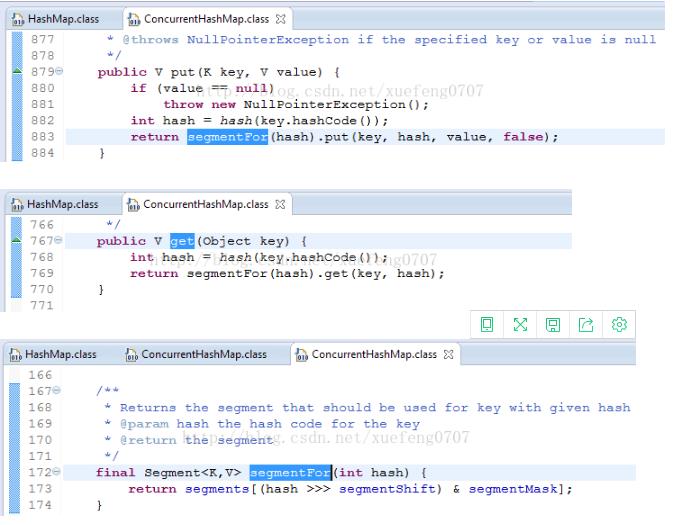
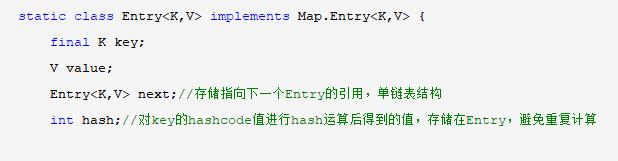
**2018/3/28 晚19：00 时长50多分钟 技术一面**

本来约了晚上8点的，我六点左右出去吃饭的时候，电话就打过来了，问能不能提前，我说我现在出来了，但是7点肯定可以。18：57打来了面试电话。

1. 自我介绍
2. Hashtable 、hashMap和ConcurrentHashMap的区别
3. HashMap不是线程安全的，因此多线程操作时需要格外小心。可以使用collections.synchronizedMap(HashMap)实现同步;
4. HashTable是线程安全的，每个方法加synchronized。
5. 在ConcurrentHashMap中，就是把Map分成了N个Segment，put和get的时候，都是现根据key.hashCode()算出放到哪个Segment中，不同的segment之间不需要考虑“线程安全”问题；ConcurrentHashMap中默认是把segments初始化为长度为16的数组。根据ConcurrentHashMap.segmentFor的算法，3、4对应的Segment都是segments[1]，7对应的Segment是segments[12]。

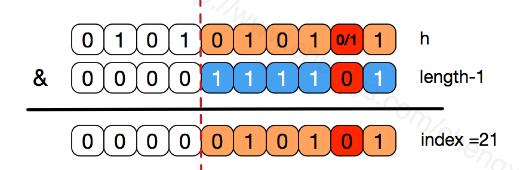


1. hashMap内部具体如何实现的
2. 哈希冲突的解决方案有多种:开放定址法（发生冲突，继续寻找下一块未被占用的存储地址），再散列函数法，链地址法
3. 而HashMap即是采用了链地址法，也就是**数组+链表**的方式
4. HashMap的主干是一个Entry数组。Entry是HashMap的基本组成单元，每一个Entry包含一个key-value键值对和指针，指向下一个entry对象。



1. 数组的长度一定是2的次幂，至于为什么这么做，后面会有详细分析

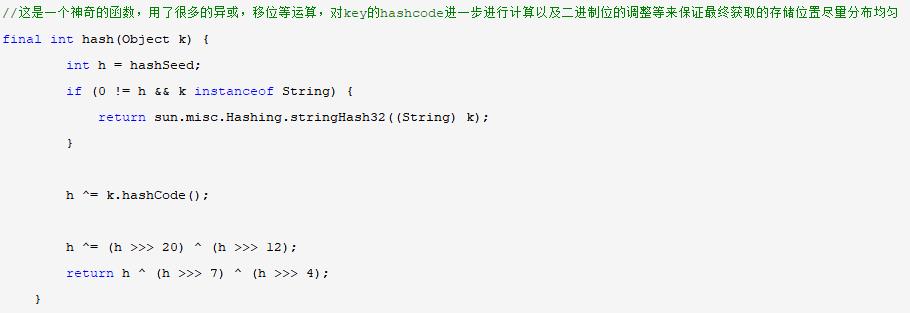
原因1：数组长度保持2的次幂，length-1的低位都为1，会使得获得的数组索引index更加均匀，哈希冲突的几率会小一点



原因2：hashMap的数组长度一定保持2的次幂，比如16的二进制表示为 10000，那么length-1就是15，二进制为01111，同理扩容后的数组长度为32，二进制表示为100000，length-1为31，二进制表示为011111。从下图可以我们也能看到这样会保证低位全为1，而扩容后只有一位差异，也就是多出了最左位的1，这样在通过 h&(length-1)的时候，只要h对应的最左边的那一个差异位为0，**就能保证得到的新的数组索引和老数组索引一致(大大减少了之前已经散列良好的老数组的数据位置重新调换)**，个人理解

1. HashMap有4个构造器，其他构造器如果用户没有传入initialCapacity 和loadFactor这两个参数，会使用默认值initialCapacity默认为16，loadFactory默认为0.75
2. Put方法







1. get方法



**在重写equals的方法的时候**，必须注意重写hashCode方法，同时还要保证通过equals判断相等的两个对象，调用hashCode方法要返回同样的整数值。而如果equals判断不相等的两个对象，其hashCode可以相同（只不过会发生哈希冲突，应尽量避免）。

1. 如果hashMap的key是一个自定义的类，怎么办

**重写hashcode和equal方法**

1. 为什么重写equals还要重写hashcode

尽管我们在进行get和put操作的时候，使用的key从逻辑上讲是等值的（通过equals比较是相等的），但由于没有重写hashCode方法，

**所以put操作时，**key(hashcode1)-->hash-->indexFor-->最终索引位置 ，

**而通过key取出value的时候** key(hashcode2)-->hash-->indexFor-->最终索引位置，由于hashcode1不等于hashcode2，导致没有定位到一个数组位置而返回逻辑上**错误的值null**（也有可能碰巧定位到一个数组位置，但是也会判断其entry的hash值是否相等，上面get方法中有提到。）

1. ArrayList和LinkedList的区别，如果一直在list的尾部添加元素，用哪个效率高？
2. ArrayList数组实现，动态数组，由于地址物理上是连续的，所以可以直接通过数组下表直接获得，取得某个位置元素效率高于linkedlist
3. LinkedList的特有方法(本身定义的方法)如:addFirst()、addLast()、getFirst()、getLast()、removeFirst()、removeLast()等，可以实现队列和栈的功能
4. 因为ArrayList是使用数组实现的,若要从数组中删除或插入某一个对象，需要移动后段的数组元素，从而会重新调整索引顺序,调整索引顺序会消耗一定的时间，所以速度上就会比LinkedList要慢许多. 相反,LinkedList是使用链表实现的,若要从链表中删除或插入某一个对象,只需要改变前后对象的引用即可!

**1.ArrayList是实现了基于动态数组的数据结构，LinkedList基于双向链表的数据结构。   
2.对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。   
3.对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据**。

1. 介绍一下Syncronized锁。如果用这个关键字修饰一个静态方法，锁住了什么？如果修饰成员方法，锁住了什么？
2. 在Java中，synchronized关键字是用来控制线程同步的，就是在多线程的环境下，控制synchronized代码段不被多个线程同时执行。synchronized既可以加在一段代码上，也可以加在方法上。
3. 实际上，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。即锁住了对象而已；注意：每一个对象，都会有自己的非static的synchronized方法，
4. static synchronized方法，static方法可以直接类名加方法名调用，方法中无法使用this，所以它锁的不是this，而是类的Class对象，所以，static synchronized方法也相当于全局锁，相当于锁住了代码段。
5. synchronized(Sync.class)实现了全局锁的效果，synchronized锁住的是括号里的对象，而不是代码



1. 介绍一下volatile
2. 在并发编程中，我们通常会遇到以下三个问题：原子性问题，可见性问题，有序性问题
3. 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。
4. 禁止进行指令重排序。
5. 但是volatile没办法保证对变量的操作的原子性

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

可见性：是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

有序性：即程序执行的顺序按照代码的先后顺序执行

指令重排序：处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性

也就是说，**要想并发程序正确地执行，必须要保证原子性、可见性以及有序性**。只要有一个没有被保证，就有可能会导致程序运行不正确。

实现自增操作的原子性：

使用synchnonied

public class Test {

public int inc = 0;

public synchronized void increase() {

inc++;

}

public static void main(String[] args) {

final Test test = new Test();

for(int i=0;i<10;i++){

new Thread(){

public void run() {

for(int j=0;j<1000;j++)

test.increase();

};

}.start();

}

while(Thread.activeCount()>1) //保证前面的线程都执行完

Thread.yield();

System.out.println(test.inc);

}

}

**采用Lock：**

public class Test {

public int inc = 0;

Lock lock = new ReentrantLock();

public void increase() {

lock.lock();

try {

inc++;

} finally{

lock.unlock();

}

}

public static void main(String[] args) {

final Test test = new Test();

for(int i=0;i<10;i++){

new Thread(){

public void run() {

for(int j=0;j<1000;j++)

test.increase();

};

}.start();

}

while(Thread.activeCount()>1) //保证前面的线程都执行完

Thread.yield();

System.out.println(test.inc);

}

}

**采用AtomicInteger：**

public class Test {

public AtomicInteger inc = new AtomicInteger();

public void increase() {

inc.getAndIncrement();

}

public static void main(String[] args) {

final Test test = new Test();

for(int i=0;i<10;i++){

new Thread(){

public void run() {

for(int j=0;j<1000;j++)

test.increase();

};

}.start();

}

while(Thread.activeCount()>1) //保证前面的线程都执行完

Thread.yield();

System.out.println(test.inc);

}

}

1. 多线程中的i++线程安全吗？为什么？

不是

i++和++i的线程安全分为两种情况：

1、**如果i是局部变量（在方法里定义的）**，那么是线程安全的。因为局部变量是线程私有的，别的线程访问不到，其实也可以说没有线程安不安全之说，因为别的线程对他造不成影响。

2、**如果i是全局变量（类的成员变量）**，那么是线程不安全的。因为如果是全局变量的话，同一进程中的不同线程都有可能访问到。

如果有大量线程同时执行i++操作，i变量的副本拷贝到每个线程的线程栈，当同时有两个线程栈以上的线程读取线程变量，假如此时是1的话，那么同时执行i++操作，再写入到全局变量，最后两个线程执行完，i会等于3而不会是2，所以，出现不安全性。

1. 如何线程安全的实现一个计数器？

上面三个都可以

1. 讲一下TCP的连接和释放连接。

****（1）、三次握手的过程****

   1）主机A向主机B发送TCP连接请求数据包，其中包含主机A的初始序列号seq(A)=x。（其中报文中同步标志位SYN=1，ACK=0，表示这是一个TCP连接请求数据报文；序号seq=x，表明传输数据时的第一个数据字节的序号是x）；  
   2）主机B收到请求后，会发回连接确认数据包。（其中确认报文段中，标识位SYN=1，ACK=1，表示这是一个TCP连接响应数据报文，并含主机B的初始序列号seq(B)=y，以及主机B对主机A初始序列号的确认号ack(B)=seq(A)+1=x+1）  
   3）第三次，主机A收到主机B的确认报文后，还需作出确认，即发送一个序列号seq(A)=x+1；确认号为ack(A)=y+1的报文；

****（2）为什么需要第三次握手?****

     简而言之：第三次握手，主机A发送一次确认是为了防止：如果客户端迟迟没有收到服务器返回的确认报文，这时他会放弃连接，重新启动一条连接请求；但问题是：服务器不知客户端没收到，所以他会收到两个连接请求，白白浪费了一条连接开销。

****（1）四次握手过程****

　　假设主机A为客户端，主机B为服务器，其释放TCP连接的过程如下：

    1） 关闭客户端到服务器的连接：首先客户端A发送一个FIN，用来关闭客户到服务器的数据传送，然后等待服务器的确认。其中终止标志位FIN=1，序列号seq=u

 　 2） 服务器收到这个FIN，它发回一个ACK，确认号ack为收到的序号加1。  
  　3） 关闭服务器到客户端的连接：也是发送一个FIN给客户端。  
 　 4） 客户段收到FIN后，并发回一个ACK报文确认，并将确认序号seq设置为收到序号加1。

     首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

****（3）为什么TCP释放连接需要四次？****

      TCP建立连接要进行三次握手，而断开连接要进行四次。这是由于TCP的半关闭造成的。因为TCP连接是全双工的(即数据可在两个方向上同时传递)所以进行关闭时每个方向上都要单独进行关闭。这个单方向的关闭就叫半关闭。当一方完成它的****数据发送任务****，就发送一个FIN来向另一方通告将要终止这个方向的连接。

****注意：发送了FIN只是表示这端不能继续发送数据(应用层不能再调用send发送)，但是还可以接收数据****

1. 讲一下浏览器从接收到一个URL到最后展示出页面，经历了哪些过程。

总体来说分为以下几个过程:

1）DNS解析：将网址解析成对应的IP地址，DNS解析是一个**递归查询**的过程。查找www.google.com的IP地址过程。首先在本地域名服务器中查询IP地址，如果没有找到的情况下，本地域名服务器会向根域名服务器发送一个请求，如果根域名服务器也不存在该域名时，本地域名会向com顶级域名服务器发送一个请求，依次类推下去。直到最后本地域名服务器得到google的IP地址并把它缓存到本地，供下次查询使用。

2）TCP连接：HTTP协议是使用TCP作为其传输层协议的

3）发送HTTP请求：HTTP报文是包裹在TCP报文中发送的，服务器端收到TCP报文时会解包提取出HTTP报文。但是这个过程中存在一定的风险，HTTP报文是明文，如果中间被截取的话会存在一些信息泄露的风险。那么在进入TCP报文之前对HTTP做一次加密就可以解决这个问题了。HTTPS协议的本质就是HTTP + SSL(or TLS)。在HTTP报文进入TCP报文之前，先使用SSL对HTTP报文进行加密。从网络的层级结构看它位于HTTP协议与TCP协议之间。

HTTP请求报文是由三部分组成: 请求行, 请求报头和请求正文。

**请求行**

格式如下:  
Method Request-URL HTTP-Version CRLF

eg: GET index.html HTTP/1.1

**请求报头**

请求报头允许客户端向服务器传递请求的附加信息和客户端自身的信息。  
PS: 客户端不一定特指浏览器，有时候也可使用Linux下的CURL命令以及HTTP客户端测试工具等。  
常见的请求报头有: Accept, Accept-Charset, Accept-Encoding, Accept-Language, Content-Type, Authorization, Cookie, User-Agent等。

从图中可以看出，**请求报头中使用了Accept, Accept-Encoding, , Connection, Cookie等**字段。Accept用于指定客户端用于接受哪些类型的信息，Accept-Encoding与Accept类似，它用于指定接受的编码方式。Connection设置为Keep-alive用于告诉客户端本次HTTP请求结束之后并不需要关闭TCP连接，这样可以使下次HTTP请求使用相同的TCP通道，节省TCP连接建立的时间。

**请求正文**

**当使用POST, PUT等方法时，通常需要客户端向服务器传递数据**。这些数据就储存在请求正文中。在请求包头中有一些与请求正文相关的信息，例如: 现在的Web应用通常采用Rest架构，请求的数据格式一般为json。这时就需要设置Content-Type: application/json。

1. 服务器处理请求并返回HTTP报文：

HTTP响应报文也是由三部分组成: 状态码, 响应报头和响应报文。

**状态码**

状态码是由3位数组成，第一个数字定义了响应的类别，且有五种可能取值:

1xx：指示信息–表示请求已接收，继续处理。

2xx：成功–表示请求已被成功接收、理解、接受。

3xx：重定向–要完成请求必须进行更进一步的操作。

4xx：客户端错误–请求有语法错误或请求无法实现。

5xx：服务器端错误–服务器未能实现合法的请求。  
平时遇到比较常见的状态码有:200, 204, 301, 302, 304, 400, 401, 403, 404, 422, 500(分别表示什么请自行查找)

**响应报头**

常见的响应报头字段有: Server, Connection...。

**响应报文**

服务器返回给浏览器的文本信息，通常HTML, CSS, JS, 图片等文件就放在这一部分

5）浏览器解析渲染页面

6）连接结束

1、首先，在浏览器地址栏中输入url

2、浏览器先查看浏览器缓存-系统缓存-路由器缓存，如果缓存中有，会直接在屏幕中显示页面内容。若没有，则跳到第三步操作。

3、在发送http请求前，需要域名解析(DNS解析)，解析获取相应的IP地址。

4、浏览器向服务器发起tcp连接，与浏览器建立tcp三次握手。

5、握手成功后，浏览器向服务器发送http请求，请求数据包。

6、服务器处理收到的请求，将数据返回至浏览器

7、浏览器收到HTTP响应

8、读取页面内容，浏览器渲染，解析html源码

9、生成Dom树、解析css样式、js交互

10、客户端和服务器交互

11、ajax查询

1. 长连接怎么实现的

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，但任务结束就中断连接。如果客户端浏览器访问的某个HTML或其他类型的 Web页中包含有其他的Web资源，如JavaScript文件、图像文件、CSS文件等；当浏览器每遇到这样一个Web资源，就会建立一个HTTP会话。

但从 HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头有加入这行代码：

**Connection**:keep-alive

在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的 TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。实现长连接要客户端和服务端都支持长连接。

HTTP协议的长连接和短连接，实质上是TCP协议的长连接和短连接

2）什么时候用长连接，短连接？

长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况，。每个TCP连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，次处理时直接发送数据包就OK了，不用建立TCP连接。**例如：数据库的连接用长连接**

像WEB网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。**所以并发量大，但每个用户无需频繁操作情况下需用短连好。**

1. GC工具用过哪些？
2. 讲一下什么情况可以影响到新生代的回收速度。
3. Minor GC、Major GC、Full GC
4. JVM性能调优主要目的是避免Full GC的发生；Full GC的慢，其实是相对于Yong GC和Old GC。
5. Minor GC回收新生代；Major GC回收年老代，个别JVM参数配置设置了Old GC之前必须执行一次Minor GC；MIXED GC回收新生代+老年代，G1回收算法；Full GC回收新生代、老年代、元数据区/永久代
6. Full GC回收新生代、老年代、元数据区/永久代。从这个角度讲，多回收了方法区，元数据区的回收算法效率低，增加了总的回收耗时。
7. **为什么老年代垃圾回收效率比新生代低很多？**

**新生代复制算法比较快**。Eden区回收时直接全部清空，存活的对象存放到内存容量比较小的s1，少了解决内存碎片整理  加上直接copy的速度，效率很高。

**老年代标记清除算法**会导致内存碎片化，因此就引入了标记整理算法，执行完毕后，存活的对象会按序放置，移动对象的内存地址（重点），来解决碎片化，但是执行时间较长。

1. GC，即就是Java垃圾回收机制。目前主流的JVM（HotSpot）采用的是分代收集算法。与C++不同的是，Java采用的是类似于树形结构的可达性分析法来判断对象是否还存在引用。即：从gcroot开始，把所有可以搜索得到的对象标记为存活对象。

**整理：**

****大部分JVM都会采用所谓的分代收集方式去回收垃圾，什么是分代收集呢？****

根据对象的**存活周期的不同**将内存划分为好几块。一般是把java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适合的收集算法。**新生代中，每次垃圾收集时都发现大批对象死去，只有少量存活，那就选用复制算法**。**老年代因为对象存活率高**，没有额外空间对它进行分配担保，那就必须使用“标记-清理”或者“标记-整理”等重量级算法来进行回收。

1：判断对象存活还是死亡的**[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "https://blog.csdn.net/zp522123428/article/details/_blank)**（引用计数算法、可达性分析算法）、

2：常见的垃圾收集算法（复制算法、分代收集算法等以及这些算法适用于什么代）以及

3：常见的垃圾收集器的特点（这些收集器适用于什么年代的内存收集）

**新生代：**

新生代又可以划分为一个Elden区和两个Survivor（幸存）区。

按照规定，新对象会首先分配在Eden中（**如果对象过大，比如大数组，将会直接放到老年代**）。在GC中，Eden中的对象会被移动到survivor中，**直至对象满足一定的年纪**（****定义为熬过minor GC的次数****），会被移动到老年代。

****新生代采取复制算法，在Minor GC之前，to survivor区域保持清空，对象保存在Eden和from survivor区，minor GC运行时，Eden中的幸存对象会被复制到to Survivor（同时对象年龄会增加1）。而from survivor区中的幸存对象会考虑对象年龄，如果年龄没达到阈值，对象依然复制到to survivor中。如果对象达到阈值那么将被移到老年代。复制阶段完成后，Eden和From幸存区中只保存死对象，可以视为清空。如果在复制过程中to幸存区被填满了，剩余的对象将被放到老年代。最后，From survivor和to survivor会调换一下名字，下次Minor GC时，To survivor变为From Survivor。****

****调优的重要性：新生代的大小设置非常重要，如果新生代过小，会导致新生对象很快就晋升到老年代中，在老年代中对象很难被回收。如果新生代过大，会发生过多的复制过程。因而我们需要找到一个合适****

**旧生代的GC：**

旧生代与新生代不同，**对象存活的时间比较长，比较稳定**，因此采用标记（Mark）算法来进行回收，所谓标记就是扫描出存活的对象，然后再进行回收未被标记的对象，回收后对用空出的空间要么进行合并，要么标记出来便于下次进行分配，总之就是要减少内存碎片带来的效率损耗。在执行机制上JVM提供了串行 GC（SerialMSC）、并行GC（parallelMSC）和并发GC（CMS）



堆大小 = 新生代 + 老年代。其中，堆的大小可以通过参数 –Xms、-Xmx 来指定。默认的新生代 ( Young ) 与老年代 ( Old ) 的比例的值为 1:2 ，默认的，Edem : from : to = 8 :1 : 1，JVM 每次只会使用 Eden 和其中的一块 Survivor 区域来为对象服务，所以无论什么时候，总是有一块Survivor 区域是空闲着的。 因此，新生代实际可用的内存空间为 9/10 ( 即90% )的新生代空间。

 Minor GC 是****发生在新生代中****的垃圾收集动作，****所采用的是复制算法****。

    新生代几乎是所有 Java 对象出生的地方，大部分对象通常不需长久存活，具有朝生夕灭的性质。

    当一个对象被判定为 "死亡" 的时候，GC 就有责任来回收掉这部分对象的内存空间。**新生代是 GC 收集垃圾的**

**频繁区域。**

   当对象在 Eden ( 包括一个 Survivor 区域，这里假设是 from 区域 ) 出生后，****在经过一次 Minor GC 后，如****

****果对象还存活，并且能够被另外一块 Survivor 区域所容纳****，****则使用复制算法将这些仍然还存活的对象复制到另外一块 Survivor 区域 ( 即 to 区域 ) 中****，然后清理所使用过的 Eden 以及 Survivor 区域 ( 即from 区域 )，****并且将这些对象的年龄设置为1，以后对象在 Survivor 区每熬过一次 Minor GC，就将对象的年龄 + 1，当对象的年龄达到某个值时 ( 默认是 15 岁，可以通过参数 -XX:MaxTenuringThreshold 来设定 )，这些对象就会成为老年代。****但这也不是一定的，****对于一些较大的对象 ( 即需要分配一块较大的连续内存空间 ) 则是直接进入到老年代****。

 Full GC 是****发生在老年代****的垃圾收集动作，所采用的是****标记-清除算法****。

****Full GC 发生的次数不会有 Minor GC 那么频繁，并且做一次 Full GC 要比进行一次 Minor GC 的时间更长。****

**Full GC 耗时是 Minor GC 的 22.89 倍。**

1. 写SQL：找出每个城市的最新一条记录（没写出来）

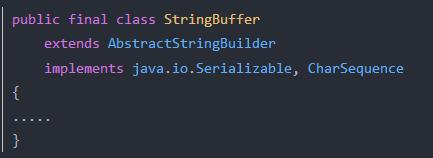
id 城市 人口 信息 创建时间  
1 北京 100 info1 时间戳  
2 北京 100 info2 时间戳  
3 上海 100 info3 时间戳  
4 上海 100 info4 时间戳

**20. 为什么选用springboot框架？**

**21. 平时怎么学技术？**

22. springboot框架源码看过吗？hashMap的源码看过吗？

**1）JDK源码分析之StringBuffer篇**



StringBuffer类跟String类一样定义成final形式，主要是为了“效率”和“安全性”的考虑







下面来看看StringBuilder类的append方法和insert方法的代码，因StringBuilder和StringBuffer的方法实现基本上一致，不同的是StringBuffer类的方法前多了个synchronized关键字，即StringBuffer是线程安全的。所以接下来我们就只分析StringBuilder类的代码了。StringBuilder类的append方法，insert方法都是Override 父类AbstractStringBuilder的方法，所以我们直接来分析AbstractStringBuilder类的相关方法

四、总结：

1、String类型的字符串对象是不可变的，一旦String对象创建后，包含在这个对象中的字符系列是不可以改变的，直到这个对象被销毁。

2、StringBuilder和StringBuffer类型的字符串是可变的，不同的是StringBuffer类型的是线程安全的，而StringBuilder不是线程安全的

3、如果是多线程环境下涉及到共享变量的插入和删除操作，StringBuffer则是首选。如果是非多线程操作并且有大量的字符串拼接，插入，删除操作则StringBuilder是首选。毕竟String类是通过创建临时变量来实现字符串拼接的，耗内存还效率不高，怎么说StringBuilder是通过JNI方式实现终极操作的。

4、StringBuilder和StringBuffer的“可变”特性总结如下：

（1）append，insert，delete方法最根本上都是调用System.arraycopy()这个方法来达到目的

（2）substring(int, int)方法是通过重新new String(value, start, end - start)的方式来达到目的。因此，在执行substring操作时，StringBuilder和String基本上没什么区别。

23. 学习中遇见了什么问题让你感觉印象最深刻？

25. 你有什么想问我的吗

**2018/4/10 晚19：00 时长44分钟 技术二面**

1. 自我介绍
2. 实验室的科研问题
3. 情景题：如果一个外卖配送单子要发布，现在有200个骑手都想要接这一单，如何保证只有一个骑手接到单子？

1)这个肯定不能直接操作数据库的，会挂的。直接读库写库对数据库压力太大，要用缓存。

　　把你要卖出的商品比如10个商品放到缓存中；然后在memcache里设置一个计数器来记录请求数，这个请求书你可以以你要秒杀卖出的商品数为基数，比如你想卖出10个商品，只允许100个请求进来。那当计数器达到100的时候，后面进来的就显示秒杀结束，这样可以减轻你的服务器的压力。然后根据这100个请求，先付款的先得后付款的提示商品以秒杀完。

2)首先，多用户并发修改同一条记录时，肯定是后提交的用户将覆盖掉前者提交的结果了。这个直接可以使用加锁机制去解决，乐观锁或者悲观锁。

　　悲观锁(Pessimistic Lock), 顾名思义，就是很悲观，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block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。

乐观锁(Optimistic Lock), 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，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。

1、在秒杀的情况下，肯定不能如此高频率的去读写数据库，会严重造成性能问题的  
必须使用缓存，将需要秒杀的商品放入缓存中，并使用锁来处理其并发情况。当接到用户秒杀提交订单的情况下，先将商品数量递减（加锁/解锁）后再进行其他方面的处理，处理失败在将数据递增1（加锁/解锁），否则表示交易成功。  
当商品数量递减到0时，表示商品秒杀完毕，拒绝其他用户的请求。

2、这个肯定不能直接操作数据库的，会挂的。直接读库写库对数据库压力太大，要用缓存。  
把你要卖出的商品比如10个商品放到缓存中；然后在memcache里设置一个计数器来记录请求数，这个请求书你可以以你要秒杀卖出的商品数为基数，比如你想卖出10个商品，只允许100个请求进来。那当计数器达到100的时候，后面进来的就显示秒杀结束，这样可以减轻你的服务器的压力。然后根据这100个请求，先付款的先得后付款的提示商品以秒杀完。

3、首先，多用户并发修改同一条记录时，肯定是后提交的用户将覆盖掉前者提交的结果了。

这个直接可以使用加锁机制去解决，乐观锁或者悲观锁。  
****乐观锁****，就是在数据库设计一个版本号的字段，每次修改都使其+1，这样在提交时比对提交前的版本号就知道是不是并发提交了，但是有个缺点就是只能是应用中控制，如果有跨应用修改同一条数据乐观锁就没办法了，这个时候可以考虑悲观锁。  
****悲观锁****，就是直接在数据库层面将数据锁死，类似于oralce中使用select xxxxx from xxxx where xx=xx for update，这样其他线程将无法提交数据。  
除了加锁的方式也可以使用接收锁定的方式，思路是在数据库中设计一个状态标识位，用户在对数据进行修改前，将状态标识位标识为正在编辑的状态，这样其他用户要编辑此条记录时系统将发现有其他用户正在编辑，则拒绝其编辑的请求，类似于你在操作系统中某文件正在执行，然后你要修改该文件时，系统会提醒你该文件不可编辑或删除。

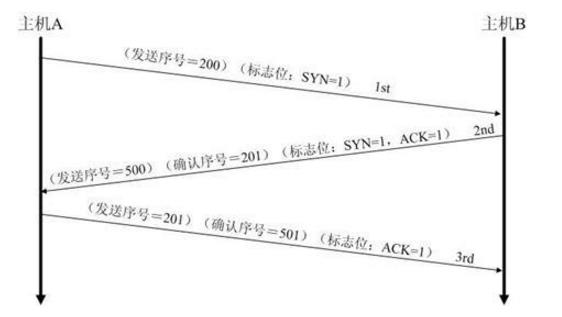
1. 介绍一下volitile。
2. 分布式集群中如何保证线程安全？

1 保证一些共享数据在一个线程更新时从开始到结束，其他线程不能占用，这个可以通过Synchronized代码块实现。以避免更新过程中出现的脏读。

<https://blog.csdn.net/fashion138/article/details/78869430>

**大规模分布式应用之海量数据和高并发解决方案总结**

1. 介绍一下TCP的三次握手？每次发送了什么信号？如果去掉最后一次握手会怎样？



（1）第一次握手：建立连接时，客户端A发送SYN包（SYN=j）到服务器B，并进入SYN\_SEND状态，等待服务器B确认。

（2）第二次握手：服务器B收到SYN包，必须确认客户A的SYN（ACK=j+1），同时自己也发送一个SYN包（SYN=k），即SYN+ACK包，此时服务器B进入SYN\_RECV状态。

（3）第三次握手：客户端A收到服务器B的SYN＋ACK包，向服务器B发送确认包ACK（ACK=k+1），此包发送完毕，客户端A和服务器B进入ESTABLISHED状态，完成三次握手。

1. 一个网页从输入URL到服务器接收到请求所经历的过程？

1）DNS解析；2）tcp连接请求；3）发送http请求；4）响应http报文；5）浏览器渲染；6）释放tcp连接

1. TCP协议在哪一层？IP协议在那一层？HTTP在哪一层？

**传输层、网络层、应用层**

1. 数据库索引介绍一下。介绍一下什么时候用Innodb什么时候用MyISAM。
2. 索引加快搜索速度，这里就出现了索引。索引是对某个字段进行排序的一种方式。**对表中的某个字段建立索引会创建另一种数据结构**，即B+树，其中保存着字段的值，每个值又指向与它相关的记录。----->例如这样一个查询：select \* from table1 where id=10000。如果没有索引，必须遍历整个表，直到ID等于10000的这一行被找到为止；有了[索引](https://baike.baidu.com/item/%E7%B4%A2%E5%BC%95/5716853" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E7%B4%A2%E5%BC%95/_blank)之后(必须是在ID这一列上建立的索引)，即可在索引中查找。由于索引是经过某种算法优化过的，因而查找次数要少的多。
3. 索引分为[聚簇索引](https://baike.baidu.com/item/%E8%81%9A%E7%B0%87%E7%B4%A2%E5%BC%95" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E7%B4%A2%E5%BC%95/_blank)和[非聚簇索引](https://baike.baidu.com/item/%E9%9D%9E%E8%81%9A%E7%B0%87%E7%B4%A2%E5%BC%95" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E7%B4%A2%E5%BC%95/_blank)两种，**所谓聚簇索引**，就是指主索引文件和数据文件为同一份文件，聚簇索引主要用在Innodb存储引擎中。在该索引实现方式中B+Tree的叶子节点上的data就是数据本身，key为主键;**非聚簇索引**就是指B+Tree的叶子节点上的data，并不是数据本身，而是数据存放的地址。主索引和辅助索引没啥区别，只是主索引中的key一定得是唯一的。主要用在MyISAM存储引擎中;**非聚簇索引比聚簇索引多了一次读取数据的IO操作，所以查找性能上会差。**
4. B+Tree的每个叶子节点增加一个指向相邻叶子节点的指针，就形成了带有顺序访问指针的B+Tree。做这个优化的目的是为了提高区间访问的性能
5. 索引的好处是特别明显的，那就是大大的提高了查询的速度。但是相对应的也带来了一些不好的地方。

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

1. 索引类型
2. 普通索引  
   是最基本的索引，它没有任何限制。它有以下几种创建方式：

CREATE INDEX index\_name ON table(column(length))

ALTER TABLE table\_name ADD INDEX index\_name ON (column(length))

1. 唯一索引  
   与前面的普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。

CREATE UNIQUE INDEX indexName ON table(column(length))

ALTER TABLE table\_name ADD UNIQUE indexName ON (column(length))

1. 主键索引  
   **是一种特殊的唯一索引，一个表只能有一个主键，不允许有空值。**一般是在建表的时候同时创建主键索引
2. 组合索引  
   指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。使用组合索引时遵循最左前缀集合
3. 全文索引  
   主要用来查找文本中的关键字

**InnoDB：**支持事务处理等  
支持外键  
支持行锁  
不支持FULLTEXT类型的索引  
不保存表的具体行数，扫描表来计算有多少行  
DELETE 表时，是一行一行的删除  
InnoDB 把数据和索引存放在表空间里面  
跨平台可直接拷贝使用  
InnoDB中必须包含AUTO\_INCREMENT类型字段的索引  
表格很难被压缩

**MyISAM：**  
不支持事务，回滚将造成不完全回滚，不具有原子性  
不支持外键  
支持全文搜索  
保存表的具体行数,不带where时，直接返回保存的行数  
DELETE 表时，先drop表，然后重建表  
MyISAM 表被存放在三个文件 。frm 文件存放表格定义。 数据文件是MYD (MYData) 。 索引文件是MYI (MYIndex)引伸  
跨平台很难直接拷贝  
MyISAM中可以使AUTO\_INCREMENT类型字段建立联合索引  
表格可以被压缩

**选择场景：**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。且MyISAM恢复速度快。可直接用备份覆盖恢复。  
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。

1. 介绍一下聚簇索引和非聚簇索引
2. 介绍一下spring
3. 知道什么设计模式
4. 介绍一下单例模式？懒汉式的单例模式如何实现单例？
5. 介绍一下策略模式
6. 平时都看什么书
7. 你有什么要问我的吗？
8. 实习时间？

## **集合类：**

　　下面先针对集合类来进行分析：集合类主要分成了三类集合：队列类型、集合类型、哈希类型。

### **1、队列类型，它是一种线性的数据结构**

　　（1）ArrayList:

　　　　底层存储是通过数组进行存储的，所以他可以直接通过数组下标进行访问，所以随机访问速度很快。

　　　　但是在扩容方面效率很低，他需要把数据进行重新进行拷贝。

　　（2）LinkedList:

　　　　底层是通过链表进行存储的，他不能支持随机访问，只能从头节点或者尾节点进行遍历，增加元素成本很低，不存在拷贝数据的操作。

　　（3）Vector:

　　　　底层存储和ArrayList一样是通过数组进行存储的，但是它的方法是同步操作的，所以说他是线程安全的，

　　　　性能方面差于ArrayList，扩容方面Vector元素是翻倍，是要多于ArrayList的50%的增长率。

　　（4）Stack:

　　　　是Vector的子类，他是实现栈的操作，只有push、pop、peek几个操作，他也是线程安全的类

　　（5）ArrayDeque:

　　　　底层存储室通过数组进行存储的双端队列，拥有一个头指针和一个尾指针，当头指针和尾指针在同一个位置的时候，会进行翻倍扩容。

　　　　同时他支持栈和队列的操作，但是不支持随机访问。

### **2、集合类型，它是一种非线性的数据结构，但是集合里面只会保留一个相同的元素**

　　（1）HashSet:

　　　　底层通过HashMap进行存储，里面不允许重复的值，是一种最常用的集合类的数据结构

　　（2）EnumSet:

　　　　是一个枚举类型的集合，它里面的元素必须全部来自单个枚举类型，在内部所有元素只占长整型的一位，所以时间和空间性能相当的好。

　　（3）LinkedHashSet:

　　　　 它是继承于HashSet的，但是它保证了插入元素的插入顺序，所以能够保证迭代时候的顺序和插入时候的顺序保持一致。

　　（4）TreeSet:

　　　　它是基于树型结构来构造的集合，它底层通过TreeMap来进行存储，

　　　　又因为TreeMap是基于数据结构里面的红黑树，所以它内部的元素是可以进行排序的。

　　　　因为结构是树形结构，所以在读取或者写入的速度方面要慢于HashSet。

### **3、哈希类型，它是一种key-value的结构，可以通过key可以快速定位到value**

　　（1）HashMap:

　　　　最常用的Map结构，它是通过数组加链表的形式进行存储的。

　　（2）EnumMap:

　　　　枚举类型的Map结构，它的key是一个枚举类型，它效率要高于HashMap，因为它内部是只有数组进行存储的，可以通过直接定位到具体元素，

　　　　而不用像上面的HashMap去进行计算，所以效率比HashMap要大大提高。

　　（3）IdentityHashMap:

　　　　它可以保存多个一样值的key，只有当k1==k2的时候才认为是重复的。然后进行查找的时候，也是根据具体的key地址进行查找，

　　　　而不是通过key的值进行查找的。

　　（4）LinkedHashMap:

　　　　类似于HashMap，但是能够保证迭代时候的顺序和插入时候的顺序保持一致。

　　（5）TreeMap:

　　　　它基于树型结构来存储的，树形结构是基于数据结构里面的红黑树，所以它内部的元素也是可以进行排序的。

　　（6）HashTable:

　　　　它类似于HashMap, 区别就是HashTable是一个线程安全的，同时它的key和value都是不能为null的。

　　（7）Properties:

　　　　继承于HashTable，所以它也是线程安全的，并且可以通过io操作来读取文件里面的内容，来构成Properties对象。

　　（8）WeakHashMap:

　　　　它里面的key都是弱引用的哈希实现，对于一个给定的键，其映射的存在并不阻止垃圾回收器对该键的丢弃，

　　　　这就使该键成为可终止的，被终止，然后被回收。某个键被终止时，它对应的键值对也就从映射中有效地移除了。

　　　　它是通过ReferenceQueue来进行实现的，这个是前面的lang包里面的类。其他方面和HashMap没有差别。

# **[JDK源码分析-Integer](https://www.cnblogs.com/wxisme/p/6308517.html)**

1：Integer类还实现了Comparable接口用以比较两个Integer的大小。

//源码public final class Integer extends Number implements Comparable<Integer>

Integer类中规定了范围大小时在-2^31~2^31-1之间。

2：接下来看一下这个不简单的parseInt方法。（判断正负号、判断非法字符、判断是否越界）

parseInt("0", 10) returns 0

parseInt("473", 10) returns 473

parseInt("+42", 10) returns 42

parseInt("-0", 10) returns 0

parseInt("-FF", 16) returns -255

parseInt("1100110", 2) returns 102

parseInt("2147483647", 10) returns 2147483647

parseInt("-2147483648", 10) returns -2147483648

parseInt("2147483648", 10) throws a NumberFormatException

parseInt("99", 8) throws a NumberFormatException

parseInt("Kona", 10) throws a NumberFormatException

parseInt("Kona", 27) returns 411787

3：Integer的缓存机制和自动拆箱装箱机制

//首先要明确一点，对象之间的==是比较内存地址，常数之间的比较是数值比较。  
　　public static void main(String[] args) {

Integer num1 = new Integer(100);

Integer num2 = new Integer(100);

System.out.println(num1 == num2);//false,因为这两个对象是独立创建的，有自己的内存空间和地址。

Integer num3 = 100;

Integer num4 = 100;

System.out.println(num3 == num4);//true，常数之间比较数值。

Integer num5 = 128;

Integer num6 = 128;

System.out.println(num5 == num6);//false，自动装箱成对象，但是超过了默认的缓存范围，同第一个。如果是127就是true。

Integer num7 = 100;

Integer num8 = new Integer(100);

System.out.println(num7 == num8);//false，两个对象之间比较内存地址，不同的是num7通过自动装箱调用valueOf方法，指向缓存的100，而num8是指向自己内存空间里的100.

int num9 = 100;

Integer num10 = new Integer(100);  
　　　　 System.out.println(num9 == num10);//true，Integer对象和int比较时，Integer会自动拆箱（intValue方法）成为int，变成两个数值比较。

Integer num11 = 100;

System.out.println(num9 == num11);//true，num11通过自动装箱调用valueOf方法指向缓存中的100，比较的时候缓存中的100对象自动拆箱成为数值100.

}

# **[理解HTTP协议](https://www.cnblogs.com/wxisme/p/6212797.html)**

1：HTTP（HyperText Transfer Protocol）协议是基于TCP的应用层协议，它不关心数据传输的细节，主要是用来规定客户端和服务端的数据传输格式

2：HTTPS是HTTP协议的安全版本，HTTP协议的数据传输是明文的，是不安全的，HTTPS使用了SSL/TLS协议进行了加密处理

**1.HTTP协议是无状态的**

就是说每次HTTP请求都是独立的，任何两个请求之间没有什么必然的联系。但是在实际应用当中并不是完全这样的，**引入了Cookie和Session机制来关联请求。**

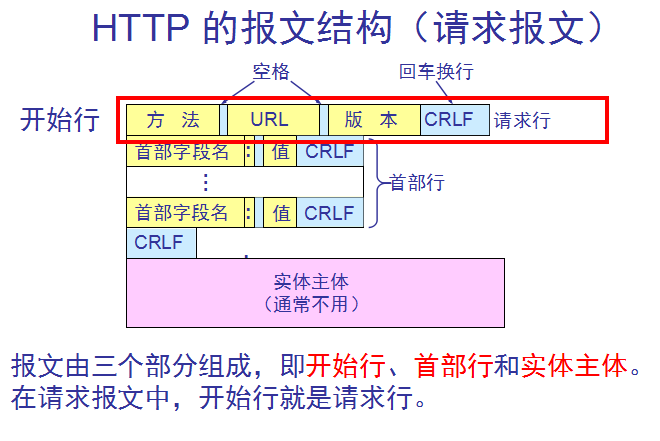
**2.多次HTTP请求**

在客户端请求网页时多数情况下并不是一次请求就能成功的，服务端首先是响应HTML页面，然后浏览器收到响应之后发现HTML页面还引用了其他的资源，例如，CSS，JS文件，图片等等，还会自动发送HTTP请求这些需要的资源。现在的HTTP版本支持管道机制，可以同时请求和响应多个请求，大大提高了效率。

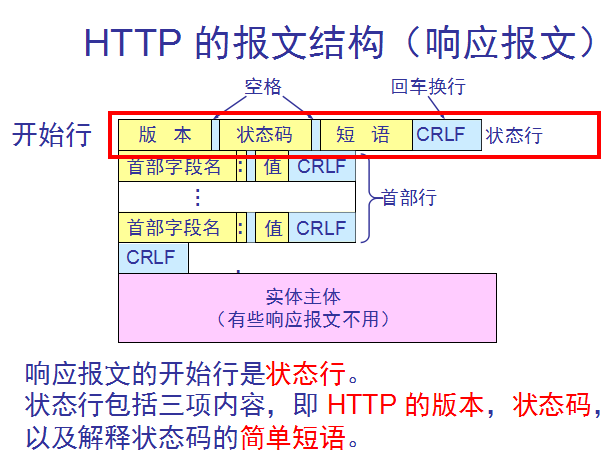
**3.基于TCP协议**

HTTP协议目的是规定客户端和服务端数据传输的格式和数据交互行为，并不负责数据传输的细节。底层是基于TCP实现的。现在使用的版本当中是默认持久连接的，也就是多次HTTP请求使用一个TCP连接。

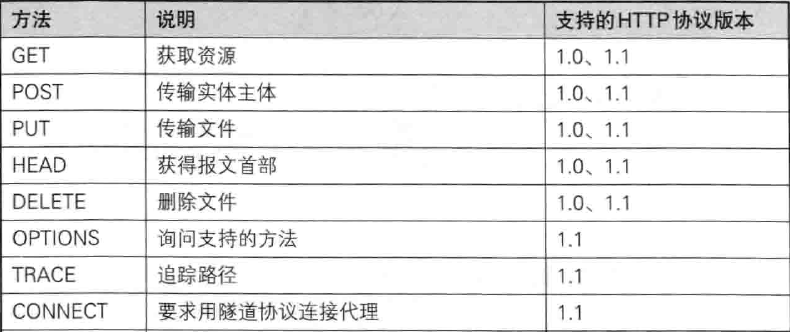
**请求报文**



其中请求行指定的是请求方法、请求URL、协议版本；请求头是键值对的形式存在的，就是字段名：值；内容实体就是要传输的数据



响应报文由状态行、响应首部字段（响应头）、响应实体组成，其中第一行是状态行，依次包含HTTP版本，状态码和状态短语组成；在一个回车换行之后是响应头，也是键值对的形式，字段名：值；然后会有一个空行也包含回车换行，之后是响应实体，就是要传输的数据





**1.GET和POST的区别**

A. 从字面意思和HTTP的规范来看，GET用于获取资源信息而POST是用来更新资源信息。

　　B. GET提交请求的数据实体会放在URL的后面，用?来分割，参数用&连接，举个栗子：/index.html?name=wang&login=1

　　C. GET提交的数据长度是有限制的，因为URL长度有限制，具体的长度限制视浏览器而定。而POST没有。

　　D. GET提交的数据不安全，因为参数都会暴露在URL上。

**3.Cookie和Session的区别和联系**

　1：首先一个最大的区别就是Cookie是保存在客户端而Session就保存在服务端的。Cookie是客户端请求服务端时服务器会将一些信息**以键值对的形式**返回给客户端，保存在浏览器中，交互的时候可以加上这些Cookie值。用Cookie就可以方便的做一些缓存。Cookie的缺点是大小和数量都有限制；Cookie是存在客户端的可能被禁用、删除、篡改，是不安全的；Cookie如果很大，每次要请求都要带上，这样就影响了传输效率。

2：Session是基于Cookie来实现的，不同的是Session本身存在于服务端，但是每次传输的时候不会传输数据，只是把代表一个客户端的唯一ID（通常是JSESSIONID）写在客户端的Cookie中，这样每次传输这个ID就可以了。Session的优势就是传输数据量小，比较安全。但是Session也有缺点，就是如果Session不做特殊的处理容易失效、过期、丢失或者Session过多导致服务器内存溢出，并且要实现一个稳定可用安全的分布式Session框架也是有一定复杂度的。在实际使用中就要结合Cookie和Session的优缺点针对不同的问题来设计解决方案。

十大排序：

1：冒泡排序

public class BubbleSort {

public static void bubbleSort(int[] arr) {

if(arr == null || arr.length == 0)

return ;

for(int i=0; i<arr.length-1; i++) {

for(int j=arr.length-1; j>i; j--) {

if(arr[j] < arr[j-1]) {

swap(arr, j-1, j);

}

}

}

}

public static void swap(int[] arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

2：选择排序

public class SelectSort {

public static void selectSort(int[] arr) {

if(arr == null || arr.length == 0)

return ;

int minIndex = 0;

for(int i=0; i<arr.length-1; i++) { //只需要比较n-1次

minIndex = i;

for(int j=i+1; j<arr.length; j++) { //从i+1开始比较，因为minIndex默认为i了，i就没必要比了。

if(arr[j] < arr[minIndex]) {

minIndex = j;

}

}

if(minIndex != i) { //如果minIndex不为i，说明找到了更小的值，交换之。

swap(arr, i, minIndex);

}

}

}

public static void swap(int[] arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

3：插入排序

public class InsertSort {

public static void insertSort(int[] arr) {

if(arr == null || arr.length == 0)

return ;

for(int i=1; i<arr.length; i++) { //假设第一个数位置时正确的；要往后移，必须要假设第一个。

int j = i;

int target = arr[i]; //待插入的

//后移

while(j > 0 && target < arr[j-1]) {

arr[j] = arr[j-1];

j --;

}

//插入

arr[j] = target;

}

}

}

4：快速排序

public class QuickSort {

public static int partition(int[] arr, int left, int right) {

int pivotKey = arr[left];

while(left < right) {

while(left < right && arr[right] >= pivotKey)

right --;

arr[left] = arr[right]; //把小的移动到左边

while(left < right && arr[left] <= pivotKey)

left ++;

arr[right] = arr[left]; //把大的移动到右边

}

arr[left] = pivotKey; //最后把pivot赋值到中间

return left;

}

public static void quickSort(int[] arr, int left, int right) {

if(left >= right)

return ;

int pivotPos = partition(arr, left, right);

quickSort(arr, left, pivotPos-1);

quickSort(arr, pivotPos+1, right);

}

public static void sort(int[] arr) {

if(arr == null || arr.length == 0)

return ;

quickSort(arr, 0, arr.length-1);

}

}

5：堆排序

public class HeapSort {

public static void heapAdjust(int[] arr, int start, int end) {

int temp = arr[start];

for(int i=2\*start+1; i<=end; i\*=2) {

//左右孩子的节点分别为2\*i+1,2\*i+2

//选择出左右孩子较小的下标

if(i < end && arr[i] < arr[i+1]) {

i ++;

}

if(temp >= arr[i]) {

break; //已经为大顶堆，=保持稳定性。 }

arr[start] = arr[i]; //将子节点上移

start = i; //下一轮筛选

}

arr[start] = temp; //插入正确的位置

}

public static void heapSort(int[] arr) {

if(arr == null || arr.length == 0)

return ;

//建立大顶堆

for(int i=arr.length/2; i>=0; i--) {

heapAdjust(arr, i, arr.length-1);

}

for(int i=arr.length-1; i>=0; i--) {

swap(arr, 0, i);

heapAdjust(arr, 0, i-1);

}

}

public static void swap(int[] arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

6：希尔排序

public class ShellSort {

public static void shellInsert(int[] arr, int d) {

for(int i=d; i<arr.length; i++) {

int j = i - d;

int temp = arr[i]; //记录要插入的数据

while (j>=0 && arr[j]>temp) { //从后向前，找到比其小的数的位置

arr[j+d] = arr[j]; //向后挪动

j -= d;

}

if (j != i - d) //存在比其小的数

arr[j+d] = temp;

}

}

public static void shellSort(int[] arr) {

if(arr == null || arr.length == 0)

return ;

int d = arr.length / 2;

while(d >= 1) {

shellInsert(arr, d);

d /= 2;

}

}

}

7：归并排序

public class MergeSort {

public static void mergeSort(int[] arr) {

mSort(arr, 0, arr.length-1);

}

/\*\*

\* 递归分治

\* @param arr 待排数组

\* @param left 左指针

\* @param right 右指针

\*/

public static void mSort(int[] arr, int left, int right) {

if(left >= right)

return ;

int mid = (left + right) / 2;

mSort(arr, left, mid); //递归排序左边

mSort(arr, mid+1, right); //递归排序右边

merge(arr, left, mid, right); //合并

}

/\*\*

\* 合并两个有序数组

\* @param arr 待合并数组

\* @param left 左指针

\* @param mid 中间指针

\* @param right 右指针

\*/

public static void merge(int[] arr, int left, int mid, int right) {

//[left, mid] [mid+1, right]

int[] temp = new int[right - left + 1]; //中间数组

int i = left;

int j = mid + 1;

int k = 0;

while(i <= mid && j <= right) {

if(arr[i] <= arr[j]) {

temp[k++] = arr[i++];

}

else {

temp[k++] = arr[j++];

}

}

while(i <= mid) {

temp[k++] = arr[i++];

}

while(j <= right) {

temp[k++] = arr[j++];

}

for(int p=0; p<temp.length; p++) {

arr[left + p] = temp[p];

}

}

}

8：计数排序

public class CountSort {

public static void countSort(int[] arr) {

if(arr == null || arr.length == 0)

return ;

int max = max(arr);

int[] count = new int[max+1];

Arrays.fill(count, 0);

for(int i=0; i<arr.length; i++) {

count[arr[i]] ++;

}

int k = 0;

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

for(int j=0; j<count[i]; j++) {

arr[k++] = i;

}

}

}

public static int max(int[] arr) {

int max = Integer.MIN\_VALUE;

for(int ele : arr) {

if(ele > max)

max = ele;

}

return max;

}

}

9：桶排序

public class BucketSort {

public static void bucketSort(int[] arr) {

if(arr == null && arr.length == 0)

return ;

int bucketNums = 10; //这里默认为10，规定待排数[0,100)

List<List<Integer>> buckets = new ArrayList<List<Integer>>(); //桶的索引

for(int i=0; i<10; i++) {

buckets.add(new LinkedList<Integer>()); //用链表比较合适 }

//划分桶

for(int i=0; i<arr.length; i++) {

buckets.get(f(arr[i])).add(arr[i]);

}

//对每个桶进行排序

for(int i=0; i<buckets.size(); i++) {

if(!buckets.get(i).isEmpty()) {

Collections.sort(buckets.get(i)); //对每个桶进行快排 }

}

//还原排好序的数组

int k = 0;

for(List<Integer> bucket : buckets) {

for(int ele : bucket) {

arr[k++] = ele;

}

}

}

/\*\*

\* 映射函数

\* @param x

\* @return

\*/

public static int f(int x) {

return x / 10;

}

}

10：基数排序

public class RadixSort {

public static void radixSort(int[] arr) {

if(arr == null && arr.length == 0)

return ;

int maxBit = getMaxBit(arr);

for(int i=1; i<=maxBit; i++) {

List<List<Integer>> buf = distribute(arr, i); //分配

collecte(arr, buf); //收集 }

}

/\*\*

\* 分配

\* @param arr 待分配数组

\* @param iBit 要分配第几位

\* @return

\*/

public static List<List<Integer>> distribute(int[] arr, int iBit) {

List<List<Integer>> buf = new ArrayList<List<Integer>>();

for(int j=0; j<10; j++) {

buf.add(new LinkedList<Integer>());

}

for(int i=0; i<arr.length; i++) {

buf.get(getNBit(arr[i], iBit)).add(arr[i]);

}

return buf;

}

/\*\*

\* 收集

\* @param arr 把分配的数据收集到arr中

\* @param buf

\*/

public static void collecte(int[] arr, List<List<Integer>> buf) {

int k = 0;

for(List<Integer> bucket : buf) {

for(int ele : bucket) {

arr[k++] = ele;

}

}

}

/\*\*

\* 获取最大位数

\* @param x

\* @return

\*/

public static int getMaxBit(int[] arr) {

int max = Integer.MIN\_VALUE;

for(int ele : arr) {

int len = (ele+"").length();

if(len > max)

max = len;

}

return max;

}

/\*\*

\* 获取x的第n位，如果没有则为0.

\* @param x

\* @param n

\* @return

\*/

public static int getNBit(int x, int n) {

String sx = x + "";

if(sx.length() < n)

return 0;

else

return sx.charAt(sx.length()-n) - '0';

}

}

有趣的位运算：

1：技巧应用：给一个数组，只有两个数出现了一次，剩下的都出现了两次，找出出现一次的两个数字。

public int[] singleNumber(int[] nums) {

//用于记录，区分“两个”数组

int diff = 0;

for(int i = 0; i < nums.length; i ++) {

diff ^= nums[i];

}

//取最后一位1

diff &= -diff;

int[] rets = {0, 0};

for(int i = 0; i < nums.length; i ++) {

//分属两个“不同”的数组

if ((nums[i] & diff) == 0) {

rets[0] ^= nums[i];

}

else {

rets[1] ^= nums[i];

}

}

return rets;

}

**取a最后一位1的位置a & (-a)**

**2：判断一个数是否是2的次幂。从二进制的角度思考，一个数如果是2的次幂，那么需要满足这个数大于0，这个数的二进制表示有且只有一个1**

public boolean isPowerOf2(int n) {

return n > 0 && (n & (n - 1)) == 0;

}

**去掉a的最后一位1 a & (a - 1)**

**3：求一个整数的二进制表示的1的个数。有了这个技巧这个问题就非常简单了，把1全部消去，看消了几次就可以了**

public int countOnes(int num) {

int count = 0;

while (num != 0) {

num = num & (num - 1);

count++;

}

return count;

}

**4：**求一个整数转化为另一个整数需要改变多少位。这个问题也就是求两个整数有多少位不同就行了，改变不同的位置就能变成另一个数。使用异或非常简单的求出有多少位不同，然后问题就变成了上一个问题，求异或结果的1的个数。

public int countOnes(int num) {

int count = 0;

while (num != 0) {

num = num & (num - 1);

count++;

}

return count;

}public int bitSwapRequired(int a, int b) {

return countOnes(a ^ b);

}

5：求一个集合的所有子集，如果集合为{1,2,3}



public List<List<Integer>> subsets(int[] nums) {

List<List<Integer>> ret = new ArrayList<>();

int n = nums.length;

for (int i=0; i<(1 << n); i++) {

List<Integer> subset = new ArrayList<Integer>();

for (int j=0; j<n; j++) {

if ((i & (1<<j)) != 0) //检查是否是1

subset.add(nums[j]);

}

ret.add(subset);

}

return ret;

}

**6：**给两个8G的文件，文件每行存储了一个正整数，求这两个文件的交集

//伪代码

bit=0while (num1 = readLine(file1)) {

bit |= (1 << num1)

}while (num2 = readLine(file2)) {

if ((bit & (1 << num2)) == 0)

continue

else

writeLine(file3, num2)

}

例如求差集、并集、对海量的URL去重、网页去重、垃圾邮件过滤等等，这些问题都可以用类似的思路去解决，只不过在真实的工程实践中很多代码可以不用写的这么底层。可以使用已经实现好的BitSet、Redis Bit和Bloomfilter。

# **[LRU(K-V)缓存](https://www.cnblogs.com/wxisme/p/4889846.html)**

1：LRU(Least Recently Used)算法是缓存技术中的一种常见思想，顾名思义，最近最少使用，也就是说有两个维度来衡量，一个是时间（最近），一个频率（最少）。如果需要按优先级来对缓存中的K-V实体进行排序的话，需要考虑这两个维度，在LRU中，最近使用频率最高的排在前面，也可以简单的说最近访问的排在前面。这就是LRU的大体思想

2：如果足够了解LinkedHashMap，实现LRUCache也是非常简单的。在LinkedHashMap中提供了可以设置容量、装载因子和顺序的构造方法。如果要实现LRUCache就可以把顺序的参数设置成true，代表访问顺序，而不是默认的FIFO的插入顺序。这里把装载因子设置为默认的0.75。并且还要重写removeEldestEntry()方法来维持当前的容量。这样一来可以有两种方法来实现LinkedHashMap版本的LRUCache。一种是继承一种是组合

3：实现代码：

public class LRUCache {

private final int initialCapacity;

private Map<Integer, Integer> cache;

public LRUCache(final int initialCapacity) {

this.initialCapacity = initialCapacity;

cache = new LinkedHashMap<Integer, Integer>(initialCapacity, 0.75f, true) {

@Override

protected boolean removeEldestEntry(

Map.Entry<Integer, Integer> eldest) {

return size() > initialCapacity;

}

};

}

public void put(int key, int value) {

cache.put(key, value);

}

public int get(int key) {

return cache.get(key);

}

public void remove(int key) {

cache.remove(key);

}

@Override

public String toString() {

StringBuilder cacheStr = new StringBuilder();

cacheStr.append("{");

for (Map.Entry<Integer, Integer> entry : cache.entrySet()) {

cacheStr.append("[" + entry.getKey() + "," + entry.getValue() + "]");

}

cacheStr.append("}");

return cacheStr.toString();

}

}

# ping某个域名的详细过程

<https://blog.csdn.net/guoweimelon/article/details/50865642>

主要用到了DNS、ARP、ICMP 注意：没有用到TCP

### [桥接模式--将抽象和实现变化分离](http://www.baidu.com/link?url=-MCm4-uVG9wEgrYAJBiix8Y8DruqcnFhjfyOpEBBeT7oObMD1TGoEZ6g7gUa1hpoKzE7djrEDMxEw8fSe7CtIz9X9bFPGMu6TQOls6rTIfa" \t "https://www.baidu.com/_blank)

<https://blog.csdn.net/yatmingli/article/details/52351058>