还热乎的阿里题

HashMap和Hashtable实现类

Hashtable是一个线程安全的Map实现，但HashMap是线程不安全的实现，所以HashMap比Hashtable性能要高一点；但如果有多条线程访问同一个Map对象时，使用Hashtable实现类会更好。

Hashtable不容许使用null作为key和value，如果试图把null放进Hashtable中，将会引发NullPointerException异常；但HashMap可以使用null做为key和value。

**总结：1、hashtable线程安全，基于synchronized实现，hashMap线程非安全 ； 2、hshTable不允许key value为空，hashMap可以； 3、hashMap默认大小16，按照2倍方式扩容，hashTable默认大小11，按照2倍+1的方式扩容 ；4、hashTable通过一次hash，计算出key的hash值，然后与表长取模，得到对象的散列位置，而hashMap通过两次hash，计算出key的hash值，然后在与表长减1，按位与运算，求得key的散列位置**

LinkedHashMap类

LinkedHashMap也使用双向链表来维护key-value对的次序，该链表定义了迭代顺序，该迭代顺序与key-value对的插入顺序保持一致。

LinkedHashMap需要维护元素的插入顺序，因此性能略低于HashMap的性能，但在迭代访问Map里的全部元素时将有很好的性能，因为它以链表来维护内部顺序。

**总结：LinkedHashMap在hashMap基础上，增加了一个双向链表，来记录entry对象的插入顺序，其entry节点包含：before+hashcode+key+value+next+after**

WeakHashMap实现类

WeakHashMap与HashMap的用法基本相似。但与HashMap的区别在于，HashMap的key保留对象的强引用，这意味着只要该HashMap对象不被销毁，该HashMap对象所有key所引用的对象不会被垃圾回收，HashMap也不会自动删除这些key所对应的key-value对象；但WeakHashMap的key只保留对实际对象的弱引用，这意味着当垃圾回收了该key所对应的实际对象后，WeakHashMap会自动删除该key对应的key-value对。

**总结：1、hashMap中的key是强引用，hashMap没有删除key-value的时候，垃圾回收器是无法进行垃圾回收扥**

**2、weakHashMap中的key是弱引用，垃圾回收可能对它进行垃圾回收，当回收了之后，weakHashMap会自动删除对应的k-v对象**

 IdentityHashMap实现类

IdentityHashMap实现类的实现机制与HashMap基本相似，但它在处理两个key相等时，比较独特：在IdentityHashMap中，当且仅当两个key严格相等时(key1 = key2)时，IdentityHashMap才认为两个key相等，对于普通HashMap而言，只要key1和key2通过equals比较返回true，且它们的hashCode值相等即可。

public static void main(String[] args){

IdentityHashMap<String, String> map = new IdentityHashMap<String, String>();

//下面两行代码会向map中添加两条key-value对

map.put(new String("语文"), "99");

map.put(new String("语文"), "100");

//下面两行代码会向map中添加一条key-value对

map.put("java", "89");

map.put("java", "69");

//打印结果为{java=69, 语文=100, 语文=99} System.out.println(map);

}

**总结：1、identityHahsMap存储key-vaue的时候，根据“==”内存地址来判断两个key是否相等的 2、hashMap判断两个key是否相等则是通过hashcode和equal来判断**

操作集合的工具类Collections

　Java提供了一个操作Set、List和Map等集合的工具类：Collections，该工具类里提供了大量方法对集合元素进行排序、查询和修改等操作，还提供了将集合对象设置为不可变、对集合对象实现同步控制等方法

**总结：1、collections是一个工具类，为集合提供诸如“排序”、“查询”等方法，以及不可变的对象方法和同步方法**

Queue有两个常用的实现类：LinkedList和PriorityQueue

1、LinkedList（双向链表，可以实现stack和queue）实现了list和deque

addLast、addFirst、removeFirst、removeLast、getFirst、getLast’

2、PriorityQueue实现类（基于最小堆、最大堆实现的）

PriorityQueue这种数据结构支持按照优先级取出里面的元素。这是和其它常用数据结构，比如 ArrayList, Queue, Stack等最大的区别。因为要支持优先级，而heap具有类似的结构，所以，PriorityQueue一般都是基于HEAP实现的

public static void main(String[] args){

PriorityQueue<Integer> pq = new PriorityQueue<Integer>();

pq.offer(3);

pq.offer(-6);

pq.offer(9);

//打印结果为[-6, 3, 9] System.out.println(pq);

//打印结果为-6 System.out.println(pq.peek());

//打印结果为-6 System.out.println(pq.poll());

}

**HashMap的key排序和value排序：**

public class TreeMapTest {

public static void main(String[] args) {

Map<String, String> map = new TreeMap<String, String>(

new Comparator<String>() {

public int compare(String obj1, String obj2) {

// 降序排序

return obj2.compareTo(obj1);

}

});

map.put("c", "ccccc");

map.put("a", "aaaaa");

map.put("b", "bbbbb");

map.put("d", "ddddd");

Set<String> keySet = map.keySet();

Iterator<String> iter = keySet.iterator();

while (iter.hasNext()) {

String key = iter.next();

System.out.println(key + ":" + map.get(key));

}

}

}

**按值排序**

public class HashMapTest {

public static void main(String[] args) {

Map<String, String> map = new HashMap<String, String>();

map.put("c", "ccccc");

map.put("a", "aaaaa");

map.put("b", "bbbbb");

map.put("d", "ddddd");

**//list表中存放的是map.entry对象 ，先将**

List<Map.Entry<String,String>> list = new ArrayList<Map.Entry<String,String>>(map.entrySet());

Collections.sort(list,new Comparator<Map.Entry<String,String>>() {

//升序排序

public int compare(Entry<String, String> o1,

Entry<String, String> o2) {

return o1.getValue().compareTo(o2.getValue());

}

});

for(Map.Entry<String,String> mapping:list){

System.out.println(mapping.getKey()+":"+mapping.getValue());

} }

}

阿里：

1.逻辑题；

**100的阶乘有几个0 判断有多少个5,25,75，100**

1. list set map 底层使用什么实现的有哪些典型实现

**List：ArrayList、Linkedlist、Vector、Stack、Queue**

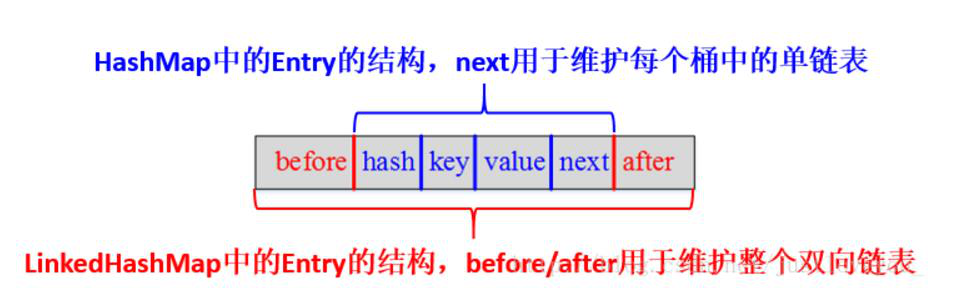
**Set：hashSet（数组+链表+红黑树）、TreeSet（红黑树）、LinkedHashSet（双向链表+hashSet）**

**Map：hashMap(数组+链表+红黑树)、TreeMap（红黑树）、LinkedHashMap（双向链表+hashMap）、hashTable、weakhashMap、IdentityHashmap**

**LinkedHashMap：**

在LinkedHashMapMap中，所有put进来的Entry都保存在如下面第一个图所示的哈希表中，但由于它又额外定义了一个以head为头结点的双向链表(如下面第二个图所示)，因此对于每次put进来Entry，除了将其保存到哈希表中对应的位置上之外，还会将其插入到双向链表的尾部。

HashMap和双向链表的密切配合和分工合作造就了LinkedHashMap。



****负载因子：****

负载因子越大则散列表的装填程度越高，也就是能容纳更多的元素，元素多了，链表大了，所以此时索引效率就会降低。

负载因子越小则链表中的数据量就越稀疏，此时会对空间造成烂费，但是此时索引效率高

HashMap什么时候扩容：

在HashMap中，我们知道随着HashMap中元素的数量越来越多，发生碰撞的概率将越来越大，所产生的子链长度就会越来越长，这样势必会影响HashMap的存取速度。为了保证HashMap的效率，系统必须要在某个临界点进行扩容处理，该临界点就是HashMap中**元素的数量在数值上等于threshold（table数组长度\*加载因子）**

**LinkedHashMap实现LRU算法：**使用LinkedHashMap实现LRU的必要前提是将accessOrder标志位设为true以便开启按访问顺序排序的模式

LinkedHashMap重写了HashMap中的recordAccess方法（HashMap中该方法为空），当调用父类的put方法时，在发现key已经存在时，会调用该方法；当调用自己的get方法时，也会调用到该方法。该方法提供了LRU算法的实现，**它将最近使用的Entry放到双向循环链表的尾部**。也就是说，当accessOrder为true时，get方法和put方法都会调用recordAccess方法使得最近使用的Entry移到双向链表的末尾；当accessOrder为默认值false时，从源码中可以看出recordAccess方法什么也不会做

**注意：实际上，可以使用LInkedList实现LRU，长度为10**

当访问元素的时候，

1. 先查看list中是否有，使用contains方法，有的话，则删除元素remove，并把元素addLast到链表尾部
2. 如果没有，则直接addLast到链表尾部
3. 每次添加元素操作的时候，都需要判断链表长度，如果大于10，则removeFirst掉第一个元素，并将元素添加到addLast到尾部



1. hashmap五个线程同时put值会发生什么后果

**线程安全问题，HashMap是非线程安全的，多个线程put的时候造成了某个key值Entry key List的死循环，问题就这么产生了。**

多线程同时put时，如果同时触发了reHash、reSize操作，会导致HashMap中的链表中出现循环节点，进而使得后面get的时候，会死循环。

1. 在多线程的环境下，存在同时其他的元素也在进行put操作，如果hash值相同，可能出现同时在同一数组下用链表表示，造成闭环，导致在get时会出现死循环，所以HashMap是线程不安全的。
2. hashmap扩容是怎么扩容的，为什么是2的幂，

**当元素个数 = table.length\*负载因子的时候，进行扩容，2倍方式；**

**在求key在table数组中实际映射位子的时候，直接可以使用&运算，提高运算速度**

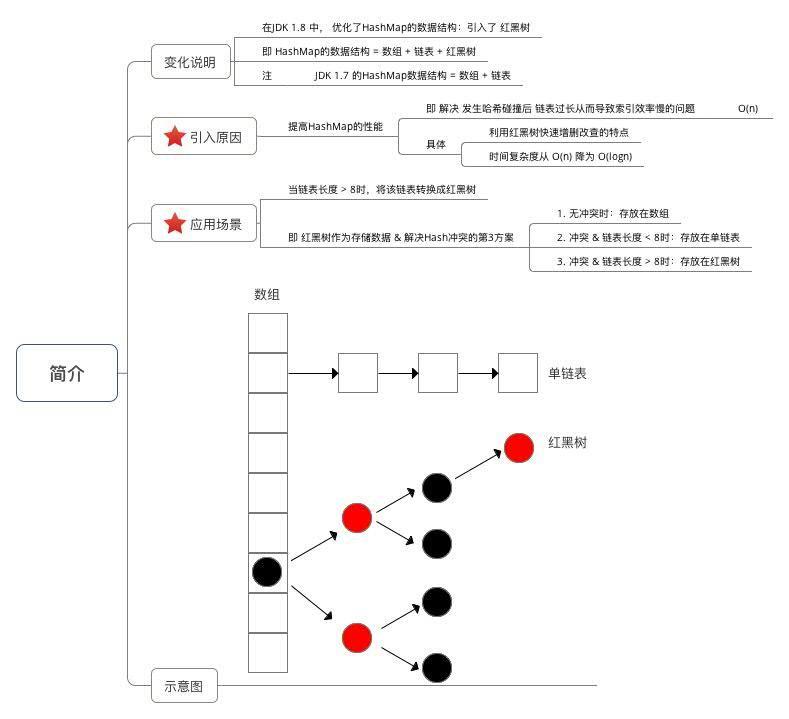
1. **concurrentHashmap**为什么线程安全，采用了什么措施应对高并发

[ConcurrentHashMap原理分析（1.7）](https://www.cnblogs.com/study-everyday/p/6430462.html)

1.7

1. 以前写过介绍HashMap的文章，文中提到过HashMap在put的时候，插入的元素超过了容量（由负载因子决定）的范围就会触发扩容操作，就是rehash，这个会重新将原数组的内容重新hash到新的扩容数组中，在多线程的环境下，存在同时其他的元素也在进行put操作，如果hash值相同，可能出现同时在同一数组下用链表表示，造成闭环，导致在get时会出现死循环，所以HashMap是线程不安全的。
2. 在JDK1.7版本中，ConcurrentHashMap的数据结构是由一个Segment数组和多个HashEntry数组+链表组成，如下图所示；每一个Segment元素存储的是HashEntry数组+链表，这个和HashMap的数据存储结构一样
3. put操作，两次hash定位：Segment继承了ReentrantLock,也就带有锁的功能，当执行put操作时，会进行第一次key的hash来定位Segment的位置，如果该Segment还没有初始化，即通过CAS操作进行赋值，然后进行第二次hash操作，找到相应的HashEntry数组的位置，这里会利用继承过来的锁的特性，在将数据插入指定的HashEntry位置时（链表的尾端）
4. Get操作：ConcurrentHashMap的get操作跟HashMap类似，只是ConcurrentHashMap第一次需要经过一次hash定位到Segment的位置，然后再hash定位到指定的HashEntry，遍历该HashEntry下的链表进行对比，成功就返回，不成功就返回null
5. 计算size：给每个Segment加上锁，然后计算ConcurrentHashMap的size返回

1.8 HashMap



TreeMap底层通过红黑树（**Red Black Tree**）实现，也就意味着containsKey(), get(), put(), remove()都有着log(n)的时间复杂度

1. **红黑树是一种近似平衡的二叉查找树，它能够确保任何一个节点的左右子树的高度差不会超过二者中较低那个的一倍**
2. 红黑树是满足如下条件的二叉查找树：

**每个节点要么是红色，要么是黑色。**

**根节点必须是黑色**

**如果节点为红色，其子节点必须为黑色。**

**对于每个节点，从该点至null（树尾端）的任何路径，都含有相同个数的黑色节点。**

1. 常用位运算

1.  判断int型变量a是奇数还是偶数      
     a&1  = 0 偶数   
     a&1 =  1 奇数   
3.  对于一个大于0的整数，判断它是不是2的几次方  
    ((x&(x-1))==0)&&(x!=0)；   
4.  比如有两个int类型变量x、y,要求两者数字交换，位运算的实现方法：性能绝对高效  
    x ^= y;   
    y ^= x;   
    x ^= y;   
6.  取模运算，采用位运算实现： hashMap中有所应用  
     a % (2^n) 等价于 a & (2^n - 1)   
7.  乘法运算   采用位运算实现  
     a \* (2^n) 等价于 a << n  
8.   除法运算转化成位运算  
      a / (2^n) 等价于 a>> n 

1. 线程池的参数有什么意义

**参数：corePoolSize、MaxmualPoolSize、keepAliveTime、unit、WorkQueue、hander（拒绝策略）**

1. 情景题:按钮点一下后台生成订单，当用户疯狂点的时候怎么正常生成订单
2. **使用Mysql唯一索引排他性（用户ID+商品ID），防止重复**
3. **按钮点击一次，就禁用disabled**

我们的抢购、秒杀就是用了这种实现以**防止超卖**。

**实现方式三：version 乐观锁**

select ...,version

update table set version+1 where version=xx

**实现方式一：利用mysql的隔离性：唯一索引**

UNIQUE KEY `catename` (`catid`)

表的'catid'字段创建名为catename的唯一索引

创建唯一索引的目的不是为了提高访问速度，而只是为了避免数据出现重复

1. 分布式框架了解吗，nginx了解吗。dubbo了解吗，docker了解吗，数据库了解吗。。（自己简历写的坑）自己作死

1. 反射能得到方法的参数名吗。int age中的age。为什么得不到

通过反射是很难获得参数名的，只能取到参数类型

作者：梦蓝樱飞  
链接：[https://www.nowcoder.com/discuss/87048?type=2&order=0&pos=11&page=1](https://www.nowcoder.com/discuss/87048?type=2&order=0&pos=11&page=1" \t "_blank)  
来源：牛客网

1. 自我介绍

2. 在学校里和实习公司里做项目用的什么技术

Java基础

1. Java读取一个文件, 有哪些方法, 考虑性能, 用哪一个类
2. **FileReader ， FileWriter ， BufferedReader 、 BufferedWriter 即IO流; 2）NIO中的FileChannel和ByteBuffer来实现**
3. **NIO读取**

RandomAccessFile raf = new RandomAccessFile(file , “rw”);

FileChannel fc = raf.getChannel();

ByteBuffer buffer = ByteBuffer.allocate(100);

Int size = Fc.read(buffer);

While(size > 0){

}

1. **IO读取**

**File file = new File(“E:/read.txt”);**

**If(!file.exist()){**

**File = File.createNewFile(file);**

**}**

**BufferedReader br = new BufferedReader(new FileReader(file,“utf-8”))**

**String line;**

**While((line = br.readLine()) != null){**

**....**

**}**

1. BIO, NIO, AIO区别

BIO == IO:面向流、堵塞、无Selector

NIO:面向Buffer和channel 、 非堵塞 、有Selector

AIO：

[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "https://blog.csdn.net/u013068377/article/details/_blank)BIO ： 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

Java NIO ： 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

Java AIO(NIO.2) ： 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，

1. NIO的组件, Selector (这个忘记了, 一脸懵逼!)

选择器，可以允许一个线程管理多个连接；

1. 什么是序列化, IO的序列化方式, 为什么需要序列化(包括在网络传输的情况下)
2. 序列化：writeObject()和ObjectOutputStream
3. 反序列化：readObject和ObjectInputStream
4. Serlizeliable接口、transient修饰属性不能序列化
5. Error和Exception区别
6. 基类为Throwable，Error和Exception继承Throwable
7. Error体系 ：  
     Error类体系描述了Java运行系统中的内部错误以及资源耗尽的情形。**应用程序无法捕获和处理的异常**。StackOverFlowError 、OutOfMemeoryError
8. Exception体系包括RuntimeException体系和其他非RuntimeException

RuntimeException：RuntimeException体系包括错误的类型转换、数组越界访问和试图访问空指针等等。如果出现这些异常，编译可以通过，在运行的时候会抛出错误。   
非RuntimeException（IOException，SqlException等等）：这类异常一般是外部错误，出现这些错误是，编译无法通过，需要使用**try catch机制捕获异常或者使用throws抛出异常。**

1. RuntimeException和非RuntimeException区别

NullPointerException - 空指针引用异常  
ClassCastException - 类型强制转换异常。  
IllegalArgumentException - 传递非法参数异常。  
ArithmeticException - 算术运算异常  
IndexOutOfBoundsException - 下标越界异常  
NumberFormatException - 数字格式异常  
classnotfoundexception-类没有找到异常

1. 什么叫线程安全, 保证线程安全的方法
2. **多线程同时访问共享数据的时候，可能会导致数据的不一致性，不确定性**
3. **同步锁（synchronized）、可重入锁（ReentrantLock）、Volatile、ThreadLocal、同步集合、并发集合**
4. **同步方法、同步块，使用原子类(atomic concurrent classes)，实现并发锁，使用volatile关键字，使用不变类和线程安全类。**
5. 加锁的方式, synchronized作用在方法上和代码块区别

**方法上，锁的粒度比较大，运行时间会更大一点**

1. synchronized(this)和synchronized(Xx.class)区别
2. synchronized(this)：synchronized(this)以及非static的synchronized方法（至于static synchronized方法请往下看），只能防止多个线程同时执行同一个对象的同步代码段。

class Sync { //锁住的对象

public synchronized void test() {

System.out.println("test开始..");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("test结束..");

}

}

class MyThread extends Thread {

public void run() {

Sync sync = new Sync();

sync.test();

}

}

public class Main {

public static void main(String[] args) {

for (int i = 0; i < 3; i++) {

Thread thread = new MyThread();

thread.start();

}

}

}

运行结果：  
*test开始..  
test开始..  
test开始..  
test结束..  
test结束..  
test结束..*

*使用同一个对象：*

*class MyThread extends Thread {*

*private Sync sync;*

*public MyThread(Sync sync) {*

*this.sync = sync;*

*}*

*public void run() {*

*sync.test();*

*}*

*}*

*public class Main {*

*public static void main(String[] args) {*

*Sync sync = new Sync(); //三个线程共享这个对象*

*for (int i = 0; i < 3; i++) {*

*Thread thread = new MyThread(sync);*

*thread.start();*

*}*

*}*

*}*

运行结果：  
*test开始..  
test结束..  
test开始..  
test结束..  
test开始..  
test结束..*

synchronized后的括号中锁同一个同一个固定对象 synchronized(Xx.class)或者static synchronized方法

synchronized(Sync.class)实现了全局锁的效果，static synchronized方法也相当于全局锁，相当于锁住了代码段

2）所以我们在用synchronized关键字的时候，能缩小代码段的范围就尽量缩小，能在代码段上加同步就不要再整个方法上加同步。这叫减小锁的粒度，使代码更大程度的并发。

Web方面:

1. Spring AOP实现原理

13. 一个页面, 一个提交按钮, 如何防止重复提交, 我说验证码, 还有其它方式吗?

 1）在 session 中存放一个特殊标志

当表单页面被请求时，生成一个特殊的字符标志串，存在 session 中，同时放在表单的隐藏域里。接受处理表单数据时，检查标识字串是否存在，并立即从 session 中删除它，然后正常处理数据。如果发现表单提交里没有有效的标志串，这说明表单已经被提交过了，忽略这次提交。这使你的 web 应用有了更高级的 XSRF 保护。

 2）使用Cookie处理 **Token(令牌)**

使用Cookie记录表单提交的状态，根据其状态可以检查是否已经提交表单。

 3）在数据库里添加约束

在数据库里添加唯一约束或创建唯一索引，防止出现重复数据。这是最有效的防止重复提交数据的方法。

1. 解决的做法就是"用JavaScript控制Form表单只能提交一次" 禁用按钮

function dosubmit(){

2 //获取表单提交按钮

3 var btnSubmit = document.getElementById("submit");

4 //将表单提交按钮设置为不可用，这样就可以避免用户再次点击提交按钮

5 btnSubmit.disabled= "**disabled"**;

6 //返回true让表单可以正常提交

7 return true;

8 }

1. ajax实现跨域请求(前端或者后台实现方法)
2. JSONP可以实现GET请求的跨域访问。
3. CORS定义一种跨域访问的机制，可以让AJAX实现跨域访问。CORS支持所有类型的HTTP请求 ；；**定义一个Filter ；；；在web.xml中配置这个filter**

CORS（**跨域资源共享**，Cross-Origin Resource Sharing）的方法。

只需要：设置HTTP响应头Access-Control-Allow-Origin，可以用通配符\*表示允许任何域的JavaScript访问资源，使用Filter的方式给服务端的所有请求都加上Access-Control-Allow-Origin响应头；

什么是跨域？：

javascript出于安全方面的考虑，不允许跨域调用其他页面的对象；a.com域名下的js无法操作b.com域名下的对象

1. cookie和session区别
2. **cookie保存在浏览器端，session保存在服务器端**
3. **Cookie存放数据大小有限制，4kb**
4. **Cookie不太安全，存放在浏览器端，通过缓存，可能访问到cookie中的数据**
5. **Session，如果保存的数据非常多，占用的空间比较大，会影响到服务器的性能**

16. forward和redirect区别

转发和重定向

1. 转发forward，发送了一次请求，redirect向服务器端发送了两次请求
2. 转发，request中保存的数据不会丢失，而重定向，数据会丢失
3. 地址栏，转发不变，重定向变化
4. 转发是访问当前web的资源，而重定向可以访问web以外的资源

数据库(这个全程被怼, 看来这个要好好复习了):

这里还有几个问题, 没听清楚, 所以就没写了!

17. 你建过索引吗? 建索引的原则

18. 索引的类型, 如主键索引

19. 查看SQL执行计划

20. 有十万条数据, 写SQL语句查询其中某字段较大值的几条数据

21. 子查询与关联查询的区别

JVM

22. 你用过的JVM命令有哪些

23. 如果一个程序发生OOM, 该怎么处理

24. Java 类加载机制

25. 为什么使用双亲委派模型

26. JVM内存模型

27. JDK7与JDK8的JVM内存模型区别

网络:

28. HTTP1.0与HTTP1.1区别

29. TCP与UDP区别

最后

30. 你有哪些问题要问我的?

个人总结:

1. 当时下午正在实习, 突然接到了杭州的电话, 话说阿里真效率, 应该是前天投的简历;

2. 问的都比较基础, 有好多问题, 我之前都复习过了, 没想到这么快就忘记了, 归根到底是没有深入理解其本质;

3. 事后发现自己回答地太快了, 有些东西没有说清楚, 估计是有点紧张啊;

4. 当时上海这边风真大, 有几个问题, 问了两遍, 硬是没有听清楚, 最后尴尬而不失礼貌地说了句"不知道".

接下来, 好好复习, 继续总结, 加油!

作者：java讨饭员  
链接：[https://www.nowcoder.com/discuss/85619?type=2&order=0&pos=8&page=1](https://www.nowcoder.com/discuss/85619?type=2&order=0&pos=8&page=1" \t "_blank)  
来源：牛客网

面试部门 蚂蚁金服中间件

问题：

1.项目balabala，项目问的很细很细

2.redis的持久化方式，redis3.0原生集群和redis读写分离+哨兵机制区别

3.工厂模式和抽象工厂模式区别

4.hashmap底层

5.分布式架构简单介绍

6.分布式架构会带来哪些问题

7.快排，归并排序原理

8.jvm垃圾回收算法和原理

9.对象进入老年代的几种方式

10.代表个人最高水平的一个技术点（好像是这个意思，应该是针对某个项目而言吧）

11.redis持久化方式