#后端开发面试知识点大纲： ##语言类（C++）： ###关键字作用解释： volatile作用

Volatile关键词的第一个特性：易变性。所谓的易变性，在汇编层面反映出来，就是两条语句，下一条语句不会直接使用上一条语句对应的volatile变量的寄存器内容，而是重新从内存中读取。

Volatile关键词的第二个特性：“不可优化”特性。volatile告诉编译器，不要对我这个变量进行各种激进的优化，甚至将变量直接消除，保证程序员写在代码中的指令，一定会被执行。

Volatile关键词的第三个特性：”顺序性”，能够保证Volatile变量间的顺序性，编译器不会进行乱序优化。

C/C++ Volatile变量，与非Volatile变量之间的操作，是可能被编译器交换顺序的。C/C++ Volatile变量间的操作，是不会被编译器交换顺序的。哪怕将所有的变量全部都声明为volatile，哪怕杜绝了编译器的乱序优化，但是针对生成的汇编代码，CPU有可能仍旧会乱序执行指令，导致程序依赖的逻辑出错，volatile对此无能为力

针对这个多线程的应用，真正正确的做法，是构建一个happens-before语义。

[C/C++ Volatile关键词深度剖析](http://hedengcheng.com/?p=725)

static

控制变量的存储方式和可见性。

(1)修饰局部变量

一般情况下，对于局部变量是存放在栈区的，并且局部变量的生命周期在该语句块执行结束时便结束了。但是如果用static进行修饰的话，该变量便存放在静态数据区，其生命周期一直持续到整个程序执行结束。但是在这里要注意的是，虽然用static对局部变量进行修饰过后，其生命周期以及存储空间发生了变化，但是其作用域并没有改变，其仍然是一个局部变量，作用域仅限于该语句块。

(2)修饰全局变量

对于一个全局变量，它既可以在本源文件中被访问到，也可以在同一个工程的其它源文件中被访问(只需用extern进行声明即可)。用static对全局变量进行修饰改变了其作用域的范围，由原来的整个工程可见变为本源文件可见。

(3)修饰函数

用static修饰函数的话，情况与修饰全局变量大同小异，就是改变了函数的作用域。

(4)C++中的static

如果在C++中对类中的某个函数用static进行修饰，则表示该函数属于一个类而不是属于此类的任何特定对象；如果对类中的某个变量进行static修饰，表示该变量为类以及其所有的对象所有。它们在存储空间中都只存在一个副本。可以通过类和对象去调用。

const的含义及实现机制

const名叫常量限定符，用来限定特定变量，以通知编译器该变量是不可修改的。习惯性的使用const，可以避免在函数中对某些不应修改的变量造成可能的改动。

(1)const修饰基本数据类型

1.const修饰一般常量及数组

基本数据类型，修饰符const可以用在类型说明符前，也可以用在类型说明符后，其结果是一样的。在使用这些常量的时候，只要不改变这些常量的值便好。

2.const修饰指针变量\*及引用变量&

如果const位于星号\*的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量；

如果const位于星号的右侧，const就是修饰指针本身，即指针本身是常量。

(2)const应用到函数中,

1.作为参数的const修饰符

调用函数的时候，用相应的变量初始化const常量，则在函数体中，按照const所修饰的部分进行常量化,保护了原对象的属性。

[注意]：参数const通常用于参数为指针或引用的情况;

2.作为函数返回值的const修饰符

声明了返回值后，const按照"修饰原则"进行修饰，起到相应的保护作用。

(3)const在类中的用法

不能在类声明中初始化const数据成员。正确的使用const实现方法为：const数据成员的初始化只能在类构造函数的初始化表中进行

类中的成员函数：A fun4()const; 其意义上是不能修改所在类的的任何变量。

(4)const修饰类对象，定义常量对象

常量对象只能调用常量函数，别的成员函数都不能调用。

http://www.cnblogs.com/wintergrass/archive/2011/04/15/2015020.html

extern

在C语言中，修饰符extern用在变量或者函数的声明前，用来说明“此变量/函数是在别处定义的，要在此处引用”。

注意extern声明的位置对其作用域也有关系，如果是在main函数中进行声明的，则只能在main函数中调用，在其它函数中不能调用。其实要调用其它文件中的函数和变量，只需把该文件用#include包含进来即可，为啥要用extern？因为用extern会加速程序的编译过程，这样能节省时间。

在C++中extern还有另外一种作用，用于指示C或者C＋＋函数的调用规范。比如在C＋＋中调用C库函数，就需要在C＋＋程序中用extern “C”声明要引用的函数。这是给链接器用的，告诉链接器在链接的时候用C函数规范来链接。主要原因是C＋＋和C程序编译完成后在目标代码中命名规则不同，用此来解决名字匹配的问题。

宏定义和展开、内联函数区别，

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

宏是预编译器的输入，然后宏展开之后的结果会送去编译器做语法分析。宏与函数等处于不同的级别，操作不同的实体。宏操作的是 token, 可以进行 token的替换和连接等操作，在语法分析之前起作用。而函数是语言中的概念，会在语法树中创建对应的实体，内联只是函数的一个属性。

对于问题：有了函数要它们何用？答案是：一：函数并不能完全替代宏，有些宏可以在当前作用域生成一些变量，函数做不到。二：内联函数只是函数的一种，内联是给编译器的提示，告诉它最好把这个函数在被调用处展开，省掉一个函数调用的开销（压栈，跳转，返回）

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

内联函数必须是和函数体申明在一起，才有效。

[宏定义和内联函数区别](http://www.cnblogs.com/chengxuyuancc/archive/2013/04/04/2999844.html)

###库函数实现： malloc,strcpy,strcmp的实现，常用库函数实现，哪些库函数属于高危函数

###STL原理及实现： STL各类型容器实现，STL共有六大组件

STL提供六大组件，彼此可以组合套用：

1、容器（Containers）：各种数据结构，如：序列式容器vector、list、deque、关联式容器set、map、multiset、multimap。用来存放数据。从实现的角度来看，STL容器是一种class template。

2、算法（algorithms）：各种常用算法，如：sort、search、copy、erase。从实现的角度来看，STL算法是一种 function template。注意一个问题：任何的一个STL算法，都需要获得由一对迭代器所标示的区间，用来表示操作范围。这一对迭代器所标示的区间都是前闭后开区间，例如[first, last)

3、迭代器（iterators）：容器与算法之间的胶合剂，是所谓的“泛型指针”。共有五种类型，以及其他衍生变化。从实现的角度来看，迭代器是一种将 operator\*、operator->、operator++、operator- - 等指针相关操作进行重载的class template。所有STL容器都有自己专属的迭代器，只有容器本身才知道如何遍历自己的元素。原生指针(native pointer)也是一种迭代器。

4、仿函数（functors）：行为类似函数，可作为算法的某种策略（policy）。从实现的角度来看，仿函数是一种重载了operator（）的class或class template。一般的函数指针也可视为狭义的仿函数。

5、配接器（adapters）：一种用来修饰容器、仿函数、迭代器接口的东西。例如：STL提供的queue 和 stack，虽然看似容器，但其实只能算是一种容器配接器，因为它们的底部完全借助deque，所有操作都由底层的deque供应。改变 functors接口者，称为function adapter；改变 container 接口者，称为container adapter；改变iterator接口者，称为iterator adapter。

6、配置器（allocators）：负责空间配置与管理。从实现的角度来看，配置器是一个实现了动态空间配置、空间管理、空间释放的class template。

这六大组件的交互关系：container（容器） 通过 allocator（配置器） 取得数据储存空间，algorithm（算法）通过 iterator（迭代器）存取 container（容器） 内容，functor（仿函数） 可以协助 algorithm（算法） 完成不同的策略变化，adapter（配接器） 可以修饰或套接 functor（仿函数）

序列式容器：

vector-数组，元素不够时再重新分配内存，拷贝原来数组的元素到新分配的数组中。

list－单链表。

deque-分配中央控制器map(并非map容器)，map记录着一系列的固定长度的数组的地址.记住这个map仅仅保存的是数组的地址,真正的数据在数组中存放着.deque先从map中央的位置(因为双向队列，前后都可以插入元素)找到一个数组地址，向该数组中放入数据，数组不够时继续在map中找空闲的数组来存数据。当map也不够时重新分配内存当作新的map,把原来map中的内容copy的新map中。所以使用deque的复杂度要大于vector，尽量使用vector。

stack-基于deque。

queue-基于deque。

heap-完全二叉树，使用最大堆排序，以数组(vector)的形式存放。

priority\_queue-基于heap。

slist-双向链表。

关联式容器：

set,map,multiset,multimap-基于红黑树(RB-tree)，一种加上了额外平衡条件的二叉搜索树。

hash table-散列表。将待存数据的key经过映射函数变成一个数组(一般是vector)的索引，例如：数据的key%数组的大小＝数组的索引(一般文本通过算法也可以转换为数字)，然后将数据当作此索引的数组元素。有些数据的key经过算法的转换可能是同一个数组的索引值(碰撞问题，可以用线性探测，二次探测来解决)，STL是用开链的方法来解决的，每一个数组的元素维护一个list，他把相同索引值的数据存入一个list，这样当list比较短时执行删除，插入，搜索等算法比较快。

hash\_map,hash\_set,hash\_multiset,hash\_multimap-基于hashtable。

[STL六大组件] (<http://blog.csdn.net/chenguolinblog/article/details/30336805>)  
什么是“标准非STL容器”？

list和vector有什么区别？

vector拥有一段连续的内存空间，因此支持随机存取，如果需要高效的随即存取，而不在乎插入和删除的效率，使用vector。

list拥有一段不连续的内存空间，因此不支持随机存取，如果需要大量的插入和删除，而不关心随即存取，则应使用list。

###虚函数： 虚函数的作用和实现原理，什么是虚函数,有什么作用?

C++的多态分为静态多态（编译时多态）和动态多态（运行时多态）两大类。静态多态通过重载、模板来实现；动态多态就是通过本文的主角虚函数来体现的。

虚函数实现原理:包括虚函数表、虚函数指针等

虚函数的作用说白了就是：当调用一个虚函数时，被执行的代码必须和调用函数的对象的动态类型相一致。编译器需要做的就是如何高效的实现提供这种特性。不同编译器实现细节也不相同。大多数编译器通过vtbl（virtual table）和vptr（virtual table pointer）来实现的。 当一个类声明了虚函数或者继承了虚函数，这个类就会有自己的vtbl。vtbl实际上就是一个函数指针数组，有的编译器用的是链表，不过方法都是差不多。vtbl数组中的每一个元素对应一个函数指针指向该类的一个虚函数，同时该类的每一个对象都会包含一个vptr，vptr指向该vtbl的地址。

结论：

每个声明了虚函数或者继承了虚函数的类，都会有一个自己的vtbl

同时该类的每个对象都会包含一个vptr去指向该vtbl

虚函数按照其声明顺序放于vtbl表中, vtbl数组中的每一个元素对应一个函数指针指向该类的虚函数

如果子类覆盖了父类的虚函数，将被放到了虚表中原来父类虚函数的位置

在多继承的情况下，每个父类都有自己的虚表。子类的成员函数被放到了第一个父类的表中

衍生问题:为什么 C++里访问虚函数比访问普通函数慢?

单继承时性能差不多，多继承的时候会慢

调用性能方面

从前面虚函数的调用过程可知。当调用虚函数时过程如下（引自More Effective C++）:

通过对象的 vptr 找到类的 vtbl。这是一个简单的操作,因为编译器知道在对象内 哪里能找到 vptr(毕竟是由编译器放置的它们)。因此这个代价只是一个偏移调整(以得到 vptr)和一个指针的间接寻址(以得到 vtbl)。

找到对应 vtbl 内的指向被调用函数的指针。这也是很简单的, 因为编译器为每个虚函数在 vtbl 内分配了一个唯一的索引。这步的代价只是在 vtbl 数组内 的一个偏移。

调用第二步找到的的指针所指向的函数。

在单继承的情况下，调用虚函数所需的代价基本上和非虚函数效率一样，在大多数计算机上它多执行了很少的一些指令，所以有很多人一概而论说虚函数性能不行是不太科学的。在多继承的情况下，由于会根据多个父类生成多个vptr，在对象里为寻找 vptr 而进行的偏移量计算会变得复杂一些，但这些并不是虚函数的性能瓶颈。 虚函数运行时所需的代价主要是虚函数不能是内联函。这也是非常好理解的，是因为内联函数是指在编译期间用被调用的函数体本身来代替函数调用的指令，但是虚函数的“虚”是指“直到运行时才能知道要调用的是哪一个函数。”但虚函数的运行时多态特性就是要在运行时才知道具体调用哪个虚函数，所以没法在编译时进行内联函数展开。当然如果通过对象直接调用虚函数它是可以被内联，但是大多数虚函数是通过对象的指针或引用被调用的，这种调用不能被内联。 因为这种调用是标准的调用方式，所以虚函数实际上不能被内联。

占用空间方面

在上面的虚函数实现原理部分，可以看到为了实现运行时多态机制，编译器会给每一个包含虚函数或继承了虚函数的类自动建立一个虚函数表，所以虚函数的一个代价就是会增加类的体积。在虚函数接口较少的类中这个代价并不明显，虚函数表vtbl的体积相当于几个函数指针的体积，如果你有大量的类或者在每个类中有大量的虚函数,你会发现 vtbl 会占用大量的地址空间。但这并不是最主要的代价，主要的代价是发生在类的继承过程中，在上面的分析中，可以看到，当子类继承父类的虚函数时，子类会有自己的vtbl，如果子类只覆盖父类的一两个虚函数接口，子类vtbl的其余部分内容会与父类重复。这在如果存在大量的子类继承，且重写父类的虚函数接口只占总数的一小部分的情况下，会造成大量地址空间浪费。在一些GUI库上这种大量子类继承自同一父类且只覆盖其中一两个虚函数的情况是经常有的，这样就导致UI库的占用内存明显变大。 由于虚函数指针vptr的存在，虚函数也会增加该类的每个对象的体积。在单继承或没有继承的情况下，类的每个对象会多一个vptr指针的体积，也就是4个字节；在多继承的情况下，类的每个对象会多N个（N＝包含虚函数的父类个数）vptr的体积，也就是4N个字节。当一个类的对象体积较大时，这个代价不是很明显，但当一个类的对象很轻量的时候，如成员变量只有4个字节，那么再加上4（或4N）个字节的vptr，对象的体积相当于翻了1（或N）倍，这个代价是非常大的。

[C++虚函数浅析](http://glgjing.github.io/blog/2015/01/03/c-plus-plus-xu-han-shu-qian-xi/)

纯虚函数，为什么需要纯虚函数？

纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加“=0”

virtual void funtion1()=0

原因：

1、为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。

2、在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。

为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual ReturnType Function()= 0;），则编译器要求在派生类中必须予以重写以实现多态性。同时含有纯虚拟函数的类称为抽象类，它不能生成对象。这样就很好地解决了上述两个问题。声明了纯虚函数的类是一个抽象类。所以，用户不能创建类的实例，只能创建它的派生类的实例。

定义纯虚函数的目的在于，使派生类仅仅只是继承函数的接口。

纯虚函数的意义，让所有的类对象（主要是派生类对象）都可以执行纯虚函数的动作，但类无法为纯虚函数提供一个合理的缺省实现。所以类纯虚函数的声明就是在告诉子类的设计者，“你必须提供一个纯虚函数的实现，但我不知道你会怎样实现它”。

[虚函数和纯虚函数的区别](http://blog.csdn.net/hackbuteer1/article/details/7558868)

为什么需要虚析构函数,什么时候不需要?父类的析构函数为什么要定义为虚函数

一般情况下类的析构函数里面都是释放内存资源，而析构函数不被调用的话就会造成内存泄漏。这样做是为了当用一个基类的指针删除一个派生类的对象时，派生类的析构函数会被调用。

当然，并不是要把所有类的析构函数都写成虚函数。因为当类里面有虚函数的时候，编译器会给类添加一个虚函数表，里面来存放虚函数指针，这样就会增加类的存储空间。所以，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。

内联函数、构造函数、静态成员函数可以是虚函数吗?

inline, static, constructor三种函数都不能带有virtual关键字。

inline是编译时展开，必须有实体；

static属于class自己的，也必须有实体；

virtual函数基于vtable（内存空间），constructor函数如果是virtual的，调用时也需要根据vtable寻找，但是constructor是virtual的情况下是找不到的，因为constructor自己本身都不存在了，创建不到class的实例，没有实例，class的成员（除了public static/protected static for friend class/functions，其余无论是否virtual）都不能被访问了。

虚函数实际上不能被内联:虚函数运行时所需的代价主要是虚函数不能是内联函。这也是非常好理解的，是因为内联函数是指在编译期间用被调用的函数体本身来代替函数调用的指令，但是虚函数的“虚”是指“直到运行时才能知道要调用的是哪一个函数。”但虚函数的运行时多态特性就是要在运行时才知道具体调用哪个虚函数，所以没法在编译时进行内联函数展开。当然如果通过对象直接调用虚函数它是可以被内联，但是大多数虚函数是通过对象的指针或引用被调用的，这种调用不能被内联。 因为这种调用是标准的调用方式，所以虚函数实际上不能被内联。

构造函数不能是虚函数。而且，在构造函数中调用虚函数，实际执行的是父类的对应函数，因为自己还没有构造好, 多态是被disable的。

静态的对象是属于整个类的，不对某一个对象而言，同时其函数的指针存放也不同于一般的成员函数，其无法成为一个对象的虚函数的指针以实现由此带来的动态机制。

构造函数中可以调用虚函数吗?

最后，总结一下关于虚函数的一些常见问题：

1) 虚函数是动态绑定的，也就是说，使用虚函数的指针和引用能够正确找到实际类的对应函数，而不是执行定义类的函数。这是虚函数的基本功能，就不再解释了。

2) 构造函数不能是虚函数。而且，在构造函数中调用虚函数，实际执行的是父类的对应函数，因为自己还没有构造好, 多态是被disable的。

3) 析构函数可以是虚函数，而且，在一个复杂类结构中，这往往是必须的。

4) 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。

5) 纯虚函数通常没有定义体，但也完全可以拥有。

6) 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。

7) 非纯的虚函数必须有定义体，不然是一个错误。

8) 派生类的override虚函数定义必须和父类完全一致。除了一个特例，如果父类中返回值是一个指针或引用，子类override时可以返回这个指针（或引用）的派生。例如，在上面的例子中，在Base中定义了 virtual Base\* clone(); 在Derived中可以定义为 virtual Derived\* clone()。可以看到，这种放松对于Clone模式是非常有用的。

[虚析构函数(√)、纯虚析构函数(√)、虚构造函数(X)](http://www.cnblogs.com/chio/archive/2007/09/10/888260.html)

为什么需要虚继承?虚继承实现原理解析，

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,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。

虚继承的特点是，在任何派生类中的virtual基类总用同一个（共享）对象表示，

[C++虚拟继承](http://blog.csdn.net/hyg0811/article/details/11951855) ###设计模式：

C++单例模式写法:

静态化并不是单例 (Singleton) 模式:

第一, 静态成员变量初始化顺序不依赖构造函数, 得看编译器心情的, 没法保证初始化顺序 (极端情况: 有 a b 两个成员对象, b 需要把 a 作为初始化参数传入, 你的类就 必须 得要有构造函数, 并确保初始化顺序).

第二, 最严重的问题, 失去了面对对象的重要特性 -- "多态", 静态成员方法不可能是 virtual 的. Log 类的子类没法享受 "多态" 带来的便利.

class Log {

public:

static void Write(char const \*logline);

static bool SaveTo(char const \*filename);

private:

static std::list<std::string> m\_data;

};

In log.cpp we need to add

std::list<std::string> Log::m\_data;

饿汉模式:

饿汉模式 是指单例实例在程序运行时被立即执行初始化:

class Log {

public:

static Log\* Instance() {

return &m\_pInstance;

}

virtual void Write(char const \*logline);

virtual bool SaveTo(char const \*filename);

private:

Log(); // ctor is hidden

Log(Log const&); // copy ctor is hidden

static Log m\_pInstance;

static std::list<std::string> m\_data;

};

// in log.cpp we have to add

Log Log::m\_pInstance;

这种模式的问题也很明显, 类现在是多态的, 但静态成员变量初始化顺序还是没保证.

懒汉模式 (堆栈-粗糙版)

单例实例只在第一次被使用时进行初始化:

class Log {

public:

static Log\* Instance() {

if (!m\_pInstance)

m\_pInstance = new Log;

return m\_pInstance;

}

virtual void Write(char const \*logline);

virtual bool SaveTo(char const \*filename);

private:

Log(); // ctor is hidden

Log(Log const&); // copy ctor is hidden

static Log\* m\_pInstance;

static std::list<std::string> m\_data;

};

// in log.cpp we have to add

Log\* Log::m\_pInstance = NULL;

Instance() 只在第一次被调用时为 m\_pInstance 分配内存并初始化. 嗯, 看上去所有的问题都解决了, 初始化顺序有保证, 多态也没问题.

程序退出时, 析构函数没被执行. 这在某些设计不可靠的系统上会导致资源泄漏, 比如文件句柄, socket 连接, 内存等等

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

懒汉模式 (局部静态变量-最佳版)

它也被称为 Meyers Singleton [Meyers]:

class Log {

public:

static Log& Instance() {

static Log theLog;

return theLog;

}

virtual void Write(char const \*logline);

virtual bool SaveTo(char const \*filename);

private:

Log(); // ctor is hidden

Log(Log const&); // copy ctor is hidden

Log& operator=(Log const&); // assign op is hidden

static std::list<std::string> m\_data;

};

在 Instance() 函数内定义局部静态变量的好处是, theLog `` 的构造函数只会在第一次调用 ``Instance() 时被初始化, 达到了和 "堆栈版" 相同的动态初始化效果, 保证了成员变量和 Singleton 本身的初始化顺序.

它还有一个潜在的安全措施, Instance() 返回的是对局部静态变量的引用, 如果返回的是指针, Instance() 的调用者很可能会误认为他要检查指针的有效性, 并负责销毁. 构造函数和拷贝构造函数也私有化了, 这样类的使用者不能自行实例化.

另外, 多个不同的 Singleton 实例的析构顺序与构造顺序相反.

[C++ Singleton (单例) 模式最优实现](http://blog.yangyubo.com/2009/06/04/best-cpp-singleton-pattern/)

用C++设计一个不能被继承的类。

构造函数或析构函数为私有函数，所以该类是无法被继承的，

如何定义一个只能在堆上定义对象的类?栈上呢

只能在堆内存上实例化的类：将析构函数定义为private，在栈上不能自动调用析构函数，只能手动调用。也可以将构造函数定义为private，但这样需要手动写一个函数实现对象的构造。

只能在栈内存上实例化的类：将函数operator new和operator delete定义为private，这样使用new操作符创建对象时候，无法调用operator new，delete销毁对象也无法调用operator delete。

[设计一个只能在堆上或栈上实例化的类](http://www.cnblogs.com/luxiaoxun/archive/2012/08/03/2621827.html)

满足上述3个条件

[C++中的单例模式](http://www.cnblogs.com/xiehongfeng100/p/4781013.html)

多重类构造和析构的顺序

先调用基类的构造函数，在调用派生类的构造函数

先构造的后析构，后构造的先析构

###内存分配：

内存分配方式有三种：

（1）从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static变量。

（2）在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

（3）从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或delete释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

c++运行时各类型内存分配（堆，栈，静态区，数据段，BSS，ELF），BSS段， sizeof一个类求大小（字节对齐原则）、

C++四种强制类型转换，

int char float,long long long类型长度 ###指针：

防止指针的越界使用，

必须让指针指向一个有效的内存地址,

1 防止数组越界

2 防止向一块内存中拷贝过多的内容

3 防止使用空指针

4 防止改变const修改的指针

5 防止改变指向静态存储区的内容

6 防止两次释放一个指针

7 防止使用野指针.

什么是指针退化及防止、

如果用一个数组作为函数入参

比如

void fun(char a[100])

{

cout<<SIZEOF(A)<

}

指针的移动问题，

指针P ++具体移动的字节数等于指针指向的变量类型大小.

Const,volatile修饰指针的含义，

堆和栈上的指针，

指针所指向的这块内存是在哪里分配的,在堆上称为堆上的指针,在栈上为栈上的指针.

在堆上的指针,可以保存在全局数据结构中,供不同函数使用访问同一块内存.

在栈上的指针,在函数退出后,该内存即不可访问.

指针的释放及内存泄露原因，

指针作为函数的参数，函数指针，

指针和引用及地址的区别，数组名，

指针与地址的区别?

区别:

1指针意味着已经有一个指针变量存在,他的值是一个地址,指针变量本身也存放在一个长度为四个字节的地址当中,而地址概念本身并不代表有任何变量存在.

2 指针的值,如果没有限制,通常是可以变化的,也可以指向另外一个地址.

地址表示内存空间的一个位置点,他是用来赋给指针的,地址本身是没有大小概念,指针指向变量的大小,取决于地址后面存放的变量类型.

指针与数组名的关系?

其值都是一个地址,但前者是可以移动的,后者是不可变的.

指针和引用的区别（一般都会问到）

相同点：1. 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。

区别：1. 指针是一个实体，而引用仅是个别名；

2. 引用使用时无需解引用(\*)，指针需要解引用；

3. 引用只能在定义时被初始化一次，之后不可变；指针可变；

4. 引用没有 const，指针有 const；

5. 引用不能为空，指针可以为空；

6. “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；

7. 指针和引用的自增(++)运算意义不一样；

8.从内存分配上看：程序为指针变量分配内存区域，而引用不需要分配内存区域。

迭代器与普通指针有什么区别

智能指针的原理,

智能指针：实际指行为类似于指针的类对象 ，它的一种通用实现方法是采用引用计数的方法。

1.智能指针将一个计数器与类指向的对象相关联，引用计数跟踪共有多少个类对象共享同一指针。

2.每次创建类的新对象时，初始化指针并将引用计数置为1；

3.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；

4.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；这是因为左侧的指针指向了右侧指针所指向的对象，因此右指针所指向的对象的引用计数+1；

5.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）。

6.实现智能指针有两种经典策略：一是引入辅助类，二是使用句柄类。这里主要讲一下引入辅助类的方法

其他：override和overload的区别，

override（重写）

1、方法名、参数、返回值相同。

2、子类方法不能缩小父类方法的访问权限。

3、子类方法不能抛出比父类方法更多的异常(但子类方法可以不抛出异常)。

4、存在于父类和子类之间。

5、方法被定义为final不能被重写。

overload（重载）

1、参数类型、个数、顺序至少有一个不相同。

2、不能重载只有返回值不同的方法名。

3、存在于父类和子类、同类中。

Overload是重载的意思，Override是覆盖的意思，也就是重写。

重载Overload表示同一个类中可以有多个名称相同的方法，但这些方法的参数列表各不相同（即参数个数或类型不同）。

重写Override表示子类中的方法可以与父类中的某个方法的名称和参数完全相同，通过子类创建的实例对象调用这个方法时，将调用子类中的定义方法，这相当于把父类中定义的那个完全相同的方法给覆盖了，这也是面向对象编程的多态性的一种表现。

子类覆盖父类的方法时，只能比父类抛出更少的异常，或者是抛出父类抛出的异常的子异常，因为子类可以解决父类的一些问题，不能比父类有更多的问题。子类方法的访问权限只能比父类的更大，不能更小。如果父类的方法是private类型，那么，子类则不存在覆盖的限制，相当于子类中增加了一个全新的方法。

写string类的构造，析构，拷贝函数

String 类的原型如下

class String

{

public:

String(const char \*str=NULL); //构造函数

String(const String &other); //拷贝构造函数

~String(void); //析构函数

String& operator=(const String &other); //等号操作符重载

ShowString();

private:

char \*m\_data; //指针

};

String::~String()

{

delete [] m\_data; //析构函数，释放地址空间

}

String::String(const char \*str)

{

if (str==NULL)//当初始化串不存在的时候，为m\_data申请一个空间存放'\0'；

{

m\_data=new char[1];

\*m\_data='\0';

}

else//当初始化串存在的时候，为m\_data申请同样大小的空间存放该串；

{

int length=strlen(str);

m\_data=new char[length+1];

strcpy(m\_data,str);

}

}

String::String(const String &other)//拷贝构造函数，功能与构造函数类似。

{

int length=strlen(other.m\_data);

m\_data=new [length+1];

strcpy(m\_data,other.m\_data);

}

String& String::operator =(const String &other)

{

if (this==&other)//当地址相同时，直接返回；

return \*this;

delete [] m\_data;//当地址不相同时，删除原来申请的空间，重新开始构造；

int length=sizeof(other.m\_data);

m\_data=new [length+1];

strcpy(m\_data,other.m\_data);

return \*this;

}

String::ShowString()//由于m\_data是私有成员，对象只能通过public成员函数来访问；

{

cout<<this->m\_data<<endl;

}

main()

{

String AD;

char \* p="ABCDE";

String B(p);

AD.ShowString();

AD=B;

AD.ShowString();

}

1 指针的四要素

1指针变量,表示一个内存地址,通常为逻辑地址,与实际的物理地址还有一个映射关系.

2指针变量的长度,在WIN32下为四个字节,

3指针指向的变量

该内存地址空间下存放的变量,具体内容可能是各种类型的变量.

4 指针指向的变量的长度,以该内存地址空间开始的内存空间大小.

##数据结构算法：

链表、树、哈希表、有效避免hash结果值的碰撞

排序算法性能比较

##操作系统：

linux的内存管理机制，内存寻址方式，什么叫虚拟内存，内存调页算法，任务调度算法、

Linux虚拟内存的实现需要6种机制的支持：地址映射机制、内存分配回收机制、缓存和刷新机制、请求页机制、交换机制和内存共享机制

内存管理程序通过映射机制把用户程序的逻辑地址映射到物理地址。当用户程序运行时，如果发现程序中要用的虚地址没有对应的物理内存，就发出了请求页要求。如果有空闲的内存可供分配，就请求分配内存(于是用到了内存的分配和回收)，并把正在使用的物理页记录在缓存中(使用了缓存机制)。如果没有足够的内存可供分配，那么就调用交换机制；腾出一部分内存。另外，在地址映射中要通过TLB(翻译后援存储器)来寻找物理页；交换机制中也要用到交换缓存，并且把物理页内容交换到交换文件中，也要修改页表来映射文件地址。

进程和线程、进程间及线程通信方式、共享内存的使用实现原理

死锁必要条件及避免算法、

1、资源不能共享，只能由一个进程使用。

2、请求与保持（Hold andwait）：已经得到资源的进程可以再次申请新的资源。

3、不可剥夺（Nopre-emption）：已经分配的资源不能从相应的进程中被强制地剥夺。

4、循环等待：系统中若干进程组成环路，该环路中每个进程都在等待相邻进程正占用的资源

处理死锁的策略：1.忽略该问题。例如鸵鸟算法，该算法可以应用在极少发生死锁的的情况下。为什么叫鸵鸟算法呢，因为传说中鸵鸟看到危险就把头埋在地底下，可能鸵鸟觉得看不到危险也就没危险了吧。跟掩耳盗铃有点像。2.检测死锁并且恢复。3.仔细地对资源进行动态分配，以避免死锁。4.通过破除死锁四个必要条件之一，来防止死锁产生。）

动态链接和静态链接的区别、

动态链接是只建立一个引用的接口，而真正的代码和数据存放在另外的可执行模块中，在运行时再装入；而静态链接是把所有的代码和数据都复制到本模块中，运行时就不再需要库了。

c程序辨别系统是16位or32位,大端or小端字节序、

16or32

法一：int k=~0;

if((unsigned int)k >63356) cout<<"at least 32bits"<<endl;

else cout<<"16 bits"<<endl;

法二：//32为系统

int i=65536;

cout<<i<<endl;

int j=65535;

cout<<j<<endl;

大or小

1) Little-Endian就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

2) Big-Endian就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。

举一个例子，比如数字0x12 34 56 78在内存中的表示形式为：

1)大端模式：

低地址 -----------------> 高地址

0x12 | 0x34 | 0x56 | 0x78

2)小端模式：

低地址 ------------------> 高地址

0x78 | 0x56 | 0x34 | 0x12

32bit宽的数0x12345678在Little-endian模式以及Big-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 小端模式存放内容 大端模式存放内容

0x4000 0x78 0x12

0x4001 0x56 0x34

0x4002 0x34 0x56

0x4003 0x12 0x78

4)大端小端没有谁优谁劣，各自优势便是对方劣势：

小端模式 ：强制转换数据不需要调整字节内容，1、2、4字节的存储方式一样。

大端模式 ：符号位的判定固定为第一个字节，容易判断正负。

BOOL IsBigEndian()

{

int a = 0x1234;

char b = \*(char \*)&a; //通过将int强制类型转换成char单字节，通过判断起始存储位置。即等于 取b等于a的低地址部分

if( b == 0x12)

{

return TRUE;

}

return FALSE;

}

联合体union的存放顺序是所有成员都从低地址开始存放，利用该特性可以轻松地获得了CPU对内存采用Little-endian还是Big-endian模式读写：

BOOL IsBigEndian()

{

union NUM

{

int a;

char b;

}num;

num.a = 0x1234;

if( num.b == 0x12 )

{

return TRUE;

}

return FALSE;

}

一般操作系统都是小端，而通讯协议是大端的。

常见CPU的字节序

Big Endian : PowerPC、IBM、Sun

Little Endian : x86、DEC

ARM既可以工作在大端模式，也可以工作在小端模式。

常见的信号、系统如何将一个信号通知到进程、

信号机制是进程之间相互传递消息的一种方法，信号全称为软中断信号，也有人称作软中断。

进程之间可以互相通过系统调用kill发送软中断信号。

SIGHUP 1 A 终端挂起或者控制进程终止

SIGINT 2 A 键盘中断（如break键被按下）

SIGQUIT 3 C 键盘的退出键被按下

SIGILL 4 C 非法指令

SIGABRT 6 C 由abort(3)发出的退出指令

SIGFPE 8 C 浮点异常

SIGKILL 9 AEF Kill信号

SIGSEGV 11 C 无效的内存引用

SIGPIPE 13 A 管道破裂: 写一个没有读端口的管道

信号机制是异步的；当一个进程接收到一个信号时，它会立刻处理这个信号，而不会等待当前函数甚至当前一行代码结束运行。信号有几十种，分别代表着不同的意义。信号之间依靠它们的值来区分，但是通常在程序中使用信号的名字来表示一个信号。在Linux系统中，这些信号和以它们的名称命名的常量均定义在/usr/include/bits/signum.h文件中。（通常程序中不需要直接包含这个头文件，而应该包含<signal.h>。）

信号事件的发生有两个来源：硬件来源(比如我们按下了键盘或者其它硬件故障)；软件来源，最常用发送信号的系统函数是kill, raise, alarm和setitimer以及sigqueue函数，软件来源还包括一些非法运算等操作。

发送信号的主要函数有：kill()、raise()、 sigqueue()、alarm()、setitimer()以及abort()。

进程可以通过三种方式来响应一个信号：（1）忽略信号，即对信号不做任何处理，其中，有两个信号不能忽略：SIGKILL及SIGSTOP；（2）捕捉信号。定义信号处理函数，当信号发生时，执行相应的处理函数；（3）执行缺省操作，

linux系统的各类同步机制、linux系统的各类异步机制、

如何实现守护进程

守护进程最重要的特性是后台运行。

1. 在后台运行。

为避免挂起控制终端将Daemon放入后台执行。方法是在进程中调用fork使父进程终止，让Daemon在子进程中后台执行。

if(pid=fork())

exit(0); //是父进程，结束父进程，子进程继续

2. 脱离控制终端，登录会话和进程组

有必要先介绍一下Linux中的进程与控制终端，登录会话和进程组之间的关系：进程属于一个进程组，进程组号（GID）就是进程组长的进程号（PID）。登录会话可以包含多个进程组。这些进程组共享一个控制终端。这个控制终端通常是创建进程的登录终端。控制终端，登录会话和进程组通常是从父进程继承下来的。我们的目的就是要摆脱它们，使之不受它们的影响。方法是在第1点的基础上，调用setsid()使进程成为会话组长：

setsid();

说明：当进程是会话组长时setsid()调用失败。但第一点已经保证进程不是会话组长。setsid()调用成功后，进程成为新的会话组长和新的进程组长，并与原来的登录会话和进程组脱离。由于会话过程对控制终端的独占性，进程同时与控制终端脱离。

3. 禁止进程重新打开控制终端

现在，进程已经成为无终端的会话组长。但它可以重新申请打开一个控制终端。可以通过使进程不再成为会话组长来禁止进程重新打开控制终端：

if(pid=fork()) exit(0); //结束第一子进程，第二子进程继续（第二子进程不再是会话组长）

4. 关闭打开的文件描述符

进程从创建它的父进程那里继承了打开的文件描述符。如不关闭，将会浪费系统资源，造成进程所在的文件系统无法卸下以及引起无法预料的错误。按如下方法关闭它们：

for(i=0;i 关闭打开的文件描述符close(i);>

5. 改变当前工作目录

进程活动时，其工作目录所在的文件系统不能卸下。一般需要将工作目录改变到根目录。对于需要转储核心，写运行日志的进程将工作目录改变到特定目录如 /tmpchdir("/")

6. 重设文件创建掩模

进程从创建它的父进程那里继承了文件创建掩模。它可能修改守护进程所创建的文件的存取位。为防止这一点，将文件创建掩模清除：umask(0);

7. 处理SIGCHLD信号

处理SIGCHLD信号并不是必须的。但对于某些进程，特别是服务器进程往往在请求到来时生成子进程处理请求。如果父进程不等待子进程结束，子进程将成为僵尸进程（zombie）从而占用系统资源。如果父进程等待子进程结束，将增加父进程的负担，影响服务器进程的并发性能。在Linux下可以简单地将 SIGCHLD信号的操作设为SIG\_IGN。

signal(SIGCHLD,SIG\_IGN);

这样，内核在子进程结束时不会产生僵尸进程。这一点与BSD4不同，BSD4下必须显式等待子进程结束才能释放僵尸进程。

标准库函数和系统调用的区别，

1、系统调用

系统调用提供的函数如open, close, read, write, ioctl等，需包含头文件unistd.h。以write为例：其函数原型为 size\_t write(int fd, const void \*buf, size\_t nbytes)，其操作对象为文件描述符或文件句柄fd(file descriptor)，要想写一个文件，必须先以可写权限用open系统调用打开一个文件，获得所打开文件的fd，例如fd=open(/"/dev/video/", O\_RDWR)。fd是一个整型值，每新打开一个文件，所获得的fd为当前最大fd加1。Linux系统默认分配了3个文件描述符值：0－standard input，1－standard output，2－standard error。

系统调用通常用于底层文件访问（low-level file access），例如在驱动程序中对设备文件的直接访问。

系统调用是操作系统相关的，因此一般没有跨操作系统的可移植性。

系统调用发生在内核空间，因此如果在用户空间的一般应用程序中使用系统调用来进行文件操作，会有用户空间到内核空间切换的开销。事实上，即使在用户空间使用库函数来对文件进行操作，因为文件总是存在于存储介质上，因此不管是读写操作，都是对硬件（存储器）的操作，都必然会引起系统调用。也就是说，库函数对文件的操作实际上是通过系统调用来实现的。例如C库函数fwrite()就是通过write()系统调用来实现的。

这样的话，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。

2、库函数调用

标准C库函数提供的文件操作函数如fopen, fread, fwrite, fclose,fflush, fseek等，需包含头文件stdio.h。以fwrite为例，其函数原型为size\_t fwrite(const void \*buffer,size\_t size, size\_t item\_num, FILE \*pf)，其操作对象为文件指针FILE \*pf，要想写一个文件，必须先以可写权限用fopen函数打开一个文件，获得所打开文件的FILE结构指针pf，例如pf=fopen(/"~/proj/filename/",/"w/")。实际上，由于库函数对文件的操作最终是通过系统调用实现的，因此，每打开一个文件所获得的FILE结构指针都有一个内核空间的文件描述符fd与之对应。同样有相应的预定义的FILE指针：stdin－standard input，stdout－standard output，stderr－standard error。

库函数调用通常用于应用程序中对一般文件的访问。

库函数调用是系统无关的，因此可移植性好。

由于库函数调用是基于C库的，因此也就不可能用于内核空间的驱动程序中对设备的操作

fd和PCB，

32位系统一个进程最多有多少堆内存，

五种I/O 模式，

五种I/O 模式：

【1】 阻塞I/O (Linux下的I/O操作默认是阻塞I/O，即open和socket创建的I/O都是阻塞I/O)

【2】 非阻塞 I/O (可以通过fcntl或者open时使用O\_NONBLOCK参数，将fd设置为非阻塞的I/O)

【3】 I/O 多路复用 (I/O多路复用，通常需要非阻塞I/O配合使用)

【4】 信号驱动 I/O (SIGIO)

【5】 异步 I/O

Apache 模型（Process Per Connection，简称PPC），TPC（ThreadPer Connection）模型，以及 select 模型和 poll 模型，epoll模型

一般来说，程序进行输入操作有两步：

1．等待有数据可以读

2．将数据从系统内核中拷贝到程序的数据区。

对于sock编程来说:

第一步: 一般来说是等待数据从网络上传到本地。当数据包到达的时候，数据将会从网络层拷贝到内核的缓存中；

第二步: 是从内核中把数据拷贝到程序的数据区中。

阻塞I/O模式 //进程处于阻塞模式时，让出CPU，进入休眠状态

阻塞 I/O 模式是最普遍使用的 I/O 模式。是Linux系统下缺省的IO模式。

大部分程序使用的都是阻塞模式的 I/O 。

一个套接字建立后所处于的模式就是阻塞 I/O 模式。（因为Linux系统默认的IO模式是阻塞模式）

对于一个UDP 套接字来说，数据就绪的标志比较简单：

（1）已经收到了一整个数据报

（2）没有收到。

而 TCP 这个概念就比较复杂，需要附加一些其他的变量。

一个进程调用 recvfrom ，然后系统调用并不返回知道有数据报到达本地系统，然后系统将数据拷贝到进程的缓存中。（如果系统调用收到一个中断信号，则它的调用会被中断）

我们称这个进程在调用recvfrom一直到从recvfrom返回这段时间是阻塞的。当recvfrom正常返回时，我们的进程继续它的操作

非阻塞模式I/O //非阻塞模式的使用并不普遍，因为非阻塞模式会浪费大量的CPU资源。

当我们将一个套接字设置为非阻塞模式，我们相当于告诉了系统内核： “当我请求的I/O 操作不能够马上完成，你想让我的进程进行休眠等待的时候，不要这么做，请马上返回一个错误给我。”

我们开始对 recvfrom 的三次调用，因为系统还没有接收到网络数据，所以内核马上返回一个EWOULDBLOCK的错误。

第四次我们调用 recvfrom 函数，一个数据报已经到达了，内核将它拷贝到我们的应用程序的缓冲区中，然后 recvfrom 正常返回，我们就可以对接收到的数据进行处理了。

当一个应用程序使用了非阻塞模式的套接字，它需要使用一个循环来不听的测试是否一个文件描述符有数据可读(称做 polling(轮询))。应用程序不停的 polling 内核来检查是否 I/O操作已经就绪。这将是一个极浪费 CPU资源的操作。这种模式使用中不是很普遍。

例如:

对管道的操作，最好使用非阻塞方式！

I/O多路复用 //针对批量IP操作时，使用I/O多路复用，非常有好。

在使用 I/O 多路技术的时候，我们调用select()函数和 poll()函数或epoll函数(2.6内核开始支持)，在调用它们的时候阻塞，而不是我们来调用 recvfrom（或recv）的时候阻塞。

当我们调用 select函数阻塞的时候，select 函数等待数据报套接字进入读就绪状态。当select函数返回的时候，也就是套接字可以读取数据的时候。这时候我们就可以调用 recvfrom函数来将数据拷贝到我们的程序缓冲区中。

对于单个I/O操作，和阻塞模式相比较，select()和poll()或epoll并没有什么高级的地方。

而且，在阻塞模式下只需要调用一个函数：

读取或发送函数。

在使用了多路复用技术后，我们需要调用两个函数了：

先调用 select()函数或poll()函数，然后才能进行真正的读写。

多路复用的高级之处在于::

它能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，select()函数就可以返回。

IO 多路技术一般在下面这些情况中被使用：

1、当一个客户端需要同时处理多个文件描述符的输入输出操作的时候（一般来说是标准的输入输出和网络套接字)，I/O 多路复用技术将会有机会得到使用。

2、当程序需要同时进行多个套接字的操作的时候。

3、如果一个 TCP 服务器程序同时处理正在侦听网络连接的套接字和已经连接好的套接字。

4、如果一个服务器程序同时使用 TCP 和 UDP 协议。

5、如果一个服务器同时使用多种服务并且每种服务可能使用不同的协议（比如 inetd就是这样的）。

异步IO模式有::

1、信号驱动I/O模式

2、异步I/O模式

信号驱动I/O模式 //自己没有用过。

我们可以使用信号，让内核在文件描述符就绪的时候使用 SIGIO 信号来通知我们。我们将这种模式称为信号驱动 I/O 模式。

为了在一个套接字上使用信号驱动 I/O 操作，下面这三步是所必须的。

（1）一个和 SIGIO信号的处理函数必须设定。

（2）套接字的拥有者必须被设定。一般来说是使用 fcntl 函数的 F\_SETOWN 参数来

进行设定拥有者。

（3）套接字必须被允许使用异步 I/O。一般是通过调用 fcntl 函数的 F\_SETFL 命令，O\_ASYNC为参数来实现。

虽然设定套接字为异步 I/O 非常简单，但是使用起来困难的部分是怎样在程序中断定产生 SIGIO信号发送给套接字属主的时候，程序处在什么状态。

1．UDP 套接字的 SIGIO 信号 (比较简单)

在 UDP 协议上使用异步 I/O 非常简单．这个信号将会在这个时候产生：

1、套接字收到了一个数据报的数据包。

2、套接字发生了异步错误。

当我们在使用 UDP 套接字异步 I/O 的时候，我们使用 recvfrom()函数来读取数据报数据或是异步 I/O 错误信息。

2．TCP 套接字的 SIGIO 信号 (不会使用)

不幸的是，异步 I/O 几乎对 TCP 套接字而言没有什么作用。因为对于一个 TCP 套接字来说，SIGIO 信号发生的几率太高了，所以 SIGIO 信号并不能告诉我们究竟发生了什么事情。

在 TCP 连接中， SIGIO 信号将会在这个时候产生：

l 在一个监听某个端口的套接字上成功的建立了一个新连接。

l 一个断线的请求被成功的初始化。

l 一个断线的请求成功的结束。

l 套接字的某一个通道（发送通道或是接收通道）被关闭。

l 套接字接收到新数据。

l 套接字将数据发送出去。

l 发生了一个异步 I/O 的错误。

一个对信号驱动 I/O 比较实用的方面是NTP（网络时间协议 Network TimeProtocol）服务器，它使用 UDP。这个服务器的主循环用来接收从客户端发送过来的数据报数据包，然后再发送请求。对于这个服务器来说，记录下收到每一个数据包的具体时间是很重要的。

因为那将是返回给客户端的值，客户端要使用这个数据来计算数据报在网络上来回所花费的时间。图 6-8 表示了怎样建立这样的一个 UDP 服务器。

异步I/O模式 //比如写操作，只需用写，不一定写入磁盘(这就是异步I/O)的好处。异步IO的好处效率高。

当我们运行在异步 I/O 模式下时，我们如果想进行 I/O 操作，只需要告诉内核我们要进行 I/O 操作，然后内核会马上返回。具体的 I/O 和数据的拷贝全部由内核来完成，我们的程序可以继续向下执行。当内核完成所有的 I/O 操作和数据拷贝后，内核将通知我们的程序。

异步 I/O 和 信号驱动I/O的区别是：

1、信号驱动 I/O 模式下，内核在操作可以被操作的时候通知给我们的应用程序发送SIGIO 消息。

2、异步 I/O 模式下，内核在所有的操作都已经被内核操作结束之后才会通知我们的应用程序。

select，poll，epoll

. Epoll 是何方神圣？

Epoll 可是当前在 Linux 下开发大规模并发网络程序的热门人选， Epoll 在 Linux2.6 内核中正式引入，和 select 相似，其实都 I/O 多路复用技术而已，并没有什么神秘的。

其实在Linux 下设计并发网络程序，向来不缺少方法，比如典型的 Apache 模型（ Process Per Connection ，简称PPC ）， TPC （ ThreadPer Connection ）模型，以及 select 模型和 poll 模型，那为何还要再引入 Epoll 这个东东呢？那还是有得说说的 …

2. 常用模型的缺点

如果不摆出来其他模型的缺点，怎么能对比出 Epoll 的优点呢。

2.1 PPC/TPC 模型

这两种模型思想类似，就是让每一个到来的连接一边自己做事去，别再来烦我。只是 PPC 是为它开了一个进程，而 TPC 开了一个线程。可是别烦我是有代价的，它要时间和空间啊，连接多了之后，那么多的进程 / 线程切换，这开销就上来了；因此这类模型能接受的最大连接数都不会高，一般在几百个左右。

2.2 select 模型

1. 最大并发数限制，因为一个进程所打开的 FD （文件描述符）是有限制的，www.linuxidc.com 由FD\_SETSIZE 设置，默认值是 1024/2048 ，因此 Select 模型的最大并发数就被相应限制了。自己改改这个 FD\_SETSIZE ？想法虽好，可是先看看下面吧 …

2. 效率问题， select 每次调用都会线性扫描全部的 FD 集合，这样效率就会呈现线性下降，把 FD\_SETSIZE 改大的后果就是，大家都慢慢来，什么？都超时了？？！！

3. 内核 / 用户空间内存拷贝问题，如何让内核把 FD 消息通知给用户空间呢？在这个问题上 select 采取了内存拷贝方法。

2.3 poll 模型

基本上效率和select 是相同的，select 缺点的 2 和 3 它都没有改掉。

3. Epoll 的提升

把其他模型逐个批判了一下，再来看看 Epoll 的改进之处吧，其实把 select 的缺点反过来那就是 Epoll 的优点了。

3.1. Epoll 没有最大并发连接的限制，上限是最大可以打开文件的数目，这个数字一般远大于 2048, 一般来说这个数目和系统内存关系很大，具体数目可以 cat /proc/sys/fs/file-max 察看。

3.2. 效率提升， Epoll 最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中， Epoll 的效率就会远远高于 select 和 poll 。

3.3. 内存拷贝， Epoll 在这点上使用了“共享内存 ”，这个内存拷贝也省略了。

4. Epoll 为什么高效

Epoll 的高效和其数据结构的设计是密不可分的，这个下面就会提到。

首先回忆一下select 模型，当有I/O 事件到来时，select 通知应用程序有事件到了快去处理，而应用程序必须轮询所有的 FD 集合，测试每个 FD 是否有事件发生，并处理事件；代码像下面这样：

int res = select(maxfd+1, &readfds,NULL, NULL, 120);

if (res > 0)

{

for (int i = 0; i <MAX\_CONNECTION; i++)

{

if (FD\_ISSET(allConnection[i], &readfds))

{

handleEvent(allConnection[i]);

}

}

}

// if(res == 0) handle timeout, res < 0handle error

Epoll 不仅会告诉应用程序有I/0事件到来，还会告诉应用程序相关的信息，这些信息是应用程序填充的，因此根据这些信息应用程序就能直接定位到事件，而不必遍历整个FD 集合。

int res = epoll\_wait(epfd, events, 20,120);

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

{

handleEvent(events[n]);

}

5. Epoll 关键数据结构

前面提到Epoll 速度快和其数据结构密不可分，其关键数据结构就是：

struct epoll\_event {

\_\_uint32\_tevents; // Epoll events

epoll\_data\_tdata; // User data variable

};

typedef union epoll\_data {

void \*ptr;

int fd;

\_\_uint32\_t u32;

\_\_uint64\_t u64;

} epoll\_data\_t;

可见epoll\_data 是一个 union 结构体 , 借助于它应用程序可以保存很多类型的信息 :fd 、指针等等。有了它，应用程序就可以直接定位目标了。

socket服务端的实现，select和epoll的区别(必问)

select的本质是采用32个整数的32位，即32\*32= 1024来标识，fd值为1-1024。当fd的值超过1024限制时，就必须修改FD\_SETSIZE的大小。这个时候就可以标识32\*max值范围的fd。

对于单进程多线程，每个线程处理多个fd的情况，select是不适合的。

1.所有的线程均是从1-32\*max进行扫描，每个线程处理的均是一段fd值，这样做有点浪费

2.1024上限问题，一个处理多个用户的进程，fd值远远大于1024

所以这个时候应该采用poll，

poll传递的是数组头指针和该数组的长度，只要数组的长度不是很长，性能还是很不错的，因为poll一次在内核中申请4K（一个页的大小来存放fd），尽量控制在4K以内

epoll还是poll的一种优化，返回后不需要对所有的fd进行遍历，在内核中维持了fd的列表。select和poll是将这个内核列表维持在用户态，然后传递到内核中。但是只有在2.6的内核才支持。

epoll更适合于处理大量的fd ，且活跃fd不是很多的情况，毕竟fd较多还是一个串行的操作

epoll哪些触发模式，有啥区别？（必须非常详尽的解释水平触发和边缘触发的区别，以及边缘触发在编程中要做哪些更多的确认）

epoll可以同时支持水平触发和边缘触发（Edge Triggered，只告诉进程哪些文件描述符刚刚变为就绪状态，它只说一遍，如果我们没有采取行动，那么它将不会再次告知，这种方式称为边缘触发），理论上边缘触发的性能要更高一些，但是代码实现相当复杂。

epoll同样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，你只需要去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（mmap）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。

另一个本质的改进在于epoll采用基于事件的就绪通知方式。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。

惊群现象，

举一个很简单的例子，当你往一群鸽子中间扔一块食物，虽然最终只有一个鸽子抢到食物，但所有鸽子都会被惊动来争夺，没有抢到食物的鸽子只好回去继续睡觉，等待下一块食物到来。这样，每扔一块食物，都会惊动所有的鸽子，即为惊群。对于操作系统来说，多个进程/线程在等待同一资源是，也会产生类似的效果，其结果就是每当资源可用，所有的进程/线程都来竞争资源，造成的后果：

1）系统对用户进程/线程频繁的做无效的调度、上下文切换，系统系能大打折扣。

2）为了确保只有一个线程得到资源，用户必须对资源操作进行加锁保护，进一步加大了系统开销。

什么是惊群

最常见的例子就是对于socket描述符的accept操作，当多个用户进程/线程监听在同一个端口上时，由于实际只可能accept一次，因此就会产生惊群现象，当然前面已经说过了，这个问题是一个古老的问题，新的操作系统内核已经解决了这一问题。

linux内核解决惊群问题的方法

对于一些已知的惊群问题，内核开发者增加了一个“互斥等待”选项。一个互斥等待的行为与睡眠基本类似，主要的不同点在于：

1）当一个等待队列入口有 WQ\_FLAG\_EXCLUSEVE 标志置位, 它被添加到等待队列的尾部. 没有这个标志的入口项, 相反, 添加到开始.

2）当 wake\_up 被在一个等待队列上调用时, 它在唤醒第一个有 WQ\_FLAG\_EXCLUSIVE 标志的进程后停止。

也就是说，对于互斥等待的行为，比如如对一个listen后的socket描述符，多线程阻塞accept时，系统内核只会唤醒所有正在等待此时间的队列的第一个，队列中的其他人则继续等待下一次事件的发生，这样就避免的多个线程同时监听同一个socket描述符时的惊群问题。

块设备和字符设备有什么区别，

(1) 字符设备：提供连续的数据流，应用程序可以顺序读取，通常不支持随机存取。相反，此类设备支持按字节/字符来读写数据。举例来说，调制解调器是典型的字符设备。

(2) 块设备：应用程序可以随机访问设备数据，程序可自行确定读取数据的位置。硬盘是典型的块设备，应用程序可以寻址磁盘上的任何位置，并由此读取数据。此外，数据的读写只能以块(通常是512B)的倍数进行。与字符设备不同，块设备并不支持基于字符的寻址。

两种设备本身并没用严格的区分，主要是字符设备和块设备驱动程序提供的访问接口（file I/O API）是不一样的。本文主要就数据接口、访问接口和设备注册方法对两种设备进行比较。

用户态和内核态的区别

虽然用户态下和内核态下工作的程序有很多差别，但最重要的差别就在于特权级的不同，即权力的不同。运行在用户态下的程序不能直接访问操作系统内核数据结构和程序，

当我们在系统中执行一个程序时，大部分时间是运行在用户态下的，在其需要操作系统帮助完成某些它没有权力和能力完成的工作时就会切换到内核态，

linux文件系统：inode，inode存储了哪些东西，目录名，文件名存在哪里

inode包含文件的元信息，具体来说有以下内容：

　　\* 文件的字节数

　　\* 文件拥有者的User ID

　　\* 文件的Group ID

　　\* 文件的读、写、执行权限

　　\* 文件的时间戳，共有三个：ctime指inode上一次变动的时间，mtime指文件内容上一次变动的时间，atime指文件上一次打开的时间。

　　\* 链接数，即有多少文件名指向这个inode

　　\* 文件数据block的位置

　inode也会消耗硬盘空间，所以硬盘格式化的时候，操作系统自动将硬盘分成两个区域。一个是数据区，存放文件数据；另一个是inode区（inode table），存放inode所包含的信息。

每个inode节点的大小，一般是128字节或256字节。inode节点的总数，在格式化时就给定，一般是每1KB或每2KB就设置一个inode。假定在一块1GB的硬盘中，每个inode节点的大小为128字节，每1KB就设置一个inode，那么inode table的大小就会达到128MB，占整块硬盘的12.8%。

每个inode都有一个号码，操作系统用inode号码来识别不同的文件。

这里值得重复一遍，Unix/Linux系统内部不使用文件名，而使用inode号码来识别文件。对于系统来说，文件名只是inode号码便于识别的别称或者绰号。

表面上，用户通过文件名，打开文件。实际上，系统内部这个过程分成三步：首先，系统找到这个文件名对应的inode号码；其次，通过inode号码，获取inode信息；最后，根据inode信息，找到文件数据所在的block，读出数据。

一般情况下，文件名和inode号码是"一一对应"关系，每个inode号码对应一个文件名。但是，Unix/Linux系统允许，多个文件名指向同一个inode号码。

这意味着，可以用不同的文件名访问同样的内容；对文件内容进行修改，会影响到所有文件名；但是，删除一个文件名，不影响另一个文件名的访问。这种情况就被称为"硬链接"（hard link）。

ln命令可以创建硬链接：ln 源文件 目标文件

文件A和文件B的inode号码虽然不一样，但是文件A的内容是文件B的路径。读取文件A时，系统会自动将访问者导向文件B。因此，无论打开哪一个文件，最终读取的都是文件B。这时，文件A就称为文件B的"软链接"（soft link）或者"符号链接（symbolic link）。

这意味着，文件A依赖于文件B而存在，如果删除了文件B，打开文件A就会报错："No such file or directory"。这是软链接与硬链接最大的不同：文件A指向文件B的文件名，而不是文件B的inode号码，文件B的inode"链接数"不会因此发生变化。

ln -s命令可以创建软链接。：ln -s 源文文件或目录 目标文件或目录

http://www.ruanyifeng.com/blog/2011/12/inode.html

/proc存在哪里（存在内存上）

/proc 文件系统是一个虚拟文件系统，通过它可以使用一种新的方法在 Linux® 内核空间和用户空间之间进行通信。在 /proc 文件系统中，我们可以将对虚拟文件的读写作为与内核中实体进行通信的一种手段，但是与普通文件不同的是，这些虚拟文件的内容都是动态创建的

http://www.ibm.com/developerworks/cn/linux/l-proc.html

##网络： TCP和UDP区别、

key:TCP是一种面向连接的、可靠的、字节流服务

1.面向链接：TCP面向链接，面向连接意味着两个使用TCP的应用（通常是一个客户和一个服务器）在彼此交换数据之前必须通过三次握手先建立一个TCP连接。在一个TCP中仅有两方彼此通信，多播和广播不能用于TCP。UDP是不可靠的传输，传输前不需要建立链接，可以应用多播和广播实现一对多的通信。

2.可靠性：TCP提供端到端的流量控制，对收到的数据进行确认，采用超时重发，对失序的数据进行重新排序等机制保证数据通信的可靠性。而UDP是一种不可靠的服务，接收方可能不能收到发送方的数据报。

3.TCP是一种流模式的协议，UDP是一种数据报模式的协议。进程的每个输出操作都正好产生一个UDP数据报，并组装成一份待发送的IP数据报。TCP应用程序产生的全体数据与真正发送的单个IP数据报可能没有什么联系。TCP会有粘包和半包的现象。

4.效率上：速度上，一般TCP速度慢，传输过程中需要对数据进行确认，超时重发，还要对数据进行排序。UDP没有这些机制所以速度快。数据比例，TCP头至少20个字节，UDP头8个字节，相对效率高。组装效率上：TCP头至少20个字节，UDP头8个字节，系统组装上TCP相对慢。

5.用途上：用于TCP可靠性，http，ftp使用。而由于UDP速度快，视频，在线游戏多用UDP，保证实时性

对于第三点的理解。TCP可能发送100个“包”，而接收到50个“包”，不是丢“包”了，而是每次接受的“包”都比发送的多，其实TCP并没有包的概念。例如，每次发10个字节，可能读得时候一次读了20个字节。TCP是一种流模式的协议，在接收到的缓存中按照发送的包得顺序自动按照顺序拼接好，因为数据基本来自同一个主机，而且是按照顺序发送过来的，TCP的缓存中存放的就是，连续的数据。感觉好像是多封装了一步比UDP。而UDP因为可能两个不同的主机，给同一个主机发送，（一个端口可能收到多个应用程序的数据），或者按照TCP那样合并数据，必然会造成数据错误。我觉得关键的原因还是，TCP是面向连接，而UDP是无连接的，这就导致，TCP接收的数据为一个主机发来且有序无误的，而UDP可能是多个主机发来的无序，可能错误的。

TCP和UDP头部字节定义，

TCP和UDP三次握手和四次挥手状态及消息类型,

time\_wait，close\_wait状态产生原因，keepalive，

TIME\_WAIT：表示收到了对方的FIN报文，并发送出了ACK报文。 TIME\_WAIT状态下的TCP连接会等待2\*MSL（Max Segment Lifetime，最大分段生存期，指一个TCP报文在Internet上的最长生存时间。每个具体的TCP协议实现都必须选择一个确定的MSL值，RFC 1122建议是2分钟，但BSD传统实现采用了30秒，Linux可以cat /proc/sys/net/ipv4/tcp\_fin\_timeout看到本机的这个值），然后即可回到CLOSED 可用状态了。如果FIN\_WAIT\_1状态下，收到了对方同时带FIN标志和ACK标志的报文时，可以直接进入到TIME\_WAIT状态，而无须经过FIN\_WAIT\_2状态。

如果使用了nginx代理，那么系统TIME\_WAIT的数量会变得比较多，这是由于nginx代理使用了短链接的方式和后端交互的原因，使得nginx和后端的ESTABLISHED变得很少而TIME\_WAIT很多。这不但发生在安装nginx的代理服务器上，而且也会使后端的app服务器上有大量的TIME\_WAIT。查阅TIME\_WAIT资料，发现这个状态很多也没什么大问题，但可能因为它占用了系统过多的端口，导致后续的请求无法获取端口而造成障碍。

虽然TIME\_WAIT会造成一些问题，但是要完全枪毙掉它也是不正当的，虽然看起来这么做没什么错。具体可看这篇文档：

http://hi.baidu.com/tim\_bi/blog/item/35b005d784ca91d5a044df1d.html

所以目前看来最好的办法是让每个TIME\_WAIT早点过期。

在linux上可以这么配置：

#让TIME\_WAIT状态可以重用，这样即使TIME\_WAIT占满了所有端口，也不会拒绝新的请求造成障碍

echo "1" > /proc/sys/net/ipv4/tcp\_tw\_reuse

#让TIME\_WAIT尽快回收，我也不知是多久，观察大概是一秒钟

echo "1" > /proc/sys/net/ipv4/tcp\_tw\_recycle

很多文档都会建议两个参数都配置上，但是我发现只用修改tcp\_tw\_recycle就可以解决问题的了，TIME\_WAIT重用TCP协议本身就是不建议打开的。

不能重用端口可能会造成系统的某些服务无法启动，比如要重启一个系统监控的软件，它用了40000端口，而这个端口在软件重启过程中刚好被使用了，就可能会重启失败的。linux默认考虑到了这个问题，有这么个设定：

#查看系统本地可用端口极限值

cat /proc/sys/net/ipv4/ip\_local\_port\_range

用这条命令会返回两个数字，默认是：32768 61000，说明这台机器本地能向外连接61000-32768=28232个连接，注意是本地向外连接，不是这台机器的所有连接，不会影响这台机器的80端口的对外连接数。但这个数字会影响到代理服务器（nginx）对app服务器的最大连接数，因为nginx对app是用的异步传输，所以这个环节的连接速度很快，所以堆积的连接就很少。假如nginx对app服务器之间的带宽出了问题或是app服务器有问题，那么可能使连接堆积起来，这时可以通过设定nginx的代理超时时间，来使连接尽快释放掉，一般来说极少能用到28232个连接。

因为有软件使用了40000端口监听，常常出错的话，可以通过设定ip\_local\_port\_range的最小值来解决：

echo "40001 61000" > /proc/sys/net/ipv4/ip\_local\_port\_range

但是这么做很显然把系统可用端口数减少了，这时可以把ip\_local\_port\_range的最大值往上调，但是好习惯是使用不超过32768的端口来侦听服务，另外也不必要去修改ip\_local\_port\_range数值成1024 65535之类的，意义不大。

因为使用了nginx代理，在windows下也会造成大量TIME\_WAIT，当然windows也可以调整：

在注册表（regedit）的HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters上添加一个DWORD类型的值TcpTimedWaitDelay，值就是秒数，即可。

windows默认是重用TIME\_WAIT，我现在还不知道怎么改成不重用的，本地端口也没查到是什么值，但这些都关系不大，都可以按系统默认运作。

------------------------------------------------------------------------------------------------------------------------

TIME\_WAIT状态

根据TCP协议，主动发起关闭的一方，会进入TIME\_WAIT状态，持续2\*MSL(Max Segment Lifetime)，缺省为240秒，在这个post中简洁的介绍了为什么需要这个状态。

值得一说的是，对于基于TCP的HTTP协议，关闭TCP连接的是Server端，这样，Server端会进入TIME\_WAIT状态，可想而知，对于访问量大的Web Server，会存在大量的TIME\_WAIT状态，假如server一秒钟接收1000个请求，那么就会积压240\*1000=240，000个TIME\_WAIT的记录，维护这些状态给Server带来负担。当然现代操作系统都会用快速的查找算法来管理这些TIME\_WAIT，所以对于新的TCP连接请求，判断是否hit中一个TIME\_WAIT不会太费时间，但是有这么多状态要维护总是不好。

HTTP协议1.1版规定default行为是Keep-Alive，也就是会重用TCP连接传输多个request/response，一个主要原因就是发现了这个问题。还有一个方法减缓TIME\_WAIT压力就是把系统的2\*MSL时间减少，因为240秒的时间实在是忒长了点，对于Windows，修改注册表，在HKEY\_LOCAL\_MACHINE\ SYSTEM\CurrentControlSet\Services\ Tcpip\Parameters上添加一个DWORD类型的值TcpTimedWaitDelay，一般认为不要少于60，不然可能会有麻烦。

对于大型的服务，一台server搞不定，需要一个LB(Load Balancer)把流量分配到若干后端服务器上，如果这个LB是以NAT方式工作的话，可能会带来问题。假如所有从LB到后端Server的IP包的source address都是一样的(LB的对内地址），那么LB到后端Server的TCP连接会受限制，因为频繁的TCP连接建立和关闭，会在server上留下TIME\_WAIT状态，而且这些状态对应的remote address都是LB的，LB的source port撑死也就60000多个(2^16=65536,1~1023是保留端口，还有一些其他端口缺省也不会用），每个LB上的端口一旦进入Server的TIME\_WAIT黑名单，就有240秒不能再用来建立和Server的连接，这样LB和Server最多也就能支持300个左右的连接。如果没有LB，不会有这个问题，因为这样server看到的remote address是internet上广阔无垠的集合，对每个address，60000多个port实在是够用了。

一开始我觉得用上LB会很大程度上限制TCP的连接数，但是实验表明没这回事，LB后面的一台Windows Server 2003每秒处理请求数照样达到了600个，难道TIME\_WAIT状态没起作用？用Net Monitor和netstat观察后发现，Server和LB的XXXX端口之间的连接进入TIME\_WAIT状态后，再来一个LB的XXXX端口的SYN包，Server照样接收处理了，而是想像的那样被drop掉了。翻书，从书堆里面找出覆满尘土的大学时代买的《UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI》，中间提到一句，对于BSD-derived实现，只要SYN的sequence number比上一次关闭时的最大sequence number还要大，那么TIME\_WAIT状态一样接受这个SYN，难不成Windows也算BSD-derived?有了这点线索和关键字(BSD)，找到这个post，在NT4.0的时候，还是和BSD-derived不一样的，不过Windows Server 2003已经是NT5.2了，也许有点差别了。

做个试验，用Socket API编一个Client端，每次都Bind到本地一个端口比如2345，重复的建立TCP连接往一个Server发送Keep-Alive=false的HTTP请求，Windows的实现让sequence number不断的增长，所以虽然Server对于Client的2345端口连接保持TIME\_WAIT状态，但是总是能够接受新的请求，不会拒绝。那如果SYN的Sequence Number变小会怎么样呢？同样用Socket API，不过这次用Raw IP，发送一个小sequence number的SYN包过去，Net Monitor里面看到，这个SYN被Server接收后如泥牛如海，一点反应没有，被drop掉了。

按照书上的说法，BSD-derived和Windows Server 2003的做法有安全隐患，不过至少这样至少不会出现TIME\_WAIT阻止TCP请求的问题，当然，客户端要配合，保证不同TCP连接的sequence number要上涨不要下降。

----------------------------------------------------------------------------------------------------------------------------

Socket中的TIME\_WAIT状态

在高并发短连接的server端，当server处理完client的请求后立刻closesocket此时会出现time\_wait状态然后如果client再并发2000个连接，此时部分连接就连接不上了,用linger强制关闭可以解决此问题，但是linger会导致数据丢失，linger值为0时是强制关闭,无论并发多少多能正常连接上,如果非0会发生部分连接不上的情况!（可调用setsockopt设置套接字的linger延时标志，同时将延时时间设置为0。）

TCP/IP的RFC文档。TIME\_WAIT是TCP连接断开时必定会出现的状态。

是无法避免掉的，这是TCP协议实现的一部分。

在WINDOWS下，可以修改注册表让这个时间变短一些

time\_wait的时间为2msl,默认为4min.

你可以通过改变这个变量:

TcpTimedWaitDelay

把它缩短到30s

TCP要保证在所有可能的情况下使得所有的数据都能够被投递。当你关闭一个socket时，主动关闭一端的socket将进入TIME\_WAIT状态，而被动关闭一方则转入CLOSED状态，这的确能够保证所有的数据都被传输。当一个socket关闭的时候，是通过两端互发信息的四次握手过程完成的，当一端调用close()时，就说明本端没有数据再要发送了。这好似看来在握手完成以后，socket就都应该处于关闭CLOSED状态了。但这有两个问题，首先，我们没有任何机制保证最后的一个ACK能够正常传输，第二，网络上仍然有可能有残余的数据包(wandering duplicates)，我们也必须能够正常处理。

通过正确的状态机，我们知道双方的关闭过程如下

图

假设最后一个ACK丢失了，服务器会重发它发送的最后一个FIN，所以客户端必须维持一个状态信息，以便能够重发ACK；如果不维持这种状态，客户端在接收到FIN后将会响应一个RST，服务器端接收到RST后会认为这是一个错误。如果TCP协议能够正常完成必要的操作而终止双方的数据流传输，就必须完全正确的传输四次握手的四个节，不能有任何的丢失。这就是为什么socket在关闭后，仍然处于 TIME\_WAIT状态，因为他要等待以便重发ACK。

如果目前连接的通信双方都已经调用了close()，假定双方都到达CLOSED状态，而没有TIME\_WAIT状态时，就会出现如下的情况。现在有一个新的连接被建立起来，使用的IP地址与端口与先前的完全相同，后建立的连接又称作是原先连接的一个化身。还假定原先的连接中有数据报残存于网络之中，这样新的连接收到的数据报中有可能是先前连接的数据报。为了防止这一点，TCP不允许从处于TIME\_WAIT状态的socket建立一个连接。处于TIME\_WAIT状态的socket在等待两倍的MSL时间以后（之所以是两倍的MSL，是由于MSL是一个数据报在网络中单向发出到认定丢失的时间，一个数据报有可能在发送图中或是其响应过程中成为残余数据报，确认一个数据报及其响应的丢弃的需要两倍的MSL），将会转变为CLOSED状态。这就意味着，一个成功建立的连接，必然使得先前网络中残余的数据报都丢失了。

由于TIME\_WAIT状态所带来的相关问题，我们可以通过设置SO\_LINGER标志来避免socket进入TIME\_WAIT状态，这可以通过发送RST而取代正常的TCP四次握手的终止方式。但这并不是一个很好的主意，TIME\_WAIT对于我们来说往往是有利的。

客户端与服务器端建立TCP/IP连接后关闭SOCKET后，服务器端连接的端口

状态为TIME\_WAIT

是不是所有执行主动关闭的socket都会进入TIME\_WAIT状态呢？

有没有什么情况使主动关闭的socket直接进入CLOSED状态呢？

主动关闭的一方在发送最后一个 ack 后

就会进入 TIME\_WAIT 状态 停留2MSL（max segment lifetime）时间

这个是TCP/IP必不可少的，也就是“解决”不了的。

也就是TCP/IP设计者本来是这么设计的

主要有两个原因

1。防止上一次连接中的包，迷路后重新出现，影响新连接

（经过2MSL，上一次连接中所有的重复包都会消失）

2。可靠的关闭TCP连接

在主动关闭方发送的最后一个 ack(fin) ，有可能丢失，这时被动方会重新发

fin, 如果这时主动方处于 CLOSED 状态 ，就会响应 rst 而不是 ack。所以

主动方要处于 TIME\_WAIT 状态，而不能是 CLOSED 。

TIME\_WAIT 并不会占用很大资源的，除非受到攻击。

还有，如果一方 send 或 recv 超时，就会直接进入 CLOSED 状态

socket-faq中的这一段讲的也很好，摘录如下：

2.7. Please explain the TIME\_WAIT state.

什么是滑动窗口，超时重传，

列举你所知道的tcp选项，

connect会阻塞检测及防止，socket什么情况下可读？

connect会阻塞，怎么解决?(必考必问)

最通常的方法最有效的是加定时器；也可以采用非阻塞模式。

设置非阻塞，返回之后用select检测状态)

如果select返回可读，结果只读到0字节，什么情况？

某个套接字集合中没有准备好，可能会select内存用FD\_CLR清该位为0；

socket什么情况下可读？

每次读操作返回前都要检查是否还有剩余数据没读完，如果是的话保持数据有效标志，不这样设计的话会出现明显的不一致，那就是数据在读缓冲但没有读有效标志。

keepalive是什么东东？如何使用？

设置Keepalive参数，检测已中断的客户连接

在TCP中有一个Keep-alive的机制可以检测死连接，原理很简单，TCP会在空闲了一定时间后发送数据给对方：

1.如果主机可达，对方就会响应ACK应答，就认为是存活的。

2.如果可达，但应用程序退出，对方就发RST应答，发送TCP撤消连接。

3.如果可达，但应用程序崩溃，对方就发FIN消息。

4.如果对方主机不响应ack, rst，继续发送直到超时，就撤消连接。这个时间就是默认

的二个小时。

UDP中使用connect的好处:

1:会提升效率.前面已经描述了.2:高并发服务中会增加系统稳定性.原因:假设client A 通过非connect的UDP与serverB,C通信.B,C提供相同服务.为了负载均衡,我们让A与B,C交替通信.A 与 B通信IPa:PORTa<----> IPb:PORTbA 与 C通信IPa:PORTa'<---->IPc:PORTc

假设PORTa 与 PORTa'相同了(在大并发情况下会发生这种情况),那么就有可能出现A等待B的报文,却收到了C的报文.导致收报错误.解决方法内就是采用connect的UDP通信方式.在A中创建两个udp,然后分别connect到B,C.

长连接和短连接,

DNS和HTTP协议，HTTP请求方式，

cookie,session,localstroage,

一致性哈希负载均衡，

描述在浏览器中敲入一个网址并按下回车后所发生的事情，

PING命令

ping命令所利用的原理是这样的:网络上的机器都有唯一确定的IP地址，我们给目标IP地址发送一个数据包，对方就要返回一个同样大小的数据包，根据返回的数据包我们可以确定目标主机的存在，可以初步判断目标主机的操作系统等。

##数据库： 谈谈你对数据库中索引的理解，索引和主键区别

* 聚集索引一个表只能有一个，而非聚集索引一个表可以存在多个。
* 聚集索引存储记录是物理上连续存在，而非聚集索引是逻辑上的连续，物理存储并不连续。
* 聚集索引：该索引中键值的逻辑顺序决定了表中相应行的物理顺序。
* 聚集索引确定表中数据的物理顺序。聚集索引类似于电话簿，后者按姓氏排列数据。由于聚集索引规定数据在表中的物理存储顺序，因此一个表只能包含一个聚集索引。但该索引可以包含多个列（组合索引），就像电话簿按姓氏和名字进行组织一样。
* 聚集索引使用注意事项
* 定义聚集索引键时使用的列越少越好。
* • 包含大量非重复值的列。
* .• 使用下列运算符返回一个范围值的查询：BETWEEN、>、>=、< 和 <=。
* • 被连续访问的列。
* • 回大型结果集的查询。
* • 经常被使用联接或 GROUP BY 子句的查询访问的列；一般来说，这些是外键列。对 ORDER BY 或 GROUP BY 子句中指定的列进行索引，可以使 SQL Server 不必对数据进行排序，因为这些行已经排序。这样可以提高查询性能。
* • OLTP 类型的应用程序，这些程序要求进行非常快速的单行查找（一般通过主键）。应在主键上创建聚集索引。
* 聚集索引不适用于：
* • 频繁更改的列 。这将导致整行移动（因为 SQL Server 必须按物理顺序保留行中的数据值）。这一点要特别注意，因为在大数据量事务处理系统中数据是易失的。
* • 宽键 。来自聚集索引的键值由所有非聚集索引作为查找键使用，因此存储在每个非聚集索引的叶条目内。
* 非聚集索引：数据存储在一个地方，索引存储在另一个地方，索引带有指针指向数据的存储位置。
* 非聚集索引中的项目按索引键值的顺序存储，而表中的信息按另一种顺序存储（这可以由聚集索引规定）。对于非聚集索引，可以为在表非聚集索引中查找数据时常用的每个列创建一个非聚集索引。有些书籍包含多个索引。例如，一本介绍园艺的书可能会包含一个植物通俗名称索引，和一个植物学名索引，因为这是读者查找信息的两种最常用的方法。
* 一个通俗的举例，说明两者的区别
* 其实，我们的汉语字典的正文本身就是一个聚集索引。比如，我们要查“安”字，就会很自然地翻开字典的前几页，因为“安”的拼音是“an”，而按照拼音排序汉字的字典是以英文字母“a”开头并以“z”结尾的，那么“安”字就自然地排在字典的前部。如果您翻完了所有以“a”开头的部分仍然找不到这个字，那么就说明您的字典中没有这个字；同样的，如果查“张”字，那您也会将您的字典翻到最后部分，因为“张”的拼音是“zhang”。也就是说，字典的正文部分本身就是一个目录，您不需要再去查其他目录来找到您需要找的内容。我们把这种正文内容本身就是一种按照一定规则排列的目录称为“聚集索引”。
* 如果您认识某个字，您可以快速地从自动中查到这个字。但您也可能会遇到您不认识的字，不知道它的发音，这时候，您就不能按照刚才的方法找到您要查的字，而需要去根据“偏旁部首”查到您要找的字，然后根据这个字后的页码直接翻到某页来找到您要找的字。但您结合“部首目录”和“检字表”而查到的字的排序并不是真正的正文的排序方法，比如您查“张”字，我们可以看到在查部首之后的检字表中“张”的页码是672页，检字表中“张”的上面是“驰”字，但页码却是63页，“张”的下面是“弩”字，页面是390页。很显然，这些字并不是真正的分别位于“张”字的上下方，现在您看到的连续的“驰、张、弩”三字实际上就是他们在非聚集索引中的排序，是字典正文中的字在非聚集索引中的映射。我们可以通过这种方式来找到您所需要的字，但它需要两个过程，先找到目录中的结果，然后再翻到您所需要的页码。我们把这种目录纯粹是目录，正文纯粹是正文的排序方式称为“非聚集索引”。
* 第一：聚集索引的约束是唯一性，是否要求字段也是唯一的呢？
* 分析：如果认为是的朋友，可能是受系统默认设置的影响，一般我们指定一个表的主键，如果这个表之前没有聚集索引，同时建立主键时候没有强制指定使用非聚集索引,SQL会默认在此字段上创建一个聚集索引，而主键都是唯一的，所以理所当然的认为创建聚集索引的字段也需要唯一。
* 结论：聚集索引可以创建在任何一列你想创建的字段上，这是从理论上讲，实际情况并不能随便指定，否则在性能上会是恶梦。
* 第二：为什么聚集索引可以创建在任何一列上，如果此表没有主键约束，即有可能存在重复行数据呢？
* 粗一看，这还真是和聚集索引的约束相背，但实际情况真可以创建聚集索引。
* 分析其原因是：如果未使用 UNIQUE 属性创建聚集索引，数据库引擎将向表自动添加一个四字节 uniqueifier 列。必要时，数据库引擎 将向行自动添加一个 uniqueifier 值，使每个键唯一。此列和列值供内部使用，用户不能查看或访问。
* 第三：是不是聚集索引就一定要比非聚集索引性能优呢？
* 如果想查询学分在60-90之间的学生的学分以及姓名，在学分上创建聚集索引是否是最优的呢？
* 答：否。既然只输出两列，我们可以在学分以及学生姓名上创建联合非聚集索引，此时的索引就形成了覆盖索引，即索引所存储的内容就是最终输出的数据，这种索引在比以学分为聚集索引做查询性能更好。
* 第四：在数据库中通过什么描述聚集索引与非聚集索引的？
* 索引是通过二叉树的形式进行描述的，我们可以这样区分聚集与非聚集索引的区别：聚集索引的叶节点就是最终的数据节点，而非聚集索引的叶节仍然是索引节点，但它有一个指向最终数据的指针。
* 第五：在主键是创建聚集索引的表在数据插入上为什么比主键上创建非聚集索引表速度要慢？
* 有了上面第四点的认识，我们分析这个问题就有把握了，在有主键的表中插入数据行，由于有主键唯一性的约束，所以需要保证插入的数据没有重复。我们来比较下主键为聚集索引和非聚集索引的查找情况:聚集索引由于索引叶节点就是数据页，所以如果想检查主键的唯一性，需要遍历所有数据节点才行，但非聚集索引不同，由于非聚集索引上已经包含了主键值，所以查找主键唯一性，只需要遍历所有的索引页就行，这比遍历所有数据行减少了不少IO消耗。这就是为什么主键上创建非聚集索引比主键上创建聚集索引在插入数据时要快的真正原因。

现在普通关系数据库用得数据结构是什么类型的数据结构，

B+树

[MySQL索引背后的数据结构及算法原理](http://blog.codinglabs.org/articles/theory-of-mysql-index.html)

索引的优点和缺点，

建立索引的优点

1.大大加快数据的检索速度;

2.创建唯一性索引，保证数据库表中每一行数据的唯一性;

3.加速表和表之间的连接;

4.在使用分组和排序子句进行数据检索时，可以显著减少查询中分组和排序的时间。

索引的缺点

1.索引需要占物理空间。

2.当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，降低了数据的维护速度。

唯一索引

唯一索引是不允许其中任何两行具有相同索引值的索引。

当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在 employee 表中职员的姓 (lname) 上创建了唯一索引，则任何两个员工都不能同姓。

主键索引

数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。

在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。

聚集索引

在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。

如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度。

关系型数据库和非关系数据库的特点，

简单来说，关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。

非关系型数据库提出另一种理念，例如，以键值对存储，且结构不固定，每一个元组可以有不一样的字段，每个元组可以根据需要增加一些自己的键值对，这 样就不会局限于固定的结构，可以减少一些时间和空间的开销。使用这种方式，用户可以根据需要去添加自己需要的字段，这样，为了获取用户的不同信息，不需要 像关系型数据库中，要对多表进行关联查询。仅需要根据id取出相应的value就可以完成查询。但非关系型数据库由于很少的约束，他也不能够提供像SQL 所提供的where这种对于字段属性值情况的查询。并且难以体现设计的完整性。他只适合存储一些较为简单的数据，对于需要进行较复杂查询的数据，SQL数 据库显的更为合适。

关系型数据库的最大特点就是事务的一致性：传统的关系型数据库读写操作都是事务的，具有ACID的特点，这个特性使得关系型数据库可以用于几乎所有对一致性有要求的系统中，如典型的银行系统。

但是，在网页应用中，尤其是SNS应用中，一致性却不是显得那么重要，用户A看到的内容和用户B看到同一用户C内容更新不一致是可以容忍的，或者 说，两个人看到同一好友的数据更新的时间差那么几秒是可以容忍的，因此，关系型数据库的最大特点在这里已经无用武之地，起码不是那么重要了。

相反地，关系型数据库为了维护一致性所付出的巨大代价就是其读写性能比较差，而像微博、facebook这类SNS的应用，对并发读写能力要求极 高，关系型数据库已经无法应付(在读方面，传统上为了克服关系型数据库缺陷，提高性能，都是增加一级memcache来静态化网页，而在SNS中，变化太 快，memchache已经无能为力了)，因此，必须用新的一种数据结构存储来代替关系数据库。

关系数据库的另一个特点就是其具有固定的表结构，因此，其扩展性极差，而在SNS中，系统的升级，功能的增加，往往意味着数据结构巨大变动，这一点关系型数据库也难以应付，需要新的结构化数据存储。

于是，非关系型数据库应运而生，由于不可能用一种数据结构化存储应付所有的新的需求，因此，非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。

必须强调的是，数据的持久存储，尤其是海量数据的持久存储，还是需要一种关系数据库这员老将。

非关系型数据库分类：

主要分为以下几类：

面向高性能并发读写的key-value数据库：

key-value数据库的主要特点即使具有极高的并发读写性能，Redis,Tokyo Cabinet,Flare就是这类的代表

面向海量数据访问的面向文档数据库：

这类数据库的特点是，可以在海量的数据中快速的查询数据，典型代表为MongoDB以及CouchDB

面向可扩展性的分布式数据库：

这类数据库想解决的问题就是传统数据库存在可扩展性上的缺陷，这类数据库可以适应数据量的增加以及数据结构的变化

[关系型数据库和非关系型数据库](http://my.oschina.net/u/1773689/blog/364548)

乐观锁与悲观锁的区别，

悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。[1] 悲观锁假定其他用户企图访问或者改变你正在访问、更改的对象的概率是很高的，因此在悲观锁的环境中，在你开始改变此对象之前就将该对象锁住，并且直到你提交了所作的更改之后才释放锁。悲观的缺陷是不论是页锁还是行锁，加锁的时间可能会很长，这样可能会长时间的限制其他用户的访问，也就是说悲观锁的并发访问性不好。

乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。[1] 乐观锁不能解决脏读的问题。 乐观锁则认为其他用户企图改变你正在更改的对象的概率是很小的，因此乐观锁直到你准备提交所作的更改时才将对象锁住，当你读取以及改变该对象时并不加锁。可见乐观锁加锁的时间要比悲观锁短，乐观锁可以用较大的锁粒度获得较好的并发访问性能。但是如果第二个用户恰好在第一个用户提交更改之前读取了该对象，那么当他完成了自己的更改进行提交时，数据库就会发现该对象已经变化了，这样，第二个用户不得不重新读取该对象并作出更改。这说明在乐观锁环境中，会增加并发用户读取对象的次数。

从数据库厂商的角度看，使用乐观的页锁是比较好的，尤其在影响很多行的批量操作中可以放比较少的锁，从而降低对资源的需求提高数据库的性能。再考虑聚集索引。在数据库中记录是按照聚集索引的物理顺序存放的。如果使用页锁，当两个用户同时访问更改位于同一数据页上的相邻两行时，其中一个用户必须等待另一个用户释放锁，这会明显地降低系统的性能。interbase和大多数关系数据库一样，采用的是乐观锁，而且读锁是共享的，写锁是排他的。可以在一个读锁上再放置读锁，但不能再放置写锁；你不能在写锁上再放置任何锁。锁是目前解决多用户并发访问的有效手段。

[乐观锁与悲观锁的区别](http://www.cnblogs.com/Bob-FD/p/3352216.html)

数据库范式

1NF的定义为：符合1NF的关系中的每个属性都不可再分,1NF是所有关系型数据库的最基本要求，

2NF在1NF的基础之上，消除了非主属性对于码的部分函数依赖。

3NF在2NF的基础之上，消除了非主属性对于码的传递函数依赖。也就是说， 如果存在非主属性对于码的传递函数依赖，则不符合3NF的要求。

BCNF范式在 3NF 的基础上消除主属性对于码的部分与传递函数依赖。

[解释一下关系数据库的第一第二第三范式？](http://www.zhihu.com/question/24696366)

数据库日志类型作用

MySQL日志文件分类

1.错误日志(Error Log)

2.二进制日志(Binary Log & Binary Log Index)

3.通用查询日志(query log)

4.慢查询日志(slow query log)

5.Innodb的在线 redo 日志(innodb redo log)

6.更新日志(update log)

innodb和myisam的区别

innodb，聚集索引，支持外键和事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))，不支持全文索引，不支持计数，统计的时候会遍历，

myisam，非聚集索引，不支持外键事务，支持全文索引，支持计数，查询效果较好

你对innodb哪里最熟悉

innodb的索引有哪几种类型

4种，hash，b-tree，spatial，full-text

1. B-Tree索引

最常见的索引类型，基于B-Tree数据结构。B-Tree的基本思想是，所有值（被索引的列）都是排过序的，每个叶节点到跟节点距离相等。所以B-Tree适合用来查找某一范围内的数据，而且可以直接支持数据排序（ORDER BY）。但是当索引多列时，列的顺序特别重要，需要格外注意。InnoDB和MyISAM都支持B-Tree索引。InnoDB用的是一个变种B+Tree，而MyISAM为了节省空间对索引进行了压缩，从而牺牲了性能。

2. Hash索引

基于hash表。所以这种索引只支持精确查找，不支持范围查找，不支持排序。这意味着范围查找或ORDER BY都要依赖server层的额外工作。目前只有Memory引擎支持显式的hash索引（但是它的hash是nonunique的，冲突太多时也会影响查找性能）。Memory引擎默认的索引类型即是Hash索引，虽然它也支持B-Tree索引。

Hash 索引仅仅能满足"=","IN"和"<=>"查询，不能使用范围查询。

例子：

CREATE TABLE testhash (

fname VARCHAR(50) NOT NULL,

lname VARCHAR(50) NOT NULL,

KEY USING HASH(fname)

) ENGINE =MEMORY;

3. Spatial (R-Tree)（空间）索引

只有MyISAM引擎支持，并且支持的不好。可以忽略。

4. Full-text索引

主要用来查找文本中的关键字，而不是直接与索引中的值相比较。Full-text索引跟其它索引大不相同，它更像是一个搜索引擎，而不是简单的WHERE语句的参数匹配。你可以对某列分别进行full-text索引和B-Tree索引，两者互不冲突。Full-text索引配合MATCH AGAINST操作使用，而不是一般的WHERE语句加LIKE。

http://segmentfault.com/q/1010000003832312

B TREE 和B+TREE的区别

innodb有全文索引吗

没有，myisam有

union和join

JOIN用于按照ON条件联接两个表，主要有四种：

INNER JOIN：内部联接两个表中的记录，仅当至少有一个同属于两表的行符合联接条件时，内联接才返回行。我理解的是只要记录不符合ON条件，就不会显示在结果集内。

LEFT JOIN / LEFT OUTER JOIN：外部联接两个表中的记录，并包含左表中的全部记录。如果左表的某记录在右表中没有匹配记录，则在相关联的结果集中右表的所有选择列表列均为空值。理解为即使不符合ON条件，左表中的记录也全部显示出来，且结果集中该类记录的右表字段为空值。

RIGHT JOIN / RIGHT OUTER JOIN：外部联接两个表中的记录，并包含右表中的全部记录。简单说就是和LEFT JOIN反过来。

FULL JOIN / FULL OUTER JOIN： 完整外部联接返回左表和右表中的所有行。就是LEFT JOIN和RIGHT JOIN和合并，左右两表的数据都全部显示。

JOIN的基本语法：

Select table1.\* FROM table1 JOIN table2 ON table1.id=table2.id

UNION运算符

将两个或更多查询的结果集组合为单个结果集，该结果集包含联合查询中的所有查询的全部行。UNION的结果集列名与UNION运算符中第一个Select语句的结果集的列名相同。另一个Select语句的结果集列名将被忽略。

其中两种不同的用法是UNION和UNION ALL，区别在于UNION从结果集中删除重复的行。如果使用UNION ALL 将包含所有行并且将不删除重复的行。

相同点：在某些特定的情况下，可以用join实现union all的功能，这种情况是有条件的，当出现这种情况的时候选择union all还是group by就可以看情况或者看两者的消耗而决定。

http://chengheng1984.blog.163.com/blog/static/17947412201012215738844/

http://www.51testing.com/html/14/446214-249265.html

##海量数据处理： bitmap

Map-Reduce原理，

BloomFilter原理、

它实际上是一个很长的二进制向量和一系列随机映射函数（Hash函数）。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。Bloom Filter广泛的应用于各种需要查询的场合中，如Orocle的数据库，Google的BitTable也用了此技术。

Bloom Filter特点：

不存在漏报（False Negative），即某个元素在某个集合中，肯定能报出来。

可能存在误报（False Positive），即某个元素不在某个集合中，可能也被爆出来。

确定某个元素是否在某个集合中的代价和总的元素数目无关。

Trie树原理、 单词搜索树

B+树原理，

LSM树原理,

大数据处理

##工具： 编译工具GCC，调试工具GDB，性能优化工具Perf,内存泄露检查工具Valgrind,makefile编写 其他工具: netstat,ps,top,df,fdisk,lsof，ifconfig,uname,kill，tcpdump，ipcs，grep

##其他：安全，加密方式（DES，SHA）

附：[软件开发者面试百问](http://blog.csdn.net/programmer_editor/article/details/4004408) #相关书籍

语言类：

C:C程序设计语言（K&R）->C和指针->C专家编程->C陷阱与缺陷->你必须知道的495个C语言问题

C++: C++ primer -> effective C++->深度探索C++对象模型 ->stl源码分析->C++必知必会

java：java编程思想->java并发编程->深入理解Java虚拟机：JVM高级特性与最佳实践

算法和数据结构：

算法导论->数据结构与算法分析(维斯)->编程之美->剑指offer

操作系统：

深入理解计算机操作系统->编译原理（龙书）

鸟哥的linux私房菜->linux内核设计与实现->深入理解linux内核

linux shell脚本攻略（短小精悍）

网络编程：

TCP/IP协议详解v1->

unix高级环境编程->

unix网络编程（卷1&卷2）->

unix编程艺术(进阶)

视野：

大型网站技术架构：核心原理与案例分析，

深入理解nginx：模块开发与架构解析，

大规模分布式存储系统 : 原理解析与架构实战

其他：

程序员自我修养，

重构，

编写可读代码的艺术，

headfirst设计模式

#相关网络资源

Coolshell：http://coolshell.cn/

Matrix67大牛的博客：http://www.matrix67.com/blog/。

July的CSDN博客：http://blog.csdn.net/v\_JULY\_v。

何海涛博客：http://zhedahht.blog.163.com/。

笔试面试的经典：Cracking the coding interview--问题与解答：

http://hawstein.com/posts/ctci-solutions-contents.html

LeetCode：http://leetcode.com/

这里有不少笔试题集锦：http://blog.csdn.net/hackbuteer1

程序员编程艺术：面试和算法心得http://taop.marchtea.com/

### 技术一面

1.手写strncpy函数的实现。  
**strncpy函数总是正好向dst写入len个字符。如果strlen(src)的值小于len，dst数组就用额外的'\0'字符填充到len长度，如果strlen(src)的值大于或等于len，那么只有len个字符被复制到dst中。**  
**注意：如果strlen(src)的值大于或等于len，那么复制到dst中的字符将不包括'\0'。**

char \* strncpy(char \*dst, const char \*src, int len)

{

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

char \*temp = dst;

int i = 0;

int src\_len = strlen(src);

if(src\_len >= len) //src length大于或者等于len的时候，会拷贝len个有效字符(不包括'\0')到dst中

{

while(i <= len)

{

\*temp++ = \*src++;

i++;

}

}

else //src length小于len的时候，先拷贝src中所有的字符，再填充len - strlen(src)个'\0'

{

while(i <= src\_len)

{

\*temp++ = \*src++;

i++;

}

while(i <= len)

{

\*temp++ = '\0';

i++;

}

}

return dst;

}

2.strncpy函数的返回值是啥？  
�6�8�6�8strncpy函数的返回值是dst。

3.在一个字符串中找另一个字符串的子串。  
�6�8�6�8(1)C语言的库函数strstr，函数原型函数原型：extern char \*strstr(char \*str1, const char \*str2);  
判断str2是否是str1的子串。  
返回值：若str2是str1的子串，则返回str2在str1的首次出现的地址；如果str2不是str1的子串，则返回NULL。  
�6�8�6�8(2)KMP算法  
[链接：KMP算法详解](http://www.cnblogs.com/yjiyjige/p/3263858.html) ps:KMP算法不是效率最高的算法，实际采用并不多。  
�6�8�6�8(3)Boyer-Moore算法  
[链接：Boyer-Moore算法详解](http://www.ruanyifeng.com/blog/2013/05/boyer-moore_string_search_algorithm.html)  
�6�8�6�8(4)Sunday算法  
[链接：Sunday算法详解](https://blog.csdn.net/q547550831/article/details/51860017)

4.Linux系统中的僵尸进程。  
�6�8�6�8**僵尸进程：**在Linux系统中，一个进程结束了，但是它的父进程没有回收它，释放它占用的资源， 那么它将变成一个僵尸进程。  
�6�8�6�8另外，如果该进程的父进程已经先结束了，那么该进程就不会变成僵尸进程。 因为每个进程结束的时候，系统都会扫描当前系统中所运行的所有进程， 看有没有哪个进程是刚刚结束的这个进程的子进程，如果是的话，就由Init 来接管子进程，成为子进程的父进程。  
�6�8�6�8**怎样来清除僵尸进程：**  
�6�8�6�8(1)父进程通过wait和waitpid等函数等待子进程结束，这会导致父进程挂起。  
�6�8�6�8(2)如果父进程很忙，那么可以用signal函数为SIGCHLD安装handler，因为子进程结束后， 父进程会收到该信号，可以在handler中调用wait回收。  
�6�8�6�8(3)如果父进程不关心子进程什么时候结束，那么可以用signal（SIGCHLD,SIG\_IGN） 通知内核，自己对子进程的结束不感兴趣，那么子进程结束后，内核会回收， 并不再给父进程发送信号。  
�6�8�6�8(4)还有一些技巧，就是fork两次，父进程fork一个子进程，然后继续工作，子进程fork一 个孙进程后退出，那么孙进程被init接管，孙进程结束后，init会回收。不过,子进程的回收还是要自己做。

5.DHCP协议的交互过程。单个和多个DHCP server的区别。存在DHCP relay的情况。  
[链接：DHCP协议报文格式](http://blog.chinaunix.net/uid-20530497-id-2203830.html)  
�6�8�6�8DHCP交互过程分为四个阶段**(最基本的情况)：**  
�6�8�6�8(1)Discover： Client在以广播的形式DHCP Discover报文(DstMAC和DstIP都是全1，SrcIP全0，SrcMAC是Client的MAC)，广播域内的所有Server都会收到该报文。另外，Client可以在选项字段中加入“request paramter list” 选项，表明自己想要获得的各种参数。  
�6�8�6�8(2)Offer：广播域内的Server收到DHCP Discover报文之后，会发送DHCP Offer报文，报文中包含了Server准备分配给Client的IP地址，以及一些选项字段。  
�6�8�6�8(3)Request：如果广播域内有多个Server，那么Client会收到多个DHCP Offer报文。Client只能选择处理其中一个，一般的做法是选择最先收到的DHCP Offer报文。**然后Client就以广播方式回答一个DHCP request请求信息，该信息中包含Client选中的DHCP Server的IP地址和Client想要的IP地址。**之所以要以广播方式回答，是为了通知所有的Server，Client将选择某台Server所提供的IP地址。  
�6�8�6�8(4)ACK：Server收到DHCP Request报文后，判断选项字段中的DHCP Server的IP地址是否与自己的地址相同。如果不相同就不回应ACK报文，并且判断自己是否给该Request报文的Client发送过DHCP Offer报文。如果发送过DHCP Offer报文，则清除相应IP地址记录。如果选项字段中的DHCP Server的IP地址与自己的IP地址相同，Server就会响应一个DHCP ACK报文，其内容同DHCP Offer类似并在选项字段中增加了IP地址使用租期选项。Client收到ACK报文后，如果Server分配的IP地址不跟其他设备的IP地址冲突，那么Client将使用它作为自己的IP。

�6�8�6�8**复杂一些的情况：**  
�6�8�6�8(1)Client收到Server发送的ACK报文之后，会先发送一个免费ARP报文(SrcIP和DstIP都是Server分配的IP)。如果Client收到ARP应答了，就说明Server分配的IP已被占用，那么Client会给Server发送一个DHCP Decline报文，在Request IP Address（option 50）字段填入Server分配的发生冲突的IP地址。发送完成后，等待一段时间再开始重新申请IP地址，直至申请到一个可用的IP地址。  
�6�8�6�8(2)如果Client之前请求过IP地址，然后重启了，那么重启之后Client首先发送单播的DHCP Request报文到指定的Server，在Request IP Address（option 50）字段填入之前使用过的IP地址，等待Server的响应。如果收到了Server的ACK报文，那么Client确认该IP不冲突之后，将直接使用它。如果Server回应的NAK报文，那么Client将发送广播的DHCP Discover报文请求IP地址。  
�6�8�6�8(3)当DHCP Server分配给Client的IP地址的使用租期到达50%时(T1)，Client发送单播的DHCP Request报文到指定的Server，等待Server的响应。如果收到了Server的ACK报文，那么Client会刷新租期。如果没有收到Server的ACK报文，那么Client会继续使用该IP。  
�6�8�6�8(4)当DHCP Server分配给Client的IP地址的使用租期到达87.5%时(T2)，Client发送广播的DHCP Request报文，等待Server的响。如果收到了Server的ACK报文，那么Client会刷新租期。如果没有收到Server的ACK报文，那么Client会继续使用该IP，知道租期结束。  
�6�8�6�8(5)当DHCP Server分配给Client的IP地址的使用租期结束时，Client发送广播的DHCP Discover报文，重新申请IP地址。  
�6�8�6�8(6)已经动态获取到了IP地址的DHCP Client，如果需要向Server请求一些参数，那么它可以向Server发送DHCP Inform报文。Server收到DHCP Inform报文后，回应DHCP ACK报文。该ACK报文不包含分配的Client的IP地址，也不包含租期信息，但是在选项字段中包含了Client需要的参数。

�6�8�6�8**存在DHCP Relay的情况：**  
�6�8�6�8(1)如果DHCP Server和DHCP Client不在同一个网段，那么可以使用DHCP Relay中转。Relay收到Client发送的广播的DHCP Discover报文和DHCP Request报文后，在报文的giaddr字段填充自己的IP地址，并将报文转换成单播报文，然后发送给指定的Server(DHCP Relay上需要配置Server的IP地址)。Relay收到Server回应的Offer和ACK报文后，将其转发给Client。注意：DHCP Server给DHCP Client分配的IP地址与DHCP Relay添加的giaddr在同一网段。

6.linux系统常用的命令，比如查看文件大小，查看磁盘分区及使用情况。  
�6�8�6�8查看文件或者文件夹的大小：du -sh file\_or\_folder\_name  
�6�8�6�8查看磁盘分区及使用情况：df -h 或者 fdisk -l

7.AVL树的特点。  
�6�8�6�81.AVL树本质上还是一棵二叉搜索树，它的特点是：  
�6�8�6�8(1)本身首先是一棵二叉搜索树。  
�6�8�6�8(2)带有平衡条件：每个结点的左右子树的高度之差的绝对值（平衡因子）最多为1。  
�6�8�6�8也就是说，AVL树，本质上是带了平衡功能的二叉查找树（二叉排序树，二叉搜索树）。  
�6�8�6�82.AVL树为了达到平衡条件，必须进行旋转：  
�6�8�6�8(1)左单旋：当前节点的右子树比左子树高(平衡因子为1)，现在在它的**右子树的右侧**插入一个新节点，右子树高度增加1导致节点平衡因子从1变为变为2而不平衡。此时，将当前节点(不平衡的节点)逆时针旋转(左旋)，使得它原先的右子节点变成它的父节点，它本身则变成原先右子节点的左子节点，原先右子节点的左子节点变成它新的右子节点。

int left\_rotate(avl\_tree \*tree, avl\_node \*node) /\* 左单旋 \*/

{

avl\_node \*p;

p = node->right; /\* 原先的右子节点 \*/

node->right = p->left; /\* 原先右子节点的左子节点 -->它新的右子节点 \*/

if(p->left)

{

p->left->parent = node;

}

p->left = node; /\* 它本身-->原先右子节点的左子节点 \*/

if(node == tree->root)

{

tree->root = p;

}

p->parent = node->parent;

node->parent = p; /\* 它原先的右子节点-->它的父节点 \*/

return 0;

}

�6�8�6�8(2)右单旋：当前节点左子树比右子树高(平衡因子为-1)，现在在**左子树的左侧**插入一个新节点，左子树的高度增加1,导致该节点的平衡因子从-1变为-2而不平衡。此时，将当前节点(不平衡的节点)顺时针旋转(右旋)，使得它原先的左子节点变成它的父节点，它本身则变成原先左子节点的右子节点，原先左子节点的右子节点变成它新的左子节点。

struct avl\_node

{

struct avl\_node \*left;

struct avl\_node \*right;

struct avl\_node \*parent;

int bf; //balance factor

void \*info;

};

struct avl\_tree

{

int (\*compare\_func)(void \*data1, void \*data2); /\* 用来比较avl树的节点的值的大小 \*/

struct avl\_node \*root;

int count;

};

int right\_rotate(avl\_tree \*tree, avl\_node \*node) /\* 右单旋 \*/

{

avl\_node \*p;

p = node->left; /\* 原先的左子节点 \*/

node->left = p->right; /\* 原先的左子节点的右子节点-->它新的左子节点 \*/

if(p->right)

{

p->right->parent = node;

}

p->right = node;/\* 它本身-->原先的左子节点的右子节点 \*/

if(node == tree->root)

{

tree->root = p;

}

p->parent = node->parent;

node->parent = p;/\* 它原先的左子节点-->它的父节点 \*/

return 0;

}

�6�8�6�8(3)左右双旋(先左旋，再右旋)：当前节点的左子树比右子树高(平衡因子为-1)，现在在它的**左子树的右子树**上插入一个新的节点，它的左子树的高度增加1，导致该节点的平衡因子从-1变为-2而不平衡。此时，将先当前节点(不平衡节点)的左子节点逆时针旋转(左旋)，再将当前节点顺时针旋转(右旋)，就可以使得AVL树恢复平衡。

int left\_right\_rotate(avl\_tree \*tree, avl\_node \*node) /\*先左旋，再右旋\*/

{

left\_rotate(tree, node->left);

right\_rotate(tree,node);

return 0;

}

�6�8�6�8(4)右左双旋(先右旋，再左旋)：当前节点的右子树比左子树高(平衡因子为1)，现在在它的**右子树的左子树**上插入一个新的节点，它的右子树的高度增加1，导致该节点的平衡因子从1变为2而不平衡。此时，将先当前节点(不平衡节点)的右子节点顺时针旋转(右旋)，再将当前节点逆时针旋转(左旋)，就可以使得AVL树恢复平衡。

int right\_left\_rotate(avl\_tree \*tree, avl\_node \*node) /\*先右旋，再左旋\*/

{

right\_rotate(tree, node->right);

left\_rotate(tree,node);

return 0;

}

[链接：AVL树插入操作(旋转)图解](https://www.cnblogs.com/MrListening/p/5788842.html?hmsr=toutiao.io&utm_medium=toutiao.io&utm_source=toutiao.io)

8.报文检测系统(我简历中写的一个项目)的具体实现。

9.socket调用时可能出现的错误，以及解决方法。  
[链接：socket常见错误](https://blog.csdn.net/epeaktop/article/details/48714435)  
[链接：socket常见异常](https://blog.csdn.net/qq_29951983/article/details/79379424)

10.如何判断链表是否有环？  
�6�8�6�8判断链表是否有环可以用快慢指针，fast指针一次走两步，slow指针一次走一步。如果链表确实有环，那么fast和slow会相遇；否则不会相遇。  
�6�8�6�8假设链表有环，环长度为r，fast指针走过的距离为s1，slow指针走过的距离为s2。  
�6�8�6�8可以得到s1 = s2 + n\*r，(n >= 1)

typedef struct listnode

{

struct listnode \*next;

struct listnode \*prev;

void \*data;

}NODE;

typedef struct list

{

struct listnode \*head;

struct listnode \*tail;

unsigned int count;

int (\*compare\_func)(void \*data1, void \*data2); /\* 用来比较链表元素的值的大小 \*/

}LIST;

NODE \* have\_loop\_or\_not(LIST \*list) /\* 判断链表是否有环 \*/

{

NODE \*fast, \*slow;

if(!list)

return NULL;

slow = fast = list->head;

while(slow && fast && fast->next)

{

slow = slow->next;

fast = fast->next->next;

if(slow == fast)

return slow;

}

return NULL;

}

[链接：链表是否有环及相关引申问题](https://www.cnblogs.com/dancingrain/p/3405197.html)

11.如何判断两个链表(没有环)是否相交？  
�6�8�6�8如果两个没有环的链表相交，那么相交节点之后的所有节点必定是完全相同的。两个相交链表呈Y型。  
�6�8�6�8假设这两个链表分别是listA和listB。现在让listA的最后一个节点指向它的head，如果listA和listB相交，那么listB会有环。判断listB是否有环就可以判断A,B是否相交。

12.snmp相关的内容。比如trap上报了哪些信息？比如网元侧的AgentX如何处理snmp报文？

### 技术二面

1.双主控板(我简历中写的一个分布式路由器项目)的时候，主用和备用主控板之间如何保持通信？如何倒换(备用主控什么时候变成主用)？

2.假如某个时刻存在双主用，那么如何处理，怎么决策出一个主用主控板？

3.linux内核的收包过程。

4.Linux中的守护进程。  
�6�8�6�8守护进程的概念：  
�6�8�6�8守护进程（Daemon）是一种运行在后台的一种特殊的进程，它独立于控制终端并且周期性的执行某种任务或等待处理某些发生的事件。由于在linux中，每个系统与用户进行交流的界面成为终端，每一个从此终端开始运行的进程都会依附于这个终端，这个终端被称为这些进程的控制终端，当控制终端被关闭的时候，相应的进程都会自动关闭。但是守护进程却能突破这种限制，它脱离于终端并且在后台运行，并且它脱离终端的目的是为了避免进程在运行的过程中的信息在任何终端中显示并且进程也不会被任何终端所产生的终端信息所打断。它从被执行的时候开始运转，知道整个系统关闭才退出（当然可以认为的杀死相应的守护进程）。如果想让某个进程不因为用户或中断或其他变化而影响，那么就必须把这个进程变成一个守护进程。  
[链接：深入理解Linux操作系统守护进程的意义](https://blog.csdn.net/farmwang/article/details/73477632)

5.字符串的搜索，比如CLI命令行的快速匹配与搜索。

6.链表的插入操作。

NODE \* listnode\_add(LIST \*list, void \*val)

{/\* 在链表尾部插入节点 \*/

NODE \*node;

if((!list) || (!val))

return NULL;

node = (NODE \*)calloc(1, sizeof(NODE));

if(!node)

return NULL;

node->prev = list->tail;

node->data = val;

if(list->head == NULL)

list->head = node;

else

list->tail->next = node;

list->tail = node;

list->count++;

return node;

}

NODE \* listnode\_add\_sort(LIST \*list, void \*val)

{/\* 往有序链表中插入元素

\*(链表元素的值按照从小到大顺序排序)

\*/

NODE \*n, \*node;

if((!list) || (!val))

return NULL;

node = (NODE \*)calloc(1, sizeof(NODE));

if(!node)

return NULL;

node->data = val;

if(list->compare\_func)

{

for(n = list->tail; n; n = n->prev)

{

if((list->compare\_func(val, n->data)) >= 0)

{/\* 在中间或者尾部插入元素 \*/

node->prev = n;

node->next = n->next;

if(n->next)

n->next->prev = node;

else

list->tail = node;

n->next = node;

list->count++;

return node;

}

}

}

/\* 在头部插入元素 \*/

node->next = list->head;

if(list->head)

list->head->prev = node;

else

list->tail = node;

list->head = node;

list->count++;

return node;

}

7.如何处理广播风暴？  
�6�8�6�8(1)什么是广播风暴：网络上的广播帧（由于被转发）数量急剧增加而影响正常的网络通讯的反常现象，广播风暴会占用相当可观的网络带宽，造成整个网络无法正常工作。  
�6�8�6�8(2)广播风暴只会在二层出现，广播报文跨越不了网段。  
�6�8�6�8(3)广播风暴的产生有几个可能的原因：网卡损坏、网络环路、网络病毒。  
�6�8�6�8(4)广播风暴的避免/抑制/解决方法：  
�6�8�6�8�6�8�6�8a)开启交换机端口的广播风暴控制，当端口收到的广播帧累计到预定门限值时，端口将自动丢弃收到的广播帧。  
�6�8�6�8�6�8�6�8b)vlan隔离：按照端口/MAC/子网划分vlan，可以将广播风暴的影响范围限制在vlan内。  
�6�8�6�8�6�8�6�8c)交换机开启生成树协议(STP/RSTP/MSTP)，可以避免网络环路造成的广播风暴。  
�6�8�6�8�6�8�6�8d)开启安全防护软件，可以减少网络病毒造成的广播风暴。

8.分布式的一致性问题。  
[链接：解决分布式一致性问题的方案](https://www.cnblogs.com/soundcode/p/5590710.html)

9.平时有看过那些技术书籍？

10.平时有看过哪些开源的代码，或者参与过哪些开源的项目？

总的来说深信服的技术面会参考应聘者的简历，问一些跟简历相关的内容。  
另外，他们还会让应聘者手写代码，实现基本的算法，考察应聘者数据结构与算法的功底