面试官：Handler内存泄露的原因是什么？我：就这？太简单了吧，但我却被挂了...

2021-03-12阅读 1.1K0

**前言**

嗨，大家好，问大家一个“**简单**”的问题：

Handler内存泄露的原因是什么？

你会怎么答呢？

**这是错误的回答**

有的朋友看到这个题表示，就这？太简单了吧。

"内部类持有了外部类的引用，也就是Hanlder持有了Activity的引用，从而导致无法被回收呗。"

其实这样回答是**错误**的，或者说没回答到**点子**上。

**内存泄漏**

Java虚拟机中使用**可达性分析**的算法来决定对象是否可以被回收。即通过GCRoot对象为起始点，向下搜索走过的路径（引用链），如果发现某个对象或者对象组为不可达状态，则将其进行回收。

而内存泄漏指的就是有些对象（短周期对象）没有用了，但是却被其他有用的类（长周期对象）所引用，从而导致无用对象占据了内存空间，形成内存泄漏。

所以上面的问题，如果仅仅回答内部类持有了外部类的引用，没有指出内部类被谁所引用，那么按道理来说是不会发生内存泄漏的，因为内部类和外部类都是无用对象了，是可以被正常回收的。

所以这一题的关键在于，内部类被谁引用了？也就是Handler被谁引用了？

一起通过实践研究下吧～

**Handler发生内存泄漏的情况**

**1、发送延迟消息**

第一种情况，是通过handler发送延迟消息：

class MainActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_handler)

btn.setOnClickListener {

//跳转到HandlerActivity

startActivity(Intent(this, HandlerActivity::class.java))

}

}

}

class HandlerActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_handler2)

//发送延迟消息

mHandler.sendEmptyMessageDelayed(0, 20000)

btn2.setOnClickListener {

finish()

}

}

val mHandler = object : Handler() {

override fun handleMessage(msg: Message?) {

super.handleMessage(msg)

btn2.setText("2222")

}

}

}

我们在HandlerActivity中，发送一个延迟20s的消息。然后打开HandlerActivity后，马上finish。看看会不会内存泄漏。

**查看内存泄漏并分析**

现在查看内存泄漏还是蛮方便的了，AndroidStudio自带对堆转储（Heap Dump）文件进行分析，并且会把内存泄漏点明确标出来。

我们运行项目，点击Profiler——Memory，就能看到以下图片了，一个正在运行的内存情况实时图：

可以看到图片中有两个按钮我标出来了：

* 捕获堆转储文件按钮，也就是生成hprof文件，这个文件会展示Java堆的使用情况，点击这个按钮后，AndroidStudio会帮我们生成这个堆转储文件并且进行分析。
* GC按钮，一般我们在我们捕获堆转储文件之前，点一下GC，就能把一些弱引用给回收，防止给我们分析带来干扰。

所以我们打开HandlerActivity后，马上finish，然后点击GC按钮，再点击捕获堆转储文件按钮。AndroidStudio会自动跳转到以下界面：

可以看到左上角有一个Leaks，这就是你内存泄漏的点，点击就能看到内存泄漏的类了。右下角就是内存泄漏类的引用路径。

从这张图可以看到，我们的HandlerActivity发生了内存泄漏，从引用路径来看，是被匿名内部类的实例mHandler持有引用了，而Handler的引用是被Message持有了，Message引用是被MessageQueue持有了...

结合我们所学的Handler知识和这次引用路径分析，这次内存泄漏完整的引用链应该是：

**主线程 —> threadlocal —> Looper —> MessageQueue —> Message —> Handler —> Activity**

所以这次引用的头头就是主线程，主线程肯定是不会被回收的，只要是运行中的线程都不会被JVM回收，跟静态变量一样被JVM特殊照顾。

这次内存泄漏的原因算是搞清楚了，当然Handler内存泄漏的情况不光这一种，看看第二种情况：

**2、子线程运行没结束**

第二个实例，是我们常用到的，在子线程中工作，比如请求网络，然后请求成功后通过Handler进行UI更新。

class HandlerActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_handler2)

//运行中的子线程

thread {

Thread.sleep(20000)

mHandler.sendEmptyMessage(0)

}

btn2.setOnClickListener {

finish()

}

}

val mHandler = object : Handler() {

override fun handleMessage(msg: Message?) {

super.handleMessage(msg)

btn2.setText("2222")

}

}

}

同样运行后看看内存泄漏情况：

可以发现，这里的内存泄漏主要的原因是因为这个运行中的子线程，由于子线程这个匿名内部类持有了外部类的引用，而子线程本身是一直在运行的，刚才说过运行中的线程是不会被回收的，所以这里内存泄漏的引用链应该是：

**运行中的子线程 —> Activity**

当然，这里的Handler也是持有了Activity的引用的，但主要引起内存泄漏的原因还是在于子线程本身，就算子线程中不用Handler，而是调用Activity的其他变量或者方法还是会发生内存泄漏。

所以这种情况我觉得不能看作Handler引起内存泄漏的情况，其根本原因是因为子线程引起的，如果解决了子线程的内存泄漏，比如在Activity销毁的时候停止子线程，那么Activity就能正常被回收，那么也不存在Handler的问题了。

**延伸问题1：内部类为什么会持有外部类的引用**

这是因为内部类虽然和外部类写在同一个文件中，但是编译后还是会生成不同的class文件，其中内部类的构造函数中会传入外部类的实例，然后就可以通过this$0访问外部类的成员。

其实也挺好理解的吧，因为在内部类中可以调用外部类的方法，变量等等，所以肯定会持有外部类的引用的。

贴一段内部类在编译后用JD-GUI查看的class代码，也许你能更好的理解：

//原代码

class InnerClassOutClass{

class InnerUser {

private int age = 20;

}

}

//class代码

class InnerClassOutClass$InnerUser {

private int age;

InnerClassOutClass$InnerUser(InnerClassOutClass var1) {

this.this$0 = var1;

this.age = 20;

}

}

**延伸问题2：kotlin中的内部类与Java有什么不一样吗**

其实可以看到，在上述的代码中，我都加了一句

btn2.setText("2222")

这是因为在kotlin中的匿名内部类分为两种情况：

* 在Kotlin中，匿名内部类如果没有使用到外部类的对象引用时候，是不会持有外部类的对象引用的，此时的匿名内部类其实就是个静态匿名内部类，也就不会发生内存泄漏。
* 在Kotlin中，匿名内部类如果使用了对外部类的引用，像我刚才使用了btn2，这时候就会持有外部类的引用了，就会需要考虑内存泄漏的问题。

所以我特意加了这一句，让匿名内部类持有外部类的引用，复现内存泄漏问题。

同样kotlin中对于内部类也是和Java有区别的：

* Kotlin中所有的内部类都是默认静态的，也就都是静态内部类。
* 如果需要调用外部的对象方法，就需要用inner修饰，改成和Java一样的内部类，并且会持有外部类的引用，需要考虑内存泄漏问题。

**解决内存泄漏**

说了这么多，那么该怎么解决内存泄漏问题呢？其实所有内存泄漏的解决办法都大同小异，主要有以下几种：

* 不要让长生命周期对象持有短生命周期对象的引用，而是用长生命周期对象持有长生命周期对象的引用。

比如Glide使用的时候传的上下文不要用Activity而改用Application的上下文（这句有问题，并无此说法，在此修正）。还有单例模式不要传入Activity上下文。

* 将对象的强引用改成弱引用

强引用就是对象被强引用后，无论如何都不会被回收。 弱引用就是在垃圾回收时，如果这个对象只被弱引用关联（没有任何强引用关联他），那么这个对象就会被回收。 软引用就是在系统将发生内存溢出的时候，回进行回收。 虚引用是对象完全不会对其生存时间构成影响，也无法通过虚引用来获取对象实例，用的比较少。

所以我们将对象改成弱引用，就能保证在垃圾回收时被正常回收，比如Handler中传入Activity的弱引用实例：

MyHandler(WeakReference(this)).sendEmptyMessageDelayed(0, 20000)

//kotlin中内部类默认为静态内部类

class MyHandler(var mActivity: WeakReference<HandlerActivity>):Handler(){

override fun handleMessage(msg: Message?) {

super.handleMessage(msg)

mActivity.get()?.changeBtn()

}

}

* 内部类写成静态类或者外部类

跟上面Hanlder情况一样，有时候内部类被不正当使用，容易发生内存泄漏，解决办法就是写成外部类或者静态内部类。

* 在短周期结束的时候将可能发生内存泄漏的地方移除

比如Handler延迟消息，资源没关闭，集合没清理等等引起的内存泄漏，只要在Activity关闭的时候进行消除即可：

@Override

protected void onDestroy() {

//移除handler所有消息

if(mHanlder != null){

mHandler.removeCallbacksAndMessages(null)

}

super.onDestroy();

}

**总结**

**Handler内存泄露的原因是什么？**

Handler导致内存泄漏一般发生在发送延迟消息的时候，当Activity关闭之后，延迟消息还没发出，那么主线程中的MessageQueue就会持有这个消息的引用，而这个消息是持有Handler的引用，而handler作为匿名内部类持有了Activity的引用，所以就有了以下的一条引用链。

主线程 —> threadlocal —> Looper —> MessageQueue —> Message —> Handler —> Activity

其根本原因是因为这条引用链的头头，也就是主线程，是不会被回收的，所以导致Activity无法被回收，出现内存泄漏，其中Handler只能算是导火索。

而我们平时用到的子线程通过Handler更新UI，其原因是因为运行中的子线程不会被回收，而子线程持有了Actiivty的引用（不然也无法调用Activity的Handler），所以就导致内存泄漏了，但是这个情况的主要原因还是在于子线程本身。

所以综合两种情况，在发生内存泄漏的情况中，Handler都不能算是罪魁祸首，罪魁祸首（根本原因）都是他们的头头——**线程**。

**面试前做好准备战！**

接下来将分享面试的一个复习路线，如果你也在准备面试但是不知道怎么高效复习，可以参考一下我的复习路线，有任何问题也欢迎一起互相交流，加油吧！

**这里给大家提供一个方向，进行体系化的学习：**

**1、看视频进行系统学习**

前几年的Crud经历，让我明白自己真的算是菜鸡中的战斗机，也正因为Crud，导致自己技术比较零散，也不够深入不够系统，所以重新进行学习是很有必要的。我差的是系统知识，差的结构框架和思路，所以通过视频来学习，效果更好，也更全面。关于视频学习，个人可以推荐去B站进行学习，B站上有很多学习视频，唯一的缺点就是免费的容易过时。

另外，我自己也珍藏了好几套视频，有需要的我也可以分享给你。

**2、进行系统梳理知识，提升储备**

客户端开发的知识点就那么多，面试问来问去还是那么点东西。所以面试没有其他的诀窍，只看你对这些知识点准备的充分程度。so，出去面试时先看看自己复习到了哪个阶段就好。

系统学习方向：

* **架构师筑基必备技能：**深入Java泛型+注解深入浅出+并发编程+数据传输与序列化+Java虚拟机原理+反射与类加载+动态代理+高效IO
* **Android高级UI与FrameWork源码：**高级UI晋升+Framework内核解析+Android组件内核+数据持久化
* **360°全方面性能调优：**设计思想与代码质量优化+程序性能优化+开发效率优化
* **解读开源框架设计思想：**热修复设计+插件化框架解读+组件化框架设计+图片加载框架+网络访问框架设计+RXJava响应式编程框架设计+IOC架构设计+Android架构组件Jetpack
* **NDK模块开发：**NDK基础知识体系+底层[图片处理](https://cloud.tencent.com/product/ip?from=10680)+音视频开发
* **微信小程序：**小程序介绍+UI开发+API操作+微信对接
* **Hybrid 开发与Flutter：**Html5项目实战+Flutter进阶

知识梳理完之后，就需要进行查漏补缺，所以针对这些知识点，我手头上也准备了不少的电子书和笔记，这些笔记将各个知识点进行了完美的总结。

**3、读源码，看实战笔记，学习大神思路**

“编程语言是程序员的表达的方式，而架构是程序员对世界的认知”。所以，程序员要想快速认知并学习架构，读源码是必不可少的。阅读源码，是解决问题 + 理解事物，更重要的：看到源码背后的想法；程序员说：读万行源码，行万种实践。

**主要内含微信 MMKV 源码、AsyncTask 源码、Volley 源码、Retrofit源码、OkHttp 源码等等。**

**4、面试前夕，刷题冲刺**

面试的前一周时间内，就可以开始刷题冲刺了。请记住，刷题的时候，技术的优先，算法的看些基本的，比如排序等即可，而智力题，除非是校招，否则一般不怎么会问。

关于面试刷题，我个人也准备了一套系统的面试题，帮助你举一反三。

**以上这些内容均免费分享给大家，需要完整版的朋友，**[**点这里可以看到全部内容**](https://github.com/Xiaxiao-1024/Android-P7-/blob/master/Android%E5%BC%80%E5%8F%91%E8%BF%98%E4%B8%8D%E4%BC%9A%E8%BF%99%E4%BA%9B%EF%BC%9F%E5%A6%82%E4%BD%95%E9%9D%A2%E8%AF%95%E6%8B%BF%E9%AB%98%E8%96%AA%EF%BC%81.md)**。**