## 安装虚拟机<http://jingyan.baidu.com/article/14bd256e0ca52ebb6d26129c.html>

**linux操作系统**

SCSI(**['skʌzɪ]**)硬盘是采用SCSI接口的硬盘，SCSI是Small Computer System Interface（小型计算机系统接口）的缩写，使用50针接口，外观和普通硬盘接口有些相似。SCSI硬盘和普通IDE硬盘相比有很多优点：接口速度快，并且由于主要用于服务器，因此硬盘本身的性能也比较高，硬盘转速快，缓存容量大，CPU占用率低，扩展性远优于IDE硬盘，并且支持热插拔。

在linux下一切都是文件，大致分为四种：**普通文件**，**设备文件**，**目录文件**和**链接文件**，文件系统是一个树，所有的文件都是挂载（mount）到这个文件树上的，挂载在文件树中的位置就是挂载点，当然这种关系可以中断和改变。每一个分区和某个目录对应。分区有：主分区，扩展分区，逻辑分区，win下的c盘就安装在主分区上。当运行大型程序时候linux会划出一个区域作为临时内存，在win下称为：虚拟内存。在linux下称为：交换分区SWAP。

分区的文件格式：win是NTFS,FAT32.\_\_\_\_linux则是EXT2，EXT3，XFS.

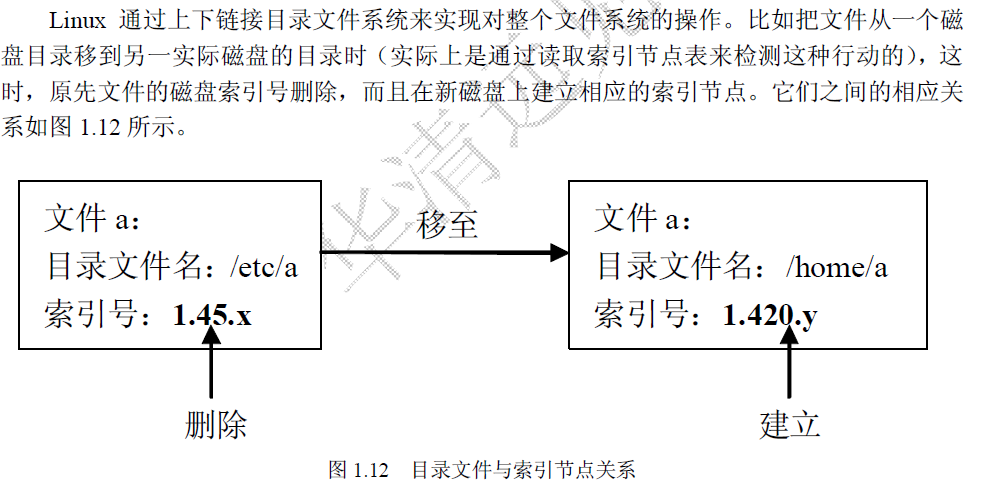
GRUB是一种操作系统的引导装入器，负责装载内核并引导操作系统。

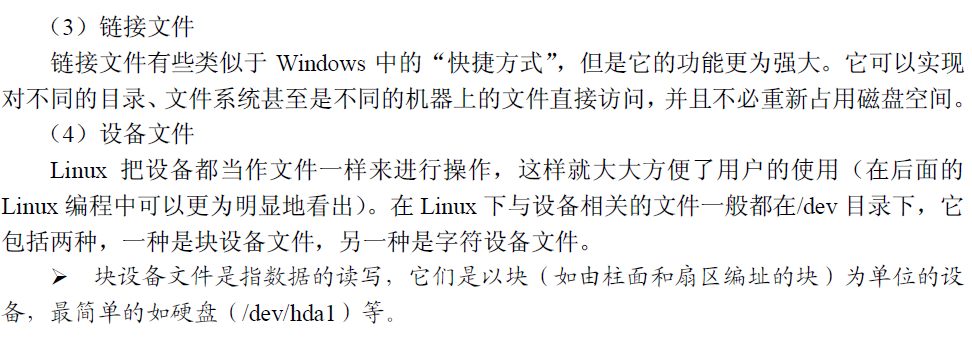
Root权限是linux的超级用户，默认的主目录在/root下。

注意：普通文件如同win下的文本文件，word，exe等等。而目录文件时linux中存储文件名的唯一的地方，对目录文件进行操作时候一般只是对目录名和文件名的对应关系进行操作，不涉及文件的内容。为了区别每个文件，系统给每个文件都赋予了一个唯一的数值，称为索引节点。索引节点存储在一个索引节点表中（Inode Table）.每个磁盘或者分区都有自己的索引节点表。文件的索引节点包含文件的所有信息，如文件地址类型大小。

系统把1号索引节点给了根目录，这也是根目录文件在磁盘上的位置。根目录文件的内容包括：旗下所有的文件名，目录名，及其各自的索引节点号的列表

Linux可以通过查找根目录开始的一个目录链来找到系统中的任何文件。





## 2.常用命令：

命令帮助:ls --help

详细帮助信息:man ls

**获取root权限：**

|  |
| --- |
| sudo  passwd root  先设置密码  su root ,切换root账号 |

创建文件夹:

**mkdir** /home/u1 创建文件夹/home/u1  
chown oracle /home/u1 表示改变目录所有者为oracle账户;  
chgrp dba /home/u1 改变/home/u1目录为dba所有;  
chmod 755 /home/u1 表示oracle账户对/home/u1目录有755权限;  
**rmdir** /home/u1 表示删除/home/u1目录  
创建文件

**touch** /etc/fdfs/a.txt

编辑文件

vi /etc/fdfs/a.txt,保存退出

|  |
| --- |
| 有3种模式，命令行模式（ESC进入），插入模式（i进入），底行模式（：进入）。一般编辑完文档后，先按ESC退出插入模式，然后在底行输入:x来保存退出 |

**pwd**显示当前路径;  
**df**查看系统的硬件信息  
**ls -lrt** 第一个参数**l**表示显示详细列表，**-t**表示按时间排序,**-r**反向排序  
  
cat orcl\_ora\_3436.trc|grep bucket :查看文件内容

以下查看相关文件内容:  
more /etc/oratab more一页一页显示  
cat /etc/passwd 链接多个文档显示  
cat /etc/group

**Cd ..返回上层目录**

**清屏 Clear**

**查看nginx 进程信息**：ps -ef|grep nginx

查看进程列表 ps aux

查看实时的进程状态：top

## 3.脚本文件 .sh

Sh 指令执行一个文件(shell)

|  |
| --- |
| Sh ./some.sh |

如果权限不够使用sudo

运行文件的时候使用sudo ./some.pl

|  |
| --- |
| sudo 执行命令的流程是当前用户切换到root（或其它指定切换到的用户），然后以root（或其它指定的切换到的用户）身份执行命令，执行完成后，直接退回到当前用户；而这些的前提是要通过sudo的配置文件/etc/sudoers来进行授权； |

**压缩解压缩：**

1.压缩命令：

　　命令格式：tar  -zcvf   压缩文件名.tar.gz   被压缩文件名

      可先切换到当前目录下。压缩文件名和被压缩文件名都可加入路径。

2.解压缩命令：

　　命令格式：tar  -zxvf   压缩文件名.tar.gz

　　解压缩后的文件只能放在当前的目录。

**下载文件工具 wget**

[**http://www.cnblogs.com/peida/archive/2013/03/18/2965369.html**](http://www.cnblogs.com/peida/archive/2013/03/18/2965369.html)

|  |
| --- |
| 1、使用wget下载单个文件  wget [http://some.zip](http://some.zip/)  2、使用wget -O下载并以不同的文件名保存  3、使用wget –limit -rate限速下载  4、使用wget -c断点续传  5、使用wget -b后台下载  9、使用wget -i下载多个文件  首先，保存一份下载链接文件  cat > filelist.txt  url1  url2  接着使用这个文件和参数-i下载  wget -i filelist.txt  10、使用wget –mirror镜像网站  下面的例子是下载整个网站到本地。   wget –mirror -p –convert-links -P ./LOCAL URL  –miror:开户镜像下载  -p:下载所有为了html页面显示正常的文件  –convert-links:下载后，转换成本地的链接  -P ./LOCAL：保存所有文件和目录到本地指定目录  14、使用wget -r -A下载指定格式文件  可以在以下情况使用该功能  下载一个网站的所有图片  下载一个网站的所有视频  下载一个网站的所有PDF文件  **wget -r -A.pdf url** |

## 4.目录结构说明

<http://www.blogjava.net/qileilove/archive/2013/10/24/405584.html>

**/ (root, 根目录)：与开机系统有关；**

/： 根目录，一般根目录下只存放目录，不要存放文件，/etc、/bin、/dev、/lib、/sbin应该和根目录放置在一个分区中

　　/bin:/usr/bin: 可执行二进制文件的目录，如常用的命令ls、tar、mv、cat等。

　　/boot： 放置linux系统启动时用到的一些文件。

**/usr (unix software resource)**：与软件安装**/**执行有关；

**/var (variable)**：与系统运作过程有关。

/dev： 存放linux系统下的设备文件，访问该目录下某个文件，相当于访问某个设备，常用的是挂载光驱mount /dev/cdrom /mnt。 device

/etc： 系统配置文件存放的目录，不建议在此目录下存放可执行文件，重要的配置文件有/etc/inittab、/etc/fstab、/etc/init.d、/etc/X11、/etc/sysconfig、/etc/xinetd.d修改配置文件之前记得备份。注：/etc/X11存放与x windows有关的设置。

## 5. gcc,g++,make

http://www.cnblogs.com/baochuan/archive/2012/07/11/2583593.html

单独编译c语言用gcc，编译c++使用g++,

编译一个项目使用make指令，（保证编译必要文件，保证）

3.make命令

make命令的一般格式：

make【选项】【make描述文件】【宏定义】【目的文件】

make首先分析所有的宏定义参数（带嵌入等号的参数）并进行赋值，宏命令行覆盖在makefile中进行的定义，然后检验选项标志，最后递归的对每一个目的依赖文件进行检查。

unix下常用的选项如下：

f file　 指定file文件为描述文件，如果file参数为"-"符，那么描述文件指向标准输入。如果没有"-f"参数，则系统将默认当前目录下名为 makefile或者名为**Makefile**的文件为描述文件。

描述文件内容结构

#用“井”号表明注释。

target（要生成的文件）: dependencies（被依赖的文件）

#命令前面用的是“tab”而非空格。误用空格是初学者容易犯的错误！

命令1

命令2

命令3

.

.

.

命令n

#可以使用“\”表示续行。注意，“\”之后不能有空格！

Tab 必须是纯粹的8字符tab，不能是空格，qt编辑器默认不是。

|  |
| --- |
| runexe:ma.o mb.o  gcc -g ma.o mb.o -o runexe  ma.o:ma.h ma.c  gcc -g -c ma.c  mb.o:mb.h mb.c  gcc -g -c mb.c  clean:  rm -f rune\*.o |

从上到下依赖顺序，object文件依赖于源文件，可执行文件依赖于object的链接

很显然make执行时候如果发现依赖项在后面的声明中就进入依赖项目命令中，再依次递归

执行顺序如下

|  |
| --- |
| gcc -g -c ma.c  gcc -g -c mb.c  gcc -g ma.o mb.o -o runexe |

目标和条件之间的关系是：欲更新目标，必须首先更新它的所有条件；所有条件中只要有一个条件被更新了，目标也必须随之被更新。所谓“更新”就是执行一遍规则中的命令列表，命令列表中的每条命令必须以一个Tab开头，注意不能是空格，Makefile的格式不像C语言的缩进那么随意，对于Makefile中的每个以Tab开头的命令，make会创建一个Shell进程去执行它。

可以发现有字符串反复出现可以用变量代替

|  |
| --- |
| 变量：“变量”指的是用一个字符串代替另一个字符串的功能。在makefile中可以使用“=”号来定义变量，使用“$（变量名）”来使用变量；还可以用“:=”追加变量的内容。习惯上，变量名使用大写。    定义：  　　变量名=字符串  使用：  　　$(变量名)  追加：  　　变量名:=字符串 |

|  |
| --- |
| OBJECT=ma.o mb.o  TARGET=runexe  $(TARGET):$(OBJECT)  gcc -g $(OBJECT) -o $(TARGET)  ma.o:ma.h ma.c  gcc -g -c ma.c  mb.o:mb.h mb.c  gcc -g -c mb.c  clean:  rm -f \*.txt |

**伪目标**

 　　首先要明确，并不是所有的目标文件都对应于磁盘文件，有的目标文件的存在只是为了形成一条规则，从而使make完成特定的工作，并不生成新的目标文件，这样的目标称为伪目标。——如上面Makefile中的clean。常用的还有all。

　　条件语句可以将一个变量与其他变量的值进行比较，或将一个变量与一个字符串常量相比较。——这样就可以根据变量的值执行或忽略Makefile文件中的一部分脚本。

　　注意：条件语句用于控制make时间看见的Makefile文件部分，而不能用于执行时控制shell命令。

　　条件语句3条指令：ifeq, else 和endif

|  |
| --- |
| ifeq($(VAR), 1)  gcc -o exe1 module  else  gcc -o exe2 module  endif |

Make install ,编译成功之后安装可执行文件到系统中，一般为/usr/local/bin目录。

内置变量

|  |
| --- |
| 有时候为了简化命令的书写，可以在Makefile中定义一些宏和使用缩写，下面是几个很使用的缩写：    　　　　 $@ 代表该目标的全名  　　　　 $\* 代表已经删除了后缀的目标名  　　　　 $< 指代第一个前置条件。比如，规则为 t: p1 p2，那么$< 就指代p1。 |

|  |
| --- |
| runexe:ma.o mb.o  gcc -g ma.o mb.o -o $@  ma.o:ma.h ma.c  gcc -g -c $\*.c  mb.o:mb.h mb.c  gcc -g -c $\*.c  clean:  rm -f \*.txt |

### **模式匹配**

Make命令允许对文件名，进行类似正则运算的匹配，主要用到的匹配符是%。比如，假定当前目录下有 f1.c 和 f2.c 两个源码文件，需要将它们编译为对应的对象文件。

%.o: %.c

等同于下面的写法。

f1.o: f1.c

f2.o: f2.c

关闭指令回显，调用shell的变量$$，每行指令都是一个独立的shell

|  |
| --- |
| OBJECT=ma.o mb.o  runexe:ma.o mb.o  gcc -g ma.o mb.o -o $@  ma.o:ma.h ma.c  @gcc -g -c $\*.c;echo "ljmljm$(OBJECT)" $$HOME  mb.o:mb.h mb.c  gcc -g -c $\*.c  .PHONY:clean  clean:  rm -f \*.o |

显示

|  |
| --- |
| ljm@ubuntu:~/Project/C/TestPuerC/TestPuerC$ make  ljmljmma.o mb.o /home/ljm  gcc -g -c mb.c  gcc -g ma.o mb.o -o runexe |

## 5.1文件描述符file descriptor **fd**

**相当于文件句柄**

文件描述符在形式上是一个非负整数。实际上，它是一个索引值，指向[内核](http://baike.baidu.com/view/1366.htm)为每一个进程所维护的该进程打开文件的记录表。当程序打开一个现有文件或者创建一个新文件时，内核向进程返回一个文件描述符。

文件描述符是由[无符号整数](http://baike.baidu.com/view/6052699.htm)表示的句柄，进程使用它来标识打开的文件。文件描述符与包括相关信息（如文件的打开模式、文件的位置类型、文件的初始类型等）的文件对象相关联，这些信息被称作文件的上下文。

基于文件描述符的I/O操作兼容[POSIX](http://baike.baidu.com/view/209573.htm)标准。

例如写一个文件

|  |
| --- |
| int main(){  // PB();  // PA();  int fd; int numbytes; char path[] = "file"; char buf[256];  /\*  \* O\_CREAT: 如果文件不存在则创建  \* O\_RDONLY:以只读模式打开文件  \*/  fd = open(path, O\_CREAT | O\_RDONLY, 0644);  if(fd < 0)  { perror("open()");  exit(EXIT\_FAILURE);  }  memset(buf, 0x00, 256);  while((numbytes = read(fd, buf, 255)) > 0){  printf("%d bytes read: %s", numbytes, buf);  memset(buf, 0x00, 256);  }  printf("fd=%d",fd);  getchar();  return 0;  } |

## 5.2几种io操作策略select,poll,epoll,kqueue

1. 遍历方式的区别。select判断是否有事件发生是遍历的，而epoll是事件响应的，一旦句柄上有事件来了，就马上选出来。

2. 数目的区别。select一般由一个内核参数（1024）限制了监听的句柄数，但是epoll通常受限于打开文件的数目，通常会打得多。

3. epoll自身，还有两种触发方式。水平触发和边缘触发。边沿触发的效率更高（高了不少，但是编程的时候要小心处理每个时间，防止漏掉处理某些事件）。

Select

|  |
| --- |
| int select (int n, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout); |

readfds：期望读取操作的文件，如果方法返回的时候输入的值还存在则表示可以操作。

Timeout: 方法阻塞的时间。

Epoll., 很适合同一个时间只有部分socket是活跃的情况

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。

三个函数

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()[上面一个函数]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。

例子:客户端 python

|  |
| --- |
| import socket  cli = socket.socket( socket.AF\_INET, socket.SOCK\_STREAM )  cli.connect(("127.0.0.1", 8888))  # link socket  while True:  inputed = raw\_input("press any key to continue...")  #get input info  if len(inputed)==0:  print "exiting..."  exit(0)  cli.send(inputed)  #send msg to server  print cli.recv(4096)  #get msg from server |

服务端:c++

大致流程:

创建socket对象

创建epfd对象,

epoll\_ctrl绑定socket事件,

bind(),listen();

while(true){ epoll\_wait();}

|  |
| --- |
| #include <iostream>  #include <sys/socket.h>  #include <sys/epoll.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <fcntl.h>  #include <unistd.h>  #include <stdio.h>  #include <errno.h>  #include <stdlib.h>  #include <strings.h>  #include <string.h>  using namespace std;  int TIMEOUT\_MS=6000;  int SERVER\_PORT=8888;  void perror\_and\_exit(const char \* szErrorInfo){  printf("errno=%d,", errno);  perror(szErrorInfo);  exit(1);  }  int main( int argc, char \*argv[] ){  int epfd = epoll\_create(256);  if( epfd == -1 ){  perror\_and\_exit("epoll\_create failed");  }  int server\_listenfd = socket( AF\_INET, SOCK\_STREAM, 0 );  if( server\_listenfd == -1 ){  perror\_and\_exit("socket() failed");  }  {  struct epoll\_event ev;  ev.events = EPOLLIN|EPOLLET;  ev.data.fd = server\_listenfd;  // 新增链接的监听函数,可读|边沿触发  cout<<"init server fd="<<server\_listenfd<<endl;  epoll\_ctl( epfd, EPOLL\_CTL\_ADD, server\_listenfd, &ev );  }  // listening on the port  {  //初始化链接地址  struct sockaddr\_in serveraddr;  bzero( &serveraddr, sizeof(serveraddr) );  serveraddr.sin\_family = AF\_INET;  serveraddr.sin\_port = htons(SERVER\_PORT);  const char \*local\_ip = "0.0.0.0";  inet\_aton( local\_ip, &serveraddr.sin\_addr );//配置ip地址  // 将socket绑定到地址上去  bind( server\_listenfd, (sockaddr\*)&serveraddr, sizeof(serveraddr) );  listen( server\_listenfd, 10 );//开始监听  }  static const int MAX\_READY\_EVENTS = 256;  struct epoll\_event events[MAX\_READY\_EVENTS];  while (true){  printf("working...\n");  int nfds = epoll\_wait( epfd, events, MAX\_READY\_EVENTS, TIMEOUT\_MS );  // 开始等待事件发生,如果返回0则表示超时,如果不超时则返回需要处理的事件数目,  printf("eploo\_wait's return value ntds=%d\n", nfds);  for( int nIndex = 0; nIndex<nfds; ++nIndex ){  struct epoll\_event & active\_event = events[nIndex];  if( active\_event.data.fd == server\_listenfd ) {  // 当一个客户端链接过来的时候执行如下代码  cout<<"accepting new client..."<<endl;  struct sockaddr\_in clientaddr;  socklen\_t clilen = sizeof(clientaddr);  int connfd = accept( server\_listenfd, (sockaddr\*)&clientaddr, &clilen );  // 获取客户端链接过来产生的fd,以后客户端发送的消息都是这个fd发送的.  if( connfd<0 ){  perror\_and\_exit("connfd<0");  }  char \* remote\_ip = inet\_ntoa( clientaddr.sin\_addr);  cout<<"accept a connection from " << remote\_ip<<"client's fd="<<connfd<<endl;  //绑定fd事件  epoll\_event clientEv;  clientEv.data.fd = connfd;  clientEv.events = EPOLLIN|EPOLLET;  int add\_result = epoll\_ctl( epfd, EPOLL\_CTL\_ADD, connfd, &clientEv);  if ( -1==add\_result ){  perror\_and\_exit( "EPOLL\_CTL\_ADD failed" );  }  }  else if( active\_event.events & EPOLLIN ){  int client\_link\_fd=active\_event.data.fd;  cout<<"EPOLLIN event emit,fd="<<client\_link\_fd<<endl;  char line[1024]={0};  int nReaded = read( client\_link\_fd, line, sizeof(line)-1 );  if( nReaded <= 0 ){  // 客户端传递的内容为空,说明关闭了,  perror("nReaded<=0");  if ( nReaded == 0 ){  printf("client closed gracefully!\n");  }  //删除事件  epoll\_ctl( epfd, EPOLL\_CTL\_DEL, client\_link\_fd, 0 );  close( client\_link\_fd );//关闭链接  }  else{  // 打印出客户端传递的内容  cout<<"Readed: ["<<line<<"]"<<endl;  char\* ok="getmsg\_ok";  int nSended = send( client\_link\_fd, ok, strlen(ok), 0 );  // 返回内容  if( -1==nSended ){  perror( "send" );  printf("errno=%d\n", errno );  if( errno == EAGAIN ){  printf("It's time to do something. the buffer is not sent.\n");  }  }  }  }  else{  printf("errno=%d, unkown event type!\n", errno );  }    }  }  return 0;  } |

先运行服务端.

## 6.qt 调试redis

1.下载redis源码

2.新建纯c项目

3.修改编译配置,删除原来的qmake,新增make step,注意配置目录. 点击编译.如果正常采用

attach to running program,调试redis.