# C++为什么效率比Java等其他语言高？

java实际上也要用指针，不过都是虚拟机JVM来完成的，所以多了一道手续，自然慢一拍

因为C/C++是基于系统底层的语言,直接编译成二进制代码执行,而JAVA编译后由虚拟机执行,虚拟机的效率自然要低一些了.

# 5大内存分区

C++编译时内存分为几个区

堆（new delete），栈，自由存储区（就是那些由malloc等分配的内存块，它和堆是十分相似的，不过它是用free来结束自己的生命的），全局/静态存储区(全局变量和static)，常量区（const常量）

堆，栈，静态存储区，常量区，自由存储区

# C++/C之间区别

设计思想上：

C++是面向对象的语言，而C是面向过程的结构化编程语言

语法上：

C++具有封装、继承和多态三种特性

C++相比C，增加多许多类型安全的功能，比如强制类型转换、

C++支持范式编程，比如模板类、函数模板等

特点：

c++是面向对象的编程语言， 其强调的是在运行阶段进行决策

# 从源文件到二进制文件的过程

一个源程序到一个可执行程序的过程：预编译、编译、汇编、链接。  
其中，编译是主要部分，其中又分为六个部分：词法分析、语法分析、语义分析、中间代码生成、目标代码生成和优化。  
链接中，分为静态链接和动态链接，本文主要是静态链接。

# C++关键字

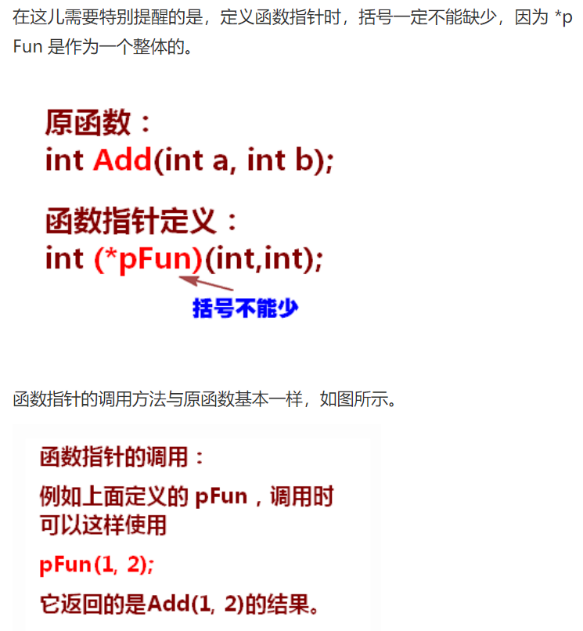
基本的**数据类型**关键字：void, int, char, float, double, bool  
类型**修饰关键**字：long, short, singed, unsigned   
布尔型字面值：true, false  
非常重要的**变量声明**修饰符：const, inline  
**存储类别**关键字：auto(**普通局部栈变量，是自动存储**), static, extern(外部变量声明，是指这是一个已在别的地方定义过的对象，这里只是对变量的一次重复引用，不会产生新的变量), register CPU的寄存器中，从而加快程序的运行)  
控制结构关键字：for, while, if, else, do  
switch语句关键字：switch, case, default  
路径跳转关键字：break, continue, return, goto  
动态创建变量关键字：new, delete  
长度运算符：sizeof  
复合类型关键字：class, struct, enum, union, typedef  
与**类成员相关**关键字：this, friend, virtual, mutable, explicit, operator   
派生类继承方式：private, protected, public  
模板：template, typename  
命名空间：namespace, using  
异常处理：catch, throw, try,  
各种操作符的替代名：and, and\_eq, bitand, bitor, compl, not, not\_eq, or, or\_eq, xor, xor\_eq  
其他不常用的：asm, export, typeid, volatile

# 函数指针

int func(int x); /\* 声明一个函数 \*/

int (\*f) (int x); /\* 声明一个函数指针 \*/

f=func; /\* 将func函数的首地址赋给指针f \*/



回调函数数组：

Void (\*af[])(int\*,int) af是个装有函数指针的数组，一定要括号括起来。





# 友元函数

friend class 类名;

其中：friend和class是关键字，类名必须是程序中的一个已定义过的类。

friend 类型 函数名(形式参数);

放在类的公有和私有都行。

通过友元，一个不同函数或另一个类中的成员函数可以访问类中的私有成员和保护成员。c++中的友元为封装隐藏这堵不透明的墙开了一个小孔，外界可以通过这个小孔窥视内部的秘密。友元提供了不同类的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制。

友元的正确使用能提高程序的运行效率，但同时也破坏了类的封装性和数据的隐藏性，导致程序可维护性变差。

# ★虚函数重点介绍

Virtual void fun();

基类的关键字virtual不能省略，派生类省略virtual编译器可以识别

1. 多态和范式编程需求
2. 运行时决议，而不是编译时决定
3. （实现原理）虚函数指针和虚函数表；拥有虚函数的类的所有对象都会因为虚函数产生额外的开销，并且也会在一定程度上降低程序速度。每个类的实例化对象都会有虚表指针并排列在对象地址首部，按照一定顺序组织成表状结构，称为虚表。
4. （继承原则）同名覆盖；保留基类不同的；
5. （表的特点）针对类而言。一个类公用一张虚表，每个实例对象都有一个虚指针指向虚表，虚表内放置各个虚函数的地址。虚表种保存了基类和子类的虚表，极大节约了时间

Class Base{

Public:

Virtual void func() const { Cout<<”base!”<<endl;}; ①

}

Class derived{

Public:

Virtual void func(){ Cout<<” derived!”<<endl;} ; ②

}

Void show(Base& b)

{

b.func();

}

Base base;

Derived derived;

Int main()

{

show(base);

show(derived);

base.func();

derived.func();  
}

输出结果是：

Base， base，base，derived

1. ②不是同名，因为一个有类型修饰，一个没有

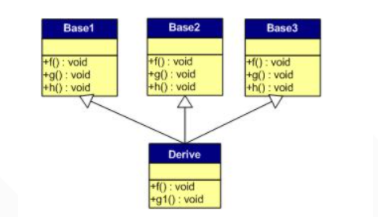
Show向上转换（）；传入是base，调用的func就是bese形式的func，显然子类没有同名的，所以第三个打印出base（这个时候就是重载而不是多态）

注意：虚函数表多态体现一定是完全一样的虚函数

**多态实现的过程（虚表和虚函数指针的工作过程**）

子类的成员函数被放到了第一个父类的表中。（所谓的第一个父类是按照声明顺序来判断的），这样做就是为了解决不同的父类类型的指针指向同一个子类实例，而能够调用到实际的函数。

继承关系：



虚表：



虚表的建立阶段是在编译器，动态绑定方式

**对象的首地址就存放着虚表指针。**

从内存分布上来看，含有**虚函数的类实例化一个对象**

**虚表和类定义放在数据段（存放全局和静态变量的）**

**每个对象都有一个指针指向虚表**

**同名覆盖，虚表内容修改时运行时候根据具体实例修改的，也就是虚表是编译时候生产（由虚函数的类就会生成一个虚表）**

**多态是运行时绑定虚表中函数指针从而实现多态和运行时决定。**

# 虚函数的缺点

虚函数的缺点；虚函数的**多重继承、菱形继承**等问题

为什么构造函数不能使用虚函数？

构造函数调用过程？

# 为什么析构函数必须是虚函数？C++默认的析构函数不是虚函数？

一个基类指针指向派生类，删除该基类指针的时候，虚函数可以调用派生类的析构，然后调用基类的析构函数

类析构顺序：1）派生类本身的析构函数；2）对象成员析构函数；3）基类析构函数。

C++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表指针，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，而是只有当需要当作父类时，设置为虚函数

# 请你说明一下函数指针

（定义）指向函数的指针变量。每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。有了指向函数的指针变量后，可用该指针变量调用函数。

（用途）调用函数和做函数的参数，比如回调函数（把函数指针作为参数传入某个函数，什么用？根据参数改变灵活调用函数）。

**类会有一个虚表指针去指向虚函数表的首地址**

# 请你来说一下fork函数

（定义）创建一个当前进程映像一样的进程可以通过fork（）函数调用

（使用方法）#include <sys/types.h>

#include <unistd.h>

Pid\_t fork(void);

(函数用法讲解) 成功调用fork( )会创建一个新的进程，在子进程中成功fork（）调用会返回0；在父进程种fork（）返回紫禁城的pid。如果出现错误返回一个负值。

常见fork用法：创建一个新进程，然后使用exec（）载入二进制映像，替换当前进程的映像。

# C++析构函数的作用

**（功能**）当对象结束其生命周期，如对象所在的函数已调用完毕时，系统会自动执行析构函数。如果用户没有编写析构函数，编译系统会自动生成一个缺省的析构函数，自己定义了编译器也会为我们合成）；许多类种没有显式的析构函数；

**（内存泄漏处理）**如果一个类种有指针，且使用的过程中动态申请了内存，最好显式构造析构函数在销毁类之前释放掉申请的内存空间，避免内存泄漏。

（**派生列的析构顺序**）派生类自身的析构函数—>对象成员的析构函数🡪基类析构函数

（表现形式）析构函数名与类名相同，只是在函数名前加一个位取反符~。不能带任何参数，也没有返回值。智能有一个析构函数，不能重载。

# 静态函数和虚函数的区别

静态函数在编译的时候就已经确定运行时机；虚函数在运行的时候动态邦定哪个。虚函数因为用了虚表机制，调用的时候会增加一次内存开销。

什么叫做动态绑定: 当基类指针指向派生类的时候，实际上既可以获得派生类的虚表指针，通过虚表指针，基类既可以去调用派生类中的成员函数。如此不同的派生类的虚函数不同，调用的时候实现的效果也就不同，也就符合了多态的定义，面对同一消息的不同应答。

# 创建对象过程

（分配内存）对于全局对象，静态对象以及分配在栈区域内的对象，对它们的内存分配是在编译阶段完成；

分配在堆区域内的对象，他们的分配是在程序运行阶段完成的。

（初始化成员变量）包括了初始化和赋值连个部分。初始化通过实现构造函数的初始列表来实现。赋值通过构造函数的实现体（构造还能输种{}内部分）来实现。

构造函数实现了对象的初始化和赋值两个过程：对象的初始化是通过初始化列表来完成，而对象的赋值则是通过构造函数。

（调用构造函数）构造函数的重载；构造函数可以使用默认参数；构造函数的初始化列表（如果成员变量是const或者是引用类习惯，必须使用初始化列表初始化）

# 什么情况下必须使用初始化列表来初始化成员变量而不能使用构造函数赋值实现？

struct foo

{

string name ;

int id ;

foo(string s, int i):name(s), id(i){} ; // 初始化列表

};

struct Test2

{

Test1 test1 ;

Test2(Test1 &t1):test1(t1){}

}

struct foo

{

int i ;

int j ;

foo(int x):i(x), j(i){}; // ok, 先初始化i，后初始化j

};

类成员变量是const/引用类型必须使用初始化列表。原因：

1. const不能在类声明种初始化该成员。因为对象话没有被创建
2. 引用类型

# new和delete是如何实现的，new 与 malloc的异同处

delete会调用对象的析构函数,和new对应。free只会释放内存，new调用构造函数。malloc与free是C++/C语言的标准库函数，new/delete是C++的运算符。它们都可用于申请动态内存和释放内存。对于非内部数据类型的对象而言，光用maloc/free无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于malloc/free是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于malloc/free。因此C++语言需要一个能完成动态内存分配和初始化工作的运算符**new，**以及一个能完成清理与释放内存工作的运算符**delete。**注意**new/delete**不是库函数。

# 堆和栈的区别

管理方式不同。栈由操作系统自动分配释放，无需我们手动控制；堆的申请和释放工作由程序员控制，容易产生内存泄漏；

空间大小不同。每个进程拥有的栈的大小要远远小于堆的大小。

生长方向不同。堆向高地址生长，栈向低地址生长。

分配方式不同。堆都是动态分配的，栈有2种分配方式：静态分配和动态分配。静态分配是由操作系统完成的，比如局部变量的分配。动态分配由alloca函数进行分配。

|  |  |
| --- | --- |
| 栈 | 堆 |
| 存储内容 | 局部变量 | 变量 |
| 作用域 | 函数作用域、语句块作用域 | 函数作用域、语句块作用域 |
| 编译期间大小是否确定 | 是 | 否 |
| 大小 | 1MB | 4GB |
| 内存分配方式 | 地址由高向低减少 | 地址由低向高增加 |
| 内容是否可以修改 | 是 | 是 |

# 请你说一说重载和覆盖

重载是两个函数名相同，但是参数列表不同（个数，类型），返回值类型没有要求，在同一作用域中

重写：子类继承了父类，父类中的函数是虚函数，在子类中重新定义了虚函数，这种情况是重写。

# 请你说一说strcpy和strlen

Char\* strcpy(char\* dest,cont char\* src)

从src逐字节拷贝到dest，直到遇到’\0’结束，因为没有指定长度，可能会导致拷贝越界，造成缓冲区溢出漏洞，安全版本是strncpy

Strlen函数是计算字符串长度的函数，返回从开始到’\0’之间的字符个数

# 请你说一说你理解的虚函数和多态

多态的实现主要分为静态多态和动态多态，静态多态主要是重载，在编译的时候确定；

动态多态是用虚函数机制实现的，在运行期间动态绑定。

**虚函数的实现**：在有虚函数的类中，类的最开始部分是一个虚函数表的指针，这个指针指向一个虚函数表，表中放了虚函数的地址，实际的虚函数在代码段(.text)中。当子类继承了父类的时候也会继承其虚函数表，当子类重写父类中虚函数时候，会将其继承到的虚函数表中的地址替换为重新写的函数地址。使用了虚函数，会增加访问内存开销，降低效率。

**const char \* arr = "123";**

字符串保留在常量区，const本来是修饰arr指向的值不能通过arr修改，但字符串本来就在常量区，本来就不能改变，所以加不加const效果都一样。

**char \* brr = "123";**

字符串123保存在常量区，a同样不能通过brr去修改123；会编译错误

**const char crr[] = "123";**

“123“本来在栈上的，但是编译器可能会做优化，将其放在常量区

**char drr[] = "123"说明区别**

字符串123保存在栈区，可以通过drr去修改

说明：“123“是 const char\*类型指针，不能用来初始化char \* 但是可以强制转换成char

“123”永远都是保存在常量去

要么指针指向常量区123“”

要么数组复制“123”到栈区

请你来说一下C++里是怎样定义常量的？常量存放在内存的哪个位置？

Const修饰的就是常量，常量定义必须初始化，对于局部对象，常量存放在栈区，对于全局对象，常量存放在全局/静态存储区。对于字面值常量，常量存放在常量存储区。

# 请你来回答一下const修饰成员的目的是什么？

Const修饰的成员函数调用不会堆对象做出任何更改，实际上，如果确认不会对对象做更改，就应该为函数加上const限定，这样无论const对象还是非const对象都能调用该函数。

**如果同时定义了两个函数，一个带const，一个不带，会有问题吗？**

不会，这相当于函数的重载

**请你来说一说隐式类型转换**

对于内置类型的低精度的变量给高精度变量赋值会发生隐式类型转换；

其次，对于只存在单个参数的构造函数的对象构造来说，函数调用可以直接使用该参数传入，编译器会自动调用其构造函数生成临时对象。

**说说你了解的类型转换**

**reinterpret\_cast**：可以用于任意类型的指针之间的转换，对转换的结果不做任何保证。  
**dynamic\_cast**：这种其实也是不被推荐使用的，更多使用static\_cast，dynamic本身只能用于存在虚函数的父子关系的强制类型转换，对于指针，转换失败则返回nullptr，对于引用，转换失败会抛出异常。  
**const\_cast**：对于未定义const版本的成员函数，我们通常需要使用const\_cast来去除const引用对象的const，完成函数调用。另外一种使用方式，结合static\_cast，可以在非const版本的成员函数内添加const，调用完const版本的成员函数后，再使用const\_cast去除const限定。

**static\_cast**：完成基础数据类型；同一个继承体系中类型的转换；任意类型与空指针类型void\* 之间的转换。向上转换是安全的，向下不安全。

**请你来说一说C++函数栈空间的最大值**

默认是1M，不过可以调整

**请你来说一说extern“C”**

C++调用C函数需要extern C，因为C语言没有函数重载。

**extern "C"{** //因为cpp文件默认定义了该宏),则采用C语言方式进行编译

#include"moduleA.h"

#endif

… //其他代码

#ifdef \_\_cplusplus

}

extern "C"的主要作用就是为了能够正确实现C++代码调用其他C语言代码。加上extern "C"后，会指示编译器这部分代码按C语言（而不是C++）的方式进行编译。由于C++支持函数重载，因此编译器编译函数的过程中会将函数的参数类型也加到编译后的代码中，而不仅仅是函数名；而C语言并不支持函数重载，因此编译C语言代码的函数时不会带上函数的参数类型，一般只包括函数名。

**请你回答一下new/delete与malloc/free的区别是什么**

首先，new/delete是C++的关键字，而malloc/free是C语言的库函数，后者使用必须指明申请内存空间的大小，对于类类型的对象，后者不会调用构造函数和析构函数。

**请你说说你了解的RTTI（**通过运行时类型识别Run Time Type Identification**）**

运行时类型检查，在C++层面主要体现在dynamic\_cast和typeid,VS中虚函数表的-1位置存放了指向type\_info的指针。对于存在虚函数的类型，typeid和dynamic\_cast都会去查询type\_info

**请你说说虚函数表具体是怎样实现运行时多态的?**

子类若重写父类虚函数，虚函数表中，该函数的地址会被替换，对于存在虚函数的类的对象，在VS中，对象的对象模型的头部存放指向虚函数表的指针，通过该机制实现多态

**请你说说C语言是怎么进行函数调用的？**

每一个函数调用都会分配函数栈，在栈内进行函数执行过程。调用前，先把返回地址压栈，然后把当前函数的esp（堆栈寄存器）指针压栈。

**请你说说C语言参数压栈顺序**

从右到左

**请你说说C++如何处理返回值？**

生成一个临时变量，把它的引用作为函数参数传入函数内

**请你回答一下C++中拷贝赋值函数的形参能否进行值传递**

不能。如果是这种情况下，调用拷贝构造函数的时候，首先要将实参传递给形参，这个传递的时候又要调用拷贝构造函数。。如此循环，无法完成拷贝，栈也会满

**请你回答一下malloc与new区别**

Malloc开辟内存的区间是在自由存储区，new在堆

       void \*malloc(size\_t size);  
       void free(void \*ptr);

Malloc需要给定申请内存的大小，返回指针需要强转

New会调用构造函数，不用指定内存大小，返回指针不用强转。

Malloc是C的库函数，new是C++的关键词



这个内存分配位置错了

**请你说一说select？？？**

select在使用前，先将需要监控的描述符对应的bit位置1，然后将其传给select,当有任何一个事件发生时，select将会返回所有的描述符，需要在应用程序自己遍历去检查哪个描述符上有事件发生，效率很低，并且其不断在内核态和用户态进行描述符的拷贝，开销很大

**请你说一说static关键字**

1. 加了static关键字的全局变量只能在本文件中使用。例如在a.c中定义了static int a=10;那么在b.c中用extern int a是拿不到a的值得，a的作用域只在a.c中。  
   2.static定义的静态局部变量分配在数据段上，普通的局部变量分配在栈上，会因为函数栈帧的释放而被释放掉。  
   3. 对一个类中成员变量和成员函数来说，加了static关键字，则此变量/函数就没有了this指针了，必须通过类名才能访问

# 容器和算法

**请你来说一下map和set有什么区别，分别又是怎么实现的？**

都是关联容器，底层实现是RB——TREE。所有操作行为都是转调RB的（左旋右旋）的操作行为。

区别：map键和值，可以修改键对应的值，但是不能修改键，因为修改了键就会造成RB树的结构发生变化，当前迭代器就不知道是指向什么位置。可以以键作为索引（即支持下标操作）。如果键不存咋，则插入一个具有该关键码和mappped\_tye类型默认值的元素值到map中。如果非要删除map中对应的值，map.erase(it++)

Set不支持下标操作，而且无法删除/修改键，set将接待器改成const

**请你来介绍一下STL的allocator**

STL标准模板库

STL的代码从广义上讲分为三类：algorithm（算法）、container（容器）和iterator（[迭代器](https://baike.baidu.com/item/%E8%BF%AD%E4%BB%A3%E5%99%A8/3803342)），几乎所有的代码都采用了[模板类](https://baike.baidu.com/item/%E6%A8%A1%E6%9D%BF%E7%B1%BB/6768650)和模板函数的方式，这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。

Allocator是STL分配器用于封装STL让其内存管理上的内存细节。在C++中，其内存配置和释放如下：

New分为两个阶段：①调用：：operator new配置内存；②调用构造函数构造对象内容

Delete运算分为连个阶段：①调用对象的析构函数 ②调用:: operator delete释放内存

为了精密分工STL allocator将两个阶段操作区分开：内存配置alloc::allocate（）负责，对象构造又::construct()负责，对象析构由::destroy()负责，delete释放内存

为了提高内存的管理效率，减少申请小内存造成的内存碎片问题，SGI STL

**请你来说一说STL迭代器删除元素**

1. vector，deque来说，使用erase（iterator）后，后面的每个元素的迭代器都会失效，但是后边每个元素都会往前移动一个位置，但是erase会返回下一个有效的迭代器；
2. map,set来说，使用erase（iterator）后，当前元素的迭代器失效，内部是红黑树解雇，删除当前元素，不会影响到下一个元素的迭代器，所以在调用erase之前，记录下一个迭代器即可。
3. 对于list来说，它使用了不连续分配内存，并且它的erase方法也会返回下一个有效的iterator

**请你说一说STL中MAP数据存放形式**

红黑树。Unordered\_map底层结构是哈希表

**请你讲讲STL有什么基本组成**

STL主要由：以下几部分组成：  
容器、迭代器、仿函数、算法、分配器、配接器  
他们之间的关系：

1. 分配器给容器分配存储空间
2. 算法通过迭代器获取容器中的内容
3. 仿函数可以协助算法完成各种操作
4. 配接器用来套接适配仿函数

**请你说说STL中map与Multimap**

Map：所有元素是pair，同时拥有键-值

底层是红黑树

所有元素都会根据元素的键值自动排序。不允许键值重复

适用于：有序键值对不重复映射

Multimap

多重映射。Multimap的所有元素都是pair，同时哦那个有键-实值。

Pair第一元素被视为键值，第二元素被视为实值。

允许重复

允许重复的原理？  
相同key可以出现在不同的节点上

**请你说一说vector和list的区别，应用，越详细越好**

Vector 底层实现是数组；两倍容量增增长；访问性能O(1)

插入：在后面插入，空间够很快，不够需要内存申请和释放，以及堆之前的数据拷贝

中间插入：空间够内存拷贝；不够，需要申请内存释放内存，对之前数据进行拷贝

在最后面删除：很快

中间删除：拷贝

适用场景：经常随机访问，且不经常堆非结尾节点进行插入删除

List

动态链表，在堆上分配空间。每插入一个元素都会分配空间，每删除一个元素都释放空间

底层：双向链表

性能：

随机访问性能差，智能快速访问头尾节点。

插入：很快O(1)

删除：O(1)

区别：

1. vector底层实现是数组；list是双向链表（原理）
2. vector支持随机访问，list不支持
3. vector是顺序内存，list不是
4. vector在中间节点插入或者删除会进行内存拷贝，list不会
5. vector随机访问性能好，插入删除性能差；list随机访问性能差，插入访问性能好

应用：

Vector适用随机访问，list适用高效的插入和删除

**●请你来说一下STL中迭代器的作用，有指针为何还要迭代器**

Iterator（迭代器）模式又称Cursor（游标）模式，用于提供一种方法顺序访问一个聚合对象中各个元素

**迭代器和指针的区别**

迭代器不是指针，是类模板，表现的像指针。他只是模拟了指针的一些功能，通过重载了指针的一些操作符，->、\*、++、--等。迭代器封装了指针，是一个“可遍历STL（ Standard Template Library）容器内全部或部分元素”的对象， 本质是封装了原生指针，是指针概念的一种提升（lift），提供了比指针更高级的行为，相当于一种智能指针，他可以根据不同类型的数据结构来实现不同的++，--等操作。

迭代器返回的是对象引用而不是对象的值，所以cout只能输出迭代器使用\*取值后的值而不能直接输出其自身。

**迭代器产生原因**

Iterator类的访问方式就是把不同集合类的访问逻辑抽象出来，使得不用暴露集合内部的结构而达到循环遍历集合的效果。

**请你说一说STL迭代器是怎么删除元素的呢**

**n个整数的无序数组，找到每个元素后面比它大的第一个数，要求时间复杂度为O(N)**

单调栈，从后往前，维护递减栈

该无序数组是v,大小为n

Stack<int> stk;

For(int i=n-1;i>=0;i--)

{

While(stk.size()&&v[i]>=stk.top()) stk.pop();

If(stk.size()) Res[i]=stk.top();

Else res[i]=-1;

Stk.push(v[i]);

}

Return res;

每个元素后面比它小的第一个元素：从后往前，维护递增栈

每个元素前面比它大的第一个元素，从前往后维护递减栈

**请你来说一下C++中类成员的访问权限**

成员访问限定符：public、protected、private个关键字来控制成员变量和成员函数的访问权限

类内部都可以访问；

基类成员对派生类的可见性：所有继承方式的派生类都能访问基类的公有和保护成员，基类的私有成员不可见

基类成员对派生对象的可见性：

public继承的的派生类，其对象可以访问基类公有成员

Protect继承的派生类，基类所有成员不可见

Private继承的派生类，基类所有成员不可见

基类成员对派生的的派生类的可见性：

Protect继承：公有成员和保护成员是可见的，而私有成员是不可见的。

Private继承: 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（派生类的派生类就不可以再访问基类的成员了），这就是跟保护继承的区别，只是用于实现。

**请你来说一下C++中struct和class的区别**

在C++中，可以用struct和class定义类，都可以继承。区别在于：structural的默认继承权限和默认访问权限是public，而class的默认继承权限和默认访问权限是private。

另外，class还可以定义模板类形参，比如template <class T, int i>。

**请你回答一下C++类内可以定义引用数据成员吗？**

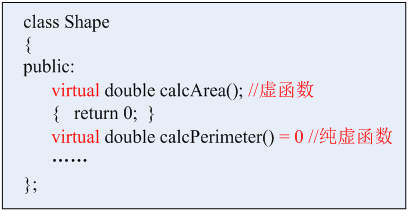
可以，必须通过构造函数函数初始化列表初始化。

引用和const常量都是通过初始化列表进行初始化，构造函数进行赋值

引用和const只能在构造函数被赋值，其他情况不能被改变

**友元函数详细说明一下？**

**抽象类和纯虚函数**



含有纯虚函数的类叫做抽象类

普通虚函数虚表中是有地址的，纯虚函数是没有地址

纯虚函数是没法实例化的，只有子了把纯虚函数完全实现才能实例化

**请你回答一下什么是右值引用，跟左值又有什么区别？**

int &var = 10;

会编译报错，因为10无法进行取地址操作

const int &var = 10;

使用常引用来引用常量数字10，因为此刻内存上产生了临时变量保存了10，这个临时变量是可以进行取地址操作的。

以上都是左值引用。

**左值引用和右值引用的区别**

* **可以取地址的，有名字的，非临时的就是左值；**
* **不能取地址的，没有名字的，临时的就是右值；**

**右值不能被引用也不能被赋值，不可改变**

右值引用：

右值引用的存在并不是为了取代左值引用，而是充分利用右值(特别是临时对象)的构造来减少对象构造和析构操作以达到提高效率的目的。右值引用用来绑定到右值，绑定到右值以后本来会被销毁的右值的生存期会延长至与绑定到它的右值引用的生存期。

int &&var = 10;

带右值引用参数的拷贝构造和赋值重载函数，又叫移动构造函数和移动赋值函数，这里的移动指的是把临时量的资源移动给了当前对象，临时对象就不持有资源，为nullptr了，实际上没有进行任何的数据移动，没发生任何的内存开辟和数据拷贝

**请你来说一下一个C++源文件从文本到可执行文件经历的过程？**

对于C++源文件，从文本到可执行文件一般需要四个过程：

预处理阶段：对源代码文件中文件包含关系（头文件）、预编译语句（宏定义）进行分析和替换，生成预编译文件。

编译阶段：将经过预处理后的预编译文件转换成特定汇编代码，生成汇编文件

汇编阶段：将编译阶段生成的汇编文件转化成机器码，生成可重定位目标文件

链接阶段：将多个目标文件及所需要的库连接成最终的可执行目标文件

**请你来回答一下include头文件的顺序以及双引号””和尖括号<>的区别？**

双引号和尖括号的区别：编译器预处理阶段查找头文件的路径不一样。

使用双引号包含的头文件，查找头文件路径顺序：当前头文件目录；编译器设置的头文件路径；系统比那辆指定的头文件路径

<>: 编译器设置的头文件路径

系统变量CPLUSE\_INCLUDE\_PATH/C\_INCLUDE\_PATH

**请你回答一下malloc的原理，另外brk系统调用和mmap系统调用的作用分别是什么？**

Malloc函数用于动态分配内存。为了减少内存碎片和系统调用的开销，malloc其采用内存池的方式，先申请大块内存作为堆区，然后将堆区分为多个内存块，以块作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空闲块。Malloc采用隐式链表结构将堆区分成连续的、大小不一的块，包含已分配块和未分配块；同时malloc采用显示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。

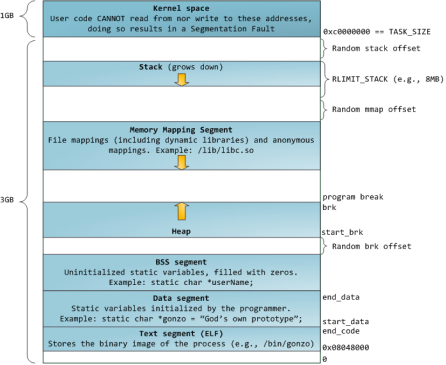
当进行内存分配时，Malloc会通过隐式链表遍历所有的空闲块，选择满足要求的块进行分配；当进行内存合并时，malloc采用边界标记法，根据每个块的前后块是否已经分配来决定是否进行块合并。

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brk在堆区中分配；而当申请内存大于128K时，会使用系统函数mmap在映射区分配。

Brk和mmap是linux进程分配的两种方式

**这两种方式分配的都是虚拟内存，没有分配物理内存**。**在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系。**在标准C库中，提供了malloc/free函数分配释放内存，这两个函数底层是由brk，mmap，munmap这些系统调用实现的。

**请你来说一下C++/C的内存分配**



32bitCPU可寻址4G线性空间，每个进程都有各自独立的4G逻辑地址。3-4G是内核空间；0-3G是用户态空间

静态区域：**文本代码段**（只读存储区（字符串常量）和文本区（程序的机器代码））

**Data数据**区（存储已初始化的全局变量和静态变量）；

**bss**（未初始化的全局变量和静态变量，程序运行main会统一清0）；

堆： malloc（内部是brk和mmp负责快中断）和new

映射区：存储动态链接库等文件映射、申请大内存（malloc时调用mmap会将物理地址映射成虚拟地址）

栈：使用栈空间存储函数的返回地址、参数、局部变量、返回值，从高地址向低地址增长。在创建进程时会有一个最大栈大小，Linux可以通过ulimit命令指定

**请你回答一下如何判断内存泄漏？**

存泄漏通常是由于调用了malloc/new等内存申请的操作，但是缺少了对应的free/delete。为了判断内存是否泄露，我们一方面可以使用linux环境下的内存泄漏检查工具Valgrind,另一方面我们在写代码时可以添加内存申请和释放的统计功能，统计当前申请和释放的内存是否一致，以此来判断内存是否泄露。

**请你来说一下什么时候会发生段错误**

段错误通常发生在访问非法内存地址的时候，具体来说分为以下几种情况：

使用野指针

试图修改字符串常量的内容

**请你来回答一下什么是memory leak，也就是内存泄漏**

内存泄漏(memory leak)是指由于疏忽或错误造成了程序未能释放掉不再使用的内存的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。

内存泄漏的分类：

1. 堆内存泄漏 （Heap leak）。对内存指的是程序运行中根据需要分配通过malloc,realloc new等从堆中分配的一块内存，再是完成后必须通过调用对应的 free或者delete 删掉。如果程序的设计的错误导致这部分内存没有被释放，那么此后这块内存将不会被使用，就会产生Heap Leak.

2. 系统资源泄露（Resource Leak）。主要指程序使用系统分配的资源比如 Bitmap,handle ,SOCKET等没有使用相应的函数释放掉，导致系统资源的浪费，严重可导致系统效能降低，系统运行不稳定。

3. 没有将基类的析构函数定义为虚函数。当基类指针指向子类对象时，如果基类的析构函数不是virtual，那么子类的析构函数将不会被调用，子类的资源没有正确是释放，因此造成内存泄露。

**请你来回答一下new和malloc的区别**

1、new分配内存按照数据类型进行分配，malloc分配内存按照指定的大小分配；

2、new返回的是指定对象的指针，而malloc返回的是void\*，因此malloc的返回值一般都需要进行类型转化。

3、new不仅分配一段内存，而且会调用构造函数，malloc不会。

4、new分配的内存要用delete销毁，malloc要用free来销毁；delete销毁的时候会调用对象的析构函数，而free则不会。

5、new是一个操作符可以重载，malloc是一个库函数。

6、malloc分配的内存不够的时候，可以用realloc扩容。扩容的原理？new没用这样操作。

7、new如果分配失败了会抛出bad\_malloc的异常，而malloc失败了会返回NULL。

8、申请数组时： new[]一次分配所有内存，多次调用构造函数，搭配使用delete[]，delete[]多次调用析构函数，销毁数组中的每个对象。而malloc则只能sizeof(int) \* n

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

new会先调用operator new函数，申请足够的内存（通常底层使用malloc实现）。然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete先调用析构函数，然后调用operator delete函数释放内存（通常底层使用free实现）。

malloc/free是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。

**？？【linux】请你来说一下共享内存相关api**

Linux允许不同进程访问同一个逻辑内存，提供了一组API，头文件在sys/shm.h中。

1）新建共享内存shmget

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

key：共享内存键值，可以理解为共享内存的唯一性标记。

size：共享内存大小

shmflag：创建进程和其他进程的读写权限标识。

返回值：相应的共享内存标识符，失败返回-1

2）连接共享内存到当前进程的地址空间shmat

void \*shmat(int shm\_id,const void \*shm\_addr,int shmflg);

shm\_id：共享内存标识符

shm\_addr：指定共享内存连接到当前进程的地址，通常为0，表示由系统来选择。

shmflg：标志位

返回值：指向共享内存第一个字节的指针，失败返回-1

3）当前进程分离共享内存shmdt

int shmdt(const void \*shmaddr);

4）控制共享内存shmctl

和信号量的semctl函数类似，控制共享内存

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

shm\_id：共享内存标识符

command: 有三个值

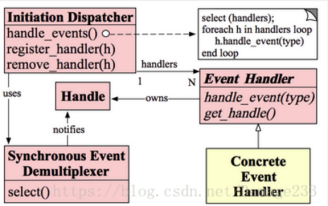
IPC\_STAT:获取共享内存的状态，把共享内存的shmid\_ds结构复制到buf中。

IPC\_SET:设置共享内存的状态，把buf复制到共享内存的shmid\_ds结构。

IPC\_RMID:删除共享内存

buf：共享内存管理结构体。

**？？【网络编程模型一种事件处理机制】请你来说一下reactor模型组成**  
Reactor则是一种事件驱动的机制，这在C#，Qt等程序中都有所体现，即应用程序提供接口注册到Reactor上，相应的事件发生时，Reactor自动调用这个接口，这个接口也被称为“回调函数”。（QT信号和槽机制）



reactor模型要求主线程只负责监听文件描述上是否有事件发生，有的话就立即将该事件通知工作线程。

**？？请自己设计一下如何采用单线程的方式处理高并发**

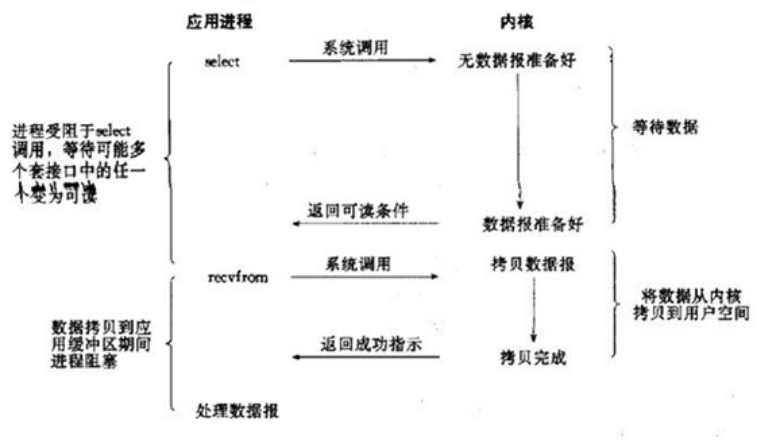
在单线程模型中，可以采用I/O复用来提高单线程处理多个请求的能力，然后再采用事件驱动模型，基于异步回调来处理事件来

事件驱动模型有哪些？

什么叫做异步回调？

**？？请你说一说C++ STL 的内存优化**

**请你说说select，epoll的区别，原理，性能，限制都说一说**



阻塞IO和IO复用最大区别就是复用IO使用了select，使得一个线程能够管理多个套接字

Select/poll/epoll会轮询套接字，某个socket有数据到达就会通知用户进程。

IO复用和阻塞IO起始没有太大的不同，实际上，还更差一些。因为系统调用涉及到select和recfrom，而阻塞IO的系统调用只涉及到recvfrom。Select的优势在于可以处理多个connection。

使用场合高并发场合。如果处理的连接数不是很高使用select/epoll的服务器不一定比使用多线程+IO阻塞性能更好，因为延迟可能更大，毕竟是轮询问

Select/epoll的优势在于处理更多请求而不是对单个连接处理更快

在IO复用模型中，实际上每个socket一般都设置成非阻塞，一直可以被查询；只是线程process被select阻塞了。

# Select详解

Select是最初解决IO阻塞问题的方法，使用fd\_set结构体告诉内核监听多个文件的描述符，该结构体被陈伟描述符集。由数组来维持哪些描述符被置位了。对结构体的操作封装在三个宏定义李。通过轮询查找是否由描述符要被处理，

存在问题：内置数组的形式使得文件数量受限与FD\_SIZE；

每次调用seclect前都要重新初始化描述符集，将fd从用户态拷贝到内核太。

论询文件描述符很多，效率低下

# Poll详解

通过一个可变长度的数组解决select文件描述符受限问题。数组中元素是结构体，该结构体保存描述符的信息，没增加一个文件描述符就向数组中加入一个结构体。结构体只需拷贝一次到内核态。Poll解决了select重复初始化问题。但是轮询效率低没得到解决。

# Epoll详解

解决了轮询的问题。Epoll采用只返回状态变化的操作符。

Epoll对文件描述符的操作由两种模式：LT（level trigger）和ET(edge trigger) LT是默认工作模式

ET是高速工作方式，只支持非阻塞套接字，在这种模式下，当描述符从未就绪编程就绪，内核通过epoll通知用户态。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了。（once only）

ET模式在很大程度上减少了epoll事件被重复触发的次数，因此效率要比LT模式高。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。

Epoll中LT和ET工作模式区别

3、LT模式与ET模式的区别如下：

LT模式：当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用epoll\_wait时，会再次响应应用程序并通知此事件。  
ET模式：当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用epoll\_wait时，不会再次响应应用程序并通知此事件。

RAII

资源分配即初始化。C++标准保证任何情况下，已构造的对象最终会销毁，即它的析构函数最终会被调用。简单的说，RAII 的做法是使用一个对象，在其构造时获取资源，在对象生命期控制对资源的访问使之始终保持有效，最后在对象析构的时候释放资源。

# C++中的锁

## 互斥锁

头文件：<pthread.h>

类型：pthread\_mutex\_t，

函数：  
pthread\_mutex\_init(pthread\_mutex\_t \* mutex, const phtread\_mutexattr\_t \* mutexattr);  
动态方式创建锁，相当于new动态创建一个对象  
pthread\_mutex\_destory(pthread\_mutex\_t \*mutex)  
释放互斥锁，相当于delete  
pthread\_mutex\_lock(pthread\_mutex\_t \*mutex)//以阻塞方式运行的。如果之前mutex被加锁了，那么程序会阻塞在这里。  
pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex)  
int pthread\_mutex\_trylock(pthread\_mutex\_t \* mutex);//会尝试对mutex加锁。如果mutex之前已经被锁定，返回非0,；如果mutex没有被锁定，则函数返回并锁定mutex

## 条件锁

常见的在线程池中，起初没有任务时任务队列为空，此时线程池中的线程因为“任务队列为空”这个条件处于阻塞状态。一旦有任务进来，就会以信号量的方式唤醒一个线程来处理这个任务。这个过程中就使用到了条件变量pthread\_cond\_t。

头文件：<pthread.h>

类型：pthread\_cond\_t

函数：pthread\_cond\_init(pthread\_cond\_t \* condtion, const phtread\_condattr\_t \* condattr);//对条件变量进行动态初始化，相当于new创建对象  
pthread\_cond\_destory(pthread\_cond\_t \* condition);//释放动态申请的条件变量，相当于delete释放对象  
pthread\_cond\_t condition = PTHREAD\_COND\_INITIALIZER;//静态初始化条件变量  
pthread\_cond\_wait(pthread\_cond\_t \* cond, pthread\_mutex\_t \* mutex);//该函数以阻塞方式执行。如果某个线程中的程序执行了该函数，那么这个线程就会以阻塞方式等待，直到收到pthread\_cond\_signal或者pthread\_cond\_broadcast函数发来的信号而被唤醒。  
注意：pthread\_cond\_wait函数的语义相当于：首先解锁互斥锁，然后以阻塞方式等待条件变量的信号，收到信号后又会对互斥锁加锁。

## 读写锁

读写锁初始化：  
int pthread\_rwlock\_init(pthread\_rwlock\_t \* rwlock,  
const pthread\_rwlockattr\_t \* attr);  
该函数第一个参数为读写锁指针，第二个参数为读写锁属性指针。函数按读写锁属性对读写锁进行初始化。  
加读锁：  
int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*rwlock);  
该函数参数为读写锁指针。函数用于对读写锁加读锁。  
加写锁：  
int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \*rwlock);  
该函数参数为读写锁指针。函数用于对读写锁加写锁。  
释放读写锁：  
int pthread\_rwlock\_unlock(pthread\_rwlock\_t \*rwlock);  
该函数参数为读写锁指针。函数用于释放读写锁，包括读锁与写锁。  
销毁读写锁：  
int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock);  
该函数参数为读写锁指针。函数用于销毁读写锁。

# C++11有哪些特性

auto关键字：编译器可以根据初始值自动推导出类型。但是不能用于函数传参以及数组类型的推导

nullptr关键字：nullptr是一种特殊类型的字面值，它可以被转换成任意其它的指针类型；而NULL一般被宏定义为0，在遇到重载时可能会出现问题。

智能指针：C++11新增了std::shared\_ptr、std::weak\_ptr等类型的智能指针，用于解决内存管理的问题。

初始化列表：使用初始化列表来对类进行初始化

右值引用：基于右值引用可以实现移动语义和完美转发，消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率

？？atomic原子操作用于多线程资源互斥操作

**??请你详细介绍一下C++11中的可变参数模板、右值引用和lambda这几个新特性。**

Lamada [](){}

**请你说说C++如何处理内存泄漏**

使用varglind，mtrace检测

# B+树和B-树

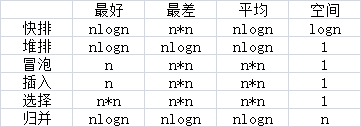
数据库引擎

# QT特性

<https://blog.csdn.net/u013295518/article/details/78775800>

# ★选择题





堆排序实现原理

选择排序

main()

{

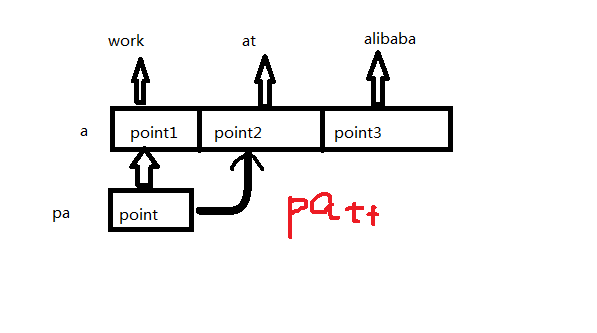
    char\*a[]={"work","at","alibaba"}; //“work”是char\* 元素a是指向第一个元素位置

    char\*\*pa=a;

    pa++;

    printf("%s",\*pa);

}



输出at

# 在TCP/IP建立连接过程中,客户端和服务器端的状态转移说法错误的是?

经历TIME\_WAIT状态是在拆除阶段

TCP建立连接时，首先客户端和服务器处于close状态。然后客户端发送SYN同步位，此时客户端处于SYN-SEND状态，服务器处于lISTEN状态，当服务器收到SYN以后，向客户端发送同步位SYN和确认码ACK，然后服务器变为SYN-RCVD，客户端收到服务器发来的SYN和ACK后，客户端的状态变成ESTABLISHED(已建立连接)，客户端再向服务器发送ACK确认码，服务器接收到以后也变成ESTABLISHED。然后服务器客户端开始数据传输

注意状态和发送内容：服务器状态从close->Listen--（收到同位码；向客户端发送同位码syn和确认码ack）->syn\_recv-（收到客户端的确认码）-->establish 客户端从cloe---（向服务器发送syn）->syn\_send-（收到服务器的同位码）->establish

三次发送和确认

# 假设在n进制下,下面的等式成立,n值是() 240\*12=2880

因为10进制相等的两个数转化为任意进制都会相等，所以答案是多个

N为任意值，能出现8说明是大于等于9的任意值

# 关于linux的I/O复用接口select和epoll,下列说法错误的是()

select调用时会进行线性遍历,epoll采用回调函数机制,不需要线性遍历√

select的最大连接数为FD\_SETSIZE √

select较适合于有大量并发连接,且活跃链接较多的场景 × 根本不适合大并发连接，由于上面一条原因

epoll较适用于有大量并发连接，但活跃连接不多的场景√

epoll的效率不随FD数目增加而线性下降√

epoll通过共享存储实现内核和用户的数据交互√ 只和活跃的socket有关

select 和 epoll效率差异的原因：select采用轮询方式处理连接，epoll是触发式处理连接。  
  
服务器的特点是经常维护着大量连接，但其中某一时刻读写的操作符数量却不多。epoll先通过epoll\_ctl注册一个描述符到内核中，并一直维护着而不像poll每次操作都将所有要监控的描述符传递给内核；在描述符读写就绪时，通过回掉函数将自己加入就绪队列中，之后epoll\_wait返回该就绪队列。也就是说，epoll基本不做无用的操作，时间复杂度仅与活跃的客户端数有关，而不会随着描述符数目的增加而下降

# 下面哪个不是线性表?

循环链表

队列

栈

关联数组 ❌

空字符串数组

双向链表

# 32位：指针类型占4个字节 64位：指针类型占8个字节

结构体的总大小,也就是**sizeof**的结果,.必须是其内部最大成员的"最宽基本类型成员"的整数倍.不足的要补齐.

# ？？？字符串部分匹配算法KMP

朴素匹配算法 时间复杂度O（(N-P+1)\*P）

KMP匹配算法 时间复杂度为O（N+P）