

# Linux 网络编程笔记

郑华

2018 年 5 月 13 日



# 目录

<b>第一章 架构篇</b>	<b>15</b>
1.1 架构篇 . . . . .	15
1.1.1 队列 + 连接池 . . . . .	15
1.1.2 缓存 . . . . .	15
1.1.3 数据库读写分离 . . . . .	16
1.1.4 数据分区 . . . . .	16
1.1.5 应用服务器的负载均衡 . . . . .	16
1.1.6 服务器性能的四大杀死 . . . . .	16
1.2 架构演化流程 . . . . .	17
1.2.1 服务器与数据库分离 . . . . .	17
1.2.2 缓存处理 . . . . .	18
1.2.3 服务器集群 + 读写分离 . . . . .	18
1.2.4 负载均衡 . . . . .	18
1.2.5 CDN、分布式缓存、分库分表 . . . . .	19
1.2.6 多数据中心 + 分布式存储与计算 . . . . .	19
<b>第二章 进程与线程篇</b>	<b>21</b>
2.1 进程、线程-比对 . . . . .	21
<b>第三章 TCP/IP 基础篇</b>	<b>23</b>

3.1	层次结构与关系	23
3.1.1	OSI-ISO 7 层模型	23
3.1.2	TCP 4 层模型与 OSI 7 层模型的关系	24
3.1.3	端口	24
3.2	数据链路层相关知识	25
3.2.1	最大传输单元 (MTU)/路径 (MTU)	25
3.2.2	以太网帧格式	25
3.2.3	ARP-地址解析协议：将 IP 地址解析到 MAC 地址	25
3.2.4	RARP-反地址解析协议：将 MAC 地址解析到 IP 地址	25
3.3	网络 (IP) 层相关知识	25
3.3.1	ICMP	25
3.3.2	IP 数据报格式	26
3.3.3	网际校验和	26
3.3.4	路由	26
3.4	传输层相关知识：TCP/UDP	26
3.4.1	TCP	26
3.4.2	UDP	27
<b>第四章</b>	<b>socket 编程篇</b>	<b>29</b>
4.1	基本概念	29
4.1.1	socket	29
4.1.2	网络字节序	30
4.1.3	套接字类型	31
4.1.4	流式套接字的粘包问题	31
4.1.5	获取本机网络地址	31
4.2	基本的 C/S 通信模型	32

4.2.1	通信模型	32
4.2.2	客户端使用的函数及作用	32
4.2.3	服务器使用的函数及作用	33
4.2.4	REUSEADDR	35
4.2.5	close 与 shutdown 的区别	35
4.2.6	FIN 与 RST	36
4.2.7	SIGPIPE	37
4.2.8	TCP No Delay	37
4.2.9	多进程服务器-允许多 client 访问	37
4.3	Peer To Peer 点对点通信: P2P	39
4.3.1	信号机制	39
4.3.2	服务器端	39
4.3.3	客户端	41
4.4	fork 进程	41
4.5	僵尸进程	42
4.5.1	参考	42
4.5.2	wait 与 waitpid	42
4.6	孤儿进程	43
4.7	守护进程	43
4.7.1	守护进程的输出	43
4.7.2	守护进程的启动方法有	44
4.7.3	分类	44
4.7.4	UID,EUID	45
4.8	TCP	45
4.8.1	11 种状态	45

4.8.2	3 次握手: 建立链接	47
4.8.3	4 次握手: 终止链接	49
4.9	套接字选项	52
4.9.1	相关函数	52
4.9.2	低水位标记	53
4.9.3	高水位标记	53
4.9.4	SO_KEEPALIVE	54
4.9.5	SO_LINGER	55
4.9.6	SO_RCVBUF	56
4.9.7	SO_SNDBUF	56
4.9.8	SO_RCVLOWAT	57
4.9.9	SO_SNDLOWAT	57
4.9.10	SO_RCVTIMEO	57
4.9.11	SO_SNDTIMEO	57
4.9.12	SO_REUSEADDR	57
4.9.13	SO_BROADCAST	58
4.9.14	SO_DEBUG	58
4.9.15	SO_OOBINLINE	58
4.9.16	TCP_MAXSEG	58
4.9.17	TCP_NODELAY	58
4.10	带外数据-Urgent 标志	58
4.11	心搏函数-弥补 keepAlive2 小时等待时间	59
4.12	高级 IO 函数	59
4.12.1	创建文件描述符系列	59
4.12.2	读写数据系列	62

4.12.3 控制 IO 行为和属性系列	63
4.13 IPC	63
4.13.1 IPC 简介	63
4.13.2 信号	63
4.13.3 消息传递	64
4.13.4 信号量	64
4.13.5 共享内存	66
4.14 阻塞与非阻塞、同步与异步	69
4.14.1 同步/异步主要针对 C 端	69
4.14.2 阻塞/非阻塞主要针对 S 端	69
4.14.3 Linux 下的 5 种 I/O 模型	70
4.14.4 参考	75
4.15 Select 模型：管理多个 IO	75
4.15.1 相关函数	75
4.15.2 使用 select 实现回射服务程序	78
4.16 Poll 模型：不再限制最大描述符值	81
4.16.1 相关函数	81
4.16.2 实例	82
4.17 Epoll 模型：实现更高效率	82
4.17.1 相关函数	82
4.17.2 使用 epoll 实现回射服务器	83
4.17.3 Epoll 触发方式	89
4.18 select、poll、epoll 比较	90
4.18.1 select	90
4.18.2 poll	90

4.18.3	epoll	91
4.18.4	区别对比	92
4.19	UDP	93
4.20	套接字与流之间的转换	93
4.20.1	转换	93
4.20.2	缓冲	93
4.21	非阻塞 socket	93
4.22	Unix 域套接字编程：本地进程间通信	93
4.23	HTTP	94
4.23.1	概要	94
4.23.2	技术架构	94
4.24	提高性能的措施	94
4.24.1	池	94
4.24.2	数据拷贝	95
4.24.3	锁与上下文切换	95
<b>第五章</b>	<b>负载均衡</b>	<b>97</b>
5.1	负载均衡	97
5.1.1	概述	97
5.1.2	设计思路	98
5.1.3	Nginx	98
5.1.4	LVS	98
5.2	群集技术	98
5.2.1	集群系统	98
5.2.2	服务器集群	98
5.2.3	RPC	98



5.2.4 分布式系统 . . . . .	98
<b>第六章 常见服务器模型</b>	<b>99</b>
6.1 阻塞 IO 服务器模型之单线程服务器模型 . . . . .	99
6.2 并发式模式 . . . . .	102
6.3 pre-Fork or pre-Threaded . . . . .	103
6.4 ReActor 模式 . . . . .	104
6.4.1 例子学习 ReActor . . . . .	105
6.4.2 reActor + thread per request . . . . .	106
6.4.3 参考 . . . . .	106
6.5 reactor + thread pool . . . . .	107
6.6 multiple reactors . . . . .	108
6.7 multiple reactors + thread pool . . . . .	109
6.8 ProActor . . . . .	110
6.9 Reactor with Proactor . . . . .	111
6.9.1 reactor . . . . .	111
6.9.2 proactor . . . . .	111
6.9.3 区别 . . . . .	111
6.10 半同步半异步模型 . . . . .	112
6.10.1 同步 . . . . .	112
6.10.2 异步 . . . . .	112
6.10.3 半同步半异步 . . . . .	112
6.11 领导者追随者模式 . . . . .	114
<b>第七章 使用多台机器并行处理数据</b>	<b>115</b>
7.1 集群服务器 . . . . .	115

7.2 分布式服务器 . . . . .	115
<b>第八章 网络库</b>	<b>117</b>
8.1 muduo . . . . .	117
8.1.1 bind/function . . . . .	117
8.1.2 面向对象的编程 . . . . .	117
8.1.3 基于对象的编程 . . . . .	118
8.1.4 具体细节 . . . . .	119
<b>第九章 Redis with Memcache</b>	<b>121</b>
9.1 NoSQL . . . . .	121
9.1.1 解决问题 . . . . .	121
9.1.2 四大分类 . . . . .	121
9.2 Memcache with Redis . . . . .	122
9.2.1 Memcache . . . . .	122
9.2.2 Redis . . . . .	122
9.3 Memcache . . . . .	123
<b>第十章 编译</b>	<b>125</b>
10.1 常规 . . . . .	125
10.2 Makefile . . . . .	125
10.3 CMake . . . . .	125
10.3.1 流程 . . . . .	125
10.3.2 单目录结构 . . . . .	125
10.3.3 多目录结构 . . . . .	126
10.3.4 使用其他链接库 . . . . .	127

<b>第十一章 调试</b>	<b>129</b>
11.1 core 文件 . . . . .	129
11.1.1 什么是 Core Dump . . . . .	129
11.1.2 如何使用 core 文件 . . . . .	129
11.1.3 为什么没有 core 文件生成 . . . . .	129
11.1.4 如何定位到行 . . . . .	130
11.2 ACK 与 Seq . . . . .	131
11.2.1 建立与释放阶段 . . . . .	131
11.2.2 数据传输阶段 . . . . .	131
11.3 strace . . . . .	132
11.3.1 指定追踪类型 . . . . .	132
11.4 tcpdump . . . . .	133
11.5 nc . . . . .	133
11.6 lsof . . . . .	133
11.7 netstat . . . . .	133
11.8 vmstat . . . . .	133
11.9 ifstat . . . . .	133
11.10 mpstat . . . . .	134
11.11 top . . . . .	134
11.12 ping . . . . .	134
11.13 iptables . . . . .	134
11.14 TTCP . . . . .	134
<b>第十二章 理论</b>	<b>135</b>
12.1 网络的性能 . . . . .	135
12.1.1 速率 . . . . .	135

12.1.2	带宽	135
12.1.3	吞吐量	135
12.1.4	时延	135
12.1.5	时延带宽积	136
12.1.6	往返时间 RTT	136
12.1.7	利用率	137
12.2	应用层	137
12.2.1	域名系统 DNS	137
12.2.2	文件传送 FTP	137
12.2.3	万维 WWW	138
12.2.4	动态主机配置 DHCP	138
12.2.5	电子邮件 SMTP IMAP	138
12.2.6	简单网络管理 SNMP	138
12.2.7	APP 分别使用传输层的协议对照	138
12.3	传输层 TCP	138
12.3.1	基本概念对照	138
12.3.2	UDP	140
12.3.3	TCP	141
12.4	网络层 IP	141
12.4.1	分类的 IP 地址	141
12.4.2	子网掩码	142
12.5	数据链路层 Hardware	143

## 第十三章 资源 145

13.1	读书路线	145
13.2	项目学习	145

13.3 参考 . . . . .	145
-------------------	-----



# 第一章 架构篇

## 1.1 架构篇

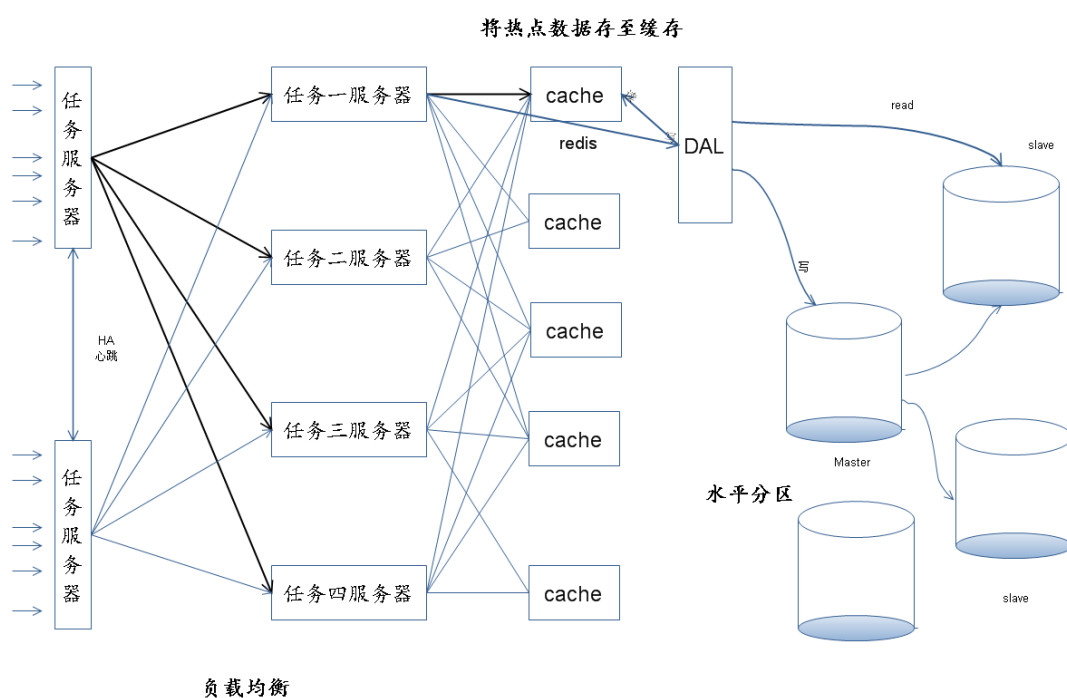


图 1.1: 基本架构

### 1.1.1 队列 + 连接池

主要的业务逻辑挪到应用服务器处理，数据库只做辅助的业务处理，解决数据库的瓶颈问题。

### 1.1.2 缓存

缓存更新（缓存同步）。缓存 time out

如果缓存失效，重新去数据库查询，实时性比较差，替而代之的，一旦数据库中的数据更新，立即通知前端的缓存更新，实时性较高。

缓存换页内存不够，将不活跃的数据换出内存。常用的有以下算法（FIFO 先进先出, LRU 最近最少使用, LFU 最不频繁使用）

no-sql

分布式缓存-redis

### 1.1.3 数据库读写分离

将数据库的读操作与写操作分离

对数据库进行负载均衡 replication 机制

### 1.1.4 数据分区

分库垂直分区分表水平分区（常用）

### 1.1.5 应用服务器的负载均衡

增加一个任务服务器来实现，任务服务器可以监视应用服务器的负载，cpu 高，IO 高，并发高，内存换页高，查询到这些信息后选取负载最低的服务器分配任务。应用服务器被动接受任务。

理想状态... 应用服务器主动到任务服务器接受任务进行处理。

### 1.1.6 服务器性能的四大杀死

- 数据拷贝缓存
- 环境切换理性创建多线程
- 内存分配内存池
- 锁竞争



# 1.2 架构演化流程

## 1.2.1 服务器与数据库分离

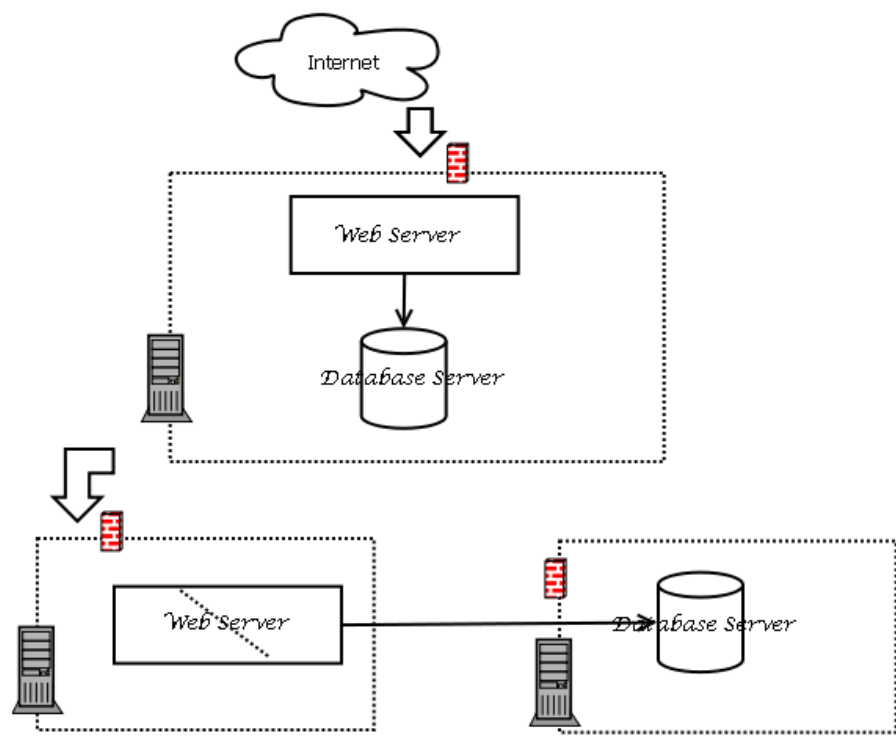


图 1.2: web server 与数据库分离

## 动静分离

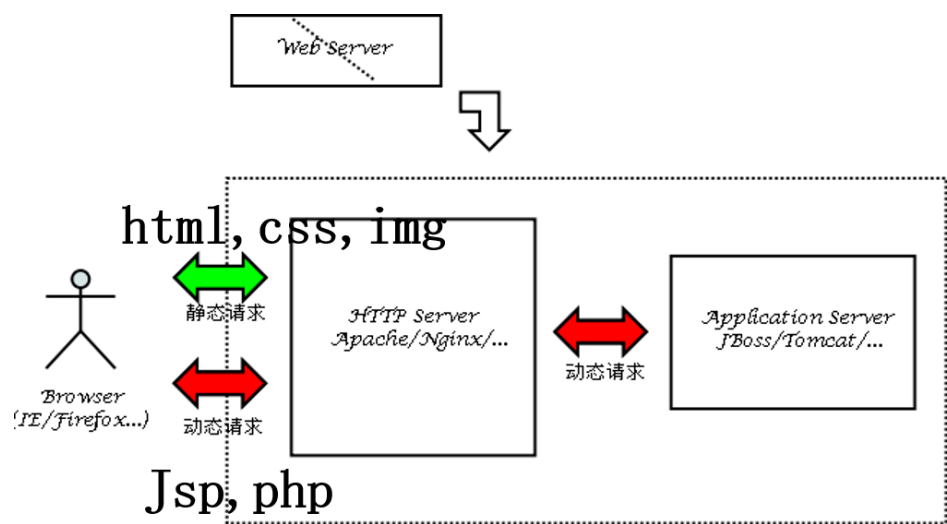


图 1.3: 动态请求与静态请求分离

### 1.2.2 缓存处理

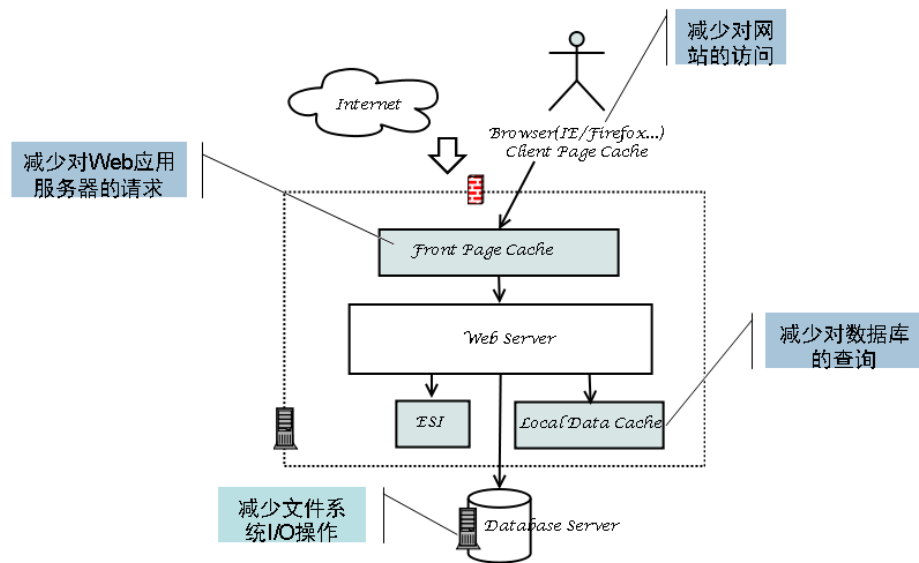


图 1.4: 缓存

### 1.2.3 服务器集群 + 读写分离

### 1.2.4 负载均衡

#### 前端负载均衡

- DNS 负载均衡：在 DNS 服务器中，可以为多个不同的地址配置同一个名字，对于不同的客户机访问同一个名字，得到不同的地址。
- 反向代理：使用代理服务器将请求发给内部服务器，让代理服务器将请求均匀转发给多台内部 web 服务器之一，从而达到负载均衡的目的。标准代理方式是客户端使用代理访问多个外部 Web 服务器，而这种代理方式是多个客户端使用它访问内部 Web 服务器，因此也被称为反向代理模式。
- 基于 NAT 的负载均衡
- LVS
- F5 硬件负载均衡技术

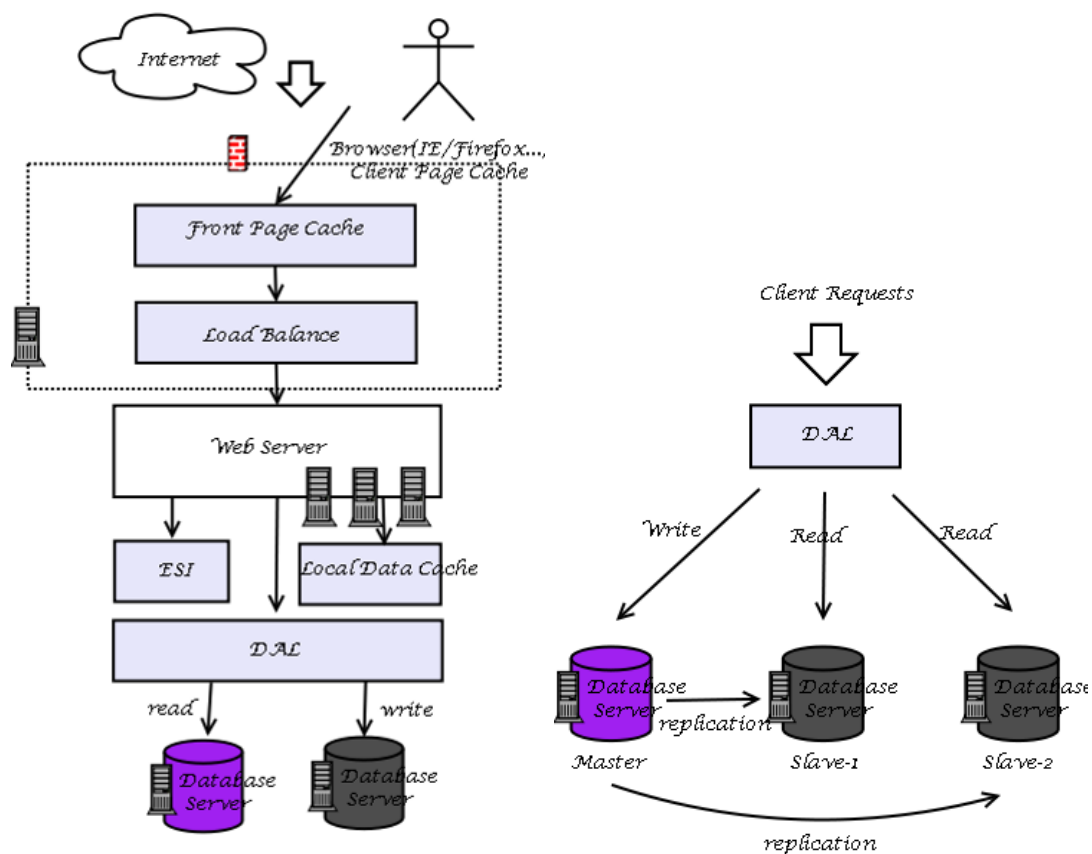


图 1.5: 读写分离

应用服务器负载均衡

数据库负载均衡

### 1.2.5 CDN、分布式缓存、分库分表

CDN

内容分割网络

分布式缓存

目前流行分布式缓存方案：memcached、membase、redis 等，基本上当前的 NoSQL 方案都可以用来做分布式缓存方案

### 1.2.6 多数据中心 + 分布式存储与计算

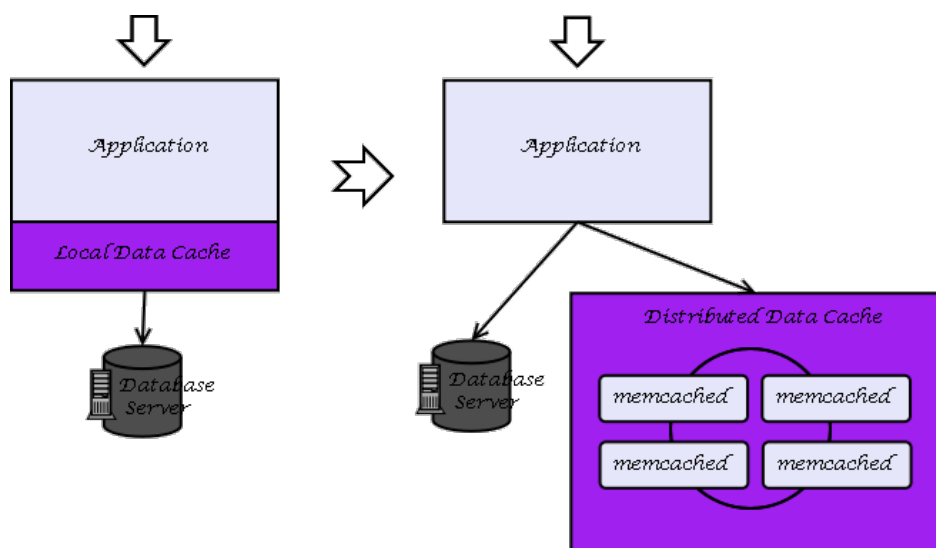


图 1.6: 分布式缓存

## 第二章 进程与线程篇

### 2.1 进程、线程-比对

每个人都有自己的记忆 (Memory)，人与人通过谈话 (消息传递) 来交流，谈话即可以面谈 (同一台服务器)，也可以电话里谈 (不同的服务器、有网络通信)。面谈和电话谈的区别在于，面谈可以立即知道对方是否死了，而电话谈只能通过周期性的心跳来判断对方是否还活着。

有了这些比喻，设计分布式系统时可以采取“角色扮演”，团队里的几个人各自扮演一个进程，人的角色由进程的代码决定 (管登陆的、管消息分发的、管买卖的等)。每个人有自己的记忆，但不知道别人的记忆，要想知道别人的看法，只能通过交谈 (暂不考虑共享内存这种方式)，然后就可以思考：

- **容错**：万一有人突然死了
- **扩容**：新人中途加进来
- **负载均衡**：把甲的活儿挪给乙做
- **退休**：甲要修复 Bug, 先别派新任务，等他做完手上的事情就把他重启

等等场景，十分便利。

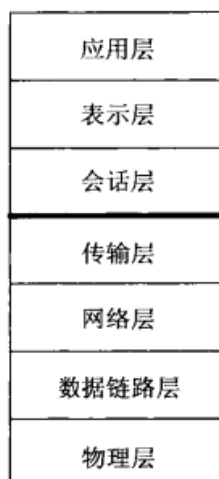
线程的特点是共享地址空间，从而可以高效的共享数据。一台机器上的多个进程能高效的共享代码段，但不能共享数据。如果多个进程大量共享内存，等于是把多进程程序当成多线程来写，掩耳盗铃。



## 第三章 TCP/IP 基础篇

### 3.1 层次结构与关系

#### 3.1.1 OSI-ISO 7 层模型

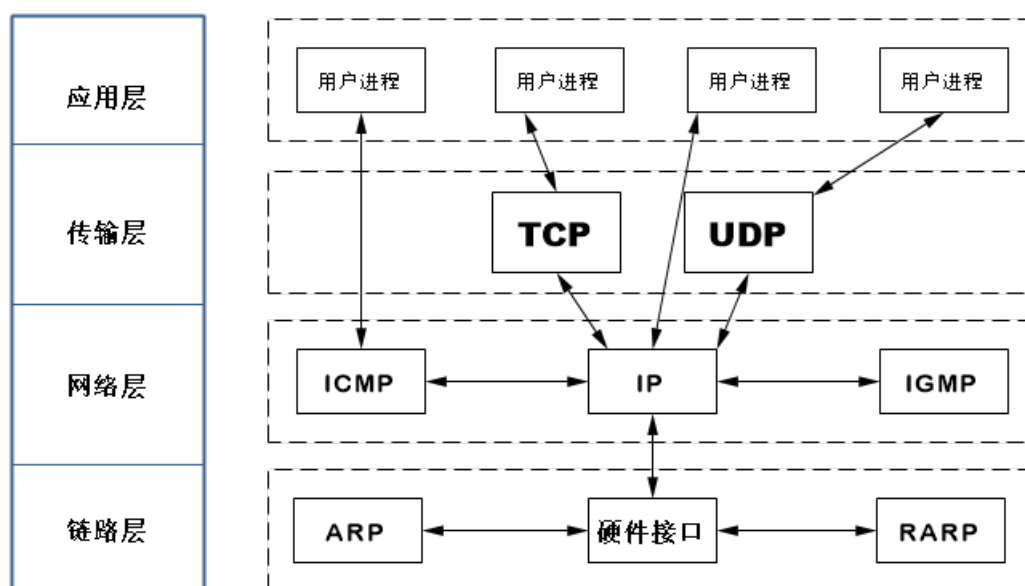


当然，每个层有每个层的功能，每个对等层之间传输的单位是 PDU(propoal Data Unit)，也有每个对等层之间的协议（如：应用与应用的协议），具体功能见下：

1. **应用层**：提供应用程序间通信，Application-PDU，应用层与应用程序界面沟通，以达到展示给用户的目的。在此常用的协议有 **HTTP**, **HTTPS**, **FTP**, **TELNET**, **SSH**, **SMTP**, **POP3** 等
2. **表示层**：处理数据格式、数据加密等，Presentation-PDU，对网络传输的数据进行交换，使得多个主机之间传送的信息能够互相理解，包括数据的压缩、加密、格式转换等
3. **会话层**：建立、维护和管理会话，Session-pdu，管理主机之间会话过程，包括会话建立、终止和绘画过程中的管理
4. **传输层**：建立端到端链接，Segment，提供可靠的数据传输服务，它检测路由器丢弃的包，然后产生一个重传请求，能够将乱序收到的数据包重新排序

5. **网络层**：寻址和路由选择，数据包 Packet，负责将各个子网之间的数据进行路由选择，分组与重组，属于本层的协议有 *IP, IPX, RIP, ICMP, IGMP* 等，实际使用中的设备如路由器属于本层
6. **数据链路层**：介质访问，链路管理，帧 Frame，对物理层的比特流进行数据成帧。提供可靠的数据传输服务，实现无差错数据传输，属于本层的规范有 *SDLC, PPP, STP*，帧中断等，实际使用中的设备如 *switch* 交换机属于本层
7. **物理层**：比特流传输，比特 Bit，定义所有电子及设备的规范，为上层的传输提供一个物理介质，属于本层的协议有 *RJ-45* 等，实际使用的设备如网卡等属于本层

### 3.1.2 TCP 4 层模型与 OSI 7 层模型的关系



1. **应用层**：由 OSI 的应用层、表示层、会话层组成，用户进程 *s,ftp,telnet, qq* 等
2. **传输层**：由 OSI 的传输层组成，协议包括 *TCP, UDP*
3. **网络层**：由 OSI 的网络层组成，协议包括 *ICMP, IP, IGMP*
4. **链路层**：由 OSI 的数据链路层、物理层组成，协议包括 *ARP, RARP*

### 3.1.3 端口

IP 可以选择中某一个主机，但是如何区分使用该主机的哪个服务就要使用端口了，因为一台机器有多个服务在使用。

端口分类如下：



- 众所周知的端口：从 0 到 1023，这些端口有 IANA 分配和控制他们紧密绑定于一些服务，通常这些端口的通讯明确的表明了某种服务的协议，例如：21 端口为 FTP 服务端口
- 注册端口：从 1024 到 49151，不受 IANA 控制，但是由 IANA 登记并提供使用情况清单。他们松散的绑定于一些服务，也就是说有许多服务绑定于这些端口，这些端口同样用于许多其他目的。如 1433 MicrosoftSQL 服务端口
- 动态或私有端口：从 49152 到 65535，IANA 不管这些端口，实际上，机器通常从 1024 起分配动态端口，但也有例外：SUN 的 RPC 端口从 32768 开始

## 3.2 数据链路层相关知识

### 3.2.1 最大传输单元 (MTU)/路径 (MTU)

以太网和 IEEE802.3 对数据帧的长度都有限制，其最大值分别是 1500 和 1492 字节，将这个限制称作最大传输单元 MTU(Maximum Transmission Unit).

如果 IP 层有一个数据报要传，而且数据的长度比链路层的 MTU 还大，那么 IP 层就要进行分片，把数据报分成若干片，这样每一片都小于 MTU.

当网络上的两台主机相互进行通信时，两台主机之间要经过多个网络，每个网络的链路层可能有不同的 MTU，其中两台通信主机路径中的最小 MTU 被称作路径 MTU.

### 3.2.2 以太网帧格式

### 3.2.3 ARP-地址解析协议：将 IP 地址解析到 MAC 地址

### 3.2.4 RARP-反地址解析协议：将 MAC 地址解析到 IP 地址

## 3.3 网络 (IP) 层相关知识

### 3.3.1 ICMP

ICMP 协议用于传递差错信息、时间、回显、网络信息等控制数据，以广播方式-如 ping。

### 3.3.2 IP 数据报格式

### 3.3.3 网际校验和

### 3.3.4 路由

## 3.4 传输层相关知识：TCP/UDP

### 3.4.1 TCP

#### TCP 特点

- 基于字节流: 段、无格式、粘包问题
- 面向连接:
- 可靠传输:
- 缓冲传输:
- 全双工:
- 流量控制:

#### TCP 报文格式

1. 确定哪个服务的端口号，包括来源端口与目的端口号
2. 保证可靠传输的 TCP 报文序号与 TCP 确认序号
3. 区别报文类型的类型字段，包括 URG、ACK、PSH、RST、SYN、FIN
4. 保证传输流畅性的滑动窗口大小
5. 保证数据正确性的数据校验和
6. 如果存在紧急指针选项，需要紧急指针
7. 其他附加选项

## TCP 如何保证可靠性

- 应用数据被分割成 TCP 认为最合适发送的数据块。这和 UDP 完全不同，应用程序产生的数据长度不变。由 TCP 传递给 IP 的信息单位称为报文段 (segment)。
- **超时重传**：当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。
- 当 TCP 收到发自 TCP 连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒。
- TCP 将保持它首部和数据的校验和，目的是检验数据在传输过程中的任何变化。如果收到段的校验和有差错，TCP 将丢弃这个报文段并且不确认（导致对方超时重传）
- TCP 承载于 IP 数据报来传输，而 IP 数据报的到达可能会失序，因此 TCP 报文段的到达也可能会失序，TCP 将对收到的数据进行重新排序。
- IP 数据报会发生重复，TCP 的接收端必须丢弃重复的数据。
- TCP 还能提供流量控制。TCP 连接的每一方都有一定大小的缓冲空间。

## 3.4.2 UDP

### 滑动窗口协议

### UDP 特点

- 无连接
- 不可靠
- 一般情况下 UDP 更高效

### UDP 报文格式

- 确认服务的端口号
- 正确传输的数据校验和



## 第四章 socket 编程篇

### 4.1 基本概念

#### 4.1.1 socket

可以看成是用户进程与内核网络协议栈的编程接口。

socket 不仅可以用于本机进程间通信，还可以用于网络上不同主机的进程间通信。

所以每个套接口都必须有地址标记，而地址则包含如下特性：

- 地址家族 (sin\_family): 对于 IPv4 协议来说必须是 AF\_INET.
- IPv4 地址 (sin\_addr): IP 地址
- 端口 (sin\_port): IP 主机上服务程序占有的端口号

通用地址结构：

```
// 通用地址结构
struct sockaddr{
    uint8_t sin_len; //整个sockaddr结构体的长度
    sa_family_t sin_family; //指定改地址家族
    char sa_data[14]; //由sin_family 决定它的形式,共占14个字节
}
```

IPv4 地址结构：

```
// IPv4 地址结构
struct sockaddr_in{
    uint8_t sin_len;
    sa_family_t sin_family;
    in_port_t sin_port; //端口号, 占2个字节
```

```

    struct in_addr sin_addr; //IP地址, 占4个字节
    char sin_zero[8]; //暂不使用, 一般将其设置为0, 占8个字节
}

```

## 4.1.2 网络字节序

### 字节序

**大端字节序** 最高有效位 (MSB:most Significant Bit) 存储于最低内存地址处, 最低有效位 (LSB:Lowest Significant Bit) 存储于最高内存地址处。

**小端字节序** 最高有效位 (MSB:most Significant Bit) 存储于最高内存地址处, 最低有效位 (LSB:Lowest Significant Bit) 存储于最低内存地址处。

### 主机字节序

不同的主机有不同的字节序, 如 x86 为小端字节序, Motorola 6800 为大端字节序, ARM 字节序是可配置的。

可以通过以下程序进行测试本机的主机字节序

```

unsigned int x = 0x12345678;
unsigned char *p = (unsigned char*) &x;
printf("%0x_ %0x_ %0x_ %0x\n", p[0], p[1], p[2], p[3]);

```

### 网络字节序

网络字节序规定为 **大端字节序**

### 字节序转换函数

- *host to network long* 长整数-主机字节序转网络字节序: `uint32_t htonl(uint32_t hostlong);`
- *host to network short*: `uint16_t htons(uint16_t hostshort);`
- *network to host long* 长整数-网络字节序转主机字节序: `uint32_t ntohl(uint32_t netlong);`
- *network to host short*: `uint16_t ntohs(uint16_t netshort);`

## 地址转换函数

在机器中地址 (00110001010101) 一般用 32 位整数表示, 而我们在操作中使用的都为点分式 IP(192.168.1.32) 地址。所以就涉及一个转换。

- 整数 转点分 IP: `char * inet_ntoa(struct in_addr in);`
- 点分 IP 转整数: `in_addr_t inet_addr(const char* cp);` // `in_addr_t` 结构体其实就是一个无符号 32 位整数.

### 4.1.3 套接字类型

- 流式套接字 (SOCK\_STREAM)[TCP]: 提供面向连接的、可靠的数据传输服务, 数据无差错、无重复的发送、且按发送顺序接受、消息无边界。
- 数据报式套接字 (SOCK\_DGRAM)[UDP]: 提供无连接服务、不提供无错保证、数据可能丢失和重复、并且接受顺序混乱、消息有边界。

### 4.1.4 流式套接字的粘包问题

因为是流式传输, 所以消息的格式随意且长度随意即无边界, 导致接受方读取接受信息时可能多读或少读的问题称为粘包问题。

#### 产生原因

处理方案 本质上是要在应用层维护消息与消息的边界:

- 定长包: `write_N -> read_N`
- 包围加结束标记 `\r \n` 等
- 报文头部包括信息长度.

### 4.1.5 获取本机网络地址

- `getsockname(int 套接字描述符, struct sockaddr*(Client 的)/赋值/ 通用地址-即将取出请求的 socket 的地址赋值给该参数通过指针返回, 地址长度): 获取本地地址-IP、端口: 127.0.0.1-51323;`

- `gethostname(char *host[赋值],size_t lenOfHostName)`: 获取主机名
- `struct hostent* gethostbyname(char *host)`: 通过主机名返回该主机的网络地址 `list,ip = hostent->h_addr_list[i]` 或 `host->h_addr`; 192.168.23.12、127.0.0.1、127.0.1.1;

## 4.2 基本的 C/S 通信模型

### 4.2.1 通信模型

### 4.2.2 客户端使用的函数及作用

- `int socket(int 协议族,int TCPorUDPorRaw,int 协议类型)`: 类似于电话机, 失败返回-1.
- `connect(int 套接字描述符,struct sockaddr*-(Server 的)[传值] 使用通用地址格式, 地址长度)`: 连接请求
- `write()`:
- `read()`: 接受并从缓存区移除消息
- `recv()`: 接受但不移除缓冲区消息

```
int main()
{
    /*
    申请套接字: 申请使用电话机
    */
    int sockfd;
    if((sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        exit(1);

    /*
    明确服务器地址(IP、协议族、端口)
    */
    struct sockaddr_in serverAddr;
    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(5188);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    /*
    连接服务器
    */
}
```



```

    */
    if((connect(sockfd, (struct sockaddr*)&serverAddr, sizeof(serverAddr)))<0)
        exit(1);

    /*
    通信
    */
    char sendbuf[1024] = {0};
    char recvbuf[1024] = {0};
    while(fgets(sendbuf,sizeof(sendbuf),stdin) != NULL)
    {
        write(sockfd, sendbuf, strlen(sendbuf));
        read(sockfd, recvbuf, sizeof(recvbuf));
        fputs(recvbuf, stdout);

        memset(recvbuf, 0, sizeof(recvbuf));
        memset(sendbuf, 0, sizeof(sendbuf));
    }
    return 0;
}

```

### 4.2.3 服务器使用的函数及作用

- `socket()`:
- `int bind(int 套接字描述符, struct sockaddr*-使用通用地址格式, 地址长度)`: 绑定本地地址到套接字-使用通用类型. 包括 IP、端口号、协议族。
- `int listen(int 套接字描述符, int 此套接字排队的最大连接个数)`: 确定最大的并发数-等价于确认未完成连接 + 已完成连接是否小于最大并发连接数。
  - 未完成连接: 未完成 3 次握手
  - 已完成连接: 完成 3 次握手
- `int accept(int 套接字描述符, struct sockaddr*(Client 的)[赋值] 通用地址-即将取出请求的 socket 的地址赋值给该参数通过指针返回, 地址长度)`: 从已完成队列返回第一个连接, 如果已连接队列为空, 则阻塞
- `read()`:
- `write()`:

```

int main()
{
    /*
    申请套接字：申请使用电话机
    */
    int listenfd;
    if((listenfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        exit(1);

    /*
    明确服务器地址(IP、协议族、端口)
    */
    struct sockaddr_in serverAddr;
    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(5188);
    // 确定主机 IP 的3种方式
    /*serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);*/
    /*serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");*/
    /*inet_aton("127.0.0.1",&serverAddr.sin_addr);*/
    /*
    解决多次调用bind() 产生的TIME_WAIT问题：利用ReuseAddr
    */
    int on = 1;
    if(setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0)
        exit(1);

    /*
    绑定本地地址即服务器地址(IP、协议族、端口)到 套接字
    */
    if(bind(listenfd, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) < 0)
        exit(1);

    /*
    确定连接的最大数-即最大并发数
    */
    if(listen(listenfd, SOMAXCONN) < 0)
        exit(1); // 等待过多

    /*
    从已完成队列-取出第一个连接,该套接字将不再处于监听状态
    */
    struct sockaddr_in peerAddr;

```

```

socklen_t peer_len = sizeof(peerAddr);
int conn;
if((conn = accept(listenfd, (struct sockaddr*)&peerAddr, &peer_len)) < 0)
    exit(1);

/*
消息通信
*/
// 根据具体应用书写如:
char recvbuf[1024];
while(1)
{
    memset(recvbuf, 0, sizeof(buf));
    read(conn, recvbuf, sizeof(recvbuf));
    fputs(recvbuf, stdout);
    write(conn, recvbuf, strlen(recvbuf));
}

return 0;
}

```

## 4.2.4 REUSEADDR

当本机服务器关闭后立即重启时会存在 Bind 错误: Address already use. 因为服务器重启又要绑定地址, 这时已经将该端口号绑定给该程序, 当改程序关闭时, 端口会处在 TIME\_WAIT 状态- 等待一定时间后才会释放, 所以在这个状态下再绑定会出现 bind 错误。

**解决方案** 实现参考上述代码

1. 服务器尽可能使用 ReuseAddr
2. 在绑定 (bind()) 前尽可能调用 setsockopt 来设置 ReuseAddr 套接字选项。
3. 使用 ReuseAddr 选项可以使得不必等待TIME\_WAIT 状态消失就可以重启服务器。

## 4.2.5 close 与 shutdown 的区别

[http://blog.csdn.net/jnu\\_simba/article/details/9068059](http://blog.csdn.net/jnu_simba/article/details/9068059)

发送方 `send() + shutdown(WR)FIN + read()->0(等待服务器) + close()`

接收方 `read()->0 + if nothing more to send + close()`

## shutdown 与 close

- `shutdown(SD_SEND)` 先把socket 缓冲区中的数据发送出去, 然后发送FIN数据包, 所谓的“从容关闭”, 仅仅做这些事, 并且这个时候, 并没有释放本地资源, 也就是, 一切皆文件, socket 在内核分配的资源没有释放, 像File 对象、Inode 节点, dentry, file\_struct 等等一个文件所对应的资源, 都没有被释放。

1. SHUT\_RD: 关闭连接的读端。也就是该套接字不再接受数据, 任何当前在套接字接受缓冲区的数据将被丢弃。进程将不能对该套接字发出任何读操作。对TCP 套接字该调用之后接受到的任何数据将被确认然后无声的丢弃掉。

2. SHUT\_WR: 关闭连接的写端, 进程不能在此套接字发出写操作

3. SHUT\_RDWR: 相当于调用shutdown 两次: 首先是以SHUT\_RD, 然后以SHUT\_WR

- `close` 会把socket 缓冲区中的数据先马上丢弃, 然后发送FIN数据包, 所谓的“暴力关闭”并且, 关闭本地文件对象, 释放资源。

1. `close` 把描述符的引用计数减 1, 仅在该计数变为 0 时才关闭套接字。

2. `close` 终止读和写两个方向的数据传送。

## 4.2.6 FIN 与 RST

### FIN

Socket 正常情况下发送的最后一个报文

### RST

RST 表示复位, 用来异常的关闭连接。发送RST 包关闭连接时, 不必等缓冲区的包都发出去, 直接就丢弃缓存区的包发送RST 包。而接收端收到RST 包后, 也不必发送ACK 包来确认。

下面列出几种会出现RST 的情况:<https://my.oschina.net/costaxu/blog/127394>

1. 服务器程序端口未打开而客户端来连接。
2. 提前关闭

### 4.2.7 SIGPIPE

在接受到对方关闭消息FIN 时，如果继续向对方发消息，那么对方会返回RST 段，如果还继续写，那么将产生SIGPIPE 信号，终止该进程。

处理可以是选择性的:

1- 常规捕捉信号 `signal(SIGPIPE,void handle(int sig))`

2- 忽略该信号 `signal(SIGPIPE, SIG_IGN)`, 一般情况下程序启动时忽略SIGPIPE 信号，防止服务器意外退出。

### 4.2.8 TCP No Delay

nagle's 算法等待每一次的确认，造成的延迟问题

解决: buffering , 或 disable nagle's 算法 (TCP\_NODELAY)

### 4.2.9 多进程服务器-允许多 client 访问

上述的服务器代码是无法完成多 client 的访问的，因为 accept 只取了一次已完成的套接字，所以改进就在此处，而问题并不是 listen 应该创建个线程或进程。

fork 函数要点: fork 后，子进程中的文件会从父进程拷贝一份，然而每个文件都有一个引用计数，此时对应的文件就会引用计数加一

close 函数要点: 只有 socket 文件引用计数为 0 时才会真正的关闭并发送 FIN

```
void do_service(int conn)
{
    char recvbuf[1024];
    while(1)
    {
        memset(recvbuf, 0, sizeof(buf));
        int ret = read(conn,recvbuf,sizeof(recvbuf));
        if(ret == 0)
        {
            printf("Client_Closed\n");
        }
    }
}
```

```

        break;
    }
    fputs(recvbuf, stdout);
    write(conn, recvbuf, strlen(recvbuf));
}
}

int main()
{
    ...
    // 取套接字
    struct sockaddr_in peerAddr;
    socklen_t peer_len = sizeof(peerAddr);
    int conn;

    pid_t pid;
    while(1)
    {
        // 从完成队列中取出第一个套接字
        if((conn = accept(listenfd, (struct sockaddr*)&peerAddr, &peer_len)) < 0)
            exit(1);
        // 打印客户端 IP、 端口号
        printf("Client Ip=%s Port=%d\n", inet_ntoa(peerAddr.sin_addr), ntohs(peerAddr.
            sin_port));

        // 为每个新的 Client 分出新的进程处理请求
        pid = fork();
        if(pid == -1)
            exit(1);

        // 如果是子进程
        if(pid == 0)
        {
            do_service(conn);
            close(conn);
            exit(EXIT_SUCCESS);
        }
        // 父进程收尾
        else
            ..
    }

    close(listenfd);
    return 0;
}

```

```
}
```

## 4.3 Peer To Peer 点对点通信：P2P

点对点的意思就是：

- 服务器与客户端都-既可以发消息也可以收消息。
- 仍然存在客户与服务概念

### 4.3.1 信号机制

发信号与收信号. `kill(pid, SIGUSR1) -> signal(SIGUSR1, handle_func);`

**应用**：可以通过发信号的方式通知某个不知道是否该结束的等待进程退出。如下，在 P2P 的情况下，当客户端退出时服务器的收消息进程知道客户端退出了，但是服务器的发消息进程并不知道客户端退出了，这是需要使用信号通知发消息进程退出。服务器退出同理。

### 4.3.2 服务器端

接受请求并开始点对点通信.

```
/*
信号处理函数
*/
void sig_handle_func(int sig)
{
    printf("Got sig=%d, and Close\n", sig);
    exit(EXIT_SUCCESS);
}

int main()
{
    ...

    if((conn = accept(listenfd, (struct sockaddr*)&peerAddr, &peer_len)) < 0)
        exit(1);
    // 打印客户端 IP、 端口号
```

```

printf("Client Ip=%s Port=%d\n",inet_ntoa(peerAddr.sin_addr),ntohs(peerAddr.sin_port
));

// 为每个新的 Client 分出新的进程处理请求
pid = fork();
if(pid == -1)
    exit(1);

// 如果是子进程:负责发消息
if(pid == 0)
{
    /*
    接收消息
    */
    signal(SIGUSR1,sig_handle_func);

    char sendbuf[1024] = {0};
    while(fgets(sendbuf, sizeof(sendbuf), stdin) != NULL)
    {
        write(conn, sendbuf, strlen(sendbuf));
        memset(sendbuf, 0, sizeof(sendbuf));
    }
    exit(EXIT_SUCCESS);
}
// 父进程: 负责读消息
else
{
    char recvbuf[1024];
    while(1)
    {
        memset(recvbuf, 0 sizeof(buf));
        int ret = read(conn, recvbuf, sizeof(recvbuf));
        if(ret == 0)
        {
            printf("Client_Closed\n");
            break;
        }
        fputs(recvbuf,stdout);
    }
    /*
    发消息
    */
    kill(pid, SIGUSR1);
    exit(EXIT_SUCCESS);
}

```



```
}  
    return 0;  
}
```

### 4.3.3 客户端

请求连接，并要求服务器完成点对点服务。

关闭发送消息进程也是类似于服务器的消息机制。

## 4.4 fork 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;  
}
```

**fork** 调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：

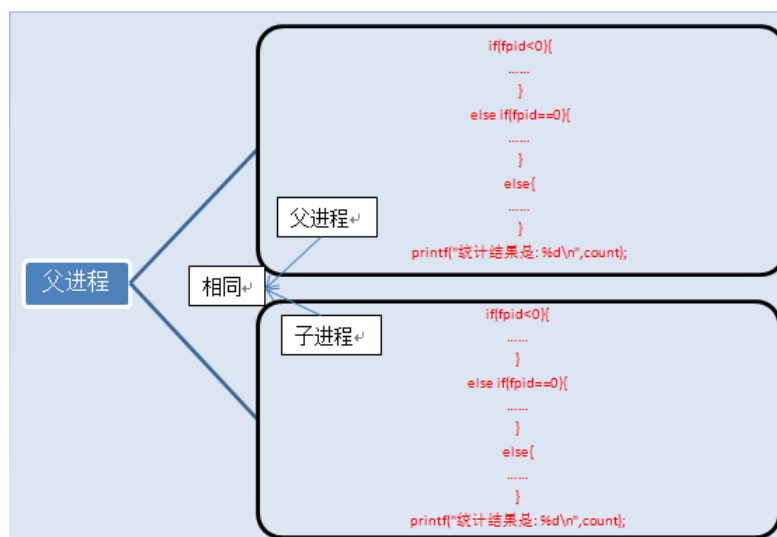
1. 在父进程中，fork 返回新创建子进程的进程 ID；
2. 在子进程中，fork 返回 0；

3. 如果出现错误，fork 返回一个负值；

fork 出错可能有两种原因：

1. 当前的进程数已经达到了系统规定的上限，这时 errno 的值被设置为 EAGAIN。
2. 系统内存不足，这时 errno 的值被设置为 ENOMEM。

fork 执行完毕后，出现两个进程，作用域代码共享，如下图所示：



## 4.5 僵尸进程

如果子进程先于父进程退出，同时父进程又没有调用 `wait/waitpid`，则该子进程将成为僵尸进程。

### 4.5.1 参考

介绍: <http://www.cnblogs.com/yuxingfirst/p/3165407.html>

避免: <http://blog.chinaunix.net/xmlrpc.php?r=blog/article&uid=26872853&id=4574069>

### 4.5.2 wait 与 waitpid

<http://blog.csdn.net/kevinhg/article/details/7001719>

`waitpid`: 父进程利用工作的简短间歇察看子进程的是否退出，如退出就收集它。(相当于管理一个队列)

wait: 父进程将自己阻塞，直到有子进程退出为止 (相当于等待系统资源)

## 4.6 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。

## 4.7 守护进程

守护进程是在后台运行且不与任何控制终端相关联的进程。通常由系统初始化脚本启动，当然也可以在 shell 提示符下用命令行启动，不过这种守护进程必须亲自脱离于控制终端的关联。

- 创建子进程，终止父进程-变成孤儿进程
- 在子进程中创建新会话，调用setsid 的三个作用：让进程摆脱原会话的控制、让进程摆脱原进程组的控制和让进程摆脱原控制终端的控制。
- 改变工作目录
- 重设文件创建掩码

### 4.7.1 守护进程的输出

<http://blog.csdn.net/smsstong/article/details/8919803>

由于守护进程没有终端,所以它的消息用fprintf到stderr 上,从守护进程登记消息使用syslog函数, 作为替换, 也可以使用openlog closelog。

syslog 是一种标准的协议，分为客户端和服务端，客户端是产生日志消息的一方，而服务器端负责接收客户端发送来的日志消息，并做出保存到特定的日志文件中或者其他方式的处理。

在 Linux 中，常见的 syslog 的服务器端程序是 syslogd 守护程序。

```
void syslog(int priority,const char *message,...);
```

- priority: 由 level 与 facility 共同决定

level ->

— LOG\_ERR

- LOG\_WARNING
- LOG\_NOTICE
- LOG\_INFO

facility ->

- LOG\_LOCAL1
- LOG\_KERN
- LOG\_DAEMON

- message: 要传递的消息

example-> syslog(LOG\_INFO|LOG\_LOCAL2, "read name(%s,%s)", str, str2);

## 4.7.2 守护进程的启动方法有

1. 系统初始化阶段, 由系统初始化脚本启动。这些脚本通常位于/etc、/etc/rc 开头的某个目录中。由这些脚本启动的守护进程从一开始就有root 特权。例如: inetd 超级服务器、Web 服务器、邮件服务器、syslogd 守护进程等都用这种方式启动。
2. 靠inetd 超级服务器启动。inetd 超级服务器监听网络请求, 每当有一个请求到达时, 启动相应的实际服务器。
3. 从用户终端或者后台启动。这么做往往是为了测试守护程序或重启因某种原因而终止的守护进程。

## 4.7.3 分类

- Netd 就是 Network Daemon 的缩写, 表示 **Network 守护进程**. Netd 负责跟一些涉及网络的配置, 操作, 管理, 查询等相关的功能实现, 比如, 例如带宽控制 (Bandwidth), 流量统计, 带宽控制, 网络地址转换 (NAT), 个人局域网 (pan), PPP 链接, soft-ap, 共享上网 (Tether), 配置路由表, interface 配置管理, 等等. 好像 Andorid 用的
- inetd 是**监视一些网络请求**的守护进程, 其根据网络请求来调用相应的服务进程来处理连接请求。它可以为多种服务管理连接, 当 inetd 接到连接时, 它能够确定连接所需的程序, 启动相应的进程, 并把 socket 交给它。

<http://blog.csdn.net/u012103747/article/details/45338571>

#### 4.7.4 UID,EUID

实际 ID, uid: 标示你是谁的一个 ID;

有效 ID, euid: 标示你的有效身份是谁, 即你有没有权限访问某个东西。

## 4.8 TCP

### 4.8.1 11 种状态

TCP 从开始建立到最后关闭的所有 11 种状态如下图4.1 所示

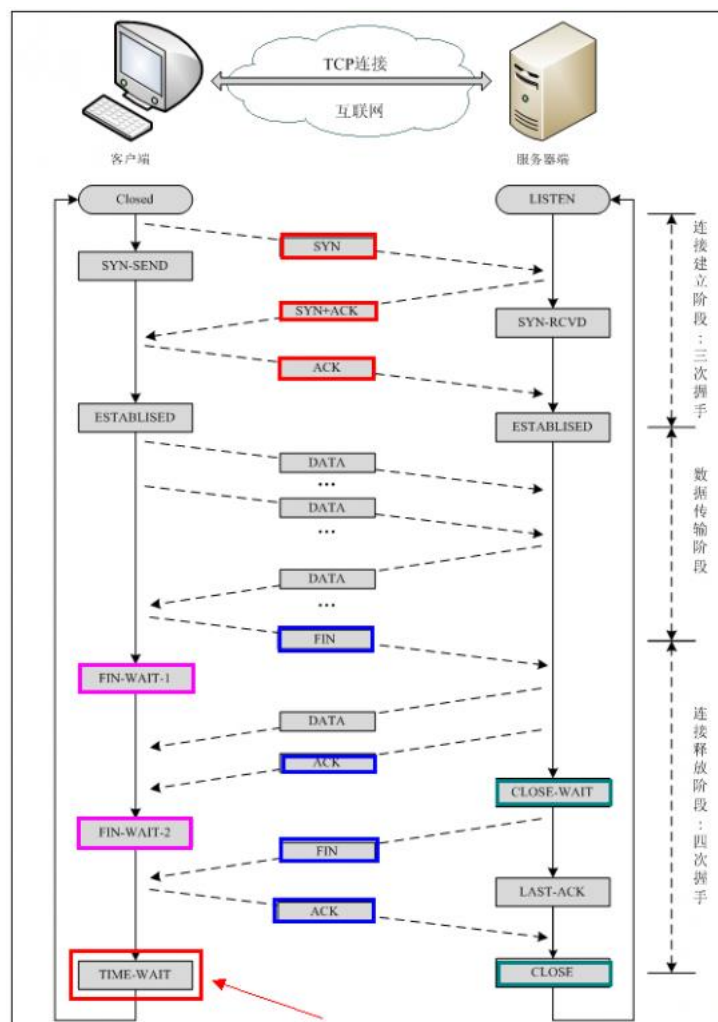


图 4.1: 整个过程

- LISTEN
- SYN\_SENT
- SYN\_RCVD
- ESTABLISHED
- FIN\_WAIT\_1: 往一个已经接受 FIN 的套接字中写是允许的，接受到 FIN 仅代表对方不再发送数据了。
- CLOSE\_WAIT
- FIN\_WAIT\_2
- LAST\_ACK
- TIME\_WAIT
- CLOSED
- CLOSING: 双方同时关闭的情况

### 4.8.2 3 次握手: 建立链接

首先 Client 端发送连接请求报文, Server 段接受连接后回复 ACK 报文, 并为这次连接分配资源。Client 端接收到 ACK 报文后也向 Server 段发生 ACK 报文, 并分配资源, 这样 TCP 连接就建立了, 如图4.2.

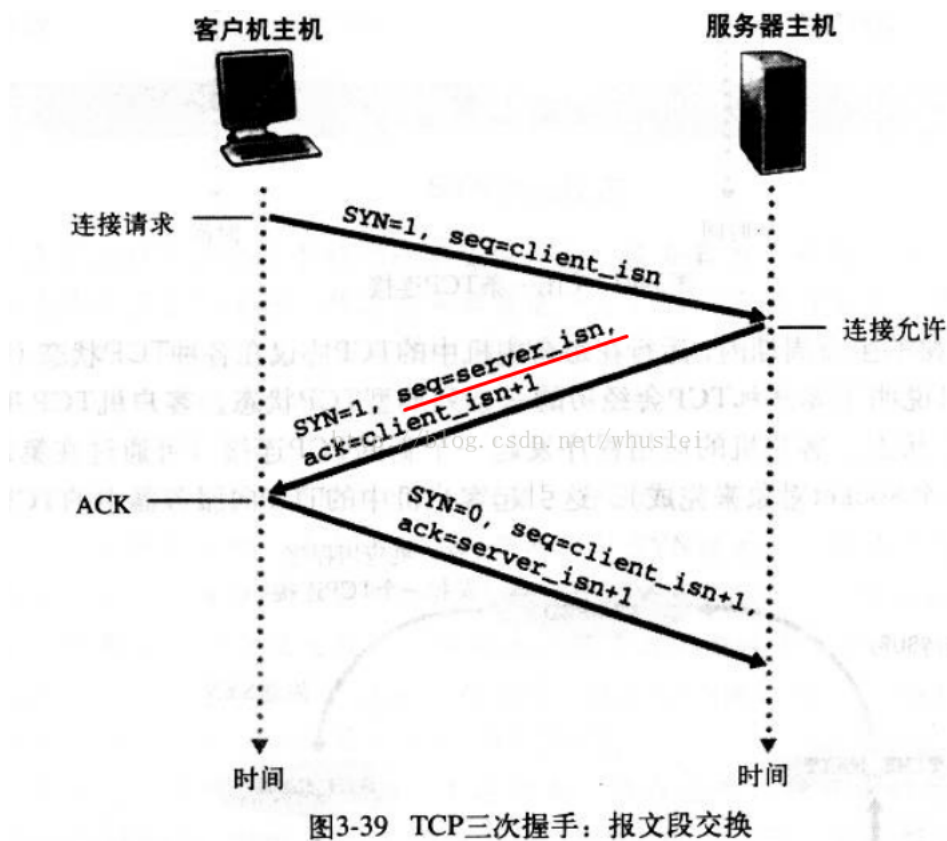


图 4.2: 3 次握手

- Client 请求连接:SYN\_SENT

客户端 TCP 首先给服务器端 TCP 发送一个特殊的 TCP 数据段。该数据段不包含应用层数据, 并将头部中的SYN 位设置为1, 所以该数据段被称为 SYN 数据段。另外, 客户选择一个初始序列号SEQ, 设SEQ=x 并将这个编号放到初始的 TCP SYN 数据段的序列号字段中。该数据段被封装到一个IP 数据报中, 并发送给服务器。

- Server 同意连接并发送确认代码:SYN\_RCVD

一旦装有 TCP SYN 数据段的 IP 数据报到达了服务器主机, 服务器将从该数据报中提取出 TCP SYN 数据段, 给该连接分配 TCP 缓冲区和变量, 并给客户 TCP 发送一个允许连接的数据段。这个允许连接的数据段也不包含任何应用层数据。但是, 它的头部中装载着 3 个重要信息。首先, SYN 被设置为 1; 其次, TCP 数据段头部的确认字段被设置为x+1; 最后, 服务器选择自己的初始顺序号, SEQ=y, 并将该值放到 TCP 数据段头部的序列号字段中。

- Client 确认连接:ESTABLISHED

在接收到允许连接数据段之后，客户也会给连接分配缓冲区和变量。客户端主机还会给服务器发送另一个数据段，对服务器的允许连接数据段给出确认。



### 4.8.3 4 次握手: 终止链接

断开连接简单的过程如下

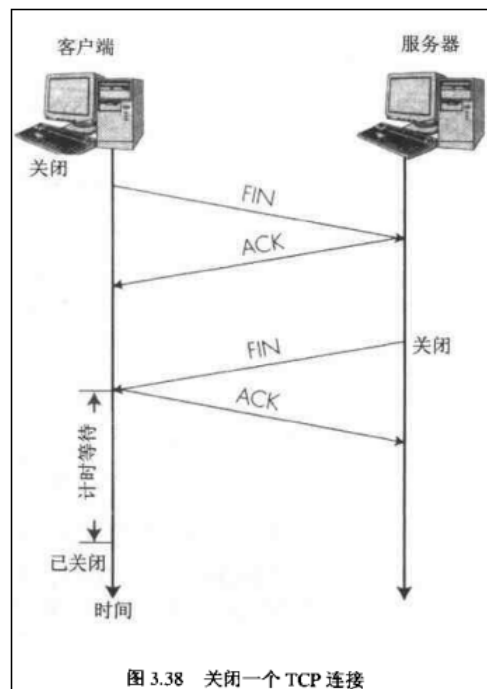


图 4.3: Tcp 终止连接 4 次握手

- 一方 A 关闭: FIN\_WAIT\_1

由进行数据通信的任意一方提出要求释放连接请求报文段

- 另一方 B 读取到FIN 消息后并发送确认代码: CLOSE\_WAIT
- 另一方 B 当所有数据也都已经发送完毕后发送最后一个确认代码FIN

接收端收到此请求后, 会发送确认报文段, 同时当接收端的所有数据也都已经发送完毕后, 接收端会向发送端发送一个带有其自己序号的报文段

- 一方 A 未确认超过TIME\_WAIT 时间自动向服务器发送关闭确认

中断连接端可以是 Client 端, 也可以是 Server 端

假设 Client 端发起中断连接请求, 也就是发送 FIN 报文。Server 端接到FIN 报文后, 意思是说”我 Client 端没有数据要发给你了”, 但是如果你还有数据没有发送完成, 则不必急着关闭 Socket, 可以继续发送数据。所以你先发送ACK, ”告诉 Client 端, 你的请求我收到了, 但是我还没准备好, 请继续你等我的消息”。这个时候 Client 端就进入FIN\_WAIT 状态, 继续等待 Server 端的FIN 报文。当 Server 端确定数据已发送完成, 则向 Client 端发送FIN 报文, ”告诉 Client 端, 好了, 我这边数据发完了, 准备好关闭连接了”。Client 端收到FIN 报文后, ”就知道可以关

闭连接了，但是他还是不相信网络，怕 Server 端不知道要关闭，所以发送ACK 后进入TIME\_WAIT 状态，如果 Server 端没有收到ACK 则可以重传。“，Server 端收到ACK 后，” 就知道可以断开连接了”。Client 端等待了2MSL 后依然没有收到回复，则证明 Server 端已正常关闭，那好，我 Client 端也可以关闭连接了。Ok，TCP 连接就这样关闭了！

## Client 经历的状态

整个过程 Client 端所经历的状态如下：

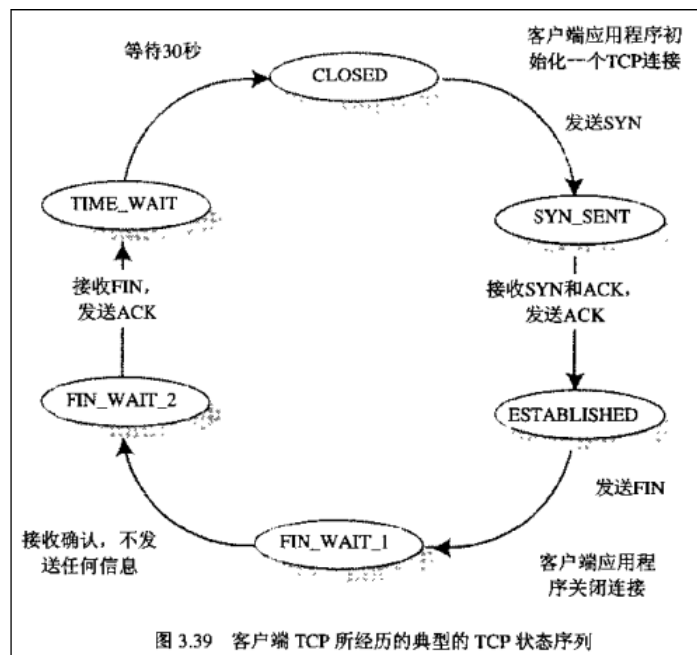


图 4.4: Client 所有状态

## Server 经历的状态

整个过程 Server 端所经历的状态如下：

### Notice

在TIME\_WAIT 状态中,如果 TCP client 端最后一次发送的ACK 丢失了,它将重新发送。TIME\_WAIT 状态中所需要的时间是依赖于实现方法的。典型的值为 30 秒、1 分钟和 2 分钟。等待之后连接正式关闭，并且所有的资源 (包括端口号) 都被释放

为什么连接的时候是三次握手,关闭的时候却是四次握手？ 答: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

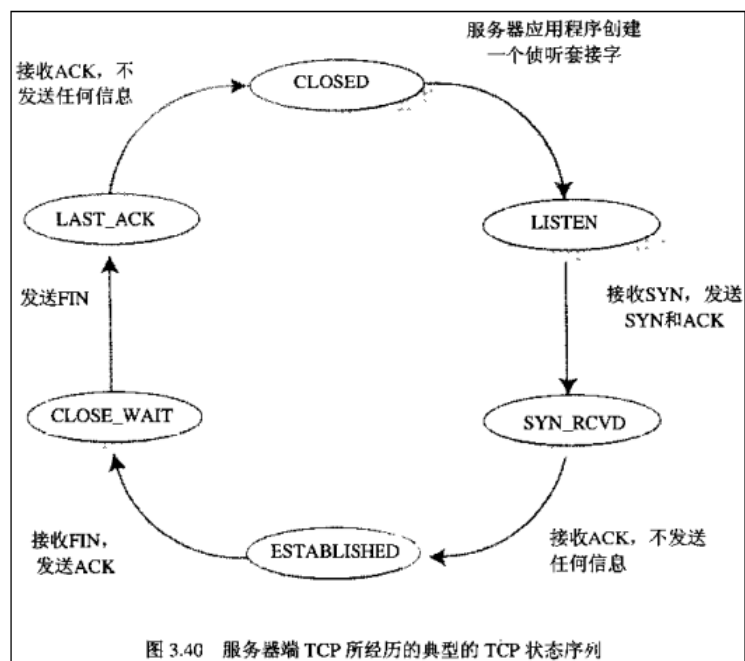


图 4.5: Server 所有状态

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。

### TIME\_WAIT 存在的理由

- 虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE 状态了，但是我们必须假象网络是不可靠的，有可以最后一个ACK 丢失。所以TIME\_WAIT 状态就是用来重发可能丢失的 ACK 报文，否则被动关闭端会出现 RST 错误。
- TIME\_WAIT 第二个原因是为防止来自某个链接的老的重复分组在该链接已经终止后再现，从而被误解成属于同一链接的某个新的化身。为做到这一点，TCP 将不给处于TIME\_WAIT 状态的连接发起新的化身。

而为什么是 2MSL，有以下两方面因素

- 足以让某个被动关闭方向上的分组最多存活 MSL 即被丢弃
- 主动关闭方向上的应答最多存活 MSL 秒也被丢弃

## 4.9 套接字选项

### 4.9.1 相关函数

- `int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen);`
- `int setsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen);`

1. `sockfd`(套接字描述符)

2. `level`(级别), 如果想要在套接字级别上设置选项, 就必须把 `level` 设置为 `SOL_SOCKET`

- `SOL_SOCKET`: 基本套接口
- `IPPROTO_IP`: IPv4 套接口
- `IPPROTO_IPV6`: IPv6 套接口
- `IPPROTO_TCP`: TCP 套接口

3. `optname`(选项名): 选项名称

4. `optval`(选项值): 是一个指向变量的指针

5. `optlen`(选项长度): `optval` 的大小

->Typical Code

```
int nZero=0;
setsockopt(socket, SOL_SOCKET, SO_SNDBUF, (char *)&nZero, sizeof(nZero));
```

- `int fcntl(int fd, int cmd, ... /* arg */);`

```
// 将描述符设置为 非阻塞
int flags;
flags = fcntl(fd, F_GETFL, 0);
flags |= O_NONBLOCK;
fcntl(fd, F_SETFL, flags);

// 关闭非阻塞
flags &= ~O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
```

- `*ioctl()`

## 4.9.2 低水位标记

分为接受低水位标记和发送低水位标记，目的在于：它允许应用进程控制在 `select` 返回可读或可写条件之前有多少数据可读或有多少空间可用于写。

-> **Example**: 如果我们知道除非至少存在 64 个字节的数据，否则我们的应用进程没有任何有效工作可做，那么可以把接受低水位标记设置为 64，以防少于 64 个字节的数据准备好读时 `select` 唤醒我们。

## 4.9.3 高水位标记

当你能够迅速从进程向进程发送消息时，你很快就会发现，**内存是宝贵的资源，并且很容易填满**。除非你理解这个问题，并采取预防措施，否则在进程中某处几秒钟的延迟，都可能变成撑爆一个服务器的积压。

**问题** 如果有一个进程A 正发送消息给进程B，而进程B 突然变得非常繁忙（执行垃圾收集，CPU 过载，等等），那么进程A 要发送的消息会发生什么情况呢？有些消息会位于B 的网络缓冲区，有些消息会位于以太网线路本身，有些消息会位于 A 的网络缓冲区，其余的会积聚在 A 的内存中。如果不采取一些预防措施，A 很容易就会耗尽内存并且崩溃。这是消息代理的一个经典问题。

**解决办法** 一个办法是将问题交给上游。A 是从别的地方得到的消息，所以可以告诉该进程停下来，排队，这就是所谓的**流量控制**。这听起来不错，但如果你发送的是 Twitter 的微博呢？你能告诉整个世界，在B 采取行动时停止发微博吗？

**流量控制**在某些情况下能够工作，而在有些情况下不能工作。传输层也不能叫应用层“停止”，好比地铁系统不可以告诉一个大型企业，“请让你的员工继续工作半小时。我太忙了。”

对于**消息传递**，解决办法是**设置缓冲区的大小限制**，然后，当我们到达这些限制时，采取一些明智的行动。在某些情况下（不过不是一个地铁系统），答案是**丢弃消息**。在其他情况下，最好的策略是**等待**。

.MQ 使用**高水位标记**（high-water mark, HWM）的概念来定义其内部管道的容量。每个从套接字出来或连入套接字的连接都有它自己用于发送和或接收的管道和 HWM，这取决于套接字类型。有些套接字（PUB、PUSH）只有发送缓冲区，有些套接字（SUB、PULL、REQ、REP）只有接收缓冲区，有些套接字（DEALER、ROUTER、PAIR）同时具有发送缓冲区和接收缓冲区。

在.MQ v2.x 版本中，HWM 默认是无限的。在.MQ v3.x 版本，它在默认情况下为 1000，这是比较明智的设置。如果你还在使用.MQ v2.x 版本，你应该总是在你的套接字上设置 HWM，无论是设置为 1000 以匹配.MQ v3.x，还是设置为另一个考虑到你的消息大小的数字。

当你的套接字达到其 HWM 时，根据不同的套接字类型，**这将阻塞或删除数据**。如果它们达到其 HWM，PUB 和 ROUTER 套接字**将丢弃数据**，而其他类型的套接字**将阻塞**。通过 inproc 传输时，发送者和接收者共享同一个缓冲区，所以真正的 HWM 等于由两侧设置的 HWM 的总和。

#### 4.9.4 SO\_KEEPALIVE

给一个 TCP 套接字设置保持存活选项后，如果 **2 小时内**在该套接字的任何一方向上都没有数据交换，TCP 就自动给对端发送一个保持存活探测分节。这是一个对端必须响应的 tcp 分节，它会导致以下三种情况之一：

- 对端**以期望的ACK 响应**。应用进程得不到通知（因为一切正常）。在又经过仍无动静的 2 小时后，TCP 将发出另一个探测分节。
- 对端**以RST 响应**，它告知本端TCP：对端已崩溃且已重新启动。该套接字的待处理错误被置为ECONNRESET，套接字本身则被关闭。
- 对端对保持存活探测分节**没有任何响应**。

如果根本没有对 TCP 的探测分节的响应，该套接字的待处理错误就被置为ETIMEOUT，套接字本身则被关闭。然而如果该套接字收到一个ICMP错误作为某个探测分节的响应，那就返回响应的错误，套接字本身也被关闭。

本选项的功能是**检测对端主机是否崩溃或变的不可达**（譬如拨号调制解调器连接掉线，电源发生故障等等）。如果对端进程崩溃，它的 TCP 将跨连接发送一个 FIN，这可以通过调用 select 很容易的检测到。

本选项一般由服务器使用，不过客户也可以使用。服务器使用本选项时因为他们花大部分时间阻塞在等待穿越 TCP 连接的输入上，也就是说在等待客户的请求。然而如果客户主机连接掉线，电源掉电或者系统崩溃，服务器进程将永远不会知道，并将继续等待永远不会到达的输入。我们称这种情况为**半开连接**。保持存活选项将检测出这些半开连接并终止他们。

可以缩短 TCP 的保持存活定时器参数改为一个小的多的值，但是这些参数通常是按照内核而不是按照每个套接字维护的。

KeepAlive, 参数修改：<http://blog.csdn.net/callinglove/article/details/38380673>

情 形	对方进程崩溃	对方主机崩溃	对方主机不可达
本地 TCP 正主动发送数据	对方 TCP 发送一个 FIN, 我们通过使用 select 判断可读条件立即能检测出来。如果本地 TCP 发送另外一个分节, 对方 TCP 就以 RST 响应。如果再发送另外一个分节, 本地 TCP 就给我们发一个 SIGPIPE 信号	本地 TCP 将超时, 且套接口的待处理错误被设置为 ETIMEDOUT	本地 TCP 将超时, 且套接口的待处理错误被设置为 EHOSTUNREACH
本地 TCP 正主动接收数据	对方 TCP 将发送一个 FIN, 我们将把它作为一个(可能是过早的)文件结束符读入	我们将停止接收数据	我们将停止接收数据
连接空闲, 保持存活选项已设	对方 TCP 发送一个 FIN, 我们通过使用 select 判断可读条件立即能检测出来	在毫无动静 2 小时后, 发送 9 个保持存活探测分节, 然后套接口的待处理错误被设置为 ETIMEDOUT	在毫无动静 2 小时后, 发送 9 个保持存活探测分节, 然后套接口的待处理错误被设置为 EHOSTUNREACH
连接空闲, 保持存活选项未设	对方 TCP 发送一个 FIN, 我们通过使用 select 判断可读条件立即能检测出来	(无)	(无)

图 4.6: keepAlive

### 4.9.5 SO\_LINGER

本选项指定 close 函数对面向连接的协议 (例如 TCP 和 SCTP, 但不是 UDP) 如何操作。默认操作是 close 立即返回, 但是如果有数据残留在套接字发送缓冲区中, 系统将试着把这些数据发送给对端。

-> 如果选择此选项, close 或 shutdown 将等到所有套接字里排队的消息成功发送或到达延迟时间后才会返回。否则, 调用将立即返回。

```
struct linger {
    int l_onoff; // 1 is on, 0 is off
    int l_linger; // wait time
};
```

如果在发送数据的过程中 (send() 没有完成, 还有数据没发送) 而调用了 closesocket(), 以前我们一般采取的措施是”从容关闭”shutdown(s, SD\_BOTH), 但是数据是肯定丢失了, 如何设置让程序满足具体应用的要求 (即让没发完的数据发送出去后在关闭 socket)?

```
linger m_sLinger;
m_sLinger.l_onoff=1; // 在 closesocket() 调用, 但是还有数据没发送完毕的时候容许延时关闭
```

```
m_sLinger.l_linger=5;//容许逗留的时间为5秒

setsockopt(s,SOL_SOCKET,SO_LINGER,(const char*)&m_sLinger,sizeof(linger));
```

### 4.9.6 SO\_RCVBUF

每个套接字都有一个发送缓冲区和一个接收缓冲区。

接收缓冲区被 TCP, UDP 和 SCTCP 用来保存接收到的数据,直到由应用进程读取。对于 TCP 来说,套接字接收缓冲区可用空间的大小限制了 TCP 通告对端的窗口大小。TCP 套接字接收缓冲区不可以溢出,因为不允许对端发出超过本端所通告窗口大小的数据。这就是 TCP 的流量控制,如果对端无视窗口大小而发出了超过窗口大小的数据,本端 TCP 将丢弃它们。然而对于 UDP 来说,当接收到的数据报装不进套接字接收缓冲区时,该数据报就被丢弃。回顾一下,UDP 是没有流量控制的:较快的发送端可以很容易的淹没较慢的接收端,导致接收端的 UDP 丢弃数据报。

这两个套接字选项允许我们改变着两个缓冲区的默认大小。对于不同的实现,默认值得大小可以有很大的差别。如果主机支持 NFS,那么 UDP 发送缓冲区的大小经常默认为 9000 字节左右的一个值,而 UDP 接收缓冲区的大小则经常默认为 40000 字节左右的一个值。

当设置 TCP 套接字接收缓冲区的大小时,函数调用的顺序很重要。这是因为 TCP 的出口规模选项是在建立连接时用 SYN 分节与对端互换得到的。

对于客户,这意味着 SO\_RCVBUF 选项必须在调用 connect 之前设置;

对于服务器,这意味着该选项必须在调用 listen 之前给监听套接字设置。给已连接套接字设置该选项对于可能存在的出口规模选项没有任何影响,因为 accept 直到 TCP 的三路握手完成才会创建并返回已连接套接字。这就是必须给监听套接字设置本选项的原因。

### 4.9.7 SO\_SNDBUF

在 send() 的时候,返回的是实际发送出去的字节 (同步) 或发送到 socket 缓冲区的字节 (异步);

系统默认的状态发送和接收一次为 8688 字节 (约为 8.5K); 在实际的过程中发送数据和接收数据量比较大,可以设置 socket 缓冲区,而避免了 send(),recv() 不断的循环收发:

```
// 接收缓冲区
int nRecvBuf=32*1024;//设置为32K
setsockopt(s,SOL_SOCKET,SO_RCVBUF,(const char*)&nRecvBuf,sizeof(int));
```



```
//发送缓冲区
int nSendBuf=32*1024;//设置为32K
setsockopt(s,SOL_SOCKET,SO_SNDBUF,(const char*)&nSendBuf,sizeof(int));
```

## 4.9.8 SO\_RCVLOWAT

## 4.9.9 SO\_SNDLOWAT

## 4.9.10 SO\_RCVTIMEO

这两个选项允许我们给套接字的接收和发送设置一个超时值。注意，访问他们的`getsockopt`和`setsockopt`函数的参数是指向`timeval`结构的指针，与`select`所用参数相同。这可让我们用秒数和微妙数来规定超时。我们通过设置其值为 0s 和 0 s 来禁止超时。默认情况下着两个超时都是禁止的。

接收超时影响 5 个输入函数：`read`,`readv`,`recv`,`recvfrom`和`recvmsg`。

发送超时影响 5 个输出函数：`write`,`writen`,`send`,`sendto`和`sendmsg`。

## 4.9.11 SO\_SNDTIMEO

在`send()`,`recv()`过程中有时由于网络状况等原因，发收不能预期进行，而设置收发时限：

```
int nNetTimeout=1000;//1秒
//发送时限
setsockopt(socket, SOL_SOCKET,SO_SNDTIMEO, (char *)&nNetTimeout,sizeof(int));
//接收时限
setsockopt(socket, SOL_SOCKET,SO_RCVTIMEO, (char *)&nNetTimeout,sizeof(int));
```

## 4.9.12 SO\_REUSEADDR

关闭套接字后，一般不会立即关闭而经历`TIME_WAIT`的过程，后想立即重用该地址：

```
BOOL bReuseaddr=TRUE;
setsockopt(s,SOL_SOCKET ,SO_REUSEADDR,(const char*)&bReuseaddr,sizeof(BOOL));
```

### 4.9.13 SO\_BROADCAST

本选项开启或禁止进程发送广播消息的能力。只有数据报套接字支持广播，并且还必须是在支持广播消息的网络上（例如以太网，令牌环网等）。我们不可能在点对点链路上进行广播，也不可能基于连接的传输协议（例如 TCP 和 SCTP）之上进行广播。

### 4.9.14 SO\_DEBUG

本选项仅由 TCP 支持。当给一个 TCP 套接字开启本选项时，内核将为 TCP 在该套接字发送和接受的所有分组保留详细跟踪信息。这些信息保存在内核的某个环形缓冲区中，并可使用 `trpt` 程序进行检查。

### 4.9.15 SO\_OOBINLINE

默认是禁止的，这样对于接受套接字来说，该数据字节不放入套接字接收缓冲区，而是被放入该链接的一个独立的单字节带外缓冲区。接受进程从单字节缓冲区读入数据的唯一方法是使用 `MSG_OOB` 标志调用 `recv`、`recvfrom`、`recvmsg`。

当本选项开启时，带外数据将留在正常的输入队列中。这种情况，接受进程不能使用 `MSG_OOB` 读入数据字节。

### 4.9.16 TCP\_MAXSEG

TCP 最大分节大小

### 4.9.17 TCP\_NODELAY

禁止 Nagle 算法。

参考 [http://blog.sina.com.cn/s/blog\\_b4ef897e0102vrtt.html](http://blog.sina.com.cn/s/blog_b4ef897e0102vrtt.html)

## 4.10 带外数据-Urgent 标志

即使数据的流动会因为 TCP 的流量控制而停止，紧急通知却总是无障碍的发送到对端 TCP。

<http://blog.csdn.net/ordeder/article/details/43243425>

## 4.11 心搏函数-弥补 keepAlive2 小时等待时间

## 4.12 高级 IO 函数

### 4.12.1 创建文件描述符系列

#### pipe

pipe 函数可用于创建一个单向管道，以实现 进程间通信。

- 头文件 `#include <unistd.h>`
- 函数原型: `int pipe(int filedes[2]);` pipe() 会建立管道,并将文件描述符由参数filedes 数组返回
  1. `filedes[0]` 为管道里的读取端
  2. `filedes[1]` 则为管道的写入端

```
#include <unistd.h>
#include <stdio.h>

int main( void )
{
    int filedes[2];
    char buf[80];
    pid_t pid;

    pipe( filedes );
    pid=fork();
    if (pid > 0)
    {
        printf( "This is in the father process, here write a string to the pipe.\n" );
        char s[] = "Hello world, this is write by pipe.\n";
        write( filedes[1], s, sizeof(s) );
        close( filedes[0] );
        close( filedes[1] );
    }
    else if(pid == 0)
    {
        printf( "This is in the child process, here read a string from the pipe.\n" );
        read( filedes[0], buf, sizeof(buf) );
    }
}
```

```

        printf( "%s\n", buf );
        close( filedес[0] );
        close( filedес[1] );
    }

    waitpid( pid, NULL, 0 );

    return 0;
}

```

## socketpair

### 创建双向管道

- 函数原型: `int socketpair(int d, int type, int protocol, int fd[2]);`
  1. 第 1 个参数 `d`, 表示协议族, 只能为 `AF_LOCAL` 或者 `AF_UNIX`;
  2. 第 2 个参数 `type`, 表示类型, 只能为 0。
  3. 第 3 个参数 `protocol`, 表示协议, 可以是 `SOCK_STREAM` 或者 `SOCK_DGRAM`。用 `SOCK_STREAM` 建立的套接字对是管道流, 与一般的管道相区别的是, 套接字对建立的通道是双向的, 即每一端都可以进行读写。
  4. 第 4 个参数 `fd`, 用于保存建立的套接字对。

Demo: <http://blog.csdn.net/u012719256/article/details/52945310>

## dup.. 复制

`dup` 函数创建一个新的文件描述符, 该新文件描述符和原有文件描述符 `file_descriptor` 指向相同的文件、管道或者网络连接。并且 `dup` 返回的文件描述符总是取系统当前可用的最小整数值。

```
int dup( int oldfd );
```

### 执行 dup 前

执行 `dup` 后 `Dup` 函数总是从数组第一个元素开始扫描, 获取第一个可用的文件描述符 (也就是没有关联实际文件的 `fd`), 这就是所谓: `dup` 总是使用最小的文件描述符。理解了原理就简

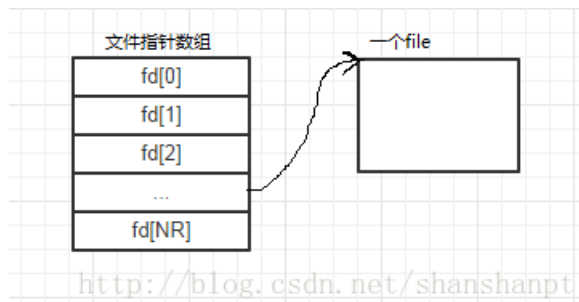


图 4.7: 执行 `dup` 前

单了。

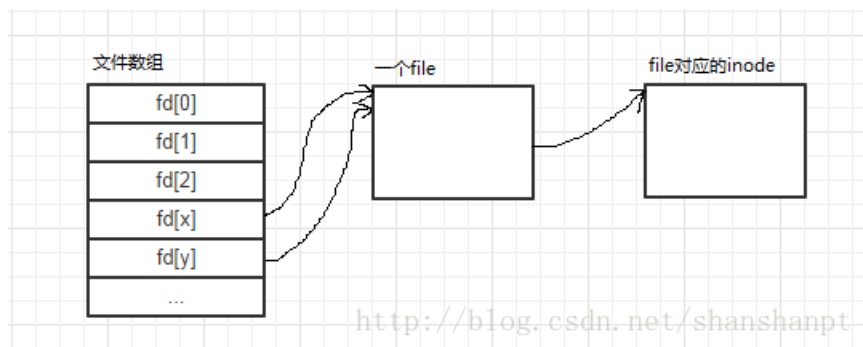


图 4.8: 执行 `dup` 后

**例子** 一般来说，初始化的时候，进程都拥有默认的三个文件描述符默认代表，标准输入，标准输出，标准错误。但是这并非硬性规定，你可以自己改呀！例如下面的代码：

```
close(0);
dup(fd[x]); /* 这是一个普通文件的文件描述符 */
```

这之后，你会发现，0 号文件描述符关联上了这个文件 (0 是最小的文件描述符，所以肯定会被 `dup` 选中！

理解：<http://blog.csdn.net/shanshanpt/article/details/39049579>

## dup2.. 重定向

类似于 linux 下的**重定向命令**，将某个输出重定向到某个文件，对于 `dup2`，相当于将 `targetfd` 重定向到 `oldfd`。`dup2` 函数成功返回时，目标描述符（`dup2` 函数的第二个参数）将变成源描述符（`dup2` 函数的第一个参数）的复制品，换句话说，两个文件描述符现在都指向同一个文件，并且是函数第一个参数指向的文件。

```
int dup2( int oldfd, int targetfd );
```

\* 例子 用 dup 复制保存原文件描述符指向，用 dup2 重定向其到其他文件，最后再还原。

```
int main()
{
    int sfd = dup(STDOUT_FILENO), testfd;

    printf("sfd=%d\n", sfd);

    testfd = open("./temp", O_CREAT | O_RDWR | O_APPEND);
    if (-1 == testfd)
    {
        printf("open_file_error.\n");
        exit(1);
    }

    /* 重定向 */
    if (-1 == dup2(testfd, STDOUT_FILENO) ) {
        printf("can't redirect_fd_error\n");
        exit(1);
    }

    /* 此时向stdout写入应该输出到文件 */
    write(STDOUT_FILENO, "file\n", 5);

    /* 恢复stdout */
    if (-1 != dup2(sfd, STDOUT_FILENO) ) {
        printf("recover_fd_ok\n");

        /* 恢复后，写入stdout应该向屏幕输出 */
        write(STDOUT_FILENO, "stdout\n", 7);
    }

    printf("gogogogogogo!\n");
    close(testfd);
}
```

## 4.12.2 读写数据系列

### readv/writev

readv 函数将数据从文件描述符读到分散的内存块中，即分散读；

writev 函数则将多块分散的内存数据一并写入文件描述符中，即集中写。

## sendfile

`sendfile` 函数在两个文件描述符之间直接传递数据（完全在内核中操作），从而避免了内核缓冲区和用户缓冲区之间的数据拷贝，效率很高，这被称为零拷贝。

## mmap/munmap

内存映射见 IPC

## splice

`splice` 函数用于在两个文件描述符之间移动数据，也是零拷贝操作。

## tee

`tee` 函数在两个管道文件描述符之间复制数据，也是零拷贝操作。它不消耗数据，因此源文件描述符上的数据仍然可以用于后续的读操作。

## 4.12.3 控制 IO 行为和属性系列

### fcntl

## 4.13 IPC

### 4.13.1 IPC 简介

### 4.13.2 信号

<http://blog.csdn.net/ljianhui/article/details/10128731>

### 4.13.3 消息传递

### 4.13.4 信号量

#### 信号量

<http://blog.csdn.net/ljianhui/article/details/10243617>

**简介** 为了防止出现因多个程序同时访问一个共享资源而引发的一系列问题，我们需要一种方法，它可以通过生成并使用令牌来授权，在任一时刻只能有一个执行线程访问代码的临界区域。

**临界区域**是指执行数据更新的代码需要独占式地执行。

而信号量就可以提供这样的一种访问机制，让一个临界区同一时间只有一个线程在访问它，也就是说信号量是用来调协进程对共享资源的访问的。

信号量是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行等待（即 P(信号变量)）和发送（即 V(信号变量)）信息操作。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**二进制信号量**。而可以取多个正整数的信号量被称为**通用信号量**。

**工作原理** 信号量只能进行两种操作-P 与 V 操作

- P(sem): 如果sem 的值大于零，就给它减 1；如果它的值为零，就挂起该进程的执行
- V(sem): 如果有其他进程因等待sem 而被挂起，就让它恢复运行，如果没有进程因等待sem 而挂起，就给它加 1

#### 使用

- `int semget(key_t key, int num_sems, int sem_flags);` // 创建一个新信号量或取得一个已有信号量
  1. `key` 是整数值（唯一非零），不相关的进程可以通过它访问一个信号量，它代表程序可能要使用的某个资源
  2. `num_sems` 指定需要的信号量数目，它的值几乎总是 1
  3. `sem_flags` 是一组标志，当想要当信号量不存在时创建一个新的信号量，可以和值 `IPC_CREAT` 做按位或操作。设置了 `IPC_CREAT` 标志后，即使给出的键是一个已有信号量的键，也



不会产生错误。而IPC\_CREAT|IPC\_EXCL 则可以创建一个新的，唯一的信号量，如果信号量已存在，返回一个错误

- `int semop(int sem_id, struct sembuf *sem_opa, size_t num_sem_ops);` // 改变信号量的值

1. `sem_id` 是由`semget` 返回的信号量标识符

2. `sembuf` 结构代表了某种操作

```
struct sembuf{
    /*
        除非使用一组信号量，否则它为0
    */
    short sem_num;

    /*
        信号量在一次操作中需要改变的数据，通常是两个数
        一个是-1，即P（等待）操作
        一个是+1，即V（发送信号）操作
    */
    short sem_op;

    /*
        通常为SEM_UNDO,使操作系统跟踪信号
        并在进程没有释放该信号量而终止时，操作系统释放信号量
    */
    short sem_flg;
};
```

- `int semctl(int sem_id, int sem_num, int command, ...);` // 直接控制信号量信息 (删除, 传值)

## 与互斥锁的区别

**信号量** 信号量是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行等待（即 P(信号变量)）和发送（即 V(信号变量)）信息操作。我们通常通过信号来解决多个进程对同一资源的访问竞争的问题，使在任一时刻只能有一个执行线程访问代码的临界区域，也可以说它是协调进程间的对同一资源的访问权，也就是用于同步进程的。

## 4.13.5 共享内存

### 虚拟内存

程序代码和数据必须驻留在内存中才能得以运行，然而系统内存数量很有限，往往不能容纳一个完整程序的所有代码和数据，更何况在多任务系统中，可能需要同时打开子处理程序，画图程序，浏览器等很多任务，想让内存驻留所有这些程序显然不太可能。因此首先能想到的就是将程序分割成小份，只让当前系统运行它所有需要的那部分留在内存，其它部分都留在硬盘。当系统处理完当前任务片段后，再从外存中调入下一个待运行的任务片段。的确，老式系统就是这样处理大任务的，而且这个工作是由程序员自行完成。但是随着程序语言越来越高级，程序员对系统体系的依赖程度降低了，很少有程序员能非常清楚的驾驭系统体系，因此放手让程序员负责将程序片段化和按需调入轻则降低效率，重则使得机器崩溃；再一个原因是随着程序越来越丰富，程序的行为几乎无法准确预测，程序员自己都很难判断下一步需要载入哪段程序。因此很难再靠预见性来静态分配固定大小的内存，然后再机械地轮换程序片进入内存执行。系统必须采取一种能按需分配而不需要程序员干预的新技术。

**虚拟内存**（之所以称为虚拟内存，是和系统中的逻辑内存和物理内存相对而言的，**逻辑内存**是站在进程角度看到的内存，因此是程序员关心的内容。而**物理内存**是站在处理器角度看到的内存，由操作系统负责管理。

**虚拟内存**可以说是映射到这两种不同视角内存的一个技术手段。）技术就是一种由操作系统接管的按需动态内存分配的方法，它允许程序不知不觉中使用大于实际物理空间大小的存储空间（其实是将程序需要的存储空间以页的形式分散存储在物理内存和磁盘上），所以说虚拟内存彻底解放了程序员，从此程序员不用过分关心程序的大小和载入，可以自由编写程序了，繁琐的事情都交给操作系统去做吧。

**虚拟内存**是将系统硬盘空间和系统实际内存联合在一起供进程使用，给进程提供了一个比内存大得多的虚拟空间。在程序运行时，只要把虚拟地址空间的一小部分映射到内存，其余都存储在硬盘上（也就是说程序虚拟空间就等于实际物理内存加部分硬盘空间）。当被访问的虚拟地址不在内存时，则说明该地址未被映射到内存，而是被存贮在硬盘中，因此需要的虚拟存储地址随即被调入到内存；同时当系统内存紧张时，也可以把当前不用的虚拟存储空间换出到硬盘，来腾出物理内存空间。系统如此周而复始地运转——换入、换出，而用户几乎无法查觉，这都是拜虚拟内存机制所赐。

### mmap 系列

mmap 函数用来将文件映射进内存。需要指出的是这里的内存指的是**虚拟内存**。

mmap 函数可以将一个文件的内容映射到内存，这样就可以直接对该内存进行操作，从而省去IO

操作。

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

1. **start**: 映射区的开始地址 (一般设置为 NULL, 让系统决定位置)

2. **length**: 映射区的长度

3. **prot**: 期望的内存保护标志, 不能与文件的打开模式冲突。

(a) **PROT\_EXEC**: 页内容可以被执行

(b) **PROT\_READ**: 页内容可以被读取

(c) **PROT\_WRITE**: 页可以被写入

(d) **PROT\_NONE**: 页不可访问

4. **flags**: 指定映射对象的类型, 映射选项和映射页是否可以共享。

(a) **MAP\_SHARED** 与其它所有映射这个对象的进程共享映射空间。

(b) **MAP\_PRIVATE** 建立一个写入时拷贝的私有映射。

(c) **MAP\_ANONYMOUS** 匿名映射, 映射区不与任何文件关联

5. **fd**: 有效的文件描述词。如果**MAP\_ANONYMOUS** 被设定, 为了兼容问题, 其值应为-1

6. **offset**: 被映射对象内容的起点

```
int munmap( void * addr, size_t len );
```

该调用在进程地址空间中解除一个映射关系, **addr** 是调用 **mmap()** 时返回的地址, **len** 是映射区的大小;

例子 ->

```
int main(int argc, char *argv[]){
    int fd, len;
    char *ptr;
    if(argc < 2){
        printf("please enter a file\n");
        return 0;
    }
    if((fd = open(argv[1], O_RDWR)) < 0){
        perror("open file error");
        return -1;
    }
}
```

```

    }
    len=lseek(fd,0,SEEK_END);

    ptr=mmap(NULL,len,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);//读写得和open函数的标志相一致,
    否则会报错

    if(ptr==MAP_FAILED){
        perror("mmap_error");
        close(fd);
        return -1;
    }
    close(fd);//关闭文件也ok
    printf("length_is_%d\n",strlen(ptr));
    printf("the_s_content_is:\n%s\n",argv[1],ptr);
    ptr[0]='c';//修改其中的一个内容
    printf("the_s_content_is:\n%s\n",argv[1],ptr);

    munmap(ptr,len);//将改变的文件写入内存

    return 0;
}

```

## shmget 系列

共享内存并未提供同步机制，也就是说，在第一个进程结束对共享内存的写操作之前，并无自动机制可以阻止第二个进程开始对它进行读取。所以我们通常需要用其他的机制来同步对共享内存的访问

- `int shmget(key_t key, size_t size, int shmflg);` 创建共享内存函数
- `void *shmat(int shmid, const void *shmaddr, int shmflg);` Map 共享内存区
- `int shmdt(const void *shmaddr);` unMap
- `int shmctl(int shmid, int cmd,struct shmid_ds* buf);` 删除共享内存或管理

例子 -> <https://github.com/ctzhenghua/C-NetworkPractice-Code/blob/Dev/IPCs/SharedMem/shmRead.cc>

## 4.14 阻塞与非阻塞、同步与异步

### 4.14.1 同步/异步主要针对 C 端

- **同步**：所谓同步，就是在 c 端发出一个功能调用时，在没有得到结果之前，该调用就不返回。也就是必须一件一件事做，等前一件做完了才能做下一件事

例如普通 B/S 模式（同步）：提交请求-> 等待服务器处理-> 处理完毕返回这个期间客户端浏览器不能干任何事

- **异步**：异步的概念和同步相对。当 c 端一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者

例如 *ajax* 请求（异步）：请求通过事件触发-> 服务器处理（这是浏览器仍然可以作其他事情）-> 处理完毕

### 4.14.2 阻塞/非阻塞主要针对 S 端

- **阻塞**：阻塞调用是指调用结果返回之前，当前线程会被挂起（线程进入非可执行状态，在这个状态下，cpu 不会给线程分配时间片，即线程暂停运行）。函数只有在得到结果之后才会返回。

有人也许会把阻塞调用和同步调用等同起来，实际上他是不同的。对于同步调用来说，很多时候当前线程还是激活的，只是从逻辑上当前函数没有返回而已。例如，我们在 socket 中调用 *recv* 函数，如果缓冲区中没有数据，这个函数就会一直等待，直到有数据才返回。而此时，当前线程还会继续处理各种各样的消息。

快递的例子：比如到你某个时候到 A 楼一层（假如是内核缓冲区）取快递，但是你不知道快递什么时候过来，你又不能干别的事，只能死等着。但你可以睡觉（进程处于休眠状态），因为你知道快递把货送来时一定会给你打个电话（假定一定能叫醒你）。

- **非阻塞**：非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。

还是等快递的例子：如果用忙轮询的方法，每隔 5 分钟到 A 楼一层（内核缓冲区）去看快递来了没有。如果没来，立即返回。而快递来了，就放在 A 楼一层，等你去取。

### 4.14.3 Linux 下的 5 种 I/O 模型

#### 阻塞 I/O: Blocking I/O

进程会一直阻塞，直到数据拷贝完成

应用程序调用一个 IO 函数，导致应用程序阻塞，等待数据准备好。如果数据没有准备好，一直等待…。数据准备好了，从内核拷贝到用户空间,IO 函数返回成功指示。

我们第一次接触到的网络编程都是从 `listen()`、`send()`、`recv()` 等接口开始的。使用这些接口可以很方便的构建服务器 / 客户机的模型。

**阻塞 I/O 模型图**：在调用 `recv()/recvfrom()` 函数时，发生在内核中等待数据和复制数据的过程。

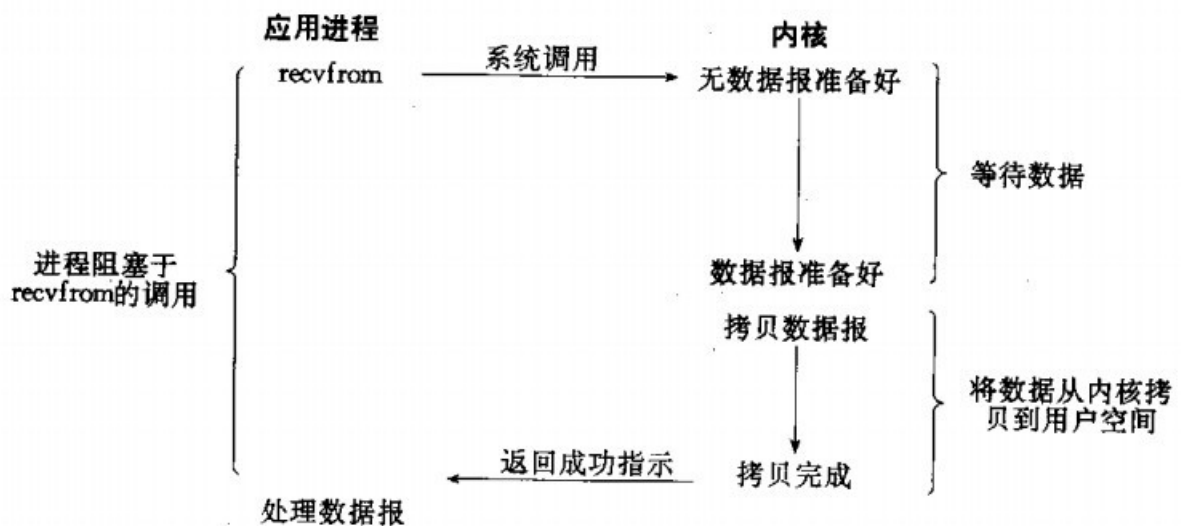


图 6.1 阻塞 I/O 模型

当调用 `recv()` 函数时，系统首先查是否有准备好的数据。如果数据没有准备好，那么系统就处于等待状态。当数据准备好后，将数据从系统缓冲区复制到用户空间，然后该函数返回。在套接应用程序中，当调用 `recv()` 函数时，未必用户空间就已经存在数据，那么此时 `recv()` 函数就会处于等待状态。

当使用 `socket()` 函数和 `WSASocket()` 函数创建套接字时，默认的套接字都是阻塞的。这意味着当调用 Windows Sockets API 不能立即完成时，线程处于等待状态，直到操作完成。

并不是所有 Windows Sockets API 以阻塞套接字为参数调用都会发生阻塞。例如，以阻塞模式的套接字为参数调用 `bind()`、`listen()` 函数时，函数会立即返回。将可能阻塞套接字的 Windows Sockets API 调用分为以下四种：

1. 输入操作：recv()、recvfrom()、WSARecv() 和 WSARecvfrom() 函数。以阻塞套接字为参数调用该函数接收数据。如果此时套接字缓冲区内没有数据可读，则调用线程在数据到来前一直睡眠
2. 输出操作：send()、sendto()、WSASend() 和 WSASendto() 函数。以阻塞套接字为参数调用该函数发送数据。如果套接字缓冲区没有可用空间，线程会一直睡眠，直到有空间
3. 接受连接：accept() 和 WSAAccept() 函数。以阻塞套接字为参数调用该函数，等待接受对方的连接请求。如果此时没有连接请求，线程就会进入睡眠状态
4. 外出连接：connect() 和 WSAConnect() 函数。对于 TCP 连接，客户端以阻塞套接字为参数，调用该函数向服务器发起连接。该函数在收到服务器的应答前，不会返回。这意味着 TCP 连接总会等待至少到服务器的一次往返时间

使用阻塞模式的套接字，开发网络程序比较简单，容易实现。当希望能够立即发送和接收数据，且处理的套接字数量比较少的情況下，使用阻塞模式来开发网络程序比较合适。

阻塞模式套接字的不足表现为，在大量建立好的套接字线程之间进行通信时比较困难。当使用“生产者-消费者”模型开发网络程序时，为每个套接字都分别分配一个读线程、一个处理数据线程和一个用于同步的事件，那么这样无疑加大系统的开销。其最大的缺点是当希望同时处理大量套接字时，将无从下手，其扩展性很差。

阻塞模式给网络编程带来了一个很大的问题，如在调用 send() 的同时，线程将被阻塞，在此期间，线程将无法执行任何运算或响应任何的网路请求。这给多客户机、多业务逻辑的网络编程带来了挑战。这时，我们可能会选择多线程的方式来解决这个问题。

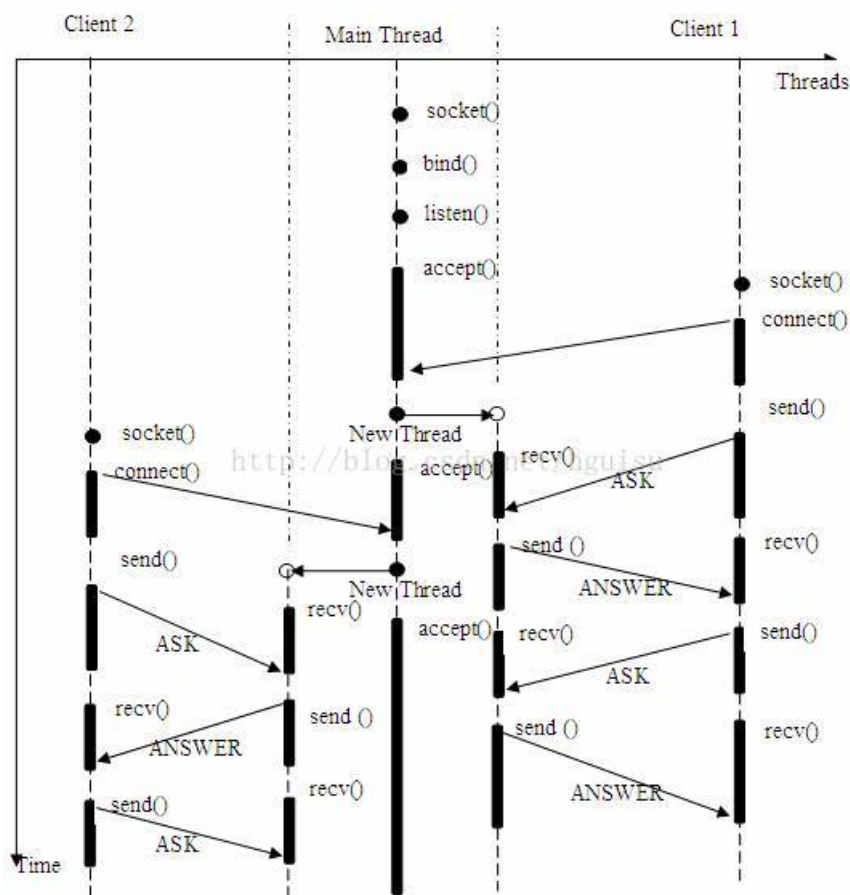
应对多客户机的网路应用，最简单的解决方式是在服务器端使用多线程（或多进程）。多线程（或多进程）的目的是让每个连接都拥有独立的线程（或进程），这样任何一个连接的阻塞都不会影响其他的连接。

具体使用多进程还是多线程，并没有一个特定的模式。传统意义上，进程的开销要远远大于线程，所以，如果需要同时为较多的客户机提供服务，则不推荐使用多进程；如果单个服务执行体需要消耗较多的 CPU 资源，譬如需要进行大规模或长时间的数据运算或文件访问，则进程较为安全。通常，使用 pthread\_create() 创建新线程，fork() 创建新进程。

多线程/进程服务器同时为多个客户机提供应答服务。模型如下：

主线程持续等待客户端的连接请求，如果有连接，则创建新线程，并在新线程中提供为前例同样的问答服务。

上述多线程的服务器模型似乎完美的解决了为多个客户机提供问答服务的要求，但其实并不尽然。如果要同时响应成百上千路的连接请求，则无论多线程还是多进程都会严重占据系统资



源，降低系统对外界响应效率，而线程与进程本身也更容易进入假死状态。

由此可能会考虑使用“线程池”或“连接池”。“线程池”旨在减少创建和销毁线程的频率，其维持一定合理数量的线程，并让空闲的线程重新承担新的执行任务。“连接池”维持连接的缓存池，尽量重用已有的连接、减少创建和关闭连接的频率。这两种技术都可以很好的降低系统开销，都被广泛应用很多大型系统，如 apache，MySQL 数据库等。

但是，“线程池”和“连接池”技术也只是在一定程度上缓解了频繁调用 IO 接口带来的资源占用。而且，所谓“池”始终有其上限，当请求大大超过上限时，“池”构成的系统对外界的响应并不比没有池的时候效果好多少。所以使用“池”必须考虑其面临的响应规模，并根据响应规模调整“池”的大小。

对应上例中的所面临的可能同时出现的上千甚至上万次的客户端请求，“线程池”或“连接池”或许可以缓解部分压力，但是不能解决所有问题。

## 非阻塞 I/O: nonBlocking I/O

非阻塞 IO 通过进程反复调用 IO 函数（多次系统调用，并马上返回）；在数据拷贝的过程中，进程是阻塞的

我们吧一个 SOCKET 接口设置为非阻塞就是告诉内核，当所请求的 I/O 操作无法完成时，



不要将进程睡眠，而是返回一个错误。这样我们的 I/O 操作函数将不断的测试数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。在这个不断测试的过程中，会大量的占用 CPU 的时间。

把 SOCKET 设置为非阻塞模式，即通知系统内核：在调用 Windows Sockets API 时，不要让线程睡眠，而应该让函数立即返回。在返回时，该函数返回一个错误代码。图所示，一个非阻塞模式套接字多次调用recv() 函数的过程。前三次调用recv() 函数时，内核数据还没有准备好。因此，该函数立即返回 WSAEWOULDBLOCK 错误代码。第四次调用recv() 函数时，数据已经准备好，被复制到应用程序的缓冲区中，recv() 函数返回成功指示，应用程序开始处理数据。

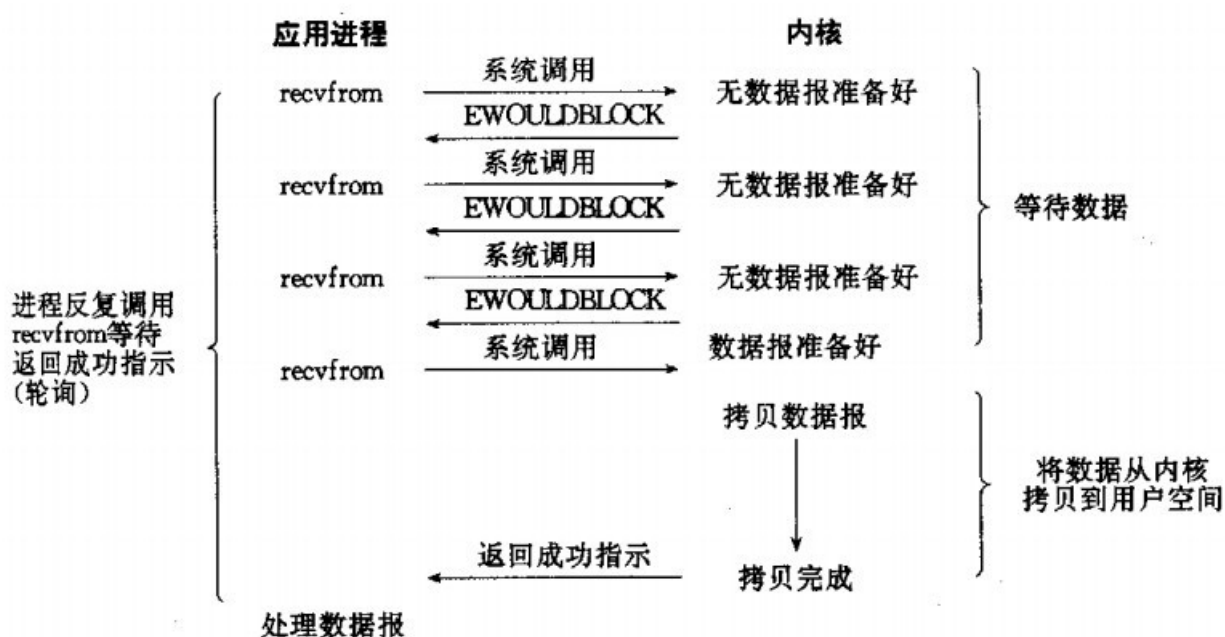


图 6.2 非阻塞 I/O 模型

当使用 `socket()` 函数和 `WSASocket()` 函数创建套接字时，默认都是阻塞的。在创建套接字之后，通过调用 `ioctlsocket()` 函数，将该套接字设置为非阻塞模式。Linux 下的函数是 `fcntl()`。套接字设置为非阻塞模式后，在调用 Windows Sockets API 函数时，调用函数会立即返回。大多数情况下，这些函数调用都会调用“失败”，并返回 `WSAEWOULDBLOCK` 错误代码。说明请求的操作在调用期间内没有时间完成。通常，应用程序需要重复调用该函数，直到获得成功返回代码。

需要说明的是并非所有的 Windows Sockets API 在非阻塞模式下调用，都会返回 `WSAEWOULDBLOCK` 错误。例如，以非阻塞模式的套接字为参数调用 `bind()` 函数时，就不会返回该错误代码。当然，在调用 `WSAStartup()` 函数时更不会返回该错误代码，因为该函数是应用程序第一调用的函数，当然不会返回这样的错误代码。

要将套接字设置为非阻塞模式，除了使用 `ioctlsocket()` 函数之外，还可以使用 `WSAAsyncselect()` 和 `WSAEventselect()` 函数。当调用该函数时，套接字会自动地设置为非阻塞方式。

由于使用非阻塞套接字在调用函数时，会经常返回 `WSAEWOULDBLOCK` 错误。所以在任何时候，都应仔细检查返回代码并作好对“失败”的准备。应用程序连续不断地调用这个函数，直到它返回成功指示为止。上面的程序清单中，在 `While` 循环体内不断地调用 `recv()` 函数，以读入 1024 个字节的数据。这种做法很浪费系统资源。

要完成这样的操作，有人使用 `MSG_PEEK` 标志调用 `recv()` 函数查看缓冲区中是否有数据可读。同样，这种方法也不好。因为该做法对系统造成的开销是很大的，并且应用程序至少要调用 `recv()` 函数两次，才能实际地读入数据。较好的做法是，使用套接字的“I/O 模型”来判断非阻塞套接字是否可读可写。

非阻塞模式套接字与阻塞模式套接字相比，不容易使用。使用非阻塞模式套接字，需要编写更多的代码，以便在每个 Windows Sockets API 函数调用中，对收到的 `WSAEWOULDBLOCK` 错误进行处理。因此，非阻塞套接字便显得有些难于使用。

但是，非阻塞套接字在控制建立的多个连接，在数据的收发量不均，时间不定时，明显具有优势。这种套接字在使用上存在一定难度，但只要排除了这些困难，它在功能上还是非常强大的。通常情况下，可考虑使用套接字的“I/O 模型”，它有助于应用程序通过异步方式，同时对一个或多个套接字的通信加以管理。

## I/O 复用：I/O multiplexing

主要是 `select` 和 `epoll`；对一个 IO 端口，两次调用，两次返回，比阻塞 IO 并没有什么优越性；关键是能实现同时对多个 IO 端口进行监听

**I/O 复用模型**会用到 `select`、`poll`、`epoll` 函数，这几个函数也会使进程阻塞，但是和阻塞 I/O 所不同的，这两个函数可以同时阻塞多个 I/O 操作。而且可以同时多个读操作，多个写操作的 I/O 函数进行检测，直到有数据可读或可写时，才真正调用 I/O 操作函数。

## 信号驱动 I/O：SIGIO

## 异步 I/O：asynchronous I/O

只有这一种才是异步 IO。数据拷贝的时候进程无需阻塞

当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成时，通过状态、通知和回调来通知调用者的输入输出操作

同步 IO 引起进程阻塞，直至 IO 操作完成。

异步 IO 不会引起进程阻塞。

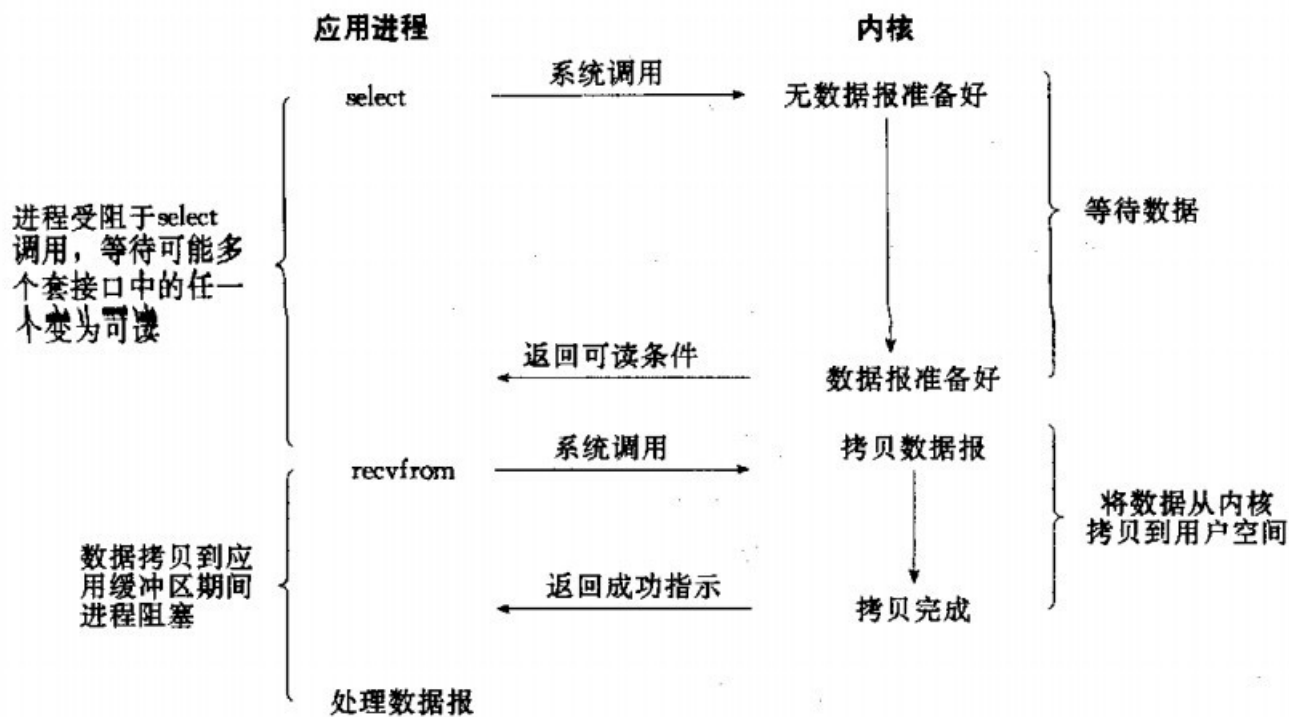


图 6.3 I/O 复用模型

IO 复用是先通过 select 调用阻塞

#### 4.14.4 参考

<http://blog.csdn.net/hguisu/article/details/7453390>

### 4.15 Select 模型：管理多个 IO

当一个服务器在处理多个客户时，它绝对不能阻塞于只与单个客户相关的某个函数调用，否则可能导致服务器挂起，拒绝为所有用户提供服务，所以是不是可以考虑半同步半异步的模式或者池的概念。

#### 4.15.1 相关函数

- `int select(int nfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval* timeout);`

– null: 永远等待下去，仅在有一个描述符准备好 I/O 才返回

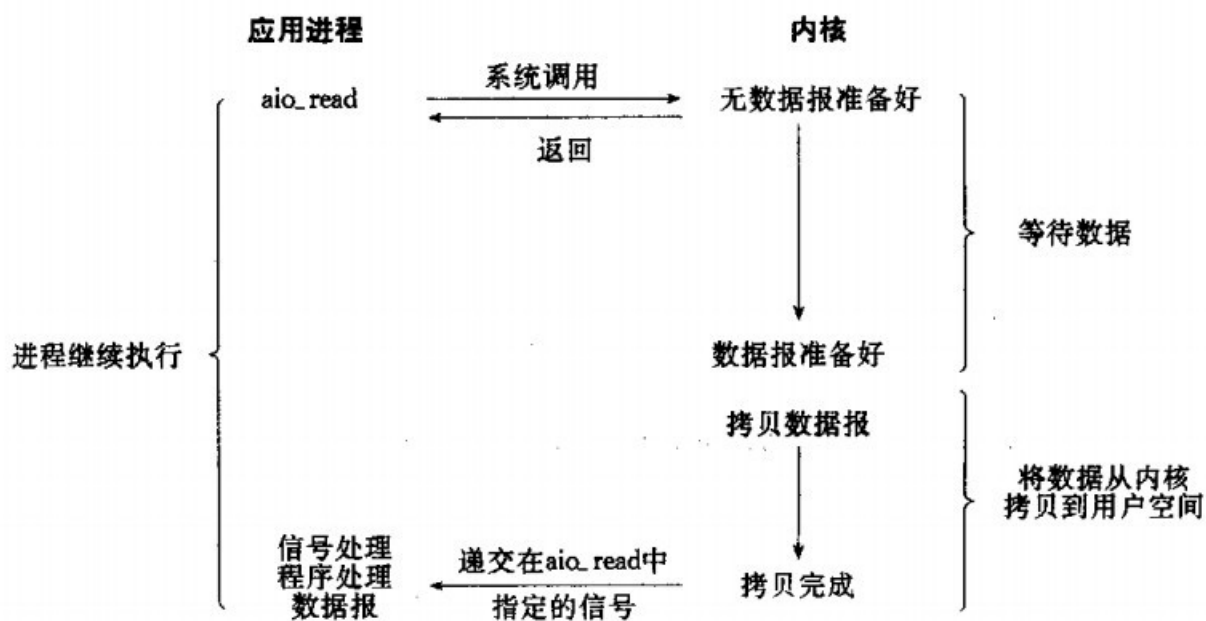


图 6.5 异步 I/O 模型

- 0: 根本不等待, 检查描述符后立即返回, 这称为轮询方式
- timeval 结构: 等待结构中固定时间。

用 select 管理多个 IO, 一旦其中一个或多个 IO 检测到我们所感兴趣的事件, 就函数返回检测到的事件个数和那些发生事件的 IO 标识符, 然后我们遍历这些事件, 分别处理

- 调用select 函数时fd\_set 为值。
- 返回时的fd\_set 为结果, 即哪些描述符就绪了。
- 在fd\_set 中, 任何与 未就绪描述符对应的位, 在返回时均清为 0。

-> Notice : 所以, 每次重新调用select 函数时, 我们都得再次把所有关心的描述符的位均置 1.

- 该函数的返回值表示所有描述符集的已就绪的总位数
- 如果是定时器到时并且没有就绪的描述符, 返回 0
- 出错, 返回-1

- void FD\_ZERO(fd\_set \*set): 将集合清空
- void FD\_SET(int fd,fd\_set \*set): 将 fd 添加到集合 set 中
- void FD\_CLR(int fd,fd\_set \*set): 将 fd 从集合 set 中移除
- int FD\_ISSET(int fd,fd\_set \*set): 判断 fd 是否在 set 中

- 命令行: `ulimit -n` 最大并发数: 用该命令可以调整最大的并发个数

## 可读事件发生条件

- 套接口缓冲区有数据可读: 该套接字的接收缓冲区中的数据字节数大于等于套接字接受缓冲区低水位标记的当前大小。

可以使用 `SO_RCVLOWAT` 选项设置该套接字的低水位标记。对于TCP与UDP 默认为 1.

- 连接的读一半关闭, 即接受到 **FIN** 段, 读操作将返回 0
- 如果是监听套接口, 且已完成连接队列不为空时
- 套接口上发生了一个错误待处理: 对这样的套接字的读操作将不阻塞并返回-1, 同时把 `errno` 设置为确切的错误条件。

这些待处理错误可以通过 `getsockopt` 指定 `SO_ERROR` 选项来获取并清除。

## 可写事件发生条件

- 套接口发送缓冲区的可用数据字节数大于等于套接字发送缓冲区的低水位标记的当前大小。

可以使用 `SO_SNDLOWAT` 来设置套接字的低水位标记, 对于TCP与UDP 默认为2048

- 连接的写一半关闭。即收到RST 段之后, 再次调用 `write` 操作-> 产生 `SIGPIPE` 信号
- 使用非阻塞式的 `connect` 的套接字已建立连接, 或者 `connect` 已经以失败告终。
- 套接口上发生一个错误待处理, 对这样的套接字的写操作将不阻塞并返回-1, 同时把 `errno` 设置为确切的错误条件。

错误可以通过 `getsockopt` 指定 `SO_ERROR` 选项来获取并清除。

-> Notice: 当套接字发生错误时, 它将由 `select` 标记为即可读又可写。

## 异常事件发生条件

- 套接口存在带外数据, `urgent` 位为 1.

**最大处理描述符值** 对于 `select` 函数, 所能处理的最大客户数目的限制是以下两个值中的较小者

- FD\_SETSIZE
- 内核允许进程打开的最大描述符的值

是描述符 fd 的值而不是个数:<http://blog.csdn.net/adam040606/article/details/46833617>

->Notice : 但是限制了最大值那么就限制了最大的个数。

## 4.15.2 使用 select 实现回射服务程序

### Client

```
// Using Select to Implemet IO

... sock stuff

fd_set rset;
FD_ZERO(&rset);

int nready;
int maxfd;
int fd_stdin = fileno(stdin);
maxfd = max(sock,fd_stdin);

while(1)
{
    FD_SET(fd_stdin, &rset);
    FD_SET(sock, &rset);

    nready = select(maxfd+1, &rset,NULL, NULL, NULL);
    if(nready == -1)
        exit(1);
    if(nready == 0)
        continue;
    if(FD_ISSET(sock, &rset)) // Sock 在发生事件的集合中
    {
        int ret = read(sock, recvbuf, sizeof(recvbuf));
        if(ret == -1)
            exit(1);
        else if(ret == 0)
        {
            cout<<"server_Closed";
            break;
        }
    }
}
```

```

    }
    fputs(recvbuf, stdout);
    memset(recvbuf,0,sizeof(recvbuf));
}
if(FD_ISSET(fd_stdin, &rset)) // 键盘事件在集合中
{
    if(fgets(sendbuf,sizeof(sendbuf),stdin) == NULL)
        break;
    write(sock,sendbuf,sizeof(sendbuf));
}
}
close(sock)

```

## Server

```

// Server Using select to Implement:
// 可以使用 Max_i 来确定select遍历的最大发生事件的下标，从而在 遍历处理时可以相对提高效率。

...Socket stuff:listenfd...
...Bind and Listen...

int client[FD_SETSIZE]; // select 最大处理的用户个数
for(auto d: client)
    d = -1; // 使得client 空闲

fd_set rset;
fd_set allSet;
FD_ZERO(&rset);
FD_ZERO(&allSet);
FD_SET(listenfd,&allSet);
while(1)
{
    rset = allSet;
    nready = select(maxfd+1, &rset, NULL, NULL, NULL);
    if(nready == -1)
    {
        if(errno == EINTR) // 信号中断
            continue;
        exit(1);
    }
    if(nready == 0) // 超时
    {
        continue;
    }
}

```

```

}
// 监听事件
if(FD_ISSET(listenfd,&rset))
{
    conn = accept(listenfd,(struct sockaddr*)&peeradr, &peerLen);

    if(conn == -1)
        exit(1);

    // 将当前连接 存储在已连接客户区
    bool flag = false;
    for(auto&d :client)
    {
        if(d == -1)
        {
            d = conn;
            flag = true;
            break;
        }
    }
    if(!flag)// 没找到空闲位置
    {
        cout<<"Too_Many_Client"<<endl;
        exit(1);
    }
    cout<< "ip="<< inet_ntoa(peeradr.sin_addr);
    FD_SET(conn,&allSet);
    if(conn > maxfd)
        maxfd = conn;

    // 如果已经将监听到的事件处理完
    if(--nready < 0)
        continue;
}

// 套接口发生事件
for(auto conn:client)
{
    if(conn == -1)
        continue;
    if(FD_ISSET(conn,&rset))
    {
        ret = read(conn, recvbuf, 1024);
        if(ret == -1)

```



```

        exit(1);
    if(ret == 0)
    {
        cout<<"client_Close"<<endl;
        // 将此连接从集合清除
        FD_CLR(conn,&allSet);

    }
    fputs(recvbuf, stdout);
    write(conn, recvbuf, strlen(recvbuf));

    if(--nready <= 0)
        break;
    }
}
}

```

## 4.16 Poll 模型：不再限制最大描述符值

### 4.16.1 相关函数

- `int poll(struct pollfd fd[], nfds_t nfds, int timeout);`

— 第 1 个参数, 关心的描述符与其上的事件的结构数组

```

struct pollfd{
    int fd; //文件描述符
    short events; //感兴趣的事件
    short revents; //发生的事件
};

```

\* POLLIN 数据可读事件

\* POLLOUT 可写

\* POLLERR 错误

— 第 2 个参数`nfds`: 要监视的描述符的数目, 等于结构数组的元素个数

— 第 3 个参数`timeout`: 是一个用毫秒表示的时间, 是指定 poll 在返回前没有接收事件时应该等待的时间。

\* -1: 一直等待直到有事件发送

\* 0 : 不等待直接返回

\* n: 等待 n 毫秒

-> 返回值

\* 该函数的返回值表示所有描述符集的已就绪的总个数，即 revents 成员值非 0 的个数。

\* 如果是定时器到时并且没有就绪的描述符，返回 0

\* 出错，返回-1

-> Typical Code

```
for(i = 1; i < maxi; ++i){
    if(client[i].revents & (POLLRDNORM | POLLERR))
    { }
}
```

## 4.16.2 实例

<http://www.cnblogs.com/Anker/archive/2013/08/15/3261006.html>

## 4.17 Epoll 模型：实现更高效率

### 4.17.1 相关函数

- `int epoll_create(int size);` hash 表实现方式
- `int epoll_create1(int flags);` 红黑树实现方式
- `int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);`

1. 第一个参数是epoll 描述符，即`epoll_create()` 的返回值。

2. 第二个参数表示动作

- `EPOLL_CTL_ADD`: 注册新的fd 到epfd 中;
- `EPOLL_CTL_MOD`: 修改已经注册的fd 的监听事件;
- `EPOLL_CTL_DEL`: 从epfd 中删除一个fd;

3. 第三个参数是需要监听的fd。

4. 第四个参数是告诉内核需要监听什么事，`struct epoll_event` 结构如下：

```

typedef union epoll_data {
    void *ptr;
    int fd;
    __uint32_t u32;
    __uint64_t u64;
} epoll_data_t;

//感兴趣的事件和被触发的事件
struct epoll_event {
    __uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
};

```

- EPOLLIN: 对应的文件描述符可以读
- EPOLLOUT: 对应的文件描述符可以写
- EPOLLET: 边缘触发模式

->Typical Code

```

epollfd = epoll_create1(EPOCH_CLOEXEC);

struct epoll_event event;
event.data.fd = listenfd;
event.events = EPOLLIN | EPOLLET;
epoll_ctl(epollfd, EPOLL_CTL_ADD, listenfd, &event);

```

- int epoll\_wait(int epfd, struct epoll\_event \*events, int maxevents, int timeout);

## 4.17.2 使用 epoll 实现回射服务器

better:<http://www.cnblogs.com/Anker/archive/2013/08/17/3263780.html>

### Server

```

#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/epoll.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <fcntl.h>

```

```

#include <stdlib.h>
#include <stdio.h>
#include <errono.h>
#include <string.h>

#include <vector>
#include <algorithm>

typedef std::vector<struct epoll_event> EventList;
#define ERR_EXIT(m) \
    do \
    { \
        perror(m);\
        exit(EXIT_FAILURE);\
    }while(0)

void activate_nonblock(int fd)
{
    int ret;
    int flags = fcntl(fd, F_GETFL);
    if(flags == -1)
        ERR_EXIT("fcntl");

    flags |= O_NONBLOCK;
    ret = fcntl(fd, F_SETFL, flags);
    if(flags == -1)
        ERR_EXIT("fcntl");
}

// 实现 读n个 ,解决TCP粘包问题
ssize_t readn(int fd, void *buf, size_t count)
{
    size_t nleft = count;
    ssize_t nread;
    char* bufp = (char*) buf;

    while(nleft > 0)
    {
        if((nread = read(fd, bufp, nleft)) < 0)
        {
            if(errno == EINTR)
                continue;
            return -1;
        }
    }
}

```

```

        else if(nread == 0)
            return count-nleft
        bufp += nread;
        nleft -= nread;
    }
    return count;
}

```

```

ssize_t writen(int fd, const void* buf, size_t count)
{
    size_t nleft = count;
    ssize_t nwritten;
    char* bufp = (char*)buf;

    while(nleft > 0)
    {
        if((nwritten = write(fd, bufp, nleft)) < 0)
        {
            if(errno == EINTR)
                continue;
            return -1;
        }
        else if(nwritten == 0)
            continue;
        bufp += nwritten;
        nleft -= nwritten;
    }
    return count;
}

```

// 实现读取一行

```

ssize_t readline(int sockfd, void *buf, size_t maxline)
{
    int ret;
    int nread;
    char *bufp = (char*)buf;
    int nleft = maxline;
    while(1)
    {
        ret = recv_peek(sockfd, bufp, nleft);
        if(ret < 0)
            return ret;
        else if(ret == 0)
            return ret;
    }
}

```

```

nread = ret;
for(int i = 0; i < nread; ++i)
{
    if(bufp[i] == '\n')
    {
        ret = readn(sockfd, bufp,i+1);
        if(ret != i+1)
            exit(EXIT_FAILURE);

        return ret;
    }
}

if(nread > nleft)
    exit(EXIT_FAILURE);

nleft -= nread;
ret = readn(sockfd, bufp, nread);
if(ret != nread)
    exit(EXIT_FAILURE);

bufp += nread;
}
return -1;
}

// 查看收到了多少字节的数据， 但是不从内存中拷走(recv),用read的话会删除的
ssize_t recv_peek(int sockfd, void *buf, size_t len)
{
    while(1)
    {
        int ret = recv(sockfd, buf, len, MSG_PEEK);
        if(ret == -1 && errno == EINTR)
            continue;
        return ret;
    }
}

int main()
{
    // 忽略SIGPIPE 信号，防止意外退出。
    signal(SIGPIPE, SIG_IGN);
    int count = 0;

```

```

int listenfd;
if((listenfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    ERR_EXIT("socket");
struct sockaddr_in servaddr;
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(5188);
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

// 使用SO_REUSEADDR , 可以让服务器崩溃后立即重启, 不会存在地址占有的错误
int on = 1;
if(setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0)
    ERR_EXIT("setsockopt");

if(bind(listenfd, (struct sockaddr*)servaddr, sizeof(servaddr)) < 0)
    ERR_EXIT("bind");

if(listen(listenfd, SOMAXCONN) < 0)
    ERR_EXIT("listen");

std::vector<int> clients;
int epollfd;
epollfd = epoll_create1(EPOOL_CLOEXEC);

struct epoll_event event;
event.data.fd = listenfd;
event.events = EPOLLIN | EPOLLET;
epoll_ctl(epollfd, EPOLL_CTL_ADD, listenfd, &event);

EventList events(16);
struct sockaddr_in peeraddr;
socklen_t peerlen;
int conn;
int i;

int ready;
while(1)
{
    nready = epoll_wait(epollfd, &*events.begin(), static_cast<int>(events.size()), -1);
    if(nready == -1)
    {
        if(errno == EINTR)
            continue;
        ERR_EXIT("epoll_wait");
    }
}

```

```

}
if(nread == 0)
    continue;
if((size_t)nready == events.size())
    events.resize(events.size()*2);

for(i = 0; i < nready; ++i)
{
    if(events[i].data.fd == listenfd)
    {
        peerlen = sizeof(peeraddr);
        conn = accept(listenfd, (struct sockaddr*)&peeraddr, &peerlen);
        if(conn == -1)
            ERR_EXIT("accept");

        printf("ip=%s port=%d\n", inet_ntoa(peeraddr.sin_addr), ntohs(peeraddr.
            sin_port));

        printf("count=%d\n", ++count);
        clients.push_back(conn);

        activate_nonblock(conn);

        event.data.fd = conn;
        event.events = EPOLLIN | EPOLLET;
        epoll_ctl(epollfd, EPOLL_CTL_ADD, conn, &event);
    }

    else if(events[i].events & EPOLLIN)
    {
        conn = events[i].data.fd;
        if(conn < 0)
            continue;
        char recvbuf[1024] = {0};
        int ret = readline(conn, recvbuf, 1024);
        if(ret == -1)
            ERR_EXIT("readline");
        if(ret == 0)
        {
            printf("client_close\n");
            close(conn);

            event = events[i];
            epoll_ctl(epollfd, EPOLL_CTL_DEL, conn, &event);
        }
    }
}

```



```

        clients.erase(std::remove(clients.begin(), clients.end(), conn), clients.
            end());
    }
    fputs(recvbuf, stdout);
    writen(conn, recvbuf, strlen(recvbuf));
}

}

}

```

### 4.17.3 Epoll 触发方式

**EPOLLIT** 完全靠Linux-kernel-epoll 驱动，应用程序只需要处理从epoll\_wait 返回的fds， 这些 fds 我们认为它们处于就绪状态，。

**EPOLLET** 此模式下，系统仅仅通知应用程序哪些 fds 变成了就绪状态，一旦 fd 变成就绪状态，epoll 将不再关注这个 fd 的任何状态信息 (从 epoll 队列移除)，直到应用程序通过读写操作（非阻塞）触发EAGAIN 状态，epoll 认为这个 fd 又变为空闲状态，那么 epoll 又重新关注这个 fd 的状态变化 (重新加入 epoll 队列)。

-> 对于采用 LT 工作模式的文件描述符：当 `epoll_wait` 检测到其上有事件发生并将此事件通知应用程序后，应用程序可以不立即处理该事件。这样，当应用程序下一次调用 `epoll_wait` 时，`epoll_wait` 还会再次向应用程序通告此事件，直到该事件被处理。

-> 对于采用 ET 工作模式的文件描述符：当 `epoll_wait` 检测到其上有事件发生并将此事件通知应用程序后，应用程序必须立即处理该事件，因为后续的 `epoll_wait` 调用将不再向应用程序通知这一事件。

可见，ET 模式在很大程度上降低了同一个 epoll 事件被重复触发的次数，因此效率要比 LT 模式高。

随着`epoll_wait`的返回，队列中的`fds`是在减少的，所以在高并发的系统中，`EPOLLET`更有优势，但是对程序员的要求也更高，因为有可能出现数据读取不完整的问题，举例如下：

假设现在对方发送了 2k 的数据，而我们先读取了 1k，然后这时调用了 `epoll_wait`，如果是边沿触发 ET，那么这个 fd 变成就绪状态就会从 `epoll` 队列移除，则 `epoll_wait` 会一直阻塞，忽略尚未读取的 1k 数据；而如果是水平触发 LT，那么 `epoll_wait` 还会检测到可读事件而返回，我们可以继续读取剩下的 1k 数据。

因此总结来说: LT 模式可能触发的次数更多, 一旦触发的次数多, 也就意味着效率会下降; 但这样也不能就说 LT 模式就比 ET 模式效率更低, 因为 ET 的使用对编程人员提出了更高更精细的要求, 一旦编程人员水平达不到 (比如本人), 那 ET 模式还不如 LT 模式;

-> Notice: 每个使用 ET 模式的 文件描述符都应该是 非阻塞的。如果文件描述符 是阻塞的, 那么读或写操作将会因为没有后续的事件而一直处于阻塞状态 (饥渴状态)。

-> Notice: 对非阻塞 socket 而言, EAGAIN 不是一种错误。在 VxWorks 和 Windows 上, EAGAIN 的名字叫做EWOULDBLOCK。

## 4.18 select、poll、epoll 比较

epoll 跟 select 都能提供多路 I/O 复用的解决方案。在现在的 Linux 内核里有都能够支持, 其中 epoll 是 Linux 所特有, 而 select 则应该是 POSIX 所规定, 一般操作系统均有实现

### 4.18.1 select

select 本质上是通过设置或者检查存放 fd 标志位的数据结构来进行下一步处理。这样所带来的缺点是:

- 单个进程可监视的 fd 数量被限制, 即能监听端口的大小有限。

一般来说这个数目和系统内存关系很大, 具体数目可以 `cat /proc/sys/fs/file-max` 察看。32 位机默认是 1024 个。64 位机默认是 2048。

- 对 socket 进行扫描时是线性扫描, 即采用轮询的方法, 效率较低

当套接字比较多的时候, 每次 `select()` 都要通过遍历 `FD_SETSIZE` 个 `Socket` 来完成调度, 不管哪个 `Socket` 的, 都遍历一遍。这会浪费很多 CPU 时间。如果能给套接字注册某个回调函数, 当他们活跃时, 自动完成相关操作, 那就避免了轮询, 这正是 `epoll` 与 `kqueue` 做的

- 需要维护一个用来存放大量 fd 的数据结构, 这样会使得用户空间和内核空间在传递该结构时复制开销大

### 4.18.2 poll

poll 本质上和 select 没有区别, 它将用户传入的数组拷贝到内核空间, 然后查询每个 fd 对应的设备状态, 如果设备就绪则在设备等待队列中加入一项并继续遍历, 如果遍历完所有 fd 后没

有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历 fd。这个过程经历了多次无谓的遍历。

它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有一个缺点

- 大量的 fd 的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义
- poll 还有一个特点是“水平触发”，如果报告了 fd 后，没有被处理，那么下次 poll 时会再次报告该 fd

### 4.18.3 epoll

epoll 支持水平触发和边缘触发，最大的特点在于边缘触发，它只告诉进程哪些 fd 刚刚变为就绪态，并且只会通知一次。还有一个特点是，epoll 使用“事件”的就绪通知方式，通过 `epoll_ctl` 注册 fd，一旦该 fd 就绪，内核就会采用类似 callback 的回调机制来激活该 fd，`epoll_wait` 便可以收到通知

**epoll 的优点：**

- **没有最大并发连接的限制**，能打开的 FD 的上限远大于 1024（1G 的内存上能监听约 10 万个端口）
- **效率提升**，不是轮询的方式，不会随着 FD 数目的增加效率下降。只有活跃可用的 FD 才会调用 callback 函数；即 Epoll 最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，Epoll 的效率就会远远高于 select 和 poll。
- **内存拷贝**，利用 `mmap()` 文件映射内存加速与内核空间的消息传递；即 epoll 使用 `mmap` 减少复制开销

#### 4.18.4 区别对比

支持一个进程所能打开的最大连接数

表 4.1: 一个进程所能打开的最大连接数

select	单个进程所能打开的最大连接数有 <code>FD_SETSIZE</code> 宏定义，其大小是 32 个整数的大小（在 32 位的机器上，大小就是 32*32，同理 64 位机器上 <code>FD_SETSIZE</code> 为 32*64），当然我们可以对其进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进行进一步的测试。
poll	poll 本质上和 select 没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的
epoll	虽然连接数有上限，但是很大，1G 内存的机器上可以打开 10 万左右的连接，2G 内存的机器可以打开 20 万左右的连接

FD 剧增后带来的 IO 效率问题

表 4.2: FD 剧增后带来的 IO 效率问题

select	因为每次调用时都会对连接进行线性遍历，所以随着 FD 的增加会造成遍历速度慢的“线性下降性能问题”。
poll	因为每次调用时都会对连接进行线性遍历，所以随着 FD 的增加会造成遍历速度慢的“线性下降性能问题”。
epoll	因为 epoll 内核中实现是根据每个 fd 上的 callback 函数来实现的，只有活跃的 socket 才会主动调用 callback，所以在活跃 socket 较少的情况下，使用 epoll 没有前面两者的线性下降的性能问题，但是所有 socket 都很活跃的情况下，可能会有性能问题。

消息传递方式

表 4.3: 消息传递方式

select	内核需要将消息传递到用户空间，都需要内核拷贝动作
poll	内核需要将消息传递到用户空间，都需要内核拷贝动作
epoll	epoll 通过内核和用户空间共享一块内存来实现的。

## 总结

- 表面上看 epoll 的性能最好，但是在连接数少并且连接都十分活跃的情况下，select 和 poll 的性能可能比 epoll 好，毕竟 epoll 的通知机制需要很多函数回调。
- select 低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善

## 4.19 UDP

## 4.20 套接字与流之间的转换

### 4.20.1 转换

`fileno()`

`fdopen()`

### 4.20.2 缓冲

- 全缓冲
- 行缓冲
- 无缓冲

## 4.21 非阻塞 socket

## 4.22 Unix 域套接字编程：本地进程间通信

在本地的进程间通信时，UNIX 域套接字的效率是 TCP 套接字的 2 倍。可以认为是 IPC 的一种手段

## 4.23 HTTP

### 4.23.1 概要

超文本传输协议（HTTP，HyperText Transfer Protocol）是互联网上应用最为广泛的一种网络协议。所有的 WWW 文件都必须遵守这个标准。设计 HTTP 最初的目的是为了提供一种发布和接收 HTML 页面的方法。

### 4.23.2 技术架构

**HTTP 是一个客户端和服务端请求和应答的标准（TCP）。客户端是终端用户，服务器端是网站。**通过使用 Web 浏览器、网络爬虫或者其它的工具，客户端发起一个到服务器上指定端口（默认端口为 80）的 HTTP 请求。（我们称这个客户端）叫用户代理（user agent）。应答的服务器上存储着（一些）资源，比如 HTML 文件和图像。（我们称）这个应答服务器为源服务器（origin server）。在用户代理和源服务器中间可能存在 http 和其他几种网络协议多个中间层，比如代理，网关，或者隧道（tunnels）。尽管 TCP/IP 协议是互联网上最流行的应用，HTTP 协议并没有规定必须使用它和（基于）它支持的层。事实上，**HTTP 可以在任何其他互联网协议上，或者在其他网络上实现。**HTTP 只假定（其下层协议提供）可靠的传输，任何能够提供这种保证的协议都可以被其使用。

通常，由 HTTP 客户端发起一个请求，建立一个到服务器指定端口（默认是 80 端口）的 TCP 连接。HTTP 服务器则在那个端口监听客户端发送过来的请求。一旦收到请求，服务器（向客户端）发回一个状态行，比如“HTTP/1.1 200 OK”，和（响应的）消息，消息的消息体可能是请求的文件、错误消息、或者其它一些信息。

**HTTP 使用 TCP 而不是 UDP 的原因**在于（打开）一个网页必须传送很多数据，而 TCP 协议提供传输控制，按顺序组织数据，和错误纠正。

通过 HTTP 或者 HTTPS 协议请求的资源由统一资源标示符（Uniform Resource Identifiers）（或者，更准确一些，URLs）来标识。

## 4.24 提高性能的措施

### 4.24.1 池

提前预分配系统资源，利用空间换取其运行效率。直接从池中取得所需资源比动态分配资源的速度要快很多，因为分配系统资源的系统调用都是很耗时的。

## 内存池

通常用于 socket 的接受缓存和发送缓存

## 进程池与线程池

处理并发减少切换上下文与动态分配消耗时间的一种手段。

## 连接池

用于服务器或服务机群的内部永久链接。

### 4.24.2 数据拷贝

高性能服务器应该避免不必要的复制，尤其是当数据复制发生在用户代码和内核之间的时刻。如果内核可以直接处理从 socket 或者文件读入的数据，则应用程序就没必要将这些数据从内核缓存区复制到应用缓存区中。

如，ftp 服务器就无需把目标文件的内容完整地读入到应用程序缓冲区中并调用 send 函数来发送，而是可以使用“零拷贝”函数 sendfile 来直接发送给客户端。

### 4.24.3 锁与上下文切换

并发程序必须考虑上下文切换的问题，即进程切换或线程切换导致的系统开销。即使是 I/O 密集型服务器，也不应该食用过多的工作线程 (进程)，否则线程间的切换将占用大量的 CPU 时间，服务器的服务时间的比重就不足了。

除此，对共享资源的互斥访问的加锁保护，通常导致服务器效率低下。所以如果有避免使用锁的方法时就尽量避免使用锁。





# 第五章 负载均衡

## 5.1 负载均衡

### 5.1.1 概述

负载均衡（Load balancing）在不同的领域有不同的概念。其基本概念是为了减轻某个或某些实体的负载，将任务通过某种策略分配到多个实体上去，实现负载在不同实体间的平衡。

负载均衡是由多台服务器以对称的方式组成一个服务器集合，每台服务器都具有等价的地位，都可以单独对外提供服务而无须其他服务器的辅助。

通过某种负载分担技术，将外部发送来的请求均匀分配到对称结构中的某一台服务器上，而接收到请求的服务器独立地回应客户的请求。均衡负载能够平均分配客户请求到服务器列阵，籍此提供快速获取重要数据，解决大量并发访问服务问题。

这种群集技术可以用最少的投资获得接近于大型主机的性能

**Example:** BOSS 一次给了小明好多项任务，小明发现怎么安排时间也做不完，于是乎他盯上了在旁边偷偷看电影的小强，小强突然觉得背后有一股凉气，一回头小明一脸坏笑看着他，“这几个任务交给你，晚上请你吃饭，要不然…嘿嘿嘿”，小强虽然不情愿，但是在小明的请求（要挟）下，只能服从。第二天，小明顺利的完成了任务，给小强买了袋辣条。

在计算机上负载均衡也类似如此，我们的大 BOSS 客户端将请求发送至服务器，然而一台服务器是无法承受很高的并发量的，我们就会将请求转发到其他服务器，当然真正的负载均衡架构并不是由一台 server 转发的另一台 server，而在客户端与服务器端中间加入了一个负责分配请求的负载均衡硬件（软件）。

### 5.1.2 设计思路

一台普通服务器的处理能力只能达到每秒几万个到几十万个请求，无法在一秒钟内处理上百万个甚至更多的请求。但若能将多台这样的服务器组成一个系统，并通过软件技术将所有请求平均分配给所有服务器，那么这个系统就完全拥有每秒钟处理几百万个甚至更多请求的能力。这就是负载均衡最初的基本设计思想。

### 5.1.3 Nginx

### 5.1.4 LVS

<http://www.cnblogs.com/gtarcoder/p/6012117.html>

## 5.2 群集技术

### 5.2.1 集群系统

集群系统是一种由互相连接的计算机组成的并行或分布式系统，可以作为单独、统一的计算资源来使用

### 5.2.2 服务器集群

### 5.2.3 RPC

RPC (Remote Procedure Call Protocol)，远程过程调用协议，一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。它是一项广泛用于支持分布式应用程序（不同组件分布在不同计算机上的应用程序）的技术

### 5.2.4 分布式系统

## 第六章 常见服务器模型

### 6.1 阻塞 IO 服务器模型之单线程服务器模型

单线程服务器模型是最简单的一个服务器模型，几乎我们所有程序员在刚开始接触网络编程（不管是 B/S 结构还是 C/S 结构）都是从这个简单的模型开始。这种模型只提供同时一个客户端访问，多个客户端访问必须要等到前一个客户端访问结束，一个一个排队，即提供一问一答服务。

特点 ->

- 不适合执行时间较长的服务
- 无法充分利用多核 CPU

图6.1展示了单线程阻塞服务器是怎样响应客户端的访问。首先，服务器必须初始化一个 `Serversocket` 实例，绑定某个端口号，并使之监听客户端的访问，以此提供一种服务。接着客户端 1 远程调用服务器的这个服务，服务器接受到请求后，对其进行处理，并返回信息给客户端 1，整个过程都是在一个线程里面完成的。最后，就算客户端 2 在服务器处理完客户端 1 之前就进行请求访问，也要等服务器对客户端 1 响应完后，才会对客户端 2 进行响应处理。

我们注意到，大部分的 `socket` 操作都是阻塞的，所谓阻塞是指调用后不马上返回调用结果，而让当前线程一直阻塞，只有当该调用获得结果或者超时时才会返回。而我们的图 2-6-1-1 也有很多个节点都是阻塞的，例如，服务器阻塞监听客户端，直到有客户端访问才返回一个 `socket`；对于客户端，建立一个 `socket` 连接后，也进行阻塞，直到服务器响应。几乎所有的 IO 操作都会产生阻塞，在网络编程中体现在 `socket` 的通信。这种阻塞给网络编程带来了一个问题，如图中黑块 1 和黑块 2 部分，当服务器处理完进行 IO 操作（这里的 IO 操作其实是给客户端发送消息）时，服务器必须要等到客户端成功接收才能继续往下处理另外一个客户端的请求，在此期间，线程将无法执行任何运算跟响应任何客户端请求。而如果这过程永远发送不到客户端，服务器就一直阻塞在那里了，不会接着处理任何事情。

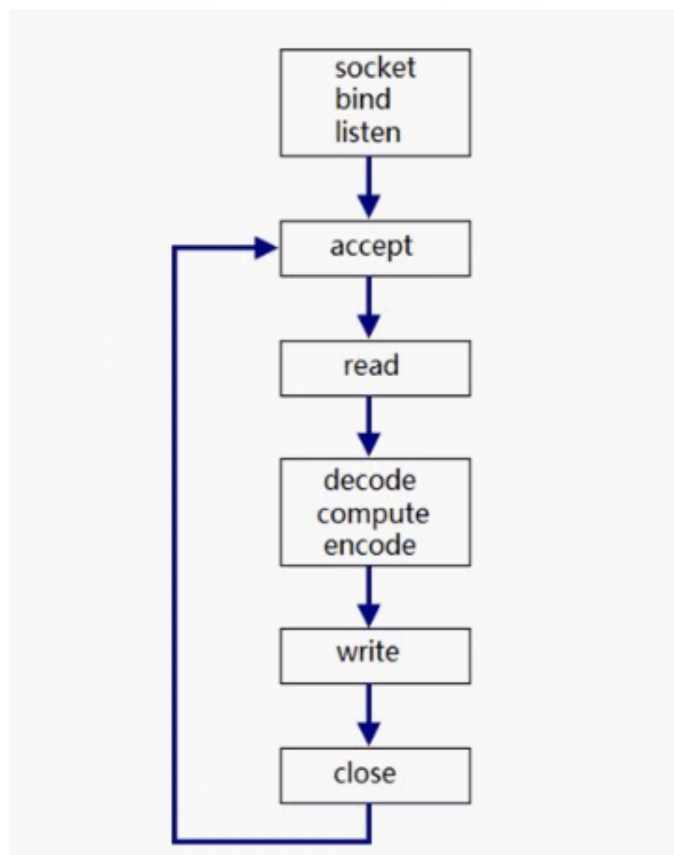
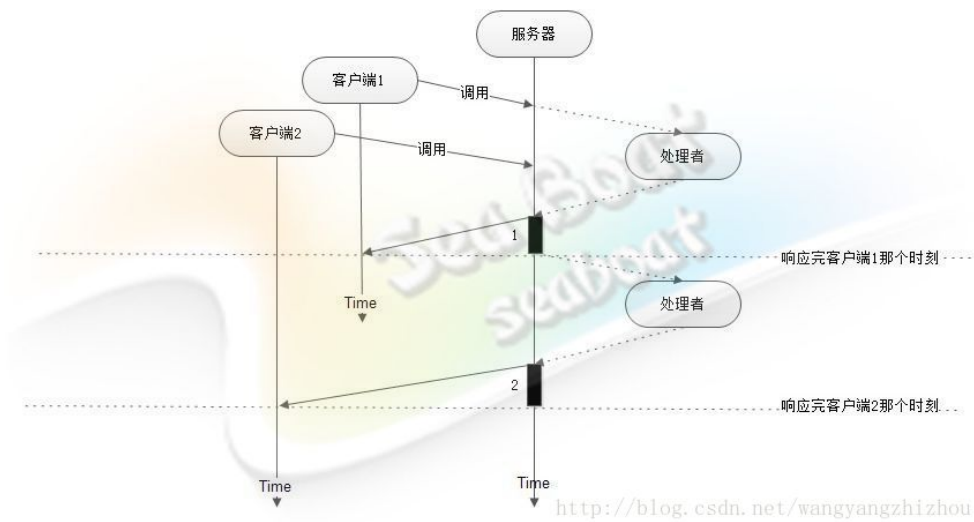


图 6.1: 单线程阻塞服务器模型

这种模型的特点是：最简单的服务器模型，整个运行过程都只有一个线程，只能支持同时处理一个客户端的请求，如果多个客户端访问必须排队等待，服务器系统资源消耗较小，但并发能力低，如果遇到 IO 操作出现错误异常将导致服务器停止运行，容错能力差。一般这种模型一般用在访问跟并发量少，请求是短暂的、无状态的，对响应时间要求不高，处理逻辑较复杂的场合。

## 6.2 并发式模式

架构如图6.2.



图 6.2: conCurrent 服务器模型

特点 ->

- one connection per process/thread
- 适合执行时间较长的服务

## 6.3 pre-Fork or pre-Threaded

架构如图6.3.

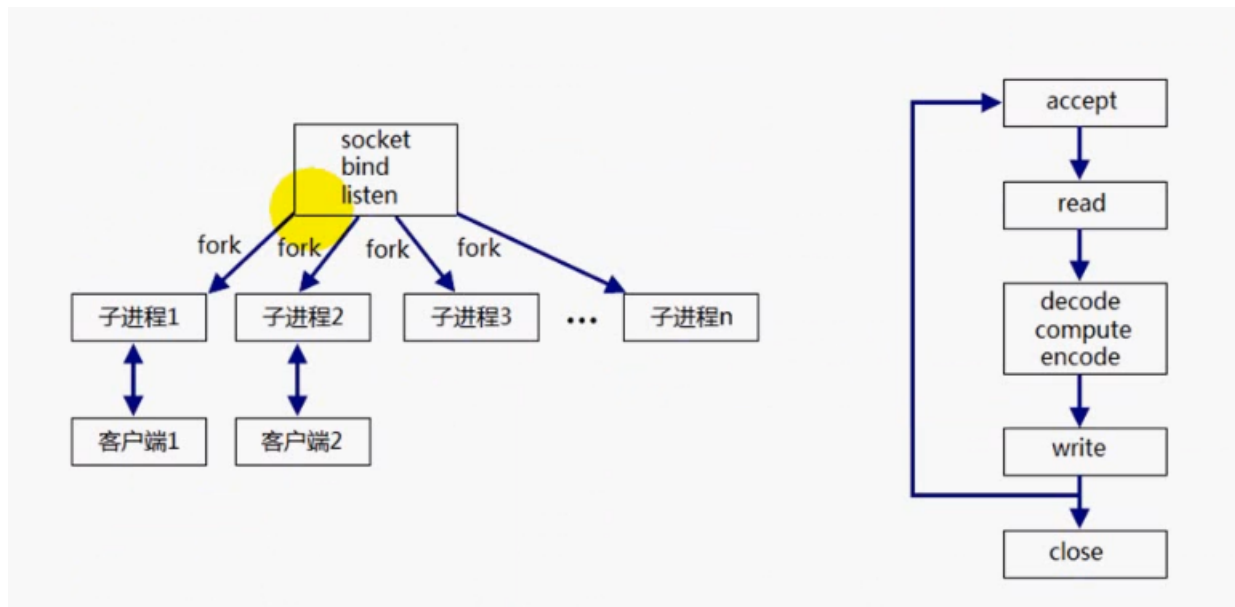


图 6.3: preFork 服务器模型

特点 ->

- 预分配定量进程或线程，省去进程创建开销
- 存在惊群现象 (unp27 章)

## 6.4 ReActor 模式

Reactor 模式首先是事件驱动的，有一个或多个并发输入源，有一个 Service Handler，有多个 Request Handlers；这个 Service Handler 会同步的将输入的请求（Event）多路复用的分发给相应的 Request Handler

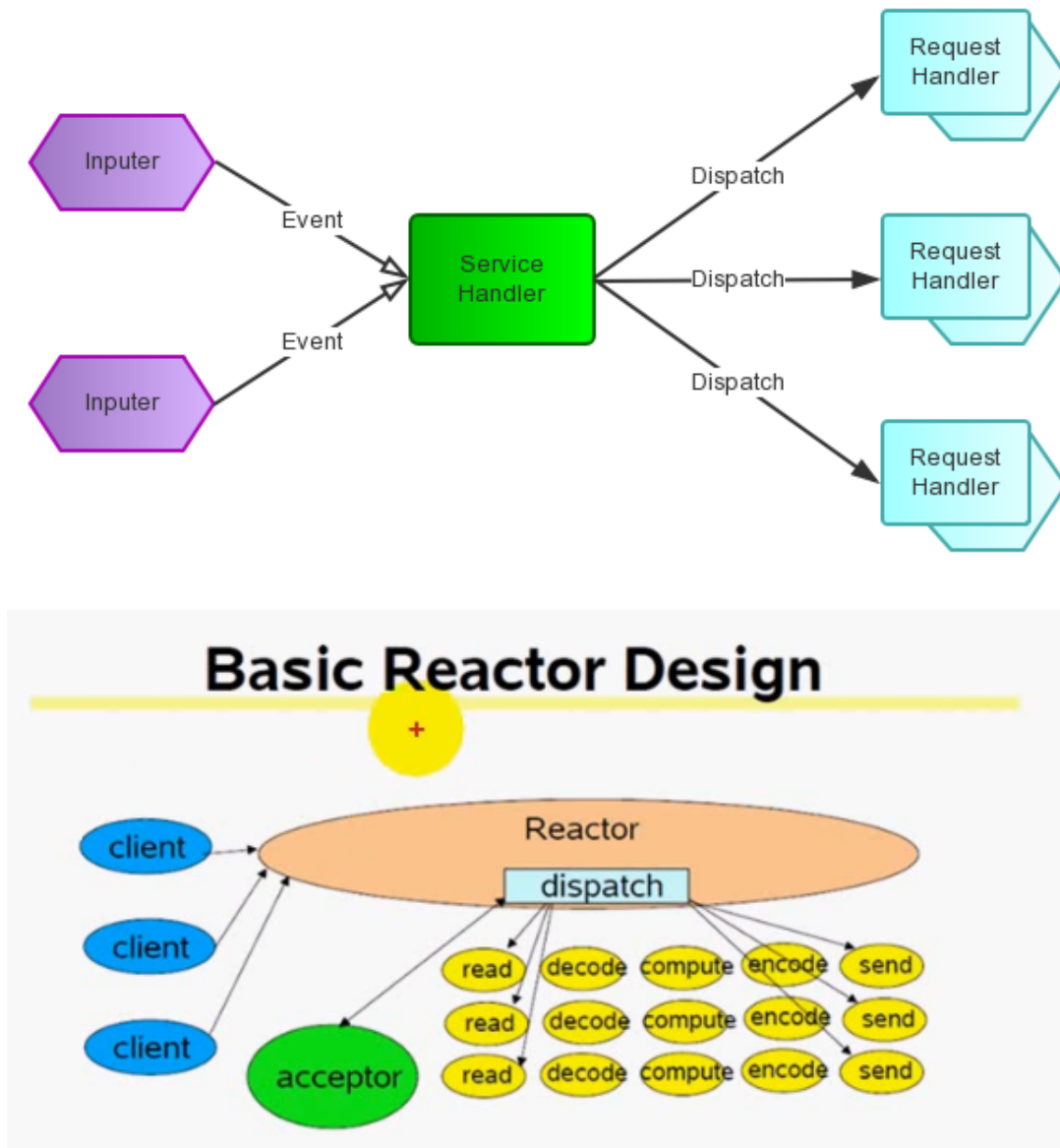


图 6.4: reactor 架构

从结构上，这有点类似生产者消费者模式，即有一个或多个生产者将事件放入一个 Queue 中，而一个或多个消费者主动的从这个 Queue 中 Poll 事件来处理；而 Reactor 模式则并没有 Queue 来做缓冲，每当一个 Event 输入到 Service Handler 之后，该 Service Handler 会主动的根据不同的 Event 类型将其分发给对应的 Request Handler 来处理。



### 6.4.1 例子学习 ReActor

以一个餐饮为例，每一个人来就餐就是一个事件，他会先看一下菜单，然后点餐。就像一个网站会有很多的请求，要求服务器做一些事情。处理这些就餐事件的就需要我们的服务人员了。

在多线程处理的方式会是这样的：一个人来就餐，一个服务员去服务，然后客人会看菜单，点菜。服务员将菜单给后厨。二个人来就餐，二个服务员去服务……五个人来就餐，五个服务员去服务……

这个就是多线程的处理方式，一个事件到来，就会有一个线程服务。很显然这种方式在人少的情况下会有很好的用户体验，每个客人都感觉自己是 VIP，专人服务的。如果餐厅一直这样同一时间最多来 5 个客人，这家餐厅是可以很好的服务下去的。

来了一个好消息，因为这家店的服务好，吃饭的人多了起来。同一时间会来 10 个客人，老板很开心，但是只有 5 个服务员，这样就不能一对一服务了，有些客人就要没有人管了。老板就又请了 5 个服务员，现在好了，又能每个人都受 VIP 待遇了。

越来越多的人对这家餐厅满意，客源又多了，同时来吃饭的人到了 20 人，老板高兴不起来了，再请服务员吧，占地方不说，还要开工钱，再请人就攒不到钱了。怎么办呢？老板想了想，10 个服务员对付 20 个客人也是能对付过来的，服务员勤快点就好了，伺候完一个客人马上伺候另外一个，还是来得及的。综合考虑了一下，老板决定就使用 10 个服务人员的线程池啦

但是这样有一个比较严重的缺点就是，如果正在接受服务员服务的客人点菜很慢，其他的客人可能就要等好长时间了。有些火爆脾气的客人可能就等不了走人了。

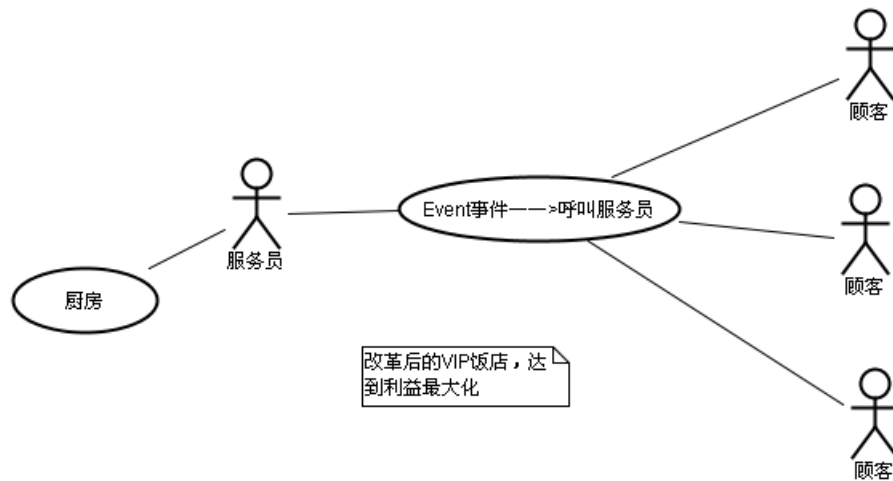
**Reactor 如何处理这个问题呢：**

老板后来发现，客人点菜比较慢，大部服务员都在等着客人点菜，其实干的活不是太多。老板能当老板当然有点不一样的地方，终于发现了一个新的方法，那就是：当客人点菜的时候，服务员就可以去招呼其他客人了，等客人点好了菜，直接招呼一声“服务员”，马上就有个服务员过去服务。嘿嘿，然后在老板有了这个新的方法之后，就进行了一次裁员，只留了一个服务员！这就是用单个线程来做多线程的事。

实际的餐馆都是用的 Reactor 模式在服务。一些设计的模型其实都是从生活中来的。

Reactor 模式主要是提高系统的吞吐量，在有限的资源下处理更多的事情。

在单核的机上，多线程并不能提高系统的性能，除非在有一些阻塞的情况发生。否则线程切换的开销会使处理的速度变慢。就像你一个人做两件事情，1、削一个苹果。2、切一个西瓜。那你可以一件一件的做，我想你也会一件一件的做。如果这个时候你使用多线程，一会儿削苹果，一会切西瓜，可以相像究竟是哪个速度快。这也就是说为什么在单核机上多线程来处理可能会更慢。



但当有阻碍操作发生时，多线程的优势才会显示出来，现在你有另外两件事情去做，1、削一个苹果。2、烧一壶开水。我想没有人会去做完一件再做另一件，你肯定会一边烧水，一边就把苹果削了。

特点 ->

- 并发处理多个请求，实际是在一个线程中完成。无法重复利用多核 CPU
- 不适合执行时间比较长的服务，所以为了让客户感觉是在“并发”处理而不是“循环”处理，每个请求必须在相对较短时间内执行

## 6.4.2 reActor + thread per request

过渡模型

## 6.4.3 参考

<http://daimojingdeyu.iteye.com/blog/828696>

<http://www.blogjava.net/DLevin/archive/2015/09/02/427045.html>

## 6.5 reactor + thread pool

架构如图6.5.

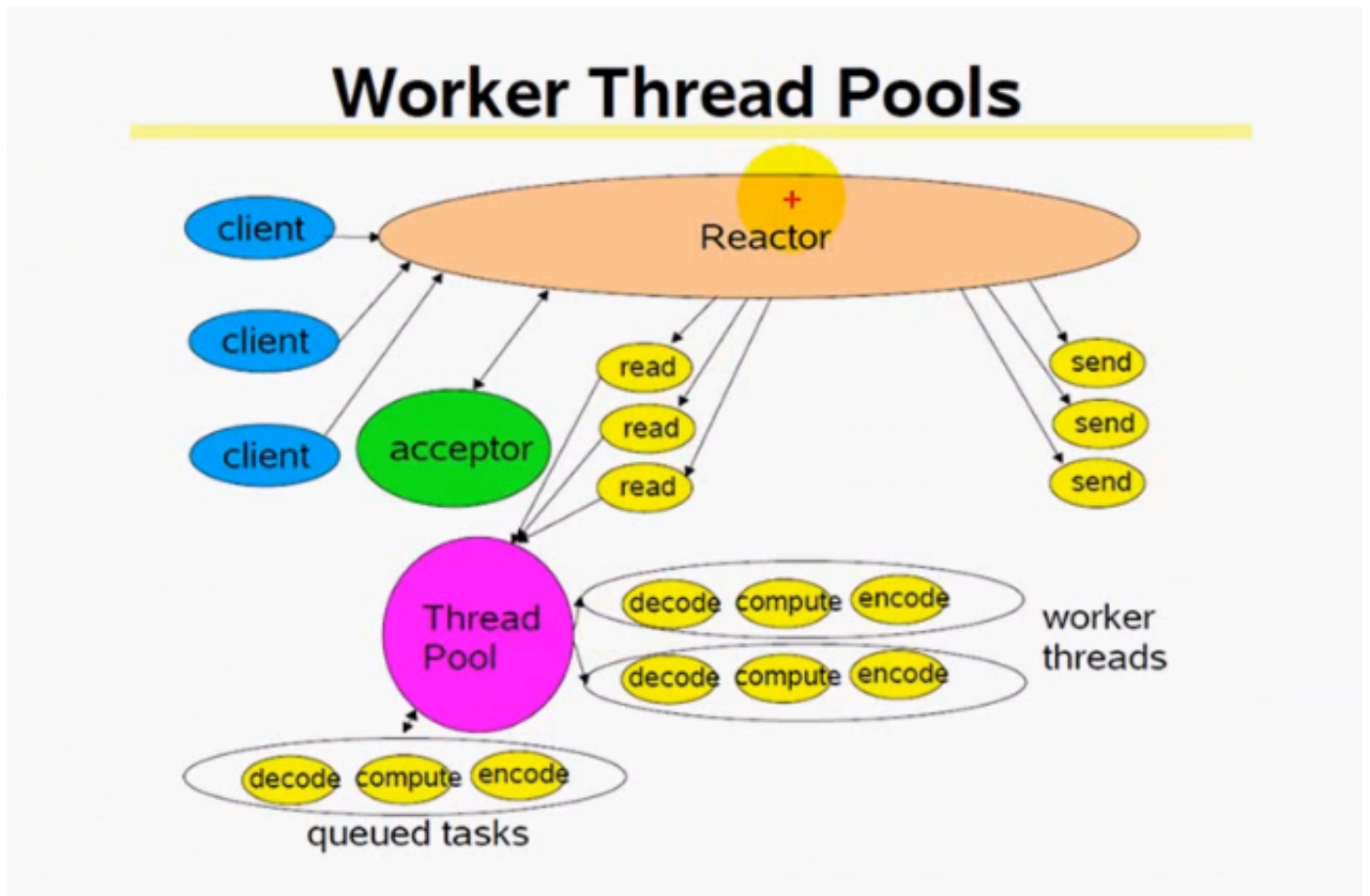


图 6.5: preFork 服务器模型

### 特点

- 能够处理密集型任务

## 6.6 multiple reactors

架构如图6.6.

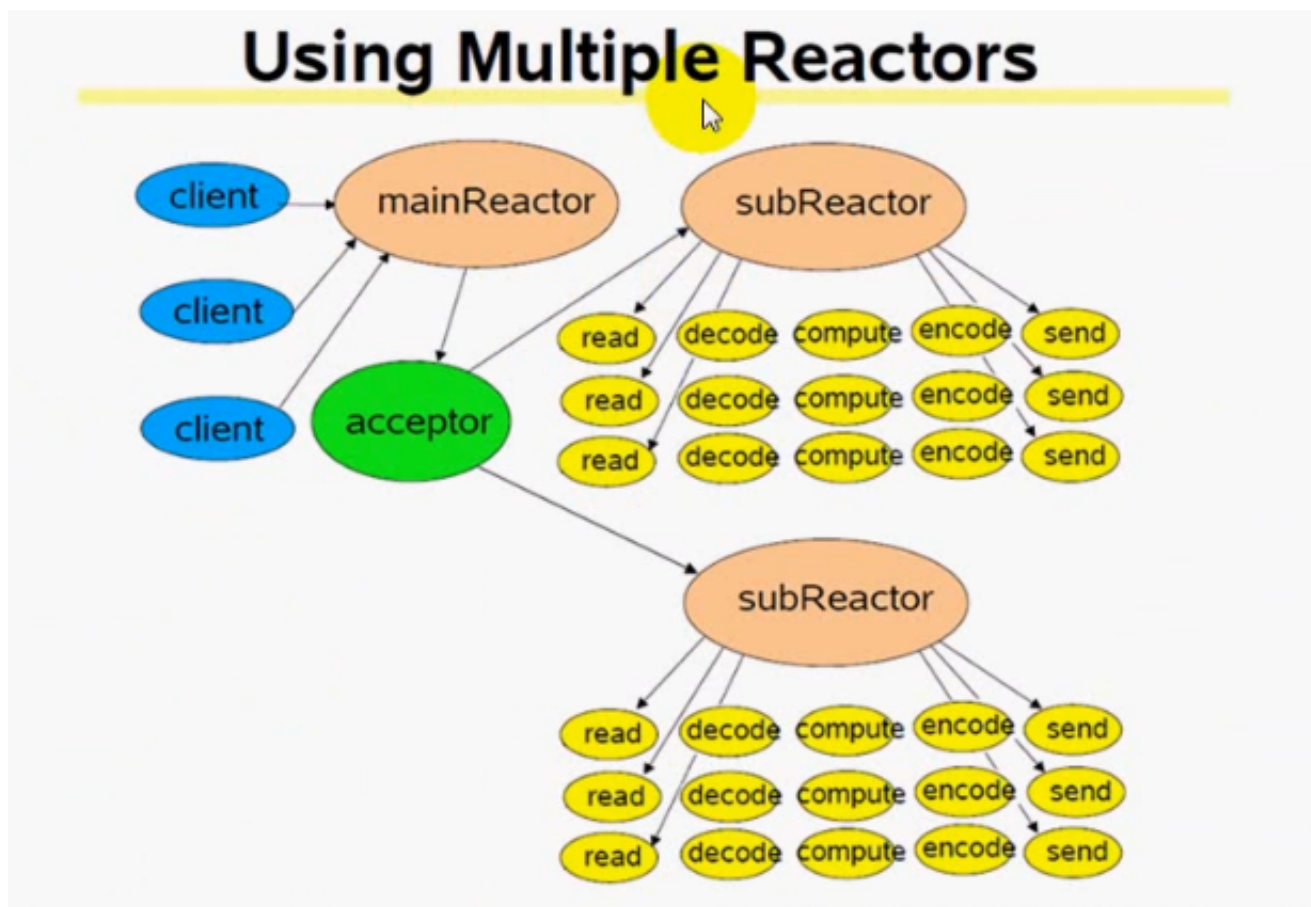


图 6.6: preFork 服务器模型

### 特点

- 能够适应更大的并发
- 能够处理突发大量 I/O 请求
- 具有负载均衡的功能

## 6.7 multiple reactors + thread pool

架构如图6.7.

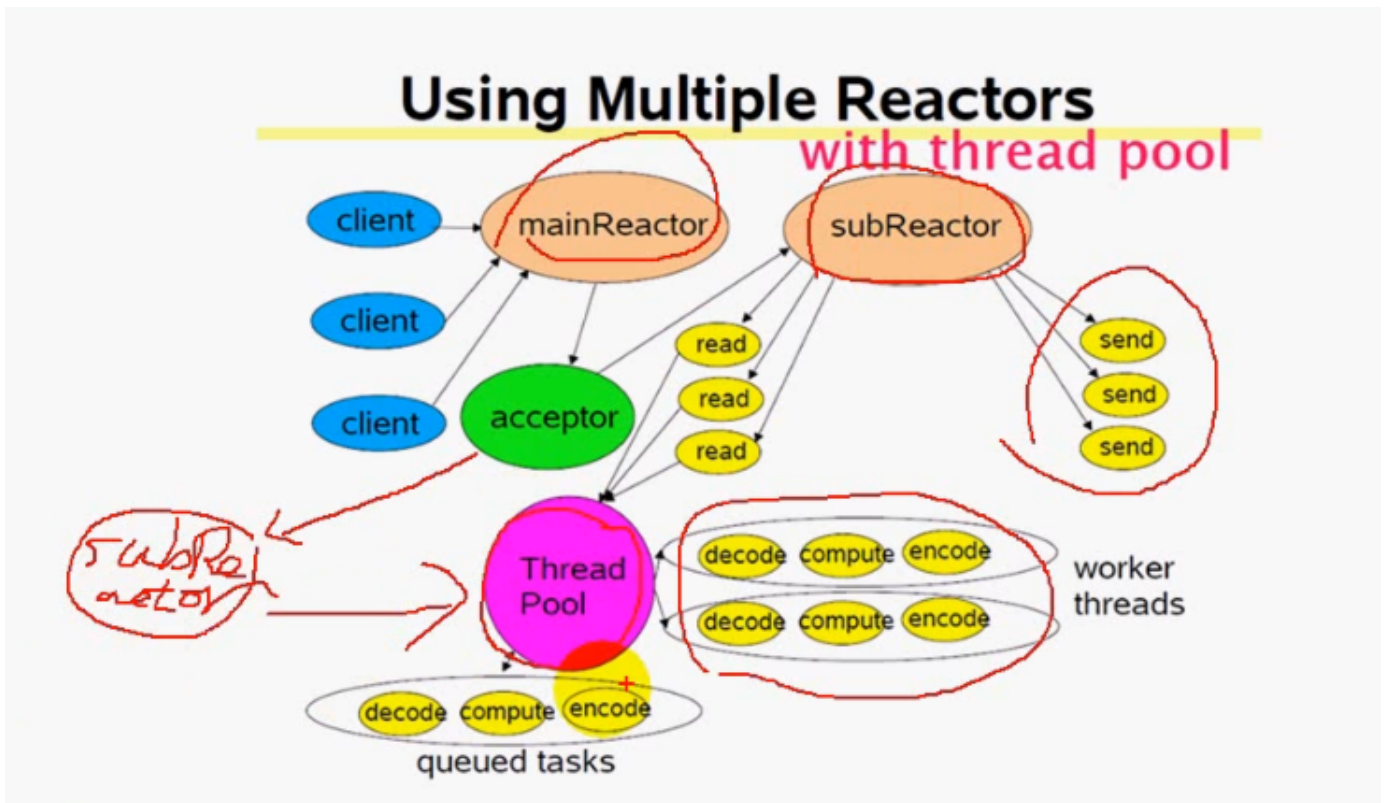


图 6.7: preFork 服务器模型

### 特点

- 能够适应更大的并发
- 更好的处理突发事件
- 更好的发挥多核作用
- 具有负载均衡的功能

## 6.8 ProActor

异步架构如图6.8.

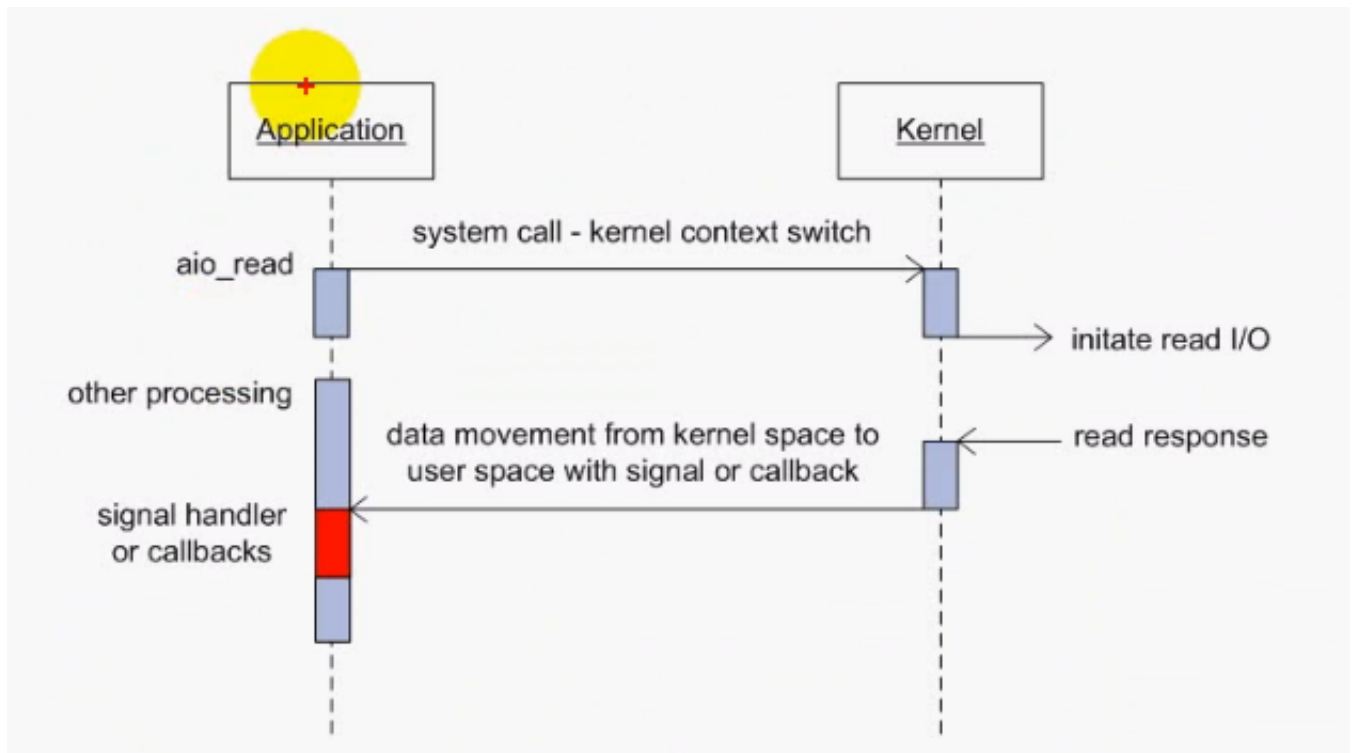


图 6.8: PreActor 服务器模型

## 6.9 Reactor with Proactor

以读取操作为例

### 6.9.1 reactor

Reactor 模式用于同步I/O

1. 应用程序注册读就需事件和相关联的事件处理器
2. 事件分离器等待事件的发生
3. 当发生读就需事件的时候，事件分离器调用第一步注册的事件处理器
4. 事件处理器首先执行实际的读取操作，然后根据读取到的内容进行进一步的处理

### 6.9.2 proactor

Proactor 运用于异步I/O 操作

1. 应用程序初始化一个异步读取操作，然后注册相应的事件处理器，此时事件处理器不关注读取就绪事件，而是关注读取完成事件，这是区别于Reactor 的关键。
2. 事件分离器等待读取操作完成事件
3. 在事件分离器等待读取操作完成的时候，操作系统调用内核线程完成读取操作，并将读取的内容放入用户传递过来的缓存区中。这也是区别于Reactor 的一点，Proactor 中，应用程序需要传递缓存区。
4. 事件分离器捕获到读取完成事件后，激活应用程序注册的事件处理器，事件处理器直接从缓存区读取数据，而不需要进行实际的读取操作。

### 6.9.3 区别

Proactor 中写入操作和读取操作，只不过感兴趣的事件是写入完成事件

Reactor和Proactor 模式的主要区别就是真正的读取和写入操作是有谁来完成的，Reactor 中需要应用程序自己读取或者写入数据，而Proactor 模式中，应用程序不需要进行实际的读写过程，它只需要从缓存区读取或者写入即可，操作系统会读取缓存区或者写入缓存区到真正的IO 设备.

## 6.10 半同步半异步模型

### 6.10.1 同步

I/O 事件为就绪状态时通知应用程序，数据的搬放由应用程序本身来完成，涉及应用缓存与内核缓存的复制效率问题。

### 6.10.2 异步

I/O 事件为完成状态时通知应用程序，数据的搬放由内核来完成，通知时已将数据从缓存中读到应用缓存，或从应用写入内核缓存中，I/O 效率更高。

### 6.10.3 半同步半异步

<http://blog.csdn.net/zhuziyu1157817544/article/details/72486157>

异步负责 I/O 事件处理，同步负责处理用户逻辑

#### 半同步半反应堆模式

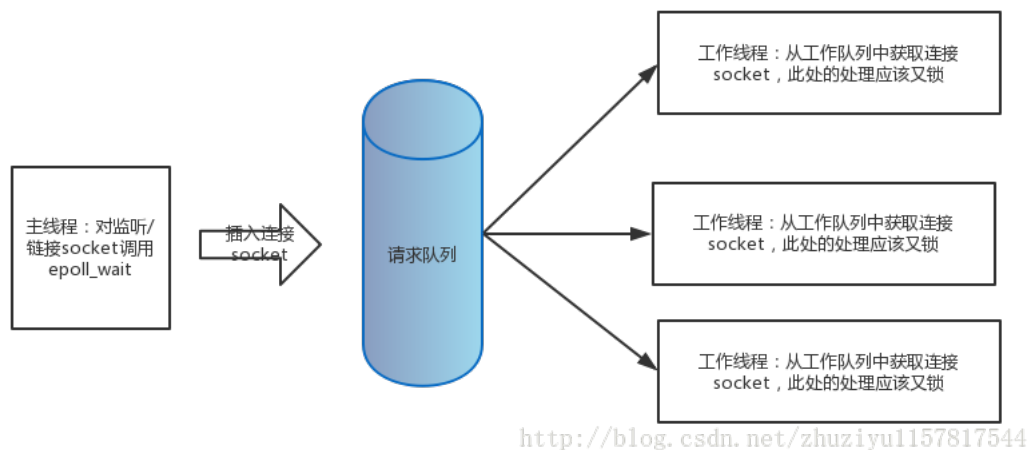


图 6.9: 半同步半异步

- 异步线程只有一个，由主线程充当，负责监听所有socket 事件。
  1. 如果监听socket 上有可读事件发生，指的是新的链接请求到来，那么异步线程接受它，往epoll 内核事件表中注册该socket 上的读写事件。



2. 如果连接socket 上有读写事件发生，要么是新的客户请求，要么是有数据要发送给客户端，主线程就把改连接 socket 插入请求队列。

- 所有的工作线程睡眠在请求队列上，有任务来的时候，空闲线程竞争，获取任务接管权。此后，该线程管理这个socket 所有的 I/O 任务，直到客户关闭连接。

缺点 ->

- 主线程和工作线程共享一个请求队列，因此一旦队列中的任务有变更，就需要加锁保护
- 每一个工作线程在同一时间只能处理一个客户请求，对于客户数良多，但是工作任务少的情况下，队友很多任务堆积，客户的响应速度越来越慢

例子 ->

<http://blog.csdn.net/u013074465/article/details/46357219>

more Efficient

改进方面：让一个工作线程能够处理很多的客户请求。即对每一个工作线程，对于各自连接的socket 们调用epoll\_wait。

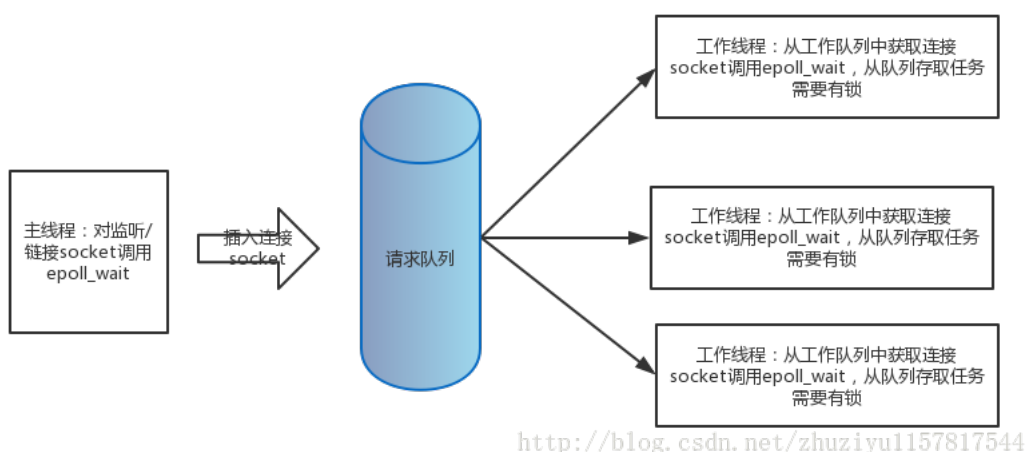


图 6.10: 半同步半异步-2

## 6.11 领导者追随者模式

放风的山贼（领导者）与睡觉（追随者）、抢劫（old 领导者）的山贼

<http://blog.csdn.net/jinchaoh/article/details/50427733>

## 第七章 使用多台机器并行处理数据

### 7.1 集群服务器

同一个业务，部署在多个服务器上

小饭店原来只有一个厨师，切菜洗菜备料炒菜全干。后来客人多了，厨房一个厨师忙不过来，又请了个厨师，两个厨师都能炒一样的菜，这两个厨师的关系是集群。为了让厨师专心炒菜，把菜做到极致，又请了个配菜师负责切菜，备菜，备料，厨师和配菜师的关系是分布式，一个配菜师也忙不过来了，又请了个配菜师，两个配菜师关系是集群

### 7.2 分布式服务器

一个业务分拆多个子业务，部署在不同的服务器上，具体的平台有 Hadoop. 分布式强调机器间的协作，其重点是任务可拆分，如某个任务需要一个机器运行 10 个小时，将该任务用 10 台机器的分布式跑，可能 2 个小时就跑完了。（子任务之间有依赖关系）。



# 第八章 网络库

## 8.1 muduo

### 8.1.1 bind/function

适配器函数

```
class Foo{
public:
    void memberFunc(double d, int i, int j)
    {
        cout<<d<<endl;
        cout<<i<<endl;
        cout<<j<<endl;
    }
};

int main()
{
    Foo foo;
    boost::function<void (int)>fp = boost::bind(&Foo::memberFunc,&foo,0.5,_1,10);
    fp(100);
    return 0;
}
```

### 8.1.2 面向对象的编程

虚函数类型多态类，面向继承

```
// EventLoop 调用TcpServer, TcpServer 利用虚函数接口回调具体的子类方法，即利用父指针调用子对象
// 的虚函数实现功能的更新
class TcpServer
{
```

```

public:
    TcpServer(){}
    virtual ~TcpServer(){}

    virtual void onConnection()=0;
    virtual void onMessage()=0;
};

class EchoServer: public TcpServer
{
public:
    EchoServer(){}
    virtual ~EchoServer(){}

    virtual void onConnection(){}
    virtual void onMessage(){}
};

```

### 8.1.3 基于对象的编程

利用 bind/function 实现多态，实现参数上的适配。

```

//EventLoop 调用TcpServer, TcpServer 利用回调机制绑定具体执行函数，所以并不涉及虚函数。
class TcpServer
{
public:
    TcpServer(){}
    ~TcpServer(){}

    typedef function<void()> func;
    void onConnection()
    {
        onConnection_func();
    }
    void onMessage()
    {
        onMessage_func();
    }
    void setConnectionCallback(func func_onConnect)
    {
        onConnection_func = func_onConnect;
    }
    void setMessageCallback(func func_Message)

```

```

    {
        onMessage_func = func_Message;
    }
private:
    func onConnection_func;
    func onMessage_func;

};

class EchoServer
{
public:
    EchoServer()
    {
        server.setConnectionCallback(bind(onConnection));
        server.setMessageCallback(bind(onMessage));
    }

    void onConnection();
    void onMessage();

    TcpServer server;
};

```

#### 8.1.4 具体细节

参考c++\_project -> muduo





# 第九章 Redis with Memcache

## 9.1 NoSQL

Not Only Sql 在大多数 web 应用都将数据保存到关系型数据库中，www 服务器从中读取数据并在浏览器中显示。

但随着数据量的增大，访问的集中，就会出现关系型数据库的负担加重，数据库响应缓慢、网站打开延迟等问题。

通过在内存中缓存数据库的查询结果，减少数据访问次数，以提高动态 web 应用的速度，提高网站架构的并发能力和可扩展性。

传统开发中用的数据库最多的就是 MySQL, 随着数据量上亿级后，它的关系型数据库的读取速度可能就不能满足我们对数据的需求，所以内存式的缓存系统就出现了。

### 9.1.1 解决问题

- 高并发读写 (传统问题)
- 海量数据的高效存储与访问 (亿级数据的访问)
- 高可扩展性和高可用性 (7\*24)

### 9.1.2 四大分类

- 键值存储 (key-value)
- 列存储
- 文档数据库
- 图形数据库

## 9.2 Memcache with Redis

### 9.2.1 Memcache

Memcache 是一个高性能的分布式内存对象缓存系统，用于动态 web 应用以减少数据库负载。它通过在内存中缓存数据和对象来减少读取数据库的次数，从而提高动态、数据库驱动网站的速度。

Memcache 基于一个存储键/值对的 hashmap。其守护进程 (daemon) 是用 C 写的，但是客户端可以用任何语言来编写，并通过 memcache 协议与守护进程通信。

内部的数据存储，使用基于 Slab 的内存管理机制，有利于减少内存碎片和频繁的分配销毁内存所带来的开销。各个 Slab 按需动态分配一个 page 的内存。

Memcache 是纯 KV 缓存，分布式实现由客户端实现。

Memcache 不能做数据的持久化工作，只是缓存。

### 9.2.2 Redis

Redis 有着更复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。

Redis 的数据类型都是基于基本数据结构的，同时对程序员透明，无需进行额外的抽象。

Redis 运行在内存中但是可以持久化到磁盘，所以对不同数据集进行高速读写时需要权衡内存，应用数据量不能大于硬件内存。

在内存数据库方面的另一个优点是：相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样 Redis 可以做很多内部复杂性很强的事情。同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

Redis 内部的数据结构最终也会落实到 key-value 对应的形式，不过从暴露给用户的数据结构来看，要比 Memcache 丰富，除了标准的键值对，Redis 还支持 List,Set,Hashes,Sorted Set 等数据结构

Redis 的分布式由服务器端实现，通过服务器配置来实现分布式。

Redis 支持两种数据持久化的方案：1 是 snapshot 快照 (隔段时间将完整的数据 Dump 下来存储在文件中)，2 是 AOF 增量 log 方式 (记录对数据的修改操作)。

Redis 可以以 master-Slave 的方式配置服务器，Slave 节点对数据进行 replica 备份，Slave 节

点也可以充当 read-Only 的节点分担数据读取的工作。

## 9.3 Memcache

介绍: <https://www.cnblogs.com/szlbn/p/5588549.html>

C++withMem:<http://www.jsjtt.com/xitongyingyong/linux/58.html>

<http://www.cnblogs.com/rooney/archive/2012/07/04/2577018.html>



## 第十章 编译

### 10.1 常规

### 10.2 Makefile

### 10.3 CMake

<https://www.ibm.com/developerworks/cn/linux/l-cn-cmake/>

#### 10.3.1 流程

在 linux 平台下使用 CMake 生成 Makefile 并编译的流程如下:

1. 编写 CmakeLists.txt。
2. 执行命令“cmake PATH”或者“ccmake PATH”生成 Makefile (PATH 是 CMakeLists.txt 所在的目录)。
3. 使用 make 命令进行编译。

#### 10.3.2 单目录结构

0. 工程目录 `mkdir Project`

1. 添加源文件

- `cd ./Project`
- `vi Main.cc ...`

## 2. 编写 CMakeLists.txt ->

```
#项目的名称
PROJECT(Main)

#限定CMake版本
CMAKE_MINIMUM_REQUIRED(VERSION 2.6)

#将当前目录中的源文件名称赋值给变量DIR_SRCS
AUX_SOURCE_DIRECTORY(. DIR_SRCS)

#指示需要编译成的可执行文件的名称 如 -o
ADD_EXECUTABLE(Main ${DIR_SRCS})
```

## 3. 编译

1. cmake . 生成 makefile
2. make 编译生成可执行文件

### 10.3.3 多目录结构

#### 代码结构 ->

```
./step2
|
+---Main.cc
|
+---src
|
|   +---Test1.h
|   |
|   +---Test1.cc
```

#### 编写项目主目录 CMakeLists.txt

```
#项目的名称
PROJECT(Main)

#限定CMake版本
CMAKE_MINIMUM_REQUIRED(VERSION 2.6)

#指明本项目包含的一个子目录
ADD_SUBDIRECTORY(src)
```

```
#将当前目录中的源文件名称赋值给变量DIR_SRCS
AUX_SOURCE_DIRECTORY(. DIR_SRCS)

#指示需要编译成的可执行文件的名称 如 -o
ADD_EXECUTABLE(Main ${DIR_SRCS})

#指明可执行文件 需要链接的库
TARGET_LINK_LIBRARIES(Main Test)
```

**编写次目录 CMakeLists.txt** 如果次目录的文件需要使用，那么就必须如基本的 CMake 流程了。

```
#将当前目录中的源文件名称赋值给变量DIR_TEST_SRCS
AUX_SOURCE_DIRECTORY(. DIR_TEST_SRCS)

#使用该命令将 当前目录中的源文件 编译成共享库
ADD_LIBRARY(Test ${DIR_TEST_SRCS})
```

## 编译

1. cmake .
2. make

### 10.3.4 使用其他链接库





# 第十一章 调试

## 11.1 core 文件

<http://blog.csdn.net/zzwdkxx/article/details/73788226>

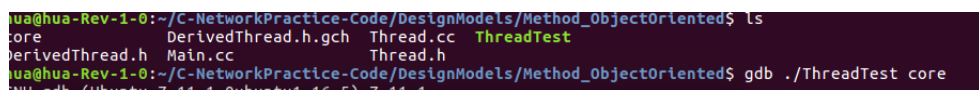
### 11.1.1 什么是 Core Dump

Core 的意思是内存, Dump 的意思是扔出来, 堆出来. 开发和使用 Unix 程序时, 有时程序莫名其妙的 down 了, 却没有任何的提示 (有时候会提示 **core dumped**). 这时候可以查看一下有没有形如 core. 进程号的文件生成, 这个文件便是操作系统把程序 down 掉时的内存内容扔出来生成的, 它可以做为调试程序的参考.

core dump 又叫核心转储, 当程序运行过程中发生异常, 程序异常退出时, 由操作系统把程序当前的内存状况存储在一个 core 文件中, 叫 core dump.

### 11.1.2 如何使用 core 文件

gdb 运行程序 core 文件



```
hua@hua-Rev-1-0:~/C-NetworkPractice-Code/DesignModels/Method_ObjectOriented$ ls
core                               DerivedThread.h.gch  Thread.cc  ThreadTest
DerivedThread.h  Main.cc              Thread.h
hua@hua-Rev-1-0:~/C-NetworkPractice-Code/DesignModels/Method_ObjectOriented$ gdb ./ThreadTest core
GNU gdb (Ubuntu 7.11-1ubuntu1~16.5) 7.11.1
```

图 11.1: 如何使用 core 文件

### 11.1.3 为什么没有 core 文件生成

有时候程序 down 了, 但是 core 文件却没有生成. core 文件的生成跟你当前系统的环境设置有关系, 可以用下面的语句设置一下, 然后再运行程序便生成 core 文件.

```
ulimit -c unlimited
```

【没有找到 core 文件，我们改改 ulimit 的设置，让它产生。1024 是随便取的，要是 core 文件大于 1024 个块，就产生不出来了。）

```
$ ulimit -c 1024 （转者注：使用-c unlimited不限制core文件大小
```

core 文件生成的位置一般于运行程序的路径相同, 文件名一般为 core. 进程号

#### 11.1.4 如何定位到行

在编译的时候开启-g 调试开关就可以了

```
gcc -o main -g a.c
```

```
gdb main /tmp/core-main-10815
```

最终看到的结果如下，好棒。

```
Program terminated with signal 11, Segmentation fault.  
#0 0x080483ba in func (p=0x0) at a.c:55 *p = 0;
```

总结一下，需要定位进程挂在哪一行我们只需要 4 个操作，

1. `ulimit -c unlimited`
2. `echo "/tmp/core-%e-%p" > /proc/sys/kernel/core_pattern`
3. `gcc -o main -g a.c`
4. `gdb main /tmp/core-main-10815`

->Notice: 通常导致段错误的几个直接原因:

1. 引用一个包含非法值的指针
2. 常常由于从系统程序中返回空指针，并未经检查就使用
3. 用删除指针的方式删除对象。

表 11.1: 建立阶段演示

序号	方向	Seq	ACK
1	A -> B	10000	0
2	B -> A	20000	$10000+1 = 10001$
3	A -> B	10001	$20000+1 = 20001$

## 11.2 ACK 与 Seq

### 11.2.1 建立与释放阶段

1. A 向B 发起连接请求，以一个随机数初始化A 的seq, 这里假设为10000，此时ACK=0
2. B 收到A 的连接请求后，也以一个随机数初始化B 的seq, 这里假设为20000，意思是：你的请求我已收到，我这方的数据流就从这个数开始。B 的ACK 是A 的seq 加1，即 $10000+1 = 10001$
3. A 收到B 的回复后，它的seq 是它的上个请求的seq 加1，即 $10000+1 = 10001$ ，意思也是：你的回复我收到了，我这方的数据流就从这个数开始。A 此时的ACK 是B 的seq 加1，即 $20000+1=20001$

### 11.2.2 数据传输阶段

表 11.2: 数据传输阶段演示

序号	方向	Seq	ACK	Size
1	A -> B	40000	70000	1514
2	B -> A	70000	$40000+1514-54 = 41460$	54
3	A -> B	41460	$70000+54-54 = 70000$	1514
4	B -> A	70000	$41460+1514-54 = 42920$	54

1. B 接收到 A 发来的 seq=40000,ack=70000,size=1514 的数据包
2. 于是 B 向 A 也发一个数据包，告诉 B，你的上个包我收到了。B 的 seq 就以它收到的数据包的 ACK 填充，ACK 是它收到的数据包的 SEQ 加上数据包的大小（不包括以太网协议头，IP 头，TCP 头），以证实 B 发过来的数据全收到了。

3. A 在收到 B 发过来的 ack 为 41460 的数据包时，一看到 41460，正好是它的上个数据包的 seq 加上包的大小，就明白，上次发送的数据包已安全到达。于是它再发一个数据包给 B。这个正在发送的数据包的 seq 也以它收到的数据包的 ACK 填充，ACK 就以它收到的数据包的 seq(70000) 加上包的 size(54) 填充，即  $ack=70000+54-54$  (全是头长，没数据项)。

#### Note -

1. 如果对方没有数据过来，则自己的确认号不变，序列号为上次的序列号加上本次应用层数据发送长度。
2. 其实在握手和结束时确认号应该是对方序列号加1，传输数据时则是对方序列号加上对方携带应用层数据的长度。

## 11.3 strace

<http://blog.csdn.net/zhoushenhe2008/article/details/63687169>

- 每一行都是一次系统调用。
- 等号左边是系统调用函数及其参数，右边是返回值。
- 系统首先调用 `execve` 开始一个新的进程，最后调用 `exit_group` 退出进程，完成整个程序的执行过程。strace 首先调用 `fork` 或者 `clone` 函数新建一个子进程，然后在子进程中调用 `exec` 载入需要执行的程序。
- `brk(0) = 0x8803000`，以 0 作为参数的调用 `brk`，返回值是内存管理的起始地址，如果程序调用 `malloc`，内存管理分配将从 `0x8803000` 地址开始。
- `access` 是检查文件是否存在
- 使用 `mmap2` 函数进行匿名内存映射，以此来获取内存空间，其第二参数就是需要获取内存空间的长度，返回值为内存空间的起始地址。

匿名内存映射就是为了不涉及具体的文件名，避免了文件的创建和打开，只能用于具有亲缘关系的进程间通信。

### 11.3.1 指定追踪类型

`-e expr`

指定一个表达式，用来控制如何跟踪

`-e trace=set` 只跟踪指定的系统调用. 例如:`-e trace=open,close,read,write` 表示只跟踪这四个系统调用. 默认的为 `set=all`.

`-e trace=file` 只跟踪有关文件操作的系统调用.

`-e trace=process` 只跟踪有关进程控制的系统调用.

`-e trace=network` 跟踪与网络有关的所有系统调用.

```
hua@Hua:~/_CPP/NetworkPracticing/basic$ strace -e trace=network ./client
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
connect(3, {sa_family=AF_INET, sin_port=htons(8888), sin_addr=inet_addr("127.0.0.1")}, 16) = 0
connected
closed
+++ exited with 0 +++
```

`-e strace=signal` 跟踪所有与系统信号有关的系统调用

`-e trace=ipc` 跟踪所有与进程通讯有关的系统调用

## 11.4 tcpdump

## 11.5 nc

## 11.6 lsof

`lsof`(list open files) 是一个列出当前系统打开文件的工具

## 11.7 netstat

`Netstat` 命令用于显示各种网络相关信息,如网络连接,路由表,接口状态 (Interface Statistics), `masquerade` 连接, 多播成员 (Multicast Memberships) 等等

## 11.8 vmstat

## 11.9 ifstat

`ifstat` 工具是个网络接口监测工具, 比较简单看网络流量

## 11.10 mpstat

mpstat 是 Multiprocessor Statistics 的缩写，是实时系统监控工具。其报告与 CPU 的一些统计信息，这些信息存放在 /proc/stat 文件中。在多 CPUs 系统里，其不但能查看所有 CPU 的平均状况信息，而且能够查看特定 CPU 的信息。mpstat 最大的特点是：可以查看多核心 cpu 中每个计算核心的统计数据；而类似工具 vmstat 只能查看系统整体 cpu 情况。

## 11.11 top

## 11.12 ping

## 11.13 iptables

## 11.14 TTCP

<http://www.feamane.org/comms/testtools/ttcp/ttcp-quickstartguide.html>

## 第十二章 理论

### 12.1 网络的性能

#### 12.1.1 速率

计算机中数据量的单位是用比特 (bit) 来表示的, 它是一个二进制数, 0 或者 1, 网络中的速率是连接在计算机网络上的主机在数字信道上传送数据的速率, 也可以说是数据率或比特率  $\text{bit/s} = \text{bps}(\text{bit per second})$ , 当比特率较高时, 就可以用  $\text{Kb/s}$   $\text{Mbps/s}$   $\text{Gb/s}$

#### 12.1.2 带宽

有两种含义, **第一种**是指某个信号具有的频带宽度, 用赫兹表示 (Hz)。 **第二种**就是我们说的计算机网络中使用的带宽, 是通信线路所能传送数据的能力, 简洁的说是最高数据率

#### 12.1.3 吞吐量

吞吐量 (throughput) 表示单位时间内通过某个网络 (或者信道、接口) 的数据量。吞吐量更经常地用于对现实世界中的网络的一种测量, 以便知道实际上到底有多少数据量能够通过网络。

吞吐量受网络的**带宽或网络的额定速率**的限制。对  $100\text{Mb/s}$  的以太网, 其典型的吞吐量可能只有  $70\text{Mb/s}$ 。

#### 12.1.4 时延

时延 (delay 或 latency) 是指数据 (一个报文或分组, 甚至比特) 从网络 (或链路) 的一端传送到另一端所需的时间。它有时也称为 **延迟或迟延**。

时延通常由以下几个方面组成:

**发送时延 Transmission Delay** 是主机或路由器发送数据帧所需要的时间.. 也就是从发送数据帧的第一个比特算起, 到该帧的最后一个比特发送完毕所需要的时间, 因此发送时延又称**传输时延**, 发送时延的计算公式如下

$$\text{发送时延} = \text{数据帧长度}(b) \div \text{信道带宽}(b/s)$$

由此可见, 对于一定的网络.. 发送时延与发送的帧长成正比, 与信道带宽成反比, 而**并非**是固定不变的..

**传播时延 Propagation Delay** 是电磁波在信道中传播一定的距离需要花费的时间.. 计算公式如下:

$$\text{传播时延} = \text{信道长度}(m) \div \text{电磁波在信道上的传播速率}(m/s)$$

**处理时延** 主机或路由器在收到分组时要处理一定的时间进行处理, 例如分析分组的首部、从分组中提取数据部分、进行差错检验或查找适当的路由等, 这就产生了处理时延.

**排队时延** 分组在经过网络传输时, 要经过许多的路由器, 但分组在进入路由器后要现在输入队列中排队等待处理. 在路由器确定了转发接口后, 还要在输出队列中排队等待转发, 这就产生了排队时延.

$$\text{总时延} = \text{发送时延} + \text{传播时延} + \text{处理时延} + \text{排队时延}$$

### 12.1.5 时延带宽积

时延带宽积 = 传播时延 × 带宽, 链路的时延带宽积 又称为 **以比特为单位的链路长度**

### 12.1.6 往返时间 RTT

往返时间 RTT(Round-Trip Time) 它表示从发送方发送数据开始, 到发送方收到来自接收方的确认总共经历的时间.



### 12.1.7 利用率

有信道利用率和网络利用率两种.. **信道利用率**指出某信道有百分之几的时间是被利用的 (有数据通过), 完全空闲的信道利用率是 0. **网络利用率**则是全网络的信道利用率的加权平均值. 信道利用率并非越高越好, 这就跟高速公路一样.. 车流量大了.. 在某些地方就会出现堵塞, 会增加行车时间.. 也就是网络时延..

所以, 信道或网络利用率过高会产生非常大的时延.

## 12.2 应用层

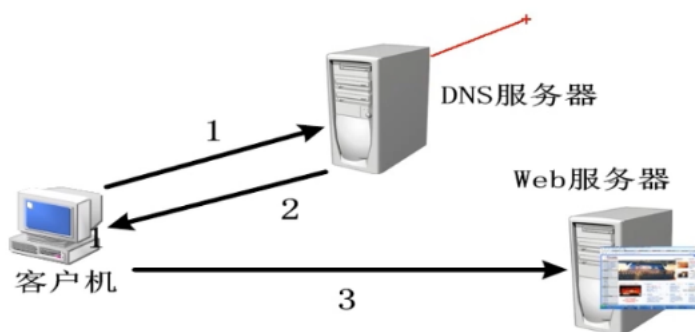
### 12.2.1 域名系统 DNS

域名系统 (Domain Name System 缩写DNS, Domain Name 被译为域名) 是因特网的一项核心服务, 它作为可以将域名和IP 地址相互映射的一个分布式数据库, 能够使人更方便的访问互联网, 而不用去记住能够被机器直接读取的IP数串

即将易记的域名www.baidu.com 转换为计算机识别的IP 地址192.168.2.1

### DNS服务的作用

- 将域名解析为IP地址
  - 客户机向DNS服务器发送域名查询请求
  - DNS服务器告知客户机Web服务器的IP地址
  - 客户机与Web服务器通信



### 12.2.2 文件传送 FTP

FTP 是File Transfer Protocol (文件传输协议) 的英文简称, 而中文简称为“文传协议”。用于Internet 上的控制文件的双向传输。同时, 它也是一个应用程序 (Application)。

基于不同的操作系统有不同的FTP 应用程序, 而所有这些应用程序都遵守同一种协议以传输文件。在 FTP 的使用当中, 用户经常遇到两个概念: ” 下载” (Download) 和” 上传” (Upload)。

”下载”文件就是从远程主机拷贝文件至自己的计算机上；”上传”文件就是将文件从自己的计算机中拷贝至远程主机上。

### 12.2.3 万维 WWW

WWW 是环球信息网的缩写，（亦作“Web”、“WWW”、“W3”，英文全称为“World Wide Web”），中文名字为“万维网”，”环球网”等，常简称为Web。分为Web客户端和Web服务器程序。

WWW 可以让 Web 客户端（常用浏览器）访问浏览 Web 服务器上的页面。是一个由许多互相链接的超文本组成的系统，通过互联网访问。在这个系统中，每个有用的事物，称为一样“资源”；并且由一个全局“统一资源标识符”（URI）标识；这些资源通过超文本传输协议（Hypertext Transfer Protocol）传送给用户，而后者通过点击链接来获得资源。

### 12.2.4 动态主机配置 DHCP

DHCP（Dynamic Host Configuration Protocol，动态主机配置协议）是一个局域网的网络协议，使用UDP 协议工作，主要有两个用途：给内部网络或网络服务供应商自动分配IP 地址，给用户或者内部网络管理员作为对所有计算机作中央管理的手段

### 12.2.5 电子邮件 SMTP IMAP

### 12.2.6 简单网络管理 SNMP

### 12.2.7 APP 分别使用传输层的协议对照

## 12.3 传输层 TCP

### 12.3.1 基本概念对照

#### TCP 可靠性

在一个 TCP 连接中，仅有两方进行彼此通信。其可靠性的保证由以下措施完成

1. 数据块分割: 应用数据被分割成 TCP 认为最适合发送的数据块。这和UDP 完全不同，应用程序产生的数据报长度将保持不变。由 TCP 传递给 IP 的信息单位称为报文段或段 (segment)

2. **自适应的超时重传策略:** 当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。
3. 当 TCP 收到发自 TCP 连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒。
4. **检验和:** TCP 将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP 将丢弃这个报文段和不确认收到此报文段（希望发端超时并重发）
5. **数据有序:** 既然 TCP 报文段作为 IP 数据报来传输，而 IP 数据报的到达可能会失序，因此 TCP 报文段的到达也可能会失序。如果必要，TCP 将对收到的数据进行重新排序，将收到的数据以正确的顺序交给应用层。
6. **丢弃重复数据:** 既然 IP 数据报会发生重复，TCP 的接收端必须丢弃重复的数据。
7. **流量控制:** TCP 连接的每一方都有固定大小的缓冲空间。TCP 的接收端只允许另一端发送接收端缓冲区所能接纳的数据。这将防止较快主机致使较慢主机的缓冲区溢出。

## 复用与分用

假定一个机关的所有部门向外单位发出的公文都由收发室负责寄出，这相当于各部门都“复用”这个收发室。当收发室收到从外单位寄出的公文时，则要完成“分用”功能，即按照信封上写明的本机关的部门地址把公文正确进行交付

运输层的复用和分用功能也是类似的。应用层所有的应用进程都可以通过运输层再传送到 IP 层，这就是**复用**。运输层从 IP 层收到数据后必须交付给指明的应用进程，这就是**分用**。如下图 12.1 所示：

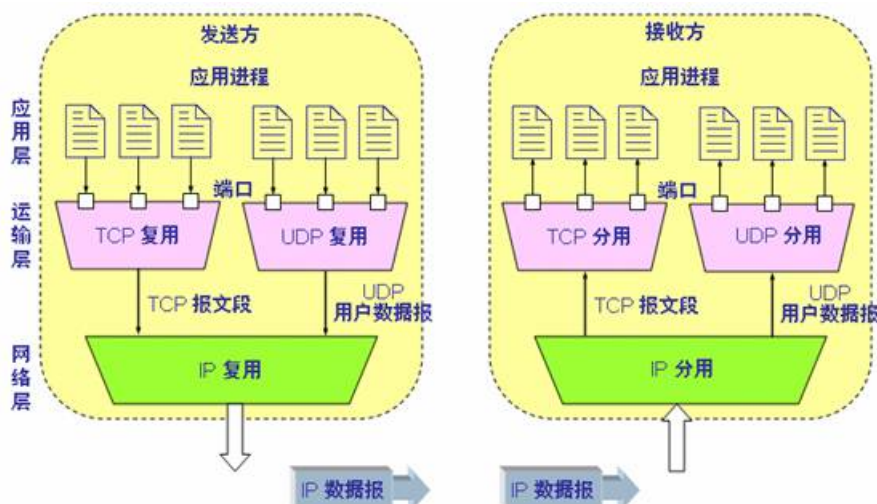


图 12.1: 分用复用图

端口

**引入问题** 运行在计算机中的进程是用进程标识符来标志的。但运行在应用层的各种应用进程却不应当让计算机操作系统指派它的进程标识符。这是因为在因特网上使用的计算机的操作系统种类很多，而不同的操作系统又使用不同格式的进程标识符，因此发送方非常可能无法识别其他机器上的进程。为了使运行不同操作系统的计算机的应用进程能够互相通信，就必须用统一的方法对 TCP/IP 体系的应用进程进行标志。而且由于进程的创建和撤销都是动态的，有时我们会改换接收报文的进程，但并不需要通知所有发送方。还有在实际应用中我们往往需要利用目的主机提供的功能来识别终点，而不需要知道实现这个功能的进程。

**解决方法** 解决这个问题的方法就是在运输层使用**协议端口号** (protocolport number)，或通常简称为端口 (port)，端口用一个16 位端口号进行标志。端口号只具有本地意义，即端口号只是为了标志本计算机应用层中的各进程。在因特网中不同计算机的相同端口号是没有联系的。虽然通信的终点是应用进程，但我们可以把端口想象是通信的终点，因为我们只要把要传送的报文交到目的主机的某一个合适的目的端口，剩下的工作（即最后交付目的进程）就由TCP 来完成

端口号分类

(1) 服务器端使用的端口号这里又分为两类

• 熟知端口号或系统端口号：0~1023,IANA 把这些端口号指派给了TCP/IP 最重要的一些应用程序.

应用程序	FTP	TELNET	SMTP	DNS	TFTP	HTTP	腾讯QQ	MSSQL Server
熟知端口号	21	23	25	53	69	80	UDP8000	5000

• 登记端口号: 1024~49151

(2) 客户端使用的端口号: 49152~65535

12.3.2 UDP

特点

- 无连接：在发送数据前不需要建立连接，当然发送数据结束完后也没有连接可释放，因此减少了开销和发送数据之前的时延.
- 尽最大努力交付：即不保证可靠交付

- **面向报文**: UDP 对应用层交下来的报文即不合并,也不拆分,而是保留这些报文的边界。也就是说,应用层交给UDP 多长的报文,UDP 就照样发送,即一次发送一个报文。

## 网络控制特点

- **UDP 没有拥塞控制**: 在网络出现堵塞也不会使源主机的发送速率降低.
- **UDP 支持 1 对 1, 1 对多, 多对 1, 多对多**
- **UDP 的首部开销小**: 只有8 个字节,比TCP 的20 个字节的首部要短

## 12.3.3 TCP

# 12.4 网络层 IP

## 12.4.1 分类的 IP 地址

现在的IP 网络使用 32 位地址,以点分十进制表示,如172.16.0.0。地址格式为:**IP地址=网络地址+主机地址**或 **IP地址=主机地址+子网地址+主机地址**

## IP 地址类型

**A 类 IP 地址** 一个A类IP地址由1字节的**网络地址**和3字节**主机地址**组成,网络地址的最高位必须是“0”,**地址范围**从1.0.0.0 到126.0.0.0。可用的A类网络有  $2^7 - 2$ (全0 与 127)共126个,每个网络能容纳1亿多个主机

**B 类 IP 地址** 一个B类IP地址由2字节的**网络地址**和2字节**的主机地址**组成,网络地址的最高位必须是“10”,地址范围从128.0.0.0 到191.255.255.255。可用的B类网络有  $2^{14} - 1$ (全0)16382个,每个网络能容纳6万多个主机

**C 类 IP 地址** 一个C类IP地址由3字节的**网络地址**和1字节**的主机地址**组成,网络地址的最高位必须是“110”。范围从192.0.0.0 到223.255.255.255。C类网络可达209万余个,每个网络能容纳254个主机

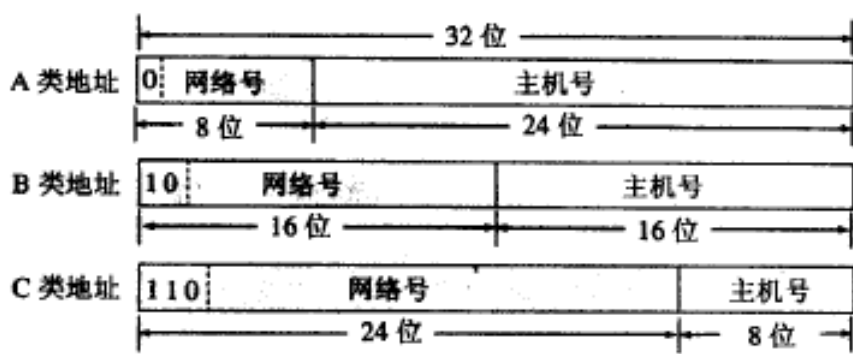


表 12.1: IP 地址的指派范围

网络类别	最大可指派的网络数	第一个可指派的网络号	最后一个可指派的网络号	每个网络中最大的主机数
A	$126(2^7 - 2)$	1	126	16777214
B	$16383(2^{14} - 1)$	128.1	191.255	65534
C	$2097151(2^{21} - 1)$	192.0.1	223.255.255	254

表 12.2: 一般不使用的 IP 地址

网络号	主机号	源地址使用	目的地址使用	代表的意思
0	0	可以	不可	在本网络上的本主机
0	host-id	可以	不可	在本网络上的某个主机host-id
全1	全1	不可	可以	只在本网络上进行广播(各路由器均不转发)
net-id	全1	不可	可以	对net-id 上的所有主机进行广播
127	非全0或全1的任何数	可以	不可以	用于本地软件环回测试之用

其他 全零 (“0. 0. 0. 0”) 地址对应于当前主机。全 “1” 的 IP 地址 (“255. 255. 255. 255”) 是当前子网的广播地址。数字127 保留给内部回环函数

## 12.4.2 子网掩码

子网掩码 (subnet mask) 又叫网络掩码、地址掩码、子网络遮罩，它是一种用来指明一个IP地址的哪些位标识的是主机所在的子网，以及哪些位标识的是主机的位掩码。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

子网掩码——屏蔽一个IP地址的网络部分的“全1”比特模式。对于 A 类地址来说，默认的子网掩码是255.0.0.0；对于 B 类地址来说默认的子网掩码是255.255.0.0；对于 C 类地址来说默认的子网掩码是255.255.255.0

利用子网掩码可以把大的网络划分成子网，也可以把小的网络归并成大的网络即超网

注意，不同的网络类型 (ABC) 也可以使用比其小的子网掩码，如 B 类地址使用 C 类子网掩码. 如

## 变长子网掩码及子网规划

IP地址: 172.16.2.121  
子网掩码: 255.255.255.0

	网络位	网络位	子网位	主机位
172.16.2.121:	10101100	00010000	00000010	01111001
255.255.255.0:	11111111	11111111	11111111	00000000
网络地址:	10101100	00010000	00000010	00000000
广播地址:	10101100	00010000	00000010	11111111

•网络地址 = 172.16.2.0

•主机地址 = 172.16.2.1–172.16.2.254

•广播地址 = 172.16.2.255

## 12.5 数据链路层 Hardware





# 第十三章 资源

## 13.1 读书路线

0.Linux 命令大全

1.Linux 高性能服务器编程

2. 构建高可用 Linux 服务器

3.Linux 多线程服务端编程

## 13.2 项目学习

参见 C++\_Project

- muduo
- nginx
- libevent
- lighttpd

## 13.3 参考

0. 学习路线 <http://www.cnblogs.com/lizhanwu/p/4164456.html>

1. 使用 socket 通信实现 FTP 服务器 <http://www.ibm.com/developerworks/cn/linux/1-cn-socketftp/>

2. 游戏服务器的见闻 [http://blog.sina.com.cn/s/blog\\_55d572ca0100uvzt.html](http://blog.sina.com.cn/s/blog_55d572ca0100uvzt.html)
3. 高性能服务器模型 <http://blog.csdn.net/zs634134578/article/details/19806429>
4. 面试题 <http://blog.csdn.net/chencheng126/article/details/44407901>  
<http://blog.chinaunix.net/uid-29087962-id-3988587.html>
5. 相关函数 <http://blog.csdn.net/caianye/article/details/8598051>