UUID唯一标识符，存入Set集合，方便查找。
4. 然后定义了一个KeyedWeakReference，查看下KeyedWeakReference是个什么玩意

```
public final class KeyedWeakReference extends WeakReference<Object> {
  public final String key;
  public final String name;

  KeyedWeakReference(Object referent, String key, String name,
      ReferenceQueue<Object> referenceQueue) {
    super(Preconditions.checkNotNull(referent, "referent"),
Preconditions.checkNotNull(referenceQueue, "referenceQueue"));
    this.key = Preconditions.checkNotNull(key, "key");
    this.name = Preconditions.checkNotNull(name, "name");
  }
}
```

原来KeyedWeakReference就是对WeakReference进行了一些加工，是一种装饰设计模式，其实就是弱引用的衍生类。配合前面的Set retainedKeys使用，retainedKeys代表的是没有被GC回收的对象，referenceQueue中的弱引用代表的是被GC了的对象，通过这两个结构就可以明确知道一个对象是不是被回收了。（一个对象在referenceQueue可以找到当时在retainedKeys中找不到，那么肯定被回收了，没有内存泄漏一说）

5. 接着看上面的执行过程，然后通过线程池开启了一个异步任务方法ensureGone。watchExecutor 看看这个实体的类实现—AndroidWatchExecutor，查看源码

```
public final class AndroidWatchExecutor implements Executor {

  static final String LEAK_CANARY_THREAD_NAME = "LeakCanary-Heap-Dump";
  private static final int DELAY_MILLIS = 5000;

  private final Handler mainHandler;
  private final Handler backgroundHandler;

  public AndroidWatchExecutor() {
    mainHandler = new Handler(Looper.getMainLooper());
    HandlerThread handlerThread = new HandlerThread(LEAK_CANARY_THREAD_NAME);
    handlerThread.start();
    backgroundHandler = new Handler(handlerThread.getLooper());
  }

  @Override public void execute(final Runnable command) {
    if (isOnMainThread()) {
      executeDelayedAfterIdleUnsafe(command);
    } else {
      mainHandler.post(new Runnable() {
        @Override public void run() {
          executeDelayedAfterIdleUnsafe(command);
        }
      });
    }
  }
```

```
  private boolean isOnMainThread() {
    return Looper.getMainLooper().getThread() == Thread.currentThread();
  }

  private void executeDelayedAfterIdleUnsafe(final Runnable runnable) {
    // This needs to be called from the main thread.
    Looper.myQueue().addIdleHandler(new MessageQueue.IdleHandler() {
      @Override public boolean queueIdle() {
        backgroundHandler.postDelayed(runnable, DELAY_MILLIS);
        return false;
      }
    });
  }
}
```

做得事情就是，通过主线程的mainHandler转发到后台backgroundHandler执行任务，后台线程延迟DELAY_MILLIS这么多时间执行

6. 具体执行的任务在ensureGone()方法里面

```
void ensureGone(KeyedWeakReference reference, long watchStartNanoTime) {
    long gcStartNanoTime = System.nanoTime();
    //记录观测对象的时间
    long watchDurationMs = NANOSECONDS.toMillis(gcStartNanoTime -
watchStartNanoTime);
    //清除在queue中的弱引用 保留retainedKeys中剩下的对象
    removeWeaklyReachableReferences();
    //如果剩下的对象中不包含引用对象，说明已被回收，返回||调试中,返回
    if (gone(reference) || debuggerControl.isDebuggerAttached()) {
      return;
    }
    //请求执行GC
    gcTrigger.runGc();
    //再次清理一次对象
    removeWeaklyReachableReferences();
    if (!gone(reference)) {
      long startDumpHeap = System.nanoTime();
      //记录下GC执行时间
      long gcDurationMs = NANOSECONDS.toMillis(startDumpHeap - gcStartNanoTime);
      //Dump导出hprof文件
      File heapDumpFile = heapDumper.dumpHeap();

      if (heapDumpFile == null) {
        // Could not dump the heap, abort.
        return;
      }
      //记录下Dump和文件导出用的时间
      long heapDumpDurationMs = NANOSECONDS.toMillis(System.nanoTime() -
startDumpHeap);
      //分析hprof文件
      heapdumpListener.analyze(
          new HeapDump(heapDumpFile, reference.key, reference.name, excludedRefs,
watchDurationMs,
              gcDurationMs, heapDumpDurationMs));
    }
  }
```

```
  private boolean gone(KeyedWeakReference reference) {
    return !retainedKeys.contains(reference.key);
  }
```

```
private void removeWeaklyReachableReferences() {
    // WeakReferences are enqueued as soon as the object to which they point to
becomes weakly
    // reachable. This is before finalization or garbage collection has actually
happened.
    KeyedWeakReference ref;
    while ((ref = (KeyedWeakReference) queue.poll()) != null) {
      retainedKeys.remove(ref.key);
    }
  }
```

这里我们思考两个问题：

1. retainedKeys和queue怎么关联起来的？这里的removeWeaklyReachableReferences方法就实现了我们说的 retainedKeys代表的是没有被GC回收的对象，queue中的弱引用代表的是被GC了的对象，之间的关联， 一个对象在queue可以找到当时在retainedKeys中找不到，那么肯定被回收了。gone()返回true说明对象已被回收，不需要观测了。
2. 为什么执行removeWeaklyReachableReferences()两次？为了保证效率，如果对象被回收，没必要再通知GC执行，Dump操作等等一系列繁琐步骤，况且GC是一个线程优先级极低的线程，就算你通知了，她也不一定会执行，基于这一点，我们分析的观测对象的时机就显得尤为重要了，在对象被回收的时候召唤观测。

何时执行观测对象

我们观测的是一个Activity，Activity这样的组件都存在生命周期，在他生命周期结束的时，观测他如果还存活的话 就肯定就存在内存泄露了，进一步推论，Activity的生命周期结束就关联到它的onDestory()方法，也就是只要重写这个方法就可以了。

```
@Override
    protected void onDestroy() {
    super.onDestroy();
    refWatcher.watch(this);
}
```

在MainActivity中加上这行代码就好了，但是我们显然不想每个Activity都这样干，都是同样的代码为啥要重复着写，当然解决办法呼之欲出：

```
private final Application.ActivityLifecycleCallbacks lifecycleCallbacks =
      new Application.ActivityLifecycleCallbacks() {
        @Override public void onActivityCreated(Activity activity, Bundle
savedInstanceState) {
        }

        @Override public void onActivityStarted(Activity activity) {
        }

        @Override public void onActivityResumed(Activity activity) {
        }

        @Override public void onActivityPaused(Activity activity) {
        }
```

```
        @Override public void onActivityStopped(Activity activity) {
        }

        @Override public void onActivitySaveInstanceState(Activity activity,
Bundle outState) {
        }

        @Override public void onActivityDestroyed(Activity activity) {
          ActivityRefWatcher.this.onActivityDestroyed(activity);
        }
      };
    void onActivityDestroyed(Activity activity) {
        refWatcher.watch(activity);
    }
```

LeakCanary源码是这样做的，通过ActivityLifecycleCallbacks转发，然后在install()中使用这个接口，这就实现了我们只需要调用LeakCanary.install(this);这句代码在Application中就可以实现监测

```
public final class LeakCanary {
    public static RefWatcher install(Application application) {
        return install(application, DisplayLeakService.class,
AndroidExcludedRefs.createAppDefaults().build());
    }
    public static RefWatcher install(Application application, Class<? extends
AbstractAnalysisResultService> listenerServiceClass, ExcludedRefs excludedRefs)
{
        if(isInAnalyzerProcess(application)) {
            return RefWatcher.DISABLED;
        } else {
            enableDisplayLeakActivity(application);
            ServiceHeapDumpListener heapDumpListener = new
ServiceHeapDumpListener(application, listenerServiceClass);
            RefWatcher refWatcher = androidWatcher(application,
heapDumpListener, excludedRefs);
            ActivityRefWatcher.installOnIcsPlus(application, refWatcher);
            return refWatcher;
        }
    }
```

不需要在每个Activity方法的结束再多写几行onDestroy()代码，但是这个方法有个缺点，看注释

// If you need to support Android < ICS, override onDestroy() in your base activity.

```
//ICS
October 2011: Android 4.0.
        public static final int ICE_CREAM_SANDWICH = 14;
```

如果是SDK 14 android 4.0以下的系统，不具备这个接口，也就是还是的通过刚才那种方式重写onDestory()方法。而且只实现了ActivityRefWatcher.installOnIcsPlus(application, refWatcher);对Activity进行监测，如果是服务或者广播还需要我们自己实现

分析hprof文件

接着分析，查看文件解析类发现他是个转发工具类

```
public final class ServiceHeapDumpListener implements HeapDump.Listener {
  ...
  @Override public void analyze(HeapDump heapDump) {
      Preconditions.checkNotNull(heapDump, "heapDump");
     //转发给HeapAnalyzerService
    HeapAnalyzerService.runAnalysis(context, heapDump, listenerServiceClass);
  }
}
```

通过IntentService运行在另一个进程中执行分析任务

```
public final class HeapAnalyzerService extends IntentService {

  private static final String LISTENER_CLASS_EXTRA = "listener_class_extra";
  private static final String HEAPDUMP_EXTRA = "heapdump_extra";

  public static void runAnalysis(Context context, HeapDump heapDump,
      Class<? extends AbstractAnalysisResultService> listenerServiceClass) {
    Intent intent = new Intent(context, HeapAnalyzerService.class);
    intent.putExtra(LISTENER_CLASS_EXTRA, listenerServiceClass.getName());
    intent.putExtra(HEAPDUMP_EXTRA, heapDump);
    context.startService(intent);
  }

  @Override protected void onHandleIntent(Intent intent) {
    String listenerClassName = intent.getStringExtra(LISTENER_CLASS_EXTRA);
    HeapDump heapDump = (HeapDump) intent.getSerializableExtra(HEAPDUMP_EXTRA);

    ExcludedRefs androidExcludedDefault = createAndroidDefaults().build();
    HeapAnalyzer heapAnalyzer = new HeapAnalyzer(androidExcludedDefault,
heapDump.excludedRefs);
    //获取分析结果
    AnalysisResult result = heapAnalyzer.checkForLeak(heapDump.heapDumpFile,
heapDump.referenceKey);
    AbstractAnalysisResultService.sendResultToListener(this, listenerClassName,
heapDump, result);
  }
}
```

查看heapAnalyzer.checkForLeak代码

```
public AnalysisResult checkForLeak(File heapDumpFile, String referenceKey) {
    long analysisStartNanoTime = System.nanoTime();

    if (!heapDumpFile.exists()) {
      Exception exception = new IllegalArgumentException("File does not exist: "
+ heapDumpFile);
      return AnalysisResult.failure(exception, since(analysisStartNanoTime));
    }
```

```
    ISnapshot snapshot = null;
    try {
      // 加载hprof文件
      snapshot = openSnapshot(heapDumpFile);
      //找到泄露对象
      IObject leakingRef = findLeakingReference(referenceKey, snapshot);

      // False alarm, weak reference was cleared in between key check and heap
dump.
      if (leakingRef == null) {
        return AnalysisResult.noLeak(since(analysisStartNanoTime));
      }

      String className = leakingRef.getClazz().getName();
      // 最短引用路径
      AnalysisResult result =
          findLeakTrace(analysisStartNanoTime, snapshot, leakingRef, className,
true);
      //如果没找到  尝试排除系统进程干扰的情况下找出最短引用路径
      if (!result.leakFound) {
        result = findLeakTrace(analysisStartNanoTime, snapshot, leakingRef,
className, false);
      }

      return result;
    } catch (SnapshotException e) {
      return AnalysisResult.failure(e, since(analysisStartNanoTime));
    } finally {
      cleanup(heapDumpFile, snapshot);
    }
  }
```

到这里，我们就找到了泄露对象的最短引用路径，剩下的工作就是发送消息给通知，然后点击通知栏跳转到我们另一个App打开绘制出路径即可。

补充—排除干扰项

但是我们在找出最短引用路径的时候，有这样一段代码，他是干什么的呢

```
// 最短引用路径
      AnalysisResult result =
          findLeakTrace(analysisStartNanoTime, snapshot, leakingRef, className,
true);
      //如果没找到  尝试排除系统进程干扰的情况下找出最短引用路径
      if (!result.leakFound) {
        result = findLeakTrace(analysisStartNanoTime, snapshot, leakingRef,
className, false);
```

查看findLeakTrace()

```
private AnalysisResult findLeakTrace(long analysisStartNanoTime, ISnapshot
snapshot,
      IObject leakingRef, String className, boolean excludingKnownLeaks) throws
SnapshotException {
```

```
    ExcludedRefs excludedRefs = excludingKnownLeaks ? this.excludedRefs :
baseExcludedRefs;

    PathsFromGCRootsTree gcRootsTree = shortestPathToGcRoots(snapshot,
leakingRef, excludedRefs);

    // False alarm, no strong reference path to GC Roots.
    if (gcRootsTree == null) {
      return AnalysisResult.noLeak(since(analysisStartNanoTime));
    }

    LeakTrace leakTrace = buildLeakTrace(snapshot, gcRootsTree, excludedRefs);

    return AnalysisResult.leakDetected(!excludingKnownLeaks, className,
leakTrace, since(analysisStartNanoTime));
  }
```

唯一的不同是excludingKnownLeaks 从字面意思也很好理解，是否排除已知内存泄露

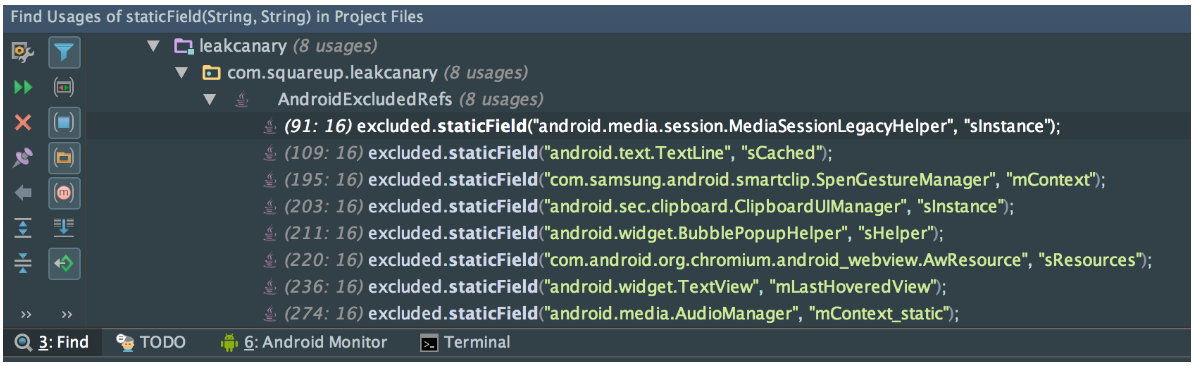
其实是这样的，在我们系统中本身就存在一些内存泄露的情况，这是上层App工程师无能为力的。但是如果是厂商或者做Android Framework层的工程师可能需要关心这个，于是做成一个参数配置的方式，让我们灵活选择岂不妙哉。当然，默认是会排除系统自带泄露情况的，不然打开App，弹出一堆莫名其妙的内存泄露，我们还无能为力，着实让人惶恐，而且我们还可以自己配置。

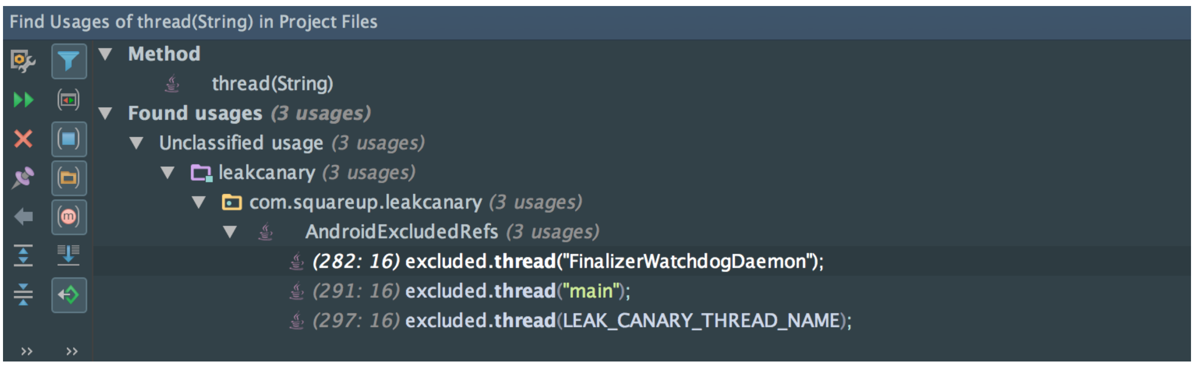
通过ExcludedRefs这个类

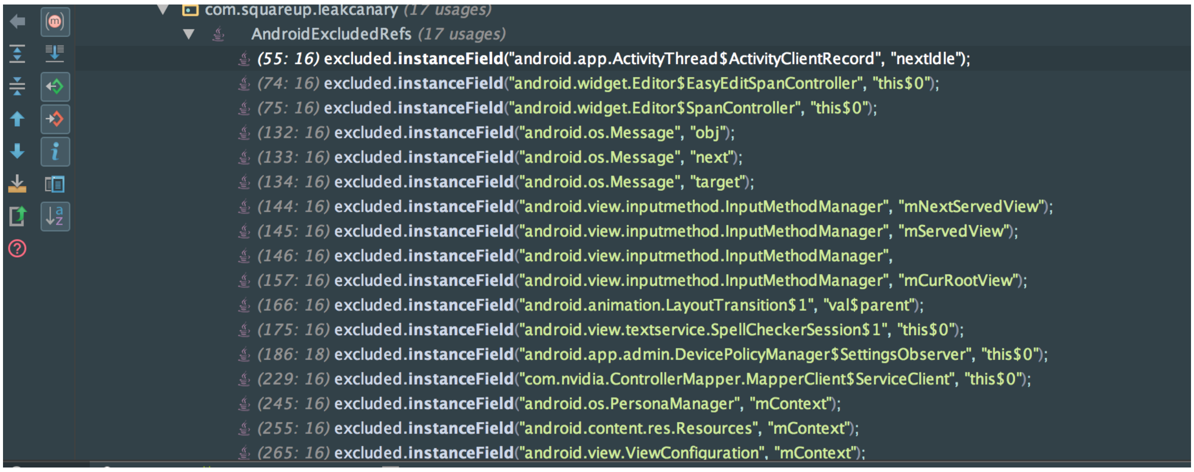
```
public final class ExcludedRefs implements Serializable {

  public final Map<String, Set<String>> excludeFieldMap;
  public final Map<String, Set<String>> excludeStaticFieldMap;
  public final Set<String> excludedThreads;

  private ExcludedRefs(Map<String, Set<String>> excludeFieldMap,
      Map<String, Set<String>> excludeStaticFieldMap, Set<String>
excludedThreads) {
    // Copy + unmodifiable.
    this.excludeFieldMap = unmodifiableMap(new LinkedHashMap<String, Set<String>>
(excludeFieldMap));
    this.excludeStaticFieldMap = unmodifiableMap(new LinkedHashMap<String,
Set<String>>(excludeStaticFieldMap));
    this.excludedThreads = unmodifiableSet(new LinkedHashSet<String>
(excludedThreads));
  }

  public static final class Builder {
    private final Map<String, Set<String>> excludeFieldMap = new
LinkedHashMap<String, Set<String>>();
    private final Map<String, Set<String>> excludeStaticFieldMap = new
LinkedHashMap<String, Set<String>>();
    private final Set<String> excludedThreads = new LinkedHashSet<String>();

    public Builder instanceField(String className, String fieldName) {
        Preconditions.checkNotNull(className, "className");
      Preconditions.checkNotNull(fieldName, "fieldName");
      Set<String> excludedFields = excludeFieldMap.get(className);
      if (excludedFields == null) {
        excludedFields = new LinkedHashSet<String>();
```

```
        excludeFieldMap.put(className, excludedFields);
      }
      excludedFields.add(fieldName);
      return this;
    }

    public Builder staticField(String className, String fieldName) {
        Preconditions.checkNotNull(className, "className");
        Preconditions.checkNotNull(fieldName, "fieldName");
      Set<String> excludedFields = excludeStaticFieldMap.get(className);
      if (excludedFields == null) {
        excludedFields = new LinkedHashSet<String>();
        excludeStaticFieldMap.put(className, excludedFields);
      }
      excludedFields.add(fieldName);
      return this;
    }

    public Builder thread(String threadName) {
        Preconditions.checkNotNull(threadName, "threadName");
      excludedThreads.add(threadName);
      return this;
    }

    public ExcludedRefs build() {
      return new ExcludedRefs(excludeFieldMap, excludeStaticFieldMap,
excludedThreads);
    }
  }
}
```

参考源码的使用方法，如下
排除staticField干扰






com.squareup.leakcanary (17 usages)
AndroidExcludedRefs (17 usages)
(55: 16) excluded.instanceField("android.app.ActivityThread$ActivityClientRecord", "nextIdle");
(74: 16) excluded.instanceField("android.widget.Editor$EasyEditSpanController", "this$0");
(75: 16) excluded.instanceField("android.widget.Editor$SpanController", "this$0");
(132: 16) excluded.instanceField("android.os.Message", "obj");
(133: 16) excluded.instanceField("android.os.Message", "next");
(134: 16) excluded.instanceField("android.os.Message", "target");
(144: 16) excluded.instanceField("android.view.inputmethod.InputMethodManager", "mNextServedView");
(145: 16) excluded.instanceField("android.view.inputmethod.InputMethodManager", "mServedView");
(146: 16) excluded.instanceField("android.view.inputmethod.InputMethodManager",
(157: 16) excluded.instanceField("android.view.inputmethod.InputMethodManager", "mCurRootView");
(166: 16) excluded.instanceField("android.animation.LayoutTransition$1", "val$parent");
(175: 16) excluded.instanceField("android.view.textservice.SpellCheckerSession$1", "this$0");
(186: 18) excluded.instanceField("android.app.admin.DevicePolicyManager$SettingsObserver", "this$0");
(229: 16) excluded.instanceField("com.nvidia.ControllerMapper.MapperClient$ServiceClient", "this$0");
(245: 16) excluded.instanceField("android.os.PersonaManager", "mContext");
(255: 16) excluded.instanceField("android.content.res.Resources", "mContext");
(265: 16) excluded.instanceField("android.view.ViewConfiguration", "mContext");