

彻底搞懂Java内存泄露



编程之乐 (/u/79a88a044955) (+ 关注)

2017.10.21 12:14* 字数 2313 阅读 2574 评论 4 喜欢 31

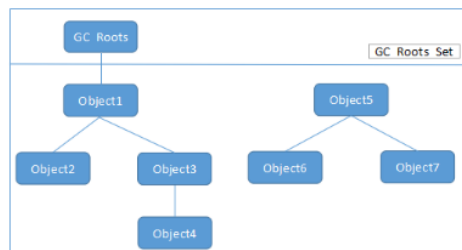
(/u/79a88a044955)

本文已授权微信公众号：鸿洋（hongyangAndroid）原创首发。
简书 编程之乐 (<https://www.jianshu.com/u/79a88a044955>)
转载请注明原创出处，谢谢！

Java内存回收方式

Java判断对象是否可以回收使用的而是可达性分析算法。

在主流的商用程序语言中(Java和C#)，都是使用可达性分析算法判断对象是否存活的。这个算法的基本思路就是通过一系列名为"GC Roots"的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的，下图对象object5, object6, object7虽然没有互相判断，但它们到GC Roots是不可达的，所以它们将会判定为是可回收对象。



Paste_Image.png

在Java语言里，可作为GC Roots对象的包括如下几种：

- a. 虚拟机栈(栈帧中的本地变量表)中的引用的对象
- b. 方法区中的类静态属性引用的对象
- c. 方法区中的常量引用的对象
- d. 本地方法栈中JNI的引用的对象

摘自《深入理解Java虚拟机》

使用leakcanary检测泄漏

关于LeakCanary使用参考以下文章：

1. LeakCanary: 让内存泄露无所遁形 (<https://link.jianshu.com?t=http://www.liaohuqiu.net/cn/posts/leak-canary/>)

2. LeakCanary 中文使用说明 (<https://link.jianshu.com?t=http://www.liaohuqiu.net/cn/posts/leak-canary-read-me/>)

LeakCanary的内存泄露提示一般会包含三个部分:

第一部分(LeakSingle类的sInstance变量)

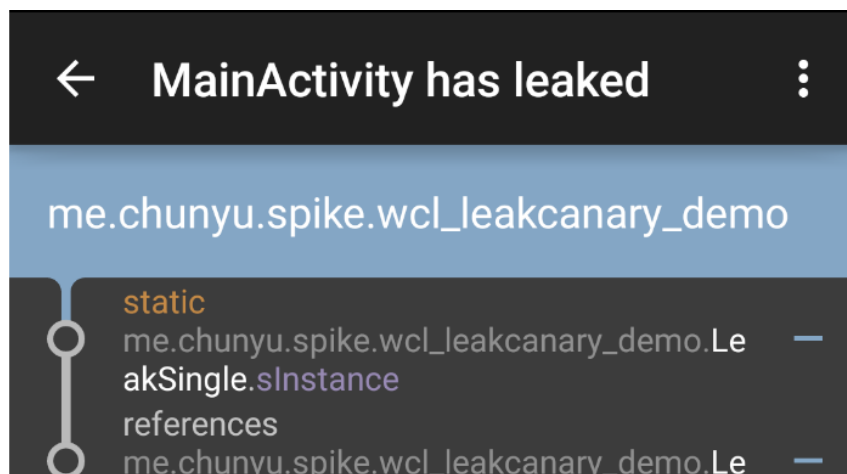
引用第二部分(LeakSingle类的mContext变量),

导致第三部分(MainActivity类的实例instance)泄露.

+

🔖

🔗



Paste_Image.png

leakcanary使用注意

即使是空的Activity，如果检测泄露时候遇到了如下这样的泄露，注意，把refWatcher.watch()放在onDestroy里面即可解决，或者忽略这样的提示。由于文章已写很多，下面的就不再修改，忽略这种错误即可。

```
* com.less.demo.TestActivity has leaked:
* GC ROOT static android.app.ActivityThread.sCurrentActivityThread
* references android.app.ActivityThread.mActivities
* references android.util.ArrayMap.mArray
* references array java.lang.Object[][1]
* references android.app.ActivityThread$ActivityClientRecord.activity
* leaks com.less.demo.TestActivity instance
```

```
protected void onDestroy() {
    super.onDestroy();
    RefWatcher refWatcher = App.getRefWatcher(this);
    refWatcher.watch(this);
}
```

leakcanary和代码示例说明内存泄露

案例一(静态成员引起的内存泄露)

```
public class App extends Application {
    private RefWatcher refWatcher;

    @Override
    public void onCreate() {
        super.onCreate();
        refWatcher = LeakCanary.install(this);
    }

    public static RefWatcher getRefWatcher(Context context) {
        App application = (App) context.getApplicationContext();
        return application.refWatcher;
    }
}
```

测试内部类持有外部类引用，内部类是静态的(GC-ROOT,将一直连着这个外部类实例)，静态的会和Application一个生命周期，这会导致一直持有外部类引用(内部类隐含了一个成员变量\$0)，即使外部类制空= null，也无法释放。

OutterClass



```

public class OutterClass {
    private String name;

    class Inner{
        public void list(){
            System.out.println("outter name is " + name);
        }
    }
}

```

TestActivity

```

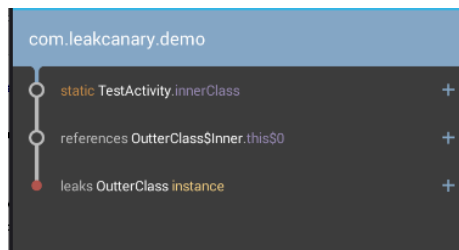
public class TestActivity extends Activity {
    // 静态的内部类实例
    private static OutterClass.Inner innerClass;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_test);
        OutterClass outterClass = new OutterClass();
        innerClass = outterClass.new Inner();

        RefWatcher refWatcher = App.getRefWatcher(this);
        refWatcher.watch(outterClass);// 监控的对象

        outterClass = null;
    }
}

```



Paste_Image.png

案例二(单例模式引起的内存泄露)

DownloadManager

```

public class DownloadManager {
    private static DownloadManager instance;
    private Task task ;

    public static DownloadManager getInstance(){
        if (instance == null) {
            instance = new DownloadManager();
        }
        return instance;
    }
    public Task newTask(){
        this.task = new Task();
        return task;
    }
}

```

Task

```

public class Task {
    private Call call;
    public Call newCall(){
        this.call = new Call();
        return call;
    }
}

```

Call



```

public class Call {
    public void execute(){
        System.out.println("=====> execute call");
    }
}

```

TestActivity

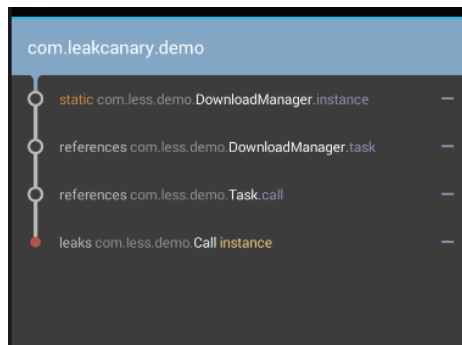
```

public class TestActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_test);
        RefWatcher refWatcher = App.getRefWatcher(this);

        Task task = DownloadManager.getInstance().newTask();
        Call call = task.newCall();
        call.execute();
        refWatcher.watch(call); // 监控的对象
        call = null; // 无法回收, DownloadManager是静态单例, 引用task, task引用了call, 即使call
    }
}

```



Paste_Image.png

部分日志打印如下:当前的GC_ROOT是DownloadManager的instance实例。

```

In com.leakcanary.demo:1.0:1.
* com.les.demo.Call has leaked:
* GC ROOT static com.les.demo.DownloadManager.instance
* references com.les.demo.DownloadManager.task
* references com.les.demo.Task.call
* leaks com.les.demo.Call instance

```

关于上面两种方式导致的内存泄露问题，这里再举两个案例说明以加强理解。

案例三(静态变量导致的内存泄露)

```

public class TestActivity extends Activity {
    private static Context sContext;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_test);
        sContext = this;

        RefWatcher refWatcher = App.getRefWatcher(this);

        refWatcher.watch(this); // 监控的对象
    }
}

```



Paste_Image.png

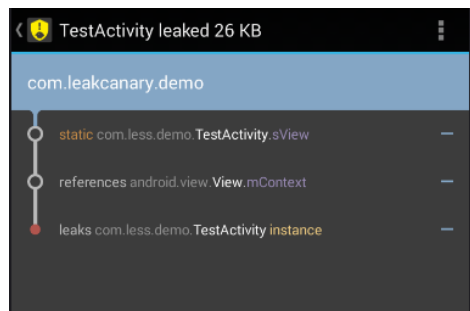
打印日志如下：

```
In com.leakcanary.demo:1.0:1.  
com.less.demo.TestActivity has leaked:  
GC ROOT static com.less.demo.TestActivity.sContext  
leaks com.less.demo.TestActivity instance
```

从这段日志可以分析出：声明static后，sContext的生命周期将和Application一样长，Activity即使退出到桌面，Application依然存在->sContext依然存在，GC此时想回收Activity却发现Activity仍然被sContext(GC-ROOT连接着)，导致死活回收不了，内存泄露。

上面的代码改造一下，如下。

```
public class TestActivity extends Activity {  
    private static View sView;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_test);  
        sView = new View(this);  
  
        RefWatcher refWatcher = App.getRefWatcher(this);  
  
        refWatcher.watch(this);  
    }  
}
```



Paste_Image.png

日志如下

```
In com.leakcanary.demo:1.0:1.  
com.less.demo.TestActivity has leaked:  
GC ROOT static com.less.demo.TestActivity.sView  
references android.view.View.mContext  
leaks com.less.demo.TestActivity instance
```

案例四(单例模式导致的内存泄露)

DownloadManager



```

public class DownloadManager {
    private static DownloadManager instance;
    private List<DownloadListener> mListeners = new ArrayList<>();

    public interface DownloadListener {
        void done();
    }

    public static DownloadManager getInstance(){
        if (instance == null) {
            instance = new DownloadManager();
        }
        return instance;
    }

    public void register(DownloadListener downloadListener){
        if (!mListeners.contains(downloadListener)) {
            mListeners.add(downloadListener);
        }
    }

    public void unregister(DownloadListener downloadListener){
        if (mListeners.contains(downloadListener)) {
            mListeners.remove(downloadListener);
        }
    }
}

```

TestActivity

```

public class TestActivity extends Activity implements DownloadManager.DownloadListener{

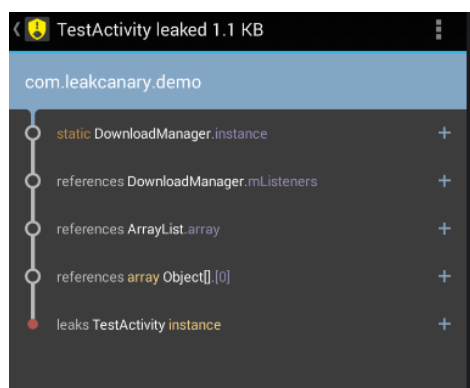
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_test);
        DownloadManager.getInstance().register(this);

        RefWatcher refWatcher = App.getRefWatcher(this);
        refWatcher.watch(this);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        // 忘记 unregister
        // DownloadManager.getInstance().unregister(this);
    }

    @Override
    public void done() {
        System.out.println("done!");
    }
}

```



Paste_Image.png

```

In com.leakcanary.demo:1.0:1.
* com.less.demo.TestActivity has leaked:
* GC ROOT static com.less.demo.DownloadManager.instance
* references com.less.demo.DownloadManager.mListeners
* references java.util.ArrayList.array
* references array java.lang.Object[][0]
* leaks com.less.demo.TestActivity instance

```

错误写法一定导致内存泄露吗？



答案是：否定的。

如下案例，有限时间内是可以挽救内存泄露发生的，所以控制好生命周期，其他情况：
如单例对象(静态变量)的生命周期比其持有的sContext
的生命周期更长时 ->内存泄露，更短时->躲过内存泄露。

```
public class TestActivity extends Activity {
    private static Context sContext;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_test);
        sContext = this;

        new Handler().postDelayed(new Runnable() {
            @Override
            public void run() {
                sContext = null;
            }
        }, 1000); // 分别测试1s和12s,前者不会内存泄露，后者一定泄露。所以如果赶在GC之前切断GC_ROOT是

    }

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

Handler 引起的内存泄露详细分析

handler导致的内存泄露可能我们大多数都犯过。

注意以下代码中的注释,非静态内部类虽然持有外部类引用，但是持有并不代表一定泄露，而是看gc时谁的命长。经过测试 **情况(1)**始终没有内存泄露。

为什么会这样，很早阅读Handler源码时候记得这几个货都是互相引用来引用去的，Message有个target字段，message.target = handler; handler.post(message);又把这个message推入了MessageQueue中，而MessageQueue是在一个Looper线程中不断轮询处理消息，而有时候message还是个老不死，能够重复利用。如果当Activity退出时候，还有消息未处理或正在处理，由于message引用 handler,handler又引用Activity，此时将引发内存泄露。

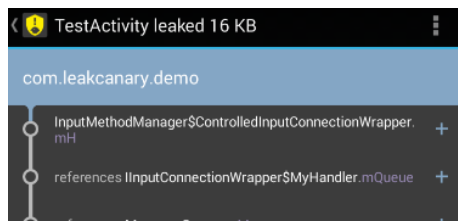
```
public class TestActivity extends Activity {
    private Handler handler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);
            System.out.println("==== handle message ===");
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_test);
        // (1) 不会导致内存泄露
        handler.sendEmptyMessageDelayed(0x123, 0);

        // (2) 会导致内存泄露，非静态内部类（包括匿名内部类，比如这个 Handler 匿名内部类）会引用外
        // 当它使用了 postDelayed 的时候，如果 Activity 已经 finish 了，而这个 handler 仍然引用
        // 因为这个 handler 会在一段时间内继续被 main Looper 持有，导致引用仍然存在。
        handler.sendEmptyMessageDelayed(0x123, 12000);
    }

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





Paste_Image.png

```
com.less.demo.TestActivity has leaked:
* GC ROOT android.view.inputmethod.InputMethodManager$ControlledInputConnectionWrapper.mH
* references com.android.internal.view.IInputConnectionWrapper$MyHandler.mQueue
* references android.os.MessageQueue.mMessages
* references android.os.Message.target, matching exclusion field android.os.Message#target
* references com.less.demo.TestActivity$1.this$0 (anonymous subclass of android.os.Handler)
* leaks com.less.demo.TestActivity instance
```

知道了原理后，即使写出易于内存泄露的代码也能够避免内存泄露啦。

上述代码只需在onDestroy()函数中调用mHandler.removeCallbacksAndMessages(null);在Activity退出的时候的移除消息队列中所有消息和所有的Runnable。

内部类引起的内存泄露

内部类种类大致如下：

1. 非静态内部类(成员内部类)
2. 静态内部类(嵌套内部类)
3. 局部内部类(定义在方法内或者作用域内的类,好似局部变量,所以不能有访问控制符和static等修饰)
4. 匿名内部类(没有名字,仅使用一次new个对象即扔掉类的定义)

为什么非静态内部类持有外部类引用,静态内部类不持有外部引用。

这个问题非常简单，就像 static的方法只能调用static的东西，非static可以调用非static和static的一样。static--> 针对class, 非static->针对 对象，我是这么简单理解的。看图：

```
public class TestActivity extends Activity {
    private String outerName;

    class Inner {
        public void invoke(){
            System.out.println("外面的那个人是: " + outerName);
            System.out.println("外面的那个人是: " + TestActivity.this.outerName);
        }
    }

    static class staticInner {
        public void invoke(){
            System.out.println("外面的那个人是啥玩意: " + outerName);
            System.out.println("外面的那个人是啥玩意: " + TestActivity.this.outerName);
        }
    }
}
```

Paste_Image.png

匿名内部类

将局部内部类的使用再深入一步，假如只创建某个局部内部类的一个对象，就不必命名了。

匿名内部类的类型可以是如下几种方式。

1. 接口匿名内部类
2. 抽象类匿名内部类
3. 类匿名内部类



匿名内部类总结:

1. 其实主要就是类定义一次就失效了，主要使用的是这个类(不知道名字)的实例。根据内部类的特性，能够实现回调和闭包。
2. JavaScript和Python的回调传递的是function,Java传递的是object。
Java中常常用到回调，而回调的本质就是传递一个**对象**，JavaScript或其他语言则是传递一个**函数(如Python，或者C,使用函数指针的方式)**。由于传递一个对象可以携带其他的一些信息，所以Java认为传递一个对象比传递一个函数要灵活的多(当然java也可以用Method反射对象传递函数)。参考《Java核心技术》

非静态内部类导致内存泄露(前提dog的命长)

下面的案例将导致内存泄露

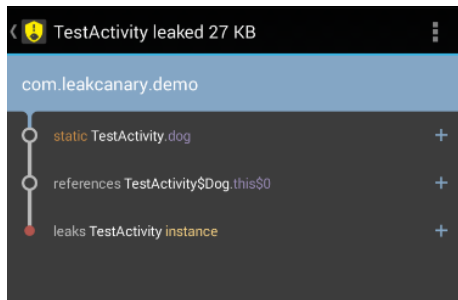
其中 `private static Dog dog ;` 导致Dog的命比TestActivity长，这就糟糕了，但是注意，如果改为 `private Dog dog ;` 即使Dog是非静态内部类，也不会导致内存泄露，要死也是Dog先死,毕竟Dog是TestActivity的家庭成员，开挂也得看主人。

```
public class TestActivity extends Activity {
    private static Dog dog ;

    class Dog {
        public void say(){
            System.out.println("I am lovely dog!");
        }
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_test);
        dog = new Dog();
    }

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



Paste_Image.png

```
In com.leakcanary.demo:1.0:1.
* com.less.demo.TestActivity has leaked:
* GC ROOT static com.less.demo.TestActivity.dog
* references com.less.demo.TestActivity$Dog.this$0
* leaks com.less.demo.TestActivity instance
```

哪些内部类或者回调函数是否持有外部类对象？

一个反射案例说明一切



```

/**
 * 作者: limitless
 * 描述: 一个案例测试所有类型内部类对外部类对象的持有情况
 * 网站: https://github.com/wangli0
 */
public class Main {

    /* 持有外部类引用 */
    private IApplister mApplister = new IApplister() {
        private String name;
        @Override
        public void done() {
            System.out.println("匿名内部类对象作为成员变量");
        }
    };
    /* 未持有 */
    private static IApplister sApplister = new IApplister() {
        private String name;
        @Override
        public void done() {
            System.out.println("匿名内部类对象作为static成员变量");
        }
    };

    public static void main(String args[]) {
        Main main = new Main();
        main.test1();
        main.test2();
        main.test3();// test3 《=》 test4
        main.test4();
        main.test5();
        main.test6();
    }

    class Dog {
        private String name;
    }

    /* 持有外部类引用 */
    public void test1(){
        Dog dog = new Dog();
        getAllFieldName(dog.getClass());
    }

    static class Cat {
        private String name;
    }
    /* 未持有 */
    private void test2() {
        Cat cat = new Cat();
        getAllFieldName(cat.getClass());
    }

    /* 持有外部类引用 */
    private void test3() {
        class Monkey{
            String name;
        }
        Monkey monkey = new Monkey();
        getAllFieldName(monkey.getClass());
    }

    /* 持有外部类引用 */
    private void test4() {
        // 常用作事件回调的地方(有可能引起内存泄露)
        IApplister iApplister = new IApplister() {
            private String name;
            @Override
            public void done() {
                System.out.println("匿名内部类");
            }
        };
        getAllFieldName(iApplister.getClass());
    }

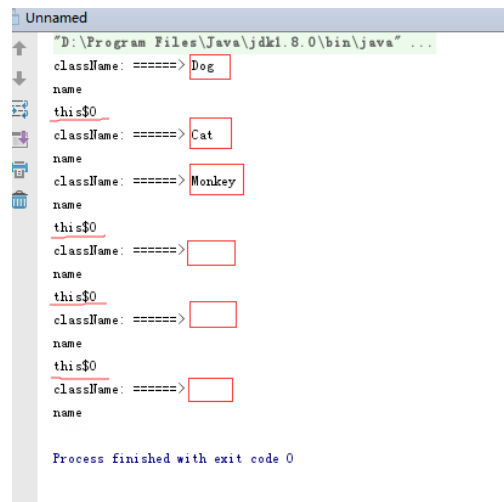
    /* 持有外部类引用 */
    private void test5() {
        getAllFieldName(mApplister.getClass());
    }

    /* 未持有 */
    private void test6() {
        getAllFieldName(sApplister.getClass());
    }

    private void getAllFieldName(Class<> clazz) {
        System.out.println("className: =====> " + clazz.getSimpleName());
        Field[] fields = clazz.getDeclaredFields();
        for (Field field : fields) {
            System.out.println(field.getName());
        }
    }
}

```





Paste_Image.png

上述结果足够说明，即使是方法内的回调对象也是持有外部类引用的，那么虽然作用域是局部的，也有存在内存泄露的可能。我多次强调 **持有某对象** 不代表一定泄露，看的是谁命长。回调在Android开发过程中几乎处处可见，如果**持有**就会内存泄露的话，那几乎就没法玩了。

一般情况下，我们常常设置某个方法内的 `xx.execute(new Listener(){xx});` 是会导致内存泄露的，这个方法执行完，局部作用域就失效了。但是如果在`execute(listener);`过程中，某个单例模式的对象 突然引用了这个listener对象，那么就会导致内存泄露。

下面用实例证明我的想法

TestActivity

```
public class TestActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_test);

        Task task = new Task();
        task.execute(new ICallback() {
            @Override
            public void done() {
                System.out.println("下载完成! ");
            }
        });
    }

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

Task

```
public class Task {
    public void execute(ICallback iCallback) {
        DownloadManager.getInstance().execute(iCallback);
    }
}
```

DownloadManager

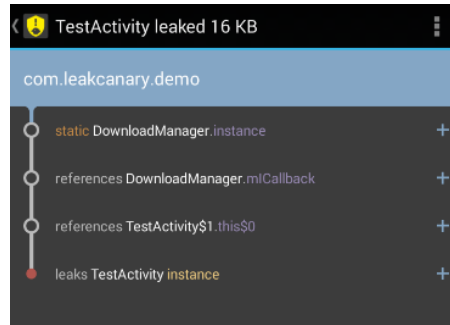


```

public class DownloadManager {
    public static DownloadManager instance;
    private ICallback mICallback;
    public static DownloadManager getInstance(){
        if (instance == null) {
            instance = new DownloadManager();
        }
        return instance;
    }

    public void execute(ICallback iCallback) {
        this.mICallback = iCallback;// 反例，千万不要这么做，将导致内存泄露,如果注释掉这行，内存
        iCallback.done();
    }
}

```



Paste_Image.png

这足以证明我的想法是正确的，Callback的不巧当使用同样会导致**内存泄露** 的发送。

总结

1. 如果某些单例需要使用到Context对象，推荐使用Application的context，不要使用Activity的context，否则容易导致内存泄露。单例对象的生命周期和Application一致，这样Application和单例对象就一起销毁。
2. 优先使用静态内部类而不是非静态的,因为非静态内部类持有外部类引用可能导致垃圾回收失败。如果你的静态内部类需要宿主Activity的引用来执行某些东西，你要将这个引用封装在一个WeakReference中，避免意外导致Activity泄露,被弱引用关联的对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收 **只被弱引用关联** 的对象，**只被** 说明这个对象本身已经没有用处了。

```

public class TestActivity extends Activity {
    private MyHandler myHandler = new MyHandler(this);
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_test);
    }

    static class MyHandler extends Handler {
        private WeakReference<Activity> mWeakReference;

        public MyHandler(Activity activity){
            mWeakReference = new WeakReference<Activity>(activity);
        }

        @Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);
            Toast.makeText(mWeakReference.get(), "xxxx", Toast.LENGTH_LONG).show();
            Log.d("xx", mWeakReference.get().getPackageName());
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        RefWatcher refWatcher = App.getRefWatcher(this);
        refWatcher.watch(this);
    }
}

```



编程之乐 (/u/79a88a044955)

写了 38284 字, 被 136 人关注, 获得了 228 个喜欢
(/u/79a88a044955)

+ 关注

博客 : <https://wangli0.github.io/> Github : <https://github.com/wangli0>

小礼物走一走, 来简书关注我


赞赏支持

喜欢 | 31



更多分享

(<http://cwb.assets.jianshu.io/notes/images/18516177/weib>)




写下你的评论...

4条评论


只看作者

按喜欢排序 按时间正序 按时间倒序




木米小雨 (/u/3d7939962bc2)
2楼 · 2017.10.21 12:24
(/u/3d7939962bc2)
666666666666

 赞  回复



蜗牛爱爬行 (/u/bae81d4060ea)
3楼 · 2017.10.21 15:13
(/u/bae81d4060ea)
写了太棒了, 帮助很大, 非常感谢。

 赞  回复



沃的爱情 (/u/43aeeab4d38a)
4楼 · 2017.10.21 15:40
(/u/43aeeab4d38a)
目前遇到过写的最好的一篇关于内存泄露的文章了, 案例和截图一下就看明白了, 没图还以为你是在胡扯呢。

 赞  回复



开发者头条_程序员必装的App (/u/6a1613a5b777)
5楼 · 2017.11.14 10:35
(/u/6a1613a5b777)
感谢分享! 已推荐到《开发者头条》: <https://toutiao.io/posts/mzq6mq>
(<https://toutiao.io/posts/mzq6mq>) 欢迎点赞支持!
欢迎订阅《编程之乐的独家号》<https://toutiao.io/subjects/291116>
(<https://toutiao.io/subjects/291116>)

 赞  回复

被以下专题收入, 发现更多相似内容

- + 收入我的专题
- 

Android开发 (/c/d1591c322c89?utm_source=desktop&utm_medium=notes-included-collection)



程序员 (/c/NEt52a?utm_source=desktop&utm_medium=notes-included-collection)

+
🔖
🔗



程序员首页投稿 (/c/89995286335f?

utm_source=desktop&utm_medium=notes-included-collection)



首页投稿 (/c/bDHhpK?utm_source=desktop&utm_medium=notes-included-collection)



Android知识 (/c/3fde3b545a35?utm_source=desktop&utm_medium=notes-included-collection)



Android... (/c/4688bad2bca2?utm_source=desktop&utm_medium=notes-included-collection)



Android... (/c/3b3218c1e593?utm_source=desktop&utm_medium=notes-included-collection)

推荐阅读

更多精彩内容 > (/)

Idea 中使用PlantUML插件生成UML (/p/12d87aab6fd...

(/p/12d87aab6fd0?

在使用过程中可能会遇到的错误 这是因为没有安装Graphviz下载安装即可，我这里windows版本<http://download.csdn.net/download/aacode/7484453> 安...

utm_campaign=maleskine&utm_content=note&utm_medi

编程之乐 (/u/79a88a044955?

utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)

动态加载so注意事项&案例 (/p/a06e6f0f402a?utm_ca...

(/p/a06e6f0f402a?

常用架构armeabi, armeabi-v7a, x86, mips, arm64-v8a, mips64, x86_64。加载so的两种方式 打包在apk中的情况，不需要开发者自己去判断...

utm_campaign=maleskine&utm_content=note&utm_medi

编程之乐 (/u/79a88a044955?

utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)

结婚后，我才懂父母“门当户对”的要求没毛病 (/p/34af...

(/p/34af2ac274b3?

结婚那年，我才二十岁。在那个爱情至上的年龄，我始终坚信爱情远比面包重要。面包没了，我们可以赚，爱情没了，就真的没了。所以我跟温先生谈恋...

utm_campaign=maleskine&utm_content=note&utm_medi

亲亲爱英 (/u/97308d7c96da?

utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)

好吃到哭！淘宝上双十一最值得购买的美食有哪些？ (/...

(/p/3884a6b7a780?

大家好，我是小丸子~ 终于，双十一马上就要到了，整理了这么多天，我特意把最重要，也是关注人数最多的淘宝美食留在了最后，希望能够在这一期间整理更多...

utm_campaign=maleskine&utm_content=note&utm_medi

小丸子的杂物集 (/u/24bca2bb387d?

utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)

赵薇：一念成佛，一念成魔，还原你不知道的“小燕子”...

(/p/c496294d338e?

文/麦大人 01 天下熙熙，皆为利来；天下攘攘，皆为利往。话说身在名利场的人，皆是匆匆过客。你方唱罢我登场，绚丽的舞台永远不会为谁而停留。 娱...

utm_campaign=maleskine&utm_content=note&utm_medi

麦大人 (/u/2b3ad4f2a058?

utm_campaign=maleskine&utm_content=user&utm_medium=pc_all_hots&utm_source=recommendation)

