美团

Java基础

1. volatile关键字，线程安全怎么保证
    - 保证可见性和不可排序性。
    - 可见性：总线，嗅探机制，对数据的更新直接强制写会内存，嗅探机制使其他缓存中的该数据直接失效。
    - 插入内存屏障：不管什么指令都不能和这条memory barrier指令重排序。StoreStore，StoreLoad，LoadStore，LoadLoad。后面的v的写和前面的普通写禁止重排序，前面的v写和后面的可能的v读/写禁止重排序，前面的v读和后面的普通写禁止重排序，前面的v读和后面的读禁止重排序
2. 泛型原理，有哪些好处
    - 好处：适用于多种数据类型执行相同的代码（代码复用）
    - 实现原理：类型擦除。java语法上支持泛型，但是在编译期会进行类型擦除，将所有泛型表示都替换成具体的类型。类型擦除的原则：①会消除类型参数声明，删除<>包含的部分。②根据类型上下界推断并替换原生态类型，如果没有上下界约束，将类型擦除为object，如果有上界限制，将类型替换为上届类型，如果有下届限制，将类型替换为object。③为了保证类型安全，必要时插入强制类型转换代码。④自动产生桥接方法，以保证类型擦除后的代码仍具有泛型的多态性。
    - 如何理解泛型的编译器检查：是针对引用的，谁是一个引用，用这个引用调用泛型方法，就会对这个引用调用的泛型方法进行类型检查，而无关它真正的引用对象。
    - 如何理解泛型的多态，桥方法：泛型擦除会造成多态的冲突，jvm的解决方法就是桥方法。子类真正覆盖的是父类的两个方法是我们看不到的桥方法，桥方法内部只是去调用我们重写的那两个方法。
3. 四个引用，有啥区别，应用场景
    - 强引用，软引用，弱引用，虚引用。
    - 强引用，任何时候都不会被jvm回收，即使oom。软引用：当内存不足时，会被回收。弱引用：每次gc的时候都会被回收，不管内存是否充足。虚引用：需要和referencequeue联合使用，当jvm扫描到虚引用的对象时，会先将次对象放到关联的队列中，因此我们可以通过判断队列中是否存在该对象，来了解被引用的对象是否将要被回收，进行一些回收前的处理。
    - 使用场景：软引用：创建缓存，弱引用：WeakHashMap类中的key，可以使用WeakHashMap<String,Map<k,v>>保存事务的信息，在事务周期中，String对象的强引用一直存在，我们就一直可以获取信息，当事务结束时，弱引用可以自动帮我们清除map信息。虚引用：用于对象销毁前的一些操作，比如资源释放等。
4. JVM垃圾回收，CMS，G1区别对比
    - 判断一个对象是否会被回收：引用计数法，gc root(可达性分析算法)，方法区的回收，finalize方法
    - 新生代(伊甸区，幸存区0，幸存区1)，老年代，young gc，major gc，full gc。
    - 标记清除，标记整理，标记复制，分代收集
    - 垃圾回收器：
        - Serial垃圾回收器，只会使用一个线程进行垃圾回收，新生代：复制算法，暂停所有用户线程，老年代：标记整理，暂停所有用户线程。
        - CMS垃圾回收器：初始标记：只是简单地标记一下gc root能关联的对象，时间很短，需要停顿，并发标记：进行GC Root Tracing的过程，耗时最长，不需要停顿，重新标记：修正并发标记期间因用户程序继续运行而产生变动的那一部分对象的标记记录，需要停顿。并

发清除：不需要停顿。缺点：吞吐量低，低停顿时间是以牺牲吞吐量为代价的，导致CPU利用率不高。无法处理浮动垃圾，可能会导致Concurrent Mode Failure。浮动垃圾是指并发清除阶段用户程序继续运行而产生的，只能等到下次GC的时候才能会被回收。由于浮动垃圾的存在，老年代需要预存一段内存，CMS不能像其他垃圾回收器一样等到老年代快满时，再进行垃圾回收。如果预留的空间不足时，可能会导致Concurrent Mode Failure，将临时启用Serial Old来代替CMS。标记-清除算法导致碎片，往往出现老年代剩余，导致无法找到一块连续的足够大的内存空间存放对象，不得不提前触发一次full gc。

- G1垃圾回收器：其他垃圾回收器都是面向新生代或者老年代，G1可以直接对新生代和老年代一起回收。G1把堆分成多个大小相等的独立区域，新生代和老年代不再物理隔离。对每一个小块进行单独的垃圾回收，这种划分方法带来很大的灵活性，使得可预测的停顿时间模型成为可能。通过记录每个Region垃圾回收时间以及回收可获得的空间，并维护一个优先列表，每次根据允许的收集时间，回收价值最大的Region。每个Region都有一个Remembered set，用来记录Region对象的引用所在的Region。通过使用Remembered set，避免在做可达性分析的时候进行全堆扫描。初始标记，并发标记，最终标记，为了修正并发标记期间用户程序继续运行而产生变动的那部分记录，虚拟机将这段时间对象变化记录在线程的Remembered set logs中，最终标记阶段需要把Remebered set logs的数据合并到remembered set中，需要停顿用户线程，但是可以并行执行。筛选回收：首先对region中回收价值和成本进行排序，根据用户期望的停顿时间来制定回收计划，此阶段也可以做到并发执行，但是因为只回收一部分region，时间是用户可控制的，而且停顿用户线程可以大幅度提高收集效率。特点：空间整合，整体来看是基于标记-整理，局部来看(两个Region)，是基于复制算法，这意味着运行期间不会产生空间碎片。可停顿的预测：能让使用者指定在一个M毫秒的时间内，消耗在GC上的时间不超过N毫秒。
- 内存分配策略：对象优先在伊甸区，大对象直接在老年代，空间分配担保，长期存活的对象进入老年代，动态对象年龄判定(当幸存区中的同年龄对象总和超过所有对象的一半时，大于等于年龄的对象直接进入老年代)。

5. Full GC什么时候触发
    - System.gc，空间担保失败，Concurrent Mode Failure，老年代空间不足，1.7及之前的永久代空间不足。
6. BlockingQueue
    - ArrayBlockingQueue(数组阻塞队列)，DelayQueue(延迟阻塞队列)，LinkedBlockingQueue(链阻塞队列)，PriorityBlockingQueue(具有优先级的阻塞队列)，SynchronousQueue(同步队列)。
    - 四组不同的行为方式：(add,remove,element) 抛出异常。(offer,poll,peek) 返回特定值。(put,take) 阻塞。(offer,poll) 超时。
    - SynchronousQueue 是一个特殊的队列，它的内部同时只能够容纳单个元素。如果该队列已有一元素的话，试图向队列中插入一个新元素的线程将会阻塞，直到另一个线程将该元素从队列中抽走。同样，如果该队列为空，试图向队列中抽取一个元素的线程将会阻塞，直到另一个线程向队列中插入了一条新的元素。 据此，把这个类称作一个队列显然是夸大其词了。它更多像是一个汇合点。
7. 多线程sleep，wait的区别
    - sleep不会释放锁资源，wait会释放锁资源。sleep到时会自动唤醒，wait不带时间的需要其他线程唤醒。带时间的如果没有被notify，自己唤醒，两种情况，直接获得锁，没有获得锁，线程进入同步队列等待获得锁。
8. 可重入锁，Synchronized、Lock
    - 可重入锁：Synchronized通过加锁次数计数器来实现可重入锁，每个对象用于一个计数器，当线程获取该对象锁后，该计数器就会加1。lock通过调用底层的AQS的state值，通过增加state的值来实现可重入锁。

- Synchronized通过获得对象监视器来实现锁，作用在方法上，代码块，类锁。方法上调用ACC_SYNCHRONIZED，底层还是获得对象监视器。通过查看字节码文件有monitor enter和monitor exit指令，每个对象在同一个时间内只能与一个对象监视器相关联，一个监视器在一个时间点内只能被一个线程获得。
- Lock：底层基于AQS，默认是非公平锁。
- 对比：Synchronized基于JVM，lock基于API，Synchronized在异常情况下会释放锁，lock不会释放锁。Synchronized是等待获得锁的过程中是不可被中断的，lock是可以被中断的。Synchronized只能实现非公平锁，lock可以实现公平锁和非公平锁。lock可以绑定多个条件，而Synchronized只能与是否获得锁条件绑定。

9. ConcurrentHashMap怎么实现线程安全
    - 1.7 Segmemt，lock+CAS
    - 1.8 node+红黑树 Synchronized+CAS
10. threadLocal什么场景下使用
    - 通过线程隔离的方式防止任务在共享资源上产生冲突，线程本地存储是一种自动化机制，可以为使用相同变量的不同线程都创建不同的存储。
    - 每个线程都维护一个ThreadLocalMap类型的threadlocals，弱引用WeakReference，key是弱引用。
    - 应用场景：数据库管理，SimpleDateFormat，每个线程内保存类似于全局变量的信息，可以让不同方法直接使用，避免参数传递的麻烦，却不想被多线程共享。全局存储用户信息
11. 线程池参数，核心线程数的选择，怎么调整核心线程数
    - 核心线程数大小，最大线程数大小，线程存活时间单位，线程存活时间，阻塞队列，拒绝策略
    - ArrayBlockingQueue，LinkedBlockingQueue，SynchronousQueue,PriorityBlockingQueue
    - 直接抛出异常，调用者所在的线程来执行任务，丢弃阻塞队列中靠最前的任务，并执行任务，直接丢弃任务。
    - CPU密集型：核数+1，IO密集型：核数 * 2
    - newFixedThreadPool： 使用无界队列LinkedBlockingQueue，线程池里的线程数量不超过核心线程数，这导致了最大线程数量和存活时间是一个无效的参数。由于使用了无界队列，所以线程池永远不会拒绝，即饱和策略失效。
    - newSingleThreadPool： 只有一个线程，如果该线程异常结束，会重新创建一个新的线程继续执任务。由于使用了无界队列，所有永远不会执行饱和策略。
    - newCachedThreadPool： 最大线程数是整数最大值。使用SynchronousQueue。
12. 设计模式，单例模式、工厂模式、桥接模式、装饰器模式
    - 设计模式
    - 单例模式
    - 工厂模式
    - 桥接模式
    - 装饰器模式

Redis

1. 数据类型，底层实现
    - String：sds，int：可以用long类型表示的整数值，raw：大于44个字节，embstr：小于44个字节
    - List：quicklist
    - Hash：ziplist(zlbytes:整个ziplist占用的大小,zltail：尾偏移量,zllen：entry的数量,zlend：终止字节)，hashtable
    - Set：intset，hashtable
    - zset：ziplist，ht+skiplist

2. 主从同步
    - SYNC，RUNID，offset，复制积压缓冲区。
    - 数据同步，命令传播
3. 持久化
    1. RDB：①手动触发save bgsave，自动触发：配置save m n，主从复制的时候，debug reload，shutdown的时候，如果没有开启aof持久化，也会触发bgsave。
        1. RDB优点：RDB文件是经过压缩的二进制文件，占用内存空间很小，它保存了Redis某个时间点的数据集，很适合用于做备份。RDB非常适用于灾后恢复：它只有一个文件，并且内容十分紧凑，可以将它传送到别的数据中心。RDB最大化redis的性能。父进程在保存RDB文件时唯一要做的就是fork一个子进程，然后这个子进程处理接下来的保存工作，父进程无需执行任何的磁盘I/O操作。RDB在恢复大数据集时的速度比AOF的恢复速度要快。
        2. RDB的缺点：RDB保存的是整个数据集的状态，它是一个比较重的操作，如果操作太频繁，可能会对redis的性能产生很大的影响。RDB保存时使用fork子进程进行数据的持久化，如果数据比较大时，fork可能会非常耗时，造成redis停止处理服务N毫秒。linux fork采用的时候copy-on-write的方式。Redis在执行RDB持久化期间，如果client写入数据频繁，将会增加Redis占用的内存。刚fork的时候，父进程和子进程共享内存，随着父进程处理写操作，主进程需要将修改的页面copy一份出来进行修改操作。极端情况下，如果所有的页面都需要修改，则此时的内存占用是原来的2倍。RDB文件是二进制的，没有可读性，AOF在了解其结构的情况下可以手动修改或者补全。
    2. AOF：保存了redis服务器所执行的所有的写命令来记录数据库的状态。并在数据库重启时，重新执行这些命令来还原数据集。
        1. AOF持久化功能的实现分为三个步骤：**命令追加、文件写入、文件同步**。。命令追加：当AOF持久化功能打开时，服务器执行完一个命令后，会将执行的命令追加到服务器状态的缓冲区的末尾。文件写入和文件同步：linux为了提升性能，使用了页缓存(page cache)。当我们将aof缓冲区中的数据写入磁盘时，此时数据并没有真正的落盘，而是在page cache中，为了将页缓存中的数据真正的落盘，需要执行某些命令来执行强制落盘。文件同步就是文件刷盘操作。
        2. AOF的优点：AOF比RDB可靠。我们在设置刷盘指令时，默认是everysec，在这种配置下，即使服务器宕机，也只是丢失了一秒钟的数据。AOF是纯追加的日志文件。即使日志因为某些原因而包含了未写入完整的命令，也能通过工具轻易的修复这种问题。当AOF文件太大时，Redis会进行AOF重写，重写后的AOF文件包含了恢复当前数据的最小命令集合。整个重写是安全的，重写是在新文件上进行的，重写完后，Redis会把新旧文件进行替换，开始把数据写到新文件上。AOF文件有序的保存了对数据库执行的所有写入操作以Redis协议的格式保存，十分容易被人读懂和分析。
        3. AOF的缺点：对于相同的数据集，AOF文件比RDB文件大。根据使用的刷盘策略，AOF的速度可能比RDB慢。因为个别命令的原因，导致AOF文件在重新载入后，无法将数据集恢复到之前的样子。
4. AOF重写
    1. AOF重写缓冲区，太大的话会创建通道，一个文件事件，通过通道将命令尽可能的传输，子进程尽可能的读取，连续20次读不到就结束
5. 哨兵机制
    1. 主从复制存在的问题可以用哨兵机制来解决。使用Sentinel来完成节点选举工作。
    2. 哨兵用于监控Redis集群中Master主服务器的工作状态，可以完成Master和Slave的主从转换。
    3. 哨兵leader的选出
    4. 主库下线的判定
    5. 新主节点的选出

    6. 故障的转移
6. redis集群
    - 主从模式，哨兵模式，redis-cluster
    - 主从复制优缺点：支持主从复制，主机会自动将数据同步到从机，可以进行读写分离。为了分载Master的读操作压力，Slave服务器可以为客户端提供只读操作的服务，写服务仍然必须由Master来完成。Slave同样可以接受其它Slaves的连接和同步请求，这样可以有效的分载Master的同步压力。Master Server是以非阻塞的方式为Slaves提供服务。所以在Master-Slave同步期间，客户端仍然可以提交查询或修改请求。Slave Server同样是以非阻塞的方式完成数据同步。在同步期间，如果有客户端提交查询请求，Redis则返回同步之前的数据。缺点：Redis不具备自动容错和恢复功能，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者手动切换前端的IP才能恢复。主机宕机，宕机前有部分数据未能及时同步到从机，切换IP后还会引入数据不一致的问题，降低了系统的可用性。Redis较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。
    - 哨兵模式的优缺点：哨兵模式是基于主从模式的，所有主从的优点，哨兵模式都具有。主从可以自动切换，系统更健壮，可用性更高。Redis较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。
    - Redis-Cluster采用无中心结构，它的特点如下：所有redis节点相连(PING-PONG机制)，内部使用二进制协议优化传输速度和带宽。节点的fail是通过集群中超过半数的节点检测失效时才生效。客户端与redis节点相连不需要中间代理层，只需要与集群中任一可用的redis节点相连即可。
    - 在redis节点上都有这么两个东西，插槽(slot)，它的取值范围是0-16383，还有一个是集群，可以理解为是集群管理的插件。当我们存取的key到达的时候，redis会根据crc16算法得出一个结果，然后把结果对16384取余数，这样每个key都会对应一个0-16383之间的哈希槽，通过这个值去找到对应这个插槽所对应的节点，然后直接跳转到这个对应节点上进行存取操作。为了保证高可用性，redis集群引入了主从模式，一个主节点对应一个或多个从节点，当主节点宕机的时候，就会启用从节点。当其他主节点ping一个主节点A的时候，如果半数以上的主节点与A通信超时，那么认为节点A宕机了。如果主节点A和从节点A1都宕机了，那么该集群就无法再提供服务了
7. 一致性Hash，扩容缩容怎么实现
    - 扩容：首先将新节点加入集群，默认是主节点，首先需要确认哪些槽需要被迁移到目标节点，然后获取槽中的key，将槽中的key全部迁移到目标节点，然后向集群所有节点广播槽全部迁移到了目标节点，
    - 缩容：首先判断下线的节点是否是主节点，以及主节点上是否有槽，若主节点上有槽，需要将槽迁移到集群中的其他主节点，槽迁移完成之后需要向所有节点广播该节点准备下线。最后需要将该下线主节点的从节点指向其他主节点
8. 缓存穿透，缓存雪崩，缓存击穿，缓存污染

Spring

1. AOP原理 AOP（Aspect Oriented Programming）是基于切面编程的，可无侵入的在原本功能的切面层添加自定义代码，一般用于日志收集、权限认证等场景。通过动态代理实现invocationHandler接口，并且把目标对象注入到代理对象中，获取应用到此方法的执行器链，如果有执行器链则创建MethodInvocation，并调用proceed方法，否则直接反射调用目标方法。

    - AOP的相关注解有哪些

        1. @Aspect：声明被注解的类是一个切面Bean
        2. @Before：前置通知。指在某个连接点之前执行的通知。

3. @After：后置通知：在某个连接点退出时执行的通知(不论正常返回还是异常退出)
4. @AfterReturning：返回后通知。
5. @AfterThrowing：异常通知，方法抛出异常导致退出时执行的通知。

- AOP的相关术语

1. Aspect：切面，一个关注点的模块化，这个关注点可能会横切多个对象。
2. Joinpoint：连接点，程序执行过程中的某一行为，即业务层中的所有方法。
3. Advice：通知，指切面对于某个连接点所产生的的动作，包括前置通知，后置通知，返回后通知，异常通知和环绕通知。
4. Pointcut：切入点，指被拦截的连接点，切入点一定是连接点，但连接点不一定是切入点。
5. Proxy：代理
6. Target：代理的目标对象，指一个或多个切面所通知的对象。
7. Weaving：织入，指把增强应用到目标对象来创建代理对象的过程。

- AOP的过程

1. AOP是从BeanPostProcessor后置处理器开始，后置处理器可以监听容器触发的Bean生命周期时间，向容器注册后置处理器以后，容器中管理的Bean就具备了接收IOC容器回调事件的能力。
2. BeanPostProcessor的调用发生在SpringIOC容器完成Bean实例对象的创建和属性的依赖注入后，为Bean对象添加后置处理器的入口是initialization方法。
3. Spring中JDK动态代理通过JdkDynamicAopProxy调用Proxy的newInstance方法来生成代理类，JdkDynamicAopProxy也实现了InvocationHandler接口，invoke方法的具体逻辑是先获取应用到此方法上的执行器链，如果有执行器则创建MethodInvocation并调用proceed方法，否则直接反射调用目标方法。因此Spring AOP对目标对象的增强是通过拦截器实现的。

2. 解释动态代理
   - 动态代理：在程序运行期间，创建目标对象的代理对象，并对目标对象中的方法进行功能性增强的一种技术。在生成代理对象的过程中，目标对象不变，代理对象中的方法是目标对象中的增强方法。可以理解为在运行期间，对象中方法的动态拦截，在拦截方法前后执行功能操作。
   - 基于接口的动态代理：jdk，基于类的动态代理：CGLIB
3. Spring中的拦截器是怎么实现的
   - 基于反射机制实现的，动态代理，在目标方法前后执行相应的操作。将拦截器按照一定的顺序连接成一条链，在访问被拦截的方法或字段时，拦截器链中的拦截器就会按照之前定义好的顺序一次执行拦截器中的方法。
4. SpringMVC请求处理流程
   1. Web容器启动时，会通知Spring初始化容器，加载Bean的定义信息并初始化所有单例Bean，然后遍历容器中的Bean，获取每一个Controller中的所有方法访问的URL，将URL和对应的Controller放到一个Map中。
   2. 所有请求都发给DispatcherServelet前端处理器处理，DispatcherServelet会请求HandlerMapping寻找被Controller注解修饰的Bean和被RequestMapping修饰的方法和类。生成Handler和HandlerInterceptor，并以HandlerExecutionChain这样一个执行器链的形式返回。
   3. DispatcherServlet使用Handler寻找对应的HandlerAdapter，HandlerAdapter执行相应的Handler方法，并把请求参数绑定到方法形参上，执行方法获得ModelAndView。
   4. DispatcherServelet将ModelAndView讲给ViewResolver进行解析，得到View的物理视图，然后对视图进行渲染，将数据填充到视图中返回给客户端。

MySQL

1. 索引
2. InnoDB、MyISAM
3. 主从同步、如何避免主从延迟
4. MySQL读写分离、路由策略、分库分表、取模之后表变化了/扩容了怎么办，怎么实现重新路由
5. MySQL事务隔离级别
6. undo log、redo log、bin log的区别，哪些地方用了
7. B+树，B树区别，节点计算公式
8. 主索引、非主索引
9. 索引方式
10. 常见索引，主键索引、普通索引的区别
11. MySQL缓存如何实现的，SQL缓存，数据缓存

- 缓存sql文本和缓存结果，用KV形式保存在服务器内存中，如果运行相同的sql，服务器直接从缓存中去获取结果，不需要再去解析、优化、执行sql。如果这个表改变了，那么使用是这个表中的所有缓存将不再有效，查询缓存值的相关条目将被清空。对于频繁更新的表，查询缓存不合适，对于一些不变的数据且有大量相同sql查询的表，查询缓存会节省很大的性能。 - 命中条件：缓存存在一张hash表中，通过查询sql，查询数据库，客户端协议等作为key，在判断命中前，mysql不会解析sql，而是使用sql去查询缓存，sql上的任何字符的不同都会导致缓存不命中。如果查询有不确定的数据，那么查询完后结果不会被缓存。 - InnoDB查询缓存：InnoDB会对每个表设置一个事务计数器，里面存储着当前最大的事务ID。当一个事务提交时，InnoDB会使用MVCC中系统事务ID最大的事务ID更新当前表的计数器。只有比这个最大ID大的事务才能使用查询缓存，其他比这个ID小的事务则不能使用查询缓存。另外，在InnoDB中，所有加锁的操作的事务都不能使用任何查询缓存。查询必须是完全相同的(逐字节相同)才能够被认为是相同的。另外，同样的查询字符串由于其它原因可能认为是不同的。使用不同的数据库、不同的协议版本或者不同默认字符集的查询被认为是不同的查询并且分别进行缓存。

12. Buffer Pool如何实现缓存，Buffer Pool脏数据如何避免，flush刷盘触发动机，change buffer的应用
    - 读多写少用change buffer，反之不要使用。(账单流水业务)
    - 为什么写缓存优化仅适用于非唯一普通索引页呢？：如果索引设置了唯一属性，在进行修改操作的时候，必须进行唯一性检查，就必须将页读入内存中才能判断，也就没有必要用change buffer。
    - 当需要更新一个数据页的时候，如果数据页在内存中就直接更新，如果不在内存中，在不影响数据一致性的前提下，InnoDB会将这些操作缓存到change buffer中，这样就不需要在磁盘中读取这个页。当下次查询需要访问这个数据页的时候，将数据页读入内存，然后执行change buffer中与这个页相关的操作。通过这种方式就能保证这个数据逻辑的正确性。
    - 将change buffer中的操作应用到原始数据页上，得到最新结果的过程叫做`merge`，除了这个数据页会定期触发merge外，系统有后台线程会定期merge。在数据库正常关闭的过程中，也会执行merge操作。
    - 如果能够将更新操作先记录到change buffer，减少读磁盘，语句的执行速度会得到明显的提升。而且数据读入内存是需要占buffer pool的，所以这种方式还能避免占用内存，提高内存利用率。
13. 联合索引，索引覆盖，索引下推，前缀索引的应用场景 当字符串本身可能比较长，而且前几个字符就开始不相同，适合使用前缀索引；相反情况下不适合使用前缀索引
14. MySQL MRR机制
    - 通过把随机磁盘读，转换为顺序磁盘读，从而提高了索引查询的性能。对于InnoDB，会按照聚簇索引键值排好序，再顺序的读取聚簇索引。顺序读带来了几个好处：1.磁盘和磁头不再需要来

回做机械运动。2.可以充分利用磁盘预读。3.在一次查询中，每一页的数据只会从磁盘读取一次。

- 索引本身就是为了减少磁盘 IO，加快查询，而 MRR，则是把索引减少磁盘 IO 的作用，进一步放大。(空间换时间算法)

15. MySQL多表连接的时候有什么优化点

ZooKeeper

RPC

分布式事务

Kafka

1. kafka基本实现原理，和其他的区别
2. zk的作用，offset的作用
3. kafka主本副本
4. leader分区选举策略
5. 如何实现零拷贝
6. 性能优化，kafka为什么这么快
7. pull和push的优缺点