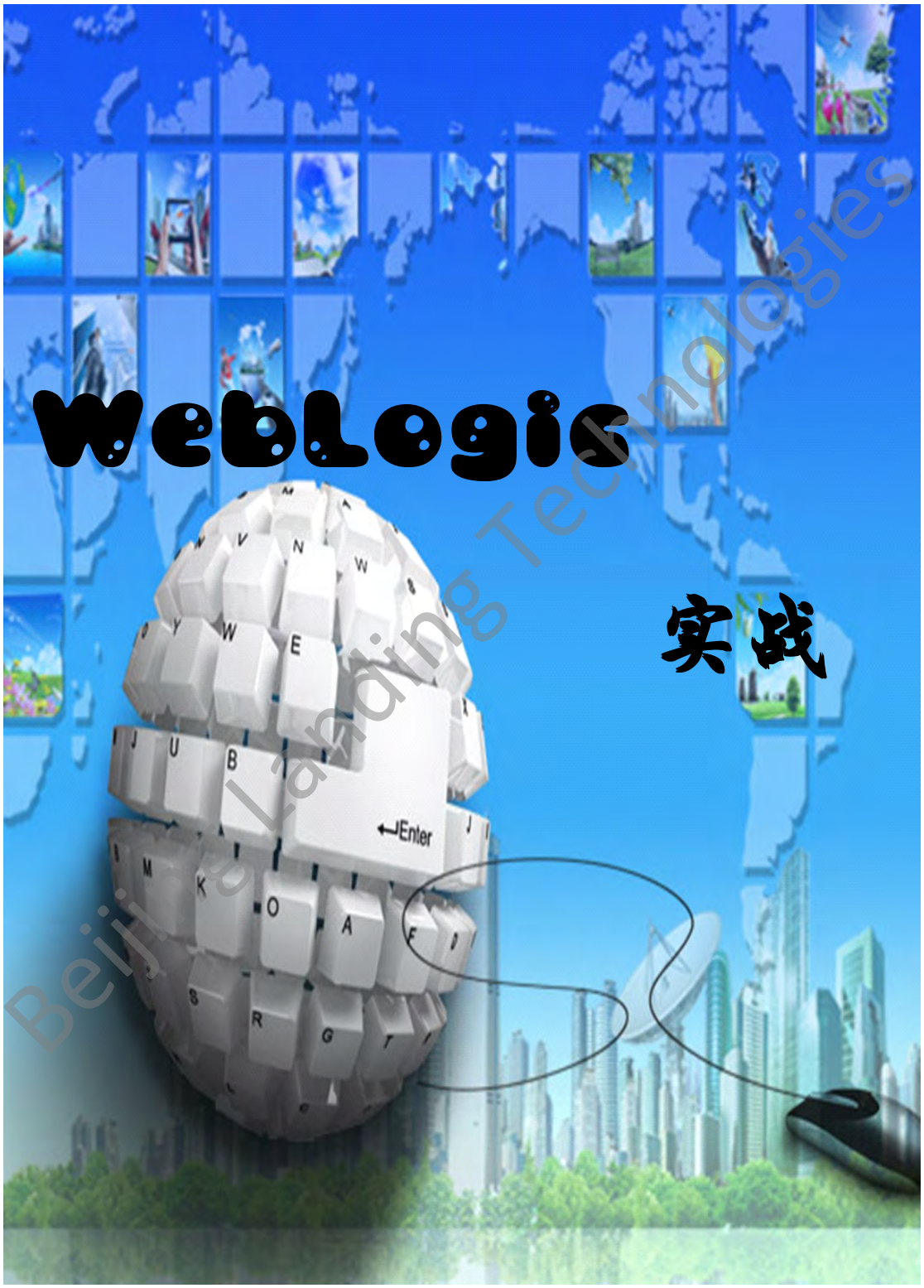
联动北方
LANDING

企业级 IT 运维宝典系列



联动北方 出品

目　录

第 11 章 如何发现问题

说明：本篇中 DOMAIN_NAME 是在其中定位域的目录的名称，而 ADMIN_SERVER_NAME 是管理服务器的名称。SERVER_NAME 为被管服务的名称。

11.1 WebLogic 监控

WebLogic 监控的目的，主要是发现系统中的隐患及系统运行是否稳定。主要从以下几方面进行检查：

11.1.1 操作系统

检查系统 cpu、内存和 I/O 等使用是否异常。如在负载不大的情况下，CPU 是否一直居高不下，内存占用和 I/O 是否一直很大，可以通过 topas、vmstat、ps 等命令查看。

11.1.2 网络

跟网络相关的检查有：

1. 位于一个 domain 中各个服务器是否能够联通；
2. WebLogic 服务器与数据库服务器的链接是否畅通；
3. 集群环境的 Multicast 广播通信是否正常。

11.1.3 WebLogic

对于跟 WebLogic 本身相关的检查有：

1. 检查是否对 JVM 进行了优化：

   比如最大堆内存、最小堆内存，以及 GC 算法是否合理。

2. 检查 GC 是否正常：

   主要是通过 WebLogic 控制台，查看 JVM 的空闲内存的变化情况，每次 GC 的回收情况。特别是可以在控制台强制垃圾回收，看看回收的内存是否太小。如果回收的内存太小，说明可能存在内存溢出的隐患。

   还可以在服务的启动脚本中添加 GC 日志，从 GC 日志中查看 GC 的回收情况是否正常。

3. 检查线程数：

   通过 WebLogic 控制台可以查看线程数的统计信息。WebLogic 9 以上的版本的线程是自优化的。但应该查看系统的线程最大数是否过大，如果超过 100 了，就要注意系统为什么会有这么大的压力。

4. 线程是否有 stuck 状态的。

   线程 stuck 状态说明存在超时的线程，也有可能存在线程的死锁。在查看线程数的时候，查看 stuck 线程统计一栏是否大于 0。如果 stuck 的线程一直大于 0，就要查看系统的运行状态、JVM 进程的状态和内存使用率是否正常等。如果有 thread dump 文件，或 java core 文件，就可以分析服务器线程之间是否存在死锁，以及哪些线程处于 stuck 状态。

   说明：产生 thread dump，可以通过命令 kill -3 <wls_pid>，后面章节会进行详细介绍这方面的知识。

5. JDBC 连接池

检查连接池中等待连接的数目是否过大，可以做适当调整。

如果用户访问系统变慢，且连接池基本占满，但是 WebLogic 的线程数量很少，就要怀疑应用是否没有释放数据库连接。

6. 系统日志

通过系统的日志分析 WebLogic 服务器及应用程序出现的错误，找到可能影响系统性能的服务器和应用的地方。

举例如下：

- 已经对用户进行响应

```
    java.lang.IllegalStateException: Cannot forward a response that is already
committed
    at weblogic.servlet.internal.RequestDispatcherImpl.forward
    (RequestDispatcherImpl.java:110)
    at com.landingbj.hebmc.ExceptionFilter.doFilter(ExceptionFilter.java:55)
```

示例 11-1

如上异常，为什么对用户的 response 已经提交给用户了，就要怀疑代码中某些地方在过滤器过滤之前已经显示的提交或关闭了对用户的响应的 outputstream。有些时候在过滤器中要做后续的一些处理，这时候 filter 中出现异常，有可能导致数据库连接不释放及一些后续的处理得不到执行。

- 线程一直处于 stuck 状态

```
    <[STUCK] ExecuteThread: '32' for queue: 'weblogic.kernel.Default (self-
tuning)' has been busy for "632" seconds working on the request "Http Request:
/stat/test/TestCtrl.statOracleTest.do", which is more than the configured time
(StuckThreadMaxTime) of "600" seconds.
```

示例 11-2

这句表名线程 32 执行时间超过 600 秒设定的时间。检查这个请求是否会导致线程执行超时，是否是长时间的处理。当前系统状态是否正常，是否是 cpu、内存等资源不足导致的线程执行缓慢。

重要的一点要查看线程 32 是否变成 unstuck 状态，否则就有可能出现线程死锁。

```
    <[STUCK] ExecuteThread: '32' for queue: 'weblogic.kernel.Default (self-
tuning)' has become "unstuck".>
```

示例 11-3

- 数据库连接问题

```
    java.sql.SQLException: [BEA][Oracle JDBC Driver]Error establishing socket
to host and port: db_landingbj:1521. Reason: Connection refused
```

示例 11-4

当出现此问题时，一般系统都执行缓慢。检查是否 ping 通数据库，是否有权限，也可以请数据库工程师查看数据库是否运行正常。检查 DNS 服务器或本地 DNS 映射是否正常，域名 db_landingbj 能否解析。

检查 WebLogic 记录的 Error 级别的日志，找到可能影响系统运行及功能的问题。

举例如下：

- URL 不存在

```
    ####<Feb 28, 2010 4:26:20 PM GMT+08:00> <Error> <HTTP> <yanzheng2> <wls6>
<ExecuteThread: '1' for queue: 'weblogic.socket.Muxer'> <<WLS Kernel>> <> <>
<1267345580932> <BEA-101215> <Malformed Request "/service/js/landingbj/mareal
</option><option value=". Request parsing failed, Code: -1>
```

示例 11-5

此出，说明线程 1 执行请求“/service/js/mareal/mareal </option><option value=”的时候，WebLogic 对这个请求解析失败.该请求的 URL 明显不很正常，可以请工程师修改对应的应用代码，以免影响应用的功能。

- class 文件找不到

```
    java.lang.ClassNotFoundException:
com.qtong.hebmc.controller.js.landingbj.mareal.jobile.com.service.LoginCtrl
    at java.lang.Class.forNameImpl(Native Method)
    at java.lang.Class.forName(Class.java:130)
    at com.landingbj.sterna.Controller.parseController(Controller.java:67)
    ... 20 more
```

示例 11-6

检查控制器是否书写正确，还是这个 java 文件没有正确编译部署。

- 邮件服务器异常

```
    org.apache.commons.mail.EmailException: Sending the email to the following
server failed : 218.207.67.75:25
    Caused by:
    javax.mail.MessagingException: Could not connect to SMTP host:
218.207.67.75, port: 25;
    nested exception is:
    java.net.SocketException: Connection timed out:could be due to invalid
address
```

示例 11-7

检查邮件服务器是否有问题，IP 是否冲突，网络连接是否正常，以免影响应用的功能。

- 是否是算法的 bug 导致的 OOM

```
    java.lang.OutOfMemoryError: Initializing Writer
    at com.sun.imageio.plugins.jpeg.JPEGImageWriter.initJPEGImageWriter
    (Native Method)
    at com.sun.imageio.plugins.jpeg.JPEGImageWriter.<init>
    (JPEGImageWriter.java:206)
    at com.sun.imageio.plugins.jpeg.JPEGImageWriterSpi.createWriterInstance
    (JPEGImageWriterSpi.java:130)
    at javax.imageio.spi.ImageWriterSpi.createWriterInstance
    (ImageWriterSpi.java:358)
    at javax.imageio.ImageIO$ImageWriterIterator.next(ImageIO.java:851)
    at javax.imageio.ImageIO$ImageWriterIterator.next(ImageIO.java:835)
    at javax.imageio.ImageIO.write(ImageIO.java:1473)
```

```
at javax.imageio.ImageIO.write(ImageIO.java:1554)
at com.landingbj.hebmc.controller.LoginCtrl.smsRandomPic
(LoginCtrl.java:267)
... 21 more
```

示例 11-8

应该查看 com.sun.imageio.plugins.jpeg.JPEGImageWriter 是否存在 bug，处理大图片时，是否会导致 OOM。

- 字符越界异常

```
java.lang.StringIndexOutOfBoundsException
at java.lang.String.substring(String.java:1088)
at com.landingbj.hebmc.controller.nethall.ChongZhiKaChaXunCtrl.transact
(ChongZhiKaChaXunCtrl.java:25)
```

示例 11-9

11.2 日志文件的获取

WebLogic 主要的日志文件有如下几种：

11.2.1 server_name.log

该日志记录的是服务（包括 admin server 和 managed server）启动过程中和关闭过程中的日志，还包括部署在服务上面的应用运行过程中产生的日志。

Server_name.log 的路径为：

```
DOMAIN_NAME/servers/SERVER_NAME/logs/server_name.log
```

示例 11-10

11.2.2 access.log

该文件具体记录在某个时间，某个 IP 地址的客户端访问了服务器上的那个文件。

access.log 文件的路径为：

```
DOMAIN_NAME/servers/SERVER_NAME/logs/access.log
```

示例 11-11

11.2.3 垃圾回收 log

记录服务的内存垃圾回收情况。该日志默认情况下并不生成，如果需要生成垃圾回收日志，需要在启动服务的脚本中指定垃圾回收日志文件的路径和名称。

也可以在域目录下的 setDomainEnv.sh 或者 wls_home 下的 commEnv.sh 文件中指定。

11.2.4 domain_name.log

记录一个 DOMIAN 的运行情况，一个 DOMAIN 中的各个 WebLogic SERVER 可以把它们的一些运行信息（比如：很严重的错误）发送给一个 DOMAIN 的 ADMINISTRATOR SERVER 上，ADMINISTRATOR SERVER 把这些信息些到 DOMAIN 日志中。

域日志文件的默认名称和位置是：

```
DOMAIN_NAME/servers/ADMIN_SERVER_NAME/logs/DOMAIN_NAME.log.
```

示例 11-12

11.2.5 jms.messages.log

JMS 服务器日志文件包含有关基本消息生命周期事件的信息，例如消息生成、使用和删除。如果承载主题消息的 JMS 目标配置为启用消息日志记录，则每个基本消息生命周期事件都会在 JMS 消息日志文件中生成一个消息日志事件。

消息日志位于服务器实例根目录下的 logs 目录中，即：

```
DOMAIN_NAME/servers/SERVER_NAME/logs/jmsServers/SERVER_NAMEJMSServer/jms.messages.log;
```

示例 11-12

11.3 启动脚本与配置参数文件的获取

WebLogic 的 admin server 的启动脚本为 DOMAIN_NAME/bin/目录下的 startWebLogic.sh，managed server 的启动脚本为 DOMAIN_NAME/bin/ startManagedWebLogic.sh。

WebLogic 的参数配置文件有两个，一个是域下的参数配置文件 DOMAIN_NAME/bin/ setDomainEnv.sh，另外一个是 WLS_HOME/common/bin/ commEnv.sh。这两个文件的区别是：commEnv.sh 的作用范围是整个 weblogic 下的所有域，setDomainEnv.sh 的作用范围仅仅是所在的 domain，如果两个文件中定义的参数有重复，则启动脚本以 setDomainEnv.sh 中的参数为准。

WebLogic 自带的启动脚本不但目录比较深，启动被管服务的时候，还得在启动脚本后面加参数，而且还不能在后台运行。

一般在生产环境中，维护人员会重新写一个 WebLogic 的启动脚本，放在一个容易找到的目录中。

下面是一个启动 managed server 脚本的例子：

```
 #!/bin/sh
USER_MEM_ARGS="-Xms4096m -Xmx4096m"
export USER_MEM_ARGS
#====================
nohup /home/weblogic/bea/user_projects/domains/landingbj_8001/bin
/startManagedWebLogic.sh App161_8001 http://127.0.0.1:7001 >>
/home/weblogic/bea/user_projects/domains/xxx2100_8001/bin/App161_8001.out &
```

示例 11-13

11.4 Thread dump 的获取和分析

11.4.1 Thread dump 是什么

每启动一个 WebLogic 实例，在系统中就会相应产生一个 java 进程。Thread dump 就是把 java 进程中所有线程的运行状况输出到指定的文件中。

11.4.2 如何获取 thread dump

Thread dump 的获取，在 Windows 下和类 unix 系统上方法并不相同。

在 Windows 下，点击激活运行 WebLogic 服务的命令行窗口，然后按 ctl+break 按钮，即可在屏幕输出中获取 thread dump。

在类 UNIX 系统中，通常 Solaris 系统执行“kill -quit wls 进程号”，其他执行“kill -3 wls 进程号” 即可生成 thread dump。

Thread dump 生成后，还会输出到启动 WebLogic 服务的时候指定的重定向文件中。

比如下面的启动脚本：

```
nohup /home/weblogic/bea/user_projects/domains/landingbj_8001/bin/
startManagedWebLogic.sh App161_8001 http://127.0.0.1:7001 >>
 /home/weblogic/log/App161_8001.out &
```

示例 11-14

Thread dump 生成后就可以在“/home/weblogic/log/App161_8001.out”文件中获取。

11.4.3 Thread dump 分析示例

Thread dumps 出来结果中包含各个线程的线程的运行状态、标识和调用的堆栈；调用的堆栈包含完整的类名，所执行的方法，如果可能的话还有源代码的行数。

这些信息可以帮助我们分析 WebLogic 的运行状况。

11.4.3.1 线程的主要状态

WebLogic 中的线程主要状态有三种，当前可以运行的线程、空闲线程和等锁的线程。

以下举例说明：

11.4.3.1.1WebLogic 中的空闲线程（线程主动 wait）

```
"[ACTIVE] ExecuteThread: '30' for queue: 'weblogic.kernel.Default (self-
tuning)'" daemon prio=1 tid=0x0000000040ae9d50 nid=0x6ccf in Object.wait()
[0x00007f6e6d48f000..0x00007f6e6d48fd90]
      at java.lang.Object.wait(Native Method)
      - waiting on <0x00007f6eab8764d8> (a weblogic.work.ExecuteThread)
      at java.lang.Object.wait(Object.java:474)
      at weblogic.work.ExecuteThread.waitForRequest(ExecuteThread.java:165)
      - locked <0x00007f6eab8764d8> (a weblogic.work.ExecuteThread)
      at weblogic.work.ExecuteThread.run(ExecuteThread.java:186)
```

示例 11-15

该线程是一个空闲的线程，“weblogic.work.ExecuteThread.waitForRequest”表示该线程在等待请求。

11.4.3.1.2正在执行的线程

```
"[ACTIVE] ExecuteThread: '219' for queue: 'weblogic.kernel.Default (self-
tuning)'" daemon prio=1 tid=0x00007f6e50345a50 nid=0x6e16 runnable
[0x00007f6e5d668000..0x00007f6e5d66bb10]
      at java.net.SocketInputStream.socketRead0(Native Method)
      at java.net.SocketInputStream.read(SocketInputStream.java:129)
      at java.io.BufferedInputStream.fill(BufferedInputStream.java:218)
      at java.io.BufferedInputStream.read(BufferedInputStream.java:235)
      - locked <0x00007f6f40cc0220> (a java.io.BufferedInputStream)
      at weblogic.net.http.MessageHeader.isHTTP(MessageHeader.java:220)
   ..........................................................................
```

示例 11-16

“runnable”表示该线程正在运行。

从“at java.net.SocketInputStream.socketRead0 (Native Method)” 以下表示该线程调用的堆栈。

11.4.3.1.3等锁的线程

```
    "[ACTIVE] ExecuteThread: '215' for queue: 'weblogic.kernel.Default (self-
tuning)'" daemon prio=1 tid=0x00007f6e50bd84d0 nid=0x6e12 waiting for monitor
entry [0x00007f6e5d72d000..0x00007f6e5d72fd10]
       at weblogic.utils.classloaders.ChangeAwareClassLoader.loadClass
(ChangeAwareClassLoader.java:35)
       - waiting to lock <0x00007f6eacbca018>
      (a weblogic.utils.classloaders.ChangeAwareClassLoader)
       at javax.xml.parsers.FactoryFinder.newInstance(FactoryFinder.java:88)
       at javax.xml.parsers.FactoryFinder.findJarServiceProvider
(FactoryFinder.java:278)
       at javax.xml.parsers.FactoryFinder.find(FactoryFinder.java:185)
       at javax.xml.parsers.DocumentBuilderFactory.newInstance
(DocumentBuilderFactory.java:98)
    ..........................................................................
```

示例 11-17

该线程 waiting for monitor entry，在等待某一个锁：waiting to lock <0x00007f6eacbca018>，该锁被其他线程释放后，线程会重新 runnable。

在实际的 thread dump 分析中，我们主要关注的是正在运行的线程和等锁的线程。

11.4.4 实际环境中 thread dump 分析示例

以下是生产环境中的一个例子，该线程执行超时。从堆栈信息“net/SocketNativeIO”和“com/informix/jdbc/IfxSqli.executeStatementQuery”看，该线程在通过网络获取数据库的数据。

执行数据库查询的代码段应该为：

```
    “com/xxxxx/prpall/service/spring/PrpWorkbenchMainServiceSpringImpl.getPrpC
FlowInfos”
    [STUCK] ExecuteThread: '136' for queue: 'weblogic.kernel.Default (self-
tuning)'" id=150 idx=0x2b0 tid=2639 prio=1 alive, in native, daemon
    at jrockit/net/SocketNativeIO.readBytesPinned
    (Ljava/io/FileDescriptor;[BIII)I(Native Method)
    at jrockit/net/SocketNativeIO.socketRead(SocketNativeIO.java:46)
    at java/net/SocketInputStream.socketRead0
    (Ljava/io/FileDescriptor;[BIII)I(SocketInputStream.java)
    at java/net/SocketInputStream.read(SocketInputStream.java:129)
    at java/io/BufferedInputStream.fill(BufferedInputStream.java:218)
    at java/io/BufferedInputStream.read1(BufferedInputStream.java:256)
    at java/io/BufferedInputStream.read(BufferedInputStream.java:313)
    ^-- Holding lock: java/io/BufferedInputStream@0x7fe572257740[thin lock]
    at com/informix/asf/IfxDataInputStream.readFully
    (IfxDataInputStream.java:146)
```

```
at com/informix/asf/IfxDataInputStream.readSmallInt
(IfxDataInputStream.java:453)
at com/informix/jdbc/IfxSqli.receiveMessage(IfxSqli.java:2495)
at com/informix/jdbc/IfxSqli.a(IfxSqli.java:1752)
at com/informix/jdbc/IfxSqli.executeStatementQuery(IfxSqli.java:1704)
at com/informix/jdbc/IfxSqli.executeStatementQuery(IfxSqli.java:1635)
at com/informix/jdbc/IfxResultSet.a(IfxResultSet.java:206)
at com/informix/jdbc/IfxStatement.executeQueryImpl
(IfxStatement.java:1229)
at com/informix/jdbc/IfxPreparedStatement.executeQuery
(IfxPreparedStatement.java:376)
at weblogic/jdbc/wrapper/PreparedStatement.executeQuery
(PreparedStatement.java:100)
at org/hibernate/jdbc/AbstractBatcher.getResultSet(AbstractBatcher.java:208)
at org/hibernate/loader/Loader.getResultSet(Loader.java:1812)
at org/hibernate/loader/Loader.doQuery(Loader.java:697)
at org/hibernate/loader/Loader.doQueryAndInitializeNonLazyCollections
(Loader.java:259)
at org/hibernate/loader/Loader.doList(Loader.java:2232)
at org/hibernate/loader/Loader.listIgnoreQueryCache(Loader.java:2129)
at org/hibernate/loader/Loader.list(Loader.java:2124)
at org/hibernate/loader/hql/QueryLoader.list(QueryLoader.java:401)
at org/hibernate/hql/ast/QueryTranslatorImpl.list
(QueryTranslatorImpl.java:363)
at org/hibernate/engine/query/HQLQueryPlan.performList
(HQLQueryPlan.java:196)
at org/hibernate/impl/SessionImpl.list(SessionImpl.java:1149)
at org/hibernate/impl/QueryImpl.list(QueryImpl.java:102)
at ins/framework/dao/EntityDaoHibernate$5.doInHibernate
(EntityDaoHibernate.java:506)
   ..........................................................
at weblogic/security/acl/internal/AuthenticatedSubject.doAs
(AuthenticatedSubject.java:321)
at weblogic/security/service/SecurityManager.runAs
(SecurityManager.java:121)
at weblogic/servlet/internal/WebAppServletContext.securedExecute
(WebAppServletContext.java:2010)
at weblogic/servlet/internal/WebAppServletContext.execute
(WebAppServletContext.java:1916)
at weblogic/servlet/internal/ServletRequestImpl.run
(ServletRequestImpl.java:1366)
at weblogic/work/ExecuteThread.execute(ExecuteThread.java:209)
at weblogic/work/ExecuteThread.run(ExecuteThread.java:181)
at jrockit/vm/RNI.c2java(JJJJJ)V(Native Method)
```

示例 11-18

11.5 Heap dump 的获取分析

11.5.1 什么是 heap dump

Java heap 是所有类实例和数组对象分配的一个运行时数据区，其间所有 Java VM 线程在执行期间共享 heap 中的数据。

那么一个 Java heap dump 相当于在一个特殊的时间点上生成的一个快照，它就像给一个繁忙的数据仓库在给定的时间上来了一个照片，我们通过这张快照可以识别哪些组件在那快照的那时间点上是可用的。

11.5.2 如何产生 heap dump

当 JVM 中对象过多，Java 堆内存（Heap Memory）耗尽时，就可能会触发产生 heap dump 文件。另外，可以使用工具或命令显式地产生该文件。使用工具如，IBM HeapAnalyzer，Sap Memory Analyzer 以及 Eclipse Memory Analyzer 都可以在指定状态产生 heap dump 文件。

11.5.3 什么是 Jps 和 Jmap

jps 用于 列出所有 java 相关线程的 pid 等信息，如：

```
[root@myjrjapp-100 ~]# jps
23178 Jps
20289 Bootstrap
```

示例 11-19

其中，“20289 Bootstrap”是指系统中运行的 tomcat 进程号和进程名。

jmap 是一个可以输出所有内存中对象的工具，它可以打印出某个 Java 进程（使用 pid ）内存中所有“对象”的情况（如：产生那些对象，及其数量）

比如，打印内存使用的摘要信息：

```
jmap pid
```

示例 11-20

jmap 甚至可以将 JVM 中的 heap，以二进制输出成文本，比如：

```
jmap -dump:format=b,file=f1 3024
```

示例 11-21

可以将 3024 进程的内存 heap 输出出来到 f1 文件里。

11.5.4 Jmap 示例

1. jmap -heap pid 查看 java 堆（heap）使用情况

```
using thread-local object allocation.
Parallel GC with 4 thread(s)  //GC 方式

Heap Configuration:      //堆内存初始化配置
MinHeapFreeRatio=40      //对应 JVM 堆最小空闲比率(default 40)
MaxHeapFreeRatio=70      //对应 JVM 堆最大空闲比率(default 70)
MaxHeapSize=512.0MB      //对应 JVM 堆的最大大小
NewSize  = 1.0MB         //对应 JVM 堆的‘新生代’的默认大小
```

```
MaxNewSize =4095MB       //对应 JVM 堆的‘新生代’的最大大小
OldSize  = 4.0MB         //对应 JVM 堆‘老生代’的大小
NewRatio  = 8            //对应 ‘新生代’和‘老生代’的大小比率
SurvivorRatio = 8        //对应年轻代中 Eden 区与 Survivor 区的大小比值
PermSize= 16.0MB         //对应 JVM 堆的‘永生代’的初始大小
MaxPermSize=64.0MB       //对应 JVM 堆的‘永生代’的最大大小

Heap Usage:                          //堆内存分步
PS Young Generation
Eden Space:                          //Eden 区内存分布
    capacity = 20381696 (19.4375MB)             //Eden 区总容量
    used     = 20370032 (19.426376342773438MB)  //Eden 区已使用
    free     = 11664 (0.0111236572265625MB)     //Eden 区剩余容量
    99.94277218147106% used                     //Eden 区使用比率
From Space:                          //其中一个 Survivor 区的内存分布
    capacity = 8519680 (8.125MB)
    used     = 32768 (0.03125MB)
    free     = 8486912 (8.09375MB)
    0.3846153846153846% used
To Space:                            //另一个 Survivor 区的内存分布
    capacity = 9306112 (8.875MB)
    used     = 0 (0.0MB)
    free     = 9306112 (8.875MB)
    0.0% used
PS Old Generation                    //当前的 Old 区内存分布
    capacity = 366280704 (349.3125MB)
    used     = 322179848 (307.25464630126953MB)
    free     = 44100856 (42.05785369873047MB)
    87.95982001825573% used
PS Perm Generation                   //当前的 “永生代” 内存分布
    capacity = 32243712 (30.75MB)
    used     = 28918584 (27.57891082763672MB)
    free     = 3325128 (3.1710891723632812MB)
    89.68751488662348% used
```

示例 11-22

2. jmap -histo pid 查看堆内存(histogram)中的对象数量，大小

```
num     #instances     #bytes    class name
序号      实例个数       字节数     类名
        ----------------------------------------------
1:       3174877      107858256  [C
2:       3171499       76115976  java.lang.String
3:       1397884       38122240  [B
4:        214690       37785440  com.tongli.book.form.Book
5:        107345       18892720  com.tongli.book.form.Book
6:         65645       13953440  [Ljava.lang.Object;
7:         59627        7648416  <constMethodKlass>
8:        291852        7004448  java.util.HashMap$Entry
9:        107349        6871176  [[B
```

```
total     9150732      353969416
```

示例 11-23

3. jmap - dump pid

将内存使用的详细情况输出到文件：

```
map -dump:format=b,file= m.dat pid
```

示例 11-24

11.5.5 Jmap 的分析方法

1. 分析方法 1：

将两个命令可以结合起来用，例如：

```
[root@myjrjapp-100 ~]# jps
23178 Jps
20289 Bootstrap

[root@myjrjapp-100 ~]# jmap 20289
Attaching to process ID 20289, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 10.0-b19

using thread-local object allocation.
Parallel GC with 8 thread(s)

Heap Configuration:
   MinHeapFreeRatio = 40
   MaxHeapFreeRatio = 70
   MaxHeapSize      = 1073741824 (1024.0MB)
   NewSize          = 1048576 (1.0MB)
   MaxNewSize       = 4294901760 (4095.9375MB)
   OldSize          = 4194304 (4.0MB)
   NewRatio         = 8
   SurvivorRatio    = 8
   PermSize         = 134217728 (128.0MB)
   MaxPermSize      = 268435456 (256.0MB)

Heap Usage:
PS Young Generation
Eden Space:
   capacity = 118358016 (112.875MB)
   used     = 38070328 (36.30669403076172MB)
   free     = 80287688 (76.56830596923828MB)
   32.165398919833194% used
From Space:
   capacity = 458752 (0.4375MB)
   used     = 155664 (0.1484527587890625MB)
   free     = 303088 (0.2890472412109375MB)
```

```
    33.932059151785715% used
From Space:
    capacity = 458752 (0.4375MB)
    used     = 0 (0.0MB)
    free     = 458752 (0.4375MB)
    0.0% used
To Space:
    capacity = 458752 (0.4375MB)
    used     = 0 (0.0MB)
    free     = 458752 (0.4375MB)
    0.0% used
PS Old Generation
    capacity = 954466304 (910.25MB)
    used     = 72784624 (69.41282653808594MB)
    free     = 881681680 (840.8371734619141MB)
    7.625688166776813% used
PS Perm Generation
    capacity = 134217728 (128.0MB)
    used     = 38192248 (36.42296600341797MB)
    free     = 96025480 (91.57703399658203MB)
    28.455442190170288% used
[root@myjrjapp-100 ~]#
```

示例 11-25

2. 分析方法 2：

使用 jmap 命令 dump 内存出来 ：

```
jmap -dump:live,format=b,file=heap.bin 8023
```

示例 11-26

之后会在当前目录创建一个”heap.bin”文件，大小可能会有好几百 M 甚至几 G。可以把此文件进行压缩，然后再传到其他 Windows 机器中进行结果分析。

```
[root@openAS-main ~]# gzip heap.bin
```

示例 11-27

分析：在测试机上安装一个分析工具，比如前面提到的 Eclipse MAT 等等。

11.6 关于 Java dump 的一些常见问题

1. 为什么发生 JVM Crash 时，JVM 没有自动生成 Java dump 文件？

这种情况大多与系统的环境变量或者 JVM 启动参数的设置有关，比如设置了 DISABLE_JAVADUMP=true，IBM_NOSIGHANDLER=true 等等，因此可以首先检查系统设置和 JVM 启动参数。当然也不排除因为一些不确定因素导致 JVM 无法产生 Java dump，虽然这种可能性比较小。

2. Java Dump 中的很多线程处于 state:CW 和 state:B 状态，它们之间有何区别？

两者都处于等待状态，不同是：

- CW - Condition Wait：条件等待。这种等待一般是线程主动等待或者正在进行某种 I/O 操作，而并非等待其它线程释放资源。比如 sleep()，wait()，join()等方法的调用。
- B - Blocked：线程被阻塞。与条件等待不同，线程被阻塞一般不是线程主动进行的，而是由于当前线程需要的资源正在被其他线程占用，因此不得不等待资源释放以后才能继续执行，例如 synchronized 代码块。

3. 为什么在 PsList 里看到的线程无法映射到 Java dump 中？

由于很多操作系统工具和命令输出的线程的 TID 都是十进制的，映射 Java dump 时首先要将其转换为十六进制数字，然后再到 Java dump 中查找对应的 native ID。Java dump 中每个线程都有两个 ID，一个是 java 线程的 TID，另一个是对应操作系统线程的 native ID。