# 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。

## 1.2 servlet

Servlet规范

## 1.3 支持协议

BIO，NIO和APR。

Tomcat7中支持这3种，Tomcat8增加了对NIO2的支持，而到了Tomcat8.5和Tomcat9.0，则去掉了对BIO的支持

BIO

APR

TODO1

# Part 2 性能优化

## 2.1 性能监控

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？

## 2.2 Thread优化

maxThreads

maxThreads规定的是最大的线程数目，并不是实际running的CPU数量；

**设置策略**

1maxThreads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。

2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。

3 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。

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** 最大的线程数。即最大连接数。Tomcat7默认值200。

**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。

**minSpareThreads**： Tomcat初始化时创建的线程数。默认值25。

**maxSpareThreads**： 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。

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

maxConnections 最大连接数

KeepAlive

maxKeepAliveRequests： 保持请求数量，默认值100。

keepAliveTimeout

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

**connnectionTimeout**： 网络连接超时，默认值60000，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。

**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

**TODO4**

**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

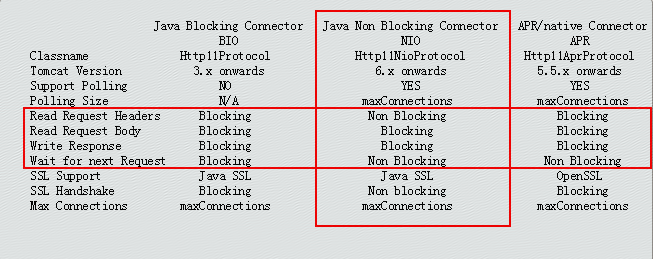
server.xml

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

# Part 4 实现原理

## 4.1 架构分析

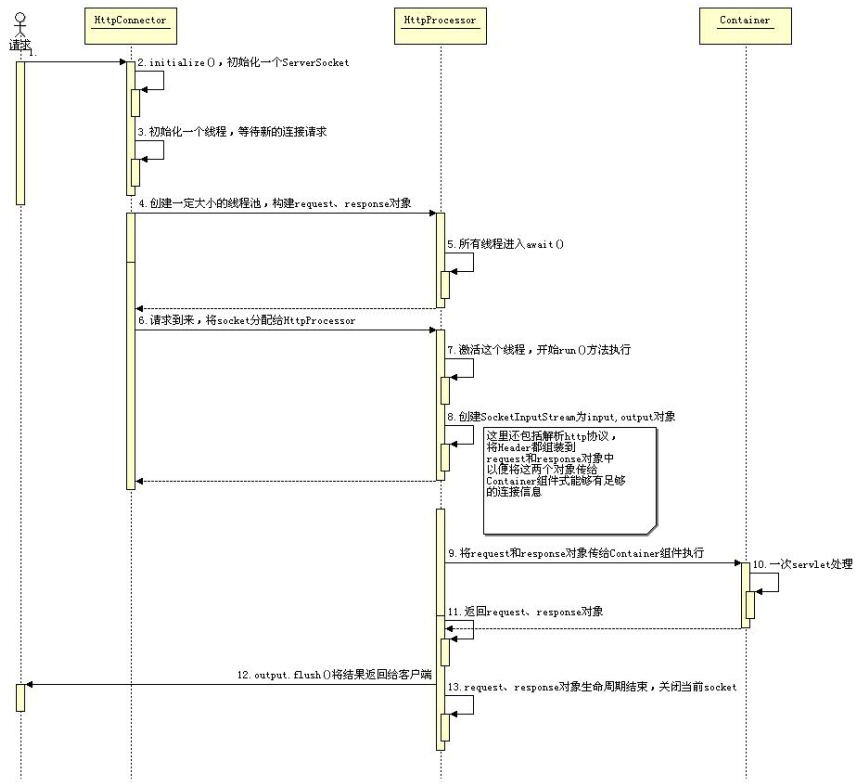
[源代码分析](https://www.ibm.com/developerworks/cn/java/j-lo-tomcat1/index.html)



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. closeSocket(socket);
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. }

处理socket的pool ，在server.xml文件配置maxThread

#### 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

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. }

### Container

Init WebApplicationContext.以standardEngine[Cataline] 为例

#### StandardEngine