快手面经

1. Spring 原理，Spring IOC、AOP

这个问题最好可以多说一点，比如对于 IOC，不妨把 Bean 如何加载、如何初始化以及如何注册到 IOC 容器中的详细过程说一下，涉及 BeanDefinition、BeanFactory 也深入细节聊一下。

2. 一个请求过来在 Spring 中发生了哪些事情？

这个问题不妨把一个请求过来 在 TCP 层面上建立连接、操作系统如何处理连接、Web 容器器接收到连接对象后做了哪些事情、Spring 如何对接收到的请求进行处理都说一下，当然最终还是 落在 Spring 容器内部如何处理一个请求，这个过程一定要说清楚，需要体现细节。在说前面的内容的时候，可以放心面试官不会打断你。

3. 手写一个单例

这个基本上大多数公司都会考察的。要写一个 基于懒汉式的 双重检测的单例。单例有三个比较关键的点：

- 私有构造方法，避免外部 new 出对象；
- 保证唯一性；
- 提供一个全局访问点。

另外懒汉式双重检测的实现方式有三点需要注意的地方：

- 全局访问点必须是静态的，外界使用可以通过类直接调用；

- 在进入锁之后还需要校验；
- 保存单例例对象的私有变量一定要用 volatile 修饰，这个地方 可以多说一些，比如 volatile 防止指令重排序，保证内存可见性(JVM 层面 和 CPU 层面 可以分别说)。volatile 这个地方能说的东西还是很多的，基本上可以与面试官再聊二十分钟了。

4. HashMap

对于 HashMap 其实一般高级岗位及以上不再会问这个东西了，一旦问了，肯定不是让你只说一下数组+链表的。对于它的实现，不同版本实现方式不一样。

在 jdk1.8 之后，HashMap 除了数组+链表之外，引入了红黑树。那么好了，你需要说明对于引入了红黑树的 HashMap 如何 put 一个元素，以及链表是在何时转化为红黑树的。比如首先需要知道这个元素落在哪一个数组里，获取 hashcode 后并不是对数组长度取余来确定的，而是高低位异或求与来得到的。这个地方首先得知道 异或、与是做什么样的运算的，然后说一下在 HashMap 中的实现，比如 hashcode 无符号右移 16 位后和原 hashcode 做异或运算，这相当于把 hashcode 的高 16 位拿过来 和 hashcode 的 低 16 位 做异或运算，因为无符号右移后 前面高 16 位都补零，这就是前面说的 "高低位异或 “，进而是 ” 求与 “，和谁求与呢，和 数组长度减 1 求与。

说到这里起码能够证明你是看过源码的，接下来说说你的思考。

比如 我们知道对于 HashMap 初始化容量决定了数组大小，一般我们对于数组这个初始容量的设置是有规律的，它应该是 2^n 。这个初始容量的设置影响了 HashMap 的效率，那又涉及到影响 HashMap 效率的主要因素，比如初始容量和负载因子。当已用数组

达到容量与负载因子的乘积之后会进行一个 rehash 的过程，这个地方涉及到的 如何 rehash 及各种算法如果有时间也是可以说的，没有时间不说也没有关系。回到刚才说的 2^n, 可以说说它为什么是 2^n。当我们说什么东西为什么是这样的时候，我们一般从两个角度考虑，一个是这样做有什么好处，另一个是不这样做有什么坏处。我们刚才说到”求与“这个过程，如果不是 2^n, 会导致较多的哈希碰撞(具体原因 可以自己分析一下 或者百度一下)，这个会影响 HashMap 的效率。

说完上面这些，既表明你看过源码，又表明你有自己的思考了，当然也可以进一步说说它是在什么条件下以及 如何进行扩容的（如果时间允许，并且面试官也有耐心继续听下去）。对于 put 操作，这才只是第一步，找到数组的位置，接下来 要看这个位置也没有元素，如果没有，直接放进去就可以，如果有，要看怎么放进去，jdk1.8 中 对于 HashMap 的实现中，是基于 Node(链表节点) 和 TreeNode(红黑树节点) 的，当然它们继承了 Entry。那么 如果数组当前位置已经有了元素，就得知道这个元素 是 链表的节点还是红黑树的节点，以便便 进一步确认接下来要 put 的元素 是以链表的方式插入还是以红黑树的方式插入，这个地方 在源码中 进行了一个类型的判断，如果是链表的节点，就以链表的方式把要 put 的节点插入到 next 为 null 的节点上，如果是红黑树的节点，就要以红黑树的方式插入一个节点。

5. Java1.8 为什么要引入红黑树？如何在红黑树中插入一个节点？

对于这两个问题，首先，引入红黑树的好处是为了提高查询效率，要说出 O(log2(n))，但是 在提高查找效率的同时也在插入的时候更加耗时，那可以说一下为什么更加耗时，自然带出第二个问题，如何在红黑树中插入一个节点，比如当插入一个节点的时候我们会默认

它是红色的(这个地方可以结合红黑树特点说一下我们为什么默认它是红色的，从黑色高度以及相邻两节点不同为红色入手)，插入后如果父节点是黑色的 就不需要动了了，但假如是红色的，就需要进行左旋和右旋操作，如果很了解，可以细说左旋右旋如何实现，如果不不是很了了解，到此为止也 ok。

说到这里，我们忽略略了一个重要的点，就是链表转换为红黑树的条件，说出链表长度到 8(相当于红黑树开始第四层) 以及数组大小达到 64 就已经够了了，也可以进一步说一下链表是如何转换为红黑树的。说完也可以说一下 ConcurrentHashMap 中也是一样的，然后接下来就引入对 ConcurrentHashMap 的理解，比如在什么地方会涉及到线程安全问题以及 ConcurrentHashMap 是如何解决的，说说 CAS，说完 CAS 再说说 AQS，自由发挥吧。

6. JVM 四种引用类型

这个问题比较简单，强引用、弱引用、软引用、虚引用，说一下它们各自的特点和 GC 对它们的不同处理方式，再说一下常见的应用场景 或者 jdk 的实现中对它们的使用，比如 ThreadLocal 的静态内部类 ThreadLocalMap，它的 Key 是弱引用的，也可以说一下 在你的理解中为什么它是弱引用的，假如不是会怎么样。

7. SpringBoot 启动过程

这个主要是从它基于 Spring 的事件发布和监听机制开始说起 就没什么问题。

8. 类加载过程

加载、链接、初始化，链接又分为验证准备和解析，每一个阶段是做了什么要说清楚。

Object a = new Object()； 这行代码做了了哪些事情，需要从类加载开始说起，这个相当于上面问题的延续，所以 一定要清楚 每一个环节 做了哪些事情的，否则这个问题不可能说清楚。说完类加载的过程 再说一下 开辟内存空间、初始化内存空间以及把内存地址赋值给变量 a，接下来可以进一步说一下 JVM 或者 CPU 层面对指令的优化，以及在某些时刻我们需要避免它做这样的优化，比如在单例中我们的实例需要用 volatile 修饰 避免指令重排序(可以说一下 在 new 一个对象的过程中如果指令重排序了会导致什么结果)。

9. 说说 Netty

从 NIO 开始说肯定是没错的，再说说 Netty 的实现方式，以及它除了 IO 之外还干了哪些事情。

10. 消息队列的熟练程度，比如问问 Kafka 分区，如何分区等

在 Kafka 实际生产过程中，每个 topic 都会有 多个 partitions。

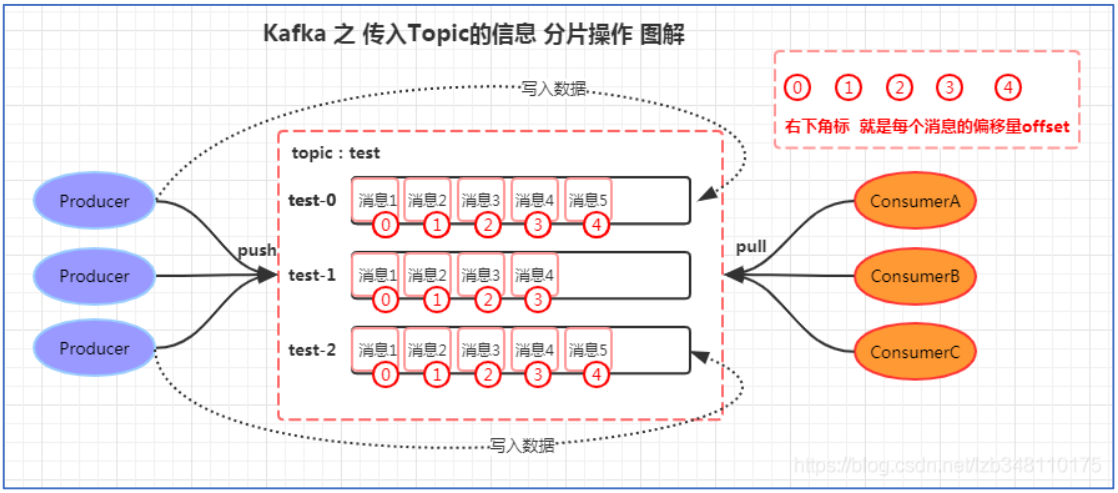
1. 多个 Partitions 有什么好处？

- 多个 partition ，能够对 broker 上的数据进行分片，通过减少消息容量来提升 IO 性能；
- 为了提高消费端的消费能力，一般情况下会通过多个 conusmer 去消费同一个 topic 中的消息，即实现消费端的负载均衡。

2. 针对多个 Partition，消费者该消费哪个分区的消息？

Kafka 存在 消费者组 group.id 的概念，组内的所有消费者协调在一起来消费订阅的 topic 中的消息（消息可能存在于多个分区中）。那么同一个 group.id 组中的 consumer 该如何去分配它消费哪个分区里的数据。

针对下图中情况，3 个分区(test-0 ~ test-3)，3 个消费者(ConsumerA ~ C)，哪个消费者应该消费哪个分区的消息呢？



对于如上这种情况，3 个分区，3 个消费者。这 3 个消费者都会分别去消费 test 中 topic 的 3 个分区，也就是每个 Consumer 会消费一个分区中的消息。

如果 4 个消费者消费 3 个分区，则会有 1 个消费者无法消费到消息；如果 2 个消费者消费 3 个分区，则会有 1 个消费者消费 2 个分区的消息。针对这种情况，分区数 和 消费者数 之间，该如何选择？此处就涉及到 Kafka 消费端的分区分配策略了。

1. 什么是分区分配策略

通过如上实例，我们能够了解到，同一个 group.id 中的消费者，对于一个 topic 中的多个 partition 中的消息消费，存在着一定的分区分配策略。

在 Kafka 中，存在着两种分区分配策略。一种是 RangeAssignor 分配策略(范围分区)，另一种是 RoundRobinAssignor 分配策略(轮询分区)。默认采用 Range 范围分区。 Kafka 提供了消费者客户端参数 partition.assignment.strategy 用来设置消费者与订阅主题之间的分区分配策略。默认情况下，此参数的值为：org.apache.kafka.clients.consumer.RangeAssignor，即采用 RangeAssignor 分配策略

RangeAssignor 范围分区

Range 范围分区策略是对每个 topic 而言的。首先对同一个 topic 里面的分区按照序号进行排序，并对消费者按照字母顺序进行排序。假如现在有 10 个分区，3 个消费者，排序后的分区将会是 0,1,2,3,4,5,6,7,8,9；消费者排序完之后将会是 C1-0,C2-0,C3-0。通过 partitions 数/consumer 数 来决定每个消费者应该消费几个分区。如果除不尽，那么前面几个消费者将会多消费 1 个分区。

例如，10/3 = 3 余 1 ，除不尽，那么 消费者 C1-0 便会多消费 1 个分区，最终分区分配结果如下：

C1-0	消费 0,1,2,3 分区
C2-0	消费 4,5,6 分区
C3-0	消费 7,8,9 分区(如果有 11 个分区的话，C1-0 将消费 0,1,2,3 分区，C2-0 将消费 4,5,6,7 分区 C3-0 将消费 8,9,10 分区)

Range 范围分区的弊端：

如上，只是针对 1 个 topic 而言，C1-0 消费者多消费 1 个分区影响不是很大。如果有 N 多个 topic，那么针对每个 topic，消费者 C1-0 都将多消费 1 个分区，topic 越多，C1-0 消费的分区会比其他消费者明显多消费 N 个分区。这就是 Range 范围分区的一个很明显的弊端了

由于 Range 范围分区存在的弊端，于是有了 RoundRobin 轮询分区策略，如下介绍↓↓↓

RoundRobinAssignor 轮询分区

RoundRobin 轮询分区策略，是把所有的 partition 和所有的 consumer 都列出来，然后按照 hascode 进行排序，最后通过轮询算法来分配 partition 给到各个消费者。

轮询分区分为如下两种情况：

- 同一消费组内所有消费者订阅的消息都是相同的。
- 同一消费者组内的消费者锁定月的消息不相同。

如果同一消费组内，所有的消费者订阅的消息都是相同的，那么 RoundRobin 策略的分区分配会是均匀的。

例如：同一消费者组中，有 3 个消费者 C0、C1 和 C2，都订阅了 2 个主题 t0 和 t1，并且每个主题都有 3 个分区(p0、p1、p2)，那么所订阅的所以分区可以标识为 t0p0、t0p1、t0p2、t1p0、t1p1、t1p2。最终分区分配结果如下：

消费者 C0	消费 t0p0 、t1p0 分区
消费者 C1	消费 t0p1 、t1p1 分区
消费者 C2	消费 t0p2 、t1p2 分区

如果同一消费者组内，所订阅的消息是不相同的，那么在执行分区分配的时候，就不是完全的轮询分配，有可能会导致分区分配的不均匀。如果某个消费者没有订阅消费组内的某个 topic，那么在分配分区的时候，此消费者将不会分配到这个 topic 的任何分区。

例如：同一消费者组中，有 3 个消费者 C0、C1 和 C2，他们共订阅了 3 个主

题：t0、t1 和 t2，这 3 个主题分别有 1、2、3 个分区(即:t0 有 1 个分区(p0)，t1 有 2 个分区(p0、p1)，t2 有 3 个分区(p0、p1、p2))，即整个消费者所订阅的所有分区可以标识为 t0p0、t1p0、t1p1、t2p0、t2p1、t2p2。具体而言，消费者 C0 订阅的是主题 t0，消费者 C1 订阅的是主题 t0 和 t1，消费者 C2 订阅的是主题 t0、t1 和 t2，最终分区分配结果如下：

消费者 C0	消费 t0p0
消费者 C1	消费 t1p0 分区
消费者 C2	消费 t1p1、t2p0、t2p1、t2p2 分区

RoundRobin 轮询分区的弊端

从如上实例，可以看到 RoundRobin 策略也并不是十分完美，这样分配其实并不是最优解，因为完全可以将分区 t1p1 分配给消费者 C1。

所以，如果想要使用 RoundRobin 轮询分区策略，必须满足如下两个条件：

- 每个消费者订阅的主题，必须是相同的。
- 每个主题的消费者实例都是相同的。(即：上面的第一种情况，才优先使用 RoundRobin 轮询分区策略)。

什么时候触发分区分配策略

当出现以下几种情况时，Kafka 会进行一次分区分配操作，即 Kafka 消费者

端的 Rebalance 操作

- 同一个 consumer 消费者组 group.id 中，新增了消费者进来，会执行 Rebalance 操作。
- 消费者离开当期所属的 consumer group 组。比如主动停机或者宕机。
- 分区数量发生变化时（即 topic 的分区数量发生变化时）。
- 消费者主动取消订阅。

Kafka 消费端的 Rebalance 机制，规定了一个 Consumer group 下的所有 consumer 如何达成一致来分配订阅 topic 的每一个分区。而具体如何执行分区策略，就是上面提到的 Range 范围分区 和 RoundRobin 轮询分区 两种内置的分区策略。

Kafka 对于分区分配策略这块，也提供了可插拔式的实现方式，除了上面两种分区分配策略外，我们也可以创建满足自己使用的分区分配策略，即：自定义分区策略。

面经资料来自网络