自我介绍
简单代码题：反转句子中的单词
static 关键字的作用
extern 关键字的作用
指针和引用的区别
c++内存分配的方式
静态内存分配和动态内存分配的区别，静态分配的优缺点
互斥锁和自旋锁的区别
线程和进程的区别
如何进行线程切换的?
线程切换需要保存的上下文、保存在哪里?
IP寄存器的作用，是通用寄存器吗?
LR寄存器了解吗（ARM里的，没听说过）
了解ARM架构吗?
线程有哪几种状态?
自旋锁等待时线程处于什么状态？互斥锁呢?
刚拿到互斥锁的线程处于什么状态？（就绪）
什么时候变为运行态（被调度后）
讲一下你了解的进程调度算法

C++中用 C代码，在C++代码文件中 extern "C"

如果把上述的Max.h代码修改如下：

```
/*C语言头文件*/
#ifndef __MAX_H__
#define __MAX_H__
#ifdef __cplusplus
extern "C"{
#endif
    int Max(int nA,int nB);
#ifdef __cplusplus
};
#endif
#endif
```

编译通过

结论：在C++中调用C的代码必须把原来的C语言声明放到extern "C"{/*code*/}中，否则在C++中无法编译通过

原因：C和C++具有完全不同的编译和链接方式。C语言编译器编译函数时不带函数的类型和作用域信息，只包含函数符号名字;而c++编译器为了实现函数的重载，在编译时会带上函数的类型和作用域信息。

自我介绍
讲一下c++智能指针
shared_ptr的底层实现了解吗?
讲一下lambda表达式，lambda表达式优点和应用场景
map 和 unordered_map 区别
unordered_map 实现了解吗?
哈希冲突是指什么?
遇到过的多线程编程的场景（说了自己项目中的多线程应用）
讲一下TCP三次握手
http协议和TCP协议的关系
https协议和http协议的关系
询问了项目总共代码量

tcp：传输控制协议

TCP协议报文格式：

什么是http

HTTP协议是一种基于请求-响应模式的应用层协议，用于在Web浏览器和Web服务器之间传递数据。它是一种无状态的协议，每个请求和响应都是独立的，没有任何关联性。

HTTP通常使用TCP作为传输层协议，使用端口号80进行通信。HTTP协议定义了客户端和服务器之间交换的消息格式和规则，包括请求方法、请求头部、请求正文、响应状态码、响应头部和响应正文等。

HTTP请求由三部分组成：请求行、请求头部和请求正文。其中，请求行包括请求方法、URL和HTTP版本号；请求头部包括请求的附加信息，如Cookie、User-Agent等；请求正文包括请求的数据内容，如表单数据、JSON数据等。

HTTP响应由三部分组成：状态行、响应头部和响应正文。其中，状态行包括HTTP版本号、状态码和状态描述；响应头部包括响应的附加信息，如Content-Type、Content-Length等；响应正文包括响应的数据内容，如HTML页面、图片等。

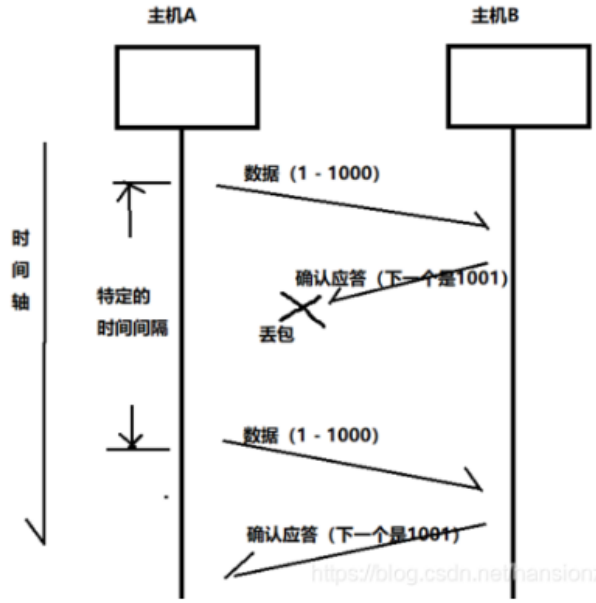
HTTP协议具有以下特点：

1. 简单易用：HTTP协议的消息格式简单明了，易于理解和使用。
2. 无状态：HTTP协议是一种无状态协议，每个请求和响应都是独立的，没有任何关联性。
3. 可扩展性：HTTP协议支持多种请求方法和响应状态码，并且可以使用扩展头部来传递附加信息。
4. 易于缓存：HTTP协议支持缓存机制，可以减少网络传输的数据量，提高系统的性能。
5. 安全性较低：HTTP协议通常不提供加密和认证等安全机制，容易受到中间人攻击和窃听。

- 16位的 紧急 指针：按序到达是 TCP协议 保证可靠性的一种机制，但是也存在一些报文想优先被处理，这时就可以设置 紧急指针 ，指向该报文即可，同时将紧急指针有效位置位 1 。
- 16位窗口大小：如果发送方发送 大量数据 ，接收方接收不过来，会导致大量数据丢失。然后接收方可以发送给发送发消息让发送方发慢一点，这是 流量控制 。接收方将自己 接收缓冲器 剩余空间的大小告诉发送方叫做 16位窗口大小 。发送发可以根据窗口大小来 适配 发送的速度和大小，窗口大小最大是2的16次方，及64KB，但也可以根据选项中的某些位置扩展，最大扩展1G。
- 16位校验和：发送端填充， CRC 校验。如果接收端校验不通过， 则认为数据有问题(此处的检验和不光包含 TCP首部 也包含 TCP数据部分)。

- 16位的 紧急 指针：按序到达是 TCP协议 保证可靠性的一种机制，但是也存在一些报文想优先被处理，这时就可以设置 紧急指针 ，指向该报文即可，同时将紧急指针有效位置位 1 。
- 16位窗口大小：如果发送方发送 大量数据 ，接收方接收不过来，会导致大量数据丢失。然后接收方可以发送给发送发消息让发送方发慢一点，这是 流量控制 。接收方将自己 接收缓冲器 剩余空间的大小告诉发送方叫做 16位窗口大小 。发送发可以根据窗口大小来 适配 发送的速度和大小，窗口大小最大是2的16次方，及64KB，但也可以根据选项中的某些位置扩展，最大扩展1G。
- 16位校验和：发送端填充， CRC 校验。如果接收端校验不通过， 则认为数据有问题(此处的检验和不光包含 TCP首部 也包含 TCP数据部分)。

当然还存在另一种可能就是 主机A 未收到 B 发来的确认应答，也可能是因为 ACK 丢失了。

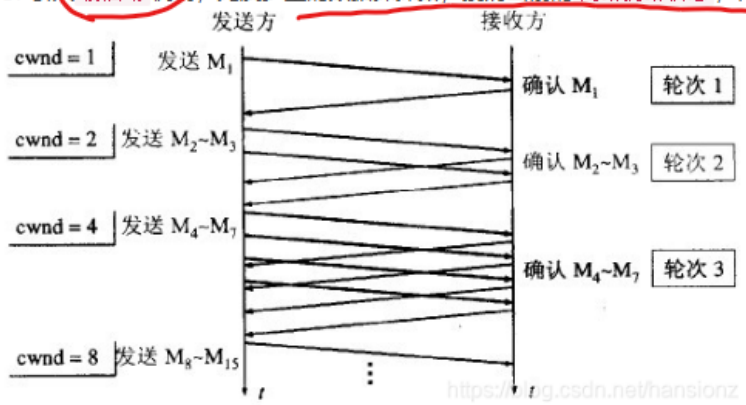


因此主机B会收到很多 重复 数据，那么TCP协议需要能够识别出那些包是 重复的包 ，并且把重复的包丢弃掉，这时候我们可以利用前面提到的 16位序列号 ，就可以很容易做到 去重 的效果。

拥塞控制:

虽然 TCP 有了滑动窗口这个大杀器能够 高效可靠 的发送大量的数据，但是如果在刚开始阶段就发送大量的数据，仍然可能引发问题，因为网络上有很多的计算机，可能当前的 网络状态 就已经比较拥堵，在不清楚当前网络状态下，贸然发送 大量 的数据是很有可能引起雪上加霜的，造成网络更加 堵塞 。

TCP引入 慢启动 机制，先发少量的数据探探路，摸清当前的 网络拥堵状态 ，再决定按照多大的速度传输数据。



图中的 cwnd 为拥塞窗口，在发送开始的时候定义 拥塞窗口 大小为1，每次收到一个ACK应答拥塞窗口 加1 。每次发送数据包的时候，将拥塞窗口和 接收端主机 反馈的窗口大小做比较，取 较小的值 作为实际发送的窗口。

像上面这样的拥塞窗口增长速度，是指数级别的。 "慢启动" 只是指初使时慢，但是增长速度非常快。为了不增长的那么快，因此不能使 拥塞窗口 单纯的加倍，此处引入一个叫做 慢启动的阈值 当拥塞窗口超过这个阈值的时候，不再按照指数方式增长， 而是按照 线性方式 增长。

拥塞控制与流量控制的区别： 拥塞控制是防止过多的数据注入到网络中，可以使网络中的路由器或链路不致过载，是一个全局性的过程。流量控制是点对点通信量 的控制，是一个端到端的问题，主要就是权衡发送端发送数据的速率，以便接收端来得及接收。 拥塞控制的标志： ·重传计时器超时 ·接收到三个重复确认 拥塞避免：（按照线性规律增长） ·拥塞避免并非完全能够避免拥塞，在拥塞避免阶将拥塞窗口控制为按线性规律增长，使网络比较不容易出现拥塞。 ·拥塞避免的思路是让拥塞窗口cwd缓慢地增大，即每经过一个往返时间RTT就把发送方的拥塞控制窗口加一。

捎带回答（ACK等和发送数据一起发送）；延迟回答（让数据先被处理一会儿）；快恢复

快恢复（与快重传配合使用）

- 采用 快恢复 算法时，慢开始只在 TCP 连接建立时和 网络出现超时 时才使用。
- 当发送方连续收到 三个重复 确认时，就执行“乘法减小”算法，把 ssthresh 门限减半。但是接下去并不执行慢开始算法。
- 考虑到如果网络出现 拥塞 的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有 出现拥塞 。所以此时不执行慢开始算法，而是将 cwnd 设置为 ssthresh 的大小，然后执行 拥塞避免 算法。

延迟应答

如果 接收数据 的主机立刻返回 ACK 应答，这时候返回的 窗口 可能比较小。假设接收端缓冲区为 1M 一次收到了 500K 的数据。如果立刻应答，返回的窗口就是 500K 。 但实际上可能处理端处理的速度很快，10ms之内就把500K数据从缓冲区消费掉了，在这种情况下，接收端处理还远没有达到自己的极限，即使窗口再放大一些也能处理过来。如果接收端 稍微 等一会再应答，比如等待 200ms 再应答，那么这个时候返回的窗口大小就是 1M 。

窗口越大，网络 吞吐量 就越大， 传输效率 就越高。我们的目标是在保证网络 不拥塞 的情况下尽量提高传输效率。

- 数量限制: 每隔 N个包 就应答一次
- 时间限制: 超过大 延迟时间 就应答一次

注：具体的数量和超时时间, 依操作系统不同也有差异; 一般 N取2 , 超时时间取 200ms

捎带应答：

在 延迟应答 的基础上，存在很多情况下，客户端服务器在应用层也是 "一发一收" 的。 意味着客户端给服务器说了 "How are you"， 服务器也会给客户端回一个 "Fine, thank you"。那么这个时候 ACK 就可以搭顺风车，和服务器回应的 "Fine, thank you" 一起回给客户端

面向字节流：

当我们创建一个 TCP 的 socket ，同时在内核中创建一个 发送缓冲区 和一个 接收缓冲区 。

- 调用 write 时，内核将数据会先写入 发送缓冲区 中，如果发送的 字节数 太长，会被拆分成多个 TCP 的数据包发出，如果发送的字节数太短，就会先在 缓冲区 里等待， 等到 缓冲区 长度达到设置长度，然后等到其他 合适的时机 发送出去。
- 调用 read 接收数据的时候， 数据也是从网卡驱动程序到达内核的 接收缓冲区 。然后应用程序可以调用 read 从接收缓冲区拿数据。TCP的一个连接，既有发送缓冲区, 也有接收缓冲区，那么对于这一个连接，既可以读数据，也可以写数据。所以是全双工的。

由于 缓冲区 的存在，TCP程序的 读和写 不需要一一匹配。**例如: 写100个字节数据时, 可以调用一次write写100个字节, 也可以调用100次write, 每次写一个字节; 读100个字节数据时, 也完全不需要考虑写的时候是怎么写的, 既可以一次read 100个字节, 也可以一次 read一个字节, 重复100次**

连接异常： 1、关机 进程关闭 正常断开连接；FIN正常发送 2、网络断开；发送方 发现对端不在，则 reset，断开连接 （保活机制，qq重新连接）

close_wait: 被断开方，在收到 FIN 并发出了 ACK应答之后，处于的状态；当我方发出 FIN后，该状态结束 如果有大量 close_wait 状态：

解决方法
基本的思想就是要检测出对方已经关闭的socket，然后关闭它。

1.代码需要判断socket，一旦read返回0，断开连接，read返回负，检查一下errno，如果不是AGAIN，也断开连接。(注:在UNP 7.5节的图7.6中，可以看到使用select能够检测出对方发送了FIN，再根据这条规则就可以处理CLOSE_WAIT的连接)
2.给每一个socket设置一个时间戳last_update，每接收或者是发送成功数据，就用当前时间更新这个时间戳。定期检查所有的时间戳，如果时间戳与当前时间差值超过一定的阈值，就关闭这个socket。
3.使用一个Heart-Beat线程，定期向socket发送指定格式的心跳数据包，如果接收到对方的RST报文，说明对方已经关闭了socket，那么我们也关闭这个socket。
4.设置SO_KEEPALIVE选项，并修改内核参数

我们的程序处于CLOSE_WAIT状态，而不是LAST_ACK状态，说明还没有发FIN给Server，那么可能是在关闭连接之前还有许多数据要发送或者其他事要做，导致没有发这个FIN packet。

原因知道了，那么为什么不发FIN包呢，难道会在关闭己方连接前有那么多事情要做吗？
还有一个问题，为什么有数千个连接都处于这个状态呢？难道那段时间内，服务器端总是主动拆除我们的连接吗？

改内核 sysctl.conf代码 (TCP 处理是 内核去处理的，所以要改 内核配置代码)

下面来看一下我对`/etc/sysctl.conf文件`的修改：

```
#对于一个新建连接，内核要发送多少个 SYN 连接请求才决定放弃,不应该大于255，默认值是5，对应于
net.ipv4.tcp_syn_retries=2
#net.ipv4.tcp_synack_retries=2
#表示当keepalive起用的时候，TCP发送keepalive消息的频度。缺省是2小时，改为300秒
net.ipv4.tcp_keepalive_time=1200
net.ipv4.tcp_orphan_retries=3
#表示如果套接字由本端要求关闭，这个参数决定了它保持在FIN-WAIT-2状态的时间
net.ipv4.tcp_fin_timeout=30
#表示SYN队列的长度，默认为1024，加大队列长度为8192，可以容纳更多等待连接的网络连接数。
net.ipv4.tcp_max_syn_backlog = 4096
#表示开启SYN Cookies。当出现SYN等待队列溢出时，启用cookies来处理，可防范少量SYN攻击，默认
net.ipv4.tcp_syncookies = 1

#表示开启重用。允许将TIME-WAIT sockets重新用于新的TCP连接，默认为0，表示关闭
net.ipv4.tcp_tw_reuse = 1
#表示开启TCP连接中TIME-WAIT sockets的快速回收，默认为0，表示关闭
net.ipv4.tcp_tw_recycle = 1

##减少超时前的探测次数
net.ipv4.tcp_keepalive_probes=5
##优化网络设备接收队列
net.core.netdev_max_backlog=3000
```

修改完之后执行`/sbin/sysctl -p`让参数生效。

长连接和短连接：

TCP Keepalive工作原理

当一个 TCP 连接建立之后，启用 TCP Keepalive 的一端便会启动一个计时器，当这个计时器数值到达 0 之后（也就是经过tcp_keep-alive_time时间后，这个参数之后会讲到），一个 TCP 探测包便会被发出，这个 TCP 探测包是一个纯 ACK 包（规范建议，不应该包含任何数据，但也可以包含1个无意义的字节，比如0x0。），其 Seq号 与上一个包是重复的，所以其实探测保活报文不在窗口控制范围内。

如果一个给定的连接在两小时内（默认时长）没有任何的动作，则服务器就向客户发一个探测报文段，客户主机必须处于以下4个状态之一：

1. 客户主机依然正常运行，并从服务器可达。客户的TCP响应正常，而服务器也知道对方是正常的，服务器在两小时后将保活定时器复位。

2. 客户主机已经崩溃，并且关闭或者正在重新启动。在任何一种情况下，客户的TCP都没有响应。服务端将不能收到对探测的响应，并在75秒后超时。服务器总共发送10个这样的探测 ，每个间隔75秒。如果服务器没有收到一个响应，它就认为客户主机已经关闭并终止连接。

3. 客户主机崩溃并已经重新启动。服务器将收到一个对其保活探测的响应，这个响应是一个复位，使得服务器终止这个连接。

4. 客户机正常运行，但是服务器不可达，这种情况与2类似，TCP能发现的就是没有收到探测的响应。

对于linux 内核来说，应用程序若想使用TCP Keepalive，需要设置SO_KEEPALIVE套接字选项才能生效。

有三个重要的参数：

1. tcp_keepalive_time，在TCP保活打开的情况下，最后一次数据交换到TCP发送第一个保活探测包的间隔，即允许的持续空闲时长，或者说每次正常发送心跳的周期，默认值为7200s（2h）。

2. tcp_keepalive_probes 在tcp_keepalive_time之后，没有接收到对方确认，继续发送保活探测包次数，默认值为9（次）

3. tcp_keepalive_intvl，在tcp_keepalive_time之后，没有接收到对方确认，继续发送保活探测包的发送频率，默认值为75s。

其他编程语言有相应的设置方法，这里只谈linux内核参数的配置。例如C语言中的setsockopt()函数，java的Netty 服务器框架中也提供了相关接口。

TCP Keepalive作用

1. 探测连接的对端是否存活

在应用交互的过程中，可能存在以下几种情况：

（1）客户端或服务器意外断电，死机，崩溃，重启。

（2）中间网络已经中断，而客户端与服务器并不知道。

利用保活探测功能，可以探知这种对端的意外情况，从而保证在意外发生时，可以释放半打开的TCP连接。

2. 防止中间设备因超时删除连接相关的连接表

中间设备如防火墙等，会为经过它的数据报文建立相关的连接信息表，并为其设置一个超时时间的定时器，如果超出预定时间，某连接无任何报文交互的话，

中间设备会将该连接信息从表中删除，在删除后，再有应用报文过来时，中间设备将丢弃该报文，从而导致应用出现异常。

此段参考：https://www.cnblogs.com/hukey/p/5481173.html

TCP Keepalive可能导致的问题

Keepalive 技术只是 TCP 技术中的一个可选项。因为不当的配置可能会引起一些问题，所以默认是关闭的。

可能导致下列问题：

1. 在短暂的故障期间，Keepalive设置不合理时可能会因为短暂的网络波动而断开健康的TCP连接
2. 需要消耗额外的宽带和流量
3. 在以流量计费的互联网环境中增加了费用开销

TCP Keepalive HTTP Keep-Alive 的关系

很多人会把TCP Keepalive 和 HTTP Keep-Alive 这两个概念搞混淆。

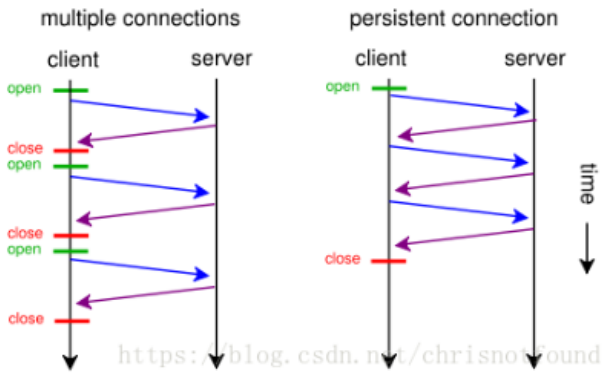
这里简单介绍下HTTP Keep-Alive 。

在HTTP/1.0中，默认使用的是短连接。也就是说，浏览器和服务器每进行一次HTTP操作，就建立一次连接，但任务结束就中断连接。如果客户端浏览器访问的某个HTML或其他类型的 Web页中包含有其他的Web资源，如JavaScript文件、图像文件、CSS文件等；当浏览器每遇到这样一个Web资源，就会建立一个HTTP会话。

但从 HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头加上Connection、Keep-Alive字段.如下图所示

```
Cache-Control: max-age=120
Connection: keep-alive
Keep-Alive: timeout=20
Content-Encoding: gzip
Content-Type: text/html; charset=GB2312
Date: Fri, 27 Apr 2018 09:43:31 GMT
Expires: Fri, 27 Apr 2018 09:45:31 GMT
Server: squid/3.5.24
Transfer-Encoding: chunked
```

HTTP 1.0 和 1.1 在 TCP连接使用方面的差异如下图所示



UDP： 没有发送缓冲区。因为不支持 重发送

UDP协议

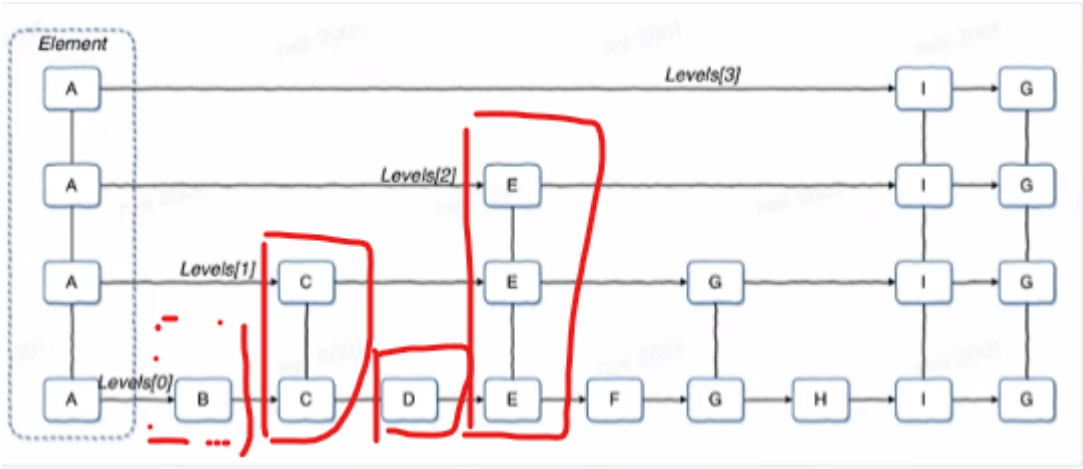
UDP协议报文格式：

16	31
源端口	目的端口
数据包长度	校验值
数据 DATA	

- 16位UDP长度表示整个数据报（UDP首部+UDP数据）的长度
- 如果校验和出错，就会直接丢弃(UDP校验首部和数据部分)

16位的最大长度表示，所以报文最大是 64K

Skip_List



所有层的Node，构成一个完整的 Node

```
//有序单链表的查找

//当前指针指向的节点的值 < 要查找的值，继续查找
//当前指针指向的节点的值 = 要查找的值，直接返回
//当前指针指向的节点的值 > 要查找的值，没有找到
```

有序单链表的查找情况：

每一层往右走，是一个 Node集合在往右走

```
prevElem := list.header
i := len(list.header.levels) - 1

for i >= 0 {
    //在每一层执行有序单链表的查找，查找终止的条件是，nextElement > findKey
    //这个位置说明该值不存在，或者是该值应该被插入的位置
    for next := prevElem.levels[i]; next != nil; next = prevElem.levels[i] {
        if findKey <= next.key {
            if findKey == next.key {
                return next.data
            }
            break
        }
        prevElem = next
    }
    i--
}

return NotFound
```

插入的 记录每一层，前面的 Node节点

```
prevElem = next
prevElemList[i] = prevElem
```

preElements是一个数组

在这段代码中，`newLevel` 是新插入节点的层数。代码的目的是为了确保跳表的每一层都有适当的前驱节点（`update` 数组中的指针）。具体来说，这部分的逻辑主要有以下几个目的：

1. **动态调整跳表层数**：
   - 跳表的层数是动态的，可能会因为新节点的插入而增加。如果新节点的层数（`newLevel`）大于当前的最大层数（`level`），就需要更新跳表的层数。
2. **更新前驱节点**：
   - 如果 `newLevel` 大于 `level`，说明新节点需要在新层上插入。由于当前 `update` 数组中的前驱节点只会在当前层（即 `level` 的高度）有效，新的层需要重新设置前驱节点。
   - 通过循环，将新增层的前驱节点设置为 `head`，这样保证了在新层中，所有小于新节点的节点仍然可以有正确的指向。
3. **更新当前层数**：
   - 最后，更新 `level` 为 `newLevel`，使得跳表的层数反映出当前节点的最大层数。

示例说明

假设当前跳表的层数为 `3`（即 `level` 为 `3`），而你要插入一个新节点，其层数为 `4`（`newLevel` 为 `4`）。这意味着跳表中现在需要有第四层。由于前驱节点在这层上可能没有被正确初始化，所以需要将 `update` 数组中新增的层的前驱节点指向 `head`，确保在跳表的任何高度都能找到对应的前驱。

```cpp
void insert(int value) {
    std::vector<Node*> update(MAX_LEVEL, nullptr);
    Node* current = head;

    // 找到每一层中需要更新的前置节点
    for (int i = level - 1; i >= 0; i--) {
        while (current->next[i] != nullptr && current->next[i]->value < value) {
            current = current->next[i];
        }
        update[i] = current;
    }

    int newLevel = randomLevel();
    if (newLevel > level) {
        for (int i = level; i < newLevel; i++) {
            update[i] = head;
        }
        level = newLevel;
    }

    Node* newNode = new Node(value, newLevel);
    for (int i = 0; i < newLevel; i++) {
        newNode->next[i] = update[i]->next[i];
        update[i]->next[i] = newNode;
    }
}
```

往上新增的 层数：的update[i]前驱节点一定是 Head

红黑树和散列表：

2.2、应用场景和优缺点

红黑树：

- 应用场景：适用于需要频繁插入、删除、查找操作且数据量较大，要求有序性的情况。
- 优点：① 具有平衡性，插入、删除、查找的时间复杂度为$O(logn)$；② 支持有序遍历；③ 可用于实现有序集合、有序映射等场景。
- 缺点：①相对于散列表来说，需要更多的内存空间；②对于基于散列的操作，如随机访问，并不擅长。

散列表：

- 应用场景：适用于需要频繁插入、删除、查找操作且数据量较大，不需要有序性的情况。
- 优点：①查找、插入、删除的平均时间复杂度为O(1)，最坏时间复杂度是O(n)；②支持随机访问，性能通常比树要好。
- 缺点：①可能会出现哈希冲突，需要使用冲突解决方法，如链表法、开放寻址法等；②散列表的元素是无序的，不支持有序遍历。

1. 红黑树（Red-Black Tree）是一种自平衡的二叉搜索树，具有以下特点：
    - 所有节点都是红色或黑色。
    - 根节点为黑色。
    - 每个叶子节点（Nil节点）都是黑色。
    - 如果一个节点是红色，则其两个子节点都是黑色。
    - 对于任意节点，从该节点到其所有后代叶子节点的简单路径上，均包含相同数目的黑色节点。

    红黑树的插入、删除和查找操作的平均时间复杂度均为 O(logN)，并且可以保持良好的有序性。
2. 散列表（Hash Table）是根据键直接进行访问的数据结构，通过散列函数将键映射到存储位置。它具有以下特点：
    - 插入和查找操作的平均时间复杂度是常数级别（O(1)）。
    - 不保持元素的有序性。
    - 可能存在散列冲突，需要解决冲突的方法（如链地址法、开放寻址法等）。

    散列表适用于需要频繁的插入和查找操作，并且对于元素的顺序不敏感的场景。
3. 在选择 `map` 和 `unordered_map` 时，可以考虑以下因素：
    - 是否需要有序性：如果需要保持元素的有序性，并且根据键进行快速查找，可以选择 `map`。
    - 插入和删除操作频率：如果要频繁地插入和删除元素，并且对于查找操作的顺序不敏感，可以选择 `unordered_map`。
    - 内存占用考虑：`unordered_map` 往往需要更多的内存空间。
4. 关于C++ STL容器的建议使用方式：
    - 在选择容器时，根据具体需求权衡使用场景并选择最适合的容器。例如，需要有序性时可选择 `map`，需要快速插入/删除操作时可选择 `unordered_map`。
    - 注意不同容器之间的特点和性能差异，了解它们的底层实现机制和时间复杂度，以便作出合理的选择。
    - 使用迭代器进行遍历和访问容器元素，可以灵活地操作数据。
    - 注意容器使用过程中的内存管理，避免内存泄漏和多余的复制操作。
    - 熟悉容器提供的成员函数和算法，利用它们来简化代码，提高开发效率。

单例模式

```cpp
class Demo {
public:
    static Demo* GetObject() {
        if (Real_object == NULL) {
            Real_object = new Demo();
        }
        return Real_object;
    }

    ~Demo() {
        if (Real_object == NULL) {
            delete Real_object;
            Real_object = NULL;
        }

        cout << "The Object has been destroyed" << endl;
    }

private:
    Demo() { cout << "Default Constrution" << endl; };
    Demo(const Demo& dd) = delete;
    const Demo& operator=(const Demo& dd) = delete;
    static Demo* Real_object;
};
```

构建模式

```cpp
#include <iostream>
#include <string>

// 产品类：计算机
class Computer {
private:
    std::string CPU;
    std::string GPU;
    std::string RAM;
    std::string Storage;

public:
    void setCPU(const std::string& cpu) { CPU = cpu; }
    void setGPU(const std::string& gpu) { GPU = gpu; }
    void setRAM(const std::string& ram) { RAM = ram; }
    void setStorage(const std::string& storage) { Storage = storage; }

    void showSpecifications() const {
        std::cout << "Computer Specifications:" << std::endl;
        std::cout << "CPU: " << CPU << std::endl;
        std::cout << "GPU: " << GPU << std::endl;
        std::cout << "RAM: " << RAM << std::endl;
        std::cout << "Storage: " << Storage << std::endl;
    }
};
```

```cpp
// 抽象 Builder 类
class ComputerBuilder {
public:
    virtual ~ComputerBuilder() = default;

    virtual void buildCPU() = 0;
    virtual void buildGPU() = 0;
    virtual void buildRAM() = 0;
    virtual void buildStorage() = 0;

    virtual Computer* getComputer() = 0;
};
```

```cpp
// 具体 Builder 类：高性能电脑
class HighEndComputerBuilder : public ComputerBuilder {
private:
    Computer* computer;

public:
    HighEndComputerBuilder() { computer = new Computer(); }

    void buildCPU() override { computer->setCPU("Intel Core i9"); }
    void buildGPU() override { computer->setGPU("NVIDIA RTX 4090"); }
    void buildRAM() override { computer->setRAM("64GB DDR4"); }
    void buildStorage() override { computer->setStorage("2TB NVMe SSD"); }

    Computer* getComputer() override { return computer; }
};

// 具体 Builder 类：低端电脑
class LowEndComputerBuilder : public ComputerBuilder {
private:
    Computer* computer;

public:
    LowEndComputerBuilder() { computer = new Computer(); }

    void buildCPU() override { computer->setCPU("Intel Core i3"); }
    void buildGPU() override { computer->setGPU("Integrated Graphics"); }
    void buildRAM() override { computer->setRAM("8GB DDR4"); }
    void buildStorage() override { computer->setStorage("256GB SSD"); }

    Computer* getComputer() override { return computer; }
};
```

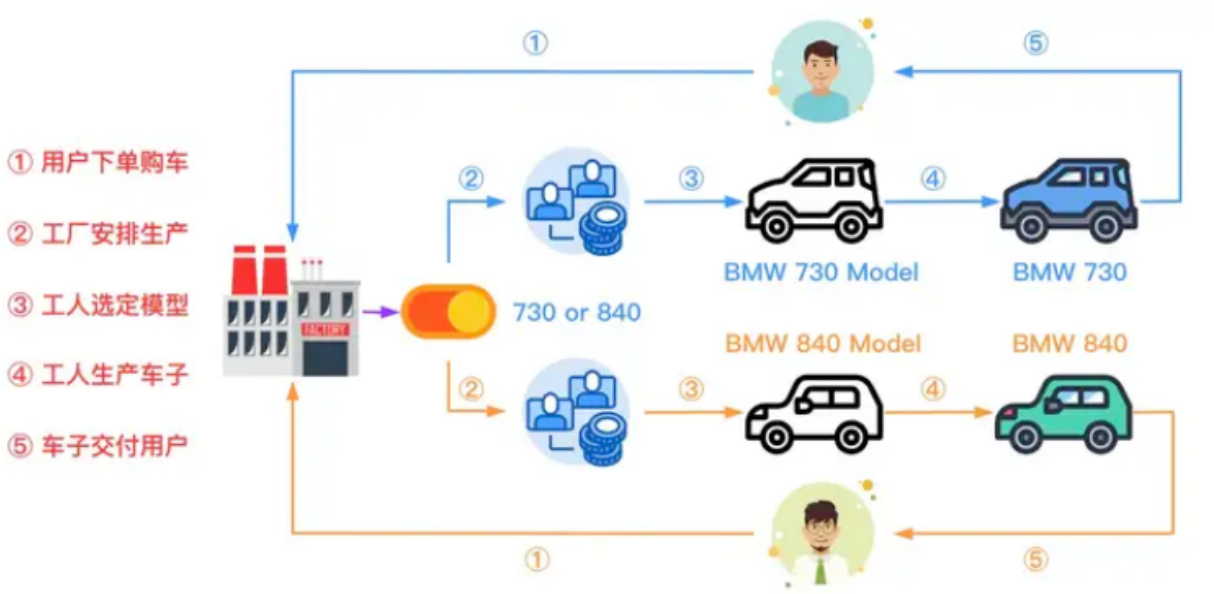
```cpp
// 指导者类
class Director {
private:
    ComputerBuilder* builder;

public:
    void setBuilder(ComputerBuilder* b) { builder = b; }

    void constructComputer() {
        builder->buildCPU();
        builder->buildGPU();
        builder->buildRAM();
        builder->buildStorage();
    }
};
```

工厂模式

简单工厂模式又叫 **静态方法模式**，因为工厂类中定义了一个静态方法用于创建对象。简单工厂让使用者不用知道具体的参数就可以创建出所需的 " 产品 " 类，即使用者可以直接消费产品而不需要知道产品的具体生产细节。



在上图中，阿宝哥模拟了用户购车的流程，小王和小秦分别向 BMW 工厂订购了 BMW730 和 BMW840 型号的车型，接着工厂会先判断用户选择的车型，然后按照对应的模型进行生产并在生产完成后交付给用户。

下面我们来看一下如何使用简单工厂来描述 BMW 工厂生产指定型号车子的过程。

```
#include <iostream>
#include <memory>  // 用于智能指针 std::unique_ptr

// 产品基类
class Product {
public:
    virtual ~Product() = default;
    virtual void show() const = 0;  // 纯虚函数，展示产品信息
};

// 具体产品类A
class ConcreteProductA : public Product {
public:
    void show() const override {
        std::cout << "ConcreteProductA" << std::endl;
    }
};
```

```
// 具体产品类B
class ConcreteProductB : public Product {
public:
    void show() const override {
        std::cout << "ConcreteProductB" << std::endl;
    }
};
```

```
// 工厂类
class SimpleFactory {
public:
    // 工厂方法，根据传入的类型创建不同的产品
    static std::unique_ptr<Product> createProduct(const std::string& type) {
        if (type == "A") {
            return std::make_unique<ConcreteProductA>();
        } else if (type == "B") {
            return std::make_unique<ConcreteProductB>();
        } else {
            return nullptr;  // 如果类型不匹配，返回空指针
        }
    }
};
```