个人资料

姚军权

加关注 发私信

博客专家

访问： 1006221次
积分： 13202
等级： BLOG 7
排名： 第447名

原创： 336篇 转载： 70篇
译文： 9篇 评论： 215条

文章分类

0.外语-训练 (12)
1.软件项目-设计 (13)
1.软件项目-开发 (10)
1.软件项目-测试 (11)
2.2移动APP-安卓 (0)
2.2移动APP-苹果 (0)
2.3互联网-界面 (10)
2.4互联网-Php (11)
2.5互联网-Java (28)
3.linux-编程开发 (11)
3.linux-日常运维 (12)
3.linux-集群运维 (10)
5.4软件开发-C (30)
5.3软件开发-C++ (65)
5.2软件开发-PB (5)
5.1软件开发-Delphi (7)
4.1基本功-结构与算法 (71)
4.2基本功-数据库 (34)
4.3基本功-计算机网络 (23)
4.5基本功-编译器 (5)

转 高性能服务器(epoll c/s样列代码)

分类： 3.linux-编程开发 2013-05-13 23:06 856人阅读 评论(0) 收藏 举报

转载

http://blog.csdn.net/piaojun_pj/article/details/6103709

epoll的优点：

1.支持一个进程打开大数目的socket描述符(FD)

select 最不能忍受的是一个进程所打开的FD是有一定限制的，由FD_SETSIZE设置，默认值是2048。对于那些需要支持的上万连接数目的IM服务器来说显然太少了。这时候你一是可以选择修改这个宏然后重新编译内核，不过资料也同时指出这样会带来网络效率的下降，二是可以选择多进程的解决方案(传统的 Apache方案)，不过虽然linux上面创建进程的代价比较小，但仍旧是不可忽视的，加上进程间数据同步远比不上线程间同步的高效，所以也不是一种完美的方案。不过 epoll则没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。

2.IO效率不随FD数目增加而线性下降

传统的select/poll另一个致命弱点就是当你拥有一个很大的socket集合，不过由于网络延时，任一时间只有部分的socket是"活跃"的，但是select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是epoll不存在这个问题，它只会对"活跃"的socket进行操作---这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。那么，只有"活跃"的socket才会主动的去调用 callback函数，其他idle状态socket则不会，在这点上，epoll实现了一个"伪"AIO，因为这时候推动力在os内核。在一些 benchmark中，如果所有的socket基本上都是活跃的---比如一个高速LAN环境，epoll并不比select/poll有什么效率，相反，如果过多使用epoll_ctl,效率相比还有稍微的下降。但是一旦使用idle connections模拟WAN环境,epoll的效率就远在select/poll之上了。

3.使用mmap加速内核与用户空间的消息传递。

这点实际上涉及到epoll的具体实现了。无论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就很重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。而如果你想我一样从2.5内核就关注epoll的话，一定不会忘记手工 mmap这一步的。

4.内核微调

这一点其实不算epoll的优点了，而是整个linux平台的优点。也许你可以怀疑linux平台，但是你无法回避linux平台赋予你微调内核的能力。比如，内核TCP/IP协议栈使用内存池管理sk_buff结构，那么可以在运行时期动态调整这个内存pool(skb_head_pool)的大小--- 通过echo XXXX>/proc/sys/net/core/hot_list_length完成。再比如listen函数的第2个参数(TCP完成3次握手的数据包队列长度)，也可以根据你平台内存大小动态调整。更甚至在一个数据包面数目巨大但同时每个数据包本身大小却很小的特殊系统上尝试最新的NAPI网卡驱动架构。

epoll简介

在linux的网络编程中，很长的时间都在使用select来做事件触发。在linux新的内核中，有了一种替换它的机制，就是epoll。
相比于select，epoll最大的好处在于它不会随着监听fd数目的增长而降低效率。因为在内核中的select实现中，它是采用轮询来处理的，轮询的fd数目越多，自然耗时越多。并且，在linux/posix_types.h头文件有这样的声明：
#define __FD_SETSIZE 1024
表示select最多同时监听1024个fd，当然，可以通过修改头文件再重编译内核来扩大这个数目，但这似乎并不治本。

epoll的接口非常简单，一共就三个函数：

1. int epoll_create(int size);
创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大。这个参数不同于select()中的第一个参数，给出最大监听的fd+1的值。需要注意的是，当创建好epoll句柄后，它就是会占用一个fd值，在linux下如果查看/proc/进程id/fd/，是能够看到这个fd的，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

2. int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
epoll的事件注册函数，它不同与select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。第一个参数是epoll_create()的返回值，第二个参数表示动作，用三个宏来表示：
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；
第三个参数是需要监听的fd，第四个参数是告诉内核需要监听什么事，struct epoll_event结构如下：

```
struct epoll_event {
__uint32_t events; /* Epoll events */
epoll_data_t data; /* User data variable */
};
```

events可以是以下几个宏的集合：
EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；
EPOLLOUT：表示对应的文件描述符可以写；
EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；
EPOLLERR：表示对应的文件描述符发生错误；
EPOLLHUP：表示对应的文件描述符被挂断；
EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。
EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

3. int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
等待事件的产生，类似于select()调用。参数events用来从内核得到事件的集合，maxevents告之内核这个events有多大，这个maxevents的值不能大于创建epoll_create()时的size，参数timeout是超时时间（毫秒，0会立即返回，-1将不确定，也有说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回0表示已超时。

下面是我在redhat9上用epoll实现的简单的C/S通信，已经运行通过了。

server.c

[c-sharp] view plain
copy

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <sys/epoll.h>

#define BUFFER_SIZE 40
#define MAX_EVENTS 10

int main(int argc, char * argv[])
{
int server_sockfd;// 服务器端套接字
int client_sockfd;// 客户端套接字
int len;
struct sockaddr_in my_addr; // 服务器网络地址结构体
struct sockaddr_in remote_addr; // 客户端网络地址结构体
int sin_size;
char buf[BUFFER_SIZE]; // 数据传送的缓冲区
memset(&my_addr,0,sizeof(my_addr)); // 数据初始化--清零
my_addr.sin_family=AF_INET; // 设置为IP通信
my_addr.sin_addr.s_addr=INADDR_ANY;// 服务器IP地址--允许连接到所有本地地址上
my_addr.sin_port=htons(8000); // 服务器端口号
// 创建服务器端套接字--IPv4协议，面向连接通信，TCP协议
if((server_sockfd=socket(PF_INET,SOCK_STREAM,0))<0)
{
perror("socket");
return 1;
}
// 将套接字绑定到服务器的网络地址上
if (bind(server_sockfd,(struct sockaddr *)&my_addr,sizeof(struct sockaddr))<0)
{
perror("bind");
return 1;
}
// 监听连接请求--监听队列长度为5
listen(server_sockfd,5);
sin_size=sizeof(struct sockaddr_in);
// 创建一个epoll句柄
int epoll_fd;
epoll_fd=epoll_create(MAX_EVENTS);
if(epoll_fd==-1)
{
perror("epoll_create failed");
exit(EXIT_FAILURE);
```

```
}
struct epoll_event ev;// epoll事件结构体
struct epoll_event events[MAX_EVENTS];// 事件监听队列
ev.events=EPOLLIN;
ev.data.fd=server_sockfd;
// 向epoll注册server_sockfd监听事件
if(epoll_ctl(epoll_fd,EPOLL_CTL_ADD,server_sockfd,&ev)==-1)
{
perror("epll_ctl:server_sockfd register failed");
exit(EXIT_FAILURE);
}
int nfds;// epoll监听事件发生的个数
// 循环接受客户端请求
while(1)
{
// 等待事件发生
nfds=epoll_wait(epoll_fd,events,MAX_EVENTS,-1);
if(nfds==-1)
{
perror("start epoll_wait failed");
exit(EXIT_FAILURE);
}
int i;
for(i=0;i<nfds;i++)
{
// 客户端有新的连接请求
if(events[i].data.fd==server_sockfd)
{
// 等待客户端连接请求到达
if((client_sockfd=accept(server_sockfd,(struct sockaddr *)&remote_addr,&sin_size))<0)
{
perror("accept client_sockfd failed");
exit(EXIT_FAILURE);
}
// 向epoll注册client_sockfd监听事件
ev.events=EPOLLIN;
ev.data.fd=client_sockfd;
if(epoll_ctl(epoll_fd,EPOLL_CTL_ADD,client_sockfd,&ev)==-1)
{
perror("epoll_ctl:client_sockfd register failed");
exit(EXIT_FAILURE);
}
printf("accept client %s/n",inet_ntoa(remote_addr.sin_addr));
}
// 客户端有数据发送过来
else
{
len=recv(client_sockfd,buf,BUFFER_SIZE,0);
if(len<0)
{
perror("receive from client failed");
exit(EXIT_FAILURE);
}
```

```
printf("receive from client:%s",buf);
send(client_sockfd,"I have received your message.",30,0);
}
}
}
return 0;
}
```

client.c

[c-sharp] view plain
copy

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdlib.h>

#define BUFFER_SIZE 40

int main(int argc, char *argv[])
{
int client_sockfd;
int len;
struct sockaddr_in remote_addr; // 服务器端网络地址结构体
char buf[BUFFER_SIZE]; // 数据传送的缓冲区
memset(&remote_addr,0,sizeof(remote_addr)); // 数据初始化--清零
remote_addr.sin_family=AF_INET; // 设置为IP通信
remote_addr.sin_addr.s_addr=inet_addr("127.0.0.1");// 服务器IP地址
remote_addr.sin_port=htons(8000); // 服务器端口号
// 创建客户端套接字--IPv4协议，面向连接通信，TCP协议
if((client_sockfd=socket(PF_INET,SOCK_STREAM,0))<0)
{
perror("client socket creation failed");
exit(EXIT_FAILURE);
}
// 将套接字绑定到服务器的网络地址上
if(connect(client_sockfd,(struct sockaddr *)&remote_addr,sizeof(struct sockaddr))<0)
{
perror("connect to server failed");
exit(EXIT_FAILURE);
}
// 循环监听服务器请求
while(1)
{
printf("Please input the message:");
scanf("%s",buf);
// exit
```

```
if(strcmp(buf,"exit")==0)
{
break;
}
send(client_sockfd,buf,BUFFER_SIZE,0);
// 接收服务器端信息
len=recv(client_sockfd,buf,BUFFER_SIZE,0);
printf("receive from server:%s/n",buf);
if(len<0)
{
perror("receive from server failed");
exit(EXIT_FAILURE);
}
}
close(client_sockfd);// 关闭套接字
return 0;
}
```

makefile

[c-sharp] view plain
copy

```
#This is the makefile of EpollTest

.PHONY:all
all:server client
server:
gcc server.c -o server
client:
gcc client.c -o client
clean:
rm -f server client
```

上一篇 高性能服务器(epoll exsample code)
下一篇 高性能服务器(libevent网络库)

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC
coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap