第 5 章

JVM 调 优

在名为“虚拟机优化”的小国中，有一位被称为“JVM 调优”的智者。他的首要任务是协助民众解决虚拟机在运行过程中出现的性能问题，以提高资源利用率，确保国家的繁荣与稳定。某日，虚拟机优化国度的一台服务器遇到性能瓶颈问题。得知此情况后，国王立刻邀请 JVM 调优前来协助。

JVM 调优毫不犹豫地接受了邀请，迅速赶往现场。经过深入分析，他发现虚拟机内存和 CPU 资源消耗过高，导致程序运行缓慢。此外，还发现了一些无用的垃圾回收器，消耗了大量的系统资源。为解决这些问题，JVM 调优采取了一系列优化措施。首先，他为服务器设定了合适的内存和 CPU 资源限制，确保虚拟机能够高效运行。接着，关闭无用的垃圾回收器，减轻系统负担。在 JVM 调优的努力下，服务器性能大幅提升。程序运行速度加快，资源消耗降低，虚拟机恢复了正常运行。国王对 JVM 调优的表现深感满意，感激他为国家的虚拟机运行带来了质的飞跃。自此，JVM 调优成了虚拟机优化国度的守护者，时刻关注虚拟机的运行状况，确保国家的繁荣与稳定。他的故事激励着虚拟机优化国度的每位成员，使他们更加关注虚拟机性能调优，共同为国家的繁荣与稳定贡献力量。

5.1 JVM 调优目的原则

JVM 调优的主要目的是减少 GC 的频率和 Full GC 的次数，并降低 STW 的停顿时间和次数，以提高系统的性能和稳定性。为达到这个目的，需要进行 JVM 优化，而 JVM 优化的原则可以归纳为以下几点。

首先，尽可能地让对象都在新生代里分配和回收。由于新生代的垃圾回收速度比老年代要快得多，因此将对象尽量分配到新生代中可以减少老年代的负担，降低 GC 的频率和 Full GC 的次数。为了避免大量对象进入老年代，可以设置适当的新生代大小和比例，以确保不会频繁地进行老年代的垃圾回收。

其次，给系统充足的内存大小。为了避免频繁地对垃圾进行回收和 Full GC，可以适当地增加系统的内存大小。此外，还可以设置合理的堆空间大小，使堆空间不会快速被占满。这样可以减少 GC 的频率和 Full GC 的次数，降低 STW 的停顿时间和次数，提高系统的稳

定性和性能。

最后,避免频繁地进行老年代的垃圾回收。老年代的垃圾回收通常是比较耗时的,因此应该尽量避免频繁地进行老年代的垃圾回收。可以通过合理设置新生代大小、年龄等参数及使用 CMS 等垃圾回收器来减少对老年代的垃圾回收,从而降低 GC 的频率和 Full GC 的次数,提高系统的性能和稳定性。

综上所述,JVM 调优是一个综合性的工作,需要根据具体的应用场景和系统需求进行优化。在实际操作中,需要结合以上原则,采取合理的措施和手段,不断地进行优化和调整,以提高系统的性能和稳定性。

5.2 Full GC 发生的原因

Full GC 是指对 Java 虚拟机堆内内存进行的垃圾回收,会对 Java 应用程序的性能造成很大影响。Full GC 发生的原因有很多,需要通过调优来避免或者减少其发生。

首先,应尽量避免使用 System.gc()方法,因为调用该方法会建议 JVM 进行 Full GC,这可能会增加 Full GC 的频率,从而增加间歇性停顿的次数。为了减少该方法的使用,可以禁止 RMI 调用 System.gc(),通过-XX:+DisableExplicitGC 参数实现。

其次,如果 Survivor 区域的对象满足晋升到老年代的条件,但是当晋升到老年代的对象大小大于老年代的可用内存时,就会触发 Full GC。为了避免这种情况,可以通过调整 JVM 参数或者设计应用程序的算法,来减少对象在老年代的数量。

从 JDK 8 开始,Metaspace 区取代了永久代(PermGen),Metaspace 使用的是本地内存。通过调整 JVM 参数限制 Metaspace 的大小。当 Metaspace 区内存达到阈值时,也会引发 Full GC。可以通过调整 JVM 参数或者设计应用程序的算法,来减少 Metaspace 区内存的使用。

当 Survivor 区域的对象满足晋升到老年代的条件时,也可能会引起 Full GC。可以通过调整 JVM 参数来控制对象的晋升行为,从而减少 Full GC 的发生。

此外,如果堆中产生的大对象超过阈值,则会引发 Full GC。可以通过调整 JVM 参数或者优化应用程序算法,来减少大对象的产生。

最后,老年代连续空间不足或者 CMS GC 时出现 promotion failed 和 concurrent mode failure 所致的 Full GC,也可以通过调整 JVM 参数或者设计应用程序算法来减少其发生。

总之,对于 Full GC 的频繁发生,需要分析具体原因,通过调整 JVM 参数或者优化应用程序的算法,来减少其发生,提高 Java 应用程序的性能。

以下是可能引起内存泄漏并导致 Full GC 的一些情况。

(1) 对象的长期存活:如果某些对象在 JVM 中存活了很长时间,则可能会导致内存泄漏,并在堆积积累的过程中触发 Full GC。

(2) 大对象:如果程序中创建了大的对象,但这些对象无法被回收,则可能会导致内存泄漏,并触发 Full GC。

(3) 永久代内存溢出：当应用程序使用大量字符串或其他可序列化的类时，可能会导致永久代内存耗尽，并触发 Full GC。

(4) 字符串：如果在程序中频繁创建字符串，并且它们不被清除，则可能会导致内存泄漏，并触发 Full GC。

(5) 无用的类和对象：如果程序中存在许多无用的类和对象，则可能会导致内存泄漏，并触发 Full GC。

(6) ThreadLocal：如果程序中使用了 ThreadLocal，但没有正确地清除线程本地存储，则可能会导致内存泄漏，并触发 Full GC。

(7) 频繁创建对象：如果程序中频繁地创建对象，但没有正确地清除这些对象，则可能会导致内存泄漏，并触发 Full GC。

综上所述，内存泄漏可能会导致 Full GC，从而降低应用程序的性能，因此，在开发和调试过程中需要注意内存的使用情况，以及时发现和解决潜在的内存泄漏问题。

5.3 常用的工具

本节介绍几种常用的工具。

5.3.1 Jstack

Jstack 是用于获取 Java 线程转储的工具，适用于在程序出现死锁、线程挂起等问题时，通过获取线程转储，更好地诊断和解决问题。在找出占用 CPU 最高的线程堆栈信息时，可以按以下步骤操作：

(1) 打开命令行窗口，并进入 Java 应用程序所在的目录。

(2) 在命令行中输入以下命令，查询 Java 应用程序的进程 id(PID)，代码如下：

```
ps -ef | grep java
```

该命令将返回所有正在运行的 Java 应用程序进程的详细信息，需要查找要分析的 Java 应用程序进程的 PID。

(3) 输入以下命令，使用 Jstack 导出 Java 应用程序的线程堆栈信息(将 PID 替换为前面查询到的 Java 应用程序进程的 PID)：

```
jstack PID >thread_dump.txt
```

该命令将会把当前时间点 Java 应用程序的线程堆栈信息导出到 thread_dump.txt 文件。

(4) 打开 thread_dump.txt 文件，查找占用 CPU 最高的线程。在文件中，可以查找到每个线程的 ID、状态和堆栈信息。找到 CPU 使用率最高的线程，查看其堆栈信息，尝试从中找到问题所在。可以使用线程 ID 在文件中搜索线程的堆栈信息，代码如下：

```
grep "nid=0x1234" thread_dump.txt
```

(5) 根据线程堆栈信息,定位并解决问题。根据线程堆栈信息,可以判断线程是否处于阻塞状态,以及是否存在死锁等问题,从而定位并解决问题。

通过上述步骤,可以很好地使用 Jstack 工具找出 CPU 使用率最高的线程堆栈信息,并定位和解决问题。

5.3.2 Jmap

Jmap 是一款 JDK 自带的命令行工具,用于获取 Java 进程的内存使用情况。通过 Jmap,可以获取 Java 进程中每个对象的内存使用情况,帮助定位可能存在的内存泄漏问题。使用 Jmap 获取内存快照的步骤如下:

(1) 在命令行窗口中进入 JDK 的 bin 目录下。

(2) 输入 jps 命令,获取 Java 进程的进程 ID,该命令会显示当前系统中所有正在运行的 Java 进程的进程 ID。

(3) 输入 jmap-dump:format=b,file=<文件名>.bin<进程 ID>命令生成 Java 进程的内存快照,其中,-dump:format=b 表示以二进制格式生成内存快照文件,<文件名>.bin 表示生成的文件名,而<进程 ID>则为第(2)步中获取的 Java 进程的进程 ID。

(4) 使用 Java 内存分析工具(如 Eclipse Memory Analyzer(MAT)、jProfiler 或 YourKit 等)对 Jmap 生成的内存快照文件进行分析。通过分析,可以识别 Java 进程中的内存泄漏问题,找出占用内存过多的对象并及时进行清理,提高系统的性能和稳定性。

使用 Jmap 获取内存快照还有助于优化程序的设计和实现,减少内存的使用,提高程序效率,因此,Jmap 是一款非常有用的工具,对于 Java 开发人员来讲是不可或缺的。

5.3.3 Jstat

Jstat 是一款由 Java 虚拟机提供的命令行工具,可以用于监控 Java 应用程序的性能指标。Jstat 可以提供多种性能指标,包括堆内存使用情况、类加载情况、垃圾回收情况、JIT 编译情况和线程情况等。使用 Jstat,可以实时观察 Java 应用程序的性能指标,以便及时识别出性能瓶颈。同时,可以根据具体需求调整数据采集的频率。

举例来讲,Jstat 可以通过以下格式来监控 Java 进程的各项性能指标:jstat -option pid interval,其中 option 表示需要监测的性能指标,pid 是 Java 进程的进程 ID,interval 是每隔多长时间(单位为毫秒)收集一次性能数据。例如,要监控 Java 进程的堆内存使用情况,可以使用 jstat -gcutil pid 1000 命令。同理,要监测类加载情况,可以使用 jstat -class pid 1000 命令;要监测垃圾回收情况,可以使用 jstat -gc pid 1000 命令;要监测 JIT 编译情况,可以使用 jstat -compiler pid 1000 命令;要监测线程情况,可以使用 jstat -thread pid 1000 命令。

总之,Jstat 是一款非常实用的工具,可以更好地了解 Java 应用程序的性能状况,以便及时诊断和解决性能问题,提高 Java 应用程序的性能表现。

5.3.4 JConsole

JConsole是Java Development Kit(JDK)自带的监控和管理工具,它能够连接到正在运行的Java应用程序,并通过Java Management Extensions(JMX)协议来实时监控应用程序的各种状态和性能指标,例如线程数、内存使用情况、GC情况等,以便更好地发现和解决问题。

使用JConsole可以获得以下功能:

(1) 监控Java应用程序的基本信息,包括内存使用情况、线程数、CPU使用情况和类加载情况等,以便更好地了解应用程序的整体运行状况。

(2) 查看Java虚拟机的垃圾回收情况,包括GC时间、频率和类型等信息,以便更好地了解GC对应用程序的影响。

(3) 利用JConsole的线程和死锁检测工具,检测和解决应用程序中的线程问题。

(4) 通过JConsole的MBean查看应用程序的特定指标,例如数据库连接池的使用情况、消息队列处理速度等。

(5) 使用JConsole进行故障排除。如果Java应用程序出现故障,则可以使用JConsole进行排除。在JConsole中可以查看Java应用程序的线程状态,找出导致故障的线程。可以使用JConsole的线程调试功能,设置断点并调试线程代码,找出导致故障的原因。在内存选项卡中可以查看Java应用程序的内存使用情况,如果发现内存泄漏,则可以使用JConsole的内存分析功能,分析内存快照并找出内存泄漏的原因。

(6) 通过JConsole的可视化界面,可以实时监控应用程序的性能指标,以及分析和调试应用程序的性能瓶颈。在JConsole中打开概述选项卡,可以查看Java应用程序的概览信息,如堆使用情况、线程数、类加载数等。在线程选项卡中,可以查看Java应用程序的线程状态,如线程数、线程状态、死锁情况等。在内存选项卡中,可以查看Java应用程序的内存使用情况,如堆内存、非堆内存、Eden空间、Survivor空间、老年代等。在GC选项卡中,可以查看Java应用程序的GC情况,如GC时间、GC频率、GC类型等。

要使用JConsole来监控Java应用程序,首先需要在启动目标应用程序时指定JMX参数。例如,可以通过命令行参数-Dcom.sun.management.jmxremote.port=909将JMX端口设置为9090,通过-Dcom.sun.management.jmxremote.ssl=false关闭JMX SSL验证,以及通过-Dcom.sun.management.jmxremote.authenticate=false关闭身份验证功能,然后在JConsole中连接到该应用程序,即可开始监控和管理它。

总体来讲,JConsole是一个非常实用的Java应用程序监控和管理工具,它可以帮助开发人员更好地了解和调试Java应用程序的各种问题。

5.3.5 VisualVM

VisualVM是一款Java虚拟机(JVM)监视和分析工具,可用于监控和分析Java应用程序的性能和内存使用情况。它可以获取Java应用程序的堆转储、线程转储、性能指标和内

存使用情况等信息,以便更好地诊断和解决问题。同时,VisualVM 也是一款独立的工具,与 Java 开发工具包(JDK)相结合,可形成 JVisualVM。实际上,JVisualVM 就是在 VisualVM 名称前添加了一个 J 的版本,以体现它与 Java 密切相关。

使用 JVisualVM 进行故障排除,可以通过导入 dump 文件来分析 Java 应用程序的详细信息和分析视图。使用 JVisualVM 的步骤如下:

(1) 下载并安装 JVisualVM,打开 https://visualvm.github.io/网址,下载最新版本的 JVisualVM 并启动该工具。

(2) 打开导入 dump 文件的选项,单击 File 选项卡,并选择 Load,然后导航到 dump 文件所在的目录并选择该文件,最后单击 Open 按钮打开文件。

(3) 导入 dump 文件后,JVisualVM 将在屏幕左侧的 Applications 选项卡中显示 Java 应用程序。单击 Java 应用程序,将显示应用程序的详细信息和分析视图,可以从概述视图(Overview)、监视视图(Monitor)、线程视图(Threads)等多个视图中选择以查看应用程序的不同方面。

(4) 如果需要分析 Java 应用程序的堆内存,则可以单击 Java 应用程序的名称,并单击 HeapDump 按钮,以打开一个新的窗口,其中包含在堆中使用的所有对象的详细信息。通过这些信息可以分析内存泄漏和其他相关问题。

总之,使用 JVisualVM 可以帮助开发者快速定位 Java 应用程序的性能和内存问题,提高开发效率。

5.3.6 Arthas

Arthas 是一款非常实用的 Java 诊断工具,它可以帮助开发和运维人员在线上快速定位问题,提高效率。Arthas 支持丰富的命令和界面,易于使用,而且非常灵活,可以满足不同场景下的需求。在使用 Arthas 时,需要注意一些细节,例如不能直接在生产环境中进行调试,需要在测试环境中模拟相应的情况进行调试。只有在必要的情况下才能使用 Arthas,以避免对系统的影响。Arthas 具有无侵入式和全方位诊断的特点,可以监控方法执行时间、线程状态、GC 情况等,对于内存泄漏和死锁等问题也能够进行针对性诊断。此外,Arthas 还集成了丰富的命令和界面,易于使用。由于 Arthas 是在 JVM 层面进行诊断,因此不适用于诊断一些操作系统级别的问题,例如 I/O 和网络等。在高并发情况下,Arthas 会对 JVM 产生一定的影响,因此在使用时需要注意。

安装 Arthas 非常简单,只需下载并解压缩。下载链接为 https://alibaba.github.io/arthas/arthas-boot.jar。

解压缩后会看到以下几个文件。

(1) arthas-boot.jar: Arthas 的启动程序,也是使用 Arthas 的主入口。

(2) arthas-client.jar: Arthas 的客户端程序,用于和 Arthas 服务器端进行通信。

(3) arthas-demo.jar: Arthas 的示例代码,内部包含了大量使用示例。

(4) asm-all-5.0.3.jar: Arthas 所依赖的 asm 库。

(5) fastjson-1.2.29.jar：Arthas 所依赖的 fastjson 库。

(6) netty-all-4.1.48.Final.jar：Arthas 所依赖的 netty 库。

(7) watch-logback.xml：Arthas 的日志配置文件。

启动 Arthas 只需在命令行中输入 java-jararthas-boot.jar 命令。成功启动后，可以使用 ps 命令查看当前系统中的 Java 进程，并使用 JVM 命令连接到指定的 Java 进程。连接成功后，可以通过 trace 命令监控方法的执行时间和调用链，通过 thread 命令监控线程的状态，通过 gc 命令监控 JVM 的 GC 状态，通过 heapdump 命令查看 JVM 的内存信息等。除此之外，Arthas 还支持其他许多命令，例如 dashboard(显示 JVM 的运行状态，包括 CPU 使用率、内存使用情况、线程状态等)、反编译 Java 类(jad)、查看类加载器信息(classloader)、监控变量的值变化(watch)和查看当前类加载器的 classpath(sc)等。

总之，Arthas 是一款非常实用的 Java 诊断工具，它能够帮助开发和运维人员在线上快速定位问题，提高效率。使用 Arthas 时要注意一些细节，例如需要在测试环境中模拟相应的情况进行调试，避免直接在生产环境中进行调试。只有在必要的情况下才能使用 Arthas，以避免对系统的影响。

5.4 JVM 排查

本节介绍 JVM 排查问题的详细步骤。

5.4.1 收集问题信息

在软件开发和维护过程中，出现问题是非常常见的。为了解决问题，首先需要收集足够的信息，以便更好地理解问题。在收集信息的过程中，可以使用工具(如 Jstack、Jmap、Jstat、JConsole、VisualVM 等)收集日志文件、异常堆栈跟踪、线程转储、内存使用情况、GC 日志等信息。这些信息可以帮助确定问题的性质、影响范围及可能的解决方案。

具体来讲，日志文件用于记录系统在运行过程中的信息，如错误日志、调试日志等，可快速定位问题所在。异常堆栈跟踪可获取相关的堆栈跟踪信息，如异常发生的位置、类型和调用堆栈等。线程转储可获取系统中各个线程的状态、调用堆栈等信息。内存使用情况可告知系统中各个对象的大小、数量等信息。GC 日志可获取相关的垃圾回收信息，如垃圾回收的频率、耗时等。

在收集信息时，需要注意信息应具有代表性且及时、全面、准确，避免因信息不足或不准确而导致难以解决问题。同时，收集的信息应进行整理和分析，以便更好地诊断和解决问题。

综上，收集足够的信息是解决问题的第 1 步，而合理使用工具和技术可以帮助更好地获取和分析信息，从而更好地解决问题。

5.4.2 确定问题的类型

为了进行JVM调优,需要根据收集到的信息来确定问题类型,如线程死锁、内存泄漏、垃圾回收性能等。在了解问题类型之后,可以选择相应的解决方案进行调优,如通过调整堆内存大小、垃圾回收算法、线程优化等手段来解决问题。调优后还需要对系统进行测试,以验证是否达到了预期效果。JVM是一个虚拟机,包含垃圾回收器、类加载器、解释器、即时编译器等组成部分,其工作原理是将Java代码编译成字节码,在运行时通过JIT即时编译器将其转换为本地机器码,再由解释器执行。同时,JVM通过垃圾回收器来管理内存,避免内存泄漏和OutOfMemoryError等问题的发生。在进行JVM调优时,需要先确定问题类型,如线程死锁、内存泄漏、垃圾回收性能等。解决线程死锁需要了解哪些线程互相持有哪些资源,并考虑采取什么方式来避免死锁。解决内存泄漏问题需要了解哪些对象无法被回收,以及这些对象的引用链是怎样的,从而采取相应的措施来避免内存泄漏。解决垃圾回收性能问题可以通过调整堆内存大小、垃圾回收算法、GC线程数等手段来解决。在进行JVM调优时,需要深入了解JVM内部的工作原理,并掌握收集信息的技术实现步骤,以实现系统的最佳性能。

5.4.3 检查JVM配置

JVM可能出现问题的原因不仅限于代码本身,还包括一些配置问题。例如,堆大小设置过小、PermGen空间设置过小、CPU核心数过少、文件描述符数设置过小等。这些问题需要得到重视,尤其是在生产环境中。

首先,堆大小对JVM运行非常重要。由于Java堆是存储对象实例的地方,因此堆大小设置过小会导致JVM无法为应用程序分配足够的内存,最终会导致出现OutOfMemoryError异常。因此,在检查堆大小设置时,应根据应用程序的实际内存使用情况进行设置,大型应用程序需要更大的Java堆,而小型应用程序则可以使用较小的Java堆。

其次,PermGen空间也是JVM运行的关键因素。PermGen空间用来存储类信息和常量池等信息,因此,如果应用程序需要加载大量的类或者使用大量常量,则PermGen空间就应该设置得更大。如果PermGen空间被设置得过小,JVM则将无法为应用程序分配足够的内存,最终导致出现OutOfMemoryError异常。

再次,CPU核心数的设置也会对应用程序的性能产生直接影响。多核处理器可以同时执行多个线程,提高应用程序的并发处理能力。如果应用程序需要处理大量并发操作,则CPU核心数就应该设置得更多,以提高应用程序的性能。如果CPU核心数被设置得过少,JVM则将无法充分利用CPU的多核处理能力,最终导致应用程序的性能受到影响。

最后,文件描述符数的设置也是一个重要的配置问题。在Linux系统下,文件描述符是指为了访问文件而打开的文件句柄。如果应用程序需要处理大量文件或者网络连接,则文件描述符数就应该被设置得更多,以提高应用程序的性能。如果文件描述符数被设置得过

小,JVM 则将无法打开足够的文件句柄,最终导致应用程序的错误或者异常,因此,在生产环境中,需要对这些配置问题进行仔细检查和优化。

5.4.4 分析堆转储

堆转储(HeapDump)是一种将 JVM 堆内存中的所有对象信息以二进制形式输出到文件的操作,可视为捕捉当前 JVM 堆内存状态的一种快照。它被广泛用于 Java 应用程序的内存分析和调试中,在开发和运维中常常用于解决 Java 应用程序占用内存过高、内存泄漏等问题。Jmap 工具是用于生成 JVM 的堆转储的常用工具,其命令格式为 jmap-dump:file=heapdump.bin<pid>,其中 pid 表示 Java 应用程序的进程 ID。执行该命令后,就会生成一个名为 heapdump.bin 的文件,该文件就是 JVM 的堆转储。MAT(Memory Analyzer Tool)是一个开源的 Java 内存分析工具,它可以快速定位内存问题,以及查看对象数量、类型、占用内存大小等信息,用于分析堆转储文件。使用 MAT 分析堆转储需要注意,需要在 64 位 JVM 上执行 MAT 工具,并分配足够的内存空间,同时分析大型堆转储文件可能需要耐心等待。总之,堆转储和 MAT 工具是 Java 应用程序开发和运维中不可或缺的工具,能够帮助开发者快速定位和解决内存问题。

5.4.5 分析 GC 日志

GC 日志是分析垃圾回收的关键。使用 Jstat 工具可以生成 GC 日志并进行分析,以便找出内存使用的问题。GC 日志包含垃圾回收器在运行过程中的各种信息,可以得出堆内存的使用情况及对应的垃圾回收器运行的效率等信息。Jstat 工具是 JDK 中自带的命令行工具,用于查看应用程序中的 JVM 统计信息,其中也包括 GC 相关信息。

在使用 Jstat 工具生成 GC 日志时,需要指定参数,例如-S、-T、-h、-gc 等。这些参数用于在每次垃圾回收后指定 Survivor 空间中对象的大小、堆中各个区域的总大小、堆中已分配和可用的内存大小及要输出的 GC 相关信息。

在分析 GC 日志时,需要根据不同的垃圾回收器进行分析。例如,对于 G1 垃圾回收器,需要关注其分区情况,以便优化垃圾回收器的运行效率。对于 CMS 垃圾回收器,则需要关注停止时间的数量和长度等信息,以便评估垃圾回收器的性能表现。

在进行 GC 日志分析时,还需要关注一些常见的问题类型,例如内存泄漏、内存浪费、Full GC 太多等。通过 GC 日志中的信息,可以寻找这些问题的原因和解决方案。例如,对于 Full GC 太多的问题,可以考虑对代码进行优化,减少对象的创建和销毁,从而减少 Full GC 的次数。

总之,分析 GC 日志是优化 Java 应用程序性能的重要一环,需要熟练掌握相关工具和技术,以便更好地进行问题排查和解决。

5.4.6 分析线程转储

通过 Jstack 工具,可以生成应用程序的线程转储,用于分析线程的状态、死锁等情况。

生成线程转储有两种方式,一种是在命令行中直接输入 Jstack 命令,另一种是在 JVM 的管理界面中进行操作。线程转储是指将当前 JVM 的所有线程在一个瞬间的状态保存到一个文件中。可以在这个文件中看到所有线程的状态信息,包括线程的 ID、状态、调用栈等。Jstack 的工作原理是通过向 JVM 发送 SIGQUIT 信号触发 JVM 进行线程转储操作,这个信号不会结束进程,但会导致 JVM 在指定的文件路径生成一个线程转储文件。

通过 Jstack 生成线程转储,可以分析出现的线程死锁、线程阻塞、内存泄漏等问题。工具的使用可以通过命令行方式、JVisualVM、EclipseMAT 等方式进行。当应用程序出现死锁或阻塞等问题时,可以使用 Jstack 生成线程转储并进行分析。解决问题的步骤需要通过 jps 命令查看 Java 进程的 PID,以生成线程转储文件,并使用工具打开线程转储文件,分析线程的调用栈和状态信息。最后,找到问题的根本原因并进行解决,可以采用调整代码、修改配置等方式解决问题。

总之,通过 Jstack 生成线程转储,可以帮助开发人员分析问题的根本原因,解决线程死锁、线程阻塞、内存泄漏等问题。需要结合工具进行分析,并且要有一定的分析思路。

5.4.7 进行代码审查

在应用程序中,代码也可能存在导致 JVM 问题的情况,因此,通过检查代码段可以找到一些潜在的问题所在,例如可能会发现一些可能死锁的同步块。为了找到可能导致问题的代码段,可以通过应用日志分析工具进行分析,并对其进行优化。此外,还可以使用调试工具来跟踪问题,并修复问题代码。

在代码分析过程中,需要对 JVM 工作原理有一定的了解,例如了解 JVM 在运行过程中内存的分配、对象的创建和销毁等细节有助于更好地分析代码中可能存在的问题。同时,还需要识别出可能引起问题的类型,如死锁、内存泄漏、线程安全问题等。在分析时还需考虑到不同情况的特殊性,例如可能存在多个线程同时访问同一资源的情况,可能存在不同的环境变量对程序性能的影响等,因此,需要针对不同情况采取不同的分析和解决方法。

对于不同的问题类型,通常可以使用一些工具来辅助分析和解决,例如性能分析工具可以定位应用程序的瓶颈,内存分析工具可以检测内存泄漏等,日志分析工具可以分析应用程序在运行时的信息等。在分析时应采用分层次、逐步深入的方法,逐渐收敛到可能导致问题的代码段,然后进行针对性优化。

5.4.8 实验和更改

为解决问题,可以考虑进行一些配置调整和代码调整,然而,这个过程可能需要进行多次实验和不断地更改,以找到最佳的解决方案。除了可以手动调整,还可以使用一些工具来分析和解决问题。例如,调试工具可以定位问题的根本原因,版本控制工具可以管理和控制源代码的变更,性能测试工具可以评估软件在不同负载情况下的表现。为了成功地解决问题,需要具备一定的技术能力和实践经验,需要建立清晰的分析思路和方法,并反复进行实验和验证。最终,才能找到最佳的解决方案。