计算机基础面试

一、计算机网络

TCP/UDP

- TCP/IP即传输控制协议，是面向连接的协议，发送数据前要先建立连接，TCP提供可靠的服务，也就是说，通过TCP连接传输的数据不会丢失，没有重复，并且按顺序到达。（类似于打电话）
- UDP它是属于TCP/IP协议族中的一种。是无连接的协议，发送数据前不需要建立连接，是没有可靠性的协议。因为不需要建立连接所以可以在在网络上以任何可能的路径传输，因此能否到达目的地，到达目的地的时间以及内容的正确性都是不能被保证的。（类似于发微信）

TCP三次握手四次挥手

三次握手：

1. **客户端发送SYN（同步）报文**： 客户端向服务器发送一个TCP报文，标志位SYN置为1，表示请求建立连接，同时选择一个初始序列号（ClientISN）。
2. **服务器回应SYN-ACK报文**： 服务器收到客户端的SYN报文后，如果同意建立连接，会回应一个TCP报文，标志位SYN和ACK都置为1，表示确认请求，并选择自己的初始序列号（ServerISN）。
3. **客户端发送ACK报文**： 客户端收到服务器的SYN-ACK报文后，向服务器发送一个确认报文，标志位ACK置为1，表示连接建立成功，双方可以开始通信。

四次挥手：

1. **客户端发送FIN报文**： 客户端要关闭连接时，先向服务器发送一个FIN报文，标志位FIN置为1，表示不再发送数据，但仍接收数据。
2. **服务器回应ACK报文**： 服务器收到客户端的FIN报文后，回应一个ACK报文，标志位ACK置为1，表示确认收到关闭请求。
3. **服务器发送FIN报文**： 服务器也要关闭连接时，会向客户端发送一个FIN报文，标志位FIN置为1。
4. **客户端回应ACK报文**： 客户端收到服务器的FIN报文后，回应一个ACK报文，标志位ACK置为1。

- 第一次挥手后进入CLOSE_WAIT状态，第三次挥手后进入TIME_WAIT状态

七层网络

1. 物理层：主要负责传输数据的物理基础，如电缆、无线等。
2. 数据链路层：负责在物理层的传输的基础上，建立通信链接，保证数据的正确传输（以太网、蓝牙、WIFI）。
3. 网络层：负责实现数据的路由和传输，从而实现网络的连通性（IP、ICMP、ARP、RIP、OSPF 等）。
4. 传输层：负责确保数据的可靠传输，如重传、检测丢包等（TCP、UDP 等）。
5. 会话层：负责管理会话的建立、维护和终止。
6. 表示层：负责处理数据的格式，以及对数据进行编码和解码。
7. 应用层：负责实现应用程序的通信（HTTP、FTP、SMTP、DNS）。

HTTP：请求-响应通信协议

请求报文格式：请求行（url、端口、请求方式、HTTP版本）、请求头（编码、服务器地址）、请求体

响应报文格式：状态行、响应头部、响应体

HTTPS：非对称加密的采用SSL/TLS 协议的请求响应协议

WebSocket：双向通信协议，例如动态数据，如实时状态、聊天消息

IPV4/IPV6：32位/64位，并在数据包格式、协议特性、安全性方面做了改进

输入URL到渲染页面的过程：

- DNS域名解析ip地址、建立TCP连接、三次握手四次挥手、HTTP请求和响应、解析HTML及其他资源、渲染页面

状态码

状态码	说明
200	响应成功
301	永久重定向，搜索引擎将删除源地址，保留重定向地址
302	暂时重定向，重定向地址由响应头中的Location属性指定（JSP中Forward和Redirect之间的区别） 由于搜索引擎的判定问题，较为复杂的URL容易被其它网站使用更为精简的URL及302重定向劫持
304	缓存文件并未过期，还可继续使用，无需再次从服务端获取
400	客户端请求有语法错误，不能被服务器识别
403	服务器接收到请求，但是拒绝提供服务（认证失败）
404	请求资源不存在
500	服务器内部错误

套接字

1. **通信端点**：套接字可以被视为通信的两个端点之一，一个端点用于发送数据，另一个端点用于接收数据。在网络中，每台计算机都有一个或多个套接字用于与其他计算机进行通信。
2. **IP地址和端口号**：每个套接字都与一个特定的IP地址和端口号相关联。IP地址用于标识计算机，端口号用于标识计算机上的应用程序。通过IP地址和端口号，套接字能够在网络中找到对应的通信对象。
3. **传输协议**：套接字可以基于不同的传输协议进行通信，如TCP（传输控制协议）和UDP（用户数据报协议）。TCP提供可靠的、面向连接的通信，而UDP提供无连接的、不可靠但更快速的通信。
4. **客户端-服务器模型**：套接字常用于构建客户端-服务器模型的应用程序。在这种模型中，服务器应用程序创建一个套接字并监听特定的端口，等待客户端连接。客户端应用程序创建一个套接字并连接到服务器，之后双方可以通过套接字进行通信。
5. **数据交换**：通过套接字，应用程序可以发送和接收数据。数据可以是文本、二进制、图像、音频等各种形式。
6. **网络编程**：套接字是网络编程的基础，开发者可以使用不同编程语言（如C、Python、Java）提供的套接字API来实现网络通信功能。

滑动窗口协议

滑动窗口协议是一种用于在不可靠的网络上传输数据的协议。发送方将数据分成多个部分，并分配序号，接收方按照序号接收数据，并发送确认信息给发送方，以确保数据的正确传输。发送方会根据接收到的确认信息进行调整，以提高传输效率。滑动窗口协议的窗口大小是一个关键参数，它限制了发送方和接收方未确认数据的数量。

对称加密和非对称加密SSL/TLS协议

HTTPS使用了非对称加密算法和对称加密算法相结合的方式来实现数据加密。其中，非对称加密算法用于在客户端和服务器之间进行密钥交换，而对称加密算法则用于在数据传输过程中对数据进行加密和解密。

对称加密算法使用相同的密钥对数据进行加密和解密，因此密钥需要在客户端和服务器之间共享。这种方式的优点是加密和解密速度快，但缺点是密钥容易被窃取，从而导致数据泄露(AES)。

非对称加密算法则使用一对密钥，分别为公钥和私钥。公钥可以公开，任何人都可以使用它对数据进行加密，但只有持有私钥的人才能解密数据。这种方式的优点是密钥不需要共享，因此更安全，但缺点是加密和解密速度较慢（RSA）。

在HTTPS中，非对称加密算法用于在客户端和服务器之间进行密钥交换，以确保传输的密钥不被窃取。然后，使用对称加密算法对数据进行加密和解密，以保证数据传输的安全性和效率。

HTTP截获报文

1. **代理服务器（Proxy Server）截获**：代理服务器是客户端和目标服务器之间的中介，它可以截获客户端发送的HTTP请求和目标服务器返回的HTTP响应。代理服务器可以在不改变请求的情况下，记录请求和响应的内容。
2. **网络分析工具（Network Sniffer）截获**：网络分析工具可以在网络上截获所有通过该工具所在的网络接口的数据包，包括HTTP请求和响应。网络分析工具可以根据协议类型过滤HTTP请求和响应，以便分析和记录。
3. **浏览器插件（Browser Extension）截获**：浏览器插件可以截获浏览器发送的HTTP请求和接收的HTTP响应。例如，Chrome浏览器的开发者工具可以截获浏览器发送的HTTP请求和接收的HTTP响应，并提供详细的请求和响应报文的分析和调试功能。

HTTP的Header

1. Accept：指定客户端可接受的MIME类型
2. Accept-Encoding：指定客户端可接受的编码类型
3. Authorization：包含用于身份验证的凭证，通常是用户名和密码的Base64编码
4. Cache-Control：指定缓存机制
5. Content-Length：指定请求或响应正文的长度
6. Content-Type：指定请求或响应正文的MIME类型
    - `application/json`：JSON 数据。
    - `application/xml`：XML 数据。
    - `text/plain`：纯文本文件，如 `.txt` 文件。
    - `text/html`：HTML 文件。

    - multipart/form-data：用于在 HTML 表单中上传文件的 MIME 类型。
7. Cookie：包含客户端的Cookie信息
8. Host：指定请求的主机名和端口号
9. If-Modified-Since：指定上次修改时间，用于条件GET请求
10. User-Agent：包含客户端的用户代理信息

TCP标志位

1. **SYN（Synchronize）**：SYN标志位被设置为1时，表示发送方请求建立连接。在三次握手中，客户端向服务器发送的第一个报文中，SYN标志位被设置为1，表示客户端请求建立连接。
2. **ACK（Acknowledgement）**：ACK标志位被设置为1时，表示接收方已经收到了发送方的数据。在三次握手和四次挥手中，ACK标志位用于确认收到对方的报文。
3. **FIN（Finish）**：FIN标志位被设置为1时，表示发送方要关闭连接。在四次挥手中，发送方向接收方发送FIN报文，表示发送方已经没有数据要发送了。

TIME_WAIT状态

主动关闭方在收到被动关闭方的FIN包后并返回ACK后，会进入TIME_WAIT状态，TIME_WAIT状态又称2MSL状态，每个TCP连接都必须有一个最大报文段生存时间MSL，在网络传输中超过这个时间的报文段将被丢弃。当TCP连接发起一个主动关闭，并发出最后一个ACK时，必须在TIME_WAIT状态停留两倍MSL时间，在2MSL等待期间，定义这个连接的插口(客户端IP地址和端口号，服务器IP地址和端口号的四元组)将不能再被使用。2MSL状态存在有两个理由：

1.允许老的重复报文分组在网络中消逝。

2.保证TCP全双工连接的正确关闭。

多路复用

多路复用是指在一个物理通道上，同时传输多个信号或数据流的技术。在计算机网络中，多路复用通常用于在一个连接上同时传输多个数据流，从而提高连接的利用率。

在传统的网络通信中，每个连接都需要占用一个独立的物理通道，这样会导致网络资源的浪费。而多路复用技术可以将多个连接合并在一个物理通道上，从而减少了网络资源的占用。常见的多路复用技术包括分时复用、频分复用和码分复用等。

在计算机网络中，常见的多路复用技术是基于传输控制协议（TCP）的多路复用。TCP多路复用使用一种称为“多路复用器”的软件组件，将多个TCP连接合并在一个物理通道上。这种技术可以提高网络连接的利用率，从而提高网络的吞吐量和性能。

HTTPS的加密过程

第一层：身份验证（Authentication）

- 客户端发起连接请求，服务器返回数字证书。
- 客户端验证服务器证书的合法性，包括检查证书是否由受信任的证书颁发机构（CA）签发、证书是否过期，以及主机名是否匹配。
- 服务器的数字证书包含服务器的公钥和其他信息，用于证明服务器的身份。

第二层：密钥交换（Key Exchange）

- 客户端生成一个随机的对称密钥（共享密钥）。
- 客户端使用服务器的公钥加密生成的共享密钥，并发送给服务器。
- 服务器使用自己的私钥解密客户端发送的共享密钥。

第三层：数据加密（Data Encryption）

- 客户端和服务器现在都有了相同的共享密钥，用于对称加密和解密通信内容。
- 通信内容被分为小块（数据块），每个块使用共享密钥进行对称加密，然后传输到接收方。
- 接收方使用共享密钥进行对称解密，以获取原始数据。

HTTP/2.0和HTTP/3.0

HTTP/1.1：流水线 + PUT、DELETE

HTTP/1.2：服务器推送 + 头部压缩

HTTP/2.0：二进制分帧 + 多路复用

- **性能优化：** HTTP/2.0 主要关注性能优化，引入了多路复用（Multiplexing）机制，允许在单个连接上同时发送和接收多个请求和响应，减少了连接的数量，提高了并发性能。
- **Header 压缩：** 使用 HPACK 算法对请求和响应的头部进行压缩，减少了头部大小，从而减少了带宽消耗。
- **服务器推送：** 服务器可以在客户端请求之前主动推送一些资源，提前加载页面所需的资源，减少页面加载时间。
- **流控制：** 引入了流控制机制，可以根据客户端的处理能力来控制数据的流动，避免了过多数据堆积。
- **二进制分帧：** 使用二进制格式来分帧，取代了 HTTP/1.x 的文本格式，更高效地传输数据。

HTTP/3.0：

- **基于 QUIC 协议：** HTTP/3.0 是基于 QUIC（Quick UDP Internet Connections）协议的，而不是基于传统的 TCP 协议。QUIC 使用 UDP 来提供可靠的、安全的、低延迟的传输。
- **多路复用：** HTTP/3.0 仍然支持多路复用，但它在 QUIC 协议层面已经内置，不需要像 HTTP/2.0 那样在应用层实现。
- **0-RTT 握手：** QUIC 支持 0-RTT（零往返时间）握手，使得连接的建立更快，减少了连接时延。
- **解决队头阻塞问题：** HTTP/3.0 解决了 HTTP/2.0 中存在的队头阻塞问题，避免了单个请求的阻塞影响其他请求的传输。
- **适应性重传：** QUIC 支持自适应重传，可以在丢包时更快地进行重传，提高了传输的可靠性。

二、数据结构

栈与堆的区别

栈和堆都是计算机内存中的存储区域，但它们有以下不同之处：

1. 存储方式：栈是一种后进先出（LIFO）的数据结构，它的存储方式类似于一摞盘子，新的数据存放在最顶部，取数据时也从顶部取出；堆则是一种动态分配内存的机制，它的存储方式是将数据存储在不连续的内存空间中。
2. 内存分配方式：栈的内存分配方式是由编译器自动管理的，它会在程序运行时自动分配和释放内存空间，不需要手动控制；堆的内存分配则需要由程序员手动申请和释放内存空间，否则可能会出现内存泄漏等问题。
3. 访问速度：由于栈的内存结构是连续的，所以访问速度较快；而堆的内存结构是不连续的，访问速度较慢。
4. 分配大小：栈的内存大小固定，通常比较小，一般在几 MB 左右；堆的内存大小不固定，可以根据需要动态分配，可以分配更大的内存空间。
5. 生命周期：栈上的变量的生命周期与其所在函数的生命周期相同，函数执行结束后，它们就会被自动释放；堆上的变量则可以在程序的任意位置被使用和访问，并且只有在程序显式地释放它们时才会被释放。
6. 适用场景：栈通常用于存储程序执行期间的临时数据，比如函数的局部变量、函数参数、函数调用的返回地址等。由于栈的存储方式是后进先出（LIFO），所以它非常适合用于函数调用和返回等场景，能够高效地管理函数的局部变量和临时数据。
   堆则通常用于存储动态分配的数据，比如数组、对象、指针等。由于堆的内存大小不固定，程序可以根据需要动态分配和释放内存空间，所以它非常适合用于动态数据结构和数据存储等场景。但是，由于堆的内存分配需要由程序员手动管理，所以在使用堆时需要注意内存泄漏等问题。

红黑树

红黑树是一种自平衡二叉查找树，它的特点是：

- 每个节点要么是红色，要么是黑色。
- 根节点是黑色的。
- 每个叶子节点（NIL节点，空节点）是黑色的。
- 如果一个节点是红色的，则它的两个子节点都是黑色的。
- 从任意一个节点到其每个叶子节点的所有路径都包含相同数目的黑色节点。

红黑树的应用场景很广泛，例如在C++ STL中的map、set、multimap、multiset等容器中都使用了红黑树来实现。红黑树的时间复杂度比较稳定，可以保证在最坏情况下的查找、插入、删除操作的时间复杂度都是O(log n)。

B+树

B+树是一种多路平衡查找树，它的特点是：

- 每个节点最多有M个子节点。
- 除根节点和叶子节点外，每个节点至少有M/2个子节点。
- 所有叶子节点都在同一层，且包含了全部关键字的信息。

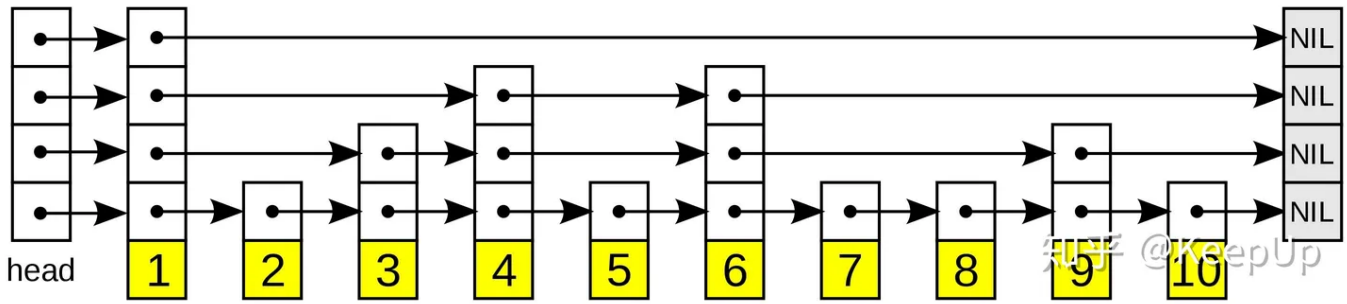
B+树的应用场景主要是在数据库系统中，用于索引和排序。B+树的特点是能够快速地进行范围查询和顺序访问，因为所有的叶子节点都在同一层，并且采用了顺序访问的方式存储数据。B+树的时间复杂度也比较稳定，可以保证在最坏情况下的查找、插入、删除操作的时间复杂度都是O(log n)。

B+树和B-树

1. **节点结构不同**：B+树的非叶子节点只存储键值信息，不存储数据，而叶子节点存储了所有数据，以及指向下一个叶子节点的指针。而B-树的所有节点都存储数据和键值信息。
2. **叶子节点的指针不同**：在B+树中，叶子节点之间形成一个有序链表，可以通过叶子节点的指针遍历整棵树。而在B-树中，每个节点都有指向相邻节点的指针，因此不需要像B+树那样维护一个链表。
3. **查询性能不同**：由于B+树的非叶子节点只存储键值信息，可以存储更多的节点信息，因此B+树的查询性能相对于B-树更好。而B-树的查询性能相对较差，因为每个节点都存储数据和键值信息，导致树的高度较低，但每个节点能够存储的信息也较少。
4. **适用范围不同**：B+树适用于大型数据集合的存储和管理，因为它可以存储更多的节点信息，提高查询性能。而B-树适用于小型数据集合的存储和管理，因为它的节点能够存储的信息较少，但树的高度较低，查询性能相对较好。

跳表

跳表全称为跳跃列表，它允许快速查询，插入和删除一个有序连续元素的数据链表。跳跃列表的平均查找和插入时间复杂度都是O(logn)。快速查询是通过维护一个多层次的链表，且每一层链表中的元素是前一层链表元素的子集（见右边的示意图）。一开始时，算法在最稀疏的层次进行搜索，直至需要查找的元素在该层两个相邻的元素中间。这时，算法将跳转到下一个层次，重复刚才的搜索，直到找到需要查找的元素为止。



Hash冲突

拉链法：每个哈希桶维护一个链表，发生冲突是添加到链表当中

开放寻执法：线性侦测、二次侦测、双重散列

Bitmap

`Bitmap`（位图）是一种常用的数据结构，用于高效地存储和查询大量布尔值（0 或 1）。在实现中，可以使用一个二进制数组来表示每个布尔值的状态，其中每个元素表示一组布尔值。例如，如果每个元素有 8 个二进制位，则可以表示 8 个布尔值。

使用 `Bitmap` 可以在很小的存储空间内存储大量布尔值，并且可以快速计算集合的交集、并集和差集等操作。例如，如果要存储一个有 1 亿个布尔值的集合，使用 `Bitmap` 只需要 12.5MB 的存储空间，而使用普通的数组则需要 100MB 的存储空间。

三、操作系统

进程和线程

- 进程（Process）： 进程是操作系统分配资源和执行代码的基本单位。每个进程都有自己的独立内存空间、代码和数据。不同进程之间不能直接共享数据，通信需要使用进程间通信（IPC）机制。由于进程之间的资源相互隔离，进程间的切换开销较大。每个进程都有自己的执行状态，如运行、就绪、阻塞等。
- 线程（Thread）：线程是进程内的执行单元，一个进程可以包含多个线程。线程共享进程的代码和数据，但拥有独立的栈空间。因为线程共享内存空间，它们之间的通信和数据共享相对容易。线程之间的切换开销较小，但由于共享内存，需要注意线程安全问题。
- 协程（Coroutine）： 协程是一种轻量级的线程，也被称为用户态线程。与操作系统线程不同，协程的调度和切换是由程序员自己控制的，而不是由操作系统。协程可以在同一个线程中切换执行，因此不需要像线程那样频繁的上下文切换。协程通常用于处理高并发、异步任务和IO密集型操作，因为它们可以更高效地处理大量任务的切换和调度。（Quasar、Project Loom）
- 线程状态：就绪、运行、阻塞、等待、终止

地址空间

1. 代码段：存储程序的代码，一般只读。
2. 数据段：存储程序的数据，可读写。
3. 栈：存储程序的函数调用栈，存储函数调用过程中的局部变量等数据。
4. 堆：动态内存分配区域，用于动态分配内存。

悲观锁、乐观锁

乐观锁：是一种不加锁的并发控制策略，在更新数据时，先判断数据是否被修改，如果没有被修改，则更新，否则不更新。乐观锁的思想是相信数据一般不会被其他事务修改，所以不加锁，从而提高效率。

悲观锁：是一种加锁的并发控制策略，在读取数据前，先加锁，在更新数据时也需要加锁，更新完数据后再释放锁。悲观锁的思想是相信数据一定会被其他事务修改，所以对数据加锁，从而保证数据的完整性。

调度算法：先来先服务、最短作业优先、最短剩余时间优先、轮询算法（RR）

页面置换算法：LRU、FIFO、NRU

死锁

死锁：在多线程环境下，多个线程竞争有限的资源造成的僵局

ex：A占有资源1，同时申请资源2；而B占有资源2，同时申请资源1；

死锁条件

1. **互斥条件（Mutual Exclusion）**： 每个资源只能同时由一个进程（线程）占用，其他进程（线程）要想使用该资源必须等待。
2. **请求与保持条件（Hold and Wait）**： 进程（线程）在等待其他资源的同时，仍然持有已获得的资源。
3. **不可抢占条件（No Preemption）**： 已分配给一个进程（线程）的资源不能被强制性地抢占，只能在使用完之后自愿释放。

4. **循环等待条件（Circular Wait）**： 存在一个进程（线程）的资源需求序列，使得每个进程（线程）都在等待下一个进程（线程）释放资源。

预防死锁

1、互斥条件

2、请求和保持条件：阻塞时释放已经获得的资源

3、不剥夺条件：不能获得全部资源则等待

4、环路等待条件：资源编号，进程获得编号小的资源才能获取到大的

线程安全

线程安全指的是多线程环境下，对共享资源的访问是安全的，不会出现数据不一致、数据丢失等问题。在多线程环境中，由于多个线程共同操作一个资源，可能会出现多个线程同时对一个资源进行读写的情况，这就会导致线程安全问题。常见的线程安全问题包括**数据竞争、死锁、饥饿**等。

为了确保线程安全，开发人员通常采用以下几种方式：

1. 加锁：使用同步机制来控制对共享资源的访问，保证同一时间只有一个线程可以访问共享资源。
2. 原子操作：原子操作指的是不能被中断的操作，保证其执行的完整性。
3. 使用线程安全的数据结构：使用线程安全的集合类，例如Vector、ConcurrentHashMap等。
4. 避免共享资源：尽量避免多个线程共同操作同一个资源，避免线程安全问题的出现。

线程同步的方法

1. 互斥锁：互斥锁是一种最常用的线程同步方式，它可以保证在同一时刻只有一个线程可以访问共享资源。当一个线程需要访问共享资源时，它会尝试获取互斥锁，如果锁已经被其他线程占用，则该线程会被阻塞，直到锁被释放。
2. 条件变量：条件变量是一种线程同步机制，它可以让一个线程等待另一个线程的通知。当一个线程需要等待某个条件满足时，它会通过条件变量通知其他线程等待，然后自己进入等待状态。当条件满足时，另一个线程会通过条件变量通知等待线程，使其重新开始执行。
3. 信号量：信号量是一种计数器，用于控制多个线程对共享资源的访问。当一个线程需要访问共享资源时，它会尝试获取信号量，如果信号量的值为0，则该线程会被阻塞，直到另一个线程释放信号量。
4. 屏障：屏障是一种同步机制，它可以让多个线程在某个点上等待，直到所有线程都到达该点后再继续执行。屏障可以用于处理多个线程之间的依赖关系，例如计算任务中的数据依赖关系。
5. 原子操作：原子操作是一种特殊的操作，它可以保证在同一时刻只有一个线程可以访问共享资源。原子操作可以用于处理一些简单的数据结构，例如计数器和标志位等。

内存管理模型

1. 静态内存分配：在程序编译时就确定了内存分配的大小和位置，程序运行时无法改变。这种内存分配方式的优点是速度快、简单易用，但是它的缺点是浪费内存空间，因为程序运行时可能不需要分配全部的内存。
2. 栈式内存分配：栈是一种后进先出的数据结构，栈式内存分配就是将内存按照栈的方式进行分配，先分配的内存地址较低，后分配的内存地址较高。栈式内存分配的优点是速度快、内存利用率高，但是它的缺点是分配的内存大小有限，无法动态改变。

3. 堆式内存分配：堆是一种动态分配内存的机制，程序可以在运行时根据需要动态地分配和释放内存。堆式内存分配的优点是灵活性高，可以根据程序的需要动态地分配和释放内存，但是它的缺点是容易出现内存泄漏和内存碎片等问题。
4. 内存池管理：内存池是一种将内存预先分配好，然后在程序运行时动态地分配和释放内存的机制。内存池管理的优点是减少了内存分配和释放的开销，提高了程序的运行效率，但是它的缺点是需要提前预估程序所需要的内存大小，如果预估不准确，可能会浪费内存空间。

四、数据库

ACID

原子性（Atomicity）：事务是数据库的逻辑工作单位，事务中包括的诸操作要么都做，要么都不做（数据库日志和锁）

一致性（Consisency）： 事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态

隔离性（Isolation）：一个事务的执行不能被其他事务干扰（MVCC版本控制）

持久性（Durability）：一个事务一旦提交，它对数据库中数据的改变就应该 是永久性的。

事物隔离级别

InnoDB默认隔离级别：可重复读

- 读未提交（Read Uncommitted）：在该隔离级别下，一个事务可以读取其他事务未提交的数据，可能导致脏读、不可重复读和幻读的情况。
- 读已提交（Read Committed）：在该隔离级别下，一个事务只能读取已经提交的数据，可以避免脏读，但仍有不可重复读和幻读的情况。
- 可重复读（Repeatable Read）：在该隔离级别下，一个事务在执行过程中所读取的数据不会被其他事务修改，但可能存在幻读的情况。
- 串行化（Serializable）：在该隔离级别下，每个事务都是独占的，不会有其他事务干扰，可以避免脏读、不可重复读和幻读的情况，但效率较低。
- 脏读：是指一个事务读取到了另一个事务未提交的修改数据，这种情况下，如果另一个事务回滚了，那么当前事务读到的数据就是不正确的。
  不可重复读：是指一个事务读取同一数据多次，但是每次读取的结果不同。这是由于在读取之间，另一个事务修改了这个数据，导致读取的结果不同。

幻读：是指一个事务读取到了不存在的数据，或者读取到了已经删除的数据，因为另一个事务在读取期间插入了数据或者删除了数据。这种情况对于数据的完整性和一致性造成了严重影响。

数据库引擎

InnoDB（支持事物、聚簇索引、行级锁）、MyISAM（锁、索引、备份和迁移）、Memory（内存引擎）

数据库日志

redolog：重做、undolog：回滚、binlog：二进制

MySQL索引

1. PRIMARY KEY 索引：是唯一索引，可以用来标识表中的每一行数据。
2. UNIQUE 索引：是唯一索引，但不同于 PRIMARY KEY 索引，可以有空值。
3. INDEX 索引：是普通索引，可以允许多个相同的值。
4. FULLTEXT 索引：是全文索引，用于搜索关键字和短语。
5. SPATIAL 索引：是空间索引，用于处理空间数据。

B+索引和Hash索引：B+ 索引的优点在于，它提供了快速查询的能力，同时还支持区间查询和排序。

回表

回表是指在数据库查询过程中，当使用了非聚集索引（例如B树索引）时，数据库首先在索引中找到满足条件的行的位置，然后需要根据找到的位置再去主表（或者数据行的存储位置）中找到具体的数据行。这个“回”指的是从索引回到主表，找到实际的数据。

数据库调优

创建索引、避免在索引上使用计算、调整where连接顺序（过滤掉最大数量记录）、使用where调换having、使用表的别名、避免使用游标、使用varchar替换char、用具体的字段代替*、update尽量不要更改全部、使用select into替代insert

慢查询优化：

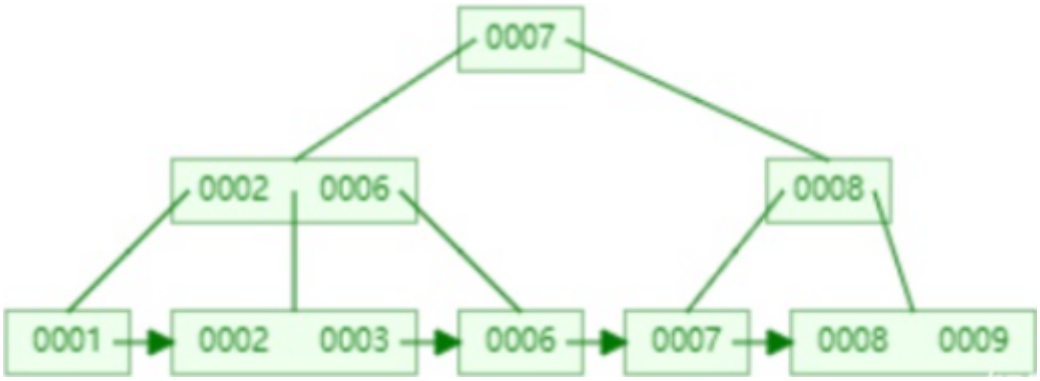
1. 定期清理无用索引：无用索引的存在会影响查询性能，因此需要定期清理无用索引。可以通过查询information_schema库中的表统计信息，找出未使用的索引并进行清理。
2. 优化查询语句：优化查询语句可以改进查询性能。一般可以从以下几个方面入手：优化SQL语句结构、增加索引、避免使用复杂查询、减少重复查询等。

聚簇索引和非聚簇索引

- 聚簇索引：将数据存储与索引放到了一块，找到索引也就找到了数据
- 非聚簇索引：将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行，myisam通过key_buffer把索引先缓存到内存中，当需要访问数据时（通过索引访问数据），在内存中直接搜索索引，然后通过索引找到磁盘相应数据，这也就是为什么索引不在key buffer命中时，速度慢的原因

B+树索引

相邻叶子节点是通过指针连起来的，并且是关键字大小排序的。



数据库的锁机制

1. 共享锁：允许多个事务同时对同一数据进行读取操作，但不允许写操作，保证数据的一致性。在事务读取数据时，会先申请共享锁，如果该数据未被排他锁占用，则可获取共享锁并进行读取，如果该数据被排他锁占用，则需要等待排他锁的释放才能获取共享锁。
2. 排他锁：一次只允许一个事务对数据进行修改或删除操作，保证数据的完整性。在事务修改或删除数据时，会先申请排他锁，如果该数据未被其他锁占用，则可获取排他锁并进行修改或删除，如果该数据被其他锁占用，则需要等待其他锁的释放才能获取排他锁。
3.
    - 记录锁（Record Lock）：仅锁定一行记录（如固定）
    - 间隙锁（Gap Lock）：锁定一个范围，但不包含记录本身
    - 临键锁（Next-key Lock）：锁定一个范围，并且锁定记录本身

主要有以下类型：

1. 行级锁：锁定数据行，只对单行数据进行加锁，其他行数据不受影响，适用于高并发场景，但可能会增加锁冲突的概率和锁开销。
2. 表级锁：锁定整张表，适用于并发量较小的场景，但可能会导致锁等待时间过长、锁冲突较多等问题。
3. 页级锁：锁定数据页，每个数据页可以包含多行数据，对一页数据进行加锁，适用于中等并发量的场景，可以减少锁开销和锁冲突。
4. 数据库级锁：锁定整个数据库，对整个数据库进行加锁，适用于并发量很小的场景，但可能会导致大量锁等待和锁冲突。

索引的优缺点

索引的优点：
① 建立索引的列可以保证行的唯一性，生成唯一的rowId

② 建立索引可以有效缩短数据的检索时间

③ 建立索引可以加快表与表之间的连接

④ 为用来排序或者是分组的字段添加索引可以加快分组和排序顺序

索引的缺点：
① 创建索引和维护索引需要时间成本，这个成本随着数据量的增加而加大

② 创建索引和维护索引需要空间成本，每一条索引都要占据数据库的物理存储空间，数据量越大，占用空间也越大（数据表占据的是数据库的数据空间）

③ 会降低表的增删改的效率，因为每次增删改索引需要进行动态维护，导致时间变长

覆盖索引：包含了查询中所需的所有列，从而可以在索引本身中完成查询，而无需再回到原始数据表中查找。

缓存问题

缓存穿透：数据本来就不存在，但是疯狂请求；

缓存雪崩：大量key值在同一个时间段过期；

缓存击穿：大量查询集中于一个或者几个key，突然这个key过期了；

CAS实现乐观锁

CAS 算法是一种原子操作，可以在不使用锁的情况下更新共享变量。它通过比较内存中的值与预期值，如果相等则更新，否则不更新。CAS 的操作过程可以分为三个步骤：

1. 读取共享变量的当前值；
2. 比较共享变量的当前值与预期值；
3. 如果相等，则更新共享变量的值，否则不更新。

在 MySQL 中实现乐观锁通常可以通过以下步骤：

1. 在表中添加一个版本号（version）字段，用于记录每次修改的版本号；
2. 在更新数据时，通过查询当前版本号是否与预期版本号相等来判断是否有其他并发事务修改了数据；
3. 如果版本号相等，则执行更新操作并将版本号加 1，否则返回更新失败。

索引失效场景

1. 对索引列使用函数或表达式：如果在查询中对索引列使用函数或表达式，那么索引就无法使用。例如，对于索引列name，查询"WHERE UPPER(name) = 'JOHN'"就无法使用索引。
2. 对索引列进行类型转换：如果在查询中对索引列进行类型转换，那么索引就无法使用。例如，对于索引列age，查询"WHERE age = '20'"就无法使用索引。
3. 对索引列进行计算：如果在查询中对索引列进行计算，那么索引就无法使用。例如，对于索引列price，查询"WHERE price * quantity > 1000"就无法使用索引。
4. 使用LIKE查询：如果在查询中使用LIKE操作符，那么索引可能无法使用。例如，查询"WHERE name LIKE '%JOHN%'"，如果没有以通配符开头，那么索引可以使用；但如果查询"WHERE name LIKE '%JOHN'"，那么索引就无法使用。
5. 数据分布不均匀：如果索引列中的数据分布不均匀，那么索引可能无法使用。例如，如果一个表中有1000行数据，其中有900行的age列值为20，那么使用age列作为索引就无法提高查询效率。

InnoDB索引

- 支持事务：InnoDB 索引支持事务，可以在事务中进行并发操作，保证数据的一致性。
- 支持行级锁定：InnoDB 索引支持行级锁定，可以在并发访问时避免数据冲突。
- 支持聚簇索引：InnoDB 索引支持聚簇索引，可以将表按照主键顺序存储，提高查询效率。
- 支持覆盖索引：InnoDB 索引支持覆盖索引，可以在索引中找到需要的数据，避免了对表的全表扫描。

MVCC实现事物隔离

在MVCC中，每个数据行都有一个版本号或时间戳。当一个事务开始时，它会读取该时刻的数据库状态，并在执行期间只看到在该时刻之前提交的版本。如果其他事务在这个事务执行期间修改了同一行数据，那么这个事务会看到该行的旧版本，而不会看到其他事务修改后的新版本。

在实现MVCC时，需要考虑以下几个方面：

1. 版本的创建和维护：每个数据行需要创建多个版本，并在事务提交时删除旧版本。同时，需要维护每个版本的时间戳或版本号，以便事务可以读取正确的版本。
2. 事务的启动和提交：在事务启动时，需要记录该事务的开始时间戳或版本号。在事务提交时，需要删除该事务创建的所有版本，并更新其他事务可以看到的版本号。
3. 并发控制：为了保证事务的隔离性，需要对并发访问进行控制。通常使用锁或CAS（比较-交换）操作来控制并发访问。

版本号

系统版本号：是一个递增的数字，每开始一个新的事务，系统版本号就会自动递增。

事务版本号：事务开始时的系统版本号。 InooDB 的 MVCC 在每行记录后面都保存着两个隐藏的列，用来存储两个版本号；

创建版本号：指示创建一个数据行的快照时的系统版本号；

删除版本号：如果该快照的删除版本号大于当前事务版本号表示该快照有效，否则表示该快照已经被删除 了。

Next-key Locks

它是 Record Locks 和 Gap Locks 的结合，不仅锁定一个记录上的索引，也锁定范围内的索引（临键锁）。

MVCC 不能解决幻读的问题，Next-Key Locks 就是为了解决这个问题而存在的。

在可重复读（REPEATABLE READ）隔离级别下，使用 MVCC + Next-Key Locks 可以解决幻读问题。

Datetime和Timestamp

Datetime默认为空，Timestamp默认为当前时间。

DATETIME 存储的范围是从 '1000-01-01 00:00:00' 到 '9999-12-31 23:59:59'，精度为秒。而 TIMESTAMP 存储的范围是从 '1970-01-01 00:00:01' UTC 到 '2038-01-19 03:14:07' UTC，精度为秒或更高级别的精度，如毫秒或微秒。

另一个区别是存储方式不同。DATETIME 存储的值是一个固定长度的字符串，需要 8 个字节的存储空间。而 TIMESTAMP 存储的值是一个 4 字节的整数，表示自 1970 年 1 月 1 日 00:00:01 UTC 以来的秒数。

此外，TIMESTAMP 还有一个特殊的属性，即它会自动更新。如果在创建表时将 TIMESTAMP 列指定为 DEFAULT CURRENT_TIMESTAMP，则每次插入新行时，TIMESTAMP 列都会自动设置为当前时间。如果更新表中的行，则 TIMESTAMP 列也会自动更新为当前时间。

主从复制

作用：

- 数据备份：通过将数据复制到从服务器上，可以保证在主服务器出现故障时，从服务器上仍然有完整的数据备份。
- 负载均衡：通过将读操作分摊到从服务器上，可以减轻主服务器的负载，提高整个系统的性能。
- 高可用性：当主服务器出现故障时，可以快速切换到从服务器上，从而保证整个系统的高可用性。
- 数据分析：通过在从服务器上进行数据分析，可以避免对主服务器的性能产生影响。

MySQL的QPS从1-10000变化：

1. 当QPS从1到10时，MySQL的性能变化不太明显。这是因为MySQL的默认配置可以轻松处理这个级别的负载。
2. 当QPS从10到100时，MySQL的性能开始有所下降。这是因为随着查询数量的增加，MySQL需要处理更多的连接和查询请求，可能会出现性能瓶颈。
3. 当QPS从100到1000时，MySQL的性能下降更为明显。这是因为MySQL需要处理更多的连接和查询请求，可能会出现锁竞争、磁盘I/O瓶颈等问题，需要对MySQL的配置和硬件进行优化。
4. 当QPS从1000到10000时，MySQL的性能可能会达到极限。这是因为MySQL需要处理大量的连接和查询请求，可能会出现CPU、内存、磁盘等多个方面的瓶颈，需要对MySQL的配置和硬件进行深度优化。

数据库范式

1. **第一范式（1NF）：** 每个属性的值都是不可再分的，即每个单元格内只包含一个值。这可以确保每个数据项都是原子性的，不再是集合或数组。
2. **第二范式（2NF）：** 满足1NF的基础上，非主键属性完全依赖于候选键。换句话说，每个非主键属性都必须完全依赖于主键，而不能依赖于主键的一部分。这样可以消除部分依赖，减少数据冗余。
3. **第三范式（3NF）：** 满足2NF的基础上，消除了传递依赖。换句话说，非主键属性不应该依赖于其他非主键属性。这有助于进一步减少冗余，并提高数据更新的一致性。
4. **巴斯-科德范式（BCNF）：** 在3NF的基础上，消除了主属性对候选键的部分依赖。这种范式确保了每个非主键属性都直接依赖于候选键，没有冗余的非主键依赖关系。
5. **第四范式（4NF）：** 满足BCNF的基础上，消除了多值依赖。这个范式处理一个表中存在多个相互独立的多值依赖的情况。
6. **第五范式（5NF）或投影连接范式（PJ/NF）：** 也称为完美范式，处理多重集合依赖的问题，确保数据不会因为多值依赖引起数据冗余。

五、消息队列

为什么要用RabbitMQ

流量消峰：将短时间内需要处理的订单，全部分散到队列内一次处理。

应用解耦：减少系统之间的调用，防止以者破坏，整个系统崩溃。

异步处理：异步非阻塞，异步编程处理完后可以通知需要的程序。

RabbitMQ优缺点

RabbitMQ的优点包括：

1. 异步消息传递：RabbitMQ允许应用程序在不同的进程和不同的机器之间进行异步消息传递，避免了系统崩溃的风险。
2. 可靠性：RabbitMQ采用了多种机制来保证消息传递的可靠性，包括持久化、重试、错误处理等。
3. 可扩展性：RabbitMQ允许在不停机的情况下增加或减少队列、节点等。
4. 灵活性：RabbitMQ支持多种消息模型，包括点对点、发布/订阅等，还支持多种交换机类型、路由方式等。

RabbitMQ的缺点包括：

1. 性能瓶颈：在高并发场景下，RabbitMQ可能会成为系统的性能瓶颈。
2. 配置复杂：RabbitMQ需要根据业务需求进行配置，对于不熟悉其机制的用户而言，配置可能会比较复杂。
3. 代码侵入性：使用RabbitMQ需要编写与其相关的代码，会对代码结构产生侵入性。

核心概念

1. 生产者（Producer）：发送消息的程序。
2. 消息队列（Message Queue）：存储消息的队列。
3. 消费者（Consumer）：接收并处理消息的程序。
4. 绑定（Binding）：生产者将消息发送到交换机时需要指定一个绑定键，消费者可以根据绑定键从交换机中接收消息。
5. 交换机（Exchange）：接收生产者发送的消息，并根据绑定键将消息发送到相应的队列中。

消息传递模型：AMQP (Advanced Message Queuing Protocol)，生产者-消费者模型，其中生产者将消息发送到队列中，而消费者从队列中接收并处理消息。

启动命令：rabbit-service start

消息模型

点对点

消息生产者向消息队列中发送了一个消息之后，只能被一个消费者消费一次

发布/订阅

消息生产者向频道发送一个消息之后，多个消费者可以从该频道订阅到这条消息并消费。

发布/订阅与观察者模式的区别

- 观察者模式中，观察者和主题都知道对方的存在；而在发布与订阅模式中，发布者与订阅者不知道对方的存 在，它们之间通过频道进行通信。
- 观察者模式是同步的，当事件触发时，主题会去调用观察者的方法，然后等待方法返回；而发布与订阅模式 是异步的，发布者向频道发送一个消息之后，就不需要关心订阅者何时去订阅这个消息。

消息过期机制

1. **消息 TTL（Time To Live）：** 消息发布时可以设置消息的 TTL，即消息的生存时间。消息的 TTL 可以通过设置消息的属性或在发送消息时指定。一旦消息在队列中存活的时间超过了设定的 TTL，它会被自动从队列中删除。

   示例代码（使用 AMQP 0-9-1 协议）：

   ```
   javaCopy codechannel.basicPublish(exchange, routingKey, new
   AMQP.BasicProperties.Builder()
           .expiration("60000") // 设置 TTL 为 60 秒
           .build(), message.getBytes());
   ```

2. **队列 TTL：** 可以为队列设置过期时间，这样队列中的所有消息都会在过期后被自动删除。当队列的 TTL 到期时，队列及其内容将被删除，无论队列中是否有未过期的消息。

   示例代码：

   ```
   javaCopy codeMap<String, Object> args = new HashMap<>();
   args.put("x-expires", 60000); // 设置队列 TTL 为 60 秒
   channel.queueDeclare(queueName, false, false, false, args);
   ```

死信队列

死信队列（Dead Letter Queue，简称DLQ）是一种用于处理无法被正常消费和处理的消息的特殊队列。当消息在正常情况下无法被成功消费时，这些消息会被发送到死信队列中，以便后续的特殊处理和分析。死信队列通常用于以下情况：

1. **消息消费失败：** 当消息在消费过程中发生错误，无法被成功处理时，可以将这些无法处理的消息发送到死信队列。
2. **消息超时：** 如果消息在一定时间内没有被消费，可能表示消费者处理速度过慢或者消费者出现了问题。这种情况下，可以将超时的消息放入死信队列。
3. **重试次数达到上限：** 如果一个消息经过多次重试后仍然无法被成功处理，可以将达到重试次数上限的消息发送到死信队列。
4. **业务处理失败：** 如果某些特殊情况下，业务逻辑无法成功处理消息，也可以将这些消息发送到死信队列，以便后续分析。

解决方案

1. **创建死信交换机和队列：**
    - 首先，你需要创建一个用于存放死信的交换机（Exchange）和队列（Queue）。
    - 创建一个新的交换机，用于路由死信消息到死信队列。
    - 创建一个新的队列，作为死信队列，用于存放无法被正常消费的消息。
2. **设置原始队列的死信配置：**
    - 在你需要进行处理的原始队列上，配置死信交换机和死信路由键。当消息成为死信时，会被发送到死信交换机，并通过死信路由键路由到死信队列。
    - 可以通过 `x-dead-letter-exchange` 和 `x-dead-letter-routing-key` 参数进行配置。
3. **消息成为死信的条件：**
    - 当消息在原始队列中满足某些条件时，例如消息重试次数超过阈值、消息超时等，它就会成为死信，被发送到死信队列。
4. **消费死信队列：**
    - 创建一个消费者来消费死信队列中的消息。
    - 在死信队列的消费者中，可以根据具体业务逻辑来处理死信消息，例如记录日志、通知管理员、重新发送等。

六、Linux基础

Linux命令

1. ls：列出目录中的文件和子目录。
2. cd：切换当前目录。
3. pwd：显示当前目录的路径。
4. mkdir：创建新目录。
5. rmdir：删除空目录。
6. rm：删除文件或目录。
7. cp：复制文件或目录。
8. mv：移动或重命名文件或目录。
9. cat：查看文件内容。
10. grep：在文件中查找匹配的字符串。
11. top：实时显示系统的进程状态。
12. ps：列出当前系统中的进程信息。
13. kill：终止指定的进程。
14. tar：打包或解压文件或目录。[-zxvf 解压；-zcvf 压缩]
15. chmod：修改文件或目录的权限。
16. chown：修改文件或目录的所有者。
17. ssh：远程登录到另一台计算机。

18. scp：安全地复制文件或目录到远程计算机。
19. ping：测试网络连接。
20. ifconfig：显示网络接口信息
21. netstat：查看端口信息
22. df：磁盘使用情况
23. tail：查看日志信息

Linux调优

1. 内存调优：可以通过修改内核参数、调整虚拟内存、使用swap分区等方法来优化系统的内存使用。例如，可以通过修改swappiness参数来控制系统在何时使用swap分区。
2. 磁盘调优：可以通过使用SSD硬盘、调整磁盘调度程序、使用RAID等方法来优化系统的磁盘性能。
3. CPU调优：可以通过使用多核CPU、调整CPU频率、调整系统调度程序等方法来优化系统的CPU性能。
4. 网络调优：可以通过调整TCP缓冲区、使用高效的网络协议、优化网络拓扑结构等方法来优化系统的网络性能。
5. 应用程序调优：可以通过优化应用程序的代码、使用高效的算法、优化数据库等方法来提高应用程序的性能。

I/O多路复用

1. **IO多路复用**： IO多路复用是一种网络编程中的技术，用于同时监视多个文件描述符的IO状态。它可以帮助应用程序在等待多个IO事件的时候，避免阻塞等待，从而提高程序的效率和性能。当某个文件描述符就绪（例如可以读或写）时，程序可以立即进行相应的IO操作，而不需要逐个轮询检查。
2. **select**： `select` 是一种最早出现的IO多路复用技术。它允许程序同时监视多个文件描述符，并在其中任何一个就绪时通知应用程序。然而，`select` 有一些限制，比如在监视的文件描述符数量上限，以及每次调用都需要将所有监视的文件描述符传递给它。
3. **poll**： `poll` 是对 `select` 的一种改进，它也允许程序监视多个文件描述符，但在一些方面对 `select` 进行了优化。`poll` 不再有文件描述符数量上限，它使用一个数组来存放待监视的文件描述符，并且不需要像 `select` 那样每次都重新传递这个数组。
4. **epoll**： `epoll` 是在 Linux 系统上出现的一种高效的IO多路复用机制。它克服了 `select` 和 `poll` 的一些缺点，使得在大规模并发连接的情况下性能更好。`epoll` 采用事件驱动的方式，使用内核事件表来管理文件描述符，能够追踪并管理大量的连接，只会在有事件到来时通知应用程序。

epoll

Epoll（事件轮询）是一种在Linux操作系统中用于高效处理大量文件描述符（sockets、文件、设备等）的I/O事件的机制。它是一种异步事件驱动的I/O模型，用于监视多个文件描述符上的事件，并在这些事件发生时通知应用程序。Epoll的设计目标是提供高性能的事件通知机制，以减少I/O操作的开销，适用于高并发的网络编程。

Epoll的主要优点包括：

1. **高效性能：** Epoll采用了基于事件驱动的模式，使得应用程序不需要通过轮询来等待I/O事件的发生，从而减少了系统调用的次数，提高了系统的性能。
2. **支持大量文件描述符：** Epoll使用红黑树（Red-Black Tree）和链表数据结构来存储文件描述符，使其能够有效地管理大量的I/O事件。

3. **边沿触发和水平触发模式：** Epoll支持两种触发模式。在边沿触发模式下，只有当状态发生变化时（如socket从不可读变为可读），Epoll才会通知应用程序。而在水平触发模式下，只要文件描述符处于可读或可写状态，Epoll就会通知应用程序。
4. **内存映射：** Epoll允许应用程序将内核空间的一部分内存映射到自己的地址空间，从而可以直接操作这些内存，提高了I/O操作的效率。

七、设计模式

常见设计模式

- **单例模式（Singleton Pattern）：** 确保一个类只有一个实例，并提供一个全局访问点。
- **工厂模式（Factory Pattern）：** 定义一个创建对象的接口，但允许子类决定要实例化的类。
- **抽象工厂模式（Abstract Factory Pattern）：** 提供一个接口用于创建相关或依赖对象的家族，而不需要指定具体的类。
- **建造者模式（Builder Pattern）：** 将一个复杂对象的构建过程与其表示分离，使得同样的构建过程可以创建不同的表示。
- **原型模式（Prototype Pattern）：** 使用原型实例来指定创建对象的种类，并通过复制这些原型来创建新的对象。
- **适配器模式（Adapter Pattern）：** 将一个类的接口转换成客户端所期望的另一个接口，使得原本不兼容的类能够一起工作。
- **装饰器模式（Decorator Pattern）：** 动态地给一个对象添加一些额外的职责，而不需要修改其代码。
- **代理模式（Proxy Pattern）：** 为其他对象提供一种代理以控制对这个对象的访问。
- **观察者模式（Observer Pattern）：** 定义对象间的一种一对多的依赖关系，使得当一个对象状态改变时，所有依赖它的对象都会得到通知。
- **策略模式（Strategy Pattern）：** 定义一系列算法，将它们封装起来，并使它们可以互相替换。
- **模板方法模式（Template Method Pattern）：** 定义一个算法的框架，将一些步骤的具体实现延迟到子类。
- **状态模式（State Pattern）：** 允许一个对象在其内部状态改变时改变其行为。
- **备忘录模式（Memento Pattern）：** 在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。
- **迭代器模式（Iterator Pattern）：** 提供一种顺序访问聚合对象中各个元素的方法，而又不暴露其内部的表示。
- **桥接模式（Bridge Pattern）：** 将抽象部分与它的实现部分分离，使它们都可以独立地变化。

六大原则

1. 单一职责原则（Single Responsibility Principle，SRP）： 一个类应该仅有一个引起它变化的原因，即一个类只负责一项职责。这样可以使类的设计更加简洁，易于理解和维护。
2. 开放封闭原则（Open-Closed Principle，OCP）： 软件实体（类、模块、函数等）应该对扩展开放，对修改封闭。即在不修改现有代码的情况下，通过扩展来实现新的功能或变化。
3. 里氏替换原则（Liskov Substitution Principle，LSP）： 子类对象应该能够替换父类对象，并且程序的行为不会受到影响。子类应该保持父类的行为规范，而不是改变原有的行为。

4. 依赖倒置原则（Dependency Inversion Principle，DIP）： 高层模块不应该依赖于低层模块，它们应该共同依赖于抽象接口。抽象不应该依赖于具体实现，具体实现应该依赖于抽象。
5. 接口隔离原则（Interface Segregation Principle，ISP）： 客户端不应该依赖于它不需要的接口。一个类不应该强迫其子类实现不需要的方法。
6. 迪米特法则（Law of Demeter，LoD）： 一个对象应该对其他对象保持最少的了解，只与直接朋友进行交流。直接朋友指的是该对象本身、该对象所创建的对象、该对象所依赖的对象以及位于成员变量、方法参数等地方的对象。

八、前端基础

webpack的作用

1. 模块化处理：将代码按照模块化的方式组织起来，方便代码的维护和管理。
2. 编译打包：将各种前端资源进行编译和打包，使其能够被浏览器识别和使用。
3. 代码优化：对代码进行压缩、去重等优化操作，减小代码体积，提高页面加载速度。
4. 静态资源管理：管理静态资源，如图片、字体等，方便页面的加载和使用。
5. 开发调试：提供了丰富的插件和工具，方便开发调试，提高开发效率。

webpack常用命令

1. webpack：执行默认的打包操作。
2. webpack --config：指定配置文件路径。
3. webpack --watch：监控文件变化并自动重新打包。
4. webpack-dev-server：启动一个开发服务器。
5. webpack --mode：指定打包模式（development、production）。
6. webpack --progress：显示打包进度。
7. webpack --color：打包输出带有颜色。
8. webpack --display-modules：显示打包模块信息。
9. webpack --display-reasons：显示打包模块被引入的原因。
10. webpack --display-error-details：显示打包错误详情。

跨域问题

跨域问题是由浏览器的同源策略引起的，同源策略要求协议、域名、端口号必须完全一致才可以相互通信。如果不满足同源策略，就会导致跨域问题。

解决跨域问题的方法有以下几种：

1. JSONP：利用 `script` 标签可以跨域的特性，将需要获取的数据包装在一个回调函数中，前端通过该回调函数获取数据。缺点是只能用于 `get` 请求。
2. CORS（跨域资源共享）：是一种浏览器允许跨域访问资源的机制，需要服务器设置相应的响应头。
3. 代理：在服务器端设置代理服务器，将跨域请求转发到目标服务器，再将结果返回给前端。
4. iframe：通过在当前页面插入一个隐藏的 `iframe`，将需要跨域的页面嵌入到 `iframe` 中，前端通过操作 `iframe` 中的页面来实现跨域。

5. WebSocket：WebSocket 协议不受同源策略的限制，可以在客户端与服务端之间建立一条长连接，实现实时通信。

面试

1. JavaScript基础知识：变量、数据类型、作用域、闭包、this、原型链、事件循环、异步编程等等。
2. HTML和CSS：HTML标签、语义化、CSS盒模型、选择器、布局、响应式设计、CSS预处理器等等。
3. 框架和库：React、Vue、Angular等前端框架或库的原理、生命周期、组件通信、性能优化等等。
4. 网络和安全：HTTP协议、缓存、跨域、XSS、CSRF等前端安全问题。
5. 工程化和工具：Webpack、Babel、ESLint等前端工具的使用和原理、模块化、性能优化等等。
6. 算法和数据结构：数组、链表、栈、队列、排序算法、查找算法等等。
7. 项目经验：面试官可能会问到你在过去的项目中遇到的问题、如何解决问题、团队协作等等。

js数据类型

1. 数值（Number）：代表数字，例如 42 或 3.14159。
2. 字符串（String）：代表文本，例如 "Hello, world!"。
3. 布尔值（Boolean）：代表真假，即 true 或 false。
4. undefined：当变量被声明但未赋值时，其值为 undefined。
5. null：表示一个空值或缺失值。
6. 对象（Object）：表示一个可变的键值对集合。对象可以包含其他对象、数组、函数等。
7. 符号（Symbol）：表示唯一的标识符，通常用于对象的属性名。

作用域

全局作用域：全局作用域是指变量或函数在代码的任何地方都可以被访问。

局部作用域：局部作用域是指变量或函数只能在其定义的代码块内部访问。

闭包

在函数内部创建的一个独立的作用域，该作用域内包含了函数定义时所能访问到的所有局部变量和参数，即使在函数执行完毕后，这些变量依然存在于内存中，可以在函数外部被访问到。

原型链

当我们访问一个对象的属性或方法时，JavaScript会首先在该对象本身查找该属性或方法，如果找不到，就会去该对象的原型对象（父对象）中查找，如果还找不到，就会继续沿着原型链向上查找，直到找到该属性或方法或者到达原型链的顶端（即Object.prototype）为止。

事件循环

实现异步编程的机制，在JavaScript中，事件循环分为两个阶段：执行栈和任务队列。执行栈是一个存储函数调用的栈结构，每当一个函数被调用时，它就会被压入执行栈中，当函数执行完毕时，它就会从执行栈中弹出。

任务队列是一个存储事件、回调函数等异步任务的队列，当异步任务完成时，它就会被放入任务队列中。当执行栈中的所有任务都执行完毕时，事件循环就会从任务队列中取出一个任务并将其放入执行栈中执行，这个过程会一直循环下去，直到任务队列为空。

vue生命周期

1. beforeCreate：在实例初始化之后，数据观测 (data observer) 和 event/watcher 事件配置之前被调用。在这个阶段，实例的属性和方法都不能被访问。
2. created：在实例创建完成后被立即调用。在这个阶段，实例已完成以下的配置：数据观测 (data observer)，属性和方法的运算，watch/event 事件回调。然而，挂载阶段还没开始，$el 属性目前不可见。
3. beforeMount：在挂载开始之前被调用：相关的 render 函数首次被调用。
4. mounted：实例被挂载后调用，这时 el 被新创建的 vm.$el 替换了。如果根实例挂载到了一个文档内的元素上，当 mounted 被调用时 vm.$el 也在文档内。
5. beforeUpdate：数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁之前。这里适合在更新之前访问现有的 DOM，比如手动移除已添加的事件监听器。
6. updated：由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。当这个钩子被调用时，组件 DOM 已经更新，所以你现在可以执行依赖于 DOM 的操作。然而在大多数情况下，你应该避免在此期间操作 DOM，因为任何的子组件都可能在更新时被激活。
7. beforeDestroy：实例销毁之前调用。在这一步，实例仍然完全可用。
8. destroyed：实例销毁后调用。这个时候，所有的事件监听器都被移除，所有的子实例也被销毁。