# 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

### Linux Socket

#### IO模式

按照《Unix网络编程》的划分，有下面五种IO模式

* 阻塞 I/O（blocking IO）
* 非阻塞 I/O（nonblocking IO）
* I/O 多路复用（IO multiplexing）
* 信号驱动 I/O（signal driven IO）
* 异步 I/O（asynchronous IO）

##### 阻塞IO

用户进程recv，如果kernel还未准备好足够数据，进程自己主动block，一直等到数据准备好了。如采用multi-threading + blocking，将创建大量线程，并有大量的线程切换。

##### 非阻塞IO

非阻塞IO在read时，如果kernel的数据没准备好，用户进程不block，而是返回error。no-blocking需要用户进程主动询问**，**效率较低，要结合IO multiplexing.

##### IO multiplexing

原理：调用select/poll/epoll函数，轮询所有socket，当某个socket有数据到达了，就通知用户进程。

**IO multiplexing VS no-blocking**

1. no-blocking需要用户进程**主动询问**kernel数据好了没有,IO multiplexing改进了no-blocking，增加了专门的监控轮询function。好处在于它可以同时处理多个socket（一次poll/select可检测所有socket情况，合理使用CUP资源）。
2. IO multiplexing每次上下文切换都是有意义的，大大减少上下文切换。

3.必须是no-blocking才能实现复用，不然监测轮询的process会被block

**应用case**

如果处理的连接数不是很高的话，使用select/epoll不一定比使用multi-threading+ blocking IO性能更好，可能延迟还更大。。

#### API

#include <sys/socket.h>

scoket🡪bind🡪listen🡪accept🡪recv🡪close

##### Socket

[Reference](https://blog.csdn.net/xc_tsao/article/details/44123331)

Unix下任何设备、管道、FIFO等都是文件形式，socket就是一个文件，socket句柄就是一个文件描述符

1. #include<sys/socket.h>
2. **int** socket(**int** domain, **int** type, **int** protocol);

Return 文件描述符fd,非负整数。实际上是一个索引值

domain 协议族。AF\_INET,PF\_INET是IPv4

type 协议类型。SOCK\_STREAM/ SOCK\_DGRAM

##### listen

进入被动监听状态，不阻塞。

1. **int** listen(**int** sockfd, **int** backlog);

sockfd 套接字类型的文件描述符。

backlog请求队列最大长度。超过时客户端连接失败的错误或重传。

##### accept

接受客户端请求，建立TCP连接，完成三次握手。Request Queue为空时阻塞。

1. **int** accept(**int** listenfd, **struct** sockaddr \*addr, socklen\_t \*addrlen);

listenfd 服务端的fd

addr 客户端地址

return 返回客户端sock\_fd

##### recv/send

通过fcntl函数可设为非阻塞

1. ssize\_t recv(**int** sockfd, **void** \*buff, **size\_t** nbytes, **int** flags);
2. ssize\_t send(**int** sockfd, **const** **void** \*buff, **size\_t** nbytes, **int** flags);

sockfd 发送端fd，服务器的角度则是客户端

buff

send(),recv()用于TCP，sendto()及recvfrom（）用于UDP

##### fcntl

函数可以改变已打开的文件性质

1. **int** fcntl(**int** fd, **int** cmd, ... /\* arg \*/ );

这里针对读写功能，设置为非阻塞后，和accept无关,accept要实现非阻塞需要用select()

##### select

select 函数监视的文件描述符分3类，分别是writefds、readfds、和exceptfds。调用后select函数会阻塞。当select函数返回后，可以 通过遍历fdset，来找到就绪的描述符。

1. **int** select(**int** maxfdp,fd\_set \*readfds,fd\_set \*writefds,
2. fd\_set \*errorfds,**struct** timeval \*timeout);

maxfdp fd数量最大值

**fd\_set**

**fd的集合，通过宏操作**

FD\_ZERO(fd\_set \*);清空集合

FD\_SET(int, fd\_set \*);将一个给定的文件描述符加入集合之中

FD\_CLR(int, fd\_set\*); 将一个给定的文件描述符从集合中删除

**return**

正值：某些文件可读写或出错 负值select错误 0：等待超时，没有可读写或错误的文件

##### poll

和select函数一样，poll返回后，需要轮询pollfd来获取就绪的描述符

1. **int** poll (**struct** pollfd \*fds, unsigned **int** nfds, **int** timeout);

pollfd

1. **struct** pollfd {
2. **int** fd; /\* file descriptor \*/
3. **short** events; /\* requested events to watch \*/
4. **short** revents; /\* returned events witnessed \*/
5. };

SELECT VS POLL

从上面看，select和poll都需要在返回后，通过遍历文件描述符来获取已经就绪的socket。(事实上，同时连接的大量客户端在一时刻可能只有很少的处于就绪状态，因此随着监视的描述符数量的增长，其效率也会线性下降)。

Return : 事件num

##### epoll

传统的select/poll另一个致命弱点就是当你拥有一个很大的socket集合，不过由于网络延时，任一时间只有部分的socket是"活跃"的，但是select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是epoll不存在这个问题

#### 缓存IO

大多数文件系统的默认 I/O 操作都是缓存 I/O。在 Linux 的缓存 I/O 机制中，OS将 I/O 的数据缓存在文件系统的页缓存（ page cache ）中。也就是说，**数据会先被拷贝到操作系统内核的缓冲区中**，然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间。

**缓存 I/O 的缺点**：

数据在传输过程中需要在应用程序地址空间和内核进行多次数据拷贝操作，这些数据拷贝操作所带来的 CPU 以及内存开销是非常大的。

非阻塞程序demo

[Reference](https://github.com/lovemooner/CMoon/blob/master/src/Thread100.cpp)

### VM底层实现

在Java中要使用非阻塞

1. SocketChannel socketChannel = SocketChannel.open();
2. socketChannel.configureBlocking(**false**);

在JDK源码中实现

1. **static** **int** configureBlocking(**int** fd, jboolean blocking){
2. **int** flags = fcntl(fd, F\_GETFL);
3. **int** newflags = blocking ? (flags & ~O\_NONBLOCK) : (flags | O\_NONBLOCK);
4. **return** (flags == newflags) ? 0 : fcntl(fd, F\_SETFL, newflags);
5. }

搜索：c++ fcntl linux 非阻塞IO

### 长连接&心跳机制

 1）TCP协议自带的keep-alive功能,打开即可



当keepAlive为true，此处服务端每隔一段时间向客户端发送一个数据包检查客户端的活动状态，如没response会将socket关闭。

2）**应用层面实现心跳机制**

自定义心跳消息头. 一般客户端主动发送, 服务器接收后进行回应(也可以不回应)

3）框架自带

MINA本身提供了一个过滤器类[KeepAliveFilter](http://mina.apache.org/mina-project/apidocs/org/apache/mina/filter/keepalive/KeepAliveFilter.html)  ,该过滤器用于在IO空闲的时候发送并且反馈心跳包

### Mina

架构

多工作线程Reactor模式



类结构



**Acceptor**：

处理连接请求

**NioProcessor**

循环读



连接管理

session相对http较长,手动管理。





长连接&心跳机制

监控无效的连接并断开

* TCP协议自带的keep-alive功能,打开即可
* 应用层面实现心跳机制，自定义心跳消息头. 一般客户端主动发送。
* 框架自带，MINA本身提供了一个过滤器类[KeepAliveFilter](http://mina.apache.org/mina-project/apidocs/org/apache/mina/filter/keepalive/KeepAliveFilter.html)

Netty

对JDK网络编程的封装，屏蔽了繁杂的编程细节，提供便于用户开发网络应用程序的api,让开发者可以更加专注于业务逻辑的实现。

**基础通信组件,RPC工具**

* 很多中间件把netty作为基础通信组件。
* 阿里分布式服务框架 Dubbo
* 淘宝的消息中间件 RocketMQ 的消息生产者和消息消费者之间, 也采用 Netty 进行高性能、异步通信。

**1.异步**

非阻塞，事件驱动机制

**2.高性能的通信框架**

不用依赖于容器去进行部署,易扩展

## 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. }

##### getActiveCount

1. **public** **int** getActiveCount() {
2. **final** ReentrantLock mainLock = **this**.mainLock;
3. mainLock.lock();
4. **try** {
5. **int** n = 0;
6. **for** (Worker w : workers)
7. **if** (w.isLocked())
8. ++n;
9. **return** n;
10. } **finally** {
11. mainLock.unlock();
12. }
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

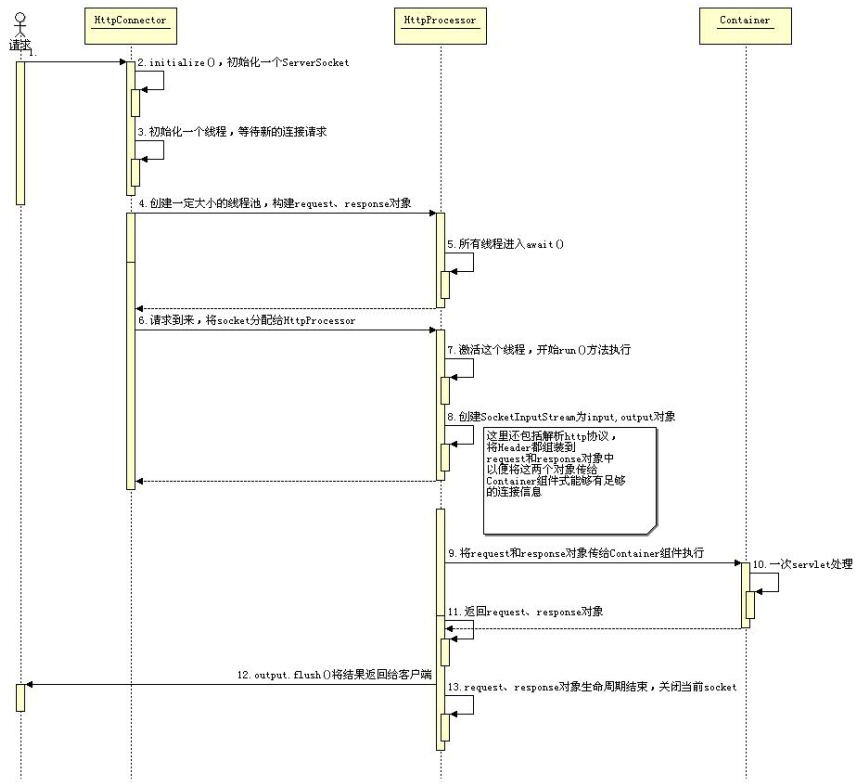
[源代码分析](https://www.ibm.com/developerworks/cn/java/j-lo-tomcat1/index.html)



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。



* 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

#### 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。