前言

基础知识部分都已整理，数据结构开始看！！！

参考链接

参考链接 https://www.cnblogs.com/nancymake/p/6516933.html

语言基础

基础知识

1、volatile关键字的作用，使用实例有哪些？（重点）

- 访问寄存器比访问内存单元要快,编译器会优化减少内存的读取，可能会读脏数据。声明变量为volatile，编译器不再对访问该变量的代码优化，仍然从内存读取，使访问稳定。

  > 总结：volatile关键字影响编译器编译的结果，用volatile声明的变量表示该变量随时可能发生变化，与该变量有关的运算不再编译优化，以免出错。

- 使用实例如下：
  多线程应用中被几个任务共享的变量（线程1刚处理过共享变量a，寄存器里也有a的值，此时线程2改变了内存a的值，线程1再使用a的值会直接从寄存器拿，导致取的值不对，所以需要使用volatile），就比如有个是否收到app连接的标志位，已连接上就给APP回数据，如果收发是两个线程的话，这个标志位就有必要

设置为volatile。

1、中断服务程序中修改的供其它程序检测的变量需要加volatile；

2、多任务环境下各任务间共享的标志应该加volatile；

3、存储器映射的硬件寄存器通常也要加volatile说明，因为每次对它的读写都可能由不同意义；

2、一个参数既可以是const还可以是volatile吗？解释为什么。

- 可以。一个例子是只读的状态寄存器。它是volatile因为它可能被意想不到地改变。它是const因为程序不应该试图去修改它。

3、一个指针可以是volatile 吗？解释为什么。

- 可以。尽管这并不很常见。一个例子当中断服务子程序修该一个指向一个buffer的指针时。

4、说说const的用法（重点）

- 在定义的时候必须进行初始化
- const修饰全局变量、局部变量
- const修饰指针，int * const a

  > 很好理解，正常定义为int * a，const 加在a前即表示指针不可修改

- const修饰指针指向的对象, const int * a

  > 相当于const int b，修饰的b值不可修改

- 定义为const的形参，即在函数内部是不能被修改的
- 类的成员函数可以被声明为常成员函数，不能修改类的成员变量

  > const修饰成员函数，说明该函数不应该修改非静态成员，但是这并不是十分可靠的，指针所指的非成员对象值可能会被改变

- const修饰类的成员变量时不能在声明中初始化，必须在构造函数列表中初始化

5、const如何做到只读？

- 这些在编译期间完成，对于内置类型，如int， 编译器可能使用常数直接替换掉对此变量的引用。而对于结构体不一定。

6、static的用法（三个明显的作用一定要答出来）

1. 在函数体（修饰局部变量），一个被声明为静态的变量在这一函数被调用过程中维持其值不变。

2. 在模块内（但在函数体外，修饰全局变量），一个被声明为静态的变量可以被模块内所用函数访问，但不能被模块外其它函数访问。它是一个**本地的全局变量**。
3. 在模块内（修饰全局函数），一个被声明为静态的函数只可被这一模块内的其它函数调用。那就是，这个函数被限制在声明它的模块的本地范围内使用。（限制他的作用域只能在本文件之内）
4. 类内的static成员变量属于整个类所拥有，不能在类内进行定义，只能在类的作用域内进行定义
5. 类内的static成员函数属于整个类所拥有，不能包含this指针，只能调用static成员函数

7、static全局变量与普通的全局变量有什么区别?static局部变量和普通局部变量有什么区别?static函数与普通函数有什么区别?

- static全局变量与普通的全局变量有什么区别：static全局变量只初使化一次，防止在其他文件单元中被引用;
- static局部变量和普通局部变量有什么区别：static局部变量只被初始化一次，下一次依据上一次结果值（多次调用一个函数，第一次static int a = 5，以后调用时直接都赋值为5）；
- static函数与普通函数有什么区别：static函数在内存中只有一份，普通函数在每个被调用中维持一份拷贝。（在类里static函数和static也都是这样的）

8、指针和引用的区别？

1. 引用是直接访问，指针是间接访问。
2. 引用是变量的别名，本身不单独分配自己的内存空间，而指针有自己的内存空间
3. 引用绑定内存空间（必须赋初值），是一个变量别名不能更改绑定（必须从一而终），可以改变对象的值。
4. 引用的创建不会调用类的拷贝构造函数

> 总的来说：引用既具有指针的效率，又具有变量使用的方便性和直观性

9、extern关键字作用?

- 声明一个外部变量。
  //file1.cpp定义并初始化和一个常量，该常量能被其他文件访问
  extern const int bufferSize = 5;
  //file1.h头文件
  extern const int bufferSize; //与file1.cpp中定义的是同一个

- extern c 作用
  告诉编译器该段代码以C语言进行编译。

10、关于静态内存分配和动态内存分配的区别及过程？

1. 静态内存分配是在编译时完成的，不占用CPU资源；动态分配内存运行时完成，分配与释放需要占用CPU资源；
2. 静态内存分配是在栈上分配的，动态内存是堆上分配的；
3. 动态内存分配需要指针或引用数据类型的支持，而静态内存分配不需要；
4. 静态内存分配是按计划分配，在编译前确定内存块的大小，动态内存分配运行时按需分配。
5. 静态分配内存是把内存的控制权交给了编译器，动态内存把内存的控制权交给了程序员；
6. 静态分配内存的运行效率要比动态分配内存的效率要高，因为动态内存分配与释放需要额外的开销；动态内存管理水平严重依赖于程序员的水平，处理不当容易造成内存泄漏。

11、头文件中的 ifndef/define/endif 干什么用

预处理，防止头文件被重复使用

```
#ifndef _TEST_H
#define _TEST_H
... //test.h的代码部分
#endif
```

分析：当第一次包含test.h时，由于没有定义_TEST_H，条件为真，这样就会包含（执行）#ifndef _TEST_H和#endif之间的代码.
当第二次包含test.h时前面一次已经定义了_TEST_H，条件为假，#ifndef _TEST_H和#endif之间的代码也就不会再次被包含，这样就避免了重定义了。主要用于防止重复定义宏和重复包含头文件

- 另一种防止头文件使用方法

```
#pragma once
... //test.h的代码部分
```

12、#define的应用

- 宏定义求两个元素的最小是

```
#define MIN(A,B) （（A） <= (B) ? (A) : (B))
```

- 分别设置和清除一个整数的第三位

```
 #define BIT3 (0x1<<3)
static int a;
void set_bit3(void){
    a |= BIT3;
}
void clear_bit3(void){
    a &= ~BIT3;
}
```

- 用预处理指令#define 声明一个常数，用以表明1年中有多少秒

```
#define SECONDS_PER_YEAR (60 * 60 * 24 * 365)UL
```

- 求两个数的乘积和商数，该作用由宏定义来实现

```
#define product(a,b) ((a)*(b))
#define divide(a,b) ((a)/(b))
```

13、预处理器标识#error的目的是什么？

抛出错误提示，标识外部宏是否被定义（从错误提示得到）。

```
#ifdef XXX
...
#error "XXX has been defined"
#else
#endif
```

这样,如果编译时出现错误,输出了XXX has been defined,表明宏XXX已经被定义了。

14、死循环的写法

```
while(1){}
for(;;){}
Loop: ... goto Loop;
```

15、用变量a给出下面的定义

一个有10个指针的数组，该指针指向一个函数，该函数有一个整型参数并返回一个整型数

```
int (*a[10])(int);
```

15、用struct关键字与class关键定义类以及继承的区别

- 定义类差别
  struct关键字也可以实现类，用class和struct关键字定义类的唯一差别在于默认访问级别：默认情况下，struct成员的访问级别为public，而class成员的为private。语法使用也相同，直接将class改为struct即可。
- 继承差别
  使用class保留字的派生类默认具有private继承，而用struct保留字定义的类某人具有public继承。其它则没有任何区别。

> 主要点就两个：默认的访问级别和默认的继承级别 class都是private

16、派生类与虚函数概述

1. 派生类继承的函数不能定义为虚函数。虚函数是希望派生类重新定义。如果派生类没有重新定义某个虚函数，则在调用的时候会使用基类中定义的版本。
2. 派生类中函数的声明必须与基类中定义的方式完全匹配。
3. 基类中声明为虚函数，则派生类也为虚函数。

17、虚函数与纯虚函数区别

1、虚函数在子类里面也可以不重载的；但纯虚必须在子类去实现
2、带纯虚函数的类叫虚基类也叫抽象类，这种基类不能直接生成对象，只能被继承，重写虚函数后才能使用，运行时动态绑定！

18、深拷贝与浅拷贝

```
 浅拷贝：
```

```
char ori[]=“hello”; char *copy=ori;
深拷贝：
char ori[]="hello"; char *copy=new char[]; copy=ori;
```

浅拷贝只是对指针的拷贝，拷贝后两个指针指向同一个内存空间，深拷贝不但对指针进行拷贝，而且对指针指向的内容进行拷贝，经深拷贝后的指针是指向两个不同地址的指针。

- 浅拷贝可能出现的问题：

1. 浅拷贝只是拷贝了指针，使得两个指针指向同一个地址，这样在对象块结束，调用函数析构的时，会造成同一份资源析构2次，即delete同一块内存2次，造成程序崩溃。
2. 浅拷贝使得两个指针都指向同一块内存，任何一方的变动都会影响到另一方。
3. 同一个空间，第二次释放失败，导致无法操作该空间，造成内存泄漏。

19、stl各容器的实现原理（必考）

1. Vector顺序容器，是一个动态数组，支持随机插入、删除、查找等操作，在内存中是一块连续的空间。在原有空间不够情况下自动分配空间，增加为原来的两倍。vector随机存取效率高，但是在vector插入元素，需要移动的数目多，效率低下。

   > 注：vector动态增加大小时是以原大小的两倍另外配置一块较大的空间，然后将原内容拷贝过来，然后才开始在原内容之后构造新元素，并释放原空间。因此，对vector空间重新配置，指向原vector的所有迭代器就都失效了。

2. Map关联容器，以键值对的形式进行存储，方便进行查找。关键词起到索引的作用，值则表示与索引相关联的数据。红黑树的结构实现，插入删除等操作都在O(logn)时间内完成。
3. Set是关联容器，set每个元素只包含一个关键字。set支持高效的关键字检查是否在set中。set也是以红黑树的结构实现，支持高效插入、删除等操作。

20、STL有7种主要容器

vector,list,deque,map,multimap,set,multiset

21、C++特点是什么，多态实现机制？（面试问过）多态作用？两个必要条件？

C++中多态机制主要体现在两个方面，一个是函数的重载，一个是接口的重写。接口多态指的是“一个接口多种形态”。每一个对象内部都有一个虚表指针，该虚表指针被

初始化为本类的虚表。所以在程序中，不管你的对象类型如何转换，但该对象内部的虚表指针是固定的，所以呢，才能实现动态的对象函数调用，这就是C++多态性实现的原理。
多态的基础是继承，需要虚函数的支持，简单的多态是很简单的。子类继承父类大部分的资源，不能继承的有构造函数，析构函数，拷贝构造函数，operator=函数，友元函数等等

- 作用：

1. 隐藏实现细节，代码能够模块化；
2. 接口重用：为了类在继承和派生的时候正确调用。

- 必要条件：

1. 一个基类的指针或者引用指向派生类的对象；
2. 虚函数

22、多重继承有什么问题？怎样消除多重继承中的二义性？

1. 增加程序的复杂度，使程序的编写和维护比较困难，容易出错；
2. 继承类和基类的同名函数产生了二义性，同名函数不知道调用基类还是继承类，C++中使用虚函数解决这个问题
3. 继承过程中可能会继承一些不必要的数据，对于多级继承，可能会产生数据很长

> 可以使用成员限定符和虚函数解决多重继承中函数的二义性问题。

23、什么叫静态关联，什么叫动态关联？
多态中，静态关联是程序在编译阶段就能确定实际执行动作，程序运行才能确定叫动态关联。

24、什么叫智能指针?常用的智能指针有哪些？智能指针的实现？
智能指针是一个存储指向动态分配（堆）对象指针的类，构造函数传入普通指针，析构函数释放指针。栈上分配，函数或程序结束自动释放，防止内存泄露。使用引用计数器，类与指向的对象相关联，引用计数跟踪该类有多少个对象共享同一指针。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，增加引用计数；对一个对象进行赋值时，减少引用计数，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，当引用计数减至0，则删除基础对象。

- std::auto_ptr，不支持复制（拷贝构造函数）和赋值（operator =），编译不会提

示出错。

- C++11引入的unique_ptr， 也不支持复制和赋值，但比auto_ptr好，直接赋值会编译出错。
- C++11或boost的shared_ptr，基于引用计数的智能指针。可随意赋值，直到内存的引用计数为0的时候这个内存会被释放。还有Weak_ptr

> 为了更好的避免内存泄漏。智能指针是一个类，当超出了类的作用域，类将会自动调用析构函数，析构函数会自动释放资源。

25、枚举与define 宏的区别

1. define 宏常量是在预编译阶段进行简单替换。枚举常量则是在编译的时候确定其值。
2. 可以调试枚举常量，但是不能调试宏常量。
3. 枚举可以一次定义大量相关的常量，而#define 宏一次只能定义一个。

26、介绍一下函数的重载

重载是在不同类型上作不同运算而又用同样的名字的函数。重载函数至少在参数个数，参数类型， 或参数顺序上有所不同。

27、派生新类的过程要经历三个步骤

1.吸收基类成员 2.改造基类成员 3.添加新成员

> 首先调用基类的构造函数， 其次调用的是成员类的构造函数， 最后才是自身的构造函数；
> 子类包含基类的信息，所以首先要调用基类的的构造函数，按照成员类的定义顺序申请空间，调用过成员类的构造函数后在调用自身的构造函数。

28、面向对象的三个基本特征，并简单叙述之?

1. 封装：将客观事物抽象成类，是实现面向对象程序设计的第一步，每个类对自身的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏，就是将数据或函数等集合在类中。（封装的意义在于保护或者防止代码（数据）被我们无意中破坏）

   > 封装可以隐藏实现细节，使得代码模块化

2. 继承：主要实现重用代码，节省开发时间。子类可以继承父类的一些东西
3. 多态：允许一个基类的指针或引用指向一个派生类对象。（同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果）

29、多态性体现都有哪些？动态绑定怎么实现？

多态性是一个接口,多种实现，是面向对象的核心。

- 编译时多态性：通过重载函数实现。
- 运行时多态性：通过虚函数实现,结合动态绑定。

30、虚函数，虚函数表里面内存如何分配？

编译时若基类中有虚函数，编译器为该的类创建一个一维数组的虚表，存放是每个虚函数的地址。基类和派生类都包含虚函数时，这两个类都建立一个虚表。构造函数中进行虚表的创建和虚表指针的初始化。在构造子类对象时，要先调用父类的构造函数，初始化父类对象的虚表指针，该虚表指针指向父类的虚表。执行子类的构造函数时，子类对象的虚表指针被初始化，指向自身的虚表。每一个类都有虚表。虚表可以继承，如果子类没有重写虚函数，那么子类虚表中仍然会有该函数的地址，只不过这个地址指向的是基类的虚函数实现。派生类的虚表中虚函数地址的排列顺序和基类的虚表中虚函数地址排列顺序相同。当用一个指针/引用调用一个函数的时候，被调用的函数是取决于这个指针/引用的类型。即如果这个指针/引用是基类对象的指针/引用就调用基类的方法；如果指针/引用是派生类对象的指针/引用就调用派生类的方法，当然如果派生类中没有此方法，就会向上到基类里面去寻找相应的方法。这些调用在编译阶段就确定了。当涉及到多态性的时候，采用了虚函数和动态绑定，此时的调用就不会在编译时候确定而是在运行时确定。不在单独考虑指针/引用的类型而是看指针/引用的对象的类型来判断函数的调用，根据对象中虚指针指向的虚表中的函数的地址来确定调用哪个函数。

31、纯虚函数如何定义？含有纯虚函数的类称为什么？为什么析构函数要定义成虚函数？

- 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。纯虚函数是虚函数再加上= 0。virtual void fun ()=0。
- 含有纯虚函数的类称为抽象类在很多情况下，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。同时含有纯虚拟函数的类称为抽象类，它不能生成对象。
- 如果析构函数不是虚函数，那么释放内存时候，编译器会使用静态联编，认为p就是一个基类指针，调用基类析构函数，这样子类对象的内存没有释放，造成内存泄漏。定义成虚函数以后，就会动态联编，先调用子类析构函数，再基类。

32、C++中哪些不能是虚函数？

1. 普通函数只能重载，不能被重写，因此编译器会在编译时绑定函数。

2. 构造函数是知道全部信息才能创建对象，然而虚函数允许只知道部分信息。
3. 内联函数在编译时被展开，虚函数在运行时才能动态绑定函数。
4. 友元函数 因为不可以被继承。
5. 静态成员函数 只有一个实体，不能被继承。父类和子类共有。

33、C++四种类型转换

static_cast, dynamic_cast, const_cast, reinterpret_cast

1. const_cast用于将const变量转为非const
2. static_cast用的最多，对于各种隐式转换，非const转const，void*转指针等, static_cast能用于多态想上转化，如果向下转能成功但是不安全，结果未知；
3. dynamic_cast用于动态类型转换。只能用于含有虚函数的类，用于类层次间的向上和向下转化。只能转指针或引用。向下转化时，如果是非法的对于指针返回NULL，对于引用抛异常。要深入了解内部转换的原理。
4. reinterpret_cast几乎什么都可以转，比如将int转指针，可能会出问题，尽量少用；
5. 为什么不使用C的强制转换？C的强制转换表面上看起来功能强大什么都能转，但是转化不够明确，不能进行错误检查，容易出错。

34、如何判断一段程序是由C 编译程序还是由C++编译程序编译的？

```
#ifdef __cplusplus
cout<<"C++";
#else
cout<<"c";
#endif
```

35、指针函数与函数指针的区别？

- 指针函数：指带指针的函数，即本质是一个函数，函数返回类型是某一类型的指针。

  > 类型标识符 函数名(参数表)
  > *int* f(x，y);

- 函数指针：指向函数的指针变量，即本质是一个指针变量。

```
    void func(int a){...}
```

```
    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 */
    f(2); //相当于调用func(2)
```

主要的区别是一个是指针变量，一个是函数。在使用是必要要搞清楚才能正确使用

36、return *this和return this有什么区别？

- return *this
  返回的是当前对象的克隆或者本身（若返回类型为A， 则是克隆， 若返回类型为A&(引用)， 则是本身 ）。
- return this
  返回当前对象的地址（指向当前对象的指针）

37、回调函数

回调函数就是一个通过函数指针调用的函数。如果你把函数的指针(地址)作为参数传递给另一个函数，当这个指针被用为调用它所指向的函数时，我们就说这是回调函数。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。

回调函数实现的机制是：

（1）定义一个回调函数；

（2）提供函数实现的一方在初始化的时候，将回调函数的函数指针注册给调用者；

（3）当特定的事件或条件发生的时候，调用者使用函数指针调用回调函数对事件进行处理。

例子：快速排序调用的比较函数，即把函数指针传入快排程序，每次比较两个数大小的时候调用该函数。

38、深入谈谈堆和栈

1).分配和管理方式不同 ：

堆是动态分配的，其空间的分配和释放都由程序员控制。

栈由编译器自动管理。栈有两种分配方式：静态分配和动态分配。静态分配由编译器完成，比如局部变量的分配。动态分配由alloca()函数进行分配，但是栈的动态分配和堆是不同的，它的动态分配是由编译器进行释放，无须手工控制。

2).产生碎片不同

对堆来说，频繁的new/delete或者malloc/free势必会造成内存空间的不连续，造成大量的碎片，使程序效率降低。

对栈而言，则不存在碎片问题，因为栈是先进后出的队列，永远不可能有一个内存块

从栈中间弹出。
3).生长方向不同
堆是向着内存地址增加的方向增长的，从内存的低地址向高地址方向增长。
栈是向着内存地址减小的方向增长，由内存的高地址向低地址方向增长。

39、内存的静态分配和动态分配的区别？
时间不同。静态分配发生在程序编译和连接时。动态分配则发生在程序调入和执行时。
空间不同。堆都是动态分配的，没有静态分配的堆。栈有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。alloca，可以从栈里动态分配内存，不用担心内存泄露问题，当函数返回时，通过alloca申请的内存就会被自动释放掉。

40、拷贝构造函数作用及用途？什么时候需要自定义拷贝构造函数？
一般如果构造函数中存在动态内存分配，则必须定义拷贝构造函数。否则，可能会导致两个对象成员指向同一地址，出现“指针悬挂问题”。

41、100万个32位整数，如何最快找到中位数。能保证每个数是唯一的，如何实现O(N)算法？
1).内存足够时：快排
2).内存不足时：分桶法：化大为小，把所有数划分到各个小区间，把每个数映射到对应的区间里，对每个区间中数的个数进行计数，数一遍各个区间，看看中位数落在哪个区间，若够小，使用基于内存的算法，否则 继续划分

42、C++虚函数是如何实现的？
使用虚函数表。 C++对象使用虚表， 如果是基类的实例，对应位置存放的是基类的函数指针；如果是继承类，对应位置存放的是继承类的函数指针（如果在继承类有实现）。所以 ，当使用基类指针调用对象方法时，也会根据具体的实例，调用到继承类的方法。

43、C++的虚函数有什么作用？
虚函数作用是实现多态，虚函数其实是实现封装，使得使用者不需要关心实现的细节。在很多设计模式中都是这样用法，例如Factory、Bridge、Strategy模式。

44、动态链接库的两种使用方法及特点？
1)．载入时动态链接，模块非常明确调用某个导出函数，使得他们就像本地函数一样。这需要链接时链接那些函数所在DLL的导入库，导入库向系统提供了载入DLL时所需的信息及DLL函数定位。
2)运行时动态链接。

代码

- memcpy函数的实现

```
void *memcpy(void *dest, const void *src, size_t count) {
  char *tmp = dest;
  const char *s = src;
  while (count--)
    *tmp++ = *s++;
  return dest;
}
```

- Strcpy函数实现

```
char *strcpy(char *dst,const char *src) {
 assert(dst != NULL && src != NULL);
 char *ret = dst;
 while((* dst++ = * src++) != '\0') ;
 return ret;
}
```

- strcat函数的实现

```
char *strcat(char *strDes, const char *strSrc){
 assert((strDes != NULL) && (strSrc != NULL));
 char *address = strDes;
 while (*strDes != '\0')
  ++ strDes;
 while ((*strDes ++ = *strSrc ++) != '\0')
     return address;
}
```

- strncat实现

```
char *strncat(char *strDes, const char *strSrc, int count){
 assert((strDes != NULL) && (strSrc != NULL));
 char *address = strDes;
```

```
 while (*strDes != '\0')
  ++ strDes;
 while (count – && *strSrc != '\0' )
  *strDes ++ = *strSrc ++;
 *strDes = '\0';
 return address;
}
```

- strcmp函数实现

```
int strcmp(const char *str1,const char *str2){
    /*不可用while(*str1++==*str2++)来比较，当不相等时仍会执行一次++，return返回的比较值实际上是下一个字符。应将++放到循环体中进行。*/
    while(*str1 == *str2){
        if(*str1 == '\0')
            return0;
        ++str1;
        ++str2;
    }
 return *str1 - *str2;
}

```

- strncmp实现

```
int strncmp(const char *s, const char *t, int count){
    assert((s != NULL) && (t != NULL));
    while (*s && *t && *s == *t && count -) {
        ++ s;
        ++ t;
    }
    return (*s - *t);
}
```

- strlen函数实现

```
int strlen(const char *str){
    assert(str != NULL);
```

```
    int len = 0;
    while (*str ++ != '\0')
        ++ len;
    return len;
}
```

- strpbrk函数实现

```
char * strpbrk(const char * cs,const char * ct){
    const char *sc1,*sc2;
    for( sc1 = cs; *sc1 != '\0'; ++sc1)
        for( sc2 = ct; *sc2 != '\0'; ++sc2)
            if (*sc1 == *sc2)
                return (char *) sc1;
    return NULL;
}
```

- strstr函数实现

```
char *strstr(const char *s1,const char *s2){
  int len2;
  if(!(len2=strlen(s2)))//此种情况下s2不能指向空，否则strlen无法测出长度，这条语句错误
   return(char*)s1;
  for(;*s1;++s1){
      if(*s1==*s2 && strncmp(s1,s2,len2)==0)
      return(char*)s1;
  }
  return NULL;
}
```

- string实现（注意：赋值构造，operator=是关键）

```
class String{
public:
  //普通构造函数
  String(const char *str = NULL);
  //拷贝构造函数
```

```
  String(const String &other);
  //赋值函数
  String & operator=(String &other) ;
  //析构函数
  ~String(void);
private:
  char* m_str;
};

//分别实现以上四个函数
//普通构造函数
String::String(const char* str){
    if(str==NULL) //如果str为NULL，存空字符串{
        m_str = new char[1]; //分配一个字节
        *m_str = '\0'; //赋一个'\0'
}else{
       str = new char[strlen(str) + 1];//分配空间容纳str内容
        strcpy(m_str, str); //复制str到私有成员m_str中
    }
}

//析构函数
String::~String(){
    if(m_str!=NULL) //如果m_str不为NULL，释放堆内存
{
        delete [] m_str;
        m_str = NULL;
    }
}

//拷贝构造函数
String::String(const String &other){
    m_str = new char[strlen(other.m_str)+1]; //分配空间容纳str内容
    strcpy(m_str, other.m_str); //复制other.m_str到私有成员m_str中
}

//赋值函数
String & String::operator=(String &other){
    if(this == &other) //若对象与other是同一个对象，直接返回本
        return *this
```

```
    delete [] m_str; //否则，先释放当前对象堆内存
    m_str = new char[strlen(other.m_str)+1]; //分配空间容纳str内容
    strcpy(m_str, other.m_str); //复制other.m_str到私有成员m_str中
    return *this;
}
```

网络编程

- 多线程和多进程的区别（重点 必须从cpu调度，上下文切换，数据共享，多核cup利用率，资源占用，等等各方面回答，然后有一个问题必须会被问到：哪些东西是一个线程私有的？答案中必须包含寄存器，否则悲催）！
  1）进程数据是分开的:共享复杂，需要用IPC，同步简单；多线程共享进程数据：共享简单，同步复杂
  2）进程创建销毁、切换复杂，速度慢 ；线程创建销毁、切换简单，速度快
  3）进程占用内存多， CPU利用率低；线程占用内存少， CPU利用率高
  4）进程编程简单，调试简单；线程 编程复杂，调试复杂
  5）进程间不会相互影响 ；线程一个线程挂掉将导致整个进程挂掉
  6）进程适应于多核、多机分布；线程适用于多核
  线程所私有的：
  线程id、寄存器的值、栈、线程的优先级和调度策略、线程的私有数据、信号屏蔽字、errno变量、
- 多线程锁的种类有哪些？
  a.互斥锁（mutex）b.递归锁 c.自旋锁 d.读写锁
- 自旋锁和互斥锁的区别？
  当锁被其他线程占用时，其他线程并不是睡眠状态，而是不停的消耗CPU，获取锁；互斥锁则不然，保持睡眠，直到互斥锁被释放激活。
  自旋锁，递归调用容易造成死锁，对长时间才能获得到锁的情况，使用自旋锁容易造成CPU效率低，只有内核可抢占式或SMP情况下才真正需要自旋锁。
- 进程间通信和线程间通信
  1）.管道 2）消息队列 3)共享内存 4)信号量 5)套接字 6)条件变量
- 多线程程序架构，线程数量应该如何设置？
  应尽量和CPU核数相等或者为CPU核数+1的个数
- 什么是原子操作，gcc提供的原子操作原语，使用这些原语如何实现读写锁？
  原子操作是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch。

- 网络编程设计模式，reactor/proactor/半同步半异步模式？
  reactor模式：同步阻塞I/O模式，注册对应读写事件处理器，等待事件发生进而调用事件处理器处理事件。 proactor模式：异步I/O模式。Reactor和Proactor模式的主要区别就是真正的读取和写入操作是有谁来完成的，Reactor中需要应用程序自己读取或者写入数据，Proactor模式中，应用程序不需要进行实际读写过程。
  Reactor是：
  主线程往epoll内核上注册socket读事件，主线程调用epoll_wait等待socket上有数据可读，当socket上有数据可读的时候，主线程把socket可读事件放入请求队列。睡眠在请求队列上的某个工作线程被唤醒，处理客户请求，然后往epoll内核上注册socket写请求事件。主线程调用epoll_wait等待写请求事件，当有事件可写的时候，主线程把socket可写事件放入请求队列。睡眠在请求队列上的工作线程被唤醒，处理客户请求。
  Proactor:
  主线程调用aio_read函数向内核注册socket上的读完成事件，并告诉内核用户读缓冲区的位置，以及读完成后如何通知应用程序，主线程继续处理其他逻辑，当socket上的数据被读入用户缓冲区后，通过信号告知应用程序数据已经可以使用。应用程序预先定义好的信号处理函数选择一个工作线程来处理客户请求。工作线程处理完客户请求之后调用aio_write函数向内核注册socket写完成事件，并告诉内核写缓冲区的位置，以及写完成时如何通知应用程序。主线程处理其他逻辑。当用户缓存区的数据被写入socket之后内核向应用程序发送一个信号，以通知应用程序数据已经发送完毕。应用程序预先定义的数据处理函数就会完成工作。
  半同步半异步模式：
  上层的任务（如：数据库查询，文件传输）使用同步I/O模型，简化了编写并行程序的难度。
  而底层的任务（如网络控制器的中断处理）使用异步I/O模型，提供了执行效率。
- 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清为0.
- connect可能会长时间阻塞，怎么解决?
  1.使用定时器；（最常用也最有效的一种方法）
  2.采用非阻塞模式：设置非阻塞，返回之后用select检测状态。
- keepalive 是什么东西？如何使用？
  keepalive，是在TCP中一个可以检测死连接的机制。
  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。默认二个小时。

- socket什么情况下可读？
  1.socket接收缓冲区中已经接收的数据的字节数大于等于socket接收缓冲区低潮限度的当前值;对这样的socket的读操作不会阻塞,并返回一个大于0的值(准备好读入的数据的字节数).
  2.连接的读一半关闭(即:接收到对方发过来的FIN的TCP连接),并且返回0;
  3.socket收到了对方的connect请求已经完成的连接数为非0.这样的soocket处于可读状态；
  4.异常的情况下socket的读操作将不会阻塞,并且返回一个错误(-1)。
- udp调用connect有什么作用？
  1).因为UDP可以是一对一，多对一，一对多，或者多对多的通信，所以每次调用sendto()/recvfrom()时都必须指定目标IP和端口号。通过调用connect()建立一个端到端的连接，就可以和TCP一样使用send()/recv()传递数据，而不需要每次都指定目标IP和端口号。但是它和TCP不同的是它没有三次握手的过程。
  2).可以通过在已建立连接的UDP套接字上，调用connect()实现指定新的IP地址和端口号以及断开连接。
- socket编程，如果client断电了，服务器如何快速知道？
  使用定时器（适合有数据流动的情况）；
  使用socket选项SO_KEEPALIVE（适合没有数据流动的情况）；
  1）自己编写心跳包程序,简单的说就是自己的程序加入一条线程,定时向对端发送数据包,查看是否有ACK,根据ACK的返回情况来管理连接。此方法比较通用,一般使用业务层心跳处理,灵活可控,但改变了现有的协议;
  2）使用TCP的keepalive机制,UNIX网络编程不推荐使用SO_KEEPALIVE来做心）跳检测。
  keepalive原理:TCP内嵌有心跳包,以服务端为例,当server检测到超过一定时间(/proc/sys/net/ipv4/tcp_keepalive_time 7200 即2小时)没有数据传输,那么会向client端发送一个keepalive packet。

liunx操作系统

- 熟练netstat tcpdump ipcs ipcrm
  netstat:检查网络状态，tcpdump:截获数据包，ipcs:检查共享内存，ipcrm:解除共享内存

- 共享内存段被映射进进程空间之后，存在于进程空间的什么位置？共享内存段最大限制是多少？
  将一块内存映射到两个或者多个进程地址空间。通过指针访问该共享内存区。一般通过mmap将文件映射到进程地址共享区。
  存在于进程数据段，最大限制是0x2000000Byte
- 进程内存空间分布情况
  程序段(Text)、初始化过的数据(Data)、未初始化过的数据(BSS)、栈 (Stack)、堆(Heap)。
- ELF是什么？其大小与程序中全局变量的是否初始化有什么关系（注意未初始化的数据放在bss段）
  可执行连接格式。可以减少重新编程重新编译的代码。
- 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，在可执行文件运行时再装入；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
- 32位系统一个进程最多有多少堆内存
  32位意味着4G的寻址空间，Linux把它分为两部分：最高的1G(虚拟地址从0xC0000000到0xffffffff)用做内核本身，成为“系统空间”，而较低的3G字节（从0x00000000到0xbfffffff）用作各进程的“用户空间”。每个进程可以使用的用户空间是3G。虽然各个进程拥有其自己的3G用户空间，系统空间却由所有的进程共享。从具体进程的角度看，则每个进程都拥有4G的虚拟空间，较低的3G为自己的用户空间，最高的1G为所有进程以及内核共享的系统空间。实际上有人做过测试也就2G左右。
- 写一个c程序辨别系统是64位 or 32位

```
void* number = 0; printf("%d\n",sizeof(&number));
```

输出8就是64位 输出4就是32位的 根据逻辑地址判断的

- 写一个c程序辨别系统是大端or小端字节序

```
union{
short value;
char a[sizeof(short)];
}test;
```

```
test.value= 0x0102;
if((test.a[0] == 1) && (test.a[1] == 2))
    cout << "big"<<endl;
else
    cout << "little" << endl;
```

- 信号：列出常见的信号，信号怎么处理？
  1).进程终止的信号 2).跟踪进程的信号 3).与进程例外事件相关的信号等
  对于信号的处理或者执行相关的操作进行处理或者直接忽略
- 说出你所知道的各类linux系统的各类同步机制（重点），什么是死锁？如何避免死锁（每个技术面试官必问）
  1).原子操作 2).信号量（其实就是互斥锁也就是锁的机制）3).读写信号量（就是读写锁） 4）.自旋锁 5.内核锁 6）.顺序锁
  死锁就是几个进程申请资源，出现了循环等待的情况！
  避免死锁的方法：
  1）.资源是互斥的 2）.不可抢占 3）占有且申请 4）.循环等待
- 如何实现守护进程？
  1）创建子进程，父进程退出
  2）在子进程中创建新会话
  3）改变当前目录为根目
  4）重设文件权限掩码
  5) 关闭文件描述符
  6) 守护进程退出处理
  当用户需要外部停止守护进程运行时，往往会使用 kill命令停止该守护进程。所以，守护进程中需要编码来实现kill发出的signal信号处理，达到进程的正常退出。
- linux的任务调度机制是什么？
  Linux 分实时进程和普通进程，实时进程应该先于普通进程而运行。实时进程：
  1） FIFO(先来先服务调度)
  2） RR（时间片轮转调度）。
  每个进程有两个优先级（动态优先级和实时优先级），实时优先级就是用来衡量实时进程是否值得运行的。 非实时进程有两种优先级，一种是静态优先级，另一种是动态优先级。实时进程又增加了第三种优先级，实时优先级。优先级越高，得到CPU时间的机会也就越大。
- 标准库函数和系统调用的区别？

系统调用：是操作系统为用户态运行的进程和硬件设备(如CPU、磁盘、打印机等)进行交互提供的一组接口，即就是设置在应用程序和硬件设备之间的一个接口层。inux内核是单内核，结构紧凑，执行速度快，各个模块之间是直接调用的关系。linux系统上到下依次是用户进程->linux内核->硬件。其中系统调用接口是位于Linux内核中的，整个linux系统从上到下可以是：用户进程->系统调用接口->linux内核子系统->硬件，也就是说Linux内核包括了系统调用接口和内核子系统两部分；或者从下到上可以是：物理硬件->OS内核->OS服务->应用程序，操作系统起到“承上启下”作用，向下管理物理硬件，向上为操作系服务和应用程序提供接口，这里的接口就是系统调用了。
库函数：把函数放到库里。是把一些常用到的函数编完放到一个lib文件里，供别人用。别人用的时候把它所在的文件名用#include<>加到里面就可以了。一类是c语言标准规定的库函数，一类是编译器特定的库函数。
系统调用是为了方便使用操作系统的接口，而库函数则是为了人们编程的方便。

- 系统如何将一个信号通知到进程？
  内核给进程发送信号，是在进程所在的进程表项的信号域设置对应的信号的位。进程处理信号的时机就是从内核态即将返回用户态度的时候。执行用户自定义的信号处理函数的方法很巧妙。把该函数的地址放在用户栈栈顶，进程从内核返回到用户态的时候，先弹出信号处理函数地址，于是就去执行信号处理函数了，然后再弹出，才是返回进入内核时的状态。
- fork()一子进程程后父进程的全局变量能不能使用？
  fork后子进程将会拥有父进程的几乎一切资源，父子进程的都各自有自己的全局变量。不能通用，不同于线程。对于线程，各个线程共享全局变量。
- 请画出socket通信连接过程
- 请用socket消息队列实现“同步非阻塞”和“异步阻塞”两种模式，并指出两者的差别和优劣
  http://blog.csdn.net/yongchurui/article/details/12780653

网络编程

- TCP头大小，包含字段？三次握手，四次断开描述过程，都有些什么状态。状态变迁图。TCP/IP收发缓冲区（2次）
  头部大小是20字节，包含数据如下：
  三次握手：
  四次释放：
  状态变迁图：
  收发缓冲区：

- 使用udp和tcp进程网络传输，为什么tcp能保证包是发送顺序，而 udp无法保证？
  因为TCP发送的数据包是按序号发送，有确认机制和丢失重传机制，而udp是不可靠的发送机制，发送的对应端口的数据包不是按顺序发送的。
- 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有EPOLLLT和EPOLLET两种触发模式，LT是默认的模式，ET是“高速”模式。LT模式下，只要这个fd还有数据可读，每次 epoll_wait都会返回它的事件，提醒用户程序去操作，而在ET（边缘触发）模式中，它只会提示一次，直到下次再有数据流入之前都不会再提示了，无论fd中是否还有数据可读。所以在ET模式下，read一个fd的时候一定要把它的buffer读光，也就是说一直读到read的返回值小于请求值。
  也就是说在LT模式的情况下一定要确认收发的数据包的buffer是不是足够大如果收发数据包大小大于buffer的大小的时候就可能会出现数据丢失的情况。
- tcp与udp的区别（必问）为什么TCP要叫做数据流？
  1）基于连接与无连接
  2）对系统资源的要求（TCP较多，UDP少）
  3）UDP程序结构较简单
  4）流模式与数据报模式
  5）TCP保证数据正确性，UDP可能丢包，TCP保证数据顺序，UDP不保证
  6）TCP有拥塞控制和流量控制，UDP没有
  TCP提供的是面向连接、可靠的字节流服务。当客户和服务器彼此交换数据前，必须先在双方之间建立一个TCP连接，之后才能传输数据。TCP提供超时重发，丢弃重复数据，检验数据，流量控制等功能，保证数据能从一端传到另一端。
  是一个简单的面向数据报的运输层协议。UDP不提供可靠性，它只是把应用程序传给IP层的数据报发送出去，但是并不能保证它们能到达目的地。由于UDP在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，故而传输速度很快
- 流量控制和拥塞控制的实现机制
  网络拥塞现象是指到达通信子网中某一部分的分组数量过多,使得该部分网络来不及处理,以致引起这部分乃至整个网络性能下降的现象,严重时甚至会导致网络通信业务陷入停顿,即出现死锁现象。拥塞控制是处理网络拥塞现象的一种机制。数据的传送与接收过程当中很可能出现收方来不及接收的情况,这时就需要对发方进行控制,以免数据丢失。
- 滑动窗口的实现机制
  滑动窗口机制，窗口的大小并不是固定的而是根据我们之间的链路的带宽的大

小，这个时候链路是否拥护塞。接受方是否能处理这么多数据了。 滑动窗口协议，是TCP使用的一种流量控制方法。该协议允许发送方在停止并等待确认前可以连续发送多个分组。由于发送方不必每发一个分组就停下来等待确认，因此该协议可以加速数据的传输。

- epoll和select的区别？
  1）select在一个进程中打开的最大fd是有限制的，由FD_SETSIZE设置，默认值是2048。不过 epoll则没有这个限制，内存越大，fd上限越大，1G内存都能达到大约10w左右。
  2）select的轮询机制是系统会去查找每个fd是否数据已准备好，当fd很多的时候，效率当然就直线下降了，epoll采用基于事件的通知方式，一旦某个fd数据就绪时，内核会采用类似callback的回调机制，迅速激活这个文件描述符，高效。
  3）select还是epoll都需要内核把FD消息通知给用户空间，epoll是通过内核于用户空间mmap同一块内存实现的，而select则做了不必要的拷贝
- 网络中，如果客户端突然掉线或者重启，服务器端怎么样才能立刻知道？
  若客户端掉线或者重新启动，服务器端会收到复位信号，每一种tcp/ip得实现不一样，控制机制也不一样。
- TTL是什么？有什么用处，通常那些工具会用到它？ping? traceroute? ifconfig? netstat?
  TTL是Time To Live，每经过一个路由就会被减去一，如果它变成0，包会被丢掉。它的主要目的是防止包在有回路的网络上死转，浪费网络资源。ping和traceroute用到它。
- linux的五种IO模式/异步模式.
  1）同步阻塞I/O
  2）同步非阻塞I/O
  3）同步I/O复用模型
  4） 同步信号驱动I/O
  5） 异步I/O模型
- 请说出http协议的优缺点.
  1.支持客户/服务器模式。2.简单快速：客户向服务器请求服务时，只需传送请求方法和路径，通信速度很快。3.灵活：HTTP允许传输任意类型的数据对象。4.无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。5.无状态：HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，导致每次连接传送的数据量增大。缺点就是不够安全，可以使用https完成使用

- NAT类型，UDP穿透原理。
  1）Full cone NAT （全克隆nat）:一对一NAT一旦一个内部地址（iAddr:port1）映射到外部地址（eAddr:port2）。
  2）Address-Restricted cone NAT（地址受限克隆nat）:任意外部主机（hostAddr:any）都能通过给eAddr:port2发包到达iAddr:port1的前提是：iAddr:port1之前发送过包到hostAddr:any. "any"也就是说端口不受限制
  3). Port-Restricted cone NAT:内部地址（iAddr:port1）映射到外部地址（eAddr:port2），所有发自iAddr:port1的包都经eAddr:port2向外发送。一个外部主机（hostAddr:port3）能够发包到达iAddr:port1的前提是：iAddr:port1之前发送过包到hostAddr:port3.
  4). Symmetric NAT（对称NAT）:同内部IP与port的请求到一个特定目的地的IP地址和端口，映射到一个独特的外部来源的IP地址和端口。同一个内部主机发出一个信息包到不同的目的端，不同的映射使用外部主机收到了一封包从一个内部主机可以送一封包回来
- 大规模连接上来，并发模型怎么设计
  Epoll+线程池（epoll可以采用libevent处理）
- tcp三次握手的，accept发生在三次握手哪个阶段？
  三次握手：C----->SYN K
  S------>ACK K+1 SYN J
  C-------->ACK J+1
  DONE!
  client 的 connect 引起3次握手
  server 在socket， bind， listen后，阻塞在accept，三次握手完成后，accept返回一个fd，
- 流量控制与拥塞控制的区别，节点计算机怎样感知网络拥塞了？
  拥塞控制是把整体看成一个处理对象的，流量控制是对单个的节点。
  感知的手段应该不少，比如在TCP协议里，TCP报文的重传本身就可以作为拥塞的依据。依据这样的原理， 应该可以设计出很多手段。

算法与数据结构

- 给定一个单向链表（长度未知），请遍历一次就找到中间的指针，假设该链表存储在只读存储器，不能被修改
  设置两个指针，一个每次移动两个位置，一个每次移动一个位置，当第一个指针到达尾节点时，第二个指针就达到了中间节点的位置。
  处理链表问题时，”快行指针“是一种很常见的技巧，快行指针指的是同时用两个

指针来迭代访问链表，只不过其中一个比另一个超前一些。快指针往往先行几步，或与慢指针相差固定的步数。

- 将一个数组生成二叉排序树
  排序，选数组中间的一个元素作为根节点，左边的元素构造左子树，右边的节点构造有子树。
- 查找数组中第k大的数字？
  因为快排每次将数组划分为两组加一个枢纽元素，每一趟划分你只需要将k与枢纽元素的下标进行比较，如果比枢纽元素下标大就从右边的子数组中找，如果比枢纽元素下标小从左边的子数组中找，如果一样则就是枢纽元素，找到，如果需要从左边或者右边的子数组中再查找的话，只需要递归一边查找即可，无需像快排一样两边都需要递归，所以复杂度必然降低。
  最差情况如下：假设快排每次都平均划分，但是都不在枢纽元素上找到第k大第一趟快排没找到，时间复杂度为O(n)，第二趟也没找到，时间复杂度为O(n/2)，第k趟找到，时间复杂度为O(n/2k)，所以总的时间复杂度为O(n(1+1/2+....+1/2k))=O(n)，明显比冒泡快，虽然递归深度是一样的，但是每一趟时间复杂度降低。
- 简述一致性hash算法。
  1）首先求memcached服务器（节点）的哈希值，并将其配置到0～232的圆（continuum）。
  2）然后采用同样的方法求出存储数据的键的哈希值，并映射到相同的圆上。
  3）然后从数据映射到的位置开始顺时针查找，将数据保存到找到的第一个服务器上。如果超过232仍然找不到服务器，就会保存到第一台memcached服务器上。
- 描述一种hash table的实现方法
  1） 除法散列法: p ，令 h(k) = k mod p ，这里， p 如果选取的是比较大的素数，效果比较好。而且此法非常容易实现，因此是最常用的方法。最直观的一种，上图使用的就是这种散列法，公式： index = value % 16，求模数其实是通过一个除法运算得到的。
  2） 平方散列法 :求index频繁的操作，而乘法的运算要比除法来得省时。公式：index = (value * value) >> 28 （右移，除以2^28。记法：左移变大，是乘。右移变小，是除）
  3） 数字选择法:如果关键字的位数比较多，超过长整型范围而无法直接运算，可以选择其中数字分布比较均匀的若干位，所组成的新的值作为关键字或者直接作为函数值。
  4） 斐波那契（Fibonacci）散列法:平方散列法的缺点是显而易见的，通过找到一

个理想的乘数index = (value * 2654435769) >> 28
冲突处理：令数组元素个数为 S ，则当 h(k) 已经存储了元素的时候，依次探查 (h(k)+i) mod S，i=1,2,3…… ，直到找到空的存储单元为止（或者从头到尾扫描一圈仍未发现空单元，这就是哈希表已经满了，发生了错误。当然这是可以通过扩大数组范围避免的）。

- hash，任何一个技术面试官必问（例如为什么一般hashtable的桶数会取一个素数？如何有效避免hash结果值的碰撞）
  不选素数的话可能会造成hash出值的范围和原定义的不一致
- 数组和链表的优缺点
  数组，在内存上给出了连续的空间。链表，内存地址上可以是不连续的，每个链表的节点包括原来的内存和下一个节点的信息(单向的一个，双向链表的话，会有两个)。
  数组优于链表的:
  A. 内存空间占用的少。
  B. 数组内的数据可随机访问，但链表不具备随机访问性。
  C. 查找速度快
  链表优于数组的:
  A. 插入与删除的操作方便。
  B. 内存地址的利用率方面链表好。
  C. 方便内存地址扩展。
- 4G的long型整数中找到一个最大的，如何做？
  每次从磁盘上尽量多读一些数到内存区，然后处理完之后再读入一批。减少IO次数，自然能够提高效率。分批读入选取最大数，再对缓存的最大数进行快排。
- 有千万个string在内存怎么高速查找，插入和删除？
  对千万个string做hash，可以实现高速查找，找到了，插入和删除就很方便了。关键是如何做hash，对string做hash，要减少碰撞频率。
- 100亿个数，求最大的1万个数，并说出算法的时间复杂度
  在内存中维护一个大小为10000的最小堆，每次从文件读一个数，与最小堆的堆顶元素比较，若比堆顶元素大，则替换掉堆顶元素，然后调整堆。最后剩下的堆内元素即为最大的1万个数，算法复杂度为O(NlogN)
- 设计一个洗牌的算法，并说出算法的时间复杂度。
  （1）全局洗牌法
  a）首先生成一个数组，大小为54，初始化为1~54
  b）按照索引1到54，逐步对每一张索引牌进行洗牌，首先生成一个余数 value = rand %54，那么我们的索引牌就和这个余数牌进行交换处理

c）等多索引到54结束后，一副牌就洗好了
（2）局部洗牌法：索引牌从1开始，到54结束。这一次索引牌只和剩下还没有洗的牌进行交换， value = index + rand（ ） %（54 - index）
算法复杂度是O(n)

- 各种排序方法
  http://blog.csdn.net/hguisu/article/details/7776068
- 哈希表冲突解决方法？
  - 常见的hash算法如下：
    1、数字分析法 2、平方取中法 3、分段叠加法4、除留余数法 5、伪随机法
  - 解决冲突的方法：
    1、开放地址法
    也叫散列法，主要思想是当出现冲突的时候，以关键字的结果值作为key值输入，再进行处理，依次直到冲突解决
    线性地址再散列法：当冲突发生时，找到一个空的单元或者全表
    二次探测再散列：冲突发生时，在表的左右两侧做跳跃式的探测
    伪随机探测再散列
    2、再哈希法
    同时构造不同的哈希函数
    3、链地址法
    将同样的哈希地址构造成一个同义词的链表
    4、建立公共溢出区
    建立一个基本表和溢出区，凡是和基本元素发生冲突都填入溢出区

系统架构

- 编写高效服务器程序，需要考虑的因素
  性能对服务器程序来说是至关重要的了，毕竟每个客户都期望自己的请求能够快速的得到响应并处理。那么影响服务器性能的首要因素应该是：
  （1）系统的硬件资源，比如说CPU个数，速度，内存大小等。不过由于硬件技术的飞速发展，现代服务器都不缺乏硬件资源。因此，需要考虑的主要问题是如何从“软环境”来提升服务器的性能。
  服务器的”软环境“
  （2）一方面是指系统的软件资源，比如操作系统允许用户打开的最大文件描述符数量
  （3）另一方面指的就是服务器程序本身，即如何从编程的角度来确保服务器的

性能。
主要就要考虑大量并发的处理这涉及到使用进程池或线程池实现高效的并发模式（半同步/半异步和领导者/追随者模式），以及高效的逻辑处理方式--有限状态机内存的规划使用比如使用内存池，以空间换时间，被事先创建好，避免动态分配，减少了服务器对内核的访问频率，数据的复制，服务器程序还应该避免不必要的数据复制，尤其是当数据复制发生在用户空间和内核空间之间时。如果内核可以直接处理从socket或者文件读入的数据，则应用程序就没必要将这些数据从内核缓冲区拷贝到应用程序缓冲区中。这里所谓的“直接处理”，是指应用程序不关心这些数据的具体内容是什么，不需要对它们作任何分析。比如说ftp服务器，当客户请求一个文件时，服务器只需要检测目标文件是否存在，以及是否有权限读取就可以了，不需要知道这个文件的具体内容，这样的话ftp服务器就不需要把目标文件读入应用程序缓冲区然后调用send函数来发送，而是直接使用“零拷贝”函数sendfile直接将其发送给客户端。另外，用户代码空间的数据赋值也应该尽可能的避免复制。当两个工作进程之间需要传递大量的数据时，我们就应该考虑使用共享内存来在他们直接直接共享这些数据，而不是使用管道或者消息队列来传递。上下文切换和锁：并发程序必须考虑上下文的切换问题，即进程切换或线程切换所导致的系统开销。即时I/O密集型服务器也不应该使用过多的工作线程（或工作进程），否则进程间切换将占用大量的CPU时间，服务器真正处理业务逻辑的CPU时间比重就下降了。因此为每个客户连接都创建一个工作线程是不可取的。应该使用某种高效的并发模式。（半同步半异步或者说领导者追随者模式）另一个问题就是共享资源的加锁保护。锁通常被认为是导致服务器效率低下的一个因素，因为由他引入的代码不仅不处理业务逻辑，而且需要访问内核资源，因此如果服务器有更好的解决方案，应该尽量避免使用锁。或者说服务器一定非要使用锁的话，尽量使用细粒度的锁，比如读写锁，当工作线程都只读一块内存区域时，读写锁不会增加系统开销，而只有当需要写时才真正需要锁住这块内存区域。对于高峰和低峰的伸缩处理，适度的缓存。

- QQ飞车新用户注册时，如何判断新注册名字是否已存在？（数量级：几亿)
  可以试下先将用户名通过编码方式转换，如转换64位整型。然后设置N个区间，每个区间为2^64/N的大小。对于新的用户名，先通过2分寻找该用户名属于哪个区间，然后在在这个区间，做一个hash。对于不同的时间复杂度和内存要求可以设置不同N的大小~
- 基础的技术面试之外的职业素养的面试问题
  1.你在工作中犯了个错误，有同事打你小报告，你如何处理？
  a.同事之间应该培养和形成良好的同事关系，就是要互相支持而不是互相拆台，互相学习，互相帮助，共同进步。

b.如果小报告里边的事情都是事实也就是说确实是本人做的不好不对的方面，那么自己应该有则改之，提高自己。如果小报告里边的事情全部不是事实，就是说确实诬陷，那么应该首先坚持日久见人心的态度，持之以恒的把本职工作做好，然后在必要的时候通过适当的方式和领导沟通，相信领导会知道的。

- 你和同事合作完成一个任务，结果任务错过了截止日期，你如何处理？
- 职业规划？
- 项目中遇到的难题，你是如何解决的?
  a.时间 b要求 c.方法