注意事项？

不好理解的概念让chat通俗易懂地解释

举例子，比喻更容易理解概念

可以作为学习笔记，通过问题形式记录

无论大小，都可以记录

可以先把问题记录，写在其它笔记软件上

学任何东西都可以在这里对自己提问

要单独做笔记的可以单独记录

充当随记的功能，记录一些有价值的信息，里面包含了自己的一些思考等

都要以提问题的形式：是什么、怎么样、为什么、凭什么…

以问号结尾

锻炼自己问问题的能力，能够提出问题，有自己的想法，如果要单独做笔记可以单独去做，不影响这个

学习一定要输出，自己能提出问题，然后回答，输出才会有印象，也可以去网上找面经和面试题拓宽思路。

什么是消息队列？

消息： 跨进程传递的数据

队列：一种先进先出的数据结构

什么是中间件？

中间件是软件

提供系统软件和应用软件之间的连接

提供软件各部件之间连接

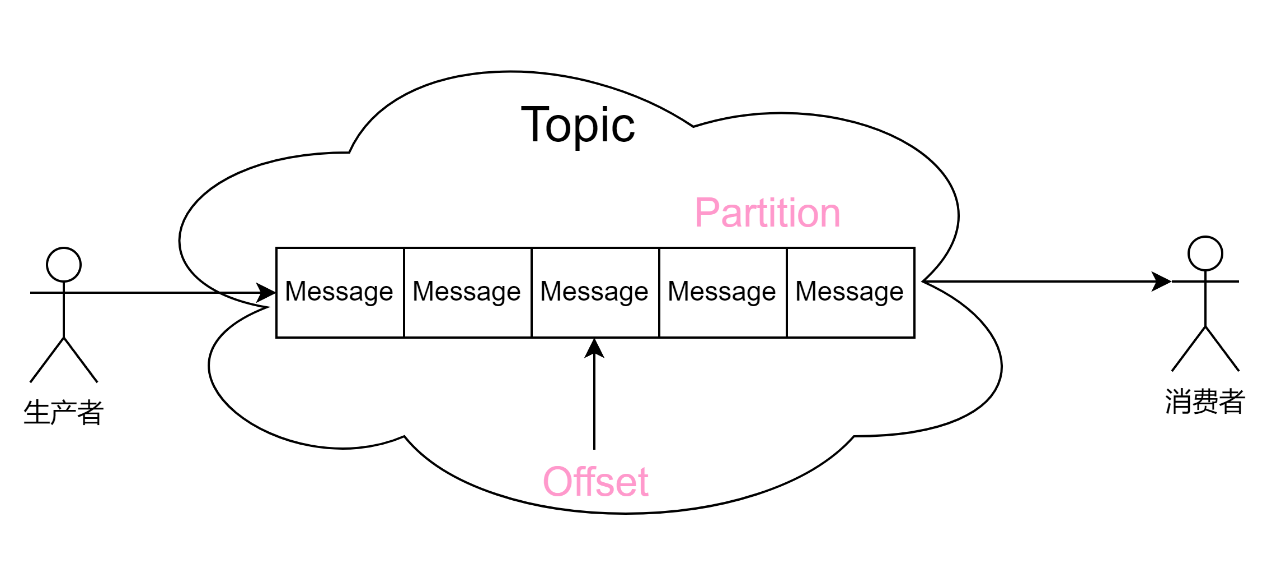
kafaka的partition是什么？

1 kafka 有 topic，生产者和消费者根据topic来确定往哪里发送消息，接受消息

2 topic下面有不同的partition，一个partition代表一个队列

3 offset 表示在partition里元素的偏移量

partition是一个数据结构



消费组用什么作用？

消费者A 属于 消费组1

消费者B 属于 消费组1

一个消息进来之后，如果消费者A消费了，消费者B就不会消费，隶属与同一个消费组的消费者同一个一个消息只会消费一次

Elasticsearch中的文档有什么特点?

1. 不同文档的字段数量和名称可以不一样。文档可以有不同结构
2. 不同文档中同名的字段，类型必须一样。相同字段必须是相同类型

AO/PO/OO分别代表什么？

面向切面

面向过程

面向对象

如何构建自己的笔记系统？

1. 改变认知：就是要开多个窗口，存放不同的信息。
2. 信息是有价值的，不同价值的信息放在不同窗口里面。

习惯开多窗口，在不同窗口间进行切换

不要怕窗口开的多，说明分类很齐全，各种笔记软件结合用，不同价值信息放在不同的窗口里面

不同软件平台，不同窗口，配合起来使用

低价值信息 – 放进流水账里

一些疑问，问题 – 放进多问为什么

单独做的笔记 – 放进CSDN

|  |  |  |  |
| --- | --- | --- | --- |
|  | 低价值信息 | 中等价值信息 | 高价值信息 |
| 放在哪里 | 流水账.md | 多问为什么.docx | CSDN |
| 内容 |  | 随记、疑惑、学习心得  以提问题的形式记录 | 单独做的笔记思考 |
| 是否同步 | 不用同步 | git同步 | 云同步 |

要对信息有一个分类，然后记录在不同的文件、软件、窗口里面，进行分流

可以不断改变这套打法，目前的打法是这样

核心：把遇到的信息进行分类归纳，放在不同的地方

关键：习惯开很多个窗口，进行信息流程，窗口多开一点，更好做各种笔记等

窗口开的越多越好，习惯多开窗口，复杂地做笔记

要习惯开很多窗口，在不同窗口进行切换，这样可以更好地进行信息流转，任务栏的窗口越多越好，窗口开了就不要随便关，就打开就是，把常用的软件固定到任务栏。

信息有价值

窗口越多越好

窗口打开了就不要关闭

任务栏软件铺满最好

要适应复杂，不要太洁净，不然不好做复杂的事情

操作习惯改变

面对复杂工程、笔记等才可以游刃有余

主动变得复杂

什么是消息？

是传输信息的载体，是生产和消费数据的最小单位

Topic、消息、生产者、消费者、生产组、消费组之间的关系？

1. 一个Topic可以有多条消息，一条消息只能属于一个Topic
2. 一个生产者可以生产多个Topic的消息
3. 一个消费者只能订阅和消费一个Topic的消息，一个Topic里的消息可以被多个不同消费组里的消费者消费

RocketMQ里的基本概念和Kafka里的基本概念总结？

RocketMQ：消息、Topic、标签、队列、分片、消息标识、消费组

Kafka： 消息、Topic、 、分区、 、 、消费组

什么是分布式锁？

在分布式系统或集群模式下，满足多进程可见并且互斥的锁

讲一下SETNX 和EXPIRE

setnx key value

expire key timevalue

如何保证setnx和expire同时成功、同时失败？

set key value nx ex timevalue

这样就可以保证原子性操作

为什么要用分布式锁？

因为普通的锁只能对同一个JVM的线程进行互斥

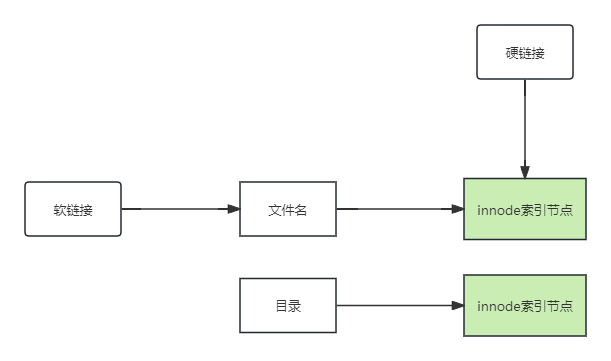
如何创建一个符号链接？

符号链接，又被称为软链接，soft link

link -symbolic source.txt linktosource.txt

为什么硬链接不能链接目录?

因为可能导致循环引用



如何查看一个日志？

tail -f

tail 会默认显示文件后10行内容

-f follow，会监视文件的变化，跟随

如果文件有新的内容，会打印在控制台

grep 参数 日志文件名

-i ignore 忽略大小写

-C context 上下文

Linux 系统中符号链接与硬链接的区别？

|  |  |  |
| --- | --- | --- |
|  | 符号链接 | 硬链接 |
| 链接类型 | 指向目标文件路径的快捷方式 | 指向目标文件innode的另一个文件名 |
| 是否指向innode | 不指向innode，指向文件路径 | 直接指向文件innode |
| 跨文件系统 | 可以 | 不可以 |
| 是否可以链接目录 | 可以 | 不可以 |
| 目标文件删除影响 | 符号链接会变成断链 | 没有影响，一个文件只有所有硬链接被删除，这个文件才会被删除 |



当Linux主机的内存不够的时候，如何使用swap增加内存？

可以写一篇

常见的Linux目录结构？

/ 根目录

/bin 用户二进制文件 ls cp命令

/sbin 系统二进制文件 system binaray fdisk systemctl命令

/etc 配置文件目录，etc.. 诸如，等等

等等，及其他（et cetera）

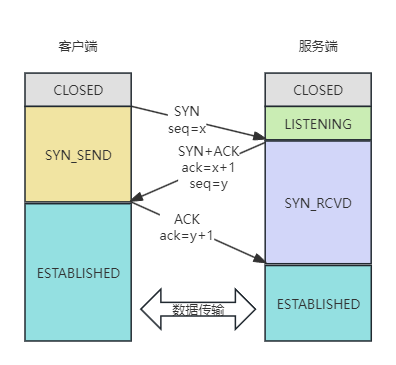
ps aux命令解释

process status all user exclude

netstat -napt命令解释？

network statistics -numerical -all -program -tcp

说一下三次握手的大致过程？



* 首先，客户端和服务端都处于CLOSED状态，服务端启动某个进程，监听某个端口，处于LISTENING状态。
* 客户端向服务端发起连接，发送SYN报文，并发送一个初始序列号ISN，客户端进入SYN\_SEND状态。
* 服务端接收到客户端发来SYN报文，同样向客户端返回SYN+ACK报文，以及服务端的初始序列号ISN，和确认号ISN+1，表示前面ISN的序列号的数据都已经正常接收。服务端进入SYNC\_RCVD状态。
* 客户端接收到服务端发来的SYN+ACK报文后，进行第三次握手，也就是最后一次握手，发送ACK确认报文，确认号为服务端发送的序列号加1。客户端进入ESTABLISHED状态，该ACK报文可以包含数据，也可以不包含数据。
* 服务端接收到客户端发送的ACK报文后，也进入ESTABLISHED状态。

ISN的英文全称？

Initial Sequential Number

序列号

为什么是三次握手而不是两次握手？

1. 阻止历史重复连接的初始化。

假设这样一个场景，客户端发送SYN报文到服务端准备建立连接，由于网络阻塞，第一次的SYN包滞留在网络中没有到达服务端，于是客户端重新发送了一个SYN报文，进行连接的初始化，但第一次发送的旧的SYN报文先于最新的SYN报文到达了服务端。如果只有两次握手，服务端接收到这个旧的SYN报文就会返回一个SYN+ACK报文，进入连接状态，但此时客户端和服务端建立连接的是一个旧的初始化连接，服务端返回的确认号并不是客户端想要的序列号，所以会造成数据混乱。如果使用三次握手，第二次握手后，服务端并不会进入连接状态，而是进入SYN\_RCVD状态，此时客户端根据上下文判断服务端发过来的确认号并不是自己想要的，所以会发送RST报文进行连接的中断。(Reset)

1. 同步双方之间的初始序列号。

如果只有两次握手，客户端发送SYN和自己的初始化序列号之后，服务端返回SYN+ACK报文，客户端的序列号被服务端同步确认，但是服务端的序列号并没有受到确认号，也就无法进行同步。

1. 避免资源浪费。

客户端发送第一次SYN报文，在网络中阻塞了，然后重新发送了一段SYN报文与服务端建立连接，双方传输完数据后通过四次挥手断开连接。此时在网络中阻塞的SYN报文到达服务端，由于只有两次握手，服务端接收到客户端的SYN报文后向返回SYN和确认报文，然后进入连接状态，但此时客户端以及关闭，所以服务端迟迟收不到客户端的回应，会一致处于连接状态，造成资源的浪费。

第三次握手可以携带数据吗？第二次握手呢？

第三次可以携带数据

如果第三次的ACK报文丢了，后面传输数据的报文，由于也携带ACK，会被服务端当作第三次握手

第一次不能携带数据的原因：防止恶意握手请求

第二次不能携带数据的原因：因为服务端通过第一次握手只确认了客户端的数据发送能力，但是还没有确认数据接收能力，所以不能携带数据。

三次握手中最后一次ACK包丢失会发生什么？

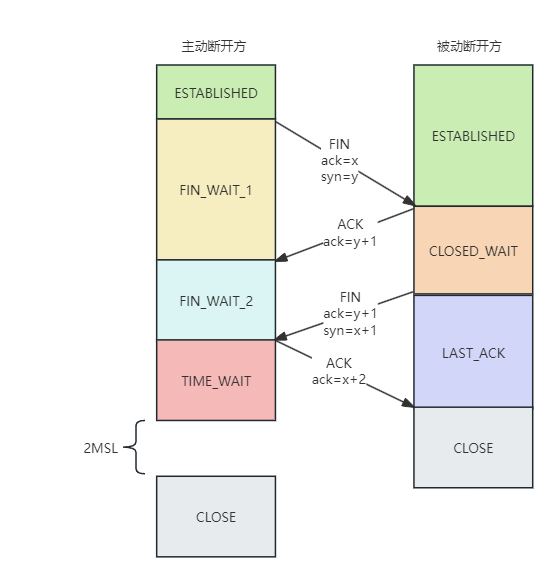
这要分情况看:

1. 后续数据包带有ACK, 服务端汇报数据包当场第三次握手。
2. 后续数据包没有ACK，服务端重新发送第二次握手的SYN+ACK报文，此时客户端会重新发送ACK报文。

为什么不是四次握手？

因为客户端向服务端发送SYN报文和ACK报文的两次握手被合并到了.

讲一下四次挥手的过程?



断开连接分为主动断开方和被动断开方，假设客户端是主动断开方，则:

* 客户端你想服务端发送FIN报文，进入FIN\_WAIT\_1状态。
* 服务端接收到FIN报文后知道客户端想要断开连接了，会返回客户端一个ACK确认报文，然后进入CLOSE\_WAIT状态，此时客户端已经没有数据要发给服务端了，但是服务端可能有一些还没有处理完的数据要发给客户端。
* 服务端处理完所有的数据后，向客户端发送FIN报文，进入LAST\_ACK状态。
* 客户端收到服务端的FIN报文后，向服务端发送ACK确认报文，服务端接收到客户端的确认报文后进入CLOSE状态。客户带进入TIME\_WAIT状态
* 客户端等待2MSL时间后进入CLOSE状态。

什么是MSL？

Maximum Segment Lifetime

报文最大生存时间

为什么等待2MSL时间?

MSL是报文在网络中的最大存货时间，从客户端发送最后一个ACK报文开始算，如果出现故障，等到服务端发送重传报文或者其它中断报文最多等待2MSL的时间。

网络中可能存在主动关闭方的数据包，这些数据包在被动关闭方接收后会响应给主动关闭方，这一来一回所以最多要等到2倍的MSL时间。

为什么需要挥手四次?

因为服务端的ACK和FIN报文不能合并到一起发送，因为在收到客户端关闭连接请求时，此时虽然从客户端到服务端已经没有数据发送，但是服务端还有数据没处理完需要发送给客户带你，所以服务端会先发给客户带一个ACK确认请求，然后等数据处理完后再发送FIN结束连接的报文。

说一下四次挥手的过程？

主动断开方和被动断开方

客户端首先向服务端发送FIN报文，进入FIN\_WAIT\_1状态

服务端向客户端返回ACK报文，进入CLOSE\_WAIT状态

客户端接收服务端的ACK报文，进入FIN\_WAIT\_2状态

服务端向客户端发送FIN报文，进入LAST\_ACK状态

客户端发送ACK报文，进入TIME\_WAIT 状态

服务端收到客户端的ACK报文，进入CLOSE状态

客户端等待2MSL时间，进入CLOSE状态

为什么需要四次呢？

因为服务端不能讲FIN和ACK合并到一起发送，当第一次握手客户端主动发起连接关闭请求时，服务端有可能还有没有处理完的数据，此时服务端只能向客户端发送ACK确认报文，只有当服务端讲数据全部处理完之后才会想客户端发送FIN报文。

TIME-WAIT 干嘛的，CLOSE\_WAIT干嘛的，在哪一个阶段？

TIME\_WAIT是主动关闭连接方，第四次握手，向被动关闭连接方发送ACK确认报文后进入的状态，此时主动关闭连接方会等待2MSL时间，保证网络中已经不存在这个连接相关的报文后进入CLOSE状态

CLOSE\_WAIT是第二次握手，被动关闭连接方向主动关闭连接方发送ACK确认报文后进入的状态，在该阶段下， 客户端到服务端的数据连接已经关闭，而服务端到客户端的连接还在，服务端还可以向客户端发送数据。

服务端主动发起关闭还是客户端主动发起关闭TCP？

都可以

主动关闭一方会有TIME\_WAIT状态，被动关闭一方会有CLOSE\_WAIT状态。

网络通信的5层模型?

物理层：两台计算机需要通信，物理层面需要用网线、光纤、电话线…等等介质连接起来，传输01电信号。物理层负责把两台计算机连接起来，然后通过高低位电频传输01这样的电信号。

数据链路层

网络层

传输层

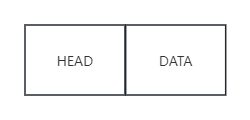
应用层

物理层：负责讲两台计算机连接起来，传输高低电频的01信号，解决两台计算机如何通信的问题

数据链路层，两台计算机通过物理层已经连接，可以传输01这样的bit流，但是这些01bit流是什么意思，如何定义一套规则来识别这些电信号，这是数据链路层要解决的问题，为了解决两台计算机之间通信的bit流识别问题，数据链路层定义了很多协议

以太网协议

帧



由帧头和数据组成

一帧的大小为64-1518个字节，如果传输的数据量很大，需要将数据切分为多个帧进行传输

一台计算机在物理层可以连接多台计算机，那么数据究竟是发给谁的，需要有一个区分的表示，这就是MAC地址

进入网络的每一个计算机都会有网卡接口，每一个网卡接口都会有一个唯一的地址，这就是MAC地址，计算机之间的数据传输，就是通过MAC地址来唯一寻址的，MAC地址一共有48位组成，在网卡生产时就被唯一标识了。

计算机A通过广播机制将数据广播到同一子网的计算机中，收到以太网帧的计算机会取出帧头里面的MAC地址，然后与自身的MAC地址做判断，如果不同则丢弃，如果相同，就会接收该数据。

ARP 地址解析协议，初始时，计算机A是不知道和自己处于同一个子网下的计算机有哪些MAC地址，于是，A会进行广播，并在目的MAC地址做特殊标识，然后同一子网下的计算机就知道需要向这个广播的源MAC地址进行回复。这样，A就获取了同一子网下的不同计算机的MAC地址，对于不同子网的计算机，在传输数据时，会有网关进行转发。

网络层，我们所处的网络是由无数个子网组成的，子网之间的数据传输需要泳道IP协议，我们如何知道哪些IP地址是属于同一个子网，通过子网掩码，将IP地址与子网掩码做and运算，如果结果相同，则说明这两台计算机处于同一个子网中。IP地址由32位组二进制数组成。通过IP地址与子网掩码做and运算，结果非零的部分就是子网。

ARP地址解析协议会广播数据，但数据里面会包含目标主机的IP地址，同一个子网的主机会把数据拿出来，并于自身IP地址做比较，如果相同，就说明A需要我回复自身的MAC地址，此时A的MAC地址我是已知的，所以广播以下，目的MAC是A的MAC地址即可。网络下处理IP地址外，还有DNS域名系统。

网络层建立了主机到主机的通信

传输层建立了端口到端口的通信，数据从计算机A发送到计算机B之后，B计算机上面运行着各种各样的程序，这些数据该给哪个进程，就需要端口号的帮助。

传输层两大协议TCP和UDP

TCP提高可靠传输，UDP提供不可靠传输

应用层，HTPP协议，STAMP协议等都是应用层的协议，用于解析和处理不同格式的数据。

计算机如何知道要把数据发给交换机还是路由器？

判断源IP和目的IP是不是同一个子网，如果是，则交给交换机

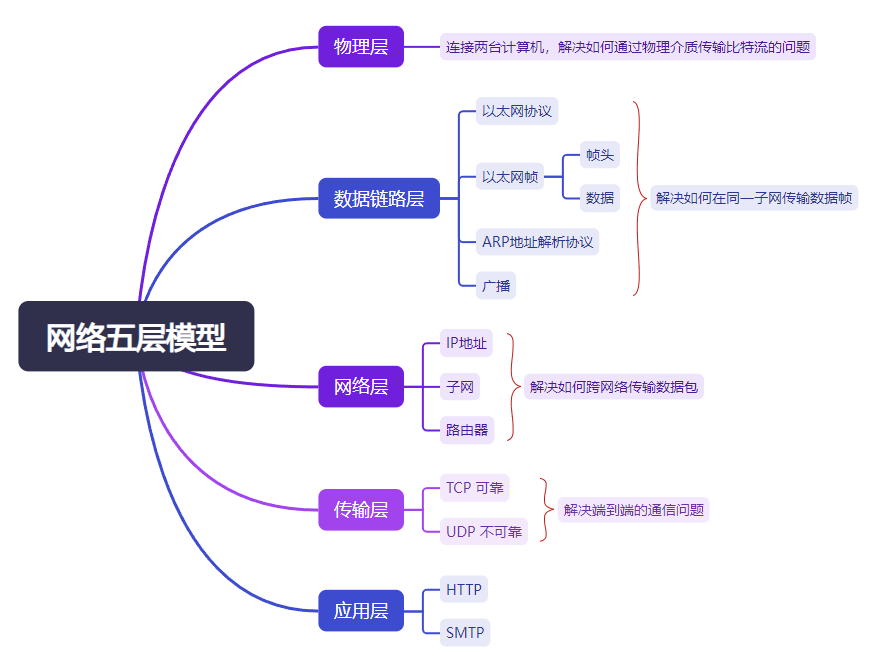
如果不是，则交给路由器

计算机A如何知道哪个设备是路由器?

发给默认网关的IP地址即可

需要在计算机A中配置这样一个网关地址

介绍一下网络五层模型？



为什么要分出四层五层这样的模型呢？

提供了一种结构化的方法来讨论整个网络协议。在实现某一层服务时，只需确保其对上层提供相应的服务，并且可以使用下层服务，至于下层服务是如何实现的，并不关心。

什么是 TCP ？举几个应用了 TCP 协议的例子？

TCP

Transmission Control Protocol

传输控制协议

TCP 是网络五层模型中传输层的协议，TCP是面向连接的、可靠的、基于字节流的通信协议。

面向连接：一对一的连接

可靠的：TCP可以保证一个报文是无损的、无冗余、按序到达的。

基于字节流：消息通过TCP传输时，会会被操作系统分组为多个TCP报文，并且这些报文之间是有序的。

应用层基于TCP的协议：HTTP、FTP、SMTP、SSH

TCP协议如何保证可靠性？

1. 传输通道可靠性：每次连接需要三次握手，释放连接需要四次挥手。
2. 数据的正确性：ARQ（Automatic Repeat Request）自动重传请求。ARQ机制通过序列号、确认ACK和超时重传等手段，保证数据的在传输过程中不丢失、不重复且按序到达。
3. 数据的有序性：序列号和校验和。
4. 整个网络层面：TCP通过慢开始、拥塞避免、快速重传和快速回复来进行拥塞控制，避免网络阻塞带来的数据接收问题。

tcp如何保证不会接受重复的报文？

这是由TCP报文中的序列号保证的，TCP在建立连接时，客户端和服务端都会将一个内核生成的随机数做回初始序列号。

在建立连接完成后，开始发送数据时，发送的一端，会将“上次发送的序列号”加上“上次发送的数据长度”作为“本次报文的序列号”，发送给对端。

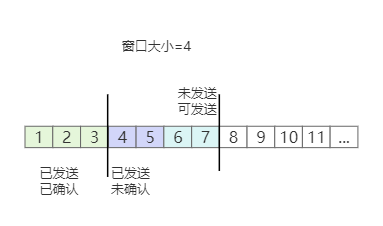
seq = 上次发送的序列号 + 上次发送的数据长度

对端在接收到数据包时，会根据序列号确定接收到的数据的顺序以及是否重复，如果重复收到某个序号的报文，则会进行丢弃，只接受一次。

需要注意的是，在三次握手和四次挥手阶段，由于没有发送数据。发送端的序列号值

seq = “上次发送的序列号” + 1

还有，序列号除了保证数据接收的有序性，还用于TCP流量控制当中。具体的，TCP两端都会维护一个发送窗口与接收窗口，发送方根据TCP报文的序列号和发送窗口大小。来确定可以发送的数据量。接收方使用序列号和接收窗口大小来确定可以接收的数据量。并通知发送方来控制发送速率，以此来确保双方数据可以被顺利被接收。



Tcp粘包拆包问题了解吗？介绍一下，如何解决？

拆包：

TCP 是在传输层面向流的协议，它本身不会按照应用开发者的期望，保持每次发送时都带有一个数据的边界，这可能导致接收端，一次收到了多个应用层报文，需要应用开发者自己分开，也就是需要实现流到数据报的功能。具体的解决：可以用特殊字符作为边界。也就是我们对于应用层的每个信息后面都加上一些特殊字符来代表信息的结束。当读到这些特殊字符，就意味着已经读完了一个完整的消息。比如HTTP协议通过设置一个回车符和一个换行符来作为HTTP报文协议的边界。但需要注意的是，如果信息内容中刚好出现了这些字符，可能会导致信息提前结束和数据错乱。这时，我们可以对这些字符进行转义，避免这种情况。

粘包：

TCP在实现的时候，为了解决大量小报文的场景，包头比数据还大，导致传输的性价比太低的情况，采用了一种叫做Nagle的算法，这个算法的效果是将开发者多次发送的小数据，粘在同一个报文中进行发送。这样可能导致先被发送的数据需要等待一段时间，与后来的数据组成一个完整的报文之后再发送出去。但在实际中这种算法带来的时延几乎可以忽略不计，绝大多数情况都是无感知的，只有在频繁进行超短信息交互，比如只有几个字节的场景，禁用Nagle算法才能显著降低时延。

TCP拥塞控制与流量控制区别？

流量控制：端到端在通信时，发送发和接收方的速率不一致，如果发送方发送速率太快，会导致接收方处理不过来，如果发送方发送的太慢，数据就会有一定的延迟，此时就需要有一个方法来控制发送方的速率，这个方法就是流量控制。

拥塞控制不同于流量控制，主要强调的是整个网络的拥塞，导致发送方的数据包被堵在了半路，而接收方迟迟没有收到数据包并返回接收到的最后一个确认报文，会让发送发误以为这个报文丢失并重新发送，这不仅造成信道资源浪费，还会使原来拥塞的网络雪上加霜。

现在我们视频面试用到哪些协议？

考虑视频对延迟很敏感，但是视频丢一些帧影响却不大，所以应该是具有传输速度快、不可靠的UDP协议。允许一定程度的丢包，但是丢包率不能太高，还应该具备TCP协议有关数据传输可靠性的保证，确保不会视频着视频着，突然画面没有的情况，所以我觉得应该是同时TCP可靠性和UDP速度快的优点，这类协议有谷歌提出的QUIC协议。

什么是 UDP？举几个应用了 UDP 协议的例子

UDP的全称是User Datagrame Protocol 用户数据报协议。它是一种无连接的传输层的协议。没有可靠性保证、顺序保证流量控制。可靠性较差，但正是因为协议控制的自动少，UDP在网络中传输的时延低，数据传输效率高。适合对可靠性要求不高但是对低时延要求高的服务。如果DNS处于性能和效率考虑，使用UDP来提供快速查询和响应。还有比如说基于互联网的电话服务使用VOIP语音，也是用的UDP，可以接受偶尔的丢包，但是时延低。

UDP 与 TCP 有啥区别？

1. TCP是面向连接的，传输数据前需要通过三次握手建立连接，解释传输需要通过四次挥手断开连接，UDP是无连接的，即可传输。
2. TCP是一对一，端到端的服务，UDP是同时支持一对一，一对多和多对多的服务。
3. TCP是可靠性交付数据，数据可以无差错，不丢失，不重复，按需到达。而UDP是尽最大努力进行交互，但不保证可靠性的交互。
4. 由于TCP需要保证数据的完整的交互，所以引入了拥塞控制、流量控制，在传输效率上要低于UDP协议。

DNS 查询到底是怎么完成的？

1. 域名对应的IP地址，都保持在DNS域名服务器

什么是DNS查询?

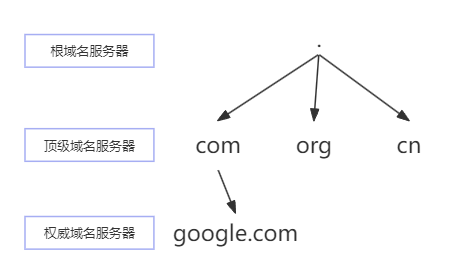
我们在浏览器输入域名地址，浏览器就会在后台，自动向DNS服务器发出请求，获取对应的IP地址。这就是DNS查询。

DNS协议是干嘛用的？

我们通常用IP地址来唯一确认一台上网的计算机，但由于IP地址不方便记忆，也不能显示地表明组织，地址等属性，所以就诞生了域名，比如[www.baidu.com](http://www.baidu.com)，容易记住，也更能体现公司组织等信息。DNS，也就是域名解析协议就是用来将域名和IP地址相互映射，使人们更加方便地访问互联网。

讲一下dns过程，给一个网址www.google.com，dns服务器如何逐级解析的？

首先域名是有层级结构的，从上到下，依次为根域名、顶级域名、二级（权威）/三级域名等等。那么根据域名查询对应的IP地址的过程也需要从上到下，先向根域名服务器查询所有顶级域名服务器的IP地址，根域名服务器的IP地址是固定的， 在通过这个地址向顶级域名服务器查询权威域名服务器的地址，最终返回想要的IP地址结束。



DNS 用的是 TCP 还是 UDP ？

由于DNS的数据包一般情况下比较小，并且需要快速、高效地完成域名解析。大多数情况下都是用UDP。主要还是数据量小，如果丢包了，就重新发送一次就好了。速度很快。并且DNS报文中允许填入序号字段，请求报文和应答报文的序号一样，多个DNS请求之间可以通过序号一一对应。

DNS 解析请求的服务器单机并且响应速度慢怎么办？

一般情况下，DNS服务器本身是没有问题的，有可能是客户端到DNS服务器的网络延迟比较大，这个问题可以通过ping DNS服务器的地址来验证。比如我试过的，正常网络情况下，有时使用Google提供的DNS服务延时100ms，地址为

8.8.8.8，这个时候就可以更换DNS服务器为电信的114.114.114.114来加快速度。除此之外，某些Linux不会配置DNS缓存，配置DNS缓存后，后续请求就更快了。

讲一讲HTTP协议?

HTTP协议是基于TCP/IP的应用层协议。

HTTP是一种请求-响应协议，用于传输HTML文本。

HTTP请求的格式：

响应头

GET / HTTP/1.1

HOST: [www.baidu.com](http://www.baidu.com)，目标域名、主机名

User-Agent: 标识客户端本时

Acccept：浏览器能够接收到的资源类型 \*/\* 标识所以，/左边标识大类型，/右边标识大类型下的小类型。

Accept-Language：浏览器偏好的语言

/r/n

请求体

HTTP响应的格式：

响应头：

HTTP/1.1 200 OK

Content-Type: text/html，表示响应的内容类型

Content-Length: 21932，表示该响应内容的长度，字节数

Content-Encoding：gizp，

状态码，状态描述文本

常见的状态码:

404 表示客户端请求了一个不存在的资源

302 表示该URL需要重定向

200 表示成功

500 表示服务器内部出错

400 Bad Request 表示客户端发送了一个错误的请求，如参数无效

什么是HTTP协议，简单介绍一下?

HTTP全称是超文本传输协议，是一个应用层的协议，通常基于TCP/IP，用于在网络两点之间传输文本，它不是一个孤立的协议，需要其它协议进行配合，最基本的就是TCP/IP，实现可靠数据的传输。还有DNS协议实现域名到IP的查找。

说说http请求报文包含什么？

HTTP请求分为请求头和请求体用一个\r\n回车换行符进行分割

请求头分为起始行和头部字段集合

起始行有请求方法+资源路径+协议版本

头部字段集合使用key-value的形式详细地说明报文，比如:

Host: 请求的域名

User-Agent：浏览器标识

Accept：可接受的文件类型 大文件类型/小文件类型

Content-Length：请求体的长度

Content-Encoding：浏览器可以接收的压缩算法

消息正文就是实际传输的数据

Content-Length知道吗？如果Content-Length的长度 大于报文的实际长度，分析下会有什么问题？

1. Conten-Length 用于指定报文中Body数据的字节长度。
2. 如果对数据本身就进行了gzip压缩的话，Content-Length 指的是压缩后的长度。
3. 对于接受者来说，Content-Length影响着对数据的截取，如果Content-Length的长度比实际的要小，那么只会截取指定的一小部分，损失一部分信息，但大部分情况下页面还会正常显示。如果Content-Length比实际大，接收端在读到消息后会继续等待下一个字节，期间一直处于无响应状态，知道响应超时，此时页面是不能超时显示的。

长链接和短连接介绍一下？

早期的HTTP 0.9和1.1就采用的短链接方式，每次发送HTTP请求都需要与服务器建立连接，收到响应报文后会立即关闭连接。因为客户端与服务器的整个连接过程很短暂，不会与服务器保持长时间的连接状态。所以被称为短链接。

短连接的缺点很明显，对于客户端来说，每次请求都先需要三次握手建立连接，之后四次挥手关闭连接，当请求增多的时候，效率就很低，严重制约了服务器的服务能力，无法处理更多的请求。

根据短链接暴露出来的缺点，HTTP提出了长连接的通信方式，也叫连接保活机制keep-alive，具体的工作方式是：对于相同客户端的多次请求，只需要在最开始时进行一次建立连接，和最后一次连接关闭。在这期间，客户端可以复用已经建立好的连接，来发送其它请求。这样服务的处理效率就高多了。

由于长连接对性能的改善效果十分显著，所以在HTTP/1.1中，默认启用长连接，如果服务端支持长连接，会在响应报文中，Connection：keep-alive 头部来表明

不过长连接也有缺点，如果TCP连接长时间不关闭，服务器就会在内存中保存它的状态。这样会占用大量的服务器资源。所以用了长连接，就必须考虑长连接在何时恰当地关闭。这有两种方法，在客户端中，可以在请求头上加上Connecttion：close字段，告诉服务器这次通信后就关闭连接。也可以在服务器端的Nginx中通过keepalive\_time来设置长连接的持续空闲时间。

要不HTTP1.0，1.1，2.0，3.0 区别也介绍一下？

HTTP的历史最早可以追溯到HTTP/0.9说起，那时的HTTP非常简单，只能用GET方法从服务器获取HTML文档。之后由于多媒体等技术发展，网络传输的数据不止局限于文本，还包括视频、音频等二进制流数据。HTTP/1.0基于此，扩充了传输数据的范围，增加了HEAD，POST等新方法，引入了头部Header等等，形式上已经和我们现在的HTTP差别不大，并没有广泛地应用。真正运用到实际中，并确立统治地位的是HTTP/1.1，它相较于HTTP/1.0又增加了PUT、DELETE等方法，允许长连接，允许响应数据分块等，一直沿用至今。

SSL/TSL是什么？

SSL/TSL是基于TCP/IP的数据传输协议，通过加密算法保证数据的安全性。SSL/TSL的功能:

1. 防止窃听：通过对数据加密
2. 数据完整性：通过消息验证码
3. 身份认证：通过数字证书验证通信双方

什么是HTTPS？

由于HTTP天生明文，缺少通信安全所需的机密性，完整性，身份认证和不可否认四大特性。HTTPS就基于HTTP，在这四个方面上增加

讲一下AOP的实现原理？

Spring的AOP是用动态代理实现的。如果我们为Spring的某个bean配置了切面，那么Spring在创建这个Bean时会直接创建这个Bean的代理对象。后续对Bean方法的调用，实际调用的是代理类重写的代理方法。Spring AOP采用了两种方法实现动态代理，分别是JDK动态代理和CGLIB动态代理。

Spring默认使用JDK动态代理实现AOP，前提条件是要实现了某个接口，然后借助reflect包下的Proxy类和InvocationHandler接口来动态生成代理对象。当我们通过代理对象调用方法时，底层将通过反射，去调用我们实现的代理方法。

正是由于JDK的动态代理存在类必须实现接口的限制，Spring在其它情况下会使用CGLIB动态代理对象。CGLIB实现动态代理的原理是，底层采用ASM字节码生成框架。直接对需要代理类的字节码进行操作，生成这个类的一个子类，并重写类中所有可以重写的方法。在重写的过程中，将我们定义的额外逻辑织入到方法中，对方法进行增强。

两种方式各有优劣，JDK动态代理是JDK原生的，不需要任何依赖即可用。缺点是目标类必须实现了某个接口，才能用JDK动态代理。CGLIB不需要目标类必须实现接口，侵入性小。但CGLIB是通过继承的方式，生成一个目标类的子类作为代理类，因此，假如目标类是final类，也无法使用动态代理，并且CGLIB实现代理方式是重写目标方法，如果目标方法是final private修饰，也无法进行代理。

AOP 用到了 Java 中的哪些技术 ？

AOP在使用时经常用注解来标记切面和切点，首先就用到了注解。

其次，由于AOP使用动态代理实现，Spring又用了两种方式实现动态代理，一种是JDK动态代理，这种方式用到了Java反射。另一种是CGLIB动态代理，主要是通过继承生成目标类的子类。

还有，用到的设计模式也很多，比如代理模式实现动态代理，观察者模式实现通知功能。策略模式实现多种通知类型等等。

Spring AOP中有哪些概念，如何理解?

1. 切面（Aspect），切面 = 通知 + 切点。
2. 连接点，允许应用通知的地方，Spring AOP支支持方法的增强，所以一个方法的前后，异常抛出的地方都可以作为连接点
3. 切点，一个程序有很多很方法，但我并不想每个方法都增强，于是出现了切点，切点定义了哪些方法会被增强，通过切点表达式：返回类型 包.类.方法名(参数列表)，切点决定了在何处
4. 通知，抽取出来的通用功能，通知决定了切面在何时以及如何影响目标方法的执行

IOC AOP 说说自己的看法？

IOC全称是Inverse Of Control，也就是控制反转。他的核心思想在于，资源不由使用资源的双方管理，而是由不使用资源的第三方管理，也就是IOC容器。在Java中，以前我们要使用某个对象，大多数的时候都需要我们new一个出来，而有了Spring IOC之后，Bean都交给了IOC容器统一管理。我们只需要在任何需要的时候到容器中获取即可。IOC使用时有很多优点，比如实现了资源的统一配置和管理，降低了使用资源双方的耦合度。

AOP是一种面向切面编程的思想，也就是通过将系统分解成不同的切面从而使系统中不同的切面可以独立地进行开发和维护。

比如项目中的日志系统，把日志记录的操作写在每个方法中，这样太臃肿和难以维护，每添加一个方法都要重新写记录日志的方法，要想修改日志逻辑，每个方法都要修改，这样太麻烦了。为了解决这个问题，我们可以用设计模式中的模板模式或者装饰者模式，但他们或多或少有一些缺陷，比如不够灵活，代码隔离性不强等等。这时候不妨通过一种方法，将这些日志记录，安全监测等逻辑，与实际业务代码隔离开来，将刚才那些逻辑写在一个个切面里面，最后在程序运行时

动态织入这些切面，并自动生成融合了原类和增强逻辑的代理类来完成功能。

总的来说，AOP主要解决的是非业务代码的抽取问题，利用AOP可以很方便地在某些方法前后增加一些功能，让开发者更加专注于业务逻辑开发。

Spring 中 Bean 的生命周期（Bean 的加载步骤）？

对于Spring Bean 来说，整个生命周期可以分为四个阶段：实例化、属性注入、初始化和销毁。

实例化：

具体来，Bean在实例化之前其实还有一些预热操作，Spring在启动的时候，需要扫描所有被XML或配置注解的Bean，然后将这些Bean的元信息封装成BeanDefination，通过BeanDefinationReader将它们以key-value的形式注册到BeanDefinationRepository中，BeanDefinationRepository本质上是一个Map。接着，Spring会遍历整个map，可选择地执行BeanFactoryPostProcessor来对这些Beandefination做最后的修改和替换，到此为止，Bean实例化预热阶段结束，在真正的Bean实例化阶段，Spring一般是通过反射的方式，选择合适的构造器来把对象创建出来，接着对其进行属性注入，属性注入完就进行Bean的初始化。

初始化

在初始化阶段，首先需要判断这个Bean是否实现了aware接口，如果存在，就填空相关的资源，Aware处理完之后，接着执行BeanPostProcessor后置处理器，他是Aop实现的关键，具体有befor和after两个方法，接着会执行Bean init 相关的方法，其中官方文档定义的执行顺序为：@Postconstrut，实现了initializingBean接口以及init-method方法。其中@PostConstruct方法经常用到，在Bean实例化之后立马去MySQL拿一些数据啥的。之后接着执行after方法，到此为止，一个Bean就初始化完成，可以拿去用了。

需要销毁的时候，需要先看看有没有定义具体的destory方法，执行就行。

需要补充一点，如果某些Bean被配置成懒加载模式，那么只有我们在需要它的时候，才会对其进行真正的实例化。如果没有选择懒加载模式，在容器启动完成后，其中有一些步骤finishBeanFactoryInitialization，来启动这些Bean的实例化阶段。通过隐士调用所有依赖对象的getBean来实例化这些Bean，完成Bean加载。

实例化和初始化是两个完全不同的过程，实例化只是给Bean分配了内存空间，初始化则是实际的为Bean添加相应的业务代码。

Spring 的常用注解？

@Component

@Controller

@Service

@Mapper

@Configuration

@PostConstruct

@PreDestory

@Autowired

@Resource

@Aspect

Spring 注入的几种方式？

Spring 常见又三种依赖注入的方式：基于字段注入、setter注入和基于构造器注入。

基于字段注入：就是直接在Bean的变量上使用注解进行依赖注入，本质上是通过反射的方式直接注入到字段中。这在平时的代码中很常见，但是Spring官方不推荐这种方式。这种方式的优点是代码看起来简洁，只需要@Autowired注解放到变量上面就行了，不需要特殊的构造或setter方法，但是这样就使得字段注入的类变得越来越难以维护，难以理清依赖项之间的关系。并且，注入字段不能实final的，因为final修饰的关键字需要显示地初始化，而依赖出入的字段是spring容器帮我们初始化，所以编译阶段就会报错。

基于setter的构造：就是在对应变量的set方法上使用注解进行依赖注入，优点是完全符合类单一职责原则，每个setter只针对一个对象，缺点是仍然不能注入final对象，并且由于setter方法的操作，意味着被注入的对象随时可能会变化。

基于构造器注入：基于构造器注入是目前比较推荐的注入方式，它可以注入final对象，也不会像setter注入那样会导致被注入的对象被修改

Java 中 final 关键字有什么用？

在Java中，final关键字用于表示一个不可变的常量或一个不可变的变量。

在Java中，final关键字可以修饰类、方法和变量，作用如下：

1. 修饰类：表示该类不能被继承。final类里的方法都是默认实final的，不能被子类重写。
2. 修饰方法：表示该方法不能被子类重写。
3. 修饰变量：表示该变量只能被赋值一次。final修饰的变量必须在声明时或构造函数中初始化，且不能再被修改。常用于定义常量。

另外，使用final修饰的变量在编译时就已经确定其值，因此，在运行时访问，比非final变量更快。

使用final关键字可以带来一些好处，例如：

安全性：将变量声明为final可以防止它被改变，从而提高安全性。

可读性：将常量声明final可以提高代码的可读性，因为常量值不能被修改。

优化：final变量在编译时被转化为常量，这可以提高程序的性能。

Spring 哪里用到了反射？

在Spring IOC容器模块，创建Bean的实例和依赖注入都使用到了反射。

构造

具体的，创建Bean实例时会通过类加载器，根据类路径所在的路径获取到类型为Class的实例，然后通过newInstance创建对象，而newInstance的执行过程，就是先通过反射获取到类的构造方法信息，然后执行默认的构造方法生成实例。

依赖注入

setter方法和构造方法注入：

利用构造器注入和setter注入时也类似，先通过反射获取到当前类的所有方法信息或构造方法信息，然后通过invoke调用setter或通过反射获取到构造器进行带参数的实例创建。

对于设置了@Autorwired的字段注入：

获取到类的所有字段信息之后，判断字段是否有@Autowired注解，然后直接对字段赋值所需要的Bean

在Spring AOP模块，Spring 可以在运行期，动态利用反射，为被代理类生成代理对象，并将其包装成一个切面，从而实现被代理对象的增强操作。

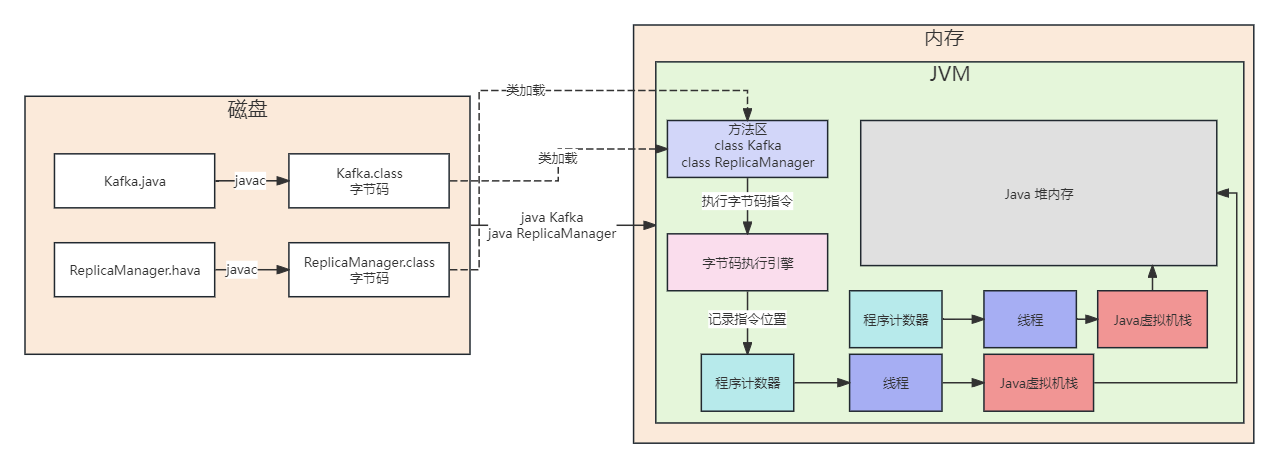
Spring 事务管理怎么实现的？

Spring 整体上提供了声明式事务和编程式事务两种事务管理方式。

编程式事务就是使用硬编码的方式，使用Spring中提供的事务相关的类来控制事务。而编程式事务通常有两种用法：一种是使用PlatformTransactionManager，一种是使用TransactionTemplate来控制事务。PlatformTransactionManager是最原始的方式，代码有点大，它是一个Sring事务管理的核心接口。往下有很多具体的数据源，如JDBC，Hibernate等等都有相对应的实现类。其中我们最常用的就是JdbcTemplate。在使用时，我们一般利用配置的jdbc数据源初始化一个PlatformTransactionManager，然后定义事务的属性如事务隔离级别，事务传播方式等等到TransactionDefination中，最后调用getTransaction开启一个事务，返回一个transactionstatus，这个对象用于确定当前事务的对台，比如最终提交commit还是回滚rollback等等。getTransaction内部简单来说就是创建一个全局共享的ThreadLocal对象用来存储当前线程的datasource到connection的映射，然后就建立一个DB连接。设置是否自动提交事务，最后将其放到ThreadLocal对象中。第二种方式就是使用TransactionTemplate，它在PlatformTransactionTemplateManager的基础上，使用模板方法，进行了一些封装，主要省区了提交或回滚事务的代码，一般是使用它的execute方法，并传入一个TransactionCallBack对象，通常用匿名内部类来实现，在里面进行实务操作，多态。

与编程式事务不同，Spring 的声明式事务是建立在AOP机制上的，其本质是对目标方法的前后进行拦截。在方法开始之前创建或加入一个事务，等到方法执行完时，根据情况选择提交事务或回滚事务。

JVM的垃圾回收机制是用来干嘛的？为什么要垃圾回收？



代码在运行的时候，起码有一个main线程会去执行所有的代码，当然也可能是你启动的别的线程。然后线程执行时必须通过自己的程序计数器来记录执行到哪一个代码指令了，这样有助于上下文切换时知道CPU知道当前线程该执行哪一个代码。另外，线程在执行方法时，为每个方法都得创建一个栈帧放入自己的Java虚拟机栈中，里面有方法的局部变量、上下文信息等。最后就是代码在运行过程中创建的各种对象，都是放在Java 堆内存当中的。

JVM的垃圾回收机制？

JVM垃圾回收机制是一个后台自动运行的线程。

这个线程会在后台不断检查，只要启动一个JVM线程，就会自带这么一个垃圾回收线程。这个线程会在后台不断检查JVM堆中的各个实例对象。

为什么要有垃圾回收？

在传统的C/C++语言中，开发人员需要手动分配和释放内存，这样就导致认为的忘记释放内存的情况，造成内存泄漏。Java的垃圾回收机制可以帮忙解决这个问题，它可以自动检测并回收不再使用的对象，也就是垃圾，可以减少内存泄漏的风险。

什么情况下JVM内存中的一个对象会被回收？

我们平时创建的对象都是优先分配在新生代里面。

如果新生代的对象越来越多，都快要满了，就会触发垃圾回收，把新生代没有人引用的对象回收掉，释放内存空间。这就是新生代的垃圾回收出发时机。

哪些对象是不能回收的，哪些对象是可以回收的？

可达性分析算法：这个算法会分析对象都有哪些变量在引用，然后一层一层上去判断，看看是否有一个GC Roots，可以被所谓GC ROOTS的有：局部变量、静态变量

引用计数器：通过记录对象被多少个变量引用，新增一个对象引用就加1，减少一个对象引用就减1，但是这样会两个对象的成员变量导致循环引用，而无法被回收，造成内存泄漏。

Java 中不同对象的引用类型有哪些?

强引用、弱引用和虚引用

|  |  |
| --- | --- |
| 引用类型 | 垃圾回收行为 |
| 强引用 | 不会被垃圾回收 |
| 弱引用 | 会被回收 |
| 虚引用 | 会被回收 |

Java 垃圾回收中是如何判断一个对象死亡的？

常用的算法有两个：引用计数法和可达性分析法

引用计数法的思路很简单，当对象被对象引用时，计数器加1，当对象失去引用时，计数器减1。当计数器为0时，说明对象不再被使用，也就是垃圾，可以被回收。

可达性分析算法的思路就是分析有哪些变量引用了对象，然后一层一层向上分析，知道到达GC ROOTs，说明对象有GC ROOTS引用，不是垃圾，如果没有GC ROOTS引用，就是垃圾对象，可以被回收。

刚才说到了引用计数法，引用计数法存在什么问题？

如果对象产生循环依赖，那么就无法判断两个对象是否为垃圾对象，造成内存泄漏。

刚才说到了可达性分析，知道哪些可以作为 GC ROOT 吗？

JVM中每个线程都有一个虚拟机栈，虚拟机栈中有栈帧，栈帧里面存储着有指向堆的引用，那么位于虚拟机栈顶的栈帧就可以是作为活跃的栈帧，因为此刻他正在被线程调用。所以，当前活跃的栈帧指向堆里的对象引用就可以是GC ROOTs，除此之外，类的静态变量，Java 本地方法索引用的对象，都可以作为GC ROOTS

方法里的局部变量、类的静态变量，本地方法里引用的对象。

JVM中有哪些垃圾回收算法，每个算法各自的优劣？

JVM 分代模型

垃圾回收算法

这两个并不是绑定的，不同的分代可以使用不同的垃圾回收算法， 一个垃圾回收算法可以应用于不同的分代。

复制算法：

1. 把内存区域分为两块，其中一块用于分配新的对象，另一块空着。
2. 当用于分配新对象的那块内存区域满了之后，触发垃圾回收。
3. 首先利用可达性分析算法，将所有垃圾对象和存活对象标记出来。（标记-清理）
4. 将存活的对象赋值到另外一块空的内存，并紧密排列。
5. 一次性删除用于分配新对象的内存块。

复制算法的好处：减少内存碎片

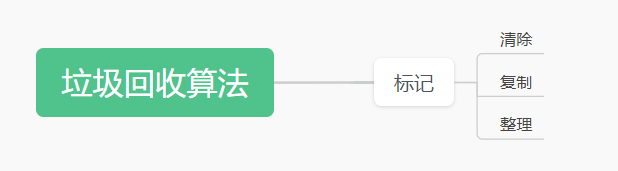
复制算法的缺点：内存利用率低

总结: 所谓的“复制算法”，就是把内存划分为两块区域，然后只是用其中一块内存，待那块内存区域快满的时候，就把里面存活的对象，一次性转移到另一块内存区域，保证没有内存碎片。接着一次性回收原来那块内存区域的垃圾对象，再次空出来一块内存区域。两块内存区域就这么重复循环的使用。

标记-清理算法

标记-移动-清理算法

垃圾回收算法介绍一下



第一步

标记：可达性分析

|  |  |  |
| --- | --- | --- |
| 第二步 | 内存碎片 | 内存利用率 |
| 清除 | 多 | 多 |
| 复制 | 少 | 少 |
| 整理 | 少 | 多 |

什么是内存碎片？

内存碎片就是指内存中对象排列不紧密，对象和对象之间有很多小空隙，虽然总的剩余内存看起来很大，但是无法找到一块连续的内存分配给大的对象。因为这些内存是碎片式分散的，所以导致没有一块完整的足够的内存空间来分配新的对象。

垃圾回收会发生在哪几个区域？

垃圾回收主要发生在堆上，尤其是在新生代中，一次垃圾回收通常可以回收70%到90%的内存空间。但其实也会对方法区进行垃圾回收，对方法区主要回收的是废弃的常量和不再使用的类。

JVM 如何判断方法区中一个类不再被使用？

1. 该类的所有实例都被回收
2. 加载该类的ClassLoader已经被回收。
3. 该类的Class对象没有被引用

什么是永久代?

方法区，存放类的元数据信息、常量等

为什么要区分年轻代和老年代？

1. 大部分对象的声明周期比较短，只有少部分对象可能会存活很长时间，所有区分出了年轻代和老年代。
2. 垃圾回收会导致Stop The World，也就是应用会暂时停止访问，为了使Stop The World持续时间尽可能短，以提高并发式GC所能应付的内存分配速率，有些垃圾回收器就将对象分为两类，存活时间短的对象所处的区域叫年轻代，存活时间长的的对象所处区域叫老年代。

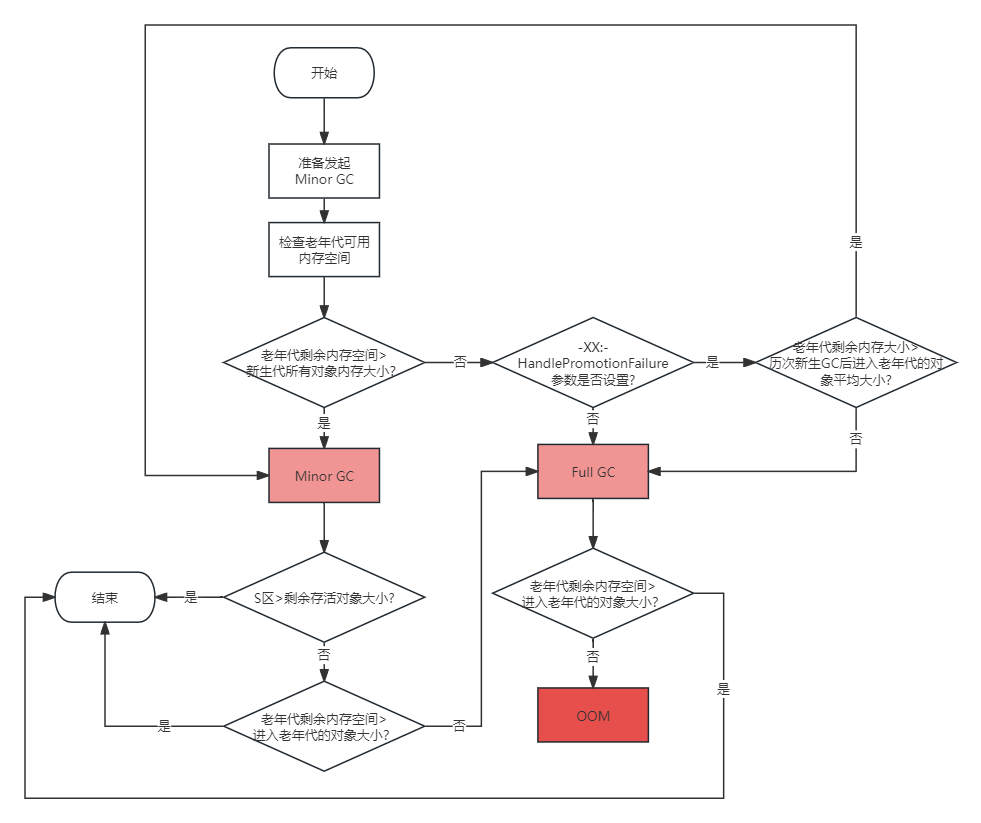
新生代里的对象一般在什么场景下会进入老年代？

1. 躲过15次GC后进入老年代 -XX:MaxTenuringThreshold
2. 动态对象年龄判断:

假设一次年轻代垃圾回收后，存活的对象为: a1,a2,…,an，且Sum(Size(a1),Size(a2),…,Size(an)) >= S区 \* 50%，那么此时，年龄大于MaxAge(a1,a2,…an)的对象，会被放入老年代。

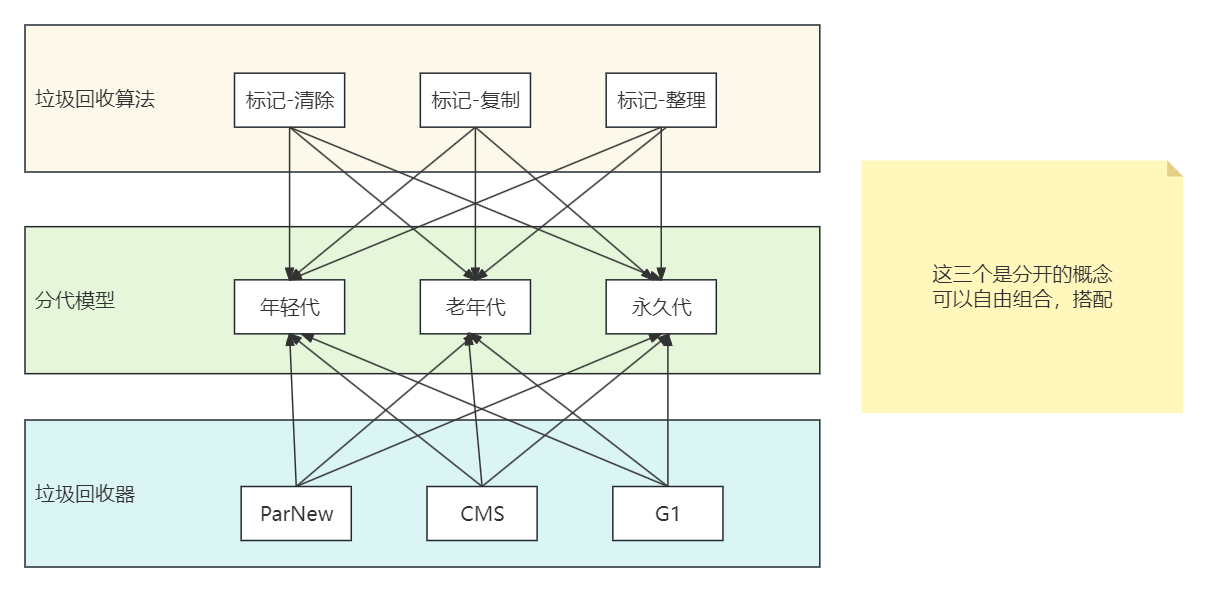
1. 大对象直接进入老年代。-XX:PretenureSizeThreshold，可以把它的值设置为字节数。
2. 年轻代GC后S区还是放不下

什么时候触发新生代GC，什么时候触发老年代GC？



1. 新生代E区满了之后，会触发Minor GC。
2. 开始Minor GC 之前，首先会判断老年代剩余内存大小是否大于新生代所有对象内存总和大小。
3. 如果大于，说明即使Minor GC之后所有对象全部存活，老年代也能装下，所以垃圾回收线程会放心进行Minor GC。
4. 如果小于，说明有可能Minor GC之后存活的对象S区放不下，进入老年代，老年代也放不下。
5. 接着会进一步判断，-XX:HandlePromotionFailure参数是否设置。
6. 如果没有设置，会直接进行Full GC。
7. 如果设置了，会判断老年代剩余内存空间大小是否大于新生代历次进入老年代的对象的平均大小。
8. 如果小于，说明有很大的概率，这次Minor GC之后存活的对象，老年代放不下，所以会直接Full GC。
9. 如果大于，说明有很大的概率，这次Minor GC之后存活的对象，老年代剩余内存可以放下，于是冒险进行一次Minor GC。
10. Minor GC 之后，会判断存活对象大小大于S区大小，如果小于，则放入S区，结束；如果大于，说明S区放不下，需要将存活对象移入老年代。
11. 判断老年代剩余内存是否能够容纳这些存活对象，如果可以，则放入老年代，结束；如果不可以，则触发一次Full GC。
12. Full GC 之后，如果老年代剩余内存还是放不下存活对象，就会导致OOM。

垃圾回收算法、垃圾回收器、分代模型之间的关系？



哪些对像会进入老年代？

1. 存活时间较长，躲过15多次Minor GC就会被移入老年代
2. 动态年龄判断，一批、50%
3. 大对象
4. Minor GC 之后S区放不下

什么时候会进行年轻代GC？

新生代E区满了的时候

什么时候会进行老年代GC？

1. Minor GC 前，老年代剩余内存大小，小于新生代历次进入老年代内存对象的平均大小
2. Minor GC 后，存活对象S区放不下， 进入老年代，老年代内存也放不下。

能说一下JVM运行时的内存区域划分吗？

JVM 运行时内存区域分为了五大块：方法区、程序计数器、Java 虚拟机栈、堆，方法区。

程序计数器：用于记录各个线程执行的字节码指令位置，通常在程序上下文切换时保存当前线程的执行信息。

虚拟机栈：保存方法的局部变量，操作数，参与方法的调用和返回，每个线程在创建时，都会创建一个虚拟机栈，每次方法调用都会创建一个栈帧，并压入虚拟机栈，栈帧中保存的就是方法的信息。如操作数栈，局部变量表等等。

本地方法栈就是用于管理native方法的调用，一般是有C语言实现的。

堆是线程共享的区域，类都实例和数组的分配几乎都来自于堆。堆被划分为新生代和老年代，新生代又被进一步划分为Eden1区和Survivor区，这些都主要跟垃圾回收机制有关。

首先方法区只是JVM的规范，具体的实现可能各个厂商不一样，在HotSpot虚拟机中，JDK 8 以前是用永久代实现的“方法区”，在JDK 8 以后，使用元空间，代理了永久代，实现方法区。方法区主要是用来存放虚拟机加载的类相关信息，比如类信息，常量池等等。其中类信息包括类中的字段、方法、父类等描述类的信息。产量池又包括静态常量池和动态常量池，静态常量池用来保存字面量，以及符号引用等相关信息，动态常量池用来存储类加载时生成的直接引用等信息。

讲一下 JVM 进程与线程模型呢？

在操作系统层面，一个JVM实例被视为一个JVM进程，这个进程用于自己独立的内存空间和执行环境。在单个JVM进程内，可以同时运行多个Java应用程序，他们之间相互独立。

JVM的线程模型在Java规范中并没有要求用哪种方法来实现：

1. JDK1.2之前，使用一种叫绿色线程的用户线程实现，绿色线程指的是由JVM调度，而不是本地操作系统调度的线程，它可以在本来不支持多线程的操作系统上实现多线程。
2. 在JDK1.2之后，采用的是内核线程来实现Java线程，内核线程是操作系统内核直接支持的线程，由内核的线程调度器进行控制和分配，程序一般直接使用内核线程，而是使用它的高级接口，轻量级线程。每个轻量级线程都有一个内核线程与之对应，所以也叫做1:1的线程模型。

堆和栈有啥区别？

主要有存储内容、内存分配和生命周期三个方面不同

1. 存储内容：方法调用时会创建对应的栈帧，并压入虚拟机栈中，栈帧中存储了方法的局部变量表、操作树以及上下文信息。堆里面我们在程序中创建的各种对象和数组。
2. 内存分配：栈通过方法调用，将栈帧压入虚拟机栈中，方法调用结束出栈；堆通过JVM垃圾回收机制回收垃圾对象。
3. 生命周期：对象在Java堆上创建后，如果一直有变量引用，就会一直存在，而虚拟机栈里面的栈帧取决于方法调用何时结束。

什么时候会出现堆栈溢出呢？

栈溢出: Stack Overflow ，由于递归调用层数过多，虚拟机栈内存不够存放栈帧，导致栈溢出。

堆溢出 OOM，新创建的对象过多，导致堆存放不下。

如何排查？

栈溢出: 可以查看控制台打印的错误信息，可能是哪里出现了死循环。

堆溢出：调整JVM的内存参数。

对象一定是在堆在分配的吗？

不一定，JVM会通过逃逸技术分析，对于逃不出方法的对象（没有返回，没有作为参数，只在方法体里面），会直接在栈上分配内存空间，随着方法调用结束出栈，这样可以快速地在栈上创建和销毁对象。

常量池了解吗，有啥用？

Java 8 以前，常量池存在于JVM的永久代中，Java 8以后存在元数据空间中。主要存储：

1. 编译期：各种字面量，这些字面量可以在程序运行时直接使用，不需要再创建，提高运行效率。
2. 运行期：类加载时生成的符号引用等信息。符号引用包括类的全限限定名，方法名等等。主要用于在类加载，动态绑定阶段，来定位具体的类，方法等等。

能说一说你为啥使用 Redis 吗？你觉得 Redis 最核心的功能是什么？

Redis 是一款灵活的高性能的key-value数据库，我们通常可以将其作为缓存、分布式锁、轻量级消息队列，计数器，排行榜，统计榜，统计等。

项目统计网站中UV的实现方案

用Redis HpyerLogLog实现统计网站UV，这里UV是指的什么？

在网站分析中，UV指的是Unique Visitor 的缩写，表示独立访客数。具体来说，UV是指在一定时间范围内（如一天、一周）

MySQL执行流程是怎么样的？

第一步：连接器，1.通过TCP三次握手建立连接；2.通过用户名密码校验；3.校验通过后查询用户对应的权限，整个连接过程都是基于此次查询到的权限。

第二步：查询缓存，很鸡肋。

第三步：解析SQL，在正是执行mysql之前，会通过词法解析器和语法解析器解析客户端请求的sql语句在语法和词法上是否正确。

第四步：执行SQL，经过解析器之后，接着就要进行执行SQL查询语句的流程了，每条select查询语句可以 分为以下三个阶段：prepare预处理、optimize优化阶段、execut执行阶段。经过优化器之后，就是执行器了，执行器会与存储引擎交互，调用存储引擎提供的接口，交互是以记录为单位的。

2 全表扫描，wihle循环会直到存储引擎将表中所有记录扫描一遍后结束。

执行器和存储引擎交互过程是怎样的？

1 以主键索引查询为例：select \* from product where id = 1; 执行该语句的查询

将id=1条件给存储引擎，存储引擎会定位第一条符合id=1的记录。如果记录存在，则会给执行器返回这条记录，如果不存在，则会报找不到记录的错误。执行器读取到记录后，接着判断记录是否符合条件，如果不符合，就跳过该条记录，执行器在一个while循环里面执行，然会后接着调用存储引擎查询下一条记录，由于id是主键索引，使用的是const查询，

如何选择索引？

假设有一张product表，该表有id和name列，其中id是主键索引，name是普通索引，查询以下sql语句:

select id from product where id > 1 and name like 'i%';

答案：会使用name普通索引

原因：mysql默认使用innodb存储引擎，由于InnoDB是聚簇索引，主键索引上面存储索引值和数据，非主键索引只存储索引值和对应的主键值，不存储数据，索引非主键索引树要小一点，但是如果利用非主键索引，如果查询的列不再非主键索引的叶子节点里，可能会导致二次查询，但是select id，查询的是id值，而id值值在非主键所引树的叶子节点上的，通过索引覆盖，最红选择name普通索引。



如何直到某条查询语句使用了哪个索引？

explain + 查询语句

explain输出的关键字段：

id：查询的标识符

select\_type：查询的类型

table：查询涉及到的表

possible\_keys：mysql可能使用到的索引

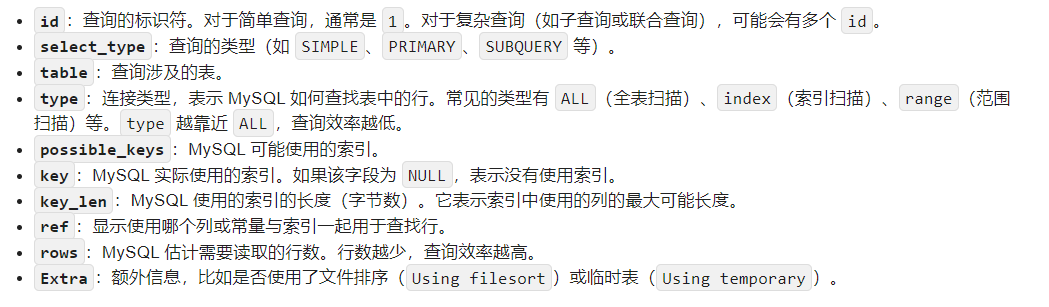
key：mysql实际使用到的索引

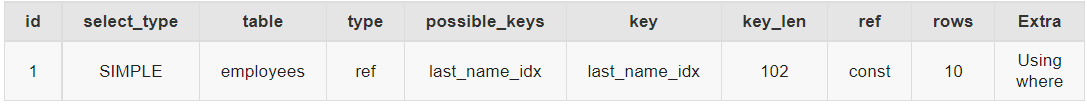
ken\_len：mysql使用的索引的长度

ref：显示使用那个列或常量与索引一起用于查找行

rows：mysql估计要读取的行数

extra：额外信息，比如是否使用了文件排序





预处理阶段干了什么事？

1 检查字段和表是否存在（解析阶段只会检查词法和语法）

2 将select \* 拓展成查询所有列

优化器干了什么事？

经过预处理阶段，还需要为sql查询语句先制定一个查询计划，这个工作就是优化器来完成。

优化器主要负责将SQL语句查询方案确定下来，比如在表里面有多个索引的时候，优化器会基于查询成本的考虑，来决定使用哪个索引。

1. 有多个索引的时候决定使用哪个索引
2. 调整and条件的位置满足复合索引最左匹配原则

查询缓存的原理是什么？

对于一条select语句，mysql服务层会先去查查询缓存，查询缓存是以key-value形式存储的，key是select语句，value是对于结果，如果有，直接返回，如果没有，会通过存储引擎层查询， 然后返回并将结果写入查询缓存。

为什么MySQL 8之后删除掉了查询缓存模块？

因为对于频繁更新的表，缓存经常会失效，存储和清理缓存反而会降低查询效率。对于更新比较频繁的表，缓存命中率是很低的，只要有一个表有操作，那么这个表的所有缓存查询都会被清除，对于myslq8之前的版本，如果想关闭查询缓存，可以通过query\_cache\_type设置成DEMAND

聊一聊索引下推？

Index Condition Pushdown

ICP

是MySQL5.6提出的查询优化策略

索引下推优化的目标：1 针对联合索引 2 针对联合查询的范围查询

以t\_user表为例，有age和reward连个字段，建立(age, reward)字段的联合索引，如果查询以下语句：

select \* from t\_user where age > 20 and reward = 100000;

只有age会用到联合索引，reward是用不到的，因为对于age>20，reward是无序的，所以用不到联合索引，而对于确定的age=20，reward才能用到联合索引，联合索引会优先保证最左边的列的有序性。

|  |  |  |
| --- | --- | --- |
| id | age | reward |
| 1 | 21 | 100000 |
| 2 | 19 | 5000 |
| 3 | 23 | 100000 |
| 4 | 19 | 100000 |
| 5 | 18 | 70000 |
| 6 | 26 | 70000 |

如果没有索引下推，查询到age>20的记录后，并不会看reward时为100000，而是获取主键值，然后回表，将完整的记录交给server层，server层判断reward是否为100000，如果成立，就保存代结果返回集合，如果不成立则跳过。

有了索引下推，不会执行回表操作，而是查出来之后，直接在存储引擎层进行判断， reward是否为100000。

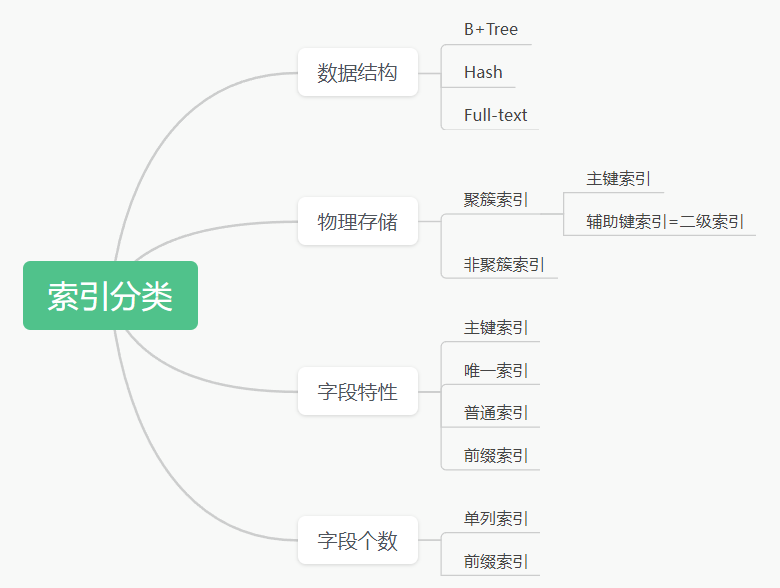
索引下推减少了回表操作，提高了查询效率。

什么是索引？

索引是种数据结构，类似目录，用于提高数据的查询效率。

你知道的索引有哪些吗？

主键索引、普通索引、复合索引、唯一索引



创建表时，InnoDB如何指定聚簇索引的索引键？

1. 有主键时，选择主键作
2. 没有主键，选择第一个满足不为null且唯一的列
3. 以上两个条件都没有时，自动生成一个隐式的自增id作为索引键

死锁的四个必要条件？

1. 互斥条件
2. 占用并等待条件
3. 不剥夺条件
4. 循环等待条件

聊一聊B+树？

B+树是一种多叉查找树，叶子节点才存放数据，非叶子节点只存放索引。而且每个节点里的数据是按主键顺序存放的。

B+树是一个多叉的查找树，一颗m阶的B+树，每个节点的最多有m-1个子节点，B+树的非叶子节点只存放索引值和指针，叶子节点存放数据，且叶子节点之间用指针连接形成一条链表，这个链表是有序的。

索引是存储在哪里的？

数据库的索引和数据都是存储在磁盘里的，读取一个节点就会进行一次IO操作

通过二级索引查询数据的过程？

需要进行回表操作

因为辅助见索引的叶子节点不存放数据，只存放索引值和对应的主键值

什么是索引覆盖？

利用辅助键索引查询数据时，对于被查询的字段，如果能够在辅助键索引对应的叶子节点种找到，就无回表，直接返回可以，降低回表次数，提高查询效率。

为什么 MySQL InnoDB 选择 B+tree 作为索引的数据结构？

本质原因：B+树的非叶子节点只存储索引值和指针，不存储数据

假设一个块为16KB，table表有

id int 4B

age int 4B

name varchar(255)

…

一条数据占用1KB

指针大小1B

如果采用B树，叶子节点存储数据，则最多存储16个数据，也就是B树的阶树最多为16阶，如果数据量很大很大，超过千万级别，则树的高度会变高，导致磁盘IO的次数变多。

事务隔离级别是怎么实现的？

什么是事务？

一组操作，要么都成功，要么都失败。

事务的四大特性？

原子性

一致性

隔离性，并发事务

持久性

并发事务会出现哪些问题？

脏读，读取了一个未提交的数据

|  |  |  |
| --- | --- | --- |
| 时刻 | 事务A | 事务B |
| t1 | 开始事务 |  |
| t2 |  | 开始事务 |
| t3 | 修改a，a = 200 |  |
| t4 |  | 读取a，a=200（读取到了脏数据） |
| t5 | 回滚，撤销修改 |  |
| t6 |  | 操作 |
| t7 |  | 提交 |

通过都提交隔离级可以解决

不可重复读取，两次读取同一数据，结构不一样。

|  |  |  |
| --- | --- | --- |
| 时刻 | 事务A | 事务B |
| t1 | 开始事务 |  |
| t2 |  | 开始事务 |
| t3 | 读取a，a=100 |  |
| t4 |  | 读取a，a=100 |
| t5 | 修改a，a = 200 |  |
| t6 | 提交 |  |
| t7 |  | 读取a，a=200（与第一次读取的数据不一样） |

不可重复读可以通过加行锁解决

幻读，同一事物，前后两次读取的表数量不一致

可以通过加表锁解决

事务的隔离级别有哪些?

读未提交

读已提交

可重复读，mysql innodb 默认的隔离级别

串行化

MVCC机制？

read view 的四个字段：

1. m\_ids，表示在创建试图时，数据库中活跃事务id列表
2. min\_trx\_id：m\_ids中最小的id
3. max\_trx\_id: m\_ids最大值加1
4. creator\_trx\_id，创建该视图的事务的id

undo log 日志

每一行有两个隐藏列:

trx\_id，当一个事务对某行进行改动时，会把该事务的id记录在trx\_id中

roll\_pointer，指针，指向上一个版本的行

MVCC 多版本并发控制

MVCC 如何解决不可重复读问题？

|  |  |  |
| --- | --- | --- |
| 时刻 | 事务A（id=1） | 事务B（id=2） |
| t1 | 开始事务 |  |
| t2 |  | 开始事务 |
| t3 | 读取a，a=100 |  |
| t4 |  | 读取a，a=100 |
| t5 | 修改a，a = 200 |  |
| t6 | 提交 |  |
| t7 |  | 读取a，a=200（与第一次读取的数据不一样） |

|  |  |
| --- | --- |
| m\_ids | 1,2 |
| min\_trx\_id | 1 |
| max\_trx\_id | 3 |
| creator\_trx\_id | 2 |

|  |  |  |
| --- | --- | --- |
| a | trx\_id | roll\_pointer |
| 100 | 0 | null |

|  |  |  |
| --- | --- | --- |
| a | trx\_id | roll\_pointer |
| 200 | 1 | null |

事务B读取a时，找到undolog链条

如果 trx\_id 在m\_ids 中，说明是正在活跃的事务，修改的数据，还未提交，事务还没有结束所以无法查询，会从roll\_pointer链条，向上查询

如果trx\_id 在m\_ids 左边，（即trx\_id < min\_trx\_id），则说明事务已经提交，不再活跃列表中，对当前事务可见，所以，可以读取，于是读取a=100，这样就不会存在读未提交错误。

如果要想解决不可重复读的问题，需要在第一次快照读就确定好read view，以后每次快照读都复用第一次的快照读。

什么是控制并发流程?

在没有并发控制的时候，多个线程并发执行实际上受的是线程调度器的控制，并不受程序员控制。在一些情况下，我们会要求某些任务先执行，一些任务后执行，这个时候就必须控制它，依靠线程调度器是不靠谱的。

能说一说你为啥使用 Redis 吗？你觉得 Redis 最核心的功能是什么？

Redis 是一款高性能的key-value数据库，我们通常可以将其用作缓存、分布式锁、轻量级消息队列，计数器、排行榜，统计等。

1. 项目中统计网站UV的实现方案

首先想到的是，用一个set集合来存储当天访问的用户IP地址

但是这样会有两个问题:

1. 如果用户量非常大，set集合存储不下
2. 多个set集合之间聚合查询非常复杂

Redis的HyperLogLog就是专门解决这个问题的，HyperLogLog是一种基数算法，用于估算集合中不同元素数量（也叫去重后的数量），可以利用极小的内存完成独立总数的统计。数据集可以实IP、Email、ID等，它同时还支持多个HyperLogLog的并集。

另一个例子是项目中的“限制用户刷新次数”功能，虽然可以将用户刷新次数保存到数据库，但是每次去连接数据库、并读取比较耗时，Redis提供的，具有天然自增功能的String数据结构可以很好地解决这个问题。并且这个自增操作incr具有原子性，可以应对高并发场景。

2、快 是Redis最大的闪光点

一提到Redis，大家可能脱口而出：“快”。这当然也是大部分人使用Redis的原因。为了实现“快”，Redis采用了单线程，基于多路复用的高性能IO模型等。

具体来说，Redis将网络IO和键值对读写都交给一个线程来完成。

单线程的Redis为什么这么快？

1、基于内存

2、单线程

3、I/O多路复用

什么是UV？

Unique Visitor，独立访客

DAU Dailly Active User，日活跃用户

HyperLogLog是一种基数算法，这里的基数表示什么意思，为什么这样说？

基数的定义：集合中不同元素的个数

什么是I/O多路复用？

I/O是指Input 和 Output

多个Input和Output操作复用一个进程

应用程序把一堆需要处理的文件描述符丢给内核，其中任何一个文件描述符背后的文件具备读写条件时，就把这些文件描述符筛选出来给应用程序处理。

你知道Redis有哪些常见数据结构以及应用场景吗？

常见的数据结构有：字符串String、列表List、集合Set、有序集合Sorted Set，除了这些，还有Bitmaps、HyperLogLog。

1 String 字符串类型

String类型是Redis最基础的数据结构，其他几种数据结构都是在字符串类型的基础上创建的。并且所有数据结构。