1. **综上所述，ArrayList相当于在没指定initialCapacity时就是会使用延迟分配对象数组空间，当第一次插入元素时才分配10（默认）个对象空间。假如有20个数据需要添加，那么会分别在第一次的时候，将ArrayList的容量变为10 (如下图一)；之后扩容会按照1.5倍增长。也就是当添加第11个数据的时候，Arraylist继续扩容变为10\*1.5=15(如下图二)；当添加第16个数据时，继续扩容变为15 \* 1.5 =22个(如下图四)。**

**每次扩容都是通过Arrays.copyOf(elementData, newCapacity) 这样的方式实现的。**

1. **Redis采用的是基于内存的采用的是单进程模型的KV数据库**

**单进程单线程好处**

**代码更清晰，处理逻辑更简单**

**不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗**

**不存在多进程切换而消耗CPU**

**单进程单线程弊端**

**无法发挥多核CPU性能，不过可以通过在单机开多个Redis实例来完善；**

1. **NIO和IO主要区别**

| **特征** | **NIO** | **IO** |
| --- | --- | --- |
|  | **基于通道（Channel）和缓冲区** | **基于字节流和字符流** |
| **阻塞** | **非阻塞IO** | **阻塞IO** |
| **选择器** | **有选择器（监听多个通道的事件）** | **无选择器** |

**1、面向流与面向缓冲**

     Java IO和NIO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。 Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

**2、阻塞与非阻塞IO**

     Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取，而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

**3、选择器（Selectors）**

     Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

**Channel与Buffer：基本上所有的IO在NIO都从一个Channel开始。Channel有点像流，数据可以从Channel读到Buffer中，也可以从Buffer 写到Channel中。**

**JAVA NIO中，Channel的主要实现方法有：FileChannel、DatagramChannel、SocketChannel、ServerSocketChannel**

**JAVA NIO中，Buffer的主要实现方法有：ByteBuffer、CharBuffer、DoubleBuffer、FloatBuffer、IntBuffer等**

**Selector ：Selector允许单线程处理多个 Channel。例如在一个聊天服务器中，一个应用打开了多个连接（通道），但每个连接的流量都很低，使用Selector就会很方便。如图：**

**2：hashMap和hashTable区别**

1. Hashtable既不支持Null key也不支持Null value；HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null
2. Hashtable是线程安全的，它的每个方法中都加入了Synchronize方法；HashMap不是线程安全的，在多线程并发的环境下，可能会产生死锁等问题，可以使用concurentHashMap或者collections.synchronizedMap();
3. 初始容量大小和每次扩充容量大小的不同 ;Hashtable默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。
4. 计算hash值的方法不同 ;Hashtable直接使用对象的hashCode;Hashtable在计算元素的位置时需要进行一次除法运算，而除法运算是比较耗时的;

HashMap为了提高计算效率，使用两次hash，是数据更加分散，从而降低了hash冲突，并使用位运算代替取模，提高了计算效率；将哈希表的大小固定为了2的幂，这样在取模预算时，不需要做除法，只需要做位运算。位运算比除法的效率要高很多。

**3：JDK之HashMap中的indexFor(int h, int length)方法**

HashMap则更加关注hash的计算效率问题。在取模计算时，如果模数是2的幂，那么我们可以直接使用位运算来得到结果，效率要大大高于做除法

今天看到了HashMap，发现其中有一个方法很奇怪，叫**indexFor(int h, int length)**，这个方法返回的是key的hashcode对应到hash table的下标位置，代码是这么实现的：

**static** **int** indexFor(**int** h, **int** length) {

1. **return h & (length-1);**
2. }

于为什么不直接用hashcode%length，是**因为位运算要比模运算快一些**。

1. 分布式是个工作方式，一个业务分拆多个子业务，部署在多个服务器上，不同的服务器节点完成不同的任务；
2. 分布式是指将不同的业务分布在不同的地方。而集群指的是将几台服务器集中在一起，实现同一业务。就比如新浪网，访问的人多了，他可以做一个群集，前面放一个响应服务器，后面几台服务器完成同一业务，如果有业务访问的时候，响应服务器看哪台服务器的负载不是很重，就将给哪一台去完成。

集群：同一个业务，部署在多个服务器上。

分布式：一个业务分拆成多个子业务，或者本身就是不同的业务，部署在不同的服务器上。（用户模块、商品模块、订单模块、评论模块）  
简单说，分布式是以缩短单个任务的执行时间来提升效率的，而集群则是通过提高单位时间内执行的任务数来提升效率。举例：就比如新浪网，访问的人多了，他可以做一个群集，前面放一个均衡服务器，后面几台服务器完成同一业务，如果有业务访问的时候，响应服务器看哪台服务器的负载不是很重，就将给哪一台去完成，并且一台服务器垮了，其它的服务器可以顶上来。分布式的每一个节点，都完成不同的业务，一个节点垮了，那这个业务可能就失败了。

#### 负载均衡功能

1:转发请求

2:故障移除

3:恢复添加

1. **公共第一个节点：**我们先要分别遍历两个链表得到它们的长度，并求出两个长度之差。在长的链表上先遍历若干次之后，再同步遍历两个链表，知道找到相同的结点，或者一直到链表结束。此时，如果第一个链表的长度为m，第二个链表的长度为n，该方法的时间复杂度为O(m+n)。

**1.判断链表是否有环**

设置两个指针slow和fast,初始值均指向链表头,slow每次向前走一步,而fast每次向前走两步.

如果链表有环,则fast先进入环里,而slow后进入环里,两个指针在环中必定相遇.

如果slow与fast没有相遇,fast遍历到链表的尾部,则表示链表没有环.

**2.链表有环,确定环的入口点**

设置slow指针指向链表头,fast指向相遇点,每次两个指针都是只走一步,两个指针必定相遇,

则相遇第一点为环入口点.

1. **Arrays.sort和Collections.sort实现原理解析**

基本类型的排序都是使用了了DualPivotQuicksort的排序方法(我看的是jdk7，)。DualPivotQuicksort是快排的一种优化，具体在这里不展开了。Java1.7的快排是一种双轴快排，顾名思义：双轴快排是基于两个轴来进行比较，跟普通的选择一个点来作为轴点的快排是有很大的区别。两枢轴，就是两个结点，把数据划分成三部分，三部分再分别递归。--->荷兰国旗问题

 当参数类型为对象数组时，在原来的版本使用的归并排序（以后将会删除 ），现在使用的timSort。（使用二分插入排序+归并排序）

1. 事实上**Collections.sort方法底层就是调用的array.sort方法**，而且不论是Collections.sort或者是Arrays.sort方法
2. 在Java 6中Arrays.sort()和Collections.sort()使用的是**MergeSort**，而在Java 7中，内部实现换成了**[TimSort](http://en.wikipedia.org/wiki/Timsort)**，其对对象间比较的实现要求更加严格：
3. Timsort是结合了**合并排序（merge sort）和二分插入排序（insertion sort）**而得出的排序算法，它在现实中有很好的效率。小于32使用二分插入排序，大于32使用归并排序
4. **Linux命令**

**文件：**-rwxrw-r‐-1 root root 1213 Feb 2 09:39 abc

- 10个字符确定不同用户能对文件干什么

- 第一个字符代表文件（-）、目录（d），链接（l）

- 其余字符每3个一组（rwx），读（r）、写（w）、执行（x）

- 第一组rwx：文件所有者的权限是读、写和执行

- 第二组rw-：与文件所有者同一组的用户的权限是读、写但不能执行

- 第三组r--：不与文件所有者同组的其他用户的权限是读不能写和执行

也可用数字表示为：r=4，w=2，x=1  因此rwx=4+2+1=7

chmod 755 abc：赋予abc权限rwxr-xr-x

1. 所以lsof可以用来查找谁用了某一个端口。具体方法：**lsof -i** :port\_number |grep "(LISTEN)"
2. 例如查看tcp有哪些端口打开了：**netstat -ano** | grep tcp
3. 查找文件名中包含某字符（如"elm"）的文件**find . -name '\*elm\*'**

从根目录开始查找所有扩展名为.trc的文本文件，并找出包含“DTB5100006”的行

find . -name "\*.trc" | grep "DTB5100006"

find . -name "\*.trc" | grep "15099998888"

1. **Find （-name 、-perm 、 -size）**
2. find / -name passwd # 查找文件名为passwd的文件
3. find . -perm 0755 # 查找当前目录中文件权限的0755的文件
4. **find . -size +12k #**查找当前目录中大于12KB的文件，注意c表示byte

**5）cp命令**

1. cp -a file1 file2 #连同文件的所有特性把文件file1复制成文件file2
2. **cp file1 file2 file3 dir**#把文件file1、file2、file3复制到目录dir中
3. **mv命令**
4. **mv file1 file2 file3 dir** # 把文件file1、file2、file3移动到目录dir中
5. **mv file1 file2 #** 把文件file1重命名为file2

****1.  Xargs用法详解****

****由于很多命令不支持|管道来传递参数，而日常工作中有有这个必要，所以就有了xargs命令，例如：****

****find /sbin -perm +700 |ls -l       这个命令是错误的****

****find /sbin -perm +700 |xargs ls -l   这样才是正确的****

****tar –zxvf file  ----压缩****

****tar –cxvf 文件名 file ----解压缩****

****实例1：从远处复制文件到本地目录****

****scp   root@192.168.120.204:/opt/soft/nginx-0.5.38.tar.gz    /opt/soft/****

****df -h 查看磁盘使用情况****

**du -h /opt/test 查看/opt/test目录的磁盘使用情况**

****Free -g 查看内存使用情况****

****wget –l chttp://****

#查看desc.txt的内容cat desc.txt

#分页查看desc.txt的内容**more desc.txt**

#查看desc.txt的最后100行内容**tail -100 desc.txt**

#创建目录/opt/test的符号链接**ln -s /opt/test ./link2test**

#列出所有端口**netstat -a**

#查看/opt/test目录的磁盘使用情**况du -h /opt/test**

#显示系统**当前进程信息top**

1. 改变文件或者目录的访问权限**：chomd +755 abc**

#查看ls命令的帮助文档**man ls**

**18：More和less**

less 与 more 类似，但使用 less 可以随意浏览文件，可以向前翻看或者向后翻看，而 more 仅能向前移动，却不能向后移动，**而且 less 在查看之前不会加载整个文件。**

实例1：ps查看进程信息并通过less分页显示同时显示行号 ps -ef|less -N

实例2.浏览多个文件 less test2.log test.log 按n和p

[pagedown]： 向下翻动一页

[pageup]：   向上翻动一页

/字符串：向下搜索“字符串”的功能

?字符串：向上搜索“字符串”的功能

 在 more 的时候，我们并没有办法向前面翻， 只能往后面看，但若使用了 less 时，就可以使用 [pageup] [pagedown] 等按 键的功能来往前往后翻看文件，更容易用来查看一个文件的内容！除此之外，在 less 里头可以拥有更多的搜索功能，不止可以向下搜，也可以向上搜。

1. **http 的长连接和短连接  
   HTTP 协议有 HTTP/1.0 版本和 HTTP/1.1 版本**。 **HTTP1.1 默认保持长连接，数据传输完成了保持 TCP 连接不断开（不发 RST 包、不四次握手），下次继续用这个通道传输数据**；相反的就是短连接。  
   在 HTTP/1.0 中，默认使用的是短连接。也就是说，**浏览器和服务器每进行一次 HTTP 操作，就建立一次连接，任务结束就中断连接**。 从 HTTP/1.1 起，默认使用的是长连接， 用以保持连接特性。
2. **http 常见的状态码有哪些？**

200 OK //客户端请求成功  
400 Bad Request //客户端请求有语法错误， 不能被服务器所理解  
401 Unauthorized //请求未经授权，

403 Forbidden //服务器收到请求，但是拒绝提供服务  
404 Not Found //请求资源不存在，

500 Internal Server Error //服务器发生不可预期的错误  
503 Server Unavailable //服务器当前不能处理客户端的请求，一段时间后可能恢复正常

1. **如何解决 get 和 post 乱码问题？**

1.1 **解决 post 请求乱码**:我们可以在 **web.xml 里边配置一个** CharacterEncodingFilter 过滤器。 设置为 utf-8.  
1.2 **解决 get 请求的乱码**:有两种方法。对于 get 请求中文参数出现乱码解决方法有两个: 1）**修改 tomcat 配置文件添加编码与工程编码一致。**

1. **NIO和BIO**

****面向Stream和面向Buffer****

Java NIO和IO之间最大的区别是IO是面向流（Stream）的，NIO是面向块（buffer）的，所以，这意味着什么？

**面向流意味着从流中一次可以读取一个或多个字节**，拿到读取的这些做什么你说了算，这里没有任何缓存而且只能顺序从流中读取数据，如果需要跳过一些字节或者再读取已经读过的字节，你必须将从流中读取的数据先缓存起来。

**面向块的处理方式有些不同，数据是先被 读/写到buffer中的，根据需要你可以控制读取什么位置的数据或者读取多少数据。**然而，你需要额外做的工作是检查你需要的数据是否已经全部到了buffer中，你还需要保证当有更多的数据进入buffer中时，buffer中未处理的数据不会被覆盖

****阻塞IO和非阻塞IO****

**所有的Java IO流都是阻塞的**，这意味着，当一条线程执行read()或者write()方法时，**这条线程会一直阻塞直到读取到了一些数据或者要写出去的数据已经全部写出，在这期间这条线程不能做任何其他的事情！**

**java NIO的非阻塞模式允许一条线程从channel中读取数据，通过返回值来判断buffer中是否有数据，如果没有数据，NIO不会阻塞，这条线程就可以去做其他的事情，过一段时间再回来判断一下有没有数据**

NIO的写也是一样的，**一条线程将buffer中的数据写入channel，它不会等待数据全部写完才会返回，而是调用完write()方法就会继续向下执行**

****Selectors****

**Java NIO的selectors允许一条线程去监控多个channels的输入**，你可以向一个selector上注册多个channel，然后调用selector的**select()方法判断是否有新的连接进来或者已经在selector上注册时channel是否有数据进入**。selector的机制让**一个线程管理多个channel**变得简单。

1. **IO是面向流的，NIO是面向缓冲的；**
2. **IO是阻塞的，NIO是非阻塞的；**
3. **IO是单线程的，NIO 是通过选择器来模拟多线程的；**
4. **GET 1kb和 POST 的区别？**

从表面现像上面看 GET 和 POST 的区别：  
1. GET 请求的数据会附在 URL 之后，**以?分割 URL 和传输数据**，参数之间  
以&相连，如： login.action?name=zha&password=12。 POST 把提交的数据则放置在是 HTTP 包的包体中。  
2. **GET方式提交的数据最多只能是 1024 字节**，**理论上 POST 没有限制**，可传较大量的数据。其实这样说是错误的，不准确的：  
3.**POST 的安全性要比 GET 的安全性高。**注意：通过 GET 提交数据，用户名和密码将**明文出现在 URL 上**，因为(1)登录页面有可能被**浏览器缓存**， (2)其他人查看浏览器的**历史纪录**，那么别人就可以拿到你的账号和密码了，

1. **http 中重定向和请求转发的区别？**  
   本质区别： 转发是服务器行为，重定向是客户端行为。  
   **重定向特点：两次请求**，浏览器地址发生变化，可以访问自己 web 之外的资源，传输的数据会丢失。  
   **请求转发特点：一次请求**，浏览器地址不变，访问的是自己本身的 web 资源，传输的数据不会丢失。
2. **Cookie 和 session 的不同点：**1、无论客户端做怎样的设置， session 都能够正常工作。当客户端禁用 cookie 时将无法使用 cookie。
3. 在存储的数据量方面： session 能够存储任意的 java 对象， cookie 只能存储 String 类型的对象
4. Cookie方法的是把数据保存在客户端，而Session是把数据保存在服务器

1、cookie数据存放在客户的浏览器上，session数据放在服务器上。

2、cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗  
   考虑到安全应当使用session。

3、session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能  
   考虑到减轻服务器性能方面，应当使用COOKIE。

4、单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。

1. **Redis 的特点？**

Redis是一种高级key-value数据库。它跟memcached类似，不过数据可以持久化，而且支持的数据类型很丰富。有字符串，链表，集 合和有序集合。支持在服务器端计算集合的并，交和补集(difference)等，还支持多种排序功能。所以Redis也可以被看成是一个数据结构服务 器。  
Redis的所有数据都是保存在内存中，然后不定期的通过异步方式保存到磁盘上(这称为“半持久化模式”)；也可以把每一次数据变化都写入到一个append only file(aof)里面(这称为“全持久化模式”)。

由于Redis的数据都存放在内存中，如果没有配置持久化，redis重启后数据就全丢失了，于是需要开启redis的持久化功能，将数据保存到磁盘上，当redis重启后，可以从磁盘中恢复数据。

半持久化模式是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。

全持久化模式以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。

1. 典型的 NoSQL 数据库服务器， Redis 是一个 key-value 存储系统，它支持丰富的数据类型，如： string、 list、 set、 zset(sorted set)、 hash
2. Redis 本质上是一个 Key-Value 类型的**内存数据库**，很像 memcached，**整个 数据库统统加载在内存当中进行操作**，**定期通过异步操作把数据库数据 flush 到硬盘 上进行保存**。因为是**纯内存操作， Redis 的性能非常出色**，**每秒可以处理超过 10 万次读写操作**，是已知性能最快的 Key-Value DB。
3. Redis 的主要缺点是数据库容量受到物理内存的限制，**不能用作海量数据的高性能读写**，因此 Redis 适合的场景主要局限**在较小数据量的高性能读写**。
4. Redis 为了达到最快的读写速度将**数据都读到内存中**，并通过**异步的方式将数据写入磁盘**。所以 redis 具有**快速和数据持久化的特征**。如果不将数据放在内存中，**磁盘 I/O 速度为严重影响 redis 的性能。**
5. **优点：**  
   **a) 性能极高 – Redis 能支持超过 100K+ 每秒的读写频率。  
   b) 支持丰富的数据类型 – Redis 支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。**

**c）Redis支持数据的备份，即master-slave模式的数据备份。  
d）Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。**

**e）可以用于缓存数据、实现消息队列、限流防刷等等功能**

**6、区别**

**数据类型**

redis数据类型丰富，支持set liset等类型

memcache支持简单数据类型，需要客户端自己处理复杂对象

**持久性**

redis支持数据落地持久化存储

memcache不支持数据持久存储

**分布式存储**

redis支持master-slave复制模式

memcache可以使用一致性hash做分布式

**数据一致性不同**

redis使用的是单线程模型，保证了数据按顺序提交。

memcache需要使用cas保证数据一致性。CAS（Check and Set）是一个确保并发一致性的机制，属于“乐观锁”范畴；原理很简单：拿版本号，操作，对比版本号，如果一致就操作，不一致就放弃任何操作

1. **JAVA 8**
2. **Lambda表达式（也称为闭包）**是整个Java 8发行版中最受期待的在Java语言层面上的改变。一个lambda可以由用逗号分隔的参数列表、–>符号与函数体三部分表示。

Arrays.asList( "a", "b", "d" ).forEach( e -> System.out.println( e ) );

1. **接口的默认方法与静态方法**

**private** **interface** Defaulable {

**default** String notRequired() {

**return** "Default implementation";

    }

}

默认方法与抽象方法不同之处在于抽象方法必须要求实现，但是默认方法则没有这个要求。相反，**每个接口都必须提供一个所谓的默认实现，这样所有的接口实现者将会默认继承它**（如果有必要的话，可以覆盖这个默认实现）

1. **Optional**是Java8提供的为了**解决null安全问题的一个API**

**public** **static** String getName(User u) {

**if** (u == **null**)

**return** "Unknown";

**return** u.name;

}

**public** **static** String getName(User u) {

**return** **Optional.ofNullable(u)**

**.map(user->user.name)**

**.orElse("Unknown");**

}

1. Java 8之所以费这么大功夫**引入函数式编程Stream**，原因有二：

**代码简洁，可读性强**，函数式编程写出的代码简洁且意图明确，使用stream接口让你从此告别for循环。

**多核友好**，**parallel()方法** Java函数式编程使得编写并行程序从未如此简单，你需要的全部就是调用一下**parallel()方法。**

1. **Stream API**

1、无存储。**stream不是一种数据结构，它只是某种数据源的一个视图**，数据源可以是一个数组，Java容器或I/O channel等。**对stream的任何修改都不会修改背后的数据源。**

2、**为函数式编程而生。对stream的任何修改都不会修改背后的数据源**，比如对stream执行过滤操作并不会删除被过滤的元素，而是会产生一个不包含被过滤元素的新stream。

3、// 使用Stream.forEach()迭代

Stream<String> stream = Stream.of("I", "love", "you", "too"); **stream.forEach(**str -> System.out.println(str));

1. // 保留长度等于3的字符串

Stream<String> stream= Stream.of("I", "love", "you", "too");

**stream.filter(**str -> str.length()==3)

.forEach(str -> System.out.println(str));

5、Stream<String> stream= Stream.of("I", "love", "you", "too", "too");

**stream.distinct()**

.forEach(str -> System.out.println(str));

上述代码会输出去掉一个too之后的其余字符串。

1. Stream<String> stream= Stream.of("I", "love", "you", "too");

**stream.sorted((**str1, str2) -> str1.length()-str2.length())

.forEach(str -> System.out.println(str));

1. Stream<String> stream　= Stream.of("I", "love", "you", "too");

**stream.map(**str -> str.toUpperCase())

.forEach(str -> System.out.println(str));

1. **[AtomicInteger源码分析——基于CAS的乐观锁实现](http://www.baidu.com/link?url=gBzabL_T6M50sHkgy1X21a3aGHxr0eh2k24tsNa11QgK_ckiYKluz-tXuzgZbzKU" \t "https://www.baidu.com/_blank)**

**//volatile----保证可见性**

**private volatile int value;**

    //构造器

    public AtomicInteger(int initialValue) {

        value = initialValue;

    }

    public AtomicInteger() {

    }

......

**//CAS机制---保证原子性**

**getAndIncrement()；**

**高并发的数据库乐观锁实现有两种方案，一种是在表里面添加Version字段，另外一种方案就是使用JDK的Atomic**

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如Java里面的同步原语synchronized关键字的实现也是悲观锁。

乐观锁：顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write\_condition机制，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。

CAS的全称是Compare And Swap 即比较交换，其算法核心思想如下:

执行函数：CAS(V,E,N): V表示要更新的变量;E表示预期值;N表示新值

如果V值等于E值，则将V的值设为N。若V值和E值不同，则说明已经有其他线程做了更新，则当前线程什么都不做。

解决ABA问题：

1. AtomicStampedReference原子类是一个带有时间戳的对象引用，在每次修改后，AtomicStampedReference不仅会设置新值而且还会记录更改的时间。当AtomicStampedReference设置对象值时，对象值以及时间戳都必须满足期望值才能写入成功，这也就解决了反复读写时，无法预知值是否已被修改的窘境
2. 表中添加Version字段
3. **单个 Lock 可能与多个 Condition 对象关联**

**Condition fullLock = lock.newCondition();**

**Condition emptyLock = lock.newCondition();**

1. **java.util.concurrent 包描述：**

 执行程序（线程池）  
 并发队列  
 同步器  
 并发 Collocation

ReentrantReadWriteLock

可重入读写锁（读写锁的一个实现）

　ReentrantReadWriteLock lock = new ReentrantReadWriteLock()

　ReadLock r = lock.readLock();

　WriteLock w = lock.writeLock();

两者都有lock,unlock方法。写写，写读互斥；读读不互斥。可以实现并发读的高效线程安全代码

而ReentrantReadWriteLock允许多个读线程同时访问，但不允许写线程和读线程、写线程和写线程同时访问。相对于排他锁，提高了并发性。在实际应用中，大部分情况下对共享数据（如缓存）的访问都是读操作远多于写操作

示例二：使用 ReentrantReadWriteLock 来提高 Collection 的并发性

class RWDictionary {

private final Map<String, Data> m = new TreeMap<String, Data>();

private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

private final Lock r = rwl.readLock(); //读锁

private final Lock w = rwl.writeLock(); //写锁

public Data get(String key) {

r.lock();

try { return m.get(key); }

finally { r.unlock(); }

}

public String[] allKeys() {

r.lock();

try { return m.keySet().toArray(); }

finally { r.unlock(); }

}

public Data put(String key, Data value) {

w.lock();

try { return m.put(key, value); }

finally { w.unlock(); }

}

public void clear() {

w.lock();

try { m.clear(); }

finally { w.unlock(); }

}

}

1. **ThreadPoolExecutor 线程池执行者**

ThreadPoolExecutor 是 ExecutorService 接口的一个实现 ，

ThreadPoolExecutor执行顺序：

线程池按以下行为执行任务

**当线程数小于核心线程数时，创建线程。**

**当线程数大于等于核心线程数，且任务队列未满时，将任务放入任务队列。**

**当线程数大于等于核心线程数，且任务队列已满**

**若线程数小于最大线程数，创建线程**

**若线程数等于最大线程数，根据“拒绝策略”进行相应处理**

****重点讲解：****   
其中比较容易让人误解的是：**corePoolSize，maximumPoolSize，workQueue**之间关系。   
1.当线程池小于corePoolSize时，新提交任务将创建一个新线程执行任务，即使此时线程池中存在空闲线程。   
2.当线程池达到corePoolSize时，新提交任务将被放入workQueue中，等待线程池中任务调度执行   
3.当workQueue已满，且maximumPoolSize>corePoolSize时，新提交任务会创建新线程执行任务   
4.当提交任务数超过maximumPoolSize时，新提交任务由RejectedExecutionHandler处理   
5.当线程池中超过corePoolSize线程，**空闲时间达到keepAliveTime时，关闭空闲线程   
构造方法参数列表解释：**  
**corePoolSize** - 池中所保存的线程数，包括空闲线程。  
**maximumPoolSize** - 池中允许的最大线程数。  
**keepAliveTime** - 当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间。  
**unit** - keepAliveTime 参数的时间单位。  
**workQueue** - 执行前用于保持任务的队列。此队列仅保持由 execute 方法提交的 Runnable 任务。

**Handler**- 拒绝策略RejectedExecutionHandler；

****AbortPolicy：****如果线程池队列满了丢掉这个任务并且抛出RejectedExecutionException异常

****DiscardPolicy：****如果线程池队列满了，会直接丢掉这个任务并且不会有任何异常

****DiscardOldestPolicy：****丢弃最老的。也就是说如果队列满了，会将最早进入队列的任务删掉腾出空间，再尝试加入队列

****CallerRunsPolicy：****使用此策略，如果添加到线程池失败，**那么主线程会自己去执行该任务**，不会等待线程池中的线程去执行

**ExcutorService pool = Excutors.newFixedThreadPool();**

**Pool.submit(()->{**

**})**

**14：[mysql那些事](https://www.cnblogs.com/phpfensi/p/8143299.html)**

1. 手机号最好使用varchar(20)的方式进行存储，这样存储可以支持模糊查询；
2. 存储小数(浮点数)-将数据扩大10的倍数达到使用整数类型存储目的；或者使用decimal类型
3. SELECT DISTINCT name,id FROM pp\_test WHERE `name`<>'' ORDER BY id DESC这句话查询的是name+id不重复的
4. SELECT DISTINCT name FROM **(SELECT name FROM pp\_test WHERE `name`<>''**

**ORDER BY id DESC)**

10.条件查询的执行顺序:   
(1)先执行FROM,确定查哪一个张表,   
(2)再执行Where,过滤,   
(3)接着执行SELECT,筛选哪一些列   
(4)接着执行ORDER子句,对查询的结果排序   
(5)再接着执行GROUP BY子句,分组查询   
(6)再接着执行HAVING子句,对分组的结果再筛选

1. **myisam引擎使用b+tree作为索引结构，叶节点的data域存放的是数据记录的地址**

**MyISAM索引的实现：MyISAM索引文件和数据文件是分离的，索引文件中保存是数据记录所在的地址，通过这些地址来读取到数据记录；**

1. **InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，InnoDB的所有辅助索引都引用主键作为data域**

**mysql中，创建表的时候，指定引擎：**

**create table test(**

**id int(10) unsigned not null auto\_increment,**

**name varchar(10) character set utf8,**

**age int(10),**

**primary key(id)**

**)engine = MyISAM default charset = utf8;**

**InnoDB索引**和**MyISAM索引**的区别：

**一是主索引的区别**，InnoDB的数据文件本身就是索引文件，**叶节点data域保存了完整的数据记录**。而MyISAM的索引和数据是分开的，**叶节点data域保存了数据记录的地址**。

**二是辅助索引的区别**：InnoDB的辅助索引data域存储相应记录**主键的值而不是地址**。而MyISAM的辅助索引和主索引没有多大区别，。

**说明：1、为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。再例如，2、用非单调的字段作为主键在InnoDB中不是个好主意，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。**

1. **[分布式缓存技术redis](https://blog.csdn.net/pfnie/article/details/52234646)**
2. **redis是NoSQL（No Only SQL，非关系型数据库）的一种，NoSQL是以Key-Value的形式存储数据。当前主流的分布式缓存技术redis,mongodb等。既可以把redis理解为缓存技术。**
3. **Redis支持的数据类型要丰富得多，常用的数据类型主要有五种：String、List、Hash、Set和Sorted Set。**
4. 线程池
5. **创建：ExecutorService pool = Executors.newSingleThreadExecutor();**

**newCachedThreadPoor();** **newFixedThreadPoor();**

1. **执行：**

- execute(Runnable) 没有办法获知task的执行结果

- submit(Runnable) 返回一个Future对象，我们可以检查提交的任务是否执行完毕

- submit(Callable) 会**返回一个Future对象**，但是除此之外，submit(Callable)接收的是一个Callable的实现，Callable接口中的call()方法有一个返回值，**可以返回任务的执行结果**

- invokeAny(...)

- invokeAll(...)

1. **关闭线程池**

shutdown()或者shutdownNow()执行完毕

**1.2 Cglib动态代理（目标类未实现接口）**

**1）CGLib采用了非常底层的字节码技术，通过字节码技术为目标类创建子类代理对象，**

**2）并在子类中采用方法拦截的技术拦截所有父类方法的调用，并在不改变父类方法的基础上，对父类的功能进行扩充。**

**CGLib创建的动态代理对象性能比JDK创建的动态代理对象的性能高不少，但是CGLib在创建代理对象时所花费的时间却比JDK多得多**

**1.3 JDK动态代理**

**1）java的动态代理机制中，有两个重要的部分，一个是 InvocationHandler(Interface)、另一个则是 Proxy(Class)**

**2）Proxy这个类的作用就是用来动态创建一个代理对象，newProxyInstance 这个方法**

**3）接口：每个动态代理类都必须要实现InvocationHandler这个接口，并重写invoke（）这个方法，当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为invoke 方法来进行调用委托类中的方法。**

****动态代理步骤****：  
1.创建一个实现接口InvocationHandler的类，它必须实现invoke方法

publicobject invoke(Object obj,Method method, Object[] args)

在实际使用时，第一个参数obj一般是指代理类，method是被代理的方法，如上例中的request()，args为该方法的参数数组。这个抽象方法在代理类中动态实现。

public class InvocationHandlerImpl implements InvocationHandler

{

// 这个就是我们要代理的真实对象

//通过动态代理对象，真正访问的是委托类中的方法

private Object subject;

public InvocationHandlerImpl(Object subject)

{

this.subject = subject;

}

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable

{

//在代理真实对象前我们可以添加一些自己的操作

System.out.println("在调用之前，我要干点啥呢？");

System.out.println("Method:" + method);

//当代理对象调用真实对象的方法时，其会自动的跳转到代理对象关联的handler对象的invoke方法来进行调用

Object returnValue = method.invoke(subject, args);

//在代理真实对象后我们也可以添加一些自己的操作

System.out.println("在调用之后，我要干点啥呢？");

return returnValue;

}

}

2.创建被代理的类以及接口

接口：

// 需要动态代理的接口

public interface Subject

{

public String SayHello(String name);

public String SayGoodBye();

}

**动态代理类（委托类，实际对象）**

public class RealSubject implements Subject

{

public String SayHello(String name)

{

return "hello " + name;

}

public String SayGoodBye()

{

return " good bye ";

}

}  
3.通过Proxy的静态方法  
newProxyInstance(ClassLoaderloader, Class[] interfaces, InvocationHandler h)创建一个代理

public class DynamicProxyDemonstration

{

public static void main(String[] args)

{

//代理的真实对象

Subject realSubject = new RealSubject();

/\*\*

\* InvocationHandlerImpl 实现了 InvocationHandler 接口，并能实现方法调用从代理类到委托类的分派转发

\* 即：要代理哪个真实对象，就将该对象传进去，最后是通过该真实对象来调用其方法

\*/

InvocationHandler handler = new InvocationHandlerImpl(realSubject);

ClassLoader loader = realSubject.getClass().getClassLoader();

Class[] interfaces = realSubject.getClass().getInterfaces();

/\*\*

\* 该方法用于为指定类装载器、一组接口及调用处理器生成动态代理类实例

\*/

Subject subject = (Subject) Proxy.newProxyInstance(loader, interfaces, handler);//代理类

System.out.println("动态代理对象的类型："+subject.getClass().getName());

//当代理对象调用真实对象的方法时，其会自动的跳转到代理对象关联的handler对象的invoke方法来进行调用

String hello = subject.SayHello("jiankunking");

System.out.println(hello);

// String goodbye = subject.SayGoodBye();

// System.out.println(goodbye);

}

}

  1：首先，定义业务逻辑接口

public interface BookFacade {

public void addBook();

}

 2：然后，实现业务逻辑接口创建业务实现类

public class BookFacadeImpl implements BookFacade {

@Override

public void addBook() {

System.out.println("增加图书方法。。。");

}

}

 3：最后，实现 调用管理接口InvocationHandler  创建动态代理类

public class BookFacadeProxy implements InvocationHandler {

private Object target;//这其实业务实现类对象，用来调用具体的业务方法

/\*\*

\* 绑定业务对象并返回一个代理类

\*/

public Object bind(Object target) {

this.target = target; //接收业务实现类对象参数

//通过反射机制，创建一个代理类对象实例并返回。用户进行方法调用时使用

//创建代理对象时，需要传递该业务类的类加载器（用来获取业务实现类的元数据，在包装方法是调用真正的业务方法）、接口、handler实现类

return Proxy.newProxyInstance(target.getClass().getClassLoader(),

target.getClass().getInterfaces(), this); }

/\*\*

\* 包装调用方法：进行预处理、调用后处理

\*/

public Object invoke(Object proxy, Method method, Object[] args)

throws Throwable {

Object result=null;

System.out.println("预处理操作——————");

//调用真正的业务方法

result=method.invoke(target, args);

System.out.println("调用后处理——————");

return result;

}

}

public static void main(String[] args) {

BookFacadeImpl bookFacadeImpl=new BookFacadeImpl();

BookFacadeProxy proxy = new BookFacadeProxy();

BookFacade bookfacade = (BookFacade) proxy.bind(bookFacadeImpl);

bookfacade.addBook();

}

**cglib是针对类来实现代理的，原理是对指定的业务类生成一个子类，并覆盖其中业务方法实现代理。因为采用的是继承，所以不能对final修饰的类进行代理。**

1：首先定义业务类，无需实现接口（当然，实现接口也可以，不影响的）

public class BookFacadeImpl1 {

public void addBook() {

System.out.println("新增图书...");

}

}

2：实现 MethodInterceptor方法代理接口，创建代理类

public class BookFacadeCglib implements MethodInterceptor {

private Object target;//业务类对象，供代理方法中进行真正的业务方法调用

//相当于JDK动态代理中的绑定

public Object getInstance(Object target) {

this.target = target; //给业务对象赋值

Enhancer enhancer = new Enhancer(); //创建加强器，用来创建动态代理类 enhancer.setSuperclass(this.target.getClass()); //为加强器指定要代理的业务类（即：为下面生成的代理类指定父类）

//设置回调：对于代理类上所有方法的调用，都会调用CallBack，而Callback则需要实现intercept()方法进行拦  
 enhancer.setCallback(this);   
 // 创建动态代理类对象并返回   
 return enhancer.create();   
 }  
 // 实现回调方法   
 public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {   
 System.out.println("预处理——————");  
 proxy.invokeSuper(obj, args); //调用业务类（父类中）的方法  
 System.out.println("调用后操作——————");  
 return null;   
 }

    3：创建业务类和代理类对象，然后通过  代理类对象.getInstance(业务类对象)  返回一个动态代理类对象（它是业务类的子类，可以用业务类引用指向它）。最后通过动态代理类对象进行方法调用。

public static void main(String[] args) {

BookFacadeImpl1 bookFacade=new BookFacadeImpl1()；

BookFacadeCglib cglib=new BookFacadeCglib();

BookFacadeImpl1 bookCglib=(BookFacadeImpl1)cglib.getInstance(bookFacade);

bookCglib.addBook();

}

1. spring学习——Ioc基础四（Di之循环依赖）

**（构造注入不能解决，会抛出异常，set方式注入可以解决单态bean的循环依赖问题）**

1. 如何优化一个有100万条记录的数据库表

**一：使用缓存和使用索引；  
二：按照时间来创建分区或者分表；  
三：定期将老数据备份到其他位置；  
四：尽量通过分页查询的形式进行，不要一次性读取过多数据**

一：如果有唯一字段的，创建主键和索引，提高查询主键字段效率；  
二：如果这些数据是变更的，可以通过时间来创建分区；  
三：如果部分老数据都是不需要的数据（不常用），可以定期将老数据备份到其他位置；  
四：如果是查询的话，可以通过分页查询的形式进行，不要一次性读取过多数据。

1. **JMM(java memory model)**，在JMM中**主内存**包括了**堆和方法区**，而**工作内存**数据线程私有数据区域，**包括程序计数器、虚拟机栈以及本地方法栈**。

**1）方法中的本地变量是基本数据类型**，将直接存储在工作内存的帧栈结构中，但倘**若本地变量是引用类型**，那么该变量的引用会存储在功能内存的帧栈中，而对象实例将存储在主内存(共享数据区域，堆)中。

**2）但对于实例对象的成员变量**，不管它是基本数据类型或者包装类型(Integer、Double等)还是引用类型，都会被存储到堆区。以及static变量以及类本身相关信息将会存储在主内存中。

**3）需要注意的是**，**在主内存中的实例对象可以被多线程共享**，倘若两个线程同时调用了同一个对象的同一个方法，那么两条线程会将要操作的数据拷贝一份到自己的工作内存中，执行完成操作后才刷新到主内存中去。

1. **[jvm的内存分配总结](http://www.cnblogs.com/lifescolor/p/5481588.html)**

**线程共享区域**为：

1. java堆

 java堆细分为新生代和老年代，新生代又分为Eden空间、From Survivor空间、To Survivor空间，新生代中垃圾回收算法为复制算法。

  老年代中采用的GC算法为标记-清除算法或者标记-整理算法。

1. 方法区

方法区又被称为永久区，线程共享，是用来存储已被JVM加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

  这个区域很少进行垃圾回收，回收目标主要是针对常量池的回收和对类型的卸载

**线程私有区域**为：

1. **JVM栈**
2. **本地方法栈**

本地方法栈属于线程私有的数据区域，这部分主要与虚拟机用到的 Native 方法相关

**5、程序计数器PC**

属于线程私有的数据区域，是一小块内存空间，主要代**表当前线程所执行的字节码行号指示器**

 在jvm划分的内存区域中JVM栈和本地方法栈可能会抛出**StackOverflowError异常**。java堆和方法区可能会**抛出OutOfMemoryError异常**。程序计数器中没有地方规定会抛出这两个异常。

Java内存模型概述

JVM定义了一组规则，通过这组规则来决定一个线程对共享变量的写入何时对另一个线程可见，这组规则也称为Java内存模型（即JMM），JMM是围绕着程序执行的原子性、有序性、可见性展开的

总而言之，我们应该清楚知道，**JMM是一种抽象的概念，它定义了一组规则**，这组规则意在解决在**并发编程可能出现的线程安全问题**，并提供了**内置解决方案**（happen-before原则）及其**外部可使用的同步手段**(synchronized/volatile等)，确保了程序在**多线程环境中**执行时的原子性，可视性及其有序性。

算法：

1）一个数组中某个数出现次数大于一半，最快找出该数。1 2 2 2 4 3

2）找出数组中只出现一次的数字，其它数字都出现了两次

3）一个整型数组里除了两个数字之外，其他的数字都出现了两次（拆分）

4）给你1-1000个连续自然数，然后从中随机去掉两个，再打乱顺序，要求只遍历一次，求出被去掉的两个数

5）数组A中，除了某一个数字x之外，其他数字都出现了三次，而x出现了一次。请给出最快的方法，找到x。

6）一个数组存着负数与正数，将正数放在前面，负数放在后面

7）求两个有序数组的中位数

8）给定一个乱序数组，找到其中第K大的值，要求时间复杂度最低division

1. Java 的类加载器的种类都有哪些？  
   1、根类加载器(Bootstrap) --C++写的 ，看不到源码  
   2、扩展类加载器（Extension） --加载位置 ： **jre\lib\ext** 中

3、系统(应用)类加载器(System\App) --加载位置 ： classpath 中  
4、自定义加载器(必须继承 ClassLoader)

1. 类什么时候被初始化？  
   1）创建类的实例，也就是 new 一个对象  
   2）访问某个类或接口的静态变量，或者对该静态变量赋值  
   3）调用类的静态方法  
   4）反射（Class.forName("com.lyj.load")）  
   5）初始化一个类的子类（会首先初始化子类的父类）
2. 主要讲解类的加载体系：  
   java 程序中的 .java 文件编译完会生成 .class 文件，而 .class 文件就是通过被称为类加载器的 ClassLoader加载的，而 ClassLoder 在加载过程中会使用“双亲委派机制”来加载 .class 文件，先上图：

BootStrapClassLoader ： 启 动 类 加 载 器 ， 该 ClassLoader 是 jvm 在 启 动 时 创 建 的 ， 用 于 加载 $JAVA\_HOME$/jre/lib 下面的类库

ExtClassLoader：扩展类加载器， ExtClassLoader 会加载$JAVA\_HOME/jre/lib/ext 下的类

AppClassLoader：应用程序类加载器， AppClassLoader 会加载 java 环境变量  
CLASSPATH 所 指 定 的 路 径 下 的 类 库

**双亲委派机制是这样的** ：  
1）当 AppClassLoader 加载一个 class 时，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器 ExtClassLoader 去完成。  
2）当 ExtClassLoader 加载一个 class 时，它首先也不会自己去尝试加载这个类，而是把类加载请求委派给BootStrapClassLoader 去完成。  
3）如果 BootStrapClassLoader 加载失败（例如在$JAVA\_HOME$/jre/lib 里未查找到该 class），会使用ExtClassLoader 来尝试加载；  
4）若 ExtClassLoader 也加载失败，则会使用 AppClassLoader 来加载，如果 AppClassLoader 也加载失败，则会报出异常 ClassNotFoundException。

**双亲委派托机制为什么安全 ：**

**主要是防止内存中出现多份同样的字节码。**

从反向思考这个问题，如果没有双亲委派模型而是由各个类加载器自行加载的话，如果用户编写了一个java.lang.Object的同名类并放在ClassPath中，多个类加载器都去加载这个类到内存中，系统中将会出现多个不同的Object类，**那么类之间的比较结果及类的唯一性将无法保证**，而且如果不使用这种双亲委派模型将**会给虚拟机的安全带来隐患**。所以，要让类对象进行比较有意义，前提是他们要被同一个类加载器加载。

**例如类java.lang.Object，它由启动类加载器加载**。双亲委派模型保证任何类加载器收到的对java.lang.Object的加载请求，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此**Object类在程序的各种类加载器环境中都是同一个类**。

**相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己编写了一个称为java.lang.Object的类，并用自定义的类加载器加载，那系统中将会出现多个不同的Object类**，Java类型体系中最基础的行为也就无法保证，应用程序也将会变得一片混乱。

### **1、什么是类加载机制？**

　　JVM把class文件加载到**JVM 类加载机制详解**

内存里面，并对数据进行**校验、准备、解析和初始化**，最终能够被形成被JVM可以直接使用的Java类型的过程。

### ****3、加载****

将class文件加载在内存中。在堆中生成一个代表这个类的java.lang.Class对象，作为数据访问的入口。

### ****4、链接****

* 验证:确保加载的类符合JVM规范与安全。
* 准备:为static变量在方法区中分配空间，设置变量的初始值。例如static int a=3，在此阶段会a被初始化为0，其他数据类型参考成员变量声明。
* 解析:虚拟机将常量池的符号引用转变成直接引用。例如"aaa"为常量池的一个值，直接把"aaa"替换成存在于内存中的地址。

### ****5、初始化****

初始化阶段是类加载最后一个阶段，前面的类加载阶段之后，除了在加载阶段可以自定义类加载器以外，其它操作都由JVM主导。**到了初始阶段，才开始真正执行类中定义的Java程序代码**。

**注意以下几种情况不会执行类初始化：**

**1、通过子类引用父类的静态字段，只会触发父类的初始化，而不会触发子类的初始化。**

**3、通过类名获取Class对象，不会触发类的初始化。**

**4、通过Class.forName加载指定类时，如果指定参数initialize为false时，也不会触发类初始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。**

4、通过ClassLoader默认的loadClass方法，也不会触发初始化动作。

**子类继承父类时的初始化顺序**

    1.首先初始化父类的static变量和块，按出现顺序

   2.初始化子类的static变量和块，按出现顺序

   3.初始化父类的普通变量，调用父类的构造函数

   4.初始化子类的普通变量，调用子类的构造函数

# **防止Java内存泄露的一些小细节**

**有两个条件会触发主GC:**

**(1)当应用程序空闲时,即没有应用线程在运行时,GC会被调用。**因为GC在优先级最低的线程中进行,所以当应用忙时,GC线程就不会被调用,但以下条件除外。

**(2)Java堆内存不足时,GC会被调用。**当应用线程在运行,并在运行过程中创建新对象,若这时内存空间不足,JVM就会强制地调用GC线程,以便回收内存用于新的分配。若GC一次之后仍不能满足内存分配的要求,JVM会再进行两次GC作进一步的尝试,若仍无法满足要求,则 JVM将报“out of memory”的错误,Java应用将停止。

**(6)尽量少用静态对象变量**静态变量属于全局变量,不会被GC回收,它们会一直占用内存。

**(3)对象不用时最好显式置为Null**一般而言,为Null的对象都会被作为垃圾处理,所以将不用的对象显式地设为Null,**有利于GC收集器判定垃圾,从而提高了GC的效率。**

1. 尽早释放无用对象的引用。特别是大对象和集合对象，通过置为NULL，暗示垃圾收集器来收集该对象，防止发生内存泄露。
2. 没有及时地close掉io流

2）**程序中如果出现大量使用字符串处理，避免使用 String ，应使用 StringBuffer**。

1. **尽量少用静态变量，因为静态变量是全局的， GC 不会回收的**；
2. **代码中是否有死循环或递归调用**
3. **是否有大循环重复产生新对象实体**
4. **检查List、MAP等集合对象是否有使用完后，未清除的问题**。List、MAP等集合对象会始终存有对对象的引用，使得这些对象不能被GC回收。

**引起内存溢出的原因有很多种，常见的有以下几种**：

**1.内存中加载的数据量过于庞大，如一次从数据库取出过多数据；  
2.集合类中有对对象的引用，使用完后未清空，使得 JVM 不能回收；  
3.代码中存在死循环或循环产生过多重复的对象实体；  
4.启动参数内存值设定的过小；**

**内存溢出的解决方案：**  
第一步，修改 JVM 启动参数，直接增加内存。 (-Xms， -Xmx ，-Xss)

第二步，检查错误日志，查看“OutOfMemory”错误前是否有其它异常或错误。  
第三步，对代码进行优化，对代码进行走查和分析，找出可能发生内存溢出的位置。

**虚拟机不回收内存（内存泄漏）和内存溢出**

-Xms 为jvm启动时分配的内存，比如-Xms200m，表示分配200M

-Xmx 为jvm运行过程中分配的最大内存，比如-Xms500m，表示jvm进程最多只能够占用500M内存

-Xss 为jvm启动的每个线程分配的内存大小，默认JDK1.4中是256K，JDK1.5+中是1M

1. 谈谈你对 Spring 的理解  
   1） Spring 是一个 IOC 和 AOP 容器框架。  
   Spring 容器的主要核心是：  
   2）控制反转（IOC），传统的 java 开发模式中，当需要一个对象时，我们会自己使用 new 或者 getInstance 等直接或者间接调用构造方法创建一个对象。而在 **spring 开发模式中， spring 容器使用了工厂模式为我们创建了所需要的对象，不需要我们自己创建了，直接调用 spring 提供的对象就可以了**，这是控制反转的思想。  
   3）面向切面编程（AOP），在面向对象编程（oop）思想中，我们将事物纵向抽成一个个的对象。而在面向切面编程中，我们将一个个的对象**某些类似的方面**横向抽成一个切面，对这个切面进行一些如权限控制、事物管理，记录日志等公用操作处理的过程就是面向切面编程的思想。 AOP 底层是动态代理，如果是接口采用 JDK 动态代理，如果是类采用CGLIB 方式实现动态代理。 **便于减少系统的重复代码，降低模块之间的耦合度**
2. Spring 中的设计模式（2017-11-13-lyq）
3. 单例模式——在 spring 的配置文件中设置 bean 默认为单例模式
4. 工厂模式——在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用同一个接口来指向新创建的对象。 Spring 中使用 beanFactory 来创建对象的实例。
5. 代理模式
6. Spring 能帮我们做什么？
7. Spring 能帮我们根据配置文件创建及组装对象之间的依赖关系。  
   b. Spring 面向切面编程能帮助我们无耦合的实现日志记录，性能统计，安全控制。  
   c. Spring 能非常简单的帮我们管理数据库事务。  
   d. Spring 还能与第三方数据库访问框架（如 Hibernate、 JPA）无缝集成，而且自
8. 什么是 Spring IOC 容器？ （2017-12-03-gxb）  
   IOC 控制反转： Spring IOC 负责创建对象，管理对象。通过依赖注入（DI），装配对象，配置对象。
9. Spring 支持的几种 bean 的作用域。 （2017-12-04-gxb）  
   Spring 框架支持以下五种 bean 的作用域：  
   **singleton :** bean 在每个 Spring ioc 容器中只有一个实例。  
   **prototype：**一个 bean 的定义可以有多个实例。  
   **request**：每次 http 请求都会创建一个 bean，该作用域仅在基于 web 的 Spring ApplicationContext 情形下有效。  
   **session** ： 在 一 个 HTTP Session 中 ， 一 个 bean 定 义 对 应 一 个 实 例 。 该 作 用 域 仅 在 基 于 web 的Spring ApplicationContext 情形下有效。  
   **global-session：**在一个全局的 HTTP Session 中，一个 bean 定义对应一个实例。该作用域仅在基于 web 的Spring ApplicationContext 情形下有效。
10. Spring 框架中的单例 bean 是线程安全的吗?   
    Spring 框架中的单例 bean 不是线程安全的

4. SQL 之 sql 注入

1）Web表单中输入（恶意）SQL语句得到一个存在安全漏洞的网站上的数据库，而不是按照设计者意图去执行SQL语句

2） SQL注入是比较常见的网络攻击方式之一，它不是利用**[操作系统](http://lib.csdn.net/base/operatingsystem" \o "操作系统知识库" \t "https://blog.csdn.net/qq_37787456/article/details/_blank)**的BUG来实现攻击，而是针对程序员编程时的疏忽，通过SQL语句，实现无帐号登录，甚至篡改**[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "https://blog.csdn.net/qq_37787456/article/details/_blank)**。

3）****（简单又有效的方法）PreparedStatement**** 采用预编译语句集

****4）使用正则表达式过滤传入的参数****

5. Mysql 性能优化（2017-11-15-lyq）

**1、 当只要一行数据时使用 limit 1**查询时如果已知会得到一条数据，这种情况下加上 limit 1 会增加性能

**2、 选择正确的数据库引擎**Mysql 中有两个引擎 MyISAM 和 InnoDB，每个引擎有利有弊。

3. **用 not exists 代替 not in**Not exists 用到了连接能够发挥已经建立好的索引的作用， not in 不能使用索引

4. 对操作符的优化，尽量不采用不利于索引的**操作符**  
如： in not in is null is not null <> ！= not等

5、某个字段总要拿来搜索，为其建立索引：  
Mysql 中可以利用 alter table 语句来为表中的字段添加索引，语法为： alter table 表明 add index (字段名)；

脏读

不可重复读

（针对其他提交前后，读取数据本身的对比）

幻读

（针对其他提交前后，读取数据条数的对比） 幻读是指同样一笔查询在整个事务过程中多次执行后，查询所得的结果集是不一样的。幻读针对的是多笔记录。

|  |  |  |
| --- | --- | --- |
| 读未提交 是 | 是 | 是 |
| 读已提交 否 | 是 | 是 |
| 可重复读 否 | 否 | 是 |
| 可串行读 否 | 否 | 否 |

**3. synchronized 和 volatile 关键字的作用**  
一旦一个共享变量（类的成员变量、类的静态成员变量）被 volatile 修饰之后，那么就具备了两层语义：  
1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。  
2）禁止进行指令重排序。  
synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。  
**1.volatile 仅能使用在变量级别；**synchronized 则可以使用在变量、方法、和类级别的  
**2.volatile 仅能实现变量的修改可见性，并不能保证原子性；**synchronized 则可以保证变量的修改可见性和原子性  
**3.volatile 不会造成线程的阻塞**；

1、最基本的分层方式，结合了SSH[架构](http://lib.csdn.net/base/architecture" \o "大型网站架构知识库" \t "https://www.cnblogs.com/hoojjack/p/_blank)。**MVC**

**modle层：**就是对应的[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "https://www.cnblogs.com/hoojjack/p/_blank)表的实体类(如User类)。

**Dao层：**一般可以再分为\*\*\*Dao接口和\*\*\*DaoImpl实现类，如userDao接口和userDaoImpl实现类,接口负责定义数据库curd的操作方法，实现类负责具体的实现，即实现Dao接口定义的方法。

**Service层：**引用对应的Dao层数据库操作，在这里可以编写自己需要的代码（比如简单的判断），也可以再细分为Service接口和ServiceImpl实现类。

**Action层：**引用对应的Service层实现业务逻辑，在这里结合Struts的配置文件，跳转到指定的页面，当然也能接受页面传递的请求数据。

**2、大数据和高并发**成了软件设计必须考虑的问题，那么大数据和高并发本质是什么呢？

**本质很简单，一个是慢，一个是等。关键是如何解决慢和等，核心一个是短，一个是少，一个是分流。**

短是指路径要短。典型的mvc结构是请求->controller->model->dao->view，然后把页面返回给用户。要想短的话，

**1，页面静态化-** 用户可以直接获取页面，不用走那么多流程，比较适用于页面不频繁更新。

**2，使用缓存-** 第一次获取数据从数据库准提取，然后保存在缓存中，以后就可以直接从缓存提取数据。不过需要有机制维持缓存和数据库的一致性。

**3，使用储存过程-**那些处理一次请求需要多次访问数据库的操作，可以把操作整合到储存过程，这样只要一次数据库访问就可以了。

**4，批量读取** - 高并发情况下，可以把多个请求的查询合并到一次进行，以减少数据库的访问次数

**5，延迟修改** - 高并发情况下，可以把多次修改请求，先保存在缓存中，然后定时将缓存中的数据保存到数据库中，风险是可能会断电丢失缓存中的数据，

**6,  使用索引** - 索引可以看作是特殊的缓存，尽量使用索引就要求where字句中精确的给出索引列的值。

**少是指查询的数据要少。**

**1，分表 -** 把本来同一张表的内容，可以按照地区，类别等分成多张表，很简单的一个思路，但是要尽量避免分出来的多表关联查询。

**分流三种。**

1，集群 - **将并发请求分配到不同的服务器上**，可以是业务服务器，也可以是数据库服务器。

2，分布式 - **分布式是把单次请求的多项业务逻辑分配到多个服务器上**，这样可以同步处理很多逻辑，一般使用与特别复杂的业务请求。

3，CDN加速 - 在域名解析层面的分流，例如将华南地区的用户请求分配到华南的服务器，华中地区的用户请求分配到华中的服务器。

4说下ConcurrentHashMap 你知道些什么？怎么设计的？为什么不同的key会放在相同的位置？

（1）[ConcurrentHashMap实现原理及源码分析](http://www.cnblogs.com/chengxiao/p/6842045.html)

ConcurrentHashMap是Java并发包中提供的一个**线程安全且高效**的HashMap实现，ConcurrentHashMap在并发编程的场景中使用频率非常之高

（2）HashMap ：先说HashMap，HashMap是线程不安全的，所以，在并发环境中使用HashMap是非常危险的。

　　HashTable ：但是HashTable**线程安全的策略实现代价却太大了**，简单粗暴，get/put所有相关操作都是synchronized的，这相当于给整个哈希表加了一把大锁，多线程访问时候，**只要有一个线程访问或操作该对象，那其他线程只能阻塞，**相当于将所有的操作串行化，在竞争激烈的并发场景中性能就会非常差。

（3）ConcurrentHashMap采用了非常精妙的"分段锁"策略，ConcurrentHashMap的主干是个Segment数组。

final Segment<K,V>[] segments;

　　Segment继承了ReentrantLock，所以它就是一种可重入锁（ReentrantLock)。在ConcurrentHashMap，一个Segment就是一个子哈希表，Segment里维护了一个HashEntry数组，并发环境下，对于不同Segment的数据进行操作是不用考虑锁竞争的。（就按默认的ConcurrentLeve为16来讲，理论上就允许16个线程并发执行，有木有很酷）

**所以，对于同一个Segment的操作才需考虑线程同步，不同的Segment则无需考虑。**

Segment类似于HashMap，一个Segment维护着一个HashEntry数组

transient volatile HashEntry<K,V>[] table;

HashEntry是目前我们提到的最小的逻辑处理单元了。一个ConcurrentHashMap维护一个Segment数组，一个Segment维护一个HashEntry数组。

（4）ConcurrentHashMap作为一种线程安全且高效的哈希表的解决方案，尤其其中的"分段锁"的方案，相比HashTable的全表锁在性能上的提升非常之大

5volatile有什么用？可见性怎么实现的？

**1）能够保证变量修改的可见性 2）禁止指令重排序优化**

6讲讲知道的数据结构（ArrayList,LinkinedList,HashMap,HashSet,Stack）

**HashMap和Hashtable的区别**

1. 两者最主要的区别在于Hashtable是线程安全，而HashMap则非线程安全  
   Collections.synchronizedMap()和[Collections.synchronizedList](http://www.baidu.com/link?url=X42LVvwfTxN3bRDkPxOW1gcNu9KCCDEgx0nGhClQJw0rReIwKVsEBlfRe6xJcQmzng21y3_72HJJBZ_EeFMW__" \t "https://www.baidu.com/_blank)
2. 方法来获取一个线程安全的集合

ConcurrentHashMap实现高效线程安全

2）HashMap可以使用null作为key，而Hashtable则不允许null作为key

3）HashMap的初始容量为16，Hashtable初始容量为11，两者的填充因子默认都是0.75  
4）HashMap扩容时是当前容量翻倍即:**capacity\*2**，Hashtable扩容时是容量翻倍+1即:**capacity\*2+1**

5）两者计算hash的方法不同  
Hashtable计算hash是直接使用key的hashcode对table数组的长度直接进行取模

HashMap计算hash对key的hashcode进行了二次hash，以获得更好的散列值，然后对table数组长度取摸

6）HashMap和Hashtable的底层实现都是**数组+链表**结构实现

HashMap：

Put：

首先对key做null检查。如果key是null，会被存储到table[0]，因为null的hash值总是0。

key的hashcode()方法会被调用，然后计算hash值。hash值用来找到存储Entry对象的数组的索引。有时候hash函数可能写的很不好，所以JDK的设计者添加了另一个叫做hash()的方法，它接收刚才计算的hash值作为参数。如果你想了解更多关于hash()函数的东西，可以参考：hashmap中的hash和indexFor方法

indexFor(hash,table.length)用来计算在table数组中存储Entry对象的精确的索引。

在我们的例子中已经看到，如果两个key有相同的hash值(也叫冲突)，他们会以链表的形式来存储。所以，这里我们就迭代链表。

· 如果在刚才计算出来的索引位置没有元素，直接把Entry对象放在那个索引上。

· 如果索引上有元素，然后会进行迭代，一直到Entry->next是null。当前的Entry对象变成链表的下一个节点。

· **如果我们再次放入同样的key会怎样呢？逻辑上，它应该替换老的value**。事实上，它确实是这么做的。在迭代的过程中，会调用equals()方法来检查key的相等性(key.equals(k))，如果这个方法返回true，它就会用当前Entry的value来替换之前的value。

Get：

当你传递一个key从hashmap总获取value的时候：

对key进行null检查。如果key是null，table[0]这个位置的元素将被返回。

key的hashcode()方法被调用，然后计算hash值。

indexFor(hash,table.length)用来计算要获取的Entry对象在table数组中的精确的位置，使用刚才计算的hash值。

在获取了table数组的索引之后，**会迭代链表，调用equals()方法检查key的相等性**，如果equals()方法返回true，**get方法返回Entry对象的value，否则，返回null。**

HashSet：

除开HashMap和Hashtable外，还有一个hash集合HashSet，有所区别的是HashSet不是key value结构，仅仅是存储不重复的元素，相当于简化版的HashMap，只是包含HashMap中的key而已

通过查看源码也证实了这一点，HashSet内部就是使用HashMap实现，只不过**HashSet里面的HashMap所有的value都是同一个Object而已，因此HashSet也是非线程安全的**

ArrayList：

**List->Array ：**

List<String> list=new ArrayList<String>();

    list.add("王利虎");

    list.add("张三");

    list.add("李四");

    int size=list.size();

String[] array = (String[])list.toArray(**new String[size]**); **Array->List:**

String[] array=new String[3];

 array[0]="王利虎";

 array[1]="张三";

 array[2]="李四";

 List<String> list=**Arrays.asList(array);**

LinkinedList：

1)LinkedList 是一个继承于AbstractSequentialList的双向链表。它也可以**被当作堆栈、队列或双端队列进行操作。**

2)LinkedList **实现 List 接口**，能进行队列操作。

3)LinkedList **实现 Deque 接口**，即能将LinkedList当作**双端队列**使用。

* **ArrayList** 就是动态数组，是Array的复杂版本，动态的增加和减少元素.当更多的元素加入到ArrayList中时,其大小将会动态地增长。它的元素可以通过get/set方法直接访问，因为ArrayList本质上是一个数组。
* **Vector** 和ArrayList类似, 区别在于**Vector是同步类(synchronized).因**此,开销就比ArrayList要大。
* **LinkedList** 是一个双链表,在添加和删除元素时具有比ArrayList更好的性能.但在get与set方面弱于ArrayList.当然,这些对比都是指数据量很大或者操作很频繁的情况下的对比。**它还实现了 Queue 接口**,该接口比List提供了更多的方法,包括 offer(),peek(),poll()等.

1. 数据库索引，
2. 索引是数据库管理系统中的一种排序的数据结构，底层是由B树和B+树实现的，可以加快数据检索速度，但是降低了数据增加和删除的速度；
3. 在索引字段列中不可进行计算操作、不可使用函数、不可进行数据类型转换操作；不能使用null，not null；不能使用not、！=和<>等操作符

## **B树：（M=3）**

B-树的搜索，从根结点开始，对结点内的关键字**（有序）序列进行二分查找**，如果

命中则结束，否则进入查询关键字所属范围的儿子结点；重复，直到所对应的儿子指针为

空，或已经是叶子结点；

**B+树是B-树的变体，也是一种多路平衡查找树**

5.**为所有叶子结点增加一个链指针**；

6.**所有关键字都在叶子结点出现**；

# **B+的特性：**

1.所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好

是有序的；

2.不可能在非叶子结点命中；

3.非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储

B+的搜索与B-树也基本相同，区别是B+树只有达到叶子结点才命中（B-树可以在非叶子结点命中），其性能也等价于在关键字全集做一次二分查找；

**B+树：**在B-树基础上，为叶子结点增加链表指针，所有关键字都在叶子结点

中出现，非叶子结点作为叶子结点的索引；B+树总是到叶子结点才命中；

1. **jvm内存区域划分，**

JVM 将内存区域划分：

0）Method Area（Non-Heap）（方法区） ——线程共享

1）Heap（堆） ——线程共享

2）Program Counter Register（程序计数器） ——非线程共享

3）VM Stack（虚拟机栈，也有翻译成JAVA 方法栈的）——非线程共享

4）Native Method Stack （ 本地方法栈 ）——非线程共享

JVM**运行的时候会分配好** Method Area（方法区） 和Heap（堆），

**而JVM 每遇到一个线程，就为其分配一个 Program Counter Register（程序计数器） , VM Stack（虚拟机栈）和Native Method Stack （本地方法栈）**， **当线程终止时，三者（虚拟机栈，本地方法栈和程序计数器）所占用的内存空间也会被释放掉**。   
**非线程共享的那三个区域的生命周期与所属线程相同，而线程共享的区域与JAVA程序运行生命周期相同，**   
**所以gc只发生在线程共享的区域（大部分发生在Heap上）的原因**。

1. **数据库引擎**

MyISAM与InnoDB的区别  
　　InnoDB和MyISAM是许多人在使用MySQL时最常用的两个表类型，这两个表类型各有优劣，视具体应用而定。基本的差别为：MyISAM类型不支持事务处理等高级处理，而InnoDB类型支持。

1）**MyISAM类型的表强调的是性能，其执行数度比InnoDB类型更快**，但是不提供事务支持，**而InnoDB提供事务支持已经外部键、支持行锁等高级数据库功能**。

2）因为MyISAM相对简单所以在效率上要优于InnoDB.如果系统读多，写少，对原子性要求低。那么MyISAM最好的选择。  
3）**如果系统读少，写多的时候，尤其是并发写入高的时候。InnoDB就是首选了。**

# **[Java并发之AQS详解](http://www.cnblogs.com/waterystone/p/4920797.html)6**

# **一、概述**

　　谈到并发，不得不谈ReentrantLock；而谈到ReentrantLock，不得不谈AbstractQueuedSynchronizer（AQS）！

类如其名，**抽象的队列式的同步器**，**AQS定义了一套多线程访问共享资源的同步器框架**，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch

基本实现原理：

**AQS使用一个int成员变量来表示同步状态，通过内置的FIFO队列来完成获取资源线程的排队工作。**

**状态信息**通过procted类型的**getState**，**setState**，**compareAndSetState**进行操作

AQS支持两种同步方式：

**1.独占式2.共享式**

　　这样方便使用者实现不同类型的同步组件，独占式如ReentrantLock，共享式如Semaphore

**如何使用**

　　首先，我们需要去继承AbstractQueuedSynchronizer这个类，然后我们根据我们的需求去重写相应的方法，比如要实现一个独占锁，那就去重写tryAcquire，tryRelease方法，要实现共享锁，就去重写tryAcquireShared，tryReleaseShared；最后，在我们的组件中调用AQS中的模板方法就可以了，而这些模板方法是会调用到我们之前重写的那些方法的。

ReentrantLock的实现原理，它是如何保证可重入性，又是如何实现公平锁的？？？

ReentrantLock是基于AQS的，AQS是Java并发包中众多同步组件的构建基础，它通过一个int类型的状态变量state和一个FIFO队列来完成共享资源的获取，线程的排队等待等。**AQS是个底层框架，采用模板方法模式，它定义了通用的较为复杂的逻辑骨架，比如线程的排队，阻塞，唤醒等**，将这些复杂但实质通用的部分抽取出来，这些都是需要构建同步组件的使用者无需关心的，使用者仅需重写一些简单的指定的方法即可（其实就是对于共享变量state的一些简单的获取释放的操作）。

## **二、ReentrantLock**

可重入锁，和同步锁功能类似，不过需要显示的创建、销毁。特点：

1.ReentrantLock有tryLock方法，如果锁被其他线程持有，返回false,可避免形成死锁。

2.创建时可自定义是否可抢占。

2、**同步执行，类似synchronized（也是使用最多的）**

ReentrantLock lock = new ReentrantLock(); //参数默认false，**不公平锁**：可抢占

ReentrantLock lock = new ReentrantLock(true); //**公平锁**：严格按照请求锁的排队顺序获取锁

3、**尝试等待固定时间再次获取**

ReentrantLock lock = new ReentrantLock(true); //公平锁

try {

if (lock.tryLock(5, TimeUnit.SECONDS)) {

//如果已经被lock，尝试等待5s，看是否可以获得锁，如果5s后仍然无法获得锁则返回false

4、**可中断锁的同步执行**

ReentrantLock lock = new ReentrantLock(true); //公平锁

lock.lockInterruptibly();

### ReentrantLock和synchronized关键字的区别

1）ReentrantLock在**等待锁时可以使用lockInterruptibly()方法选择中断**， 改为处理其他事情，**而synchronized关键字，线程需要一直等待下去**。同样的，**tryLock()方法可以设置超时时间**，用于在超时时间内一直获取不到锁时进行中断。

2）**ReentrantLock可以实现公平锁，而synchronized的锁是非公平的。**

3）**ReentrantLock可以同时绑定多个Condition对象**，而**synchronized中，锁对象的wait()和notify()或notifyAll()方法可以实现一个隐含的条件**，如果要和多于一个条件关联时，只能再加一个额外的锁，而ReentrantLock只需要多次调用newCondition方法即可

1、[CountDownLatch](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CountDownLatch.html" \o "CountDownLatch" \t "http://www.importnew.com/_blank)是一个同步工具类，**它允许一个或多个线程一直等待，直到其他线程的操作执行完后再执行。**CountDownLatch类位于java.util.concurrent包下

CountDownLatch类只提供了一个构造器：

|  |  |
| --- | --- |
| 1 | public CountDownLatch(int count) {  };  //参数count为计数值 |

 　　然后下面这3个方法是CountDownLatch类中最重要的方法：

|  |  |
| --- | --- |
| 1  2  3 | public void **await() t**hrows InterruptedException { };   //调用await()方法的线程会被挂起，它会等待直到count值为0才继续执行  public void **countDown()** { };  //将count值减1 |

CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行；

1. Semaphore翻译成字面意思为 信号量，**Semaphore可以控制同时访问的线程个数**，通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可。Semaphore类位于java.util.concurrent包下，它提供了2个构造器：

|  |
| --- |
| public Semaphore(int permits) {          //参数permits表示许可数目，即同时可以允许多少线程进行访问      sync = new NonfairSync(permits);  }  public Semaphore(int permits, boolean fair) {    //这个多了一个参数fair表示是否是公平的，即等待时间越久的越先获取许可      sync = (fair)? new FairSync(permits) : new NonfairSync(permits);  } |

 　　下面说一下Semaphore类中比较重要的几个方法，首先是acquire()、release()方法：

|  |  |
| --- | --- |
| 1  2  3  4 | public void acquire() throws InterruptedException {  }     //获取一个许可  public void acquire(int permits) throws InterruptedException { }    //获取permits个许可  public void release() { }          //释放一个许可  public void release(int permits) { }    //释放permits个许可 |

acquire()用来获取一个许可，若无许可能够获得，则会一直等待，直到获得许可。

release()用来释放许可。注意，在释放许可之前，必须先获获得许可。

**Semaphore在限制资源访问量的问题上用处很大，比如限制一个文件的并发访问次数等**

1. **notify()方法**不能唤醒某个具体的线程，所以只有一个线程在等待的时候它才有用武之地。而notifyAll()唤醒所有线程并允许他们争夺锁确保了至少有一个线程能继续运行。
2. **线程安全** 就是多线程并发访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。**不会出现数据不一致或者出现脏读等**
3. 有三个线程T1，T2，T3，怎么确保它们按顺序执行？   
   在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。

## **JAVA中独占锁（悲观锁）与乐观锁**

在多线程编程的时候，为了保证多个线程对一个对象可以安全进行访问时,我们需要加同步锁synchronized,保证对象的在使用时的正确性，synchronized就是一种独占锁，它会导致所有需要此锁的线程挂起，等待锁的释放。

**乐观锁（CAS）**的思想却是不加锁，那不加锁如何确保某一变量的操作没有被其他线程修改过？

这里就需要CAS操作（CompareAndSwap）来实现。

CAS的原理 **ABA**

CAS的英文是Compare and Swap,中文意思是比较并交换。JUC包中借助CAS实现了区别于synchronized同步锁的一种乐观锁。CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。重点来了，CAS的原理:CAS通过调用JNI的代码实现的。JNI:Java Native Interface为JAVA本地调用，允许Java调用其他语言(C)。很多实现原子性操作的调用，都是依赖于sun.misc.Unsafe类。比如:compareAndSwapInt()方法就是借助C来调用CPU底层指令实现的。

AQS的原理

**AQS（AbstractQueuedSynchronizer）是JDK1.5提供的一个基于volatile int state（代表共享资源）和一个FIFO线程等待队列来实现同步器的基础框架**，这个基础框架的重要性可以这么说，JUC包里面几乎所有的同步工具类都是基于AQS这个框架。**AQS的核心思想是**如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制。

**CAS（CompareAndSet）是最小粒度的操作，保证了原子性，通过硬件指令集实现。因为AQS也是基于CAS实现的。**

**模板方法模式:定义一个算法的骨架，将骨架中的特定步骤延迟到子类中。模板方法模式使得子类可以不改变算法的结构即可重新定义该算法的某些特定步骤（int、double、float型的冒泡排序）**

**策略模式：定义了一组算法、对这些算法进行封装、算法之间可以相互替换、冒泡排序、快速排序、堆排序**

# **java.util.concurrent - Java并发工具包**

**0）ReentraintLock 可重入锁**

**1）ConcurrentHashMap 线程安全和高效**

**2）闭锁 CountDownLatch join（）**

**4）信号量 Semaphore 生产者和消费者问题**

**5）执行器服务 ExecutorService 线程池**

**6） AtomicInteger，一个提供原子操作的Integer的类。**

**在Java语言中，++i和i++操作并不是线程安全的**，在使用的时候，

不可避免的会用到synchronized关键字。而AtomicInteger则通过一种线程安全的加减操作接口。AtomicInteger为什么能够达到多而不乱，处理高并发应付自如呢？这里面用到了一种**并发技术：CAS**（由硬件提供原子操作指令实现的）。

Java.util.concurrent中实现的原子操作类包括：

AtomicBoolean、AtomicInteger、AtomicIntegerArray、AtomicLong、AtomicReference、AtomicReferenceArray。

**并发编程**

**一、[谈谈Java中的ThreadLocal](http://www.cnblogs.com/chengxiao/p/6152824.html)**

1）ThreadLocal一般称为**线程本地变量，**它是一种特殊的线程绑定机制，将变量与线程**绑定**在一起，为每一个线程维护一个独立的变量副本。

2）因为ThreadLocal在每个线程中对该变量会创建一个副本，即每个线程内部都会有一个该变量，且在线程内部任何地方都可以使用**，线程之间互不影响，这样一来就不存在线程安全问题，**也不会严重影响程序执行性能。

但是要注意，虽然ThreadLocal能够解决上面说的问题，但是由于在每个线程中都创建了副本，所以要考虑它对资源的消耗，比如内存的占用会比不使用ThreadLocal要大。

3）总结一下：

1、实际的通过ThreadLocal创建的副本是存储在每个线程自己的threadLocals中的；

2、为何threadLocals的类型ThreadLocalMap的键值为ThreadLocal对象，因为每个线程中可有多个threadLocal变量，就像上面代码中的longLocal和stringLocal；

3、在进行get之前，必须先set，否则会报空指针异常；

ThreadLocal的应用场景

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

1. **[谈谈Java中的volatile](http://www.cnblogs.com/chengxiao/p/6528109.html)**

1）volatile是Java提供的一**种轻量级的同步机制**，在并发编程中，它也扮演着比较重要的角色。同synchronized相比（synchronized通常称为重量级锁），volatile更轻量级，相比使用synchronized所带来的庞大开销，倘若能恰当的合理的使用volatile，自然是美事一桩。

2）volatile是一种轻量级的同步机制，它主要有两个特性：一是保证共享变量对所有线程的可见性；**二是禁止指令重排序优化**

3）但是需要注意的是，我们一直在拿volatile和synchronized做对比，仅仅是因为这两个关键字在某些内存语义上有共通之处，volatile并不能完全替代synchronized，它依然是个轻量级锁，在很多场景下，volatile并不能胜任，比如num++非原子操作；**所以我们之前提到过的volatile是无法解决num++的原子性问题的**

**[CopyOnWriteArrayList实现原理及源码分析](http://www.cnblogs.com/chengxiao/p/6881974.html)**

1. **CopyOnWriteArrayList**是Java并发包中提供的一个并发容器，它是个**线程安全且读操作无锁的ArrayList**，写操作则通过创建底层数组的新副本来实现，是一种**读写分离**的并发策略，我们也可以称这种容器为"**写时复制器**"，Java并发包中类似的容器还有CopyOnWriteSet。
2. 我们都知道，集合框架中的ArrayList是非线程安全的，Vector虽是线程安全的，但由于简单粗暴的锁同步机制，性能较差。而CopyOnWriteArrayList则提供了另一种不同的并发处理策略（当然是针对特定的并发场景）
3. CopyOnWriteArrayList容器**允许并发读，读操作是无锁的，性能较高**。至于写操作，比如向容器中添加一个元素，**则首先将当前容器复制一份，然后在新副本上执行写操作，结束之后再将原容器的引用指向新容器。**

**优点：**读操作性能很高，因为无需任何同步措施，比较适用于读多写少的并发场景。

**缺点：**缺点也很明显，**一是内存占用问题**，毕竟每次执行写操作都要将原容器拷贝一份，数据量大时，对内存压力较大，可能会引起频繁GC；**二是无法保证实时性**，Vector对于读写操作均加锁同步，可以保证读和写的强一致性。

**[Java线程间通信之wait/notify](http://www.cnblogs.com/chengxiao/p/7043529.html)**

**1）Java中的wait/notify/notifyAll可用来实现线程间通信，是Object类的方法，这三个方法都是native方法，是平台相关的，常用来实现生产者/消费者模式，final修饰，不能被重写。**

1. 全程在问我有没有关注java新技术等，

1）lambda表达式本质是匿名方法，下面是一些lambda表达式：

(int x, int y) -> x + y

() -> 42

(String s) -> { System.out.println(s); }

lambda表达式的语法由**参数列表、箭头符号->和函数体组成**。函数体既可以是一个表达式，也可以是一个语句块：

players.forEach((player)->System.out.print(player+";"));

**Lambda在多核处理器条件下对集合遍历时的性能提高极大**

**用极少的代码完成了之前一个类做的事情！**

**1.替代匿名内部类**

而如果使用lambda表达式：

@Test

public void runable() {

new Thread(() -> System.out.println("It's a lambda function!")).start();

}  
11、Object都有哪些方法？

**9种，hashcode，notify，notifyAll、wait、equals、finalize、toString、getClass、clone**

设计模式：

1. 单例模式、工厂模式、策略模式、代理模式

Spring中设计模式：

**1、工厂模式，**在各种BeanFactory以及ApplicationContext创建中都用到了

**2、模版模式，**在各种BeanFactory以及ApplicationContext实现中也都用到了如：**JDBCTemplate**

**3、代理模式，**Spring AOP 利用了 AspectJ AOP实现的! AspectJ AOP 的底层用了动态代理

目标方法有接口时候自动选用 JDK 动态代理 目标方法没有接口时候选择 CGLib 动态代理

**4、策略模式，加载资源文件的方式，使用了不同的方法，比如：ClassPathResourece，FileSystemResource，ServletContextResource，UrlResource但他们都有共同的借口Resource；在Aop的实现中，采用了两种不同的方式，JDK动态代理和CGLIB代理**

**5、单例模式，**比如在创建bean的时候。

**模板设计模式**：

**定义一个操作中算法的骨架，而将一些步骤延迟到子类中**，模板方法使得子类可以不改变算法的结构即可重定义该算法的某些特定步骤。

一面：

1. 自我介绍一下，然后问你最擅长哪一方面(我说虚拟机、集合)  
   2.他说HashMap我现在都不敢问了，出过很多笑话，很多人都是背的，自己没办法判断是看的还是背的

**数组+链表，put和get方法 indexFor**  
3.简单说下HashMap的get方法实现

**判断key==null 2）hashcode（）和equals（）**  
4.HashMap数组的第一个元素存的是什么

**Key==null的entry对象**

1. HashMap和Hashtable的区别

**(线程安全不安全，能否存null，数组初始化容量，扩容不一样，hashcode方法)**  
7.Hashtable在null的处理是怎样的

**不能存null的entry对象**  
8.线程的几种状态

**五中状态（新建、就绪、运行、堵塞、死亡）**

1. 这些状态之间怎么转换的，哪些方法转换

**Sleep、join、yeild、wait、notify、notifyAll**  
10.一个线程连着调用start两次会出现什么情况(这个讨论了好久好久，他说给你设计这个start你怎么处理这种情况，直接懵逼...提示结合那个线程状态机制想下)

**throw new IllegalThreadStateException()**  
11.多线程这块对哪些熟悉

**(synchronized、volatile、ReentrantLock、ConcurrentHashMap、CopyOnWriteArrayList、CountDownLatch、AutomicInteger、Semophore、ExecutorService)**12.synchronized和lock的区别

**Lock（类、在finally释放锁、可重入、可中断、可使用公平锁，lock、unlock、tryLock、lockInterrupt）其中，ReentrainLock是lock的一个实现类；**

**Synchronized（关键字、可重入、不可中断等待、非公平锁）**  
13.由于我说了一般unlock放在finally里面，他就问try后面一定要跟catch吗，我说可以不跟finally但是一定要跟catch，他说try后面不能跟catch是14.lock看过多少  
15.Java里面用过哪些包

1. **我说lang util juc math io.他说util除了用过集合还用过哪些，我说Arrays（sort、asList）、Collections（sort、SynchronizedMap、SynchronizedList）**16.String是可变的吗，答final数组不可变，问是final的什么数组，答char

**内容不可变**17.String s="abc"和String s=new String("abc")区别，然后撤常量池还扯String那个intern()方法

**intern()方法设计的初衷，就是重用String对象，以节省内存消耗**  
18.StringBuffer和StringBuilder区别

**线程安全（String和StringBuffer是线程安全的）和非安全，以及速度上面**  
20.Java的引用类型你知道吗

**接口、类、数组**21.那强引用弱引用这些知道吧？

java对象的引用包括  
强引用， new强引用引用的对象永远不会被垃圾回收，JVM宁愿抛出OutOfMemory错误也不会回收这种对象。

软引用，如果一个对象具有软引用，内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存

弱引用，当JVM进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。

虚引用，在任何时候都可能被垃圾回收器回收。

Java中提供这四种引用类型主要有两个目的：

第一是可以让程序员通过代码的方式决定某些对象的生命周期；

第二是有利于JVM进行垃圾回收。

1. 然后问内存管理，内存管理记得清楚吧

**JVM内存分配、JVM垃圾回收**

1. Minor GC和Full GC，又扯了好久，反正基本把第二章第三章里面的内容给问了个遍还问的特别细...

**Minor gc发生在新生代，复制算法**

**Full gc发生在老年代，标记整理算法**25.JVM里面的参数知道哪些，默认大小是多少，我说只知道

**Xmx Xms Xss**  
27.CMS为什么会停顿，停顿时间(我说如果不停顿那个GC Roots没办法进行下去，必须停一下等着GC Roots遍历完成，就和你妈给你打扫卫生时，你肯定不能还扔垃圾，肯定要停下来一样)

28.类加载了解吗？

1. **什么是类加载；2）类加载5个步骤；3）双亲外派机制**
2. **类加载：将.calss的文件加载到方法区中去，并在堆中创建一个java.lang.class的对象，用来指定方法区中的类所用到的数据结构**
3. **类的加载、连接（验证、准备、解析）、初始化**

29.数据结构掌握程度怎么样？(再次瑟瑟发抖，面试官说链表我能问吗？问怎么判断两个单链表是否相交...就这题他提示我好久，最后还是没想出来，就是这么的渣...他说直接判断最后一个节点是否相交就好，我说是不是类似一个竹子从中间砍了一点，然后前面分叉，后面相交这样，他说对对对，是这样的)  
30.数据库熟悉吗？

**1）数据库优化；2）数据库索引；3）数据库事务；4）数据库引擎；5）数据库视图和触发器**

---------------------------------------------------------------------

1. 起手式：Object类有哪些方法？

**Wait、notify、notifyAll、hashCode、equals、toString、getClass、finalize、clone**

2. wait方法能不能被重写，wait能不能被中断

**wait()、notify()和notifyAll()方法是本地方法,并且为final方法,无法被重写**

3. 里面哪些方法能被重写

4. hashcode方法有什么用

**1）HashCode的存在主要是为了查找的快捷，HashCode是用来在散列存储结构中确定对象的存储地址的**

**2）如果两个对象equals相等，那么这两个对象的HashCode一定也相同**

**3）如果对象的equals方法被重写，那么对象的HashCode方法也尽量重写**

**4）如果两个对象的HashCode相同，不代表两个对象就相同，只能说明这两个对象在散列存储结构中，存放于同一个位置**

5. HashSet方法里面的hashcode存在哪(我说类似HashMap存在Node里面，他还是问了我好久，没看过源码很虚)

Entry节点中，hashcode、Key、Value、Next

8. 哪些方法能使线程从运行态变为阻塞态

**Sleep、join、wait**

1. 说说线程池，线程池中一个线程执行完任务后处于那种状态

**ExecutorService实现线程池，线程池概念、作用**

**线程池作用：减少线程创建和销毁的次数，对线程进行统一的管理、提高线程的使用效率**

**threadPoolExecutor：参数corePoolSize，共6个参数**

**四个线程池|newFixThreadPool、newCacheThreadPool、newSingleThreadExecutor、newScheduleThreadPool**

**Execute（）和submit（）、shutdown（）、shutdownNow（）**

10. Thread类的interrupt,interrupted,isInterrupted方法的区别

11. rt.jar被什么类加载器加载，什么时间加载

12. 自己写的类被什么加载，什么时间加载

13. 自己写的两个不同的类是被同一个类加载器加载的吗？为什么？

14. CPU能不能直接和二级缓存打交道，接下来一顿问...

15. 三次握手

16. 网络编程会吗

17. socket中accept什么用

18. 两个不同ip地址的计算机之间如何通信

19. 项目中用过Spring吗

20. Spring优点哪些，源码看过吗

22. 数据库连接池用过吗，来说一说

23. 项目中的MySQL、Nginx这些都是自己配置的吗

24. 有什么想问我的？

二面 （部门主管）：  
1、说一下Spring的IOC和AOP，底层什么原理

1. 动态代理有几种，Jdk与Cglib区别

3、数据库三大范式

第一范式（所有字段是单一属性，不可再分的）

第二范式（在第一范式基础上，**所有非主属性都完全依赖于主键**）

没有包含在主键中的列必须完全依赖于主键，而不能只依赖于主键的一部分。

第三范式（）

即不能存在：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况。

举例来说：Employee(emp\_id,emp\_name,emp\_age,dept\_id,dept\_name,dept\_info)，当员工表中emp\_id能够唯一确定员工员工信息，但是dept\_name可由dept\_id唯一确定，此时，该表不符合第三范式，此时可以删除除了dept\_id之外的其他部门信息，把所有部门信息单独建立一张部门表。

4、左连接和右连接说一下，内连接呢

**左联接：是以左表为基准，将a.stuid = b.stuid的数据进行连接，然后将左表没有的对应项显示，右表的列为NULL**

**右连接：是以右表为基准，将a.stuid = b.stuid的数据进行连接，然以将右表没有的对应项显示，左表的列为NULL**

**完整外部联接返回左表和右表中的所有行。当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值。如果表之间有匹配行，则整个结果集行包含基表的数据值**5、数据库索引有几种

普通索引、唯一索引、主键索引、组合索引、全文索引

1. 数据库引擎你认识几种，innodb 和myisam 区别，你的项目用到哪个引擎

Innodb：使用的是聚簇索引，支持事务、支持外键、支持行锁、不支持全文检索，适合读少写多，并发写入高的场景

MyIsam：使用了非聚簇索引，不支持事务、不支持外键、不支持行锁，支持全文索引，适合读多写少，原子性要求不高的场景

Engine = innodb  
7、若hashcode方法永远返回1会产生什么结果

Hashcode的作用，主要是方便比较，加快元素之间的比较，比如hashSet中保证元素不重复，就是用到了hashcode和equal配合使用完成的  
8、Error与RuntimeException的区别

Error:只系统内部出现的错误，比如内存溢出，栈溢出、系统崩溃等待，是应用程序无法捕获并处理的错误；

RunntimeExeception：比如空指针异常、数组越界、类型转换异常、类没有找到异常、方法参数异常等异常，在编译的时候，能够正常通过，运行的时候，程序可以捕获和处理

非RunntimeExeception：比如IO异常、Sql异常，编译的时候无法通过，需要使用tyr catch异常捕获机制处理异常

9、引用计数法与GC Root可达性分析法区别  
10、双亲委派机制说一下

1. 算法题：找出一个数组中第100个小的数字（堆思想解决）

**最小堆和快速排序**

先取出前100个数，维护一个100个数的最小堆，遍历一遍剩余的元素，在此过程中维护堆就可以了。具体步骤如下：   
step1：取前m个元素（例如m=100），建立一个小顶堆。保持一个小顶堆得性质的步骤，运行时间为O（lgm);建立一个小顶堆运行时间为m\*O（lgm）=O(m lgm);         
step2:顺序读取后续元素，直到结束。每次读取一个元素，如果该元素比堆顶元素小，直接丢弃   
如果大于堆顶元素，则用该元素替换堆顶元素，然后保持最小堆性质。最坏情况是每次都需要替换掉堆顶的最小元素，因此需要维护堆的代价为(N-m)\*O(lgm);   
最后这个堆中的元素就是前最大的10W个。时间复杂度为O(N lgm）

三面&四面 （这里合在一起写了，分别是部门A总监与B总监，本以为三面完了就是HR面，结果隔天B总监打过来说补一轮技术面~~三面和四面更多的是问项目，基础问得不多）：

1. volatile关键字作用

2、看你项目用到策略模式和工厂模式，说一下区别

3、模板方法模式

4、开闭原则懂吗，说一下

5、NIO说一下你的理解

**IO的方式通常分为几种，同步阻塞的BIO、同步非阻塞的NIO、异步非阻塞的AIO。**

1. AtomicInteger底层原理

首先使用Volatile关键字修饰变量，保证变量的可见性；

然后，在使用getAndIncreament或者getAndDecreament方法的时候，使用CAS机制，保证变量增加或者减少的原子性操作！！

7、CAS机制会出现什么问题

ABA问题，可以使用version版本号解决问题

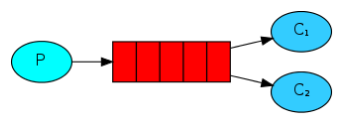
8、还用过并发包哪些类  
copyOnWriteArrayList、copyOnWriteArraySet、concurrentHashMap、concurrentSkipListMap、concurrentSkipListSet、countDownLatch、Semaphore、CyclicBarrier、  
12、RabbitMQ消息队列丢失消息，重复消费问题

**重复消费问题：**

描述：当消费者消费了一条消息后，会发送ACK状态码给MQ，可能由于网络等原因，ACK状态码丢失了，从而导致消息一直存留在MQ中，当新的消费者启动时会重新消费；   
解决方案：消费者每次消费了一条消息后，可以将消费记录存在redis的list中去，当有新的消费者消费的时候，先查询redis，若已经消费，则直接向MQ发送一个ACK，将消息从MQ中移除；否则，去MQ中取出消息进行消费

RabbitMQ中，消息丢失可以简单的分为两种：客户端丢失和服务端丢失。针对这两种消息丢失，RabbitMQ都给出了相应的解决方案。

**防止客户端丢失消息**



生产者P向队列中生产消息，C1和C2消费队列中的消息，默认情况下，RabbitMQ会平均的分发消费给C1C2（Round-robin dispatching），假设一个任务的执行时间非常长，在执行过程中，客户端挂了（连接断开），那么，该客户端正在处理且未完成的消息，以及分配给它还没来得及执行的消息，都将丢失。因为默认情况下，RabbitMQ分发完消息后，就会从内存中把消息删除掉。

处理：为了解决上述问题，RabbitMQ引入了消息确认机制，当消息处理完成后，给Server端发送一个确认消息，来告诉服务端可以删除该消息了，如果连接断开的时候，Server端没有收到消费者发出的确认信息，则会把消息转发给其他保持在线的消费者。

4.消息的持久化

现在，消费者宕机已经无法影响到我们的消息了，但如果RabbitMQ重启了，消息依然会丢失。所幸的是，RabbitMQ提供了持久化的机制，将内存中的消息持久化到硬盘上，即使重启RabbitMQ，消息也不会丢失。但是，仍然有一个非常短暂的时间窗口（RabbitMQ收到消息还没来得及存到硬盘上）会导致消息丢失，如果需要严格的控制，可以参考[官方文档](https://www.rabbitmq.com/confirms.html" \t "https://blog.csdn.net/hanghangaidoudou/article/details/_blank)

要使用RabbitMQ的消息持久化，在声明队列时设置一个参数即

**RabbitMQ的消息确认机制和持久化 防止消息丢失**