page 1 user: sakura

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
> 6. 将一些计算从运行期提前到编译期, 例如使用constexpr关键字修饰成常量表达式
