C++面试总结

字节

- mmap的原理
  - 调用 `msync` 函数主动进行数据同步（主动）。
  - 调用 `munmap` 函数对文件进行解除映射关系时（主动）。
  - 进程退出时（被动）
  - 系统关机时（被动）

- C++对象模型。内存对齐的Padding，成员函数的布局
  - 通过一个类对象调用成员函数，如果确定这个成员函数属于这个类呢?

    对象调用成员函数时，在编译期，编译器就可以确定这些函数的地址，并通过传入this指针和其他参数，完成函数的调用，所以类中就没有必要存储成员函数的信息。
  - （1）成员变量在类中内存的先后顺序与其在类中被声明的先后顺序一致，也就是说先声明的变量在内存的前边。

    （2）类中成员变量初始化的先后顺序与其声明先后顺序一致，与constructor中被初始化先后顺序无关。

    （3）如果类中含有虚函数，那么类中第一个位置的变量是vptr。

    （4）类大小的是在编译期间就被确定了，所以sizeof可以作用与一个类名。
- 输入一个https 的url，发生了什么
  - DNS域名解析，本地DNS之间的查询方式是递归查询；在本地DNS服务器与根域及其子域之间的查询方式是迭代查询，根DNS → 顶级域DNS → 权威域DNS,**从浏览器缓存中查找→本地的hosts文件查找→找本地DNS解析器缓存查找→本地DNS服务器查找**
  - 判断是Http/Https，HTTP报文是包裹在TCP报文中发送的，服务器端收到TCP报文时会解包提取出HTTP报文。三次握手
  - 发送Http请求，浏览器发 HTTP 请求要携带三样东西：请求行（request line）、请求头（header）和请求体（body）
    - 请求行：get/post 版本

    - 请求头
    - 请求体：**请求体只有在POST方法下存在**

      服务器接受到请求，就解析请求头，如果头部有缓存相关信息如if-none-match与if-modified-since，则验证缓存是否有效，若有效则返回状态码为304，若无效则重新返回资源，状态码为200.
  - 响应行、响应头和响应体。
  - 渲染页面
  - 四次挥手断开连接
- HTTPS证书、非对称加密、TLS握手过程
  - "client hello" 消息：客户端通过发送 “client hello” 消息向服务器发起握手请求，该消息包含了客户端所支持的 TLS 版本，支持的算法列表和密码组合以供服务器进行选择，还有一个 “client random” 随机字符串。

    "server hello" 消息：服务器发送 “server hello” 消息对客户端进行回应，该消息包含了服务器选择的加密算法，服务器选择的密码组合，数字证书和 “server random” 随机字符串。

    验证：客户端对服务器发来的证书进行验证，确保对方的合法身份，验证过程可以细化为以下几个步骤：

    检查数字签名
    验证证书链
    检查证书的有效期
    检查证书的撤回状态（撤回代表证书已失效）
    "premaster secret" 字符串：客户端向服务器发送另一个随机字符串 “premaster secret（预主密钥）”，这个字符串是经过服务器的公钥加密过的，只有对应的私钥才能解密。

    使用私钥：服务器使用私钥解密 “premaster secret”。

    生成共享密钥：客户端和服务器均使用 client random，server random 和 premaster secret，并通过相同的算法生成相同的共享密钥 KEY。

    客户端就绪：客户端发送经过共享密钥 KEY 加密过的 “finished” 信号，为了防止握手过程遭到篡改，该消息的内容是前一阶段所有握手消息的MAC值。

    服务器就绪：服务器发送经过共享密钥 KEY 加密过的 “finished” 信号，该消息的内容是前一阶段所有握手消息的 MAC 值。

> 达成安全通信：握手完成，双方使用对称加密进行安全通信。

- 拥塞控制、流量控制原理
- inline 和 define 区别（对内联函数的处理发生在编译期，对宏定义的处理发生在预处理；内联函数能够提供类型检查，相比宏定义更加安全；内联函数只是给编译器在编译时提供优化建议，使内容短小简单的处理不以发生函数调用的方式进行，减少函数调用的寻找函数入口地址，保存现场以及返回时发生的开销，当内联函数过于复杂时，编译器会选择按照一般函数的方式编译，不会在调用的位置展开替换，而宏定义一定会展开替换）
- C++ 编译和预处理（预处理：将宏定义展开替换到代码中的指定位置，删除注释内容等；编译：词法和语法分析，代码优化）
- https://www.nowcoder.com/discuss/post/411287816003485696
- 具体讲解dns解析，从效率角度分析
- 虚拟内存管理的特点，优势
  - 1、多次性：无需在作业运行时，一次性全部装入内存，可以被分成多次调入内存
    2、对换性：在作业运行时，无需一直常驻内存，允许作业在运行过程中，将作业换入换出
    3、虚拟性：从逻辑上扩充了内存的容量，是用户看到的内存容量远大于实际容量。

    虚拟内存的实现：
    1、请求分页存储管理
    2、请求分段存储管理
    3、请求段页式存储管理
- 虚拟地址和物理地址如何转化
- 缺页中断的过程
  - 首先硬件会陷入内核，在堆栈中保存程序计数器。大多数机器将当前指令的各种状态信息保存在CPU中特殊的寄存器中。

    启动一个汇编代码例程保存通用寄存器及其它易失性信息，以免被操作系统破坏。这个例程将操作系统作为一个函数来调用。

    （在页面换入换出的过程中可能会发生上下文换行，导致破坏当前程序计数器及通用寄存器中本进程的信息）
    当操作系统发现是一个页面中断时，查找出来发生页面中断的虚拟页面（进程地址空间中的页面）。这个虚拟页面的信息通常会保存在一个硬件寄存器中，如果没有的话，操作系统必须检索程序计数器，取出这条指令，用软件分析该指令，通过分析找出发生页面中断的虚拟页面。

> 检查虚拟地址的有效性及安全保护位。如果发生保护错误，则杀死该进程。
>
> 页面调度（也就是算法过程，看算法整理笔记）
>
> 操作系统查找一个空闲的页框(物理内存中的页面)，如果没有空闲页框则需要通过页面置换算法找到一个需要换出的页框。
>
> 如果找的页框中的内容被修改了，则需要将修改的内容保存到磁盘上，此时会引起一个写磁盘调用，发生上下文切换（在等待磁盘写的过程中让其它进程运行）。
>
> (注：此时需要将页框置为忙状态，以防页框被其它进程抢占掉)
> 页框干净后，操作系统根据虚拟地址对应磁盘上的位置，将保持在磁盘上的页面内容复制到“干净”的页框中，此时会引起一个读磁盘调用，发生上下文切换。
>
> 当磁盘中的页面内容全部装入页框后，向操作系统发送一个中断。操作系统更新内存中的页表项，将虚拟页面映射的页框号更新为写入的页框，并将页框标记为正常状态。
>
> 恢复缺页中断发生前的状态，将程序指令器重新指向引起缺页中断的指令。
>
> 调度引起页面中断的进程，操作系统返回汇编代码例程。
>
> 汇编代码例程恢复现场，将之前保存在通用寄存器中的信息恢复。
>
> 其实缺页中断的过程涉及了用户态和内核态之间的切换，虚拟地址和物理之间的转换（这个转换过程需要使用MMU和TLB），同时涉及了内核态到用户态的转换。

- 页面置换算法
- 共享内存 信号量的优缺点
  - 优点：采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据[1]： 一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建 立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回 文件的。因此，采用共享内存的通信方式效率是非常高的。
    缺点： 共享内存没有提供同步的机制，这使得我们在使用共享内存进行进程间通信时，往往要借助其他的手段（信号量）来进行进程间的同步工作。
- 如何实现客户端超时断开(https://blog.csdn.net/qq_46495964/article/details/122761499)
- 在TCP里面是如何判断客户端断开的

- 如何实现断点续传
- HTTP协议的格式，还问了请求行，请求头，请求体里面有啥
- QUIC的原理
  - QUIC 基于连接 ID 唯一识别连接。当源地址发生改变时，QUIC 仍然可以保证连接存活和数据正常收发
  - 低连接延时
  - 不管是原始包还是重传包，都带有一个新的序列号(seq)，这使得Quic能够区分ACK是重传包还是原始包，从而避免了TCP重传模糊的问题
  - QUIC 的传输控制不再依赖内核的拥塞控制算法，而是实现在应用层上，这意味着我们根据不同的业务场景，实现和配置不同的拥塞控制算法以及参数
  - TCP 队头阻塞的主要原因是数据包超时确认或丢失阻塞了当前窗口向右滑动，我们最容易想到的解决队头阻塞的方案是不让超时确认或丢失的数据包将当前窗口阻塞在原地，TCP 为了保证可靠性，使用了基于字节序号的 Sequence Number 及 Ack 来确认消息的有序到达。
  - QUIC 使用的Packet Number 单调递增的设计，可以让数据包不再像TCP 那样必须有序确认，QUIC 支持乱序确认，当数据包Packet N 丢失后，只要有新的已接收数据包确认，当前窗口就会继续向右滑动。QUIC使用Stream ID 来标识当前数据流属于哪个资源请求，有了Stream Offset 字段信息，属于同一个Stream ID 的数据包也可以乱序传输了
- 物理内存4G，能运行大于4G内存的程序吗？物理内存和虚拟内存之间怎么交换
- **死锁产生条件和避免方法**
- cookie和session
  - **Cookie通过在客户端记录信息确定用户身份，Session通过在服务器端记录信息确定用户身份。**
- constexpr
- 静态库里面是什么
- unicode和utf-8
- DNS、DNS劫持、get和post区别、TCP和UDP区别、三次握手为什么不是2或4、四次挥手
- TCP和UDP区别以及各自使用场景，UDP不可靠的话那为什么咱们现在视频这么流畅，是依靠什么

- https://www.nowcoder.com/discuss/post/353156790680494080
- SIGSEGV信号
- 怎么知道一个进程死亡?
- 怎么hook一个进程的准确死亡时间?
- 十亿个用户id怎么去重? (BitMap)
- 数据是先到交换机还是路由器(先到交换机 查找MAC表再由路由器分发)
- 三次握手第三次丢了怎么办?

  - 第一次丢失:客户端会等不到服务端的ACK就会超时重传
  - 第二次丢失:客户端迟迟等不到第一次握手的确认,就会触发超时重传机制,进行超时重传;服务器等不到自己SYN连接的确认,也会进行超时重传。
  - 第三次丢失:第三次握手丢失,此时客户端已经处于established状态了,因为它通过两次握手已经验证了自己的发送和接收能力嘛。但是此时第三次握手丢失,服务器迟迟得不到ACK报文,但是ACK报文丢失,ACK 报文是不会有重传的,当 ACK 丢失了,就由对方重传对应的报文。所以当到达服务器的超时重传时间后,服务器会超时重传第二次报文,当达到最大超时重传次数还没得到ACK报文,服务器就会断开连接。
- 四次挥手的状态?为什么有TIME_WAIT?

  - 可靠地终止TCP连接。
  - 保证让迟来的TCP报文段有足够的时间被识别并丢弃。
  - 第一次丢失:

    如果第一次挥手丢失了,那么客户端迟迟收不到被动方的 ACK 的话,也就会触发超时重传机制,重传 FIN 报文,重发次数由 `tcp_orphan_retries` 参数控制。

    当客户端重传 FIN 报文的次数超过 `tcp_orphan_retries` 后,就不再发送 FIN 报文,则会在等待一段时间(时间为上一次超时时间的 2 倍),如果还是没能收到第二次挥手,那么直接进入到 `close` 状态。
  - 第二次丢失:ACK 报文是不会重传的,所以如果服务端的第二次挥手丢失了,客户端就会触发超时重传机制,重传 FIN 报文,直到收到服务端的第二次挥手,或者达到最大的重传次数。(shutdown死等待)
  - 第三次丢失:服务端发送FIN等待ACK答复,如果没有收到就会重新发送ACK,等待一段时间后还没有收到就会主动断开,此时客户端也在FIN_WAIT_2状态,等待FIN报文到达,等待一段时间后没有报文就会主动关闭

    - 第四次丢失：当服务端重传第三次挥手报文达到 2 时，由于 tcp_orphan_retries 为 2， 达到了最大重传次数，于是再等待一段时间（时间为上一次超时时间的 2 倍），如果还是没能收到客户端的第四次挥手（ACK 报文），那么服务端就会断开连接。客户端在收到第三次挥手后，就会进入 TIME_WAIT 状态，开启时长为 2MSL 的定时器，如果途中再次收到第三次挥手（FIN 报文）后，就会重置定时器，当等待 2MSL 时长后，客户端就会断开连接。
- 左值引用不能绑定到右值。但是常量左值引用可以（为什么）
    - 当常量左值引用绑定右值时，编译器根据常亮左值引用的作用域，在栈或堆上新建临时变量，其生命周期与常量左值引用保持一致。那么倒推一下，因为常量左值引用不可修改的属性，允许其绑定匿名变量也是可以的。
- 右值引用能接受左值吗？（std::move()）
- 万能引用（auto &&）
- extern C
- **i++是否原子操作 不是**
- **B Tree 和 B+ Tree的区别**
- **锁的底层实现**
    - 在对变量进行计算之前(如 ++ 操作)，首先读取原变量值，称为 **旧的预期值 A**
    - 然后在更新之前再获取当前内存中的值，称为 **当前内存值 V**
    - 如果 A==V 则说明变量从未被其他线程修改过，此时将会写入**新值 B**
    - 如果 A≠V 则说明变量已经被其他线程修改过，当前线程应当什么也不做
- 查看程序对应的进程号： ps -ef | grep 进程名字 查看进程号所占用的端口号: netstat -nltp | grep 进程号 查看端口号所使用的进程号: lsof -i:端口号
- include怎么处理
- MFC句柄
    - 首先讲一下MFC中句柄的概念，句柄是一种资源标识，他是一个整数，由操作系统分配，用户（程序员）没法指定。举个例子来说，当你创建了一个对话框，那么系统就会为这个对话框分配一定的资源，并且为这个资源分配一个整数来标识这块资源，那么这个整数就是句柄。 什么是资源，资源当然就是内存，堆栈。对话框的所有资源封装在一个类中，形成自己的数据结构，这个数据结构占用系统的一块内存。

    - 其次，句柄用HANDLE表示， 定义方式，HANDLE hWnd； 其实句柄的实现方式就是指针， 句柄是一种指针的指针，系统中应该会有一张资源标识符表， 资源标识符表中放的应该就是指向资源的指针，通过句柄可以找到这个存放资源指针的地
- 栈溢出 原因
- 物理空间充足new失败的原因
- 多进程的好处
    - 每个进程互相独立，有独立的虚拟地址空间，子程序不影响主程序的稳定性，子进程崩溃没关系，比如谷歌浏览器；
    - 尽量减少数据共享的安全问题和线程加锁/解锁的影响；
    - 可用地址空间比较大，4GB(32位，232)，264(64位)。
- 如何理解虚拟地址
    - 虚拟地址实际上是操作系统为应用程序提供的一个统一的内存访问接口，这样做的好处是—所有的应用程序只需要面向虚拟地址进行编写，而不用考虑实际的物理地址的使用情况。
    - 程序可以使用一系列相邻的虚拟地址来访问物理内存中不相邻的大内存缓冲区。
    - 不同进程使用的虚拟地址彼此隔离。一个进程中的代码无法更改正在由另一进程使用的物理内存。
    - 一个系统如果同时运行着很多进程，为各进程分配的内存之和可能会大于实际可用的物理内存，虚拟内存管理使得这种情况下各进程仍然能够正常运行
      程序可以使用一系列虚拟地址来访问大于可用物理内存的内存缓冲区。当物理内存的供应量变小时，内存管理器会将物理内存页（通常大小为 4 KB）保存到磁盘文件。数据或代码页会根据需要在物理内存与磁盘之间移动。
    - **虚拟内存管理最 主要的作用是让每个进程有独立的地址空间 （进程间的安全）**
    - 执行错误指令或恶意代码的破坏能力受到了限制，顶多使当前进程因段错误终止，而不会影响整个系统的稳定性。
    - MMU：将虚拟地址分割为三分部，高10位作为页目录中元素的下标，中间10位作为页表中元素的下标，最后12位作为页内偏移
- 通过分段，可以将不同段的代码映射到不同的物理内存地址处，从而避免了虚拟内存中未使用的部分占用实际的物理内存。分段管理空间，将空间分成不同的片之后，会存在外部碎片，随着时间的推移，导致分配内存比较困难，所以操作系统又引入了分页。

  分段 容易按照逻辑模块实现信息的共享和保护

- 虚拟地址的大小、栈空间的大小
- 静态变量类外定义：**而定义是分配内存。**静态成员变量在类中仅仅是声明，没有定义，所以要在类的外面定**义，实际上是给静态成员变量分配内存**
- 为什么模板类不能够声明和定义分离？

  模板类是编译器生成具体的类的依据，只有模板被使用时才会编译。首先一般编译器都是以一个.cpp文件为一个编译单元，如果模板类的声明和实现是分离的，那么对模板类的定义文件编译，生成.o文件，此时只有模板，没有模板的实例类。c++编译器的工作流程分为预处理、编译、汇编、链接，而模板实例化发生在编译期间，当编译器没有找到模板类的一个特例时，它会认为该特例在另外的文件中（.o或.so），而将问题交给链接器去处理，但是模板的实现文件中没有该实例，无法找到符号，所以一般这种问题的抱错都是”ld error“。

  那么为什么模板不被使用就无法编译呢。我们知道，c语言对内存的管理是底层的面向系统的，如果类型参数化的模板类而言，无法得知模板的类型，就无法得知模板类的占用内存。编译器无法为一个不知道大小的类分配内存。所谓模板类，不是一个类，而是一个生成类的模板。

  在分离式编译的环境下，编译器编译某一个cpp文件时并不知道另一个cpp文件的存在，也不会去查找（当遇到未决符号时它会寄希望于链接器）。

  这种模式在没有模板的情况下运行良好，但遇到模板时就傻眼了，因为模板仅在需要的时候才会实例化出来。所以，当编译器只看到模板的声明时，它不能实例化该模板，只能创建一个具有外部链接的符号并期待链接器能够将符号的地址决议出来。

  然而当实现该模板的cpp文件中没有用到模板的实例时，编译器懒得去实例化，所以，整个工程中就找不到一行模板实例的二进制代码，于是链接器也黔驴技穷了。
- 用户态和内核态的区别
  - 内核态与用户态是操作系统的两种运行级别，当程序运行在3级特权级上时，就可以称之为运行在用户态。**因为这是最低特权级，是普通的用户进程运行的特权级，大部分用户直接面对的程序都是运行在用户态；**
  - 当程序运行在0级特权级上时，就可以称之为运行在内核态。
  - 运行在用户态下的程序不能直接访问操作系统内核数据结构和程序。当我们在系统中执行一个程序时，大部分时间是运行在用户态下的，在其需要操作系统帮助完成某些它没有权力和能力完成的工作时就会切换到内核态（比如操作硬件）。
  - 这两种状态的主要差别是
    - 处于用户态执行时，进程所能访问的内存空间和对象受到限制，其所处于占有的处理器是可被抢占的

    - 处于内核态执行时，则能访问所有的内存空间和对象，且所占有的处理器是不允许被抢占的。
  - 用户态→内核态：1、系统调用 2、异常 3、中断（硬盘的读写）
- 虚函数表 的底层数据结构 数组
- 红黑树 复杂度 $O(log_n)$ 哈希表是O(1)
- condition_variable wait() 函数 内部实现

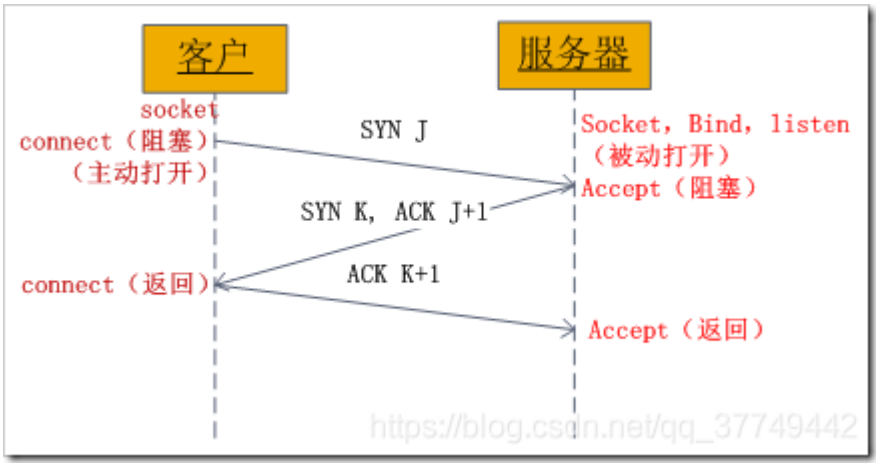
  1、void wait (unique_lock<mutex>& lck);
  2、void wait (unique_lock<mutex>& lck, Predicate pred);
  第一种形式只有一个参数unique_lock<mutex>&，调用wait时，若参数互斥量lck被锁定，则wait会阻塞。

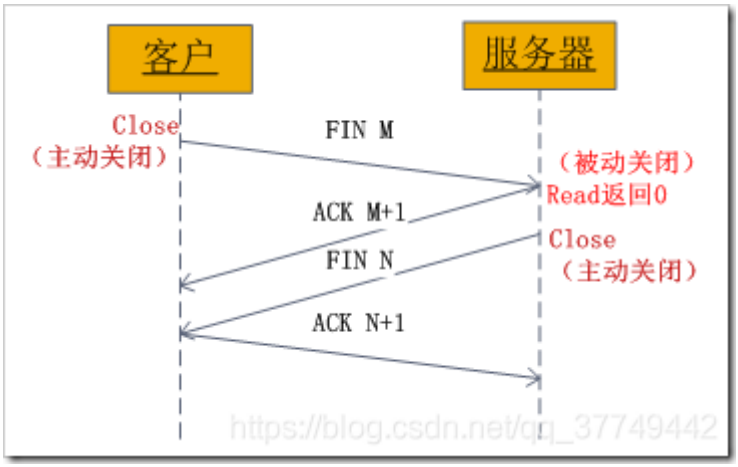
  第二种形式除了unique_lock<mutex>&参数外，第二个参数pred，即函数指针。当函数运行到该wait()函数时，若互斥量lck被锁定（？存疑），或者pred()返回值为false，满足两个条件之一，则wait阻塞。其等同于下面的形式：

  ```
  while (!pred()) wait(lck);
  ```
- 快速排序 归并排序的时间复杂度 $O(nlog_n)$
- 内存页面置换算法
  - FIFO LRU LFU
- 在kill掉一个进程之后 操作系统做了什么工作
  - kill -9 表示发送 SIGKILL 信号，这个信号是不能被用户程序捕获的，它的处理过程完全在内核态完成，核心过程在于调用 do_group_exit 来执行所谓的“组退出”过程杀死整个线程组。

    do_group_exit 函数会杀死 current 线程组中的其他进程（如果存在的话），它会向所有不同于 current 的同一个 tgid 中的其它进程发送 SIGKILL 信号，这些进程最终都将调用 do_exit 函数，从而终止运行。

    do_exit 是一个相当复杂的函数，它的主要目的是回收进程使用的资源，这也是我们调用了 kill -9 没有出问题的根本原因——内核替我们完成了这些必要的回收工作。
  - 进程在退出系统之前要释放所有的资源，在任务创建过程中从**父进程继承的资源**有存储空间、打开文件、工作目录、信号处理表等等，相应的在 do_exit 中就有__exit_mm()、__exit_files()、__exit_sighand。

    - 正是因为内核在 do_exit 中针对用户态程序使用的不同资源进行了回收，这才让 kill -9 这样的方式不至于导致存储空间泄露。
    - kill命令用于终止Linux进程，默认情况下，如果不指定信号，kill 等价于kill -15。kill -15执行时，系统向对应的程序发送SIGTERM(15)信号，该信号是可以被执行、阻塞和忽略的，所以应用程序接收到信号后，可以做一些准备工作，再进行程序终止。有的时候，kill -15无法终止程序，因为他可能被忽略，这时候可以使用kill -9，系统会发出SIGKILL(9)信号，该信号不允许忽略和阻塞，所以应用程序会立即终止。这也会带来很多副作用，如数据丢失等，所以，在非必要时，不要使用kill -9命令，尤其是那些web应用、提供RPC服务、执行定时任务、包含长事务等应用中，因为 kill -9 没给spring容器、tomcat服务器、dubbo服务、流程引擎、状态机等足够的时间进行收尾。
- 25匹马找出最快的5匹
- 三次握手发生在socket哪些函数:
    - 

    - 

- static inline : c++17 static inline 可以不用在类外重新定义
- 前置申明 可以 减少编译和连接的时间
- placement new （vector resize）

- STL分配器
- 自己实现itoa
- main 函数return 后做了什么工作
- 谁调用的main()是真正的入口吗
  - 通常，main()被启动代码调用，而启动代码是由编译器添加到程序中的，是程序和操作系统之间的桥梁。事实上，函数头int main()描述的是main()和操作系统之间的接口。
  - 符号_start是程序的入口点。也就是说，该符号的地址是程序启动时跳转到的地址。通常，名称为_start的函数由名为crt0.o的文件提供，该文件包含C运行时环境的启动代码。它设置一些东西，填充参数数组argv，计算那里有多少参数，然后调用main。返回main后，将调用exit
- 线程detach后能保证安全推出吗
- MFC消息机制：
  - MFC使用消息映射机制来处理消息，具体表现就是消息和消息处理函数一一对应的消息映射表，以及消息处理函数的声明和是实现。当窗口收到消息时，会在消息映射表中寻找对应的处理函数，然后由消息处理函数做出相应的处理。
- 自定义类型存入vector需要注意什么？深拷贝 移动语义等 符号重载
- ```
  class A{
  public:
    void outputP(){}
  };

  A *a = nullptr;
  请问a->output()会不会崩溃？
  不会崩溃，会转化成void output(A* this){},只要函数里不操作this指针就不会崩溃，因为函数地址编译器就确认了，是可以调用的，防止此类错误可以用assert(this);
  ```

算法题：

- 和为0的三元数
- 字符串大数相减
- 力扣42.接雨水
  ```

- 手撕，判断一个树是二叉搜索树，自己建，树的遍历 非递归
- 合并区间
- 合并k个链表（不让写归并）
- 用两个栈实现一个队列，包括入队出队和获取队的长度 （入栈 s1，出栈 s2）
- 用c++宏实现3个数比较大小
- LC347 （优先队列）
- 求二叉树宽度 （层遍历）
- 划分为k个相等的子集
- 部分翻转链表
- LRU/LFU
- bitmap/布隆过滤器
- 子集
- 打家劫舍
- 岛屿
- 子集/组合
- 股票问题
- 正则表达式问题

地平线:

1、拷贝构造函数 移动构造函数等实现

2、memset memcpy底层实现 对性能有什么影响

3、一个空指针的类调用,能否调用成员函数 和 虚函数? 过程有什么不同?

```
1 class Foo {
2 public:
3   void f1();
4   virtual f2();
5 }
```

```
Foo *f = nullptr;
f->f2();
f->f1();
```

3、单例

4、vector resize 如何分配内存构造对象的? placement new

4、实现自旋锁

```
class SpinLock {
public:
 SpinLock() : flag_(false) {}
 void lock() {
   bool expect = false;
   while (!flag_.compare_exchange_weak(expect, true)) {
     expect = false;
   }
 }

 void unlock() {
   flag_.store(false);
 }

private:
 std::atomic<bool> flag_;

};


bool compare_and_swap(int *pAddr, int nExpected, int nNew)
{
    if(*pAddr == nExpected)
    {
        *pAddr = nNew;
        return true;
    }
    else
        return false;
}
```

5、shared_ptr weak_ptr weak_ptr如何取得数据，如何升级为shared_ptr ?lock获取一个相同的shared_ptr获取数据

程序题

1、生产者与消费者

```
#pragma once
#include <iostream>
#include <deque>
#include <mutex>
#include <condition_variable>
using namespace std;

template<typename T>
class Foo {
public:
        Foo(int s) :_size(s) {}
        ~Foo() {}

        void Product(T& t) {
                std::unique_lock<std::mutex> lck(_mutex);
                while (_dq.size() >= _size) {
                        _full.wait(lck);
                }

                _dq.push_back(t);
                _empty.notify_one();
        }

        void Con(T& t) {
                std::unique_lock<std::mutex> lck(_mutex);
                while (_dq.empty())
                {
                        _empty.wait(lck);
                }

                if (_dq.empty())
                        return;
                else {
```

```
                        t = _dq.front();
                        _dq.pop_front();
                        _full.notify_one();
                        return;
                }

        }

        bool isEmpty() const {
                return _dq.empty();
        }
private:
        std::deque<T> _dq;
        size_t _size;
        std::mutex _mutex;
        std::condition_variable _empty, _full;
};
```