# 数据结构

数据结构是以某种特定的布局方式存储数据的容器

## 数组

### 数组类型

* 一维数组
* 多维数组

### 数组的基本操作

Insert——在指定索引位置插入一个元素

Get——返回指定索引位置的元素

Delete——删除指定索引位置的元素

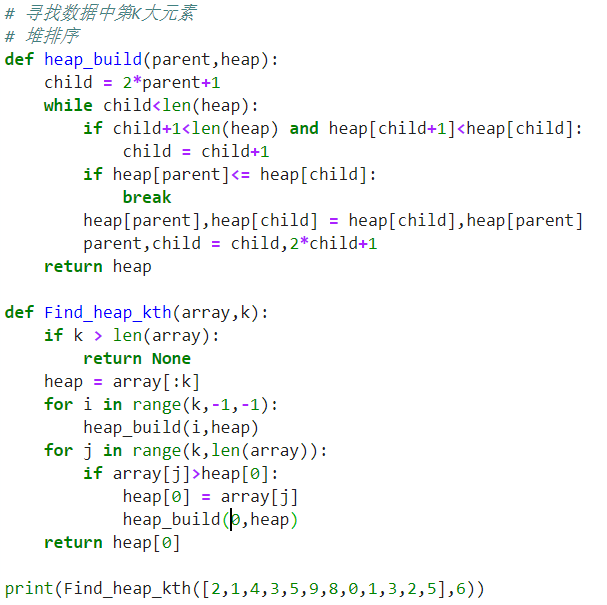
Size——得到数组所有元素的数量栈

### 面试中常见问题

#### 寻找无序数组中第k大的元素

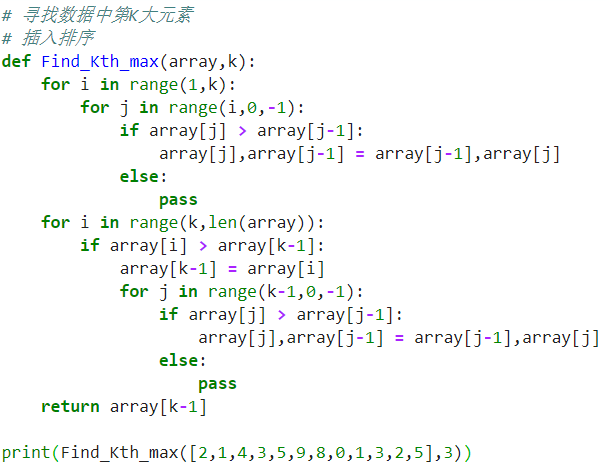
解法1：

**堆排序**：采用元素下沉法，维护一个k大小的最小堆，对于数组中的每一个元素判断与堆顶的大小，若堆顶较大，则不管，否则，弹出堆顶，将当前值插入到堆中，继续调整最小堆。时间复杂度O(n \* logk)



解法2：

插入排序：由于是要找 k 个最大的数，所以没有必要对所有数进行完整的排序。每次只保留 k 个当前最大的数就可以，然后每次对新来的元素跟当前 k 个树中最小的数比较，新元素大的话则插入到数组中，否则跳过。循环结束后数组中最小的数即是我们要找到第 k 大的数。 时间复杂度 **(n-k)logk**



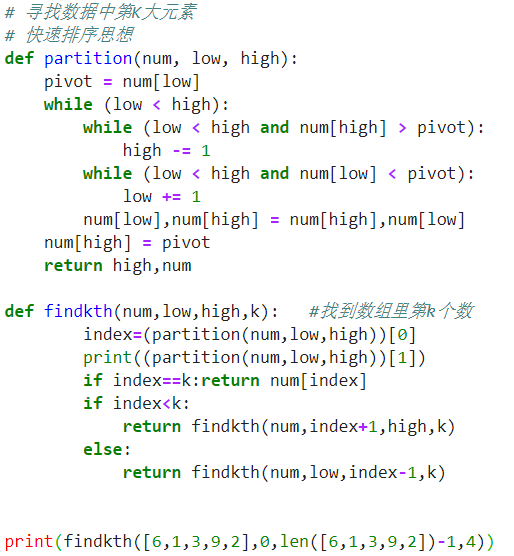
解法3：

快速排序的思想：从数组S中随机找出一个元素X，把数组分为两部分Sa和Sb。Sa中的元素大于等于X，Sb中元素小于X。

这时有两种情况：

1. Sa中元素的个数小于k，则Sb中的第k-|Sa|个元素即为第k大数；

2. Sa中元素的个数大于等于k，则返回Sa中的第k大数。时间复杂度近似为O(n)



* 找到数组中第一个不重复出现的整数
* 合并两个有序数组
* 重新排列数组中的正值和负值

## 队列（先进先出）

### 基本操作

Enqueue() —— 在队列尾部插入元素

Dequeue() ——移除队列头部的元素

isEmpty()——如果队列为空，则返回true

Top() ——返回队列的第一个元素

### 常见问题

* 使用队列表示栈

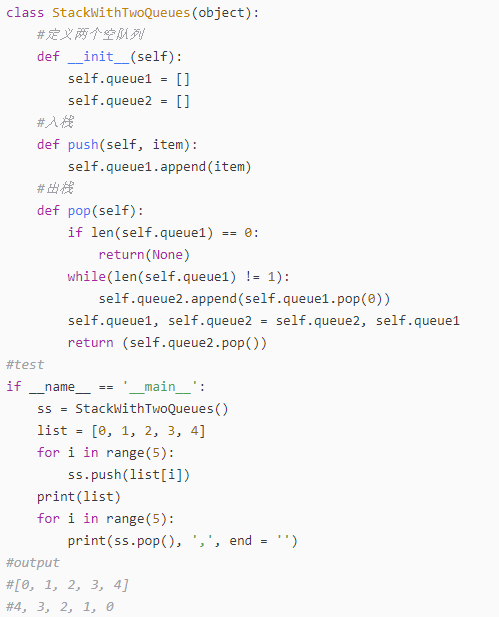
**两个队列实现一个栈？**

进栈：元素入队列A

出栈：判断如果队列A只有一个元素，则直接出队列

否则，把队A中的元素出队并入队B，直到队A中只有一个元素，再直接出队。为了下一次继续操作，互换队A和队B





* 对队列的前k个元素倒序
* 使用队列生成从1到n的二进制数

## 栈（后进先出）

### 基本操作

Push——在顶部插入一个元素

Pop——返回并移除栈顶元素

isEmpty——如果栈为空，则返回true

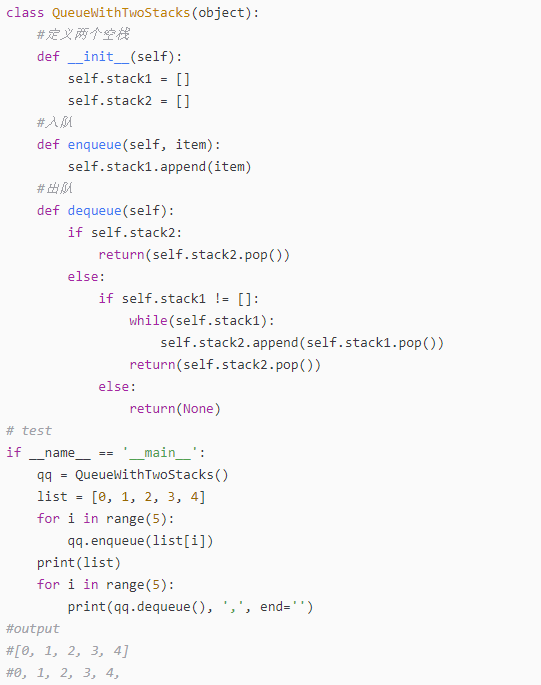
Top——返回顶部元素，但并不移除它

### 常见问题

* 两个栈实现一个队列

入队：元素进栈A

出队：先判断栈B是否为空，为空则将栈A中的元素 pop 出来并 push 进栈B，再栈B出栈，如不为空则栈B直接出栈。



* 使用栈计算后缀表达式
* 对栈的元素进行排序
* 判断表达式是否括号平衡

## 链表

其中每个节点包含着数据和指向后续节点的指针。 链表还包含一个头指针，它指向链表的第一个元素，但当列表为空时，它指向null或无具体内容

主要用途：

链表一般用于实现**文件系统**、**哈希表**和**邻接表**

### 基本操作

单链表（单向）、双向链表（双向）

InsertAtEnd - 在链表的末尾插入指定元素

InsertAtHead - 在链接列表的开头/头部插入指定元素

Delete  - 从链接列表中删除指定元素

DeleteAtHead - 删除链接列表的第一个元素

Search  - 从链表中返回指定元素

isEmpty - 如果链表为空，则返回true

### 常见问题

* 反转链表
* 检测链表中的循环
* 返回链表倒数第N个节点
* 删除链表中的重复项

## 树

层级式：由顶点（节点）和连接它们的边组成，不存在环路

### 常见问题

* 求二叉树的高度
* 在二叉搜索树中查找第k个最大值
* 查找与根节点距离k的节点
* 在二叉树中查找给定节点的祖先节

## 图

一组以网络形式相互连接的节点

权重/成本，显示从顶点x到y所需的成本

无向图、有向图

邻接矩阵、邻接表

**遍历算法：广度优先搜索、深度优先搜索**

### 常见问题

* 实现广度和深度优先搜索
* 检查图是否为树
* 计算图的边数
* 找到两个顶点之间的最短路径

## 字典树

快速检索，用于搜索字典中的单词，在搜索引擎中自动提供建议

### 常见问题

* 计算字典树中的总单词数
* 打印存储在字典树中的所有单词
* 使用字典树对数组的元素进行排序
* 使用字典树从字典中形成单词
* 构建T9字典（字典树+ DFS）

## 哈希表

用于唯一标识对象并将每个对象存储在一些预先计算的唯一索引（称为“键（key）”）中的过程。键值对

* 哈希函数
* 哈希表的大小
* 碰撞处理方法

### 常见问题

* 在数组中查找对称键值对
* 追踪遍历的完整路径
* 查找数组是否是另一个数组的子集
* 检查给定的数组是否不相交

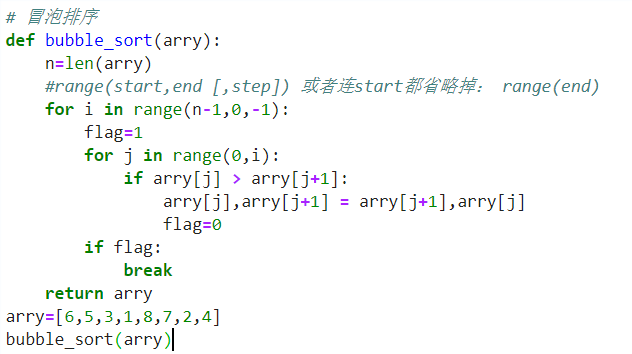
# 算法

## 排序

参考博客：<https://www.cnblogs.com/zhizhan/p/4549099.html>

### 冒泡排序

冒泡排序的原理非常简单，它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。



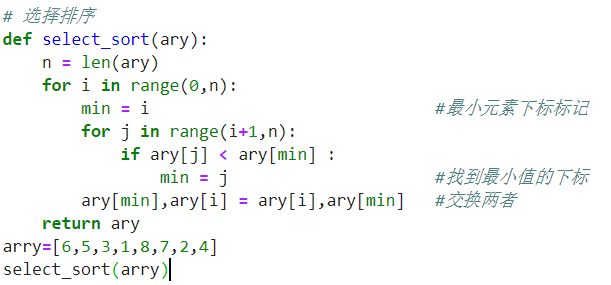
### 选择排序

选择法也算是入门的一种排序算法，比起冒泡法，它的方法巧妙了一些，它的出发点在于“挑”，每次挑选数组的最值，与前置元素换位，然后继续挑选剩余元素的最值并重复操作。

最简单直观的排序，工作原理如下：

* 在未排序序列中找到最小（大）元素，存放到排序序列的起始位置。
* 再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。
* 以此类推，直到所有元素均排序完毕。

Python 实现



### 插入排序

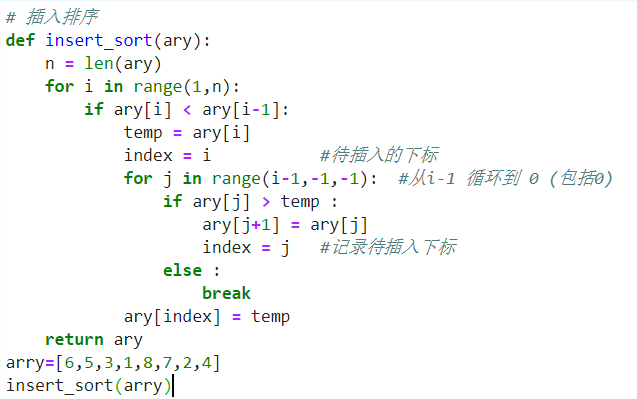
插入法和之前两个排序对比，并不在于如何按顺序的“取”，而在于如何按数序的“插”。

对每个未排序的数据，在已排序序列中从后向前扫描，找到相应位置并插入。

步骤：

* 从第一个元素开始，该元素可以认为已经被排序
* 取出下一个元素，在已经排序的元素序列中从后向前扫描
* 如果被扫描的元素（已排序）大于新元素，将该元素后移一位
* 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置
* 将新元素插入到该位置后
* 重复步骤2~5

Python实现



### 快速排序

是一种分区交换排序算法。采用分治策略对两个子序列再分别进行快速排序，是一种递归算法。

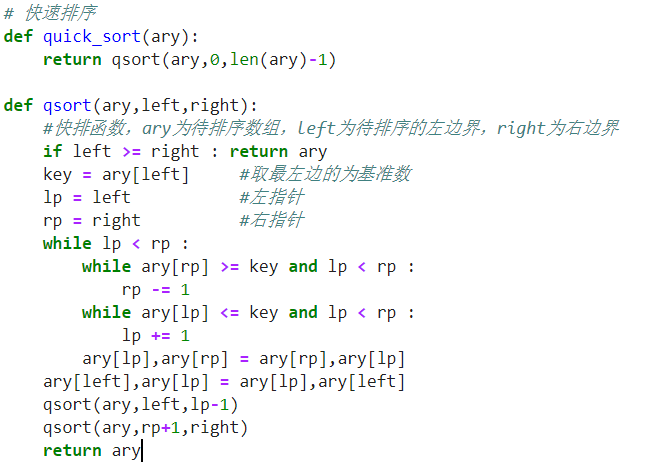
稳定：否

时间复杂度：

* 最优：O(nlog(n))
* 最差：O(n^2)
* 平均：O(nlog(n))



Python实现 **数组**实现



python的这种列表推导的写法，简化了代码书写，却牺牲了资源——这也就是快速排序难的部分，需要在原数组进行排序，也就是不使用额外的空间

### 归并排序

归并排序是采用分治法的一个非常典型的应用。归并排序的思想就是先递归分解数组，再合并数组。

稳定：是

时间复杂度：

* 最优：O(nlog(n))
* 最差：O(nlog(n))
* 平均：O(nlog(n))

Python实现



### 堆排序

堆排序在 top K 问题中使用比较频繁。堆排序是采用二叉堆的数据结构来实现的，虽然实质上还是**一维数组**。二叉堆是一个**近似完全二叉树**。

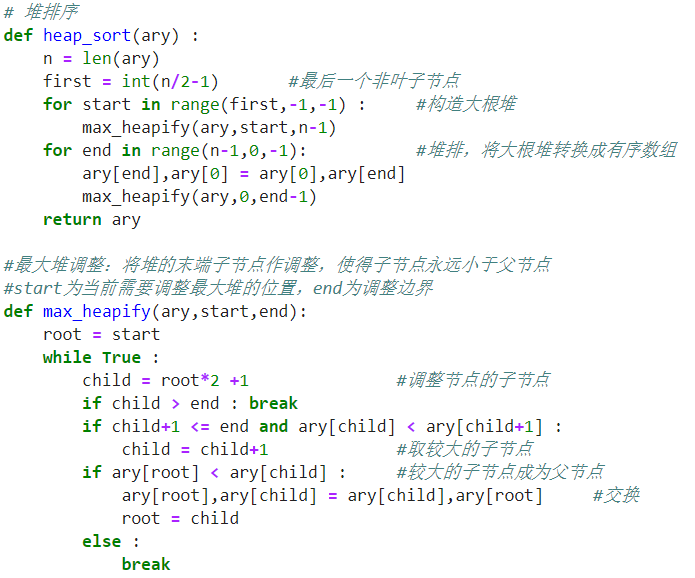
**二叉堆**的性质：

* 父节点的键值总是>=(<=)任何一个子节点的键值
* 每个节点的左右子树都是一个二叉堆（最大堆或最小堆）

步骤：

* **构造最大堆**（Build\_Max\_Heap）：若数组下标范围为0~n，考虑到单独一个元素是大根堆，则从下标n/2开始的元素均为大根堆。于是只要从n/2-1开始，向前依次构造大根堆，这样就能保证，构造到某个节点时，它的左右子树都已经是大根堆。
* **堆排序**（HeapSort）：由于堆是用数组模拟的。得到一个大根堆后，数组内部并不是有序的。因此需要将堆化数组有序化。思想是移除根节点，并做最大堆调整的递归运算。第一次将heap[0]与heap[n-1]交换，再对heap[0...n-2]做最大堆调整。第二次将heap[0]与heap[n-2]交换，再对heap[0...n-3]做最大堆调整。重复该操作直至heap[0]和heap[1]交换。由于每次都是将最大的数并入到后面的有序区间，故操作完后整个数组就是有序的了。
* 最大堆调整（Max\_Heapify）：该方法是提供给上述两个过程调用的。目的是将堆的末端子节点作调整，使得子节点永远小于父节点

Python 实现：



### 二叉树排序

这是另一个使用树进行排序的方法，和堆排序不同的是，这种方法需要真正的构建二叉树，而不是使用数组的二叉树形式。它的核心在与构建二叉树时的顺序以及输入二叉树时的顺序。

步骤：

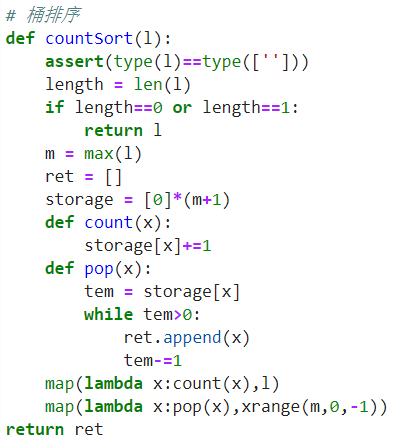
* 依次读取待排序数组的元素，并将其添加为一个二叉树的节点；
* 添加的时候，按值的大小放在节点的左右，如果左右节点已经被占用，则递归到子节点进行添加。
* 二叉树输出的时候，采取前序遍历或者后序遍历的方式输出

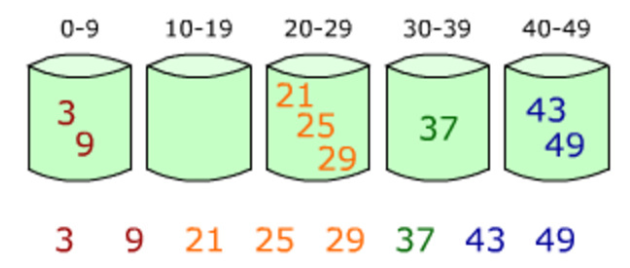
### 桶排序

桶排序是一种将元素分到一定数量的桶中的排序算法。每个桶内部采用其他算法排序，或递归调用桶排序。

时间复杂度：

* 最优：Ω(n + k)
* 最差: O(n^2)
* 平均：Θ(n + k)





例题：

一年的全国高考考生人数为**500 万**，分数使用标准分，最低100 ，最高900 ，没有小数，你把这500 万元素的数组排个序。

**分析**：对500W数据排序，如果基于比较的先进排序，平均比较次数为O(5000000\*log5000000)≈1.112亿。但是我们发现，这些数据都有特殊的条件： 100=<score<=900。那么我们就可以考虑桶排序这样一个“投机取巧”的办法、让其在毫秒级别就完成500万排序。

**方法**：创建801(900-100)个桶。将每个考生的分数丢进f(score)=score-100的桶中。这个过程从头到尾遍历一遍数据只需要500W次。然后根据桶号大小依次将桶中数值输出，即可以得到一个有序的序列。而且可以很容易的得到100分有\*\*\*人，501分有\*\*\*人。

实际上，桶排序对数据的条件有特殊要求，如果上面的分数不是从100-900，而是从0-2亿，那么分配2亿个桶显然是不可能的。所以桶排序有其局限性，适合元素值集合并不大的情况。

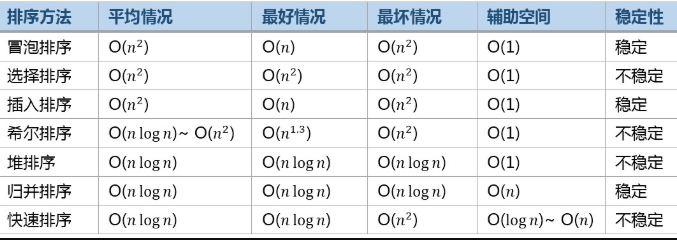
### 基数排序

基数排序类似于桶排序，将元素分发到一定数目的桶中。不同的是，基数排序在分割元素之后没有让每个桶单独进行排序，而是直接做了合并操作。

时间复杂度：

* 最优：Ω(nk)
* 最差: O(nk)
* 平均：Θ(nk)

### 总结



冒泡、插入、选择排序的时间消耗最大

其他纯比较模型的排序中，快排最快

**一种广泛采取的排序算法：**

**是在数据量很大的时候，采取快速排序的方式；**

**而在当分组很小的时候，使用其他稳定的排序方法。**

### 适用场景：

* 若**n较小**(如n≤50)，可采用**直接插入或直接选择**排序。

当记录规模较小时，直接插入排序较好；否则因为直接选择移动的记录数少于直接插人，应选直接选择排序为宜。

* 若文件初始状态**基本有序**(指正序)，则应选用**直接插人、冒泡或随机的快速排序**为宜；
* 若**n较大**，则应**采用时间复杂度为O(nlgn)**的排序方法：**快速排序、堆排序或归并排序**。

快速排序是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；

堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况。这两种排序都是**不稳定**的。

若要求排序稳定，则可选用归并排序。

但前面介绍的从单个记录起进行两两归并的排序算法并不值得提倡，通常可以将它和直接插入排序结合在一起使用。先利用直接插入排序求得较长的有序子序列，然后再两两归并之。因为直接插入排序是稳定的，所以改进后的归并排序仍是稳定的。

## 查找（图算法）

### 深度优先搜索

### 广度优先搜索

拓扑排序

Dijkstra算法

Bellman-Ford算法

Floyd-Warshall 算法

最小生成树算法

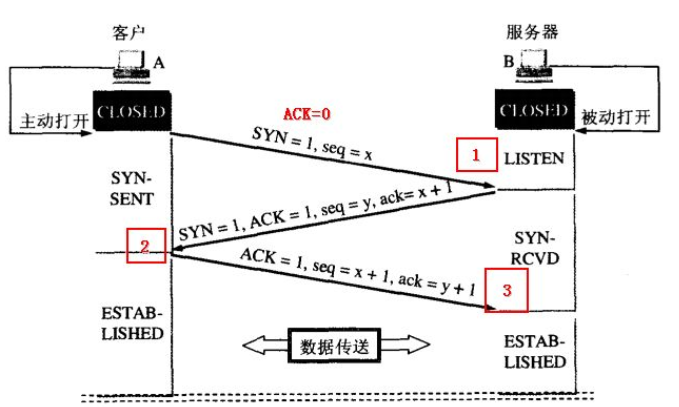
Kruskal 算法

贪心算法

位运算

# 计算机网络

## Tcp三次握手



第一次握手：客户端随机选取一个序列号作为自己的初始序号发送给服务端

第二次握手：服务端使用ack对客户端的数据进行确认，因为已经收到了序列号为x的数据包，准备接收序列号为x+1的包，所以ack=x+1，同时B告诉A自己的初始序列号，就是seq=y；

第三次握手：客户端告诉告诉服务端收到了服务端的确认消息并准备建立连接，A自己此条消息的序列号是x+1，所以seq=x+1，而ack=y+1是表示A正准备接收B序列号为y+1的数据包。

* **为什么需要第三次握手？**

主要是为了防止已失效的连接请求报文段突然又传送到了B，因而产生错误。“已失效的保温请求报文”是指网络延迟，以致延误到连接释放后的某个时间才到达B，但B受到此失效报文后，误以为是新的连接请求，于是建立新的连接，向A发出确认报文。若不采用三次握手，那么只要B发出确认，新的连接就建立了，由于A并没有发出建立连接的请求，因此不会理睬B的确认，也不会向B发送数据，但B却以为新的运输连接已经建立，并一直等待A发来数据，故B的许多资源就这样白白浪费了。

## Tcp四次挥手

TCP连接是双向的，所以需要客户端和服务器双方都释放连接并确认

## 点击按钮后完整过程

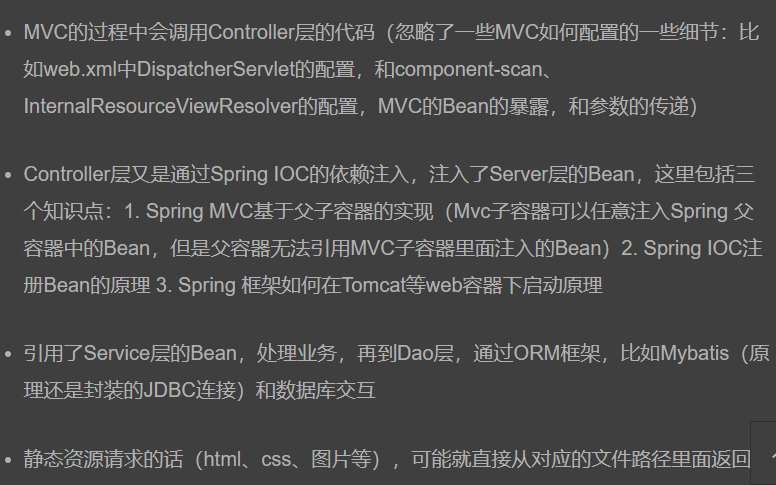
1. 域名解析（DNS解析）：逐级从浏览器、本地操作系统、路由器、 运营商缓存，还找不到，电信运行商会根据域名去对应的域名服务器查找IP信息
2. TCP三次握手（已经DNS解析了域名对应IP，请求URL地址，默认端口80，实现TCP连接）
3. 成功建立TCP连接开始通讯
4. 浏览器发起HTTP请求，一个完整的http请求信息由3部分组成：请求行、请求头部、请求数据

请求行：Request Method（GET、POST、PUT、DELETE等）、Request URL、请求的协议（http1.0、http1.1，现在浏览器大部分默认都是http1.1协议)

请求头部：Connection属性、Accept Encoding（服务端可以发送的压缩数据的格式）、Accept Charset等信息

请求数据：get请求的话就是空的，post请求里面就会有请求的相关参数和参数值

1. 服务器处理收到的请求



1. 服务器返回一个响应报文，包括状态行、响应头部、响应正文

状态行：请求协议、状态码、状态码描述

响应头部：Content-Type（Server: Apache/2.2.8 PHP/5.2.5）、Content-Length、Content-Charset等信息

响应正文：html、js、css等

1. 浏览器解析html代码并请求html中的js/css/图片等资源
2. 浏览器进行页面渲染呈现给用户
3. 数据传输完毕，TCP 4次挥手断开连接。

# 操作系统

## 进程线程的区别

并发的认识：服务器要同时对多个客户端进行相应

并发的方式：多进程、多线程、IO多路复用

进程：是资源（cpu、资源）分配的基本单位，是程序执行的一个实例

线程：程序执行的最小单位，是cpu调度和分派的基本单位，线程间共享进程的所有资源

**线程和进程各自有什么区别和优劣呢？**

* 进程是资源分配的最小单位，线程是程序执行的最小单位。
* 进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。而线程是共享进程中的数据的，使用相同的地址空间，因此CPU切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多。
* 线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式（IPC)进行。不过如何处理好同步与互斥是编写多线程程序的难点。
* 但是多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间。

## 内存中堆栈区别

**栈**：**由系统自动分配和释放的，是由高地址向低地址扩展的**数据结构，是一段连续的内存区域，是对数据结构中的栈这种手段的实现。栈的顶地址和最大容量是系统预先设定好的，在程序编译时，就是一个确定的常数。分配**局部变量**

**堆**：向上增长，程序员动态分配释放的内存，malloc（），他的实现跟操作系统和编译器有关，一般内存中的堆使用链表实现的

|  |  |  |
| --- | --- | --- |
|  | 栈 | 堆 |
| 申请方式 | Char a系统自动分配 | Malloc（10）程序员根据需要自己申请 |
| 回收方式 | 系统自动回收，栈上的数据生存周期只是在函数的运行过程中，运行后就释放 | 程序员手动释放 |
| 申请后系统的响应 | 栈的剩余空间>所申请空间，则系统将为程序提供内存，否则报错栈溢出 |  |

# SQL

# 实践项目