

深入分析

Java Web

技术内幕 (修订版)

|| 许令波 著 ||



内 容 简 介

本书围绕 Java Web 相关技术从三方面全面、深入地进行了阐述。首先介绍前端知识，主要介绍在 Java Web 开发中涉及的一些基本知识，包括 Web 请求过程、HTTP、DNS 技术和 CDN 技术。其次深入介绍了 Java 技术，包括 I/O 技术、中文编码问题、Javac 编译原理、class 文件结构解析、ClassLoader 工作机制及 JVM 的内存管理等。最后介绍了 Java 服务端技术，主要包括 Servlet、Session 与 Cookie、Tomcat 与 Jetty 服务器、Spring 容器、iBatis 框架和 Velocity 框架等原理介绍，并介绍了服务端的一些优化技术。本书不仅介绍这些技术和框架的工作原理，而且结合示例来讲解，通过通俗易懂的文字和丰富、生动的配图，让读者充分并深入理解它们的内部工作原理，同时还结合了设计模式来介绍这些技术背后的架构思维。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

深入分析 Java Web 技术内幕 / 许令波著. —修订本. —北京：电子工业出版社，2014.8
（阿里巴巴集团技术丛书）
ISBN 978-7-121-23293-0

I. ①深… II. ①许… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2014）第 106866 号

策划编辑：刘 皎
责任编辑：徐津平
印 刷：三河市鑫金马印装有限公司
装 订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×980 1/16 印张：30.5 字数：600 千字
版 次：2012 年 9 月第 1 版
2014 年 8 月第 2 版
印 次：2014 年 8 月第 1 次印刷
印 数：4000 册 定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

第 8 章

JVM 内存管理

与其他高级语言（如 C 和 C++）不太一样，在 Java 中我们基本上不会显式地调用分配内存的函数，我们甚至不用关心到底哪些程序指令需要分配内存、哪些不需要分配内存。因为在 Java 中，分配内存和回收内存都由 JVM 自动完成了，很少会遇到像 C++ 程序中那样令人头疼的内存泄漏问题。

虽然 Java 语言的这些特点很容易“惯坏”开发人员，使得我们不需要太关心到底程序是怎么使用内存的，使用了多少内存。但是我们最好也了解 Java 是如何管理内存的，当我们真的遇到 `OutOfMemoryError` 时不会奇怪地问，为什么 Java 也有内存泄漏。要快速地知道到底什么地方导致了 `OutOfMemoryError`，并能根据错误日志快速地定位出错原因。

本章首先从操作系统层面简单介绍物理内存的分配和 Java 运行的内存分配之间的关系，也就是先搞明白在 Java 中使用的内存与物理内存有何区别。其次介绍 Java 如何使用从物理内存申请下来的内存，以及如何来划分它们，后面还会介绍 Java 的核心技术：如何分配和回收内存。最后通过一些例子介绍如何解决 `OutOfMemoryError`，并提供一些处理这类问题的常用手段。

8.1 物理内存与虚拟内存

所谓物理内存就是我们通常所说的 RAM（随机存储器）。在计算机中，还有一个存储单元叫寄存器，它用于存储计算单元执行指令（如浮点、整数等运算时）的中间结果。寄存器的大小决定了一次计算可使用的最大数值。

连接处理器和 RAM 或者处理器和寄存器的是地址总线，这个地址总线的宽度影响了物理地址的索引范围，因为总线的宽度决定了处理器一次可以从寄存器或者内存中获取多少个 bit。同时也决定了处理器最大可以寻址的地址空间，如 32 位地址总线可以寻址的范围为 0x0000 0000~0xffff ffff。这个范围是 $2^{32}=4\ 294\ 967\ 296$ 个内存位置，每个地址会引用一个字节，所以 32 位总线宽度可以有 4GB 的内存空间。

通常情况下，地址总线和寄存器或者 RAM 有相同的位数，因为这样更容易传输数据，但是也有不一致的情况，如 x86 的 32 位寄存器宽度的物理地址可能有两种大小，分别是 32 位物理地址和 36 位物理地址，拥有 36 位物理地址的是 Pentium Pro 和更高型号。

除了在学校的编译原理的实践课或者要开发硬件程序的驱动程序时需要直接通过程序访问存储器外，我们大部分情况下都调用操作系统提供的接口来访问内存，在 Java 中甚至不需要写和内存相关的代码。

不管是在 Windows 系统还是 Linux 系统下，我们要运行程序，都要向操作系统先申请内存地址。通常操作系统管理内存的申请空间是按照进程来管理的，每个进程拥有一段独立的地址空间，每个进程之间不会相互重合，操作系统也会保证每个进程只能访问自己的内存空间。这主要是从程序的安全性来考虑的，也便于操作系统来管理物理内存。

其实上面所说的进程的内存空间的独立主要是指逻辑上独立，也就是这个独立是由操作系统来保证的，但是真正的物理空间是不是只能由一个进程来使用就不一定了。因为随着程序越来越庞大和设计的多任务性，物理内存无法满足程序的需求，在这种情况下就有了虚拟内存的出现。

虚拟内存的出现使得多个进程在同时运行时可以共享物理内存，这里的共享只是空间上共享，在逻辑上它们仍然是不能相互访问的。虚拟地址不但可以让进程共享物理内存、提高内存利用率，而且还能够扩展内存的地址空间，如一个虚拟地址可能被映射到一段物

理内存、文件或者其他可以寻址的存储上。一个进程在不活动的情况下，操作系统将这个物理内存中的数据移到一个磁盘文件中（也就是通常 Windows 系统上的页面文件，或者 Linux 系统上的交换分区），而真正高效的物理内存留给正在活动的程序使用。在这种情况下，在我们重新唤醒一个很长时间没有使用的程序时，磁盘会吱吱作响，并且会有一个短暂的停顿得到印证，这时操作系统又会把磁盘上的数据重新交互到物理内存中。但是我们必须要避免这种情况的经常出现，如果操作系统频繁地交互物理内存的数据和磁盘数据，则效率将会非常低，尤其是在 Linux 服务器上，我们要关注 Linux 中 swap 的分区活跃度。如果 swap 分区被频繁使用，系统将会非常缓慢，很可能意味着物理内存已经严重不足或者某些程序没有及时释放内存。

8.2 内核空间与用户空间

一个计算机通常有一定大小的内存空间，如使用的计算机是 4GB 的地址空间，但是程序并不能完全使用这些地址空间，因为这些地址空间被划分为内核空间和用户空间。程序只能使用用户空间的内存，这里所说的使用是指程序能够申请的内存空间，并不是程序真正访问的地址空间。

内核空间主要是指操作系统运行时所使用的用于程序调度、虚拟内存的使用或者连接硬件资源等的程序逻辑。为何需要内存空间和用户空间的划分呢？很显然和前面所说的每个进程都独立使用属于自己的内存一样，为了保证操作系统的稳定性，运行在操作系统中的用户程序不能访问操作系统所使用的内存空间。这也是从安全性上考虑的，如访问硬件资源只能由操作系统来发起，用户程序不允许直接访问硬件资源。如果用户程序需要访问硬件资源，如网络连接等，可以调用操作系统提供的接口来实现，这个调用接口的过程也就是系统调用。每一次系统调用都会存在两个内存空间的切换，通常的网络传输也是一次系统调用，通过网络传输的数据先是从内核空间接收到远程主机的数据，然后再从内核空间复制到用户空间，供用户程序使用。这种从内核空间到用户空间的数据复制很费时，虽然保住了程序运行的安全性和稳定性，但是也牺牲了一部分效率。但是现在已经出现了很多其他技术能够减少这种从内核空间到用户空间的数据复制的方式，如 Linux 系统提供了 sendfile 文件传输方式。

内核空间和用户空间的大小如何分配也是一个问题，是更多地分配给用户空间供用户程序使用，还是首先保住内核有足够的空间来运行，这要平衡一下。如果是一台登录服务

器，很显然，要分配更多的内核空间，因为每一个登录用户操作系统都会初始化一个用户进程，这个进程大部分都在内核空间里运行。在当前的 Windows 32 位操作系统中默认内核空间 and 用户空间的比例是 1:1（2GB 的内核空间，2GB 的用户空间），而在 32 位 Linux 系统中默认的比例是 1:3（1GB 的内核空间，3GB 的用户空间）。

8.3 在 Java 中哪些组件需要使用内存

Java 启动后也作为一个进程运行在操作系统中，那么这个进程有哪些部分需要分配内存空间呢？

8.3.1 Java 堆

Java 堆是用于存储 Java 对象的内存区域，堆的大小在 JVM 启动时就一次向操作系统申请完成，通过 `-Xmx` 和 `-Xms` 两个选项来控制大小，`Xmx` 表示堆的最大大小，`Xms` 表示初始大小。一旦分配完成，堆的大小就将固定，不能在内存不够时再向操作系统重新申请，同时当内存空闲时也不能将多余的空间交还给操作系统。

在 Java 堆中内存空间的管理由 JVM 来控制，对象创建由 Java 应用程序控制，但是对象所占的空间释放由管理堆内存的垃圾收集器来完成。根据垃圾收集（GC）算法的不同，内存回收的方式和时机也会不同。

8.3.2 线程

JVM 运行实际程序的实体是线程，当然线程需要内存空间来存储一些必要的数。每个线程创建时 JVM 都会为它创建一个堆栈，堆栈的大小根据不同的 JVM 实现而不同，通常在 256KB~756KB 之间。

线程所占空间相比堆空间来说比较小。但是如果线程过多，线程堆栈的总内存使用量可能也非常大。当前有很多应用程序根据 CPU 的核数来分配创建的线程数，如果运行的应用程序的线程数量比可用于处理它们的处理器数量多，效率通常很低，并且可能导致比较差的性能和更高的内存占用率。

8.3.3 类和类加载器

在 Java 中的类和加载类的类加载器本身同样需要存储空间，在 Sun JDK 中它们也被存储在堆中，这个区域叫做永久代（PermGen 区）。

需要注意的一点是 JVM 是按需来加载类的，曾经有个疑问：JVM 如果要加载一个 jar 包是否把这个 jar 包中的所有类都加载到内存中？显然不是的。JVM 只会加载那些在你的应用程序中明确使用的类到内存中。要查看 JVM 到底加载了哪些类，可以在启动参数上加上 `-verbose:class`。

理论上使用的 Java 类越多，需要占用的内存也会越多，还有一种情况是可能会重复加载同一个类。通常情况下 JVM 只会加载一个类到内存一次，但是如果是自己实现的类加载器会出现重复加载的情况，如果 PermGen 区不能对已经失效的类做卸载，可能会导致 PermGen 区内存泄漏。所以需要注意 PermGen 区的内存回收问题。通常一个类能够被卸载，有如下条件需要被满足。

- ⊙ 在 Java 堆中没有对表示该类加载器的 `java.lang.ClassLoader` 对象的引用。
- ⊙ Java 堆没有对表示类加载器加载的类的任何 `java.lang.Class` 对象的引用。
- ⊙ 在 Java 堆上该类加载器加载的任何类的所有对象都不再存活（被引用）。

需要注意的是，JVM 所创建的 3 个默认类加载器 `Bootstrap ClassLoader`、`ExtClassLoader` 和 `AppClassLoader` 都不可能满足这些条件，因此，任何系统类（如 `java.lang.String`）或通过应用程序类加载器加载的任何应用程序类都不能在运行时释放。

8.3.4 NIO

Java 在 1.4 版本以后添加了新 I/O（NIO）类库，引入了一种基于通道和缓冲区来执行 I/O 的新方式。就像在 Java 堆上的内存支持 I/O 缓冲区一样，NIO 使用 `java.nio.ByteBuffer.allocateDirect()` 方法分配内存，这种方式也就是通常所说的 NIO direct memory。`ByteBuffer.allocateDirect()` 分配的内存使用的是本机内存而不是 Java 堆上的内存，这也进一步说明每次分配内存时会调用操作系统的 `os::malloc()` 函数。另外一方面直接 `ByteBuffer` 产生的数据如果和网络或者磁盘交互都在操作系统的内核空间中发生，不需要将数据复制

到 Java 内存中，很显然执行这种 I/O 操作要比一般的从操作系统的内核空间到 Java 堆上的切换操作快得多，因为它们可以避免在 Java 堆与本机堆之间复制数据。如果你的 I/O 频繁地发送很小的数据，这种系统调用的开销可能会抵消数据在内核空间和用户空间复制带来的好处。

直接 `ByteBuffer` 对象会自动清理本机缓冲区，但这个过程只能作为 Java 堆 GC 的一部分来执行，因此它们不会自动响应施加在本机堆上的压力。GC 仅在 Java 堆被填满，以至于无法为堆分配请求提供服务时发生，或者在 Java 应用程序中显示请求时发生。当前在很多 NIO 框架中都在代码中显式地调用 `System.gc()` 来释放 NIO 持有的内存。但是这种方式会影响应用程序的性能，因为会增加 GC 的次数，一般情况下通过设置 `-XX:+DisableExplicitGC` 来控制 `System.gc()` 的影响，但是又会导致 NIO direct memory 内存泄漏问题。

8.3.5 JNI

JNI 技术使得本机代码（如 C 语言程序）可以调用 Java 方法，也就是通常所说的 `native memory`。实际上 Java 运行时本身也依赖于 JNI 代码来实现类库功能，如文件操作、网络 I/O 操作或者其他系统调用。所以 JNI 也会增加 Java 运行时的本机内存占用。

8.4 JVM 内存结构

前面介绍了内存的不同形态：物理内存和虚拟内存。介绍了内存的使用形式：内核空间和用户空间。接着又介绍了 Java 有哪些组件需要使用内存。下面着重介绍在 JVM 中是如何使用内存的。

JVM 是按照运行时数据的存储结构来划分内存结构的，JVM 在运行 Java 程序时，将它们划分成几种不同格式的数据，分别存储在不同的区域，这些数据统一称为运行时数据（`Runtime Data`）。运行时数据包括 Java 程序本身的数据信息和 JVM 运行 Java 程序需要的额外数据信息，如要记录当前程序指令执行的指针（又称为 PC 指针）等。

在 Java 虚拟机规范中将 Java 运行时数据划分为 6 种，分别为：

- ◎ PC 寄存器数据；

- ⊙ Java 栈;
- ⊙ 堆;
- ⊙ 方法区;
- ⊙ 本地方法区;
- ⊙ 运行时常量池。

8.4.1 PC 寄存器

PC 寄存器严格来说是一个数据结构,它用于保存当前正常执行的程序的内存地址。同时 Java 程序是多线程执行的,所以不可能一直都按照线性执行下去,当有多个线程交叉执行时,被中断线程的程序当前执行到哪条的内存地址必然要保存下来,以便于它被恢复执行时再按照被中断时的指令地址继续执行下去。这很好理解,它就像一个记事员一样记录下哪个线程当前执行到哪条指令了。

但是 JVM 规范只定义了 Java 方法需要记录指针信息,而对于 Native 方法,并没有要求记录执行的指针地址。

8.4.2 Java 栈

Java 栈总是和线程关联在一起,每当创建一个线程时,JVM 就会为这个线程创建一个对应的 Java 栈,在这个 Java 栈中又会含有多个栈帧(Frames),这些栈帧是与每个方法关联起来的,每运行一个方法就创建一个栈帧,每个栈帧会含有一些内部变量(在方法内定义的变量)、操作栈和方法返回值等信息。

每当一个方法执行完成时,这个栈帧就会弹出栈帧的元素作为这个方法的返回值,并清除这个栈帧,Java 栈的栈顶的栈帧就是当前正在执行的活动栈,也就是当前正在执行的方法,PC 寄存器也会指向这个地址。只有这个活动的栈帧的本地变量可以被操作栈使用,当在这个栈帧中调用另外一个方法时,与之对应的一个新的栈帧又被创建,这个新创建的栈帧又被放到 Java 栈的顶部,变为当前的活动栈帧。同样现在只有这个栈帧的本地变量才能被使用,当在这个栈帧中所有指令执行完成时这个栈帧移出 Java 栈,刚才的那个栈帧又变为活动栈帧,前面的栈帧的返回值又变为这个栈帧的操作栈中的一个操作数。如果

前面的栈帧没有返回值，那么当前的栈帧的操作栈的操作数没有变化。

由于 Java 栈是与 Java 线程对应起来的，这个数据不是线程共享的，所以我们不用关心它的数据一致性问题，也不会存在同步锁的问题。

8.4.3 堆

堆是存储 Java 对象的地方，它是 JVM 管理 Java 对象的核心存储区域，堆是 Java 程序员最应该关心的，因为它是我们的应用程序与内存关系最密切的存储区域。

每一个存储在堆中的 Java 对象都会是这个对象的类的一个副本，它会复制包括继承自它父类的所有非静态属性。

堆是被所有 Java 线程所共享的，所以对它的访问需要注意同步问题，方法和对应的属性都需要保证一致性。

8.4.4 方法区

JVM 方法区是用于存储类结构信息的地方，如在第 7 章介绍的将一个 class 文件解析成 JVM 能识别的几个部分，这些不同的部分在这个 class 被加载到 JVM 时，会被存储在不同的数据结构中，其中的常量池、域、方法数据、方法体、构造函数，包括类中的专用方法、实例初始化、接口初始化都存储在这个区域。

方法区这个存储区域也属于后面介绍的 Java 堆中的一部分，也就是我们通常所说的 Java 堆中的永久区。这个区域可以被所有的线程共享，并且它的大小可以通过参数来设置。

这个方法区存储区域的大小一般在程序启动后的一段时间内就是固定的了，JVM 运行一段时间后，需要加载的类通常都已经加载到 JVM 中了。但是有一种情况是需要注意的，那就是在项目中如果存在对类的动态编译，而且是同样一个类的多次编译，那么需要观察方法区的大小是否能满足类存储。

方法区这个区域有点特殊，由于它不像其他 Java 堆一样会频繁地被 GC 回收器回收，它存储的信息相对比较稳定，但是它仍然占用了 Java 堆的空间，所以仍然会被 JVM 的 GC 回收器来管理。在一些特殊的场合下，有时通常需要缓存一块内容，这个内容也很少变动，但是如果把它置于 Java 堆中它会不停地被 GC 回收器扫描，直到经过很长的时间后会进入

Old 区。在这种情况下，通常是能控制这个缓存区域中数据的生命周期的，我们不希望它被 JVM 内存管理，但是又希望它在内存中。面对这种情况，淘宝正在开发一种技术用于在 JVM 中分配另外一个内存存储区域，它不需要 GC 回收器来回收，但是可以和其他内存中对象一样来使用。

8.4.5 运行时常量池

在 JVM 规范中是这样定义运行时常量池这个数据结构的：Runtime Constant Pool 代表运行时每个 class 文件中的常量表。它包括几种常量：编译期的数字常量、方法或者域的引用（在运行时解析）。Runtime Constant Pool 的功能类似于传统编程语言的符号表，尽管它包含的数据比典型的符号表要丰富得多。每个 Runtime Constant pool 都是在 JVM 的 Method area 中分配的，每个 Class 或者 Interface 的 Constant Pool 都是在 JVM 创建 class 或接口时创建的。

上面的描述可能使你有点迷惑，这个常量池与前面方法区的常量池是否是一回事？答案是肯定的。它是方法区的一部分，所以它的存储也受方法区的规范约束，如果常量池无法分配，同样会抛出 OutOfMemoryError。

8.4.6 本地方法栈

本地方法栈是为 JVM 运行 Native 方法准备的空间，它和前面介绍的 Java 栈的作用是类似的，由于很多 Native 方法都是用 C 语言实现的，所以它通常又叫 C 栈，除了在我们的代码中包含的常规的 Native 方法会使用这个存储空间，在 JVM 利用 JIT 技术时会将一些 Java 方法重新编译为 Native Code 代码，这些编译后的本地代码通常也是利用这个栈来跟踪方法的执行状态的。

在 JVM 规范中没有对这个区域的严格限制，它可以由不同的 JVM 实现者自由实现，但是它和其他存储区一样也会抛出 OutOfMemoryError 和 StackOverflowError。

8.5 JVM 内存分配策略

在分析 JVM 内存分配策略之前我们先介绍一下通常情况下操作系统都是采用哪些策略来分配内存的。

8.5.1 通常的内存分配策略

我想大家都学过操作系统，在操作系统中将内存分配策略分为三种，分别是：

- ⊙ 静态内存分配；
- ⊙ 栈内存分配；
- ⊙ 堆内存分配。

静态内存分配是指在程序编译时就能确定每个数据在运行时的存储空间需求，因此在编译时就可以给它们分配固定的内存空间。这种分配策略不允许在程序代码中有可变数据结构（如可变数组）的存在，也不允许有嵌套或者递归的结构出现，因为它们都会导致编译程序无法计算准确的存储空间需求。

栈式内存分配也可称为动态存储分配，是由一个类似于堆栈的运行栈来实现的。和静态内存分配相反，在栈式内存方案中，程序对数据区的需求在编译时是完全未知的，只有到运行时才能知道，但是规定在运行中进入一个程序模块时，必须知道该程序模块所需的数据区大小才能够为其分配内存。和我们所熟知的数据结构中的栈一样，栈式内存分配按照先进后出的原则进行分配。

在编写程序时除了在编译时能确定数据的存储空间和在程序入口处能知道存储空间外，还有一种情况就是当程序真正运行到相应代码时才会知道空间大小，在这种情况下我们就需要堆这种分配策略。

这几种内存分配策略中，很明显堆分配策略是最自由的，但是这种分配策略对操作系统和内存管理程序来说是一种挑战。另外，这个动态的内存分配是在程序运行时才执行的，它的运行效率也是比较差的。

8.5.2 Java 中的内存分配详解

从前面的 JVM 内存结构的分析我们可知，JVM 内存分配主要基于两种，分别是堆和栈。先来说说 Java 栈是如何分配的。

Java 栈的分配是和线程绑定在一起的，当我们创建一个线程时，很显然，JVM 就会为

这个线程创建一个新的 Java 栈，一个线程的方法的调用和返回对应于这个 Java 栈的压栈和出栈。当线程激活一个 Java 方法时，JVM 就会在线程的 Java 堆栈里新压入一个帧，这个帧自然成了当前帧。在此方法执行期间，这个帧将用来保存参数、局部变量、中间计算过程和其他数据。

栈中主要存放一些基本类型的变量数据 (int、short、long、byte、float、double、boolean、char) 和对象句柄 (引用)。存取速度比堆要快，仅次于寄存器，栈数据可以共享。缺点是，存在栈中的数据大小与生存期必须是确定的，这也导致缺乏了其灵活性。

如下面这段代码：

```
public void stack(String[] arg) {
    String str = "junshan";
    if (str.equals("junshan")) {
        int i = 3;
        while (i > 0) {
            long j = 1;
            i--;
        }
    } else {
        char b = 'a';
        System.out.println(b);
    }
}
```

这段代码的 stack 方法中定义了多个变量，这些变量在运行时需要存储空间，同时在执行指令时 JVM 也需要知道操作栈的大小，这些数据都会在 Javac 编译这段代码时就已经确定，下面是这个方法对应的 class 字节码：

```
public void stack(java.lang.String[]);
Code:
    Stack=2, Locals=6, Args_size=2
    0:   ldc #3; //String junshan
    2:   astore_2
    3:   aload_2
    4:   ldc #3; //String junshan
    6:   invokevirtual   #4; //Method java/lang/String.equals:(Ljava/lang/
Object;)Z
    9:   ifeq    30
```

```

12:  iconst_3
13:  istore_3
14:  iload_3
15:  ifle     27
18:  lconst_1
19:  lstore   4
21:  iinc     3, -1
24:  goto     14
27:  goto     40
30:  bipush  97
32:  istore_3
33:  getstatic #5; //Field java/lang/System.out:Ljava/io/PrintStream;
36:  iload_3
37:  invokevirtual #6; //Method java/io/PrintStream.println:(C)V
40:  return
LineNumberTable:
  line 15: 0
  line 16: 3
  line 17: 12
  line 18: 14
  line 19: 18
  line 20: 21
  line 21: 24
  line 22: 27
  line 23: 30
  line 24: 33
  line 26: 40

LocalVariableTable:
  Start  Length  Slot  Name  Signature
  21      3      4    j      J
  14     13      3    i      I
  33      7      3    b      C
  0      41      0  this      Lheap/StackSize;
  0      41      1  arg      [Ljava/lang/String;
  3      38      2  str      Ljava/lang/String;

```

在这个方法的 attribute 中就已经知道了 stack 和 local variable 的大小，分别是 2 和 6。还有一点不得不提，就是这里的大小指定的是最大值，为什么是最大值呢？因为 JVM 在

真正执行时分配的 `stack` 和 `local variable` 的空间是可以共用的。举例来说,上面的 6 个 `local variable` 除去变量 0 是 `this` 指针外,其他 5 个都是在这个方法中定义的,这 6 个变量需要的 `Slot` 是 $1+1+1+1+2+1=7$,但是实际上使用的 `Slot` 只有 4 个,这是因为不同的变量作用范围如果没有重合,`Slot` 则可以重复使用。

每个 Java 应用都唯一对应一个 JVM 实例,每个实例唯一对应一个堆。应用程序在运行中所创建的所有类实例或数组都放在这个堆中,并由应用程序所有的线程共享。在 Java 中分配堆内存是自动初始化的,所有对象的存储空间都是在堆中分配的,但是这个对象的引用却是在堆栈中分配的。也就是说在建立一个对象时两个地方都分配内存,在堆中分配的内存实际建立这个对象,而在堆栈中分配的内存只是一个指向这个堆对象的指针(引用)而已。

Java 的堆是一个运行时数据区,这些对象通过 `new`、`newarray`、`anewarray` 和 `multianewarray` 等指令建立,它们不需要程序代码来显式地释放。堆是由垃圾回收来负责的,堆的优势是可以动态地分配内存大小,生存期也不必事先告诉编译器,因为它是在运行时动态分配内存的,Java 的垃圾收集器会自动收走这些不再使用的数据。但缺点是,由于要在运行时动态分配内存,存取速度较慢。

如下代码描述新对象是如何在堆上分配内存的:

```
public static void main(String[] args) {
    String str = new String("hello world!");
}
```

上面的代码创建了一个 `String` 对象,这个 `String` 对象将会在堆上分配内存,JVM 创建对象的字节码指令如下:

```
public static void main(java.lang.String[]);
  Code:
    Stack=3, Locals=2, Args_size=1
    0:   new #7; //class java/lang/String
    3:   dup
    4:   ldc #8; //String hello world!
    6:   invokespecial   #9; //Method java/lang/String."<init>":(Ljava/lang/
String;)V
    9:   astore_1
   10:   return
  LineNumberTable:
```

```

line 29: 0
line 35: 10

LocalVariableTable:
  Start Length Slot Name Signature
    0     11     0  args    [Ljava/lang/String;
   10     1     1  str     Ljava/lang/String;

```

先执行 `new` 指令，这个 `new` 指令根据后面的 16 位的 “#7” 常量池索引创建指定类型的对象，而该 #7 索引所指向的入口类型首先必须是 `CONSTANT_Class_info`，也就是它必须是类类型，然后 JVM 会为这个类的新对象分配一个空间，这个新对象的属性值都设置为默认值，最后将执行这个新对象的 `objectref` 引用压入栈顶。

`new` 指令执行完成后，得到的对象还没有初始化，所以这个新对象并没有创建完成，这个对象的引用在这时不应该赋值给 `str` 变量，而应该接下去就调用这个类的构造函数初始化类，这时就必须将 `objectref` 引用复制一份，在新对象初始化完成后再将这个引用赋值给本地变量。调用构造函数是通过 `invokespecial` 指令完成的，构造函数如果有参数要传递，则先将参数压栈。构造函数执行完成后再将 `objectref` 的对象引用赋值给本地变量 1，这样一个新对象才创建完成。

上面的内存分配策略定义从编译原理的教材中总结而来，除静态内存分配之外，都显得很呆板和难以理解，下面撇开静态内存分配，集中比较堆和栈。

从堆和栈的功能和作用来通俗地比较，堆主要用来存放对象，栈主要用来执行程序，这种不同主要是由堆和栈的特点决定的。

在编程中，如 C/C++ 中，所有的方法调用都是通过栈来进行的，所有的局部变量、形式参数都是从栈中分配内存空间的。实际上也不是什么分配，只是从栈顶向上用就行，就好像工厂中的传送带一样，栈指针会自动指引你到放东西的位置，你所要做的只是把东西放下来就行。在退出函数时，修改栈指针就可以把栈中的内容销毁。这样的模式速度最快，当然要用来运行程序了。需要注意的是，在分配时，如为一个即将要调用的程序模块分配数据区时，应事先知道这个数据区的大小，也就是说虽然分配是在程序运行时进行的，但是分配的大小是确定的、不变的，而这个“大小多少”是在编译时确定的，而不是在运行时。

堆在应用程序运行时请求操作系统给自己分配内存，由于操作系统管理内存分配，所

以在分配和销毁时都要占用时间，因此用堆的效率非常低。但是堆的优点在于，编译器不必知道要从堆里分配多少存储空间，也不必知道存储的数据要在堆里停留多长时间，因此，用堆保存数据时会得到更大的灵活性。事实上，由于面向对象的多态性，堆内存分配是不可避免的，因为多态变量所需的存储空间只有在运行时创建了对象之后才能确定。在 C++ 中，要求创建一个对象时，只需用 `new` 命令编制相关的代码即可。执行这些代码时，会在堆里自动进行数据的保存。当然，为达到这种灵活性，必然会付出一定的代价——在堆里分配存储空间时会花掉更长的时间。

8.6 JVM 内存回收策略

Java 语言和其他语言的一个很大不同之处就是 Java 开发人员不需要了解内存这个概念，因为在 Java 中没有什么语法和内存直接有联系，不像在 C 或 C++ 中有 `malloc` 这种语法直接操作内存。但是程序执行都需要内存空间来支持，不然我们的那些数据存在哪里？Java 语言没有提供直接操作内存的语法，那我们的数据又是如何申请内存的呢？就 Java 语言本身来说，通常显式的内存申请有两种：一种是静态内存分配，另一种是动态内存分配。

8.6.1 静态内存分配和回收

在 Java 中静态内存分配是指在 Java 被编译时就已经能够确定需要的内存空间，当程序被加载时系统把内存一次性分配给它。这些内存不会在程序执行时发生变化，直到程序执行结束时内存才被回收。在 Java 的类和方法中的局部变量包括原生数据类型（`int`、`long`、`char` 等）和对象的引用都是静态分配内存的，如下面这段代码：

```
public void staticData(int arg){
    String s="String";
    long l=1;
    Long lg=1L;
    Object o = new Object();
    Integer i = 0;
}
```

其中参数 `arg`、`l` 是原生的数据类型，`s`、`o` 和 `i` 是指向对象的引用。在 `Javac` 编译时就已经确定了这些变量的静态内存空间。其中 `arg` 会分配 4 个字节，`long` 会分配 8 个字节，`String`、

Long、Object 和 Integer 是对象的类型，它们的引用会占用 4 个字节空间，所以这个方法占用的静态内存空间是 $4+4+8+4+4+4=28$ 字节。

静态内存空间当这段代码运行结束时回收，根据第 7 章的介绍，我们知道这些静态内存空间是在 Java 栈上分配的，当这个方法运行结束时，对应的栈帧也就撤销，所以分配的静态内存空间也就回收了。

8.6.2 动态内存分配和回收

在前面的例子中变量 lg 和 i 存储与值虽然与 l 和 arg 变量一样，但是它们存储的位置是不一样的，后者是原生数据类型，它们存储在 Java 栈中，方法执行结束就会消失，而前者是对象类型，它们存储在 Java 堆中，它们是可以被共享的，也不一定随着方法执行结束而消失。变量 l 和 lg 的内存空间大小显然也是不一样的，l 在 Java 栈中被分配 8 个字节空间，而 lg 被分配 4 个字节的地址指针空间，这个地址指针指向这个对象在堆中的地址。很显然在堆中 long 类型数字 1 肯定不只 8 个字节，所以 Long 代表的数字肯定比 long 类型占用的空间要大很多。

在 Java 中对象的内存空间是动态分配的，所谓的动态分配就是在程序执行时才知道要分配的存储空间大小，而不是在编译时就能够确定的。lg 代表的 Long 对象，只有 JVM 在解析 Long 类时才知道在这个类中有哪些信息，这些信息都是哪些类型，然后再为这些信息分配相应的存储空间存储相应的值。而这个对象什么时候被回收也是不确定的，只有等到这个对象不再使用时才会被回收。

从前面的分析可知内存的分配是在对象创建时发生的，而内存的回收是以对象不再引用为前提的。这种动态内存的分配和回收是和 Java 中一些数据类型关联的，Java 程序员根本不需要关注内存的分配和回收，只需关注这些数据类型的使用就行了。

那么如何确定这个对象什么时候不被使用，又如何来回收它们，这正是 JVM 的一个很重要的组件——垃圾收集器要解决的问题。

8.6.3 如何检测垃圾

垃圾收集器必须能够完成两件事情：一件是能够正确地检测出垃圾对象，另一件是能够释放垃圾对象占用的内存空间。其中如何检测出垃圾是垃圾收集器的关键所在。

从前面的分析已经知道，只要是某个对象不再被其他活动对象引用，那么这个对象就可以被回收了。这里的活动对象指的是能够被一个根对象集合到达的对象，如图 8-1 所示。

在图 8-1 中除了 f 和 h 对象之外，其他都可以称为活动对象，因为它们都可以被根对象集合到达。h 对象虽然也被 f 对象引用，但是 h 对象不能够被根对象集合达到，所以它们都是非活动对象，可以被垃圾收集器回收。

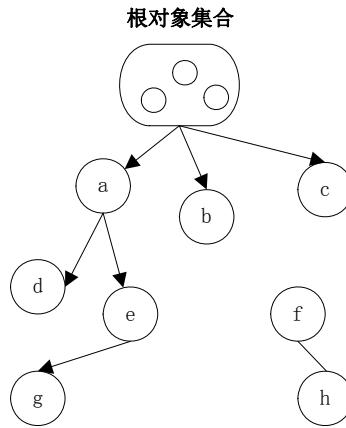


图 8-1 活动对象

那么在这个根对象集合中又都是些什么呢？虽然根对象集合和 JVM 的具体实现也有关系，但是大都会包含如下一些元素。

- ⊙ 在方法中局部变量区的对象的引用：如在前面的 `staticData` 方法中定义的 `lg` 和 `o` 等对象的引用就是根对象集合中的一个根对象，这些根对象直接存储在栈帧的局部变量区中。
- ⊙ 在 Java 操作栈中的对象引用：有些对象是直接的操作栈中持有的，所以操作栈肯定也包含根对象集合。
- ⊙ 在常量池中的对象引用：每个类都会包含一个常量池，这些常量池中也会包含很多对象引用，如表示类名的字符串就保存在堆中，那么常量池中只会持有这个字符串对象的引用。
- ⊙ 在本地方法中持有的对象引用：有些对象被传入本地方法中，但是这些对象还没有被释放。

- 类的 Class 对象：当每个类被 JVM 加载时都会创建一个代表这个类的唯一数据类型的 Class 对象，而这个 Class 对象也同样存放在堆中，当这个类不再被使用时，在方法区中类数据和这个 Class 对象同样需要被回收。

JVM 在做垃圾回收时会检查堆中的所有对象是否都会被这些根对象直接或者间接引用，能够被引用的对象就是活动对象，否则就可以被垃圾收集器回收。

8.6.4 基于分代的垃圾收集算法

经过这么长时间的发展，垃圾收集算法已经有很多种，算法各有优缺点，这里将主要介绍在 hotspot 中使用的基于分代的垃圾收集方式。

该算法的设计思路是：把对象按照寿命长短来分组，分为年轻代和年老代，新创建的对象被分在年轻代，如果对象经过几次回收后仍然存活，那么再把这个对象划分到年老代。年老代的收集频度不像年轻代那么频繁，这样就减少了每次垃圾收集时所扫描的对象的数量，从而提高了垃圾回收效率。

这种设计的思路是把堆划分成若干个子堆，每个子堆对应一个年龄代，如图 8-2 所示。

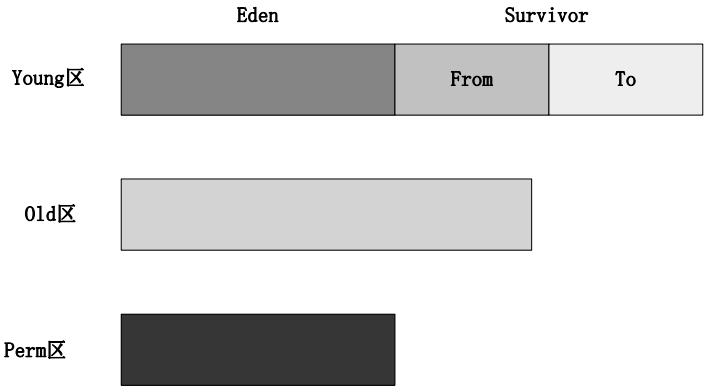


图 8-2 基于分代的堆结构

JVM 将整个堆划分为 Young 区、Old 区和 Perm 区，分别存放不同年龄的对象，这三个区存放的对象有如下区别。

- Young 区又分为 Eden 区和两个 Survivor 区，其中所有新创建的对象都在 Eden 区，当 Eden 区满后会触发 minor GC 将 Eden 区仍然存活的对象复制到其中一个

Survivor 区中，另外一个 Survivor 区中的存活对象也复制到这个 Survivor 中，以保证始终有一个 Survivor 区是空的。

- ④ Old 区存放的是 Young 区的 Survivor 满后触发 minor GC 后仍然存活的对象，当 Eden 区满后会将对象存放到 Survivor 区中，如果 Survivor 区仍然存不下这些对象，GC 收集器会将这些对象直接存放到 Old 区。如果在 Survivor 区中的对象足够老，也直接存放到 Old 区。如果 Old 区也满了，将会触发 Full GC，回收整个堆内存。
- ④ Perm 区存放的主要是类的 Class 对象，如果一个类被频繁地加载，也可能导致 Perm 区满，Perm 区的垃圾回收也是由 Full GC 触发的。

在 Sun 的 JVM 中提供了一个 visualvm 工具，其中有个 Visual GC 插件可以观察到 JVM 的不同代的垃圾回收情况，如图 8-3 所示。

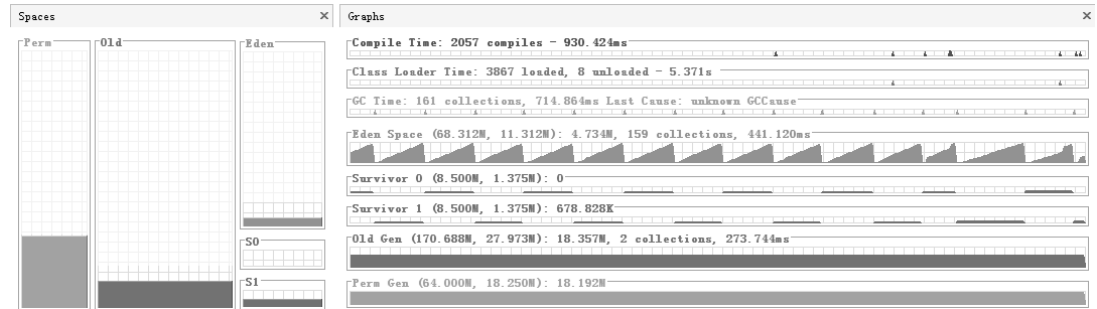


图 8-3 Visual GC 插件

通过 Visual GC 插件可以观察到每个代的当前内存大小和回收的次数等。

Sun 对堆中的不同代的大小也给出了建议，一般建议 Young 区的大小为整个堆的 1/4，而 Young 区中 Survivor 区一般设置为整个 Young 区的 1/8。

GC 收集器对这些区采用的垃圾收集算法也不一样，Hotspot 提供了三类垃圾收集算法，下面详细介绍这三类垃圾收集算法的区别和使用方法。这三类垃圾收集算法分别是：

- ④ Serial Collector;
- ④ Parallel Collector;
- ④ CMS Collector。

1. Serial Collector

Serial Collector 是 JVM 在 client 模式下默认的 GC 方式。可以通过 JVM 配置参数 `-XX:+UseSerialGC` 来指定 GC 使用该收集算法。我们指定所有的对象都在 Young 区的 Eden 中创建，但是如果创建的对象超过 Eden 区的总大小，或者超过了 `PretenureSizeThreshold` 配置参数配置的大小，就只能在 Old 区分配了，如 `-XX:PretenureSizeThreshold= 30720` 在实际使用中很少发生。

当 Eden 空间不足时就触发了 Minor GC，触发 Minor GC 时首先会检查之前每次 Minor GC 时晋升到 Old 区的平均对象大小是否大于 Old 区的剩余空间，如果大于，则将直接触发 Full GC，如果小于，则要看 `HandlePromotionFailure` 参数 (`-XX:-HandlePromotionFailure`) 的值。如果为 `true`，仅触发 Minor GC，否则再触发一次 Full GC。其实这个规则很好理解，如果每次晋升的对象大小都超过了 Old 区的剩余空间，那么说明当前的 Old 区的空间已经不能满足新对象所占空间的大小，只有触发 Full GC 才能获得更多的内存空间。

如这个例子：

```
public static void main2(String[] args) throws Exception {
    int m = 1024 * 1024;
    byte[] b = new byte[2*m];
    byte[] b2 = new byte[2*m];
    byte[] b3 = new byte[2*m];
    byte[] b4 = new byte[2*m];
    byte[] b5 = new byte[2*m];
    byte[] b6 = new byte[2*m];
    byte[] b7 = new byte[2*m];
}
```

Java 参数为 `java -Xms20M -Xmx20M -Xmn10M -XX:+UseSerialGC -XX:+PrintGCDetails`。

GC 日志如下：

```
[DefNew: 6979K->305K(9216K), 0.0108985 secs] 6979K->6449K(19456K), 0.0110814
secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
[Full GC [Tenured: 6144K->8192K(10240K), 0.0182885 secs] 12767K->12594K
(19456K), [Perm : 2555K->2555K(12288K)], 0.0184352 secs] [Times: user=0.00
sys=0.02, real=0.02 secs]
Heap
  def new generation      total 9216K, used 6587K [0x03ab0000, 0x044b0000,
```

```

0x044b0000)
    eden space 8192K, 80% used [0x03ab0000, 0x0411ee70, 0x042b0000)
    from space 1024K, 0% used [0x043b0000, 0x043b0000, 0x044b0000)
    to space 1024K, 0% used [0x042b0000, 0x042b0000, 0x043b0000)
    tenured generation total 10240K, used 8192K [0x044b0000, 0x04eb0000,
0x04eb0000)
    the space 10240K, 80% used [0x044b0000, 0x04cb0040, 0x04cb0200,
0x04eb0000)
    compacting perm gen total 12288K, used 2563K [0x04eb0000, 0x05ab0000,
0x08eb0000)
    the space 12288K, 20% used [0x04eb0000, 0x05130e90, 0x05131000,
0x05ab0000]

```

从 GC 日志可以看出, Minor GC 每晋升到 Old 区的大小为(6979KB-305KB)–(6979KB-6449 KB) =6144 KB, 而 Old 区的剩余空间为 10240 KB–6144 KB =4096 KB, 显然前者大于后者, 所以直接触发了一次 Full GC。

当 Minor GC 时, 除了将 Eden 区的非活动对象回收以外, 还会把一些老对象也复制到 Old 区中。这个老对象的定义是通过配置参数 MaxTenuringThreshold 来控制的, 如 -XX:MaxTenuringThreshold=10, 则如果这个对象已经被 Minor GC 回收过 10 次后仍然存活, 那么这个对象在这次 Minor GC 后直接放入 Old 区。还有一种情况, 当这次 Minor GC 时 Survivor 区中的 To Space 放不下这些对象时, 这些对象也将直接放入 Old 区。如果 Old 区或者 Perm 区空间不足, 将会触发 Full GC, Full GC 会检查 Heap 堆中的所有对象, 清除所有垃圾对象, 如果是 Perm 区, 会清除已经被卸载的 classloader 中加载的类的信息。

JVM 在做 GC 时由于是串行的, 所以这些动作都是单线程完成的, 在 JVM 中的其他应用程序会全部停止。

2 . Parallel Collector

Parallel GC 根据 Minor GC 和 Full GC 的不同分为三种, 分别是 ParNewGC、ParallelGC 和 ParallelOldGC。

1) ParNewGC

可以通过 -XX:+UseParNewGC 参数来指定, 它的对象分配和回收策略与 Serial Collector 类似, 只是回收的线程不是单线程的, 而是多线程并行回收。在 Parallel Collector 中还有一个 UseAdaptiveSizePolicy 配置参数, 这个参数是用来动态控制 Eden、From Space

和 To Space 的 TenuringThreshold 大小的, 以便于控制哪些对象经过多少次回收后可以直接放入 Old 区。

2) ParallelGC

在 Server 下默认的 GC 方式, 可以通过-XX:+UseParallelGC 参数来强制指定, 并行回收的线程数可以通过-XX:ParallelGCThreads 来指定, 这个值有个计算公式, 如果 CPU 和核数小于 8, 线程数可以和核数一样, 如果大于 8, 值为 $3 + (\text{cpu core} * 5) / 8$ 。

可以通过-Xmn 来控制 Young 区的大小, 如-Xmn10m, 即设置 Young 区的大小为 10MB。在 Young 区内的 Eden、From Space 和 To Space 的大小控制可以通过 SurvivorRatio 参数来完成, 如设置成-XX:SurvivorRatio=8, 表示 Eden 区与 From Space 的大小为 8:1, 如果 Young 区的总大小为 10 MB, 那么 Eden、s0 和 s1 的大小分别为 8 MB、1 MB 和 1 MB。但在默认情况下以-XX:InitialSurvivorRatio 设置的为准, 这个值默认也为 8, 表示的是 Young:s0 为 8:1。

当在 Eden 区中申请内存空间时, 如果 Eden 区不够, 那么看当前申请的空间是否大于等于 Eden 的一半, 如果大于则这次申请的空间直接在 Old 中分配, 如果小于则触发 Minor GC。在触发 GC 之前首先会检查每次晋升到 Old 区的平均大小是否大于 Old 区的剩余空间, 如大于则再触发 Full GC。在这次触发 GC 后仍然会按照这个规则重新检查一次。也就是如果满足上面这个规则, Full GC 会执行两次。

如下面这个例子:

```
public static void main(String[] args) throws Exception {
    int m = 1024 * 1024;
    byte[] b = new byte[2*m];
    byte[] b2 = new byte[2*m];
    byte[] b3 = new byte[2*m];
    byte[] b4 = new byte[2*m];
    byte[] b5 = new byte[2*m];
    byte[] b6 = new byte[2*m];
    byte[] b7 = new byte[2*m];
}
```

JVM 参数为 java -Xms20M -Xmx20M -Xmn10M -XX:+UseParallelGC -XX:+PrintGCDetails。

GC 日志如下:

```
[PSYoungGen: 6912K->368K(8960K)] 6912K->6512K(19200K), 0.0054194 secs]
```

```
[Times: user=0.06 sys=0.00, real=0.01 secs]
  [Full GC [PSYoungGen: 368K->0K(8960K)] [PSOldGen: 6144K->6450K(10240K)]
6512K->6450K(19200K) [PSPermGen: 2548K->2548K(12288K)], 0.0142088 secs] [Times:
user=0.01 sys=0.00, real=0.01 secs]
  [Full GC [PSYoungGen: 6227K->4096K(8960K)] [PSOldGen: 6450K->8498K(10240K)]
12677K->12594K(19200K) [PSPermGen: 2554K->2554K(12288K)], 0.0145918 secs]
[Times: user=0.02 sys=0.00, real=0.02 secs]
Heap
  PSYoungGen      total 8960K, used 6529K [0x084a0000, 0x08ea0000, 0x08ea0000)
    eden space 7680K, 85% used [0x084a0000,0x08b007e8,0x08c20000)
    from space 1280K, 0% used [0x08c20000,0x08c20000,0x08d60000)
    to   space 1280K, 0% used [0x08d60000,0x08d60000,0x08ea0000)
  PSOldGen        total 10240K, used 8498K [0x07aa0000, 0x084a0000, 0x084a0000)
    object space 10240K, 82% used [0x07aa0000,0x082ec850,0x084a0000)
  PSPermGen       total 12288K, used 2563K [0x03aa0000, 0x046a0000, 0x07aa0000)
    object space 12288K, 20% used [0x03aa0000,0x03d20f78,0x046a0000]
```

从 GC 日志可以看出, Minor GC 每次晋升到 Old 区的大小为(6912KB-368 KB)-(6912 KB-6512 KB) = 6144 KB, 而 Old 区的剩余空间为 10240 KB-6227 KB = 4013 KB, 显然前者大于后者, 所以直接触发了两次 Full GC。

在 Young 区的对象经过多次 GC 后有可能仍然存活, 那么它们晋升到 Old 区的规则可以通过如下参数来控制: AlwaysTenure, 默认为 false, 表示只要 Minor GC 时存活就晋升到 old; NeverTenure, 默认为 false, 表示永远晋升到 old 区。如果在上面两个都没设置的情况下设置 UseAdaptiveSizePolicy, 启动时以 InitialTenuringThreshold 值作为存活次数的阈值, 在每次 GC 后会动态调整, 如果不想使用 UseAdaptiveSizePolicy, 则以 MaxTenuringThreshold 为准, 不使用 UseAdaptiveSizePolicy 可以设置为-XX:-UseAdaptiveSizePolicy。如果 Minor GC 时 To Space 不够, 对象也将会直接放到 Old 区。

当 Old 或者 Perm 区空间不足时会触发 Full GC, 如果配置了参数 ScavengeBeforeFullGC, 在 Full GC 之前会先触发 Minor GC。

3) ParallelOldGC

可以通过-XX:+UseParallelOldGC 参数来强制指定, 并行回收的线程数可以通过-XX:ParallelGCThreads 来指定, 这个数字的值有个计算公式, 如果 CPU 和核数小于 8, 线程数可以和核数一样, 如果大于 8, 值为 3+(cpu core*5)/8。

它与 ParallelGC 有何不同呢？其实不同之处在 Full GC 上，前者 Full GC 进行的动作为清空整个 Heap 堆中的垃圾对象，清除 Perm 区中已经被卸载的类信息，并进行压缩。而后者是清除 Heap 堆中的部分垃圾对象，并进行部分的空间压缩。

GC 垃圾回收都是以多线程方式进行的，同样也将暂停所有的应用程序。

3 . CMS Collector

可通过-XX:+UseConcMarkSweepGC 来指定，并发的线程数默认为 4（并行 GC 线程数+3），也可通过 ParallelCMSThreads 来指定。

CMS GC 与上面讨论的 GC 不太一样，它既不是上面所说的 Minor GC，也不是 Full GC，它是基于这两种 GC 之间的一种 GC。它的触发规则是检查 Old 区或者 Perm 区的使用率，当达到一定比例时就会触发 CMS GC，触发时会回收 Old 区中的内存空间。这个比例可以通过 CMSInitiatingOccupancyFraction 参数来指定，默认是 92%，这个默认值是通过 $((100 - \text{MinHeapFreeRatio}) + (\text{double})(\text{CMSTriggerRatio} * \text{MinHeapFreeRatio}) / 100.0) / 100.0$ 计算出来的，其中的 MinHeapFreeRatio 为 40、CMSTriggerRatio 为 80。如果让 Perm 区也使用 CMS GC 可以通过-XX:+CMSClassUnloadingEnabled 来设定，Perm 区的比例默认值也是 92%，这个值可以通过 CMSInitiatingPermOccupancyFraction 设定。这个默认值也是通过一个公式计算出来的： $((100 - \text{MinHeapFreeRatio}) + (\text{double})(\text{CMSTriggerPermRatio} * \text{MinHeapFreeRatio}) / 100.0) / 100.0$ ，其中 MinHeapFreeRatio 为 40，CMSTriggerPermRatio 为 80。

触发 CMS GC 时回收的只是 Old 区或者 Perm 区的垃圾对象，在回收时和前面所说的 Minor GC 和 Full GC 基本没有关系。

在这个模式下的 Minor GC 触发规则和回收规则与 Serial Collector 基本一致，不同之处只是 GC 回收的线程是多线程而已。

触发 Full GC 是在这两种情况下发生的：一种是 Eden 分配失败，Minor GC 后分配到 To Space，To Space 不够再分配到 Old 区，Old 区不够则触发 Full GC；另外一种情况是，当 CMS GC 正在进行时向 Old 申请内存失败则会直接触发 Full GC。

这里还需要特别提醒一下，在 Hotspot 1.6 中使用这种 GC 方式时在程序中显式地调用了 System.gc，且设置了 ExplicitGCInvokesConcurrent 参数，那么使用 NIO 时可能会引发内存泄漏，这个内存泄漏将在后面介绍。

CMS GC 何时执行 JVM 还会有一些时机选择, 如当前的 CPU 是否繁忙等因素, 因此它会有一个计算规则, 并根据这个规则来动态调整。但是这也会给 JVM 带来另外的开销, 如果要去掉这个动态调整功能, 禁止 JVM 自行触发 CMS GC, 可以通过配置参数 `-XX:+UseCMSInitiatingOccupancyOnly` 来实现。

4 . 组合使用这三种 GC (如表 8-1 所示)

表 8-1 三种 GC

GC 组合	Young 区	Old 区
<code>-XX:+UseSerialGC</code>	串行 GC	串行 GC
<code>-XX:+UseParallelGC</code>	PSGC	并行 MSCGC
<code>-XX:+UseParNewGC</code>	并行 GC	串行 GC
<code>-XX:+UseParallelOldGC</code>	PSGC	并行 CompactingGC
<code>-XX:+UseConcMarkSweepGC</code>	ParNewGC	并发 GC 当出现 <code>concurrentMode failure</code> 时采用串行 GC
<code>-XX:+UseConcMarkSweepGC</code> <code>-XX:-UseParNewGC</code>	串行 GC	并发 GC 当出现 <code>ConcurrentMode failure</code> 或 <code>promotionfailed</code> 时则采用串行 GC
不支持的组合方式	(1) <code>-XX:+UseParNewGC-XX:+UseParallelOldGC</code> (2) <code>-XX:+UseParNewGC-XX:+UseSerialGC</code>	

5 . GC 参数列表集合 (如表 8-2 所示)

表 8-2 GC 参数列表集合

GC 方式	参 数 集 合
Heap 堆配置	<code>-Xms</code> /堆初始大小
	<code>-Xmx</code> /堆最大值
	<code>-Xmn</code> /Young 区大小
	<code>-XX:PermSize</code> /Perm 区大小
	<code>-XX:MaxPermSize</code> /Perm 区最大值

Serial Collector（串行）	-XX:+UseSerialGC/GC 方式	
续表		
GC 方式	参 数 集 合	
	-XX:SurvivorRatio/默认为 8，代表 eden:s0	
	-XX:MaxTenuringThreshold/默认为 15，代表对象在新生代经历多少次 MinorGC 后才晋升到 Old 区，效率高，当 Heap 过大时，应用程度暂停时间较长	
Parallel Collector（并行）	ParNewGC	-XX:+UseParNewGC/GC 方式
		-XX:SurvivorRatio/默认为 8，代表 eden:s0
		-XX:MaxTenuringThreshold/默认为 15
		-XX:+UseAdaptiveSizePolicy
Parallel Collector（并行）	ParallelGC	-XX:+UseParallelGC/GC 方式
		-XX:ParallelGCThreads/并发线程数
		-XX:InitialSurvivorRatio/默认为 8，Young:s0 的比值
		-XX:+UseAdaptiveSizePolicy
		-XX:MaxTenuringThreshold/默认为 15
	-XX:+ScavengeBeforeFullGC/FullGC 前触发 MinorGC	
	ParallelOldGC	-XX:+UseParallelOldGC/GC 方式，其他同上
CMS Collector（并发）	-XX:+UseConcMarkSweepGC/GC 方式	
	-XX:ParallelCMSThreads/设置并发 CMS GC 时的线程数	
	-XX:CMSInitiatingOccupancyFraction/当旧生代使用占到多少百分比时触发 CMS GC	
	-XX:+UseCMSInitiatingOccupancyOnly/默认为 false，代表允许 hotspot 根据成本来决定什么时候执行 CMSGC	
	-XX:+UseCMSCompactAtFullCollection/当 Full GC 时执行压缩	
	-XX:CMSMaxAbortablePrecleanTime=5000/设置 preclean 步骤的超时时间，单位为毫秒	
	-XX:+CMSClassUnloadingEnabled/PermGen 采用 CMS GC 回收	

6 . 三种 GC 优缺点对比 (如表 8-3 所示)

表 8-3 三种 GC 的优缺点对比

GC	优 点	缺 点
Serial Collector（串行）	在适合内存有限的情况下	回收慢
Parallel Collector（并行）	效率高	当 Heap 过大时，应用程序暂停时间较长

CMS Collector (并发)	Old 区回收暂停时间短	产生内存碎片、整个 GC 耗时较长、比较耗 CPU
----------------------	--------------	---------------------------

8.7 内存问题分析

8.7.1 GC 日志分析

有时候我们可能并不知道何时会发生内存溢出，但是当溢出已经发生时我们却并不知道原因，所以在 JVM 启动时就加上一些参数来控制，当 JVM 出问题能记下一些当时的情况。还有就是记录下来的 GC 的日志，我们可以观察 GC 的频度以及每次 GC 都回收了哪些内存。

GC 的日志输出如下参数。

- ⊙ -verbose:gc，可以辅助输出一些详细的 GC 信息。
- ⊙ -XX:+PrintGCDetails，输出 GC 的详细信息。
- ⊙ -XX:+PrintGCApplicationStoppedTime，输出 GC 造成应用程序暂停的时间。
- ⊙ -XX:+PrintGCDateStamps，GC 发生的时间信息。
- ⊙ -XX:+PrintHeapAtGC，在 GC 前后输出堆中各个区域的大小。
- ⊙ -Xloggc:[file]，将 GC 信息输出到单独的文件中。

每种 GC 的日志形式如表 8-4 所示。

表 8-4 每种 GC 的日志形式

GC 方式	日 志 形 式	
Serial Collector (串行)	[GC [DefNew: 11509K->1138K(14336K), 0.0110060 secs] 11509K->1138K(38912K), 0.0112610 secs] [Times: user=0.00 sys=0.01, real=0.01 secs]	
	[Full GC [Tenured: 9216K->4210K(10240K), 0.0066570 secs] 16584K->4210K(19456K), [Perm : 1692K->1692K(16384K)], 0.0067070 secs][Times: user=0.00 sys=0.00, real=0.01 secs]	
Parallel Collector (并行)	ParNewGC	[GC [ParNew: 11509K->1152K(14336K), 0.0129150 secs] 11509K->1152K(38912K), 0.0131890 secs] [Times: user=0.05 sys=0.02, real=0.02 secs]
		[GC [ASParNew: 7495K->120K(9216K), 0.0403410 secs] 7495K->7294K(19456K), 0.0406480 secs] [Times: user=0.06 sys=0.15, real=0.04 secs]

续表

GC 方式	日 志 形 式	
Parallel Collector (并行)	ParallelGC	[GC [PSYoungGen: 11509K->1184K(14336K)] 11509K->1184K(38912K), 0.0113360 secs][Times: user=0.03 sys=0.01, real=0.01 secs]
		[Full GC [PSYoungGen: 1208K->0K(8960K)] [PSOldGen: 6144K->7282K(10240K)] 7352K->7282K(19200K) [PSPermGen: 1686K->1686K(16384K)], 0.0165880 secs] [Times: user=0.01 sys=0.01, real=0.02 secs]
	ParallelOldGC	[Full GC [PSYoungGen: 1224K->0K(8960K)] [ParOldGen: 6144K->7282K(10240K)] 7368K->7282K(19200K) [PSPermGen: 1686K->1685K(16384K)], 0.0223510 secs] [Times: user=0.02 sys=0.06, real=0.03 secs]
CMS Collector (并发)	[GC [1 CMS-initial-mark: 13433K(20480K)] 14465K(29696K), 0.0001830 secs]	
	[Times: user=0.00 sys=0.00, real=0.00 secs]	
	[CMS-concurrent-mark: 0.004/0.004 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]	
	[CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]	
	CMS: abort preclean due to time [CMS-concurrent-abortable-preclean: 0.007/5.042 secs]	
	[Times: user=0.00 sys=0.00, real=5.04 secs]	
	[GC[YG occupancy: 3300 K (9216 K)][Rescan (parallel) , 0.0002740 secs]	
	[weak refs processing, 0.0000090 secs]	
	[1 CMS-remark: 13433K(20480K)] 16734K(29696K), 0.0003710 secs]	
	[Times: user=0.00 sys=0.00, real=0.00 secs]	
	[CMS-concurrent-sweep: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]	
	[CMS-concurrent-reset: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]	

除 CMS 的日志与其他 GC 的日志差别较大外，它们都可以抽象成如下格式：

```
[GC [<collector>: <starting occupancy1> -> <ending occupancy1>(total size1),
<pause time1> secs] <starting occupancy2> -> <ending occupancy2>(total size2),
<pause time2> secs]
```

其中说明如下：

◎ <collector>GC 表示收集器的名称。

- ⊙ <starting occupancy1>表示 Young 区在 GC 前占用的内存。
- ⊙ <ending occupancy1>表示 Young 区在 GC 后占用的内存。
- ⊙ <pause time1>表示 Young 区局部收集时 JVM 暂停处理的时间。
- ⊙ <starting occupancy2>表示 JVM Heap 在 GC 前占用的内存。
- ⊙ <ending occupancy2> 表示 JVM Heap 在 GC 后占用的内存。
- ⊙ <pause time2>表示在 GC 过程中 JVM 暂停处理的总时间。

可以根据日志来判断是否有内存在泄漏，如果 <ending occupancy1>-<starting occupancy1>=<ending occupancy2>-<starting occupancy2>，则表明这次 GC 对象 100% 被回收，没有对象进入 Old 区或者 Perm 区。如果等号前面的值大于等号后面的值，那么差值就是这次回收对象进入 Old 区或者 Perm 区的大小。如果随着时间的延长，<ending occupancy2>的值一直在增长，而且 Full GC 很频繁，那么很可能就是内存泄漏了。

除去日志文件分析外，还可以直接通过 JVM 自带的一些工具分析，如 jstat，使用格式为 jstat -gcutil [pid] [interval] [count]，如下面这个日志：

```
[junshan@tbskip027085 junshan]$ sudo /opt/taobao/java/bin/jstat -gcutil
29723 500 100
  S0    S1     E      O      P    YGC     YGCT    FGC     FGCT     GCT
  0.00   4.05  30.56  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  32.79  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  35.96  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  38.23  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  40.45  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  43.07  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  46.16  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  49.05  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  50.86  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  54.49  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  57.27  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  59.62  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  63.03  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  65.81  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  68.51  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  71.16  39.49  45.22  5401    68.689    66    22.519   91.209
  0.00   4.05  74.38  39.49  45.22  5401    68.689    66    22.519   91.209
```

0.00	4.05	76.94	39.49	45.22	5401	68.689	66	22.519	91.209
0.00	4.05	79.27	39.49	45.22	5401	68.689	66	22.519	91.209
0.00	4.05	82.74	39.49	45.22	5401	68.689	66	22.519	91.209
0.00	4.05	85.85	39.49	45.22	5401	68.689	66	22.519	91.209
0.00	4.05	88.35	39.49	45.22	5401	68.689	66	22.519	91.209
0.00	4.05	91.05	39.49	45.22	5401	68.689	66	22.519	91.209
0.00	4.05	93.45	39.49	45.22	5401	68.689	66	22.519	91.209

在上面日志中的参数含义如下：

- ⊙ S0 表示 Heap 上的 Survivor space 0 区已使用空间的百分比。
- ⊙ S1 表示 Heap 上的 Survivor space 1 区已使用空间的百分比。
- ⊙ E 表示 Heap 上的 Eden space 区已使用空间的百分比。
- ⊙ O 表示 Heap 上的 Old space 区已使用空间的百分比。
- ⊙ P 表示 Perm space 区已使用空间的百分比。
- ⊙ YGC 表示从应用程序启动到采样时发生 Young GC 的次数。
- ⊙ YGCT 表示从应用程序启动到采样时 Young GC 所用的时间（单位为秒）。
- ⊙ FGC 表示从应用程序启动到采样时发生 Full GC 的次数。
- ⊙ FGCT 表示从应用程序启动到采样时 Full GC 所用的时间（单位为秒）。
- ⊙ GCT 表示从应用程序启动到采样时用于垃圾回收的总时间（单位为秒）。

8.7.2 堆快照文件分析

可通过命令 `jmap -dump:format=b,file=[filename] [pid]` 来记录下堆的内存快照，然后利用第三方工具（如 `mat`）来分析整个 Heap 的对象关联情况。

如果内存耗尽那么可能导致 JVM 直接垮掉，可以通过参数：`-XX:+HeapDumpOnOutOfMemoryError` 来配置当内存耗尽时记录下内存快照，可以通过 `-XX:HeapDumpPath` 来指定文件的路径，这个文件的命名格式如 `java_[pid].hprof`。

8.7.3 JVM Crash 日志分析

JVM 有时也会因为一些原因而导致直接垮掉，因为 JVM 本身也是一个正在运行的程序，这个程序本身也会有很多情况直接出问题，如 JVM 本身也有一些 Bug，这些 Bug 可能会导致 JVM 异常退出。JVM 退出一般会在工作目录下产生一个日志文件，也可以通过 JVM 参数来设定，如-XX:ErrorFile=tmp/log/hs_error_%p.log。

下面是一个日志文件：

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
# SIGSEGV (0xb) at pc=0x00002ab12ba7103a, pid=7748, tid=1363515712
#
# JRE version: 6.0_26-b03
# Java VM: OpenJDK 64-Bit Server VM (20.0-b11-internal mixed mode linux-
amd64 )
# Problematic frame:
# V [libjvm.so+0x8bf03a] jni_GetFieldID+0x22a
#
# If you would like to submit a bug report, please visit:
# http://java.sun.com/webapps/bugreport/crash.jsp
#
----- T H R E A D -----
Current thread (0x00002aabd0ba5000):  JavaThread "http-0.0.0.0-7001-32"
daemon [_thread_in_vm, id=8192, stack(0x0000000051359000,0x000000005145a000)]

siginfo:si_signo=SIGSEGV: si_errno=0, si_code=1 (SEGV_MAPERR), si_addr=
0x0000000000000010

Registers:
...
Top of Stack: (sp=0x0000000051455620)
...
Instructions: (pc=0x00002ab12ba7103a)
0x00002ab12ba7101a:  01 00 00 48 8b 5f 10 48 8d 43 08 48 3b 47 18 0f
0x00002ab12ba7102a:  87 53 02 00 00 48 89 47 10 48 89 13 48 83 c2 10
0x00002ab12ba7103a:  48 8b 0a 48 89 d7 4c 89 fe 31 c0 ff 51 58 49 8b
0x00002ab12ba7104a:  47 08 48 85 c0 0f 85 f7 01 00 00 48 c7 45 90 00
Register to memory mapping:
```

```

RAX=
[error occurred during error reporting (printing register info), id 0xb]

Stack: [0x0000000051359000,0x000000005145a000], sp=0x0000000051455620,
free space=1009k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native
code)
V [libjvm.so+0x8bf03a] jni_GetFieldID+0x22a
C [libocijdbc10.so+0xcc81] Java_oracle_jdbc_driver_T2CConnection_
t2cDescribeError+0x205
C [libocijdbc10.so+0x878b] Java_oracle_jdbc_driver_T2CConnection_
t2cCreateState+0x193
j oracle.jdbc.driver.T2CConnection.t2cCreateState([BI[BI[BI[BISI[S[B[B]I+0
j oracle.jdbc.driver.T2CConnection.logon()V+589
j oracle.jdbc.driver.PhysicalConnection.<init>(Ljava/lang/String;Ljava/
lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/util/Properties;Loracl
e/jdbc/driver/OracleDriverExtension;)V+370
j oracle.jdbc.driver.T2CConnection.<init>(Ljava/lang/String;Ljava/lang/
String;Ljava/lang/String;Ljava/lang/String;Ljava/util/Properties;Loracle/jdb
c/driver/OracleDriverExtension;)V+10
j oracle.jdbc.driver.T2CDriverExtension.getConnection(Ljava/lang/String;
Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/util/Properties;
)Ljava/sql/Connection;+67
j oracle.jdbc.driver.OracleDriver.connect(Ljava/lang/String;Ljava/util/
Properties;)Ljava/sql/Connection;+831
...
----- P R O C E S S -----

Java Threads: ( => current thread )
0x000000004d11e800 JavaThread "IdleRemover" daemon [_thread_blocked,
id=8432, stack(0x00000000584ca000,0x00000000585cb000)]
=>0x00002aabd0ba5000 JavaThread "http-0.0.0.0-7001-32" daemon [_thread_in_
vm, id=8192, stack(0x0000000051359000,0x000000005145a000)]
...
VM state:not at safepoint (normal execution)

VM Mutex/Monitor currently owned by a thread: None

Heap
par new generation total 1474560K, used 1270275K [0x00002aaaae0f0000,

```

```

0x00002aab120f0000, 0x00002aab120f0000)
    eden space 1310720K,  89% used [0x00002aaaae0f0000, 0x00002aaaf5d634e0,
0x00002aaafe0f0000)
    from space 163840K,  57% used [0x00002aab080f0000, 0x00002aab0dcfd748,
0x00002aab120f0000)
    to   space 163840K,   0% used [0x00002aaafe0f0000, 0x00002aaafe0f0000,
0x00002aab080f0000)
    concurrent mark-sweep generation total 2555904K,  used  888664K
[0x00002aab120f0000, 0x00002aabae0f0000, 0x00002aabae0f0000)
    concurrent-mark-sweep perm gen total 262144K,  used  107933K
[0x00002aabae0f0000, 0x00002aabbe0f0000, 0x00002aabbe0f0000)

Code Cache [0x00002aaaab025000, 0x00002aaaabcd5000, 0x00002aaaae025000)
  total_blobs=3985  nmethods=3447  adapters=491  free_code_cache=37205440
largest_free_block=30336

Dynamic libraries:
40000000-40009000 r-xp 00000000 ca:06 224241                /opt/
xxx/install/jdk-1.6.0_26/bin/java
...
----- S Y S T E M -----

OS:Red Hat Enterprise Linux Server release 5.4 (Tikanga)

```

在这个文件中的信息主要分为 4 种：退出原因分类、导致退出的 Thread 信息、退出时的 Process 状态信息、退出时与操作系统相关信息。

JVM 退出一般有三种主要原因，在上面这个例子中是 SIGSEGV (0xb)，这三种原因分别如下。

1. EXCEPTION_ACCESS_VIOLATION

正在运行 JVM 自己的代码，而不是外部的 Java 代码或其他类库代码。这种情况很可能是 JVM 自己的 Bug，遇到这种错误时，可以根据出错信息到 <http://bugreport.sun.com/bugreport/index.jsp> 去搜索一下已经发行的 Bug。

在大部分情况下是由于 JVM 的内存回收导致的，所以可以观察 Process 部分的信息，查看堆的内存占用情况。

2 . SIGSEGV

JVM 正在执行本地或 JNI 的代码，出这种错误很可能是第三方的本地库有问题，可以通过 gbd 和 core 文件来分析出错原因。

3. EXCEPTION_STACK_OVERFLOW

这是个栈溢出的错误，注意 JVM 在执行 Java 线程时出现的栈溢出通常不会导致 JVM 退出，而是抛出 `java.lang.StackOverflowError`，但是在 Java 虚拟机中，Java 的代码和本地 C 或 C++ 代码共用相同的栈，这时如果出现栈溢出的话，就有可能直接导致 JVM 退出。建议将 JVM 的栈尺寸调大，主要涉及两个参数：`-Xss` 和 `-XX:StackShadowPages=n`。

日志文件的 `Thread` 部分的信息对我们排查这个问题的原因最有帮助，这部分有两个关系信息，包括 `Machine Instructions` 和 `Thread Stack`。`Machine Instructions` 是当前系统执行的机器指令，是 16 进制的。我们可以将它转成指令，通过 `udis86` 工具来转换，该工具可以在 <http://udis86.sourceforge.net/> 下载，安装在 Linux 中，将上面的 16 进制数字复制到命令行中用如下方式执行转换：

```
[junshan@xxx ~]$ echo "47 08 48 85 c0 0f 85 f7 01 00 00 48 c7 45 90 00" |
udcli -64 -x
0000000000000000 47084885      or [r8-0x7b], r9b
0000000000000004 c00f85      ror byte [rdi], 0x85
0000000000000007 f701000048c7    test dword [rcx], 0xc7480000
000000000000000d 4590      xchg r8d, eax
```

可以得到汇编指令，由于是 64 位机器，所以是 `udcli -64 -x`，如果是 32 机器，则改成 `udcli -32 -x`。可以通过这个指令来判断当前正在执行什么指令而导致了垮掉。例如，如果当前在访问寄存器地址，那么这个地址是否合法，以及如果是除法指令，操作数是否合法等。

而 `Stack` 信息最直接，可以帮助我们看到到底是哪个库的哪行代码出错，如在上面的错误信息中显示的是由于执行 Oracle 的 Java 驱动程序引起出错的。我们还可以通过生成的 `core` 文件来更详细地看出是执行到哪行代码出错的，如下所示：

```
$gdb /opt/xxx/java/bin/java /home/admin/xxxx/target/core.14595
GNU gdb Fedora (6.8-37.el5)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
...
```

```
(gdb) bt
#0  0x000000320ea30265 in raise () from /lib64/libc.so.6
#1  0x000000320ea31d10 in abort () from /lib64/libc.so.6
#2  0x00002b4ba59d80e9 in os::abort () from /opt/taobao/install/jdk-1.6.0_26/jre/lib/amd64/server/libjvm.so
#3  0x00002b4ba59d1e0f in VMError::report_and_die () from /opt/taobao/install/jdk-1.6.0_26/jre/lib/amd64/server/libjvm.so
...
```

通过 `gdb` 来调试 `core` 文件可以看到更详细的信息，还可以通过 `frame n` 和 `info local` 组合命令来更进一步地查看这一行所包含的 `local` 变量值，但这只能是程序使用 `-g` 命名编译的结果，也就是编译后的程序包含 `debug` 信息。

日志文件的第三部分包含的是 **Process** 信息，这里详细列出了该程序产生的所有线程，以及线程正处于的状态。由于在同一时刻只能有一个线程具有 **CPU** 使用权，所以可以看到，其他所有线程的状态都是 `_thread_blocked`，而执行的正是那个出错的线程。

这部分最有价值的部分就是记录下来了当前 **JVM** 的堆信息，如下所示：

```
Heap
  par new generation   total 1474560K, used 1270275K [0x00002aaaae0f0000,
0x00002aab120f0000, 0x00002aab120f0000)
    eden space 1310720K,  89% used [0x00002aaaae0f0000, 0x00002aaaf5d634e0,
0x00002aaafe0f0000)
    from space 163840K,   57% used [0x00002aab080f0000, 0x00002aab0dcfd748,
0x00002aab120f0000)
    to   space 163840K,    0% used [0x00002aaafe0f0000, 0x00002aaafe0f0000,
0x00002aab080f0000)
  concurrent mark-sweep generation total 2555904K,  used  888664K
[0x00002aab120f0000, 0x00002aabae0f0000, 0x00002aabae0f0000)
  concurrent-mark-sweep perm gen total 262144K,  used  107933K
[0x00002aabae0f0000, 0x00002aabbe0f0000, 0x00002aabbe0f0000]
```

通过每个分区当前所使用的空间大小，尤其是 **Old** 区的空间是否已经满了，可以判断出当前的 **GC** 是否正常。

还有一个信息也比较有价值，那就是当前 **JVM** 的启动参数，设置的堆大小和使用的 **GC** 方式等都可以从这里看出。

最后一部分是 **System** 信息，这部分主要记录了当前操作系统的状态信息，如操作系统的 **CPU** 信息和内存情况等。

8.8 实例 1

这里有一个 JVM 内存泄漏的实例，是在淘宝的一个系统中发生的，这个问题是由一个模板引擎导致的，这个模板引擎在后面的章节中再介绍。

当时的情况是系统 load 偏高，达到了 6 左右，而平时基本在 1 左右，整个系统响应较慢，但是重启系统之后就恢复正常。于是查看 GC 的情况，发现 FGC 明显超出正常情况，并且在 GC 过程中出现 concurrent mode failure。如下日志所示：

```
[GC 642473.656: [ParNew: 2184576K->2184576K(2403008K), 0.0000280 secs]
642473.656: [CMS2011-09-30T03:53:22.209+0800: 642473.656: [CMS-concurrent-
abortable-preclean: 0.064/1.953 secs] [Times: user=0.06 sys=0.00, real=1.95
secs]
(concurrent mode failure): 1408475K->1422713K(1572864K), 6.0568470 secs]
3593051K->1422713K(3975872K), [CMS Perm : 77027K->77005K(200704K)], 6.0570590
secs] [Times: user=6.05 sys=0.00, real=6.06 secs]
```

这说明在 Old 区分配内存时出现分配失败的现象，而且整个内存占用达到了 6GB 左右，超出了平时的 4GB，于是得出可能是有内存泄漏的问题。

通过 `jmap -dump:format=b,file=[filename] [pid]` 命令查看 Heap，再通过 MAT 工具分析，如图 8-4 所示。

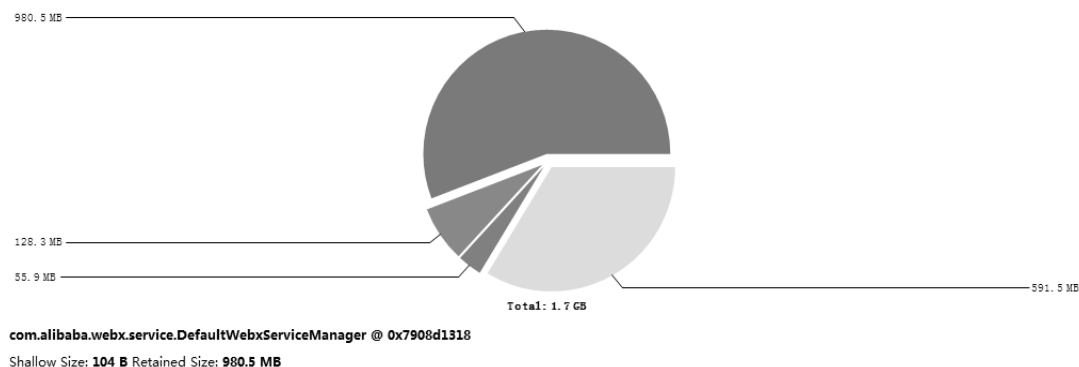


图 8-4 MAT 工具分析

图 8-4 中最大的一个对象占用了 900 多兆内存，这显然有问题。图 8-5 是 MAT 给出的

可能有问题的对象的说明，指出了 Map 集合占用了 55%的空间。

▼ ● Problem Suspect 1

One instance of "com.alibaba.webx.service.DefaultWebxServiceManager" loaded by "org.jboss.mx.loading.UnifiedClassLoader3 @ 0x7909290a0" occupies 1,028,119,560 (55.83%) bytes. The memory is accumulated in one instance of "java.util.concurrent.ConcurrentHashMap\$Segment[]" loaded by "<system class loader>".**Keywords**
java.util.concurrent.ConcurrentHashMap\$Segment[]
org.jboss.mx.loading.UnifiedClassLoader3 @ 0x7909290a0
com.alibaba.webx.service.DefaultWebxServiceManager

Details >

图 8-5 MAT 对可能有问题的对象的说明

再看一下到底这个对象都持有了哪些对象，如图 8-6 所示。

Class Name	Shallow Heap	Retained Heap
com.taobao.sketch.compile.SketchCompilationContext @ 0x79aeb8ea0	264	331,952,584
<class> class com.taobao.sketch.compile.SketchCompilationContext @ 0x7f2d53ff	0	0
log com.alibaba.common.logging.spi.log4j.Log4jLogger @ 0x79008bc40	24	24
sketchConfig com.taobao.sketch.runtime.SketchRuntimeServer @ 0x7908f0930	112	1,027,674,056
classLoader com.taobao.sketch.compile.resource.SketchTemplateJavaLoader @ 0x7908f0930	152	2,528
sketchTemplateInstance com.taobao.sketch.runtime.SketchTemplateInstance @ 0x7908f0930	104	332,231,488
staticText java.util.HashMap @ 0x79bcaa528	64	648
reference java.util.HashMap @ 0x79bcaa558	64	331,308,544
<class> class java.util.HashMap @ 0x7f00b9080 System Class	24	24
entrySet java.util.HashMap\$EntrySet @ 0x79c1a9500	24	24
table java.util.HashMap\$Entry[32] @ 0x79c1e68c8	280	331,308,456
<class> class java.util.HashMap\$Entry[] @ 0x7f00bd628	0	0
[1] java.util.HashMap\$Entry @ 0x79c2fac40	48	752
[2] java.util.HashMap\$Entry @ 0x79c2fac60	48	760
<class> class java.util.HashMap\$Entry @ 0x7f00bd060 System Class	0	0
key java.lang.String @ 0x79c2fb240 _Jasq\$remove	40	88
value com.taobao.sketch.runtime.RefClass @ 0x79de29ae0	80	624
<class> class com.taobao.sketch.runtime.RefClass @ 0x7f3a44eb0	0	0
obj com.taobao.hesper.biz.core.query.AuctionSearchQuery @ 0x7982f0930	200	166,014,544
<class> class com.taobao.hesper.biz.core.query.AuctionSearchQuery @ 0x7982f0930	152	28,048
<class> class java.lang.Class @ 0x7f0021390 System Class, Native	48	72
<classloader> org.jboss.mx.loading.UnifiedClassLoader3 @ 0x7909290a0	216	3,168,520
catsForHateDaily java.lang.Integer[4] @ 0x792cf2880	56	56
personalizedCatsDaily java.util.HashSet @ 0x797ffbfe8	24	288
catsForPersonalizedDaily java.lang.Integer[1] @ 0x797ffc000	32	32
validIndex java.util.HashSet @ 0x797ffc648	24	576
AUCTION_ALLOW_PARAMS java.util.HashSet @ 0x797ffc660	24	19,136

图 8-6 对象大小

在图 8-6 中第一列是类名，第二列 Shallow Heap 表示这个对象本身的大小，所谓对象本身大小就是在这个对象的一些域中直接分配的存储空间，如定义 byte[] byte=new byte[1024]，这个 byte 属性的大小就直接包含在这个对象的 Shallow 中。而如果这个对象的某些属性指向一个对象，那么所指向的那个对象的大小就计算在 Retained Heap 中。

图 8-6 中 `SketchCompileContext` 对象持有一个 `Map` 集合，这个 `Map` 集合所占用的空间很大，仔细查看后发现这个 `Map` 持有一个 `DO` 对象，这个对象的确是一个大对象，它的大小并没有超出我们的预期，仔细查看其他集合，没有发现所持有对象有什么不对的地方。但是仔细计算整个对象集合的大小发现，虽然所有的对象都是应该存在的，但是比我们计算的正常大小多了将近一倍，于是我们想到可能是持有了两份同样的对象。

按照这个思路仔细搜索这个 `Map` 集合中的对象的数值，果然发现同样一个数值有两个不同的 `DO` 对象对应，但是为什么有两个 `DO` 对象呢？正常情况应该单一啊，怎么会产生两份 `DO` 对象？后面仔细检查这个 `DO` 对象的业务逻辑，原来是这个 `DO` 要在每天凌晨两点更新一次，更新后老对象会自动释放，但是我们这个新引擎是要保存这些对象，以便于做编译优化，不能及时地释放这个更新后的老对象，所以导致这个大对象在内存中保存了两份。

8.9 实例 2

这个例子和前面介绍的 CMS GC 回收方式的一个 JVM 的 Bug 相关，淘宝的某应用在某天突然导致线上部分机器报警，Java 的内存使用非常严重，达到了 6GB，超过了平时的 4GB，而且有几台机器进行一段时间后导致 OOM、JVM 退出。当时相关人员的第一反应是重启部分机器，保留几台有问题的机器来寻找原因。

观察重启后的机器，发现应用恢复正常，但是发现 JVM 进程占用的内存一直在增长，可以大体推断出是 JVM 有内存泄漏。然后检查最近是否有系统改动，是否是 Java 代码问题导致了内存泄漏。检查后发现最近一周 Java 代码改动很少，而且也没有发现有内存泄漏问题。

同时检查 GC 的日志，发现有问题的机器的 Full GC 没有异常，甚至 Full GC 比平时还少，CMS GC 也很少。从日志中没有发现可能有内存问题。

为了进一步确认 GC 是否正常，我们找出 JVM 的 Heap，用 MAT 分析堆文件，堆的使用情况如图 8-7 所示。

可以看出，整个 Heap 只有不到 1GB 的空间，而且从 Leak Suspects 给出的报告中可以看到占有最大空间的对象是一个 `DO` 对象，如图 8-8 所示。

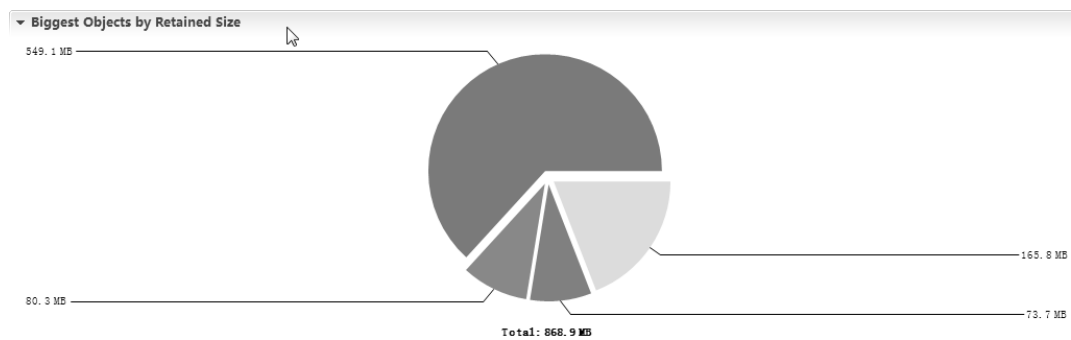


图 8-7 堆使用情况

▼ ⓘ Problem Suspect 1

One instance of "com.taobao forest.domain.dataobject.ForestDO" loaded by "org.jboss.mx.loading.UnifiedClassLoader3 @ 0x2aab12125080" occupies 575,820,464 (63.20%) bytes. The memory is accumulated in one instance of "com.taobao forest.domain.dataobject.ForestDO" loaded by "org.jboss.mx.loading.UnifiedClassLoader3 @ 0x2aab12125080". Keywords com.taobao forest.domain.dataobject.ForestDO org.jboss.mx.loading.UnifiedClassLoader3 @ 0x2aab12125080

[Details >](#)

图 8-8 DO 对象

而这个对象的大小也符合我们的预期，所以可以得出判断，不是 JVM 的堆内存有问题。但是既然 JVM 的堆占有的内存并不多，那么 Java 进程为什么占用那么多内存呢？

我们于是想到了可能是堆外分配的内存有泄漏，从前面的分析中我们已经知道，JVM 除了堆需要内存外还有很多方面也需要在运行时使用内存，如 JVM 本身 JIT 编译需要内存，JVM 的栈也需要内存，JNI 调用本地代码也需要内存，还有 NIO 方式也会使用 Direct Buffer 来申请内存。

从这些因素中我们推断可能是 Direct Buffer 导致的，因为在上次发布中引入过一个 Apache 的 Mina 包，在这个包中肯定使用了 Direct Buffer，但是为什么 Direct Buffer 没有正常回收呢？很奇怪。

这时想到了可能是 JVM 的一个 Bug，详见 http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6919638。外部调用了 System.gc，且设置了 -XX:+DisableExplicitGC，所以导致 System.gc()变成了空调用，而应用 GC 却很少，这里的 GC 包括 CMS GC 和 Full GC。所以 Direct Buffer 对象还没有被及时回收，相应的 native memory 不能被释放。为了验证这一点，相关人员还写了一个工具来检查当前 JVM 中 NIO direct memory 的使用情况，如下

所示：

```
[sajia@xxx ~]$ sjdirectmem `s pgrep java`
Attaching to process ID 3543, please wait...
WARNING: Hotspot VM version 20.0-b11-internal does not match with SA version
20.1-b02. You may see unexpected results.
Debugger attached successfully.
Server compiler detected.
JVM version is 20.0-b11-internal
NIO direct memory: (in bytes)
  reserved size = 163.417320 MB (171355480 bytes)
  max size      = 3936.000000 MB (4127195136 bytes)
[sajia@xxx ~]$ sjdirectmem 25332
Attaching to process ID 25332, please wait...
WARNING: Hotspot VM version 20.0-b11-internal does not match with SA version
20.1-b02. You may see unexpected results.
Debugger attached successfully.
Server compiler detected.
JVM version is 20.0-b11-internal
NIO direct memory: (in bytes)
  reserved size = 1953.929705 MB (2048843794 bytes)
  max size      = 3936.000000 MB (4127195136 bytes)
```

可以看出一段时间后 NIO direct memory 的确增长了很多，所以可以肯定是 NIO direct memory 没有释放从而导致 Java 进程占用的内存持续增长。

这个问题的解决办法是去掉 `-XX:+DisableExplicitGC`，换上 `-XX:+ExplicitGCInvokesConcurrent`，使得外部的显示 `System.gc` 调用生效，这样即使 Java GC 不频繁时也可以通过外部主动调用 `System.gc` 来回收 NIO direct memory 分配的内存。

8.10 实例 3

下面再介绍一个 NIO direct memory 发生内存泄漏的例子。

JVM 的配置参数如下：

```
-server -Xms2024m -Xmx2024m -XX:NewSize=320m -XX:MaxNewSize=320m -XX:
PermSize=96m -XX:MaxPermSize=256m -Djava.awt.headless=true -Xdebug -Xrunjdpw:
```

```
transport=dt_socket,server=y,suspend=n,address=8001
-Djava.net.preferIPv4Stack=true -Dsun.net.client.defaultConnectTimeout=10000
-Dsun.net.client.defaultReadTimeout=30000 -Djava.awt.headless=true -Dcom.sun.
management.jmxremote.port=1090 -Dcom.sun.management.jmxremote.ssl=false -Dcom.
sun.management.jmxremote.authenticate=false -Djava.rmi.server.hostname=v101208.
sqa.cm4 -XX:+UseCompressedOops
```

问题表现如下所述。

(1) 一个系统在运行 20~30 分钟后, 系统 swap 区突增, 直到 swap 区使用率达到 100%, 机器死机, 如图 8-9 所示。

---total-cpu-usage---					-dsk/total-			---load-avg---			-----memory-usage-----				-net/total-		-swp/total-	
usr	sys	idl	wai	hiq	siq	read	writ	1m	5m	15m	used	buff	cach	free	recv	send	used	free
3	13	85	0	0	0	0	0	0	0.1	0.1	3358M	37M	433M	12M	32k	42k	17M	1010M
4	12	85	0	0	0	0	448k	0	0.1	0.1	3385M	29M	414M	12M	41k	48k	17M	1010M
2	13	84	0	0	0	0	1232k	0	0.1	0.1	3414M	23M	391M	12M	41k	48k	17M	1010M
3	14	83	0	0	1	0	0	0	0.1	0.1	3444M	22M	362M	12M	33k	43k	17M	1010M
2	21	77	0	0	0	0	56k	0	0.1	0.1	3474M	22M	333M	11M	38k	46k	17M	1010M
3	18	79	0	0	0	0	0	0	0.1	0.1	3505M	22M	303M	10M	36k	49k	17M	1010M
16	18	66	0	0	0	0	72k	0	0.1	0.1	3542M	22M	266M	11M	36k	38k	17M	1010M
39	44	17	0	0	0	0	1128k	0	0.1	0.1	3614M	16M	198M	12M	19k	41k	17M	1010M
3	21	74	1	0	0	0	0	0	0.1	0.1	3643M	9264k	176M	12M	34k	37k	17M	1010M
4	17	79	0	0	0	0	744k	0	0.1	0.1	3672M	8036k	148M	12M	45k	59k	17M	1010M
3	12	85	0	0	0	0	0	0	0.1	0.1	3700M	6144k	122M	12M	37k	39k	17M	1010M
3	10	87	0	0	0	8192B	32k	0	0.1	0.1	3728M	3612k	97M	12M	43k	58k	17M	1010M
5	9	87	0	0	0	0	1352k	0	0.1	0.1	3759M	3592k	67M	11M	39k	49k	17M	1010M
3	7	89	0	0	0	0	8192B	0	0.1	0.1	3786M	548k	42M	12M	39k	47k	18M	1010M
4	10	87	0	0	0	0	59M	0	0.1	0.1	3789M	140k	39M	12M	46k	54k	47M	980M
3	13	81	4	0	0	464k	62M	0	0.1	0.1	3790M	152k	39M	12M	41k	50k	78M	950M
2	12	86	0	0	0	0	60M	0	0.1	0.1	3790M	160k	39M	11M	32k	40k	108M	920M
4	11	85	0	0	0	0	63M	0	0.1	0.1	3790M	148k	39M	12M	40k	47k	139M	888M
7	14	73	6	0	0	4088k	69M	0	0.1	0.1	3791M	140k	38M	9.9M	36k	46k	174M	854M
33	13	3	52	0	0	0	137M	0	0.1	0.1	3789M	144k	39M	12M	13k	21k	248M	780M

图 8-9 系统资源使用情况图

(2) 系统内存已经达到 3.5GB, 已经超过了 Heap 堆设置的上线, 但是 GC 却很少, 如图 8-10 所示。

S0	S1	E	O	P	YGC	YGCT	FGC	FGCT	GCT
0.18	0.00	84.86	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	85.58	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	86.95	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	86.99	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	88.36	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	89.70	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	90.43	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	91.16	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	92.53	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	92.56	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	93.93	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	94.62	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	95.99	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	97.36	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	97.40	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	98.77	24.86	63.86	16	5.618	1	0.124	5.742
0.18	0.00	99.47	24.86	63.86	16	5.618	1	0.124	5.742

图 8-10 GC 统计信息

(3) Old 区的空间也几乎没有变化, 如图 8-11 所示。

首先 Java 进程内存增长非常迅速, 进行压力测试 20 分钟后就将 2GB 内存用光, 并且

将内存耗光后开始使用 swap 区，很快消耗了 swap 区的空间，最终导致机器死机，所以可以肯定发生了内存泄漏。

```
[junshan@localhost ~]$ sudo jstat -gcold 4158 1000
```

PC	PU	OC	OU	YGC	FGC	FGCT	GCT
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.4	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.8	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.8	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.8	1744896.0	433998.7	20	1	0.124	5.813
98304.0	62785.8	1744896.0	433998.7	20	1	0.124	5.813

图 8-11 JVM 堆统计信息

但是通过 jstat 分析 JVM Heap 堆情况和 GC 统计信息，发现 GC 很少，尤其是 Full GC 几乎没有，如果是 JVM 堆内存被耗光，Full GC 应该非常频繁，所以初步判断这次内存泄漏不在 JVM 堆中。

但是为了进一步排除是 JVM 堆的内存问题，通过 jmap dump 出内存快照，通过 MAT 分析内存数据占用情况，如图 8-12 所示。

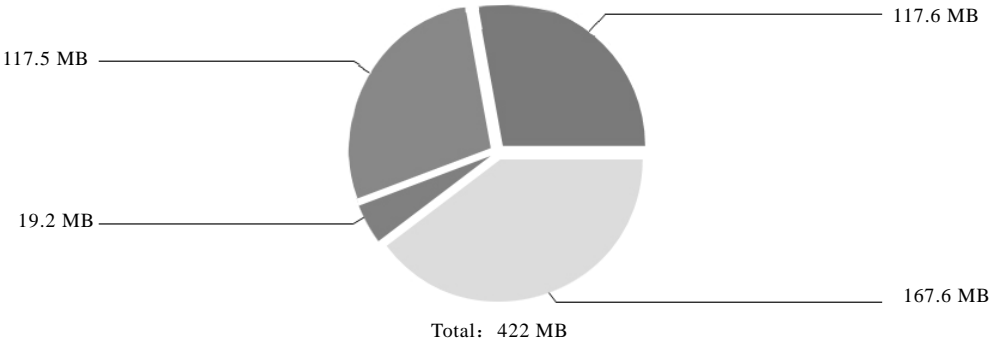


图 8-12 MAT 分析结果

这个堆只使用了近 500MB 内存，和 jstat 得出的堆信息是一致的，而两个最大的对象内容如图 8-13 所示。

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
org.apache.mina.transport.socket.nio.SocketSessionImpl @ 0x75b340588	296	123,340,192
<class> class org.apache.mina.transport.socket.nio.SocketSessionImpl	0	0
ch.sun.nio.ch.SocketChannelImpl @ 0x75b3404f0	176	448
key sun.nio.ch.SelectionKeyImpl @ 0x75b340560	64	64
writeRequestQueue java.util.concurrent.ConcurrentLinkedQueue @ 0x75b3405e0	32	123,336,840
<class> class java.util.concurrent.ConcurrentLinkedQueue @ 0x75b3405e0	40	40
tail java.util.concurrent.ConcurrentLinkedQueue\$Node @ 0x72500f0	32	192
head java.util.concurrent.ConcurrentLinkedQueue\$Node @ 0x75e2400	32	123,336,616
<class> class java.util.concurrent.ConcurrentLinkedQueue\$Node	24	24
item org.apache.mina.common.io.Filter\$WriteRequest @ 0x75e2400	40	224
next java.util.concurrent.ConcurrentLinkedQueue\$Node @ 0x75e2400	32	123,336,360
<class> class java.util.concurrent.ConcurrentLinkedQueue\$Node	24	24
item org.apache.mina.common.io.Filter\$WriteRequest @ 0x75e2400	40	280
next java.util.concurrent.ConcurrentLinkedQueue\$Node @ 0x75e2400	32	123,336,048
Σ Total: 3 entries		
Σ Total: 3 entries		
Σ Total: 3 entries		

图 8-13 MAT 分析的堆中的对象

两个都是 `org.apache.mina.transport.socket.nio.SocketSessionImpl` 对象，由于占用的空间不大（100MB）、持有的对象数也不多，没有引起注意。

于是可以得出：要么是 NIO direct memory，要么就是 native memory 泄漏。使用查看 NIO direct memory 的工具检查 direct memory 占用的空间大小，如图 8-14 所示。

```
[junshan@localhost ~]$ java -Xmx4G -Xms4G -XX:DirectMemorySize=4158 \
Attaching to process ID 4158, please wait...
WARNING: Hotspot VM version 20.0-b11-internal does not match with SA version 20.1-b02. You may see unexpected results.
Debugger attached successfully.
Server compiler detected.
JVM version is 20.0-b11-internal
NO direct memory:
  reserved size = 25.9840024MB (27246224 bytes)
  max size = 1984.000000MB (2080374784 bytes)
```

图 8-14 direct memory 统计信息

显示只有 25MB 左右，在这当中还使用了 gcore 命令 dump 出 java 进程的 core 文件，然后通过 jmap 将 core dump 转化成 Heap dump，转化后的 Heap dump 文件和前面的类似。另外通过 jstack dump 出内存也没有发现有线程堵塞情况，所以怀疑是 native memory 出现了泄漏，于是开始往这个方向考虑。

想使用 Oprofiler 热点分析工具分析当前系统执行的热点代码，如果是当前的 native memory 有泄漏，那么肯定会出现分配内存的代码是热点的情况，用 Oprofiler 分析的 CPU 的消耗情况如图 8-15 所示。

如图 8-15 所示，和预想的情况并不吻合，一时找不到更好的办法，于是使用土办法，一部分、一部分地去掉功能模块，看看到底是哪个模块导致的内存泄漏，进一步缩小范围。

```

CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
samples  % app name symbol name
365070 98.4528 no-vmlinux /no-vmlinux
1688 0.4552 anon (tgid:15394 range:0x2aaaaec01000-0x2aaaaef92000) anon (tgid:15394 range:0x2aaaaec01000-0x2aaaaef92000)
305 0.0823 libperl.so /usr/lib64/perl5/5.8.8/x86_64-linux-thread-multi/CORE/libperl.so
240 0.0647 libzip.so inflate_fast
121 0.0326 libpthread-2.5.so __write_nocancel
105 0.0283 anon (tgid:15394 range:0x2aaaaef92000-0x2aaab1c02000) anon (tgid:15394 range:0x2aaaaef92000-0x2aaab1c02000)
102 0.0275 libzip.so crc32
92 0.0248 libzip.so huft_build
89 0.0240 libc-2.5.so _int_malloc
64 0.0173 libjvm.so
CardTableModRefBS::dirty_card_range_after_reset(MemRegion, bool, int)
63 0.0170 libjvm.so IndexSetIterator::advance_and_next()
52 0.0140 libjvm.so PhaseChaitin::Split(unsigned int)
41 0.0111 static-python /home/tops/bin/static-python
40 0.0108 libjvm.so PhaseChaitin::build_ifg_physical(ResourceArea*)
40 0.0108 libpthread-2.5.so pthread_getspecific
37 0.0100 libjvm.so Copy::fill_to_memory_atomic(void*, unsigned long, unsigned char)
35 0.0094 libjvm.so jni_GetIntField
34 0.0092 libc-2.5.so memcpy
34 0.0092 libjvm.so SymbolTable::lookup(int, char const*, int, unsigned int)
32 0.0086 libjvm.so PhaseChaitin::gather_lrg_masks(bool)

```

图 8-15 Oprofiler 分析结果

通过删除可能会出问题的几个模块后，最后确定是调用 mina 框架给 varnish 发送失效请求时导致的，而且发送的请求数频率越高内存泄漏越严重，但是 mina 框架没有使用 native memory 的地方，于是又陷入僵局。

使用 Perftools 来分析 JVM 的 native Memory 分配情况，通过 Perftools 得到的分析结果如图 8-16 所示。

```

[junshan@v024085 ~]$ cat pfl.txt | sort -n -r -k4 | more
2682.1 99.0% 99.0% 2682.1 99.0% os::malloc
0.0 0.0% 100.0% 2657.1 98.1% Unsafe_AllocateMemory
0.0 0.0% 100.0% 2656.9 98.1% 0x00002aaaaec3266
0.0 0.0% 100.0% 2656.8 98.1% 0x00002aaaaefdfb77
18.3 0.7% 99.7% 18.3 0.7% zcalloc
0.0 0.0% 100.0% 17.8 0.7% javaMain
0.0 0.0% 100.0% 17.7 0.7% Threads::create_vm
0.0 0.0% 100.0% 17.7 0.7% JNI_CreateJavaVM
0.0 0.0% 100.0% 17.6 0.6% init_globals
0.0 0.0% 100.0% 17.4 0.6% universe_init
0.0 0.0% 100.0% 17.2 0.6% universe::initialize_heap
0.0 0.0% 100.0% 17.2 0.6% GenCollectedHeap::initialize
0.0 0.0% 100.0% 11.5 0.4% CMSCollector::CMSCollector
0.0 0.0% 100.0% 5.3 0.2% GenerationSpec::init
0.0 0.0% 100.0% 5.0 0.2% ParNewGeneration::ParNewGeneration
0.0 0.0% 100.0% 4.2 0.2% Hashtable::new_entry
0.0 0.0% 100.0% 4.2 0.2% BasicHashtable::new_entry
3.3 0.1% 99.8% 3.3 0.1% apr_palloc

```

图 8-16 Perftools 分析结果

图 8-16 显示内存的分配和使用都是在操作系统中，没有发现和应用代码相关的情况，也排除了已知的误用 Inflater/Deflater 的 native memory 的问题。还是没有找到问题所在！

于是又回到 Java 代码，这时发现在代码中使用 org.apache.mina.filter.codec.textline.TextLineEncoder 类来发送和序列化发送的数据，并且这个类使用的是 direct memory 内存：

```

ByteBuffer buf = ByteBuffer.allocate(value.length()).setAutoExpand(true);

```

将这个类的代码改成使用 JVM Heap 来存放数据：

```
ByteBuffer.allocate(value.length(),false);
```

按照这个思路，也就是将可能发生的 direct memory 转变成 Heap 堆内存泄漏，如果真是这个代码有问题，必然会导致 JVM Heap 暴涨，这样我们也能通过 MAT 来分析 JVM 堆中的对象情况。

修改代码后再运行，果不其然，当达到 JVM 堆配置的上限时，GC 非常频繁，使用 MAT 分析 dump 下来的堆，如图 8-17 所示。

这时显示堆空间都被 SocketSessionImpl 的 writeRequestQueue 队列持有，这个队列是 mina 的写队列，也就是 mina 不能及时地将数据发送出去，导致数据都堵在了这个队列中，进而导致了内存泄漏。所以根据这个分析认为，还是使用 mina 导致了 direct memory 泄漏。

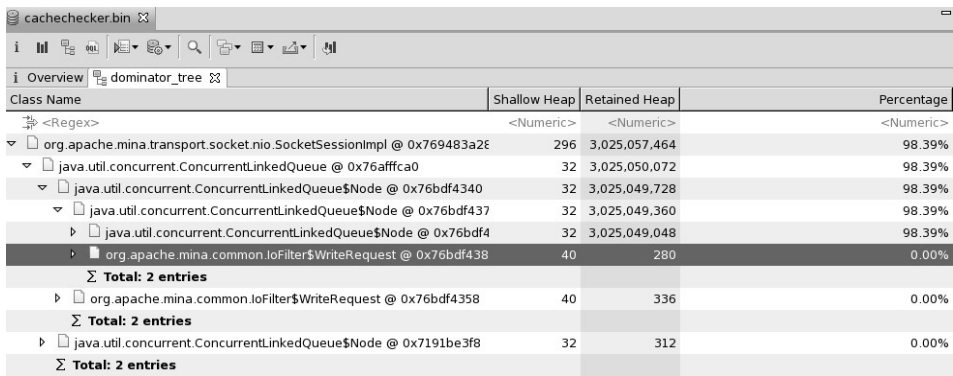


图 8-17 MAT 第二次分析结果

8.11 总结

本章介绍了 JVM 的内存结构、JVM 的内存分配策略、JVM 的内存回收策略及常见的内存问题，最后列举 3 个在实际使用中遇到的 JVM 内存泄漏的例子，并介绍了排查这些问题的方法。

第 9 章

Servlet 工作原理解析

Java Web 技术是当今主流的互联网 Web 应用技术之一，而 Servlet 是 Java Web 技术的核心基础。因而掌握 Servlet 的工作原理是成为一名合格的 Java Web 技术人员的基本要求。本章将带你认识 Java Web 技术是如何基于 Servlet 工作的，你将知道：Servlet 容器是如何工作的（以 Tomcat 为例）；一个 Web 工程在 Servlet 容器中是如何启动的；Servlet 容器如何解析你在 web.xml 中定义的 Servlet；用户的请求是如何被分配给指定的 Servlet 的；Servlet 容器如何管理 Servlet 生命周期。你还将了解到最新的 Servlet 的 API 类层次结构，以及如何分析 Servlet 中的一些难点问题。

9.1 从 Servlet 容器说起

要介绍 Servlet 必须先把 Servlet 容器说清楚，Servlet 与 Servlet 容器的关系有点像枪和子弹的关系，枪是为子弹而生的，而子弹又让枪有了杀伤力。虽然它们是彼此依存的，但是又相互独立发展，这一切都是为了适应工业化生产。从技术角度来说是为了解耦，通过标准化接口来相互协作。既然接口是连接 Servlet 与 Servlet 容器的关键，那我们就从它们的接口说起。

Servlet 容器作为一个独立发展的标准化产品，目前其种类很多，但是它们都有自己的市场定位，各有特点，很难说谁优谁劣。例如，现在比较流行的 Jetty，在定制化和移动领域有不错的发展。我们这里还是以大家最为熟悉的 Tomcat 为例来介绍 Servlet 容器是如何管理 Servlet 的。Tomcat 本身也很复杂，我们从 Servlet 与 Servlet 容器的接口部分开始介绍，关于 Tomcat 的详细介绍可以参考本书相关章节。

在 Tomcat 的容器等级中，Context 容器直接管理 Servlet 在容器中的包装类 Wrapper，所以 Context 容器如何运行将直接影响 Servlet 的工作方式。Tomcat 容器模型如图 9-1 所示。

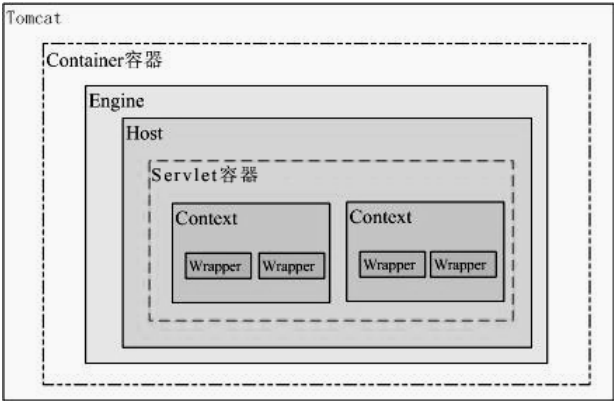


图 9-1 Tomcat 容器模型

从图 9-1 可以看出，Tomcat 的容器分为 4 个等级，真正管理 Servlet 的容器是 Context 容器，一个 Context 对应一个 Web 工程，在 Tomcat 的配置文件中可以很容易地发现这一点，如下所示：

```
<Context path="/projectOne " docBase="D:\projects\projectOne" reloadable="true" />
```

下面详细介绍 Tomcat 解析 Context 容器的过程，包括如何构建 Servlet。

9.1.1 Servlet 容器的启动过程

Tomcat 7 也开始支持嵌入式功能，增加了一个启动类 org.apache.catalina.startup.Tomcat。创建一个实例对象并调用 start 方法就可以很容易地启动 Tomcat。我们还可以通过这个对象来增加和修改 Tomcat 的配置参数，如可以动态增加 Context、Servlet 等。下面我们就利用这个 Tomcat 类来管理一个新增的 Context 容器，选择 Tomcat 7 自带的 examples

Web 工程，并看看它是如何加到这个 Context 容器中的。

```
Tomcat tomcat = getTomcatInstance();
File appDir = new File(getBuildDirectory(), "webapps/examples");
tomcat.addWebapp(null, "/examples", appDir.getAbsolutePath());
tomcat.start();
ByteChunk res = getUrl("http://localhost:" + getPort() +
    "/examples/servlets/servlet/HelloWorldExample");
assertTrue(res.toString().indexOf("<h1>Hello World!</h1>") > 0);
```

这段代码创建了一个 Tomcat 实例并新增了一个 Web 应用，然后启动 Tomcat 并调用其中的一个 HelloWorldExample Servlet，看看有没有正确返回预期的数据。

Tomcat 的 addWebapp 方法的代码如下：

```
public Context addWebapp(Host host, String url, String path) {
    silence(url);
    Context ctx = new StandardContext();
    ctx.setPath( url );
    ctx.setDocBase(path);
    if (defaultRealm == null) {
        initSimpleAuth();
    }
    ctx.setRealm(defaultRealm);
    ctx.addLifecycleListener(new DefaultWebXmlListener());
    ContextConfig ctxCfg = new ContextConfig();
    ctx.addLifecycleListener(ctxCfg);
    ctxCfg.setDefaultWebXml("org/apache/catalin/startup/NO_DEFAULT_XML");
    if (host == null) {
        getHost().addChild(ctx);
    } else {
        host.addChild(ctx);
    }
    return ctx;
}
```

前面已经介绍了一个 Web 应用对应一个 Context 容器，也就是 Servlet 运行时的 Servlet 容器。添加一个 Web 应用时将会创建一个 StandardContext 容器，并且给这个 Context 容器设置必要的参数，url 和 path 分别代表这个应用在 Tomcat 中的访问路径和这个应用实际的物理路径，这两个参数与 Tomcat 配置中的两个参数是一致的。其中最重要的一个配置是 ContextConfig，这个类将会负责整个 Web 应用配置的解析工作，后面将会对其进行详细介绍。

绍。最后将这个 Context 容器加到父容器 Host 中。

接下来将会调用 Tomcat 的 start 方法启动 Tomcat。如果你清楚 Tomcat 的系统架构，那么会很容易理解 Tomcat 的启动逻辑。Tomcat 的启动逻辑是基于观察者模式设计的，所有的容器都会继承 Lifecycle 接口，它管理着容器的整个生命周期，所有容器的修改和状态的改变都会由它去通知已经注册的观察者（Listener）。Tomcat 启动的时序图如图 9-2 表示。

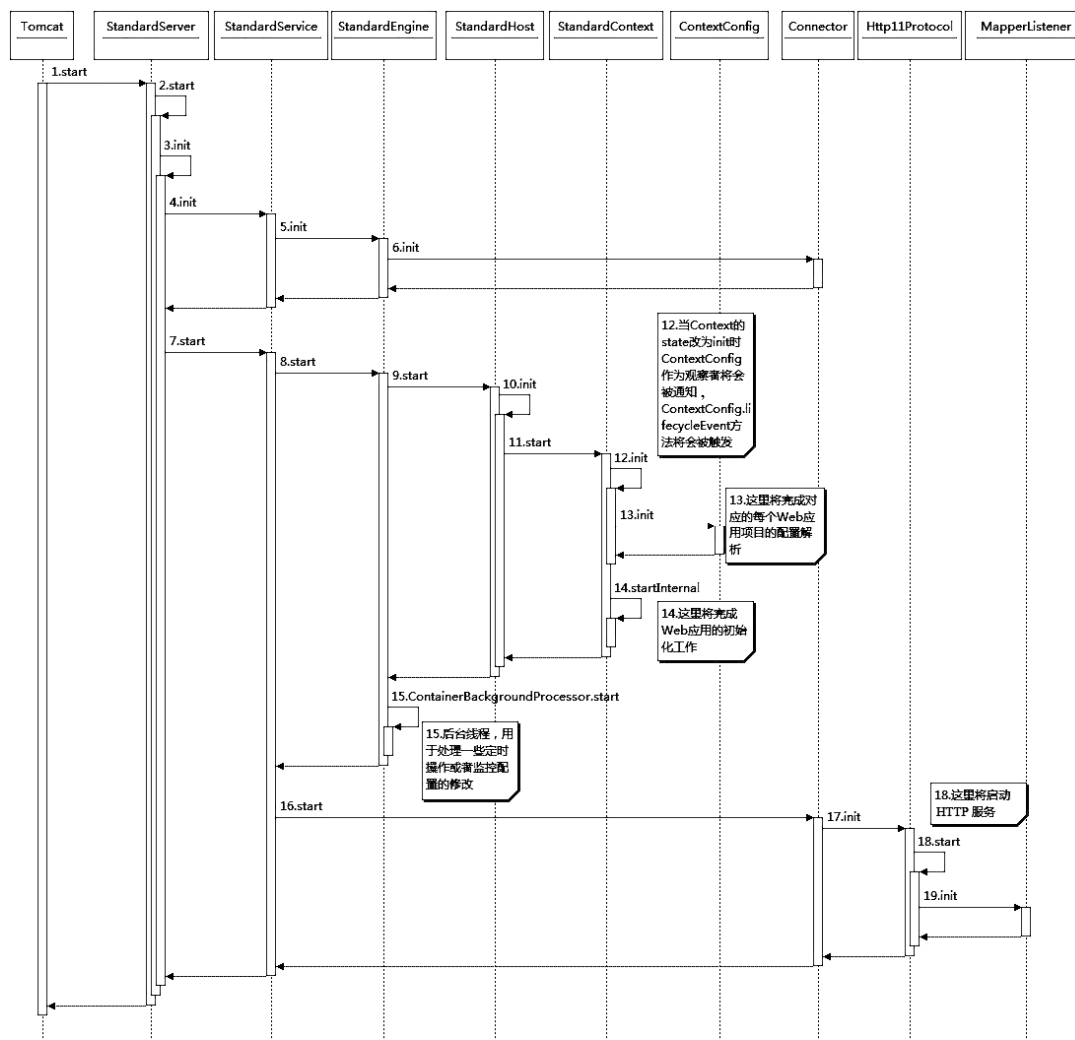


图 9-2 Tomcat 主要类的启动时序图

图 9-2 描述了在 Tomcat 的启动过程中主要类之间的时序关系，下面我们将会重点关注添加 examples 应用所对应的 StandardContext 容器的启动过程。

当 Context 容器初始化状态设为 init 时，添加到 Context 容器的 Listener 将会被调用。ContextConfig 继承了 LifecycleListener 接口，它是在调用 Tamcat.addWebapp 时被加入到 StandardContext 容器中的。ContextConfig 类会负责整个 Web 应用的配置文件的解析工作。

ContextConfig 的 init 方法将会主要完成以下工作。

- ⊙ 创建用于解析 XML 配置文件的 contextDigester 对象。
- ⊙ 读取默认的 context.xml 配置文件，如果存在则解析它。
- ⊙ 读取默认的 Host 配置文件，如果存在则解析它。
- ⊙ 读取默认的 Context 自身的配置文件，如果存在则解析它。
- ⊙ 设置 Context 的 DocBase。

ContextConfig 的 init 方法完成后，Context 容器就会执行 startInternal 方法，这个方法的启动逻辑比较复杂，主要包括如下几部分。

- ⊙ 创建读取资源文件的对象。
- ⊙ 创建 ClassLoader 对象。
- ⊙ 设置应用的工作目录。
- ⊙ 启动相关的辅助类，如 logger、realm、resources 等。
- ⊙ 修改启动状态，通知感兴趣的观察者（Web 应用的配置）。
- ⊙ 子容器的初始化。
- ⊙ 获取 ServletContext 并设置必要的参数。
- ⊙ 初始化“load on startup”的 Servlet。

9.1.2 Web 应用的初始化工作

Web 应用的初始化工作是在 ContextConfig 的 configureStart 方法中实现的，应用的初始化主要是解析 web.xml 文件，这个文件描述了一个 Web 应用的关键信息，也是一个 Web 应用的入口。

Tomcat 首先会找 `globalWebXml`，这个文件的搜索路径是 `engine` 的工作目录下的 `org/apache/catalina/startup/NO_DEFAULT_XML` 或 `conf/web.xml`。接着会找 `hostWebXml`，这个文件可能会在 `System.getProperty("catalina.base")/conf/${EngineName}/${HostName}/web.xml.default` 中，接着寻找应用的配置文件 `examples/WEB-INF/web.xml`。`web.xml` 文件中的各个配置项将会被解析成相应的属性保存在 `WebXml` 对象中。如果当前的应用支持 Servlet 3.0，解析还将完成额外的 9 项工作，这额外的 9 项工作主要是 Servlet 3.0 新增的特性（包括 jar 包中的 `META-INF/web-fragment.xml`）的解析及对 annotations 的支持。

接下来会将 `WebXml` 对象中的属性设置到 `Context` 容器中，这里包括创建 `Servlet` 对象、`filter`、`listener` 等，这段代码在 `WebXml` 的 `configureContext` 方法中。下面是解析 `Servlet` 的代码片段：

```
for (ServletDef servlet : servlets.values()) {
    Wrapper wrapper = context.createWrapper();
    String jspFile = servlet.getJspFile();
    if (jspFile != null) {
        wrapper.setJspFile(jspFile);
    }
    if (servlet.getLoadOnStartup() != null) {
        wrapper.setLoadOnStartup(servlet.getLoadOnStartup().intValue());
    }
    if (servlet.getEnabled() != null) {
        wrapper.setEnabled(servlet.getEnabled().booleanValue());
    }
    wrapper.setName(servlet.getServletName());
    Map<String,String> params = servlet.getParameterMap();
    for (Entry<String, String> entry : params.entrySet()) {
        wrapper.addInitParameter(entry.getKey(), entry.getValue());
    }
    wrapper.setRunAs(servlet.getRunAs());
    Set<SecurityRoleRef> roleRefs = servlet.getSecurityRoleRefs();
    for (SecurityRoleRef roleRef : roleRefs) {
        wrapper.addSecurityReference(
            roleRef.getName(), roleRef.getLink());
    }
    wrapper.setServletClass(servlet.getServletClass());
    MultipartDef multipartdef = servlet.getMultipartDef();
    if (multipartdef != null) {
        if (multipartdef.getMaxFileSize() != null &&
```

```

        multipartdef.getMaxRequestSize() != null &&
        multipartdef.getFileSizeThreshold() != null) {
            wrapper.setMultipartConfigElement(new MultipartConfig-
Element(
                multipartdef.getLocation(),
                Long.parseLong(multipartdef.getMaxFileSize()),
                Long.parseLong(multipartdef.getMaxRequestSize()),
                Integer.parseInt(
                    multipartdef.getFileSizeThreshold()));
        } else {
            wrapper.setMultipartConfigElement(new MultipartConfig-
Element(
                multipartdef.getLocation()));
        }
    }
    if (servlet.getAsyncSupported() != null) {
        wrapper.setAsyncSupported(
            servlet.getAsyncSupported().booleanValue());
    }
    context.addChild(wrapper);
}

```

这段代码清楚地描述了如何将 Servlet 包装成 Context 容器中的 StandardWrapper，这里有个疑问，为什么要将 Servlet 包装成 StandardWrapper 而不直接包装成 Servlet 对象？这里 StandardWrapper 是 Tomcat 容器中的一部分，它具有容器的特征，而 Servlet 作为一个独立的 Web 开发标准，不应该强耦合在 Tomcat 中。

除了将 Servlet 包装成 StandardWrapper 并作为子容器添加到 Context 中外，其他所有的 web.xml 属性都被解析到 Context 中，所以说 Context 容器才是真正运行 Servlet 的 Servlet 容器。一个 Web 应用对应一个 Context 容器，容器的配置属性由应用的 web.xml 指定，这样我们就能理解 web.xml 到底起什么作用了。

9.2 创建 Servlet 实例

前面已经完成了 Servlet 的解析工作，并且被包装成 StandardWrapper 添加在 Context 容器中，但是它仍然不能为我们工作，它还没有被实例化。下面我们将介绍 Servlet 对象

是如何创建的，以及是如何被初始化的。

9.2.1 创建 Servlet 对象

如果 Servlet 的 load-on-startup 配置项大于 0，那么在 Context 容器启动时就会被实例化，前面提到在解析配置文件时会读取默认的 globalWebXml，在 conf 下的 web.xml 文件中定义了一些默认的配置项，其中定义了两个 Servlet，分别是 org.apache.catalina.servlets.DefaultServlet 和 org.apache.jasper.servlet.JspServlet。它们的 load-on-startup 分别是 1 和 3，也就是当 Tomcat 启动时这两个 Servlet 就会被启动。

创建 Servlet 实例的方法是从 Wrapper.loadServlet 开始的。loadServlet 方法要完成的就是获取 servletClass，然后把它交给 InstanceManager 去创建一个基于 servletClass.class 的对象。如果这个 Servlet 配置了 jsp-file，那么这个 servletClass 就是在 conf/web.xml 中定义的 org.apache.jasper.servlet.JspServlet 了。

创建 Servlet 对象的相关类结构如图 9-3 所示。

9.2.2 初始化 Servlet

初始化 Servlet 在 StandardWrapper 的 initServlet 方法中，这个方法很简单，就是调用 Servlet 的 init() 方法，同时把包装了 StandardWrapper 对象的 StandardWrapperFacade 作为 ServletConfig 传给 Servlet。对于 Tomcat 容器为何要传 StandardWrapperFacade 给 Servlet 对象将在后面做详细解析。

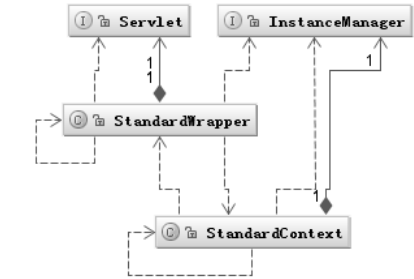


图 9-3 创建 Servlet 对象的相关类结构

如果该 Servlet 关联的是一个 JSP 文件，那么前面初始化的就是 JspServlet，接下来会模拟一次简单请求，请求调用这个 JSP 文件，以便编译这个 JSP 文件为类，并初始化这个类。

这样 Servlet 对象就初始化完成了，事实上 Servlet 从被 web.xml 解析到完成初始化，这个过程非常复杂，中间有很多过程，包括各种容器状态的转化引起的监听事件的触发、各种访问权限的控制和一些不可预料的错误发生的判断行为等。我们在这里只抓了一些关键环节进行阐述，以便于让大家有个总体脉络。

图 9-4 是这个过程的一个完整的时序图，在其中也省略了一些细节。

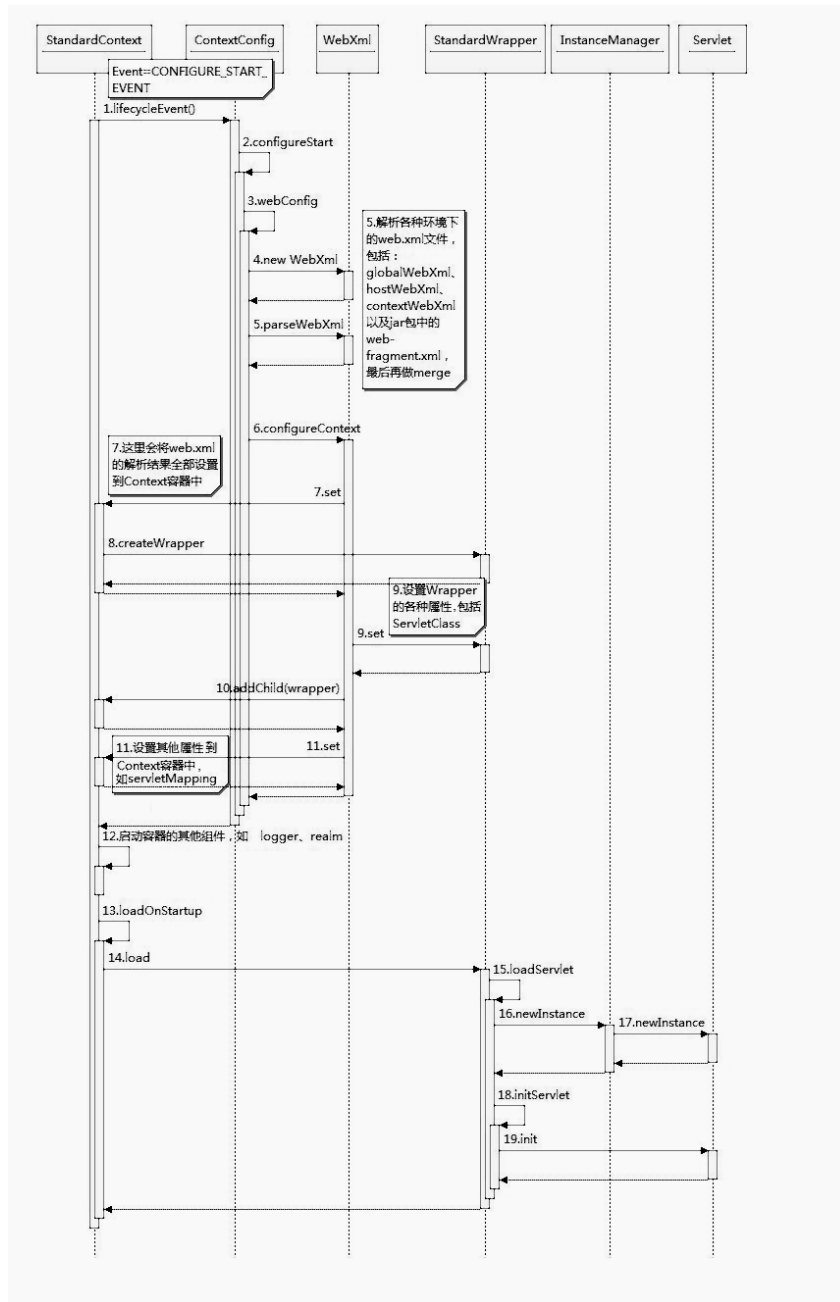


图 9-4 初始化 Servlet 的时序图

9.3 Servlet 体系结构

我们知道 Java Web 应用是基于 Servlet 规范运转的，那么 Servlet 本身又是如何运转的呢？为何要设计这样的体系结构呢？Servlet 顶层类的关联图如图 9-5 所示。

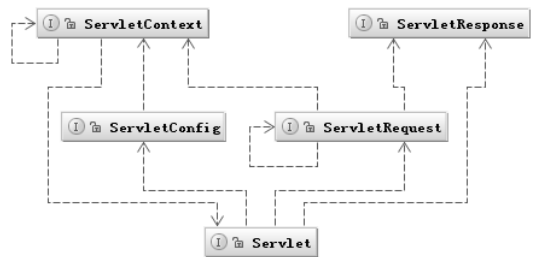


图 9-5 Servlet 顶层类关联图

从图 9-5 可以看出，Servlet 规范就是基于这几个类运转的，与 Servlet 主动关联的是三个类，分别是 ServletConfig、ServletRequest 和 ServletResponse。这三个类都是通过容器传递给 Servlet 的，其中 ServletConfig 在 Servlet 初始化时就传给 Servlet 了，而后两个是在请求达到时调用 Servlet 传递过来的。我们很清楚 ServletRequest 和 ServletResponse 在 Servlet 运行时的意义，但是 ServletConfig 和 ServletContext 对 Servlet 有何价值？仔细查看 ServletConfig 接口中声明的方法会发现，这些方法都是为了获取这个 Servlet 的一些配置属性，而这些配置属性可能在 Servlet 运行时被用到。ServletContext 又是干什么的呢？Servlet 的运行模式是一个典型的“握手型的交互式”运行模式。所谓“握手型的交互式”就是两个模块为了交换数据通常都会准备一个交易场景，这个场景一直跟随这个交易过程直到这个交易完成为止。这个交易场景的初始化是根据这次交易对象指定的参数来定制的，这些指定参数通常就是一个配置类。所以对号入座，交易场景就由 ServletContext 来描述，而定制的参数集合就由 ServletConfig 来描述。而 ServletRequest 和 ServletResponse 就是要交互的具体对象，它们通常都作为运输工具来传递交互结果。

ServletConfig 是在 Servlet init 时由容器传过来的，那么 ServletConfig 到底是个什么对象呢？

图 9-6 是 ServletConfig 和 ServletContext 在 Tomcat 容器中的类关系图。

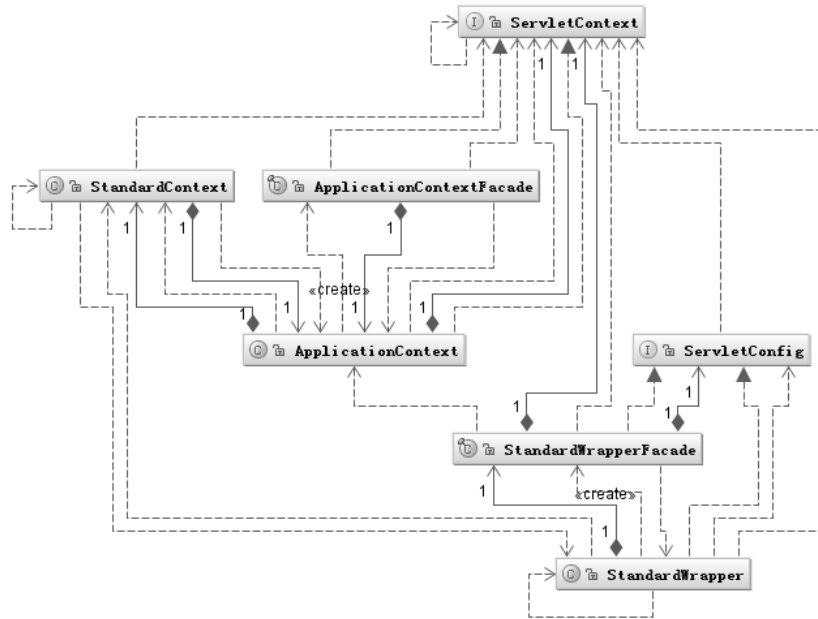


图 9-6 ServletConfig 在容器中的类关联图

可以看出，StandardWrapper 和 StandardWrapperFacade 都实现了 ServletConfig 接口，而 StandardWrapperFacade 是 StandardWrapper 门面类。所以传给 Servlet 的是 StandardWrapperFacade 对象，这个类能够保证从 StandardWrapper 中拿到 ServletConfig 所规定的数据，而又不把 ServletConfig 不关心的数据暴露给 Servlet。

同样 ServletContext 也与 ServletConfig 有类似的结构，在 Servlet 中能拿到的 ServletContext 的实际对象也是 ApplicationContextFacade 对象。ApplicationContextFacade 同样保证 ServletContext 只能从容器中拿到它该拿的数据，它们都起到对数据的封装作用，它们使用的都是门面设计模式。

通过 ServletContext 可以拿到 Context 容器中的一些必要信息，如应用的工作路径、容器支持的 Servlet 最小版本等。

在 Servlet 中定义的两个 ServletRequest 和 ServletResponse 实际的对象又是什么呢？我们在创建自己的 Servlet 类时通常使用的都是 HttpServletRequest 和 HttpServletResponse，它们继承了 ServletRequest 和 ServletResponse。为何 Context 容器传过来的 ServletRequest、ServletResponse 可以被转化为 HttpServletRequest 和 HttpServletResponse 呢？

图 9-7 是 Tomcat 创建的 Request 和 Response 的类结构图。Tomcat 接到请求首先将会创建 `org.apache.coyote.Request` 和 `org.apache.coyote.Response`，这两个类是 Tomcat 内部使用的描述一次请求和相应的信息类，它们是一个轻量级的类，作用就是在服务器接收到请求后，经过简单解析将这个请求快速分配给后续线程去处理，所以它们的对象很小，很容易被 JVM 回收。接下来当交给一个用户线程去处理这个请求时又创建 `org.apache.catalina.connector.Request` 和 `org.apache.catalina.connector.Response` 对象。这两个对象一直贯穿整个 Servlet 容器直到要传给 Servlet，传给 Servlet 的是 Request 和 Response 的门面类 `RequestFacade` 和 `ResponseFacade`，这里使用门面模式与前面一样都是基于同样的目的——封装容器中的数据。一次请求对应的 Request 和 Response 的类转化如图 9-8 所示。

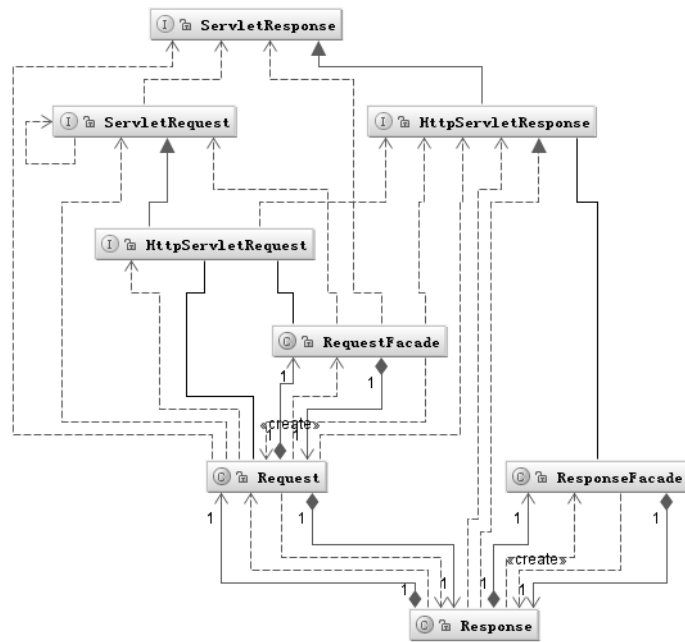


图 9-7 与 Request 相关的类结构图

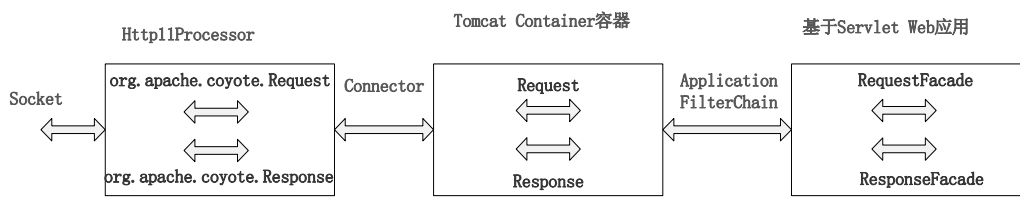


图 9-8 Request 和 Response 的转变过程

9.4 Servlet 如何工作

我们已经清楚了 Servlet 是如何被加载的、如何被初始化的，以及 Servlet 的体系结构，现在的问题就是它是如何被调用的？

用户从浏览器向服务器发起的一个请求通常会包含如下信息：`http://hostname: port /contextpath/servletpath`，`hostname` 和 `port` 用来与服务器建立 TCP 连接，后面的 URL 才用来选择在服务器中哪个子容器服务用户的请求。服务器是如何根据这个 URL 来到达正确的 Servlet 容器中的呢？

在 Tomcat 7 中这件事很容易解决，因为这种映射工作由专门一个的类来完成，这个类就是 `org.apache.tomcat.util.http.mapper`，这个类保存了 Tomcat 的 Container 容器中的所有子容器的信息，`org.apache.catalina.connector.Request` 类在进入 Container 容器之前，Mapper 将会根据这次请求的 `hostname` 和 `contextpath` 将 `host` 和 `context` 容器设置到 `Request` 的 `mappingData` 属性中，如图 9-9 所示。所以当 `Request` 进入 Container 容器之前，对于它要访问哪个子容器就已经确定了。

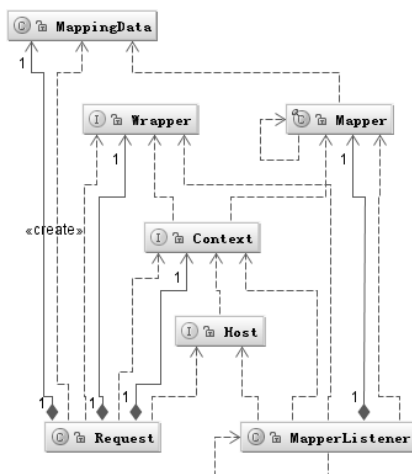


图 9-9 Request 的 Mapper 类关系图

可能你有疑问，在 `Mapper` 中怎么会有容器的完整关系？这要回到图 9-2 中第 19 步 `MapperListener` 类的初始化过程，下面是 `MapperListener` 的 `init` 方法的代码：

```
public void init() {
```

```
findDefaultHost();
Engine engine = (Engine) connector.getService().getContainer();
engine.addContainerListener(this);
Container[] conHosts = engine.findChildren();
for (Container conHost : conHosts) {
    Host host = (Host) conHost;
    if (!LifecycleState.NEW.equals(host.getState())) {
        host.addLifecycleListener(this);
        registerHost(host);
    }
}
}
```

这段代码的作用就是将 `MapperListener` 类作为一个监听者加到整个 `Container` 容器的每个子容器中，这样只要任何一个容器发生变化，`MapperListener` 都将会被通知到，相应的保存容器关系的 `MapperListener` 的 `mapper` 属性也会被修改。在 `for` 循环中就是将 `host` 及下面的子容器注册到 `mapper` 中。

图 9-10 描述了一次 `Request` 请求是如何到达最终的 `Wrapper` 容器的，我们现在知道了请求如何到达正确的 `Wrapper` 容器，但是在请求达到最终的 `Servlet` 前还要完成一些步骤，必须要执行 `Filter` 链，以及通知你在 `web.xml` 中定义的 `listener`。

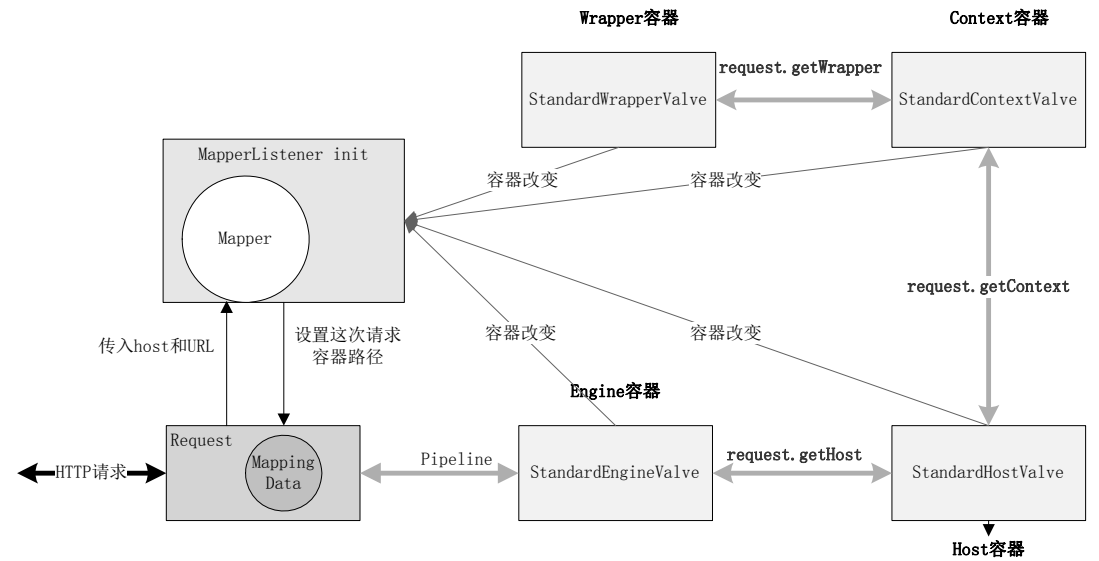


图 9-10 Request 在容器中的路由图

接下来就要执行 Servlet 的 service 方法了。通常情况下，我们自己定义的 servlet 并不直接去实现 javax.servlet.servlet 接口，而是去继承更简单的 HttpServlet 类或者 GenericServlet 类，我们可以有选择地覆盖相应的方法去实现要完成的工作。

Servlet 的确已经能够帮我们完成所有的工作了，但是现在的 Web 应用很少直接将交互的全部页面用 Servlet 来实现，而是采用更加高效的 MVC 框架来实现。这些 MVC 框架的基本原理是将所有的请求都映射到一个 Servlet，然后去实现 service 方法，这个方法也就是 MVC 框架的入口。

当 Servlet 从 Servlet 容器中移除时，也就表明该 Servlet 的生命周期结束了，这时 Servlet 的 destroy 方法将被调用，做一些扫尾工作。

9.5 Servlet 中的 Listener

在整个 Tomcat 服务器中，Listener 使用得非常广泛，它是基于观察者模式设计的，Listener 的设计为开发 Servlet 应用程序提供了一种快捷的手段，能够方便地从另一个纵向维度控制程序和数据。目前在 Servlet 中提供了 6 种两类事件的观察者接口，它们分别是：EventListeners 类型的 ServletContextAttributeListener、ServletRequestAttributeListener、ServletRequestListener、HttpSessionAttributeListener 和 LifecycleListeners 类型的 ServletContextListener、HttpSessionListener，如图 9-11 所示。

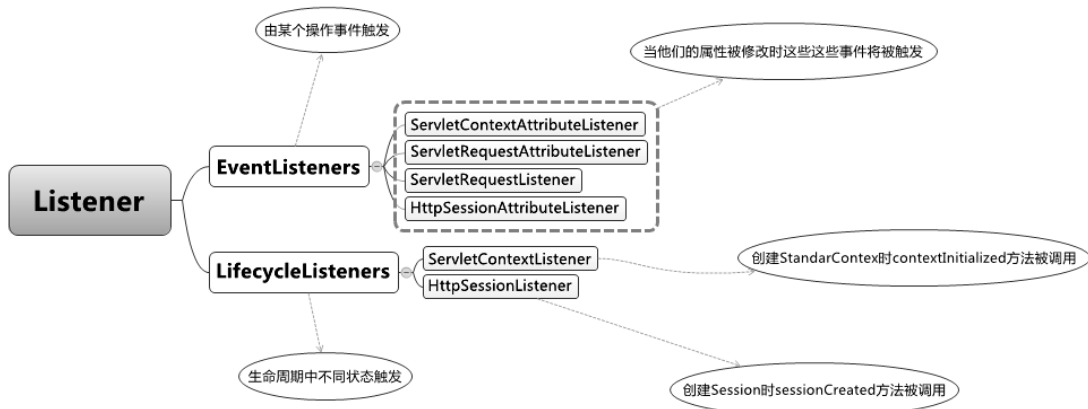


图 9-11 Servlet 中的 Listener

实际上，这 6 个 Listener 都继承了 EventListener 接口，每个 Listener 各自定义了需要实现的接口，这些接口如表 9-1 所示。

表 9-1 Listener 需要实现的接口及说明

Listener 类	含有的接口	接 口 说 明
ServletContextAttributeListener	AttributeAdded (ServletContextAttributeEvent scab)	当调用 servletContext.setAttribute 方法时触发这个接口
	AttributeRemoved (ServletContextAttributeEvent scab)	当调用 servletContext.removeAttribute 方法时触发这个接口
	AttributeReplaced (ServletContextAttributeEvent scab)	如果在调用 servletContext.setAttribute 之前该 attribute 已经存在，则替换这个 attribute 时，这个接口被触发
ServletRequestAttributeListener	AttributeAdded (ServletRequestAttributeEvent srae)	当调用 request.setAttribute 方法时触发这个接口
	AttributeRemoved (ServletRequestAttributeEvent srae)	当调用 request.removeAttribute 方法时触发这个接口
	AttributeReplaced (ServletRequestAttributeEvent srae)	如果在调用 request.setAttribute 之前该 attribute 已经存在，则替换这个 attribute 时这个接口被触发
ServletRequestListener	requestInitialized (ServletRequestEvent sre)	当 HttpServletRequest 对象被传递到用户的 Servlet 的 service 方法之前该方法被触发
	requestDestroyed (ServletRequestEvent sre)	当 HttpServletRequest 对象在调用完用户的 Servlet 的 service 方法之后该方法被触发
HttpSessionAttributeListener	attributeAdded (HttpSessionBindingEvent se)	session.setAttribute 方法被调用时该接口被触发
	attributeRemoved (HttpSessionBindingEvent se)	session.removeAttribute 方法被调用时该接口被触发
	attributeReplaced (HttpSessionBindingEvent se)	如果在调用 session.setAttribute 之前该 attribute 已经存在，则替换这个 attribute 时这个接口被触发
ServletContextListener	contextInitialized (ServletContextEvent sce)	Context 容器初始化时触发，在所有的 Filter 和 Servlet 的 init 方法调用之前 contextInitialized 接口先被调用

续表

Listener 类	含有的接口	接 口 说 明
ServletContextListener	contextDestroyed (ServletContextEvent sce)	Context 容器销毁，在所有的 Filter 和 Servlet 的 destroy 方法调用之后 contextDestroyed 接口被调用
HttpSessionListener	SessionCreated (HttpSessionEvent se)	当一个 session 对象被创建时触发
	SessionDestroyed (HttpSessionEvent se)	当一个 session 对象被失效时触发

它们基本上涵盖了整个 Servlet 生命周期中你感兴趣的每种事件。这些 Listener 的实现类可以配置在 web.xml<listener>标签中。当然也可以在应用程序中动态添加 Listener，需要注意的是 ServletContextListener 在容器启动之后就不能再添加新的，因为它所监听的事件已经不会再出现了。掌握这些 Listener 的使用方法，能够让我们的程序设计得更加灵活。

如 Spring 的 org.springframework.web.context.ContextLoaderListener 就实现了一个 ServletContextListener，当容器加载时启动 Spring 容器。ContextLoaderListener 在 contextInitialized 方法中初始化 Spring 容器，有几种办法可以加载 Spring 容器，通过在 web.xml 的 <context-param> 标签中配置 Spring 的 applicationContext.xml 路径，文件名可以任意取，如果没有配置，将在/WEB-INF/路径下查找默认的 applicationContext.xml 文件。ContextLoaderListener 的 contextInitialized 方法代码如下：

```
public void contextInitialized(ServletContextEvent event) {
    this.contextLoader = createContextLoader();
    if (this.contextLoader == null) {
        this.contextLoader = this;
    }
    this.contextLoader.initWebApplicationContext(event.getServletContext());
}
```

9.6 Filter 如何工作

Filter 也是在 web.xml 中另外一个常用的配置项，可以通过<filter>和<filter-mapping>组合来使用 Filter。实际上 Filter 可以完成与 Servlet 同样的工作，甚至比 Servlet 使用起来更加灵活，因为它除了提供了 request 和 response 对象外，还提供了一个 FilterChain 对象，这个对象可以让我们更加灵活地控制请求的流转。下面看一下与 Filter 相关的类结构图，如图 9-12 所示。

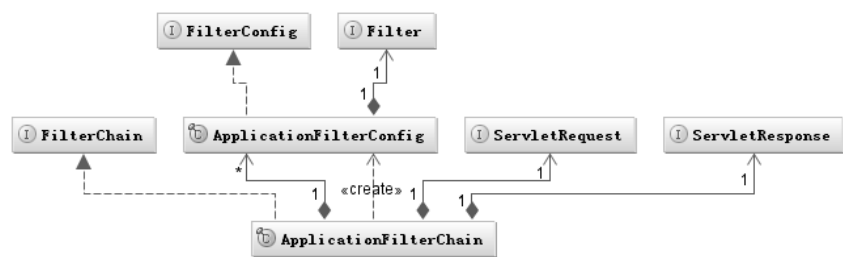


图 9-12 Filter 相关的类结构图

在 Tomcat 容器中，FilterConfig 和 FilterChain 的实现类分别是 ApplicationFilterConfig 和 ApplicationFilterChain，而 Filter 的实现类由用户自定义，只要实现 Filter 接口中定义三个接口就行，这三个接口与在 Servlet 中的类似。只不过还有一个 ApplicationFilterChain 类，这个类可以将多个 Filter 串联起来，组成一个链，这个链与 Jetty 中的 Handler 链有异曲同工之妙。下面详细看一下 Filter 类中的三个接口方法。

- ◉ **init(FilterConfig):** 初始化接口，在用户自定义的 Filter 初始化时被调用，它与 Servlet 的 init 方法的作用是一样的，FilterConfig 与 ServletConfig 也类似，除了都能取到容器的环境类 ServletContext 对象之外，还能获取在<filter>下配置的<init-param>参数值。
- ◉ **doFilter (ServletRequest, ServletResponse, FilterChain):** 在每个用户的请求进来时这个方法都会被调用，并在 Servlet 的 service 方法之前被调用。而 FilterChain 就代表当前的整个请求链，所以通过调用 FilterChain.doFilter 可以将请求继续传递下去。如果想拦截这个请求，可以不调用 FilterChain.doFilter，那么这个请求就直接返回了。所以 Filter 是一种责任链设计模式。
- ◉ **destroy:** 当 Filter 对象被销毁时，这个方法被调用。注意，当 Web 容器调用这个方法之后，容器会再调用一次 doFilter 方法。

Filter 类的核心还是传递的 FilterChain 对象，这个对象保存了到最终 Servlet 对象的所有 Filter 对象，这些对象都保存在 ApplicationFilterChain 对象的 filters 数组中。在 FilterChain 链上每执行一个 Filter 对象，数组的当前计数都会加 1，直到计数等于数组的长度，当 FilterChain 上所有的 Filter 对象执行完成后，就会执行最终的 Servlet。所以在 ApplicationFilterChain 对象中会持有 Servlet 对象的引用。图 9-13 是 Filter 对象的执行时序图。

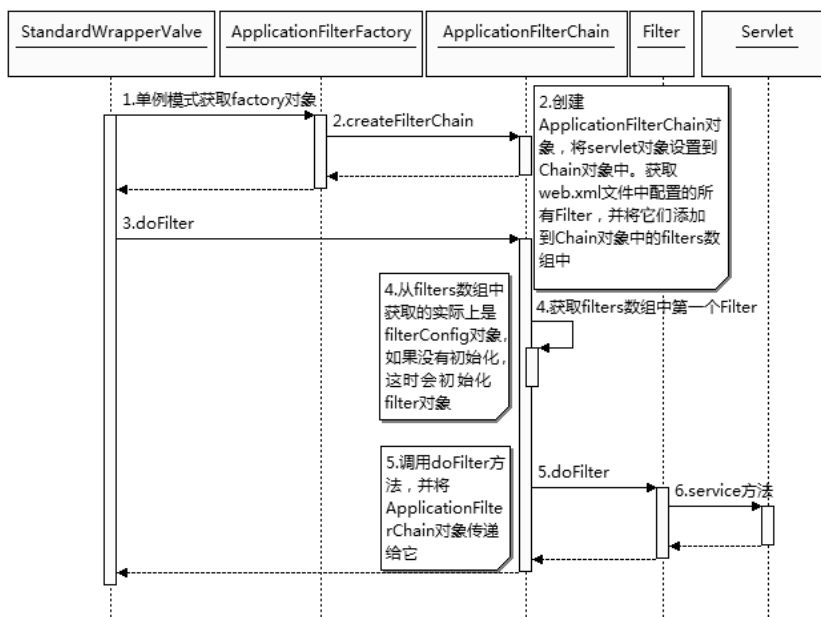


图 9-13 Filter 执行时序图

Filter 存在的意义就好比是你要去北京，它是你的目的地，但是提供一个机制让你在去的途中可以做一些拦截工作，如可以将你的一些行李包存放在某个“存放处”，当你返回时你可以再从这个地方取回。总之它可以在你的途中增加一些东西，或者减少一些东西。

9.7 Servlet 中的 url-pattern

在 web.xml 中<servlet-mapping>和<filter-mapping>都有<url-pattern>配置项，它们的作用都是匹配一次请求是否会执行这个 Servlet 或者 Filter，那么这个 URL 是怎么匹配的，又是何时匹配的呢？

先看看 Servlet 是何时匹配的。在 9.4 节中介绍了一个请求最终被分配到一个 Servlet 中是通过 org.apache.tomcat.util.http.Mapper 类完成的，这个类会根据请求的 URL 来匹配在每个 Servlet 中配置的<url-pattern>，所以它在一个请求被创建时就已经匹配了。

Filter 的 url-pattern 匹配是在创建 ApplicationFilterChain 对象时进行的, 它会把所有定义的 Filter 的 url-pattern 与当前的 URL 匹配, 如果匹配成功就将这个 Filter 保存到 ApplicationFilterChain 的 filters 数组中, 然后在 FilterChain 中依次调用。

在 web.xml 加载时, 会首先检查<url-pattern>配置是否符合规则, 这个检查是在 StandardContext 的 validateURLPattern 方法中检查的, 如果检查不成功, Context 容器启动会失败, 并且会报 java.lang.IllegalArgumentException:Invalid<url-pattern> /a/*.htm in Servlet mapping 错误。

<url-pattern>的解析规则, 对 Servlet 和 Filter 是一样的, 匹配的规则有如下三种。

- ⊙ 精确匹配: 如/foo.htm 只会匹配 foo.htm 这个 URL。
- ⊙ 路径匹配: 如/foo/*会匹配以 foo 为前缀的 URL。
- ⊙ 后缀匹配: 如*.htm 会匹配所有以.htm 为后缀的 URL。

Servlet 的匹配规则在 org.apache.tomcat.util.http.mapper.Mapper.internalMapWrapper 中定义, 对 Servlet 的匹配来说如果同时定义了多个<url-pattern>, 那么到底匹配那个 Servlet 呢? 这个匹配顺序是: 首先精确匹配, 如定义了两个 Servlet, Servlet1 为/foo.htm, Servlet2 是/*, 请求 URL 为 http://localhost/foo.htm, 那么只有 Servlet1 匹配成功; 如果精确匹配不成功, 那么会使用第二个原则“最长路径匹配”, 如 Servlet1 为/foo/*, Servlet2 为/*, 这时请求的 URL 为 http://localhost/foo/foo.htm, 那么 Servlet1 匹配成功; 最后根据后缀进行匹配, 但是一次请求只会成功匹配到一个 Servlet。

Filter 的匹配规则在 ApplicationFilterFactory.matchFiltersURL 方法中定义。Filter 的匹配原则和 Servlet 有些不同, 只要匹配成功, 这些 Filter 都会在请求链上被调用。<url-pattern>的其他写法 (如/foo/、/*.htm 和 */foo) 都是不对的。

9.8 总结

本章从 Servlet 容器的启动、Servlet 的初始化及 Servlet 的体系结构等内容中选出一些重点来讲述, 目的是让读者有一个总体的完整结构图, 同时本章也详细分析了其中的一些难点问题, 希望对大家有所帮助。

第 10 章

深入理解 Session 与 Cookie

Session 与 Cookie 不管是对 Java Web 的初学者还是熟练使用者来说都是一个令人头疼的问题。在初入职场时恐怕很多程序员在面试时都被问过这个问题。其实这个问题回答起来既简单又复杂，简单是因为它们本身只是 HTTP 中的一个配置项，在 Servlet 规范中也只是对应到一个类而已；说它复杂原因在于当我们的系统大到需要用到很多 Cookie 时，我们不得不考虑 HTTP 对 Cookie 数量和大小的限制，那么如何才能解决这个瓶颈呢？Session 也会有同样的问题，当我们有一个应用系统有几百台服务器时，如何解决 Session 在多台服务器之间共享的问题？它们还有一些安全问题，如 Cookie 被盗、Cookie 伪造等问题应如何避免。本章将详细解答这些问题，同时也将分享淘宝在解决这些问题时总结的一些经验。

Session 与 Cookie 的作用都是为了保持访问用户与后端服务器的交互状态。它们有各自的优点，也有各自的缺陷，然而具有讽刺意味的是它们的优点和它们的使用场景又是矛盾的。例如，使用 Cookie 来传递信息时，随着 Cookie 个数的增多和访问量的增加，它占用的网络带宽也很大，试想假如 Cookie 占用 200 个字节，如果一天的 PV 有几亿，那么它要占用多少带宽？所以有大访问量时希望用 Session，但是 Session 的致命弱点是不容易在多台服务器之间共享，这也限制了 Session 的使用。

10.1 理解 Cookie

Cookie 的作用我想大家都知道，通俗地说就是当一个用户通过 HTTP 访问一个服务器时，这个服务器会将一些 Key/Value 键值对返回给客户端浏览器，并给这些数据加上一些限制条件，在条件符合时这个用户下次访问这个服务器时，数据又被完整地带回给服务器。

这个作用就像你去超市购物时，第一次给你办张购物卡，在这个购物卡里存放了一些你的个人信息，下次你再来这个连锁超市时，超市会识别你的购物卡，下次直接购物就好了。

当初 W3C 在设计 Cookie 时实际上考虑的是为了记录用户在一段时间内访问 Web 应用的行为路径。由于 HTTP 是一种无状态协议，当用户的一次访问请求结束后，后端服务器就无法知道下一次来访问的还是不是上次访问的用户。在设计应用程序时，我们很容易想到两次访问是同一人访问与不同的两个人访问对程序设计和性能来说有很大的不同。例如，在一个很短的时间内，如果与用户相关的数据被频繁访问，可以针对这个数据做缓存，这样可以大大提高数据的访问性能。Cookie 的作用正是如此，由于是同一个客户端发出的请求，每次发出的请求都会带有第一次访问时服务端设置的信息，这样服务端就可以根据 Cookie 值来划分访问的用户了。

10.1.1 Cookie 属性项

当前 Cookie 有两个版本：Version 0 和 Version 1，它们有两种设置响应头的标识，分别是“Set-Cookie”和“Set-Cookie2”。这两个版本的属性项有些不同，表 10-1 和表 10-2 是对这两个版本的属性介绍。

表 10-1 Version 0 属性项介绍

属 性 项	属性项介绍
NAME=VALUE	键值对，可以设置要保存的 Key/Value，注意这里的 NAME 不能和其他属性项的名字一样
Expires	过期时间，在设置的某个时间点后该 Cookie 就会失效，如 expires=Wednesday, 09-Nov-99 23:12:40 GMT
Domain	生成该 Cookie 的域名，如 domain="xulingbo.net"
Path	该 Cookie 是在当前哪个路径下生成的，如 path=/wp-admin/
Secure	如果设置了这个属性，那么只会在 SSH 连接时才会回传该 Cookie

表 10-2 Version 1 属性项介绍

属 性 项	属性项介绍
NAME=VALUE	与 Version 0 相同
Version	通过 Set-Cookie2 设置的响应头创建必须符合 RFC2965 规范，如果通过 Set-Cookie 响应头设置，则默认值为 0；如果要设置为 1，则该 Cookie 要遵循 RFC 2109 规范
Comment	注释项，用户说明该 Cookie 有何用途
CommentURL	服务器为此 Cookie 提供的 URI 注释
Discard	是否在会话结束后丢弃该 Cookie 项，默认为 false
Domain	类似于 Version 0
Max-Age	最大失效时间，与 Version 0 不同的是这里设置的是在多少秒后失效
Path	类似于 Version 0
Port	该 Cookie 在什么端口下可以回传服务端，如果有多个端口，则以逗号隔开，如 Port="80,81,8080"
Secure	类似于 Version 0

在以上两个版本的 Cookie 中设置的 Header 头的标识符是不同的，我们常用的是 Set-Cookie: userName="junshan"; Domain="xulingbo.net"，这是 Version 0 的形式。针对 Set-Cookie2 是这样设置的: Set-Cookie2: userName="junshan"; Domain="xulingbo.net"; Max-Age=1000。但是在 Java Web 的 Servlet 规范中并不支持 Set-Cookie2 响应头，在实际应用中 Set-Cookie2 的一些属性项却可以设置在 Set-Cookie 中，如这样设置: Set-Cookie: userName="junshan"; Version="1"; Domain="xulingbo.net"; Max-Age=1000。

10.1.2 Cookie 如何工作

当我们用如下方式创建 Cookie 时:

```
String getCookie(Cookie[] cookies, String key) {
    if (cookies != null) {
        for (Cookie cookie : cookies) {
            if (cookie.getName().equals(key)) {
                return cookie.getValue();
            }
        }
    }
    return null;
}
```

```

    }

    @Override
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {
        Cookie[] cookies = request.getCookies();
        String userName = getCookie(cookies, "userName");
        String userAge = getCookie(cookies, "userAge");
        if (userName == null) {
            response.addCookie(new Cookie("userName", "junshan"));
        }
        if (userAge == null) {
            response.addCookie(new Cookie("userAge", "28"));
        }
        response.getHeaders("Set-Cookie");
    }
}

```

Cookie 是如何加到 HTTP 的 Header 中的呢？当我们用 Servlet 3.0 规范来创建一个 Cookie 对象时，该 Cookie 既支持 Version 0 又支持 Version 1，如果你设置了 Version 1 中的配置项，即使你没有设置版本号，Tomcat 在最后构建 HTTP 响应头时也会自动将 Version 的版本设置为 1。下面看一下 Tomcat 是如何调用 addCookie 方法的，图 10-1 是 Tomcat 创建 Set-Cookie 响应头的时序图。

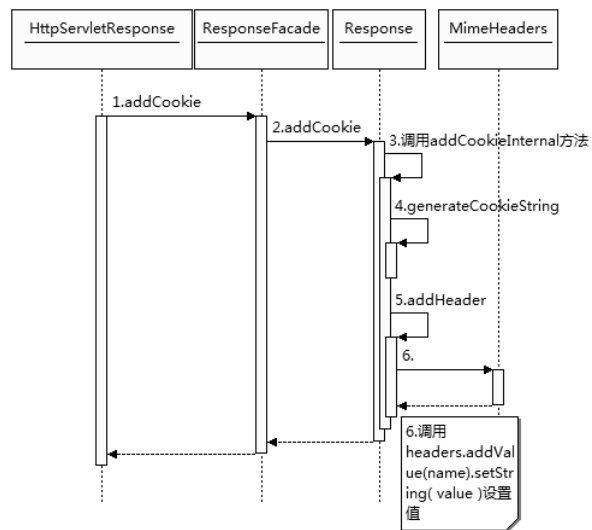


图 10-1 Tomcat 创建 Set-Cookie 响应头的时序图

从图 10-1 中可以看出，真正构建 Cookie 是在 `org.apache.catalina.connector.Response` 类中完成的，调用 `generateCookieString` 方法将 Cookie 对象构造成一个字符串，构造的字符串的格式如 `userName="junshan";Version="1";Domain="xulingbo.net";Max-Age=1000`。然后将这个字符串命名为 Set-Cookie 添加到 `MimeHeaders` 中。

在这里有以下几点需要注意。

- ⊙ 所创建 Cookie 的 NAME 不能和 Set-Cookie 或者 Set-Cookie2 的属性项值一样，如果一样会抛出 `IllegalArgumentException` 异常。
- ⊙ 所创建 Cookie 的 NAME 和 VALUE 的值不能设置成非 ASCII 字符，如果要使用中文，可以通过 `URLEncoder` 将其编码，否则会抛出 `IllegalArgumentException` 异常。
- ⊙ 当 NAME 和 VALUE 的值出现一些 TOKEN 字符（如 “\”、“,” 等）时，构建返回头会将该 Cookie 的 Version 自动设置为 1。
- ⊙ 当在该 Cookie 的属性项中出现 Version 为 1 的属性项时，构建 HTTP 响应头同样会将 Version 设置为 1。

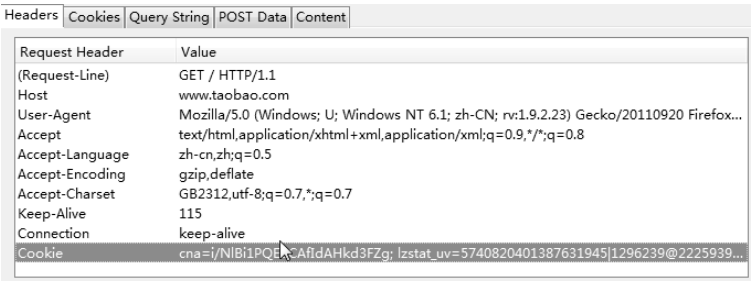
不知道你有没有注意到一个问题，就是当我们通过 `response.addCookie` 创建多个 Cookie 时，这些 Cookie 最终是在一个 Header 项中的还是以独立的 Header 存在的，通俗地说也就是我们每次创建 Cookie 时是否都是创建一个以 NAME 为 Set-Cookie 的 `MimeHeaders`？答案是肯定的。从上面的时序图中可以看出每次调用 `addCookie` 时，最终都会创建一个 Header，但是我们还不知道最终在请求返回时构造的 HTTP 响应头是否将相同 Header 标识的 Set-Cookie 值进行合并。

我们找到 Tomcat 最终构造 Http 响应头的代码，这段代码位于 `org.apache.coyote.http11.Http11Processor` 类的 `prepareResponse` 方法中，如下所示：

```
int size = headers.size();
for (int i = 0; i < size; i++) {
    outputBuffer.sendHeader(headers.getName(i), headers.getValue(i));
}
```

这段代码清楚地表示，在构建 HTTP 返回字节流时是将 Header 中所有的项顺序地写出，而没有进行任何修改。所以可以想象浏览器在接收 HTTP 返回的数据时是分别解析每一个 Header 项的。

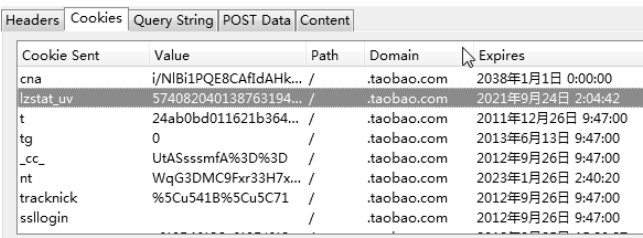
另外，目前很多工具都可以观察甚至可以修改浏览器中的 Cookie 数据。例如，在 Firefox 中可以通过 HttpFox 插件来查看返回的 Cookie 数据，如图 10-2 所示。



Request Header	Value
(Request-Line)	GET / HTTP/1.1
Host	www.taobao.com
User-Agent	Mozilla/5.0 (Windows; U; Windows NT 6.1; zh-CN; rv:1.9.2.23) Gecko/20110920 Firefox...
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language	zh-cn,zh;q=0.5
Accept-Encoding	gzip,deflate
Accept-Charset	GB2312,utf-8;q=0.7,*;q=0.7
Keep-Alive	115
Connection	keep-alive
Cookie	cna=i/NiBi1PQE8CAfdAHk3FZg; lzstat_uv=5740820401387631945j1296239@2225939...

图 10-2 HttpFox 插件展示的 Header 数据

在 Cookie 项中可以详细查看 Cookie 属性项，如图 10-3 所示。



Cookie Sent	Value	Path	Domain	Expires
cna	i/NiBi1PQE8CAfdAHk...	/	.taobao.com	2038年1月1日 0:00:00
lzstat_uv	574082040138763194...	/	.taobao.com	2021年9月24日 2:04:42
t	24ab0bd011621b364...	/	.taobao.com	2011年12月26日 9:47:00
tg	0	/	.taobao.com	2013年6月13日 9:47:00
cc	UtASssmfA%3D%3D	/	.taobao.com	2012年9月26日 9:47:00
nt	WqG3DMC9Fxr33H7x...	/	.taobao.com	2023年1月26日 2:40:20
tracknick	%5Cu541B%5Cu5C71	/	.taobao.com	2012年9月26日 9:47:00
ssllogin		/	.taobao.com	2012年9月26日 9:47:00

图 10-3 HttpFox 插件展示的 Cookie 数据

前面主要介绍了在服务端如何创建 Cookie，下面看一下如何从客户端获取 Cookie。

当我们请求某个 URL 路径时，浏览器会根据这个 URL 路径将符合条件的 Cookie 放在 Request 请求头中传回给服务端，服务端通过 request.getCookies()来取得所有 Cookie。

10.1.3 使用 Cookie 的限制

Cookie 是 HTTP 头中的一个字段，虽然 HTTP 本身对这个字段并没有多少限制，但是 Cookie 最终还是存储在浏览器里，所以不同的浏览器对 Cookie 的存储都有一些限制，表 10-3 是一些通常的浏览器对 Cookie 的大小和数量的限制。

表 10-3 浏览器对 Cookie 的大小和数量的限制

浏览器版本	Cookie 的数量限制	Cookie 的总大小限制
IE6	20 个/每个域名	4095 个字节

IE7	50 个/每个域名	4095 个字节
续表		
浏览器版本	Cookie 的数量限制	Cookie 的总大小限制
IE8	50 个/每个域名	4095 个字节
IE9	50 个/每个域名	4095 个字节
Chrome	50 个/每个域名	大于 80000
FireFox	50 个/每个域名	4097 个字

10.2 理解 Session

前面已经介绍了 Cookie 可以让服务端程序跟踪每个客户端的访问，但是每次客户端的访问都必须传回这些 Cookie，如果 Cookie 很多，则无形地增加了客户端与服务端的数据传输量，而 Session 的出现正是为了解决这个问题。

同一个客户端每次和服务端交互时，不需要每次都传回所有的 Cookie 值，而是只要传回一个 ID，这个 ID 是客户端第一次访问服务器时生成的，而且每个客户端是唯一的。这样每个客户端就有了一个唯一的 ID，客户端只要传回这个 ID 就行了，这个 ID 通常是 JSESSIONID 的一个 Cookie。

10.2.1 Session 与 Cookie

下面详细讲一下 Session 是如何基于 Cookie 来工作的。实际上有以下三种方式可以让 Session 正常工作。

- 基于 URL Path Parameter，默认支持。
- 基于 Cookie，如果没有修改 Context 容器的 Cookies 标识，则默认也是支持的。
- 基于 SSL，默认不支持，只有 `connector.getAttribute("SSLEnabled")` 为 TRUE 时才支持。

在第一种情况下，当浏览器不支持 Cookie 功能时，浏览器会将用户的 SessionCookieName 重写到用户请求的 URL 参数中，它的传递格式如 `/path/Servlet;name=value;name2=value2?Name3=value3`，其中“Servlet;”后面的 K-V 就是要传递的 Path Parameters，服务器会从

这个 Path Parameters 中拿到用户配置的 SessionCookieName。关于这个 SessionCookieName，如果在 web.xml 中配置 session-config 配置项，其 cookie-config 下的 name 属性就是这个 SessionCookieName 的值。如果没有配置 session-config 配置项，默认的 SessionCookieName 就是大家熟悉的“JSESSIONID”。需要说明的一点是，与 Session 关联的 Cookie 与其他 Cookie 没有什么不同。接着 Request 根据这个 SessionCookieName 到 Parameters 中拿到 Session ID 并设置到 request.setRequestedSessionId 中。

请注意，如果客户端也支持 Cookie，则 Tomcat 仍然会解析 Cookie 中的 Session ID，并会覆盖 URL 中的 Session ID。

如果是第三种情况，则会根据 javax.servlet.request.ssl_session 属性值设置 Session ID。

10.2.2 Session 如何工作

有了 Session ID，服务端就可以创建 HttpSession 对象了，第一次触发通过 request.getSession()方法。如果当前的 Session ID 还没有对应的 HttpSession 对象，那么就创建一个新的，并将这个对象加到 org.apache.catalina. Manager 的 sessions 容器中保存。Manager 类将管理所有 Session 的生命周期，Session 过期将被回收，服务器关闭，Session 将被序列化到磁盘等。只要这个 HttpSession 对象存在，用户就可以根据 Session ID 来获取这个对象，也就做到了对状态的保持。

与 Session 相关的类图如图 10-4 所示。

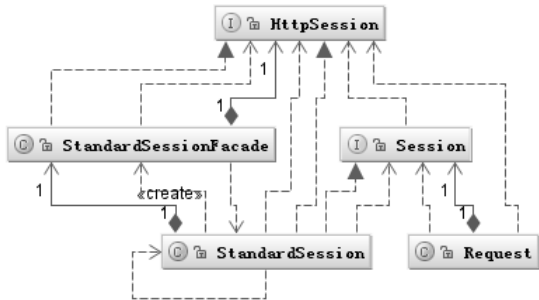


图 10-4 Session 相关类图

从图 10-4 中可以看出，从 request.getSession 中获取的 HttpSession 对象实际上是 StandardSession 对象的门面对象，这与前面的 Request 和 Servlet 是一样的原理。图 10-5

是 Session 工作的时序图。

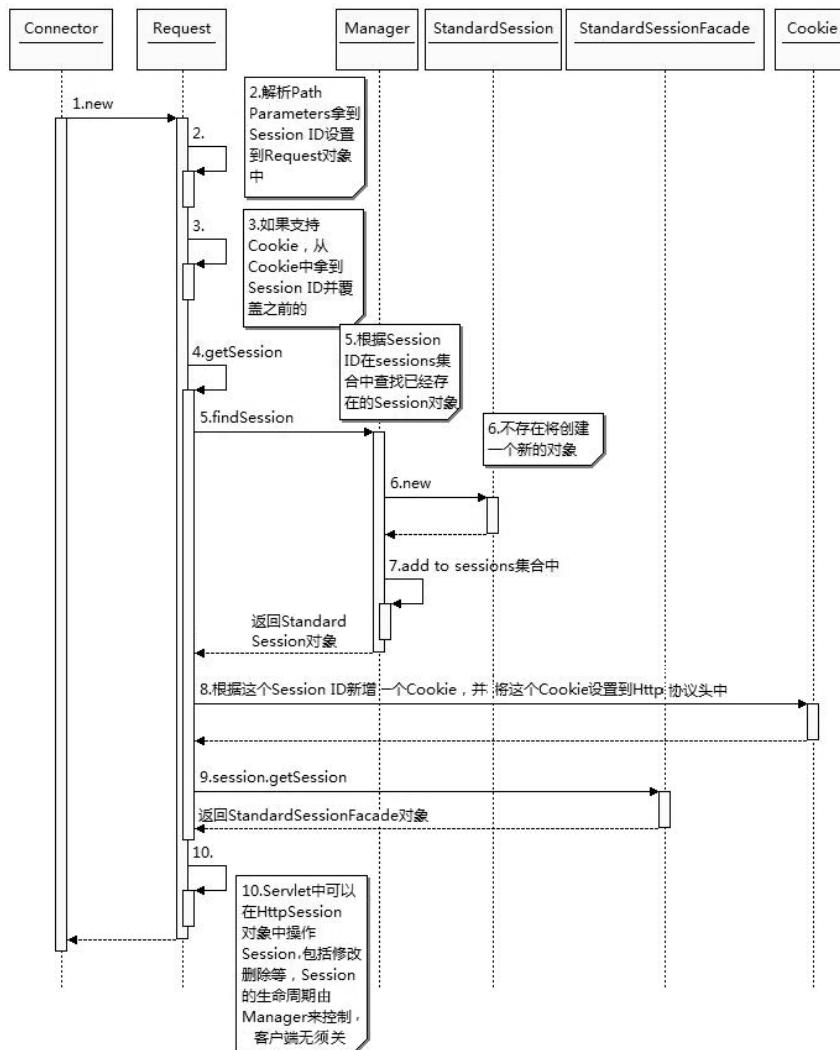


图 10-5 Session 工作的时序图

从时序图中可以看出，从 Request 中获取的 Session 对象保存在 org.apache.catalina.Manager 类中，它的实现类是 org.apache.catalina.session.StandardManager，通过

requestedSessionId 从 StandardManager 的 sessions 集合中取出 StandardSession 对象。由于一个 requestedSessionId 对应一个访问的客户端，所以一个客户端也就对应一个 StandardSession 对象，这个对象正是保存我们创建的 Session 值的。下面我们看一下 StandardManager 这个类是如何管理 StandardSession 的生命周期的。

图 10-6 是 StandardManager 与 StandardSession 的类关系图。

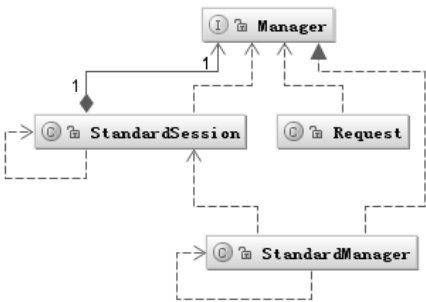


图 10-6 StandardManager 与 StandardSession 的类关系图

StandardManager 类负责 Servlet 容器中所有的 StandardSession 对象的生命周期管理。当 Servlet 容器重启或关闭时，StandardManager 负责持久化没有过期的 StandardSession 对象，它会将所有的 StandardSession 对象持久化到一个以“SESSIONS.ser”为文件名的文件中。到 Servlet 容器重启时，也就是 StandardManager 初始化时，它会重新读取这个文件，解析出所有 Session 对象，重新保存在 StandardManager 的 sessions 集合中。Session 的恢复状态图如图 10-7 所示。



图 10-7 Session 恢复状态图

当 Servlet 容器关闭时 StandardManager 类会调用 unload 方法将 sessions 集合中的 StandardSession 对象写到“SESSIONS.ser”文件中，然后在启动时再按照上面的状态图重新恢复，注意要持久化保存 Servlet 容器中的 Session 对象，必须调用 Servlet 容器的 stop

和 start 命令，而不能直接结束（kill）Servlet 容器的进程。因为直接结束进程，Servlet 容器没有机会调用 unload 方法来持久化这些 Session 对象。

另外，在 StandardManager 的 sessions 集合中的 StandardSession 对象并不是永远保存的，否则 Servlet 容器的内存将很容易被消耗尽，所以必须给每个 Session 对象定义一个有效时间，超过这个时间则 Session 对象将被清除。在 Tomcat 中这个有效时间是 60s（maxInactiveInterval 属性控制），超过 60s 该 Session 将会过期。检查每个 Session 是否失效是在 Tomcat 的一个后台线程中完成的（backgroundProcess()方法中）。过期 Session 的状态图如图 10-8 所示。

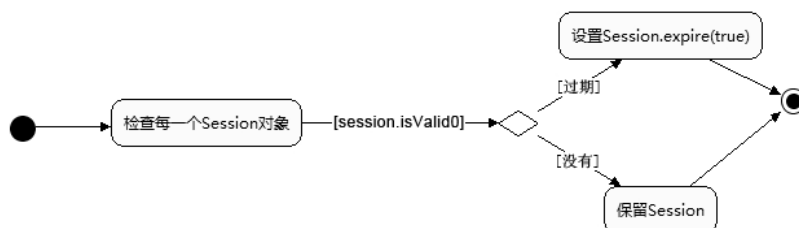


图 10-8 过期 Session 状态图

除了后台进程检查 Session 是否失效外，当调用 request.getSession()时也会检查该 Session 是否过期。值得注意的是，request.getSession()方法调用的 StandardSession 永远都会存在，即使与这个客户端关联的 Session 对象已经过期。如果过期，则又会重新创建一个全新的 StandardSession 对象，但是以前设置的 Session 值将会丢失。如果你取到了 Session 对象，但是通过 session.getAttribute 取不到前面设置的 Session 值，请不要奇怪，因为很可能它已经失效了，请检查一下 <Manager pathname="" maxInactiveInterval="60" /> 中 maxInactiveInterval 配置项的值，如果不想让 Session 过期则可以设置为-1。但是你要仔细评估一下，网站的访问量和设置的 Session 的大小，防止将你的 Servlet 容器内存撑爆。如果不想自动创建 Session 对象，也可以通过 request.getSession(boolean create)方法来判断与该客户端关联的 Session 对象是否存在。

10.3 Cookie 安全问题

虽然 Cookie 和 Session 都可以跟踪客户端的访问记录，但是它们的工作方式显然是不同的，Cookie 通过把所有要保存的数据通过 HTTP 的头部从客户端传递到服务端，又从服

务端再传回到客户端，所有的数据都存储在客户端的浏览器里，所以这些 Cookie 数据可以被访问到，就像我们前面通过 Firefox 的插件 HttpFox 可以看到所有的 Cookie 值。不仅可以查看 Cookie，甚至可以通过 Firecookie 插件添加、修改 Cookie，所以 Cookie 的安全性受到了很大的挑战。

相比较而言 Session 的安全性要高很多，因为 Session 是将数据保存在服务端，只是通过 Cookie 传递一个 SessionID 而已，所以 Session 更适合存储用户隐私和重要的数据。

10.4 分布式 Session 框架

从前面的分析可知，Session 和 Cookie 各自有优点和缺点。在大型互联网系统中，单独使用 Cookie 和 Session 都是不可行的，原因很简单。因为如果使用 Cookie，则可以很好地解决应用的分布式部署问题，大型互联网应用系统的一个应用有上百台机器，而且有很多不同的应用系统协同工作，由于 Cookie 是将值存储在客户端的浏览器里，用户每次访问都会将最新的值带回给处理该请求的服务器，所以也就解决了同一个用户的请求可能不在同一台服务器处理而导致的 Cookie 不一致的问题。

10.4.1 存在哪些问题

这种“谁家的孩子谁抱走”的处理方式的确是大型互联网的一个比较简单但的确可以解决问题的处理方式，但是这种处理方式也会带来了很多其他问题，如下所述。

- ◎ 客户端 Cookie 存储限制。随着应用系统的增多，Cookie 数量也快速增加，但浏览器对于用户 Cookie 的存储是有限制的。例如，对 IE7 之前的 IE 浏览器，Cookie 个数的限制是 20 个；而对后续的版本，包括 Firefox 等，Cookie 个数的限制都是 50 个，总大小不超过 4KB，超过限制就会出现丢弃 Cookie 的现象，这会严重影响应用系统的正常使用。
- ◎ Cookie 管理的混乱。在大型互联网应用系统中，如果每个应用系统都自己管理每个应用使用的 Cookie，则会导致混乱，由于通常应用系统都在同一个域名下，Cookie 又有上面一条提到的限制，所以没有统一管理很容易出现 Cookie 超出限制的情况。

- ⊙ 安全令人担忧。虽然可以通过设置 `HttpOnly` 属性防止一些私密 Cookie 被客户端访问，但是仍然不能保证 Cookie 无法被篡改。为了保证 Cookie 的私密性通常会对 Cookie 进行加密，但是维护这个加密 Key 也是一件麻烦的事情，无法保证定期更新加密 Key 也是带来安全性问题的一个重要因素。

当我们对以上问题不能再容忍下去时，就不得不想其他办法处理了。

10.4.2 可以解决哪些问题

既然 Cookie 有以上问题，Session 也有它的好处，那么为何不结合使用 Session 和 Cookie 呢？下面是分布式 Session 框架可以解决的问题。

- ⊙ Session 配置的统一管理。
- ⊙ Cookie 使用的监控和统一规范管理。
- ⊙ Session 存储的多元化。
- ⊙ Session 配置的动态修改。
- ⊙ Session 加密 key 的定期修改。
- ⊙ 充分的容灾机制，保持框架的使用稳定性。
- ⊙ Session 各种存储的监控和报警支持。
- ⊙ Session 框架的可扩展性，兼容更多的 Session 机制如 `wapSession`。
- ⊙ 跨域名 Session 与 Cookie 如何共享的问题。现在同一个网站可能存在多个域名，如何将 Session 和 Cookie 在不同的域名之间共享是一个具有挑战性的问题。

10.4.3 总体实现思路

分布式 Session 框架的架构图如图 10-9 所示。

为了达成上面所说的几个目标，我们需要一个服务订阅服务器，在应用启动时可以从这个订阅服务器订阅这个应用需要的可写 Session 项和可写 Cookie 项，这些配置的 Session 和 Cookie 可以限制这个应用能够使用哪些 Session 和 Cookie，甚至可以控制 Session 和

Cookie 可读或者可写。这样可以精确地控制哪些应用可以操作哪些 Session 和 Cookie，可以有效控制 Session 的安全性和 Cookie 的数量。

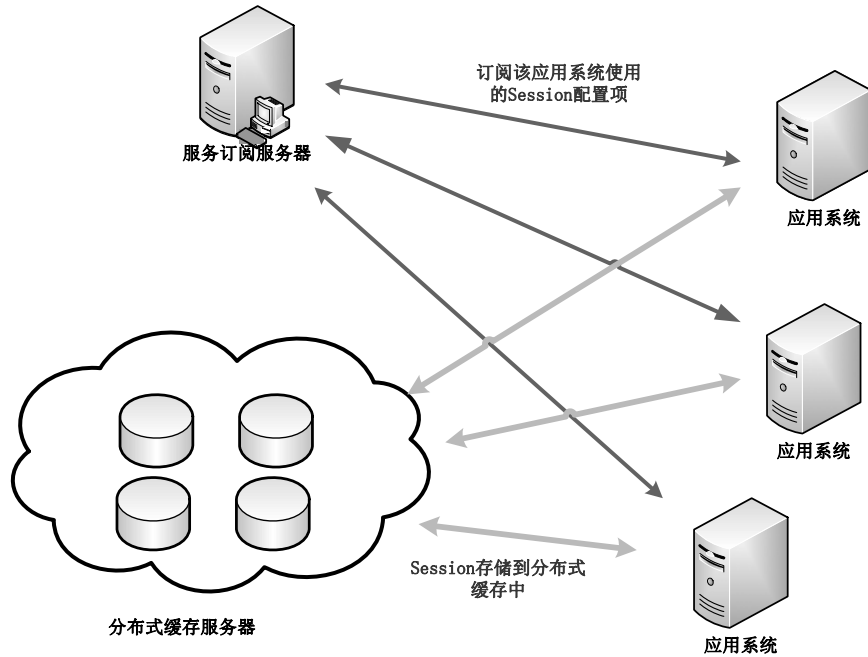


图 10-9 Session 框架的架构图

如 Session 的配置项可以为如下形式：

```
<session>
  <key>sessionID</key>
  <cookiekey>sessionID</cookiekey>
  <lifeCycle>9000</lifeCycle>
  <base64>true</base64>
</session>
```

Cookie 的配置可以为如下形式：

```
<cookie>
  <key>cookie</key>
  <lifeCycle></lifeCycle>
  <type>1</type>
  <path>/wp</path>
```

```
<domain>xulingbo.net</ domain>
<decrypt>false</decrypt>
<httpOnly>false</ httpOnly >
</cookie>
```

统一通过订阅服务器推送配置可以有效地集中管理资源，所以可以省去每个应用都来配置 Cookie，简化 Cookie 的管理。如果应用要使用一个新增的 Cookie，则可以通过一个统一的平台来申请，申请通过才将这个配置项增加到订阅服务器。如果是一个所有应用都要使用的全局 Cookie，那么只需将这个 Cookie 通过订阅服务器统一推送过去就行了，省去了要在每个应用中手动增加 Cookie 的配置。

关于这个订阅服务器现在有很多开源的配置服务器，如 Zookeeper 集群管理服务器，可以统一管理所有服务器的配置文件。

由于应用是一个集群，所以不可能将创建的 Session 都保存在每台应用服务器的内存中，因为如果每台服务器有几十万的访问用户，那么服务器的内存肯定不够用，即使内存够用，这些 Session 也无法同步到这个应用的所有服务器中。所以要共享这些 Session 必须将它们存储在一个分布式缓存中，可以随时写入和读取，而且性能要很好才能满足要求。当前能满足这个要求的系统有很多，如 MemCache 或者淘宝的开源分布式缓存系统 Tair 都是很好的选择。

解决了配置和存储问题，下面看一下如何存取 Session 和 Cookie。

既然是一个分布式 Session 的处理框架，必然会重新实现 HttpSession 的操作接口，使得应用操作 Session 的对象都是我们实现的 InnerHttpSession 对象，这个操作必须在进入应用之前完成，所以可以配置一个 filter 拦截用户的请求。

先看一下如何封装 HttpSession 对象和拦截请求，图 10-10 是时序图。

我们可以在应用的 web.xml 中配置一个 SessionFilter，用于在请求到达 MVC 框架之前封装 HttpServletRequest 和 HttpServletResponse 对象，并创建我们自己的 InnerHttpSession 对象，把它设置到 request 和 response 对象中。这样应用系统通过 request.getSession() 返回的就是我们创建的 InnerHttpSession 对象了，我们可以拦截 response 的 addCookies 设置的 Cookie。

在时序图中，应用创建的所有 Session 对象都会保存在 InnerHttpSession 对象中，当用户的这次访问请求完成时，Session 框架将会把这个 InnerHttpSession 的所有内容再更新到

分布式缓存中，以便于这个用户通过其他服务器再次访问这个应用系统。另外，为了保证一些应用对 Session 稳定性的特殊要求，可以将一些非常关键的 Session 再存储到 Cookie 中，如当分布式缓存存在问题时，可以将部分 Session 存储到 Cookie 中，这样即使分布式缓存出现问题也不会影响关键业务的正常运行。

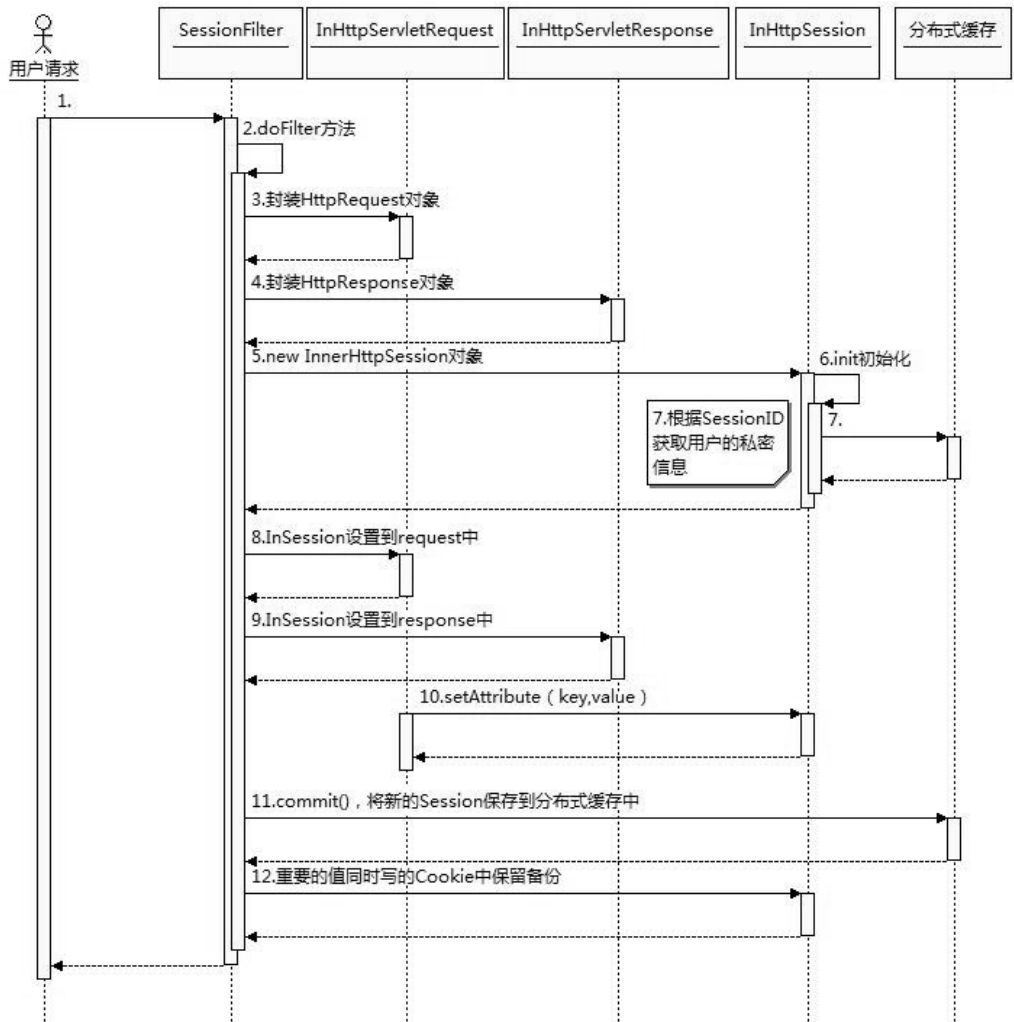


图 10-10 HttpSession 拦截请求时序图

还有一个非常重要的问题就是如何处理跨域名来共享 Cookie 的问题。我们知道 Cookie

是有域名限制的，也就是在一个域名下的 Cookie 不能被另一个域名访问，所以如果在一个域名下已经登录成功，如何访问到另外一个域名的应用且保证登录状态仍然有效，对这个问题大型网站应该经常会遇到。如何解决这个问题呢？下面介绍一种处理方式，如图 10-11 所示。

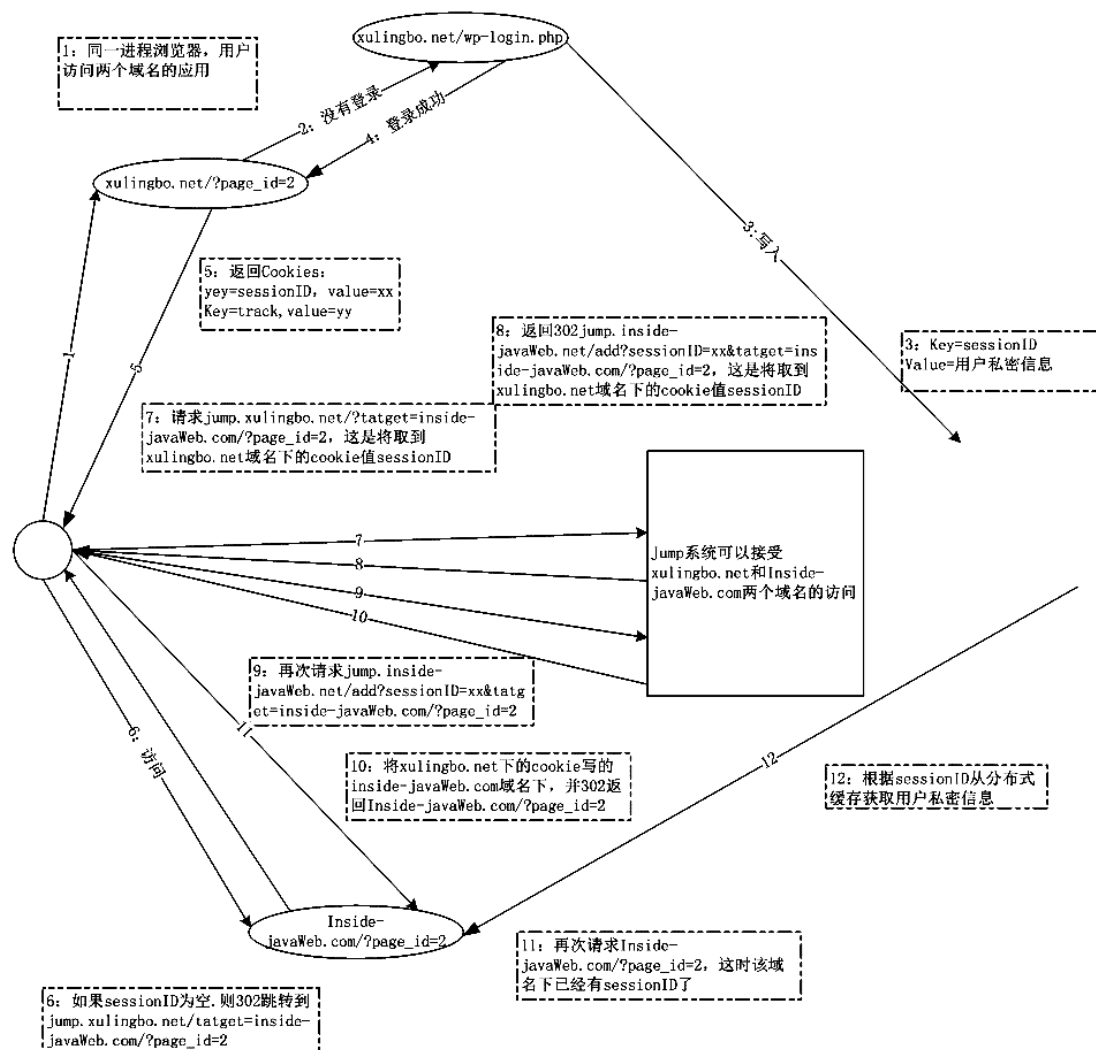


图 10-11 跨域名同步 session

从图中可以看出，要实现 Session 同步，需要另外一个跳转应用，这个应用可以被一个或者多个域名访问，它的主要功能是从一个域名下取得 sessionID，然后将这个 sessionID

同步到另外一个域名下。这个 `sessionID` 其实就是一个 `Cookie`，相当于我们经常遇到的 `JSESSIONID`，所以要实现两个域名下的 `Session` 同步，必须要将同一个 `sessionID` 作为 `Cookie` 写到两个域名下。

总共 12 步，一个域名不用登录就取到了另外一个域名下的 `Session`，当然这中间有些步骤还可以简化，也可以做一些额外的工作，如可以写一些需要的 `Cookie`，而不仅仅是传一个 `sessionID`。

除此之外，该框架还能处理 `Cookie` 被盗取的问题。如你的密码没有丢失，但是你的账号却有可能被别人登录的情况，这种情况很可能就是因为你登录成功后，你的 `Cookie` 被别人盗取了，盗取你的 `Cookie` 的人将你的 `Cookie` 加入到他的浏览器，然后他就可以通过你的 `Cookie` 正常访问你的个人信息了，这是一个非常严重的问题。在这个框架中我们可以设置一个 `Session` 签名，当用户登录成功后我们根据用户的私密信息生成的一个签名，以表示当前这个唯一的合法登录状态，然后将这个签名作为一个 `Cookie` 在当前这个用户的浏览器进程中和服务器传递，用户每次访问服务器都会检查这个签名和从服务端分布式缓存中取得的 `Session` 重新生成的签名是否一致，如果不一致，则显然这个用户的登录状态不合法，服务端将清除这个 `sessionID` 在分布式缓存中的 `Session` 信息，让用户重新登录。

10.5 Cookie 压缩

`Cookie` 在 HTTP 的头部，所以通常的 `gzip` 和 `deflate` 针对 HTTP Body 的压缩不能压缩 `Cookie`，如果 `Cookie` 的量非常大，则可以考虑将 `Cookie` 也做压缩，压缩方式是将 `Cookie` 的多个 `k/v` 对看成普通的文本，做文本压缩。压缩算法同样可以使用 `gzip` 和 `deflate` 算法，但是需要注意的一点是，根据 `Cookie` 的规范，在 `Cookie` 中不能包含控制字符，仅能包含 ASCII 码为 34~126 的可见字符。所以要将压缩后的结果再进行转码，可以进行 `Base32` 或者 `Base64` 编码。

可以配置一个 `Filter` 在页面输出时对 `Cookie` 进行全部或者部分压缩，如下代码所示：

```
private void compressCookie(Cookie c, HttpServletResponse res) {
    try {
        ByteArrayOutputStream bos = null;
        bos = new ByteArrayOutputStream();
        DeflaterOutputStream dos = new DeflaterOutputStream(bos);
```

```

        dos.write(c.getValue().getBytes());
        dos.close();
        System.out.println("before compress length:" + c.getValue().
getBytes().length);
        String compress = new sun.misc.BASE64Encoder().encode(bos.
toByteArray());
        res.addCookie(new Cookie("compress", compress));
        System.out.println("after compress length:" + compress.getBytes().
length);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

上面的代码是用 `DeflaterOutputStream` 对 Cookie 进行压缩的, `Deflater` 压缩后再进行 BASE64 编码, 相应地用 `InflaterInputStream` 进行解压。

```

private void unCompressCookie(Cookie c) {
    try {
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        byte[] compress = new sun.misc.BASE64Decoder().decodeBuffer(new
String(c.getValue().getBytes()));
        ByteArrayInputStream bis = new ByteArrayInputStream(compress);
        InflaterInputStream inflater = new InflaterInputStream(bis);
        byte[] b = new byte[1024];
        int count;
        while ((count = inflater.read(b)) >= 0) {
            out.write(b, 0, count);
        }
        inflater.close();
        System.out.println(out.toByteArray());
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

2KB 大小的 Cookie 在压缩前与压缩后的字节数相差 20% 左右, 如果你的网站的 Cookie 在 2~3KB 左右, 一天有 1 亿的 PV, 那么一天就能够产生 4TB 的带宽流量了, 从节省带宽成本来说压缩还是很有必要的。

10.6 表单重复提交问题

在网站中有很多地方都存在表单重复提交的问题，如用户在网速慢的情况下可能会重复提交表单，又如恶意用户通过程序来发送恶意请求等，这时都需要设计一个防止表单重复提交的机制。

要防止表单重复提交，就要标识用户的每一次访问请求，使得每一次访问对服务端来说都是唯一确定的。为了标识用户的每次访问请求，可以在用户请求一个表单域时增加一个隐藏表单项，这个表单项的值每次都是唯一的 token，如：

```
<form id="form" method="post">
<input type="hidden" name="csrf_token" value="xxxx" />
</form>
```

当用户在请求时生成这个唯一的 token 时，同时将这个 token 保存在用户的 Session 中，等用户提交请求时检查这个 token 和当前的 Session 中保存的 token 是否一致。如果一致，则说明没有重复提交，否则用户提交上来的 token 已经不是当前这个请求的合法 token。其工作过程如图 10-12 所示。

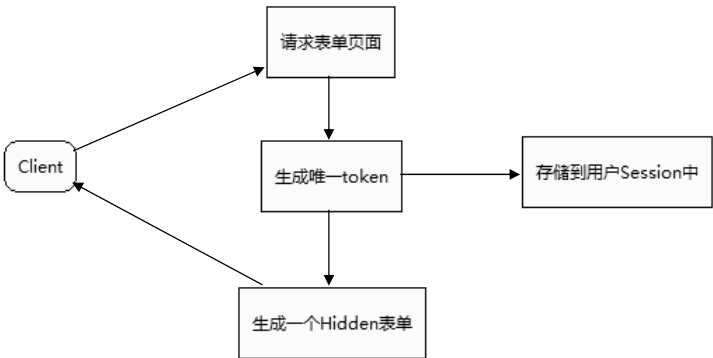


图 10-12 工作过程

图 10-12 是用户发起的对表单页面的请求过程，生成唯一的 token 需要一个算法，最简单的就是可以根据一个种子作为 key 生成一个随机数，并保存在 Session 中，等下次用户提交表单时做验证。验证表单的过程如图 10-13 所示。

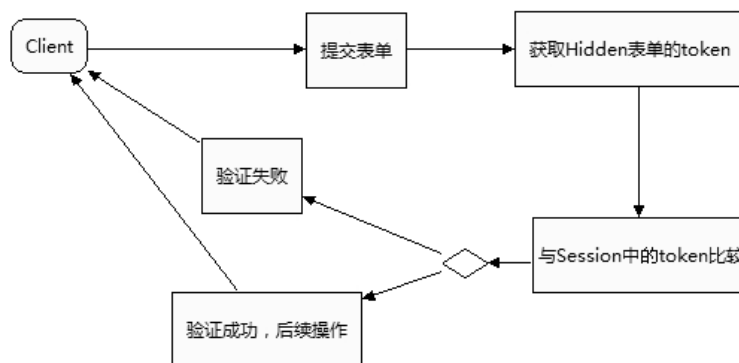


图 10-13 验证表单的过程

当用户提交表单时会将请求时生成的 token 带回来，这样就可以和在 Session 中保存的 token 做对比，从而确认这次表单验证是否合法。

10.7 多终端 Session 统一

当前大部分网站都有了无线端，对无线端的 Cookie 如何处理也是很多程序员必须考虑的问题。

在无线端发展初期，后端的服务系统未必和 PC 的服务系统是统一的，这样就涉及在一端调用多个系统时如何做到服务端 Session 共享的问题了。有两个明显的例子：一个是在无线端可能会通过手机访问无线服务端系统，同时也会访问 PC 端的服务系统，如果它们两个的登录系统没有统一的话，将会非常麻烦，可能会出现二次登录的情况；另一个是在手机上登录以后再在 PC 上同样访问服务端数据，Session 能否共享就决定了客户端是否要再次登录。

针对这两种情况，目前都有理想的解决方案。

1) 多端共享 Session

多端共享 Session 必须要做的工作是不管是无线端还是 PC 端，后端的服务系统必须统一会话架构，也就是两边的登录系统必须要基于一致的会员数据结构、Cookie 与 Session 的统一。也就是不管是 PC 端登录还是无线端登录，后面对应的数据结构和存储要统一，写到客户端的 Cookie 也必须一样，这是前提条件。

那么如何做到这一点？就是要按照我们在前面所说的实现分布式的 Session 框架。如下图 10-14 所示。

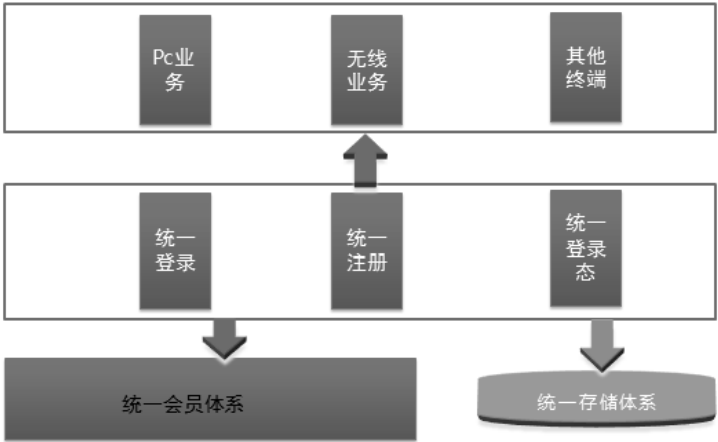


图 10-14 服务端统一 Session 示意图

上面服务端统一 Session 后，在同一个终端上不管是访问哪个服务端都能做到登录状态统一。例如不管是 Native 还是内嵌 Webview，都可以拿统一的 Session ID 去服务端验证登录状态。

2) 多终端登录

目前很多网站都会出现无线端和 PC 端多端登录的情况，例如可以通过扫码登录等。这些是如何实现的呢？其实比较简单，如图 10-15 所示。

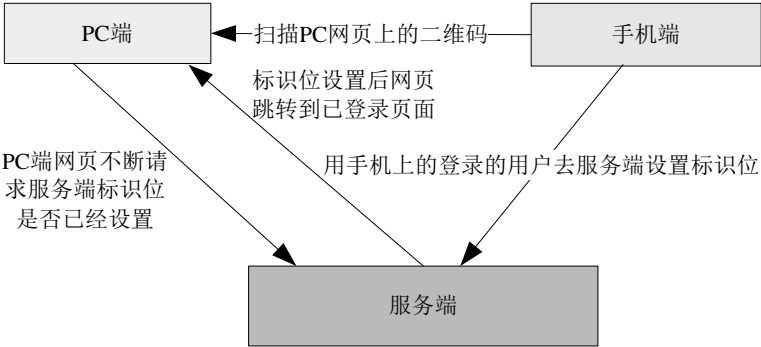


图 10-15 多终端登录示意图

这里手机端在扫码之前必须是已经登录的状态，因为这样才能获取到底是谁将要登录的信息，同时扫码的二维码也带有一个特定的标识，标识是这个客户端通过手机端登录了。当手机端扫码成功后，会在服务端设置这个二维码对应的标识为已经登录成功，这时 PC 客户端会通过将“心跳”请求发送到服务端，来验证是否已经登录成功，这样就成为一种便捷的登录方式。

10.8 总结

Cookie 和 Session 都是为了保持用户访问的连续状态，之所以要保持这种状态，一方面是为了方便业务实现，另一方面就是简化服务端的程序设计，提高访问性能，但是这也带来了另外一些挑战，例如安全问题、应用的分布式部署带来的 Session 的同步问题及跨域名 Session 的同步问题等。本章分析了 Cookie 和 Session 的工作原理，并介绍了一种分布式 Session 的解决方案。