# Java基础

## 如何合理设置线程池的大小？

1. 首先确定是属于IO密集型还是CPU密集型,或者还是混合型的.IO密集型主要是读写的操作非常频繁,CPU密集型主要是运算操作非常频繁.混合型的自然是两者的操作都足够频繁.

2. 注意一般来说,大家都普遍认为线程池的大小应该这样设置(N为CPU的核数)

* + - 如果是CPU密集型应用,则线程池大小设置为N+1
    - 如果是IO密集型应用，则线程池大小设置为2N+1

如果一台服务器上只部署这一个应用并且只有这一个线程池，那么这种估算或许合理，具体还需自行测试验证。所以这种方式存在局限性和不准确性的.

1. 但是，IO优化中，这样的估算公式可能更适合：

* 最佳线程数目 = （（线程等待时间+线程CPU时间）/线程CPU时间 ）\* CPU数目

因为很显然，线程等待时间所占比例越高，需要越多线程。线程CPU时间所占比例越高，需要越少线程。

下面举个例子：比如平均每个线程CPU运行时间为0.5s，而线程等待时间（非CPU运行时间，比如IO）为1.5s，CPU核心数为8，那么根据上面这个公式估算得到：((0.5+1.5)/0.5)\*8=32。这个公式进一步转化为：

最佳线程数目 = （线程等待时间与线程CPU时间之比 + 1）\* CPU数目

## HashMap的实现原理?

1. 首先HashMap底层由数组+链表+红黑树组成，可接受null值，非线程安全

**put一个元素的过程**

1. 首先初始化16容量大小的数组

2. 根据key计算hashcode值,hashcode计算index下标

3. 如果多个元素出现相同的index下标,这就是hash碰撞,这时候使用到了链表

4. 先遍历当前链表的所有元素,如果key值完全一样,会替换掉新的key

5. 新插入的元素会放到链接的头部(插入快,不需要遍历节点)

2. 扩容因子,扩容机制

默认扩容因子为0.75,达到扩容因子的阈值,就进行扩容操作

# elasticsearch面试题

## elasticsearch写入数据流程?

**elasticsearch写入数据时涉及到的核心概念讲解：**

**倒排索引:**即把文件ID对应到关键词的映射转换为关键词到文件ID的映射，每个关键词都对应着一系列的文件，这些文件中都出现这个关键词。

       得到倒排索引的结构如下：

       “关键词1”：“文档1”的ID，“文档2”的ID，…………。

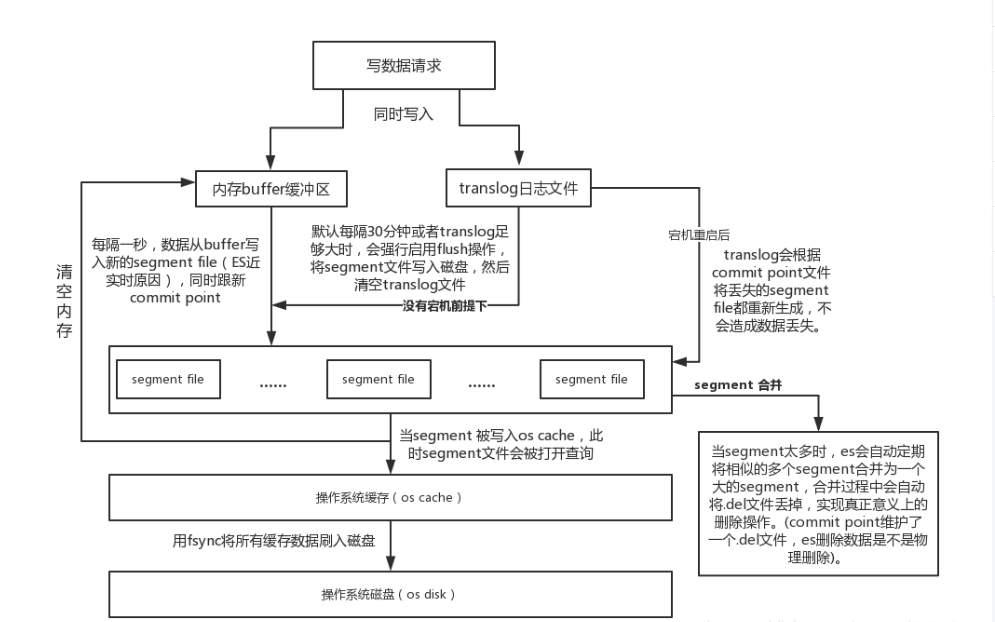
       “关键词2”：带有此关键词的文档ID列表。

**segment file:** 存储倒排索引的文件，每个segment本质上就是一个倒排索引，每秒都会生成一个segment文件，当文件过多时es会自动进行segment merge（合并文件），合并时会同时将已经标注删除的文档物理删除；

**commit point（重点理解）:** 记录当前所有可用的segment，每个commit point都会维护一个.del文件（es删除数据本质是不属于物理删除），当es做删改操作时首先会在.del文件中声明某个document已经被删除，文件内记录了在某个segment内某个文档已经被删除，当查询请求过来时在segment中被删除的文件是能够查出来的，但是当返回结果时会根据commit point维护的那个.del文件把已经删除的文档过滤掉；

**translog日志文件:** 为了防止elasticsearch宕机造成数据丢失保证可靠存储，es会将每次写入数据同时写到translog日志中(图中会有详解)。

**完整elasticsearch的写入数据流程如下：**



补充：将es中比较困惑的几个概念简单总结一下，这三种操作对理解es底层原理和优化很有帮助！

**refresh**

es接收数据请求时先存入内存中，默认每隔一秒会从内存buffer中将数据写入filesystem cache，这个过程叫做refresh；

**fsync**

translog会每隔5秒或者在一个变更请求完成之后执行一次fsync操作，将translog从缓存刷入磁盘，这个操作比较耗时，如果对数据一致性要求不是跟高时建议将索引改为异步，如果节点宕机时会有5秒数据丢失;

**flush**

es默认每隔30分钟会将filesystem cache中的数据刷入磁盘同时清空translog日志文件，这个过程叫做flush。

# Redis面试题

## 什么叫redis缓存穿透?解决方案?

**缓存穿透**

查询一个必然不存在的数据,那么请求就会穿过reids缓存,并且请求数据库返回空,如果大批量的请求都是查询必然不存在的数据,那么这些请求就会透过redis层去查询数据库,导致后台压力过大.

**解决方案**

**方案一 缓存层缓存空值**  
–缓存太多空值，占用更多空间。（优化：给个空值过期时间）   
–存储层更新代码了，缓存层还是空值。（优化：后台设置时主动删除空值，并缓存把值进去）

**方案二** **使用布隆过滤器**

将数据库中所有的查询条件，放到布隆过滤器中。当一个查询请求来临的时候，先经过布隆过滤器进行检查，如果请求存在这个条件中，那么继续执行，如果不在，直接丢弃。

将所有可能存在的数据缓存放到布隆过滤器中，当黑客访问不存在的缓存时迅速返回避免缓存及DB挂掉.

|  |
| --- |
|  |

## 什么叫redis缓存雪崩?解决方案?

**缓存雪崩**

是指在某一个时间段，缓存集中过期失效。产生雪崩的原因之一，比如在写本文的时候，马上就要到双十二零点，很快就会迎来一波抢购，这波商品时间比较集中的放入了缓存，假设缓存一个小时。那么到了凌晨一点钟的时候，这批商品的缓存就都过期了。而对这批商品的访问查询，都落到了数据库上，对于数据库而言，就会产生周期性的压力波峰。

**解决方案**

小编在做电商项目的时候，一般是采取不同分类商品，缓存不同周期。在同一分类中的商品，加上一个随机因子。这样能尽可能分散缓存过期时间，而且，热门类目的商品缓存时间长一些，冷门类目的商品缓存时间短一些，也能节省缓存服务的资源。

其实集中过期，倒不是非常致命，比较致命的缓存雪崩，是缓存服务器某个节点宕机或断网。因为自然形成的缓存雪崩，一定是在某个时间段集中创建缓存，那么那个时候数据库能顶住压力，这个时候，数据库也是可以顶住压力的。无非就是对数据库产生周期性的压力而已。而缓存服务节点的宕机，对数据库服务器造成的压力是不可预知的，很有可能瞬间就把数据库压垮。

## 什么叫redis缓存击穿?解决方案?

**缓存击穿**

是指一个key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开了一个洞。

小编在做电商项目的时候，把这货就成为“爆款”。

**解决方案**

其实，大多数情况下这种爆款很难对数据库服务器造成压垮性的压力。达到这个级别的公司没有几家的。所以，务实主义的小编，对主打商品都是早早的做好了准备，让缓存永不过期。即便某些商品自己发酵成了爆款，也是直接设为永不过期就好了。

大道至简，mutex key互斥锁真心用不上。

# 数据库面试题

## count(列名),count(常量),count(\*)区别?

**一、从执行效果来看**

1. count(1) and count(\*)：

基本没差别

count(\*)包括了所有的列，相当于行数，在统计结果的时候，不会忽略NULL

2. count(1) and count(列名)：

（1） count(1) 会统计表中的所有的记录数，不会忽略NULL，包含字段为null 的记录。

（2） count(列名) 会统计该列字段在表中出现的次数，会忽略字段为null 的情况，即不统计字段为null 的记录。

**二、从执行效率来看**

若列名为主键，count(列名)会比count(1)快

若列名不为主键，count(1)会比count(列名)快

若表多个列并且没有主键，则 count（1） 的执行效率优于 count（\*）

若表有主键，则 select count（主键）的执行效率是最优的

若表只有一个字段，则 select count（\*）最优。

所以实际业务中一般用count(1)比较普遍，但是如果需要聚合多个列，则用count(列名)比较合适。

# 分布式系统面试题

## 分布式系统唯一ID生成方案

****1. 数据库自增长序列或字段****

最常见的方式。利用数据库，全数据库唯一。

优点：

1）简单，代码方便，性能可以接受。

2）数字ID天然排序，对分页或者需要排序的结果很有帮助。

缺点：

1）不同数据库语法和实现不同，数据库迁移的时候或多数据库版本支持的时候需要处理。

2）在单个数据库或读写分离或一主多从的情况下，只有一个主库可以生成。有单点故障的风险。

3）在性能达不到要求的情况下，比较难于扩展。

4）如果遇见多个系统需要合并或者涉及到数据迁移会相当痛苦。

5）分表分库的时候会有麻烦。

优化方案：

1. 针对主库单点，如果有多个Master库，则每个Master库设置的起始数字不一样，步长一样，可以是Master的个数。比如：Master1 生成的是 1，4，7，10，Master2生成的是2,5,8,11 Master3生成的是 3,6,9,12。这样就可以有效生成集群中的唯一ID，也可以大大降低ID生成数据库操作的负载。

****2. UUID****

常见的方式。可以利用数据库也可以利用程序生成，一般来说全球唯一。

优点：

1）简单，代码方便。

2）生成ID性能非常好，基本不会有性能问题。

3）全球唯一，在遇见数据迁移，系统数据合并，或者数据库变更等情况下，可以从容应对。

缺点：

1）没有排序，无法保证趋势递增。

2）UUID往往是使用字符串存储，查询的效率比较低。

3）存储空间比较大，如果是海量数据库，就需要考虑存储量的问题。

4）传输数据量大

5）不可读。

****4. Redis生成ID****

当使用数据库来生成ID性能不够要求的时候，我们可以尝试使用Redis来生成ID。这主要依赖于Redis是单线程的，所以也可以用生成全局唯一的ID。可以用Redis的原子操作 INCR和INCRBY来实现。

可以使用Redis集群来获取更高的吞吐量。假如一个集群中有5台Redis。可以初始化每台Redis的值分别是1,2,3,4,5，然后步长都是5。各个Redis生成的ID为：

A：1,6,11,16,21

B：2,7,12,17,22

C：3,8,13,18,23

D：4,9,14,19,24

E：5,10,15,20,25

这个，随便负载到哪个机确定好，未来很难做修改。但是3-5台服务器基本能够满足器上，都可以获得不同的ID。但是步长和初始值一定需要事先需要了。使用Redis集群也可以方式单点故障的问题。

另外，比较适合使用Redis来生成每天从0开始的流水号。比如订单号=日期+当日自增长号。可以每天在Redis中生成一个Key，使用INCR进行累加。

优点：

1）不依赖于数据库，灵活方便，且性能优于数据库。

2）数字ID天然排序，对分页或者需要排序的结果很有帮助。

缺点：

1）如果系统中没有Redis，还需要引入新的组件，增加系统复杂度。

2）需要编码和配置的工作量比较大。

****5.SnowFlake雪花算法****

雪花ID生成的是一个64位的二进制正整数，然后转换成10进制的数。64位二进制数由如下部分组成：

**snowflake id生成规则**

1位标识符：始终是0，由于long基本类型在Java中是带符号的，最高位是符号位，正数是0，负数是1，所以id一般是正数，最高位是0。

41位时间戳：41位时间截不是存储当前时间的时间截，而是存储时间截的差值（当前时间截 - 开始时间截 )得到的值，这里的的开始时间截，一般是我们的id生成器开始使用的时间，由我们程序来指定的。

10位机器标识码：可以部署在1024个节点，如果机器分机房（IDC）部署，这10位可以由 5位机房ID + 5位机器ID 组成。

12位序列：毫秒内的计数，12位的计数顺序号支持每个节点每毫秒(同一机器，同一时间截)产生4096个ID序号

**优点**

简单高效，生成速度快。

时间戳在高位，自增序列在低位，整个ID是趋势递增的，按照时间有序递增。

灵活度高，可以根据业务需求，调整bit位的划分，满足不同的需求。

**缺点**

依赖机器的时钟，如果服务器时钟回拨，会导致重复ID生成。

在分布式环境上，每个服务器的时钟不可能完全同步，有时会出现不是全局递增的情况.

|  |
| --- |
| /\*\*  \* Twitter\_Snowflake<br>  \* SnowFlake的结构如下(每部分用-分开):<br>  \* 0 - 0000000000 0000000000 0000000000 0000000000 0 - 00000 - 00000 - 000000000000 <br>  \* 1位标识，由于long基本类型在Java中是带符号的，最高位是符号位，正数是0，负数是1，所以id一般是正数，最高位是0<br>  \* 41位时间截(毫秒级)，注意，41位时间截不是存储当前时间的时间截，而是存储时间截的差值（当前时间截 - 开始时间截)  \* 得到的值），这里的的开始时间截，一般是我们的id生成器开始使用的时间，由我们程序来指定的（如下下面程序IdWorker类的startTime属性）。  \* 41位的时间截，可以使用69年，年T = (1L << 41) / (1000L \* 60 \* 60 \* 24 \* 365) = 69<br>  \* 10位的数据机器位，可以部署在1024个节点，包括5位datacenterId和5位workerId<br>  \* 12位序列，毫秒内的计数，12位的计数顺序号支持每个节点每毫秒(同一机器，同一时间截)产生4096个ID序号<br>  \* 加起来刚好64位，为一个Long型。<br>  \* SnowFlake的优点是，整体上按照时间自增排序，并且整个分布式系统内不会产生ID碰撞(由数据中心ID和机器ID作区分)，并且效率较高，  \* 经测试，SnowFlake每秒能够产生26万ID左右。  \*/public class SnowflakeIdWorker {  // ==============================Fields===========================================  /\*\* 开始时间截 (2015-01-01) \*/  private final long twepoch = 1420041600000L;  /\*\* 机器id所占的位数 \*/  private final long workerIdBits = 5L;  /\*\* 数据标识id所占的位数 \*/  private final long datacenterIdBits = 5L;  /\*\* 支持的最大机器id，结果是31 (这个移位算法可以很快的计算出几位二进制数所能表示的最大十进制数) \*/  private final long maxWorkerId = -1L ^ (-1L << workerIdBits);  /\*\* 支持的最大数据标识id，结果是31 \*/  private final long maxDatacenterId = -1L ^ (-1L << datacenterIdBits);  /\*\* 序列在id中占的位数 \*/  private final long sequenceBits = 12L;  /\*\* 机器ID向左移12位 \*/  private final long workerIdShift = sequenceBits;  /\*\* 数据标识id向左移17位(12+5) \*/  private final long datacenterIdShift = sequenceBits + workerIdBits;  /\*\* 时间截向左移22位(5+5+12) \*/  private final long timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits;  /\*\* 生成序列的掩码，这里为4095 (0b111111111111=0xfff=4095) \*/  private final long sequenceMask = -1L ^ (-1L << sequenceBits);  /\*\* 工作机器ID(0~31) \*/  private long workerId;  /\*\* 数据中心ID(0~31) \*/  private long datacenterId;  /\*\* 毫秒内序列(0~4095) \*/  private long sequence = 0L;  /\*\* 上次生成ID的时间截 \*/  private long lastTimestamp = -1L;  //==============================Constructors=====================================  /\*\*  \* 构造函数  \* @param workerId 工作ID (0~31)  \* @param datacenterId 数据中心ID (0~31)  \*/  public SnowflakeIdWorker(long workerId, long datacenterId) {  if (workerId > maxWorkerId || workerId < 0) {  throw new IllegalArgumentException(String.format("worker Id can't be greater than %d or less than 0", maxWorkerId));  }  if (datacenterId > maxDatacenterId || datacenterId < 0) {  throw new IllegalArgumentException(String.format("datacenter Id can't be greater than %d or less than 0", maxDatacenterId));  }  this.workerId = workerId;  this.datacenterId = datacenterId;  }  // ==============================Methods==========================================  /\*\*  \* 获得下一个ID (该方法是线程安全的)  \* @return SnowflakeId  \*/  public synchronized long nextId() {  long timestamp = timeGen();  //如果当前时间小于上一次ID生成的时间戳，说明系统时钟回退过这个时候应当抛出异常  if (timestamp < lastTimestamp) {  throw new RuntimeException(  String.format("Clock moved backwards. Refusing to generate id for %d milliseconds", lastTimestamp - timestamp));  }  //如果是同一时间生成的，则进行毫秒内序列  if (lastTimestamp == timestamp) {  sequence = (sequence + 1) & sequenceMask;  //毫秒内序列溢出  if (sequence == 0) {  //阻塞到下一个毫秒,获得新的时间戳  timestamp = tilNextMillis(lastTimestamp);  }  }  //时间戳改变，毫秒内序列重置  else {  sequence = 0L;  }  //上次生成ID的时间截  lastTimestamp = timestamp;  //移位并通过或运算拼到一起组成64位的ID  return ((timestamp - twepoch) << timestampLeftShift) //  | (datacenterId << datacenterIdShift) //  | (workerId << workerIdShift) //  | sequence;  }  /\*\*  \* 阻塞到下一个毫秒，直到获得新的时间戳  \* @param lastTimestamp 上次生成ID的时间截  \* @return 当前时间戳  \*/  protected long tilNextMillis(long lastTimestamp) {  long timestamp = timeGen();  while (timestamp <= lastTimestamp) {  timestamp = timeGen();  }  return timestamp;  }  /\*\*  \* 返回以毫秒为单位的当前时间  \* @return 当前时间(毫秒)  \*/  protected long timeGen() {  return System.currentTimeMillis();  }  //==============================Test=============================================  /\*\* 测试 \*/  public static void main(String[] args) {  SnowflakeIdWorker idWorker = new SnowflakeIdWorker(0, 0);  for (int i = 0; i < 1000; i++) {  long id = idWorker.nextId();  System.out.println(Long.toBinaryString(id));  System.out.println(id);  }  }} |