## 计算机网络

### TCP UDP

#### TCP/UDP区别及应用场景

**区别 6点：连接，可靠性，实时性，一对几，系统资源，字节流/报文段**

1、TCP面向连接；UDP是无连接的。即发送数据之前不需要建立连接

2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达。通过**确认和重传机制**来保证数据传输的可靠性;UDP尽最大努力交付，即不保证可靠交付TCP还**提供了拥塞控制、滑动窗口等机制**来保证传输的质量，而UDP都没有TCP通过校验和，重传控制，序号标识，滑动窗口、确认应答实现可靠传输。如丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制。

3、UDP具有较好的实时性，工作效率比TCP高，适用于对高速传输和实时性有较高的通信或广播通信。

4.每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信

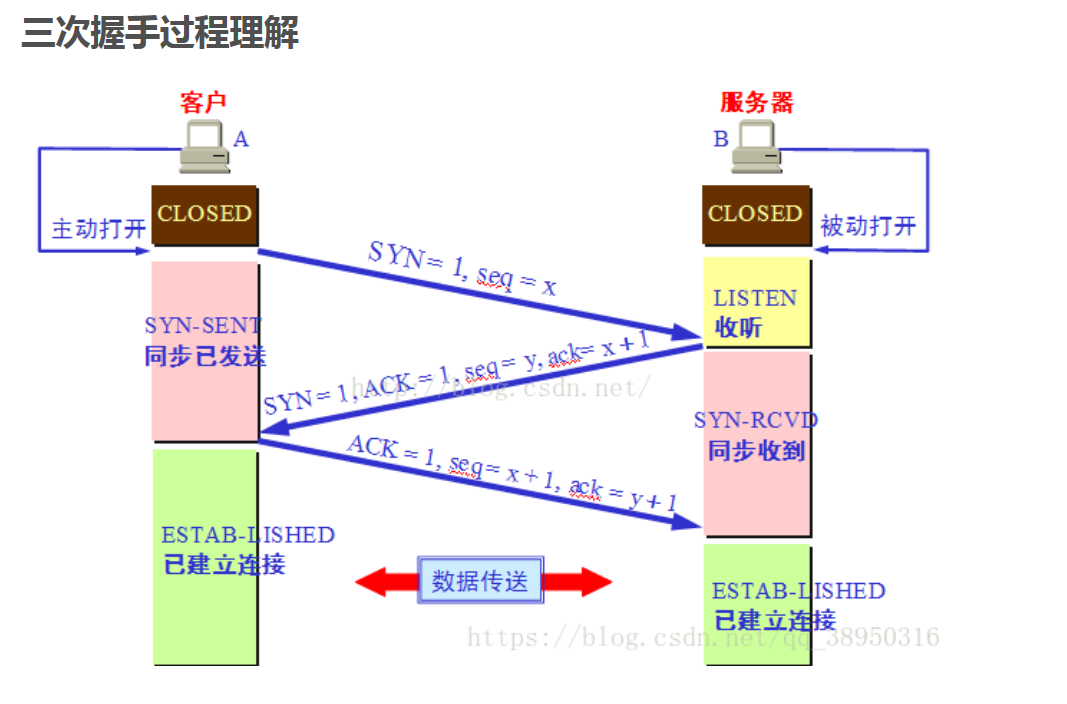
5、TCP对系统资源要求较多，UDP对系统资源要求较少。

6.TCP是基于**字节流**的，将数据看做无结构的字节流进行传输，当应用程序交给TCP的数据长度太长，超过MSS时，TCP就会对数据进行分段，因此TCP的数据是无边界的；而UDP是**面向报文**的，无论应用程序交给UDP层多长的报文，UDP都不会对数据报进行任何拆分等处理，因此UDP保留了应用层数据的边界

TCP：文件传输、重要状态的更新。通信数据完整性需比通信实时性重要，则应该选用 TCP

UDP：视频传输、实时通信。对实时性要求较高

#### TCP 三次握手

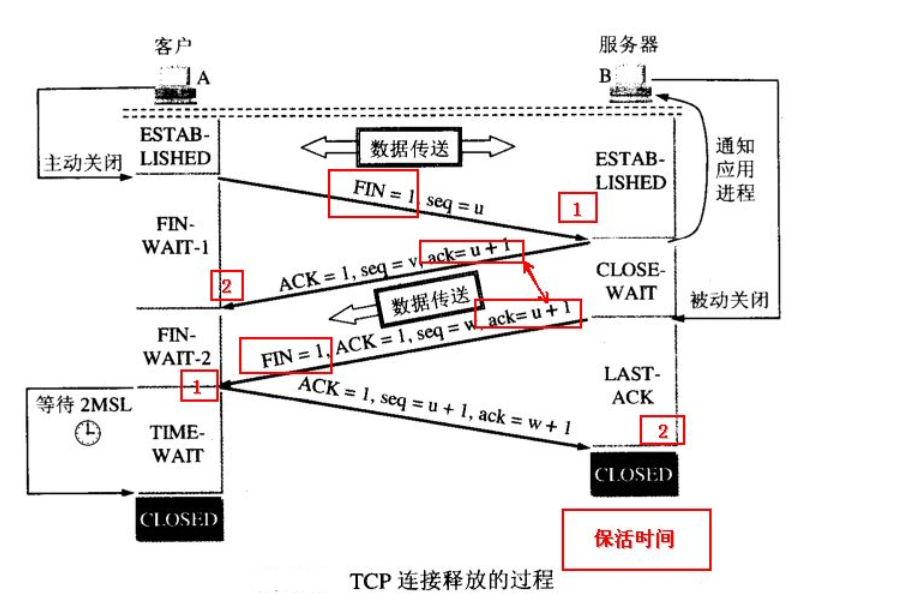


第一次握手：客户端主动打开，发送连接请求报文段，将SYN标识位置为1，序列号置为x（TCP规定SYN=1时不能携带数据，x为随机产生的一个值），然后进入SYN\_SEND状态,等待服务器确认.

第二次握手：服务器收到SYN报文段进行确认，将SYN标识位置为1，ACK置为1，序列号置为y，确认号置为x+1，然后进入SYN\_RECV状态，这个状态被称为半连接状态.

第三次握手：客户端再进行一次确认，将ACK置为1（此时不用SYN），序列号置为x+1，确认号置为y+1发向服务器，最后客户端与服务器都进入ESTABLISHED状态

#### TCP四次挥手



1. 客户端发送一个报文给服务端（没有数据），其中FIN设置为1，序列号置为u，客户端进入FIN\_WAIT\_1状态
2. 服务端收到来自客户端的请求，发送一个ACK给客户端，确认号置为u+1，同时发送序列号为v，服务端年进入CLOSE\_WAIT状态
3. 服务端发送一个FIN给客户端，ACK置为1，Sequence置为w，Acknowledge置为u+1，用来关闭服务端到客户端的数据传送，服务端进入LAST\_ACK状态
4. 客户端收到FIN后，进入TIME\_WAIT状态，接着发送一个ACK给服务端，Acknowledge置为w+1，Sequence Number置为u+1，最后客户端和服务端都进入CLOSED状态

#### 为什么要进行3次挥手

一句话，主要防止已经失效的连接请求报文突然又传送到了服务器，从而产生错误。

如果使用的是两次握手建立连接，假设有这样一种场景，客户端发送了第一个请求连接并且没有丢失，只是因为在网络结点中滞留的时间太长了，由于TCP的客户端迟迟没有收到确认报文，以为服务器没有收到，此时重新向服务器发送这条报文，此后客户端和服务器经过两次握手完成连接，传输数据，然后关闭连接。此时此前滞留的那一次请求连接，网络通畅了到达了服务器，这个报文本该是失效的，但是，两次握手的机制将会让客户端和服务器再次建立连接，这将导致不必要的错误和资源的浪费。

如果采用的是三次握手，就算是那一次失效的报文传送过来了，服务端接受到了那条失效报文并且回复了确认报文，但是客户端不会再次发出确认。由于服务器收不到确认，就知道客户端并没有请求连接。

#### 为什么建立连接需要三次握手，而断开连接需要四次握手

**因为每个方向都需要一个FIN和ACK，当一端发送了FIN包之后，处于半关闭状态，此时仍然可以接收数据包。**   
在建立连接时，服务器**可以把SYN和ACK放在一个包中**发送。   
但是在断开连接时，如果一端收到FIN包，但此时仍有数据未发送完，此时就需要先向对端回复FIN包的ACK。等到将剩下的数据都发送完之后，再向对端发送FIN，断开这个方向的连接。   
因此很多时候**FIN和ACK需要在两个数据包中**发送，因此需要四次握手

#### TIME\_WAIT状态持续时间及原因

持续时间为2MSL，一个数据包在网络中的最长生存时间为MSL。   
**假设最后客户端回复的ACK丢失，服务器端会在超时时间到来时，重传最后一个FIN包。**   
ACK和FIN在网络中的最长生存时间就为2MSL，这样就可以可靠的断开TCP的双向连接。

#### 超时重传和快速重传

超时重传：当超时时间到达时，发送方还未收到对端的ACK确认，就重传该数据包

快速重传：当后面的序号先到达，如接收方接收到了1、 3、 4，而2没有收到，就会立即向发送方重复发送三次ACK=2的确认请求重传。如果发送方连续收到3个相同序号的ACK，就重传该数据包。而不用等待超时 。

#### TCP拥塞控制

TCP通过维护一个拥塞窗口来进行拥塞控制，拥塞控制的原则是，只要网络中没有出现拥塞，拥塞窗口的值就可以再增大一些，以便把更多的数据包发送出去，但只要网络出现拥塞，拥塞窗口的值就应该减小一些，以减少注入到网络中的数据包数。

**拥塞控制与流量控制的区别：**  
拥塞控制是防止过多的数据注入到网络中，可以使网络中的路由器或链路不致过载，是一个全局性的过程。流量控制是点对点通信量的控制，是一个端到端的问题，主要就是抑制发送端发送数据的速率，以便接收端来得及接收。

拥塞的标志1.重传计时器超时 2.接收到三个重复确认

**拥塞控制机制：**慢开始1.慢开始不是指cwnd的增长速度慢（指数增长），而是指TCP开始发送设置cwnd=1。

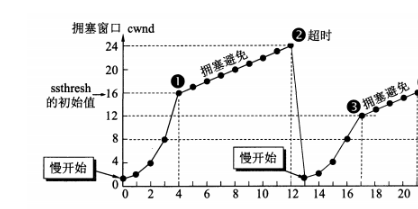
2.思路：不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。这里用报文段的个数的拥塞窗口大小举例说明慢开始算法，实时拥塞窗口大小是以字节为单位的

慢开始门限（ssthresh）

为了防止cwnd增长过大引起网络拥塞，设置一个慢开始门限（ssthresh状态变量）   
当cnwd＜ssthresh，使用慢开始算法   
当cnwd=ssthresh，既可使用慢开始算法，也可以使用拥塞避免算法   
当cnwd＞ssthresh，使用拥塞避免算法

拥塞避免（按线性规律增长）

拥塞避免并非完全能够避免拥塞，是说在拥塞避免阶段将拥塞窗口控制为按线性规律增长，使网络比较不容易出现拥塞。   
2.思路：让拥塞窗口cwnd缓慢地增大，即每经过一个往返时间RTT就把发送方的拥塞控制窗口加一。

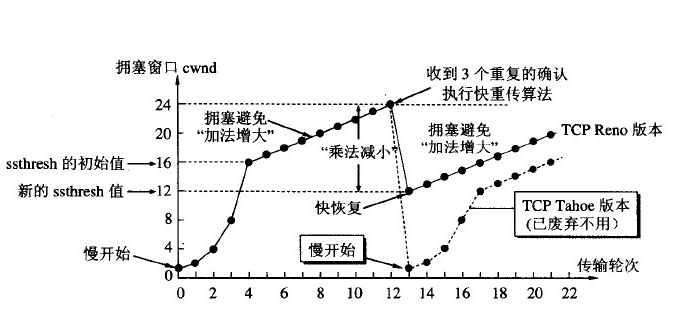
**无论是在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞（其根据就是没有收到确认，虽然没有收到确认可能是其他原因的分组丢失，但是因为无法判定，所以都当做拥塞来处理），就把慢开始门限设置为出现拥塞时的发送窗口大小的一半。然后把拥塞窗口设置为1，执行慢开始算法。**   


乘法减小：无论是慢开始阶段还是拥塞避免，只要出现了网络拥塞（超时），就把慢开始门限值ssthresh减半   
加法增大：执行拥塞避免算法后，拥塞窗口线性缓慢增大，防止网络过早出现拥塞

增加快重传与快恢复

快重传

1.快重传要求接收方在收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时捎带确认。快重传算法规定，**发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器时间到期。**



2.由于不需要等待设置的重传计时器到期，能尽早重传未被确认的报文段，能提高整个网络的吞吐量。

快恢复（与快重传配合使用）

1.采用快恢复算法时，慢开始只在TCP连接建立时和网络出现超时时才使用。

2.当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把ssthresh门限减半。但是接下去并不执行慢开始算法。

3.考虑到如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。所以此时不执行慢开始算法，而是将cwnd设置为ssthresh的大小，然后执行拥塞避免算法。

注意

发送方窗口的上限值=Min（接受窗口rwnd，拥塞窗口cwnd）

rwnd＞cwnd 接收方的接收能力限制发送方窗口的最大值

rwnd＜cwnd 网络的拥塞限制发送方窗口的最大值

#### 10.TCP如何保证可靠性：

校验和

序列号

确认应答

超时重传：发送方没有介绍到响应的ACK报文原因可能有两点：

1. 数据在传输过程中由于网络原因等直接全体丢包，接收方根本没有接收到。

2. 接收方接收到了响应的数据，但是发送的ACK报文响应却由于网络原因丢包了。

发送方在发送完数据后等待一个时间，时间到达没有接收到ACK报文，那么对刚才发送的数据进行重新发送。如果是刚才第一个原因，接收方收到二次重发的数据后，便进行ACK应答。如果是第二个原因，接收方发现接收的数据已存在（判断存在的根据就是序列号，所以上面说序列号还有去除重复数据的作用），那么直接丢弃，仍旧发送ACK应答。

那么发送方发送完毕后等待的时间是多少呢？如果这个等待的时间过长，那么会影响TCP传输的整体效率，如果等待时间过短，又会导致频繁的发送重复的包。如何权衡？

由于TCP传输时保证能够在任何环境下都有一个高性能的通信，因此这个最大超时时间（也就是等待的时间）是动态计算的。

连接管理：三次握手 四次挥手

流量控制：发送方把数据发送得过快，接收方可能会来不及接收，这就会造成数据的丢失。

TCP的流量控制是利用**滑动窗口机制**实现的，接收方在返回的ACK中会包含自己的接收窗口的大小，以控制发送方的数据发送。

但是当某个ACK报文丢失了，就会出现A等待B确认，并且B等待A发送数据的死锁状态。为了解决这种问题，TCP引入了持续计时器（Persistence timer），当A收到rwnd=0时，就启用该计时器，时间到了则发送一个1字节的探测报文，询问B是很忙还是上个ACK丢失了，然后B回应自身的接收窗口大小，返回仍为0（A重设持续计时器继续等待）或者会重发rwnd=x。

拥塞控制

### 2.HTTP HTTPs

#### HTTPs请求过程

#### 客户端发送请求https连接。

2、服务器返回加密公钥，通常是SSL证书。

3、客户端从这个SSL证书解析出公钥，并随机生成一个key，通过公钥加密这个key发送给服务器（这一步是安全的因为只有服务器才有私钥能读出这个key）。

4、服务器通过私钥解密出key。

5、客户端使用这个key来加密需要传输的数据。

6、服务器使用key来解析数据。

简单的来说SSL加密的方式是使用一个密钥来加密另一个密钥（key），在使用被加密的密钥来加密数据。这样的做法固然保证了安全性，但每次连接时都需要使用密钥加密，导致请求会需要额外的开销，同时服务器第一次返回的公钥的可靠性需要第三方来保证，通常是购买SSL证书。这也会造成额外的经济开销。

#### Http和Https的区别

1.Http协议运行在TCP之上，明文传输，客户端与服务器端都无法验证对方的身份；

Https是身披SSL(Secure Socket Layer)外壳的Http，运行于SSL上，SSL运行于TCP之上，是添加了加密和认证机制的HTTP。

2.端口不同：Http与Https使用不同的连接方式，http是80，https是443；

3.资源消耗：和HTTP通信相比，Https通信会由于加密解密处理消耗更多的CPU和内存资源；

4.开销：Https通信需要证书，而证书一般需要向认证机构购买；   
Https的加密机制是一种共享密钥加密和公开密钥加密并用的混合加密机制。

#### Http状态码

200 服务器成功返回网页

400(错误请求)服务器不理解请求的语法。

403 forbidden，表示对请求资源的访问被服务器拒绝（服务器已经理解请求，但是拒绝执行它）

404 请求的网页不存在

500(服务器内部错误)服务器遇到错误，无法完成请求。

503 服务器超时

#### Https作用

1．地址栏绿色小锁（主要体现在浏览器地址栏，让网友客户更加信任，体现品牌形象，让客户更安全的访问）

2．防劫持（移动数据劫持、HTTP协议劫持，流量劫持等最为频繁，对于做百度推广或移动广告的，这一点必须要关注，不然广告都是给别人做的！全站Https是根治运营商、中间人流量劫持的解决方案，不仅可以杜绝网页中被插入的小广告，更可以保护用户隐私安全。）

3.认证网站服务器的真实身份，网站安装由第三方权威机构颁发的网站ssl证书，也就意味着，网站所在的服务器真实身份已经认证过了，网站的信息也是真实的，用户不用担心网站是不是安全的网站。访客发访问网站的时候，可以看到地址栏那有一个绿色是锁的标识，点击就可查看网站的身份信息，安装ssl证书的网站可以有效的避免网站钓鱼等不安全因素造成经济损失。

4.保护网站的数据安全。通过对数据传输层加密，确保数据在传输的过程中，数据不被窃取，保证数据的完整性，安全。防止用户信息，财务信息等一些重要的数据被篡改，以及被窃取。也就是说安装ssl证书的网站数据安全性比没有安装的网站安全性更高。

#### get post 区别？

1.GET请求在URL中传送的参数是有长度限制的，而POST没有。

2.GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。

3.GET参数通过URL传递，POST放在Request body中。

4.GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。

5.GET请求只能进行url编码，而POST支持多种编码方式。

6.GET请求会被浏览器主动cache，而POST不会，除非手动设置。

7.GET产生的URL地址可以被Bookmark，而POST不可以。

8.GET在浏览器回退时是无害的，而POST会再次提交请求。

#### 网页从输入url到完整呈现出来都经历了哪几个过程

1输入网址；

2发送到DNS服务器，并获取域名对应的web服务器对应的ip地址；

3与web服务器建立TCP连接；

4浏览器向web服务器发送http请求；

5.web服务器响应请求，并返回指定url的数据（或错误信息，或重定向的新的url地址）；

6浏览器下载web服务器返回的数据及解析html源文件；

7生成DOM树，解析css和js，渲染页面，直至显示完成；

#### Cookie和session 区别及联系？

**区别六点：存放位置，形式，用途，路径，安全性，大小**

**位置：**cookie数据保存在客户端，session数据保存在服务器端

**形式：**session 是以对象的形式保存，cookie 以字符串的形式保存；

**用途**：session 适合做客户的身份验证，cookie 适合保存用户的个人设置，爱好等；

**路径：**session 不能区分路径，同一用户在访问一个网站期间，所有的 session 在任何一个地方都可以访问到；cookie 中如果设置了参数路径，那么同一个网站下的 cookie 互相访问不到；

**安全性：**cookie 不是很安全，别人可以分析存放在本地的 cookie 并进行 cookie 欺骗，session 较 cookie 更安全一些；

**大小：**单个 cookie 在客户端的限制是 3K，就是说一个站点在客户端存放的 COOKIE 不能 3K。不同浏览器所含 cookie 的最大个数不同，一般 30 到 50 个；一般认为 session 没有大小限制。

**联系：**

session 需要借助cookie才能正常工作。如果客户端完全禁止cookie，session 将失效，因为session是由服务器维持的一个服务端的存储空间，用户在连接服务器时，会由服务器生成唯一的 session.id，用该 session.id 为标识来存取服务端的 session 空间。而 session.id 存储在 cookie 中，用户提交页面时会将这个 session id 提交到服务端，来存取 session 数据.这一过程是不用开发人员干预的，所以一旦客户端禁用 cookie，那么 session 也会失效。

session 会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能，如果主要考虑到减轻服务器性能方面，应当使用 cookie；

Session.id 是服务器和客户端链接时候随机分配的。

## 数据结构

**面试中链表和树比较高频**

### 队列

队列与广度优先遍历算法紧密联系

### 堆

堆就是用**数组实现**的二叉树，所有它没有使用父指针或者子指针。堆根据“堆属性”来排序，“堆属性”决定了树中节点的位置。

堆的常用方法：

构建优先队列

支持堆排序

快速找出一个集合中的最小值（或者最大值）

堆分为两种：最大堆**和**最小堆，两者的差别在于节点的排序方式。

在最大堆中，父节点的值比每一个子节点的值都要大。在最小堆中，父节点的值比每一个子节点的值都要小。这就是所谓的“堆属性”，并且这个属性对堆中的每一个节点都成立。

根据这一属性，那么最大堆总是将其中的最大值存放在树的根节点。而对于最小堆，根节点中的元素总是树中的最小值。堆属性非常的有用，因为堆常常被当做优先队列使用，因为可以快速的访问到“最重要”的元素。

**注意：**堆的根节点中存放的是最大或者最小元素，但是其他节点的排序顺序是未知的。例如，在一个最大堆中，最大的那一个元素总是位于 index 0 的位置，但是最小的元素则未必是最后一个元素。唯一能够保证的是最小的元素是一个叶节点，但是不确定是哪一个。

#### 堆和普通树的区别

堆并不能取代二叉搜索树（二叉查找树，二叉排序树Binary Search Tree），它们之间有相似之处也有一些不同。我们来看一下两者的主要差别：

**节点的顺序。**在二叉搜索树中，左子节点必须比父节点小，右子节点必须必比父节点大。但是在堆中并非如此。在最大堆中两个子节点都必须比父节点小，而在最小堆中，它们都必须比父节点大。

**内存占用。**普通树占用的内存空间比它们存储的数据要多。你必须为节点对象以及左/右子节点指针分配额外的内存。**堆仅仅使用一个数组，且不使用指针。**

**平衡。**二叉搜索树必须是“平衡”的情况下，其大部分操作的复杂度才能达到**O(log n)**。你可以按任意顺序位置插入/删除数据，或者使用 AVL 树或者红黑树，但是在堆中实际上不需要整棵树都是有序的。我们只需要满足对属性即可，所以在堆中平衡不是问题。因为堆中数据的组织方式可以保证**O(log n)** 的性能。

**搜索。**在二叉树中搜索会很快，但是在堆中搜索会很慢。在堆中搜索不是第一优先级，因为使用堆的目的是将最大（或者最小）的节点放在最前面，从而快速的进行相关插入、删除操作。

#### 来自数组的树

用数组来实现树相关的数据结构也许看起来有点古怪，但是它在时间和空间上都是很高效的。

我们准备将上面的例子中的树这样存储：[ 10, 7, 2, 5, 1 ]

就这多！我们除了一个简单的数组以外，不需要任何额外的空间。

如果我们不允许使用指针，那么我们怎么知道哪一个节点是父节点，哪一个节点是它的子节点呢？问得好！**节点在数组中的位置index 和它的父节点以及子节点的索引之间有一个映射关系。**

如果 i 是节点的索引，那么下面的公式就给出了它的父节点和子节点在数组中的位置：

parent(i) = floor((i - 1)/2)

left(i) = 2i + 1

right(i) = 2i + 2

注意 right(i) 就是简单的 left(i) + 1。左右节点总是处于相邻的位置。

我们将写公式放到前面的例子中验证一下。

| Node | Array index (**i**) | Parent index | Left child | Right child |
| --- | --- | --- | --- | --- |
| 10 | 0 | -1 | 1 | 2 |
| 7 | 1 | 0 | 3 | 4 |
| 2 | 2 | 0 | 5 | 6 |
| 5 | 3 | 1 | 7 | 8 |
| 1 | 4 | 1 | 9 | 10 |

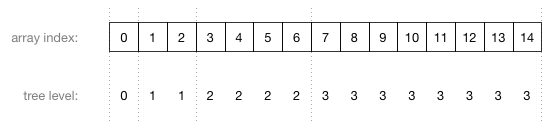
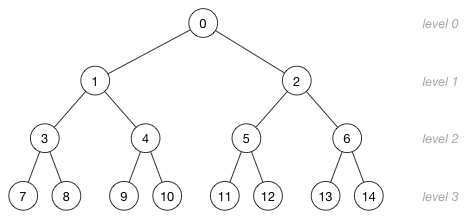
**注意：**根节点**(10)**没有父节点，因为 **-1** 不是一个有效的数组索引。同样，节点 **(2)**，**(5)**和**(1)** 没有子节点，因为这些索引已经超过了数组的大小，所以我们在使用这些索引值的时候需要保证是有效的索引值。

复习一下，在**最大堆**中，父节点的值总是要大于（或者等于）其子节点的值。这意味下面的公式对数组中任意一个索引 i都成立：

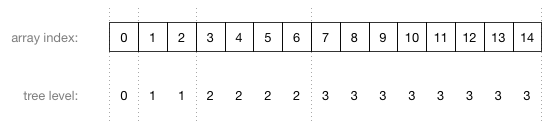
array**[**parent**(i)] >=** array**[i]**

如你所见，这些公式允许我们不使用指针就可以找到任何一个节点的父节点或者子节点。事情比简单的去掉指针要复杂，但这就是交易：我们**节约了空间，但是要进行更多计算。幸好这些计算很快并且只需要O(1)的时间。**

理解数组索引和节点位置之间的关系非常重要。这里有一个更大的堆，它有15个节点被分成了4层：

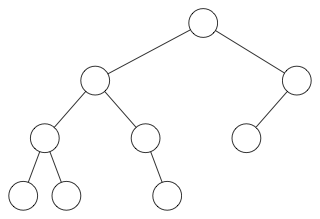


图片中的数字不是节点的值，而是存储这个节点的数组索引！这里是数组索引和树的层级之间的关系：

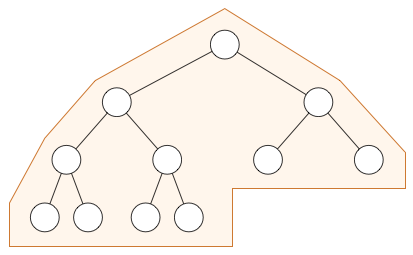


由上图可以看到，数组中父节点总是在子节点的前面。

注意这个方案与一些限制。你可以在普通二叉树中按照下面的方式组织数据，但是在堆中不可以：



在堆中，在当前层级所有的节点都已经填满之前不允许开是下一层的填充，所以堆总是有这样的形状：

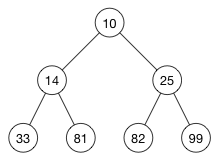


**注意：**你可以使用普通树来模拟堆，但是那对空间是极大的浪费。

小测验，假设我们有这样一个数组：

[ 10, 14, 25, 33, 81, 82, 99 ]

这是一个有效的堆吗？答案是 yes ！**一个从低到高有序排列的数组是以有效的最小堆**，我们可以将这个堆画出来：



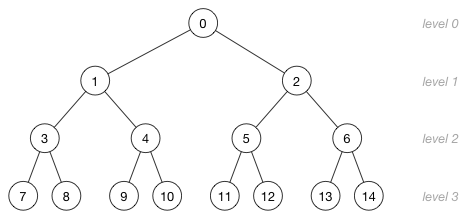
堆属性适用于每一个节点，因为父节点总是比它的字节点小。（你也可以验证一下：**一个从高到低有序排列的数组是一个有效的最大堆**）

**注意：**并不是每一个最小堆都是一个有序数组！要将堆转换成有序数组，需要使用堆排序。

#### 更多数学公式

树的高度是指从树的根节点到最低的叶节点所需要的步数，或者更正式的定义：高度是指节点之间的边的最大值。一个高度为 h 的堆有 h+1 层。

下面这个堆的高度是3，所以它有4层：



**如果一个堆有 n 个节点，那么它的高度是** h = floor(log2(n))。这是因为我们总是要将这一层完全填满以后才会填充新的一层。上面的例子有 15 个节点，所以它的高度是 floor(log2(15)) = floor(3.91) = 3。

如果最下面的一层已经填满，那么那一层包含 2^h 个节点。树中这一层以上所有的节点数目为 2^h - 1。同样是上面这个例子，最下面的一层有8个节点，实际上就是 2^3 = 8。前面的三层一共包含7的节点，即：2^3 - 1 = 8 - 1 = 7。

所以整个堆中的节点数目为：\* 2^(h+1) - 1\*。上面的例子中，2^4 - 1 = 16 - 1 = 15

叶节点总是位于数组的 floor(n/2) 和 n-1 之间。

#### 可以用堆做什么？

有两个原始操作用于保证插入或删除节点以后堆是一个有效的最大堆或者最小堆：

* shiftUp(): 如果一个节点比它的父节点大（最大堆）或者小（最小堆），那么需要将它同父节点交换位置。这样是这个节点在数组的位置上升。
* shiftDown(): 如果一个节点比它的子节点小（最大堆）或者大（最小堆），那么需要将它向下移动。这个操作也称作“堆化（heapify）”。

shiftUp 或者 shiftDown 是一个递归的过程，所以它的时间复杂度是 **O(log n)**。

基于这两个原始操作还有一些其他的操作：

* insert(value): 在堆的尾部添加一个新的元素，然后使用 **shiftUp** 来修复堆。
* remove(): 移除并返回最大值（最大堆）或者最小值（最小堆）。为了将这个节点删除后的空位填补上，**需要将最后一个元素移到根节点的位置，然后使用 shiftDown 方法来修复堆**。
* removeAtIndex(index): 和 remove() 一样，差别在于可以移除堆中任意节点，而不仅仅是根节点。当它与子节点比较发现无序时使用 shiftDown()，如果与父节点比较发现无序则使用 shiftUp()。
* replace(index, value)：将一个更小的值（最小堆）或者更大的值（最大堆）赋值给一个节点。由于这个操作破坏了堆属性，所以需要使用 **shiftUp**() 来修复堆属性。

**上面所有的操作的时间复杂度都是 O(log n)，**因为 shiftUp 和 shiftDown 都很费时。还有少数一些操作需要更多的时间：

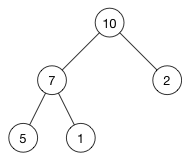
* search(value):**堆不是为快速搜索而建立的**，但是 replace() 和 removeAtIndex() 操作需要找到节点在数组中的index，所以你需要先找到这个index。时间复杂度：**O(n)**。
* buildHeap(array):通过反复调用 insert() 方法将一个（无序）数组转换成一个堆。如果你足够聪明，你可以在 **O(n)** 时间内完成。
* 堆排序：由于堆就是一个数组，我们可以使用它独特的属性将数组从低到高排序。时间复杂度：**O(n lg n)**。

堆还有一个 peek() 方法，不用删除节点就返回最大值（最大堆）或者最小值（最小堆）。时间复杂度 **O(1)** 。

**注意：**到目前为止，堆的常用操作还是使用 insert() 插入一个新的元素，和通过 remove()移除最大或者最小值。两者的时间复杂度都是**O(log n)**。其他的操作是用于支持更高级的应用，比如说建立一个优先队列。

##### 插入

我们通过一个插入例子来看看插入操作的细节。我们将数字 16 插入到这个堆中：

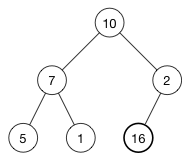
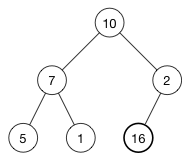
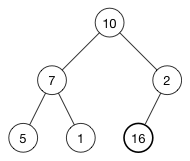


堆的数组是： [ 10, 7, 2, 5, 1 ]。

第一股是将新的元素插入到数组的尾部。数组变成：

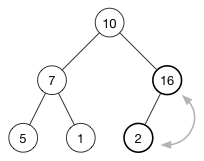
[ 10, 7, 2, 5, 1, 16 ]

相应的树变成了：



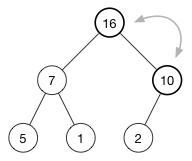
不行的是，现在堆属性不满足，因为 2 在 16 的上面，我们需要将大的数字在上面（这是一个最大堆）

为了恢复堆属性，我们需要交换 16 和 2。



现在还没有完成，因为 10 也比 16 小。我们继续交换我们的插入元素和它的父节点，**直到它的父节点比它大或者我们到达树的顶部**。这就是所谓的 **shift-up**，每一次插入操作后都需要进行。它将一个太大或者太小的数字“浮起”到树的顶部。

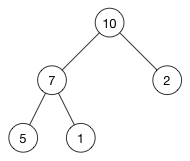
最后我们得到的堆：



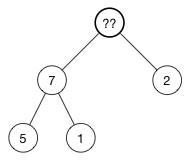
现在每一个父节点都比它的子节点大。

##### 删除根节点

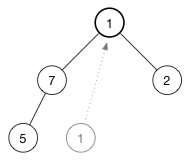
我们将这个树中的 (10) 删除：



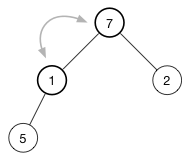
现在顶部有一个空的节点，怎么处理？



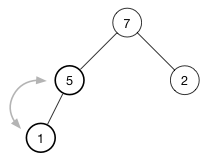
当插入节点的时候，我们将新的值返给数组的尾部。现在我们来做相反的事情：我们取出数组中的最后一个元素，将它放到树的顶部，然后再修复堆属性。



现在来看怎么 **shift-down** (1)。为了保持最大堆的堆属性，我们需要树的顶部是最大的数据。现在有两个数字可用于交换 7 和 2。我们选择这两者中的较大者称为最大值放在树的顶部，所以交换 7 和 1，现在树变成了：



继续堆化直到该节点没有任何子节点或者它比两个子节点都要大为止。对于我们的堆，我们只需要再有一次交换就恢复了堆属性：

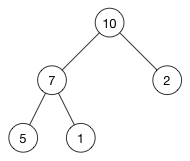


##### 删除任意节点

绝大多数时候你需要删除的是堆的根节点，因为这就是堆的设计用途。

但是，删除任意节点也很有用。这是 **remove()** 的通用版本，它可能会使用到 **shiftDown** 和 **shiftUp**。

我们还是用前面的例子，删除 (7):



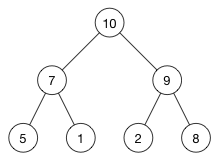
对应的数组是[ 10, 7, 2, 5, 1 ]

你知道，移除一个元素会破坏最大堆或者最小堆属性。我们需要将**删除的元素和最后一个元素交换：**

[ 10, 1, 2, 5, 7 ]

最后一个元素就是我们需要返回的元素；然后调用 **removeLast()** 来将它删除。 1比它的子节点小，所以需要 shiftDown() 来修复。

然而，shift down 不是我们要处理的唯一情况。也有可能我们需要 shift up。考虑一下从下面的堆中删除 (5) 会发生什么：



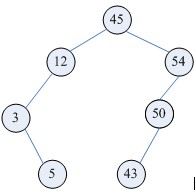
现在 (5) 和 (8) 交换了。因为 (8) 比它的父节点大，我们需要 shiftUp()。

### 栈

1. 后进先出
2. 与递归紧密联系

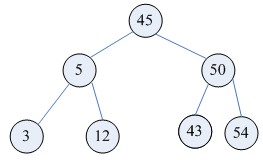
### 树

#### 二叉查找树（二叉排序树）BST



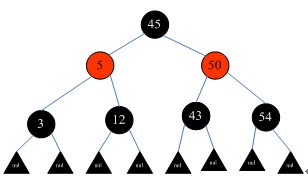
二叉查找树是一种动态查找表（图a），具有这些质：                                   
（1）若它的左子树不为空，则左子树上的所有节点的值都小于它的根节点的值；  
（2）若它的右子树不为空，则右子树上所有节点的值都大于它的根节点的值；  
（3）其他的左右子树也分别为二叉查找树；  
（4）二叉查找树是动态查找表，在查找的过程中可见添加和删除相应的元素，在这些操作中需要保持二叉查找树的以上性质。

#### 平衡二叉树（AVL树）



含有相同节点的二叉查找树可以有不同的形态，而二叉查找树的平均查找长度与树的深度有关，所以需要找出一个查找平均长度最小的一棵，那就是平衡二叉树（图b），具有以下性质：  
（1）要么是棵空树，要么其根节点左右子树的深度之差的绝对值不超过1；  
（2）其左右子树也都是平衡二叉树；  
（3）二叉树节点的平衡因子定义为该节点的左子树的深度减去右子树的深度。则平衡二叉树的所有节点的平衡因子只可能是-1,0,1。

#### 红黑树

（图c）

红黑树是一种自平衡二叉树，在平衡二叉树的基础上每个节点又增加了一个颜色的属性，节点的颜色只能是红色或黑色。具有以下性质：  
（1）根节点只能是黑色；  
（2）红黑树中所有的叶子节点后面再接上左右两个空节点，这样可以保持算法的一致性，而且所有的空节点都是黑色；  
（3）其他的节点要么是红色，要么是黑色，红色节点的父节点和左右孩子节点都是黑色，及黑红相间；  
（4）在任何一棵子树中，从根节点向下走到空节点的路径上所经过的黑节点的数目相同，从而保证了是一个平衡二叉树。

#### B树

B-Tree是为磁盘等外存储设备设计的一种平衡查找树。因此在讲B-Tree之前先了解下磁盘的相关知识。

系统从磁盘读取数据到内存时是以**磁盘块**（block）为基本单位的，位于同一个磁盘块中的数据会被一次性读取出来，而不是需要什么取什么。

InnoDB存储引擎中有页（Page）的概念，页是其磁盘管理的最小单位。InnoDB存储引擎中默认每个页的大小为16KB，可通过参数innodb\_page\_size将页的大小设置为4K、8K、16K，在[MySQL](http://lib.csdn.net/base/mysql" \t "https://www.cnblogs.com/vianzhang/p/_blank" \o "MySQL知识库)中可通过如下命令查看页的大小：

mysql> show variables like 'innodb\_page\_size';

而系统一个磁盘块的存储空间往往没有这么大，因此InnoDB每次申请磁盘空间时都会是若干地址连续磁盘块来达到页的大小16KB。InnoDB在把磁盘数据读入到磁盘时会以页为基本单位，在查询数据时如果一个页中的每条数据都能有助于定位数据记录的位置，这将会减少磁盘I/O次数，提高查询效率。

B-Tree结构的数据可以让系统高效的找到数据所在的磁盘块。为了描述B-Tree，首先定义一条记录为一个二元组[key, data] ，key为记录的键值，对应表中的主键值，data为一行记录中除主键外的数据。对于不同的记录，key值互不相同。

一棵m阶的B-Tree有如下特性：   
1. 每个节点最多有m个孩子。   
2. 除了根节点和叶子节点外，其它每个节点至少有Ceil(m/2)个孩子。 （上界）  
3. 若根节点不是叶子节点，则至少有2个孩子   
4. 所有叶子节点都在同一层，且不包含其它关键字信息   
5. 每个非终端节点包含n个关键字信息（P0,P1,…Pn, k1,…kn）   
6. 关键字的个数n满足：ceil(m/2)-1 <= n <= m-1   
7. ki(i=1,…n)为关键字，且关键字升序排序。   
8. Pi(i=1,…n)为指向子树根节点的指针。P(i-1)指向的子树的所有节点关键字均小于ki，但都大于k(i-1)

B-Tree中的每个节点根据实际情况可以包含大量的关键字信息和分支，如下图所示为一个3阶的B-Tree：   


每个节点占用一个盘块的磁盘空间，一个节点上有两个升序排序的关键字和三个指向子树根节点的指针，指针存储的是子节点所在磁盘块的地址。两个关键词划分成的三个范围域对应三个指针指向的子树的数据的范围域。以根节点为例，关键字为17和35，P1指针指向的子树的数据范围为小于17，P2指针指向的子树的数据范围为17~35，P3指针指向的子树的数据范围为大于35。

模拟查找关键字29的过程：

1. 根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】
2. 比较关键字29在区间（17,35），找到磁盘块1的指针P2。
3. 根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】
4. 比较关键字29在区间（26,30），找到磁盘块3的指针P2。
5. 根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】
6. 在磁盘块8中的关键字列表中找到关键字29。

分析上面过程，发现需要3次磁盘I/O操作，和3次内存查找操作。由于内存中的关键字是一个有序表结构，可以利用二分法查找提高效率。而3次磁盘I/O操作是影响整个B-Tree查找效率的决定因素。B-Tree相对于AVLTree缩减了节点个数，使每次磁盘I/O取到内存的数据都发挥了作用，从而提高了查询效率。

#### B+Tree

B+Tree是在B-Tree基础上的一种优化，使其更适合实现外存储索引结构，InnoDB存储引擎就是用B+Tree实现其索引结构。

从上一节中的B-Tree结构图中可以看到每个节点中不仅包含数据的key值，还有data值。而每一个页的存储空间是有限的，如果data数据较大时将会导致每个节点（即一个页）能存储的key的数量很小，当存储的数据量很大时同样会导致B-Tree的深度较大，增大查询时的磁盘I/O次数，进而影响查询效率。在B+Tree中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储key值信息，这样可以大大加大每个节点存储的key值数量，降低B+Tree的高度。

B+Tree相对于B-Tree有几点不同：

1. 非叶子节点只存储键值信息。
2. 所有叶子节点之间都有一个链指针。
3. 数据记录都存放在叶子节点中。

将上一节中的B-Tree优化，由于B+Tree的非叶子节点只存储键值信息，假设每个磁盘块能存储4个键值及指针信息，则变成B+Tree后其结构如下图所示：   


通常在B+Tree上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（即数据节点）之间是一种链式环结构。因此可以对B+Tree进行两种查找运算：一种是对于主键的范围查找和分页查找，另一种是从根节点开始，进行随机查找。

可能上面例子中只有22条数据记录，看不出B+Tree的优点，下面做一个推算：

InnoDB存储引擎中页的大小为16KB，一般表的主键类型为INT（占用4个字节）或BIGINT（占用8个字节），指针类型也一般为4或8个字节，也就是说一个页（B+Tree中的一个节点）中大概存储16KB/(8B+8B)=1K个键值（因为是估值，为方便计算，这里的K取值为〖10〗^3）。也就是说一个深度为3的B+Tree索引可以维护10^3 \* 10^3 \* 10^3 = 10亿 条记录。

实际情况中每个节点可能不能填充满，因此在数据库中，B+Tree的高度一般都在2~4层。[mysql](http://lib.csdn.net/base/mysql" \t "https://www.cnblogs.com/vianzhang/p/_blank" \o "MySQL知识库)的InnoDB存储引擎在设计时是将根节点常驻内存的，也就是说查找某一键值的行记录时最多只需要1~3次磁盘I/O操作。

数据库中的B+Tree索引可以分为聚集索引（clustered index）和辅助索引（secondary index）。上面的B+Tree示例图在数据库中的实现即为聚集索引，聚集索引的B+Tree中的叶子节点存放的是整张表的行记录数据。辅助索引与聚集索引的区别在于辅助索引的叶子节点并不包含行记录的全部数据，而是存储相应行数据的聚集索引键，即主键。当通过辅助索引来查询数据时，InnoDB存储引擎会遍历辅助索引找到主键，然后再通过主键在聚集索引中找到完整的行记录数据。

## 操作系统

### 1.进程和线程以及区别

线程：线程是进程的一个实体，是CPU**调度和分派**的基本单元。  
进程：进程是具有一定独立功能的程序，它是系统进程资源分配和调度的一个独立单元。

**区**别：

1）一个线程只属于一个进程，一个进程包含一个或者多个线程。

2）进程拥有独立的内存单元，而多个线程共享内存。

3）进程的创建调用fork或者vfork,而线程的创建调用pthead\_create，进程结束后它拥有的所有线程都将销毁，而线程的结束不会影响同个进程中的其他线程的结束。

4）线程是轻量级的进程，它的创建和销毁所需要的时间比进程小很多，所有操作系统中的执行功能都是创建线程去完成的。

5）线程中执行时一般都要进行同步和互斥，因为他们共享同一进程的资源。

**进程与线程的区别？**

进程是CPU资源分配的基本单位，线程是独立运行和独立调度的基本单位（CPU上真正运行的是线程）。

进程拥有自己的资源空间，一个进程包含若干个线程，线程与CPU资源分配无关，多个线程共享同一进程内的资源。

线程的调度与切换比进程快很多。

1）进程有自己的独立地址空间，线程没有

2）进程是资源分配的最小单位，线程是CPU调度的最小单位

3）进程和线程通信方式不同(线程之间的通信比较方便。同一进程下的线程共享数据（比如全局变量，静态变量），通过这些数据来通信不仅快捷而且方便，当然如何处理好这些访问的同步与互斥正是编写多线程程序的难点。而进程之间的通信只能通过[进程通信](http://baike.baidu.com/view/549640.htm)的方式进行。

4）进程上下文切换开销大，线程开销小

5）一个进程挂掉了不会影响其他进程，而线程挂掉了会影响其他线程

6）对进程操作一般开销都比较大，对线程开销就小了

### 2死锁(deadlock)？

死锁 :两个或两个以上的进程在执行过程中,因争夺资源而造成的一种**互相等待**的现象,若无**外力**作用,它们都将无法推进下去

**产生原因：**

1）系统资源不足。

2）进程运行推进顺序不合里。

3）资源分配不当。

**必要条件：**

1）**互斥条件**：一个资源每次只能被一个进程使用。

2）**请求与保持条件：**一个进程因请求资源而阻塞时，对已获得的资源保持不放。

3）**不剥夺条件:**进程已获得的资源，在末使用完之前，不能强行剥夺。

4）**循环等待条件:**若干进程之间形成一种头尾相接的循环等待资源关系。

**死锁的解除与预防：**

在系统设计、进程调度等方面注意不让四个必要条件成立，确定资源的合理分配算法，避免进程永久占据系统资源。此外，也要防止进程在处于等待状态的情况下占用资源。

### 3.虚拟内存？优缺点？

**定义**：

具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充得一种存储器系统。其逻辑容量由内存之和和外存之和决定。

与传统存储器比较虚拟存储器有以下三个主要特征：

1）多次性，是指无需在作业运行时一次性地全部装入内存，而是允许被分成多次调入内存运行。

2）对换性，是指无需在作业运行时一直常驻内存，而是允许在作业的运行过程中，进行换进和换出。

3）虚拟性，是指从逻辑上扩充内存的容量，使用户所看到的内存容量，远大于实际的内存容量。

虚拟内存的实现有以下两种方式：

**请求分页存储管理。**

**请求分段存储管理。**

### 4.进程间通信有哪些方式？它们的区别？

1.管道（pipe）：管道是一种半双工的通信方式，数据只能单向流动，而且只能在有血缘关系的进程间使用，进程的血缘关系通常是指父子进程关系。

2.命名管道（named pipe）：也是半双工的通信方式，但是它允许无亲缘关系关系进程间通信。

3.信号（signal）：是一种比较复杂的通信方式，用于通知接收进程某一事件已经发生。

4.信号量（semophere）：信号量是一个计数器，可用来控制多个进程对共享资源的访问。它通常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

5.消息队列（message queue）:消息队列是由消息组成的链表，存放在内核中，并由消息队列标识符标识。消息队列克服了信号传递消息少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

6.共享内存（shared memory）:就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问，共享内存是最快的IPC方式，它是针对其他进程间的通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量等配合使用，来实现进程间的同步和通信。

7.套接字（socket）：套接口也是进程间的通信机制，与其他通信机制不同的是它可用于不同及其间的进程通信。

几种方式的比较：

管道：速度慢、容量有限

消息队列：容量收到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题。

信号量：不能传递复杂信息，只能用来同步。

共享内存：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全。

### 5.进程的调度算法有哪些？

1.先来先服务（FCFS）:此算法的原则是按照作业到达后备作业队列（或进程进入就绪队列）的先后次序选择作业（或进程）

2.短作业优先（SJF:Shortest Process First）：这种算法主要用于作业调度，它从作业后备序列中挑选所需运行时间最短的作业进入主存运行。

3.时间片轮转调度算法：当某个进程执行的时间片用完时，调度程序便终止该进程的执行，并将它送到就绪队列的末尾，等待分配下一时间片再执行。然后把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证队列中的所有进程，在已给定的时间内，均能获得一时间片处理机执行时间。

4.高响应比优先：按照高响应比（已等待时间+要求运行时间）/要求运行时间 优先的原则，在每次选择作业投入运行时，先计算此时后备作业队列中每个作业的响应比RP。选择最大的作业投入运行。

5.优先权调度算法：按照进程的优先权大小来调度。使高优先权进程得到优先处理的调度策略称为优先权调度算法。注意：优先数越多，优先权越小。

6.多级队列调度算法：多队列调度是根据作业的性质和类型的不同，将就绪队列再分为若干个队列，所有的作业（进程）按其性质排入相应的队列中，而不同的就绪队列采用不同的调度算法。