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。

Trustzone

# 1.helloworld程序

本程序下载自ARM官网，主要实现了在pc上通过DS-5开发工具、使用调试工具(JTAG)连接开发板(Versatile Express A9x4/VE-A9x4 FVP)后，调试实现trustzone技术的简单实现。

## 1.1.readme

通过程序中readme文件得知，该程序的运行平台要求是Versatile Express A9x4/VE-A9x4 FVP，应通过DS-5软件、使用调试的方式对其进行代码调试。

首先对其中的瀑布图进行修改后描述如下，以方便理解和阅读：



下面主要对代码执行过程部分进行记录(exploring the examp)：

下载程序到开发板时需要注意，包含了secure world的image和它的调试特征(debug symbols)、normal world的image文件的trustzone-versatile.axf文件将被下载到板子上，注意，该文件不包含normal world的debug symbols。在下载该文件的同时，还会将normal.axf文件下载到板子上，这里包含了normal world的debug symbols。

1. 程序的执行是从startup\_secure.s中的secureStart函数开始的。该程序的功能是初始化secure world，包括：设置svc模式的堆栈；使缓存(caches)和TLB(内存中的页表的cache)无效；设置页表(page tables)；使能MMU。执行这些之后就跳转到\_\_main里的C库运行时初始化部分。

secure state(安全态)需要处理器处于svc模式，可通过如下两种方式确认处理器是否处于svc：一是在寄存器列表中查看CPSR中的M位和CP15\_SCR中的NS位的值；二就是直接输出CLI中的这两个值进行查看。

1. 在b main\_secure.c::main处设置断点并F8运行到这里。这里必须设置断点，因为这个可执行文件中包含了两个main函数(secure、normal)。secure的main函数将在调用monitor.s中的monitorInit函数之前完成如下操作：设置缓存(caches)；设置TZPC；
2. 在语句b monitorInit处设置断点并F8运行至此。monitor的初始化将完成如下操作：安装(install)monitor的向量表(vector table)；初始化monitor mode的栈指针(stack point)；在secure world调用结束、返回前，设置虚拟的normal world的状态(state)，这个state将在第一次进入到normal world时使用；
3. 单步运行到CPS #Mode\_MON语句并执行。注意从SVC模式切换到MON模式时CPSR的值变化情况，反切换可以通过之后的几条语句实现；
4. 这时候secure world的main函数将再次被执行回来，它将打印“hello world from secure world”语句，然后通过yield函数、执行SMC指令，返回到monitor中；
5. 在yield函数里设置断点并单步运行以查看SMC指令。这将触发SMC异常，使程序到达monitor的向量表中的SMC入口处(SMC entry)。注意这里从SVC模式切换到MON模式相应的CPSR的变化。
6. 单步运行直到SMC\_Handler的末尾，在语句MOVS pc,lr执行前观察NS位的变化，一旦该语句执行，那么程序将跳转到normalStart语句(normal world)；
7. normalStart将对normal world执行初始化操作，包括：识别当前的CPU；将除CPU0之外的所有其他CPU休眠；设置堆栈；使caches和TLBs无效；设置页表；使能MMU；设置Vector Base Address Register(由于NS VBAR没有被设置值，所以这个是必须的)。然后跳转到\_\_main中C库运行时初始化程序；
8. 在语句b main\_normal.c::man设置断点并运行到这里。normal world的main函数将使能caches，并打印“hello world from normal world”，之后通过yield函数调用SMC指令切换到monitor状态中；
9. 在yield函数中设置断点并查看SMC指令的执行过程。这将再一次的触发SMC异常并指向monitor向量表中的SMC入口。观察从SVC到MON的CPSR的反映；
10. 执行到movs pc,lr语句，观察从MON到SVC的切换在CPSR上的反映。执行命令后将跳转到secure world的main函数；
11. secure world的main函数将再次打印“hello world from secure world”并通过调用yield函数再次切换到monitor中；
12. 至此，将进入到两个world的循环切换中……；

在一个真实的系统中，TZPC(trustzone protection controller)能够通过设置来确定 允许/禁止 安全/非安全 进入到内存和外设中。注意，TZPC本身只能在secure world中才能被访问，它依赖于安全域。

# 2.基于HelloWorld程序的功能添加

## 2.1.secure world是如何运行的

阅读了ARM公司的官方文档《ARM1176JZ-S Technical Reference Manual》中专门讲解trustzone技术是如何实现安全的章节2.2.2，本章节原名字是“how the secure model works”，暂时翻译做了2.1中的标题。

这一章节从程序的角度描述了安全模式(secure model)是如何工作的，包含了如下几方面的内容：

* NS位和安全监视模式(The NS bit and Secure Monitor mode)；
* 安全内存管理(secure memory management)；
* 系统启动顺序(system boot sequence)；
* 安全中断(secure interrupts)；
* 安全外设(secure peripherals)；
* 安全调试(secure debug)；

### 2.1.1.NS位和安全监视模式

NS位决定了程序在secure world还是non-secure world中。NS位存储在cp15协处理器的SCR(Secure Configuration Register)中。对于处理器来说，**除了安全监控(secure monitor)外**的其他所有模式，都可以被secure world或non-secure world两个world操作。所以安全用户模式(secure user mode)和非安全用户模式(non-secure user mode)同时存在，同理其他模式。

注意：试图在用户态(user mode)，不管是secure world还是non-secure world；或者在非安全特权模式(non-secure privileged mode)，直接访问SCR都将使处理器进入一个未定义异常(undefined exception trap)中。**SCR能且只能被安全特权模式(secure privileged mode)访问。**

安全监控模式(secure monitor mode)是一个特权模式，并总是处于安全域(secure world)中，不论当前的NS位是什么值。也就是说，NS位只能判断其他几种cpu模式处于哪个域中，安全监控模式不处于被判断的模式，它总是安全模式的。

安全监控(secure monitor)是运行在安全监控模式的代码，作用是使处理器从安全域中切换出去或从其他域中切换到安全域中。这段代码的安全依赖于安全启动(secure boot)的代码。

安全监控(secure monitor)在两个域之间切换的时候必须保存处理器环境(processor context)，包括诸如寄存器等。对于CP15中的寄存器，硬件将自动进行切换。除此之外，NS位需要被改写。尽管所有安全特权模式下都可以修改NS位，但ARM有充分理由强烈建议开发者只在安全监控代码(secure monitor)中修改这个值！

较新的ARM指令集中包含了汇编指令SMC，它的作用是进入安全监控模式(secure monitor mode)并执行一个安全监控内核服务调用(secure monitor kernel service call)。该命令只能在特权模式下使用，其他模式下使用将产生一个未定义的异常。

可通过如下命令进入安全监控(secure monitor)：SMC{<cond>} <imm16>。其中cond代表条件选项(condition)；安全监控使用imm16参数决定哪个服务器将被使用，处理器将忽略这个参数(The processor ignores this 16-bit immediate value, but the secure monitor can use it to determine the service to provide.)。

处理器通过如下命令可以从secure monitor中返回：MOVS PC, R14\_mon。

推荐使用如下两步返回到非安全域：设置SCR中的NS位；执行一条movs、subs或rfe命令。ARM处理器的实现保证了在安全许可进入(secure access permission)的情况下，处理器不能在movs、subs或类似指令后，处理预取指令(prefetched instrucions)。

注意：为了避免消弱两个域之间切换的安全性，在执行MSR指令后强烈建议跟上一条IMB指令。

### 2.1.2.安全内存管理

Trustzone的内存管理遵循的原则是：将物理内存划分成安全和非安全的两个区域。安全区域的安全性可以通过检查进入内存或外设的物理入口实现(The secure protection is ensured by checking all physical access to memory or peripherals)。做完了内存划分后，一旦非安全区域的代码试图访问安全区域的入口，将触发cpu的external abort。总线信号(AXI signal) AxPROT[1]表示当前的入口(current access)是安全的还是非安全的，并被用来检查这个入口。

为了保证数据的完整性，安全信息(secure infomation)在内存管理的一些级别(stage)上存在：在L2级别(L2 stage)上，你可以将内存分别映射成secure和non-secure区域；在MMU中，secure和non-secure的描述符(descriptor)可以共存，差别仅在于NSTID。在描述符(descriptor)中，NS属性(attribute)决定了对应的物理内存是secure的还是non-secure的。

对于non-secure的描述符，其NSTID=Non-secure，NS属性强制被置为Non-secure的值。non-secure域只能够映射到non-secure的内存中。

对于secure的描述符，NSTID=Secure，NS属性决定了目标物理内存是secure的还是non-secure的：

* 在cache(指令的或数据的)中，每一行都被标记为secure的或non-secure的，所以这两类数据可以共存。每次cache中作出改变，对应的NS标签(NS tag)都对应的做出改变；
* 对于外部的入口(访问内存或外设？)，AxPROT[1]决定了入口是安全的还是非安全的；

下图描述了secure和non-secure之间内存使用的基本连接方式：



安全域和非安全域的虚拟内存地址映射，表现为了单独的块(block)。下图展示了secure和non-secure的虚拟地址空间是如何映射到物理地址空间中的。



在这个例子中：

* 非安全区的描述符存储在非安全内存(non-secure memory)中，并只能访问非安全区的内存；
* 安全区的描述符存储在安全内存(secure memory)中，并能访问安全和非安全的内存；

## 2.2.优化smc调用

原Hello程序中，对于smc调用采用了“\_\_smc(0) void yield(void)”的方式，声明一个函数yield来实现，但该方式有如下问题：无法传递参数。但在实际应用中我们需要在两个world之间传递参数，所以希望对smc的使用进行补充和优化。

### 2.1.3. 系统启动顺序

trustzone安全扩展技术保证了一个软件环境的安全，但**该技术不能保护处理器不受硬件攻击**，所以一定要保证存放boot代码的硬件设备是安全的！

secure os的代码，从重启向量表中设置指向(The secure os code at the reset vector must:)，应遵循如下规则：

1. 初始化secure os，这与通常的boot操作类似，例如：
   1. 生成页表；如果需要cache或内存保护(memory protection)，就设置MMU；
   2. 设置栈；
   3. 设置运行时环境(run time environment)，并为每种系统模式设置栈；
2. 初始化安全监控，诸如以下一些操作需要被包含：
   1. 分配secure monitor的代码需要的TCM内存区域；
   2. 分配临时工作区(scratch work space)；
   3. 设置安全监控的栈指针，并初始化它的状态块(state block)；
3. 编程实现分区检查(partition checker)，并分配non-secure os可用的物理内存；
4. 将控制权转移给non-secure os，该os将执行boot操作；

### 2.2.1.SMC指令

在2.1.1中简要说了SMC这条汇编指令的基本使用规则，这里再将ARM官网上关于该指令的详细描述进行记录。

在较早版本的ARM汇编语言中，SMC指令被叫做SMI指令。汇编指令调用格式是：SMC{<cond>} #imm4。

在ARM汇编中，关键字“\_\_smc”用来声明一个SMC函数，格式为：\_\_smc(int smc\_num) return-type function-name([argument-list])；各参数含义：

* smc\_num，一个4个bit的立即数，即SMC指令中的imm4；含义同2.1.1中所述；
* argument-list，参数列表；

使用SMC指令有其限制条件：必须是支持安全扩展(security extensions)的ARM框架的处理器，才可以使用该指令。对于不支持该技术的处理器，如果使用了这个函数声明，将产生一个错误。

### 2.2.2.增加参数列表

按照ARM官方的说法，SMC需要传递参数的时候只要将参数加入到参数列表中即可。观察了普遍的SMC\_HANDLER程序后，发现这些处理程序都无一例外的将{r0—r3}这四个寄存器的内容通过栈进行了保护。更确切的说，是在SMC\_HANDLER开始的时候将这四个寄存器的值保存到栈里，在SMC\_HANDLER执行结束的时候再将这些值从栈里释放到对应的寄存器里。

假设要从Secure os通过smc指令跳转到non-secure os里，要传递的参数是arg0—arg3，我们直接将它们作为参数列表写到[argument-list]里。执行SMC\_HANDLER的时候，这些参数值将通过栈被保存，并在SMC\_HANDLER执行结束的时候重新赋值到r0—r3中。但此时已经从secure os进入到了non-secure os里，所以只需要读取这几个寄存器的值就知道想要从secure os里传递过来的是什么参数了。

读取时直接在汇编程序里写读取函数即可，当前每次只能读取一个寄存器的值，尚没有好的办法能一次读取四个寄存器的值。可参考Code中对应的代码，路径：../Code /Trustzone /Extension\_HelloWorld/ TZHW\_pass\_arg.tar.gz。

## 2.3.secure world里调用静态库中的方法

从应用的角度来说，肯定希望os提供的是系统基本服务，应用程序建立在os上，以应用层的层次存在。这里由于我们没有打算实现C库等依赖性库文件，所以希望所有的应用以静态库的形式进行封装，secure os里直接调用这些库中的函数。

这里的代码可参见路径：../Code /Trustzone /Extension\_HelloWorld/call\_liba\_fun.tar.gz。解压后包含两个文件夹，其中gene\_lib\_a是生成最简单的一个静态库，该库文件提供一个函数：print\_hello()，该函数的功能是打印出一条语句来；另外一个就是之前一直在讨论的hello world的程序，其中在main\_secure.c文件中添加了调用该库文件函数的语句和头文件，并在makefile里做出了改动。

在makefile里的改动不是一般的调用方式里使用的“-l”的方式实现的，而是直接将lib库文件链接到了目标文件中，因为这里目标文件采用了不同的编译器进行编译，链接命令也是自己定义的，无法像从前直接使用gcc一条指令完成编译、链接两个操作。

密码学

# 1.DES

## 1.1.DES的ECB模式实现

ECB模式的DES算法，输入8字节的明文/密文、8字节的密钥，可执行加密/解密操作得到密文/明文。

DES本身是一个对称算法，加密和解密使用同一组算法即可实现。

### 1.1.1.概述

算法原理见图1：

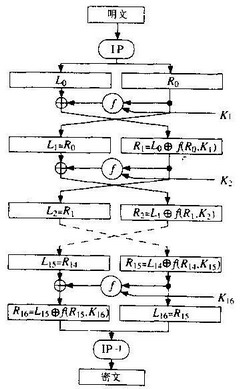


图1 DES算法原理图

详细描述如下：

**Algo. DES**

Input : unsigned char input[8];

Output : unsigned char output[8];

1. 将输入的8字节input内容，转换成64bit的表现形式后，通过IP\_table进行置换，得到全新序列的64bit数据；
2. 对step1中得到的数据，分组成为两个32bit的数据，表示为Li和Ri；
3. 对Li和Ri循环执行16轮变换，变换规则为：Li+1 = Ri; Ri+1 = Li xor f(Ri, Ki)；
4. 16轮循环结束后，将Left和Right的值互换；
5. 将互换后的left和right拼接成64bit数据后，通过IP\_inv\_table进行反置换；
6. 将step4得到的64bit数据组合成8字节的形式保存到output中，并返回；

可见，算法中除step3外，其他步骤比较容易理解。对于每个步骤中使用到的表格，将在程序中给出定义，且这些表格是DES标准中已经定义好的，无需做任何改动。IP\_table定义如下：

const static unsigned char IP\_table[DES\_BLOCK\_BITS\_SIZE] = {

58, 50, 42, 34, 26, 18, 10, 2, 60, 52, 44, 36, 28, 20, 12, 4,

62, 54, 46, 38, 30, 22, 14, 6, 64, 56, 48, 40, 32, 24, 16, 8,

57, 49, 41, 33, 25, 17, 9, 1, 59, 51, 43, 35, 27, 19, 11, 3,

61, 53, 45, 37, 29, 21, 13, 5, 63, 55, 47, 39, 31, 23, 15, 7

};

所谓的置换就是将原来的64bit数据位置打乱后重新放置，放置的规则就是上面的表格所规定的。例如原来的第58个bit，需要置换到第1个bit的位置上，等等。下面依据表格进行置换，都将按照这种规则进行。

IP\_inv\_table定义如下：

const static unsigned char ip\_inv\_table[DES\_BLOCK\_BITS\_SIZE] = {

40, 8,48,16,56,24,64,32,39, 7,47,15,55,23,63,31,

38, 6,46,14,54,22,62,30,37, 5,45,13,53,21,61,29,

36, 4,44,12,52,20,60,28,35, 3,43,11,51,19,59,27,

34, 2,42,10,50,18,58,26,33, 1,41, 9,49,17,57,25

};

step3中，Li+1无需赘述，主要的实现集中在Ri+1上。可以看到，要得到Ri+1，除了需要Li、Ri外，还需要一个Ki，该参数是由密钥扩展(1.1.2)得到的。

对于DES用户来说，无论是加密和解密，除去明文/密文外，自己肯定还要持有一个密钥，否则显然无法进行加解密的操作。这个密钥是64bit的数据。用户在进行加解密的时候，首先需要对这个密钥进行扩展，得到16个48bit的扩展密钥组合。这个扩展密钥集将在16轮循环中被依次使用，也就是上面step3中使用到的Ki。

除此之外，step3中对于还用到了一个函数：f()。该函数(1.1.3)将通过扩展置换、s盒代替、p盒置换三个步骤实现。

### 1.1.2.密钥扩展

如上所述，密钥扩展将8字节的用户手持密钥扩展得到[16][48bit]的密钥集，该密钥集将依次参与加密/解密过程。其流程图参见图2：

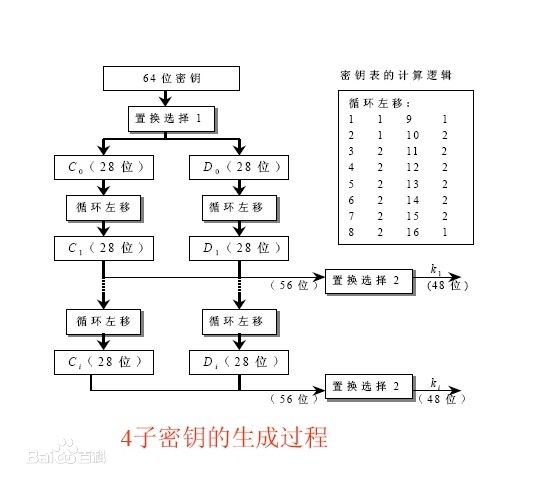


图2 密钥扩展(子密钥集合的生成)

其算法描述如下：

Input : key, 8byte data;

Output : subkey\_set, [16][ 48bit] data;

1. 对输入的8字节密钥，转换得到64bit形式；通过密钥置换表(key\_perm\_table)将其转换成56bit的数据；
2. 将step1中得到的56bit数据，分为Ci和Di两部分，长度都是28bit；
3. 对Ci和Di执行循环左移的操作，得到Ci+1和Di+1；循环左移的位数由表格rls\_table决定，16轮循环中每一轮对应各自的左移位数；
4. 得到的Ci+1和Di+1拼接成56bit的数据后，执行压缩置换；通过comp\_perm\_table(压缩置换表)压缩成为48bit的数据后保存；
5. 循环执行step3和step4，循环次数为16，最终得到16个48bit的数据集：subkey\_set[16][48bits]；

step1中用到了密钥置换表，该表的定义如下：

const static unsigned char key\_perm\_table[DES\_KEY\_PERM\_LEN] = {

57,49,41,33,25,17, 9, 1,58,50,42,34,26,18,

10, 2,59,51,43,35,27,19,11, 3,60,52,44,36,

63,55,47,39,31,23,15, 7,62,54,46,38,30,22,

14, 6,61,53,45,37,29,21,13, 5,28,20,12, 4

};

执行该操作后，将有选择性的将64bit的数据，置换成56bit的数据并投入使用。step3中使用到的rls\_table定义如下：

const static unsigned char rls\_table[DES\_ROTATE\_ROUNDS] = {

1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1

};

该表格定义了每一轮循环执行时，Ci和Di将要循环左移的位数。step4中定义的压缩置换表定义如下：

const static unsigned char comp\_perm\_table[DES\_EXTENSION\_LEN] = {

14,17,11,24, 1, 5, 3,28,15, 6,21,10,

23,19,12, 4,26, 8,16, 7,27,20,13, 2,

41,52,31,37,47,55,30,40,51,45,33,48,

44,49,39,56,34,53,46,42,50,36,29,32

};

该表格将56bit的数据，有选择性的进行压缩置换，置换后得到的数据将是48bit。

### 1.1.3.一轮DES详述

有了扩展密钥集后，将使用该密钥集和经过处理的明文/密文(1.1.1中step1)，执行16轮DES循环操作。每一轮操作的算法都是相同的，只不过输入参数是上一轮的输出，我们单独描述一轮DES的循环。

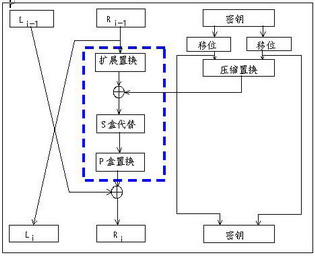


图3 一轮DES

详细描述如下：

Input : Li-1, Ri-1, Ki-1

Output : Li, Ri

1. 将Ri-1赋值给Li；
2. 通过扩展置换表ext\_perm\_table，将Ri-1从32bit扩展到48bit；
3. 将step2中得到的48bit数据，与对应的Ki-1做异或操作；
4. 将step3中得到的48bit数据，执行s盒代替，得到新的32bit数据；
5. 将step4中得到的32bit数据做p盒置换，得到32bit数据；
6. 将step5中得到的32bit数据，与32bit的Li-1进行异或操作后，结果赋值到Ri中；

扩展置换表将32bit的数据，扩展得到48bit的数据，其原理如图4所示：

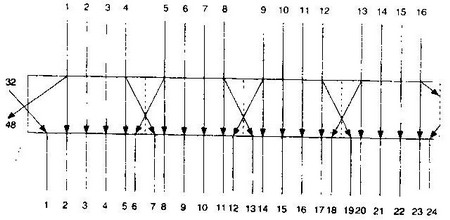


图4 扩展置换

其定义如下：

const static unsigned char ext\_perm\_table[DES\_EXTENSION\_LEN] = {

32, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9,

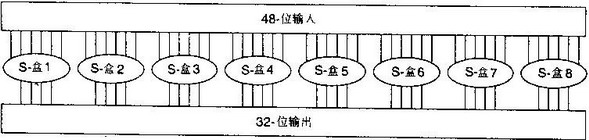
8, 9,10,11,12,13,12,13,14,15,16,17,

16,17,18,19,20,21,20,21,22,23,24,25,

24,25,26,27,28,29,28,29,30,31,32, 1

};

s盒代替是DES算法中的关键操作，因为整个算法中只有该操作不是线性的，也是算法安全性的保证。s盒代替将48bit的数据转换得到32bit的数据，其执行流程如图5所示：



详细描述如下：

Input : 48bit data;

Output : 32bit data;

1. 48bit数据分成8个6bit的数据；
2. 对每个6bit的数据，第1个bit和第6个bit组合，得到0—3之间的数字，标记为row\_num；中间4个bit组合得到0—f之间的数字，标记为column\_num；
3. 根据row\_num和column\_num，在s\_box中寻找对应的数值(1字节)，并将该值分解成4bit的形式，替换之前的6bit数据的位置；
4. 循环执行step2和step3两个操作，直到所有的8个6bit数据都替换成了4bit数据；

s\_box\_table的定义如下所示：

const static unsigned char s\_box\_table[DES\_BLOCK\_BYTES\_SIZE][DES\_BLOCK\_BITS\_SIZE] = {

{//S1

14, 4,13, 1, 2,15,11, 8, 3,10, 6,12, 5, 9, 0, 7,

0,15, 7, 4,14, 2,13, 1,10, 6,12,11, 9, 5, 3, 8,

4, 1,14, 8,13, 6, 2,11,15,12, 9, 7, 3,10, 5, 0,

15,12, 8, 2, 4, 9, 1, 7, 5,11, 3,14,10, 0, 6,13

},

{//S2

15, 1, 8,14, 6,11, 3, 4, 9, 7, 2,13,12, 0, 5,10,

3,13, 4, 7,15, 2, 8,14,12, 0, 1,10, 6, 9,11, 5,

0,14, 7,11,10, 4,13, 1, 5, 8,12, 6, 9, 3, 2,15,

13, 8,10, 1, 3,15, 4, 2,11, 6, 7,12, 0, 5,14, 9

},

{//S3

10, 0, 9,14, 6, 3,15, 5, 1,13,12, 7,11, 4, 2, 8,

13, 7, 0, 9, 3, 4, 6,10, 2, 8, 5,14,12,11,15, 1,

13, 6, 4, 9, 8,15, 3, 0,11, 1, 2,12, 5,10,14, 7,

1,10,13, 0, 6, 9, 8, 7, 4,15,14, 3,11, 5, 2,12

},

{//S4

7,13,14, 3, 0, 6, 9,10, 1, 2, 8, 5,11,12, 4,15,

13, 8,11, 5, 6,15, 0, 3, 4, 7, 2,12, 1,10,14, 9,

10, 6, 9, 0,12,11, 7,13,15, 1, 3,14, 5, 2, 8, 4,

3,15, 0, 6,10, 1,13, 8, 9, 4, 5,11,12, 7, 2,14

},

{//S5

2,12, 4, 1, 7,10,11, 6, 8, 5, 3,15,13, 0,14, 9,

14,11, 2,12, 4, 7,13, 1, 5, 0,15,10, 3, 9, 8, 6,

4, 2, 1,11,10,13, 7, 8,15, 9,12, 5, 6, 3, 0,14,

11, 8,12, 7, 1,14, 2,13, 6,15, 0, 9,10, 4, 5, 3

},

{//S6

12, 1,10,15, 9, 2, 6, 8, 0,13, 3, 4,14, 7, 5,11,

10,15, 4, 2, 7,12, 0, 5, 6, 1,13,14, 0,11, 3, 8,

9,14,15, 5, 2, 8,12, 3, 7, 0, 4,10, 1,13,11, 6,

4, 3, 2,12, 9, 5,15,10,11,14, 1, 7, 6, 0, 8,13

},

{//S7

4,11, 2,14,15, 0, 8,13, 3,12, 9, 7, 5,10, 6, 1,

13, 0,11, 7, 4, 0, 1,10,14, 3, 5,12, 2,15, 8, 6,

1, 4,11,13,12, 3, 7,14,10,15, 6, 8, 0, 5, 9, 2,

6,11,13, 8, 1, 4,10, 7, 9, 5, 0,15,14, 2, 3,12

},

{//S8

13, 2, 8, 4, 6,15,11, 1,10, 9, 3,14, 5, 0,12, 7,

1,15,13, 8,10, 3, 7, 4,12, 5, 6,11, 0,14, 9, 2,

7,11, 4, 1, 9,12,14, 2, 0, 6,10,13,15, 3, 5, 8,

2, 1,14, 7, 4,10, 8,13,15,12, 9, 0, 3, 5, 6,11

}

};

最后执行p\_box的置换操作，将32位数据进行再一次的置换，该表格的定义如下：

const static unsigned char p\_box\_table[DES\_HALF\_PART\_LEN] = {

16, 7,20,21,29,12,28,17, 1,15,23,26, 5,18,31,10,

2, 8,24,14,32,27, 3, 9,19,13,30, 6,22,11, 4,25

};

### 1.1.4.加密、解密

解密与加密的流程大致相同，不同之处是：将subkey\_set按照15->0的顺序参与循环运算；

### 1.1.5.Key的长度

DES算法，允许key的长度不是8，当key的长度不是8的时候:

* Key的长度小于8，补0在最后，补齐到8bytes；
* Key的长度大于8，只使用前8个字节作为key；

### 1.1.6.局限性

由于DES算法严格要求输入数据的长度，是8字节的整数倍，因此限制了算法的使用。经过网上查看资料，有一些算法可以解决这个问题，但是会使得算法本身更加复杂，似乎得不偿失，不如直接在调用者调用的时候，就进行填充来的容易些。

填充算法，简单来说，可以在最后一个不足8bytes的块，补充0x00或者0xff，记录补充的字节数后，解密后将这些填充位忽略；但该操作由应用来做更加合适，可适当选择使用的填充值，算法自身严格要求8字节的倍数作为输入长度；

## 1.2.DES3算法的ECB模式

由于DES算法提出年代较早，安全性随着计算机的发展受到了很大挑战，所以3des算法被提出，通过3轮密切相关的des运算，提高安全性。

算法原理描述如下：

3DES Algo.

input : key 16byte or 24byte, input, 8byte;

output : output, 8byte;

1. 对key按照8个字节为一个单位，执行密钥扩展，得到3个(或只有2个)subkey\_set；
2. 使用subkey\_set0/subkey\_set2对input执行加密/解密操作，得到ret0；
3. 使用subkey\_set1对ret0执行解密/加密操作，得到ret1；
4. 使用subkey\_set2/subkey\_set0对ret1执行加密/解密操作，得到ret2；
5. ret2即为所需要的结果；

对该算法有如下需要注意的地方：

* 执行3DES加密操作时，step2-4的顺序是：加密、解密、加密；执行解密时顺序是：解密、加密、解密；
* 使用密钥的顺序也不同，加密操作是：subkey\_set0、1、2；解密操作时，subkey\_set2、1、0；
* 使用3DES算法时，密钥的有效长度可以是16byte和24byte两种。如果是16byte，将生成两个密钥子集set1和set2，set1将同样被当做set3使用；而24byte则使用3个完全不同的set；

## 1.3.DES算法的CBC模式

相比于ECB模式(电子密码本)模式，CBC模式(密码分组链接模式)密级更高，但也更加复杂；原理如下:

加密：

1. 将输入数据，按照8bytes一组，分为D1,D2,…,Dn；
2. 第一组数据D1，先与初始化向量Iv异或，再进行DES加密，得到第一组明文C1；
3. D2与C1异或后，在进行DES加密，得到C2；
4. 以此类推，直至Dn；
5. 将C1,C2,…,Cn连接起来，即为加密结果；

解密：

1. 将密文数据，按照8bytes一组，分为C1,C2,…,Cn；
2. 将C1先使用DES解密，再与初始化向量Iv执行异或，得到D1；
3. 将C2先解密，再与C1执行异或后得到D2；
4. 以此类推，直至Dn；
5. 将D1,D2,…,Dn连接起来就是明文；

两种模式的差别是：

ECB模式更加简单、有利于并行计算、无差不会被传递；但也不能隐藏明文的模式，可对明文产生攻击；

CBC模式不容易主动攻击，安全性更高；但不利于并行计算，影响效率，误差会传递，需要初始化向量Iv；

Iv的选择，一般是随机生成的；每一次加解密使用相同的Iv，下一次加解密时更换该值；这样就可以保证即便是相同的明文和密钥，得到的密文也是不同的了；密级也因此更高一些。

# 2.AES算法

AES算法中，一个字节占用8bit，一个字是4个字节，占用32bit的空间。下面的所有描述都是基于这个基本概念的。

## 2.1.AES算法的ECB模式实现

AES算法属于分组密码算法，它的输入分组、输出分组和加解密过程中产生的中间分组都是128bit的数据。密钥的长度可以由如下3个选择：128bit，192bit和256bit；也可分别描述为4、6、8个字。我们用Nk来表示密钥的字数。

一般来讲，用Nr表示对一个数据分组加密的轮数，Nr与Nk有一一对应的关系，如下表2.1所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AES类型 | 密钥长度 | 分组大小 | 加密轮数 | 扩展密钥字节数 |
| AES-128 | 4字 | 4字 | 10 | 176 |
| AES-192 | 6字 | 4字 | 12 | 208 |
| AES-256 | 8字 | 4字 | 14 | 240 |

表2.1 密钥长度及轮数对应关系

AES算法，ECB模式下的加解密原理可通过图2.1概要描述：



图2.1 AES算法概述

显然，每一轮都需要一个和输入分组具有相同长度的扩展密钥Ke的参与。由于外部输入的加密密钥K长度有限，所以在AES中同样要使用密钥扩展程序(key\_expansion)把外部密钥K扩展成更长的数组，以完成各轮的操作。

# 3.SHA1散列算法

Algorithm SHA1

Function : 计算给定字符串的SHA1散列值；

Input : 字符集合(一般以字符串的形式体现)str\_input；

Output : 该字符集的SHA1散列值digest，确定为160bit，也就是20byte；

1. 补位、补长度(详见3.1)，使str\_input🡪exp\_input；
2. 将exp\_input按照指定规则(详见3.2)分块，得到M0，M1，……，Mn-1；
3. 初始化数据：i = 0；H[5]、K[4]分别进行初始化(详见3.3)；
4. 使用Mi对H[5]进行刷新(详见3.4)，得到新一组H[5]的值；
5. 如果I == n – 1 : 执行step 6；

否则：执行step 4；

1. 将H[5]分解成20个byte的数据，赋值到digest中(详见3.5)，就是输入数据的散列值；

## 3.1.补位、补长度

### 3.1.1.补位

所谓补位是指输入数据必须进行补位，以使其长度在对512bit取模以后的余数是448bit。也就是说，（补位后的消息长度）%512 = 448。即使原始长度已经满足对512bit取模后余数是448bit，补位也必须要进行。

补位是这样进行的：先补一个bit的1，然后再补0，直到长度满足对512取模后余数是448。总而言之，补位是至少补一位，最多补512位。还是以前面的“abc”为例显示补位的过程。

原始信息： 01100001 01100010 01100011

补位第一步：01100001 01100010 01100011 1 //首先补一个“1”

补位第二步：01100001 01100010 01100011 10…..0  //然后补423个“0”

我们可以把最后补位完成后的数据用16进制写成下面的样子

61626380 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000

### 3.1.2.补长度

执行过上面的补位操作后，原始数据已经满足了长度对512bit取模后的余数是448bit，还需要在最后再补充64bit的长度值，使原始数据的长度满足时512bit的长度。

这里的长度值是原始数据的长度，不包括补位和补长度的值，单位是bit。

通常用一个64位的数据来表示原始消息的长度。如果消息长度不大于2^64，那么第一个字就是0。在进行了补长度的操作以后，整个消息就变成下面这样了（16进制格式）

61626380 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000018

## 3.2.分块

经过3.1中的操作，输入数据input已经被补充成为了exp\_input，且exp\_input的长度显然是512bit(64byte)的整数倍。

这里所谓的分块就是将exp\_input按照512bit(64byte)为基本单位进行划分，得到N个分块，表示成为：M[0], ……, M[N-1]。显然任何一个M[i]的长度都是64个byte。

## 3.3.初始化数据

进行SHA1的计算之前，还需要进行一些初始化操作，主要指的是：

### 3.3.1.一系列常量

在后面3.4中对H进行刷新的时候，要进行80轮相关运算，运算过程中需要使用到一下的K值。针对不同的轮数，K的值不同，t代表了轮数。

Kt = 0X5A827999 (0 <= t <= 19)

Kt = 0X6ED9EBA1 (20 <= t <= 39)

Kt = 0X8F1BBCDC (40 <= t <= 59)

Kt = 0XCA62C1D6 (60 <= t <= 79)

### 3.3.2.使用到的函数

同样在3.4中的80轮操作中，需要使用到一些定义好的函数，这些函数针对不同的轮数，也有不同的定义：

ft(B,C,D) = (B and C) or ( (not B) and D ) ( 0 <= t <= 19)

ft(B,C,D) = B xor C xor D              (20 <= t <= 39)

ft(B,C,D) = (B and C) or (B and D) or (C and D) (40 <= t <= 59)

ft(B,C,D) = B xor C xor D                     (60 <= t <= 79).

### 3.3.3.变量初值

由上可知，3.4中主要对H[5]通过某种方法不断改变其值，最终H[5]将构造出结果。

但显然，H[5]需要有一个初值，这里算法规定如下：

H[0] = 0X67452301;

H[1] = 0XEFCDAB89;

H[2] = 0X98BADCFE;

H[3] = 0X10325476;

H[4] = 0XC3D2E1F0;

## 3.4.刷新H[5](算法核心)

对每一个M[i]，都需要执行本部分操作，并借此刷新H[5]的值。直到M[N-1]执行完，H[5]将作为最终值，稍作格式变化后输出。这里描述针对每一个M[i]的操作。

这里引入SHA1算法中关于字的定义：**一个字**指的是占用4个字节的基本单位，也就是一个word占用了32个bit。

### 3.4.1.循环左移函数RSL

给定数字number，给定移位数量shift\_bit，循环左移函数可如下实现：

#define RSL(number, shift\_bit) ((number << shift\_bit) | (number >> (32 – shift\_bit)))

这里注意，在C语言中，如果是有符号数，进行右移时编译器会将符号位考虑进来，写程序时应特别注意这点。

### 3.4.2.W[80]的生成

1. 将M[i]中64个byte的数据，每四个组成一个字后，按照顺序赋值到W[0]—W[15]中；例如，M[i] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, …} 🡪 W[0]=0x01020304, W[1] = 0x05060708,….；
2. 对于W[16]—W[79]的生成，遵循如下规则：W[i] = RSL( (W[i-3] xor W[i-8] xor W[i-14] xor W[i-16]), 1 )；

### 3.4.3.H[5]的刷新

取临时变量：A = H[0], B = H[1], C= H[2], D = H[3], E = H[4]。

如下操作，循环执行80次：

temp = RSL(A, 5) + ft(B, C, D) + E + W[i] + K[t];

E = D; D = C; C = RSL(B, 30); B = A; A = temp;

第一条语句中ft()函数、K[t]常量参见3.3.1和3.3.2，W[i]参见3.4.2，RSL函数参见3.4.1,。在执行完上述80次循环后，执行如下操作：

H0 = H0 + A

H1 = H1 + B,

H2 = H2 + C,

H3 = H3 + D,

H4 = H4 + E

这样H[5]就完成了针对M[i]的刷新操作。

## 3.5.构造结果

得到的H[5]是5个字，SHA1算法规定的输出时20个byte，所以需要将每个字变换成4个byte的形式返回。

此时算法结束。

# 4.MD5散列算法

MD5算法全程是Message Digest Algorithm 5，中文名是消息摘要算法第五版，是一种散列算法，用来提供信息的完整性保护。

## 4.1.算法步骤

MD5算法以512位为一个分组处理输入的信息，每一个分组又被划分为16个32个子分组，经过一系列数学运算后，输出四个32位的分组数据，这4个32位的分组数据级联成128bit的散列。

### 4.1.1.填充

如果输入信息的长度(单位为bit！)对512求余的结果不等于448，就需要填充，使其长度对512求余结果为448。填充方法是一个bit的1和若干个bit的0.填充完成后，信息的长度满足：N\*512+448(bit)；

### 4.1.2.记录长度

用64bit的数据记录**填充之前**的数据长度，这64bit的数据紧随4.1.1中填充之后的数据，这样信息的长度就成为了：（N+1）\*512bit；

要注意的是，这里添加长度时的顺序一定是小端的，例如“abc”这个字符串，三个字节的长度(\0不计算在长度内)，换算成bit就是24，折合成十六进制就是0x18。按照正常的大端，这里要添加的就会是0x0000 0000 0000 0018，但是在MD5算法中要求的一定是：0x1800 0000 0000 0000的形式；

### 4.1.3.装入标准幻数

标准幻数(物理顺序)是：A=(0x01234567), B=(0x89ABCDEF), C=(0Xfedcba98), D=(0x76543210);

但是在程序中要注意，能够使用的应该是字节序相反的变量定义：A=(0x67452301), B=(0xefcdab89), C=(0x98badcfe), D=(0x10325476);

### 4.1.4.四轮循环运算

首先做如下一些基本函数定义：

F(X, Y, Z) = (X & Y) | ((~X) & Z);

G(X, Y, Z) = (X & Z) | (Y & (~Z));

H(X, Y, Z) = (X ^ Y ^ Z);

I(X, Y, Z) = Y ^ (X | (~Z));

这是四个非线性函数，这四个函数说明，如果X、Y和Z的对应位是独立和均匀的，那么结果的每一位也是独立和均匀的；

假设M[j]表示消息的第j个子分组，<<<表示循环左移，那么：

FF(a, b, c, d, M[j], s, ti)表示：a = b + ( (a + F(b, c, d) + M[j] + ti) <<< s );

GG(a, b, c, d, M[j], s, ti)表示：a = b + ( (a + G(b, c, d) + M[j] + ti) <<< s );

HH(a, b, c, d, M[j], s, ti)表示：a = b + ( (a + H(b, c, d) + M[j] + ti) <<< s );

II(a, b, c, d, M[j], s, ti)表示：a = b + ( (a + I(b, c, d) + M[j] + ti) <<< s );

1. for(i = 0; i < N+1; i++) //数据长度为(N+1)\*512bits，对每一个512bits的分组执行如下的操作
2. 将512bits分为16个分组，每个分组的size为32bits，定义这些分组为M[0]—M[15]，注意这些分组组合成32bit的数字的顺序也是小端的！例如，32bit的char是[0x12, 0x34, 0x56, 0x78];那么M[x]=0x78563412,而不是0x12345678！
3. 对这16个分组进行判断：

a = A; b = B; c = C; d = D;

//传说中的对M[j]的第一轮循环

        FF(a,b,c,d,M[0],7,0xd76aa478);

        FF(d,a,b,c,M[1],12,0xe8c7b756);

        FF(c,d,a,b,M[2],17,0x242070db);

        FF(b,c,d,a,M[3],22,0xc1bdceee);

        FF(a,b,c,d,M[4],7,0xf57c0faf);

        FF(d,a,b,c,M[5],12,0x4787c62a);

        FF(c,d,a,b,M[6],17,0xa8304613);

        FF(b,c,d,a,M[7],22,0xfd469501) ;

        FF(a,b,c,d,M[8],7,0x698098d8) ;

        FF(d,a,b,c,M[9],12,0x8b44f7af) ;

        FF(c,d,a,b,M[10],17,0xffff5bb1) ;

        FF(b,c,d,a,M[11],22,0x895cd7be) ;

        FF(a,b,c,d,M[12],7,0x6b901122) ;

        FF(d,a,b,c,M[13],12,0xfd987193) ;

        FF(c,d,a,b,M[14],17,0xa679438e) ;

        FF(b,c,d,a,M[15],22,0x49b40821);

        //传说中对M[j]的第二轮循环

        GG(a,b,c,d,M[1],5,0xf61e2562);

        GG(d,a,b,c,M[6],9,0xc040b340);

        GG(c,d,a,b,M[11],14,0x265e5a51);

        GG(b,c,d,a,M[0],20,0xe9b6c7aa) ;

        GG(a,b,c,d,M[5],5,0xd62f105d) ;

        GG(d,a,b,c,M[10],9,0x02441453) ;

        GG(c,d,a,b,M[15],14,0xd8a1e681);

        GG(b,c,d,a,M[4],20,0xe7d3fbc8) ;

        GG(a,b,c,d,M[9],5,0x21e1cde6) ;

        GG(d,a,b,c,M[14],9,0xc33707d6) ;

        GG(c,d,a,b,M[3],14,0xf4d50d87) ;

        GG(b,c,d,a,M[8],20,0x455a14ed);

        GG(a,b,c,d,M[13],5,0xa9e3e905);

        GG(d,a,b,c,M[2],9,0xfcefa3f8) ;

        GG(c,d,a,b,M[7],14,0x676f02d9) ;

        GG(b,c,d,a,M[12],20,0x8d2a4c8a);

        //传说中对M[j]的第三轮循环

        HH(a,b,c,d,M[5],4,0xfffa3942);

        HH(d,a,b,c,M[8],11,0x8771f681);

        HH(c,d,a,b,M[11],16,0x6d9d6122);

        HH(b,c,d,a,M[14],23,0xfde5380c) ;

        HH(a,b,c,d,M[1],4,0xa4beea44) ;

        HH(d,a,b,c,M[4],11,0x4bdecfa9) ;

        HH(c,d,a,b,M[7],16,0xf6bb4b60) ;

        HH(b,c,d,a,M[10],23,0xbebfbc70);

        HH(a,b,c,d,M[13],4,0x289b7ec6);

        HH(d,a,b,c,M[0],11,0xeaa127fa);

        HH(c,d,a,b,M[3],16,0xd4ef3085);

        HH(b,c,d,a,M[6],23,0x04881d05);

        HH(a,b,c,d,M[9],4,0xd9d4d039);

        HH(d,a,b,c,M[12],11,0xe6db99e5);

        HH(c,d,a,b,M[15],16,0x1fa27cf8) ;

        HH(b,c,d,a,M[2],23,0xc4ac5665);

        //传说中对M[j]的第四轮循环

        II(a,b,c,d,M[0],6,0xf4292244) ;

        II(d,a,b,c,M[7],10,0x432aff97) ;

        II(c,d,a,b,M[14],15,0xab9423a7);

        II(b,c,d,a,M[5],21,0xfc93a039) ;

        II(a,b,c,d,M[12],6,0x655b59c3) ;

        II(d,a,b,c,M[3],10,0x8f0ccc92) ;

        II(c,d,a,b,M[10],15,0xffeff47d);

        II(b,c,d,a,M[1],21,0x85845dd1) ;

        II(a,b,c,d,M[8],6,0x6fa87e4f) ;

        II(d,a,b,c,M[15],10,0xfe2ce6e0);

        II(c,d,a,b,M[6],15,0xa3014314) ;

        II(b,c,d,a,M[13],21,0x4e0811a1);

        II(a,b,c,d,M[4],6,0xf7537e82) ;

        II(d,a,b,c,M[11],10,0xbd3af235);

        II(c,d,a,b,M[2],15,0x2ad7d2bb);

        II(b,c,d,a,M[9],21,0xeb86d391);

        A += a;

        B += b;

        C += c;

        D += d;

1. 对所有子分组执行如上的循环操作；

### 4.1.5.最终结果

处理完所有512分组的内容后，得到一组新的A B C D的值，将这些值按照ABCD的级联顺序排序，然后输出即为最终的MD5值。

需要注意的是，MD5是小端序列，所以输出时要做一次转换，例如A有32bit，分为A1A2A3A4，那么输出的就应该是A4A3A2A1。

### 4.1.6.一些典型的MD5值

md5(“”) = d41d8cd98f00b204e9800998ecf8427e

MD5 ("a") = 0cc175b9c0f1b6a831c399e269772661

MD5 (“ab”) = 187ef4436122d1cc2f40dc2b92f0eba0

MD5 ("abc") = 900150983cd24fb0d6963f7d28e17f72

MD5 (“abcde”) = ab56b4d92b40713acc5af89985d4b786

MD5 ("message digest") = f96b697d7cb7938d525a2f31aaf161d0

MD5 ("abcdefghijklmnopqrstuvwxyz") = c3fcd3d76192e4007dfb496cca67e13b

MD5 ("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz") = f29939a25efabaef3b87e2cbfe641315

# 5.RC4算法

RC4算法是一种在电子信息领域加密的技术手段，常用于无线通信领域。和DES算法一样，它是一种对称加密算法，也就是说使用的密钥为私钥。

不同于DES的是，RC4不是对明文进行分组处理，而是以字节流的形式加密每一个字节，解密的时候也是依次对密文中的每一个字节执行解密。

算法流程如下：

## 5.1.关键变量

### 5.1.1.密钥流

RC4算法的关键是根据明文和密钥生成相应的密钥流，密钥流的长度和明文的长度是对应的，也就是说明文的长度是500字节，那么密钥流也是500字节。当然，加密生成的密文也是500字节，因为密文第i字节=明文第i字节^密钥流第i字节；

### 5.1.2.状态向量S

长度为256，S[0],S[1].....S[255]。每个单元都是一个字节，算法运行的任何时候，S都包括0-255的8比特数的排列组合，只不过值的位置发生了变换；

### 5.1.3.临时向量T

长度也为256，每个单元也是一个字节。如果密钥的长度是256字节，就直接把密钥的值赋给T，否则，轮转地将密钥的每个字节赋给T；

### 5.1.4.密钥K

长度为1-256字节，注意密钥的长度 keylen 与明文长度、密钥流的长度没有必然关系，通常密钥的长度取值为16字节（128比特）。

## 5.2.算法原理

### 5.2.1.初始化S和T

for i=0 to 255 do

S[i] =i;

T[i]=K[ I mod keylen ];

### 5.2.2.初始排列S

for i=0 to 255 do

j= ( j+S[i]+T[i])mod256;

swap(S[i],S[j]);

### 5.2.3.产生密钥流

for r=0 to len do //r为明文长度，r字节

i=(i+1) mod 256;

j=(j+S[i])mod 256;

swap(S[i],S[j]);

t=(S[i]+S[j])mod 256;

k[r]=S[t];

### 5.2.4.生成密文/明文

密文第i字节=明文第i字节^密钥流第i字节；

或者：明文第i字节=密文第i字节^密钥流第i字节；

# 6.BASE64算法

很多场合下，特殊字符的存在不被允许，例如邮件只允许可见字符的传送，但很多ASCII码的控制字符并不是可见字符，就不能通过邮件传送了。BASE64就是一种转换算法，将所有字符都转换为可见字符。

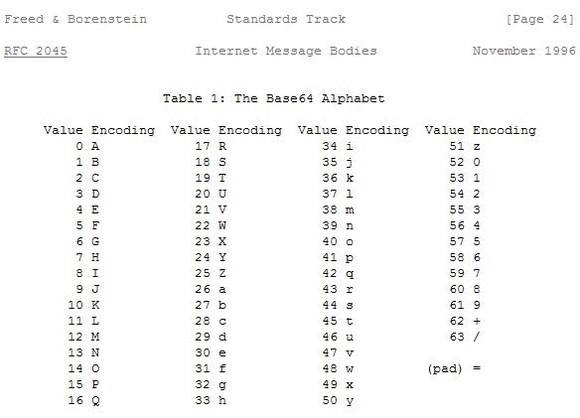
之所以被取名为base64，是因为其加密后的密文只会由A-Z, a-z, 0-9和+ / 这64个可见字符组成，不会出现其他字符；

## 6.1.原理

查阅网上存在的几种原理说明文档后，发现如下的这种通俗的举例式方法更容易被人理解，引用之：

1. 假设有3个字符要做base64加密，首先将其转换成二进制，那么此时得到一个24bits的格式；
2. Base64只支持64个有效字符，二进制的话就是26个，也就是6个bit；所以24个bit的话，就需要4 \* 6bit；因此将这个24bits的数据分成4份，每份6个bit；
3. 对每6个bits的数据，前面增加00后填充为8bit的数据，这就是一个字节的数据了，得到这个数据的值；
4. 按照下图所示的Key-value的字典，将这个数值转换为base64支持的字符；
5. 得到了4个字符，就是BASE64加密后的数据了。

下图为标准协议(RFC2045)规定的Key-value的字典：



按照上面的流程，可以轻松处理3\*N个字符的base64加密；但是并不总是需要对3N个字节的数据进行处理，针对3N+1和3N+2个字节的数据的加密，处理方式如下所述：

* BASE64规定的处理原则是：不足之处使用”=”补齐；
* 比如，对A这个字符进行处理:
  + 0xA=(01000001)2;
  + 转换为两个base64支持的字符就是：00010000, 00010000；
  + 查找map表，这两个的value是Q；到此得到了QQ；
  + 但是还不满足BASE64的密文长度是4的整数倍，缺少2个字节的数据，按照协议规定，此时补充=，因此最终结果就是:QQ==;

## 6.2.变种

所谓的BASE64的变种，大概有两种方式：

1. 对原始内容进行修改，按照自行约定的规则，在原始字符串的指定位置增加一些信息，解密后忽略这些信息即可；
2. 对key-value的map进行调整；

ARM板级开发

# 1.ARM汇编程序设计

本部分是学习《汇编语言程序实践—基于ARM体系结构》一书时做的笔记，主要记录ARM汇编的基本知识、规范等。

## 1.1.ARM汇编语言程序设计(第四章)

### 1.1.1.源程序的组成(4.1.2)

#### 1.1.1.1.源程序的结构

一个示例ARM汇编程序：

AREA Init,CODE,READONLY

ENTRY

Start

LDR R0,=0X3FF5000

LDR R1,0XFF

STR R1,[R0]

LDR R0,=0X3FF5008

LDR R1,0X01

STRR1 ,[R0]

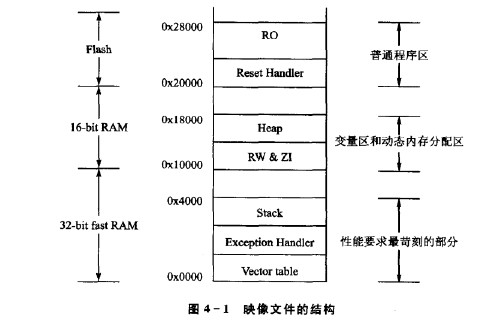
……..

END

在ARM汇编中，以**程序段**为单位组织代码。段是相对独立的指令或数据序列，具有特定的名称。段可以分为代码段、数据段和通用段。以**AREA**伪操作定义一个段，用来说明该段的属性，例如代码段的属性一般是READONLY，数据段的默认属性是READWRITE。

一个汇编程序至少应该有一个代码段。ENTRY伪操作标识程序的入口点，一个ARM程序中可以有多个ENTRY(**至少要有一个**)。程序的末尾应该是END伪操作，用来告知编译器源文件到此结束。每一个汇编程序都**必须**有一条END指令，指示代码段的结束。

当程序较长时，可以分割为多个代码段和数据段，多个段在程序编译连接时最终形成一个可执行的映像文件。连接器则根据系统默认或用户设定的规则，将各个段安排在存储器中的相应位置。因此源程序中段之间的相对位置与可执行文件中段的相对位置一般不会相同。一个典型的映像文件的结构如下图所示。



#### 1.1.1.2.语句格式

ARM汇编语言程序的每行语句可以由4个部分组成：

|  |  |  |  |
| --- | --- | --- | --- |
| [LABEL] | OPERATION | [OPERAND] | [;COMMENT] |
| 标号域 | 操作助记符域 | 操作数域 | 注释域 |

标号域用来表示程序的地址、变量、过程名、数据的地址和常量。标号是一个自行设计的标识符或名称，语句标号**可以是大小写字母的混合**。语句标号**必须**在一行的开头书写，不能留空格。

操作助记符域可以是以下几类：指令、伪操作、宏指令或伪指令。

每一条指令的助记符可以全部用大写或全部用小写，但**不允许**在一条指令中大小写混用。所有的指令都**不能**在行的开头书写，**必须**在指令的前面有空格。指令助记符和后面的操作数(寄存器)之间**必须**有空格，**不可以**在这之间使用逗号。

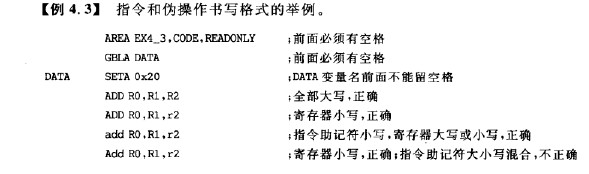
伪操作是ARM汇编中的一些特殊助记符，作用是完成汇编程序的各种准备工作，一旦汇编结束，这些伪操作的使用也就随之完成。

宏指令是一段独立的程序代码，可插在源程序中，它通过伪操作来定义。宏在使用之前**必须**定义好，宏之间可以互相调用，也可以递归调用。

伪指令是ARM汇编中特殊的指令助记符，不在程序运行期间被执行。它们在汇编时将被合适的机器指令代替成ARM或Thumb指令，从而实现真正的指令操作。

操作数域表示操作的对象，可以使常量、变量、标号、寄存器名或表达式，不同对象之间**必须**用逗号分开。

注释域以“；”开始。



### 1.1.2.常用到的符号(4.1.3)

汇编语言中用到的符号命名应遵循如下规则：

* 符号由大小写字母、数字以及下划线组成；
* 符号区分大小写；
* 符号在其作用域内必须唯一；
* 自定义的符号不能与系统的保留字相同；
* 符号名不能与指令或伪指令同名；

#### 1.1.2.1.程序中的变量

ARM(Thumb)汇编语言中，支持的变量有：数值变量、逻辑变量和字符串变量。

全局数值变量使用伪操作**GBLA**(global arith.)定义，局部数值变量使用伪操作**LCLA**(local arith)定义。数值变量使用**SETA**伪操作赋值。

逻辑变量用于在程序的运行中保存逻辑值：真(TRUE)或假(FALSE)。全局逻辑变量使用伪操作**GBLL**定义，局部逻辑变量使用伪操作**LCLL**定义。逻辑变量的赋值使用伪操作**SETL**实现。

字符串变量在32位芯片上长度不应该超过512字节，最小长度**为0**。全局变量使用伪操作**GBLS**定义，局部变量使用**LCLS**定义。使用**SETS**赋值。

#### 1.1.2.2.程序中的常量

ARM(Thumb)汇编支持的常量包括：数值常量、逻辑常量和字符串常量。

ARM汇编中使用EQU定义数值常量，有以下表示方式：

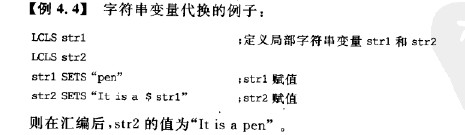
* 十进制数：直接表达，例如1,2,345；
* 十六进制数：两种方式，第一种使用前缀0x，第二种用前缀&；两种方式是等效的；
* n进制数：形如n\_XXX，其中n的范围是2-9；
* ASCII表示：例如‘A’表达的就是0x42的数值，命令“MOV R1，#’B’”等同于“MOV R1，#0X42”；

逻辑常量只有两种取值：{TRUE}、{FALSE}，**注意带大括号**。

字符串常量是固定字符串，可以使用c语言中的转义字符，例如“abcd\0c\r\n”等。

#### 1.1.2.3.汇编时的变量代换

如果在字符串变量前加代换操作符“$”，编译器会自动将其进行代换，如下图：



如果数值变量前面有代换操作符“$”，编译器会将该数值转成十六进制的值后进行代换，例如：a1 SETA 12; str1 SETS “The number is $ a1”。汇编后str1的值将是：“The number is 0000000c”。

如果逻辑变量加该操作符，将直接代换为其真值。

#### 1.1.2.4.表达式和运算符

##### 数值表达式及运算符

|  |  |  |  |
| --- | --- | --- | --- |
| 类别 | 符号 | 含义 | 使用方式 |
| 算术运算符 | + | 加法 | X+Y |
| - | 减法 | X-Y |
| \* | 乘法 | X\*Y，这么用对不对，需要测试一下 |
| / | 除法 | X/Y |
| MOD | 求模 | X:MOD:Y |
| 移位运算符 | ROL | 循环左移 | X:ROL:Y |
| ROR | 循环右移 | X:ROR:Y |
| SHL | 左移 | X:SHL:Y |
| SHR | 右移 | X:SHR:Y |
| 位逻辑运算符 | AND | 逻辑与 | X:AND:Y |
| OR | 逻辑或 | X:OR:Y |
| NOT | 逻辑非 | :NOT:Y |
| EOR | 逻辑异或 | X:EOR:Y |

例如：MOV R5，#0X20：MOD:0X04就等同于MOV R5，#0；

##### 逻辑表达式及运算符

|  |  |  |  |
| --- | --- | --- | --- |
| 类别 | 符号 | 含义 | 使用方式 |
| 关系操作符 | = | 两数相等的关系 | X=Y |
| > | 大于 | X>Y |
| < | 小于 | X<Y |
| >= | 大于等于 | X>=Y |
| <= | 小于等于 | X<=Y |
| <> | 不等于 | X<>Y |
| /= | 不等于 | X/=Y |
| 逻辑操作符 | LAND | 逻辑与 | X:LAND:Y |
| LOR | 逻辑或 | X:LOR:Y |
| LNOT | 逻辑非 | :LNOT:Y |
| LEOR | 逻辑异或 | X:LEOR:Y |

对于逻辑表达式来说，返回值都是{TRUE}或{FALSE}中的一个。

##### 字符串表达式及运算符

|  |  |  |
| --- | --- | --- |
| 符号 | 含义 | 使用方式 |
| LEN | 返回字符串的长度 | :LEN:X |
| CHR | 将整数转换为一个含ASCII字符的字符串 | :CHR:M |
| STR | 将一个数值表达式或逻辑表达式转换为一个字符串 | :STR:X |
| LEFT | 返回某个字符串左端的一个子串 | X:LEFT:Y，返回X串左端Y个字符的子串 |
| RIGHT | 返回右端的字串 | X:RIGHT:Y， |
| CC | 连接两个字符串 | X:CC:Y，将Y连接到X后面 |

##### 基于寄存器和程序计数器(PC)的表达式

|  |  |  |
| --- | --- | --- |
| 符号 | 含义 | 使用方式 |
| BASE | 返回基于寄存器的表达式中寄存器的编号 | :BASE:X，X为与寄存器相关的表达式 |
| INDEX | 返回基于寄存器的表达式中相对于其基址寄存器的偏移量 | :INDEX:X，X同上 |

##### 其他常用运算符

|  |  |  |
| --- | --- | --- |
| 符号 | 含义 | 使用方式 |
| ？ | 返回某代码行所生成的可执行代码的长度 | ?X，返回定义符号为X的行的可执行代码的字节数 |
| DEF | 判断是否定义某个符号 | :DEF:X，如果符号X已经定义，返回{TRUE}，否则{FALSE} |

##### 表达式中各元素运算次序的优先级

表达式中各元素的优先级如下：

* 括号的优先级最高；
* 相邻的单目运算符的运算顺序从右到左，单目运算符的优先级高于其他运算符；
* 优先级相同的双目运算符的运算顺序是从左到右；

### 1.1.3.ARM汇编器的伪操作(4.2)

伪操作不像机器指令那样在程序执行期间由机器执行，它只是为完成汇编程序做准备工作的，一旦汇编结束伪操作的使命就完成。在ARM中有如下几种伪操作：

#### 1.1.3.1.符号定义伪操作

用于定义全局变量的：GBLA, GBLL, GBLS；

用于定义局部变量的：LCLA, LCLL, LCLS；

用于对变量赋值的：SETA, SETL, SETS；

为通用寄存器列表定义名称的：RLIST；

#### 1.1.3.2.数据定义伪操作

用于为特定的数据分配存储单元，同时可完成已分配存储单元的初始化。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指令名称 | 功能 | 格式 | 示例 | 备注 |
| DCB | 分配一片连续的字节存储单元并初始化 | 标号 DCB 表达式 | Str DCB “hello world” | 该指令也可以用“=”代替 |
| DCW(DCWU) | 分配一片连续的半字节存储单元并初始化 | 标号 DCW(DCWU) 表达式 |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

#### 1.1.3.3.汇编控制伪操作

#### 1.1.3.4.框架描述伪操作

#### 1.1.3.4.信息报告伪操作

#### 1.1.3.5.其他伪操作

## 1.2.工作模式切换编程(4.5)

# 2.Embedded programming with the GNU toolchain

这个资料是从google上找到的，主要介绍了一些入门级的知识，刚好对于现阶段的水平很合适使用。可以再google上直接输入书名(Embedded programming with the GNU toolchain)，找到网站就可以阅读。暂时没有找到pdf或其他版本的下载版本，所以暂时就这么看了。

## 2.1.Ubuntu 12.04上qemu入门

### 2.1.1.安装

本书中使用的是qemu模拟硬件平台实现代码的功能测试，故需首先安装qemu。

#apt-get install qemu

执行该语句后，查看/usr/bin下面，可以看到qemu-system-i386等的命令，说明已经自动安装了很多qemu的功能。

但由于我们要使用的是ARM平台的模拟，网上搜索后发现需要使用qemu-system-arm命令，这里没有发现，所以继续搜索后发现需要执行如下命令：#apt-get install qemu-system。执行成功后，可发现已经有了“qemu-system-arm”命令。执行“qemu-system-arm –M ?”命令，可以查看这时候支持的模拟硬件平台。

如果安装过程中提示“could not initialize SDL”，可执行“apt-get install libsdl1.2-dev”解决。这个问题我安装时没有遇到，不知道这个解决方案是否可行，只是从网上看到有，就拿过来了。

### 2.1.2.基本使用

## 2.2.Hello Arm(本书中第3节)

### 2.2.1.程序及分析

这里写了简单的汇编程序，实现加法操作：

.text

.globl \_start

\_start:

mov r0, #5

mov r1, #4

add r2, r0, r1

stop:

b stop

可以通过arm-linux-as和arm-linux-ld命令编译和连接，最终生成可执行程序。当然，前提是安装了交叉编译工具链(coretex a9开发板使用的第3章)。

这里还介绍了新的命令：arm-linux-nm和arm-linux-objcopy。

使用arm-linux-nm命令可以查看一个elf文件的符号，arm-linux-objcopy可以将一个类型的文件转化成另一类型，例如可以讲elf转化成bin文件。详细使用可参见下面的makefile：

CROSS\_COMPILE = arm-linux-

CC = gcc

AS = as

LD = ld

NM = nm #可以察看elf文件中函数的地址

OBJCOPY = objcopy #可以将elf文件转化成bin文件，也可以做其他形式的格式转化

SRC = add.s

OBJ = add.o

TARGET = add.elf

BIN = add.bin

$(TARGET):

$(CROSS\_COMPILE)$(AS) -o $(OBJ) $(SRC)

$(CROSS\_COMPILE)$(LD) -Ttext=0x00000000 -o $(TARGET) $(OBJ)

nm:

$(CROSS\_COMPILE)$(NM) $(TARGET)

bin:

$(CROSS\_COMPILE)$(OBJCOPY) -O binary $(TARGET) $(BIN)

clean:

rm -rf \*~ $(TARGET) $(OBJ) $(BIN)

在执行arm-linux-ld命令进行连接操作的时候，使用了“-Ttext”属性，该属性可以强制使程序从指定的位置开始存放。

也很明显的可以看到，最终生成的bin文件肯定是16个字节。因为每条汇编指令在内存中肯定占用4字节，而这个add程序只有4条汇编指令，所以肯定是16个字节的长度。

### 2.2.2.在qemu中模拟运行该程序

ARM处理器重新上电后，将从地址0X0开始执行。本书中使用的是connex的模拟开发板，该板子上有一个16M的flash，这个flash的起始地址是0x0。所以存放在这个位置的指令将直接执行。

在qemu模拟硬件的时候，首先要指定一个文件用来模拟flash。这里创建flash文件使用了如下命令：

$dd if=/dev/zero of=flash.bin bs=4096 count=4096

$dd if=add.bin of=flash.bin bs=4096 conv=notrunc

第一条语句将生成一个全0的、大小是16M的(4096\*4096)flash文件，第二条语句将add.bin文件作为flash文件的一部分，放置在其最开始的位置。之后可通过如下语句模拟开发板的启动及执行：

$qemu-system-arm –M connex –pflash flash.bin –nographic –serial /dev/null

其中，“–M connex”指定machine是模拟的connex；“–pflash flash.bin”指定模拟的flash是flash.bin文件；“–nographic”说明不需要图形显示，直接命令行的方式显示即可；“–serial /dev/null”则说明串口连接到null上，也就是说没有串口的功能。

执行该命令后，正确情况下终端将显示“(qemu)”的状态，可以通过qemu的监控命令(monitor command)查看寄存器的状态：info registers。执行该命令后，可以查看所有的寄存器状态，发现r0=5, r1=4, r2=9，说明add程序在开机时就直接被执行，并成功修改了寄存器的值。

### 2.2.3.更多监控命令

help：显示帮助信息；

quit：退出qemu的模拟器；

xp /fmt addr：显示从addr开始的物理内存的内容？原文：physical memory dump from addr；

system\_reset：重启系统；

关于xp命令需要做明确说明：fmt参数由3部分组成：<count><size><format>。count标明要显示几条data；size说明每条data的长度，b说明是8bit，h说明是16bit，w说明是32bit，g说明是64bit；format说明以何种形式展示，x说明以十六进制的形式体现，d说明以有符号十进制展示，u说明以无符号十进制，o说明以八进制，c说明以char类型，i则说明以汇编指令的方式展示。

可以看出，如果使用format为i的形式就可以达到反汇编的目的。例如在这里我们使用：xp /4iw 0x0命令理论上就可以得到如下结果：

0X00000000： mov r0, #5

0X00000004： mov r1, #4

0X00000008： add r2, r1, r0

0X0000000c： b 0xc

但是我再实验的时候，得到的是全0的结果，无法得到上面的预期结果，当前还没有分析出原因，需要以后确认！

## 2.3.More Assembly Directives(chapter4 in book)

使用了两个例子展示更多的汇编指令：计算数组各元素的和，以及计算一个字符串的长度。

### 2.3.1.数组和的计算

本示例定义一个数组，计算数组中所有元素相加的结果。代码目录为：./Code/embedded programming with the gnu toolchain/More Directives/sum\_an\_array。

两个新指令：.byte和.align。

.byte用来声明长度为8bit的元素，后跟多个元素可以表示一个数组，在内存中这些元素是占用连续内存的。同样的有.2byte和.4byte分别用来声明占用16bit和32bit的元素。

可以使用各种方式声明变量，例如：.byte 10; .byte 0xa; .4byte 0XABABABAB; .byte ‘a’; .byte ‘z’ – ‘a’ + 1; .byte 0b11011101等。

由于指令的长度要求是32bit对齐的，所以产生了.align命令。使用该命令可以使长度不足32bit的默认填充到4byte后再执行下一条语句，当然，这条命令只有在byte/2byte类型的数据被使用的时候才有必要用到这个。

### 2.3.2.计算字符串长度

指定一个字符串，计算其长度。代码目录是：./Code/embedded programming with the gnu toolchain/More Directives/string\_length。

这个程序里主要出现了两个新指令：.asciz和.equ。

命令“.asciz”用在字符串定义上，定义的字符串将在内存中占用一串连续的空间，且汇编语言将自动为字符串的末尾添加一个nul符号(\0)；命令“.ascii”同样可以声明一个字符串，区别在于汇编将不会给它声明的字符串自动添加nul作为结尾。

.equ命令可以手动声明一个值到某一个变量中，这个值可以由多种表现形式(参见.byte的数字表现形式)。

## 2.4.Linker

### 2.4.1.Using RAM

对于开发板(嵌入式设备)来说，flash是一个有用的第二级别的存储设备，就像硬盘。其本身是一个EEPROM设备，不方便存储变量。变量更适合直接存放在RAM里，这样它们也更方便的被改变。

connex里，从0XA000 0000开始的64M空间是RAM空间。

### 2.4.2.Linker概况

多文件的程序，每个文件单独编译得到obj文件后，使用linker将它们链接在一起并最终可执行。

linker在链接obj的时候，主要做两个工作：符号解析(symbol resolution)，重定位数据(relocation)。

### 2.4.3.符号解析

对于有多个文件组成的程序，在C语言中一个文件定义的函数，只要没有明确的指明类型是“static”的，就认为它是全局的，其他文件可以引用它；汇编则恰恰相反，只要没有明确声明是“.globl”的，就认为是静态的，除本文件中的函数外，其他文件中不能引用该方法。

上面已经知道，执行“arm-linux-nm”命令可以查看目标文件的符号表，这里对于“.o”文件该命令也适用。例如，执行“arm-linux-nm sum\_an\_array.o”后，得到的结果就是：

00000004 D \_start

00000000 d arr

00000000 t entry

00000003 d eoa

00000010 d loop

00000020 d stop

该符号表里的第二列代表的就是符号类型。“t”说明这个symbol是定义在代码段(data section)里的；“u”代表这个符号(symbol)没定义；“d”代表这个符号是数据段里的；大写的这些字母分别代表各自意义的全局符号(.globl)。

### 2.4.4.重定位

由于各文件生成的目标文件(.o)都是从0x0默认开始放置的，且都有自己的代码段和数据段定义，所以当连接他们的时候，必须做重定位。

汇编中，以“.text”声明的是代码段，“.data”声明的是数据段。代码段希望在内存中放置在只读区域，而数据段则希望在可读可写的区域，所以需要区别对待。需要注意的是，对于一个.o来说，其代码段和数据段都认为是从0X0开始的(虚拟地址？)。

#### 2.4.4.3.段合并(section merging)

linker将两个(或更多).o文件合并的时候，需要将他们的代码段、数据段分别重置位置。例如：a.o和b.o自身都认为代码段的起始地址是0x0，但当两者连接的时候显然就需要有一个.o将自己的起始地址设置到另一个的后面，只能有一个从0X0开始执行。

#### 2.4.4.4.段重置(section placement)

默认的段都是从0X0开始的，实际中我们可以通过linker使其更换开始地址，最典型的就是我们之前使用过的“-Ttext”参数，可以直接将开始位置更换到另外一个地址。

### 2.4.5.链接脚本文件(Linkder Script File)

程序员自己可以通过链接脚本文件来指定“段”合并的方式、在内存中放置的位置等。基本的链接脚本如下：

SECTIONS {

. = 0x00000000;

.text : {

abc.o (.text);

def.o (.text);

}

}

“SECTIONS”标识是最关键的标示符，表示这里要对段进行设置；“. = 0x00000000”表示下面的段的起始地址是0x0；下面代表代码段由abc.o和def.o两个目标文件的代码段组成。

如果所有.o文件的代码段都需要连接到一起，那么可以直接使用：.text : {\*(.text);}的形式来实现。当然，目标文件中还会有数据段，类似于代码段的写法，我们写出下面这个简单的脚本文件：

SECTIONS {

. = 0x00000000;

.text : { \* (.text);}

. = 0x00000400;

.data: { \* (.data);}

}

这样就表示所有的代码段、数据段都组合到一起，连接到一个文件里。代码段的起始地址是0X0，数据段的起始地址是0X400。如果不显式声明“. = 0x00000400;”，那么数据段将紧跟在代码段之后。

使用该脚本文件实现的例程，代码目录是：./Code/embedded programming with the gnu toolchain/Linder\_script\_file/linker\_script\_file。

## 2.5.数据段置于RAM中

### 2.5.1.RAM的不稳定性(volatile)

众所周知，RAM是一种不稳定存储设备，一旦断电，其上的所有内容将瞬间丢失。也同样因为这个，我们不能在上电是就直接指定某块区域的值，而必须上电后对这块区域赋值。

换句话说，所有的数据、代码等在上电前必须保存在flash上；上电后，使用一段代码将要放到ram上的数据拷贝到RAM上去。这样对于一个程序的数据段(数据段在RAM上效率会提高)来说，就存在两个地址概念：LMA(Load Memory Address，flash上的存储地址)；VMA(Virtual Memory Address，Ram上的运行时地址)。

### 2.5.2.确定LMA

可以在链接脚本文件中通过AT关键字确定LMA和VMA之间的关系，例如脚本文件的一部分内容：

endtext = .;

. = 0xa0000000;

.data : AT(endtext) { \* (.data); }

这里就说明了endtext这个变量代表的地址是LMA，是从flash上下载数据的地址；而0xa0000000则是ram上存放该数据的地址，是VMA。

### 2.5.3.复制数据到RAM

要复制一段数据，显然需要下面几个不可缺少的因素：从哪里开始复制，复制到哪里去，复制多少字节。这里同样是这样：从flash的什么地址开始复制(flash\_sdata)，复制到RAM的什么地址上去(ram\_sdata)；复制多少字节(data\_size)。

这些信息可以在链接脚本文件中得到：

SECTIONS {

. = 0X00000000;

.text : {

\* (.text);

}

flash\_sdata = .;

. = 0XA0000000;

ram\_sdata = .;

.data : AT (flash\_sdata) {

\* (.data);

}

ram\_edata = .;

data\_size = ram\_edata - ram\_sdata;

}

通过该文件可以得到所需要的参数，在汇编程序开始的时候，直接执行复制操作，将数据段复制到RAM上，就可以实现数据在RAM上的执行程序。

具体程序代码路径：./Code/embedded programming with the gnu toolchain/ Data\_in\_RAM /data\_in\_ram。

## 2.6.异常处理

前面几节讲的内容都存在一个主要的bug，因为内存映射(memory map)的前8个word是要保留给异常向量表的！当一个异常发生的时候，cpu将跳转到这8个位置执行。各异常对应的地址如下：

|  |  |
| --- | --- |
| 异常名称 | 地址 |
| 重启(Reset) | 0X00 |
| 未定义(Undefined Instruction) | 0X04 |
| 软中断(Software Interrupt, SWI) | 0X08 |
| 预取址中止(Prefetch Abort) | 0X0C |
| 数据中止(Data Abort) | 0X10 |
| 预留(Reserved, not used) | 0X14 |
| 中断(IRQ) | 0X18 |
| 快速中断(FIQ) | 0X1C |

在这些地址处要写入对应的跳转程序，才能使处理器在发生异常、跳转到这些地址时，有正确的处理程序。

上面章节中介绍的所有程序都可以添加上下面这种处理方式，以增加异常处理的功能。首先在汇编程序中要添加如下处理语句：

.section “vectors”

reset: b start

undef: b undef

swi: b swi

pabt: b pabt

dabt:b dabt

nop

irq: b irq

fiq: b fiq

添加这样的代码后，异常处理将成为无限循环。要查看是发生了哪种异常，可以通过查看当前的pc寄存器的值。注意的是，重启异常发生的时候，调用的是start功能。

为了使这些程序可用，链接脚本文件也要做出修改：

SECTIONS {

. = 0x00000000;

.text ： {

\* (vectors);

\* (.text);

……

}

……

}

两点需要明确：vectors的设置必须在所有其他段之前；vectors必须在0x0地址开始。

## 2.7.开机执行C程序(C Startup)

想让C语言编译得到的程序像之前汇编编译得到的程序一样直接开机执行时不可行的。对于C程序来说，还需要其他一些必不可少的设置。这一节通过数组相加的例子，介绍如何开机执行C语言的程序，程序路径：./Code/embedded programming with the gnu toolchain/ c\_startup。

首先明确，在将控制权转交到C代码前，必须要正确设置下面值：

* 栈(Stack)；
* 全局变量，包括定义的和未定义的；
* 只读数据(ReadOnly data)；

### 2.7.1.栈Stack

C语言中使用栈来存放：局部变量(local/auto variables)，函数参数(passing function arguments)，返回值地址(return address)等等。所以在将cpu的控制权转给C代码之前，提前设置栈是必须的！

ARM架构对于栈的实现完全是依赖于软件的，所以是高度宽泛的(flexible)一个标准，具体可以参考附录C中的“ARM Stacks”。

为了保证所有的编译器编译出来的程序都可以在ARM上执行，ARM制定了标准：ARM architecture procedure call standard，简称AAPCS。在这个标准里规定了R13这个寄存器将被用作栈顶指针，且栈是逆向增长的(full-descending)。

我们知道，connex的内存时从0XA0000000到0XA4000000的，可以将栈顶指针设置到0XA4000000这个地址，栈内容将顺势向上存储。只需要使用语句：ldr sp, =0xa4000000，就可以实现该功能。

需要注意，0XA4000000这个地址已经不是RAM的地址了，RAM正常的可用地址是到0X3FFFFFFF的。但我们可以使用这个地址作为栈顶指针，因为当执行push操作的时候，首先会将栈顶指针减1，再向这个位置存放内容。显然，第一个push进来的数据，就存放在了0X3FFFFFFF这个地址里。

### 2.7.2.全局变量

当C代码被编译的时候，已初始化了的全局变量将被放在.data这个段里，与其他的数据一起在上电的时候从flash上copy到RAM中。

而未初始化的全局变量则不同，它们默认被初始化成0。编译器会使用一个单独的段---- .bss ----来存放这些变量。上电后，需要将RAM中对应存放这些位置的区域置零。

### 2.7.3.只读数据

对于标记为const的全局变量，GCC会单独开辟一块区域----.rodata----存放。这块区域同样用来存放string类型的常量。

因为.rodata中的内容是不会被改变的，所以他们可以被放在flash上，要对链接脚本linker script做出对应的修改。

### 2.7.4.开机代码(Startup code)

为了适应上面所说的条件，对linker script做出修改：

SECTIONS {

. = 0X00000000;

.text : {

\* (vectors); 这里是异常向量表的设置，具体参见第6章

\* (.text);

}

.rodata : {

\* (.rodata); 这里设置只读数据

}

flash\_sdata = .;

. = 0XA0000000;

ram\_sdata = .;

.data : AT(flash\_sdata) {

\* (.data);

}

ram\_edata = .;

data\_size = ram\_edata - ram\_sdata;

sbss = .;

.bss : {

\* (.bss);

}

ebss = .;

bss\_size = ebss - sbss;

}

同样的，上电后的启动代码也需要做出修改才能适应需求，具体代码参见代码目录下的startup.s文件。

执行过之后，同样可以通过qemu查看结果。执行过程不再赘述，主要说明查看结果的方式：

查看要执行加法的数组数据可以：(qemu)xp /6dw 0xa0000000；查看加法后得到的结果：(qemu)xp /1dw 0xa000018.

## 2.8.附录A(ARM Programmer’s Model)

作为基础，一个ARM程序员需要知道最基本的ARM信息，这里介绍的是寄存器。

### 2.8.1.R0—R15

ARM中有16个寄存器在任何时候都是可用的，每一个都是32bit的。其中r0—r13是可以在程序中使用的，所有可以用r0进行的操作，都可以用其他的r1-r13的寄存器实现。r14和r15一般有专有用处：r14用来作为linker register，专门存放子程序的返回地址；r15用来做程序计数器(program counter)，包含了下一条即将要执行的指令地址。

当然，之前我们也说过，r13其实一般都用来作为栈指针使用，不作为普通寄存器使用。

### 2.8.2.CPSR

当前程序状态寄存器，Current Program Status Register，简称CPSR。它是一个专有的32bit的寄存器，包含了如下几部分：

* 状态标号，Condition Flags；
* 中断记号，Interrupt Masks；
* 处理器模式，Processer Mode；
* 处理器状态，Processer State；

这里只简单介绍了第一个部分----状态标号----的取值及含义：

|  |  |
| --- | --- |
| Flag | Meaning |
| C(进位，Carry) | Operation caused a carry，操作产生了进位 |
| O(Overflow) | Operation caused an overflow，溢出 |
| Z(Zero) | Operation resulted in 0 |
| N(Negative) | Operation resulted in a negative value，负数 |

## 2.9.其他相关

### 2.9.1.dd命令

dd命令时linux/unix下一个非常有用的命令，作用是用指定大小的块拷贝一个文件，并在拷贝的同时执行指定的转换。其参数分别列举如下：

|  |  |  |
| --- | --- | --- |
| 参数名 | 含义 | |
| if, input file | 指定源文件 | |
| of, output file | 指定目标文件 | |
| ibs/obs/bs=N | ibs，从if中一次读入N个byte，也就是说指定源文件一个块大小是N个字节； | |
| obs，一次向of中输出N个byte，指定目标文件的块大小； | |
| bs，同时设置输入/输出文件的块大小是N个byte | |
| cbs=M | 指定转换缓冲区大小；不理解 | |
| skip=S | 从输入文件开头跳过S个**块**后再开始复制 | |
| seek=S | 从输出文件开头跳过S个**块**后再开始复制，只有输出文件是磁盘或磁带的时候这个指定才有效 | |
| count=C | 指定要拷贝多少个**块** | |
| conv=condition[,cond,…] | 用指定的参数来转换文件，参数及含义如下： | |
| ascii | 将ebcdic转换成ascii |
| ebcdic | 将ascii转换成ebcdic |
| ibm | 将ascii转换成alternate ebcdic |
| block | 把每一行转换成长度为cbs，不足补空格 |
| unblock | 使每一行长度都是cbs，不足补空格 |
| lcase | 把所有大写字母转换成小写 |
| ucase | 把所有小写字母转换成大写 |
| swab | 交换输入的每对字节 |
| noerror | 出错时不停止 |
| notrunc | 不截短输出文件 |
| sync | 将每个输入块填充到ibs个字节，不足部分用空字符(NULL)补齐； |

# 3.ARM链接脚本文件

前面《ARM板级开发》2.4章节中简单介绍了一种链接脚本文件，这里详细介绍这部分内容。

就像makefile对于项目开发不可或缺一样，链接脚本文件对于板级的嵌入式开发同样重要。与ARM汇编的分类对应，GNU ARM汇编定义了linker script file格式的文件实现链接时的区域分配等功能；ARM汇编则提供了scatter file格式的文件实现同样功能。本部分根据网上资料，整理出详细的内容，帮助学习。

## 3.1.大家一起学scatter file

这是从网上找到的中文的资料，介绍说是讲了scatter file的格式和使用等，但是下载下来的只有这么多，可以看到后面有“待续”的部分。。。。但是作为入门的参考还是有价值的。

### 3.1.1.简介

#### 3.1.1.1.概述

Scatter file (分散加载描述文件)用于armlink的输入参数，它指定映像文件内部各区域的download与运行时位置。Armlink将会根据scatter file生成一些区域相关的符号，他们是全局的供用户建立运行时环境时使用。

注意：当使用了scatter file 时将不会生成以下符号 Image$$RW$$Base, Image$$RW$$Limit, Image$$RO$$Base, Image$$RO$$Limit, Image$$ZI$$Base, and Image$$ZI$$Limit。

#### 3.1.1.2.什么时候使用scatter file

当然首要的条件是你在利用ARM 汇编及其开发工具、编译链(armasm、armlink等)进行项目开发，下面我们看看更具体的一些情况：

* 存在复杂的地址映射：例如代码和数据需要分开放在在多个区域；
* 存在多种存储器类型：例如包含 Flash、ROM、SDRAM、快速SRAM。我们根据代码与数据的特性把他们放在不同的存储器中，比如中断处理部分放在快速SRAM内部来提高响应速度，而把不常用到的代码放到速度比较慢的Flash内。
* 函数的地址固定定位：可以利用Scatter file实现把某个函数放在固定地址，而不管其应用程序是否已经改变或重新编译；
* 利用符号确定堆与堆栈；
* 内存映射的IO：采用scatter file可以实现把某个数据段放在精确的地指处。

因此对于嵌入式系统来说scatter file是必不可少的，因为嵌入式系统采用了ROM，RAM，和内存映射的IO。

#### 3.1.1.3.scatter file 实例

##### 3.1.1.3.1.简单的内存映射

LOAD\_ROM(下载区域名称) 0x0000(下载区域起始地址) 0x8000(下载区域最大字节数)

{

        EXEC\_ROM(第一执行区域名称) 0x0000(第一执行区域起始地址) 0x8000(第一执行区域最大字节数)

       {

       \*(+RO(代码与只读数据))

       }

      RAM(第二执行区域名称) 0x10000(第二执行区域起始地址) 0x6000(第二执行区域最大字节数)

      {

       \*(+RW(读写变量), +ZI(未初始化变量))

      }

}

##### 3.1.1.3.2.复杂内存映射

LOAD\_ROM\_1 0x0000(下载区域一起始地址)

{

       EXEC\_ROM\_1 0x0000(第一执行区域开始地址)

       {

        program1.o(+RO) (program1.o内的Code与RO data 放在第一执行区域)

       }

       DRAM 0x18000(第二执行区域开始地址) 0x8000(第二执行区域最大字节数)

      {

       program1.o (+RW, +ZI) (program1.o内的RW data与 ZI data 放在第二执行区域)

      }

}

LOAD\_ROM\_2 0x4000(下载区域二起始地址)

{

       EXEC\_ROM\_2 0x4000

       {

       program2.o(+RO) (program2.o内的Code与RO data 放在第一执行区域)

      }

      SRAM 0x8000 0x8000

      {

       program2.o (+RW, +ZI) (program2.o内的RW data与 ZI data 放在第二执行区域)

      }

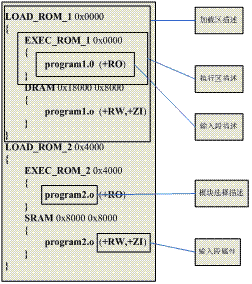
}

### 3.1.2.基本语法

#### 3.1.2.1.BNF符号与语法

* “” ： 引号内的符号保持其字面原意，如A”+”B表示A+B；
* A ::= B ：定义A为B；
* [A] ：标示可选部分，如A[B]C用来标示ABC或AC；
* A+ ：用来标示A可以重复任意次，如A+可标示A,AA,AAA, …；
* A\* ：同A+；
* A | B ：用来标示选择其一，不能全选。如A|B用来标示A或者B；
* (A B) ：标示一个整体，当和|符号或复杂符号的多次重复一起使用时尤其强大，如（AB）+(C|D)标示ABC,ABD,ABABC,ABABD, …；

#### 3.1.2.2.分散加载文件各部分描述

[](http://blog.photo.sina.com.cn/showpic.html#url=http://static7.photo.sina.com.cn/orignal/4c18b5bdd9c9b3a478256)

(2.1)

如图2.1所示为一个完整的分散加载脚本描述结构图。下面我们对图示中各个部分进行讲述。

##### 3.1.2.2.1.加载区描述

每个加载区有：

* 名称：供连接器确定不同下载区域；
* 基地址：相对或绝对地址；
* 属性：可选；
* 最大字节数：可选；
* 执行区域列：确定执行时各执行区域的类型与位置；

load\_region\_name (base\_address | ("+" offset)) [attribute\_list] [ max\_size ]

"{"

execution\_region\_description+

"}"

load\_region\_name：下载区域名称，最大有效字符数31。（并不像执行区域段名用于Load$$region\_name，而是仅仅用于标示下载区域）。

base\_address：本区域内部目标被连接到的地址（按字对齐）。

+offset：相对前一个下载区域的偏移量（4的整数倍，如果为第一个区域）。

##### 3.1.2.2.2.执行区描述

每个执行区有：

* 名称：供连接器确定不同下载区域；
* 基地址：相对或绝对地址；
* 属性：确定执行区域的属性；
* 最大字节数：可选；
* 输入段：确定放在该执行区域的模块；

exec\_region\_name (base\_address | "+" offset) [attribute\_list] [max\_size]

"{"

input\_section\_description+

"}"

exec\_region\_name：执行区域名称，最大有效字符数31。

base\_address：本执行区域目标要被链接到的位置，按字对齐。

+offset：相对于前一个执行区域结束地址的偏移量，4的整数倍；如果没有前继之能够行区域（本执行区域为该下载区域的第一个执行区域），则该偏移量是相对于该下载区域的基址偏移量。

attribute\_list：PI，OVERLAY，ABSOLUTE，FIXED，UNINIT。

* PI: 位置独立；
* OVERLAY: 覆盖；
* ABSOLUTE: 绝对地址；
* FIXED: 固定地址，下载地址与执行地址具有该地址指示确定；
* UNINIT: 未初始化数据；
* RELOC：无法明确指定执行区域具有该属性，而只能通过继承前一个执行区或父区域获得；

对于PI，OVERLAY，ABSOLUTE，FIXED，我们只能选择一个，缺省属性为ABSOLUTE。一个执行区域要么直接继承其前面的执行区域的属性或者具有属性为ABSOLUTE。

具有PI，OVERLAY，RELOC属性的执行区域允许其地址空间重叠，对于BSOLUTE，FIXED 属性执行区域地址空间重叠Armlink会报错。

max\_size：可选，他用于指使Armlink在实际分配空间大于指定值时报错。

input\_section\_description：指示输入段的内容。

##### 3.1.2.2.3.输入段描述

输入段：

* 模块名：目标文件名，库成员名，库文件名。名称可以使用通配符；
* 输入段名，或输入段属性(READ-ONLY,CODE)；

module\_select\_pattern

["("

("+" input\_section\_attr | input\_section\_pattern)

([","] "+" input\_section\_attr | "," input\_section\_pattern))\*

")"]

###### 3.1.2.2.3.1．模块名

module\_select\_pattern：选择的模块名称（目标文件，库文件成员，库文件），模块名可以使用通配符（\*匹配任意多个字符，？匹配任意一个字符），名称不区分字母大小写，它是供选择的样本。

例1：\*libtx.a (+RO)

libtx.a为threadX库文件。

例2：tx\_ill.o (INIT)

       tx\_ill.o为threadX中断向量目标文件。

###### 3.1.2.2.3.2．输入段属性选择子

input\_section\_attr：输入段属性选择子，每个选择子以”+”开头，选择子不区分大小写字符。

选择子可选RO-CODE，RO-DATA，RO（ selects both RO-CODE and RO-DATA），RW-DATA，RW-CODE，RW（ selects both RW-CODE and RW-DATA），ZI，ENTRY（ that is a section containing an ENTRY point）。

以下同义词可以选择：CODE (for RO-CODE)，CONST( for RO-DATA)，TEXT (for RO)，DATA (for RW)，BSS (for ZI)。

还有两个伪属性：FIRST，LAST。如果各段的先后顺序比较重要时，可以使用FIRST，LAST标示一个执行区域的第一个和最后一个段。

例1：os\_main\_init.o (INIT ,+FIRST)

FIRST表示放于本执行区域的开始处。

例2：\*libtx.a (+RO)

       RO 表示\*libtx.a的只读部分。

###### 3.1.2.2.3.3．输入段名

input\_section\_pattern：输入段名。

例1：os\_main\_init.o (INIT ,+FIRST)

INIT 为os\_main\_init.o的一个段。

例2：os\_stackheap.o (heap)

       heap 为os\_stackheap.o的一个段。

例3：os\_stackheap.o (stack)

       stack为os\_stackheap.o的一个段。

### 3.1.3.提高篇

#### 3.1.3.1.在scatter file中指定胶合段

胶合段用于实现ARM代码到Thumb代码的切换或者实现代码的长转移。使用scatter file可以指定怎样放置胶合输入段。通常，在scatter file中一个执行区域可以拥有胶合段选择\*(Venner$$Code)。

Armlink把胶合输入段放到拥有段选择符\*(Veneer$$Code)的执行区域中，这样做是安全的。

可能由于地址范围问题或者受执行区域大小限制无法完成把胶合段分配个某个执行区域。如果当发生胶合段无法加到指定区域时，他将会被加到那些包含了生成胶合段的可重载输入段的执行区域。

### 3.1.3.2.创建根执行区域

根执行区域就是指那些执行与加载时地址相同的区域。

当你为映像文件指定初始化入口点或者由于你只使用一个ENTRY导向符而使得连接器创建初始化入口位置时，你就必须确保该入口点位于根执行区域。如果入口点不在根执行区域，连接将会失败，连接器会报错。

如：ENTRY point (0x00000000) lies within non-root region ER\_ROM

可以通过以下方式实现在scatter file中指定根执行区域。

① 显示或缺省的指定执行区的属性为ABSOLUTE，同时使得加载区域与第一个执行区域具有相同的地址。

② 使用FIXED属性使得执行区域的加载地址与其执行时地址保持不变。

### 3.1.3.3.创建根执行区域

可以通过在scatter file中为某个执行区域指定FIXED属性来实现该区域加载于运行时地址保持不变。

FIXED可以用来在一个加载区中创建多个根执行区域。因此我们可以通过它实现把某个函数或一段数据放到目标地址，从而可以通过指针方便地访问该地址。比如，我们可以实现把常量表和checksum放到ROM上的固定地址处。

注意：

① 为了使得代码更加易于维护和调试，请尽量少使用scatter file来指定放置位置，而是应该尽可能多地让连接器来确定函数和数据的位置。

②

##### 3.1.3.3.1.怎样把函数或数据放到指定地址

  通常，编译器处理来自单个源文件的RO,RW,和ZI段。这些区域包括源文件的代码与数据。如果打算把单个函数或数据项放到某个固定地址，我们就必须让编译器单独处理这些函数和数据。

我么可以通过以下方式处理单个目标：

① 把函数和数据放到其源文件。

② 使用编译选项 –zo为每个函数单独生成一个目标文件。(参看ARM Compiler Guide)

③ 在C,C++源文件内利用 #pragma arm section 来生成多命名段。

④ 在汇编源文件内利用AREA 导向符来生成可重载段。

##### 3.1.3.3.2.怎样放置单个目标文件的内容

##### 3.1.3.3.3.怎样使用ARM的 section pragma

  通常把函数和数据放到其源代码文件，然后放到其目标文件的相应段中。然而，我们也可以#pragma 和scatter file实现单独处理某个命名段。

// file adder.c

int x1 = 5;                       // in.data

int y1[100];                      // in.bss

int const z1[3] = {1,2,3};           // in.constdata

int sub1(int x)                    // in.text

{

return x-1;

}

#pragma arm section rwdata = "foo", code ="foo"

int x2 = 5;                      // in foo (data part of region)

char \*s3 = "abc";                 // s3 in foo, "abc" in .constdata

int add1(int x)

{

return x+1;

}                             // in foo (.text part of region)

#pragma arm section code, rwdata   // return to default placement

FLASH 0x24000000 0x4000000

{

FLASH 0x24000000 0x4000000

{

init.o (Init, +First)                 ; place code from init.o first

\* (+RO)                         ; sub1(), z1[]

}

32bitRAM 0x0000

{

vectors.o (Vect, +First)

\* (+RW,+ZI)                     ; x1, y1

}

ADDER 0x08000000

{

adder.o (foo)                     ; x2, string s3, and add1()

}

}

### 3.1.4.待续......

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 |

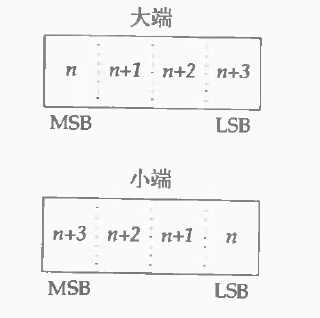
能够执行close，但还是提供了shutdown函数来关闭套接字的原因如下：

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；

C语言

# 2.C Primer Plus

阅读《C primer plus中文版第4版》时做的笔记。

## 2.1.文件输入/输出(第13章)

### 2.1.1.fopen()函数

fopen()函数打开文件，在“stdio.h”中声明，包含两个参数：第一个是要打开的文件的路径，字符串形式；第二个是文件打开的模式，同样是字符串模式。关于模式，参见下表：

|  |  |
| --- | --- |
| 模式字符串 | 含义 |
| “r” | 可以读取打开的文件 |
| “w” | 可以向打开的文件写入内容，写入前先将文件清空。文件不存在时，创建 |
| “a” | 向打开的文件写入内容，直接在文件末尾追加，不清空文件。文件不存在时，同样创建文件。 |
| “r+” | 可以读、写打开的文件。*该文件必须存在，否则打开失败。* |
| “w+” | 可以读写打开的文件，如果文件存在就清空文件并添加内容；否则创建文件。 |
| “a+” | 可以读写打开的文件。如果文件存在直接在末尾追加内容，否则先创建文件。**可以读取整个文件，但写入时只能追加。** |
| “rb”,”wb”,”ab”,  ”rb+”,”r+b”,”wb+”,  “w+b”,”a+b”,”ab+” | 类似以上的模式，只是使用二进制模式打开文件，而不是文本模式 |

**需要注意的是**，一旦使用了“w”相关的模式打开文件，文件内容将被清空！应慎重使用！

打开文件成功后，fopen()函数将返回一个文件指针(FILE \*fp, file pointer)，其他文件操作函数通过这个文件指针操作该文件。FILE在“stdio.h”中声明。

打开文件失败，fopen()函数将返回空指针(NULL，stdio.h中定义)。磁盘已满、文件名非法、存取权限不够或者硬件问题都会导致该函数执行失败。

**注意：**FILE \*fp = NULL;

fp = fopen(“test.txt”, “r+”);

fclose(fp);

fp = NULL;

这是应该遵守的编码风格，最后fp指向NULL可以避免野指针的出现，减少内存空间的浪费。

**二进制与文本模式的区别：**

* 在windows系统中，文本模式下文件以“\r\n”代表换行。如果以文本模式打开文件，并执行fputs等函数写入“\n”时，实际写入的是“\r\n”；
* 在linux系统中文本模式下，文件以“\n”代表换行。所以linux系统中文本模式与二进制模式没有任何差别。

**注意：**

**Windows上的路径和Linux上的是不同的表示方法，例如：**

**fopen(“/home/wujl/test.txt”, “r”);**

**fopen(“C:\\folder\\test.txt”, “r”); //这里使用了两个反斜杠，不是直接从目录路径复制过来就能直接使用的**

### 2.1.2.getc()和putc()函数

ch = getc(fp);将fp指向的文件中的一个字符读出，并赋值到ch中。通过：

while((ch = getc(fp)) != EOF) {}

语句，可以以字符为单位遍历整个文件。

putc(ch, fp);语句将字符ch的内容写入到fp指向的文件中。当fp为stdout时，该字符将被输出到标准输出(普遍意义上来说，就是显示器)上，这时候该函数的功能与putchar()函数完全相同。

### 2.1.3.fclose()函数

fclose(fp)关闭文件指针fp指向的文件。如果关闭成功返回0，否则返回EOF。

### 2.1.4.fprintf()和fscanf()函数

#include <stdio.h>

int fprintf(FILE \*stream, const char \*format[, arguments, ...]);

int fsancf(FILE \*stream, char \*format[, arguments, ...]);

fprintf函数将指定格式的内容输出到指定的文件中；fscanf函数从一个流中执行格式化输入，遇到空格和换行时结束输入。**注意，空格也是要结束的，相比较而言，fgets函数遇到空格是不结束的。**

例如：fprintf(fp, “%s%c\n”, str, ch);

fscanf(fp, “%d%s\n”, i, str);

fscanf函数的返回值是整形，表示成功读入的参数个数。

### 2.1.5.fgets()函数

char \*fgets(char \*buf, int bufsize, FILE \*stream);

该函数从文件指针stream中读取数据，每次读取一行。读取的数据保存在buf指向的字符数组中，每次最多读取bufsize-1个字符(第bufsize个字符将自动赋“\0”)。如果文件中的该行不足bufsize个字符，那么读完该行就结束。

函数成功返回buf，失败或读到文件结尾返回NULL。因此直接通过返回值判断函数是否出错时不正确的，应该借助feof函数或者ferror函数来判断。

如果使用fgets()函数读取文件，第一次读取的bufsize是5，而文件的第一行有10个字符(算上”\n”)，那么读取文件的指针会偏移到当前读取完的这个字符之后的位置。也就是说，第二次读取文件的时候，会继续读取其后的字符。

如果使用该函数读取文件的时候bufsize大于该行的字符总数加2(多出来的2个，一个是”\n”，另一个是字符串本身的结束符”\0”)，文件不会继续读取，在读取该行结束后直接结束读取，将文件指针指向下一行的开始位置。

与gets()函数相比，fgets()函数由于制定了bufsize的值，可以一定程度上减少内存越界、堆栈溢出等问题，所以更加安全。

### 2.1.6.fputs()函数

int fputs(char \*str, FILE \*fp);

该函数向指定文件写入一个字符串(不自动写入字符串结束标记符“\0”)。成功写入后，文件指针的位置会随之自动后移，函数返回一个非负整数；否则返回EOF(符号常量，值为-1)。

### 2.1.7.fseek()函数

int fseek(FILE \*stream, long offset, int fromwhere);

函数设置文件指针stream的位置。如果执行成功，stream将指向以fromwhere为基准，偏移offset(可以为负值)个字节的位置；如果执行失败(例如offset超过文件自身大小)，不改变stream的位置。

fromwhere的取值：

* SEEK\_SET，文件头；
* SEEK\_CUR，当前位置
* SEEK\_END，文件尾

执行成功，返回0；否则返回-1，并设置errno的值，可以使用perror()函数输出错误。

### 2.1.8.ftell()函数

long ftell(FILE \*fp);

返回文件指针当前位置相对于文件头的偏移字节数。

利用该函数可以得到一个文件的长度：fseek(fp, 0L, SEEK\_END); len = ftell(fp) + 1;

### 2.1.9.fread()和fwrite()函数

size\_t fread(void \*buffer, size\_t size, size\_t count, FILE \*stream);

该函数从文件中读取数据，最多读取count个元素，每个元素size字节。如果读取成功返回实际读取到的字节数；否则返回0.

size\_t fwrite(const void \*buffer, size\_t size, size\_t count, FILE \*stream);

该函数向文件写入一个数据块buffer，写入成功将返回实际写入的数据项个数count。

C++ Primer

本部分为阅读《C++ primer》的读书笔记。

# 1.变量和基本类型（第2章）

## 1.1.基本类型

Float尽可以表示到小数点后6位的精确度，double一般来讲都最少可以表示到10位以后。

三种数字表示方法：20(十进制)，024(八进制)，0X14(十六进制)。 注意，以“0”开头的表示是八进制的表示方法。

单词true和false是**布尔型**的字面值：bool test = true;

变量初始化C++支持两种形式：复制初始化和直接初始化。复制初始化使用等号(=)，直接初始化把初始化式放在括号里，例如：int a(10)是直接初始化，int a = 10是复制初始化。

在C++中需要理解“**初始化不是赋值**”。初始化是创建变量并给它赋初值，而赋值则是擦除对象的当前值并用新值代替。直接初始化语法更加灵活，而且效率更高。但对于内置数据类型的初始化来说，直接初始化和复制初始化的区别并不大。

## 1.2.变量

**声明和定义**

变量的定义用于为变量分配存储空间，还可以指定初始值。在一个程序中，变量有且只能有一个定义！

声明用于向程序表明变量的类型和名字，可以通过extern关键字声明变量名而不定义它。Extern声明不是定义，所以不分配存储空间。实际上，它只是说明变量定义在程序的其他地方。

程序中的变量可以声明多次，但只能定义一次。

只有当声明也是定义时才可以分配存储空间，例如extern double pi = 3.14;这条语句，尽管使用了extern关键字，但还是分配了存储空间，因为它执行了声明的同时完成了定义操作。但是要注意，**只有当extern声明位于函数外部时，才可以含有初始化式**。

Extern double pi = 3.14; //声明+定义，√

Extern double pi; //声明，√

Extern double pi = 3.1415; //重复定义，×

**Const限定符**

可以通过使用const限定符使一个变量被限制成常量，从而达到不能被修改的目的。注意，**由于修饰后的是一个常量，所以在声明的时候必须初始化**：const int a = 0;是正确的，而const int a;就是不正确的使用方式。

在全局作用域里定义非const变量时，它在整个程序里都可以访问，例如：

//file1.cpp

int a;

//file2.cpp

Extern int a;

…

与这种变量不同，除非特别说明，在全局作用域中声明的const变量时定义该对象的文件的局部变量。此变量只存在于那个文件中，不能被其他文件访问。

所谓特殊说明即在声明时就是用extern关键字说明可以外部访问，例如：

//file1.cpp

Extern Const int a = 0;

//file2.cpp

Extern const int a;

…

**枚举**

枚举的基本使用如下：enum week {Monday, tusday, sunday};默认的，第一个枚举成员赋值为0，后面的每个枚举成员比前面的大1.

枚举成员是常量。枚举成员的值可以相同，例如：enum points {a = 2, b, c= 3, d};这样定义下b和c的值显然都是3，是正确的定义方式。

每个enum都定义了一个新的类型。和其他类型一样，可以定义和初始化points类型的对象，也可以以不同的方式使用这些对象。枚举类型的对象的初始化和赋值，只能通过枚举成员或同一枚举类型的其他对象来进行：

Points p1 = a; //这是正确的赋值方式

Points p2 = 3; //不正确

P2 = Monday; //不正确，Monday不属于points这个枚举类型

P2 = b; //这样赋值是可以的，以points中的一个成员赋值

**头文件**

头文件一般包含：类的定义，extern变量的声明和函数的声明。

# 2.标准库类型(第3章)

## 2.1.标准库string类型

string类型支持长度可变的字符串，使用前应：

#include <string>

using std::string;

### 2.1.1.string对象的定义和初始化

string s1; //s1声明为一个空的字符串

string s2(s1); //s2声明为s1的副本；

string s3(“value”); //s3是值为value的字符串；

string s4(n, ‘c’); //s4初始化为字符c的n个副本

### 2.1.2.string对象的操作

#### size和empty操作

size操作获得string对象的长度，也就是该对象包含的字符的个数。该操作返回的是string::size\_type类型的结果，该类型是为了兼容各种机器的硬件而出现的。虽然可以直接使用int等类型存储size操作的结果，但仍旧建议使用该类型。例如：

for(string::size\_type ix = 0; ix < str.size(); ix++) {}

可使用empty操作判断string对象是否为空，返回true说明为空，否则不空。

#### 关系操作符

可直接使用“==”、“！=”、“<=”、“>=”、“<”、“>”等操作符直接对两个string对象进行比较操作，比较的规则与C语言strcmp()函数相同。

#### 赋值

string s1, s2;

s2 = “test”;

s1 = s2;

最后一句将s2赋值给s1.

#### 两个string对象相加

支持“+”和“+=”运算符，将两个string对象的内容进行连接，例如：

string s1(“Hello”), s2(“ world\n”);

string s3 = s1 + s2;

s1 += s2;

#### string对象和字符串字面值的连接

可以如下连接：

string s1(“Hello”), s2(“world”);

string s3 = s1 + “ ” + s2 + “\n”;

**注意：执行此类操作时，“+”操作符的左右操作数必须至少有一个是string对象！**

例如，该表达式就是不对的：string s4 = “Hello” + “ world\n”；因为它直接将两个字面值进行了连接，操作符的左右没有至少一个string对象。

##### 从string对象获取字符

string类型支持通过下标操作符[]访问其中的每个字符，下标需要使用size\_type类型的值来确定需要访问的字符的位置，例如：

string::size\_type idx = 1;

string s = “string test”;

cout << s[idx] <<endl;

当index超出下标范围时将引发溢出错误，所以一般应配合string.size()操作。

下标操作可以做左值，例如：s[idx] = ‘\*’;操作时允许的，将改变指定index的字符。

### 2.1.3.string对象中字符的处理

这些字符处理函数都在cctype头文件中定义。

|  |  |
| --- | --- |
| 函数名 | 功能 |
| isalpha(c) | 指定字符是否是字母 |
| isalnum(c) | 指定字符是否是数字或字母 |
| isdigit(c) | 指定字符是否是数字 |
| isxdigit(c) | 指定字符是否是十六进制数字 |
| tolower(c) | 指定字符转换成小写，如果本身就是小写不做改变 |
| toupper(c) | 指定字符转换成大写，如果本身就是大写不做改变 |
| islower(c) | 指定字符是否是小写字符 |

上面的判断类型的函数，返回0表示不满足判断条件，返回一个非零值(无任何意义)表示满足判断条件。

## 2.2.标准库vector类型

vector是同一种类型的对象的集合，每个对象都有一个对应的整数索引值。使用vector前需要包含头文件：

#include <vector>

using std::vector;

vector不是一种数据类型，而只是一种类模板，可以用来定义任意多种数据类型。例如，vector<int>和vector<string>就都是数据类型。

### 2.2.1.vector对象的定义和初始化

vector<T> v1; //默认构造函数，v1为空

vector<T> v2(v1); //v2是v1的副本

vector<T> v3(n, i); //v3初始化为包含n个元素i的vector对象

vector<T> v4(n); //n个该类型默认初始值的vector对象， 例如int的默认初始是0

注意：vector对象的动态增长效率极高，所以一般的使用方式是先初始化一个空的vector对象，再动态的向其中增加元素。

### 2.2.2.vector对象的操作

##### size和empty

类似于string对象，vector对象同样有size和empty函数，前者统计vector中有多少元素，后者判断是否为空。

size函数返回值是vector<T>::size\_type，使用同string。

注意，使用size\_type类型时，必须包含元素类型，类似于vector::size\_type的表述是错误的，必须是类似于vector<int>::size\_type的表述才是正确的。

##### 向vector添加元素

push\_back()函数用来接收一个新的元素，并将它插入到vector对象的后面，例如：

string word;

vector<string> text;

while(ci >> word) {

text.push\_back(word);

}

##### 下标操作

vector同样支持下标操作，类似于string类型的使用，可以读取指定下标的元素值。

但要向vector中增加元素的时候，必须使用push\_back函数，要直接对空的vector通过下标引用的方式进行赋值，是无法成功的。

必须是已存在的元素才能用下标操作符进行索引，通过下标操作进行赋值时不会添加任何元素。

## 2.3.迭代器iterator

除了使用下标可以访问到vector中的元素外，标准库还提供了另一种方法：迭代器。迭代器是一种检查容器内元素，并遍历元素的数据类型。

注意：标准库为每一种标准容器定义了对应的迭代器类型。并不是所有的容器都支持下标操作，但所有的容器都支持迭代器。现代C++更倾向于使用迭代器而不是下标访问容器元素，即使对支持下标操作的容器也是如此。

### 2.3.1.容器的iterator类型

每种容器都有自己的迭代器类型，例如vector相关的定义可以是：

vector<int>::iterator iter\_vect\_int；

这行语句就定义了一个名为iter\_vect\_int的迭代器，数据类型是由vector<int>定义的iterator类型。

### 2.3.2.begin和end操作

如果容器中有元素，begin()返回的迭代器指向第一个元素，例如：vector<int>::iterator iter = ivec.begin()；假设vector类型的ivec不空，iter就是ivec[0]。

由end操作返回指向容器的“末端元素的下一个”。

如果容器为空，begin和end指向的将是同一个对象。

end操作返回的迭代器并不指向vector中任何实际的元素，而只是起到哨兵(sentinel)的作用，标明我们已经处理完vector中的所有元素。

### 2.3.3.自增和解引用运算

迭代器可以使用解引用操作符(\*)访问迭代器指向的元素。假设iter指向vector对象ivec的第一个元素，那么\*iter和ivec[0]就指向同一个元素。语句“\*iter = 0”就是将这个元素的值赋0.

迭代器使用自增元素符向前移动，指向下一个元素，类似于指针的概念。例如：

for(vector<int>::iterator iter = ivec.begin(); iter != ivec.end(); ++iter) {

\*iter = 0;

}

该操作可以使用下标操作完成，但是现代C++建议使用上面的迭代器的方式实现。

### 2.3.4.迭代器的算术操作

除了自增操作外，vector迭代器(**其他容器迭代器很少支持该操作！**)还支持其他的算术运算，最常用的是：iter + / - n；该操作将直接将iter指向当前位置之前/之后n个元素的元素。

例如，vector<int>::iterator mid = ivec.begin() + ivec.size() / 2;将直接使迭代器指向最接近vector中间元素的位置。

## 2.4.标准库bitset类型

Bitset类用来处理二进制格式的数据，使用该类型需包含：

#include <bitset>

Using std::bitset;

### 2.4.1.初始化bitset对象

bitset<n> b; //b有n位，每一位都默认初始化为0

bitset<n> b(u); b是unsigned long型u的一个副本

bitset<n> b(s); b是string对象s中含有的位串的副本

bitset<n> b(s, pos, n); b是s中从pos位置开始的n个位的副本

**用unsigned值初始化bitset对象**

可以直接用unsigned long型的值作为bitset对象的初始值，该值将转化为二进制的位模式。

* bitset类型长度大于unsigned long值的二进制位数，多出的高位置0；
* bitset类型长度小于unsigned long值的二进制位数，只使用其低阶位，超过bitset长度的高阶位直接被丢弃；

例如，32位机器上，unsigned long型占用32个bit，那么：

* bitset<16> b1(0xffff); //b1有16个bit，而unsigned long有32个bit，所以截断，只将十六个1赋值到bitset中；
* bitset<32> b2(0xffff); //b2有32个bit，恰好存放unsigned long的值，所以b2[0]…b2[15]都是1，而b[16]…b[31]都是0；
* bitset<128> b3(0xffff); //b3有128个bit，长度大于unsigned long的长度，所以[0…31]存放的是0xffff的值，而其余高位存放的是0；

**使用string对象初始化bitset对象**

当用string对象初始化bitset对象时，string对象直接表示成位模式。从string对象读入位集的顺序是**从右向左**：

String str(“1100”);

Bitset<32> bitvec(str);

Bitvec的位模式第二和第三位都是1，其余位置都是0.如果string对象的字符个数小于bitset类型的长度，高阶位直接置为0.

不一定把整个string对象都作为bitset对象的初始值。相反，可以只用某个子串作为初始值：

String str(“1111111000000011001101”);

Bitset<32> bitvc5(str, 5, 4);

Bitset<32> bitvec6(str, str.size() - 4);

将使用str从str[5]开始的包含4个字符的子串，初始化bitvec5。从str[5]开始的4个字符分别是1、1、0、0，所以bitset从0到3是0011，剩余位都是0。

如果省略了第三个参数，意味着从开始位置一直到string对象的所有字符将初始化bitset。所以bitvec6将使用str的后4位初始化，故此bitvec第0到3位将是1011，其余位都是0.

### 2.4.2.bitset支持的操作

|  |  |  |
| --- | --- | --- |
| 函数 | 含义 | 说明 |
| b.any() | b中是否存在置为1的二进制位 | 有1返回true |
| b.none() | b中是否全是二进制0 | 全0返回0 |
| b.count() | 统计b中二进制1的个数 | 返回值类型为size\_t，该类型定义在cstddef头文件中 |
| b.size() | 计算b中二进制位的个数 | 返回值同样是size\_t类型 |
| b(pos) | 获得b中pos位置的二进制位的值 |  |
| b.test(pos) | 明确b中pos位置的二进制位是否是1 | 返回值是bool类型的true或false |
| b.set() | 设置b中所有位为1 |  |
| b.set(pos) | 设置b中pos位置的二进制位为1 |  |
| b.reset() | 设置b中所有位为0 |  |
| b.reset(pos) | 设置b中pos位置的二进制位为0 |  |
| b.flip() | 对b中所有二进制位取反 |  |
| b.flip(pos) | 对b中pos位置的二进制位取反 |  |
| b.to\_ulong() | 将b转换为unsigned long类型值返回 | 如果bitset对象包含的二进制位数超过u long的长度，运行时异常！ |

## 2.5.数组和指针

### 2.5.1.指针和const限定符

#### 2.5.1.1.指向const对象的指针

如果指针指向const对象，不允许使用指针改变它指向的值。为了保证这一点，C++强制要求指向const对象的指针必须也具有const属性：

const double \*cptr；

这里声明的cptr是一个指向const double类型的指针，const限制的是cptr所指向的对象，而不是cptr。cptr可以指向另外一个const的double对象，但不允许通过cptr修改所指向对象的值，例如：\*cptr = 40.00;该语句试图通过指针修改所指向对象的值的方法，是行不通的。

* 把一个const对象的地址赋给一个普通的、非const对象的指针也会有编译错误：const double pi = 3.14; double \*ptr = &pi的赋值方式不被接受，必须是类似const double \*cptr = &pi才可以；
* 不能使用void\*指针保存const对象的地址，必须使用const void \*类型的指针：const int a = 10; const void \*cp = &a;是正确的赋值方式；
* 允许将非const对象的地址赋给指向const对象的指针：double d = 3.14; const double \*cptr = &d;是可以执行的，但无法通过cptr这个指针修改d的值！

#### 2.5.1.2.const指针

const指针不同于指向const的指针，它是自身不能被修改！例如：int a = 0; int \*const p = &a;那么p就是指向int型变量的const指针，任何企图给const指针赋值的行为都会导致编译时错误。

const指针同样需要在定义的时候完成初始化操作。

const指针不能被修改，但它指向的值如果不是const类型的，可以利用其进行修改：int a = 10; int \*const p = &a; \*p = 11;将是可以正确执行的。

#### 2.5.1.3.指向const对象的const指针

const double pi = 3.14;

const double \*const pi\_ptr = &pi;

这里就既不能修改pi\_ptr所指向的值，也不能修改其自身的指向。

#### 2.5.1.4.指针和typedef

string s;

typedef string \*pstring;

const pstring cstr1 = &s;

pstring const cstr2 = &s;

string \*const cstr3 = &s;

三种定义方式的含义是完全相同的，都是定义一个const指针，指向一个string类型的对象。

### 2.5.2.C风格字符串

#### 2.5.2.1.创建动态数组

C语言使用一对标准库函数malloc和free在自由存储区(堆)上分配存储空间，而C++使用new和delete实现同样的功能。

##### 动态数组的定义

int \*pia = new int[10];

该语句分配了一个含有10个int型元素的数组，并返回指向该数组的第一个元素的指针，这个返回值初始化了指针pia。

##### 初始化动态分配的数组

动态分配的数组，如果数组元素具有类类型，将使用该类的默认构造函数实现初始化，否则无初始化：

string \*psa = new string[10];

int \*pia = new int[10];

第一个数组是string类型的，在分配空间后将直接调用string类型的默认构造函数依次初始化数组中的元素；第二个数组是int类型的，内置类型默认没有构造函数，所以数组没有初始化。

也可以在数组长度后加一对空的圆括号，对数组元素做初始化：

int \*pia = new int[10]();

圆括号要求编译器对数组做初始化，在这里就是将数组设置成全0的。但注意，动态分配的数组，采用这种方式，**只能初始化成为这种类型的默认值，不能使用自己定义的值进行初始化。**

##### 动态空间的释放

动态分配的内存在使用完后必须显示的调用释放函数delete执行释放操作，否则内存将逐步耗尽。例如：delete []pia;将释放pia动态分配的空间，注意：中间的一对方括号是必不可少的，否则将只释放pia指向的一个内存单元，其他单元将得不到释放而造成内存泄露。

#### 2.5.2.2.新旧代码的兼容

##### 混合使用标准库string类和C风格字符串

可以把C风格字符串用在任何可以使用字符串字面值的地方：

* 可以使用C风格字符串对string对象进行初始化或赋值；
* string类型的加法需要两个操作数，可以使用C风格字符串作为其中一个，也可以将C风格字符串用作复合赋值操作的右操作数；

反之则不成立：在要求C风格字符串的地方不可以直接使用string类型对象。例如：char \*ch\_arr = str;的操作就是不成立的，要执行此类操作需要调用string类提供的标准库函数：

const char \*ch\_arr = str.c\_str();

注意，c\_str()函数返回的是const char \*的类型。

##### 使用数组初始化vector对象

要使用数组初始化vector对象，必须指出用于初始化的第一个元素以及最后一个元素的下一个位置的地址：

const size\_t arr\_size = 6;

int int\_arr[arr\_size] = {0, 1, 2, 3, 4, 5};

vector<int> ivec(int\_arr, int\_arr + arr\_size);

这样就可以将int\_arr整个数组复制到ivec中。当然也可以只复制一部分数组内容：

vector<int> ivec2(int\_arr + 1, int\_arr + 4);

这个初始化创建了含有三个元素的ivec，分别是int\_arr[1..3]的副本。

# 3.顺序容器(第9章)

高质量程序设计指南(C++/C语言)

# 1.入门(第4章)

⊙﹏⊙b汗，这一章哥好多都不知道……

## 1.1.基本概念

### 1.1.1.main函数

main函数应该返回int，但具体返回什么类型可以由实现来定义。不过所有实现版本都应该至少允许以下两种形式：

int main();

int main(int argc, char \*argv[]);

### 1.1.2.内部规范

在C语言中，所有函数不是局限于编译单元(文件作用域)的static函数，就是具有extern连接类型和global作用域的全局函数。这种情况下，除了两个不同编译单元中的static函数可以同名，其他都不可能出现同名的情况。所以C语言采用了简单的函数名称区分规则：仅在所有函数前添加“\_”，从唯一识别函数的作用来讲，实际上和不添加前缀没有差别。例如，main函数在连接时会被命名为\_main。

在C++中，允许用户在不同的作用域中定义同名的函数、类型、变量等，这些作用域不仅限于编译单元，还包括class、struct、union、namespace等，甚至在同一个作用域中也定义同名的函数(重载函数)。这种情况下编译器如果依旧只增加前缀“\_”，毫无疑问将出现同名函数等。

因此，C++的编译器都需要进行“Name-Mangling”(名字修饰 / 名字改编)的动作，会将所属作用域的名称(class、namespace等)及重载函数的参数信息(参数类型和个数等)作为修饰，由此作为其内部名称。

由于C++标准没有规定Name-Mangling的规则，所以各个编译器的连接器不能兼容。

# 2.C++/C编译预处理(第9章)

## 2.1.文件包含

为了避免同一个编译单元、包含同一个头文件的内容超过一次(这将导致**类型重复定义的错误**)，需要在头文件中使用**内部包含卫哨**。

例如：

#ifndef \_\_TEST\_H\_\_

#define \_\_TEST\_H\_\_

…

#endif

## 2.2.宏定义

宏分为带参数的宏和不带参数的宏。

宏定义具有文件作用域，不论宏定义出现在文件的哪个地方(函数体内、类型内部、名字空间内部等)，在它后面的任何地方都可以引用宏。

* 宏定义不是C++/C语句，不需要以“；”结尾；
* 宏定义可以嵌套：

#define PI 3.14

#define PI\_2 (PI\*2)

* 宏不可以被调试，因为宏不进入符号表(符号表是编译器创建的，而编译时宏已经消失了)；即使宏替换之后出现了语法错误，编译器也会将错误定位到源程序中，而不是具体的某个宏定义中；
* 带参数的宏，宏体和参数应分别用“()”括起来，例如：

#define SQUARE(x) ((x)\*(x))

* 不在宏的参数列表中使用增量或减量运算符，例如“int n = 5; SQUARE(n++);”，得到的结果取决于编译器的不同而不同。针对复合表达式中子表达式的顺序，不同的编译器可能有不同的执行标准。
* 如果需要公布某个宏，那么将该宏定义放在头文件中；否则放在实现文件中；
* 给宏添加注释时应使用块注释(/\*\*/)，而不使用行注释(//)。有些编译器可能会把行注释也理解成宏的一部分。

### 2.2.1.带参数的宏

* **不转义**

#define PI 3.14

#define PI\_2 (PI\*2)

#define SQUARE(x, x) ( (x)\*(x) )

* **转义成字符串**

使用“#”将宏参数转换成字符串，简单说就是可以将输入的参数左右分别加上“”””，使其以字符串的形式输出。

#define TEXT(str) #str

printf(TEXT(vck)); //输出将是“vck”这个字符串；

#define CHECK(EXP) do{if(EXP) fprintf(stderr, “Error : ” #EXP “\n”);}while(0)

CHECK(divider == 0)在编译时将被替换为：

do{if(divider == 0) fprintf(stderr, “Error : ” “divider == 0” “\n”);}while(0);

* **贴合**

使用“##”可以将两个宏参数贴合在一起，并不在乎前后两个参数是否是输入参数，只要是参数即可。

#define CONS(a, b) int(a##e##b)

printf(“%d”, CONS(2,3)); //输出将是2000。因为转换后是int(2e3)，2e3代表的是2\*103；

* **变参宏**

使用“…”可以定义一个变参宏，“…”必须出现在最后一个参数的位置，例如：

#define myPrint\_0(format,…) fprintf(stderr, format, ##\_\_VA\_ARGS\_\_)

#define myPrint\_1(format,args…) fprintf(stderr, format, args)

但是这种变参宏，只有C99标准才支持；而且如myPrint\_0所示的，会有多余的逗号出现，必须使用“##”加以限制；

## 2.3.条件编译

#if #elif #endif

#ifdef #ifndef #endif

## 2.4.预定义符号常量

|  |  |
| --- | --- |
| 符号常量 | 含义 |
| \_\_FILE\_\_ | 源文件名字 |
| \_\_LINE\_\_ | 引用该符号的代码行号 |
| \_\_DATE\_\_ | 引用该符号的源文件被编译的日期(字符串) |
| \_\_TIME\_\_ | 引用该符号的源文件被编译的时间(字符串) |
| \_\_TIMESTAMP\_\_ | 引用该符号的源文件被编译的日期和时间(字符串) |
|  |  |

# 3.C++面向对象程序设计方法概述

## 3.1.动态特性

绝大多数情况下，程序的功能是在编译的时候就确定的，称之为静态特性；反之，如果实在运行时才确定的，称之为动态特性。

C++虚函数、抽象基类、动态绑定(Dynamic bingding)和多态(Polymorphism)构成了出色的动态特性。

### 3.1.1.虚函数

虚函数的声明方法是在函数原型之前加上关键字virtual。

假定基类Shape，派生类Circle、Rectangle、Ellipse等，每个派生类都能绘制属于自己的图形。但是希望每个子类使用同样的绘制接口，那么就最好在Shape中定义接口函数Draw()，并让程序在运行时动态的确定应该使用哪个派生类的Draw()函数。

基类Shape中定义虚函数Draw，并在派生类中重新定义Draw，并使其绘制属于自己的形状，这种方法叫做**覆盖(override)**。

一旦类的一个函数声明为虚函数，那么派生类的对应函数也自动成为虚函数，这样一级级的传递下去。虽然如此，为了增加程序的清晰性，应该为每一级派生层次中，将其显示的声明为虚函数(virtual关键字)。

### 3.1.2.抽象基类

如果将基类的虚函数声明为纯虚函数，那么该类就被定义为了抽象基类。纯虚函数是在声明时将其函数“初始化”为0的函数，例如：virtual void Draw(void) = 0;

将一个函数初始化为0，意味着函数的地址是0，这就告诉编译器：不要为该函数编址，从而阻止该类的实例化。在C++中，**只有虚函数**才能被初始化为0.

抽象基类的主要用途是“接口和实现分离”：不仅要把数据成员(信息)隐藏起来，还可以把实现隐藏起来，只留一些接口给外部调用。

## 3.2.C++对象模型

TODO，本部分内容，当前看的一知半解。不抄书，看懂了在做记录。

# 4.对象的初始化、拷贝和析构

每个类只有一个析构函数，但可以有多个构造函数：默认构造函数，拷贝构造函数，其他构造函数；也可以有多个赋值函数：默认赋值函数，拷贝赋值函数，其他赋值函数。如果程序员不显示的定义这些函数，编译器将自动创建public inline的函数，但一旦类中包含指针、类有多层继承关系等现象，默认函数按位赋值的缺点将导致程序出现不可预期的错误。

## 4.1.为什么需要构造和析构函数

当给一个对象分配好原始空间的时候，这个对象就已经创建起来了。但此时的内存，没有任何有意义的值被分配，直接使用极大可能会出现问题。因此，创建起原始的内存空间后，需要为这个对象做初始化，使其内存空间有意义。

初始化不同于赋值：

* 初始化指的是对象创建的同时，使用初值直接填充对象的内存空间，因此不存在数据类型转换等中间过程，也不存在临时变量；
* 赋值是对象创建后任何时刻都可以调用的函数，由于它调用的是“=”运算符，因此可能需要进行类型转换，也就会产生临时变量；

C++对象可以使用构造函数执行初始化动作：构造函数是对象创建时自动调用的第一个成员函数，也是为每个对象仅调用一次的成员函数。

所以构造函数的作用就是：当对象的内存分配完成后，把对象从原始状态转变成良好的、可用的内存状态。

## 4.2.构造函数的成员初始化列表

构造函数初始化列表的使用规则如下：

* 如果类存在继承关系，派生类可以直接在其初始化列表里调用基类的特定构造函数；
* 类的非静态const数据成员和引用成员，只能在初始化列表里初始化，因为他们只存在初始化语义，不存在赋值语义；
* 类的数据成员的初始化可以采用初始化列表和函数体内赋值两种，其效率并不相同：
  + 如果数据成员是内部数据类型，如int char等，那么两种赋值方式的效率基本相同；
  + 如果是自定义数据类型，如其他类，那么初始化列表赋值只需要调用该类的拷贝构造函数即可；但构造函数体内赋值的方式需要先为成员变量创建对象，在调用该对象的赋值函数，才能完成赋值动作。显然前者的效率更高。

## 4.3.构造函数和赋值函数的重载

构造函数分三类：默认构造函数，拷贝构造函数，其他构造函数；

默认构造函数是这样的构造函数：或者没有参数，或者所有的参数都有默认值；

拷贝构造函数是这样的构造函数：第一个参数为本类对象的引用、const引用、volatile引用或const volatile引用，且没有其他参数，或其他参数都有默认值；

要注意，拷贝构造函数的第一个参数，**一定是引用**，而不能是对象值！

如果没有显示的定义默认构造函数，却定义了带参数的其他构造函数，那么后者的存在就会阻止编译器产生前者！导致的结果，就是这个类没有默认构造函数。

## 4.4.如何实现派生类的基本函数

基类的构造函数、析构函数、赋值函数都不能被派生类继承。如果类之间存在继承关系，在编写上述函数时需要注意如下几点：

1. 派生类的构造函数，应在其初始化列表里显示的调用基类的构造函数；
2. 如果基类是多态类，那么必须把基类的析构函数定义成虚函数！这样可以实现动态绑定，否则很可能造成内存泄漏。
3. 在编写派生类的赋值函数时，要对基类的数据成员重新赋值，这可以调用基类的赋值函数来实现；

# 5.C++函数的高级特征

## 5.1.成员函数的重载、覆盖和隐藏

成员函数被重载的特征是：

* 具有相同的作用域(位于同一个类中)；
* 函数名字相同；
* 参数类型、顺序或数目不同(包括const和非const参数的区别)；
* Virtual关键字可有可无；

成员函数被覆盖的特征是：

* 不同的作用域(分别位于基类和派生类中)；
* 函数名称完全相同；
* 函数参数列表完全相同；
* 基类函数必须声明为virtual函数；

隐藏函数指的是派生类的成员函数遮蔽了同名的基类函数，特征是：

* 派生类的函数与基类的函数同名，但是参数有差异。此时，不论有无virtual关键字，基类的函数在派生类中都将被隐藏；
* 派生类的函数名称和参数列表，都与基类的相同，但是基类没有virtual关键字。这时候，基类的函数将被隐藏；

## 5.2.参数的默认值

参数默认值的使用规则是：

* 把参数默认值放在函数声明中，而不是定义中；
* 如果函数存在多个参数，那么参数只能**从后向前**依次默认！

## 5.3.运算符重载

在C++中，使用关键字operator加上运算符来表示函数，称作运算符重载函数。例如，复数相加的函数，可以定义为：

Complex operator+(const Complex &a, const Complex &b);

如果运算符被重载为全局函数，那么只有一个参数的运算符叫做一元运算符，有两个参数的是二元运算符；

如果运算符被重载为类的成员函数，那么一元运算符没有参数(++和—的后置版本例外)，二元运算符有一个参数，因为对象自己成为了左侧的参数；

### 5.3.1.运算符重载的特殊性

1. 如果重载为成员函数，那么this对象发起对它的调用；
2. 如果重载为全局函数，那么第一个参数发起对它的调用；
3. 禁止用户自定义出运算符集合中不存在的运算符！
4. 除了函数调用运算符”()”外，其他运算符重载函数不能有默认参数值；

### 5.3.2.不能重载的运算符

1. 不能重载“.”；
2. 不能重载反引用类成员指针“.\*”；
3. 不能重载作用域解析运算符“:：”；
4. 不能重载三元运算符“条件？A：B”；
5. 不能重载“sizeof”和“typeid”；
6. 不能重载“#”和“##”等预处理操作符；
7. 不能重载C++的新式类型转换运算符：static\_cast<>; dynamic\_cast<>;const\_cast<>;reinterpret\_cast<>；

### 5.3.4.重载++和—

当为一个类型重载“++”或“--”的前置版本时，不需要参数；而当为其重载后置版本时，需要一个int类型的参数作为标志，也就是哑元。

例如，前置版本：Complex operator++() {}

后置版本：Complex operator++(int) {}

当“++”和“--”应用于基本数据类型时，前置和后置效率并无明显差距；但当应用于用户自定义类型时，前置版本的效率将比后置版本高出很多。

## 5.4.函数内联

内联函数的目的是提高函数的执行效率，因为其省去了参数压栈、保存现场等工作。

与宏不同的是，内联函数是可以调试的。在程序的debug版本里，内联函数就是一个普通函数，因此可以调试；而在release版本里，内联函数才真正被内联进去，不可调试。

需要明确的是，inline关键字必须放在函数定义体之前才会生效，仅仅只是放在函数声明之前是不会生效的；如果声明时没有使用inline，只在定义时使用了inline，那么同样会按照内联函数处理。

所以说，内联是一种“用于实现的关键字”，而不是“用于声明的关键字”。

以下情况不应该使用内联：

1. 函数的代码过长，内联后将导致可执行程序的体积过大；
2. 函数体内有循环或其他复杂的控制结构，那么函数体的执行时间巨大，内联并无疑义；

## 5.5.Const成员函数

Const成员函数的定义，需要将关键字放在函数末尾，例如：int GetCount() const;

Const成员函数：

* 不能修改成员变量的值；
* 不能调用其它非const成员函数；

# 6.内存管理

GIT版本控制工具

# 1.GIT入门教程(廖雪峰)

GIT是分布式的版本管理工具，理论上来讲不需要中央服务器的参与，但一般为了资料共享的方便性总是会人为构造一个或几个服务器。

## 1.1.安装GIT

#apt-get install git

完成安装后，需要设置自己的名字和email地址，以作为这台机器上所有git仓库的配置信息，设置命令：

$git config –global user.name “Your name”

$git config –golbal user.email “Your [email@xxx.com](mailto:email@xxx.com)”

## 1.2.创建版本库

版本库又叫做仓库，英文名是repository。简单的理解，它就是一个目录，这个目录里的所有文件都可以被git管理起来，修改、删除、还原等操作都可以管理。

$mkdir test\_git

$cd test\_git

$git init

执行了该命令后，将在该文件夹中自动创建一个“.git”的隐藏文件，这几句是git来管理版本库的；而test\_git就认为这是一个git仓库，且提示你是一个空的仓库(empty git repository)。该文件非特殊情况，不要修改，否则导致git仓库被破坏。

将一个文件添加到git仓库中，需要两步：

$git add file.txt

$git commit –m “Create file.txt and save in git”

commit操作将文件提交到仓库，“-m”后的描述信息将记录在版本控制信息中，以供后续、其他人员查看文件记录。

可以使用“git add”命令添加多个文件，使用一次“commit”操作将其一次性的加入到版本控制仓库里。

## 1.3.时光机穿梭

命令：git status可以查看仓库当前的状态；如果使用该命令后，git告知有文件被修改过，可以通过命令“git diff mod\_file\_name”来查看修改的内容。

### 1.3.1.版本回退

要查看git仓库的历史记录，可以通过命令“git log”实现。该命令将记录显示出来，从最近到最远。可以通过加入参数，将主要信息在一行中显示出来以方便查看：“git log –pretty=oneline”。

要进行版本的回退，需要知道当前版本是哪个版本：在git中使用HEAD表示当前版本，也就是最新的提交，上一个版本就是HEAD^，上N个版本是HEAD~N。

可以使用“reset”命令将版本进行回退，例如要回退到上一个版本：“git reset –hard HEAD^”或“git reset –hard HEAD~1”都可以实现。

还可以有更加直接的方式，执行git log命令后，将每一次的版本修改信息显示的同时，还有一个十六进制的数字串，用来表示这个版本的唯一标识，假设这个值是“cb926e7ea50ad11b8f9e909c05226233bf755030”，那么可以通过命令“git reset –hard cb926e7ea50ad11b8f9e909c05226233bf755030”回到这个版本。

但每次都输入这么长的数字显然麻烦，且没有必要，因为每个版本的这个值都是不同的，所以只需要有选择的输入前几个数字，git可以自行查找匹配的版本并进行跳转，例如：“git reset –hard cb926e”。

如果从当前版本返回到了上一个版本，又想重新回到当前版本，就相当于要重返未来，需要使用“git reflog”命令找到“未来”版本的标示符，并通过该值实现跳转。

### 1.3.2.工作区和暂存区

所谓工作区就是存放在个人电脑上的、个人可见的目录，比如上面建立的test\_git文件夹就是一个工作区。

版本库的概念已经描述过，又称作仓库，英文名repository，是工作区中的隐藏目录“.git”。

GIT的版本库里包含了很多东西，最重要的就是称为stage(或者index)的暂存区，还有git自动为我们创建的第一个分支master，以及指向master的一个指针HEAD。

执行“git add”的时候，可以理解为将文件存放到了stage中；而执行“git commit”的时候，则是从stage中将文件传到了master中。

### 1.3.3.撤销修改

命令“git checkout -- filename.txt”可以将该文件在工作区的修改全部撤销，可以分为两种情况：

* 如果该文件在工作区修改了，还没有执行add命令添加到暂存区，那么执行该命令工作区的文件就会退回到与版本库一样的版本；
* 如果工作区修改了后使用add命令添加到了暂存区，那么执行命令就会退回到暂存区的状态；

总体来说，就是让工作区的文件回到最后一次“git commit”或者“git add”时的状态。

对于第一种情况，执行checkout命令就已经将工作区的修改撤销了，由于暂存区没有改变，所以就已经达到了目的；但对于第二种情况，执行该命令后只是达到了暂存区的状态，显然没有达到我们的目的，我们想要将暂存区和工作区的内容都还原到版本库里的。

要达到这个目的，可以首先执行“git reset HEAD filename.txt”命令，将暂存区的文件更新到版本库里的最新版本；再执行checkout命令，就可以将工作区同步暂存区，显然达到了我们的目的。

如果已经将修改的文件上传到了版本库里，想要撤销修改就需要参考“版本回退”里的指令了。

### 1.3.4.删除文件

使用“git rm”命令，可以从版本库里删除指定文件，当然执行完该命令后，同样要执行“git commit”命令才能达到最终效果。

如果在本地工作区里删除了文件，想要从版本库里恢复可以使用“git checkout -- filename”。

“checkout”命令实际上是用版本库里的版本替换工作区的，无论工作区是修改还是删除都可以“一键还原”。

## 1.4.远程仓库

可以自行搭建服务器作为远程仓库，每个人都访问这个服务器；也可以使用免费的github提供的远程仓库。这里讲的是后者。

首先登陆github网站，使用自己的邮箱注册一个账户。由于git仓库和github仓库之间的传输是通过ssh加密的，所以需要做如下设置：

1. 创建本机器的ssh key。在用户主目录下，看是否存在.ssh目录，如果有该目录再确认目录下是否存在id\_rsa和id\_rsa.pub两个文件，如果已经有了可以不做余下操作；否则在终端：$ssh-keygen –t rsa –C “[email@xxx.com](mailto:email@xxx.com)”。正常来说，执行结束后应该在主目录下找到前面说的几个文件。
2. 登录github后，打开account settings—ssh keys页面后：

点击“add ssh key”，填上自己的title，在key文本框里粘贴id\_rsa.pub文件的内容，如下图所示：



点击“add key”后，可见如下图所示的结果：



github允许一个人添加多个key，这样就可以保证每台电脑都可以向github上推送内容。但注意，免费版的github仓库是任何人都可以读的，如果有敏感信息需要保密，最好自行搭建一个中央服务器。

### 1.4.1.添加远程库(本地->远程)

首先在github网站上建立对应的远程仓库：



在点击了“Create a new repo”按钮后，就应该已经成功创建了一个新的仓库。但显然，这时候的远程仓库还是空的， 需要将本地的仓库与其做关联。

* git remote add origin [git@github.com:EricWuJl/test\_git.git](mailto:git@github.com:EricWuJl/test_git.git)：该命令将本地的git仓库与远程仓库进行关联；远程仓库的名字指定是origin(默认指定，可以修改，但不建议修改)；“EricWuJl”是在github网站上注册的账户名，要更改成自己的；
* git push -u origin master：将本地仓库的master分支推送到远程仓库；由于是第一次推送分支，所以加上“-u”参数，建立本地的master分支和远程master分支的关联，以后推送内容的时候该参数可以省略；
* git push origin master：master分支有了改动后，无需“-u”参数即可成功推送到远程仓库；

### 1.4.2.从远程库克隆(远程->本地)

先在github上创建一个仓库gitskills，作为远程仓库：



为了方便这里查看效果，选择的是“initialize this repository with a README”的选项，这样将直接建立一个拥有README.md文件的远程仓库gitskills。之后可以再本地对这个仓库进行克隆：

git clone [git@github.com:EricWuJl/gitskills.git](mailto:git@github.com:EricWuJl/gitskills.git)：该命令将克隆远程仓库的内容到本地仓库；

git支持多种网络协议，包括https等，但是使用ssh将拥有最快的交互速度。

## 1.5.分支管理

### 1.5.1.创建与合并分支

创建git的时候，git将自动创建一个分支：master。这时候，git用master指向最新的提交，再用HEAD指向master，就能很容易的确定当前分支以及其提交点：



每次执行“commit”命令的时候，master分支会向前移动；随着提交的增加，master分支也不断延长。

当创建新分支(例如dev)，git只是新建一个指针dev，它指向master相同的提交，再把HEAD指向dev就说明当前分支在dev上：



经过上面的操作，已经将当前的分支修改为了dev(HEAD指向dev，当前分支就是dev)。在此之后对工作区的修改和提交，将针对dev分支，比如新提交一次的情况：



当dev分支上的工作完成后，可将其合并到master分支上。git直接将master指向dev的当前提交，就完成了合并操作：



一旦确定dev分支已经完成了其使命，可以直接将其删除：



下面介绍实现这些功能的git指令：

* git checkout –b dev：创建分支dev，并将当前分支切换为dev；
  + git branch dev：只创建分支dev；
  + git checkout dev：将当前分支切换到dev；
* git branch：查看分支情况，以“\*”标记的是当前分支；
* git merge dev：合并指定分支到当前分支，这里就是将dev合并到活动的master分支上；默认采用的是fast-forward(快进模式)的合并，速度快，但不是所有情况下都可用；
* git branch –d dev：删除dev分支；

由于git对分支的操作非常快，所以git鼓励开发者使用分支完成任务，合并后再删除分支即可，这种开发更加安全可靠。

### 1.5.2.解决冲突

上面的合并操作，直接使用默认的“快进模式”，git就完成了所有工作。但是，如果出现如下情况：



新建了一个分支feature1，对文件进行了修改；回到master分支后，对文件同样进行了修改。之后执行“merge”命令试图将其合并：

$ git merge feature1

Auto-merging readme.txt

CONFLICT (content): Merge conflict in readme.txt

Automatic merge failed; fix conflicts and then commit the result.

可以看到，自动合并失败了，原因是你同时修改了两个版本。由此可见，对于同一个文件，两个分支进行了同一个位置的修改，将导致无法git无法进行自动的版本管理，必须执行手动合并操作：

查看导致冲突的文件readme.txt，发现其内部有用“<<<<<<<”、“=======”、“>>>>>>>”标记的内容，这部分就是不同分支对应的不同修改，将这部分内容手动进行修改后，再次执行“commit”操作，效果图：



这里其实不是执行了理论上的合并，而只是将需要合并的内容进行了修正后，重新在master分支上进行了一次提交。这次修正兼顾了feature1分支的修改，也就达到了合并的效果。

也可以通过命令：git log –graph –pretty=oneline –abbrev-commit，查看分支的合并信息。当然在执行了这些操作后，可以使用：git branch –d feature1操作将该分支直接删除。

### 1.5.3.分支管理策略

使用“fast forward”模式执行合并操作时，分支历史上无法查看到关于合并的信息；如果想要查看该信息，需要使用禁用该模式，具体来讲：

$git merge --no-ff -m "merge with no-ff" dev

由于要查看信息，所以“-m”参数肯定是需要的，方便查看信息的时候可以明确合并操作；参数“--no-ff”显然就是禁用“fast forward”模式。

git对于分支的管理，应有如下几个基本原则：

* master分支应该保证稳定。具体来说，应只是用该分支发布新版本，不在上面进行具体的开发工作；
* 开发工作宜在dev分支实现；每个开发者拥有自己的分支，并将自己的开发结果不断向dev合并；达到某一个稳定dev版本后，将dev向master合并即可。

### 1.5.4.Bug分支

在研发工作进行到一半的时候，现场的bug修复任务下发，需要马上修复bug。这时候直接将研发工作合并到分支里显然不合适，会造成其他人员拿到不能用的代码；放弃不理显然也不合适，会造成工作的浪费。这时候git提供了现场保存的功能。

执行命令“git stash”，git将当前分支上的内容保存起来。这时候执行“git status”可以发现没有了任何改动痕迹，是干净的。这时候就可以创建bug分支，并对bug进行修改后提交到指定分支上。

这时候需要恢复现场，执行“git stash list”命令可以查看有多少现场信息被保存，例如：

$ git stash list

stash@{0}: WIP on dev: 6224937 add merge

就说明有一个现场，id为“stash@{0}”，dev分支上的，被保存了。可以通过两种方式进行现场的恢复：

* git stash pop：恢复现场的同时，将stash里的内容也删除了；
* git stash apply：回复现场后stash的内容并不删除；git stash drop可以删除stash里的内容；

如果有多个现场信息被保存，恢复的时候可以指定现场号进行恢复：git stash apply stash@{0}，就是将stash@{0}的现场恢复。

**修复bug的标准流程应该就是如此：首先将没有完成的工作现场stash一下，然后新建分支修复bug，修复完成后将bug的分支删除，最后恢复现场。**

### 1.5.5.多人协作

当你从远程仓库克隆时，实际上git是把本地的master分支和远程的master分支对应起来了。且，远程仓库的默认名称是origin。

* “git remote origin”命令可以查看远程库的信息，要查看详细信息可以增加参数：“git remote –v origin”。
* git push origin master：将本地的master分支的内容推送到远程仓库对应的分支上；同理，“git push origin dev”就是推送dev分支的内容；
* git clone ...：将远程仓库的内容同步到本地工作区；这里只能同步master分支，如果需要dev等其他分支，需：git checkout –b dev origin/dev；
* git pull remote：假设你远程同步了dev分支，在修改过程中，另一个开发者对这个分支进行了修改并推送到了远程仓库，当你提交的时候，显然将面临版本的问题。这条指令就是处理这种冲突的。

该指令将最新的提交从当前的origin/dev分支上抓取下来，并与本地版本进行合并：如果提示该操作有错误，有可能出现1.5.2中说的冲突问题，需要手动解决；否则可再次尝试push操作，将其推送到远程。

* git branch --set-upstream branch-name origin/branch-name：如果git pull执行时提示“no tracking information”的错误，说明本地分支和远程分支没有建立连接关系，就需要执行本指令进行链接；

## 1.6.标签管理

发布一个程序版本时，通常先在版本库里打一个标签，将来要取某一个版本的程序，可以直接将打标签时刻的历史版本取出来。

### 1.6.1.创建标签

* git tag v1.0：对当前的分支、当前的版本，添加一个新标签“v1.0”。
* git tag：查看所有标签；
* git tag tagname commit\_id：在提交版本号是commit\_id处，新建一个标签；该功能可以为提交了很长时间的程序添加标签；该id可以通过git log操作获得；
* git tag –a tagname –m “infomation” commit\_id：在上一条指令基础上，增加了描述信息；
* git show tagname：查看指定标签的描述信息；

### 1.6.2.管理标签

* git tag –d tagname：删除本地标签，该标签尚未上传到远程仓库；
* git push origin tagname：将本地的某个标签(名字为tagname)推送到远程仓库；
* git push origin --tags：将本地的所有标签都推送到远程仓库；
* 删除远程标签需两步操作：
  + git tag –d tagname：先从本地删除；
  + git push origin ：refs/tags/tagname：再从远程删除；

## 1.7.自定义git

在1.1章节中只介绍了配置git的user.name和user.email属性，其实还有很多其他配置项可以配置。

例如：git config --global color.ui true，将在执行git命令的时候以不同颜色区分不同语句。

### 1.7.1.忽略特殊文件

有些文件，必须放在工作区，但又不能放到远程仓库里，例如数据库密码的配置文件、个人信息文件等。这些文件的存在将导致每次执行“git status”命令时总会显示“untracked files”。

为了解决这些问题git提供了“.gitignore”文件：在git工作区的根目录下创建该文件，把要忽略的文件名填进去，git就会忽略这些文件，不再提示上述问题。该文件有明确的格式信息，只需要按照规则填写、组合即可。

忽略文件的原则是：

* 忽略操作系统自动生成的文件，比如缩略图等；
* 忽略编译生成的中间文件、可执行文件等；
* 忽略包含个人信息的敏感文件，比如存放口令的配置文件等；

可以在：<https://github.com/github/gitignore> 上预览很多开源项目的“.gitignore”文件，并根据该类文件编写自己的忽略文件。

教程中并没有明确讲该文件的格式，初步观察应该是“#”后面可以跟类似于注释的内容，将需要忽略的文件直接放到文件中即可。

### 1.7.2.配置别名

可以使用简写代替原始的名字，这样可以照顾自己的简写并减少拼写错误出现的可能性。例如：git config --global alias.st status命令执行后，就可以执行“git st”，而不是“git status”查看效果。

还有很多其他缩写，例如：co----checkout；ci----commit；br----branch等。

### 1.7.3.搭建git服务器

# 2.GIT常用命令汇总

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Order** | **Usage(作用)** | |
| **ch** | **en** |
| **Create(创建)** | $git init | 新建本地库 | create a new local repository |
| $git clone *ssh://user@domain.com/repo.git* | 克隆远程库到本地 | clone an existing repository |
| **Local changes(本地更改)** | $git status | 工作区的文件的修改记录 | file change records in your working directory |
| $git diff *filename* | 指定文件的修改处 | changes to tracked file |
| $git add *filename* | 添加工作区文件到暂存区 |  |
| $git add . | 提交工作区中所有文件的所有改动 | add all current changes to the next commit |
| $git commit -m “*infomation*” | 提交当前暂存区内容 | commit previously staged changes |
| $git commit -a -m “*infomation*” | 提交所有工作区内容 | commit all local changes |
| **commit history(提交历史)** | $git log | 所有的提交记录，从最新的开始 | show all commits, start with newest one |
| $git log -p *filename* |  | show changes over time for a specific file |
| $git blame *filename* |  | who changed what and when |
| **branches&tags(分支和标签)** | $git branch | 显示所有分支 | list all exist branches |
| $git checkout *branchname* | 切换活动分支 | switch HEAD branch |
| $git branch *branchname* | 基于HEAD创建新分支 | Create new branch based on current HEAD |
| $git branch –d *branchname* | 删除一个本地分支 | delete a local branch |
| $git tag *tagname [commit\_id]* | 指定提交点，打标签 | Mark a tag in *commit\_id* |
| **update&publish(更新)** | $git pull *remotename* *branch* | 获取远程的最新修正并直接合并到HEAD | download changes and directly merge/integrete into HEAD |
| $git push *remotename* *branch* | 推送分支到远程 | publish local changes on remote |
| $git push remotename --tags | 所有标签推送到远程 | publish your tags to remote reop. |
| **merge (合并)** | $git merge *branchname* | 合并分支到活动分支 | merge branchname to HEAD |
| $git rm *filename* | 从版本库里删除文件 |  |
| **undo(撤销)** | $git reset --hard HEAD | 抛弃工作区所有改变 | discard all local changes |
| $git checkout HEAD *filename* |  |  |
| $git reset --hard *commit\_id* |  |  |
| $git reset *commit\_id* |  |  |

# 3.Github的使用

想要将代码共享在服务器上，以方便在不同的工作地点都能够对同一套代码进行管理，个人使用者不可避免的要使用到github。

Github官网：<https://github.com/>；

创建用户是必须的操作，不做赘述。

对于github上代码的上传和下载等操作，实际使用后作如下记录：

## 3.1.上传新的代码库

要将一个目录上传到github上，可以在linux上直接执行，也可以在windows上通过模拟终端程序来实现:

* linux上直接使用“sudo apt-get install git”按照git即可；
* windows上下载msysgit后安装，之后就可以在任何目录通过右键找到“git bash here”，打开后即可同样通过命令行执行git操作；

### 3.1.1.创建一个仓库

要上传一部分代码到github上，首先要在github上创建一个新的代码仓库，这样github才能够为这个新上传的代码分配一个ssh路径，才能够执行上传的动作。

在github上创建新的仓库极为简单：在个人主页中，找到“new repository”后，点击创建即可；

### 3.1.2.创建ssh key

可以有https和ssh两种方式与github交互，我使用的是ssh，较为推荐的也是ssh。

本地要与github进行ssh交互，首先要在本地创建一个ssh key，之后将这个key注册给github上已经创建好的代码仓库，具体操作如下：

#### 本地创建ssh key

使用如下命令，在本地创建ssh key：ssh-keygen –t rsa –C “wujl\_0351@163.com”

创建时，会提示输入用户名密码等信息，不做任何输入，直接回车即可；

一旦创建完成，将会在 ~/.ssh目录下生成两个key，查看id\_rsa.pub文件内容如下：

$ cat id\_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQD9A0uSZVuDtyxAsZwd/nAMwz/1djmhwlIs/KrnkuconlTBYvSNFdS2DozROC99Jf9YQ04qJ5RwH9p1RCLMpo9/q3dgasRMyvhCBg7cRxiaGTVt2gBE2SIm2OMiJ8zYCntkkTfzan7q+A5ldpFGjS5HTZZC1kDUP4fMj+L8MLP0WDDYximlSwMuMEj2VVbMKin8dm5Ou+R1Sxi+bf2S/k9IcLtLmU2noBjBLeSu9cu/0bloHZ0lHbMMSdTa6I2kMsE0MffOwSA6lKzj9i2vwFPzsx94iBhIsoCTW/dB0Y2LxWXNFE32CDjCNZ4AwRJreT1OJjF/+cXM0ZIniTTrnjK7 wujl\_0351@163.com

这个key的内容就是github中需要使用到的；

#### 将ssh key加入到代码仓库中

在github的个人页面中，选择“view profile and more”后，进入settings页面，选择“SSH and GPG keys”栏，在SSH keys中选择New SSH key按钮后，将提示输入一个名字和一个key；名字可以按照你对代码仓库的使用者来自行输入，key就要输入上述的id\_rsa.pub中的全部内容了。

注意要输入的是全部内容！包括最前面的“ssh-rsa”，也包括最后的邮箱名字！

#### 测试是否成功

使用如下命令可以测试ssh是否添加成功：ssh -T git@github.com

在没有成功添加key到github上的时候，其输出应该是：

$ ssh -T git@github.com

The authenticity of host 'github.com (192.30.253.112)' can't be established.

RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.

Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'github.com,192.30.253.112' (RSA) to the list of known hosts.

Permission denied (publickey).

在成功添加到github上之后，其输出应该是:

$ ssh -T git@github.com

Hi EricJinleiWu! You've successfully authenticated, but GitHub does not provide shell access.

### 3.1.3.本地代码提交

要将本地代码提交到github上，首先要将本地代码置入git管理中。

在要提交的代码目录中，执行如下命令：git init；执行该命令后，将在该目录下生成一个.git文件，这个文件中记录了git管理的所有信息；

之后再改目录中，执行：git add \*；可以将这个目录下的所有文件加入到git暂存区中，等待上传；

最后执行：git commit -m “”；就可以将这个目录中的内容，加入到本地的git管理中；

需要注意的是，由于windows、linux、macos对于一些符号的处理不同（最典型的就是换行符），因此执行add时如果出现大量的“LF will be replaced by CRLF in …”，就是不同平台导致的。

解决该问题的方法是：git config --global core.autocrlf false；这个命令可以将自动根据平台切换这些字符的处理方式这个特性置为false，那么再上传文件时就会将所有的文件按照原来的原始内容上传，不会做任何改动了。

### 3.1.4.将本地代码提交到github仓库中

首先，git config --global user.name “EricJinleiWu”

第二，git config --global user.email “wujl\_0351@163.com”

这两步操作执行完之后，用户信息将被记录，之后在github上查询到的commit记录中，都会有这两个信息。

第三，git remote add origin [git@github.com:EricJinleiWu/moCodes.git](mailto:git@github.com:EricJinleiWu/moCodes.git)

第四，git push -u origin master

执行完成后，github的个人主页上查看，此时就已经有了新提交的代码了。

## 3.2.从github上下载代码

下载已有的代码仓库到本地来，除创建用户、ssh key等动作外，需要执行的命令是clone，例如：git clone …/.git；

# 4.MyFAQ

## 4.1.同步最新代码到github出错

代码修改后，在本地执行了git add和git commit，将其作为本地版本记录存储成功后，执行git push命令向github同步时出错：

$ git push origin master

To github.com:EricJinleiWu/moCodes.git

! [rejected] master -> master (fetch first)

error: failed to push some refs to 'git@github.com:EricJinleiWu/moCodes.git'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

网上查询后发现出现该原因是没有远程同步github上的版本，需要首先：git pull origin master；之后再执行push命令即可成功。

出现这个的原因，是我在github上增加了一个readme文档，但是本地没有同步这个文件，导致出现了冲突。后来没有文件冲突的情况下进行push，都没有出现过这个错误。

VIM使用

# 1.VIM简明教程

网上博客找到的vim的教程，相对来说较为简单明了，网址：<http://blog.csdn.net/niushuai666/article/details/7275406>。该博客是翻译的国外牛人的，国外牛人的原地址：<http://yannesposito.com/Scratch/en/blog/Learn-Vim-Progressively/>。

其推荐的vim学习分成了四个等级：存活，感觉良好，觉得更好、更快、更强，使用vim的超能力。

## 1.1.存活

|  |  |  |
| --- | --- | --- |
| **命令符** | **含义** | **备注** |
| i | 从normal模式进入insert模式，光标位于当前位置 | “a”同样进入insert模式，光标位于下一个字符处 |
| x | 删除当前光标所在的一个字符 |  |
| :wq | 保存退出；:w，保存；:q，退出； |  |
| dd | 删除当前行，并将删除的内容复制到剪贴板 |  |
| p | 将剪贴板的内容粘贴到当前位置 |  |
| hjkl | 上下左右，也可以使用小键盘的上下左右键实现 | 推荐使用hjkl实现，但不强制 |
| :help <command> | 对某个命令的帮助信息 | 可以不跟command；退出需:q |

## 1.2.感觉良好

|  |  |  |  |
| --- | --- | --- | --- |
| **类别** | **命令符** | **含义** | **备注** |
| **各种插入模式** | a | 光标后插入 |  |
| o | 当前行后插入新行 |  |
| O | 当前行前插入新行 |  |
| **简单的移动光标** | 0 | 到行头 |  |
| ^ | 到本行第一个非blank字符 | blank指空格、tab、换行、回车等 |
| $ | 到本行行尾 |  |
| g\_ | 到本行最后一个非blank字符 | blank同上 |
| /pattern | 查找“值为pattern”的字符串 | 按n，可到下一个查找位置 |
| **拷贝/粘贴** | p | 将内容粘贴到当前位置之后 |  |
| P | 将内容粘贴到当前位置之前 |  |
| yy | 拷贝当前行，相当于ddP |  |
| **undo/redo** | u | undo |  |
| ctrl+r | redo |  |
| **保存等文件操作** | :e < filepath> | 打开指定路径的文件 |  |
| :saveas <path> | 将文件另存到指令路径 |  |
| :q! | 退出，不保存修改 |  |
| :bn，:bp | 同时打开多个文件，使用这两条命令进行切换 | 可以使用“:n”切换到下一个文件 |

## 1.3.更好、更快、更强

|  |  |  |  |
| --- | --- | --- | --- |
| **目的** | **命令符** | **含义** | **备注** |
| **更好** | .(小数点) | 重复上一次的命令 |  |
| N(command) | 重复某个命令N次 | 2dd，删除2行；3p，粘贴文本3次 |
| **更强** | : N | 到第N行 |  |
| gg | 到第一行 | 相当于: 1 |
| G | 到最后一行 |  |
| e | 到光标所在单词的结尾 | 这里的单词默认是字母、数字、下划线组成的(程序变量) |
| w | 到下一个单词的开头，见图1.3.1 |
| E/W | 功能同e/w，见图1.3.1 | 单词是由blank分隔的(程序语句) |
| **\*/#** | **匹配光标所在的单词，移动光标到下一处(\*)/上一处(#)** |  |
| % | 匹配括号移动，包括(、{、[ | 光标要移动到括号上 |
| **更快** | <start position> <command> <end position> | 命令与光标位置的组合使用 | 0y$:先到行头，在复制，复制到行尾；  ye：从当前位置拷贝到本单词的最后； |
| v | 可视化的选择 | 按v后移动光标，可看到文本被选择 |
| gU/gu | 选中内容变大写/小写 | 可用v可视化选中 |
| d | 删除选中内容 | 同上 |



图1.3.1 命令e/w与E/W的差别

排序和查找

# 1.基本排序算法

按照是否使用外存，排序算法可分为内部排序和外部排序两种。

内部排序又分为如下五种：

* 插入排序
  + 直接插入排序
  + 希尔排序
* 选择排序
  + 简单选择排序
  + 堆排序
* 交换排序
  + 冒泡排序
  + 快速排序
* 归并排序
* 基数排序

## 1.1.直接插入排序

### 1.1.1.基本思想

在要排序的一组数中，假设前面(n-1)[n>=2]个数已经是排好了顺序的，现在要把第n个数字插到前面的有序数组中，使得这n个数也是排好顺序的。如此反复循环，直到全部都排好了顺序。

### 1.1.2.实例

初始状态：57 68 59 52

处理：

1. 57 68 59 52 由于57小于68，不处理
2. 57 59 68 52 将59插入到有序数列中，保持有序状态
3. 52 57 59 68 将52插入有序数列

### 1.1.3.复杂度分析

时间复杂度应为O(n2)；在极端情况(数组本来就是有序的)下，可以达到O(n)的最好结果。

## 1.2.希尔排序(最小增量排序)

### 1.2.1.基本思想

希尔排序又称为缩小增量排序，是直接插入排序的改进和优化。

其基本思想是先将序列分成若干子序列分别进行直接插入排序，待整个序列基本有序时再对整体进行一次直接插入排序。

算法特点是子序列不是简单的“逐段分割”，而是相邻某个“增量”的记录构成一个子序列。一般增量序列的选择直接使用上次的一半，例如整个序列长度为N，则第一次的增量序列可以选择为N/2。并注意，最后一个增量必须是1，这也是结束条件。

### 1.2.2.实例

Org, n = 10: 57 68 59 52 73 61 29 84 65 32

Round 0, d = 5: 57---------------61

68--------------29

59-------------84

52---------------65

73---------------32

57 29 59 52 32 61 68 84 65 73

Round 1, d = 2: 57----59----32----68-----65

29----52---61----84-----73

32 29 57 52 59 61 65 73 68 84

Round 2, d = 1(结束条件): 直接插入排序

## 1.3.简单选择排序

### 1.3.1.基本思想

在要排序的一组数中，选出最小的一个数与第一个位置的数交换；然后在剩下的数中再找出最小的与第二个位置的数交换，如此循环到倒数第二个数和最后一个数比较为止。

### 1.3.2.实例

Org: 57 68 59 52

Round 0 : 52 68 59 57

Round 1: 52 57 68 59

Round 2: 52 57 59 68

## 1.4.堆排序

### 1.4.1.基本思想

当且仅当满足以下条件时，关键字序列K1—Kn可以称为堆：树中任一非叶子结点的关键字均不大于(不小于)其左右孩子节点的关键字。

用大顶堆排序的基本思想：

1. 先将输入的数字序列初始化成为大顶堆；
2. 将堆顶(也肯定是序列里最大的关键字)R[1]和序列最后一个关键字R[n]交换，由此得到新的无需区R[1]..R[n]，而R[n]显然已经是有序区；
3. 调整R[1]..R[n]为有序堆，再将其堆顶与最末的元素交换，循环执行，直至所有元素都执行过排序为止。

## 1.5.冒泡排序

### 1.5.1.基本思想

冒泡算法的运作如下(**从后往前**)：

1. 比较相邻元素，如果第一个比第二个大，就交换他们；
2. 对每一对相邻元素做如此操作，从开始第一对到结尾最后一对。执行完之后，最后一个元素应该是最大的数；
3. 针对所有的元素重复以上操作，除了最后一个；
4. 持续每次对越来越少的元素重复之前步骤，直到没有任何一对数字要比较；

### 1.5.2.实例

Org: 57 68 59 52

Round 0 : 52 57 59 68

Round 1: 52 57 59 68

Round 2: 52 57 59 68

## 1.6.快速排序

### 1.6.1.基本思想

快速排序采用了分治的策略解决问题，分治法的核心思想是：将问题分解成若干个规模更小但结构与原问题相似的子问题。递归的解决这些子问题，最终将子问题的解组合成为原问题的解。

设当前待排序的无序区为R[low…high]，利用分治法可以将快速排序的基本思想描述为：

1. 在数列中随机选取一个记录作为基准pivot(一般直接选取第一个)，以此基准将当前序列划分为左右两个较小的子区间Rl和Rr，并使左区间中记录都小于pivot，右区间中值都大于pivot。此时pivot已经处在了正确的位置上，所以无需再次参加后续的排序。
2. 通过递归方法对左区间和右区间再次进行划分；
3. 递归返回时，序列已经成为有序，排序结束。

这里注意两点：

* 一般pivot会直接选择第一个记录，但随机选取的方法能够更好的照顾到算法对所有类型数列的时间消耗，对各种类型数列可以提高排序的效率；不论pivot选择哪里，总会有特殊情况导致算法耗时达到O(n2)的级别。
* 由于普遍认为20以内的数组排序，直接插入排序的效率是最高的，所以当数组长度小于20时，可以直接采用直接插入排序进行排序，而不是再次执行递归。

## 1.7.归并排序

归并排序是将两个(或以上)有序表合并成一个新的有序表，即把待排序的序列分成若干个有序的子序列，再把有序的子序列合并成整体有序的排序过程。

归并排序是建立在归并操作基础上的算法，该算法是采用分治法的一个典型应用。

### 1.7.1.归并操作实例

归并操作指的是将两个有序序列合并成一个有序序列的方法，例如有数列{6, 202, 100, 31, 38, 8, 1}:

|  |  |  |
| --- | --- | --- |
| 归并次数 | 结果 | 比较次数 |
| 第一次归并 | {6, 202}, {31, 100}, {8, 38}, {1} | 3 |
| 第二次归并 | {6, 31, 100, 202}， {1, 8, 38} | 4 |
| 第三次归并 | {1, 6, 8, 31, 38, 100, 202} | 5 |

由上表可以看出，总的比较次数是：3 + 4 + 5 = 12.

### 1.7.2.归并操作算法描述

1. 申请空间，申请的大小应是两个已经有序的序列的长度之和，用来存放合并后的序列；
2. 设定两个指针，最初位置分别是两个已经排序序列的起始位置；
3. 比较两个指针指向的元素，选择相对小的放入合并后的空间，指针顺势移动到下一个元素；
4. 重复步骤3，直至某一个指针超出序列尾；
5. 将另一个序列剩下的所有元素直接复制到合并空间中；

## 1.8.基数排序

## 1.9.八种排序算法的总结

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 类别 | 排序方法 | 时间复杂度 | | | 空间复杂度 | 稳定性 |
| 平均情况 | 最好情况 | 最坏情况 | 辅助存储 |
| 插入排序 | 直接插入 | O(n2) | O(n) | O(n2) | O(1) | 稳定 |
| 希尔排序 | O(n1.3) | O(n) | O(n2) | O(1) | 不稳定 |
| 选择排序 | 直接选择 | O(n2) | O(n2) | O(n2) | O(1) | 不稳定 |
| 堆排序 | O(nlog2n) | O(nlog2n) | O(nlog2n) | O(1) | 不稳定 |
| 交换排序 | 冒泡排序 | O(n2) | O(n) | O(n2) | O(1) | 稳定 |
| 快速排序 | O(nlog2n) | O(nlog2n) | O(n2) |  | 不稳定 |
| 归并排序 | | O(nlog2n) | O(nlog2n) | O(nlog2n) | O(n) | 稳定 |
| 基数排序 | | O(d(r+n)) | O(d(rd+n)) | O(d(r+n)) | O(rd + n) | 稳定 |
| 基数排序中，r代表关键字的基数，d代表长度，n代表关键字的个数 | | | | | | |

## 1.10.效率比较

通过耗时测试，得出如下结果：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数组类型 | 数组长度 | 循环次数 | 排序算法 | 平均耗时(ms) |
| 顺序数组 | 5000 | 50 | 直接插入排序 | 0.003036 |
| 希尔排序 | 0.073832 |
| 简单选择排序 | 3.630109 |
| 堆排序 | 0.186747 |
| 冒泡排序 | 3.508217 |
| 快速排序 | 2.766099 |
| 归并排序 |  |
| 10000 | 10 | 直接插入排序 | 0.001236 |
| 希尔排序 | 0.055970 |
| 简单选择排序 | 2.896902 |
| 堆排序 | 0.090035 |
| 冒泡排序 | 2.286321 |
| 快速排序 | 2.277021 |
| 归并排序 |  |
| 反序数组 | 5000 | 50 | 直接插入排序 | 9.442874 |
| 希尔排序 | 17.744187 |
| 简单选择排序 | 3.761798 |
| 堆排序 | 0.177772 |
| 冒泡排序 | 9.382973 |
| 快速排序 | 2.726488 |
| 归并排序 |  |
| 10000 | 10 | 直接插入排序 | 7.351613 |
| 希尔排序 | 14.342757 |
| 简单选择排序 | 2.956789 |
| 堆排序 | 0.092651 |
| 冒泡排序 | 7.383245 |
| 快速排序 | 2.193542 |
| 归并排序 |  |
| 随机数组 | 5000 | 50 | 直接插入排序 | 4.614691 |
| 希尔排序 | 9.143959 |
| 简单选择排序 | 3.521202 |
| 堆排序 | 0.190860 |
| 冒泡排序 | 8.903948 |
| 快速排序 | 0.130144 |
| 归并排序 |  |
| 10000 | 10 | 直接插入排序 | 3.755292 |
| 希尔排序 | 7.189396 |
| 简单选择排序 | 3.819253 |
| 堆排序 | 0.108193 |
| 冒泡排序 | 7.143905 |
| 快速排序 | 0.069189 |
| 归并排序 |  |

测试平台：virtual box上装的ubuntu 12.04虚拟机，虚拟出768M内存；

# 2.基本查找算法

## 2.1.顺序查找

从表的一端开始，顺序的扫描表，如果找到要查找的值就结束，否则查找失败。

这种算法查找的效率低下，但适用于线性表的顺序存储结构和链式存储结构的查找。

## 2.2.二分查找

1. Mid = (low + high) / 2;
2. 比较待查值a与mid指向节点的值，如果相等则查找成功，返回；如果a<[mid]，则high= mid-1；否则low=mid+1；
3. 重复执行step1和step2。如果low>high时查找失败。

**该算法要求要查找的序列式有序序列！**

虽然二分查找的效率较高，但由于表本身要求按照关键字排序，且排序本身就耗时较大，所以二分法更加适合顺序存储结构，且是一经建立就很少改动的结构。

## 2.3.分块查找

其他相关算法知识

# 1.最大公约数和最小公倍数

辗转相除法，又称欧几里得算法(Euclidean algorithm)，是求两个正整数最大公约数的算法。

辗转相除法基于以下原理：两个整数的最大公约数等于其中较小的数和两数的相除余数的最大公约数。例如，252和105的最大公约数是21，而252/105=2余42，所以105和42的最大公约数也是21.在这个过程中，两个数中的较大的一直在减小，当余数变为0的时候，除数就是两个数的最大公约数。

最小公倍数和最大公约数的关系是：两数相乘的结果，除以这两个数的最大公约数，结果就是最小公倍数。

最大公约数：greatest common divisor

最小公倍数：least common multiple;

# 2.杨辉三角

所谓杨辉三角，就是类似于如下的数字排列组合：

1

1. 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

……..

编写程序时应注意到的规律：

1. 第一列和对角线元素都是1；
2. 其余数字都等于本列上一行数据与前一列上一行数字之和；

# 3.素数判断

素数(质数)指的是能被1和自身整除的数字。

素数判断只需判断[2, sqrt(n)]之间是否存在数字能够被n整除即可，无需判断sqrt(n)—n-1之间的其他数字，原因是如果有一个数字m能够被n整除，那么n/m和m之间一定有一个是存在于[2, sqrt(n)]之间的。

这种判断适用于小数，对于大数应有更加有效率的算法来进行判断。

LibCurl基本使用

# 1.概述

#include <curl/curl.h>

curl\_global\_init() ：初始化libcurl；

curl\_global\_cleanup()：释放资源占用；

curl\_easy\_init() ：得到easy interface指针；

curl\_easy\_setopt() ：设置传输选项；

curl\_easy\_perform() ：完成传输任务；

curl\_easy\_cleanup() ：释放内存；

# 2.各函数描述

## 2.1.curl\_global\_init函数

函数原型：CURLcode curl\_global\_init(long flags);

本函数只能使用一次，如需再次使用，需要使用curl\_easy\_cleanup函数后再次使用；

如果在调用curl\_easy\_init前还没有调用该函数，libcurl库将自动执行一遍该调用，但需要注意，该函数不是线程安全的！

所以多线程中，应当主动在主线程中调用该函数执行初始化，防止出现多线程不安全的情况。

Flags取值：

* CURL\_GLOBAL\_ALL：初始化所有可能的调用；
* CURL\_GLOBAL\_SSL：初始化支持安全套接字层；
* CURL\_GLOBAL\_WIN32：初始化win32套接字库；
* CURL\_GLOBAL\_NOTHING：没有额外的初始化；

## 2.2.curl\_global\_cleanup函数

函数原型：void curl\_global\_cleanup(void);

结束libcurl的使用时，调用该函数执行curl\_global\_init操作的工作清理。

## 2.3.curl\_easy\_init()函数

函数原型：CURL \*curl\_easy\_init(void);

该函数用来初始化一个CURL的指针(类似于FILE指针)，一般而言，该函数意味着一个会话的开始，返回一个easy\_handler，用在其他的easy系列的函数中。

调用结束后，使用curl\_easy\_cleanup函数可以反初始化该函数。

## 2.4.curl\_easy\_cleanup()函数

函数原型：void curl\_easy\_cleanup(CURL \*handle);

配合curl\_easy\_init函数使用，清空其资源占用；

## 2.5.curl\_easy\_setopt()函数

函数原型：CURLcode curl\_easy\_setopt(CURL \*handle, CURLoption option, parameter);

此函数告知curl库接下来要执行何种操作，比如要查看一个网页的html源代码等。

## 2.6.curl\_easy\_perform()函数

函数原型：CURLcode curl\_easy\_perform(CURL \*handle);

在执行完curl\_easy\_setopt函数后调用该函数，可以执行设置的需求。

## 2.7.curl\_easy\_getinfo()函数

函数原型：CURLcode curl\_easy\_getinfo(CURL \*handle, CURLINFO info, …);

该函数可以获取到curl会话中的附加信息。需要注意，只有在curl函数返回CURLE\_OK的时候，该函数才会有意义。该函数一般会在perform函数(例如curl\_easy\_perform)执行完成后被调用。

第二个参数的含义及其后紧跟的第三个参数的限制，如下所示：

### 2.7.1.CURLINFO\_SIZE\_DOWNLOAD

使用该选项作为第二个参数时，需要第三个参数时一个double类型的指针，这样会在一次传输结束后，将本次传输所下载的字节数赋值到指针所指向的double变量中。

这个字节数，只能反映最近一次的下载。

### 2.7.2.CURLINFO\_SPEED\_DOWNLOAD

使用该选项作为第二个参数时，传入double型指针作为第三个参数。这里的速度，指的是下载的平均速度，单位是**字节每秒**。

### 2.7.3.CURLINFO\_TOTAL\_TIME

使用该选项作为第二个参数时，需要传入double型指针作为第三个参数。这个参数记录了传输的总耗时(单位是**秒**)。

这里的总耗时，包括了域名解析、TCP连接建立等所有时间。

### 2.7.4.CURLINFO\_CONTENT\_TYPE

获得从http服务器中收到的header中的content-type信息。传入的第三个参数，是一个char \*\*类型的指针，例如：

char \*contype;

curl\_easy\_getinfo(pCurl, CURLINFO\_CONTENT\_TYPE, &contype);

### 2.7.5.CURLINFO\_RESPONSE\_CODE

获得http服务器返回的返回码，第三个参数应为long类型的指针。

一般而言，http服务器返回码是200的情况下，代表了操作正常。

# 3.curl\_easy\_setopt函数选项

## 3.1.CURLOPT\_URL

设置要访问的URL。

## 3.2.CURLOPT\_WRITEFUNCTION,CURLOPT\_WRITEDATA

回调函数原型：size\_t callback\_function(void \*ptr, size\_t size, size\_t nmemb, void \*stream);

回调函数在libcurl接收到数据后将被调用，因此函数大多用作数据保存的功能，例如处理下载文件。

CURLOPT\_WRITEDATA用于表明callback\_function中stream指针的来源。

如果没有通过CURLOPT\_WRITEFUNCTION设置回调函数，libcurl将提供一个默认的回调函数，这个回调函数简单的将数据打印在标准输出中。

如果通过CURLOPT\_WRITEDATA属性给默认回调函数传递一个已经打开的文件指针，数据将输出到该文件中。

## 3.3.CURLOPT\_HEADERFUNCTION,CURLOPT\_HEADERDATA

回调函数原型：size\_t function(void \*ptr, size\_t size, size\_t nmemb, void \*stream);

Libcurl一旦接收到http包头后将调用该回调函数。

CURLOPT\_HEADERDATA类似CURLOPT\_WRITEDATA。

## 3.4.CURLOPT\_READFUNCTION,CURLOPT\_READDATA

回调函数原型是：size\_t function(void \*ptr, size\_t size, size\_t nmemb, void \*stream);

Libcurl需要读取数据传递给远程主机时调用该函数。

CURLOPT\_READERDATA类似CURLOPT\_WRITEDATA。

## 3.5.CURLOPT\_NOPROGRESS,CURLOPT\_PROGRESSFUNCTION,CURLOPT\_PROGRESSDATA

与数据传输进度相关的参数。

CURLOPT\_PROGRESSFUNCTION指定的回调函数，正常情况下每秒被libcurl调用一次，为了使其被调用，CURLOPT\_NOPROGRESS必须设置成false；

CURLOPT\_PROGRESSDATA指定的参数将作为CURLOPT\_PROGRESSFUNCTION函数的第一个参数。

typedef int (\*curl\_progress\_callback)(void \*clientp,

double dltotal,

double dlnow,

double ultotal,

double ulnow);

## 3.6.CURLOPT\_TIMEOUT,CURLOPT\_CONNECTIONTIMEOUT

CURLOPT\_TIMEOUT设置文件传输的超时时间；

CURLOPT\_CONNECTIONTIMEOUT设置连接的等待时间；

## 3.7.CURLOPT\_FOLLOWLOCATION

设置重定位url。

## 3.8.CURLOPT\_RANGE,CURLOPT\_RESUME\_FROM

断点续传相关功能。

CURLOPT\_RANGE指定char \*参数传递给libcurl，用于指明http域的RANGE头域，例如：

* bytes=0-499：头500个字节；
* bytes=500-999：第二个500字节；
* bytes=-500：最后500个字节；
* bytes=500-：500字节以后的范围；
* bytes=0-0,-1：第一个和最后一个字节；
* bytes=500-600,601-999：同时指定多个范围；

CURLOPT\_RESUME\_FROM传递一个long参数给libcurl，指定希望开始传递的偏移量。

## 3.9.CURLOPT\_INTERFACE

指定网卡名字，或网卡ip值。

## 3.10.CURLOPT\_NOBODY

设置是否同步传输http的body，将该值设置为非零值，将不传输body，这个可以用作HEAD方式的传输实现。

# 4.curl\_easy\_perform函数说明

## 4.1.返回值

* CURLE\_OK：curl任务正确完成；
* CURLE\_UNSUPPORTED\_PROTOCOL：不支持的协议，由URL的头部指定；
* CURLE\_COULDNT\_CONNECT：不能连接到remote主机或者代理；
* CURLE\_REMOTE\_ACCESS\_DENED：访问被拒绝；
* CURLE\_HTTP\_RETURNED\_ERROR：http返回错误；
* CURLE\_READ\_ERROR：读取本地文件错误；

需要更加详细的错误描述字符串，可以通过const char \*curl\_easy\_strerror(CURLcode errornum)函数获取；

# 5.HTTP消息头

当使用libcurl发送http请求时，它会自动添加一些http包头，可以通过CURLOPT\_HTTPHEADER属性手动替换、添加或删除一些http包头。

* Host：http1.1版本都要求客户端请求提供这个信息头；
* Pragma=“no-cache”：表示无需缓冲数据；
* Accept=“\*/\*”：允许接收任何类型的数据；
* Expect：以POST的方式向http服务器提交请求时，libcurl会设置该消息头为“100-continue”，它要求服务器在正式处理该请求之前，返回一个“OK”消息。如果POST的数据很小，libcurl可能不会设置该消息头。

## 5.1.自定义选项

越来越多的协议都构建在http协议之上（如soap），libcurl对这些做了支持。

http支持get，head，post提交请求，可以设置CURLOPT\_CUSTOMREQUEST来设置自定义的请求方式(libcurl默认以get方式提交)：

curl\_easy\_setopt(easy\_handler, CURLOPT\_CUSTOMREQUEST, “MyOwnRequest”);

## 5.2.修改消息头

http协议提供消息头：请求消息头用于告诉服务器如何处理请求，响应消息头告诉浏览器如何处理接收到的数据。

struct curl\_slist \*headers = NULL;

headers = curl\_slist\_append(headers, “Server : who are you?”);

headers = curl\_slist\_append(headers, “X-silly-content : yes”);

curl\_easy\_setopt(easyhandler, CURLOPT\_HTTPHEADER, headers);

curl\_easy\_perform(easyhander);

curl\_slist\_free\_all(headers);

## 5.3.重设消息头

对于已经存在的消息头，可以重设其值：

headers = curl\_slist\_append(header, “Accept: Agent-007”);

headers = curl\_slist\_append(header, “Host: munged.host.line”);

## 5.4.删除消息头

对于已经存在的消息头，设置它的内容为空就可以删除该消息头：

headers = curl\_slist\_append(header, “Host:”);

# 6.获取http应答头信息

发送http请求后，服务器会返回应答头信息和数据，如果只是打印应答头的信息，可以通过：curl\_easy\_setopt(curl, CURLOPT\_HEADERFUNCTION,callback\_function);的方式来完成。

如果需要获取应答头中的信息如应答码、cookies列表等，需要如下函数：CURLcode curl\_easy\_getinfo(CURL \*curl, CURLINFO info, …);实现，其中info就是要获取的内容，罗列几个参数值：

* CURLINFO\_RESPONSE\_CODE：返回的应答码；
* CURLINFO\_HEADER\_SIZE：返回的头大小；
* CURLINFO\_COOKIELIST：cookie列表；

除了获取应答信息外，这个函数还可以curl的一些内部信息，如请求时间、连接时间等。

# 7.Curl无法正常工作

传输失败总是有可能的。

可以将CURLOPT\_VERBOSE属性设置为1，libcurl会自动输出通信过程中的一些细节，如果使用的是http协议，请求头/响应头也会被输出；将CURLOPT\_HEADER设为1，这些头信息将出现在消息的内容中。

# 8.密码

客户端向服务器发送请求时，许多协议都需要提供用户名和密码，libcurl提供了多种方式来设置：

## 8.1.CURLOPT\_USERPWD

参数形式：user:password；

函数：curl\_easy\_setopt(easy\_handler, CURLOPT\_USERPWD, “username:password”);

## 8.2.CURLOPT\_PROXYUSERPWD

访问代理服务器时，可能时时需要提供用户名和密码进行用户身份验证，这种情况下libcurl提供了另一个属性：CURLOPT\_PROXYUSERPWD实现：

curl\_easy\_setopt(easy\_handler, CURLOPT\_PROXYUSERPWD, “username:password”);

# 9.返回值

## 9.1.CURLE\_OK

Value = 0；

Desc：正确返回

## 9.2.CURLE\_OPERATION\_TIMEDOUT

Value = 28;

desc：超时

## 9.3.CURLE\_PARTIAL\_FILE

Value = 18;

desc：文件传输短于或大于预期部分文件，只有部分文件被传输；

## 9.4.CURLE\_TFTP\_NOTFOUND

Value = 68;

desc：tftp server上不存在需要下载的文件；

## 9.5.CURLE\_COULDNT\_CONNECT

Value = 7；

Desc ：http服务器无法连接

sqlite3

# 1.概述

## 1.1.linux安装

sudo apt-get install sqlite3

sudo apt-get install libsqlite3-dev

## 1.2.基本使用

### 1.2.1.创建数据库

sqlite3 test.db

### 1.2.2.创建表

sqlite> create table testTable(id integer primary key, value text);

该表包含两个字段：id，value。其中id是主键字段。

需要注意，在1.2.1中创建了数据库之后，最少要为该数据库创建一个表或视图，这样数据库才会保存到磁盘，否则数据库不会被创建。

如果需要判断表是否存在，可以使用：

sqlite> create table if not exists testTable(id integer primary key, value text);

那么只有在数据库不存在的情况下，才会创建表，否则不予创建；

### 1.2.3.写入数据

sqlite> insert into testTable(id, value) values(1, “Michale”);

sqlite> insert into testTable(value) values( “Pony”);

### 1.2.4.查询数据

sqlite> select \* from test;

可以设置格式化查询结果：

sqlite> .mode column;

sqlite> .header on;

sqlite> select \* from test;

其结果可能如下所示：

id value

--------- -------------

1. Michale
2. Pony

其中，.mode column设置为列显示模式，header将显示列名。

### 1.2.5.修改表结构

#### 增加列

sqlite> alter table testTable add column email text not null ‘’ collate nocase;;

#### 创建视图

sqlite> create view nameview as select \* from testTable;

#### 创建索引

create index test\_idx on testTable(value);

### 1.2.6.一些有用的sqlite命令

#### 显示表结构

sqlite> .schema [table]

#### 获取所有表和视图

sqlite> .tables

#### 获取指定表的索引列表

sqlite> .indices [table]

#### 导出数据库到sql文件

sqlite> .output [filename]

sqlite> .dump

sqlite> .output stdout

#### 从sql文件导入数据库

sqlite> .read [filename]

#### 格式化输出到csv格式

sqlite> .output [filename.csv]

sqlite> .separator ,

sqlite> select \* from test;

sqlite> .output stdout

#### 从csv文件导入数据库

sqlite> create table newtable(id integer primary key, value text);

sqlite> .import [filename.csv] newtable

#### 备份数据库

sqlite3 mytable.db .dump > backup.sql

#### 恢复数据库

sqlite3 mytable.db < backup.sql

# 2.C/C++使用sqlite

使用sqlite，需包含头文件：#include "sqlite3.h"

## 2.1.sqlite3\_open

函数原型：sqlite3\_open(const char \*filename, sqlite3 \*\*ppDb);

该函数打开一个指向sqlite数据库文件的链接，返回一个用于其他sqlite程序的数据库连接对象。

如果filename是NULL或“.memory”，那么该函数将会在RAM中创建一个内存数据库，这只会在session的有效时间内持续。

如果filename不是NULL，那么该函数将使用这个参数值尝试打开数据库文件。如果该文件并不存在，函数将创建一个新的命名为该名称的数据库文件并打开。

## 2.2.sqlite3\_exec

函数原型：sqlite3\_exec(sqlite3 \*pDb, const char \*sql, sqlite\_callback , void \*data, char \*\*errmsg);

该函数执行一条sqlite命令，sql命令由参数提供，可以将多个命令一起执行。

参数说明：

* 第一个参数是打开的sqlite3数据库对象；
* sqlite\_callback是一个回调函数，data作为其第一个参数；
* errmsg可以获取程序执行的错误信息；

该函数解析并执行sql参数所给的每条命令，直到字符串结束或执行出现错误为止。

### 2.2.1.回调函数

int sqlite3\_callback(

void \*data, /\*Data provided in the 4th argument of sqlite3\_exec()\*/

int n, /\*The number of columns in row\*/

char \*\*cont, /\*An array of strings representing fields in the row\*/

char \*\*colName);

如下所示：

static int selectCallback(void \*data, int nColumn, char \*\*columnValue, char \*\*columnName){

printf("data = [%s]\n", (**const** **char** \*)data);

**int** i = 0;

**for**(i = 0; i < argc; i++)

{

printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : NULL);

}

printf("\n");

**return** 0;

}

针对该函数的参数，进行如下说明：

1. data：在sqlite3\_exec中传入的void\*参数，通过该参数，可以传入一些特殊的指针(类、结构体)，然后在这里强制类型转换到对应的类型，再对该数据进行操作；
2. nColumn：代表了这一条记录有多少个字段(也就是有多少列)；
3. columnValue：保存了所有查询得到的数据，实际上，这是个一维数组，而不是二维数组，每个元素都是一个char\*的类型，相当于一个字符串，代表了一个value；
4. columnName：与columnValue相对应，表示这个字段的字段名称；

针对函数的返回值，作如下说明：

1. 返回0，sqlite3\_exec()函数将继续执行查询操作；
2. 返回非零值，sqlite3\_exec()函数将立即返回，并且返回SQLITE\_ABORT；

这里的执行逻辑是，假设需要有N条记录被select到，那么每一条记录被读取到后，都会先调用该回调函数，如果调用返回0，那么将继续执行查询操作，否则停止查询动作。

通常来说，只有在查询操作时，回调函数才是有意义的，insert等操作没有必要使用回调。

除了可以通过回调函数得到当前select的结果外，还可以直接调用函数sqlite3\_get\_table来得到当前表的内容，示例如下：

char \*errmsg = NULL;

**char** \*\*pDbTableResult = NULL;

**int** nRow = 0, nColumn = 0;

int ret=sqlite3\_get\_table(pDb, strSQL, &pDbTableResult, &nRow, &nColumn, &errmsg );

**if**(SQLITE\_OK != ret)

{

printf("sqlite3\_get\_table with strSQL [%s] failed, ret = %d, Msg is [%s], errmsg returned is [%s]\n", strSQL, ret, sqlite3\_errmsg(pDb), errmsg);

sqlite3\_free\_table( pDbTableResult );

sqlite3\_free(errmsg);

sqlite3\_close(pDb);

**return** ;

}

//Show this infomation

**int** i = 0, j = 0;

**for**(i = 0; i <= nRow; i++)

{

**for**(j = 0; j < nColumn; j++)

{

printf("%s\t", pDbTableResult[i \* nColumn + j]);

}

printf("\n");

}

sqlite3\_free\_table( pDbTableResult );

sqlite3\_free(errmsg);

sqlite3\_close(pDb);

## 2.3.sqlite3\_get\_table

可以通过回调函数获取表信息，也可以通过函数sqlite3\_get\_table函数查询：

int sqlite3\_get\_table(sqlite3 \*db, const char \*sqlCmd, char \*\*queryResult, int \*nRow, int \*nColumn, char \*\*errmsg);

第三个参数是一个一位数组，其内存布局是：字段名称，后面紧跟每个字段的值；

第四个参数是查询出多少行记录；注意，这里不包含字段名那一行！

第五个参数是查询出多少列(也就是多少字段)；

所以，queryResult时机上包含的字符串数量，应该是：(\*nRow + 1) \* (\*nColumn)，因为第一行的字段名没有计入到nRow里面去。

此外，执行结束后，需要调用：sqlite3\_free\_table(char \*\*queryResult);来释放资源。

## 2.4.sqlite3\_close

函数原型：sqlite3\_close(sqlite3 \*pDb);

该函数关闭之前调用sqlite3\_open函数打开的数据库对象。所有与连接相关的语句都应在关闭之前完成。

如果有正在执行的查询操作，sqlite3\_close()将返回SQLITE\_BUSY禁止关闭的错误消息。

## 2.5.错误码

执行sqlite函数的返回值有其特定含义，于sqlite.h中定义，如下：

|  |  |  |
| --- | --- | --- |
| 错误码 | 值 | 含义 |
| SQLITE\_OK | 0 | Successful result |
| SQLITE\_ERROR | 1 | SQL error or missing database  如：执行的sql语句有错误，重复创建表等； |
| SQLITE\_INTERNAL | 2 | Internal logic error in SQLite |
| SQLITE\_PERM | 3 | Access permission denied |
| SQLITE\_ABORT | 4 | Callback routine requested an abort  回调函数返回结果不为零； |
| SQLITE\_BUSY | 5 | The database file is locked |
| SQLITE\_LOCKED | 6 | A table in the database is locked |
| SQLITE\_NOMEM | 7 | A malloc() failed |
| SQLITE\_READONLY | 8 | Attempt to write a readonly database |
| SQLITE\_INTERRUPT | 9 | Operation terminated by sqlite3\_interrupt() |
| SQLITE\_IOERR | 10 | Some kind of disk I/O error occurred |
| SQLITE\_CORRUPT | 11 | The database disk image is malformed |
| SQLITE\_NOTFOUND | 12 | Unknown opcode in sqlite3\_file\_control() |
| SQLITE\_FULL | 13 | Insertion failed because database is full |
| SQLITE\_CANTOPEN | 14 | Unable to open the database file  例如：指定的数据库文件处于只读目录下； |
| SQLITE\_PROTOCOL | 15 | Database lock protocol error |
| SQLITE\_EMPTY | 16 | Database is empty |
| SQLITE\_SCHEMA | 17 | The database schema changed |
| SQLITE\_TOOBIG | 18 | String or BLOB exceeds size limit |
| SQLITE\_CONSTRAINT | 19 | Abort due to constraint violation  例如：指定了一个属性是primary key，重复添加相同的值给该属性； |
| SQLITE\_MISMATCH | 20 | Data type mismatch |
| SQLITE\_MISUSE | 21 | Library used incorrectly |
| SQLITE\_NOLFS | 22 | Uses OS features not supported on host |
| SQLITE\_AUTH | 23 | Authorization denied |
| SQLITE\_FORMAT | 24 | Auxiliary database format error |
| SQLITE\_RANGE | 25 | 2nd parameter to sqlite3\_bind out of range |
| SQLITE\_NOTADB | 26 | File opened that is not a database file |
| SQLITE\_NOTICE | 27 | Notifications from sqlite3\_log() |
| SQLITE\_WARNING | 28 | Warnings from sqlite3\_log() |
| SQLITE\_ROW | 100 | sqlite3\_step() has another row ready |
| SQLITE\_DONE | 101 | sqlite3\_step() has finished executing |
|  |  |  |

发现sqlite相关函数执行出错时，可以通过sqlite3\_errmsg(sqlite3 \*pDb)函数将当前的错误原因打印出来；

# 3.并发

## 3.1.多线程

sqlite支持三种线程模式：

* **单线程：**这种模式下，没有执行任何互斥，多线程使用将不安全。该模式下，sqlite会禁用所有的mutex锁，一旦并发使用就会出错。
* **多线程：**这种模式下， 只要一个数据库连接不被多个线程同时使用就是安全的。在sqlite源码中，是启用bCoreMutex，禁用bFullMutex。实际上就是禁用数据库连接和prepared statement上的锁，因此不能在多个线程中并发使用同一个数据库连接和prepared statement。
* **串行：**这种模式下，sqlite是线程安全的。源码中将启用所有的锁，包括bCoreMutex和bFullMutex，因为数据库连接和prepared statement都已经加锁，所以对这些的操作也都变成串行了。

这三种模式的设置方式：

|  |  |  |  |
| --- | --- | --- | --- |
| **模式** | **编译时设置**  **SQLITE\_THREADSAFE** | **初始化前设置**  **Sqlite3\_config()函数** | **备注** |
| 单线程 | 0 | SQLITE\_CONFIG\_SINGLETHREAD | Sqlite3\_config函数需要在初始化前进行设置，这里所说的初始化，指的是调用sqlite3\_initialize函数，而这个函数在sqlite3\_open时自动调用 |
| 多线程 | 2 | SQLITE\_CONFIG\_MULTITHREAD |
| 串行 | 1 | SQLITE\_CONFIG\_SERIALIZED |
|  |  |  |

因此，如果需要达到线程安全，需要将sqlite3设置为串行模式，一般而言，默认的编译产物就是串行模式。

在串行模式下，互斥可以达到**写写互斥、读写互斥；读读可以同时进行**。一旦资源正在被占用，那么执行sqlite3\_exec函数将返回错误值SQLITE\_BUSY，以此表示资源无法被使用，具体而言描述如下：

* 有写操作时，其他读操作被驳回；
* 有写操作时，其他写操作被驳回；
* 有读操作时，其他写操作被驳回；
* 有读操作时，其他读操作可以并发执行；
* 开启事务时，提交事务之前，其他写操作被驳回；
* 开启事务时，提交事务前，其他事务请求被驳回；

## 3.2.多进程

针对多进程，sqlite同样提供了互斥操作，其互斥级别同样是写写互斥、读写互斥，读读OK。但由于sqlite自身并不支持存储过程(procedure)，因此各个进程之间同时对同一个数据表进行写入时，极有可能造成数据写入混乱的情况，需针对性处理该情况。

针对这种情况，学术上可以通过其延时执行的功能来解决。通过微改sqlite内部的延时执行功能，数据库操作一个临时结果，等待一个周期检查对应的执行结果和预期结果是否一致。如果一直就把临时结果赋值为正式结果，否则就认为操作执行失败。最后设计出来的程序可能并不能保证一定是按照预期的顺序执行的，但最少可以保证结果是正确的。

## 3.3.超时处理

在多线程、多进程中，一旦资源正在被占用，另外的申请者再次请求资源可能会出现资源被锁住无法使用的情况，这时候的返回值将是SQLITE\_BUSY。本部分所述即为针对这种情况的解决方案，通用的解决方案有两个：

1. 使用进程或线程间的同步机制以避免该情况的出现：如使用信号量(semphore)、互斥锁(mutex)等，本部分属于在应用层自行处理资源占用逻辑，避免调用sqlite时出现SQLITE\_BUSY的情况；
2. 使用sqlite自己提供的两个busy handler函数，但需要注意的是，这两个函数会互相影响，设置一个的同时会清除另外一个！

这里只对第二种方法进行描述。

### 3.3.1.sqlite3\_busy\_handler

函数原型：int sqlite3\_busy\_handler(sqlite3 \*pDb, int(\*)(void \*, int), void \*)

不注册此函数时默认的回调函数是NULL，申请不到资源将直接返回；

函数可以自己定义一个回调函数，当出现资源申请失败的时候，执行该回调函数进行延时并返回非零重试本次操作，回调函数的第二个参数代表了因为本次BUSY而需要回调该函数的最大次数，如果达到了最大次数(可以由此得到最大超时时间)还没有得到资源，才真正返回SQLITE\_BUSY；

最后传入的参数将作为回调函数第一个参数被使用；

回调函数返回非零，数据库会重试当前的操作；返回0则当前操作返回SQLITE\_BUSY；

### 3.3.2.sqlite3\_busy\_timeout

函数原型：int sqlite3\_busy\_timeout(sqlite3 \*pDb, int timeout);

不调用该函数时，默认超时等待时间为0，申请资源失败将直接返回SQLITE\_BUSY。

注册该函数，定义一个timeout(**单位是毫秒**)，那么在timeout后会重试一次资源申请，此次申请如果仍旧失败，才会返回SQLITE\_BUSY。

但是针对该函数的使用需要明确如下：因为sqlite的源码中注册了一个默认的sqlite3\_busy\_handler( sqliteDefaultBusyCallback )，而这个回调函数在编译时可能使得传入的timeout必须要大于1000并且是其整数倍才有意义，因此该timeout的设置可能会超时时间较大，且依赖性较强。

因此使用3.3.1中所述回调函数更为优越。

# 4.事务

事务(transaction)是一个对数据库的执行工作单元。它是以逻辑顺序完成的工作单元或序列，可以由用户手动执行，也可以由数据库程序自动完成。

事务具有以下四个标准属性，简称为ACID：

* 原子性(Atomicity)：确保工作单位内的所有操作都能够完成，如果未能完成，事务会在出现故障时终止，之前的操作也会回滚到执行该事务之前的操作；
* 一致性(Consistency)：确保数据库在成功提交的事务上正确的改变状态；
* 隔离性(Isolation)：事务操作相互之间独立、透明；
* 持久性(Durability)：确保已经提交事务的结果或效果在系统发生故障的情况下依然存在；

## 4.1.命令行控制事务

使用下面的命令控制事务：

* BEGIN TRANSACTION：开始事务处理；
* COMMIT：保存更改，或者可以使用END TRANSACTIOIN命令；
* ROLLBACK：回滚所作的修改；

事务控制命令只能与DML命令(INSERT UPDATE DELETE)一起使用，不能在创建表或删除表时使用，因为这些操作在数据库中是自动提交的。

### 4.1.1.BEGIN TRANSACTION

事务可以使用“BEGIN TRANSACTION”或者“BEGIN”来启动。事务会持续执行下去，直到遇到下一个COMMIT或ROLLBACK命令。不过在数据库关闭或者发生错误时，事务也会回滚。

### 4.1.2.COMMIT

COMMIT命令用于把事务调用的更改保存到数据库中，它会把从上次COMMIT或ROLLBACK命令以来的所有事务保存到数据库。

COMMIT命令与“END TRANSACTION”作用完全相同。

### 4.1.3.ROLLBACK

ROLLBACK命令用于撤销尚未保存到数据库中的事务，只能用于撤销 自上次发出COMMIT或ROLLBACK命令 以后的事务。

## 4.2.C/C++接口控制事务

通常来说一次sqlite3\_exec就是一个事务，如果需要删除一万条数据，sqlite就做了一万次如下操作：开始新事务🡪删除一条数据🡪提交事务🡪开始事务……。这个操作对系统资源的占用是巨大的，因为时间都花在了开始事务、提交事务上。

这种情况下，可以将这些操作做成一个事务，这样一来可以减少开始事务、提交事务的时间消耗，而来一旦发生了错误还可以执行回滚操作。

事务的操作并没有特别的接口函数，只是一个普通的sql语句，如下实例：

int ret = sqlite3\_exec(pDb, “begin transaction”, 0, 0, &errmsg); //开始事务

int ret = sqlite3\_exec(pDb, “commit transaction”, 0, 0, &errmsg); //提交事务

int ret = sqlite3\_exec(pDb, “rollback transaction”, 0, 0, &errmsg); //回滚事务

# 5.数据同步

## 5.1.设置方式

PRAGMA synchronous=FULL; //其值是2

PRAGMA synchronous=NORMAL; //值是1

PRAGMA synchronous=OFF; //值是0

## 5.2.参数含义

FULL(2)，sqlite数据库引擎会在紧急时刻暂停，以确认数据已经写入到磁盘(sync函数)。这样在系统断电或者电源出问题时就能够确保数据库重启后数据不出问题，但这种模式下，很慢。

NORMAL(1)，大部分紧急时刻会暂停，但不像FULL模式下频繁。这种模式下又很小的几率是回出现由于电源等问题导致数据库损坏的情况的，但一般来说这种情况下硬盘都已经发生了故障。

OFF(0)，sqlite在传递数据给系统以后直接继续而不暂停。如果运行sqlite的应用程序崩溃，数据不会损伤，但在系统崩溃或写入时意外断电，可能导致数据库损坏。但是这种模式下，一些操作可能速度会有极大提升。

在sqlite2中，默认设置为NORMAL；在sqlite3中，默认设置为FULL。

**//TODO，这里设置为OFF的操作执行成功了，但在事务中并未发现明显的效率提高，需要以后明确该部分的用例实现**

FFMPEG

# 1.多媒体基础

视频文件的解析流程：

解复用是将源文件的音频流、视频流、字幕流等，分别解析出来；

解码是将解析到的文件，从压缩格式解析到原始格式，得到可以呈现的帧。

syslog

# 1.概述

在2001年定义的RFC3164中， 描述了BSD syslog协议，不过这个协议的很多内容都不是强制性的，大多是“建议”或“约定”，也由于这个规范出的比较晚，很多设备并不遵守或不完全遵守这个约定。

约定发送syslog的设备为Device，转发syslog的设备是Relay，接收syslog的设备是Collector。但这个划分并不绝对。例如，一个设备作为Relay接收了Device发送的syslog，但并没有转发，而是直接使用了，那他就同样是一个Collector。

syslog消息发送到collector的UDP的514端口，不需要接收方应答，RFC3164建议Device也是用514作为源端口。

syslog消息使用UDP协议发送，UDP报文不能超过1024个字节，并且全部由可打印的字符组成。完整的syslog消息，由三部分组成：PRI，HEADER，MSG。大部分syslog都包含PRI和MSG部分，而HEADER可能没有。

# 2.格式

如下为一个syslog消息：

<30>Oct 9 22:33:20 hlfedora auditd[1787]: The audit daemon is exiing;

这里的“30”是PRI部分，“Oct 9 22:33:20 hlfedora”是HEADER部分，其余是message部分。

## 2.1.PRI部分

PRI部分由一对尖括号包含的一个数字组成，这个数字包含了程序模块(facility)、严重性(severity)两个信息，其规则是：facility\*8 + severity = PRI。

facility定义如下：

|  |  |
| --- | --- |
| Numerical | Facility |
| 0 | Kernel message |
| 1 | User-level message |
| 2—15 | System using |
| 16—23 | Local use |
|  |  |

由此可见，user(1)和16—23都可以给自有的程序使用。

severity定义如下：

|  |  |
| --- | --- |
| Numerical | Severity |
| 0 | Emergency：system is unusable |
| 1 | Alert: action must be taken immediately |
| 2 | Critical: critical conditions |
| 3 | Error: error conditions |
| 4 | Warning: warning conditions |
| 5 | Notice: normal but significant condition |
| 6 | Infomational: informational message |
| 7 | Debug: debug message |

PRI不允许以0开头，也就是说，除非PRI的值就是0，否则像001和01和00都是不允许的情况，应该是1或者0.

## 2.2.HEADER部分

HEADER部分包括两个字段：时间和主机名(或IP)。

### 2.2.1.时间

时间紧跟在PRI后面，中间没有空格，格式必须是“Mmm dd hh:mm:ss”，不包括年份。

* “日”的数字如果是0-9之间，那么前面将补一个空格，也就是月份后面将出现两个空格；
* 而“时分秒”如果不足两位，则前面补零；
* 月份取值包括：Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec

### 2.2.2.主机名

时间后面跟一个空格，然后是主机名(或IP)，主机名不得包括域名。

XML

# 1.libxml

Libxml2是一个C语言实现的xml程序库，可以简单方便的提供对xml文档的各种操作，支持XPATH查询，并部分支持SXLT转换等功能。

## 1.1.安装方式

sudo apt-get install libxml2

sudo apt-get install libxml2-dev

或可以在官网(http://xmlsoft.org)下载源码后，自行编译并安装；

## 1.2.基本数据类型

### 1.2.1.内部字符类型xmlChar

typedef unsigned char xmlChar; //xmlstring.h

使用unsigned char作为内部字符格式，是因为它能很好的适应UTF-8编码格式，而UTF-8编码正是libxml2的内部编码，其他格式的编码都要经过转换才能再libxml2中使用。

因为总是会用到xmlChar和char之间的类型转换，所以定义了一个宏BAD\_CAST：

#define BAD\_CAST (xmlChar \*) //xmlstring.h

这里只是进行了一个强制类型转换。

### 1.2.2.文件类型及指针

文件类型：xmlDoc

文件类型指针：xmlDocPtr

xmlDoc是个结构体，保存了一个xml文件的相关信息，如文件名、文件类型、子节点等；

xmlDocPtr是一个指针，等同于xmlDoc \*；

### 1.2.3.节点类型及指针

节点类型：xmlNode

节点类型指针：xmlNodePtr

typedef struct \_xmlNode xmlNode;

typedef xmlNode \*xmlNodePtr;

节点之间以链表和树两种形式同时组织起来的，结构体中有next和prev两个指针，组成双向链表；有parent和children两种指针，组织成树；其中还包含如下重要属性：

* content：节点中的文字内容；
* doc：节点所属的文件；
* name：节点的名字；
* ns：节点的namespace；
* properties：节点的属性列表；

### 1.2.4.节点集合类型及指针

节点集合类型：xmlNodeSet

节点集合类型指针：xmlNodeSetPtr

节点集合代表一个由节点组成的变量，节点集合只会作为xpath的查询结果出现，因此被定义在xpath.h中。

typedef struct \_xmlNodeSet xmlNodeSet;

typedef xmlNodeSet \*xmlNodeSetPtr;

struct \_xmlNodeSet

{

int nodeNr; //number of nodes in this set

int nodeMax; //size of array as allocated;

xmlNodePtr \*nodeTab; //array of nodes in no particular order

};

可以看出，节点集合共有三个成员，分别是节点集合的节点数、最大能容纳的节点数、节点数组的头指针。

## 1.3.基本函数

#include <libxml/parser.h>

#include <libxml/tree.h>

#include <libxml/xpath.h>

### 1.3.1.空白字符处理

int xmlKeepBlanksDefault(int val);

该函数可以控制如下场景：解析xml时，是否去除空白字符。如果设置为不去除空白字符，那么这些字符也会被当做一个node来处理。

val == 0：去除空白字符；

val == 1：不去除；

返回1表示设置成功，0表示设置失败。

### 1.3.2.文件载入和保存

xmlDocPtr xmlParseFile(const char \*filename);

将xml文件从硬盘上载入到内存中，并生成DOM树。使用完毕后，需要使用xmlFreeDoc函数释放资源；

如果函数执行成功，返回这个文档的根节点；否则返回NULL；

int xmlSaveFormatFileEnc(const char \*filename, xmlDocPtr cur, const char \*encoding, int format);

将内存中的DOM树，保存在硬盘上，生成一个带格式的xml文件；

cur：需要保存的xml文档；

encoding：需要保存的编码类型，可以为NULL；

format：是否执行格式化。0表示不执行格式化，1表示格式化。只有在xmlIndentTreeOutput设置为1、或者xmlKeepBlanksDefault(0)时，format设置为1才能够生效。

该函数将返回成功写入的字节数。

### 1.3.3.内存载入和输出

xmlDocPtr xmlParseMemory(const char \*buffer, int size);

将一块内存中的数据，以xml格式进行解析，并生成一个DOM树。使用完毕后，需要使用xmlFreeDoc函数释放申请的资源。

buffer：存放xml格式数据的内存区；

size：内存中XML格式数据的长度；

返回值：如果执行成功，将返回根节点，否则返回NULL；

void xmlDocDumpFormatMemoryEnc(xmlDocPtr out\_doc, xmlChar \*\* doc\_txt\_ptr, int \* doc\_txt\_len, const char \*txt\_encoding, int format);

将DOM树导入到内存中，形成一个xml格式的数据。

out\_doc：需要输出成xml格式的xml文档节点；

doc\_txt\_ptr：输出文档的内存区。由本函数在内部申请。使用完成后，需要调用xmlFree()函数释放资源。

doc\_txt\_len：输出文档内存区的长度；

txt\_encoding：输出文档的编码类型；

format：是否执行格式化；同1.3.2中format。

### 1.3.4.创建和释放xml文档

xmlDocPtr xmlNewDoc(const xmlChar \*version);

在内存中创建一个新的xml文档，需使用xmlFreeDoc函数释放资源。

version：xml标准的版本，目前只能设置为“1.0”；

void xmlFreeDoc(xmlDocPtr cur);

释放内存中的xml文档。

### 1.3.5.节点操作

#### 获得根节点

xmlNodePtr xmlDocGetRootElement(xmlDocPtr doc);

获得指定xml文档的根节点；

* 使用得到的根节点，可以遍历整个xml文档；
* xmlNodePtr的next字段，指向下一个同级的xml节点；
* properties字段为xmlAttr类型，指向该xml节点的属性；
* children字段为xmlNodePtr类型，指向该xml节点的子节点；

#### 设置根节点

xmlNodePtr xmlDocSetRootElement(xmlDocPtr doc, xmlNodePtr root);

设置xml文档的根节点。

如果该文档原来有根节点，则返回根节点；否则返回NULL；

#### 获取节点的内容

xmlChar \* xmlNodeGetContent(xmlNodePtr cur);

cur：节点指针；

返回该节点的节点内容。如果该节点没有内容，就返回NULL；如果返回值不是NULL，那么需要调用xmlFree函数释放资源。

#### 设置节点的内容、长度

void xmlNodeSetContentLen(xmlNodePtr cur, const xmlChar \* content, int len);

cur：节点指针；

content：节点的新文本内容；

len：新文本长度；

#### 在节点内容后增加新内容

void xmlNodeAddContentLen(xmlNodePtr cur, const xmlChar \*content, int len);

在节点的原有内容后面增加新的内容；

#### 获取节点的属性

xmlChar \* xmlGetProp(xmlNodePtr node, const xmlChar\* name);

如果指定节点存在name为输入值的属性，那么返回该属性；否则返回NULL；

如果返回值不是NULL，需要使用xmlFree函数释放资源。

#### 设置节点的属性

xmlAttrPtr xmlSetProp(xmlNodePtr node, const xmlChar \*name, const xmlChar \*value);

设置节点的属性，如果已经存在这个属性了，那么使用输入值替换原有的属性值。

返回该属性节点的指针；

## 1.4.XPATH函数

### 1.4.1.生成xpath的上下文句柄

xmlXPathContextPtr xmlXPathNewContext(xmlDocPtr doc);

返回值：该文档的xpath上下文句柄或NULL；

如果返回值不是NULL，那么需要调用xmlXPathFreeContext()函数执行资源的释放。

### 1.4.2.释放xpath的上下文句柄

void xmlXPathFreeContext(xmlXPathContextPtr ctxt);

### 1.4.3.执行xpath表达式

xmlXPathObjectPtr xmlXPathEvalExpression(const xmlChar \* str, xmlPathContextPtr ctxt);

执行xpath的表达式，返回结果是节点集合。

str：xpath表达式；

ctxt：xpath的上下文句柄；

返回值：节点集合或NULL。如果返回值不是NULL，需要调用xmlXPathFreeObject()函数来释放资源。

### 1.4.4.释放xpath表达式结果集合

void xmlXPathFreeObject(xmlXPathObjectPtr obj);

## 1.5.XPATH基础

XPath使用路径表达式，来选取XML文档中的节点或节点集合，节点是通过路径(path)或者步(step)来选取的。

### 1.5.1.路径表达式

|  |  |
| --- | --- |
| 表达式 | 含义 |
| Nodename | 选取此节点的所有子节点 |
| / | 从根节点开始，选取指定节点 |
| // | 不考虑位置，选择所有符合条件的节点 |
| . | 选取当前节点 |
| .. | 选取当前节点的父节点 |
| @ | 选取属性 |

以如下xml文件为示例：

<?xml version="1.0" encoding="UTF-8"?>

<bookstore>

<book>

<title lang="eng">Harry Porter</title>

<price>29.99</price>

</book>

<book>

<title lang="eng">Learning everyday</title>

<price>29.98</price>

</book>

</bookstore>

|  |  |
| --- | --- |
| 表达式 | 结果 |
| bookstore | 选取bookstore节点的所有子节点 |
| /bookstore | 从根节点开始，选取路径为bookstore的节点；  如果表达式起始于“/”，那么这个path就代表了一个绝对路径； |
| bookstore/book | 选取所有bookstore的子节点中的book节点 |
| //book | 选取所有book节点，不论其在文档中的位置 |
| bookstore//book | 选取所有bookstore节点的子节点中的book节点，不论其在文档中的位置 |
| //@lang | 选取所有名字是lang的属性，不论其位置 |

### 1.5.2.谓语

谓语用来查找某个特定的节点，或者查找包含某个特定的值的节点。

谓语被嵌在“[]”中。

|  |  |
| --- | --- |
| 表达式 | 结果 |
| /bookstore/book[1] | 选取bookstore节点的第一个book子元素 |
| /bookstore/book[last()] | 选取bookstore节点的最后一个book子元素 |
| /bookstore/book[last()-1] | 选取bookstore节点的倒数第二个book子元素 |
| /bookstore/book[position()<3] | 选取bookstore节点的所有子元素中的前两个 |
| //title[@lang] | 选取所有拥有名字是lang的属性的titile节点 |
| //title[@lang=”eng”] | 选取所有title元素，并且这些元素都有名字是lang的属性，且这些属性的值都是“eng”；  还支持：> >= < <= !=等比较运算符 |
| /bookstore/book[price>35.00] | 选取所有price元素的值大于35的book元素 |
| /bookstore/book[price>35.00]/title | 更进一步，选取所有title元素 |
|  |  |

### 1.5.3.通配符

|  |  |
| --- | --- |
| 通配符 | 描述 |
| \* | 匹配任何元素节点 |
| @\* | 匹配任何属性节点 |
| node() | 匹配任何类型的节点 |
| | | 逻辑或 |

比如：

/bookstore/\*：选取所有bookstore的子节点；

//\*：选取文档中的所有元素；

//title[@\*]：选取所有有属性的title元素；

//book/title | //book/price：选取book元素的所有title和price元素；

## 1.6.ICONV基础

在Linux上进行编码转化时，既可以利用iconv函数族编程实现，也可以利用iconv命令实现。前者在程序中被调用，后者针对文件，将文件从一种编码格式转换为另一种。

### 1.6.1.iconv函数族

#include <iconv.h>

iconv\_t iconv\_open(const char \*tocode, const char \*fromcode);

该函数指定要进行哪两种格式的字符集转换。该函数返回的是一个句柄，供以下两个函数使用。

size\_t iconv(iconv\_t cd, char \*\*inbuf, size\_t \*inbytesleft, char \*\*outbuf, size\_t \*outbytesleft);

此函数执行具体的转换工作。

从inbuf中读取字符，转换后输出到outbuf中。

inbytesleft用来记录还没有转换的字节数，outbytesleft用来记录输出缓冲的剩余空间。

int iconv\_close(iconv\_t cd);

关闭句柄，释放资源。

### 1.6.2.iconv命令

用法：iconv [选项…] [文件…]

#### 选项

-f：也可以“--from-code=”，用来指定原始的文本编码；

-t：也可以“--to-code=”，用来指定目标的文本编码格式；

-l：也可以“--list”，列举所有已知的字符集；

-c：从输出中，忽略无效的字符；

-o：也可以“--output=FILE”，指定输出文件；

--verbose：打印进度信息；

-V：也可以“--version”，打印版本信息；

#### 示例

iconv -f utf-8 -t gb2313 aaa.txt >bbb.txt

将aaa.txt文件，从utf-8类型转换为gb2312类型，并将结果重定向到bbb.txt中。

安防监控

# 1.N制式和P制式

N制式，即NTSC制式，National Television System Committee的缩写，主要使用于日本、美国等地；

P制式，即PAL制式，Phase Alternating Line的缩写，主要适用于中国、中东、欧洲等地；

这两种制式不可兼容!

P制式的分辨率为720\*576，而N制式是720\*480，在分辨率上P制式更占优势；

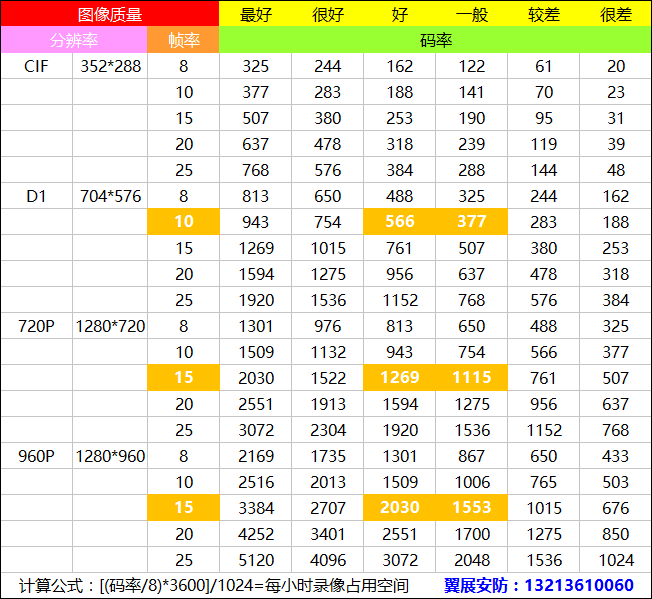
P制式每秒25个帧；N制式每秒30个帧；

# 2.QCIF

QCIF是常用的标准化图像格式，意为四分之一公共中间格式，QuarterCommon Intermediate Format。

CIF = 352\*288像素，规定于H.323协议；

目前监控行业采用的分辨率主要有：QCIF(176\*144)，CIF(352\*288)，HALF D1(704\*288)，D1(704\*576)；



# 3.主码流和子码流

双码流采用一路高码率的码流用于本地高清存储，例如QCIF/CIF/D1编码；一路低码率的码流用于网络传输，例如QCIF/CIF编码。这样可以兼顾本地存储和远程网络传输。

QT

# 1.开发环境搭建

## 1.1.Windows下开发环境搭建

在如下网址：<https://www.qt.io/download-open-source/> 下载得到了如下文件：qt-unified-windows-x86-online.exe，该文件是一个安装程序，可以在线下载安装QT开发需要的环境，包括：MinGW，QT运行需要的库文件，QT Creator等；

本安装方式，需要实时的网络下载！

# 2.HelloWorld

点击“New Project”，创建一个新的项目“helloworld”，创建时选择“模板”为“Application”，具体为“Qt widgets application”。类名等信息做相应修改。

项目创建完成后，自动进入的是“编辑”模式，选择“界面文件”中的“mainwindow.ui”文件，双击后将进入该ui的“设计”模式。

在“设计”模式下，选取一个“label”控件，直接拉到ui上，修改内容为“helloworld”，之后点击左下角的“运行”按钮，就可以将该显示。

如果要发布release版本的程序，需要在编译时选择release方式，这样编译完成后就可以在workspace中找到对应的release目录，例如我这里有一个：build-helloworld-Desktop\_Qt\_5\_7\_0\_MinGW\_32bit-Release 目录，其中包含了release目录，该目录下游需要的helloworld.exe文件。

需要发布的时候，除该exe文件外，还需要包含所需要的dll文件，这里我们需要的是：

1. libwinpthread-1.dll；
2. Qt5Core.dll；
3. Qt5Gui.dll；
4. Qt5Widgets.dll；

这几个文件，这几个文件在安装目录下可以找到，我这里的安装目录是：E:\QT\InstallFiles\5.6\mingw49\_32\bin；

将exe文件和这几个dll文件打包在同一个文件夹中，在任何位置都可以直接点击exe文件，弹出helloworld对话框了。

## 2.1.更换应用程序图标

如果要给生成的exe文件更换一个自定义的图标，可以如下操作：

1. 在项目中增加一个myapp.rc文件(文件自定义)，然后再里面输入内容：IDI\_ICON1               ICON    DISCARDABLE     "appico.ico" (这里的appico.ico就是自定义需要使用的图标);
2. 在.pro文件中增加下边一条代码：RC\_FILE = myapp.rc
3. 重新编译程序；

需要注意的是，需要使用一个符合icon文件格式的图标，否则编译失败，将提示：windres: icon file `./../helloworld/appico.ico' does not contain icon data；

## 2.2.无需每次复制dll文件的方法

每次对dll文件拷贝显然是麻烦的，针对开发者，还可以有如下两种方式，无需每次拷贝文件：

1. 设置PATH环境变量，这里我的设置是：E:\QT\InstallFiles\5.6\mingw49\_32\bin；
2. 产出成静态文件(文件体积增加，不推荐)；

# 3.快捷键

## 3.1.更改QT Creator的自动补全快捷键

“工具(tools)”—“选项(options)”，选择“键盘”后，输入CompleteThis，将自动过滤出现在的快捷键，可以看到是“ctrl+space”，但是这个经常与输入法的中英文切换冲突，改成eclipse里用习惯了的“Alt+/”；

注意这里修改的时候，是让QT Creator自动捕获键盘动作，而不是手动输入的。

## 3.2.快速切换已经打开的文件

Ctrl+tab

## 3.3.注释 / 取消选中的代码

Ctrl+/

## 3.4.自动缩进选中代码

Ctrl+i

## 3.5.添加固定格式的整段注释的快捷方式

1. “工具”—“选项”—“文本编辑器”；
2. 选中“片段”中的“添加”按钮，新建一个“触发”，名字自己定为了“funcComment”，触发种类同样适用到了这个名字；
3. 填写自己的注释内容：

/\*\*

\* @brief

\* @param

\* @author

\* @date

\*/

执行如上设置后，在qtcreator中输入“funcComment”，就会提示该段落；

## 3.6.改变编辑器中字体大小

Ctrl + +

Ctrl + -

## 3.7.编辑 信号/槽

在“设计”模式下，按下F4可以进入编辑“信号/槽”的界面；

# 4.设置字体编码格式

可以通过如下方式改变编码格式，一般在中文乱码显示时需要进行设置：

#include <QTextCodec>

QTextCodec::setCodecForLocale(QTextCodec::codecForName("GB2312"));

需要注意的是，一定要在主函数执行逻辑动作之前就进行设置才能全部执行过程中都生效；

# 5.两种捕获按钮按下动作的方法

## 5.1.槽回调函数

选中要设置动作的按钮，选择“选中槽”后，选择某一个动作(例如左键单击对应的动作就是clicked())，函数将跳转到一个函数处(例如on\_pushButton\_clicked())，这个就是自动生成的槽，已经在头文件中做了声明，只需要修改其函数实现即可。

## 5.2.信号/槽

在“设计”模式下，按下F4进入“信号/槽”的编辑模式，选中需要设置按下动作的按钮，并选择某一个动作和对应的接收对象即可。

QT Creator快速入门

对《QT Creator快速入门》第二版的读书笔记；

# 3.窗口部件

QT Creator提供的默认基类，有QMainWindow、QWidget和QDialog三种；

* QWidget是所有窗口部件的基类；
* QDialog是各种对话框的基类；
* QMainWindow是带有菜单栏和工具栏的主窗口类；

## 3.1.基础窗口部件QWidget

### 3.1.1.窗口、子部件以及窗口类型

#### 窗口与子部件

一段示例代码：

#include <QtGui>

int main(int argc, char \*\*argv)

{

QApplication a(argc, argv);

//代码中支持中文

QTextCodec::setCodecForTr(QTextCodec::codecForName(“UTF-8”));

//新建QWidget对象，默认parent设置为空，是一个窗口

QWidget \* widget = new QWidget();

//新建label对象，默认parent设置为空，是一个窗口

QLabel \* label = new QLabel();

//label2也是一个label对象，但是指定父窗口为widget，所以不是一个窗口

QLabel \* label2 = new QLabel(widget);

//在显示器上显示

label->show();

widget->show();

int ret = a.exec();

//释放对象，注意，label2不执行delete

delete label;

delete widget;

//结束

return ret;

}

需要注意的点：

1. 头文件包含的是QtGui，因为用到的类：QApplication、QWidget都包含在这个模块里；
2. 因为label没有父窗口，所以需要手动释放它申请的空间；而label2的父窗口是widget，**QT中销毁父对象的时候，会自动销毁子对象**，因此label2不需要手动释放空间，因为delete widget的时候已经将它释放了；

#### 窗口类型

默认的窗口，一般都存在边框和标题栏，但其实这并不是必须的；

QWidget的构造函数有两个参数：QWidget \* parent = 0 和 Qt::WindowFlags f = 0；

* 第一个参数“parent”指定其父窗口部件，默认值为0表示不存在父窗口；
* 第二个参数f的类型Qt::WindowFlags是一个枚举，分为：窗口类型(WindowType)和窗口标志(WindowFlags)。

简单的例子，对于窗口类型来说，Qt::Dialog可以指定窗口的样式是对话框类型；Qt::SplashScreen指定窗口类型是欢迎窗口类型；Qt::Widget是默认值，如果有父窗口，就是其子部件，否则就是一个独立窗口；

对于窗口标志来说，作用主要就是更改窗口的标题栏和边框，而且可以和窗口类型进行或操作，例如：Qt::FramelessWindowHint用来产生一个没有边框的窗口；Qt::WindowStaysOntTopHint用来使本窗口停留在所以其他窗口上面；

### 3.1.2.窗口几何布局

对于窗口的大小和位置，根据是否包含边框和标题栏两种情况，要用不同的函数来获取。例如：

* 包含框架的：x(), y(), frameGeometry(), pos(), move()等；
* 不包含框架的：geometry(), width(), height(), rect(), size()等；

### 3.1.3.程序调试

#### 设置断点

将光标移动到要调试的函数上，按下F1键，可以查看到函数的帮助文档；

设置好断电后，按下F5键可以开始调试；

按下F11，可以进入单步调试；

将光标移动到一个类名或者函数上，按下F2键，或者右击后选择“跟踪光标位置的符号”，编辑器会跳转到源码处；

#### 使用qDebug函数

使用qDebug函数，可以将调试信息直接输出到控制台；

示例：

1. qDebug(“ret = %d”, ret);
2. qDebug << “ret = ” << ret << enl << “desc : ” << desc;

如果使用方式1输出，使用类似于printf的输出格式，无需新增头文件；

方式2输出，无需手动指定format，更加方便，但是需要包含头文件：#include <QDebug>；

方式2中，endl与std中的标准输出换行作用相同；

## 3.2.对话框Dialog

### 3.2.1.模态和非模态对话框

QDialog类是所有对话框窗口类的基类。

对话框窗口是一个经常用来玩成一个短小任务或者和用户有简单交互的顶层窗口；

模态对话框就是没有关闭它之前，不能再与同一个应用程序的其他窗口进行交互，像新建项目时弹出的对话框就是这种；

非模态对话框就是既可以与它交互，也可以与同一个程序中的其他窗口交互，例如word中查找字符串的功能；

最简单的，想让一个对话框作为模态存在，只需要使用其exec()函数来执行；而想做为一个非模态的，只需要使用show()函数的默认方式来执行；

需要注意的是，使用show函数也可以创建一个模态对话框：

QDialog \* pD = new Dialog(this);

pD->setModal(true); //默认该属性为false

pD->show();

如上进行设置后，就可以将该对话框设置为模态的了。

但是与exec函数的效果还是不同的：exec函数一旦被执行，该对话框不消失的情况下，其他对话框(包括父窗口)都将无法显示；而show函数不同，其他窗口依然存在，只是无法交互了；

与stModal()函数类似的还有一个函数：setWindowModality函数。

该函数功能更加丰富，它专门有一个参数设置模态对话框要阻塞的类型：

* Qt::NonModal:不阻塞任何窗口，就是非模态对话框；
* Qt::WindowModal:阻塞它的父窗口和所有祖先窗口，以及它们的子窗口；
* Qt::ApplicationModal:阻塞整个应用程序的所有窗口；

setModal函数的功能，其实与Qt::ApplicationModal相同；

### 3.2.2.多窗口切换

#### 信号和槽

QT中使用信号和槽，完成对象之间的协同操作；

声明槽函数：

public slots:

void showChildDialog();

槽函数的声明，必须使用“slots”关键字；这里使用public，说明这个槽可以在类外被调用。

实现该函数，可以通过快捷键实现：选中槽函数，按下Alt+enter后，选中“在\*\*\*.cpp中添加定义”，编辑器会自动创建该函数的声明，只需要增加代码即可；该方法同样适用于先在源文件中增加了定义，然后自动在头文件中增加声明的情况；

槽函数声明后，想在控件点击或其他动作时触发这个槽，还需要进行关联：connect(ui->showChildBtn, SIGNAL(clicked()), this, SLOT(showChildDialog())); 这里使用了connect函数进行关联，connect函数的四个参数分别是：发送信号的对象、发送的信号、接收信号的对象和要执行的槽；信号和槽分别使用SIGNAL和SLOT宏包起来。

信号和槽的关联，还有一种方式是自动关联。

自动关联就是将将关联函数整合到槽的命名中，但是必须使用QT部件已经提供的信号。例如：on\_showChildBtn\_clicked()，由“on”+“要发射信号的部件对象名”+“信号名”组成。这样就可以不适用connect函数进行关联了。

快捷方式修改所有某一个函数出现的地方：选中该函数，右键，弹出的菜单中选择“重构🡪Rename Symbol Under Cursor”；

#### Demo

主窗口、登录窗口，可以互相切换；

### 3.2.3.标准对话框

#### 颜色对话框

头文件：#include <QColorDialog>

颜色对话框类 QColorDialog提供了一个可以获取指定颜色的对话框部件。两种显示颜色对话框的调用方式：

1. 静态调用：QColor color = QColorDialog::getColor(Qt::red, this, tr(“颜色对话框”), QColorDialog::ShowAlphaChannel);四个参数的含义分别是：设置初始颜色、父窗口、对话框标题、显示alpha的设置内容；
2. 动态调用：QColorDialog dialog(Qt::red, this); dialog.setOption(QColorDialog::ShowAlphaChannel); dialog.exec(); QColor color = dialog.currentColor();

方式1的优点是不用创建对象，写起来更方便；但如果想更灵活的设置颜色对话框的属性，就需要用到方式2，也就是先创建对象，之后进行设置；

#### 文件对话框

头文件：#include <QFileDialog>

文件对话框类QFileDialog提供了一个允许用户选择文件或者文件夹的对话框。同样分为静态和动态两种调用方式：

1. 静态调用：QString filename = QFileDialog::getOpenFileName(this, tr(”文件对话框”), “F: ”, tr(“图片文件(\*png \*jpg)”));四个参数的含义分别是：父窗口、对话框名字、默认打开的文件路径、文件类型过滤器。
2. 动态调用：创建对象、设置属性；

设置文件类型过滤器，可以使打开的文件对话框中只显示这些类型的文件；如果不指定该值，默认打开所有类型的文件；**注意，\*png和\*jpg之间有一个空格**！

如果要显示更多种类的文件，**需要以两个分号隔开**！例如：QFileDialog::getOpenFileName(this, tr(”文件对话框”), “F: ”, tr(“图片文件(\*png \*jpg);;文本文件(\*txt)”))；

如果允许一次选中多个文件，需要使用QFileDialog类提供的getSaveFileNames函数实现，其返回值类型是QStringList，所有文件名置于其中；

QFileDialog类还提供了两个函数：

* getSaveFileName函数实现了保存文件对话框和文件另存为对话框；
* getExistingDirectory函数获取一个已经存在的文件夹路径；

#### 字体对话框

头文件：#include <QFontDialog>

字体对话框提供了一个可以选择字体的对话框部件。调用方式，同样分为静态和动态两种，例如：

bool ok;

QFont font = QFontDialog::getFont(&ok, this);

if(ok)

ui->btn1->setFont(font);

else

qDebug() << tr(“没有所选择的字体”);

这里使用的就是静态的、getFont函数来获取选择的字体；

#### 输入对话框

头文件：#include <QInputDialog>

输入对话框QInputDialog类用来提供一个简单方便的对话框，从用户那里获取一个单一的数值或者字符串，例如：

* QInputDialog::getText(this, tr(“输入字符串对话框”), tr(“请输入用户名”), QLineEdit::Normal, tr(“admin”), &ok);提供了一个可以输入字符串的对话框，参数的作用分别是:
  + 父窗口；
  + 窗口标题；
  + 对话框中的标签的显示文本；
  + 字符串的显示模式，例如密码可以设置成小黑点，这里用了Normal，是显示用户输入的时机内容；
  + 输入框中的默认字符串；
  + 按下按钮的bool变量，捕获动作；
* QInputDialog::getInt(this, tr(“输入整数对话框”), tr(“请输入一个整数，范围在[-1000, 1000]”), 100, -1000, 1000, 10, &ok); 这里100表示的是对话框的默认值；-1000和1000表示的分别是允许输入的最小值和最大值；10标识箭头按钮，数值每次变化10；
* QInputDialog::getDouble(this, tr(“输入浮点数对话框”), tr(“请输入-1000到1000内的数值”), 0.00, -1000, 1000, 2, &ok); 其他参数类似，倒数第二个参数2，指定的是小数的位数；
* QStringList items; QInputDialog::getItem(this, tr(“输入条目对话框”), tr(“请选择一个条目”), items, 0, true, &ok); 需要首先提供一些条目，例如这里的items；参数0表示的是默认显示列表中的第0个条目；参数true表示的是是否允许修改条目，true就是可以修改；

如上的例子里，用到了静态函数的方式，也可以使用静态函数的方式来实现；

#### 消息对话框

头文件：#include <QMessageBox>

消息对话框提供了一个模态的对话框，来通知用户一些信息，或者向用户提出一个问题并获取答案，例如：

* QMessageBox::question(this, tr(“问题对话框”), tr(“你了解QT吗？”), QMessageBox::Yes, QMessageBox::No);
* QMessageBox::information(this, tr(“提示对话框”), tr(“学习QT吧！”), QMessageBox::Ok);
* QMessageBox::warning(this, tr(“警告对话框”), tr(“有错误！”), QMessageBox::Abort);
* QMessageBox::critical(this, tr(“严重错误对话框”), tr(“致命错误！”), QMessageBox::YesAll);
* QMessageBox::about(this, tr(“关于对话框”), tr(“这是一个说明这个工具是干什么的对话框”));

#### 进度对话框

#include <QProgressDialog>

进度对话框对一个耗时较长的操作，进行进度条的反馈，例如：

QProgressDialog dialog(tr(“文件复制进度信息”), tr(“取消”), 0, 5000, this);

dialog.setWindowTitle(“进度对话框”); //设置窗口标题

dialog.setWindowModality(Qt::WindowModal); //**设置为模态对话框**

dialog.show(); //开始显示

for(int I = 0; I < 5000; i++)

{

dialog.setValue(i); //设置进度值

QCoreApplication::processEvents(); //**避免页面冻结**

if(dialog.wasCanceled()) //用户按下了取消按钮

break;

}

dialog.setValue(5000); //这样才能显示到5000，for只能加到4999

需要注意：

1. 示例中将其设置为了模态对话框，还可以使用非模态的，但是需要定时器等其他类的配合；
2. QCoreApplication的使用，是为了避免长时间没有用户动作，导致界面冻结；
3. 构造函数的参数含义分别是：对话框的标签内容，取消按钮的显示文本，最小值，最大值，父窗口；

#### 错误信息对话框

头文件：#include <QErrorMessage>

错误消息提示框提供了一个显示错误信息的对话框。例如：

QErrorMessage \* dialog = new QErrorMessage(this);

dialog->setWindowTitle(tr(“错误消息对话框”));

dialog->showMessage(tr(“这里是出错信息”));

这种方式实现，对话框以非模态的形式体现；

在错误消息对话框中，默认有一个Show this message again的复选框，用来选择以后是否还显示相同的错误信息。

#### 向导对话框

头文件：#include <QWizard>

向导对话框提供了一个设计向导界面的框架。最典型的向导对话框的使用，就是安装、卸载软件时候的向导。

暂略。

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；

Valgrind

# 1.安装

官网：<http://valgrind.org/>；

可以在官网下载最新的valgrind程序；

首先执行配置操作：./configure --prefix=/home/wujl/test/valgrind-3.12.0/output；使用“prefix”指定安装到哪个目录，默认会安装到/usr/local/中，因此如果不设置这个值，要添加root权限；

如果是交叉编译，需要指定交叉编译工具链：./configure --host=arm-hisiv100nptl-linux CC=arm-hisiv100nptl-linux-gcc CPP=arm-hisiv100nptl-linux-cpp CXX=arm-hisiv100nptl-linux-g++ --prefix=/home/wujl/test/valgrind-3.12.0/output；

此外，如果板子的型号并不是armv7，而是其他arm型号，需要手动修改配置文件configure，在其中的armv7\*) 改为 armv7\* | arm )；

之后:make；make install；

执行完成后，所有的相关文件，会产出到指定的目录中，例如我的output目录中；

# 2.基本使用

./valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --log-file=valgrind.log /mnt/a.out

* --tool：指定使用哪个工具检测，检测内存泄露的是memcheck，如果不适用—tool指定，那么默认使用的也是memcheck；
* --leak-check：指定是否检测泄漏；
* --show-reachable：是否检测“程序结束后依然可达”的内存泄漏；
* --log-file：输出检测结果到文件；
* Others

# 3.MyFAQ

Question：valgrind failed to start tool 'memcheck' for platform 'arm-linux': not a directory

Answer：运行时提示如上错误，是由于编译产出物中，有lib目录，如果默认路径安装，是安装在/usr/local/lib中的，可以直接找到并使用；如果不是默认路径安装，而是使用了prefix来指定，valgrind进程找不到库就会报这个错。解决方法是： export VALGRIND\_LIB="/mnt/output/lib/valgrind/"；

Question：illegal instruction

Answer：非法的汇编指令，我在板子上就遇到了这个问题，而且一直没有解决！问题出现的原因，是编译的时候连接的动态库，和板子上部署的动态库根本不是同一个；而且如果动态库没有调试信息，也会报这个错误；

看网上有人说，架构不同也会导致这个问题，而且valgrind现在只能支持armv7，因此需要在Makefile中增加：-march=armv7-a 这个信息，但是我增加了这个也没有效果；

Question：无法捕获到内存出错信息

Answer：有几个原因可能导致捕获不到内存出错信息：1. 编译时使用的编译级别过高，将调试信息都删除了，例如：CFLGAS修改O2-->O0；2.程序没有正常终止；

moDemos

# 1.socket网络编程

## 1.1.同步/异步、阻塞/非阻塞

同步和异步的概念，主要针对客户端来说；阻塞/非阻塞的概念，主要针对服务器端来说。

* 同步：所谓同步，指的是client发出一个调用后，在没有得到结果之前，这个调用就不返回。也就是必须一件一件事情的顺序执行，一件事做完了才能做下一件事；
* 异步：异步则不同；指的是client发出一个调用后，不能立即得到需要的结果，但是调用会立即返回；一旦这个调用在server端执行完成后，server端将通过 状态、通知或者回调 来通知调用者。
* 阻塞：阻塞指的是server端处理完这个请求之前，处理线程将被挂起；
* 非阻塞：指的是server端收到调用请求后，立即返回，而后通过其他方式通知client这个调用的执行结果；

## 1.2.Linux下的五种I/O模型

### 1.2.1.阻塞I/O模型

特点是：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，导致该应用程序阻塞，直到数据准备好了才能够继续执行，如果数据一直没有准备好，将一直等待；

调用recv/recvfrom函数时，发生在内核中的等待数据和赋值数据的过程如下图：



这种模型下，一般来说，一旦接收到一个请求，会开辟一个线程或进程处理这个请求，主进程/主线程继续等待下一个请求；

#### 源码路径

moDemos/socket/block下的client和server目录；

#### 交互序列

客户端和服务器端使用TCP的方式进行阻塞式网络通信时:

**客户端**的依次函数调用顺序是：

1. Socket：创建一个socket；
2. Bind：非必需，如果不执行bind，系统默认会找一个可用的port来给客户端使用；
3. Connect：在服务器端已经就绪的情况下，以此函数向其发起握手请求；
4. Send：发送数据给服务器端；
5. Recv：收取服务器端发送过来的数据；
6. Close：关闭这个已经创建的socket；

**服务器端**的依次函数调用顺序是：

1. Socket：创建一个socket；
2. Bind：必须，对于服务器端来说，一般来说一定要手动指定一个port；否则客户端在connect的时候无法获知需要连接到哪个端口；
3. Listen：指定监听能力；
4. Accept：监听connect请求；
5. Recv：收取客户端发送来的request；
6. Send：发送数据给客户端；
7. Close：关闭这个socket；

使用UDP的方式进行阻塞式网络通信时：

**客户端**的依次函数调用顺序是：

1. Socket：创建一个socket；
2. Bind：非必需，如果不执行bind，系统默认会找一个可用的port来给客户端使用；
3. ~~Connect：在服务器端已经就绪的情况下，以此函数向其发起握手请求；~~
4. Sendto：发送数据给服务器端；
5. Recvfrom：收取服务器端发送过来的数据；
6. Close：关闭这个已经创建的socket；

**服务器端**的依次函数调用顺序是：

1. Socket：创建一个socket；
2. Bind：必须，对于服务器端来说，一般来说一定要手动指定一个port；否则客户端在connect的时候无法获知需要连接到哪个端口；
3. ~~Listen：指定监听能力；~~
4. ~~Accept：监听connect请求；~~
5. Recvfrom：收取客户端发送来的request；
6. Sendto：发送数据给客户端；
7. Close：关闭这个socket；

#### FAQ

**1.SIGPIPE引起的程序自动退出**

Client连接server时，如果server没有启动，一定概率下，程序执行到send/sendto函数的时候，会突然自行退出，并提示是收到了“SIGPIPE”的信号量所致。

正常情况下，如果一个进程收到“SIGPIPE”信号，默认的动作就是退出进程。

所以针对这个情况有两个处理方式:1.自行实现一个函数，注册给SIGPIPE这个信号量；2.将SIGPIPE信号量忽略。

代码里用到了后者： signal(SIGPIPE, SIG\_IGN);

这样处理后，程序不会退出了。我们还可以根据send/sendto函数执行出错后的errno进行判断，如果errno等于SIGPIPE，就知道是收到了这个信号量。

**2.程序退出后，短时间内不能再次启动**

在client/server进程执行结束退出后，如果立刻再次执行该进程，进程bind到port的函数将执行失败。因为对于端口的使用，默认是要过一段时间才会释放，一般系统设置为2—4分钟，所以需要等待这么长时间之后才能再次执行bind。

但是如果希望不等待直接能够执行，可以在创建了socket之后，对该socket置状态：int on=1; int ret = setsockopt(mSockId,SOL\_SOCKET, SO\_REUSEADDR,&on,sizeof(on));

设置该状态后，可以确保退出后即释放端口，马上可以再次启动。

### 1.2.2.非阻塞I/O模型

特点是：进程反复调用IO函数，直到能够收到数据为止；

把一个socket设置成非阻塞，就是要告诉内核，当所请求的I/O操作无法完成的时候，不要将进程睡眠，而是返回一个错误值。一旦进程发现返回值是这个特定的错误值，就知道是由于数据还没有就绪导致的返回出错，就可以等待后再次申请数据，直到有数据为止。

这个返回值就是errno=EWOULDBLOCK；

图示如下：



#### 代码路径

moDemos/socket/noblock下的client和server目录；

#### 设置方式

有两种方式可以用来设置一个socket为非阻塞的：

* 方式1：int b\_on = 1; ioctl(mSockId, FIONBIO, &b\_on);需要包含的头文件是：#include <sys/ioctl.h>;
* 方式2：int flags = fcntl(mSockId, F\_GETFL, 0); ret = fcntl(mSockId, F\_SETFL, flags | O\_NONBLOCK);需要包含的头文件是：#include <unistd.h> #include <fcntl.h>;函数fcntl返回负值说明执行出错了。

### 1.2.3.IO复用模型

特点是：同样是非阻塞IO，但是可以一次对多个IO端口进行监测，这是比单纯的非阻塞IO优越的地方。

主要有select、poll、epoll三种方式，图示如下：



其中，select是POSIX规定支持的，一般操作系统都支持；而epoll是linux特有的；

Select和poll，本质上都是通过设置或检查存放fd标志位的数据结构，来决定后续的处理方式；epoll则不同，是通过注册事件的方式，一旦某个fd就绪会通过回掉函数激活处理；

**支持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FD\_SETSIZE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FD\_SETSIZE为32\*64），当然我们可以对进行修改，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**

|  |  |
| --- | --- |
| 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 | 同上 |
| 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**

|  |  |
| --- | --- |
| 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 | 同上 |
| epoll | epoll通过内核和用户空间共享一块内存来实现的。 |

**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。**

#### 1.2.3.1.Select

原理是：当一个客户端连接上服务器时，服务器将其fd加入到fd\_set集合中，等到这个连接准备好读或写的时候，就通知程序进行IO操作，与客户端进行通信。

**函数原型**

int select(

int maxfdp, //winsock中该参数无意义

fd\_set \* readfds,

fd\_set \* writefds,

fd\_set \* exceptfds,

const struct timeval\* timeout //非阻塞模式下的最大超时时间

);

各参数的含义如下：

* Maxfdp：意为“最大的fd加1(max fd plus 1)”.在readfds、writefds和exceptfds中找到最大的fd值后加1；或者直接设置为FD\_SETSIZE，这是<sys/types.h>中的常数，是系统能支持的最大的fd个数，但对于大多数程序而言这个值太大了，不适合使用。
* Readfds：要监视读变化的所有文件描述符集合。如果这个集合中的文件描述符有1个可读，select就会返回一个大于0的值；如果没有可读的文件，根据timeout参数判断是否超时，如果超时了返回0，否则返回负值。可以传入NULL，表示不关心所有文件的读变化。
* Writefds：要监视写变化的所有文件描述符集合。同上。
* Exceptfds：要监视的异常错误的文件描述符集合。同上。
* Timeout：如果传入NULL，select处于阻塞状态，一定要等到文件描述符中某个文件发生变化为止；如果设置为0，select为纯粹的非阻塞函数，不管有没有文件描述符变化，都会立即返回，文件无变化返回0，有变化返回一个正数；如果设置为大于0的数值，在timeout内阻塞，返回值如上所述；

**使用步骤**

1. 创建所关注的事件的文件描述符集合:readfds，writefds，exceptfds；
2. 调用select，等待事件发生，需要注意，**select函数的阻塞与socket是否是阻塞的并无直接关联**；
3. 轮询所有fd\_set中的每一个fd，检查是否有事件，如果有，进行处理；

**相关宏**

FD\_ZERO(fd\_set \* fds); //清除所有标记位

FD\_SET(int fd, fd\_set \* fds); //设置fd\_set中对应fd的位为1

FD\_ISSET(int fd, fd\_set \* fds); //判断fd\_set中的某个fd是否被设置

FD\_CLR(int fd, fd\_set \* fds); //从fd\_set中清除fd对应的位，不再监视该文件

**返回值**

Select返回处于就绪状态并且已经包含在fd\_set结构中的描述字总数；如果超时了返回0；如果有监视错误返回负值；

* 0-：出错，对文件描述符的监视失败了，将清空所有文件描述符；
* 0：超时；
* 0+：有文件可操作，通过FD\_ISSET遍历可用的文件描述符；

Select返回后，在所有的文件描述符集合中，依旧是1的位就是准备好的描述符。这也是每次select成功后，都要通过FD\_ISSET判断文件描述符的原因。

**优缺点**

* 优点：单线程执行，CPU资源消耗小，同时能够应对多客户端；
* 缺点1：单个进程所能监视的文件描述符的最大值有限制(/proc/sys/fs/file-max)；
* 缺点2：需复制大量的句柄信息，空间消耗大；
* 缺点3：返回值是句柄列表，需要轮询处理，时间消耗大；
* 缺点4：触发方式是**水平触发**，应用程序一旦没有对一个已经就绪的描述符做IO操作，之后每次调用select还是会将这个描述符通知给进程；与之相对的是**边缘触发**；

**其他**

客户端的一个connect动作，将在服务器端激发一个“可读事件”，所以select能够探测到来自客户端的connect动作；

如果select发现某个句柄有了“可读事件”，服务器应尽快做recv动作；如果发现了“可写事件”，应及时准备send动作；

#### 1.2.3.2.Poll

Poll与select的相似度极高，且优缺点相同，不做研究。

#### 1.2.3.3.Epoll

由于select和poll在处理大量网络请求时的缺陷，epoll技术被提出。在linux内核版本2.6中正式引入了这个实现，其优点概述为：

* Epoll没有最大并发连接数的限制，上限是系统最大可以打开的文件数量，这个数字远远大于select限制的2048(甚至1024)；一般来说这个数字与系统内存关系很大，可以通过：cat /proc/sys/fs/file-max查看；
* 效率提升：epoll只关心“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的效率远高于select和poll；
* 内存拷贝：epoll使用了mmap的内核与用户空间的共享内存的方式，省去了内存拷贝的问题；

**两种工作方式**

Epoll支持两种工作方式：水平触发和边缘触发。

水平触发(Level triggered)是缺省的工作方式，支持block和non-block两种。这种模式下，内核告诉使用者一个文件描述符是否就绪了，然后就可以对这个fd进行操作。如果不做任何操作，内核还会继续通知使用者。

边缘触发(edge triggered)是高速的工作方式，只支持non-block这一种。这种模式下，当描述符变为就绪时，内核发出通知。之后内核不会再对这个描述符发送更多的就绪通知，直到使用者做了某些操作(数据读取完成等)使这个文件描述符不再是就绪了。

简单说，一旦就绪了，水平触发方式会不断的发就绪通知直到使用者处理完成；而边缘触发方式发送一次后将不再发送。

**API**

int epoll\_create(int size);

创建一个epoll的句柄。

Size指定了能够监听的数目一共有多少。

需要注意的是，一旦创建完成，epoll句柄会占用一个fd值，在linux下的/proc/processId/fd下，是可以查找到这个fd的，所以在使用完epoll之后，一定要close()关闭，否则会造成句柄泄漏；

成功时返回一个非负的文件描述符；失败时返回-1并置errno；

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

epoll的事件注册函数；

参数说明：epfd是epoll\_create时创建的epoll的句柄；op代表了三种可能的动作：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；参数fd是需要监听的文件描述符；event参数告诉内核需要监听的事件是什么，其结构体定义是：

Typedef union epoll\_data

{

Void \*ptr;

Int fd;

\_\_uint32\_t u32;

\_\_uint64\_t u64;

}epoll\_data\_t;

Struct epoll\_event

{

\_\_uint32\_t events;

Epoll\_data\_t data;

};

这个结构体中events属性的取值，可以是以下几个宏的组合：

* EPOLLIN:对应的文件描述符可读(包括对端的SOCKET正常关闭)；
* EPOLLOUT:对应的文件描述符可写；
* EPOLLPRI:对应的文件描述符有紧急的数据可读(也有说是表示有带外数据到来)；
* EPOLLERR:对应的文件描述符发生错误；
* EPOLLHUP:对应的文件描述符被挂断；
* EPOLLET:将epoll设置为边缘触发模式(edge triggered)；
* EPOLLONESHOT:只监听一次事件，监听结束后如果需要再次监听这个文件，需要再次将文件描述符加入到epoll队列中；

函数执行成功返回0；执行失败返回-1并置errno；

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

收集在epoll监控的事件中已经就绪的事件。参数events不得为NULL!epoll会把就绪的事件赋值到events数组中。Maxevents告知内核这个events有多大，不能大于创建epoll时指定的size；timeout是超时时间(**单位是毫秒**)，0会立即返回，-1将永久阻塞或行为不确定。

如果函数调用成功，返回对应IO上已经准备好的文件描述符数目，返回0标识超时，返回-1说明有错误发生，errno被置。

**通用框架**

几乎所有的epoll程序都使用这样的框架：

    for( ; ; )

    {

        nfds = epoll\_wait(epfd,events,20,500);

        for(i=0;i<nfds;++i)

        {

            if(events[i].data.fd==listenfd) //有新的连接

            {

                connfd = accept(listenfd,(sockaddr \*)&clientaddr, &clilen);

                ev.data.fd=connfd;

                ev.events=EPOLLIN|EPOLLET;

                epoll\_ctl(epfd,EPOLL\_CTL\_ADD,connfd,&ev); //将新的fd添加到epoll的监听队列中

            }

            else if( events[i].events&EPOLLIN ) //接收到数据，读socket

            {

                n = read(sockfd, line, MAXLINE)) < 0    //读

                ev.data.ptr = md;     //md为自定义类型，添加数据

                ev.events=EPOLLOUT|EPOLLET;

                epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev);//修改标识符，等待下一个循环时发送数据，异步处理的精髓

            }

            else if(events[i].events&EPOLLOUT) //有数据待发送，写socket

            {

                struct myepoll\_data\* md = (myepoll\_data\*)events[i].data.ptr;    //取数据

                sockfd = md->fd;

                send( sockfd, md->ptr, strlen((char\*)md->ptr), 0 );        //发送数据

                ev.data.fd=sockfd;

                ev.events=EPOLLIN|EPOLLET;

                epoll\_ctl(epfd,EPOLL\_CTL\_MOD,sockfd,&ev); //修改标识符，等待下一个循环时接收数据

            }

            Else

            {

                //其他的处理

            }

        }

    }

**MyFAQ**

执行accept的时候出错，errno=22，desc=[Invalid argument]

有两个原因可能导致这个问题：

1. Accept的参数没送对：accept(serverSockId, &clientSockAddr, &clientAddrLen);后两个参数可以不使用，但是要送入；
2. 没有listen：执行accept之前要首先让server开始listen；

#### 1.2.3.4.Libevent

Libevent是一个事件触发的网络库，适用于windows、linux、bsd等多种平台，内部使用select、epoll、kqueue等系统调用管理事件机制。其可以做到跨平台，性能也很卓越。

Libevent库还可以提供其他一些功能，例如HTTP、DNS等，主要的优点是:

* 事件驱动，性能高；
* 轻量级，专注于网络；
* 跨平台，支持windows、linux、mac os等；
* 支持多种I/O复用技术:epoll，kqueue，select，dev/poll等；
* 支持IO，定时器和信号等事件；

**组成概述**

通用的libevent，通常包含如下几个部分：

1. Event及event\_base事件管理，包括各种IO(socket等)、定时器、信号等，也是libevent被使用最广泛的模块；
2. Evbuffer，一个高效的缓存管理；
3. Evdns，一个异步的DNS查询功能；
4. Evhttp，一个轻量级的http服务，包括客户端和服务器端；

Libevent也支持ssl，这对于有安全要求的网络程序非常重要，但是并不是特别完善，比如http server的实现就不支持ssl。

#### 1.2.3.5.libev

# 2.线程池

## 2.1.使用原因

网络服务器等应用场合有一个共同点，就是很有可能在单位时间内需要处理数目极为巨大的连接请求，但是任务的处理时间却很短。

传统的多线程方案，是在收到请求后创建一个新的线程，由该线程执行任务。任务执行完成后线程退出，这就是“即时创建、即时销毁”的处理方式。但是如果处理任务的时间非常短，而执行频率又很高，那么服务器很可能处于不断的创建线程、销毁线程的状态。

线程执行时间分为三部分：T1—线程创建的时间；T2—线程执行时间；T3—线程销毁时间；如果T2很短，那么（T1+T3）时间占比将很高。

因此出现了线程池的概念，在初始化的时候将线程创建出来，使用的时候只占用T2的时间，可以大大降低任务的整体执行时间和系统消耗。

## 2.2.基本组成部分

线程池一般最少由以下四个部分组成：

* 线程管理器：创建并管理这个线程池；
* 工作线程：线程池中的所有线程；
* 任务接口：每个具体的任务提供的接口，供如上的工作线程调用以实现具体功能；
* 任务队列：存放还没有被处理到的任务；

字符串匹配算法(moUtils\_Search)

字符串匹配算法，指的是从一个字符串(主串)中，查找其是否包含另外一个字符串(模式串)；

该匹配原理可以扩展，并不一定局限于字符串(“\0”结尾)，而是所有char类型的数组，都可以使用该算法进行匹配；

# 1.BF算法

Brute Force算法(暴力破解算法)，时间复杂度为O(n\*m)，其中n为主串的长度，m为模式串的长度；

从主串的第一个字节开始进行匹配，如果匹配失败了跳转到下一个字符继续进行匹配；

# 2.KMP算法

KMP算法是三位发明者的名字简称组成的，分别是：Knuth，Morris，Pratt；

## 2.1.部分匹配表

KMP算法的关键，是一张部分匹配表(Partial Match Table)；

### 2.1.1.前缀、后缀

两个概念需要明确：

* 前缀：除了最后一个字符外，一个字符串的全部的头部组合；
* 后缀：除了第一个字符外，一个字符串的全部的尾部组合；

例如，字符串bread，前缀就是：b, br, bre, brea；后缀就是：read, ead, ad, d；

### 2.1.2.部分匹配值

部分匹配值就是这个字符串的“前缀”和“后缀”的**最长的**共有元素的长度，例如对字符串“ABCDABD”来说：

* A:前缀为空，后缀为空，共有元素为0个，长度自然是0；
* AB:前缀为["A”]，后缀为[“B”]，共有元素为0个，长度自然是0；
* ABC:前缀为["A”, “AB”]，后缀为[“BC”, “C”]，共有元素为0个，长度自然是0；
* ABCD:前缀为["A”, “AB”, “ABC”]，后缀为[“BCD”, “CD”, “D”]，共有元素为0个，长度自然是0；
* ABCDA:前缀为["A”, “AB”, “ABC”, “ABCD”]，后缀为[“BCDA”, “CDA”, “DA”, “A”]，共有元素为1个(“A”)，长度是1；
* ABCDAB:前缀为["A”, “AB”, “ABC”, “ABCD”, “ABCDA”]，后缀为[“BCDAB”, “CDAB”, “DAB”, “AB”, “B”]，共有元素为1个(“AB”)，长度是2；
* ABCDABD:前缀为["A”, “AB”, “ABC”, “ABCD”, “ABCDA”, “ABCDAB”]，后缀为[“BCDABD”, “CDABD”, “DABD”, “ABD”, “BD”, “D”]，共有元素为0个，长度是0；

部分匹配的实质，是有时候字符串的头部和尾部会有所重复。比如，“ABCDAB”中有两个AB，那么部分匹配值就是2。搜索词移动的时候，第一个AB向后移动4个字节，就可以来到第二个AB的位置。

### 2.1.3.部分匹配表

将所有字符的部分匹配值组织在一起，为一张部分匹配表；

如2.1.2.中ABCDABD的部分匹配表就是：

A B C D A B D

0 0 0 0 1 2 0

## 2.2.匹配过程

将模式串与主串从第一个字节开始、从前向后进行比较:

* 如果模式串的第一个字符与当前的主串对应处不匹配，模式串后移一位；
* 如果模式串的前m个字节已经与主串匹配，第m+1的字节匹配失败，那么**移动的数量 = 已匹配的字符数 - 最后一个匹配成功的字符对应的部分匹配值；**

例如，主串为BBC ABCDAB ABCD，模式串为2.1.2.中所述的ABCDABD，那么匹配过程为：

1. 第一个字符B,与模式串的第一个字符匹配失败，后移一位继续比较；
2. 后三个字节：B B 空格，都不能匹配，依然后移；
3. 字符A，第一个字节匹配成功了，继续匹配，知道空格和模式串最后一个字符“D”匹配失败了，此时匹配成功的数量是6，最后一个匹配成功的字符(B)对应的部分匹配值是2，因此移动位数为4；
4. 。。。。。。

# 3.BM算法

BM算法(Boyer Mooer algorithm)，通过引入坏字符表(bad character table)和好后缀表(good suffix table)，并采用自后向前的比较方式，进一步减少比较次数，可以取得一个较好的时间复杂度：O(n)；

所谓坏字符，指的是从后向前开始比较，一旦出现字节不相同的，就是坏字符，例如主串是“abc”，模式串是“de”，首先将ab和de进行比较，从后向前进行比较的话，第一个字符是b和e，不相等，那么b就是相对于模式串de的一个坏字符；

所谓好后缀，指的是从后向前开始比较，有部分数据是相同的，这一些相同的数据就是好后缀，例如：主串是“abcdefg”，模式串是“xbc”；开始比较后，将“abc”和“xbc”进行比较，从后向前，c匹配成功，b匹配成功，a和x匹配失败，那么“bc”就是好后缀，而如前所述，a就是坏字符；

## 3.1.坏字符规则

当发现坏字符时，需要将模式串向右移动，移动规则是:

* 后移位数 = 坏字符出现的位置 – 搜索词中**上一次出现**的位置；
* 如果坏字符不包含在搜索词中，“上一次出现的位置”为-1；

所谓上一次出现，指的是从坏字符出现的位置开始、向前找到的最后一次出现的位置。例如，主串是xaa，模式串是axa，那么坏字符“a”出现的位置是1，且上一次出现的位置是0，而不是2，那么移动位数就应该是（1-0）=1；

## 3.2.好后缀规则

好后缀规则表述为:

后移位数 = 好后缀的位置 – 搜索词中**上一次出现**的位置；

例如，假设字符串“ABCDAB”的后一个“AB”是好后缀，那么“好后缀的位置”是5**(从0开始计数，取最后一个字符的位置值)**，在“搜索词中上一次出现的位置”是1(第一个B出现的位置)，那么后移位数就应该是：5 - 1 = 4；

又比如，如果“ABCDEF”中的“EF”是好后缀，那么位置同样是5，上一次出现的位置就是-1(因为并未出现)，所以后移位数就是5 - (-1) = 6；

需要注意如下几个关键点：

* “好后缀的位置”以**最后一个字符**为准，从0开始计数；
* 如果好后缀在搜索词中只出现一次，那么“上一次出现的位置”为-1；
* 如果好后缀有多个
  + 对于最长的那个好后缀，“上一次出现的位置”可以出现在字符串的任何位置，例如“ABCDABCDAB”，如果好后缀是“AB”，那么上一次出现的位置就是4，而不是0；
  + 其他的好后缀，“上一次出现的位置”必须在头部：例如“BABCDAB”，如果好后缀是DAB、AB、B，那么由于DAB没有重复出现，AB虽然有重复出现但是并不在头部，因此好后缀使用的是B，且上一次出现的位置是0；

## 3.3.坏字符表和好后缀表

根据好后缀规则和好后缀规则，可以看到，这两个规则都只是和模式串有关，和主串并无关联，因此可以在进行比较之前，初始化这两个表格。一旦需要移位，直接从表中获取移动位数即可；

后移位数，取 坏字符表 和 好后缀表 中移动位数更多的值；

实际的算法实现中，后移位数并不是如上的实现，而是如果最后一个字节就没有匹配上，那么就对其使用了坏字符规则；如果有好后缀，就使用好后缀规则；

# 4.Sunday算法

Sunday算法同样是一种高效的字符串查找算法，其效率远高于KMP算法，对于随机字符串，很多情况下效率比BM算法更高，最关键的是，该算法的原理更为简单。

Sunday算法与BM算法很相似，只不过Sunday算法是从前向后匹配，一旦发现匹配失败的字符，关注的是主串中相对于模式串最末字符的下一个字符。例如，主串是“abc”，模式串是“d”，那么其开始比较后，将a和d首先比较，比较失败后关注的将是“b”；

后移规则是：

* 所关注的字符存在于模式串中：后移位数 = 模式串长度 – 该字符在模式串中的位置(从0开始计数)；
* 所关注的字符不存在于模式串中：后移位数= 模式串长度 + 1；

特别需要注意，该算法对最后一个字符的下一个字符的处理逻辑，谨防溢出！

校验算法(moUtils\_Check)

通讯过程的数据校验，指的是传输之前，在数据的尾部增加一些额外的附加信息，收到信息后，通过对这些信息进行校验，确认收到的数据和发出的数据是否相同。

# 1.奇偶校验

奇偶校验，是在要校验的数据后，补充0或者1，使数据中0或者1的bit总数为奇数或者偶数个。

例如，0x1a，二进制 0001 1010:

* 奇校验：1的总数为奇数个，后面补充0，1的个数是3，符合条件；
* 偶校验：1的总数为偶数个，后面补充1，1的个数就是4了，符合条件；

奇偶校验的优势很明显，位操作，容易实现，尤其容易硬件实现，常见的串口协议RS232，用的就是该校验方式；

缺点也很明显，有50%的可能性校验失败；

# 2.累加和校验

累加和校验，指的是将需要校验的数据累加，并将结果写入到最后一个字节中。这里累加的时候，将忽略进位的按照字节累加，如此才能保证最后的结果是一个字节的长度。

累加和实现起来非常简单，也经常被使用。但是这种校验方式，检验错误的能力一般，有1/256的机会出错。

# 3.CRC校验

**呃……**

**居然每个人对于算法的介绍都不同！！！**

**这个算法先搁置吧，受不了，找不到标准。**

CRC算法的基本思想，是将传输的数据当作一个位数很长的数字，使用这个数字除以另一个数，得到的余数作为校验值，加入到原始数据的后面。

最常用的几种生成多项式：

* CRC8 : X8 + X5 + X4 + X0；
* CRC12 = X12 + X11 + X3 + X2 + X0;
* CRC16 = X16 + X15 + X2 + X0;
* CRC-CCITT = X16 + X12 + X5 + X0;
* CRC-32 = X32 + X26 + X23 + X22 + X16 + X12 + X11 + X10 + X8 + X7 + X5 + X4 + X2 + X1 + X0;

位宽(width)，不是多项式对应的二进制数的位数，而是位数减1。比如CRC8的多项式，转换成二进制，就是1001 1001 0.这里说他的位数是8，其实是9-1；

普遍的数据，都采用16进制表示。由于生成多项式的最高位一定是1，且位宽限定了多项式的长度，因此第一位都被缩写了。例如CRC32的生成多项式，其实记作：0x04c1 1db7，最高位的1被省略了。

CRC算法规定，假设位宽为w，那么就要在原始数据后补充w个0，例如CRC8就需要补充8个0；

## 3.1.余数初始值

所谓余数初始值，就是在计算CRC值开始的时候，为CRC结果寄存器中赋的不是0，而是一个初始值。

|  |  |  |  |
| --- | --- | --- | --- |
|  | CRC-CCITT | CRC-16 | CRC-32 |
| 位宽w | 16 | 16 | 32 |
| 多项式值(除数) | 0x1021 | 0x8005 | 0x04c11db7 |
| 余数初始值 | 0xffff | 0x0000 | 0xffffffff |
| 结果异或值 | 0x0000 | 0x0000 | 0xffffffff |

## 3.2.算法描述

CRC计算过程算法描述：

1. 设置CRC寄存器的值，为余数初始值；
2. 将数据

数据结构与算法分析

# 第4章 树

定义树的一种自然的方式是递归的方法：一棵树是一些节点的集合。这个集合可是是空集；如果非空，那么一棵树由根节点r(root)以及0个或多个非空的子树T1，T2，……，Tk组成，这些子树的根节点都被来自r的一条有向的边(edge)所连接。

每一棵子树的根叫做r的儿子(child)，r是每一棵子树的父亲(parent)。从递归定义可以得知，一棵树是由N个节点和N-1条边组成的。

**深度**：对任意节点Ni，其深度指的是从根到Ni的唯一路径的长。特殊情况下，根节点的深度是0.

**高度**：对任意节点Ni，其高度指的是从Ni到一个树叶节点的最长路径的长度。特殊情况下，叶子结点的高是0，一棵树的高度等于根节点的高度。

一棵树的深度，总是等于其最深的叶子结点的深度；而这个深度，与这棵树的高度是相同的。

## 4.1.二叉树

二叉树是一棵树，其中每个节点都不能有超过2个的儿子节点。

二叉树有许多与搜索无关的重要应用，其主要用处之一是在编译器的设计领域，通过表达式树探索该问题。

### 4.1.1.表达式树

表达式树的树叶是操作数，其他的节点是操作符。例如如下的树：



预期的表达式结果是:(a + ( b \* c)) + (((d \* e) + f) \* g);

要得到这个预期的表达式，我们需要递归的产生一个带括号的左表达式，然后打印出根节点的运算符，再递归的产生一个右表达式。这种遍历方法(左-中-右)，称之为**中序遍历**。

另一个遍历策略，是递归打印出左子树、右子树，最后打印运算符，上面的树按照这种顺序应该输出：abc\*+de\*f+g\*+；这种方式(左-右-中)称为**后序遍历**。

第三种遍历策略，是先打印出运算符，再递归的打印子树，按照这种策略上面的树应输出：++a\*bc\*+\*defg；这种方式(中-左-右)称为**前序遍历**；

### 4.1.2.查找树ADT之二叉查找树

ADT, Attribute Distribute Tree，属性分散树；

二叉树的一个重要应用，是在查找中。

为了描述方便，假定每个节点都有一个关键字的值，这些关键字都假定为整数，且所有关键字互异，以后再处理有重复的情况。

使二叉树成为二叉查找树(Binary Search Tree)的性质是:对于树中的每个节点X，其左子树的所有关键字的值小于X的关键字值；右子树中所有节点关键字的值大于X的关键字的值。

这意味着，二叉查找树可以用某种统一的方式排序。

#### 插入操作

对于二叉查找树的插入操作，概念上是简单的。

为了将X插入到T中，首先使用find进行查找。如果找到了X就做一些属性值的刷新或什么都不做(做什么取决于T对于重复节点的处理策略)；否则将X插入到遍历的路径的最后一个节点上。

#### 删除操作

针对删除，需要考虑几种不同的情况：

* 叶子节点：直接删除；
* 有一个儿子节点的节点：该节点的父节点更改指向，指向其子节点，之后删除该节点即可；
* 有两个儿子节点的节点：首先找到该节点的右子树中最小的节点值；使用这个最小节点的内容更换到该节点中；删除该最小节点。由于该最小节点不可能有左儿子了，因此删除操作将异常简单。

删除更多时候可以考虑懒惰删除，也就是对节点增加一个标记位，执行删除操作时只是置位，并不实际删除。

### 4.1.3.AVL树

AVL(Adelson-Velskii and Landis)树，是其每个节点的左子树和右子树的高度最多差1的二叉查找树；

约定，空树的高度为-1；

对于AVL树，除去插入节点的动作，其他动作(查询、删除)都可以在L(logN)的时间复杂度下完成。插入操作之所以更加复杂，是因为插入一个新的节点，很有可能破坏原有的AVL特性，所以就需要调整树结构，以恢复AVL特性。这种修正AVL树特性的动作，一般称之为**旋转**(rotation)。

在插入发生时，只有那些从插入点到根节点的路径上的节点可能被打破平衡，因为只有这些节点的子树可能发生变化。当沿着这条路径，上行到根，并更新平衡信息时，我们可以找到一个节点，他的新平衡破坏了AVL条件。

我们假设这个必须重新平衡的节点是a。由于每个节点最多有2个儿子节点，因此高度不平衡时，a节点的两个子树的高度相差为2，共有4种可能性：

1. 对a节点的左儿子的左子树进行一次插入；
2. 对a节点的左儿子的右子树进行一次插入；
3. 对a节点的右儿子的左子树进行一次插入；
4. 对a节点的右儿子的右子树进行一次插入；

情况1和4，是针对a节点的镜像对称；情况2和3也是针对a节点的镜像对称。因此，理论上只需要讨论两种情况。

对于发生在“外边”的情况(情况1和4，也就是左左或右右)，通过对树进行一次单旋转(single rotation)；对于发生在“里边”的情况(情况2和3，也就是左右或右左)，通过双旋转(double rotation)来处理。

AVL树的平衡的重新建立，流程是：找到节点a(插入节点后，左右子树的高度相差2了)；分析情况；针对不同情况执行不同的旋转方式:左左单旋转、右右单旋转、左右双旋转、右左双旋转。

杂项

# 1.virtualBox使用代理服务器更新

virtualBox虚拟机中，装了ubuntu的系统，实体机是windows10，代理上网， 希望ubuntu能够apt-get到资源。

sudo vi /etc/apt/apt.conf

在您的apt.conf文件中加入下面这行（根据你的实际情况替换yourproxyaddress和proxyport）。

Acquire::http::Proxy "[http://proxyaddress:port](http://proxyaddress:port/)";

如果需要用户名密码登陆：

Acquire::http::Proxy "[http://username:password@proxyaddress:port](http://username:password@proxyaddress:port/)";  
保存apt.conf文件。（其他的协议自己可以适当修改）

# 2.svn客户端查看log时总是显示不全

[**最近**](javascript:;)遇到好几次，show log查看[**日志**](javascript:;)时，看不到最新的日志，只能看到几天前的  
日期处，也显示几天前，无法选择当前日期，还以为是版本太低，换了新版，也遇到同样情况，查了下原来是log被缓存了

[**解决**](javascript:;)**方案：**

TortoiseSVN-setting-Saved Data：

clear清空Log messages(show log dialog)

另外，也可以删除Log Caching

TortoiseSVN-setting-Log Caching-Cached Repositories

Cached Repositoried,删除对应的url记录

树莓派

# 1.系统安装和环境配置

## 1.1.系统安装

安装的是树莓派官网提供的树莓派裁剪过的debian系统，按照官网所述，现在树莓派也有专门裁剪过的ubuntu系统了，但是还没有尝试过。

系统直接安装在了tf卡上。

1. tf卡，使用读卡器，插在电脑上(windows系统即可)；
2. 使用sdFormater工具，将sd卡格式化；
3. 使用win32DiskImager工具，将下载好的镜像文件(img)，烧写在tf卡上；
4. 弹出tf卡，插在树莓派上；
5. 树莓派，连接网线、hdmi线到显示器，插键盘(蓝牙鼠键也可，默认支持蓝牙)；
6. 启动，系统正常启动，默认用户名为pi，密码是raspberry；

至此，系统安装文成；

## 1.2.环境配置

### 1.2.1.ssh服务

ssh服务，是首先需要被配置的，因为我们不可能每次都使用树莓派连接在hdmi上，而更应该在内网里，使用secureCRT或者putty等工具，远程访问；

树莓派的系统，是支持ssh服务的，但是自2016年起，它取消了ssh的默认启动。要启动，很简单，进入其根目录下，创建一个空的ssh文件，就可以了。

因为我电脑上用的linux是在虚拟机下的，所以我直接在windows下进行了这个操作：tf卡插在读卡器上，插在电脑上，进入到tf卡目录下，直接创建一个ssh文件；拔出tf卡，插到树莓派上，启动；

验证ssh服务是否开启，可以通过“ps -ef”命令，查看是否有sshd这个进程的存在；

进程存在后，局域网内，通过putty或者secureCRT，访问树莓派的IP，通过ssh协议进行连接，就可以连接了。

### 1.2.2.samba服务

我做树莓派的目的，是作为家庭服务器使用(虽然硬件很低，但是我需求也少啊……)；

所以samba服务是不可缺少的，这个可以帮助我们，在windows下直接访问linux下的目录和文件。

安装samba服务：apt-get install samba samba-common-bin；

修改配置文件：vim /etc/samba/smb.conf，最末尾加上如下内容：

[Share] #指定了windows下访问的文件名，我修改为Share，表示是共享的

comment =share directory to windows # 共享文件夹说明

path = /home/wujl/shares # 共享文件夹目录

read only = no # 不只读

create mask = 0777 # 创建文件的权限

directory mask = 0777 # 创建文件夹的权限

guest ok = yes # guest访问，无需密码

browseable = yes # 可见