Questions

JAVA

1. 类加载机制原理
    - 类加载过程

      .java 文件经过javac 编译成 .class文件，然后经过类加载器加载到JVM的内存。class字节码里面存储类名称的全限定符，找到这个类的字节码文件，经过验证(文件格式验证、字节码验证、符号引用验证)、准备(静态常量赋**默认的初始值**)、解析(符号引用解析成直接引用)、最终初始化（程序设置的初始值）
    - 类加载器

      主要三类：启动类加载器(bootstrap)、扩展类加载器(extension)、应用类加载器(application)，自定义类加载器。自定义类加载器一般是继承ClassLoader类，然后通过组合模式声明URLClassLoader，实现findClass()方法。
    - 双亲委派模型

      优先级层次关系，避免类的重复加载；JDK lib/目录下核心API 的类由启动类加载器加载，其他类加载器加载抛出SecurityException，具有安全性。

      加载详细过程：loadClass方法先从当前类加载器的缓存（JVM内存模型中的方法区）中查找Class对象，找到则不加载。找不到，则委托给父类加载器调用loadClass方法加载，如果父类加载器也加载不了，最终调用自己的findClass方法加载（参考URLClassLoader源码）。
    - 热部署原理

      JAVA对象的唯一性：类的全限定名+当前的类加载器。同一个Class文件由多个类加载器加载，即热部署，调用loadClass方法会首先检查类是否缓存，或已经被加载，而直接调用findClass可以绕过检查，从而实现一个Class文件被多个类加载器加载。
    - 线程上下文加载器

      核心类是由启动类加载器加载的，即SPI中定义的一些接口由启动类加载器加载，而SPI接口的具体的实现是由第三方提供，而loadClass方法是按照双亲委派模型来加载类的，比如说：`this.getClass().getClassLoder()`当前执行环境是在启动类加载器中，因此就无法加载这些第三方提供的类（根据名称检查）。因此，就出现了线程上下文类加载器，当前执行线程设置其上下文类加载器为自定义的类加载器(默认为AppClassLoader)，就可以顺利地将第三方类加载到JVM了。

2. ThreadLocal 原理

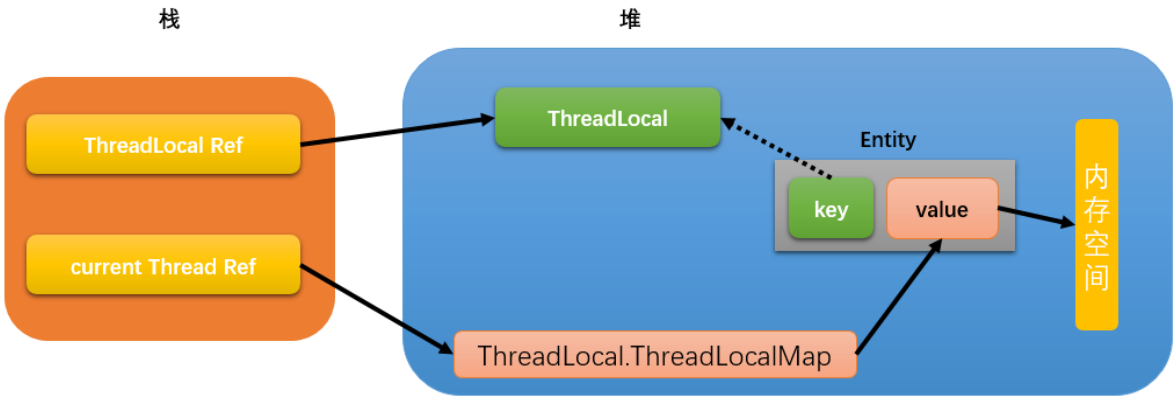
   ThreadLocal用来解决线程之间的数据隔离，又能方便在本线程之中数据共享（也可通过扩展Runnable接口定义实例变量来实现本线程中的数据共享访问---线程的实例变量，当Runnable类中的各个方法当然能够访问了）。每个线程实例有一个ThreadLocalMap属性，它是一个HashMap，Key是持有ThreadLocal对象的WeakReference引用，Value是线程想要保存的数据。

   线程通过ThreadLocal get方法获取数据时，首先Thread.currentThread()获取当前线程，进而获取线程的ThreadLocalMap实例，然后以threadLocal对象(this参数)作为key，查找ThreadLocalMap获取该线程保存的数据。

   ThreadLocal存在内存泄漏：由于ThreadLocal对象是ThreadLocalMap的Entry的key，而数据是Entry的value。key是`WeakReference<ThreadLocal>`由弱引用持有，当ThreadLocal对象释放后，value指向的真正数据并没有立即释放，从而导致内存泄漏。

要解决内存泄漏问题，在每次使用完ThreadLocal获取数据后，再调用remove方法（这种做法限制了业务使用场景）。

WeakReference与WeakHashMap。当没有强引用引用对象，只有弱引用引用该对象时，该对象可能会被GC掉。

- 每个Thread对象都有一个ThreadLocalMap实例，key是ThreadLocal，value是数据。这样，每个线程可以使用多个的ThreadLocal对象来保存多个数据。
- ```
  //java.lang.ThreadLocal#get
  //栈帧中的局部变量表的第0个slot保存了该方法所属对象的实例引用，即：ThreadLocal对象
  public T get() {
          Thread t = Thread.currentThread();//获取当前线程
          ThreadLocalMap map = getMap(t);//获取该线程的 ThreadLocalMap 实例
          if (map != null) {
              ThreadLocalMap.Entry e = map.getEntry(this);//ThreadLocal对象作为key查找
              T result = (T)e.value;//获得 数据
  ```
- 

-

3. 自增原子性：volatile，i++，AtomicLong，LongAddr。在多线程竞争环境下，LongAddr 比 AtomicLong 更有效率。
4. xxx

JVM 和 GC

1. 什么时候触发GC?

   JVM堆内存结构：新生代和老年代。新生代有Eden区和两个Survivor区，当Eden区和其中一个Survivor区无法容纳新对象时，触发Minor GC。

   新生代的对象有年龄，当年龄到达15时晋升到老年代。老年代没有足够空间容纳晋升的对象时，触发Major GC。

   这种触发GC的方法叫做Allocation Failure（分配失败）

   还有一种触发GC的方式是：GC Locker：
  ```

另外，触发GC事件与特定的垃圾回收器有关，以CMS为例：有一个参数`XX:CMSInitiatingOccupancyFraction=75`当老年代使用量超过75%时触发GC。CMS也会预判当新生代对象不能顺利晋升到老年代时，会提前触发老年代GC。

2. 什么时候触发OOM?

   对于Parallel垃圾收集器（相对于The Mostly Concurrent Collectors而言）当使用了98%的时间却回收了不到2%的堆内存时，抛出OOM Error。

3.

多线程和线程池

1. 为什么使用线程池?

   线程复用。用线程池来管理线程比手工new Thread 再调用start启动线程的好处实在是太多了.....

2. 线程池的实现方式?

   自定义线程池从继承ThreadPoolExecutor开始，指定参数：core pool size，max pool size，工作队列(SynchronousQueue、LinkedBlockquque/LinkedTransferQueue、ArrayBlockingQueue)、线程工厂、拒绝策略。

   pool size决定线程的数量，这个可根据任务类型来确定。线程数量公式(感觉也是基于Little's Law) $N_{cpu} * U_{cpu} * (1 + \frac{W}{C})$。或者，CPU密集型作业$N_{cpu} + 1$，IO密集型作业$2 * N_{cpu} + 1$。

   定义工作队列需要确定队列的长度，单从任务的角度来考虑：可基于Little's Law确定，$任务队列长度 = 任务的到达速率 * 任务的响应时间$。任务的响应时间分成2部分：队列中的等待时间、执行时间。

   而在实际的系统中，不仅仅需要考虑 任务的提交/到达速度，也需要考虑服务器的处理能力。而服务器的处理能力，又影响了任务的响应时间。比如10000QPS的服务器，处理每个任务耗时0.1s，任何时刻该服务器只能承担10000*0.1=1000个任务。

3.

Bloom Filter

优势：

1. 空间优势（相比于HashMap），采用**多个**哈希函数将元素是否存在映射成位操作（有点类似时间换空间--只能判断元素是否存在，不能根据key获取value）
2. 判断海量数据。解决 HashMap 大数据量的冲突严重问题
3. 非常适用于判断，某个元素不在某个大集合中

算法原理：

1. 两个待指定的参数：待写入的数据量n、误报的概率p
2. 一个默认参数：murmur3_128 哈希策略
3. 已知待写入的数据量n，和期望的误报概率p，**如何确定bloom filter的长度?**

   Guava计算Array数组位数（底层是个long类型数组而不是BitSet）：n是数据量、p是误报率。

   $m = \frac{-n*log(p)}{log2*log2}$

底层数组长度m 远大于 数据量n。

4. 计算哈希函数的个数：n是数据量，m是上一步计算出来的位数

   $k = max(1, \frac{m}{n} * log2)$

5. 对于一个元素e，多少位会被置为1？k哈希函数个数，m是数组的长度

   k个位。对每个哈希函数，$index = hash_i(e)$，$0 \leq index \leq m-1$，将对应位置为1：Array[index]=1

6. 什么样的哈希函数适合作为bloom filter的hash函数?

   独立同分布、尽可能地快（不要选用加密类型的hash函数，比如SHA1）

7. **bloom filter的长度如何确定？哈希函数的个数如何确定?**

   n待写入的数据量，p是误报率(false negative)，m是bloom filter的长度，k是哈希函数的个数，公式：

   $p = (1 - e^{\frac{-kn}{m}})^k$

   根据待写入的数据量n和误报率p，二者来调整m和k。**那么到底如何选择bloom filter的长度?**

   第一步：确定待写入的数据量n

   第二步：大概选择bloom filter的长度m

   第三步：根据公式$\frac{m}{n} * ln2$地计算最优的哈希函数个数k

   第四步：根据公式$p = (1 - e^{\frac{-kn}{m}})^k$计算误报率，如果这个误报率可接受，则结束，否则回到第二步重新计算。

   在Google Guava中，只需要指定待写入的数据量n和误报率p，哈希函数个数和bloom filter长度会自动计算出来。

8. **哈希函数的个数对bloom filter的影响是什么?**

   哈希函数的个数越多，bloom filter越慢，因为要执行多次哈希映射。但是，哈希函数越少，元素被hash后标记为1的位数就越少，也即越容易冲突，从而误报率会上升。

   在给定bloom filter的长度m 和 待写入的数据量n 时，哈希函数的个数k的最优值是$\frac{m}{n} * ln2$

9. **bloom filter的时空复杂度?**

   时间复杂度：添加元素和测试一个元素是否在bloom filter的时间复杂度都是O(k)，k为哈希函数的个数。因为，添加元素就是将相应位设置为1，而测试元素是否存在，也是看相应位是否为1

   空间复杂度：不好说，与期望的误报率p有关，误报率越小，空间复杂度越高。也与待写入的数据量有关，待写入的数据量越大，空间复杂度越高。

10. xxx

skip list（跳跃表）

1. JDK包：java.util.concurrent.ConcurrentSkipListMap 和 Redis的 sorted list都采用了skip list。skip list与红黑树的各个操作的时间复杂度都一样：插入、删除、查找都是：O(logN)，但红黑树有复杂的平衡操作，此外skip list更适合于并发数据结构的实现。

   > The reason is that there are no known efficient lock-free insertion and deletion algorithms for search trees.

2. 链式结构，已排序的数据分布在多层链表中，当插入节点时以一个概率值决定新节点是否晋升到高层链表中，因此有节点冗余，是一种以空间换时间的数据结构。

3. 基本特征

- Skip List由若干层链表组成，每层链表都是有序的。
- 每个元素有四个基本的指针（前、后、上、下）如果一个元素出现在第 i 层，所有比 i 小的层都会包含该元素。
- 第 i 层的元素通过一个向下的指针 指向 下一层 相同值的元素。头指针最高层的第一个元素，尾指针指向最高层最后一个元素。

4. 查找操作

   从头节点(顶层)开始，按照右指针直到右节点的值大于待查找的值。判断是否还有更低层次的链表，若有则移动到当前节点的下一层，直到最底层。如果最底层节点匹配，查找成功，否则查找失败。

5. 插入操作

   先按查找操作找到待插入的位置，更新链表节点指针插入节点。再以随机概率决定新加入节点的是否晋升到上一层链表（看论文细节）

6. skip list 的间隔和层数如何确定?

7. xxx

Elasticsearch

熟悉各个操作的**底层实现原理**、清楚该操作的基本**执行流程。**

1. Elasticsearch 数据副本模型

   primary shard 和 replica

   - in-sync 副本集合
   - 如何确定哪个分片是primary，哪些是replica?
   -
   - global checkpoint
   - local checkpoint
   - xxx

2. Elasticsearch master节点选举

   - master选举什么时候发起?

     比如当一个新节点(node.master设置为true)加入ES集群时，它会通过ZenDiscovery模块ping其他节点询问当前master，当发现超过minimum_master_nodes个节点响应都没有连接到master时，发起master选举。

     总之，当一个节点发现包括自己在内的多数派的master-eligible节点认为集群没有master时，就可以发起master选举。

   - 选举哪个节点作为master?

     在选举过程中有两个集合，一个是active masters，另一个是Master Candidates。如果Active masters不为空则从里面选择一个节点，比较节点的ID，节点ID最小的作为选出来的master。如果active masters为空，则从master Candidates里面选择节点，按cluster state最新、节点ID最小的节点作为选出来的master。

     选出来的master获得多数派的master-eligible投票后，成为真正的master。

     active masters是那些被认为是当前集群中的master的节点（节点5认为节点1是master，节点4认为节点2是master，那么节点1和节点2都将作为activate master）。而master candidates是配置文中node.master设置为true的节点。

   - 如何避免split brain?

选择出来的master由所有的master eligible node (node.master=true)投票，只有获得大多数投票的节点，最终才能成为真正的master。每个节点只能投一票，通过选举周期来区分同一节点不同阶段的投票（有待ES源码验证）。

- ES master节点选举会考虑两个因素：节点ID和集群状态(cluster state)，节点ID是在集群启动时随机生成的并且会持久化，ES倾向于将节点ID最小的那个节点选为master，这与Bully算法很相似。而最新的cluster state version保证选出的master能够知道最新数据的分布(比如哪个shard拥有最新写入的文档)

3. Elasticsearch 索引(index)机制
   - translog机制

     每次文档的更新/写入/删除等操作都提交给Lucene生效代价很大（Lucene commit），势必影响吞吐量。

     > Changes to Lucene are only persisted to disk during a Lucene commit, which is a relatively expensive operation and so cannot be performed after every index or delete operation.

     Internal Lucene index处理完Index/delete/update操作之后 **写Translog 之后才给Client acknowledged确认。**但并不会立即执行Lucene commit

     > All index and delete operations are written to the translog after being processed by the internal Lucene index but before they are acknowledged.

     因此，故障之后Shard恢复时，可以从Translog恢复。

     > In the event of a crash, recent transactions that have been acknowledged but not yet included in the last Lucene commit can instead be recovered from the translog when the shard recovers

   - translog 配置参数

     `index.translog.sync_interval` tranlog 异步刷新到磁盘，这是**translog的提交。**不管有没有写操作，默认每5s写磁盘一次。

     `index.translog.durability` 默认配置为request，即：在有写操作(index、delete、update)时，每次操作之后translog都要刷新到磁盘。

     结合上面2个配置参数：在没有请求时，translog每5s秒刷新到磁盘，在有操作时，则是每次操作都刷新到磁盘。

     `index.translog.flush_threshold_size` 为了防止translog刷新到磁盘之后translog过大，当translog到达512MB时，触发Lucene commit，同时清空该translog

     > Once the maximum size has been reached a flush will happen, generating a new Lucene commit point. Defaults to `512mb`.

   - xxx

4. Elasticsearch分片分配原理

   分片分配主要有2个过程：找出待分配的最佳节点，决定是否将分片分配到该节点上，分配决策由主节点完成。主要有2个问题：

   - 哪些分片分配给哪些节点?

     AllocationService.reroute()

     SameShardAllocationDecider

- 哪个分片作为主分片，哪些作为副本?

  同步副本集合in-sync list

  PrimaryShardAllocator

  ReplicaShardAllocator

- xxx

5. Elasticsearch文档路由原理

   文档如何分配给分片？根据document id 和 routing 参数计算 shard id 的过程。

   > shard_num = hash(_routing) % num_primary_shards

   默认情况下文档id就是路由参数_routing，这个公式直接将文档路由（采用murmur3哈希函数）到某个具体的分片上。

   当采用自定义路由参数时，为了避免数据分布不均匀，引入了routing_partition_size参数，先将文档路由到一组分片上，然后再从这组分片里面选择一个分片存储文档。

   > shard_num = (hash(*routing)* + *hash(*id) % routing_partition_size) % num_primary_shards

   在源码中：num_primary_shards其实是 routingNumShards（参数路由的分片数量）“并不是”定义索引时指定的分片数量。这是为了支持shrink操作。

   哈希计算出分片后，查找RoutingTable得到该分片所在的节点地址，然后将文档发送到该节点上。

6. Elasticsearch写操作

   - docId 是如何自动生成的?
   - doc--->in memroy buffer--->refresh segment--->commit disk。写操作首先将文档保存到内存，默认1s refresh 成为segment，segment是可被搜索的，然后是默认30min flush到磁盘（Lucene commit）。refresh API是将in memory buffer 刷新成Segment，flush API 则是将各个内存中的小段合并成大段，进行Lucene提交。同时，translog过大，也会导致Lucene提交。
   -

   写blog记录理解org.apache.lucene.util.SetOnce，如何实现只允许一次修改，多次读取的场景？

7. ElasticSearch查询原理（Search操作）
8. ElasticSearch GET操作

   GET 根据docId获取文档。先将docId转换成分片id，然后发送到相应的节点上获取文档。

   GET 操作默认是实时的(realtime=true)

   > Realtime GET support allows to get a document once indexed regardless of the "refresh rate" of the index. It is enabled by default.

   并可以指定refresh参数(默认为false)。也即：当成功index一篇文档后，GET能获取该文档，但search却不一定能搜索到。Search的**可见性**依赖于refresh。

   > When a document is indexed, its indexed, its not "soon to be indexed". When it becomes visible for search is the question (and thats the async refresh). Fetch by Id will work even if it has not been refreshed yet.

9. xxx

Kafka

1，如何保证Kafka中的消息不丢失？（可靠性）

涉及到三个方面：生产者、Kafka本身、消费者。

生产者方面涉及到2个参数：ack和retries。生产者在发送消息时可能遇到错误，对于可重试的错误，retries设置了一个值，生产者自动重新发送(默认间隔100ms)，直至达到最大可重试次数，返回失败。(消息的丢失可能发生在生产者端)

生产者ACK参数指定需要有多少个副本成功写入消息，才认为消息写入成功。ACK可配置为0，1，All。ack=1时意味着首领副本写入成功，就返回ack给生产者。而ack=all，意味着当消息写入到**所有的同步副本**中，则返回ack。

而对于Kafka本身，是个集群，数据是有副本备份的。每个topic有多个分区，每个分区配置若干个副本。副本有2种类型：首领副本(leader replica)和跟随者副本(follow replica)。broker接收到生产者发送的消息后**先写入首领副本**，然后将消息同步给跟随者副本（副本数量默认为3），请求得到最新消息的副本称为同步副本(显然首领副本肯定是个同步副本)，同步副本的个数由参数min.insync.replicas决定。**只有当消息都写入到了同步副本中才认为这条消息是已提交的，只有已提交了的消息才能被Kafka消费者消费。**（更严谨地说：当消息都写入到了min.insync.replicas个同步副本中，才认为这条消息是已提交的）

消息丢失的一个示例：

生产者ack=1，min.sync.replicas=1。生产者将消息发送给首领副本，首领副本写入成功后即可返回ACK。但是首领副本还未来得及将消息同步到其他副本（**其他副本仍然认为是同步的**，因为判定副本不同步需要一段时间）就宕机了，但消费者收到了ACK，认为此消息已成功发送。因此，这条消息就丢失了。因此，**当 min.sync.replicas=1 时，无法避免消息的丢失。**因此，为了保证消息的可靠性，ACK=all且min.insync.replicas大于1。

不完全的Leader副本选举：

允许将非同步副本作为首领副本，称为不完全的Leader选举。比如3个副本，首领副本在一直接收生产者写入的消息，因网络问题其他两个副本已经与首领副本不同步了，若此时首领副本所在节点宕机，controller会从其他2个副本中选择一个作为新的首领副本。不完全的首领副本选举会导致数据丢失，而禁用不完成Leader副本选举会造成服务不可用，因为生产者发送的消息必须先写入首领副本，然后再同步给其他副本，如果首领副本挂了，生产者就不能写入消息了。

如何判断同步副本？

- 与zk之间有一个活跃的会话(过去6s内向zk发送过心跳)
- 过去10s内从首领副本那里同步过消息

而对于消费者而言，必须等到消息成功处理后（比如已经写入到其他第三方存储ES）才能向broker提交消息确认（比如采用同步提交方式）。

总之，Kafka的消息可靠性保证是由生产者、Kafka、消费者三者共同保证的。其他任何一个方面有问题，都有可能导致消息丢失。

2，如何保证Kafka消息的重复性问题？

相比于可靠性问题，重复性问题主要体现在消费者端，比如消息者处理了一批消息之后未来得及提交就挂掉了，那么就可能收到重复的消息。这个时候，如果消息处理具有幂等性，那就无须过多处理，或者消息者能够依据外部系统（比如将消息写入ES，如果消息的唯一ID作为ES的文档ID，那么如果ES里面有这条消息，那就意味着此消息已经被处理了，就不需要处理）判断重复性。

机器学习

1. 线性回归与逻辑回归的区别?
2. 朴素贝叶斯假设?
3.

有用的参考:

Google Guava之BloomFilter源码分析及基于Redis的重构

bloomfilter-tutorial

Java GC Causes Distilled

抛出OOM Error 的8个症状【已下载PDF版本】

跳跃表Skip List的原理和实现(Java) 有图解

SkipList的那点事儿实现参考

《skiplist a probabilistic alternative to balanced trees》

Realtime GET #1060

elasticsearch-realtime-get-support

ElasticSearch master 选举

Bully Election Algorithm Example

深入理解ThreadLocal

深入理解类加载器

真正理解线程上下文类加载器