

学习笔记

编写: xxx

I. 计算机网络

1.1. 基础

1.1.1. 网络协议模型

OSI 模型，即开放式通信系统互联参考模型(Open System Interconnection,OSI/RM,Open Systems Interconnection Reference Model)，是国际标准化组织(ISO)提出的一个试图使各种计算机在世界范围内互连为网络的标准框架，简称 OSI



OSI 网络模型是最为经典的网络模型，但是由于其结构过于复杂，我们常用的网络模型为 TCP/IP 模型，现在 TCP/IP 已经成为 Internet 上通信的工业标准，TCP/IP 总共有四层结构应用层、传输层、网络层和网络接口：

应用层：各种服务及应用程序通过该层利用网络，常用协议：HTTP, FTP, SMTP

传输层：确认数据传输进行纠错处理，常用协议：TCP UDP

网络层：负责数据传输、路径及地址选择，常用协议：IP ARP(地址解析协议)

网络接口：是针对不同物理网络的连接形式的协议：Ethernet

1.1.2. ARP 用来做什么？

在以太网中，一台主机要把数据帧发送到同一局域网上的另一台主机时，设备驱动程序必须知道以太网地址(MAC 地址)才能发送数据。而我们只知道 IP 地址，这时就需要采用 ARP 协议将 IP 地址映射为以太网地址

1.1.3. 子网掩码有什么用

子网掩码是一种用来指明一个 IP 地址所标示的主机处于哪个子网中。子网掩码不能单独存在，它必须结合 IP 地址一起使用。子网掩码只有一个作用，就是将某个 IP 地址划分成网络地址和主机地址两部分

当配合 IP 地址 172.20.0.4 使用时，则表示该 IP 地址是 B 类地址，172.20 是网络标识，0.4 是主机标识。也即子网掩码 AND IP 地址的部分是网络标识，子网掩码取反后 AND IP 地址的部分即为主机标识

1.1.4. 对称加密与非对称加密区别

一般分为两种，一种是可以对称的加密算法，现在大多用的是 AES 和 DES 等，因为不管服务端还是客户端都用的是一个相同的密钥所以可以说是对称加密，比如客户端用这个密钥给一段文字加密服务端收到这段字符串后会用同样的密钥进行解密；另外一种非对称加密用的多的就是 RSA，这个加密之所以叫非对称是因为客户端和服务器用的不是同样的密钥分为公钥和私钥。打个比方，甲方生成了一对密钥然后把公钥公开提供给乙方也可以是乙 1，乙 2，乙 3...，然后乙方拿着公钥进行加密，甲方拿着私钥进行解密。

1.1.5. dns 为什么用 udp

客户端向 DNS 服务器查询域名，一般返回的内容都不超过 512 字节，用 UDP 传输即可。不用经过 TCP 三次握手，这样 DNS 服务器负载更低，响应更快

1.2. IP

1.2.1. 地址分为几类

其中 A 类、B 类、和 C 类这三类地址用于 TCP/IP 节点，其它两类 D 类和 E 类被用于特殊用途。

A、B、C 三类 IP 地址的特征：当将 IP 地址写成二进制形式时，A 类地址的第一位总是 0，B 类地址的前两位总是 10，C 类地址的前三位总是 110。

1.2.2. 解决 ipv4 地址不足技术

1. NAT

我们一般使用私网 IP 作为局域网内部的主机标识，使用公网 IP 作为互联网上通信的标识，在整个 NAT 的转换中，最关键的流程有以下几点

网络被分为私网和公网两个部分，NAT 网关设置在私网到公网的路由出口位置，双向流量必须都要经过 NAT 网关

网络访问只能先由私网侧发起，公网无法主动访问私网主机；

NAT 网关在两个访问方向上完成两次地址的转换或翻译，出方向做源信息替换，入方向做目的信息替换；

NAT 网关的存在对通信双方是保持透明的；

NAT 网关为了实现双向翻译的功能，需要维护一张关联表，把会话的信息保存下来。

1.3. http

1.3.1. 常见的 HTTP 状态码：

1. 2XX——表明请求被正常处理了

200 OK：请求已正常处理。

204 No Content：请求处理成功，但没有任何资源可以返回给客户端，一般在只需要从客户端往服务器发送信息，而对客户端不需要发送新信息内容的情况下使用。

206 Partial Content：是对资源某一部分的请求，该状态码表示客户端进行了范围请求，而服务器成功执行了这部分的 GET 请求。响应报文中包含由 Content-Range 指定范围的实体内容。

2. 3XX——表明浏览器需要执行某些特殊的处理以正确处理请求

301 Moved Permanently：资源的 uri 已更新，你也更新下你的书签引用吧。永久性重定向，请求的资源已经被分配了新的 URI，以后应使用资源现在所指的 URI。

3. 4XX——表明客户端是发生错误的原因所在。

400 Bad Request：服务器端无法理解客户端发送的请求，请求报文中可能存在语法错误。

403 Forbidden：不允许访问那个资源。该状态码表明对请求资源的访问被服务器拒绝了。（权限，未授权 IP 等）

404 Not Found：服务器上没有请求的资源。路径错误等。

4. 5XX——服务器本身发生错误

1.3.2. http 网址访问过程

DNS 查找过程：

1. 浏览器会检查缓存中有没有这个域名对应的解析过的 IP 地址，如果缓存中有，这个解析过程就将结束。

2. 如果用户的浏览器缓存中没有，浏览器会查找操作系统缓存（hosts 文件）中是否有这个域名对应的 DNS 解析结果。

3. 若还没有，此时会发送一个数据包给 DNS 服务器，DNS 服务器找到后将解析所得

IP 地址返回给用户。

在应用层，浏览器会给 web 服务器发送一个 HTTP 请求；

请求头为：GET http://www.baidu.com/HTTP/1.1

在传输层，（上层的传输数据流分段）HTTP 数据包会嵌入在 TCP 报文段中；

TCP 报文段需要设置端口，接收方（百度）的 HTTP 端口默认是 80，本机的端口是一个 1024-65535 之间的随机整数，这里假设为 1025，这样 TCP 报文段由 TCP 首部（包含发送方和接收方的端口信息）+HTTP 数据包组成。

在网络层中，TCP 报文段再嵌入 IP 数据包中；

IP 数据包需要知道双方的 IP 地址，本机 IP 地址假定为 192.168.1.5，接受方 IP 地址为 220.181.111.147（百度），这样 IP 数据包由 IP 头部（IP 地址信息）+TCP 报文段组成。

在网络接口层，IP 数据包嵌入到数据帧（以太网数据包）中在网络上传送；

数据帧中包含源 MAC 地址和目的 MAC 地址（通过 ARP 地址解析协议得到的）。这样数据帧由头部（MAC 地址）+IP 数据包组成。

数据包经过多个网关的转发到达百度服务器，请求对应端口的服务；

服务接收到发送过来的以太网数据包开始解析请求信息，从以太网数据包中提取 IP 数据包>TCP 报文段>HTTP 数据包

请求处理完成之后，服务器发回一个 HTTP 响应；

响应头为：HTTP/1.1 200 OK

浏览器以同样的过程读取到 HTTP 响应的内容（HTTP 响应数据包），然后浏览器对收到的 HTML 页面进行解析，把网页显示出来呈现给用户。

客户端接收到返回数据，去掉对应头信息，形成也可以被浏览器认识的页面 HTML 字符串信息，交与浏览器翻译为对应页面规则信息展示为页面内容。

1.3.3. get 和 post 的区别

GET 和 POST 的请求都能使用额外的参数，但是 GET 的参数是以查询字符串出现在 URL 中，而 POST 的参数存储在实体主体部分。

GET 的传参方式相比于 POST 安全性较差，因为 GET 传的参数在 URL 是可见的，可能会泄露私密信息。并且 GET 只支持 ASCII 字符，如果参数为中文则可能会出现乱码，而 POST 支持标准字符集。

看过一个例子之后，我们再来看看如图 2-8 所示的一个请求报文的通用格式。我们看到该通用格式与我们前面的例子密切对应。然而，你可能已经注意到了在首部行（和附加的回车和换行）后有一个“实体体”（entity body）。使用 GET 方法时实体体为空，而使用 POST 方法时才使用该实体体。当用户提交表单时，HTTP 客户常常使用 POST 方法，例如当用户向搜索引擎提供搜索关键词时。使用 POST 报文时，用户仍可以向服务器请求一个 Web 页面，但 Web 页面的特定内容依赖于用户在表单字段中输入的内容。如果方法字段的值为 POST 时，则实体体中包含的就是用户在表单字段中的输入值。

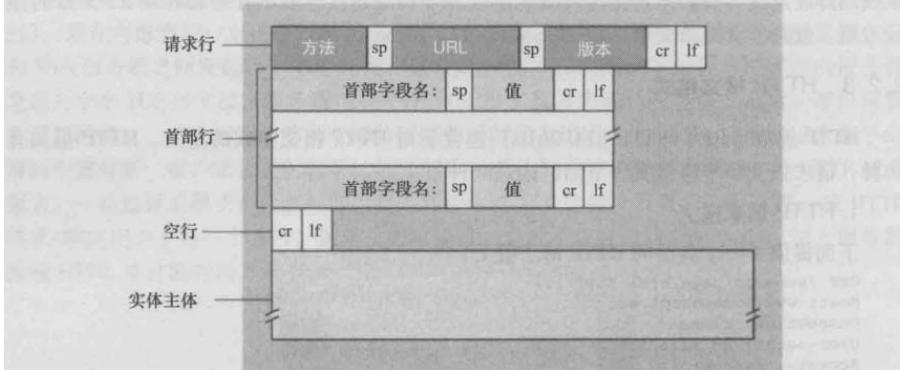


图 2-8 一个 HTTP 请求报文的通用格式

当然，如果不提“用表单生成的请求报文不是必须使用 POST 方法”这一点，那将是失职。HTML 表单经常使用 GET 方法，并在（表单字段中）所请求的 URL 中包括输入的数据。例如，一个表单使用 GET 方法，它有两个字段，分别填写的是“monkeys”和“bananas”，这样，该 URL 结构为 www.somesite.com/animalsearch?monkeys&bananas。在

1.3.4. http https 区别

HTTP 协议传输的数据都是未加密的，也就是明文的，因此使用 HTTP 协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了 SSL（Secure Sockets Layer）协议用于对 HTTP 协议传输的数据进行加密，从而就诞生了 HTTPS。

简单来说，HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，要比 http 协议安全。

HTTPS 和 HTTP 的区别主要如下：

- 1、https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用。
- 2、http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。
- 3、http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。

4、http 的连接很简单，是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全。

1.3.5. 如何解析 http 报头？

1. HTTP 请求报文由 3 部分组成（请求行+请求头+请求体）



图 15-4 HTTP 请求报文

2. HTTP 的响应报文也由三部分组成（响应行+响应头+响应体）



1.4. TCP

1.4.1. tcp 报头

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG
源端口																																
目标端口																																
TCP首部																																
序列号																																
确认号																																
TCP首部长度																																
保留																																
URG ACK PSH RST SYN FIN																																
校验和																																
选项 (0到40字节)																																
数据部分 (可选)																																
http://blog.csdn.net/wilsonpeng3																																

源端口、目标端口：计算机上的进程要和其他进程通信是要通过计算机端口的，而一个计算机端口某个时刻只能被一个进程占用，所以通过指定源端口和目标端口，就可以知道是哪两个进程需要通信。源端口、目标端口是用 16 位表示的，可推算计算机的端口个数为 2^{16} 个。

序列号：表示本报文段所发送数据的第一个字节的编号。在 TCP 连接中所传送的字节流的每一个字节都会按顺序编号。

确认号：表示接收方期望收到发送方下一个报文段的第一个字节数据的编号。

TCP 首部长度：由于 TCP 首部包含一个长度可变的选项部分，所以需要这么一个值来指定这个 TCP 报文段到底有多长。或者可以这么理解：就是表示 TCP 报文段中数据部分在整个 TCP 报文段中的位置。该字段的单位是 32 位字，即：4 个字节。

URG：表示本报文段中发送的数据是否包含紧急数据。URG=1，表示有紧急数据。后面的紧急指针字段只有当 URG=1 时才有效。

ACK：表示是否前面的确认号字段是否有效。ACK=1，表示有效。只有当 ACK=1 时，前面的确认号字段才有效。TCP 规定，连接建立后，ACK 必须为 1。

PSH：告诉对方收到该报文段后是否应该立即把数据推送给上层。如果为 1，则表示对方应当立即把数据提交给上层，而不是缓存起来。

RST：只有当 RST=1 时才有用。如果你收到一个 RST=1 的报文，说明你与主机的连接出现了严重错误（如主机崩溃），必须释放连接，然后再重新建立连接。或者说明你上次发送给主机的数据有问题，主机拒绝响应。

SYN：在建立连接时使用，用来同步序号。当 SYN=1，ACK=0 时，表示这是一个请求建立连接的报文段；当 SYN=1，ACK=1 时，表示对方同意建立连接。SYN=1，说明这是一个请求建立连接或同意建立连接的报文。只有在前两次握手时 SYN 才置为 1。

FIN：标记数据是否发送完毕。如果 FIN=1，就相当于告诉对方：“我的数据已经发送完毕，你可以释放连接了”

窗口大小：表示现在允许对方发送的数据量。也就是告诉对方，从本报文段的确认号开始允许对方发送的数据量。

校验和：提供额外的可靠性。具体如何校验，参考其他资料。

紧急指针：标记紧急数据在数据字段中的位置。

1.4.2. 三次握手四次挥手的状态字，3次/4次，状态转化图

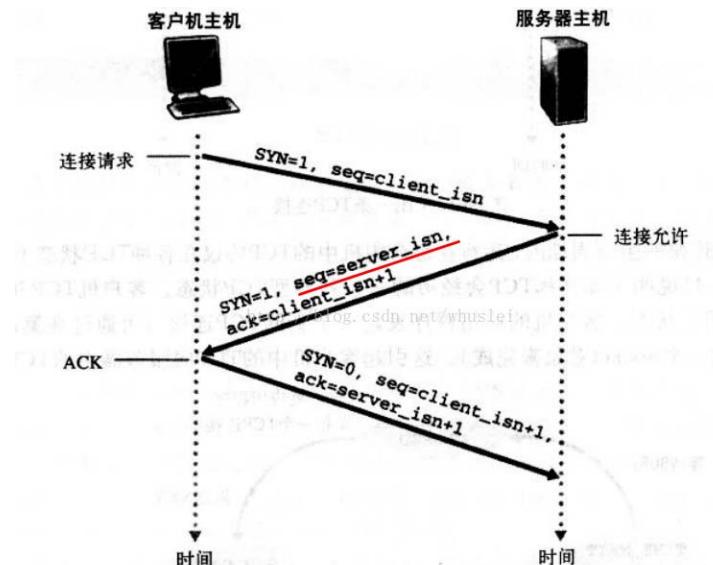
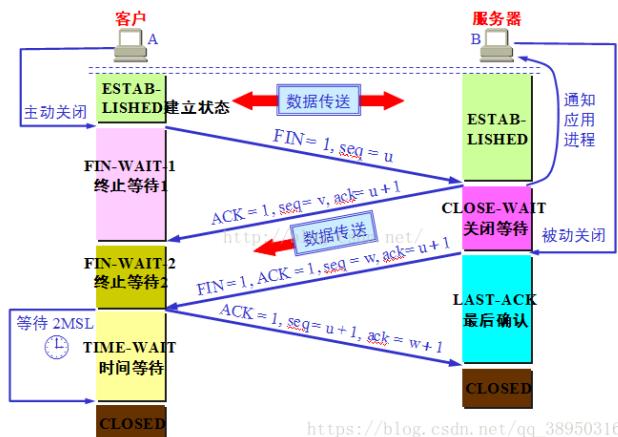
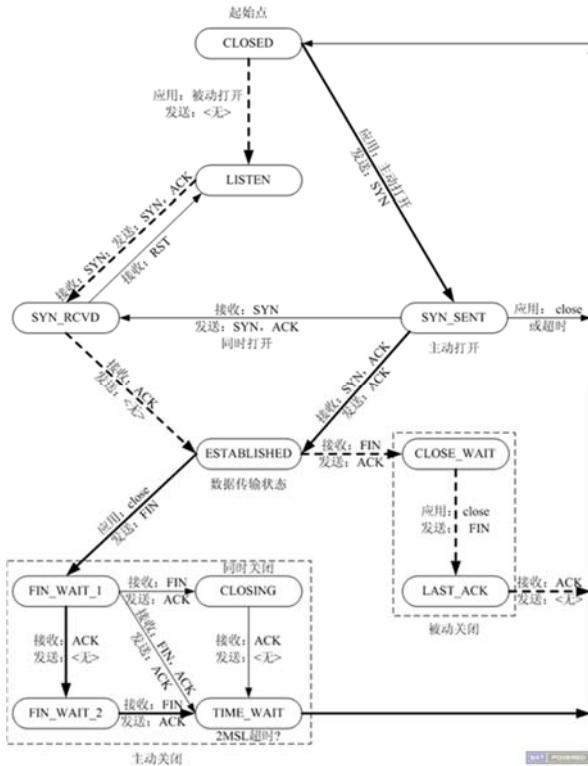


图3-39 TCP三次握手：报文段交换



https://blog.csdn.net/qq_38950316



1.4.3. 为什么连接的时候是三次握手，关闭的时候却是四次握手？

因为当 Server 端收到 Client 端的 SYN 连接请求报文后，可以直接发送 SYN+ACK 报文。其中 ACK 报文是用来应答的，SYN 报文是用来同步的。但是关闭连接时，当 Server 端收到 FIN 报文时，很可能并不会立即关闭 SOCKET，所以只能先回复一个 ACK 报文，告诉 Client 端，“你发的 FIN 报文我收到了”。只有等到我 Server 端所有的报文都发送完了，我才能发送 FIN 报文，因此不能一起发送。故需要四步握手。

1.4.4. 为什么有 TIME_WAIT

1. 为实现 TCP 全双工连接的可靠释放

由 TCP 状态变迁图可知，假设发起主动关闭的一方（client）最后发送的 ACK 在网络中丢失，由于 TCP 协议的重传机制，执行被动关闭的一方（server）将会重发其 FIN，在该 FIN 到达 client 之前，client 必须维护这条连接状态，也就是说这条 TCP 连接所对应的资源（client 方的 local_ip,local_port）不能被立即释放或重新分配，直到另一方重发的 FIN 达到之后，client 重发 ACK 后，经过 2MSL 时间周期没有再收到另一方的 FIN 之后，该 TCP 连接才能恢复初始的

CLOSED 状态。如果主动关闭一方不维护这样一个 TIME_WAIT 状态，那么当被动关闭一方重发的 FIN 到达时，主动关闭一方的 TCP 传输层会用 RST 包响应对方，这会被对方认为是有错误发生，然而这事实上只是正常的关闭连接过程，并非异常。

2. 为使旧的数据包在网络因过期而消失

为说明这个问题，我们先假设 TCP 协议中不存在 TIME_WAIT 状态的限制，再假设当前有一条 TCP 连接：(local_ip, local_port, remote_ip, remote_port)，因某些原因，我们先关闭，接着很快以相同的四元组建立一条新连接。本文前面介绍过，TCP 连接由四元组唯一标识，因此，在我们假设的情况下，TCP 协议栈是无法区分前后两条 TCP 连接的不同的，在它看来，这根本就是同一条连接，中间先释放再建立的过程对其来说是“感知”不到的。这样就可能发生这样的情况：前一条 TCP 连接由 local peer 发送的数据到达 remote peer 后，会被该 remote peer 的 TCP 传输层当做当前 TCP 连接的正常数据接收并向上传递至应用层（而事实上，在我们假设的场景下，这些旧数据到达 remote peer 前，旧连接已断开且一条由相同四元组构成的新 TCP 连接已建立，因此，这些旧数据是不应该被向上传递至应用层的），从而引起数据错乱进而导致各种无法预知的诡异现象。作为一种可靠的传输协议，TCP 必须在协议层面考虑并避免这种情况的发生，这正是 TIME_WAIT 状态存在的第 2 个原因。

1.4.5. TIME_WAIT 状态如何避免

首先服务器可以设置 SO_REUSEADDR 套接字选项来通知内核，如果端口忙，但 TCP 连接位于 TIME_WAIT 状态时可以重用端口。在一个非常有用的场景就是，如果你的服务器程序停止后想立即重启，而新的套接字依旧希望使用同一端口，此时 SO_REUSEADDR 选项就可以避免 TIME_WAIT 状态。

1.4.6. 服务器端不调用 accept 会发生什么

listen() 维护两个队列，一个是未完成三次握手的，一个是已完成三次握手的，accept() 是从已完成三次握手的队列中取出一个而已(backlog 指的就是已经完成握手了的队列的大小)

在被动状态的 socket 有两个队列，一个是正在进行三次握手的 socket 队列，一个是完成三次握手的 socket 队列。在握手完成后会从正在握手队列移到握手完成的队列，此时已经建立连接。accept 就是从已经完成三次握手的 socket 队列里面取，不 accept 客户端能完成的连接就是此队列的大小。

1.4.7. 大量的 CLOSE_WAIT 会怎么样？

在服务器与客户端通信过程中，因服务器发生了 socket 未关导致的

closed_wait 发生，最终造成配置的 port 被占满出现 socket.error: [Errno 24] Too many open files，原因是每个程序默认只能打开 1024 个文件描述符

1.4.8. TCP 和 UDP 区别

TCP(Transmission Control Protocol): 传输控制协议

UDP(User Datagram Protocol): 用户数据报协议

TCP 提供面向连接的、可靠的数据流传输；而 UDP 提供的是非面向连接的、不可靠的数据流传输。

TCP 传输单位称为 TCP 报文段；UDP 传输单位称为用户数据报。

TCP 注重数据安全性；UDP 数据传输快，因为不需要连接等待，少了许多操作，但是其安全性却一般。

TCP 对应的协议和 UDP 对应的协议：

FTP(File Transfer Protocol): 文件传输协议，使用 21 端口。

Telnet: 一种用于远程连接服务器的协议，它使用 23 端口。用户可以以自己的身份远程连接到服务器，可提供基于 DOS 模式下的通信服务。

SMTP(Simple Mail Transfer Protocol): 简单邮件传送协议。用于发送邮件。服务器开放的是 25 号端口。

POP3(Post Office Protocol - Version 3): 邮局协议版本 3。它和 SMTP 相对应，POP3 用于接收邮件。POP3 协议使用 110 端口。

HTTP((HyperText Transfer Protocol): 超文本传输协议。是从 Web 服务器传输超文本到本地浏览器的传输协议。

UDP 对应的协议：

DNS: 用于域名解析服务，将域名地址转换为 IP 地址。DNS 使用 53 端口。

SNMP: 简单网络管理协议，使用 161 号端口，是用来管理网络设备的。由于网络设备很多，无连接的服务就体现出其优势。

TFTP(Trival File Transfer Protocol)，简单文件传输协议，该协议在熟知端口 69 上使用 UDP 服务。

1.4.9. IP 和 32 位整数呈双射关系

```
1.  unsigned int  IPToValue(const string& strIP){  
2.      //inet_pton  
3.      int a[4];  
4.      string IP = strIP;  
5.      string strTemp;  
6.      size_t pos;  
7.      size_t i = 3;  
8.  
9.      do{  
10.         pos = IP.find(".");
```

```
11.
12.     if (pos != string::npos){
13.         strTemp = IP.substr(0, pos);
14.         a[i] = atoi(strTemp.c_str());
15.         i--;
16.         IP.erase(0, pos + 1);
17.     }
18.     else{
19.         strTemp = IP;
20.         a[i] = atoi(strTemp.c_str());
21.         break;
22.     }
23.
24. } while (1);
25.
26. unsigned int nResult = (a[3] << 24) + (a[2] << 16) + (a[1] << 8) + a[0];
27. return nResult;
28. }
29.
30. string ValueToIP(const int& nValue){
31. //inet_ntop
32.     char strTemp[20];
33.     sprintf(strTemp, "%d.%d.%d.%d",
34.             (nValue & 0xff000000) >> 24,
35.             (nValue & 0x00ff0000) >> 16,
36.             (nValue & 0x0000ff00) >> 8,
37.             (nValue & 0x000000ff));
38.
39.     return string(strTemp);
40. }
```

1.4.10. 服务端编程

1. 创建 socket
2. 调用 bind 函数，将 socket 和地址（包括 ip、port）绑定。
3. listen 监听，将接收到的客户端连接放入队列
4. 调用 accept 函数，从队列获取请求，返回 socket 描述符，如果无请求，将会阻塞，直到获得链接
5. 发送或接收数据 send recv
6. 关闭 accept 返回的 socket

1.4.11. TCP 的流量控制

所谓的流量控制就是让发送方的发送速率不要太快，让接收方来得及接受。利用滑动窗口机制可以很方便的在 TCP 连接上实现对发送方的流量控制。TCP 的窗口单位是字节，不是报文段，发送方的发送窗口不能超过接收方给出的接收窗口的数值。

1.4.12. 拥塞控制方法

拥塞控制的四种算法

1. 慢开始 Slow-start

TCP 在连接过程的三次握手完成后，开始传数据，并不是一开始向网络通道中发送大量的数据包，这样很容易导致网络中路由器缓存空间耗尽，从而发生拥塞；而是根据初始的 cwnd 大小逐步增加发送的数据量，cwnd 初始化为 1 个最大报文段(MSS)大小（这个值可配置不一定是 1 个 MSS）；每当有一个报文段被确认，cwnd 大小指数增长。

2. 拥塞避免 Congestion Avoidance

让拥塞窗口 cwnd 缓慢地增大，即每经过一个往返时间 RTT 就把发送方的拥塞窗口 cwnd 加 1，而不是加倍。这样拥塞窗口 cwnd 按线性规律缓慢增长，比慢开始算法的拥塞窗口增长速率缓慢得多

3. 快重传 Fast Retransmit

超时重传是 TCP 协议保证数据可靠性的一个重要机制，其原理是在发送一个数据以后就开启一个计时器，在一定时间内如果没有得到发送数据报的 ACK 报文，那么就重新发送数据，直到发送成功为止。这是数据包丢失的情况下给出的一种修补机制。一般来说，重传发生在超时之后，但是如果发送端接收到 3 个以上的重复 ACK，就应该意识到，数据丢了，需要重新传递。这个机制不需要等到重传定时器溢出，所以叫做快速重传，而快速重传以后，因为走的不是慢启动而是拥塞避免算法，所以这又叫做快速恢复算法。

快速重传和快速恢复算法一般同时使用。快速恢复算法是认为，你还有 3 个 Duplicated Ack 说明网络也不那么糟糕，所以没有必要像 RTO 超时那么强烈，并不需要重新回到慢启动进行，这样可能降低效率

4. 快恢复 Fast Recovery

当 TCP 收到 3 次重复的 ACK 时，将拥塞窗口减半，并在后续再收到重复的 ACK 时线性增加窗口。

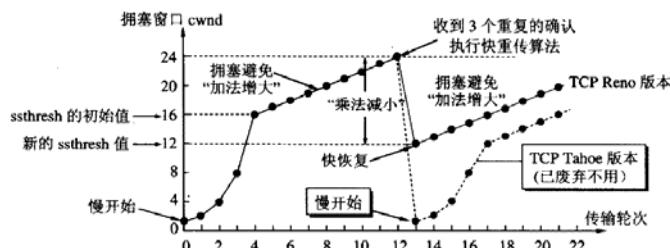


图 5-27 从连续收到三个重复的确认转入拥塞避免

<https://www.cnblogs.com/wxgblogs/p/5616829.html>

1.4.13. TCP 端口扫描方式

服务器上所开放的端口就是潜在的通信通道，也就是一个入侵通道。对目标计算机进行端口扫描，能得到许多有用的信息，进行端口扫描的方法很多，可以是手工进行扫描、也可以用端口扫描软件进行。扫描器通过选用远程 TCP/IP 不同的端口的服务，并记录目标给予的回答，通过这种方法可以搜集到很多关于目标主机的各种有用的信息，例如远程系统是否支持匿名登陆、是否存在可写的 FTP 目录、是否开放 TELNET 服务和 HTTPD 服务等

connect 扫描；

SYN 扫描；

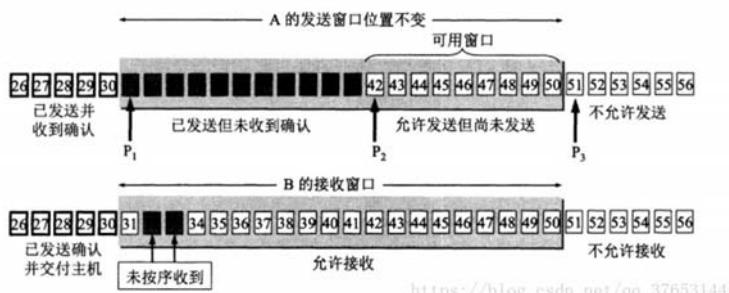
FIN 扫描；

1.4.14. TCP 可靠性怎么保证的？

校验和、序列号、确认应答、超时重传、连接管理、流量控制、拥塞控制

1.4.15. 滑动窗口

用于网络数据传输时的流量控制，以避免拥塞的发生。该协议允许发送方在停止并等待确认前发送多个数据分组。由于发送方不必每发一个分组就停下来等待确认，因此该协议可以加速数据的传输，提高网络吞吐量。



1.4.16. TCP 的粘包怎么解决？

UDP 是基于报文发送的，从 UDP 的帧结构可以看出，在 UDP 首部采用了 16bit 来指示 UDP 数据报文的长度，因此在应用层能很好的将不同的数据报文区分开，从而避免粘包和拆包的问题。而 TCP 是基于字节流的，虽然应用层和 TCP 传输层之间的数据交互是大小不等的数据块，但是 TCP 把这些数据块仅仅看成一连串无结构的字节流，没有边界；另外从 TCP 的帧结构也可以看出，在 TCP 的头部没有表示数据长度的字段，基于上面两点，在使用 TCP 传输数据时，

才有粘包或者拆包现象发生的可能

解决问题的关键在于如何给每个数据包添加边界信息，常用的方法有如下几个：

1、发送端给每个数据包添加包首部，首部中应该至少包含数据包的长度，这样接收端在接收到数据后，通过读取包首部的长度字段，便知道每一个数据包的实际长度了。

2、发送端将每个数据包封装为固定长度（不够的可以通过补 0 填充），这样接收端每次从接收缓冲区中读取固定长度的数据就自然而然的把每个数据包拆分开来。

3、可以在数据包之间设置边界，如添加特殊符号，这样，接收端通过这个边界就可以将不同的数据包拆分开

1.4.17. ping 会使用到哪些协议

在主机 A 上运行“Ping www.baidu.com”后，都发生了些什么呢？首先进行 DNS 查找过程，Ping 命令会构建一个固定格式的 ICMP 请求数据包，然后由 ICMP 协议将这个数据包连同获取的 ip 一起交给 IP 层协议（和 ICMP 一样，实际上是一组后台运行的进程），IP 层协议将以地址“192.168.0.5”作为目的地址，本机 IP 地址作为源地址，加上一些其他的控制信息，构建一个 IP 数据包，并想办法得到 192.168.0.5 的 MAC 地址（物理地址，这是数据链路层协议构建数据链路层的传输单元——帧所必需的），以便交给数据链路层构建一个数据帧。关键就在这里，IP 层协议通过机器 B 的 IP 地址和自己的子网掩码，发现它跟自己属同一网络，就直接在本网络内查找这台机器的 MAC，如果以前两机有过通信，在 A 机的 ARP 缓存表应该有 B 机 IP 与其 MAC 的映射关系，如果没有，就发一个 ARP 请求广播，得到 B 机的 MAC，一并交给数据链路层。后者构建一个数据帧，目的地址是 IP 层传过来的物理地址，源地址则是本机的物理地址，还要附加上一些控制信息，依据以太网的介质访问规则，将它们传送出去

1.5. select/poll/epoll

I/O 多路复用（multiplexing）的本质是通过一种机制（系统内核缓冲 I/O 数据），让单个进程可以监视多个文件描述符，一旦某个描述符就绪（一般是读就绪或写就绪），能够通知程序进行相应的读写操作，与多进程和多线程技术相比，I/O 多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销

select、poll、epoll 本质上也都是同步 I/O

select:

1> 每次调用 select，都需要把 fd_set 集合从用户态拷贝到内核态，如果

fd_set 集合很大时，那这个开销也很大

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

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

poll:

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

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

2> 获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核 IO 事件异步唤醒而加入 Ready 队列的描述符集合就行了

<https://www.jianshu.com/p/397449cadc9a>

epoll 中 ET 模式与 LT 模式的区别:

水平触发 (LT): 默认工作模式，即当 epoll_wait 检测到某描述符事件就绪并通知应用程序时，应用程序可以不立即处理该事件；下次调用 epoll_wait 时，会再次通知此事件

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

2. 操作系统

2.1. 文件系统

2.1.1. 什么叫软连接和硬链接，他们的区别是什么

1. 硬链接是有着相同 inode 号仅文件名不同的文件，因此硬链接存在以下几点特性：

文件有相同的 inode 及 data block;

只能对已存在的文件进行创建；

不能交叉文件系统进行硬链接的创建；

不能对目录进行创建，只可对文件创建；

删除一个硬链接文件并不影响其他有相同 inode 号的文件。

2. 软链接与硬链接不同，若文件用户数据块中存放的内容是另一文件的路径名的指向，则该文件就是软连接。软链接就是一个普通文件，只是数据块内容有点特殊。软链接有着自己的 inode 号以及用户数据块（见 图 2.）。因此软链接的创建与使用没有类似硬链接的诸多限制：

软链接有自己的文件属性及权限等；
可对不存在的文件或目录创建软链接；
软链接可交叉文件系统；
软链接可对文件或目录创建；
创建软链接时，链接计数 `i_nlink` 不会增加；

删除软链接并不影响被指向的文件，但若被指向的原文件被删除，则相关软连接被称为死链接（即 `dangling link`，若被指向路径文件被重新创建，死链接可恢复为正常的软链接）。

<https://www.ibm.com/developerworks/cn/linux/l-cn-hardandsymbolic-links/index.html>

2.2. 中断异常和系统调用

2.2.1. 用户态和内核态切换过程(`utlk146`)

1. 从当前进程的描述符中提取其内核栈的 `ss0` 及 `esp0` 信息。
2. 使用 `ss0` 和 `esp0` 指向的内核栈将当前进程的 `cs,eip,eflags,ss,esp` 信息保存起来，这个过程也完成了由用户栈到内核栈的切换过程，同时保存了被暂停执行的程序的下一条指令。
3. 将先前由中断向量检索得到的中断处理程序的 `cs,eip` 信息装入相应的寄存器，开始执行中断处理程序，这时就转到了内核态的程序执行了。

`esp` 寄存器是 CPU 栈指针，存放内核栈栈顶地址。在 X86 体系中，栈开始于末端，并朝内存区开始的方向增长。从用户态刚切换到内核态时，进程的内核栈总是空的，此时 `esp` 指向这个栈的顶端。

在 X86 中调用 `int` 指令型系统调用后会把用户栈的`%esp` 的值及相关寄存器压入内核栈中，系统调用通过 `iret` 指令返回，在返回之前会从内核栈弹出用户栈的`%esp` 和寄存器的状态，然后进行恢复。所以在进入内核态之前要保存进程的上下文，中断结束后恢复进程上下文，那靠的就是内核栈。

这里有个细节问题，就是要想在内核栈保存用户态的 `esp,eip` 等寄存器的值，首先得知道内核栈的栈指针，那在进入内核态之前，通过什么才能获得内核栈的栈指针呢？答案是：TSS

2.2.2. 系统调用和库函数有什么区别和联系

系统调用

1. 使用 INT 和 IRET 指令，内核和应用程序使用的是不同的堆栈，因此存在堆栈的切换，从用户态切换到内核态，从而可以使用特权指令操控设备
2. 依赖于内核，不保证移植性
3. 在用户空间和内核上下文环境间切换，开销较大
4. 是操作系统的一个入口点

函数调用

1. 使用 CALL 和 RET 指令，调用时没有堆栈切换
2. 平台移植性好
3. 属于过程调用，调用开销较小
4. 一个普通功能函数的调用

2.3. 死锁

1、死锁是怎么产生的

死锁产生的 4 个必要条件

互斥：某种资源一次只允许一个进程访问，即该资源一旦分配给某个进程，其他进程就不能再访问，直到该进程访问结束。

占有且等待：一个进程本身占有资源（一种或多种），同时还有资源未得到满足，正在等待其他进程释放该资源。

不可抢占：别人已经占有了某项资源，你不能因为自己也需要该资源，就去把别人的资源抢过来。

循环等待：存在一个进程链，使得每个进程都占有下一个进程所需的至少一种资源。

2、死锁预防：

资源一次性分配：（破坏请求和保持条件）

可剥夺资源：即当某进程新的资源未满足时，释放已占有的资源（破坏不可剥夺条件）

资源有序分配法：系统给每类资源赋予一个编号，每一个进程按编号递增的顺序请求资源，释放则相反（破坏环路等待条件）

3、死锁避免：

预防死锁的几种策略，会严重地损害系统性能。因此在避免死锁时，要施加较弱的限制，从而获得较满意的系统性能。由于在避免死锁的策略中，允许进程动态地申请资源。因而，系统在进行资源分配之前预先计算资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配给进程；否则，进

程等待。其中最具有代表性的避免死锁算法是银行家算法。

银行家算法是从当前状态出发，按照系统各类资源剩余量逐个检查各进程需要申请的资源量，找到一个各类资源申请量均小于等于系统剩余资源量的进程 P1。然后分配给该 P1 进程所请求的资源，假定 P1 完成工作后归还其占有的所有资源，更新系统剩余资源状态并且移除进程列表中的 P1，进而检查下一个能完成工作的客户，……。如果所有客户都能完成工作，则找到一个安全序列，银行家才是安全的。若找不到这样的安全序列，则当前状态不安全。

4、死锁检测：

通过将资源分配图简化的方法来检测系统状态 S 是否为死锁状态，当发现有进程死锁后，便应立即把它从死锁状态中解脱出来，常采用的方法有：

剥夺资源：从其它进程剥夺足够数量的资源给死锁进程，以解除死锁状态。

撤消进程：可以直接撤消死锁进程或撤消代价最小的进程，直至有足够的资源可用，死锁状态消除为止

2.4. 信号

kill -1 可以查看

SIGINT:来自键盘的终端(ctrl+c)

SIGCHLD:子进程结束时发送到父进程的信号

SIGSEGV:无效内存引用

SIGKILL

SIGSTOP

作用：1> 让进程知道发生了一个特定的事件 2> 强迫进程执行它代码中的信号处理程序

进程对信号的三种响应：

- 1)忽略信号
- 2)捕获并处理信号
- 3)执行默认操作

注意：SIGKILL 和 SIGSTOP 不能被显示的忽略、捕获或阻塞

用户可以提供自己的信号处理函数，然后使用 signal 函数将处理函数加载，函数原型

```
#include <signal.h>
void (*signal (int signo,void (*func)(int)))(int);
```

signo 表示信号值，func 表示一个函数的指针，用来捕获指定的信号。func 可去如下值之一：

SIG_IGN 忽略该信号

SIG_DFL 使用系统默认的方式处理

SIG_ERR

signal 函数的返回值也会是一个函数的指针，这个指针执行上一次的信号处理程序。如果出错，返回 SIG_ERR

使用 kill 想进程或进程组发送信号，函数原型如下：

```
#include <signal.h>
int kill(pid_t pid, int signo);
```

pid>0 将信号发送给进程 id 为 pid 的进程

pid==0 将此信号发送给进程组 id 和该进程相同的进程

pid<0 将此信号发送给进程组内的进程 id 为 pid 的进程

pid=-1 将此信号发给系统的所有进程

2.5. 进程

2.5.1. 进程地址空间

进程的地址空间 (*address space*) 由允许进程使用的全部线性地址组成。每个进程所看到的线性地址集合是不同的，一个进程所使用的地址与另外一个进程所使用的地址之间没有关系。后面我们会看到，内核可以通过增加或删除某些线性地址区间来动态地修改进程的地址空间。

内核通过所谓线性区的资源来表示线性地址区间，线性区是由起始线性地址、长度和一些访问权限来描述的。为了效率起见，起始地址和线性区的长度都必须是 4096 的倍数，以便每个线性区所识别的数据完全填满分配给它的页框。下面是进程获得新线性区的一些典型情况：

- 当用户在控制台输入一条命令时，shell 进程创建一个新的进程去执行这个命令。结果是，一个全新的地址空间（也就是一组线性区）分配给了新进程（参见本章后面的“创建和删除进程的地址空间”一节和第二十章）。
- 正在运行的进程有可能决定装入一个完全不同的程序。在这种情况下，进程标识符仍然保持不变，可是在装入这个程序以前所使用的线性区却被释放，并有一组新的线性区被分配给这个进程（参见第二十章中的“exec 函数”一节）。
- 正在运行的进程可能对一个文件（或它的一部分）执行“内存映射”。在这种情况下，内核给这个进程分配一个新的线性区来映射这个文件（参见第十六章中的“内存映射”一节）。
- 进程可能持续向它的用户态堆栈增加数据，直到映射这个堆栈的线性区用完为止。

在这种情况下，内核也许会决定扩展这个线性区的大小（参见本章后面的“缺页异常处理程序”一节）。

- 进程可能创建一个IPC共享线性区来与其他合作进程共享数据。在这种情况下，内核给这个进程分配一个新的线性区以实现这个方案（参见第十九章中的“IPC共享内存”一节）。
- 进程可能通过调用类似malloc()这样的函数扩展自己的动态区（堆）。结果是，内核可能决定扩展给这个堆所分配的线性区（参见本章后面的“堆的管理”一节）。

表9-1显示了与前面提到的任务相关的一些系统调用。除brk()在本章的最后进行讨论外，其余的系统调用在其他章节阐述。

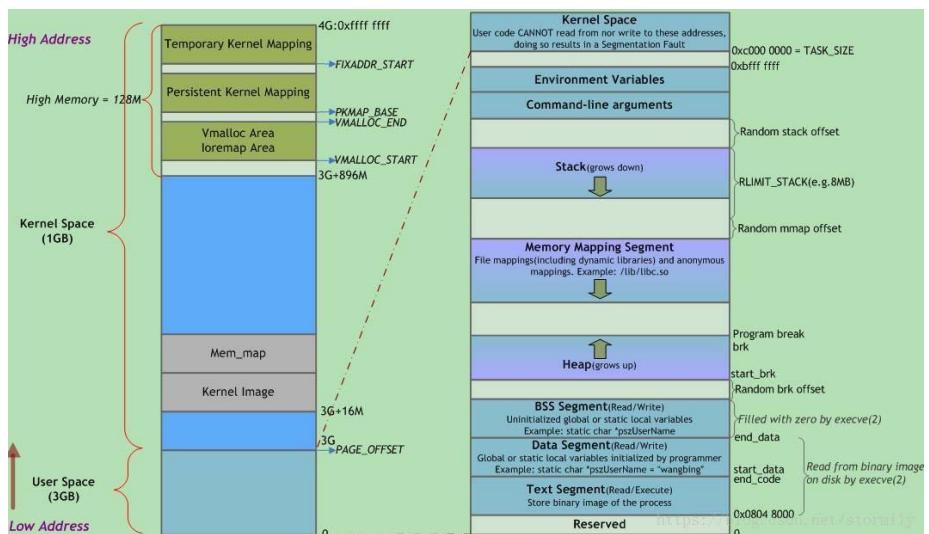
表9-1：与创建、删除线性区相关的系统调用

系统调用	说明
brk()	改变进程堆的大小
execve()	装入一个新的可执行文件，从而改变进程的地址空间
_exit()	结束当前进程并撤销它的地址空间
fork()	创建一个新进程，并为它创建新的地址空间
mmap(), mmap2()	为文件创建一个内存映射，从而扩大进程的地址空间
mremap()	扩大或缩小线性区
remap_file_pages()	为文件创建非线性映射（参见第十六章）
munmap()	撤销对文件的内存映射，从而缩小进程的地址空间
shmat()	创建一个共享线性区
shmrdt()	撤消一个共享线性区

内存区域可以包含各种内存对象，比如：

- 可执行文件代码的内存映射，称为代码段（text section）。
- 可执行文件的已初始化全局变量的内存映射，称为数据段（data section）。

- 包含未初始化全局变量，也就是 bss 段^②的零页（页面中的信息全部为 0 值，所以可用于映射 bss 段等目的）的内存映射。
- 用于进程用户空间栈（不要和进程内核栈混淆，进程的内核栈独立存在并由内核维护）的零页的内存映射。
- 每一个诸如 C 库或动态连接程序等共享库的代码段、数据段和 bss 也会被载入进程的地址空间。
- 任何内存映射文件。
- 任何共享内存段。
- 任何匿名的内存映射，比如由 malloc()^③分配的内存。



2.5.2. fork

fork()系统调用会通过复制一个现有进程来创建一个全新的进程。进程被存放在一个叫做任务队列的双向循环链表当中。链表当中的每一项都是类型为

task_struct 成为进程描述符的结构.也就是我们写过的进程 PCB。

像，那么所有的拷贝都将前功尽弃。Linux 的 fork() 使用写时拷贝（copy-on-write）页实现。写时拷贝是一种可以推迟甚至免除拷贝数据的技术。内核此时并不复制整个进程地址空间，而是让父进程和子进程共享同一个拷贝。

只有在需要写入的时候，数据才会被复制，从而使各个进程拥有各自的拷贝。也就是说，资源的复制只有在需要写入的时候才进行，在此之前，只是以只读方式共享。这种技术使地址空间上的页的拷贝被推迟到实际发生写入的时候才进行。在页根本不会被写入的情况下（举例来说，fork() 后立即调用 exec()）它们就无须复制了。

fork() 的实际开销就是复制父进程的页表以及给子进程创建唯一的进程描述符。在一般情况下，进程创建后都会马上运行一个可执行的文件，这种优化可以避免拷贝大量根本就不会被使用的数据（地址空间里常常包含数十兆的数据）。由于 Unix 强调进程快速执行的能力，所以这个优化是很重要的。

与 vfork 区别：

1. fork 父子进程交替运行，vfork 保证子进程先运行，父进程阻塞，直到子进程结束（或子进程调用了 exec 或 exit）
2. fork 实现了写时拷贝. 而 vfork 直接让父子进程共用公用资源，避免多开辟空间拷贝

- 1) 在父进程中，fork 返回新创建子进程的进程 ID;
- 2) 在子进程中，fork 返回 0;
- 3) 如果出现错误，fork 返回一个负值;

应用：

在 fork 函数执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork 函数返回 0，在父进程中，

fork 返回新创建子进程的进程 ID。我们可以通过 fork 返回的值来判断当前进程是子进程还是父进程。

```
#include <unistd.h>
#include <stdio.h>
int main ()
{
    pid_t fpid; //fpid表示fork函数返回的值
    int count=0;
    fpid=fork();
    if (fpid < 0)
        printf("error in fork!");
    else if (fpid == 0) {
        printf("i am the child process, my process id is %d/n",getpid());
        printf("我是爹的儿子/n");//对某些人来说中文看着更直白。
        count++;
    }
    else {
        printf("i am the parent process, my process id is %d/n",getpid());
        printf("我是孩子他爹/n");
        count++;
    }
    printf("统计结果是: %d/n",count);
    return 0;
}
```

2.5.3. Copy on write

写入时复制（英语：Copy-on-write，简称 COW）是一种计算机程序设计领域的优化策略。其核心思想是，如果有多个调用者（callers）同时请求相同资源（如内存或磁盘上的数据存储），他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源的内容时，系统才会真正复制一份专用副本（private copy）给该调用者，而其他调用者所见到的最初的资源仍然保持不变。这过程对其他的调用者都是透明的（transparently）。

Linux 的 fork() 使用写时拷贝（copy-on-write）页实现。写时拷贝是一种可以推迟甚至免除拷贝数据的技术。内核此时并不复制整个进程地址空间，而是让父进程和子进程共享同一个拷贝。只有在需要写入的时候，数据才会被复制，从而使各个进程拥有各自的拷贝。也就是说，资源的复制只有在需要写入的时候才进行，在此之前，只是以只读方式共享。

2.5.4. 进程同步

在 Linux 下，进程同步的解决方式主要有四种：

1. 信号量

2. 文件锁
3. 无锁 CAS
4. 校验方式 (CRC32 校验)

<https://blog.csdn.net/okiwilldoit/article/details/78487507>

2.5.5. 进程和线程的区别

- 1、进程是运行中的程序，线程是进程的内部的一个执行序列
- 2、进程是资源分配的单元，线程是执行行单元
- 3、进程间切换代价大，线程间切换代价小（地址空间）
- 4、进程拥有资源多，线程拥有资源少
- 5、多个线程共享进程的资源

2.5.6. 线程间同步的方式

原子操作:

避免由于“读－修改－写”指令引起的竞争条件的最容易的办法，就是确保这样的操作在芯片级是原子的。任何一个这样的操作都必须以单个指令执行，中间不能中断，且避免其他的CPU访问同一存储器单元。这些很小的原子操作 (*atomic operations*) 可以建立在其他更灵活机制的基础之上以创建临界区。

互斥锁(mutex):

互斥量是最简单的同步机制，即互斥锁。多个进程(线程)均可以访问到一个互斥量，通过对互斥量加锁，从而来保护一个临界区，防止其它进程(线程)同时进入临界区，保护临界资源互斥访问。

自旋锁(spin lock):

自旋锁 (spin lock) 是用来在多处理器环境中工作的一种特殊的锁。如果内核控制路径发现自旋锁“开着”，就获取锁并继续自己的执行。相反，如果内核控制路径发现锁由运行在另一个CPU上的内核控制路径“锁着”，就在周围“旋转”，反复执行一条紧凑的循环指令，直到锁被释放。

自旋锁的循环指令表示“忙等”。即使等待的内核控制路径无事可做（除了浪费时间），它也在CPU上保持运行。不过，自旋锁通常非常方便，因为很多内核资源只锁1毫秒的时间片段；所以说，释放CPU和随后又获得CPU都不会消耗多少时间。

一般来说，由自旋锁所保护的每个临界区都是禁止内核抢占的。在单处理器系统上，这种锁本身并不起锁的作用，自旋锁原语仅仅是禁止或启用内核抢占。请注意，在自旋锁忙等期间，内核抢占还是有效的，因此，等待自旋锁释放的进程有可能被更高优先级的进程替代。

读写锁:

读写锁适合于使用在读操作多，写操作少的情况，比如数据库。读写锁读锁可以同时加很多，但是写锁是互斥的。当有进程或者线程要写时，必须等待所有的读进程或者线程都释放自己的读锁方可以写。数据库很多时候可能只是做一些查询。

条件变量(cond):

互斥锁不同，条件变量是用来等待而不是用来上锁的。条件变量用来自动阻塞一个线程，直到某特殊情况发生为止。通常条件变量和互斥锁同时使用。条件变量分为两部分：条件和变量。条件本身是由互斥量保护的。线程在改变条件状态前先要锁住互斥量。条件变量使我们可以睡眠等待某种条件出现。

信号量(sem):

如同进程一样，线程也可以通过信号量来实现通信，虽然是轻量级的。信号量函数的名字都以"sem_"打头。

2.5.7. Linux 下有哪些原子操作

atomic_read(v)	返回 *v
atomic_set(v, i)	把 *v 置成 i
atomic_add(i, v)	给 *v 增加 i
atomic_sub(i, v)	从 *v 中减去 i
atomic_sub_and_test(i, v)	从 *v 中减去 i，如果结果为 0，则返回 1；否则，返回 0
atomic_inc(v)	把 1 加到 *v
atomic_dec(v)	从 *v 减 1
atomic_dec_and_test(v)	从 *v 减 1，如果结果为 0，则返回 1；否则，返回 0
atomic_inc_and_test(v)	把 1 加到 *v，如果结果为 0，则返回 1；否则，返回 0
atomic_add_negative(i, v)	把 i 加到 *v，如果结果为负，则返回 1；否则，返回 0
atomic_inc_return(v)	把 1 加到 *v，返回 *v 的新值
atomic_dec_return(v)	从 *v 减 1，返回 *v 的新值
atomic_add_return(i, v)	把 i 加到 *v，返回 *v 的新值
atomic_sub_return(i, v)	从 *v 减 i，返回 *v 的新值

另一类原子函数操作作用于位掩码（参见表 5-5）。

表5-5：Linux中的原子位处理函数

函数	说明
test_bit(nr, addr)	返回 *addr 的第 nr 位的值
set_bit(nr, addr)	设置 *addr 的第 nr 位
clear_bit(nr, addr)	清 *addr 的第 nr 位
change_bit(nr, addr)	转换 *addr 的第 nr 位
test_and_set_bit(nr, addr)	设置 *addr 的第 nr 位，并返回它的原值
test_and_clear_bit(nr, addr)	清 *addr 的第 nr 位，并返回它的原值
test_and_change_bit(nr, addr)	转换 *addr 的第 nr 位，并返回它的原值

2.5.8. 乐观锁/悲观锁

1. 乐观锁

乐观锁(Optimistic Lock)，顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于 write_condition 机制的其实都是提供的乐观锁

2. 悲观锁

悲观锁(Pessimistic Lock)，顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会 block 直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行 retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适

2.5.9. 多线程和多进程的优缺点

多线程的优点：

无需跨进程边界； 程序逻辑和控制方式简单； 所有线程可以直接共享内存和变量等； 线程方式消耗的总资源比进程方式好；

线程缺点：

线程之间的同步和加锁控制比较麻烦； 一个线程的崩溃可能影响到整个程

序的稳定性；到达一定的线程数程度后，即使再增加 CPU 也无法提高性能，例如 Windows Server 2003，大约是 1500 个左右的线程数就快到极限了（线程堆栈设定为 1M），如果设定线程堆栈为 2M，还达不到 1500 个线程总数；线程能够提高的总性能有限，而且线程多了之后，线程本身的调度也是一个麻烦事儿，需要消耗较多的 CPU

多进程优点：

每个进程互相独立，不影响主程序的稳定性，子进程崩溃没关系；通过增加 CPU，就可以容易扩充性能；可以尽量减少线程加锁/解锁的影响，极大提高性能，就算是线程运行的模块算法效率低也没关系；每个子进程都有 2GB 地址空间和相关资源，总体能够达到的性能上限非常大

多线程缺点：逻辑控制复杂，需要和主程序交互；需要跨进程边界，如果有大数据量传送，就不太好，适合小数据量传送、密集运算 多进程调度开销比较大

最好是多进程和多线程结合，即根据实际的需要，每个 CPU 开启一个子进程，这个子进程开启多线程可以为若干同类型的数据进行处理。当然你也可以利用多线程+多 CPU+轮询方式来解决问题

2.5.10. 孤儿进程、僵尸进程和守护进程

孤儿进程：

一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被 init 进程(进程号为 1)所收养，并由 init 进程对它们完成状态收集工作。

僵尸进程：

一个进程使用 fork 创建子进程，如果子进程退出，而父进程并没有调用 wait 或 waitpid 获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程。

unix 提供了一种机制可以保证只要父进程想知道子进程结束时的状态信息，就可以得到。这种机制就是：在每个进程退出的时候，内核释放该进程所有的资源，包括打开的文件，占用的内存等。但是仍然为其保留一定的信息(包括进程号 the process ID, 退出状态 the termination status of the process, 运行时间 the amount of CPU time taken by the process 等)。直到父进程通过 wait / waitpid 来取时才释放。但这样就导致了问题，如果进程不调用 wait / waitpid 的话，那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵死进程，将因为没有可用的进程号而导致系统不能产生新的进程。此即为僵尸进程的危害，应当避免。

每当出现一个孤儿进程的时候，内核就把孤儿进程的父进程设置为 init，而 init 进程会循环地 wait() 它的已经退出的子进程。这样，当一个孤儿进程凄凉地结束了其生命周期的时候，init 进程就会代表党和政府出面处理它的一切善后工作。因此孤儿进程并不会有什么危害任何一个子进程(init 除外)在 exit() 之后，并非马上就消失掉，而是留下一个称为僵尸进程(Zombie)的数据结构，等待父进程处理。这是每个子进程在结束时都要经过的阶段。如果子进程在 exit() 之后，父进程没有来得及处理，这时用 ps 命令就能看到子进程的状态是“Z”。如果父进程能及时 处理，可能用 ps 命令就来不及看到子进程的僵尸状态，但这并不等于子进程不经过僵尸状态。如果父进程在子进程结束之前退出，则子进程将由 init 接管。init 将会以父进程的身份对僵尸状态的子进程进行处理

守护进程：

由于在 Linux 中，每一个系统与用户进行交流的界面称为终端，每一个从此终端开始运行的进程都会依附于这个终端，这个终端就称为这些进程的控制终端，当控制终端被关闭时，相应的进程都会自动关闭。但是守护进程却能够突破这种限制，它从被执行开始运转，直到整个系统关闭时才退出。如果想让某个进程不因为用户或终端或其他地变化而受到影响，那么就必须把这个进程变成一个守护进程。

2.5.11. 调度算法

每个 Linux 进程总是按照下面的调度类型被调度：

SCHED_FIFO

先进先出的实时进程。当调度程序把 CPU 分配给进程的时候，它把该进程描述符保留在运行队列链表的当前位置。如果没有其他可运行的更高优先级实时进程，进程就继续使用CPU，想用多久就用多久，即使还有其他具有相同优先级的实时进程处于可运行状态。

SCHED_RR

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

SCHED_NORMAL

普通的分时进程。

调度算法根据进程是普通进程还是实时进程而有很大不同。

普通进程的调度

每个普通进程都有它自己的静态优先级，调度程序使用静态优先级来估价系统中这个进程与其他普通进程之间调度的程度。内核用从 100（最高优先级）到 139（最低优先级）的数表示普通进程的静态优先级。注意，值越大静态优先级越低。

新进程总是继承其父进程的静态优先级。不过，通过把某些“nice 值”传递给系统调用 nice() 和 setpriority()（参见本章稍后“与调度相关的系统调用”一节），用户可以改变自己拥有的进程的静态优先级。

实时进程的调度

每个实时进程都与一个实时优先级相关，实时优先级是一个范围从 1（最高优先级）~ 99（最低优先级）的值。调度程序总是让优先级高的进程运行，换句话说，实时进程运行的过程中，禁止低优先级进程的执行。与普通进程相反，实时进程总是被当成活动进程（参见上一节）。用户可以通过系统调用 sched_setparam() 和 sched_setscheduler() 改变进程的实时优先级（参见本章稍后“与调度相关的系统调用”一节）。

Linux 调度的总体思想是：实时进程优先于普通进程，实时进程以进程的紧急程度为优先顺序，并为实时进程赋予固定的优先级；普通进程则以保证所有进程能平均占用处理器时间为原则。所以其具体做法就是：

2.6. 内存管理

2.6.1. 逻辑地址到物理地址过程/分段分页机制

逻辑地址：

每个逻辑地址包括两个部分，段和偏移量。

线性地址：

也通常称为虚拟地址，在 32 位系统中最大可达 4G。

物理地址：

物理内存上的地址。

1. 逻辑地址到线性地址的转换

段选择符和段寄存器

一个逻辑地址由两部分组成：一个段标识符和一个指定段内相对地址的偏移量。段标识符是一个 16 位长的字段，称为段选择符（*Segment Selector*）如图 2-2 所示，而偏移量是一个 32 位长的字段。我们将在本章“快速访问段描述符”一节描述段选择符字段。

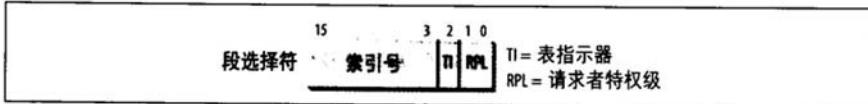


图 2-2：段选择符格式

为了快速方便地找到段选择符，处理器提供段寄存器，段寄存器的唯一目的是存放段选择符。这些段寄存器称为 cs, ss, ds, es, fs 和 gs。尽管只有 6 个段寄存器，但程序可以把同一个段寄存器用于不同的目的，方法是先将其值保存在内存中，用完后再恢复。

6 个寄存器中 3 个有专门的用途：

- cs 代码段寄存器，指向包含程序指令的段。
- ss 栈段寄存器，指向包含当前程序栈的段。
- ds 数据段寄存器，指向包含静态数据或者全局数据段。

其他 3 个段寄存器作一般用途，可以指向任意的数据段。

cs 寄存器还有一个很重要的功能：它含有一个两位的字段，用以指明 CPU 的当前特权级（Current Privilege Level, CPL）。值为 0 代表最高优先级，而值为 3 代表最低优先级。Linux 只用 0 级和 3 级，分别称之为内核态和用户态。

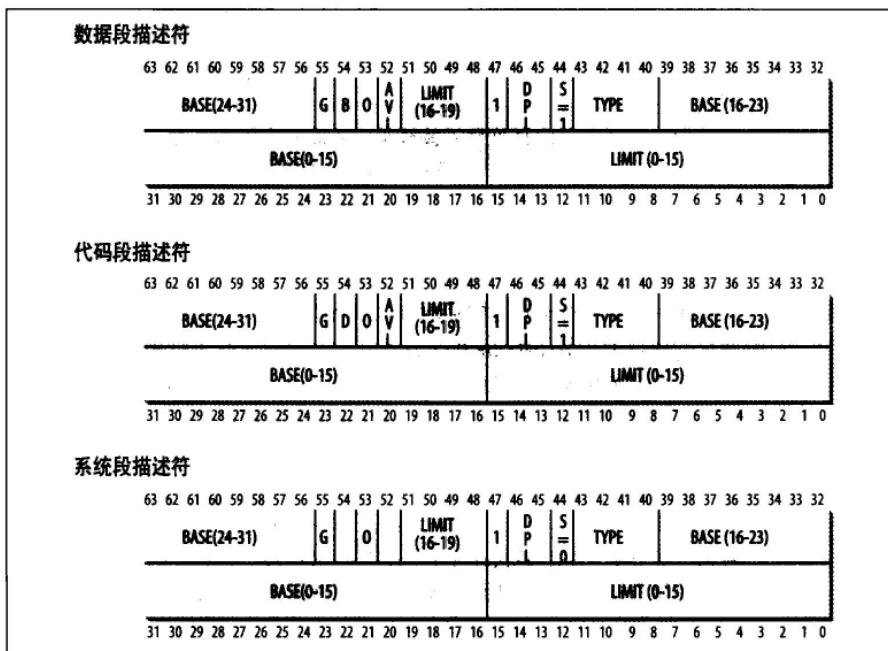


图 2-3：段描述符格式

图2-5详细显示了一个逻辑地址是怎样转换成相应的线性地址的。分段单元(*segmentation unit*)执行以下操作：

- 先检查段选择符的TI字段，以决定段描述符保存在哪一个描述符表中。TI字段指明描述符是在GDT中(在这种情况下，分段单元从gdtr寄存器中得到GDT的线性地址)还是在激活的LDT中(在这种情况下，分段单元从ldtr寄存器中得到LDT的线性地址)。
- 从段选择符的index字段计算段描述符的地址，index字段的值乘以8(一个段描述符的大小)，这个结果与gdtr或ldtr寄存器中的内容相加。
- 把逻辑地址的偏移量与段描述符Base字段的值相加就得到了线性地址。

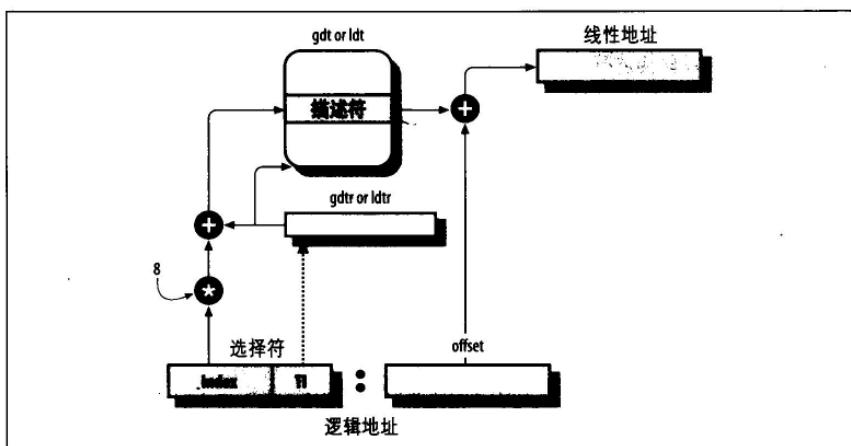


图2-5：逻辑地址的转换

2. 线性地址到物理地址的转换

Linux 中的分页

Linux 采用了一种同时适用于 32 位和 64 位系统的普通分页模型。正像前面“64 位系统中的分页”一节所解释的那样，两级页表对 32 位系统来说已经足够了，但 64 位系统需要更多数量的分页级别。直到 2.6.10 版本，Linux 采用三级分页的模型。从 2.6.11 版本开始，采用了四级分页模型（注 5）。图 2-12 中展示的 4 种页表分别被为：

- 页全局目录 (Page Global Directory)
- 页上级目录 (Page Upper Directory)
- 页中间目录 (Page Middle Directory)
- 页表 (Page Table)

页全局目录包含若干页上级目录的地址，页上级目录又依次包含若干页中间目录的地址，而页中间目录又包含若干页表的地址。每一个页表项指向一个页框。线性地址因此被分成五个部分。图 2-12 没有显示位数，因为每一部分的大小与具体的计算机体系结构有关。

对于没有启用物理地址扩展的 32 位系统，两级页表已经足够了。Linux 通过使“页上级目录”位和“页中间目录”位全为 0，从根本上取消了页上级目录和页中间目录字段。不过，页上级目录和页中间目录在指针序列中的位置被保留，以便同样的代码在 32 位系统和 64 位系统下都能使用。内核为页上级目录和页中间目录保留了一个位置，这是通过把它们的页目录项数设置为 1，并把这两个目录项映射到页全局目录的一个适当的目录项而实现的。

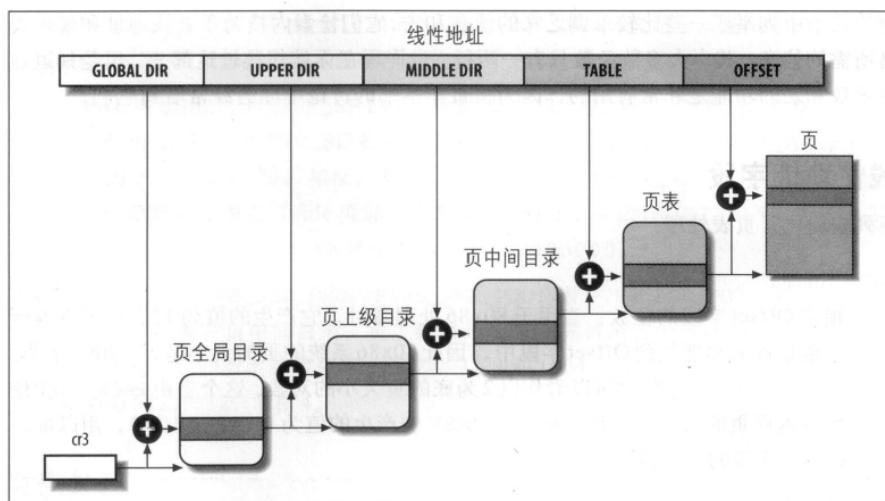


图 2-12：Linux 分页模式

启用了物理地址扩展的32位系统使用了三级页表。Linux的页全局目录对应80x86的页目录指针表（PDPT），取消了页上级目录，页中间目录对应80x86的页目录，Linux的页表对应80x86的页表。

最后，64位系统使用三级还是四级分页取决于硬件对线性地址的位的划分（见表2-4）。

Linux的进程处理很大程度上依赖于分页。事实上，线性地址到物理地址的自动转换使下面的设计目标变得可行：

- 给每一个进程分配一块不同的物理地址空间，这确保了可以有效地防止寻址错误。
- 区别页（即一组数据）和页框（即主存中的物理地址）之不同。这就允许存放在某个页框中的一个页，然后保存到磁盘上，以后重新装入这同一页时又可以被装在不同的页框中。这就是虚拟内存机制的基本要素（参见第十七章）。

在本章剩余的部分，为了具体起见，我们将涉及80x86处理器使用的分页机制。

我们将在第九章看到，每一个进程有它自己的页全局目录和自己的页表集。当发生进程切换时（参见第三章“进程切换”一节），Linux把cr3控制寄存器的内容保存在前一个执行进程的描述符中，然后把下一个要执行进程的描述符的值装入cr3寄存器中。因此，当新进程重新开始在CPU上执行时，分页单元指向一组正确的页表。

2.6.2. Linux 内核地址空间

内存管理区

在一个理想的计算机体系结构中，一个页框就是一个内存存储单元，可用于任何事情：存放内核数据和用户数据、缓冲磁盘数据等等。任何种类的数据页都可以存放在任何页框中，没有什么限制。

但是，实际的计算机体系结构有硬件的制约，这限制了页框可以使用的方式。尤其是，Linux 内核必须处理 80x86 体系结构的两种硬件约束：

- ISA 总线的直接内存存取 (DMA) 处理器有一个严格的限制：它们只能对 RAM 的前 16MB 寻址。
- 在具有大容量 RAM 的现代 32 位计算机中，CPU 不能直接访问所有的物理内存，因为线性地址空间太小。

为了应对这两种限制，Linux 2.6 把每个内存节点的物理内存划分为 3 个管理区 (zone)。在 80x86 UMA 体系结构中的管理区为：

ZONE_DMA

包含低于 16 MB 的内存页框

ZONE_NORMAL

包含高于 16 MB 且低于 896 MB 的内存页框

ZONE_HIGHMEM

包含从 896MB 开始高于 896 MB 的内存页框

ZONE_DMA 和 ZONE_NORMAL 区包含内存的“常规”页框，通过把它们线性地映射到线性地址空间的第 4 个 GB，内核就可以直接进行访问（参见第二章的“内核页表”一节）。相反，ZONE_HIGHMEM 区包含的内存页不能由内核直接访问，尽管它们也线性地映射到了线性地址空间的第 4 个 GB（参见本章后面“高端内存页框的内核映射”一节）。在 64 位体系结构上 ZONE_HIGHMEM 区总是空的。

通常 32 位 Linux 内核地址空间划分 0~3G 为用户空间，3~4G 为内核空间。Linux 将内核地址空间划分为三部分 ZONE_DMA、ZONE_NORMAL 和 ZONE_HIGHMEM，高端内存 HIGH_MEM 地址空间范围为 0xF8000000 ~ 0xFFFFFFFF (896MB~1024MB)。那么如内核是如何借助 128MB 高端内存地址空间是如何实现访问所有物理内存？

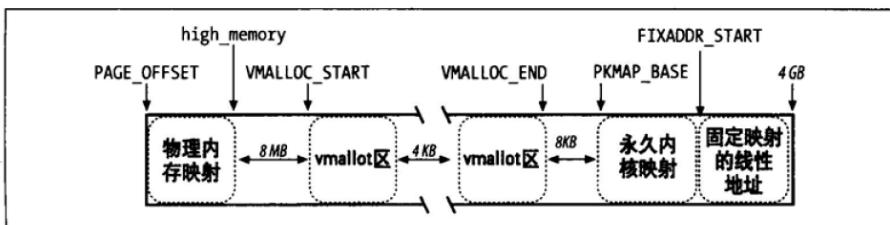
当内核想访问高于 896MB 物理地址内存时，从 0xF8000000 ~ 0xFFFFFFFF 地址空间范围内找一段相应大小空闲的逻辑地址空间，借用一会。借用这段逻辑地址空间，建立映射到想访问的那段物理内存（即填充内核 PTE 页面表），临时用一会，用完后归还。这样别人也可以借用这段地址空间访问其他物理内存，实现了使用有限的地址空间，访问所有物理内存。用户进程没有高端内存概念。

只有在内核空间才存在高端内存。用户进程最多只可以访问 3G 物理内存，而内核进程可以访问所有物理内存。

非连续内存区的线性地址

要查找线性地址的一个空闲区，我们可以从 PAGE_OFFSET 开始查找（通常为 0xc0000000，即第 4 个 GB 的起始地址）。图 8-7 显示了如何使用第 4 个 GB 的线性地址：

- 内存区的开始部分包含的是对前 896MB RAM 进行映射的线性地址（参见第二章“进程页表”一节）；直接映射的物理内存末尾所对应的线性地址保存在 high_memory 变量中。
- 内存区的结尾部分包含的是固定映射的线性地址（参见第二章“固定映射的线性地址”一节）。
- 从 PKMAP_BASE 开始，我们查找用于高端内存页框的永久内核映射的线性地址（参见本章前面“高端内存页框的内核映射”一节）。
- 其余的线性地址可以用于非连续内存区。在物理内存映射的末尾与第一个内存区之间插入一个大小为 8MB（宏 VMALLOC_OFFSET）的安全区，目的是为了“捕获”对内存的越界访问。出于同样的理由，插入其他 4KB 大小的安全区来隔离非连续的内存区。



直接映射区的作用是为了保证能够申请到物理地址上连续的内存区域，因为动态映射区，会产生内存碎片，导致系统启动一段时间后，想要成功申请到大量的连续的物理内存，非常困难，但是动态映射区带来了很高的灵活性(比如动态建立映射，缺页时才去加载物理页)。

alloc_page: 可以在高端内存区域分配，也可以在低端内存区域分配，最大 4M($2^{MAX_ORDER-1}$)个 PAGE) <伙伴系统>

kmalloc: 只能在低端内存区域分配(基于 ZONE_NORMAL)，最大 32 个 PAGE，共 128K，kzalloc/kcalloc 都是其变种 (slab.h 中如果定义了 KMALLOC_MAX_SIZE 宏，那么可以达到 8M 或者更大) <slab>

vmalloc: 只能在高端内存区域分配(基于 ZONE_HIGHMEM)

内存分配函数	分配原理	分配的最大内存	适用场合
<code>__get_free_pages</code>	直接对页框进行操作	4M	适用于分配较大量的连续物理内存
<code>kmem_cache_alloc</code>	基于slab机制实现	128KB	适合需要频繁申请释放相同大小内存块时使用
<code>kmalloc</code>	基于kmem_cache_alloc实现	128KB	最常见的分配方式，需要小于页框大小的内存时可以使用
<code>vmalloc</code>	建立非连续物理内存到虚拟地址的映射		物理不连续，适合需要大内存，但是对地址连续性没有要求的场合
<code>dma_alloc_coherent</code>	基于 <code>__alloc_pages</code> 实现	4M	适用于DMA操作
<code>ioremap</code>	实现已知物理地址到虚拟地址的映射		适用于物理地址已知的场合，如设备驱动
<code>alloc_bootmem</code>	在启动kernel时，预留一段内存，内核看不见		小于物理内存大小，内存管理要求较高

非连续内存区管理

从前面的讨论中我们已经知道，把内存区映射到一组连续的页框是最好的选择，这样会充分利用高速缓存并获得较低的平均访问时间。不过，如果对内存区的请求不是很频繁，那么，通过连续的线性地址来访问非连续的页框这样一种分配模式就会很有意义。这种模式的主要优点是避免了外碎片，而缺点是必须打乱内核页表。显然，非连续内存区的大小必须是4096的倍数。Linux在几个方面使用非连续内存区，例如，为活动的交换区分配数据结构（参见第十七章中的“激活和禁用交换区”一节），为模块分配空间（参见附录二），或者给某些I/O驱动程序分配缓冲区。此外，非连续内存区还提供了另一种使用高端内存页框的方法（参见后面的“分配非连续内存区”一节）。

<https://blog.csdn.net/abc3240660/article/details/81484984>

2.6.3. 伙伴系统

伙伴系统算法

内核应该为分配一组连续的页框而建立一种健壮、高效的分配策略。为此，必须解决著名的内存管理问题，也就是所谓的外碎片（*external fragmentation*）。频繁地请求和释放不同大小的一组连续页框，必然导致在已分配页框的块内分散了许多小块的空闲页框。由此带来的问题是，即使有足够的空闲页框可以满足请求，但要分配一个大块的连续页框就可能无法满足。

从本质上说，避免外碎片的方法有两种：

- 利用分页单元把一组非连续的空闲页框映射到连续的线性地址区间。
- 开发一种适当的技术来记录现存的空闲连续页框块的情况，以尽量避免为满足对小块的请求而分割大的空闲块。

基于以下三种原因，内核首选第二种方法：

- 在某些情况下，连续的页框确实是必要的，因为连续的线性地址不足以满足请求。一个典型的例子就是给 DMA 处理器分配缓冲区的内存请求（参见第十三章）。因为当在一次单独的 I/O 操作中传送几个磁盘扇区的数据时，DMA 忽略分页单元而直接访问地址总线，因此，所请求的缓冲区就必须位于连续的页框中。
- 即使连续页框的分配并不是很必要，但它在保持内核页表不变方面所起的作用也是不容忽视的。修改页表会怎样呢？从第二章我们知道，频繁地修改页表势必导致平均访问内存次数的增加，因为这会使 CPU 频繁地刷新转换后援缓冲器（TLB）的内容。
- 内核通过4MB的页可以访问大块连续的物理内存。这样减少了转换后援缓冲器的失效率，因此提高了访问内存的平均速度 [参见第二章“转换后援缓冲器（TLB）”一节]。

Linux 采用著名的伙伴系统（buddy system）算法来解决外碎片问题。把所有的空闲页框分组为 11 个区块链表，每个区块链表分别包含大小为 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 和 1024 个连续的页框。对 1024 个页框的最大请求对应着 4MB 大小的连续 RAM 块。每个块的第一个页框的物理地址是该块大小的整数倍。例如，大小为 16 个页框的块，其起始地址是 16×2^{12} ($2^{12} = 4096$ ，这是一个常规页的大小) 的倍数。

我们通过一个简单的例子来说明该算法的工作原理。

假设要请求一个 256 个页框的块（即 1MB）。算法先在 256 个页框的链表中检查是否有一个空闲块。如果没有这样的块，算法会查找下一个更大的页块，也就是，在 512 个页框的链表中找一个空闲块。如果存在这样的块，内核就把 256 的页框分成两等份，一半用作满足请求，另一半插入到 256 个页框的链表中。如果在 512 个页框的区块链表中也没找到空闲块，就继续找更大的块——1024 个页框的块。如果这样的块存在，内核把 1024 个页框块的 256 个页框用作请求，然后从剩余的 768 个页框中拿 512 个插入到 512 个页框的链表中，再把最后的 256 个插入到 256 个页框的链表中。如果 1024 个页框的链表还是空的，算法就放弃并发出错信号。

以上过程的逆过程就是页框块的释放过程，也是该算法名字的由来。内核试图把大小为 b 的一对空闲伙伴块合并为一个大小为 $2b$ 的单独块。满足以下条件的两个块称为伙伴：

- 两个块具有相同的大小，记作 b 。
- 它们的物理地址是连续的。
- 第一块的第一个页框的物理地址是 $2 \times b \times 2^{12}$ 的倍数。

该算法是迭代的，如果它成功合并所释放的块，它会试图合并 $2b$ 的块，以再次试图形成更大的块。

2.6.4. Slab

伙伴系统算法采用页框作为基本内存区，这适合于对大块内存的请求，但我们如何处理对小内存区的请求呢？比如说几十或几百个字节？

显然，如果为了存放很少的字节而给它分配一个整页框，这显然是一种浪费。取而代之的正确方法就是引入一种新的数据结构来描述在同一页框中如何分配小内存区。但这样也引出了一个新的问题，即所谓的内碎片 (internal fragmentation)。内碎片的产生主要是由于请求内存的大小与分配给它的大小不匹配而造成的。

slab 分配器试图在几个基本原则之间寻求一种平衡：

- 频繁使用的数据结构也会频繁分配和释放，因此应当缓存它们。
- 频繁分配和回收必然会导致内存碎片（难以找到大块连续的可用内存）。为了避免这种现象，空闲链表的缓存会连续地存放。因为已释放的数据结构又会放回空闲链表，因此不会导致碎片。
- 回收的对象可以立即投入下一次分配，因此，对于频繁的分配和释放，空闲链表能够提高其性能。
- 如果分配器知道对象大小、页大小和总的高速缓存的大小这样的概念，它会做出更明智的决策。
- 如果让部分缓存专属于单个处理器（对系统上的每个处理器独立而唯一），那么，分配和释放就可以在不加 SMP 锁的情况下进行。
- 如果分配器是与 NUMA 相关的，它就可以从相同的内存节点为请求者进行分配。
- 对存放的对象进行着色 (color)，以防止多个对象映射到相同的高速缓存行 (cache line)。

slab 层把不同的对象划分为所谓高速缓存组，其中每个高速缓存组都存放不同类型的对象。每种对象类型对应一个高速缓存。例如，一个高速缓存用于存放进程描述符 (task_struct 结构的一个空闲链表)，而另一个高速缓存存放索引节点对象 (struct inode)。有趣的是，kmalloc() 接口建立在 slab 层之上，使用了一组通用高速缓存。

然后，这些高速缓存又被划分为 slab (这也是这个子系统名字的来由)。slab 由一个或多个物理上连续的页组成。一般情况下，slab 也就仅仅由一页组成。每个高速缓存可以由多个 slab 组成。

每个 slab 都包含一些对象成员，这里的对象指的是被缓存的数据结构。每个 slab 处于三种状态之一：满、部分满或空。一个满的 slab 没有空闲的对象 (slab 中的所有对象都已被分配)。一个空的 slab 没有分配出任何对象 (slab 中的所有对象都是空闲的)。一个部分满的 slab 有一些对象已分配出去，有些对象还空闲着。当内核的某一部分需要一个新的对象时，先从部分满的 slab 中进行分配。如果没有部分满的 slab，就从空的 slab 中进行分配。如果没有空的 slab，就要创建一个 slab 了。显然，满的 slab 无法满足请求，因为它根本就没有空闲的对象。这种策略能减少碎片。

2.6.5. mmap

mmap() 系统调用使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以像访问普通内存一样对文件进行访问，不必再调用 read(), write() 等操作。

实际上，mmap() 系统调用并不是完全为了用于共享内存而设计的。它本身

提供了不同于一般对普通文件的访问方式，进程可以像读写内存一样对普通文件的操作。而 Posix 或 System V 的共享内存 IPC 则纯粹用于共享目的，当然 mmap()实现共享内存也是其主要应用之一。

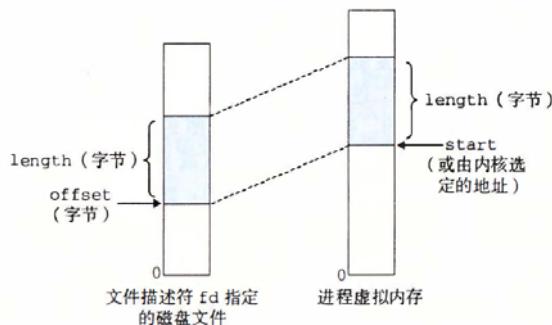
Linux 进程可以使用 mmap 函数来创建新的虚拟内存区域，并将对象映射到这些区域中。

```
#include <unistd.h>
#include <sys/mman.h>

void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

返回：若成功时则为指向映射区域的指针，若出错则为 MAP_FAILED(-1)。

mmap 函数要求内核创建一个新的虚拟内存区域，最好是从地址 start 开始的一个区域，并将文件描述符 fd 指定的对象的一个连续的片(chunk)映射到这个新的区域。连续的对象片大小为 length 字节，从距文件开始处偏移量为 offset 字节的地方开始。start 地址仅仅是一个暗示，通常被定义为 NULL。为了我们的目的，我们总是假设起始地址为 NULL。图 9-32 描述了这些参数的意义。



2.6.6. malloc 涉及的系统调用

每个 Unix 进程都拥有一个特殊的线性区，这个线性区就是所谓的堆 (*heap*)，堆用于满足进程的动态内存请求。内存描述符的 `start_brk` 与 `brk` 字段分别限定了这个区的开始地址和结束地址。

进程可以使用下面的 API 来请求和释放动态内存：

`malloc(size)`

请求 `size` 个字节的动态内存。如果分配成功，就返回所分配内存单元第一个字节的线性地址。

`calloc(n, size)`

请求含有 `n` 个大小为 `size` 的元素的一个数组。如果分配成功，就把数组元素初始化为 0，并返回第一个元素的线性地址。

`realloc(ptr, size)`

改变由前面的 `malloc()` 或 `calloc()` 分配的内存区字段的大小。

`free(addr)`

释放由 `malloc()` 或 `calloc()` 分配的起始地址为 `addr` 的线性区。

`brk(addr)`

直接修改堆的大小。`addr` 参数指定 `current->mm->brk` 的新值，返回值是线性区新的结束地址（进程必须检查这个地址和所请求的地址值 `addr` 是否一致）。

`sbrk(incr)`

类似于 `brk()`，不过其中的 `incr` 参数指定是增加还是减少以字节为单位的堆大小。

`brk()` 函数和以上列出的函数有所不同，因为它是唯一以系统调用的方式实现的函数，而其他所有的函数都是使用 `brk()` 和 `mmap()` 系统调用实现的 C 语言库函数（注 14）。

当用户态的进程调用 `brk()` 系统调用时，内核执行 `sys_brk(addr)` 函数。该函数首先验证 `addr` 参数是否位于进程代码所在的线性区。如果是，则立即返回，因为堆不能与进程代码所在的线性区重叠：

2.6.7. 页面置换算法

程序运行过程中，有时要访问的页面不在内存中，而需要将其调入内存。但是内存已经无空闲空间存储页面，为保证程序正常运行，系统必须从内存中调出一页程序或数据送到磁盘对换区，此时需要一定的算法来决定到底需要调出那个页面。通常将这种算法称为“页面置换算法”

1. 最佳置换算法 (OPT)：从主存中移出永远不再需要的页面；如无这样的页面存在，则选择最长时间不需要访问的页面。于所选择的被淘汰页面将是以

后永不使用的，或者是在最长时间内不再被访问的页面，这样可以保证获得最低的缺页率。最佳置换算法是一种理想化算法。

2. 先进先出置换算法 (FIFO): 是最简单的页面置换算法。这种算法的基本思想是：当需要淘汰一个页面时，总是选择驻留主存时间最长的页面进行淘汰，即先进入主存的页面先淘汰。其理由是：最早调入主存的页面不再被使用的可能性最大。

3. 最近最久未使用 (LRU) 算法：这种算法的基本思想是利用局部性原理，根据一个作业在执行过程中过去的页面访问历史来推测未来的行为。它认为过去一段时间里不曾被访问过的页面，在最近的将来可能也不会再被访问。所以，这种算法的实质是：当需要淘汰一个页面时，总是选择在最近一段时间内最久不用的页面予以淘汰。

2.7. 进程间通信

进程间通信 (IPC, InterProcess Communication) 是指在不同进程之间传播或交换信息。

IPC 的方式通常有管道 (包括无名管道和命名管道)、信号量、消息队列、共享内存、Socket、Streams 等。其中 Socket 和 Streams 支持不同主机上的两个进程 IPC。

https://blog.csdn.net/wh_sjc/article/details/70283843

2.7.1. 管道

管道，通常指无名管道，是 UNIX 系统 IPC 最古老的形式。

1、特点：

它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端。

它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）。

它可以看成是一种特殊的文件，对于它的读写也可以使用普通的 read、write 等函数。

但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

2、原型

```
#include <unistd.h>
Int pipe(int fd[2]); // 返回值：若成功返回 0，失败返回 -1
```

当一个管道建立时，它会创建两个文件描述符：fd[0]为读而打开，fd[1]为写而打开。若要数据流从父进程流向子进程，则关闭父进程的读端 (fd[0]) 与子进程的写端 (fd[1])；反之，则可以使数据流从子进程流向父进程。

2.7.2. FIFO

FIFO，也称为命名管道，它是一种文件类型。

1、特点

FIFO 可以在无关的进程之间交换数据，与无名管道不同。

FIFO 有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

2、原型

```
#include <sys/stat.h>
// 返回值：成功返回 0，出错返回 -1
int mkfifo(const char *pathname, mode_t mode);
```

其中的 mode 参数与 open 函数中的 mode 相同。一旦创建了一个 FIFO，就可以用一般的文件 I/O 函数操作它。

当 open 一个 FIFO 时，是否设置非阻塞标志 (O_NONBLOCK) 的区别：

若没有指定 O_NONBLOCK (默认)，只读 open 要阻塞到某个其他进程为写而打开此 FIFO。类似的，只写 open 要阻塞到某个其他进程为读而打开它。

若指定了 O_NONBLOCK，则只读 open 立即返回。而只写 open 将出错返回 -1 如果没有进程已经为读而打开该 FIFO，其 errno 置 ENXIO。

2.7.3. 消息队列

消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列 ID）来标识。

1、特点

消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。

消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。

消息队列可以实现消息的随机查询，消息不一定要以先进先出的次序读取，也可以按消息的类型读取。

2、原型

```
#include <sys/msg.h>
// 创建或打开消息队列：成功返回队列 ID，失败返回 -1
int msgget(key_t key, int flag);
// 添加消息：成功返回 0，失败返回 -1
int msgsnd(int msqid, const void *ptr, size_t size, int flag);
```

```
// 读取消息：成功返回消息数据的长度，失败返回-1  
int msgrcv(int msqid, void *ptr, size_t size, long type,int flag);  
// 控制消息队列：成功返回 0，失败返回-1  
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

在以下两种情况下，msgget 将创建一个新的消息队列：

如果没有与键值 key 相对应的的消息队列，并且 flag 中包含了 IPC_CREAT 标志位。

key 参数为 IPC_PRIVATE。

函数 msgrcv 在读取消息队列时，type 参数有下面几种情况：

type == 0，返回队列中的第一个消息；

type > 0，返回队列中消息类型为 type 的第一个消息；

type < 0，返回队列中消息类型值小于或等于 type 绝对值的消息，如果有多个，则取类型值最小的消息。

可以看出，type 值非 0 时用于以非先进先出次序读消息。也可以把 type 看做优先级的权值。

2.7.4. 信号量

信号量（semaphore）与已经介绍过的 IPC 结构不同，它是一个计数器。信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。

1、特点

信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。

每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

支持信号量组。

2、原型

最简单的信号量是只能取 0 和 1 的变量，这也是信号量最常见的一种形式，叫做二值信号量（Binary Semaphore）。而可以取多个正整数的信号量被称为通用信号量。

Linux 下的信号量函数都是在通用的信号量数组上进行操作，而不是在一个单一的二值信号量上进行操作。

```
#include <sys/sem.h>  
// 创建或获取一个信号量组：若成功返回信号量集 ID，失败返回-1
```

```
int semget(key_t key, int num_sems, int sem_flags);
// 对信号量组进行操作，改变信号量的值：成功返回 0，失败返回 -1
int semop(int semid, struct sembuf semoparray[], size_t numops);
// 控制信号量的相关信息
int semctl(int semid, int sem_num, int cmd, ...);
```

当 `semget` 创建新的信号量集合时，必须指定集合中信号量的个数（即 `num_sems`），通常为 1；如果是引用一个现有的集合，则将 `num_sems` 指定为 0。

2.7.5. 内存共享

共享内存（Shared Memory），指两个或多个进程共享一个给定的存储区。

1、特点

共享内存是最快的一种 IPC，因为进程是直接对内存进行存取。

因为多个进程可以同时操作，所以需要进行同步。

信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问。

2、原型

```
#include <sys/shm.h>
// 创建或获取一个共享内存：成功返回共享内存 ID，失败返回 -1
int shmget(key_t key, size_t size, int flag);
// 连接共享内存到当前进程的地址空间：成功返回指向共享内存的指针，失败返回 -1
void *shmat(int shm_id, const void *addr, int flag);
// 断开与共享内存的连接：成功返回 0，失败返回 -1
int shmdt(void *addr);
// 控制共享内存的相关信息：成功返回 0，失败返回 -1
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
```

当用 `shmget` 函数创建一段共享内存时，必须指定其 `size`；而如果引用一个已存在的共享内存，则将 `size` 指定为 0。

当一段共享内存被创建以后，它并不能被任何进程访问。必须使用 `shmat` 函数连接该共享内存到当前进程的地址空间，连接成功后把共享内存区对象映射到调用进程的地址空间，随后可像本地空间一样访问。

`shmdt` 函数是用来断开 `shmat` 建立的连接的。注意，这并不是从系统中删除该共享内

存，只是当前进程不能再访问该共享内存而已。

shmctl 函数可以对共享内存执行多种操作，根据参数 cmd 执行相应的操作。常用的是 IPC_RMID（从系统中删除该共享内存）。

2.7.6. ftok

系统建立 IPC 通讯（如消息队列、共享内存等） 必须指定一个 ID 值。通常情况下，该 id 值通过 ftok 函数得到。

ftok 原型如下：

```
key_t ftok(const char *pathname, int proj_id);
```

pathname 参数： 必须是一个已经存在且程序可范围的文件。

proj_id 参数： 虽然定义为一个整数，其实实际只有 8 个 bit 位有效，即如果该参数大于 255，则只有后 8bit 有效。

2.8. linux 程序编译、链接、调试

2.8.1. 基本过程

基本格式： gcc [options] file1 file2... //若不加入参数，则按默认参数依次执行编译、汇编和链接操作，生成的可执行文件名为 a.out

```
-E //只执行预处理操作, gcc -E circle.c -o circle.i  
-S //只执行到编译操作完成, 不进行汇编操作, 生成的是汇编文件(.s 或 .asm), 内容为汇编语言  
-c //执行编译和汇编, 但不进行链接, 即只生成可重定位目标文件(.o), 为二进制文件, 不生成完整的可执行文件  
-o filename //将操作后的内容输出到 filename 指定的文件中  
-static //对于支持动态链接的系统, 使用静态链接而不是动态链接进行链接操作  
-g //编译时生成 debug 有关的程序信息(供 gdb 使用)  
-O1、-O2 //规定编译器的优化等级, 优化级数越高执行效率一般越好, 但是优化会改变原有程序结构, 使得其汇编不易理解
```

编译、汇编和链接链接过程如下：

驱动程序首先运行 C 预处理器 cpp，它将 C 的源程序 main.c 翻译成一个 ASCII 码的中间文件.i

驱动程序运行 C 编译器 cc1，它将.i 翻译成一个 ASCII 汇编语言文件.s

驱动程序运行汇编器 as，它将 main.s 翻译成一个可重定位目标文件

驱动程序运行链接器程序 ld，将 .o 以及一些必要的系统目标文件组合起来，创建一个可执行目标文件

要运行可执行文件，我们在 Linux shell 的命令行上输入它的名字：

```
linux> ./a.out
```

shell 调用操作系统中一个叫做加载器 loader 的函数，它将可执行文件中的代码和数据复制到内存，然后将控制转移到这个程序的开头。

2.8.2. 静态链接

链接器读取一组可重定位目标文件，并把它们链接起来，形成一个输出的可执行文件。为了构造可执行文件，链接器必须完成两个主要任务：

- 符号解析(symbol resolution)
- 重定位(relocation)

实际上，所有的编译系统都提供一种机制，将所有相关的目标模块打包成为一个单独的文件，称为静态库(static library)。它可以用做链接器的输入。当链接器构造一个输出的可执行文件时，它只复制静态库里被应用程序引用的目标模块。

静态链接的缺点很明显，一是浪费空间，因为每个可执行程序中对所有需要的目标文件都要有一份副本，所以如果多个程序对同一个目标文件都有依赖，如多个程序中都调用了 printf() 函数，则这多个程序中都含有 printf.o，所以同一个目标文件都在内存存在多个副本；另一方面就是更新比较困难，因为每当库函数的代码修改了，这个时候就需要重新进行编译链接形成可执行程序。但是静态链接的优点就是，在可执行程序中已经具备了所有执行程序所需要的任何东西，在执行的时候运行速度快。

2.8.3. 动态链接

动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。

动态链接的优点显而易见，就是即使需要每个程序都依赖同一个库，但是该库不会像静态链接那样在内存中存在多分副本，而是这多个程序在执行时共

享同一份副本；另一个优点是，更新也比较方便，更新时只需要替换原来的目标文件，而无需将所有的程序再重新链接一遍。当程序下一次运行时，新版本的目标文件会被自动加载到内存并且链接起来，程序就完成了升级的目标。但是动态链接也是有缺点的，因为把链接推迟到了程序运行时，所以每次执行程序都需要进行链接，所以性能会有一定损失。

2.8.4. 常用的调试方法和工具

- `print` 语句：基本调试，获得关键变量
- 查询/`proc` 文件系统：获取有关文件系统支持，可用内存，CPU，运行程序的内核状态等信息
- `strace / ltrace`：最初的问题诊断，系统调用或库调用的相关问题，了解程序流程，`strace` 的和 `ltrace` 是两个在 Linux 中用来追踪程序的执行细节的跟踪工具
- `valgrind`：应用程序内存空间的问题
- `gdb`：检查应用程序运行时的行为，分析应用程序崩溃

2.8.5. coredump

我们经常听到大家说到程序 `core` 掉了，需要定位解决，这里说的大部分是指对应程序由于各种异常或者 `bug` 导致在运行过程中异常退出或者中止，并且在满足一定条件下（这里为什么说需要满足一定的条件呢？下面会分析）会产生一个叫做 `core` 的文件。

通常情况下，`core` 文件会包含了程序运行时的内存，寄存器状态，堆栈指针，内存管理信息还有各种函数调用堆栈信息等，我们可以理解为是程序工作当前状态存储生成第一个文件，许多的程序出错的时候都会产生一个 `core` 文件，通过工具分析这个文件，我们可以定位到程序异常退出的时候对应的堆栈调用等信息，找出问题所在并进行及时解决。

`coredump` 的原因有很多，这里总结一些比较常用的：

1. 内存访问越界
2. 多线程读写的数据未加锁保护。
3. 非法指针
4. 堆栈溢出

2.8.6. cmake 和 makefile 的区别

`makefile + make` 可理解为类 unix 环境下的项目管理工具，但它太基础了，

抽象程度不高，而且在 windows 下不太友好(针对 visual studio 用户)，于是就有了跨平台项目管理工具 cmake

cmake 是跨平台项目管理工具，它用更抽象的语法来组织项目。虽然，仍然是目标，依赖之类的东西，但更为抽象和友好，比如你可用 math 表示数学库，而不需要再具体指定到底是 math.dll 还是 libmath.so，在 windows 下它会支持生成 visual studio 的工程，在 linux 下它会生成 Makefile，甚至它还能生成 eclipse 工程文件。也就是说，从同一个抽象规则出发，它为各个编译器定制工程文件

2.8.7. 简述 cmake 到可执行文件的过程

- 编写 CMake 配置文件 CMakeLists.txt
- 执行命令 cmake PATH 或者 ccmake PATH 生成 Makefile
- 使用 make 命令进行编译，然后链接生成可执行文件

2.8.8. Debug 和 Release

Debug 通常称为调试版本，通过一系列编译选项的配合，编译的结果通常包含调试信息，而且不做任何优化，以为开发人员提供强大的应用程序调试能力。而 Release 通常称为发布版本，是为用户使用的，一般客户不允许在发布版本上进行调试。所以不保存调试信息，同时，它往往进行了各种优化，以期达到代码最小和速度最优。

2.9. shell

- Linux 下删除同一文件夹下所有满足条件的文件

```
find ./ -name "a.out" -exec rm -rf {} \;
```

- find

- grep

1、查找指定进程

命令：ps -ef|grep xx

2、查找指定进程个数

命令：ps -ef|grep -c xx

3、从文件中查找关键词，忽略大小写，默认情况区分大小写

命令：grep 'linux' test.txt

- ps

Linux 下显示系统进程的命令 ps，最常用的有 ps -ef 和 ps aux

- netstat.

netstat 是一个告诉我们系统中所有 tcp/udp/unix socket 连接状态的命令行工具。它会列出所有已经连接或者等待连接状态的连接。该工具在识别某个应用监听哪个端口时特别有用，我们也能用它来判断某个应用是否正常的在监听某个端口。

-a (all)显示所有选项， 默认不显示 LISTEN 相关

-t (tcp)仅显示 tcp 相关选项

-u (udp)仅显示 udp 相关选项

-l 仅列出有在 Listen (监听) 的服务状态

-p 显示建立相关链接的程序名

-r 显示路由信息， 路由表

-c 每隔一个固定时间， 执行该 netstat 命令。

3. 数据结构/算法

3.1. 哈希冲突解决

hash 的原理:

哈希表就是一种以 键-值(key-indexed) 存储数据的结构，我们只要输入待查找的值即 key，即可查找到其对应的值。

使用哈希函数将被查找的键转换为数组的索引。在理想的情况下，不同的键会被转换为不同的索引值，但是在有些情况下我们需要处理多个键被哈希到同一个索引值的情况。所以哈希查找的第二个步骤就是处理冲突

常见的几种 Hash 函数:

<https://blog.csdn.net/yt618121/article/details/81162836>

直接定址法: 取 Key 或者 Key 的某个线性函数值为散列地址。 $\text{Hash}(k) = k$ ，或者 $\text{Hash}(k) = a*k + b$, (a, b 均为常数)

数字分析法:

学习笔记

数字分析法：需要知道Key的集合，并且Key的位数比地址位数多，选择Key数字分布均匀的位。

Hash(Key) 取六位：

列数： 1 (2) 3 (4) 5 (6) (7) 8 (9) 10 11 12 (13)

key1: 5 2 4 2 7 5 8 5 3 6 5 1 3

key2: 5 4 4 8 7 7 7 5 4 8 9 5 1

key3: 3 1 5 3 7 8 5 4 6 3 5 5 2

key4: 5 3 6 4 3 2 5 4 5 3 2 6 4

平均取中法：

平方取中法：取Key平方值的中间几位作为Hash地址。因为在设置散列函数时不一定知道所有关键字，选取哪几位不确定。一个数的平方的中间几位和数本身的每一位都有关，这样可以使随机分布的Key，得到的散列地址也是随机分布的。如下：

Key	Key值平方	Hash地址(5位)
111112113	12345901655324769	01655
010111101	00102234363432201	34363
210222134	44193345623513956	45623

常用的解决冲突方法有：

开放定址法

这种方法也称再散列法，其基本思想是：当关键字key的哈希地址 $p=H(key)$ 出现冲突时，以 p 为基础，产生另一个哈希地址 p_1 ，如果 p_1 仍然冲突，再以 p_1 为基础，产生另一个哈希地址 p_2 ，...，直到找出一个不冲突的哈希地址 p_i ，将相应元素存入其中。这种方法有一个通用的再散列函数形式：

$$H_i = (H(key) + d_i) \% m \quad i=1, 2, \dots, n$$

其中 $H(key)$ 为哈希函数， m 为表长， d_i 称为增量序列。增量序列的取值方式不同，相应的再散列方式也不同。主要有以下三种：

线性探测再散列

$$d_i = 1, 2, 3, \dots, m-1$$

这种方法的特点是：冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。

二次探测再散列

$$d_i = 1^2, -1^2, 2^2, -2^2, \dots, k^2, -k^2 \quad (k \leq m/2)$$

这种方法的特点是：冲突发生时，在表的左右进行跳跃式探测，比较灵活。

伪随机探测再散列

$$d_i = \text{伪随机数序列}.$$

具体实现时，应建立一个伪随机数发生器，($i=(i+p) \% m$)，并给定一个随机数做起点。

例如，已知哈希表长度 $m=11$ ，哈希函数为： $H(key) = key \% 11$ ，则 $H(47) = 3$, $H(26) = 4$, $H(60) = 5$ ，假设下一个关键字为69，则 $H(69) = 3$ ，与47冲突。

如果用线性探测再散列处理冲突，下一个哈希地址为 $H1 = (3+1) \% 11 = 4$ ，仍然冲突，再找下一个哈希地址为 $H2 = (3+2) \% 11 = 5$ ，还是冲突，继续找下一个哈希地址为 $H3 = (3+3) \% 11 = 6$ ，此时不再冲突，将69填入5号单元。

如果用二次探测再散列处理冲突，下一个哈希地址为 $H1 = (3+1^2) \% 11 = 4$ ，仍然冲突，再找下一个哈希地址为 $H2 = (3-1^2) \% 11 = 2$ ，此时不再冲突，将69填入2号单元。

如果用伪随机探测再散列处理冲突，且伪随机数序列为：2, 5, 9, ..., 则下一个哈希地址为 $H1 = (3+2) \% 11 = 5$ ，仍然冲突，再找下一个哈希地址为 $H2 = (3+5) \% 11 = 8$ ，此时不再冲突，将69填入8号单元。

再哈希法

这种方法是同时构造多个不同的哈希函数：

$$H_i = RH_i(key) \quad i=1, 2, \dots, k$$

当哈希地址 $H_i = RH_i(key)$ 发生冲突时，再计算 $H_i = RH_{i+1}(key)$，直到冲突不再产生。**这种方法不易产生聚集，但增加了计算时间。**

链地址法

这种方法的基本思想是将所有哈希地址为 i 的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第 i 个单元中，因而查找、插入和删除主要在同义词链中进行。**链地址法适用于经常进行插入和删除的情况。**

3.2. 一致性哈希

在解决分布式系统中负载均衡的问题时候可以使用 Hash 算法让固定的一部分请求落到同一台服务器上，这样每台服务器固定处理一部分请求（并维护这些请求的信息），起到负载均衡的作用。

但是普通的余数 hash($hash(\text{用户 id}) \% \text{服务器机器数}$)算法伸缩性很差，当新增或者下线服务器机器时候，用户 id 与服务器的映射关系会大量失效。一致性 hash 则利用 hash 环对其进行改进。

<https://blog.csdn.net/bntX2jSQfEHy7/article/details/79549368>

3.3. 查找

3.3.1. 100 亿 URL 中判断某个 URL 是否存在/海量 url 去重

1. Bitmap

Bitmap 又叫做位图，判断数组中的某位索引值下的位是否为 1，来表示这个数是否存在。假如 53 存在，那么就将数组中的第 53 位置 1。这样 Bitmap 极大地压缩了所需要的内存空间，并且还额外地完成了对原始大型数据的排序工作。使用位图可以大大节省元素存储空间，并进行快速查找、排序、判重、删除等工作。

很显然，对于小数据量、数据取值很稀疏，上面的方法并没有什么优势，但对于海量的、取值分布很均匀的集合进行去重，Bitmap 极大地压缩了所需要的内存空间。

2. Bloom Filter

布隆过滤器实际上是一个很长的二进制矢量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率远远超过一般的算法，缺点是有一定的误识别率和删除困难。

失误率要保持在多少，数组长度，哈希函数的个数分别要设置多少就需要根据实际情况来选择了。

3.3.2. 链表中倒数第 k 个结点

[leetcode019. Remove Nth Node From End of List](#)

3.3.3. 字符串匹配 kmp

[leetcode028. Implement strStr\(\)](#)

3.4. 二叉树

3.4.1. 怎么把一颗二叉树原地变成一个双向链表？

[leetcode426. Convert Binary Search Tree to Sorted Doubly Linked List](#)

3.4.2. 红黑树（RB-tree）和 AVL

1. AVL 树（平衡二叉树）

AVL 树是带有平衡条件的二叉查找树，一般是用平衡因子差值判断是否平衡并通过旋转来实现平衡，左右子树树高不超过 1，和红黑树相比，AVL 树是严格的平衡二叉树，平衡条件必须满足（所有节点的左右子树高度差不超过 1）。不管我们是执行插入还是删除操作，只要不满足上面的条件，就要通过旋转来保持平衡，而的英文旋转非常耗时的，由此我们可以知道 AVL 树适合用于插入与删除次数比较少，但查找多的情况

由于维护这种高度平衡所付出的代价比从中获得的效率收益还大，故而实际的应用不多，更多的地方是用追求局部而不是非常严格整体平衡的红黑树。当然，如果应用场景中对插入删除不频繁，只是对查找要求较高，那么 AVL 还

是较优于红黑树。

2. RB-tree 红黑树

一种二叉查找树，但在每个节点增加一个存储位表示节点的颜色，可以是红或黑（非红即黑）。通过对任何一条从根到叶子的路径上各个节点着色的限制，红黑树确保没有一条路径会比其它路径长出两倍，因此，红黑树是一种弱平衡二叉树（由于是弱平衡，可以看到，在相同的节点情况下，AVL 树的高度低于红黑树），相对于要求严格的 AVL 树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，我们就用红黑树。

就插入节点导致树失衡的情况，AVL 和 RB-Tree 都是最多两次树旋转来实现复衡 rebalance，旋转的量级是 $O(1)$

删除节点导致失衡，AVL 需要维护从被删除节点到根节点 root 这条路径上所有节点的平衡，旋转的量级为 $O(\log N)$ ，而 RB-Tree 最多只需要旋转 3 次实现复衡，只需 $O(1)$ ，所以说 RB-Tree 删除节点的 rebalance 的效率更高，开销更小

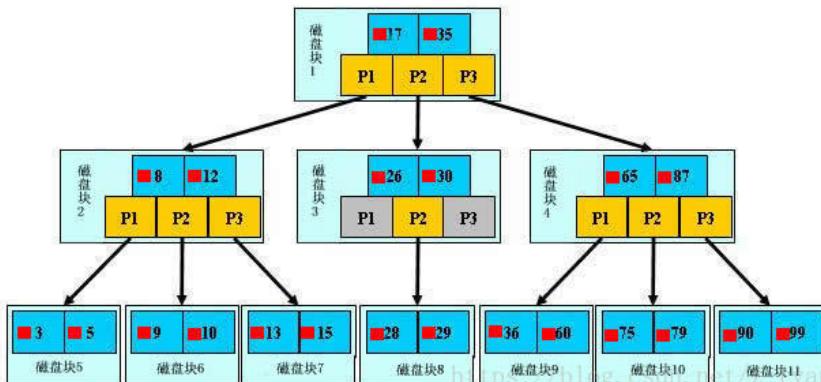
3.4.3. B 树和 B+树的区别，应用场景？

我们都知道二叉查找树的查找的时间复杂度是 $O(\log N)$ ，其查找效率已经足够高了，那为什么还有 B 树和 B + 树的出现呢？难道它两的时间复杂度比二叉查找树还小吗？

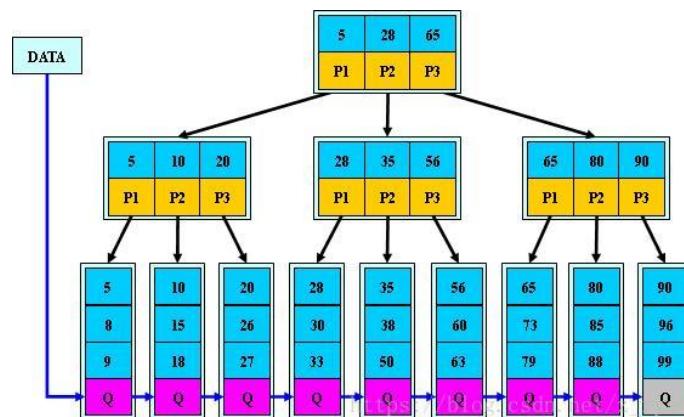
答案当然不是，B 树和 B + 树的出现是因为另外一个问题，那就是磁盘 IO；众所周知，IO 操作的效率很低，那么，当在大量数据存储中，查询时我们不能一下子将所有数据加载到内存中，只能逐一加载磁盘页，每个磁盘页对应树的节点。造成大量磁盘 IO 操作（最坏情况下为树的高度）。平衡二叉树由于树深度过大而造成磁盘 IO 读写过于频繁，进而导致效率低下。

所以，我们为了减少磁盘 IO 的次数，就必须降低树的深度，将“瘦高”的树变得“矮胖”。

3 阶 B 树：



B+树：

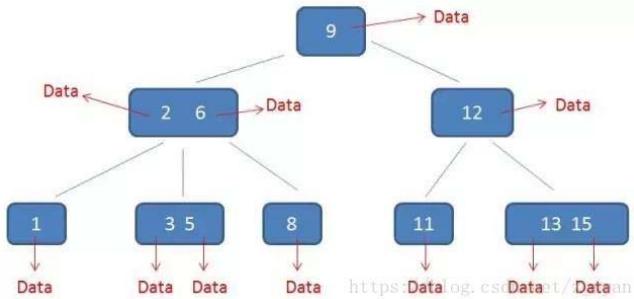


B+树的优势在于查找效率上，下面我们做一具体说明：

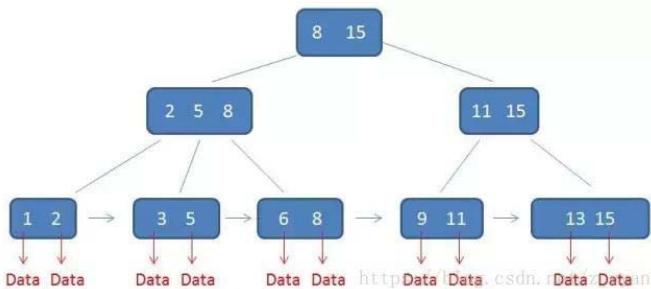
首先，B+树的查找和B树一样，类似于二叉查找树。起始于根节点，自顶向下遍历树，选择其分离值在要查找值的任意一边的子指针。在节点内部典型的使用是二分查找来确定这个位置。

不同的是，B+树中间节点没有卫星数据（索引元素所指向的数据记录），只有索引，而B树每个结点中的每个关键字都有卫星数据；这就意味着同样的大小的磁盘页可以容纳更多节点元素，在相同的数据量下，B+树更加“矮胖”，IO操作更少

B 树的卫星数据：



B+树的卫星数据：



其次，因为卫星数据的不同，导致查询过程也不同；B树的查找只需找到匹配元素即可，最好情况下查找到根节点，最坏情况下查找到叶子结点，所说性能很不稳定，而B+树每次必须查找到叶子结点，性能稳定

在范围查询方面，B+树的优势更加明显，B树的范围查找需要不断依赖中序遍历。首先二分查找到范围下限，在不断通过中序遍历，知道查找到范围的上限即可。整个过程比较耗时。

而B+树的范围查找则简单了许多。首先通过二分查找，找到范围下限，然后同过叶子结点的链表顺序遍历，直至找到上限即可，整个过程简单许多，效率也比较高。

B+树相比B树的优势：

1. 单一节点存储更多的元素，使得查询的IO次数更少；
2. 所有查询都要查找到叶子节点，查询性能稳定；
3. 所有叶子节点形成有序链表，便于范围查询。

3.5. 排序算法

表 15-1 各种排序算法的性质

算法种类	时间复杂度			空间复杂度	是否稳定
	最好情况	平均情况	最坏情况		
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
希尔排序				$O(1)$	否
快速排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$	$O(\log_2 n)$	否
堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	否
二路归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$	是
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	是

3.5.1. 选择排序

每趟从待排序的记录中选出关键字最小的记录，顺序放在已排好序的最后，直到全部记录排序完毕，时间复杂度 $O(n^2)$ 。

第 1 趟从 $R[0] \sim R[n-1]$ 中选取最小值，与 $R[0]$ 交换；

第 2 趟从 $R[1] \sim R[n-1]$ 中选取最小值，与 $R[1]$ 交换；

第 i 趟从 $R[i-1] \sim R[n-1]$ 中选取最小值，与 $R[i-1]$ 交换；

```

1 //选择排序
2 template <typename T>
3 void selectionSort(T arr[], int n)
4 {
5     for (int i = 0; i < n; ++i)
6     {
7         //寻找[i,n)区间里的最小值
8         int minIndex = i; //最小值下标
9         for (int j = i + 1; j < n; ++j)
10            if (arr[j] < arr[minIndex])
11                minIndex = j; //始终指向最小值
12         swap(arr[i], arr[minIndex]);
13     }
14 }
```

3.5.2. 插入排序

7.2 插入排序

7.2.1 算法

最简单的排序算法之一是插入排序(insertion sort)。插入排序由 $N - 1$ 趟(pass)排序组成。对于 $P=1$ 趟到 $P=N-1$ 趟，插入排序保证从位置 0 到位置 P 上的元素为已排序状态。插入排序利用了这样的事实：位置 0 到位置 $P-1$ 上的元素是已排过序的。图 7-1 显示一个简单的数组在每一趟插入排序后的情况。

图 7-1 表达了一般的方法：在第 P 趟，我们将位置 P 上的元素向左移动到它在前 $P+1$ 个元素中的正确位置上。图 7-2 中的程序实现该想法。第 2 行到第 5 行实现数据移动而没有明显使用交换。位置 P 上的元素存于 Tmp ，而(在位置 P 之前)所有更大的元素都被向右移动一个位置。然后 Tmp 被置于正确的位罝上。这种方法与在实现二叉堆时所用到的技巧相同。

初始	34	8	64	51	32	21	移动的位置
在 $p=1$ 之后	8	34	64	51	32	21	1
在 $p=2$ 之后	8	34	64	51	32	21	0
在 $p=3$ 之后	8	34	51	64	32	21	1
在 $p=4$ 之后	8	32	34	51	64	21	3
在 $p=5$ 之后	8	21	32	34	51	64	4

图 7-1 每趟后的插入排序

```

void
InsertionSort( ElementType A[ ], int N )
{
    int j, P;
    ElementType Tmp;
    for( P = 1; P < N; P++ )
    {
        Tmp = A[ P ];
        for( j = P; j > 0 && A[ j - 1 ] > Tmp; j-- )
            A[ j ] = A[ j - 1 ];
        A[ j ] = Tmp;
    }
}

```

图 7-2 插入排序例程

7.2.2 插入排序的分析

由于嵌套循环的每一个都花费 N 次迭代，因此插入排序为 $O(N^2)$ ，而且这个界是精确的，因为以反序输入可以达到该界。精确计算指出对于 P 的每一个值，第 4 行的测试最多执行 $P+1$ 次。对所有的 P 求和，得到总数为

$$\sum_{i=2}^N i = 2 + 3 + 4 + \dots + N = \Theta(N^2)$$

另一方面，如果输入数据已预先排序，那么运行时间为 $O(N)$ ，因为内层 for 循环的检测总是立即判定不成立而终止。事实上，如果输入几乎被排序(该术语将在下一节更严格地定义)，那么插入排序将运行得很快。由于这种变化差别很大，因此值得我们去分析该算法平均情形的行为。实际上，和各种其他排序算法一样，插入排序的平均情形也是 $\Theta(N^2)$ ，详见下节的分析。

3.5.3. 冒泡排序

15.2.1 冒泡排序

冒泡排序算法的基本思想是：假设待排序表长为 n ，从后往前（或从前往后）两两比较相邻元素的值，若为逆序（即 $A[i-1] > A[i]$ ），则交换它们，直到序列比较完。我们称它为一趟冒泡，结果将最小的元素交换到待排序列的第一个位置（关键字最小的元素如气泡一般逐渐往上“漂浮”直至“水面”，这就是冒泡排序名字的由来）。下一趟冒泡时，前一趟确定的最小元素不再参与比较，待排序列减少一个元素，每趟冒泡的结果把序列中的最小元素放到了序列的最终位置，……，这样最多做 $n-1$ 趟冒泡就能把所有元素排好序。

```
void BubbleSort(ElemType A[], int n){
    //用冒泡排序法将序列 A 中的元素按从小到大排列
    for(i=0;i<n-1;i++) {
        flag=false;           //表示本趟冒泡是否发生交换的标志
        for(j=n-1;j>i;j--)   //一趟冒泡过程
            if(A[j-1].key>A[j].key){ //若为逆序
                swap(A[j-1], A[j]); //交换
                flag=true;
            }
        if(flag==false)
            return ;           //本趟遍历后没有发生交换，说明表已经有序
    }
}
```

冒泡排序算法的性能分析如下：

空间复杂度为 $O(1)$ ，最坏情况下时间复杂度为 $O(n^2)$ ，最好情况下（表中元素基本有序）时间复杂度为 $O(n)$ ，其平均时间复杂度为 $O(n^2)$ 。

稳定性：冒泡排序是一个稳定的排序方法。

3.5.4. 希尔排序

希尔排序的基本思想是：先将待排序表分割成若干形如 $L[i, i+d, i+2d, \dots, i+kd]$ 的“特殊”子表，分别进行直接插入排序，当整个表中元素已呈“基本有序”时，再对全体记录进行一次直接插入排序。希尔排序的排序过程如下：

先取一个小于 n 的步长 d_1 ，把表中全部记录分成 d_1 个组，所有距离为 d_1 的倍数的记录放在同一个组中，在各组中进行直接插入排序；然后取第二个步长 $d_2 < d_1$ ，重复上述过程，直到所取到的 $d_i=1$ ，即所有记录已放在同一组中，再进行直接插入排序，由于此时已经具有较好的局部有序性，故可以很快得到最终结果。到目前为止，尚未求得一个最好的增量序列，希尔提出的方法是 $d_1=n/2$, $d_{i+1}=\lfloor d_i/2 \rfloor$ ，并且最后一个增量等于 1。

希尔排序算法的性能分析如下：

空间复杂度为 $O(1)$ 。

时间效率：由于希尔排序的时间复杂度依赖于增量序列的函数，这涉及数学上尚未解决的难题，所以其时间复杂度分析比较困难。当 n 在某个特定范围时，希尔排序的时间复杂度约为 $O(n^{1.3})$ 。在最坏情况下希尔排序的时间复杂度为 $O(n^2)$ 。

3.5.5. 堆排序

堆的结构可以分为大根堆和小根堆，是一个完全二叉树，而堆排序是根据堆的这种数据结构设计的一种排序，初始化建堆的时间复杂度为 $O(N)$ ，排序重建堆的时间复杂度为 $O(N \log N)$

1. 首先将无序元素构造成一个堆（新插入的数据与其父结点比较）
2. 将堆顶元素与末尾元素交换，将最大元素“沉”到数组末端重新调整结构，使其满足堆定义
3. 然后继续交换堆顶元素与当前末尾元素，反复执行调整+交换步骤，直到整个序列有序。

3.5.6. 归并排序

递归形式的二路归并排序算法是基于分治的，其过程如下：

分解：将含有 n 个元素的待排序表分成各含 $n/2$ 个元素的子表，采用二路归并排序算法对两个子表递归地进行排序；

合并：合并两个已排序的子表得到排序结果。代码如下：

```
void MergeSort(ElemType A[], int low, int high) {
    if(low<high) {
        int mid=(low+high)/2;           //从中间划分两个子序列
        MergeSort(A, low, mid);        //对左侧子序列进行递归排序
        MergeSort(A, mid+1, high);     //对右侧子序列进行递归排序
        Merge(A, low, mid, high);      //归并
    } //if
}
```

Merge() 的功能是将前后相邻的两个有序表归并为一个有序表的算法。设两段有序表 A[low…mid]、A[mid+1…high] 存放在同一顺序表中相邻的位置上，先将它们复制到辅助数组 B 中。每次从对应 B 中的两个段取出一个记录进行关键字的比较，将较小者放入 A 中，当数组 B 中有一段超出其表长时，将另一段中的剩余部分直接复制到 A 中。算法如下：

```
ElemType *B=(ElemType *)malloc((n+1) * sizeof(ElemType)); //辅助数组 B
void Merge(ElemType A[], int low, int mid, int high) {
    //表 A 的两段 A[low…mid] 和 A[mid+1…high] 各自有序，将它们合并成一个有序表
    for(int k=low;k<=high;k++)
        B[k]=A[k];           //将 A 中所有元素复制到 B 中
    for(i=low, j=mid+1, k=i;i<=mid&&j<=high;k++) {
        if(B[i]<B[j])       //比较 B 的左右两段中的元素
            A[k]=B[i++];    //将较小值复制到 A 中
        else
            A[k]=B[j++];
    } //for
    while(i<=mid)   A[k++]=B[i++]; //若第一个表未检测完，复制
    while(j<=high)  A[k++]=B[j++]; //若第二个表未检测完，复制
}
```

3.5.7. 快排

```
1. int partition(vector<int> &v, int begin, int end) {
2.     int k = end;
3.     while (begin<end) {
4.         while (begin<end && v[begin]<= v[k]) {
5.             ++begin;
6.         }
7.         while (begin<end && v[end]>= v[k]) {
8.             --end;
9.         }
10.        swap(v[begin], v[end]);
11.    }
12.    swap(v[begin], v[k]);
13.    return begin;
14. }
```

```

9.         }
10.        swap(v[begin], v[end]);
11.    }
12.    swap(v[begin], v[k]);
13.    return begin;
14. }
15. void quick_sort(vector<int> &v, int begin, int end) {
16.     if (begin >= end) return;
17.     int index = partition(v, begin, end);
18.     quick_sort(v, begin, index - 1);
19.     quick_sort(v, index + 1, end);
20. }

```

3.6. 多路归并排序

外部排序最常用的算法是多路归并排序，即将原文件分解成多个能够一次性装入内存的部分，分别把每一部分调入内存完成排序。然后，对已经排序的子文件进行归并排序。

多路归并排序算法在常见数据结构书中都有涉及。从二路到多路(k 路)，增大 k 可以减少外存信息读写时间，但 k 个归并段中选取最小的记录需要比较 $k-1$ 次，为了降低选出每个记录需要的比较次数 k ，引出了“败者树”的概念。

败者树是对树形选择排序的一种变形，可以视为一棵完全二叉树。每个叶结点存放各归并段在归并过程中当前参加比较的记录，内部结点用来记忆左右子树中的“失败者”，而让胜者往上继续进行比较，一直到根结点。如果比较两个数，大的为失败者、小的为胜利者，则根结点指向的数为最小数。

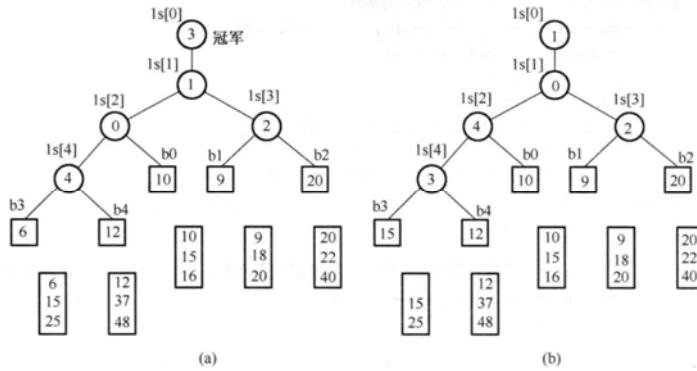


图 15-5 实现五路归并的败者树

如图 15-5(a)所示， b_3 与 b_4 比较， b_4 是败者，因此将段号 4 写入父结点 ls_4 。 b_1 与 b_2 比较， b_2 是败者，将段号 2 写入 ls_3 。 b_3 与 b_4 的胜者 b_3 与 b_0 比较， b_0 是败者，将段号 0 写入 ls_2 。最后两个胜者 b_3 与 b_1 比较， b_1 是败者，段号写入 ls_1 。而将胜者 b_3 的段号写入 ls_0 。此时，根结点 ls_0 所指的段的关键字最小。 b_3 中的 6 输出后，将下一关键字填入 b_3 ，继续比较。

注意图中第一个叶子结点编号为 b_0 。

因为 k 路归并的败者树深度为 $\lceil \log_k k \rceil$ ，因此 k 个记录中选择最小关键字，最多需要 $\lceil \log_k k \rceil$ 次比较。显然比依次比较的 $k-1$ 次小得多。

3.7. Topk (海量数据)

局部排序：冒泡算法，内存受限，无需排序

并归排序：无需排序

堆排序:

只找到 topK，不管顺序。上述的 bubbleSortK，其实 topK 是有顺序的。这相对题目而言，没必要。那么可以从这里入手，不管 topK 的顺序，再优化。先用前 k 个元素生成一个小顶堆，这个小顶堆用于存储，当前最大的 k 个元素。接着，从第 k+1 个元素开始扫描，和堆顶（堆中最小的元素）比较，如果被扫描的元素大于堆顶，则替换堆顶的元素，并调整堆，以保证堆内的 k 个元素，总是当前最大的 k 个元素。直到，扫描完所有 n-k 个元素，最终堆中的 k 个元素，就是的 TopK

3.8. 手撕

1、 char *s1, const char *s2, 删除 s1 中 s2 出现过的字符

例如:

```
char str1[] = "Abc*defghicc";
```

```
char str2[] = "Ac*ic";
```

处理结果为: str1[] = “bdefgh”

```
1. for(i = 0;str1[i] !='\0';i++){
2.     while(str1[i] != str2[j] && j <len) j++;
3.
4.     if(j>=len){
5.         str1[k] = str1[i];
6.         k++;
7.     }
8.     j = 0;
9. }
```

2、 LRU cache

<https://leetcode.com/problems/lru-cache/>

3、 洗牌算法

Fisher-Yates Shuffle 算法:

最早提出这个洗牌方法的是 Ronald A. Fisher 和 Frank Yates，即 Fisher - Yates Shuffle，其基本思想就是从原始数组中随机取一个之前没取过的数字到新的数组中，具体如下：

1. 初始化原始数组和新数组，原始数组长度为 n(已知);
 2. 从还没处理的数组（假如还剩 k 个）中，随机产生一个[0, k)之间的数字 p (假设数组从 0 开始);
 3. 从剩下的 k 个数中把第 p 个数取出;
 4. 重复步骤 2 和 3 直到数字全部取完;
- 时间复杂度为 O(n*n),空间复杂度为 O(n).

```
1. for (int i=0;i<M;++i)
2. {
3.     k=rand()%arr.size();
4.     res.push_back(arr[k]);
5.     arr.erase(arr.begin()+k);
6. }
```

Knuth-Durstenfeld Shuffle:

Knuth 和 Durstenfeld 在 Fisher 等人的基础上对算法进行了改进，在原始数组上对数字进行交互，省去了额外 $O(n)$ 的空间。该算法的基本思想和 Fisher 类似，每次从未处理的数据中随机取出一个数字，然后把该数字放在数组的尾部，即数组尾部存放的是已经处理过的数字。

算法步骤为：

1. 建立一个数组大小为 n 的数组 arr ，分别存放 1 到 n 的数值；
2. 生成一个从 0 到 $n - 1$ 的随机数 x ；
3. 输出 arr 下标为 x 的数值，即为第一个随机数；
4. 将 arr 的尾元素和下标为 x 的元素互换；
5. 同 2，生成一个从 0 到 $n - 2$ 的随机数 x ；
6. 输出 arr 下标为 x 的数值，为第二个随机数；
7. 将 arr 的倒数第二个元素和下标为 x 的元素互换；

```
1. for (int i=arr.size()-1;i>=0;--i)
2. {
3.     srand((unsigned)time(NULL));
4.     swap(arr[rand()%(i+1)],arr[i]);
5. }
```

4. C++

4.1. 类

4.1.1. struct 和 class 区别

默认的继承访问权限 struct 是 public 的，class 是 private 的；struct 作为数据结构的实现体，它默认的数据访问控制是 public 的，而 class 作为对象的实现体，它默认的成员变量访问控制是 private 的

4.1.2. 为什么要字节对齐

许多计算机系统对基本数据类型的合法地址做出了一些限制，要求某种类型对象的地址必须是某个值 K (通常是 2、4 或 8)的倍数。这种对齐限制简化了形成处理器和内存系统之间接口的硬件设计。例如，假设一个处理器总是从内存中取 8 个字节，则地址必须为 8 的倍数。如果我们能保证将所有的 `double` 类型数据的地址对齐成 8 的倍数，那么就可以用一个内存操作来读或者写值了。否则，我们可能需要执行两次内存访问，因为对象可能被分放在两个 8 字节内存块中。

无论数据是否对齐，x86-64 硬件都能正确工作。不过，Intel 还是建议要对齐数据以提高内存系统的性能。对齐原则是任何 K 字节的基本对象的地址必须是 K 的倍数。可以

4.1.3. C++空类有哪些成员函数？

```
class Empty
{
public:
    Empty(); // 缺省构造函数
    Empty( const Empty& ); // 拷贝构造函数
    ~Empty(); // 析构函数
    Empty& operator=( const Empty& ); // 赋值运算符
    Empty* operator&(); // 取址运算符
    const Empty* operator&() const; // 取址运算符 const
};
```

4.1.4. 静态成员函数能调用非静态成员函数吗

因为类的静态成员和普通成员其实就一种区别，那就是静态成员本身没有 `this` 指针，所以静态成员属于类而不属于类对象。如果我们想在类的静态成员函数里面调用类的普通成员，只需要把类指针当做参数传入静态成员函数里面，静态成员函数可以使用这个指针调用类的普通成员

4.1.5. 深拷贝浅拷贝的区别

浅拷贝只是对指针的拷贝，拷贝后两个指针指向同一个内存空间，深拷贝不但对指针进行拷贝，而且对指针指向的内容进行拷贝，经深拷贝后的指针是指向两个不同地址的指针。

浅拷贝带来问题的本质在于析构函数释放多次堆内存，使用 `std::shared_ptr`，可以完美解决这个问题

4.1.6. 类如何防止被拷贝

```
1. class noncopyable
2. {
```

```

3. protected:
4.     noncopyable() {}
5.     ~noncopyable() {}
6. private:
7.     noncopyable( const noncopyable& );
8.     noncopyable& operator=( const noncopyable& );
9. };
10.
11. class noncopyable
12. {
13. protected:
14.     //constexpr noncopyable() = default;
15.     // ~noncopyable() = default;
16.     noncopyable( const noncopyable& ) = delete;
17.     noncopyable& operator=( const noncopyable& ) = delete;
18. };
19.

```

4.2. 面向对象

4.2.1. 面向对象主要有四大特性

1. 抽象

忽略一个主题中与当前目标无关的东西,专注的注意与当前目标有关的方面.(就是把现实世界中的某一类东西,提取出来,用程序代码表示,抽象出来的一般叫做类或者接口).抽象并不打算了解全部问题,而是选择其中的一部分,暂时不用部分细节.抽象包括两个方面,一个数据抽象,而是过程抽象.

数据抽象 -->表示世界中一类事物的特征,就是对象的属性.比如鸟有翅膀,羽毛等(类的属性)

过程抽象 -->表示世界中一类事物的行为,就是对象的行为.比如鸟会飞,会叫(类的方法)

2. 封装

封装就是把过程和数据包围起来,对数据的访问只能通过特定的界面.如私有变量,用 set,get 方法获取

3. 继承

一种联结类的层次模型,并且允许和鼓励类的重用,提供一种明确表达共性的方法.对象的一个新类可以从现有的类中派生,这个过程称为类继承.新类继承了原始类的特性,新类称为原始类的派生类(子类),原始类称为新类的基类(父类).派生类可以从它的父类哪里继承方法和实例变量,并且类可以修改或增加新的方法使之更适合特殊的需要.因此可以说,继承为了重用父类代码,同时为实现多态性作准备.

4. 多态

多态是指允许不同类的对象对同一消息做出响应.多态性包括参数化多态性

和包含多态性.多态性语言具有灵活/抽象/行为共享/代码共享的优势,很好的解决了应用程序函数同名问题.总的来说,方法的重写,重载与动态链接构成多态性.java 引入多态的概念原因之一就是弥补类的单继承带来的功能不足.

动态链接 -->对于父类中定义的方法,如果子类中重写了该方法,那么父类类型的引用将调用子类中的这个方法,这就是动态链接.

4.2.2. C 语言实现 C++的继承

```

1. typedef void(*FUN)(); //重定义一个函数指针类型
2.
3. //父类
4. struct Base
5. {
6.     FUN _f;
7. };
8.
9. //子类
10. struct Derived
11. {
12.     Base _b; //在子类中定义一个基类的对象即可实现对父类的继承
13. };
14.
15.
16. void FunB()
17. {
18.     printf("%s\n", "Base::fun()");
19. }
20. void FunD()
21. {
22.     printf("%s\n", "Derived::fun()");
23. }
24.
25. void Test2()
26. {
27.     Base b; //父类对象
28.     Derived d; //子类对象
29.
30.     b._f = FunB(); //父类对象调用父类同名函数
31.     d._b._f = FunD(); //子类调用子类的同名函数
32.
33.     Base *pb = &b; //父类指针指向父类对象
34.     pb->_f();
35.
36.     pb = (Base *)&d; //让父类指针指向子类的对象,由于类型不匹配所以要进行强
   转
37.     pb->_f();
38.
39. }
```

4.2.3. 虚析构作用

总的来说虚析构函数是为了避免内存泄露, 而且是当子类中会有指针成员

变量时才会使用得到的。也就说虚析构函数使得在删除指向子类对象的基类指针时可以调用子类的析构函数达到释放子类中堆内存的目的，而防止内存泄露的。

1. 如果父类的析构函数不加 virtual 关键字

当父类的析构函数不声明成虚析构函数的时候，当子类继承父类，父类的指针指向子类时，`delete` 掉父类的指针，只调动父类的析构函数，而不调动子类的析构函数。

2. 如果父类的析构函数加 virtual 关键字

当父类的析构函数声明成虚析构函数的时候，当子类继承父类，父类的指针指向子类时，`delete` 掉父类的指针，先调动子类的析构函数，再调动父类的析构函数。

4.2.4. 如何实现多态的

C++多态性是通过虚函数来实现的，多态与非多态的实质区别就是函数地址是早绑定还是晚绑定。如果函数的调用，在编译器编译期间就可以确定函数的调用地址，并生产代码，是静态的，就是说地址是早绑定的。而如果函数调用的地址不能在编译器期间确定，需要在运行时才确定，这就属于晚绑定。

存在虚函数的类都有一个一维的虚函数表叫做虚表。当类中声明虚函数时，编译器会在类中生成一个虚函数表；类的对象有一个指向虚表开始的虚指针。虚表是和类对应的，虚表指针是和对象对应的，在程序运行时，根据对象的类型去初始化 `vptr`，从而让 `vptr` 正确的指向所属类的虚表，从而在调用虚函数时，就能够找到正确的函数。

4.2.5. 虚函数与纯虚函数的区别

```
virtual void function() {函数体}  
virtual void function()=0;
```

虚函数必须定义自己的实现方法，在 C++ 中是实现多态的机制，纯虚函数（所在的类为抽象类不能生成对象）不能定义自己的实现方法，是用来规范派生类的行为的，即接口。

4.2.6. 重载和重写的区别

● 重载

在同一个作用域内，函数名相同，参数列表不同（参数个数不同，或者参数类型不同，或者参数个数和参数类型都不同），返回值类型可相同也可不同，仅返回值不同不能重载；这种情况叫做 C++ 的重载。

C++ 函数重载达到的效果：调用函数名相同的函数，会根据实参的类型和实

参顺序以及实参数选择相应的函数；

C++函数重载是一种静态多态（又叫做静态联编，静态绑定，静态决议）

- 覆盖（又叫重写）

覆盖（重写）的前提条件：父类函数为虚函数；

覆盖（重写）的概念：当在子类中定义了一个与父类完全相同的虚函数时，则称子类的这个函数重写（也称覆盖）了父类的这个虚函数。

- 隐藏

如果在父类和子类中有相同名字的成员；那么在子类中。会将父类的成员隐藏；隐藏以后的直接效果就是：无论在子类的内部或者外部（通过子类成员）访问该成员；全都是访问子类的同名成员；如果在子类内部或者外部（通过子类成员）访问同名的成员函数，则需要根据函数调用的规则来调用子类的同名成员函数；

4.2.7. 为什么鼓励使用组合不使用继承

只有在对象之间关系具有很强的 is a 关系的时候才使用继承，继承和组合都能达到一个代码复用的效果，但是类的继承通常是白箱复用，对象组合通常为黑箱复用。我们在使用继承的时候同时也拥有了父对象中的保护成员，增加了耦合度。而对象组合就只需要在使用的时候接口稳定，耦合度低。

4.2.8. 重写的函数中含有默认参数

默认参数的值只和静态类型有关，是静态绑定的，虽然虚函数的是动态绑定的，但默认参数是静态绑定的。只有动态绑定的东西才应该被重写

4.2.9. 访问权限说明符

成员	派生类的成员和(派生类的)友元	类的用户
public成员	可访问	可访问
protected成员	只能访问派生类对象中的基类部分的protected成员	不可访问
private成员	不可访问	不可访问

```
struct Base {
public:
    string pub_string = "public string";
protected:
    string pro_string = "protected string";
private:
    string pri_string = "private string";//只有类自己能访问
};

struct Derived : public Base {
public:
    void access_parent_public(const Base &b){
        cout << b.pub_string << endl;
    }
    //不能直接访问基类对象的protected成员
    void access_parent_protected(const Base &b){
        //cout << b.pro_string << endl;
    }
    //派生类只能访问派生类对象中基类部分的protected成员
    void access_protected_in_derived(){
        cout << pro_string << endl;
    }
};
```

对于派生类的派生类(包括其成员和友元):

继承方式	public成员	protected成员	private成员
public继承	可访问	只能访问继承到的受保护成员	不可访问
protected继承	只能访问继承到的public成员	不可访问?	不可访问
private继承	不可访问	不可访问	不可访问

对于派生类的用户:

继承方式	public成员	protected成员	private成员
public继承	可访问	不可访问	不可访问
protected继承	不可访问	不可访问	不可访问
private继承	不可访问	不可访问	不可访问

4.3. C++11 新特性有哪些?

<https://blog.csdn.net/caogenwangbaoqiang/article/details/79438279>

4.3.1. Lambda

根据算法接受一元谓词还是二元谓词，我们传递给算法的谓词必须严格接受一个或两个参数。但是，有时我们希望进行的操作需要更多参数，超出了算法对谓词的限制。例如，为上一节最后一个练习所编写的程序中，就必须将大小 5 硬编码到划分序列的谓词中。如果在编写划分序列的谓词时，可以不必为每个可能的大小都编写一个独立的谓词，显然更有实际价值。

一个相关的例子是，我们将修改 10.3.1 节（第 345 页）中的程序，求大于等于一个给定长度的单词有多少。我们还会修改输出，使程序只打印大于等于给定长度的单词。

我们将此函数命名为 `biggies`, 其框架如下所示:

```
void biggies(vector<string> &words,
             vector<string>::size_type sz)
{
    elimDups(words); // 将 words 按字典序排序, 删除重复单词
    // 按长度排序, 长度相同的单词维持字典序
    stable_sort(words.begin(), words.end(), isShorter);
    // 获得一个迭代器, 指向第一个满足 size()>= sz 的元素
    // 计算满足 size >= sz 的元素的数目
    // 打印长度大于等于给定值的单词, 每个单词后面接一个空格
}
```

我们的新问题是在 `vector` 中寻找第一个大于等于给定长度的元素。一旦找到了这个元素, 根据其位置, 就可以计算出有多少元素的长度大于等于给定值。

我们可以使用标准库 `find_if` 算法来查找第一个具有特定大小的元素。类似 `find` (参见 10.1 节, 第 336 页), `find_if` 算法接受一对迭代器, 表示一个范围。但与 `find` 不同的是, `find_if` 的第三个参数是一个谓词。`find_if` 算法对输入序列中的每个元素调用给定的这个谓词。它返回第一个使谓词返回非 0 值的元素, 如果不存在这样的元素, 则返回尾迭代器。

编写一个函数, 令其接受一个 `string` 和一个长度, 并返回一个 `bool` 值表示该 `string` 的长度是否大于给定长度, 是一件很容易的事情。但是, `find_if` 接受一元谓词——我们传递给 `find_if` 的任何函数都必须严格接受一个参数, 以便能用来自输入序列的一个元素调用它。没有任何办法能传递给它第二个参数来表示长度。为了解决此问题, 需要使用另外一些语言特性。

介绍 lambda

我们可以向一个算法传递任何类别的可调用对象 (callable object)。对于一个对象或一个表达式, 如果可以对其使用调用运算符 (参见 1.5.2 节, 第 21 页), 则称它为可调用的。即, 如果 `e` 是一个可调用的表达式, 则我们可以编写代码 `e(args)`, 其中 `args` 是一个逗号分隔的一个或多个参数的列表。

到目前为止, 我们使用过的仅有的两种可调用对象是函数和函数指针 (参见 6.7 节, 第 221 页)。还有其他两种可调用对象: 重载了函数调用运算符的类, 我们将在 14.8 节 (第 506 页) 介绍, 以及 **lambda 表达式** (`lambda expression`)。

一个 `lambda` 表达式表示一个可调用的代码单元。我们可以将其理解为一个未命名的内联函数。与任何函数类似, 一个 `lambda` 具有一个返回类型、一个参数列表和一个函数体。但与函数不同, `lambda` 可能定义在函数内部。一个 `lambda` 表达式具有如下形式

```
[capture list] (parameter list) -> return type { function body }
```

其中, `capture list` (捕获列表) 是一个 `lambda` 所在函数中定义的局部变量的列表 (通常为空); `return type`、`parameter list` 和 `function body` 与任何普通函数一样, 分别表示返回类型、参数列表和函数体。但是, 与普通函数不同, `lambda` 必须使用尾置返回 (参见 6.3.3 节, 第 206 页) 来指定返回类型。

我们可以忽略参数列表和返回类型, 但必须永远包含捕获列表和函数体

```
auto f = [] { return 42; };
```

调用 `find_if`

使用此 `lambda`, 我们就可以查找第一个长度大于等于 `sz` 的元素:

```
// 获取一个迭代器，指向第一个满足 size()>= sz 的元素
auto wc = find_if(words.begin(), words.end(),
[sz](const string &a)
{ return a.size() >= sz; });
```

这里对 `find_if` 的调用返回一个迭代器, 指向第一个长度不小于给定参数 `sz` 的元素。如果这样的元素不存在, 则返回 `words.end()` 的一个拷贝。

4.3.2. 自动类型推导和 `decltype`

编程时常常需要把表达式的值赋给变量, 这就要求在声明变量的时候清楚地知道表达式的类型。然而要做到这一点并非那么容易, 有时甚至根本做不到。为了解决这个问题, C++11 新标准引入了 `auto` 类型说明符, 用它就能让编译器替我们去分析表达式所属的类型。和原来那些只对应一种特定类型的说明符 (比如 `double`) 不同, `auto` 让编译器通过初始值来推算变量的类型。显然, `auto` 定义的变量必须有初始值:

有时会遇到这种情况: 希望从表达式的类型推断出要定义的变量的类型, 但是不想用该表达式的值初始化变量。为了满足这一要求, C++11 新标准引入了第二种类型说明符 `decltype`, 它的作用是选择并返回操作数的数据类型。在此过程中, 编译器分析表达式并得到它的类型, 却不实际计算表达式的值:

```
decltype(f()) sum = x; // sum 的类型就是函数 f 的返回类型
```

4.3.3. `=deleted`

```
1 int func()=delete;
2 //防止对象拷贝的实现
3 struct NoCopy
4 {
5     NoCopy & operator =(const NoCopy &) = delete;
6     NoCopy(const NoCopy &) = delete;
7 };
8 NoCopy a;
9 NoCopy b(a); //编译错误, 拷贝构造函数是 deleted 函数
```

4.3.4. 右值引用

新标准的一个最主要的特性是可以移动而非拷贝对象的能力。如我们在 13.1.1 节 (第 440 页) 中所见, 很多情况下都会发生对象拷贝。在其中某些情况下, 对象拷贝后就立即被销毁了。在这些情况下, 移动而非拷贝对象会大幅度提升性能。

为了支持移动操作，新标准引入了一种新的引用类型——**右值引用 (rvalue reference)**。所谓右值引用就是必须绑定到右值的引用。我们通过`&&`而不是`&`来获得右值引用。如我们将要看到的，右值引用有一个重要的性质——只能绑定到一个将要销毁的对象。因此，我们可以自由地将一个右值引用的资源“移动”到另一个对象中。

虽然不能将一个右值引用直接绑定到一个左值上，但我们可以显式地将一个左值转换为对应的右值引用类型。我们还可以通过调用一个名为 **move** 的新标准库函数来获得绑定到左值上的右值引用，此函数定义在头文件 **utility** 中。**move** 函数使用了我们将在 16.2.6 节（第 610 页）中描述的机制来返回给定对象的右值引用。

```
int &&rr3 = std::move(rr1); // ok
```

move 调用告诉编译器：我们有一个左值，但我们希望像一个右值一样处理它。我们必须认识到，调用 **move** 就意味着承诺：除了对 **rr1** 赋值或销毁它外，我们将不再使用它。在调用 **move** 之后，我们不能对移后源对象的值做任何假设。



我们可以销毁一个移后源对象，也可以赋予它新值，但不能使用一个移后源对象的值。

4.3.5. nullptr

nullptr 是一个新的 C++ 关键字，它是空指针常量，它是用来替代高风险的 **NULL** 宏和 0 字面量的。**nullptr** 是强类型的，所有跟指针有关的地方都可以用 **nullptr**，包括函数指针和成员指针：

```
1 void f(int); //#1
2 void f(char *); //#2
3 //C++03
4 f(0); //调用的是哪个 f?
5 //C++11
6 f(nullptr) //毫无疑问，调用的是 #2
7
8 const char *pc=str.c_str(); //data pointers
9 if (pc != nullptr)
10     cout << pc << endl;
11 int (A::*pmf) ()=nullptr; //指向成员函数的指针
12 void (*pmf) ()=nullptr; //指向函数的指针
```

4.3.6. 智能指针类

C++98 定义的唯一的智能指针类 **auto_ptr** 已经被弃用，C++11 引入了新的智能指针类 **shared_ptr** 和 **unique_ptr**。它们都是标准库的其它组件兼容，可以安全地把智能指针存入标准容器，也可以安全地用标准算法“倒腾”它们。

4.4. 动态内存

4.4.1. new 和 malloc

- 属性

new/delete 是 C++关键字，需要编译器支持。malloc/free 是库函数，需要头文件支持 c。

- 参数

使用 new 操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算。而 malloc 则需要显式地指出所需内存的尺寸。

- 返回类型

new 操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，故 new 是符合类型安全性的操作符。而 malloc 内存分配成功则是返回 void *，需要通过强制类型转换将 void*指针转换成我们需要的类型。

- 分配失败

new 内存分配失败时，会抛出 bad_alloc 异常。malloc 分配内存失败时返回 NULL。

- 自定义类型

new 会先调用 operator new 函数，申请足够的内存（通常底层使用 malloc 实现）。然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete 先调用析构函数，然后调用 operator delete 函数释放内存（通常底层使用 free 实现）。malloc/free 是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。

- 重载

C++允许重载 new/delete 操作符，特别的，布局 new 的就不需要为对象分配内存，而是指定了一个地址作为内存起始区域，new 在这段内存上为对象调用构造函数完成初始化工作，并返回此地址。而 malloc 不允许重载。

- 内存区域

new 操作符从自由存储区(free store)上为对象动态分配内存空间，而 malloc 函数从堆上动态分配内存。自由存储区是 C++基于 new 操作符的一个抽象概念，凡是通过 new 操作符进行内存申请，该内存即为自由存储区。而堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配，C 语言使用 malloc 从堆上分配内存，使用 free 释放已分配的对应内存。

在 C++中，内存区分为 5 个区，分别是堆、栈、自由存储区、全局/静态存储区、常量存储区；在 C 中，C 内存区分为堆、栈、全局/静态存储区、常量存

储区：

4.4.2. new 运算符的原理

operator new → malloc

4.4.3. 设计一个内存池

XX

4.4.4. C++内存模型

C++内存分为 5 个区域：

- 堆 heap

由 new 分配的内存块，其释放编译器不去管，由我们程序自己控制（一个 new 对应一个 delete）。如果程序员没有释放掉，在程序结束时 OS 会自动回收。

涉及的问题：“缓冲区溢出”、“内存泄露”

- 栈 stack

是那些编译器在需要时分配，在不需要时自动清除的存储区。存放局部变量、函数参数。

存放在栈中的数据只在当前函数及下一层函数中有效，一旦函数返回了，这些数据也就自动释放了。

- 全局/静态存储区 (.bss 段和.data 段)

全局和静态变量被分配到同一块内存中。在 C 语言中，未初始化的放在.bss 段中，初始化的放在.data 段中；在 C++里则不区分了。

- 常量存储区 (.rodata 段)

存放常量，不允许修改（通过非正当手段也可以修改）

- 代码区 (.text 段)

C++内存分布：

- 带虚函数的类

```
1. class commonClass{
2.     int i;
3.     char c;
4.     virtual void virt_fun1(){}
5.     virtual void virt_fun2(){}
6.     void comm_fun1(){}
7.     void comm_fun2(){}
8. };
```

该类占用 12 个字节，如下图所示：

```
class commonClass size(12):
    +---|
    0 | {vptr}
    4 | i
    8 | c
    | <alignment member> (size=3)
    +---|  
  
commonClass::$vftable@:
    | &commonClass_meta
    | 0
    0 | &commonClass::virt_fun1
    1 | &commonClass::virt_fun2
```

四字节对其 2 个变量共占用 8 个字节，由于存在虚函数，所以在类开始位置插入了一个虚函数指针，该指针占用 4 个字节，共计 12 字节。类的非虚函数其实不占用类对象的内存（函数编译后形成二进制文件放在内存中的代码段区）

● 简单继承

```
1. class baseClass{
2.     int i;
3.     char c;
4. };
5. class inheritClass : public baseClass{
6.     char c;
7. };
```

```
class baseClass size(8):
    +---|
    0 | i
    4 | c
    | <alignment member> (size=3)
    +---|  
  
class inheritClass size(12):
    +---|
    | +--- (base class baseClass)
    0 | | i
    4 | | c
    | | <alignment member> (size=3)
    | +---|
    8 | c
    | <alignment member> (size=3)
    +---|
```

● 基类带有虚函数

```
1. class baseClass{
2. private:
3.     int i;
4.     char c;
5. public:
6.     virtual void virt_fun1(){ cout << "baseClass virt_fun1" << endl; }
7.     virtual void virt_fun2(){ cout << "baseClass virt_fun2" << endl; }
8.     void comm_fun1(){ cout << "baseClass comm_fun1" << endl; }
9.     void comm_fun2(){ cout << "baseClass comm_fun2" << endl; }
10. };
11.
```

```
12. class inheritClass : public baseClass{  
13.     private:  
14.         char c;  
15.     public:  
16.         virtual void virt_fun1(){ cout << "inheritClass virt_fun1" << endl; }  
17.         virtual void virt_fun3(){ cout << "inheritClass virt_fun2" << endl; }  
18.         void comm_fun1(){ cout << "inheritClass comm_fun1" << endl; }  
19.         void comm_fun3(){ cout << "inheritClass comm_fun3" << endl; }  
20.     };
```

基类和派生类使用同一个虚函数表。虚函数表的排列顺序遵循先基类后派生类，由于基类和派生类存在同名的虚函数（virt_fun1），所以派生类的虚函数覆盖了基类的同名虚函数从而排到了最前面

```
class inheritClass  size(16):  
+---  
| +--- (base class baseClass)  
0 | | {vptr}  
4 | | i  
8 | | c  
| | <alignment member> (size=3)  
+---  
12 | c  
| <alignment member> (size=3)  
+---  
  
inheritClass::$vftable@:  
| &inheritClass_meta  
| 0  
0 | &inheritClass::virt_fun1  
1 | &baseClass::virt_fun2  
2 | &inheritClass::virt_fun3
```

- 虚继承/多重继承

<https://www.jianshu.com/p/cc797aa4ace1>

4.4.5. 智能指针的原理、使用、实现

<https://www.cnblogs.com/wxquare/p/4759020.html>

4.4.6. 智能指针出现循环引用

```

void Funtest()
{
    shared_ptr<Node<int>> sp1(new Node<int>(1));
    shared_ptr<Node<int>> sp2(new Node<int>(2));

    cout << "sp1.use_count:" << sp1.use_count() << endl;
    cout << "sp2.use_count:" << sp2.use_count() << endl;

    sp1->_pNext = sp2;
    sp2->_pPre = sp1;

    cout << "sp1.use_count:" << sp1.use_count() << endl;
    cout << "sp2.use_count:" << sp2.use_count() << endl;
}
int main()
{
    Funtest();
    system("pause");
    return 0;
}

```

http://blog.csdn.net/m0_37968340

我们可以看出，并没有调用析构函数，也就是没有对空间进行释放。

从上面 `shared_ptr` 的实现中我们知道了只有当引用计数减为 0 时，析构时才会释放对象，而上述情况造成了一个僵局，那就是析构对象时先析构 `sp2`，可是由于 `sp2` 的空间 `sp1` 还在使用中，所以 `sp2.use_count` 减为 1 后不释放，`sp1` 也是相同的道理，由于 `sp1` 的空间 `sp2` 还在使用中，所以 `sp1.use_count` 减为 1 后也不释放。`sp1` 等着 `sp2` 先释放，`sp2` 等着 `sp1` 先释放，二者互不相让，导致最终都没能释放，内存泄漏。

解决方案：使用弱指针 `weak_ptr`(不增加引用计数)

```

1. weak_ptr<Node<T>> _pPre;
2. weak_ptr<Node<T>> _pNext;

```

4.5. C++中内存泄漏问题

1. `new` 出来的内存没有通过 `delete` 合理的释放掉
2. `new` 创建了一组对象数组，内存回收的时候却只调用了 `delete` 而非 `delete []` 来处理，导致只有对象数组的第一个对象的析构函数得到执行并回收了内存占用，数组的其他对象所占内存得不到回收，导致内存泄露
3. 没有将基类的析构函数定义为虚函数

解决内存泄漏最有效的办法就是使用智能指针（Smart Pointer）。使用智能指针就不用担心这个问题了，因为智能指针可以自动删除分配的内存。

4.6. 成员函数的前后 `const`

在类中将成员函数修饰为 `const`（函数后）表明在该函数体内，不能修改对象的数据成员而且不能调用非 `const` 函数。

`const` 修饰函数返回值，也是用 `const` 来修饰返回的指针或引用，保护指针

指向的内容或引用的内容不被修改，也常用于运算符重载。归根究底就是使得函数调用表达式不能作为左值

4.7. C++有几种转换方法

reinterpret_cast

reinterpret_cast 通常为运算对象的位模式提供较低层次上的重新解释。举个例

第4章 表达式

子，假设有如下的转换



```
int *ip;
char *pc = reinterpret_cast<char*>(ip);
```

我们必须牢记 pc 所指的真实对象是一个 int 而非字符，如果把 pc 当成普通的字符指针使用就可能在运行时发生错误。例如：

static_cast

编译器隐式执行的任何类型转换都可以由 static_cast 显式完成：

```
double d=97.0;
int i=static_cast<int>(d);
```

等价于：

```
double d=97.0;
int i=d;
```

仅当类型之间可隐式转换时（除类层次间的下行转换以外），static_cast 的转换才是合法的，否则将出错。

```
class base{};  
class child:public base{}
```

• 176 •

» 第 1 篇 程序设计基础及数据结构基础

```
base* b;  
child* c;  
c=static_cast<child*>(b); //下行转换，正确  
c=b; //编译不正确
```

但使用 `static_cast` 完成下行转换（把基类指针或引用转换成子类指针或引用），由于没有动态类型检查，所以是不安全的。

const_cast

`const_cast`，顾名思义，将转换掉表达式的 `const` 性质。

```
const char *pc_str;  
char *pc=const_cast<char*>(pc_str);
```

只有使用 `const_cast` 才能将 `const` 性质转换掉。在这种情况下，试图使用其他三种形式的强制转换都会导致编译时的错误。类似地，除了添加或删除 `const` 特性，用 `const_cast` 符来执行其他任何类型转换，都会引起编译错误。

dynamic_cast 运算符 (`dynamic_cast` operator) 的使用形式如下所示：

```
dynamic_cast<type*>(e)  
dynamic_cast<type&>(e)  
dynamic_cast<type&&>(e)
```

其中，`type` 必须是一个类类型，并且通常情况下该类型应该含有虚函数。在第一种形式中，`e` 必须是一个有效的指针（参见 2.3.2 节，第 47 页）；在第二种形式中，`e` 必须是一个左值；在第三种形式中，`e` 不能是左值。

在上面的所有形式中，`e` 的类型必须符合以下三个条件中的任意一个：`e` 的类型是目标 `type` 的公有派生类、`e` 的类型是目标 `type` 的公有基类或者 `e` 的类型就是目标 `type` 的类型。如果符合，则类型转换可以成功。否则，转换失败。如果一条 `dynamic_cast` 语句的转换目标是指针类型并且失败了，则结果为 0。如果转换目标是引用类型并且失败了，

指针类型的 dynamic_cast

举个简单的例子，假定 Base 类至少含有一个虚函数，Derived 是 Base 的公有派生类。如果有一个指向 Base 的指针 bp，则我们可以在运行时将它转换成指向 Derived 的指针，具体代码如下：

```
if (Derived *dp = dynamic_cast<Derived*>(bp))
{
    // 使用 dp 指向的 Derived 对象
} else { // bp 指向一个 Base 对象
    // 使用 bp 指向的 Base 对象
}
```

如果 bp 指向 Derived 对象，则上述的类型转换初始化 dp 并令其指向 bp 所指的 Derived 对象。此时，if 语句内部使用 Derived 操作的代码是安全的。否则，类型转换的结果为 0，dp 为 0 意味着 if 语句的条件失败，此时 else 子句执行相应的 Base 操作。

4.8. 指针和引用的区别

1. 引用不能为空，即不存在对空对象的引用，指针可以为空，指向空对象。
2. 引用必须初始化，指定对哪个对象的引用，指针不需要。
3. 引用初始化后不能改变，指针可以改变所指对象的值。
4. 引用访问对象是直接访问，指针访问对象是间接访问。
5. 不能定义引用数组（声明引用数组没有办法分配空间，因为根本就没有空间可以分配给引用）

4.9. 静态类型获取与动态类型获取

typeid 关键字：

用于获取类型信息， typeid 返回一个 type_info 类对象，当参数是类型时，返回静态类型信息，当参数是变量时：不存在虚函数表，返回静态类型信息；存在虚函数表，返回动态类型信息

dynamic_cast：

用于类继承层次间的指针或引用转换。主要还是用于执行“安全的向下转型（safe downcasting）”，也即是基类对象的指针或引用转换为同一继承层次的其他指针或引用。至于“向上转型”（即派生类指针或引用类型转换为其基类类型），本身就是安全的，尽管可以使用 dynamic_cast 进行转换，但这是没必要的，普通的转换已经可以达到目的，毕竟使用 dynamic_cast 是需要开销的。

4.10. RAII

“资源获取即初始化”，也就是说在构造函数中申请分配资源，在析构函数中释放资源。因为 C++ 的语言机制保证了，当一个对象创建的时候，自动调用构造函数，当对象超出作用域的时候会自动调用析构函数。所以，在 RAII 的指导下，我们应该使用类来管理资源，将资源和对象的生命周期绑定。

智能指针（`std::shared_ptr` 和 `std::unique_ptr`）即 RAII 最具代表的实现，使用智能指针，可以实现自动的内存管理，再也不需要担心忘记 `delete` 造成的内存泄漏。

4.11. 什么函数不能声明为 `virtual`

1. 普通的函数

不能被覆盖，在编译的时候就绑定函数。

2. 构造函数

构造函数无法是虚函数，因为调用虚函数需要虚函数表指针，而在执行构造函数之前是没有虚函数表指针的。

3. 静态成员函数

静态成员函数不可以是虚函数。静态函数是属于类的，不属于对象本身，自然无法有自己的虚函数表指针。

4. 内联函数

内联函数目的是在代码中直接展开（编译期），而虚函数是为了继承后能动态绑定执行自己的动作（动态绑定），因此本质是矛盾的，因此即使内联函数声明为虚函数，编译器遇到这种情况是不会进行 `inline` 展开的，而是当作普通函数来处理。因此声明了虚函数不能实现内敛的，即内敛函数可以声明为虚函数，但是毫无了内联的意义

5. 友员函数

C++不支持友元函数继承，友元函数不属于类的成员函数，不能被继承。

4.12. STL 中的容器

4.12.1. 常见容器

1. 顺序容器：

`vector`: 可变大小数组;

`deque`: 双端队列;

`list`: 双向链表;

`forward_list`: 单向链表;

`array`: 固定大小数组;

`string`: 与 `vector` 相似的容器，但专门用于保存字符。

2. 关联容器：

按关键字有序保存元素：（底层实现为红黑树）

`map`: 关联数组；保存关键字-值对；

`set`: 关键字即值，即只保存关键字的容器；

`multimap`: 关键字可重复的 `map`;

`multiset`: 关键字可重复的 `set`;

3. 无序集合:

`unordered_map`: 用哈希函数组织的 `map`;

`unordered_set`: 用哈希函数组织的 `set`;

`unordered_multimap`: 哈希组织的 `map`; 关键字可以重复出现;

`unordered_multiset`: 哈希组织的 `set`; 关键字可以重复出现。

4. 其他项:

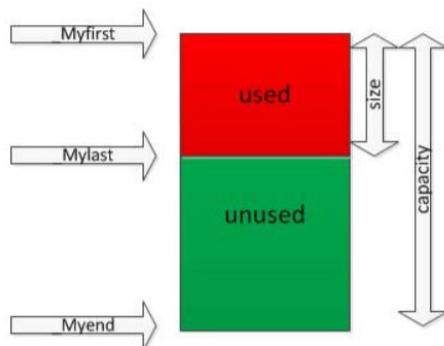
`stack`、`queue`、`valarray`、`bitset`

4.12.2. `vector` 的底层实现

```

1 template<class _Ty,
2       class _Ax>
3 class vector
4     : public _Vector_val<_Ty, _Ax>
5   { //varying size array of values
6   public:
7     /*****
8 protected:
9     pointer _Myfirst; //pointer to beginning of array
10    pointer _Mylast; //pointer to current end of sequence
11    pointer _Myend; //pointer to end of array
12  };

```



两个关键大小:

大小: `size=_Mylast - _Myfirst;`

容量: `capacity=_Myend - _Myfirst;`

分别对应于`resize()`、`reserve()`两个函数。

只有在执行 `insert` 操作时 `size` 与 `capacity` 相等, 或者调用 `resize` 或 `reserve` 时给定的大小超过当前 `capacity`, `vector` 才可能重新分配内存空间。会分配多少超过给定容量的额外空间, 取决于具体实现。

4.12.3. `vector` 与 `list` 的区别

`vector` 就是动态数组. 它也是在堆中分配内存, 元素连续存放, 有保留内存, 如果减少大小后, 内存也不会释放. 如果新值>当前大小时才会再分配内存.

它拥有一段连续的内存空间, 并且起始地址不变, 因此它能非常好的支持随即存取, 即`[]`操作符, 但由于它的内存空间是连续的, 所以在中间进行插入和删除会造成内存块的拷贝, 另外, 当该数组后的内存空间不够时, 需要重新申请一块足够大的内存并进行内存的拷贝。这些都大大影响了 `vector` 的效率。

对最后元素操作最快(在后面添加删除最快), 此时一般不需要移动内存, 只有保留内存不够时才需要

对中间和开始处进行添加删除元素操作需要移动内存, 如果你的元素是结构或是类, 那么移动的同时还会进行构造和析构操作, 所以性能不高 (最好将结构或类的指针放入 `vector` 中, 而不是结构或类本身, 这样可以避免移动时的构造与析构)。

访问方面, 对任何元素的访问都是 $O(1)$, 也就是常数的, 所以 `vector` 常用来保存需要经常进行随机访问的内容, 并且不需要经常对中间元素进行添加删除操作.

`list` 就是双向链表, 元素也是在堆中存放, 每个元素都是放在一块内存中, 它的内存空间可以是不连续的, 通过指针来进行数据的访问, 这个特点使得它的随机存取变的非常没有效率, 因此它没有提供`[]`操作符的重载。但由于链表的特点, 它可以以很好的效率支持任意地方的删除和插入。

`list` 没有空间预留习惯, 所以每分配一个元素都会从内存中分配, 每删除一个元素都会释放它占用的内存.

`list` 在哪里添加删除元素性能都很高, 不需要移动内存, 当然也不需要对每个元素都进行构造与析构了, 所以常用来做随机操作容器.

但是访问 `list` 里面的元素时就开始和最后访问最快, 访问其它元素都是 $O(n)$, 所以如果需要经常随机访问的话, 还是使用其它的好

4.12.4. allocator

`new` 有一些灵活性上的局限，其中一方面表现在它将内存分配和对象构造组合在了一起。类似的，`delete` 将对象析构和内存释放组合在了一起。我们分配单个对象时，通常希望将内存分配和对象初始化组合在一起。因为在这种情况下，我们几乎肯定知道对象应有什么值。

当分配一大块内存时，我们通常计划在这块内存上按需构造对象。在此情况下，我们希望将内存分配和对象构造分离。这意味着我们可以分配大块内存，但只在真正需要时才真正执行对象创建操作（同时付出一定开销）。

一般情况下，将内存分配和对象构造组合在一起可能会导致不必要的浪费。例如：

标准库 `allocator` 类定义在头文件 `memory` 中，它帮助我们将内存分配和对象构造分离开来。它提供一种类型感知的内存分配方法，它分配的内存是原始的、未构造的。表 12.7 概述了 `allocator` 支持的操作。在本节中，我们将介绍这些 `allocator` 操作。在 13.5 节（第 464 页），我们将看到如何使用这个类的典型例子。

类似 `vector`，`allocator` 是一个模板（参见 3.3 节，第 86 页）。为了定义一个 `allocator` 对象，我们必须指明这个 `allocator` 可以分配的对象类型。当一个 `allocator` 对象分配内存时，它会根据给定的对象类型来确定恰当的内存大小和对齐位置：

```
allocator<string> alloc;           // 可以分配 string 的 allocator 对象
auto const p = alloc.allocate(n);   // 分配 n 个未初始化的 string
```

这个 `allocate` 调用为 `n` 个 `string` 分配了内存。

`allocator` 分配的内存是未构造的（`unconstructed`）。我们按需要在此内存中构造对象。在新标准库中，`construct` 成员函数接受一个指针和零个或多个额外参数，在给定位置构造一个元素。额外参数用来初始化构造的对象。类似 `make_shared` 的参数（参见 12.1.1 节，第 401 页），这些额外参数必须是与构造的对象的类型相匹配的合法的初始化器：

```
auto q = p; // q 指向最后构造的元素之后的位置
alloc.construct(q++); // *q 为空字符串
alloc.construct(q++, 10, 'c'); // *q 为 cccccccccc
alloc.construct(q++, "hi"); // *q 为 hi!
```

当我们用完对象后，必须对每个构造的元素调用 `destroy` 来销毁它们。函数 `destroy` 接受一个指针，对指向的对象执行析构函数（参见 12.1.1 节，第 402 页）：

```
while (q != p)
    alloc.destroy(--q); // 释放我们真正构造的 string
```

4.12.5. priority_queue 的底层实现

缺省情况下 `priority_queue` 优先级队列是利用一个 max-heap 最大堆完成，后者是一个以 `vector` 表现的完全二叉树。

如果要用到小顶堆，则一般要把模板的三个参数都带进去。STL 里面定义

了一个仿函数 `greater<>`，对于基本类型可以用这个仿函数声明小顶堆

4.13. 关键字

1. `extern “C”`

作为面向对象的语言，C++为了支持函数重载，函数在被 C++ 编译后在符号库中的名字与 C 语言的不同。假如某个函数的原型为 `void foo(int x, int y);`；该函数被 C 编译器编译后在符号库中的名字为 `_foo`，而 C++ 编译器则会产生 `_foo_int_int` 之类的名字。`_foo_int_int` 这样的名字是包含了函数名以及形参，C++ 就是靠这种机制来实现函数重载的。

被 `extern “C”` 修饰的函数或者变量是按照 C 语言方式编译和链接的，所以可以用一句话来概括 `extern “C”` 的真实目的：实现 C++ 与 C 的混合编程。

2. `sizeof`

首先空类的大小 1，这是因为在实例化的过程中每个实例在内存中都得有一个独一无二的地址，为了达到这个目的，编译器会给空类隐含加一个字节

对于有虚函数的空类，该函数在基本的虚函数表中占用一项，因此 `sizeof(nullClassVirtual)` 大小为 4

对于类中包含 `static/ static const` 成员的类，该静态成员不影响类的大小，该成员只有一个实例存在，无论类是否被实例化

3. `volatile`

定义为 `volatile` 的变量是说这变量可能会被意想不到地改变，即在你程序运行过程中一直会变，你希望这个值被正确的处理，每次从内存中去读这个值，而不是因编译器优化从缓存的地方读取，比如读取缓存在寄存器中的数值，从而保证 `volatile` 变量被正确的读取。

在多任务环境中，虽然在一个函数体内部，在两次读取变量之间没有对变量的值进行修改，但是该变量仍然有可能被其他的程序（如中断程序、另外的线程等）所修改。如果这时还是从寄存器而不是从 RAM 中读取，就会出现被修改了的变量值不能得到及时反应的问题。

4. `static`

全局变量与 `static` 变量：

者在存储方式上并无不同。这两者的区别在于非静态全局变量的作用域是整个源程序，当一个源程序由多个原文件组成时，非静态的全局变量在各个源文件中都是有效的。而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其它源文件中不能使用它

static 函数与普通函数：

作用域不同，仅在本文件。只在当前源文件中使用的函数应该说明为内部函数(static 修饰的函数)，内部函数应该在当前源文件中说明和定义。对于可在当前源文件以外使用的函数，应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件

5. inline

inline 只适合函数体内代码简单的函数使用；

inline 函数仅仅是一个对编译器的建议；

定义在类中的成员函数缺省都是内联的；

inline 函数（即内联函数）对编译器而言必须是可见的，以便能够在调用点展开该函数，与非 inline 函数不同的是，inline 函数必须在调用该函数的每个文件中定义

6. const 和 #define

const 可以明确指定类型，而宏定义没有数据类型。编译器可以对 const 进行类型安全检查，而宏定义只是简单的字符替换，有时候会产生意想不到的错误。

可以使用 C++ 的作用域规则将定义限制在特定的函数或是文件中。在默认的情况下，全局变量的链接性为外部的，但 const 全局变量的链接性为内部的。

const 更加方便调试

4.14. c++里面的同步和互斥怎么实现的

std::mutex

Defined in header <`<mutex>`>
`class mutex;` (since C++11)

The mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads.

mutex offers exclusive, non-recursive ownership semantics:

- A calling thread *owns* a mutex from the time that it successfully calls either `lock` or `try_lock` until it calls `unlock`.
- When a thread owns a mutex, all other threads will block (for calls to `lock`) or receive a `false` return value (for `try_lock`) if they attempt to claim ownership of the mutex.
- A calling thread must not own the mutex prior to calling `lock` or `try_lock`.

The behavior of a program is undefined if a mutex is destroyed while still owned by any threads, or a thread terminates while owning a mutex. The mutex class satisfies all requirements of `Mutex` and `StandardLayoutType`.

`std::mutex` is neither copyable nor movable.

std::unique_lock

Defined in header `<mutex>`

`template< class Mutex > class unique_lock;` (since C++11)

The class `unique_lock` is a general-purpose mutex ownership wrapper allowing deferred locking, time-constrained attempts at locking, recursive locking, transfer of lock ownership, and use with condition variables.

The class `unique_lock` is movable, but not copyable -- it meets the requirements of `MoveConstructible` and `MoveAssignable` but not of `CopyConstructible` or `CopyAssignable`.

The class `unique_lock` meets the `BasicLockable` requirements. If `Mutex` meets the `Lockable` requirements, `unique_lock` also meets the `Lockable` requirements (ex.: can be used in `std::lock`); if `Mutex` meets the `TimedLockable` requirements, `unique_lock` also meets the `TimedLockable` requirements.

std::condition_variable

Defined in header `<condition_variable>`

`class condition_variable;` (since C++11)

The `condition_variable` class is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, until another thread both modifies a shared variable (the `condition`), and notifies the `condition_variable`.

The thread that intends to modify the variable has to

1. acquire a `std::mutex` (typically via `std::lock_guard`)
2. perform the modification while the lock is held
3. execute `notify_one` or `notify_all` on the `std::condition_variable` (the lock does not need to be held for notification)

Even if the shared variable is atomic, it must be modified under the mutex in order to correctly publish the modification to the waiting thread.

Any thread that intends to wait on `std::condition_variable` has to

1. acquire a `std::unique_lock<std::mutex>`, on the same mutex as used to protect the shared variable
2. execute `wait`, `wait_for`, or `wait_until`. The wait operations atomically release the mutex and suspend the execution of the thread.
3. When the condition variable is notified, a timeout expires, or a spurious wakeup occurs, the thread is awakened, and the mutex is atomically reacquired. The thread should then check the condition and resume waiting if the wake up was spurious.

`std::condition_variable` works only with `std::unique_lock<std::mutex>`; this restriction allows for maximal efficiency on some platforms. `std::condition_variable_any` provides a condition variable that works with any `BasicLockable` object, such as `std::shared_lock`.

Condition variables permit concurrent invocation of the `wait`, `wait_for`, `wait_until`, `notify_one` and `notify_all` member functions.

The class `std::condition_variable` is a `StandardLayoutType`. It is not `CopyConstructible`, `MoveConstructible`, `CopyAssignable`, or `MoveAssignable`.

多个线程按顺序执行：

```
1. std::mutex data_mutex;
2. std::condition_variable data_var;
3. bool flag = true;
4.
5. void printA(){
6.     while(1){
7.         std::this_thread::sleep_for(std::chrono::seconds(1));
8.         std::unique_lock<std::mutex> lck(data_mutex) ;
9.         data_var.wait(lck,[&]{return flag;});
10.        std::cout<<"thread: "<< std::this_thread::get_id() << "  printf: " << "A" <<std::endl;
11.        flag = false;
12.        data_var.notify_one();
13.    }
14. }
15.
16. void printB(){
17.     while(1){
```

```
18.     std::unique_lock<std::mutex> lck(data_mutex) ;
19.     data_var.wait(lck,[&]{return !flag;});
20.     std::cout<<"thread: "<< std::this_thread::get_id() << "  printf: " << "B" <<std::endl;
21.     flag = true;
22.     data_var.notify_one();
23. }
24. }
25.
26. int main(){
27.     std::thread tA(printA);
28.     std::thread tB(printB);
29.     tA.join();
30.     tB.join();
31.     return 0;
32. }
```

4.15. c++怎么实现一个函数先于 main 函数运行

定义在 main()函数之前的全局对象、静态对象的构造函数在 main()函数之前执行

4.16. 拷贝构造函数必须为引用传递

当一个对象需要以值方式传递时，编译器会生成代码调用它的拷贝构造函数以生成一个复本。如果类 A 的拷贝构造函数是以值方式传递一个类 A 对象作为参数的话，当需要调用类 A 的拷贝构造函数时，需要以值方式传进一个 A 的对象作为实参；而以值方式传递需要调用类 A 的拷贝构造函数；结果就是调用类 A 的拷贝构造函数导致又一次调用类 A 的拷贝构造函数，这就是一个无限递归

4.17. 类模板和函数模板区别

模板定义以关键字 `template` 开始，后接模板形参表，模板形参表是用尖括号括住的一个或多个模板形参的列表，形参之间以逗号分隔。模板形参表不能为空。

模板形参表很像函数形参表，函数形参表定义了特定类型的局部变量但并不初始化那些变量，在运行时再提供实参来初始化形参。同样，模板形参表示可以在类或函数的定义中使用的类型或值。

例如，`callWithMax` 函数声明一个名为 T 的类型形参。在 `callWithMax` 内部，可以使用名字 T 引用一个类型，T 表示哪个实际类型由编译器根据所用的函数参数而确定。

```
template <typename T>
void callWithMax(const T &a, const T &b) {
    f(a>b ? a : b);
}
```

类型形参 T 跟在关键字 `class` 或 `typename` 之后定义，在这里 `class` 和 `typename` 没有区别。模板形参可以是表示类型的类型形参，也可以是表示常量表达式的非类型形参。模板形参选择的名字没有本质含义。可以给模板形参赋予的唯一含义是区别形参是类型形参还是非类型形参。如果是类型形参，我们就知道该形参表示未知类型，如果是非类型形参，我们就知道它是一个未知值。**模板非类型形参是模板定义内部的常量值。**

就像可以定义函数模板一样，也可以定义类模板。如下：

```
template <class Type> class Queue {  
public:  
    Queue (); // default constructor  
    Type &front (); // return element from head of Queue  
    const Type &front () const;  
    void push (const Type &); // add element to back of Queue  
    void pop(); // remove element from head of Queue  
    bool empty() const; // true if no elements in the Queue  
private:  
    // ...  
};
```

使用类模板时，必须为模板形参显式指定实参：

```
Queue<int> qi;
```

编译器使用实参来实例化这个类的特定类型版本，即编译器用用户提供的实际特定类型代替 Type，重新编写 Queue 类。

4.18. 变量的初始化顺序

1. 基类的静态变量或全局变量
2. 派生类的静态变量或全局变量
3. 基类的成员变量
4. 派生类的成员变量
5. 注意：
 6. 成员变量在使用初始化列表初始化时，与构造函数中初始化成员列表的顺序无关，只与定义成员变量的顺序有关。
 7. 如果不使用初始化列表初始化，在构造函数内初始化时，此时与成员变量在构造函数中的位置有关。
 8. 类中 const 成员常量必须在构造函数初始化列表中初始化。
 9. 类中 static 成员变量，必须在类外初始化。

4.19. 标准库

1. `shrink_to_fit();`

有一种方法来把它从曾经最大的容量减少到它现在需要的容量。这样减少容量的方法常常被称为“收缩到合适 (`shrink_to_fit`)”。该方法只需一条语句：

```
vector<int>(ivec).swap(ivec);
```

表达式 `vector<int>(ivec)` 建立一个临时 vector，它是 `ivec` 的一份拷贝：vector 的拷贝构造函数做了这个工作。但是，vector 的拷贝构造函数只分配拷贝的元素需要的内存，所以这个临时 vector 没有多余的容量。然后我们让临时 vector 和 `ivec` 交换数据，这时我们完成了，`ivec` 只有临时变量的修整过的容量，而这个临时变量则持有曾经在 `ivec` 中的没用到的过剩容量。在这里（这个语句结尾），临时 vector 被销毁，因此释放了以前 `ivec` 使用的内存，收缩到合适

4.20. 实现常见函数

4.20.1. memcpy 的实现

memcpy 和 memmove 他们的作用是一样的，唯一的区别是，当内存发生局部重叠的时候，memmove 保证拷贝的结果是正确的，memcpy 不保证拷贝的结果的正确，以下为考虑内存重叠后的实现。

```

1. void *memcpy(void *dst, const void *src, size_t len) {
2.     if (NULL == dst || NULL == src) {
3.         return NULL;
4.     }
5.     void *ret = dst;
6.     if (dst <= src || (char *)dst >= (char *)src + len){ //没有内存重叠，从低地址
开始复制
7.         while (len--) {
8.             *(char *)dst = *(char *)src;
9.             dst = (char *)dst + 1;
10.            src = (char *)src + 1;
11.        }
12.    }
13.    else { //有内存重叠，从高地址开始复制
14.        src = (char *)src + len - 1;
15.        dst = (char *)dst + len - 1;
16.        while (len--) {
17.            *(char *)dst = *(char *)src;
18.            dst = (char *)dst - 1;
19.            src = (char *)src - 1;
20.        }
21.    }
22.    return ret;
23. }
```

4.20.2. strcpy 的实现

```

1. char *strcpy(char *strDest, const char *strSrc){
2.     assert((strDest != NULL) && (strSrc != NULL));
3.     char *ret = strDest;
4.     char *d, *s;
5.     int size = 0;
6.     d = (char *)strDest;
7.     s = (char *)strSrc;
8.
9.     size = strlen(strSrc) + 1;
10.    if (strDest >= strSrc && strDest <= strSrc + size) {
11.        d += size - 1;
12.        s += size - 1;
13.        while (size--) {
14.            *d-- = *s--;
15.        }
16.    }
17.    else {
18.        while (size--)
```

```

19.           *strDest++ = *strSrc++;
20.       }
21.   return ret;
22. }
```

4.20.3. 手写 strncpy

strncpy 把源字符串的字符复制到目标数组。然而，它总是正好向 dst 写入 len 个字符。如果 strlen(src) 的值小于 len，dst 数组就用

额外的 NULL 字节填充到 len 长度，如果 strlen(src) 的值大于或等于 len，那么只有 len 个字符被复制到 dst 中。注意！它的结果将不会以 NUL 字节结尾。

不考虑内存重叠的写法：

```

1. char * my_strncpy(char *strDest, const char *strSrc, int num)
2. {
3.     assert((strDest != NULL) && (strSrc != NULL));
4.     //if (strDest == NULL || strSrc == NULL) return NULL;
5.
6.     //保存目标字符串的首地址
7.     char *strDestcopy = strDest;
8.     while ((num--)&&(*strDest++ = *strSrc++) != '\0');
9.     //如果 num 大于 strSrc 的字符个数，将自动补'\0'
10.    if (num > 0)
11.    {
12.        while(--num)
13.        {
14.            *strDest++ = '\0';
15.        }
16.    }
17.    return strDestcopy;
18. }
```

5. 智力

5.1. 4 刀把一个圆柱形蛋糕切 16 块

第一刀，切成 2 块；第 2 刀，把 2 块排列放置，变成 4 块；第三刀，把 4 块排列放置，变成 8 块；第 4 刀，把 8 块排列放置，变成 16 块

5.2. 有 2 个盒子，要把 50 个红球和 50 个蓝球全放完，从 2 个盒子里抽出 1 个球，怎样才能使抽到红球的可能性大

把一个红球单独放到一个盒子，另外 49 个红球和 50 个篮球放在一个盒子，这样摸到红球的概率就是：单独放红球的盒子的概率是 0.5，另外放 49 个红球和 50 个篮球的盒子的概率是 $0.5 \times [49/(49+50)]$ ，两个盒子摸到红球合起来的概率就是 $148/198$ ，约为 0.75

6. 其他

1. git pull 和 git fetch 的区别

git pull：首先比对本地的 FETCH_HEAD 记录与远程仓库的版本号，然后 git fetch 获得当前指向的远程分支的后续版本的数据，然后再利用 git merge 将其与本地的当前分支合并。所以可以认为 git pull 是 git fetch 和 git merge 两个步骤的结合

2. 单例模式
3. git 的 pull 、 push 和 merge 的联系和区别
4. 手写单例

xx

- 5.