C++ 你想要的C++面经都在这(附答案 | 持续更新)

C++面试题集

C++

1、c/c++内存模型
2、内联函数是什么，是不是只要声明为内联函数就一定会在原地展开
3、编译的四个过程
4、c++和c的区别
5、指针和引用有什么区别
6、什么是内存泄漏，如何防止内存泄漏，如何判断内存泄漏
7、malloc/free和new/delete的区别
8、什么是左值和右值
9、什么是野指针，野指针处理方式
10、函数重载
11、const关键字的作用
12、static关键字的作用
13、内存对齐的原因
14、什么是指针
15、联合体和结构体的区别
16、数组和指针的区别
17、C++中拷贝赋值函数的形式能否进行值传递
18、纯虚函数的作用
19、为什么析构函数必须是虚函数，而C++默认不是虚函数
20、C++函数栈空间的最大值
21、vector和list的区别
22、迭代器和指针的区别
23、说一说STL迭代器是怎么删除元素的
24、C++中的struct和class区别
25、哪些成员变量必须在初始化列表初始化
26、include头文件的“”和<>的区别
27、什么时候会发生段错误
28、栈和堆的区别
29、变量的声明和定义有什么区别
30、sizeof和strlen的区别
31、说说面向对象的三大特性
32、实现一个string类
33、重写、重载和隐藏的区别
34、说说你对虚函数的理解
35、构造函数可以是虚函数吗
36、深浅拷贝的区别及实现一个深拷贝
37、实现一个vector类
38、请你说说C++中智能指针如何防止内存泄漏的
39、请你介绍一下C++中四种智能指针的实现原理
40、请你回答智能指针存在内存泄漏的情况
41、简单实现一个shared_ptr智能指针
42、右值引用的作用
43、说一说C++中四种cast类型转换
44、使用const定义常量比#define好的原因
45、extern和volatile的作用
46、malloc、calloc、realloc的区别

网络

1、TCP如何保证可靠性
2、http和https的区别
3、TCP和UDP的区别及使用场景

4、post和get的区别
5、OSI七层模型和TCP/IP五层模型
6、解释HTTP状态码及说出常见HTTP状态码
7、三次握手及四次挥手
8、cookie和session的区别
9、浏览器输入url回车后，都发生了什么?
10、滑动窗口实现原理
11、拥塞控制讲一讲
12、HTTPS协议加密流程
13、说说HTTP的新特性
14、cookie个session的工作流程

操作系统

1、进程和线程的区别
2、什么是死锁，死锁的条件及如何预防死锁
3、虚拟地址空间了解吗
4、请你说说操作系统中的缺页中断
5、进程间通信方式
6、僵尸进程和孤儿进程
7、并发和并行的理解
8、有了进程为什么还要线程
9、你了解IPC吗
10、fork和vfork的区别
11、请问单核机器上写多线程程序，是否需要加锁
12、线程同步方式，并说出系统调用
13、多线程和多进程的优缺点和各自使用场景
14、操作系统的缺页置换及算法
15、什么是进程，什么是线程
16、什么是重定向
17、软链接和硬链接的区别
18、如何判断大小端
19、用户态和内核态的区别及转换方式
20、说说生产者和消费者线程模型
21、系统调用及使用过哪些
22、说一下你所认识的锁
23、说说乐观锁如何实现的
24、请你介绍一下5种IO模型
25、说说select、poll、epoll的区别
26、线程池的意义及如何实现一个线程池
27、水平触发(LT)和边缘触发(ET)

数据结构

1、快排和归并的联系与区别
2、链表和数组的区别
3、说出你所知道的排序算法及其复杂度
4、topK问题-如何从海量数据取最大的k个数
5、快排非递归
6、堆排序
7、直接插入排序
8、希尔排序
9、选择排序
10、冒泡排序

算法编程题

1、统计一个数的二进制1的个数
2、股票的最大利润

other

1、给一个超过100G大小的log file, log中存着IP地址, 设计算法找到出现次数最多的IP地址?
2、与上题条件相同， 如何找到top K的IP?
3、给定100亿个整数，设计算法找到只出现一次的整数?
4、给两个文件，分别有100亿个整数，我们只有1G内存，如何找到两个文件交集?
5、1个文件有100亿个int，1G内存，设计算法找到出现次数不超过2次的所有整数

C++

1、c/c++内存模型

C++ 内存分为栈、堆、数据段、BSS段、代码段、映射区
栈是存放函数的局部变量和函数参数等，由编译器自动分配和释放。堆是动态分配的内存空间，必须由程序员手动申请和释放。数据段是存储程序中已经初始化的全局变量和静态变量。BSS段是存储未初始化的全局变量和静态变量，以及所有被初始化为0的全局变量和静态变量。代码段是存放代码，函数，以及字符串常量的地方。映射区存储了动态链接库以及调用mmap函数进行的文件映射

2、内联函数是什么，是不是只要声明为内联函数就一定会在原地展开

在函数声明或定义前加inline关键字，则该函数被称为内联函数。c++编译器会直接将调用内联函数的地方直接展开成相对应的指令，没有了函数栈帧的开销，从而提高了程序的运行效率。
不是，当代码很长，或者循环递归不会展开。

3、编译的四个过程

1. **预处理阶段**：引入头文件，宏替换，删除注释…，生成以.i为结尾的预编译文件
2. **编译阶段**：检查语义语法规错误，如果没有错误则将代码解释为汇编汇编，生成以.s为结尾的汇编文件
3. **汇编阶段**： 将汇编代码解释为二进制的cpu指令，生成.o为结尾的可重定向目标文件
4. **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

4、c++和c的区别

1. C是面向过程的语言，而C++是面向对象的语言
2. c++具有封装、继承和多态三种特性
3. C和C++动态管理内存的方法不一样，C是使用malloc/free函数，而C++除此之外还有new/delete关键字
4. C++支持函数重载，而C不支持函数重载

5、指针和引用有什么区别

1. 引用必须在定义时初始化，指针没有要求。
2. 引用在初始化引用一个实体后，不能更改引用实体而指针可以。
3. 没有NULL引用，但是有NULL指针
4. 有多级指针，但是没有多级引用
5. 访问实体时，指针需要解引用，而引用是编译器自行处理
6. 引用比指针用起来相对安全
7. 引用自加自减都是实体值的改变，而指针自加自减是地址的偏移
8. sizeof求的是指针的大小，4个字节或者8个字节；而sizeof求引用变量时求实体变量的大小

6、什么是内存泄漏，如何防止内存泄漏，如何判断内存泄漏

内存泄漏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。

1. 养成良好的编码规范，申请的内存空间记着匹配的去释放内存
2. 使用内存泄漏工具检测
3. 采用RAII思想或者智能指针来管理资源

在linux环境下可以使用内存泄漏检测工具Valgrind，我们写代码是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。

7、malloc/free和new/delete的区别

1. malloc和free是函数，new和delete是操作符
2. malloc的返回值为void*, 在使用时必须强转；而new后跟的是空间的类型，就不需要强转
3. malloc申请空间失败时，返回的是NULL，而new申请空间失败时，会抛异常
4. 申请自定义类型对象时，malloc/free只会开辟空间，不会调用构造函数与析构函数，而new在申请空间后会调用构造函数完成对象的初始化，delete在释放空间前会调用析构函数完成空间中资源的清理

8、什么是左值和右值

一般情况下，左值指的是既能出现在赋值等号左边，也能出现在赋值等号的右边，并且可以进行取地址的变量。右值指的是只能出现在赋值等号右边的值，且不能进行取地址的变量。在C++中右值有常量、匿名变量和将亡值。其他的都是左值

9、什么是野指针，野指针处理方式

野指针就是指向一个已删除的对象或者指向一个未申请并且访问受限的内存区域的指针

1. 定义指针时未初始化
2. 释放指针后未置指针为NULL
3. 指针的操作超越了变量作用域

处理方式:

1. 初始化指针时置指针为NULL
2. 释放指针后置指针为NULL

10、函数重载

函数重载是用来描述同名函数具有相同或者相似功能，在同一作用域中，当两个函数的函数名相同但是参数列表不同(个数，类型)，这两个函数就形成了函数重载

11、const关键字的作用

1、const修饰的局部变量或者全局变量声明时必须初始化，且该变量只能读不能写
2、const修饰指针时，如果const在* 号的左边，则修饰的是指针指向的内容不能变，const在 * 号右边时，则修饰的是指针，表示指针的指向不能发送改变。如果 *号左边和右边都有const，则表示指针指向的内容和指针本身都是不可变的
3、const修饰的成员函数不能修改对象的成员变量
4、const修饰的函数参数在函数体内不能被修改

12、static关键字的作用

1、未初始化的静态变量的值默认为0
2、修饰全局变量或者函数时，则该变量和函数只能在当前文件可见
3、修饰局部变量时，变量的生命周期随程序，只有程序结束，变量的生命周期才结束
4、static修饰的成员函数没有this指针，不能调用非静态成员函数和非静态变量，只能通过类名来访问
5、static修饰的成员变量在整个类中只有一份，被所有对象共享，必须在类外初始化，也是只能通过类名来访问

13、内存对齐的原因

第一个原因是不是所有的硬件平台都可以访问任意地址上的任意数据，某些平台只能在特定的地址上获得特定的数据，否则就会抛异常；第二个原因是使用内存对齐，可以高CPU访问内存的效率
修改默认对齐数： `#pragma pack(n)`

14、什么是指针

指针就是一种变量的类型，定义一个指针变量，这个变量中可以存放一块内存空间的地址，通过这个地址可以访问这块内存空间。

15、联合体和结构体的区别

结构体和联合体都是由多个不同的类型变量组成的，但是联合体所有成员共用一块内存，而结构体每个成员都有各自的内存。给联合体的不同变量赋值, 会覆盖掉其他变量的值，而给结构体的不同变量赋值是互不影响的。

16、数组和指针的区别

1. 数组是保存多个同类型数据的集合；而指针是一个变量，用来保存其他变量的内存地址的
2. sizeof数组是求数组中全部元素占内存空间的大小；而sizeof指针求的是指针的大小，不是4个字节就是8个字节
3. 当使用数组作为函数参数时会退化为指针

17、C++中拷贝赋值函数的形式能否进行值传递

不能。如果是进行值传递，调用拷贝构造函数时，首先要将实参传给形参，这个传递的时候也要调用拷贝构造函数，如此循环会造成无限递归，无法完成拷贝

18、纯虚函数的作用

为了方便使用多态特性，我们会在基类中定义纯虚函数。但在很多情况下，基类本身生成对象是不合情理的。如果将函数定义为纯虚函数，则编译器要求在派生类中必须进行重写来实现多态性。同时含有纯虚函数的类称为抽象类，它不能生成实例化生成对象

19、为什么析构函数必须是虚函数，而C++默认不是虚函数

将父类的析构函数设置为虚函数，可以保证我们使用父类指针指向一个子类对象时，释放父类指针的同时也可以释放子类的空间，防止内存泄漏。而C++默认析构函数不是虚函数，因为虚函数需要额外的虚函数表的虚表指针，占用额外的内存。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。

20、C++函数栈空间的最大值

默认为1M，可以调整

21、vector和list的区别

区别:
1、vector底层是通过数组实现的；list底层是通过链表来实现的
2、vector支持随机访问；list不支持随机访问
3、vector迭代器是原生指针；list迭代器是封装链表结点的一个类
4、vector在插入和删除时可能会导致迭代器失效；list只在删除的时候会导致迭代器的失效
5、vector不容易造成内存碎片，空间利用率高；list容易造成内存碎片，空间利用率低
6、vector在非尾插尾删的时间复杂度都为O(n)，list在任何地方插入和删除的时间复杂度都为O(1)
使用场景:
如果频繁地随机访问，且不关心插入删除效率，就使用vector；具有大量插入和删除操作，而不关心随机访问，就是用list

22、迭代器和指针的区别

迭代器不是指针，而是类模板。它模拟了指针的一些功能，通过重载了指针的一些操作符，封装了指针，提供了比指针更高级的行为

23、说一说STL迭代器是怎么删除元素的

对于vector，deque来说，删除元素后，后边的每个元素的迭代器都会失效，但是后边的每个元素都会向前移动一个位置。返回的是下一个有效的迭代器
对于list来说，它使用了不连续的内存，删除元素后会返回下一个有效的迭代器
对于关联容器map，set来说，删除元素后，当前元素迭代器失效，但是其结构是红黑树，删除当前元素不会影响到下一个元素的迭代器，所以调用erase之前，记录下一个元素的迭代器即可

24、C++中的struct和class区别

都可以定义类，都可以被继承，区别在于struct的默认访问权和默认继承权都是公有的，而class的默认访问权和默认继承权都是私有的

25、哪些成员变量必须在初始化列表初始化

引用、const、没有默认构造函数的自定义成员变量

26、include头文件的“”和<>的区别

对于双引号包含的头文件，查找头文件默认为当前头文件目录下查找。而尖括号查找头文件默认是编译器设置的头文件路径下查找

27、什么时候会发生段错误

非法访问内存地址。例如使用野指针、试图修改字符串常量的内容

28、栈和堆的区别

1. 栈是由高地址向低地址扩展，而堆是由低地址向高地址扩展
2. 堆中的内存需要手动申请和释放，而栈是操作系统自动申请和释放
3. 堆中频繁调用malloc和new，容易产生内存碎片，而栈不会产生内存碎片
4. 堆的分配效率低，而栈的分配效率高

29、变量的声明和定义有什么区别

1. 变量的声明是不会分配地址的，而变量的定义时才会为变量分配地址内存
2. 一个变量可以在多个地方声明，但是只能在一个地方定义

关键字 `extern` 是声明一个变量，表示该变量在其他地方已经定义好了

30、sizeof和strlen的区别

1. sizeof是一个操作符，strlen是库函数
2. sizeof的参数可以是数据的类型，也可以是变量，而strlen的参数只能是以\0结尾的字符串
3. sizeof计算的是数据对象占内存的大小，而strlen计算的是字符串的实际长度

31、说说面向对象的三大特性

面向对象的三大特性为封装、继承和多态
封装：将数据和操作数据的方法进行有机结合，隐藏对象的属性和实现细节，仅对外公开接口来与对象进行交互
继承：继承是面向对象程序设计使代码复用的重要手段，它允许程序员在保持原有类的特性的基础上进行扩展，增加功能，产生新的一个类，被继承的类称为基类，继承基类的类称为派生类
多态：多态分为静态多态和动态多态，静态多态主要是重载，在编译的时候就已经确定了；动态多态主要是通过虚函数实现的，在运行期间动态绑定。例如当父类指针指向一个子类对象时，此时的父类指针调用子类重写父类的虚函数时，会去调用子类重写过的虚函数

32、实现一个string类

```
class String
{
public:
    String(const char* str = "")
    {
        assert(str != nullptr);
        _str = new char[strlen(str) + 1];
        strcpy(_str, str);
    }

    String(const String& s)
        :_str(nullptr)
    {
        String tmp(s._str);
        swap(_str, tmp._str);

    }

    String(String&& s)
        :_str(s._str)
    {
        s._str = nullptr;
    }

    String& operator=(String&& s)
    {
        if (this != &s)
        {
            delete[] _str;
            _str = s._str;

```

```
            s._str = nullptr;
        }
        return *this;
    }

    String& operator=(String s)
    {
        if (this != &s)
        {
            swap(_str, s._str);
        }
        return *this;
    }

    ~String()
    {
        if (_str != nullptr)
        {
            delete[] _str;
            _str = nullptr;
        }
    }
private:
    char* _str;
};
```

33、重写、重载和隐藏的区别

函数重载：同一作用域下同名函数具有不同的参数列表，且不关心返回值
函数隐藏：在父类和子类中，只要函数名相同，不管参数列表和返回值是否相同，就会产生函数隐藏
重写覆盖：在父类和子类中，当父类中存在虚函数，子类中存在函数与父类的虚函数同名，且返回值和参数列表都相同，则就产生重写

34、说说你对虚函数的理解

虚函数最主要的目的就是为了实现类的多态。在有虚函数的类中，类的前4个字节用来保存虚表指针，这个虚表指针指向一个虚表，这个虚表是在数据段中的，表中存放的是虚函数的地址，而实际的虚函数是在代码段中的。当父类指针指向子类对象时，会调用子类重写过的虚函数来实现多态

35、构造函数可以是虚函数吗

构造函数是不能是虚函数的，虚函数地址存放在虚表中的，要找到虚表就必须通过虚表指针找到，但是虚表指针是存在对象中的。如果构造函数是虚的，就需要通过虚表指针来调用，可是对象还没有实例化，就没有虚表指针，就找不到构造函数，也就不能完整对象的实例化。所以构造函数不能是虚函数

36、深浅拷贝的区别及实现一个深拷贝

浅拷贝是按字节拷贝，我们默认不写编译器生成的拷贝构造就是浅拷贝，当我们类中存在资源时，如果调用拷贝构造，此时两个对象都共用了一份资源，各自调用析构时会产生二次释放的问题；**深拷贝**是重新申请一个新的空间分配给新拷贝的对象，两个对象各自使用不同内存空间的资源，调用析构时就不存在二次释放的问题

```
class String
{
public:
    //构造函数
    String(const char* str = "")
    {
        //str不能为空
        assert(str != nullptr);
        //开辟一个与传入的字符串的长度 + 1的空间 +1是保存 \0
        _str = new char[strlen(str) + 1];
        //将传入的字符串内容拷贝到当前类的内容中
        strcpy(_str, str);
    }
    String(const String& s)
        //先给新的对象开辟空间
        :_str(new char[strlen(s._str) + 1])
    {
        //再将拷贝的内容拷贝到新的对象中
        strcpy(_str, s._str);
```

```
    }
private:
    char* _str;
};
```

37、实现一个vector类

```
template<class T>
class Vector
{
public:
    typedef T* iterator;
    typedef const T* const_iterator;
    Vector()
        :_start(nullptr)
        ,_finish(nullptr)
        ,_endOfStorage(nullptr)
    {}
        //n个val的构造函数
    Vector(int n, const T& val = T())
        :_start(new T[n])
        ,_finish(_start +n)
        ,_endOfStorage(_finish)
    {
        for (int i = 0; i < n; ++i)
        {
            _start[i] = val;
        }
    }
    //通过迭代器产生的构造函数
    template<class InputIterator>
    Vector(InputIterator first, InputIterator last)
        :_start(nullptr)
        , _finish(nullptr)
        , _endOfStorage(nullptr)
    {
        while (first != last)
        {
            pushBack(*first);
            ++first;
        }
    }
    Vector(const Vector<T>& v)
    {
        //深拷贝--->开空间--->拷贝
        _start = new T[v.Capacity()];
        for (size_t i = 0; i < v.Size(); i++)
        {
            _start[i] = v[i];
        }
        _finish = _start + v.Size();
        _endOfStorage= _start + v.Capacity();
    }
            Vector<T> operator=(Vector<T> v)
        {
            Swap(v);
            return *this;
        }

    void Swap(Vector<T>& v)
    {
        swap(_start, v._start);
        swap(_finish, v._finish);
        swap(_endOfStorge, v._endOfStorge);
    }
    ~Vector()
    {
        if (_start)
        {
            delete[] _start;
            _start = _finish = _endOfStorage = nullptr;
```

```
        }
    }
private:
    iterator _start;
    iterator _finish;
    iterator _endOfStorage;
};
```

38、请你说说C++中智能指针如何防止内存泄漏的

智能指针主要管理对上分配的内存空间，将一个普通的指针封装为一个栈对象。当栈对象的生命周期结束时，会调用它的析构函数释放普通指针的内存，从而防止内存泄漏

39、请你介绍一下C++中四种智能指针的实现原理

auto_ptr、unique_ptr和shared_ptr都是通过RAII的思想来实现的，其中auto_ptr在进行赋值拷贝时会将资源进行转移，是一种有缺陷的智能指针；而unique是在auto_ptr的基础上，直接将拷贝构造和赋值运算符重载函数设置为删除函数，不允许外部进行赋值拷贝操作；而shared_ptr是C++11中最常用的智能指针，它采用引用计数的方法，记录当前资源被多少个智能指针引用。当新增一个引用时计数器会+1，当减少一个引用时计数器会-1。只有计数器为0时，智能指针才会释放内存资源。但是shared_ptr会造成循环引用的问题。第四种指针也就是weak_ptr就是来解决这种问题的，weak_ptr不能单独使用，只能搭配shared_ptr一起使用

40、请你回答智能指针存在内存泄漏的情况

当两个对象互相使用一个shared_ptr成员变量指向对方时，会造成循环引用，使引用计数器失效，从而造成内存泄漏。为了解决这个问题，引入了weak_ptr弱指针，weak_ptr不会修改引用计数器的值，也不会对对象的内存进行管理

41、简单实现一个shared_ptr智能指针

```
template <class T>
class Share_ptr
{
public:
    Share_ptr(T* ptr)
        :_ptr = ptr
        ,_cntPtr(new size_t(1))
        ,_mtx(new mutex)
    {}
    T& operator*()
    {
        return *_ptr;
    }
    T* operator->()
    {
        return _ptr;
    }
    ~Share_ptr()
    {
        if (subCnt() == 0)
        {
            if (_ptr != nullptr)
            {
                delete _ptr;
                delete _cntPtr;
                delete _mtx;
                _ptr = nullptr;
                _cntPtr = nullptr;
                _mtx = nullptr;
            }
        }
    }

    Share_ptr(Share_ptr<T>& sp)
        :_ptr(sp._ptr)
        ,_cntPtr(sp._cntPtr)
        ,_mtx(sp._mtx)
    {
        addCnt();
    }

    Share_ptr<T>& operator=(Share_ptr<T>& sp)
```

```
    {
        if (_ptr != sp._ptr)
        {
            if (subCnt() == 0)
            {
                delete _ptr;
                delete _cntPtr;
                delete _mtx;
            }
            _ptr = sp._cntPtr;
            _cnt = sp._cntPtr;
            _mtx = sp._mtx;

            addCnt();
        }

        return *this;
    }

    void addCnt()
    {
        _mtx.lock();
        ++(*_cntPtr);
        _mtx->unlock();
    }

    size_t subCnt()
    {
        _mtx->lock();
        --(*_cntPtr);
        _mtx->unlock();
        return *_cntPtr;
    }
private:
    T* _ptr;
    size_t* _cntPtr;
    mutex* _mtx;
};
```

42、右值引用的作用

1、可以实现移动语义，减少拷贝次数来提高代码运行效率
2、给中间临时变量取名字
3、实现完美转发

43、说一说C++中四种cast类型转换

C++中四种cast类型转化分别有：static_cast、reinterpret_cast、const_cast、dynamic_cast。相较于C语言的转换可以进行错误检查，且转换明确。
static_cast主要用于隐式类型转换，可以用于多态中子类对象转换成父类对象，也可以将父类对象转换成子类对象，但是不安全；const_cast主要是将const变量转换为非const变量；dynamic_cast只能用于存储在虚函数的类中，主要用于多态中父类对象的指针或引用转换为子类对象中的指针或引用，如果转换不成功会返回空或者抛异常；reinterpret_cast主要是将一种类型转换为另一种不同的类型，但有可能会出现问题

44、使用const定义常量比#define好的原因

1、定义的const常量有类型检查
2、define宏是在预处理阶段展开，，而const定义的变量时在编译运行阶段使用，便于调试

45、extern和volatile的作用

当**extern**用在一个变量前面，表示声明一个变量，该变量在其他地方已经定义过了，直接使用就行。
还有一个作用是**extern ‘C’**，表示指定代码用C的编译规则来进行编译
volatile是用于修饰一个变量，保持变量的内存可见性，防止编译器的过度优化

46、malloc、calloc、realloc的区别

mallo和calloc都是申请一块新的空间，但是malloc申请的空间都没有进行初始化，而calloc申请的空间都初始化为0。realloc是在原有空间的基础上进行扩充，虽然其参数中有要扩展的空间，最好在用指针来接收这块扩展了的空间。

网络

1、TCP如何保证可靠性

1、第一是通过序列化、确认应答机制和超时重传机制。发送端向接收端发送数据，当接收端收到数据时，需要向发送方发送一个确认应答，表示已收到该数据，并且确认序号会说明下一次需要接收的数据的起始序号。如果发送端一直没有收到确认应答，则表示发送的数据丢失或者确认应答丢失，此时发送方会等待一定时间对数据进行重传
2、第二是滑动窗口机制，主要进行数据流量管理，平衡两端之间的数据吞吐量，解决因接收缓冲区溢出而导致的丢包问题
3、第三是拥塞控制，通过进行网络探测，以一种慢启动，快增长的传输方式，根据网络状态调整发送速度的机制，解决因网络不好导致的丢包问题

2、http和https的区别

1. HTTP协议是以明文的方式在网络中传输数据，而HTTPS协议传输的数据是经过加密后的数据，HTTPS具有更高的安全性
2. HTTP协议端口是80，HTTPS协议端口是443
3. HTTPS协议需要通信前服务端需要申请证书，浏览器端安装对应的根证书，而HTTP不需要申请证书

3、TCP和UDP的区别及使用场景

区别：

1. TCP是面向连接的传输层协议，传输数据前必须建立好连接；而UDP是无连接的传输层协议，不需要两端建立连接就可以传输数据
2. TCP是点对点服务，即一条TCP连接只能有两个端点；UDP支持一对一，一对多，多对一，多对多的交互通信
3. TCP可以保证数据的可靠行，UDP是尽最大努力交付，不保证可靠性
4. TCP首部开销大，20个字节，UDP首部仅有8个字节

使用场景：TCP传输数据可靠但是速度较慢，而UDP传输速度快但不可靠，若数据通信完整性比实时性重要则选用TCP，反之选UDP

4、post和get的区别

1. get的请求数据是通过url传递的，而post是将数据放在正文中传递的
2. get请求在url中传递的数据是有长度限制的，而post没有
3. post比get更安全，因为get参数是直接暴露在url中的
4. get请求会产生一个TCP数据包，而post会产生两个TCP数据包（原因是get请求会将头部和数据一起发送，服务端会进行响应；而post会先进行头部传输，得到一次响应后再发送数据，服务端会再进行响应，所以会产生两TCP数据包）
5. get请求只能进行url编码，而post支持多种编码方式

5、OSI七层模型和TCP/IP五层模型

七层：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层
物理层：负责光电信号的传输 以太网协议
数据链路层：负责相邻设备之间的数据帧传输和网卡硬件地址的描述 以太网协议
网络层：负责地址管理和路由选择 IP协议
传输层：负责服务端和客户端应用程序之间的数据传输 udp、tcp
会话层：负责建立和断开通信连接
表示层：负责设备的数据格式与网络标准数据格式之间的转换
应用层：负责应用程序之间的沟通，约定通信数据的格式 HTTP、FTP、DNS、DHCP

五层：物理层、数据链路层、网络层、传输层、应用层
物理层：负责光电信号的传输 以太网协议
数据链路层：负责相邻设备之间的数据帧传输和网卡硬件地址的描述 以太网协议
网络层：负责地址管理和路由选择 IP协议、ICMP协议
传输层：负责服务端和客户端应用程序之间的数据传输 udp、tcp
应用层：负责应用程序之间的沟通，约定通信数据的格式 HTTP、FTP、DNS、DHCP

6、解释HTTP状态码及说出常见HTTP状态码

以1开头表示服务器收到请求，需要继续执行操作；以2开头的状态码表示请求已成功被接受；以3开头的表示重定向，需要进一步的操作才能完成请求；以4开头的表示是客户端的请求错误；以5开头的表示服务器错误，服务器在处理请求的过程中发生了错误；
200：表示客户端请求成功

301：表示永久重定向，表示请求的资源已被永久的移动到新URI
302：表示临时重定向，请求的资源临时从不同的URI中获取
400：表示客户端请求的语法错误，服务器无法理解
404：表示服务器无法根据客户端的请求找到资源（网页）
500：表示服务器的内部错误，无法完成请求
502：表示作为网关或者代理工作的服务器尝试执行请求时，从远程服务器接收到了一个无效的响应
504：表示充当网关或代理的服务器，未及时从远端服务器获取请求

7、三次握手及四次挥手

网络 卧槽！牛皮了，面试官居然把TCP三次握手四次挥手问的这么详细

8、cookie和session的区别

1、cookie和session都是会话技术，cookie是运行在客户端上的，而session是运行在服务器上的
2、cookie有大小的限制及浏览器存cookie的大小也是有限制的。而session是没有大小限制的，而是和服务器的内存大小有关的
3、cookie存在客户端，存在安全隐患，当别人通过拦截或者从本地文件中找到你的cookie后可以获取你cookie的信息。而session是保存在服务器上的，相对安全些，但是session过多也会增加服务器额压力

9、浏览器输入url回车后，都发生了什么?

浏览器向DNS服务器发送输入进来的url，DNS服务器收到后会返回url对应的IP地址。浏览器就根据IP地址与目标服务器建立TCP连接，然后获取请求页面的html代码，在浏览器的显示窗口内对HTML进行解析渲染，当窗口关闭时，浏览器终止与服务器的连接。

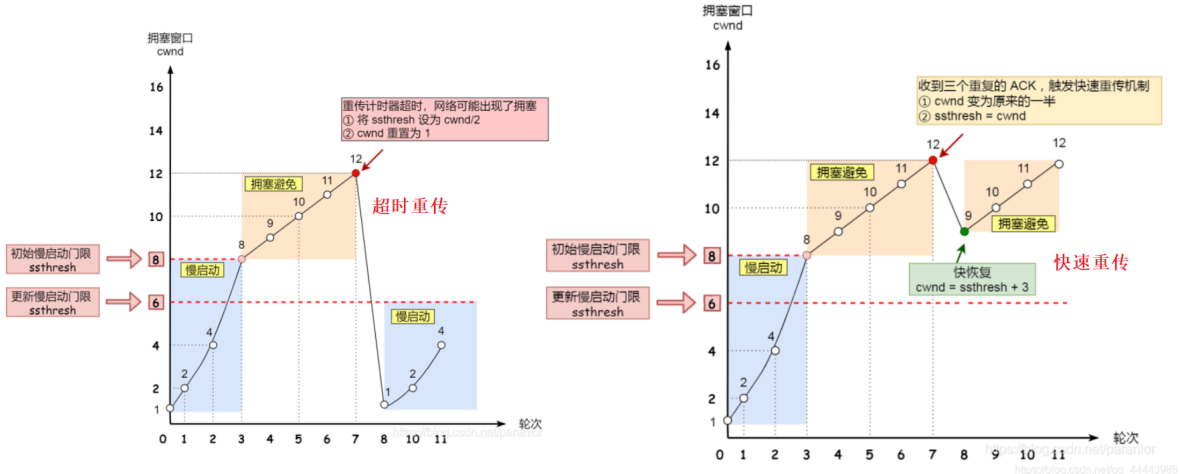
10、滑动窗口实现原理

滑动窗口是实现流量控制，解决丢包问题的一种机制。发送方和接收方都有一个滑动窗口。滑动窗口的大小是由接收方处理数据的能力决定的，表示的是发送的最大数据量。发送方的发送窗口只有收到接收方的ack确认响应之后才会移动发送窗口的左边界。当接收方对前面所有的数据都确认回复后才会移动接收窗口的左边界，如果前面数据未收到而后面的数据先到，此时窗口也不会移动，并不会对已经到达的数据进行回复，确保对端会对这些数据进行重传

11、拥塞控制讲一讲

拥塞控制主要解决因网络不好而导致大量丢包的问题。拥塞控制主要有4个算法，分别是慢启动、拥塞避免、拥塞发生和快速恢复。在慢启动算法中，每当发送方收到一个ack确认响应，拥塞窗口的大小就会+1。也就是拥塞窗口在慢启动算法下呈指数增加。有一个叫慢启动门限变量(默认64k)，当拥塞窗口大小大于该变量时，会使用拥塞避免算法。使拥塞窗口的大小增长速度缓慢些，呈线性增长。当网络状态不好时，会对数据包进行重传，如果是超时重传，慢启动门限变量会变为当前拥塞窗口的一半，且拥塞窗口大小重置为1，重新使用慢启动算法。如果是快速重传，拥塞窗口的大小会变为原来大小的一半，慢启动门限变量变为拥塞窗口大小，然后进入快速恢复算法，此时的拥塞窗口大小就为当前拥塞窗口大小+3，并进入拥塞避免算法。

有可能会画图：



12、HTTPS协议加密流程

https加密流程是将ca认证和混合加密合在一起用的，服务器先生成一个公钥和私钥，然后到权威机构请求颁发一个证书；通信双方建立连接后，服务器会将证书发送给客户端，客户端收到证书会对证书进行解析，根据解析出来的信息进行身份验证；身份验证通过后，使用公钥进行加密，也就是产生一个随机数和自己所支持的对称加密算法列表发送给服务器；服务器收到数据使用私钥进行解密，给客户端回复一个随机数和自己所支持的对称加密算法列表；双方通过自己的随机数与对方发送过来的随机数配合对称加密算法列表进行计算得到一个对称密钥；往后通信都使用对称密钥进行通信

13、说说HTTP的新特性

在http0.9版本中，只支持了html数据传输，而且只有get请求方式，也没有规定标准格式；在1.0版本中规定了http协议的标准格式，也增加了许多请求方式，支持短连接；而在1.1版本是在1.0版本的基础上添加了更多的请求方法和头部信息，并且支持了长连接管线化传输。在2.0版本中采用的是二进制流进行传输，支持多路复用，并允许服务端主动推送数据

14、cookie个session的工作流程

服务端会为每个登录的用户创建一个sesseion对象和一个session ID，其中session保存了客户的重要信息，然后再利用cookie来保存session ID，发送给客户端。客户端每次要访问服务器时只需要传递带有session ID的cookie，服务端就根据已保存的session ID来确认客户端的身份。

操作系统

1、进程和线程的区别

1. 进程是资源分配的最小单位，线程是CPU调度的最小单位
2. 进程有独立的系统资源，而同一进程内的线程共享进程的大部分系统资源，每个线程只拥有一些在运行中必不可少的私有属性
3. 一个进程崩溃，不会对其他进程产生影响；一个线程崩溃，会让同一进程内的其他线程也死掉
4. 进程在创建、切换和销毁开销比较大，而线程比较小。进程创建的时候需要分配系统资源，销毁时要释放系统资源。
5. 进程间通信比较复杂，而同一进程的线程由于共享代码段和数据段，所以通信比较容易

2、什么是死锁，死锁的条件及如何预防死锁

死锁：死锁是指多个进程因竞争资源而造成的一种僵局（互相等待），若无外力作用，这些进程都将无法向前推进
原因：进程在运行过程中，请求和释放资源的顺序不当，会导致死锁。
四个必要条件：1、**互斥条件**：一个资源同一时间只能被一个进程访问，若其他进程请求该资源，则只能等待2、**请求与保持条件**：进程已经保持了至少一个资源，但是又提出了新的资源请求，如果新资源已被占有，这只能进行等待，但对自己以占有的资源保持不释放3、**不可剥夺条件**：进程获得资源在未使用完之前，不能被其他进程强行夺走，只能由获得该资源的进程来释放4、**循环等待条件**: 若干个进程间形成了收尾相接循环等待资源的情况
预防死锁：破坏产生死锁的4个必要条件，主要是破坏请求与保持条件和破坏循环等待条件。破坏请求与保持条件可以用预先静态分配方法，即进程在运行前一次申请完它所需要的全部资源，在它的资源未满足前，不把它投入运行。一旦投入运行后，这些资源就一直归它所有，也不再提出其他资源请求，这样就可以保证系统不会发生死锁。破坏循环等待条件可以采用顺序资源分配法，首先给系统中的资源进行编号，规定进程必须按编号递增的顺序请求资源。例如一个进程只有已占有小编号资源时，才可以申请更大编号的资源

3、虚拟地址空间了解吗

虚拟地址空间是操作系统为了防止不同进程同一时刻在物理内存中运行而对物理内存的争夺和践踏而引入的。虚拟地址空间是操作系统为一个进程描述的虚拟的、连续的、完整的、线性的地址空间，在linux下是一个mm_struct结构体。好处是保证每个进程在各自虚拟地址空间运行，互相不能干扰对方，采用虚拟地址空间，通过页表映射，可以实现进程中的数据在物理内存上的离散式存储，减少了内存碎片，提高了内存的利用率

4、请你说说操作系统中的缺页中断

缺页中断指的是在请求分页系统中，当通过页表要访问的页面不在内存中时，会产生缺页中断，此时操作系统会根据页表中的外存地址在外存中找到所缺的一页，将其调入内存。

5、进程间通信方式

操作系统根据不同的场景提供了不同的方式，**管道，共享内存，消息队列和信号量。**其中管道是内核中的一块缓冲区，分为匿名管道和命名管道。匿名管道只能用于具有亲缘关系的进程间；而命名管道可用于同一主机上任意进程间通信。共享内存的本质是一块物理内存，多个进程将同一块物理内存映射到自己的虚拟地址空间中，再通过页表映射到物理地址达到进程间通信，它是最快的进程间通信方式，相较其他通信方式少了两步数据拷贝操作。消息队列是内核中的一个优先级队列，多个进程通过访问同一个队列，在队列当中添加或者获取节点来实现进程间通信。信号量的本质是内核中的一个计数器，主要实现进程间的同步与互斥，对资源进行计数，有两种操作，分别是在访问资源之前进行的p操作，还有产生资源之后的v操作。

6、僵尸进程和孤儿进程

僵尸进程：子进程先于父进程退出，但是父进程没有进行进程等待，导致无法获取子进程的退出状态信息，使操作系统无法释放子进程的资源，这时候的子进程就是僵尸进程
孤儿进程：父进程先于子进程退出，则子进程称为孤儿进程，此时孤儿进程的父进程成为了1号进程，并且这个孤儿进程运行在后台，并不占据前台终端
如何处理僵尸进程：kill -9命令。子进程退出时向父进程发送SIGCHLD信号，父进程处理SIGCHLD信号，在信号处理函数中调用wait进行处理僵尸进程

7、并发和并行的理解

并发：并发指同一时间段内多个程序的运行
并行：并行指同一时间多个线程或多个程序同时进行

8、有了进程为什么还要线程

进程可以使多个程序并发执行，提高资源利用率和系统的吞吐量，但是进程同一时间只能干一件事，进程在执行的过程中如果阻塞，整个进程就会挂起，即使进程中有些工作不依赖于等待资源，扔不会执行。操作系统引入线程，作为并发执行的基本单位，从而减少程序在并发执行时所付出的时空开销，提高并发性。

9、你了解IPC吗

1、进程间通信方式是操作系统为用户提供的进程之间实现数据通信的方式，因为进程之间具有独立性，无法直接通信。各进程都使用的是自己的虚拟地址空间，无法通过同一地址访问同一块内存
2、操作系统根据不同的场景提供了不同的方式，管道，共享内存，消息队列和信号量。其中管道是内核中的一块缓冲区，分为匿名管道和命名管道。匿名管道只能用于具有亲缘关系的进程间；而命名管道可用于同一主机上任意进程间通信。共享内存的本质是一块物理内存，多个进程将同一块物理内存映射到自己的虚拟地址空间中，再通过页表映射到物理地址达到进程间通信，它是最快的进程间通信方式，相较其他通信方式少了两步数据拷贝操作。消息队列是内核中的一个优先级队列，多个进程通过访问同一个队列，在队列当中添加或者获取节点来实现进程间通信。信号量的本质是内核中的一个计数器，主要实现进程间的同步与互斥，对资源进行计数，有两种操作，分别是在访问资源之前进行的p操作，还有产生资源之后的v操作。

10、fork和vfork的区别

1. fork()创建出来的子进程拷贝父进程大部分资源，也有自己的虚拟地址空间；而vfork创建出来的子进程与父进程共享数据段
2. fork()的父子进程执行顺序是不确定的，而vfork保证子进程先运行，只有子进程退出之后父进程才能运行

11、请问单核机器上写多线程程序，是否需要加锁

需要，在抢占式操作系统中，通常为每个线程分配一个时间片，当某个线程时间片耗尽时，操作系统会将其挂起，然后去运行另一个线程。如果这两个线程共享某些数据，不使用线程加锁，也是有可能导致线程的不安全的

12、 线程同步 方式，并说出系统调用

信号量：信号量可以实现线程的同步与互斥，通过自身计数器对资源进行计数，并通过该计数器判断当前线程是否具有访问的条件，如果可以访问就访问，不可以访问就会挂到pcb等待队列中，直到被其他线程唤醒，才会去争抢资源
int sem_wait(sem_t *sem)信号量减1，如果此时信号量为0则阻塞
int sem_post(sem_t *sem)信号量加1，当信号量大于1时会唤醒阻塞的线程
条件变量和互斥锁：通过条件判断当前线程是否具备访问的权利，如果不满足条件则阻塞，直到等待条件满足才唤醒阻塞线程。因为条件变量本身就是一个临界资源，所以通常搭配互斥锁一起使用
pthread_mutex_init 初始化互斥锁
pthread_mutex_destroy 销毁互斥锁
pthread_mutex_lock(pthread_mutex_t *mutex)给互斥锁加锁，如果以加锁则阻塞，直到持有互斥锁者解锁
pthread_mutex_unlock(pthread_mutex_t *mutex)给互斥锁解锁
pthread_cond_init 初始化条件变量
pthread_cond_destroy 销毁条件变量
pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex) 等待满足条件进行加锁
pthread_cond_signal(pthread_cond_t *cond)唤醒至少一个线程

13、多线程和多进程的优缺点和各自使用场景

多线程之间共享一个进程的地址空间，线程间通信简单，线程的创建和销毁都比多进程简单，速度快，占用内存少。但是线程间会相互影响，一个线程的崩溃也有可能导致其他线程的终止，可靠性若；多进程都有各自独立的虚拟地址空间，进程间不会相互影响，可靠性强。但是进程间通信复杂，进程的创建和销毁复杂，占用的内存多
多线程适用于I/O密集型程序，多进程适用于CPU密集型程序

14、操作系统的缺页置换及算法

当访问内存中不存在的页，且内存已满时，就会从内存中调出一页送至交换区，再从交换区置换出另一页，这种现象就是缺页置换。
最近最少使用(LRU)算法：将最近一段时间以来最长时间未被访问的页面置换出去。实现：使用一个栈，将新页面或者命中的页面移动到栈底，替换时替换栈顶页面
先进先出(FIFO)算法：将内存中驻留时间最久的页面置换出去。实现：使用一个队列，新加入的页面放入队尾，置换时置换队首页面

15、什么是进程，什么是线程

进程：进程是一个正在运行的程序，在操作系统角度讲，进程就是一个pcb，pcb就是操作系统对一个正在运行的程序的描述，通过这个描述实现对指定程序的调度管理。在linux中是一个task_struct结构体
线程：线程就是进程中的一条执行流，是cpu调度的基本单位，但是这个执行流在linux下是通过pcb实现的，因此实际上linux下的线程就是一个pcb，并且在linux下的pcb共用一个虚拟地址空间(进程的虚拟地址空间)，共享了大部分资源。相比进程更加的轻量化，所以线程也被称为轻量级进程

16、什么是重定向

通过改变原本文件的文件描述符对应的文件描述信息，从而导致改变了数据的流向，将数据不再写入原本的文件而写入到新指定的文件

17、软链接和硬链接的区别

1、通过ln创建的文件是硬链接文件，硬链接文件和源文件共用一个inode结点，也就是说硬链接文件只是源文件的一个别名。而通过ln -s创建的软链接文件是拥有自己独立的inode结点号，是一个新的文件，只是这个文件的内容保存了一个文件的路径名
2、删除源文件后，软链接文件会失效，而硬链接文件只是链接数-1
3、由于软链接是保存的是源文件的路径，所以可以跨分区链接。而硬链接是使用了同一个inode结点号，跨分区将无法找到唯一的那个源文件

18、如何判断大小端

大端指的是低地址存储数据的高位，小端值的是低地址存储数据的低位

```
int main()
{
    union
    {
        int i;
        char c;
    }un;
    un.i = 1;//低位为1
    if (un.c)//取低地址
        cout << "小端" << endl;
    else
        cout << "大端" << endl;
    return 0;
}

```

19、用户态和内核态的区别及转换方式

两者的最大区别是权限不同，运行在用户态的程序不能直接访问操作系统内核数据结构　和程序，只有运行在内核态的程序才可以访问。两者转换方式主要是。系统调用，异常和中断

20、说说生产者和消费者线程模型

生产者和消费者模型通常用于大量数据的产生与处理的场景。其是通过一个缓冲队列来实现的。开启一个或者多个线程来生产任务，生产的任务放到缓冲队列中。再开启一个或多个线程从缓冲队列中取出任务并进行处理。这样子降低了生产者和消费之间的耦合性，并且支持忙闲不均，例如生成任务速度过快，这时候就可以多开几个消费者线程来处理任务

21、系统调用及使用过哪些

系统调用指运行时用户态的程序向操作系统内核请求需要更高权限的运行服务。系统调用提供了用户程序与操作系统之间的接口。例如对文件的读写操作是调用的open、write、read等还有创建进程时调用的fork和vfork

22、说一下你所认识的锁

悲观锁：悲观锁是认为我们在使用共享资源时会被其他线程修改，容易导致线程安全的问题，所以在访问前总是先对共享资源进行加锁，阻塞其他线程。适用于经常进行写操作的场景
乐观锁：乐观锁通常以CAS操作和版本号机制来实现。其悲观锁相反，会直接对共享资源进行修改，但是在提交更新修改结果前会验证这修改期间有没有其他线程对该资源进行修改，如果没有则提交更新，如果有则放弃本次操作。适用于经常进行读操作而很少进行写操作的场景
互斥锁：互斥锁是最底层的一种锁，其原理是当一个线程占据了这个锁后，其他线程想加锁会加锁失败并且会被阻塞。适用于加锁部分代码的执行时间长
自旋锁：自旋锁也是基于CAS操作来实现的，任何尝试获取该锁的线程都将一直进行自旋，占用cpu的资源。直到获得该锁，并且同一时间内只能由一个线程能获得该锁。适用于加锁部分代码的执行时间短

读写锁：读锁是当没有线程持有写锁时，读锁就能被多个线程并发持有，从而提高资源利用率和访问效率，并且不存在线程安全问题；写锁是任何一个线程持有写锁时，其他线程想要获取读锁或者写锁都会被阻塞。适用于并发要求高的场景

23、说说乐观锁如何实现的

乐观锁的实现主要是通过cas机制和版本号机制来实现的。在cas机制中有3个操作数，分别是内存变量值，旧预期值，新预期值。当我们要修改一个变量时，会对内存变量值和旧预期值进行比较，如果相同则将交换新预期值和内存变量值；如果不同则将内存变量值作为旧预期值，继续重复比较。而版本号机制主要是用来解决ABA问题，每当修改一次都会标记为新的版本号，只有内存变量值和新旧预期值的版本号相同，才会进行修改

24、请你介绍一下5种IO模型

第一个是**阻塞IO模型**当调用者进行IO请求调用时，由于请求调用的条件不满足，就会一直阻塞等待，直到条件满足才会进行下一步的操作。第二个是**非阻塞IO模型**当发起IO请求调用后，会每隔一段时间去检测IO事件是否就绪，没有就绪就可以做其他事情。第三个是**信号驱动IO模型**，其原理是通过定义信号处理函数，当进程收到SIGIO信号时表示IO就绪，进程就会去处理IO事件。第四个是**异步IO模型**，通过自定义IO完成信号处理方式，发起异步调用，告诉操作系统要完成指定功能，剩下的IO功能完全由操作系统来完成，完成后通过信号来通知进程。第五个是**IO复用模型** ，其可以用select、poll和epoll函数来实现，可以同一时间对大量描述符进行IO就绪事件监控，当知道有数据可读或可写时才会调用IO操作函数

25、说说select、poll、epoll的区别

select所能监控的描述符数量是有限的，默认是1024个，poll和epoll对监控的描述符数量没有限制；select和poll采用的是轮旋遍历，会随着监控的描述符的增多，性能也会随之降低，而epoll监控采用的是异步阻塞，不会随着描述符的增多导致性能下降；select的跨平台移植比poll和epoll好；select和poll每次监控调用返回后需要程序员遍历判断哪些描述符是否就绪，而epoll会直接向程序员返回就绪的文件描述符，可以直接对描述符进行操作；epoll支持水平触发和边缘触发

26、线程池的意义及如何实现一个线程池

线程池是使用了已经创建好的线程进行循环处理任务，避免了大量线程的频繁创建与销毁的时间成本。要实现线程池，我们可以设置一个生产者消费者队列，作为临界资源，然后初始化多个线程，并让其运行起来，加锁去队列取任务并运行。当任务队列为空时，所有线程都阻塞。当生产者队列来了一个任务后，先对队列加锁，把任务挂载到队列上，然后使用条件变量去通知阻塞中的一个线程，让其获得任务并处理

27、水平触发(LT)和边缘触发(ET)

水平触发是不断查询可用的描述符，当有可用的描述符就会触发事件；
边缘触发是只有当IO的转态改变时才会触发事件，并且会一次性把数据全部处理完

数据结构

1、快排和归并的联系与区别

联系：原理都基于分治法，首先将待排序数组分为两组，对两组的数据进行排序，最后将两组结果合并起来。
区别：分解和合并的策略不同。 快速排序 是根据元素的值来分的，大于某个值的元素一组，小于某个值的元素一组。合并时只要前后相连即可。 归并排序 是直接将待排序数组的前一半分为一组，后一半分为一组，然后进行排序。合并时还要再将两个有序数组进行排序。

2、链表和数组的区别

1. 数组的元素在内存地址中是连续存放的。链表的元素在内存中不一定是连续存放的，但是逻辑上是连续的
2. 数组的访问速度很快，因为数组可以根据下标进行快速定位。链表的访问速度较慢，因为链表访问元素需要移动指针
3. 数组增加或者删除元素时间复杂度通常是O(n),都需要大量移动元素。而链表在任意位置插入元素和删除元素的速度快，只要修改指针指向就可以，时间复杂度为O(1)

3、说出你所知道的排序算法及其复杂度

1、**直接插入排序**：一开始假设第一个元素为有序，然后从第二个元素开始向前找到合适的位置并插入。稳定，平均时间复杂度为O(n^2)
2、**希尔排序**：是对直接插入排序的优化和扩展，把数据按下标的一定增量分组，对每组使用直接插入排序算法排序；当增量减至 1 时，整个元素恰被分成一组，进行整体排序后终止排序。不稳定，时间复杂度为O(n^1.3) 到O(n^2)
3、**选择排序**：第一次从数组中找到最小的元素放到起始位置，然后从之后未排序元素中再找到最小的元素，放到以排序元素的末尾。不稳定，时间复杂度为O(n^2)
4、**堆排序**：构建一个大堆，将堆顶元素和末尾元素进行交换，然后再对n-1个元素重新构建一个大堆，如此重复循环，就可以得到一个有序的数组了。不稳定，时间复杂度为O(nlogn)
5、**冒泡排序**：循环n次，每次循环都通过相邻比较交换，每轮循环找出未排序数组中的最大元素，将其放未排序部分的末尾。稳

定，时间复杂度为O(n^2)
6、**快速排序**：通过一趟排序将待排数据分隔成独立的两部分，其中一部分数据是比关键字小的，另一部分是比关键字大的，则可分别对这两部分数据继续进行排序，以达到整个序列有序。不稳定，O(nlogn)到O(n^2)，当数据全部反向排序时时间复杂度最高
7、**归并排序**：首先把一个数组中的元素，按照某一方法，先拆分了之后，按照一定的顺序各自排列，然后再归并到一起，使得归并后依然是有一定顺序的。稳定，时间复杂度为O(nlogn)

4、topK问题-如何从海量数据取最大的k个数

最小堆法：先读取k个数，建成小堆，然后将剩余元素依次和堆顶元素比较，如果小于或者等于堆顶元素，则继续比较下一个，如果大于堆顶元素，则删除堆顶元素，并将新元素插入堆中，重新建立一个小堆。当遍历完全部元素后，最小堆中的数据即为最大的k个数

```
void adjustSmall(vector<int>& arr, int sz, int idx)
{
    int child = 2 * idx + 1;
    while (child < sz)
    {
        if ((child + 1 < sz) && (arr[child] > arr[child + 1]))
            ++child;
        if (arr[idx] > arr[child])
        {
            swap(arr[idx], arr[child]);
            idx = child;
            child = 2 * idx + 1;
        }
        else
            break;
    }
}

void topK(vector<int>& arr, int k)
{
    vector<int> res(arr.begin(), arr.begin() + k);
    make_heap(res.begin(), res.end(), greater<int>());
    for (size_t i = k; i < arr.size(); ++i)
    {
        if (arr[i] > res[0])
        {
            swap(arr[i], res[0]);
            adjustSmall(res, k, 0);
        }
    }
    for (const auto& e : res)
        cout << e << " ";
}
```

5、快排非递归

通过一趟排序将待排数据分隔成独立的两部分，其中一部分数据是比关键字小的，另一部分是比关键字大的，则可分别对这两部分数据继续进行排序，以达到整个序列有序

```
int getMid(vector<int>& arr, int low, int high)
{
    int mid = low + ((high - low) >> 1);

    if (arr[mid] > arr[high])
    {
        swap(arr[mid], arr[high]);
    }
    if (arr[low] > arr[high])
    {
        swap(arr[low], arr[high]);
    }
    if (arr[mid] > arr[low])
    {
        swap(arr[mid], arr[low]);
    }

    return arr[low];

```

```
}

int helper(vector<int>& arr, int left, int right)
{
    int key =getMid(arr, left, right);
    int start = left;
    while (left < right)
    {
        while (left < right && arr[right] >= key)
            --right;
        while (left < right && arr[left] <= key)
            ++left;
        swap(arr[left], arr[right]);
    }
    swap(arr[left], arr[start]);
    return left;
}

void quickSort(vector<int>& arr)
{
    stack<int> st;
    st.push(arr.size() - 1);
    st.push(0);

    while (!st.empty())
    {
        int left = st.top();
        st.pop();
        int right = st.top();
        st.pop();
        int div = helper(arr, left, right);
        if (left < div - 1)
        {
            st.push(div - 1);
            st.push(left);
        }
        if (div + 1 < right)
        {
            st.push(right);
            st.push(div + 1);
        }
    }
}
```

6、堆排序

构建一个大堆，将堆顶元素和末尾元素进行交换，然后再对n-1个元素重新构建一个大堆，如此重复循环，就可以得到一个有序的数组了

```
void adjust(vector<int>& arr, int sz, int idx)
{
    int child = 2 * idx + 1;
    while (child < sz)
    {
        if ((child + 1 < sz) && (arr[child] < arr[child + 1]))
            ++child;
        if (arr[idx] < arr[child])
        {
            swap(arr[idx], arr[child]);
            idx = child;
            child = 2 * idx + 1;
        }
        else
            break;

    }
}

void makeHeap(vector<int>& arr)
{
    for (int i = (arr.size()-2)/2; i >= 0; --i)
```

```
    {
        adjust(arr, arr.size() ,i);
    }
}

void heapSort(vector<int>& arr)
{
    makeHeap(arr);
    int end = arr.size()-1;
    for (int i = 0; i < arr.size(); ++i)
    {
        swap(arr[0], arr[end]);
        adjust(arr, end, 0);
        --end;
    }
}
```

7、直接插入排序

一开始假设第一个元素为有序，然后从第二个元素开始向前找到合适的位置并插入

```
void insertSort(vector<int>& arr)
{
    for (int i = 1; i < arr.size(); ++i)
    {
        int data = arr[i];
        int end = i - 1;
        while (end >= 0 && arr[end] > data)
        {
            arr[end + 1] = arr[end];
            --end;
        }
        arr[end + 1] = data;
    }
}
```

8、希尔排序

是对直接插入排序的优化和扩展，把数据按下标的一定增量分组，对每组使用直接插入排序算法排序；当增量减至 1 时，整个元素恰被分成一组，进行整体排序后终止排序

```
void shellSort(vector<int>& arr)
{
    int gap = arr.size() / 2;
    while (gap > 0)
    {
        for (int i = gap; i < arr.size(); ++i)
        {
            int data = arr[i];
            int end = i - gap;
            while (end >= 0 && arr[end] > data)
            {
                arr[end + gap] = arr[end];
                end -= gap;
            }
            arr[end + gap] = data;
        }
        gap /= 2;
    }
}
```

9、选择排序

第一次从数组中找到最小的元素放到起始位置，然后从之后未排序元素中再找到最小的元素，放到已排序元素的末尾

```
void selectSort(vector<int>& arr)
{
    for (int i = 0; i < arr.size(); ++i)
    {
        int minIdx = i;

```

```
        for (int j =  i + 1; j < arr.size(); ++j)
        {
            if (arr[j] < arr[minIdx])
                minIdx = j;
        }
        swap(arr[i], arr[minIdx]);
    }
}
```

10、冒泡排序

循环n次，每次循环都通过相邻比较交换，每轮循环找出未排序数组中的最大元素，将其放未排序部分的末尾

```
void bubbleSort(vector<int>& arr)
{
    int n = arr.size() - 1;
    int pos = 0;
    for (int i = 0; i < arr.size(); ++i)
    {
        bool flag = false;
        for (int j = 0; j < n; ++j)
        {
            if (arr[j + 1] < arr[j])
            {
                swap(arr[j + 1], arr[j]);
                flag = true;
                pos = j;
            }
        }
        if (!flag)
            break;
        n = pos;
    }
}
```

算法编程题

1、统计一个数的二进制1的个数

```
int fun(int v)
{
    int num = 0;

    while (v){
        v &= (v - 1);
        num++;
    }
    return num;
}
```

2、股票的最大利润

剑指 Offer 63. 股票的最大利润

other

1、给一个超过100G大小的log file, log中存着IP地址, 设计算法找到出现次数最多的IP地址?

利用**哈希切割**。假设要将文件分为100份。先将ip地址通过哈希函数转换为1个数值，然后再用该值模上100，最后得到的值就是该ip将要存放的第几个文件的位置，例如这100个文件标记为0-99。然后一个ip经过哈希函数再模上100得到10，则将该ip存放到第10个文件中，此时就可以将同样的IP存放在同一个文件中。而且每个文件的大小并不是很庞大，如果还是很大我们可以再次通过不同的哈希函数进行二次切割。最后统计每个文件中的ip地址出现的次数。就可以得到出现次数最多的ip地址了

2、与上题条件相同， 如何找到top K的IP?

在上题中我们已经计算出每个ip所出现的次数，可以将对应的ip地址和ip出现次数弄成一个键值对。此时我们可以建立一个大小为k的小堆，然后通过IP出现的次数依次和堆顶元素进行比较，如果比堆顶元素大，则替换堆顶元素，重新进行调整成小堆，依次比较遍历，最后堆中存在的ip就是top K的IP

3、给定100亿个整数，设计算法找到只出现一次的整数?

可以利用位图，但是要将位图的比特位状态变为2位，00表示不存在，01表示只出现一次，10则表示出现2次及以上。然后遍历位图，只要bit位为01的数，则就表示该数字只出现过一次

4、给两个文件，分别有100亿个整数，我们只有1G内存，如何找到两个文件交集?

可以利用两个位图，同一个哈希函数，每个位图都对应着一个文件中的整数。然后最后再遍历判断两个位图中同位置是否为1，为1则表示该位上对应的数字为交集的其中一个。

5、1个文件有100亿个int，1G内存，设计算法找到出现次数不超过2次的所有整数

可以利用位图，但是要将位图的比特位状态变为2位，00表示不存在，01表示只出现一次，10则表示出现2次，11表示出现3次及及以上。然后遍历位图，只要bit位为10的数，则就表示该数字出现次数不超过2次

6、给两个文件，分别有100亿个请求，我们只有1G内存，如何找到两个文件交集？分别给出精确算法和 近似算法

精确算法：可以利用两个位图，同一个哈希函数，每个位图都对应着一个文件中的整数。然后最后再遍历判断两个位图中同位置是否为1，为1则表示该位上对应的数字为交集的其中一个

近似算法：可以利用两个布隆过滤器，每个布隆过滤器都有500M的内存，其中布隆过滤器的哈希函数都是一样的，最后再用两个布隆过滤器进行比较，如果相同位置的bit位为1则表示这个请求是属于交集的其中一个

7、请设计一个类，只能在堆上创建对象

堆上创建则表示对象只能通过new来实例化对象，要将构造函数私有化，并且要防止外部通过拷贝构造或者赋值运算符重载进行拷贝赋值。所以要将这两个函数设置为删除函数。向外提供一个获得对象的函数，函数中是返回一个在堆上创建的对象。由于类中的函数含有this指针，所以在第一次要获取对象时必须要先有对象才能调用，所以要将该函数置为static函数

```
class HeapObj
{
public:
    static HeapObj* createObj()
    {
        return new HeapObj();
    }
private:
    HeapObj()
    {}
    HeapObj(const HeapObj& obj) = delete;
    HeapObj& operator=(const HeapObj& obj) = delete;
};
```

8、请设计一个类，只能在栈上创建对象

将构造函数私有化，外部无法通过构造函数来创建对象。创建一个函数，该函数时返回一个在栈上创建的对象。并且该函数为static函数。拷贝构造和赋值运算符重载函数可有可无

```
class StackObj
{
public:
    static StackObj createObj()
    {
        return StackObj();
    }
private:
    StackObj()
    {}
};
```

8、请设计一个类，该类不能被拷贝

将拷贝构造和赋值运算符重载函数设置为删除函数

```
class CopyBan
{
private:
    CopyBan(const CopyBan& cb) = delete;
    CopyBan& operator=(const CopyBan& cb) = delete;
};
```

9、请设计一个类，该类不能被继承

```
class A final
{
    //...
};
```

设计模式

1、请问你用过哪些设计模式

单例模式、工厂模式、观察者模式、装饰器模式
单例模式主要解决一个全局使用的类频繁的创建和销毁的问题。而使用单例模式可以确保一个类只有一个实例，而且该类实例化后是会被整个系统可见。**工厂模式**中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象，这样做的目的就是将产品类的实例化操作延迟到工厂子类中完成。**观察者模式**是定义对象一种一对多的关系，当一个对象的状态发生改变时，所有依赖它的对象都得到通知并自动更新。装饰器模式是一种用于代替继承的技术，它可以动态地给一个对象增加一些额外的功能，就增加功能而言，它比继承更加灵活，同时也避免了类型体系的快速膨胀

2、请问如何实现一个单例模式，说出思路并写出代码

单例模式的实现主要分为饿汉式和懒汉式，**饿汉式**是在程序启动时就把对象的资源初始化好了，定义一个静态的该类的一个对象，然后在类外初始化，在类中提供获取该对象的方法**懒汉式**是在使用时才实例化对象，实现原理也是将构造函数私有化，然后在类中提供静态的该类的一个对象指针，并在类外初始化为空。然后再类中提供一个可以获取到该对象指针的函数，当该指针为空时就创建该对象指针，并返回，如果不为空则则返回已经实例化了的对象指针
饿汉式:

```
//饿汉式
class Singleton
{
public:
    //获取唯一实例的静态方法
    static Singleton* getSingleton()
    {
        return &_singleton;
    }

private:
    //私有构造
    Singleton() {};
    Singleton(Singleton const&) = delete;
    Singleton& operator=(Singleton const&) = delete;

    static Singleton _singleton;
};
Singleton Singleton::_instance;
```

懒汉式:

```
//懒汉式
class Singleton
{
public:
    static Singleton* getSingleton()
    {
        if (_singleton == nullptr)
        {
            _mutex.lock();
            if (_singleton == nullptr)
            {
                _singleton = new Singleton();
            }
            _mutex.unlock();
        }
        return _singleton;
    }

private:
    Singleton() {};

```

```
    Singleton(Singleton const&) = delete;
    Singleton& operator=(Singleton const&)delete;

    static mutex _mutex;//互斥锁
    static Singleton* _singleton;
};
Singleton* Singleton::_singleton = nullptr;
mutex Singleton::_mutex;
```

3、单例模式中的懒汉式实现，在多线程下如何保证线程安全

使用互斥锁，第一次判断为空则不加锁，过为空则在加锁判断是否为空，若为空则生成对象

4、说说工厂模式的优缺点及使用场景

工厂模式的优点是代码复用率高，更改功能容易，实现了对象创建和使用的分离；缺点是系统中的类成对增加，增加了系统的复杂度和理解度；适用于不关心对象的创建和实现的场景

5、说说装饰器模式的优缺点及使用场景

装饰器模式的优点是装饰类和被装饰类可以独立发展，不会互相耦合可以动态的扩展一个实现类的功能；缺点是当适用的修饰类过多时容易造成错误，排查过程也比较繁琐；适用于可以动态地增加一个类的功能和动态地撤销一个类的功能的场景