**MySQL的事务四大特性是怎么实现的**

**ACID原一隔持**

ACID四大特性：

原子性（**A**tomicity）

含义：事务中所有操作要么全部完成，要么全部失败回滚。

实现：MySQL通过日志和回滚段来实现。

一致性（**C**onsistency）

含义：事务在执行前后数据库从一个一致性状态转移到另一个一致性状态。

实现：MySQL通过检查约束、触发器和存储过程等手段来保证一致性。

隔离性（**I**solation）

含义：多个事务并发执行感觉不到其他的存在。

实现：MySQL通过锁机制和多版本并发控制（MVCC）实现事务隔离级别：

读未提交、读已提交、可重复读、串行化。

持久性（**D**urability）

含义：事务一旦提交，改变永久保存。

实现：MySQL通过将事务日志写入磁盘来实现。

**详细讲一讲MVCC**

**版读写隔收**

多版本并发控制（**M**ulti-**V**ersion **C**oncurrency **C**ontrol）

允许多个事务读写并发执行，提高了数据库系统的性能。

1. **版本号**： 每个数据行一个版本号，通常是一个递增的整数。
2. **读操作**：根据事务启动时间和版本号判断。只有那些在事务启动前提交的数据版本对当前事务是可见的。
3. **写操作**：不直接修改，而是创建新数据版本。原始数据版本依然对其他事务可见，直到新的版本被成功提交。
4. **事务隔离级别**：例如，可重复读级别在整个事务期间只看到一个固定快照，而读已提交级别则在每个读操作时都获取当前最新的数据版本。
5. **回收过期数据**：定期清理已经过期的数据版本。通过后台任务如MySQL中的purge线程完成。

**JWT是什么**

**头载签**

JWT（**J**SON **W**eb **T**oken）

是一种开放标准（RFC 7519），用于在身份验证和信息交换。

主要由头部、载荷、签名 通过.连接成字符串。

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c

1. **头部（Header）**：令牌的元信息，例如使用的算法（例如HMAC SHA256或RSA）、令牌的类型（JWT）等。
2. **载荷（Payload）**：要传递的信息，是关于声明的JSON对象。包含一些预定义的声明（例如iss-签发者、sub-主题、exp-过期时间等）。
3. **签名（Signature）**：对头部和载荷的签名，可以验证消息来源和完整性。

通常在身份验证过程中，用户通过提供用户名和密码获取JWT令牌，然后将该令牌包含在后续的请求中，以验证用户的身份和获取相应的资源。

需要注意的是，JWT是基于文本的，因此在传输过程中可能会被轻松解码，但可以通过使用HTTPS等安全传输方式来增加安全性。

此外，令牌一旦签发，就无法撤销，因此令牌的过期时间需要谨慎设置。

**Redis的数据结构有哪些，怎么实现的**

**字哈列集序位H地**

**字符串（String）**： Simple Dynamic Strings实现，允许动态地扩展缩小长度

**哈希表（Hash）**：通过哈希值将数据分布到不同的槽位上，以提高查找效率。

**列表（List）**：双向链表结构实现。在两端执行插入和删除操作的效率高。

**集合（Set）**：哈希表结构实现，支持高效的添加、删除和判断元素是否存在的操作。

**有序集合（Sorted Set）**：跳表和哈希表结合。跳表提供有序性，哈希表提供快速查找。

**位图（Bitmap）**

**HyperLogLog**：用于估计基数（集合中不重复元素的数量）

**地理空间索引（Geospatial Index）**： ZSET的分数表示地理位置的坐标。

**Redis为什么快？**

**内单非数持协**

**内存存储：** Redis主要将数据存储在内存中，内存的读写速度远高于磁盘

**单线程模型：** Redis采用单线程模型，通过事件循环来处理多个客户端请求。避免了多线程之间的锁竞争，减少了上下文切换的开销。

**非阻塞式 I/O：** Redis使用非阻塞式的I/O操作，使得在进行网络通信和文件操作时，能够更好地利用系统资源，提高吞吐量。

**高效的数据结构**

**持久化选项：** Redis支持多种持久化选项，包括快照和追加式文件（AOF）。

**优化的网络协议：** Redis使用自定义的协议进行客户端和服务器之间的通信。这种协议是文本协议，易于理解和调试。同时，它也是二进制安全的，支持存储和传输任意类型的数据。

**详细讲一讲IO多路复用**

**注册监听通知操作**

**Select poll epoll kqueue**

1.定义：

IO多路复用它允许单一的线程同时监听多个IO通道上的事件。通过这种方式，一个线程能够有效地管理多个IO通道，从而减少线程的数量，提高系统的性能和资源利用率。如果一个IO通道上的IO操作是阻塞的，可能会影响整个IO多路复用的性能，因此通常建议使用非阻塞IO。

2.常见的IO多路复用机制：

select：通过select系统调用，允许程序监视一组文件描述符，一旦其中一个文件描述符准备好进行IO操作，就通知程序进行相应的读写操作。

poll：与select类似，但采用链表，避免了select中的文件描述符数量限制。

epoll：是Linux下效率最高的IO多路复用机制，使用红黑树实现高效的事件查找。epoll的性能随着文件描述符的增加而近乎线性增长，适用于高并发的场景。

kqueue：是BSD和macOS中的IO多路复用机制，与epoll类似，通过红黑树实现，具有高性能和可扩展性。

3. 工作流程：

程序通过IO多路复用调用（如select、epoll）将一组文件描述符注册到内核中。

内核监视这些文件描述符上的事件，当有IO事件发生时，通知应用程序。

程序通过遍历通知，找到准备好的文件描述符，进行相应的IO操作。

这样，一个线程就能够同时管理多个IO通道，而不需要为每个IO通道创建一个独立的线程。

**消息队列的选型，Kafka和RabbitMQ的区别，各自的应用场景**

**Kafka更稳定大规模**

**RabbitMQ更灵活易用**

**Kafka:**

设计理念：

Kafka设计为分布式、高可用、持久化的消息系统。它的主要目标是提供高吞吐量、低延迟和容错性。

数据模型：

Kafka采用发布-订阅的模型，消息以主题（Topic）为单位组织。消息存储在分布式日志中，允许多个消费者并行消费。

可靠性：

Kafka强调消息的持久性，一旦消息被写入，就会被持久存储。它支持多副本复制，确保数据可靠性。

性能：

Kafka在高吞吐量和低延迟方面表现出色，特别适合大规模数据流处理。

适用场景：

日志收集与分析、事件溯源、大数据处理、实时数据流处理等。

**RabbitMQ:**

设计理念：

RabbitMQ设计为一个灵活、易用的消息队列系统。它提供多种消息传递模式，包括点对点和发布-订阅。

数据模型：

RabbitMQ支持多种消息传递模式，包括点对点、发布-订阅和路由。消息存储在队列中，每个队列有一个或多个消费者。

可靠性：

RabbitMQ提供可靠性传递，支持持久化消息。它可以通过不同的交换机类型和队列设置来实现不同的消息路由策略。

性能：

RabbitMQ在一般情况下表现良好，但相对于Kafka，它在处理大量数据和高吞吐量方面可能稍显逊色。

适用场景：

任务队列、点对点通信、发布-订阅、应用解耦等。

**如何选择：**

数据处理特点：

如果需要处理大规模的实时数据流、日志等，Kafka可能是更合适的选择。

如果强调消息传递的可靠性和灵活性，以及更广泛的消息模型，RabbitMQ可能更适用。

复杂性和易用性：

RabbitMQ通常被认为更易于入门和使用，适用于相对简单的消息场景。

可靠性需求：

如果对消息的持久性和数据不丢失有较高要求，Kafka可能更符合需求。

**怎么设计一个feed流，更多地推送关注的公众号的帖子，要考虑到用户量和帖子数量可能都很多**

**1. 系统架构设计：**

分布式架构：将系统拆分成多个服务，例如用户服务、推荐服务、帖子服务等，通过消息队列或异步通信进行解耦。

缓存：使用缓存中间件如Redis来存储热门的帖子和用户关注信息，以减轻数据库压力，提高访问速度。

**2. 推荐算法：**

个性化推荐： 利用用户的历史行为、关注信息和兴趣标签。

协同过滤： 根据用户关注的公众号和其他用户的关注行为，推荐相似兴趣的公众号帖子。

实时推荐： 使用实时推荐算法，确保用户在订阅新公众号后能够迅速看到相关推送，以提升用户体验。

**3. 性能优化：**

分页和懒加载： 使用分页和懒加载的方式，不一次性加载所有帖子信息，减轻服务器和客户端的负担，提高页面加载速度。

异步处理： 使用异步任务队列处理后台任务，如推送通知、计算用户兴趣等，以提高系统的吞吐量和响应速度。

负载均衡： 部署负载均衡机制，确保系统在用户量和帖子数量增加时能够平稳运行，提高系统的可伸缩性。

**4. 用户体验：**

定期更新： 定期更新feed流，确保用户能够看到新鲜、有趣的内容，增加用户粘性。

用户反馈： 提供用户反馈机制，例如举报、不感兴趣等，用于优化推送算法和改进用户体验。

推送策略： 设计合理的推送策略，避免频繁推送相同内容，同时确保用户不错过重要信息。

**什么是feed流**

**Feed流（Feed）**

是一种以时间顺序排列的信息流，用于展示用户关注的内容或事件。

**Feed流的主要特点**

是持续不断地向用户展示最新的信息，用户可以通过滚动页面或类似的方式不断获取更新。

**Feed流的目的**

是向用户提供个性化、实时且不断更新的信息，以保持用户的兴趣和参与度。Feed流通常基于用户的关注列表、个人兴趣和社交网络等信息，通过智能推荐算法为用户提供个性化的内容推荐。同时也为平台提供了更多的机会展示广告、推广内容等。社交媒体中的朋友圈、Twitter的Timeline、Facebook的News Feed等都是Feed流的典型例子。

**怎么实现扫码登录？**

**1. 生成二维码：**

可以使用qrcode库来生成二维码

**2. 展示二维码：**

**3. 扫描二维码：**

**4. 扫描结果处理：**

在服务端接收到扫描结果后，进行相应的处理。通常，扫描结果包含一个唯一标识或认证信息。

**5. 验证认证信息：**

服务端根据扫描结果中的信息，进行用户身份验证。可以验证用户是否存在、是否具有相应权限等。

**6. 登录处理：**

如果身份验证通过，标识用户已登录。可以生成一个用户凭证（如Token）返回给客户端，客户端用于后续的请求验证。

**7. 定期检查扫描状态：**

在用户扫描二维码后，可以定期检查扫描状态，确认用户是否已经完成登录。这通常通过前端轮询或WebSocket等机制来实现。

**注意事项：**

使用HTTPS协议确保通信的安全性。

考虑二维码的有效期，以及生成和验证的安全性。

在实际应用中，可能需要通过OAuth等协议实现更复杂的身份验证流程。

**加了过期时间的LRU缓存，缓存里的内容x秒后会过期，如何实现**

实现带有过期时间的LRU（Least Recently Used）缓存通常需要结合两个主要概念：LRU算法和定时过期。以下是一个简单的Python示例，使用collections.OrderedDict来实现LRU缓存，并通过定时任务来处理过期时间：

这个示例中，LRUCacheWithExpiration类继承自OrderedDict，通过使用OrderedDict的有序字典来实现LRU缓存，同时通过定时任务检查过期时间并清理过期的缓存项。注意，这里使用了一个简单的线程来定期清理过期项，你可能需要根据实际需求进行调整，例如使用定时器、协程等机制。

**线程和协程的区别与联系**

**区别：**

执行单元：

线程： 是操作系统调度的最小执行单元，由操作系统进行管理和调度。

协程： 是在用户空间进行调度的执行单元，由程序员显式控制。

调度方式：

线程： 由操作系统进行抢占式调度，可以被中断和切换。

协程： 由程序员控制，在需要时显式地挂起和恢复。

并发性：

线程： 线程可以并行执行，由操作系统管理多个线程的调度。

协程： 协程一般在单个线程内执行，通过非抢占式调度实现协作式并发。

状态保存：

线程： 线程的状态由操作系统负责保存和恢复。

协程： 协程的状态由程序员显式保存和恢复，通常保存在栈上。

**联系：**

并发编程：

无论是线程还是协程，都是为了实现并发编程，让程序能够有效地处理多个任务。

异步编程：

协程常用于异步编程，通过非阻塞的方式处理I/O等操作，提高程序的性能。

资源消耗：

协程通常消耗的资源较少，因为它在用户空间进行调度，减少了操作系统线程切换的开销。

可读性和控制：

协程通常具有更好的可读性，因为它在编写代码时更接近自然的同步写法，而不需要考虑锁和线程间通信的复杂性。

程序员对协程的控制更细粒度，可以手动控制协程的执行顺序。

**总体而言**

线程更适用于 CPU 密集型任务

协程更适用于 I/O 密集型任务，以及需要更好的可读性和控制的情况。

**浏览器键入baidu.com所有用到的协议**

当你在浏览器中键入baidu.com并按下回车时，浏览器会根据输入的网址进行一系列操作，涉及到多个协议。以下是在浏览器键入baidu.com时可能用到的主要协议：

DNS 协议（Domain Name System）：

浏览器首先会将域名 baidu.com 转换成对应的 IP 地址。这个过程使用 DNS 协议进行域名解析。

HTTP 协议（Hypertext Transfer Protocol）：

一般情况下，浏览器会默认使用 HTTP 协议进行网页的请求和响应。浏览器向服务器发送 HTTP 请求，请求获取 baidu.com 的网页内容。

HTTPS 协议（Hypertext Transfer Protocol Secure）：

如果网站启用了 HTTPS，浏览器会使用 HTTPS 协议进行加密的通信。这涉及到 SSL/TLS 协议用于加密数据传输，确保数据的安全性。

TCP 协议（Transmission Control Protocol）：

HTTP 和 HTTPS 协议都是基于 TCP 协议的，它负责在网络上建立可靠的连接，确保数据的可靠传输。

IP 协议（Internet Protocol）：

IP 协议用于在网络上进行数据包的寻址和路由。它是 TCP 和 UDP 的底层协议，负责在网络上进行数据包的传递。

ICMP 协议（Internet Control Message Protocol）：

在进行网络通信时，可能涉及 ICMP 协议，用于处理网络错误、诊断问题等。

以上是在浏览器键入 baidu.com 时可能用到的主要协议。实际情况中，根据网站的具体配置和浏览器的支持情况，可能还涉及到其他协议和技术，如 QUIC 协议、WebSocket 等。

**TCP和UDP区别**

TCP（Transmission Control Protocol）和UDP（User Datagram Protocol）是两种常用的传输层协议，它们在网络通信中有一些重要的区别：

1. 连接性：

TCP： 提供面向连接的通信。在数据传输之前，必须先建立连接，然后进行可靠的数据传输，最后再释放连接。

UDP： 是面向无连接的通信。每个数据包都是独立的，不需要先建立连接，直接发送数据。因此，UDP通信更为轻量，但不保证可靠性。

2. 可靠性：

TCP： 提供可靠的数据传输，通过序列号、确认和重传机制确保数据的完整性和顺序性。如果数据包丢失，TCP会进行重传。

UDP： 不提供可靠性保证，数据包可能会丢失，也不保证数据包的顺序性。对于某些实时性要求高的应用，可以容忍少量的丢包。

3. 数据传输方式：

TCP： 数据以流的方式进行传输，被划分为小的数据段，然后通过序列号进行重新组装。数据传输是可靠而有序的。

UDP： 数据以数据包的形式传输，每个数据包都是独立的。UDP更适用于需要快速传输的场景，如实时音视频。

4. 头部开销：

TCP： TCP头部相对较大，包含序列号、确认号、窗口大小等字段，用于保证可靠性和流控制。

UDP： UDP头部较小，只包含源端口、目标端口、长度和校验和等基本信息，开销较小。

5. 适用场景：

TCP： 适用于对数据完整性和顺序性有要求的应用，如文件传输、网页访问等。

UDP： 适用于实时性要求高、可以容忍少量数据丢失的应用，如语音通话、视频直播等。

6. 连接和断开：

TCP： 建立连接时有三次握手的过程，断开连接有四次挥手的过程。

UDP： 无连接的特性，不需要建立和断开连接的过程。

总体来说，选择TCP还是UDP取决于具体的应用需求。TCP适用于要求可靠性和有序性的场景，而UDP适用于实时性要求高、可以容忍少量数据丢失的场景。在某些应用中，也可以结合使用TCP和UDP，根据不同的数据传输需求选择合适的协议。

**线程和进程区别**

线程（Thread）和进程（Process）是操作系统中进行多任务处理的两个基本概念，它们有一些关键的区别：

1. 定义：

线程： 线程是进程内的一个独立执行单元，共享进程的资源，包括内存空间和文件句柄等。多个线程可以在同一进程内并发执行。

进程： 进程是一个独立的执行环境，包括独立的内存空间、文件句柄等，可以包含多个线程。

2. 资源分配：

线程： 线程是进程内的轻量级执行单元，共享进程的资源。线程之间的切换开销较小。

进程： 进程是独立的执行环境，有独立的内存空间和资源。进程之间的切换开销较大。

3. 通信和同步：

线程： 线程之间共享进程的数据和资源，因此需要进行同步操作以避免数据竞争。

进程： 进程之间的通信较为复杂，通常需要使用进程间通信（IPC）机制，如管道、消息队列、共享内存等。

4. 创建和销毁开销：

线程： 创建和销毁线程的开销相对较小，因为线程共享进程的资源。

进程： 创建和销毁进程的开销相对较大，因为每个进程都有独立的资源空间。

5. 独立性：

线程： 线程是进程的一部分，对于同一进程内的线程，它们共享相同的地址空间和文件描述符等资源。

进程： 进程是相对独立的，它们有独立的地址空间和资源，通常需要通过IPC进行通信。

6. 安全性：

线程： 多个线程共享同一进程的地址空间，因此需要使用同步机制来保证数据的一致性。

进程： 进程拥有独立的地址空间，相对于线程更为安全，不容易出现数据竞争。

7. 适用场景：

线程： 适用于需要轻量级并发的场景，如GUI应用、网络服务器等。

进程： 适用于需要独立执行环境、数据隔离的场景，如操作系统、数据库系统等。

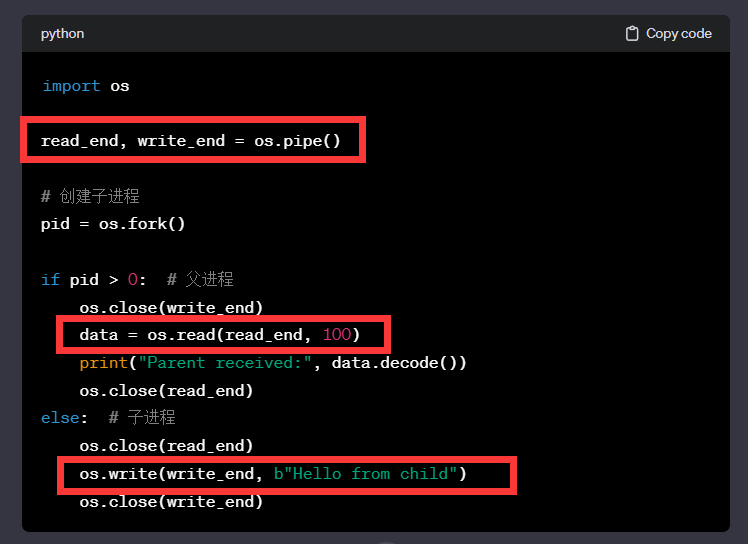
总体来说，线程适用于并发执行的场景，能够更高效地利用系统资源；而进程适用于需要隔离和独立执行环境的场景，但其开销相对较大。在实际应用中，线程和进程可以结合使用，根据具体需求选择合适的并发模型。

**进程通信**

进程通信（Inter-Process Communication，IPC）是指在多进程环境中，不同进程之间进行数据交换和通信的机制。

1. 管道（Pipe）：

管道是一种半双工的通信机制，适用于具有父子关系的进程之间。它由一个读端和一个写端组成，数据从写端写入，从读端读取。管道可以通过系统调用 pipe() 创建。



2. 消息队列（Message Queue）：

消息队列是一种在进程之间传递消息的通信方式，进程可以通过系统调用 msgget()、msgsnd() 和 msgrcv() 来使用消息队列。



3. 共享内存（Shared Memory）：

共享内存是一种允许多个进程共享同一块物理内存的通信方式。进程可以通过映射共享内存到它们的地址空间，并进行读写操作。

4. 信号（Signal）：

信号是一种进程间通信的简单方式，用于通知进程发生了某个事件。可以使用系统调用 signal() 来处理信号。

5. 套接字（Socket）：

套接字是一种用于在网络上进行通信的机制，也可以用于在同一台机器上的不同进程间通信。

**内存管理**

1. 内存分区：

连续内存分配： 将内存分为若干连续的区域，每个进程占用一段连续的内存空间。这种方式包括最先适应算法、最佳适应算法等。

分页和分段： 将内存划分为若干固定大小的页或段，进程的地址空间可以由多个页或段组成。分页和分段可以提高内存的利用率和灵活性。

2. 虚拟内存：

虚拟内存是一种将磁盘空间作为辅助存储器的技术，使得每个进程都能拥有比实际物理内存更大的地址空间。通过分页机制，操作系统可以将进程的部分数据保存在磁盘上，需要时再将其加载到物理内存中。

3. 地址映射和转换：

逻辑地址和物理地址： 进程使用的地址空间是逻辑地址空间，而物理内存的地址是实际的硬件地址。操作系统负责将逻辑地址映射到物理地址，通过地址转换实现。

页表和段表： 使用页表或段表来记录逻辑地址到物理地址的映射关系。页表将逻辑地址划分为页，并映射到物理页框；段表则映射逻辑地址的不同段到物理内存。

4. 内存保护：

通过访问权限位，操作系统可以实现对内存的保护。例如，只读、读写、执行等权限设置，可以避免程序越界访问和恶意代码执行。

5. 内存回收和释放：

垃圾回收： 对于使用动态内存分配的语言，垃圾回收机制负责识别和回收不再使用的内存，防止内存泄漏。

内存释放： 当一个进程终止或释放了一块动态分配的内存时，操作系统需要及时回收这些内存，以供其他进程使用。

6. 内存交换和页面置换：

内存交换： 当内存不足时，操作系统可以将部分进程的数据保存到磁盘上，称为内存交换。

页面置换： 在分页系统中，当需要加载某一页而内存不足时，需要选择一个页面将其替换出去。常见的算法包括最近最少使用（LRU）、先进先出（FIFO）等。

7. 内存碎片：

内部碎片： 由于分配的内存块可能比需要的稍大，导致一部分内存浪费。

外部碎片： 分配的内存是离散的，可能导致无法找到足够大的连续内存块。

8. 页面调度：

页面置换算法： 在分页系统中，选择合适的页面置换算法是关键，不同算法有不同的性能特点。

**MySQL的索引相关内容？有哪些索引？**

1. B树索引（B-tree Index）：

B树（Balanced Tree）是一种平衡的多路搜索树，用于加速数据的查找。

MySQL的InnoDB存储引擎默认使用B树索引。

特点：

平衡树结构，保证查询效率稳定。

2. 唯一索引（Unique Index）：【唯一 不重复NULL】

唯一索引确保列的值在表中唯一，用于保证数据的一致性和防止重复数据的插入。

特点：

列值唯一。

可以包含NULL值，但NULL值不重复。

3. 主键索引（Primary Key Index）：【唯一 不允许NULL】

主键索引是一种特殊的唯一索引，用于唯一标识表中的每一行记录。

4. 全文索引（Full-Text Index）：

全文索引用于对文本数据进行全文搜索，适用于对大段文本进行搜索的场景。

特点：

用于对文本数据进行全文搜索。

支持关键词搜索、模糊匹配等。

5. 空间索引（Spatial Index）：

空间索引用于支持对空间数据类型（如点、线、面）的查询。

特点：

用于处理地理空间数据。

支持空间关系查询，如距离、相交等。

6. 组合索引（Composite Index）：

组合索引是基于多个列的索引，用于支持对这些列的组合查询。

特点：

可以包含多个列，提供多列的查询性能。

查询时遵循最左前缀原则。

7. 前缀索引（Prefix Index）：

前缀索引是对列值的前缀进行索引，可用于节省索引空间。

特点：

只索引列值的一部分前缀，减小索引大小，提高性能。

8. 哈希索引（Hash Index）：

哈希索引使用哈希函数将列值映射为哈希码，适用于等值查询。

特点：

适用于等值查询，不支持范围查询和排序。

9. 位图索引（Bitmap Index）：

特点：

适用于离散值列、低基数列，，如性别、国家等，空间效率高，但不适用于高基数列。

**聚簇索引和非聚簇索引？**

1. 聚簇索引（Clustered Index）：

定义： 聚簇索引决定了表中数据的物理排序顺序，表中的行按照聚簇索引的顺序存储。一个表只能有一个聚簇索引，通常是主键。

特点：

表中的数据按照聚簇索引的顺序进行物理存储。

主键通常是聚簇索引，但如果主键被定义为非唯一，数据库可能会选择一个唯一非空列作为聚簇索引。

聚簇索引的叶子节点包含实际数据行。

优点：

数据的物理存储顺序与索引顺序一致，减少了磁盘I/O的次数，提高查询性能。

范围查询的效率较高。

缺点：

插入和更新数据时可能导致页的分裂，影响性能。

数据的物理存储顺序与索引顺序紧密相关，不适用于频繁的插入、更新操作。

2. 非聚簇索引（Non-Clustered Index）：

定义： 非聚簇索引定义了数据的逻辑顺序，但实际数据行的物理存储顺序与索引无关。一个表可以有多个非聚簇索引。

特点：

索引的叶子节点不包含实际数据行，而是包含指向实际数据行的指针。

表中的数据在物理上按照一定的方式存储，而不是按照索引的顺序。

优点：

插入和更新数据的性能相对较好，不容易导致页的分裂。

适用于频繁的插入、更新操作。

缺点：

查询性能相对较低，因为查询时需要先通过索引找到实际数据行的指针，再去检索实际数据。

**唯一索引在插入的时候为什么可能比普通索引更慢？**

唯一索引在插入数据时可能比普通索引更慢，主要原因包括以下几点：

1. 唯一性检查：

唯一索引要求索引列的值在整个表中是唯一的。在插入新数据时，数据库系统需要检查索引列的值是否已经存在，以确保新插入的数据满足唯一性约束。这个唯一性检查可能导致插入速度变慢，特别是在表中已经存在大量数据的情况下。

2. 锁和并发控制：

在进行唯一性检查时，数据库系统可能需要获取相应的锁，以确保在检查唯一性的过程中不会有其他并发事务插入相同的值。这可能导致锁的竞争，影响插入操作的并发性能。

3. 数据结构调整：

为了维护唯一性，数据库系统可能需要对索引数据结构进行调整，例如在B树中插入新值或调整相邻节点的位置。这些调整可能会增加插入操作的成本，使得插入速度变慢。

4. 事务回滚：

如果在插入操作中发生了唯一性冲突，数据库系统可能需要执行事务回滚操作，撤销已经进行的插入操作。事务回滚会引入额外的开销，影响插入性能。

5. 维护索引结构的成本：

唯一索引的维护通常比普通索引更为复杂。在插入新数据时，数据库系统需要确保插入的数据不会导致索引中出现重复值，这可能涉及到更复杂的索引结构维护操作，增加了插入的成本。

**SpringBoot有哪些常用的注解**

1. Spring Boot应用注解：

@SpringBootApplication: 用于标识Spring Boot应用的启动类。包含了@Configuration、@EnableAutoConfiguration和@ComponentScan。

2. Controller层注解：

@Controller: 标识控制器类。

@RestController: 结合@Controller和@ResponseBody，常用于RESTful风格的控制器。

@RequestMapping: 映射HTTP请求路径。

3. 服务层注解：

@Service: 标识服务层的类。

@Component: 通用的Spring组件注解，可用于标识任何Spring组件，包括服务层。

4. 数据访问层注解：

@Repository: 标识数据访问层的类。

@Entity: JPA实体类的标识。

@Table: 定义数据库表的注解。

5. RESTful API相关注解：

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping: 映射HTTP请求的GET、POST、PUT、DELETE方法。

@PathVariable: 用于接收路径变量。

@RequestParam: 用于接收请求参数。

6. 数据验证和绑定注解：

@Valid: 配合@RequestBody使用，表示对请求体进行验证。

@ModelAttribute: 用于绑定请求参数到模型对象。

7. 配置相关注解：

@Configuration: 标识配置类。

@Value: 用于注入属性值。

@PropertySource: 指定属性文件的位置。

8. 事务管理注解：

@Transactional: 用于声明事务。

9. 定时任务相关注解：

@Scheduled: 用于声明定时任务。

10. Spring Boot自动配置相关注解：

@EnableAutoConfiguration: 开启Spring Boot的自动配置。

@ConditionalOnProperty: 根据配置文件中的属性条件来决定是否启用某个配置。

11. 测试相关注解：

@SpringBootTest: 用于启动整个Spring应用上下文的测试。

@RunWith(SpringRunner.class): 与JUnit一起使用，提供Spring测试支持。

12. 其他常用注解：

@Autowired: 自动注入依赖。

@Qualifier: 结合@Autowired使用，指定具体的依赖注入。

@ComponentScan: 扫描指定包下的组件。

@Conditional: 根据条件来决定是否创建一个Bean。

**SpringMVC的过程**

客户端发送请求：

客户端（通常是浏览器）发送HTTP请求到Web应用程序。

DispatcherServlet接收请求：

请求被前端控制器 DispatcherServlet 接收。

处理器映射器（Handler Mapping）：

DispatcherServlet 使用处理器映射器（HandlerMapping）根据请求的URL映射到对应的处理器（Controller）。

处理器执行：

映射到的处理器（Controller）执行相应的业务逻辑，通常通过调用服务层或其他组件来处理请求。

ModelAndView对象：

处理器返回一个 ModelAndView 对象，其中包含模型数据和视图名称。

视图解析器（View Resolver）：

DispatcherServlet 使用视图解析器（View Resolver）将逻辑视图名称解析为具体的视图对象。

视图渲染：

视图对象负责渲染模型数据，生成HTML等内容，然后返回给 DispatcherServlet。

响应发送：

DispatcherServlet 将视图生成的内容作为HTTP响应返回给客户端。

在这个过程中，DispatcherServlet 充当前端控制器，负责协调整个请求的处理流程。处理器映射器用于将请求映射到具体的处理器，而视图解析器用于将逻辑视图名称映射到实际的视图。

**Redis的跳表具体是怎么实现的**

**Reactor、多路IO复用了解吗？**

**Redis的cluster，哨兵机制**

**主观下线与客观下线**

**代码规范性如何？**

**反问：问了一下部门业务、有几面？也是想做AGI，两次面试。**

**MySQL的事务特性是怎么实现的**

**隔离级别是怎么实现的，以某个级别为例子说明（从普通select到锁定读到insert分析了一遍**

**Redis的数据结构有哪些？**

**Hash是怎么实现的？**

**消息队列常见的使用场景有哪些？**

**削峰具体怎么做？**

**输入url显示页面的过程？**

**计算机网络分为哪几层？每层做什么有什么协议？**

**数据链路层为什么要成帧？**

**最小生成树算法（我只记得克鲁斯卡尔和Prim这俩人名儿了，大概说了下贪心找最小安全边**

**Go和Java有哪些区别？**

**进程、线程、协程的区别？**

**虚拟内存的作用是什么？**

**MySQL的事务特性是怎么实现的**

**隔离级别是怎么实现的，以某个级别为例子说明（从普通select到锁定读到insert分析了一遍**

**Redis的数据结构有哪些？**

**Hash是怎么实现的？**

**消息队列常见的使用场景有哪些？**

**削峰具体怎么做？**

**输入url显示页面的过程？**

**计算机网络分为哪几层？每层做什么有什么协议？**

**数据链路层为什么要成帧？**

**最小生成树算法（我只记得克鲁斯卡尔和Prim这俩人名儿了，大概说了下贪心找最小安全边**

**Go和Java有哪些区别？**

**进程、线程、协程的区别？**

**虚拟内存的作用是什么？**