# J2SE

## 1.1 Data Type

**常量**

和变量相对，不能更改。在Java中，指final修饰的变量。

好处：

* 使用final关键字，JVM会对方法、变量及类进行优化
* 提高了性能。JVM和Java应用都会缓存final变量。

在多线程环境下进行共享，而不需要额外的同步开销。

**常量池**

实现常量池技术的基本类型包装类：Integer, Long，String

static

TODO：static内部类

### 变量

**局部变量**：方法中的变量。分配在stack

**类变量**：方法之外的变量，用 static 修饰。分配在方法区。

**实例变量**：方法之外的变量，不过没有 static 修饰。分配在堆

**成员变量**（全局变量）：包括实例变量和类变量

总结: 方法内定义的变量和引用全部在栈中，方法外定义的**实例变量全部在堆**中，静态变量在方法区。

### 数组

final数组

数组也是对象，引用arr1的地址不能改变，内容可改变



数组复制

**浅拷贝**: 不进行递归拷贝，直接拷贝内存本身的值，所以产生的结果是基本数据类型是值拷贝，对象只是引用拷贝

Arrays.copyOf 底层调用System.arraycopy，浅拷贝

**深拷贝**

Object.clone 对于对象而言，它是深拷贝，但是对于数组而言，它是浅拷贝

### 引用类型

StrongReference

强引用。普遍的引用。

SoftReference

软引用。如果内存空间不足了，就会回收这些对象的内存软引用可用来实现内存敏感的高速缓存

1. String str=**new** String("abc");
2. SoftReference<String> softRef=**new** SoftReference<String>(str);

WeakReference

弱引用。不管当前内存空间足够与否，都会回收它的内存

PhantomReference

虚引用。在任何时候都可能被垃圾回收器回收。主要用来跟踪对象被垃圾回收器回收的活动，虚引用必须和引用队列 （ReferenceQueue）联合使用

### Object

equals



“==”比较两个对象的的内存地址，可见equals相同的两个对象一定相等

重写equals为何要重写hashCode

hashCode的存在主要是用于查找的快捷性，用来在散列存储结构中确定对象的存储地址的。

从官方文档的定义，可以得到equals与hashCode间的关系：

1、两个对象equals，hashCode一定要相同。

2、两个对象的hashCode相同，它们并不一定equals。

wait ()/notify ()/notifyAll ()

Code：WaitNotifyCase100

使用方法



warning:

线程wait后被notify，要重新获得lock才能从中断处继续执行。

### Integer

包装类：

Java的8种基本类型都有对应的包装类。包装类与基本类型的转换：

**自动拆装箱**



装箱:Integer.valueOf()

拆箱：Integer.intValue()

需要注意拆箱时可能发生NPE？

valueOf

缓存机制



可见在-128~128范围



Integer.MAX\_VALUE

nteger是4位，4\*8=32,去掉一个符号位，-2^31~2^31-1)

### final

**final Class:**类不能被继承。方法不能rewrite

**final Method:**方法不能rewrite

**final Variable** 修饰的基本类型的变量，初始化后不能更改。修饰引用类型不能执行另一个对象，但对象内容可以更改。

修饰基本类型



### String

#### immutable



数组value是也是对象，内容是可变的，可见String的不可变性在于底层的实现，而不是一个final。

**目的**

设计成immutable的类，可以像普通类型一样赋值，而不是普通的类。

**好处**

主要是为了“效率” 和 “安全性” 的缘故。

效率：对象的不可变性，所以可以共享。节省内存空间。

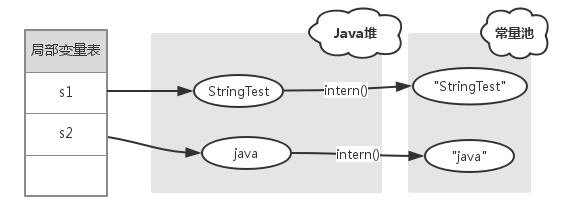
安全性：string作为对象，可以像基本类型值传递，而不改变对象的值。

不仅String 是final的。 Long, Double, Integer 之类的全都是final的

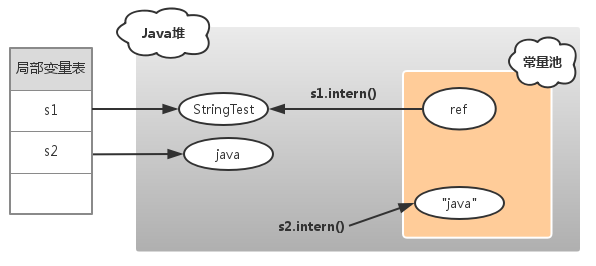
#### 常量池

[Reference](https://www.jianshu.com/p/0d1c003d2ff5)

1. **class** Test {
2. **public** **static** **void** main(String args[]) {
3. String s1 = **new** StringBuilder().append("String").append("Test").toString(); System.out.println(s1.intern() == s1);
5. String s2 = **new** StringBuilder().append("ja").append("va").toString();
6. System.out.println(s2.intern() == s2);
7. }
8. }
9. **JDK6的执行结果：false, false**



1. **JDK7的执行结果：true false**



String str=”a”



ldc先在常量池创建对象

String str=new String(“a”)



ldc “str”过程

触发oop StringTable::intern

“+=”的重载

“+=” 是Java中仅有的两个重载过的操作符,而java并不允许我们重载任何操作符。

**对+的重载**

加号被转换为StringBuffer的append()方法来操作,最后toString生成新的对象。



String.valueOf和Integer.toString的区别、

## 1.2 Collection



### 1.2.1 List

#### ArrayList

1. **public** **class** ArrayList<E> {
2. **private** **static** **final** **int** DEFAULT\_CAPACITY = 10;
3. **private** **transient** Object[] elementData;
4. **private** **int** size;
5. }

 DEFAULT\_CAPACITY = 10

Add

1. **public** **boolean** add(E e) {
2. ensureCapacityInternal(size + 1);  // Increments modCount!!
3. elementData[size++] = e;
4. **return** **true**;
5. }

扩容

1. **private** **void** grow(**int** minCapacity) {
2. // overflow-conscious code
3. **int** oldCapacity = elementData.length;
4. **int** newCapacity = oldCapacity + (oldCapacity >> 1);
5. **if** (newCapacity - minCapacity < 0)
6. newCapacity = minCapacity;
7. **if** (newCapacity - MAX\_ARRAY\_SIZE > 0)
8. newCapacity = hugeCapacity(minCapacity);
9. // minCapacity is usually close to size, so this is a win:
10. elementData = Arrays.copyOf(elementData, newCapacity);
11. }

minCapacity = size +1

Arrays.copyof

其使用的是内存复制，省去了大量的数组寻址访问等时间

#### LinkedList

链表 VS 数组

### 1.2.3 Map

#### HashMap

1）结构

[Reference](http://blog.csdn.net/dongnan591172113/article/details/8770923)

HashMap底层是一个数组结构，数组中的每一项又是一个链表。

1. **public** **class** HashMap<K,V>{
3. **transient** Entry[] table;
4. **transient** **int** size;
6. **static** **class** Entry<K,V> **implements** Map.Entry<K,V> {
7. **final** K key;
8. V value;
9. Entry<K,V> next;
10. **final** **int** hash;
11. }
12. }

2）读取



3）扩容机制

1. **if** size>capacity \* loadFactor（loadFactor=0.75）
2. 用一个新的数组代替已有的容量小的数组
3. newCapacity=2 \* table.length
4. Entry[] newTable = **new** Entry[newCapacity];
5. end **if**;

4）碰撞冲突

对应位置链表的next

## 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

### 1.4.1 多线程

#### 实现原理

[Reference](https://juejin.im/post/5a31d1be6fb9a04517053523)

linux用户态的进程、线程基本满足上述概念，但内核态不区分进程和线程。可以认为，内核中统一执行的是进程，但有些是“普通进程”（对应进程process），有些是“轻量级进程”（对应线程pthread或npthread），都使用task\_struct结构体保存保存。

使用fork创建进程，使用pthread\_create创建线程。两个系统调用最终都都调用了do\_dork，而do\_dork完成了task\_struct结构体的复制，并将新的进程加入内核调度。

TODO1:Linux线程实现

#### 线程上下文切换

引起切换的原因

* **时间片**用完，CPU正常调度下一个任务
* 被其他优先级更高的任务抢占
* 执行任务碰到**IO阻塞**，调度器挂起当前任务，切换执行下一个任务
* **用户代码主动**挂起当前任务让出CPU时间
* 多任务抢占资源，由于没有抢到被挂起
* 硬件中断

进程切换，线程切换？

### 1.5.2 线程同步

#### 线程状态

[Reference](http://blog.csdn.net/dongnan591172113/article/details/51967711)

New->Runable(start)->Running->Blocked

Blocked

Blocked VS Wait:

。

* 未获得锁进去等待队列EntrySet,已经获得锁主动释放，进入WaitSet。
* 从Linux内核来看，这些线程都是等待状态，没本质区别。jvm出于管理的需要，做了这种区分。
* Blocked由jvm唤醒，wait由另一个线程。

Interrupt

Demo:ThreadInterrupt100

每个线程都有一个打断标志。

* 线程在 sleep/wait/join 时，别的进程调用此线程的interrupt() 方法，此线程会被唤醒并被要求处理InterruptedException；
* 线程运行时执行interrupt，线程只会设置 “打断标志” 。

所以说，interrupt() 方法能够中断线程的等待过程,不会中断一个正在运行的线程。

### 2.2.1 线程池

在Java5之后，并发线程这块发生了根本的变化，最重要的莫过于新的启动(submit)、调度、管理(shutdown)线程的一大堆API了。

作用

TODO1 待总结

1.降低资源消耗：通过重用已经创建的线程来降低线程创建和销毁的消耗  
2.提高响应速度：任务到达时不需要等待线程创建就可以立即执行。  
3.提高线程的可管理性：线程池可以统一**管理、分配、调优和监控**。

#### ThreadPoolExecutor

[Reference](https://www.jianshu.com/p/ade771d2c9c0)

##### 构造器

1. **public** ThreadPoolExecutor(**int** corePoolSize,
2. **int** maximumPoolSize,
3. **long** keepAliveTime,
4. TimeUnit unit,
5. BlockingQueue<Runnable> workQueue,
6. ThreadFactory threadFactory,
7. RejectedExecutionHandler handler) {
8. **this**.corePoolSize = corePoolSize;
9. **this**.maximumPoolSize = maximumPoolSize;
10. **this**.workQueue = workQueue;
11. **this**.keepAliveTime = unit.toNanos(keepAliveTime);
12. **this**.threadFactory = threadFactory;
13. **this**.handler = handler;
14. }

**corePoolSize**  worker数。比如newFixedThreadPool初始化定义corePoolSize 个线程。当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；

**maximumPoolSize**

**newFixedThreadPool** corePoolSize和maximumPoolSize的大小是一样的

无界queue的话maximumPoolSize参数是没有意义的

BlockingQueue 超过corePoolSize，用来存储等待执行的任务

keepAliveTime  超过corePoolSize数的空闲线程存活时间

RejectedExecutionHandler 线程拒绝策略

TimeUnit  keepAliveTime的单位

##### ctl

1. **private** **final** AtomicInteger ctl = **new** AtomicInteger(ctlOf(RUNNING, 0));

ctl包含两个field，workerCount和runState，workerCount是占据着一个atomic integer的后29位的，而状态占据了前3位，所以，workerCount上限是(2^29)-1。

workerCount

表示当前有效的线程数，也就是Worker的数量

runState

当前**线程池**的状态，runState是整个线程池的运行生命周期，有如下取值：

* RUNNING：可以新加线程，同时可以处理queue中的线程。
* SHUTDOWN：不增加新线程，但是处理queue中的线程。
* STOP 不增加新线程，同时不处理queue中的线程。
* TIDYING 所有的线程都终止了（queue中），同时workerCount为0，此时进入TIDYING
* TERMINATED terminated()方法结束，变为TERMINATED

##### submit

1. **public** Future<?> submit(Runnable task) {
2. **if** (task == **null**) **throw** **new** NullPointerException();
3. RunnableFuture<Void> ftask = newTaskFor(task, **null**);
4. execute(ftask);
5. **return** ftask;
6. }

##### execute

1. **public** **void** execute(Runnable command) {
2. **int** c = ctl.get();
3. **if** (workerCountOf(c) < corePoolSize) {
4. **if**(addWorker(command, **true**)){
5. **return**;
6. }
7. }
8. **if** (isRunning(c) && workQueue.offer(command)) {
9. ...
10. }**else** **if** (!addWorker(command, **false**)){
11. reject(command);
12. }
13. }

##### addWorker

1. **private** **boolean** addWorker(Runnable firstTask, **boolean** core) {
3. **boolean** workerStarted = **false**;
4. **boolean** workerAdded = **false**;
5. Worker w = **new** Worker(firstTask);
6. **final** Thread t = w.thread;
7. workers.add(w);
8. workerAdded = **true**;
9. **if** (workerAdded) {
10. t.start();
11. workerStarted = **true**;
12. }
14. **return** workerStarted;
15. }

[Reference](https://blog.csdn.net/cjh94520/article/details/70545202)

1. 线程被包装成worker



2. getTask()中有条件的进行着死循环，从而可以不断接受任务来进行。

1. **final** **void** runWorker(Worker w) {
2. **while** (task != **null** || (task = getTask()) != **null**) {
3. task.run();
4. }
5. }

执行任务时直接调用线程的run方法。所以池内线程可以执行不同任务。

##### 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

#### 1.5.5.1 Pool

#### 1.5.2 AQS

即AbstractQueuedSynchronizer 。 AQS实现FIFO等待队列，构建锁和同步容器提供了框架。Concurrent包内许多类都是基AQS构建。

##### CLH Lock

[Reference](https://www.jianshu.com/p/f43e581976b9)

CLH锁是一个自旋锁，提供先来先服务的公平性。线程必须等待占用的锁被释放后，通过竞争锁来获取资源的控制权。等待获取锁方式有“被动”和“主动”两种。

被动模式 VS 主动模式

被动模式：竞争线程自身进入阻塞状态，由某种调度机制唤醒；

主动模式：通常使用自旋的方式实现。竞争不激烈，锁资源很快就可以被释放的前提下，自旋锁的计算开销，要低于被动模式。

FIFO

在自旋模式下，依赖线程自身的抢占可能会导致其中某些线程始终无法获得锁，因此需要一个策略来保证争夺锁的线程可以以某种顺序获得锁的控制权，

实现方式：1.线程请求锁的顺序 2.线程优先级来。

##### AQS原理

AQS基于CLH锁的思想，主要从两方面进行了改造：**节点的结构与节点等待机制。**

* 在结构上引入了头结点和尾节点。
* 在等待机制上由原来的自旋改成阻塞唤醒。

##### 实现细节

**AbstractOwnableSynchronizer**

1. **class** AbstractOwnableSynchronizer{
2. //当前占用该锁的线程
3. **private** **transient** Thread exclusiveOwnerThread;
4. //同步状态
5. **private** **volatile** **int** state;
6. **private** **transient** **volatile** Node head;
7. **private** **transient** **volatile** Node tail;
8. }

state

1 占用 0 未占用，需控制同步

###### Node

节点。

1. Node {
2. **int** waitStatus;
3. Node prev;
4. Node next;
5. Node nextWaiter;
6. Thread thread;
7. }

waitStatus

1.CANCELLED( 1)，表示当前节点的线程因为超时或中断被取消了；

2.（0） 初始化状态。表示当前节点在sync队列中，等待着获取锁。

3.SIGNAL(-1)，当前节点的后续节点中的线程通过 park 被阻塞了，需要通过unpark解除它的阻塞；

4.CONDITION(-2)，表示当前节点在等待condition，也就是在condition队列中；

5.PROPAGATE(-3)共享模式的头结点可能处于此状态，表示无条件往下传播，引入此状态是为了优化锁竞争，使队列中线程有序地一个一个唤醒

nextWaiter

节点模式。

FIFO链表

AQS构造的队列实际是Node的链表



AQS在结构上引入了头结点和尾节点，他们分别指向队列的头和尾，尝试获取锁、入队列、释放锁等实现都与头尾节点相关，并且每个节点都引入前驱节点和后后续节点的引用。

**ReentrantLock.lock()**

1. **final** **void** lock() {
2. acquire(1);
3. }

###### acquire

tryAcquire🡪 addWaiter🡪 acquireQueued

1. **public** **final** **void** acquire(**int** arg) {
2. **if** (!tryAcquire(arg) &&
3. acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
4. selfInterrupt();
5. }

selfInterrupt: Thread.interrupted()

###### tryAcquire

获取锁，根据state判断锁是否被占用。

fairTryAcquire

1. **protected** **final** **boolean** tryAcquire(**int** acquires) {
2. **final** Thread current = Thread.currentThread();
3. **int** c = getState();
4. **if** (c == 0) {
5. **if** (!hasQueuedPredecessors() &&
6. compareAndSetState(0, acquires)) {
7. setExclusiveOwnerThread(current);
8. **return** **true**;
9. }
10. }
11. **return** **false**;
12. }

Nonfair Lock没有hasQueuedPredecessors。这里乐观锁cas对state同步。

hasQueuedPredecessors

[Reference](https://blog.csdn.net/tomato__/article/details/25782747)

判断是否有等待线程 false没有，true有

1. **public** **final** **boolean** hasQueuedPredecessors() {
2. Node t = tail; // Read fields in reverse initialization order
3. Node h = head;
4. Node s;
5. **return** h != t &&
6. ((s = h.next) == **null** || s.thread != Thread.currentThread());
7. }
8. Queued没有元素
9. Queued有元素，但是currentThread Node

**addWaiter**

入队。通过CAS把当前线程追加到队尾

1. **private** Node addWaiter(Node mode) {
2. Node node = **new** Node(Thread.currentThread(), mode);
3. ... Node追加到队尾
4. }

###### acquireQueued

阻塞Node (after addWaiter)。

1. **final** **boolean** acquireQueued(**final** Node node, **int** arg) {
2. **boolean** interrupted = **false**;
3. **for** (;;) {  //自旋
4. **final** Node p = node.predecessor();
5. **if** (shouldParkAfterFailedAcquire(p, node) &&
6. parkAndCheckInterrupt())
7. interrupted = **true**;
8. }
10. }

**shouldParkAfterFailedAcquire**

阻塞前Set pred node waitStatus🡪SIGNAL ？

1. **private** **static** **boolean** shouldParkAfterFailedAcquire(Node pred, Node node) {         **int** ws = pred.waitStatus;
2. **if** (ws == Node.SIGNAL)
3. **return** **true**;
4. **if** (ws > 0) {
5. **do** {
6. node.prev = pred = pred.prev;
7. } **while** (pred.waitStatus > 0);
8. pred.next = node;
9. } **else** {
10. compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
11. }
12. **return** **false**;
13. }

**parkAndCheckInterrupt**

1. **private** **final** **boolean** parkAndCheckInterrupt() {
2. LockSupport.park(**this**);
3. **return** Thread.interrupted();
4. }

**unLock**

1. **public** **void** unlock() {
2. sync.release(1);
3. }

###### release

tryRelease🡪 unparkSuccessor 从head出释放等待队列

1. **public** **final** **boolean** release(**int** arg) {
2. **if** (tryRelease(arg)) {
3. Node h = head;
4. **if** (h != **null** && h.waitStatus != 0)
5. unparkSuccessor(h);  //
6. **return** **true**;
7. }
8. **return** **false**;
9. }

**tryRelease**

释放锁

1. **protected** **boolean** tryRelease(**int** releases) {
2. setExclusiveOwnerThread(**null**);
3. setState(0);
4. **return** **true**;
5. }

unparkSuccessor

 唤醒Node。release处传入head Node

1. **private** **void** unparkSuccessor(Node node) {
2. **int** ws = node.waitStatus;
3. **if** (ws < 0)
4. compareAndSetWaitStatus(node, ws, 0);
5. Node s = node.next;
6. **if** (s == **null** || s.waitStatus > 0) {
7. s = **null**;
8. **for** (Node t = tail; t != **null** && t != node; t = t.prev)
9. **if** (t.waitStatus <= 0)
10. s = t;
11. }
12. **if** (s != **null**)  //next node存在
13. LockSupport.unpark(s.thread);
14. }

#### 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



#### 1.5.5.4 BlockingQueue

线程容器，为线程同步提供保障。

##### API

添加

add(anObject):可以添加返回true,否则抛异常

offer(anObject):可以添加返回true,否则返回false.

put(anObject): 添加时没有空间,调用的线程阻断到有空间再继续.

移除

poll(time):取走BlockingQueue里排在首位的对象,若不能立即取出,则可以等time参数规定的时间,取不到时返回null

take():取走BlockingQueue里排在首位的对象,若BlockingQueue为空,阻断进入等待状态直到Blocking有新的对象被加入为止

##### 实现类

ArrayBlockingQueue

规定大小的BlockingQueue,其构造函数必须带一个int参数来指明其大小.其所含的对象是以FIFO(先入先出)顺序排序的。

LinkedBlockingQueue

对象是以FIFO(先入先出)顺序排序的 。

maximumPoolSize的值也就无效了

大小不定的BlockingQueue,若其构造函数带一个规定大小的参数,生成的BlockingQueue有大小限制,若不带大小参数,所生成的BlockingQueue的大小由Integer.MAX\_VALUE来决定.

SynchronousQueue

特殊的BlockingQueue,对其的操作必须是放和取交替完成的.

PriorityBlockingQueue

类似于LinkedBlockQueue,但其所含对象的排序不是FIFO,而是依据对象的自然排序顺序或者是构造函数的Comparator决定的顺序.

LinkedBlockingQueue和ArrayBlockingQueue比较起来,它们背后所用的数据结构不一样,导致LinkedBlockingQueue的数据吞吐量要大于ArrayBlockingQueue,但在线程数量很大时其性能的可预见性低于ArrayBlockingQueue.

### 1.5.5 ThreadLocal

#### 实现原理

Code: ThreadLocal101

在多线程中，ThreadLocal为每个线程提供一份共享变量的的副本。类Thread有一个成员变量ThreadLocalMap。它的Key值为ThreadLocal对象，线程使用共享变量时，会new一个存在ThreadLocalMap中。

warning:

多线程中中，每个线程都new一个共享对象，如果是同一个对象还是有并发访问问题。

创建不同的ThreadLocal对象来存储多个变量；

ThreadLocal VS Thread同步机制

都是为了解决线程安全问题。对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，访问串行化，对象共享化。ThreadLocal采用了“以空间换时间”的方式：访问并行化，对象独享化。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

#### 使用场景

一般情况下，通过ThreadLocal.set() 到线程中的对象是该线程自己使用的对象，其他线程是不需要访问的，也访问不到的。各个线程中访问的是不同的对象。

最常见的ThreadLocal使用场景为 用来解决 数据库连接、Session管理等。

如：

[Reference](http://www.cnblogs.com/dolphin0520/p/3920407.html)

Spring中使用

**ThreadLocal.set()**将新建对象的引用保存到各线程的ThreadLocalMap。



传递参数

将ThreadLocal声明为static，这样同一线程在任何地方都可以获取set的值，

#### 内存泄漏问题

## JVM

### 1.5.1 JVM优化

### 1.5.2 类加载机制

### 对象内存管理

#### 运行时数据区



寄存器

即程序计数器，**保存当前执行指令的地址**，线程私有(每个线程都有自己的程序计数器)。

Stack

每条线程都有自己私有的虚拟机栈, 这个栈与线程同时创建，用于存储栈帧（Frames）。

线程工作内存(高速缓存)，存放基本型，以及对象引用。

栈帧

栈帧随方法调用而创建，随方法结束而销毁。和方法一一对应。

Frames

|  |
| --- |
| 局部变量表 |
| 操作数栈 |
| 动态链接 |
| 方法返回地址 |
| 方法调用 |

局部变量表和操作数栈编译期确定，通过method\_info的code属性保存，供栈帧使用。

**局部变量表**

存放方法参数和方法内的局部变量。以变量槽（slot）为单位，slot为32位。

**操作数栈**

编译后中产生的操作数压入。

**动态链接**

指向运行时常量池中该栈帧所属方法的引用,持有这个引用是为了支持方法调用过程中的动态连接.

方法调用：调用方法的reference

方法区

很少GC，又叫永久区(Permanent)。Java8开始，取而代之的是Metaspace元数据空间。

用于存储类信息（field、方法、接口、版本等信息）、类变量、final常量、编译器的字节码等

metaspace

**TODO2**

[Reference](http://ifeve.com/java-permgen-removed/)

aa

运行时常量池

aa

本地方法栈

aa

直接内存（Direct Memory）

直接内存并不是JVM管理的内存，是JVM以外的机器内存。JDK中有一种基于通道（Channel）和缓冲区（Buffer）的内存分配方式，将由C语言实现的native函数库分配在直接内存中，用存储在JVM堆中的DirectByteBuffer来引用。由于直接内存收到本机器内存的限制，所以也可能出现OutOfMemoryError的异常。

#### 内存分配

#### Cache

cache的意义

缓解CPU和内存之间速度的不匹配问题。

原理

如果CPU需要的内容在cache里

局部性原理。/2-8理论

* 时间局部性：如果某个数据被访问，那么在不久的将来它很可能被再次访问；
* 空间局部性：如果某个数据被访问，那么与它相邻的数据很快也可能被访问；

cache的写回方式

cache的写操作方式可以追溯到大学教程《计算机组成原理》一书。

* write through（写通）：每次CPU修改了cache中的内容，立即更新到内存。每次CPU写共享数据，都会导致总线事务，导致总线事务的竞争，效率非常低。
* write back（写回）：每次CPU修改了cache中的数据，不会立即更新到内存，而是等到cache line在某一个必须或合适的时机才会更新到内存中；

无论是写通还是写回，在多线程环境下都需要处理缓存cache一致性问题。

线程安全

从cache的角度，产生线程不安全的原因：

当程序在运行过程中，会将运算需要的数据从主存复制一份到CPU的高速缓存当中，那么CPU进行计算时就可以直接从它的高速缓存读取数据和向其中写入数据，当运算结束之后，再将高速缓存中的数据刷新到主存当中。

如果一个变量在多个CPU中都存在缓存（一般在多线程编程时才会出现），那么就可能存在缓存不一致的问题

缓存一致性

1）通过在总线加LOCK#锁的方式

2）通过缓存一致性协议，比如MESI协议

MESI协议

处理缓存cache一致性问题。

# J2EE

为企业开发而生.

J2EE规范

J2EE是由SUN提出的用于简化开发企业级应用程序的一系列规范的组合，J2EE基于中间层集成的框架的方式为应用开发提供了一个统一的开发平台、基于容器管理、组件化的模型为企业建立一个高可用性，高可靠性可扩展的应用平台提供支持，降低了开发分布式应用程序的难度，降低了开发成本，而且提供一整套方便与旧的遗留系统相集的强有力支持，并提供了安全性及规范的打包安装的支持。开发人员使用的J2EE规范开发的应用程序将受厂商及客户的支持。

J2EE是由一整套服务，应用程序接口及协议构成，提供对开发多层Web程序应用提供支持

JDBC/JNDI/EJB/JSP/SERVLET/XML/JMS/JavaMail/JAF

RMI

EJB使用的一种更底层的协议；

JTA

## JAVA Web

### Servlet规范

#### API简介

1. **public** **interface** Servlet {
2. **void** init(ServletConfig var1);
3. ServletConfig getServletConfig();
4. **void** service(ServletRequest var1, ServletResponse var2)
5. String getServletInfo();
6. **void** destroy();
7. }

init()和destroy() 管理生命周期，还有一个处理请求的service()

servlet容器

应用服务器，要处理请求还需要servlet容器,比如常用的tomcat，从原理上讲，Servlet可以响应任何类型的请求，但绝大多数情况下Servlet只用来扩展基于HTTP协议的Web服务器。

**处理过程**：

一个http请求到来，容器将请求封装成servlet中的request对象，在request中可以得到所有的http信息🡪然后取出来操作，最后再把数据封装成servlet的response对象🡪应用容器将respose对象解析之后封装成一个http response。

Security

1. 自定义filter
2. Spring-security
3. Shiro

权限

Oauth

生命周期

实例化🡺init🡺service🡺destroy。

HttpServletRequest

HTTP请求头中的所有信息都封装在这个对象中。

Code：RequestDemo01

* request.getRequestURL 获得客户机信息
* request.getParameter 表单POST数据
* request.getSession.setAttribute() 作用域是整个会话期间
* request.setAttribute()，作用域是请求和被请求页面之间。request.setAttribute()是只在此action的下一个forward需要使用的时候使用；

判断未登录或session过期

request.getSession (false) ==null

request.getSession (false).getAttribute (“user”) ==null

HttpServletResponse

response.sendRedirect (“redirectURL”);

Filter

过滤请求。过滤器的执行顺序跟xml文件中定义的先后关系有关。

过滤字符编码，逻辑判断，如是否登录，

权限控制

**Filter vs AOP**

AOP无法拦截直接访问静态资源的请求。Filter仅仅过滤请求。

Listener

它也是随web应用的启动而启动，只初始化一次，随web应用的停止而销毁。主要用做一些初始化

### 2.1.2 MVC

* 应用分层



1. **终端显示层**：PC端，移动端展示等。
2. **开放接口层**：可直接封装 Service 方法暴露成 RPC 接口； 通过 Web 封装成 http 接口； 需要进行网关安全控制、流量控制等。
3. **Web 层**：即Controller层，主要是对访问控制进行转发，各类基本参数校验，或者不复用的业务简单处理等。
4. **Service层**：相对具体的业务逻辑服务层。
5. **Manager 层**：通用业务处理层，它有如下特征：

1） 对第三方平台封装的层，预处理返回结果及转化异常信息；

1. 对 Service 层通用能力的下沉，如缓存方案、中间件通用处理；

3）与 DAO 层交互，对多个 DAO 的组合复用。

1. **DAO 层**：数据访问层，与底层 MySQL、Oracle、HBase 等进行数据交互。

7.**外部接口或第三方平台**：包括其它部门 RPC 开放接口，基础平台，其它公司的 HTTP 接口。

* 常规校验

在使用平台资源，譬如短信、邮件、电话、下单、支付，必须实现正确的防重放限制，

如数量限制、疲劳度控制、验证码校验，避免被滥刷、资损。

说明： 如注册时发送验证码到手机，如果没有限制次数和频率，那么可以利用此功能骚扰到其它用户，并造成短信平台资源浪费。

* 参数校验

用户请求传入的任何参数必须做有效性验证。

说明：忽略参数校验可能导致：

 page size 过大导致内存溢出

 恶意 order by 导致数据库慢查询

 任意重定向

 SQL 注入

 反序列化注入

 正则输入源串拒绝服务 ReDoS

说明：Java 代码用正则来验证客户端的输入，有些正则写法验证普通用户输入没有问题，

但是如果攻击人员使用的是特殊构造的字符串来验证，有可能导致死循环的结果。

**参数的安全性判断应该写在controller还是service？异常处理尼？**

哪一层都是很有必要的，一般在controller进行，在service层进行校验更好的复用。

* 统一异常处理

1．大多数情况下我们选择异常出现后只进行记录日志和UI用户提示

2．在web层设置统一异常处理机制

3．记录日志时，尽可能带上参数信息，便于调试

4．开放接口层要将异常处理成错误码和错误信息方式返回。页面要跳转到友好错误页面， 加上用户容易理解的错误提示信息。

* 自定义异常

DAO层和Service层，产生的异常类型有很多，无法用细粒度的异常进行 catch，避免直接throw new Exception,应使用有业务含义的自定义异常。推荐业界已定义过的自定义异常，如：DAOException / ServiceException等。

## 2.3 Spring

### Bean

生命周期

[Reference](https://www.zhihu.com/question/38597960)



作用域

singleton：生命周期同Spring Container，在容器中只有一个实例。

prototype：每次调用都会得到一个新的实例。

request, session, global session

以上三类是只有引入了Spring ApplicationContext才存在，生命周期大致对应HTTP中的Request, Session和Application。

TODO1:Spring管理的对象，通常是单例的吗？不然？为什么不都用单例的？

### Feature

AOP

方便进行面向切面的编程，很多传统OOP不容易实现的功能可以通过AOP轻松应对。

实现原理

1. Aspectj静态代理



2)CGLIB动态代理/JDK动态代理: JDK针对接口

Object obj=new Object();

如何对new做AOP？

声明式事务

通过声明的方式灵活地进行事务管理，开发不用写事务管理代码（提高开发效率和质量）。

**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。

### 事务管理

使用AOP切面实现



**Propagation**

传播行为，ServiceB.methodB 🡪 ServiceA.methodA

* REQUIRED：没有就起一个新的事务。methodA或methodB出现异常，事务回滚。
* SUPPORTS：支持当前事务，如果当前没有事务，就以非事务方式执行。
* NOT\_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
* REQUIRES\_NEW：新建事务，如果当前存在事务，把当前事务挂起。
* MANDATORY：支持当前事务，如果当前没有事务，就抛出异常。
* NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

**Isolation**

与数据库的隔离级别相对应。

（READ\_UNCOMMITTED/ READ\_COMMITTED/ REPEATABLE\_READ/ SERIALIZABLE）

**Rollback-for**

人为控制事务

**Read-only**

[Reference](https://www.cnblogs.com/hackem/p/3890656.html)

保证可重复读。mysql默认的Isolation(隔离级别)就是Repeatable-Read。Oracle默认是read commit，设置readonly后能达到同样效果。

spring ejb区别

## 应用服务器

### Tomcat

### 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

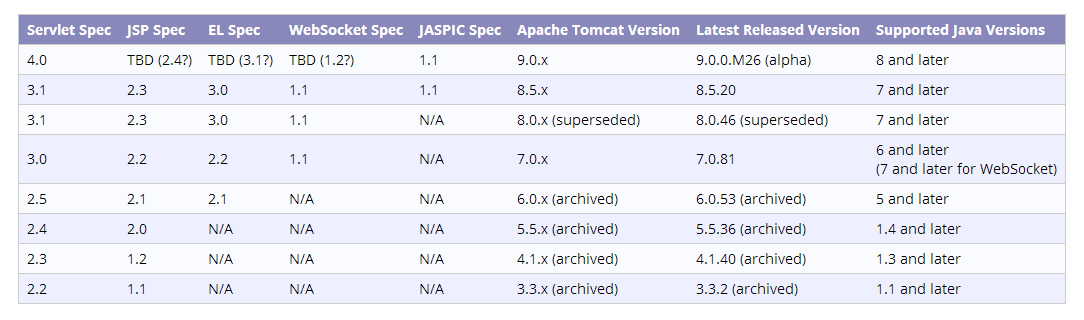
* 支持HTTP/2

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 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 3 实现原理

#### 4.1 架构分析

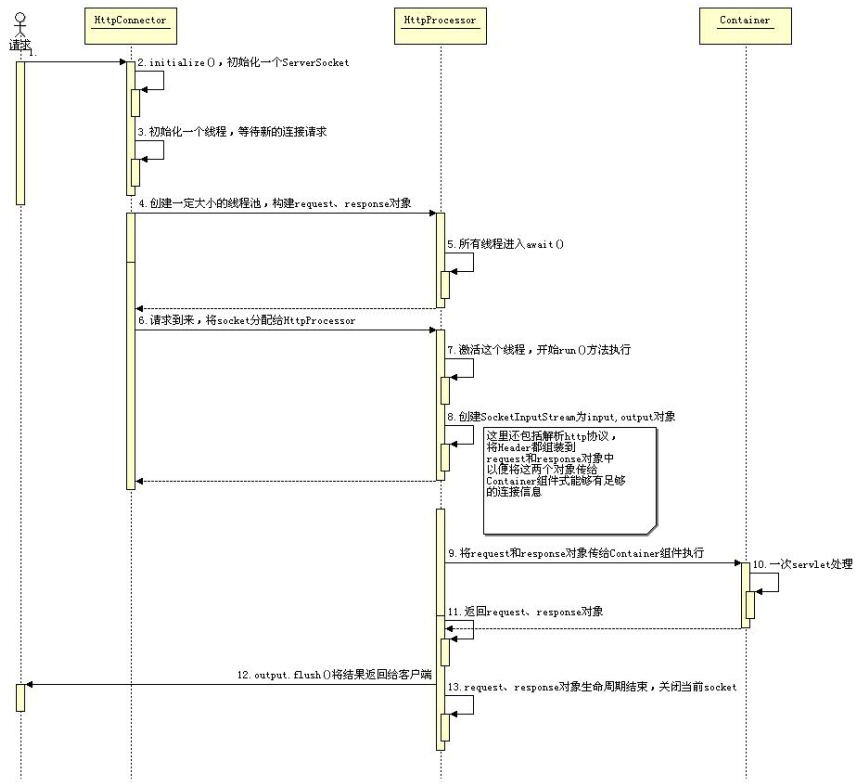
[源代码分析](https://www.ibm.com/developerworks/cn/java/j-lo-tomcat1/index.html)



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

## Tools

findBug

[Reference](http://www.cnblogs.com/f1194361820/p/4796116.html)

### Git

## 网络

# 第三篇 分布式应用

## 其他

### 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。

### Dubbo

TODO4

### Docker

虚拟机

cas单点登录

Zookeeper服务的注册和发现

Zookeeper配置中心

### SpringCloud

### Zookeeper

Zookeeper是一个高性能分布式应用协调服务。分布式应用可以基于它实现同步，配置管理，集群管理，命名名空间

 随着线上项目变的日益庞大，每个项目都散落着各种配置文件；因为采用分布式的开发模式，项目之间的相互引用随着服务的不断增多，相互之间的调用复杂度成指数升高，每次投产或者上线新的项目时苦不堪言，因此需要引用配置中心治理。

分布式琐。

### 分布式计算

### 分布式存储