cortex a9开发板的使用

Ubuntu12.04(实体机，虚拟机的使用出现问题自行解决)下使用minicom对开发板进行调试，开发板芯片为coretex a9。

# 1.minicom安装及配置

## 1.1.安装

sudo apt-get install minicom；

## 1.2.配置

root@ubuntu:~# minicom –s

正常应弹出如下图示：

Serial port setup [Enter]

+-------------------------------------------------------------+

| A - Serial Device : /dev/ttyUSB0 |

| B - Lockfile Location : /var/lock |

| C - Callin Program : |

| D - Callout Program -: |

| E - Bps/Par/Bits : 115200 8N1 |

| F - Hardware Flow Control : No |

| G - Software Flow Control : No |

| |

| Change which setting? |

+-------------------------------------------------------------+

在打开的 [configuration] 界面中，通过上下键选中：Serial port setup选项，回车进行配置。

此时所示图标在“Change which setting”中：

键入“A”，此时光标移到第A项对应处：将原字符串修改为 ttyS1，注意这里是根据串口在系统中被识别的名字进行命名的，可到/dev目录下查看所有tty设备进行查询当前系统将其识别成了什么名字的设备。例如我使用的是usb转串口设备，系统将其识别成了ttyUsb0的设备。键入“回车”回到“Change which setting”中。

键入“F”，此时第F项显示为：Hardware Flow Control : No 。

其他选项为默认设置。

键入“回车”，退到 [configuration]界面，选择 Save setup as dfl，保存为系统默认配置。

选择 Exit from Minicom ，退出 minicom。

这时候正常来说就已经将minicom 的配置完成了，它可以正常识别名称为“ttyS1”的串口设备。重新打开minicom，命令如下：

root@ubuntu:~# minicom

开发板设备发串口数据，此时在minicom 上就会显示收到的数据。

要退出minicom，需如下操作：在minicom界面，先按 ctrl+A 然后q键，选择 yes 退出。

# 2.板子与pc传输文件

原本希望使用lrzsz通过串口传输文件，但是后来调试后发现速度不快，而且还需要再开发板上安装库，就没有使用这种方式。

最终采用了局域网内部、nfs连接的方式进行文件的传输。

## 2.1.安装nfs服务器

#apt-get install nfs-kernel-server；该命令会在本机上安装nfs服务器库，可以使本机当做一个服务器端使用，而开发板当做客户端使用。如果需要进行本机的测试(本机既当做服务器又当做客户端)执行如下命令：#apt-get install nfs-common；

安装结束后，开始配置nfs，配置文件是/etc/exports，主要是设置服务器的共享目录以及权限的：#sudo vim /etc/exports，将自己的共享目录添加在最后即可，例如我的配置就是：

lmz@lmz-desktop:~/arm9$ cat /etc/exports

# /etc/exports: the access control list for filesystems which may be exported

# to NFS clients. See exports(5).

#

# Example for NFSv2 and NFSv3:

# /srv/homes hostname1(rw,sync) hostname2(ro,sync)

#

# Example for NFSv4:

# /srv/nfs4 gss/krb5i(rw,sync,fsid=0,crossmnt)

# /srv/nfs4/homes gss/krb5i(rw,sync)

#

/arm9test \*(rw,sync,no\_root\_squash)

其中/arm9test表示服务器要共享的目录，把这个改成你要共享的目录就可以了，\*表示允许同一网段内所有的ip地址访问服务器，当然也可以指定ip访问，例如写死为“/arm9test 127.0.0.3(rw,sync,no\_root\_squash)”，就只允许127.0.0.3的ip地址的板子访问该共享区域；设置好配置文件后，保存退出，执行

#sudo exportfs -r 更新exportfs；执行这个命令的时候可能会提示一些错误，如下：

lmz@lmz-desktop:~/arm9$ sudo exportfs -r

exportfs: /etc/exports [1]: Neither 'subtree\_check' or 'no\_subtree\_check' specified for export "218.192.161.80:/arm9test".

Assuming default behaviour ('no\_subtree\_check').

NOTE: this default has changed since nfs-utils version 1.0.x

exportfs: /etc/exports [2]: Neither 'subtree\_check' or 'no\_subtree\_check' specified for export "218.192.161.65:/arm9test".

Assuming default behaviour ('no\_subtree\_check').

NOTE: this default has changed since nfs-utils version 1.0.x

具体原因我也不清楚是为什么，待以后搞明白了再上来说清楚，要指出的是，这些问题并不影响后面的使用。

然后重启nfs服务，执行下面命令：#sudo /etc/init.d/portmap restart；#sudo /etc/init.d/nfs-kernel-server restart；成功启动后，执行：#showmount –e命令查看NFS server的export list.

我的机器上的输出是：

lmz@lmz-desktop:~/arm9$ showmount -e

Export list for lmz-desktop:

/arm9test 127.0.0.3

## 2.2.本机测试nfs连接

至此，可以在自己的机子上测试一下nfs服务了，具体过程如下：

#mount serverip:/arm9test /mnt

其中serverip实际上就是服务器的ip,千万不要少了serverip后面的那个“:”哦，会出错的。

如果挂载成功的话进到/mnt是可以看到/arm9test下面的文件的，如果出错的话就要查找一下原因了，最好去鸟哥那里看一下，因为上面讲得很清楚。

## 2.3.开发板和pc通过nfs互联

### 2.3.1.修改开发板ip

正常情况下执行“ifconfig”命令就可以查看当前的开发板ip地址设置信息，但是minicom里执行该命令后发现没有任何信息，经google后发现可通过指定网口实现，故此使用命令“ifconfig eth0”，ip地址信息成功弹出。

如果使用该命令依旧有错误，可通过尝试重启网络接口修复，方式如下：ifconfig eth0 down; ifconfig eth0 up；

修改ip地址：ifconfig eth0 new\_ip；例如“ifconfig eth0 138.138.138.133”，就可以将ip地址修改成指定值；注意开发板的ip最好与pc的ip在同一个网段。

### 2.3.2.编译开发板可执行程序

特别需要注意的是编译时应添加“-static”选项，使用静态的方式进行编译，否则无法将动态库编译到程序中，在板子上执行时将无法找到动态库，报错。

### 2.3.3.开发板挂载共享文件夹

正常的挂载命令时：mount –t nfs –o nolock 138.138.138.132:/home/wujl/NfsFolder /mnt，该命令正确执行后，开发板上的/mnt目录将挂载为服务器上的……/NfsFolder目录，内容将完全与该目录一致。

但由于我使用的是android系统的开发板，直接执行该命令报出“mount no this valid”之类的错误，后来google后发现只需在最前增加busybox即可，例如：busybox mount –t nfs –o nolock 138.138.138.132:/home/wujl/NfsFolder /mnt，执行该命令后再进入/mnt目录发现目录下已经完全跟NfsFolde目录一致了。

# 3.交叉编译环境搭建

pc上写的程序要再开发板上运行，需要搭建交叉编译环境，这里描述基本的搭建方法和步骤。

下载交叉编译工具链：arm-linux-gcc-4.4.3-20100728.tar.gz；

解压到某一目录；

解压得到的文件夹中将其toolschain目录拷贝到/usr/local目录下，现在交叉编译程序集都在/usr/local/toolschain/4.4.3/bin目录中了；

修改环境变量，将交叉编译器的路径加入到PATH环境变量中，方便以后直接使用该命令：vim /etc/bash.bashrc;在打开的文件最后加入如下两行语句：export PATH=$PATH:/usr/local/toolschain/4.4.3/bin (换行后)export PTAH。如果第二句报错，可直接将其删除；

保存上面的设置：source /root/.bashrc；

查看是否保存环境变量成功：echo $PATH，如果有了最新加入的语句，说明成功。也可以直接在终端执行“arm-linux-gcc -v”命令，看是否有正确的版本信息返回确定是否已经设置成功。

UNIX环境高级编程第二版

# 1.文件I/O(第三章)

//TODO

# 2.文件和目录(第四章)

## 2.1.stat、fstat、lstat

#include <sys/stat.h>

int stat(const char \*restrict pathname, struct stat \*restrict buf);

int fstat(int filedes, struct stat \*buf);

int lstat(const char \*restrict pathname, struct stat \*restrict buf);

fstat函数获取已经在文件描述符fileds上打开的文件的信息，lstat类似于stat，但当命名的文件是一个符号链接时，lstat返回该符号连接的有关信息，而不是该链接文件指向文件的信息。

struct stat

{

mode\_t st\_mode; /\*file type & mode : 可以判断是文件还是目录\*/

ino\_t st\_ino; /\*i-node number\*/

dev\_t st\_dev; /\*device number : file system\*/

dev\_t st\_rdev; /\*device number for special files\*/

nlink\_t st\_nlink; /\*number of links\*/

uid\_t st\_uid; /\*User id of owner\*/

gid\_t st\_gid; /\*group id of owner\*/

off\_t st\_size; /\*size in bytes, for regular files\*/

time\_t st\_atime; /\*time of last access \*/

time\_t st\_mtime; /\*time of last modification\*/

time\_t st\_ctime; /\*time of last file status change\*/

blksize\_t st\_blksize; /\*best I/O block size\*/

blkcnt\_t st\_blocks; /\*number of disk blocks allocated\*/

}

可以根据如上结构体中的st\_mode判断文件类型，如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| **输入参数** | **宏** | **文件类型** | **备注** |
| st\_mode | S\_ISREG() | 普通文件(regular file) | 使用：if(S\_ISREG(st\_mode)); |
| S\_ISDIR() | 目录文件(directory) | 同上 |
| S\_ISCHR() | 字符特殊文件 | 同上 |
| S\_ISBLK() | 块特殊文件 | 同上 |
| S\_ISFIFO() | 管道 / FIFO | 同上 |
| S\_ISLNK() | 符号链接 | 同上 |
| S\_ISSOCK() | 套接字 | 同上 |
|  |  |  |  |

可以使用如上结构体中的st\_size获取文件大小，但是该字段只对普通文件、目录文件和符号链接有意义。

如上结构体中的st\_blksize和st\_blocks有其特殊含义：

* st\_blksize表示在当前的系统中，最优的块大小，但文件系统并不一定按照此设置；
* st\_blocks表示文件占用了多少个块，一般而言，现在的linux的文件系统，一个块的size是512字节；

文件占用的磁盘空间，是要按照块大小为依据的。即便一个文件不到512字节，分配的时候也要以块为基本单位对其进行分配，所以du得到的size如果以字节为单位，得到的是所有文件的真实bytes值；而如果以KiloBytes为单位，得到的就是块大小的值。

### 2.1.1.du的简易实现版本

linux系统提供了“du”命令，用来查看文件、目录所占用的磁盘空间大小，这里实现了一个读取文件夹所占磁盘空间大小的程序。



有如下几个技术要点需要明确：

* gDirPathList是一个单向链表，是主要的数据结构，内部节点是传入的目录下的所有字目录的绝对路径；
* 如果要取的是目录所占的字节数，那么取的是各个文件的st\_size选项；
* 如果要取的是目录所占的KiloBytes(千字节数)，那么取的是各个文件的st\_blocks选项；

使用“du”命令查看一个目录的大小，会发现按照字节数获取到的size和按照KiloBytes获取到的size，并不能按照1024倍的关系简单的整除。

造成这个的原因在于，读取字节数的时候，是按照每个文件的size来的，累加这个st\_size选型；而按照KiloBytes读的时候，是按照所占用的磁盘块来读的。

磁盘空间的大小，在struct stat中是st\_blocks(所占用的磁盘块)指定了该值，在一般的linux系统中，每一个块的大小是512字节，也就是说st\_blocks的单位是512字节，而所要求的是KiloBytes，所以这里还需要做一次单位变换。

## 2.2.chmod、fchmod

这两个函数可以使得我们更改现有文件的访问权限：

#include <sys/stat.h>

int chmod(const char \*pathname, mode\_t mode);

int fchmod(int fields, mode\_t mode);

执行成功返回0，出错返回-1；

为了改变一个文件的权限位，进程的有效用户ID必须等于文件的所有者ID，或者该进程必须具有超级用户权限。

其mode常量的取值支持如下所示宏的与或操作：

|  |  |
| --- | --- |
| **Mode** | **说明** |
| S\_ISUID | 执行时设置用户ID |
| S\_ISGID | 执行时设置组ID |
| S\_ISVTX | 保存正文(粘住位) |
| S\_IRWXU | 用户(所有者)读、写、执行 |
| S\_IRUSR | 用户(所有者)读 |
| S\_IWUSR | 用户(所有者)写 |
| S\_IXUSR | 用户(所有者)执行 |
| S\_IRWXG | 组读写执行 |
| S\_IRGRP | 组读 |
| S\_IWGRP | 组写 |
| S\_IXGRP | 组执行 |
| S\_IRWXO | 其他读写执行 |
| S\_IROTH | 其他读 |
| S\_IWOTH | 其他写 |
| S\_IXOTH | 其他执行 |

## 2.3.truncate、ftruncate

如果是要将文件清零，那么打开文件的时候使用“w”模式可以做到这一点，但很多时候需要对一个文件的末尾处截去一些数据以缩短文件，那么就需要使用到如下函数：

#include <unistd.h>

int truncate(const char \*filepath, off\_t length);

int ftruncate(int fileds, off\_t length);

这两个函数将指定文件截短道length个字节，如果原来的文件size大于length，那么多出来的部分将无法再被访问；如果小于length，那么各个操作系统的处理方式可能并不相同，遵循XSI的系统将增加该文件的长度至length。

## 2.4.link、unlink、remove、rename

为一个已有文件创建连接：

#include <unistd.h>

int link(const char \*existingpath, const char \*newpath);

该函数执行成功返回0，执行失败返回-1。针对该函数，需要注意如下几点：

1. 如果源文件不存在，执行出错；
2. 如果newpath的上级目录不存在，执行出错；
3. 如果传入的是目录，执行出错；
4. 如果newpath已经存在了，执行出错；
5. 该函数创建的是**硬链接**；

删除一个文件(或符号链接)可以使用：

#include <unistd.h>

int unlink(const char \*pathname);

该函数执行成功返回0，执行失败返回-1.针对该函数，需注意：

1. 如果源文件不存在，执行出错；
2. 如果源文件是符号链接， 只删除该符号链接，符号链接所指向的文件依然存在；
3. 超级用户可以调用unlink的时候传入一个目录，这样可以删除一个目录，但更通常来说，使用rmdir是对目录操作的首选，而不使用unlink；

ISO C指定使用remove函数删除一个文件，而不是UNIX历来使用的unlink，因为很多非UNIX系统并不支持文件链接：

#include <stdio.h>

int remove(const char \*pathname);

对于文件来说，remove等同于unlink；对于目录来说，remove等同于rmdir。

使用rename函数可以给文件或目录更换名字：

#include <stdio.h>

int rename(const char \*oldname, const char \*newname);

1. 如果oldname是一个文件，那为该文件或符号链接更名。如果newname已经存在，而且是一个目录，那么操作将失败；如果newname已经存在，而且是一个文件，那么rename函数将删除原来的newname文件，并将oldname复制为新的newname文件；
2. 如果oldname是一个目录，那为该目录更名。如果newname是一个目录，那么只有在newname是一个空目录的情况(目录下只包含.和..)下才能执行成功。这种情况下，rename函数将删除原来的newname目录，并更名。
3. 如果是位目录更名，newname不能以oldname为前缀。例如，不能将/usr/foo改为/usr/foo/function；
4. 如果oldname是符号链接，那么rename只针对符号链接做处理，而对文件本身并不做处理；
5. 作为一个特例，如果oldname和newname是同一个文件，函数不做任何改变直接返回正确返回值。

## 2.5.mkdir、rmdir

#include <sys/stat.h>

int mkdir(const char \*path, mode\_t mode);

返回0表示创建成功，返回-1说明创建失败；

此函数创建一个空目录，默认创建除“.”和“..”目录项。所指定的目录访问权限由进程的文件模式创建屏蔽字修改。

但需要注意，对于目录来说，仅仅有读写权限是不够的，最少需要设置一个可执行的权限，以允许访问目录中的文件。

执行mkdir操作时，其上级目录必须存在，否则函数执行出错。

mode代表了新创建目录的权限，其取值与chmod的mode相同。

#include <unistd.h>

int rmdir(const char \*pathname);

删除一个**空目录**，所谓空目录，指的是只包含“.”和“..”的目录。

如果指定的目录不为空，那么函数将返回错误。

## 2.6.读目录

对某个目录具有访问权限的任何一个用户都可以读该目录，但为了防止文件系统发生混乱，只有内核才能写目录。

#include <dirent.h>

DIR \*opendir(const char \*pathname); /\*Input dir path, return DIR pointer\*/

struct dirent \*readdir(DIR \*dp); /\*read it\*/

void rewinddir(DIR \*dp);

int closedir(DIR \*dp); /\*release resource\*/

long telldir(DIR \*dp);

void seekdir(DIR \*dp, long loc);

所使用到的结构体dirent的定义：

struct dirent

{

ino\_t d\_ino;

char d\_name[NAME\_MAX + 1];

}

其中，d\_name属性指定了文件名字。

### 2.6.1.opendir的错误返回值

函数opendir执行错误时，返回NULL，可以通过获取系统的errno得知出错的原因。

#include <error.h>

printf(“%d, %s”, errno, strerr(errno));

|  |  |
| --- | --- |
| Errno | Desc |
| EACCESS | 权限不足 |
| EMFILE | 达到了一个进程可以打开的文件数的上限 |
| ENFILE | 达到了系统中可以同时打开的文件数的上限 |
| ENOTDIR | 传入的path并不是一个目录 |
| ENOENT | 传入的path不存在，或path是个NULL值 |
| ENOMEM | 核心内存不足 |

## 2.7.getcwd

#include <unistd.h>

char \*getcwd(char \*buf, size\_t size);

该函数执行成功，返回buf，否则返回NULL。

该函数执行成功能够得到当前的绝对路径，传入的buf将存储绝对路径值，size指定了buf的大小，以字节为单位。size如果不足以存储绝对路径名加一个NULL终止符，那么将返回出错。

# 3.线程(第11章)

每个进程有单独的进程ID，类似的，每个线程也有单独的线程ID。

进程ID在这个系统中，是唯一的；但线程ID不同，只在其所属的进程环境中唯一。

## 3.1.创建线程

#include <pthread.h>

int pthread\_create(pthread\_t \*restrict tidp, const pthread\_attr\_t \*restrict attr, void \*(\*start\_rtn)(void), void \*restrict arg);

返回值：创建成功返回0，否则返回错误编号；

当函数执行成功时，由tidp指向的内存单元被设置成新创建线程的线程id。attr指定不同的线程属性，一般设置成NULL。新线程将从start\_rtn函数的地址开始运行，该函数只有一个无类型指针参数arg，如果需要向start\_rtn传递的参数不止一个，需要将参数放到一个结构中后，将这个结构的地址作为arg参数传入。

## 3.2.线程终止

如果进程中任意一个线程调用了exit，\_Exit或者\_exit，整个进程都会终止执行。

在不终止整个进程的情况下停止线程，有如下三种方式：

* 线程只是从启动例程中返回，返回值是线程的退出码；
* 线程被同一个进程中的其他线程取消；
* 线程自身调用pthread\_exit函数；

#include <pthread.h>

int pthread\_join(pthread\_t th\_id, void \*\*rval\_ptr);

调用该函数的线程将一直阻塞，直到指定线程(线程id为th\_id)调用pthread\_exit、从启动例程中返回或被取消。

如果对线程的返回值不感兴趣，可以直接将rval\_ptr设置为NULL。

void pthread\_exit(void \*rval\_ptr);

调用该函数可以终止自身线程，并设置线程返回值为rval\_ptr，例如：pthread\_exit((void \*)2)，就结束线程，返回值为2.

该返回值可以通过上面介绍的pthread\_join()获得，例如：void \*th\_ret;pthread\_join(th\_id, &th\_ret);th\_ret的值通过强制类型转换后可以查看应该是2.

#include <pthread.h>

int pthread\_cancel(pthread\_t tid);

在默认的情况下，pthread\_cancel函数会使tid标识的线程，如同调用了参数为PTHREAD\_CANCELED的pthread\_exit函数。但是，该线程(tid)可以选择忽略取消方式或是控制取消方式。而且，pthread\_cancel并不等待线程终止，只是提出请求。

**//TODO，pthread\_cleanup\_push,pthread\_cleanup\_pop**

## 3.3.线程标识

每个线程都有一个自己的id，但不同于进程的id在系统中是唯一的，线程的id只在所属的进程环境中有效。

#include <pthread.h>

pthread\_t pthread\_self(void);

该函数可以获得当前线程的id，返回值为pthread\_t类型。该类型在各个操作系统中是不同的， 所以不能简单实用“==”判断两个线程的id是否相同，需要使用如下函数：

int pthread\_equal(pthread\_t th\_id1, pthread\_t th\_id2);

如果两者相同就返回非0值，否则返回0.

## 3.4.线程同步

### 3.4.1.互斥锁

互斥变量用pthread\_mutex\_t数据类型表示，其初始化函数：

#include <pthread.h>

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrict attr);

要用默认的属性初始化互斥量，只需要将attr设置成NULL即可。

对互斥量进行加锁操作，需要调用pthread\_mutex\_lock函数，如果互斥量已经加锁了，调用线程将被阻塞，直到互斥量解锁：

#include <pthread.h>

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

如果线程不希望被阻塞，可以调用pthread\_mutex\_trylock函数。该函数调用时，如果互斥量没有锁定，那么线程将锁定互斥量并返回0；如果已经被锁定了，该函数将直接返回EBUSY，不会阻塞等待。

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

**避免死锁：避免出现两个线程互相需要对方的锁定资源的情况！**

### 3.4.2.读写锁

读写锁与互斥量很类似，但拥有更高的并行性。互斥量只有占有和等待两个状态，读写锁有三种：读模式下加锁；写模式下加锁；不加锁。一次只能有一个线程占有写模式的读写锁，但是多个线程可以同时占有读模式的读写锁。

读写锁非常适合于对数据结构读的次数远远大于写的次数的情况，读写锁的初始化：

#include <pthread.h>

int pthread\_rwlock\_init(pthread\_rwlock\_t \*restrict rwlock, const pthread\_rwlockattr\_t \*restrict attr);

如果执行成功返回0，否则返回错误编号。如果不希望自己指定读写锁的属性，可以将attr属性设置为NULL。

在释放读写锁占用的内存前，需要做清理工作：

int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock);

该函数将清理在init时分配给读写锁的资源，读写锁的使用：

int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*rdlock); //读模式下加锁

int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \*wrlock); //写模式下加锁

int pthread\_rwlock\_unlock(pthread\_rwlock\_t \*rwlock); //释放锁

这三个函数返回值为0说明执行成功，否则返回错误编号。

与上面讲的互斥锁类似，读写锁同样有如下函数实现有条件的读写：

int pthread\_rwlock\_tryrdlock(pthread\_rwlock\_t \*rdlock);

int pthread\_rwlock\_trywrlock(pthread\_rwlock\_t \*wrlock);

两者的返回值都是：成功返回0，失败返回错误编号。

### 3.4.3.条件变量

条件变量给多个线程提供了一个会和的场所，条件变量和互斥量一起使用时，允许线程以无竞争的方式等待特定的的条件发生。

条件变量本身是由互斥量保护的。线程在试图改变条件变量之前必须首先锁住互斥量，其他线程在获取互斥量之前不会察觉到这种变化，因此必须锁定互斥量之后才能计算条件。

条件变量在使用之前必须初始化，有两种方式：

1. 使用常量PTHREAD\_COND\_INITIALIZER赋值给静态分配的条件变量；
2. pthread\_cond\_init函数动态分配条件变量；

动态初始化和销毁条件变量的函数原型如下：

#include <pthread.h>

int pthread\_cond\_init(pthread\_cond\_t \*restrict cond, pthread\_condattr\_t \*restrict attr);

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

使用如下函数等待条件变量变为真，且可以控制timeout，如果在指定的timeout之内没有条件没有满足将给出一个代表出错码的返回变量：

int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex);

int pthread\_cond\_timedwait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex, const struct timespec \*restrict timeout);

传递给pthread\_cond\_wait的互斥量对条件变量进行保护，调用者把锁住的互斥量传递给函数。函数把调用线程放到等待条件的线程列表上，然后对互斥量解锁，这两个操作是原子操作。这样就关闭了条件检查和线程进入休眠状态等待条件改变这两个操作之间的时间通道，这样线程就不会错过条件的任何变化。pthread\_cond\_wait返回时，互斥量再次被锁住。

struct timespec

{  
 time\_t tv\_sec; //seconds

long tv\_nsec; //nano seconds

}

使用这个结构后，可以指定等待最多多长时间，这个时间是一个绝对数而不是相对数。例如，如果愿意等待最多3分钟，那么就需要首先获取当前时间，再在其基础上加3分钟后，转换为timespec结构，而不是直接将3分钟转换为timespec结构。示例代码如下：

void maketimeout(struct timespec \*tsp, long minutes)

{

struct timeval now;

gettimeofday(&now);

tsp->tv\_sec = now.tv\_sec;

tsp->tv\_nsec = now.tv\_usec \* 1000;

tsp->tv\_sec += miniutes \* 60;

}

如果到了时间条件变量还没有出现，pthread\_cond\_timedwait将重新获取互斥量并返回错误值ETIMEDOUT。

从pthread\_cond\_wait或pthread\_cond\_timedwait调用成功返回时，线程需要重新计算条件，因为此时其他的线程可能已经将条件进行了改变。

有两个函数可以用于通知线程条件已经满足：pthread\_cond\_signal函数唤醒等待该条件的某个线程；pthread\_cond\_broadcast函数唤醒等待该条件的所有线程。函数原型如下：

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

调用这两个函数，一定要注意一定要在改变条件状态之后再给线程发送信号！

## 3.5.线程池

moSdk—moUtils—ThreadPool；

# 4.线程属性

# 4.网络IPC：套接字(第16章)

## 4.1.套接字描述符创建与销毁

要创建一个套接字，使用函数如下：

#include <sys/socket.h>

#include <sys/types.h>

int socket(int domain, int type, int protocol);

函数执行成功，返回套接字描述符；否则返回-1。各参数含义及取值参见下表：

|  |  |  |  |
| --- | --- | --- | --- |
| domain，域 | 描述 | type，类型 | 描述 |
| AF\_INET | ipv4因特网域 | SOCK\_DGRAM | 长度固定的、无连接的不可靠报文传递 |
| AF\_INET6 | ipv6因特网域 | SOCK\_RAW | IP协议的数据报接口 |
| AF\_UNIX | unix域 | SOCK\_SEQPACKET | 长度固定、有序、可靠的面向连接报文传递 |
| AF\_UNSPEC | 未指定 | SOCK\_STREAM | 有序、可靠、双向的面向连接字节流 |

参数protocol通常是0，表示按照给定的域和套接字类型选择默认协议。也可以自行定义该数值，选择一个特定协议。AF\_INET+SOCK\_STREAM的默认协议是TCP(传输控制协议)；AF\_INET+SOCK\_DGRAM默认协议是UDP(用户数据报协议)。

**数据报**协议提供的是无连接的服务，与对方通信时不需要逻辑连接，只需要送出一个报文，其地址是对方进程所使用的套接字；**字节流**要求在交换数据之前，在本地套接字和远程套接字之间建立一个逻辑连接。

套接字使用完毕，无需再次使用时，可使用**close()函数**将该套接字描述符关闭。该函数关闭该描述符后，该描述符可以被系统再次分配给其他需要的进程。

套接字通信是双向的，可以使用如下函数禁止套接字上的输入/输出：

#include <sys/socket.h>

int shutdown(int sockfd, int how);

how的取值及含义如下表所示：

|  |  |
| --- | --- |
| 参数how取值 | 含义 |
| SHUT\_RD | 无法从套接字读取数据 |
| SHUT\_WR | 无法使用套接字发送数据 |
| SHUT\_RDWR | 同时无法使用套接字读取和发送数据 |

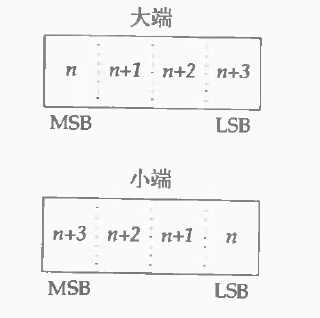
能够执行close，但还是提供了shutdown函数来关闭套接字的原因如下：

1. close只有在最后一个活动被关闭时才释放网络节点。这就是说如果一个套接字被复制了(使用dup)，那么这个套接字将直到最后一个引用它的文件描述符被关闭了，它才会被关闭。而shutdown则允许使一个套接字处于不活动状态，不论引用它的文件描述符有多少；
2. 有时候只关闭套接字双向传输中的一个方向会很方便。例如，如果想让所通信的进程能够确定数据发送何时结束，可以关闭该套接字的写端，然后通过该套接字的读端让就可以继续接受数据。

## 4.2.寻址

要做到通信，最少要知道如何确定一个目标进程。网络通信进程的标识分为两部分：计算机的网络地址；本机服务编号。

### 4.2.1.字节序



如果处理器架构支持大端(big-endian)字节序，那么最大字节地址对应于数字最低有效字节(LSB)上；小端(little-endian)则相反，数字最低字节对应于最小字节地址。

对于TCP/IP应用程序，提供了四个通用函数以实施在处理器字节序和网络字节序之间的转换。

#include <arpa/inet.h>

uint32\_t htonl(uint32\_t hostint32); //返回以网络字节序表示的32位整形数；

uint16\_t htons(uint16\_t hostint16);//返回以网络字节序表示的16位整形数；

uint32\_t ntohl(uint32\_t netint32); //返回以主机字节序表示的32位整形数；

uint16\_t ntohs(uint16\_t netint16); //返回以主机字节序表示的16位整形数；

这里，“h”表示host(主机)，“n”表示net(网络)，l表示long(4个字节)，s表示short(2个字节)。

### 4.2.2.地址格式

地址标示了特定通信域中的套接字端点，地址格式与特定的通信域有关。为了使不同的地址能够被传入到套接字函数，地址被强制转换成通用的地址结构：

struct sockaddr

{

sa\_family\_t sa\_family;

char sa\_data[];

…

};

套接字实现可以自由的添加额外的成员并且定义sa\_data的size，例如在linux中，该结构被定义为：

struct sockaddr

{

sa\_family\_t sa\_family;

char sa\_data[14];

};

针对IPv4协议，linux系统对struct sockaddr采用了如下实现：

struct sockaddr\_in {

short int sin\_family; /\*address family\*/

unsigned short int sin\_port; /\*port number\*/

struct in\_addr sin\_addr; /\*internet address\*/

unsigned char sin\_zero[8]; /\*same size as struct sockaddr\*/

};

typedef struct in\_addr {

unsigned long s\_addr;

};

其中**sin\_zero必须被设置为0**；

sin\_family参数代表了协议族，例如AF\_INET代表了IPv4。简单的使用示例如下：

struct sockaddr\_in sock;

bzero((char \*)&sock, sizeof(sock));

sock.sa\_family = AF\_INET;

sock.sin\_port = htons(1234);

sock.sin\_addr.s\_addr = inet\_addr(“192.168.6.132”);

使用inet\_addr()函数将形如“192.168.6.132”的IP地址转化为无符号整形：

ina.sin\_addr.s\_addr = inet\_addr(“192.168.6.132”);

该函数如果执行过程中出现错误，将返回-1。

使用inet\_ntoa()可以将地址转化成数字和句点的形式：

printf(“%s”, inet\_ntoa(ina.sin\_addr));

这样就可以打印出IP地址，该函数返回的是一个指向字符串的指针。

### 4.2.3.地址查询

//Nothing

### 4.2.4.将套接字与地址绑定

#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t len);

本函数执行成功返回0，否则返回-1。需要注意的是，端口号必须大于1024，除非该进程具有相应的特权(超级用户)。客户端的套接字进行绑定意义不大，但对于服务器来说，需要绑定一个特定的地址，以周知所有客户端。

对于因特网域，如果指定IP地址是**INADDR\_ANY**，套接字端点可以被绑定到所有的系统网络接口。这意味着可以收到这个系统所安装的所有网卡的数据包。

如果调用connect或listen之前，没有执行bind函数，系统会选择一个地址并将其绑定到socket上。

#include <sys/socket.h>

int getsockname(int sockfd, struct sockaddr \*restrict addr, socklen\_t \*restrict alenp);

该函数用来发现绑定到一个套接字的地址，成功返回0，否则返回-1。

在调用该函数之前，将alenp设置成一个指向整数的指针，这个整数将指定缓冲区sockaddr的大小。返回时，这个整数将被设置成返回地址的大小。如果地址与提供的缓冲区长度不匹配，将其截断但不报错！如果当前没有绑定到该套接字的地址，结果没有明确定义。

#include <sys/socket.h>

int getpeername(int sockfd, struct sockaddr \*restrict addr, socklen\_t \*restrict alenp);

如果套接字已经和对方连接，该函数可以找到对方的地址。除了会返回对方的地址外，其他返回与getsockname相同。

## 4.3.建立连接

如果处理的是面向连接的网络服务(SOCK\_STREAM或SOCK\_SEQPACKET)，在开始交换数据之前，需要在请求服务的进程套接字(客户端)和提供服务的进程套接字(服务器)之间建立一个连接。建立函数：

#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t len);

本函数执行成功返回0， 执行失败返回-1。

在连接服务器时，出于一些原因连接很可能失败。例如，要连接的机器必须开启并且正在运行，服务器必须绑定到一个想与之连接的地址，在服务器的等待连接队列中需要有足够的空间。因此，应用程序必须能够处理connect函数返回的错误，这些错误可能是一些瞬时变化引起的。

#include <sys/socket.h>

int listen(int sockfd, int backlog);

服务器通过调用listen函数来宣告可以接收连接请求，函数执行成功返回0，失败返回-1。参数backlog提供一个提示，用于表示该进程所要入队的连接请求数量，实际值由操作系统决定。

对于TCP协议，backlog的最大值默认是128。一旦队列满，系统将拒绝多余的连接请求。一旦服务器开启了listen，套接字就可以接收连接请求，使用如下函数：

#include <sys/socket.h>

int accept(int sockfd, struct sockaddr \*restrict addr, socklen\_t \*restrict len);

该函数返回值是套接字描述符，这个描述符连接到调用connect的客户端。这个新的套接字描述符和原始套接字(sockfd)具有相同的套接字类型和地址族。传给accept的原始套接字没有关联到这个客户端，而是继续保持可用状态并接受其他连接请求。

如果不关心客户端标识，可以将参数addr和len都设置成NULL。否则，在调用该函数前，需要将addr设置有足够大的缓冲区来存放地址，将len设置成指向代表这个缓冲区大小的整数的指针。在函数返回时，将会自动填充缓冲区并更新len的值。

如果没有连接请求等待处理，accept会阻塞直到一个请求到来。如果sockfd处于非阻塞模式，accept将会返回-1并将errno设置成EAGAIN或EWOULDBLOCK。

## 4.4.数据传输

以下所列函数，头文件为：<sys/socket.h>；

### 4.4.1.send和recv函数

ssize\_t send(int sockfd, const void \*msg, size\_t nbytes, int flags);

该函数执行成功返回发送的字节数，失败-1。第一个参数是目的套接字描述符，是准备发送到的地址；第二个参数是希望发送的数据的指针；第三个是数据的字节长度；flags一般直接置零。

即便send成功返回，也并不就代表连接另一端的进程接收了数据，而仅仅是表示数据已经没有错误的发送到了网络上。如果返回的字节数小于实际想要发的字节数，必须手动发送剩下的数据。

**使用send时必须已经建立了连接。**

ssize\_t recv(int sockfd, void \*buf, int len, unsigned int flags);

第一个参数是要读取的套接字描述符；第二个参数是保存读取的信息的地址；第三个参数是缓冲区的最大长度；第四个参数一般直接设置为0。

系统调用recv函数返回实际读取到的缓冲区的字节数，函数执行过程中出现错误返回-1。

### 4.4.2.sendto和recvfrom

由于使用UDP协议时，并不与远程主机/服务器进行连接，所以在发送数据包的时候，需要指定目标地址：

ssize\_t sendto(int sockfd, const void \*buf, size\_t nbytes, int flags, const struct sockaddr \*destaddr, socklen\_t destlen);

该函数执行成功同样返回发送的字节数，失败返回-1.

除了后面两个参数，其他参数与send函数相同含义。destaddr参数是包含了目的IP地址和端口号的数据接口的指针；参数destlen可以取值为sizeof(struct sockaddr)。

**使用sendto允许在无连接的套接字上指定一个目标地址。**

ssize\_t recvfrom(int sockfd, void \*buf, int len, unsigned int flags, struct sockaddr \*from, int \*fromlen);

参数from是指向包含源IP和端口号的指针，fromlen的值同样可以取为sizeof(struct sockaddr)。该函数执行成功返回接受到的字节数，失败返回-1.

### 4.4.3.sendmsg和recvmsg

ssize\_t sendmsg(int sockfd, const struct msghdr \*msg, int flags);

该函数可以通过将数据组织在struct msghdr中实现信息发送的目标。POSIX.1定义了该结构，至少应包括如下成员：

struct msghdr

{

void \*msg\_name;

socklen\_t msg\_namelen;

struct iovec \*msg\_iov;

int msg\_iovlen;

void \*msg\_control;

socklen\_t msg\_controllen;

int msg\_flags;

};

ssize\_t recvmsg(int sockfd, struct msghdr \*msg, int flags);

# 5.进程控制(第8章)

## 5.1.进程标识符

每个进程都有一个非负型整数表示其唯一进程ID，系统中有一些专用的进程：

* ID为0的进程通常是调度进程，常被称为交换进程(swapper)。该进程是内核的一部分，并不执行任何磁盘上的程序，因此也被称作是系统进程。
* ID为1的进程是init进程，在自举过程结束后由内核调用，负责启动一个UNIX系统。

#include <unistd.h>

pid\_t getpid(void); //返回值：调用进程的进程ID；

pid\_t getppid(void); //返回值：调用进程的父进程ID；

uid\_t getuid(void); //返回值：调用进程的实际用户ID；

uid\_t geteuid(void); //返回值：调用进程的有效用户ID；

gid\_t getgid(void); //返回值：调用进程的实际组ID；

gid\_t getegid(void); //返回值：调用进程的有效组ID；

## 5.2.fork函数

一个现有进程可以通过调用fork函数创建一个新进程，函数原型如下：

#include <unistd.h>

pid\_t fork(void);

fork函数被调用一次，但返回两次：子进程的返回值是0，而父进程的返回值是子进程的PID。

子进程PID会返回给父进程，是因为一个父进程可能有多个子进程，而且没有提供一个函数可以获得所有子进程的PID，所以父进程可能需要记录子进程的PID。

子进程自己得到的返回值是0，是因为一个进程只可能有一个父进程，而且子进程可以通过getpid获取到父进程的PID，而0代表了内核交换进程的PID，不可能给任何子进程自己的PID，所以可以用来做标记。

子进程是父进程的副本。例如，子进程获得父进程数据空间、堆和栈的副本。父子进程并不共享这部分资源，而是子进程复制一份作为自己使用。父子进程之间共享**正文段**。

一般来说，在fork之后，父子进程谁先执行并不确定，这取决于内核所使用的调度算法。

fork失败的两个主要原因是：

1. 系统中已经有了太多的进程；
2. 该实际用户ID的进程总数超过了系统设置；

## 5.3.vfork函数

#include <unistd.h>

pid\_t vfork(void);

与fork类似，vfork同样也创建一个子进程，但它并不将父进程的地址空间完全复制到子进程中，因为子进程会立即调用exec(或exit)，于是也就不会访问父进程的地址空间。但是，在子进程调用exec或exit之前，它是在父进程的空间中运行的。

vfork和fork的另一个区别是，vfork保证子进程先运行，在它调用exec或exit之后父进程才可能被调度。

注意：如果在调用exec或exit之前，子进程依赖于父进程的进一步动作，那么会发生**死锁**。

## 5.4.进程终止

如果父进程在进程之前终止了，那么子进程的父进程将变为init进程。这个过程称之为**init进程领养**。其操作过程如下：当一个进程终止时，内核逐个检查所有活动进程，以判断它是否是正要终止的进程的子进程，如果是，更改其父进程的ID到1。这样可以保证所有的进程都存在一个父进程。

如果子进程在父进程之前终止，父进程如何得到其终止状态：内核为每个终止进程保存了一定量的信息，所以当终止进程的父进程调用wait或者waitpid时，可以得到这些信息。这些信息至少包含了进程ID、终止状态、子进程占用的CPU时间总量。内核可以释放终止进程所使用所有存储区，关闭其所有打开文件。

在UNIX术语中，一个已经终止、但是父进程尚未对其进行善后处理(获取终止子进程的有关信息、释放其资源)的进程，成为僵死进程(zombie)。使用ps命令查看时，该类进程状态是Z。

## 5.5.wait和waitpid函数

当一个进程正常或异常终止时，内核向其父进程发送SIGCHLD信号。因为子进程终止是异步事件(可以在父进程运行的任何时刻发生)，所以这种信号也是内核向父进程发起的异步通知。父进程可以选择忽略该信号，也可以提供一个在信号到达时执行的函数(信号处理函数)。对于这种信号，系统的默认动作是忽略它。

#include <sys/wait.h>

pid\_t wait(int \*statloc);

pid\_t waitpid(pid\_t pid, int \*statloc, int options);

这两个函数执行成功则返回进程ID或0，出错返回-1。这两个函数的区别如下：

* 在一个子进程终止前，wait使调用者阻塞，而waitpid有一个选项，可以使调用者不阻塞；
* waitpid并不等待在其调用之后的第一个终止子进程，它有若干个选项，可以控制它所等待的进程；

调用**wait**的进程可能发生的情况如下：

* 如果其所有子进程都还在运行，那么该进程将阻塞；
* 如果其中一个子进程终止，那么父进程可以获取到其终止状态，且取得该子进程的终止状态后立即返回，**父进程将不再理会其他子进程的终止状态**；
* 如果该进程没有任何子进程，立即返回错误；

因为函数返回的是子进程的ID，所以它总是能够知道是哪一个子进程终止了。

这两个函数中的参数statloc是一个整形指针。如果它不是一个空指针，那么终止进程的终止状态，就存放在它所指向的内存单元内。如果并不关心终止状态，这个参数赋空指针即可。

当子进程正常终止时，可以通过 WIFEXISTED(status) 宏来检查终止状态，具体如下：如果子进程正常终止，那么WIFEXISTED返回为真。这种情况下，可以执行WEXITSTATUS(status)，取得子进程传送给exit、\_exit或\_Exit的低8位的值，使用示例：

int status;

waitpid(&status);

if(WIFEXISTED(status)) WEXITSTATUS(status);

## 5.6.exec函数

当进程调用一种exec函数时，该进程执行的程序完全替换为新程序，而新程序则从其main函数开始执行。因为调用exec函数并不创建新进程，所以进程ID不发生改变。调用exec函数，只是用一个全新的程序替换了当前进程的正文段、数据段、堆和栈。

#include <unistd.h>

int execl(const char \*pathname, const char \*arg0, … /\* (char \*)0 \*/);

int execv(const char \*pathname, char \*const argv[]);

int execle(const char \*pathname, const char \*arg0, … /\* (char \*)0, char const envp[] \*/);

int execve(const char \*pathname, char \*const argv[], char \*const envp[]);

int execlp(const char \*filename, const char \*arg0, … /\* (char \*)0 \*/);

int execvp(const char \*filename, char \*const argv[]);

以上六个函数只有在执行出错的时候返回-1，**执行成功时不返回值**；

### 5.6.1.路径名和文件名

以上6个函数的第一个区别是：前四个函数传入的参数pathname是路径名，而后两个函数传入的filename是文件名。当指定filename作为参数时：

* 如果filename中带有“/”，则将其视为路径名；
* 否则就按照PATH环境变量，在它所指定的各个目录中搜寻可执行文件；

如果execlp或者execvp使用PATH中的目录找到了一个同名文件，但是这个文件不是由连接编辑器产生的机器可执行二进制文件，就认为这个文件是shell脚本，将尝试调用/bin/sh对其执行。

### 5.6.2.参数传递方式

以上6个函数的第二个区别是：参数传递方式不同(l表示list，v表示vector)。

函数execl、execlp、execle要求将新程序的每个命令行参数都说明为一个单独的参数。这个参数表以空指针结尾。另外三个函数(execv、execve、execvp)，需要首先构造一个指向各个参数的指针数组，然后将该数组地址作为这三个函数的参数。

execl、execlp、execle三个函数表示命令行参数的一般方法是：

char \*arg0, char \*arg1, ……, char \*argn, (char \*)0

最后的空指针传入是有意义的。如果用常数0来表示一个空指针，那么必须将它强制转换为字符指针，否则它会被解释成整形参数。如果一个整形数的长度与char \*的长度不同，那么exec函数的实际参数就会出错。

### 5.6.3.传递环境表

以e结尾的两个函数(execle、execve)可以传递一个指向环境字符串指针数组的指针。其他四个函数将使用调用进程的environ变量为新程序复制现有的环境，而这两个函数可以指定一个特殊的环境给新进程。

在很多unix实现中，只有execve是内核的系统调用。其他5个只是库函数，最终都要调用这个系统调用以实现功能。其关系如下：



## 5.7.system函数

#include <stdlib.h>

int system(const char \*cmdstring);

因为system函数在其实现中调用了fork、exec和waitpid，因此有三种返回值：

* 如果fork失败或者waitpid返回除EINTR之外的错误，那么system返回-1，而且errno中设置了错误类型值；
* 如果exec失败(不能执行shell)，那么返回值如同shell执行了exit(127)一样；
* 否则，fork、exec、waitpid都已经执行成功，这时候system的返回值是shell的终止状态，其格式与waitpid相同；

Markdown

# 1.Markdown语法

# 1.1.标题

Mardown的标题，只需要在一段文字前面加 # 即可，例如:

# 一级标题

## 二级标题

### 三级标题

……

有两点注意：

* 共支持六级标题；
* #后面应紧跟一个空格，标准的markdown语法要求如此，虽然很多markdown解释器也支持没有空格的解析，但最好还是加上空格；

## 1.2.列表

### 1.2.1.无序列表

在一行的前面加 – 或者 \* ，可以将一行变为无序列表的一行；

注意，需要在 – 或者 \* 后面紧跟一个空格；

### 1.2.2.有序列表

在一行的前面以 数字+英文句点+空格 的格式标识，可以将一行变为有序列表的一行；

注意，空格不可少；

## 1.3.引用

如果要引用一段别人的句子(名人名言之类)，可以在段落的最前面以 > 标识;

注意，引用标识 > 后面同样需要跟一个空格；

## 1.4.图片与链接

### 1.4.1.图片

插入图片的格式为：![ImageDescription](ImageUrl “title”)

title是可选项，可以没有该项内容.

ImageUrl是一个url，需要图床，这里使用百度云盘作为图床了：

1. 登录百度云盘(手机号为账号)；
2. 上传图片；
3. 选择图片，并点击下载；
4. 下载完成后，右键--复制下载地址，得到的url可用；

注意，edge已经不能执行step4了，所以要用ie来执行；

#### 裁剪图片

对于上边的这种图片方式，不能很好的对图片进行裁剪，因此找到了如下这种增加html语句的方式来进行图片展示的效果，这个方式也不需要图床，我觉得更加适合我的使用。

<div align="center">

<img src="./picture/1FirstSeeU.jpg" width="200" height="350" alt="FirstSeeU" />

</div>

第一行指定了图片居中；

第二行指定了图片的本地位置，宽高和名字；

### 1.4.2.链接

与插入图片的格式类似，区别在于最前面的 ! 符号：[UrlDescription](UrlValue “title”)

与图片类似，title也是可选的；

## 1.5.粗体与斜体

用两个 \* 包含的就是粗体；

用一个 \* 包含的就是斜体；

注意，这里的 \* 和文本之间，不应该存在空格；

## 1.6.表格

表格的示例如下：

| ColumName1 | ColumName2 | ColumName3 |  
|:---------- |:----------:| ----------:|  
| value1     | value2     | value3     |  
| value1     | value2     | value3     |

其中，第一行代表这个表格的各个列的名字；第二行是格式：左边有一个冒号说明是左对齐，右边有一个冒号说明是右对齐，两边都有冒号说明是中间对齐，都没有冒号是默认的左对齐；下面的行是内容；

## 1.7.分割线

要增加一条分割线，只需要 \*\*\* 即可；

# 2.Markdownpad2工具使用

## 2.1.默认不支持表格操作的解决办法

默认情况下，使用1.6中所述方法，并不能生成一个表格，这个是工具的问题，需要做如下操作：工具--选项--markdown，将解释器(processer)从markdown修改为markdown扩展(extra)，即可展示表格；

## 2.2.生成html文件

文件--导出，导出为html即可；

## 2.3.生成pdf文件

文件--导出，导出为pdf即可；

使用该工具生成的pdf文件，图片没有办法正常显示；因此可以如下这种：生成html文件，使用浏览器打开后，选择打印输出，选择打印机为pdf，即可；

Android

为了做moCloud项目的客户端，研究一下android 的简单实用。

# 1.开发环境搭建

从最熟悉的工具eclipse开始，搭建开发环境。过程极为艰辛，而且最终由于机器性能的原因，也并未最终完成！

安装步骤，主要有如下几个：

1. Eclipse安装；
2. Sdk安装；
3. Adt安装；
4. Eclipse配置；
5. Hello world运行；

## 1.1.eclipse安装

Eclipse选择了官网上的最新版本，并使用了java ee的版本；

Eclipse不需要安装，直接解压就可以运行，当然前提是jdk已经安装好了。

## 1.2.Sdk manager安装

网上下载了23版本的sdk manager，下载完成后，解压，目录下存在sdk manager.exe这个文件。

这是一个应用程序，用来更新android开发过程中需要用到的sdk。但是直接打开它，发现没有如网上所说的一大堆下拉框，也没有发现各个版本的sdk供下载使用。

原因是google的源被防火墙挡住了。

打开软件的“tools--optioins”选项卡，将：

* http proxy server更改为mirrors.neusoft.edu.cn
* Port设置为80
* 选中“force https to http”选项卡；

设置完之后，关闭sdk manager工具，再次打开，就可以发现自动出现了各个版本的可用的更新了。因为这个是修改为了东软的源，国内源可以访问。

下载了最新的6.0版本的API，和一些其他包。

## 1.3.adt安装

Sdk和eclipse都安装完了，但是两者还没有建立起联系来。

需要adt来建立这两者之间的联系。

直接下载一个对应sdk23版本的adt，eclipse中，help--install software选项卡里，增加一个ADT的本地文件选项，就可以将adt安装完成。

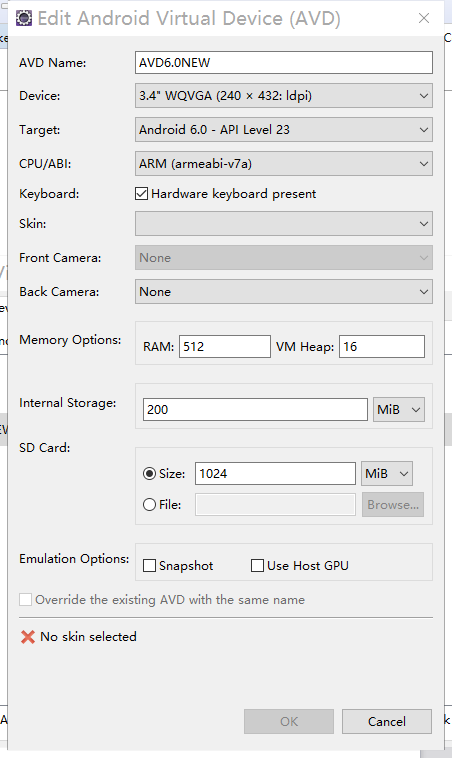
安装完成后，会提示重启eclipse，直接重启。

## 1.4.设置sdk manager和avd

重启eclipse之后，需要将sdk manager设置上，这样就可以用上android 的sdk进行代码开发了；还需要设置一个可用的avd，这样就可以使用模拟器了。

直接打开“windows--preference”，将sdk的路径设置进去，adt和sdk版本没有问题的话，会直接捕获到sdk的版本，并显示出来。Apply后退出就可以了。

再打开“windows--android virtual device manager”，设置一个模拟器。这里模拟器的设置参数，参考下图：



有几个需要注意的地方：

1. Device，是要选择的模拟器的分辨率，别选太大的，机器配置不好的情况下，显示器分辨率不高的情况下，选太大的要么整个屏幕都是一个模拟器，要么根本显示不出来模拟器；
2. Cpu/abi选择arm；
3. Target选择自己的api版本；

## 1.5.Hello world

Hello world程序，创建起来很简单。

File-new- androidproject，创建。

需要注意，创建的时候，会提示选择目标sdk版本和最低能支持的sdk版本，这里最好选择同样的版本，不要选择差距太大的，比如min选了2.2，target选了6.0，那么创建出来的项目，会各种sdk不识别的错误。

创建完成后，开启模拟器，我就是在这里没有成功。

因为模拟器根本拉不起来，机器配置太差。。。。