# Part1 概述

## 简介

遵循J2EE Servlet规范，实现Servlet和JSP的支持，又叫Servlet容器。在中小型系统和并发访问用户不是很多的场合下被普遍使用。

Tomcat6

发布时间：2007

Tomcat7

发布时间：2011

* Servlet 3.0、JSP 2.2、EL 2.2
* 部分支持WebSocket，标准未正式指定

Tomcat8

发布时间：2013

* 支持Java EE 7规范，包括Java Servlet 3.1、JSP 2.3、EL 3.0等.
* 增加了对NIO2的支持
* Java WebSocket 1.0

Tomcat8.5

发布时间：2016

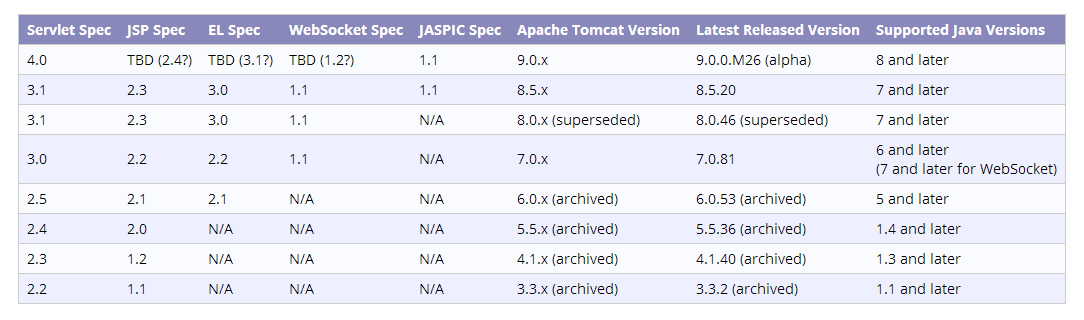
* 支持HTTP/2

Tomcat9

beta

* Servlet4.0 JSP2.4 EL3.1

JDK版本关系



安装Tomcat7要安装JDK1.6及之后版本才能正常启动Tomcat

## Servlet

Servlet接口是一套处理网络请求的规范，J2EE规范之一，主要为了扩展java的web功能。应用开发人员根据规范编写业务程序，servlet容器实现HTTP服务(面向接口编程)。

Java Servlet 3.1

* 实现了非阻塞式的I/O通信。
* 升级HTTP协议。原本使用HTTP通信协议升级成WebSockets通信

## 1.3 支持协议

BIO，NIO和APR。

Tomcat7中支持这3种

BIO

到了Tomcat8.5和Tomcat9.0，BIO模式因为效率太低已被淘汰。

APR

是Apache HTTP服务器的支持库。可以简单地理解为Tomcat将以JNI的形式调用Apache HTTP服务器的核心动态链接库来处理文件读取或网络传输操作，从而大大地提高Tomcat对静态文件的处理性能

# Part 2 性能优化

## 2.1 性能检测

### 2.1.1 检测技巧

很多时候在线上出现问题时，往往会伴随着某些指标的异常。大部分情况下，在问题发生之前，某些指标就会提前有异常显示。

step 1 概括

top查看系统负荷，正常情况单核负荷在0.7，四核就是3一下。

监控进程 top

java线程 ps

客户连接数 netstat

压力测试

<https://blog.csdn.net/wyingquan/article/details/414608>

ps 查看java进程

jstack单个线程

jconsole 线程的变化

jvisula线程池

### 2.1.1 Manager

[Reference](https://tomcat.apache.org/tomcat-7.0-doc/manager-howto.html)

Server Status

<http://localhost:8080/manager/status>

<http://localhost:8080/manager/status?XML=true>

**Stage**

P: Parse and prepare request

S: Service

F: Finishing

R: Ready

K: Keepalive

TODO2 linux 命令 tomcat 每次只能accept一个request？

Thread Dump

<http://localhost:8080/manager/text/threaddump>

VM Info

<http://localhost:8080/manager/text/vminfo>

<http://localhost:8080/manager/status?XML=true>

### 2.1.4 Thread

windows

tasklist

#### ps

查看进程占用内存情况。Process Status



SID 会话ID（Session id）

%CPU 进程的cpu占用率

%MEM 进程的内存占用率

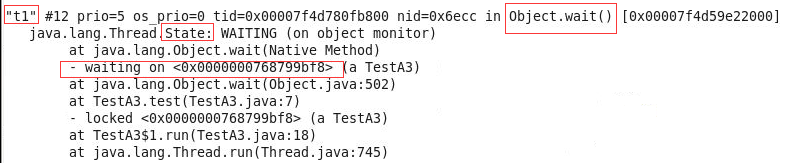
#### jstack

[Reference](http://www.hollischuang.com/archives/110)

生成线程快照。

**主要目的**：定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等。 通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做什么事情，或者等待什么资源。

**用法**：jstack <pid>



线程状态

A处为线程状态。NEW

调用修饰

表示线程在方法调用时,额外的重要的操作。线程Dump分析的重要信息。修饰上方的方法调用。

locked <地址> 目标：使用synchronized申请对象锁成功,监视器的拥有者。

waiting to lock <地址> 目标：使用synchronized申请对象锁未成功,在迚入区等待。

waiting on <地址> 目标：使用synchronized申请对象锁成功后,释放锁幵在等待区等待。

parking to wait for <地址> 目标

**RUNNABLE**

**locked**进入synchronized方法, 申请对象锁成功才会出现此句



BLOCKED

**waiting to lock**申请锁失败，申请成功应该是-locked <地址>。



WATING

**waiting on ：**synchronized修饰的方法调用wait()。Thread释放锁后会**等待锁**



申请对象锁成功后,释放锁进入等待区。线程状态：WAITING

parking to wait for <地址> 目标

TIMED\_WAITING

at java.lang.Thread.sleep

**Thread State**

Wait on condition

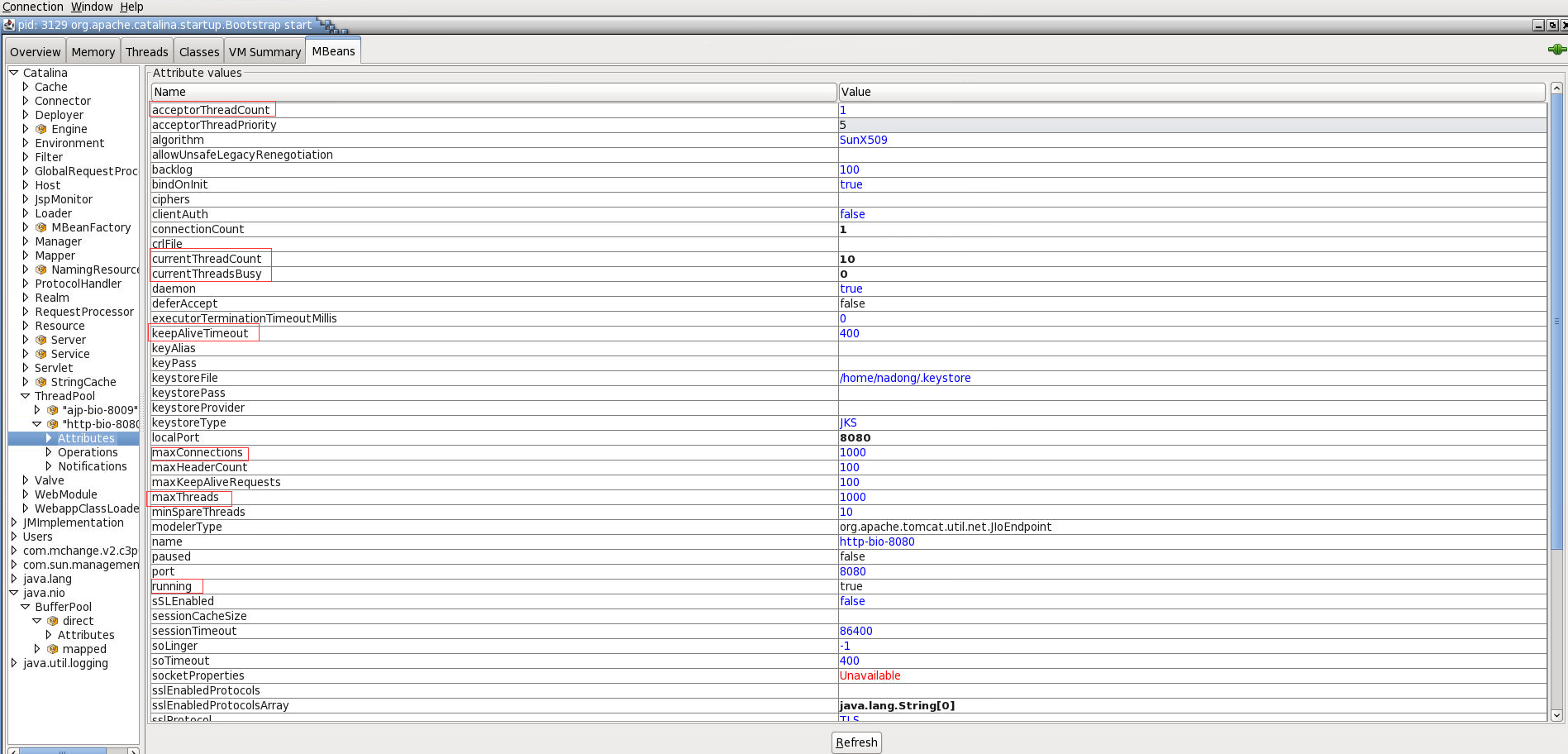
死锁

http://www.hollischuang.com/archives/2409

http://www.hollischuang.com/archives/110

#### jconsole

jconsole查询Acceptor属性



currentThreadCount

currentThreadBusy 线程池

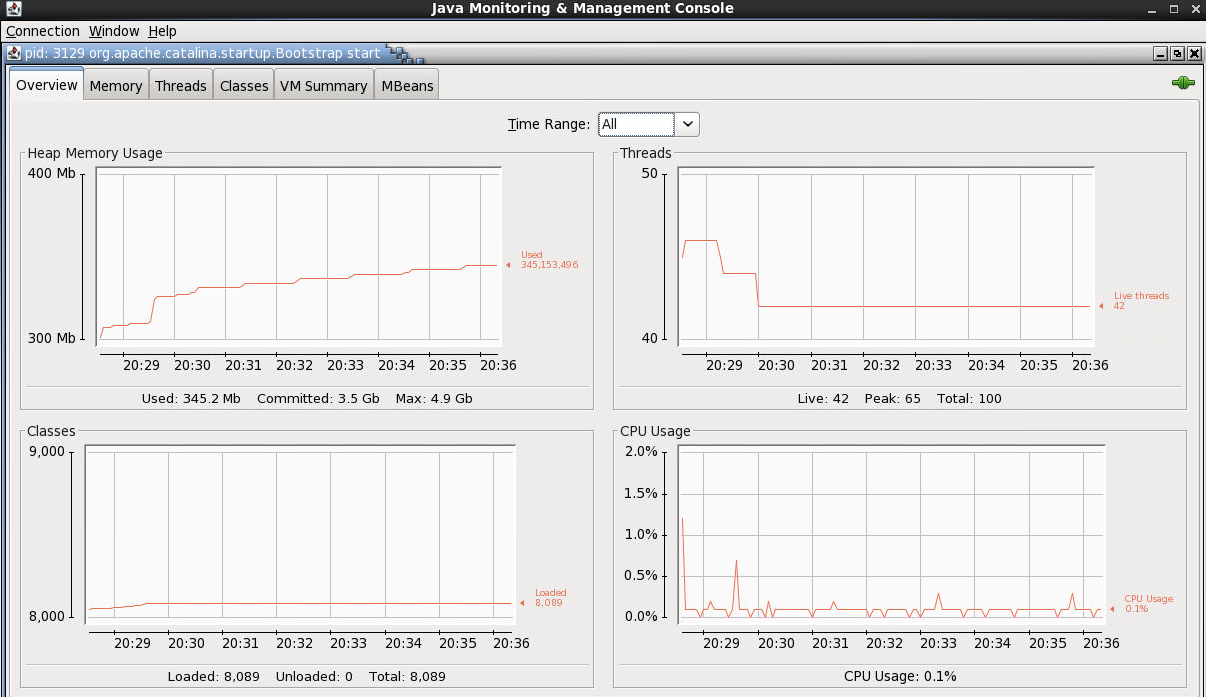
#### jconsole

配置

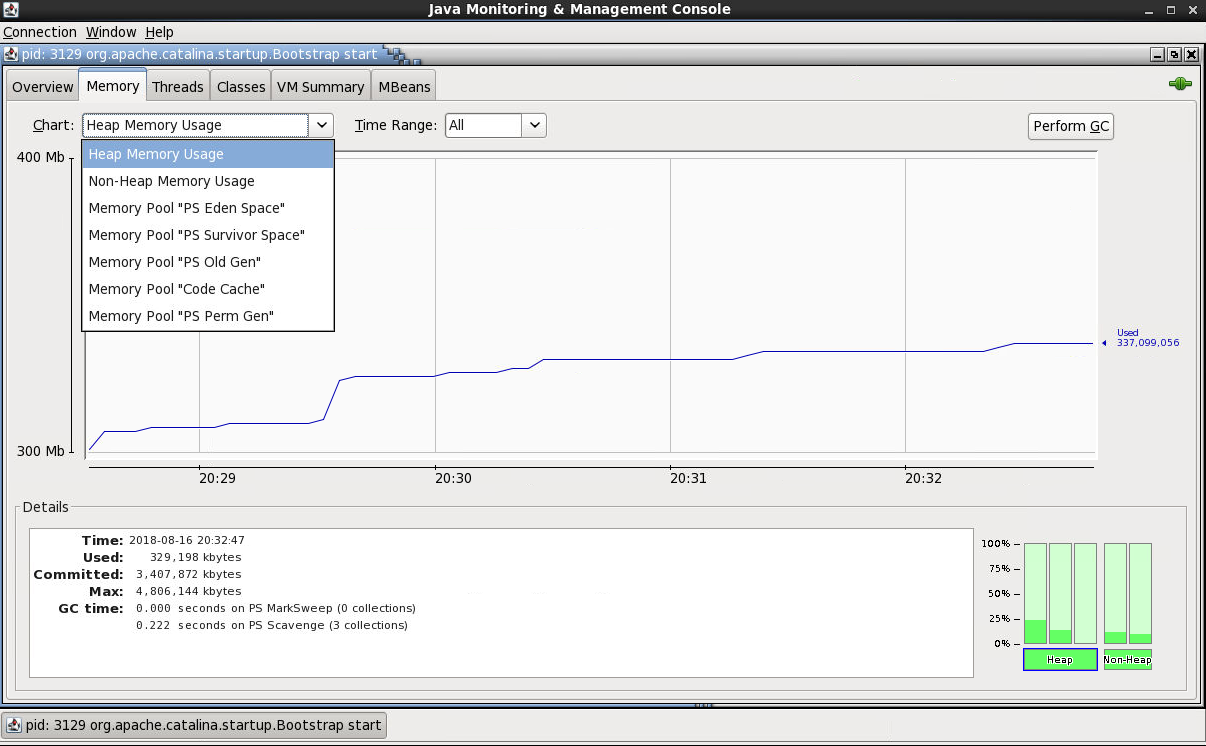
JAVA\_OPTS

-Djava.rmi.server.hostname=127.0.0.1 -Dcom.sun.management.jmxremote.port=8090 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false

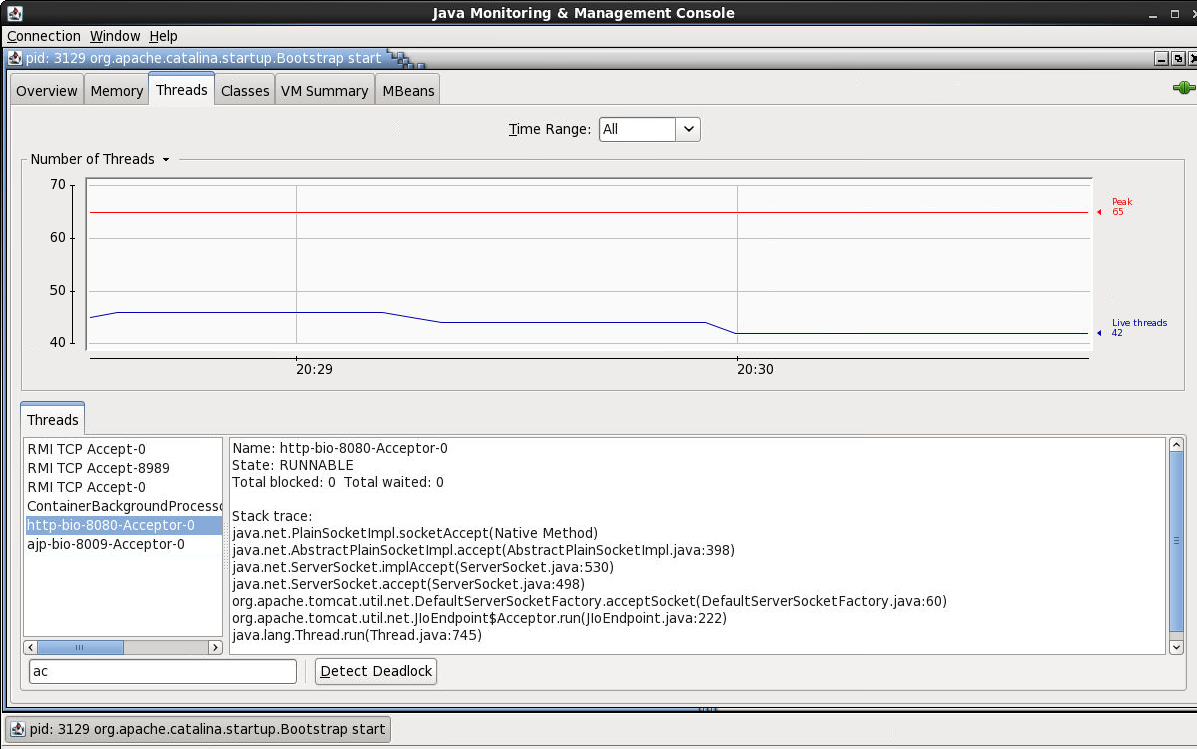
Overview



Memory



Thread



Total blocked

Total waited:

**state**

Wait on condition

#### Jvisual

JDK自带性能检测工具



**Running**: thread is still running.   
**Sleeping**: thread is sleeping (method yield () was called on the thread object)  
**Wait**: thread was blocked by a mutex or a barrier, and is waiting for another thread to release the lock

**Park**: parked threads are suspended until they are given a permit. Unparking a thread is usually done by calling method unpark () on the thread object  
**Monitor**: threads are waiting on a condition to become true to resume execution

#### 线程池检测

### 2.1.3 网络连接

Linux查看连接

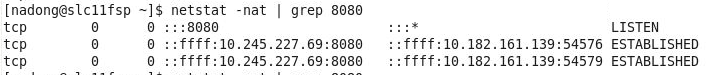
netstat -na|grep -i "8080"

netstat -na | grep ESTABLISHED | grep 8080 | wc –l

wc –l

-c 自动刷新

netstat -nat



### 2.1.3 内存

#### jstat

jstat -option pic < interval >

**gcutil**



S0：年轻代中第一个survivor（幸存区）已使用的占当前容量百分比   
S1：年轻代中第二个survivor（幸存区）已使用的占当前容量百分比   
E：年轻代中Eden（伊甸园）已使用的占当前容量百分比   
O：old代已使用的占当前容量百分比

P：perm代已使用的占当前容量百分比   
YGC：从应用程序启动到采样时年轻代中gc次数   
YGCT：从应用程序启动到采样时年轻代中gc所用时间(s)

FGC：从应用程序启动到采样时old代(全gc)gc次数   
FGCT：从应用程序启动到采样时old代(全gc)gc所用时间(s)   
GCT：从应用程序启动到采样时gc用的总时间(s)   
**gc**



 S0C：年轻代中第一个survivor（幸存区）的容量 (byte)   
 S1C：年轻代中第二个survivor（幸存区）的容量 (byte)   
 S0U：年轻代中第一个survivor（幸存区）目前已使用空间 (byte)   
 S1U：年轻代中第二个survivor（幸存区）目前已使用空间 (byte)   
 EC：年轻代中Eden（伊甸园）的容量 (byte)   
 EU：年轻代中Eden（伊甸园）目前已使用空间 (byte)   
OC：Old代的容量 (byte)   
OU：Old代目前已使用空间 (byte)   
PC：Perm(持久代)的容量 (byte)   
PU：Perm(持久代)目前已使用空间 (字节)

**gcnew**

年轻代对象的信息



**gcold**

 old代对象的信息



**gcoldcapacity**

old代对象的信息及其占用量



OGCMN：old代中初始化(最小)的大小 (字节)   
OGCMX：old代的最大容量 (字节)   
OGC：old代当前新生成的容量 (字节)   
OC：Old代的容量 (byte)

**gcnewcapacity:**



 NGCMN：young初始化(最小)的大小 (字节)   
 NGCMX：年轻代(young)的最大容量 (字节)   
 NGC：年轻代(young)中当前的容量 (字节)

S0CMX：年轻代中第一个survivor（幸存区）的最大容量 (字节)   
S1CMX ：年轻代中第二个survivor（幸存区）的最大容量 (字节)

ECMX：年轻代中Eden（伊甸园）的最大容量 (字节)   
**gcpermcapacity**

 PGCMN：perm代中初始化(最小)的大小 (字节)   
  PGCMX：perm代的最大容量 (字节)     
  PGC：perm代当前新生成的容量 (字节)   
PC：Perm(持久代)的容量 (byte)

#### jmap

## 2.2 Thread优化

tomcat的线程优化，本质上就是java 的线程池优化。结合socket特点来优化。实现线程资源和任务的一种平衡。

### 设置策略

maxThreads

maximumPoolSize

最大的线程数。即最大连接数。Tomcat7默认值200。maxThreads规定的是最大的线程数目，并不是实际running的CPU数量；实际running的是maxConnections。

maxThreads应设大点。因为，处理线程真正用于计算的时间可能很少，大多数线程都在等待；大多数时间可能在阻塞，如等待数据库返回数据、等待硬盘读写数据等。当然，maxThreads的值并不是越大越好，如果maxThreads过大，那么CPU会花费大量的时间用于线程的切换，整体效率会降低。

acceptCount

所有可用的线程数都被使用时，可以放到处理队列中的请求数。默认值10。

通过前面的介绍可以知道，虽然tomcat同时可以处理的连接数目是maxConnections，但服务器中可以同时接收的连接数为maxConnections+acceptCount 。acceptCount的设置，与应用在连接过高情况下希望做出什么反应有关系。如果设置过大，后面进入的请求等待时间会很长；如果设置过小，后面进入的请求立马返回connection refused

maxThreads和acceptCount。如果要加大并发连接数，应同时加大这两个参数。

maxConnections

maxConnections的设置与Tomcat的运行模式有关。如果tomcat使用的是BIO，那么maxConnections的值应该与maxThreads一致；如果tomcat使用的是NIO，那么类似于Tomcat的默认值，maxConnections值应该远大于maxThreads。

KeepAlive

maxKeepAliveRequests： 保持请求数量，默认值100。

keepAliveTimeout 默认值60000

keepAliveTimeout VS connnectionTimeout

**connnectionTimeout**： 网络连接超时，默认值60000，单位：毫秒。通常可设置为30000毫秒。设置为0表示永不超时，这样设置有隐患的

minSpareThreads：

最小空闲线程数。Tomcat初始化时创建的线程数。默认值25。

maxSpareThreads： 最大空闲线程数。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。默认值75。

**acceptorThreadCount**： acceptor数量

bufferSize： 输入流缓冲大小，默认值2048 bytes。

compression： 压缩传输，取值on/off/force，默认值off。 其中和最大连接数相关的参数为

enableLookups： 是否反查域名，默认值为true。为了提高处理能力，应设置为false

### 线程数量

**1) Max Number of threads:**

(MaxProcessMemory - JVMMemory - ReservedOsMemory) / ThreadStackSize

* MaxProcessMemory 指的是一个进程的最大内存。在32位的 windows下是 2G
* JVMMemory         JVM内存
* ReservedOsMemory  保留的操作系统内存
* ThreadStackSize      线程栈的大小(-Xss)

2)线程数量

((CUP时间+CUP等待时间)/CUP时间)\*CUP数量

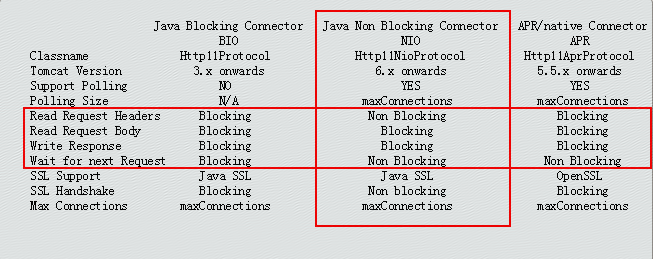
server.xml

1. **<Connector** port="8080" protocol="HTTP/1.1"
2. connectionTimeout="20000" maxThreads="1000" minSpareThreads="60"
3. maxSpareThreads="600" acceptCount="120"  redirectPort="8443" URIEncoding="utf-8"**/>**

## 2.3 IO优化

配置文件server.xml，略。

IO模型支持



tomcat要实现servlet规范所以不能最大发挥NIO的特性，servlet3.0之前完全是同步阻塞模型，Read http body 以及 response还是同步阻塞，因为servlet规范规定的就是这样。

## 2.4 内存优化

参考JVM优化

# Part3 部署发布

## 3.1 部署

热部署vs热加载，热加载是运行时通过重新加载改变类信息，直接改变程序行为。

1)静态部署

2)动态部署:不用重新启动服务器。

## 3.2 优雅停机

shutdown.sh

## 3.3 集群

## 3.4负载均衡

# Part 4 实现原理

## 4.1 架构分析

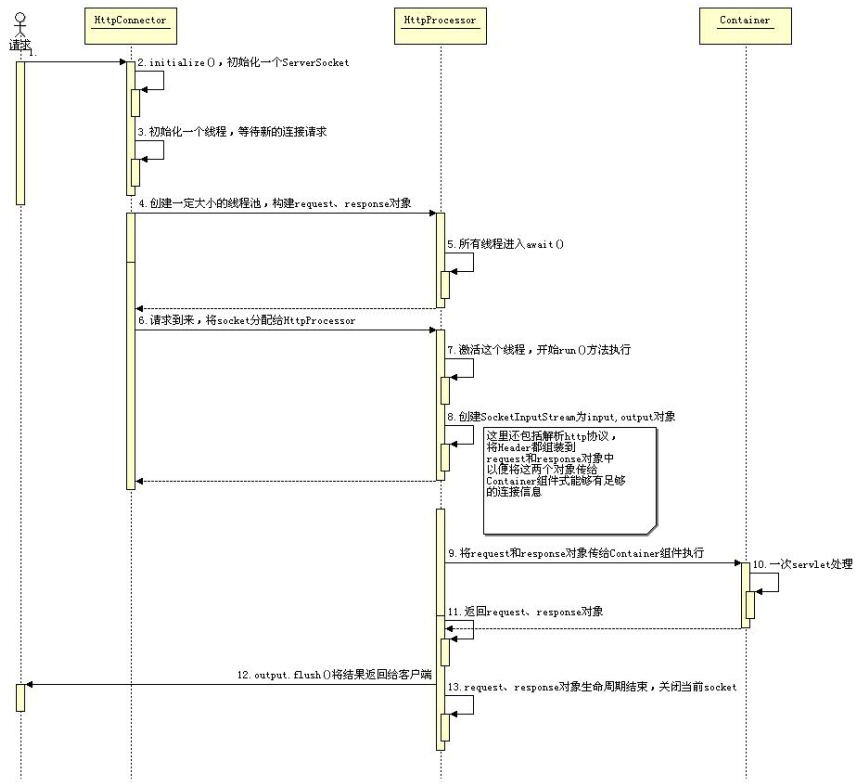
[源代码分析](https://www.ibm.com/developerworks/cn/java/j-lo-tomcat1/index.html)



Connector 接受请求

Container 处理connector接受的请求

Process



## 4.2 源码

### Connector

NIO

**源码分析：**<http://tyrion.iteye.com/blog/2256896>

**分层建模**

tomcat架构的高度模块化。这些细分的模块，使得tomcat非常健壮，通过一些配置和模块定制化，可以很大限度的扩展tomcat。



* Wrapper封装了具体的访问资源，例如 index.html
* Context 封装了各个wrapper资源的集合，例如 app
* Host 封装了各个context资源的集合，例如 [www.mydomain.com](http://www.mydomain.com)

### Bootstrap

入口

### Catalina.java

**start**

1. **public** **void** start() {
2. //调用 server 的start 方法启动服务器
3. getServer().start();  //StandardServer
4. }

### StandardService

1. @Override
2. **protected** **void** startInternal() **throws** LifecycleException {
3. //启动 容器
4. container.start();  //standardEngine[Cataline]
5. // 启动线程池
6. **for** (Executor executor: executors) {
7. executor.start();
8. }
9. // Start our defined Connectors second
10. **for** (Connector connector: connectors) {
11. // 启动 Connector 链接
12. connector.start();
13. }
14. }

container start

connector start

**Connector BIO**



Http11ConnectionHandler

|- Http11ConnectionHandler

|- JIoEndpoint

#### process

connector start🡪startInternal🡪 protocolHandler.start() [Http11Protocol] 🡪**endpoint.start()**[JIoEndpoint]

### JIoEndpoint

extends AbstractEndpoint

start

1. **public** **final** **void** start() **throws** Exception {
2. bind();;
3. startInternal();
4. }

#### bind

bind Socket

1. @Override
2. **public** **void** bind() **throws** Exception {
3. // Initialize thread count defaults for acceptor
4. **if** (acceptorThreadCount == 0) {
5. acceptorThreadCount = 1;
6. }
7. if(getMaxConnections()==0){
8. setMaxConnections(getMaxThreadsInternal());
9. }
10. serverSocketFactory = **new** DefaultServerSocketFactory(**this**);
12. **if** (serverSocket == **null**) {
13. serverSocket = serverSocketFactory.createSocket(getPort(),
14. getBacklog(), getAddress());
16. }
17. }

#### startInternal

1. @Override
2. **public** **void** startInternal() **throws** Exception {
3. // Create worker collection
4. createExecutor();
5. initializeConnectionLatch();
6. startAcceptorThreads();
7. }

#### Acceptor

1. **protected** **class** Acceptor **extends** AbstractEndpoint.Acceptor {
2. @Override
3. **public** **void** run() {
4. **while** (running) {
5. //if we have reached max connections, wait
6. countUpOrAwaitConnection();
7. // 此处用来接收 请求 监听客户端连接
8. Socket socket = serverSocketFactory.acceptSocket(serverSocket);             // Hand this socket off to an appropriate processor
9. **if** (!processSocket(socket)) {
10. countDownConnection();
11. // Close socket right away
12. socket.close();
13. }
14. }
15. }

#### processSocket

1. **protected** **boolean** processSocket(Socket socket) {
2. // Process the request from this socket
3. **try** {
4. SocketWrapper<Socket> wrapper=**new** SocketWrapper<Socket>(socket);            wrapper.setKeepAliveLeft(getMaxKeepAliveRequests());
5. wrapper.setSecure(isSSLEnabled());
6. getExecutor().execute(**new** SocketProcessor(wrapper));
7. **return** **true**;
8. }

#### SocketProcessor

### AbstractEndpoint

1. **public** **abstract** **class** AbstractEndpoint<S> {
2. **public** **abstract** **class** Acceptor
4. **public** **interface** Handler;
6. **protected** Acceptor[] acceptors;
7. **protected** **int** acceptorThreadCount = 0;
9. **private** **int** maxConnections;
10. **private** **int** maxThreads = 200;
12. **private** Executor executor = **null**;
13. **private**  int maxKeepAliveRequests=100;
14. **private** Integer keepAliveTimeout = **null**;
15. **private** **volatile** LimitLatch connectionLimitLatch = **null**
16. }

#### executor

1. **public** **void** createExecutor() {
2. internalExecutor = **true**;
3. TaskQueue taskqueue = **new** TaskQueue();
4. TaskThreadFactory tf = **new** TaskThreadFactory(getName() + "-exec-", daemon, getThreadPriority());
5. executor = **new** ThreadPoolExecutor(getMinSpareThreads(), getMaxThreads(), 60, TimeUnit.SECONDS,taskqueue, tf);
6. taskqueue.setParent( (ThreadPoolExecutor) executor);
7. }

TaskQueue LinkedBlockingQueue

maxThreads maximumPoolSize

minSpareThreads corePoolSize

#### maxConnections

1. **protected** LimitLatch initializeConnectionLatch() {
2. **if** (connectionLimitLatch==**null**) {
3. connectionLimitLatch = **new** LimitLatch(getMaxConnections());
4. }
5. **return** connectionLimitLatch;
6. }

#### countUpOrAwaitConnection

1. **protected** **void** countUpOrAwaitConnection() **throws** InterruptedException {
2. LimitLatch latch = connectionLimitLatch;
3. **if** (latch!=**null**) latch.countUpOrAwait();
4. }

超过maxConnections数，Acceptor将await

#### acceptors

**startAcceptorThreads**

1. **protected** **final** **void** startAcceptorThreads() {
2. **int** count = getAcceptorThreadCount();
3. acceptors = **new** Acceptor[count];
4. //启动acceptorThreadCount个线程，每个线程由Acceptor代理
5. **for** (**int** i = 0; i < count; i++) {
6. acceptors[i] = createAcceptor();
7. String threadName = getName() + "-Acceptor-" + i;
8. acceptors[i].setThreadName(threadName);
9. Thread t = **new** Thread(acceptors[i], threadName);
10. t.setPriority(getAcceptorThreadPriority());
11. t.setDaemon(getDaemon());
12. t.start();
13. }
14. }
15. }

Acceptor是缺省的优先级 Thread.NORM\_PRIORITY;

### Container

Init WebApplicationContext.以standardEngine[Cataline] 为例

#### StandardEngine