Data Structures

Array

Sort

Time complexity

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Merge sort

时间复杂性稳定，但是占用空间大，空间换效率

```
void merge(int arr[], int l, int m, int r)
{
    // Find sizes of two subarrays to be merged
    int n1 = m - l + 1;
    int n2 = r - m;

    /* Create temp arrays */
    int L[] = new int[n1];
    int R[] = new int[n2];

    /*Copy data to temp arrays*/
    for (int i = 0; i < n1; ++i)
        L[i] = arr[l + i];
```

```
    for (int j = 0; j < n2; ++j)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays */

    // Initial indexes of first and second subarrays
    int i = 0, j = 0;

    // Initial index of merged subarry array
    int k = l;
    // 轮流放入原array
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy remaining elements of L[] if any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy remaining elements of R[] if any */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Main function that sorts arr[l..r] using
// merge()
void sort(int arr[], int l, int r)
{
    if (l < r) {
        // Find the middle point
        int m = (l + r) / 2;
```

```
        // Sort first and second halves
        sort(arr, l, m);
        sort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
```

Quick sort

每次寻找一个pivot，将比pivot小的划至左侧，大的划至右侧。对左右侧依次分割排序。

```
// Java program for implementation of QuickSort
class QuickSort
{
    /* This function takes last element as pivot,
       places the pivot element at its correct
       position in sorted array, and places all
       smaller (smaller than pivot) to left of
       pivot and all greater elements to right
       of pivot */
    int partition(int arr[], int low, int high)
    {
        int pivot = arr[high];
        int i = (low-1); // index of smaller element
        for (int j=low; j<high; j++)
        {
            // If current element is smaller than the pivot
            if (arr[j] < pivot)
            {
                i++;

                // swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // swap arr[i+1] and arr[high] (or pivot)
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;

        return i+1;
```

```
    }


    /* The main function that implements QuickSort()
      arr[] --> Array to be sorted,
      low  --> Starting index,
      high  --> Ending index */
    void sort(int arr[], int low, int high)
    {
        if (low < high)
        {
            /* pi is partitioning index, arr[pi] is
              now at right place */
            int pi = partition(arr, low, high);

            // Recursively sort elements before
            // partition and after partition
            sort(arr, low, pi-1);
            sort(arr, pi+1, high);
        }
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i]+" ");
        System.out.println();
    }

    // Driver program
    public static void main(String args[])
    {
        int arr[] = {10, 7, 8, 9, 1, 5};
        int n = arr.length;

        QuickSort ob = new QuickSort();
        ob.sort(arr, 0, n-1);

        System.out.println("sorted array");
        printArray(arr);
    }
}
/*This code is contributed by Rajat Mishra */
```

Tim sort

java 内置的排序算法，调用方法为Arrays.sort(arr)。

如需传入自定义比较方式可创建一个继承自Comparator 的类

```
// 创建一个comparator类
class NumComparator implements Comparator<NameTag> {
    public int compare (NameTag left,NameTag right) {
        return(left.getNumber() - right.getNumber());
    }
}
// 调用如下
Arrays.sort(tags, new NumComparator());
```

LinkedList

快慢指针能解决很多问题

Tree

通常tree类的题目会使用很多的递归，做题时要想好返回条件，遍历顺序。

Binary search tree

Algorithms

Graph

BFS

把遍历的node加入一个**queue**，first in first out。

可以用一个while-loop来实现

DFS

访问的node加入一个stack，first in last out。

可以用while-loop加上stack实现或者递归。

Network

时延

总时延 = 排队时延 + 处理时延 + 传输时延 + 传播时延

排队时延

分组在路由器的输入队列和输出队列中排队等待的时间，取决于网络当前的通信量。

处理时延

主机或路由器收到分组时进行处理所需要的时间，例如分析首部、从分组中提取数据、进行差错检验或查找适当的路由等。

传输时延

主机或路由器传输数据帧所需要的时间。

$delay = \frac{l}{v}$

其中 l 表示数据帧的长度，v 表示传输速率。

传播时延

电磁波在信道中传播所需要花费的时间，电磁波传播的速度接近光速。

其中 l 表示信道长度，v 表示电磁波在信道上的传播速度。

网络结构

1. 五层协议

- **应用层**： 为**特定应用程序**提供数据传输服务，例如 HTTP、DNS 等协议。数据单位为**报文**。
- **传输层**： 为**进程**提供通用数据传输服务。由于应用层协议很多，定义通用的传输层协议就可以支持不断增多的应用层协议。运输层包括两种协议：传输控制协议 TCP，提供面向连接、可靠的数据传输服务，数据单位为**报文段**；用户数据报协议 UDP，提供无连接、尽最大努力的数据传输服务，数据单位为用户数据报。TCP 主要提供完整性服务，UDP 主要提供及时性服务。
- **网络层**： 为**主机**提供数据传输服务。而传输层协议是为主机中的进程提供数据传输服务。网络层把传输层传递下来的报文段或者用户数据报**封装成分组**。
- **数据链路层**： 网络层针对的还是主机之间的数据传输服务，而主机之间可以有很多链路，链路层协议就是为同一链路的主机提供数据传输服务。数据链路层把网络层传下来的分组**封装成帧**。
- **物理层**： 考虑的是怎样在传输媒体上传输数据比特流，而不是指具体的传输媒体。物理层的作用是尽可能屏蔽传输媒体和通信手段的差异，使数据链路层感觉不到这些差异。

数据链接层

信道复用技术

- 频分复用： 相同时间占用不同频率的带宽资源

- 时分复用： 不同时间占用相同的频率带宽资源
- 统计时分复用： 不固定时分复用帧的位置，有数据就集中一组发送

MAC

固定的机器编码，用与交换机的机器对端口映射

交换机会自学习，更新翻译表

网络层

VPN

VPN 使用公用的互联网作为本机构各专用网之间的通信载体。专用指机构内的主机只与本机构内的其它主机通信；虚拟指好像是，而实际上并不是，它有经过公用的互联网。

（场所A）-- （路由A）-- 【互联网/隧道】-- （路由B）-- （场所B）

NAT

将内网ip+端口转化为外网ip+端口，内网主机公用一个外网ip，使用不同端口来转发不同内网ip

路由协议

1. RIP
    1. 两个路由之间的跳跃数计做开销，最多跳跃15次
    2. 周期性与相邻路由器更新路由表，最终知道到达每一个网络内路由的距离以及下一跳的路由
    3. 实现简单但是限制网络规模
2. OSPF
    1. 向所有路由广播自己的链接状态（仅在链接变化时广播）。最终所有路由都具有全网的拓扑结构
    2. 路径计算使用Dijkstra的最短路径算法
    3. OSPF收敛速度比RIP快
3. BGP
    1. 每个AS之间选用比较好的路由尽心通信
    2. 可以限制其他AS通过自己转发
    3. 必须配置BGP发言人来进行路由信息交换

传输层

网络层只把分组发送到目的主机，但是真正通信的并不是主机而是主机中的进程。传输层提供了进程间的逻辑通信，传输层向高层用户屏蔽了下面网络层的核心细节，使应用程序看**起来像是在两个传输层实体之间有一条端到端的逻辑通信信道。**

UDP

- 用户数据报协议 UDP（User Datagram Protocol）是无连接的，尽最大可能交付，没有拥塞控制，面向报文（对于应用程序传下来的报文不合并也不拆分，只是添加 UDP 首部），支持一对一、一对多、多对一和多对多的交互通信。

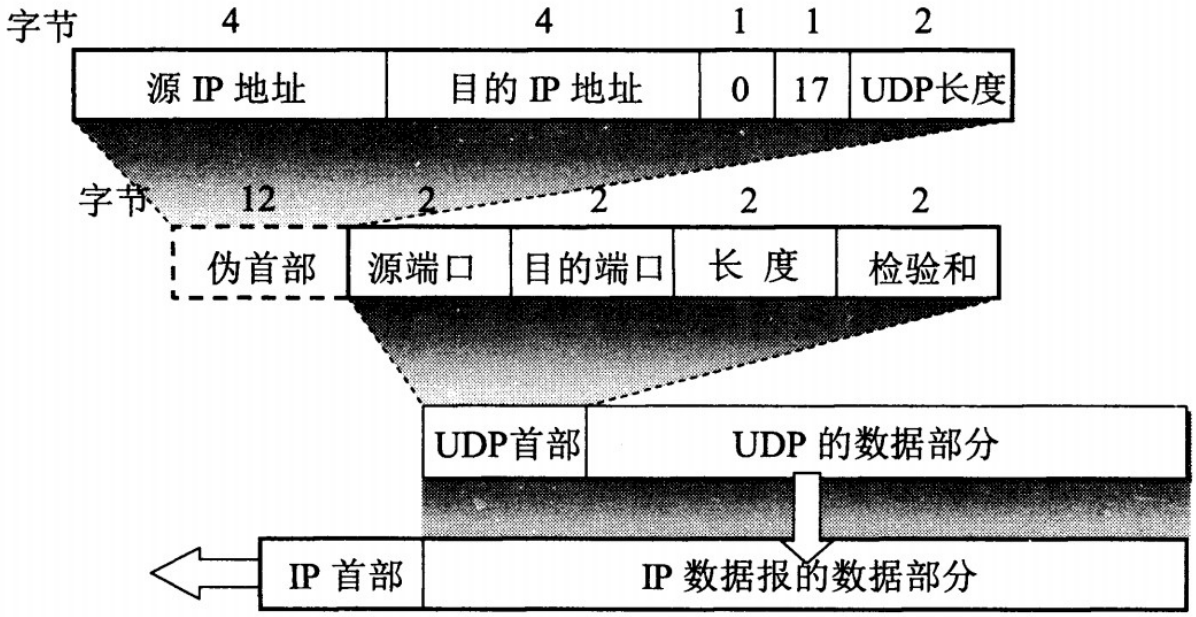

图 5-5　UDP 用户数据报的首部和伪首部

首部字段只有 8 个字节，包括源端口、目的端口、长度、检验和。12 字节的伪首部是为了计算检验和临时添加的。

TCP

- 传输控制协议 TCP（Transmission Control Protocol）是面向连接的，提供可靠交付，有流量控制，拥塞控制，提供全双工通信，面向字节流（把应用层传下来的报文看成字节流，把字节流组织成大小不等的数据块），每一条 TCP 连接只能是点对点的（一对一）。

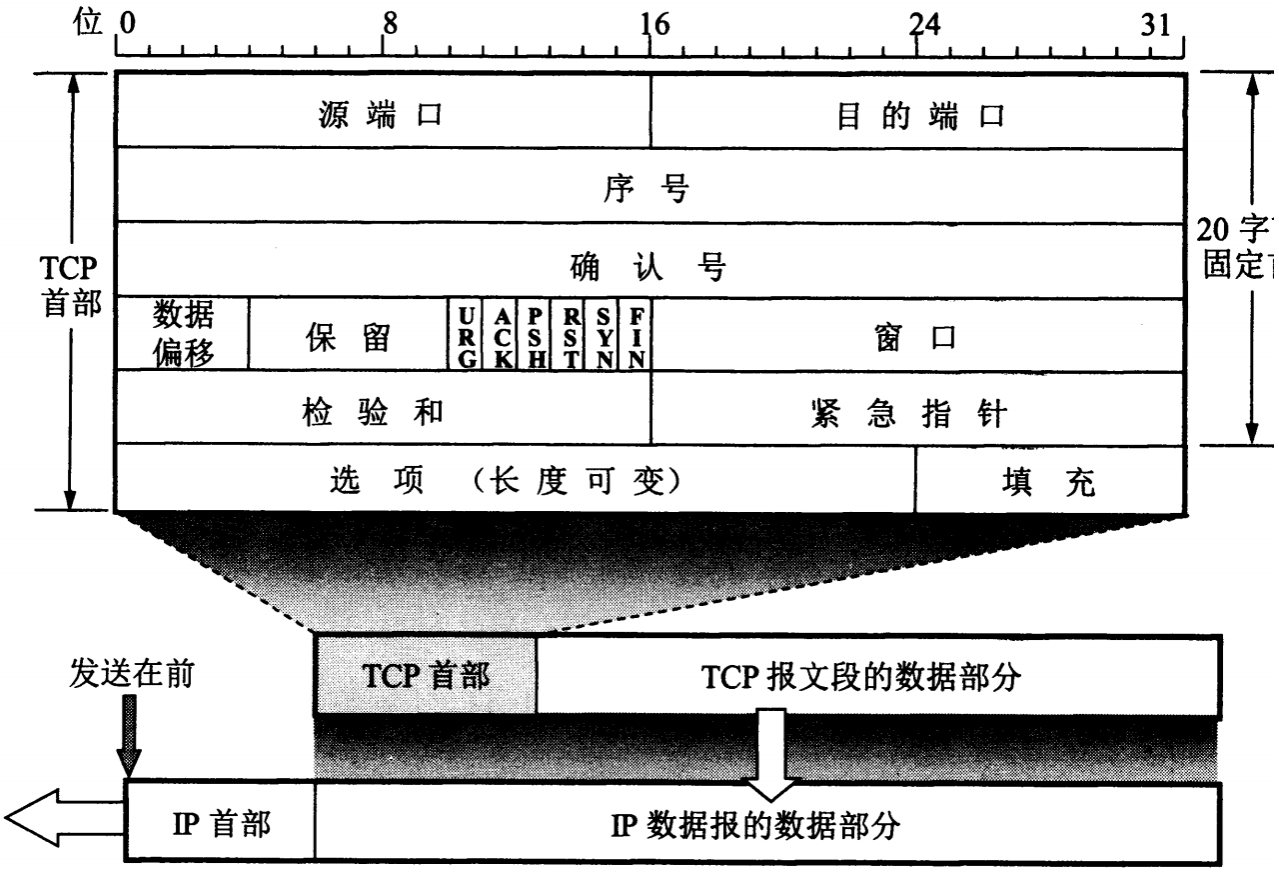

图 5-14　TCP 报文段的首部格式

- **序号**： 用于对字节流进行编号，例如序号为 301，表示第一个字节的编号为 301，如果携带的数据长度为 100 字节，那么下一个报文段的序号应为 401。
- **确认号**： 即ack，期望收到的下一个报文段的序号。例如 B 正确收到 A 发送来的一个报文段，序号为 501，携带的数据长度为 200 字节，因此 B 期望下一个报文段的序号为 701，**B 发送给 A 的确认报文段**中确认号就为 701。
- **数据偏移**： 指的是数据部分距离报文段起始处的偏移量，实际上指的是首部的长度。
- **确认 ACK**： 当 ACK=1 时确认号字段有效，否则无效。TCP 规定，在连接建立后所有传送的报文段都必须把 ACK 置 1。
- **同步 SYN**： 在连接建立时用来同步序号。当 SYN=1，ACK=0 时表示这是一个连接请求报文段。若对方同意建立连接，则响应报文中 SYN=1，ACK=1。
- **终止 FIN**： 用来释放一个连接，当 FIN=1 时，表示此报文段的发送方的数据已发送完毕，并要求释放连接。
- **窗口**： 窗口值作为接收方让发送方设置其发送窗口的依据。之所以要有这个限制，是因为接收方的数据缓存空间是有限的

TCP三次握手

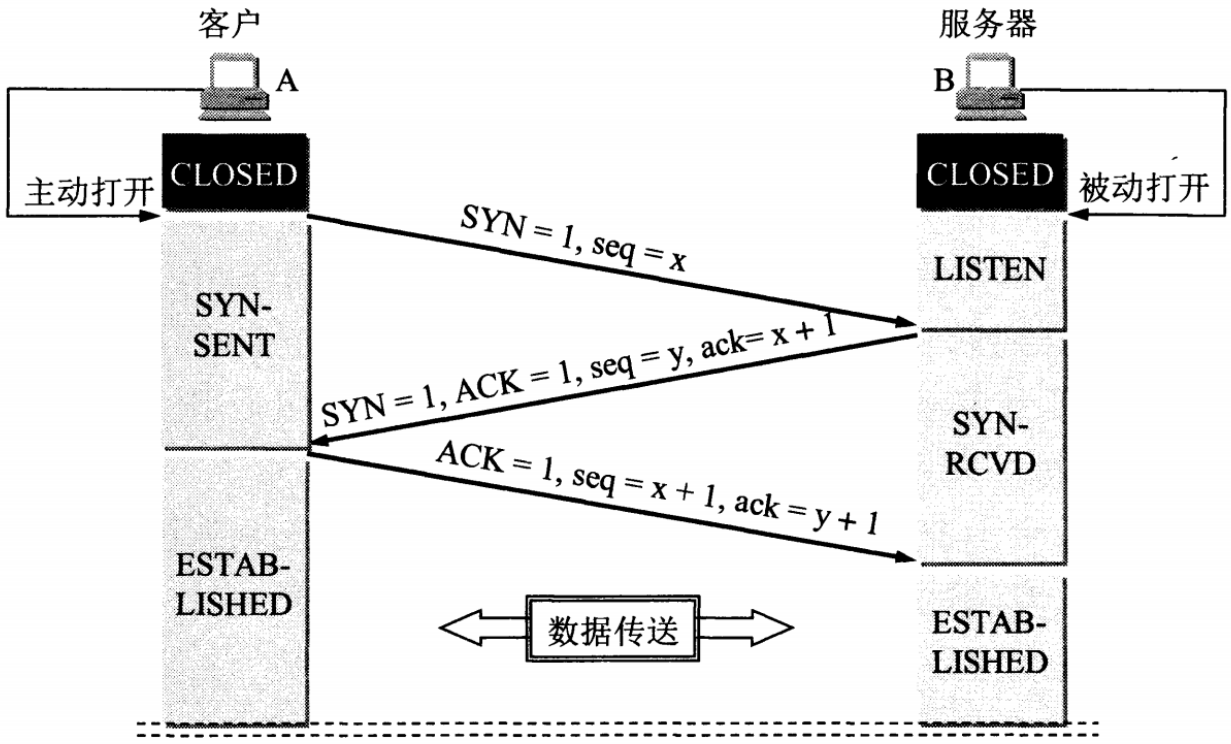

图 5-28 用三报文握手建立 TCP 连接

假设 A 为客户端，B 为服务器端。

- 首先 B 处于 LISTEN（监听）状态，等待客户的连接请求。
- A 向 B 发送连接请求报文，SYN=1，ACK=0，选择一个初始的序号 x。
- B 收到连接请求报文，如果同意建立连接，则向 A 发送连接确认报文，SYN=1，ACK=1，确认号为 x+1，同时也选择一个初始的序号 y。
- A 收到 B 的连接确认报文后，还要向 B 发出确认，确认号为 y+1，序号为 x+1。
- B 收到 A 的确认后，连接建立。

三次握手的原因（建立连接）

第三次握手是为了防止失效的连接请求到达服务器，让服务器错误打开连接。

客户端发送的连接请求如果在网络中滞留，那么就会隔很长一段时间才能收到服务器端发回的连接确认。客户端等待一个超时重传时间之后，就会重新请求连接。但是这个滞留的连接请求最后还是会到达服务器，如果不进行三次握手，那么服务器就会打开两个连接。如果有第三次握手，客户端会忽略服务器之后发送的对滞留连接请求的连接确认，不进行第三次握手，因此就不会再次打开连接。

TCP四次挥手

以下描述不讨论序号和确认号，因为序号和确认号的规则比较简单。并且不讨论 ACK，因为 ACK 在连接建立之后都为 1。

- A 发送连接释放报文，FIN=1。
- B 收到之后发出确认，此时 TCP 属于半关闭状态，B 能向 A 发送数据但是 A 不能向 B 发送数据。
- 当 B 不再需要连接时，发送连接释放报文，FIN=1。
- A 收到后发出确认，进入 TIME-WAIT 状态，等待 2 MSL（最大报文存活时间）后释放连接。

- B 收到 A 的确认后释放连接。

四次挥手的原因

客户端发送了 FIN 连接释放报文之后，服务器收到了这个报文，就进入了 CLOSE-WAIT 状态。这个状态是为了让服务器端发送还未传送完毕的数据，传送完毕之后，服务器会发送 FIN 连接释放报文。

TIME_WAIT

客户端接收到服务器端的 FIN 报文后进入此状态，此时并不是直接进入 CLOSED 状态，还需要等待一个时间计时器设置的时间 2MSL。这么做有两个理由：

- 确保最后一个确认报文能够到达。如果 B 没收到 A 发送来的确认报文，那么就会重新发送连接释放请求报文，A 等待一段时间就是为了处理这种情况的发生。
- 等待一段时间是为了让本连接持续时间内所产生的所有报文都从网络中消失，使得下一个新的连接不会出现旧的连接请求报文。

应用层

应用	应用层协议	端口号	传输层协议	备注
域名解析	DNS	53	UDP/TCP	长度超过 512 字节时使用 TCP
动态主机配置协议	DHCP	67/68	UDP	
简单网络管理协议	SNMP	161/162	UDP	
文件传送协议	FTP	20/21	TCP	控制连接 21，数据连接 20
远程终端协议	TELNET	23	TCP	
超文本传送协议	HTTP	80	TCP	
简单邮件传送协议	SMTP	25	TCP	
邮件读取协议	POP3	110	TCP	
网际报文存取协议	IMAP	143	TCP	

DNS域名系统

DNS 是一个分布式数据库，提供了主机名和 IP 地址之间相互转换的服务。这里的分布式数据库是指，每个站点只保留它自己的那部分数据。

域名具有层次结构，从上到下依次为：根域名、顶级域名、二级域名。

通常使用UDP传输，限制大小为512字节。

当大小超过512字节或者主域名服务器向辅助服务器发送变化的时候使用TCP协议

动态主机配置协议

DHCP (Dynamic Host Configuration Protocol) 提供了**即插即用**的连网方式，用户不再需要手动配置 IP 地址等信息。

DHCP 配置的内容不仅是 IP 地址，还包括子网掩码、网关 IP 地址。

DHCP 工作过程如下：

1. 客户端发送 Discover 报文，该报文的目的地址为 255.255.255.255:67，源地址为 0.0.0.0:68，被放入 UDP 中，该报文被广播到同一个子网的所有主机上。如果客户端和 DHCP 服务器不在同一个子网，就需要使用中继代理。
2. DHCP 服务器收到 Discover 报文之后，发送 Offer 报文给客户端，该报文包含了客户端所需要的信息。因为客户端可能收到多个 DHCP 服务器提供的信息，因此客户端需要进行选择。
3. 如果客户端选择了某个 DHCP 服务器提供的信息，那么就发送 Request 报文给该 DHCP 服务器。
4. DHCP 服务器发送 Ack 报文，表示客户端此时可以使用提供给它的信息。

电子邮件协议

一个电子邮件系统由三部分组成：用户代理、邮件服务器以及邮件协议。

邮件协议包含发送协议和读取协议，发送协议常用 SMTP，读取协议常用 POP3（读取后会删除邮件） 和 IMAP（读取后服务器上邮件不会被删除）。

Web网页请求过程

1. DHCP配置主机信息
    1. 向DHCP服务器请求IP地址，之后就配置它的 IP 地址、子网掩码和 DNS 服务器的 IP 地址，并在其 IP 转发表中安装默认网关。
2. ARP(address resolution protocol)解析MAC地址
    1. 现在只知道网关路由器的IP，需要得到MAC地址
    2. 主机生成ARP查询报文并广播，网管路由器收到后解开ARP、报文发现查询的IP与自己一致，回复MAC地址给主机
3. DNS解析域名
    1. 网关路由器将DNS解析请求转发给DNS服务器，DNS查询域名并通过网关发送回主机
4. HTTP请求页面
    1. 有了网页服务器IP后可以申请TCP连接
    2. 通过HTTP GET向主机请求web内容，服务器将web页面放入报文并发送回诸暨
    3. 渲染web页面内容并显示

2. OSI

其中表示层和会话层用途如下：

- **表示层**： 数据压缩、加密以及数据描述，这使得应用程序不必关心在各台主机中数据内部格式不同的问题。
- **会话层**： 建立及管理会话。

五层协议没有表示层和会话层，而是将这些功能留给应用程序开发者处理。

3. TCP/IP

它只有四层，相当于五层协议中数据链路层和物理层合并为网络接口层。

TCP/IP 体系结构不严格遵循 OSI 分层概念，应用层可能会直接使用 IP 层或者网络接口层。

4. 数据在各层之间的传递过程

在向下的过程中，需要添加下层协议所需要的首部或者尾部，而在向上的过程中不断拆开首部和尾部。

路由器只有下面三层协议，因为路由器位于网络核心中，不需要为进程或者应用程序提供服务，因此也就不需要传输层和应用层。

Machine Learning

Precision: 挑出的瓜中好瓜的比例

Recall：好瓜中挑出的比例

mAP：mean average percision，pr曲线下面积对处以p

ssd：用vgg16作为骨干网络，在不同的卷积层引出一个全联接层做classification，好处是对于不同大小的物体可以更好的提取特征

Java语言特性

可能的面试问题

1. 垃圾回收机制。。。(主要从下面几方面解答 GC原理、最好画图解释一下年轻代（Eden区和Survival区）、年老代、比例分配及为啥要这样分代回收)
2. 对象分配问题，堆栈里的问题，详细的会问道方法区、堆、程序计数器、本地方法栈、虚拟机栈，问题入口从String a,new String("")开始
3. 关键字，private protected public static final 组合着问
4. Object类里面有哪几种方法，作用
5. equals 和 hashCode方法，重写equals的原则()
6. 向上转型
7. Java引用类型(强引用，软引用，弱引用，虚引用)
8. 线程相关的，主要是volitate，synchorized，wait()，notify()，notifyAll()，join()
9. Exception和Error
10. 反射的用途
11. HashMap实现原理(数组+链表)，查找数据的时间复杂度
12. List有哪些子类，各有什么区别
13. NIO相关，缓冲区、通道、selector。。。(不熟，面了这么多，挂在这里。其实主要是表现在同步阻塞和异步，传输方式不同。标准IO无法实现非阻塞模式、文件锁、读选择、分散聚集等)
14. 内存泄露，举个例子

15. OOM是怎么出现的，有哪几块JVM区域会产生OOM，如何解决(对于该问题，建议去《Java特种兵》的3.6章)
16. Java里面的观察者模式实现
17. 单例实现(我一般用enum写，不容易被挑毛病)
18. 用Java模拟一个栈，并能够做到扩容，并且能有同步锁。（用数组实现）
19. Java泛型机制，泛型机制的优点，以及类型变量

Java 引用类型

https://juejin.im/post/6844903665241686029

通过表格来说明一下，如下：

引用类型	被垃圾回收时间	用途	生存时间
强引用	从来不会	对象的一般状态	JVM停止运行时终止
软引用	当内存不足时	对象缓存	内存不足时终止
弱引用	正常垃圾回收时	对象缓存	垃圾回收后终止
虚引用	正常垃圾回收时	跟踪对象的垃圾回收	垃圾回收后终止

强引用 strong reference

在一个**方法的内部**有一个**强引用**，这个引用保存在`Java`**栈**中，而真正的引用内容(`Object`)保存在`Java`**堆**中。 当这个**方法运行完成**后，就会退出**方法栈**，则引用对象的**引用数**为`0`，这个对象会被回收。

当**内存空间不足**时，`Java`虚拟机宁愿抛出`OutOfMemoryError`错误，使程序**异常终止**，也不会靠随意**回收**具有**强引用**的**对象**来解决内存不足的问题。 如果强引用对象**不使用时**，需要弱化从而使`GC`能够回收

可以显式的将引用设为null，这样对象就会被回收 `object=null;`

软引用 soft reference

如果一个对象只具有**软引用**，则**内存空间充足**时，**垃圾回收器**就**不会**回收它；如果**内存空间不足**了，就会**回收**这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。

> 软引用可用来实现内存敏感的高速缓存。

```
// 强引用
String strongReference = new String("abc");
// 软引用
String str = new String("abc");
SoftReference<String> softReference = new SoftReference<String>(str);
```

软引用可以和一个**引用队列**(`ReferenceQueue`)联合使用。如果**软引用**所引用对象被**垃圾回收**，`JAVA`虚拟机就会把这个**软引用**加入到与之关联的**引用队列**中。

```
    ReferenceQueue<String> referenceQueue = new ReferenceQueue<>();
    String str = new String("abc");
    SoftReference<String> softReference = new SoftReference<>(str,
referenceQueue);

    str = null;
    // Notify GC
    System.gc();

    System.out.println(softReference.get()); // abc

    Reference<? extends String> reference = referenceQueue.poll();
    System.out.println(reference); //null
```

> 注意：软引用对象是在JVM内存不够的时候才会被回收，我们调用System.gc()方法只是起通知作用，JVM什么时候扫描回收对象是JVM自己的状态决定的。就算扫描到软引用对象也不一定会回收它，只有内存不够的时候才会回收。

应用场景：

浏览器的后退按钮。按后退时，这个后退时显示的网页内容是重新进行请求还是从缓存中取出呢？这就要看具体的实现策略了。

1. 如果一个网页在浏览结束时就进行内容的回收，则按后退查看前面浏览过的页面时，需要重新构建；
2. 如果将浏览过的网页存储到内存中会造成内存的大量浪费，甚至会造成内存溢出。

这时候就可以使用软引用，很好的解决了实际的问题：

```
    // 获取浏览器对象进行浏览
    Browser browser = new Browser();
    // 从后台程序加载浏览页面
    BrowserPage page = browser.getPage();
    // 将浏览完毕的页面置为软引用
    SoftReference softReference = new SoftReference(page);

    // 回退或者再次浏览此页面时
    if(softReference.get() != null) {
        // 内存充足，还没有被回收器回收，直接获取缓存
        page = softReference.get();
    } else {
        // 内存不足，软引用的对象已经回收
        page = browser.getPage();
        // 重新构建软引用
```

```
        softReference = new SoftReference(page);
    }
```

弱引用 weak reference

弱引用与**软引用**的区别在于：只具有**弱引用**的对象拥有**更短暂**的**生命周期**。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有**弱引用**的对象，不管当前**内存空间足够与否**，都会**回收**它的内存。不过，由于垃圾回收器是一个**优先级很低的线程**，因此**不一定**会**很快**发现那些只具有**弱引用**的对象。

```
    String str = new String("abc");
    WeakReference<String> weakReference = new WeakReference<>(str);
    str = null;
```

`JVM`首先将**软引用**中的**对象**引用置为`null`，然后通知**垃圾回收器**进行回收：

```
    str = null;
    System.gc();
```

> 注意：如果一个对象是偶尔(很少)的使用，并且希望在使用时随时就能获取到，但又不想影响此对象的垃圾收集，那么你应该用Weak Reference来记住此对象。

下面的代码会让一个**弱引用**再次变为一个**强引用**：

```
    String str = new String("abc");
    WeakReference<String> weakReference = new WeakReference<>(str);
    // 弱引用转强引用
    String strongReference = weakReference.get();
```

同样，**弱引用**可以和一个**引用队列**(`ReferenceQueue`)联合使用，如果**弱引用**所引用的**对象**被**垃圾回收**，`Java`虚拟机就会把这个**弱引用**加入到与之关联的**引用队列**中。

虚引用 phantom reference

虚引用顾名思义，就是**形同虚设**。与其他几种引用都不同，**虚引用**并**不会**决定对象的**生命周期**。如果一个对象**仅持有虚引用**，那么它就和**没有任何引用**一样，在任何时候都可能被垃圾回收器回收。

应用场景：

虚引用主要用来**跟踪对象**被垃圾回收器**回收**的活动。 **虚引用**与**软引用**和**弱引用**的一个区别在于：

> 虚引用必须和引用队列(ReferenceQueue)联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

```
String str = new String("abc");
ReferenceQueue queue = new ReferenceQueue();
// 创建虚引用，要求必须与一个引用队列关联
PhantomReference pr = new PhantomReference(str, queue);
```

程序可以通过判断引用**队列**中是否已经加入了**虚引用**，来了解被引用的对象是否将要进行**垃圾回收**。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的**内存被回收之前**采取必要的行动。

垃圾回收

引用计数算法（Reference Counting Collector）

堆中每个对象（不是引用）都有一个引用计数器。当一个对象被创建并初始化赋值后，该变量计数设置为1。每当有一个地方引用它时，计数器值就加1（a = b， b被引用，则b引用的对象计数+1）。当引用失效时（一个对象的某个引用超过了生命周期（出作用域后）或者被设置为一个新值时），计数器值就减1。任何引用计数为0的对象可以被当作垃圾收集。当一个对象被垃圾收集时，它引用的任何对象计数减1。

优点：引用计数收集器执行简单，判定效率高，交织在程序运行中。对程序不被长时间打断的实时环境比较有利（OC的内存管理使用该算法）。

缺点： 难以检测出对象之间的循环引用。同时，引用计数器增加了程序执行的开销。所以Java语言并没有选择这种算法进行垃圾回收。

早期的JVM使用引用计数，现在大多数JVM采用对象引用遍历（***根搜索算法***）。

根搜索算法（Tracing Collector）

首先了解一个概念：***根集(Root Set)***

所谓根集(Root Set)就是正在执行的Java程序可以访问的引用变量（注意：不是对象）的集合(包括局部变量、参数、类变量)，程序可以使用引用变量访问对象的属性和调用对象的方法。

这种算法的基本思路：

（1）通过一系列名为“GC Roots”的对象作为起始点，寻找对应的引用节点。

（2）找到这些引用节点后，从这些节点开始向下继续寻找它们的引用节点。

（3）重复（2）。

（4）搜索所走过的路径称为引用链，**当一个对象到GC Roots没有任何引用链相连时，就证明此对象是不可的。**

`Java和C#中都是采用根搜索算法来判定对象是否存活的。`

标记可达对象：

JVM中用到的所有现代GC算法在回收前都会先找出所有仍存活的对象。根搜索算法是从离散数学中的图论引入的，程序把所有的引用关系看作一张图。下图3.0中所展示的JVM中的内存布局可以用来很好地阐释这一概念：

图

3.0 标记（marking）对象

首先，垃圾回收器将某些特殊的对象定义为GC根对象。所谓的GC根对象包括：

（1）虚拟机栈中引用的对象（栈帧中的本地变量表）；

（2）方法区中的常量引用的对象；

（3）方法区中的类静态属性引用的对象；

（4）本地方法栈中JNI（Native方法）的引用对象。

（5）活跃线程。

接下来，垃圾回收器会对内存中的整个对象图进行遍历，它先从GC根对象开始，然后是根对象引用的其它对象，比如实例变量。回收器将访问到的所有对象都标记为存活。

存活对象在上图中被标记为蓝色。当标记阶段完成了之后，所有的存活对象都已经被标记完了。其它的那些（上图中灰色的那些）也就是GC根对象不可达的对象，也就是说你的应用不会再用到它们了。这些就是垃圾对象，回收器将会在接下来的阶段中清除它们。

关于标记阶段有几个关键点是值得注意的：

1. 开始进行标记前，需要先暂停应用线程，否则如果对象图一直在变化的话是无法真正去遍历它的。暂停应用线程以便JVM可以尽情地收拾家务的这种情况又被称之为**安全点**（Safe Point），这会触发一次Stop The World(STW)暂停。触发安全点的原因有许多，但最常见的应该就是垃圾回收了。
2. 暂停时间的长短并不取决于堆内对象的多少也不是堆的大小，而是存活对象的多少。因此，调高堆的大小并不会影响到标记阶段的时间长短。
3. 在根搜索算法中，要真正宣告一个对象死亡，至少要经历两次标记过程：
    1. 如果对象在进行根搜索后发现没有与GC Roots相连接的引用链，那它会被第一次标记并且进行一次筛选。筛选的条件是此对象是否有必要执行 finalize（）方法（可看作析构函数，类似于OC中的dealloc，Swift中的deinit）。当对象没有覆盖finalize（）方法，或finalize（）方法已经被虚拟机调用过，虚拟机将这两种情况都视为没有必要执行。

2. 如果该对象被判定为有必要执行finalize（）方法，那么这个对象将会被放置在一个名为F-Queue队列中，并在稍后由一条由虚拟机自动建立的、低优先级的Finalizer线程去执行finalize（）方法。finalize（）方法是对象逃脱死亡命运的最后一次机会（因为一个对象的finalize（）方法最多只会被系统自动调用一次），稍后GC将对F-Queue中的对象进行第二次小规模的标记，如果要在finalize（）方法中成功拯救自己，只要在finalize（）方法中让该对象重新引用链上的任何一个对象建立关联即可。而如果对象这时还没有关联到任何链上的引用，那它就会被回收掉。

4. 实际上GC判断对象是否可达看的是强引用。

   当标记阶段完成后，GC开始进入下一阶段，删除不可达对象。

Java 堆划分

1.年轻代（Young Generation）

几乎所有新生成的对象首先都是放在年轻代的。新生代内存按照**8:1:1**的比例分为**一个Eden区和两个Survivor (Survivor0, Survivor1)区**。大部分对象在Eden区中生成。当新对象生成，Eden Space申请失败（因为空间不足等），则会发起一次**GC(Scavenge GC)**。回收时先将Eden区存活对象复制到一个Survivor0区，然后清空Eden区，当这个Survivor0区也存放满了时，则将Eden区和Survivor0区存活对象复制到另一个Survivor1区，然后清空Eden和这个Survivor0区，此时Survivor0区是空的，然后将Survivor0区和Survivor1区交换，即保持Survivor1区为空， 如此往复。当Survivor1区不足以存放 Eden和Survivor0的存活对象时，就将存活对象直接存放到老年代。当对象在Survivor区躲过一次GC的话，其对象年龄便会加1，默认情况下，如果对象年龄达到15岁，就会移动到老年代中。若是老年代也满了就会触发一次Full GC，也就是新生代、老年代都进行回收。新生代大小可以由-Xmn来控制，也可以用-XX:SurvivorRatio来控制Eden和Survivor的比例。

2.年老代（Old Generation）

在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发Major GC即Full GC，Full GC发生频率比较低，老年代对象存活时间比较长，存活率标记高。一般来说，大对象会被直接分配到老年代。所谓的大对象是指需要大量连续存储空间的对象，最常见的一种大对象就是大数组。比如：

```
byte[] data = new byte[4*1024*1024]
```

这种一般会直接在老年代分配存储空间。

当然分配的规则并不是百分之百固定的，这要取决于当前使用的是哪种垃圾收集器组合和JVM的相关参数。

3.持久代（Permanent Generation）

用于存放静态文件（class类、方法）和常量等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，例如Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。对永久代的回收主要回收两部分内容：废弃常量和无用的类。

永久代空间在Java SE8特性中已经被移除。取而代之的是元空间（MetaSpace）。因此不会再出现“java.lang.OutOfMemoryError: PermGen error”错误。

5.2 堆内存分配策略明确以下三点：

（1）对象优先在Eden分配。

（2）大对象直接进入老年代。

（3）长期存活的对象将进入老年代。

5.3 对垃圾回收机制说明以下三点：

新生代GC（Minor GC/Scavenge GC）：发生在新生代的垃圾收集动作。因为Java对象大多都具有朝生夕灭的特性，因此Minor GC非常频繁(不一定等Eden区满了才触发)，一般回收速度也比较快。在新生代中，每次垃圾收集时都会发现有大量对象死去，只有少量存活，因此可选用复制算法来完成收集。

老年代GC（Major GC/Full GC）：发生在老年代的垃圾回收动作。Major GC，经常会伴随至少一次Minor GC。由于老年代中的对象生命周期比较长，因此Major GC并不频繁，一般都是等待老年代满了后才进行Full GC，而且其速度一般会比Minor GC慢10倍以上。另外，如果分配了Direct Memory，在老年代中进行Full GC时，会顺便清理掉Direct Memory中的废弃对象。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用标记—清除算法或标记—整理算法来进行回收。

新生代采用空闲指针的方式来控制GC触发，指针保持最后一个分配的对象在新生代区间的位置，当有新的对象要分配内存时，用于检查空间是否足够，不够就触发GC。当连续分配对象时，对象会逐渐从Eden到Survivor，最后到老年代。

用Java VisualVM来查看，能明显观察到新生代满了后，会把对象转移到旧生代，然后清空继续装载，当老年代也满了后，就会报outofmemory的异常

Java 函数引用

```
public class User {
    private String username;
    private Integer age;

    public User() {
    }

    public User(String username, Integer age) {
        this.username = username;
        this.age = age;
    }

    @Override
    public String toString() {
        return "User{" +
                "username='" + username + '\'' +
```

```
                ", age=" + age +
                '}';
    }

    // Getter&Setter
}

// Function<para1, para2, ...,  returnVal>
public static void main(String[] args) {
    // 使用双冒号::来构造静态函数引用
    Function<String, Integer> fun = Integer::parseInt;
    Integer value = fun.apply("123");
    System.out.println(value);

    // 使用双冒号::来构造非静态函数引用
    String content = "Hello JDK8";
    Function<Integer, String> func = content::substring;
    String result = func.apply(1);
    System.out.println(result);

    // 构造函数引用
    BiFunction<String, Integer, User> biFunction = User::new;
    User user = biFunction.apply("mengday", 28);
    System.out.println(user.toString());

    // 函数引用也是一种函数式接口，所以也可以将函数引用作为方法的参数
    sayHello(String::toUpperCase, "hello");
}

// 方法有两个参数，一个是
private static void sayHello(Function<String, String> func, String parameter){
    String result = func.apply(parameter);
    System.out.println(result);
}
```

比较：== 和equals（）

在我们实现自己的equals方法之前，equals等价于`==`,而`==`运算符是判断两个对象是不是同一个对象，即他们的**地址是否相等**。而覆写equals更多的是追求两个对象在**逻辑上的相等**，你可以说是**值相等**，也可说是**内容相等**。

> **自反性**：对于任何非空引用值 x，x.equals(x) 都应返回 true。

> **对称性**：对于任何非空引用值 x 和 y，当且仅当 y.equals(x) 返回 true 时，x.equals(y) 才应返回 true。

> **传递性**：对于任何非空引用值 x、y 和 z，如果 x.equals(y) 返回 true， 并且 y.equals(z) 返回 true，那么 x.equals(z) 应返回 true。

> **一致性**：对于任何非空引用值 x 和 y，多次调用 x.equals(y) 始终返回 true 或始终返回 false， 前提是对象上 equals 比较中所用的信息没有被修改。

> **非空性**：对于任何非空引用值 x，x.equals(null) 都应返回 false。

```
public boolean equals(Object anObject) {
        if (this == anObject) {
            return true;
        }
        if (anObject instanceof String) {
            String anotherString = (String)anObject;
            int n = value.length;
            if (n == anotherString.value.length) {
                char v1[] = value;
                char v2[] = anotherString.value;
                int i = 0;
                while (n-- != 0) {
                    if (v1[i] != v2[i])
                        return false;
                    i++;
                }
                return true;
            }
        }
        return false;
    }
```

上面的equals有以下几点诀窍：

- **使用==操作符检查“参数是否为这个对象的引用”**：如果是对象本身，则直接返回，拦截了对本身调用的情况，算是一种性能优化。
- **使用instanceof操作符检查“参数是否是正确的类型”**：如果不是，就返回false，正如对称性和传递性举例子中说得，不要想着兼容别的类型，很容易出错。在实践中检查的类型多半是equals所在类的类型，或者是该类实现的接口的类型，比如Set、List、Map这些集合接口。
- **把参数转化为正确的类型**： 经历了上一步的检测，基本会成功。
- **对于该类中的“关键域”，检查参数中的域是否与对象中的对应域相等**：基本类型的域就用`==`比较，float域用Float.compare方法，double域用Double.compare方法，至于别的引用域，我们一般递归调用它们的equals方法比较，加上判空检查和对自身引用的检查，一般会写成这样：`(field == o.field || (field != null && field.equals(o.field)))`,而上面的String里使用的是数组，

所以只要把数组中的每一位拿出来比较就可以了。

- **编写完成后思考是否满足上面提到的对称性，传递性，一致性等等。**

还有一些注意点。

覆盖equals时一定要覆盖hashCode

equals函数里面一定要是Object类型作为参数

equals方法本身不要过于智能，只要判断一些值相等即可。

Integer == equals

object一般是不能直接比较的，但是比较两个Integer的时候如果value 在-128 - 127之间会自动拆箱使用Integer.valueOf() 来比较值的大小。若超出范围则比较引用地址，就是false了