# J2SE

## 1.3 NIO

### NIO

实现原理

在linux中，基于IO多路复用（即事件驱动）的网络模式，调用select/poll/epoll，当某个socket有数据到达了，就通知用户进程。

**selector**.select()

每次select都会block，select监听注册的事件，包括Accept，Read，write等

### Mina

类结构



## 1.4 Thread

### 2.2.1 线程池

#### ThreadPoolExecutor

##### shutdown

1. **public** **void** shutdown() {
2. **final** ReentrantLock mainLock = **this**.mainLock;
3. mainLock.lock();
4. **try** {
5. checkShutdownAccess();
6. advanceRunState(SHUTDOWN);
7. interruptIdleWorkers();
8. onShutdown(); // hook for ScheduledThreadPoolExecutor
9. } **finally** {
10. mainLock.unlock();
11. }
12. tryTerminate();
13. }

#### ExecutorService

API

1. **public** **interface** ExecutorService **extends** Executor {
2. **void** shutdown();
3. List<Runnable> shutdownNow();
4. **boolean** isShutdown();
5. **boolean** isTerminated();
6. **boolean** awaitTermination(**long** timeout, TimeUnit unit);
7. <T> Future<T> submit(Callable<T> task);
8. <T> Future<T> submit(Runnable task, T result);
9. }

submit有返回值Future，而execute没有

TODO2:submit方便Exception处理。

Future

1. **public** **interface** Future<V> {
2. **boolean** cancel(**boolean** mayInterruptIfRunning);
3. **boolean** isCancelled();
4. V get() **throws** InterruptedException, ExecutionException;
5. V get(**long** timeout, TimeUnit unit);
6. }

isDone()： 执行状态

get(): 获取执行结果。 Waits if necessary for the computation to complete, and then retrieves its result.

**Executor**

1. **public** **interface** Executor {
2. **void** execute(Runnable command);
3. }

### 1.5.5 Concurrent包

#### 1.5.5.3 ReentrantLock

实现原理

[Reference](https://www.cnblogs.com/maypattis/p/6403682.html)

**ReentrantLock源码**



可见：默认为非公平锁

内部类继承关系



# 第二篇 JAVA Web

## 2.3 Spring

**方便程序的测试**：可以用非容器依赖的编程方式进行几乎所有的测试工作。在Spring里，测试不再是昂贵的操作，而是随手可做的事情。

**方便集成各种优秀的框架**，降低各种框架的使用难度。

**降低Java EE API的使用难度**：spring对j2ee API(如JDBC、远程调用)封装

**Java源码是经典的学习范例**。spring源码是java技术的最佳实践，值得研究学习。

## 应用服务器

### Tomcat

### Part 2 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 3 实现原理

#### 4.1 架构分析

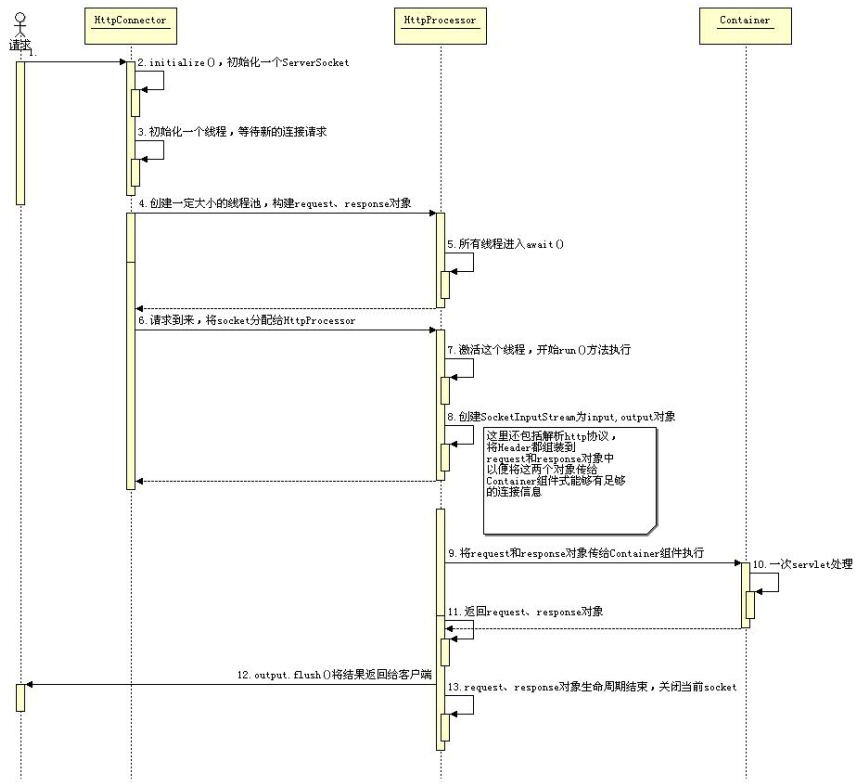
[源代码分析](https://www.ibm.com/developerworks/cn/java/j-lo-tomcat1/index.html)



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。

##### 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

#### Atom

聚合和发布日志内容而设计的，统一格式——每个条目都包括title，entry，id，和content/summary——这使得我能从一条Atom feed中得到有用的信息，而不管   这条feed来自哪一类型的应用。