# J2SE

## 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

## 1.5 Thread

### 1.5.1 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？

### 1.5.5 Concurrent包

#### 1.5.5.1 Pool

##### ExecutorService

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。

**submit VS execute**

Code: ThreadPoolFutureGetException101

submit有返回值，而execute没有,submit方便Exception处理。



Code: ThreadPool101



Future



* 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.

##### 实现原理

[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。

#### 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 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

## 1.6 JVM

### 运行时数据区

#### 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。

## 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

## 2.2 RPC远程调用

## Tools

findBug

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

### Git

## 网络

### NIO

非阻塞：

事件驱动：