1.为什么说TCP是可靠的服务?

> （1）、TCP有一套连接管理的机制：收发双方通信前要建立连接，即进行三次握手，然后才进行应用层的通信；应用层通信完成之后要释放TCP连接，即进行四次握手。
>
> （2）、在通信的过程中，对每一个发出的数据包都要进行确认，在规定时间内收不到确认的话要重发该数据包。
>
> （3）、TCP还通过滑动窗口的机制来进行流量控制，使收发双方的通信速率得到匹配。

2.如何理解面向对象?

> 面向对象就是看事物的一种方式，一种视觉角度，分析方式，我们可以把任意一个事物看成是一个对象，分析它身上具备的主要特征，这个就是面向对象思维；
>
> 我们把事物当做一个对象，分析它的主体特征，注意力聚焦在主体特征，而不是聚焦于更细节的内容及实现，如我们把手机当做一对象，现在我们要开发手机，手机是个对象，它有哪些特征呢，比如颜色、手机壳、打电话、发短信等等，而我们不会去想到 发短信具体是怎么实现的，集成电路如何设计，如何接收信号，电磁波如何发出，甚至更加细节的问题；
>
> 这样做有利于我们宏观掌控设计手机的主体功能，在整体上思考设计方针的准确性，等我们确定了每一个功能模块，主体特征后，我们在考虑如何一步一步去实现这个主体特征；
>
> 把事物当做对象，宏观分析它的具备的主体特征，这已经是面向对象的思维了。

3.为什么要用到线程池?

> 大量的线程也会抢占cpu的资源，cpu不停的在各个线程上下文切换中,反而没有时间去处理线程运行的时候该处理的任务。
>
> 因此，为了避免频繁的创建和销毁线程，让创建的线程进行复用，就有了线程池的概念。 线程池里会维护一部分活跃线程，如果有需要，就去线程池里取线程使用，用完即归还到线程池里，免去了创建和销毁线程的开销，且线程池也会线程的数量有一定的限制。

4.如何避免死锁呢?

> - 避免多次锁定，多检查;
> - 对共享资源访问完毕之后，一定要解锁，或者在加锁的使用trylock;
> - 如果程序中有多把锁，可以控制对锁的访问顺序(顺序访问共享资源，但在有些情况下是做不到的)，另外也可以在对其他互斥锁做加锁操作之前，先释放当前线程拥有的互斥锁;

> - 项目程序中可以引入一些专门用于死锁检测的模块;

5.select,poll,epoll的区别?

1、支持一个进程所能打开的最大连接数

> **select**:单个进程所能打开的最大连接数有上限,当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进一步的测试。
>
> **poll** : poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的。
>
> **epoll**:虽然连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接。

2、FD剧增后带来的IO效率问题

> **select**:因为每次调用时都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的“线性下降性能问题”。
>
> **poll**:同上
>
> **epoll**:因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。

3、消息传递方式

> **select**:内核需要将消息传递到用户空间，都需要内核拷贝动作
>
> **poll**:同上
>
> **epoll** : epoll通过内核和用户空间共享一块内存来实现的。

6.epoll的效率一定高于select和poll吗?

在选择select,poll,，epoll时要根据具体的使用场合以及这三种方式的自身特点。

1、表面上看epol的性能最好，但是在连接数少并且连接都十分活跃的情况下，select和poll的性能可能比epoll好，毕竟epoll的通知机制需要很多函数回调。

> select，poll实现需要自己不断轮询所有fd集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll其实也需要调用epoll_wait不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪fd放入就绪链表中，并唤醒在epoll_wait中进入睡眠的进程。虽然都要睡眠和交替，但是select和poll在"醒着"的时候要遍历整个fd集

> 合，而epoll在°醒着"的时候只要判断一下就绪链表是否为空就行了，这节省了大量的CPU时间。这就是回调机制带来的性能提升。

2、select低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善

> select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把current往设备等待队列中挂一次，而epoll只要一次拷贝，而且 把current往等待队列上挂也只挂一次（在epoll_wait的开始，注意这里的等待队列并不是设备等待队列，只是一个epol内部定义的等待队列)。这也能节省不少的开销。

7.单例模式的应用场景?

> 1. Windows的Task Manager（任务管理器）就是很典型的单例模式（这个很熟悉吧），想想看，是不是呢，你能打开两个windows task manager吗？ 不信你自己试试看哦~
> 2. windows的Recycle Bin（回收站）也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。
> 3. 网站的计数器，一般也是采用单例模式实现，否则难以同步。
> 4. 应用程序的日志应用，一般都何用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。

总结以上，不难看出, 单例模式应用的场景一般发现在以下条件下：

> （1）资源共享的情况下，避免由于资源操作时导致的性能或损耗等。如上述中的日志文件，应用配置。
>
> （2）控制资源的情况下，方便资源之间的互相通信。如线程池等。

8.如何理解消息队列(MQ)?

> **首先, 将 MQ 掰开了揉碎了来看，都是「一发一存一消费」，再直白点就是一个「转发器」。**
>
> 生产者先将消息投递一个叫做「队列」的容器中，然后再从这个容器中取出消息，最后再转发给消费者，仅此而已。
>
> [**消息**]：就是要传输的数据，可以是最简单的文本字符串，也可以是自定义的复杂格式（只要能按预定格式解析出来即可）。
>
> [**队列**]：大家应该再熟悉不过了，是一种先进先出数据结构。它是存放消息的容器，消息从队尾入队，从队头出队，入队即发消息的过程，出队即收消息的过程。

目前，MQ 的应用场景非常多，大家能倒背如流的是：**系统解耦**、**异步通信**和**流量削峰**。除此之外，还有延迟通知、最终一致性保证、顺序消息、流式处理等等。

举例:

> 假设我们有一个系统A能够产生一系列消息M, 而我们有两个系统a和b负责处理A产生的消息M, 因此可以说系统a和b是依赖于系统A的, 而如果我们在二者之间加入消息队列MQ,那么A会把产生的消息放进消息队列MQ, 而a和b则从消息队列MQ获取消息M来进行处理;

- 现在, A与a和b之间多了一个中间节点「队列」进行消息转储，将同步变成了**异步**, 减少了整体耗时, 提升了系统的吞吐量;
- 对A与a和b之间的关系也进行了**解耦**; 即a和b不再依赖与系统A, 转而依赖消息队列MQ了, 有利于我们程序的后续扩展和修改;
- 同时呢, 由于队列本身是可以储存消息的, 那么对于超出系统承载能力的场景, MQ也可以充当"漏斗"进行限流保护, 即所谓的**流量削峰**。

9.Mysql中的左连接和右连接是什么?

> **内连接**: 获取两个表字段关系匹配的记录,必须是两个的字段匹配都满足;
>
> **左连接**: 获取左表(即LEFT JOIN左侧的表)所有记录,即使右表(即ON右侧的表)没有匹配的记录,关联上就展示,没关联上就展示null;
>
> **右连接**: 获取右表(即RIGHT JOIN右侧的表)所有记录,即使左表没有匹配的记录,关联上就展示,没关联上就展示null;

10.对于性能优化,你有什么看法?

> 1. 避免冗余的变量拷贝
> 2. 避免频繁的内存申请、释放操作
> 3. 避免冗余的循环计算
> 4. 空间换时间
> 5. 位运算代替乘除法

11.为什么要线程同步?怎么实现?

避免竞态资源的访问冲突: 当我们有多个线程要同时访问一个变量或对象时，如果这些线程中既有读又有写操作时，就会导致变量值或对象的状态出现混乱，从而导致程序异常。

> 常用的线程同步方式有四种：互斥锁、读写锁、条件变量、信号量。

12.虚函数的作用是什么?

> 实现多态性, 多态性本身是为了将接口与实现进行分离, 使代码易扩展, 易修改;

13.如何解决网络通信中的粘包问题?

> 我们通常使用**添加包头**的方式轻松地解决掉这个问题。
>
> 关于数据包的包头大小可以根据自己的实际需求进行设定，这里没有啥特殊需求，因此规定包头的固定大小为4个字节，用于存储当前数据块的总字节数。

14.使用多进程与多线程，请问有什么区别?

> **根本区别**：用多进程每个进程有自己的地址空间(address space)，线程则共享地址空间, 所有其它区别都是由此而来的：
>
> 1. **速度**：线程产生的速度快，线程间的通讯快、切换快等，因为他们在同一个地址空间内。
> 2. **资源利用率**：线程的资源利用率比较好也是因为他们在同一个地址空间内。
> 3. **同步问题**：线程使用公共变量/内存时需要使用同步机制还是因为他们在同一个地址空间内。

15.谈谈进程间通信的方式.

> 1. **消息传递**：消息传递是进程间实现通信和同步等待的机制，使用消息传递，进程间的交流不需要共享变量，直接就可以进行通信；消息传递分为发送方和接收方
> 2. **先进先出队列**：先进先出队列指的是两个不相关联进程间的通信，两个进程之间可以彼此相互进程通信，这是一种全双工通信方式
> 3. **管道**：管道用于两个相关进程之间的通信，这是一种半双工的通信方式，如果需要全双工，需要另外一个管道。
> 4. **直接通信**：在这种进程通信的方式中，进程与进程之间只存在一条链接，进程间要明确通信双方的命名。
> 5. **间接通信**：间接通信是通信双方不会直接建立连接，而是找到一个中介者，这个中介者可能是个对象等等，进程可以在其中放置消息，并且可以从中删除消息，以此达到进程间通信的目的。
> 6. **消息队列**：消息队列是内核中存储消息的链表，它由消息队列标识符进行标识，这种方式能够在不同的进程之间提供全双工的通信连接。

7. **共享内存**：共享内存也叫内存映射区mmap, 是使用所有进程之间的内存来建立连接，这种类型需要同步进程访问来相互保护。

16.调度算法都有哪些?

先来先服务，最短作业优先，最短剩余时间优先

17.什么是内核?

在计算机中，内核是一个计算机程序，它是操作系统的核心，可以控制操作系统中所有的内容。内核通常是在 boot loader 装载程序之前加载的第一个程序。

这里还需要了解一下什么是 boot loader :

boot loader 又被称为引导加载程序，能够将计算机的操作系统放入内存中。在电源通电或者计算机重启时，BIOS 会执行一些初始测试，然后将控制权转移到引导加载程序所在的主引导记录(MBR) 。

18.简述TCP和UDP的特点和区别.

用户数据报协议 UDP（User Datagram Protocol）

UDP是无连接的，尽最大可能交付，没有拥塞控制，面向报文（对于应用程序传下来的报文不合并也不拆分，只是添加 UDP 首部），支持一对一、一对多、多对一和多对多的交互通信。

传输控制协议 TCP（Transmission Control Protocol）

TCP 是面向连接的，提供可靠交付，有流量控制，拥塞控制，提供全双工通信，面向字节流（把应用层传下来的报文看成字节流，把字节流组织成大小不等的数据块），每一条 TCP 连接只能是点对点的（一对一）。

19.TCP长连接和短链接的区别是什么?

短连接：Client 向 Server 发送消息，Server 回应 Client，然后一次读写就完成了，这时候双方任何一个都可以发起 close 操作，不过一般都是 Client 先发起 close 操作。短连接一般只会在 Client/Server 间传递一次读写操作。

> **长连接**：Client 与 Server 完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

20.Mysql中where和having的区别是什么?

> - having是在分组后对数据进行过滤
> - where是在分组前对数据进行过滤
> - having后面可以使用聚合函数
> - where后面不可以使用聚合
>
> 聚合语句(sum,min,max,avg,count)要比having子句更早执行，所以having后面可以使用聚合函数。而where子句在查询过程中执行优先级别高于聚合语句(sum,min,max,avg,count)，所以where条件中不能使用聚合函数。

21.请描述一下C++的内存分区模型.

> 1. **栈区(stack)**: 由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。
> 2. **堆区(heap)**: 一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。
> 3. **全局区(静态区)(static)**: 全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。-程序结束后由系统释放。
> 4. **常量区**: 常量字符串就是放在这里的。程序结束后由系统释放
> 5. **代码段**: 存放函数体的二进制代码，直接的操作数也是存储在这个位置的。如int a=4;

22.C语言如何实现C++的面向对象特性?

> **封装** :C语言中是没有class类这个概念的，但是有struct结构体，我们可以考虑使用struct来模拟；使用函数指针把属性与方法封装到结构体中。
>
> **继承** :结构体嵌套
>
> **多态** :类与子类方法的函数指针不同

23.简述数据库的三大范式.

> **第一范式**：1NF是对属性的原子性约束，要求字段具有原子性，不可再分解；
>
> **第二范式**：2NF是在满足第一范式的前提下，非主键字段不能出现部分依赖主键；
>
> **第三范式**：3NF是在满足第二范式的前提下，非主键字段不能出现传递依赖；

24.char和varchar有什么区别?

> **char** 是一种**固定**长度的字符串类型，（如果数据类型不足固定长度的话，会自动用0补齐）
>
> **varchar 是一种可变**长度的字符串类型，（如果数据类型不足长度的话，自动将长度缩放成我们所输入的数据类型的长度）

25.事务是什么？事务的四大特性是什么?

> **事务是一组原子性的 SQL 语句，或者说一个独立的工作单元。**
>
> **原子性**：不可分割的操作单元，事务中所有操作，要么全部成功；要么撤回到执行事务之前的状态；
>
> **一致性**：如果在执行事务之前数据库是一致的，那么在执行事务之后数据库也还是一致的；
>
> **隔离性**：事务操作之间彼此独立和透明互不影响。事务独立运行。这通常使用锁来实现。一个事务处理后的结果，影响了其他事务，那么其他事务会撤回。事务的100%隔离，需要牺牲速度。
>
> **持久性**：事务一旦提交，其结果就是永久的。即便发生系统故障，也能恢复。

26.索引是什么？它是如何加快查询性能的?

> 索引是对数据库表中一列或多列的值进行排序的一种**数据结构**，也就是说索引是一种数据结构。数据库在执行一条Sql语句的时候，默认的方式是根据搜索条件进行全表扫描，遇到匹配条件的就加入搜索结果集合。如果我们对某一字段增加索引，查询时就会先去索引列表中通过二分法等高效率算法一次定位到特定值的行数，大大减少遍历匹配的行数，所以能明显增加查询性能。类似新华字典的目录，如果没有目录的话，我们想要查找一个汉字的话，就必须检索整本字典，而正因为有了目录，我们只要知道我们所要查找的偏旁或者拼音首字母，就可以快速地定位到我们想要查找汉字的所在页码。

27.C++会为一个空类提供哪些函数?

> 当空类Empty_one定义一个对象时Empty_one pt;sizeof(pt)仍是为1，但编译器会生成6个成员函数：一个缺省的构造函数、一个拷贝构造函数、一个析构函数、一个赋值运算符、两个取址运算符(const限定和非const限定)。
>
> 但实际上,我们只需要了解前面四种即可, 即缺省的构造函数, 拷贝构造, 析构函数, 赋值运算符重载。

28.什么是内存对齐?为什么要内存对齐?

> 举个例子:
>
> 理论上，32位系统下，int占4byte，char占一个byte，那么将它们放到一个结构体中应该占4+1=5byte；但是实际上，通过运行程序得到的结果是8 byte，这就是内存对齐所导致的。
>
> 现代计算机中内存空间都是按照 byte 划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但是**实际的计算机系统对基本类型数据在内存中存放的位置有限制，它们会要求这些数据的首地址的值是某个数k（通常它为4或8）的倍数，这就是所谓的内存对齐。**

> 尽管内存是以字节为单位，但是大部分处理器并不是按字节块来存取内存的.它一般会以双字节,四字节,8字节,16字节甚至32字节为单位来存取内存，我们将上述这些存取单位称为**内存存取粒度**.
>
> 现在考虑4字节存取粒度的处理器取int类型变量（32位系统），该处理器只能从地址为4的倍数的内存开始读取数据。
>
> 假如没有内存对齐机制，数据可以任意存放，现在一个int变量存放在从地址1开始的联系四个字节地址中，该处理器去取数据时，要先从0地址开始读取第一个4字节块,剔除不想要的字节（0地址）,然后从地址4开始读取下一个4字节块,同样剔除不要的数据（5，6，7地址）,最后留下的两块数据合并放入寄存器.这需要做很多工作。

扩展内容 : 内存对齐规则

> 每个特定平台上的编译器都有自己的默认“对齐系数”（也叫对齐模数）。gcc中默认**#pragma pack(4)，**可以通过预编译命令**#pragma pack(n)**，n = 1,2,4,8,16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。
>
> 了解了上面的概念后，我们现在可以来看看内存对齐需要遵循的**规则**：

> (1) 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。
>
> (2) 结构体的总大小为 有效对齐值 的整数倍，如有需要编译器会在最末一个成员之后加上填充字节。

举例:

```
//32位系统
#include<stdio.h>
struct
{
    int i;    //4
    char c1;  //1
    char c2;  //1
}x1;
struct{
    char c1;  //1
    int i;    //4
    char c2;  //1
}x2;
struct{
    char c1;  //1
    char c2;  //1
    int i;    //4
}x3;
int main(){
    printf("%d\n",sizeof(x1));  // 输出8
    printf("%d\n",sizeof(x2));  // 输出12
    printf("%d\n",sizeof(x3));  // 输出8
    return 0;
}
```

#此部分转载自知乎(忆臻); 强烈建议看看这位大佬的文章, 受益匪浅!

文章地址:https://zhuanlan.zhihu.com/p/30007037