**七层和五层模型？**

七层模型相对于五层模型，多出了会话层和表示层，这使得七层模型在**理论上更加完备和通用**，但在实际应用中，五层模型更为常见，因为它更简单、更实用。

**⾮对称加密了解过吗？它主要⽤在哪些地⽅？**

非对称加密主要用于以下几个方面：

**加密通信：**在加密通信中，发送方可以使用接收方的公钥来加密数据，而只有接收方拥有**对应的私钥才能解密数据**。这样可以确保通信的机密性，即使数据在传输过程中被截获，也不会泄露其内容。

**数字签名：**非对称加密也用于数字签名。发送方**可以使用自己的私钥来对数据进行签名**，**接收方可以使用发送方的公钥来验证签名的有效性**。这样可以确保数据的完整性和来源的可信性。

**密钥交换（混合加密）：非对称加密**也用于**安全地交换对称加密算法所需的密钥**。发送方可以使用接收方的公钥来加密对称密钥，并将其发送给接收方，接收方再使用自己的私钥解密对称密钥。

**对称和非对称区别：**

* **对称加密**：使用同一个密钥进行加密和解密。速度较快，**但安全性不是很好**，**⽆法做到安全的密钥交换。适用于需要快速处理大量数据的场景**，如文件加密、数据库加密、网络数据传输加密等。
* **非对称加密**：使用一对密钥，一个公开密钥（公钥）用于加密，一个私有密钥（私钥）用于解密。公钥可以公开分享，而私钥必须保持私密。速度要慢不少，但提供了更高的安全性。常用于**安全敏感的通信**中，如**数字签名、SSL/TLS证书验证、安全电子邮件**等。**由于其速度较慢，通常用于加密少量数据或用于加密对称加密中使用的密钥**。

**混合加密（HTTPS）**

**用公钥对对称密钥加密，发送给接收方，再用自己的私钥解密得到这个对称密钥。既有安全性又有速度。**

**https加密过程，**

**说白了就是在混合加密基础上多了个CA机构，**

**客户端向服务器端发送请求报⽂，请求与服务端建⽴连接。服务器把公钥给CA机构，CA机构用自己私钥签名生成数字证书，把数字证书给客户端，客户端拿出浏览器里的CA机构的公钥验证数字证书的合法性，合法则取出服务器公钥，客户端再生成一个对称密钥，用服务器公钥对其加密，发给服务器，服务器用自己私钥解密，获得对称密钥，之后双方就用对称加密传输数据。**

HTTPS和HTTP的区别？

HTTP 是超⽂本传输协议，信息是**明⽂传输**，存在安全⻛险的问题。**HTTPS 则解决 HTTP 不安全的缺陷，**在 TCP 和 HTTP ⽹络层之间加⼊了 SSL/TLS 安全协议，使得报⽂能够**加密传输**。

HTTP 连接建⽴相对简单， TCP 三次握⼿之后便可进⾏ HTTP 的报⽂传输。⽽ HTTPS 在 **TCP 三次握⼿之后， 还需进⾏ SSL/TLS 的握⼿过程，才可进⼊加密报⽂传输**。

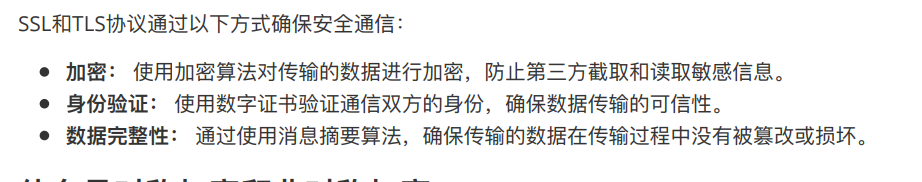
两者的默认端⼝不⼀样，HTTP **默认端⼝号是 80**，HTTPS **默认端⼝号是 443**。

HTTPS 协议需要向 CA（证书权威机构）申请数字证书，来**保证服务器的身份是可信的**。

SSL/TLS是什么

SSL：Secure Socket Layer 安全套接字

TSL：Transport Layer Security 安全传输层协议

HTTPS（HyperText Transfer Protocol Secure）是基于TLS/SSL的安全版本的HTTP协议

2. 优点 在数据传输过程中，使⽤秘钥加密，安全性更⾼ 可认证⽤户和服务器，确保数据发送到正确的⽤户和服务器

3. 缺点 握⼿阶段延时较⾼：在会话前还需进⾏SSL握⼿

部署成本⾼：**需要购买CA证书；需要加解密计算**，占⽤CPU资源，需要**服务器配置或数⽬⾼**

### 典型算法

* **对称加密算法**：AES（高级加密标准）、DES（数据加密标准）、3DES（三重数据加密算法）、RC4等。
* **非对称加密算法**：RSA、ECC（椭圆曲线密码学）、Diffie-Hellman密钥交换协议、ElGamal等。

**说下事务：**

事务的四⼤特性 ACID ACID（Atomicity、Consistency、Isolation、Durability，即原⼦性、⼀致性、隔离性、持久性）

1. 原⼦性 **事务是⼀个不可分割的⼯作单元，要么完全执⾏，要么完全不执⾏**。如果在事务执⾏的过程中发⽣了错误，系统会 撤销事务中已经执⾏的操作，将数据库恢复到事务开始前的状态。原⼦性是通过 **undo log（回滚⽇志） 来保证** 的。

2. ⼀致性 确保事务将数据库从⼀个⼀致的状态转变为另⼀个⼀致的状态。事务执⾏的结果必须满⾜数据库的完整性约束和规则，保持数据库的⼀致性。**⼀致性则是通过持久性+原⼦性+隔离性来保证的。**

3. **隔离性** 多个事务**并发执⾏**时，每个**事务都不能看到其他事务的中间状态。**每个事务都应该感觉就像它是唯⼀在数据库上运 ⾏的事务⼀样。**防⽌了多个事务之间的相互⼲扰**。**隔离性是通过 MVCC（多版本并发控制） 或锁机制来保证的。**

4. **持久性** ⼀旦事务**被提交**，其**结果将永久保存在数据库中**，即使系统发⽣故障。即使系统发⽣崩溃，事务的结果也不应该丢 失，持久性是通过 **redo log （重做⽇志）**来保证的。

**MVCC机制相比锁机制具有以下优点：**

**锁机制也能实现并发控制，但是他会导致阻塞和死锁问题**

**提高并发性能：**读操作不会阻塞写操作，写操作也不会阻塞读操作，**有效地提高数据库的并发性能。** **降低死锁风险**：由于**无需使用显式锁来进行并发控制**，MVCC可以降低死锁的风险。

**mvcc怎么实现的？**

Readview:

Read View 有四个重要的字段:

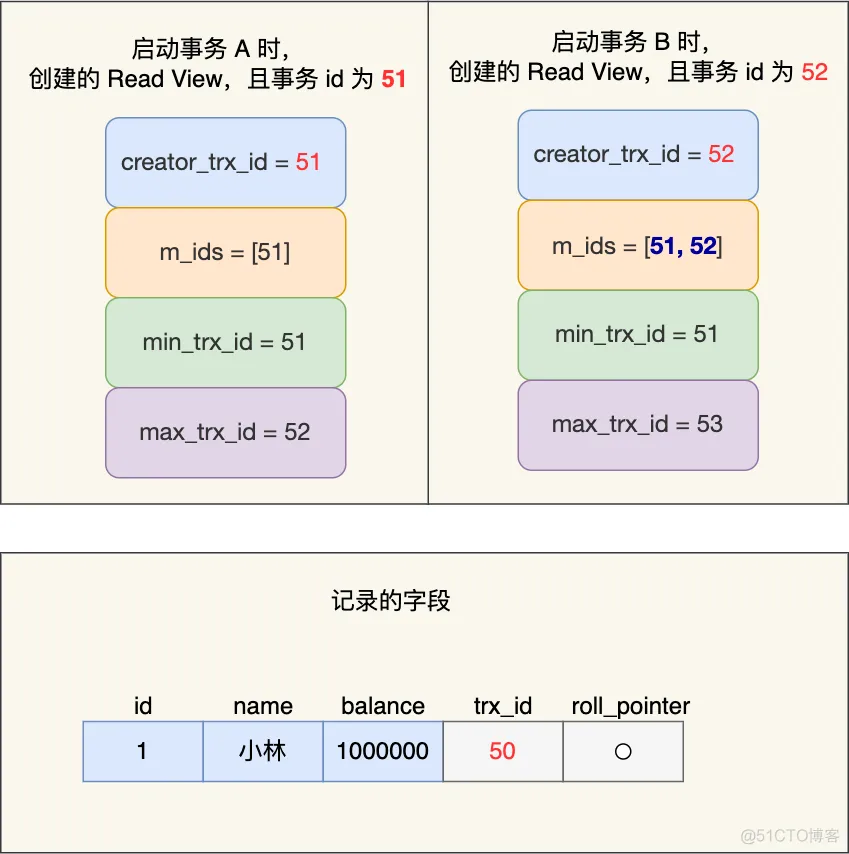
**Read View 有四个重要的字段:**

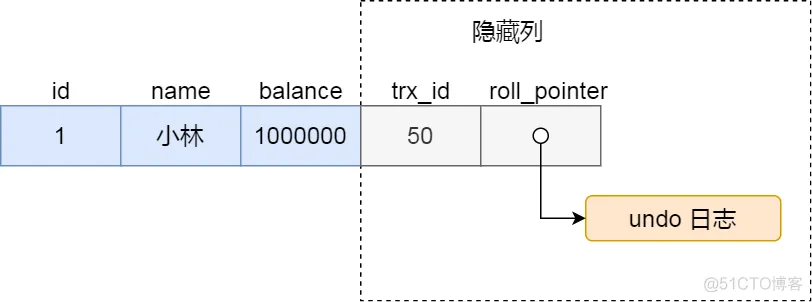
**m ids:**指的是在创健 RedVew 时，当前教据库中活跃事务id列表#，注意是一个列表，“**活跃事务”指的就是，启动了但还没提交的事务**

**min tx id**:指的是在创建 Read View 时，当前数据库中活跃事务中事务Id最小的事务，也就是 m ids 的最小值

**max tx id:** 这个并不是 m ids 的最大值，而是创建 Read View 时当前**数据库中应该给下-事务的 id 值**，也就是全局最大的事务id +1.

.

**creator trx id :** 指的是创建该 Read View 的事务的事务 id。



**trx id，**当一个事务对某条聚族索引记录进行改动时，**就会把该事务的事务 id 记录在 trx id 隐藏列里它找到修改前的记录**。

**rol pointer，**每次对某条聚族索引记录进行改动时，都会把**旧版本的记录写入到 undo 日志中**，然后这隐藏列是指针，指向每一个旧版本记录，于是就可以找到修改前记录。



我的回答

**MVCC 是通过 ReadView + undo log（回滚日志） 实现的，，ReadView是事务执行过程中使用的数据视图，用于确定事务在读取数据时所能够看到的版本。事务只能读取已提交的版本，且不能读取其他事务尚未提交的版本。**

**ReadView有好几个字段，首先是当前事物的id,然后是活跃事务id表以及这个id表的最小值，所谓的“活跃事务”指的就是，启动了但还没提交的事务。最后还有数据库下一个要启动的事物的id。**

**当你查询一个记录时，记录会有两个隐藏列，第一个隐藏列有上一个修改这个事物的id号，如果这个id号小于活跃事务列表ID的最小值的话，表示这个版本的记录是在创建 Read View前已经提交的事务生成的，所以该版本的记录对当前事务可见。**

**如果这个id在活跃事务表内或者大于等于下一个要启动事务id号，那么该版本的记录对当前事务不可见。这时要说到回滚日志了，每次记录进行改动时，都会把旧版本的记录写入到 回滚日志中，然后记录的第二个隐藏列是指针，指向回滚日志的旧版本记录，旧版本记录又更旧的记录，顺着这样一条链就可以找到对当前事务可见的的旧版本记录。这种通过版本链来控制并发事务访问同一个记录的行为就叫MVCC（多版本并发事务）。「读提交」隔离级别是在每个 select（查询） 都会⽣成⼀个新的 Read View，也意味着，事务期间多次读取同一条数据，前后两次读的数据可能会出现不一致，因为可能这期间另外一个事务修改了该记录并提交了事务**

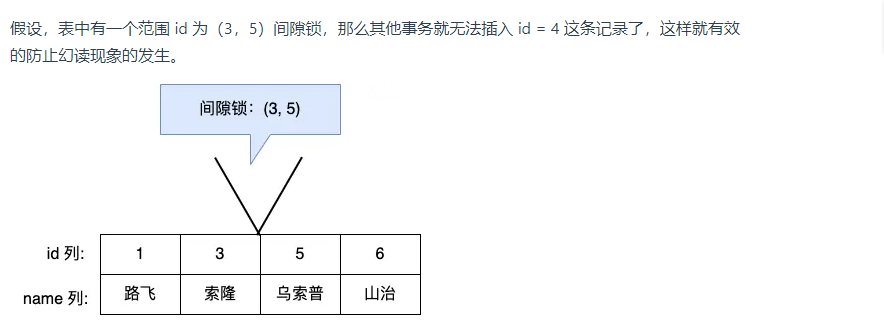
**「可重复读」隔离级别是启动事务时⽣成⼀个 Read View，然后整个事务期间都在⽤这个 Read View， 这样就保证了在事务期间读到的数据都是事务启动前的记录。**

**说下行级锁和间隙锁**

**分类**

**Record Lock**，记录锁，仅仅把⼀条记录锁上，记录锁分为排他锁和共享锁。

**Gap Lock**，间隙锁，锁定⼀个范围，不如别的记录插入修改，**但是不包含记录本身，只存在于可重复读隔离级别，⽬的是为了解决可重复读隔离级别下幻读的现象**。间隙锁之间是兼容的，两个事务可以同时持有包含共同间隙范围的间隙锁，并 不存在互斥关系。

****

**Next-Key Lock：**Record Lock + Gap Lock 的组合，锁定⼀个范围，并且锁定记录本身。next-key lock 即能 保护该记录，⼜能**阻⽌其他事务将新纪录插⼊到被保护记录前⾯的间隙中。**

**说下隔离级别：**

1. **读未提交**：最低的隔离级别。在这个级别下，**⼀个事务可以读取到另⼀个事务未提交的数据。**

2. **读提交**： 在这个级别下，**⼀个事务只能读取到已经提交的其他事务的数据。**

3. **可重复读** 在这个级别下，⼀个事务在其⽣命周期内多次执⾏相同的查询，**将始终看到相同的数据**。 也是MySQL InnoDB 引擎的默认隔离级别；

4. **可串⾏化** 提供了最⾼的隔离级别。会对记录加上读写锁，在**多个事务对这条记录进⾏读写操作时，如果发⽣了读写冲突**的时候，后访问的事务必须等前⼀个事务执⾏完成，才能继续执⾏，

事务的执⾏效果就好像它们是按顺 序执⾏的，事务之间**没有并发**。，但可能导致性能下降，因为并发性降低。

**什么是页面置换？**在**地址映射过程中**，发现所要访问的页面无效，不在内存中，则产生缺页中断。当发生缺页中断时，如果**操作系统内存中没有空闲页面**，则操作系**统必须在内存选择一个页面将其移出内存，换出到磁盘**，然后把需要访问的⻚⾯换⼊物理内存。页面置换算法就是选择哪一页换出。

**注意：**缺⻚中断在指令执⾏「期间」产⽣和处理中断信号，⽽⼀般中断在⼀条指令执⾏「完成」后检查和处 理中断信号。

缺⻚中断返回到该指令的开始重新执⾏「该指令」，⽽⼀般中断返回回到该指令的「下⼀个指令」执 ⾏。

1. **最佳⻚⾯置换算法(OPT)**

置换在「未来」最⻓时间不访问的⻚⾯,但是实际系统中⽆法实现，因为程序访问⻚⾯时是动态的 我们是⽆法预知每个⻚⾯在「下⼀次」访问前的等待时间，因此作为实际算法效率衡量标准。

2、**先进先出置换算法(FIFO)** 顾名思义，将⻚⾯以队列形式保存，先进⼊队列的⻚⾯先被置换进⼊磁盘。

3、**最近最久未使⽤的置换算法（LRU）** 根据⻚⾯未被访问时⻓⽤升序列表将⻚⾯排列（最近使⽤的⻚⾯在表头），每次将最近最久未被使⽤⻚⾯（链表尾）置换出去。 开销很大

4、**时钟⻚⾯置换算法** 把所有的⻚⾯都保存在⼀个类似钟⾯的「环形链表」中，⻚⾯包含⼀个访问位。 当发⽣缺⻚中断时，顺时针遍历⻚⾯，如果访问位为1，将其改为0，继续遍历，直到访问到访问位为0（未被访问过）⻚⾯，进⾏ 置换。

5、**最不常⽤算法** 记录每个⻚⾯访问次数，当发⽣缺⻚中断时候，**将访问次数最少的⻚⾯置换出去**，此⽅法需要对每个⻚⾯访问次数 统计，额外开销。

**说下分段和分页：**

**内存分段是将⼀个程序的内存空间分为不同的逻辑段 segments ，每个段代表程序的⼀个功能模块或数据类型，如代码段、数据段、堆栈段等。每个段都有其⾃⼰的⼤⼩和权限。**

**内存分⻚是把整个虚拟和物理内存空间分成固定⼤⼩的⻚(如4KB)。这样⼀个连续并且尺⼨固定的内存空间，我们叫⻚ Page**

**页是定长，段不是定长。**

**分⻚地址空间是一维的，分段的地址空间是二维的。**

**分段会产生外部碎片，分页消除了外部碎片，但会产生内部碎片**（在页式虚拟存储系统中, 用户作业的地址空间被划分成若干大小相等的页面, 存储空间也分成也页大小相等的物理块, 但一般情况下, 作业的大小不可能都是物理块大小的整数倍, 因此作业的最后一页中仍有部分空间被浪费掉了. 由此可知, 页式虚拟存储系统中存在内碎片.

在段式虚拟存储系统中, 作业的地址空间由若干个逻辑分段组成, 每段分配一个连续的内存区, 但各段之间不要求连续, 其内存的分配方式类似于动态分区分配.由此可知, 段式虚拟存储系统中存在外部碎片）

**分⻚**主要⽤于实现虚拟内存，从⽽获得更⼤的地址空间；

**分段**主要是为了使程序和数据可以被划分为逻辑上独⽴的地址空间并且有助于共享和保护。

结合：

程序的地址空间逻辑上划分成多个拥有独⽴地址空间的段，每个段上的地址空间划分成⼤⼩相同的⻚。 这样既拥有分段系统的共享和保护，⼜拥有分⻚系统的虚拟内存功能.

内部碎片：

内碎片是指分配给进程的内存空间中未被利用的部分。**由于一个进程占的大小并不一定与分配给他的大小相等, 因此, 分区中有一部分存储空间浪费掉**了. 由此可知, 固定式分区分配中存在内碎片.

外碎片是**指**内存中的空闲空间被分割成多个不连续的⼩块，⽽不是⼀个连续的⼤块。无法给**较大的内存请求分配连续的内存空间，这些小块就是外部碎片。**

**聚簇索引、联合索引，结合场景描述**

1. 聚簇索引将数据行物理上存储在索引的叶子节点上，而不是存储在独立的数据页中。换句话说，聚簇索引决定了数据的物理排序顺序，使得索引和数据在存储上相互紧密结合。
2. 联合索引是由多个列组成的索引，它可以根据这些列的组合进行查询和检索。联合索引可以包含两个或多个列，并且可以按照定义的顺序进行排序。

聚簇索引并不等同于主键索引，但通常情况下，数据库管理系统会将表的主键列作为聚簇索引的键。

主键索引是一种特殊的索引，它是一种唯一标识表中每行记录的索引。主键索引的目的是确保表中的每行数据都具有唯一的标识符，通常情况下，主键索引是通过定义主键约束来创建的，而主键约束要求主键列的值必须是唯一的且不能为NULL

说一下多线程。多进程区别：



**说一下HTTP缓存？**

**资源的副本存储在客户端或中间代理服务器上，以便将来的请求可以直接从缓存中获 取。**减少不必要的⽹络传输，节约带宽。更快的加载⻚⾯ **。**减少服务器负载，避免服务过载的情况出现

**分为强缓存和协商缓存**

**强缓存**：浏览器判断**请求的⽬标资源是否有效命中**强缓存，如果命中，则可以**直接从内存中读取⽬标资源**，⽆需与服务器做任何通讯。

**强缓存获取资源后还会设置一个过期时间，过期了就无法读**

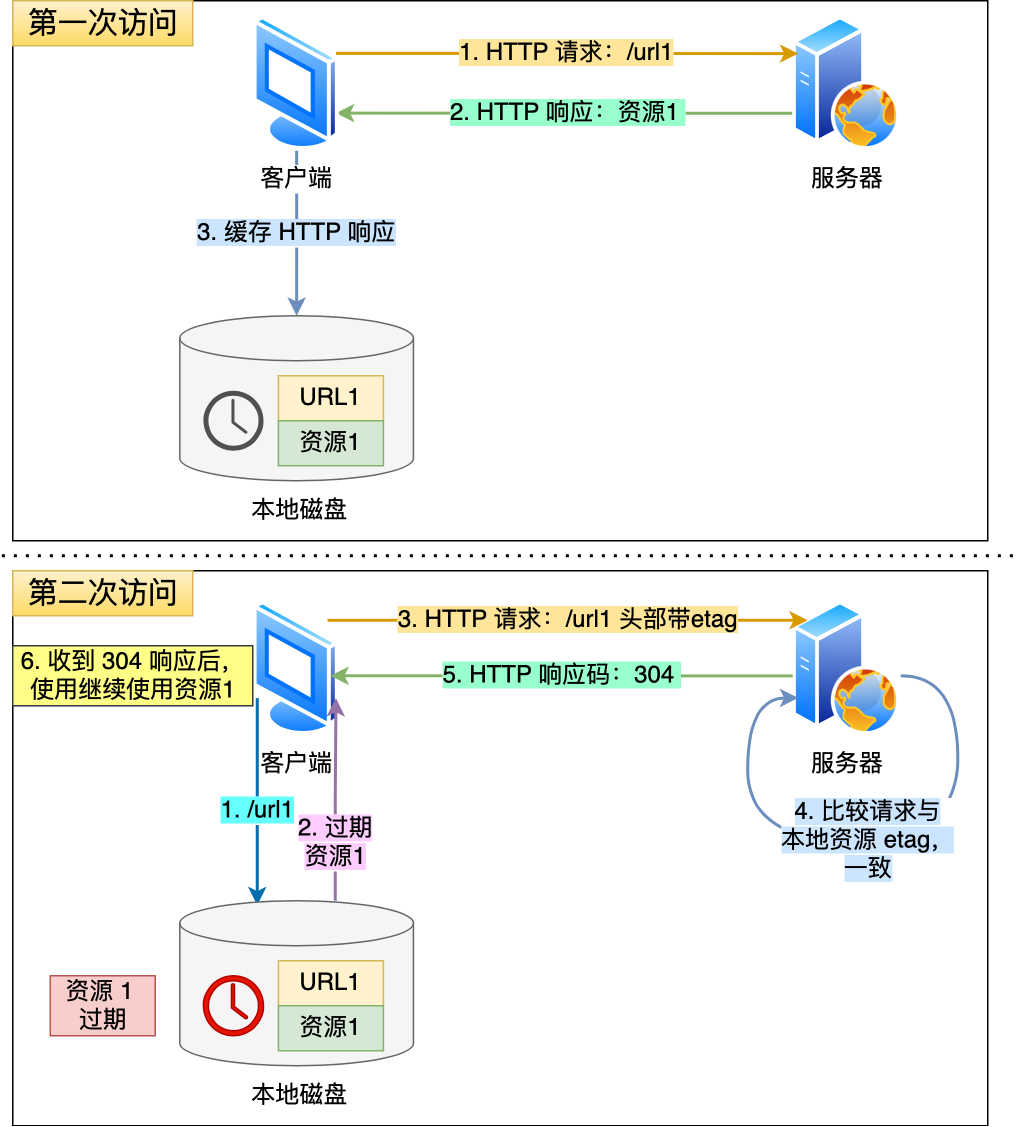
协商缓存

与强制缓存不同，**协商缓存依赖于客户端和服务器之间的交互**，在协商缓存中，**服务器在响应中提供了资源的⼀些** 标识信息，

客户端在后续请求中申请同一个资源会携带上次的标识信息发给服务端，服务器通过**这些信息来判断要申请的目标资源是否发⽣了变化**，进⽽**判断是否需要重新传输资源，**没变化就传**304码**，表示你可**以继续用本地缓存资源**。

。 比较常用的就是**e-tag和最后修改时间**。

E\_tag可以是文件内容算出来的哈希值。



### **http协议的报文结构？**

**HTTP请求报文**

首先，HTTP请求报文主要由**请求行、请求头部和请求体**三个部分构成。

请求行包含了**请求的方法，比如GET或POST**，请求的**URL路径**，以及**HTTP协议的版本。**

紧接着**请求头部，**包含多个键值对，提供**一些请求的额外信息，比如Host字段标明请求的目标主机**，。**Content-Length字段指示了请求体的长度。**User-Agent字段是用来标识发起请求的客户端

**请求体**，包含客户端提交给服务器的数据。GET请求的请求体是空的。

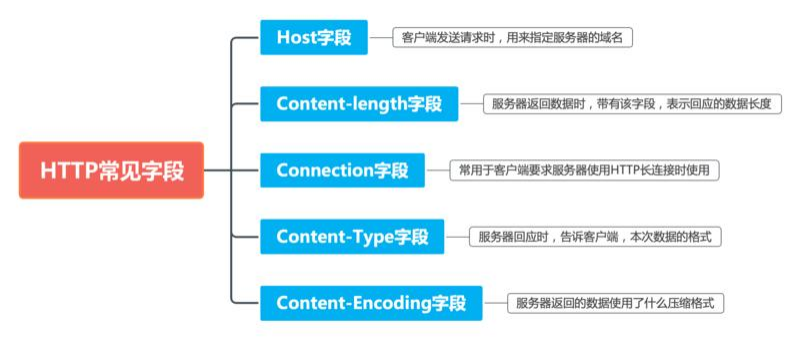
**HTTP响应报文**

结构与请求报文相似，由**状态行、响应头部和响应体**组成。

**状态行**包含**HTTP协议版本、状态码以及状态描述信息**。比如，状态码200表示请求成功，404则表示资源未找到。

**响应头部**也包含多个键值对，**提供有关响应的附加信息**，如Content-Type描述响应体的内容类型，**Content-Length指明响应体的长度**。

**响应体**则包含**服务器返回给客户端的实际数据内容**，如HTML文件、图片或其他媒体资源。





### **HTTP/1.1相对HTTP/1.0做了哪些优化改进？**

首先，HTTP/1.1引入了持久连接的概念，也就是所谓的**长连接**。在HTTP/1.0中，**每次请求都需要建立一个新的TCP连接**，**请求处理完毕后立即断开连接**。这种会导致大量的连接建立和断开操作，造成资源浪费和时间延迟。

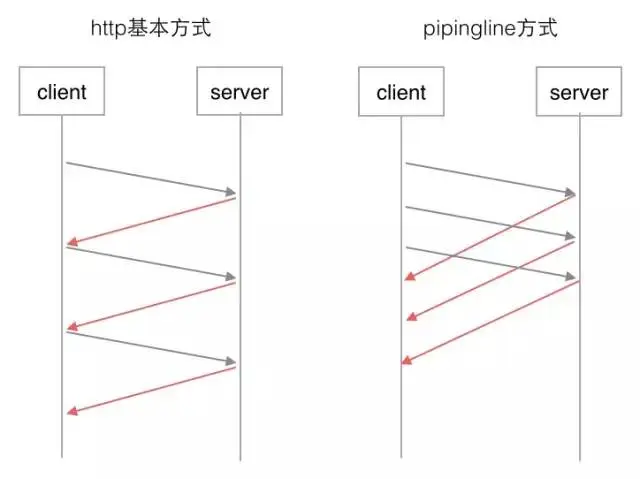
而HTTP/1.1则允许在**同一个TCP连接上连续发送多个请求**，从而**减少了连接建立和断开的开销，提高了网络资源的利用率。**

其次，HTTP/1.1支持**请求管道化**，这意味着**客户端可以同时发送多个请求**，**而不需要等待前一个请求的响应**。这样，从而减少了等待时间，提高了整体的响应速度。需要注意的是，虽然请求管道化可以并行发送请求，**但服务器仍然需要按照先后顺序处理请求发送响应。（也就是没有解决响应的队头阻塞问题，如果服务端处理某个请求消耗的时间⽐较⻓，那么只能等响应完这个请求后， 才能处理下⼀个请求）**

另外，HTTP/1.1还增加了更多的**错误**状态码，**更多的请求头和响应头**，为开发人员提供了更多的错误信息以及控制和灵活性。

最后，值得一提的是，HTTP/1.1还改进了**缓存机制**，允许服务器发送**“ETag”等响应头**，以便客户端进行**条件性请求**，这**避免了不必要的资源传输，提高了网络的利用率。**

总的来说，HTTP/1.1相对于HTTP/1.0在连接管理、请求管道化、错误处理、缓存机制等方面都做了显著的优化和改进，从而提高了网络应用的性能和效率。



**HTTP/2.0相对HTTP/1.1做了哪些优化改进？**

1. **⼆进制帧：HTTP/2 将数据分割成⼆进制帧进⾏传输，分为头信息帧（Headers Frame）和数据帧（Data）Frame）增加了数据传输的效率。**
2. **头部压缩：HTTP/2 使⽤ HPACK 压缩算法对请求和响应头部进⾏压缩，减少了传输的头部数据量，降低了延迟。**
3. **多路复用：**HTTP/2.0引入了**多路复用技术**，允许通过单一的HTTP/2连接**同时发起多个请求-响应消息**。具体来说，每个HTTP 请求⽤**独⼀⽆⼆的Stream ID 来区分**，接收端可以通过流 ID 将收到的数据帧有序组装成 HTTP 消息，不同流的帧是可以乱序发送，因此并发不同的流 ，也就是 HTTP/2 可以**并⾏交错地发送请求和响应，解决了HTTP/1.1中的队头阻塞问题，**从而更有效地使用TCP连接，提高了网络资源的利用率。

4.**HTTP/2.0还引入了服务器推送技术**，服务器可以对客户端的⼀个请求发送多个响应，即服务器可以**额外的向客户端推送资源**，**⽽⽆需客户端明确的请求**。所以极⼤的提升推送资源的速度。

综上所述，HTTP/2.0通过二进制分帧、多路复用、头部压缩和服务器推送等技术，显著提升了网络性能，降低了网络延迟，为用户提供了更好的网络体验。

### **HTTP/3.0相对HTTP/2.0做了哪些优化改进？**

首先，HTTP/3.0使用了**基于UDP协议的QUIC作为传输层协议**，而不是传统的TCP协议。这一改变**显著减少了连接建立的延迟**，因为QUIC协议在建立连接时无需进行多次握手，从而**加速了数据传输的速度。**

其次，HTTP/3.0**支持真正的多路复用**，这意味着在同一连接上可以**并行处理多个请求和响应**，而不会像HTTP/2那样在**TCP层面上存在队头阻塞的问题**。这大大提高了网络资源的利用率和传输效率。

此外，HTTP/3.0**还具备更好的流量控制能力**，可以有效防止单一流的资源占用过多带宽，从而保证网络传输的稳定性和公平性。

在**安全性方面**，HTTP/3.0默认使用**TLS 1.3加密**，为数据传输提供了**更高的安全性**。同时，QUIC协议还提供了**更好的错误恢复机制**，当数据包丢失或损坏时，能够**更快速地恢复连接并进行数据传输**。

总的来说，HTTP/3.0通过采用QUIC协议、支持真正的多路复用、提供更好的流量控制和安全性能等方面的优化和改进，进一步提升了网络传输的效率和性能。这些改进使得HTTP/3.0成为现代网络中更具优势和应用前景的协议。

**TCP粘包和拆包发生原因：**

TCP是⼀个**面向字节流协议，所谓流**，就是没有界限的⼀⻓串数据。

**会根据TCP缓冲区的实际情况进⾏数据包的划分**，所以在**业务上认为是⼀个完整的包**，**可能 会被TCP拆分成多个包进⾏发送，也有可能把多个⼩的包封装成⼀个⼤的数据包发送**，

这就是**拆包，粘包。**

**原因：**

**1.当 TCP 发送缓冲区剩余空间不足时，一个完整的包可能会被拆分为多个包进行发送，即可能发生拆包情况。**

**2.当 TCP 发送缓冲区剩余空间足够时，多个小的包也有可能被封装成一个大的包进行发送，即可能发生粘包情况。**

3.进⾏**MSS（最⼤报⽂⻓度）**⼤⼩的 TCP分段，当TCP报⽂⻓度—TCP header⻓度（TCP的数据报文长度）>MSS 的时候会发⽣拆包；

4.接收⽅法**不及时读取套接字缓冲区数据**，也会发⽣粘包。

怎么处理TCP拆包和粘包

本质问题在于无法区分包的边界：

1. 数据包的头部中增加数据包**长度字段**，比如固定数据的前四位是数据的长度，这样接收方就能区分数据包的开始和结束位置
2. 指定分隔符区别包的界限，这样当我们获取到指定的符号时，说明**包获取完整**3.关闭发送方的nagle算法

**为什么关闭：TCP通常**会使用一个叫做Nagle的算法.该算法是指,发送端即使有要发送的数据,如果很少的话,会延迟发送.如果应用层给TCP传送数据很快的话,就会把两个应用层数据包“粘”在一起,TCP最后只发一个TCP数据包给接收端，

## UDP 存在粘包和拆包的问题吗？

TCP 之所以存在拆包和粘包问题，本质就是 TCP 是面向字节流的协议，字节流协议即无边界协议；**而像 UDP 是面向报文的，当客户端连续发送多个包，并不会发生粘包现象，每一个包都是独立的，发送的时候也是以一个一个包为单位。**

那么问题来了，不会发生粘包，如果应用程序 write 一个大的包，那么到底层进行发送的时候会不会发生拆包呢？

答案是：**不会**。UDP 协议发送时，用 sendto 函数最大能发送数据的长度为：65535- IP 头(20) - UDP 头(8) ＝ 65507 字节。用 sendto 函数发送数据时，**如果发送数据长度大于该值，则函数直接返回错误，不会发生拆包**，而 TCP 流协议是会发生拆包的

**说下B+树VS的好处**

**一、树的高度**

**B+树相较于B树，其树的高度更低**。这是因为B+树的**非叶子节点不存储数据，只存储索引**，因此**可以容纳更多的子节点**，**使得整个树的结构更加扁平化**。这种设计**减少了查询时需要经过的层级数，最重要是减少了磁盘I/O的次数，从而提高了查询效率**。相比之下，B树的**每个节点都存储数据，导致节点容纳的子节点数量有限，树的高度相对较高。**

二、**插入删除效率**

**在插入和删除操作方面，B+树也表现出更高的效率。**由于B+树的**非叶子节点不存储数据**，因此在插入或删除数据时，只需要**调整索引结构，而不需要频繁地移动大量数据**。相比之下，B树在插入或删除数据时**可能需要更复杂的操作来保持树的平衡**。

**三、范围查询效率**

**B+树在范围查询方面具有显著优势**。由于B+树的**叶子节点包含所有数据，叶子节点之间通过指针相连，形成了一个有序链表结构**。这种结构使得范**围查询变得非常简单和高效，**因为我们可以直接通过叶子节点的指针顺序访问范围内的数据。而在**B树**中，**范围查询需要中序遍历遍历所有节点，效率就比较低**。

**数据库中要优化查询性能，最常见的方式是什么？**

数据库中优化查询性能的最常见方式包括：**创建索引**、**合理设计数据库表结构**、**优化 SQL 查询语句**、**分表分库**、**使用缓存**等。

**数据库索引是干什么用的？**

数据库**索引**是一种**数据结构，用于快速定位和访问表中的记录，大大提⾼数据查询的效率，就像书的⽬录⼀样。**

通过在表的某个字段上创建索引，可以加**快查询速度，提高数据库的性能。**

MySQL 在查询⽅⾯主要就是两种⽅式：

1. **全表扫描（⼀个⼀个挨个找）**

2. **根据索引检索**

**索引问题大集合：**

**一、索引与主键的区别**

**1. 主键是为了标识数据库记录唯⼀性,不允许记录重复,且键值不能为空,主键也是⼀个特殊索引**

**2. 数据表中只允许有⼀个主键,但是可以有多个索引**

**3. 使⽤主键的数据库会⾃动创建主索引,也可以在⾮主键上创建索引,⽅便查询效率**

**4. 索引可以提⾼查询速度,它就相当于字典的⽬录,可以通过它很快查询到想要的结果,⽽不需要进⾏全表扫描**

**5. 主键索引外索引的值可以为空. 主键也可以由多个字段组成,组成复合主键,同时主键肯定也是唯⼀索引**

**6. 唯⼀索引则表示该索引值唯⼀,可以由⼀个或⼏个字段组成,⼀个表可以有多个唯⼀索引，**

**2、索引的优点与缺点**

**索引的优点**：

1. **提高查询速度**：索引可以极大地加快数据访问的速度。这也是主要原因
2. **排序数据记录**：索引可以在数据库中为数据记录进行排序，从而减少了排序操作（ORDER BY）的需要，同时也加快了排序查询的响应时间。
3. **唯一性约束**：唯一索引可以保证每一行数据的唯一性。这经常用于主键索引以确保主键不重复。
4. **更有效的数据访问计划**：对于复杂的查询语句，包含多个连接（JOINs）和条件子句（WHERE）的查询，数据库优化器能够利用索引来创建一个更有效的数据访问策略。
5. **支持索引的SQL语句**：通过使用索引，可以快速完成许多SQL操作，如COUNT, MIN, MAX等。

**缺点：**

**1. 当对表中的数据进⾏增加、删除和修改的时候，索引也需要更新，时间开销会更大。**

**2. 索引也需要占储存空间，比起无索引情况下需要更多的存储空间。**

**3.维护索引也要耗费额外的时间**

**聚簇索引和⾮聚簇索引的区别？**

**聚簇索引：**

**1. 以主键创建的索引**

**2. 在叶⼦节点存储的是表中的数据**

**3. 不是⼀种索引类型,⽽是⼀种存储数据的⽅式.Innodb的聚簇索引是在同⼀个数据结构中保存了索引和数据**

**4. 聚簇索引记录的排序顺序和索引的排序顺序⼀致，所以查询效率快。**

**【因为只要找到第⼀个索引值记录，其余的连续性的记录在物理表中也会连续存放，⼀起就可以查询到】**

**缺点：**

**新增⽐较慢**

**【因为为了保证表中记录的物理顺序和索引顺序⼀致，在记录插⼊的时候，会对数据⻚᯿新排序】**

**⾮聚簇索引：**

**1. 以⾮主键创建的索引（⼆级索引）**

**2. 在叶⼦节点存储的是主键和索引列**

**3. 索引的逻辑顺序与磁盘上⾏的物理存储顺序不同，⾮聚集索引在叶⼦节点存储的是主键和索引列**

**索引什么时候是否会失效，最左匹配原则是什么？**

**最左匹配原则：**

**最左优先，以最左边的为起点任何连续的索引都能匹配上，如不连续，则匹配不上；**

**索引失效在以下情况：**

**1. 不满⾜最左【前缀】匹配规则**

**2. 在索引列上做任何操作如计算、函数、（⼿动或⾃动）类型转换等操作，会导致索引失效⽽进⾏全表扫描**

**3. 使⽤不等于（!= 、<>）**

**4. like 中以通配符开头(’%abc’)**

**5. 字符串不加单引号索引失效**

**6. or 连接索引失效。**

1. **索引的类型**

**按「数据结构」分类：B+tree索引、Hash索引、Full-text索引。**

**按「物理存储」分类：聚簇索引（主键索引）、⼆级索引（辅助索引）。**

**按「字段特性」分类：主键索引、唯⼀索引、普通索引、前缀索引。**

**按「字段个数」分类：单列索引、联合索引。**

**（1）从数据结构来看**

B+树索引：常见的多叉树结构，具有平衡性和有序性，适用范围广，适合范围查询和排序操作。

哈希索引：基于哈希表实现。适用于等值查询，查询速度快，但不适合范围查询和排序操作。

全文索引：**用于文本类型列**，支持全文搜索和复杂的文本匹配。

1. **从物理存储来看可以分为主键索引和二级索引：**

**主键索引和二级索引：**

**• 主键索引的 B+Tree 的叶子节点存放的是实际数据，所有完整的用户记录都存放在主键索引的 B+Tree 的叶子节点里；**

**• 二级索引的 B+Tree 的叶子节点存放的是主键值，而不是实际数据。**

**第二张图叫回表，，也就是说要查两个 B+Tree 才能查到数据**

**当查询的数据是能在⼆级索引的 B+Tree 的叶⼦节点⾥查询到，这时就不⽤再查主键索引查。这种在⼆级索引 的 B+Tree 就能查询到结果的过程就叫作「覆盖索引」 由于覆盖索引可以减少树的搜索次数，显著提升查询性能，所以使⽤覆盖索引是⼀个常⽤的性能优化⼿段。**

1. **按照字段特性分类**

**主键索引：建⽴在主键字段上的索引，⼀张表最多只能有⼀个主键索引。主键列的值必须唯一，不能重复。不允许有空值。**

**唯⼀索引：建⽴在 UNIQUE 字段上的索引，⼀张表可以有多个唯⼀索引，索引列的值必须唯⼀，不能重复，但是允许有空值**

**普通索引：建⽴在普通字段上的索引，列的值可以重复，可以为空**

**前缀索引：对字符类型字段的前⼏个字符建⽴的索引，⽽不是在整个字段上建⽴的索引，可以减⼩索引的⼤⼩，适⽤于较⻓列值的情况。**

**4. 按照字段个数分类**

**单列索引：建⽴在单列上的索引称为单列索引，⽐如主键索引；**

**联合索引：由多个列组合⽽成的索引。适⽤于多列的查询条件**

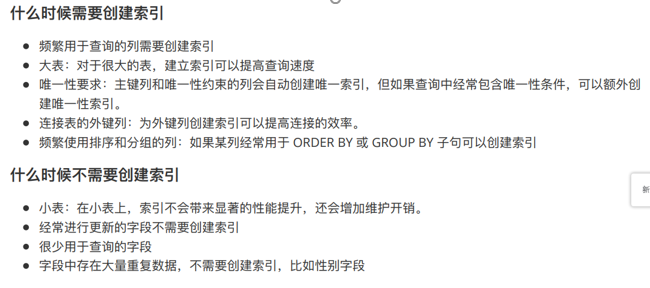
**（3）优化手段**

**前缀索引：**

在对⼀个⽐较⻓的字符串进⾏索引时,可**以仅索引开始的⼀部分字符,这样可以⼤⼤的节约索引空间,**从⽽提⾼索引效率.但是这样也会降低索引的选择性

**覆盖索引**

**主键索引最好是自增的，**不需要重新移动数据，不是自增的话会随机插入

****

**怎么查看⼀个SQL语句是否使⽤了索引进⾏检索**

**在 SQL 语句前，添加 explian 关键字**

**当 type=ALL 时，表示使⽤ 全表查询（未使⽤索引）**

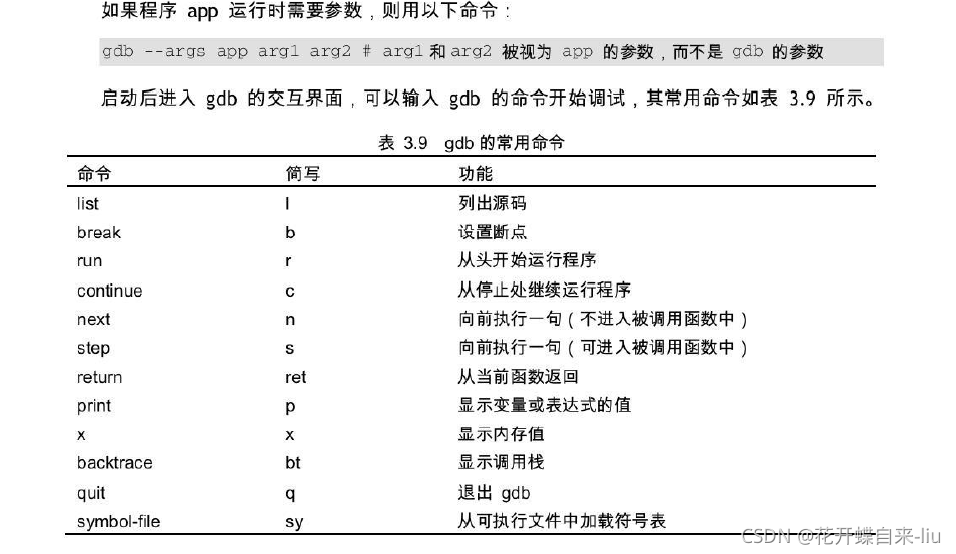
**当 type=RES 时，表示使⽤索引**



**GDB怎么调试：**

1. 编译时包含调试信息：在编译程序时，确保包含了调试信息。通常，使用 -**g 标志来告诉编译器生成调试信息**。例如，在使用 GCC 编译 C 程序时，可以使用 gcc -g <source\_file.c> -o <executable>。**gcc -g test.c -o test**
2. 启动 gdb：在终端中运行 gdb 命令，并将可执行文件作为参数传递给 gdb。例如：gdb <executable\_file>。
3. 设置断点：确定你想要在程序中设置断点的位置，以便在那里停止执行并检查程序状态。可以使用 break 或 **b** 命令来设置断点。例如：break main。
4. 运行程序：在 gdb 中，使用 run 或 r 命令来运行程序。如果程序崩溃或达到断点，它将停止执行并返回到 gdb 提示符。
5. 查看程序状态：一旦程序停止，你可以使用各种 gdb 命令来查看程序状态。例如，使用 **print 命令查看变量的值**，使用 backtrace 或 **bt 命令查看调用堆栈**，使用 info locals 命令查看当前函数的局部变量等等。
6. 单步执行：使用 step 或 s 命令逐行执行程序。这可以帮助你跟踪程序的执行路径，并查看每一步的变量状态。
7. 继续执行：一旦达到断点或停止执行，你可以使用 continue 或 **c 命令继续执行程序，**直到下一个断点或程序结束。
8. 分析问题：通过观察程序状态和执行路径，尝试确定问题的根本原因。
9. 修复错误：一旦找到问题所在，可以修改代码来修复错误。在进行修改之前，确保理解问题并确保修复不会引入新的问题。
10. 验证修复：修复错误后，确保进行适当的测试和验证，以确保问题已经解决。

最关键的命令：n（next） b （break）c （continue）s（step）



**与step命令不同，next命令将不会进入函数调用，如果当前行是一个函数调用，next会将整个函数作为一个单独的步骤执行，并在函数返回后停在调用该函数的下一行。**

**GDB运行报错如和通过core文件找错误？**

1. 确保生成了 core 文件：在程序崩溃时，默认情况下，操作系统可能会生成一个名为 core 的文件。如果没有生成 core 文件，你可能需要配置系统以生成它。在 Linux 系统上，可以使用 ulimit 命令来设置核心文件大小限制，以便生成 core 文件。例如，ulimit -c unlimited 可以设置为无限制。
2. 分析 core 文件：使用 gdb 来分析 core 文件。运行 gdb 命令，然后将程序的可执行文件和 core 文件作为参数传递给 gdb。例如：gdb <executable\_file> core。
3. 查看堆栈信息：在 gdb 中，可以使用 **bt 命令来打印堆栈跟踪信息。这将显示程序在崩溃时的调用堆栈**，以及导致崩溃的函数和代码行。
4. 查看变量状态：通过 gdb，你可以查看在程序崩溃时各个变量的状态。可以使用 print 命令来查看特定变量的值，或者使用 info locals 命令来查看当前函数的局部变量。或者frame。结合崩溃点的调用栈和局部变量，来定位可能的错误原因。
5. 分析内存状态：如果程序崩溃与内存相关，你可以使用 gdb 来查看内存状态。例如，使用 info malloc 命令可以显示动态内存分配的状态。
6. 重现错误：如果可能的话，在调试过程中尽量重现错误。有时候，问题可能是由于特定的输入或特定的执行路径导致的。
7. 修复错误：一旦找到问题所在，就可以开始修复错误。这可能涉及更改代码、修复内存错误或调整程序的逻辑。
8. 验证修复：修复错误后，确保进行适当的测试和验证，以确保问题已经解决.

**查看进程相关的命令：**

**ps -aux** **静态查看当前进程信息**

ps -ef

**top** 实时查看当前进程信息，**获取到的进程信息是动态的**

**pgrep** <进程名>: **查找包含指定名称的进程的** PID。

**kill <PID>:** 向指定PID的进程发送默认的 TERM 信号，通常用于优雅地终止进程。

• kill -9 pid 强制杀死进程

• killall name 杀死进程名为name的进程

psgrep: 结合 ps 和 grep 命令来查找和筛选进程

**ps aux | grep <进程名>: 查找包含指定名称的进程**。

在Linux中，有许多用于网络管理和查询的命令。以下是一些常用的命令：

**ifconfig** / ip addr: **查看和配置网络接口的IP地址、子网掩码、广播地址等信息**。

ifconfig: 查看当前网络接口的信息（在较新的发行版中可能已经废弃）。

ip addr show: 显示所有网络接口的详细信息。

route / ip route: 查看和配置路由表。

route -n: 显示路由表，不进行域名解析。

ip route show: 显示路由表的详细信息。

**netstat:** 显示网络连接、路由表、接口统计等信息。

netstat -tuln: 显示所有正在监听的TCP端口。

netstat -r: 显示路由表。

ss: 类似于 netstat，用于显示套接字统计信息。

ss -tuln: 显示所有正在监听的TCP端口。

**ping:** **用于测试与目标主机的连通性**。

ping <目标主机>: 向目标主机发送ICMP回显请求。

traceroute / tracepath: 跟踪数据包从本机到目标主机的路径。

traceroute <目标主机>: 显示数据包经过的路由器。

tracepath <目标主机>: 类似于 traceroute，但更简洁。

nslookup / dig: 用于查询DNS信息。

nslookup <域名>: 查询域名的IP地址。

dig <域名>: 更为强大和灵活的DNS查询工具。

curl / wget: 用于下载文件或访问网络资源。

curl <URL>: 下载指定URL的内容。

wget <URL>: 下载指定URL的文件。

**tcpdump: 抓包工具，用于捕获和分析网络数据包。**

tcpdump -i <接口>: 监听指定网络接口上的数据包。

Git命令：

git init: **在当前目录初始化一个新的 Git 仓库。**

**git clone** <仓库地址>: **克隆远程仓库到本地。**

git add <文件名>: **将文件添加到暂存区。**

git commit -m "提交信**息": 将暂存区的文件提交到本地仓库。**

git status: 查看工作区、暂存区的状态。

git diff: 查看工作区与暂存区的差异。

git diff --cached: 查看暂存区与最新提交的差异。

git diff <commit1> <commit2>: 查看两个提交之间的差异。

git log: 查看提交日志。

git branch: 查看本地分支列表。

git branch <分支名>: 创建新分支。

git checkout <分支名>: 切换到指定分支。

git merge <分支名>: 将指定分支合并到当前分支。

git remote -v: 查看远程仓库地址。

git push: 将本地提交推送到远程仓库。

git pull: 拉取远程仓库的更新到本地。

**git fetch: 从远程仓库拉取最新的提交，但不合并。**

git reset --hard <commit>: 将当前分支重置到指定提交。

git revert <commit>: 创建一个新的提交来撤销指定提交的更改。

git tag: 查看标签列表。

git tag <标签名>: 创建新标签。

git show <标签名>: 查看标签的详细信息。

git stash: 将当前工作目录的变更暂存起来。

git stash apply: 恢复最近一次暂存的变更。

**说下封装**

**封装的作用：将数据和方法封装成一个抽象的类，隐藏内部细节，提高安全性和可维护性。**

**说下三种继承**

**继承：**

**公有继承（public）**

**基类的 public 和 protected 成员的访问属性在派生类中保持不变，但基类的 private 成员不可直接访问。**

**派生类中的成员函数可以直接访问基类中的 public 和 protected 成员，但不能直接访问基类的 private 成员。**

**通过派生类的对象只能直接访问基类的 public 成员。**

**保护继承（protected）**

**基类的 public 和 protected 成员都以 protected 身份出现在派生类中，但基类的 private 成员不可直接访问。**

**派生类中的成员函数可以直接访问基类中的 public 和 protected 成员，但不能直接访问基类的 private 成员。**

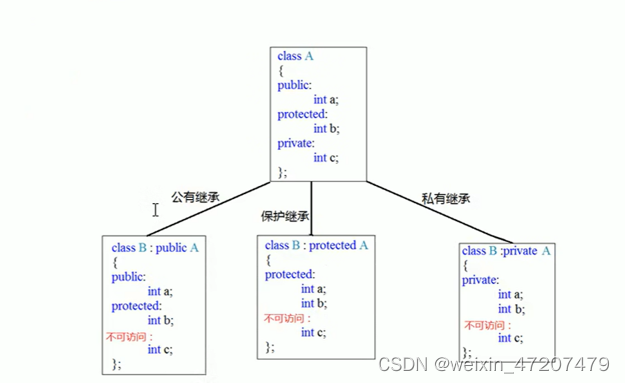
**通过派生类的对象不能直接访问基类中的任何成员**

**私有继承（private**

**基类的 public 和 protected 成员都以 private 身份出现在派生类中，但基类的 private 成员不可直接访问。**

**派生类中的成员函数可以直接访问基类中的 public 和 protected 成员，但不能直接访问基类的 private 成员。**

**通过派生类的对象不能直接访问基类中的任何成员。**



**作用：子类可以继承父类的属性和方法，从而不必重复编写相同的代码，同时可以对这些方法功能进行扩展。**

**多态实现的条件？**

1. 必须是**公有继承** ②必须是通过基类的**指针或引用** 指向派生类对象 **访问派生类方法** ③基类的方法必须是**虚函数**，且子类完成了虚函数的重写

不同的对象对同一个消息做出不同的响应。

**本质是同一事物表现出不同事物的能力**

**说下你对多态的理解**

**编译时多态性（重载）**

**同⼀作⽤域内**，允许**多个同名函数，**但具有**不同的参数类型和数量，编译器根据函数的参数类型和数量来选择调用合适的函数**

**运行时多态**

面向对象编程中**靠虚函数和继承实现了多态的机制**

通过在基类中声明虚函数，并在派生类中重写（override）这些虚函数。当通过**基类类型的指针或引用**指向派⽣类对象来调用其虚函数时，系统会根据**实际创建的对象类型动态决定使用哪个版本的函数**。这种机制称为动态绑定或**运行时多态**。

**说下虚函数实现多态背后的原理**

**虚函数的实现通常依赖**于⼀个被称为虚函数表（虚表）的**数据结构**。每个类（包括抽象类）都有⼀个虚表，其中包含了该类的**虚函数的地址**。每个对象都包含⼀个指向**所属类的虚表的指针，这个指针被称为虚指针（vptr）。**

当调⽤⼀个虚函数时，**编译器**会使⽤**对象的虚指针查找虚表**，并通**过虚表中的函数地址来执⾏相应的虚函数**。这就是为什么在能够实现运行时多态的原因。

**什么是纯虚函数？**

**纯虚函数是在基类中声明但没有实现的虚函数**。它的声明形式为 **virtual 返回类型 函数名() = 0;，它要求派生类必须实现这个函数**。**纯虚函数使得基类变成了抽象类，不能直接实例化对象，只能作为接口被派生类继承和实现。**

**说一下C++11新特性**

nullptr是C++11引入的空指针常量，用于表示空指针。它是一个特殊的关键字，具有与0或NULL相同的作用，**但更加明确和类型安全。**

类型安全：nullptr会自动转换类型指针（隐式匹配指针类型，比NULL（void\*,需要强制转换）好用，初始化空指针会使程序更加健壮。

C++11引⼊nullptr关键字来区分空指针和0。nullptr 的类型为 nullptr\_t，**能够转换为任何指针或成员指针的类型**， 也可以进⾏相等或不等的⽐较

举例：

Int\*ptr=NULL

Int\*ptr1=nullptr

int \*ptr3=(int\*)ptr

int \*ptr4=ptr1

auto关键字： **允许编译器⾃动推断变量的类型**，减少**类型声明的冗余**。

不允许使用auto的四个场景：

1. 不能作为函数参数function(auto a)
2. 不能用于类的非静态成员变量的初始化
3. 不能用auto定义数组
4. 无法使用auto推导出模板参数。

**匿名函数：**

也叫lambda函数

匿名函数**是指没有显式名称的函数。可以在调用的地方直接定义和使用**，而无需为其命名。通常用于需要临时定义、短小且一次性使用的函数逻辑，如**回调函数、排序函数**等。

**Int N=100,M=10**

**auto g=[N,&M](int i){**

**M=20;**

**Return N\*I;**

**}**

**I是输入参数，auto自动推断返回类型，[ ]里面是捕获变量，空就是不捕获任何变量,加引用就是可以修改外围变量值。**

**// 使用 Lambda 表达式定义匿名函数并传递给 sort 函数**

**std::sort(nums.begin(), nums.end(), [](int a, int b) {**

**return a < b;**

**});**

**Linux 中进程内存布局知道吗？**

**地址空间从低往高**

**代码段（Text Segment）：**

存放程序执行代码的一块内存区域。只读，代码段的头部还会包含一些只读的常数变量。

**数据段（Data Segment）：**

存放程序中**已初始化的全局变量和静态变量**的一块内存区域。

**BSS 段：**

存放程序中**未初始化的全局变量和静态变量**的一块内存区域

**堆（Heap）：**

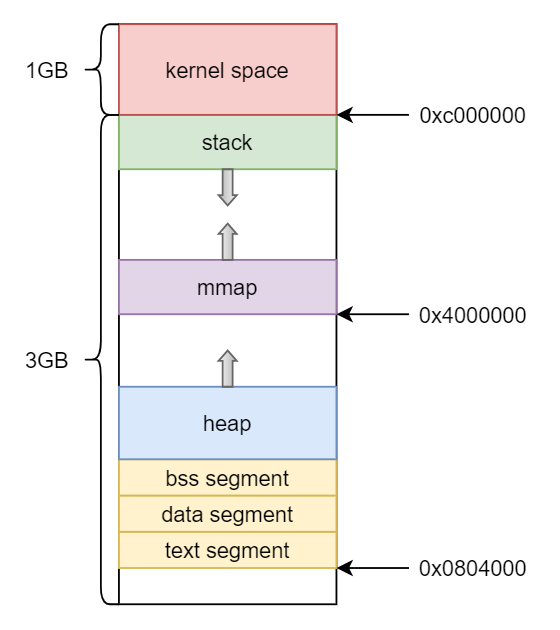
**堆是动态分配内存的区域**，用于存储程序运行时动态分配的内存块。

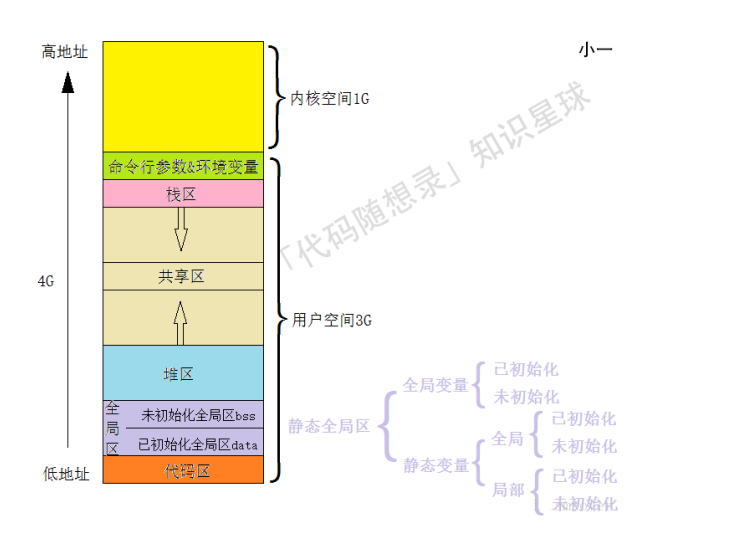
**栈（Stack）：**

**栈用于存储函数调用的参数、局部变量和函数调用的返回地址等信息。每当一个新的函数被调用时，其局部变量和其他相关信息会被压入栈中**；**当函数返回时，这些信息会被弹出**。栈是一种后进先出（LIFO）的数据结构。

**内核空间：**

内核空间包含了操作系统的核心代码和数据结构，它通常在进程的地址空间的上方。用户空间的进程无法直接访问内核空间，必须通过**系统调用来请求内核执行特定的操作。**



****

各个分区存放哪些数据？

**临时变量是存在哪里的？**

临时变量通常存放在函数的栈区中。当函数被调用时，局部变量和函数参数都会被分配到栈上，并在函数执行完毕后自动释放。

**在堆上分配内存，在 C++ 中怎么去申请？**

在 C++ 中，**可以使用 new 操作符在堆上动态分配内存**，**返回指向分配内存的指针**。例如：

int\* p = new int; // 分配一个整型变量的内存空间

**假如现在一个数据量比较大，比如达到兆级，一般放在哪里？**

一般来说，**大数据量的数据可以放在磁盘上，可以使用文件系统进行管理。如果数据需要频繁访问和修改，可以考虑放在内存中**，或者使用数据库等持久化存储方式。

**进程间的通信方式有哪些？**

进程间的通信方式包括管道、消息队列、共享内存、信号量，信号，套接字等。

每个进程的**⽤户地址空间都是独⽴的，⼀般⽽⾔是不能互相访问的**，但内核空间是每个进程都共享的，所以进程之间要通信必须通过内核。

内核提供了以下进程间通信的机制：

1. 管道：半双⼯的通信⽅式，数据只能单向流动，双向通信则要两个管道。

分为两类：

匿名管道，它的通信范围是存在⽗⼦关系的进程

命名管道，它可以在不相关的进程间也能相互通信

**不管是匿名管道还是命名管道，进程写⼊的数据都是缓存在内核中，另⼀个进程读取数据时候⾃然也是从 内核中获取，并且遵守先进先出的原则。另外，管道传输的数据是⽆格式的流且缓冲区⼤⼩受限。**

1. 消息队列

消息队列是保存在内核中的消息链表，分成⼀个⼀个独⽴消 息体（数据块），消息体是⽤户⾃定义的数据类型。克服了管道**只能承载⽆格式字节流以及缓冲区⼤⼩受限**等缺点。

A 进程如果要用消息队列给 B 进程发送消息，A 进程只需要把数据放在对应的消息队列后就可以正常返回了，B 进程需要的时候再去读取数据就可以了。

但是：消息队列不适合⽐较⼤数据的传输，因为在内核中每个消息体都有⼀个最⼤⻓度的限制。消息队列通信过程中，也存在⽤户态与内核态缓冲区之间的数据拷⻉开销

1. 共享内存

共享内存是最快的进程通信⽅式。共享内存的机制，就是每个进程拿出⼀块虚拟地址空间来映射到相同的物理内存中。这样一个进程写⼊的东⻄， 另外⼀个进程⻢上就能看到了，大大提高了进程间通信的速度，而且他**直接对内存操作**，还不涉及用户态和内核态缓冲区之间的拷贝开销。

4.信号量：是⼀个计数器，可以⽤作常作为⼀种锁机制（互斥手段），控制多个进程对共享资源的访问，防⽌某进程正在访问 共享资源时，其他进程也访问该资源。也可以作为进程间以及同⼀进程内不同线程之间的同步⼿段。

信号初始化为 1 ，就代表着是互斥信号量，

信号初始化为 0 ，就代表着是同步信号量

5.信号：是进程间通信机制中唯⼀的异步通信机制。⽤于通知进程某个事件已经发⽣，从⽽迫使进程执⾏信号处理程序，

6. Socket 通信是⼀种**⽹络编程中常⻅的通信⽅式**，但它也可以在同⼀台机器上的不同进程之间进⾏通信。**使不同机器的进程也能通信**。

**多个线程间的有哪些通信方式？**

**两个进程间的线程通信，相当于进程间通信：**  
信号量, socket网络连接, 共享内存 ,管道,共享文件。

如果是同进程的多线程，他们的通信主要是为了同步

线程同步机制是指在多线程编程中，为了保证线程之间的互不⼲扰，⽽采⽤的⼀种机制。1. 互斥锁：互斥锁是最常⻅的线程同步机制之一。它可以确保在任意时刻只有一个线程可以进入被保护的临界区（共享资源），其他线程必须等待互斥锁被释放后才能访问

2. 条件变量：条件变量⽤于线程间通信，允许⼀个线程等待某个条件满⾜，⽽其他线程可以通知等待线程条件满足了，来唤醒他。通常与互斥锁⼀起使⽤。等待线程在等待条件变量时会释放互斥锁，当条件满足时唤醒线程会重新获取互斥锁。

3. 读写锁： 读写锁允许多个线程同时读取共享资源，但只允许⼀个线程写⼊资源，并且写的时候，别的也不能读。

4. 信号量：信号量是一种计数器，用于控制对共享资源的访问

**说下MMU：**

内存管理单元，MMU可以通过页面表（Page Table）实现虚拟内存管理。页面表是一种数据结构，记录了**每个虚拟页面和其对应的物理页面之间的映射关系。**

当CPU发出**一个虚拟地址时，MMU会通过页面表查找并将其转换为对应的物理地**址。

此外，MMU还可以通过页面表实现内存保护和共享等功能，从而提高系统的安全性和效率。

MMU还可以通过快表（一种高速缓存），加速地址转换过程，避免了每次都要访问内存中的页表来获取映射关系。

**Linux 下查看 Linux 服务器的性能指标通过什么查看？比如内存、CPU 使用情况。**

1. CPU信息：`lscpu` 命令可以查看CPU的详细信息，例如核心数、频率等。

2. 内存信息：`free` 命令可以查看内存使用情况，包括总内存、已用内存、可用内存等。

3. 磁盘空间：`df` 命令用于显示磁盘分区的空间使用情况。

4. 进程信息：`ps` 命令可以列出当前运行的进程，`top` 命令可以实时显示进程的系统资源使用情况。

5. 网络连接：`netstat` 或 `ss` 命令可以列出当前的网络连接情况。

6. 日志文件：`/var/log/` 目录下存储了系统的各种日志文件，您可以查看这些日志文件以了解系统运行情况和潜在问题。

7. 系统负载：`uptime` 命令可以显示系统的负载情况，即CPU平均负载。

8. 网络性能： `ping` 命令可以测试与其他主机的网络连接，`traceroute` 命令可以跟踪数据包的路径。

9. 硬件信息： `lshw` 或 `lspci` 命令可以获取硬件设备的信息，例如CPU、内存、显卡等。

10. 系统信息：`uname` 命令用于查看系统的基本信息，例如内核版本、操作系统类型等。

11. 系统服务：`systemctl` 命令用于管理系统服务，可以查看服务的状态、启动、停止等。

12. 系统安全：`ufw` 命令用于管理防火墙规则，`fail2ban` 可以保护系统免受暴力破解攻击。

别人作为客户端给 Linux 服务器上的程序发消息，但是你这边没收到，Linux 下面用什么方式命令可以快速知道这个包有没有到你这边？

**可以使用 tcpdump 命令来抓取网络数据包，查看是否收到了客户端发送的消息。**例如：

php

Copy code

sudo tcpdump -i <网络接口> host <客户端IP> and port <端口号>

其中，<网络接口> 是你的网络接口名，<客户端IP> 是客户端的 IP 地址，<端口号> 是你监听的端口号。

**调试程序有 BUG 是通过什么方式调试的？**

调试程序可以使用一些调试工具，比如 gdb（GNU 调试器。**通过在代码中插入断点、打印调试信息，以及使用调试器的单步执行、查看变量值等功能，来定位和解决 BUG。**

5 层网络协议是哪 5 层？

5 层网络协议是指 OSI 模型中的 5 层网络协议，包括物理层、数据链路层、网络层、传输层和应用层。

TCP 属于哪一层？

TCP 属于 OSI 模型中的传输层。

数据链路层通过什么来寻址？

数据链路层通过 **MAC 地址来寻址，MAC 地址是网卡的硬件地址**，用于唯一标识网络设备。

网络层通过什么来寻址？

网络层通过 IP 地址来寻址，**IP 地址是唯一标识网络中设备的逻辑地址**。

**不在同一局域网的两台机器，分别只有 IP，在客户端发出消息到服务端收到地址，经过哪些协议？**

在不同局域网的机器通信时，消息经过的协议包括 ARP（地址解析协议）、IP（Internet 协议）、ICMP（Internet 控制消息协议）、TCP（传输控制协议）或 UDP（用户数据报协议）等。

**包到达路由器下的局域网了，拿到的只是一个 IP，在局域网下面怎么把 IP 换成 MAC 的？**

在局域网下，可以使用 **ARP 协议（地址解析协议）将 IP 地址映射到 MAC 地址。**

**ARP 协议的流程是什么？**

ARP 协议的流程包括：1. 主机 A 检查 **ARP 高速缓存（ARP Cache）是否有目标 IP 对应的 MAC 地址**；

2. 如果缓存中没有，主机 A 以广播的形式向局域网发送 ARP 请求；

3. **目标主机 收到 ARP 请求后，将自己的 MAC 地址作为响应发送给主机 A**；

4. 主机 A 收到响应后，拿到了MAC地址，并将目标 **IP 地址和 MAC 地址的映射关系存储到 ARP 高速缓存中。**

**Socket 编程，TCP 和 UDP 在 API 层面主要有哪些区别？**

**TCP 是可靠传输，我们在编程的时候相比 UDP 是不是要绑定端口这些？**

是的，TCP 是面向连接的协议，需要在通信之前建立连接。在编程时，需要调用 bind() 函数绑定端口，然后调用 listen() 函数开始监听，最后调用 accept() 函数接受客户端连接。

Udp客户端直接socket后，bind无所谓，直接send recv

服务端 必须要先bind

**网络通信是属于大端还是小端？**

网络通信一般是**大端序（Big Endian）**，即**高位字节存放在低地址处，低位字节存放在高地址处。**这是因为大多数网络协议都采用了网络字节序（Big Endian），例如 TCP/IP 协议。

**HTTP 里面的状态码知道吗？200 代表什么？**

HTTP 返回码是**服务器对客户端请求的响应状态**。其中，200 表示成功，表示服务器成功处理了客户端的请求，并返回了请求的内容。

如果要把状态码区分为客户端还是服务端的问题，你觉得客户端问题的返回码主要是什么，

以哪些开头的返回码是客户端的问题？

客户端**问题的返回码主要以 4 开头**，比如 400（Bad Request）、401（Unauthorized）、403（Forbidden）、404（Not Found）等。

**快排的时间复杂度是多少？**

快速排序的**平均时间复杂度为 O(nlogn)，最坏情况下的时间复杂度为 O(n^2)。**

**浅拷贝和深拷贝：**

浅拷贝只做了简单的赋值，**不开辟新空间**，只是增加一个指针，**指向原有的内存，共享相同的资源。**

深拷贝会开辟新的内存空间，**把原有的对象复制过来，拥有独立的资源**，。

**默认拷贝构造函数执行的是浅拷贝。**

**要深拷贝的话你自己定义**

**浅拷贝这会有问题，如果是动态成员，**在销毁对象时，**两个对象的析构函数将对同一个内存空间释放两次，这就会引发错误。**

另外对于stactic变量也不好操作

拷贝构造函数必备的参数：

是本类型的一个**引用变量，**但并不限制为const，一般普遍的会加上const限制。

**说一下你对Reactor模型的理解**

Reactor 模式也叫**做事件分发模式，**主要由 **Reactor 和处理资源池**这两个核⼼部分组成，

其中

1.**Reactor负责监听和分发事件**，事件类型包含连接事件、读写事件；

2.**处理资源池包含多个业务线程or进程）负责事件的处理**

他有三种常用模式

**第一个是单 Reactor 单进程 / 线程模式**

所有的工作都在一个进程/线程内完成，**Reactor 通过I/O多路复用模块监听事件并分发，Acceptor对象处理连接事件建立连接**，**Handler对象处理读写事件**。

然而，这种模式存在两个问题：一**无法充分利用多核 CPU 的性能**，二是在**处理业务时会阻塞整个进程，导致响应延迟。**

**第二个是单 Reactor 多进程 / 线程模式**

与前面有一些不同，总的来说，**主线程依然负责建立连接，监听分发事件，还有数据的I/O操作**（handler）,但是**数据业务处理交给其中一个子线程负责了**

**优势：利用到了多核优势，从而实现并行处理。**

**最大问题：依然是⼀个 Reactor 对象承担所有事件的监听和分发，⽽且只在主线程中运⾏，在⾯对瞬间⾼并发的场景时，容易导致性能出现瓶颈。**

**第三个是多 Reactor 多进程 / 线程（MUDUO就是这种）**

在这种模式下，**主线程只负责接收新连接，将新的连接分配给某个⼦线程**；每个子线程都有一个reactor对分发的连接进行事件监听，当有事件发生时，调用handler对象来处理事件，包括数据的I/O以及业务处理。

**优势在于**他的每个子线程的 Reactor 可以专注于自己负责的连接，**避免了单个 Reactor 处理大量连接时可能出现的性能瓶颈。**

说下两个高性能网络模式的区别

Reactor 是⾮阻塞同步⽹络模式，感知的是就绪可读写事件。在每次感知到有事件发⽣（⽐如可读、写就 绪事件）后，就需要应⽤进程**主动调⽤ read/write ⽅法来完成数据的读取/写入**，然后进行数据处理

Proactor 是异步⽹络模式， 感知的是已完成的读写事件。在发起异步读写请求时，需要传⼊数据缓 冲区的地址（⽤来存放结果数据）等信息，这样系统内核可以⾃动帮我们完成数据的读写⼯作，并不需要像 Reactor 那样还需要应⽤进程主动发起 read/write 来读写数据，操作系统完成读写⼯作后，就会通知应⽤进程直接处理数据。

**说一下struct和class的区别：**

**1.默认访问控制权限：**

在 struct 中，**默认的成员访问权限是 public**，即结构体中定义的成员默认为公共成员，**可以在外部直接访问。**

在 class 中，**默认的成员访问权限是 private**，即类中定义的成员默认为私有成员，**不能在类外直接访问。**

**2.继承方式：**

在 struct 中，**默认的继承方式是 public 继承**，即基类的 public 和 protected 成员在派生类中仍然保持相同的访问权限。

在 class 中，**默认的继承方式是 private 继承**，即基类的 public 和 protected 成员在派生类中会变成 private 成员。

**进程，线程，协程是否有自己独立的栈堆区？**

**进程有独立的堆区和栈区，线程共享进程的堆区但拥有独立的栈区，而协程通常共享堆区**，并且拥有自己的、通常是**程序运行时分配的栈区**。**协程的栈区通常有别于传统的操作系统提供的线程栈，更加灵活和高效**

**虚函数能被声明为static类么**

因此虚函数不能被声明静态

虚函数是**通过指针的动态绑定实现多态性的，它需要根据对象的实际类型来确定调用哪个函数**。**静态成员函数与特定对象实例无关**，它属于类而不属于对象，没有虚函数表的概念，不支持动态绑定，也不会在子类中被重写。

**构造函数调用虚函数**

**不太合适：**

**​**

当一个派生类对象被创建时，它的构造函数会先调用其基类的构造函数，然后按照声明顺序初始化其成员，并最后执行其自身的构造函数代码。所以当在一个构造函数中调用虚函数时，此时**派生类的对象还没有被完全构造好**，**只会使用基类的虚表，执行基类版本的虚函数，而不会执行派生类版本的虚函数。**从而导致出现错误，

**为什么需要虚析构函数？**

当你想通过基类指针去删除派⽣类对象中时，如果析构函数不是虚函数**，不会触发动态绑定，**只会调⽤基类析构函数，导致派⽣类资源⽆法释放，**造成内存泄漏。**

只有当一个类被用来作为基类的时候，才把析构函数写成虚函数。

**构造函数为什么不能被声明为虚函数。**

不能也不需要

1. **虚函数的执行依赖于虚函数表**，而虚函数表是**通过构造函数中初始化**的。如果构造函数是虚函数，那么你执行虚构造函数就需要找虚函数表，但是你构造函数都还没执行，哪来的虚函数表。所以不能是虚函数。
2. 构造函数目的是**初始化对象，对象类型已经确定，在对象创建时被调用，无需动态绑定**。

**说一下PCB**

PCB 它是操作系统中用于存储关于进程信息的一个重要数据结构。

**PCB 是操作系统用来管理和跟踪进程状态的一种方式，确保进程能够有序地执行和切换。**

通常包含以下信息：

1. **进程标识符**（PID）：用于区分不同的进程的唯一的标识号。
2. **进程状态**：如就绪、运行、阻塞、终止）等。
3. **程序计数器**：指向进程下一个要执行的指令地址。
4. **CPU 寄存器信息**：进程被中断或切换时，保存CPU 寄存器中的数据，以便恢复时可以继续执行。
5. **CPU 调度信息**：包括**进程优先级、调度队列指针和其他调度参数**。

### RAII原则的核心思想：

RAII的核心思想是将资源的获取(即分配)和释放与对象的生命周期绑定,通常**通过构造函数来获取资源,并在析构函数中释放资源。**

这样做有几个好处：

**安全性**：**由于资源的释放是自动的**，无论是因为**正常流程结束还是因为异常导致的提前退出，对象的析构函数都会被调用，资源也相应地会被释放**。

**简洁性**：因为资源的管理是自动的，代码通常更加简洁，不需要**程序员显式编写资源释放代码**。

**内存管理：**

标准库中的**智能指针如std::unique\_ptr和std::shared\_ptr遵循RAII原则**，自动管理动态分配的内存。当**智能指针对象销毁时，其所拥有的内存会被自动删除，防止内存泄漏。**

**线程安全性增强：**

RAII常用于实现互斥锁、信号量等线程同步机制，如std::**lock\_guard用于自动锁定和解锁互斥量。在进入作用域时获得锁，在离开作用域时自动释放，避免了忘记释放锁造成的死锁问题。**

### RAII在STL中的应用：

在STL中，RAII广泛应用于各种容器和其他组件中。例如：

* **智能指针**：std::unique\_ptr 和 std::shared\_ptr 是智能指针类，它们对动态分配的内存进行管理。**当智能指针的实例离开作用域时，其析构函数会自动释放其所管理的内存。**
* **容器类**：如 std::vector、std::string、std::map 等，都负责自己内部数据的内存管理。**当一个容器对象被销毁时，它的析构函数会释放所有占用的内存，并适当地销毁其元素。**
* **锁管理类**：如 std::**lock\_guard** 和 std::unique\_lock，它们在构造时获取锁，在析构时释放锁，**从而确保在持有锁的代码块执行完毕后，无论是正常退出还是因异常退出，锁都会被释放。**

通过将资源管理与对象生命周期紧密关联,RAII 提高了程序效率和安全性。

**内存分配**方式有三种：

（1）**从静态存储区域分配**。这块内存在**程序编译的时候就已经分配好**，在程序的整个运行期间都存在。例如**全局变量，static变量**。

（2）**在栈上创建**。在执行函数时，可以在栈上创建函数内局部变量的存储单元，**函数执行结束时这些存储单元自动被释放**。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

（3）从**堆上分配，亦称动态内存分配**。程序在运行的时候用malloc或new申请任意量的内存，程序员自己在任意时候用free或delete释放内存。**动态内存的生存期由我们决定，使用非常灵活**，但问题也最多。

**C++ 没有内存回收，C++ 中分配内存要考虑到什么问题？**

内存泄漏（Memory Leaks）：

在C++中，**动态分配内存后**需要**确保在不再需要时及时释放，否则没有及时释放会导致内存泄漏。导致**系统资源耗尽、性能下降等着

内存溢出（Memory Overflow）：

内存溢出发生在程序试图写入超出其分配内存范围的数据时，可能会导致程序出现**不可预测的行为或崩溃**。在C++中，手动管理内存意味着需要确**保分配的内存足够大以容纳程序需要的数据**，并且要避免写入超过内存边界的数据。

野指针

**野指针是指指向已释放内存的指针。当程序释放了某个对象的内存但仍然保留了指向该内存的指针时**，如果后续程序尝试使用该指针，可能会导致未定义的行为或崩溃。

**避免内存泄漏的方法有哪些？**

避免内存泄漏的方法包括：，

**手动管理内存**，确保在动态分配内存后，每个 new 操作都有对应的 delete 操作，并确保 **delete 操作在适当的时候被调用，以避免内存泄漏。**

合理使用 RAII（资源获取即初始化）技术，

使用智能指针管理内存，及时自动释放动态分配的内存

**那有没有其他更好的方式避免内存泄漏？**

智能指针它们可以自动管理内存的生命周期，确保在对象不再需要时自动释放内存，从而避免内存泄漏的发生。。

**智能指针是用什么方式避免内存泄漏的？**

**第一是共享指针**，共享指针允许多个指针共同管理同一个对象。共享指针**内部维护了一个引用计数**，用于记录有多少个共享指针指向同一个对象。当引用计数为0时，表示没有任何共享指针指向该对象，对象会被销毁。

共享指针使用引用计数来管理内存，但会导致循环引用的问题。**如果两个或多个对象之间形成了循环引用，**它们的引用计数永远不会归零，导致内存泄漏。

**第二个是weak\_ptr**

**weak\_ptr是为了配合shared\_ptr⽽引⼊的⼀种智能指针 它的最⼤作⽤在于协助shared\_ptr⼯作，像旁观者那样观测资源的使⽤情况，但weak\_ptr没有共享资源，它的构造 不会引起指针引⽤计数的增加。**

所以弱指针可以打破AB的循环引用，B对A用弱指针，A对B用共享指针，所以可以先释放A，A没了就释放指向B的共享指针，B的引用计数为0，就能成功释放了

1. 转换为 shared\_ptr 进行访问： 当需要访问所管理的对象时，可以通过将弱指针转换为共享指针来实现。这样，如果对象还存在，则转换操作成功并返回一个有效的共享指针；如果对象已经被释放，则转换操作失败，返回一个空的共享指针
2. **unique\_ptr**”唯⼀”拥有其所指对象 同⼀时刻只能有⼀个unique\_ptr指向给定对象，当 std::unique\_ptr 被销毁时，它所拥有的资源会被自动释放，从而避免了内存泄漏。

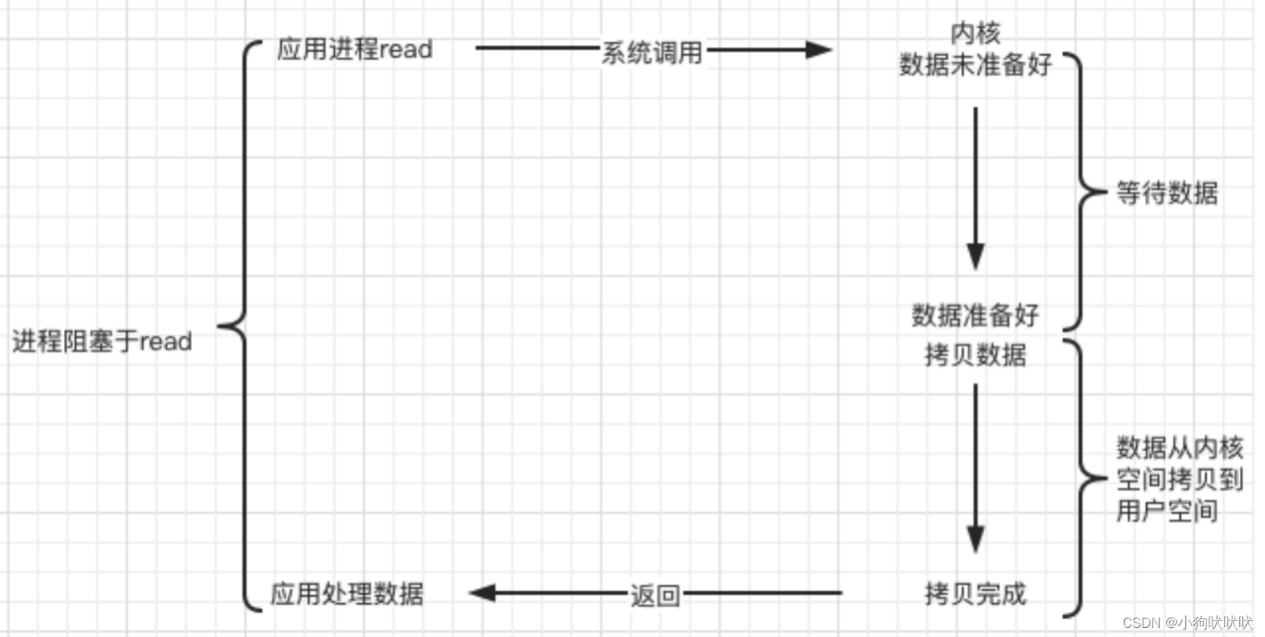
**你怎么检测内存泄露：**

**说下你知道的I/O模式：**

I/O总结

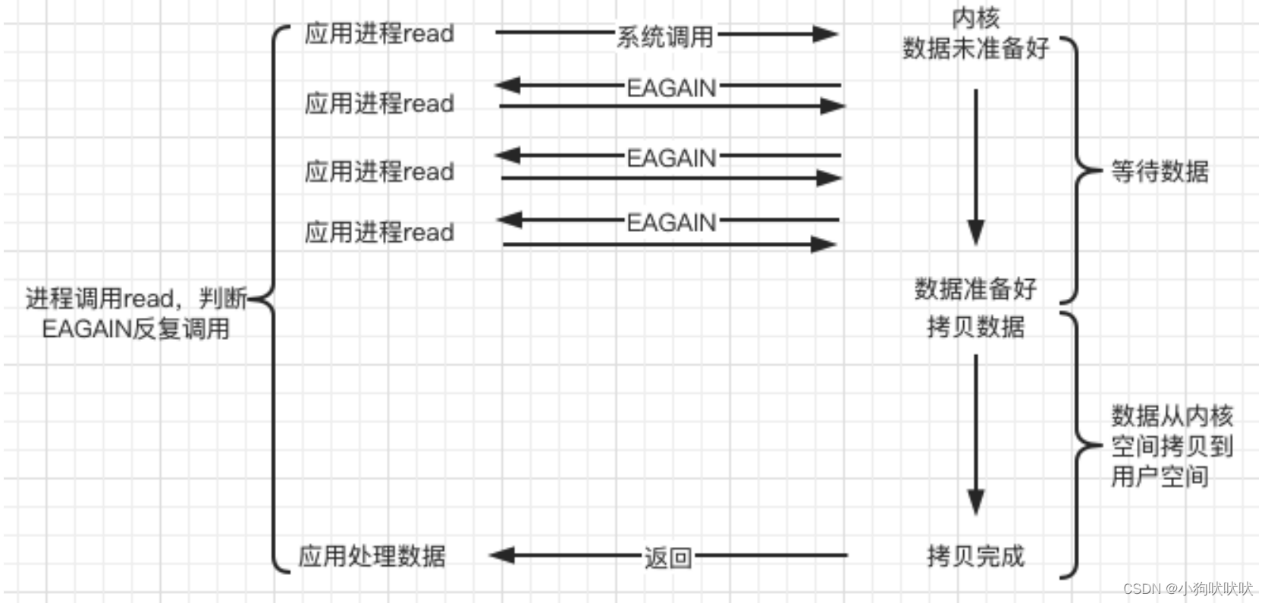
以读数据为例子

阻塞IO（调用read后，进程会被阻塞，⼀直等到内核数据准备好，并把数据从内核缓冲区拷⻉到应⽤程序的缓冲区中，当拷⻉过程完成后才返回）

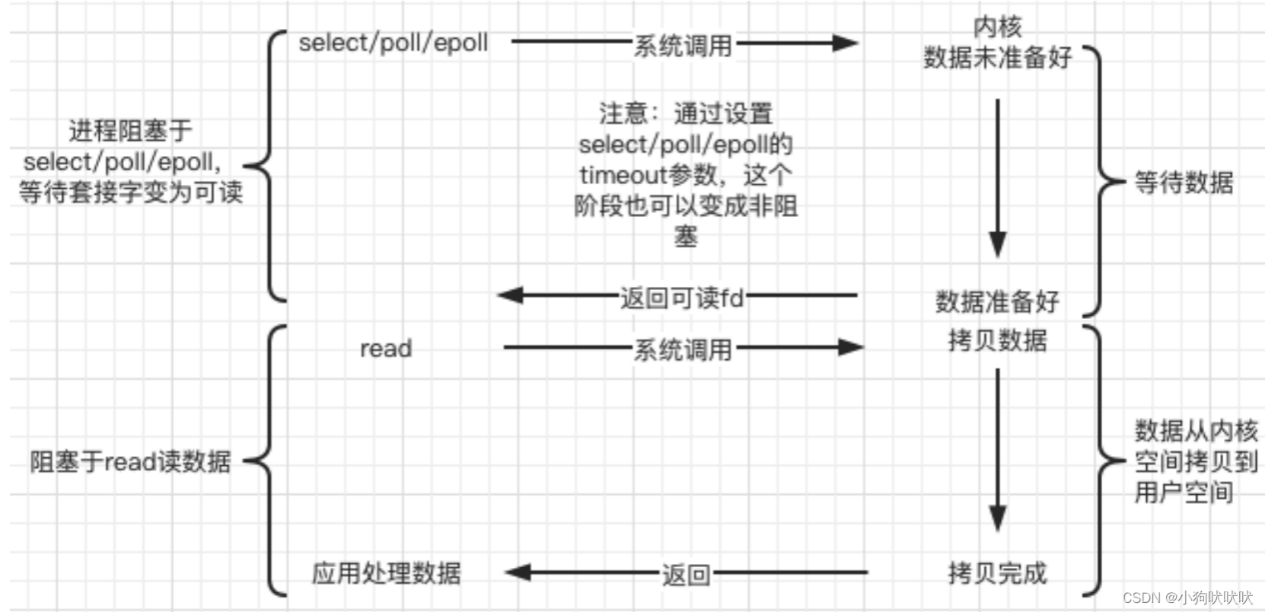


非阻塞I/O

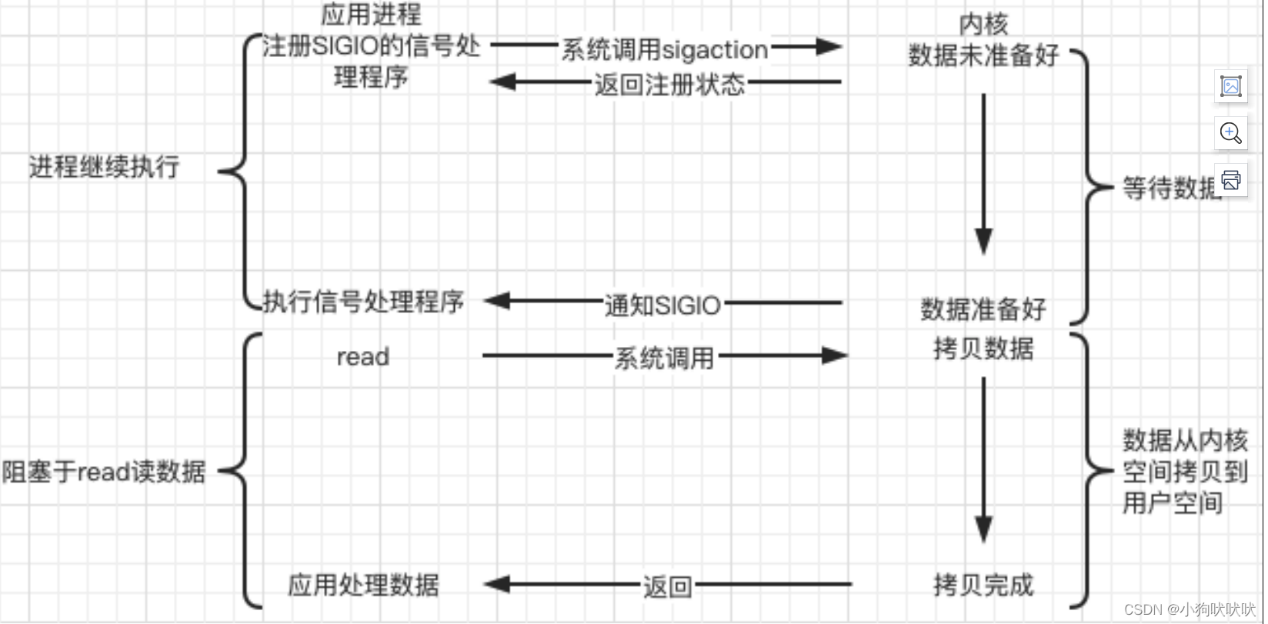
（每隔一段时间调用read轮询内核否准备好数据，没准备好就先返回干自己的事，准备好了就进行数据拷贝）



I/O多路复用 （先阻塞在epoll,select等系统调⽤中之上等待套接字发生事件，**当检测到就绪事件发生，**也就是内核准备好数据，就返回可读的套接字，执行read调用区拷贝数据）



信号驱动I/O（sigaction 系统调⽤注册⼀个信号处理函数，然后⽴即返回，不阻塞，进程继续执⾏，**内核数据准备好后，发送信号通知执行信号处理函数**，使用read系统调用拷贝到用户空间。相当于半异步）



异步I/O（⽤户进程调⽤aio\_read函数，给**内核传递描述符、缓冲区指针、缓冲区⼤⼩等数据**，**让内核完成所有的操作，然后进程继续执行**，当数据拷贝完成后，**内核会通知你处理数据;**

相比于前几个，**异步IO通知的是事件完成，用户程序不用承担数据拷贝所花费的时间，拷贝操作已经由内核帮你完成，**

什么是IO多路复⽤

**IO多路复用是一种高效的IO处理方式，它允许单个进程或线程同时监视多个文件描述符，如网络连接或文件句柄。当这些描述符中的任何一个就绪时，比如有数据可读或可写，就能够通知应用程序进行相应的操作。**

这种机制的核心优势在于，它可以在不增加额外线程或进程的情况下，处理大量的并发连接，从而显著地提高系统的并发性和响应能力。

常见的IO多路复用技术包括select、poll和epoll等。

**说一下select,poll,epoll**

* select是最早的一种I/O多路复用技术,它可以监视多个⽂件描述符的可读、可写和错误状态。select有几个明显的缺点。首先，它监视的文件描述符数量有限。其次，每次调用select时，都需要将整个⽂件描述符集合从用户态拷贝到内核态，并且在内核中需要**遍历所有被监视的文件描述符**来检查是否有就绪的，在文件描述符数量很多时开销会变得非常大。最后，select返回后，**还需要再次遍历**⽂件描述符集合**来找出哪些文件描述符是就绪的**，这进一步增加了处理开销。

poll:

* 与select相比，**poll没有文件描述符数量的限制，因为它基于链表来存储**。然而，poll跟select一样仍然需要在**每次调用**时拷贝⽂件描述符集合到内核态，也需要遍历整个集合**检查每个文件描述符的状态**。因此，当文件描述符数量很多时，poll的性能也会下降。

Epoll:

epoll它是linux中的一种**事件驱动型**的I/O多路复用技术，。epoll的优势在于它可以同时处理大量的文件描述符，而且不会随着文件描述符数量的增加而降低效率。

epoll 通过两个核心数据结构，红黑树和就绪队列，很好解决了 select/poll 的问题。

* 。第一点\_，epoll在内核里使**用红黑树来跟踪进程所有待检测的文件描述字,**通过epoll\_ctl()函数**把需要监控的文件描述符加入内核中的红黑树里**。红黑树是个高效的数据结构，增删改一般时间复杂度是 O(logn)。select/poll**内核里是没有能保存所有待检测文件描述的数据结构的**，所以select/poll**每次操作时都传入整个socket集合给内核。**而epoll在**内核维护了红黑树**，**可以保存所有待检测的socket**，这大大减少了内核和用户空间之间的拷贝开销。

第⼆点，就绪队列：内核⾥维护了⼀个链表来记录就绪事件，当某个 socket就绪，通过回调函数，内核会将其加⼊到这个就绪事件列表中。最后，用户**调用epoll—wait返回时就能知道哪些文件描述符是就绪的，因此无需再像select和poll那样进行**遍历整个文件描述符集合，⼤⼤提⾼了检测的效率。

**说一下红黑树**

红黑树是一种**自平衡二叉查找树**，具有以下特点：

1. 每个节点要么是红色，要么是黑色。
2. 根节点是黑色。
3. 叶子节点（NIL节点）是黑色。
4. 如果**一个节点是红色，则它的子节点必须是黑色**。
5. 从**任一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**通过对节点着色和旋转等操作来维持平衡**，**保证从根到所有叶子节点的最长路径不会超过最短路径的两倍。以及**最坏情况下的查询、插入、删除等操作的**时间复杂度为 O(log n)**，是广泛应用于数据结构和算法中的一种重要数据结构。

红黑树通常用于实现**关联容器**，如 C++ STL 中的 map 和 set。它的平衡性质使得在**最坏情况下的时间复杂度保持在 O(log n)**，适用于大多数的动态集合操作场景。

**与平衡二叉树和B/B+树相比，红黑树有以下几点区别：**

1. 平衡二叉树（如AVL树）：平衡二叉树要求左右子树的高度差不超过1，因此需要频繁地进行旋转操作来维持平衡，相比之下红黑树更加灵活，在实际应用中性能更好。
2. B/B+树：B树和B+树是多路搜索树，适合磁盘存储等场景，可以一次读取多个关键字。而红黑树更适用于内存中数据结构的实现，虽然红黑树也可以作为B树和B+树的基础实现。
3. 红黑树相对于B树和B+树来说，**对于数据的插入、删除等操作可能更为简单高效**，但在某些特定场景下B树和B+树可能更适合。

**Map和unordermap:**

map和unordered\_map都是**C++中的关联容器，用于存储键值对。**其**主要区别在于**底层实现方式和性能表现。

**1、底层实现方式**

  map内部使用红黑树（一种自平衡二叉查找树）来实现，而unordered\_map底层则使用**哈希表来实现**。这意味着，在map中，**元素是按照键的大小进行有序排列的**，而在unordered\_map中，**则不保证元素的顺序。**

**2、性能表现**

  当**需要有序地遍历元素时**，**map的性能比unordered\_map更好**。但是，**当需要快速查找特定的元素时，unordered\_map通常比map更快**。**map在插入和删除操作时需要维护红黑树的平衡**，而unordered\_map**则只需要计算哈希值并将元素放入相应的桶中即可**。

最好可以达到时间复杂度O(1)的访问性能，但在最坏情况下可能退化为线性时间复杂度O(n)。

**如何⽤哈希设计出类似红⿊树的效果（**

在设计哈希表时，可以借鉴红黑树的一些特性来实现类似的效果。下面是一些可能的方法：

**平衡哈希表：**可以设计一种平衡的哈希表结构，使得在哈希表中存储的元素在空间上分布均匀，避免出现大量冲突导致链表过长的情况。这样可以保证在进行哈希查找时，平均查找时间较短。

**分级哈希：**可以将哈希表分成多个级别，每个级别对应一个哈希表。当某个哈希表的冲突较多时，可以将其中的元素重新哈希到下一个级别的哈希表中。这样可以避免单个哈希表过度冲突导致性能下降的问题。

**自平衡哈希表：**可以设计一种自平衡的哈希表结构，使得在进行插入和删除操作时，哈希表可以自动调整，保持平衡。这样可以避免出现某些位置过度拥挤而导致的性能问题。

**哈希表 + 树结构：**可以将哈希表与树结构相结合，使得在哈希冲突的情况下，使用树结构来解决冲突。例如，可以在哈希表的每个槽位上维护一个小型的红黑树，来存储发生冲突的元素。这样可以保持哈希表的高效性，并且在冲突较多时仍能保持较高的性能。

总的来说，设计类似红黑树效果的哈希表需要综合考虑哈希算法、冲突处理、平衡性等因素，采取合适的策略来保证哈希表的高效性和稳定性。

**说下epoll的触发模式：**

边缘触发（edge-triggered，ET）和⽔平触发（level-triggered， LT）。

1边缘触发（Edge-Triggered，ET）：

* **当某个文件描述符上有事件发生时，epoll只会通知应用程序一次。**
* **如果应用程序没有完全处理事件，**比如读缓冲区内数据没被读完**，下一次调用epoll\_wait将不会再次通知该事件。**
* **所以应用程序需要一次性处理完事件**，比如⼀次性将内核缓冲区的数据读取完。

2.水平触发（Level-Triggered，LT）：

* **如果应用程序没有处理完事件，下一次调用epoll\_wait时会再次通知应用程序，直到事件被处理完毕。**
* 比如有可读事件发⽣时，服务器端不断地从 epoll\_wait 中苏 醒，直到内核缓冲区数据被 read 函数读完才结束

**悬挂指针和野指针**

**悬挂指针**: **悬挂指针产生于指针所指向的内存已被释放或者失效后，指针本身没有及时更新或清空。**在该内存释放之后，任何通过这个悬挂指针的引用或操作都是不安全的，因为这块内存可能已经重新分配给了其他的数据。

示例：当一个指针指向动态分配（比如使用malloc或new）的内存，并且随后该内存被释放掉（使用free或delete），而没有将指针设置为NULL，此时这个指针就变成了悬挂指针。

**野指针**: 野指针通常是**指未初始化的指针，它没有被设置为任何有效的地址。**由于它可能指向任意位置，对野指针的解引用是危险的，并且可能会导致难以预测的行为甚至程序崩溃。

示例：声明了一个指针变量但是没有给它赋予确定的初始值，然后就开始使用这个指针。

尽管两者看似相似，但是产生原因和解决方式有所不同：

* **悬挂指针问题**可以通过确保指针在释放关联的内存资源后立即被设为NULL来避免。
* **野指针问题**则需要确保每个指针变量在使用前都被明确初始化为一个合法的地址或NULL。

处理这两种类型的指针时，编程中的最佳实践是始终确保你的指针在声明后得到适当的初始化，在资源被释放之后更新状态，并且在解引用之前检查其有效性。

常量指针和指针常量

核心区别在于：

* 常量指针主要是这个指针指向的数据不能通过指针来改变，但允许改变指针指向的地址。const int \*ptr;
* 指针常量：一旦指针初始化了，他所指向的地址就不能改变了，但允许通过这个指针改变所指向的数据（除非它同时也是一个常量指针，下文将说明）。int \*const ptr = &someVariable

还有一种情况是两者结合使用，创建一个指向常量数据的固定指针（即指针的地址和它指向的数据都不能改变）：

const int \*const ptr = &someConstantVariable;

在这种声明中，ptr是一个指针常量，同时指向一个整型常量，因此无法修改ptr的值（地址），也不能通过ptr来修改其指向的数据。

# **如果A这个对象对应的类是一个空类，那么sizeof(A)的值是多少？**

1. 空类的实例在内存中至少占用一个字节的空间，这是为了确保每个实例都有独一无二的地址。

**栈和堆的区别？**

**分配方式：**

栈由**系统自动分配和释放内存**，分配内存连续有序。

堆由程序员**手动分配和释放内存**，若没有及时释放**容易造成内存泄露**。

**大小：**

栈**大小固定**。

堆**大小不固定**，受限于系统虚拟内存大小。

**生命周期：**

**栈上的数据生命周期短暂，随函数调用结束自动销毁。**

**堆上的数据由程序员手动控制生命周期。**

**访问速度：**

栈访问速度快。

堆访问速度相对慢。

**储存数据**

**栈用于存储局部变量、函数调用的参数和控制流信息，适用于临时数据**。

堆用于**存储动态分配的数据**，如动态数组、对象实例等，适用于**需要长期存储和共享的数据**。

# **在C++中，用堆和用栈谁更快一点？**

一般情况下，栈数据的访问速度更快。

**栈相对于堆更快的原因主要有以下几点：**

**硬件支持：**操作系统底层提供对栈的专门支持，包括**专门的寄存器存储栈的地址和专门的指令执行栈操作**。这些硬件支持使得栈的操作更加高效。

**操作简单：分配内存仅仅是涉及到移动栈指针的操作**。由于栈是线性且连续的内存区域，增加或减少栈空间只需要调整栈顶指针的位置。

相比之下，**堆的分配由C++库函数提供支持**，堆的分配过程涉及到更多的步骤，包括在堆的数据结构中寻找合适大小的空闲块，添加新的内存页到进程空间，**这些过程需要一定的算法支持和复杂的内存管理操作**。

**内存局部性：**栈上的数据通常具有更好的缓存局部性。因为栈是连续分配的，最近分配的变量很可能在CPU缓存中，而堆上的对象可能分布在内存的不同地方，导致缓存命中率降低。缓存局部性可以提高数据访问速度。

**静态内存分配和动态内存分配？**

静态内存分配和动态内存分配是两种不同的内存管理方式，它们的主要区别在于分配时间、大小变化和灵活性：

1. 静态内存分配：在**程序编译阶段，内存就已经被分配好**。程序运行时，静态分配的内存大小是固定不变的。比如全局变量，常量，以及函数中声明的局部变量等都是静态分配内存的。优点是**管理起来比较简单，缺点是可能会造成内存空间的浪费**。
2. 动态内存分配：在程序**运行过程中**，根据需要动态地分配内存空间。

使用者可以自己控制何时**申请内存，何时释放内存，具有很大的灵活性**。例如在C语言中，我们可以通过malloc，calloc，realloc等函数进行动态内存分配。**优点是可以更有效地利用内存资源，缺点是如果管理不当（例如忘记释放内存），就可能导致内存泄漏问题。**

**在窗口满了的情况下，write 函数的行为取决于文件描述符的设置：**

**阻塞模式：**在窗口满了的情况下，如果缓冲区已满，则 write 函数会一直阻塞**直到有足够的空间**写入数据或者发生错误。

**非阻塞模式：**在窗口满了的情况下，**如果缓冲区已满，则 write 函数会立即返回 -1**，并设置错误码 errno 为 EAGAIN 或者 EWOULDBLOCK，表示资源暂时不可用或者操作会阻塞。这样，**应用程序可以根据返回的错误码进行相应的处理，如等待一段时间后再尝试写入。**

**左值，右值：**

1.左值是一个**表示数据的表达式**，比如：变量名、解引用的指针变量。一般地，**出现在赋值符号的左边。**我们可以****获取它的地址****和****对它赋值****，

右值也是一个表示数据的表达式，比如：*字面常量、表达式返回值，传值返回函数的返回值（是传值返回，而非传引用返回）*，**右值不能出现在赋值符号的左边且不能取地址和赋值**。

**左值引用就是对左值的引用，给左值取别名。**

意义：传值传参和传值返回都会产生拷贝，有的甚至是深拷贝，代价很大。而左值引用的实际意义在于做参数和做返回值都可以减少拷贝，从而提高效率。

右值引用引用右值，会使右值被存储到特定的位置。

也就是说，**右值引用变量其实是左值，可以对它取地址和赋值**（const右值引用变量可以取地址但不可以赋值，因为 const 在起作用）。

当然，取地址是指取变量空间的地址（右值是不能取地址的）。

比如：

double&& rr2 = x + y;

&rr2;

rr2 = 9.4;

右值引用 rr2 引用右值 x + y 后，该表达式的返回值被存储到特定的位置，不能取表达式返回值 x + y 的地址，但是可以取 rr2 的地址，也可以修改 rr2 。

**move**，本文指std::move（C++11），作用是将一个左值强制转化为右值，以实现移动语义。  
左值被 move 后变为右值，于是右值引用可以引用。

**右值引用，用来支持移动语义和完美转发。**

1. 支持移动语义：在移动构造函数和移动赋值运算符中，**通过右值引用可以高效地将资源的所有权从一个对象转移到另一个对象，**而不必创建并删除临时对象。这样就**避免了拷贝复制浪费资源与效率**。

**注意：C++中的move函数只是做了类型转换，并不会真正的实现值的移动！！！** 要实现真正的移动，得自己手动重载移动构造函数和移动复制函数。**我们需要在自己的类中实现移动语义，避免深拷贝，充分利用右值引用和std::move的语言特性。**

1. 完美转发是指在函数模板中，完全依照模板的参数类型，将参数传递给当前函数模板中的另外一个函数。因此，为了实现完美转发，除了使用万能引用之外，我们还要用到std::forward（C++11），它在传参的过程中保留对象的原生类型属性。这样右值引用在传递过程中就能够保持右值的属性。

万能引用：

**确定类型**的 && 表示右值引用（比如：int&& ，string&&），  
但函数模板中的 && 不表示右值引用，而是万能引用，**模板类型必须通过推断才能确定**，其接收**左值**后会被推导为**左值引用**，接收**右值**后会被推导为**右值引用**。

**说一下静态变量，全局变量，局部变量。**

全局变量：

* **全局变量是在任何函数之外定义的变量，其生存期延长到整个程序运行期间。作用域是整个程序，可以在程序的任何地方访问。不过**全局变量的使用会增加了程序的耦合度，容易导致代码难以维护和理解。
* **全局变量用于在整个程序中需要共享的数据**

局部变量：

* 局部变量是在函数内部声明的变量，作用域仅限于声明它的函数内部，只能在声明它的函数内部访问。**生存期仅限于函数的执行过程中**

**局部变量常用于临时存储数据、函数内部计算中的中间结果**等。

static 关键字主要⽤于控制变量和函数的⽣命周期、作⽤域以及访问权限。

静态变量ᰁ

使⽤ **static 关键字修饰的变量称为静态变量**。

静态变量**在程序的整个⽣命周期内存在，不会因为离开作⽤域⽽被销毁**，默认初始化为零（对于基本数据类型）

1.**静态局部变量**：

**只能在声明它的函数内部访问**。不管几次函数调用都**只会初始化一次**

**用于需要保持状态的函数**，例如**统计函数调用次数（计数器）、缓存中间结果**等。

**静态全局变量**：**它的作用域是整个文件，但是只能在声明它的文件内部访问。静态全局变量的生存期延长到整个程序运行期间，但是它的可见性被限制在当前文件内部**，不会被其他文件引用

2. **静态成员函数**

在类内部使⽤ static 关键字修饰的函数是静态成员函数。

**静态成员函数属于类⽽不是类的实例，可以通过类名直接调⽤，⽽⽆需创建对象**。

**静态函数不能直接访问⾮静态成员变量或⾮静态成员函数。**

3. **静态成员变量**

在类中使⽤ static 关键字修饰的成员变量是静态成员变量。

意味着它为该类的所有实例共享,**也就是说当某个类的实例修改了该成员变量,其所有实例的静态成员变量的值也会一起修改**

静态成员变量必须在类外部单独定义，以便为其分配存储空间。

场景：

1. **静态变量用于需要保持状态的函数中，如计数器、缓存**等。
2. **全局变量用于在整个程序中需要共享的数据**，但应尽量避免过多的全局变量，以免导致代码的可维护性降低。
3. **局部变量用于临时存储数据，其作用域仅限于函数内部**，不会污染全局命名空间，提高了代码的封装性和可维护性。

**静态函数**

静态函数是指在函数声明或定义中使用 static 关键字修饰的函数。

**静态函数的作用域被限制在声明它的文件内部，它们不能被其他文件中的函数调用。**这样可以防止其他文件中的函数意外地调用到静态函数，增强了程序的封装性。静态函数的生命周期与程序的运行周期相同，它们在程序启动时被加载到内存中，并一直存在直到程序结束。

**Socket套接字**

**在网络编程中，可以通过设置套接字选项 SO\_REUSEADDR来实现端口复用**。在TCP/IP协议中，允许多个套接字（即多个连接）可以绑定到相同的IP地址和端口上，这也称为地址复用（Address Reuse）。使得在一个端口上可以同时监听多个套接字。

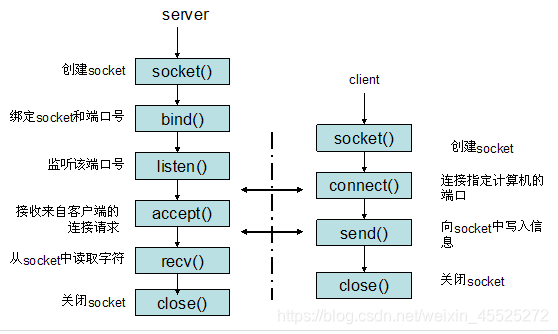
int opt = 1;

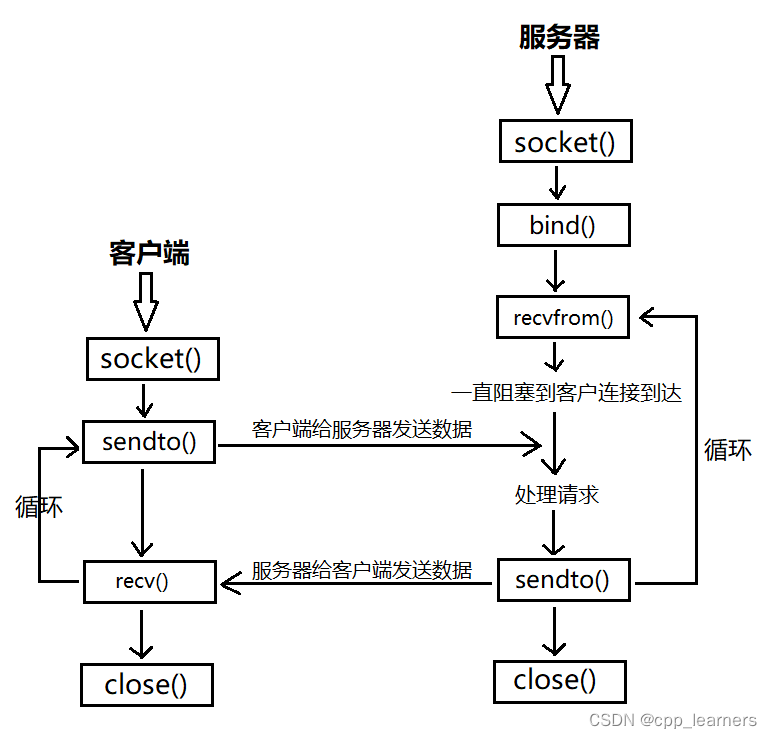
if (setsockopt(sockfd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt)) < 0) {

perror("setsockopt");

exit(EXIT\_FAILURE);

**TCP的UDP的Socket套接字编程区别：**





UDP特性，发送数据，使用sendto

接收数据，服务器端使用recvfrom

     客户端使用recv

**Send和recv常用于TCP协议,sendto和recvfrom常用于UDP协议**

1.

recv 适用于已连接的套接字，不需要提供发送端地址信息。

**recvfrom 适用于未连接的套接字，需要提供发送端地址信息。**

2.

send 适用于已连接的套接字，不需要提供接收端地址信息。

**sendto 适用于未连接的套接字，需要提供接收端地址信息**

ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags, **struct sockaddr \*src\_addr, socklen\_t \*addrlen**);

**多了两个参数，**目标地址和地址长度等。

**设s为套接字**

**send仅仅是把buf（应用程序）中的数据copy到s对应的内核中的发送缓冲区的剩余空间里，是TCP协议把的发送缓冲中的数据传到连接的另一端的。**如果send函数copy数据成功，就返回实际copy的字节数，copy时出错，那么它返回-1；0代表对端连接关闭或者，发送0字节。

**recv函数仅仅是从s的内核的接收缓冲区copy数据，真正的接收数据是TCP协议来完成的**，recv函数返回其实际copy的字节数。如果recv在copy时出错，那么它返回-1；如果recv函数在等待协议接收数据时网络中断了，那么它返回0。

**非阻塞和阻塞问题：**

**在阻塞模式下send**操作将会**等待所有数据均被拷贝到发送缓冲区后才会返回。**

**阻塞的send操作成功的话，返回的值必然是你参数中的发送长度的大小。**

**在非阻塞模式下send**操作调用会立即返回。一**次能copy多少是多少**，所以**返回的值可能小于你设置的发送长度**。**如果缓存区满了话直接返回-1；**

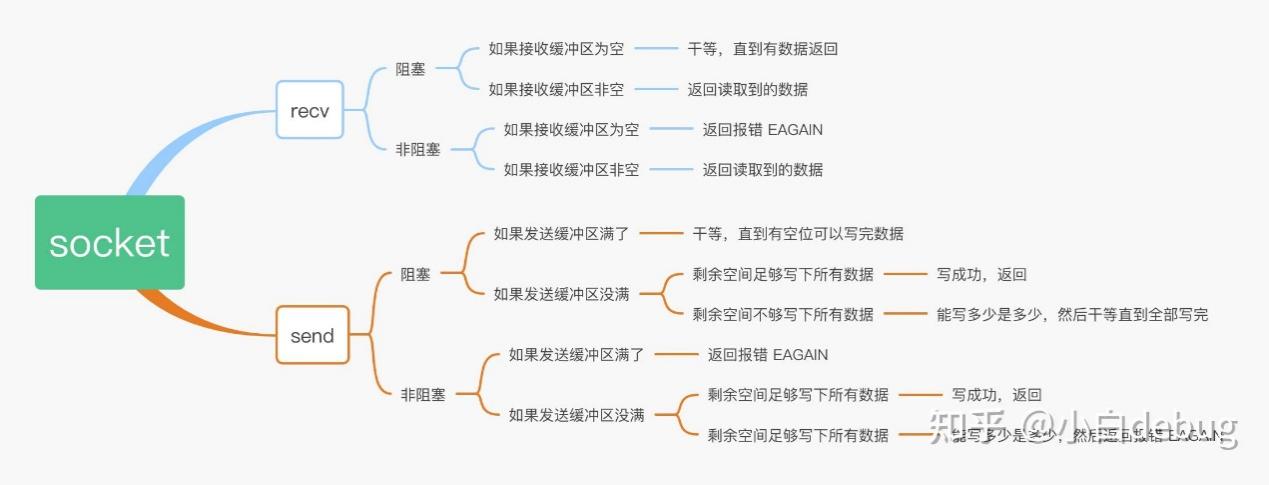
**在阻塞模式下的sendto**操作不会阻塞。

UDP并没有真正的发送缓冲区，**它所做的只是将应用缓冲区拷贝给下层协议栈，在此过程中加上UDP头，IP头，所以实际不存在阻塞。**

**在非阻塞模式下sendto**操作 不会阻塞（与阻塞一致，不作说明）。

**在阻塞模式下recv，recvfrom**操作在**缓冲区没有任何数据1情况下**将会阻塞 到缓冲区里有数据才返回。

**在非阻塞模式下recv，recvfrom操作将会立即返回**。如果缓冲区 有任何一个字节数据（TCP）或者一个完整UDP数据报，**它们将会返回接收到的数据大小**。而**如果缓冲区没有任何数据则返回错误-1；** WSAEWOULDBLOCK/E WOULDBLOCK。



**ping是哪个协议哪⼀层**

ping 是一种**网络诊断工具，用于测试网络的连通性**和测量数据包往返时间（RTT）。

ping 使用 **ICMP**协议来发送和接收 ICMP 报文。

ICMP 协议位于网络层（OSI 模型中的第三层），它是 **IP 协议的一个辅助协议**，**用于在IP网络中传递控制信息和错误消息**。

功能：

**发现网络错误：**当一个数据包在传输过程中出现错误时，ICMP协议通过向发送方发送错误通知来发现网络错误。

**检查网络是否可达：**通过发送ICMP ECHO请求并接收ICMP ECHO回复消息，可以确定目标主机是否可达。

**发现主机错误：**当一个主机无法正常工作时，ICMP协议通过向发送方发送错误通知来发现主机错误。

**发送路由信息：**ICMP协议可以向其他主机发送路由信息，以帮助它们在网络中找到合适的路由。

**ping 使用 ICMP 报文发送 ICMP 回显请求（Echo Request）给目标主机**，**目标主机收到请求后会回复 ICMP 回显应答（Echo Reply）给源主机，从而实现网络连通性的测试**

**TCP为什么不适合流媒体音视频？**

TCP的确认应答，超时重传、拥塞控制、和流量控制等机制虽然保证可靠性和有序性 但使 **无法提供低延迟的实时传输**，而这对音视频传输来说非常关键。所以不合适。

**进程，线程，协程区别？协程特点**

首先，我们来谈谈进程。进程是操作系统中进行**资源分配基本单位**，它拥有自己的**独立内存空间和系统资源**。每个进程都有独立的堆和栈，不与其他进程共享。进程间**通信需要通过特定的机制**，如管道、消息队列、信号量等。由于进程**拥有独立的内存空间，因此其稳定性和安全性相对较高**，但同时**上下文切换的开销也较大，因为需要保存和恢复整个进程的状态。**  
接下来是线程。**线程是进程内的一个执行单元，也是CPU调度的最小单位。与进程不同**，线程**共享进程的内存空间，所以**线程之间通信更加高效。**然而，由于多个线程共享内存空间，因此存在数据竞争和线程安全的问题，需要通过同步和互斥机制来解决。**线程的上下文切换开销较小，因为只需要**保存和恢复线程的上下文**（栈（局部变量）和寄存器），**而不是整个进程的状态**。

最后是协程。协程是一种**用户态的轻量级线程**，其**调度完全由用户程序控制，而不需要内核的参与**。协程拥有**自己的寄存器上下文和栈**，但与**其他协程共享堆内存**。协程的切换开销非常小，因为只需要**保存和恢复协程的上下文，而无需进行内核级的上下文切换**。这使得协程在处理大量并发任务时具有非常高的效率。然而，协程需要程序员**显式地进行调度和管理**，相对于线程和进程来说，**其编程模型更为复杂。**

说⼀说僵⼫进程和孤⼉进程 ，还有守护进程

1. 孤⼉进程：⼀个⽗进程退出，⽽它的⼀个或多个⼦进程还在运⾏，那么那些⼦进程将成为孤⼉进程。孤⼉进程 将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集⼯作。

2. 僵⼫进程：⼀个进程使⽤fork创建⼦进程，如果⼦进程退出，⽽⽗进程并没有调⽤wait或waitpid获取⼦进程 的状态信息，那么⼦进程的**进程描述符仍然保存在系统中**。这种进程称之为僵⼫进程。

3.  
守护进程（Daemon Process）是在计算机系统**后台运行的**一**种特殊类型的进程**，通常独立于用户会话，并且不受用户直接控制。守护进程通常在系统启动时启动，并在系统关闭时自动终止，它们在后台执行一些特定的任务，例如系统服务、监控、日志记录等。

**线程共享了进程哪些资源？**

**共享资源：**

**进程的代码段，数据段（全局变量、静态变量）和堆内存（动态分配的内存）**。

**文件描述符：**所有线程共享进程的打开文件描述符，可以访问相同的文件、管道、套接字等。

**信号处理器：**所有线程共享进程的信号处理器，可以共同处理收到的信号。

**进程相关的属性：**如进程 ID、进程组 ID、会话 ID 等，所有线程共享这些属性。

**程序计数器**

**环境变量**

**独立资源：**

**线程栈：**每个线程都有自己的栈空间，用于存储局部变量和函数调用信息。

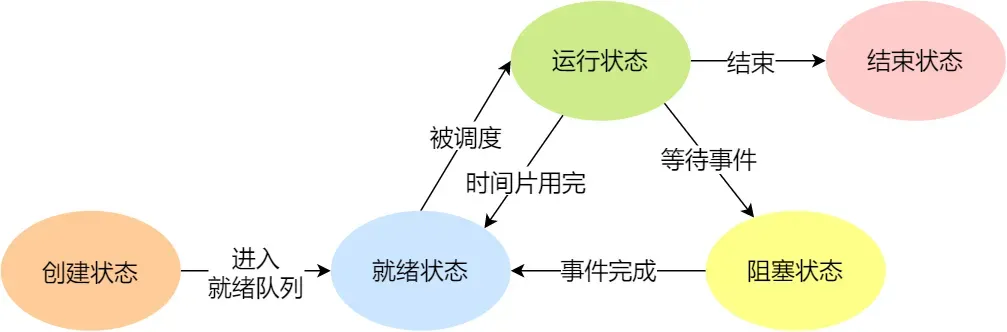
**寄存器集合：**每个线程都有自己的寄存器集合**，用于存储线程的执行状态和上下文信息。**

**线程 ID：**每个线程都有自己唯一的线程 ID，用于标识线程的身份。

**线程局部变量：**每个线程可以拥有自己的局部变量，不同线程之间互不影响。

**线程的调度策略、优先级等**。

### **进程的状态（五种状态），如何切换？**

一个完整的进程状态的变迁如下图：

进程五种状态的变迁 再来详细说明一下进程的状态变迁：

创建状态，就绪状态，阻塞状态，运行状态

* \_NULL -> 创建状态\_：一个新进程被创建时的第一个状态；
* \_创建状态 -> 就绪状态\_：当进程被创建完成并初始化后，一切就绪准备运行时，变为就绪状态，这个过程是很快的；
* \_就绪态 -> 运行状态\_：处于就绪状态的进程被操作系统的进程调度器选中后，就分配给 CPU 正式运行该进程；
* \_运行状态 -> 结束状态\_：当进程已经运行完成或出错时，会被操作系统作结束状态处理；
* \_运行状态 -> 就绪状态\_：处于运行状态的进程在运行过程中，由于分配给它的运行时间片用完，操作系统会把该进程变为就绪态，接着从就绪态选中另外一个进程运行；
* \_运行状态 -> 阻塞状态\_：当**进程请求某个事件且必须等待时，例如请求 I/O 事件**；
* \_阻塞状态 -> 就绪状态\_：当**进程要等待的事件完成时，它从阻塞状态变到就绪状态；**

**进程上下文切换和线程上下文切换的区别？**

进程上下文切换**不仅包含了虚拟内存、栈、全局变量等用户空间的资源**，还包括了**内核堆栈、寄存器等内核空间的资源**。这些资源**在切换时都需要被保存和恢复**，以确保新进程能够在切换后顺利执行。

在**发生场景**上，进程上下文切换主要发生在以下几种情况：

一是**进程主动调用某些系统调用时**，如**等待IO完成**或者**获取锁**，这时进程无法继续运行，**操作系统会触发上下文切换**；

二是**进程分配到的时间片用完，或者有更高优先级的进程需要抢占CPU时**，也会发生上下文切换。

接下来是**线程**上下文切换。在**切换内容**方面，线程上下文切换主要涉及线程在执行过程中的运行条件和状态，**如程序计数器、栈信息、寄存器的值等**。由于线程**共享进程的内存空间，**因此线程上下文切换不需要像**进程上下文切换那样涉及大量的内存和资源管理。**

线程上下文切换**发生场景**主要包括：

**线程主动让出CPU，**例如调用了Thread.sleep()或Object.wait()等方法；当一个线程的时间片用完，需要切换到另一个线程继续执行；

或者**线程因为阻塞或等待某个事件而无法继续执行时**，调度器会**切换到另一个线程继续执行。**

**讲一下内联函数和宏定义：**

**​ 在函数名前添加关键字inline，该函数就被声明为内联函数。每当程序中出现对该函数的调用时，C++编译器会将函数体中的代码直接插入到调用处，同时使用实参代替形参，从而使程序运行时不再进行函数调用。**

**​ 引入内联函数主要是为了消除调用函数时的系统开销，以提高代码运行速度。**

* **内联函数在第一次被调用之前必须进行完整的定义**
* **在内联函数体内一般不能含有复杂的控制语句，如for语句和switch语句等**
* **类中的成员函数默认是内联函数（要求成员函数没有循环和递归，当其中存在循环的递归的时候，编译器会将其默认为一个普通函数处理）**
* **使用内联函数是一种空间换时间的措施，若内联函数较长，较复杂且调用较为频繁时不建议使用**
* **使用内联函数替代宏定义，能消除宏定义的不安全性（宏定义不检查函数参数，返回值**

**宏定义（Macro Definition）**

* **#define命令是C++中的宏定义，它用来将一个标识符定义为一个字符串，该标识符被称为宏名，被定义的字符串称为替换文本。**
* **宏函数，也就是带参数的宏定义。**

**不过宏函数有个缺点，宏是原地替换的，所以可能遇到运算符优先级的不同而带来错误。最好每个变量都加上括号。**

#define DOUBLE(x) 2 \* x

**DOUBLE(1 + 1)**在预处理时被替换为：2 \* x + 1，自然结果就不是我们所想的。所以，宏函数中的参数都尽量**带上括号**

**三、内联函数与宏定义的区别**

**内联函数和宏定义都可以用来实现代码的快速执行，但它们有以下几点区别：**

* **参数类型安全性**： 内联函数会对参数进行类型检查，而宏定义不会。内联函数比宏定义更加类型安全。
* **编译器优化：** 内联函数是在编译期间展开的，因此它可以进行更多的编译器优化。而宏定义则是在预处理器展开，不能进行编译器优化。因此，使用内联函数通常可以获得更好的性能。
* 宏和内联函数的展开还是有区别的，**宏函数定义时每个参量都最好加上括号，而内联函数本质上还是函数并没有这么多的要求**。
* **调试：** 内联函数比宏定义更容易进行调试。**而宏定义则无法通过调试器进行调试。**因为内联函数是实际函数的一份副本，可以通过调试器跟踪到内联函数的执行过程。
* **名称空间：** 内联函数位于名称空间中，而宏定义不属于任何名称空间。这意味着，**内联函数可以避免名称冲突问题，而宏定义可能会导致名称冲突。**
* **大小和可读性： 内联函数比宏定义更易于阅读和维护**。宏定义的代码通常比较冗长，而内联函数则可以使用常规的C++语法编写，更加简洁易懂。另外，内联函数可以利用C++的函数重载和模板等特性，提高代码的可读性和可维护性。
* **内联函数也有一定的局限性**。**就是函数中的执行代码不能太多了，不能存在任何形式的循环语句。不能存在过多的条件判断语句，在第一次被调用之前必须进行完整的定义。**如果，内联函数的函数体过大，一般的编译器会放弃内联方式，而采用普通的方式调用函数。这样，内联函数就和普通函数执行效率一样。

**两者相对于普通函数都有提高效率的作用，在不同场景使用根据需求使用宏或者内联，两者各有千秋。**

**LRU算法的核心结构是什么？**

其中最常见的是使用一个**双向链表**和一个**哈希表**。**双向链表**用于维护缓存数据的访问顺序，**哈希表**用于快速查找缓存数据。具体来说，当新的数据被访问时，先在**哈希表**中查找该数据是否已经存在于缓存中，如果存在，则将该数据移动到**双向链表**的头部，表示该数据是最近访问的数据；如果不存在，则需要将该数据添加到缓存中，并将其添加到双向链表的头部。当**缓存空间已满时，需要淘汰双向链表中最后一个节点，同时在哈希表中删除对应的缓存数据**

**代码判断是16位还是32位系统**

**判断指针大小**，sizeof(void\*) 返回指针的大小（以字节为单位）。如果指针的大小为2字节，则表示系统是16位系统；如果指针的大小为4字节，则表示系统是32位系统；否则表示系统是64位或更大位数的系统。

**说下七层网络模型：**

每⼀层负责的职能都不同，如下：

* 应⽤层，**⽹络服务与最终⽤户的⼀个接⼝**。**这⼀层为⽤户的应⽤程序提供⽹络服务**；常⻅的协议有：FTP、SMTP、HTTP、DNS。
* 表示层，**负责把数据转换成兼容另⼀个系统能识别的格式**；
* 会话层，**负责建⽴、管理和终⽌多个表示层实体之间的会话连接；**
* 传输层，**负责端到端的数据传输服务**； TCP（传输控制协议）和UDP（⽤户数据报协议）来管理数据传输
* ⽹络层，负责⽹络包的**封装、分⽚、路由、转发**；**传输单位是数据报**。常⻅的协议有ICMP、ARP、IP
* 数据链路层，负责数据的封帧和差错检测，以及 MAC 寻址； 传输单位 是桢
* 物理层：负责数据**物理传输媒介的传输**，例如电缆、光纤或⽆线信号。这⼀层的 **数据叫做⽐特**

**说下TCP/IP网络模型和具体例子：**

* **应⽤层**，**⽹络服务与最终⽤户的⼀个接⼝**。**这⼀层为⽤户的应⽤程序提供⽹络服务**；，⽐如 HTTP、DNS、FTP 等;
* **传输层**，**负责端到端的通信**； TCP（传输控制协议）和UDP（⽤户数据报协议）来管理。
* **⽹络层**，负责⽹络包的**封装、分⽚、路由、转发**，常⻅的协议有ICMP、ARP、IP
* **⽹络接⼝层**，**负责⽹络包在物理⽹络中的传输，⽐如⽹络包的封帧、 MAC 寻址、差错检测，以及通 过⽹卡将⽹络帧转换为电信号发送出去等**；

**具体例子**

假设你正在浏览网页，通过浏览器发送HTTP请求到服务器。

在发送之前，HTTP 报文会被传递给传输层，传输层，给HTTP报文加上TCP头形成TCP报文。TCP 报文然后被传递到网络层，网络层添加 IP 头信息。在网络接口层，网络包被封装成帧，并添加了 MAC 头。最后，通过网卡将帧转换成比特流，并通过物理介质发送到服务器。服务器收到数据包后，会按照相反的顺序处理数据包，直到 HTTP 请求被正确地解析并返回响应。

**之后的操作：交换机和路由器**

**网卡出来后通过交换机转发到达路由器**

**交换机的端⼝不核对接收⽅ MAC 地址，⽽是直接接收所有的包并存放**

**到缓冲区中。因此，和⽹卡不同，交换机的端⼝不具有 MAC 地址**

**交换机根据 MAC 地址表查找 MAC 地址，然后将数据发送到相应的端⼝。**

**区别：**

**因为路由器是基于 IP 设计的，俗称三层⽹络设备，路由器的各个端⼝都具有 MAC 地址和 IP 地址；**

**⽽交换机是基于以太⽹设计的，俗称⼆层⽹络设备，交换机的端⼝不具有 MAC 地址。**

**说下路由器的工作原理**

**接收阶段：**

**路由器接收到电信号并将其转换成数字信号，然后进行错误校验。**

**如果校验无误，则检查数据包的目标 MAC 地址，如果与路由器端口的 MAC 地址匹配，则将数据包放入接收缓冲区；否则丢弃该包。**

**路由选择：**

**路由器查询路由表，根据目标 IP 地址确定转发目标，也就是下一个路由器的IP。**

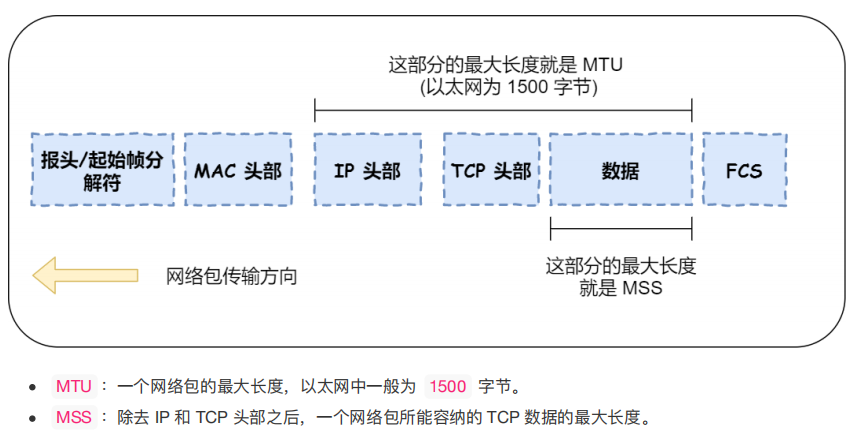
**如果找不到匹配路由，则选择默认路由。**

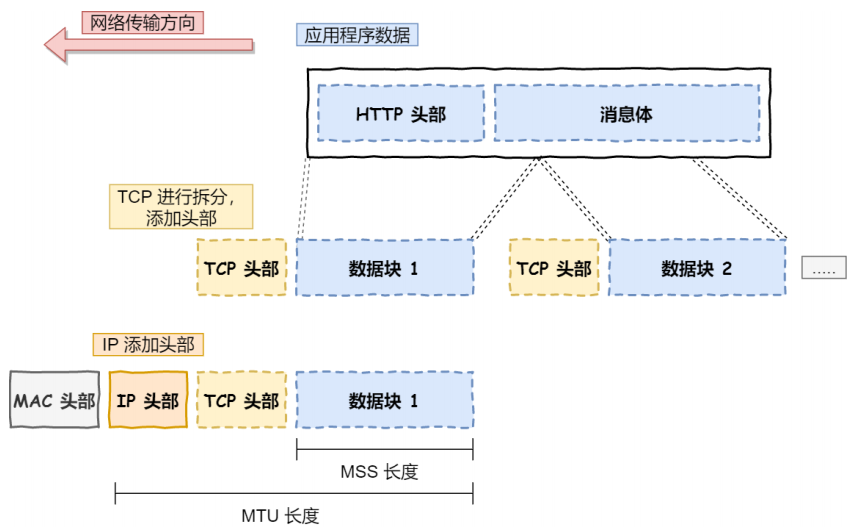
**发送阶段：**

**根据要转发的下个目标地址（IP 地址或网关地址），使用 ARP 协议查询对应的 MAC 地址。并将数据包发送过去，发送出去的⽹络包会通过交换机到达下⼀个路由器。**

**接下来，下⼀个路由器会将包转发给再下⼀个路由器，经过层层转发之后，⽹络包就到达了最终的⽬的地。**

**说下TCP和IP的分片：**





如果 HTTP 请求消息⽐较⻓，超过了 **MSS 的⻓度**（ MSS

（最⼤报⽂段⻓度）），**这时 TCP 就需要把 HTTP 的数据拆解成⼀块块的数据，给每块数据加上TCP头部发送，**⽽ 不是⼀次性发送所有数据。

⽹络层最常使⽤的是 IP 协议（Internet Protocol），**IP 协议会将传输层的报⽂作为数据部分，再加上 IP 包头组装成 IP 报⽂，如果 IP 报⽂⼤⼩超过 MTU（以太⽹中⼀般为 1500 字节）就会对数据部分进⾏分⽚，对每一个分片的数据加上IP头，得到多个 IP 报⽂**。

4520字节的IP报文进行分片



**说下IP协议**

**IP协议是网络层负责数据传输的协议**

**主要作用是：**

**分片：**IP 协议会将传输层的报⽂作为数据部分，再加上 IP 包头组装成 IP 报⽂，如果 IP 报⽂⼤⼩超过 MTU（以太⽹中⼀般为 1500 字节）就会进⾏分⽚，对每一个分片的数据加上IP头，得到多个 IP 报⽂。

**设备编号与网络划分：通过IP地址对设备进行编号，作为其在网络中的唯一标识信息。IPv4协议使用32位地址，分为网络号和主机号。利用子网掩码进行计算，帮助确定设备所属的子网。**

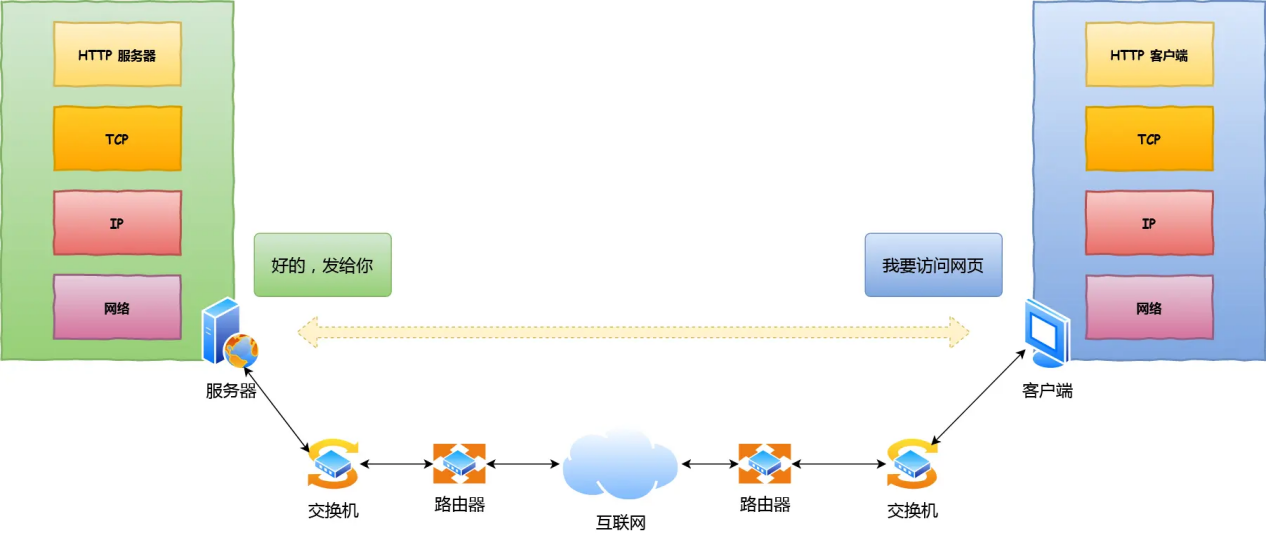
**寻址和路由：根据IP地址进行寻址，根据路由表和路由算法确定数据包的转发路径，使得数据包能够在网络中正确地传输和转发，最终到达目标设备。**

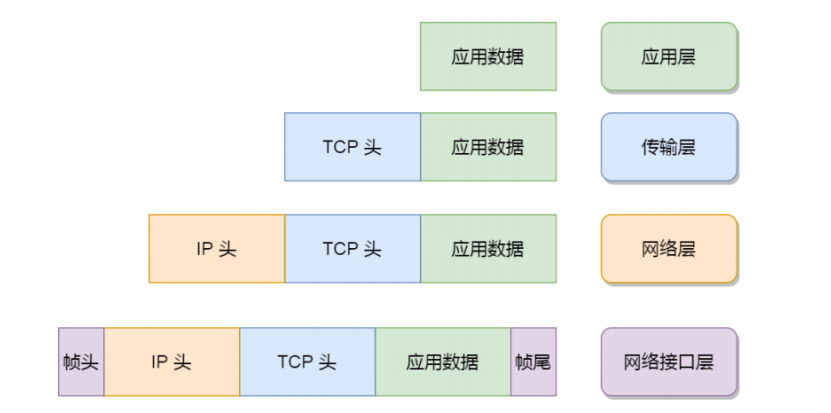
**0结尾的是网络地址，255结尾的是广播地址。**

**注意：**

**⼀个是⽹络号，负责标识该 IP 地址是属于哪个「⼦⽹」的；**

**⼀个是主机号，负责标识同⼀「⼦⽹」下的不同主机；**





TCP与UDP的区别

1、**连接**

**TCP是⾯向连接的，在传输前需要三次握⼿建⽴连接,，四次挥手断开连接。**

**UDP不需要连接，直接发送数据包，没有连接建⽴和关闭的过程。**

**2、可靠性与有序性**

TCP保证数据可靠交付，拥有序列号、确认应答和重传机制，确保数据的有序性和完整性

UDP尽可能交付，没有上述机制，不保证可靠性 ，也**不保证数据包的有序性，**可能出现乱序和丢包

3**、流量控制和拥塞控制**

TCP拥有流量控制、拥塞控制机制，确保数据发送的速率不会超过接收⽅的处理能⼒，并能**防⽌⽹络拥塞。**

UDP不进⾏流量控制和拥塞控制，数据发送的速率不受限制。

**4、传输⽅式**

TCP基于字节流，没有数据边界。

UDP，基于数据包，有数据边界。

**5.⾸部开销**

**TCP的⾸部⼤⼩通常为20字节**，但在选项字段被使⽤的情况下，可能会更⼤。TCP⾸部包含源端⼝号、⽬标端 ⼝号、序列号、确认号、窗⼝⼤⼩、校验和等字段。 UDP的⾸部⼤⼩固定为8字节。UDP⾸部包含源端⼝号、⽬标端⼝号、包⻓度和校验和字段（各16位）。

6.**适⽤场景：**

TCP适⽤于那些对**数据传输可靠性要求较⾼的应⽤**，如**⽂件传输、电⼦邮件、⽹⻚浏览**等。

UDP**传输速度较快**，适⽤于**实时传输的场景，如实时游戏直播，流媒体**等，

7、分⽚⽅式 **TCP数据⼤于MSS时会在TCP层将数据进⾏分⽚传输，到达⽬的地后同样在传输层进⾏合**，如果有某个⽚丢失则 只需要重传丢失的分⽚即可; **UDP数据⼤于MTU时会在IP层分⽚，则会在IP层合并，如果某个IP分⽚丢失，⽬标主机收到后，在 IP 层组装完数据，接着再传给传输层。**

1. HTTP对⽂件分段是怎么设计的？ 14. HTTP⻓连接了解吗，是从哪个版本开始采⽤⻓连接的？ 15. HTTP的GET和POST请求的区别是什么？ 16. Linux中查找⽂件、查看系统CPU性能的命令分别是什么？ 17. C++的运⾏时多态是怎样的？ 18. 内存泄漏如何检测和防⽌？ 19. 智能指针的实现原理是什么？

**HTTP对⽂件分段是怎么设计的？**

HTTP对文件分段设计通常采用的是HTTP分块传输编码（Chunked Transfer Encoding）。这种方式允许服务器将文件切分成若干个块（chunks），每个块都会附带一个大小信息，然后分段传输给客户端。客户端接收到每个块后会立即处理，而不需要等待整个文件传输完成。

**HTTP⻓连接了解吗，是从哪个版本开始采⽤⻓连接的？**

**HTTP长连接是指在一个TCP连接上可以传输多个HTTP请求和响应。它可以减少TCP连接的建立和关闭的开销，提高传输效率。长连接最早出现在HTTP/1.1版本中，通过在请求头中添加Connection: keep-alive字段来告知服务器保持连接。**

**HTTP的GET和POST请求的区别是什么**

* GET请求将参数写在URL上，而POST请求将参数放在请求体中。
* GET请求的**参数长度有限制**，**因为URL的长度有限**，而POST请求则没有这个限制。
* GET请求会**被浏览器缓存，可以被书签保存，而POST请求不会**。
* GET 从服务端**获取资源**。 P**OST 向服务器提交数据.**
* **GET ⽅法就是安全且幂等的，⽆论操作多少次，服务器上的数据都是安全的且每次的结果都是相同的。POST 不安全幂等的的，会修改服务器上的数据，且多次提交数据就会创建多个资源。**
* GET 请求只能进⾏ URL 编码，POST ⽀持多种编码⽅式
* GET 产⽣⼀个 TCP 数据包；POST 产⽣两个 TCP 数据包。对于 GET ⽅式的请求，浏览器会把 header 和 data ⼀并发送出去，服务器响应 200（返回数据）；⽽对于 POST，浏览器先发送 Header，服务器响应 100 continue，浏览器再发送 data，服务器响应 200 ok（返回数据）

**Linux中查找⽂件、查看系统CPU性能的命令分别是什么？**

在Linux中，查找**文件的命令是find**，**查看系统CPU性能的命令是top或htop**。

**内存泄漏如何检测？**

**检测内存泄漏可以通过工具如Valgrind进行，它可以跟踪程序的内存分配和释放，检测出未释放的内存。**

**全局变量和静态变量的区别：**

在C/C++中，static变量和全局变量的主要区别在于作用域和生命周期。

**全局变量的作用域是整个整个程序，生命周期从程序启动到结束**；而static变量的**作用域限制在定义它的函数内部或文件内部**，但它们的**生命周期与全局变量相同**。

在内存上，全局变量和static变量**都存储在静态存储区**，但全局变量和全局**静态变量和类的静态成员变量**在main函数执行前被分配内存初始化，**而局部static变量在**首次调用函数时进行初始化，且只会初始化一次。

**全局变量和static全局变量区别？**

**非静态全局变量的作用域是整个源程序， 当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。 而静态全局变量则限制了其作用域， 即只在定义该变量的源文件内有效， 在同一源程序的其它源文件中不能使用它。**

**三次握手**

**三次握手过程详细说下：**

一开始，客户端和服务端都处于CLOSE状态。

**当服务端准备好接受连接时，它会进入LISTEN状态，等待客户端的连接请求。**

**第一次握手时，客户端会发送一个SYN请求连接报文给服务端，这个报文中包含了客户端的初始序列号。之后，客户端等待服务端的确认。**

服务端收到SYN请求后，会进行第二次握手。**服务端发送一个SYN报文和ACK报文，其中SYN报文包含服务端序列号，ACK报文包含客户端序列号+1**。此时服务端的状态变为SYN-RCVD，等待客户端的确认

**客户端收到服务端的SYN+ACK报文后，会进行第三次握手。**客户端**将服务端的序列号加1放入确认应答报文ACK发送给服务端。此时，客户端的状态变为ESTABLISHED，**表示连接已建立。服务端收到客户端的**ACK报文后，状态也变为ESTABLISHED。**

这个三次握手的过程**确保**了**双方的发送与接收能力是正常的，可以建立可靠的通信连接。**

**为什么需要三次握手？**

三次握⼿才可以**阻⽌重复历史连接的初始化(主因)和避免资源浪费。**

2. 三次握⼿才可以**同步双⽅的初始序列号**

**假设以下场景**

1. 当因为**⽹络阻塞**原因，客户端向服务器发送了**两次SYN报⽂**

2. **旧的SYN报⽂先到达**服务端，服务端回⼀个**ACK+SYN**报⽂

3. 客户端收到后**可以根据⾃身的上下⽂**（序列号），判断这是⼀个历史连接（**序列号过期或超时**），那么客户端就会发送 **RST 报⽂给服务端**，表示中⽌这⼀次连接。

4. 服务器收到RST报⽂，会释放连接

5. **新的SYN报⽂抵达之后，客户端和服务器之间进⾏正常的三次握⼿**

**所以三次握手阻止历史连接的建立**

**但是两次握手情况下，服务端没有中间状态给客户端阻止历史连接，导致服务端可能建立一个历史连接发送数据造成资源的浪费。**

除此之外，由于没有第三次握⼿，服务器**不清楚客户端是否收到了⾃⼰发送的建⽴连接的 ACK 确认信号**，所以**每收到⼀ 个 SYN 就只能先主动建⽴⼀个连接**， 这会建⽴多个冗余的⽆效链接，造成不必要的资源浪费。

「四次握手」：**三次握手就已经理论上最少的保证可靠连接建立，所以不需要使用更多的通信次数。**

**为什么第三次握手可以携带客户到服务器的数据，前两次不行**

如果前两次握手能够携带数据，那么一旦有人想攻击服务器，那么他**只需要在第一次握手中的 SYN 报文中放大量数据，那么服务器势必会消耗更多的时间和内存空间去处理这些数据，增大了服务器被攻击的风险。**

第三次握手，此时客户端已经处于ESTABLISHED状态。

**第三次握手，对于客户端来说，他已经建立起连接了，并且已经知道服务器的接收和发送能力是正常的。所以也就可以携带数据了，**

**三次握手每次的目的是？**

三次握手最根本的目的就是**确认双方的发送与接收能力是否正常，建立可靠的通信连接。**

* **第一次握手**：服务端收到SYN报文，确认客户端的发送能力 和自己的接受能力没有问题
* **第二次握手**：客户端收到SYN+ACK 确认双方的发送能力和接收能力没有问题。但服务器还不知道，所以客户端再发一个ACK过去
* **第三次握手**：服务器收到ACK后也确认了两边的接收能力和发送能力没有问题。
* 这时建立了可靠的连接

**四次挥手：**

### **断开连接过程？（四次挥手）**

**客户端是主动发起断开连接的一方**

* **第一次挥手是由客户端发起的，它会发送一个FIN报文段给服务端。**这个FIN报文段的作用是告诉服务端：“我已经没有数据要发送了，你可以开始关闭你那边到我这边的连接了”。**同时，客户端会进入FIN\_WAIT\_1状态，等待服务端的确认。**
* **接着第二次挥手：在服务端收到FIN报文段后。服务端会发送一个ACK报文段给客户端**，作为对FIN报文段的确认。这个ACK报文段的作用是告诉客户端：“我已经收到你的关闭请求了，我这边会准备关闭连接，但是在我发送完所有的数据之前，连接还不会完全关闭”。此时，**服务端进入CLOSE\_WAIT状态，等待所有数据发送完毕。而客户端在收到ACK后，会进入FIN\_WAIT\_2状态，等待服务端的FIN报文段。**
* **然后是第三次挥手：服务端发送完所有数据后，他会发送一个FIN报文段给客户端**，告知客户端它也没有数据要发送了，可以关闭连接了。这个FIN报文段的作用是告诉客户端：“我这边也没有数据要发送了，我们可以关闭连接了”。**服务端在发送FIN报文段后，会进入LAST\_ACK状态，等待客户端的确认。**
* **最后是第四次挥手**：**是客户端在收到服务端的FIN报文段后会发送一个ACK报文段给服务端**，作为对FIN报文段的确认。这个ACK报文段的作用是告诉服务端：“我已经收到你的关闭请求了，我会关闭连接”。**在发送完ACK报文段后，客户端会进入TIME\_WAIT状态。这个状态会持续一段时间（通常是2MSL**，即数据包在网络中的最大生存时间），**以确保确认ACK报文能够被对方正确接收**。**如果在这段时间内没有收到对方的重发的请求，那么客户端最终会关闭连接，进入CLOSED状态。而服务端在收到ACK报文段后，会立即关闭连接，进入CLOSED状态**。

通过这个过程，TCP连接**能够在确保双方数据传输完成的前提下安全地关闭**。

### 为什么挥⼿需要四次？

TCP连接是全双工的，这意味着**数据可以在两个方向上同时传输**。因此，当一方想**要关闭连接时**，它需要通知另一方，**并确保双方的数据传输都已完**成，然后才能完全关闭连接。所以服务端（被关闭连接的一方）的 **ACK** 和 **FIN** ⼀般都会分开发送

具体来说：

* **客户端主动关闭连接时**，客户端向服务端发送 **FIN** 时，仅仅表示客户端 **不再发送数据** 了，但 **还能接收数据**
* 服务器收到客户端的 **FIN** 报⽂时，先返回 **ACK** 应答报⽂，⽽服务端可能还有数据需要处理和发送，等服务端不再发送数据时，才发送 **FIN** 报⽂给客户端来表示同意现在关闭连接

**主动关闭的一方才有time wait**

**为什么需要time wait：**

如果**最后的⼀次ACK报⽂丢失（第四次挥⼿）**，客户端没有 TIME\_WAIT 状态，直接进⼊ClOSE，服务端⼀直在等待 ACK状态，⼀直没有等到，就会**重发FIN报⽂**，⽽客户端已经进⼊到关闭状态，在收到服务端重传的 FIN 报⽂后， 就会回 RST 报⽂,服务端收到这个 RST 并将其解释为⼀个错误, 为了防⽌这种情况出现，客户端**必须等待⾜够⻓的时 间，确保服务端能够收到 ACK.**

注意：如果在 TIME-WAIT 时间内，因为客户端的 ACK 没有传输到服务端，客户端⼜接收到了服务端**重发的 FIN 报⽂**，那么 **2MSL 时间将重新计时**。

**为什么需要2MSL：**

MSL是 Maximum Segment Lifetime ，报⽂最⼤⽣存时间，它是任何报⽂在⽹络上存在的最⻓时间。

⽹络中可能存在发送⽅的数据包，当这些发送⽅的数据包被接收⽅处理后⼜会向对⽅发送响 应，所以⼀来⼀回需要等待 2 倍的时间。

经过 2MSL 这个时间，**⾜以让两个⽅向上的数据包都被丢弃，使得旧连接的数据包在⽹络中都消失，再出现 的数据包⼀定都是新建⽴连接所产⽣的。**

**TIME\_WAIT 过多有什么危害？**

**过多的TIME-WAIT 状态主要的危害有两种：**

**1. 内存资源占⽤；**

**2. 对端⼝资源的占⽤，⼀个 TCP 连接⾄少消耗⼀个本地端⼝；**

**如果发起连接⼀⽅的 TIME\_WAIT 状态过多，占满了所有端⼝资源，则会导致⽆法创建新连接。**

**Define和const的区别？**

· 作用域不同：

* **#define定义的常量是一个预处理宏，**它在编译之前被替换，作用域为定义处到文件结束。
* **const定义的常量是一个真正的变量，**其作用域根据定义的位置而定，可以是局部或全局作用域。

· 类型安全性：

* **#define不具有类型检查**，在预处理阶段只是**简单地进行文本替换**，容易导致一些潜在的错误。
* **const定义的常量具有类型检查**，**编译器会对其进行类型检查，提供更好的类型**安全性。

· 调试信息：

* #define在预处理阶段进行文本替换，因此在**调试时无法查看使用#define定义的常量的值。**
* const定义的常量是真正的变量，**可以被调试器识别并显示其值**

**长连接和短连接各自适合什么场景？**

### 短连接

在短连接模式下，客户端与服务器之间的每个请求/响应对都会打开一个新的连接，并在完成后立即关闭连接。短连接适用于以下场景：

* **低频请求**：如果客户端只偶尔与服务器通信，使用短连接可以减少服务器维护空闲连接所需的资源。
* **简单的请求/响应交互**：对于简单的请求，短连接可以快速建立，完成数据交换后即刻释放资源。
* **负载分配**：在负载均衡的环境中，短连接有助于将来自不同客户端的请求更公平地分配到不同的服务器。

### 长连接

HTTP/1.1默认采用长连接（持久连接），在此模式下，TCP连接在多个请求/响应之间保持开放状态，直到由客户端或服务器明确关闭。长连接适合于：

* **高频请求**：当客户端需要频繁地向服务器发送请求时，长连接**减少了因为建立和关闭连接而带来的额外开销。**
* **减少延迟**：对于需要快速响应的应用，如网页浏览和在线游戏，长连接能够减少每次请求所需的往返时间（RTT），提高用户体验。
* **实时通信**：在聊天应用、实时数据更新等场景中，长连接可以保持一个持续的数据流，允许服务器主动向客户端推送信息。

**但是连接数量很多的话，会导致服务器资源不足，负担太大**

**移动语义的作用和原理：**

移动语义是 C++11 引入的一个重要特性，用于提高程序的性能和效率。**它允许将资源（如堆内存、文件句柄等）的所有权从一个对象转移到另一个对象，而不是传统的拷贝操作。**

移动语义的核心是通过移动构造函数和移动赋值运算符来实现的。**当使用右值引用（Rvalue reference）时，编译器可以识别并调用移动构造函数或移动赋值运算符，从而避免不必要的内存拷贝操作。**

移动语义对于临时对象的构造和销毁具有重要意义，它能够避免临时对象的拷贝，提高程序的效率。特别是在处理大型数据结构或频繁进行资源管理的情况下，移动语义可以显著减少内存和性能开销。

**说下信号：**

信号是进程间唯一的一种异步通信机制，⽤于通知进程某个事件已经发⽣，从⽽迫使进程执⾏信号处理程序

**信号发送**：**信号可以由操作系统、其他进程或者进程自身发起。**比如，用户在终端上按下某个键组合、操作系统检测到某个硬件错误或者其他事件发生时，会向相应的进程发送信号。

**信号注册**：**接收信号的进程可以注册信号处理函数，用于处理接收到的信号。**可以使用 signal() 或者 sigaction() 等函数注册信号处理函数。

**信号处理：当进程接收到信号后，会调用注册的信号处理函数来处理信号。**信号处理函数可以执行特定的操作，比如打印一条消息、关闭进程、重新启动进程等。**在信号处理函数执行完毕后，进程会继续执行原来的代码。**

**默认处理**：**如果进程没有注册对某个信号的处理函数，系统会采用默认的处理方式。**比如，对于 **SIGINT 信号（通常由 Ctrl+C 触发）**，默认处理是终止进程。

MySQL是一个关系型数据库管理系统，使用SQL 语言对数据库进行管理。

**什么是关系型数据库？**

关系型数据库，是指**采用了关系模型来组织数据的数据库，其以行和列的形式存储数据，为便于用户理解，关系型数据库将这一系列的行和列称为表，一组组表组成了数据库。**

**讲讲cookie和session**

Cookie和Session是用于在**Web应用程序中跟踪用户状态和数据**的两种主要方法。

**都⽤于管理⽤户的状态和身份, Cookie 在客户端记录信息确定⽤户身份**， **Session 在服务器端记录信息确定⽤户身份**。

Cookie

Cookie 是存储在**⽤户浏览器中的⼩型⽂本⽂件，⽤于在⽤户和服务器之间传递数据**。通常，服

**务器会将⼀个或多个 Cookie 发送到⽤户浏览器，然后浏览器将这些 Cookie 存储在本地。**

服务器在**接收到来⾃客户端浏览器的请求之后，就能够通过分析存放于请求头的Cookie得到客户**

**端特有的信息，**从⽽动态⽣成与该客户端相对应的内容。

Session

客户端浏览器访问服务器的时候，**服务器把客户端信息以某种形式记录在服务器上**。这就是 Session 。Session 主要⽤于维护⽤户登录状态、存储⽤户的临时数据和上下⽂信息等。

**存储位置：**Cookie 数据存储在⽤户的浏览器中，⽽ Session 数据存储在服务器上。

**数据容量：**Cookie 存储容量较⼩，⼀般为⼏ KB。Session 存储容量较⼤，通常没有固定

限制，取决于服务器的配置和资源。

**安全性：**由于 Cookie 存储在⽤户浏览器中，因此可以被⽤户读取和篡改。相⽐之下，

Session 数据存储在服务器上，更难被⽤户访问和修改。

**传输⽅式：**Cookie 在每次 HTTP 请求中都会被⾃动发送到服务器，⽽ Session ID 通常通

过 Cookie 或 URL 参数传递。

**Vector和自建数组的区别：**

vector和自建数组在很多方面有所不同，主要区别包括：

**大小可变性：**

vector是**C++标准库提供的动态数组，可以自动扩容和缩容，大小可变。**

自建数组需要指定大小，**并且大小固定，无法自动扩容或缩容**。

**内存管理：**

vector**会自动处理内存的分配和释放，不需要手动管理内存**。

自建数组**需要手动分配内存**，并且需要确保在不使用时正**确释放内存**，否则容易出现**内存泄漏**或访问非法内存的情况。

**便利性：**

**vector提供了丰富的成员函数和操作符重载，**方便对数组进行操作和访问。**Push\_back,pop\_back**

**自建数组需要自行实现对应的功能函数**，代码量可能较多且容易出错。

**性能：**

vector的**性能在大多数情况下都很好**，因为其内部实现通常采用了动态数组和缓冲区分配等高效算法。

自建数组的性能受到实现的影响，如果没有很好地优化和管理，可能性能较差。

**vector如何扩容：**

vector在需要扩容时会**重新分配一块更大的内存空间，并将原有数据拷贝到新的空间中，然后释放原来的内存空间。**一般情况下，vector的扩容策略是以倍增的方式进行，即每次扩容后的容量是原容量的两倍。这种策略可以保证扩容操作的平摊时间复杂度为O(1)。

**说下new/delete和malloc/free:**



而new、delete 是C++的运算符。malloc、free 是**库函数**，

new 申请空间时，无需指定分配空间的大小，编译器会根据类型自行计算；malloc 在申请空间时，需要指定所申请空间的大小。

new 申请空间时，返回的是**对象的指针类型**，无需强制类型转换，是类型安全的；malloc 申请空间时，返回的是 **void\* 类型，需要进行强制类型的转换**，转换为对象类型的指针。

new 分配失败时，**会抛出 bad\_alloc 异常**，malloc 分配失败时返回空指针NULL。

New和delete分配或者释放内存时会**调用构造与析构函数**。而malloc、free 只分配和释放内存。

**delete 释放的内存块的指针值会被设置为 nullptr ，以避免野指针。**

**free 不会修改指针的值，可能导致野指针问题。**

**数组的释放：**

**delete 可以正确释放通过 new[] 分配的数组。**

**free 不了解数组的⼤⼩，不适⽤于释放通过 malloc 分配的数组**

1. delete 用于释放 new 分配的空间；free 用来释放 malloc 分配的空间

2. delete 释放空间的时候会调用 相应的析构函数

     同时，new时调用构造函数，而malloc不会，它只是分配内存。

3. 调用free 之前需要检查 需要释放的指针是否为空，使用delete 释放内存则不需要检查指针是否为NUL

**一个对象先malloc后delete可以么**？

1. 构造函数未被调用：使用 malloc 分配内存时，对象的构造函数不会被执行，导致对象可能处于未初始化的状态。
2. 析构函数问题：如果你使用 malloc 分配而后使用 delete 释放内存，会尝试调用对象的析构函数，但因为构造函数从未被调用过，这可能导致未定义行为，比如资源泄露、崩溃或数据损坏等问题。
3. 兼容性问题：malloc 和 delete 内部实现可能有差异，混用可能导致堆损坏。
4. 异常安全性：由于 malloc 和 delete 的异常处理机制不同（new 可以抛出异常，而 malloc 返回NULL），这会进一步使得代码难以维护，并可能导致错误处理不当。

**指针和引用的区别**

* 指针**存放某个对象的地址**，其本身就是变量（命了名的对象）。本身就有地址，所以可以有指向指针的指针；
* 可变，**可以改变指针的指向，使其指向不同的内存地址，也可以修改变量的值。**
* **存在指向空值的指针。**
* **引⽤就是变量的别名，在已存在的变量上创建的，**
* **不能改变所指的对象，必须声明时初始化**，**整个⽣命周期中**将⼀直引⽤同⼀个对象，不需要新的内存。
* 不存在指向空值的引⽤。

⽤途：

指针： 通常⽤于**动态内存分配**、数组操作以及函数参数传递。

引⽤： 通常⽤于**函数参数传递、操作符重载以及创建别名**。

**说下中断和异常的区别**

中断和异常

**中断和异常都会导致处理器暂停当前正在执⾏的任务，并转向执⾏⼀个特定的处理程序**（中断处理程序或异常处理程序）。然后在处理完这些特殊情况后，处理器会返回到被打断的任务继续执⾏。

中断是由计算机系统外部事件触发的，**通常与硬件设备相关**。中断的⽬的是为了及时响应重要事件⽽暂时中断正常的程序执⾏。

时钟中断、I/O设备中断（如键盘输⼊、⿏标事件）和硬件错误中断

异常是由计算机系统内部事件触发的，通常与正在执⾏的程序或指令有关，

⾮法操作码、地址越 界、运算溢出等错误。

**虚拟内存有什么好处？**

**虚拟内存是一种内存管理技术，通过将实际的物理内存和磁盘空间结合起来，使得每个进程能够拥有一个连续的、私有的地址空间，从而提供了更大的地址空间和更灵活的内存管理方式。**

内存隔离 ：虚拟内存允许每个运⾏的程序都有⾃⼰独⽴的虚拟地址空间，这样不同程序之间的内存不会相互⼲扰。这提供了强⼤的内存隔离，防⽌⼀个程序的错误或崩溃影响其他程序。

更⼤的地址空间 ：虚拟内存可以将物理内存和硬盘上的存储空间组合起来，使得每个程序看起来都拥有⼀个巨 ⼤的地址空间。

**内存映射：**允许将⽂件或其他对象映射到进程的地址空间中，有了这样的映射，进程利用指针直接读写虚拟地址就可以**完成对文件的读写操作，⽽⽆需繁琐的⽂件操作。**

⻚⾯置换 ：能够将不活动的内存⻚交换到硬盘上，从⽽在物理内存不⾜时为活动的程序提供⾜够的空间。这种⻚⾯置换策略有助于维护系统的稳定性和性能。

内存分配的简化 ：虚拟内存允许程序使⽤连续的虚拟地址空间，⽽⽆需关⼼物理内存的碎⽚问题。这简化了内 存分配和管理，减少了程序员的⼯作负担。

**页表：**

⻚表是虚拟内存管理的核⼼。通过⻚表，操作系统可以将虚拟地址（由应⽤程序⽣成）映射到物理地址（实际的内存地址）。，还可以将不活动的⻚交换到磁盘上以释放内存。

。

**内存隔离：**每个**应⽤程序都有⾃⼰的⻚表，因此使它们的虚拟地址空间相互隔离,不会相互⼲扰。**

**内存保护**： ⻚表**允许操作系统设置不同的权限和保护位**，以限制对内存的访问。例如，**只读⻚表项可以防⽌ 写⼊特定的内存区域，从⽽提⾼系统的安全性。**

**三级页表的优劣：**

**优势：**

1. **节省内存空间：**

**相比一级页表，三级页表采用了分级的结构，除了一级页表必须创建，用来覆盖整个虚拟地址空间以外，二三级页表只有在需要时才创建，因此可以根据实际需要动态分配内存，节省了大量的内存空间。特别是在虚拟地址空间较大且实际使用的页表项较少的情况下，能够极大地减少内存占用。**

**2.** **地址映射的快速性：** 三级⻚表中的多级结构可以减少地址映射的时间复杂度，因为它将地址映射分成多个步 骤，每个步骤都相对较⼩和⾼效。这可以提⾼内存访问的速度

劣势：

1.**地址映射的复杂性**： 多级⻚表的实现和管理相对复杂，需要**更多的硬件⽀持和操作系统的代码**。这可能增加 了**系统的复杂性和维护难度**。

2. **访问速度的折衷**： 尽管多级⻚表可以**提⾼地址映射的速度，但它们也会引⼊额外的访问延迟，因为需要多次 查找⻚表。这可能会对性能产⽣⼀定的负⾯影响。**

**数组和链表的区别：**

**数组：**

* 数组是一种线性数据结构，**元素在内存中连续存储**。
* **数组在创建时需要指定大小**。
* 可以**通过索引直接访问数组中的元素，时间复杂度为 O(1)**。
* **插入和删除元素时，需要移动其他元素来保持连续性，时间复杂度为 O(n)。**
* 适合用于**元素个数固定、对随机访问要求较高**的场景。

链表：

* 链表是一种线性数据结构，**元素在内存中的储存不连续，**通过指针将元素串联在一起，。
* **链表的大小可以动态调整**，**不需要预先指定大小**。
* 链表**不能通过索引直接访问元素，需要从头节点开始遍历，时间复杂度为 O(n)**。
* 链表**插入和删除元素的操作简单**，只需改变指针的指向即可，时间复杂度为 O(1)。
* 适合用于**频繁插入和删除操作**、**元素个数变化较大**的场景。

**二叉树和链表的区别**

结构：

* 1. 链表是由**节点顺序连接而成的线性数据结构**，**每个节点包含数据域和指向下一个节点的指针。**
  2. 二叉树是一种**树状结构，每个节点最多有两个子节点，分别为左子节点和右子节点。**

**存储方式：**

* 1. 链表的节点在内存中同城是不连续存储的，通过指针进行连接。
  2. 二叉树的节点在内存中可以采用链式存储（使用指针连接）

**操作效率：**

* 1. 在链表中，插入和删除操作的时间复杂度为 O(1)，查找操作的时间复杂度为 O(n)。
  2. 在二叉树中，对于平衡二叉搜索树，**插入、查找、删除等操作的时间复杂度为 O(log n)。**

应用场景：

* 1. 链表适合**动态管理数据集合，特别是频繁需要插入和删除操作的场景**。
  2. 二叉树适合用于**搜索、排序等需要快速查找的应**用，如二叉搜索树用于快速查找。

**说下各种二叉树：**

1.**二叉搜索树：**

二叉搜索树是一种特殊的二叉树，对于**每个节点，其左子树中的所有节点值都小于该节点的值，右子树中的所有节点值都大于该节点的值。**

**二叉搜索树的中序遍历结果是有序的。**

2.**平衡二叉树：**

* 平衡二叉树是一种**特殊的二叉搜索树，它保持左右子树的高度差不超过1，**以确保树的高度平衡。
* 平衡二叉树的**插入和删除操作会导致树的自平衡调整**，以保持平衡性。

3.**完全二叉树：**

* 完全二叉树是一个二叉树，**除了最后一层外，其他层都是满的，且最后一层的节点从左向右依次排列。**
* 完全二叉树通常使用数组来存储，可以利用数组索引计算节点之间的关系。

4.满二叉树：

* 满二叉树是一种特殊的二叉树，**每个节点要么没有子节点，要么有两个子节点。**
* 满二叉树的**叶子节点都在同一层**，**且所有非叶子节点都有两个子节点。**
* **k层，2k-1个节点。**

**总结：**

* 二叉树是具有两个子节点的树状结构。
* 二叉搜索树是一种特殊的二叉树，左子树节点值小于根节点，右子树节点值大于根节点。
* 平衡二叉树是一种保持平衡性的二叉搜索树。
* 完全二叉树是除了最后一层外都是满的二叉树。
* 满二叉树是每个节点要么没有子节点，要么有两个子节点的二叉树。

**栈和队列的区别：**

栈：

* 栈是一种**后进先出**（LIFO，Last In First Out）的数据结构，类似于一摞盘子，只能在栈顶进行操作。
* **栈的插入（压栈）和删除（弹栈）操作都发生在栈顶元素**，时间复杂度为 O(1)。
* 栈通常用于**实现函数调用、表达式求值、括号匹配**等场景。
* 常见的栈包括数组实现的顺序栈和链表实现的链式栈。

队列：

* 队列是一种**先进先出**（FIFO，First In First Out）的**数据结构**，类似于排队等候的过程，新元素在队尾插入，从队头删除。
* 队列的**插入（入队）和删除（出队）操作分别在队尾和队头进行**，时间复杂度为 O(1)。
* 队列通常用于**任务调度、缓冲区管理、消息传递**等场景。
* 常见的队列包括数组实现的顺序队列和循环队列，以及链表实现的链式队列。

**适用场景：**

* 栈适合用于需要后进先出的场景，如**函数调用**、逆波兰表达式求值等。
* 队列适合用于需要先进先出的场景，如**任务调度、消息传递**等。
* 在实际应用中，根据具体需求选择合适的数据结构可以提高程序效率和简化算法设计。

**Keepalive:**

**HTTP 的 Keep-Alive，是由应⽤层（⽤户态） 实现的，称为 HTTP ⻓连接；**

每次请求都要经历这样的过程：建⽴ TCP -> 请求资源 -> 响应资源 -> 释放连接，这就是HTTP短

连接，但是这样每次建⽴连接都只能请求⼀次资源，所以HTTP 的 Keep-Alive实现了使⽤同⼀个

TCP 连接来发送和接收多个 HTTP 请求/应答，**避免了连接建⽴和释放的开销**，就就是 HTTP ⻓连

接。

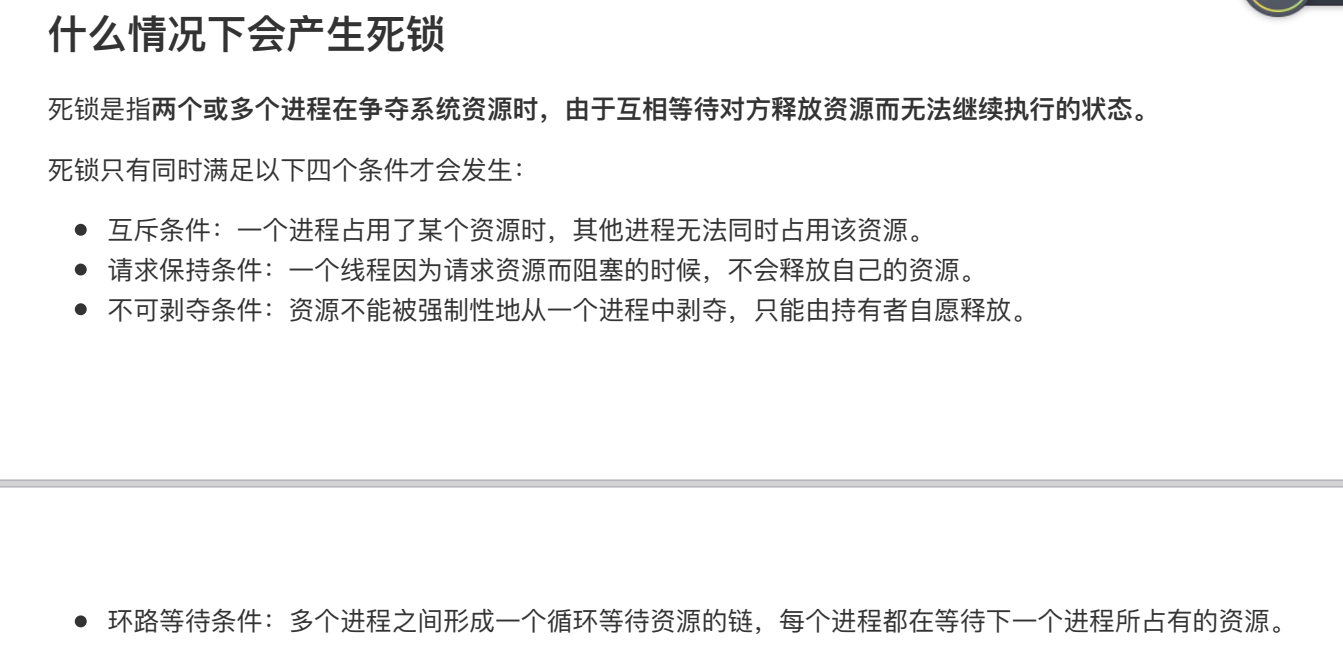
**TCP 的 Keepalive，是由 TCP 层（内核态） 实现的，**称为 TCP 保活机制；

通俗地说，**就是TCP有⼀个定时任务做倒计时，超时后会触发任务，内容是发送⼀个探测报⽂给对**

**端，⽤来判断对端是否存活。**

在很多情况下，默认的TCP连接没有开启Keepalive，因为频繁地发送Keepalive探测包可能会增加不必要的网络流量。但是，在长时间打开的连接中（例如数据库连接），可能需要使用**Keepalive来确保持续的服务可用性。**

**死锁：**



破坏请求保持条件

1. 进程运行前前请求分配好所需要的全部资源。

破坏不剥夺条件

1.允许进程抢占资源。

2. 要求当⼀个进程申请资源得不到满足时，释放所拥有的资源

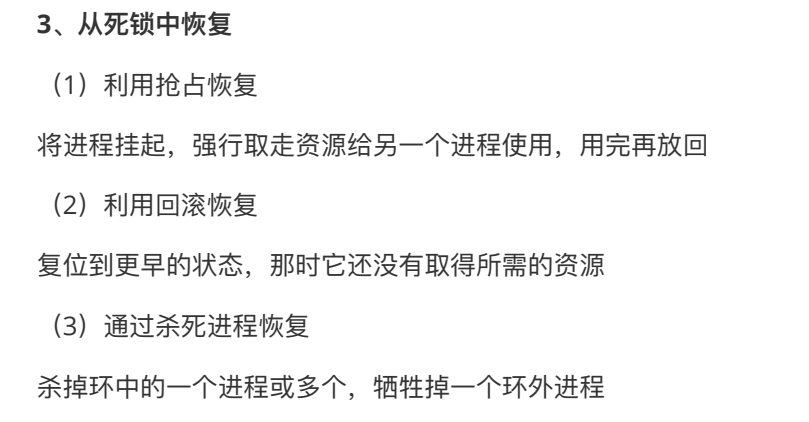
破坏环路等待

给资源统⼀编号，进程只能按编号顺序来请求资源

死锁检测和恢复:

死锁的检测和恢复技术是指定期启动一个软件检测系统的状态。若发现有死锁存在，则采取措施恢复之。

死锁的检测:检查死锁的办法就是由软件检查系统中由进程和资源构成的有向图是否构成一个或多个环路，若是，则存在死锁，否则不存在。



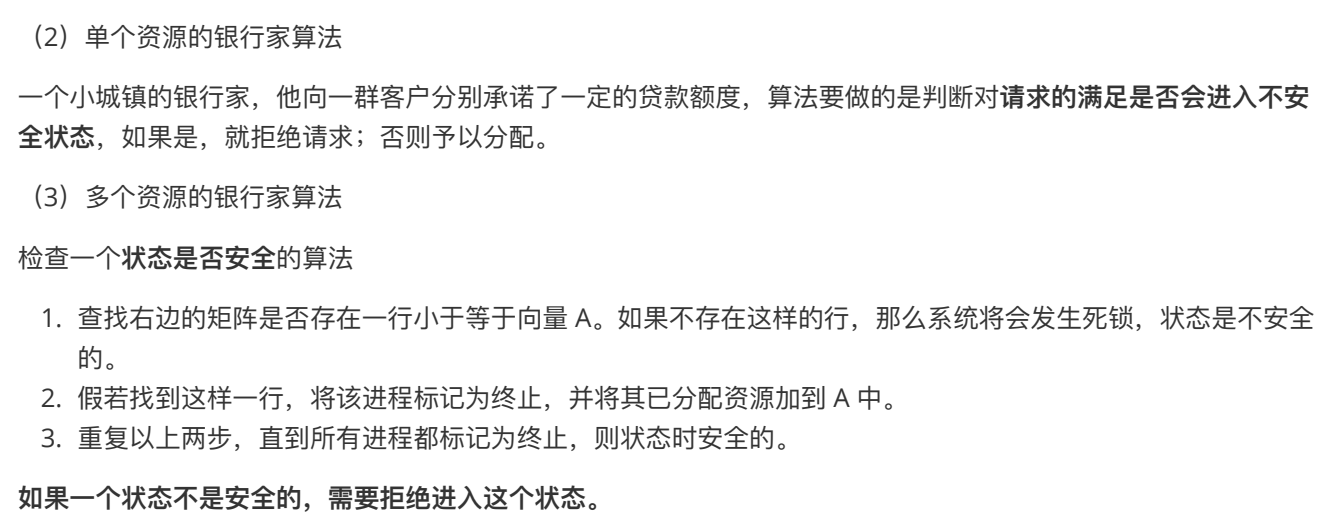
**死锁的避免（预防死锁）**

安全序列：如果系统安装这种序列分配资源，则每个进程都能顺利完成。只要找出一个安全序列，系统就是安全状态。当然，安全序列可能有多个。

如果分配了资源之后，系统中找不到任何一个安全序列，系统就进入了不安全状态。这就意味着之后可能所有进程都无法顺利的执行下去。当然，如果有进程提前归还一些资源，系统也有可能重新回到安全状态，不过我们在分配资源之前总是考虑到最坏的情况。

如果系统处于安全状态，就一定不会发生死锁。如果系统进入不安全状态，系统未必发生死锁。但发生死锁一定是在不安全状态。

因此可以在资源分配之前预先判断这次分配是否导致系统进入不安全状态，以此决定是否答应资源分配请求，“银行家算法”的核心思想。



1. **⽤户态和内核态的区别：**

⽤户态 （User Mode） 和内核态 （Kernel Mode） 是**操作系统**为了保护系统资源和实现权限控制⽽设计的**两种不同的CPU运⾏级别**，可以控制进程或程序**对计算机硬件资源的访问权限和操作范围。**

⽤户态：在⽤户态下，进程或程序只能访问**受限**的资源和执⾏受限的指令集，**不能直接访问操作系统的核⼼部分**，也**不能直接访问硬件资源**。

内核态：

内核态是操作系统的**特权级别**，允许进程或程序**执⾏特权指令和访问操作系统的核⼼部分**。在核⼼态下，进程可以直接访问硬件资源，执⾏系统调⽤，管理内存、⽂件系统等操作。

1. **在什么场景下，会发⽣内核态和⽤户态的切换**

* **系统调⽤：**当⽤户程序需要请求操作系统提供的服务时，会通过系统调⽤进⼊内核态。
* **异常**：当程序执⾏过程中出现错误或异常情况时，CPU会⾃动切换到内核态，以便操作系统能够处理这些异 常。
* **中断**：外部设备（如键盘、⿏标、磁盘等）产⽣的中断信号会使CPU从⽤户态切换到内核态。操作系统会处理 这些中断，执⾏相应的中断处理程序，然后再将CPU切换回⽤户态。

### ****64位编译器****

char ：1个字节  
char\*(即指针变量): 8个字节  
short int : 2个字节  
int：  4个字节  
unsigned int : 4个字节  
float:  4个字节  
double:   8个字节  
long:   8个字节  
long long:  8个字节  
unsigned long:  8个字节

# 函数参数的入栈顺序是什么，从左到右还是从右到左

在C++中，函数参数的入栈顺序是从右到左的。也就是说，最后一个参数先入栈，第一个参数最后入栈。

幂等：多次重复操作和一次操作产生的影响是一样的；  
非幂等：多次重复操作和一次性操作产生的影响是不一样的；

**说一下hello world经历的过程：**

**预处理:预编译过程主要处理那些源代码文件中的以"#"开始的预编译指令，比如“#include" ,“#define"等。(.i**

**编译:编译过程就是把预处理完的文件进行一系列词法分析，语法分析，语义分析，代码优化后生成相应的汇编代码文件。**

**汇编:汇编过程就是由汇编器将汇编代码转变成机器可以执行的二进制指令，生成多个目标文件。(.o)**

**链接: .链接器将所有的目标文件和必要的库文件链接在一起，经过符号解析和重定位，生成最终的可执行文件。**

**程序运行: 最终生成的可执行文件被加载到内存中，并由操作系统负责执行。（exe程序->虚拟内存->物理内存。）**

**多线程的问题：**

**多线程的编程和调试比较复杂，一个线程挂掉可能会导致整个进程挂掉，死锁问题，线程同步问题。**

**从输⼊url到⻚⾯加载完成发⽣了什么？**

**简单版：**

**1.浏览器接收到⽤户请求，先检查浏览器缓存⾥是否有缓存该资源，如果有直接返回；如果没有进⼊下⼀步⽹络请求。**

**2. ⽹络请求前，进⾏ DNS解析 ，以获取请求域名的 IP地址 。**

**3. 浏览器根据IP建⽴TCP连接（三次握⼿）。连接建⽴后，浏览器构建http请求并发送服务端。**

**4. 服务器接收到处理请求，发送响应给浏览器。**

**5. 浏览器解析响应，渲染页面，构建DOM树。**

**6.关闭TCP连接（四次挥⼿）。**

**复杂版：**

1.浏览器接收到⽤户请求，先检查浏览器缓存⾥是否有缓存该资源，如果有直接返回；如果没有进⼊下⼀

步⽹络请求。

2. ⽹络请求前，进⾏ DNS解析 ，以获取请求域名的 IP地址 。如果请求协议是 HTTPS ，那么还需要

建⽴TLS连接 。DNS解析时会按本地浏览器缓存->本地 Host ⽂件->路由器缓存-> DNS 服务器->

根 DNS 服务器的顺序查询域名对应 IP ，直到找到为⽌。

3. 浏览器与服务器IP建⽴TCP连接。连接建⽴后，浏览器端会构建请求⾏、请求头等信息，并把和该域名

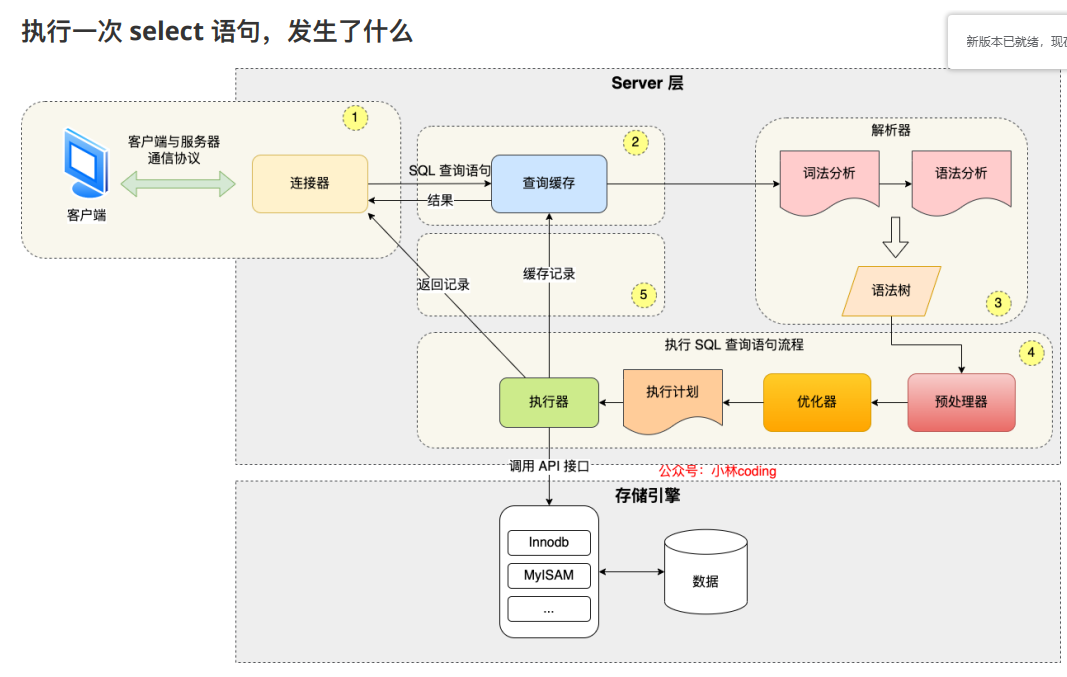
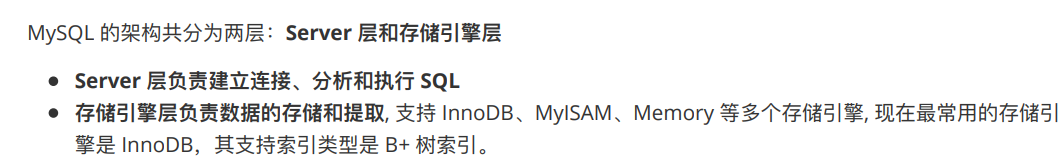
相关的 Cookie 等数据附加到请求头中，向服务器构建请求信息。

4. 服务器接收到请求信息，根据请求⽣成响应数据。

5. 浏览器解析响应头。若响应头状态码为 301、302 ，会重定向到新地址；若响应数据类型是字节流类

型，⼀般会将请求提交给下载管理器；若是HTML类型，会进⼊下⼀部渲染流程。

6. 浏览器解析 HTML ⽂件，创建 DOM 树，解析 CSS 进⾏样式计算，然后将CSS和DOM合并，构建渲染树；最后布局和绘制渲染树，完成⻚⾯展示。





**说下你知道（用过的）SQL语句：**

SELECT：从数据库中查询数据。

SELECT \* FROM table\_name;

INSERT：向数据库中的表插入数据。

INSERT INTO table\_name (column1, column2) VALUES (value1, value2);

UPDATE：更新数据库中表的数据。

UPDATE table\_name SET column1 = value1 WHERE condition;

DELETE：从数据库中的表删除数据。

DELETE FROM table\_name WHERE condition;

CREATE TABLE：创建新的表。

CREATE TABLE table\_name (column1 datatype, column2 datatype);

ALTER TABLE：修改表结构，比如添加或删除列。

ALTER TABLE table\_name ADD column\_name datatype;

DROP TABLE：删除表。

DROP TABLE table\_name;

JOIN：从多个表中查询数据。

数据库数据量增加，怎么优化性能？

（1）软优化：

1. **数据库参数调优**

2. 分析慢查询SQL语句，分析执⾏计划，进⾏**sql改写和程序改写**

3. **优化数据库索引结构**

4. **优化数据表结构**

（2）硬优化

**提升系统硬件（更快的IO、更多的内存）：带宽、CPU、硬盘**

**（3）分库分表：**

1、单库问题 **单个数据库处理能⼒有限，所在的服务器上的磁盘空间也有限，单库存在I/O操作瓶颈。** 主要⽅案：切分成更多更⼩的库

2、单表太⼤ CRUD都成问题，**索引膨胀，查询超时**。 主要⽅案：切分成多个数据集更⼩的表

**垂直分表**

**将⼀个表字段拆分多个表，每个表存储部分字段**

**拆分**原则： 1. 把不常⽤的字段单独放在⼀张表; 2. 把text，blob等⼤字段拆分出来放在附表中; 3. 业务经常组合查询的列放在⼀张表中

**好处：** **避免IO时锁表的次数**，**分离热点字段和⾮热点字段，避免⼤字段IO导致性能下**降.

但是这 些表还是在**同⼀个库中，**

垂直分库

**针对的是⼀个系统中的不同业务进⾏拆分，根据业务将表分类，放到不同的数据库服务器上。**

**好处：**避免表之间竞争同个物理机的资源，**⽐如CPU/内存/硬盘/⽹络IO**

垂直分库可以更好**解决业务层⾯的耦合，业务清晰，且⽅便管理和维护**

**原则：** 根据业务相关性进⾏划分，领域模型，微服务划分⼀般就是垂直分库

**问题：依然没有解决单表数据量过⼤的问题**

**水平分表：**核⼼是把⼀个⼤表，分割N个⼩表， 每个表的结构是⼀样的，数据不⼀样，全部表的数据合起来就是全部数据。

**单个表的数据量少了，业务SQL执⾏效率⾼，降低了系统的IO和CPU压⼒**

但是这 些表还是在**同⼀个库中，所以单数据库操作还是有IO瓶颈**，主要是解决单表数据量过⼤的问题。

**⽔平分库**

把**同个表的数据按照⼀定规则分到不同的数据库中**，数据库在不同的服务器上。它是**对数据⾏的拆分，不影响表结构，**每个库的结构都⼀样,**但每个库的数 据都不⼀样，没有交集，所有库的并集就是全量数据**。

**好处：多个数据库，降低了系统的IO和CPU压⼒。**

**常⻅的问题**

1、**跨节点数据库Join关联查询**，

数据库切分前：多表关联查询，可以通过sql join进⾏实现

**分库分表后： 数据可能分布在不同的节点上，sql join带来的问题就⽐较麻烦**。

2、**分库操作带来的分布式事务问题**

操作内容同时分布在不同库中，**不可避免会带来跨库事务问题，即分布式事务**

3、执⾏的SQL排序、翻⻚、函数计算问题

分库后： **数据分布再不同的节点上， 跨节点多库进⾏查询时，会出现limit分⻚、order by排序等问题** ⽽且当排序字段⾮分⽚字段时，更加复杂了，要在不同的分⽚节点中将数据进⾏排序并返回，然后将不同分⽚返回 的结果集进⾏汇总和再次排序（也会带来更多的CPU/IO资源损耗）

4、数据库全局主键重复问题 **常规表的id是使⽤⾃增id进⾏实现，分库分表后，由于表中数据同时存在不同数据库中，如果⽤⾃增id，则会出现 冲突问题**

5、容量规划,分库分表后⼆次扩容问题

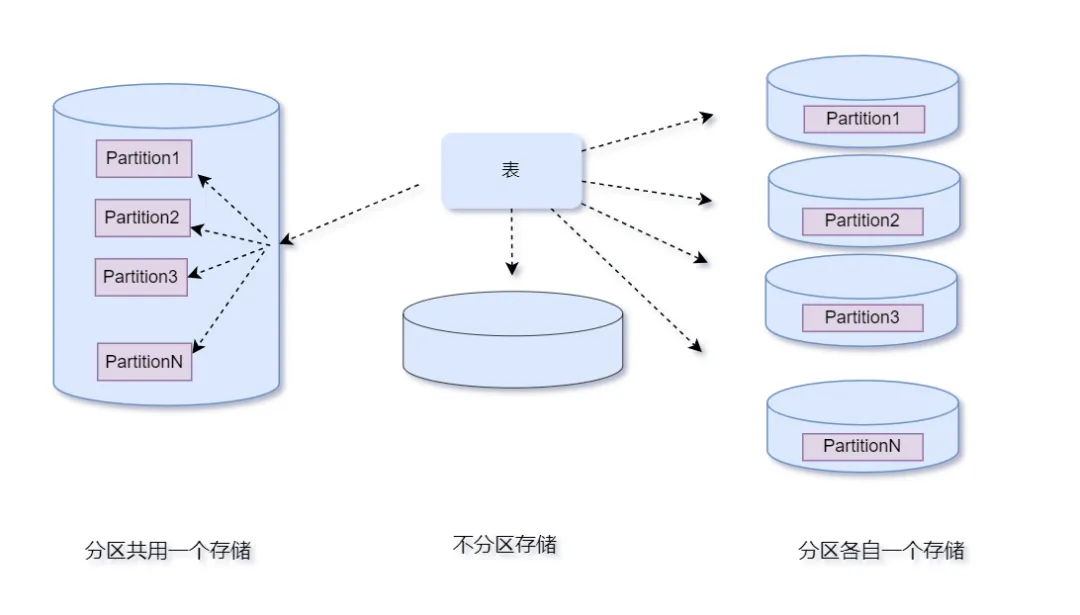
业务发展快，**初次分库分表后，满⾜不了数据存储，导致需要多次扩容**

6、**分库分表技术选型问题**

市场分库分表中间件相对较多，框架各有各的优势与短板，应该如何选择。

对于大数据量，你们怎么操作

分区：



就是把一张表的数据分块储存，在逻辑上看最终还是一张表，**但底层分配到不同的硬盘或者文件系统。（适合大型表）**

**好处：**

相对于单个文件系统或是硬盘，分区可以存储更多的数据；

数据管理比较方便，比如要清理或废弃某年的数据，就可以直接删除该日期的分区数据即可；

精准定位分区查询数据，不需要全表扫描查询，大大提高数据检索效率；

可跨多个分区磁盘查询，来提高查询的吞吐量；

在涉及聚合函数查询时，可以很容易进行数据的合并；

**分库，分表。**

**主从分离（读写分离）**

**将主服务器上的数据复制到多个从服务器上**

**主库（Master）：**

**负责所有的写操作（INSERT、UPDATE、DELETE），将数据更改同步到从库。**

**从库（Slave）：**

**负责处理读操作（SELECT），从主库复制数据更新。**

**好处：**

**数据冗余和备份**

**主从架构提供数据冗余，即使主库故障，从库可以作为备份，提供数据恢复。**

**分摊读取负载**

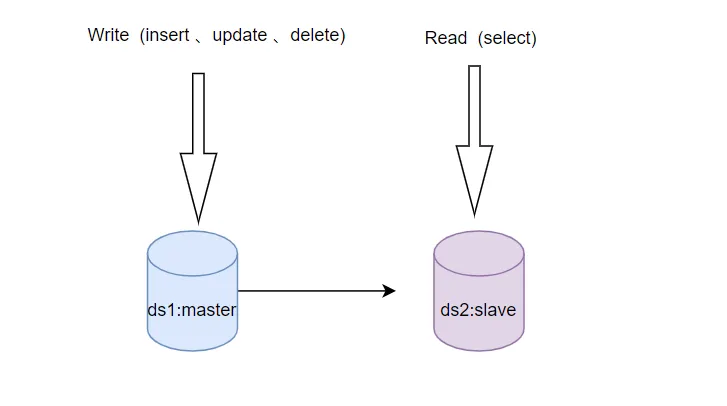
**读操作分布到多个从库中，减少主库的读负载，提高系统整体读性能。**

**提供高可用性**

**当主服务器发生故障或维护时，可以快速将从服务器提升为新的主服务器，保持系统的连续性和可用性，减少服务中断时间。**

**数据分析和报表生成**

**通过从服务器进行数据分析、查询和报表生成等操作，可以减少对主服务器的影响，保持主服务器的高性能。从服务器可以用于处理复杂的查询和数据分析任务，从而提供更好的用户体验。**



**坏处：**

**数据一致性延迟：主从复制过程中存在一定的延迟，从库的数据可能不是最新的。**

**写性能瓶颈：主库需要处理所有写操作，写性能可能成为瓶颈，**

**运行 ps -aux 可能会得到如下输出：**

**复制代码**

**USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND**

**root 1 0.0 0.1 22560 3460 ? Ss 10:21 0:00 /sbin/init**

**root 117 0.0 0.3 45056 6740 ? Ss 10:21 0:00 /lib/systemd/systemd-journald**

**...**

**输出字段说明：**

**USER：进程所有者的用户名**

**PID：进程ID**

**%CPU：进程占用的CPU百分比**

**%MEM：进程使用的物理内存百分比**

**VSZ：进程使用的虚拟内存大小（以KB为单位）**

**RSS：进程占用的物理内存大小（以KB为单位）**

**TTY：与进程关联的终端（如果有的话）**

**STAT：进程状态码**

**START：进程启动时间**

**TIME：进程使用的总CPU时间**

**COMMAND：启动进程的命令**

**怎么设计一个数据库的表：**

**命名规范（表名、字段名、索引名 要具有规范性、易读性）**

**选择合适的字段类型（占用尽量少的空间，字段长度一般设置为 2^n ）**

**整数（tinyint、smallint、int、bigint）**

**小数（尽量使用 decimal，少用 float 或 double）**

**时间（datetime）**

**字符串（char、varchar、text、longtext）**

**合理的主键（尽量与业务逻辑无关）**

**优先考虑逻辑删除，而不是物理删除**

**添加通用字段**

**id：主键（必须）**

**gmt\_create：创建时间（必须）**

**gmt\_modified：修改时间（必须）**

**version：版本号，用于乐观锁（非必须）**

**remark：数据记录备注（非必须）**

**creator：创建人（非必须）**

**modified\_by：修改人（非必须）**

**单表字段数量不宜过多（一般不超过 20 个）**

**尽量使用 not null 定义字段**

**合理添加索引（单表索引数量一般不超过 5 个）**

**不需要严格遵守 3NF（数据库三范式），字段合理冗余**

**第一范式：对属性的原子性，要求属性具有原子性，不可再分解；**

**第二范式：对记录的唯一性，要求记录有唯一标识，即实体的唯一性，不存在部分依赖；**

**第三范式：对字段的冗余性，要求任何字段不能由其它字段派生出来，即字段无冗余，不存在传递依赖。**

**避免使用 MySQL 保留字（select、insert…）**

**尽量不使用外键关联（一般通过业务逻辑保证关联）**

**尽量使用 InnoDB 存储引擎**

**选择合适的字符集（utf8、utf8mb4、GBK、latin1）**

**字段尽量添加注释**

**说下MySQL的锁**

**1. 全局锁**

**全局锁主要应⽤于做全库逻辑备份，这样在备份数据库期间，不会因为数据或表结构的更新，**

**⽽出现备份⽂件的数据与预期的不⼀样，加上全局锁，意味着整个数据库都是只读状态。**

**2. 表级锁**

**元数据锁（MDL）：对数据库表进⾏操作时，会⾃动给这个表加上元数据锁，为了保证当⽤户对表执⾏ CRUD 操作时，其他线程对这个表结构做了变更。元数据锁在事务提交后才会释放。**

**意向锁：**意向锁⽤于指示⼀个事务在未来可能会请求对某些资源（如数据⾏）的锁定 flush tables with read lock (FTWRL)。意向共享锁表示事务打算在资源上获得共享锁。其他事务可以继续获得共享锁，但不能获得排他锁。 意向排他（独占）锁： 表示事务打算在资源上获得排他锁。

**AUTO-INC 锁：表⾥的主键通常都会设置成⾃增的，之后可以在插⼊数据时，可以不指定主键的值，数据库会⾃动给主键赋值递增的值通过 AUTO-INC 锁实现的。在插⼊数据时，会加⼀个表级别的 AUTO-INC 锁，然后为被 AUTO\_INCREMENT 修饰的字段赋值递增的值，等插⼊语句执⾏完成后，才会把 AUTO-INC 锁释放掉。其他事务的如果要向该表插⼊语句都会被阻塞，从⽽保证插⼊数据时字段的值是连续递增的。**

**3. ⾏锁**

**记录锁：锁住的是⼀条记录，记录锁分为排他锁和共享锁。**

**间隙锁：只存在于可重复读隔离级别，⽬的是为了解决可重复读隔离级别下幻读的现象。锁定⼀个范围，不如别的记录插入修改**

间隙锁之间是兼容的，两个事务可以同时持有包含共同间隙范围的间隙锁，并不存在互斥关系。

**Next-Key Lock：Next-Key Lock临键锁，是 Record Lock + Gap Lock 的组合，锁定⼀个范围，并且锁定记录本身。next-key lock 即能保护该记录，⼜能阻⽌其他事务将新纪录插⼊到被保护记录前⾯的间隙中。**

**插⼊意向锁：⼀个事务在插⼊⼀条记录的时候，需要判断插⼊位置是否已被其他事务加了**

**间隙锁（next-key lock 也包含间隙锁）。如果有的话，插⼊操作就会发⽣阻塞，直到拥有**

**间隙锁的那个事务提交为⽌，在此期间会⽣成⼀个插⼊意向锁，表明有事务想在某个区间**

**插⼊新记录，但是现在处于等待状态。**

**表锁：每次操作锁表，开销小，加锁快，并发度最低**

**行锁：**每次操作锁住⼀⾏数据 ，开销⼤，加锁慢，并发性高，锁冲突概率低。

**说下操作系统的锁：**

**两个基础的锁：**

互斥锁：互斥锁是⼀种最常⻅的锁类型，⽤于实现互斥访问共享资源。**在任何时刻，只有⼀个线程可以持有互斥锁，其他线程必须等待直到锁被释放。这确保了同⼀时间只有⼀个线程能够访问被保护的资源。**

**⾃旋锁：⾃旋锁是⼀种基于忙等待的锁，即线程在尝试获取锁时会不断轮询，直到锁被释放。**

**睡眠锁**

**其他的锁都是基于这两个锁的**

**读写锁：允许多个线程同时读共享资源，只允许⼀个线程进⾏写操作。分为读（共享）和写（排他）两种状态。**

**悲观锁：认为多线程同时修改共享资源的概率⽐较⾼，所以访问共享资源时候要上锁**

**乐观锁：先不管，修改了共享资源再说，如果出现同时修改的情况，再放弃本次操作。**

**DNS（Domain Name System）域名管理系统，是当⽤户使⽤浏览器访问⽹址之后，使⽤的第⼀个重要协议。DNS 要解决的是域名和 IP 地址的映射问题。**

1. ⾸先⽤户在浏览器输⼊URL地址后，会先查询浏览器缓存是否有该域名对应的IP地址。

2. 如果浏览器缓存中没有，会去**计算机本地的Host⽂件中查询是否有对应的缓存。**

3. 如果Host⽂件中也没有则会向**本地的DNS解析器（通常由你的互联⽹服务提供商（ISP）提供）**发送⼀个DNS查询请求。

4. 如果本地DNS解析器没有缓存该域名的解析记录，它会向**根DNS服务器发出查询请求。**根DNS服务器**并不负责解析域名，但它能告诉本地DNS解析器应该向哪个顶级域**（.com/.net/.org）的DNS服务器继续查询。

5. 本地DNS解析器接着向指定的顶级域DNS服务器发出查询请求。顶级域DNS服务器也不负责具体的域名解析，但它能告**诉本地DNS解析器应该前往哪个权威DNS服务器查询下⼀步的信息。**

6. 本地DNS解析器最后向权威DNS服务器发送查询请求。 权威DNS服务器是负责存储特定域名和IP地址映射的服务器。当**权威DNS服务器收到查询请求时，它会查找"example.com"域名对应的IP**地址，并将结果返回给本地DNS解析器。

7. **本地DNS解析器将收到的IP地址返回给浏览器**，并且还会**将域名解析结果缓存在本地，以便下次访问时更快地响应。**

**进程线程同步互斥，怎么解决**

**进程同步是指多个并发执⾏的进程之间协调和管理它们的执⾏顺序，以确保它们按照⼀定的顺序或时间间隔执⾏。**⽐如说，你想要和你的队友⼀起完成⼀个副本，你们需要相互配合，有时候等待对⽅的信号或者消息，有时候按照对⽅的要求执⾏某些动作，这就是进程同步。

**互斥指的是在某⼀时刻只允许⼀个进程访问某个共享资源。当⼀个进程正在使⽤共享资源时，其他进程不能同时访问该资源。**

**解决进程同步和互斥的问题有很多种⽅法，其中⼀种常⻅的⽅法是使⽤信号量和 PV 操作。**信号量是⼀种特殊的变量，它**表示系统中某种资源的数量或者状态**。PV 操作是⼀种对信号量进⾏增加或者减少的操作，它们可以⽤来控制进程之间的同步或者互斥。

举个例⼦，假设有⼀个信号量 s 表示⼀个祭坛是否可⽤，初始值为 1。如果 s 的值为 1，表示祭坛空闲；如果 s 的值为 0，表示祭坛被占⽤；如果 s 的值为 -1，表示有⼀个⼈在等待使⽤祭坛。那么我

们可以⽤ PV 操作来实现对祭坛的互斥访问：

如果你想要使⽤祭坛，你就执⾏ P(s) 操作，将 s 的值减 1。如果结果为 0 或者正数，表示你可

以使⽤祭坛；如果结果为负数，表示有⼈在使⽤祭坛，你就必须等待。

如果你使⽤完了祭坛，你就执⾏ V(s) 操作，将 s 的值加 1。如果结果为正数或者 0 ，表示没有⼈在等待使⽤祭坛；如果结果为负数，表示有⼈在等待使⽤祭坛，你就需要唤醒他们中的⼀个。这样就可以保证每次只有⼀个⼈能够使⽤祭坛，实现了进程互斥。

**除此之外，下⾯的⽅法也可以解决进程同步和互斥问题：**

**临界区（Critical Section）：** 将可能引发互斥问题的代码段称为临界区。为了实现互斥，每个进程在进⼊临界区前必须获取⼀个锁，退出临界区后释放该锁。这确保同⼀时间只有⼀个进程可以进⼊临界区。

**互斥锁（Mutex）： 互斥锁是⼀种同步机制，⽤于实现互斥。每个共享资源都关联⼀个互斥锁，进程在访问该资源前需要先获取互斥锁，使⽤完后释放锁。只有获得锁的进程才能访问共享资源。**

**条件变量（Condition Variable）： 条件变量⽤于在进程之间传递信息，以便它们在特定条件下等待或唤醒。通常与互斥锁⼀起使⽤，以确保等待和唤醒的操作在正确的时机执⾏。**

**TCP流量控制：**

**使用滑动窗口机制实现的**

**接收方维护一个接收窗口，表示可以接收的数据段的范围。**

**当接收方收到数据后，会通过确认（ACK）消息告知发送方已成功接收数据，并在确认消息中加入当前的接收窗口大小的值。发送方根据接收窗口大小来确定可以发送的数据量，确保不超出接收方的处理能力。**

**如果接收方的缓冲区已满或者处理能力不足，它可以通过减小窗口大小的方式来通知发送方降低发送速率。反之，如果接收方的缓冲区有足够的空间，它可以通过增大窗口大小的方式来提高发送速率。**

**发送方能根据接收方能力和网络状况动态调整自己的发送速率。**

**TCP拥塞控制是怎么实现的？**

TCP拥塞控制可以在**⽹络出现拥塞**时动态地调整数据传输的速率，以防⽌⽹络过载。

TCP拥塞控制的主要机制包括 以下⼏个⽅⾯：

1. **慢启动（Slow Start）**： **初始阶段，**TCP发送⽅会以**较⼩的发送窗⼝开始传输数据**。随着每次成功收到确认 的数据，发送⽅**逐渐增加发送窗⼝的⼤⼩，实现指数级的增⻓，这称为慢启动**。这有助于在⽹络刚开始传输时 谨慎地逐步增加速率，以避免引发拥塞。

2. **拥塞避免（Congestion Avoidance）**： ⼀旦达到**⼀定的阈值（通常是慢启动阈值**），TCP发送⽅就会进⼊ 拥塞避免阶段。在拥塞避免阶段，发送⽅以**线性增加的⽅式**增加发送窗⼝的⼤⼩，⽽不再是指数级的增⻓。**这 有助于控制发送速率，以避免引起⽹络拥塞。**

3. **快速重传（**Fast Retransmit）：当发送⽅发送的数据包丢失或⽹络出现拥塞时，接收⽅会发送**重复确认（duplicate ACK）通知发送⽅有数据包丢失。**当发送⽅收到⼀定数量的重复确认时，它会⽴即重传丢失的数据包，**⽽不是等待超时。**这有助于更快地恢复由于拥塞引起的数据包丢失。

4. **快速恢复**（Fast Recovery）： **在发⽣快速重传后，TCP进⼊快速恢复阶段**。在这个阶段，**发送⽅不会回到慢 启动阶段**，⽽是将**慢启动阈值设置为当前窗⼝的⼀半**，**并将拥塞窗⼝⼤⼩设置为慢启动阈值加上已确认但未被 快速重传的数据块的数量。这有助于更快地从拥塞中恢复。**

**TCP连接确保可靠性⽅法如下：**

1. **序列号：** TCP给每个数据包指定序列号，接收⽅根据序列号对数据包进⾏排序，并根据序列号对数据包去重。

2. **流量控制：** TCP连接的每⼀⽅都有固定⼤⼩的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收⽅来不及处理发送⽅的数据，能提示发送⽅降低发送的速率，防⽌包丢失。TCP利⽤滑动窗⼝实现流量控制。

3. **拥塞控制：** 当⽹络拥塞时，减少数据的发送。

4. **确认应答**： 通过 ARQ 协议实现。基本原理是每发完⼀个分组就停⽌发送，等待对⽅确认。**如果没收到确认，会重发数据包**，直到确认后再发下⼀个分组。

5.**超时重传：** 当TCP发出⼀个数据段后，它启动⼀个定时器，等待⽬的端确认收到这个报⽂段。如果**不能及时收到⼀个确认**，将重发这个报⽂段。

6.**数据块⼤⼩控制**： 应⽤数据被分割成TCP认为最合适发送的数据块，再传输给⽹络层，数据块被称为报⽂段或段。

7. **校验和：** TCP将保持它⾸部和数据的校验和。这是⼀个端到端的检验和，⽬的是检测数据在传输过程中的任何变化。如果收到报⽂的检验和有差错，TCP将丢弃这个报⽂段和不确认收到此报⽂段。

**MSQL执行引擎**

**主要有MyISAM、InnoDB、Memery等引擎：**

**InnoDB 引擎提供了对事务ACID的⽀持，还提供了⾏级锁和外键的约束。**

**MyISAM 引擎不⽀持事务，也不⽀持⾏级锁和外键约束。**

**Memery 就是将数据放在内存中，数据处理速度很快，但是安全性不⾼。**

**MSQL日志文件**

**undo log 是 Innodb 存储引擎层⽣成的⽇志，实现了事务中的原⼦性，主要⽤于事务回滚和MVCC。**

**redo log 是物理⽇志，记录了某个数据⻚做了什么修改，每当执⾏⼀个事务就会产⽣⼀条或者多条物理⽇志。**

**binlog (归档⽇志）是Server 层⽣成的⽇志，主要⽤于数据备份和主从复制。**

**relay log 中继⽇志，⽤于主从复制场景下， slave 通过io线程拷⻉master的 bin**

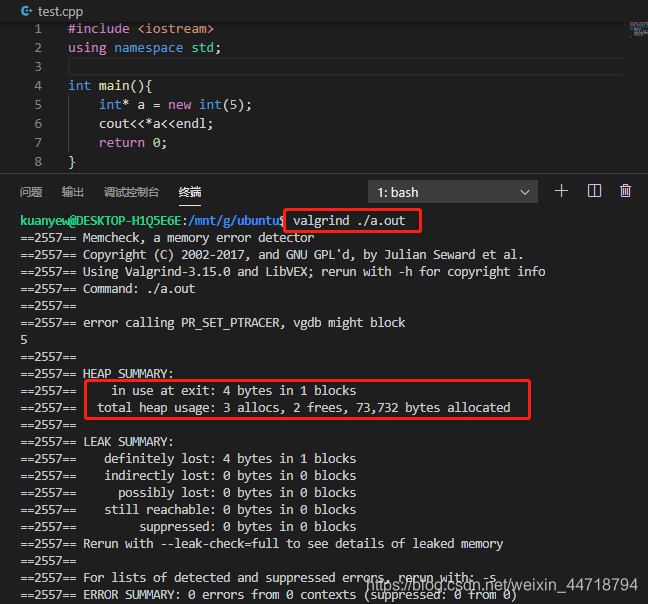
**log 后本地⽣成的⽇志.**

**什么方法可以检测内存泄露：**

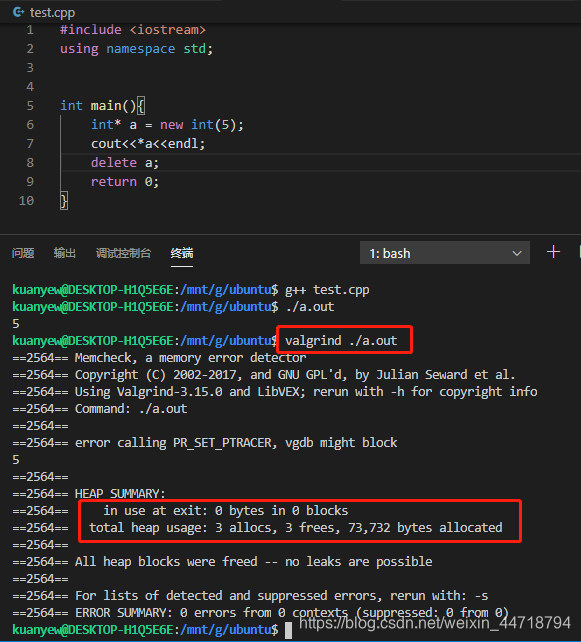
**内存泄露检测工具valgrind**

**这是一个linux下的内存泄露检测工具**

**内存泄露情况：**



**无内存泄露：**



**内存泄露几种常见的原因：**

**未释放动态分配的内存。**

#### ****异常导致的资源泄漏****

**基类未使用虚析构**

#### ****智能指针的循环引用****

**malloc的使用**

malloc函数其实就是找一片指定大小的内存空间，返回值是指向一段可用内存的起始位置的指针，void\*类型。

**int \*p;**

**p = (int \*)malloc(sizeof(int));**

**Malloc的实现**

[malloc](https://so.csdn.net/so/search?q=malloc&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/youwuwei2012/article/details/_blank)函数的实质体现在，它会将系统中可用的内存块以空闲块链表的形式保存起来。

调用malloc时，扫描空闲块链表以找到一个足够大的块。**如果这个块太大，那么将它分为两部分：大小合适的块返回给用户，剩下的部分留在空闲块链表内；**如果找不到一个足够大的块，**那么就sbrk系统调用从操作系统中申请一个大的堆内存块加入空闲块链表中**，再重新分配内存。

当**释放内存时**，free函数会将**释放后的内存块重新加入空闲链表**，如果被释放的内存块与另一空闲内存块相邻，那么将这两个块合并起来再加入链表，**以确保不会有太多的内存碎片**。

生成可执行文件，g++ -g -o main main.cpp

。

**简易版：系统调用的过程一般包括以下步骤：**

1. **准备参数： 用户程序将需要的参数传递给系统调用。这些参数可能包括文件描述符、缓冲区地址、数据长度等信息，具体取决于调用的系统服务。**
2. **触发中断： 用户程序通过软中断（例如，int 0x80或类似的指令）或陷阱指令（例如，syscall指令）触发一个中断，使得处理器从用户态切换到内核态。通常涉及到修改处理器的特权级别，以便让内核代码执行。**
3. **确定系统调用号： 内核通过查看特定的寄存器（例如eax寄存器）或内存中的某个位置，确定用户程序请求的具体系统调用。**
4. **执行系统调用： 根据系统调用号，内核调用相应的系统调用处理函数。**

**这个函数执行用户请求的服务，并使用用户提供的参数。**

1. **返回结果：系统调用处理完毕后，将结果返回给用户程序。通常，保存的状态被恢复，特权级别重新设置，使得程序从内核态切换回用户态。**
2. **处理结果： 用户程序检查系统调用的返回值，以了解调用是否成功。如果发生错误，通常可以通过查看错误码来获取更多信息。**

软链接和硬链接

硬链接：

一个文件有一个或多个文件名，相当于有多个文件名能同时修改同一个文件

多个目录项中的索引节点指向同一个文件的 inode，即同一个文件有多个硬链接。

硬链接只能在同一个文件系统内使用，因为每个文件系统有自己的 inode 数据结构和列表。

**硬链接会增加⽬标⽂件的链接计数，也会增加inode的链接计数。当链接计数为零时，⽂件系 统才会释放相关的数据块**

当**删除文件的所有硬链接以及源文件时，系统才会彻底删除该文件。**

软链接：

**软链接相当于创建了一个新的文件，它有独立的 inode。**

**软链接的内容是另一个文件的路径，因此访问软链接实际上是访问到了另一个文件。**

**软链接可以跨文件系统使用，即使目标文件被删除，链接文件仍然存在，只是指向的文件找不到了而已**。

软链接只会增加⽬标⽂件的链接计数，⽽不会增加inode的链接计数