# 现代网络技术及其应用

## 一 介绍

**大纲：熟悉计算机网络的性能指标的计算，例如速率、时延（传输时延、传播时延、处理时延、排队时延）等**

### 1 速率

**计算机网络 “速率” 简介 :**

① “速率”概念 : 计算机网络上, 主机在数字信道上传送数据位数的速率

② 其它别名 : 又称为数据率, 数据传输率, 比特率

③ 速率单位 : b/s , kb/s , Mb/s , Gb/s , Tb/s , 注意是比特/s , bit , 不是字节Byte

④ 速率单位换算 :

1Tb/s = 103Gb/s=106Mb/s=109Kb/s=1012b/s

**注意速率单位与存储单位的区别 :**

① 存储单位 : 存储单位一般使用 Byte 字节 作为单位 ,

1B（字节） = 8b（比特）

② 存储单位之间的换算 :

1TB = 210GB=1024GB

1GB = 210MB=1024MB

1MB = 210KB=1024KB

1KB = 210B=1024B

1B = 23b = 8bit

### 2 带宽

**概念 :**

* **频域定义** : 某个信号具有的频带宽度,最高频率减去最低频率的差值,单位是赫兹 ( Hz )
* **时域定义** : 计算机网络中的时域定义, 表示单位时间内 网络通信线路传输数据的能力, 网络设备支持的**最高速度**, 单位是 b/s , kb/s , Mb/s , Gb/s , Tb/s
* 带宽指的是网络接口传输数据的能力, 传输介质是光纤/双绞线, 速度都是一样的

### 3 吞吐量

**概念:** 单位时间内,通过某个网络接口的数据量,单位是 b/s , kb/s , Mb/s , Gb/s , Tb/s

吞吐量受**网络带宽/网络额定速率**限制

### 4 时延

**概念:** 数据 从 网络 的一端 传送到 另一端 花费的时间 , 又称为 延迟, 单位是 秒 ( s ) , 毫秒 ( ms )

① **发送时延** : 从 数据分组 的第一个比特位开始计时 , 到最后一个比特位发送完毕 , 所需要的时间

**计算公式为**



上面的信道带宽又称为**发送速率** , 是网络接口的最大传输能力

② **传播时延** : 电磁波在网络介质上传播的时间

**计算公式为**



电磁波在网络介质中的传播速度：

③ **排队时延** : 路由器上 , 等待 可用的 输入 / 输出 链路的时间

④ **处理时延** : 主机 , 路由器上 , 处理接收到的分组信息 , 如提取分组的首部信息 , 检错 , 查找出口 等操作 ; 上述操作花费的时间成为处理时延

### 5 时延带宽积

描述数据量的属性 , 表示在链路上的数据量

**计算公式：**



### 6 往返时延RTT

从 发送方 发送数据开始 , 到 发送方 接收到 接收方 的 确认数据 , 总共经历的时延

**往返时延 RTT 包含的时间：**

* **2倍的传播时延**
* **接收方的处理时延**
* **路由器转发处理时延**

**RTT不包含从 发送方 的发送时延、处理时延**

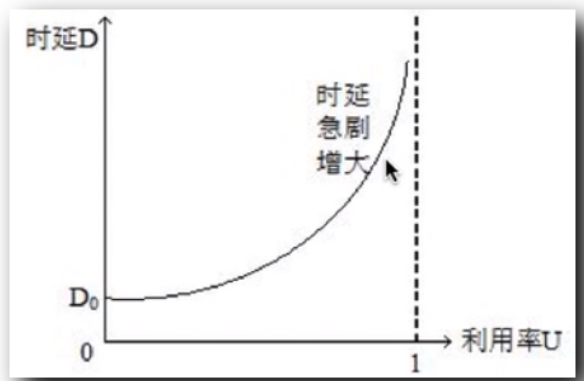
### 7 利用率

**信道利用率不同于网络利用率**



**网络利用率：**将上述所有的信道利用率 , 加权 , 求平均值 ;

**时延与利用率之间的关系** : 利用率越高 , 时延越大 , 此时就会产生网络堵塞 ;



## 二 应用层

### 1 HTTP

#### URL

**URL:** 统一资源定位符，通过下面格式，可以看出，就是用来定位我们所需要资源在服务器上的位置。

**格式：**<协议>://<主机>:<端口>/<路径>

**协议：**http

**主机：**域名/IP地址，原理度一样，到头来还是会转换为IP地址，通过这个才能找到目标服务器

**端口:** 在传输层需要使用的，访问目的主机的哪个端口号。

**路径：**精准的定位我们所需要的资源位置

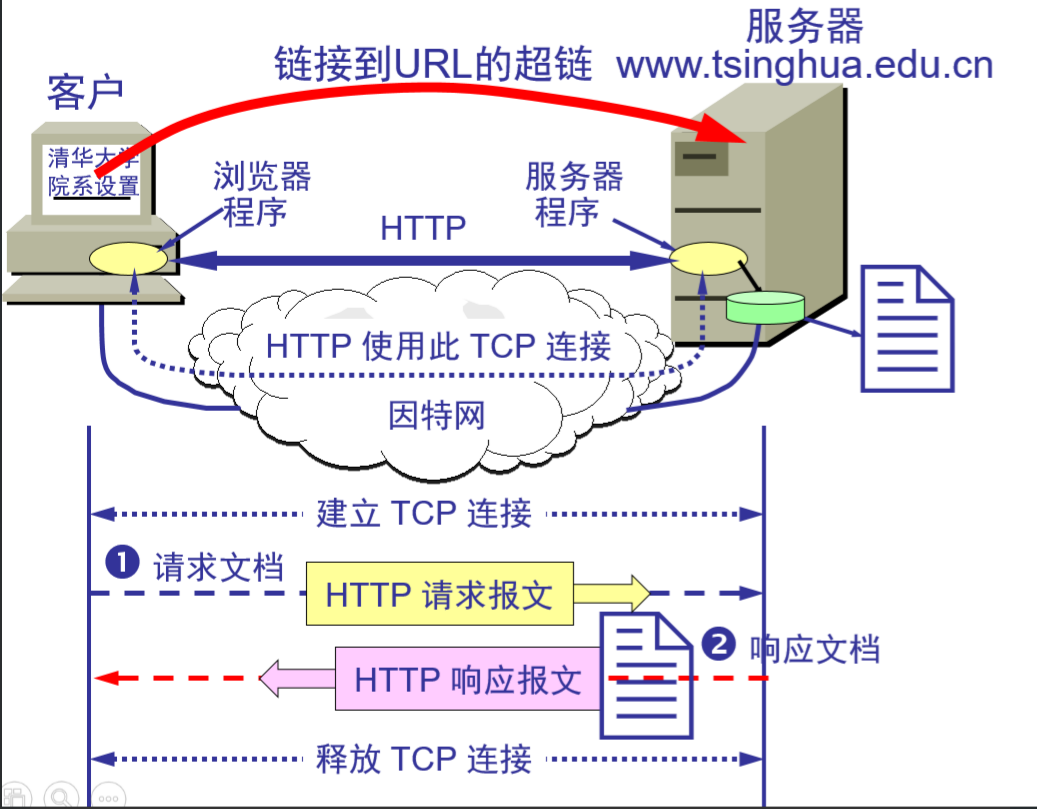
#### 用途

规定了**浏览器如何向万维网服务器请求万维网文档、服务器如何把文档传给浏览器。**

#### 工作流程

每个万维网的网点都有一个服务器进程，它不断的监听TCP端口80，以便发现是否有浏览器向它发出连接请求，一旦监听到连接建立请求，就通过三次握手建立TCP连接，然后浏览器会向服务器发出浏览某个页面的请求，服务器接着返回所请求的页面作为响应，然后TCP连接就被释放了。

  这些响应和请求报文都遵循一定的格式，这就是HTTP协议所规定的。



#### 请求和响应报文的格式

**格式都一样，内容不一样**

请求行 响应行

请求头部 响应体

请求数据 响应数据

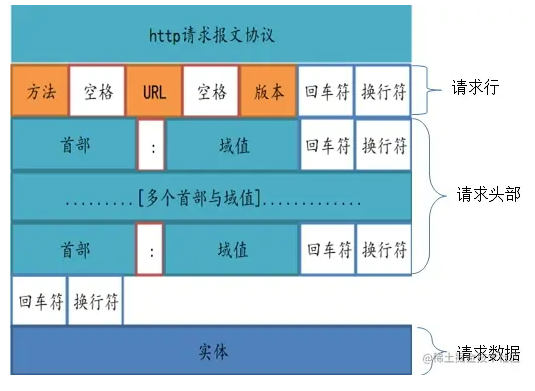
**请求报文格式**

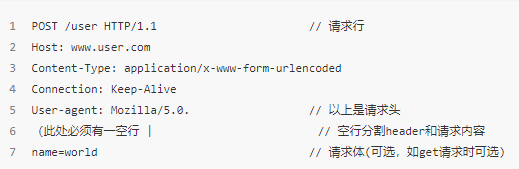
一个HTTP请求报文由请求行（request line） 、请求头（header）、空行和请求体4个部分组成。

**请求行**由三部分组成：**请求方法**、**请求URL**（不包括域名 ） 、**HTTP协议版本**

**请求头**由关键字/值对组成，每行一对

请求头之后是一个**空行**，通知服务器以下不再有请求头





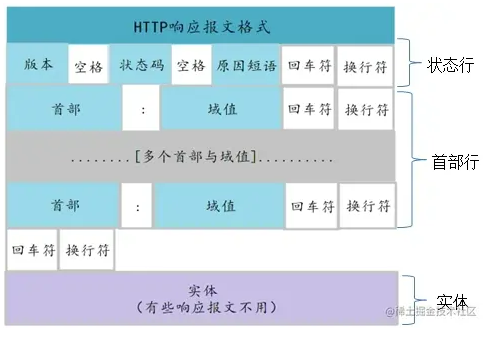
**响应报文格式**

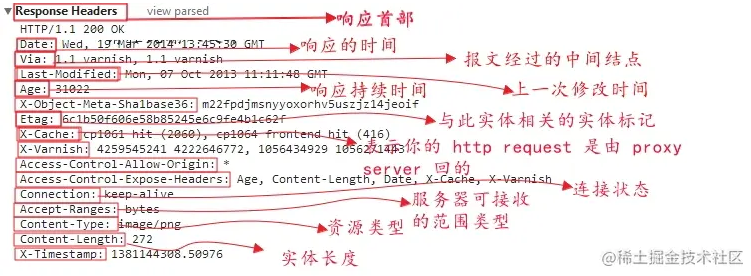
状态行由三部分组成：**服务器HTTP协议版本**，**响应状态码**，**状态码的文本描述**

**状态码：**由3位数字组成，第一个数字定义了响应的类别

**响应头（首部行）**：位于响应报文状态行之后

**响应体**：位于响应首部（首部行）之后





#### 非持久HTTP

非持久HTTP连接（non-persistent HTTP）是在**http/1.0**中定义

**每次服务器发送一个对象后相关的TCP连接就被关闭**，也就是说每个连接没有持续到可以传输其他对象。每个TCP连接只能传送一个请求消息和响应消息。

**缺点：**

1. **客户要为每个待请求的对象建立并维护一个新的连接**。对于每个连接，TCP 必须同时在客户端和服务器端分配 TCP 缓冲区，并维护 TCP 变量。对于有可能同时为来自数百个不同客户的请求提供服务的Web服务器来说，这会严重增加服务器的负担；

2. **对每个对象请求都有2个 RTT 的响应延迟**：一个 RTT 用于建立 TCP 连接，另一个 RTT 用于请求和接收对象；

3. **每个对象都要经历 TCP 缓启动**，因为每个TCP连接都要起始于 slow start 阶段。并行 TCP 连接的使用能够部分减轻 RTT 延迟和缓启动的影响。

##### 案例

某网页由最基本的 HTML 和10个JPEG 图像构成，10个JPEG 图像文件存放在同一台服务器中。如果用户请求该网页并采用“**非持久连接**”，那么在HTTP 客户（通常是用户浏览器）和服务器之间将发生以下操作：

1. HTTP 客户端初始化一个与服务器主机www.server.com中的HTTP服务器的TCP 连接。服务器使用默认端口80监听来自HTTP客户的建立连接请求。

2. HTTP客户端经由与TCP关联的本地Socket发出一个HTTP请求消息（Request）。这个消息中包含路径名 …/index.html。

3. HTTP服务器经由与TCP关联的本地Socket接收到这个请求消息，再从服务器主机的内存或者硬盘中取出文件 …/index.html，经由同一Socket向 HTTP客户端发送包含该文件的响应消息 (Response Message)

4. HTTP服务器通知TCP服务层关闭这个TCP连接；TCP服务器并不立即关闭这个连接，而是在客户收到刚才那个响应消息后才会真正终止这个连接。

5. HTTP客户端经由同一 Socket 接收这个响应消息 (Response Message)。TCP连接随后终止。客户端所收到的消息中封装了客户端所请求的 HTML文件。客户端浏览器从中取出这个文件，加以分析后发现这个文件中还有10个JPEG对象引用。

6. 对每个引用到的JPEG对象重复步骤1～4。

上述例子，用户请求的那个Web页面就产生了**11个TCP连接（1个网页请求连接和10个图象请求连接）**。

#### 持久HTTP

持久HTTP连接（persistent HTTP）是在**http/1.1**中定义，其**默认带流水线**

在持久连接情况下，服务器在发出响应后**保持TCP连接继续打开着**。同一客户/服务器之间的后续请求和响应可以通过这个连接传递。整个Web页面上，比如 1 个基本HTML和 10 个JPEG对象的页面，可以通过单个的持久TCP连接发送。**甚至存放在同一个服务器中的多个Web页面也可以通过单个持久TCP连接发送**。通常，HTTP服务器在某个连接闲置了一段时间后就关闭它，而这段时间通常是可以配置的。

#### 流水线与非流水线的持久HTTP

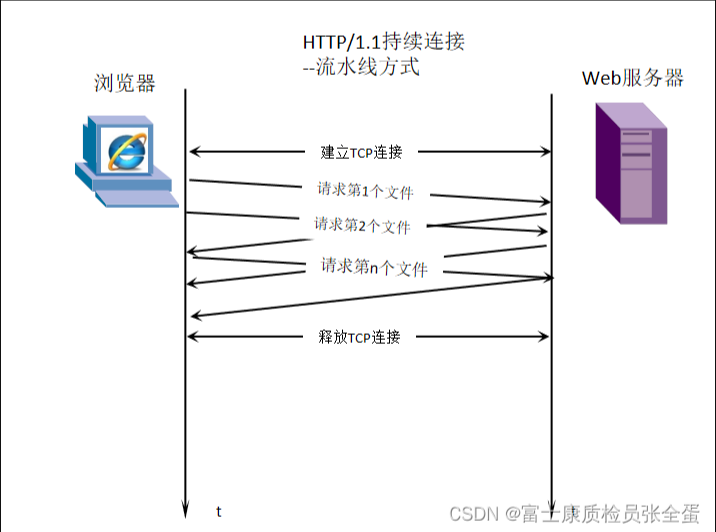
**针对HTTP1.1**

**非流水线方式**：**客户在收到前一个响应后才能发出下一个请求**。这比非持续连接的两倍 RTT 的开销节省了建立 TCP 连接所需的一个 RTT 时间。但服务器在发送完一个对象后，其 TCP 连接就处于空闲状态，浪费了服务器资源。



浏览器访问这个网站，先建立TCP连接，3次握手，请求第一个文件，然后收到，再请求第二个文件，收到。

**流水线方式**：**客户在收到 HTTP 的响应报文之前就能够接着发送新的请求报文**。一个接一个的请求报文到达服务器后，服务器就可连续发回响应报文。使用流水线方式时，**客户访问所有的对象只需花费一个 RTT时间**，使 TCP 连接中的空闲时间减少，提高了下载文档效率。



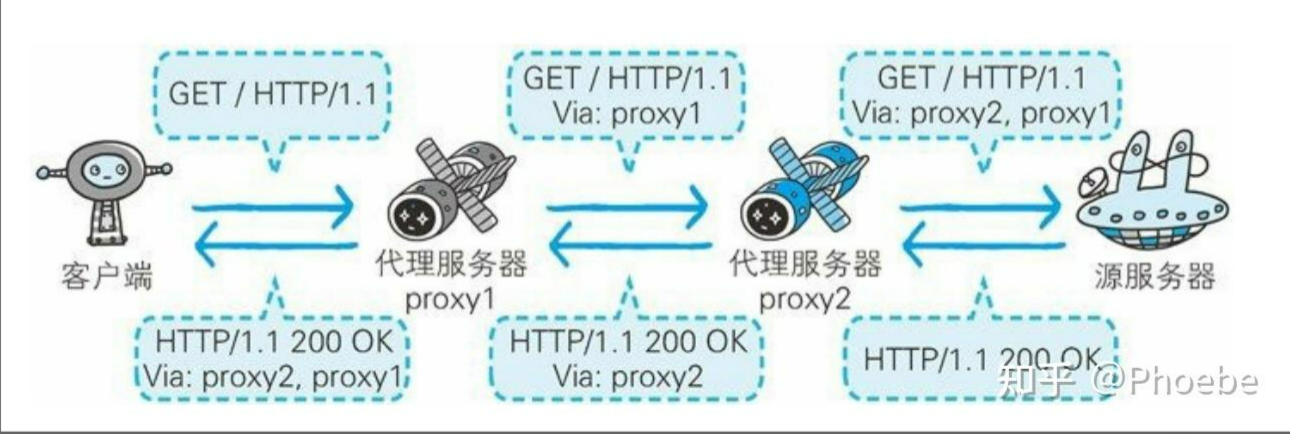
**更快一点的就是流水线方式，一次性发多个请求，一块发，然后一个一个的往回返。（不用等待响应也可以发送下一个请求）**

#### Web caches

**缓存是指代理服务器或客户端本地磁盘内保存的资源副本**。利用缓存可减少对源服务器的访问，因此也就节省了通信流量和通信时间。缓存服务器是代理服务器的一种，并归类在缓存代理类型中。换句话说，当代理转发从服务器返回的响应时，代理服务器将会保存一份资源的副本。

**访问服务器的方式**：

1. 直接访问 web 服务器，直接从源服务器获取对象。
2. 设置 web 代理服务器，通过代理获取对象。



如果代理服务器没有对象，他就向原始的服务器请求对象，同时在它的代理服务器的文件系统中将这个对象缓存下来。

下次如果有其他的用户再访问同样的对象，代理服务器就直接从本地文件系统当中把它读取出来，形成 response 报文反转给用户。我们把这个行为叫做对象在代理中被命中了。

**为什么要使用 Web 缓存？**

* 对用户来说，降低客户端的请求响应时间
* 对 ISP （互联网服务提供商 Internet Service Provider，比如移动。）来说，可以大大减少一个机构内部网络与 Internent 接入链路上的流量
* 对 ICP （网络内容服务商，全称 Internet Content Provider，比如谷歌）来说，互联网大量采用了缓存：可以使较弱的 ICP 也能够有效提供内容

如果原始服务器的对象有变化，代理服务器返回给用户的，就是错误的对象。所以，HTTP 协议也有了相应的处理措施。

**HTTP 1.0**

为了减轻服务器的压力，在 HTTP/1.0 中提供了 Cache 机制。

Expires：强制缓存。缓存服务器上的过期时间。用来指定资源到期的时间

问题：受限于本地时间。如果用户改了本地时间，就没用了

Last-Modified：协商缓存。

资源在服务器上的最后修改时间。

**问题：一个资源被修改了，但其实际内容根本没发生改变，也会返回整个实体给客户端（即使客户端缓存里有个一模一样的资源）。**

**客户端会为资源标记上该信息**，下次再次请求时，会把该信息附带在请求报文中一并带给服务器去做检查，若传递的时间值与服务器上该资源最终修改时间是一致的，则说明该资源没有被修改过，直接返回304状态码，内容为空，这样就节省了传输数据量 。如果两个时间不一致，则服务器会返回该资源并返回200状态码，和第一次请求时类似。这样保证不向客户端重复发出资源，也保证当服务器有变化时，客户端能够得到最新的资源。一个304响应比一个静态资源通常小得多，这样就节省了网络带宽。

**HTTP/1.1**

随着技术的发展，很快 HTTP/1.0 也不能满足需求了。

Cache-Control： 强制缓存

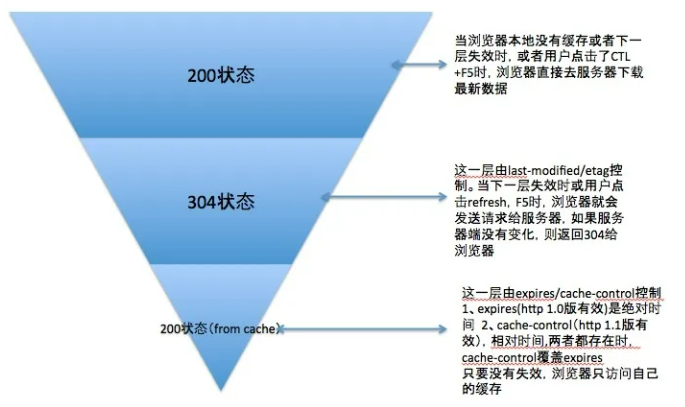
Expires http1.1 新增了 Cache-Control 来定义缓存过期时间 max-age=3600。

**注意：若报文中同时出现了 Expires 和 Cache-Control，则以 Cache-Control 为准。**

ETag：协商缓存：根据内容计算出的唯一标识符

为了解决上述 Last-Modified 可能存在的不准确的问题，Http1.1 还推出了 ETag 实体首部字段。

**强制缓存 与 协商缓存**

* 浏览器在加载资源时，根据请求头的 expires 和 cache-control 判断是否命中强制缓存，是则直接从缓存读取资源，不会发请求到服务器。
* 如果没有命中强制缓存，浏览器一定会发送一个请求到服务器，通过 last-modified 和 etag 验证资源是否命中协商缓存，如果命中，服务器会将这个请求返回，但是不会返回这个资源的数据，依然是从缓存中读取资源
* 如果前面两者都没有命中，直接从服务器加载资源

### 2 DNS

#### 简介

Domain Name System 域名系统。也可以叫做域名解析协议。在我们在浏览器访问网页的时候，通常度是用我们所熟悉的一连串有意义的英文字符标识，比如www.baidu.com、www.sohu.com等。

但是我们学了前面的知识，计算机并不是通过这些字符串去找到对应的计算机，而是通过32位的二进制，也就是我们的IP地址来找。所以就有了DNS协议。他的作用就是将域名解析成对应的IP地址。

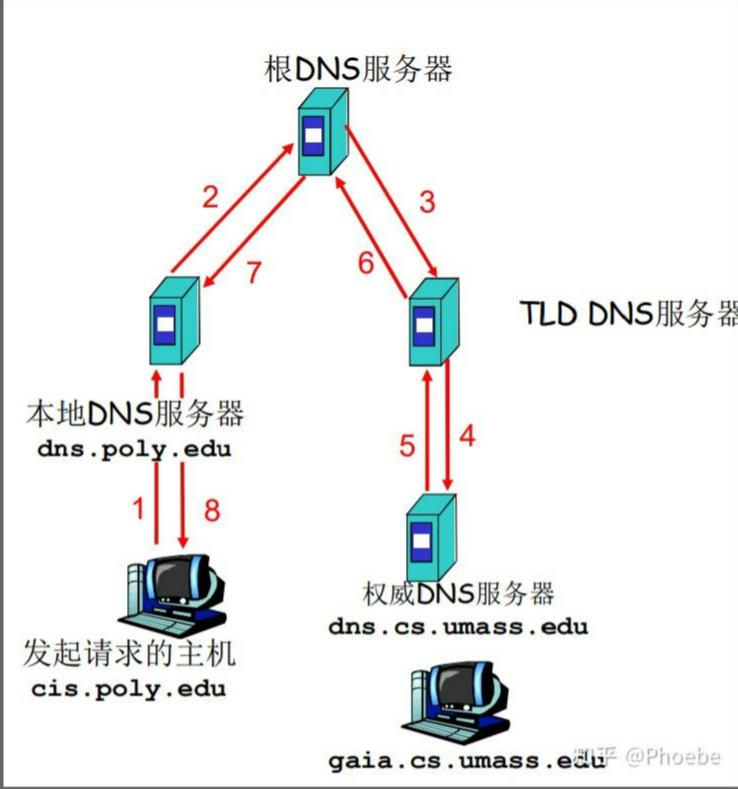
因为让我们人去记那些IP地址，很难记得住，所以就想办法让IP地址转变为了现在的域名，在进行访问的时候，只需要将域名解析为对应的IP地址就行了，这个域名也很有讲究，其中分为好多层域名，是独一无二的。这里不细讲这个，只要我们知道，域名通过DNS能找到对应的IP地址就行了。

#### 用途

将域名转换为对应的IP地址

#### 解析过程

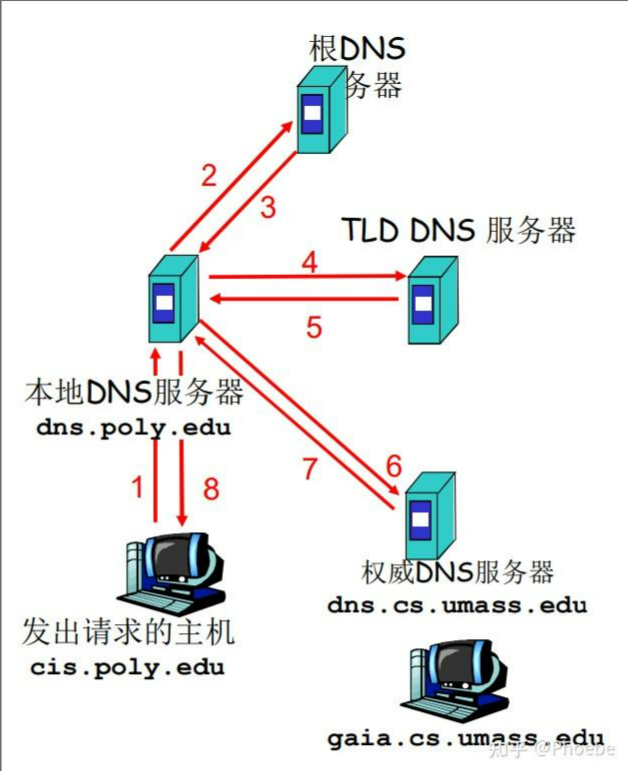
**递归查询**



从图上我们可以看出，名字解析负担都放在当前联络的名字服务器上。本地不知道问根，根不知道问上层，查询到之后再依次返回。

在这个过程当中，需要维护整个查询的过程。而且根服务器，需要做很多工作，负担太重。

**迭代查询**



主机 http://cis.poly.edu 想知道主机 http://gaia.cs.umass.edu 的 IP 地址，它向本地 DNS 服务器发出询问。

* 根(及各级域名)服务器 返回的不是查询结果，而是下一个 DNS 的地址。
* 最后由权威名字服务器给出解析结果。
* 当前联络的服务器给出可以联系的服务器的名字，相当于每个服务器会告诉你“我不知道这个名字，但你可以向这个服务器请求“

### 3 CDN

#### 简介

CDN（Content Delivery Network，内容分发网络）是构建在现有互联网基础之上的一层智能虚拟网络，通过在网络各处部署节点服务器，实现将源站内容分发至所有CDN节点，使用户可以就近获得所需的内容。

#### 解决的问题

**CDN服务缩短了用户查看内容的访问延迟，提高了用户访问网站的响应速度与网站的可用性，解决了网络带宽小、用户访问量大、网点分布不均等问题。**

**CDN的优势:**

* **提高了用户访问网站的响应速度**
* **为了实现跨运营商、跨地域的全网覆盖**

互联不互通、区域ISP地域局限、出口带宽受限制等种种因素都造成了网站的区域性无法访问。CDN加速可以覆盖全球的线路，通过和运营商合作，部署IDC资源，在全国骨干节点商，合理部署CDN边缘分发存储节点，充分利用带宽资源，平衡源站流量。

* **为了保障你的网站安全**

CDN的负载均衡和分布式存储技术，可以加强网站的可靠性，相当无无形中给你的网站添加了一把保护伞，应对绝大部分的互联网攻击事件。防攻击系统也能避免网站遭到恶意攻击。

* **为了异地备援**

当某个服务器发生意外故障时，系统将会调用其他临近的健康服务器节点进行服务，进而提供接近100%的可靠性，这就让你的网站可以做到永不宕机。

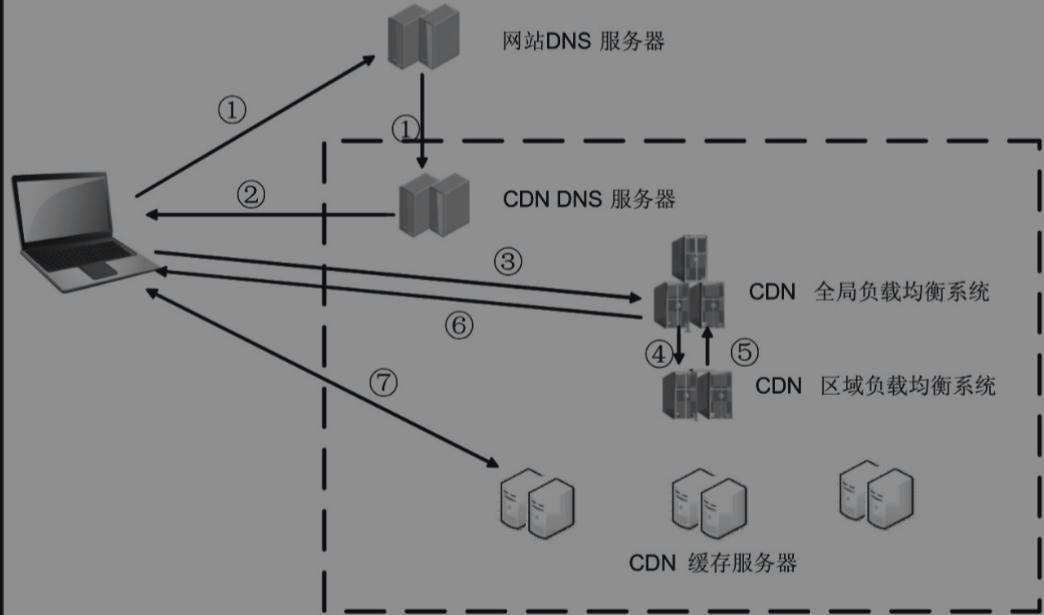
* **为了节约成本**

投入使用CDN加速可以实现网站的全国铺设，你根据不用考虑购买服务器与后续的托管运维，服务器之间镜像同步，也不用为了管理维护技术人员而烦恼，节省了人力、精力和财力。

* **为了让你更专注业务本身**

CDN加速厂商一般都会提供一站式服务，业务不仅限于CDN，还有配套的云存储、大数据服务、视频云服务等，而且一般会提供7x24运维监控支持，保证网络随时畅通，你可以放心使用。并且将更多的精力投入到发展自身的核心业务之上。

#### 基本工作流程



1. 当用户点击网站页面上的内容URL，经过本地DNS系统解析，DNS系统会最终将域名的解析权交给CNAME指向的CDN专用DNS服务器。

2. CDN的DNS服务器将CDN的全局负载均衡设备IP地址返回用户。

3. 用户向CDN的全局负载均衡设备发起内容URL访问请求。

4. CDN全局负载均衡设备根据用户IP地址，以及用户请求的内容URL，选择一台用户所属区域的区域负载均衡设备，告诉用户向这台设备发起请求。

5. 区域负载均衡设备会为用户选择一台合适的缓存服务器提供服务，选择的依据包括：根据用户IP地址，判断哪一台服务器距用户最近；根据用户所请求的URL中携带的内容名称，判断哪一台服务器上有用户所需内容；查询各个服务器当前的负载情况，判断哪一台服务器尚有服务能力。基于以上这些条件的综合分析之后，区域负载均衡设备会向全局负载均衡设备返回一台缓存服务器的IP地址。

6. 全局负载均衡设备把服务器的IP地址返回给用户。

7. 用户向缓存服务器发起请求，缓存服务器响应用户请求，将用户所需内容传送到用户终端。如果这台缓存服务器上并没有用户想要的内容，而区域均衡设备依然将它分配给了用户，那么这台服务器就要向它的上一级缓存服务器请求内容，直至追溯到网站的源服务器将内容拉到本地。

### 4 P2P与CS

#### 纯P2P架构的特点

* 没有服务器
* 端系统之间直接通信
* 端系统经常改变IP **间歇性连接**

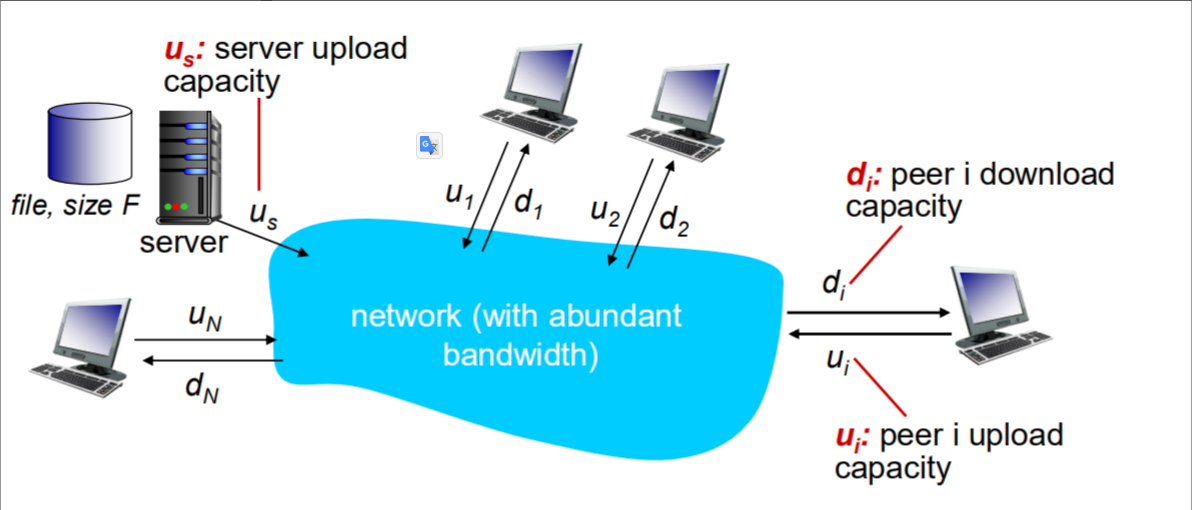
#### 文件分发计算

在P2P文件分发中，每个对等方能够重新分发它所有的该文件的任何部分，从而在分发过程中协助该服务器。

**P2P体系结构的扩展性**

**为什么文件分发时P2P会比客户-服务器更加有效？**

用F表示该发文件的长度，N表示要获得该文件副本的对等方数量。分发时间是所有N个对等方得到该文件的副本所需要的时间。



**（1）客户-服务器体系结构**

服务器必须向N个对等方传输该文件的一个副本，因此该服务器必须传输比特。而服务器上载速率是，则分发该文件的时间必定至少为。

客户机下载的速度为，具有最小下载速率的对等方的下载速率为，具有最小下载速率的对等方不可能在少于时间内获得该文件的所有F比特，则有**最小分发时间**至少为。

所以，最小分发时间。

**（2）P2P体系结构**

每个对等方能够帮助服务器分发该文件。当一个对等方接收到某些文件数据，它能够使用自己的上载能力重新将数据分发给其他对等方。

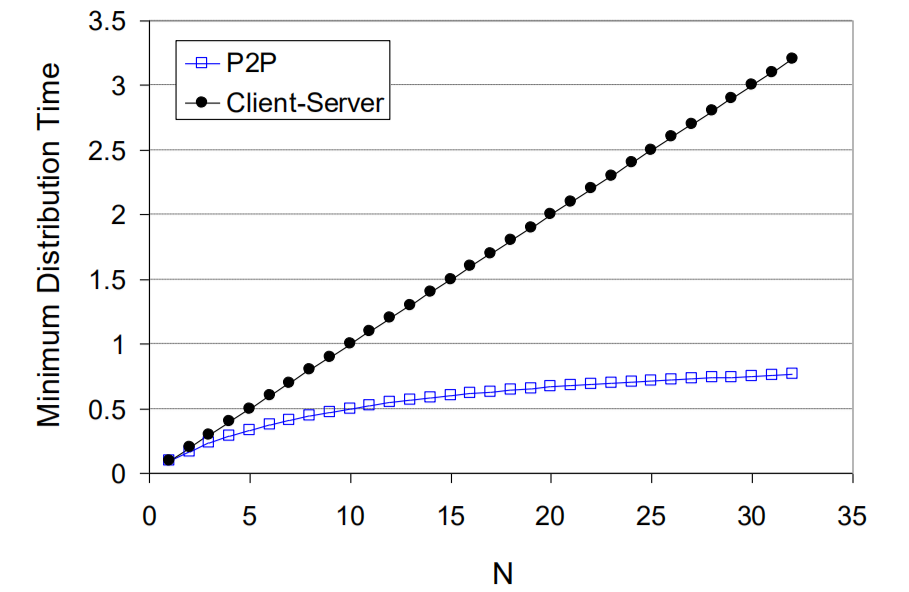
服务器必须经其链路至少发送该文件每个比特一次。**最小分发时间**为。

客户机下载的速度为，则有**最小分发时间**至少为。

观察到**系统整体的总上载能力等于服务器的上载速率加上每个单独的对等方上载速率**，即。系统必须向这N个对等方的每个交付F比特，这不能以快于的速率完成，因此**最小分发时间**至少是。

所以，最小分发时间。（此时我们认为一旦每个对等方接收到一个比特就能够重新分发一个比特。）

对于客户-服务器体系结构，随着对等方数量的增加，分发时间呈线性增长并且没有界。然而，对于P2P体系结构，最小分发时间不仅总是小于客户-服务器体系结构的分发时间，并且对于任意的对等方数量N，总是小于1小时。



因此，具有P2P体系结构的应用程序能够是自扩展的。这种扩展性的直接成因是：**对等方除了是比特的消费者还是它们的重新分发者**。

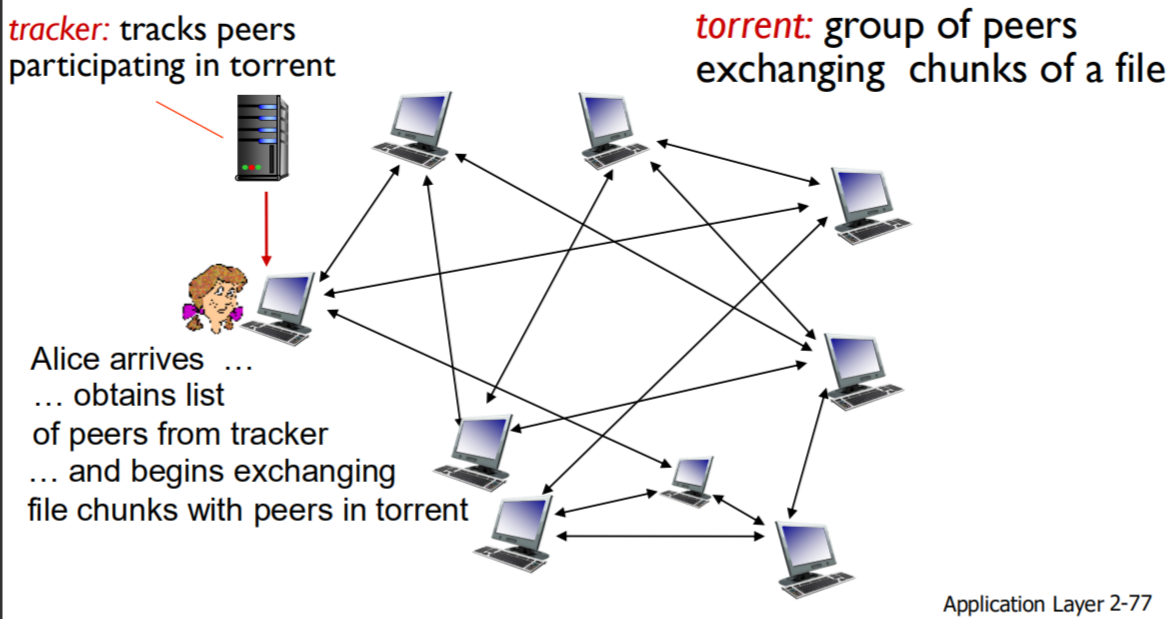
#### BitTorrent

BitTorrent是一直用于文件分发的流行P2P协议。

在BT系统里，**文件是分片的**。

当一个peer 节点开始加入一个 洪流（Torrent） 时，它没有文件块。随着时间的推移，它将累积越来越多的文件块。当它下载文件块时，也为其他 peer 节点上载了多个文件块。peer 节点一但获得了整个文件，它可以(自私地)离开 Torrent，或(大公无私地)留在 Torrent 中并继续向其他 peer节点上载文件块。

每一个洪流具有一个基础设施结点，称为追踪器。当一个对等方加入某洪流时，他向追踪器注册自己，并周期性地通知追踪器它仍在该洪流中。以这种方式，追踪器跟踪正参与在洪流中的对等方。一个给定的洪流可能在任何时刻具有数以百计或数以千计的对等方。



当一个 peer 节点 Alice 加入 Torrent 时，追踪服务器随机选择一些 peer 节点，并将这些peer 节点的 IP 地址发送给 Alice。

Alice 持有这些 peer 节点的列表，试着与该列表上的多个 peer 节点创建并行的 TCP 连接。这里称所有与 Alice 成功地创建 TCP 连接的 peer 节点为“邻近 peer 节点”。

随着时间的推移，其中的一些 peer 节点可能离开，而其他 peer 节点可能试着与 Alice创建 TCP 连接。因此，邻近 peer 节点将随着时间而改变。

在任何时刻，每个 peer 节点都拥有来自某文件块的子集，且不同的 peer 节点具有不同的文件块子集。Alice 周期性地(经 TCP 连接)询问每个邻近 peer 节点它们所具有的块列表。

如果 Alice 有 L 个邻居，那么她将获得 L 个块列表。因此，Alice 将对她当前还没有的块发出请求(仍通过 TCP 连接)。

Alice 使用一种称为**最稀罕优先的策略**，其思路是根据她没有的块从她的邻居中确定最稀罕的块(**最稀罕的块就是在她的邻居中拷贝数量最少的那些块**)，并优先请求那些最稀罕的块。按照此方式，**最稀罕的块更迅速地重新分发，其目标(大致)是均衡每个块在洪流中的拷贝数量。**

如果多个用户向她请求文件块，为了决定她响应哪个请求，BitTorrent 使用了一种**对换算法**。**其基本思想是那些当前能够以最高的速率供给她数据的邻居具有较高的优先权**。Alice对于她的每个邻居都持续地测量她们之间连接的速率，确定以最高速率流入的 4 个邻居。然后，她将数据块发给这 4 个邻居。每过 10 秒，她重新计算该速率并可能修改这 4 个 peer节点。更重要的是，每过 30 秒，她要随机地选择一个另外的邻居并向它发送块。

在 P2P 文件共享中，**搭免费车**(free-riding)是一个常见的问题，这是指 peer 节点从文件共享系统中下载文件而不上传文件。BitTorrent 的对换算法有效地消除了这种搭免费车问题。

## 三 传输层

TCP发送的报文段是交给IP层传送的，但IP层只能提供尽最大努力交付的服务，也就是说，TCP下面的网络所提供的是不可靠的传输。因此，TCP采用了一些适当的措施来提供可靠的传输，使得两个传输层直接的通信变得可靠。

### 1 可靠传输

#### 要求

* 传输信道无差错
* 不管发送方以多快的速度发送数据，接收方总是来得及处理收到的数据（通过累计确认、超时重传、拥塞控制三大模块保证）

这里有两层意思，一是能够正确地传输数据，二是接收方能够及时处理发送方发送的数据。

#### 工作原理

**首先**，采用三次握手来建立TCP连接，四次握手来释放TCP连接，从而保证建立的传输信道是可靠的。

**其次**，TCP采用了**连续ARQ协议**（回退N，Go-back-N；超时自动重传）来保证数据传输的正确性，使用**滑动窗口协议**来保证接方能够及时处理所接收到的数据，进行流量控制。

**最后**，TCP使用慢开始、拥塞避免、快重传和快恢复来进行拥塞控制，避免网络拥塞。

#### TCP三次握手与四次挥手

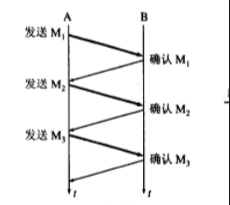


#### ARQ协议

##### 停止等待ARQ协议

* “停止等待”就是每发送完一个分组就停止发送，等待对方的确认。在收到确认后再发送下一个分组。
* 停止等待协议对信道的利用率不高，因为发送方每发送一个消息，都需要等待确认消息回来，只要确认消息没有正确地到达，就一直等；

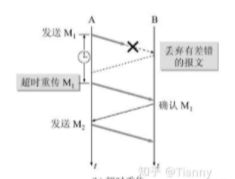
**无差错情况**



如上图（a）所示，表示无差错情况下：

* A 发送分组 M0，发完就暂停发送，等待 B 的确认。
* B 收到了M0就向 A 发送确认。
* A 在收到了对 M0的确认后，就再发送下一个分组 M1。
* 同样，在收到 B 对 M1 的确认后，再发送 M2。

**超时重传**



如上图所示，是传输过程中出现了差错的情况下的超时重传机制。

**出现差错分为两种：**

* M1在传输的过程中丢失了
* B 接收 M1 时检测出了差错，直接丢弃 M1

上面两种情况下，B 都不会发送任何消息给 A。

**为了解决上述场景遇到的问题，可靠传输协议是这样设计的：**

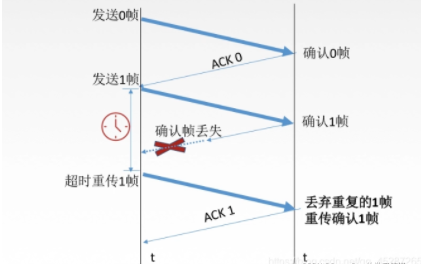
* A只要超过了一段时间仍然没有收到确认，就认为刚才发送的分组丢失了，因而重传前面发送过的分组，这就叫做**超时重传**
* 要实现超时重传，就要在没发送完一个分组就设置一个超时计时器，如果超时计时器到期之前就收到了对方的确认，就撤销已设置的超时计时器。

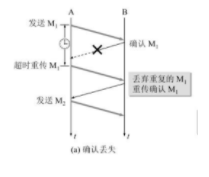
**必须保证以下几点：**

* A发送完一个分组后，必须暂时**保留已发送的分组副本**，只有收到确认后才能清除分组副本
* 分组和确认分组都必须进行**编号**
* 超时计时器设置的重传时间应当比数据在分组传输的往返时间更长一些

**确认丢失**

接收方向发送方发送确认帧1，但是在传输中丢失，接收方在固定时间内没有收到确认帧，会再超时重传1帧。





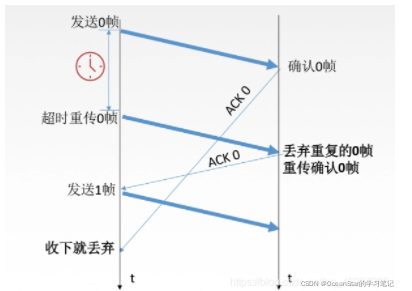
如上图中所示，假设 B 所发送的对 M1 的确认丢失了，那么 A 在设定的超时重传时间内没有收到确认。原因有很多，可能是 A 自己发送的分组出错、丢失，或者是 B 发送的确认丢失了。 因此 A 在超时计时器到期后就要重传 M1。

**现在应注意 B 的动作。假定 B 又收到了重传的分组 M1**。这时应采取两个行动：

* 丢弃这个重复的分组 M1，不向上层交付
* 向 A 发送确认。**不能认为已经发送过确认就不再发送**，**因为 A 之所以重传 M1就表示 A 没有收到对 M1 的确认**

**确认迟到**

* 发送方发送完之后确认帧一直没有传过来，就会超时重传一个0号帧，接收方收到0号帧之后发现刚刚已经收到过0号帧了，就会把重复发送的0号帧丢弃掉，再返回给发送端一个确认帧；
* 同理，如果发送端之前已经收到0的确认帧，那么对于后来又收到的0的确认帧，收到就丢弃。



* 如上图（b）所示，假设传输过程中没有出现差错，但 B 对分组 M1 的确认迟到了。A 会收到重复的确认。
* 对重复的确认的处理很简单：收下后就丢弃。 B 仍然会收到重复的 M1，并且同样要丢弃重复的 M1，并重传确认分组。

##### 连续ARQ协议

停止等待ARQ协议效率比较低，所以引入了ARQ（Automatic Repeat reQuest：自动重传请求）。

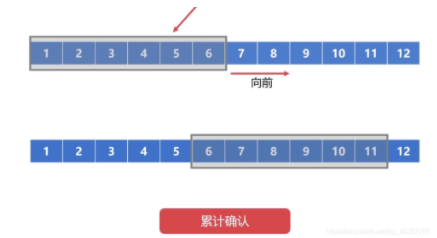
* 连续ARQ协议可以提高信道的利用率。它引入了滑动窗口，通过滑动窗口可以批量发送数据，发送方维持一个发送窗口，凡是位于发送窗口内的分组都可以连续发送出去，而不需要等待对方确认。
* 接收方一般采用累计确认，对按序到达的最后一个分组发送确认，表明到这个分组为止的所有分组都已经正确收到了。



滑动窗口大小为6，可以批量发送6个消息；当收到1和2的确认消息之后，滑动窗口就可以往后移两个为止，将窗口内还没有被发送的数据发送出去。

对于滑动窗口，如果每一个报文都需要确认，那么确认消息的开销会非常大，因此对于滑动窗口并不需要对每个报文都进行确认，而是采用累计确认的方法。

假设现在同时发送了1到6这6个报文，在某一时刻发送方收到第5个报文的确认消息，如果采用累计确认的方法，我们就默认1到5这5个消息都收到了，因此会把窗口往前推动五格。



**累积确认方式**

为了提高信道的利用率，实际上采用了流水线传输的方案。这时就需要使用连续ARQ协议和滑动窗口协议。发送方和接收方各自维持着发送窗口和接受窗口，发送方每收到一个确认，就把发送窗口向前滑动一个分组的位置。接收方一般采用累计确认方式，即接收方不必对收到的分组逐个发送确认，而是可以在收到几个分组后，对按序到达的最后一个分组发送确认，这样就表示：到这个分组位置的所有分组都已经正确收到了。



**看个例子：**

由图可见，描述一个发送窗口的状态需要三个指针P1、P2、P3

* P3−P1=A 的发送窗口(通知窗口)
* P2−P1=已发送但尚未收到确认的字节数
* P3−P2=允许发送但尚未发送的字节数(可用窗口或有效窗口)

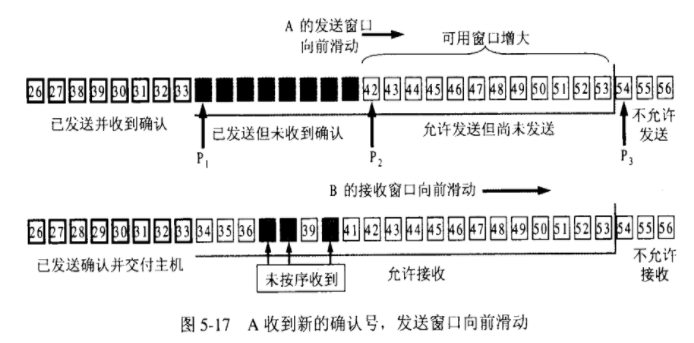
**解析一**：此时B没有收到序号为31的数据，因此只能对按序收到的数据中的最高的序号给出确认，即此时Ｂ的确认号还是30(因为B没有收到序号31的数据，只能发送序号30的确认号)



**解析二**：此时B收到了序号31的数据，那么B会进行如下处理：

* 把31~33的数据交给主机，B删除这些数据缓存
* 接着把接收窗口向前移动3个序号，同时给A发送确认号33

A收到B的确认号后，就可以把发送窗口向前滑动3个序号，但P2指针不变





**解析三**：A在继续发送完序号42~53的数据后，指针P2P2和P3P3重合。发送窗口内的序号都已经用完，但还没有收到确认。由于发送窗口已满，可用窗口已减小到了零，因此必须停止发送。此时A设置了超时计时器，超时计时器到期时就重传这部分数据，重置超时计时器，知道收到B的确认为止。如果A收到的确认号落在发送窗口内，那么A就可以使发送窗口继续向前滑动，并发送新的数据。

**发送缓存与发送窗口、接收缓存与接收窗口**

**发送缓存用来暂时存放(即发送窗口)：**

* 发送应用程序传送给发送方TCP准备发送的数据
* TCP已经发送出但尚未收到确认的数据

**接收缓存用来暂时存放：**

* 按序到达的、但尚未被接收应用程序读取的数据
* 未按序到达的数据

**发送缓存和接收缓存小结**

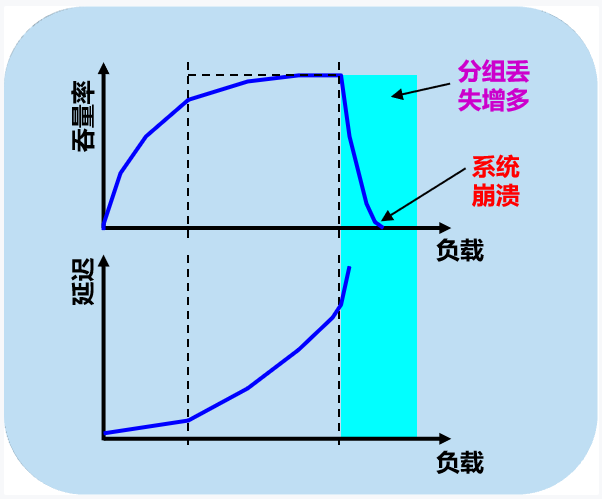
* A的发送窗口根据B的接收窗口设置，但是二者并以总是一样大
* 对于不按序到达的数据应如何处理，TCP标准并无明确规定(一般都是临时保存在接收窗口中，等到字节流完整接收后再按序提交给上层的应用进程)
* TCP要求接收方必须有累积确认的功能，这样子可以减少传输开销

### 2 拥塞控制

#### 一般原理

在某段时间，若对网络中某资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏。这种现象称为拥塞 (congestion)。

最坏结果：系统崩溃。

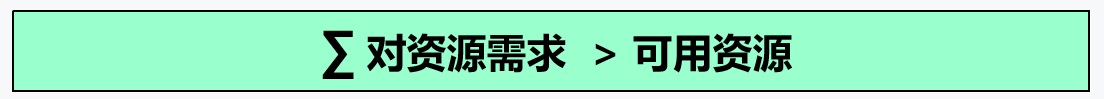


#### 拥塞原因

网络拥塞往往是由许多因素引起的。例如：

* 点缓存的容量太小；
* 链路的容量不足；
* 处理机处理的速率太慢；
* 拥塞本身会进一步加剧拥塞；

**出现拥塞的原因：**



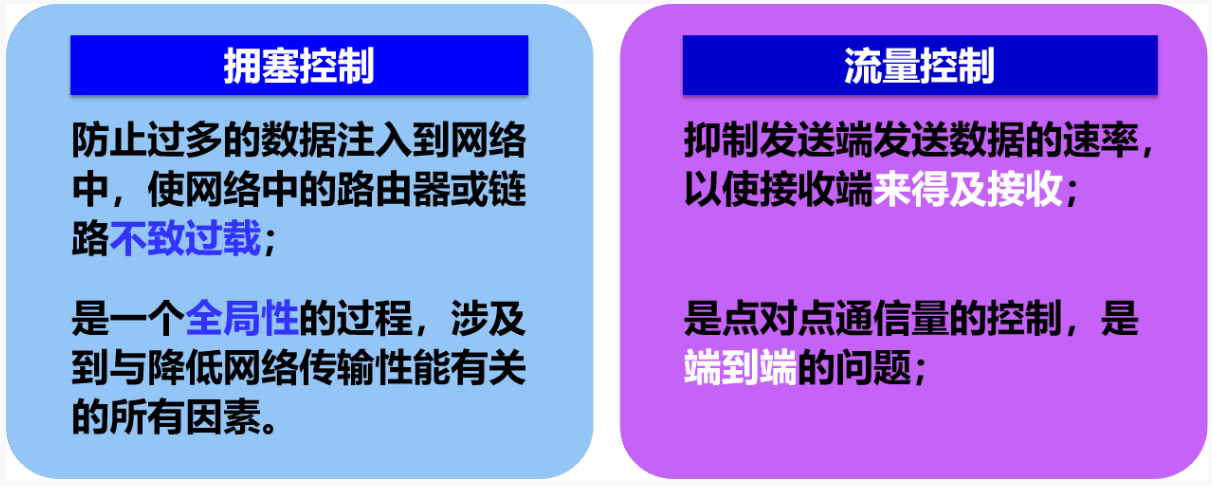
**增加资源能解决拥塞吗？**

不能。这是因为网络拥塞是一个非常复杂的问题。简单地采用上述做法，在许多情况下，不但不能解决拥塞问题，而且还可能使网络的性能更坏。

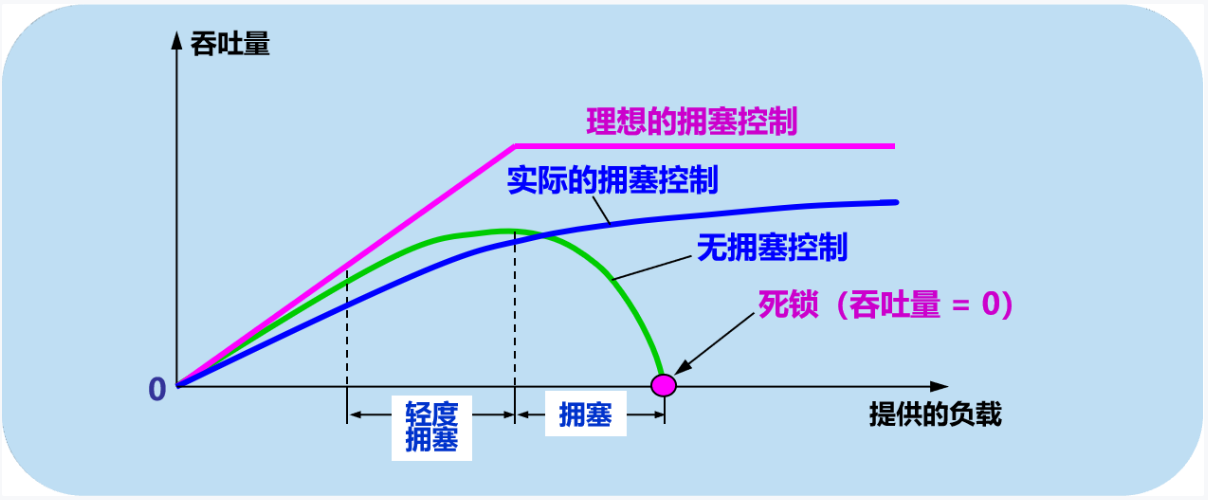
**网络拥塞往往是由许多因素引起的。例如：**

* 增大缓存，但未提高输出链路的容量和处理机的速度，排队等待时间将会大大增加，引起大量超时重传，解决不了网络拥塞；
* 提高处理机处理的速率会将瓶颈转移到其他地方；

**拥塞控制与流量控制的区别**



**拥塞控制所起的作用**



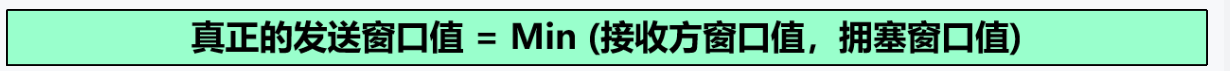
#### 拥塞控制方法

TCP 采用**基于窗口**的方法进行拥塞控制。该方法属于闭环控制方法。

TCP发送方维持一个**拥塞窗口 cwnd** (Congestion Window)

发送端利用**拥塞窗口**根据网络的拥塞情况调整发送的数据量。

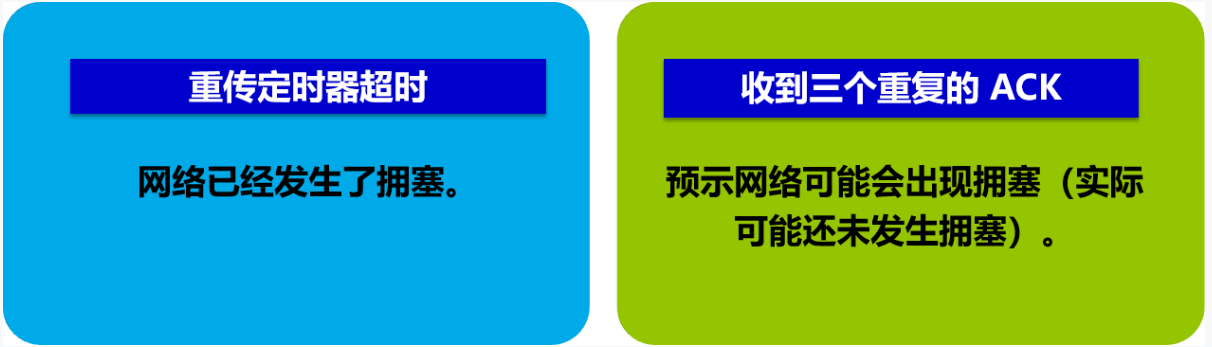
发送窗口大小不仅取决于接收方窗口，还取决于网络的拥塞状况，所以**真正的发送窗口值**为：



**窗口原则**

* 只要网络没有出现拥塞，拥塞窗口就可以再增大一些，以便把更多的分组发送出去，这样就可以提高网络的利用率。
* 但只要网络出现拥塞或有可能出现拥塞，就必须把拥塞窗口减小一些，以减少注入到网络中的分组数，以便缓解网络出现的拥塞。

**拥塞的判断**



##### 慢开始 (slow-start)

**目的**：用来确定网络的负载能力或拥塞程度。

算法的**思路**：由小到大逐渐增大拥塞窗口数值。

主机开发发送数据报时，如果立即将大量的数据注入到网络中，可能会出现网络的拥塞。慢启动算法就是在主机刚开始发送数据报的时候先探测一下网络的状况，如果网络状况良好，发送方每发送一次文段都能正确的接受确认报文段。那么就从小到大的增加拥塞窗口的大小，即增加发送窗口的大小。

例子：开始发送方先设置cwnd（拥塞窗口）=1,发送第一个报文段M1，接收方接收到M1后，发送方接收到接收方的确认后，把cwnd增加到2，接着发送方发送M2、M3，发送方接收到接收方发送的确认后cwnd增加到4，慢启动算法每经过一个**传输轮次**（认为发送方都成功接收接收方的确认），拥塞窗口cwnd就**加倍**。

##### 拥塞避免 (congestion avoidance)

为了防止cwnd增加过快而导致网络拥塞，所以需要设置一个慢开始门限ssthresh状态变量,它的用法：

* 当cwnd < ssthresh,使用慢启动算法，
* 当cwnd > ssthresh,使用拥塞控制算法，停用慢启动算法。
* 当cwnd = ssthresh，这两个算法都可以。

拥塞避免的**思路**：是让cwnd缓慢的增加而不是加倍的增长，每经历过一次往返时间就使cwnd增加1，而不是加倍，这样使cwnd缓慢的增长，比慢启动要慢的多。

无论是慢启动算法还是拥塞避免算法，只要判断网络出现拥塞，就要把慢启动开始门限(ssthresh)设置为设置为发送窗口的**一半**（>=2），cwnd(拥塞窗口)设置为1，然后在使用慢启动算法，这样做的目的能迅速的减少主机向网络中传输数据，使发生拥塞的路由器能够把队列中堆积的分组处理完毕。拥塞窗口是按照线性的规律增长，比慢启动算法拥塞窗口增长块的多。

**实例：**

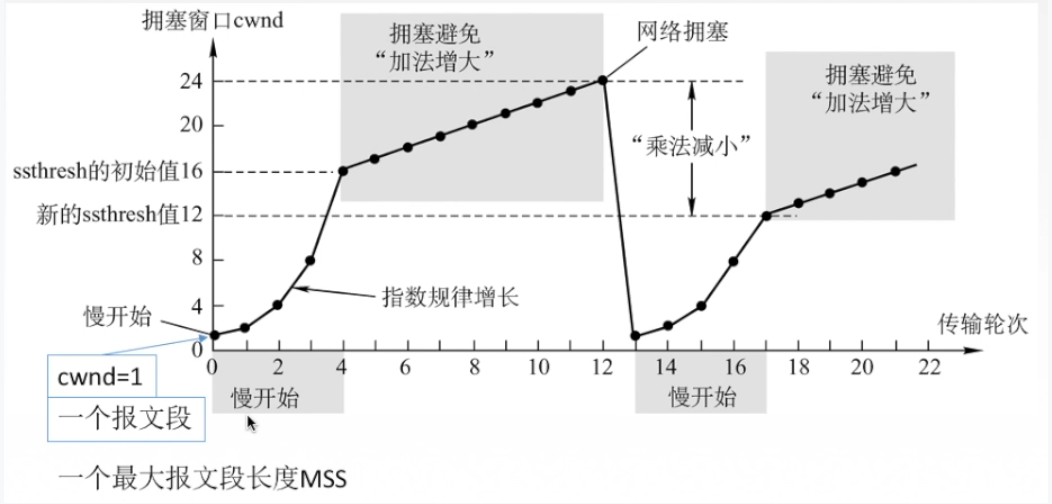
1. TCP连接进行初始化的时候，cwnd=1,ssthresh=16。
2. 在慢启动算法开始时，cwnd的初始值是1，每次发送方收到一个ACK拥塞窗口就增加1，当ssthresh =cwnd时，就启动拥塞控制算法，拥塞窗口按照规律增长，
3. 当cwnd=24时，网络出现超时，发送方收不到确认ACK，此时设置ssthresh=12,(二分之一cwnd),设置cwnd=1,然后开始慢启动算法，当cwnd=ssthresh=12,慢启动算法变为拥塞控制算法，cwnd按照线性的速度进行增长。

##### 快重传 (fast retransmit)

**快重传算法要求首先接收方收到一个失序的报文段后就立刻发出重复确认，而不要等待自己发送数据时才进行捎带确认**。接收方成功的接受了发送方发送来的M1、M2并且分别给发送了ACK，现在接收方没有收到M3，而接收到了M4，显然接收方不能确认M4，因为M4是失序的报文段。如果根据可靠性传输原理接收方什么都不做，但是按照快速重传算法，在收到M4、M5等报文段的时候，不断重复的向发送方发送M2的ACK,**如果接收方一连收到三个重复的ACK,**那么发送方不必等待重传计时器到期，由于发送方尽早重传未被确认的报文段。

##### 快恢复 (fast recovery)

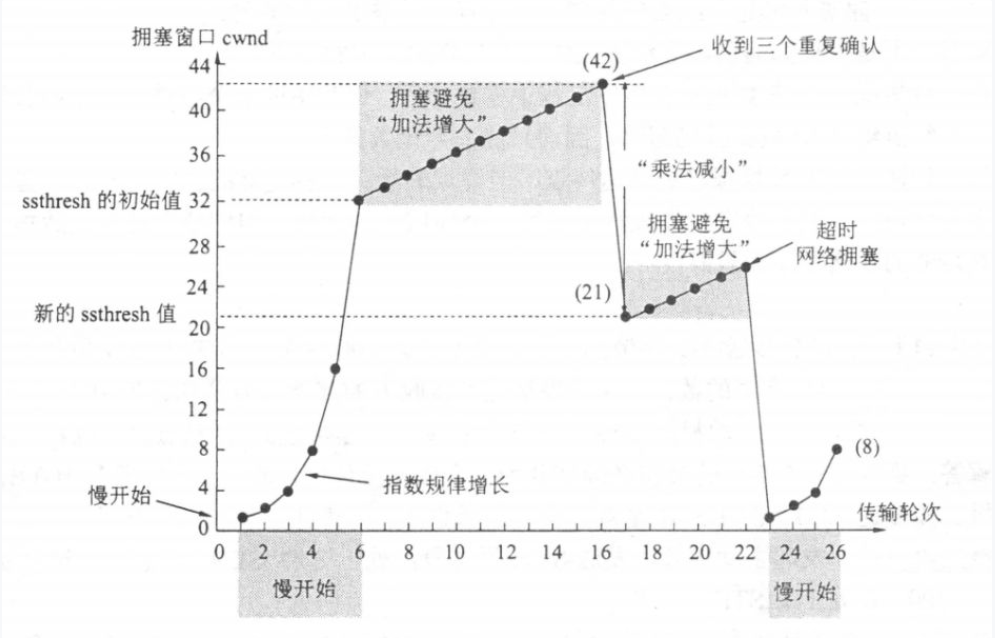
1. 当发送方连续接收到三个确认时，就执行乘法减小算法，把慢启动开始门限（ssthresh）减半，但是接下来并不执行慢开始算法。
2. 此时不执行慢启动算法，而是把cwnd设置为ssthresh的一半， 然后执行拥塞避免算法，使拥塞窗口缓慢增大



##### 例题

TCP的拥塞窗口cwnd大小与传输轮次n的关系如下表所示：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Cwnd | 1 | 2 | 4 | 8 | 16 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| n | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| Cwnd | 40 | 41 | 42 | 21 | 22 | 23 | 24 | 25 | 26 | 1 | 2 | 4 | 8 |
| n | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |



**指明TCP工作在慢开始阶段的时间间隔。**

[1,6]和[23,26]

**指明TCP工作在拥塞避免阶段的时间间隔。**

[6,16]和[17,22]

**在第16轮次和第22轮次之后发送方是通过收到三个重复的确认还是通过超时检测到丢失了报文段？**

* 在第16轮次之后发送方通过接受三个重复的确认，检测到丢失了报文段，因为题目给出，下一个轮次的拥塞窗口减半了。
* 在第22轮次之后发送方是通过超时检测到丢失了报文段，因为题目给出，下一个轮次的拥塞窗口下降到1了。

**在第1轮次，第18轮次和第24轮次发送时，门限ssthresh分别被设置为多大？**

* 在第 1 轮次发送时，门限 ssthresh 被设置为 32，因为从第 6 轮次起就进入了拥塞避免状态，拥塞窗口每个轮次加 1。
* 在第 18 轮次发送时，门限 ssthresh 被设置为发生拥塞时拥塞窗口 42 的一半，即 21。
* 在第 24 轮次发送时，门限 ssthresh 被设置为发生拥塞时拥塞窗口 26 的一半，即 13。

**在第几轮次发送出第70个报文段？**

* 第 1 轮次发送报文段 1。（cwnd = 1）
* 第 2 轮次发送报文段 2， 3。（cwnd = 2）
* 第 3 轮次发送报文段 4 ~ 7。（cwnd = 4）
* 第 4 轮次发送报文段 8 ~ 15。（cwnd = 8）
* 第 5 轮次发送报文段 16 ~ 31。（cwnd = 16）
* 第 6 轮次发送报文段 32 ~ 63。（cwnd = 32）
* 第 7 轮次发送报文段 64 ~ 96。（cwnd = 33）
* 因此第 70 报文段在第 7 轮次发送出。

**假定在第26轮次之后收到了三个重复的确认，因而检测出了报文段的丢失，那么拥塞窗口cwnd和门限ssthresh应设置为多大？**

检测出了报文段的丢失时拥塞窗口 cwnd 是 8，因此拥塞窗口 cwnd 的数值应当减半，等于 4，而门限 ssthresh 应设置为检测出报文段丢失时的拥塞窗口 8 的一半，即 4。

## 四 网络层

### 数据层面

**转发（依靠路由表）——本地。传统：地址+转发表；SDN：多个字段+流表。**

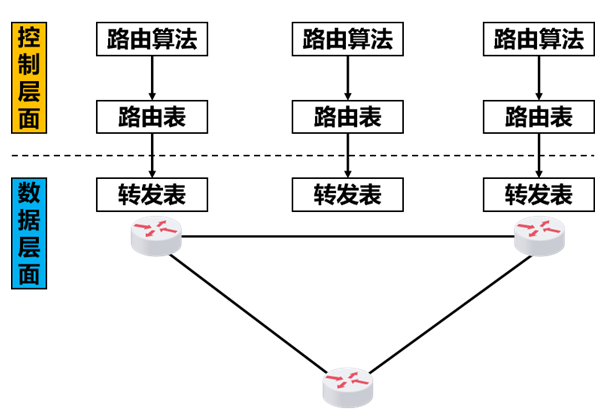
#### SDN

路由器的功能：

* 为主机间的通信提供转发服务
* 路由选择

路由器之间传送的信息：

* 主机间的分组
* 路由信息

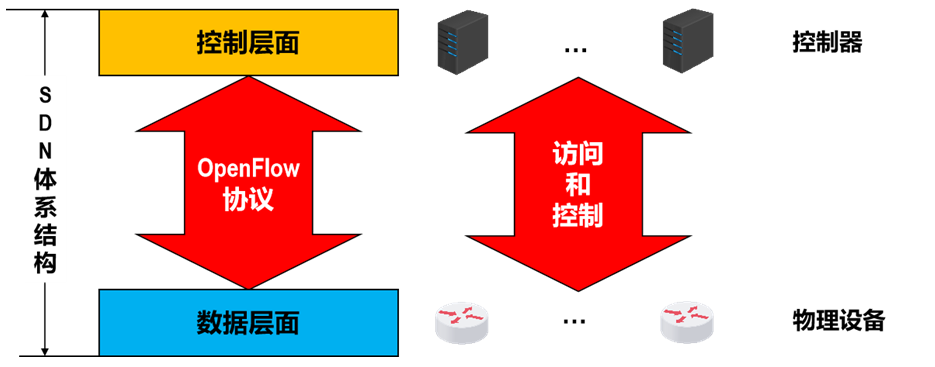


* 在SDN体系结构中，路由器中的路由软件都不存在了。因此，**路由器之间不再交换路由信息**。
* 在控制层面中，有一个在逻辑上集中的远程控制器。
* 逻辑上集中的远程控制器在物理上可由不同地点的多个服务器组成。
* 远程控制器掌握各主机和整个网络的状态。
* 远程控制器能够为每一个分组计算出最佳的路由。
* 远程控制器为每一个路由器生成其正确的转发表。
* SDN这种新型网络体系结构的核心思想：把网络的控制层面和数据层面分离，而让控制层面利用软件来控制数据层面中的许多设备。



#### OpenFlow

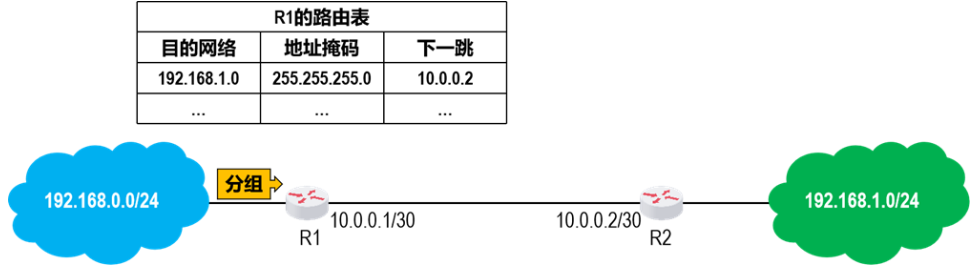
* OpenFlow协议是一个得到高度认可的标准，在讨论SDN时往往与OpenFlow一起讨论。
* OpenFlow协议可被看成是SDN体系结构中**控制层面与数据层面之间的通信接口**。
* OpenFlow协议使得控制层面的控制器可以对数据层面中的物理设备进行直接访问和控制。
* OpenFlow协议的技术规范由非营利性的产业联盟开放网络基金会（Open Networking Foundation，ONF）负责制定。
  + ONF的任务是致力于SDN的发展和标准化。
  + SDN并未规定必须使用OpenFlow，只不过大部分SDN产品采用了OpenFlow作为其控制层面与数据层面的通信接口。
  + OpenFlow从2009年底发表的1.0版开始，每年都被更新，历经12次更新，到2015年3月发布了1.5.1版，目前较为成熟的是1.3版本。



传统意义上的数据层面的任务：**根据转发表转发分组**。

转发分组分为以下两个步骤：

* 进行“**匹配**”：查找转发表中的**网络前缀**，进行最长前缀匹配。
* 执行“**动作**”：把分组从匹配结果指明的接口转发出去。



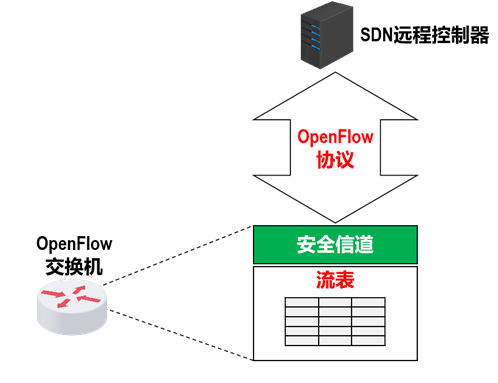
**SDN的广义转发分为以下两个步骤：**

* 进行“**匹配**”：能够对网络体系结构中**各层（数据链路层、网络层、运输层）首部中的字段**进行匹配。
* 执行“**动作**”：不仅**转发分组**，还可以**负载均衡、重写IP首部**（类似NAT路由器中的地址转换）、人为地**阻挡或丢弃**一些分组（类似防火墙一样）。

在SDN的广义转发中，完成“匹配+动作”的设备并不局限在**网络层**工作，因此不再称为路由器，而称为**“OpenFlow交换机”**或“**分组交换机**”，或更简单地称为“**交换机**”。

相应的，在**SDN**中取代传统路由器中转发表的是“**流表（Flow Table）**”。

* 一个流就是穿过网络的一种**分组序列**，而在此序列中的每个分组都**共享分组首部某些字段的值**。例如，某个流可以是具有相同源IP地址和目的IP地址的一连串分组。
* OpenFlow交换机中的**流表是由SDN远程控制器来管理的**。SDN远程控制器通过一个安全信道，使用OpenFlow协议来管理OpenFlow交换机中的流表。
* 网络设备可以由不同厂商来生产，可以使用在不同类型的网络中。
* 从SDN远程控制器看到的，是统一的逻辑交换功能。

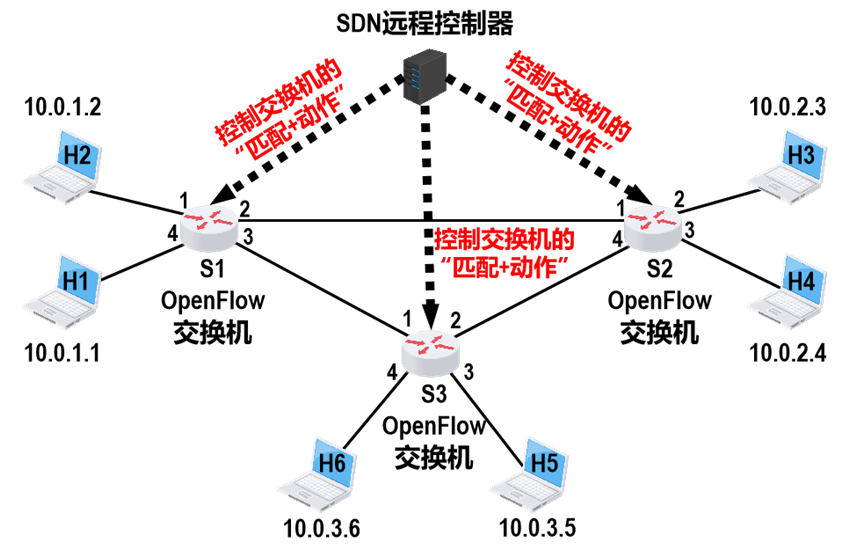


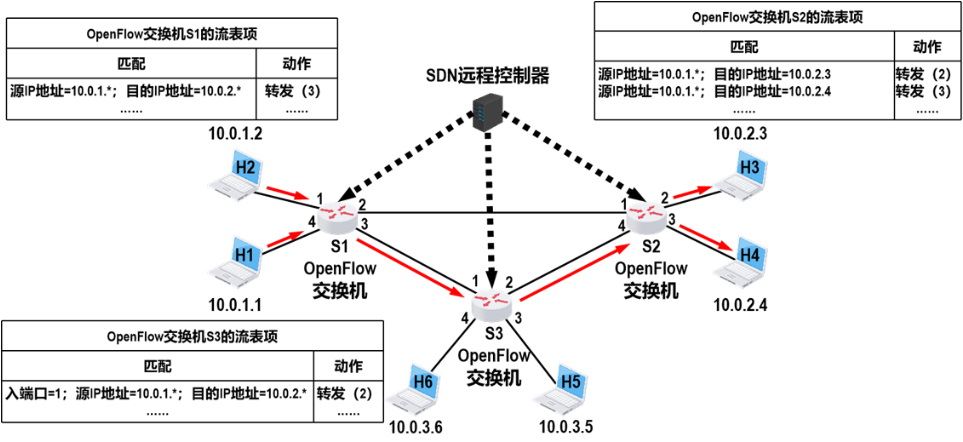
* 每个OpenFlow交换机必须有**一个或多个流表**。
* 每一个流表可以包含**多个流表项**。
* 每个流表项包含三个字段：**首部字段值**（或称匹配字段）、**计数器、动作**。



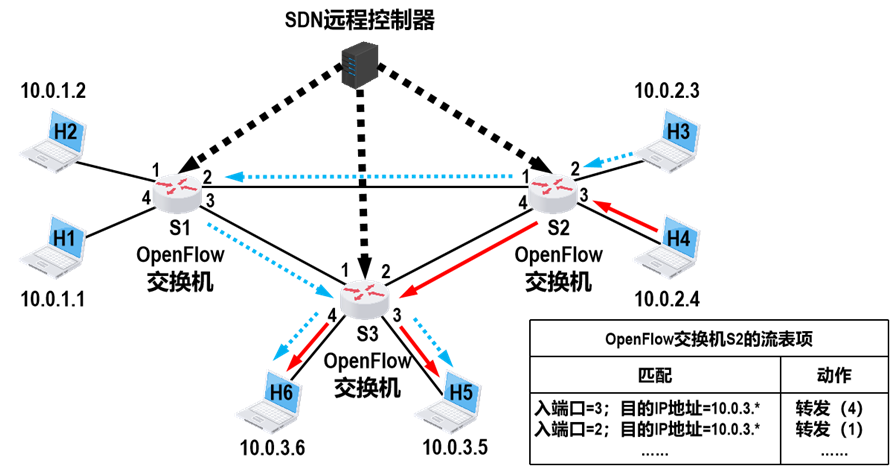
* **首部字段值**字段包含有一组字段，用来使入分组（Incoming Packet）的对应首部与之匹配，因此又称为**匹配字段**。匹配不上的分组就被丢弃，或被发送到SDN远程控制器做更多的处理。
* 在OpenFlow交换机中，既可以处理**数据链路层的帧**，也可以处理**网际层的IP数据报**，还可以处理**运输层的TCP或UDP报文**。
* **计数器**字段是一组计数器：
  + 记录已经与该流表项匹配的分组数量的计数器；
  + 记录该流表项上次更新到现在经历时间的计数器。
* **动作**字段是一组动作，当分组匹配某个流表项时，执行该流表项中动作字段指明的以下某个或多个动作：
  + 把分组转发到指明的端口
  + 丢弃分组
  + 把分组进行复制后再从多个端口转发出去
  + 重写分组的首部字段（包括数据链路层、网际层以及运输层的首部）。

**【举例1】：简单转发**

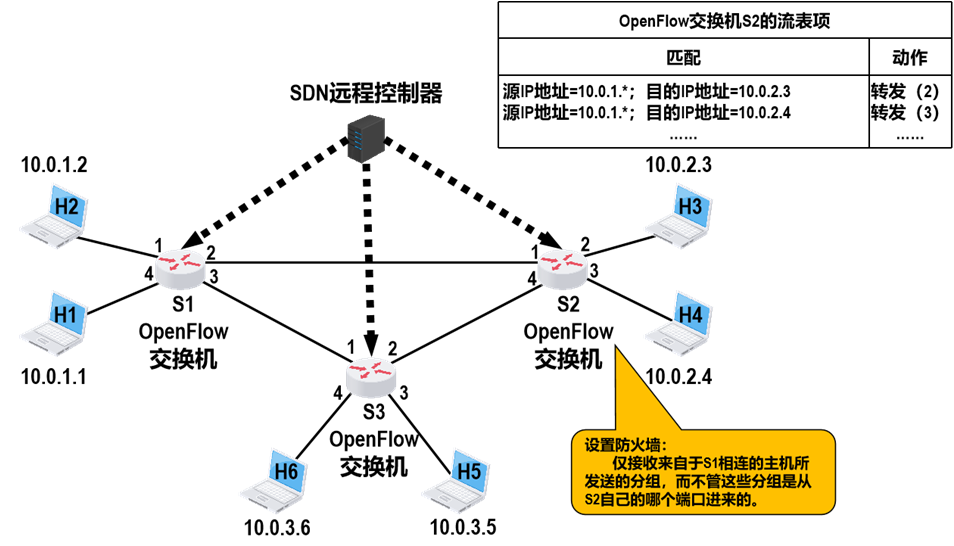




**【举例2】：负载均衡**

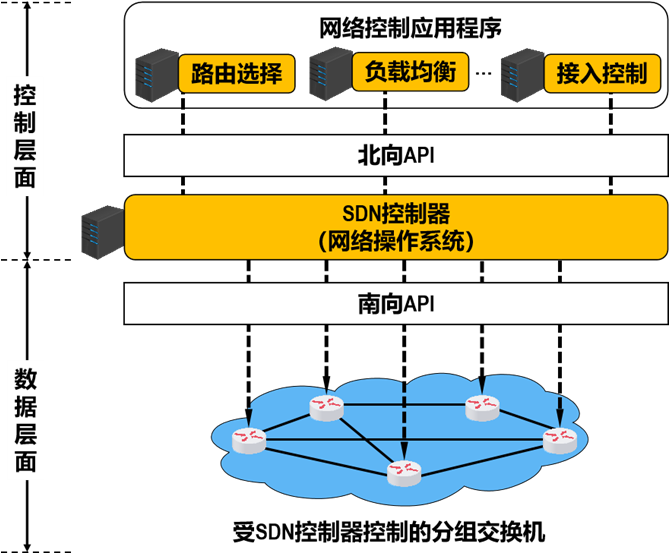


**【举例3】：防火墙**

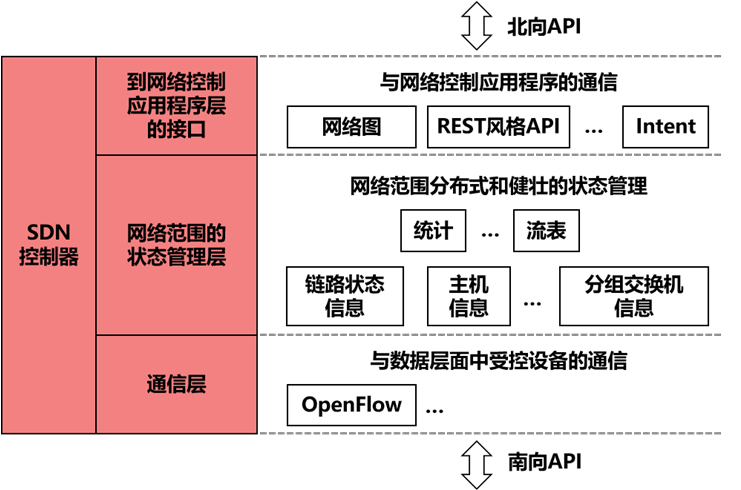


**SDN体系结构及其四个关键特征：**

* 基于流的转发
* 数据层面与控制层面分离
* 位于数据层面分组交换机之外的网络控制功能
* 可编程的网络



**SDN控制器 ：**



### 控制层面

**路由（生成路由表）——全局。传统：路由器中实现；SDN：远程的服务器实现。**

路由器是网络间的连接设备，它重要工作之一是**路径选择**。这个功能是路由器智能的核心，它是由管理员的配置和一系列的路由算法实现的。**路由算法**可分为**distance vector（DV）algorithm**和**link\_state（LS）algorithms**两种。

**工作原理的不同**

**LS算法中，网络拓扑和所有的链路费用都是已知的**，也就是说可用来做LS算法的输入。在实践中，这是通过让每个节点向网络中所有其他路由器广播状态分组来完成，其中每个链路分组包含它所连接的链路的特征和费用，这经常用链路广播算法来完成。节点广播的结果是所有节点具有了该网络的同一个以及完整的视图。于是每个节点都可像其他节点一样，运行LS算法并计算相同的最低费用路径集。在此算法中，每个节点经广播与所有其他节点交谈。

**DV算法中**，**每个节点仅与他的直接邻居交谈**，但它为他的邻居提供了从其自己到网络中所有其他节点的最低费用。DV算法要求每个路由器都要向其他邻接的路由器发布一个距离向量，距离向量是该路由器到其他已知的各网络的相对距离。通告内容基本上是告诉其邻接的路由器：你可以通过我到达网络X，距离是Y。从本地路由器到X的有效距离等于该路由器从各邻接路由器接收到的最短距离，再加上他与该邻接路由器之间的链路距离。

**算法结构不同**

显然**LS算法**是一种**全局信息**的算法，

而**DV算法**是一种迭代的，异步的和分布式的算法。说它是**分布式**的，是因为每个节点都要从一个或多个直接相连邻居接受某些信息，执行计算，然后将计算结果发回给邻居。说它是**迭代的**，是因为此过程一直要持续到邻居之间没有更多要交换的信息为止。说该算法是异步的，是因为他不要求所有节点相互之间锁步，即步伐一致的工作。

**时间复杂性**

**考虑LS算法**，即给定n个节点，考虑最坏情况下需要经多少次计算才能找到从源节点到所有目的节点的最低费用呢。

1）在第一次迭代时，我们需要搜索所有的n个节点以确定出节点w，w不在n-1中且具有最低费用。

2）第二次我们需要检查所有的n-1个节点以确定最低费用。

3）第三次，需要检查n-2个节点，依此类推。

因此，我们在所有迭代中需检索n+(n-1)+(n-2)+…+1=n(n+1)/2个节点。所以我们可以得出LS算法在最差情况下的时间复杂度为n的平方阶。

**考虑DV算法**，它存在对好消息的反应迅速，但对坏消息却反应迟钝。特别是对坏消息存在计数无穷大问题。当链路的权值变化很大时，它的时间复杂性也就很难确定了。

**报文复杂性**

**LS算法**要求每个节点知道网络中每条链路的费用。这就要发送O(|N||E|)个报文。

**DV算法**要求在每次迭代时，在两个直接相连的邻居之间交换报文。在链路费用改变时，DV算法仅当在新费用导致与该链路相连节点的最低费用路径发生改变时，才传播已改变的链路费用。

**收敛速度**

收敛是路由算法选择时所遇到的一个重要问题。收敛时间是指从网络的拓扑结构发生变化到网络上所有的相关路由器都得知这一变化，并且相应地做出改变所需要的时间。这一时间越短，网络变化对全网的扰动就越小。收敛时间过长会导致路由循环的出现。

LS在直接相连的路由之间维护正常的邻居关系。这允许路由更快收敛。链路状态路由协议在会话期间通过交换Hello包（也叫链路状态信息）创建对等关系，这种关系加速了路由的收敛。

而DV算法收敛较慢，且在收敛过程中会遇到选路环路。DV算法还会遭到计数无穷大的问题。

**跳数的限制**

DV使用跳数或向量来确定从一个设备到另一个设备的距离。不考虑每跳链路的速率。

LS没有跳数的限制，使用“图形理论”算法或最短路径优先算法。

**路由表的更新**

DV算法根据相邻节点的路由信息更新自己的路由表并发送整个路由表。DV的这种特性随着路由表的增大，需要消耗更多的CPU资源，并消耗了内存。

LS算法，每个节点都有一个全局的拓扑结构，根据此拓扑结构计算路由表。不像DV那样，更新时发送整个路由表。LS只广播更新的或改变的网络拓扑，这使得更新信息更小，节省了带宽和CPU利用率。另外，如果网络不发生变化，更新包只在特定的时间内发出（通常为30min到2h）。

**健壮性**

**如果一台路由器发生故障，行为错乱或受到破坏时情况会怎样呢？**

对于LS算法，路由器能够向其连接的的一条链路广播不正确费用。一个节点作为LS广播的一部分也可损坏或丢弃它收到的任何LS广播分组。但是一个LS节点仅计算自己的转发表：其他节点为自己做类似的计算。这就意味这在LS算法下，路由计算有些是孤立的，提供了一定程度的健壮性。

在DV算法下，一个节点可向任意或所有的目的节点通告其不正确的最低费用路径。DV算法中一个不正确的节点计算值会扩散到整个网络。

**可扩展性**

DV算法可扩展性差。

LS算法可扩展性好，可靠。

**环路**

尽管最基本的DV协议描述和实现起来非常简单，但是会导致网络中的最短通路树在短时间内出现环路。一个基于DV的网络路由可能需要花费几十秒到几分钟才能汇聚到一个无环路的拓扑。如果本地路由器对于本地链路以外的网络属性只能掌握一些二手的，已被解释过的信息，那么在任何一种这样的方案中低速汇聚都是一种根本的局限性。

LS协议，虽然与DV协议相比较，即便是对于最基本的形式，它的描述和实现都是很复杂的，但它有一个优点，当链路状态发生变化时，在变化的信息传播到网络中以后，我们立即就可以得到不存在环路的最短通路树。

#### LS算法

全称：**链路状态算法**

**在链路状态算法中，通过让每个节点向所有其他路由器广播链路状态分组**，每个链路状态分组包含它所连接的链路的特征和费用，**从而网络中每个节点都建立了关于整个网络的拓扑。**

**Dijkstra 算法计算从源节点到网络中所有其他节点的最低费用路径.**

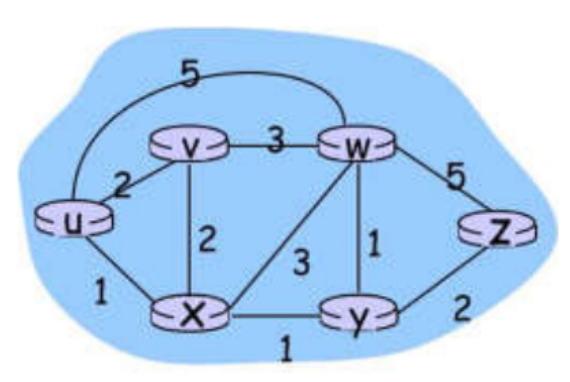
Dijkstra 算法是迭代算法，经算法的第 k 次迭代后，可知道到 k 个目的节点的最低费用路径。

**定义下列记号:**

* D(V)随着算法进行本次迭代，从源节点到目的节点的最低费用路径的费用。
* P(v)从源节点到目的节点 v 沿着当前最低费用路径的前一节点的邻居。
* N`节点子集；如果从源节点到目的节点 v 的最低费用路径已找到，那么 v 在 N`中。

Dijkstra 全局选路算法由一个初始化步骤和循环组成。循环执行的次数与网络中的节点个数相同。在结束时，算法会计算出从源节点 u 到网络中每个其他节点的最短路径。

1. 考虑图中的网络，计算从 u 到所有可能目的地的最低费用路径。

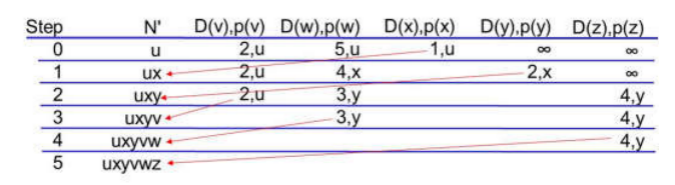


在**初始化阶段**，从 u 到与其直接相连的邻居 v、x、w 的当前已知最低费用路径分别初始化为 2，1 和 5。到 y 与 z 的费用被设为无穷大，因为它们不直接与 u 连接。

在**第一次迭代**时，需要检查那些还未加到集合 N`中的节点，找出在前一次迭代结束时具有最低费用的节点。那个节点是 x 其费用是 1，因此 x 被加到集合 N`中。然后更新所有节点的 D(v)，产生下表中第 2 行(步骤)所示的结果。到 v 的路径费用未变。经过节点 x 到 w 的 路径的费用被确定为 4。因此沿从 u 开始的最短路径到 w 的前一个节点被设为 x。类似地， 到 y 经过 x 的费用被计算为 2，且该表项也被更新。

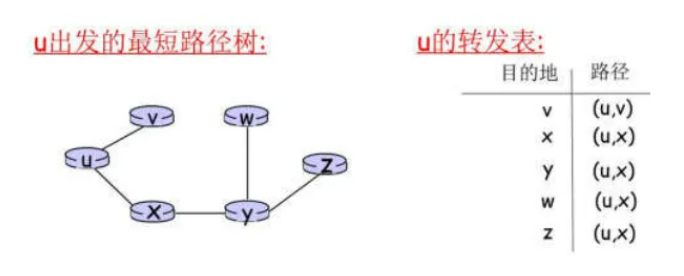
在**第二次迭代**时，节点 v 与 y 被发现具有最低费用路径 2。任意选择将 y 加到集合 N` 中，使得 N’中含有 u、x 和 y。通过更新，产生如表中第 3 行所示的结果。

以此类推…



当 LS 算法结束时，对于每个节点都得到从源节点沿着它的最低费用路径的前继节点， 对于每个前继节点，又有它的前继节点，按照此方式可以构建从源节点到所有目的节点的完 整路径。

根据从 u 出发的最短路径，可以构建一个节点(如节点 u)的转发表。



#### DV算法

全称：**距离向量算法**

LS 算法是一种使用全局信息的算法，而DV距离矢量算法是一种迭代的、异步的和分布式的算法。

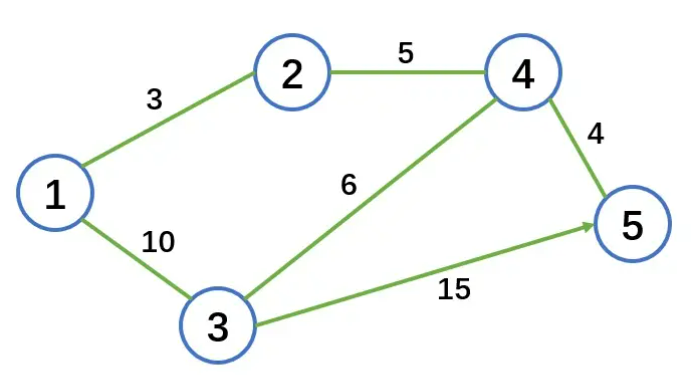
**Bellman-Ford 方程：（弗洛伊德算法）**

设 dx(y)是从节点 x 到节点 y 的最低费用路径的费用，则有 **dx(y) = min {c(x,v) + dv(y) }**

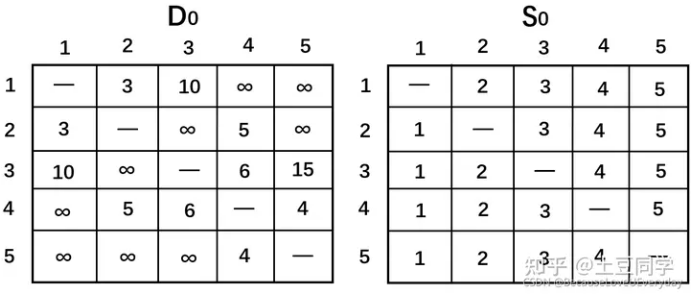


**PS：方程中的 min，是指取遍 x 的所有邻居。**

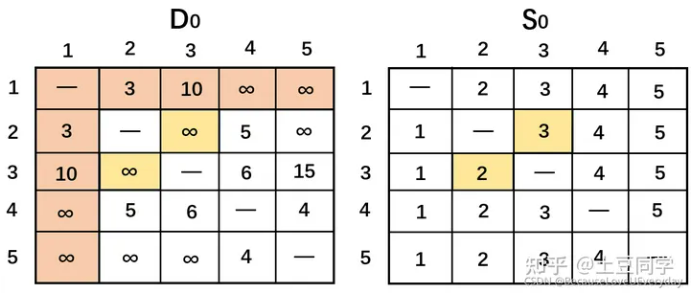
Bellman-Ford 方程含义相当直观，意思是从 x 节点出发到 y 的最低费用路径肯定经过 x 的某个邻居，而且 x 到这个邻居的费用加上这个邻居到达目的节点 y 费用之和在所有路径 中其总费用是最小的。实际上，从 x 到 v 遍历之后，如果取从 v 到 y 的最低费用路径，该路 径费用将是 c(x,v)+ dv(y)。因此必须从遍历某些邻居 v 开始，从 x 到 y 的最低费用是对所有邻 居的 c(x,v)+dv(y)的最小值。



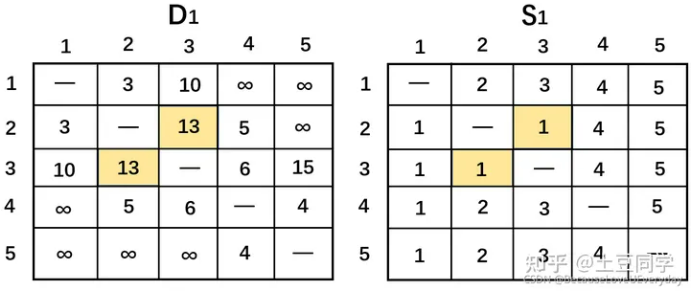
迭代0：矩阵和代表初始的网络。可以看到矩阵除了外(因为弧(3,5)是单向弧)，是对称的。



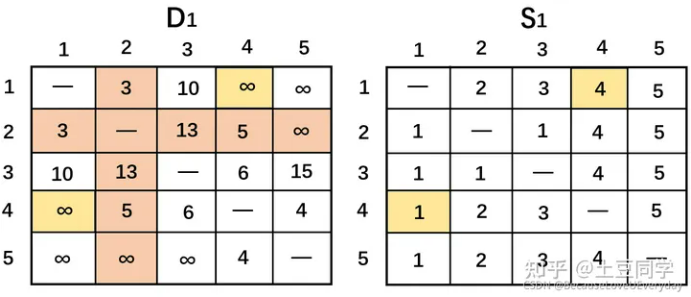
迭代1：令. 矩阵中的黄色阴影表示的第1行和第1列为枢轴行和枢轴列。根据三重操作发现可以改进的元素是和，即

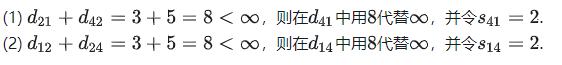


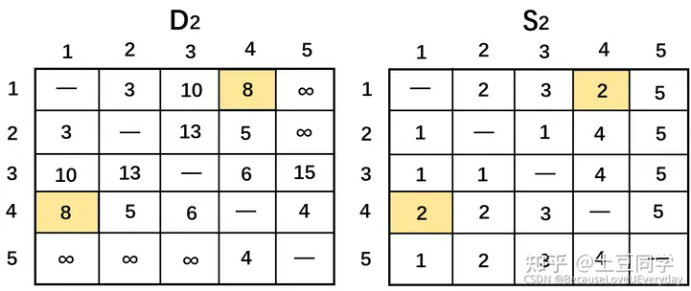




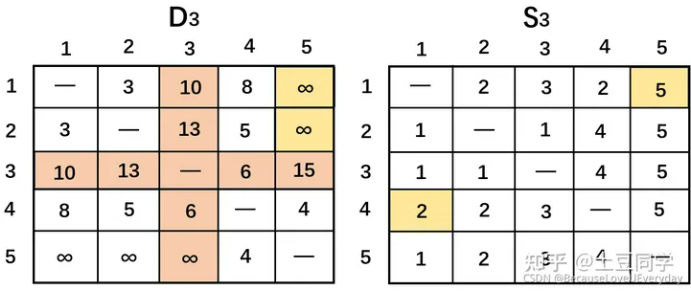
迭代2：令. 矩阵中的黄色阴影表示的第2行和第2列为枢轴行和枢轴列。根据三重操作发现可以改进的元素是和，即

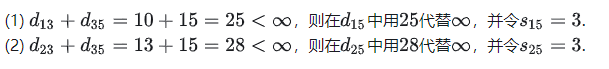


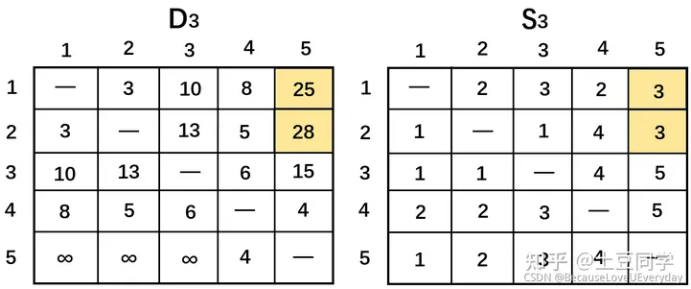




迭代3：令. 矩阵中的黄色阴影表示的第3行和第3列为枢轴行和枢轴列。根据三重操作发现可以改进的元素是和，即

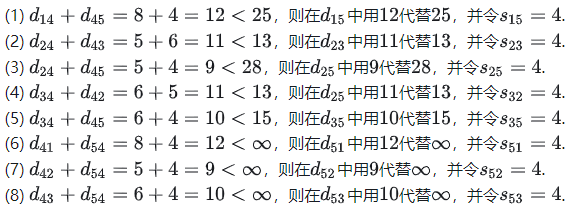


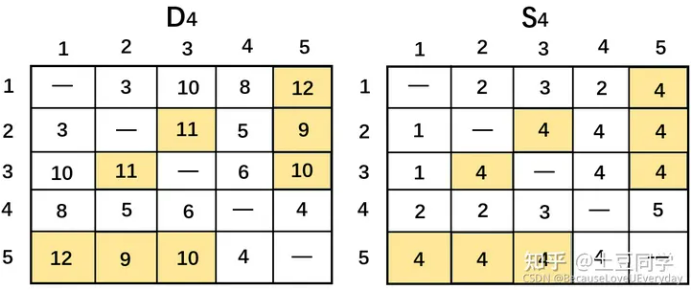




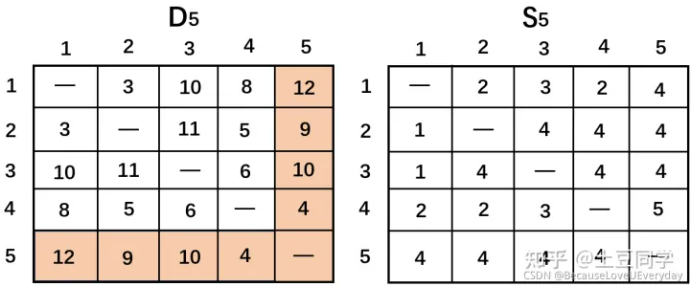
迭代4：令. 矩阵中的黄色阴影表示的第4行和第4列为枢轴行和枢轴列。根据三重操作发现可以改进的元素是、、、、、、、，即



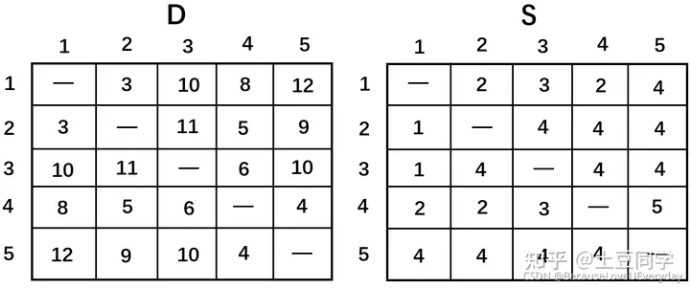




迭代5：令. 矩阵中的黄色阴影表示的第5行和第5列为枢轴行和枢轴列。根据三重操作发现没有可以改进的元素



最后得到的矩阵为：



这两个矩阵包含了网络中任意两个节点最短路径的所有信息。如从矩阵D中可以看出节点1到节点5的最短路径长度为12.从矩阵S中发现，节点1到节点5的中间节点是节点4，即节点1→节点4→节点5，再看节点1→节点4中间是节点2，即节点1需要通过节点2到达节点4，即节点1→节点2→节点4；而节点4可以直接到节点5，中间没有节点，因此可以得到节点1到节点5的最短路径是节点1→节点2→节点4→节点5.

在该 DV 算法中，当节点 x 看到它的直接相连的链路费用变化，或从某个邻居接收到一个距离矢量的更新时，就根据 Bellman-Ford 方程更新其距离矢量表。

## 五 数据链路层

### 1 差错控制

数据在传输过程中，由于信道受到噪声干扰的影响，信号波形传到接收方就可能会发生错误，为了把这些错误减到预期要求，就需要进行差错控制。

差错控制的原理很简单，就是**在被传送的K位信息后附加r位的冗余位**，接收方对收到的信息应用同一算法，将结果与发送方的结果进行比较，若不相等则数据出现了差错。

如果接收方知道有差错发生，但不知道是怎样的差错，然后向发送方请求重传，这种策略称为**检错**；

如果接收方知道有差错，而且知道是怎样的差错，这种策略称为**纠错**。

差错控制有两种，分别是**位出错和帧出错**。

**位出错**：就是传输的比特位出错，如1变为0，0变为1。

**帧出错**：有三种可能分别是**丢失，重复，失序**。实例如下：

* 发送方：发送 帧1、帧2、帧3 三个帧。
* 接收方（丢失）：只收到帧1、帧3，这就是丢失。
* 接收方（重复）：收到帧1、帧2、帧2、帧3，就是重复了。
* 接收方（失序）：收到帧1、帧3、帧2，这就是帧失序。



比如PCA要与PCB进行通信，那么需要一条通信的链路及一系列的通信设备，如果整条通信线路没有差错控制机制，那么PCA的数据帧要发送到PCB之后，才发现数据是错误的；如果在通信链路上的路由器上有差错控制机制，那么从PCA向PCB发送的数据，就会在距PCA最近的路由器处被检测出错有数据帧出错，需要PCA重新发送，正确的数据帧可以先发向PCB端，那么这整条链路就可以节省资源，这就是数据链路层的差错控制方式。

数据链路层的差错控制主要针对在链路上传输的**比特位出错**，主要有两种控制方法，分别是**检错编码**和**纠错编码**，上面提过相关的概念。**检错编码包含奇偶校验码和循环冗余码CRC，而纠错编码包含海明码。**

### 2 奇偶校验码

奇偶校验码由**N-1位信息元和1位校验元组成**。N-1位的信息元就是我们发送信息里的有效数据，而1位校验元就是用于检错的冗余码。

**奇偶校验码可以分为奇校验码和偶校验码。**

* 奇校验：这串序列1的个数如果为偶数则在**前面或最后面**加个1，**使1的个数变成奇数**，否则加0。
* 偶校验：这串序列1的个数如果为奇数则在**前面或最后面**加个1，**使1的个数变成偶数**，否则加0。

**实例：**

（1）1111 奇校验就是 11111 偶校验就是 01111。

　　（2）1110 奇校验就是 01110 偶校验就是 11110。

**奇偶校验码的特点：**只能检查出**奇数个**比特错误，检错的能力为**50%**，**无法纠错**。

　　怎么理解这个特点呢？

　　我们来看一个实例：比如PCA向PCB发送一串比特位是1101010，采用奇校验码，那么通过奇校验加入冗余位的比特位是11101010，当PCB收到比特位是11101011，那么通过奇校验，我们就知道了在传输过程中有数据发生错误，因为此时1的个数变成了偶数。当如果PCB收到的比较位是11101001，1的个数任然是奇数，此时采用的奇校验是无法检测出错误的，所以，采用奇检验只有50%的检错能力。

### 3 CRC循环冗余码

CRC循环冗余检验码是一种通过多项式除法检测错误的方法，将每个比特串看作一个多项式，采用模2运算。下面是一个简单的CRC循环冗余检验码的原理图：



然后，我们再来做一个实例，就很容易的可以理解CRC循环冗余检验码，当发送方要发送的**帧为1101011011**，生成多项式 G（x）= x4 + x + 1，则 r = 4。

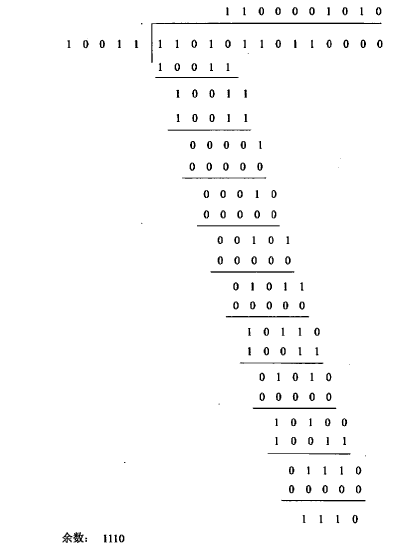
r为何是4，其实就是生成多项式最大的x4 ，此时 r = 4 ，所以则在帧后附加4个0。

**帧**为：1101011011

**生成多项式** G（x）= x4 + x + 1 = 10011

（对应x位写1，没有对应的位填0，就成为相应的生成多项式G（x））。

**发送端计算的过程是**：要发送的数据帧 + r 位的校检序列 得到要发送的数据帧然后与生成多项式进行模2除法运算，**取余数**，这个余数就是需要增加的FCS帧检验序列。



通过计算，可以得到余数为1110，那么实际发送的帧为11010110111110。

**接收端计算的过程是**：当接收端收到帧11010110111110 与 生成多项式进行模2除法运算，然后检查得到的余数，如果余数为0，则表示传输过程是正常的则接收数据，否则是存在错误的则丢弃数据。

**注**：在数据链路层仅使用CRC循环冗余检验码差错控制技术，只能做到对帧的无差错接收，有错误的帧会被丢弃。链路层使用的CRC循环冗余检验码差错控制技术，**能实现比特帧的无差错传输**，**但这还不是可靠传输**。