计算机知识笔记

/\*\*\*\*\*\*\*\*\*\*\*基础知识\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、在函数内部修改形参并不能真正的改变传入形参的值

2、函数内的局部自动变量，在函数返回后，内存已经被释放。

3、申请内存两部曲：申请并判断是否申请成功；用完释放并置为NULL（否则可能变成一个“野”指针）

4、浮点型变量并不精确，所以不可将float变量用“==”或“！=”与数字比较，应该设法转化成“>=”或“<=”形式。

所以判断一个浮点数是否为0，应该

const float EPSINON = 0.00001;

if ((x >= - EPSINON) && (x <= EPSINON)

5、#define MIN（A，B） （(A)>(B)?(B):(A)）不能加分号哟 一定要注意这一大堆括号，简单来说就是每个参数都要加括号，整个宏也要加括号

6、标准头文件

#ifndef sth 这里是为了防止被重复引用

#define sth

#ifdef \_\_cplusplus

extern "C" 这里是因为cpp支持函数重载，而c不支持，这是因为cpp编译器会产生像\_foo\_int\_int这样的名字，而c只会编译为\_foo，为了实现二者的混合编程，所以用extern“C”来解决名字匹配问题，使得编译器都按照c语言规则编译

{

#endif

.....

#ifdef \_\_cplusplus

}

#endif

#endif /\*sth\*/

7、

static关键字的作用

（1）函数体内的static变量作用范围为整个函数体，其内存只分配一次，在下次调用时仍维持上次的值

（2）模块内的static全局变量可以被模块内所有函数访问，但不能被模块外的其它函数访问

（3）类中的static成员变量属于整个类所有，对类的所有对象只有一份拷贝

（4）类中的static成员函数属于整个类所有，只能访问类的static成员变量，不接受this指针

const关键字的作用

（1）防止变量被改变，定义时需要初始化

（2）修饰函数形参，表示输入参数，在函数内部不可改变

（3）修饰类的成员函数，表示一个常函数，不能修改类的成员变量

（4）修饰成员函数的返回值

8、采用Little-endian模式的CPU对操作数的存放方式是从低字节到高字节，而Big-endian模式对操作数的存放方式是从高字节到低字节。例如，16bit宽的数0x1234在Little-endian模式CPU内存中的存放方式（假设从地址0x4000开始存放）为：

内存地址0x4000 0x4001

存放内容0x34 0x12

而在Big-endian模式CPU内存中的存放方式则为：

内存地址0x4000 0x4001

存放内容0x12 0x34

所以可用如下函数检查大小端

int Check(){

int a = 0x12345678;

if(\*((char\*)&a) == 0x12){

return 0; 大端

}

else{

return 1; 小端

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*计算机结构与操作系统（是系统软件而不是应用软件）\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、操作系统的主要功能是管理计算机上的所有硬件和软件资源（硬件资源主要有中央处理器、主存储器、辅助存储器和各种输入输出设备。软件资源有编译程序、编辑程序等各种程序以及有关数据。），主要包括5个方面的功能：进程与处理机管理，作业管理，存储管理，设备管理，文件管理。

2、实时系统要求在规定的时间内对外界的请求必须给予及时相应

3、系统调用的目的是请求系统服务，使得用户可以使用操作系统提供的有关设备管理、输入/输入系统、文件系统和进程控制、通信以及存储管理、资源等方面的功能

4、分布式系统

一组独立的计算机展现给用户的是一个统一的整体，就好像是一个系统似的。系统拥有多种通用的物理和逻辑资源，可以动态的分配任务，分散的物理和逻辑资源通过计算机网络实现信息交换。系统中存在一个以全局的方式管理计算机资源的分布式操作系统。通常，对用户来说，分布式系统只有一个模型或范型。在操作系统之上有一层软件中间件（middleware）负责实现这个模型。一个著名的分布式系统的例子是万维网（World Wide Web），在万维网中，所有的一切看起来就好像是一个文档（Web页面）一样。

5、网络系统

在计算机网络中，这种统一性、模型以及其中的软件都不存在。用户看到的是实际的机器，计算机网络并没有使这些机器看起来是统一的。如果这些机器有不同的硬件或者不同的操作系统，那么，这些差异对于用户来说都是完全可见的。如果一个用户希望在一台远程机器上运行一个程序，那么，他必须登陆到远程机器上，然后在那台机器上运行该程序。

分布式系统和计算机网络系统的共同点是：多数分布式系统是建立在计算机网络之上的，所以分布式系统与计算机网络在物理结构上是基本相同的。

区别在于：共享性的差别。网络操作系统要求网络用户在使用网络资源时首先必须了解网络资源，网络用户必须知道网络中各个计算机的功能与配置、软件资源、网络文件结构等情况，在网络中如果用户要读一个共享文件时，用户必须知道这个文件放在哪一台计算机的哪一个目录下；分布式操作系统是以全局方式管理系统资源的，它可以为用户任意调度网络资源，并且调度过程是“透明”的。当用户提交一个作业时，分布式操作系统能够根据需要在系统中选择最合适的处理器，将用户的作业提交到该处理程序，在处理器完成作业后，将结果传给用户。在这个过程中，用户并不会意识到有多个处理器的存在，这个系统就像是一个处理器一样。

6、独享设备：在一个用户作业未完成或退出之前，此设备不能分配给其他作业用。所有字符设备都是独享设备。如输入机、磁带机、打印机等。需要装驱动。

共享设备：多个用户作业或多个进程可以“同时”从这些设备上存取信息。软硬盘、光盘等块设备都是共享设备。无需驱动。

虚拟设备：通过软件技术将独享设备改造成共享设备。例如：通过SPOOLing技术将一台打印机虚拟成多台打印机。实质还是独享设备，需要驱动。

7、操作系统的发展与分类

手工操作阶段，其实就是没有操作系统的阶段，所有工作都需要人工干预。随着计算机硬件的发展，人机矛盾（速度和资源利用）越来越大，必须寻求新的解决办法。

手工操作阶段有两个突出的缺点：

用户独占全机。不会出现因资源已被其他用户占用而等待的现象，但资源利用率低。

CPU等待手工操作，CPU的利用不充分。

批处理阶段，操作系统开始出现，分为单道批处理系统和多道批处理系统。为了解决人机矛盾及CPU和I/O设备之间速度不匹配的矛盾而出现的。

单道批处理系统：系统对作业的处理是成批进行的，但内存中始终保持一道作业。

特征有：自动性。在顺利的情况下，在磁带上的一批作业能自动地逐个依次运行，而无需人工干预；顺序性。磁带上的各道作业是顺序地进入内存，各道作业的完成顺序与它们进入内存的顺序，在正常情况下应完全相同，亦即先调入内存的作业先完成；单道性。内存中仅有一道程序运行，即监督程序每次从磁带上只调入一道程序进入内存运行，当该程序完成或发生异常情况时，才换入其后继程序进入内存运行。

问题：在于每次主机内存中仅存放一道作业，每当它运行期间（注意这里是“运行时”，并不是“完成后”）发出输入/输出请求后，高速的CPU便处于等待低速的I/O完成状态。为了进一步提高资源的利用率和系统的吞吐量，引入了多道程序技术。

多道批处理系统：允许多个程序同时进入内存并运行。即同时把多个程序放入内存，并允许它们交替在CPU中运行，它们共享系统中的各种硬、软件资源。当一道程序因I/O请求而暂停运行时，CPU便立即转去运行另一道程序。它没有用某些机制提高某一技术方面的瓶颈问题，而是让系统的各个组成部分都尽量去“忙”，花费很少时间去切换任务，达到了系统各部件之间的并行工作，使其整体在单位时间内的效率翻倍。系统由一个程序转而运行另一个程序时需要使用中断机构中断正在运行的程序。

特征有：多道：计算机内存中同时存放多道相互独立的程序，共享系统资源（共享性，多道程序设计按照一定的作业调度算法将作业队列中的作业调入内存，使他们共享CPU和各种资源。因此也需要更大的内存以容纳超过一道的作业。）；宏观上并行：同时进入系统的多道程序都处于运行过程中，即它们先后开始了各自的运行，但都未运行完毕（并发性）；微观上串行：内存中的多道程序轮流占有CPU，交替执行。

缺点是用户响应的时间较长。不提供人机交互能力，用户既不能了解自己程序的运行情况，也不能控制计算机。

问题有：如何分配处理器；多道程序的内存分配问题；I/O设备如何分配；如何组织和存放大量的程序和数据，以便于用户使用和保证其安全性与一致性。

核心技术：作业调度：作业的现场保存和恢复；资源共享：资源的竞争和同步——互斥机制；内存使用：提高内存使用效率

分时操作系统：在操作系统中釆用分时技术就形成了分时系统。所谓分时技术就是把处理器的运行时间分成很短的时间片，按时间片轮流把处理器分配给各联机作业使用。若某个作业在分配给它的时间片内不能完成其计算，则该作业暂时停止运行，把处理器让给其他作业使用，等待下一轮再继续运行。由于计算机速度很快，作业运行轮转得很快，给每个用户的感觉好像是自己独占一台计算机。分时操作系统是多个用户通过终端同时共享一台主机，这些终端连接在主机上，用户可以同时与主机进行交互操作而互不干扰。所以，实现分时系统最关键的问题是如何使用户能与自己的作业进行交互，即当用户在自己的终端上键入命令时，系统应能及时接收并及时处理该命令，再将结果返回用户，所以也可以认为具有实时性。分时系统也是支持多道程序设计的系统，但它不同于多道批处理系统。多道批处理是实现作业自动控制而无需人工干预的系统，而分时系统是实现人机交互的系统。

需要用到的技术有：多道程序设计技术；作业说明书；终端命令解释程序；中断处理

特征有：同时性。也称多路性，指允许多个终端用户同时使用一台计算机，即一台计算机与若干台终端相连接，终端上的这些用户可以同时或基本同时使用计算机；交互性。用户能够方便地与系统进行人-机对话；独立性。系统中多个用户可以彼此独立地进行操作，互不干扰，单个用户感觉不到别人也在使用这台计算机，好像只有自己单独使用这台计算机一样；及时性。用户请求能在很短时间内获得响应。分时系统釆用时间片轮转方式使一台计算机同时为多个终端服务，使用户能够对系统的及时响应感到满意。

问题有：虽然分时操作系统比较好地解决了人机交互问题，但是在一些应用场合，需要系统能对外部的信息在规定的时间（比时间片的时间还短）内作出处理（比如飞机订票系统或导弹制导系统)。因此，实时系统应运而生。

如何选取时间片：确定时间片长度要从进程数目、切换开销、系统效率和响应时间等多方面因素加以考虑。如果时间片取值太小，将导致大多数进程/线程都不可能在一个时间片内运行完毕，就会频繁切换，开销显著增大，所以从系统效率来讲，时间片应该大些好；如果时间片长度较大，那么随着就绪队列中进程/线程数目的增加，轮转一次所耗费的总时间加长，即对每个进程/线程的响应速度放慢，甚至时间片大到让进程/线程足以完成其所有任务，时间片调度算法便退化为FCFS算法。为了满足用户对响应时间的要求，要么限制就绪队列中进程/线程的数量，要么采用变化的时间片长度，根据当前负载情况及时调整时间片大小。

实时操作系统：为了能在某个时间限制内完成某些紧急任务而不需时间片排队，诞生了实时操作系统。这里的时间限制可以分为两种情况：如果某个动作必须绝对地在规定的时刻（或规定的时间范围）发生，则称为硬实时系统。例如，飞行器的飞行自动控制系统，这类系统必须提供绝对保证，让某个特定的动作在规定的时间内完成。如果能够接受偶尔违反时间规定，并且不会引起任何永久性的损害，则称为软实时系统，如飞机订票系统、银行管理系统。在实时操作系统的控制下，计算机系统接收到外部信号后及时进行处理，并且要在严格的时限内处理完接收的事件。实时橾作系统的主要特点是及时性和可靠性。

8、Unix操作系统允许用户在汇编语言上使用系统调用。

9、计算机结构分析自底向上依次是：

硬件->驱动程序->内核->用户接口程序->应用程序。

用户接口程序：基于文本的叫shell，基于图标的叫图形用户界面（GUI）如Gnome、KDE。

10、OS的不确定性是指同一道程序的多次运行所需的时间不确定，同一批程序的多次运行的运行序列和总运行时间不确定。

11、并发和共享是操作系统的两个最基本的特性，它们又是互为存在条件。一方面资源共享是以程序（进程）的并发性执行为条件的，若系统不允许程序并发执行，自然不存在资源共享问题。另一方面若系统不能对资源共享实施有效管理，则也必将影响到程序并发执行。

12、并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔内发生。CPU的数据处理和I/O操作往往是并行处理的。

13、cpu工作状态分为系统态（或称管理态，管态）和用户态（或称目态）。 引入这两个工作状态的原因是：为了避免用户程序错误地使用特权指令，保护操作系统不被用户程序破坏。具体规定为：当cpu处于用户态时，不允许执行特权指令，当cpu处于系统态时，可执行包括特权指令在内的一切机器指令

三种情况下会出现由用户态转为内核态，中断，异常，系统调用

凡是涉及到计算机根本运行的事情都应该在内核态下执行，而中断、时钟日期、存储映象图都属于系统级（相对应的是用户级）的资源，对这些资源的修改都必须在核心态，但是读取则没有强制要求。

14、设备管理从下到上：设备》设备控制器》通道》处理机。一条大型机通道（channel）某种程度上类似于PCI 总线（bus），它能将一个或多个控制器连接起来，而这些控制器又控制着一个或更多的设备（磁盘驱动器、终端、LAN端口，等等。）

过程：CPU正在工作，突然想到要与外设通信，于是发命令给通道，然后接着做自己的工作。通道接到命令后，接通外设与内存，并在他们之间传递数据，等数据传递完成后，通知CPU进行处理。所以说通道完成了内存与外设之间数据的传输。

15、当出现多道的时候，就会出现优先级的问题，因此特权指令是为了提供在优先级不同的条件下，用户状态的切换，跳转指令就是从我们的程序跳转到另一个了，可能会感觉时钟会有用，给每一个程序分配一定的时间，让其运行，但是这个并不是必须的，因为可以根据优先级来采取抢占式的执行，所以时钟不是特别提供的，特权指令和跳转指令在单道批处理下是不能够起到作用的

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*文件系统\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、采用顺序结构的文件既适合进行顺序访问，也适合进行随机访问。

2、文件分配对应于文件的物理结构，是指如何为文件分配磁盘块。常用的磁盘空间分配方法有三种：连续分配、链接分配和索引分配。

顺序分配：顺序 分配方法要求每个文件在磁盘上占有一组连续的块。

隐式链接分配： 每个文件对应一个磁盘块的链表；磁盘块分布在磁盘的任何地方，除最后一个盘块外，每一个盘块都有指向下一个盘块的指针，这些指针对用户是透明的。

显式链接分配：是指把用于链接文件各物理块的指针，显式地存放在内存的一张链接表中。 该表在整个磁盘仅设置一张，每个表项中存放链接指针，即下一个盘块号。 在该表中，凡是 属于某一文件的第一个盘块号，或者说是每一条链的链首指针所对应的盘块号，均作为文件 地址被填入相应文件的FCB的“物理地址”字段中。由于查找记录的过程是在内存中进行 的，因而不仅显著地提高了检索速度，而且大大减少了访问磁盘的次数。由于分配给文件的 所有盘块号都放在该表中，故称该表为文件分配表（File Allocation Table, FAT)。MS-DOS 采用的就是这种方式。

3、文件的存储方法依赖于文件的物理结构和存放文件的存储设备的特性

4、文件系统最基本的目标是按名存取，它主要是通过目录管理功能来实现的。

5、文件的物理结构指文件在外存物理存储介质上的结构，它可分为连续分配、链接分配和索引分配三种形式。

6、按使用情况，文件可分为临时文件、永久文件和档案文件。

7、文件的逻辑机构是从用户的观点出发所观察到的文件组织形式，是用户可以直接处理的数据及其结构，它独立于文件的物理特性，又称为文件组织。文件的物理结构，又称为文件的存储结构，是指文件在外存上的存储组织形式。这不仅与存储性能有关，而且与所采用的外存分配方式有关。

8、"删除文件"操作的系统处理过程。

用户用本操作向系统提出删除一个文件的要求，系统执行时把指定文件的名字从目录和索引表中除去，并收回它所占用的存储区域，但删除一个文件前应先关闭该文件。

"打开文件"操作的系统处理过程。

用户要使用一个已经存放在存储介质上的文件前，必须先提出"打开文件"要求。这时用户也必须向系统提供参数：用户名、文件名、存取方式、存储设备类型、口令等。系统在接到用户的"打开文件"要求后，找出该用户的文件目录，当文件目录不在主存储器中时还必须把它读到主存储器中；然后检索文件目录，指出与用户要求相符合的目录项，取出文件存放的物理地址。对索引文件还必须把该文件的索引表存放在主存储器中，以便后继的读写操作能快速进行。

9、文件是具有符号名的、在逻辑上具有完整意义的一组相关信息项的有序序列。文件系统就是操作系统中实现文件统一管理的一组软件、被管理的的文件以及为实施文件管理所需的一些数据结构的总称。

文件系统的功能：1、统一管理文件存储空间（即外存），实施存储空间的分配与回收；2、确定文件信息的存放位置及存放形式；3、实现文件从名字空间到外存地址空间的映射，即实现文件的按名存取；4、有效实现对文件的各种控制操作（如建立、撤消、打开、关闭文件等）和存取操作（如读、写、修改、复制、转储等）；5、实现文件信息的共享，并且提供文件保密和保护措施。文件系统的优点：1、按名存取文件，以对用户透明的方式实现对名字空间的管理和信息浮动，使用方便灵活；2、采取保护、保密措施，安全可靠；3、实现文件共享，节省空间和时间开销。

10、根据不同角度，可以将文件划分为不同类别：1、按性质和用途可分为：系统文件；库文件；用户文件；2、按信息的保存期限可分为：临时文件；永久性文件；档案文件；3、按文件的保护方式可分为：只读文件；读写文件；可执行文件；无保护文件；4、按文件的逻辑结构可分为：流式文件；记录式文件；5、按文件的物理结构可分为：顺序文件；链接文件；索引文件；Hash文件；索引顺序文件6、按文件的存取方式可分为：顺序存取文件；随机存取文件；7、按文件内容可分为：普通文件；目录文件；特殊文件

11、文件控制块的有序集合构成文件目录，每个目录项即是一个文件控制块。为了实现文件目录的管理，通常将文件目录以文件的形式保存在外存空间，这个文件就被称为目录文件。目录文件是长度固定的记录式文件。系统为用户提供一个目前正在使用的工作目录，称为当前目录。

12、文件目录结构一般有一级目录结构、二级目录结构和多级目录结构。一级目录结构的优点是简单，缺点是文件不能重名，限制了用户对文件的命名。二级目录结构实现了文件从名字空间到外存地址空间的映射：用户名->文件名文件内容。其优点是有利于文件的管理、共享和保护；适用于多用户系统；不同的用户可以命名相同文件名的文件，不会产生混淆，解决了命名冲突问题。缺点是不能对文件分类；当用文件较多时查找速度慢。多级目录结构的优点是便于文件分类，可为每类文件建立一个子目录；查找速度快，因为每个目录下的文件数目较少；可以实现文件共享；缺点是比较复杂。

13、为了提高检索速度，对文件目录应做怎样的改进？

可以利用目录项分解法解决这一问题，即把目录项（文件控制块）分为两部分：名号目录项，包含文件名以及相应的文件内部号；基本目录项，包含了除文件名外文件控制块的其他全部信息。目录文件也分为名号目录文件和基本目录文件。查找一个目录项就分成两步：首先访问名号目录文件，根据文件名查找相应的文件内部号；然后访问基本目录文件，根据文件内部号，可直接计算出相应基本目录项所在基本目录文件中的相对位置和物理位置，并将它直接读入内存。目录项分解法的优点是提高了文件目录检索的速度。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*进程和线程\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、从执行状态挂起的进程解除挂起时进入就绪状态。

挂起（等待,阻塞）进程在操作系统中可以定义为暂时被淘汰出内存的进程。机器的资源是有限的，在资源不足的情况下，操作系统对在内存中的程序进行合理的安排，其中有的进程被暂时调离出内存，当条件允许的时候，会被操作系统再次调回内存，重新进入等待被执行的状态即就绪态。

我们知道进程分为三个状态，运行，就绪，阻塞。 某进程在运行过程中等待的事件已发生，注意是在运行中，运行中是可以通过等待I/O来中断这个这个进程，此时开始打印，但是CPU就会去处理其他的进程，这个进程就会处于 阻塞状态。

进程若已获得除CPU以外的所有运行所需资源，此时申请CPU得不到满足，应该依然处于就绪状态。正在运行的进程，因为某种原因而暂停，此时处于阻塞状态(也称等待状态、挂起状态)。如果资源腾出来了，在等待的进程也就是阻塞的进程就应该进入就绪状态等待被执行

2、动态优先权是指在创建进程时所赋予的优先权，是可以随进程的推进或随其等待时间的增加而改变的，以便获得更好的调度性能。例如，我们可以规定，在就绪队列中的进程，随其等待时间的增长，其优先权以速率a提高。若所有的进程都具有相同的优先权初值，则显然是最先进入就绪队列的进程将因其动态优先权变得最高而优先获得处理机，此即FCFS算法。

3、进程的调度（属于处理机管理）方式包括非剥夺方式和剥夺方式。

非剥夺方式： 分派程序一旦把处理机分配给某进程后便让它一直运行下去，直到进程完成或发生某事件而阻塞时，才把处理机分配给另一个进程。

剥夺方式： 当一个进程正在运行时，系统可以基于某种原则，剥夺已分配给它的处理机，将之分配给其它进程。剥夺原则有：优先权原则、短进程优先原则、时间片原则。

4、周转时间：是指，从一个批处理作业提交时刻开始直到该作业完成时刻为止的统计的平均时间

响应时间：是指：作业等待时间+作业计算时间

运行时间：作业计算时间

等待时间：作业从后备作业到被调度程序选中的时间

5、创建进程的步骤

申请空白PCB（进程控制块，是系统为了管理进程设置的一个专门的数据结构。系统用它来记录进程的外部特征，描述进程的运动变化过程。同时，系统可以利用PCB来控制和管理进程，所以说，PCB是系统感知进程存在的唯一标志。）；为新进程分派资源；初始化PCB；将新进程插入就绪队列；

调用进程时才分配CPU

6、并发执行时，系统中的各资源将为它们所共享，而这些资源的状态也有这些程序来改变，致使其中的任意程序在运行时，其环境必然会受到其他程序的影响。初始数据的更改也会影响结果，两个程序去改一个变量，总要乱的.因此程序失去了封闭性（变量只受一个程序的更改）。程序运行结果的不确定性，即失去了再现性（运行结果的确定性）

7、信号量表示可用的资源数，s初值为2表示有2个可用资源。而当前值为-1表示有一个进程没有分到资源而在等待，所以有1个等待进程。有一个进程获取资源则S--；有一个进程释放资源则S++。

8、原语：它是由若干条机器指令所构成，用以完成特定功能的一段程序，为保证其操作的正确性，它应当是原子操作，即原语是一个不可分割的操作。所以，原语在执行的过程中，也是不能响应中断的。

9、JCB: Job Control Block

在操作系统的作业管理中，系统为每个作业建立一个 作业控制块 JCB。系统通过JCB感知作业的存在。JCB包括的主要内容有作业名、作业状态、资源要求、作业控制方式、作业类型以及作业优先权等。

PCB: Process Control Block

进程的静态描述由三部分组成

PCB、有关程序段和该程序段对其进行操作的数据结构集。

在Unix或类Unix系统中，进程是由进程控制块，进程执行的程序，进程执行时所用数据，进程运行使用的工作区组成。其中进程控制块是最重要的一部分。

进程控制块是用来描述进程的当前状态，本身特性的数据结构，是进程中组成的最关键部分，其中含有描述进程信息和控制信息，是进程的集中特性反映，是操作系统对进程具体进行识别和控制的依据。

PCB一般包括：

（1.程序ID（PID、进程句柄）：它是唯一的，一个进程都必须对应一个PID。PID一般是整型数字

（2.特征信息：一般分系统进程、用户进程、或者内核进程等

（3.进程状态：运行、就绪、阻塞，表示进程现的运行情况

（4.优先级：表示获得CPU控制权的优先级大小

（5.通信信息：进程之间的通信关系的反映，由于操作系统会提供通信信道

（6.现场保护区：保护阻塞的进程用

（7.资源需求、分配控制信息

（8.进程实体信息，指明程序路径和名称，进程数据在物理内存还是在交换分区（分页）中

（9.其他信息：工作单位，工作区，文件信息等

10、如果系统只有用户态线程，则线程对操作系统是不可见的，操作系统只能调度进程；

如果系统中有内核态线程，则操作系统可以按线程进行调度；

11、并发性是指两个或多个事件在同一 时间间隔 内发生。

同时性是指两个或多个事件在同一 时刻 发生

幷发的实质是一个物理CPU（也可以多个物理CPU）在若干道程序之间多路复用，并发性是对有限物理资源强制行驶多用户共享以提高效率。并发性是关于软件过程分解成进程、线程并处理相关的效率、原子性、同步和调度问题。

实现幷发技术的关键之一是如何对系统内的多个活动（进程）进行切换。

并行性指的是两个或两个以上的事件或活动在同一 时刻 发生。在多道程序环境下，并行性使多个程序同一时刻可在不同CPU上同时执行。

并行性和并发性的区别：

并行的时间或者活动一定是并发的，但是反之并发的时间或者活动未必是并行的。并行性是并发性的特例，而并发性是并行性的拓展。

12、Unix把进程分成两大类：

一类是系统进程，另一类是用户进程。系统进程执行操作系统程序，提供系统功能，工作于核心态。用户进程执行用户程序，在操作系统的管理和控制下执行，工作于用户态。进程在不同的状态下执行时拥有不同的权力。

在Unix系统中进程由三部分组成，分别是进程控制块、正文段和数据段。Unix系统中把进程控制块分成proc结构和user结构两部分

proc存放的是系统经常要查询和修改的信息，需要快速访问，因此常将其装入内存

Linux创建进程的时候，同时创建进程控制块和进程对应的堆栈。该堆栈分为用户栈和系统栈，也就是核心栈。用户栈在用户态使用，用来保存局部变量，函数参数等，而系统栈在进程陷入内核态使用，里面保存的是用户栈的地址，以及在进行进程上下文切换时需要保存的参数，返回值等。因为每个进程都有可能发生系统调用陷入内核，从用户态到内核态发生一个中断，那么内核栈就会保存这个进程操作内核的调用信息，如果常驻内存的话就会被下个进程发生的系统调用的信息覆盖掉，所以内核栈是不会常驻内存的。

13、抢占式保证了高优先级的进程可以直接抢夺低优先级进程的资源，优先运行；

静态优先权是指：在进程运行前就已经分配了固定的优先权，在进程运行的过程中不再对优先权进行调整，这样就导致了低优先级的进程一直得不到资源，即“饥饿”状态！

14、在多道程序环境中，只有进程才能在系统中运行。因此为了使程序运行必须为其创建进程，而导致进程创建的时间典型的有四种：

用户登录。可以理解为，一个新的用户来了，需要为它提供服务，这个服务之前没有，所以要创建。

作业调度。系统会为调度的作业分配资源，从后备队列中将其放入内存中，并为其创建进程。

提供服务。

应用请求。

设备中断和执行系统调用只是会造成进程模式的切换，由用户态切换到系统态

15、程序和进程的区别：

进程由程序和数据两部分组成，进程是竞争计算机系统有限资源的基本单位，也是进程处理机调度的基本单位。程序是静态的概念；进程是程序在处理机上一次执行的过程，是动态的概念。一个程序可以作为多个进程的运行程序；一个进程也可以运行多个程序。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*互斥和死锁\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、互斥的使用临界资源是通过互斥的进入临界区实现的。互斥指的是某种资源一次只允许一个进程使用，即你在使用的时候我不能使用；我在使用的时候你不能使用。这就是一种协调，一种"步伐"上的一致，因而也就是一种同步。但是，为了求解实际问题，将"同步"与"互斥"加以区别是有好处的，因为这两种问题的求解方法是不同的。

2、避免死锁：银行家算法

在避免死锁方法中允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次分配资源的安全性，若分配不会导致系统进入不安全状态，则分配，否则等待。该方法将系统的状态分为安全和不安全,只要系统处于安全状态,便可避免 死锁 的发生。

算法原理：在银行中，客户申请贷款的数量是有限的，每个客户在第一次申请贷款时要声明完成该项目所需的最大资金量，在满足所有贷款要求时，客户应及时归还。银行家在客户申请的贷款数量不超过自己拥有的最大值时，都应尽量满足客户的需要。在这样的描述中，银行家就好比操作系统，资金就是资源，客户就相当于要申请资源的进程。

所以有几个关键的数据结构：每个进程当前需要的资源NEED[i][j] 每个进程最多需要的资源MAX[i][j] 每个进程当前已分配到的资源ALLOCATION[i][j] 当前可用的资源AVAILABLE[i][j]

算法假设：每个进程都可以得到他想要的资源，并且得到资源后执行，执行后释放所用的资源

算法伪代码：

P - 进程的集合

Mp - 进程p的最大的请求数目

Cp - 进程p当前被分配的资源

A - 当前可用的资源

while(P!=NULL)

{

found=FALSE;

for each P{

if(Mp-Cp<=A)

{

A=A+Cp;

P=P-{p};

found=TRUE

}

}

if(!found)

return FAIL;

}

注意：银行家算法中的安全序列不唯一，进程获得资源的顺序可以不同。比起其他解决死锁的方法，该算法需要较多的数据结构，实现起来比较困难，但是资源利用率最高。

3、解除死锁：消进程法（强制撤销部分、甚至全部死锁进程并剥夺这些进程的资源。撤销的原则可以按进程优先级和撤销进程代价的高低进行。具体来说，可以考虑如下几点：优先权；进程已执行的时间；估计剩余的执行时间；已产生的输出量；已获得的资源量和资源类型；还需要的资源量；进程的类型；需要撤销的进程数目），剥夺资源（挂起某些死锁进程，并抢占它的资源，将这些资源分配给其他的死锁进程。但应防止被挂起的进程长时间得不到资源，而处于资源匮乏的状态。），进程回退法（让一（多）个进程回退到足以回避死锁的地步，进程回退时自愿释放资源而不是被剥夺。要求系统保持进程的历史信息，设置还原点）

4、预防死锁：资源静态分配法.静态分配资源，所有进程在开始运行之前，一次性地申请其在整个运行过程所需的全部资源。但在分配资源时，只要有一种资源不能满足某进程的要求，即使它所需的其他资源都空闲，也不分配给该进程，而让进程等待。在进程的等待期间，它并未占有任何资源，摒弃了“保持”条件，避免发生死锁。该方法最容易实现，但是系统资源利用率低。即有序分配资源可以预防死锁。

5、检测死锁：资源分配图简化法。资源分配图中用圆圈代表一个进程，用框代表一类资源。由于一种类型的资源可能有多个，用框中的一个点代表一类资源中的一个资源。从进程到资源的有向边叫请求边，表示该进程申请一个单位的该类资源；从资源到进程的边叫分配边，表示该类资源已经有一个资源被分配给了该进程。http://c.biancheng.net/cpp/html/2607.html

可以通过将资源分配图简化的方法来检测系统状态S是否为死锁状态。简化方法如下：1)在资源分配图中，找出既不阻塞又不是孤点的进程Pi（即找出一条有向边与它相连，且该有向边对应资源的申请数量小于等于系统中已有空闲资源数量。若所有的连接该进程的边均满足上述条件，则这个进程能继续运行直至完成，然后释放它所占有的所有资源）。消去它所有的请求边和分配边，使之成为孤立的结点。2)进程Pi所释放的资源，可以唤醒某些因等待这些资源而阻塞的进程，原来的阻塞进程可能变为非阻塞进程。在图2-17中，进程P2就满足这样的条件。根据第1)条中的方法进行一系列简化后,若能消去图中所有的边，则称该图是可完全简化的。S为死锁的条件是当且仅当S状态的资源分配图是不可完全简化的,该条件为死锁定理。

6、所谓死锁： 是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

死锁的四个必要条件：不可剥夺（系统中存在一个资源只能被占有它的进程所释放，而不能被别的进程强行抢占）；互斥（系统中存在一个资源一次只能被一个进程所使用）；请求和保持（占着有的资源，伸手要没有的）；环路等待（你等着我我却也等着你）。破坏一个即可破坏死锁。

产生死锁的原因有：资源不足或进程推进次序不当。

7、生产者和消费者问题是一个既有同步又有互斥的问题。生产者和消费者对缓冲区的使用是一个互斥问题，生产者生产数据，消费者消费数据，这样看的话就又是一个同步问题。

8、管程：引入管程是为了让系统自动处理临机资源的互斥使用问题。

使用PV原语实现的同步机制对于共享变量及信号变量的操作被分期在各个进程中，造成如下缺点：程序可读性差；不利于维护；正确性很难保证；为了更易于编写正确的程序，所以提出了一种高级同步原语，即管程。它由过程、变量及数据结构组成一个集合，是一个特殊的模块或软件包。一次只能有一个进程可以在管程内活动是它的一个重要特性。这个特性是管程本身实现的，程序员不必显示地编写代码去实现这种同步制约。同时管程属于语言特性，由编译器提供支持，c语言不支持它。总的来说，管程是为了让程序员从同步机制的细节中解脱的一种方法。

9、系统中的资源可以分为两类，一类是可剥夺资源，是指某进程在获得这类资源后，该资源可以再被其他进程或系统剥夺。例如，优先权高的进程可以剥夺优先权低的进程的 处理机 。又如，内存区可由 存储器管理 程序，把一个进程从一个存储区移到另一个存储区，此即剥夺了该进程原来占有的存储区，甚至可将一进程从内存调到外存上，可见， CPU 和 主存 均属于可剥夺性资源。另一类资源是不可剥夺资源，当系统把这类资源分配给某进程后，再不能强行收回，只能在进程用完后自行释放，如 磁带机 、打印机等。

10、进程调度算法各种各样，如果选择不当，就会造成进程长时间等待，而不会造成死锁。

11、在计算机中有许多资源一次只能允许一个进程使用，如果多个进程同时使用这些资源，则有可能造成系统的混乱，这些资源被称为临界资源 。

12、何谓安全分配方式和不安全分配方式？

安全分配方式和不安全分配方式的说明如下：（1）安全分配是一种"摒弃请求和保持条件"的资源分配方式。在这种方式中，一个进程请求资源一旦获得（例如I/O请求时获得所需的设备），该进程就由运行状态变为阻塞状态，使它不可能再请求新资源。相反，当该进程再开始运行时（例如I/O完成后被唤醒），它已不再占有该资源。因此，这种分配摒弃了造成死锁的一个条件，分配是安全的。这种分配方式的缺点是进程推进速度慢，因为进程与I/O是串行的。（2）不安全的分配方式是指进程在提出字请求时（例如I/O请求）并不将它阻塞，而是允许它继续使用CPU，并提出第二次资源请求。这样，若第二次请求的资源已被其他进程占用使该进程被阻塞时，则该进程具备了"请求和保持"条件，可能产生死锁，因而说，这种分配是不安全的分配。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*内存管理\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、虚拟存储的实现式基于程序局部性（程序的局部性原理是指在一段时间内，一个程序执行往往呈现高度局部性，对内存访问时不均匀的，这种局部性包括时间局部性和空间局部性。）原理，其实质是借助外存将内存较小的物理地址空间转化为较大的逻辑地址空间。

根据程序执行的互斥性和局部性两个特点，我们允许作业装入的时候只装入一部分，另一部分放在磁盘上，当需要的时候再装入到主存，另一方面，操作系统将内存中暂时不使用的内容换出到外存上，从而腾出空间存放将要调入的信息。这样以来，在一个小的主存空间就可以运行一个比它大的作业。同时，用户编程的时候也摆脱了一定要编写小于主存容量的作业的限制。也就是说，用户的逻辑地址空间可以比主存的绝对地址空间要大。对用户来说，好像计算机系统具有一个容量很大的主存储器，称为“虚拟存储器”。充分利用了地址空间而非提高运行速度

虚拟存储（Storage Virtualization）是指将多个不同类型、独立存在的物理存储体，通过软、硬件技术，集成转化为一个逻辑上的虚拟的存储单元，集中管理供用户统一使用。这个虚拟逻辑存储单元的存储容量是它所集中管理的各物理存储体的存储量的总和，而它具有的访问带宽则在一定程度上接近各个物理存储体的访问带宽之和。

虚拟存储器最大实际容量= min(计算机地址，内存+辅存)。计算机地址（CPU逻辑地址给出的空间大小）=2^段号\*2^段内地址。

其逻辑容量由内存容量和外存容量之和所决定 ，其运行速度接近于内存速度，而每位的成本却又接近于外存。

实存要求一程序全部装入内存才开始运行，虚存允许程序在执行的过程中逐步装入。

2、适应于请求段的内存分配方法是最佳适应和可变分区。

最佳适应算法：空闲分区按容量递增形成分区链，找到第一个能满足要求的空闲分区。这种方法能使碎片尽量小。

例如 空闲容量 首地址

60KB 330KB

80KB 100KB

90KB 190KB

若容量需要是40KB, 因此选择首地址是330

最先适应算法：通常将空闲区按地址从小到大排列。查找时总是从低地址开始，可使高地址尽量少用，以保持一个大空闲区，有利于大作业的装入；缺点是内存低地址和高地址两端的分区利用不平衡，回收分区较麻烦。

最坏适应算法：通常将空闲区按长度递减顺序排列。查找时从最大的一个空闲区开始，总是挑选一个最大的空闲区分割给作业使用，其优点是使剩下的空闲区不致于太小，这样有利于中小型作业，但不利于大作业。

3、如果一个程序为多个程序所共享，那么该程序的代码在执行的过程中不能被修改，即程序应该是可重入码。可重入码：当被多个线程调用的时候，不会引用任何共享数据，他们是线程安全（被多个并发线程反复调用时，会一直产生正确的结果）的。（注意与线程安全做区别 可重入是线程安全的真子集）

4、能够装入内存任何位置的代码程序必须是可动态链接的。

程序的链接有以下三种方式：

静态链接：在程序运行之前，先将各目标模块及它们所需的库函数链接成一个完整的可执行程序，以后不再拆开。

装入时动态链接：将用户源程序编译后所得到的一组目标模块，在装入内存时，釆用边装入边链接的链接方式。

运行时动态链接：对某些目标模块的链接，是在程序执行中需要该目标模块时，才对它进行的链接。其优点是便于修改和更新，便于实现对目标模块的共享。

内存的装入模块在装入内存时，同样有以下三种方式：

1)绝对装入。在编译时，如果知道程序将驻留在内存的某个位置，编译程序将产生绝对地址的目标代码。绝对装入程序按照装入模块中的地址，将程序和数据装入内存。由于程序中的逻辑地址与实际内存地址完全相同，故不需对程序和数据的地址进行修改。

绝对装入方式只适用于单道程序环境。另外，程序中所使用的绝对地址,可在编译或汇编时给出，也可由程序员直接赋予。而通常情况下在程序中釆用的是符号地址，编译或汇编时再转换为绝对地址。

2)可重定位（重定位：就是把作业地址空间中使用的逻辑地址变换为存储空间的物理地址的过程，又称为地址映射）装入。在多道程序环境下，多个目标模块的起始地址通常都是从0开始，程序中的其他地址都是相对于起始地址的,此时应釆用可重定位装入方式。根据内存的当前情况，将装入模块装入到内存的适当位置。装入时对目标程序中指令和数据的修改过程称为重定位，地址变换通常是在装入时一次完成的，所以又称为静态重定位。

静态重定位的特点是在一个作业装入内存时，必须分配其要求的全部内存空间，如果没有足够的内存，就不能装入该作业。此外，作业一旦进入内存后，在整个运行期间不能在内存中移动，也不能再申请内存空间。

3)动态运行时装入，也称为动态重定位，程序在内存中如果发生移动，就需要釆用动态的装入方式。装入程序在把装入模块装入内存后，并不立即把装入模块中的相对地址转换为绝对地址，而是把这种地址转换推迟到程序真正要执行时才进行。因此，装入内存后的所有地址均为相对地址。这种方式需要一个重定位寄存器的支持。总结来说就是将指令中的有效地址 与 重定位寄存器中的基址 相加得到 访问地址 ，从而实现地址动态修改。

动态重定位的特点是可以将程序分配到不连续的存储区中；在程序运行之前可以只装入它的部分代码即可投入运行，然后在程序运行期间，根据需要动态申请分配内存；便于程序段的共享，可以向用户提供一个比存储空间大得多的地址空间。

综上，动态重定位依赖于重定位寄存器；地址转换器；目标程序（因为要对目标程序的指令和数据做一个修改，从而实现地址映射）

5、页式管理有：静态页式管理；动态页式管理；其中，静态页式管理是在作业或进程执行前，把作业或进程全部装进内存中，如果内存中可用页面数小于请求页面数，该作业或进程等待。动态页式管理不会把作业或进程一次性全部装进内存，只装入被反复调用或执行的部分，其他部分在执行过程中动态装入。

6、主存与辅存的区别：

概括的说，CPU对所需要的数据进行计算时，要求很高的存储速度，且不需要能永久保存这些数据，高速存储设备的成本很高。

但其他设备对存储速度的要求不像CPU这么高，一般要求永久保存数据。一般低速的存储设备就可以满足，且低速的存储成本也低。

所以有主存和辅存之分：

内存（主存）直接给CPU提供存储，高速，低容量，价格贵，不能永久保存数据，断电消失，需要从辅存中重新调入数据。

外存（辅存）给主存提供数据，低速，大容量，价格低，能永久保存数据。

所以更高缓存的CPU和更大的内存能够大大提升系统的性能。

常见主存有：CPU的高速缓存，电脑的内存条。

常见辅存有：硬盘、光盘、U盘、磁盘、移动硬盘等等。

7、I/O中断：例如缺页中断（在请求分页存储管理中，当访问的页面不在内存时，便产生缺页中断）

外部中断：一般是由计算机外设发出的中断请求，如：键盘中断、打印机中断、定时器中断等。

访管中断：用户程序在运行中请求操作系统为其提供服务而执行一条“访管指令”所引起的中断,又称软件中断。访管中断是进程所期待的,它是自愿性的中断,又称自愿中断,其它几种中断不是运行进程所期望的,属于强迫性中断事件。

8、一般而言，计算机中虚拟存储器容量（个数）最多。要注意ROM不是硬盘，只有BIOS那一小块。

9、在内存管理中，内部碎片是已经被分配出去的的内存空间大于请求所需的内存空间。外部碎片是指还没有分配出去，但是由于大小太小而无法分配给申请空间的新进程的内存空间空闲块。采用固定分区存在内部碎片，可变式分区分配会存在外部碎片；页式虚拟存储系统存在内部碎片；段式虚拟存储系统，存在外部碎片。为了有效的利用内存，使内存产生更少的碎片，要对内存分页，内存以页为单位来使用，最后一页往往装不满，于是形成了内部碎片。为了共享要分段，在段的换入换出时形成外部碎片，比如5K的段换出后，有一个4k的段进来放到原来5k的地方，于是形成1k的外部碎片。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*输入输出\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、根据通道的工作方式，通道分为字节多路通道、选择通道、数组多路通道三种类型。一个系统可以兼有多种类型的通道，也可以只有其中一、二种。

字节多路通道：字节多路通道是一种简单的共享通道，主要用于连接控制多台低速外设，以字节交叉方式传送数据。例如，某个外设的数据传输率只有1000B/s，即传送1个字节的时间间隔是1ms，而通道从设备接收或发送一个字节只需要几百ns，因此，通道在传送两个字节之间有很多空闲时间，字节多路通道正是利用这个空闲时间为其他设备提供服务。每个设备分时占用一个很短的时间片，不同的设备在各自分得的时间片内与通道建立连接，实现数据的传输。总结：字节多路通道是分时并行的，但不适用于高速设备。

选择通道：选择通道又称高速通道，在物理上它可以连接多个设备，但是这些设备不能同时工作，在某一个时间段内通道只能选择一个设备进行工作。选择通道很像一个单道程序的处理器，在一段时间内只允许执行一个设备的通道程序，只有当这个设备的通道程序全部执行完毕后，才能执行其他设备的通道程序。选择通道主要用于连接高速外围设备，如磁盘、磁带等，信息以成组方式高速传输。由于数据传输率很高，如达到1.5MB/s，通道在传送两个字节之间只有很少的空闲时间，所以，在数据传送期间只为一台设备服务是合理的。但是，这类设备的寻址等辅助操作的时间往往很长，在这样长的时间里通道一直处于等待状态，因此，整个通道的利用率还不是很高。总结：数组选择通道有很高的传输速率，但只含有一个分配型子通道，一段时间只能执行一道通道程序。

数组多路通道：连接控制多个高速外设并以成组交叉方式传送数据的通道称为数组多路通道。数组多路通道是对选择通道的一种改进，当某个设备进行数据传送时，通道只为该设备提供服务；当设备在执行寻址等控制性动作时，通道暂时断开与该设备的连接，挂起该设备的通道程序，而转去为其他设备提供服务，即执行其他设备的通道程序。所以，数组多路通道很像一个多道程序的处理器。对于磁盘一类的高速外设，采用数组多路通道，可在其中一个外设占用通道进行数据传送时，让其他外设进行寻址等辅助操作，使一个设备的数据传送操作与其他设备的寻址操作彼此重叠，实现成组交叉方式的数据传送，从而使通道具备多路并行工作的能力，充分发挥通道高速信息交换的效能。由于数组多路通道既保留了选择通道高速传送数据的优点，又充分利用控制性操作的时间间隔为其他设备提供服务，使通道的效率得到充分的发挥，因此，数组多路通道在实际系统中得到较多的应用。总结：数组多路通道是上述两者的结合，兼具分时并行和高传输速率的优点。

2、SPOOLing （即 外部设备 联机并行操作），即Simultaneous Peripheral Operation On-Line的缩写，它是关于慢速字符设备如何与计算机主机交换信息的一种技术，通常称为“ 假脱机 技术”。

SPOOLing技术特点：提高了I/O速度.从对低速I/O设备进行的I/O操作变为对输入井或输出井的操作,如同脱机操作一样,提高了I/O速度,缓和了CPU与低速I/O设备速度不匹配的矛盾；设备并没有分配给任何进程.在输入井或输出井中,分配给进程的是一存储区和建立一张I/O请求表；实现了虚拟设备功能.多个进程同时使用一独享设备,而对每一进程而言,都认为自己独占这一设备,不过,该设备是逻辑上的设备.

3、I/O通道的目的是为了建立独立的I/O通道，使得原来一些由CPU处理的I/O任务由通道来承担，从而解脱cpu。通道所能执行的命令局限于I/O操作的指令，也就是执行I/O指令集。

IO重定向是指将命令的执行结果重新导出到其他的设备或者文件。主要包括输入重定向和输出重定向

4、对于单缓冲：

假定从磁盘把一块数据输入到缓冲区的时间为T，操作系统将该缓冲区中的数据传送到用户区的时间为M，而CPU对这一块数据处理的时间为C。由于T和C是可以并行的，当T>C时，系统对每一块数据的处理时间为M十T，反之则为M+C，故可把系统对每一块数据的处理时间表示为Max(C, T)+M。

对于双缓冲：

系统处理一块数据的时间可以粗略地认为是MAX(C,T)。如果C<T，可使块设备连续输入；如果C>T，则可使CPU不必等待设备输入。对于字符设备，若釆用行输入方式，则釆用双缓冲可使用户在输入完第一行之后，在CPU执行第一行中的命令的同时，用户可继续向第二缓冲区输入下一行数据。而单缓冲情况下则必须等待一行数据被提取完毕才可输入下一行的数据。总之就是双缓冲的时候可以并行处理，基本就是在“把信息输入到双缓冲区”的时间内可以同时完成其他的事情

5、在操作系统中，响应时间指用户发出请求或者指令到系统做出反应（响应）的时间

6、通道是一个独立与CPU的专管输入/输出控制的处理机，它控制设备与\*内存\*直接进行数据交换。引入通道的目的是让数据的传输独立于CPU，使CPU从繁重的I/O工作中解脱出来。它有自己的通道指令，这些指令受CPU启动，并在操作结束向CPU发出中断信号。通道技术主要是为了减轻CPU的工作负担，增加了计算机系统的并行工作程度。

虚拟存储器：它使得应用程序认为它拥有连续的可用的内存，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。

并行技术可以分为多进程编程和多线程编程。通常用IPC的形式来实现进程间的同步，如管道，信号量，消息队列或者共享存储。在所有IPC敏感词享存储器是最快的。

缓冲技术是为了缓和CPU和I/O设备速度不匹配的矛盾，提高CPU和I/O设备的并行性，减少对CPU的中断频率，放宽对蓄洪大响应时间的限制，在现代操作系统中，几乎所以的I/O设备在与处理机交换数据时都用了缓存区，并提供获得和释放缓冲区的手段。总的来说，缓冲区技术用到了缓冲区，而缓冲区的引入是为了缓和CPU和I/O设备速度不匹配，从而可以有效的减少CPU的终端频率，提高CPU和I/O设备的并行性。

7、共享设备允许多个作业同时使用，这里的"同时使用"的含义是多个作业可以交替地启动共享设备，在某一时刻仍只有一个作业占有。

8、存储型设备输入输出操作的信息传输单位是"块"，而输入输出型设备输入输出操作的信息传输单位是"字符"。

9、简述计算机系统的中断机制及其作用。

中断机制包括硬件的中断装置和操作系统的中断处理服务程序。中断装置由一些特定的寄存器和控制线路组成，中央处理器和外围设备等识别到的事件保存在特定的寄存器中，中央处理器每执行完一条指令，均由中断装置判别是否有事件发生。若无事件发生，CPU继续执行；若有事件发生，则中断装置中断原占有CPU的程序的执行，让操作系统的处理事件服务程序占用CPU，对出现的事件进行处理，事件处理完后，再让原来的程序继续占用CPU执行。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*存储管理\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、optimal最优算法是往后看，选最后出现的页面替换

LRU least recently used 最近最少使用算法 是往前看，选最早出现的页面替换。

2、缺页中断：一个页(Page)是一个固定容量的内存区块，是物理内存和外部存储(如硬盘等)传输的单位。当一个程序访问一个映射到地址空间却实际并未加载到物理内存的页（page）时， 硬件向软件发出的一次中断（或异常）就是一个缺页中断或叫页错误（page fault）。

3、抖动 在分页存储管理系统中，内存中只存放了那些经常使用的页面， 而其它页面则存放在外存中，当进程运行需要的内容不在内存时， 便启动磁盘读操作将所需内容调入内存，若内存中没有空闲物理块， 还需要将内存中的某页面置换出去。也就是说，系统需要不断地在内外存之间交换信息。 若在系统运行过程中，刚被淘汰出内存的页面，过后不久又要访问它， 需要再次将其调入。而该页面调入内存后不久又再次被淘汰出内存，然后又要访问它。 如此反复，使得系统把大部分时间用在了页面的调入/换出上， 而几乎不能完成任何有效的工作，这种现象称为抖动。

危害：系统时间消耗在低速的I/O上，大大降低系统效率。进程对当前换出页的每一次访问，与对RAM中页的访问相比，要慢几个数量级。

原因：调度算法不科学，交换算法不科学

解决：好的页替换算法减少运行的进程数

页式、段式、段页式存储分配方式都有可能导致抖动。

4、在请求分页管理中，未被修改的页直接从文件区读入，而被置换时不需要调出；已被修改的页面，被置换时需要调出到交换区，以后从交换区调入。

5、内存的静态分配和动态分配的区别主要是两个：

一是时间不同。静态分配发生在程序编译和连接的时候。动态分配则发生在程序调入和执行的时候。

二是空间不同。堆都是动态分配的，没有静态分配的堆。

解决碎片问题，以及使程序可浮动的最好的办法是采用内存动态分配技术。

6、在段页式存储管理系统中其虚拟地址空间是三维的，段号，页号，页内偏移量。

7、固定分区分配，是满足多道程序设计的存储技术，可以将内存划分成多个区，每个区运行一个作业，从而达到多道程序设计的多个作业同时在内存中的设计。

但是固定分区分配是提前将内存划分成多个区，区的大小可以不同，但是划分好后，个数和大小都不能发生改变。

作业到来，选择一个合适大小的区放置，但是作业长度不一，不会有太合适的区恰好满足其大小，从而会造成区空间的浪费，并且此浪费非常严重。

8、页表项（页描述子)中各个位的作用：页号；块号（页框号）；中断位:用于判断该页是不是在内存中，如果是0，表示该页不在内存中，会引起一个缺页中断；保护位(存取控制位):用于指出该页允许什么类型的访问，如果用一位来标识的话，1表示只读，0表示读写；修改位(脏位):用于页面的换出，如果某个页面被修改过(即为脏),在淘汰该页面时，必须将其返回写回磁盘，反之，可以直接丢弃该页面；访问位：不管是读还是写(read or set),系统都会设置该页面的访问位。他的值会帮助操作系统在发生缺页中断时，选择要被淘汰的页，即用于页面置换；高速缓存禁止位（辅存地址位):对于映射到设备寄存器而不是常规内存的页面来说，这个位很重要；例如：操作系统正在循环等待着某个I/O设备对他的指令做出响应，保证硬件是不断的从设备中读取数据而不是访问一个旧的被高速缓存的副本是非常重要的，通过这一位就可以禁止高速缓存。

9、分段对应的是内存具体存储管理的一种方式，是对具体内存进行管理，段号+基地址，分段尺寸最大为具体内存。在动态链接时先将主程序所对应的目标程序装入内存并启动运行，运行过程中需要调用某段时才将该段内存合并进行链接。而作业的大小不受内存大小限制，由虚拟存储器解决空间不够问题，允许作业装入的时候只装入一部分，另一部分放在 磁盘 上，当需要的时候再装入到主存，这样以来，在一个小的主存空间就可以运行一个比它大的作业。同 时，用户编程的时候也摆脱了一定要编写小于主存容量的作业的限制。

10、在请求分页系统中，只要求将当前需要的一部分页面装入内存，便可以启动作业运行。在作业执行过程中，当所要访问的页面不在内存时，再通过调页功能将其调入，同时还可以通过置换功能将暂时不用的页面换出到外存上，以便腾出内存空间。

为了实现请求分页，系统必须提供一定的硬件支持。除了需要一定容量的内存及外存的计算机系统，还需要有页表机制、缺页中断机构和地址变换机构。请求分页系统的页表机制不同于基本分页系统，请求分页系统在一个作业运行之前不要求全部一次性调入内存，因此在作业的运行过程中，必然会出现要访问的页面不在内存的情况，如何发现和处理这种情况是请求分页系统必须解决的两个基本问题。为此，在请求页表项中增加了四个字段：

增加的四个字段说明如下：

状态位P：用于指示该页是否已调入内存，供程序访问时参考。

访问字段A：用于记录本页在一段时间内被访问的次数，或记录本页最近己有多长时间未被访问，供置换算法换出页面时参考。

修改位M：标识该页在调入内存后是否被修改过。

外存地址：用于指出该页在外存上的地址，通常是物理块号，供调入该页时参考。

在请求分页系统中，每当所要访问的页面不在内存时，便产生一个缺页中断，请求操作系统将所缺的页调入内存。此时应将缺页的进程阻塞（调页完成唤醒)，如果内存中有空闲块，则分配一个块，将要调入的页装入该块，并修改页表中相应页表项，若此时内存中没有空闲块，则要淘汰某页（若被淘汰页在内存期间被修改过，则要将其写回外存)。

11、在可变分区存储管理中的拼接技术可以集中空闲区。

12、通常所说的"存储保护"的基本含义是防止程序间相互越界访问。

13、分段是一组有逻辑意义的信息集合。

分段后，段表包含以下信息：段号+段长+段基地址+存取控制信息

所以段长是可以不固定的，但是每个段内地址是连续的。

这根分页有点区别，分页系统中，每个页面大小是固定的。

14、磁盘和主存可以用来存储页表

15、分页：

分页存储管理器把进程的逻辑地址分成若干页，并为各页加上编号。相应的，内存也会被分成若干个物理块。为了正确的找到页对应的物理块，系统为每个进程添加了一份页表，页表需要的内存空间是连续的（使用多级页表可以缓解这个问题）。页表中存放了对应的物理块号，因此页表是一维的。

分段：

为了方便用户编程，作业的空间可以被分成若干个段，每个段的起始地址都是从0开始的，也就是说，每一段的地址是不连续的。但是段内的地址是连续的。在分段系统中，每个进程也会拥有一个段表，系统为每个段分配连续的内存空间。段表中存放着段长度和段的起始地址。因此，段表是二维的。

16、cpu利用率的瓶颈是硬盘的读写慢：

一，安装 一个更快的硬盘。相当于拓宽公路。因此A肯定没有为题。

二，减少进程数。相当于本来拥堵的公路车采取限号，让整个路更通畅了。这里有一个误区，并不是进程数越多，cpu利用率越高。就好像公路上车越多，单位时间内通过的车就越多一样。也可能情况相反，还不如让车少一点，单位时间内通过的车可能会更多。因此D也是对的。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*计算机网络\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、OSI分层： 五层协议 TCP、IP分层

物理层 物理层 网络接口层 通过媒介传输比特，确定机械以及电气规范（比特Bit）RJ45、IEEE802.3（中继器、集线器）

数据链路层 数据链路层 将比特组装成帧和点到点的传递（帧Frame） MAC、VLAN、FR（网桥、交换机）

网络层 网络层 网络互连层 负责数据包从信源到信宿的传递和网际互联（包Packet）IP、ICMP、ARP、RARP、IGRP（路由器）

传输层 传输层 传输层 提供端到端的可靠报文传递和错误恢复（段Segment）TCP、UDP、SPX

会话层 建立、管理和终止会话（会话协议数据单元SPDU）NFS、SQL

表示层 对数据进行翻译、加密和压缩（表示协议数据单元PPDU）JPEG、MPEG、ASII

应用层 应用层 应用层 允许访问网络环境的手段（应用协议数据单元APDU）FTP、DNS、HTTP、SMTP、Telnet、WWW

2、IP地址的分类

A类地址 以0开头 第一个字节0-127 （1.0.0.0-127.255.255.255）

B类地址 以10开头 第一个字节128-191 （128.0.0.0-191.255.255.255）

C类地址 以110开头 第一个字节192-223 （192.0.0.0-223.255.255.255）

私有地址 10.0.0.0.0-10.255.255.255 172.16.0.0-172.32.255.255 192.168.0.0-192.168.255.255

IP地址与子网掩码相与即得到网络号，主机号全0时表示网络号，全1时表示广播地址

3、ARP的工作原理

首先每个主机会在自己的ARP缓冲区建立一个ARP列表，表示IP和MAC之间的对应关系。然后当源主机要发送数据时，首先检查其ARP列表中是否有对应目的主机的IP和MAC地址，如果有就正常发送数据，否则向本网段的所有主机发送ARP数据包（广播），这个数据包的内容包括源主机的IP和MAC以及目的主机的IP地址。当本网络的所有主机收到这个ARP数据包之后，首先检查其中的目的IP地址是否是自己的IP地址，如果不是就直接忽略，否则用数据包中的源主机IP和MAC更新自己的ARP列表，然后将自己的MAC地址写入ARP响应包中（单播），告诉源主机自己是他要找的MAC地址。源主机收到ARP响应包之后，更新自己的ARP列表，然后就可以发送数据了。如果源主机一直没收到ARP响应数据包，表示查询失败。

RARP是逆地址解析协议，作用是完成硬件地址到IP地址的映射，主要用于无盘工作站，因为给无盘工作站配置的IP地址不能保存。工作流程：在网络中配置一台RARP服务器，里面保存着IP地址和MAC地址的映射关系，当无盘工作站启动后，就封装一个RARP数据包，里面有其MAC地址，然后广播到网络上去，当服务器收到请求包后，就查找对应的MAC地址的IP地址装入响应报文中发回给请求者。因为需要广播请求报文，因此RARP只能用于具有广播能力的网络。

4、一些常见协议的介绍

ICMP因特网控制报文协议，是TCP|IP协议族的一个子协议，用于在IP主机、路由器之间传递控制消息

HTTP超文本传输协议，应用层的面向对象的协议

DHCP动态主机配置协议，给内部网络或网络服务供应商自动分配IP地址

NAT网络地址转换，将私有地址转化为合法IP地址

5、TCP三次握手

客户端 服务器

发送syn包（syn=x），进入SYN\_SEND状态

收到syn包，确认客户的SYN（ack=x+1）,同时发送一个SYN包（syn=y）

即SYN+ACK包，进入SYN\_RECV状态

收到SYN+ACK包，发送确认包ACK(ack=y+1),发送完毕

后二者进入ESTABLISHED状态，完成三次握手

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去

TCP的三次握手最主要是防止已过期的连接再次传到被连接的主机。如果采用两次的话，会出现下面这种情况。比如是A机要连到B机，结果发送的连接信息由于某种原因没有到达B机；

于是，A机又发了一次，结果这次B收到了，于是就发信息回来，两机就连接。传完东西后，断开。结果这时候，原先没有到达的连接信息突然又传到了B机，于是B机发信息给A，然后B机就以为和A连上了，这个时候B机就在等待A传东西过去。浪费服务器资源。

四次挥手

发送FIN，关闭到被动关闭方的数据传送“我已经不

会再给你发送数据了！”，此时还能接收数据（ESTABLISHED建立状态-FIN\_WAIT1终止等待）

收到FIN，发送ACK，确认序号为收到序号+1（ESTABLISHED-CLOSE\_WAIT关闭等待）

发送FIN，关闭被动关闭方到主动关闭方的数据传送“我的数据也发送完 了！”（LAST\_ACK-CLOSED）

收到FIN，发送ACK，确认序号为收到序号+1，完成（FIN\_WAIT2-TIME\_WAIT-CLOSED）

四次挥手

6、在浏览器输入www.baidu.com之后的全部过程

客户端浏览器通过DNS解析到www.baidu.com的IP地址，于是浏览器发起一个HTTP会话到该地址，然后通过TCP进行封装数据包，这里是将HTTP会话请求分成报文段，添加源和目的端口，数据包输入网络层后，查找路由表确定如何到达服务器，其间可能经过多个路由器，在链路层，则通过ARP查找IP地址对应的MAC地址。

7、TCP和UDP的区别

TCP提供面向连接的可靠数据流传输，注重数据安全性，其传输单位为TCP报文段，对应的协议有FTP（文件传输协议，port21）Telnet（远程登录，port23）SMTP（邮件传送，port25）POP3（接收邮件，port110）HTTP

UDP提供无连接的不可靠的数据流传输，传输快，安全性一般，传输单位是用户数据报，对应的协议有DNS（域名解析服务，将域名地址转换为IP地址，port53）TFTP（简单文件传输协议，port69）

8、面向连接和非面向连接的服务的特点是什么？

面向连接的服务，通信双方在进行通信之前，要先在双方建立起一个完整的可以彼此沟通的通道，在通信过程中，整个连接的情况一直可以被实时地监控和管理。

非面向连接的服务，不需要预先建立一个联络两个通信节点的连接，需要通信的时候，发送节点就可以往网络上发送信息，让信息自主地在网络上去传，一般在传输的过程中不再加以监控

9、交换机工作在数据链路层

路由器工作在网络层，一个作用是连通不同的网络，另一个作用是选择信息传送的线路

网关经常用于连接局域网和internet，也指把一种协议转成另一种协议的设备，其最重要的一点就是端口映射，子网内用户在外网看来只是外网的IP地址对应着不同的端口，就可以保护子网内的用户

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*C++\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、当类中包括指针类成员变量时，一定要重载其拷贝构造函数、赋值函数和析构函数，这是对C++程序员的基本要求

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*字符串\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、strcpy会从源地址一直往后拷贝，直到遇到'\0'为止。所以拷贝的长度是不定的。如果一直没有遇到'\0'导致越界访问非法内存，程序就崩了。所以一定要注意数组或者字符串的长度问题

2、写出完整版的strcpy函数,为了实现链式操作，将目的地址返回,是加分项

char \* strcpy( char \*strDest, const char \*strSrc )

{

assert( (strDest != NULL) && (strSrc != NULL) );

char \*address = strDest;

while( (\*strDest++ = \* strSrc++) != ‘\0’ );

return address;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*数组\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、数组名做函数形参时，退化为指针，可以自增自减。平时的话则有

char str[10];

sizeof(str)=10;

2016/03/13补充(整理到了第八页)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*进程和线程\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、进程和应用程序的区别

进程（Process）是最初定义在Unix等多用户、多任务操作系统环境下用于表示应用程序在内存环境中基本执行单元的概念。以Unix操作系统为例，进程是Unix操作系统环境中的基本成分、是系统资源分配的基本单位。Unix操作系统中完成的几乎所有用户管理和资源分配等工作都是通过操作系统对应用程序进程的控制来实现的。

C、C++、Java等语言编写的源程序经相应的编译器编译成可执行文件后，提交给计算机处理器运行。这时，处在可执行状态中的应用程序称为进程。从用户角度来看，进程是应用程序的一个执行过程。从操作系统核心角度来看，进程代表的是操作系统分配的内存、CPU时间片等资源的基本单位，是为正在运行的程序提供的运行环境。进程与应用程序的区别在于应用程序作为一个静态文件存储在计算机系统的硬盘等存储空间中，而进程则是处于动态条件下由操作系统维护的系统资源管理实体。

多任务环境下应用程序进程的主要特点包括：

●进程在执行过程中有内存单元的初始入口点，并且进程存活过程中始终拥有独立的内存地址空间；

●进程的生存期状态包括创建、就绪、运行、阻塞和死亡等类型；

●从应用程序进程在执行过程中向CPU发出的运行指令形式不同，可以将进程的状态分为用户态和核心态。处于用户态下的进程执行的是应用程序指令、处于核心态下的应用程序进程执行的是操作系统指令。

在Unix操作系统启动过程中，系统自动创建swapper、init等系统进程，用于管理内存资源以及对用户进程进行调度等。在Unix环境下无论是由操作系统创建的进程还要由应用程序执行创建的进程，均拥有唯一的进程标识（PID）

2、进程：一个程序在一个数据集合上的一次运行过程。所以一个程序在不同数据集合上运行，乃至一个程序在同样数据集合上的多次运行都是不同的进程

进程间的通信可以用管道、消息队列、共享内存、信号、信号量

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*数学基础\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、五种基本关系代数运算

(1)并(∪)：两个关系需有相同的关系模式，并的对象是元组，由两个关系所有元组构成。

RUS≡{t| t∈R ∨t∈S}

(2) 差(-)：同样，两个关系有相同的模式，R和S的差是由属于R但不属于S的元组构成的集合。

R-S≡{t| t∈R ∧t 不属于S}

(3)笛卡尔积（×）：对两个关系R和S进行操作，产生的关系中元组个数为两个关系中元组个数之积。

R×S≡{t| t=< tr,ts>∧tr∈R∧ts ∈S}

(4) 投影(σ)：对关系进行垂直分割，消去某些列，并重新安排列的顺序。

(5) 选择(π)：根据某些条件关系作水平分割，即选择符合条件的元组。

2、泊松分布的参数λ是单位时间(或单位面积)内随机事件的平均发生率。 泊松分布适合于描述单位时间内随机事件发生的次数。

3、现在的数列题可能都会带有指数，底数也可能为1

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*编程基础\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、SAX解析xml文件的优点

优点：不用事先调入整个文档，占用资源少。尤其在嵌入式环境，如Android,极力推荐采用SAX进行解析。不需要像dom解析那样在内存中建立一个dom对象，占用内存，sax解析是逐行解析的，每次读入内存的只是一行xml，所以速度快，效率高点。不过sax一般是处理固定格式的xml。

缺点：不像DOM一样将文档树长期留驻在内存，数据不是长久的。事件过后，若没保存数据，那么数据就会丢失。

使用场合：机器有性能限制，尤其是在嵌入式环境。

2、后缀表示式主要是为了便于计算，规则：从左到右遍历中缀表达式的每个数字和符号，如果是数字就输出，即成为后缀表达式的一部分，如果是符号，就判断它和栈顶符号的优先级，是右括号或优先级【不高于】栈顶符号（乘除优先加减）则栈顶符号依次出栈并输出，并将当前符号进栈，一直到最终输出后缀表达式为止

3、窗口句柄就相当于一个身份证，是一个整数，如果两个窗口的句柄相同的话，就会出现消息不知道传给哪个窗口了。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*设计模式\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、数据访问对象 DAO模式

解决问题：根据数据源不同，数据访问也不同。根据存储的类型（关系数据库、面向对象数据库、纯文件等）和供应商实现不同，持久性存储（如数据库）的访问差别也很大。如何对存储层以外的模块屏蔽这些复杂性，以提供统一的调用存储实现。程序的分布式问题 解决方案：将数据访问逻辑抽象为特殊的资源，也就是说将系统资源的接口从其底层访问机制中隔离出来；通过将数据访问的调用打包，数据访问对象可以促进对于不同数据库类型和模式的数据访问。 本质：分层，是系统组件和数据源中间的适配器。（一层屏蔽一种变化）

2、外观模式：为子系统中的一组接口提供一个一致的界面，此模式定义了一个高层接口，这个接口使得这一子系统更加容易调用——类比基金和普通炒股的差距

组合模式：将对象组合成树形结构以表示‘部分-整体’的层次结构，组合模式使得用户对单个对象和组合对象的使用具有一致性——类比公司和各地办事处架构

单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点

**class** **Foo** {  
 **private** Helper helper = **null**;  
 **public** Helper getHelper() {  
 **if** (helper == **null**) {  
 **synchronized**(**this**) {  
 **if** (helper == **null**) {  
 helper = **new** Helper();  
 }  
 }  
 }  
 **return** helper;  
 }

}

桥接模式：将抽象部分和实现部分分离，使他们都可以独立变化——类比手机品牌与手机功能

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*软件工程\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、快速原型模型 （这些模型大部分都是在需求阶段考虑的问题）

快速原型模型需要迅速建造一个可以运行的软件原型 ，以便理解和澄清问题，使开发人员与用户达成共识，最终在确定的客户需求基础上开发客户满意的软件产品。 快速原型模型允许在 需求分析 阶段对软件的需求进行初步而非完全的分析和定义，快速设计开发出软件系统的原型，该原型向用户展示待开发软件的全部或部分功能和性能；用户对该原型进行测试评定，给出具体改进意见以丰富细化 软件需求 ；开发人员据此对软件进行修改完善，直至用户满意认可之后，进行软件的完整实现及测试、维护。

2、LOC（ lines of code）是软件规模的一种度量

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*计算机网络\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、192.168.1.110属于C类IP，C类IP前24位表示网络号，后8位表示主机号。

/27说明网络掩码的长度是27位的1和32-27=5位的0组成。也就是说IP地址192.168.1.110前24位表示网络号，25到27位表示子网号，剩下5位表示主机号。

第25到27位也就是最后一个字节的前三位，最后一字节110写成二进制的为01101110

其前三位是二进制1011，所以接下来就是在选项中找出最后一字节前三位二进制是011的ip地址

94=0101 1110

96=0110 0000

124=0111 1100

126=0111 1110

BCD选项都符合前三位是011，但是B选项除去011之后剩下全零，不能作为主机号

2、HTTPS：http://kingj.iteye.com/blog/2103662

用到了非对称加密技术、对称加密技术、哈希算法、数字证书

3、 MSL 是Maximum Segment Lifetime英文的缩写，中文可以译为“报文最大生存时间”，他是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。因为tcp报文 （segment）是ip数据报（datagram）的数据部分，具体称谓请参见《数据在网络各层中的称呼》一文；

ip头中有一个TTL域， TTL是 time to live的缩写，中文可以译为“生存时间”，这个生存时间是由源主机设置初始值但不是存的具体时间，而是存储了一个ip数据报可以经过的最大路由数，每经 过一个处理他的路由器此值就减1，当此值为0则数据报将被丢弃，同时发送ICMP报文通知源主机。RFC 793中规定MSL为2分钟，实际应用中常用的是30秒，1分钟和2分钟等。

TTL与MSL是有关系的但不是简单的相等的关系，MSL要大于等于TTL。

3、 RTT是客户到服务器往返所花时间（round-trip time，简称RTT），TCP含有动态估算RTT的算法。TCP还持续估算一个给定连接的RTT，这是因为RTT受网络传输拥塞程序的变化而变化

4 、2MSL即两倍的MSL，TCP的TIME\_WAIT状态也称为2MSL等待状态，当TCP的一端发起主动关闭，在发出最后一个ACK包后，即第3次握 手完成后发送了第四次握手的ACK包后就进入了TIME\_WAIT状态，必须在此状态上停留两倍的MSL时间，等待2MSL时间主要目的是怕最后一个 ACK包对方没收到，那么对方在超时后将重发第三次握手的FIN包，主动关闭端接到重发的FIN包后可以再发一个ACK应答包。在TIME\_WAIT状态 时两端的端口不能使用，要等到2MSL时间结束才可继续使用。当连接处于2MSL等待阶段时任何迟到的报文段都将被丢弃。不过在实际应用中可以通过设置 SO\_REUSEADDR选项达到不必等待2MSL时间结束再使用此端口。

5、在TCP层，有个FLAGS字段，这个字段有以下几个标识：SYN, FIN, ACK, PSH, RST, URG.

其中，对于我们日常的分析有用的就是前面的五个字段。

它们的含义是：

URG:Urget pointer is valid (紧急指针字段值有效)

SYN: 表示建立连接

FIN: 表示关闭连接

ACK: 表示响应

PSH: 表示有 DATA数据传输

RST: 表示连接重置。

其中，ACK是可能与SYN，FIN等同时使用的，比如SYN和ACK可能同时为1，它表示的就是建立连接之后的响应，如果只是单个的一个SYN，它表 示的只是建立连接。TCP的几次握手就是通过这样的ACK表现出来的。但SYN与FIN是不会同时为1的，因为前者表示的是建立连接，而后者表示的是断开 连接。RST一般是在FIN之后才会出现为1的情况，表示的是连接重置。一般地，当出现FIN包或RST包时，我们便认为客户端与服务器端断开了连接；而 当出现SYN和SYN＋ACK包时，我们认为客户端与服务器建立了一个连接。PSH为1的情况，一般只出现在 DATA内容不为0的包中，也就是说PSH为1表示的是有真正的TCP数据包内容被传递。

6、回送地址127.x.x.x

7、FTP端口号21 TELNET（远程登录协议、应用层） 23 DNS 53 HTTP 80 mysql 3306 tomcat 8080

8、平时说的端口带宽就是半双工带宽

9、OSI七层模型

对等实体在一次交互作用中传送的信息单位称为协议数据单元，包括控制信息和用户数据两部分

上下层实体之间的接口称为服务访问点SAP，网络层的SAP也成为网络地址，分为网络号和主机地址两部分

物理层：透明的传输比特流

数据链路层

网络层，路由，IP，ARP（由IP得到MAC，主机A就会在网络上发送一个广播（ARP request），目标MAC地址是“FF.FF.FF.FF.FF.FF”，这表示向同一网段内的所有主机发出这样的询问：“192.168.38.11的MAC地址是什么？”网络上其他主机并不响应ARP询问，只有主机B接收到这个帧时，才向主机A做出这样的回应（ARP response）：“192.168.38.11的MAC地址是（00-BB-00-62-C2-02）”。这样，主机A就知道主机B的MAC地址，它就可以向主机B发送信息。同时它还更新自己的ARP缓存表，下次再向主机B发送信息时，直接从ARP缓存表里查找就可。ARP缓存表采用老化机制，在一段时间内如果表中的某一行没有使用，就会被删除，这样可以大大减少ARP缓存表的长度，加快查询速度）

传输层，TCP

会话层

表示层

应用层，HTTP

10、HTTP状态码

1xx消息 100 continue

2xx成功206 partial content

3xx重定向

4xx客户端错误403 forbidden 404 not found

5xx服务器错误502 bad gateway

11、socket的过程

socket相当于进行网络通信两端的插座，只要对方的socket和自己的socket有通信连接，双方就可以发送和接收数据了

TCP情况下

服务端

（1）socket()创建一个套接字

（2）bind()绑定ip地址和端口 listem开始监听

（3）循环调用accept（）接受连接，对于每个接受的连接，可以启动多线程进行处理，在线程中调用send recv发送和接收数据 输出流和输入流分别用socket类的getoutstream和getinputstream

（4）关闭套接字close（）

客户端

（1）socket创建套接字

（2）connect连接服务器

（3）send recv发送和接受数据，输出流和输入流同上

（4）关闭close

UDP情况下，服务器端没有listen，循环recvfrom，客户端用sendto

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*数据结构与算法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、循环队列，可用数组C[1..m]表示，m为数组长度，假设f为队首指针（队头元素在数组中的位置），r为队尾指针（队尾元素的后一位置），则队列中元素个数为（r-f+m）mod m

2、插入排序的原理：始终定义第一个元素为有序的，将元素逐个插入到有序排列之中，其特点是要不断的移动数据，空出一个适当的位置，把待插入的元素放到里面去

选择排序的原理：每次在无序队列中“选择”出最小值，放到有序队列的最后，并从无序队列中去除该值（具体实现略有区别）

3、稀疏矩阵

设m\*n 矩阵中有t 个非零元素且t<<m\*n，这样的矩阵称为稀疏矩阵。很多科学管理及工程计算中，常会遇到阶数很高的大型稀疏矩阵。如果按常规分配方法，顺序分配在计算机内，相当浪费内存的。

压缩的存储方法是

三元组

为此提出另外一种存储方法，仅仅存放非零元素。但对于这类矩阵，通常零元素分布没有规律，为了能找到相应的元素，所以仅存储非零元素的值是不够的，还要记下它所在的行和列。于是采取如下方法：将非零元素所在的行、列以及它的值构成一个三元组（i,j,v），然后再按某种规律存储这些三元组，这种方法可以节约存储空间。

或十字链表

| row | col | value |

| down | right |

down指向同列下一个节点，right指向同行下一个节点，这种方式可以在矩阵计算时避免大量移动元素

4、几种常见数据结构的操作性能对比

查找 插入 删除 遍历

数组 O(N) O(1) O(N)

有序数组 O(logn) O(N) O(N) O(N)

链表 O(N) O(1) O(N)

有序链表 O(N) O(N) O(N) O(N)

二叉树（一般情况） O(logn) O(logn) O(logn) O(N)

二叉树（最坏情况） O(N) O(N) O(N) O(N)

平衡树 O(logn) O(logn) O(logn) O(N)

哈希表 O(1) O(1) O(1)

5、非空字符串的子串数目为1+（1+n）\*n/2

6、KMP算法下长为n的字符串中匹配长度为m的子串的复杂度为O(M+N)

next数组可以这样求，第一个是0，第二个是1，这个是确定好了的，之后从第3个开始可以这样看，看第3个的前面的字符串的前缀子串和后缀子串有几个字符相等，这里的话前缀为a 后缀为a，一个相等，则值为1+1=2，第4个，前缀aa 后缀aa两个相等，next值为2+1=3，我们可以得出这样一个结论，如果前后缀一个字符相等，则next值为2，两个字符相等则为3，n个相等next值为n+1，就是要看前缀和后缀子串最大能有多少个字符相同，next为其最大值+1

7、冒泡排序 O(n^2) O(1) 稳定,最好O(n)

快速排序 O(nlogn) O(nlogn) 不稳定 平均比较次数最少。当所有元素已经有序时，最差为n^2 适用于相对无序的情况

堆排序 O(nlogn) O(1) 不稳定 无论什么时候都是nlogn 适用于基本有序的情况，由于初始构建堆所需的比较次数较多，并不适合待排序序列个数较少的情况

直接插入排序 O(n^2) O(1) 稳定 平均n^2 最好为n

希尔排序 O(n(logn)^2) O(1) 不稳定 最坏情况下是O(n^x)(1<x<2)

归并排序 O(nlogn) O(n)+O(logn)（改为迭代方法后可以减少为O(n)） 稳定

8、循环链表，就是将一个链表的尾元素指针指向队首元素，p->link=head

9、前序遍历：根左右，先访问根节点，然后前序遍历左子树再前序遍历右子树

中序遍历：左根右，中序遍历根节点的左子树，访问根节点，再中序遍历右子树

后序遍历：左右根，从左到右先叶子后节点的方式遍历访问左右子树，最后访问根节点

10、线性探测法：处理散列冲突的方法，f(key)=(f(key)+d)MOD m,开放定址法的一种

11、二叉搜索树

要么是一颗空树，要么具有如下性质

若左子树非空，则左子树上所有节点的值均小于其根结构的值

若右子树非空，则右子树上所有节点的值军大于其根节点的值

左右子树都是二叉搜索树

搜索、插入、删除的复杂度都是logn，这是在平衡的前提下的

引申出了平衡二叉树的概念AVL

是一种二叉排序树，其中每一个节点的左子树和右子树的高度差至多等于1

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*C/C++\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、vector在erase之后，指向下一个元素的位置，其实进行erase操作时将后面所有元素都向前移动，迭代器位置没有移动。itor=array.erase(itor) erase返回下一个元素的地址，相当于给itor一个新值。

2、C语言中输入输出函数占位符介绍 :

占位符 数据类型

%d int

%ld long int

%c char

%f float

&lf double

%x 十六进制

%O 八进制

%s 字符串

3、C++的多态肯定是使用父类的指针指向子类的对象，所以肯定是释放子类的对象，如果不使用虚函数的话，父类的指针就只能够释放父类的对象。

4、typedef char \*String\_t 定义了一个新的类型别名，有类型检查。而#define String\_d char \* 只是做了个简单的替换，无类型检查，前者在编译的时候处理，后者在预编译的时候处理。

同时定义多个变量的时候有区别主要区 在于这种使用方式String\_t a,b; String\_d c,d; a,b ,c 都是 char\*类型，而d为char类型

由于 typedef 还要做类型检查#define 没有所以 typedef 比#define 安全

5、多重继承：每个父类都有一个虚表，子类的虚函数放在第一个父类虚表的最后

6、模板是泛型编程的基础，体现了一种编译时多态性或参数式多态性。

模板定义以template<typename T1, typename T2>,尖括号内的东西成为模板形参，不能为空。class和typename都是关键字，没有区别。

template<typename T> inline T min(const T&, const T&)

除了定义函数模板外，也可以定义类模板

template <class Type> class Queue{

public:

Queue();

Type& front();

const Type& front() const;

void push(const Type&);

bool empty() const;

……

}

使用时对应的有Queue<int> qi

7、拷贝构造函数， B b = A 肯定是可以的。

当类的一个数据成员是指针或有成员表示在构造函数中分配的其他资源，而另一些类在创建新对象时必须做一些特定工作的时候，需要拷贝构造函数。

拷贝构造函数用于初始化、函数形参、返回时复制

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*数据库\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、BEGIN开始事务处理 COMMIT保存更改 ROLLBACK回滚

2、创建视图（虚表）

CREATE [TEMP | TEMPORARY] VIEW view\_name AS

SELECT column1, column2.....

FROM table\_name

WHERE [condition];

删除视图DROP VIEW view\_name

3、创建表、删除表CREATE TABLE；DROP TABLE

4、关系数据库的优点：数据冗余度小；独立性高；共享性好

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Linux\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、cat 查看文件内容

chmod 001执行权限 010 写入权限 100读取权限

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*文件系统\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、文件的逻辑结构是从用户观点出发看到的文件的组织形式。文件的物理结构是从实现观点出发，又称为文件的存储结构，是指文件在外存上的存储组织形式。文件的逻辑结构与存储介质特性无关，但文件的物理结构与存储介质的特性有很大关系。

按逻辑结构，文件有无结构文件和有结构文件两种类型：无结构文件和有结构文件。

有结构文件按记录的组织形式可以分为：

1) 顺序文件。2) 索引文件。3) 索引顺序文件。4) 直接文件或散列文件(Hash File)

常见的文件物理结构有以下几种：1、顺序结构又称连续结构。2、链接结构3、索引结构4、Hash结构又称杂凑结构或散列结构。5、索引顺序结构索引表每一项在磁盘上按顺序连续存放在物理块中。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*操作系统\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、Cache（高速缓冲器）容量小于主存，但速度快于主存，慢于CPU，相当于CPU和主存间的一个缓冲器，Cache中存放最近使用过的内存内容（基于最近使用过的内容很可能被再次使用的原理）。若CPU寻访的内容在Cache中存放，则优先从Cache中读取，称为命中，否则称为脱靶，脱靶只能从主存中读取内容了。当Cache存储满的时候，用替换算法清理掉不用的内容，保留下最新或最常使用的内容，称为替换。Cache设计目标是提高命中率。替换算法确实是影响Cache命中率，但还有Cache容量、存储单元大小、组数多少、地址比较方法、写操作方法等都会影响Cache命中率。

2、磁盘会一直朝某个方向旋转，不会因为处理数据而停止，且磁盘不会反向旋转，此题重点在于要按顺序，所以总要转回来

3、

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*算法编程题\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、求最长重复子串

#include <iostream>

2 #include <string>

3 #include <algorithm>

4 #include <vector>

5

6 using namespace std;

7

8 class LongestCommonString

9 {

10 vector<string> suffixArray;

11 size\_t len;

12 public:

13

14 //构造方法

15 LongestCommonString(string s)

16 {

17 //构造后缀数组

18 for(size\_t i = 0; i < s.length(); ++i)

19 {

20 suffixArray.push\_back(s.substr(i));

21 }

22 //排序

23 sort(suffixArray.begin(), suffixArray.end());

24 len = suffixArray.size();

25 }

26

27 //两两比较，返回最长长度的子串

28 string lcpCompare()

29 {

30 size\_t maxLength = 0;

31 size\_t index = 0;

32 for(size\_t i = 0; i < len - 1; ++i)

33 {

34 size\_t temp = lcp(suffixArray.at(i), suffixArray.at(i+1));

35 if(temp > maxLength)

36 {

37 maxLength = temp;

38 index = i;

39 }

40 }

41 return suffixArray.at(index).substr(0, maxLength);

42 }

43

44 //返回重复的长度

45 size\_t lcp(const string& a, const string& b)

46 {

47 size\_t i = 0;

48 while(a[i] == b[i])

49 {

50 ++i;

51 }

52 return i;

53 }

54 };

55

56 int main()

57 {

58 string h = "hello";

59 //cout << h.substr(2, 3);

60 string word = "aacaagmtttacaagmc";

61 LongestCommonString slcp(word);

62 cout << "最长重复子串为：" << slcp.lcpCompare() << endl;

63 }

2、快速排序 时间复杂度 nlogn 最坏情况下是n^2 空间复杂度logn 最坏情况下是n

代码

void quicksort(List\* L)

{

qsort(L,1,length);

}

qsort(List\* L, int low, int high)

{

int pivot;

if(low<high)

{

pivot=partition(L,low,high);//将List一分为二，左边的都小于pivot指向的值，后边的都大于

qsort(L,low,pivot-1);//对低表递归排序

qsort(L,pivot+1,high);//对高表递归排序

}

}

int partition(List\* L, int low, int high)

{

int pivotkey=L->r[low];//用子表的第一个记录作为枢纽记录

while(low<high)//从两端交替向中间扫描

{

while(low<high && L->r[high]>=pivotkey)

high--;

swap(L,low,high);//将比枢纽小的值交换到低表

while(low<high && L->r[low]<=pivotkey)

low++;

swap(L,low,high);//将比枢纽大的值交换到高表

}

return low;//返回枢纽所在位置

}

3、实现strlen

int stringlength(const char\* str)

{

int length = 0;

if (str == NULL)

return -1;

while (\*(str++) != '\0')

length++;

return length;

}

4、实现strncpy

char\* stringncopy(const char\* src,char\* dst, int count)

{

assert(src != NULL && dst != NULL);

while (\*src != '\0' && count != 0)

{

\*dst++ = \*src++;

count--;

}

\*dst = '\0';

return dst;

}

20160316补充

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*计算机网络\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、网路上两个主机互相发信息的过程

产生数据，传给路由器，路由器根据路由表和路由算法，确定该把这个包送向哪里，然后送过去，一点点的送到接受主机

2、TCP如何保证传输的可靠性

发送确认；超时TCP重传

在开始传送一个段时，为准备重传而首先将该段插入到发送队列之中，同时启动时钟。其后，如果收到了接受端对该段的ACK信息，就将该段从队列中删去。如果在时钟规定的时间内，ACK未返回，那么就从发送队列中再次送出这个段。TCP在协议中就对数据可靠传输做了保障，握手与断开都需要通讯双方确认，数据传输也需要双方确认成功

3、

滑动窗口TCP头部的window字段

滑动窗口协议是传输层进行流控的一种措施，接收方通过通告发送方自己的窗口大小，从而控制发送方的发送速度，从而达到防止发送方发送速度过快而导致自己被淹没的目的。TCP的滑动窗口解决了端到端的流量控制问题，允许接受方对传输进行限制，直到它拥有足够的缓冲空间来容纳更多的数据。4、

拥塞窗口

拥塞窗口也看做是发送端用来进行流量控制的窗口。

但是，实际上，TCP还必须应付互联网中的拥塞现象。拥塞是指一个或者多个交换点的数据报超载而导致时延剧烈增加的现象。为了控制拥塞，TCP使用了第二个窗口限制，即拥塞窗口限制。对于拥塞窗口大小的限制采用慢开始和乘法减小两种技术。

乘法减小的拥塞避免策略：一旦发现报文段丢失，就把拥塞窗口的大小减半（直到减至最小的窗口，窗口中应至少包含一个报文段）。

慢开始恢复：拥塞窗口随着一个确认的到达，拥塞窗口的大小每次增加一个报文段。

慢启动为发送方的TCP增加了另一个窗口：拥塞窗口(congestion window)，记为cwnd。当与另一个网络的主机建立TCP连接时，拥塞窗口被初始化为1个报文段（即另一端通告的报文段大小）。每收到一个ACK，拥塞窗口就增加一个

报文段（cwnd以字节为单位，但是慢启动以报文段大小为单位进行增加）。发送方取拥塞窗口与通告/滑动窗口中的最小值作为发送上限。拥塞窗口是发送方使用的流量控制，而通告/滑动窗口则是接收方使用的流量控制。

发送方开始时发送一个报文段，然后等待ACK。当收到该ACK时，拥塞窗口从1增加为2，即可以发送两个报文段。

当收到这两个报文段的ACK时，拥塞窗口就增加为4。这是一种指数增加的关系。

4、首先，服务器端启动进程，调用Socket创建一个基于TCP协议的流套接字描述符。

其次，服务进程调用bind命名套接字，将套接字描述符绑定到本地地址和本地端口上。

再次，服务器端调用listen，开始侦听客户端的Socket连接请求。

接下来阻塞，直到收到了客户端的connect请求，调用accept进行相应。

因此，不阻塞bind和listen

5、互联网的网络地址分为A~E五类，其中

A类地址：0.0.0.0 ~ 127.255.255.255

主机号是后24位

B类地址：128.0.0.0 ~ 191.255.255.255

主机号是后16位

C类地址：192.0.0.0 ~ 223.255.255.255

主机号是后8位

D类地址：224.0.0.0 ~ 239.255.255.255

后28位为多播组号

E类地址：240.0.0.0 ~ 255.255.255.255

后27位待用

我们能使用的只有A~C类，D类为多播地址，E类保留使用

主机号全为0的时候，表示一个网段

主机号全为1的时候，是一个指向网络的广播。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*C/C++\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、多态：函数、运算符重载，编译时的多态性；类的多态，模板（静态），虚函数，动态多态

2、类指针的声明，是不会调用构造函数的；但是指向一个类实例（new）就会调用构造函数。 但是类的声明，会调用默认构造函数

3、用“w”打开的文件只能向该文件写入。 若打开的文件不存在，则以指定的文件名建立该文件，若打开的文件已经存在，则将该文件删去，重建一个新文件。"w+"先建立文件并进行写操作后，"w+"可以从头开始读

4、重载运算符必须和用户定义的自定义类型的对象一起使用。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*编程基础\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、线程安全是编程中的术语，指某个函数、函数库在多线程环境中被调用时，能够正确地处理各个线程的局部变量，使程序功能正确完成。正确处理多个线程共享的变量，比如加mutex

2、静态库会将汇编生成的目标文件.o与引用到的库一起链接打包到可执行文件中。因此对应的链接方式称为静态链接。静态库与汇编生成的目标文件一起链接为可执行文件，那么静态库必定跟.o文件格式相似。其实一个静态库可以简单看成是一组目标文件（.o/.obj文件）的集合，即很多目标文件经过压缩打包后形成的一个文件。静态库特点总结：静态库对函数库的链接是放在编译时期完成的。程序在运行时与函数库再无瓜葛，移植方便。浪费空间和资源，因为所有相关的目标文件与牵涉到的函数库被链接合成一个可执行文件。

动态库在程序编译时并不会被连接到目标代码中，而是在程序运行是才被载入。不同的应用程序如果调用相同的库，那么在内存里只需要有一份该共享库的实例，规避了空间浪费问题。动态库在程序运行是才被载入，也解决了静态库对程序的更新、部署和发布页会带来麻烦。用户只需要更新动态库即可，增量更新。动态库特点总结：动态库把对一些库函数的链接载入推迟到程序运行的时期。实现进程之间的资源共享。（因此动态库也称为共享库）；将一些程序升级变得简单。甚至可以真正做到链接载入完全由程序员在程序代码中控制（显示调用）。

3、同步是指：发送方发出数据后，等接收方发回响应以后才发下一个数据包的通讯方式。

异步是指：发送方发出数据后，不等接收方发回响应，接着发送下个数据包的通讯方式。

阻塞

阻塞调用是指调用结果返回之前，当前线程会被挂起。函数只有在得到结果之后才会返回。有人也许会把阻塞调用和同步调用等同起来，实际上他是不同的。对于同步调用来说，很多时候当前线程还是激活的，只是从逻辑上当前函数没有返回而已。例如，我们在CSocket中调用Receive函数，如果缓冲区中没有数据，这个函数就会一直等待，直到有数据才返回。而此时，当前线程还会继续处理各种各样的消息。如果主窗口和调用函数在同一个线程中，除非你在特殊的界面操作函数中调用，其实主界面还是应该可以刷新。socket接收数据的另外一个函数recv则是一个阻塞调用的例子。当socket工作在阻塞模式的时候，如果没有数据的情况下调用该函数，则当前线程就会被挂起，直到有数据为止。

非阻塞

非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*进程和线程\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、单个CPU一次只能运行一个任务。进程就好比工厂的车间，它代表CPU所能处理的单个任务。任一时刻，CPU总是运行一个进程，其他进程处于非运行状态。线程就好比车间里的工人。一个进程可以包括多个线程。一个进程的内存空间是共享的，每个线程都可以使用这些共享内存。一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。一个防止他人进入的简单方法，就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。这就叫"互斥锁"（Mutual exclusion，缩写Mutex），防止多个线程同时读写某一块内存区域。某些内存区域，只能供给固定数目的线程使用。这时的解决方法，就是在门口挂n把钥匙。进去的人就取一把钥匙，出来时再把钥匙挂回原处。后到的人发现钥匙架空了，就知道必须在门口排队等着了。这种做法叫做"信号量"（Semaphore），用来保证多个线程不会互相冲突。

不难看出，mutex是semaphore的一种特殊情况（n=1时）。也就是说，完全可以用后者替代前者。但是，因为mutex较为简单，且效率高，所以在必须保证资源独占的情况下，还是采用这种设计。操作系统的设计，因此可以归结为三点：

（1）以多进程形式，允许多个任务同时运行；

（2）以多线程形式，允许单个任务分成不同的部分运行；

（3）提供协调机制，一方面防止进程之间和线程之间产生冲突，另一方面允许进程之间和线程之间共享资源。

2、守护进程（daemon）是指在UNIX或其他多任务操作系统中在后台执行的电脑程序，并不会接受电脑用户的直接操控。此类程序会被以进程的形式初始化。守护进程程序的名称通常以字母“d”结尾：例如，syslogd就是指管理系统日志的守护进程。

孤儿进程指的是在其父进程执行完成或被终止后仍继续运行的一类进程。用户也可能会刻意使进程成为孤儿进程，以使之与用户会话脱钩，并转至后台运行。这一做法常应用于启动需要长时间运行的进程，也即守护进程

僵尸进程是指完成执行（通过exit系统调用，或运行时发生致命错误或收到终止信号所致）但在操作系统的进程表中仍然有一个表项（进程控制块PCB），处于"终止状态"的进程。这发生于子进程需要保留表项以允许其父进程读取子进程的exit status：一旦退出态通过wait系统调用读取，僵尸进程条目就从进程表中删除，称之为"回收（reaped）"。正常情况下，进程直接被其父进程wait并由系统回收。进程长时间保持僵尸状态一般是错误的并导致资源泄漏。

3、进程间的通信

管道：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。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。

FIFO：也称为命名管道，它是一种文件类型。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。FIFO的通信方式类似于在进程中使用文件来传输数据，只不过FIFO类型文件同时具有管道的特性。在数据读出时，FIFO管道中同时清除数据，并且“先进先出”。FIFO的通信方式类似于在进程中使用文件来传输数据，只不过FIFO类型文件同时具有管道的特性。在数据读出时，FIFO管道中同时清除数据，并且“先进先出”。

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。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。

信号量：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。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。

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

使用【共享内存+信号量+消息队列】的组合来实现服务器进程与客户进程间的通信。共享内存用来传递数据；信号量用来同步；消息队列用来 在客户端修改了共享内存后 通知服务器读取。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*数据结构和算法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、常用hash算法

直接定址，取关键字的某个线性函数值f(key)=a\*key+b

数字分析法，比如存手机号

平方取中，先平方，再去中间的几位

折叠法，对于关键字很长的情况，分为几个小组，再把各个小组的数字加到一起

除留余数f(key)=key mod p

随机数法

2、处理hash冲突

开放定址法，f(key)=(f(key)+d) mod m 线性探测法，对应的还有二次探测法

再散列函数法，换另一种hash算法重新算一遍

链表地址法

3、AVL如何插入删除

插入的同时观察是否破坏了树的平衡性（BF，每一个结点左子树深度减去右子树深度），如果是，就找出最小不平衡子树，在保持AVL特性的前提下，调整最小不平衡子树中各节点之间的链接关系，如果BF大于1，就右旋，如果BF小于-1，就左旋。

快速计算AVL深度：结点总数n，深度为n+1取以2为底的对数，向上取整

4、B树、B+树

多路查找树，B树每一个非根的分支节点都有k-1个元素和k个孩子，为了提高磁盘访问速度而设计，但是每一个非终端节点都有有效信息，B+树则不同，非终端节点只是索引，读写代价更低

5、O(g(n))=｛f(n)|存在正常数c和n0使得所有n>=n0有: 0<=f(n)<=cg(n)}

O(f(n))+o(g(n))=O(max{f(n),g(n)})

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*存储管理\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、分页

用户程序的地址空间被划分成若干固定大小的区域，称为“页”，相应地，内存空间分成若干个物理块，页和块的大小相等。可将用户程序的任一页放在内存的任一块中，实现了离散分配。

分页存储管理的地址机构

15 12 11 0

页号P 页内位移量W

页号4位，每个作业最多2的4次方=16页，表示页号从0000~1111（24-1），页内位移量的位数表示页的大小，若页内位移量12位，则2的12次方=4k，页的大小为4k，页内地址从000000000000~111111111111

若给定一个逻辑地址为A，页面大小为L，则

页号P=INT[A/L]，页内地址W=A MOD L

11406D=0010|110010001110B=2C8EH

页号为2,位移量为C8EH=3214D

或11406 DIV 4096=2

11406 MOD 4096=3214

分页系统中，允许将进程的每一页离散地存储在内存的任一物理块中，为了能在内存中找到每个页面对应的物理块，系统为每个进程建立一张页面映射表，简称页表。页表的作用是实现从页号到物理块号的地址映射。

页表：

页号 物理块号 存取控制

0 2

1 15（F）

2 14（E）

3 1

地址变换

（1） 程序执行时，从PCB中取出页表始址和页表长度（4），装入页表寄存器PTR。（2）由分页地址变换机构将逻辑地址自动分成页号和页内地址。例：11406D=0010|110010001110B=2C8EH 页号为2,位移量为C8EH=3214D

或11406 DIV 4096=2

11406 MOD 4096=3214

（3） 将页号与页表长度进行比较(2<4)，若页号大于或等于页表长度，则表示本次访问的地址已超越进程的地址空间，产生越界中断。（4）将页表始址与页号和页表项长度的乘积相加，便得到该页表项在页表中的位置。（5）取出页描述子得到该页的物理块号。 2 14（E）（6） 对该页的存取控制进行检查。（7）将物理块号送入物理地址寄存器中，再将有效地址寄存器中的页内地址直接送入物理地址寄存器的块内地址字段中，拼接得到实际的物理地址。

例：0010|110010001101B

1110|110010001101B=EC8EH=60558D

或 14\*4096+3214=60558D

2、分段

将用户程序地址空间分成若干个大小不等的段，每段可以定义一组相对完整的逻辑信息。存储分配时，以段为单位，段与段在内存中可以不相邻接，也实现了离散分配。

分段存储方式的引入：方便编程；分段共享；分段保护；动态链接；动态增长

分段地址结构

作业的地址空间被划分为若干个段，每个段定义了一组逻辑信息。例程序段、数据段等。每个段都从0开始编址，并采用一段连续的地址空间。

段的长度由相应的逻辑信息组的长度决定，因而各段长度不等。整个作业的地址空间是二维的。

15 12 11 0

段号 段内位移量

段号4位，每个作业最多24=16段，表示段号从0000~1111（24-1）；段内位移量12位，212=4k，表示每段的段内地址最大为4K（各段长度不同），从000000000000~111111111111

段表

段号 段长 起始地址 存取控制

0 1K 4096

1 4K 17500

2 2K 8192

地址变换

(1). 程序执行时，从PCB中取出段表始址和段表长度（3），装入段表寄存器。(2). 由分段地址变换机构将逻辑地址自动分成段号和段内地址。

例：7310D=0001|110010001110B=1C8EH

段号为1,位移量为C8EH=3214D

(3). 将段号与段表长度进行比较(1<3)，若段号大于或等于段表长度，则表示本次访问的地址已超越进程的地址空间，产生越界中断。(4). 将段表始址与段号和段表项长度的乘积相加，便得到该段表项在段表中的位置。(5). 取出段描述子得到该段的起始物理地址。1 4K 17500

(6). 检查段内位移量是否超出该段的段长(3214<4K)，若超过，产生越界中断。(7). 对该段的存取控制进行检查。(8). 将该段基址和段内地址相加，得到实际的物理地址。

例：0001|110010001101B

起始地址17500D+段内地址3214D=20714D

3、分页和分段有许多相似之处,比如两者都不要求作业连续存放.但在概念上两者完全不同,主要表现在以下几个方面:

(1)页是信息的物理单位,分页是为了实现非连续分配,以便解决内存碎片问题,或者说分页是由于系统管理的需要.段是信息的逻辑单位,它含有一组意义相对完整的信息,分段的目的是为了更好地实现共享,满足用户的需要.(2)页的大小固定,由系统确定,将逻辑地址划分为页号和页内地址是由机器硬件实现的.而段的长度却不固定,决定于用户所编写的程序,通常由编译程序在对源程序进行编译时根据信息的性质来划分.(3)分页的作业地址空间是一维的.分段的地址空间是二维的.

4、段页式存储管理

分页系统能有效地提高内存的利用率，而分段系统能反映程序的逻辑结构，便于段的共享与保护，将分页与分段两种存储方式结合起来，就形成了段页式存储管理方式。

在段页式存储管理系统中，作业的地址空间首先被分成若干个逻辑分段，每段都有自己的段号，然后再将每段分成若干个大小相等的页。对于主存空间也分成大小相等的页，主存的分配以页为单位。

段页式系统中，作业的地址结构包含三部分的内容：段号 页号 页内位移量

程序员按照分段系统的地址结构将地址分为段号与段内位移量，地址变换机构将段内位移量分解为页号和页内位移量。

为实现段页式存储管理，系统应为每个进程设置一个段表，包括每段的段号，该段的页表始址和页表长度。每个段有自己的页表，记录段中的每一页的页号和存放在主存中的物理块号。

地址变换的过程：

（1）程序执行时，从PCB中取出段表始址和段表长度，装入段表寄存器。（2）由地址变换机构将逻辑地址自动分成段号、页号和页内地址。（3）将段号与段表长度进行比较，若段号大于或等于段表长度，则表示本次访问的地址已超越进程的地址空间，产生越界中断。（4）将段表始址与段号和段表项长度的乘积相加，便得到该段表项在段表中的位置。（5）取出段描述子得到该段的页表始址和页表长度。（6）将页号与页表长度进行比较，若页号大于或等于页表长度，则表示本次访问的地址已超越进程的地址空间，产生越界中断。（7）将页表始址与页号和页表项长度的乘积相加，便得到该页表项在页表中的位置。（8）取出页描述子得到该页的物理块号。（9）对该页的存取控制进行检查。（10）将物理块号送入物理地址寄存器中，再将有效地址寄存器中的页内地址直接送入物理地址寄存器的块内地址字段中，拼接得到实际的物理地址。

5、页面置换：把一个页面从内存调换到磁盘的对换区中

抖动：在具有虚存的计算机中，由于频繁的调页活动使访问磁盘的次数过多而引起的系统效率降低的一种现象

常用的页面置换算法：先进先出法：（置换次数比较多）；最佳置换法（OPT）：选择将来不再使用或在最远的将来才被访问的页调换出去（不便于实现）；最近最少使用置换法（LRU）:当需要置换一页时，选择在最近一段时间里最久没有使用过的页面予以淘汰；最近未使用置换法（NUR）：是LRU算法的近似方法，选择在最近一段时间里未被访问过的页面予以淘汰

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Linux\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、ctrl z 就是挂起一个进程，暂停的意思

fg是将进程放到前台并唤醒

bg是将进程放到后台并唤醒

ctrl+h,在shell就是向左删除的作用

2、磁盘阵列的配置文件为/etc/raidtab 逻辑设备为 /dev/md0磁盘阵列

3、chmod ［who］ ［+ | - | =］ ［mode］ 文件名¼

u 表示“用户（user）”，即文件或目录的所有者。

g 表示“同组（group）用户”，即与文件属主有相同组ID的所有用户。

o 表示“其他（others）用户”。

a 表示“所有（all）用户”。它是系统默认值。

操作符号可以是：

+ 添加某个权限。

- 取消某个权限。

= 赋予给定权限并取消其他所有权限（如果有的话）。

数字设定法的一般形式为：

chmod ［mode］ 文件名¼

我们必须首先了解用数字表示的属性的含义：0表示没有权限，1表示可执行权限，2表示可写权限，4表示可读权限，然后将其相加。所以 数字属性的格式应为3个从0到7的八进制数，其顺序是（u）（g）（o）。

例如，如果想让某个文件的属主有“读/写”二种权限，需要把4（可读）+2（可写）＝6（读/写）。

4、新建一个管理员用户admin,需要使用的参数useradd -u 0 -o admin

-u 用户号 指定用户的用户号；因为系统用户的用户号为0，故指定用户号为0 如果同时有-o选项，则可以重复使用其他用户的标识号；因为系统本身存在用户号为0的系统用户，故应该使用该参数。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*数据库\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、第一范式1NF：在任何一个关系数据库中，第一范式（1NF）是对关系模式的基本要求，不满足第一范式（1NF）的数据库就不是关系数据库。如果一个关系模式R中的所有属性都是不可分的基本数据项，就满足第一范式

第二范式（2NF）：第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。第二范式（2NF）要求数据库表中的每个实例或行必须可以被唯一地区分。为实现区分通常需要为表加上一个列，以存储各个实例的唯一标识。第二范式（2NF）要求实体的属性完全依赖于主关键字。

第三范式（3NF）：满足第三范式（3NF）必须先满足第二范式（2NF）。简而言之，第三范式（3NF）要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。

2、where子句对被选择的列施加条件，

having子句对group by子句所产生的组施加条件。

order by 用于对结果集进行排序

group by 根据一个或多个列对结果集进行分组。返回结果，所依赖的列中元素不会重复

SELECT column1, column2 FROM table1, table2

WHERE [ conditions ]

GROUP BY column1, column2

HAVING [ conditions ]

ORDER BY column1, column2

3、MySQL数据库中，变量分为 系统变量（以"@@"开头）和用户自定义变量。系统变量分为全局系统变量(global)和会话系统变量(session)。

@@global 仅用于访问全局系统变量的值；

@@session 仅用于访问会话系统变量的值；

@@ 先访问会话系统变量的值，若不存在则去访问全局系统变量的值；

sql\_mode 为系统变量，既是全局系统变量，又是会话系统变量。

4、脏读又称无效数据的读出，是指在数据库访问中，事务T1将某一值修改，然后事务T2读取该值，此后T1因为某种原因撤销对该值的修改，这就导致了T2所读取到的数据是无效的。

5、MySQL实现了四种通信协议

TCP/IP协议，通常我们通过来连接MySQL，各种主要编程语言都是根据这个协议实现了连接模块

Unix Socket协议，这个通常我们登入MySQL服务器中使用这个协议，因为要使用这个协议连接MySQL需要一个物理文件，文件的存放位置在配置文件中有定义，值得一提的是，这是所有协议中最高效的一个。

Share Memory协议，这个协议一般人不知道，肯定也没用过，因为这个只有windows可以使用，使用这个协议需要在配置文件中在启动的时候使用–shared-memory参数，注意的是，使用此协议，一个host上只能有一个server，所以这个东西一般没啥用的，除非你怀疑其他协议不能正常工作，实际上微软的SQL Sever也支持这个协议

Named Pipes协议，这个协议也是只有windows才可以用，同shared memory一样，使用此协议，一个host上依然只能有一个server，即使是使用不同的端口也不行，Named Pipes 是为局域网而开发的协议。内存的一部分被某个进程用来向另一个进程传递信息，因此一个进程的输出就是另一个进程的输入。第二个进程可以是本地的（与第一个进程位于同一台计算机上），也可以是远程的（位于联网的计算机上）。正因为如此，假如你的环境中没有或者禁用TCP/IP环境，而且是windows服务器，那么好歹你的数据库还能工作。使用这个协议需要在启动的时候添加–enable-named-pipe选项

6、DECLARE @count SELECT @count=1将变量count值赋值为1 declare表示声明，@表示赋值

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*20160326\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*算法和数学\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

？？？？？？？？？？？？？？？？？？？？？？？？

1、标号1-n的n个人首尾相接，1到3报数，报到3的退出，求最后一个人的标号

2、给定一个字符串，求第一个不重复的字符（abbcad->c）要求时间复杂度为O(n)

3、递归求斐波那契数列，复杂度

int fun(int n){

if(n==0 || n==1)

return 1;

else

return fun(n-1)+fun(n-2);

}

4、蒲丰投针概率问题

5、两个整数求和，整数存在单链表中，高位数字存在头结点，以此类推，返回一个链表

6、实现next\_permutation，即下一字典序列，输入输出都为string

7、朴素字符串匹配算法复杂度O((n-m+1)\*m),KMP算法O（n-m）

8、哪些排序算法是稳定（序列中相同的元素排序后位置不变）的：冒泡、插入、归并

9、写一个程序，打印a b c ... z ab ac ad ... az bc bd.. bz cd ce ..cz...yz abc abd ...abz acd ... xyz abcd..xyz

10、自己实现一个hashtable

关键在于设计好hash算法和处理碰撞的方法，基本可按照教材上的写法

要注意的是，用链地址法其实不算高效，还是开放定址法更好一些，处理碰撞可以用线性探测法就好

hash算法常见的就是key%hash\_length

我们还希望散列函数满足下面几点:1、散列函数的输出值尽量接近均匀分布；2、x的微小变化可以使f(x)发生非常大的变化，即所谓“雪崩效应”(Avalanche effect)；

class HashItem{

int key, val;

public:

HashItem(int k, int k): key(k), val(v){}

const int& getKey(){

return key;

}

const int& getVal(){

return val;

}

};

class hashtable{

static const int SIZE = 256;

HashItem \*\* table; // 注意这是二级指针，指向对个HashItem\*

public:

HashTable(){

table = new HashItem\*[SIZE](); // 这里的括号是为了初始化为0

}

void set(key, val){

int idx = key%SIZE;

while(table[idx] && table[idx]->getKey() != key)

idx = (idx+1)%SIZE; // 当SIZE不够大时，这里会陷入死循环。可以检测一下。

if(table[idx]) delete table[idx];

table[idx] = new HashItem(key, val);

}

const int get(key){

int idx = key%SIZE;

while(table[idx] && table[idx]->getKey() != key)

idx = (idx+1)%SIZE; // SIZE不够大时，这里也面临死循环的问题

return table[idx] ? table[idx]->getVal() : -1; // 注意这里需要判断key不存在的情况

}

~HashTable(){

for(int i=0; i<SIZE; i++)

if(table[i]) delete table[i];

delete[] table; // 别忘了table本身也是要销毁的

}

}

11、输出1-1000的素数

所谓素数，就是只能被1和它本身整除的数字

int how\_many;

while (cin >> how\_many){

cout << "2 ";

for (int i = 3; i <= how\_many; i += 2){//偶数肯定不是素数，所以每次加2

bool sushu = true;

for (int j = 3; j <= sqrt(i); j++){//j代表除数，最多取到被除数的开方即可

if (i%j == 0){

sushu = false;

break;

}

}

if (sushu)

cout << i << " ";

}

}

12、单链表的翻转

从第2个节点到第N个节点，依次逐节点插入到第1个节点(head节点)之后，最后将第一个节点挪到新表的表尾

ActList\* ReverseList3(ActList\* head)

{

ActList\* p;

ActList\* q;

p=head->next;

while(p->next!=NULL){

q=p->next;

p->next=q->next;

q->next=head->next;

head->next=q;

}

p->next=head;//相当于成环

head=p->next->next;//新head变为原head的next

p->next->next=NULL;//断掉环

return head;

}

13、递归和迭代的区别

递归的基本概念:程序调用自身的编程技巧称为递归,是函数自己调用自己.使用递归要注意的有两点:1)递归就是在过程或函数里面调用自身;2)在使用递归时,必须有一个明确的递归结束条件,称为递归出口.迭代:利用变量的原值推算出变量的一个新值.如果递归是自己调用自己的话,迭代就是A不停的调用B.递归中一定有迭代,但是迭代中不一定有递归,大部分可以相互转换.能用迭代的不用递归,递归调用函数,浪费空间,并且递归太深容易造成堆栈的溢出.

//这是递归

int funcA(int n)

{

if(n > 1)

return n+funcA(n-1);

else

return 1;

}

//这是迭代

int funcB(int n)

{

int i,s=0;

for(i=1;i<n;i++)

s+=i;

return s;

}

14、在进行特征选择时，即分析那些协变量对目标变量有较大的影响时，可以用互信息，特征样本分布分析

15、P(B|A)=P(AB)/P(A)

16、简单随机样本,所以各样本间相互独立,那么就有：

E(X1+X2+……+Xn) = E(X1)+E(X2)+……+E(Xn)

D(X1+X2+……+Xn) = D(X1)+D(X2)+……+D(Xn)

所以有h~N(70,25), h'=(h1+h2+...h100)/100, 则 h'~N(70,0.25)

17、u检验的应用条件是样本例数n较大或样本例数数量虽小但总体标准差已知

Z检验的条件：样本来自正态分布且方差已知的情况

T检验的条件：样本来自正态分布且方差未知的情况，两独立样本T检验主要用于检验两个样本的平均数差异。

18、正态分布曲线性质中有 ：P（μ-σ<X≤μ+σ）=68.3%P（μ-2σ<X≤μ+2σ）=95.4%P（μ-3σ<X≤μ+3σ）=99.7%

19、一个矩阵的不同特征值的特征向量线性无关

20、在相同样本量下，重复抽样与不重复抽样的抽样平均误差大小关系是：重复抽样误差大

21、输出金字塔

int level = 0;

while (cin >> level){

for (int i = 1; i <= level; i++){

for (int j = level; j > i; j--){

cout << " ";

}

for (int m = i; m > 1; m--){

cout << m;

}

for (int k = 1; k <= i; k++){

cout << k;

}

cout << endl;

}

}

22、括号匹配返回不匹配的位置，输出 false

stack<char> mystack;

stack<int> position;

char input[250] = { 0 };

while (cin >> input){

int length = strlen(input);

int idx = 0;

int flag = 0;

while (input[idx] != '\0'){

if (input[idx] == '('){

mystack.push(input[idx]);

position.push(idx);

}

if (input[idx] == ')'){

if ( mystack.size()==0 || mystack.top() != '('){

cout << "wrong at the" << idx << " )";

flag = 1;

break;

}

else if (mystack.top() == '('){

mystack.pop();

position.pop();

}

}

idx++;

}

if (mystack.size() != 0)

cout << "wrong at the" << position.top() << " (";

else if (mystack.size() == 0 && flag == 0)

cout << "right";

while (mystack.size() != 0)

mystack.pop();

while (position.size() != 0)

position.pop();

}

如果要求只用递归而不用循环来实现

只目能使用递归并且不能出现循环语句。这个时候，我们应该如何处理呢？其实告诉了大家递归，就比较好想了：怎么定义好问题和子问题。

如果字符串中的括号是匹配的，则'('的数量和')'的数量是相等的，反之是不相等的。这样，在递归的过程中，可以保存一个变量，用来记录'('的 数量和')'的数量是否匹配。这样定义递归问题f(p,count)，表示当前字符p之前的字符串中'('的数量和')'的数量的匹配情况，p表示指向当 前字符的指针。初始的时候，f(p, 0)，递归的过程如下：

如果p为空，则考察count是否为0，如果为0，则匹配；如果不为0，则不匹配；

如果不为空，则考察当前字符p，如果p='('，则递归调用f(p++, count++);如果p=')'，则递归调用f(p++, count--)。如果p是其他的字符，并不是'('和')'，则递归调用f(p++, count)，count不变，继续考虑下一次字符。其中需要检查和保证count>=0.

void f(char \*p, int count)

{

if (\*p == '\0' && count == 0)

{

printf("yes\n");

return;

}

else if (\*p == '\0' && count != 0)

{

printf("no\n");

return;

}

else if (\*p != '\0' && count >= 0)

{

if (\*p == '(')

{

count++;

p++;

f(p, count);

}

else if (\*p == ')')

{

count--;

p++;

f(p, count);

}

else

{

p++;

f(p, count);

}

}

else

{

printf("no\n");

return;

}

}

int main()

{

char str[200];

while (scanf("%s", str))

{

char \*p = str;

f(p, 0);

}

return 0;

}

23、输入一串字符，输出不重复的单词，过略掉重复的单词

int main(void){

char input[250] = { 0 };

while (cin >> input){

vector<string> output;

int idx\_s = 0;

int idx\_e = 0;

int word\_len = 0;

int down\_flag = 0;

vector<string>::iterator it;

while (input[idx\_e] != '\0'){

if (input[idx\_e] != '\_'){

idx\_e++;

if (input[idx\_e] == '\0')

goto down;

}

else{

down: word\_len = idx\_e - idx\_s;

char tmp[250] = { 0 };

strncpy(tmp, input + idx\_s, word\_len);

it = find(output.begin(), output.end(), tmp);

if (it == output.end()){

output.push\_back(tmp);

idx\_s = idx\_e+1;

idx\_e++;

}

else{

idx\_s = idx\_e + 1;

idx\_e++;

}

if (down\_flag)

break;

}

}

string out\_str;

for (it = output.begin(); it < output.end(); it++){

out\_str += \*it;

out\_str += '\_';

}

cout << out\_str;

}

}

24、给出秒数，输出是几天几小时几分。格式为： xxx： xx： xx： xx

和除法类似

int main(void){

int a;

int day\_part=24\*60\*60;

int hour\_part=60\*60;

int min\_part=60;

while(cin>>a){

int day=a/day\_part;

a=a%day\_part;

int hout=a/hour\_part;

a=a%hout\_part;

int min=a/min\_part;

a=a%min\_part;

cout<<day<<':'<<hour<<':'<<min<<':'<<a<<endl;

}

}

25、给出一串字符串，将小写转换为大写，数字去掉后输出。

int main(void){

//设最多250个字

char input[250] = {0};

while (cin >> input){

int length = strlen(input);

char\* output = new char[length + 1];

int i\_idx = 0;

int o\_idx = 0;

while (input[i\_idx] != '\0'){

if (input[i\_idx] >= '0' && input[i\_idx] <= '9'){

i\_idx++;

continue;

}

else if (input[i\_idx] >= 'a' && input[i\_idx] <= 'z'){

output[o\_idx++] = input[i\_idx++] + 'A' - 'a';

}

else

output[o\_idx++] = input[i\_idx++];

}

output[o\_idx] = '\0';

cout << output;

}

}

26、求分数的精确值 例如： 1/3=3 1/7=142857

将除法的结果保存在一个数组中，再依次输出。除法的过程可以模仿人工做除法的方式，先将被除数乘以10，得到一位商以后，将余数乘以10作为下一轮计算的被除数：

160/19->8余8

80/19->4余4

...假设要求的是16/19的值

当某次余数为0时，则表明除尽。

程序如下

int main(void){

//设最多保存200位

int a,b;

while(cin>>a>>b){

assert(a>=10&&a<b&&b<100);

char\* result=new char[201];

int integer\_part=a/b; //结果的整数部分

a=a%b;

int cnt=0;

while(a!=0 && cnt<200){

result[cnt++]=(a\*10)/b+'0';

a=(a\*10)%b;

}

result[cnt]='\0';

cout<<result;

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*计算机网络\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、https和http的区别，RSA是什么，数字签名是什么？？？？？？？？？？？？？？？？？？？

2、HTTP状态码

1xx消息 100 Continue 101 Switching Protocols 102 Processing

2xx成功 200 OK 201 Created 202 Accepted 206 Partial Content

3xx重定向 301 Moved Permanently 302 Found

4xx客户端错误 400 Bad Request 403 Forbidden 404 Not Found

5xx服务器错误 500 Internal Server Error 502 Bad Gateway 503 Service Unavailable

3、内部重定向、外部重定向

4、htons 把unsigned short类型从主机序转换到网络序

htonl 把unsigned long类型从主机序转换到网络序

ntohs 把unsigned short类型从网络序转换到主机序

ntohl 把unsigned long类型从网络序转换到主机序

WSANtohs() 将一个以网络字节顺序表示的无符号短整形数转换为主机字节顺序。

5、利用滑动窗口实现流量控制

如果发送方把数据发送得过快，接收方可能会来不及接收，这就会造成数据的丢失。所谓流量控制就是让发送方的发送速率不要太快，要让接收方来得及接收。

6、TCP的拥塞控制

拥塞：即对资源的需求超过了可用的资源。若网络中许多资源同时供应不足，网络的性能就要明显变坏，整个网络的吞吐量随之负荷的增大而下降。

拥塞控制：防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。拥塞控制所要做的都有一个前提：网络能够承受现有的网络负荷。拥塞控制是一个全局性的过程，涉及到所有的主机、路由器，以及与降低网络传输性能有关的所有因素。

流量控制：指点对点通信量的控制，是端到端正的问题。流量控制所要做的就是抑制发送端发送数据的速率，以便使接收端来得及接收。

拥塞控制代价：需要获得网络内部流量分布的信息。在实施拥塞控制之前，还需要在结点之间交换信息和各种命令，以便选择控制的策略和实施控制。这样就产生了额外的开销。拥塞控制还需要将一些资源分配给各个用户单独使用，使得网络资源不能更好地实现共享。

几种拥塞控制方法

慢开始( slow-start )、拥塞避免( congestion avoidance )、快重传( fast retransmit )和快恢复( fast recovery )。

慢开始和拥塞避免

发送方维持一个拥塞窗口 cwnd ( congestion window )的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方让自己的发送窗口等于拥塞。

发送方控制拥塞窗口的原则是：只要网络没有出现拥塞，拥塞窗口就再增大一些，以便把更多的分组发送出去。但只要网络出现拥塞，拥塞窗口就减小一些，以减少注入到网络中的分组数。

慢开始算法：当主机开始发送数据时，如果立即所大量数据字节注入到网络，那么就有可能引起网络拥塞，因为现在并不清楚网络的负荷情况。因此，较好的方法是先探测一下，即由小到大逐渐增大发送窗口，也就是说，由小到大逐渐增大拥塞窗口数值。通常在刚刚开始发送报文段时，先把拥塞窗口 cwnd 设置为一个最大报文段MSS的数值。而在每收到一个对新的报文段的确认后，把拥塞窗口增加至多一个MSS的数值（指数级）。用这样的方法逐步增大发送方的拥塞窗口 cwnd ，可以使分组注入到网络的速率更加合理。

另，慢开始的“慢”并不是指cwnd的增长速率慢，而是指在TCP开始发送报文段时先设置cwnd=1，使得发送方在开始时只发送一个报文段（目的是试探一下网络的拥塞情况），然后再逐渐增大cwnd。

为了防止拥塞窗口cwnd增长过大引起网络拥塞，还需要设置一个慢开始门限ssthresh状态变量（如何设置ssthresh）。慢开始门限ssthresh的用法如下：

当 cwnd < ssthresh 时，使用上述的慢开始算法。

当 cwnd > ssthresh 时，停止使用慢开始算法而改用拥塞避免算法。

当 cwnd = ssthresh 时，既可使用慢开始算法，也可使用拥塞控制避免算法。

拥塞避免算法：让拥塞窗口cwnd缓慢地增大，即每经过一个往返时间RTT就把发送方的拥塞窗口cwnd加1，而不是加倍。这样拥塞窗口cwnd按线性规律缓慢增长，比慢开始算法的拥塞窗口增长速率缓慢得多。 无论在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞（其根据就是没有收到确认），就要把慢开始门限ssthresh设置为出现拥塞时的发送方窗口值的一半（但不能小于2）。然后把拥塞窗口cwnd重新设置为1，执行慢开始算法。这样做的目的就是要迅速减少主机发送到网络中的分组数，使得发生拥塞的路由器有足够时间把队列中积压的分组处理完毕。 强调：“拥塞避免”并非指完全能够避免了拥塞。利用以上的措施要完全避免网络拥塞还是不可能的。“拥塞避免”是说在拥塞避免阶段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。

快重传算法首先要求接收方每收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时才进行捎带确认。快重传算法还规定，发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段M3，而不必继续等待M3设置的重传计时器到期。

与快重传配合使用的还有快恢复算法，其过程有以下两个要点：当发送方连续收到三个重复确认，就执行“乘法减小”算法，把慢开始门限ssthresh减半。这是为了预防网络发生拥塞。请注意：接下去不执行慢开始算法；由于发送方现在认为网络很可能没有发生拥塞，因此与慢开始不同之处是现在不执行慢开始算法（即拥塞窗口cwnd现在不设置为1），而是把cwnd值设置为慢开始门限ssthresh减半后的数值，然后开始执行拥塞避免算法（“加法增大”），使拥塞窗口缓慢地线性增大。

TCP的拥塞控制主要原理依赖于一个拥塞窗口(cwnd)来控制，在之前我们还讨论过TCP还有一个对端通告的接收窗口(rwnd)用于流量控制。由于需要考虑拥塞控制和流量控制两个方面的内容，因此TCP的真正的发送窗口=min(rwnd, cwnd)。但是rwnd是由对端确定的，网络环境对其没有影响，所以在考虑拥塞的时候我们一般不考虑rwnd的值。

TCP是如何确定网络进入了拥塞状态的，TCP认为网络拥塞的主要依据是它重传了一个报文段。上面提到过，TCP对每一个报文段都有一个定时器，称为重传定时器(RTO)，当RTO超时且还没有得到数据确认，那么TCP就会对该报文段进行重传，当发生超时时，那么出现拥塞的可能性就很大，某个报文段可能在网络中某处丢失，并且后续的报文段也没有了消息，在这种情况下，TCP反应比较“强烈”：把ssthresh降低为cwnd值的一半；把cwnd重新设置为1；重新进入慢启动过程。

其实TCP还有一种情况会进行重传：那就是收到3个相同的ACK。TCP在收到乱序到达包时就会立即发送ACK，TCP利用3个相同的ACK来判定数据包的丢失，此时进行快速重传，快速重传做的事情有：把ssthresh设置为cwnd的一半；把cwnd再设置为ssthresh的值(具体实现有些为ssthresh+3)；重新进入拥塞避免阶段。

后来的“快速恢复”算法是在上述的“快速重传”算法后添加的，当收到3个重复ACK时，TCP最后进入的不是拥塞避免阶段，而是快速恢复阶段。快速重传和快速恢复算法一般同时使用。快速恢复的思想是“数据包守恒”原则，即同一个时刻在网络中的数据包数量是恒定的，只有当“老”数据包离开了网络后，才能向网络中发送一个“新”的数据包，如果发送方收到一个重复的ACK，那么根据TCP的ACK机制就表明有一个数据包离开了网络，于是cwnd加1。如果能够严格按照该原则那么网络中很少会发生拥塞，事实上拥塞控制的目的也就在修正违反该原则的地方。

具体来说快速恢复的主要步骤是：当收到3个重复ACK时，把ssthresh设置为cwnd的一半，把cwnd设置为ssthresh的值加3，然后重传丢失的报文段，加3的原因是因为收到3个重复的ACK，表明有3个“老”的数据包离开了网络；再收到重复的ACK时，拥塞窗口增加1；当收到新的数据包的ACK时，把cwnd设置为第一步中的ssthresh的值。原因是因为该ACK确认了新的数据，说明从重复ACK时的数据都已收到，该恢复过程已经结束，可以回到恢复之前的状态了，也即再次进入拥塞避免状态。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*数据库\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、MySQL数据库优化？？？？？？？？？？？？？？？？？？？？？？

2、DML数据操纵语言是SQL语言中，负责对数据库对象运行数据访问工作的指令集，以INSERT、UPDATE、DELETE三种指令为核心，分别代表插入、更新与删除

INSERT INTO myTable (col1, col2, col3, col4, col5) VALUES (1, 2, 3, '4', '5')

UPDATE myTable SET Col1 = 3, Col2 = 5, Col4 = 5 WHERE Col0 = 198273

DELETE FROM myTable WHERE col0 = 1918299

类型：自含式，可由用户独立地通过交互方式进行操作；嵌入式，不能独立地进行操作，必须嵌入某一种宿主语言（如C、PL\1等）中才能使用

3、数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用B树及其变种B+树。索引问题就是一个查找问题。减少磁盘读取是重点。

B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)。一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。在B树中查找给定关键字的方法是：首先把根结点取来，在根结点所包含的关键字K1,…,kj查找给定的关键字（可用顺序查找或二分查找法），若找到等于给定值的关键字，则查找成功；否则，一定可以确定要查的关键字在某个Ki或Ki+1之间，于是取Pi所指的下一层索引节点块继续查找，直到找到，或指针Pi为空时查找失败。

B+树非叶节点中存放的关键码并不指示数据对象的地址指针，非也节点只是索引部分。所有的叶节点在同一层上，包含了全部关键码和相应数据对象的存放地址指针，且叶节点按关键码从小到大顺序链接。B+树有两种搜索方法：一种是按叶节点自己拉起的链表顺序搜索。一种是从根节点开始搜索，和B树类似，不过如果非叶节点的关键码等于给定值，搜索并不停止，而是继续沿右指针，一直查到叶节点上的关键码。所以无论搜索是否成功，都将走完树的所有层。

这两种处理索引的数据结构的不同之处：

a，B树中同一键值不会出现多次，并且它有可能出现在叶结点，也有可能出现在非叶结点中。而B+树的键一定会出现在叶结点中，并且有可能在非叶结点中也有可能重复出现，以维持B+树的平衡。

b，因为B树键位置不定，且在整个树结构中只出现一次，虽然可以节省存储空间，但使得在插入、删除操作复杂度明显增加。B+树相比来说是一种较好的折中。

c，B树的查询效率与键在树中的位置有关，最大时间复杂度与B+树相同(在叶结点的时候)，最小时间复杂度为1(在根结点的时候)。而B+树的时候复杂度对某建成的树是固定的。B+ 树中，数据对象的插入和删除仅在叶节点上进行。

4、数据库事务

是数据库管理系统执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成。它的存在包含有以下两个目的：

a、为数据库操作序列提供了一个从失败中恢复到正常状态的方法，同时提供了数据库即使在异常状态下仍能保持一致性的方法。

b、当多个应用程序在并发访问数据库时，可以在这些应用程序之间提供一个隔离方法，以防止彼此的操作互相干扰。

当事务被提交给了DBMS（数据库管理系统），则DBMS（数据库管理系统）需要确保该事务中的所有操作都成功完成且其结果被永久保存在数据库中，如果事务中有的操作没有成功完成，则事务中的所有操作都需要被回滚，回到事务执行前的状态;同时，该事务对数据库或者其他事务的执行无影响，所有的事务都好像在独立的运行。

但在现实情况下，失败的风险很高。在一个数据库事务的执行过程中，有可能会遇上事务操作失败、数据库系统/操作系统失败，甚至是存储介质失败等情况。这便需要DBMS对一个执行失败的事务执行恢复操作，将其数据库状态恢复到一致状态（数据的一致性得到保证的状态）。为了实现将数据库状态恢复到一致状态的功能，DBMS通常需要维护事务日志以追踪事务中所有影响数据库数据的操作

并非任意的对数据库的操作序列都是数据库事务。数据库事务拥有以下四个特性，习惯上被称之为ACID特性。

原子性（Atomicity）：事务作为一个整体被执行，包含在其中的对数据库的操作要么全部被执行，要么都不执行[3]。

一致性（Consistency）：事务应确保数据库的状态从一个一致状态转变为另一个一致状态。一致状态的含义是数据库中的数据应满足完整性约束[3]。

隔离性（Isolation）：多个事务并发执行时，一个事务的执行不应影响其他事务的执行[3]。

持久性（Durability）：已被提交的事务对数据库的修改应该永久保存在数据库中

5、并发访问数据库

数据库管理系统（DBMS）中的并发控制的任务是确保在多个事务同时存取数据库中同一数据时不破坏事务的隔离性和统一性以及数据库的统一性。两个事务读入同一数据并同时修改，其中一个事务提交的结果破坏了另一个事务提交的结果，导致其数据的修改被丢失，破坏了事务的隔离性。并发控制要解决的就是这类问题。

乐观并发控制（又名“乐观锁”，Optimistic Concurrency Control，缩写“OCC”）是一种并发控制的方法。它假设多用户并发的事务在处理时不会彼此互相影响，各事务能够在不产生锁的情况下处理各自影响的那部分数据。在提交数据更新之前，每个事务会先检查在该事务读取数据后，有没有其他事务又修改了该数据。如果其他事务有更新的话，正在提交的事务会进行回滚。乐观并发控制多数用于数据争用不大、冲突较少的环境中，这种环境中，偶尔回滚事务的成本会低于读取数据时锁定数据的成本，因此可以获得比其他并发控制方法更高的吞吐量。

悲观并发控制（又名“悲观锁”，Pessimistic Concurrency Control，缩写“PCC”）是一种并发控制的方法。它可以阻止一个事务以影响其他用户的方式来修改数据。如果一个事务执行的操作读某行数据应用了锁，那只有当这个事务把锁释放，其他事务才能够执行与该锁冲突的操作。

悲观并发控制主要用于数据争用激烈的环境，以及发生并发冲突时使用锁保护数据的成本要低于回滚事务的成本的环境中。

6、数据库有哪些锁

共享锁 修改锁 独占锁 结构锁 意向锁 批量修改锁

7、数据库模式定义语言DDL(Data Definition Language)，是用于描述数据库中要存储的现实世界实体的语言。如CREATE ALTER DROP TRUNCATE

数据查询语言（Data Query Language, DQL）是SQL语言中，负责进行数据查询而不会对数据本身进行修改的语句，这是最基本的SQL语句。如SELECT、FROM、WHERE、GROUP BY和ORDER BY

数据控制语言 (Data Control Language) 在SQL语言中，是一种可对数据访问权进行控制的指令，它可以控制特定用户账户对数据表、查看表、预存程序、用户自定义函数等数据库对象的控制权。由 GRANT 和 REVOKE 两个指令组成。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*进程和线程\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、举一个线程不安全的例子，如何解决线程安全问题

线程安全是编程中的术语，指某个函数、函数库在多线程环境中被调用时，能够正确地处理多个线程之间的共享变量，使程序功能正确完成。很多C库代码（比如某些strtok的实现，它将“多次调用中需要保持不变的状态”储存在静态变量中，导致不恰当的共享）不是线程安全的。

2、进程间的同步方式（与进程间的通信不同，不要混淆）

临界区（Critical Section）、互斥量（Mutex）、信号量（Semaphore）、事件（Event）

临界区：通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。在任意时刻只允许一个线程对共享资源进行访问，如果有多个线程试图访问公共资源，那么在有一个线程进入后，其他试图访问公共资源的线程将被挂起，并一直等到进入临界区的线程离开，临界区在被释放后，其他线程才可以抢占。

互斥量：采用互斥对象机制。 只有拥有互斥对象的线程才有访问公共资源的权限，因为互斥对象只有一个，所以能保证公共资源不会同时被多个线程访问。互斥不仅能实现同一应用程序的公共资源安全共享，还能实现不同应用程序的公共资源安全共享 .互斥量比临界区复杂。因为使用互斥不仅仅能够在同一应用程序不同线程中实现资源的安全共享，而且可以在不同应用程序的线程之间实现对资源的安全共享。

信号量：它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目 .信号量对象对线程的同步方式与前面几种方法不同，信号允许多个线程同时使用共享资源，这与操作系统中的PV操作相同。它指出了同时访问共享资源的线程最大数目。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。

事件：通过通知操作的方式来保持线程的同步，还可以方便实现对多个线程的优先级比较的操作 .

总结：互斥量与临界区的作用非常相似，但互斥量是可以命名的，也就是说它可以跨越进程使用。所以创建互斥量需要的资源更多，所以如果只为了在进程内部是用的话使用临界区会带来速度上的优势并能够减少资源占用量。因为互斥量是跨进程的互斥量一旦被创建，就可以通过名字打开它。 互斥量（Mutex），信号灯（Semaphore），事件（Event）都可以被跨越进程使用来进行同步数据操作，而其他的对象与数据同步操作无关，但对于进程和线程来讲，如果进程和线程在运行状态则为无信号状态，在退出后为有信号状态。所以可以使用WaitForSingleObject来等待进程和线程退出。

进程间通信就是在不同进程之间传播或交换信息，进程间通信主要包括管道, 系统IPC(包括消息队列,信号量,共享存储), SOCKET.

进程/线程同步机制与进程间通信机制比较：很明显2者有类似，但是差别很大

同步主要是临界区、互斥、信号量、事件

进程间通信是管道、内存共享、消息队列、信号量、socket（以上几种方式的比较：管道：速度慢，容量有限，只有父子进程能通讯；FIFO：任何进程间都能通讯，但速度慢；消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题；信号量：不能传递复杂消息，只能用来同步；共享内存区：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全，当然，共享内存区同样可以用作线程间通讯，不过没这个必要，线程间本来就已经共享了同一进程内的一块内存）

共通之处是，信号量和消息（事件）

3、生产者消费者问题

生产者消费者问题是一个著名的线程同步问题，该问题描述如下：有一个生产者在生产产品，这些产品将提供给若干个消费者去消费，为了使生产者和消费者能并发执行，在两者之间设置一个具有多个缓冲区的缓冲池，生产者将它生产的产品放入一个缓冲区中，消费者可以从缓冲区中取走产品进行消费，显然生产者和消费者之间必须保持同步，即不允许消费者到一个空的缓冲区中取产品，也不允许生产者向一个已经放入产品的缓冲区中再次投放产品。

注意两点：第一．从缓冲区取出产品和向缓冲区投放产品必须是互斥进行的。可以用关键段和互斥量来完成。第二．生产者要等待缓冲区为空，这样才可以投放产品，消费者要等待缓冲区不为空，这样才可以取出产品进行消费。并且由于有二个等待过程，所以要用二个事件或信号量来控制。

如果将消费者改成2个，缓冲池改成拥有4个缓冲区的大缓冲池，用二个信号量就可以解决这种缓冲池有多个缓冲区的情况——用一个信号量A来记录为空的缓冲区个数，另一个信号量B记录非空的缓冲区个数，然后生产者等待信号量A，消费者等待信号量B就可以了。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*数据结构\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、树的度

二叉树：n2=n0-1

树的分支数=2\*n2+n1=n0+n1+n2-1

普通树的结点数：n0+n1+n2+...nk

普通树的度数：0\*n0+1\*n1+2\*n2+...k\*nk

普通树的节点数=普通数的度数-1

2、哈希表的实现方式

数组（如果是用链地址法处理哈希冲突，则每个数组元素可能还是一个链表）

4、哈希表和B+树

B+树是一个平衡的多叉树，从根节点到每个叶子节点的高度差值不超过1，而且同层级的节点间有指针相互链接。在B+树上的常规检索，从根节点到叶子节点的搜索效率基本相当，不会出现大幅波动，而且基于索引的顺序扫描时，也可以利用双向指针快速左右移动，效率非常高。因此，B+树索引被广泛应用于数据库、文件系统等场景。

哈希索引就是采用一定的哈希算法，把键值换算成新的哈希值，检索时不需要类似B+树那样从根节点到叶子节点逐级查找，只需一次哈希算法即可立刻定位到相应的位置，速度非常快。

B+树索引和哈希索引的明显区别是：

如果是等值查询，那么哈希索引明显有绝对优势，因为只需要经过一次算法即可找到相应的键值；当然了，这个前提是，键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据；如果是范围查询检索，这时候哈希索引就毫无用武之地了，因为原先是有序的键值，经过哈希算法后，有可能变成不连续的了，就没办法再利用索引完成范围查询检索；

同理，哈希索引也没办法利用索引完成排序，以及like ‘xxx%’这样的部分模糊查询（这种部分模糊查询，其实本质上也是范围查询）；哈希索引也不支持多列联合索引的最左匹配规则；B+树索引的关键字检索效率比较平均，不像B树那样波动幅度大，在有大量重复键值情况下，哈希索引的效率也是极低的，因为存在所谓的哈希碰撞问题。

5、操作系统中的堆（从任一结点出发,到根的路径上所经过的结点序列按其关键字有序）和栈（数据结构中的栈通常采用的存储结构是顺序存储结构和链表存储结构）

栈是随函数被调用时分配的空间。栈上分配的空间是临时的，在函数退出后将被系统释放，不会造成内存泄露，不得用delete或free操作，因为栈的空间小所以在栈上不能获得大量的内存块，一般最大也就不到10M

堆是在整个进程的未分配空间中分配的内存，由malloc或new分配，一般必须由free或delete释放。堆上可以分配大量的内存，只要你的机器吃得消。一般来说，由new和malloc分配的内存都在堆上，全局变量也在堆上（但是不是new，malloc出来的也会自动清理）。函数内部的其他变量和常量都在栈上。

从堆和栈的功能和作用来通俗的比较,堆主要用来存放对象的，栈主要是用来执行程

序的.而这种不同又主要是由于堆和栈的特点决定的:

在编程中，例如C/C++中，所有的方法调用都是通过栈来进行的,所有的局部变量,形式参数都是从栈中分配内存空间的。实际上也不是什么分配,只是从栈顶向上用就行,就好像工厂中的传送带(conveyor belt)一样,Stack Pointer会自动指引你到放东西的位置,你所要做的只是把东西放下来就行.退出函数的时候，修改栈指针就可以把栈中的内容销毁，这样的模式速度最快,当然要用来运行程序了.需要注意的是,在分配的时候,比如为一个即将要调用的程序模块分配数据区时,应事先知道这个数据区的大小,也就说是虽然分配是在程序运行时进行的,但是分配的大小多少是确定的,不变的,而这个"大小多少"是在编译时确定的,不是在运行时.

堆是应用程序在运行的时候请求操作系统分配给自己内存，由于从操作系统管理的内存分配,所以在分配和销毁时都要占用时间，因此用堆的效率非常低.但是堆的优点在于,编译器不必知道要从堆里分配多少存储空间，也不必知道存储的数据要在堆里停留多长的时间,因此,用堆保存数据时会得到更大的灵活性。事实上,面向对象的多态性,堆内存分配是必不可少的,因为多态变量所需的存储空间只有在运行时创建了对象之后才能确定.在C++中，要求创建一个对象时，只需用new命令编制相关的代码即可。执行这些代码时会在堆里自动进行数据的保存.当然，为达到这种灵活性，必然会付出一定的代价:在堆里分配存储空间时会花掉更长的时间！这也正是导致效率低的原因

6、完全二叉树的存储结构通常采用顺序存储结构

7、树、二叉树、森林相互的转化

从树到二叉树的转化方法一般是“兄弟孩子法”，也就是把树结点的第一个孩子变为左孩子结点，兄弟结点变为右孩子节点。由此得到的二叉树，根节点必无右孩子。

森林到二叉树树的转化，是依次把森林的每棵树转化为二叉树，把最后一颗转化后的二叉树的根节点作为上一颗二叉树的右孩子节点，这样把两颗二叉树接成一颗二叉树。然后依次向前连接，直到只有一颗二叉树。

森林的先序遍历和中序遍历即为其对应的二叉树的先序遍历和中序遍历

树的先根遍历和后根遍历即为二叉树的先序遍历和中序遍历

8、哈夫曼树的特点是：它是带权路径长度WPL最小的二叉树。

9、将N条长度均为M的有序链表进行合并，合并以后的链表也保持有序，时间复杂度

利用堆来合并，（ O（N） + O（log N \* N ）) \* M。

先利用最链表第一个数，N个数建立堆，复杂度 O （N）

重构堆，并排序，复杂度 O（logN \* N ）

每个链表M个数，上述两步重复M次。结果为

M \* （O(N) + O(logN \* N)）= O (M \* N \* logN)

10、判断一个单向链表中是否存在环

让快慢指针都从链表表头开始，快指针一次向前移动连个位置，慢指针每次向前移动一个位置。如果快指针到达了NULL，说明不存在环，但如果快指针追上了慢指针，说明存在环。

11、从大的方面来说，数据结构是反映数据的一种形式，它具体分为逻辑结构和物理结构，逻辑结构：它是表现数据之间的一种关系的结构，分为线性结构和非线性结构；物理结构：它是表现数据的是如何存储的结构，计算机内部是如何安排该数据的存储，通常分为顺序结构，链式结构，索引结构，哈希结构；

12、双链表

如果链表数据是无序的，则单向搜索与双向搜索平均速度相同

如果链表是有序的，而要搜索的数据距离最小值（最大值）较近，这种情况下双向搜索平均速度更快。

因此双向搜索更稳定，方差更小

13、在（非空）广义表中：表头head可以是原子或者一个表，表尾tail一定是一个表

14、栈的大小是固定的，堆的大小受限于系统中有效的虚拟内存；栈的空间由系统决定何时释放，堆需要自己决定何时去释放；堆的使用容易产生碎片，但是用起来最方便

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*设计模式\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、哪种设计模式可以时调用者不关心具体算法

工厂模式，定义一个用于创建对象的接口，让子类决定实例化哪一个类，工厂方法使一个类的实例化延迟到其子类。

可以使代码结构清晰，有效地封装变化。在编程中，产品类的实例化有时候是比较复杂和多变的，通过工厂模式，将产品的实例化封装起来，使得调用者根本无需关心产品的实例化过程，只需依赖工厂即可得到自己想要的产品；对调用者屏蔽具体的产品类。如果使用工厂模式，调用者只关心产品的接口就可以了，至于具体的实现，调用者根本无需关心。即使变更了具体的实现，对调用者来说没有任何影响；降低耦合度。产品类的实例化通常来说是很复杂的，它需要依赖很多的类，而这些类对于调用者来说根本无需知道，如果使用了工厂方法，我们需要做的仅仅是实例化好产品类，然后交给调用者使用。对调用者来说，产品所依赖的类都是透明的。

组装汽车是工厂模式优点的常见案例。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*编程基础\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、面向对象的特点：封装、多态（覆盖：虚函数、接口；重载：同名函数）、继承（继承：实现继承、可视继承；组合：接口继承、纯虚类）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*C++\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、动态申请二维数组

cin>>m>>n; //m,n分别为行数和列数

int \*\* p;

p = new int\*[m];

for( int i = 0; i < m; i++ )

　　p[i] = new int[n];

然后就可以如二维数组给p[i][j]赋值了.

最后别忘了释放:

for( int i = 0; i < m; i++ )

　　delete[]p[i];

delete[]p;

2、STL algorithm头文件很实用 sort函数的功能很易于扩展，我们可以给它一个比较函数，让它按我们希望的方式工作。如果想要自定义比较函数，就像这样：

bool less\_int(int a,int b){

return b<a;

}

...

sort(numbers.begin(),numbers.end(),less\_int);

函数名称随便，合法就可以。传入的参数类型与vector 的元素类型相同。函数的返回值是bool 类型的，如果第一个参数比第二个小就返回true，反之返回 false。上面的例子就是当b小于a时，认为a小于b。所以排序的结果就是将元素按从大到小的顺序排序。

3、接口和抽象的区别

使用abstract class的根本原因在于,人们希望通过这样的方式,表现不同层次的抽象.

而interface的本质是一套协议.在程序设计的发展中,人们又发现接口可以用来表示对行为的抽象,不过,这只是interface的一种用法不是其本质.在接口中包含数据成员.这几乎肯定是错的,因为协议是规范是标准,不应该跟具体实现有任何牵连,也不应该给具体实现造成任何负担.

比如，一家生产门的公司，需要先定义好门的模板，以便能快速生产出各种规格的门。 这里的模板通常会有两类模板：抽象类模板和接口模板。抽象类模板：这个模板里面应该包含所有门都应该具有的共同属性（如，门的形状和颜色等）和共同行为（如，开门和关门）。接口模板：有些门可能需要具有报警和指纹识别等功能，但这些功能又不是所有门必须具有的，所以像这样的行为应该放在单独的接口中。

抽象类用来抽象自然界一些具有相似性质和行为的对象。而接口用来抽象行为的标准和规范，用来告诉接口的实现者必要按照某种规范去完成某个功能。

4、DLL的创建和调用

应用程序使用DLL可以采用两种方式：一种是隐式链接(调用)，另一种是显式链接。显式链接是应用程序在执行过程中随时可以加载DLL文件，也可以随时卸载DLL文件，这是隐式链接所无法作到的，所以显式链接具有更好的灵活性，对于解释性语言更为合适，在使用DLL之前首先要知道DLL中函数的结构信息。VS在VC\bin目录下提供了一个名为Dumpbin.exe的小程序，用它可以查看DLL文件中的函数结构。显式调用的问题：在DLL文件中，dll工程中函数名称在编译生成DLL的过程中发生了变化（C++编译器），在DLL文件中称变化后的字符为“name标示”。GetProcAddress中第二个参数可以由DLL文件中函数的顺序获得，或者直接使用DLL文件中的”name标示”，这个标示可以通过Dumpbin.exe小程序查看。为了让函数名更规范，最常用的方式是：创建dll过程中使用C编译器来编译函数，这样DLL文件中的函数名和原dll工程中的函数名就一致了。隐式链接采用静态加载的方式，比较简单，需要.h、.lib、.dll三件套。动态链接库虽然一定程度上实现了“黑盒复用”，但仍存在着诸多不足，笔者能够想到的有下面几点。dll节省了编译期的时间，但相应延长了运行期的时间，因为在使用dll的导出函数时，不但要加载dll，而且程序将会在模块间跳转，降低了cache的命中率。若采用隐式调用，仍然需要.h、.lib、.dll文件（“三件套”），并不能有效支持模块的更新。显式调用虽然很好地支持模块的更新，但却不能导出类和变量。dll不支持Template。二进制级别的代码复用相比源码级别的复用已经有了很大的进步，但在二进制级别的代码复用中，dll显得太古老。想真正完美实现跨平台、跨语言的黑盒复用，采用COM才是正确的选择。

5、一个由c/C++编译的程序占用的内存分为以下几个部分：栈区（stack）—由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈；堆区（heap）——一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表；全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后有系统释放；文字常量区—常量字符串就是放在这里的。程序结束后由系统释放；程序代码区—存放函数体的二进制代码。

5、如何限制一个类只在堆上分配和栈上分配

在C++中，类的对象建立分为两种，一种是静态建立，如A a；另一种是动态建立，如A\* ptr=new A；这两种方式是有区别的。

静态建立一个类对象，是由编译器为对象在栈空间中分配内存，是通过直接移动栈顶指针，挪出适当的空间，然后在这片内存空间上调用构造函数形成一个栈对象。使用这种方法，直接调用类的构造函数。

动态建立类对象，是使用new运算符将对象建立在堆空间中。这个过程分为两步，第一步是执行operator new()函数，在堆空间中搜索合适的内存并进行分配；第二步是调用构造函数构造对象，初始化这片内存空间。这种方法，间接调用类的构造函数。

所以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。但是如前所述，new还是会调用构造函数，所以不行。转而使用类似单例模式+protected的方法，如下

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。而且protected可以有效解决继承问题，类外无法访问protected成员，子类则可以访问。

对于只能建立在栈上的问题，因为只有使用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

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(){}

};

6、从c++文件到生成exe文件经过哪些步骤

将源代码转换为机器可认识代码的过程。编译程序读取源程序（字符流），对之进行词法和语法的分析，将高级语言指令转换为功能等效的汇编代码，再由汇编程序转换为机器语言，并且按照操作系统对可执行文件格式的要求链接生成可执行程序。

它主要包括以下步骤：源程序－>编译预处理－>编译－>优化程序－>汇编程序－>链接程序－>可执行文件

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Linux\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、系统调用（System Call）是操作系统为在用户态运行的进程与硬件设备（如CPU、磁盘、打印机等）进行交互提供的一组接口。当用户进程需要发生系统调用时，CPU 通过软中断切换到内核态开始执行内核系统调用函数。Linux 下三种发生系统调用的方法：

通过 glibc 提供的库函数；使用 syscall 直接调用，函数原型如下：

long int syscall (long int sysno, ...)

sysno 是系统调用号，每个系统调用都有唯一的系统调用号来标识。在 sys/syscall.h中有所有可能的系统调用号的宏定义。...为剩余可变长的参数，为系统调用所带的参数。返回值，该函数返回值为特定系统调用的返回值，在系统调用成功之后你可以将该返回值转化为特定的类型，如果系统调用失败则返回 -1，错误代码存放在 errno 中。

系统调用和库函数调用的区别

系统调用实际上就是指最底层的一个调用，在linux程序设计里面就是底层调用的意思。面向的是硬件。而库函数调用则面向的是应用开发的，相当于应用程序的api,采用这样的方式有很多种原因，第一：双缓冲技术的实现。第二，可移植性。第三，底层调用本身的一些性能方面的缺陷。第四：让api也可以有了级别和专门的工作面向。

使用库函数也有系统调用的开销，为什么不直接使用系统调用呢？这是因为，读写文件通常是大量的数据（这种大量是相对于底层驱动的系统调用所实现的数据操作单位而言），这时，使用库函数就可以大大减少系统调用的次数。这一结果又缘于缓冲区技术。在用户空间和内核空间，对文件操作都使用了缓冲区，例如用fwrite写文件，都是先将内容写到用户空间缓冲区，当用户空间缓冲区满或者写操作结束时，才将用户缓冲区的内容写到内核缓冲区，同样的道理，当内核缓冲区满或写结束时才将内核缓冲区内容写到文件对应的硬件媒介。

典型的系统调用：chdir fork write brk

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Windows\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、在Windows操作系统中我们常常接触的有三种对象类型：

Windows内核对象 （事件对象，文件对象，进程对象，I/O完成端口对象，互斥量对象，进程对象，线程对象等等）：由执行体（Excutive）对象管理器（Object Manager）管理，内核对象结构体保存在系统内存空间（0x80000000-0xFFFFFFFF），句柄值与进程相关。

Windows GDI对象 （画笔对象，画刷对象等）：由Windows子系统管理，句柄值在系统，会话范围 （system-wide / session-wide） 有效。

Windows USER对象 （窗口对象，菜单对象等） ：由Windows子系统管理，句柄值在系统，会话范围 （system-wide / session-wide） 有效。

2、windows消息

Windows系统预定义了许多消息，每个消息都是一个两个字节的整数，定义为为宏定义只是为了便于记忆和使用。例如#define WM\_MOUSEMOVE 0x0200；Windows是基于消息驱动的，在Windows中发生的一切都可以用消息来表示，消息用于告诉操作系统发生了什么，所有的Windows应用程序都是消息驱动的；一个消息是由消息的名称(UINT)和两个参数（WPARAM, LPARAM）组成。消息的参数中包含有重要的信息。例如对鼠标消息而言，LPARAM中一般包含鼠标的位置信息，而WPARAM参数中包含了发生该消息时，SHIFT、CTRL等键的状态信息，对于不同的消息类型来说，两个参数也都相应地具有明确意义；此外，一个消息必须由一个窗口接收。

举个例子，当双击时，第一次触发LBUTTONDOWN和LBUTTONUP，第二次点击时触发双击事件LBUTTONDBLCLK（doubleclick），放掉再触发LBUTTONUP

3、windows线程创建

windows线程创建时，不会绑定在特定的CPU上，需要手动绑定，或者调用 SetThreadAffinityMask接口进行绑定

\_beginthread 函数可创建在 start\_address 开始执行例程的线程。 start\_address 中的例程必须使用 \_\_cdecl （用于本机代码）或 \_\_clrcall （用于托管代码）调用约定，并且应没有返回值，会在创建线程时主动创建并管理\_tiddata。

传递给 \_beginthreadex 的 start\_address 中的例程必须使用 \_\_stdcall （用于本机代码）或 \_\_clrcall （用于托管代码）调用约定，并且必须返回线程退出代码。

CreateThread仅在运行到需要\_tiddata的函数时才创建，并且在线程结束时不会释放\_tiddata空间，最终造成泄漏

综上，如果在代码中有使用标准C运行库中的函数时，尽量使用\_beginthreadex()来代替CreateThread()，因为\_beginthreadex()比较于CreateThread()有更高的线程安全性，不会造成多个线程共用同一个全局变量的情况

4、Windows程序中设备上下文DC

一个DC实际上是GDI内部保存的数据结构。与特定的显示设备（如显示器或打印机）相关。它定义了一系列图形对象的集合以及它们相关的属性，以及影响输出效果的一些图形模式，这些图形对象包括一个画线的笔，一个填充和painting的画刷，一个用来向屏幕拷贝的位图，一个定义了一系列颜色集合的调色板，一个用来剪裁等操作的区域，一个做painting和drawing操作的路径，GetDC取得与窗口客户区相关的dc，GetWindowDC取得与整个窗口（包括客户区和非客户区）相关的dc。

5、windows进入点函数wWinMain

WinMain 函数的原型声明如下：

int WINAPI WinMain(

HINSTANCE hInstance , // handle to current instance

HINSTANCE hPrevInstance , // handle to previous instance

LPSTR lpCmdLine , // command line

int nCmdShow // show state

);

第一个参数 hInstance 表示该程序当前运行的实例的句柄，这是一个数值。

第二个参数 hPrevInstance 表示当前实例的前一个实例的句柄。

第三个参数 lpCmdLine 是一个以空终止的字符串，指定传递给应用程序的命令行参数。

第四个参数 nCmdShow 指定程序的窗口应该如何显示，例如最大化、最小化、隐藏等。

/\*\*\*\*\*\*\*\*\*\*\*\*20160405补充\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*java\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、==和equals的区别(Integer -128-127 如果超过这个范围会生成一个新的integer object instance，这时候再用==就会得出false的结果，而用equals就还是true的结果)

java中的数据类型，可分为两类：

1.基本数据类型，也称原始数据类型。byte,short,char,int,long,float,double,boolean

他们之间的比较，应用双等号（==）,比较的是他们的值。

2.复合数据类型(类)

当他们用（==）进行比较的时候，比较的是他们在内存中的存放地址，所以，除非是同一个new出来的对象，他们的比较后的结果为true，否则比较后结果为false。 JAVA当中所有的类都是继承于Object这个基类的，在Object中的基类中定义了一个equals的方法，这个方法的初始行为是比较对象的内存地 址，但在一些类库当中这个方法被覆盖掉了，如String,Integer,Date在这些类当中equals有其自身的实现，而不再是比较类在堆内存中的存放地址了。其实也是比较值

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*浏览器\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、移动端浏览器click事件的300ms延迟是因为要等待双击确认，如果300ms后没有再次点击，就确定是单击事件了。解决方案是google fastbutton

2、javascript函数内部有var声明的局部变量，没var声明的是全局变量

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*存储管理\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、内存访问速度通常在50-80ns之间，SSD硬盘访问速度一般是SATA硬盘（毫秒级别）的一千多倍

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*计算机网络\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、ping主要目的就是检测目标主机是否不可连通，实际上就是发送一个ICMP请求报文给目标主机，并等待ICMP应答。ping不通一个地址，不代表这个地址一定不存在或没有连接到网络上，因为对方主机可能做了限制，比如安装了防火墙等等，因此ping不通不代表不能用ftp或者telnet连接。

ping得到的结果包括字节数、反应时间（在ICMP报文中存放发送请求的时间然后在收到的时候就能计算出来了）、生存时间（在IP数据包的头部就能获取）

ICMP是TCP/IP协议族的一个子协议，工作在网络互联层（网络层）。ICMP协议是一种面向无连接的协议，用于传输出错报告控制信息。用于在IP主机、路由器之间传递控制消息。控制消息是指网络通不通、主机是否可达、路由是否可用等网络本身的消息。这些控制消息虽然并不传输用户数据，但是对于用户数据的传递起着重要的作用。

ICMP的一个重要应用就是分组网间探测PING（Packe InterNet Groper），用来测试主机之间的连通性。PING使用了ICMP回送请求与回送回答报文。PING是应用层直接使用网络层ICMP的一个例子，没有经过传输层的TCP或UDP。ping一个域名往往还涉及到DNS和ARP。

参见http://blog.csdn.net/guoweimelon/article/details/50865642

2、tracert用于探测路由，可以看到IP数据包到达目的地经过的路由。IP数据报每经过一个路由就会把TTL值减去1，所以TTL值相当于路由器的计数器。当路由器收到TTL为0的数据报时就直接将其丢弃，同时发送一个ICMP超时信息给源主机。ping和tracert结合起来看TTL就可以判断主机的TTL值。一般来说LINUX主机的TTL值是64或255，而windows为32或128

3、网络层IP数据包格式中有一个协议号字段，传输层如果采用TCP链接，则协议号为6，如果采用UDP，协议号为17

4、传输层通过端口与应用层关联，用netstat可以查看端口号

HTTPS端口号为443；QQ默认端口号为1080/udp

5、HTTP的幂等性

是指一次和多次请求某一个资源应该具有相同的副作用

HTTP GET用于获取资源，不应有副作用，是满足幂等性

DELETE删除资源，有副作用，也应该满足幂等性

POST和PUT都可以用于创建资源，本质差异就在于幂等性，前者不具有，而后者具有，两次相同的POST会创建两份资源，而两次PUT只会创建一份

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*数据库\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、表的主键的每一列必须非空，必须唯一，而且一个表只能定义一个主键

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*进程和线程\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、临界区

用于多线程的互斥访问，进入和离开临界区必须是成对操作，否则共享资源永远得不到释放；临界区中使用的代码最好剪短，减少其他线程的等待时间，提高性能；临界区同步速度很快，但是只能用来同步本进程里面的线程，不能同步多个进程中的线程；临界区是用户态下的对象，非内核对象，使用时无需再用户态和内核态之间切换，效率更高

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*编程基础\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、指针实质上是一个内存地址，32位系统上一个指针占据4字节，64位系统上一个指针占据8字节，这根CPU的寻址有关，32位机上CPU用32位表示一个内存地址，64位机上则用64位表示一个内存地址

2、面向对象设计的五大原则

S 单一功能 对象应该只具有单一功能

O 开闭原则 软件对于扩展开放，对于修改封闭

L 里氏替换原则 程序中的对象可以在不改变程序正确性的前提下被其子类所替换

I 接口隔离原则 多个特定客户端接口要好于一个宽泛用途的接口

D 依赖反转原则 一个方法应该依赖于抽象而不是一个实例

====================================20160415======================================

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*算法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、编程求一篇文章中某个字符串出现次数

string s;

string s\_to\_find;

int pos;

int cnt;

while(pos=s.find\_first\_of(s\_to\_find.c\_str(),pos)!=s.npos()){

cnt++;

pos++;

}

2、阴性阳性遗传

3、100亿个数字中找出最大的1000个

问题在于数据非常多，放不进内存

用小顶堆，首先放入前1000个数字，调整好堆，然后每次读入一个数字，和堆顶比较，如果堆顶更大，则丢弃这个数字，否则弹出堆顶，将新的数字放入堆，调整结构

4、有什么数据结构可以实现每次插入一个数字取出最大的

堆

5、min stack,min deque

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*编程基础\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、OO设计的原则

找名词=》一个名词就是一个类。

2、虚函数的作用：多态、动态联编

虚函数表

多重继承是利用多张虚函数表实现的

参见http://blog.csdn.net/haoel/article/details/1948051

3、new和malloc的区别

new会调用构造函数，对应delete

malloc对应free

待查？？？？？？？？？？？？？？？？

4、什么是原子操作，a+b怎么实现原子操作？？？？？？？？？？？？？？？

原子操作就是指不可中断的一个或一系列操作。

5、//find 函数 stl中的 string头文件下的 返回jk 在s 中的下标位置

//str.find\_first\_of("apple");//返回 apple 中任何一个字符首次在 str 中出现的位置

position = s.find("jk");

if (position != s.npos) //如果没找到，返回一个特别的标志c++中用npos表示，我这里npos取值是4294967295，

{

cout << "position is : " << position << endl;

}

else

{

cout << "Not found the flag"<<endl;

}

十分好用，跟KMP说再见

6、内存泄漏

Windows平台下面VisualStudio调试器和C运行时(CRT)库为我们提供了检测和识别内存泄漏的有效方法，原理大致如下：内存分配要通过CRT在运行时实现，只要在分配内存和释放内存时分别做好记录，程序结束时对比分配内存和释放内存的记录就可以确定是不是有内存泄漏。在vs中启用内存检测的方法如下：

#define \_CRTDBG\_MAP\_ALLOC

#include <stdlib.h>

#include <crtdbg.h>

在添加了上述语句之后，可以通过在程序中包括以下语句（通常应恰好放在程序退出位置之前）来转储内存泄漏信息：

\_CrtDumpMemoryLeaks();

定位内存泄漏的另一种技术涉及在关键点对应用程序的内存状态拍快照。 CRT 库提供一种结构类型 \_CrtMemState，您可用它存储内存状态的快照：\_CrtMemState s1, s2, s3;

若要在给定点对内存状态拍快照，请向 \_CrtMemCheckpoint 函数传递 \_CrtMemState 结构。 该函数用当前内存状态的快照填充此结构：

\_CrtMemCheckpoint( &s1 );

若要确定代码中某一部分是否发生了内存泄漏，可以在该部分之前和之后对内存状态拍快照，然后使用 \_CrtMemDifference 比较这两个状态：

\_CrtMemCheckpoint( &s1 );

// memory allocations take place here

\_CrtMemCheckpoint( &s2 );

if ( \_CrtMemDifference( &s3, &s1, &s2) )

\_CrtMemDumpStatistics( &s3 );

\_CrtMemDifference 比较两个内存状态（s1 和 s2），生成这两个状态之间差异的结果（s3）。 在程序的开始和结尾放置 \_CrtMemCheckpoint 调用，并使用\_CrtMemDifference 比较结果，是检查内存泄漏的另一种方法。 如果检测到泄漏，则可以使用 \_CrtMemCheckpoint 调用通过\*二进制搜索技术\*

来划分程序和定位泄漏。

Linux下面也有原理相同的方法——mtrace，http://en.wikipedia.org/wiki/Mtrace。方法类似，或者用一个非常强大的工具valgrind

7、如何在类中定义static const string

首先

class A {

private:

static const string RECTANGLE;

};

然后

// In one of the implementation files

const string A::RECTANGLE = "rectangle";

8、什么是临时对象？

C++真正的临时对象是不可见的匿名对象，不会出现在你的源码中，但是程序在运行时确实生成了这样的对象.

通常出现在以下两种情况：

1）为了使函数调用成功而进行隐式类型转换的时候。

传递某对象给一个函数，而其类型与函数的形参类型不同时，如果可以通过隐式转化的话可以使函数调用成功，那么此时会通过构造函数生成一个临时对象，当函数返回时临时对象即自动销毁。 注意这样的转化只会出现在两种情况下：函数参数以传值（by value）的方式传递 或者 对象被传递到一个 reference-to-const 参数上。

2）当函数返回对象的时候

当函数返回一个对象时，编译器会生成一个临时对象返回。所以有返回值优化，也就是用 将要被返回值赋值的对象 替代 函数内原本的匿名对象

9、函数返回引用

函数返回值和返回引用是不同的

函数返回值时会产生一个临时变量作为函数返回值的副本，而返回引用时不会产生值的副本，既然是引用，那引用谁呢？这个问题必须清楚，否则将无法理解返回引用到底是个什么概念。以下是几种引用情况：

1）引用函数的参数，当然该参数也是一个引用

const string &shorterString(const string &s1,const string &s2)

{

return s1.size()<s2.size()?s1:s2;

}

2）千万不要返回局部对象的引用

const string &mainip(const string &s)

{

string ret=s;

return ret;

}

当函数执行完毕，程序将释放分配给局部对象的存储空间。此时，对局部对象的引用就会指向不确定的内存。

3）在类的成员函数中，返回引用的类对象，当然不能是函数内定义的类对象（会释放掉），一般为this指向的对象

String& String::operator =(const String &str) //注意与“+”比较，函数为什么要用引用呢？a=b=c，可以做为左值

{

if (this == &str)

{

return \*this;

}

delete [] m\_string;

int len = strlen(str.m\_string);

m\_string = new char[len+1];

strcpy(m\_string,str.m\_string);

return \*this;

}

4）引用返回左值（上例的=赋值也是如此，即a=b=c是可以的）

char &get\_val(string &str,string::size\_type ix)

{

return str[ix];

}

使用语句调用:

string s("123456");

cout<<s<<endl;

get\_val(s,0)='a';

cout<<s<<endl;

10、char的范围在-128-127，unsigned char的范围才是在0-255，因此ASCII值在128-255之间的字符，如果保存为了char型，其转化为int值的范围是在-128--1之间，所以在常见的字符哈希表设计中有，若char\* pcur

while(\*pCur != '\0')

{

if(\*pCur>=0)

hashtable[\*(pCur++)]++;

else

hashtable[\*(pCur++)+256]++;

}

11、extern关键字详解

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。

第一个,当它与"C"一起连用时，如: extern "C" void fun(int a, int b);则告诉编译器在编译fun这个函数名时按着C的规则去翻译相应的函数名而不是C++的

第二，当extern不与"C"在一起修饰变量或函数时，如在头文件中: extern int g\_Int; 它的作用就是声明函数或全局变量的作用范围的关键字，其声明的函数和变量可以在本模块活其他模块中使用，记住它是一个声明不是定义!也就是说B模块(编译单元)要是引用模块(编译单元)A中定义的全局变量或函数时，它只要包含A模块的头文件即可,在编译阶段，模块B虽然找不到该函数或变量，但它不会报错，它会在连接时从模块A生成的目标代码中找到此函数。

extern用在变量声明中常常有这样一个作用，你在\*.c文件中声明了一个全局的变量，这个全局的变量如果要被引用，就放在\*.h中并用extern来声明。例如在test1.h中有下列声明:

#ifndef TEST1H

#define TEST1H

extern char g\_str[]; // 声明全局变量g\_str

void fun1();

#endif

在test1.cpp中

#include "test1.h"

char g\_str[] = "123456"; // 定义全局变量g\_str

void fun1() { cout << g\_str << endl; }

以上是test1模块， 它的编译和连接都可以通过,如果我们还有test2模块也想使用g\_str,只需要在原文件中引用就可以了

#include "test1.h"

void fun2() { cout << g\_str << endl; }

如果函数的声明中带有关键字extern，仅仅是暗示这个函数可能在别的源文件里定义，没有其它作用。即下述两个函数声明没有明显的区别：extern int f(); 和int f();当然，这样的用处还是有的，就是在程序中取代include “\*.h”来声明函数

extern 和 static的关系

(1) extern 表明该变量在别的地方已经定义过了,在这里要使用那个变量.

(2) static 表示静态的变量，分配内存的时候, 存储在静态区,不存储在栈上面.

static 作用范围是内部连接的关系, 和extern有点相反.它和对象本身是分开存储的,extern也是分开存储的,但是extern可以被其他的对象用extern 引用,而static 不可以,只允许对象本身用它. 具体差别首先，static与extern是一对“水火不容”的家伙，也就是说extern和static不能同时修饰一个变量；其次，static修饰的全局变量声明与定义同时进行，也就是说当你在头文件中使用static声明了全局变量后，它也同时被定义了；最后，static修饰全局变量的作用域只能是本身的编译单元，也就是说它的“全局”只对本编译单元有效，其他编译单元则看不到它。一般定义static全局变量时，都把它放在原文件中而不是头文件，这样就不会给其他模块造成不必要的信息污染，同样记住这个原则吧！

extern 和const的关系

C++中const修饰的全局常量据有跟static相同的特性，即它们只能作用于本编译模块中，但是const可以与extern连用来声明该常量可以作用于其他编译模块中, 如extern const char g\_str[]; 然后在原文件中别忘了定义: const char g\_str[] = "123456";

所以当const单独使用时它就与static相同，而当与extern一起合作的时候，它的特性就跟extern的一样了！需要注意的是，const char\* g\_str = "123456" 与 const char g\_str[]="123465"是不同的，前面那个const修饰的是char\*而不是g\_str,它的g\_str并不是常量，它被看做是一个定义了的全局变量（可以被其他编译单元使用）， 所以如果你像让char\*g\_str遵守const的全局常量的规则，最好这么定义const char\* const g\_str="123456".

12、转换构造函数和隐式转换函数

在C/C++中见到过多次标准类型数据间的转换方式了，这种形式用于在程序中将一种指定的数据转换成另一指定的类型，也即是强制转换，比如：int a = int(1.23),其作用是将1.23转换为整形1。然而对于用户自定义的类类型，编译系统并不知道如何进行转换，所以需要定义专门的函数来告诉编译系统改如何转换，这就是转换构造函数和类型转换函数

转换构造函数(conversion constructor function) 的作用是将一个其他类型的数据转换成一个类的对象

当一个构造函数只有一个参数，而且该参数又不是本类的const引用时，这种构造函数称为转换构造函数。

转换构造函数是对构造函数的重载。

例如Complex(double r)

{

real=r;

imag=0;

} 其作用是将double型的参数r转换成Complex类的对象,将r作为复数的实部,虚部为0｡用户可以根据需要定义转换构造函数,在函数体中告诉编译系统怎样去进行转换

不仅可以将一个标准类型数据转换成类对象,也可以将另一个类的对象转换成转换构造函数所在的类对象｡如可以将一个学生类对象转换为教师类对象,可以在Teacher类中写出下面的转换构造函数:

Teacher(Student& s)

{

num=s.num;

strcpy(name,s.name);

sex=s.sex;

} 注意：

1.用转换构造函数可以将一个指定类型的数据转换为类的对象｡但是不能反过来将一个类的对象转换为一个其他类型的数据(例如将一个Complex类对象转换成double类型数据)

2.如果不想让转换构造函数生效，也就是拒绝其它类型通过转换构造函数转换为本类型，可以在转换构造函数前面加上explicit

类型转换函数

用转换构造函数可以将一个指定类型的数据转换为类的对象｡但是不能反过来将一个类的对象转换为一个其他类型的数据(例如将一个Complex类对象转换成double类型数据)｡而类型转换函数就是专门用来解决这个问题的！

类型转换函数的作用是将一个类的对象转换成另一类型的数据｡

如果已声明了一个Complex类,可以在Complex类中这样定义类型转换函数:

operator double( )

{

return real;

}

类型转换函数的一般形式为：

operator 类型名( )

{

实现转换的语句

}

注意事项：

1.在函数名前面不能指定函数类型,函数没有参数｡

2.其返回值的类型是由函数名中指定的类型名来确定的｡

3.类型转换函数只能作为成员函数,因为转换的主体是本类的对象，不能作为友元函数或普通函数｡ 4.从函数形式可以看到,它与运算符重载函数相似,都是用关键字operator开头,只是被重载的是类型名｡double类型经过重载后,除了原有的含义外,还获得新的含义(将一个Complex类对象转换为double类型数据,并指定了转换方法)｡这样,编译系统不仅能识别原有的double型数据,而且还会把Complex类对象作为double型数据处理

13、虚函数&纯虚函数&抽象类&接口&虚基类

在面向对象语言中，接口的多种不同实现方式即为多态。多态是指，用父类的指针指向子类的实例(对象)，然后通过父类的指针调用实际子类的成员函数。多态性就是允许将子类类型的指针赋值给父类类型的指针，多态是通过虚函数实现的。多态可以让父类的指针有“多种形态”，这是一种泛型技术。（所谓泛型技术，就是试图使用不变的代码来实现可变的算法）。C++支持多种形式的多态，从表现的形式来看，有虚函数、模板、重载等，从绑定时间来看，可以分成静态多态和动态多态，也称为编译期多态和运行期多态。

什么是动态多态？

动态多态的设计思想：对于相关的对象类型，确定它们之间的一个共同功能集，然后在基类中，把这些共同的功能声明为多个公共的虚函数接口。各个子类重写这些虚函数，以完成具体的功能。客户端的代码（操作函数）通过指向基类的引用或指针来操作这些对象，对虚函数的调用会自动绑定到实际提供的子类对象上去。从上面的定义也可以看出，由于有了虚函数，因此动态多态是在运行时完成的，也可以叫做运行期多态，这造就了动态多态机制在处理异质对象集合时的强大威力（当然，也有了一点点性能损失）。

什么是静态多态？

静态多态的设计思想：对于相关的对象类型，直接实现它们各自的定义，不需要共有基类，甚至可以没有任何关系。只需要各个具体类的实现中要求相同的接口声明，这里的接口称之为隐式接口。客户端把操作这些对象的函数定义为模板，当需要操作什么类型的对象时，直接对模板指定该类型实参即可（或通过实参演绎获得）。相对于面向对象编程中，以显式接口和运行期多态（虚函数）实现动态多态，在模板编程及泛型编程中，是以隐式接口和编译器多态来实现静态多态。

动态多态和静态多态的比较

静态多态

优点：

由于静多态是在编译期完成的，因此效率较高，编译器也可以进行优化；有很强的适配性和松耦合性，比如可以通过偏特化、全特化来处理特殊类型；最重要一点是静态多态通过模板编程为C++带来了泛型设计的概念，比如强大的STL库。

缺点：

由于是模板来实现静态多态，因此模板的不足也就是静多态的劣势，比如调试困难、编译耗时、代码膨胀、编译器支持的兼容性；不能够处理异质对象集合

动态多态

优点：

OO设计，对是客观世界的直觉认识；实现与接口分离，可复用；处理同一继承体系下异质对象集合的强大威力

缺点：

运行期绑定，导致一定程度的运行时开销；编译器无法对虚函数进行优化；笨重的类继承体系，对接口的修改影响整个类层次；

不同点：

本质不同，静态多态在编译期决定，由模板具现完成，而动态多态在运行期决定，由继承、虚函数实现；动态多态中接口是显式的，以函数签名为中心，多态通过虚函数在运行期实现，静态多台中接口是隐式的，以有效表达式为中心，多态通过模板具现在编译期完成

相同点：

都能够实现多态性，静态多态/编译期多态、动态多态/运行期多态；都能够使接口和实现相分离，一个是模板定义接口，类型参数定义实现，一个是基类虚函数定义接口，继承类负责实现；

定义虚函数的限制

1）非类的成员函数不能定义为虚函数，类的成员函数中静态成员函数和构造函数也不能定义为虚函数，但可以将析构函数定义为虚函数。为什么类的静态成员函数不能为虚函数：如果定义为虚函数，那么它就是动态绑定的，也就是在派生类中可以被覆盖的，这与静态成员函数的定义（在内存中只有一份拷贝，通过类名或对象引用访问静态成员）本身就是相矛盾的。为什么构造函数不能为虚函数：因为如果构造函数为虚函数的话，它将在执行期间被构造，而执行期则需要对象已经建立，构造函数所完成的工作就是为了建立合适的对象，因此在没有构建好的对象上不可能执行多态（虚函数的目的就在于实现多态性）的工作。在继承体系中，构造的顺序就是从基类到派生类，其目的就在于确保对象能够成功地构建。构造函数同时承担着虚函数表的建立，如果它本身都是虚函数的话，如何确保vtbl的构建成功呢？

2）只需要在声明函数的类体中使用关键字“virtual”将函数声明为虚函数，而定义函数时不需要使用关键字“virtual”。

3）如果声明了某个成员函数为虚函数，则在该类中不能出现和这个成员函数同名并且返回值、参数个数、参数类型都相同的非虚函数。在以该类为基类的派生类中，也不能出现这种非虚的同名同返回值同参数个数同参数类型函数。

虚析构函数

C++开发的时候，用来做基类的类的析构函数一般都是虚函数。当基类中有虚函数的时候，析构函数也要定义为虚析构函数。如果不定义虚析构函数，当删除一个指向派生类对象的指针时，会调用基类的析构函数，派生类的析构函数未被调用，造成内存泄露。虚析构函数工作的方式是：最底层的派生类的析构函数最先被调用，然后各个基类的析构函数被调用。这样，当删除指向派生类的指针时，就会首先调用派生类的析构函数，不会有内存泄露的问题了。一般情况下，如果类中没有虚函数，就不用去声明虚析构函数。当且仅当类里包含至少一个虚函数的时候才去声明虚析构函数。只有当一个类被用来作为基类的时候，才把析构函数写成虚函数。

虚函数的实现——虚函数表

虚函数是通过一张虚函数表来实现的，简称V-Table。类的虚函数表是一块连续的内存，每个内存单元中记录一个JUMP指令的地址。编译器会为每个有虚函数的类创建一个虚函数表，该虚函数表将被该类的所有对象共享，类的每个虚函数成员占据虚函数表中的一行。在这个表中，主要是一个类的虚函数的地址表。这张表解决了继承、覆盖的问题，保证其真实反应实际的函数。在有虚函数的类的实例中，分配了指向这个表的指针的内存，所以，当用父类的指针来操作一个子类的时候，这张虚函数表就指明了实际所应该调用的函数。通常,编译器处理虚函数的方法是:给每一个对象添加一个隐藏成员.隐藏成员中保存了一个指向函数地址数组的指针.这种数组称为虚函数表(virtual function table, vtbl).虚函数表中存储了为类对象进行声明的虚函数的地址.例如,基类对象包含一个指针,访指针指向基类中所有虚函数地址表.派生类对象将包含一个指向独立地址表的指针.如果派生类提供了虚岁函数的新定义,访虚函数表将保存新函数的地址;如果派生类没有重新定义虚函数.该vtbl将保存函数原始版本的地址.如果派生类定义了新的虚函数,则该函数的地址也将被添加到vtbl中.注意,无论类中包含的虚函数是1还是10 个,都只需要在对象中添加1个地址成员,只是表的大小不同而已.

调用虚函数时,程序将查看存储在对象中的vtbl地址,然后转向相应的函数地址表.如果使用类声明中定义的第一个虚函数,则程序将使用数组中的第一个函数地址,并执行具有该地址的函数.如果使用类声明中的第三个虚函数,程序将使用地址为数组中第三个元素的函数.简而言之,使用虚函数时,在内存和执行速度方面有一定成本,包括:每个对象都将增大,增大量为存储地址的空间.每个类,编译器都创建一个虚岁函数地址表.每个函数调用都有需要执行一步额外的操作,即到表中查找地址。虽然非虚函数的效率比虚函数稍高,但不具备动态联编功能.

纯虚函数

许多情况下，在基类中不能对虚函数给出有意义的实现，则把它声明为纯虚函数，它的实现留给该基类的派生类去做。纯虚函数的声明格式：virtual <函数返回类型说明符> <函数名> ( <参数表> )=0;纯虚函数的作用是为派生类提供一个一致的接口

抽象类(abstract class)

抽象类是指含有纯虚函数的类(至少有一个纯虚函数)，该类不能创建对象(抽象类不能实例化)，但是可以声明指针和引用，用于基础类的接口声明和运行时的多态。抽象类中，既可以有抽象方法，也可以有具体方法或者叫非抽象方法。一个继承于抽象类的子类，只有实现了父类所有的抽象方法才能够是非抽象类

接口

接口是一个概念。它在C++中用抽象类来实现，在C#和Java中用interface来实现。接口是专门被继承的。接口存在的意义也是被继承。和C++里的抽象类里的纯虚函数是相同的。不能被实例化。定义接口的关键字是interface，例如：

public interface MyInterface{

public void add(int x,int y);

public void volume(int x,int y,int z);

}

继承接口的关键字是implements，相当于继承类的extends。需要注意的是，当继承一个接口时，接口里的所有函数必须全部被覆盖。

当想继承多个类时，开发程序不允许，报错。这样就要用到接口。因为接口允许多重继承，而类不允许（C++中可以多重继承）。所以就要用到接口。

虚基类

在派生类继承基类时，加上一个virtual关键词则为虚拟基类继承，如：

class derive : virtual public base

{

};

虚基类是相对于它的派生类而言的，它本身可以是一个普通的类。只有它的派生类虚继承它的时候，它才称作虚基类，如果没有虚继承的话，就称为基类。比如类B虚继承于类A，那类A就称作类B的虚基类，如果没有虚继承，那类B就只是类A的基类。虚继承主要用于一个类继承多个类的情况，避免重复继承同一个类两次或多次。例如由类A派生类B和类C，类D又同时继承类B和类C，这时候类D就要用虚继承的方式避免重复继承类A两次。

14、常量指针 不能改变指的内容 \*在最后

定义：具有只能够读取内存中数据，却不能够修改内存中数据的属性的指针，称为指向常量的指针，简称常量指针。

声明：const int \* p; int const \* p;

注：可以将一个常量的地址赋值给一个对应类型的常量指针，因为常量指针不能够通过指针修改内粗数据。只能防止通过指针引用修改内存中的数据，并不保护指针所指向的对象。

指针常量 不能改变指的方向 const在最后

定义：指针常量是指指针所指向的位置不能改变，即指针本身是一个常量，但是指针所指向的内容可以改变。

声明：int \* const p=&a;

注：指针常量必须在声明的同时对其初始化，不允许先声明一个指针常量随后再对其赋值，这和声明一般的常量是一样的。

15、友元函数和友元类

友元提供了一种 普通函数或者类成员函数 访问另一个类中的私有或保护成员 的机制。也就是说有两种形式的友元：

（1）友元函数：普通函数对一个访问某个类中的私有或保护成员。

（2）友元类：类A中的成员函数访问类B中的私有或保护成员。

优点：提高了程序的运行效率。

缺点：破坏了类的封装性和数据的透明性。

友元函数

声明和定义

在类声明的任何区域中声明，而定义则在类的外部。

friend <类型><友元函数名>(<参数表>);

注意，友元函数只是一个普通函数，并不是该类的类成员函数，它可以在任何地方调用，友元函数中通过对象名来访问该类的私有或保护成员。

class A

{

public:

A(int \_a):a(\_a){};

friend int getA\_a(A &\_classA);//友元函数

private:

int a;

};

int getA\_a(A &\_classA)

{

return \_classA.a;//通过对象名访问私有变量

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

A \_classA(3);

std::cout<<getA\_a(\_classA);//友元函数只是普通函数，可以在任意地方调用

return 0;

}

友元类

声明和定义

友元类的声明在该类的声明中，而实现在该类外。

friend class <友元类名>;

友元类的实例则在main函数中定义。

class B

{

public:

B(int \_b):b(\_b){};

friend class C;//声明友元类C

private:

int b;

};

class C//实现友元类C

{

public:

int getB\_b(B \_classB){

return \_classB.b;//访问友元类B的私有成员

};

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

B \_classB(3);

C \_classC;//定义友元类实例

std::cout<<\_classC.getB\_b(\_classB);

return 0;

}

注意：友元关系没有继承性：假如类B是类A的友元，类C继承于类A，那么友元类B是没办法直接访问类C的私有或保护成员；友元关系没有传递性：加入类B是类A的友元，类C是类B的友元，那么友元类C是没办法直接访问类A的私有或保护成员，也就是不存在“友元的友元”这种关系

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*进程\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、操作系统的进程调度算法

先来先服务、短作业优先、时间片轮转、优先级调度

linux进程调度？？？？？？？？？？

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*计算机网络\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、为什么TCP挥手比握手多一次

被动关闭方发送ACK和FIN需要两次，发送ACK的时候还有数据没有传完

2、TCP三次握手的漏洞

恶意ip会一直请求连接，服务器发送ack确认，但是永远没有回复，就会导致服务器超载，这就是TCP半开回话的洪水攻击

那么如何解决呢？？？？？？？？？？？？？？

3、TCP|IP分层，如果是四层，就是将物理层和数据链路层看做一层

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*工程\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、安卓、PC摄像头、麦克风同时直播怎么做

2、

=======================20160511补充=====================================

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*编程基础\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、时间复杂度的定义

一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数，用T(n)表示，若有某个辅助函数f(n)，使得当n趋近于无穷大时，T(n)/f(n)的极限值为不等于零的常数，则称f(n)是T(n)的同数量级函数。记作T(n)=O(f(n))，称O(f(n))为算法的渐进时间复杂度(O是数量级的符号 )，简称时间复杂度。

求解步骤可以简化为： 找到执行次数最多的语句；计算语句执行次数的数量级；用大O来表示结果

1) 加法规则

T(n,m) = T1(n) + T2(n) = O (max ( f(n), g(m) )

2) 乘法规则

T(n,m) = T1(n) \* T2(m) = O (f(n) \* g(m))

3) 一个特例（问题规模为常量的时间复杂度）

在大O表示法里面有一个特例，如果T1(n) ＝ O(c)， c是一个与n无关的任意常数，T2(n) = O ( f(n) ) 则有

T(n) = T1(n) \* T2(n) = O ( c\*f(n) ) = O( f(n) )

也就是说，在大O表示法中，任何非0正常数都属于同一数量级，记为O(1)。

4) 一个经验规则

复杂度与时间效率的关系：

c < log2n < n < n\*log2n < n^2 < n^3 < 2^n < 3^n < n! （c是一个常量）

|--------------------------|--------------------------|-------------|

较好 一般 较差

其中c是一个常量，如果一个算法的复杂度为c 、 log2n 、n 、 n\*log2n,那么这个算法时间效率比较高 ，如果是 2n , 3n ,n!,那么稍微大一些的n就会令这个算法不能动了，居于中间的几个则差强人意。

2、C语言各种数据类型在系统中占的字节和取值范围

符号属性 长度属性 基本型 所占位数(bit) 取值范围 输入符举例 输出符举例 备注

-- -- char 8 -2^7 ~ 2^7-1 %c %c、%d、%u char为有符号型，但与signed char是不同的类型。

signed -- char 8 -2^7 ~ 2^7-1 %c %c、%d、%u

unsigned -- char 8 0 ~ 2^8-1 %c %c、%d、%u

[signed] short [int] 16 -2^15 ~ 2^15-1 %hd

unsigned short [int] 16 0 ~ 2^16-1 %hu、%ho、%hx

[signed] -- int 32 -2^31 ~ 2^31-1 %d

unsigned -- [int] 32 0 ~ 2^32-1 %u、%o、%x

[signed] long [int] 32 -2^31 ~ 2^31-1 %ld

unsigned long [int] 32 0 ~ 2^32-1 %lu、%lo、%lx

[signed] long long [int] 64 -2^63 ~ 2^63-1 %I64d %lld %I64d

unsigned long long [int] 64 0 ~ 2^64-1 %I64u、%I64o、%I64x

-- -- float 32 +/- 3.40282e+038 %f、%e指数形式、%g自动形式

-- -- double 64 +/- 1.79769e+308 %lf、%le、%lg %f、%e、%g

-- long double 96 +/- 1.79769e+308 %Lf、%Le、%Lg

如果要处理的数据中存在字节值大于127的情况，使用unsigned char较为妥当。程序中若涉及位运算，也应该使用unsigned型变量。

int的长度，是16位还是32位，与编译器字长有关。16位编译器（如TC使用的编译器）下，int为16位；32位编译器（如VC使用的编译器cl.exe）下，int为32位。

64位整数全解

64位整形引起的混乱主要在两方面，一是数据类型的声明，二是输入输出。

首先是如果我们在自己机器上写程序的话，情况分类如下：

(1) 在linux下的gcc/g++里面，数据类型声明写作

long long a;

输入输出时候用 %lld

(2) 在win下的其它IDE里面[包括高版本Visual Studio]，数据类型声明用上面两种(\_\_int64)均可

输入输出用 %I64d

3、名字空间

名字空间（namespace）是由标准C++引入的，是一种新的作用域级别。原来C++标识符的作用域分为三级：代码块（{…}和函数体）、类域和全局作用域。如今，在类作用域和全局作用域之间，C++标准又添加了名字空间域这一个作用域级别。访问一个具体的标识符的时候，可以使用如下形式：space\_name::identifier。即用作用域指示符“::”将名字空间的名称和该空间下的标识符连接起来，这要，即使使用同名的标识符，由于它们处于不同的名字空间，也不会发生冲突。

有两种形式的命名空间——有名的和无名的。

有名的命名空间：

namespace 命名空间名 {

声明序列可选

}

匿名的命名空间：

namespace {

声明序列可选

}

（1）一个名字空间可以在多个头文件或源文件中实现，成为分段定义。如果想在当前文件访问定义在另一个文件中的同名名字空间内的成员变量，需要在当前文件的名字空间内部进行申明。如标准C++库中的所有组件都是在一个被称为std的名字空间中声明和定义的。这些组件当然分散在不同的头文件和源文件中。

（2）名字空间内部可以定义类型、函数、变量等内容，但名字空间不能定义在类和函数的内部。

（3）在一个名字空间中可以自由地访问另一个名字空间的内容，因为名字空间并没有保护级别的限制。

（4）虽然经常可以见到using namespace std;这样的用法，我们也可以用同样的方法将名字空间中的内容一次性“引入”到当前的名字空间中来，但这并不是一个值得推荐的用法。因为这样做的相当于取消了名字空间的定义，使发生名称冲突的机会增多。所以，用using单独引入需要的内容，这样会更有针对性。例如，要使用标准输入对象，只需用using std::cin;就可以了。

（5）不能在名字空间的定义中声明另一个嵌套的子命名空间，只能在命名空间中定义子命名空间。

（6）名字空间的成员，可以在命名空间的内部定义，也可以在名字空间的外部定义，但是要在名字空间进行声明。

命名空间成员的外部定义的格式为：

名字空间名::成员名 ……

（7）名字空间在进行分段定义时，不能定义同名的变量，否则连接出现重定义错误。因为名字空间不同于类，具有外部连接的特性。由于外部连接特性，请不要将名字空间定义在头文件，因为当被不同的源文件包含时，会出现重定义的错误。

（8）为了避免命名空间的名字与其他的命名空间同名，可以用较长的标识符作为命名空间的名字。但是书写较长的命名空间名时，有些冗余，因此，我们可以在特定的上下文环境中给命名空间起一个相对简单的别名。

namespace MyNewlyCreatedSpace{

void show(){

std::cout<<"a function within a namespace"<<std::endl;

}

}

int main(int argc,char\* argv[])

{

namespace sp=MyNewlyCreatedSpace;

sp::show();

}

匿名名字空间提供了类似在全局函数前加 static 修饰带来的限制作用域的功能。它的这种特性可以被用在struct和class上， 而普通的static却不能。比如，在两个源文件中定义了相同的全局变量（或函数），就会发生重定义的错误。如果将它们声明为全局静态变量（函数）就可以避免重定义错误。在C++中，除了可以使用static关键字避免全局变量（函数）的重定义错误，还可以通过匿名名字空间的方式实现。

//main.cpp

#include <iostream>

using namespace std;

namespace{

double dvar=1.8;

}

void show1(){

cout<<"dvar:"<<dvar<<endl;

}

int main(int argc,char\* argv[])

{

void show2();

show1();

show2();

}

//a.cpp

#include <iostream>

using namespace std;

double dvar=2.8;

void show2(){

cout<<"dvar:"<<dvar<<endl;

}

未命名的名字空间中定义的变量（或函数）只在包含该名字空间的文件中可见，但其中的变量的生存期却从程序开始到程序结束。如果有多个文件包含未命名的名字空间，这些名字空间是不相关的，即使这些名字空间中定义了同名的变量（函数），这些标识符也代表不同的对象。

但是，用static修饰的变量（函数）具有内部连接特性，而具有内部连接特性的变量（函数）是不能用来实例化一个模板的。参考如下程序。

#include <iostream>

using namespace std;

template <char\*p> class Example{

public:

void display(){

cout<<\*p<<endl;

}

};

static char c='a';

int main(int argc,char\* argv[])

{

Example<&c> a; //编译出错

a.display();

}

通过以上程序，可以看出匿名名字空间与static的区别：包含在匿名名字空间中的全局变量（函数）具有外部连接特性，而用static修饰的全局变量具有内部连接特性，不能用来实例化模板的非类型参数。

4、static关键字

C 语言的 STATIC 关键字的两种用法

用于函数内部修饰变量，即函数内的静态变量。这种变量的生存期长于该函数，使得函数具有一定的“状态”。使用静态变量的函数一般是不可重入的，也不是线程安全的。

用在文件级别（函数体之外），修饰变量或函数，表示该变量或函数只在本文件可见，其他文件看不到也访问不到该变量或函数。即具有 internal linkage

C++ 语言的 STATIC 关键字的四种用法

C++继承了C的static语法的同时，引入了另外两种用法：

用于修饰 class 的数据成员，即所谓“静态成员”。这种数据成员的生存期大于 class 的对象（实体 instance）。静态数据成员是每个 class 普通数据成员是每个 instance 有一份。

用于修饰 class 的成员函数，即所谓“静态成员函数”。这种成员函数只能访问 class variable 和其他静态程序函数，不能访问 instance variable 或 instance method。

5、将一个函数定义为纯虚函数，实际上是将这个类定义为抽象类，不能实例化对象。 纯虚函数通常没有定义体，但也完全可以拥有。 析构函数可以是纯虚的，但纯虚析构函数必须有定义体，因为析构函数的调用是在子类中隐含的。

虚表的地址存在对象内存空间里的头4个字节

6、map的排序

首先map本身会按照key的升序来排序；但是我们看一下map的定义

template < class Key, class T, class Compare = less<Key>,

class Allocator = allocator<pair<const Key,T> > > class map;

跟sort类似，有一个compare，可以选择如何排序，先来看一下less的实现，是一个函数对象

template <class T> struct less : binary\_function <T,T,bool> {

bool operator() (const T& x, const T& y) const

{return x<y;}

};类似的还有

template <class T> struct greater : binary\_function <T,T,bool> {

bool operator() (const T& x, const T& y) const

{return x>y;}

};所以我们也可以自己写一个

struct CmpByKeyLength {

bool operator()(const string& k1, const string& k2) {

return k1.length() < k2.length();

}

};相应的定义map时要这样

map<string, int, CmpByKeyLength> name\_score\_map;

以上都是如何按照key排序，下面来看一下如何按照value排序

首先不能用sort，sort只能对线性容器管用，但也给我们带来一个思路，就是先把map存到序列容器vector中，再用sort排序，同时再实现比较，又已知map里面就是pair，pair本身有一个比较

template<class \_T1, class \_T2>

inline bool

operator<(const pair<\_T1, \_T2>& \_\_x, const pair<\_T1, \_T2>& \_\_y)

{ return \_\_x.first < \_\_y.first

|| (!(\_\_y.first < \_\_x.first) && \_\_x.second < \_\_y.second); }

不能直接重载，改造一下就有

typedef pair<string, int> PAIR;

bool cmp\_by\_value(const PAIR& lhs, const PAIR& rhs) {

return lhs.second < rhs.second;

}

或

struct CmpByValue 这是一个函数对象{

bool operator()(const PAIR& lhs, const PAIR& rhs) {

return lhs.second < rhs.second;

}

};

最后就可以如此实现

int main() {

map<string, int> name\_score\_map;

name\_score\_map["LiMin"] = 90;

name\_score\_map["ZiLinMi"] = 79;

name\_score\_map["BoB"] = 92;

name\_score\_map.insert(make\_pair("Bing",99));

name\_score\_map.insert(make\_pair("Albert",86));

//把map中元素转存到vector中

vector<PAIR> name\_score\_vec(name\_score\_map.begin(), name\_score\_map.end());

sort(name\_score\_vec.begin(), name\_score\_vec.end(), CmpByValue());//注意这里的括号

// sort(name\_score\_vec.begin(), name\_score\_vec.end(), cmp\_by\_value);//而这里没有

for (int i = 0; i != name\_score\_vec.size(); ++i) {

cout << name\_score\_vec[i] << endl;

}

return 0;

}

7、指针和引用都是地址的概念；指针指向一块内存，它的内容是所指内存的地址；引用是某块内存的别名。

(1)引用总是指向一个对象,没有所谓的 null reference .所有当有可能指向一个对象也由可能不指向对象则必须使用指针. 由于没有所谓的 null reference 所以所以在使用前不需要进行测试其是否有值.,而使用指针则需要测试其的有效性. 这个事实意味着使用引用的代码效率比使用指针的要高。

(2)指针可以被重新赋值而reference则总是指向最初的对象.

(3)必须使用reference的场合. Operator[] 操作符 由于该操作符很特别地必须返回 [能够被当做assignment 赋值对象] 的东西,所以需要给他返回一个 reference.

(4)其实引用在函数的参数中使用很经常.

void Get\*\*\*(const int& a) //这样使用了引用有可以保证不修改被引用的值

(5)“sizeof 引用”得到的是所指向的变量(对象)的大小，而“sizeof 指针”得到的是指针本身(所指向的变量或对象的地址)的大小；typeid(T) == typeid(T&) 恒为真，sizeof(T) == sizeof(T&) 恒为真

8、assert宏的原型定义在<assert.h>中，其作用是如果它的条件返回错误，则终止程序执行，原型定义：

#include <assert.h>

void assert( int expression );

assert的作用是现计算表达式 expression ，如果其值为假（即为0），那么它先向stderr打印一条出错信息，

然后通过调用 abort 来终止程序运行。

使用assert的缺点是，频繁的调用会极大的影响程序的性能，增加额外的开销。

在调试结束后，可以通过在包含#include <assert.h>的语句之前插入 #define NDEBUG 来禁用assert调用，示例代码如下：

#include <stdio.h>

#define NDEBUG

#include <assert.h>

在Release下assert不会起作用

9、debug和release的区别

Debug通常称为调试版本，它包含调试信息，并且不作任何优化，便于程序员调试程序。Release称为发布版本，它往往是进行了各种优化，使得程序在代码大小和运行速度上都是最优的，以便用户很好地使用。

Debug 和 Release 的真正区别，在于一组编译选项。

Debug 版本

参数 含义

/MDd /MLd 或 /MTd 使用 Debug runtime library(调试版本的运行时刻函数库)

/Od 关闭优化开关

/D "\_DEBUG" 相当于 #define \_DEBUG,打开编译调试代码开关(主要针对assert函数)

/ZI

创建 Edit and continue(编辑继续)数据库，这样在调试过程中如果修改了源代码不需重新编译

GZ 可以帮助捕获内存错误

Release 版本 参数含义

/MD /ML 或 /MT 使用发布版本的运行时刻函数库

/O1 或 /O2 优化开关，使程序最小或最快

/D "NDEBUG" 关闭条件编译调试代码开关(即不编译assert函数)

/GF 合并重复的字符串，并将字符串常量放到只读内存，防止被修改

Debug 和 Release 并没有本质的界限，他们只是一组编译选项的集合，编译器只是按照预定的选项行动。

一些相关的人生经验

变量：debug跟release在初始化变量时所做的操作是不同的，debug是将每个字节位都赋成0xcc(注1)，而release的赋值近 似于随机(我想是直接从内存中分配的，没有初始化过)。这样就明确了，如果你的程序中的某个变量没被初始化就被引用，就很有可能出现异常：用作控制变量将 导致流程导向不一致；用作数组下标将会使程序崩溃；更加可能是造成其他变量的不准确而引起其他的错误。所以在声明变量后马上对其初始化一个默认的值是最简 单有效的办法，否则项目大了你找都没地方找。代码存在错误在debug方式下可能会忽略而不被察觉到，如debug方式下数组越界也大多不会出错，在 release中就暴露出来了，这个找起来就比较难了

自定义消息的消息参数：MFC为我们提供了很好的消息机制，更增加了自定义消息，好处我就不用多说了。这也存在debug跟release的问题吗？答案是肯定的。在自定义消息 的函数体声明时，时常会看到这样的写法：afx\_msg LRESULT OnMessageOwn(); Debug情况下一般不会有任何问题，而当你在Release下且多线程或进程间使用了消息传递时就会导致无效句柄之类的错误。导致这个错误直接原因是消 息体的参数没有添加，即应该写成：afx\_msg LRESULT OnMessageOwn(WPARAM wparam, LPARAM lparam);

release模式下不出错，但debug模式下报错：这种情况下大多也是因为代码书写不正确引起的，查看MFC的源码，可以发现好多ASSERT的语句(断言)，这个宏只是在debug模式下才有效，那么就 清楚了，release版不报错是忽略了错误而不是没有错误，这可能存在很大的隐患，因为是Debug模式下，比较方便调试，

数据溢出的问题：

如：char buffer[10];

int counter;

strcpy(buffer, "abcdefghik");

在debug版中buffer的NULL覆盖了counter的高位，但是除非counter>16M,什么问题也没有。但是在release版 中，counter可能被放在寄存器中，这样NULL就覆盖了buffer下面的空间，可能就是函数的返回地址，这将导致ACCESS ERROR。

DEBUG版和RELEASE版的内存分配方式是不同的：如果你在DEBUG版中申请ele为6\*sizeof(DWORD)=24bytes,实际上分配给你的是32bytes（debug版以32bytes为单位分配）， 而在release版，分配给你的就是24bytes（release版以8bytes为单位），所以在debug版中如果你写ele[6],可能不会有 什么问题，而在release版中，就有ACCESS VIOLATE。

10、怎样“调试” Release 版的程序

前面已经提过，Debug和Release只是一组编译选项的差别，实际上并没有什么定义能区分二者。我们可以修改Release版的编译选项来缩小错误范围。如上所述，可以把Release 的选项逐个改为与之相对的Debug选项，如/MD改为/MDd、/O1改为/Od，或运行时间优化改为程序大小优化。注意，一次只改一个选项，看改哪个选项时错误消失，再对应该选项相关的错误，针对性地查找。这些选项在Project\Settings...中都可以直接通过列表选取，通常不要手动修改。由于以上的分析已相当全面，这个方法是最有效的。在编程过程中就要时常注意测试 Release 版本，以免最后代码太多，时间又很紧。在 Debug 版中使用 /W4 警告级别，这样可以从编译器获得最大限度的错误信息，比如 if( i =0 )就会引起 /W4 警告。不要忽略这些警告，通常这是你程序中的 Bug 引起的。

11、开发的程序在别人电脑不能运行的问题

程序，运行的时候还需要一些dll动态库的支持，比如MFC42.这些动态库没有安张VC是没有的。所以要有这些dll才能运行你的程序。

建议你使用release编译你的程序，这样依赖dll最少。

如果是MFC工程在工程配置-General中选择MFC static library，再编译程序

建议你使用安装程序，setup。安装程序打包可以把这些dll自动放进去，方便发布程序。

12、运行时库

运行时库是程序在运行时所需要的库文件，通常运行时库是以LIB或DLL形式提供的。

Visual C++提供了两种版本的C运行时库。一个版本供单线程应用程序调用，另一个版本供多线程应用程序调用。多线程运行时库与单线程运行时库有两个重大差别：

　　（1）类似errno的全局变量，每个线程单独设置一个；这样从每个线程中可以获取正确的错误信息。

　　（2）多线程库中的数据结构以同步机制加以保护。这样可以避免访问时候的冲突。

Visual C++提供的多线程运行时库又分为静态链接库和动态链接库两类，而每一类运行时库又可再分为debug版和release版，因此Visual C++共提供了6个运行时库。如下表：

C运行时库 库文件

Single thread(static link) libc.lib

Debug single thread(static link) libcd.lib

MultiThread(static link) libcmt.lib

Debug multiThread(static link) libcmtd.lib

MultiThread(dynamic link) msvert.lib

Debug multiThread(dynamic link) msvertd.lib

C运行时库除了给我们提供必要的库函数调用（如memcpy、printf、malloc等）之外，它提供的另一个最重要的功能是为应用程序添加启动函数。C运行时库启动函数的主要功能为进行程序的初始化，对全局变量进行赋初值，加载用户程序的入口函数。

各种C运行时库的区别：

静态链接的单线程库：静态链接的单线程库只能用于单线程的应用程序，C运行时库的目标代码最终被编译在应用程序的二进制文件中。通过/ML编译选项可以设置Visual C++使用静态链接的单线程库。

静态链接的多线程库：静态链接的多线程库的目标代码也最终被编译在应用程序的二进制文件中，但是它可以在多线程程序中使用。通过/MD /MT编译选项可以设置Visual C++使用静态链接的单线程库。

动态链接的运行时库：动态链接的运行时库将所有的C库函数保存在一个单独的动态链接库MSVCRTxx.DLL中，MSVCRTxx.DLL处理了多线程问题。使用/ML编译选项可以设置Visual C++使用动态链接的运行时库。

/MDd、 /MLd 或 /MTd 选项使用 Debug runtime library(调试版本的运行时刻函数库)，与/MD、 /ML 或 /MT分别对应。Debug版本的 Runtime Library 包含了调试信息，并采用了一些保护机制以帮助发现错误，加强了对错误的检测，因此在运行性能方面比不上Release版本。

如果不使用运行库，每个程序中都会包括很多重复的代码，而使用运行库，可以大大缩小编译后的程序的大小。但另一方面，由于使用了运行库，所以在分发程序时就必须带有这些库，比较麻烦。如果在操作系统中找不到相应的运行库程序就无法运行。为了解决这个矛盾，Windows总是会带上它自己开发的软件的最新的运行库。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*操作系统\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、带权周转时间是指作业周转时间与作业实际运行时间的比值：

2、实现虚拟存储器最关键的技术是请求调页（段）

3、DMA

DMA方式，Direct Memory Access，也称为成组数据传送方式，有时也称为直接内存操作。DMA方式的主要优点是速度快。一个设备接口试图通过总线直接向另一个设备发送数据(一般是大批量的数据)，

它会先向CPU发送DMA请求信号。外设通过DMA的一种专门接口电路――DMA控制器（DMAC），向CPU提出接管总线控制权的总线请求，CPU收到该信号后，在当前的总线周期结束后，会按DMA信号的优先级和提出DMA请求的先后顺序响应DMA信号。CPU对某个设备接口响应DMA请求时，会让出总线控制权。于是在DMA控制器的管理下，外设和存储器直接进行数据交换，而不需CPU干预。数据传送完毕后，设备接口会向CPU发送DMA结束信号，交还总线控制权。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*计算机网络\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、VLAN

VLAN（Virtual Local Area Network，虚拟局域网）技术的出现，主要为了解决交换机在进行局域网互连时无法限制广播的问题。这种技术可以把一个LAN划分成多个逻辑的LAN——VLAN，每个VLAN是一个广播域，VLAN内的主机间通信就和在一个LAN内一样，而VLAN间则不能直接互通，这样，广播报文被限制在一个VLAN内。

【VLAN的优点】

限制广播域。广播域被限制在一个VLAN内，节省了带宽，提高了网络处理能力；增强局域网的安全性。不同VLAN内的报文在传输时是相互隔离的，即一个VLAN内的用户不能和其它VLAN内的用户直接通信，如果不同VLAN要进行通信，则需要通过路由器或三层交换机等三层设备；灵活构建虚拟工作组。用VLAN可以划分不同的用户到不同的工作组，同一工作组的用户也不必局限于某一固定的物理范围，网络构建和维护更方便灵活。

它在以太网帧的基础上增加了VLAN头，用VLAN ID把用户划分为更小的工作组。

【VLAN的划分】

根据端口来划分VLAN；根据MAC地址划分VLAN；根据网络层划分VLAN；根据IP组播划分VLAN；基于规则的VLAN

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*数据结构\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、线性表

线性表是最基本、最简单、也是最常用的一种数据结构。线性表中数据元素之间的关系是一对一的关系，即除了第一个和最后一个数据元素之外，其它数据元素都是首尾相接的（注意，这句话只适用大部分线性表，而不是全部。比如，循环链表逻辑层次上也是一种线性表（存储层次上属于链式存储），但是把最后一个数据元素的尾指针指向了哨位结点）。

我们说“线性”和“非线性”，只在逻辑层次上讨论，而不考虑存储层次，所以双向链表和循环链表依旧是线性表。在数据结构逻辑层次上细分，线性表可分为一般线性表和受限线性表。一般线性表也就是我们通常所说的“线性表”，可以自由的删除或添加结点。受限线性表主要包括栈和队列，受限表示对结点的操作受限制。线性表的逻辑结构简单，便于实现和操作。因此，线性表这种数据结构在实际应用中是广泛采用的一种数据结构。在实际应用中，线性表都是以栈、队列、字符串等特殊线性表的形式来使用的。

下列哪个不是线性表（D）

A. 链表 B. 队列 C.栈 D.关联数组

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*20160606待刷题目\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、数据结构期末题目

http://www.cnblogs.com/xingyunblog/p/4229958.html第七题

2、线程通信

http://www.cnblogs.com/tocy/archive/2012/06/03/2532559.html

https://msdn.microsoft.com/en-us/library/ms686301(v=vs.85) sleepcondition

3、libxml

http://www.blogjava.net/wxb\_nudt/archive/2007/11/18/161340.html

4、ostream stringstream

5、java: wait notify notifyall

http://www.importnew.com/16453.html

6、虚函数const=0

7、移植librtmp

8、多线程 多核

9、几道算法题

还原字符串 排序数组循环移位后查找 无序数组，大小相邻的最大间隔

11、什么是原子操作 c语言

12、combination sum

<https://leetcode.com/problems/combination-sum/>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*安卓\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、android进程间通信

２．安卓如何传递对象？

使用intent，两种方式.

一个是serializable方式，序列化，将一个对象转换成可存储或可传输的状态。序列化之后的对象可以在网络上进行传输也可以在本地存储。只要让一个类去实现serializable接口即可。传入方法很简单，还是intent.putExtra("person\_data",person),获取方法稍有不同，是(Person)getIntent().getSerializableExtra("person\_data").

另一个是Parcelable方式，他不是做序列化，而是把一个完整的对象进行分解，分解后的每一部分都是Intent支持的数据类型，从而可以实现传递。首先还是要让类去实现Parcelable接口，然后主要重写writeToParcel方法，调用writeXxx方法将类的字段一一写出。然后再在类中提供一个Creator常量，其实就是实现Parcelable.Creator接口的一个实现，重写createFromParcel和newArray方法，在前者中读取刚才写出的字段，注意读取的顺序要和写入的顺序完全相同。最后的获取代码就是(Person)getIntent().getParcelableExtra("person\_data").

第一种方法实现简单但是需要把整个对象序列化，效率更低一些。

３．安卓有哪些动画，view动画和属性动画的区别

view动画（类似flash的补间动画，比如rotate，scale等），drawable动画（像幻灯片一样）和属性动画．View动画只能够为View添加动画，如果想为非View对象添加动画须自己实现；且View动画支持的种类很少；*尤其是他改变的是View的绘制效果，View的属性没有改变，其位置与大小都不变；* View动画代码量少，使用简单方便；属性动画弥补了View动画的缺陷，你可以为一个对象的任意属性添加动画，对象自己的属性会被真的改变；当对象的属性变化的时候，属性动画会自动刷新屏幕；属性动画改变的是对象的真实属性，而且属性动画不止用于View，还可以用于任何对象。

４．哪些安卓进程不能被杀死？App怎么防止被强杀？

安卓进程的优先级排序：

前台进程：拥有用户正在交互的 Activity；拥有某个 Service，后者绑定到用户正在交互的 Activity； 拥有正在“前台”运行的 Service（服务已调用 startForeground()）；拥有正执行一个生命周期回调的 Service（onCreate()、onStart() 或 onDestroy()）；拥有正执行其 onReceive() 方法的 BroadcastReceiver

可见进程：拥有不在前台、但仍对用户可见的 Activity（已调用 onPause())；拥有绑定到可见（或前台）Activity 的 Service

服务进程：正在运行 startService() 方法启动的服务，且不属于上述两个更高类别进程的进程

后台进程：对用户不可见的 Activity 的进程（已调用 Activity的onStop() 方法）

空进程：保留这种进程的的唯一目的是用作缓存，以缩短下次在其中运行组件所需的启动时间

Android 中对于内存的回收，主要依靠 Lowmemorykiller 来完成，是一种根据 OOM\_ADJ 阈值级别触发相应力度的内存回收的机制。理论上没有不能杀死的进程，但是前台进程＼可见进程＼系统进程＼native进程的优先级最高，后两者非安卓进程，是linux进程

如何防止被强杀：

４．１利用 Activity 提升权限：监控手机锁屏解锁事件，在屏幕锁屏时启动1个像素的 Activity，在用户解锁时将 Activity 销毁掉。注意该 Activity 需设计成用户无感知。通过该方案，可以使进程的优先级在屏幕锁屏时间由4提升为最高优先级1。

４．２利用 Notification 提升权限：通过实现一个内部 Service，在 LiveService 和其内部 Service 中同时发送具有相同 ID 的 Notification，然后将内部 Service 结束掉。随着内部 Service 的结束，Notification（前台服务会和一个notificaion绑定在一起） 将会消失，但系统优先级依然保持为2。

进程被杀死后如何复活

４．３利用响应系统广播事件来复活

４．４将service设置为Start\_Sticky，系统会自动在service挂掉后拉活，但是如果短时间被杀死多次则不再复活

５．ANR几秒

Android系统会监控程序的响应状况，一旦出现下面两种情况，则弹出ANR对话框：应用在5秒内未响应用户的输入事件（如按键或者触摸），其实就是UI线程阻塞５秒；BroadcastReceiver未在10秒内完成相关的处理，其实就是广播阻塞时间超过１０秒

如何避免

基本的思路就是将IO操作在工作线程来处理，减少其他耗时操作和错误操作：使用[AsyncTask](http://droidyue.com/blog/2014/11/08/bad-smell-of-asynctask-in-android/)处理耗时IO操作；使用Thread或者HandlerThread时，调用Process.setThreadPriority(Process.THREAD\_PRIORITY\_BACKGROUND)设置优先级，否则仍然会降低程序响应，因为默认Thread的优先级和主线程相同；使用[Handler](http://droidyue.com/blog/2014/12/28/in-android-handler-classes-should-be-static-or-leaks-might-occur/)处理工作线程结果，而不是使用Thread.wait()或者Thread.sleep()来阻塞主线程；Activity的onCreate和onResume回调中尽量避免耗时的代码；BroadcastReceiver中onReceive代码也要尽量减少耗时，建议使用IntentService处理

如果开发机器上出现问题，我们可以通过查看/data/anr/traces.txt即可，最新的ANR信息在最开始部分。我们从stacktrace中即可找到出问题的具体行数

有的时候实际上已经发生了ANR，只是没有进行对话框弹出而已。这种ANR就是background ANR，即后台程序的ANR，我们可以通过过滤日志验证．通过在开发者选项中打开显示所有应用程序无响应即可看到了

安卓提供了异步消息处理机制的两种方式来解决线程之间的通信问题：Handler（Thread 负责子线程的耗时操作，Handler 负责线程间的通信）和AsyncTask（ Android 为了简化操作，提供了 AsyncTask 类来实现异步任务，并且轻松实现子线程和主线程间的通信）

写在前面：在Android的多线程编程当中，有两条非常重要的原则必须遵守：不能在UI Thread当中进行耗时操作，以免阻塞UI Thread;不能在UI Thread之外的线程当中操纵UI元素

６．Handler:可以在子线程中创建

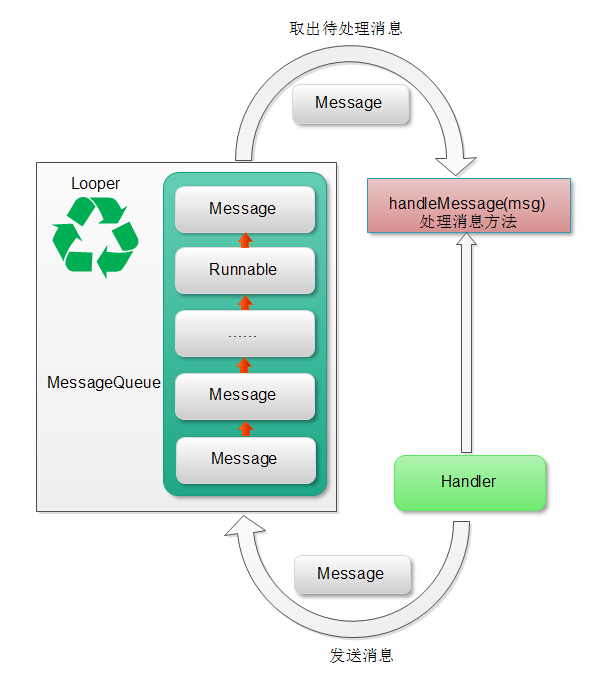
如果在子线程中尝试进行UI操作，程序就有可能会崩溃。相信大家在日常的工作当中都会经常遇到这个问题，解决的方案应该也是早已烂熟于心，即创建一个Message对象，然后借助Handler发送出去，之后在Handler的handleMessage()方法中获得刚才发送的Message对象，然后在这里进行UI操作就不会再出现崩溃了

|  |
| --- |
| private Handler handler = new Handler() {  public void handleMessage(Message msg) {  switch (msg.what) {  case UPDATE\_TEXT:  text.setText("Nice to meet you");  break;  default:  break;  }  }  };  ……  new Thread(new Runnable() {  @Override  public void run() {  Message message = handler.obtainMessage();  message.what = UPDATE\_TEXT;  handler.sendMessage(message);  }  }).start(); |

如果从源码的角度理解，粗略的描述是这样的：

先是调用了handler的obtainMessage()方法得到Message对象。在obtainMessage()方法里做的事情是：调用了Message.obtain(this)方法，把handler作为对象传进来。在Message.obtain(this)方法里做的事情是：生成message对象，把handler作为参数赋值给message的target属性。总的来说，一个Handler对应一个Looper对象，一个Looper对应一个MessageQueue对象，使用Handler生成Message，所生成的Message对象的Target属性，就是该对象。而一个Handler可以生成多个Message，所以说，Handler和Message是一对多的关系。

Handler是Android类库提供的用于接受、传递和处理消息或Runnable对象的处理类，它结合Message、MessageQueue和Looper类以及当前线程实现了一个消息循环机制，用于实现任务的异步加载和处理。整个异步消息处理流程的示意图如下图所示：

* MessageQueue：用于存放Message或Runnable对象的消息队列。它由对应的Looper对象创建，并由Looper对象管理。每个线程中都只会有一个MessageQueue对象。
* Looper：是每个线程中的MessageQueue的管家，循环不断地管理MessageQueue接收和分发Message或Runnable的工作。调用Looper的loop()方法后，就会进入到一个**无限循环**中然后每当发现MessageQueue中存在一条消息，就会将它取出，并调用Handler的handlerMessage()方法。每个线程中也只会有一个Looper对象。

一般我们在实际的开发过程中用的比较多一种情况的就是主线程的Handler将子线程中处理过的耗时操作的结果封装成Message(消息),并将该Message(利用主线程里的MessageQueue和Looper)传递到主线程中，最后主线程再根据传递过来的结果进行相关的UI元素的更新，从而实现任务的异步加载和处理，并达到线程间的通信。

Handler类的两个主要用途：

(1)执行定时任务：

指定任务时间，在某个具体时间或某个时间段后执行特定的任务操作，例如使用Handler提供的postDelayed(Runnable r,long delayMillis)方法指定在多久后执行某项操作

(2)线程间的通信

Handler的post

对于Handler的Post方式来说，它会传递一个Runnable对象到消息队列中（这句话稍后会进行详细解释），在这个Runnable对象中，重写run()方法。一般在这个run()方法中写入需要在UI线程上的操作。Post允许把一个Runnable对象入队到消息队列中。它的方法有：post(Runnable)、postAtTime(Runnable,long)、postDelayed(Runnable,long)。详细解释如下：

* boolean post(Runnable r)：把一个Runnable入队到消息队列中，UI线程从消息队列中取出这个对象后，立即执行。
* boolean postAtTime(Runnable r,long uptimeMillis)：把一个Runnable入队到消息队列中，UI线程从消息队列中取出这个对象后，在特定的时间执行。
* boolean postDelayed(Runnable r,long delayMillis)：把一个Runnable入队到消息队列中，UI线程从消息队列中取出这个对象后，延迟delayMills秒执行
* void removeCallbacks(Runnable r)：从消息队列中移除一个Runnable对象。

|  |
| --- |
| public void onClick(View v) {  new Thread(new Runnable() {  @Override  public void run() {  // 使用postDelayed方式修改UI组件tvMessage的Text属性值  // 并且延迟3S执行  handler.postDelayed(new Runnable() {    @Override  public void run() {这里的代码，其实运行在主线程上，虽然写在子线程中  tvMessage.setText("使用Handler.postDelayed在工作线程中发送一段执行到消息队列中，在主线程中延迟3S执行。");    }  }, 3000);  }  }).start();  } |

使用Post()方法的好处在于：避免了在主线程和子线程中将数据传来传去的麻烦。

对于Message对象，一般并不推荐直接使用它的构造方法得到，而是建议通过使用Message.obtain()这个静态的方法或者Handler.obtainMessage()获取。Message.obtain()会从消息池中获取一个Message对象，如果消息池中是空的，才会使用构造方法实例化一个新Message，这样有利于消息资源的利用。并不需要担心消息池中的消息过多，它是有上限的，上限为10个。Handler.obtainMessage()具有多个重载方法，如果查看源码，会发现其实Handler.obtainMessage()在内部也是调用的Message.obtain()。

前面都是子线程给主线程发送消息，也可以从主线程往子线程发送消息

|  |
| --- |
| //当用户点击按钮时，发送Message的对象msg  button.setOnClickListener(new OnClickListener() { //使用匿名内部类为button绑定监听器    @Override  public void onClick(View v) {  Log.i("onClick:", Thread.currentThread().getName());  Message msg = handler.obtainMessage() ;  handler.sendMessage(msg) ;  }  }) ;    WorkerThread wt = new WorkerThread() ;  wt.start() ;  ……  //在WorkerThread生成handler  class WorkerThread extends Thread {  @Override  public void run() {  //准备Looper对象，必须要先有这一步  Looper.prepare() ;  //在WorkerThread当中生成一个Handler对象  handler = new Handler() {  @Override  public void handleMessage(Message msg) {  Log.i("handleMessage:", Thread.currentThread().getName());  Log.i("后台输出", "收到了消息对象");这里是在子线程中运行的  }  };  //调用Looper的loop()方法之后，Looper对象将不断地从消息队列当中取出对象，然后调用handler的handleMessage()方法，处理该消息对象  //如果消息队列中没有对象，则该线程阻塞  Looper.loop() ; //通过Looper对象将消息取出来  }    } |

首先执行Looper的prepare()方法，这个方法有两个作用：一是生成Looper对象，二是把Looper对象和当前线程对象形成键值对（线程为键），存放在ThreadLocal当中，然后生成handler对象，调用Looper的myLooper()方法，得到与Handler所对应的Looper对象，这样的话，handler、looper 、消息队列就形成了一一对应的关系，然后执行上面的第三个步骤，即Looper在消息队列当中循环的取数据。每个线程通过threadlocal创建looper对象，threadlocal用于存储线程相关的局部变量，所有的线程共享一个threadlocal，但是都有自己的标识，互不影响。Android的消息机制中，Looper便是采用ThreadLocal作为存储结构，所以looper对象的存储只会在当前线程中，子线程若是使用消息机制的话，必须调用Looper.prepare方法来在线程中新建一个Looper的对象。

７．AsyncTask必须在UI线程中创建

要掌握AsyncTask，我们就必须要一个概念，总结起来就是: 3个泛型，4个步骤。

我们来看看AsyncTask这个抽象类的定义，当我们定义一个类来继承AsyncTask这个类的时候，需要为其指定3个泛型参数：

AsyncTask　<Params, Progress, Result>

* Params: 指定的是我们传递给异步任务执行时的参数的类型
* Progress: 指定的是我们的异步任务在执行的时候将执行的进度返回给UI线程的参数的类型
* Result: 指定的是异步任务执行完后返回给UI线程的结果的类型

我们在定义一个类继承AsyncTask类的时候，必须指定好这三个泛型的类型，如果都不指定的话，则都将其写成Void，例如：

AsyncTask <Void, Void, Void>

4个步骤：当我们执行一个异步任务时，需要按照下面的4个步骤分别执行：

* onPreExecute(): 这个方法是在执行异步任务之前的时候执行，并且是在UI Thread当中执行的，通常我们在这个方法里做一些UI控件的初始化的操作，例如弹出ProgressDialog
* doInBackground(Params... params): 在onPreExecute()方法执行完后，会马上执行这个方法，这个方法就是来处理异步任务的方法，Android操作系统会在后台的线程池当中开启一个worker thread来执行这个方法（即在worker thread当中执行），执行完后将执行结果发送给最后一个 onPostExecute 方法，在这个方法里，我们可以从网络当中获取数据等一些耗时的操作
* onProgressUpdate(Progess... values): 这个方法也是在UI Thread当中执行的，在异步任务执行的时候，有时需要将执行的进度返回给UI界面，例如下载一张网络图片，我们需要时刻显示其下载的进度，就可以使用这个方法来更新进度。这个方法在调用之前，我们需要在 doInBackground 方法中调用一个 publishProgress(Progress) 的方法来将进度时时刻刻传递给 onProgressUpdate 方法来更新
* onPostExecute(Result... result): 当异步任务执行完之后，就会将结果返回给这个方法，这个方法也是在UI Thread当中调用的，我们可以将返回的结果显示在UI控件上

AsyncTask使用起来比较轻量，但是其自身也存在一些问题。主要表现在

７.1.cancel方法实现不是很好.

如果你调用了AsyncTask的cancel(false)，doInBackground()仍然会执行到方法结束，只是不会去调用onPostExecute()方法。但是实际上这是让应用程序执行了没有意义的操作。那么是不是我们调用cancel(true)前面的问题就能解决呢？并非如此。如果mayInterruptIfRunning设置为true，会使任务尽早结束，但是如果的doInBackground()有不可打断的方法会失效，比如这个BitmapFactory.decodeStream() IO操作。但是你可以提前关闭IO流并捕获这样操作抛出的异常。但是这样会使得cancel()方法没有任何意义

结束任务的间接实现方式：

7.1.1、判断标志位的办法：

我们要知道在java的线程中，没有办法停止一个正在运行中的线程。在Android的AsyncTask中也是一样的。如果必须要停止一个线程，我们可以采用这个线程中设置一个标志位，然后在线程run方法或AsyncTask的doInBackground方法中的关键步骤判断这个标志位以决定是否继续执行。然后在需要终止此线程的地方改变这个标志位以达到停止线程的目的。

7.1.2、合理的利用Exception

从外部调用AsyncTask的cancel方法并不能停止一个已经启动的AsyncTask。这个cancel方法的作用与线程的interrupt方法相似，调用了一个线程的interrupt方法之后线程仍然运行，但是如果该线程的run方法里面调用过sleep的或者wait方法后，处于sleep或wait状态，则sleep和wait立即结束并且抛出InterruptedException异常。AsyncTask的cancel方法也一样，如果在这个Task的doInBackground方法中调用了sleep或wait方法，当在UI线程中调用了这个Task实例的cancel方法之后，sleep或wait立即结束并且抛出InterruptedException异常，但是如果捕获该异常的代码后面还有其他代码，则这些代码还会继续执行。

7.1.3、可以在UI上做手脚

如果用户在后台线程正获取内容时做出了取消的行为，我们可以根据用户的这种行为在UI上立即做出反馈，此时，即使线程完成了数据的Loading，我们也不让数据显示出来，算是一种投机取巧的办法吧。

７.2.内存泄露问题

还有一种常见的情况就是，在Activity中使用非静态匿名内部AsyncTask类，由于Java内部类的特点，AsyncTask内部类会持有外部类的隐式引用。由于AsyncTask的生命周期可能比Activity的长，当Activity进行销毁AsyncTask还在执行时，由于AsyncTask持有Activity的引用，导致Activity对象无法回收，进而产生内存泄露。

7.3.结果丢失

另一个问题就是在屏幕旋转等造成Activity重新创建时AsyncTask数据丢失的问题。当Activity销毁并创新创建后，还在运行的AsyncTask会持有一个Activity的非法引用即之前的Activity实例。导致onPostExecute()没有任何作用。

在使用AsyncTask做异步任务的时候必须要遵循的原则：

* AsyncTask类必须在UI Thread当中加载，在Android Jelly\_Bean版本后这些都是自动完成的
* AsyncTask的对象必须在UI Thread当中实例化
* execute方法必须在UI Thread当中调用
* 不要手动的去调用AsyncTask的四个方法，这些都是由Android系统自动调用的
* AsyncTask任务只能被执行一次

８．为什么Android中要设计为只能在UI线程中去更新UI呢？

（1）解决多线程并发问题（根本原因）

（2）提高界面更新的性能问题

（3）架构设计的简单

如果非要的话也可以在子线程中更新UI,在线程中更新UI时会调用ViewParent.invalidateChild()方法检查当前的thread是否是Mainthread。但是，ViewRootImpl这个类是在activity的onResume()方法中创建的。就算在子线程中更新UI，只要在ViewRootImpl创建之前更新UI（比如，程序在执行onCreate方法时，我就去执行setText方法区更新UI），就可以逃避掉checkThread()的检查。

9、网络优化的方法

13、自定义view过程，三个测量模式，如何从onMesure的两个参数中获取长度

14、事件分发机制

15、View A、View B事件传递过来，B onInterrupt返回true，接下来的事件是怎样的

16、类加载、双亲委托以及如何打破、两个类全限定名一样，都会加载吗，如何自己继承写一个classloader

17、listview加载网络图片，如何防止图片错乱

18、java的四种引用

19、dp和px转换公式，为什么用dp

20、打包签名流程，keystore的作用，二次打包。包名一样，签名不一样可以同时安装吗

21、一键结束全部程序，activity会不会调用onstop，ondestroy，被强杀怎样恢复数据

23、volatile和synchronized区别、锁有哪些、static synchronize修饰的方法是什么锁，两个线程同时i++会怎么样，线程池

所有加上synchronized 和 块语句，在多线程访问的时候，同一时刻只能有一个线程能够用

用volatile修饰的变量，线程在每次使用变量的时候，都会读取变量修改后的最新的值。

Volatile是轻量级的synchronized，它在多处理器开发中保证了共享变量的“可见性”。可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。

经典应用

public class Thread1 extends Thread {

private boolean flag = false;

public void run() {

while(!flag) {

// ...

}

}

public void close() {

flag = true;

}

}

java多线程同时操作一个变量要怎么处理？

如果只是读操作，没有写操作，可以不加锁，建议使用final关键字

如果同时有写操作，但是变量的写操作和当前变量无关，同时也与其他变量不关联，则可以考虑加上volatile关键字，写操作要用synchronize

如果写操作依赖变量的当前值，比如自增，则get方法和写方法都需要synchronize

synchronize四大用法

1. 修饰一个类，其作用的范围是synchronized后面括号括起来的部分，*作用的对象是这个类的所有对象*。

class ClassName {  
 public void method() {  
 synchronized(ClassName.class) {  
 // todo  
 }  
 }  
}

2. 修饰一个方法，被修饰的方法称为同步方法，其作用的范围是整个方法，*作用的对象是调用这个方法的对象*；

public synchronized void method()  
{  
 // todo  
}

3. 修改一个静态的方法，其作用的范围是整个静态方法，*作用的对象是这个类的所有对象*；

public synchronized static void method() {  
 // todo  
}

4. 修饰一个代码块，被修饰的代码块称为同步语句块，其作用的范围是大括号{}括起来的代码，*作用的对象是调用这个代码块的对象*；

synchronized(this)

24、activity中有个线程跑完后会显示textview文字，但是没跑完就按返回了，会发生什么情况

25、servlet生命周期、是单利的吗

26、JVM内存模型

27、垃圾回收算法

28、常用集合类

29、hashmap和hashtable的区别和原理

30、乐观锁悲观锁

31、装饰着模式、代理模式的区别和用处

32、java反射机制

34、GC垃圾收集有哪些算法、收集对象、时机，什么时候会触发

36、sleep()和wait()有什么区别

两者都有使线程暂停的作用，但是sleep不会释放线程锁，而wait用在synchronized代码块中，会释放线程锁

37、final\finally\finalize有什么区别

final，对于类，表示该类的功能已经完善，不希望被修改或继承，对于变量，表示常量；finally和try,catch搭配使用，最终一定会调用finally方法；finalize() 方法是在垃圾收集器删除对象之前对这个对象调用的。 finalize()方法并不能等同与析构函数。Java中是没有析构函数的。C++的析构函数是在对象消亡时运行的。由于C++没有垃圾回收，对象空间手动回收，所以一旦对象用不到时，程序员就应当把它delete()掉。所以析构函数中经常做一些文件保存之类的收尾工作。但是在Java中很不幸，如果内存总是充足的，那么垃圾回收可能永远不会进行，也就是说filalize()可能永远不被执行，显然指望它做收尾工作是靠不住的。

38、安卓四大组件

38.1. Activity，

38.1.1 生命周期：

|----usr navi back---<--killed-<-other need mem---|---------------------|

| |---running->come bg--| |

onCreate----->onStart-visi->onResume------->onPause---invisible-->onStop-------->onDestroy

| |----<----come fg---------| |

|----------<---onRestart---------<--------come fg--------|

注意：从activity是否可见来说，onstart()和onStop()是配对的，从activity是否在前台来说，onResume()和onPause()是配对的；当activity中弹出dialog对话框的时候，activity不会回调onPause。然而当activity启动dialog风格的activity的时候，此activity会回调onPause函数。

38.1.2 异常情况下的生命周期：

(1)系统资源配置变化，例如旋转屏幕，在onStop之前调用onSavedInstance保存状态，重建之后在onStart之后调用onRestoreInstance恢复状态。

38.1.2 启动模式lauchmode：

singleTop:栈顶复用，若新activity不位于栈顶则新建，否则不会被重新创建，调用onNewIntent

singleInstance：共用一个activity实例

singleTask：栈内复用，清空上方activity并移至栈顶

standard：不断地new并且压栈，谁启动了我，我就进入它所属的任务栈中

38.2 Service

38.2.1 本地服务

调用者和service在同一个进程里，所以运行在主进程的main线程中。所以不能进行耗时操作，可以采用在service里面创建一个Thread来执行任务。service影响的是进程的生命周期，讨论与Thread的区别没有意义。任何 Activity 都可以控制同一 Service，而系统也只会创建一个对应 Service 的实例。

（1）start启动方式

1，定义一个类继承service2，manifest.xml文件中配置service3，使用context的startService(Intent)方法启动service4，不在使用时，调用stopService(Intent)方法停止服务

生命周期：

onCreate() -- > onStartCommand() -- > onDestory()

注意：如果服务已经开启，不会重复回调onCreate()方法，如果再次调用context.startService()方法，service而是会调用onStart()或者onStartCommand()方法。停止服务需要调用context.stopService()方法，服务停止的时候回调onDestory被销毁。

特点：一旦服务开启就跟调用者（开启者）没有任何关系了。开启者退出了，开启者挂了，服务还在后台长期的运行，开启者不能调用服务里面的方法。

（2）bind启动方式

3，使用context的bindService(Intent,ServiceConnection,int)方法启动service4，不再使用时，调用unbindService(ServiceConnection)方法停止该服务

生命周期：

onCreate() -- > onBind() --> onUnbind() -- > onDestory()

注意：绑定服务不会调用onStart()或者onStartCommand()方法

特点：bind的方式开启服务，绑定服务，调用者挂了，服务也会跟着挂掉。绑定者可以调用服务里面的方法。

38.2.2 远程服务

调用者和service不在同一个进程中，service在单独的进程中的main线程，是一种垮进程通信方式

38.2.3 IntentService

IntentService是Service的子类，比普通的Service增加了额外的功能。先看Service本身存在两个问题：Service不会专门启动一条单独的进程，Service与它所在应用位于同一个进程中；Service也不是专门一条新线程，因此不应该在Service中直接处理耗时的任务；

IntentService特征:：会创建独立的worker线程来处理所有的Intent请求；会创建独立的worker线程来处理onHandleIntent()方法实现的代码，无需处理多线程问题；所有请求处理完成后，IntentService会自动停止，无需调用stopSelf()方法停止Service；为Service的onBind()提供默认实现，返回null；为Service的onStartCommand提供默认实现，将请求Intent添加到队列中；

38.3 BroadcastReceiver：

广播被分为两种不同的类型：“普通广播（Normal broadcasts）”和“有序广播（Ordered broadcasts）”。普通广播是完全异步的，可以在同一时刻（逻辑上）被所有接收者接收到，消息传递的效率比较高，但缺点是：接收者不能将处理结果传递给下一个接收者，并且无法终止广播Intent的传播；然而有序广播是按照接收者声明的优先级别（声明在intent-filter元素的android:priority属性中，数越大优先级别越高,取值范围:-1000到1000。也可以调用IntentFilter对象的setPriority()进行设置），被接收者依次接收广播。如：A的级别高于B,B的级别高于C,那么，广播先传给A，再传给B，最后传给C。A得到广播后，可以往广播里存入数据，当广播传给B时,B可以从广播中得到A存入的数据。

Context.sendBroadcast或Context.sendOrderedBroadcast

静态注册（通过manifest.xml）和动态注册（通过registerReceiver方法声明），动态注册广播不是常驻型广播，也就是说广播跟随activity的生命周期。注意: 在activity结束前，移除广播接收器。静态注册是常驻型，也就是说当应用程序关闭后，如果有信息广播来，程序也会被系统调用自动运行。

用处：线程间通信，通过发送广播更新UI，监听系统事件

生命周期：很短，onReceive后就结束。所以不要在广播里面处理复杂事务

38.4 ContentProvider

用于跨程序共享数据。如果是使用系统的ContentProvider，比如获取通讯录，一般传url就可以，如果是自定义ContentProvider，需要实现类似数据库操作的全部内容。

39、常用布局

RelativelyLayout，LinearLayout、GridLayout、FrameLayout

40、简述JNI调用流程和步骤

1）system.loadlibrary()方法，加载类库，在java中声明native方法

2）使用javah声明类的头文件，编写对应的c文件实现native方法

3）编写mk文件，指定编译的类库和c文件

4）使用ndk编译生成so文件，放入对应目录

5）在java中调用native方法

41、Android程序中的内存优化技术

减小对象的内存占用：使用更加轻量的数据结构，比如使用arraymap，不使用enum，减小bitmap的内存占用，使用更小的图片

内存对象的重复利用：避免在onDraw等频繁调用的方法中创建对象，使用stringbuilder代替频繁的‘+’，复用系统自带的资源

避免对象的内存泄漏：注意activity的泄露，通常来说，Activity的泄漏是内存泄漏里面最严重的问题，它占用的内存多，影响面广，我们需要特别注意以下两种情况导致的Activity泄漏：内部类引用导致Activity的泄漏：最典型的场景是Handler导致的Activity泄漏，如果Handler中有延迟的任务或者是等待执行的任务队列过长，都有可能因为Handler继续执行而导致Activity发生泄漏。为了解决这个问题，可以在UI退出之前，执行remove Handler消息队列中的消息与runnable对象。Activity Context被传递到其他实例中，这可能导致自身被引用而发生泄漏：内部类引起的泄漏不仅仅会发生在Activity上，其他任何内部类出现的地方，都需要特别留意！我们可以考虑尽量使用static类型的内部类。

再比如各种监听器，需要自己手动unregister。

内存使用策略优化：比如在设计listview和gridview就要考虑硬件的性能；资源文件也要放在合适的文件夹下，否则会因为拉伸而显著增加内存占用；谨慎使用static对象，生命周期过长，占用内存；珍惜services资源，多使用intentservices等等

退出时显示调用finish()方法，释放资源

<http://hukai.me/android-performance-oom/>

内存优化的不好，就会发生OOM

42、GET和POST的区别

get是向服务器索取数据的请求，用于数据的“查”，规范所称的安全（不会改变其他数据）和幂等（相同的url多次获取的结果相同），直接将参数加在链接后面，安全性不高；post是向服务器提交数据的请求，用于数据的“更新”，可以将参数装在一个包体内发送给服务器，安全性更高

43、android解析XML有三种方式

SAX 事件驱动 startDocument startElement endDocument endElement saxparserFactory

DOM 整个文件读入内存然后用dom的api遍历数据 documentBuilder Element

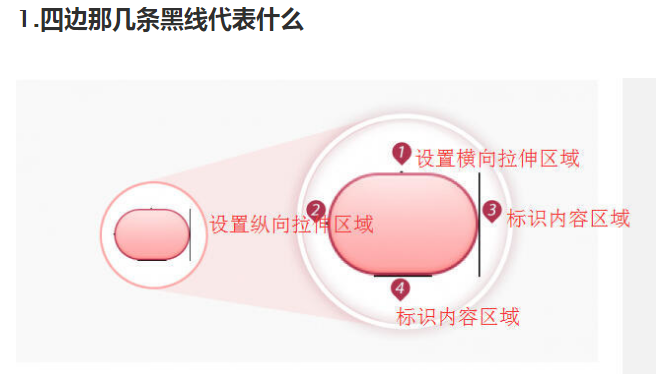
pull 事件驱动 xmlpullparser next() getAttribute

44、点9图示一种可伸缩的位图，如果你某个View用了点9图做background，

Android会根据点9图的设置来自动为你调整、适应内容。

点9图是标准的PNG格式图像，被一圈1像素宽的边缘包围，

并且保存的时候扩展名一定要是“.9.png”，还要保存在“res/drawable/”目录。



45、退出多个activity的方法

容器式：搞个list，存activity，然后挨个finish()

广播式：接收广播，然后finish()

广播+singleTask：把mainActivity设置为singletask，再发送广播来finish()

只用singleTask

public class MainActivity extends BaseActivity {  
  
 public static final String EXTRA\_EXIT = "extra\_exit";  
  
 @Override  
 protected void onNewIntent(Intent intent) {  
 super.onNewIntent(intent);  
 boolean shouldExit = intent.getBooleanExtra(EXTRA\_EXIT, false);  
 if (shouldExit) {  
 finish();  
 }  
 }  
  
}再调用

Intent startMainIntent = new Intent(this, MainActivity.class);  
startMainIntent.putExtra(MainActivity.EXTRA\_EXIT, true);  
startActivity(startMainIntent);

设置IntentFlag

Intent intent = new Intent(this, WelcomeActivity.class);  
intent .setFlags(Intent.FLAG\_ACTIVITY\_CLEAR\_TOP);  
startActivity(intent);和singleTask异曲同工

当然还有一些不优雅的方法

比如抛出异常，强制退出

再比如递归退出，利用startActivityforResult和onActivityResult

46、force close

出现force close一般是因为没有正确获取到Exceptions，比如NullPointExection（空指针），IndexOutOfBoundsException（下标越界），就连Android API使用的顺序错误也可能导致（比如setContentView()之前进行了findViewById()操作）等等一系列未捕获异常

如何在force close后自动重启或自动关闭？

调用thread.setDefaultUncaughtExceptionHandler，这样当应用崩溃时就会自动调用UNCaughtException方法，方法里面做重启或关闭

在 Activity 的 onCreate() 方法中调用如下代码：

Thread.setDefaultUncaughtExceptionHandler(this);

47、使用sql数据库

Android在运行时集成了SQLite ， 所以每个Android应用程序都可以使用SQLite数据库。

数据库存放的位置：data/<项目文件夹>/databases/

第一部是创建数据库，使用sqliteOpenhelper，只要继承他就可以轻松使用sqlite数据库，提供了getReadableDatabase,getWriteableDatabase来获得数据库对象并通过它们进行数据库操作。还提供了onCreate和onUpgrade方法来创建、删除数据库。具体的操作使用execSQL方法，就是传入命令行

48、HttpClient

apache

String url="http://10.0.2.2:8080/JSP/get.jsp?name=DARKER&password=49681888";  
 HttpGet hget=new HttpGet(url);  
 HttpClient htclient=new DefaultHttpClient();  
 try {  
 HttpResponse hresonse=htclient.execute(hget);  
 if(hresonse.getStatusLine().getStatusCode()==HttpStatus.SC\_OK){  
 String resul=EntityUtils.toString(hresonse.getEntity());  
 tv.setText(resul);  
 }else{  
 tv.setText("连接失败...");  
 }  
 }

还可以以键值对的形式加上http请求头以及取出响应头中的信息

==================================

1.algorithm中的find怎么用

template <class InputIterator, class T>  
 InputIterator find (InputIterator first, InputIterator last, const T& val);

int myints[] = { 10, 20, 30, 40 };

p = std::find (myints, myints+4, 30);  
 if (p != myints+4)  
 std::cout << "Element found in myints: " << \*p << '\n';  
 else  
 std::cout << "Element not found in myints\n";

std::vector<int> myvector (myints,myints+4);  
 std::vector<int>::iterator it;  
  
 it = find (myvector.begin(), myvector.end(), 30);  
 if (it != myvector.end())  
 std::cout << "Element found in myvector: " << \*it << '\n';  
 else  
 std::cout << "Element not found in myvector\n";

2.algorithm中的sort怎么用

如果只是寻常数组，其实很简单，需要注意的是默认按照从小到大排列sort（first，end，method）

如果数组元素是结构体，或者是对vector排序，则有所不同，如下

struct Info {int x;};

bool copare(const Info\* pfirst,const Info\* psecond) //如果该vector存入的是对象的话该函数参数须是对象的引用，而不该是指针

{

return pfirst->x>=psecond->x;

}

int main(int argc, char\* argv[])

{ vector m\_vecInfo;

Info\* pInfo;

for (int i=0;i<10;i++)

{

pInfo = new Info;

pInfo->x=i;

m\_vecInfo.push\_back(pInfo);

}

sort(m\_vecInfo.begin(),m\_vecInfo.end(),copare); //按照自己写的排序函数进行排序

return 0;

}

如果是字符串数组或者一大段话里面的单词进行排序，就可以用到vecotr加sort了

3.如何计算二进制数中1的位数

常规方法，O(n)复杂度，逐位与1相与，1不停地左移。需要注意的是左移：m<<n将m左移n位，左移后低位补充0；

右移：m>>n将m右移n位，右移后高位补充的是符号位，负数补充1，整数补充0.（正数的边界值为（1，ox7FFFFFFF）,负数的边界值为（ox80000000,oxFFFFFFFF））

int numberOf1(int n)

{

int count=0;

int tag=1;

while(tag)

{

if(tag&n)

count++;

tag=tag<<1;

}

return count;

}

牛逼方法，将原数字减去1后与原数字相与，循环，直至数字变为0，数字中有几个1就循环几次

int numberOf1(int n)

{

int count=0;

while(n)

{

++count;

n=(n-1)&n;

}

return count;

}

引申：

判断一个整数是不是2的整数次方？

如果是的话，该数字只有一位是1，所以可用上面的牛逼方法，一句代码搞定

输入两个整数m和n，判断m需要改变几位才能变成n

异或

4.c语言运算符优先级

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operator** | **Description** | **Associativity** |
| **1** | :: | [Scope resolution](http://en.cppreference.com/w/cpp/language/identifiers#Qualified_identifiers) | Left-to-right |
| **2** | a++ a-- | Suffix/postfix [increment and decrement](http://en.cppreference.com/w/cpp/language/operator_incdec) |
| *type*() *type*{} | [Functional cast](http://en.cppreference.com/w/cpp/language/explicit_cast) |
| a() | [Function call](http://en.cppreference.com/w/cpp/language/operator_other#Built-in_function_call_operator) |
| a[] | [Subscript](http://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_subscript_operator) |
| . -> | [Member access](http://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_member_access_operators) |
| **3** | ++a --a | Prefix [increment and decrement](http://en.cppreference.com/w/cpp/language/operator_incdec) | Right-to-left |
| +a -a | Unary [plus and minus](http://en.cppreference.com/w/cpp/language/operator_arithmetic#Unary_arithmetic_operators) |
| ! ~ | [Logical NOT](http://en.cppreference.com/w/cpp/language/operator_logical) and [bitwise NOT](http://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_logic_operators) |
| (*type*) | [C-style cast](http://en.cppreference.com/w/cpp/language/explicit_cast) |
| \*a | [Indirection](http://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_indirection_operator) (dereference) |
| &a | [Address-of](http://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_address-of_operator) |
| sizeof | [Size-of](http://en.cppreference.com/w/cpp/language/sizeof)[[note 1]](http://en.cppreference.com/w/cpp/language/operator_precedence#cite_note-1) |
| new new[] | [Dynamic memory allocation](http://en.cppreference.com/w/cpp/language/new) |
| delete delete[] | [Dynamic memory deallocation](http://en.cppreference.com/w/cpp/language/delete) |
| **4** | .\* ->\* | [Pointer-to-member](http://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_pointer-to-member_access_operators) | Left-to-right |
| **5** | a\*b a/b a%b | [Multiplication, division, and remainder](http://en.cppreference.com/w/cpp/language/operator_arithmetic#Multiplicative_operators) |
| **6** | a+b a-b | [Addition and subtraction](http://en.cppreference.com/w/cpp/language/operator_arithmetic#Additive_operators) |
| **7** | << >> | Bitwise [left shift and right shift](http://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_shift_operators) |
| **8** | < <= | For [relational operators](http://en.cppreference.com/w/cpp/language/operator_comparison) < and ≤ respectively |
| > >= | For [relational operators](http://en.cppreference.com/w/cpp/language/operator_comparison) > and ≥ respectively |
| **9** | == != | For [relational operators](http://en.cppreference.com/w/cpp/language/operator_comparison) = and ≠ respectively |
| **10** | a&b | [Bitwise AND](http://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_logic_operators) |
| **11** | ^ | [Bitwise XOR](http://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_logic_operators) (exclusive or) |
| **12** | | | [Bitwise OR](http://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_logic_operators) (inclusive or) |
| **13** | && | [Logical AND](http://en.cppreference.com/w/cpp/language/operator_logical) |
| **14** | || | [Logical OR](http://en.cppreference.com/w/cpp/language/operator_logical) |
| **15** | a?b:c | [Ternary conditional](http://en.cppreference.com/w/cpp/language/operator_other#Conditional_operator)[[note 2]](http://en.cppreference.com/w/cpp/language/operator_precedence#cite_note-2) | Right-to-left |
| throw | [throw operator](http://en.cppreference.com/w/cpp/language/throw) |
| = | [Direct assignment](http://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_direct_assignment) (provided by default for C++ classes) |
| += -= | [Compound assignment](http://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_compound_assignment) by sum and difference |
| \*= /= %= | [Compound assignment](http://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_compound_assignment) by product, quotient, and remainder |
| <<= >>= | [Compound assignment](http://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_compound_assignment) by bitwise left shift and right shift |
| &= ^= |= | [Compound assignment](http://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_compound_assignment) by bitwise AND, XOR, and OR |
| **16** | , | [Comma](http://en.cppreference.com/w/cpp/language/operator_other#Built-in_comma_operator) | Left-to-right |

5.内存屏障

**内存屏障**，也称**内存栅栏**，**内存栅障**，**屏障指令**等，是一类[同步屏障](https://zh.wikipedia.org/wiki/%E5%90%8C%E6%AD%A5%E5%B1%8F%E9%9A%9C)指令，是CPU或编译器在对内存随机访问的操作中的一个同步点，使得此点之前的所有读写操作都执行后才可以开始执行此点之后的操作。大多数现代计算机为了提高性能而采取[乱序执行](https://zh.wikipedia.org/wiki/%E4%B9%B1%E5%BA%8F%E6%89%A7%E8%A1%8C)，这使得内存屏障成为必须。语义上，内存屏障之前的所有写操作都要写入内存；内存屏障之后的读操作都可以获得同步屏障之前的写操作的结果。因此，对于敏感的程序块，写操作之后、读操作之前可以插入内存屏障。

大多数处理器提供了内存屏障指令:

* 完全内存屏障（full memory barrier）保障了早于屏障的内存读写操作的结果提交到内存之后，再执行晚于屏障的读写操作。
* 内存读屏障（read memory barrier）仅确保了内存读操作；
* 内存写屏障(write memory barrier)仅保证了内存写操作。

Windows API的内存屏障实现

下述同步函数使用适当的屏障来确保内存有序：

* 进出临界区(critical section)的函数
* 触发(signaled)同步对象的函数
* 等待函数(Wait function)
* 互锁函数(Interlocked function)

6、一个算法应该具有以下五个重要的特征：

有穷性（Finiteness）算法的有穷性是指算法必须能在执行有限个步骤之后终止；

确切性(Definiteness)算法的每一步骤必须有确切的定义；

输入项(Input)一个算法有0个或多个输入，以刻画运算对象的初始情况，所谓0个输入是指算法本身定出了初始条件；

输出项(Output)一个算法有一个或多个输出，以反映对输入数据加工后的结果。没有输出的算法是毫无意义的；

可行性(Effectiveness)算法中执行的任何计算步骤都是可以被分解为基本的可执行的操作步，即每个计算步都可以在有限时间内完成（也称之为有效性）。

7、DNS同时占用UDP和TCP端口53，DNS在进行区域传输的时候使用TCP协议，其它时候则使用UDP协议； DNS的规范规定了2种类型的DNS服务器，一个叫主DNS服务器，一个叫辅助DNS服务器。在一个区中主DNS服务器从自己本机的数据文件中读取该区的DNS数据信息，而辅助DNS服务器则从区的主DNS服务器中读取该区的DNS数据信息。当一个辅助DNS服务器启动时，它需要与主DNS服务器通信，并加载数据信息，这就叫做区传送（zone transfer）。 原因：UDP报文的最大长度为512字节，而TCP则允许报文长度超过512字节。当DNS查询超过512字节时，协议的TC标志出现删除标志，这时则使用TCP发送。通常传统的UDP报文一般不会大于512字节。

HTTP只使用TCP，QQ用的是UDP，主要在于可以更好地利用带宽，降低抖动的影响。FTP用TCP，20端口传数据，21端口做控制。

8、虚存=min（内存+辅存，逻辑地址）

9、Linux系统32位与64位数据类型长度比较

可通过getconf命令来获取。

[Linux](http://www.2cto.com/os/linux/)系统32位与64位GCC编译器基本数据类型长度对照表

GCC 32位

sizeof(char)=1

sizeof(double)=8

sizeof(float)=4

sizeof(int)=4

sizeof(short)=2

sizeof(long)=4

sizeof(long long)=8

sizeof(long double)=12

sizeof(complex long double)=24

指针是4字节；

GCC 64位

sizeof(char)=1

sizeof(double)=8

sizeof(float)=4

sizeof(int)=4

sizeof(short)=2

sizeof(long)=8

sizeof(long long)=8

sizeof(long double)=16

sizeof(complex long double)=32

指针是8字节

10、linux创建文件

touch之后用vi或vim或gedit进行编辑

一个应用实例：在系统需要维护的时候，不希望普通用户登录到系统，需要对普通用户进行配置，其实很简单的，只需要在 /etc 目录下创建一个名为 nologin 的文件，可以在文件中写入一些提示信息

11、C++继承

公有：公有继承的特点是基类的公有成员和保护成员作为派生类的成员时，它们都保持原有的状态，而基类的私有成员仍然是私有的，不能被这个派生类的子类所访问。

私有：私有继承的特点是基类的公有成员和保护成员都作为派生类的私有成员，并且不能被这个派生类的子类所访问。

保护：保护继承的特点是基类的所有公有成员和保护成员都成为派生类的保护成员，并且只能被它的派生类成员函数或友元访问，基类的私有成员仍然是私有的。

下面列出三种不同的继承方式的基类特性和派生类特性。

|  |  |  |  |
| --- | --- | --- | --- |
|  | public | protected | private |
| 共有继承 | public | protected | 不可见 |
| 私有继承 | private | private | 不可见 |
| 保护继承 | protected | protected | 不可见 |

##### **对于公有继承方式**

(1) 基类成员对其对象的可见性：

公有成员可见，其他不可见。这里保护成员同于私有成员。

(2) 基类成员对派生类的可见性：

公有成员和保护成员可见，而私有成员不可见。这里保护成员同于公有成员。

(3) 基类成员对派生类对象的可见性：

公有成员可见，其他成员不可见。

所以，在公有继承时，派生类的对象可以访问基类中的公有成员；派生类的成员函数可以访问基类中的公有成员和保护成员。这里，一定要区分清楚派生类的对象和派生类中的成员函数对基类的访问是不同的。

##### **对于私有继承方式**

(1) 基类成员对其对象的可见性：

公有成员可见，其他成员不可见。

(2) 基类成员对派生类的可见性：

公有成员和保护成员是可见的，而私有成员是不可见的。

(3) 基类成员对派生类对象的可见性：

所有成员都是不可见的。

所以，在私有继承时，基类的成员只能由直接派生类访问，而无法再往下继承。

##### **对于保护继承方式**

这种继承方式与私有继承方式的情况相同。

private,public,protected方法的访问范围.（public继承下）默认private

private: 只能由该类中的函数、其友元函数访问,不能被任何其他访问，该类的对象也不能访问.

protected: 可以被该类中的函数、子类的函数、以及其友元函数访问,但不能被该类的对象访问

public: 可以被该类中的函数、子类的函数、其友元函数访问,也可以由该类的对象访问

12、构造函数执行的顺序是，先构造父类，再构造子类，其中父类的构造顺序是从左到右。然后析构函数执行的顺序是跟构造函数正好相反的，先执行自身的析构函数，然后再依次从右到左执行父类的析构函数。这是继承的情况

但如果是下面这样

class B  
 {  
 public:  
 // Notice  
 B(): a(A()), c(C()) {cout << "Construct B" << endl;}  
 ~B(){cout << "Destruct B" << endl;}  
 C c;  
 A a;  
 };

ABC不再是继承关系，而是组合关系。此时，列表初始化是先于构造函数的调用的，而且列表初始化是与初始化顺序无关，只与数据成员定义的顺序有关。在上面的例子中C类型的变量定义在A类型的变量前面，因此会先构造C，之后构造A。

如果同时有继承关系和组合关系呢？此时先按照继承顺序调用父类的构造函数，再按照变量定义顺序调用成员的构造函数。

13、虚拟继承

虚拟继承是多重继承中特有的概念。虚拟基类是为解决多重继承而出现的。如:类D继承自类B1、B2，而类B1、B2都继承自类A，因此在类D中两次出现类A中的变量和函数。为了节省内存空间，可以将B1、B2对A的继承定义为虚拟继承，而A就成了虚拟基类。实现的代码如下：

class A

class B1:public virtual A;

class B2:public virtual A;

class D:public B1,public B2;

虚拟继承在一般的应用中很少用到，所以也往往被忽视，这也主要是因为在C++中，多重继承是不推荐的，也并不常用，而一旦离开了多重继承，虚拟继承就完全失去了存在的必要因为这样只会降低效率和占用更多的空间。

对于第12题，如果有虚拟继承，则先调用虚拟基类的构造函数

14、[String s=new String("abc")创建了几个对象?](http://www.cnblogs.com/ydpvictor/archive/2012/09/09/2677260.html)

两个。这行代码分成String str、=、"abc"和new String()四部分来看待。String str只是定义了一个名为str的String类型的变量，因此它并没有创建对象；=是对变量str进行初始化，将某个对象的引用（或者叫句柄）赋值给它，显然也没有创建对象；

来看一下被我们调用了的String的构造器：

public String(String original) { //other code ... } 大家都知道，我们常用的创建一个类的实例（对象）的方法有以下两种:

一、使用new创建对象。

二、调用Class类的newInstance方法，利用反射机制创建对象。

我们正是使用new调用了String类的上面那个构造器方法创建了一个对象，并将它的引用赋值给了str变量。

引号内包含文本，这种方式是String特有的，并且它与new的方式存在很大区别。

String str="abc";

毫无疑问，这行代码创建了一个String对象。

String a="abc"; String b="abc"; 那这里呢？

答案还是一个。

String a="ab"+"cd"; 再看看这里呢？

答案是三个。如果考虑编译器优化的话，是两个。

说到这里，我们就需要引入对字符串池相关知识的回顾了。

在JAVA虚拟机（JVM）中存在着一个字符串池，其中保存着很多String对象，并且可以被共享使用，因此它提高了效率。由于String类是final的，它的值一经创建就不可改变，因此我们不用担心String对象共享而带来程序的混乱。字符串池由String类维护，我们可以调用intern()方法来访问字符串池。

我们再回头看看String a="abc";，这行代码被执行的时候，JAVA虚拟机首先在字符串池中查找是否已经存在了值为"abc"的这么一个对象，它的判断依据是String类equals(Object obj)方法的返回值。如果有，则不再创建新的对象，直接返回已存在对象的引用；如果没有，则先创建这个对象，然后把它加入到字符串池中，再将它的引用返回。

只有使用引号包含文本的方式创建的String对象之间使用“+”连接产生的新对象才会被加入字符串池中。对于所有包含new方式新建对象（包括null）的“+”连接表达式，它所产生的新对象都不会被加入字符串池中，对此我们不再赘述。因此我们提倡大家用引号包含文本的方式来创建String对象以提高效率，实际上这也是我们在编程中常采用的。

引申：

下面语句创建了几个对象（）

String s1 = "Hello World";

String s2 = "Hello World";

String s3 = s1;

String s4 = new String("Hello World");

String s5 = new String("Hello World");

三个

15、java int -- integer : int i = 0; Integer wrapperi = new Integer(i);

integer -- int :Integer wrapperi = new Integer(0); int i = wrapperi.intValue();

JDK1.5以后，Java为我们提供了更为丰富的转换方法。

其中最值得一提的就是自动装包/自动拆包(AutoBoxing/UnBoxing)。

此功能大大丰富了基本类型(primitive type)数据与它们的包装类(Wrapper Class)

的使用。

由于AutoBoxing的存在，以下代码在JDK1.5的环境下可以编译通过并运行。

**int -- Integer int** i = 0; Integer wrapperi = i;

以及**public** **static** Integer valueOf(**int** i)

**int** i = 0; Integer wrapperi = Integer.valueOf(i);

此方法与new Integer(i)的不同处在于：

方法一调用类方法返回一个表示指定的 int 值的 Integer 实例。

方法二产生一个新的Integer对象。

查看valueOf方法。

1. /\*
2. \* 返回一个表示指定的 int 值的 Integer 实例。如果不需要新的 Integer 实例，则
3. \* 通常应优先使用该方法，而不是构造方法 Integer(int)，因为该方法有可能通过
4. \* 缓存经常请求的值而显著提高空间和时间性能。
5. \* @param i an <code>int</code> value.
6. \* @return a <tt>Integer</tt> instance representing <tt>i</tt>.
7. \* @since 1.5
8. \*/
9. **public** **static** Integer valueOf(**int** i) {
10. **final** **int** offset = 128;
11. **if** (i >= -128 && i <= 127) { // must cache
12. **return** IntegerCache.cache[i + offset];
13. }
14. **return** **new** Integer(i);
15. }

可以看到对于范围在-128到127的整数，valueOf方法做了特殊处理。

采用IntegerCache.cache[i + offset]这个方法。

1. /\*
2. \* IntegerCache内部类
3. \* 其中cache[]数组用于存放从-128到127一共256个整数
4. \*/
5. **private** **static** **class** IntegerCache {
6. **private** IntegerCache(){}
8. **static** **final** Integer cache[] = **new** Integer[-(-128) + 127 + 1];
10. **static** {
11. **for**(**int** i = 0; i < cache.length; i++)
12. cache[i] = **new** Integer(i - 128);
13. }
14. }

这就是valueOf方法真正的优化方法，当-128=<i<=127的时候，返回的是IntegerCache中的数组的值；当 i>127 或 i<-128 时，返回的是Integer类对象。

再举一个经常被提到的例子

1. Integer i=100;
2. Integer j=100;
3. //print true
4. System.out.println(i==j);

此时的 i=IntegerCache.cache[i + 128] = IntegerCache.cache[228]，

同样j = IntegerCache.cache[j + 128] = IntgerCache.cache[228]

因此 Integer引用i中存储的是cache数组第228号元素的地址。同理j也是同一个cache数组的第228号元素的地址(因为cache是Integer的static数组，只有一个)。

i==j比较的是引用地址，因此返回true。

1. Integer i=200;
2. Integer j=200;
3. //print false
4. System.out.println(i==j);

此时的 i=new Integer(200); 同样j=new Integer(200) 。

两次都在堆中开辟了Integer的对象。

i 和 j 中存储的堆的对象地址是完全不同的。i==j 自然返回false。

引入缓存机制通过缓存经常请求的值而显著提高空间和时间性能。 valueOf(int i)的优化只针对于范围在-128到127的整数。

**JDK1.5以后的Integer转int**

由于UnBoxing的存在，以下代码在JDK1.5的环境下可以编译通过并运行。

1. Integer wrapperi = **new** Integer(0);
2. **int** i = wrapperi;

**附：AutoBoxing与UnBoxing带来的转变**

以下代码在JDK1.5中成为了可能，试想下在JDK1.5之前该如何实现这段代码？

1. **int** x = 1;
2. Collection collection = **new** ArrayList();
3. collection.add(x);//AutoBoxing,自动转换成Integer.
4. Integer y = **new** Integer(2);
5. collection.add(y + 2); //y + 2为UnBoxing，自动转换成int。之后再次转换为Integer。

此特性同样适用于Map

1. Map map = **new** HashMap();
2. **int** x = 1;
3. Integer y = **new** Integer(2);
4. **int** z = 3;
5. map.put(x,y + z);//x自动转换成Integer。y+z自动转换成int。之后再次转换为Integer。

15、java main中的String args[]

String[] args这个字符串数组是保存运行main函数时输入的参数 的，例如 main函数所在的类名为test那么你在cmd运行java test a b c时

args[0]=a,args[1]=b,args[2]=c,你就可以在你的程序中调用你输入的这些变量。

16、java继承

IS-A关系：IS-A就是说:一个对象是另一个对象的一个分类。

关键字extends和implements都可以实现IS-A关系。

HAS-A 关系

HAS-A代表类和它的成员之间的从属关系。这有助于代码的重用和减少代码的错误。

例子

public class Vehicle{}  
public class Speed{}  
public class Van extends Vehicle{  
 private Speed sp;  
}

Java只支持单继承（继承基本类和抽象类），但是我们可以用接口来实现（多继承接口来实现）,脚本结构如：

public class Apple extends Fruit implements Fruit1, Fruit2{}

17、C++ string转int

atoi(string.c\_str)

18、TCP的报文到达确认ACK机制

TCP的报文到达确认（ACK），是对接收到的数据的最高序列号的确认，并向发送端返回一个下次接收时期望的TCP数据包的序列号（Ack Number）。例如， 主机A发送的当前数据序号是400，数据长度是100，则接收端收到后会返回一个确认号是501的确认号给主机A。

TCP提供的确认机制，可以在通信过程中可以不对每一个TCP数据包发出单独的确认包（Delayed ACK机制），而是在传送数据时，顺便把确认信息传出， 这样可以大大提高网络的利用率和传输效率。同时，TCP的确认机制，也可以一次确认多个数据报，例如，接收方收到了201，301，401的数据报，则只 需要对401的数据包进行确认即可，对401的数据包的确认也意味着401之前的所有数据包都已经确认，这样也可以提高系统的效率。

若发送方在规定时间内没有收到接收方的确认信息，就要将未被确认的数据包重新发送。接收方如果收到一个有差错的报文，则丢弃此报文，并不向发送方 发送确认信息。因此，TCP报文的重传机制是由设置的超时定时器来决定的，在定时的时间内没有收到确认信息，则进行重传。这个定时的时间值的设定非 常重要，太大会使包重传的延时比较大，太小则可能没有来得及收到对方的确认包发送方就再次重传，会使网络陷入无休止的重传过程中。接收方如果收到 了重复的报文，将会丢弃重复的报文，但是必须发回确认信息，否则对方会再次发送。

TCP协议应当保证数据报按序到达接收方。如果接收方收到的数据报文没有错误，只是未按序号，这种现象如何处理呢？TCP协议本身没有规定，而是由TCP 协议的实现者自己去确定。通常有两种方法进行处理：一是对没有按序号到达的报文直接丢弃，二是将未按序号到达的数据包先放于缓冲区内，等待它前面 的序号包到达后，再将它交给应用进程。后一种方法将会提高系统的效率。例如，发送方连续发送了每个报文中100个字节的TCP数据报，其序号分别是1， 101，201，…,701。假如其它7个数据报都收到了，而201这个数据报没有收到，则接收端应当对1和101这两个数据报进行确认，并将数据递交给相关的应用 进程，301至701这5个数据报则应当放于缓冲区，等到201这个数据报到达后，然后按序将201至701这些数据报递交给相关应用进程，并对701数据报进行 确认，确保了应用进程级的TCP数据的按序到达。

19、

|  |
| --- |
| SharedPreferences   * Use for things that should always be remembered, no matter if the phone is turned off (eg for settings chosen in the settings screen of your app   onSavedInstanceState   * Use this for remembering things about the current state of your activity such as the currently selected tab on the screen. This allows you to recreate the same state after a rotation or if the app was killed due to low memory. * The things saved in onSaveInstanceState will be forgotten after reboot, and when starting a new instance of an activity they will not be passed, so they are only for remembering the state of the activity |

static变量 vs saveInstance

Even if Android terminates the app, it can keep around the saved state information so that it can be restored at the last place the user left it (e.g. from the recent apps list). With static variables, that data is gone once the process terminates.

20、Thread和Runnable的区别

先说结论：推荐使用runnable

public class DemoRunnable implements Runnable{

public void run(){

...

}

}with a call like new Thread(DemoRunnable).start()

public class DemoThread extends Thread{

public void run(){

...}

}with a call like demoThread.start();

至于网传所谓runnable可以资源共享而thread不行其实是错误的。

线程同步大致有以下几个方法

同步run方法 synchronized方法

同步关键变量 syncchronized代码块

同步class对象

refer：<https://my.oschina.net/leejun2005/blog/483999>

21、双核CPU

就是双CPU。

双核cpu是不是能同时执行两条指令？事实上，从Pentium开始，一个CPU就可以同时执行超过1条指令。

Pentium拥有两条独立的管线，因此同一时刻可以发射两条指令。

Pentium 4不但具有2个ALU，同时还可以在一个时钟内执行2条简单指令，因此一个时钟周期可以执行4条指令。

以上并行被称之为指令级并行，换一句话说，对于编程语言和系统来说，它是透明的，它如同一个更快的顺序处理器，

双核心其实就是将两个独立的处理器核心做在一个封装中，因此无论从编程还是系统的角度看，它就是双处理器。

22、override和overload的区别

如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写 (Overriding)，重写的父类方法不能是private的，否则相当于在子类中重新定义了一个新方法，重写的方法要和父类中原来的方法完全一样，甚至包括抛出异常。如果在一个类中定义了多个同名的方法，它们或有不同的参数个数或有不同的参数类型，则称为方法的重载 (Overloading)，如果参数都一样，仅仅是返回值不一样，不能实现重载，同样不能重载private方法

23、空类的sizeof为1，因为必须在内存中占有一定的空间，否则无法使用这些实例，具体占用多少内存，由编译器决定，visual studio中每个空类型的实例占用1字节的空间

如果增加了构造函数和析构函数，还是1.析构函数，跟构造函数这些成员函数，是跟sizeof无关的，也不难理解因为我们的sizeof是针对实例，而普通成员函数，是针对类体的，一个类的成员函数，多个实例也共用相同的函数指针，所以自然不能归为实例的大小

但是如果增加了虚函数，就是4.因为要在每一个实例中添加一个指向虚函数表的指针。

如果一个类B是另一个类A的子类，那么求sizeofB的时候要把A的大小也加进来。

静态成员不算入sizeof，因为也不是实例独占的。

24、public class StaticDemo3

{

int var=13;

public static void main(String[] args){

System.out.println("成员变量var的值为："+var);

}

}

编译错误，其实很简单，静态方法只能访问静态成员。也不能在静态方法里面使用this。

需要注意的是静态成员变量要么在声明的同时初始化，要么在静态块里面初始化

25，函数指针

c语言

#include <stdio.h>

#include <stdlib.h>

int fun1()

{

printf("this is fun1 call\n");

return 1;

}

void fun2(int k, char c)

{

printf("this is fun2 call:%d %c\n", k, c);

}

int main()

{

int (\*pfun1)() = NULL;　　返回值 (\*函数名) (形参列表) (const)

void (\*pfun2)(int, char) = NULL;

int a,b;

pfun1 = fun1; //第一种赋值方法

a = pfun1(); //第一种调用方法（推荐）

printf("%d\n",a);

b = (\*pfun1)();//第二种调用方法

printf("%d\n",b);

pfun2 = &fun2;//第二种赋值方法（推荐，因为和其他数据指针赋值方法一致）

pfun2(1,'a');

(\*pfun2)(2,'b');

return 0;

}

c++

#include <iostream>

using namespace std;

class test

{

public:

test()

{

cout<<"constructor"<<endl;

}

int fun1(int a, char c)

{

cout<<"this is fun1 call:"<<a<<" "<<c<<endl;

return a;

}

void fun2(double d)const

{

cout<<"this is fun2 call:"<<d<<endl;

}

static double fun3(char buf[])

{

cout<<"this is fun3 call:"<<buf<<endl;

return 3.14;

}

};

int main()

{

// 类的静态成员函数指针和c的指针的用法相同

double (\*pstatic)(char buf[]) = NULL;//不需要加类名

pstatic = test::fun3; //可以不加取地址符号

pstatic("myclaa");

pstatic = &test::fun3;

(\*pstatic)("xyz");

//普通成员函数

int (test::\*pfun)(int, char) = NULL; //一定要加类名

pfun = &test::fun1; //一定要加取地址符号

test mytest;

(mytest.\*pfun)(1, 'a'); //调用是一定要加类的对象名和\*符号

//const 函数（基本普通成员函数相同）

void (test::\*pconst)(double)const = NULL; //一定要加const

pconst = &test::fun2;

test mytest2;

(mytest2.\*pconst)(3.33);

// //构造函数或者析构函数的指针，貌似不可以，不知道c++标准有没有规定不能有指向这两者的函数指针

// (test::\*pcon)() = NULL;

// pcon = &test.test;

// test mytest3;

// (mytest3.\*pcon)();

return 0;

}