**Java工程面试**

| | | | |
|---|---|---|---|
| **笔记本:** | 面试 | | |
| **创建时间:** | 2022/9/22 21:45 | **更新时间:** | 2023/12/13 19:06 |
| **作者:** | 润冬之初 | | |
| **URL:** | https://www.nowcoder.com/exam/interview/71282604/test?paperId=47570073... | | |

1. **redis过期策略**

在 Redis 中，有三种类型的过期策略可以设置，分别是：
过期时间（Expire）：在设置键值对的同时，可以设置一个过期时间，Redis 会自动在该键值对在指定的时间内过期，过期后会自动删除该键值对。

定时删除（Evict）：通过配置 maxmemory 限制 Redis 的内存使用量，在内存满时 Redis 会将一些键值对从内存中删除，优先删除的是那些过期时间最短的键值对，以此来保证 Redis 的内存使用量不会超过限制。

惰性删除（Lazy deletion）：在访问一个键值对时，Redis 会先检查该键值对是否过期，如果过期则会立即删除该键值对。惰性删除的优点是不需要额外的删除操作，节省了服务器资源，缺点是可能会有大量过期的键值对占用内存。一般情况下，使用过期时间是最常见的过期策略，而惰性删除可以作为补充策略来保证 Redis 的内存使用量不会超过限制。当然，在特定场景下也可以使用其他的过期策略。

2. **关系数据库的特点**

关系型数据库，是指采用了关系模型来组织数据的数据库。
关系模型中常用的概念：
· 关系：可以理解为一张二维表，每个关系都具有一个关系名，就是通常说的表名
· 元组：可以理解为二维表中的一行，在数据库中经常被称为记录
· 属性：可以理解为二维表中的一列，在数据库中经常被称为字段
· 域：属性的取值范围，也就是数据库中某一列的取值限制
· 关键字：一组可以唯一标识元组的属性，数据库中常称为主键，由一个或多个列组关系模式：指对关系的描述。其格式为：关系名(属性1，属性2， ... ... ，属性N)，在数据库中成为表结构
关系型数据库的优点：
· 容易理解：二维表结构是非常贴近逻辑世界的一个概念，关系模型相对网状、层次等其他模型来说更容易理解
· 使用方便：通用的SQL语言使得操作关系型数据库非常方便
· 易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率

3. **父类的析构函数为什么要定义为虚函数**

不定义为虚函数的话，delete父类只释放了父类的内存空间，而子类的内存空间没有释放，造成了内存泄漏。而定义了虚函数则不会造成内存泄漏。

4. **override和overload的区别**

java中,override：在继承时，子类方法覆盖父类方法，返回值、方法名和参数列表和父类函数相同；overload：同一个类中，方法名相同，但参数个数、顺序和类型至少其中一个不同，则被视为不同的方法。

5. **MyBatis 中$和#的区别**

#{}: 底层使用PreparedStatement。特点：先进行SQL语句的编译，然后给SQL语句的占位符问号?传值。可以避免SQL注入的风险。
${}：底层使用Statement。特点：先进行SQL语句的拼接，然后再对SQL语句进行编译。存在SQL注入的风险。
优先使用#{}，这是原则。避免SQL注入的风险。

- #{}将传入的数据都当成一个字符串，会对自动传入的数据加一个双引号。如：order by #{user_id}，如果传入的值是111,那么解析成sql时的值为order by "111", 如果传入的值是id，则解析成的sql为order by "id".
- ${}将传入的数据直接显示生成在sql中。如：order by ${user_id}，如果传入的值是user_id,那么解析成sql时的值为order by user_id, 如果传入的值是id，则解析成的sql为order by id.
- #{}方式能够很大程度防止sql注入。
- ${}方式无法防止Sql注入。
- ${}方式一般用于传入数据库对象，例如传入表名.
- 一般能用#的就别用$.

MyBatis排序时使用order by 动态参数时需要注意，用$而不是#

6. **java类文件中包含了哪些信息**


https://blog.csdn.net/m0_61955744/article/details/131760006
https://blog.csdn.net/qq_33762043/article/details/95318721

7. **Spring boot 启动的时候配置文件的加载顺序**

bootstrap.properties -> bootstrap.yml -> application.properties -> application.yml>application.yaml

其中 bootstrap.properties 配置为最高优先级，先加载的会被后加载的覆盖掉，所以.properties和.yml同时存在时，.properties会失效，.yml会起作用。

bootstrap/ application 的应用场

- bootstrap.yml 和application.yml 都可以用来配置参数。
- bootstrap.yml 可以理解成系统级别的一些参数配置，这些参数一般是不会变动的。
- application.yml配置文件这个容易理解，application.yml 可以用来定义应用级别的，主要用于 Spring Boot 项目的自动化配置。

bootstrap 配置文件有以下几个应用场景：

- 使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；
- 一些固定的不能被覆盖的属性
- 一些加密/解密的场景；

8. **java中如何唯一确定一个类**

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在Java虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间。这句话可以表达得更通俗一些：比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个Class文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等。
**https://blog.csdn.net/thetimelyrain/article/details/115325819**

9. **什么是双亲委派模型**

## 类加载器（ClassLoader）

主要负责动态加载Java类到Java虚拟机的内存空间中。类通常是按需加载，也就是第一次使用该类时才加载

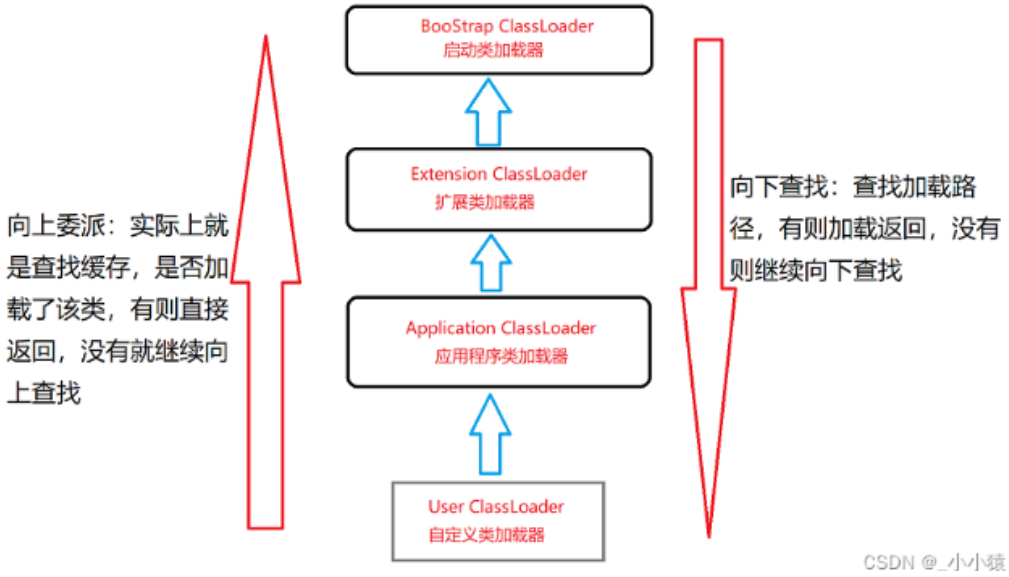
*双亲委派除了顶层的启动类加载器之外，其余加载器都应当有自己的父类加载器*

双亲委派模式是在Java 1.2后引入的，原理就是 如果一个类加载器收到类加载的请求，它不会自己先去加载，而是先把这个请求委托给他的父类加载器，如果他的父类加载器还有父类加载器，它就会进一步向上委托，直到委托到顶层的启动类加载器。如果父加载器可以加载这个类请求的话就会成功返回，反之，父加载器无法加载此任务，子类加载器才会尝试自己去加载。

双亲委派这种加载方式可以有效避免类的重复加载，如果父类加载器已经加载过，子ClassLoader就没有比要在重新加载一次了。
如果要通过网络传递一个伪造的类，通过双亲委派发现这个类已经被加载过，就不会重新加载传递过来的类，而是直接返回已经被加载过的类，便可以防止Java核心类被随意篡改。

**向上委派到顶层加载器为止，向下查找找到发起加载的加载器为止。**

委派到顶层后，如果缓存还是没有，则到加载路径中查找，有则加载返回，没有则继续向下查找。




10. **java中this关键字是如何来的**

https://blog.csdn.net/weixin_41920291/article/details/98118035

1. 构造方法、实例方法的第一个参数是this！这是由编译器自动添加的（回头看看4-19行的默认构造方法的逻辑，第八行的aload_0指令就是将this引用推送至栈顶，即压入栈）。

2. this引用变量的数据类型是this所在方法的所属类。即编码时，this出现在哪个类中，this的数据类型就是这个类（回头看看17-19行，默认的无参构造方法这个唯一的局部变量就是this！且其静态类型为Base！）。
3. 既然编译器会自动给实例方法添加一个this参数，那么就不难理解，当调用某个实例对象的方法时，编译器会将该实例对象当做参数传递到调用方法中了，这也是我们能在无参的方法中使用this的原因。

11. **不加锁实现并发**

https://blog.csdn.net/shy111111111/article/details/129553866
（1）CAS原子操作
（2）阻塞队列
（3）copyOnWriteList数据结构
（4）ThreadLocal类，给每个线程保存一份变量副本

12. **Hashmap扩容机制**

hashmap是一种基于数组和链表（或红黑树）的数据结构，它可以存储键值对的映射关系。hashmap的扩容机制是指当hashmap中的元素个数超过数组长度乘以负载因子时，就会重新分配一个更大的数组，并将原来的元素重新计算哈希值并插入到新的数组中。不同版本的Java实现了不同的扩容机制。

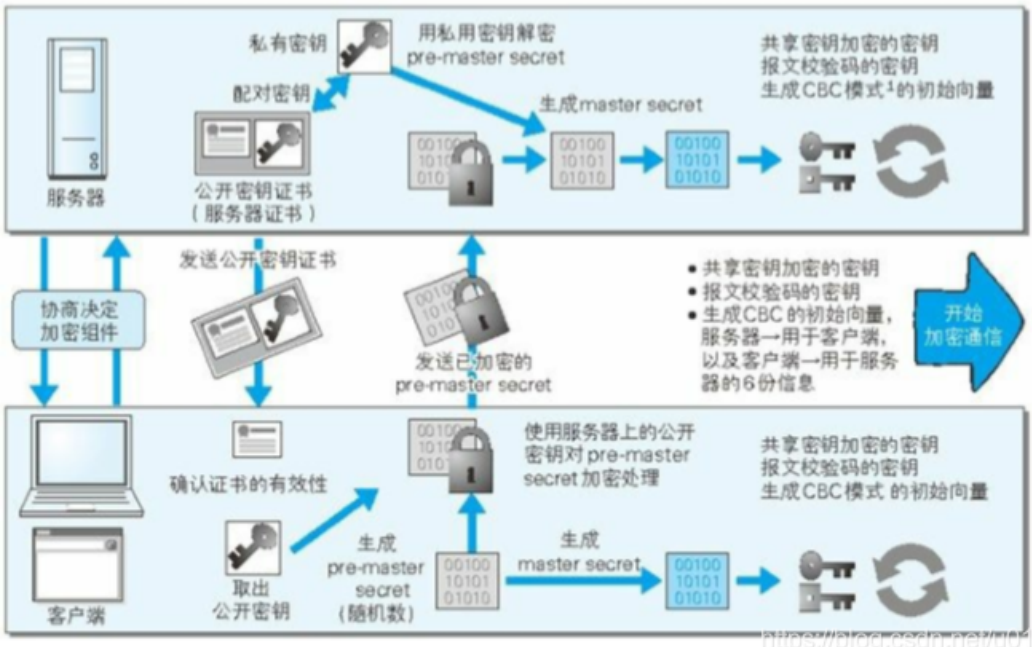
在JDK1.7中，hashmap的扩容机制有以下特点：

- hashmap的初始容量默认是16，负载因子默认是0.75，阈值默认是12（16*0.75）。
- hashmap的容量必须是2的幂次方，这样可以保证哈希值和数组长度取模时只需要进行位运算，提高效率。
- 当hashmap中的元素个数超过阈值时，就会触发扩容，新的容量是原来的2倍，新的阈值也是原来的2倍。
- 在扩容过程中，hashmap会遍历原来的数组中的每个链表，并将链表中的每个节点重新计算哈希值，找到新数组中对应的位置，以头插法插入到新链表中。这样做可能会导致链表反转和多线程环境下的死循环问题。

在JDK1.8中，hashmap的扩容机制有以下改进：

- hashmap在第一次调用put方法时才会初始化数组，而不是在创建对象时就初始化。
- hashmap在初始化或扩容时，会根据指定或默认的容量找到不小于该容量的2的幂次方，并将其赋值给阈值。然后在第一次调用put方法时，会将阈值赋值给数组长度，并让新的阈值等于数组长度乘以负载因子。
- 在扩容过程中，hashmap不需要重新计算节点的哈希值，而是根据哈希值最高位判断节点在新数组中的位置，要么在原位置，要么在原长度加上原位置处。
- 在扩容过程中，hashmap会正序遍历原来的数组，并保持链表中节点的相对顺序不变。
- 如果某个链表中的节点数超过8个，并且数组长度大于等于64，则会将链表转化为红黑树，提高查找效率。以上就是我对hashmap扩容机制的简要介绍。

13. **ssl 建立过程**



ssl证书还提供了数据完整性保护。证书中包含了一个摘要算法，用于生成数据的hash值。在数据传输的过程中，网站会对数据进行hash运算，并将hash值和数据一起发送给用户。用户收到数据以后，会使用相同的摘要算法对数据进行hash运算，并将结果和收到的hash值进行比较。如果hash值匹配则数据没有被篡改。

浏览器会通过摘要算法验证证书有效性。

手动检查证书有效性：

1. 检查浏览器地址栏：在浏览器地址栏中，有效的SSL证书会显示一个锁形状的图标，或者显示网站的名称前面有一个绿色的锁。点击锁图标可以查看证书的详细信息。
2. 验证证书颁发机构（CA）：SSL证书由受信任的证书颁发机构（CA）签发。在浏览器中，可以查看证书的颁发机构。确保证书颁发机构是可信的，如Symantec、Comodo、Let's Encrypt等。
3. 检查证书有效期：SSL证书有一个有效期限，通常为一年或更长。确保证书没有过期，否则可能会导致安全风险。
4. 验证证书的域名匹配：SSL证书是与特定域名相关联的。确保证书上的域名与访问的网站域名完全匹配，包括子域名。
5. 使用在线工具进行验证：有一些在线工具可以帮助验证SSL证书的有效性。例如，SSL Shopper和SSL Labs提供了SSL证书的检测工具，可以检查证书的有效性和安全性。

https://blog.csdn.net/u012957549/article/details/105331956

14. **数据库加锁原理**

https://blog.csdn.net/hebeind100/article/details/84690385

15. **redis如何保持主从一致**

先删除缓存，更新数据库；先更新数据库再删除缓存；先删除缓存，再写入数据库，等待一段时间，再删除缓存（双删超时策略）；利用数据库binlog队列异步刷新缓存。
（1）先删除缓存，更新数据库
（2）先删除缓存，再写入数据库，等待一段时间，再删除缓存（双删超时策略）
（3）利用数据库binlog队列异步刷新缓存
https://www.cnblogs.com/tsaiccj/p/15787349.html

## 16. java 泛型的好处

Java泛型是Java SE1.5中引入的一个新特性，其本质是参数化类型，也就是说所操作的数据类型被指定为一个参数（type parameter）这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法。好处：
（1）类型安全
泛型的主要目标是提高Java程序的类型安全。通过知道使用泛型定义的变量的类型限制，编译器可以在非常高的层次上验证类型假设。没有泛型，这些假设就只存在于系统开发人员的头脑中。
（2）消除强制类型转换
泛型的一个附带好处是，消除源代码中的许多强制类型转换。这使得代码更加可读，并且减少了出错机会
（3）提高运行效率
在非泛型编程中，将简单类型作为Object传递时会引起Boxing（装箱）和Unboxing（拆箱）操作，这两个过程都是具有很大开销的。引入泛型后，就不必进行Boxing和Unboxing操作了，所以运行效率相对较高，特别在对集合操作非常频繁的系统中，这个特点带来的性能提升更加明显。
（4）提升代码复用能力

## 17. 抽象类的好处

抽象是一种编程技术，它用于隐藏复杂性并将代码逻辑分离。Java中，抽象主要是通过抽象类和接口实现的。通过使用抽象类和接口，我们可以定义通用的行为和属性，提高代码的可维护性和可扩展性。

- 降低了某些方法的复杂性（只需要调用一些抽象类里面实现的功能函数，就能完成某些业务逻辑函数编写）。
- 避免代码重复,提高可重用性。
- 只向用户提供重要细节,才能提高应用的安全性（只需要集成抽象类，实现一些特殊业务逻辑就行了，然后不用暴露更多的细节，提高安全性）。
- 抽象类继承接口，不实现其中的方法的好处是，首先为抽象类增加了一个接口对应的功能，这个接口的功能可以在子类中灵活实现。

## 18. 设计模式的原则

**开闭原则：**OOP 中最基础的原则，指一个软件实体（类、模块、方法等）应该对扩展开放，对修改关
闭。强调用抽象构建框架，用实现扩展细节，提高代码的可复用性和可维护性。
**单一职责原则：**一个类、接口或方法只负责一个职责，降低代码复杂度以及变更引起的风险。
**依赖倒置原则：**程序应该依赖于抽象类或接口，而不是具体的实现类。
**接口隔离原则：**将不同功能定义在不同接口中实现接口隔离，避免了类依赖它不需要的接口，减少了接
口之间依赖的冗余性和复杂性。

**里氏替换原则：**开闭原则的补充，规定了任何父类可以出现的地方子类都一定可以出现，可以约束继承
泛滥，加强程序健壮性。
**迪米特原则：**也叫最少知道原则，每个模块对其他模块都要尽可能少地了解和依赖，降低代码耦合度。
**合成/聚合原则：**尽量使用组合(has-a)/聚合(contains-a)而不是继承(is-a)达到软件复用的目的，避免滥
用继承带来的方法污染和方法爆炸，方法污染指父类的行为通过继承传递给子类，但子类并不具备执行
此行为的能力；方法爆炸指继承树不断扩大，底层类拥有的方法过于繁杂，导致很容易选择错误。

19. **缓存穿透的解决方案**

**概念**
缓存穿透是指查询一个缓存中和数据库中都不存在的数据，导致每次查询这条数据都会透过缓存，直接查库，最后返回

空。当用户使用这条不存在的数据疯狂发起查询请求的时候，对数据库造成的压力就非常大，甚至可能直接挂掉。

**解决方案**
（1）缓存空对象
当数据库中查不到数据的时候，我缓存一个空对象，然后给这个空对象的缓存设置一个过期时间，这样下次再查询该数
据的时候，就可以直接从缓存中拿到，从而达到了减小数据库压力的目的。但这种解决方式有两个缺点：（1）需要缓存
层提供更多的内存空间来缓存这些空对象，当这种空对象很多的时候，就会浪费更多的内存；（2）会导致缓存层和存储
层的数据不一致，即使在缓存空对象时给它设置了一个很短的过期时间，那也会导致这一段时间内的数据不一致问题。

（2）布隆过滤器
所谓布隆过滤器，就是一种数据结构，它是由一个长度为m bit的位数组与n个hash函数组成的数据结构，位数组中每个
元素的初始值都是0。在初始化布隆过滤器时，会先将所有key进行n次hash运算，这样就可以得到n个位置，然后将这n
个位置上的元素改为1。这样，就相当于把所有的key保存到了布隆过滤器中了。然后每次查询先通过布隆过滤器
check，是否key未过期，且从数据库里面load过数据了。

20. **String、StringBuilder、 StringBuffer的区别**

| | | StringBuffer | StringBuilder |
|---|---|---|---|
| 执行速度 | 最差 | 其次 | 最高 |
| 线程安全 | 线程安全 | 线程安全 | 线程不安全 |
| 使用场景 | 少量字符串操作 | 多线程环境下的大量操作 | 单线程环境下的大量操作 |

- 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。
- 
- 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
- 
- 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。
- 
- StringBuffer和StringBuilder二者的功能和方法完全是等价的
- 
- StringBuffer线程安全，StringBuilder线程不安全
- 
- StringBuffer的公开方法都是被synchronized 修饰的，而 StringBuilder并没有。
- 
- StringBuffer与StringBuilder都继承自AbstractStringBuilder。
- 
- StringBuffer从JDK1.0就有了，StringBuilder是JDK5.0才出现

21. **kafka: ZAB 协议**

ZAB协议是zookeeper用来实现一致性的原子广播协议，该协议描述了Zookeeper是如何实现一致性的，分为三个阶段：
（1）leader节点选取： 从Zookeeper集群中选出一个节点作为leader，所有的写请求都会由leader节点处理
（2）数据同步阶段： 集群中所有节点中的数据要和leader节点保持一致，如果不一致则需要同步
（3）请求广播阶段： 当leader节点接收到写请求时，会利用两阶段提交（a.预提交：收到写请求以后生成日志，然后把日志发送给follower，等待ack；b. 收到超过半数follower节点的ack后，发送commit命令，然后leader和follower根据日志更新缓存）来广播改写请求，使得写请求像事务一样在其他节点执行，保持节点上的数据一致。

值得注意的是，Zookeeper只是尽量的在达到强一致性，实际上只是保持最终一致。

22. **为甚Zookeeper可以作为注册中心**

可以利用Zookeeper的临时节点和watch机制来实现注册中心的自动注册和发现，另外Zookeeper中的数据都是存在内存中的，并且Zookeeper底层采用了nio，多线程模型，所以Zookeeper的性能是比较高的，所以可以用来作为注册中心。但是如果考虑到注册中心应该是注册可用性的话，那么Zookeeper不太合适，因为Zookeeper是cp的，注重一致性，集群数据不一致的时候，集群将不可用。

### 23. Zookeeper中leader选举的流程

pk是指将自己的选票和别人发过来的选票进行pk

对于Zookeeper集群，整个集群需要从集群节点中选出一个节点作为Leader，大体流程如下：

1. 集群中各个节点首先都是观望状态，一开始都会投票给自己，认为自己比较适合作为leader
2. 然后相互交互投票，每个节点会收到其他节点发过来的选票，然后pk，先比较zxid，zxid大者获胜，zxid如果相等则比较myid，myid大者获胜
3. 一个节点收到其他节点发过来的选票，经过PK后，如果PK输了，则改票，此节点就会投给zxid或myid更大的节点，并将选票放入自己的投票箱中，并将新的选票发送给其他节点
4. 如果pk是平局则将接收到的选票放入自己的投票箱中
5. 如果pk赢了，则忽略所接收到的选票
6. 当然一个节点将一张选票放入到自己的投票箱之后，就会从投票箱中统计票数，看是否超过一半的节点都和自己所投的节点是一样的，如果超过半数，那么则认为当前自己所投的节点是leader
7. 集群中每个节点都会经过同样的流程，pk的规则也是一样的，一旦改票就会告诉给其他服务器，所以最终各个节点中的投票箱中的选票也将是一样的，所以各个节点最终选出来的leader也是一样的，这样集群的leader就选举出来了

### 24. Zookeeper管理的节点之间的数据如何同步

1. 首先集群启动时，会先进行领导者选举，确定哪个节点是Leader，哪些节点是Follower和Observer
2. 然后Leader会和其他节点进行数据同步，采用发送快照和发送Diff日志的方式
3. 集群在工作过程中，所有的写请求都会交给Leader节点来进行处理，从节点只能处理读请求
4. Leader节点收到一个写请求时，会通过两阶段机制来处理
5. Leader节点会将该写请求对应的日志发送给其他Follower节点，并等待Follower节点持久化日志成功
6. Follower节点收到日志后会进行持久化，如果持久化成功则发送一个Ack给Leader节点
7. 当Leader节点收到半数以上的Ack后，就会开始提交，先更新Leader节点本地的内存数据
8. 然后发送commit命令给Follower节点，Follower节点收到commit命令后就会更新各自本地内存数据
9. 同时Leader节点还是将当前写请求直接发送给Observer节点，Observer节点收到Leader发过来的写请求后直接执行更新本地内存数据
10. 最后Leader节点返回客户端写请求响应成功
11. 通过同步机制和两阶段提交机制来达到集群中节点数据一致

### 25. Dubbo支持的负载均衡策略

随机：从多个服务提供者中选择一个来处理本次请求，请求量越多则分布越均匀，并支持按权重设置随机概率
轮询：依次选择服务提供者来处理请求，并支持按权重进行轮询，底层采用的是平滑加权轮询算法
最小活跃调用数：统计服务提供者当前正在处理的请求，下次请求过来的时候交给活跃数最小的服务器来处理
一致性哈希：相同参数的请求总是发到同一个服务提供者

### 26. Spring Cloud和Dubbo的区别

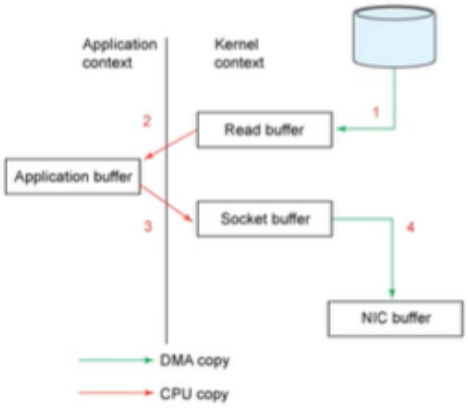
Spring Cloud 是一个微服务框架，提供了微服务领域中很多功能组件，Dubbo一开始是一个RPC调用框架，核心是解决服务调用的问题，Spring Cloud 是一个大而全的框架， Dubbo更侧重于服务调用，Dubbo提供的功能没有Spring Cloud全面，但是Dubbo的嗲用性能比Spring Cloud高，不过Spring Cloud和Dubbo并不对立，是可以结合起来一起使用的。

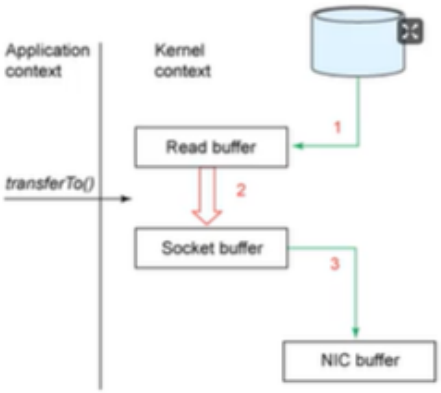
27. **BIO NIO AIO分别是啥**

（1）BIo： 同步阻塞io， 使用bio读取数据时， 线程会阻塞住，并且需要线程主动去查询是否有数据可读， 并且需要处理完一个socket之后才能处理另一个socket。（服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销）
（2）NIO： 同步非阻塞io，使用nio读取数据时，线程不会阻塞，但是需要线程主动查询是否有io事件。（服务器实现模式为一个线程处理多个请求（连接），即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求就进行处理）
（3）AIO：异步非阻塞io，Aio读取数据时，线程不会阻塞，不需要服务器线程线程主动查询（操作系统加载数据完毕后，操作系统会主动通知服务器启动线程处理数据，采用回调函数）

28. **零拷贝**

零拷贝值得是，应用程序需要把内核中的一块区域的数据转移到另一块内核区域去时，不需要经过先复制到用户空间，在转移到目标内核区域，而是直接转移。




29. **Kafka的pull和push的优缺点**

（1） pull表示消费者主动拉取，可以批量拉取，也可以单条拉取，所以pull可以由消费者自己控制，根据自己的消息处理能力来控制，但是消费者不能及时知道是否有消息，可能会拉取到空消息
（2）push表示Broker主动给消费者推送消息，所以肯定是有消息才能推送，但是消费者不能按自己的能力来消费消息，所以推过来多少消息，消费者就得到多少消息，所以可能造成网络阻塞，消费者压力大的问题。

30. **CopyOnWriteList底层原理**

（1）内部也是通过数组来实现的，在添加元素时，会复制出一个新数组，写操作在新数组上进行，读操作在原数组上进行
（2）写操作会加锁，防止并发写入丢失数据的问题
（3）写操作结束之后会把原数组指向新数组
（4）CopyOnWriteList允许在写操作时读取数据，大大提高了读的性能，因此适合读多写少的场景，但是它比较占内存，同时独到的数据可能不是实时数据，所以不适合实时性很高的场景

### 31. Synchronized 的偏向锁 轻量级锁 重量级锁

（1）偏向锁：在锁对象的对象头记录一下当前获取到锁的线程id，该线程如果下次又来获取该锁能够直接获取到。
（2）轻量级锁： 由偏向锁升级而来，当一个线程获取到锁以后，此时这把锁是偏向锁，当第二个线程来竞争的时候，偏向锁就会升级成轻量级锁，轻量级锁底层采用自旋的方式等待获取锁，不会造成线程阻塞。
（3）如果自选次数超过一定次数仍然没有获取到锁，则会升级成重量级锁，会造成线程阻塞。

### 32. 分布式锁加锁方案

目前主流的分布式锁的实现方案有两种：

1. zookeeper：利用的是zookeeper的临时节点、顺序节点、watch机制来实现的，zookeeper分布式锁的特点是高一致性，因为zookeeper保证的是CP，所以由它实现的分布式锁更可靠，不会出现混乱
2. redis：利用redis的setnx、lua脚本、消费订阅等机制来实现的，redis分布式锁的特点是高可用，因为redis保证的是AP，所以由它实现的分布式锁可能不可靠，不稳定（一旦redis中的数据出现了不一致），可能会出现多个客户端同时加到锁的情况

### 33. MySql: 索引覆盖是什么

索引覆盖就是一个sql在执行时，可以利用索引来快速查找，并且此sql要查询的字段在当前索引对应的字段中都包含了，那么就表示此sql走完索引后就不用回表了，所需要的字段都在当前索引的叶子结点上存在，可以直接作为结果返回。

### 34. 最左前缀原则是什么

当一个sql想要利用索引时，就一定要提供该索引所对应的字段中最左边的字段，也就是排在最前面的字段，比如针对a，b，c三个字段建立一个联合索引，那么在写一个sql的时候就一定要提供a字段作为条件，这样才能用到联合索引，这是由于在建立a，b，c三个字段的联合索引时，底层的B+树是按照a,b,c三个字段从左往右去比较大小进行排序的，所以想要利用B+树进行快速查找也得符合这个规则。实际上就是搜索数据的时候sql中的条件和根据联合索引查找的时候，会首先对a进行匹配，然后对b进行匹配，然后对c进行匹配。

### 35. innodb 如何实现事务的

Innodb通过Buffer Pool，LogBuffer，Redo Log，Undo Log来实现事务，以一个update语句为例：

1．Innodb在收到一个update语句后，会先根据条件找到数据所在的页，并将该页缓存在Buffer Pool中
2．执行update语句，修改Buffer Pool中的数据，也就是内存中的数据
3．针对update语句生成一个RedoLog对象，并存入LogBuffer中
4．针对update语句生成undolog日志，用于事务回滚
5．如果事务提交，那么则把RedoLog对象进行持久化，后续还有其他机制将Buffer Pool中所修改的数据页持久化到磁盘中
6．如果事务回滚，则利用undolog日志进行回滚

## 36. MySql的锁有哪些

按照锁的粒度：
（1）行锁： 锁某一行数据，锁粒度最小，并发度最高
（2）表锁：锁整张表，锁粒度最大
（3）间隙锁：锁的是一个区间

还可以分为：
共享锁： 也就是读锁，一个事务给某行加了读锁，其他事务也可以读，但是不能写
排他锁： 也就是写锁， 一个事务给某行加了写锁，其他事务不能读，也不能写

还可以分为：
乐观锁：不会锁定某行记录，而是采用CAS和版本号的机制来实现，给某行添加一个版本，用于观测数据是否改动
悲观锁：行锁，闭锁，间隙锁等都是悲观锁。

## 37. 并行和并发的区别

并发和并行的主要区别在于：1.处理任务不同；2.存在不同；3.CPU资源不同；

一、处理任务不同
并发(Concurrent)
并发是一个CPU处理器同时处理多个线程任务。（宏观上是同时处理多个任务，微观上其实是CPU在多个线程之间快速的交替执行CPU把运行时间划分成若干个（微小）时间段，公平的分配给各个线程执行，在一个时间段的线程运行时，其他线程处于挂起状态，这种就称之为并发。）

并行（parallel）
并行是多个CPU处理器同时处理多个线程任务。（当一个CPU执行一个线程时，另一个CPU可以执行另一个线程，两个线程互不抢占CPU资源，可以同时进行，这就被称之为并行。）

二、存在不同

并发(Concurrent)
并发可以在一个CPU处理器和多个CPU处理器系统中都存在。（多个CPU处理器系统其中的一个CPU也可以进行并发操作）

并行（parallel）
并行在多个CPU处理器系统存在。

三、CPU资源不同

并发(Concurrent)
并发过程中，线程之间会去抢占CPU资源，轮流使用。（其实CPU会多个各个线程公平的分配时间片和进行执行。）

并行（parallel）
并行过程中，线程间不会抢占CPU资源。（因为是多个CPU处理器，各做各的。）

38. **Bean的生命周期**

https://blog.csdn.net/a910247/article/details/125286156
**从对象的创建到销毁的过程。**而Spring中的一个Bean从开始到结束经历很多过程，但总体可以分为**六个阶段Bean定义、实例化、属性赋值、初始化、生存期、销毁。**

39. **kafka生产者和消费者如何确认消息有没有丢失**

生产者：kafka， ack机制确认消息发送成功
消费者： 通过消息的偏移量（offset），确认消费消息没有丢失

40. **泛型为啥不能传基础数据类型**

（1）泛型要求包容的是对象类型，int是基本数据类型，基本数据类型不是对象，所以不能被转化
（2）基本数据类型（如int、double）的封装类对象类型(如Integer、Double)，可以作为泛型包容对象

泛型<...>会在编译期间进行泛型擦除（泛型擦除：将原来的数据类型变为Object类型），刚好int、double等基本数据类型的父类不是Object类型（它们本身也没有父类），所以转化失败

问题：为什么Object a=1;没有报错，是因为java编译器帮你调用了Integer.valuOf(1);将int类型转化成了Integer类型

41. **缓存淘汰策略**

https://blog.csdn.net/weixin_44259720/article/details/122995882
（1）FIFO
FIFO（First in First out），先进先出，这个概念在队列 Queue 里也提到过，它的核心原则就是：如果一个数据最先进入缓存中，则应该最早淘汰掉。

最常见是实现是使用一个双向链表保存数据：

- 新访问的数据插入FIFO队列尾部，数据在FIFO队列中顺序移动；
- 如果Cache存满数据，则把队列头部数据删除，然后把新的数据添加到队列末尾；
- 在访问数据的时候，如果在Cache中存在该数据的话，则返回对应的value值，否则返回-1。

（2） LRU

LRU 表示以时间作为参考，淘汰最长时间未被使用的数据。其核心思想是：如果数据最近被访问过，那么将来被访问的几率也更高。
LRU 也是最常用的淘汰策略，在 Redis 中不管是 allkeys-lru 和 volatile-lru，其实底层原理都一样。
最常见的实现是使用一个链表保存缓存数据：

- 新访问的数据插入到链表头部；
- 每当缓存命中（即链表数据被访问），将该数据从新插入到链表头部；
- 当链表满的时候，将链表尾部的数据（不常用数据）丢弃。

（3）LFU

LFU 表示以次数为参考，淘汰一定时期内被访问次数最少的数据，其核心思想是：如果数据过去被访问多次，那么将来被访问的频率也更高。

- 新加入数据插入到队列尾部（引用计数初始值为 1）；
- 当队列中的数据被访问后，引用计数 +1，队列按次数重新排序；
- 当需要淘汰数据时，将排序的队列末尾的数据（访问次数最少）删除。

42. **有了cas 为什么还要有sychronize， 各自的适用场景**

对于资源竞争较少（线程冲突较轻）的情况（适合cas）：
使用`synchronized`同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。
对于资源竞争严重（线程冲突严重）的情况（适合synchronized）
CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。

cas适合控制单个变量的并发修改，适用于简单的单值修改场景；

synchronized适合需要控制复杂的对象的并发修改的场景（比如期望一个对象内很多变量在线程执行任务期间不会被别的线程修改）

43. **消息队列适用的业务场景**

https://www.zhihu.com/question/588645249/answer/2929824409
实现系统解耦（将相关联的系统模块拆分出来，通过消息队列才完成参数传递和任务命令传递）
异步处理（通过监听消息队列的方式获取之前提交的任务的反馈，进行进一步处理）
缓冲机制（当突然来了大量请求的时候，能缓存没来得及处理的请求，实现缓冲）
日志收集： Java消息队列可以将分散在各个节点上的日志数据进行收集和整合，方便后续的日志分析。

44. **concurrentHashmap put函数原理**

**concurrentHashMap的底层就是一个hash槽数组+cas+sychronize+链表+红黑树**
（1）首先检查hash槽是不是空的，如果是空的直接创建一个节点，采用cas操作把该节点设置到这个槽
（2）如果hash值对应的hash槽不是空的，那么就采用对has槽加锁，然后首先找到hash槽的链表或者红黑树中有没有对应的节点，有就直接修改，没有就创建一个节点采用尾插法插入。

45. TreeMap

TreeMap（按照顺序存储元素的map）不会像hashMap一样简单的根据key的hash值和其对象本身来保存不同的key和value值，

他会根据传入的comparator接口进行比较，然后如果两个node比较起来是相同的就会替换掉之前的。

46. **如何检查线上问题， 线上突然cpu爆满，如何查看定位哪个线程或者哪个对象消耗了很多资源**

https://www.dbs724.com/340055.html

top命令是Linux上最常用的监视系统资源使用情况的工具之一。该命令可以不间断地显示系统资源的使用情况，包括CPU使用率、内存使用率、进程数量、进程占用CPU的百分比等。 运行top命令后，它会自动按照CPU占用率的降序排列所有进程，并在最顶部显示系统总体的CPU和内存使用情况。同时，它还会显示每个进程的编号、占用CPU和内存的百分比等信息。

vmstat命令是Linux系统监视资源使用情况的另一条命令。该命令可以实时监视系统的IO、内存、CPU等方面的使用情况。vmstat命令输出的信息中包括了每秒产生的上下文切换次数、内存使用情况、磁盘IO活动情况、CPU使用情况等内容。这些信息可以帮助我们了解系统的整体运行情况

free命令可以显示系统内存的使用情况。该命令可以显示可用内存、已用内存和交换空间的使用情况。但需要注意的是，free命令显示的“已用内存”数量并不一定等于系统的实际内存使用量。因为Linux系统往往会将一部分未被占用的内存留作缓存，以提高系统性能。

ps命令用于查看系统运行的进程信息。可以用它来查看某一进程的 CPU、内存占用情况等。使用ps命令时，可以指定进程PID、用户、内存占用率等参数，来实现进程信息的筛选和查找。该命令还可以结合其他命令，如grep，实现更精确的进程查找和筛选。

sar命令是一款强大的系统性能分析工具。该命令可以监视系统的CPU、内存、磁盘IO等资源的使用情况，并生成报告。

（1）首先使用top vmstat free ps sar命令定位使用cpu和内存排名靠前的引用程序，找到耗资源最多的java进程；
（2）然后生成下载线上部署的服务器应用java虚拟机中dump文件，然后用vidualVm工具分析对象使用情况，垃圾回收情况，线程运行情况等。

47. **redis 为什么高性能**

https://baijiahao.baidu.com/s?id=1708807538121555902&wfr=spider&for=pc

**基于内存实现**

数据都存储在内存里，减少了一些不必要的 I/O 操作，操作速率很快。

**高效的数据结构**

底层多种数据结构支持不同的数据类型，支持 Redis 存储不同的数据；
不同数据结构的设计，使得数据存储时间复杂度降到最低。

**合理的数据编码**

根据字符串的长度及元素的个数适配不同的编码格式。

**合适的线程模型**

I/O 多路复用模型同时监听客户端连接；
单线程在执行过程中不需要进行上下文切换，减少了耗时。

### 48. redis的数据类型有哪些

String Hash List Set Zset

### 49. java线程池的创建方式

1. 使用Java自带的Executors工具类：通过调用Executors中的静态方法来创建线程池，例如newFixedThreadPool、newCachedThreadPool等。

2. 手动创建ThreadPoolExecutor：可以使用ThreadPoolExecutor的构造函数来手动创建线程池，设置线程池的核心线程数、最大线程数、线程存活时间、任务队列等属性。

3. Spring框架中的任务调度：可以使用Spring框架中的@Scheduled注解来定时任务，在注解中指定cron表达式或者固定的间隔时间，框架会自动创建线程池来执行任务。

4. 使用第三方工具类：可以使用一些开源的线程池工具类，如Guava、Apache Commons等，这些工具类提供了简单的API，使用起来非常方便。

### 50. 线程池参数设置的公式， 按照什么设置线程池参数

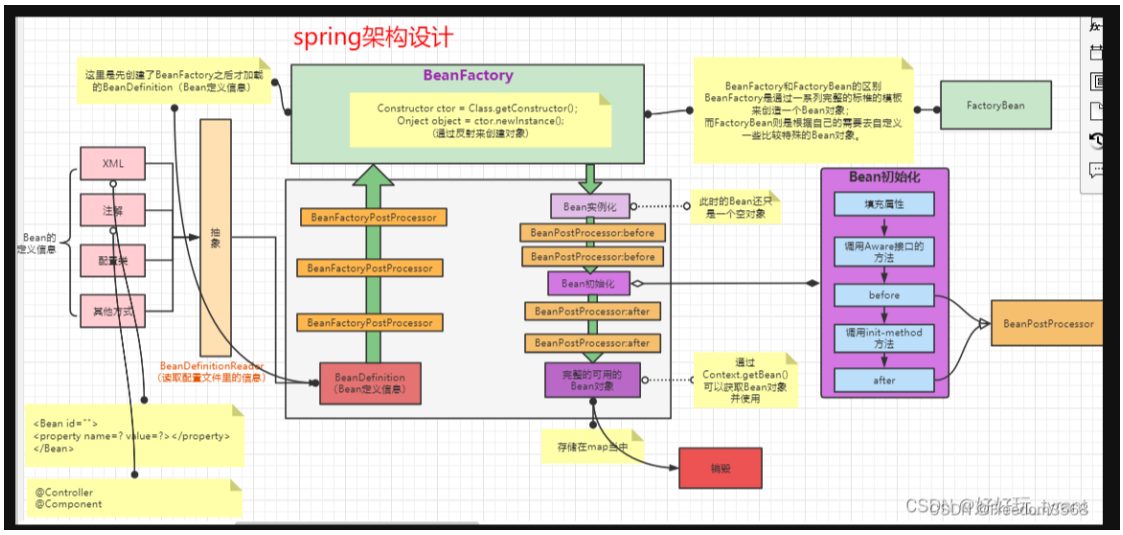
```
计算公式分为:
CPU密集型（CPU-bound）
线程数一般设置为：
线程数 = CPU核数+1 (现代CPU支持超线程)

IO密集型（I/O bound）
线程数一般设置为：
线程数 = （（线程等待时间+线程CPU时间）/线程CPU时间 ）* CPU数目
```

### 50. Spark 两个计算流并表的原理

### 50. Spring bean初始化和创建的相关函数和理解

https://blog.csdn.net/zhanggqianglovec/article/details/126159068

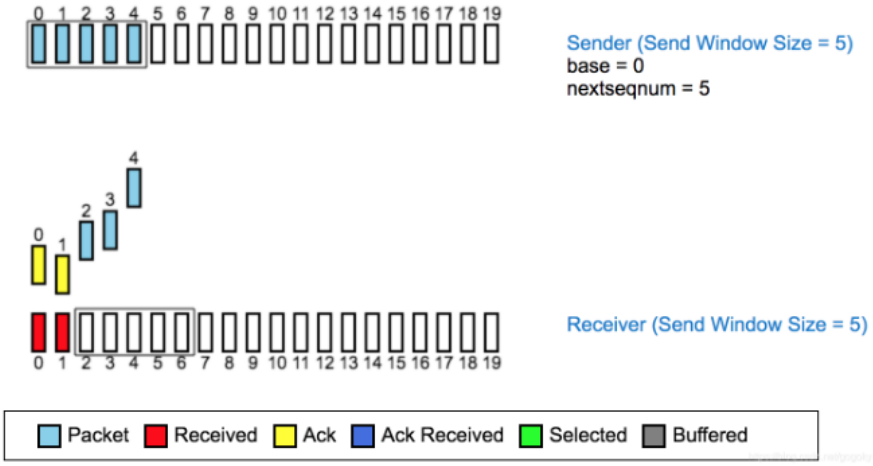



## 50. 限流 降级的方案

限流和降级目的： 当流量快速增长的时候，一定要保证核心服务的可用，即便是有损服务。方案包括： 有限重试， 快速失败， 降级限流方案。
对于高并发C端业务系统，一般都会采取相应的手段来保护系统不被意外的请求流量压垮，进行服务的降级处理， 包括熔断和限流:

熔断：类似于电流的保险丝机制，当我们的服务出问题或者影响到核心服务流程的时候需要暂时熔断某些功能+友好的提示，等高峰或者问题解决后再打开;
限流：当流量快速增长、防止脉冲式流量导致服务可能会出现问题(响应超时)，或者非核心服务影响到核心流程时， 仍然需要保证服务的可用性， 即便是有损服务。
缓存：目的是提升系统访问速度和增大系统处理的容量，可以说是 抗高并发流量的银弹;

限流： 限流的目的是防止恶意请求流量、恶意攻击、或者防止流量超过系统峰值。通过对并发访问/请求进行限速或者一个时间窗口内的请求进行限速来保护系统。
限流方案：
a. 滑动窗口算法： 采用滑动窗口来处理窗口内部的请求，窗口外部的请求则需要排队等待有资源空闲的时候，窗口才会开启新的线程处理。窗口内部请求的状态一般是要么完成，要么进行中，要么刚提交，完成的请求最后会滑到窗口外部，接着就可以处理后续的请求了

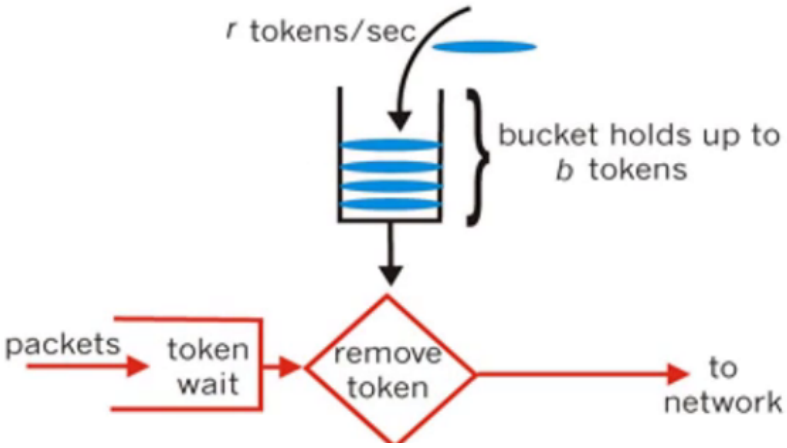



b.漏桶算法：
桶本身具有一个恒定的速率(接口的响应时间)往下漏水，而上方时快时慢的会有水进入桶内。当桶还未满时，上方的水可以加入。一旦水满，上方的水就无法加入。桶满正是算法中的一个关键的触发条件(即流量异常判断成立的条件)。而此条件下如何处理上方流下来的水，有两种方式在桶满水之后，常见的两种处理方式为:
1) 阻塞：暂时拦截住上方水的向下流动，等待桶中的一部分水漏走后，再放行上方水;
2) 抛弃：溢出的上方水直接抛弃，执行拒绝策略;


c.令牌桶算法：
令牌桶算法是网络流量整形(Traffic Shaping)和速率限制(Rate Limiting)中最 常使用的一种算法。典型情况下，令牌桶算法用来控制发送到网络上的数据的数目， 能解决突发请求的流量；令牌桶是一个存放固定容量令牌(token)的桶，按照固定速率往桶里添加令牌; 令牌是按一定的速率来生成；1个令牌/10ms；
令牌桶算法实际上由两部分组成:

**两个流：**分别是 **令牌流、数据流**

**一个桶： 令牌桶**




## 50. 同一个thread对象被启动两次，会有啥问题

会抛出IllegalThreadStateException的运行时异常

## 50. kafa 消息水满原理

kafka中HW（High Watermark）有两个作用

一是用来表示哪些消息可以被消费者消费，相当于分界线

二是帮助kafka完成副本的同步

位移值小于高水位的是已提交消息，可被消费者消费，大于等于高水位的消息，属于

未提交消息，不可被消费者消费

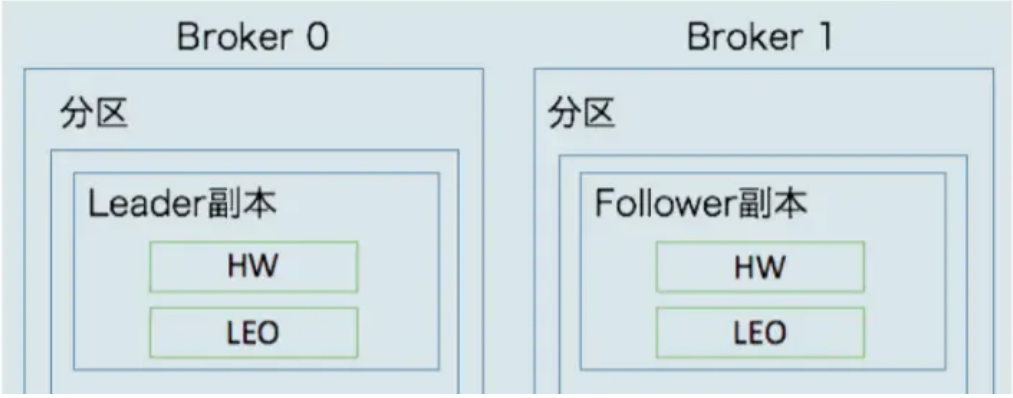
Log End Offset 日志末端位移，LEO是表示副本写入下一条消息的位移，介于高水位

和LEO之间的消息就是未提交消息，所以同一个副本中，高水位是不会超过LEO的

Kafka 使用 Leader 副本的高水位来定义所在分区的高水位。换句话说，分区的高水位就是其

Leader 副本的高水位

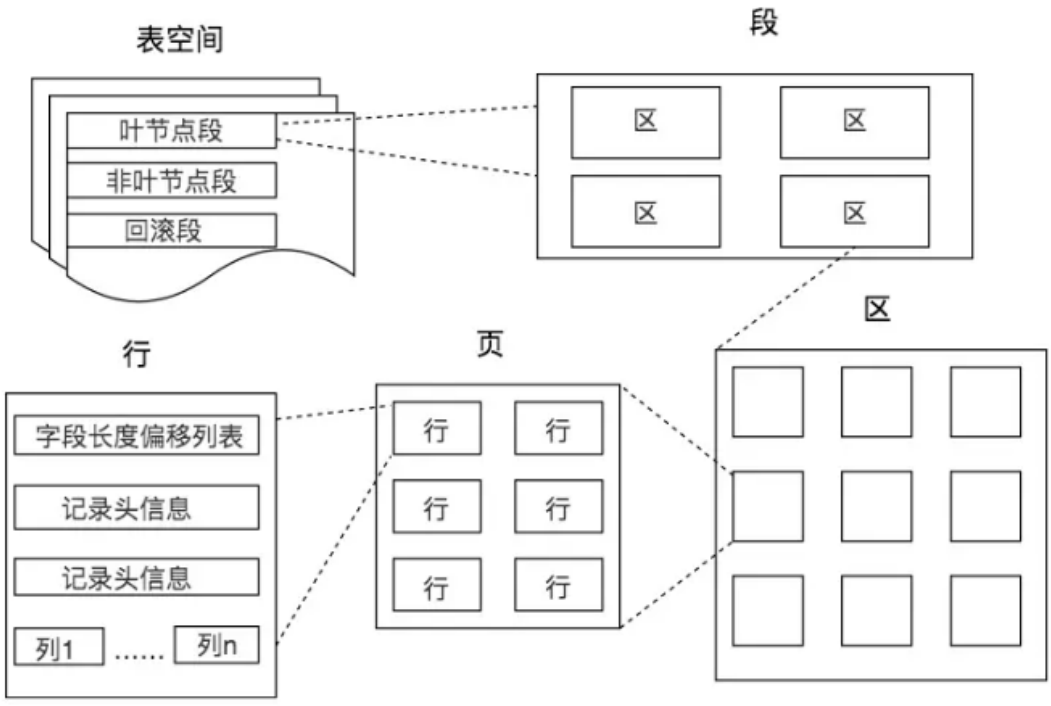
**HW更新机制**



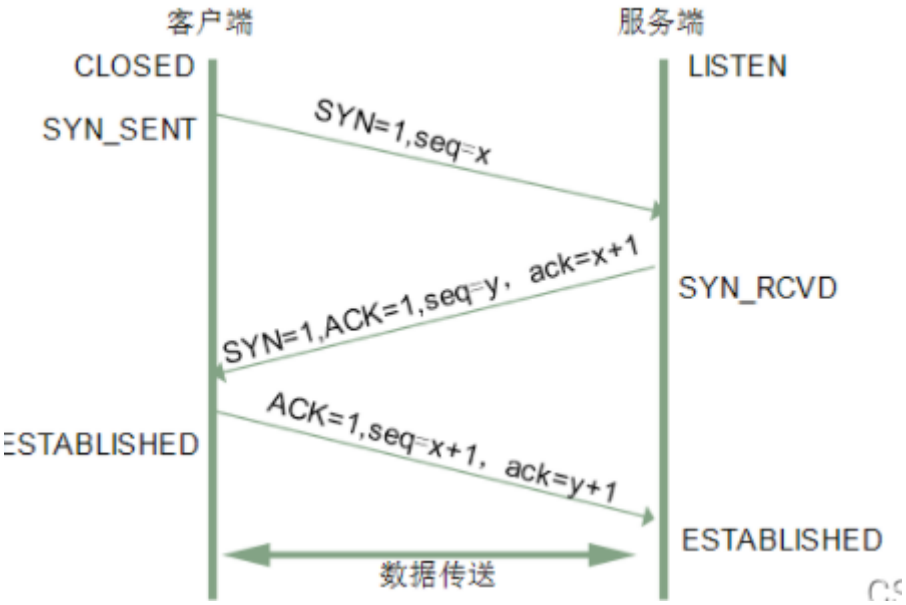

50. **数据库分页和计算机系统分页的区别**

操作系统的页实际上是由字组成的一定大小的连续地址区块，对应到的是系统底层 的内存加载粒度。

数据库页（逻辑概念，也就是数据库的页只是连续行数据逻辑上分为一页的概念，如果数据库页的大小是16k，操作系统底层一页是4k，读取一页的数据库页，就会加载相当于4页的操作系统内存块）：
https://www.jianshu.com/p/b3a3d0b78b01
记录是按照行来存储的，但是数据库的读取并不以行为单位，否则一次读取（也就是一次 I/O 操作）只能处理一行数据，效率会非常低。因此在数据库中，不论读一行，还是读多行，都是将这些行所在的页进行加载。也就是说，数据库管理存储空间的基本单位是页（Page）。
一个页中可以存储多个行记录（Row），同时在数据库中，还存在着区（Extent）、段（Segment）和表空间（Tablespace）。行、页、区、段、表空间的关系如下图所示：




50. **tcp三次握手**




第一次握手：标志位SYN = 1，随机生成一个序列号seq1 = x
第二次握手：标志位SYN ,ACK = 1，确认号ack = x + 1，随机生成一个序列号seq2=y
第三次握手：标志位ACK = 1, 确认号ack = y + 1，seq2= x + 1
https://blog.csdn.net/qq_42224683/article/details/124156907

50. **redis集群选主机制，主从同步机制**

50. **jvm进程和操作系统进程的区别**
51. **线程池拒绝策略**
52. **kafa如何处理消息积压**
53. **redis 缓存高效的原因**
54. **静态工具类如何注入bean**
55. **spring boot项目启动的时候如何设置监听事件**
56. **最长回文子串**
57. **bean注入的方式**
58. **双亲委派模型的原理好处，然后如何重写双亲委派，可以截断双亲委派的机制吗**
59. **有哪些类加载器**
60. **深拷贝的集中方式**
61. **mysql事务实现的原理**