2022 年头条精选 50 面试题及答案

1. C++智能指针如何解决内存泄露问题.

1.shared_ptr 共享的智能指针
std::shared_ptr 使用引用计数，每一个 shared_ptr 的拷贝都指向相同的内存。在最后一个 shared_ptr 析构的时候，内存才会被释放。
可以通过构造函数、std_make_shared 辅助函数和 reset 方法来初始化 shared_ptr：

```
// 构造函数初始化
std::shared_ptrp ( new int(1) ) ;
std::shared_ptrp2 = p ;
// 对于一个未初始化的智能指针，可以通过 reset 方法来初始化。
std::shared_ptrptr; ptr.reset ( new int (1) ) ;
if (ptr) {cout << “ptr is not null.\n” ; }
```

不能将一个原始指针直接赋值给一个智能指针：

```
std::shared_ptrp = new int(1) ;// 编译报错，不允许直接赋值
```

获取原始指针：
通过 get 方法来返回原始指针

```
std::shared_ptrptr ( new int(1) ) ;
int * p =ptr.get () ;
```

指针删除器：
智能指针初始化可以指定删除器

```
void DeleteIntPtr ( int * p ) {
    delete p ;
}
std::shared_ptrp ( new int , DeleteIntPtr ) ;
```

当 p 的引用技术为 0 时，自动调用删除器来释放对象的内存。删除器也可以是一个 lambda 表达式，例如

```
std::shared_ptrp ( new int , [](int * p){delete p} ) ;
```

注意事项：
(1).不要用一个原始指针初始化多个 shared_ptr。
(2).不要再函数实参中创建 shared_ptr，在调用函数之前先定义以及初始化它。
(3).不要将 this 指针作为 shared_ptr 返回出来。
(4).要避免循环引用。
2.unique_ptr 独占的智能指针
unique_ptr 是一个独占的智能指针，他不允许其他的智能指针共享其内部的指针，不允许通过赋值将一个 unique_ptr 赋值给另外一个 unique_ptr。
unique_ptr 不允许复制，但可以通过函数返回给其他的 unique_ptr，还可以通过 std::move 来转移到其他的 unique_ptr，这样它本身就不再拥有原来指针的所有权了。

如果希望只有一个智能指针管理资源或管理数组就用 unique_ptr，如果希望多个智能指针管理同一个资源就用 shared_ptr。

3.weak_ptr 弱引用的智能指针

弱引用的智能指针 weak_ptr 是用来监视 shared_ptr 的，不会使引用计数加一，它不管理 shared_ptr 内部的指针，主要是为了监视 shared_ptr 的生命周期，更像是 shared_ptr 的一个助手。

weak_ptr 没有重载运算符*和->，因为它不共享指针，不能操作资源，主要是为了通过 shared_ptr 获得资源的监测权，它的构造不会增加引用计数，它的析构不会减少引用计数，纯粹只是作为一个旁观者来监视 shared_ptr 中关离得资源是否存在。

weak_ptr 还可以用来返回 this 指针和解决循环引用的问题。

2. 常见 web 安全问题，SQL 注入、XSS、CSRF，基本原理以及如何防御

1.SQL 注入

原理:

1).SQL 命令可查询、插入、更新、删除等，命令的串接。而以分号字元为不同命令的区别。（原本的作用是用于 SubQuery 或作为查询、插入、更新、删除……等的条件式）

2).SQL 命令对于传入的字符串参数是用单引号字元所包起来。（但连续 2 个单引号字元，在 SQL 资料库中，则视为字串中的一个单引号字元）

3).SQL 命令中，可以注入注解

预防:

1).在设计应用程序时，完全使用参数化查询（Parameterized Query）来设计数据访问功能。

2).在组合 SQL 字符串时，先针对所传入的参数作字元取代（将单引号字元取代为连续个单引号字元）。

3).如果使用 PHP 开发网页程序的话，亦可打开 PHP 的魔术引号（Magic quote）功能（自动将所有的网页传入参数，将单引号字元取代为连续 2 个单引号字元）。

4).其他，使用其他更安全的方式连接 SQL 数据库。例如已修正过 SQL 注入问题的数据库

5).连接组件，例如 ASP.NET 的 SqlDataSource 对象或是 LINQ to SQL。

使用 SQL 防注入系统。

2. XSS 攻击

原理:

xss 攻击可以分成两种类型：

1.非持久型 xss 攻击

非持久型 xss 攻击是一次性的，仅对当次的页面访问产生影响。非持久型 xss 攻击要求用户访问一个被攻击者篡改后的链接，用户访问该链接时，被植入的攻击脚本被用户游览器执行，从而达到攻击目的。

2.持久型 xss 攻击

持久型 xss 攻击会把攻击者的数据存储在服务器端，攻击行为将伴随着攻击数据一直存在。下面来看一个利用持久型 xss 攻击获取 session id 的实例。

防范：

1.基于特征的防御

XSS 漏洞和著名的 SQL 注入漏洞一样，都是利用了 Web 页面的编写不完善，所以每一个漏洞所利用和针对的弱点都不尽相同。这就给 XSS 漏洞防御带来了困难：不可能以单一特征来概括所有 XSS 攻击。

传统 XSS 防御多采用特征匹配方式，在所有提交的信息中都进行匹配检查。对于这种类型的 XSS 攻击，采用的模式匹配方法一般会需要对“javascript”这个关键字进行检索，一旦发现提交信息中包含“javascript”，就认定为 XSS 攻击。这种检测方法的缺陷显而易见：骇客可以通过插入字符或完全编码的方式躲避检测：

1). 在 javascript 中加入多个 tab 键，得到

```
<IMG SRC="jav ascript:alert('XSS');">;
```

2). 在 javascript 中加入(空格)字符，得到

```
<IMG SRC="javascri pt:alert('XSS');">;
```

3). 在 javascript 中加入(回车)字符，得到

```
<IMG SRC="javasc
ript:alert('XSS');">;
```

4). 在 javascript 中的每个字符间加入回车换行符，得到

```
<IMG SRC="javascrip\r\nt:alert('XSS');">
```

5). 对”javascript:alert(‘XSS’)”采用完全编码，得到

```
<IMGSRC=javascrip?74:alert('XSS')>
```

上述方法都可以很容易的躲避基于特征的检测。而除了会有大量的漏报外，基于特征的

还存在大量的误报可能：在上面的例子中，对上述某网站这样一个地址，由于包含了关键字“javascript”，也将会触发报警。

2.基于代码修改的防御

和 SQL 注入防御一样，XSS 攻击也是利用了 Web 页面的编写疏忽，所以还有一种方法就是从 Web 应用开发的角度来避免：

对所有用户提交内容进行可靠的输入验证，包括对 URL、查询关键字、HTTP 头、POST 数据等，仅接受指定长度范围内、采用适当格式、采用所预期的字符的内容提交，对其他的一律过滤。

实现 Session 标记(session tokens)、CAPTCHA 系统或者 HTTP 引用头检查，以防功能被第三方网站所执行。

确认接收的的内容被妥善的规范化，仅包含最小的、安全的 Tag(没有 javascript)，去掉任何对远程内容的引用(尤其是样式表和 javascript)，使用 HTTP only 的 cookie。

3. CSRF 攻击

原理：

CSRF 攻击原理比较简单，假设 Web A 为存在 CSRF 漏洞的网站，Web B 为攻击者构建的恶意网站，User C 为 Web A 网站的合法用户。

1.用户 C 打开浏览器，访问受信任网站 A，输入用户名和密码请求登录网站 A；
2.在用户信息通过验证后，网站 A 产生 Cookie 信息并返回给浏览器，此时用户登录网站 A 成功，可以正常发送请求到网站 A；
3.用户未退出网站 A 之前，在同一浏览器中，打开一个 TAB 页访问网站 B；
4.网站 B 接收到用户请求后，返回一些攻击性代码，并发出一个请求要求访问第三方站点 A；
5.浏览器在接收到这些攻击性代码后，根据网站 B 的请求，在用户不知情的情况下携带 Cookie 信息，向网站 A 发出请求。网站 A 并不知道该请求其实是由 B 发起的，所以会根据用户 C 的 Cookie 信息以 C 的权限处理该请求，导致来自网站 B 的恶意代码被执行。

防范：

1.检查 Referer 字段

HTTP 头中有一个 Referer 字段，这个字段用以标明请求来源于哪个地址。在处理敏感数据请求时，通常来说，Referer 字段应和请求的地址位于同一域名下。以上文银行操作为例，Referer 字段地址通常应该是转账按钮所在的网页地址，应该也位于 www.examplebank.com 之下。而如果是 CSRF 攻击传来的请求，Referer 字段会是包含恶意网址的地址，不会位于 www.examplebank.com 之下，这时候服务器就能识别出恶意的访问。

2.添加校验 token

由于 CSRF 的本质在于攻击者欺骗用户去访问自己设置的地址，所以如果要求在访问敏感数据请求时，要求用户浏览器提供不保存在 cookie 中，并且攻击者无法伪造的数据作为校验，那么攻击者就无法再执行 CSRF 攻击。这种数据通常是表单中的一个数据项。服务器将其生成并附加在表单中，其内容是一个伪乱数。当客户端通过表单提交请求时，这个伪乱数也一并提交上去以供校验。正常的访问时，客户端浏览器能够正确得到并传回这个伪乱数，而通过 CSRF 传来的欺骗性攻击中，攻击者无从事先得知这个伪乱数的值，服务器端就会因为校验 token 的值为空或者错误，拒绝这个可疑请求。

3. linux 中软连接和硬链接的区别.

原理上，硬链接和源文件的 inode 节点号相同，两者互为硬链接。软连接和源文件的 inode 节点号不同，进而指向的 block 也不同，软连接 block 中存放了源文件的路径名。实际上，硬链接和源文件是同一份文件，而软连接是独立的文件，类似于快捷方式，存储着源文件的位置信息便于指向。

使用限制上，不能对目录创建硬链接，不能对不同文件系统创建硬链接，不能对不存在的文件创建硬链接；可以对目录创建软连接，可以跨文件系统创建软连接，可以对不存在的文件创建软连接。

4. STL 内存分配方式

在 STL 中考虑到小型区块所可能造成的内存碎片问题，SGI STL 设计了双层级配置器，第一级配置器直接使用 malloc()和 free();第二级配置器则视情况采用不同的策略：当配置区块超过 128bytes 时，则视之为足够大，便调用第一级配置器；当配置区块小于 128bytes 时，则视之为过小，为了降低额外负担，便采用复杂的内存池的方式来整理，而不再求助于第一级配置器。

每次配置器需要向系统要内存的时候，都不是按客户需求向系统申请的，而是一次性向系统要了比需求更多的内存，放在内存池里，有一个 free_start 和 free_end 指示剩余的空间（也就是说内存池剩余的空间都是连续的，因此每次重新向 system heap 要空间的时候，都会把原先内存池里没用完的空间分配给合适的 free list。）当 free-list 中没有可用区块了的时候，会首先从内存池里要内存，同样，也不是以按客户需求要多少块的，而是一次可能会要上 20 块，如 果内存池内空间允许的话，可能会得到 20 个特定大小的内存，如果内存池给不了那么多，那么就只好尽力给出；如果连一个都给不出，那么就要开始向系统即 system heap 要空间了。换算的标准是 bytes_to_get=2*total_bytes+ROUND_UP(heap_size>>4)。这个时候使用的是 malloc，如果没成功，就尝试着从大块一点的 freelist 那里要一个来还给内存池，如果还是不行，那么会调用第一级空间配置器的 malloc::allocate，看看 out-of-memory 机制能做点什么。

假设我们向系统要 x 大小的内存，
（1）x 大于 128byte，用第一级配置器直接向系统 malloc，至于不成功的处理，过程仿照 new，通过 set_new_handler 来处理，直到成功返回相应大小的内存或者是抛出异常或者是干脆结束运行；
（2）x 小于 128byte，用第二级配置器向内存池相应的 free_list 要内存，如果该 freelist 上面没有空闲块了，
（2.1）从内存池里面要内存，缺省是获得 20 个节点，如果内存池中剩余的空间不能完全满足需求量，但足够供应一个（含）以上的区块，则应尽力满足需求。
（2.2） 如果一个都不能够满足的话，则从系统的 heap 里面要内存给到内存池，换算的标准是 bytes_to_get=2*total_bytes+ROUND_UP(heap_size>>4)，申请的大小为需求量的两倍加上一个 附加值，如果内存池中还有剩余的内存，则将残余零头分配给适当的 free list，这时使用的是系统的 malloc，如果要不到：
（2.3）从比较大的 freelist 那里要内存到内存池，如果还是要不到：
（2.4） 从系统的 heap 里面要内存给到内存池，换算标准跟 2.2 一样，但是这时候使用的是第一级配置器的 allocate，主要是看看能不能通过 out_of_memory 机制得到一点内存。所以，freelist 总是从内存池里要内存的，而内存池可能从 freelist 也可能从系统 heap 那里 要内存。
SGI STL 的 alloc 的主要开销就在于管理这些小内存，管理这些小内存的主要开销就在于，每次 freelist 上的内存块用完了，需要重新要空间，然后建立 起这个 list 来。freelist 上的内存，会一直保留着直到程序退出才还给系统。但这不会产生内存泄漏，一来是管理的都是小内存，二来是，占用的内存 只会是整个程序运行过程中小内存占用量最大的那一刻所占用的内存。

5. TCP 的拥塞控制机制是什么？请简单说说

我们知道 TCP 通过一个定时器（timer）采样了 RTT 并计算 RTO，但是，如果网络上的延时突然增加，那么，TCP 对这个事做出的应对只有重传数据，然而重传会导致网络的负担更重，于是会导致更大的延迟以及更多的丢包，这就导致了恶性循环，最终形成“网络风暴” —— TCP 的拥塞控制机制就是用于应对这种情况。

首先需要了解一个概念，为了在发送端调节所要发送的数据量，定义了一个“拥塞窗口”（Congestion Window），在发送数据时，将拥塞窗口的大小与接收端 ack 的窗口大小做比较，取较小者作为发送数据量的上限。

拥塞控制主要是四个算法：

1.慢启动：意思是刚刚加入网络的连接，一点一点地提速，不要一上来就把路占满。

连接建好的开始先初始化 cwnd = 1，表明可以传一个 MSS 大小的数据。

每当收到一个 ACK，cwnd++; 呈线性上升

每当过了一个 RTT，cwnd = cwnd*2; 呈指数让升

阈值 ssthresh（slow start threshold），是一个上限，当 cwnd >= ssthresh 时，就会进入“拥塞避免算法”

2.拥塞避免：当拥塞窗口 cwnd 达到一个阈值时，窗口大小不再呈指数上升，而是以线性上升，避免增长过快导致网络拥塞。

每当收到一个 ACK，cwnd = cwnd + 1/cwnd

每当过了一个 RTT，cwnd = cwnd + 1

拥塞发生：当发生丢包进行数据包重传时，表示网络已经拥塞。分两种情况进行处理：

等到 RTO 超时，重传数据包

sshthresh = cwnd /2

cwnd 重置为 1

3.进入慢启动过程

在收到 3 个 duplicate ACK 时就开启重传，而不用等到 RTO 超时

sshthresh = cwnd = cwnd /2

进入快速恢复算法——Fast Recovery

4.快速恢复：至少收到了 3 个 Duplicated Acks，说明网络也不那么糟糕，可以快速恢复。

cwnd = sshthresh + 3 * MSS （3 的意思是确认有 3 个数据包被收到了）

重传 Duplicated ACKs 指定的数据包

如果再收到 duplicated Acks，那么 cwnd = cwnd +1

如果收到了新的 Ack，那么，cwnd = sshthresh ，然后就进入了拥塞避免的算法了。

6. Mysql 连接语句如何实现快速查找?

1.like 语句（和数据库做模糊匹配）"%" 可用于定义通配符

```
SELECT * FROM Persons WHERE City LIKE 'N%'1
```

2.IN 语句（表示某个字段是否在集合中）

```
SELECT * FROM Persons WHERE LastName IN ('Adams','Carter')
```

3.BETWEEN … AND 会选取介于两个值之间的数据范围。这些值可以是数值、文本或者日期。

```
SELECT * FROM Persons WHERE LastName BETWEEN 'Adams' AND 'Carter'
```

4.Alias （给返回的字段或者查询的表设置别名）

```
SELECT po.OrderID, p.LastName, p.FirstName FROM Persons AS p,
Product_Orders AS po WHERE p.LastName='Adams' AND p.FirstName='John'
```

5.join(匹配两个表数据)

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo FROM Persons
INNER JOIN Orders ON Persons.Id_P = Orders.Id_P
```

常用几种连接语句的区别
JOIN: 如果表中有至少一个匹配，则返回行
LEFT JOIN: 即使右表中没有匹配，也从左表返回所有的行
RIGHT JOIN: 即使左表中没有匹配，也从右表返回所有的行
FULL JOIN: 只要其中一个表中存在匹配，就返回行

UNION（注意跟 join 是不同的）（UNION 操作符用于合并两个或多个 SELECT 语句的结果集。）
（默认地，UNION 操作符选取不同的值。如果允许重复的值，请使用 UNION ALL。）
注意：请注意，UNION 内部的 SELECT 语句必须拥有相同数量的列。列也必须拥有相似的数据类型。同时，每条 SELECT 语句中的列的顺序必须相同。

```
SELECT column_name(s) FROM table_name1
UNION
SELECT column_name(s) FROM table_name2
```

SELECT INTO（语句从一个表中选取数据，然后把数据插入另一个表中。）

```
SELECT * INTO Persons_backup FROM Persons
SELECT * INTO Persons IN 'Backup.mdb' FROM Persons
```

7. 输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字，例如，如果输入如下 4 X 4 矩阵： 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 则依次打印出数字 1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10.

```
public static ArrayList<Integer> printMatrix(int [][] matrix) {
    if (matrix.length == 0){
        return null;
    }
    ArrayList<Integer> list = new ArrayList<>();

    // 四个边界值
    int top = 0;
```

```
int left = 0;
int right = matrix[0].length;
int bottom = matrix.length;
// 横着和竖着相加的值（1/-1）
int colFlage = 1;
int rowFlage = 1;
//  现在是否是横着移动
boolean flag = true;
int i = 0;
int j = 0;

while (bottom != top && right != left){
    // 横向移动
    if (flag){
        for (; j < right && j >= left; j += rowFlage) {
            list.add(matrix[i][j]);
        }
        // 缩小墙壁
        if (rowFlage > 0) {
            top++;
        } else {
            bottom--;
        }
        // 指针恢复
        i += rowFlage;
        // 切换下次横向移动方向
        rowFlage = -rowFlage;
        j += rowFlage;
    }
    // 纵向移动
    if (!flag){
        for (; i < bottom && i >= top; i += colFlage) {
            list.add(matrix[i][j]);
        }
        // 缩小墙壁
        if (colFlage > 0) {
            right--;
        } else {
            left++;
        }
        // 切换下次纵向移动方向
        colFlage = -colFlage;
        // 恢复指针位置
        i += colFlage;
```

```
            j += colFlage;
        }
        // 切换横向和竖向移动
        flag = !flag;
    }

    return list;
}
```

8. 利用快速排序对单链表进行排序

```
#include<iostream>
#include<ctime>
using namespace std;

struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};

class Solution {
public:
    ListNode *sortList(ListNode *head) {
        if (head == NULL)
            return head;

        ListNode* tail=head;
        ListNode* end = tail;
        while (tail->next != NULL) {
            //if (tail->next->next == NULL)
            //  end = tail;
            tail = tail->next;

        }
        qSortList(head, tail);
        return head;
    }
    ListNode* Partition(ListNode* head, ListNode* tail)
    {

        ListNode* newHead = new ListNode(0);
        newHead->next = head;
        ListNode* ipt = newHead, *jpt = head;
```

```
    int ar = tail->val;
    while (jpt != tail)
    {
        if (jpt->val < ar)
        {
            ipt = ipt->next;
            swap(ipt->val, jpt->val);
        }
        jpt = jpt->next;
    }
    ipt = ipt->next;
    swap(ipt->val, tail->val);
    return ipt;
}
void qSortList(ListNode*head, ListNode*tail)
{

    if (head == NULL)
        return;
    if (tail == NULL)
        return;
    if (tail->next!=NULL && tail->next == head)
        return;
    else if (tail->next!=NULL &&
            tail->next->next!=NULL &&
            tail->next->next== head)
        return;

    if (head == tail)
        return;
    if (head->next == tail)
    {
        if (head->val > tail->val)
            swap(head->val,tail->val);
        return;
    }

    ListNode* mid = Partition(head, tail);
    ListNode*tail1 = head;
    if (tail1!=mid)
        while (tail1->next != mid) tail1 = tail1->next;
    ListNode*head2 = mid->next;
    qSortList(head, tail1);
    qSortList(head2, tail);
```

```
    }
};
int main()
{
    ListNode* head0 = new ListNode(200);
    ListNode* head = head0;
    srand(time(NULL));
    for (int i = 0; i < 10000; i++)
    {

        ListNode* mNode = new ListNode(rand() % 4);
        head0->next = mNode;
        head0 = head0->next;

    }
    Solution sln;
    ListNode*res=sln.sortList(head);
    while (res->next != NULL)
    {
        cout << res->val << " ";
        res = res->next;
    }
    return 0;
}
```

9. Mysql 如何实现快速查找?

```
#include<iostream>
#include<ctime>
using namespace std;

struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};

class Solution {
public:
    ListNode *sortList(ListNode *head) {
        if (head == NULL)
            return head;
```

```
    ListNode* tail=head;
    ListNode* end = tail;
    while (tail->next != NULL) {
        //if (tail->next->next == NULL)
        //  end = tail;
        tail = tail->next;

    }
    qSortList(head, tail);
    return head;
}
ListNode* Partition(ListNode* head, ListNode* tail)
{

    ListNode* newHead = new ListNode(0);
    newHead->next = head;
    ListNode* ipt = newHead, *jpt = head;
    int ar = tail->val;
    while (jpt != tail)
    {
        if (jpt->val < ar)
        {
            ipt = ipt->next;
            swap(ipt->val, jpt->val);
        }
        jpt = jpt->next;
    }
    ipt = ipt->next;
    swap(ipt->val, tail->val);
    return ipt;
}
void qSortList(ListNode*head, ListNode*tail)
{

    if (head == NULL)
        return;
    if (tail == NULL)
        return;
    if (tail->next!=NULL && tail->next == head)
        return;
    else if (tail->next!=NULL &&tail->next->next!=NULL
            &&  tail->next->next== head)
        return;
```

```
        if (head == tail)
            return;
        if (head->next == tail)
        {
            if (head->val > tail->val)
                swap(head->val,tail->val);
            return;
        }

        ListNode* mid = Partition(head, tail);
        ListNode*tail1 = head;
        if (tail1!=mid)
            while (tail1->next != mid) tail1 = tail1->next;
        ListNode*head2 = mid->next;
        qSortList(head, tail1);
        qSortList(head2, tail);
    }
};
int main()
{
    ListNode* head0 = new ListNode(200);
    ListNode* head = head0;
    srand(time(NULL));
    for (int i = 0; i < 10000; i++)
    {

        ListNode* mNode = new ListNode(rand() % 4);
        head0->next = mNode;
        head0 = head0->next;

    }
    Solution sln;
    ListNode*res=sln.sortList(head);
    while (res->next != NULL)
    {
        cout << res->val << " ";
        res = res->next;
    }
    return 0;
}
```

10. 将一个链表分为奇偶两个链表

```
#include<iostream>
#include<malloc.h>
using namespace std;
typedef struct node{
    int data;
    node *next;
}LNode;
LNode *L;
LNode* create(LNode *L, int arr[], int n){
    L = (LNode*)malloc(sizeof(LNode));
    L->next = NULL;
    LNode *p = L;
    for(int i = 0; i < n; ++i){
        LNode *newNode = (LNode*)malloc(sizeof(LNode));
        newNode->next = NULL;
        newNode->data = arr[i];
        //头插法
        p->next = newNode;
        p = p->next;
    }
    return L;
}

//分离的方法
void spilt(LNode *L, LNode *B){
    LNode *p = L;
    B = (LNode*)malloc(sizeof(LNode));
    B->next = NULL;
    LNode *q = B;
    LNode *r;
    while(p->next != NULL){
        if(p->next->data % 2 == 0){
            //下面处理的是取下链表中的偶数节点插入到链表中
            //偶数的时候取出来插入到链表 B 中
            r = p->next;
            p->next = p->next->next;
            //q 指针向下移动
            q->next = r;
            r->next = NULL;
            q = q->next;
```

```
        }else{
            p = p->next;
        }
    }
    //输出链表 A 的内容
    LNode *s = L;
    while(s->next != NULL){
        printf("%d ", s->next->data);
        s = s->next;
    }

    cout << endl;
    //输出链表 B 的内容
    s = B;
    while(s->next != NULL){
        printf("%d ", s->next->data);
        s = s->next;
    }
}

int main(void){
    int arr[12] = {12, 43, 2, 10, 100, 89, 11, 56, 45, 34, 77, 11};
    LNode *head = create(L, arr, 12);
    LNode *B;
    spilt(head, B);
    return 0;
}
```

11. 有序数组二分查找，返回查找元素最后一次出现的位置，若不存在则返回-1

```
#include<iostream>
using namespace std;

void last(int[] a, int key) {
    int min=0,max=a.length-1;
    int mid=0;
    while(min<=max){
        mid=(max+min+1)/2;
        if (key>=a[mid]) {
            min=mid;
        }else {
            max=mid-1;
```

```
            }
            if (max==min) {
                break;
            }
        }
        if (a[max]!=key) {//当查找元素不存在时，返回-1
            return -1;
        }else {
            return max;
        }
    }

    int main(void){
        int[] a={3,5,10,10,10,13,13,19,23};
        last(a,10);
        return 0;
    }
```

12. 在一个二维 01 矩阵中找到全为 1 的最大正方形，返回其面积。

样例
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
返回 4

```
public int maxSquare(int[][] matrix) {
        // write your code here

        int row = matrix.length;  //行大小
        int line = matrix[0].length;  //列大小

        //一个与 matrix 相同大小的辅助数组
        int[][] tmp = new int[row][line];

        //将 matrix 的第一行和第一列元素直接存放到
        for(int i=0;i<row;i++){
            tmp[i][0] = matrix[i][0];
        }
        for(int i=0;i<line;i++){
            tmp[0][i] = matrix[0][i];
```

```
        }

        for(int i=1;i<row;i++){
            for(int j=1;j<line;j++){
                if(matrix[i][j] == 1){
                    tmp[i][j] =
                    Math.min(Math.min(
                        tmp[i-1][j],
                        tmp[i][j-1]),
                        tmp[i-1][j-1]) + 1;
                }
                if(matrix[i][j] == 0){
                    tmp[i][j] = 0;
                }

            }
        }

        int max=0;  //记录 tmp 中最大元素的值（tmp 中元素值表示正方形的边长）
        for(int i=0;i<row;i++){
            for(int j=0;j<line;j++){
                if(tmp[i][j] > max){
                    max = tmp[i][j];
                }
            }
        }

        return max*max;
    }
```

13. 常见的 HTTP 状态码

1XX 系列响应代码仅在与 HTTP 服务器沟通时使用。

100("Continue")

这是对 HTTP LBYL（look-before-you-leap）请求的一个可能的响应。该响应代码表明：客户端应重新发送初始请求，并在请求中附上第一次请求时未提供的（可能很大或者包含敏感信息的）表示。客户端这次发送的请求不会被拒绝。对 LBYL 请求的另一个可能的响应是 417("Expectation Failed")。

请求报头：要做一个 LBYL 请求，客户端必须把 Expect 请求报头设为字符串 "100-continue"。除此以外，客户端还需要设置其他一些报头，服务器将根据这些报头决定是响应 100 还是 417。

101("Switching Protocols")

当客户端通过在请求里使用 Upgrade 报头，以通知服务器它想改用除 HTTP 协议之外的其他协议时，客户端将获得此响应代码。101 响应代码表示“行，我现在改用另一个协议了”。通常 HTTP 客户端会在收到服务器发来的 101 响应后关闭与服务器的 TCP 连接。101 响应代码意味着，该客户端不再是一个 HTTP 客户端，而将成为另一种客户端。

尽管可以通过 Upgrade 报头从 HTTP 切换到 HTTPS，或者从 HTTP1.1 切换到某个未来的版本，但实际使用 Upgrade 报头的情况比较少。Upgrade 报头也可用于 HTTP 切换到一个完全不同的协议（如 IRC）上，但那需要在 Web 服务器切换为一个 IRC 服务器的同时，Web 客户端切换为一个 IRC 的客户端，因为服务器将立刻在同一个 TCP 连接上开始使用新的协议。

请求报头：客户端把 Upgrade 报头设置为一组希望使用的协议。

响应报头：如果服务器同意切换协议，它就返回一个 Upgrade 报头，说明它将切换到那个协议，并附上一个空白行。服务器不用关闭 TCP 链接，而是直接在该 TCP 连接上开始使用新的协议。

2XX: 成功

2XX 系列响应代码表明操作成功了。

200("OK")

一般来说，这是客户端希望看到的响应代码。它表示服务器成功执行了客户端所请求的动作，并且在 2XX 系列里没有其他更适合的响应代码了。

实体主体：对于 GET 请求，服务器应返回客户端所请求资源的一个表示。对于其他请求，服务器应返回当前所选资源的一个表示，或者刚刚执行的动作的一个描述。

-201("Created")

当服务器依照客户端的请求创建了一个新资源时，发送此响应代码。

响应报头：Location 报头应包含指向新创建资源的规范 URI。

实体主体：应该给出新创建资源的描述与链接。若已经在 Location 报头里给出了新资源的 URI，那么可以用新资源的一个表示作为实体主体。

-202("Accepted")

客户端的请求无法或将不被实时处理。请求稍后会被处理。请求看上去是合法的，但在实际处理它时有出现问题的可能。

若一个请求触发了一个异步操作，或者一个需要现实世界参与的动作，或者一个需要很长时间才能完成且没必要让 Web 客户端一直等待的动作时，这个相应代码是一个合适的选择。

响应报头：应该把未处理完的请求暴露为一个资源，以便客户端稍后查询其状态。Location 报头可以包含指向该资源的 URI。

实体主体：若无法让客户端稍后查询请求的状态，那么至少应该提供一个关于何时能处理该请求的估计。

203("Non-Authoritative Information")

这个响应代码跟 200 一样，只不过服务器想让客户端知道，有些响应报头并非来自该服务器--他们可能是从客户端先前发送的一个请求里复制的，或者从第三方得到的。

响应报头：客户端应明白某些报头可能是不准确的，某些响应报头可能不是服务器自己生成的，所以服务器也不知道其含义。

204("No Content")

若服务器拒绝对 PUT、POST 或者 DELETE 请求返回任何状态信息或表示，那么通常采用此响应代码。服务器也可以对 GET 请求返回此响应代码，这表明“客户端请求的资源存在，但其表示是空的”。注意与 304("Not Modified")的区别。204 常常用在 Ajax 应用里。服务器通过这个响应代码告诉客户端：客户端的输入已被接受，但客户端不应该改变任何 UI 元素。

实体主体：不允许。

205("Reset Content")

它与 204 类似，但与 204 不同的是，它表明客户端应重置数据源的视图或数据结构。假如你在浏览器里提交一个 HTML 表单，并得到响应代码 204，那么表单里的各个字段值不变，可以继续修改它们；但假如得到的响应代码 205，那么表单里的各个字段将被重置为它们的初始值。从数据录入方面讲：204 适合对单条记录做一系列编辑，而 205 适于连续输入一组记录。

206("Partial Content")

它跟 200 类似，但它用于对部分 GET 请求（即使用 Range 请求报头的 GET 请求）的响应。部分 GET 请求常用于大型二进制文件的断点续传。

请求报头：客户端为 Range 请求报头设置一个值。

响应报头：需要提供 Date 报头。ETag 报头与 Content-Location 报头的值应该跟正常 GET 请求相同。

若实体主体是单个字节范围（byte range），那么 HTTP 响应里必须包含一个 Content-Range 报头，以说明本响应返回的是表示的哪个部分，若实体主体是一个多部分实体（multipart entity）（即该实体主体由多个字节范围构成），那么每一个部分都要有自己的 Content-Range 报头。

实体主体：不是整个表示，而是一个或者多个字节范围。

3XX 重定向

3XX 系列响应代码表明：客户端需要做些额外工作才能得到所需要的资源。它们通常用于 GET 请求。他们通常告诉客户端需要向另一个 URI 发送 GET 请求，才能得到所需的表示。那个 URI 就包含在 Location 响应报头里。

300("Multiple Choices")

若被请求的资源在服务器端存在多个表示，而服务器不知道客户端想要的是哪一个表示时，发送这个响应代码。或者当客户端没有使用 Accept-*报头来指定一个表示，或者客户端所请求的表示不存在时，也发送这个响应代码。在这种情况下，一种选择是，服务器返回一个首选表示，并把响应代码设置为 200，不过它也可以返回一个包含该资源各个表示的 URI 列表，并把响应代码设为 300。

响应报头：如果服务器有首选表示，那么它可以在 Location 响应报头中给出这个首选表示的 URI。跟其他 3XX 响应代码一样，客户端可以自动跟随 Location 中的 URI。

实体主体：一个包含该资源各个表示的 URI 的列表。可以在表示中提供一些信息，以便用户作出选择。

301("Moved Permanently")

服务器知道客户端试图访问的是哪个资源，但它不喜欢客户端用当前 URI 来请求该资源。它希望客户端记住另一个 URI，并在今后的请求中使用那个新的 URI。你可以通过这个响应代码来防止由于 URI 变更而导致老 URI 失效。

响应报头：服务器应当把规范 URI 放在 Location 响应报头里。

实体主体：服务器可以发送一个包含新 URI 的信息，不过这不是必需的。

302("Found")

重要程度：应该了解，特别市编写客户端时。但我不推荐使用它。

这个响应代码市造成大多数重定向方面的混乱的最根本原因。它应该是像 307 那样被处理。实际上，在 HTTP 1.0 中，响应代码 302 的名称是”Moved Temporarily”，不幸的是，在实际生活中，绝大多数客户端拿它像 303 一样处理。它的不同之处在于当服务器为客户端的 PUT，POST 或者 DELETE 请求返回 302 响应代码时，客户端要怎么做。

为了消除这一混淆，在 HTTP 1.1 中，该响应代码被重命名为"Found"，并新加了一个响应代码 307。这个响应代码目前仍在广泛使用，但它的含义市混淆的，所以我建议你的服务发送 307 或者 303,而不要发送 302.除非你知道正在与一个不能理解 303 或 307 的 HTTP 1.0 客户端交互。

响应报头：把客户端应重新请求的那个 URI 放在 Location 报头里。

实体主体：一个包含指向新 URI 的链接的超文本文档（就像 301 一样）。

303("See Other")

请求已经被处理，但服务器不是直接返回一个响应文档，而是返回一个响应文档的 URI。该响应文档可能是一个静态的状态信息，也可能是一个更有趣的资源。对于后一种情况，303 是一种令服务器可以“发送一个资源的表示，而不强迫客户端下载其所有数据”的方式。客户端可以向 Location 报头里的 URI 发送 GET 请求，但它不是必须这么做。

303 响应代码是一种规范化资源 URI 的好办法。一个资源可以有多个 URIs，但每个资源的规范 URI 只有一个，该资源的所有其他 URIs 都通过 303 指向该资源的规范 URI，例如：303 可以把一个对 http://www.example.com/software/current.tar.gz 的请求重定向到 http://www.example.com/software/1.0.2.tar.gz。

响应报头：Location 报头里包含资源的 URI。

实体主体：一个包含指向新 URI 的链接的超文本文档。

304("Not Modified")

这个响应代码跟 204("No Content")类似：响应实体主体都必须为空。但 204 用于没有主体数据的情况，而 304 用于有主体数据，但客户端已拥有该数据，没必要重复发送的情况。这个响应代码可用于条件 HTTP 请求（conditional HTTP request).如果客户端在发送 GET 请求时附上了一个值为 Sunday 的 If-Modified-Since 报头，而客户端所请求的表示在服务器端自星期日（Sunday）以来一直没有改变过，那么服务器可以返回一个 304 响应。服务器也可以返回一个 200 响应，但由于客户端已拥有该表示，因此重复发送该表示只会白白浪费宽带。

响应报头：需要提供 Date 报头。Etag 与 Content-Location 报头的值，应该跟返回 200 响应时的一样。若 Expires, Cache-Control 及 Vary 报头的值自上次发送以来已经改变，那么就要提供这些报头。

实体主体：不允许。

305("Use Proxy")

这个响应代码用于告诉客户端它需要再发一次请求，但这次要通过一个 HTTP 代理发送，而不是直接发送给服务器。这个响应代码使用的不多，因为服务器很少在意客户端是否使用某一特定代理。这个代码主要用于基于代理的镜像站点。现在，镜像站点（如 http://www.example.com.mysite.com/）包含跟原始站点（如 http://www.example.com/）一样的内容，但具有不同的 URI，原始站点可以通过 307 把客户端重新定向到镜像站点上。假如有基于代理的镜像站点，那么你可以通过把 http://proxy.mysite.com/设为代理，使用跟原始 URI（http://www.example.com/）一样的 URI 来访问镜像站点。这里，原始站点 example.com 可以通过 305 把客户端路由到一个地理上接近客户端的镜像代理。web 浏览器一般不能正确处理这个响应代码，这是导致 305 响应代码用的不多的另一个原因。

响应报头：Location 报头里包含代理的 URI。

306 未使用

重要程度：无

306 响应代码没有在 HTTP 标准中定义过。

307("Temporary Redirect")

请求还没有被处理，因为所请求的资源不在本地：它在另一个 URI 处。客户端应该向那个 URI 重新发送请求。就 GET 请求来说，它只是请求得到一个表示，该响应代码跟 303 没有区别。当服务器希望把客户端重新定向到一个镜像站点时，可以用 307 来响应 GET 请求。但对于 POST，PUT 及 DELETE 请求，它们希望服务器执行一些操作，307 和 303 有显著区别。对 POST，PUT 或者 DELETE 请求响应 303 表明：操作已经成功执行，但响应实体将不随本响应一起返回，若客户端想要获取响应实体主体，它需要向另一个 URI 发送 GET 请求。而 307 表明：服务器尚未执行操作，客户端需要向 Location 报头里的那个 URI 重新提交整个请求。

响应报头： 把客户端应重新请求的那个 URI 放在 Location 报头里。

实体主体：一个包含指向新 URI 的链接的超文本文档。

4XX：客户端错误

这些响应代码表明客户端出现错误。不是认证信息有问题，就是表示格式或 HTTP 库本身有问题。客户端需要自行改正。

400("Bad Request")

这是一个通用的客户端错误状态，当其他 4XX 响应代码不适用时，就采用 400。此响应代码通常用于“服务器收到客户端通过 PUT 或者 POST 请求提交的表示，表示的格式正确，但服务器不懂它什么意思”的情况。

实体主体：可以包含一个错误的描述文档。

401("Unauthorized")

客户端试图对一个受保护的资源进行操作，却又没有提供正确的认证证书。客户端提供了错误的证书，或者根本没有提供证书。这里的证书（credential）可以是一个用户名/密码，也可以市一个 API key，或者一个认证令牌。客户端常常通过向一个 URI 发送请求，并查看收到 401 响应，以获知应该发送哪种证书，以及证书的格式。如果服务器不想让未授权的用户获知某个资源的存在，那么它可以谎报一个 404 而不是 401。这样做的缺点是：客户

端需要事先知道服务器接受哪种认证--这将导致 HTTP 摘要认证无法工作。

响应报头：WWW-Authenticate 报头描述服务器将接受哪种认证。

实体主体：一个错误的描述文档。假如最终用户可通过“在网站上注册”的方式得到证书，那么应提供一个指向该注册页面的链接。

402("Payment Required")

除了它的名字外，HTTP 标准没有对该响应的其他方面作任何定义。因为目前还没有用于 HTTP 的微支付系统，所以它被留作将来使用。尽管如此，若存在一个用于 HTTP 的微支付系统，那么这些系统将首先出现在 web 服务领域。如果想按请求向用户收费，而且你与用户之间的关系允许这么做的话，那么或许用得上这个响应代码。

注：该书印于 2008 年

403("Forbidden")

客户端请求的结构正确，但是服务器不想处理它。这跟证书不正确的情况不同--若证书不正确，应该发送响应代码 401。该响应代码常用于一个资源只允许在特定时间段内访问，

或者允许特定 IP 地址的用户访问的情况。403 暗示了所请求的资源确实存在。跟 401 一样，若服务器不想透露此信息，它可以谎报一个 404。既然客户端请求的结构正确，那为什么还要把本响应代码放在 4XX 系列（客户端错误），而不是 5XX 系列（服务端错误）呢？因为服务器不是根据请求的结构，而是根据请求的其他方面（比如说发出请求的时间）作出的决定的。

实体主体：一个描述拒绝原因的文档（可选）。

404("Not Found")

这也许是最广为人知的 HTTP 响应代码了。404 表明服务器无法把客户端请求的 URI 转换为一个资源。相比之下，410 更有用一些。web 服务可以通过 404 响应告诉客户端所请求的 URI 是空的，然后客户端就可以通过向该 URI 发送 PUT 请求来创建一个新资源了。但是 404 也有可能是用来掩饰 403 或者 401.

405("Method Not Allowd")

客户端试图使用一个本资源不支持的 HTTP 方法。例如：一个资源只支持 GET 方法，但是客户端使用 PUT 方法访问。

响应报头：Allow 报头列出本资源支持哪些 HTTP 方法，例如：Allow：GET，POST

406("Not Acceptable")

当客户端对表示有太多要求，以至于服务器无法提供满足要求的表示，服务器可以发送这个响应代码。例如：客户端通过 Accept 头指定媒体类型为 application/json+hic，但是服务器只支持 application/json。服务器的另一个选择是：忽略客户端挑剔的要求，返回首选表示，并把响应代码设为 200。

实体主体：一个可选表示的链接列表。

407("Proxy Authentication Required")

只有 HTTP 代理会发送这个响应代码。它跟 401 类似，唯一区别在于：这里不是无权访问 web 服务，而是无权访问代理。跟 401 一样，可能是因为客户端没有提供证书，也可能是客户端提供的证书不正确或不充分。

请求报头：客户端通过使用 Proxy-Authorization 报头（而不是 Authorization）把证书提供给代理。格式跟 Authrization 一样。

响应报头：代理通过 Proxy-Authenticate 报头（而不是 WWW-Authenticate）告诉客户端它接受哪种认证。格式跟 WWW-Authenticate 一样。

408("Reqeust Timeout")

假如 HTTP 客户端与服务器建立链接后，却不发送任何请求（或从不发送表明请求结束的空白行），那么服务器最终应该发送一个 408 响应代码，并关闭此连接。

409("Conflict")

此响应代码表明：你请求的操作会导致服务器的资源处于一种不可能或不一致的状态。例如你试图修改某个用户的用户名，而修改后的用户名与其他存在的用户名冲突了。

响应报头：若冲突是因为某个其他资源的存在而引起的，那么应该在 Location 报头里给出那个资源的 URI。

实体主体：一个描述冲突的文档，以便客户端可以解决冲突。

410("Gone")

这个响应代码跟 404 类似，但它提供的有用信息更多一些。这个响应代码用于服务器知道被请求的 URI 过去曾指向一个资源，但该资源现在不存在了的情况。服务器不知道

该资源的新 URI，服务器要是知道该 URI 的话，它就发送响应代码 301.410 和 310 一样，都有暗示客户端不应该再请求该 URI 的意思，不同之处在于：410 只是指出该资源不存在，但没有给出该资源的新 URI。RFC2616 建议“为短期的推广服务，以及属于个人但不继续在服务端运行的资源”采用 410.

411("Length Required")

若 HTTP 请求包含表示，它应该把 Content-Length 请求报头的值设为该表示的长度（以字节为单位）。对客户端而言，有时这不太方便（例如，当表示是来自其他来源的字节流时）。

所以 HTTP 并不要求客户端在每个请求中都提供 Content-Length 报头。但 HTTP 服务器可以要求客户端必须设置该报头。服务器可以中断任何没有提供 Content-Length 报头的请求，并要求客户端重新提交包含 Content-Length 报头的请求。这个响应代码就是用于中断未提供 Content-Lenght 报头的请求的。假如客户端提供错误的长度，或发送超过长度的表示，服务器可以中断请求并关闭链接，并返回响应代码 413。

412("Precondition Failed")

客户端在请求报头里指定一些前提条件，并要求服务器只有在满足一定条件的情况下才能处理本请求。若服务器不满足这些条件，就返回此响应代码。If-Unmodified-Since 是一个常见的前提条件。客户端可以通过 PUT 请求来修改一个资源，但它要求，仅在自客户端最后一次获取该资源后该资源未被别人修改过才能执行修改操作。若没有这一前提条件，客户端可能会无意识地覆盖别人做的修改，或者导致 409 的产生。

请求报头：若客户但设置了 If-Match，If-None-Match 或 If-Unmodified-Since 报头，那就有可能得到这个响应代码。If-None-Match 稍微特别一些。若客户端在发送 GET 或 HEAD 请求时指定了 If-None-Match，并且服务器不满足该前提条件的话，那么响应代码不是 412 而是 304，这是实现条件 HTTP GET 的基础。若客户端在发送 PUT，POST 或 DELETE 请求时指

定了 If-None-Match,并且服务器不满足该前提条件的话，那么响应代码是 412.另外，若客户端指定了 If-Match 或 If-Unmodified-Since(无论采用什么 HTTP 方法)，而服务器不满足该前提条件的话，响应代码也是 412。

413("Request Entity Too Large")

这个响应代码跟 411 类似，服务器可以用它来中断客户端的请求并关闭连接，而不需要等待请求完成。411 用于客户端未指定长度的情况，而 413 用于客户端发送的表示太大，以至于服务器无法处理。客户端可以先做一个 LBYL（look-before-you-leap）请求，以免请求被 413 中断。若 LBYL 请求获得响应代码为 100，客户端再提交完整的表示。

响应报头：如果因为服务器方面临时遇到问题（比如资源不足），而不是因为客户端方面的问题而导致中断请求的话,服务器可以把 Retry-After 报头的值设为一个日期或一个间隔时间，以秒为单位，以便客户端可以过段时间重试。

414("Request-URI Too Long")

HTTP 标准并没有对 URI 长度作出官方限制，但大部分现有的 web 服务器都对 URI 长度有一个上限，而 web 服务可能也一样。导致 URI 超长的最常见的原因是：表示数据明明是该放在实体主体里的，但客户端却把它放在了 URI 里。深度嵌套的数据结构也有可能引起 URI 过长。

415("Unsupported Media Type")

当客户端在发送表示时采用了一种服务器无法理解的媒体类型，服务器发送此响应代码。比如说，服务器期望的是 XML 格式，而客户端发送的确实 JSON 格式。

如果客户端采用的媒体类型正确，但格式有问题，这时最好返回更通用的 400。

416("Requestd Range Not Satisfiable")

当客户端所请求的字节范围超出表示的实际大小时，服务器发送此响应代码。例如：你请求一个表示的 1-100 字节，但该表示总共只用 99 字节大小。

请求报头：仅当原始请求里包含 Range 报头时，才有可能收到此响应代码。若原始请求提供的是 If-Range 报头，则不会收到此响应代码。

响应报头：服务器应当通过 Content-Range 报头告诉客户端表示的实际大小。

417("Expectation Failed")

此响应代码跟 100 正好相反。当你用 LBYL 请求来考察服务器是否会接受你的表示时，如果服务器确认会接受你的表示，那么你将获得响应代码 100，否则你将获得 417。

5XX 服务端错误

这些响应代码表明服务器端出现错误。一般来说，这些代码意味着服务器处于不能执行客户端请求的状态，此时客户端应稍后重试。有时，服务器能够估计客户端应在多久之后重试。并把该信息放在 Retry-After 响应报头里。

5XX 系列响应代码在数量上不如 4XX 系列多，这不是因为服务器错误的几率小，而是因为没有必要如此详细--对于服务器方面的问题，客户端是无能为力的。

500("Internal Server Error")

这是一个通用的服务器错误响应。对于大多数 web 框架，如果在执行请求处理代码时遇到了异常，它们就发送此响应代码。

501("Not Implemented")

客户端试图使用一个服务器不支持的 HTTP 特性。

最常见的例子是：客户端试图做一个采用了拓展 HTTP 方法的请求，而普通 web 服务器不支持此请求。它跟响应代码 405 比较相似，405 表明客户端所用的方法是一个可识别的方法，但该资源不支持，而 501 表明服务器根本不能识别该方法。

502("Bad Gateway")

只有 HTTP 代理会发送这个响应代码。它表明代理方面出现问题，或者代理与上行服务器之间出现问题，而不是上行服务器本身有问题。若代理根本无法访问上行服务器，响应代码将是 504。

503("Service Unavailable")

此响应代码表明 HTTP 服务器正常，只是下层 web 服务服务不能正常工作。最可能的原因是资源不足：服务器突然收到太多请求，以至于无法全部处理。由于此问题多半由客户端反复发送请求造成，因此 HTTP 服务器可以选择拒绝接受客户端请求而不是接受它，并发送 503 响应代码。

响应报头：服务器可以通过 Retry-After 报头告知客户端何时可以重试。

504("Gateway Timeout")

跟 502 类似，只有 HTTP 代理会发送此响应代码。此响应代码表明代理无法连接上行服务器。

505("HTTP Version Not Supported")

14.25 匹马，5 个赛道，最少赛多少次找出前三没有计时，只有先后.

7 次。理由如下:

1.先分开赛 5 组（A-E）， 5 次， 每组的最后两名肯定会被淘汰，（-10）。

2.5 组第一名赛一次，假设 A1 > B1 > C1 > D1＞E1，那么 A1 肯定是总体第一名。则 D，E 全部被淘汰(-6）. 现在需要在剩下的里面取 2 个,那么 C2,C3,B3 也会被淘汰(-3) 。

3.那么就剩下 A2，A3，B1，B2，C1 了，再赛一次，取前两名（-3)。

最多 7 次比赛，前 5 次总共淘汰 10 匹，第 6 次淘汰 9 匹，第 7 次淘汰 3 匹。 总共淘汰 22 匹。

15. 计算机为什么能识别二进制机器码?

1、计算机的理论基础

布尔代数是计算机的理论基础，

Boolean（布尔运算）通过对两个以上的物体进行并集、差集、交集的运算，从而得到新的物体形态。系统提供了 4 种布尔运算方式：Union（并集）、Intersection（交集）和 Subtraction（差集，包括 A-B 和 B-A 两种）。

1）与逻辑和乘法

乘法原理中自变量是因变量成立的必要条件，与逻辑的定义正好和乘法原理的描述一致，所以与逻辑和乘法对应。

2）或逻辑和加法

加法原理中自变量是因变量成立的充分条件，或逻辑的定义正好和加法原理的描述一致，所以或逻辑和加法对应。

乘法就是广义的与逻辑运算，加法就是广义的或逻辑运算。与逻辑运算可以看作是乘法的特例。或逻辑运算可以看作是加法的特例。

总之，乘法原理、加法原理可以看作是与逻辑和或逻辑的定量表述；与逻辑和或逻辑可以看作是乘法原理、加法原理的定性表述。

通俗来讲：这是一门运用”与“”或“”“非”“假”“真”来描述任意两个量（可以是任何具体事物的或者抽象概念）的逻辑关系。

2、逻辑代数与计算机电路

应用于逻辑中，解释 0 为假，1 为真，∧ 为与，∨ 为或，¬为非。涉及变量和布尔运算的表达式代表了陈述形式，两个这样的表达式可以使用上面的公理证实为等价的，当且仅当对应的陈述形式是逻辑等价的。由于逻辑代数小的逻辑单元与二进制高度契合，再加上电路最为简单的开和关恰好也对应 0 和 1，于是就有了依据逻辑代数理论创建一系列的电路在表达基础的逻辑理论，这就是计算机具有判断、计算能力的基础。

3、二进制机器识别过程

根据前面两点可以知道，如果选用二进制原理作为计算机的判断计算依据，将会使得电路制造的实现成为可能，但是自然界是不存在二进制的，为了处理这个问题，统一人为规定将其他非二进制数据表示成二进制机器码，供计算机读取。然而。随着对数据的处理能力要求越来越高，处理数据也越来越大，为了解决这个问题，汇编器出现，替代了将非二进制数据转化为二进制数据，但是这远远不足，为了更好处理，直接将硬件与汇编器组合，单独发展更高级汇编器（实质就是现在熟知的各类程序），这样，硬件与软件彻底分开。实质上就是将数据转化与判断和数据的录入、存储、输出彻底分开，使计算机的使用者可以完全不必再关注计算机的具体运算。

也就是说，计算机为什么能够识别二进制机器码，是因为有以逻辑代数原理制造的数字电路，为什么选用二进制已经解释过了。同时，也应该明白，为何程序会出现假，1 为真，∧ 为与，∨ 为或，¬为非这五个元素，算法为何而来，指的就是优化数据之间的逻辑代数关系。

16. 如何理解 mac 寻址?

MAC 地址是网卡的物理地址，每块网卡都有一个属于自己的独有的 MAC 地址，如 00-0A-EB-97-5F-65。

IP 地址就是我们常见的 202.106.46.151 192.168.1.1 这样的。

通过 MAC 地址寻找主机是 MAC 地址寻址，通过 IP 地址寻找主机叫 IP 地址寻址。它们适用于不同的协议层。

17. 如何理解 IO 多路复用的三种机制 Select，Poll，Epoll?

1.Select

首先先分析一下 select 函数

```
int select(
    int maxfdp1,
    fd_set *readset,
    fd_set *writeset,
    fd_set *exceptset,
    const struct timeval *timeout
);
```

【参数说明】

int maxfdp1 指定待测试的文件描述字个数，它的值是待测试的最大描述字加 1。

fd_set *readset , fd_set *writeset , fd_set *exceptset

fd_set 可以理解为一个集合，这个集合中存放的是文件描述符(file descriptor)，即文件句柄。中间的三个参数指定我们要让内核测试读、写和异常条件的文件描述符集合。如果对某一个的条件不感兴趣，就可以把它设为空指针。

const struct timeval *timeout timeout 告知内核等待所指定文件描述符集合中的任何一个就绪可花多少时间。其 timeval 结构用于指定这段时间的秒数和微秒数。

【返回值】

int 若有就绪描述符返回其数目，若超时则为 0，若出错则为-1

select 运行机制

select()的机制中提供一种 fd_set 的数据结构，实际上是一个 long 类型的数组，每一个数组元素都能与一打开的文件句柄(不管是 Socket 句柄,还是其他文件或命名管道或设备句柄）建立联系，建立联系的工作由程序员完成，当调用 select()时，由内核根据 IO 状态修改 fd_set 的内容，由此来通知执行了 select()的进程哪一 Socket 或文件可读。

从流程上来看，使用 select 函数进行 IO 请求和同步阻塞模型没有太大的区别，甚至还多了添加监视 socket，以及调用 select 函数的额外操作，效率更差。但是，使用 select 以后最大的优势是用户可以在一个线程内同时处理多个 socket 的 IO 请求。用户可以注册多个 socket，然后不断地调用 select 读取被激活的 socket，即可达到在同一个线程内同时处理多个 IO 请求的目的。而在同步阻塞模型中，必须通过多线程的方式才能达到这个目的。

select 机制的问题

每次调用 select，都需要把 fd_set 集合从用户态拷贝到内核态，如果 fd_set 集合很大时，那这个开销也很大

同时每次调用 select 都需要在内核遍历传递进来的所有 fd_set，如果 fd_set 集合很大时，那这个开销也很大

为了减少数据拷贝带来的性能损坏，内核对被监控的 fd_set 集合大小做了限制，并且这个是通过宏控制的，大小不可改变(限制为 1024).

2.Poll

poll 的机制与 select 类似，与 select 在本质上没有多大差别，管理多个描述符也是

进行轮询，根据描述符的状态进行处理，但是 poll 没有最大文件描述符数量的限制。也就是说，poll 只解决了上面的问题 3，并没有解决问题 1，2 的性能开销问题。

下面是 pll 的函数原型：

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
typedef struct pollfd {
        int fd;                           // 需要被检测或选择的文件描述符
        short events;                     // 对文件描述符 fd 上感兴趣的事件
        short revents;                    // 文件描述符 fd 上当前实际发生的事件
} pollfd_t;
```

poll 改变了文件描述符集合的描述方式，使用了 pollfd 结构而不是 select 的 fd_set 结构，使得 poll 支持的文件描述符集合限制远大于 select 的 1024

【参数说明】

struct pollfd *fds fds 是一个 struct pollfd 类型的数组，用于存放需要检测其状态的 socket 描述符，并且调用 poll 函数之后 fds 数组不会被清空；一个 pollfd 结构体表示一个被监视的文件描述符，通过传递 fds 指示 poll() 监视多个文件描述符。其中，结构体的 events 域是监视该文件描述符的事件掩码，由用户来设置这个域，结构体的 revents 域是文件描述符的操作结果事件掩码，内核在调用返回时设置这个域

nfds_t nfds 记录数组 fds 中描述符的总数量

【返回值】

int 函数返回 fds 集合中就绪的读、写，或出错的描述符数量，返回 0 表示超时，返回 -1 表示出错；

Epoll

epoll 在 Linux2.6 内核正式提出，是基于事件驱动的 I/O 方式，相对于 select 来说，epoll 没有描述符个数限制，使用一个文件描述符管理多个描述符，将用户关心的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的 copy 只需一次。

Linux 中提供的 epoll 相关函数如下：

```
int epoll_create(int size);
int epoll_ctl(
    int epfd,
    int op, int fd,
    struct epoll_event *event
);
int epoll_wait(
    int epfd,
    struct epoll_event * events,
    int maxevents,
    int timeout
);
```

1).epoll_create 函数创建一个 epoll 句柄，参数 size 表明内核要监听的描述符数量。调用成功时返回一个 epoll 句柄描述符，失败时返回-1。

2).epoll_ctl 函数注册要监听的事件类型。四个参数解释如下：

epfd 表示 epoll 句柄
op 表示 fd 操作类型，有如下 3 种
EPOLL_CTL_ADD 注册新的 fd 到 epfd 中
EPOLL_CTL_MOD 修改已注册的 fd 的监听事件
EPOLL_CTL_DEL 从 epfd 中删除一个 fd
fd 是要监听的描述符
event 表示要监听的事件
epoll_event 结构体定义如下：

```
struct epoll_event {
    __uint32_t events;  /* Epoll events */
    epoll_data_t data;  /* User data variable */
};
typedef union epoll_data {
    void *ptr;
    int fd;
    __uint32_t u32;
    __uint64_t u64;
} epoll_data_t;
```

3). epoll_wait 函数等待事件的就绪，成功时返回就绪的事件数目，调用失败时返回 -1，等待超时返回 0。
⑴epfd 是 epoll 句柄
⑵events 表示从内核得到的就绪事件集合
⑶maxevents 告诉内核 events 的大小
⑷timeout 表示等待的超时事件

epoll 是 Linux 内核为处理大批量文件描述符而作了改进的 poll，是 Linux 下多路复用 IO 接口 select/poll 的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统 CPU 利用率。原因就是获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核 IO 事件异步唤醒而加入 Ready 队列的描述符集合就行了。
epoll 除了提供 select/poll 那种 IO 事件的水平触发（Level Triggered）外，还提供了边缘触发（Edge Triggered），这就使得用户空间程序有可能缓存 IO 状态，减少 epoll_wait/epoll_pwait 的调用，提高应用程序效率。

⑴水平触发（LT）：默认工作模式，即当 epoll_wait 检测到某描述符事件就绪并通知应用程序时，应用程序可以不立即处理该事件；下次调用 epoll_wait 时，会再次通知此事件
⑵边缘触发（ET）： 当 epoll_wait 检测到某描述符事件就绪并通知应用程序时，应用程序必须立即处理该事件。如果不处理，下次调用 epoll_wait 时，不会再次通知此事件。（直到你做了某些操作导致该描述符变成未就绪状态了，也就是说边缘触发只在状态由未就绪变为就绪时只通知一次）。

LT 和 ET 原本应该是用于脉冲信号的，可能用它来解释更加形象。Level 和 Edge 指的就是触发点，Level 为只要处于水平，那么就一直触发，而 Edge 则为上升沿和下降沿的时候触发。比如：0->1 就是 Edge，1->1 就是 Level。

ET 模式很大程度上减少了 epoll 事件的触发次数，因此效率比 LT 模式下高。

18. 给定一个数，算出平均值是整数的算法。例如：5 平均 2 份，则结果为，2 和 3；8 平均为 3 份，则为 2，3，3；11 平均 4 份，则为 2，3，3，3；15 平均 3 份，则为：5，5，5。

```
public class Divide {
  public static int sum(int[] res) {
    int sum=0;
    for(int i:res) {
      sum += i;
    }
    return sum;
  }
  public static void aveDivide(int m,int n) {
    if(n <= 0) {
      return;
    }
    int res [] = new int[n];
    int i = 0;
    while(true) {
      if(sum(res) == m) {
        break;
      } else {
        res[i%n] += 1;
        i++;
      }
    }
  }
  public static void main(String[] args) {
    aveDivide(15,10);
  }
}
```

19. linux 内核调度详细说一下

1.1、调度策略

定义位于 linux/include/uapi/linux/sched.h 中

SCHED_NORMAL：普通的分时进程，使用的 fair_sched_class 调度类

SCHED_FIFO：先进先出的实时进程。当调用程序把 CPU 分配给进程的时候，它把该进程描述符保留在运行队列链表的当前位置。此调度策略的进程一旦使用 CPU 则一直运行。如果

没有其他可运行的更高优先级实时进程，进程就继续使用 CPU，想用多久就用多久，即使还有其他具有相同优先级的实时进程处于可运行状态。使用的是 rt_sched_class 调度类。

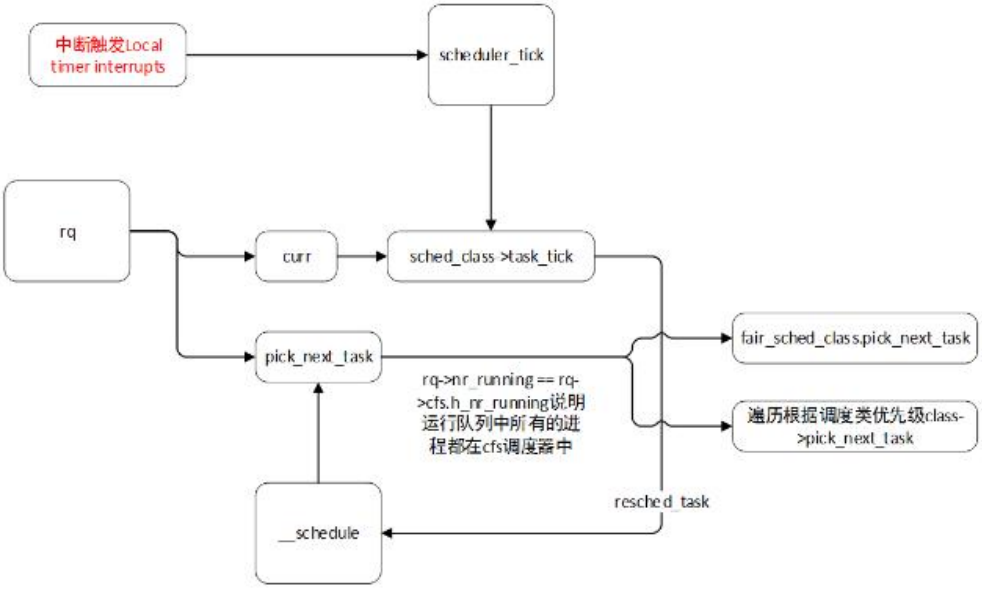
SCHED_RR：时间片轮转的实时进程。当调度程序把 CPU 分配给进程的时候，它把该进程的描述符放在运行队列链表的末尾。这种策略保证对所有具有相同优先级的 SCHED_RR 实时进程进行公平分配 CPU 时间，使用的 rt_sched_class 调度类

SCHED_BATCH：是 SCHED_NORMAL 的分化版本。采用分时策略，根据动态优先级，分配 CPU 资源。在有实时进程的时候，实时进程优先调度。但针对吞吐量优化，除了不能抢占外与常规进程一样，允许任务运行更长时间，更好使用高速缓存，适合于成批处理的工作，使用的 fair_shed_class 调度类

SCHED_IDLE：优先级最低，在系统空闲时运行，使用的是 idle_sched_class 调度类，给 0 号进程使用

SCHED_DEADLINE：新支持的实时进程调度策略，针对突发型计算，并且对延迟和完成时间敏感的任务使用，基于 EDF（earliest deadline first）,使用的是 dl_sched_class 调度类。

1.2、调度触发



调度的触发主要有两种方式，一种是本地定时中断触发调用 scheduler_tick 函数，然后使用当前运行进程的调度类中的 task_tick，另外一种则是主动调用 schedule，不管是哪一种最终都会调用到__schedule 函数，该函数调用 pick_netx_task，通过 rq->nr_running ==rq->cfs.h_nr_running 判断出如果当前运行队列中的进程都在 cfs 调度器中，则直接调用 cfs 的调度类(内核代码里面这一判断使用了 likely 说明大部分情况都是满足该条件的)。如果运行队列不都在 cfs 中，则通过优先级 stop_sched_class->dl_sched_class->rt_sched_class->fair_sched_class->idle_sched_class 遍历选出下一个需要运行的进程。然后进程任务切换。

处于 TASK_RUNNING 状态的进程才会被进程调度器选择，其他状态不会进入调度器。系统发生调度的时机如下：

à调用 cond_resched()时

à显式调用 schedule()时

à从中断上下文返回时

当内核开启抢占时，会多出几个调度时机如下：

à在系统调用或者中断上下文中调用 preemt_enable()时（多次调用系统只会在最后一次调用时会调度）

à在中断上下文中，从中断处理函数返回到可抢占的上下文时

2、CFS 调度

该部分代码位于 linux/kernel/sched/fair.c 中

定义了 const struct sched_classfair_sched_class，这个是 CFS 的调度类定义的对象。其中基本包含了 CFS 调度的所有实现。

CFS 实现三个调度策略：

1> SCHED_NORMAL 这个调度策略是被常规任务使用

2> SCHED_BATCH 这个策略不像常规的任务那样频繁的抢占，以牺牲交互性为代价下，因而允许任务运行更长的时间以更好的利用缓存，这种策略适合批处理

3> SCHED_IDLE 这是 nice 值甚至比 19 还弱，但是为了避免陷入优先级导致问题，这个问题将会死锁这个调度器，因而这不是一个真正空闲定时调度器

CFS 调度类：

n enqueue_task(…) 当任务进入 runnable 状态，这个回调将把这个任务的调度实体（entity）放入红黑树并且增加 nr_running 变量的值

n dequeue_task(…) 当任务不再是 runnable 状态，这个回调将会把这个任务的调度实体从红黑树中取出，并且减少 nr_running 变量的值

n yield_task(…) 除非 compat_yield sysctl 是打开的，这个回调函数基本上就是一个 dequeue 后跟一个 enqueue，这那种情况下，他将任务的调度实体放入红黑树的最右端

n check_preempt_curr(…) 这个回调函数是检查一个任务进入 runnable 状态是否应该抢占当前运行的任务

n pick_next_task(…) 这个回调函数选出下一个最合适运行的任务

n set_curr_task(…) 当任务改变他的调度类或者改变他的任务组，将调用该回调函数

n task_tick(…) 这个回调函数大多数是被 time tick 调用。他可能引起进程切换。这就驱动了运行时抢占

2.1、CFS 调度

Tcik 中断，主要会更新调度信息，然后调整当前进程在红黑树中的位置。调整完成以后如果当前进程不再是最左边的叶子，就标记为 Need_resched 标志，中断返回时就会调用 scheduler()完成切换、否则当前进程继续占用 CPU。从这里可以看出 CFS 抛弃了传统时间片概念。Tick 中断只需要更新红黑树。

红黑树键值即为 vruntime，该值通过调用 update_curr 函数进行更新。这个值为 64 位的变量，会一直递增，__enqueue_entity 中会将 vruntime 作为键值将要入队的实体插入到红黑树中。__pick_first_entity 会将红黑树中最左侧即 vruntime 最小的实体取出。

21. 如何从 10 亿数据中找到前 1000 大的数？

```
public class TopN {
    //  当前节点的父节点
```

```
private int parent(int n){
    return (n-1)/2;
}
//  当前节点的左子节点
private int left(int n){
    return 2*n+1;
}
//  当前节点的右子节点
private int right(int n){
    return 2*n+2;
}
//  构建堆
private void buildHeap(int n, int[] data){
    for(int i=1;i<n;i++){
        int t=i;
        while(t!=0 && data[parent(t)]>data[t]){
            int temp = data[parent(t)];
            data[parent(t)]=data[t];
            data[t]=temp;
            t=parent(t);
        }
    }
}
//  调整堆，为小顶堆
private void adjust(int i, int n, int[] data){
    if(data[0]>=data[i]){
        return;
    }
    int temp = data[i];
    data[i] = data[0];
    data[0] = temp;
    int t=0;
    //  调整时，堆顶比子节点大，从较小的子节点开始调整
    while((left(t)<n&&data[t]>data[left(t)])||
            (right(t)<n&&data[t]>data[right(t)])){
        if((right(t) < n && data[right(t)] < data[left(t)])){
            temp=data[t];
            data[t]=data[right(t)];
            data[right(t)]=temp;
            t=right(t);
        }else{
            temp=data[t];
            data[t]=data[left(t)];
            data[left(t)]=temp;
```

```
                t=left(t);
            }
        }
    }
    //  寻找 topN 数
    public void findTopN(int n, int[] data){
        buildHeap(n,data);
        for(int i=n;i<data.length;i++){
            adjust(i,n,data);
        }
    }
    //  打印
    public void print(int[] data){
        for(int i=0;i<data.length;i++){
            System.out.print(data[i]+",");
        }
    }
}
```

22. 用 UDP 通讯如何保证对方百分百收到数据？

在 UDP 通讯中，当你的数据包发出去后，至于对方有没有正确收到数据，并不知道，那么，如何保证你发出去的数据，对方一定能收到呢？？？我们可以借鉴 TCP 协议的做法（回复+重发+编号　机制）

1）接收方收到数据后，回复一个确认包，如果你不回复，那么发送端是不会知道接收方是否成功收到数据的

比如 A 要发数据“{data}”到 B，那 B 收到后，可以回复一个特定的确认包“{OK}”，表示成功收到。

但是如果只做上面的回复处理，还是有问题，比如 B 收到数据后回复给 A 的数据"{OK}"的包，A 没收到，怎么办呢？？？

2）当 A 没有收到 B 的"{OK}"包后，要做定时重发数据，直到成功接收到确认包为止，再发下面的数据，当然，重发了一定数量后还是没能收到确认包，可以执行一下 ARP 的流程，防止对方网卡更换或别的原因。

但是这样的话，B 会收到很多重复的数据，假如每次都是 B 回复确认包 A 收不到的话。

3）发送数据的包中加个标识符，比如 A 要发送的数据"{标识符|data}"到 B，B 收到后，先回复“{OK}"确认包，再根据原有的标识符进行比较，如果标识符相同，则数据丢失，如果不相同，则原有的标识符 = 接收标识符，且处理数据。

当 A 发送数据包后，没有收到确认包，则每隔 x 秒，把数据重发一次，直到收到确认包后，更新一下标识符，再进行后一包的数据发送。

经过上面 1），2），3）点的做法，则可以保证数据百分百到达对方，当然，标识符用 ID 号来代替更好。

23. 如何初始化一个指针数组?

例如:定义一个指针数组*a[5]
让其各个指针头地址分别指向
{1,1,1,1,1}，{2,2,2,2,2}，{3,3,3,3,3}，{4,4,4,4,4}，{5,5,5,5,5}
如何初始化？

```
int a[5]={....}
int* pt[5];
for(i=0;i<5;i++)
{
    pt[i] = &a[i];
}
```

指针数组不可以直接赋值，因为指针本质上是一组地址。但可以通　过“指向”赋值

24. 你所知道的设计模式有哪些？

有 23 种设计模式，不需要所有的回答，但是其中常用的几种设计模式应该去掌握。

总体来说设计模式分为三大类：

创建型模式共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

这里主要讲解简单工厂模式,代理模式,适配器模式,单例模式 4 中设计模式.

1、简单工厂模式

主要特点是需要在工厂类中做判断，从而创造相应的产品，当增加新产品时，需要修改工厂类。使用简单工厂模式，我们只需要知道具体的产品型号就可以创建一个产品。

缺点：工厂类集中了所有产品类的创建逻辑，如果产品量较大，会使得工厂类变的非常臃肿。

2、代理模式

代理模式：为其它对象提供一种代理以控制这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象,而代理对象可以在客户端和目标对象之间起到中介作用。

优点：

职责清晰。真实的角色只负责实现实际的业务逻辑，不用关心其它非本职责的事务，通过后期的代理完成具体的任务。这样代码会简洁清晰。

代理对象可以在客户端和目标对象之间起到中介的作用，这样就保护了目标对象。

扩展性好。

3、适配器模式

适配器模式可以将一个类的接口转换成客户端希望的另一个接口,使得原来由于接口不

兼容而不能在一起工作的那些类可以在一起工作。通俗的讲就是当我们已经有了一些类，而这些类不能满足新的需求，此时就可以考虑是否能将现有的类适配成可以满足新需求的类。适配器类需要继承或依赖已有的类，实现想要的目标接口。

缺点：过多地使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

4、单例模式

单例模式顾名思义，保证一个类仅可以有一个实例化对象，并且提供一个可以访问它的全局接口。实现单例模式必须注意一下几点：

单例类只能由一个实例化对象。

单例类必须自己提供一个实例化对象。

单例类必须提供一个可以访问唯一实例化对象的接口。

单例模式分为懒汉和饿汉两种实现方式。

25. 如何计算 struct 占用的内存?

1.每个成员按其类型大小和指定对齐参数 n 中较小的一个进行对齐

2.确定的对齐参数必须能够整除起始地址（或偏移量）

3.偏移地址和成员占用大小均需对齐

4.结构体成员的对齐参数为其所有成员使用的对齐参数的最大值

5.结构体总长度必须为所有对齐参数的整数倍

```
#include<stdio.h>
struct test
{
    char a;
    int   b;
    float c;
};

int main(void)
{
    printf("char=%d\n",sizeof(char));
    printf("int=%d\n",sizeof(int));
    printf("float=%d\n",sizeof(float));
    printf("struct test=%d\n",sizeof(struct test));
    return 0;
}
```

执行结果为 1，4，4，12

占用内存空间的计算过程：

对齐参数为 4。假设结构体的起始地址为 0x0

a 的类型为 char，因此所占内存空间大小为 1 个字节，小于对齐参数 4，所以选择 1 为对齐数，而地址 0x0 能够被 1 整除，所以 0x0 为 a 的起始地址，占用空间大小为 1 个字节；

b 的类型为 int ，所占内存空间大小为 4 个字节，与对齐参数相同，因此 4 为对齐数，0x1 不能被 4 整除，因此不能作为 b 的起始地址，依次往下推，只能选用 0x4 作为 b 的起始地址，因此中间会空出 3 个字节的空余空间；

c 的类型为 float，占用 4 个字节，因此 4 为对齐数，0x8 能被 4 整除，所以 c 的起始地址为 0x8

因此整个结构体占用的内存大小为 12 字节。

26. 根据要求编程

时间限制：1 秒

空间限制：32768K

有 n 个字符串，每个字符串都是由 A-J 的大写字符构成。现在你将每个字符映射为一个 0-9 的数字，不同字符映射为不同的数字。这样每个字符串就可以看做一个整数，唯一的要求是这些整数必须是正整数且它们的字符串不能有前导零。现在问你怎样映射字符才能使得这些字符串表示的整数之和最大？

输入描述:

每组测试用例仅包含一组数据，每组数据第一行为一个正整数 n ， 接下来有 n 行，每行一个长度不超过 12 且仅包含大写字母 A-J 的字符串。 n 不大于 50，且至少存在一个字符不是任何字符串的首字母。

输出描述:

输出一个数，表示最大和是多少。

输入例子 1:

2
ABC
BCA

输出例子 1:

1875

```
#include <iostream>
#include <String>
#include <algorithm>
#include <cmath>
using namespace std;
struct weight{
    long long num;
    bool head;
}A[10];
int cmp(const weight &a,const weight &b)
{
    return a.num<b.num;
}
int main() {
```

```
    int n;
    cin >> n;
    string str;
    long long MAX = 0;
    while(n--) {
        cin >> str;
        for(int i=str.length()-1; i>=0; i--) {
            A[str[i]-'A'].num += pow(10, str.length()-i-1);
            if(i==0)
                A[str[i]-'A'].head = true;
        }
    }
    sort(A, A+10,cmp);
    if(A[0].head){
        int i;
        for(i=1;i<10;i++){
            if(!A[i].head){
                swap(A[0],A[i]);
                break;
            }
        }
        sort(A+1, A+i+1,cmp);
    }
    for(int i=0; i<=9; i++) {
        MAX += (A[i].num * i);
    }
    cout << MAX;
}
```

27. 试题描述：

小 a 和小 b 玩一个游戏，有 n 张卡牌，每张上面有两个正整数 x,y。取一张牌时，个人积分增加 x，团队积分增加 y。

求小 a，小 b 各取若干张牌，使得他们的个人积分相等。

输入：

第一行一个整数 n。

接下来 n 行，每行两个整数 x,y，用空格隔开。

输出：

一行一个整数

表示小 a 的积分和小 b 的积分相等的时候，团队积分的最大值。

输入示例：

4

3 1
2 2
1 4
1 4
输出示例：
10

其他：
对于 100%的数据，0<n<=100，1<x<=1e3，0<y<=1e6。

```
#include<iostream>
#include<cstring>
#include<cmath>
#include<cstdio>
using namespace std;
int n,m,dp[300][200000],x[100000],y[100000];
int main()
{
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
    {
        scanf("%d%d",&x[i],&y[i]);
    }
    memset(dp,-1,sizeof(dp));
    dp[0][100000]=0;
    for(int i=1;i<=n;i++)
    {
        for(int j=20000;j<=200000;j++)
        {
            dp[i][j]=max(dp[i-1][j+x[i]],dp[i-1][j-x[i]]);
            if(dp[i][j]!=-1)dp[i][j]+=y[i];
            if(dp[i][j]<dp[i-1][j])dp[i][j]=dp[i-1][j];
        }
    }
     printf("%d",dp[n][100000]);
}
```

28. 试题描述：

一个球场 C 的球迷看台可容纳 M*N 个球迷。官方想统计一共有多少球迷群体，最大的球迷群体有多少人。球迷选座特性：同球迷群体会选择相邻座位，不同球迷群体选择不相邻的座位。（相邻包括前后相邻、左右相邻、斜对角相邻）；

给定一个 M*N 的二维球场，0 代表该位置没人，1 代表该位置有人，希望输出球队群体个数 P,最大的球队群体人数 Q。

输入：
第一行，2 个数字，M、N，使用英文逗号隔开。
接下来 M 行，每行 N 个数字，使用英文逗号隔开。

输出：
一行，2 数字，P 和 Q。
输入样例：
10,10
0,0,0,0,0,0,0,0,0,0
0,0,0,1,1,0,1,0,0,0
0,1,0,0,0,0,0,1,0,1
1,0,0,0,0,0,0,0,1,1
0,0,0,1,1,1,0,0,0,1
0,0,0,0,0,0,1,0,1,1
0,1,1,0,0,0,0,0,0,0
0,0,0,1,0,1,0,0,0,0
0,0,1,0,0,1,0,0,0,0
0,1,0,0,0,0,0,0,0,0
输出样例：
6,8

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<cmath>
#include<stdio.h>
using namespace std;
int n,m,l,k,sum,ans,cnt;
char a[4000][4000],op;
bool b[4000][4000]={0};
int dfs(int x,int y)
{
    if(a[x-1][y]=='1'&&b[x-1][y]==0)
    {
        b[x-1][y]=1;
        dfs(x-1,y);
        ans++;
    }
    if(a[x][y+1]=='1'&&b[x][y+1]==0)
    {
        b[x][y+1]=1;
        dfs(x,y+1);
        ans++;
    }
```

```
    if(a[x-1][y+1]=='1'&&b[x-1][y+1]==0)
    {
        b[x-1][y+1]=1;
        dfs(x-1,y+1);
        ans++;
    }
    if(a[x+1][y]=='1'&&b[x+1][y]==0)
    {
        b[x+1][y]=1;
        dfs(x+1,y);
        ans++;
    }
    if(a[x][y-1]=='1'&&b[x][y-1]==0)
    {
        b[x][y-1]=1;
        dfs(x,y-1);
        ans++;
    }
    if(a[x+1][y-1]=='1'&&b[x+1][y-1]==0)
    {
        b[x+1][y-1]=1;
        dfs(x+1,y-1);
        ans++;
    }
    if(a[x+1][y+1]=='1'&&b[x+1][y+1]==0)
    {
        b[x+1][y+1]=1;
        dfs(x+1,y+1);
        ans++;
    }
    if(a[x-1][y-1]=='1'&&b[x-1][y-1]==0)
    {
        b[x-1][y-1]=1;
        dfs(x-1,y-1);
        ans++;
    }
    return ans;
}
int main()
{
    scanf("%d%c%d",&n,&op,&m);
    for(int i=1;i<=n;i++)
    {
```

```
            for(int j=1;j<=m;j++)
            {
                getchar();
                a[i][j]=getchar();

            }
        }
        for(int i=1;i<=n;i++)
        {
            for(int j=1;j<=m;j++)
            {
                ans=0;
                if(a[i][j]=='0')b[i][j]=1;
                 if(a[i][j]=='1'&&b[i][j]==0)
                 {
                    sum++;
                    cnt=max(cnt,dfs(i,j));
                 }
            }
        }
        char p=',';
        printf("%d",sum);
        putchar(p);
        printf("%d",cnt);
    }
```

29. mysql 为什么要使用 B+树作为索引呢？

b 树的特点:

一个 M 阶的 b 树具有如下几个特征： (如下图 M=3)(下文的关键字可以理解为 有效数据,而不是单纯的索引)

定义任意非叶子结点最多只有 M 个儿子，且 M>2；

根结点的儿子数为[2, M]；

除根结点以外的非叶子结点的儿子数为[M/2, M]，向上取整； (儿子数:[2,3])

非叶子结点的关键字个数=儿子数-1；(关键字=2)

所有叶子结点位于同一层；

k 个关键字把节点拆成 k+1 段,分别指向 k+1 个儿子,同时满足查找树的大小关系。(k=2)

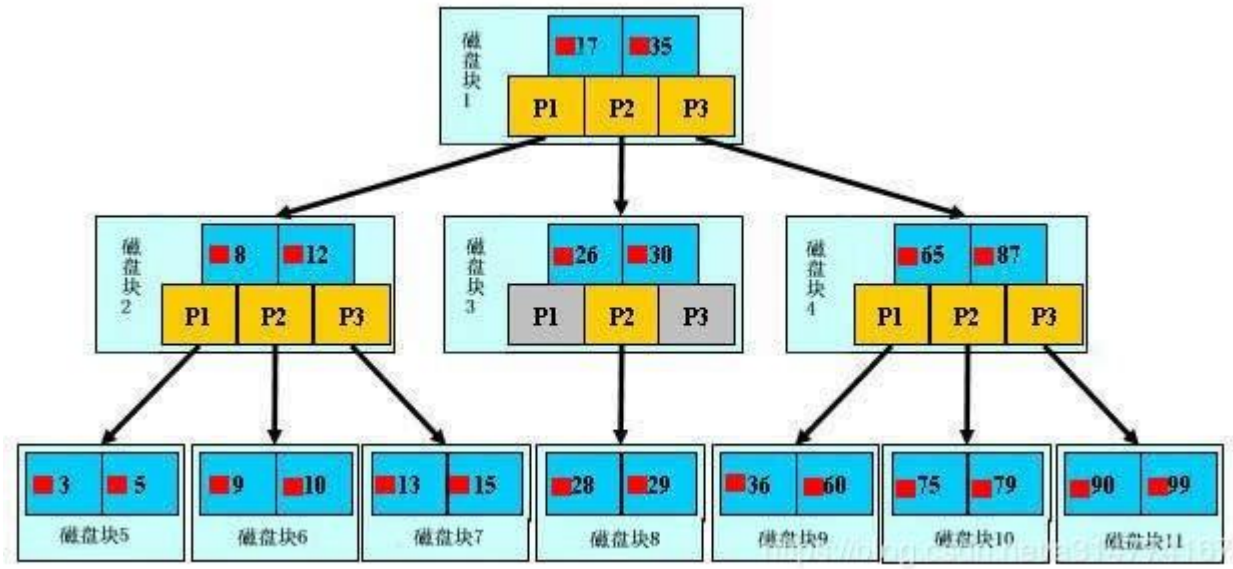
有关 b 树的一些特性，注意与后面的 b+树区分：

关键字集合分布在整颗树中；

任何一个关键字出现且只出现在一个结点中；

搜索有可能在非叶子结点结束；

其搜索性能等价于在关键字全集内做一次二分查找；



b+树，是 b 树的一种变体，查询性能更好。m 阶的 b+树的特征：

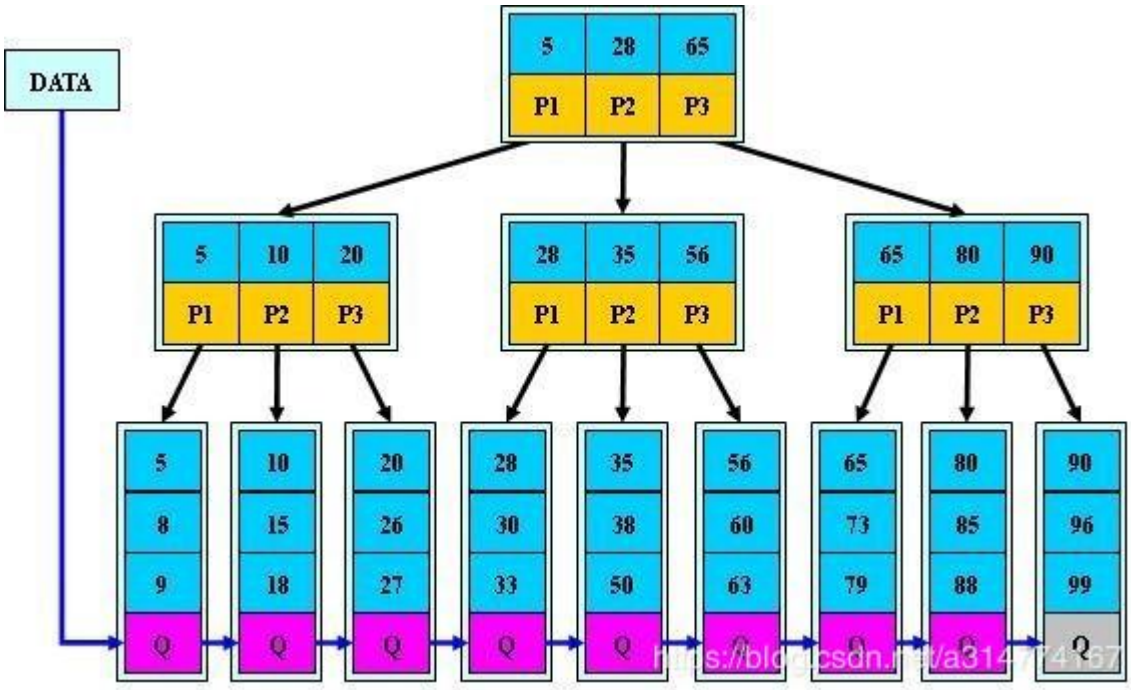
有 n 棵子树的非叶子结点中含有 n 个关键字（b 树是 n-1 个），这些关键字不保存数据，只用来索引，所有数据都保存在叶子节点（b 树是每个关键字都保存数据）。

所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。

所有的非叶子结点可以看成是索引部分，结点中仅含其子树中的最大（或最小）关键字。

通常在 b+树上有两个头指针，一个指向根结点，一个指向关键字最小的叶子结点。

同一个数字会在不同节点中重复出现，根节点的最大元素就是 b+树的最大元素。



————————————————————

选用 B+树作为数据库的索引结构的原因有

B+树的中间节点不保存数据，是纯索引,但是 B 树的中间节点是保存数据和索引的,相对来说,B+树磁盘页能容纳更多节点元素，更“矮胖”；

B+树查询必须查找到叶子节点，B 树只要匹配到即可不用管元素位置，因此 b+树查找更稳定（并不慢）；

对于范围查找来说，B+树只需遍历叶子节点链表即可，B 树却需要重复地中序遍历,在项目中范围查找又很是常见的

增删文件（节点）时，效率更高，因为 B+树的叶子节点包含所有关键字，并以有序的链表结构存储，这样可很好提高增删效率。

30. 输入 ping IP 后敲回车，发包前会发生什么？

ping 目标 ip 时，先查路由表，确定出接口

如果落在直连接口子网内，此时若为以太网等　多路访问网络　则先查询 arp 缓存，命中则直接发出,否则在该接口上发 arp 询问目标 ip 的 mac 地址,取得后发出,若为 ppp 等　点对点网络　，则直接可以发出；

如果查表落在缺省路由上，此时若为以太网等　多路访问网络　则先查询网关 arp 缓存，命中则直接发出，否则在该接口上发 arp 询问网关的 mac 地址，取得后发出，若为 ppp 等　点对点网络　，则直接可以发出；

若查表未命中，则返回不可达。

31. 对于一颗二叉树，如何对此进行层次遍历，并且按行输出。

解法 1:

```
class Solution {
public:
    vector<vector<int>> levelOrder(Node* root) {
        BFS(root, 0);
        return res;
    }
public:
    void BFS(Node* node, int level)
    {
        if(node == nullptr) return;
        if(level == res.size())
            res.push_back(vector<int> ());
        res[level].push_back(node->val);
        for(auto n:node->children)
            BFS(n, level+1);
    }
private:
```

```
    vector<vector<int>> res;
};
```

解法 2:

```
class Solution {
public:
    vector<vector<int>> levelOrder(Node* root) {
        vector<vector<int>> output;
        if (!root) return output;
        queue<Node*> treeTemp;
        treeTemp.push(root);
        auto treeNum = treeTemp.size();

        while (treeNum) {
            vector<int> valTemp;
            for (auto i=0; i<treeNum; ++i) {
                root = treeTemp.front();
                valTemp.push_back(root->val);
                treeTemp.pop();
                for (auto childTemp:root->children)
                    treeTemp.push(childTemp);
            }
            output.push_back(valTemp);
            treeNum = treeTemp.size();
        }
        return output;
    }
};
```

32. 如何实现一个高效的单向链表逆序输出？

```
    static class Node{
        private int data;
        private Node  next;
        Node(int d){
            data = d;
            next = null;
        }

        public int getData() {
            return data;
        }

        public void setData(int data) {
```

```
        this.data = data;
    }

    public Node getNext() {
        return next;
    }

    public void setNext(Node next) {
        this.next = next;
    }
}

public static void reverse(Node head)
{
    if(null == head || null == head.getNext()){
        return;
    }

    Node prev=null;
    Node pcur = head.getNext();
    Node next;

    while(pcur!=null){
        if(pcur.getNext()==null){
            pcur.setNext(prev);
            break;
        }
        next=pcur.getNext();
        pcur.setNext(prev);
        prev=pcur;
        pcur=next;
    }

    head.setNext(pcur);
    Node tmp = head.getNext();
    while(tmp!= null){
        System.out.print(tmp.getData()+" ");
        tmp=tmp.getNext();
    }
}
```

33. 从 innodb 的索引结构分析，为什么索引的 key 长度不能太长？

key 太长会导致一个页当中能够存放的 key 的数目变少，间接导致索引树的页数目变多，索引层次增加，从而影响整体查询变更的效率。

34. MySQL 的数据如何恢复到任意时间点？

恢复到任意时间点以定时的做全量备份，以及备份增量的 binlog 日志为前提。恢复到任意时间点首先将全量备份恢复之后，再此基础上回放增加的 binlog 直至指定的时间点。

35. 请解释下为什么鹿晗发布恋情的时候，微博系统会崩溃，如何解决？

鹿晗首先是一个明星，流量明星。粉丝量众多，所以，他已公布恋情，瞬间的流量很大。但是我们要注意到，这里面有一个问题。就是这个瞬间流量增大，增的不仅是浏览量。如果仅仅是阅读，我们只需把鹿晗的这条微博放入 Redis 缓存，以微博技术，不可能挂得了的吧。

这个之所以微博挂掉，是因为这个时间段，转发 + 评论量非常的大，并不是只有阅读量大。

另外针对明星的微博，会有一个消息推送功能。第一时间热点数据，只要你联的有网，都能够收到推送。

最后总结如下:

1. 获取微博通过 pull 方式还是 push 方式
2. 发布微博的频率要远小于阅读微博
3. 流量明星的发微博，和普通博主要区分对待，比如在 sharding 的时候，也要考虑这个因素

36. 设计和实现一个 LRU（最近最少使用）缓存数据结构，使它应该支持一下操作：get 和 put。

get(key) - 如果 key 存在于缓存中，则获取 key 的 value（总是正数），否则返回 -1。

```
class LRUCache{
    public:
    LRUCache(int capacity) {
```

```
        cap = capacity;
    }
    int get(int key) {
        auto it = m.find(key);
        if (it == m.end()) return -1;
        l.splice(l.begin(), l, it->second);
        return it->second->second;
    }
    void set(int key, int value) {
        auto it = m.find(key);
        if (it != m.end())
            l.erase(it->second);
        l.push_front(make_pair(key, value));
        m[key] = l.begin();
        if (m.size() > cap) {
            int k = l.rbegin()->first;
            l.pop_back();
            m.erase(k);
        }
    }
}
```

37. 已知 sqrt(2)约等于 1.414，要求不用数学库，求 sqrt(2)精确到小数点后 10 位。

1. 已知 sqrt(2)约等于 1.414，那么就可以在(1.4, 1.5)区间做二分查找，如：
high=>1.5
low=>1.4
mid => (high+low)/2=1.45
1.45*1.45>2 ? high=>1.45 : low => 1.45
循环到 c)
2. 退出条件
前后两次的差值的绝对值<=0.0000000001, 则可退出。

代码：

```
const double EPSINON = 0.0000000001;
double sqrt2( )
{
    double low = 1.4, high = 1.5;
    double mid = (low + high) / 2;
    while (high - low > EPSINON)
    {
        if (mid*mid < 2)
```

```
            {
                high = mid;
            }
            else
            {
            low = mid;
            }
            mid = (high + low) / 2;
        }
        return mid;
    }
```

38. 给定一个二叉搜索树(BST)，找到树中第 K 小的节点。

```
    /**
    * Definition for a binary tree node.
    * public class TreeNode {
    * int val;
    * TreeNode left;
    * TreeNode right;
    * TreeNode(int x) { val = x; }
    * }
    */
    class Solution {

        private class ResultType {
            // 是否找到
            boolean found;
            // 节点数目
            int val;
            ResultType(boolean found, int val) {
                this.found = found;
                this.val = val;
            }
        }
        public int kthSmallest(TreeNode root, int k) {
            return kthSmallestHelper(root, k).val;
        }
        private ResultType kthSmallestHelper(TreeNode root, int k) {
            if (root == null) {
                return new ResultType(false, 0);
            }
            ResultType left = kthSmallestHelper(root.left, k);
            // 左子树找到，直接返回
```

```
            if (left.found) {
                return new ResultType(true, left.val);
            }
            // 左子树的节点数目 = K-1，结果为 root 的值
            if (k - left.val == 1) {
                return new ResultType(true, root.val);
            }
            // 右子树寻找
            ResultType right = kthSmallestHelper(root.right, k - left.val - 1);
            if (right.found) {
                return new ResultType(true, right.val);
            }
            // 没找到，返回节点总数
            return new ResultType(false, left.val + 1 + right.val);
        }
    }
```

39. 如何用 socket 编程实现 ftp 协议？

以 linux 为例，使用 socket 编程中的 read()函数和 write()函数已可实现文件的发送接收，为啥还要专门建立 ftp 协议呢？单单使用 read()函数和 write()函数，数据接口和命令接口未分开，效率低。而 ftp 将数据接口和命令接口分开，提高了文件传输效率和安全性。

ftp 协议的实现仍是使用 socket 编程，首先是实现 tcp 连接。

Socket 客户端编程主要步骤如下：

socket() 创建一个 Socket

connect() 与服务器连接

write() 和 read() 进行会话

close() 关闭 Socket

Socket 服务器端编程主要步骤如下：

socket() 创建一个 Socket

bind()

listen() 监听

accept() 接收连接的请求

write() 和 read() 进行会话

close() 关闭 Socket

建立 tcp 连接代码简示如下:

```
SOCKET control_sock;
struct hostent *hp;
struct sockaddr_in server;
memset(&server, 0, sizeof(struct sockaddr_in))
/* 初始化 socket */
control_sock = socket(AF_INET, SOCK_STREAM, 0);
```

```
hp = gethostbyname(server_name);
memcpy(&server.sin_addr, hp->h_addr, hp->h_length);
server.sin_family = AF_INET;
server.sin_port = htons(port);
/* 连接到服务器端 */
connect(control_sock,(struct sockaddr *)&server, sizeof(server));
/* 客户端接收服务器端的一些欢迎信息 */
read(control_sock, read_buf, read_len);
```

ftp 客户端与服务器建立起 tcp 连接后，然后向服务器发送命令。（这一步建立的是命令接口的 tcp 连接，数据接口连接尚未建立。）

通常第一步发送 USER 和 PASS 命令验证账号和密码后登陆服务器。若是匿名服务器则另当别论。

登陆服务器代码简示如下:

```
/* 命令 ”USER username\r\n” */
sprintf(send_buf,"USER %s\r\n",username);
/*客户端发送用户名到服务器端 */
write(control_sock, send_buf, strlen(send_buf));
/* 客户端接收服务器的响应码和信息，正常为
 ”331 User name okay, need password.”
*/
read(control_sock, read_buf, read_len);
/* 命令 ”PASS password\r\n” */
sprintf(send_buf,"PASS %s\r\n",password);
/* 客户端发送密码到服务器端 */
write(control_sock, send_buf, strlen(send_buf));
/* 客户端接收服务器的响应码和信息，正常为 ”230 User logged in, proceed.” */
read(control_sock, read_buf, read_len);
```

接下来是选择发送 PORT 命令选择主动模式还是发送 PASV 命令选择被动模式。

主动模式还是被动模式是相对服务器来说的。被动模式即命令接口连接和数据接口连接都由客户端主动建立；主动模式是数据接口连接由服务器主动建立。这里仅简述被动模式的建立，主动模式可依理建立。

被动模式建立连接代码:

```
/* 命令 ”PASV\r\n” */
sprintf(send_buf,"PASV\r\n");
/* 客户端告诉服务器用被动模式 */
write(control_sock, send_buf, strlen(send_buf));
/*客户端接收服务器的响应码和新开的端口号，* 正常为
”227 Entering passive mode (<h1,h2,h3,h4,p1,p2>)”
*/
read(control_sock, read_buf, read_len);
```

客户端发送 PASV 命令让服务器进入被动模式。服务器会打开数据端口并监听。并返回

响应码 227 和数据连接的端口号。

接下来通过数据端口下载上传查看文件就不赘述了。

```
/* 连接服务器新开的数据端口 */
connect(data_sock,(struct sockaddr *)&server, sizeof(server));
/* 命令 ”CWD dirname\r\n” */
sprintf(send_buf,"CWD %s\r\n", dirname);
/* 客户端发送命令改变工作目录 */
write(control_sock, send_buf, strlen(send_buf));
/* 客户端接收服务器的响应码和信息，正常为 ”250 Command okay.” */
read(control_sock, read_buf, read_len);
/* 命令 ”SIZE filename\r\n” */
sprintf(send_buf,"SIZE %s\r\n",filename);
/* 客户端发送命令从服务器端得到下载文件的大小 */
write(control_sock, send_buf, strlen(send_buf));
/* 客户端接收服务器的响应码和信息，正常为 ”213 <size>” */
read(control_sock, read_buf, read_len);
/* 命令 ”RETR filename\r\n” */
sprintf(send_buf,"RETR %s\r\n",filename);
/* 客户端发送命令从服务器端下载文件 */
write(control_sock, send_buf, strlen(send_buf));
/* 客户端接收服务器的响应码和信息，正常为 ”150 Opening data connection.” */
read(control_sock, read_buf, read_len);
/* 客户端创建文件 */
file_handle = open(disk_name, CRFLAGS, RWXALL);
for( ; ; ) {
    ... ...
    /* 客户端通过数据连接 从服务器接收文件内容 */
    read(data_sock, read_buf, read_len);
    /* 客户端写文件 */
    write(file_handle, read_buf, read_len);
    ... ...
}
/* 客户端关闭文件 */
rc = close(file_handle);
```

下载完成后客户端退出服务器，关闭连接。

```
/* 客户端关闭数据连接 */
close(data_sock);
/* 客户端接收服务器的响应码和信息，正常为 ”226 Transfer complete.” */
read(control_sock, read_buf, read_len);
/* 命令 ”QUIT\r\n” */
sprintf(send_buf,"QUIT\r\n");
/* 客户端将断开与服务器端的连接 */
write(control_sock, send_buf, strlen(send_buf));
```

```
/* 客户端接收服务器的响应码，正常为 ”200 Closes connection.” */
read(control_sock, read_buf, read_len);
/* 客户端关闭控制连接 */
close(control_sock);
```

断点续传的实现

由于网络不稳定，在传输文件的过程中，可能会发生连接断开的情况，这时候需要客户端支持断点续传的功能，下次能够从上次终止的地方开始接着传送。需要使用命令 REST。如果在断开连接前，一个文件已经传输了 512 个字节。则断点续传开始的位置为 512，服务器会跳过传输文件的前 512 字节。

40. 给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

```
package test;
import java.util.Scanner;

public class Many {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        String str = "";
        str = sc.nextLine(); // 读入数组中所有元素;

        String arr[] = str.split(" "); // 用空格分割字符串
        int n = arr.length;
        int arra[] = new int[n];
        for (int i = 0; i < arra.length; i++) {
            arra[i] = Integer.parseInt(arr[i]);
            // System.out.print(arra[i] + " ");
        } // 将字符串转化为整数数组

        int max = 0;    // 盈利最大
        for (int i = 0; i < arra.length - 1; i++) {
            int t = arra[i+1] - arra[i];
            if(t > 0) max += t;
        }
        System.out.println(max);
```

```
        }

    }
```

41. 描述实时系统的基本特性

实时系统是指在系统工作时，能在特定的时间内完成特定的任务，其各种资源可以根据需要进行动态的分配，因此其处理事务的能力强，速度快。

1）高精度计时系统

计时精度是影响实时性的一个重要因素。在实时应用系统中，经常需要精确确定实时地操作某个设备或执行某个任务，或精确的计算一个时间函数。这些不仅依赖于一些硬件提供的时钟精度，也依赖于实时操作系统实现的高精度计时功能。

2）多级中断机制

一个实时应用系统通常需要处理多种外部信息或事件，但处理的紧迫程度有轻重缓急之分。有的必须立即作出反应，有的则可以延后处理。因此，需要建立多级中断嵌套处理机制，以确保对紧迫程度较高的实时事件进行及时响应和处理。

3）实时调度机制

实时操作系统不仅要及时响应实时事件中断，同时也要及时调度运行实时任务。但是，处理机调度并不能随心所欲的进行，因为涉及到两个进程之间的切换，只能在确保“安全切换”的时间点上进行，实时调度机制包括两个方面，一是在调度策略和算法上保证优先调度实时任务；二是建立更多“安全切换”时间点，保证及时调度实时任务。

42. 求任意一颗二叉树最长路径长度

样例:如下所示

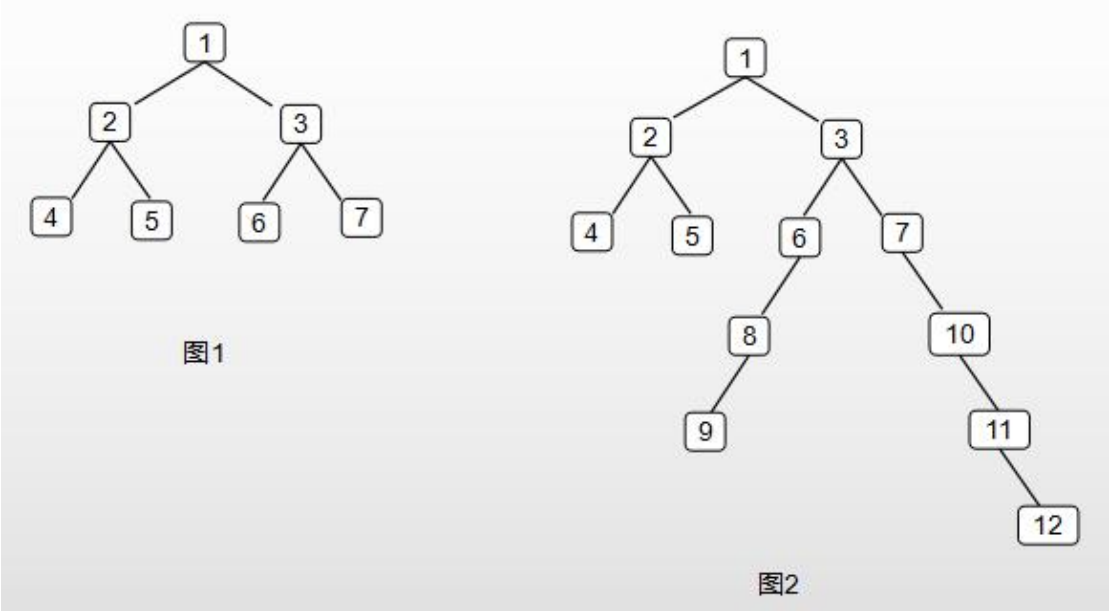


图 1 树的最长路径长度为 4，图 2 的最长路径长度为 7，图 1 最长路径经过根节点，顶点为 1，图 2 不经过，顶点为 3

思路:

树中任意两个节点之间，连接起来的路径最长。方法就是求出每个节点的左子树和右子树的高度，两者相加就是当前节点的最长路径，然后比较每个节点的最长路径，最大的就是结果

实现方法:

定义一个静态变量 MaxLength 记录每一步最大长度，采取前序遍历来遍历每一个节点，在遍历过程中，对当前节点的最长路径进行比较，对于每一个节点最长路径求法，先求出它左子树和右子树的高度(节点数最多的路径),然后相加即为当前节点最长路径

```
static  Integer MaxLength=0;//记录最长路径
//遍历整棵树，得到最长路径
public void getLength(TreeNode t){
    if(t!=null){
        MaxLength=Math.max(LengthTree(t),MaxLength);
        getLength(t.lchild);
        getLength(t.rchild);
    }
}
//得到当前节点的最长路径
public int LengthTree(TreeNode t){
    if (t==null)
        return 0;
    int left=heighTree(t.lchild);
    int right=heighTree(t.rchild);
    int CurMax=left+right;
    return CurMax;
}
//求二叉树最大高度
public int heighTree(TreeNode t){
    if (t==null)
        return 0;
    else
        return Math.max(heighTree(t.lchild),heighTree(t.rchild))+1;
}
```

43. 手写冒泡排序算法,计算冒泡排序算法的时间复杂度?

```
public void swap(int[] array, int i, int j) {
    array[i] = array[i] + array[j];
    array[j] = array[i] - array[j];
    array[i] = array[i] - array[j];
}
public void sort(int[] array) {
    int n = array.length;
    for (int i = 0; i < n - 1; i++) {
```

```
            //在这里 j 不需要遍历到 n-1 了，因为 n-1-i~n-1 之间的元素
            //已经排好序了，不需要再比较
            for (int j = 0; j < n - 1 - i; j++) {
                //将最大元素移动到数组末尾
                if (array[j] > array[j + 1]) {
                    swap(array, j, j + 1);
                }
            }
        }
    }
```

从代码中可以看出一共遍历了 n-1 + n-2 + … + 2 + 1 = n * (n-1) / 2 = 0.5 * n ^ 2 - 0.5 * n，

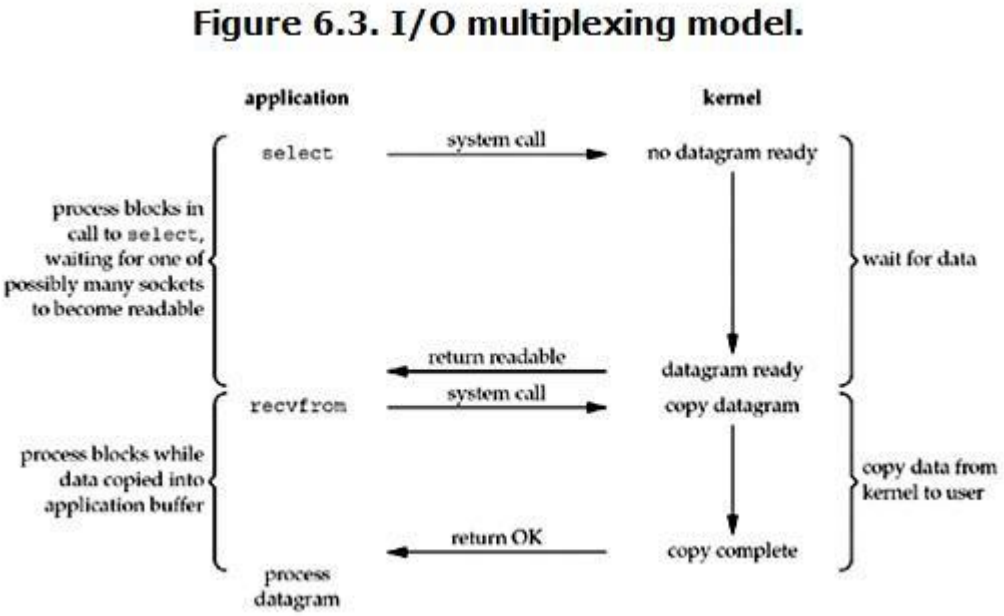
那么时间复杂度是 O(N^2)。

44. redis 中的网络 IO 有了解过吗，它是单线程的还是多线程的，为什么要用单线程。

redis 采用网络 IO 多路复用技术来保证在多连接的时候，系统的高吞吐量。

多路-指的是多个 socket 连接，复用-指的是复用一个线程。多路复用主要有三种技术：select，poll，epoll。epoll 是最新的也是目前最好的多路复用技术。

这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络 IO 的时间消耗），且 Redis 在内存中操作数据的速度非常快（内存内的操作不会成为这里的性能瓶颈），主要以上两点造就了 Redis 具有很高的吞吐量。



因为 Redis 是基于内存的操作，CPU 不是 Redis 的瓶颈，Redis 的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且 CPU 不会成为瓶颈，所以采用单线程

的方案。

45. Learning to Rank 了解吗，三种模式说一下（pair wise、point wise、 list wise）

在机器学习的 ranking 技术——learning2rank，包括 pointwise、pairwise、listwise 三大类型。

1、Pointwise Approach

1.1 特点

Pointwise 类方法，其 L2R 框架具有以下特征：

输入空间中样本是单个 doc（和对应 query）构成的特征向量；

输出空间中样本是单个 doc（和对应 query）的相关度；

假设空间中样本是打分函数；

损失函数评估单个 doc 的预测得分和真实得分之间差异。

这里讨论下，关于人工标注标签怎么转换到 pointwise 类方法的输出空间：

如果标注直接是相关度 s_j，则 doc x_j 的真实标签定义为 y_j=s_j

如果标注是 pairwise preference s_{u,v}，则 doc x_j 的真实标签可以利用该 doc 击败了其他 docs 的频次

如果标注是整体排序 π，则 doc x_j 的真实标签可以利用映射函数，如将 doc 的排序位置序号当作真实标签

1.2 根据使用的 ML 方法不同，pointwise 类可以进一步分成三类：基于回归的算法、基于分类的算法，基于有序回归的算法。

（1）基于回归的算法

此时，输出空间包含的是实值相关度得分。采用 ML 中传统的回归方法即可。

（2）基于分类的算法

此时，输出空间包含的是无序类别。对于二分类，SVM、LR 等均可；对于多分类，提升树等均可。

（3）基于有序回归的算法

此时，输出空间包含的是有序类别。通常是找到一个打分函数，然后用一系列阈值对得分进行分割，得到有序类别。采用 PRanking、基于 margin 的方法都可以。

1.3 缺陷

回顾概述中提到的评估指标应该基于 query 和 position，

ranking 追求的是排序结果，并不要求精确打分，只要有相对打分即可。pointwise 类方法并没有考虑同一个 query 对应的 docs 间的内部依赖性。一方面，导致输入空间内的样本不是 IID 的，违反了 ML 的基本假设，另一方面，没有充分利用这种样本间的结构性。其次，当不同 query 对应不同数量的 docs 时，整体 loss 将会被对应 docs 数量大的 query 组所支配，前面说过应该每组 query 都是等价的。

损失函数也没有 model 到预测排序中的位置信息。因此，损失函数可能无意的过多强调那些不重要的 docs，即那些排序在后面对用户体验影响小的 doc。

1.4 改进

如在 loss 中引入基于 query 的正则化因子的 RankCosine 方法。

2、Pairwise Approach

2.1 特点

Pairwise 类方法，其 L2R 框架具有以下特征：

输入空间中样本是（同一 query 对应的）两个 doc（和对应 query）构成的两个特征向量；

输出空间中样本是 pairwise preference；

假设空间中样本是二变量函数；

损失函数评估 doc pair 的预测 preference 和真实 preference 之间差异。

2.2 基于二分类的算法

Pairwise 类方法基本就是使用二分类算法即可。

经典的算法有 基于 NN 的 SortNet，基于 NN 的 RankNet，基于 fidelity loss 的 FRank，基于 AdaBoost 的 RankBoost，基于 SVM 的 RankingSVM，基于提升树的 GBRank。

2.3 缺陷

虽然 pairwise 类相较 pointwise 类 model 到一些 doc pair 间的相对顺序信息，但还是存在不少问题，回顾概述中提到的评估指标应该基于 query 和 position，

如果人工标注给定的是第一种和第三种，即已包含多有序类别，那么转化成 pairwise preference 时必定会损失掉一些更细粒度的相关度标注信息。

doc pair 的数量将是 doc 数量的二次，从而 pointwise 类方法就存在的 query 间 doc 数量的不平衡性将在 pairwise 类方法中进一步放大。

pairwise 类方法相对 pointwise 类方法对噪声标注更敏感,即一个错误标注会引起多个 doc pair 标注错误。

pairwise 类方法仅考虑了 doc pair 的相对位置，损失函数还是没有 model 到预测排序中的位置信息。

pairwise 类方法也没有考虑同一个 query 对应的 doc pair 间的内部依赖性，即输入空间内的样本并不是 IID 的，违反了 ML 的基本假设，并且也没有充分利用这种样本间的结构性。

2.4 改进

pairwise 类方法也有一些尝试，去一定程度解决上述缺陷，比如：

Multiple hyperplane ranker，主要针对前述第一个缺陷

magnitude-preserving ranking，主要针对前述第一个缺陷

IRSVM，主要针对前述第二个缺陷

采用 Sigmoid 进行改进的 pairwise 方法，主要针对前述第三个缺陷

P-norm push，主要针对前述第四个缺陷

Ordered weighted average ranking，主要针对前述第四个缺陷

LambdaRank，主要针对前述第四个缺陷

Sparse ranker，主要针对前述第四个缺陷

3、Listwise Approach

3.1 特点

Listwise 类方法，其 L2R 框架具有以下特征：

输入空间中样本是（同一 query 对应的）所有 doc（与对应的 query）构成的多个特征向量（列表）；

输出空间中样本是这些 doc（和对应 query）的相关度排序列表或者排列；
假设空间中样本是多变量函数，对于 docs 得到其排列，实践中，通常是一个打分函数，根据打分函数对所有 docs 的打分进行排序得到 docs 相关度的排列；
损失函数分成两类，一类是直接和评价指标相关的，还有一类不是直接相关的。具体后面介绍。

3.2 根据损失函数构造方式的不同，listwise 类可以分成两类直接基于评价指标的算法，间接基于评价指标的算法。

（1）直接基于评价指标的算法

直接取优化 ranking 的评价指标，也算是 listwise 中最直观的方法。但这并不简单，因为前面说过评价指标都是离散不可微的，具体处理方式有这么几种：
优化基于评价指标的 ranking error 的连续可微的近似，这种方法就可以直接应用已有的优化方法，如 SoftRank，ApproximateRank，SmoothRank
优化基于评价指标的 ranking error 的连续可微的上界，如 SVM-MAP，SVM-NDCG，PermuRank
使用可以优化非平滑目标函数的优化技术，如 AdaRank，RankGP
上述方法的优化目标都是直接和 ranking 的评价指标有关。现在来考虑一个概念，informativeness。通常认为一个更有信息量的指标，可以产生更有效的排序模型。而多层评价指标（NDCG）相较二元评价（AP）指标通常更富信息量。因此，有时虽然使用信息量更少的指标来评估模型，但仍然可以使用更富信息量的指标来作为 loss 进行模型训练。

（2）非直接基于评价指标的算法

这里，不再使用和评价指标相关的 loss 来优化模型，而是设计能衡量模型输出与真实排列之间差异的 loss，如此获得的模型在评价指标上也能获得不错的性能。经典的如 ，ListNet，ListMLE，StructRank，BoltzRank。

3.3 缺陷

listwise 类相较 pointwise、pairwise 对 ranking 的 model 更自然，解决了 ranking 应该基于 query 和 position 问题。listwise 类存在的主要缺陷是：一些 ranking 算法需要基于排列来计算 loss，从而使得训练复杂度较高，如 ListNet 和 BoltzRank。此外，位置信息并没有在 loss 中得到充分利用，可以考虑在 ListNet 和 ListMLE 的 loss 中引入位置折扣因子。

3.4 改进

pairwise 类方法也有一些尝试，去一定程度解决上述缺陷，比如：
Multiple hyperplane ranker，主要针对前述第一个缺陷
magnitude-preserving ranking，主要针对前述第一个缺陷
IRSVM，主要针对前述第二个缺陷
采用 Sigmoid 进行改进的 pairwise 方法，主要针对前述第三个缺陷
P-norm push，主要针对前述第四个缺陷
Ordered weighted average ranking，主要针对前述第四个缺陷
LambdaRank，主要针对前述第四个缺陷
Sparse ranker，主要针对前述第四个缺陷

以上，这三大类方法主要区别在于损失函数。不同的损失函数决定了不同的模型学习过程和输入输出空间。

46. 用户输入 M,N 值，从 1 至 N 开始顺序循环数数，每数到 M 输出该数值，直至全部输出。写出 C 程序。

```
#include <stdio.h>
#include <stdlib.h>

typedef struct node{
    int data;
    node* next;
}node;

void CreatList(node*& head, node*& tail, int n){
    if(n<1){
        head = tail = NULL;
        return;
    }
    head = new node();
    head->data = 1;
    head->next = NULL;
    node* p = head;
    for(int i=2; i<n+1; i++){
        p->next =new node();
        p = p->next;
        p->data = i;
        p->next = NULL;
    }
    tail = p;
    tail -> next = head;
}

void Print(node*& head){
    node* p = head;
    while(p && p->next != head){
        printf("%d", p->data);
        p = p->next;
    }
    if(p){
        printf("%d\n", p->data);
    }
}

void CountPrint(node*& head, node*& tail, int m){
```

```
    node* pre = tail;
    node* cur = head;
    int cnt = m;
    while(cur && cur->next != cur){
        if(cnt != 1){
            cnt--;
            pre = cur;
            cur = cur->next;
        }else{
            printf("%d", cur->data);
            pre->next =cur->next;
            delete cur;
            cur = pre->next;
            cnt = m;
        }
    }
    if(cur){
        printf("%d", cur->data);
        delete cur;
        head = tail =NULL;
    }
}

int main(){
    node* head;
    node* tail;
    int m;
    int n;
    scanf_s("%d", &n);
    scanf_s("%d", &m);
    CreatList(head, tail, n);
    Print(head);
    CountPrint(head,tail,m);
    system("pause");
    return 0;
}
```

47. zset 的底层是用什么数据结构实现的。

zset 底层的存储结构包括 ziplist 或 skiplist，在同时满足以下两个条件的时候使用 ziplist，其他时候使用 skiplist，两个条件如下：

有序集合保存的元素数量小于 128 个

有序集合保存的所有元素的长度小于 64 字节

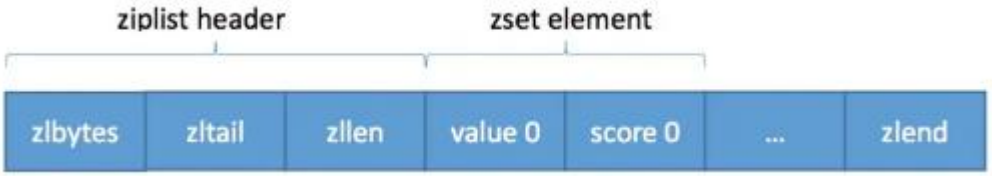
当 ziplist 作为 zset 的底层存储结构时候，每个集合元素使用两个紧挨在一起的压缩

列表节点来保存，第一个节点保存元素的成员，第二个元素保存元素的分值。

当 skiplist 作为 zset 的底层存储结构的时候，使用 skiplist 按序保存元素及分值，使用 dict 来保存元素和分值的映射关系。

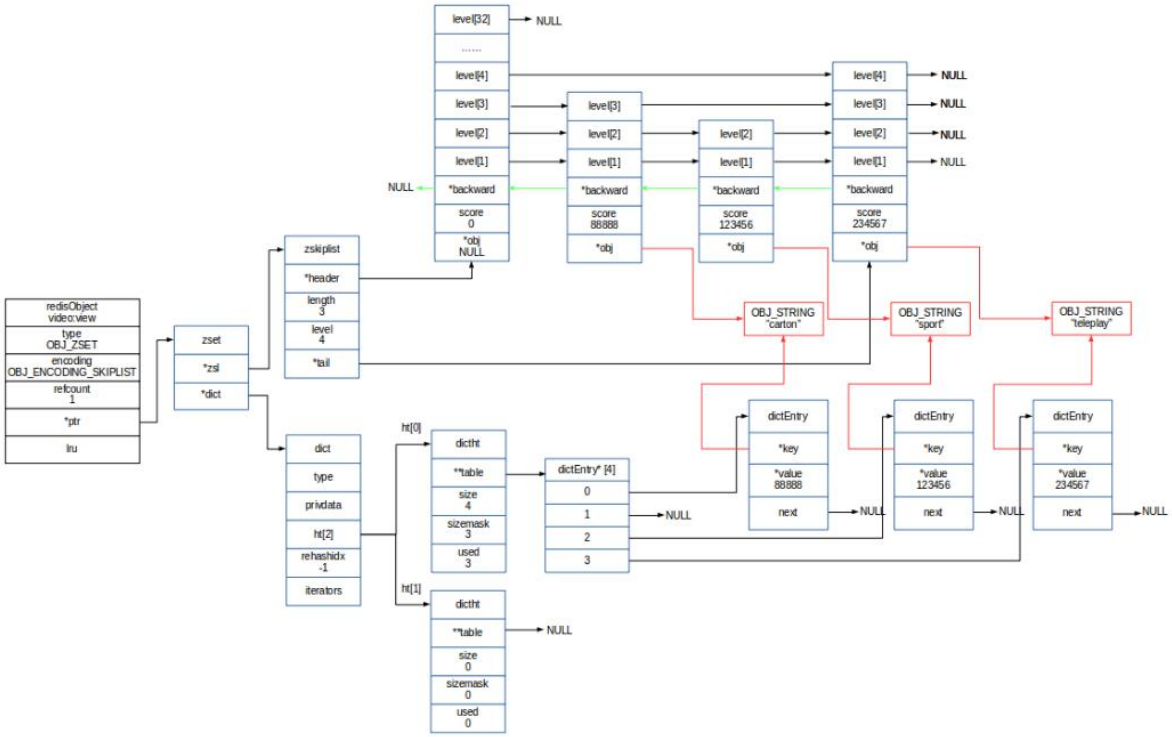
ziplist 数据结构

ziplist 作为 zset 的存储结构时，格式如下图，细节就不多说了，我估计大家都看得懂，紧挨着的是元素 memeber 和分值 socore，整体数据是有序格式。



skiplist 数据结构

skiplist 作为 zset 的存储结构，整体存储结构如下图，核心点主要是包括一个 dict 对象和一个 skiplist 对象。dict 保存 key/value，key 为元素，value 为分值；skiplist 保存的有序的元素列表，每个元素包括元素和分值。两种数据结构下的元素指向相同的位置。



zset 存储过程

zset 的添加过程我们以 zadd 的操作作为例子进行分析，整个过程如下：

解析参数得到每个元素及其对应的分值

查找 key 对应的 zset 是否存在不存在则创建

如果存储格式是 ziplist，那么在执行添加的过程中我们需要区分元素存在和不存在两种情况，存在情况下先删除后添加；不存在情况下则添加并且需要考虑元素的长度是否超出限制或实际已有的元素个数是否超过最大限制进而决定是否转为 skiplist 对象。

如果存储格式是 skiplist，那么在执行添加的过程中我们需要区分元素存在和不存在两种情况，存在的情况下先删除后添加，不存在情况下那么就直接添加，在 skiplist 当中添加完

以后我们同时需要更新 dict 的对象。

48. 如果你要对班里的学生根据分数进行排名，你觉得用 redis 里的哪个数据结构比较好。

ZSET

49. 在第一象限内，有一些(n 个）离散的点（x,y 均为自然数，且某一行、列都只有一个点）

代码需要输出一些“最大点”（设待考察点为 x0,y0，如果存在 x,y，满足 x>x0 且 y>y0,则 x0,y0 不是最大点），输出顺序为 x 值递增顺序。

思路：

(1)对于所有输入的点，按照 x 值递增排序。——因为 x 值没有重复，且已知 x 的范围，所以使用位图法排序，复杂度 O(n)。直接 sort 就当做是 nlog(n)咯！

(2)从最右侧的点（xn，yn）开始研究，他的右边没有点了，题干的条件被破坏，那么（xn，yn）是最大点；同时用 max 来表示当前研究点右侧（包含当前）点的最大 y 值。

(3)研究（x n-1,y n-1）,因为 x n-1 的右侧已经有点 x n，那么（x n-1,y n-1）是不是最大点的关键就在于，y n-1<max？如果小于，那 x n-1 不是最大点，否则他是最大点。更新 max 值

(4)循环第三步，直到待考察的点为空。——复杂度 O（n）

50. 给定一个序列，如【6,2,1】，要求输出：在某一个区间上的最小值 min 与这个区间上所有值的和 sum 的乘积，最终只输出最大的乘积值。

思路；肯定是用两个 index 来模拟区间范围，然后在区间里寻找 min 和 sum 了。

那么可以这么做：

```
int max=-1;
for(int i=0;i<n;i++){
    int sum = 0;
    int min = array[i];
    for(int j=i;j<n;++){
```

```
            if(array[j]<min)min=array[j];//更新 min
            sum+=array[j];//更新 sum
            if (min*sum>max) max=min*sum;
        }
    }
```