## C++相关

### 左值和右值

1. 左值是有名字的变量（对象），可以被赋值，可以在多条语句中使用。左值声明符号为&。
2. 右值是临时变量，没有名字，只在一条语句中出现，不能被赋值，右值声明符号为&&。

(左值：int &a = 1。右值：int &&a=1)

左值[lvalue : locator value]: 存储在内存，有明确地址的值

右值[rvalue : read value]：提供数据的值，不一定可寻址；比如寄存器中的数据。右值又可以分为纯右值和将亡值

* 有名称的，可以获取存储地址的值即为左值，否则为右值
* 左值引用 T& 右值引用 T&&
* 同左值引用T&一样，右值引用T&&创建时就必须初始化且只能使用右值进行初始化

左右值最大的区别就是是否可以取地址，左值可以，右值不可以。

右值引用主要是为了用于移动语义和完美转发

### 右值引用的意义

1. 右值主要用于移动语义和完美转发
2. 为临时变量延长生命周期。因为右值在表达式结束后就消亡了，如果想继续使用右值，那就会动用（昂贵的）拷贝构造函数。
3. 移动语义：将资源从一个对象转移到一个对象，避免不必要的拷贝
   1. 右值引用可以消除两个对象交互时的不必要的对象拷贝，节省运算存储资源，提高效率。
   2. 右值引用是用来支持转移语义的。转移语义可以将资源（堆，系统对象等）从一个对象转移到另一个对象，这样能够减少不必要的临时对象的创建、拷贝和销毁，能够大幅度提高C++应用程序的性能。
4. 完美转发：函数模板可以将自己的参数 完美 的转发其内部调用的函数
   1. 完美的意思是 不仅准确的转发参数的值，而且还能保证转发参数的左，右属于不变。
   2. 内部调用时使用模板函数forward()，forward()函数用于转发，依赖于类型T
5. std::move() 将传入的参数转为右值，std::move()实际上是static\_cast<T&&>()的简单封装。
6. std::forward() 有两种实现，分别对左值引用和右值引用使用不同的模板函数，内部调用static\_cast

### emplace\_back()和push\_back()区别

1. 在c++11引入右值引用之前，push\_back() 是先调用构造函数创建一个临时对象，然后调用拷贝构造函数，将临时对象拷贝到容器中，最后释放临时对象，有一个申请空间，拷贝，和释放空间的过程, 导致临时变量申请资源的浪费。

在引入右值引用之后，则是创建临时对象，然后使用移动构造函数，将里临时对象加入对象。从而避免了一次拷贝过程和开辟空间的消耗，在原来的基础上提高效率。

1. emplace\_back()则是在这基础上，进一步改进，直接在容器中就地构造，没有移动这一步，进一步提高效率。

### [C++](https://www.nowcoder.com/jump/super-jump/word?word=C++)中struct与class区别，C与[C++](https://www.nowcoder.com/jump/super-jump/word?word=C++)中struct区别[C++](https://www.nowcoder.com/jump/super-jump/word?word=C++)中:

1. 在c中，struct主要是正对数据结构，是面向对象的设计，struct中不能包含函数。
2. c++中，struct得到很大的扩充，作用几乎和class类似
   1. struct可以包含成员函数
   2. struct可以实现继承
   3. struct可以实现多态
3. 在c++中strcut和class的区别：
   1. struct默认是公有继承，class默认是私有继承。
   2. class关键字可用于定于模板参数，功能和typename类似。struct不能。

### 1 C++面向对象的三大特性：

* 1. 封装: 将代码模块化，将其封装起来，对外隐藏，解决可扩展性。
  2. 继承：子类继承父类，实现了代码的可重用。
  3. 多态：一种接口，多种形态。

作用： 一种接口，多种方法【不同对象对同一行为会有不同的状态,实现接口重用】

实现形式： 静态多态 和 动态多态

静态多态： 通过 函数重载 和 函数模板 来实现

同一函数，其参数的个数，顺序，类型不同来实现重载

在编译时就已经确定对象的行为，又称为静态绑定。

动态多态：是面向对象的一大特色，通过继承方式使得程序在运行时才确定相应的调用函数，又称为动态绑定。通过 虚函数 来实现

在运行时确定具体调用那个函数。

### 寻函数，纯虚函数的理解

1. 虚函数

**虚函数作用： 实现多态**

**实现原理： 虚函数表（vtable）+ 虚表指针(vptr)**

虚表指针指向一个由虚函数的地址构成的一个数组（指针数组）

含有虚函数的类有一个虚函数表，每个类对象由一个虚表指针，指向虚函数表

虚函数表是全局共享的，属于类，全局共享一个，虚函数表类似一个数组，存储的是虚函数的地址，也即是虚函数的原始是指向类成员函数的指针。虚函数表的实现是编译器来实现的，编译器不同，可能实现方式不同

1. 纯虚函数

virtual bool function() = 0;

### [C++](https://www.nowcoder.com/jump/super-jump/word?word=C++)11的特性

* + 1. nullptr :
       1. 新引入的空指针，NULL本质是一个宏定义，本质是0,可能会导致二义性

eg： void fa(char\* p); void fb(int n);

* + 1. 类型推导： auto 和 decltype

auto 一般用于变量类型， auto使用时就必须初始化，在使用迭代器的时候比较方便

decltype用于表达式类型推导 ： 编译器期间完成，只进行类型推导，不会实际执行表达式。

* + 1. 右值引用 见1.1
    2. 移动语义
    3. 基于范围的for循环
    4. explicit : 修饰构造函数， 防止由构造函数定义的隐式转换。
    5. const 和 constexpr

两者都表示可读，const表明read-only constexptr才是真正的常量，在编译期就计算出来，整个运行期间不可改变。

const修饰的是类型，constexpr修饰的是用来算出值的那段代码。

* + 1. final 和 overrid关键字：
       1. final用于修饰一个类 表示禁止该类禁止被派生和虚函数进一步重载
       2. override修饰成员函数，用于重写基类函数，若声明为override，但是对父类没有该虚函数或重新，编译器报错
       3. override 可以避免开发者在重写基类函数时无意产生错误。
    2. Lambda

### RAII

   RAII（Resource Acquisition Is Initialization）是一种利用对象生命周期来控制程序资源（如内存、文件句柄、网络连接、互斥量等等）的简单技术。

用法：在对象构造时获取资源，使用资源，在对象析构时释放资源，借此，把管理资源的认为交给了一个对象。应用比如智能指针。

好处：不需要显示的释放资源，在对象所需的资源在其生命周期内始终保持有效，

RAII的本质时用对象代表资源，把管理资源的认为交给对象，把资源的申请和释放与对象的构造和析构对应起来，从而确保在对象的生命周期内资源始终有效，在对象销毁时资源被释放，避免内存泄漏等情况的发生。也即使对现在，资源在，对象消失，资源释放。

### 智能指针

* + - 1. 智能指针(smart pointer)是对象指针的类，用于生存期控制，能够确保自动正确的销毁动态分配的对象，防止内存泄露（利用自动调用类的析构函数来释放内存）。它是C++ RAII的一种应用。它可以使用使用引用计数(shared\_ptr)和所有权模式(unique\_ptr)两种方式。基于引用计数的智能指针类将一个计数器与类指向的对象相关联，引用计数跟踪该类有多少个对象共享同一指针。每次创建类的新对象时，初始化指针并将引用计数置为1；当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数；对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数所指对象的引用计数；调用析构函数时，构造函数减少引用计数（如果引用计数减至0，则删除基础对象）
      2. 普通指针容易造成内存泄漏，二次释放，程序发生异常时内存泄漏等问题，使用智能指针可用很好的管理内存，避免内存泄漏。
      3. shared\_ptr：采用引用计数的指针，多个shared\_ptr可用指向同一给对象，并维护一个共享的引用计数器。
      4. weak\_ptr：用于配合shared\_ptr的使用，并不邮箱对象的生命周期，也即是weak\_ptr存在与否不影响对引用计数，
      5. unique\_ptr ： 基于所有权模式的智能指针，任意时刻只能有一个拥有者，也不支持拷贝构造和赋值

### C++的this指针

1. this指针是类指针
2. this指针是一个隐士的指针，作为普通成员函数的参数，由编译器来完成
3. this指针的存储位置可能在栈，寄存器中，具体与编译器实现有关
4. this指针只能在普通成员函数中使用，在全局函数，静态函数中不能使用
5. this指针只有在成员函数中才有定义，不能通过对象来使用this指针，无法直接知道一个this指针的位置，但是可以在成员函数中指定this指针的位置。

[C++](https://www.nowcoder.com/jump/super-jump/word?word=C++)11的特性

### C++的类型转换有哪些

1. static\_cast：
   1. 静态类型转换[编译时转换]
   2. 用于各种类型的转换，
   3. 在编译时完成
2. dynamic\_cast
   1. 用于动态类型转换
   2. 只能用于含有虚函数的类，用于类层次的向和向下转换，
   3. 失败返回NULL
3. const\_cast：

用于将const变量转为非const【去除常量属性】

1. reinterupt\_cast：几乎什么都可以转，比如int转为指针，但可能会出问题
2. 为什么不用C的强制类型转换？
   1. C的强制类型转换看起来很强大，什么都可以转，但是转化不明确，不进行错误检查，容易出错。

### new和malloc区别以及 malloc原理

malloc用于动态分配内存。为了减少内存碎片和系统调用的开销，malloc采用内存池的方式，先申请大块内存作为堆区，然后将堆区分为多个内存块，以块为内存管理的基本单位。当用户申请内存时，直接从堆区分配一块合适的内存块。malloc采用隐式链表结构将堆区分成连续的、大小不一的块，包含分配块和非分配块；同时malloc采用显式链表结构来管理所有的空闲块，即使用一个双向链表将空闲块连接起来，每一个空闲块记录了一个连续的、未分配的地址。

malloc在申请内存时，一般会通过brk或者mmap系统调用进行申请。其中当申请内存小于128k时，会使用系统函数

malloc是从堆中分配的，

brk与sbrk， mmap与munmap

a. brk/sbrk/mmap属于系统调用，如果每次申请内存，都调用其中一个，系统调用开销较大，若每次都用系统调用，则会影响性能。

b.这样申请内存容易产生内存碎片，因为堆是从低地址往高地址方向增长，若低地址没有释放，高地址内存就不能回收，

因此，malloc采用内存池的方法，来减少内存碎片的产生和减少系统调用的开销。先申请一个大内存，然后讲其分为不同大小的内存块，并以块为管理的基本单位。用户申请书内存时，就直接从内存池中选择一块相近的内存块即可。

malloc使用两种链表来管理内存块：

隐式空闲链表：将堆区分为地址连续，大小不一的块，包含已分配和未分配的内存块。

显示空闲链表：用一个链表将可用的内存块连接起来，组成一个双向的空闲链表，链表节点含有一个前向指针和后继指针，每个节点记录一个地址连续，未被使用的内存块，节点同时记录内存首地址和大小。

malloc申请内存小于128k时，使用 brk分配内存，申请内存大于128k时，使用mmap分配内存brk和mmap这两种方式分配的都是虚拟内存，没有分配物理内存；当第一次访问已分配的虚拟地址空间时，会发生缺页中断，操作系统负责物理分配物理内存，然后建立虚拟内存和物理内存之间的映射关系。



new/delete 是运算符，malloc/free是函数。

new内存分配失败时，会抛出bac\_alloc异常，它**不会返回NULL**；malloc分配内存失败时返回NULL。

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

C++提供了new[]与delete[]来专门处理数组类型:，至于malloc，它并知道你在这块内存上要放的数组还是啥别的东西，反正它就给你一块原始的内存，在给你个内存的地址就完事。

new操作符从**自由存储区（free store）**上为对象动态分配内存空间，而malloc函数从**堆**上动态分配内存。

1. 自由存储是C++中通过new与delete动态分配和释放对象的抽象概念，而堆（heap）是C语言和操作系统的术语，是操作系统维护的一块动态分配内存。
2. new所申请的内存区域在C++中称为自由存储区。藉由堆实现的自由存储，可以说new所申请的内存区域在堆上。
3. 堆与自由存储区还是有区别的，它们并非等价。

### new、operator new 和 placement new的确保：

1. **new operator:**

不能被重载，执行过程如下：

1. 调用operator new 分配内存
2. 调用并执行构造函数
3. 返回相应类型的指针

delete过程 ：

* + 1. 调用析构函数
    2. 调用operator delete,释放空间

1. **operator new**

实现内存分配，重载时new时，实际上重载的是operator new,

要实现不同的内存分配行为，应该重载operator new

1. **placement new：**

operator new 只是operator new的一个重载版本

* 1. 并不分配内存，只是返回指向已分配的某段内存的一个指针
  2. 因此，不能删除，但需要调用对象的析构函数。
  3. placement new 允许在一个已分配的内存中构造一个新对象。
  4. 引入 placement new的理由：

**用 placement new 解决buffer问题：**

*若是用new分配数组缓冲时，先要申请分配内存，然后调用构造函数，效率不急啊；使用 placement new 可以直接在已预分配的内存上构造函数*。

增大时空效率问题：

*使用 new 申请分配空间时，需要在堆中查找足够大的空间，然后分配，这个操作比较慢，同时可能出现内存不足，内存分配异常；而 placement new 是在一个预先已分配的内存缓冲区将那些，不需要查找内存，这样内存分配的时间就就是常数，且不会出现内存不足的异常。因此，placement new非常适合对时间要求较高，长时间运行不希望被打断的应用程序。*

### 引用和指针有什么区别

1. 指针有自己的一块空间，而引用只是一个别名。
2. 指针可以被初始化为NULL，而引用必须被初始化且必须是一个已有对象的引用。
3. 使用sizeof看一个指针的大小是4，而引用则是被引用对象的大小。
4. 指针在使用中可以指向其他对象，但是引用只能是一个对象的引用，不可被改变。

### sizeof和strlen()的区别:

1. sizeof 是一个操作符， strlen()是一个库函数
2. sizeof的参数可以是数据类，也可以是变量， 而strlen()只能以结尾’\0’的字符串作为参数
3. sizeof在编译时期就确定了结果，strlen()需要在运行时才能计算出结果
4. sizeof 计算的是分配的内存大小，strlen()则是字符串实际的长度
5. 数组作为sizeof参数时，不退化， 作为strlen()参数时，则退化为指针。

### 产生coredump文件的原因

1. 访问内存越界（如使用下标访问字符串未进行结束符判断，strcpy，strcmp,strcat等越界）
2. 非法指针(使用空指针或随意转使用指针转换)
3. 堆栈溢出
4. 多线程使用了线程不安全的函数
5. 多线程读写的数据没有加锁保护

定位coredump文件：

1. 先编译生成含有调试信息的可执行文件 gcc –g test.cpp –o test
2. 在执行 ./test,从而产生coredump文件
3. 在执行 gdb ./test ./core
4. 在gdb下输入 where

### gdb调试

g++ -g test.cpp –o test

./test

l (list) 罗列出当前代码信息

b (break point) 设置断点

r (run) 运行直到断点或结束

n (next) 单步调试，不会进入函数 类似 vs的 F10

s (step) 单步调试 会进入函数体 类似 vs的 F11

f (frame) 切换函数的栈帧

p (print) 打印

t (thread) 切换线程

c (continue) 程序继续执行直到下一个断点或程序结束

bt (backtrace) 显示堆栈信息

i (info) 查看信息

d (delete) 删除断点

fin (finish) 退出函数

### 缓存淘汰算法

LRU（least recently used, 最近最少使用），根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

1. 新数据插入到链表头部；
2. 每当缓存命中（即缓存数据被访问），则将数据移到链表头部；
3. 当链表满的时候，将链表尾部的数据丢弃。

### C++ 11智能指针介绍

智能指针主要用于管理在**堆**上分配的内存，它将普通的指针封装为一个**栈对象**。当栈对象的生存周期结束后，会在析构函数中释放掉申请的内存，从而防止内存泄漏。

C++ 11中最常用的智能指针类型为**shared\_ptr,它采用引用计数的方法，记录当前内存资源被多少个智能指针引用**。该引用计数的内存在堆上分配。当新增一个时引用计数加1，当过期时引用计数减一。只有引用计数为0时，智能指针才会自动释放引用的内存资源。对shared\_ptr进行初始化时不能将一个普通指针直接赋值给智能指针，因为一个是指针，一个是类。可以通过make\_shared函数或者通过构造函数传入普通指针。并可以通过get函数获得普通指针。

**1)为什么要使用智能指针**

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

### Valgrind

* + - * 1. 内存分析工具，由**Memcheck**，Cachegrind，Helgrind等组成。
        2. **Memcheck**是一款 内存泄漏 检测工具
        3. **Memcheck**能够检测出内存问题的原理：关键在于建立两个全局表：

**Valid-Value表：**对于进程的整个地址空间的每一个字节(Byte）,都与之对应的 8 bit，这些bit负责记录该字节或寄存器的值是否有效，是否被初始化等。（CPU的每个继承寄存器，也有与之对的bit向量）

**Valid-Address 表：**对于进程整个地址空间中每一个字节（Byte）,还有与之对应的1bit,负责记录该地是否能够被读写。

* + - * 1. **MemCheck原理：**

当要读写某个字节时，首先检测该字节的 A bit,若A bit显示无效位置，memcheck则报告错误。

当字节加载到cpu时，它对应的 v bit也被加载到cpu环境中，首先检测对应的 v bit，若为初始化，则memcheck报错。

### 进程通讯的几种方法

1. 管道。管道的机制类似于缓存，是单向传输的，效率较为低下，比如：a进程给b进程传输数据，只能等待b进程取了数据之后a进程才能返回。
2. 消息队列。优点：把进程的数据放在某内存以后，无需等待其他进程来取就返回。缺点：

如果a进程发送的数据占内存比较大，并且两个进程之间通信较为频繁，消息队列模型就不合适，因为a发送的数据很大的话，发送消息（拷贝）这个过程需要花很多时间来读内存。

1. 共享内存。系统加载一个进程的时候，分配给进程的内存并不是实际物理内存，二是虚拟内存空间。将两个进程的虚拟内存空间映射到同一个实际物理内存中，就实现了内存共享的机制。
2. 信号量。为了解决共享内存的线程安全问题，引入信号量的通信方式。
3. Socket。

### linux进程间共享内存如何同步

1. **利用POSIX有名信号灯实现共享内存的同步。**两个进程，对同一个共享内容读写。利用两个有名信号量semr，smew。Semr信号量控制能否读，初始化为0。Smew信号量控制能否写，初始为1。

* 读共享内存的程序示例代码：

Semr = sem\_open(“mysem\_r”,O\_CREAT | O\_RDWR, 0666,0);

1. **利用POSIX无名信号灯实现**。POSIX无名信号量是基于内存的信号量，可以用于线程间同步也可以用于进程间同步，需要在共享内存中创建无名信号量。
2. **利用信号实现共享内存的同步**。信号是软件层次上对中断机制的一种模拟，是一种异步通信方式。利用信号也可以实现共享内存的同步。

### 文件读写基本流程

**读文件**

1. 进程调用库函数向内核发起读文件请求；
2. 内核通过检查进程的文件描述符定位到虚拟文件系统的已打开文件列表表项；
3. 调用该文件可用的系统调用函数read()；
4. Read()函数通过文件表链接到目录项模块，根据传入的文件路径，在目录项模块中检索，找到该文件的inode；
5. 在inode中，通过文件内容偏移量计算出要读取的页；
6. 通过inode找到文件对应的address\_space
7. 在address\_space中访问该文件的页缓存树，查找对应的页缓存节点：
8. 如果页缓存命中，那么直接返回文件内容；
9. 如果页缓存缺失，那么产生一个页缺失异常，创建一个页缓存页，同时通过inode找到文件该页的磁盘地址，读取相应的页填充该缓存页；重新进行第7步查找页缓存；
10. 文件内容读取成功。

**写文件**

前6步和读文件一致，在address\_space中查询对应页的页缓存是否存在：

7）如果页缓存命中，直接将文件内容修改更新在此页中。写文件就结束了。这时候文件修改位于页缓存，并没有写回到磁盘文件中去。

8）如果页缓存缺失，那么产生一个缺页异常，创建一个页缓存页，同时通过inode找到文件该页的磁盘地址，读取相应的页填充该缓存页。此时缓存页命中，进行第7步。

9）一个页缓存中的页如果被修改，那么会被标记为脏页。脏页需要写回到磁盘中的文件块。有两种方式：

（1）手动调用sync()或者fsync()系统调用能够把脏页写回。

（2）pdflush进程会定时把脏页写回到磁盘

同时注意，脏页不能被置换出内存，如果脏页正在被写回，那么会被设置写回标记，这时候该页就被上锁，其他写请求被阻塞直到锁释放。

### 编译的过程

编译的分别四部，分别如下：

1. **预处理** :
   1. 命令 g++ -E test.cpp -o test.i
   2. 主要处理以 # 开始的预编译指令， 如 #include、 #define 和 注释
   3. #pragma 则保留，编译器需要它们
2. **编译 :** 
   1. 命令 ： g++ -S test.i –o test.s
   2. 主要进行词法分析，语法分析，语义分析等
3. **汇编：**
   1. 命令 ：g++ -c tes.s –o test.o
4. **链接：**
   1. 静态链接 ： 在编译期间就确定了
      1. 对库函数的链接是在编译期间完成的。 Libxxx.a
   2. 动态链接 ： 在运行时再载入
      1. 在程序运行时期才载入

### g++与gcc的区别:

1. 它们都可以编译.c和.cpp文件；后缀为 .c 的文件， gcc把它当作C程序， g++ 则将它当作 C++ 程序，对于 .cpp文件，gcc和g++都将其视为 c++程序。
2. 编译阶段，g++会调用gcc，对于cpp文件，两者等价，但gcc 命令不能自动和C++程序使用的库链接。因此，对于cpp文件，一般使用g++。
3. 对于 extern ”C” ，gcc/g++都将以C的方式进行编译。

### epoll 怎么实现的？

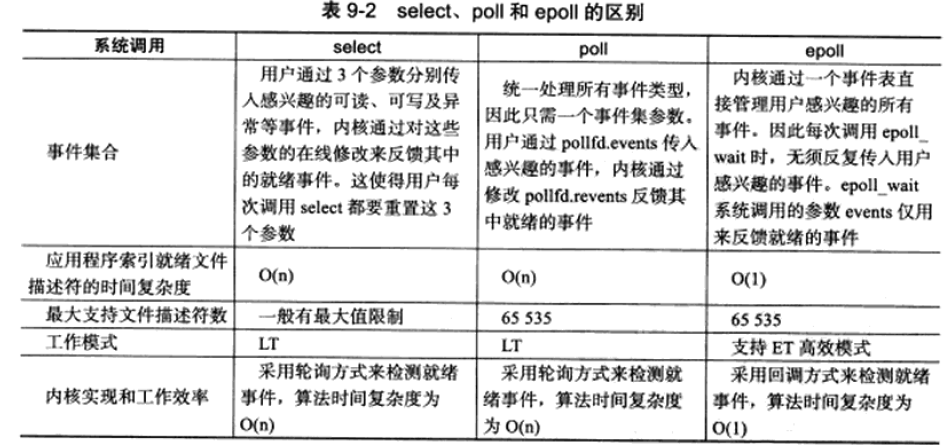
1.linux epoll机制是通过红黑树和双向链表实现的。

* 1. 首先通过epoll\_create()系统调用在内核中创建一个eventpoll类型的句柄，其中包括红黑树的根节点和双向链表的头结点。
  2. 然后通过epoll\_ctl()系统调用，向epoll对象的红黑树结构中添加、删除和修改感兴趣的事件，返回0表示成功，返回-1表示失败
  3. 。最后通过epoll\_wait()系统调用判断双向链表是否为空，如果为空则阻塞。当文件描述符状态改变，fd上的回调函数被调用，该函数将fd加入到双向链表中，此时epoll\_wait()函数被唤醒，返回就绪好的事件。

1. 什么是epoll
2. epoll是为处理大批量句柄而做了改进的poll，是性能最好的多路I/O就绪通知方法；
3. 只有三个系统调用：epoll\_create, epoll\_ctl, epoll\_wait;
4. epoll\_ctl ，epoll的事件注册函数，它不同于select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里注册要监听的事件类型；
5. epoll工作原理
6. epoll只告知那些就绪的文件描述符，而且当我们调用epoll\_wait()获得就绪文件描述符时，返回的不是实际的描述符，而是一个代表就绪描述符数量的值。
7. 只需要去epoll指定的一个数组中依次取得相应数量的文件描述符即可，这里也使用了内存映射技术，这样便彻底省掉了这些文件描述符在系统调用时复制的开销。
8. 在select/poll中，进程只有在调用一定的方法后，内核才对所有监视的文件描述符进行扫描；而epoll事先通过epoll\_ctl()来注册一个文件描述符，一旦基于某个文件描述符就绪时，内核会采用类似callback的回调机制，迅速激活这个文件描述符，当进程调用epoll\_wait()时便得到通知；
9. .epoll总结：

**一颗红黑树，一张准备就绪句柄链表**，少量的内核cache，就帮我们解决了大并发下的socket处理问题。

* 1. 执行epoll\_create()时，创建了红黑树和就绪链表；
  2. 执行epoll\_ctl()时，如果增加socket句柄，则检查在红黑树中是否存在，存在立即返回，不存在则添加到树干上，然后向**内核注册回调函数**，用于事件发生时向准备就绪链表中插入数据；
  3. 执行epoll\_wait()时立刻返回准备就绪链表里的数据即可



### 1.25为什么析构函数必须是虚函数？为什么C++默认的析构函数不是虚函数？

答：

1. 将可能被继承的父类的析构函数设置为虚函数，可以保证当我们new一个子类，然后使用基类指针指向该子类对象时，释放基类指针时可以释放掉子类的空间。
2. 默认的析构函数不是虚函数是因为虚函数需要额外的虚函数表和虚函数指针，占用额外的内存。而对于不会被继承的类来说，其析构函数如果是虚函数，就会浪费内存。因此C++默认的析构函数不是虚函数，而是只有当需要当做父类时，才设置为虚函数。
3. **构造函数为什么不能为虚函数？**
   1. 在构造对象时必须知道确切的类型
   2. 在构造函数之前，对象时不存在的，从而就无法使用指向此对象的指针来调用构造函数**。**

### 容器相关

| **容器**  **类型** | **容器特性** |
| --- | --- |
| vector | 1.序列容器，内存是连续，可以随机访问元素，任意元素的读取、修改具有常数时间复杂度，在序列尾部进行插入、删除是常数时间复杂度，但在序列的头部插入、删除的时间复杂度是O(n)，插入和删除操作会使得其之后的迭代器失效，因为vector是连续的地址，插入和删除时，需要移动元素。  2. 在尾部添加元素的push\_back()平均复杂度为O(1) ,推导如下：  若插入N个元素，采用2那里增长的方式，则会引发 lgN次的内存扩充，每次内存扩充，都会有一个过程：申请内存 –> 拷贝元素 –> 释放原来的内存，则拷贝的次数为 2^0 ,2^1, 2^2,…, 2^lgN;则所有拷贝次数相加可以得到的总拷贝次数：2^0 + 2^1 + 2^2 + … + 2^lgN = 2\*2^lgN – 1 ≈ 2N ,若在push\_back()，则又会扩容，每个元素拷贝一次，共拷贝了N次，所有总的操作次数为3N,平均下来就是3次 因此每个push\_back()平均复杂度为常数事件复杂度O(1).  3是动态可增长数组。增长时，一般为现有capacity的1.5或2倍方式，1.5倍的方式可以使用前面重复的内存,2倍增长则不可以。  4.适合于大量的随机访问，若在任意位置插入删除较多，则不适应。 |
| deque | 1. deque是双端队列，是一种序列容器，和vector类似，但又有不同。  2．deque是由一段一段的定量连续的空间构成， deque采用map作为主控，map是一小块连续空间，每个元素都是指针，指向另外一段线性连续空间；deque提供了两级数组结构，第一季完全类似vector，代表实际容器，另外一级维护容器的首地址。此外，deque还支持高效首/尾插入和删除操作。push\_front()和push\_back()。deque在元素添加时扩充空间，因此没有capacity。  3.缺点就是占用内存过多。 |
| list | 1.list是双向链表，非连续的地址空间。每个元素都维护一个前向和后续指针，因而支持前向和后向遍历。  2.支持高效的随机插入和删除操作，但不能随机访问。  3.C++11新增了forward\_list是单项链表。 |
| set | 1.set是关联容器，元素不允许有重复，底层实现是采用的是红黑树，查找的速度非常快，时间复杂度是O(logN) |
| multiset | 1. 关联容器，和set一样，允许有重复的元素，具备时间复杂度O(logN)查找功能。 |
| map | 1.关联容器，按照{键，值}方式组成集合，底层实现是采用的是红黑树，查找的时间复杂度O(logN)，其中键不允许重复 |
| multimap | 和map一样，区别是键可以重复 |

## 2. 网络相关

### 网路模型：

七层 五层 四层

应用层 应用层 应用层

表示层

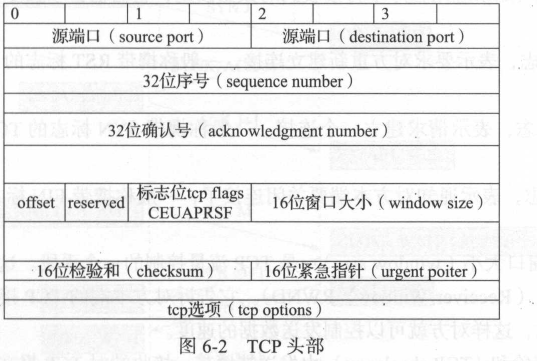
会话层

传输层 传输层 传输层

网络层 网路层 网络层

数据链路层 数据链路层

物理层 物理层 网际接口层



### TCP如何保证传输可靠性

1. 校验和。发送的数据包的二进制相加然后取反，目的是检测数据在传输过程中是否变化。如果检验和有差错，TCP将丢弃这个报文段和不确认收到此报文段。
2. 确认应答和序列号。TCP给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。
3. 超时重传。当TCP发出一个段后，启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。
4. 流量控制。TCP连接的每一方都有固定大小的缓冲空间，TCP的接受端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送速率，防止包丢失。TCP使用的流量控制协议是可变大小的滑动窗口协议。接受方有即时窗口（滑动窗口），随ACK报文发送。
5. 拥塞控制。当网络拥塞时，减少数据的发送。（慢启动、拥塞避免、拥塞发送、快速恢复）

### http1.1、http1.0和http2.0区别

* http1.1和http1.0区别：

1）1.1默认支持长连接。

2）1.1带宽优化，并支持断点续传。这是因为1.1在请求消息里加入range头域，它支持只请求资源的某个部分。在响应消息中Content-Range头域声明了返回的这部分对象的偏移值和长度。

3）新增例如ETag, If-None-Match等更多的缓存控制策略

4）Host头域

5）新增了24个错误状态响应码。

* http2.0 与 http1.1相比有以下不同：

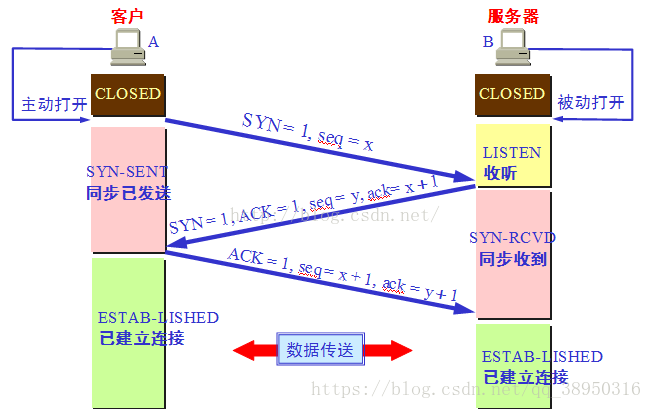
1. 多路复用，可以做到在一个连接并行的处理多个请求
2. Header压缩
3. 服务端推送；
4. 解析格式不同，1.0和1.1的解析是基于文本，2.0协议解析采用二进制，实现方便且健壮。

### https和http

https特点：

1. 内容加密：采用混合加密技术，中间者无法直接查看明文内容
2. 验证身份：通过证书认证客户端访问的是自己的服务器
3. 保护数据完整性：防止传输的内容被中间人冒充或者篡改

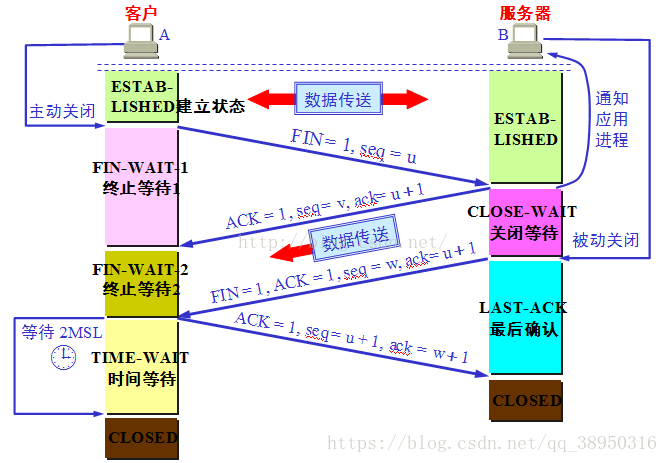
### TCP三次握手和四次挥手过程理解



第一次握手：建立连接时，客户端发送syn包（syn=x）到服务器，并进入SYN\_SENT状态，等待服务器确认；SYN：同步序列编号（Synchronize Sequence Numbers）。

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=x+1），同时自己也发送一个SYN包（syn=y），即SYN+ACK包，此时服务器进入SYN\_RECV状态；

第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=y+1），此包发送完毕，客户端和服务器进入ESTABLISHED（TCP连接成功）状态，完成三次握手。



1）客户端进程发出连接释放报文，并且停止发送数据。释放数据报文首部，FIN=1，其序列号为seq=u（等于前面已经传送过来的数据的最后一个字节的序号加1），此时，客户端进入FIN-WAIT-1（终止等待1）状态。 TCP规定，FIN报文段即使不携带数据，也要消耗一个序号。

2）服务器收到连接释放报文，发出确认报文，ACK=1，ack=u+1，并且带上自己的序列号seq=v，此时，服务端就进入了CLOSE-WAIT（关闭等待）状态。TCP服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个CLOSE-WAIT状态持续的时间。

3）客户端收到服务器的确认请求后，此时，客户端就进入FIN-WAIT-2（终止等待2）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的数据）。

4）服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，FIN=1，ack=u+1，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为seq=w，此时，服务器就进入了LAST-ACK（最后确认）状态，等待客户端的确认。

5）客户端收到服务器的连接释放报文后，必须发出确认，ACK=1，ack=w+1，而自己的序列号是seq=u+1，此时，客户端就进入了TIME-WAIT（时间等待）状态。注意此时TCP连接还没有释放，必须经过2∗∗MSL（最长报文段寿命）的时间后，当客户端撤销相应的TCB后，才进入CLOSED状态。

6）服务器只要收到了客户端发出的确认，立即进入CLOSED状态。同样，撤销TCB后，就结束了这次的TCP连接。可以看到，服务器结束TCP连接的时间要比客户端早一些。

### 2.5 从输入网址到获得页面的过程

1) 浏览器查询DNS，获取域名对应的IP地址：具体过程包括浏览器搜索自身的DNS缓存、搜索操作系统的DNS缓存、读取本地的Host文件和向本地DNS服务器进行查询等。对于向本地DNS服务器进行查询，如果要查询的域名包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析；如果要查询的域名不由本地DNS服务器区域解析，但该服务器已经缓存了此网址映射关系，则调用这个IP地址映射，完成域名解析。如果本地域名服务器并未缓存该网址映射关系，那么将根据其设置发起递归查询或迭代查询。

2）浏览器获得域名对应的IP地址后，浏览器向服务器请求建立连接，发起三次握手

3）TCP/IP连接建立起来后，浏览器向服务器发送HTTP请求

4）服务器接收到这个请求，根据路径参数映射到特定的请求处理器进行处理，并将处理结果返回给浏览器

5）浏览器根据请求到的资源，最终向用户呈现一个完整的页面

## 3．算法

### 3.1 堆排序

a. 将无序序列构建成一个堆，根据升序降序需求选择大顶堆或小顶堆；

b. 将堆顶元素与末尾元素交换，将最大元素“沉”到数组末端；

c. 重新调整结构，使其满足堆定义，然后继续交换堆顶元素与当前末尾元素，反复执行调整和交换步骤，直到整个序列有序。

## 4．设计模式

### 4.1 单例模式

class Singleton

{

public:

~Singleton() {

}

Singleton(const Singleton& other) = delete;

static Singleton& getInstance()

{

static Singleton instance;

return instance;

}

private:

Singleton() {

}

};

新建一个实例：

Singleton &a = Singleton::getInstance();

### 4.2 工厂模式

工厂模式主要解决接口选择的问题。该模式下定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，使其创建过程延迟到子类进行。

## 操作系统

### 5.1 进程与线程的区别

1）一个进程可以有多个线程，而一个线程只能属于一个进程程。

2）进程是资源分配的最小单位，线程是系统调度的最小单位。

3）进程用于自己的独立地址空间。系统创建一个进行时，会为其分配地址空间，建立数据段，代码段和堆栈段；线程没有自己的地址空间，只用于一些必不可少的资源，如线程栈，程序状态寄存器wpc， 程序计数器pc等。同一个进程的线程共享进行的地址空间。

4）进程的切换开销比线程的切换开销大。进程创建和销毁时，系统都要为之分配或回收资源，而线程所拥有的资源很少，因此，操作系统所付出的开销显著大于创建或撤销线程时的开销。在进行进程切换时，涉及到整个当前进程CPU环境的保存以及新被调度运行的进程的CPU环境的设置。而线程切换只须保存和设置少量寄存器内容。

5）多进程程序更安全。一个进程崩溃不会对另外一个进程造成影响，而同一进程的多个线程，若某线程崩溃，可能导致整个进程崩溃。

6）线程之间通信更加方便，同一个进程间的多个线程，可以通过全局变量等方式实现通信，而进程之间需要通过IPC方式进行。

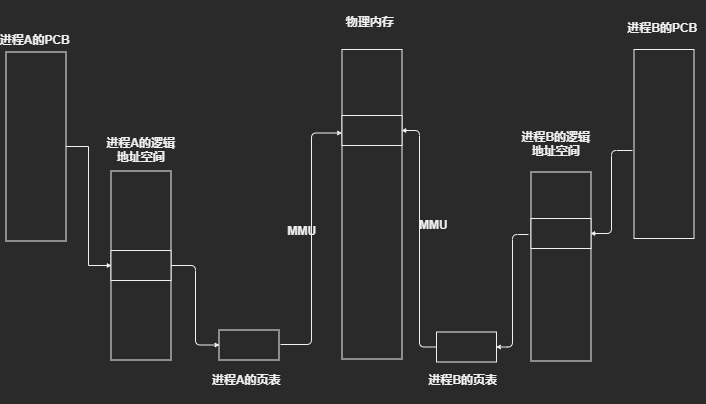
### 5.2 有了进程为什么还需要线程？

线程产生的原因： 进程可以使得多个程序能并发的执行以提高资源的利用率和系统吞吐量，但是同一进程在同一时刻只能干一件事。

若进程阻塞，整个进程都会挂起，即使进程中某些认为不需要等待的资源，因此，操作系统引入了比进程更小的线程作为并发执行的基本单位，从而减少程序在并发执行时所需要付出的时空开销，进一步提高并发性。

### 5.3进程间通信 IPC

1. 管道：
   1. 包括无名管道和命名管道。无名管道用于具有亲缘关系的父子进程间通信。命名管道则没限制。
   2. 管道是一种两个进程间单向通信的机制，是半双工通信。
   3. 管道的本质是固定大小的内核缓冲区，因而大小是由限制的。可以通过unlimited –a 查看，大小一般为 4k Byte。
2. 消息队列：
   1. 用于同一台机器上的进程间通信，在内核中用来保存消息的队列。
   2. 在内核中以消息链表的形式出现。
   3. 相比于管道，消息队列可以独立发送和结束，可以通过消息类型来选择性的结束数据。
3. 信号量(system V)：
   1. 信号量用于实现进程间互斥和同步。
4. **共享内存**:
   1. 多个进程将通过MMU将逻辑地址映射到同一物理地址上，从而进程间共享内存。允许两个或多个进程访问同一内存，当某一个进程改变了内存中的内容时，其他进程都会察觉到这个变化。
   2. 共享内存没有提供同步机制，需要使用同步机制来保证对内存访问，避免产生竞态条件，通常使用信号量机制来实现进程间同步与互斥。
   3. 相关API
      * 1. shmget():创建或获取一段共享内存
        2. shmat()：关联到进程的地址空间
        3. shmdt()：从进程地址空间分离
        4. shmctl()：更改共享内存属性
   4. 原理： 每个进程有自己的PCB和地址空间及页表。页表通过MMU管理，将逻辑地址地址和物理地址进程映射。可以将两个进程映射都同一块物理内存，实现共享内存。



AB两个进程通过共享内存来通信示意

* 1. 共享内存优点：

进程间通信效率高，通过直接访问内存实现通信，避免数据拷贝和系统调用。

但是需要解决进程间同步比如信号量等方式，来避免竞态。

1. Socket: 注意用于网络间的通信。
   1. socket(), bind(), listen, accept()

### 5.4 进程间通信相关命令(ipcs)

ipcs是一个Linux命令，用于报告系统的消息队列，信号量共享内存等。

ipcs –a 列出本用户所有相关ipcs参数。

ipcs –q 列出进程中的消息队列

ipcs -s 列出所有信号量

ipcs –m 列出所有的共享内存信息

ipcs –l 列出系统的限额

ipcs –t 列出最后的访问时间

ipcs –u 列出当前的使用情况

### 5.5 线程间同步通信

1）临界区

2）锁机制

1. 互斥锁(mutex)：

为了确保同一时间只有一个线程访问数据，在访问共享资源前需要对互斥量上锁。一旦对互斥 量上锁后，任何其他试图再次对互斥量上锁的线程都会被阻塞，即进入等待队列，当其他线程释放互斥 量后，操作系统会激活那个被挂起的线程，让其投入运行。[加锁失败，被阻塞进入等待队列]

1. 自旋锁(reader-writer lock)：

自旋锁与互斥量最不同的是：非阻塞锁，它不会被挂起，而是在获取锁之前一直处于忙等，即 不停在消耗cpu，执行循环。适用于多核处理器、临界区无阻塞情况，其中一个 CPU 上的线程进入临界区，另一个 CPU 上的线程尝试获取锁会自旋，因为它不会阻塞，所以只要稍稍等一下下就能进入临界区（预计线程等待锁的时间很短，短到比线程两次上下文切换时间要少的情况下），对于多核 CPU 来说，会提高并发率。[加锁失败，不被阻塞，而是忙等]

1. 读写锁(spin lock)：

可以多个线程同时读，但是不能多个线程同时写，锁处于读模式时可以线程共享，而锁处于写 模式时只能独占，所以读写锁又叫做共享-独占锁，读写锁比互斥锁更加具有适用性和并行性，最适用于对数据结构的读操作读操作次数多余写操作次数的场合。(适合读多写少的场景)

3）信号量(sem):

信号量的实现中一般具有两种操作，分别是P操作和V操作。对一个信号量进行P操作时，首先检查其值是否大于0，如果大于0，则将其值减一之后返回进行后续操作，如果值小于等于0，则该进程将进行阻塞。(P，V都是原子操作)

信号量（sem）和互斥锁的区别：互斥锁只允许一个线程进入临界区，而信号量允许多个线程 进入临界区。

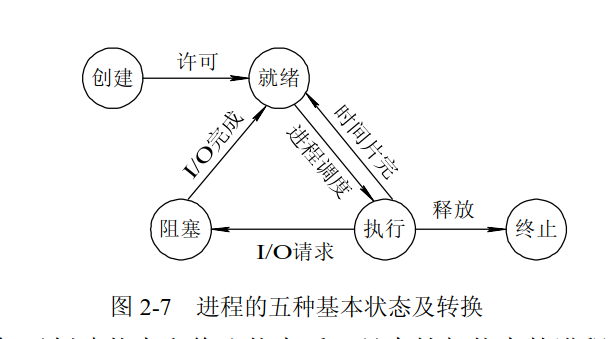
4）条件变量(cond)

当线程在等待满足某些条件时使线程进入睡眠状态，一旦条件满足，就唤醒线程

在条件变量的内部，有一次解锁加锁过程，条件不满足，将本线程加入等待队列，同时将传入的 mutex变量解锁，一旦等待队列中的线程被唤醒，会再次对传入的 mutex变量加锁！[条件不满足时，将线程放入队列中，对mutex变量进行解锁（这两步都是原子操作）一旦线程被唤醒，就对mutex变量进行加锁]

条件变量产生的原因：

互斥量不是万能的，比如某个线程正在等待共享数据内某个条件出现，可能需要重复对数 据对象加锁和解锁（轮询），但是这样轮询非常耗费时间和资源，而且效率非常低，所以互斥锁不太适



### 5.6 进程相关概念

* 1. 进程一般由3部分组成：数据段，堆栈段和堆栈段，其中进程控制块PCB存放在堆栈段。
  2. PCB进程控制块，PCB是进程存在的唯一标识，系统通过感知PCB的存在而感知进程的存在。
  3. 系统通过对PCB来对进程管理和调度，比如进程的创建，调度，优先级等都是通过PCB来实现处理的。
  4. 由两种创建进程的方式:
     + 1. 系统创建:比如Linux,开机时内核执行0号进程，它是所有进程的主线，0号进程创建1号进程(inti 1 内核进程)再继续执行下去。大致过程为：

0号进程->1号内核进程->1号用户进程。[0号进程通过系统自举时由系统创建]

* + - 1. 由父进程创建(fork,vfork,clone)
         1. fork返回两次的原因：

由于复制时复制了父进程的堆栈段，因此两个进程都停留在fork函数中，fork返回两次，一次是在父进程中返回(返回的子进程的id)，另一次是在子进程中返回(0)，返回值不同;出错返回-1。

* + - * 1. fork与vfork及clone

fork

创建一个和父进程一样的进程，完全复制父进程的资源，现在的Linux系统一般都采用copy-on-write(COW)技术，降低开销，也即是在一开始并不会复制父进程的资源，当要对数据写入时，才进行拷贝，从而降低开销。

fork()会返回两次，父子进程执行顺序不定。

vfork ：

vfork也是创建子进程，但是vfork创建的子进程与父进程共享地址空间，也即是子进程完全允许在父进程地址空间上。

vfork()创建的子进程要显示调用 exit()来结束。

vfork创建的子进程的执行顺序是固定的，先执行子进程，然后执行父进程。

vfork返回值和fork一样。

clone:

fork和vfork都是无参的，clone带有参数， fork是全部复制，vfork是共享，clone则是将父进程的资源有选择性的复制给子进程，而为复制的数据结构则通过复制指针的方式让子进程共享，具体复制哪些资源有参数列表指定。

clone()返回的是子进程的pid。

### 5.7 返回值

正常退出返回的方式：

1. return : 是函数执行完后返回，return把控制权交给调用函数
2. exit() ： 是一个函数，带有参数，exit执行后把控制权交给系统, 0 时，代表进程正常终止，若为其他值，表示程序执行过程中有错误发生。
3. \_exit()
4. exit与 \_exit()的区别：
   1. 两个都是函数，用来终止进程的。
   2. 头文件不同，exit()在stdlib.h, \_exit()在unistd.h
   3. \_exit()不会刷新流(stdin,stdout,stderr), exit()会刷新数据流。
   4. exit()是在\_exit()基础之上一个封装。
   5. 两个最大的区别：exit()会把文件缓冲区的内容写回文件，而\_exit()不会写回，导致缓冲区的数据丢失。

异常退出的返回方式：

* + - 1. 调用abort(),
      2. 进程收到信号，使得程序终止

### 5.8 孤儿进程与僵尸进程

子进程是通过父进程创建的。子进程的结束和父进程的运行是一个异步过程，即父进程永远无法预测子进程到底什么时候结束。于是就产生了孤儿进程和僵尸进程

孤儿进程：父进程退出，而其子进程还在运行，则这些子进程将成为孤儿进程，这些孤儿进程将被init进程(进程id为 1)接收，并由init进程处理这些子进程的善后工作。

僵尸进程: 子进程退出，而父进程并没有调用wait() 或waitpid()获取子进程的状态信息，对子进程进行善后工作，导致资源浪费，如子进程的进程描述符未回收。

孤儿进程和僵尸进程的区别：孤儿进程是父进程已退出，而子进程未退出；僵尸进程是父进程未退出，而子进程已退出。

守护进程是脱离于终端并且在后台运行的进程

### 5.9 虚拟内存 与 内存管理机制：

虚拟内存

### 5.10 缺页中断 ：

malloc及mmap等内存分配函数，在申请内存时，分配的只是逻辑地址空间，并没有分配实际物理地址，当访问这些地址时，会触发 缺页异常

缺页中断： 在 请求分页系统中， 可以通过查询 页表 中的状态来确定所访问的页面是否存在于内存中；当要访问的页面不在内存中时，会产生 缺页中断， 此时会按照一定的调度算法 从外存中将所却的页面调入内存。

缺页中断也是一种中断，过程如下：

* + - 1. 保护CPU现场(保存程序计数器，程序状态寄存器和栈数据等信息)
      2. 分析中断原因
      3. 转入 缺页中断处理程序进程处理
      4. 恢复CPU现场，继续执行

缺页异常因为要访问的页面不在内存中，是硬件所产生的一种特殊中断，与一般中断有如下区别：

1. 是在指令执行期间产生和处理缺页中断

2. 一条指令期间，可能产生多次缺页中断

### 5.9 mysql 引擎

1）InnoDB

InnoDB是一个事务型的存储引擎，有行级锁定和外键约束。mysql运行时InnoDB会在内存中建立缓冲池，用于缓冲数据和索引。但是它没有保存表的行数。

**适用场景：**经常更新的表，适合处理多重并发的更新请求。

1. MyISAM

是mysql默认的引擎，但是它没有提供对数据库事务的支持，也不支持行级锁定和外键，因此当insert或者update会锁定整个表。

## Linux相关

### 6.1 Linux系统卡顿，请问怎么排查：

1. CPU的使用情况： 使用top 命令产看，CPU使用后情况。
2. 内存使用 ： 可以用 free –g 来查看内存使用清理，若 free 栏为0或接近0，表示内存基本被吃完了，因此就需要释放内存。
3. 磁盘使用：可以用 df –h 来查看 磁盘使用情况，若发现磁盘使用率很高，就需要释放磁盘空间，删除一些不必要的文件。 可以使用 du 相关命令查看目录和文件的磁盘占用情况 比如 dh –sh \* 查看党情目录和文件使用情况
4. 磁盘I/O使用 :可以使用 iostax-x l查看磁盘I/O情况。

### 6.2 Linux 开机启动过程？

1. 主机加电自检，加载 BIOS 硬件信息。
2. 读取 MBR 的引导文件(GRUB、LILO)。
3. 引导 Linux 内核。
4. 运行第一个进程 init (进程号永远为 1 )。
5. 进入相应的运行级别。
6. 运行终端，输入用户名和密码