有鱼科技

c 语言字节对齐的目的，和系统访问的时候是怎么访问的

https://blog.csdn.net/lanzhihui_10086/article/details/44353381

对齐跟数据在内存中的位置有关。如果一个变量的内存地址正好位于它长度的整数倍，他就被称做自然对齐。比如在32位cpu下，假设一个整型变量的地址为0x00000004，那它就是自然对齐的。

需要字节对齐的根本原因在于CPU访问数据的效率问题。假设上面整型变量的地址不是自然对齐，比如为0x00000002，则CPU如果取它的值的话需要访问两次内存，第一次取从0x00000002-0x00000003的一个short，第二次取从0x00000004-0x00000005的一个short然后组合得到所要的数据，如果变量在0x00000003地址上的话则要访问三次内存，第一次为char，第二次为short，第三次为char，然后组合得到整型数据。而如果变量在自然对齐位置上，则只要一次就可以取出数据。一些系统对对齐要求非常严格，比如sparc系统，如果取未对齐的数据会发生错误.

平台原因(移植原因):不是所有的硬件平台都能访问任意地址上的任意数据的;某些硬件平台只能在某些地址 处取某些特定类型的数据,否则抛出硬件异常。

常用端口：

https://blog.csdn.net/zhanghuiyu01/article/details/80830045

端口对照表

HTTP协议代理服务器常用端口号：80/8080/3128/8081/9098

FTP（文件传输）协议代理服务器常用端口号：21

SSH（安全登录）、SCP（文件传输）、端口号重定向，默认的端口号为22/tcp

Telnet（远程登录）协议代理服务器常用端口号：23

TFTP（Trivial File Transfer Protocol，不安全的文本传送），默认端口号为69/udp

HTTPS（securely transferring web pages）服务器，默认端口号为443/tcp 443/udp

Oracle 数据库，默认的端口号为1521

指针函数

指针函数，函数指针

指针函数是指带指针的函数，即本质是一个**函数**。当一个函数声明其返回值为一个指针时，实际上就是返回一个地址给调用函数，以用于需要指针或地址的表达式中。

有语句char str[] = "abcde";请问表达式sizeof(str)的值是（）

6

宏和内敛函数的的区别是啥

参考网址

内联函数是代码被插入到调用者代码处的函数。如同 #define 宏，内联函数通过避免被调用的开销来提高执行效率，尤其是它能够通过调用（“过程化集成”）被编译器优化。 宏定义不检查函数参数，返回值什么的，只是展开，相对来说，内联函数会检查参数类型，所以更安全。　　内联函数和宏很类似，而区别在于，宏是由预处理器对宏进行替代，而内联函数是通过编译器控制来实现的。而且内联函数是真正的函数，只是在需要用到的时候，内联函数像宏一样的展开，所以取消了函数的参数压栈，减少了调用的开销。你可以象调用函数一样来调用内联函数，而不必担心会产生于处理宏的一些问题。

当然，内联函数也有一定的局限性。就是函数中的执行代码不能太多了，如果，内联函数的函数体过大，一般的编译器会放弃内联方式，而采用普通的方式调用函数。这样，内联函数就和普通函数执行效率一样了。

阻塞非祖塞同步异步区别

阻塞和非阻塞:调用者在事件没有发生的时候,一直在等待事件发生,不能去处理别的任务这是阻塞。调用者在事件 没有发生的时候,可以去处理别的任务这是非阻塞。 同步和异步:调用者必须循环自去查看事件有没有发生,这种情况是同步。调用者不用自己去查看事件有没有发生, 而是等待着注册在事件上的回调函数通知自己,这种情况是异步

防止死锁

- 间接的死锁预防（防止死锁条件1~3）
    - **预防互斥**：一般来说，不可能禁止
    - **预防占有且等待**：可以要求进程一次性地请求所有需要的资源，并且阻塞进程直到所有请求都同时满足。这种方法在两个方面是低效的：1）为了等待满足其所有请求的资源，进程可能被阻塞很长时间。但实际上只要有一部分资源，就可以继续执行；2）分配的资源有可能有相当长的一段时间不会被使用，且在此期间，这些资源不能被其它进程使用；除此之外，一个进程可能事先并不会知道它所需要的所有资源
    - **预防不可抢占**：有几种方法：1）如果占用某些资源的进程进一步申请资源时被拒，则释放其占用的资源；2）如果一个进程请求当前被另一个进程占有的一个资源，操作系统可以抢占另一个进程，要求它释放资源(方法2只有在任意两个进程优先级不同时，才能预防死锁)；此外，通过预防不可抢占来预防死锁的方法，只有在资源状态可以很容易保存和恢复的情况下才实用
- 直接的死锁预防（防止死锁条件4）
    - **预防循环等待**：可以通过定义资源类型的线性顺序来预防，如果一个进程已经分配到了R类型的资源，那么它接下来请求的资源只能是那些排在R类型之后的资源；这种方法可能是低效的，会使进程执行速度变慢，并且可能在没有必要的情况下拒绝资源访问

c++防死锁

什么是死锁 线程死锁是指由于两个或者多个线程互相持有对方所需要的资源，导致这些线程处于等待状态，无法前往执行。当线程进入对象的synchronized代码块时，便占有了资源，直到它退出该代码块或者调用wait方法，才释放资源，在此期间，其他线程将不能进入该代码块。当线程互相持有对方所需要的资源时，会互相等待对方释放资源，如果线程都不主动释放所占有的资源，将产生死锁。

2、常见的死锁场景 一个线程已经拿到了锁，未释放锁，但是又尝试拿同样的锁，这是就会死锁 两个以上线程A B ...，两个以上锁a b...，线程A已经拿到a锁，线程B已经拿到b锁，但是线程A在没有释放a锁尝试获取b锁，线程B没有释放b锁尝试获取a锁，这时也会发生死锁 3、预防死锁的办法 加锁的时候使用try_lock()，如果获取不到锁，那么就释放自己手里面得所有锁， 可以在加锁的过程中对metux进行地址的比较永远从最小地址开始加锁，这样的话就能保证所有的线程都按同一个顺序加锁，这样的话也能避免死锁

可以设置锁的属性为PTHREAD_MUTEX_ERRCHECK，这种锁如果发生死锁会返回错误，不过效率要稍微低一些

服务端的设计

???

c++设置一个类只能在栈上申请不能在堆上申请

1、只能建立在堆上

类对象只能建立在堆上，就是不能静态建立类对象，即不能直接调用类的构造函数。
容易想到将构造函数设为私有。在构造函数私有之后，无法在类外部调用构造函数来构造类对象，只能使用new运算符来建立对象。然而，前面已经说过，new运算符的执行过程分为两步，C++提供new运算符的重载，其实是只允许重载operator new()函数，而operator()函数用于分配内存，无法提供构造功能。因此，这种方法不可以。 当对象建立在栈上面时，是由编译器分配内存空间的，调用构造函数来构造栈对象。当对象使用完后，编译器会调用析构函数来释放栈对象所占的空间。编译器管理了对象的整个生命周期。如果编译器无法调用类的析构函数，情况会是怎样的呢？比如，类的析构函数是私有的，编译器无法调用析构函数来释放内存。所以，编译器在为类对象分配栈空间时，会先检查类的析构函数的访问性，其实不光是析构函数，只要是非静态的函数，编译器都会进行检查。如果类的析构函数是私有的，则编译器不会在栈空间上为类对象分配内存。 因此，将析构函数设为私有，类对象就无法建立在栈上了。代码如下：

```
class A
{
public:
    A(){}
    void destory(){delete this;}
private:
    ~A(){}
};
```

试着使用A a;来建立对象，编译报错，提示析构函数无法访问。这样就只能使用new操作符来建立对象，构造函数是公有的，可以直接调用。类中必须提供一个destory函数，来进行内存空间的释放。类对象使用完成后，必须调用destory函数。 上述方法的一个缺点就是，无法解决继承问题。如果A作为其它类的基类，则析构函数通常要设为virtual，然后在子类重写，以实现多态。因此析构函数不能设为private。还好C++提供了第三种访问控制，protected。将析构函数设为protected可以有效解决这个问题，类外无法访问protected成员，子类则可以访问。

另一个问题是，类的使用很不方便，使用new建立对象，却使用destory函数释放对象，而不是使用delete。（使用delete会报错，因为delete对象的指针，会调用对象的析构函数，而析构函数类外不可访问）这种使用方式比较怪异。为了统一，可以将构造函数设为protected，然后提供一个public的static函数来完成构造，这样不使用new，而是使用一个函数来构造，使用一个函数来析构。代码如下，类似于单例模式：

```
class A
{
protected:
    A(){}
    ~A(){}
public:
    static A* create()
    {
        return new A();
    }
    void destory()
    {
        delete this;
    }
};
```

这样，调用create()函数在堆上创建类A对象，调用destory()函数释放内存。

2、只能建立在栈上

只有使用new运算符，对象才会建立在堆上，因此，只要禁用new运算符就可以实现类对象只能建立在栈上。将operator new()设为私有即可。代码如下：

```
class A
{
private:
    void* operator new(size_t t){}     // 注意函数的第一个参数和返回值都是固定的
    void operator delete(void* ptr){} // 重载了new就需要重载delete
public:
    A(){}
    ~A(){}
};
```

哈希扩容

看过HashMap源码的人（大神请忽略）可能会有些疑问，HashMap究竟什么时候扩容？扩容的条件是什么？好的，接下啦我给大家介绍一下HashMap扩容相关的一些参数。

容量（capacity）：HashMap中数组的长度 **加载因子(Load factor)**：HashMap在其容量自动增加前可达到多满的一种尺度，默认加载因子 = 0.75 **扩容阈值（threshold）**：当哈希表的大小 ≥ 扩容阈值时，就会扩容哈希表 下面再用通俗一点的话说明一下 我们都知道HashMap的初始值是16（1,<<4）,负载因子0.75（听说这个值是经过大量实践算出来的，这个值设定最合理），初始值16指的是数组的长度（1<<4是2的4次方，这样写计算机执行更快），当数组的容量达到12（16*0.75）时，这时开始扩容，扩容为32（1<<5即2的5次方），每次扩容按照2的倍数递增，扩容是为了减少hash碰撞，让链表的数据更少（最好链表上就一个数据，即为数组的下标数据）

http://m.nowcoder.com/discuss/72668?type=0&pos=28

JDK8中HashMap扩容涉及到的加载因子和链表转红黑树的知识点经常被作为面试问答题，本篇将对这两个知识点进行小结。

链表转红黑树为什么选择数字8

在JDK8及以后的版本中，HashMap引入了红黑树结构，其底层的数据结构变成了数组+链表或数组+红黑树。添加元素时，若桶中链表个数超过8，链表会转换成红黑树。之前有写过篇幅分析选择数字8的原因，内容不够严谨。最近重新翻了一下HashMap的源码，发现其源码中有这样一段注释：

> Because TreeNodes are about twice the size of regular nodes, we use them only when bins contain enough nodes to warrant use (see TREEIFYTHRESHOLD). And when they become too small (due to removal or resizing) they are converted back to plain bins. In usages with well-distributed user hashCodes, tree bins are rarely used. Ideally, under random hashCodes, the frequency of nodes in bins follows a Poisson distribution (http://en.wikipedia.org/wiki/Poissondistribution) with a parameter of about 0.5 on average for the default resizing threshold of 0.75, although with a large variance because of resizing granularity. Ignoring variance, the expected occurrences of list size k are (exp(-pow(0.5, k) / factorial(k)). The first values are: 0: 0.60653066 1: 0.30326533 2: 0.07581633 3: 0.01263606 4: 0.00157952 5: 0.00015795 6: 0.00001316 7: 0.00000094 8: 0.00000006 more: less than 1 in ten million

翻译过来大概的意思是：理想情况下使用随机的哈希码，容器中节点分布在hash桶中的频率遵循泊松分布(具体可以查看http://en.wikipedia.org/wiki/Poisson_distribution)，按照泊松分布的计算公式计算出了桶中元素个数和概率的对照表，可以看到链表中元素个数为8时的概率已经非常小，再多的就更少了，所以原作者在选择链表元素个数时选择了8，是根据概率统计而选择的。

默认加载因子为什么选择0.75

HashMap有两个参数影响其性能：初始容量和加载因子。容量是哈希表中桶的数量，初始容量只是哈希表在创建时的容量。加载因子是哈希表在其容量自动扩容之前可以达到多满的一种度量。当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行扩容、rehash操作（即重建内部数据结构），扩容后的哈希表将具有两倍的原容量。

通常，加载因子需要在时间和空间成本上寻求一种折衷。加载因子过高，例如为1，虽然减少了空间开销，提高了空间利用率，但同时也增加了查询时间成本；加载因子过低，例如0.5，虽然可以减少查询时间成本，但是空间利用率很低，同时提高了rehash操作的次数。在设置初始容量时应该考虑到映射中所需的条目数及其加载因子，以便最大限度地减少rehash操作次数，所以，一般在使用HashMap时建议根据预估值设置初始容量，减少扩容操作。

选择0.75作为默认的加载因子，完全是时间和空间成本上寻求的一种折衷选择，至于为什么不选择0.5或0.8，笔者没有找到官方的直接说明，在HashMap的源码注释中也只是说是一种折中的选择。

epoll

epoll 对文件描述符的操作有两种模式: LT (level trigger)和 ET (edge trigger)。 LT 模式是默认模式

1. LT 模式 LT(level triggered,水平触发) 是缺省的工作方式,并且同时支持 block 和 no-block socket. 在这种做法中,内核告诉你一个文件 描述符是否就绪了,然后你可以对这个就绪的 fd 进行 IO 操作。如果你不作任何操作,内核还是会继续通知你的。
2. ET 模式 ET(edge-triggered，边缘触发) 是高速工作方式,只支持 no-block socket(非阻塞套接字) 。在这种模式下,当描述符从未就绪变 为就绪时,内核通过 epoll 告诉你。然后它会假设你知道文件描述符已经就绪,并且不会再为那个文件描述符发送更 多的就绪通知,直到你做了某些操作导致那个文件描述符不再为就绪状态了 (比如,你在

发送,接收或者接收请求, 或者发送接收的数据少于一定量时导致了一个 EWOULDBLOCK 错误)。但是请注意,如果一直不对这个 fd 作 IO 操作 (从而导致它再次变成未就绪) ,内核不会发送更多的通知 (only once) ET 模式在很大程度上减少了 epoll 事件被重复触发的次数,因此效率要比 LT 模式高。 epoll 工作在 ET 模式的时候,必须 使用非阻塞套接口,以避免由于一个文件句柄的阻塞读 / 阻塞写操作把处理多个文件描述符的任务饿死。 3 、 LT 模式与 ET 模式的区别如下:模式:当 epoll_wait 检测到描述符事件发生并将此事件通知应用程序,应用程序可以不立即处理该事件。下次调用 epoll_wait 时,会再次响应应用程序并通知此事件。 ET 模式:当 epoll_wait 检测到描述符事件发生并将此事件通知应 用程序,应用程序必须立即处理该事件。如果不处理,下次调用 epoll_wait 时,不会再次响应应用程序并通知此事 件。

static

https://blog.51cto.com/844133395/1904814

类的的变量或者函数加上static的应用场景

为什么要引入static

函数内部定义的变量，在程序执行到它的定义处时，编译器为它在栈上分配空间，大家知道，函数在栈上分配的空间在此函数执行结束时会释放掉，这样就产生了一个问题: 如果想将函数中此变量的值保存至下一次调用时，如何实现？最容易想到的方法是定义一个全局的变量，但定义为一个全局变量有许多缺点，最明显的缺点是破坏了此变量的访问范围（使得在此函数中定义的变量，不仅仅受此函数控制）。

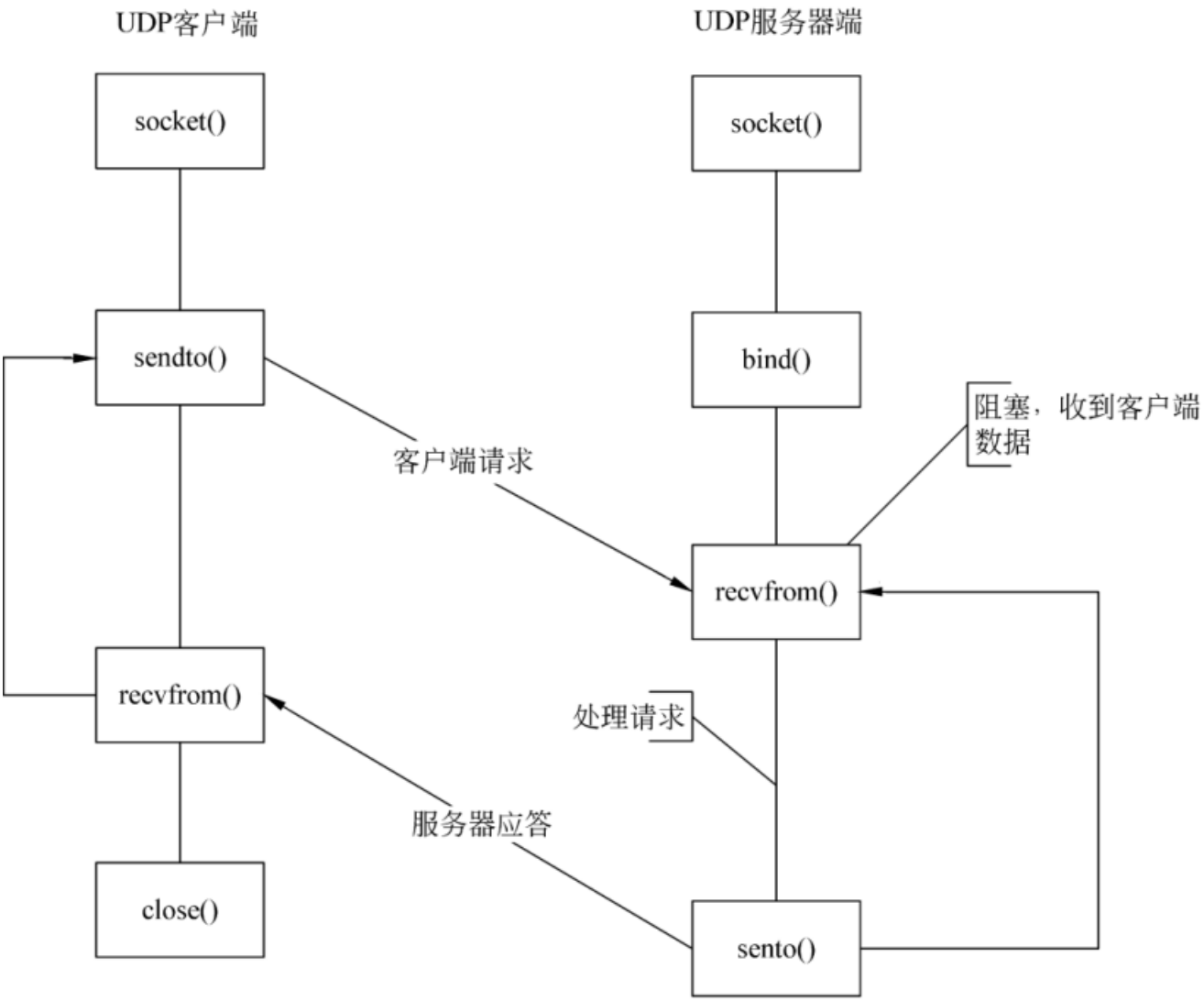
什么时候用static

需要一个数据对象为

ｔｃｐ和ｕｄｐ的区别

) TCP 和 UDP 区别 1) 连接 TCP 是面向连接的传输层协议,即传输数据之前必须先建立好连接。 UDP 无连接。 2) 服务对象 TCP 是点对点的两点间服务,即一条 TCP 连接只能有两个端点; UDP 支持一对一,一对多,多对一,多对多的交互通信。 3) 可靠性 TCP 是可靠交付:无差错,不丢失,不重复,按序到达。 UDP 是尽最大努力交付,不保证可靠交付。 4)拥塞控制,流量控制 TCP 有拥塞控制和流量控制保证数据传输的可靠性。 UDP 没有拥塞控制,网络拥塞不会影响源主机的发送效率。 5) 报文长度 1是动态报文长度,即 TCP 报文长度是根据接收方的窗口大小和当前网络拥塞情况决定的。UDP 面向报文,不合并,不拆分,保留上面传下来报文的边界。 6) 首部开销 TCP 首部开销大,首部 20 个字节。 UDP 首部开销小, 8 字节。(源端口,目的端口,数据长度,校验和) 2) TCP 和 UDP 适用场景 从特点上我们已经知道, TCP 是可靠的但传输速度慢, UDP 是不可靠的但传输速度快。因此在选用具体协议通信 时,应该根据通信数据的要求而决定。 若通信数据完整性需让位与通信实时性,则应该选用 TCP 协议(如文件传输、重要状态的更新等;反之,则使用 UDP 协议(如视频传输、实时通信等)。 包大小: mtu 最大传输单元 UDP 包的大小就应该是 1500 - IP 头 (20) - UDP 头 (8) = 1472(Bytes) TCP 包的大小就应该是 1500 - IP 头 (20) - TCP 头 TCP (20) = 1460 (Bytes)

怎样用udp编程



一面大数据，１０万个数据有重复的找出top10　分步骤解说

1.哈希成１０组数据，

2.每组数据找出ｔｏｐ１０

3.合并

２面写一个函数，输入n，求斐波那契（Fibonacci）数列的第n项。斐波那契数列的定义如下：

https://mp.weixin.qq.com/s/9hcpIaFFuVVQ4MhT2aKNAA

２面有十万数量级的IP网段，网段是从小到大排列的，知道每个网段的首地址和末地址，但是网段的大小是随机的，然后给你个IP地址，请问怎么查到它属于哪个网段?

３面两个单链表相加

一次遍历

想到用递归的方法

３面链表中位数

４面２３５硬币数无数个可以怎样凑齐１００

七牛云

close_wait发生在呢

第二次挥手后，　我打错了

extern

https://www.cnblogs.com/yc_sunniwell/archive/2010/07/14/1777431.html

基本解释：extern可以置于变量或者函数前，以标示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。此外extern也可用来进行链接指定。

epoll　ET　LT　模式

epoll 对文件描述符的操作有两种模式: LT (level trigger)和 ET (edge trigger)。 LT 模式是默认模式

1. LT 模式 LT(level triggered,水平触发) 是缺省的工作方式,并且同时支持 block 和 no-block socket. 在这种做法中,内核告诉你一个文件 描述符是否就绪了,然后你可以对这个就绪的 fd 进行 IO 操作。如果你不作任何操作,内核还是会继续通知你的。
2. ET 模式 ET(edge-triggered，边缘触发) 是高速工作方式,只支持 no-block socket(非阻塞套接字) 。在这种模式下,当描述符从未就绪变 为就绪时,内核通过 epoll 告诉你。然后它会假设你知道文件描述符已经就绪,并且不会再为那个文件描述符发送更 多的就绪通知,直到你做了某些操作导致那个文件描述符不再为就绪状态了 (比如,你在发送,接收或者接收请求, 或者发送接收的数据少于一定量时导致了一个 EWOULDBLOCK 错误)。但是请注意,如果一直不对这个 fd 作 IO 操作 (从而导致它再次变成未就绪) ,内核不会发送更多的通知 (only once) ET 模式在很大程度上减少了 epoll 事件被重复触发的次数,因此效率要比 LT 模式高。 epoll 工作在 ET 模式的时候,必须 使用非阻塞套接口,以避免由于一个文件句柄的阻塞读 / 阻塞写操作把处理多个文件描述符的任务饿死。 3 、 LT 模式与 ET 模式的区别如下:模式:当 epoll_wait 检测到描述符事件发生并将此事件通知应用程序,应用程序可以不立即处理该事件。下次调用 epoll_wait 时,会再次响应应用程序并通知此事件。 ET 模式:当 epoll_wait 检测到描述

符事件发生并将此事件通知应 用程序,应用程序必须立即处理该事件。如果不处理,下次调用 epoll_wait 时,不会再次响应应用程序并通知此事 件。

3xx错误码的含义

1xx :指示信息 -- 表示请求已接收,继续处理。 2xx :成功 -- 表示请求已被成功接收、理解、接受。 3xx :重定向 -- 要完成请求必须进行更进一步的操作。 4xx :客戶端错误 -- 请求有语法错误或请求无法实现。 5xx :服务器端错误 -- 服务器未能实现合法的请求。

fork 和vfork　exec X

https://www.jb51.net/article/133512.htm

vfrok函数拷贝数据段的例子

将上述的代码中的fork函数改成vfork查看运行结果：

```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>

int main()
{
    pid_t pid;
    int cnt = 0;
    pid = vfork();
    if(pid<0)
        printf("error in fork!\n");
    else if(pid == 0)
    {
        cnt++;
        printf("cnt=%d\n",cnt);
        printf("I am the child process,ID is %d\n",getpid());
        _exit(0);
    }
    else if(pid > 0)
    {
        cnt++;
        printf("cnt=%d\n",cnt);
        printf("I am the parent process,ID is %d\n",getpid());
    }
    return 0;
}
```

运行结果

```
cnt=1
I am the child process,ID is 4711
cnt=1
I am the parent process,ID is 4710
段错误
```

本来vfork（）是共享数据段的，结果应该是2，为什么不是预想的2 呢？先看一个知识点：**vfork 和fork 之间的另一个区别是：vfork 保证子进程先运行，在她调用exec 或exit 之后父进程才可能被调度运行。**如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致**死锁**。这样上面程序中的fork()改成vfork()后，vfork()创建子进程并没有调用exec 或exit，所以最终出现死锁。 对程序进行修改：

```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>

int main()
{
    pid_t pid;
    int cnt = 0;
    pid = vfork();
    if(pid<0)
        printf("error in fork!\n");
    else if(pid == 0)
    {
        cnt++;
        printf("cnt=%d\n",cnt);
        printf("I am the child process,ID is %d\n",getpid());
       _exit(0);
    }
    else
    {
        cnt++;
        printf("cnt=%d\n",cnt);
        printf("I am the parent process,ID is %d\n",getpid());
    }
    return 0;

}
```

如果没有*exit（0）的话，子进程没有调用exec 或exit，所以父进程是不可能执行的，在子进程调用exec 或exit 之后父进程才可能被调度运行。所以我们加上*exit(0);使得子进程退出，父进程执行，这样else 后的语句就会被父进程执行，又因在子进程调用exec 或exit之前与父进程数据是共享的,所以子进程退出后把父进程的数据段count改成1 了，子进程退出后，父进程又执行，最终就将count变成了2，看下实际运行结果：

```
cnt=1
I am the child process,ID is 4711
cnt=2
I am the parent process,ID is 4710
```

为什么会有vfork，因为以前的fork 很傻， 它创建一个子进程时，将会创建一个新的地址 空间，并且拷贝父进程的资源，而往往在子进程中会执行exec 调用，这样，前面的拷贝工 作就是白费力气了，这种情况下，聪明的人就想出了vfork，它产生的子进程刚开始暂时与 父进程共享地址空间（其实就是线程的概念了），因为这时候子进程在父进程的地址空间中 运行，所以子进程不能进行写操作，并且在儿子 霸占”着老子的房子时候，要委屈老子一 下了，让他在外面歇着（阻塞），一旦儿子执行了exec 或者exit 后，相 于儿子买了自己的

房子了，这时候就相 于分家了。

const的用法

参考网址

首先我们来了解一下现在所学的C标准，分别是C89、C99和C++99，C89标准的C语言中的const和C++中的const是有区别的，而在C99标准的C语言中是相同的。我们目前所学的C语言标准为C89标准。

const修饰的量为一个常量即不能被修改的量。但在C语言（C89）中的const可以不初始化但后续也就无法对其赋值，所以尽管不初始化不会出错。但要使用const修饰的量就需要对其进行初始化。另外，既然const修饰的量为一常量那么const修饰的量能不能用作数组的下标?

```
//main.c
#include<stdio.h>
int main()
{
    const int a = 10;
    //int array[a] = {0};//error C2057: 应输入常量表达式

    //则可以得知在C语言中const修饰的量为一个常变量不是常量
    int *p = &a;
    *p = 20;
    printf("%d   %d\n", a, *p);
    return 0;
    //20  20
}
```

程序输出结果都为20，在这里我们就可以知道常变量和普通变量的区别在于常变量不能用于左值，其余的用法和普通常量一样。在一个工程的多个.c文件中的const修饰的量可以共享。 在C++的代码中const必须初始化，这于const在C++中的编译规则(所有使用常量名字的地方去不替换成常量的初始值)有关。

```
//main.cpp

#include<iostream>
using namespace std;
int main()
{
    const int a = 20;
    int arr[a] = { 0 };//编译可以通过
    int *p = (int *)&a;
    *p = 30;

    cout << a << "  " << *p << endl;
    return 0;
//运行结果为20、30，发现并没有将const定义的量的值修改
}
```

由上可知在C++中const修饰的量为一个常量，可以做左值且不能修改它的值。只有当给const修饰量的值不明确的时候会退化成一个常变量。在一个C++工程中的多个.cpp文件中要用到某一个.cpp文件中const修饰的量是无法访问的，这是由于常量的符号类型为local的，只在当前文件可见，其余文件无法访问。如若想要访问这个const修饰的量，需在定义处加上extern。

```
一级指针和const结合的方式：
const int *p;         修饰int即*p 所指向的内容无法改变  常量指针
int const *p;         修饰int即*p 所指向的内容无法改变  常量指针
int *const p;         修饰int* 即*p  不能改变指向    指针常量
const int *const p;   修饰int和int*  指向和所指向的内容都不能改变
```

const修饰函数

```
#include<iostream>
using namespace std;

const char* test(char * ans) //const 后面必须和*搭配才有意义
{
    return ans;
}

const int lala(int a)
{
    return a;
}

int main()
{
    char* temp = test("abc");  //会报错 const char* test(char * ans) 函数返回的是一个const char*类型值，不能修改返回值
    const char* temp = test("abc");   //这样就不会错了
    cout << temp;
    int b = lala(1);   //毫无影响
    return 0;
}
```

const放在函数末尾，这种情况只能在类成员函数里使用

```
class test
{
private:
    int a;
public:
    test() :a(1){}
    int fun()const{ a++; return a; }  //这种情况错误，const放在成员函数名后面当作限定符，此函数不可更改类的属性，一般用于返回属性值
    int fun2()const{ return a; }
```

如果一定要在const成员函数中改变属性可以这样

```
class test
{
private:
    mutable int a;    //对属性加个mutable修饰符，就可以在const成员函数中修改属性了
public:
    test() :a(1){}
    int fun()const{ a++; return a; }  //这种情况错误，const放在成员函数名后面当作限定符，此函数不可更改类的属性，一般用于返回属性值
    int fun2()const{ return a; }
};
```

const引用，属于1中所说的一种例外，初始化 const引用时允许用任意表达式，只要该表达式的结果能转换为 引用类型即可。

```
也就是说，允许为一个const引用 绑定 非常量对象、字面值、甚至是一般表达式。
```

① const int &ci = 3； //正确，整型字面值常量绑定到 const引用

② int i = 1；

```
const int &cj = i;     //正确，非常量对象绑定到 const引用
```

③ const int i = 4；

```
const int &ck =  i; //正确，常量对象绑定到 const引用
```

④ const int i = 5；

```
int &r = i;     //错误，常量对象绑定到非const引用
```

位域

位域(位段)：

有时我们存储信息时并不需要用到一个完整的字节，而只需要几个二进制位。C语言提供了一种

数据结构，称为位域(位段)用于这种情况。

位域(位段)的定义：

```
struct name{

    类型名 位域名:位域长度;    // 位域长度不能大于8(即一个字节)
};
```

注意：

一个位域必须存储在同一个字节中，当有两个连续的位域，在一个字节中存放第一个位域后剩下

的空间不足以存放第二个位域，则第二个位域将存放在一个新的字节中，前一个字节多余的未补0

```
struct bits{
    int a:4;   // 第一个字节
    int b:5;   // 第二个字节
};
```

我们也可以显式使得某个位域存放在一个新的字节中，即使用空域

```
struct bits{
    int a:4;  // 第一个字节
    int :0;   // 空域
    int b:2;  // 第二个字节
};
```

当一个位域无位域名时，它只是用来作为填充或调整位置，不能被使用(int :4 仅仅是用来占4位)

四个强制指针类型转换

C++ 中四种类型转换是: static_cast, dynamic_cast, const_cast, reinterpret_cast 、 const_cast 用于将 const 变量转为非 const 2 、 static_cast 用于各种隐式转换,比如非 const 转 const , void* 转指针等 , static_cast 能用于多态向上转化,如果向下转能成功但是 不安全,结果未知; 3 、 dynamic_cast 用于动态类型转换。只能用于含有虚函数的类,用于类层次间的向上和向下转化。只能转指针或引用。向下转化时, 如果是非法的对于指针返回 NULL ,对于引用抛异常。要深入了解内部转换的原理。 向上转换:指的是子类向基类的转换 向下转换:指的是基类向子类的转换 它通过判断在执行到该语句的时候变量的运行时类型和要转换的类型是否相同来判断是否能够进行向下转换。 4 、 reinterpret_cast 几乎什么都可以转,比如将 int 转指针,可能会出问题,尽量少用;

union共用体，，，考官说的英语，没反应过来

在结构体（变量）中，结构的各成员顺序排列存储，每个成员都有自己独立的存储位置。联合(union)变量的所有成员共享同片存储区/内存。因此联合变量每个时刻里只能保存它的某一个成员的值。

联合变量也可以在定义时直接初始化，但这个初始化只能对第一个成员进行。例如下面的描述定义了一个联合变量，并进行了初始化。

```
1 uunion data
2 {
3     char n;
4     float f;
5 };
6 union data u1 = {3}; //只有u1.n被初始化
```

union的主要特征有：

union中可以定义多个成员，union的大小由最大的成员的大小决定；

union成员共享同一块大小的内存，一次只能使用其中的一个成员； 对union某一个成员赋值，会覆盖其他成员的值（但前提是成员所占字节数相同，当成员所占字节数不同时只会覆盖相应字节上的值，比如对char成员赋值就不会把整个int成员覆盖掉，因为char只占一个字节，而int占四个字节）； union量的存放顺序是所有成员都**从低地址开始存放的**。

因此，可以用联合(union)来判断CPU的大小端（印第安序）：

```
1 int checkCPU()
 2 {
 3     union w
 4     {
 5         int a;
 6         char b;
 7     }c;
 8
 9     c.a = 1;
10
11     return ( c.b == 1 );
12 }
```

实现一个ｓｔｒｉｎｇ类

```
/*************************************************************************
    > File Name: string.cpp
    > Author:
    > Mail:
    > Created Time: 2019年06月30日 星期日 18时06分23秒
 ************************************************************************/

#include<iostream>
#include <stdlib.h>
#include <string.h>
using namespace std;
class String{
    public:
    String(const char *str = ""){
        data = new char[strlen(str) + 1];
        strcpy(data,str);
    }
    String(const String & rhs) {
        data = new char[rhs.size() + 1];
        strcpy(data, rhs.c_str());
    }
    ~String(){
        delete [] data;
    }
    String& operator=(const String &rhs){
        if (this != &rhs) {
            delete [] data;
            data = new char[rhs.size() + 1];
            strcpy(data, rhs.c_str());
        }
    }
    size_t size () const {
        return strlen(data);
    }
    const char *c_str() const {
        return data;
    }
```

```
    private:
    char *data;
};



int main() {



    return 0;
}
```

可以通过”copy and swap“的策略来实现。其原理很简单：**即先对打算修改的对象做出一个副本（copy），在副本上做必要的修改。如果生成副本时出现任何异常，原对象依然能保证不变。如果修改成功，则通过不抛出任何异常的swap函数将副本和原对象进行交换（swap）。

```
String& String::operator=(const String &rhs) {
     String tmp(rhs);
     swap(data, tmp.data);
     return *this;
}
```

如果把参数中的引用传递换成值传递，那就相当于自动创建出一个临时对象来。如下是另一种等价形式。

```
String& String::operator=(String rhs) {
     swap(data, rhs.data);
     return *this;
}
```

在swap之后，相当于两指针指向的内存区域进行了互换。当退出函数时，tmp生命周期已到，将被销毁，但是它销毁时释放的是交换后的内存区域，即原来的data所指的内存区域；它本来指向的内存区域现在归data管了，不会被释放。

字节提前一面-还没给消息，应该是进入备胎池了

单例模式　脏读脏写

单例模式 7 中写法

java

c++

饿汉模式

饿汉模式 是指单例实例在程序运行时被立即执行初始化:

```
class Singleton
```

```
{
public:
    static Singleton& getInstance()
    {
        return m_data;
    }

private:
    static Singleton m_data; //static data member 在类中声明，在类外定义
    Singleton(){}
    ~Singleton(){}
};

Singleton Singleton::m_data;
```

这种模式的问题也很明显, 类现在是多态的, 但静态成员变量初始化顺序还是没保证: 假如有两个单例模式的类 ASingleton 和 BSingleton, 某天你想在 BSingleton 的构造函数中使用 ASingleton 实例, 这就出问题了. 因为 BSingleton m_data 静态对象可能先 ASingleton 一步调用初始化构造函数, 结果 ASingleton::getInstance() 返回的就是一个未初始化的内存区域, 程序还没跑就直接崩掉。恩，这只是理论分析的结果，下面给出一个简单的例子说明一下问题所在吧!

实例：ASingleton、BSingleton两个单例类，其中 ASingleton 的构造函数中使用到 BSingleton 的单例对象。

```
class ASingleton
{
public:
    static ASingleton* getInstance()
    {
        return &m_data;
    }
    void do_something()
    {
        cout<<"ASingleton do_something!"<<endl;
    }
protected:
    static ASingleton m_data; //static data member 在类中声明，在类外定义
    ASingleton();
    ~ASingleton() {}
};

class BSingleton
{
public:
    static BSingleton* getInstance()
    {
        return &m_data;
    }
    void do_something()
    {
        cout<<"BSingleton do_something!"<<endl;
    }
protected:
    static BSingleton m_data; //static data member 在类中声明，在类外定义
```

```
    BSingleton();
    ~BSingleton() {}
};

ASingleton ASingleton::m_data;
BSingleton BSingleton::m_data;

ASingleton::ASingleton()
{
    cout<<"ASingleton constructor!"<<endl;
    BSingleton::getInstance()->do_something();
}

BSingleton::BSingleton()
{
    cout<<"BSingleton constructor!"<<endl;
}
```

在这个测试例子中，我们将上述代码放在一个 main.cpp 文件中，其中 main 函数为空。

```
int main()
{
    return 0;
}
```

运行测试结果是：

```
ASingleton constructor!
BSingleton do_something!
BSingleton constructor!
```

奇怪了，为什么 BSingleton 的构造函数居然是在成员函数 do_something 之后调用的？

下面进行分析： 首先我们看到这个测试用例中，由于只有一个源文件，那么按从上到下的顺序进行编译运行。注意到：

```
ASingleton ASingleton::m_data;
BSingleton BSingleton::m_data;
```

这两个定义式，那么就会依次调用 ASingleton 的构造函数 和 BSingleton 的构造函数进行初始化。 一步一步来，首先是 ASingleto 的 m_data。 那么程序就会进入 ASingleton 的构造函数中执行，即：

```
ASingleton::ASingleton()
{
    cout<<"ASingleton constructor!"<<endl;
    BSingleton::getInstance()->do_something();
}
```

首先执行 cout，然后接着要获取 BSingleton 的单例，虽然说 BSingleton 的定义尚未执行，即 BSingleton BSingleton::m_data; 语句尚未执行到，但是 BSingleton 类中存在着其声明，那么还是可以调用到其 do_something 方法的。

ASingleton 的构造函数执行完毕，那么 ASingleton ASingleton::m_data; 语句也就执行结束了，即 ASingleton 单例对象 m_data 也就初始化完成了。

接下来执行 BSingleton BSingleton::m_data; 语句，那么也就是执行 BSingleton 的构造函数了。 所以就有了最终结果的输出了。

那么到此，我们或许会说：既然 ASingleton 的构造函数中要用到 BSingleton 单例对象，那么就先初始化 BSingleton 的单例对象咯，是的，我们可以调换一下顺序：

```
//ASingleton ASingleton::m_data;
//BSingleton BSingleton::m_data;
//修改成：
BSingleton BSingleton::m_data;
ASingleton ASingleton::m_data;
```

再运行一下，会发现输出的结果就正常了。

```
ASingleton constructor!
BSingleton constructor!
BSingleton do_something!
```

问题解决了，那么我们通过这个问题实例，我们对于 静态成员变量 初始化顺序没有保障 有了更深刻的理解了。 在这个简单的例子中，我们通过调换代码位置可以保障 静态成员变量 的初始化顺序。但是在实际的编码中是不可能的，class 文件声明在头文件(.h)中，class 的定义在源文件(.cpp)中。而类静态成员变量声明是在 .h 文件中，定义在 .cpp 文件中，那么其初始化顺序就完全依靠编译器的心情了。所以这也就是 类静态成员变量 实现单例模式的致命缺点。 当然，假如不出现这种：在某单例的构造函数中使用到另一个单例对象 的使用情况，那么还是可以接受使用的。

懒汉模式

单例实例只在第一次被使用时进行初始化:

```
class Singleton
{
public:
    static Singleton* getInstance()
    {
        if(! m_data) m_data = new Singleton();
        return m_data;
    }

private:
    static Singleton* m_data; //static data member 在类中声明，在类外定义
    Singleton(){}
    ~Singleton(){}
};
```

```
Singleton* Singleton::m_data = nullptr;
```

getInstance() 只在第一次被调用时为 m_data 分配内存并初始化. 嗯, 看上去所有的问题都解决了, 初始化顺序有保证, 多态也没问题. 但是只是看似没有问题而已，其实其中存在着两个问题：

①线程不安全：我们注意到在 static Singleton* getInstance() 方法中，是通过 if 语句判断 静态实例变量 是否被初始化来觉得是否进行初始化，那么在多线程中就有可能出现多次初始化的问题。比方说，有两个多线程同时进入到这个方法中，同时执行 if 语句的判断，那么就会出现两次两次初始化静态实例变量的情况。

②析构函数没有被执行： 程序退出时, 析构函数没被执行. 这在某些设计不可靠的系统上会导致资源泄漏, 比如文件句柄, socket 连接, 内存等等. 幸好 Linux / Windows 2000/XP 等常用系统都能在程序退出时自动释放占用的系统资源. 不过这仍然可能是个隐患。

对于这个问题, 比较土的解决方法是, 给每个 Singleton 类添加一个 destructor() 方法:

```
virtual bool destructor()
{
    // ... release resource
    if (nullptr != m_data)
    {
        delete m_data;
        m_data = nullptr;
    }

}
```

然后在程序退出时确保调用了每个 Singleton 类的 destructor() 方法, 这么做虽然可靠, 但却很是繁琐.

懒汉模式改进版：使用局部静态变量

```
class Singleton {
public:
    static Singleton& getInstance() {
        static Singleton theSingleton;
        return theSingleton;
    }
    /* more (non-static) functions here */

private:
    Singleton();                            // ctor hidden
    Singleton(Singleton const&);            // copy ctor hidden
    Singleton& operator=(Singleton const&); // assign op. hidden
    ~Singleton();                           // dtor hidden
};
```

但是这种方式也存在着很多的问题：

①任意两个单例类的构造函数不能相互引用对方的实例，否则会导致程序崩溃。如：

```
ASingleton& ASingleton::getInstance() {
    const BSingleton& b = BSingleton::getInstance();
    static ASingleton theSingleton;
    return theSingleton;
}

BSingleton& BSingleton::getInstance() {
    const ASingleton & b = ASingleton::getInstance();
    static BSingleton theSingleton;
    return theSingleton;
}
```

②多个 Singleton 实例相互引用的情况下, 需要谨慎处理析构函数. 如: 初始化顺序为 ASingleton »BSingleton » CSingleton 的三个 Singleton 类, 其中 ASingleton BSingleton 的析构函数调用了CSingleton 实例的成员函数, 程序退出时, CSingleton 的析构函数 将首先被调用, 导致实例无效, 那么后续 ASingleton BSingleton 的析构都将失败, 导致程序异常退出.

③在局部作用域下的静态变量在编译时，编译器会创建一个附加变量标识静态变量是否被初始化，会被编译器变成像下面这样（伪代码）：

```
static Singleton &Instance()
{
  static bool constructed = false;
  static uninitialized Singleton instance_;
  if (!constructed) {
    constructed = true;
    new(&s) Singleton; //construct it
  }
  return instance_;
}
```

那么，在多线程的应用场合下必须小心使用. 如果唯一实例尚未创建时, 有两个线程同时调用创建方法, 且它们均没有检测到唯一实例的存在, 便会同时各自创建一个实例, 这样就有两个实例被构造出来, 从而违反了单例模式中实例唯一的原则. 解决这个问题的办法是为指示类是否已经实例化的变量提供一个互斥锁 (虽然这样会降低效率).

加锁如下：

```
static Singleton &getInstance()
{
    Lock();
    //锁自己实现 static
    Singleton instance_;
    UnLock();

    return instance_;
}
```

但这样每次调用instance()都要加锁解锁，代价略大

五、终极方案 在前面的讨论中，单例类中的静态对象无论是作为静态局部对象还是作为类静态全局变量都有问题，那么有什么更好的解决方案呢？ boost 的实现方式是：单例对象作为静态局部变量，然后增加一个辅助类，并声明一个该辅助类的类静态成员变量，在该辅助类的构造函数中，初始化单例对象。 实现如下：

```
class Singleton
{
public:
    static Singleton* getInstance()
    {
        static Singleton instance;
        return &instance;
    }

protected:
    struct Object_Creator
    {
        Object_Creator()
        {
            Singleton::getInstance();
        }
    };
    static Object_Creator _object_creator;

    Singleton() {}
    ~Singleton() {}
};

Singleton::Object_Creator Singleton::_object_creator;
```

在前面的方案中：饿汉模式中，使用到了类静态成员变量，但是遇到了初始化顺序的问题； 懒汉模式中，使用到了静态局部变量，但是存在着线程安全等问题。 那么在这个终极方案中可以说综合了以上两种方案。即采用到了类静态成员变量，也采用到了静态局部变量。

注意到其中的辅助结构体 Object_Creator （可以称之为 proxy-class）所声明的类静态成员变量，初始化该静态成员变量时，其中的构造函数 调用了单例类的 getInstance 方法。这样就会调用到 Singleton::getInstance() 方法初始化单例对象，那么自然 Singleton 的构造函数也就执行了。

多线程抢占单例模式，为了不出现两个，不加锁怎么处理

同上

进程和线程的区别

线程切换方式，步骤

https://juejin.im/post/5b10e53b6fb9a01e5b10e9be

进程切换

保存进程A的状态**（寄存器和操作系统数据）；

1.更新PCB中的信息，对进程A的“运行态”做出相应更改；

2.将进程A的PCB放入相关状态的队列；

3.将进程B的PCB信息改为“运行态”，并执行进程B；

4.B执行完后，从队列中取出进程A的PCB，恢复进程A被切换时的上下文，继续执行A；

线程切换和进程切换的步骤也不同。进程的上下文切换分为两步：

1. **切换页目录以使用新的地址空间**；
2. **切换内核栈和硬件上下文**；

对于Linux来说，线程和进程的最大区别就在于地址空间。**对于线程切换，第1步是不需要做的，第2是进程和线程切换都要做的**。所以明显是进程切换代价大。线程上下文切换和进程上下文切换一个最主要的区别是 **线程的切换虚拟内存空间依然是相同的，但是进程切换是不同的**。这两种上下文切换的处理都是 **通过操作系统内核来完成的**。内核的这种切换过程伴随的 **最显著的性能损耗是将寄存器中的内容切换出**。

对于一个正在执行的进程包括 **程序计数器、寄存器、变量的当前值等** ，而这些数据都是 **保存在CPU的寄存器中的，且这些寄存器只能是正在使用CPU的进程才能享用，在进程切换时，首先得保存上一个进程的这些数据（便于下次获得CPU的使用权时从上次的中断处开始继续顺序执行，而不是返回到进程开始，否则每次进程重新获得CPU时所处理的任务都是上一次的重复，可能永远也到不了进程的结束出，因为一个进程几乎不可能执行完所有任务后才释放CPU），然后将本次获得CPU的进程的这些数据装入CPU的寄存器从上次断点处继续执行剩下的任务**。

拓展线程共享和独立的资源包括

共享

1.进程代码段

2.进程的公有数据(利用这些共享的数据，线程很容易的实现相互之间的通讯)

3.进程打开的文件描述符、信号的处理器、进程的当前目录和进程用户ID与进程组ID。

独立

1.线程ID

2.寄存器组的值

3.线程的堆栈

4.错误返回码

5.线程的信号屏蔽码

6.线程的优先级

进程和线程的区别

怎么一个客户端怎么链接多个client，高并发，epoll实现原理

快速排序的思想

三次招手四次挥手

编程题链表翻转

正常版，递归版

３个商人和３个随从过河，一个船，随从多余商人会吧商人杀掉

```
Node* era(Node *list) {
    if (list == NULL) return NULL;
    Node *p,*q,*c;
    p = list;
    q = p->next;
    while (p && p->next) {
        c = q->next;
        q->next = p;
        p = q;
        q = c;
    }
    list = NULL;
    return p;
}

Node *era2(Node *node) {
    if(node == NULL || node->next == NULL) return node;
    Node *node2 = node->next;
    node->next = NULL;
    Node *node3 = era2(node2);
    node2->next = node;
    return node3;
}
```

递归版，

３个商人和３个随从过河

2从

１从

２从

１从

２商

１商１从

２商

１从

２从

１从

２从