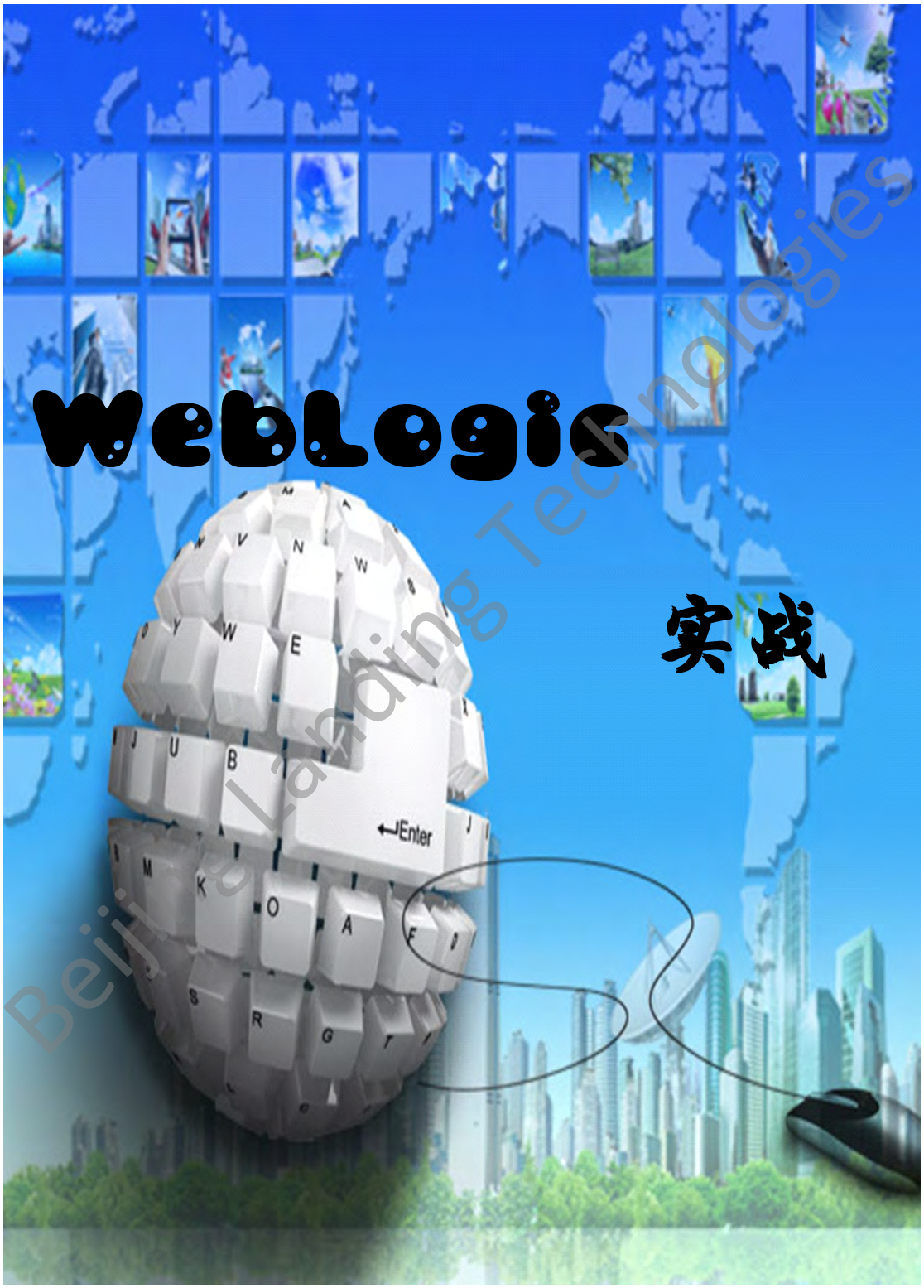
联动北方
LANDING

企业级 IT 运维宝典系列



联动北方 出品

目　录

第 12 章 常规服务器挂起故障

这一章，我们重点讨论跟 WebLogic 服务器挂起有关的常规故障，这个故障具有相当的普遍性和代表性，是应用服务器运行中最常见的问题之一。

12.1 服务器挂起概述

在展开相关内容之前，首先来看一下，服务器挂起的定义，即一般什么状况出现，可以判断为服务器挂起。

12.1.1 什么是服务器挂起

如果 WLS 实例不再响应用户请求，一般称该实例已挂起。

具体表现如：

- 向服务器发出的新请求未获接受或被忽略；
- 现有请求超时或一直没有返回响应。

12.1.2 服务器挂起分类

服务器挂起模式，具体可以分为如下几类：

1、线程占用导致服务器挂起模式；

2、垃圾回收导致服务器挂起模式；

3、代码优化中服务器挂起模式；

4、程序死锁服务器挂起模式；

5、JDBC 相关的服务器挂起模式；

6、EJB RMI 服务器挂起模式；

7、JSP 编译导致服务器挂起模式；

8、JSP 导致服务器挂起模式；

9、SUN JVM Bug 导致服务器挂起模式。

12.1.3 服务器挂起症状

当服务器挂起时，主要的故障症状一般包括：

1、请求未得到处理；

2、服务器表现为不执行任何操作；

3、如果服务器正接近挂起，它处理请求的时间会越来越长。

当挂起发生时，对服务器的影响有如下几种可能：

- A：可能会因挂起而崩溃，但这不是必然结果；
- B：可能会从挂起状态中恢复过来；

例如，如果挂起是由资源争用引起的，则当存在空闲资源时，服务器就可以恢复正常

- C：如果不手动执行某种操作，服务器可能会一直保持挂起状态。

12.2 常规服务器挂起成因

服务器挂起，总体来讲是个已经发生的结果，对于系统维护技术人员来说，更重要的是去寻找服务器挂起的原因，从而解除挂起并规避以后再出现类似情况的风险。

12.2.1 服务器挂起成因总述

1、服务器挂起通常由缺乏某种资源而引起，这种缺乏使服务器无法对请求进行处理：

- A：没有足够的线程或内存来执行处理；
- B：没有足够的文件句柄可供服务器用于管理打开的文件；
- C：资源争用或死锁也会导致挂起。

2、如果服务器似已经挂起，但又缓慢恢复正常，那么：

- A：资源空闲下来，处理继续执行；
- B：恢复过程可能漫长的让人无法接受。

3、有时候服务器看起来好像已经挂起了，但是可能是正在进行某个资源消耗比较大的操作，当操作完成以后，服务器可能就会恢复正常。

12.2.2 服务器挂起具体成因

1、配置的线程数不足：

所有线程都被占用了，没有线程可用于处理新工作；详细信息可以参考后续章节“线程资源相关服务器挂起模式”。

2、垃圾回收（GarbageCollection，GC）花费太多时间：

GC 会影响服务器的性能，因为在 GC 期间服务器处理会暂停；详细信息可以参考后续章节“垃圾回收导致服务器挂起模式”。

3、JVM 在代码优化期间挂起：

优化时可能会导致临时挂起，详细信息可以参考后续章节“代码优化中服务器挂起模式”。

4、应用程序死锁：

线程 A 锁定资源 1，然后等待锁定资源 2；线程 B 锁定资源 2，然后等待锁定资源 1；详细信息可以参考后续章节“应用程序死锁服务器挂起模式”。

5、JDBC 死锁：

发生数据库死锁时通常会导致挂起，详细信息可以参考后续章节“JDBC 中的服务器挂起模式”。

6、所有线程都在等待对远程 JVM 的 RMI 调用响应：

大量远程 JNDI 查找时也有可能导致线程挂起；详细信息可以参考后续章节“EJB RMI 服务器挂起模式”。

7、JSP 编译：

服务器在大量负载情况下可能会挂起；详细信息可以参考后续章节“JSP 编译导致服务器挂起模式”。

8、JSP 的 servlet 时间设置不当：

例如，PageCheckSeconds 的设置不当；详细信息可以参考后续章节“JSP 导致服务器挂起模式”。

9、SUN JVM 错误：

例如，轻量型线程库中有错误；详细信息可以参考后续章节“SUM JVM 错误导致服务器挂起模式”。

上面提到的这些是最常见的导致服务器挂起的因素。如果服务器挂起了，很可能是由于上面的这些因素之一造成的。另外，垃圾回收操作是一个密集处理操作，可能会造成服务器不响应。在手动进行垃圾回收以前，一定要确认是否确实需要进行垃圾回收操作。当您执行垃圾回收操作时，JVM 经常会检查堆中的每一个存活的对象。

12.3 服务器挂起探查

下面，我们来具体进行服务器挂起的一些探查。

12.3.1 基本探查步骤

1、从命令行对服务器执行 PING 命令：

执行如下命令：

```
java weblogic.Admin -url t3://localhost:7001 -username <user> -password <pass> PING;
```

示例 12-1

如果服务器正在挂起，首先使用上面提到的命令对 WebLogic 做 PING 操作。如果服务器可以响应 PING，则可能是应用程序挂起，而服务器并未挂起。

如果某个应用程序占用了所有的执行线程，正在执行需要长时间运行的代码，WebLogic 就会没有能力处理 PING 请求，直到有执行线程可用才可以。

2、检查服务器是否正在执行垃圾回收：

同样，要确认服务器确实挂起了，并且没有做垃圾回收操作。如果想要确认的话，加上-verbose:gc 参数，重启 WebLogic，并且将标准输出和标准错误输出重定向到一个文件里去，如果服务器停止了响应的话，可以根据输出信息来看它是否正在做垃圾回收操作，或者 WebLogic 是否真的挂起了。

12.3.2 查看执行线程运行状态

查看执行线程的运行情况，如果一个空闲线程都没有的话，那么很可能会导致服务器挂起，可以这样查看“default”队列里是否有空闲的执行线程：

登录控制台，然后依次：DomainName → Servers→ ServerName→Monitoring→ General→ Monitor All Active Queues→ weblogic.kernel.Default，里面的“Current Request”列会显示线程是否在进行工作。

如果一个空闲线程也没有的话，很可能就需要为应用程序配置更多数量的线程了。可以在管理控制台更改线程的数量，在更改完以后，WebLogic 会保存在 config.xml 文件里。

12.3.3 创建线程状态分布快照 Thread Dump

为了准确的把握服务器挂起时，我们需要借助于一个分析利器，即线程状态分布的快照 Thread Dump。

12.3.3.1 Thread Dump 简介

Thread Dump 是在特定时刻对 JVM（服务器）进程中所有活动线程的原样快照，因此运行 WebLogic 的 java 进程的 Thread Dump 里会记录 WebLogic 自己的执行线程以及应用程序创建的执行线程正在干什么（丢失的线程除外），然后以文本的形式提供给我们，对于探查许多类型的服务器挂起极具价值，另外对分析其它 WebLogic 故障也很有参考价值。

因为服务器的某些线程等待和挂起可能会导致服务器挂起，而 Thread Dump 里又会记录各个未丢失的线程的状态，所以我们可以借助 Thread Dump 对 WebLogic 挂起进行分析。

可以在运行着的服务器上进行，如果该服务器还能响应 PING（如果 WebLogic 无法响应 PING 做了，那么可能就不能对其做 Thread Dump 了），或在服务器崩溃（如果发生）的瞬间进行。

另外，在做 Thread Dump 时，最好接连做 3 次以上，每次间隔 5-10 秒钟，以辅助死锁的检测，也可以分析各个 Thread Dump 之间线程状态随时间发生的变化。

12.3.3.2 对运行着的服务器上进行 Thread Dump

1、针对 Unix/Linux 平台

执行：kill -3 〈WLS_pid〉，其中 WLS_pid 为 WLS 进程的进程号(PID)。

2、针对 Windows 平台

1）进入服务器的“命令提示符”窗口；

2）右键单击标题栏，然后选择“属性”；

3）在“布局”选项卡的“屏幕缓冲区大小”下，将“高度”设置为 2000 以上；

4）单击“确定”；

5）要将 Thread Dump 输出到命令窗口，按：Ctrl-Break；

6）在输出中向前回滚到到转储的起始处，即以下列词语开头的地方：“Full thread dump”。

3、针对各平台通用的命令

在 Windows/Unix/Linux 操作系统上都可以执行下面的命令，来做 Thread Dump：

```
java weblogic.Admin -url ManagedHost:Port -username weblogic -password weblogic THREAD_DUMP
```

示例 12-2

注：使用该命令时，需要 WebLogic 能够响应 PING 操作。

12.3.3.3 服务器发生故障时进行 Thread Dump

捕捉服务器发生故障前服务器线程的状态，可以采取下列措施：

1、服务器启动时启用下列选项：

```
Sun JVM:        -xx:+ShowMessageBoxOnError
```

```
JRockit JVM:  -Djrockit.waitonerror
```

示例 12-3

2、同时以前台方式启动服务器

3、之后如果 JVM 崩溃，会显示以下提示：

```
Do you want to dubug the problem ?
```

示例 12-4

4、回答此提示前，您就可以趁机捕捉 JVM 的 Thread Dump。

注：如果 WebLogic 是由于自己 trap 的错误而被强制关闭的，那么 WebLogic 就不会提示你进行 debug。这是因为 WebLogic 不是技术上的崩溃，而是由于致命错误而“优雅地”自行关闭了。

12.3.4 初始探查结果分析

如果不能对挂起的服务器做前边提到的操作，也就是说无法从挂起的服务器中获得任何信息的话，您可能需要重新启动一下 WebLogic 了，在重启时要加适当的参数，以确保在下次出现挂起时，可以获得某些数据，其中确保能做 Thread Dump 是在做分析时，最重要的一步，并且如果可能的话，对日志进行记录，以查看在出现挂起时，是否正在进行垃圾回收操作。

如果通过分析，发现在服务器挂起时 WebLogic 正在进行进行垃圾回收操作，那么您的垃圾回收模式或者设置可能是错误的，可能需要进行调整。比如在业务执行的关键时刻进行垃圾回收。由垃圾回收造成的服务器挂起模式可以参考后续章节“垃圾回收服务器挂起模式”，里面有详细的论述。

如果垃圾回收不是原因的话，下一步就要分析 Thread Dump，看一下每个线程都在做什么，然后根据每个线程当时的运行情况，再做进一步的分析。比如，如果在服务器挂起时，有很多的线程都在做 JSP 的编译工作，那么就有可能是 JSP 的编译造成的挂起，下一步就要对 JSP 编译做具体的分析了，此时可以参考后续章节“JSP 编译导致服务器挂起模式”。以此类推，如果是线程正在做其它工作，那么再针对具体的情况，再做进一步的分析。

12.4 线程资源相关服务器挂起模式

在前面探查的基础上，我们会发现，与线程资源相关的服务器挂起，是最普遍接触到的模式。

12.4.1 Socket Reader 线程引起的服务器挂起

如果服务器在挂起时，发现整个线程池中尚有空闲线程，那么 Socket Reader 线程数不足可能才是问题所在。

12.4.1.1 Socket Reader 线程简介

Socket Reader 线程实际上也是执行线程，只是专门拿出执行线程来做：接受来自监听线程队列的传入请求然后将该请求置于服务器的执行线程队列中的工作。分配执行线程作为 Socket Reader 线程能提高 WebLogic 接受客户端请求的速度和能力。

默认情况下，WebLogic 实例在启动时会创建 3 个 Socket Reader 线程，一个线程通常用于执行轮询功能，另外两个线程用于处理请求。具体线程数量是默认执行队列线程数的某个百分比。一般 Socket Reader 线程的数量应该比较小，但如果需要的话，就要增加该线程数：

集群系统需要的 Socket Reader 线程数可能要多于缺省的 Socket Reader 线程数，比如在业务高峰阶段使用了 3 个以上的 socket；

ThreadPoolPercentSocketReaders 属性控制 Socket Reader 线程的数量，它设置从执行线程中拿出来做 Socket Reader 线程的个数的最大百分比。这个数的最优数值与应用程序有关系。默认是 33，也就是拿出执行线程的 33%来做 Socket Readerr 线程，这个值的合法范围是 1-99。可以通过设置该属性，来增大 Socket Reader 线程的数量。

需要注意的是：专门用来做 Socket Reader 线程的数量和具体执行任务的执行线程之间数量，要达到一个平衡，否则的话 WebLogic 可能会出现故障。

另外，如果在 Thread Dump 里没有 Socket Reader 线程的话，那么可能 WebLogic 系统在某个地方存在 bug，以至于让 Socket Reader 线程消失了，这个时候一般需要官方的正式补丁。

12.4.1.2 Socket Reader Thread Dump 示例

Thread Dump 中的 Socket Reader 线程示例：

```
    "ExecuteThread: '2' for queue: 'weblogic.socket.Muxer'" daemon prio=10
tid=0x000 36128 nid=75 lwp_id=6888070 waiting for monitor entry
[0x1b12f000..0x1b12f530]
     at weblogic.socket.PosixSocketMuxer.processSockets
     (PosixSocketMuxer.java:92) - waiting to lock <0x25c01198> (a
java.lang.String)
     at weblogic.socket.SocketReaderRequest.execute
(SocketReaderRequest.java:32). . .
    "ExecuteThread: '1' for queue: 'weblogic.socket.Muxer'" daemon prio=10
tid=0x000 35fc8 nid=74 lwp_id=6888067 runnable [0x1b1b0000..0x1b1b0530] at
weblogic.socket.PosixSocketMuxer.poll(Native Method)
     at weblogic.socket.PosixSocketMuxer.processSockets
    (PosixSocketMuxer.java:99)
    - locked <0x25c01198> (a java.lang.String)
     at weblogic.socket.SocketReaderRequest.execute
    (SocketReaderRequest.java:32)
    . . .
    "ExecuteThread: '0' for queue: 'weblogic.socket.Muxer'" daemon prio=10
tid=0x000 35e68 nid=73 lwp_id=6888066 waiting for monitor entry
[0x1b231000..0x1b231530]
     at weblogic.socket.PosixSocketMuxer.processSockets
    (PosixSocketMuxer.java:92)
     - waiting to lock <0x25c01198> (a java.lang.String)
     at weblogic.socket.SocketReaderRequest.execute
    (SocketReaderRequest.java:32)
    . . .
```

示例 12-5

以上信息为 Thread Dump 中的 Socket Reader 线程示例：其中线程 1 在执行 poll 轮询功能，线程 0 和线程 2，在处理请求，调用 processSockets 方法。

12.4.1.3 检查监听线程

首先所有请求都是通过监听线程进入服务器的，如果缺少监听进程，则无法接收任何工作，监听线程应当伴随着 socketAccept 方法。

Thread Dump 中的监听线程示例：

```
    "ListenThread.Default" prio=10 tid=0x00037888 nid=93 lwp_id=6888343
runnable [0x 1a81b000..0x1a81b530]
    at java.net.PlainSocketImpl.socketAccept(Native Method)
    at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:353)
    - locked <0x26d9d490> (a java.net.PlainSocketImpl)
    at java.net.ServerSocket.implAccept(ServerSocket.java:439)
    at java.net.ServerSocket.accept(ServerSocket.java:410)
    at weblogic.socket.WeblogicServerSocket.accept
(WeblogicServerSocket.java:24)
    at weblogic.t3.srvr.ListenThread.accept(ListenThread.java:713)
    at weblogic.t3.srvr.ListenThread.run(ListenThread.java:290)
```

示例 12-6

所有请求都是经过监听器线程进入 WebLogic 系统的；如果监听线程没有了，就无法接收请求，这样也就不能做任何工作了。检查 Thread Dump 里确实有监听线程，而且监听线程应该在执行 socketAccept 方法。

12.4.2 无空闲线程时的服务器挂起

以上是跟监听线程相关的故障，比较少见，更多的普遍情况是线程池中的线程资源已经彻底用尽。

12.4.2.1 无空闲线程挂起症状分析

1. 如果通过分析，发现等待、死锁或其它资源限制未表现为一致的模式，但服务器挂起仍定期出现，如果此时检查 Thread Dump 信息，发现除了所有繁忙线程外，看不到其它的重复一致性，那么很可能是没有为应用程序分配足够多的线程，来做相应的工作。
2. 对于高占用率应用程序（如：web 应用程序），可能需要为其分配更多执行队列（和线程），专供它使用，对于这种情况的具体信息可以参考后续章节。
3. WebLogic Server 可能检测到卡滞线程，如果服务器检测到卡滞线程，会进入 CRITICAL 状态。及早检测到卡滞状态可以为及时探查（例如，在整个服务器挂起前进行 Thread Dump）创造条件。

12.4.2.2 进行相应调整

1. 针对上述症状的前两种情况，如果通过分析，发现确实是线程不够了，那么就需要增加执行线程的数量。
   - 增加 ThreadCount，以便服务器可以同时执行更多的操作；可以通过 config.xml 文件里执行队列元素的 ThreadCount 属性的值，来增大执行线程的数量。

注：执行线程的数量不是增大的越多越好，因为执行线程也会消耗系统资源，如果增大的执行线程数超过必须的线程数以后，可能反而会造成系统性能的下降；并且可能导致 Native 内存不足引起的 OutOfMemory 异常。

- 设置 ThreadPoolPercentSocketReaders 来定义必须的 SocketReader 线程数（%）；Socket Reader 线程及如何增大其数量已经有所介绍。

除此之外，这里再补充两点：

- 为了达到最好的 socket 性能，BEA 建议您使用本地 Socket Reader 实现，而不是使用纯 Java 实现。确认使用了单独的本地 Socket Reader 多路复用线程，即 NativeIOEnabled=true（缺省值）；
- 然而，假如您必须使用纯 JAVA Socker Reader 实现的话，您仍然可以提高 socket 通信的性能：为每个服务器实例和客户端 machine 配置适当数量的执行线程来当作 Socket Reader 线程使用。

2. 对于上述症状第三种情况的卡滞线程问题，可以设置 StuckThreadMaxTime 和 StuckThreadTimerInterval 来控制服务器检测卡滞线程的方式。
   - StuckThreadMaxTime 参数设置了线程必须持续工作多长时间以后，WebLogic 将其诊断为卡住线程，默认是 600 秒；
   - Stuck Thread Timer Interval 参数指定了 WebLogic 检测线程的时间间隔，也就是每隔多长时间以后，WebLogic 去扫描一下进程，看看它们是否持续工作了。

12.4.3 线程数足够时的服务器挂起

如果通过分析发现服务器挂起时，其线程数是足够的，那么可能会出现下面的这几种情况：

1. 连续的 Thread Dump 分析，可能显示已经达到资源极限；
   - 发生 OutOfMemory 异常
   - 发生 Too Many Open Files
   - 达到 JDBC 连接池限值
2. Thread Dump 也可能显示发生应用程序死锁；
3. 应用程序处理可能导致了挂起；
   - EJB RMI 调用
   - JSP 调用
4. 也可能存在其它情况。
   - 正在进行垃圾回收
   - 代码优化
   - SUN JVM 错误
   - JSP 编译

12.5 垃圾回收导致服务器挂起模式

在讨论这个模式之前，先来回顾一下 GC 的分类：

1. Scavenge GC：

一般情况下，当新对象生成，并且在 Eden 申请空间失败时，就会触发 Scavenge GC，对 Eden 区域进行 GC，清除非存活的对象，并且把尚存活的对象移动到 Survivor；整理 Survivor 的 From 及 To 区域。

2. Full GC：

顾名思义，这种 GC 是指整个 JVM 的一个完整的全部 GC。导致 Full GC 发生的可能原因有很多，比如：

- Old 区被写满；
- 某些情况下的 Survivor 满或在 Survivor 无法申请存储空间；
- 上一次 GC 之后 Heap 的各区域分配策略动态变化；
- Perm 域被写满；
- System.gc()函数在应用程序中被显式的调用。

12.5.1.2 什么是垃圾回收导致服务器挂起模式

Scavenge GC 在“New 区域”中完成，因此速度更快。Full GC 同时包含了“New”和“Old”的区域，因此速度比 Scavenges 要慢。

发生 Full GC 的后果是，Full GC 会对整个 Heap，包括 Eden、Survivor、Old 三个区域进行 GC，删除所有非存活对象，并且移动存活的对象。

如果仔细观察，未进行 Full GC 时，系统的 CPU 使用正常，但每次在 Full GC 期间，系统 CPU 都在高位，说明 CPU 高与 Full GC 垃圾回收有关。

当频繁的 Full GC,导致 CPU 使用过多，其他线程都等待，整体性能下降，就有可能导致服务器挂起，但这个 Full GC 有其一定的原因导致的。

12.5.1.3 垃圾回收导致服务器挂起模式的探查

1. JVM 一些参数配置的不合理：
   - New 区不宜太小，否则 Scavange GC 太频繁；但不宜太大，否则每次 GC 时间过长；
   - 设定合理的 SurvivorRatio，使得避免 Overflow 的出现；
   - Permanent 区设置太小不够用，要尽量避免；
   - 设定合理的 Heap 区的大小，如果只是追求过大，则 Full GC 时耗时过长。
2. 内存泄漏：

程序中存在许多对象占用内存不能被回收，特别是大对象，导致频繁 Full GC 垃圾回收，而每次垃圾回收后又不能清理这些对象而回收占用空间，则系统的响应时间则越长，当新对象多次申请空间时又不能满足需求，最终出现内存溢出而 WebLogic 挂起。

占用 CPU 高的进程主要是 Java 进程，即 WebLogic Server 运行进程，通过分析 JDK GC 日志，可以发现在 GC 垃圾回收占用系统资源严重，而 Full GC 垃圾回收又是整个垃圾回收的重点，而每次 Full GC 垃圾回收都是对那些在年轻代区域中不能被回收的对象进行回收。

12.5.1.4 垃圾回收导致服务器挂起解决方法

1. 合理的 JVM 配置
   - 尽量采用静态的内存使用策略(非 IBM 虚拟机)：

- -Xms = -Xmx
- 设定-Xmn
- -XX:MaxPermSize = -XX:PermSize

- 合理设定-XX:SurvivorRatio;
- 慎用 Parallel GC 策略;
- 慎用 -Xoptgc 参数;
- 使用 - XX:+DisableExplicitGC ， 禁止 system.gc()的显式调用;
- TenuringThreshold 值的最大化;
- 使用最新版本的 JDK

2. 修改程序

内存溢出问题其彻底解决办法，也只能修改程序，这是治本的办法；而调整相关参数只能起到缓解的作用，只能治标。

12.6 代码优化中服务器挂起模式

这个模式比较简单，顾名思义，当 JVM 在进行 Java 代码优化时，比如 JIT 编译时，会占用大量的 CPU 资源，有时对外表现为暂时没有响应。

一般情况下，等优化做完了服务器即能自动恢复正常，所以这种情况顺其自然就可以，没有特别刻意的处理。

12.7 应用程序死锁服务器挂起模式

另一种比较常见的情况是，应用代码编写失误，带来死锁，导致服务器挂起。

12.7.1 什么是应用程序死锁服务器挂起模式

由多线程带来的性能改善是以可靠性为代价的，主要是因为这样有可能产生线程死锁。线程死锁时，第一个线程等待第二个线程释放资源，而同时第二个线程又在等 待第一个线程释放资源。我们来想像这样一种情形：在人行道上两个人迎面相遇，为了给对方让道，两人同时向一侧迈出一步，双方无法通过，又同时向另一侧迈出 一步，这样还是无法通过。双方都以同样的迈步方式堵住了对方的去路。假设这种情况一直持续下去，这样就不难理解为何会发生死锁现象了。

12.7.2 服务器挂起初始探查

毕竟应用程序死锁服务器挂起属于常规服务器挂起的一种，所以在服务器挂起时，有必要先进行前面章节中介绍的初始探查步骤，在确定是应用程序死锁问题后，再做一步的分析。

常规服务器挂起模式中介绍了初始探查步骤，这些步骤包括：

- 确定线程是被占用还是空闲;
- 以较短间隔进行若干 Thread Dump;
- 确定线程正在执行的工作以及线程挂起的原因。

12.7.3 应用程序死锁服务器挂起探查

一般如果服务器挂起是由于应用程序死锁相关的问题引起时，在 Thread Dump 里就会记录相关的信息，以下 Thread Dump 分析就是一个死锁。

```
    "[ACTIVE] ExecuteThread: '153' for queue: 'weblogic.kernel.Default (self-
tuning)'" daemon prio=1 tid=0x00002aab370ea700 nid=0x3eec waiting for monitor
entry [0x0000000045378000..0x000000004537bea0]
    at weblogic.utils.classloaders.ChangeAwareClassLoader.loadClass
    (ChangeAwareClassLoader.java:35)
    - waiting to lock <0x00002aaabc69ddb0> (a
weblogic.utils.classloaders.ChangeAwareClassLoader)
```

示例 12-7

大部分线程都在等待这个锁，而这个锁的所有者如下：

```
    "[ACTIVE] ExecuteThread: '120' for queue: 'weblogic.kernel.Default (self-
tuning)'" daemon prio=1 tid=0x00002aab439c2f10 nid=0x2dee runnable
[0x00000000451f0000..0x00000000451f5e20]
    at java.lang.Throwable.fillInStackTrace(Native Method)
    - waiting to lock <0x00002aaab0873340> (a java.lang.ClassNotFoundException)
    at java.lang.Throwable.<init>(Throwable.java:218)
    at java.lang.Exception.<init>(Exception.java:59)
    at java.lang.ClassNotFoundException.<init>
    (ClassNotFoundException.java:65)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:200)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
    - locked <0x00002aaabadfc938> (a java.net.URLClassLoader)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:299)
    - locked <0x00002aaabbf1cd20> (a
weblogic.utils.classloaders.GenericClassLoader)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:251)
    at weblogic.utils.classloaders.GenericClassLoader.loadClass
    (GenericClassLoader.java:161)
    at weblogic.utils.classloaders.FilteringClassLoader.findClass
    (FilteringClassLoader.java:83)
    at weblogic.utils.classloaders.FilteringClassLoader.loadClass
    (FilteringClassLoader.java:68)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:299)
    - locked <0x00002aaabd5b2228> (a
weblogic.utils.classloaders.GenericClassLoader)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:299)
    - locked <0x00002aaabc69ddb0> (a
weblogic.utils.classloaders.ChangeAwareClassLoader)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:251)
    at weblogic.utils.classloaders.GenericClassLoader.loadClass
    (GenericClassLoader.java:161)
    at weblogic.utils.classloaders.ChangeAwareClassLoader.loadClass
    (ChangeAwareClassLoader.java:35)
```

```
    - locked <0x00002aaabc69ddb0> (a
weblogic.utils.classloaders.ChangeAwareClassLoader)
```

示例 12-8

这个锁的所有者有在等待另外一个锁，这个锁的所有者正是 ExecuteThread: 153，造成死锁。

12.7.4 应用程序死锁服务器挂起解决方法

解决死锁没有简单的方法，这是因为使线程产生这种问题是很具体的情况，而且往往是在高负载的情况下才会出现。大多数软件测试产生不了足够多的负载，所以不可能暴露所有的线程错误。

在每一种使用线程的语言中都存在线程死锁问题，由于使用 Java 进行线程编程比使用 C 容易，所以 Java 程序员中使用线程的人数更多，线程死锁也就更普遍。可以在 Java 代码中增加同步关键字的使用，这样可以减少死锁，但这样做也会影响性能。如果负载过重，数据库内部也有可能发生死锁。

如果程序使用了永久锁，比如锁文件，而且程序结束时没有解除锁状态，则其他进程可能无法使用这种类型的锁，既不能上锁，也不能解除锁。这会进一步导致系统不能正常工作；这时必须手动地解锁。