tomcat配置及调优

## appBase和docBase的区别

server.xml文件中appBase和docBase配置示例如下：

|  |
| --- |
| <Host name="localhost" appBase="webapps">  <Context docBase="D:\WebContent"/>  </Host> |

appBase：这个目录下面的子目录将自动被部署为应用，且war文件将被自动解压缩并部署为应用，默认为tomcat下webapps目录。

appBase采用相对路径时，其父路径是tomcat的安装目录。如：appBase="\webapps"或appBase="webapps"相当于appBase="tomcat路径\webapps\"。

docBase采用相对路径时，其父路径是appBase的路径。

http://localhost:8080/默认访问上面路径下的ROOT文件夹中的项目。

## 线程池配置

## 常用元素说明

Server元素：它代表整个容器,是Tomcat实例的顶层元素.由org.apache.catalina.Server接口来定义.它包含一个或多个<Service>元素.并且它不能做为任何元素的子元素，同时需要指定关闭端口。

Service元素：该元素由org.apache.catalina.Service接口定义,它包含一个<Engine>元素,以及一个或多个<Connector>,这些Connector元素共享用同一个Engine元素。

Connector元素：由Connector接口定义.<Connector>元素代表与客户程序实际交互的给件,它负责接收客户请求,以及向客户返回响应结果。

Engine元素：每个Service元素只能有一个Engine元素.处理在同一个<Service>中所有<Connector>元素接收到的客户请求.由org.apahce.catalina.Engine接口定义。

Host元素：它由Host接口定义.一个Engine元素可以包含多个<Host>元素.每个<Host>的元素定义了一个虚拟主机.它包含了一个或多个Web应用。

Context元素：它由Context接口定义.是使用最频繁的元素.每个<Context元素代表了运行在虚拟主机上的单个Web应用.一个<Host>可以包含多个<Context>元素.每个web应用有唯一的一个相对应的Context代表web应用自身。

Tomcat Server处理一个http请求的过程：假设来自客户的请求为：<http://localhost:8080/wsota/wsota_index.jsp>

1) 请求被发送到本机端口8080，被在那里侦听的Coyote HTTP/1.1 Connector获得2) Connector把该请求交给它所在的Service的Engine来处理，并等待来自Engine的回应

3) Engine获得请求localhost/wsota/wsota\_index.jsp，匹配它所拥有的所有虚拟主机Host

4) Engine匹配到名为localhost的Host（即使匹配不到也把请求交给该Host处理，因为该Host被定义为该Engine的默认主机）

5) localhost Host获得请求/wsota/wsota\_index.jsp，匹配它所拥有的所有Context  
6) Host匹配到路径为/wsota的Context（如果匹配不到就把该请求交给路径名为""的Context去处理）

7) path="/wsota"的Context获得请求/wsota\_index.jsp，在它的mapping table中寻找对应的servlet

8) Context匹配到URL PATTERN为\*.jsp的servlet，对应于JspServlet类

9) 构造HttpServletRequest对象和HttpServletResponse对象，作为参数调用JspServlet的doGet或doPost方法

10)Context把执行完了之后的HttpServletResponse对象返回给Host

11)Host把HttpServletResponse对象返回给Engine

12)Engine把HttpServletResponse对象返回给Connector

13)Connector把HttpServletResponse对象返回给客户browser

## Connector 连接器的配置

Tomcat 连接器的三种方式： bio、nio 和 apr，三种方式性能差别很大，apr 的性能最优， bio 的性能最差。而 Tomcat 7 使用的 Connector  默认就启用的 Apr 协议，但需要系统安装 Apr 库，否则就会使用 bio 方式。

## 打开共享的线程池

打开 ${TOMCAT\_HOME}/conf/server.xml，如下：

|  |
| --- |
| <Executor name="tomcatThreadPool" namePrefix="catalina-exec-"maxThreads="200" minSpareThreads="10" maxIdleTime="60000”maxSpareThreads ="100"/> |

name：共享线程池的名字。这是Connector为了共享线程池要引用的名字，该名字必须唯一。默认值：None；

namePrefix:在JVM上，每个运行线程都可以有一个name 字符串。这一属性为线程池中每个线程的name字符串设置了一个前缀，Tomcat将把线程号追加到这一前缀的后面。默认值：tomcat-exec-；

maxThreads :Tomcat 使用线程来处理接收的每个请求，这个值表示 Tomcat 可创建的最大的线程数，默认值是 200

maxIdleTime：在[Tomcat](http://www.gootry.com/wangzhuan/lsorts-article?wzmkind=Apache+Tomcat%E6%9C%8D%E5%8A%A1%E5%99%A8%E9%85%8D%E7%BD%AE%E4%BD%BF%E7%94%A8&wzkind=%E9%85%8D%E7%BD%AE%E6%96%87%E4%BB%B6" \t "_blank)关闭一个空闲线程之前，允许空闲线程持续的时间(以毫秒为单位)。只有当前活跃的线程数大于minSpareThread的值，才会关闭空闲线程。默认值：60000(一分钟)。

minSpareThreads：最小空闲线程数，Tomcat 启动时的初始化的线程数，表示即使没有人使用也开这么多空线程等待，默认值是 10。

maxSpareThreads：最大备用线程数，一旦创建的线程超过这个值，Tomcat 就会关闭不再需要的 socket 线程。此配置在tomcat7已经去掉。

上边配置的参数，最大线程 500（一般服务器足以），要根据自己的实际情况合理设置，设置越大会耗费内存和 CPU，因为 CPU 疲于线程上下文切换，没有精力提供请求服务了，最小空闲线程数 20，线程最大空闲时间 60 秒，当然允许的最大线程连接数还受制于操作系统的内核参数设置，设置多大要根据自己的需求与环境。当然线程可以配置在“tomcatThreadPool”中，也可以直接配置在“Connector”中，但不可以重复配置。

## 使用共享的线程池

修改<Connector …>节点，增加 executor 属性，搜索【port="8080"】：

|  |
| --- |
| <Connector executor="tomcatThreadPool"                 port="8080" protocol="HTTP/1.1"                 URIEncoding="UTF-8"                 connectionTimeout="30000"                 enableLookups="false"                 disableUploadTimeout="false"                 connectionUploadTimeout="150000"                 acceptCount="300"                 keepAliveTimeout="120000"                 maxKeepAliveRequests="1"                 compression="on"                 compressionMinSize="2048"                 compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain,image/gif,image/jpg,image/png"                 redirectPort="8443" /> |

executor：表示使用该参数值对应的线程池；

maxConnections：最大连接数，这个数值取决于你使用堵塞还是非堵塞连接器，堵塞的是BIO, 那么这个数值是BIO的最大线程数，如果是NIO，Java非堵塞I/O连接器，那么这个数值是1000，如果使用C原生非堵塞连接器APR，那么这个数值是8192。那么如何知道我们系统使用哪个连接器类型呢？缺省是在APR和BIO之间切换，如果在启动运行时，你配置编译了APR，APR库包被Tomcat发现，那么就采取APR连接类型，否则就是要BIO。APR虽然好，但是稳定性不是很强，其SSL使用OpenSSL，而使用NIO，可以支持Java自身的SSL。当连接数达到最大值后，系统会继续接收连接但不会超过acceptCount的值。

acceptCount：指定当所有可以使用的处理请求的线程数都被使用时，可传入连接请求的最大队列长度，超过这个数的请求将不予处理，默认为100个。

URIEncoding：指定 Tomcat 容器的 URL 编码格式，语言编码格式这块倒不如其它 WEB 服务器软件配置方便，需要分别指定。

connnectionTimeout： 网络连接超时，单位：毫秒，设置为 0 表示永不超时，这样设置有隐患的。通常可设置为 30000 毫秒，可根据检测实际情况，适当修改。前端ajax请求我们需要设置超时时间，在你将这个时间设置很大的时候，发现无效，请注意ajax中设置的时间是不是超过了tomcat的这个连接超时时间。

enableLookups： 是否反查域名，以返回远程主机的主机名，取值为：true 或 false，如果设置为false，则直接返回IP地址，为了提高处理能力，应设置为 false。

disableUploadTimeout：上传时是否使用超时机制。

connectionUploadTimeout：上传超时时间，毕竟文件上传可能需要消耗更多的时间，这个根据你自己的业务需要自己调，以使Servlet有较长的时间来完成它的执行，需要与上一个参数一起配合使用才会生效。

keepAliveTimeout：长连接最大保持时间（毫秒），表示在下次请求过来之前，Tomcat 保持该连接多久，默认是使用 connectionTimeout 时间，-1 为不限制超时。

maxKeepAliveRequests：每个TCP连接接受最大的Http请求数目，当处理一个keep alive请求达到这个最大值，Tomcat关闭这个连接，1表示禁用，-1表示不限制个数，默认100个，一般设置在100~200之间。这里可以去百度了解长连接的知识。

compression：是否对响应的数据进行 GZIP 压缩，off：表示禁止压缩；on：表示允许压缩（文本将被压缩）、force：表示所有情况下都进行压缩，默认值为off，压缩数据后可以有效的减少页面的大小，一般可以减小1/3左右，节省带宽。

compressionMinSize：表示压缩响应的最小值，只有当响应报文大小大于这个值的时候才会对报文进行压缩，如果开启了压缩功能，默认值就是2048。

compressableMimeType：压缩类型，指定对哪些类型的文件进行数据压缩。

noCompressionUserAgents="gozilla, traviata"： 对于以下的浏览器，不启用压缩。

processorCache：tomcat在处理每个连接时，Acceptor角色负责将socket上下文封装为一个任务SocketProcessor然后提交给线程池处理。在BIO和APR模式下，每次有新请求时，会创建一个新的SocketProcessor实例。而在NIO里，为了追求性能，对SocketProcessor也做了cache，用完后将对象状态清空然后放入cache，下次有新的请求过来先从cache里获取对象，获取不到再创建一个新的。如果不是使用servlet3.0，官方建议该数值配置为maxThreads的值。

如果已经对代码进行了动静分离，静态页面和图片等数据就不需要 Tomcat 处理了，那么也就不需要配置在 Tomcat 中配置压缩了。更多配置请参考附录中的官方文档。

## JVM调优

## JVM参数配置方法

Tomcat的启动参数位于安装目录 ${TOMCAT\_HOME}/bin目录下，Linux操作系统就是 catalina.sh文件。JAVA\_OPTS就是用来设置JVM相关运行参数的变量，还可以在CATALINA\_OPTS变量中设置。关于这2个变量，还是多少有些区别的：

JAVA\_OPTS：用于当 Java运行时选项“start”、“stop”或“run”命令执行。

CATALINA\_OPTS：用于当 Java 运行时选项“start”或“run”命令执行。

为什么有两个不同的变量？它们之间都有什么区别呢？

首先，在启动 Tomcat 时，任何指定变量的传递方式都是相同的，可以传递到执行“start”或“run”命令中，但只有设定在 JAVA\_OPTS 变量里的参数被传递到“stop”命令中。对于 Tomcat 运行过程，可能没什么区别，影响的是结束程序，而不是启动程序。

## JVM 参数属性

32 位系统下 JVM 对内存的限制：不能突破 2GB ，那么这时你的 Tomcat 要优化，就要讲究点技巧了，而在 64 位操作系统上无论是系统内存还是 JVM 都没有受到 2GB 这样的限制。

针对于 JMX 远程监控也是在这里设置，以下为 64 位系统环境下的配置，内存加入的参数如下：

|  |
| --- |
| CATALINA\_OPTS="  -server  -Xms6000M  -Xmx6000M  -Xss512k  -XX:NewSize=2250M  -XX:MaxNewSize=2250M  -XX:PermSize=128M  -XX:MaxPermSize=256M  -XX:+AggressiveOpts  -XX:+UseBiasedLocking  -XX:+DisableExplicitGC  -XX:+UseParNewGC  -XX:+UseConcMarkSweepGC  -XX:MaxTenuringThreshold=31  -XX:+CMSParallelRemarkEnabled  -XX:+UseCMSCompactAtFullCollection  -XX:LargePageSizeInBytes=128m  -XX:+UseFastAccessorMethods  -XX:+UseCMSInitiatingOccupancyOnly  -Duser.timezone=Asia/Shanghai  -Djava.awt.headless=true" |

为了看着方便，将每个参数单独写一行。上面参数好多啊，可能有人写到现在都没见过一个在 Tomcat 的启动命令里加了这么多参数，当然，这些参数只是我机器上的，不一定适合你，尤其是参数后的 value（值）是需要根据你自己的实际情况来设置的。

-server：一定要作为第一个参数，在多个CPU时性能佳，还有一种叫client的模式，特点是启动速度比较快，但运行时性能和内存管理效率不高，通常用于客户端应用程序或开发调试，在32位环境下直接运行Java程序默认启用该模式。Server模式的特点是启动速度比较慢，但运行时性能和内存管理效率很高，适用于生产环境，在具有64位能力的JDK环境下默认启用该模式，可以不配置该参数。具体参考：<https://docs.oracle.com/javase/8/docs/technotes/guides/vm/server-class.html>

-Xms：表示Java初始化堆的大小，-Xms 与-Xmx 设成一样的值，避免 JVM 反复重新申请内存，导致性能大起大落，默认值为物理内存的1/64，默认（MinHeapFreeRatio参数可以调整）空余堆内存小于40%时，JVM 就会增大堆直到-Xmx的最大限制。

-Xss：表示每个Java线程堆栈大小，JDK 5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。根据应用的线程所需内存大小进行调整，在相同物理内存下，减小这个值能生成更多的线程，但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。一般小的应用， 如果栈不是很深，应该是128k够用的，大的应用建议使用256k或512K，一般不易设置超过1M，要不然容易出现out ofmemory。这个选项对性能影响比较大，需要严格的测试。

-XX:NewSize：设置新生代内存大小。

-XX:MaxNewSize：设置最大新生代新生代内存大小

-XX:PermSize：设置持久代内存大小

-XX:MaxPermSize：设置最大值持久代内存大小，永久代不属于堆内存，堆内存只包含新生代和老年代。

-XX:+AggressiveOpts：作用如其名（aggressive），启用这个参数，则每当JDK版本升级时，你的JVM都会使用最新加入的优化技术（如果有的话）。

-XX:+UseBiasedLocking：启用一个优化了的线程锁，我们知道在我们的appserver，每个http请求就是一个线程，有的请求短有的请求长，就会有请求排队的现象，甚至还会出现线程阻塞，这个优化了的线程锁使得你的appserver内对线程处理自动进行最优调配。

-XX:+DisableExplicitGC：在 程序代码中不允许有显示的调用“System.gc()”。每次在到操作结束时手动调用System.gc() 一下，付出的代价就是系统响应时间严重降低，就和关于Xms，Xmx里的解释的原理一样，这样去调用GC导致系统的JVM大起大落。

-XX:+UseConcMarkSweepGC：设置年老代为并发收集，即CMS gc，这一特性只有 jdk1.5后续版本才具有的功能，它使用的是gc估算触发和heap占用触发。我们知道频频繁的GC会造面 JVM的大起大落从而影响到系统的效率，因此使用了CMS GC后可以在GC次数增多的情况下，每次GC的响应时间却很短，比如说使用了 CMS  
GC 后经过 jprofiler 的观察，GC被触发次数非常多，而每次GC耗时仅为几毫秒。

-XX:+UseParNewGC：对新生代采用多线程并行回收，这样收得快，注意最新的 JVM 版本，当使用-XX:+UseConcMarkSweepGC时，-XX:UseParNewGC会自动开启。因此，如果年轻代的并行GC不想开启，可以通过设置-XX：-UseParNewGC来关掉。

-XX:MaxTenuringThreshold：设置垃圾最大年龄。如果设置为0的话，则新生代对象不经过 Survivor 区，直接进入老年代。对于老年代比较多的应用（需要大量常驻内存的应用），可以提高效率。如果将此值设置为一个较大值，则新生代对象会在Survivor 区进行多次复制，这样可以增加对象在新生代的存活时间，增加在新生代即被回收的概率，减少Full GC的频率，这样做可以在某种程度上提高服务稳定性。该参数只有在串行GC时才有效，这个值的设置是根据本地的jprofiler监控后得到的一个理想的值，不能一概而论原搬照抄。

-XX:+CMSParallelRemarkEnabled：在使用UseParNewGC的情况下，尽量减少 mark的时间。

-XX:+UseCMSCompactAtFullCollection：在使用concurrent gc的情况下，防止 memoryfragmention，对live object进行整理，使memory碎片减少。

-XX:LargePageSizeInBytes：指定Java heap的分页页面大小，内存页的大小不可设置过大， 会影响Perm的大小。

-XX:+UseFastAccessorMethods：使用get，set方法转成本地代码，原始类型的快速优化。

-XX:+UseCMSInitiatingOccupancyOnly：只有在oldgeneratio在使用了初始化的比例后concurrent collector启动收集。

-Duser.timezone=Asia/Shanghai：设置用户所在时区。

-Djava.awt.headless=true：这个参数一般我们都是放在最后使用的，这全参数的作用是这样的，有时我们会在我们的J2EE工程中使用一些图表工具如：jfreechart，用于在web网页输出 GIF/JPG 等流，在winodws 环境下，一般我们的app server在输出图形时不会碰到什么问题，但是在linux/unix 环境下经常会碰到一个exception 导致你在winodws开发环境下图片显示的好好可是在linux/unix下却显示不出来，因此加上这个参数以免避这样的情况出现。

-Xmn：新生代的内存空间大小，注意：此处的大小是（eden+ 2 survivor space)。与jmap -heap中显示的New gen是不同的。整个堆大小=新生代大小+老生代大小 +永久代大小。在保证堆大小不变的情况下，增大新生代后，将会减小老生代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。

-XX:CMSInitiatingOccupancyFraction：当堆满之后，并行收集器便开始进行垃圾收集，例如，当没有足够的空间来容纳新分配或提升的对象。对于CMS收集器，长时间等待是不可取的，因为在并发垃圾收集期间应用持续在运行（并且分配对象）。因此，为了在应用程序使用完内存之前完成垃圾收集周期，CMS收集器要比并行收集器更先启动。因为不同的应用会有不同对象分配模式，JVM会收集实际的对象分配（和释放）的运行时数据，并且分析这些数据，来决定什么时候启动一次CMS垃圾收集周期。这个参数设置有很大技巧，基本上满足(Xmx-Xmn)\*(100-CMSInitiatingOccupancyFraction)/100>=Xmn 就不会出现 promotion failed。例如在应用中Xmx是6000，Xmn是512，那么 Xmx-Xmn 是 5488M，也就是老年代有5488M，CMSInitiatingOccupancyFraction=90 说明老年代到90% 满的时候开始执行对老年代的并发垃圾回收（CMS），这时还剩10%的空间是5488\*10% = 548M，所以即使Xmn（也就是新生代共512M）里所有对象都搬到老年代里，548M的空间也足够了，所以只要满足上面的公式，就不会出现垃圾回收时的promotion failed，因此这个参数的设置必须与Xmn关联在一起。

-XX:+CMSIncrementalMode：该标志将开启CMS收集器的增量模式。增量模式经常暂停CMS过程，以便对应用程序线程作出完全的让步。因此，收集器将花更长的时间完成整个收集周期。因此只有通过测试后发现正常CMS周期对应用程序线程干扰太大时，才应该使用增量模式。由于现代服务器有足够的处理器来适应并发的垃圾收集，所以这种情况发生得很少，用于但CPU情况。

-XX:NewRatio：年轻代（包括Eden和两个Survivor区）与年老代的比值（除去持久代），-XX:NewRatio=4 表示年轻代与年老代所占比值为 1:4，年轻代占整个堆栈的 1/5，Xms=Xmx 并且设置了Xmn的情况下，该参数不需要进行设置。

-XX:SurvivorRatio：Eden区与Survivor区的大小比值，设置为8，表示2个Survivor 区（JVM堆内存年轻代中默认有2个大小相等的Survivor区）与1个Eden区的比值为 2:8，即1个Survivor区占整个年轻代大小的 1/10。

-XX:+UseSerialGC：设置串行收集器。

-XX:+UseParallelGC：设置为并行收集器。此配置仅对年轻代有效。即年轻代使用并行收集，而年老代仍使用串行收集。

-XX:+UseParallelOldGC：配置年老代垃圾收集方式为并行收集，JDK6.0开始支持对年老代并行收集。

-XX:ConcGCThreads：早期JVM版本也叫-XX:ParallelCMSThreads，定义并发CMS 过程运行时的线程数。比如value=4意味着CMS周期的所有阶段都以4个线程来执行。尽管更多的线程会加快并发CMS过程，但其也会带来额外的同步开销。因此，对于特定的应用程序，应该通过测试来判断增加CMS线程数是否真的能够带来性能的提升。如果还标志未设置，JVM会根据并行收集器中的-XX:ParallelGCThreads参数的值来计算出默认的并行CMS线程数。

-XX:ParallelGCThreads：配置并行收集器的线程数，即：同时有多少个线程一起进行垃圾回收，此值建议配置与 CPU 数目相等。

-XX:OldSize：设置JVM启动分配的老年代内存大小

-XX:NewSize：设置新生代内存的初始大小 。

以上就是一些常用的配置参数，有些参数是可以被替代的，配置思路需要考虑的是Java 提供的垃圾回收机制。虚拟机的堆大小决定了虚拟机花费在收集垃圾上的时间和频度。收集垃圾能够接受的速度和应用有关，应该通过分析实际的垃圾收集的时间和频率来调整。假如堆的大小很大，那么完全垃圾收集就会很慢，但是频度会降低。假如您把堆的大小和内存的需要一致，完全收集就很快，但是会更加频繁。调整堆大小的的目的是最小化垃圾收集的时间，以在特定的时间内最大化处理客户的请求。在基准测试的时候，为确保最好的性能，要把堆的大小设大，确保垃圾收集不在整个基准测试的过程中出现。

假如系统花费很多的时间收集垃圾，请减小堆大小。一次完全的垃圾收集应该不超过 3-5秒。假如垃圾收集成为瓶颈，那么需要指定代的大小，检查垃圾收集的周详输出，研究垃圾收集参数对性能的影响。当增加处理器时，记得增加内存，因为分配能够并行进行，而垃圾收集不是并行的。

## 设置系统属性

之前说过，Tomcat的语言编码，配置起来很慢，要经过多次设置才可以了，否则中文很有可能出现乱码情况。譬如汉字“中”，以UTF-8编码后得到的是3字节的值 %E4%B8%AD，然后通过GET或者POST方式把这3个字节提交到Tomcat容器，如果你不告诉Tomcat我的参数是用UTF-8编码的，那么Tomcat就认为你是用 ISO-8859-1来编码的，而ISO8859-1（兼容URI中的标准字符集US-ASCII）是兼容 ASCII的单字节编码并且使用了单字节内的所有空间，因此Tomcat就以为你传递的用 ISO-8859-1字符集编码过的3个字符，然后它就用ISO-8859-1来解码。

设置起来不难使用“ -D<名称>=<值> ”来设置系统属性：

-Djavax.servlet.request.encoding=UTF-8

-Djavax.servlet.response.encoding=UTF-8

-Dfile.encoding=UTF-8

-Duser.country=CN

-Duser.language=zh

## 常见的 Java 内存溢出有以下三种

（1） java.lang.OutOfMemoryError: Java heap space —-JVM Heap（堆）溢出

JVM 在启动的时候会自动设置 JVM Heap 的值，其初始空间（即-Xms）是物理内存的1/64，最大空间（-Xmx）不可超过物理内存。可以利用 JVM提供的-Xmn -Xms -Xmx等选项可进行设置。Heap的大小是Young Generation和Tenured Generaion 之和。在JVM中如果98％的时间是用于GC，且可用的Heap size不足2％的时候将抛出此异常信息。

解决方法：手动设置 JVM Heap（堆）的大小。    
（2） java.lang.OutOfMemoryError: PermGen space —-PermGen space溢出。

PermGen space的全称是Permanent Generation space，是指内存的永久保存区域。为什么会内存溢出，这是由于这块内存主要是被 JVM 存放Class 和 Meta 信息的，Class在被Load的时候被放入PermGen space区域，它和存放Instance的Heap 区域不同，sun的GC不会在主程序运行期对PermGen space进行清理，所以如果你的APP会载入很多CLASS的话，就很可能出现PermGen space溢出。

解决方法：手动设置MaxPermSize大小

（3） java.lang.StackOverflowError —-栈溢出

栈溢出了，JVM依然是采用栈式的虚拟机，这个和C与Pascal都是一样的。函数的调用过程都体现在堆栈和退栈上了。调用构造函数的“层”太多了，以致于把栈区溢出了。通常来讲，一般栈区远远小于堆区的，因为函数调用过程往往不会多于上千层，而即便每个函数调用需要1K的空间（这个大约相当于在一个C函数内声明了256个int类型的变量），那么栈区也不过是需要1MB的空间。通常栈的大小是1－2MB的。  
通常递归也不要递归的层次过多，很容易溢出。

## 附录

<https://tomcat.apache.org/tomcat-7.0-doc/config/http.html>

https://blog.csdn.net/ldx891113/article/details/51735171

<https://www.cnblogs.com/starhu/p/5599773.html>

https://blog.csdn.net/huaweitman/article/details/50552983