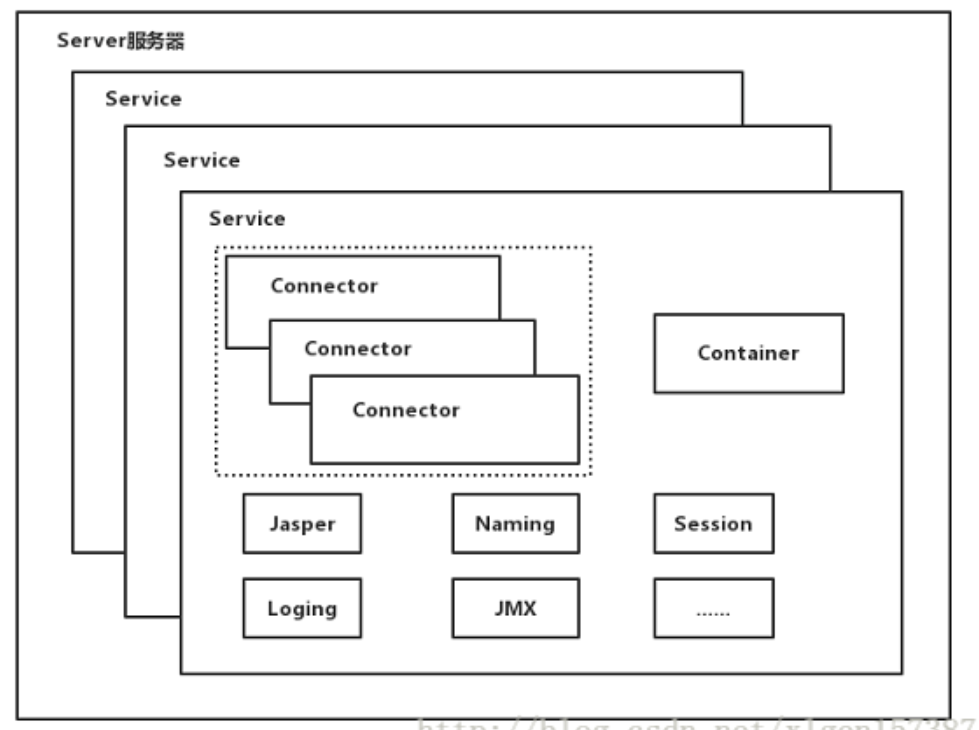
# 简介



一个tomcat只有一个server，一个server下有多个service，一个service下有一个container和多个connector。

1. Connector用于处理连接相关的事情，并提供socket与request和response相关的转化。一个tomcat下有多个connector时因为一个连接对应一个connector，可以同时提供http和https连接，也可以提供向相同协议不同端口的连接。Connector最底层是socket连接的。
2. Container用于封装和管理servlet，以及具体处理request请求。
3. service就是对外提供服务的。
4. Server控制着tomcat的整个生命周期。

# 服务过程

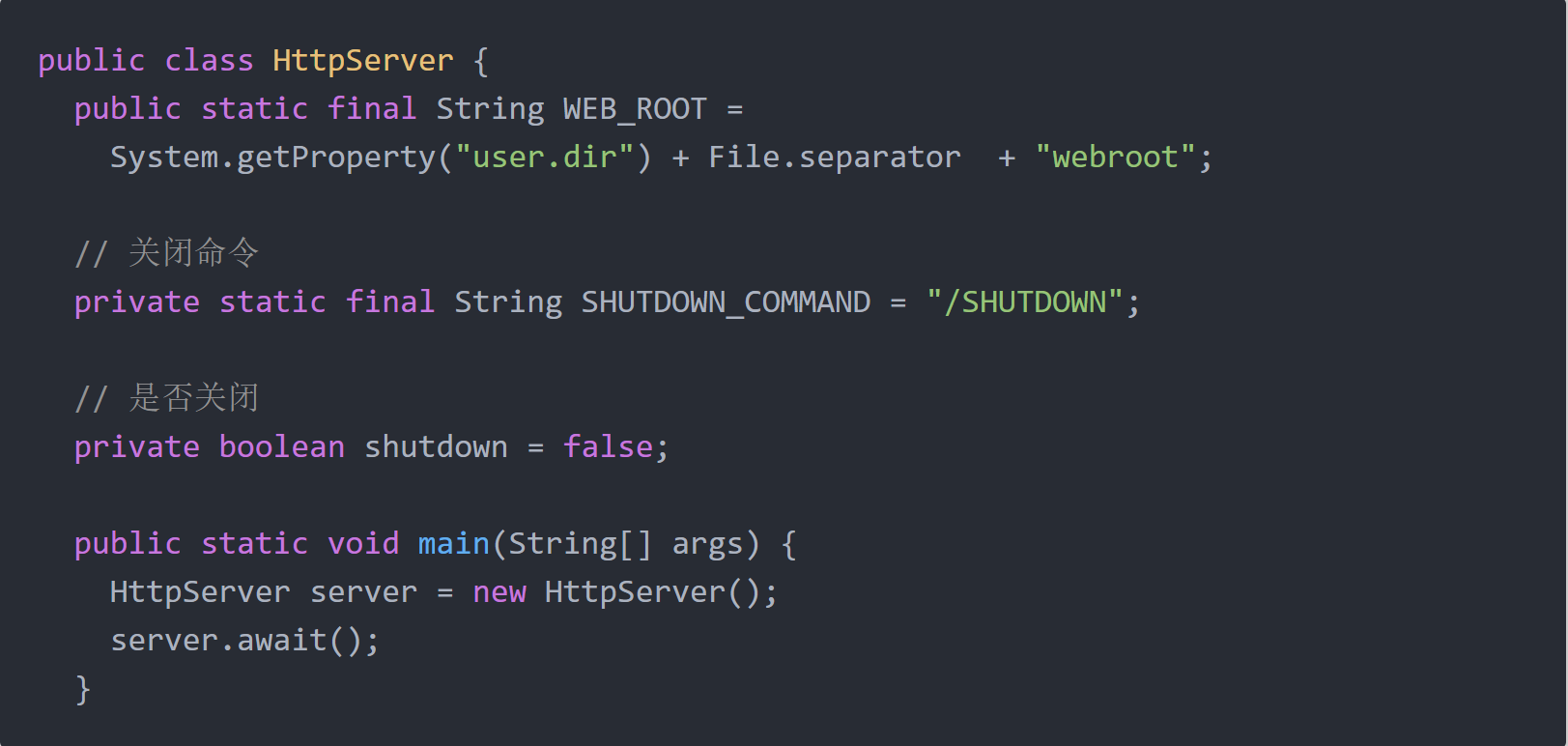
一个请求到达tomcat之后，首先经过service，然后交给connector，connector用于接收请求并将接受的请求封装为request和response来具体处理。Request和response封装完之后交给container处理，结束之后再返回给conncetor，最后由conncetor将处理的结果返回给客户端，这样整个请求就处理完了。

1. 请求到达server，server根据url映射到相应的servlet
2. 判断servlet实例是否存在，不存在则加载和实例化servlet并调用init方法。
3. Server分别创建request和response对象，调用servlet实例的service方法（service会根据http请求方法类型调用doXXX方法）
4. doXXX方法为业务逻辑实现，从request对象获取请求参数，处理完毕后将结果通过response对象返回给调用方
5. 当server不需要servlet时（一般server关闭时），server调用servlet的destroy方法。

# Context

# 底层实现

## HttpServer



这是tomcat的入口，启动tomcat之后，main方法进入到主方法，调用await方法等待请求。



创建一个ServerSocket，阻塞等待连接。当有连接请求时，就会创建一个request，并且调用它的parse方法。然后创建一个response，调用sendStaticResource方法。

## Request

### Parse方法

|  |
| --- |
| public void parse() {  // Read a set of characters from the socket  StringBuffer request = new StringBuffer(2048);  int i;  byte[] buffer = new byte[2048];  try {  // 读取流中内容  i = input.read(buffer);  }  catch (IOException e) {  e.printStackTrace();  i = -1;  }  for (int j=0; j<i; j++) {  // 将每个字节转换为字符  request.append((char) buffer[j]);  }  // 打印字符串  System.out.print(request.toString());  // 根据转换出来的字符解析URI  uri = parseUri(request.toString());  } |

创建StringBuffer，读取请求内容。将每个字节转换成字符。然后调用parseUri方法解析URI。

### parseUri方法

|  |
| --- |
| private String parseUri(String requestString) {  int index1, index2;  // 找到第一个空格  index1 = requestString.indexOf(' ');  if (index1 != -1) {  // 找到第二个空格  index2 = requestString.indexOf(' ', index1 + 1);  if (index2 > index1)  // 截取第一个空格到第二个空格之间的内容  return requestString.substring(index1 + 1, index2);  }  return null;  } |

为什么是获取第一个和第二个空格之间的内容呢？

看一下http请求：

|  |
| --- |
| POST /examples/default.jsp HTTP/1.1  Accept: text/plain; text/html  Accept-Language: en-gb  Connection: Keep-Alive  Host: localhost  User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 98)  Content-Length: 33 Content-Type: application/x-www-form-urlencoded Accept-Encoding: gzip, deflate  lastName=Franks&firstName=Michael |

第一行内容，两个空格之间的内容就是URI。

## Response

### sendStaticResource方法

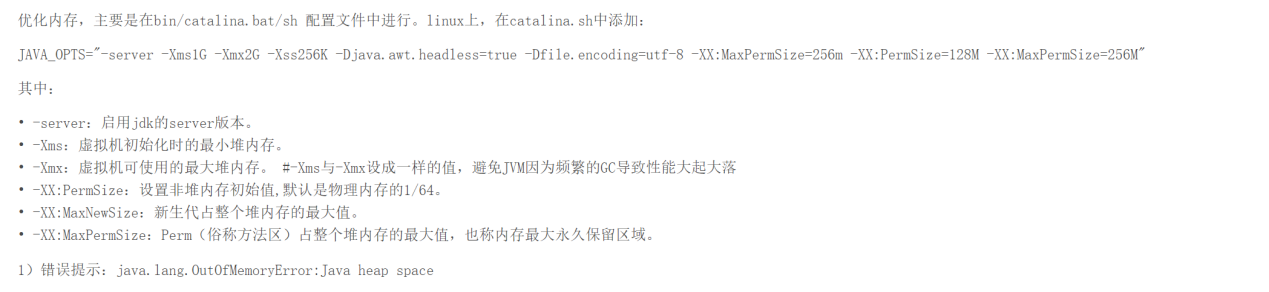
|  |
| --- |
| public void sendStaticResource() throws IOException {  byte[] bytes = new byte[BUFFER\_SIZE];  FileInputStream fis = null;  try {  File file = new File(HttpServer.WEB\_ROOT, request.getUri());  if (file.exists()) {  // 文件存在则从输出流中输出  fis = new FileInputStream(file);  int ch = fis.read(bytes, 0, BUFFER\_SIZE);  while (ch!=-1) {  output.write(bytes, 0, ch);  ch = fis.read(bytes, 0, BUFFER\_SIZE);  }  }  else {  // 没有文件返回404  String errorMessage = "HTTP/1.1 404 File Not Found\r\n" +  "Content-Type: text/html\r\n" +  "Content-Length: 23\r\n" +  "\r\n" +  "<h1>File Not Found</h1>";  output.write(errorMessage.getBytes());  }  }  catch (Exception e) {  // thrown if cannot instantiate a File object  System.out.println(e.toString() );  }  finally {  if (fis!=null)  fis.close();  }  } |

根据请求的URI创建File对象，然后通过outPutStream写出。

# 调优

## Tomcat内存大小的影响

优化内存，主要是在bin/catalina.bat/sh 配置文件中进行。linux上，在catalina.sh中添加



Tomcat本身不能直接在计算机上运行，需要依赖硬件基础之上的操作系统和一个java虚拟机。Java程序启动时JVM都会分配一个初始内存和最大内存给这个应用程序。这个初始内存和最大内存在一定程度上都会影响程序的性能。

应用程序达到最大内存时，JVM先去做垃圾回收的动作，释放一下占用的内存。

如果初始化内存比较小：

1. 应用程序启动时，但是依然还有好多对象进行初始化，虚拟机就会重复的增加内存满足使用，直到达到最大内存，应用程序启动就会变慢。
2. JVM为了保证系统尽可能的可以在指定的内存范围内运行，而频繁的进行GC操作，从而对系统的性能产生影响。
3. 所以一般将-Xms和-Xmx设为一样大。而最大值的设置受限于系统使用的物理内存，一般堆的最大值设置为可用内存的最大值的80%。

|  |
| --- |
| 内存方式的设置是在catalina.sh中，调整一下JAVA\_OPTS变量即可，因为后面的启动参数会把JAVA\_OPTS作为JVM的启动参数来处理。  具体设置如下：  JAVA\_OPTS="$JAVA\_OPTS -Xmx3550m -Xms3550m -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4"  其各项参数如下：  -Xmx3550m：设置JVM最大可用内存为3550M。  -Xms3550m：设置JVM促使内存为3550m。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。  -Xmn2g：设置年轻代大小为2G。整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。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。  -Xss128k：设置每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。更具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。  -XX:NewRatio=4:设置年轻代（包括Eden和两个Survivor区）与年老代的比值（除去持久代）。设置为4，则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  -XX:SurvivorRatio=4：设置年轻代中Eden区与Survivor区的大小比值。设置为4，则两个Survivor区与一个Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6  -XX:MaxPermSize=16m:设置持久代大小为16m。  -XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概论。 |

## 连接数

#优化连接数，主要是在conf/server.xml配置文件中进行修改。

### 优化线程数

找到Connector port="8080" protocol="HTTP/1.1"，增加maxThreads和acceptCount属性（使acceptCount大于等于maxThreads），如下：

<Connector port="8080" protocol="HTTP/1.1"connectionTimeout="20000" redirectPort="8443"acceptCount="500" maxThreads="400" />

其中：

• maxThreads：tomcat可用于请求处理的最大线程数，默认是200

• minSpareThreads：tomcat初始线程数，即最小空闲线程数

• maxSpareThreads：tomcat最大空闲线程数，超过的会被关闭

• acceptCount：当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理.默认100

### 使用线程池

在server.xml中增加executor节点，然后配置connector的executor属性，如下：

<Executor name="tomcatThreadPool" namePrefix="req-exec-"maxThreads="1000" minSpareThreads="50"maxIdleTime="60000"/>

<Connector port="8080" protocol="HTTP/1.1"executor="tomcatThreadPool"/>

其中：

• namePrefix：线程池中线程的命名前缀

• maxThreads：线程池的最大线程数

• minSpareThreads：线程池的最小空闲线程数

• maxIdleTime：超过最小空闲线程数时，多的线程会等待这个时间长度，然后关闭

• threadPriority：线程优先级

注：当tomcat并发用户量大的时候，单个jvm进程确实可能打开过多的文件句柄，这时会报java.net.SocketException:Too many open files错误。可使用下面步骤检查：

• ps -ef |grep tomcat 查看tomcat的进程ID，记录ID号，假设进程ID为10001

• lsof -p 10001|wc -l 查看当前进程id为10001的 文件操作数

• 使用命令：ulimit -a 查看每个用户允许打开的最大文件数

### Tomcat Connector三种运行模式（BIO, NIO, APR）

#### 三种模式比较

1）BIO：一个线程处理一个请求。缺点：并发量高时，线程数较多，浪费资源。Tomcat7或以下在Linux系统中默认使用这种方式。

2）NIO：利用Java的异步IO处理，可以通过少量的线程处理大量的请求。Tomcat8在Linux系统中默认使用这种方式。Tomcat7必须修改Connector配置来启动（conf/server.xml配置文件）：

<Connector port="8080"protocol="org.apache.coyote.http11.Http11NioProtocol" connectionTimeout="20000"redirectPort="8443"/>

1. APR(Apache Portable Runtime)：从操作系统层面解决io阻塞问题。Linux如果安装了apr和native，Tomcat直接启动就支持apr。

#### 修改为NIO

<Connector port="8080" protocol="HTTP/1.1"

connectionTimeout="20000"

redirectPort="8443" />

修改为

<Connector port="8080" protocol="org.apache.coyote.http11.Http11NioProtocol"

connectionTimeout="20000"

redirectPort="8443" />

如果需要修改ajp，可以这样修改：

<Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />

修改为

<Connector port="8009" protocol="org.apache.coyote.ajp.AjpNioProtocol" redirectPort="8443" />

#### apr模式安装

安装apr以及tomcat-native

yum -y install apr apr-devel

进入tomcat/bin目录，比如：

cd /opt/local/tomcat/bin/

tar xzfv tomcat-native.tar.gz

cd tomcat-native-1.1.32-src/jni/native

./configure --with-apr=/usr/bin/apr-1-config

make && make install

#注意最新版本的tomcat自带tomcat-native.war.gz，不过其版本相对于yum安装的apr过高，configure的时候会报错。

解决：yum remove apr apr-devel –y,卸载yum安装的apr和apr-devel,下载最新版本的apr源码包，编译安装;或者下载低版本的tomcat-native编译安装

安装成功后还需要对tomcat设置环境变量，方法是在catalina.sh文件中增加1行：

CATALINA\_OPTS="-Djava.library.path=/usr/local/apr/lib"

#apr下载地址：http://apr.apache.org/download.cgi

#tomcat-native下载地址：http://tomcat.apache.org/download-native.cgi

修改8080端对应的conf/server.xml

protocol="org.apache.coyote.http11.Http11AprProtocol"

<Connector executor="tomcatThreadPool"

port="8080"

protocol="org.apache.coyote.http11.Http11AprProtocol"

connectionTimeout="20000"

enableLookups="false"

redirectPort="8443"

URIEncoding="UTF-8" />

PS:启动以后查看日志 显示如下表示开启 apr 模式

Sep 19, 2016 3:46:21 PM org.apache.coyote.AbstractProtocol start

INFO: Starting ProtocolHandler ["http-apr-8081"]

## 垃圾回收策略调优

|  |
| --- |
| 垃圾回收的设置也是在catalina.sh中，调整JAVA\_OPTS变量。  具体设置如下：  JAVA\_OPTS="$JAVA\_OPTS -Xmx3550m -Xms3550m -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100"  具体的垃圾回收策略及相应策略的各项参数如下：  串行收集器（JDK1.5以前主要的回收方式）  -XX:+UseSerialGC:设置串行收集器  并行收集器（吞吐量优先）  示例：  java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100  -XX:+UseParallelGC：选择垃圾收集器为并行收集器。此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。  -XX:ParallelGCThreads=20：配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。  -XX:+UseParallelOldGC：配置年老代垃圾收集方式为并行收集。JDK6.0支持对年老代并行收集  -XX:MaxGCPauseMillis=100:设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM会自动调整年轻代大小，以满足此值。  -XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。  并发收集器（响应时间优先）  示例：java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC  -XX:+UseConcMarkSweepGC：设置年老代为并发收集。测试中配置这个以后，-XX:NewRatio=4的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn设置。  -XX:+UseParNewGC: 设置年轻代为并行收集。可与CMS收集同时使用。JDK5.0以上，JVM会根据系统配置自行设置，所以无需再设置此值。  -XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。  -XX:+UseCMSCompactAtFullCollection：打开对年老代的压缩。可能会影响性能，但是可以消除碎片 |

## 添加JMS监控

|  |
| --- |
| 对于部署在局域网内其它机器上的Tomcat，可以打开JMX监控端口，局域网其它机器就可以通过这个端口查看一些常用的参数（但一些比较复杂的功能不支持），同样是在JVM启动参数中配置即可，配置如下：  -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.authenticate=false  -Djava.rmi.server.hostname=192.168.71.38 设置JVM的JMS监控监听的IP地址，主要是为了防止错误的监听成127.0.0.1这个内网地址  -Dcom.sun.management.jmxremote.port=1090 设置JVM的JMS监控的端口  -Dcom.sun.management.jmxremote.ssl=false 设置JVM的JMS监控不实用SSL  -Dcom.sun.management.jmxremote.authenticate=false 设置JVM的JMS监控不需要认证 |

# Tomcat如何创建servlet容器

|  |
| --- |
| 当容器启动时，会读取在webapps目录下所有的web应用中的web.xml文件，然后对xml文件进行解析，  并读取servlet注册信息。然后，将每个应用中注册的servlet类都进行加载，并通过反射的方式实例化。  （有时候也是在第一次请求时实例化）在servlet注册时加上如果为正数，则在一开始就实例化，  如果不写或为负数，则第一次请求实例化。 |