学习笔记

编写：xxx

# 计算机网络

## 基础

### 网络协议模型

Xx

### 子网掩码有什么用

Xx

### 对称加密与非对称加密区别

Xx

### dns为什么用udp

xx

## IP

### 地址分为几类

其中A类、B类、和C类这三类地址用于TCP/IP节点，其它两类D类和E类被用于特殊用途。

A、B、C三类IP地址的特征：当将IP地址写成二进制形式时，A类地址的第一位总是0，B类地址的前两位总是10，C类地址的前三位总是110。

### 解决ipv4地址不足技术

1. NAT

我们一般使用私网ip作为局域网内部的主机标识，使用公网ip作为互联网上通信的标识,在整个NAT的转换中，最关键的流程有以下几点

网络被分为私网和公网两个部分，NAT网关设置在私网到公网的路由出口位置，双向流量必须都要经过NAT网关

网络访问只能先由私网侧发起，公网无法主动访问私网主机；

NAT网关在两个访问方向上完成两次地址的转换或翻译，出方向做源信息替换，入方向做目的信息替换；

NAT网关的存在对通信双方是保持透明的；

NAT网关为了实现双向翻译的功能，需要维护一张关联表，把会话的信息保存下来。

## http

### 常见的HTTP状态码：

200 - 请求成功

301 - 资源（网页等）被永久转移到其它URL

404 - 请求的资源（网页等）不存在

500 - 内部服务器错误

### http网址访问过程

DNS查找过程：

1.浏览器会检查缓存中有没有这个域名对应的解析过的IP地址，如果缓存中有，这个解析过程就将结束。

2.如果用户的浏览器缓存中没有，浏览器会查找操作系统缓存（hosts文件）中是否有这个域名对应的DNS解析结果。

3.若还没有，此时会发送一个数据包给DNS服务器，DNS服务器找到后将解析所得IP地址返回给用户。

在应用层，浏览器会给web服务器发送一个HTTP请求；

请求头为：GET http://www.baidu.com/HTTP/1.1

在传输层，（上层的传输数据流分段）HTTP数据包会嵌入在TCP报文段中；

TCP报文段需要设置端口，接收方（百度）的HTTP端口默认是80，本机的端口是一个1024-65535之间的随机整数，这里假设为1025，这样TCP报文段由TCP首部（包含发送方和接收方的端口信息）+ HTTP数据包组成。

在网络层中，TCP报文段再嵌入IP数据包中；

IP数据包需要知道双方的IP地址，本机IP地址假定为192.168.1.5，接受方IP地址为220.181.111.147（百度），这样IP数据包由IP头部（IP地址信息）+TCP报文段组成。

在网络接口层，IP数据包嵌入到数据帧（以太网数据包）中在网络上传送；

数据帧中包含源MAC地址和目的MAC地址（通过ARP地址解析协议得到的）。这样数据帧由头部（MAC地址）+IP数据包组成。

数据包经过多个网关的转发到达百度服务器，请求对应端口的服务；

服务接收到发送过来的以太网数据包开始解析请求信息，从以太网数据包中提取IP数据包>TCP报文段>HTTP数据包

请求处理完成之后，服务器发回一个HTTP响应；

响应头为：HTTP/1.1200 OK

浏览器以同样的过程读取到HTTP响应的内容（HTTP响应数据包），然后浏览器对接收到的HTML页面进行解析，把网页显示出来呈现给用户。

客户端接收到返回数据，去掉对应头信息，形成也可以被浏览器认识的页面HTML字符串信息，交与浏览器翻译为对应页面规则信息展示为页面内容。

### http https区别

xx

### 如何解析http报头？

xx

## TCP

### tcp报头



### 三次握手四次挥手的状态字， 3次/ 4次，状态转化图

Xx

### 没有time\_wait会怎样

Xx

### 服务器端不调用accept会发生什么

listen()维护两个队列，一个是未完成三次握手的，一个是已完成三次握手的，accept()是从已完成三次握手的队列中取出一个而已(backlog指的就是已经完成握手了的队列的大小)

在被动状态的socket有两个队列，一个是正在进行三次握手的socket队列，一个是完成三次握手的socket队列。在握手完成后会从正在握手队列移到握手完成的队列，此时已经建立连接。accept就是从已经完成三次握手的socket队列里面取，不accept客户端能完成的连接就是此队列的大小

### 大量的 CLOSE\_WAIT 会怎么样？

xx

### TCP和UDP区别

Xx

### 怎么用udp模拟tcp

xx

### IP和32位整数呈双射关系

1. unsigned **int**  IPToValue(**const** string& strIP){
2. //inet\_pton
3. **int** a[4];
4. string IP = strIP;
5. string strTemp;
6. **size\_t** pos;
7. **size\_t** i = 3;
9. **do**{
10. pos = IP.find(".");
12. **if** (pos != string::npos){
13. strTemp = IP.substr(0, pos);
14. a[i] = atoi(strTemp.c\_str());
15. i--;
16. IP.erase(0, pos + 1);
17. }
18. **else**{
19. strTemp = IP;
20. a[i] = atoi(strTemp.c\_str());
21. **break**;
22. }
24. } **while** (1);
26. unsigned **int** nResult = (a[3] << 24) + (a[2] << 16) + (a[1] << 8) + a[0];
27. **return** nResult;
28. }
30. string ValueToIP(**const** **int**& nValue){
31. //inet\_ntop
32. **char** strTemp[20];
33. sprintf(strTemp, "%d.%d.%d.%d",
34. (nValue & 0xff000000) >> 24,
35. (nValue & 0x00ff0000) >> 16,
36. (nValue & 0x0000ff00) >> 8,
37. (nValue & 0x000000ff));
39. **return** string(strTemp);
40. }

### 服务端编程

1. 创建socket

2. 调用bind函数，将socket和地址（包括ip、port）绑定。

3. listen监听，将接收到的客户端连接放入队列

4. 调用accept函数，从队列获取请求，返回socket描述符，如果无请求，将会阻塞，直到获得链接

5. 发送或接收数据 send recv

6. 关闭accept返回的socket

### TCP快重传

超时重传是TCP协议保证数据可靠性的一个重要机制，其原理是在发送一个数据以后就开启一个计时器，在一定时间内如果没有得到发送数据报的ACK报文，那么就重新发送数据，直到发送成功为止。这是数据包丢失的情况下给出的一种修补机制。一般来说，重传发生在超时之后，但是如果发送端接收到3个以上的重复ACK，就应该意识到，数据丢了，需要重新传递。这个机制不需要等到重传定时器溢出，所以叫做快速重传，而快速重传以后，因为走的不是慢启动而是拥塞避免算法，所以这又叫做快速恢复算法。

快速重传和快速恢复算法一般同时使用。快速恢复算法是认为，你还有3个Duplicated Acks说明网络也不那么糟糕，所以没有必要像RTO超时那么强烈，并不需要重新回到慢启动进行，这样可能降低效率

### TCP的流量控制

所谓的流量控制就是让发送方的发送速率不要太快，让接收方来得及接受。利用滑动窗口机制可以很方便的在TCP连接上实现对发送方的流量控制。TCP的窗口单位是字节，不是报文段，发送方的发送窗口不能超过接收方给出的接收窗口的数值。

### 拥塞控制方法

拥塞控制的四种算法，即慢开始（Slow-start)、拥塞避免（Congestion Avoidance)、快重传（Fast Restrangsmit)和快回复（Fast Recovery）

<https://www.cnblogs.com/wxgblogs/p/5616829.html>

### TCP端口扫描方式

服务器上所开放的端口就是潜在的通信通道，也就是一个入侵通道。对目标计算机进行端口扫描，能得到许多有用的信息，进行端口扫描的方法很多，可以是手工进行扫描、也可以用端口扫描软件进行。扫描器通过选用远程TCP/IP不同的端口的服务，并记录目标给予的回答，通过这种方法可以搜集到很多关于目标主机的各种有用的信息，例如远程系统是否支持匿名登陆、是否存在可写的FTP目录、是否开放TELNET服务和HTTPD服务等

connect扫描；

SYN扫描；

FIN扫描；

### TCP可靠性怎么保证的？

Xx

### TCP的滑动窗口影响了什么性能？

xx

### TCP的粘包怎么解决？

Xx

### ping会使用到哪些协议

xx

## select/poll/epoll

I/O多路复用（multiplexing）的本质是通过一种机制（系统内核缓冲I/O数据），让单个进程可以监视多个文件描述符，一旦某个描述符就绪（一般是读就绪或写就绪），能够通知程序进行相应的读写操作,与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销

select、poll、epoll本质上也都是同步I/O

select:

1> 每次调用select，都需要把fd\_set集合从用户态拷贝到内核态，如果fd\_set集合很大时，那这个开销也很大

2> 同时每次调用select都需要在内核遍历传递进来的所有fd\_set，如果fd\_set集合很大时，那这个开销也很大

3> 为了减少数据拷贝带来的性能损坏，内核对被监控的fd\_set集合大小做了限制，并且这个是通过宏控制的，大小不可改变(限制为1024)

poll:

1> poll改变了文件描述符集合的描述方式，使用了pollfd结构而不是select的fd\_set结构，使得poll支持的文件描述符集合限制远大于select的1024

epoll:

1> 于事件驱动的I/O方式，相对于select来说，epoll没有描述符个数限制，使用一个文件描述符管理多个描述符，将用户关心的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次

2> 获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核IO事件异步唤醒而加入Ready队列的描述符集合就行了

https://www.jianshu.com/p/397449cadc9a

epoll中ET模式与LT模式的区别:

水平触发（LT）：默认工作模式，即当epoll\_wait检测到某描述符事件就绪并通知应用程序时，应用程序可以不立即处理该事件；下次调用epoll\_wait时，会再次通知此事件

边缘触发（ET）： 当epoll\_wait检测到某描述符事件就绪并通知应用程序时，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，不会再次通知此事件。（直到你做了某些操作导致该描述符变成未就绪状态了，也就是说边缘触发只在状态由未就绪变为就绪时只通知一次）

# 操作系统

## 用户态和内核态切换过程

Xx

## 什么叫软连接和硬链接，他们的区别是什么

Xx

## 死锁

1. 死锁是怎么产生的:
2. 死锁预防：
3. 死锁避免：
4. 死锁检测：

## 信号处理函数

Xx

## 系统调用和库函数有什么区别和联系

Xx

## Copy on write

xx

## 进程

### 进程地址空间

Xx

### fork

xx

### 进程同步

在Linux下，进程同步的解决方式主要有四种：

1. 信号量

2. 文件锁

3. 无锁CAS

4. 校验方式（CRC32校验）

<https://blog.csdn.net/okiwilldoit/article/details/78487507>

### 进程和线程的区别

1. 进程是运行中的程序，线程是进程的内部的一个执行序列
2. 进程是资源分配的单元，线程是执行行单元
3. 进程间切换代价大，线程间切换代价小（地址空间
4. 进程拥有资源多，线程拥有资源少
5. 多个线程共享进程的资源

### 线程间同步的方式

xx

### 多线程和多进程的优缺点

多线程的优点：

无需跨进程边界； 程序逻辑和控制方式简单； 所有线程可以直接共享内存和变量等； 线程方式消耗的总资源比进程方式好；

线程缺点：

线程之间的同步和加锁控制比较麻烦； 一个线程的崩溃可能影响到整个程序的稳定性； 到达一定的线程数程度后，即使再增加CPU也无法提高性能，例如Windows Server 2003，大约是1500个左右的线程数就快到极限了（线程堆栈设定为1M），如果设定线程堆栈为2M，还达不到1500个线程总数； 线程能够提高的总性能有限，而且线程多了之后，线程本身的调度也是一个麻烦事儿，需要消耗较多的CPU

多进程优点：

每个进程互相独立，不影响主程序的稳定性，子进程崩溃没关系； 通过增加CPU，就可以容易扩充性能； 可以尽量减少线程加锁/解锁的影响，极大提高性能，就算是线程运行的模块算法效率低也没关系； 每个子进程都有2GB地址空间和相关资源，总体能够达到的性能上限非常大

多线程缺点：逻辑控制复杂，需要和主程序交互； 需要跨进程边界，如果有大数据量传送，就不太好，适合小数据量传送、密集运算 多进程调度开销比较大；

最好是多进程和多线程结合，即根据实际的需要，每个CPU开启一个子进程，这个子进程开启多线程可以为若干同类型的数据进行处理。当然你也可以利用多线程+多CPU+轮询方式来解决问题

### 创建子进程，fork详解

Xx

### 调度算法

Xx

### 守护进程

Xx

### 孤儿进程

Xx

### Linux下有哪些原子操作

xx

### 锁

1. 自旋锁

xx

1. 互斥锁
2. 读写锁
3. 递归锁
4. 乐观锁
5. 悲观锁

## 内存管理

### 页面置换算法

Xx

### 分段分页机制

Xx

### 伙伴系统和slab

xx

### 虚拟内存

虚拟内存是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

### 逻辑地址到物理地址过程

Xx

### Linux内核地址空间

xx

### 缓存命中和不命中

Xx

### 内部碎片，外部碎片

xx

### Mmap

Xx

### malloc涉及的系统调用

xx

## 进程间通信

进程间通信（IPC，InterProcess Communication）是指在不同进程之间传播或交换信息。

IPC的方式通常有管道（包括无名管道和命名管道）、信号量、消息队列、共享内存、Socket、Streams等。其中 Socket和Streams支持不同主机上的两个进程IPC。

<https://blog.csdn.net/wh_sjc/article/details/70283843>

### Debug和Release

Debug通常称为调试版本，通过一系列编译选项的配合，编译的结果通常包含调试信息，而且不做任何优化，以为开发人员提供强大的应用程序调试能力。而Release通常称为发布版本，是为用户使用的，一般客户不允许在发布版本上进行调试。所以不保存调试信息，同时，它往往进行了各种优化，以期达到代码最小和速度最优。

### 管道

管道，通常指无名管道，是 UNIX 系统IPC最古老的形式。

1、特点：

它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端。

它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）。

它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

2、原型

#include <unistd.h>

Int pipe(int fd[2]); // 返回值：若成功返回0，失败返回-1

当一个管道建立时，它会创建两个文件描述符：fd[0]为读而打开，fd[1]为写而打开。若要数据流从父进程流向子进程，则关闭父进程的读端（fd[0]）与子进程的写端（fd[1]）；反之，则可以使数据流从子进程流向父进程。

### FIFO

FIFO，也称为命名管道，它是一种文件类型。

1、特点

FIFO可以在无关的进程之间交换数据，与无名管道不同。

FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

2、原型

#include <sys/stat.h>

// 返回值：成功返回0，出错返回-1

int mkfifo(const char \*pathname, mode\_t mode);

其中的 mode 参数与open函数中的 mode 相同。一旦创建了一个 FIFO，就可以用一般的文件I/O函数操作它。

当 open 一个FIFO时，是否设置非阻塞标志（O\_NONBLOCK）的区别：

若没有指定O\_NONBLOCK（默认），只读 open 要阻塞到某个其他进程为写而打开此 FIFO。类似的，只写 open 要阻塞到某个其他进程为读而打开它。

若指定了O\_NONBLOCK，则只读 open 立即返回。而只写 open 将出错返回 -1 如果没有进程已经为读而打开该 FIFO，其errno置ENXIO。

### 消息队列

消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列ID）来标识。

1、特点

消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。

消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。

消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。

2、原型

#include <sys/msg.h>

// 创建或打开消息队列：成功返回队列ID，失败返回-1

int msgget(key\_t key, int flag);

// 添加消息：成功返回0，失败返回-1

int msgsnd(int msqid, const void \*ptr, size\_t size, int flag);

// 读取消息：成功返回消息数据的长度，失败返回-1

int msgrcv(int msqid, void \*ptr, size\_t size, long type,int flag);

// 控制消息队列：成功返回0，失败返回-1

int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

在以下两种情况下，msgget将创建一个新的消息队列：

如果没有与键值key相对应的消息队列，并且flag中包含了IPC\_CREAT标志位。

key参数为IPC\_PRIVATE。

函数msgrcv在读取消息队列时，type参数有下面几种情况：

type == 0，返回队列中的第一个消息；

type > 0，返回队列中消息类型为 type 的第一个消息；

type < 0，返回队列中消息类型值小于或等于 type 绝对值的消息，如果有多个，则取类型值最小的消息。

可以看出，type值非 0 时用于以非先进先出次序读消息。也可以把 type 看做优先级的权值。

### 信号量

信号量（semaphore）与已经介绍过的 IPC 结构不同，它是一个计数器。信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。

1、特点

信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。

每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

支持信号量组。

2、原型

最简单的信号量是只能取 0 和 1 的变量，这也是信号量最常见的一种形式，叫做二值信号量（Binary Semaphore）。而可以取多个正整数的信号量被称为通用信号量。

Linux 下的信号量函数都是在通用的信号量数组上进行操作，而不是在一个单一的二值信号量上进行操作。

#include <sys/sem.h>

// 创建或获取一个信号量组：若成功返回信号量集ID，失败返回-1

int semget(key\_t key, int num\_sems, int sem\_flags);

// 对信号量组进行操作，改变信号量的值：成功返回0，失败返回-1

int semop(int semid, struct sembuf semoparray[], size\_t numops);

// 控制信号量的相关信息

int semctl(int semid, int sem\_num, int cmd, ...);

当semget创建新的信号量集合时，必须指定集合中信号量的个数（即num\_sems），通常为1； 如果是引用一个现有的集合，则将num\_sems指定为 0 。

### 内存共享

共享内存（Shared Memory），指两个或多个进程共享一个给定的存储区。

1、特点

共享内存是最快的一种 IPC，因为进程是直接对内存进行存取。

因为多个进程可以同时操作，所以需要进行同步。

信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问。

2、原型

#include <sys/shm.h>

// 创建或获取一个共享内存：成功返回共享内存ID，失败返回-1

int shmget(key\_t key, size\_t size, int flag);

// 连接共享内存到当前进程的地址空间：成功返回指向共享内存的指针，失败返回-1

void \*shmat(int shm\_id, const void \*addr, int flag);

// 断开与共享内存的连接：成功返回0，失败返回-1

int shmdt(void \*addr);

// 控制共享内存的相关信息：成功返回0，失败返回-1

int shmctl(int shm\_id, int cmd, struct shmid\_ds \*buf);

当用shmget函数创建一段共享内存时，必须指定其 size；而如果引用一个已存在的共享内存，则将 size 指定为0 。

当一段共享内存被创建以后，它并不能被任何进程访问。必须使用shmat函数连接该共享内存到当前进程的地址空间，连接成功后把共享内存区对象映射到调用进程的地址空间，随后可像本地空间一样访问。

shmdt函数是用来断开shmat建立的连接的。注意，这并不是从系统中删除该共享内存，只是当前进程不能再访问该共享内存而已。

shmctl函数可以对共享内存执行多种操作，根据参数 cmd 执行相应的操作。常用的是IPC\_RMID（从系统中删除该共享内存）。

### ftok

xx

## linux程序编译、链接、调试

### 基本过程

基本格式：gcc [options] file1 file2... //若不加入参数，则按默认参数依次执行编译、汇编和链接操作，生成的可执行文件名为 a.out

-E //只执行预处理操作, gcc -E circle.c -o circle.i

-S //只执行到编译操作完成，不进行汇编操作，生成的是汇编文件(.s 或 .asm),内容为汇编语言

-c //执行编译和汇编，但不进行链接，即只生成可重定位目标文件(.o),为二进制文件，不生成完整的可执行文件

-o filename //将操作后的内容输出到filename指定的文件中

-static //对于支持动态链接的系统，使用静态链接而不是动态链接进行链接操作

-g //编译时生成debug有关的程序信息(供gdb使用)

-O1、-O2 //规定编译器的优化等级，优化级数越高执行效率一般越好，但是优化会改变原有程序结构，使得其汇编不易理解

编译、汇编和链接链接过程如下：

驱动程序首先运行C预处理器cpp，它将C的源程序main . c翻译成一个ASCII 码的中间文件.i

驱动程序运行C编译器cc1，它将.i翻译成一个ASCII汇编语言文件. s

驱动程序运行汇编器as，它将main.s翻译成一个可重定位目标文件

驱动程序运行链接器程序ld，将. o以及一些必要的系统目标文件组合起来，创建一个可执行目标文件

要运行可执行文件，我们在Linux shell的命令行上输入它的名字:

linux> . /a.out

shell调用操作系统中一个叫做加载器loader的函数， 它将可执行文件中的代码和数据复制到内存，然后将控制转移到这个程序的开头。

### 静态链接

链接器读取一组可重定位目标文件，并把它们链接起来，形成一个输出的可执行文件。为了构造可执行文件， 链接器必须完成两个主要任务:

* 符号解析(symbol resolution)
* 重定位(relocation)

实际上，所有的编译系统都提供一种机制，将所有相关的目标模块打包成为一个单独的文件，称为静态库( static library) 。它可以用做链接器的输入。当链接器构造一个输出的可执行文件时，它只复制静态库里被应用程序引用的目标模块。

静态链接的缺点很明显，一是浪费空间，因为每个可执行程序中对所有需要的目标文件都要有一份副本，所以如果多个程序对同一个目标文件都有依赖，如多个程序中都调用了printf()函数，则这多个程序中都含有printf.o，所以同一个目标文件都在内存存在多个副本；另一方面就是更新比较困难，因为每当库函数的代码修改了，这个时候就需要重新进行编译链接形成可执行程序。但是静态链接的优点就是，在可执行程序中已经具备了所有执行程序所需要的任何东西，在执行的时候运行速度快。

### 动态链接

动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。

动态链接的优点显而易见，就是即使需要每个程序都依赖同一个库，但是该库不会像静态链接那样在内存中存在多分副本，而是这多个程序在执行时共享同一份副本；另一个优点是，更新也比较方便，更新时只需要替换原来的目标文件，而无需将所有的程序再重新链接一遍。当程序下一次运行时，新版本的目标文件会被自动加载到内存并且链接起来，程序就完成了升级的目标。但是动态链接也是有缺点的，因为把链接推迟到了程序运行时，所以每次执行程序都需要进行链接，所以性能会有一定损失。

### 常用的调试方法和工具

* print语句：基本调试，获得关键变量
* 查询/proc文件系统：获取有关文件系统支持，可用内存，CPU，运行程序的内核状态等信息
* strace / ltrace：最初的问题诊断，系统调用或库调用的相关问题，了解程序流程，strace的和ltrace是两个在Linux中用来追踪程序的执行细节的跟踪工具
* valgrind：应用程序内存空间的问题
* gdb：检查应用程序运行时的行为，分析应用程序崩溃

### coredump

xx

### cmake和makefile的区别

makefile + make可理解为类unix环境下的项目管理工具，但它太基础了，抽象程度不高，而且在windows下不太友好(针对visual studio用户)，于是就有了跨平台项目管理工具cmake

cmake是跨平台项目管理工具，它用更抽象的语法来组织项目。虽然，仍然是目标，依赖之类的东西，但更为抽象和友好，比如你可用math表示数学库，而不需要再具体指定到底是math.dll还是libmath.so，在windows下它会支持生成visual studio的工程，在linux下它会生成Makefile，甚至它还能生成eclipse工程文件。也就是说，从同一个抽象规则出发，它为各个编译器定制工程文件

### 简述cmake到可执行文件的过程

* 编写 CMake 配置文件 CMakeLists.txt
* 执行命令 cmake PATH 或者 ccmake PATH 生成 Makefile
* 使用 make 命令进行编译，然后链接生成可执行文件

## shell

* Linux下删除同一文件夹下所有满足条件的文件

find ./ -name "a.out" -exec rm -rf {} \;

* find
* grep
* ps
* netstat.

# 数据结构/算法

## 哈希冲突解决

hash的原理:

xx

一致性哈希:

xx

## 海量url去重

Xx

## Bitmap

xx

## 查找

### 链表中倒数第k个结点

xx

### 100 亿URL中判断某个URL是否存在

布隆过滤器（英语：Bloom Filter）。它实际上是一个很长的二进制矢量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率远远超过一般的算法，缺点是有一定的误识别率和删除困难。

失误率要保持在多少，数组长度，哈希函数的个数分别要设置多少就需要根据实际情况来选择了。

### 字符串匹配kmp

Xx

## 二叉树

### 怎么把一颗二叉树原地变成一个双向链表？

xx

### 红黑树（RB-tree）和AVL

红黑树是牺牲了严格的高度平衡的优越条件为代价红黑树能够以O(log2 n)的时间复杂度进行搜索、插入、删除操作。此外，由于它的设计，任何不平衡都会在三次旋转之内解决。当然，还有一些更好的，但实现起来更复杂的数据结构能够做到一步旋转之内达到平衡，但红黑树能够给我们一个比较“便宜”的解决方案。红黑树的算法时间复杂度和AVL相同，但统计性能比AVL树更高

### B树和B+树的区别，应用场景?

Xx

## 排序

### 时间复杂度为O(nlogn)的排序算法有哪些？

快速排序/归并排序/堆排序

### 冒泡排序

xx

### 堆排序

堆的结构可以分为大根堆和小根堆，是一个完全二叉树，而堆排序是根据堆的这种数据结构设计的一种排序，初始化建堆的时间复杂度为*O(N)*，排序重建堆的时间复杂度为*O(NlogN)*

1. 首先将无序元素构造成一个堆（新插入的数据与其父结点比较）
2. 将堆顶元素与末尾元素交换，将最大元素"沉"到数组末端重新调整结构，使其满足堆定义
3. 然后继续交换堆顶元素与当前末尾元素，反复执行调整+交换步骤，直到整个序列有序。

### 快排

1. **int** partition(vector<**int**> &v, **int** begin, **int** end) {
2. **int** k = end;
3. **while** (begin<end) {
4. **while** (begin<end && v[begin]<= v[k]) {
5. ++begin;
6. }
7. **while** (begin<end && v[end]>= v[k]) {
8. --end;
9. }
10. swap(v[begin], v[end]);
11. }
12. swap(v[begin], v[k]);
13. **return** begin;
14. }
15. **void** quick\_sort(vector<**int**> &v, **int** begin, **int** end) {
16. **if** (begin >= end) **return**;
17. **int** index = partition(v, begin, end);
18. quick\_sort(v, begin, index - 1);
19. quick\_sort(v, index + 1, end);
20. }

### 多路归并排序

### topk

## 手撕

1. 给定2个字符串str1和str2,从str1中删除所有str2中的字符
2. LRU cache
3. 洗牌算法

# C++

## 类

### struct和class区别

xx

### 为什么要字节对齐

Xx

### C++空类有哪些成员函数？

Xx

### 静态成员函数能调用非静态成员函数吗

因为类的静态成员和普通成员其实就一种区别，那就是静态成员本身没有this指针，所以静态成员属于类而不属于类对象。如果我们想在类的静态成员函数里面调用类的普通成员，只需要把类指针当做参数传入静态成员函数里面，静态成员函数可以使用这个指针调用类的普通成员

### 深拷贝浅拷贝的区别

xx

### 类如何防止被拷贝

1. **class** noncopyable
2. {
3. **protected**:
4. noncopyable() {}
5. ~noncopyable() {}
6. **private**:
7. noncopyable( **const** noncopyable& );
8. noncopyable& operator=( **const** noncopyable& );
9. };
10. **class** noncopyable
11. {
12. **protected**:
13. //constexpr noncopyable() = default;
14. // ~noncopyable() = default;
15. noncopyable( **const** noncopyable& ) = **delete**;
16. noncopyable& operator=( **const** noncopyable& ) = **delete**;
17. };

## 面向对象

### 面向对象主要有四大特性

1. 抽象

忽略一个主题中与当前目标无关的东西,专注的注意与当前目标有关的方面.(就是把现实世界中的某一类东西,提取出来,用程序代码表示,抽象出来的一般叫做类或者接口).抽象并不打算了解全部问题,而是选择其中的一部分,暂时不用部分细节.抽象包括两个方面,一个数据抽象,而是过程抽象.

数据抽象 -->表示世界中一类事物的特征,就是对象的属性.比如鸟有翅膀,羽毛等(类的属性)

过程抽象 -->表示世界中一类事物的行为,就是对象的行为.比如鸟会飞,会叫(类的方法)

1. 封装

封装就是把过程和数据包围起来,对数据的访问只能通过特定的界面.如私有变量,用set,get方法获取

1. 继承

一种联结类的层次模型,并且允许和鼓励类的重用,提供一种明确表达共性的方法.对象的一个新类可以从现有的类中派生,这个过程称为类继承.新类继承了原始类的特性,新类称为原始类的派生类(子类),原始类称为新类的基类(父类).派生类可以从它的父类哪里继承方法和实例变量,并且类可以修改或增加新的方法使之更适合特殊的需要.因此可以说,继承为了重用父类代码,同时为实现多态性作准备.

1. 多态

多态是指允许不同类的对象对同一消息做出响应.多态性包括参数化多态性和包含多态性.多态性语言具有灵活/抽象/行为共享/代码共享的优势,很好的解决了应用程序函数同名问题.总的来说,方法的重写,重载与动态链接构成多态性.java引入多态的概念原因之一就是弥补类的单继承带来的功能不足.

动态链接 -->对于父类中定义的方法,如果子类中重写了该方法,那么父类类型的引用将调用子类中的这个方法,这就是动态链接.

### C语言实现C++的继承

1. **typedef** **void**(\*FUN)();//重定义一个函数指针类型
3. //父类
4. **struct** Base
5. {
6. FUN \_f;
7. };
9. //子类
10. **struct** Derived
11. {
12. Base \_b;//在子类中定义一个基类的对象即可实现对父类的继承
13. };

16. **void** FunB()
17. {
18. printf("%s\n", "Base::fun()");
19. }
20. **void** FunD()
21. {
22. printf("%s\n", "Derived::fun()");
23. }
25. **void** Test2()
26. {
27. Base b;//父类对象
28. Derived d;//子类对象
30. b.\_f = FunB;//父类对象调用父类同名函数
31. d.\_b.\_f = FunD;//子类调用子类的同名函数
33. Base \*pb = &b;//父类指针指向父类对象
34. pb->\_f();
36. pb = (Base \*)&d;//让父类指针指向子类的对象,由于类型不匹配所以要进行强转
37. pb->\_f();
39. }

### 虚析构作用

总的来说虚析构函数是为了避免内存泄露，而且是当子类中会有指针成员变量时才会使用得到的。也就说虚析构函数使得在删除指向子类对象的基类指针时可以调用子类的析构函数达到释放子类中堆内存的目的，而防止内存泄露的

1. 如果父类的析构函数不加virtual关键字

当父类的析构函数不声明成虚析构函数的时候，当子类继承父类，父类的指针指向子类时，delete掉父类的指针，只调动父类的析构函数，而不调动子类的析构函数。

1. 如果父类的析构函数加virtual关键字

当父类的析构函数声明成虚析构函数的时候，当子类继承父类，父类的指针指向子类时，delete掉父类的指针，先调动子类的析构函数，再调动父类的析构函数。

### 如何实现多态的

xx

### 虚函数实现机制

xx

### 虚函数与纯虚函数的区别

virtual void funtion() {函数体}

virtual void funtion()=0;

虚函数必须定义自己的实现方法，在c++中是实现多态的机制，纯虚函数（所在的类为抽象类不能生成对象）不能定义自己的实现方法，是用来规范派生类的行为的，即接口

### 重载和重写的区别

* 重载

在同一个作用域内；函数名相同，参数列表不同（参数个数不同，或者参数类型不同，或者参数个数和参数类型都不同），返回值类型可相同也可不同，仅返回值不同不能重载；这种情况叫做C++的重载

C++函数重载达到的效果：调用函数名相同的函数，会根据实参的类型和实参顺序以及实参个数选择相应的函数；

C++函数重载是一种静态多态（又叫做静态联编，静态绑定，静态决议）

* 覆盖（又叫重写）

覆盖（重写）的前提条件：父类函数为虚函数；

覆盖（重写）的概念：当在子类中定义了一个与父类完全相同的虚函数时，则称子类的这个函数重写（也称覆盖）了父类的这个虚函数。

* 隐藏

如果在父类和子类中有相同名字的成员；那么在子类中。会将父类的成员隐藏；隐藏以后的直接效果就是：无论在子类的内部或者外部（通过子类成员）访问该成员；全都是访问子类的同名成员； 如果在子类内部或者外部（通过子类成员）访问同名的成员函数，则需要根据函数调用的规则来调用子类的同名成员函数；

### 为什么鼓励使用组合不使用继承

xx

### 右值引用

Xx

### 重写的函数中包含有默认参数

默认参数的值只和静态类型有关，是静态绑定的，虽然虚函数的是动态绑定的，但默认参数是静态绑定的。只有动态绑定的东西才应该被重写

### 访问权限说明符







## C++11新特性有哪些？

Xx

## 动态内存

### new和malloc

* 属性

new/delete是C++关键字，需要编译器支持。malloc/free是库函数，需要头文件支持c。

* 参数

使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算。而malloc则需要显式地指出所需内存的尺寸。

* 返回类型

new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，故new是符合类型安全性的操作符。而malloc内存分配成功则是返回void \* ，需要通过强制类型转换将void\*指针转换成我们需要的类型。

* 分配失败

new内存分配失败时，会抛出bad\_alloc异常。malloc分配内存失败时返回NULL。

* 自定义类型

new会先调用operator new函数，申请足够的内存（通常底层使用malloc实现）。然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete先调用析构函数，然后调用operator delete函数释放内存（通常底层使用free实现）。malloc/free是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。

* 重载

C++允许重载new/delete操作符，特别的，布局new的就不需要为对象分配内存，而是指定了一个地址作为内存起始区域，new在这段内存上为对象调用构造函数完成初始化工作，并返回此地址。而malloc不允许重载。

* 内存区域

new操作符从自由存储区（free store）上为对象动态分配内存空间，而malloc函数从堆上动态分配内存。自由存储区是C++基于new操作符的一个抽象概念，凡是通过new操作符进行内存申请，该内存即为自由存储区。而堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配，C语言使用malloc从堆上分配内存，使用free释放已分配的对应内存。

在C++中，内存区分为5个区，分别是堆、栈、自由存储区、全局/静态存储区、常量存储区；在C中，C内存区分为堆、栈、全局/静态存储区、常量存储区；

### new运算符的原理

xx

### 设计一个内存池

xx

### C++内存模型

C++内存分为5个区域：

* 堆 heap

由new分配的内存块，其释放编译器不去管，由我们程序自己控制（一个new对应一个delete）。如果程序员没有释放掉，在程序结束时OS会自动回收。涉及的问题：“缓冲区溢出”、“内存泄露”

* 栈 stack

是那些编译器在需要时分配，在不需要时自动清除的存储区。存放局部变量、函数参数。

存放在栈中的数据只在当前函数及下一层函数中有效，一旦函数返回了，这些数据也就自动释放了。

* 全局/静态存储区 （.bss段和.data段）

全局和静态变量被分配到同一块内存中。在C语言中，未初始化的放在.bss段中，初始化的放在.data段中；在C++里则不区分了。

* 常量存储区 （.rodata段）

存放常量，不允许修改（通过非正当手段也可以修改）

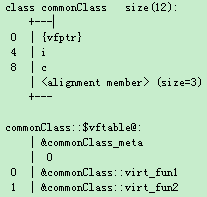
* 代码区 （.text段）

C++内存分布：

* 带虚函数的类

1. **class** commonClass{
2. **int** i;
3. **char** c;
4. **virtual** **void** virt\_fun1(){}
5. **virtual** **void** virt\_fun2(){}
6. **void** comm\_fun1(){}
7. **void** comm\_fun2(){}
8. };

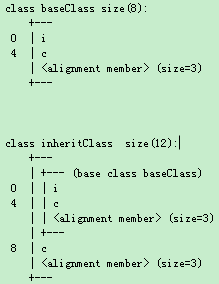
该类占用12个字节，如下图所示：



四字节对其2个变量共占用8个字节，由于存在虚函数，所以在类开始位置插入了一个虚函数指针，该指针占用4个字节，共计12字节。类的非虚函数其实不占用类对象的内存（函数编译后形成二进制文件放在内存中的代码段区）

* 简单继承

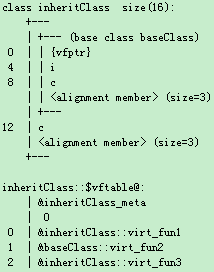
1. **class** baseClass{
2. **int** i;
3. **char** c;
4. };
5. **class** inheritClass : **public** baseClass{
6. **char** c;
7. };



* 基类带有虚函数

1. **class** baseClass{
2. **private**:
3. **int** i;
4. **char** c;
5. **public**:
6. **virtual** **void** virt\_fun1(){ cout << "baseClass virt\_fun1" << endl; }
7. **virtual** **void** virt\_fun2(){ cout << "baseClass virt\_fun2" << endl; }
8. **void** comm\_fun1(){ cout << "baseClass comm\_fun1" << endl; }
9. **void** comm\_fun2(){ cout << "baseClass comm\_fun2" << endl; }
10. };
12. **class** inheritClass : **public** baseClass{
13. **private**:
14. **char** c;
15. **public**:
16. **virtual** **void** virt\_fun1(){ cout << "inheritClass virt\_fun1" << endl; }
17. **virtual** **void** virt\_fun3(){ cout << "inheritClass virt\_fun2" << endl; }
18. **void** comm\_fun1(){ cout << "inheritClass comm\_fun1" << endl; }
19. **void** comm\_fun3(){ cout << "inheritClass comm\_fun3" << endl; }
20. };

基类和派生类使用同一个虚函数表。虚函数表的排列顺序遵循先基类后派生类，由于基类和派生类存在同名的虚函数（virt\_fun1），所以派生类的虚函数覆盖了基类的同名虚函数从而排到了最前面



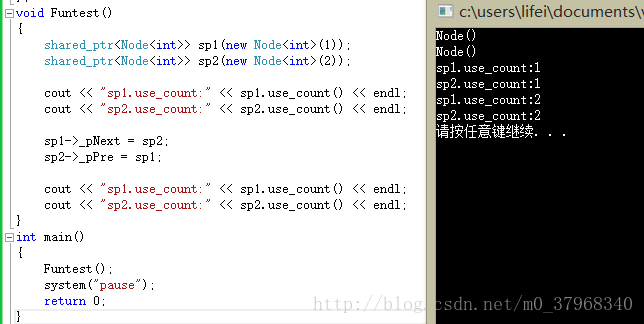
* 虚继承/多重继承

<https://www.jianshu.com/p/cc797aa4ace1>

### 智能指针的原理、使用、实现

<https://www.cnblogs.com/wxquare/p/4759020.html>

### 智能指针出现循环引用



我们可以看出，并没有调用析构函数，也就是没有对空间进行释放。

从上面shared\_ptr的实现中我们知道了只有当引用计数减减之后等于0，析构时才会释放对象，而上述情况造成了一个僵局，那就是析构对象时先析构sp2,可是由于sp2的空间sp1还在使用中，所以sp2.use\_count减减之后为1，不释放，sp1也是相同的道理，由于sp1的空间sp2还在使用中，所以sp1.use\_count减减之后为1，也不释放。sp1等着sp2先释放，sp2等着sp1先释放，二者互不相让，导致最终都没能释放，内存泄漏

解决方案：使用弱指针weak\_ptr(不增加引用计数)

1. weak\_ptr<Node<T>> \_pPre;
2. weak\_ptr<Node<T>> \_pNext;

## C++中内存泄漏问题

1. new出来的内存没有通过delete合理的释放掉
2. new创建了一组对象数组，内存回收的时候却只调用了delete而非delete []来处理，导致只有对象数组的第一个对象的析构函数得到执行并回收了内存占用，数组的其他对象所占内存得不到回收，导致内存泄露
3. 没有将基类的析构函数定义为虚函数

解决内存泄漏最有效的办法就是使用智能指针（Smart Pointer）。使用智能指针就不用担心这个问题了，因为智能指针可以自动删除分配的内存

## 成员函数的前后const

在类中将成员函数修饰为const（函数后）表明在该函数体内，不能修改对象的数据成员而且不能调用非const函数

const修饰函数返回值，也是用const来修饰返回的指针或引用，保护指针指向的内容或引用的内容不被修改，也常用于运算符重载。归根究底就是使得函数调用表达式不能作为左值

## C++有几种转换方法

xx

## 指针和引用的区别

1. 引用不能为空，即不存在对空对象的引用，指针可以为空，指向空对象。
2. 引用必须初始化，指定对哪个对象的引用，指针不需要。
3. 引用初始化后不能改变，指针可以改变所指对象的值。
4. 引用访问对象是直接访问，指针访问对象是间接访问。
5. 不能定义引用数组（声明引用数组没有办法分配空间，因为根本就没有空间可以分配给引用）

## 静态类型获取与动态类型获取

typeid关键字：

用于获取类型信息，typeid返回一个type\_info类对象，当参数是类型时，返回静态类型信息，当参数是变量时：不存在虚函数表，返回静态类型信息；存在虚函数表，返回动态类型信息

dynamic\_cast：

用于类继承层次间的指针或引用转换。主要还是用于执行“安全的向下转型（safe downcasting）”，也即是基类对象的指针或引用转换为同一继承层次的其他指针或引用。至于“向上转型”（即派生类指针或引用类型转换为其基类类型），本身就是安全的，尽管可以使用dynamic\_cast进行转换，但这是没必要的， 普通的转换已经可以达到目的，毕竟使用dynamic\_cast是需要开销的。

## RAII

“资源获取即初始化”，也就是说在构造函数中申请分配资源，在析构函数中释放资源。因为C++的语言机制保证了，当一个对象创建的时候，自动调用构造函数，当对象超出作用域的时候会自动调用析构函数。所以，在RAII的指导下，我们应该使用类来管理资源，将资源和对象的生命周期绑定。

智能指针（std::shared\_ptr和std::unique\_ptr）即RAII最具代表的实现，使用智能指针，可以实现自动的内存管理，再也不需要担心忘记delete造成的内存泄漏。

## vector的结构？vector拷贝时发生什么

xx

## 什么函数不能声明为virtual

1. 普通的函数

不能被覆盖，在编译的时候就绑定函数。

1. 构造函数

构造函数无法是虚函数，因为调用虚函数需要虚函数表指针，而在执行构造函数之前是没有虚函数表指针的。

1. 静态成员函数

静态成员函数不可以是虚函数。静态函数是属于类的，不属于对象本身，自然无法有自己的虚函数表指针。

1. 内联函数

内联函数目的是在代码中直接展开（编译期），而虚函数是为了继承后能动态绑定执行自己的动作（动态绑定），因此本质是矛盾的，因此即使内联函数声明为虚函数，编译器遇到这种情况是不会进行inline展开的，而是当作普通函数来处理。因此声明了虚函数不能实现内敛的，即内敛函数可以声明为虚函数，但是毫无了内联的意义

1. 友员函数

C++不支持友元函数继承，友元函数不属于类的成员函数，不能被继承。

## Lambda

xx

## STL中的容器

### 常见容器

1. 顺序容器：

vector：可变大小数组；

deque：双端队列；

list：双向链表；

forward\_list：单向链表；

array：固定大小数组；

string：与vector相似的容器，但专门用于保存字符。

1. 关联容器：

按关键字有序保存元素：（底层实现为红黑树）

map：关联数组；保存关键字-值对；

set：关键字即值，即只保存关键字的容器；

multimap：关键字可重复的map；

multiset：关键字可重复的set；

1. 无序集合：

unordered\_map：用哈希函数组织的map；

unordered\_set：用哈希函数组织的set；

unordered\_multimap：哈希组织的map；关键字可以重复出现；

unordered\_multiset：哈希组织的set；关键字可以重复出现。

1. 其他项：

stack、queue、valarray、bitset

### vector的底层实现

扩容原理、size、capacity、resize、reserve四个函数

Xx

### vector与list的区别

xx

### allocator

xxx

### priority\_queue的底层实现

缺省情况下priority\_queue优先级队列是利用一个max-heap最大堆完成，后者是一个以vector表现的完全二叉树。

如果要用到小顶堆，则一般要把模板的三个参数都带进去。 STL里面定义了一个仿函数 greater<>，对于基本类型可以用这个仿函数声明小顶堆

## 关键字

1. extern “C”

作为面向对象的语言，C++为了支持函数重载，函数在被C++编译后在符号库中的名字与C语言的不同。假如某个函数的原型为void foo(int x, int y);该函数被C编译器编译后在符号库中的名字为\_foo，而C++编译器则会产生\_foo\_int\_int之类的名字。\_foo\_int\_int这样的名字是包含了函数名以及形参，C++就是靠这种机制来实现函数重载的。

  被extern “C”修饰的函数或者变量是按照C语言方式编译和链接的，所以可以用一句话来概括extern “C”的真实目的：实现C++与C的混合编程。

1. sizeof

首先空类的大小1，这是因为在实例化的过程中每个实例在内存中都得有一个独一无二的地址，为了达到这个目的，编译器会给空类隐含加一个字节

对于有虚函数的空类，该函数在基本的虚函数表中占用一项，因此szieof(nuullClassVirtual)大小为4

对于类中包含static/ static const成员的类，该静态成员不影响类的大小，该成员只有一个实例存在，无论类是否被实例化

1. volatile

xx

1. static

全局变量与static变量：

者在存储方式上并无不同。这两者的区别在于非静态全局变量的作用域是整个源程序，当一个源程序由多个原文件组成时，非静态的全局变量在各个源文件中都是有效的。而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其它源文件中不能使用它

static函数与普通函数：

作用域不同，仅在本文件。只在当前源文件中使用的函数应该说明为内部函数(static修饰的函数)，内部函数应该在当前源文件中说明和定义。对于可在当前源文件以外使用的函数，应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件

xx

1. inline

inline只适合涵数体内代码简单的涵数使用；

inline函数仅仅是一个对编译器的建议；

定义在类中的成员函数缺省都是内联的；

inline函数（即内联函数）对编译器而言必须是可见的，以便能够在调用点展开该函数，与非inline函数不同的是，inline函数必须在调用该函数的每个文件中定义

1. const 和 #define

xx

## c++里面的同步和互斥怎么实现的

xx

## c++怎么实现一个函数先于main函数运行

定义在main( )函数之前的全局对象、静态对象的构造函数在main( )函数之前执行

## 拷贝构造函数必须为引用传递

当 一个对象需要以值方式传递时，编译器会生成代码调用它的拷贝构造函数以生成一个复本。如果类A的拷贝构造函数是以值方式传递一个类A对象作为参数的话，当 需要调用类A的拷贝构造函数时，需要以值方式传进一个A的对象作为实参； 而以值方式传递需要调用类A的拷贝构造函数；结果就是调用类A的拷贝构造函数导 致又一次调用类A的拷贝构造函数，这就是一个无限递归

## 模板机制，类模板和函数模板区别

Xx

## 变量的初始化顺序

1. 基类的静态变量或全局变量
2. 派生类的静态变量或全局变量
3. 基类的成员变量
4. 派生类的成员变量
5. 注意：
6. 成员变量在使用初始化列表初始化时，与构造函数中初始化成员列表的顺序无关，只与定义成员变量的顺序有关。
7. 如果不使用初始化列表初始化，在构造函数内初始化时，此时与成员变量在构造函数中的位置有关。
8. 类中const成员常量必须在构造函数初始化列表中初始化。
9. 类中static成员变量，必须在类外初始化。

## 标准库

1. shrink\_to\_fit();

有一种方法来把它从曾经最大的容量减少到它现在需要的容量。这样减少容量的方法常常被称为“收缩到合适（shrink\_to\_fit）”。该方法只需一条语句：vector<int>(ivec).swap(ivec);

表达式vector<int>(ivec)建立一个临时vector，它是ivec的一份拷贝：vector的拷贝构造函数做了这个工作。但是，vector的拷贝构造函数只分配拷贝的元素需要的内存，所以这个临时vector没有多余的容量。然后我们让临时vector和ivec交换数据，这时我们完成了，ivec只有临时变量的修整过的容量，而这个临时变量则持有了曾经在ivec中的没用到的过剩容量。在这里（这个语句结尾），临时vector被销毁，因此释放了以前ivec使用的内存，收缩到合适

## 实现常见函数

### memcpy的实现

memcpy和memmove他们的作用是一样的，唯一的区别是，当内存发生局部重叠的时候，memmove保证拷贝的结果是正确的，memcpy不保证拷贝的结果的正确，以下为考虑内存重叠后的实现。

1. **void** \*memcpy(**void** \*dst, **const** **void** \*src, **size\_t** len) {
2. **if** (NULL == dst || NULL == src) {
3. **return** NULL;
4. }
5. **void** \*ret = dst;
6. **if** (dst <= src || (**char** \*)dst >= (**char** \*)src + len){ //没有内存重叠，从低地址开始复制
7. **while** (len--) {
8. \*(**char** \*)dst = \*(**char** \*)src;
9. dst = (**char** \*)dst + 1;
10. src = (**char** \*)src + 1;
11. }
12. }
13. **else** {  //有内存重叠，从高地址开始复制
14. src = (**char** \*)src + len - 1;
15. dst = (**char** \*)dst + len - 1;
16. **while** (len--) {
17. \*(**char** \*)dst = \*(**char** \*)src;
18. dst = (**char** \*)dst - 1;
19. src = (**char** \*)src - 1;
20. }
21. }
22. **return** ret;
23. }

### strcpy的实现

1. **char** \*strcpy(**char** \*strDest, **const** **char** \*strSrc){
2. assert((strDest != NULL) && (strSrc != NULL));
3. **char** \*ret = strDest;
4. **char** \*d, \*s;
5. **int** size = 0;
6. d = (**char** \*)strDest;
7. s = (**char** \*)strSrc;
9. size = strlen(strSrc) + 1;
10. **if** (strDest >= strSrc && strDest <= strSrc + size) {
11. d += size - 1;
12. s += size - 1;
13. **while** (size--) {
14. \*d-- = \*s--;
15. }
16. }
17. **else** {
18. **while** (size--)
19. \*strDest++ = \*strSrc++;
20. }
21. **return** ret;
22. }

### 手写strncpy

xx

# 智力

## 4刀把一个圆柱形蛋糕切16块

第一刀，切成2块；第2到，把2块排列放置，变成4块；第三刀，把4块排列放置，变成8块；第4刀，把8块排列放置，变成16块

## 有2个盒子，要把50个红球和50个蓝球全放完，从2个盒子里抽出1个球，怎样才能使抽到红球的可能性大

把一个红球单独放到一个盒子，另外49个红球和50个篮球放在一个盒子，这样摸到红球的概率就是：单独放红球的盒子的概率是0.5，另外放49个红球和50个篮球的盒子的概率是0.5X[49/(49+50)]，两个盒子摸到红球合起来的概率就是148/198，约为0.75

# 其他

1. git pull和git fetch的区别

git pull : 首先比对本地的FETCH\_HEAD记录与远程仓库的版本号，然后git fetch 获得当前指向的远程分支的后续版本的数据，然后再利用git merge将其与本地的当前分支合并。所以可以认为git pull是git fetch和git merge两个步骤的结合

1. 单例模式
2. git的 pull 、 push 和 merge的联系和区别