安卓内存管理

Android 虚拟机跟踪堆中的每个内存分配。堆是系统分配 Java/Kotlin 对象的一块内存。

有一个回收未使用内存的过程，称为垃圾收集。它有以下目标：

寻找没有人需要的对象。

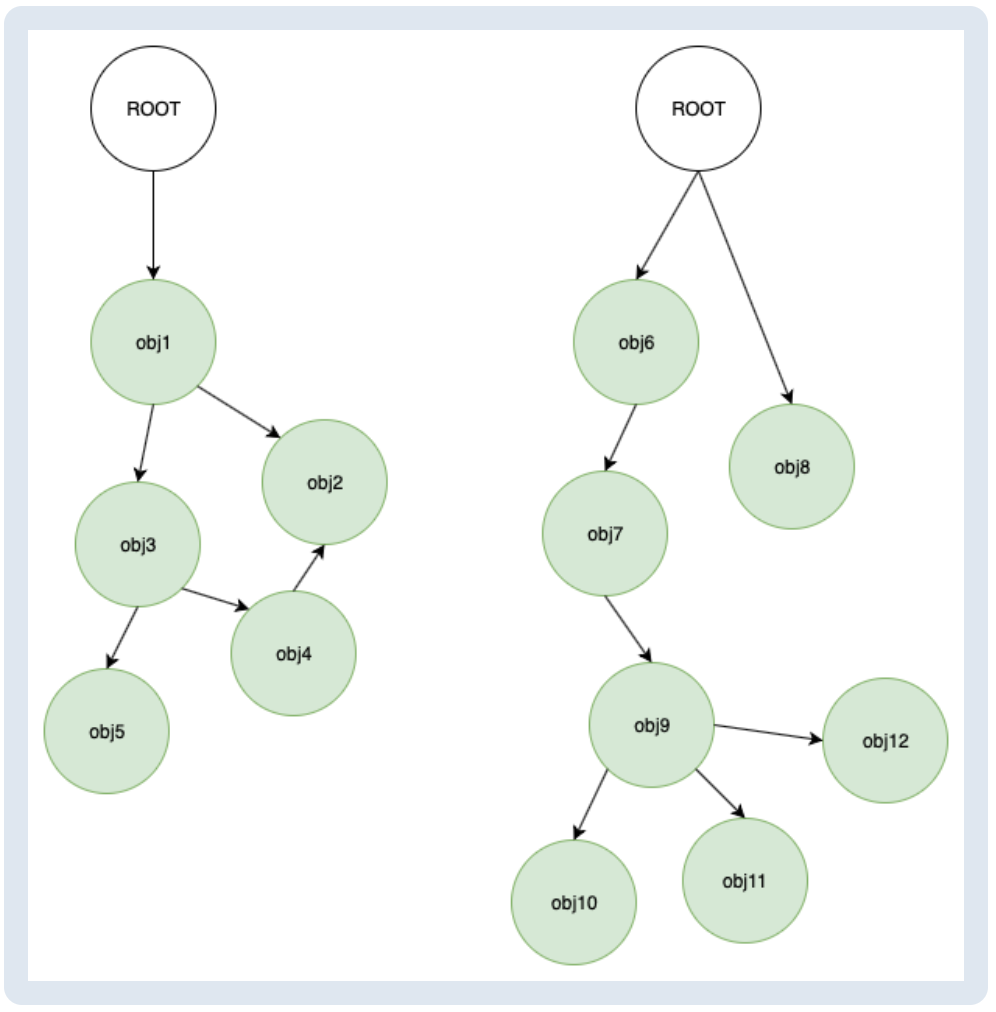
回收这些对象使用的内存并将其返回到堆中。

通常不会请求垃圾收集。相反，系统有一套运行标准来确定何时执行。

为了启用多任务环境，Android 对每个应用程序的堆大小进行了限制。此大小将根据设备的可用 RAM 大小而有所不同。当堆容量已满并且系统尝试分配更多内存时，可能会收到 OutOfMemoryError。

垃圾收集根

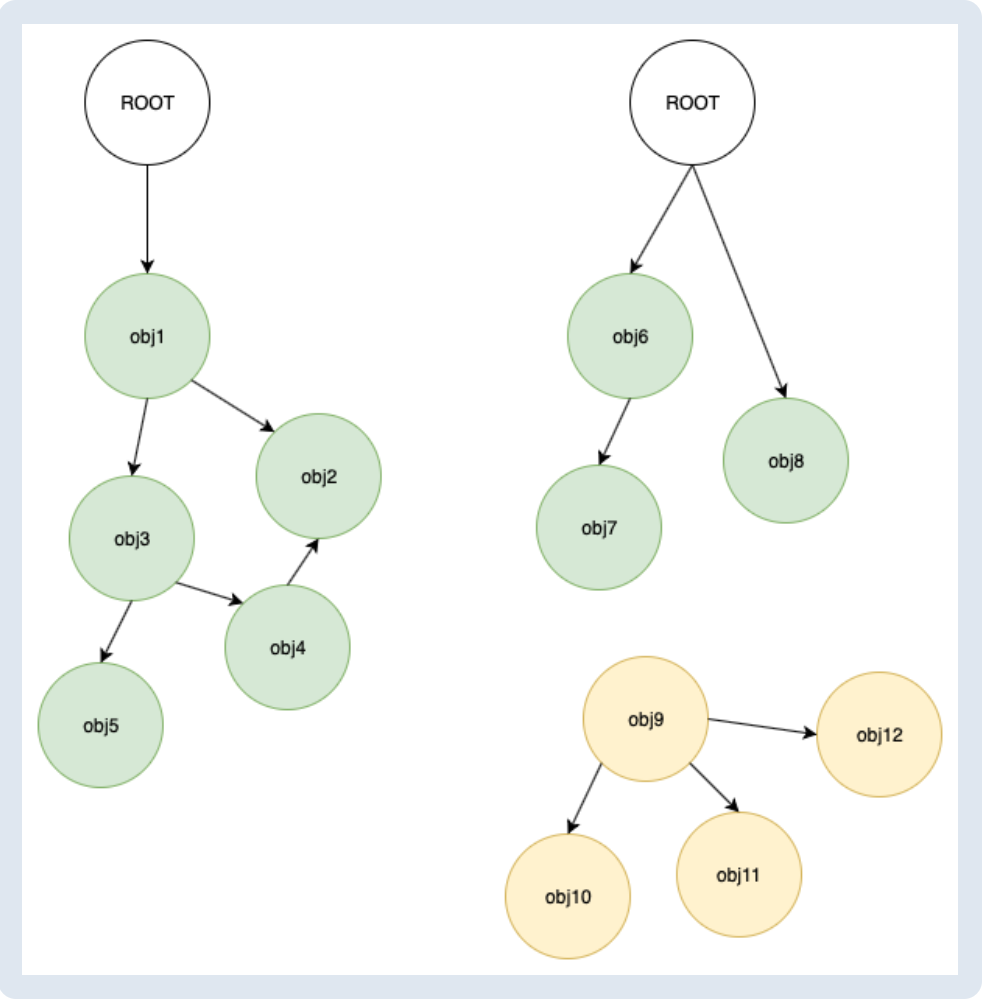
假设您在内存中有以下对象：



顶部的白色对象称为 GC Roots。堆中没有其他对象引用它们。

为简化起见，可以将 obj9 视为保留其他对象的Activity：obj10、obj11 和 obj12。

假设在那个Activity中，然后按返回键，销毁Activity。系统会清除 obj7 到 obj9 的引用：



当系统触发垃圾收集器时，它将从 GC 根开始。它将意识到 obj9 和它保留的其余对象 obj10、obj11 和 obj12 不可访问，并将收集它们。

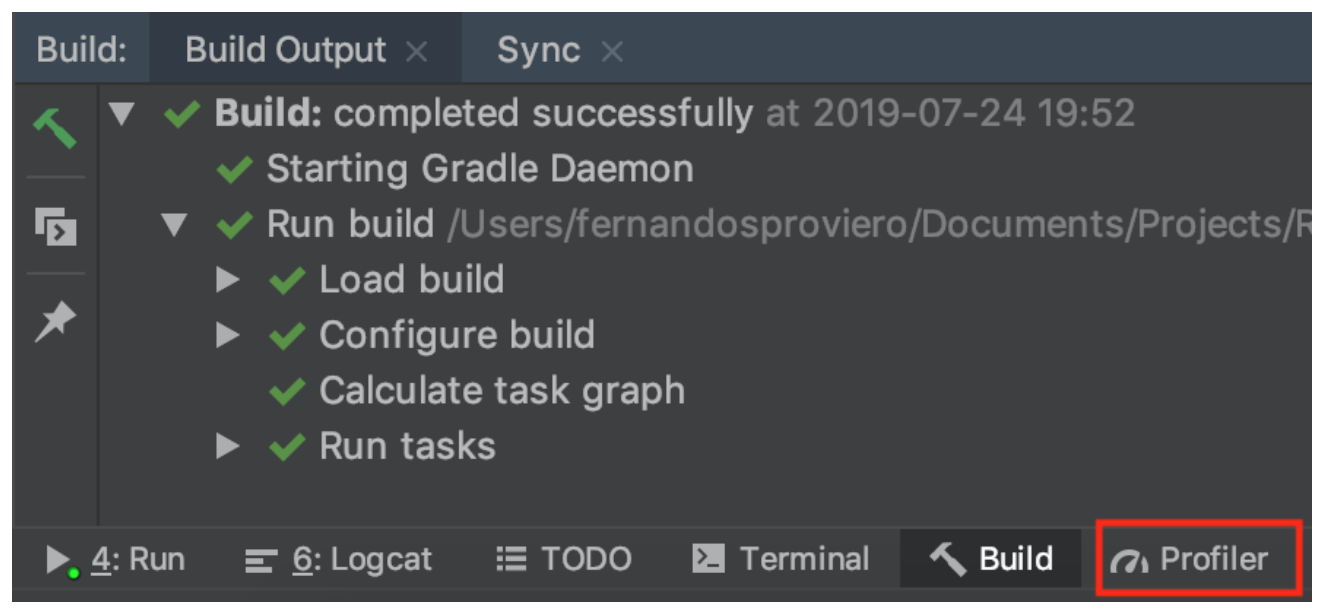
为什么你应该分析你的应用程序内存

系统必须暂停您的应用程序代码以让垃圾收集器完成其工作。通常，这个过程是不可察觉的。

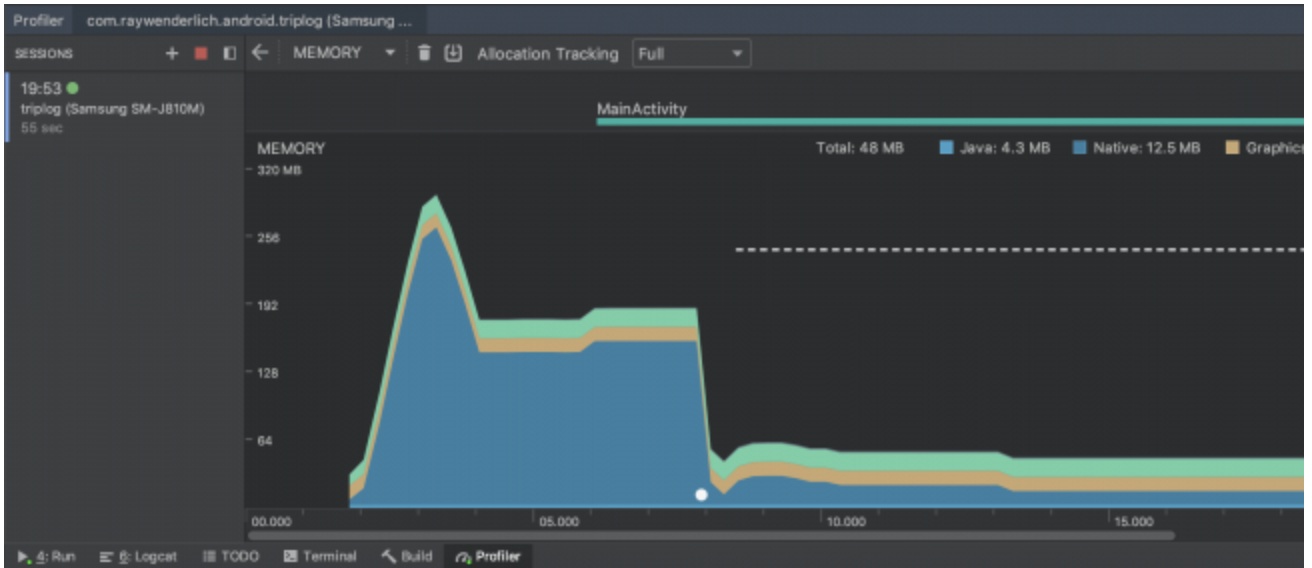
但有时您会注意到您的应用程序运行缓慢且跳帧。当您的应用程序分配内存的速度快于系统收集内存的速度时，就会发生这种情况。

当你泄漏内存时，它不能被释放回堆。这会强制执行不必要的垃圾收集事件并减慢系统的其余部分。最终，系统可能会杀死您的应用程序进程以回收内存。

使用 Android Profiler

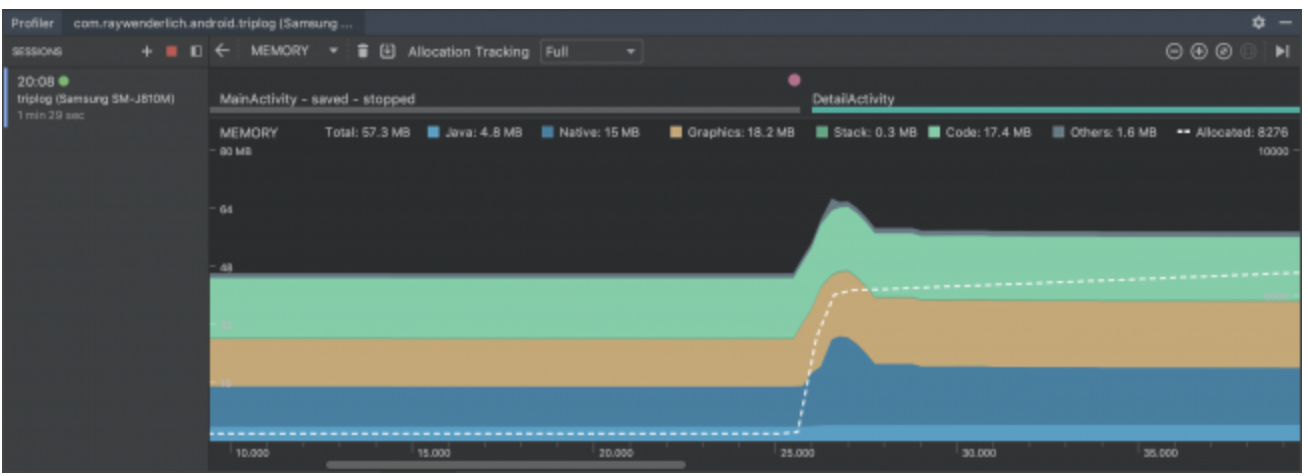


现在，单击Memory部分：



在顶部，您会看到当前Activity。在这种情况下，它是 MainActivity。

在应用程序中，按 + 按钮添加新日志并检查分析器：



它现在显示 DetailActivity 是当前Activity和一个小粉红点。每次触摸屏幕时，分析器都会显示这些点。稍后，当在设备或模拟器上按下它们时，还会看到其他图标，例如后退或键盘。

在其下方，可以看到应用程序的内存计数，分为几类：

Java：来自 Java/Kotlin 已分配的对象的内存。

Native：从 C/C++ 代码对象分配的内存。

Graphics：在屏幕上显示像素的内存。

Stack：应用程序中本机堆栈和 Java 堆栈使用的内存。当您的应用调用方法时，会在堆栈内存中创建一个块来保存本地原始值和对该方法中其他对象的引用。

Code：用于代码和资源的内存，例如 dex 字节码、.so 库和字体。

Others：系统不知道如何分类的内存。

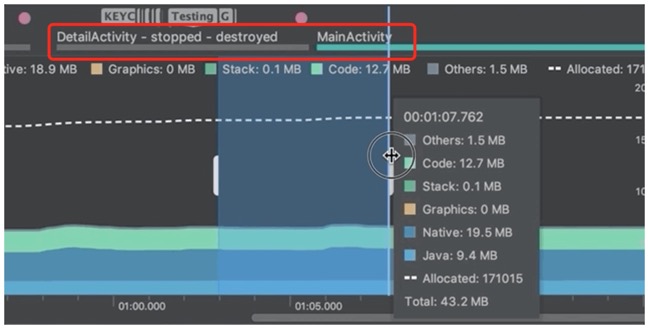
Allocated：应用已分配的 Java/Kotlin 对象的数量。

注意：如果您运行的是 Android 7.1 或更低版本的设备或模拟器，则此分配计数仅在 Memory Profiler 连接到您正在运行的应用程序时开始计算。它不计算在您开始分析之前分配的任何对象。

分配跟踪

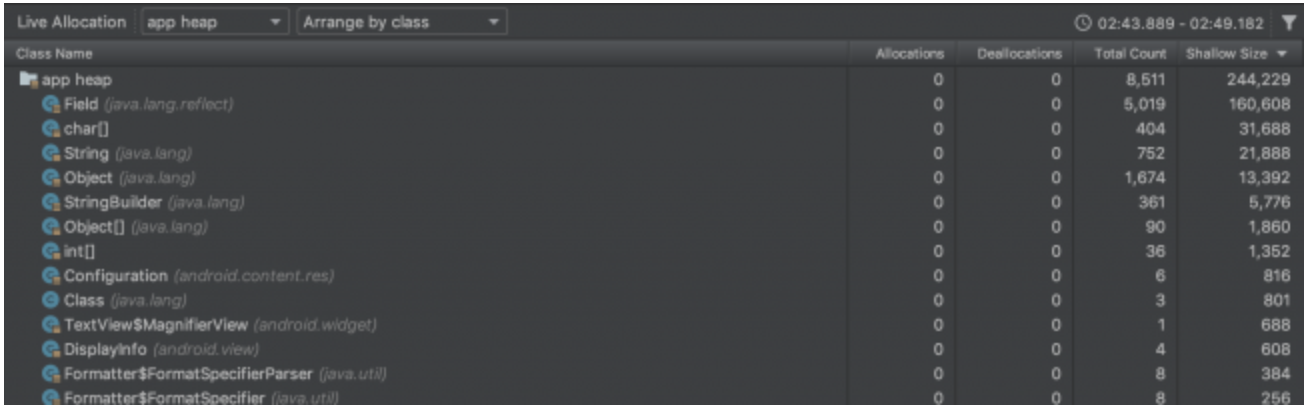
现在将看到如何分析何时在内存中创建 TripLog 对象。

在 Memory Profiler 中，如果运行的是 Android 8.0 或更高版本的设备，请拖入时间线以选择 DetailActivity 和 MainActivity 之间的区域。



注意：如果运行的是较低版本的设备，则在创建日志之前，需要单击记录内存分配。然后添加日志并按停止录制。

在时间线下方，profiler显示实时分配结果：



这显示了每个类的以下内容：

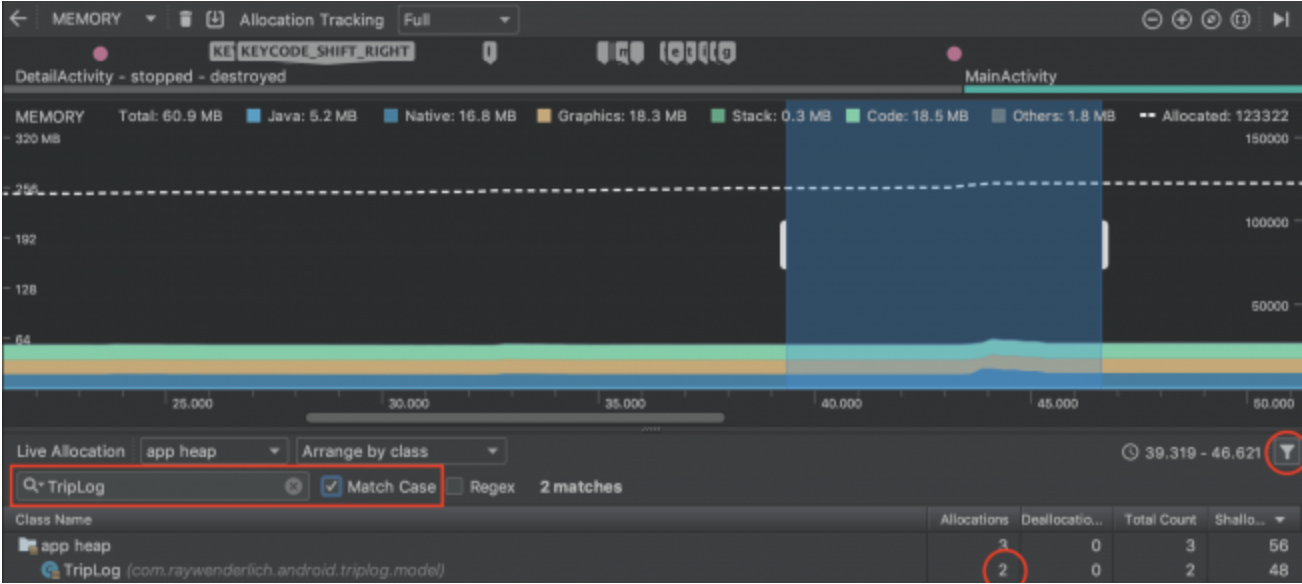
Allocations：在这段时间内分配的对象数。

Deallocations：在这段时间内解除分配的数量。

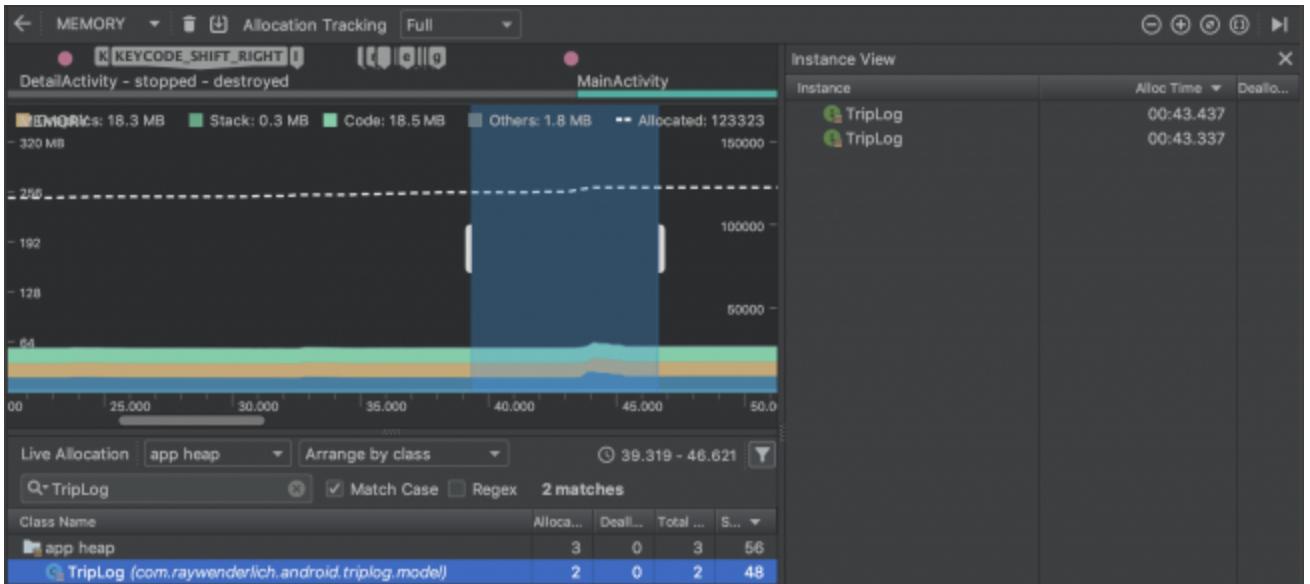
Total Count：仍分配给此类的对象数。

Shallow Size：用于此类对象的内存总字节数。

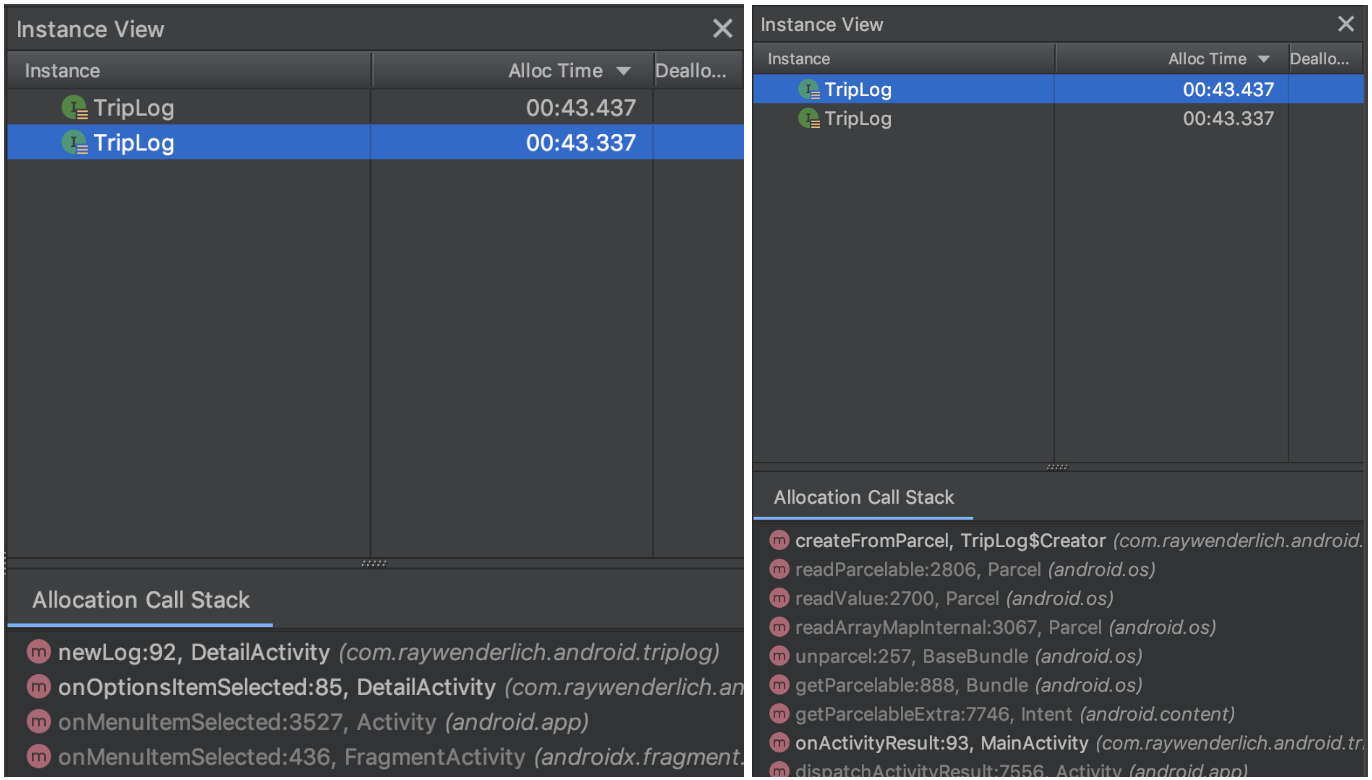
按 TripLog、匹配大小写过滤，您将看到以下内容：



虽然你可能没有想到，TripLog 有两种分配方式：



要找出原因，请单击实例视图的每一行以查看有关每个分配的更多信息：



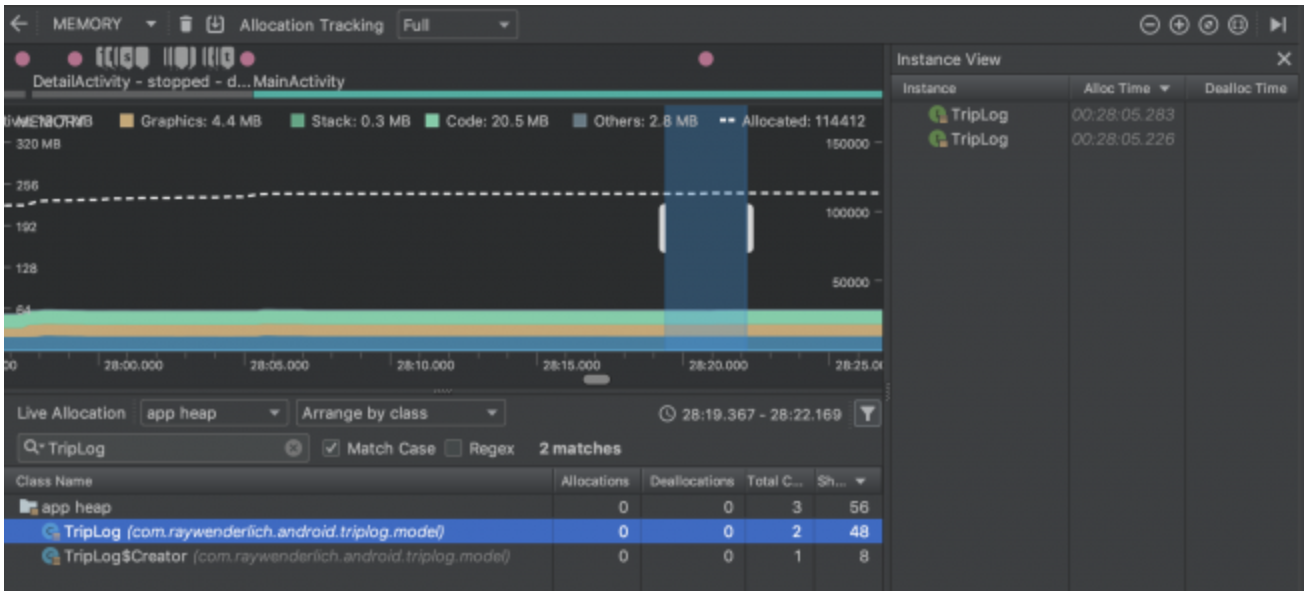
在分配调用堆栈中，在 onOptionsItemSelected() 方法中分配了一个 TripLog 实例。当按下右上角的复选按钮时会调用此方法。

由于解包，另一个是在 MainActivity 的 onActivityResult() 中创建的。

按主屏幕右上角的垃圾桶按钮删除日志。

在 Memory Profiler 中，滚动时间线直到按下按钮的那一刻。通过拖动时间线来执行分配跟踪，以查看按下该按钮时 TripLog 实例是否已解除分配。

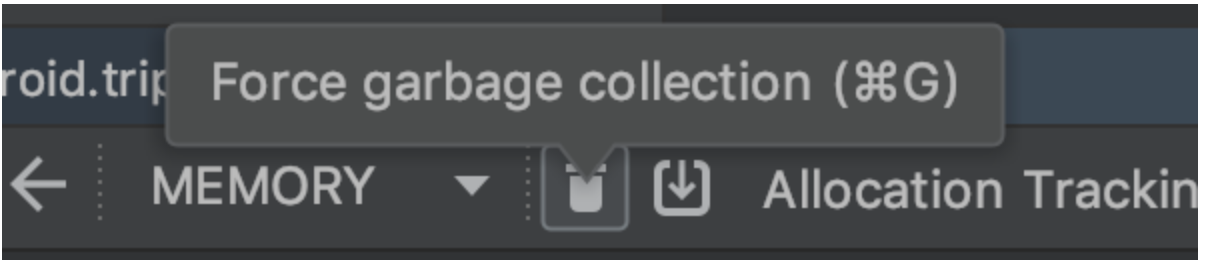
注意：如果使用的是 Android 7.1 或更低版本，请首先单击记录内存分配。然后删除日志并按停止录制。



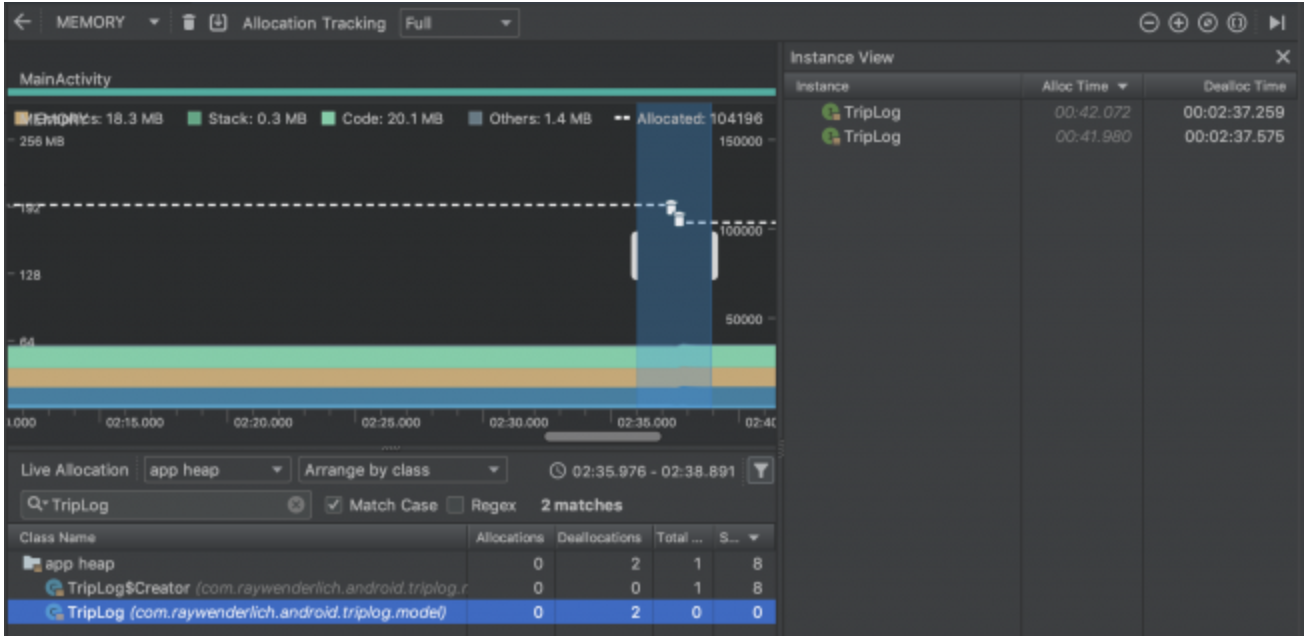
没有解除分配！你困惑吗？

这是因为垃圾收集器还没有通过。因此没有解除分配。

要强制进行垃圾回收，请按以下按钮两次：



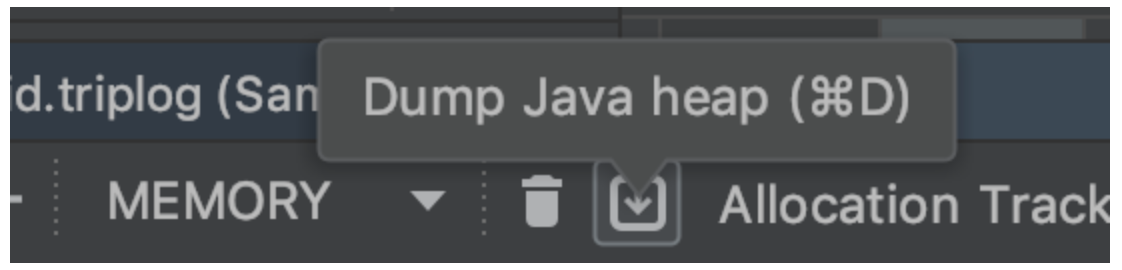
通过拖动时间线来执行分配跟踪，以查看实例最终被解除分配。



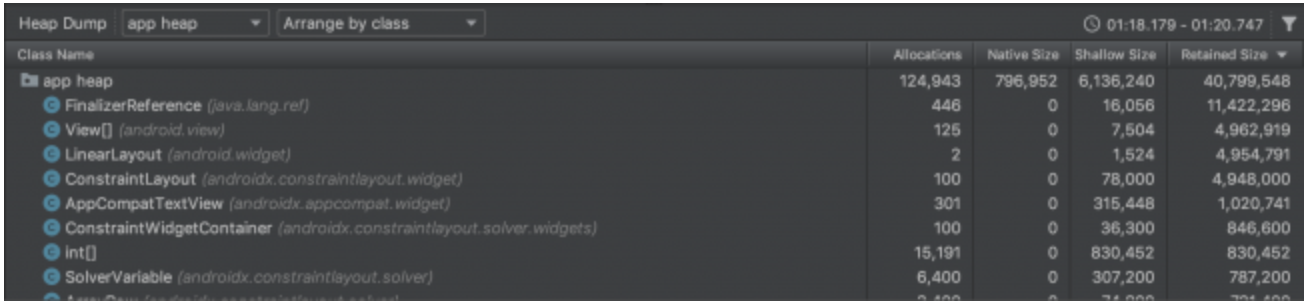
堆转储

还可以通过执行堆转储来分析内存的当前状态。

现在，在 Memory Profiler 中按下 Dump Java heap 按钮：



在时间线下方，将看到堆转储：



这显示了每个类的以下内容：

Allocations：分配的数量。

Native Size：本机内存的总字节数。

Shallow Size：Java 内存的总字节数。

Retained Size：由于此类的所有实例而保留的总字节数。

按保留大小排序，可以看到正好分配了 100 个 ConstraintLayout 对象。继续添加日志并捕获堆转储。您会看到，这个数字会随着您添加的日志数量的增加而增加。

这意味着应用程序正在为每个 TripLog 创建一个 ConstraintLayout。打开 MainActivity.kt 并检查 refreshLogs() 方法。

private fun refreshLogs() {

layoutContainer.removeAllViews()

repository.getLogs().forEach { tripLog ->

val child = inflateLogViewItem(tripLog)

layoutContainer.addView(child)

}

}

private fun inflateLogViewItem(tripLog: TripLog): View? {

return layoutInflater.inflate(R.layout.view\_log\_item, null, false).apply {

textViewLog.text = tripLog.log

textViewDate.text = dateFormatter.format(tripLog.date)

textViewLocation.text = coordinatesFormatter.format(tripLog.coordinates)

setOnClickListener {

showDetailLog(tripLog)

}

}

}

通过阅读此代码，您可以确认该应用程序正在为每个日志增加一个视图。如果你打开 view\_log\_item.xml 你会看到它是一个 ConstrainLayout。

这不好，因为它没有回收视图。这最终可能会创建 OutOfMemoryError 并在您的应用程序中生成崩溃。解决方案是使用 RecyclerView 重构应用程序。

打开 activity\_main.xml 并将 ScrollView 替换为以下内容：

<androidx.recyclerview.widget.RecyclerView

android:id="@+id/recyclerView"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"

tools:itemCount="3"

tools:listitem="@layout/view\_log\_item" />

打开 MainActivity.kt，删除 refreshLogs() 和 inflateLogViewItem() 方法。粘贴以下内容：

private fun refreshLogs() {

val adapter = TripLogAdapter(this, repository.getLogs(),

dateFormatter, coordinatesFormatter)

adapter.listener = this

recyclerView.adapter = adapter

}

让 MainActivity 实现以下监听器：TripLogAdapter.Listener {

创建一个名为 TripLogAdapter.kt 的新文件，其内容如下：

class TripLogAdapter(context: Context,

private val logs: List<TripLog>,

private val dateFormatter: DateFormatter,

private val coordinatesFormatter: CoordinatesFormatter

) : RecyclerView.Adapter<TripLogAdapter.TripLogViewHolder>() {

var listener: Listener? = null

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)

: TripLogViewHolder {

val inflater = LayoutInflater.from(parent.context)

val itemView = inflater.inflate(R.layout.view\_log\_item, parent, false)

return TripLogViewHolder(itemView)

}

override fun getItemCount() = logs.size

override fun onBindViewHolder(holder: TripLogViewHolder, position: Int) {

val tripLog = logs[position]

holder.bind(tripLog)

}

inner class TripLogViewHolder(itemView: View)

: RecyclerView.ViewHolder(itemView) {

private val textViewLog = itemView.textViewLog

private val textViewDate = itemView.textViewDate

private val textViewLocation = itemView.textViewLocation

fun bind(tripLog: TripLog) {

textViewLog.text = tripLog.log

textViewDate.text = dateFormatter.format(tripLog.date)

textViewLocation.text = coordinatesFormatter.format(tripLog.coordinates)

itemView.setOnClickListener {

listener?.showDetailLog(tripLog)

}

}

}

interface Listener {

fun showDetailLog(tripLog: TripLog)

}

}

再次分析应用程序并看到它现在创建了更少的 ConstraintLayout 对象，因为它正在回收视图。

频繁的垃圾收集

垃圾收集，也称为内存流失，发生在应用程序分配但又必须在短时间内释放对象时。

例如，如果您在循环中分配重对象，就会发生这种情况。在每次迭代中，应用程序不仅要分配一个大对象，还要从前一次迭代中释放它，这样它就不会耗尽内存。

由于频繁的垃圾收集，用户会注意到应用程序中的卡顿。这会导致糟糕的用户体验。

要查看实际情况，请在编写日志时添加一个切换按钮，以便用户也可以描述她或他的心情。因此，打开 activity\_detail.xml 并在 EditText 上方添加以下内容：

<ToggleButton

android:id="@+id/toggleButtonMood"

android:layout\_width="50dp"

android:layout\_height="50dp"

android:layout\_gravity="center"

android:background="@drawable/bg\_selector\_mood"

android:checked="true"

android:textOff=""

android:textOn="" />

要在主屏幕中查看每个日志的心情，您需要更新 TripLogAdapter 类。所以打开它并用这个更新 TripLogViewHolder(itemView: View) 内部类：

inner class TripLogViewHolder(itemView: View)

: RecyclerView.ViewHolder(itemView) {

...

private val imageView = itemView.imageView

fun bind(tripLog: TripLog) {

val happyBitmap = BitmapFactory.decodeResource(itemView.context.resources,

R.drawable.bg\_basic\_happy\_big)

val sadBitmap = BitmapFactory.decodeResource(itemView.context.resources,

R.drawable.bg\_basic\_sad\_big)

imageView.setImageBitmap(if (tripLog.happyMood) happyBitmap else sadBitmap)

...

打开将自动开始监控的 Android Profiler 或如前所述手动添加当前会话。

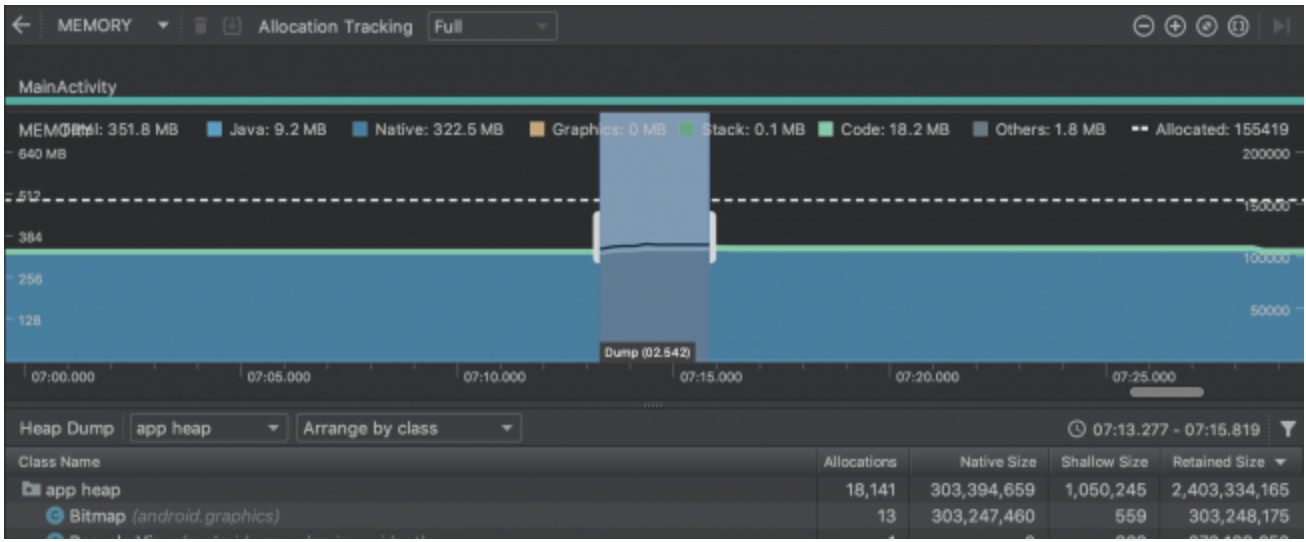
会注意到的第一件事是总内存比以前高。此外，尝试在应用程序中滚动，您会看到类似以下内容：



系统频繁触发垃圾收集器。您还可以通过在时间线中进行一些分配跟踪来确认这一点。你会看到被分配的对象数量几乎与被释放的对象数量相同。

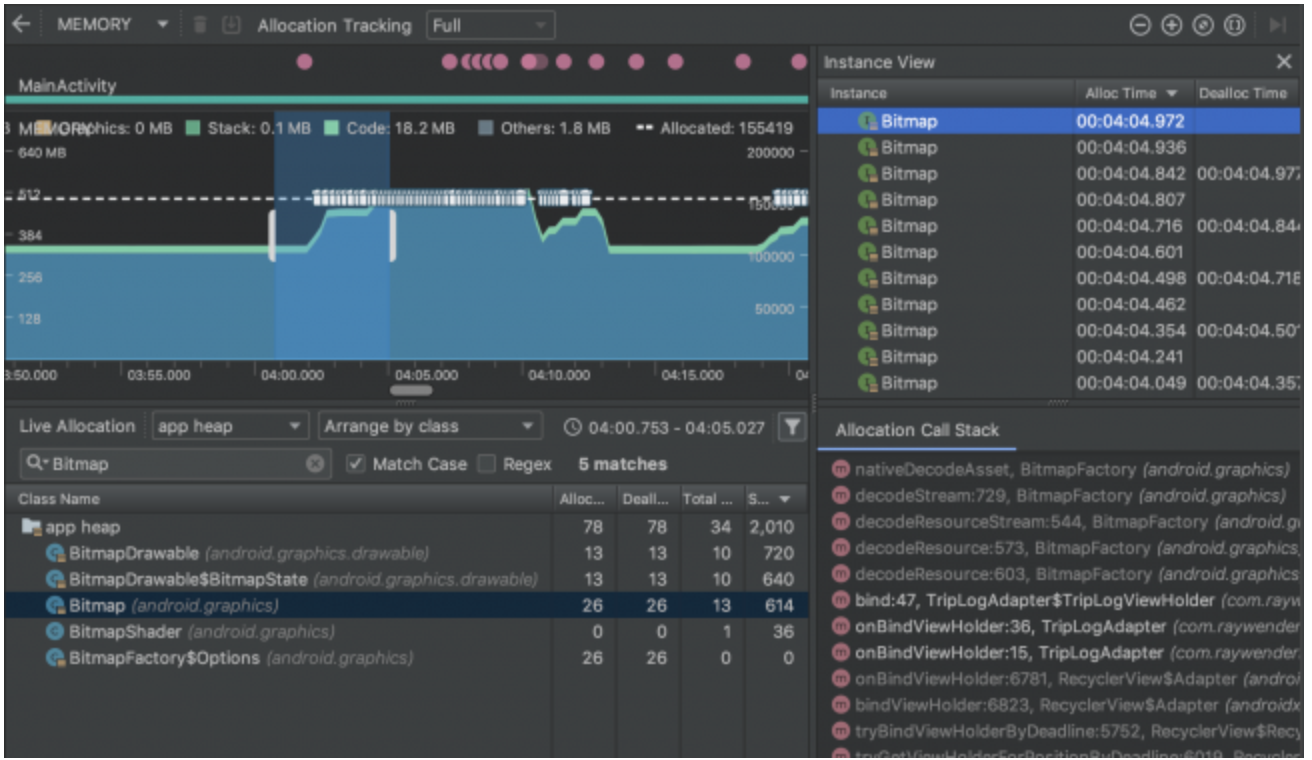
这显然不好。频繁的垃圾收集会导致性能下降。

现在，执行堆转储，您将看到如下内容：



在这里你可以找到一个线索。有一些 Bitmap 对象会保留大量内存。

因此，在创建日志时再次进行一些分配跟踪。按位图过滤。您可以单击其中一个以查看它们的创建位置：



双击分配调用堆栈中的bind方法，您将转到源代码：

fun bind(tripLog: TripLog) {

val happyBitmap = BitmapFactory.decodeResource(itemView.context.resources,

R.drawable.bg\_basic\_happy\_big)

val sadBitmap = BitmapFactory.decodeResource(itemView.context.resources,

R.drawable.bg\_basic\_sad\_big)

imageView.setImageBitmap(if (tripLog.happyMood) happyBitmap else sadBitmap)

...

似乎这些行导致了问题。每次绑定日志时，应用程序都会不必要地解码 R.drawable.bg\_basic\_happy\_big 和 R.drawable.bg\_basic\_sad\_big。

因此，删除这些行并修改 TripLogAdapter.kt，如下所示：

class TripLogAdapter(context: Context,

private val logs: List<TripLog>,

private val dateFormatter: DateFormatter,

private val coordinatesFormatter: CoordinatesFormatter

) : RecyclerView.Adapter<TripLogAdapter.TripLogViewHolder>() {

private val happyBitmap = BitmapFactory.decodeResource(

context.resources,

R.drawable.bg\_basic\_happy\_big)

private val sadBitmap = BitmapFactory.decodeResource(

context.resources,

R.drawable.bg\_basic\_sad\_big)

在这里，您只对可绘制对象进行一次解码，而不是在每次绑定时都进行解码。没有更频繁的垃圾收集，因为您为 Bitmap 对象正确分配了内存。

再次构建并运行应用程序，您会看到性能得到了增强！

内存泄漏

当您的代码为对象分配内存但从未解除分配时，就会发生内存泄漏。

不管是什么原因，当发生内存泄漏时，垃圾收集器认为仍然需要一个对象，因为它仍然被其他对象引用。但是那些引用应该已经清除了。

为泄漏对象分配的内存充当不可移动的块，迫使应用程序的其余部分在堆的剩余部分中运行。这可能会导致频繁的垃圾收集。如果应用程序继续泄漏内存，它最终会用完并崩溃。

有时，泄漏是大而明显的。其他时候它很小。较小的泄漏更难发现，因为它们通常发生在长时间使用应用程序之后。

常见的内存泄漏

你经历过一种内存泄漏的情况。现在，您将了解其他一些常见情况以及导致它们的原因。您还将学习如何使用 Android Profiler 和 Leak Canary 检测和避免它们。

匿名类

有时，匿名类实例的寿命比容器实例的预期寿命长。如果这个匿名类实例调用任何方法，或者读取或写入容器类的任何属性，它将保留容器实例。这会泄漏容器的内存。

注意：如果匿名类实例不调用任何方法，也不读取或写入容器类的任何属性，则不会保留它。在这种情况下，您不会泄漏容器的内存。

LeakCanary 如何显示内存泄漏

LeakCanary 监控被销毁的Activity。它等待五秒钟，然后强制进行垃圾收集。如果Activity仍然存在，LeakCanary 认为它被保留并可能泄漏。

转储堆后，LeakCanary 会找到从 GC 根到保留实例的最短强引用路径。这也称为泄漏跟踪。

内部类

内部类是内存泄漏的另一个常见来源，因为它们也可以引用其容器类。例如，假设您有以下 AsyncTask：

class MyActivity : AppCompatActivity() {

...

inner class MyTask : AsyncTask<Void, Void, String>() {

override fun doInBackground(vararg params: Void): String {

// Perform heavy operation and return a result

}

override fun onPostExecute(result: String) {

this@MyActivity.showResult(result)

}

}

}

如果离开Activity并且任务未完成，这将产生泄漏。可以尝试在 onDestroy() 方法中取消 AsyncTask。但是，由于 AsyncTask 的工作方式，doInBackground() 方法不会取消，仍然会继续泄漏Activity。

要修复它，应该删除内部修饰符以将 MyTask 转换为静态类。静态内部类无法访问容器类，因此您将无法泄漏活动。但是你也不能调用 showResult。

class MyActivity : AppCompatActivity() {

...

class MyTask(private val activity: MainActivity) : AsyncTask<Void, Void, String>() {

override fun doInBackground(vararg params: Void): String {

// Perform heavy operation and return a result

}

override fun onPostExecute(result: String) {

activity.showResult(result)

}

}

}

但是这样也会泄露Activity。一个可能的解决方案是使用 WeakReference：

class MyActivity : AppCompatActivity() {

...

class MyTask(activity: MainActivity) : AsyncTask<Void, Void, String>() {

private val weakRef = WeakReference<MyActivity>(activity)

override fun doInBackground(vararg params: Void): String {

// Perform heavy operation and return a result

}

override fun onPostExecute(result: String) {

weakRef.get()?.showResult(result)

}

}

}

WeakReference 像普通引用一样引用对象，但强度不足以将其保留在内存中。因此，当垃圾收集器通过并且没有找到对该对象的任何强引用时，它会收集它。任何引用收集到的对象的 WeakReference 都将设置为 null。

注意：在组合匿名类或内部类时，您应该特别注意您的代码以避免这些类型的泄漏。在处理延迟任务或与线程相关的任何内容时，您还需要特别注意，例如 Handlers、TimerTasks、Threads、AsyncTasks 或 RxJava。

静态变量

伴随对象内的变量是静态变量。这些变量与一个类相关联，而不是与该类的实例相关联。因此，自从系统加载类后，它们就会存在。

应该避免让静态变量引用Activity。即使不再需要它们，它们也不会被垃圾收集。

单例

如果单例对象包含对Activity的引用并且比Activity寿命更长，那么将泄漏它。

作为一种解决方法，您可以在单例对象中提供一个清除引用的方法。可以在Activity的 onDestroy() 中调用该方法。

注册监听器

在许多 Android API 和外部 SDK 上，必须将Activity订阅为侦听器并提供回调方法来接收事件。例如，需要为位置更新和系统事件执行此操作。

如果忘记取消订阅，这可能会导致内存泄漏。通常，订阅的对象的寿命比Activity长。