**TCP/UDP区别：**

UDP基于无连接面向数据报提供不可靠、尽最大努力交付服务，可一对一、…，首部开销小；

TCP基于字节流面向连接的可靠服务，只能端到端连接，头部开销大；

**TCP/UDP优缺点：**

UDP：优点是快，没有TCP各种机制，少了首部信息和重复确认的过程，节省了大量网络资源；缺点是不稳定，只管数据发送不管过程和结果，网络不好的时候会造成数据丢失。

TCP：优点是可靠、稳定，有确认、窗口、重传和拥塞控制机制，数据传完之后会断开连接来节约系统资源。缺点是慢，效率低，占用资源高，传输数据前要先建立连接，并且传输时各种机制消耗时间大。

**TCP/UDP适用场景：**

UDP适用于实时性应用程序，如语音通话、视频会议，允许网络不好时丢失一些数据但不允许太大延迟；

TCP适用于数据要求准确、对速度没有硬性要求的场景，如FTP（文件传输）、HTTP/HTTPS（超文本传输）。

**TCP三次握手四次挥手：**

握手：客户端首先发送一个带有自身数据通讯序号的连接请求；服务器端收到连接请求，发出应答，包含自身数据通讯序号以及对客户端请求的确认号，该确认号是客户端序号+1；客户端收到应答，向服务器端发送确认报文，确认报文同样为服务器数据序号+1；

四次挥手：客户端数据发送完毕，发送一个释放连接请求，报文段首部的终止控制fin置为1，进入fin-wait(终止等待1)状态，序号设为u；

服务器端收到客户端释放请求报文，发出确认，确认号是u+1，进入close-wait阶段(此时服务器端仍可以向客户端发数据)。

服务器端没有数据要发送了，再发送一个fin=1，序号为w的断开连接的数据报，确认号位ack=u+1，服务器端进入ack\_last阶段(最后确认阶段)；

客户端收到服务器端连接释放请求，发出确认，进入TIME-WAIT阶段，两个最长报文数据寿命后，客户端连接释放。

**GET和POST的区别：**

GET用来获取数据，长度最大为2048字节，参数会附加在url中，以“？”分割url和传输数据，因此是明文传输、参数是显式的；GET请求会保留在浏览器历史记录或web服务器日志中，会被浏览器主动缓存；

POST用来提交或者修改数据，无长度限制，参数会放在请求体中，只有用工具可以看到，不会主动缓存。

**简述C++中的多态：**

多态分为静态多态、动态多态，多态意味着同样的语句在实际运行时有多种不同的表现形态。静态多态包括函数重载、运算符重载等，在编译期间完成，动态多态发生在程序运行时，根据父类的引用或指针指向的对象来确定自己应该调哪一个类的虚函数。

**动态多态实现条件：**

要有纯虚函数，要有继承关系，要有虚函数重写，要有父类指针指向子类对象。

**动态多态实现原理：**

类中声明虚函数时，编译器会在类中生成一个虚函数表，存储类的虚函数指针。虚类成员函数被放入虚函数表中，存在虚函数时，每个对象中都有一个指向虚函数表的指针。多态调用时，虚函数表指针根据这个对象在对应类的虚函数表中查找被调用的函数从而找到函数入口地址。

**死锁简介及原因：**

两个及以上的进程在执行过程中因互相抢夺资源造成的一种相互等待的现象，若无外力作用，他们将无法继续下去。

**死锁预防：**有序资源分配法、银行家算法。

**Mysql索引：**

索引就像指向表行的指针，是一种允许查询操作快速确定哪些行符合WHERE子句中的条件，并检索到这些行的其他列值的数据结构； 索引主要有普通索引、唯一索引、主键索引、外键索引、全文索引、复合索引几种；

**只要创建了索引，就一定会走索引吗？** 不一定。 比如，在使用组合索引的时候，如果没有遵从“最左前缀”的原则进行搜索，则索引是不起作用的。

**面向对象三大特征：**

面向对象三大特征：封装继承多态。

封装是将事物的属性(成员变量)和行为(成员函数)封装到一起(类)，隐藏属性和实现细节，仅对外公开接口来与对象交互。提高了代码复用性和安全性。

继承：C++通过继承实现两个类的联系，继承基类的派生类中的成员包括基类继承的和自己增加的。继承的表现了共性，新增的体现了个性。能够提高代码复用性，增加了代码拓展性；是多态前提。

多态：通过基类指针或引用指向的对象来确定自己具体应该调用哪一个类的虚函数。改善了代码可读性和组织性，也使创建的程序具有可扩展性。

**进程通信方式：**

管道、命名管道、socket、消息队列、共享内存、信号量、内存映射；

**进程和线程区别：**

进程是资源分配的基本单位，而线程是处理器调度的基本单位；

进程有独立地址空间，线程仅有自己的堆栈和局部变量，没有独立地址空间，进程上下文切换消耗时间长，资源消耗大，效率低一些，进程并发性低线程并发性较高，进程有一个程序运行的入口、顺序执行序列和出口，但线程不能独立执行，必须依存在应用程序中由应用程序提供多个线程执行控制；系统在运行时会为每个进程分配不同的内存空间，而线程只能共享资源；一个进程崩溃不会对其他进程产生影响，但一个线程崩溃整个进程都死掉。

**epoll原理？**

**浏览器从输入URL开始到页面中显示内容中间发生了什么？**

1. 解析URL得到发给WEB的信息，产生HTTP请求信息；
2. 查询服务器域名对应IP地址（首先在本地hosts找是否有网址映射关系，若有直接用；若没有则去本地DNS解析缓存查是否有映射；若本地DNS没有，首先查本地TCP/IP设置的DNS服务器，若不存在，则开启转发模式，在上一级DNS服务器里一级一级找；或者不开启转发模式，回去找13组根DNS，根DNS服务器判断该域名归谁管，返回该IP地址）；
3. 向该IP地址的web服务器通过TCP三次握手建立连接，发送HTTP请求；
4. 若发送出去服务器返回重定向，浏览器按照重定向地址重新发送请求。若请求参数有问题，服务器返回404，服务器端挂了返回500。
5. 浏览器拿到服务器所提供数据，开始渲染页面并获取HTML中的图片、音频、视频等。

**HTTP状态码及含义：**

1xx：服务器端接受了请求；

2xx：请求已被服务器端成功接收；

3xx：重定向；

4xx：客户端请求发生了错误；

5xx：服务器端响应出了问题；

*200：成功：服务器处理了请求，即服务器提供了网页；*

*201：已创建，成功，且服务器创建了新的资源；*

*202：已接受：服务器接受了请求但还未处理；*

203：非授权：服务器成功处理了请求但返回的信息可能出自另一来源；

204：无内容：服务器成功处理了信息但未返回任何内容；

206：部分服务：服务器部分处理了GET请求；

301：永久移动：请求的网页已永久移动到新位置，返回此相应的同时将请求者转移到新位置；

302：临时移动：服务器目前从不同位置网页相应请求，但请求者继续使用原来位置进行以后的请求；

304：未修改：自从上次请求后网页未修改过，不会返回网页内容；

305：使用代理：请求者只能使用代理访问该网页；

401：未授权：需要进行身份验证(需登录等)

404：未找到：服务器找不到请求的网页；

500：服务器内部错误

503：服务器不可用(超载或停机，一般是暂时的)

**常用LINUX命令：**

cd:切换目录；ls：查看当前文件与目录；cp：复制文件or文件夹；mv：移动文件或文件夹；rm：删除文件或文件夹；ps：查看进程状况；kill：向进程发信号；tar：打包文件；cat：查看文件内容；free：查看内存使用情况；pwd：显示当前工作目录；

**HTTP 和 HTTPS 的区别**

HTTP基于TCP，HTTPS基于TLS。

HTTP端口号80，HTTPS为443。

HTTP往返时间1RTT，HTTPS往返时间3RTT。

HTTP只需一次TCP连接，而HTTPS需要两次。

HTTP传输数据几乎是明文传输，HTTPS在TCP和HTTP之间加入了SSL/TLS协议，具有信息加密、校验机制和身份证书验证，安全性高。

**TCP 如何实现可靠传输**

序列号：TCP给每一个发送的包进行编号，接收方进行排序，并丢弃重复项；

校验和：TCP保持首部校验和，检测数据在传输过程中的变化；

确认应答：若收到的数据包校验和通过，发送确认应答；若错误则丢弃

流量控制：TCP每一方连接都有固定大小的缓冲空间，TCP接收端只允许发送端发送接收端缓冲区所能接纳的数据。当接收方来不及处理发送方数据，可以提示发送方降速；利用滑动窗口实现；

拥塞控制：当网络拥塞时减少数据发送；

停止等待：每发一个分组就停止发送等对方确认，收到确认再接着发；

超时重传：发送一个段后就启动一定时器，等目的端确认收到这个报文段；

**数据库的 ACID**

**C++11、C++14、C++17、C++20 都有什么新特性**

C++11：新增long long型，char\_16t型等数据，新增auto，decltype函数，新增nullptr空地址，lambda表达式，新增unordered\_map,unordered\_set，新增显示转换运算符，继承构造函数，列表初始化

**进程的几种状态？**

创建、就绪、执行、阻塞、终止；

创建：一个进程启动，首先进入创建状态，获取系统资源创建进程管理快PCB完成资源分配；

就绪：程序已准备好，但还没获得CPU资源无法运行；等待进程调度操作

运行：获取CPU资源，被系统调度，具有时间片开始进入运行状态；若时间片用完则进入就绪状态；

阻塞：在运行期间如果进行了阻塞操作，此进程无法操作，进入到阻塞状态，操作完成后再次进入就绪态；

终止：进程结束或被系统中止；

**请你介绍一下 I/O 多路复用**

I/O多路复用是一种实现并发网络编程的机制，它允许一个进程监视多个文件描述符（sockets、pipes等），并在其中任何一个文件描述符准备好执行读写操作时进行处理。这种机制可以避免使用多线程或多进程来处理并发的I/O操作，从而提高系统的性能和效率。

在常见的操作系统中，有三种主要的I/O多路复用机制：

select：select 是最古老的I/O多路复用机制之一，它允许程序员监视一组文件描述符，并在其中任何一个文件描述符就绪时通知应用程序。但是 select 存在一些限制，如文件描述符数量受到限制，效率不高等问题。

poll：poll 是对 select 的改进，它也可以监视一组文件描述符的状态变化，但没有文件描述符数量的限制。然而，poll 仍然需要遍历整个文件描述符数组，因此可能影响性能。

epoll：epoll 是 Linux 特有的I/O多路复用机制，相比于 select 和 poll，它具有更高的性能和扩展性。epoll 使用事件驱动的方式，只在文件描述符状态变化时进行操作，避免了遍历整个文件描述符数组的开销。

使用I/O多路复用机制可以在单个线程中处理多个并发的I/O操作，从而提高系统的性能和资源利用率。在编写高性能的网络服务器程序时，通常会选择使用I/O多路复用技术来处理大量的并发连接。

**请你说说 new 的实现原理，new 和 malloc 的区别**

New实现原理：若为简单类型，直接调用operator new()，在该函数中调用malloc函数，若调用失败则会调用callnewh()，若返回0则抛出bac\_alloc异常，返回非0则继续分配内存；若为复杂类型，先调用operator new()，再在分配的内存上调用构造函数

new和malloc区别：new是操作符而malloc是函数，使用new申请内存无需指定内存大小，编译器会根据类型信息自行计算，而malloc需要显示指出所需内存大小；new分配失败会直接返回异常，malloc分配失败返回null；对非简单类型，new分配内存后会调用构造函数，且返回对应类型指针，malloc会返回空指针类型；malloc可分配任意大小指针，new只能分配所占内存整数倍大小；new可重载，malloc不可以。

**深拷贝和浅拷贝**

深拷贝和浅拷贝根本区别在于是否真正获取一个对象的复制实体，而不是“引用”。

浅拷贝又称值拷贝，将源对象的值拷贝到目标对象中，如果对象中有某个成员是指针类型，并且是在堆区创建，则浅拷贝仅仅拷贝这个指针变量的值，也就是两个对象中该指针类型数据和源对象中的该成员指向的是同一个堆空间；默认的拷贝构造函数和默认的赋值运算符重载函数都是浅拷贝。

深拷贝在拷贝的时候先开辟出和源对象大小一样的空间，然后将源对象里的内容拷贝到目标对象中，这样指针成员就指向了不同的内存位置，且里面的内容是一样的。这样析构时不会重复释放同一快内存。

**TCP如何实现可靠传输**

序列号，校验和，确认应答，流量控制，拥塞控制，停止等待，超时重传

**STL 中容器的类型，每种分别有哪些容器**

序列式容器(vector，list，array，deque)，关联式容器(set，map，multiset，multimap)，无序关联式容器(unordered\_map，unordered\_set)，容器适配器(stack，queue，priority\_queue)

**TIME\_WT状态**

TIME\_WT状态是为了保证被动关闭的一方能够正确收到主动关闭一方发送的确认数据包。

**拥塞控制机制**

网络不好的时候减少发送数据，防止太多数据进入网络中；有四种算法：慢启动(由小到大逐渐增大发送窗口)、拥塞避免(让拥塞窗口缓缓增大，即每经过一个往返时间RTT就把发送方拥塞窗口+1)、快重传(要求接收方每收到一个失序的报文就立即发出重复确认而不是等自己发数据时稍待确认)、快恢复。

**堆和栈区别**：

管理方式：堆的分配和释放由程序员控制，栈是编译器自动管理；

空间大小：栈小于堆；

碎片问题：堆会存在内存碎片，降低程序效率；栈是先进后出，在某一数据弹出之前，它之前所有的数据都已弹出；

生长方向：堆向上，沿着内存地址增加的方向；栈：向下；

分配方式：堆都是动态分配，栈有静态分配和动态分配；

分配效率：堆<栈；

**线程和协程区别：**

线程是操作系统的资源，创建、切换、停止都非常消耗资源；而协程不需要调用操作系统的功能，编译语言自动就能完成；

线程在多核环境下能够真正意义上运行，而协程是为并发而产生的；

一个具有多线程的程序可以同时运行几个线程，而协程却需要彼此协作运行；

线程都是同步机制，而协程是异步；

线程是抢占式，而协程是非抢占式的，需要用户自己释放使用权来切换，因此同一时间只有一个协程有运行权；

**TCP/IP五层模型：**

应用层：定义应用进程间的通信和交互规则。通过应用进程间的信息数据流通完成特定网络应用；如电子邮件，万维网；协议有HTTP，TCP协议；

运输层：运输层任务是向两个计算机的进程之间的通信提供一种通用的数据传输服务，应用层通过运输层可以传输报文。通常不会对应用层协议进行详细划分，多种应用层协议公用同一个运输层服务；

网络层：为负责网络上不同的主机提供通信服务，将运输层产生的报文段或用户数据封装为分组或包进行传输；网络层还需要寻找合适的路由器，让数据包能够通过路由器找到目的主机；

数据链路层：将网络层交下来的数据报组装成帧，每一帧都包括数据和控制信息；

物理层：传输比特流数据，通过高低电压实现；

**滑动窗口**

流量控制中的窗口长度会持续向前滑动。已被客户端发送但还未确认的分组的许可序号范围可以被看成一个在序号范围内长度为N的窗口，随着TCP协议运行、数据传输，这个窗口在序号空间向前滑动，因此这个窗口被称为滑动窗口。

**指针和引用区别**

指针是一种数据类型，保存地址类型数据，引用是变量的别名；

指针可以是空值，而引用必须被初始化不可以为空；

指针可以有多级，但引用只能一级；

指针需要解引用而引用无需解引用；

指针的值初始化后可以改变，即可以指向其他存储单元，而引用初始化后就不能改变了；

Sizeof(指针)得到的是指针变量本身的大小，而引用得到的引用对象大小；

指针作为形参传递的是指针变量的值，而引用作为函数参数传递时传递了实参本身而非拷贝副本；

**C++ 的重载和重写**

重载是不同函数使用相同函数名，但函数参数个数或类型不同，返回值无关；用来实现静态多态；

重写：在派生类中重新对基类的虚函数重新实现，函数名和参数都一样只是函数体都一样；方法声明必须与父类中被重写的方法相同，访问修饰符权限大于等于父类中的权限；实现动态多态；

隐藏：函数查找时名字先于类型检查，若派生类成员和基类中的成员同名，就隐藏掉。首先在相应作用域中查找函数，找到名字一样的就停止查找；

**DNS解析过程：**

DNS（Domain Name System）解析是将域名转换为 IP 地址的过程，以便在互联网上定位特定的主机。以下是 DNS 解析过程的简要步骤：

浏览器缓存：首先，浏览器会检查自己的缓存中是否有与所请求域名相关的 IP 地址记录。如果有，解析过程就到此为止。

操作系统缓存：如果浏览器缓存中没有找到相应记录，操作系统会检查自己的 DNS 缓存来查找域名对应的 IP 地址。如果在操作系统缓存中找到了解析记录，解析过程也将结束。

本地 DNS 解析器：如果在操作系统缓存中未找到相应记录，系统会向本地 DNS 解析器发送查询请求。本地 DNS 解析器通常由 Internet 服务提供商（ISP）提供，它会首先检查自己的缓存，如果找不到则继续向上级 DNS 服务器发起查询请求。

根域名服务器：如果本地 DNS 解析器无法直接解析域名，它会向根域名服务器发送查询请求。根域名服务器负责指导查询流程到达正确的顶级域名服务器。

顶级域名服务器：根域名服务器返回顶级域名服务器的地址后，本地 DNS 解析器会向顶级域名服务器发送查询请求，以获取次级域名服务器的地址。

权限域名服务器：一旦本地 DNS 解析器获得了次级域名服务器的地址，它会向该服务器发送查询请求，以获取目标域名对应的 IP 地址。

响应返回：最终，权限域名服务器会返回目标域名对应的 IP 地址给本地 DNS 解析器，然后本地 DNS 解析器将结果缓存并将其返回给操作系统，最终传递给浏览器。

通过这个过程，域名最终被解析为相应的 IP 地址，使得客户端能够与目标主机建立连接并进行通信。DNS 解析是互联网通信中重要的一环，它确保了我们能够方便地通过域名访问到各种网络资源。

**虚函数的实现原理：**

见前文；

**缓存穿透：**

客户端查询了不存在的数据，使这个请求直达存储层导致负载过大甚至宕机；

**缓冲击穿：**

一份访问量非常大的热点数据缓存失效瞬间，大量请求直达存储层导致服务崩溃；

**雪崩：**

某一时刻缓存层无法提供服务导致所有请求直达存储层造成数据宕机；

**C++struct和class区别：**

Struct是一个数据结构集合，class是一个对象数据封装‘

Struct默认public访问、继承，class默认private访问、继承；

Class可用于定义参数模板

**各数据类型 sizeof 是多少，sizeof 指针是多少，sizeof 原理：**

Bool – 1;char – 1;short – 2;int – 4;long – 4;long long – 8;

指针—32位平台是4字节，64位是8字节；

原理：编译时查找符号表判断类型然后根据基础类型来取值；

**为什么将析构函数设置成虚函数**

防止派生类释放时调用基类析构函数，造成内存泄露；

**哈希冲突的原因和影响因素、解决办法：**

**delete 和 free 的区别：**

delete是操作符free是函数；

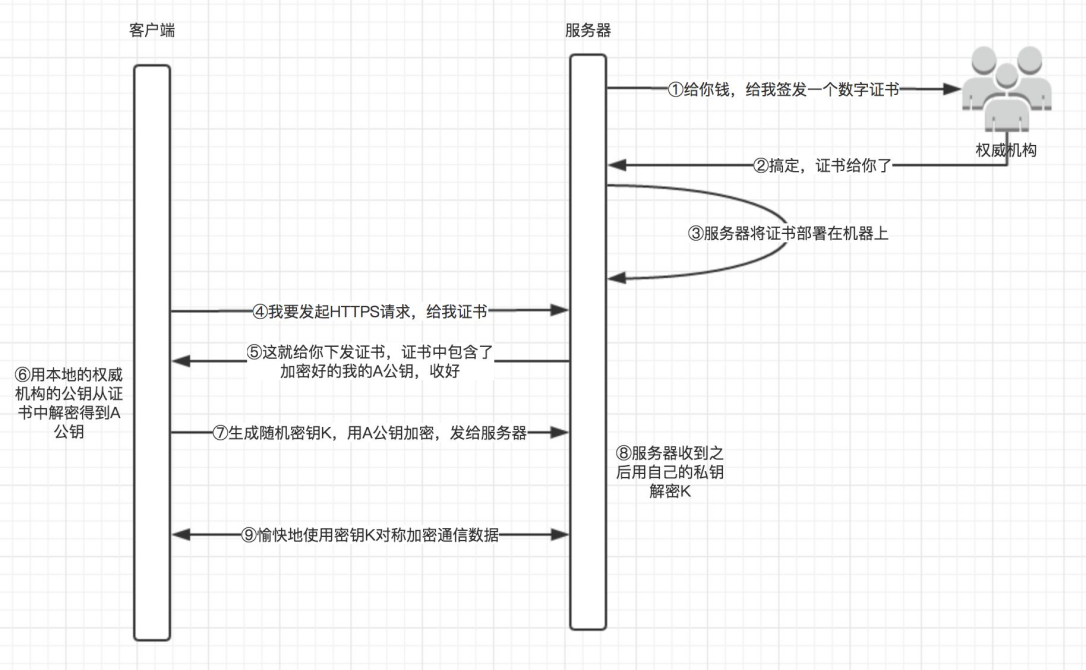
delete释放new的空间，free释放malloc空间；free不调用析构函数而delete会调用析构；调用free之前要检查指针是否为null；

**虚拟内存：**

由于指令都是直接访问物理内存的，那么若任何进程都可以修改其他进程的数据，甚至会修改内核地址空间的数据，这是不安全的；虚拟内存是计算机内存管理的一种技术，它使得程序认为自己有连续的可用内存，而实际上是被分割成多个物理碎片，还有部分暂时存储在外部磁盘存储器上，需要时进行数据交换；是通过内存映射表来实现的；

**HTTPS加密过程是怎样的？**

对数据进行对称加密，对称加密所要使用的密钥通过非对称加密传输。HTTPS加密分为两个阶段：证书认证阶段和数据传送阶段。证书认证阶段：使用非对称加解密算法对数据传输阶段的对称加密密钥进行加密和解密；数据传送阶段：通过证书认证阶段获取到的目标服务器的对称加密解密密钥，对数据进行加密传送给服务器。



1.客户端第一次向服务器发起HTTPS请求，连接到服务器的443（默认）端口。2.服务器端有一个密钥对，公钥和私钥。用来进行非对称加密使用，服务器端保存私钥，不能泄露，公钥可以发送给任何人。服务器将自己的数字证书（包含公钥）发送给客户端。3.客户端收到服务器端的数字证书之后，会对数字证书进行检查，验证合法性。如果发现数字证书有问题，那么HTTPS传输就中断。如果数字证书合格，那么客户端生成一个随机值，这个随机值是数据传输阶段时给数据对称加密的密钥，然后用数字证书中的公钥加密这个随机值密钥，这样就生成了加密数据使用的密钥的密文。到这时，HTTPS中的第一次HTTP请求就结束了。 4.客户端第二次向服务器发起HTTP请求，将对称加密密钥的密文发送给服务器。 5.服务器接收到客户端发来的密文之后，通过使用非对称加密中的私钥解密密文，得到数据传送阶段使用的对称加密密钥。然后对需要返回给客户端的数据通过这个对称加密密钥加密，生成数据密文，最后将这个密文发送给客户端。 6.客户端收到服务器端发送过来的密文，通过本地密钥对密文进行解密，得到数据明文。到这时，HTTPS中的第二次HTTP请求结束，整个HTTPS传输完成。

**什么情况会调用拷贝构造，什么时候会调用赋值操作？**

拷贝构造函数的调用时机：用一个对象初始化另外一个对象，对象以值传递的方式传递给函数参数，函数局部对象以值传递的方式从函数返回。

赋值操作的调用时机：将一个对象赋值给另外一个对象。

**虚析构函数作用？**

防止内存泄漏：防止遗漏资源的释放，若基类中的析构函数没有声明为虚函数，基类指针指向派生类对象时，若基类指针释放时不会调用派生类对象中的析构函数而是调用基类的析构函数，没有对子类析构，造成了内存泄漏。

**HTTP1.0和HTTP1.1的区别？**

HTTP1.1默认长连接而1.0默认短链接(每次都要重新建立一次连接)；

1.1有流水线和非流水线方式，流水线：客户收到HTTP响应报文前就能接着发新请求报文，非流水线：收到请求报文之后才能发下一个报文；

1.1新增了24个错误状态响应码，如409-请求资源与资源当前状态冲突，410-服务器上的某资源永久删除；

1.1引入了更多缓存策略；

1.1在请求头加入了range头域，允许只请求资源的某一部分，而1.0会送来整个对象；

**HTTP1.x和2.0区别？**

1. 二进制格式：1.x的解析是基于文本，但基于文本协议的格式解析存在天然缺陷。文本表现形式有多样性，二进制则不同，只认可0和1组合。
2. 多路复用：连接共享，每一个请求都是用作连接共享机制，一个请求对应一个id，一个连接上可以有多个请求，每个连接的请求都可以随机混杂在一起，多HTTP请求复用一个TCP连接(若发生丢包会触发重传，所有HTTP请求都要因此等待)
3. 头部压缩：1.x头部带有大量信息，2.0使用encoder来减少头部大小，通信双方各自缓存一份头部表，即避免了重复头部传输又减少了需要传输的大小；
4. 服务器推送：若请求了index.html文件，服务器会主动将它的依赖文件一起返回。

**UDP如何实现可靠传输？**

将TCP的可靠传输机制在应用层实现。在应用层提供超时重传机制；在UDP数据报定义一个首部，首部包含确认号和时间戳，时间戳是用来计算RTT，计算出合适的RTO(重传的超过时间)。然后以等-停的方式发送数据报，当时间超过，本端重传数据报同时RTO扩大为原来的两倍，重新开始计时。

**auto和decltype如何用？**

auto能够实现自动类型推断，要求进行显示初始化，让编译器能够将变量的类型设置为初始值的类型；decltype将变量的类型声明为表达式指定的类型，decltype的结果类型与表达式形式有密切关系。比如decltype()内层没有括号，那么得到的类型就是变量的类型，如果有多层括号decltyp(())那么返回的就是引用。

**虚函数可以是内联函数吗**

可以，但当虚函数表现多态性时不能内联。内联是在编译期建议编译器内联，而虚函数多态性在运行期，编译器无法知道运行期调用哪个代码，因此虚函数表现为多态性时不可内联。Inline virtual唯一可以内联的时候是编译器知道所调用的对象是哪个类，只有在编译器具有实际对象而不是对象指针或引用时才会发生。

**内联函数和函数区别？**

内联函数多了关键字inline，避免了函数调用开销——内联函数不需要寻址(寻找函数入口地址)，直接在调用位置展开代码执行，减少了不必要的时间开销，以空间换时间，但内联函数有一定限制，要去代码简单，不能包含复杂的结构控制语句，否则会自动把内联函数当成普通函数来执行；

**TCP超时重传机制，时间多少？**

自适应算法来适应互联网分组传输时延的变化，TCP监视每个连接的性能(传输时延)，由此每一个TCP连接推算出合适的RTO值，当连接时延性能变化时，TCP能够自动修改RTO。RTO=RTTs + 4\*RTTd，RTTs为加权平均往返时间，RTTd是偏差的加权平均值。以后每测量到一个新的往返时间样本，就按下面的式子重新计算一次平滑往返时间SRTT： SRTT = α ×（旧SRTT）+（1-α）×（新RTT）。

**条件变量？**

**左值、右值、左值引用、右值引用、右值引用的使用场景**

在 C++ 中可以取地址的、有名字的就是左值；右值：不能取地址、没有名字的就是右值；

左值引用：对左值进行引用，左值是一个表示数据的表达式(变量名或解引用的指针)；

C++ 11 新增了右值引用（rvalue reference），这种引用可指向右值（即可出现在赋值表达式右边的值），但不能对其应用地址运算符。右值包括字面常量（C-风格字符串除外，它表示地址）、诸如 x + y 等表达式以及返回值的函数（条件是该函数返回的不是引用），右值引用使用 && 声明： int x = 10; int y = 23; int && r1 = 13; int && r2 = x + y; double && r3 = std::sqrt(2.0); 5. 右值引用的使用场景：右值引用可以实现移动语义、完美转发。

**内存对齐？**

内存对齐：现代计算机中内存空间都按字节划分，从理论上讲似乎对任何类型的变量访问可以从任何地址开始，但是实际计算机系统对基本类型数据在内存中存放的位置有限制，会要求这些数据首地址的值是某个数的倍数。

对齐的原因：平台原因-不是所有硬件平台都能访问任意地址，有的只能在某些地址上取某些特定类型数据；性能原因-数据结构应该尽可能在自然边界上对齐，否则，为了访问未对齐的内存，处理器需要两次内存访问；

规则：#pargma pack(n)，来改变这一系数。有效对齐值：是给定值 #pragma pack(n) 和结构体中最长数据类型长度中较小的那个，有效对齐值也叫对齐单位。 - 结构体第一个成员的偏移量（offset）为0，以后每个成员相对于结构体首地址的 offset 都是该成员大小与有效对齐值中较小那个的整数倍，如有需要编译器会在成员之间加上填充字节。

**shared\_ptr 怎么知道跟它共享对象的指针释放了**

shared\_ptr底层是采用引用计数方式实现的，申请堆内存空间的同时，会为其分配一整型值，当有新对象使用此堆内存时，该整形值+1，反之，每当使用此堆内存的对象被释放，该整形值-1.当堆空间整型值为0时，即表明不再有对象是用它，该堆空间就会被释放掉，仅当最后一个指针过期时才调用delete。

**进程和线程区别？**

进程是资源分配最小单位，线程是程序执行的最小单位；进程有自己独立的地址空间，每启动一个进程，系统都会为其分配地址空间。线程是进程的一个实体，一个进程至少有一个线程,同一个进程的所有线程共享所属进程的资源。线程占用的资源比线程少很多，所以创建线程和切换线程的开销相对来说很小。但多进程程序更安全，生命力更强。一个进程的死亡不会对其他进程造成影响。而一个线程死掉，（可能会锁住资源）造成整个进程都死掉了。

**什么是进程？**

进程是运行中的程序，是并发执行的程序在执行过程中分配和管理资源的基本单位，是竞争系统资源的基本单位。

进程目的是虚拟化CPU，从而支持多个进程在少量物理CPU上运行(每个进程以为自己独享CPU)。进程可以看作是一个类或一个PCB进程控制块的结构体，该结构体中主要包含了进程编号PID、进程状态、执行优先级、进程上下文、内存地址等信息。

**什么是线程？**

线程是进程内部一个执行单元，是进程内部的调度实体，比进程更小的独立运行的基本单位。每一个进程至少有一个主执行线程，无需用户主动创建而是由系统自动创建。用户需要在应用程序中创建其他线程，多线程并发的运行于同一个进程中。内核中的任务调度实际上调度的是线程，而进程只是给线程提供了虚拟内存、全局变量等资源。

**什么是协程？**

协程是一种用户态的轻量级线程，协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方。在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。进程线程都是同步机制，而协程则是异步。协程不需要多线程的锁机制。

**协程与线程的区别？**

一个线程可以有多个协程，一个进程也可以单独拥有多个协程。

线程进程都是同步机制，而协程是异步。

协程能保留上一次调用的状态到其他地方，每次过程重入时相当于进入上一次调用的状态。

线程是抢占式而协程是非抢占式的，需要用户自己释放使用权来切换到其他协程，因此同一时间只能有一个协程拥有运行权，相当于单线程能力。

协程是抽象于线程之上，线程是被分割的CPU资源，协程是组织好的代码流程。协程需要线程来承载运行，线程是协程的资源。

**进程分类？**

按资源占用：IO密集型—适合线程/协程；CPU密集型—适合多进程

按进程优先级：实时进程，普通进程；

CPU使用权：协作式调度(非抢占式)，进程一旦被占用只有当任务完成或因某些原因主动释放CPU，否则不能被其他进程夺走；抢占式调度：进程占用CPU期间可以被其他进程夺走，由操作系统决定。

**PCB（进程描述符）？**

PCB中存放着操作系统所需的用于描述进程当前情况的全部信息以及控制进程运行的控制信息和全部资源信息。PCB常驻内存，操作系统通过PCB感知到进程存在。

创建进程时创建一个PCB，进程结束时回收PCB。PCB经常被操作系统的多个模块读或修改(调度程序，资源分配程序，中断处理程序，监督和分析程序)，因此PCB常驻于操作系统专门开辟的PCB区。操作系统是根据PCB来对并发执行的进程进行控制、管理的。

**PCB组成：**

1. 进程标识信息：标识符(进程标识符、父进程标识符、用户标识符)；
2. 处理器状态信息：用户可见寄存器；控制和状态寄存器(程序计数器(下一条待取指令的地址)、状态信息(中断允许、禁用标志、执行模式))；栈指针：每个进程有一个或多个与之相关的后进先出系统栈，栈用于保存参数和过程调用或系统调用的地址。
3. 进程控制信息，调度和状态信息：进程状态：定义待调度的进程准备情况(运行态、就绪态…)、调度信息、事件(进程等待继续执行的事件标识)。

**PCB组织方式**

线性表方式(适用于进程数目不多的)、链表方式(适用于进程数目不多的)、索引表方式(适用于进程数较多的)

**进程创建方式？**

系统初始化：操作系统启动时会创建若干进程；

系统调用创建：通常一个正在运行的进程会发出系统调用来创建一个或多个新进程协同工作；(一进程读数据另一进程取走处理..)；

用户请求创建(点击图标，输入命令等)；

批处理创建

**进程终止方式：**

主动：从main()返回，程序会自动添加exit()系统调用；

主动执行exit()调用；

被动：进程收到一个不能自己处理的信号；

进程在内核态执行时产生了一个异常；

收到sigkill等终止信号；

**进程间通信方式？**

匿名管道：半双工通信，只有在父子进程和兄弟进程可用。同步的，进程尝试从空管道读数据，会被阻塞，直到有可用数据。(shell中的|就是用管道实现的)；缓存区由操作系统维护。优点是简单方便，缺点是单向通信、只能在父子进程间用到，缓冲区有限；

命名管道(FIFO)：半双工，运行无亲缘关系进程的通信。具有严格FIFO行为。优点是可实现任意关系的进程通信，缺点是长期存在于系统中，作为设备专用文件存在，使用不当容易出错，且缓存区也有限；

信号：向一个或多个进程发异步事件信号，用于通知接收进程某一事件已经发生。如sigstop(通知正在运行的进程执行关闭操作)和sigkill(通知当前进程应被杀死)。进程可以选择阻止信号也可以选择处理信号。

信号量：一个计数器，可以控制多个线程对共享资源的访问，实现多个进程的同步与互斥，而不是用于存储进程的通信数据，通常作为一种锁机制；优点是可以同步进程，缺点是信号量有限；

消息队列：消息的链接表，存放在内核中。一个消息队列由一个标识符来标识。消息队列独立于发送和接收过程，可实现消息的随机查询。不一定以先入先出的次序读取，也可按消息类型读取。

共享内存：映射一段能被其他进程访问的内存，这段共享内存由一个进程创建，多个进程都可访问。共享工作由共享内存完成，一个进程的修改对另一个进程可见。优点是无需复制，快捷、信息量大；缺点是涉及到读写操作的同步问题，且内存实体存在于计算机中，只能同一个计算机的诸多进程共享，不方便网络通信。

套接字：提供端到端的双向通信。客户/服务器（即要进行通信的进程）系统的开发工作既可以在本地单机上进行，也可以跨网络进行。也就是说它可以让不在同一台计算机但通过网络连接计算机上的进程进行通信。也因为这样，套接字明确地将客户端和服务器区分开来。

**进程私有和共享的资源**

私有：地址空间，堆，全局变量，栈，寄存器

共享：代码段，公共数据，进程目录，进程ID；

**线程私有和共享的资源：**

私有：线程栈，寄存器，程序计数器；

共享：堆，地址空间，全局变量，静态变量；

**多线程设计的注意点**

在CPU密集型，IO密集型，以及充分利用多核CPU提升程序性能上找到一个平衡点；

尽可能支持规模更大的线程数量；

减少线程在用户态和内核态中切换带来的开销；

**并行与并发区别？**

并发：一段时间内很多线程或进程在执行，但任何时间点上只有一个在执行，多个进程或线程争抢时间片轮流执行；

并行：一段时间内和一个时间点上多个进程或线程在执行。

**线程模型？**

一对一，多对一，多对多(表示了用户态线程和内核态线程间的联系，影响并发规模及操作成本)；

一对一：每个用户线程映射单独的内核线程；优点是用户线程和内核线程一致，可扩展性好；缺点是系统会限制内核线程数量，因此用户线程数量会受限制，内核上下文切换开销大；

多对一：多个用户态线程映射给单一的内核线程；优点是上下文切换高效(线程的创建、同步、销毁、调度完全在用户态中完成，对内核透明，无需切换到内核态)，线程数量无限制(创建不受内核资源限制)；缺点是可扩展性差，且一个用户线程阻塞，其他线程也无法执行。

多对多：N个用户态线程映射到M个内核态线程（N>M）；解决了可扩展性问题和线程过多问题；缺点：管理复杂

**线程间通信方式**

锁机制：互斥锁，读写锁，自旋锁，条件变量；

互斥锁：以排他方式防止数据结构被并发修改；

读写锁：允许多线程同时共享数据，但对读写操作是互斥的；

自旋锁：与互斥锁类似，但互斥锁是当资源被占用，申请者进入睡眠状态，而自旋锁则循环检测保持者是否已经释放锁；

条件变量：可以以原子方式阻塞进程，直到某个特定条件为真为止。对条件的测试在互斥锁保护下进行，因此条件变量始终与互斥锁一起用；

信号量机制：无名线程信号量；命名线程信号量；

信号机制：类似进程间信号处理；

屏障：运行每个线程等待，直到所有合作线程到达某一点，然后从该点继续执行。

**多进程还是多线程？区别？**

**数据共享：**多进程数据共享复杂，需用IPC，但数据分开，同步简单；多线程共享进程数据，但同步复杂；

**内存**：进程占内存多；

**CPU**：进程的CPU利用率低；线程利用率高；

**创建、销毁成本**：进程的切换复杂，创建、销毁复杂，切换速度慢；

**可靠性**：进程间不会互相影响，但一个线程挂掉则整个进程挂掉；

**为什么有了进程又要创建一个线程的概念？**

多线程间会共享同一块地址空间和所有可用数据的能力，这是进程不具备的；

线程比进程更轻量级，创建一个线程要比创建一个进程快10-100倍；

若存在大量计算和大量IO处理，多线程能在这些活动中彼此重叠进行，加快程序执行速度。

**多线程是否比单线程性能好？**

多线程比单线程，省去了等待IO操作的时间，但是多了CPU上下文切换，IO操作远比CPU上下文切换要耗费时间。因此在IO操作情况下，使用多线程性能会更好。

操作系统角度，并不是核越多越好，管理起来越麻烦，会有更多多余的开销；在应用程序角度，线程数也并不是越多越好，线程的调度也都会有开销。

**什么是死锁？**

当进程需要以独占方式访问资源时，可能会发生死锁。死锁是指一个进程等待一个已被占用且不释放的资源造成的一种僵局。若无外力作用则无法推进。

产生原因：系统提供的资源数比要求该资源的进程数要少；

原因分类：资源竞争，进程推进顺序不合理。

**产生死锁的四个必要条件：**

互斥 (涉及的资源不能共享，只能有一个进程使用)、不可抢占(资源未使用完毕之前不能被其他进程强行夺走)、占有并等待(进程每次申请它需要的一部分资源，在等待新资源时，进程继续占用已分配的资源)、环路(存在一种进程首尾相连的循环链，每一个进程都在等待下一个进程所持有的资源造成这组进程永远处于互相等待的状态)。

**预防死锁：**

 破坏“不可剥夺”条件：一个进程不能获得所需要的全部资源时便处于等待状态，等待期间他占有的资源将被隐式的释放重新加入到系统的资源列表中，可以被其他的进程使用，而等待的进程只有重新获得自己原有的资源以及新申请的资源才可以重新启动。

 破坏”请求与保持条件“：第一种方法静态分配即每个进程在开始执行时就申请他所需要的全部资源。第二种是动态分配即每个进程在申请所需要的资源时他本身不占用系统资源。

 破坏“循环等待”条件：采用资源有序分配其基本思想是将系统中的所有资源顺序编号，将紧缺的，稀少的采用较大的编号，在申请资源时必须按照编号的顺序进行，一个进程只有获得较小编号的进程才能申请较大编号的进程

**避免死锁：**

引入一些算法来防止系统进入不安全状态；具体策略：若进程请求的资源导致死锁则系统拒绝启动该进程；若对一个资源的分配导致下一步死锁，系统拒绝本次分配；(银行家算法：新进程进入系统必须说明对各类资源的最大需求量，不能超过资源总数，否则不接纳该进程；进程申请一组资源，检测进程对各类资源的最大需求量，若现存各类资源数量满足则分配；否则进程必须等待，直到其他进程释放足够的资源；进程需在一定时间内无条件归还它申请的全部资源)

**死锁的检测及解除：**

判断死锁：资源分配路径中无环路，则系统不会出现死锁；资源分配路径中存在环路，则系统中可能出现死锁；环路中每个资源类都只有一个资源，则系统存在死锁；环路中每个资源类资源个数不止一个，则环路的存在是产生死锁的必要非充分条件；

解除死锁：

资源剥夺(剥夺陷入死锁的进程占用的资源，但不撤销此进程，再将这些资源分配给需要的进程，直至死锁解除)

进程撤销(一次性撤销陷入死锁的所有进程，回收占用的资源，等死锁解除再重新运行进程；或逐个撤销死锁的进程，依次回收资源并重新分配，直至死锁解除，可以优先撤销优先级低，预计执行时间长、CPU消耗时间少的进程)

进程回退(让所有进程回退到系统保存的检查点，要求系统建立并保存检查点，建立回退机制)

系统重启：结束所有进程并重启操作系统。

**什么是虚拟内存？**

传统存储管理方式：作业必须一次全部加载到内存中，方可运行。当作业很大，就无法运行。而且多道作业运行时，内存不足容纳所有作业，导致多道程序性能下降。操作系统引入了虚拟内存的概念，利用计算机的空间局部性和时间局部性原理，将程序的一部分装入内存运行，其余部分留在外存，等需要的时候再将外存的程序装入内存继续运行。虚拟内存好像给用户提供了一个比实际内存大得多的存储器。叫，虚拟存储器，大小由计算机地址结构决定。它使得[应用程序](https://baike.baidu.com/item/%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F/5985445" \t "_blank)认为它拥有连续的可用的[内存](https://baike.baidu.com/item/%E5%86%85%E5%AD%98/103614" \t "_blank)（一个连续完整的[地址空间](https://baike.baidu.com/item/%E5%9C%B0%E5%9D%80%E7%A9%BA%E9%97%B4/1423980" \t "_blank)），而实际上，它通常是被分隔成多个[物理内存](https://baike.baidu.com/item/%E7%89%A9%E7%90%86%E5%86%85%E5%AD%98/2502263" \t "_blank)碎片，还有部分暂时存储在外部[磁盘存储器](https://baike.baidu.com/item/%E7%A3%81%E7%9B%98%E5%AD%98%E5%82%A8%E5%99%A8/2386684" \t "_blank)上，在需要时进行[数据交换](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E4%BA%A4%E6%8D%A2/1586256" \t "_blank)。

虚拟内存实现方式：请求分页，请求分段，请求段页式存储管理。请求分页存储管理中，将虚拟地址内存空间划分位大小相等的页块，同时内存地址空间，也划分位等大小的页块。系统维持一个页表，存储这虚拟页号到物理快块号的映射。程序中的逻辑地址由两部分组成：页号 P 和页内位移量W。相邻的页面在内存中不一定相邻，即分配给程序的内存块之间不一定连续。逻辑地址转化为物理地址时，根据页表将页号转化为块号，块号\*块大小加上页内偏移得到物理地址。

优点：

程序可以大于物理内存：虚拟内存结合了磁盘和物理内存优势为进程提供速度快且容量大的存储；

让系统运行更多程序实例：每个用户程序可以占用较少物理内存，因此可以同时运行更多程序，增加CPU利用率和吞吐量，但没有增加响应时间、周转时间；

用户程序运行更快：加载或交换每个用户程序到内存所需的IO会更少，用户程序会运行更快；

简化了编程任务：无需关注内存管理；

提高系统安全性：虚拟内存可以控制进程对物理内存的访问，隔离不同进程的访问权限；

缺点：

若内存严重不足，处理大文件会频繁引起内存和磁盘的交换，降低了系统性能；在多程序切换会花费更多时间；

**什么是缺页中断？**

在分配时只是建立了进程虚拟地址空间，并没有分配虚拟内存对应的物理内存。当进程访问这些没有建立映射关系的虚拟内存时，处理器自动触发一个缺页异常。需要将页由外存调入内存。如果内存已满，采用页面置换[算法](https://www.nowcoder.com/jump/super-jump/word?word=%E7%AE%97%E6%B3%95" \t "_blank)将老的淘汰，载入新的。页面置换[算法](https://www.nowcoder.com/jump/super-jump/word?word=%E7%AE%97%E6%B3%95" \t "_blank)常见的有 FIFO,LRU。

**内存页面置换算法？**

最佳页面置换法：置换未来最长时间不访问的页面(需计算内存中每个逻辑页面的下一次访问时间，选择未来最长时间不访问的页面)，是用来衡量算法效率，现实中很难实现；

先进先出置换法：选择内存中驻留时间最长的页面进行置换；

最近最久未使用置换法(LRU)：发生缺页时，选择最长时间未访问的页面进行置换(需维护一个所有页面的链表，最近最少使用的页面在表尾，最多使用的在表头，开销大，费时)；

最不常用置换法(LFU)：发生缺页中断时，选择访问次数最少的那个页面并将其淘汰；实现方式：对页面设置一个计数器，每一个页面被访问，该页面的访问计数器加1，发生缺页中断时，淘汰计数器值最小的那个页面；硬件成本较高，且只考虑频率而没考虑时间问题；LFU 算法只考虑了频率问题，没考虑时间的问题，比如有些页面在过去时间里访问的频率很高，但是现在已经没有访问了，而当前频繁访问的页面由于没有这些页面访问的次数高，在发生缺页中断时，就会可能会误伤当前刚开始频繁访问，但访问次数还不高的页面。这个问题的解决的办法是：定期减少访问的次数，比如当发生中断时，把过去时间访问的页面的访问次数除以 2，也就说，随着时间的流失，以前的高访问次数的页面会慢慢减少，相当于加大了被置换的概率。

**用户态和内核态？**

**1.用户空间和内核空间**

操作系统将虚拟地址空间分为用户空间和内核空间(L系统3：1，W系统2：2)；

区分用户/内核空间原因：开发者直接使用CPU指令集，若使用不规范，可能直接影响操作系统内核及其他所有正在运行的程序，影响计算机系统运行，操作系统需要维护自身健壮性，所以需要屏蔽开发人员对硬件操作的可能，不让开发人员直接操作这些CPU指令集。

**2.内核态与用户态**

运行于用户态的进程可以执行的操作和访问的资源都会受到极大的限制，而运行在内核态的进程则可以执行任何操作并且在资源的使用上没有限制。很多程序开始时运行于用户态，但在执行的过程中有时候会需要切换到内核态执行。

**内核态：**

内核空间：一个任务执行系统调用而陷入内核代码中执行时，则称进入了内核运行态，系统运行在内核空间；

指令级别：处于最高的特权级(0级)内核代码中运行；

访问权限：可以执行任何指令，所运行的代码也不收任何限制，可自由访问任何有效地址，也可以直接进行端口访问；

进程栈：内核代码会使用当前进程的内核栈，每个进程都有自己的内核栈。

**用户态：**

用户空间：进程执行用户自己的代码时，称其处于用户运行态，此时系统运行在用户空间；

指令级别：处理器在特权级最低的用户代码中运行。当正在执行用户程序而突然被中断程序中断时，用户程序也可象征性称为处于进程的内核态；

访问权限：被执行的代码要经过CPU诸多检查，只能访问映射其地址空间的页表项所规定的用户态可访问页面的虚拟地址，且只能对可访问端口直接访问；

栈：使用进程自己的进程栈，因为中断处理程序将使用当前进程的内核栈。

**3.用户态和内核态的切换**

**用户态-内核态：**

系统调用(主动切换，如fork(创建新进程)，核心是使用操作系统为用户特别开放的一个中断来实现)；

异常：CPU执行用户态进程时发生了一些未预知的异常，当前运行进程会切换到处理此异常的内核相关进程中(如缺页异常)。

设备中断：CPU执行用户态进程，外围设备完成用户请求的操作后，会向CPU发出相应中断信号

**网页如果同时接到很多请求，会怎么样？**

当一个网页同时接收到大量请求时，可能会导致以下一些情况发生：

服务器负载过高：大量的请求会导致服务器的负载急剧增加，如果服务器未能及时处理这些请求，可能会导致性能下降甚至崩溃。

响应时间延长：服务器在处理大量请求时，可能会导致每个请求的响应时间延长，从而影响用户体验。

资源竞争：大量并发请求会导致服务器内部的资源（如数据库连接、内存、CPU 等）出现竞争，进而可能导致资源耗尽或者争夺失败。

请求队列阻塞：如果服务器未能及时处理所有请求，可能会导致请求队列中的请求堆积，甚至导致拥塞，这会使得新的请求无法被及时处理。

为了应对大量请求的情况，可以考虑以下解决方案：

水平扩展：通过增加服务器实例的数量，将负载分散到多台服务器上，以提高系统整体的处理能力。

负载均衡：使用负载均衡器将请求分发到多台服务器上，以避免单台服务器过载。

缓存：利用缓存技术减轻服务器的压力，例如将静态内容缓存，减少对数据库和其他后端服务的请求。

异步处理：将一些耗时的操作改为异步处理，比如通过消息队列等方式进行异步处理，提高系统的并发能力。

优化代码和数据库查询：对代码和数据库查询进行优化，减少不必要的计算和数据库访问，提高系统的响应速度。

综上所述，面对大量请求的情况，需要根据实际情况采取相应的措施，以确保系统能够有效地应对并发压力，并保持良好的性能和稳定性。

**手机访问一个网页的过程？**

当手机访问一个网页时，通常会经历以下过程：

输入 URL：用户在手机浏览器中输入要访问的网页的 URL（统一资源定位符）或通过搜索引擎进行搜索。

DNS 解析：手机浏览器向 DNS 服务器发送请求，将域名解析为对应的 IP 地址，以便能够连接到服务器。

建立 TCP 连接：手机浏览器与 web 服务器之间建立 TCP 连接，这是一种可靠的传输协议，确保数据能够安全地在两者之间传输。

发起 HTTP 请求：浏览器向 web 服务器发送 HTTP 请求，请求获取网页的内容，请求中包含了用户所需的信息，比如浏览器版本、支持的压缩算法等。

服务器处理请求：web 服务器接收到请求后，会根据请求内容执行相应的处理，比如读取数据库、处理业务逻辑等。

构建网页：服务器根据请求处理结果，构建相应的 HTML 页面，并可能会包括其他资源文件，比如 CSS、JavaScript、图片等。

发送响应：服务器将构建好的网页内容通过 TCP 连接发送给手机浏览器。

浏览器渲染页面：手机浏览器接收到服务器返回的网页内容后，开始解析 HTML、CSS 和 JavaScript，并渲染出最终的页面供用户查看。

加载其他资源：网页中可能包含其他资源文件，浏览器会根据需要加载这些资源文件，比如样式表、脚本文件、图片等。

交互操作：用户可以与页面进行交互操作，点击链接、提交表单等。

总的来说，手机访问网页的过程涉及到 DNS 解析、建立 TCP 连接、发送 HTTP 请求、服务器处理请求、构建页面、发送响应、浏览器渲染等多个步骤，通过这些步骤完成数据的传输和页面的展示。

**讲到接收方 socket 时，有问有没有优化方案？**

当涉及到接收方的 socket 时，可以考虑一些优化方案来提高系统的性能和效率，以下是一些常见的优化方案：

使用非阻塞 I/O：将 socket 设置为非阻塞模式，可以使程序不必等待数据的到达而继续执行其他任务，提高系统的并发处理能力。

使用事件驱动模型：采用事件驱动的方式处理 socket，如使用 epoll、kqueue 等机制，可以更有效地管理大量的并发连接。

使用线程池或者异步处理：通过使用线程池或异步处理技术，可以避免为每个连接创建一个新线程，提高系统的资源利用率。

调整接收缓冲区大小：根据实际情况调整 socket 接收缓冲区的大小，以适应不同的网络环境和数据传输需求。

使用快速用户空间数据包处理技术：如 DPDK（Data Plane Development Kit）等技术，可以将数据包的处理移到用户空间，减少内核态和用户态的切换，提高数据包处理效率。

优化数据处理算法：对接收到的数据进行高效的处理，避免不必要的数据拷贝和处理操作，提高数据处理的效率。

合理设置 TCP 参数：根据实际情况设置 TCP 的参数，如调整 TCP 缓冲区大小、超时时间等，以提高数据传输的效率和稳定性。

避免频繁的连接和断开：尽量避免频繁地建立和断开连接，可以减少连接管理的开销，提高系统的性能。

通过采取这些优化方案，可以有效地提高系统对 socket 的处理效率和性能，更好地应对高并发和大规模数据传输的需求。

**只能一个 socket 一个进程吗？**

不一定。在处理网络通信时，并不是每个 socket 都需要对应一个单独的进程。实际上，可以通过多种方式来管理多个 socket 连接，而不是为每个 socket 连接创建一个新的进程。

以下是一些常见的技术和方法来处理多个 socket 连接：

多线程：可以使用多线程来处理多个 socket 连接，每个线程负责处理一个或多个 socket 连接。这样可以实现并发处理，提高系统的性能。

多路复用：使用多路复用技术如 select、poll 或 epoll，可以在一个进程中管理多个 socket 连接，通过监听多个 socket 的事件，实现对多个连接的有效管理和处理。

线程池：可以创建一个线程池来处理多个 socket 连接，避免频繁地创建和销毁线程，提高系统的资源利用率。

异步 I/O：使用异步 I/O 模型，比如使用 epoll 或者异步 I/O 操作，可以在单个线程中处理多个 socket 连接的读写操作，提高系统的并发能力。

进程池：使用进程池来处理多个 socket 连接，可以减少进程创建和销毁的开销，提高系统的效率。

总的来说，并不是每个 socket 都需要对应一个单独的进程，可以通过合适的技术和方法来管理和处理多个 socket 连接，实现高效的网络通信和数据处理。选择适合场景的技术和方法可以提高系统性能、降低资源消耗，并更好地满足实际需求。

**线程怎么共享内存？**

线程可以通过共享内存来实现线程间的数据共享。在多线程编程中，线程之间可以访问同一块内存区域，从而实现数据的共享。以下是一些常见的方法来实现线程间的内存共享：

全局变量：定义全局变量，所有线程都可以访问和修改这些全局变量。需要注意线程安全性，通过锁或其他同步机制来保护全局变量的访问。

静态变量：静态变量在程序运行期间一直存在，可以被多个线程共享。但同样需要考虑线程安全性。

动态内存分配：通过动态内存分配（如malloc、free）来分配一块共享内存，多个线程可以访问该内存区域。

线程参数传递：创建线程时可以传递参数，这些参数可以是指向共享内存的指针，从而实现线程间的数据共享。

信号量和互斥锁：使用信号量和互斥锁等同步机制来保护共享数据的访问，防止多个线程同时修改造成数据不一致。

条件变量：条件变量可以用来在线程之间传递信号，用于线程同步和通信，实现对共享数据的访问控制。

线程局部存储：每个线程拥有自己的局部存储空间，可以在其中存储线程私有的数据，避免线程间的数据共享问题。

总的来说，线程间共享内存需要注意线程安全性和同步机制的设计，确保多个线程对共享数据的访问不会造成数据竞争和不一致性。合理地设计和管理共享内存，可以实现线程间高效的数据共享和通信。

**怎么避免多线程内存乱了？**

避免多线程内存混乱（Memory Corruption）是非常重要的，可以通过以下方法来确保多线程程序中的内存安全性：

使用同步机制：使用互斥锁（Mutex）、信号量（Semaphore）、条件变量（Condition Variable）等同步机制来保护共享数据的访问，确保在同一时间只有一个线程能够访问共享内存，避免多个线程同时修改数据造成混乱。

避免数据竞争：尽量避免多个线程对同一块内存区域进行读写操作，如果必须同时访问，则需要使用同步机制来保护数据的访问顺序。

避免死锁：在使用同步机制时，要注意避免死锁的发生，即多个线程相互等待对方释放锁而无法继续执行的情况。

合理设计数据结构：设计数据结构时要考虑线程安全性，避免多个线程同时访问同一数据结构造成数据不一致。

使用原子操作：对于简单的数据操作，可以使用原子操作来确保操作的原子性，避免并发访问导致的数据混乱。

避免指针错误：尽量避免在多线程程序中使用裸指针，尤其是在多线程环境下容易出现野指针或者指针指向已释放的内存的情况。

内存管理规范：严格管理内存的分配和释放，避免内存泄漏和悬空指针的问题。

使用工具检测问题：可以使用内存检测工具、静态代码分析工具等来检测和修复潜在的内存混乱问题。

通过以上方法，可以有效地避免多线程程序中出现内存混乱的情况，确保程序的正确性和稳定性。同时，良好的代码设计和规范的编程实践也是确保多线程程序内存安全的关键。

**原子变量怎么实现的？**

原子变量是一种特殊的变量类型，支持原子操作，即在多线程并发访问时能够保证操作的原子性，避免数据竞争和混乱。原子变量的实现通常依赖于硬件的原子指令或者操作系统提供的原子操作接口。

在多线程编程中，原子变量通常通过以下几种方式来实现：

硬件支持：一些现代处理器提供原子指令（Atomic Instructions），如Compare-and-Swap (CAS)、Fetch-and-Add等，可以在单个指令中完成对内存的读取、修改和写入，从而实现原子操作。这些原子指令能够确保在执行期间不会被其他线程打断，保证操作的原子性。

操作系统提供的原子操作接口：操作系统提供了一些原子操作的接口，用于实现原子变量。例如，在Linux系统中，可以使用内核提供的原子操作函数（如atomic\_t、atomic\_add等）来操作原子变量。

编译器提供的原子操作支持：一些编译器（如GCC、Clang）提供了内置的原子操作函数或者语义扩展，可以直接在代码中使用这些函数来实现原子操作。

使用锁实现：在一些极端情况下，原子变量的实现可能会借助于锁来确保操作的原子性。这种方式虽然效率较低，但在没有硬件原子指令支持的情况下仍然可以实现原子操作。

总的来说，原子变量的实现涉及到硬件支持、操作系统提供的接口以及编译器的优化等多方面因素。使用原子变量能够简化多线程编程中的同步和互斥操作，提高程序的并发性能和可靠性。

**Https 怎么加密的？**

HTTPS（Hypertext Transfer Protocol Secure）是在HTTP基础上加入了安全套接层（SSL/TLS）的协议，用于在网络传输过程中对数据进行加密和身份验证。下面是HTTPS的加密过程：

握手阶段：

客户端发送一个加密连接请求给服务器，并携带自己支持的加密算法列表。

服务器从客户端发来的加密算法列表中选择一种加密算法，并发送自己的证书给客户端。

客户端验证服务器的证书是否合法，包括验证证书的有效性、签名的合法性和证书链的完整性。

客户端生成一个随机的对称密钥，使用服务器的公钥加密后发送给服务器。

密钥交换阶段：

服务器使用自己的私钥解密客户端发来的对称密钥。

服务器和客户端都获得了同样的对称密钥，用于后续的数据加密和解密。

加密通信阶段：

客户端将HTTP请求的内容使用对称密钥进行加密。

服务器使用对称密钥解密客户端发送的加密数据。

服务器将HTTP响应的内容使用对称密钥进行加密。

客户端使用对称密钥解密服务器发送的加密数据。

在整个过程中，使用了非对称加密和对称加密两种加密算法：

非对称加密算法（公钥加密）用于在握手阶段进行身份验证和密钥交换，确保通信双方安全可靠。

对称加密算法用于实际的数据传输阶段，提供高效的加密和解密性能。

通过使用SSL/TLS协议，HTTPS能够保护数据在传输过程中的机密性和完整性，防止被窃听和篡改。同时，服务器的证书也能够验证服务器的身份，确保客户端与正确的服务器建立连接，防止中间人攻击。

**程序申请内存的过程？**

程序申请内存的过程通常涉及操作系统的内存管理机制。下面是一般情况下程序申请内存的过程：

程序发起内存申请请求：

当程序需要内存空间来存储数据结构、变量或者动态分配内存时，会向操作系统发起内存申请请求。

内存分配器分配内存：

操作系统的内存分配器会根据程序的申请大小，从进程的内存空间中分配一块合适大小的内存空间。

内存分配器会维护一份内存空闲块链表或其他数据结构，用于记录可用的内存块。

内存映射和页表更新：

操作系统会将分配给程序的内存空间映射到程序的虚拟地址空间中，通常通过页表实现虚拟地址到物理地址的映射关系。

如果需要在新分配的内存空间中写入数据，操作系统会将相应的页面标记为可写，并进行必要的权限设置。

返回内存地址：

内存分配器将分配的内存空间的起始地址返回给程序，程序可以通过该地址来访问和操作新分配的内存空间。

程序使用内存：

程序可以通过返回的内存地址来读取或写入数据，对新分配的内存空间进行操作。

内存释放：

当程序不再需要使用已经分配的内存空间时，通常会调用相应的内存释放函数（如free()）来告知内存分配器释放该内存空间。

内存分配器会回收被释放的内存空间，并将其标记为可用，以便后续程序再次申请使用。

总的来说，程序申请内存的过程涉及到内存分配器的管理、虚拟地址空间的映射、权限设置等操作系统的内存管理机制。正确地管理内存的申请和释放对程序的性能和稳定性至关重要，避免内存泄漏和内存访问越界等问题。

**页表为什么要分多级？**

页表之所以要分多级主要是出于对内存管理的效率和灵活性考虑。以下是为什么页表要分多级的几个原因：

节约内存空间：

在大内存地址空间下，如果采用单一级的页表结构，页表可能会非常庞大，占用大量内存空间。

通过采用多级页表结构，可以将一个大的页表分解成多个更小的页表，从而节约内存空间。

快速查找：

在单一级页表中，需要线性搜索整个页表来找到对应的页表项，当页表非常大时会导致查找效率低下。

多级页表结构通过分层查找的方式，可以将查找过程分解成多个步骤，每一级页表只需查找较小的范围，提高了查找效率。

实现虚拟内存的部分映射：

多级页表结构可以更灵活地支持虚拟内存的部分映射，即只有部分虚拟地址空间被映射到物理内存中。

通过在某些层级设置空指针或标记位，可以实现虚拟地址空间的部分映射，以节省物理内存空间。

支持大内存地址空间：

针对64位系统等具有大内存地址空间的情况，多级页表结构能更好地支持大范围的地址映射。

多级页表可以根据需要动态调整层级结构，以适应不同大小的内存地址空间的管理需求。

总的来说，多级页表结构能够提高内存管理的效率、节约内存空间，并支持更灵活的虚拟内存管理，适应不同规模和需求的系统环境。因此，多级页表结构在现代操作系统中被广泛应用于内存管理中。

**讲讲 OOM？**

OOM（Out of Memory）是指在计算机系统中，当系统无法满足进程或操作系统的内存需求时发生的情况。这种情况可能导致系统或进程的异常行为，甚至导致系统崩溃。

以下是关于OOM的一些重要信息：

原因：

OOM通常发生在系统内存资源不足时。这可能由于系统负载过重、内存泄漏、程序错误导致的内存占用过高等原因引起。

当系统无法满足新的内存分配请求或者无法继续执行现有的内存分配请求时，就会发生OOM。

影响：

当系统发生OOM时，可能导致正在运行的程序出现异常，甚至导致部分或整个系统崩溃。

对于操作系统而言，OOM也可能导致系统无法继续正常工作，从而需要进行重启或者紧急处理。

OOM处理：

在Linux等操作系统中，通常会有OOM Killer机制，用于在系统内存不足时选择终止一些进程，以腾出内存空间来保证系统的正常运行。OOM Killer会根据一定的策略选择要终止的进程。

对于应用程序开发者来说，可以通过合理管理内存、避免内存泄漏等手段来减少发生OOM的可能性。

监控和调优：

系统管理员和开发人员可以通过监控系统的内存使用情况，及时发现系统内存不足的问题，并采取相应的调优措施，如增加物理内存、优化程序内存占用等，以避免OOM的发生。

总的来说，OOM是一个在计算机系统中可能发生的严重问题，需要系统管理员和开发人员密切关注系统内存使用情况，及时处理和预防OOM问题的发生。