# C

## 关键字及一些概念杂项

### 静态库

静态库的特点

1. 编译阶段加入程序，所以程序可以单独使用，不需要库文件
2. 编译阶段加入程序，所以每个程序使用单独静态库文件，浪费资源
3. 编译阶段加入程序，所以更新库需要重新编译文件

vs静态库生成

1. 编写库文件
2. 项目设置为静态库（找不到入口，是没有选所有配置）
3. 生成静态库
4. 找到.lib文件使用，.h文件可以不使用

静态库的使用

1. 库文件 #pragma comment(lib,"./静态库.lib")
2. 如果要使用头文件的话，也包含一下头文件

### 动态库

动态库特点：

1. 运行阶段加入程序，所以程序运行需要动态库文件
2. 多个程序可以使用一个动态库，节省空间
3. 运行阶段加入程序，所以库文件可以单独更新不需要重新编译程序

vs动态库的生成

1. 编写库文件
   1. .h文件，每个函数声明前面需要添加\_\_declspec(dllexport)
2. 项目设置为动态库
3. 生成动态库文件
4. 找到.lib文件和.dll文件放到当前目录下使用，.h文件可以不使用

动态库的使用

1. 关联库文件 #pragma comment(lib,"./动态库.lib")
2. 如果要使用头文件的话，也包含一下头文件

### 编译步骤

预处理：gcc -E hello.c -o hello.i

编 译：gcc -S hello.i -o hello.s

汇 编：gcc -c hello.s -o hello.o

链 接：gcc hello.o -o hello

### 转义字符

| **转义字符** | **含义** | **ASCII码值（十进制）** |
| --- | --- | --- |
| \a | 警报 | 007 |
| \b | 退格(BS) ，将当前位置移到前一列 | 008 |
| \f | 换页(FF)，将当前位置移到下页开头 | 012 |
| \n | 换行(LF) ，将当前位置移到下一行开头 | 010 |
| \r | 回车(CR) ，将当前位置移到本行开头 | 013 |
| \t | 水平制表(HT) （跳到下一个TAB位置） | 009 |
| \v | 垂直制表(VT) | 011 |
| \\ | 代表一个反斜线字符"\" | 092 |
| \' | 代表一个单引号（撇号）字符 | 039 |
| \" | 代表一个双引号字符 | 034 |
| \? | 代表一个问号 | 063 |
| \0 | 数字0 | 000 |
| \ddd | 8进制转义字符，d范围0~7 | 3位8进制 |
| \xhh | 16进制转义字符，h范围0~9，a~f，A~F | 3位16进制 |

### printf打印格式

%d 十进制有符号整数(int)

%u 十进制无符号整数

%o 无符号以八进制表示的整数

%x 无符号以十六进制表示的整数

%f 浮点数

%lf double

%s 字符串

%c 单个字符(char)

%p 指针的值

%e 指数形式的浮点数

%g 自动选择合适的表示法

%ld long

%02X 十六进制打印不足两位的补0

### 运算符优先级

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **优先级** | **运算符** | **名称或含义** | **使用形式** | **结合方向** | **说明** |
| **1** | **[]** | 数组下标 | 数组名[常量表达式] | 左到右 | -- |
| **()** | 圆括号 | (表达式）/函数名(形参表) | -- |
| **.** | 成员选择（对象） | 对象.成员名 | -- |
| **->** | 成员选择（指针） | 对象指针->成员名 | -- |
|  | | | | | |
| **2** | **-** | 负号运算符 | -表达式 | **右到左** | 单目运算符 |
| **~** | 按位取反运算符 | ~表达式 |
| **++** | 自增运算符 | ++变量名/变量名++ |
| **--** | 自减运算符 | --变量名/变量名-- |
| **\*** | 取值运算符 | \*指针变量 |
| **&** | 取地址运算符 | &变量名 |
| **!** | 逻辑非运算符 | !表达式 |
| **(类型)** | 强制类型转换 | (数据类型)表达式 | -- |
| **sizeof** | 长度运算符 | sizeof(表达式) | -- |
|  | | | | | |
| **3** | **/** | 除 | 表达式/表达式 | 左到右 | 双目运算符 |
| **\*** | 乘 | 表达式\*表达式 |
| **%** | 余数（取模） | 整型表达式%整型表达式 |
| **4** | **+** | 加 | 表达式+表达式 | 左到右 | 双目运算符 |
| **-** | 减 | 表达式-表达式 |
| **5** | **<<** | 左移 | 变量<<表达式 | 左到右 | 双目运算符 |
| **>>** | 右移 | 变量>>表达式 |
| **6** | **>** | 大于 | 表达式>表达式 | 左到右 | 双目运算符 |
| **>=** | 大于等于 | 表达式>=表达式 |
| **<** | 小于 | 表达式<表达式 |
| **<=** | 小于等于 | 表达式<=表达式 |
| **7** | **==** | 等于 | 表达式==表达式 | 左到右 | 双目运算符 |
| **!=** | 不等于 | 表达式!= 表达式 |
|  | | | | | |
| **8** | **&** | 按位与 | 表达式&表达式 | 左到右 | 双目运算符 |
| **9** | **^** | 按位异或 | 表达式^表达式 | 左到右 | 双目运算符 |
| **10** | **|** | 按位或 | 表达式|表达式 | 左到右 | 双目运算符 |
| **11** | **&&** | 逻辑与 | 表达式&&表达式 | 左到右 | 双目运算符 |
| **12** | **||** | 逻辑或 | 表达式||表达式 | 左到右 | 双目运算符 |
|  | | | | | |
| **13** | **?:** | 条件运算符 | 表达式1?  表达式2: 表达式3 | **右到左** | 三目运算符 |
| **14** | **=** | 赋值运算符 | 变量=表达式 | **右到左** | -- |
| **/=** | 除后赋值 | 变量/=表达式 | -- |
| **\*=** | 乘后赋值 | 变量\*=表达式 | -- |
| **%=** | 取模后赋值 | 变量%=表达式 | -- |
| **+=** | 加后赋值 | 变量+=表达式 | -- |
| **-=** | 减后赋值 | 变量-=表达式 | -- |
| **<<=** | 左移后赋值 | 变量<<=表达式 | -- |
| **>>=** | 右移后赋值 | 变量>>=表达式 | -- |
| **&=** | 按位与后赋值 | 变量&=表达式 | -- |
| **^=** | 按位异或后赋值 | 变量^=表达式 | -- |
| **|=** | 按位或后赋值 | 变量|=表达式 | -- |
|  | | | | | |
| **15** | **，** | 逗号运算符 | 表达式,表达式,… | 左到右 | -- |

### 防止头文件重复、嵌套包含

#pragma once

一种是 #ifndef 方式

### typedef(取别名)

1格式：

typedef 数据类型 需要别名

优点：

1. 简化结构体变量创建 可以省略struct关键字
2. 区分数据类型：char\* p1,p2，此种情况p1是指针，p2不是，使用typedef char\* Pchar 即可解决。
3. 提高可移植性，同一数据类型使用别名，需要改的时候，直接改一下别名就行

### sizeof(获取数据占用的空间)

注意：

1．返回的是unsingned int(无符号整型)

2．用于指针时只能获取指针的大小，一般4字。取\*也只能获取指针类型的大小

### static(静态修饰符)

1. 运行前分配内存
2. 程序运行期间一直存活
3. 只有第一次初始化有效
4. 只能在当前文件使用

### const(常量修饰符)

1. 放在全局区的const变量，无法通过指针修改
2. 一般函数参数使用const修饰的话表示这是一个传入参数，方便阅读

### 待补充

sscanf待补充

extern 待补充

### #define宏

1．预处理阶段展开

2．宏函数

* 1. #define MYADD(x,y) ((x)+(y))
  2. 多用括号防止运算优先级原因出错

3．用的多的东西可以用宏，方便修改，维护

### 位运算，左右移

~：按位取反

一个数取反结果：0变1，1变0,

&：按位与

1．一个数与另外一个数，结果：同1为1，其他为0

2．将某几位置0

|：按位或

1．一个数或另外一个数，结果：同0为0，其他为1

2．将某几位置1

^：按位异或

1. 一个数异或另外一个数，结果，相同为0，不同为1
2. 和1异或，将某几位翻转

左移：

格式：数据 << N

数据：表示需要左移的数据

<<：位移运算符

N：表示左移几位

注意：

1. 左移后，左侧头部数据被挤出，后面数据补0
2. 位运算符，不会改变数据原有的值
3. 多次位移，分开操作，

### 内存对齐

内存对齐的大小，看结构体里面的“最大的数据类型”和“对齐模数”，谁小用谁

* 1. #pragma pack(show) #pragma pack(1)，查看默认对齐模数、设置对齐模数
  2. 每一次对齐都是看“最大的数据类型”和“对齐模数”
  3. 即使结构体结束，末尾的地方也要对齐
  4. 数组，结构体，展开单个数据来看，不做整体看

## 内存、字符串操作

头文件#include <string.h>

一般来说字符串操作函数遇到’\0’都会结束

### memset()

void \*memset(void \*s, int c, size\_t n);

参数：

s：需要替换的起始地址

c：需要替换的内容，直接给字符或者ascii都行

n：替换多少个字节

返回值：指向s的指针 (基本不用不太清楚)

注意：

1. 完全使用内存操作系列函数组包时，如果写入txt可能导致乱码，用其他软件打开解决，如notepad++,vs,qt,写字板等等，原因不明，听说记事本前几个字符有默认东西

### memcmy()比较

int memcmp(const void \*s1, const void \*s2, size\_t n);

参数：

s1：比较数据1

* s2：比较数据2
* n：比较长度（字节）

返回值：

相同返回：0

不同返回：非0值

注意：

1．不同情况下返回的非0值根据字符串大小决定，字符串大小根据ASCII表决定

### memcpy()拷贝

void \*memcpy(void \*dest, const void \*src, size\_t n);

参数：

dest：拷贝目标地址

src：存有数据的地址

n：拷贝几个字节

返回值：指向dest的指针

注意：src的dest所指的内存区域不能重叠

### 分割字符串strtok()

char \*strtok(char \*str, const char \*delim);

参数：

str：需要分割的字符串

delim：分割符，可以是字符串，只要匹配到其中任何一个字符都会分割

返回值：

还能继续分割返回：分割出来的字符串

分割完成返回：NULL

注意：

1. 函数内部会自动保存剩下需要分割的字符串，所以第一次以后传NULL即可
2. 会改变参数str字符串，字符串中每个找到的分割符，都会被替换成’\0’

示例代码：

void lrStrtok()

{

char p[] = "192.1,6,8.1.1";

char pp[] = ".,";

char \*ppp;

printf("%s\n", strtok(p, pp));

while ((ppp = strtok(NULL, pp))!=NULL)

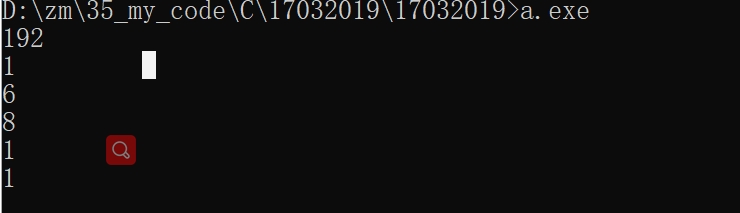
{

printf("%s\n", ppp);

}

}

运行结果：



### 查找指定字符串strstr()

char \*strstr(const char \*haystack, const char \*needle)

参数：

haystack：被查找字符串(数据)

needle：需要查找的字符（串）（务必使用字符串，不要使用字符，\0关系）

返回值：

找到返回：needle第一次出现的位置

没找到返回：NULL

注意：

1．查找指定字符第一次出现的位置：char \*strchr(const char \*s, int c);

2．查找指定字符最后一次出现的位置：char \*strrchr(const char \*s, int c);

### 拼接字符串strcat()

char \*strcat(char \*dest, const char \*src);

参数：

dest：被追加字符串

src：追加字符串

返回值：dest字符串的首地址

注意：

1. dest需要有足够的空间来存储src的数据
2. char \*strncat(char \*dest, const char \*src, size\_t n);
   1. 可以指定src的长度

### 拷贝字符串strcpy()

char \*strcpy(char \*dest, const char \*src);

参数：

dest：目标字符串

src：有数据的字符串

返回值：dest字符串起始地址

注意：

1. 如果dest所指空间内存不够，会造成缓冲溢出(buffer Overflow)错误。
2. char \*strncpy(char \*dest, const char \*src, size\_t n);
   1. 可以指定src的拷贝长度

### 字符串比较strcmp()

int strcmp(const char \*s1, const char \*s2);

参数:

s1：比较参数1

s2：比较参数2

返回值：

相同返回：0

不同返回：非0值

注意：

1. 不同情况下返回的非0值根据字符串大小决定，字符串大小根据ASCII表决定
2. int strncmp(const char \*s1, const char \*s2, size\_t n);
   1. 可以指定比较前几个字节
3. int strcasecmp(const char \*s1, const char \*s2);
4. int strncasecmp(const char \*s1, const char \*s2, size\_t n);
   1. 3.4条函数比较时可以忽略大小写

## fopen()文件操作

### fopen()

#include <stdio.h>

FILE \* fopen(const char \* filename, const char \* mode);

功能：打开文件

参数：

filename：需要打开的文件名，根据需要加上路径

mode：打开文件的模式设置

“r”： 以只读方式打开一个文本文件（不创建文件，若文件不存在则报错）

“w”：以写方式打开文件(如果文件存在则清空文件，文件不存在则创建一个文件)

“a”：以追加方式打开文件，在末尾添加内容，若文件不存在则创建文件

“r+”：以可读、可写的方式打开文件(不创建新文件)

“w+”：以可读、可写的方式打开文件(如果文件存在则清空文件，文件不存在则创建一个文件)

“a+”：以添加方式打开文件，打开文件并在末尾更改文件,若文件不存在则创建文件

返回值：

成功：文件指针

失败：NULL

注意：

1. 第二个参数每种打开模式都可以加b表示二进制方式打开，只在windows下有效
2. windows的磁盘文本文件使用\r\n（占用两个字节）的方式换行，不加b读写时都会自动转换

### fclose()

#include <stdio.h>

int fclose(FILE \* stream);

功能：关闭先前fopen()打开的文件。此动作让缓冲区的数据写入文件中，并释放系统所提供的文件资源。

参数：

stream：文件指针

返回值：

成功：0

失败：-1

### fputc()单个字符写

#include <stdio.h>

int fputc(int ch, FILE \* stream);

功能：将ch转换为unsigned char后写入stream指定的文件中

参数：

ch：需要写入文件的字符

stream：文件指针

返回值：

成功：成功写入文件的字符

失败：返回-1

### fgetc()单个字符读

#include <stdio.h>

int fgetc(FILE \* stream);

功能：从stream指定的文件中读取一个字符

参数：

stream：文件指针

返回值：

成功：返回读取到的字符

失败：-1

### fputs()写一个字符串

#include <stdio.h>

int fputs(const char \* str, FILE \* stream);

功能：将str所指定的字符串写入到stream指定的文件中。

参数：

str：需要写的数据

stream：文件指针

返回值：

成功：0

失败：-1

注意：

1. 写字符串，遇\0结束，遇到\n不会结束

### fgets()读一行字符串

#include <stdio.h>

char \* fgets(char \* str, int size, FILE \* stream);

功能：从stream指定的文件内读入字符，保存到str所指定的内存空间，直到出现换行字符、读到文件结尾或是已读了size - 1个字符为止，最后会自动加上字符 '\0' 作为字符串结束。

参数：

str：读取到的数据存放的地方

size：指定最大读取字符串的长度（size - 1）

stream：文件指针

返回值：

成功：成功读取的字符串

读到文件尾或出错： NULL

注意：

1. 遇到\n结束

### fwrite()块写

#include <stdio.h>

size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

功能：以数据块的方式给文件写入内容

参数：

ptr：准备写入文件数据的地址

size： size\_t 为 unsigned int类型，此参数指定写入文件内容的块数据大小

nmemb：写入文件的块数，写入文件数据总大小为：size \* nmemb

stream：已经打开的文件指针

返回值：

成功：实际成功写入文件数据的块数目，此值和 nmemb 相等

失败：0

### fread()块读

#include <stdio.h>

size\_t fread(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

功能：以数据块的方式从文件中读取内容

参数：

ptr：存放读取出来数据的内存空间

size： size\_t 为 unsigned int类型，此参数指定读取文件内容的块数据大小

nmemb：读取文件的块数，读取文件数据总大小为：size \* nmemb

stream：已经打开的文件指针

返回值：

成功：实际成功读取到内容的块数，如果此值比nmemb小，但大于0，说明读到文件的结尾。

失败：0

### fprintf()格式化写

#include <stdio.h>

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

功能：根据参数format字符串来转换并格式化数据，然后将结果输出到stream指定的文件中，指定出现字符串结束符 '\0' 为止。

参数：

stream：已经打开的文件

format：字符串格式，用法和printf()一样

返回值：

成功：实际写入文件的字符个数

失败：-1

注意：

1. 示例 fprintf(fp, "%d %d %d\n", 1, 2, 3);

### fscanf()格式化读

#include <stdio.h>

int fscanf(FILE \* stream, const char \* format, ...);

功能：从stream指定的文件读取字符串，并根据参数format字符串来转换并格式化数据。

参数：

stream：已经打开的文件

format：字符串格式，用法和scanf()一样

返回值：

成功：参数数目，成功转换的值的个数

失败： - 1

注意：

1. 示例
   1. int a = 0;
   2. int b = 0;
   3. int c = 0;
   4. fscanf(fp, "%d %d %d\n", &a, &b, &c);
   5. printf("a = %d, b = %d, c = %d\n", a, b, c);

### feof()是否读到文件尾

#include <stdio.h>

int feof(FILE \* stream);

功能：检测是否读取到了文件结尾。判断的是最后一次“读操作的内容”，不是当前位置内容(上一个内容)。

参数：

stream：文件指针

返回值：

非0值：已经到文件结尾

0：没有到文件结尾

### stat()获取文件状态

#include <sys/types.h>

#include <sys/stat.h>

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

功能：获取文件状态信息

参数：

path：文件名

buf：传入参数，保存文件信息的结构体

返回值：

成功：0

失败-1

注意：

1. 使用到的结构体

struct stat {

dev\_t st\_dev; //文件的设备编号

ino\_t st\_ino; //节点

mode\_t st\_mode; //文件的类型和存取的权限

nlink\_t st\_nlink; //连到该文件的硬连接数目，刚建立的文件值为1

uid\_t st\_uid; //用户ID

gid\_t st\_gid; //组ID

dev\_t st\_rdev; //(设备类型)若此文件为设备文件，则为其设备编号

off\_t st\_size; //文件字节数(文件大小)

unsigned long st\_blksize; //块大小(文件系统的I/O 缓冲区大小)

unsigned long st\_blocks; //块数

time\_t st\_atime; //最后一次访问时间

time\_t st\_mtime; //最后一次修改时间

time\_t st\_ctime; //最后一次改变时间(指属性)

};

### 删除、重命名文件

#include <stdio.h>

int remove(const char \*pathname);

功能：删除文件

参数：

pathname：文件名

返回值：

成功：0

失败：-1

#include <stdio.h>

int rename(const char \*oldpath, const char \*newpath);

功能：把oldpath的文件名改为newpath

参数：

oldpath：旧文件名

newpath：新文件名

返回值：

成功：0

失败： - 1

### fflush()强制刷新缓冲区

#include <stdio.h>

int fflush(FILE \*stream);

功能：更新缓冲区，让缓冲区的数据立马写到文件中。

参数：

stream：文件指针

返回值：

成功：0

失败：-1

### fopen()缓冲区的刷新机制

1. 满刷新
2. 关闭刷新
3. 强制刷新

注意：

1. 没有行刷新
2. 程序崩溃，没有正常结束不会触发关闭刷新

# C++

## 一些关键字和概念

### namespace命名空间

1. 用于解决名称冲突
2. 命名空间必须声明在全局作用域下
3. 命名空间可以存放变量、函数、结构体、类

namespace A

{

int m\_A = 0;

void func() {};

struct Person {};

class Animal {};

}

1. 命名空间可以嵌套命名空间

namespace B

{

int m\_A = 100;

namespace C

{

int m\_A = 200;

}

}

1. 命名空间是开放的，可以随时向命名空间下添加成员

namespace B

{

int m\_A = 100;

}

namespace B

{

int m\_B = 1000;

}

1. 命名空间可以是匿名的

namespace

{

int m\_C = 10000; // 如果命名空间 是 匿名 变量前隐式加了关键字 static

int m\_D = 20000;

}

1. 命名空间可以起别名

namespace C

{

int m\_E = 100000;

}

namespace D = C;

void test()

{

namespace E =C;

cout << E::m\_E << endl;

cout << D::m\_E << endl;

cout << C::m\_E << endl;

}

### const

1. 全局区域const变量无法修改
2. 局部区域const变量如果使用常量赋值，也无法修改，但是：
   1. 编译能成功
   2. 运行也能成功
   3. 结果不会改变，因为当前变量指向的是符号表
   4. 修改的只是一个临时空间，
3. c语言的const默认是外部链接属性，c++的默认是内部链接属性
4. c++尽量使用const代替#define

### using

好像有声明，编译指令的意思 不太懂

1. using声明

using namespace std;

1. 提前告诉编译器 sunWuKongId用 KingGlory下的

using KingGlory::sunWuKongId;

### 引用

1.语法：

数据类型 & 引用名 = 原名

2.引用必须初始化

3.对数组引用

//直接建立引用

//pArr就是arr的引用名

int arr[5] = { 1, 2, 3, 4, 5 };

int(&pArr)[5] = arr;

//先定义数组类型，再通过类型定义数组的引用

typedef int(ARRAY\_TYPE)[5];

// 数据类型 & 别名 = 原名

ARRAY\_TYPE & pArr2 = arr;

4.引用本质上是一个指向不可修改的指针，不同的引用操作，编译器会做不同的“优化”

### 内联函数

1. 函数的声明和实现都加关键字inline

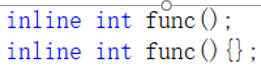
2. 用于替换宏函数

3. 类的成员函数都隐藏的加了关键字inline

4. inline只是建议编译器把该函数当做内联函数处理，具体是否是内联函数不一定

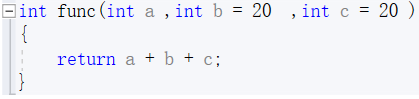
5. 不加inline关键字，编译器也可能把一些函数当做内联函数

示例：

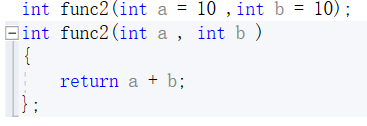


### 函数的默认参数

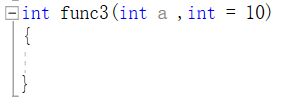
1. 如果某个位置已经有了默认参数，那么从这个位置起，后面的参数都必须有默认参数

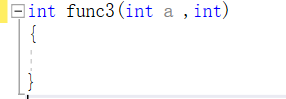


2. 函数的声明和实现只能有一个默认参数



3.占位参数





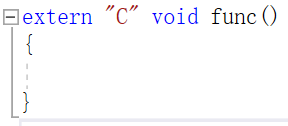
### 函数重载

1. 在同一作用域下，函数的名称相同，函数的参数个数、类型、顺序不同，代表了不同的函数

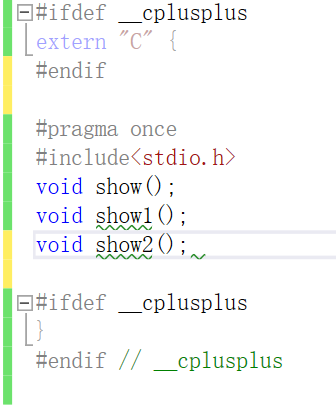
2. 实现原理是，编译器会改变函数真正的名称，所以这些函数的名称也是不同的

### exterm C

1. 方式一，单个使用



2. 方式二，批量使用、头文件使用，通过宏 \_\_cplusplus 在编译时自动判断当前编译器是C还是C++



### 异常处理

1. 关键字

throw 抛出异常

try 捕获异常 相当于if

catch 捕获异常 相当于else

2. 示例

[src\cpp\异常抛出\main.cpp](src/cpp/异常抛出/main.cpp)

多态继承的示例

[src\cpp\异常抛出\main1.cpp](src/cpp/异常抛出/main1.cpp)

### 标准输入cin

#### 1.从标准输入中读取一个字符

char ch = cin.get();

#### 2.从标准输入中读取一个字符串

cin.get( buf , sizeof(buf) );//不会读取换行符，换行符遗留在缓冲区中

cin.getline( buf , sizeof(buf) )//不会读取换行符，换行符扔掉，不会留在缓冲区

#### 3.忽略字符

cin.ignore();//忽略一个字符，括号中可以有参数，填几就忽略几个字符

#### 4.偷窥一个字符

cin.peek();//偷窥就是查看缓冲区中的下一个字符，但是不会取走

#### 5.放回字符

char ch =cin.get();

cin.putback(ch);//放回字符，原来样放回

#### 6.代码示例

[src\cpp\标准输入cin\main.cpp](src/cpp/标准输入cin/main.cpp)

### 标准输出cout

随便记一下，代码示例：

[src\cpp\标准输出cout\08 标准输出流.cpp](src/cpp/标准输出cout/08%20标准输出流.cpp)

### 文件IO

[src\cpp\文件IO\main.cpp](src/cpp/文件IO/main.cpp)

## 面向对象

### 封装

### 继承

#### 1.继承的权限

a. pulic，protected，private继承后权限不变，如果权限不够自动降级,注意父类私有属性子类无法访问

b. 详细：

公共继承

父类中公共权限成员，到子类中变为公共权限

父类中保护权限成员，到子类中变为保护权限

父类中私有权限成员，到子类中不可访问

保护继承

父类中公共权限成员，到子类中变为保护权限

父类中保护权限成员，到子类中变为保护权限

父类中私有权限成员，到子类中不可访问

私有继承

父类中公共权限成员，到子类中变为私有权限

父类中保护权限成员，到子类中变为私有权限

父类中私有权限成员，到子类中不可访问

#### 2.查看对象模型

a. 找到vs开发人员命令提示符

b. 切换到文件所在目录，输入下面命令

c. cl /d1 reportSingleClassLayout类名 文件名

#### 3.不会被继承的函数

默认构造 拷贝构造 析构 operator=

#### 4.函数重定义

子类中只要有和父类相同名称的函数，父类中的所有该名称函数将被隐藏，如需访问，可以同过加作用域访问

#### 5.创建子类对象时需要创建父类对象

a. 一般来说父类对象必须要有默认构造

b. 如果父类对象没有默认构造，子类对象的构造，需要初始化列表法给父类构造传参

### 多态

多态就是父类 指针 或 引用 调用子类对象

#### 1.动态多态的满足条件

继承关系，父类中写了虚函数，子类重写了父类中的虚函数

#### 2.多态的原理

vfptr虚函数指针， vftable虚函数表，有兴趣在深入了解

#### 3.虚函数

多态使用： 父类的指针或者引用 指向子类的对象

#### 4.纯虚函数

a. 关键字 virtual

b. 纯虚函数可以不写实现，virtual void fun()=0;

c. 类中如果有纯虚函数，这个类就是抽象类，不可以创建对象，

d. 子类继承抽象类，如果想要创建对象就必须重写父类中的纯虚函数

#### 5.虚析构和纯虚析构

a. 纯虚析构需要在类外自己写实现

b. 存在的意义，释放父类时可以自己找到子类的析构调用，避免内存泄漏

c. 可以自己释放真实对象，不用虚析构

d. 使用虚析构后delete运作模式会改变，别乱delete会出错

#### 5.函数重写

1. 子类重写父类中的虚函数就是重写

2. 重写和重载一样需要返回值、函数名、形参列表一致，才是重写

### 构造函数

1. 创建对象需要调用构造方法

2. 构造函数的语法



3. 拷贝构造，初始化对象时调用，默认是值拷贝，所以如果有堆区数据就是拷贝地址（浅拷贝）

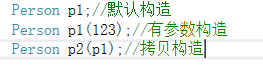
4. 默认构造的提供规则

a.默认会有无参构造和拷贝构造

b.只要提供了任何构造方法，就不会再提供无参构造

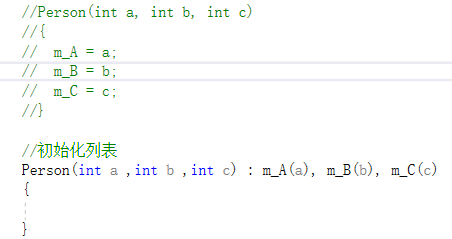
c.只要不自己提供拷贝构造，都会有默认的拷贝构造

5. 构造的调用





6. 初始化列表，其中参数有对象也可以使用



### new

1. 创建存放在堆区的对象，用完需要delete释放；

2. 如果利用new在堆区开辟 自定义类型数据的数组，必须有默认构造函数

a.Person \* personArray = new Person[10];

b.释放堆区数组 delete [] personArray;

### struct和class

1. 在c++下struct和class是一样的，只有默认权限不同

2. struct默认权限是public class是private

### 类的静态成员

1. 独立于对象存在，所以只有一份所有对象共用，也可以不使用对象单独使用

2. 独立于对象所以，静态属性函数外部赋值

3. 独立于对象所以，静态方法无法使用非对象静态成员

### 单例模式

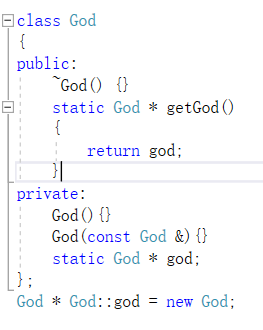
1. 类只能创建一个对象就叫单例模式

2. 私有化所有构造

3. 创建一个私有的静态成员存储唯一对象的指针，该成员在类的外部赋值

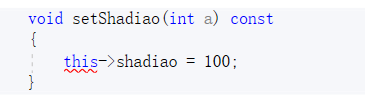
4. 可以有一个公共方法返回这个唯一对象指针

5. 分文件使用时类外创建对象要放在CPP里面



### 类的常函数和常对象

1. 常函数表示this的值无法修改，(声明该函数不会改变成员属性的值)



2. 常对象成员属性(this)的值无法修改，常对象只能调用常函数，这是为了防止常对象的成员属性被修改



3. 可以有特殊成员，用 mutable 关键字声明，声明后该值可以在常函数中修改



### 友元

1. 友元就是允许对方访问该类的私有成员

2. 全局函数作为友元



3. 成员函数为友元



4. 类作为友元



### 运算符重载

1. 主要是让对象能使用运算符，方便操作，减少代码量

2. operator关键字

3. 代码示例

[src\cpp\运算符重载\test.cpp](src/cpp/运算符重载/test.cpp)

[src\cpp\运算符重载\MyString.h](src/cpp/运算符重载/MyString.h)

[src\cpp\运算符重载\MyString.cpp](src/cpp/运算符重载/MyString.cpp)

## 模板

模板也叫泛形编程 关键字 template<>

示例1函数模板：



示例2类模板：



代码示例1模板应用：

[src\cpp\模板\main.cpp](src/cpp/模板/main.cpp)

[src\cpp\模板\Person.hpp](src/cpp/模板/Person.hpp)

代码示例2自定义数组案例：

[src\cpp\模板\自定义数组案例\ARR.hpp](src/cpp/模板/自定义数组案例/ARR.hpp)

[src\cpp\模板\自定义数组案例\mian.cpp](src/cpp/模板/自定义数组案例/mian.cpp)

笔记示例：

[src\cpp\模板\Day18笔记.docx](src/cpp/模板/Day18笔记.docx)

## string类

### 1. 构造函数：

string();//创建一个空的字符串 例如: string str;

string(const string& str);//使用一个string对象初始化另一个string对象

string(const char\* s);//使用字符串s初始化

string(int n, char c);//使用n个字符c初始化

### 2. 基本赋值操作：

string& operator=(const char\* s);//char\*类型字符串 赋值给当前的字符串

string& operator=(const string &s);//把字符串s赋给当前的字符串

string& operator=(char c);//字符赋值给当前的字符串

string& assign(const char \*s);//把字符串s赋给当前的字符串

string& assign(const char \*s, int n);//把字符串s的前n个字符赋给当前的字符串

string& assign(const string &s);//把字符串s赋给当前字符串

string& assign(int n, char c);//用n个字符c赋给当前字符串

string& assign(const string &s, int start, int n);//将s从start开始n个字符赋值给字符串

### 3. 基本字符操作：

char& operator[](int n);//通过[]方式取字符

char& at(int n);//通过at方法获取字符

### 4. string拼接操作：

string& operator+=(const string& str);//重载+=操作符

string& operator+=(const char\* str);//重载+=操作符

string& operator+=(const char c);//重载+=操作符

string& append(const char \*s);//把字符串s连接到当前字符串结尾

string& append(const char \*s, int n);//把字符串s的前n个字符连接到当前字符串结尾

string& append(const string &s);//同operator+=()

string& append(const string &s, int pos, int n);//把字符串s中从pos开始的n个字符连接到当前字符串结尾

string& append(int n, char c);//在当前字符串结尾添加n个字符c

### 5. string比较操作：

compare函数在>时返回 1，<时返回 -1，==时返回 0。

比较区分大小写，比较时参考字典顺序，排越前面的越小。

大写的A比小写的a小。

int compare(const string &s) const;//与字符串s比较

int compare(const char \*s) const;//与字符串s比较

### 6. string查找和替换：没找到返回-1

int find(const string& str, int pos = 0) const; //查找str第一次出现位置,从pos开始查找

int find(const char\* s, int pos = 0) const; //查找s第一次出现位置,从pos开始查找

int find(const char\* s, int pos, int n) const; //从pos位置查找s的前n个字符第一次位置

int find(const char c, int pos = 0) const; //查找字符c第一次出现位置

int rfind(const string& str, int pos = npos) const;//查找str最后一次位置,从pos开始查找

int rfind(const char\* s, int pos = npos) const;//查找s最后一次出现位置,从pos开始查找

int rfind(const char\* s, int pos, int n) const;//从pos查找s的前n个字符最后一次位置

int rfind(const char c, int pos = 0) const; //查找字符c最后一次出现位置

string& replace(int pos, int n, const string& str); //替换从pos开始n个字符为字符串str

string& replace(int pos, int n, const char\* s); //替换从pos开始的n个字符为字符串s

### 7. string子串：

string substr(int pos = 0, int n = npos) const;//返回由pos开始的n个字符组成的字符串

### 8.string插入和删除操作：

string& insert(int pos, const char\* s); //插入字符串

string& insert(int pos, const string& str); //插入字符串

string& insert(int pos, int n, char c);//在指定位置插入n个字符c

string& erase(int pos, int n = npos);//删除从Pos开始的n个字符

### 9. string和c-style字符串转换：

string 转 char\*

string str = "itcast";

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

char\* 转 string

char\* s = "itcast";

string str(s);

10. 案例：

[src\cpp\string类\main.cpp](src/cpp/string类/main.cpp)

## STL标准模板库

### 1. 迭代器

//迭代器的定义

//容器::iterator 迭代器名 = 返回迭代器的方法;

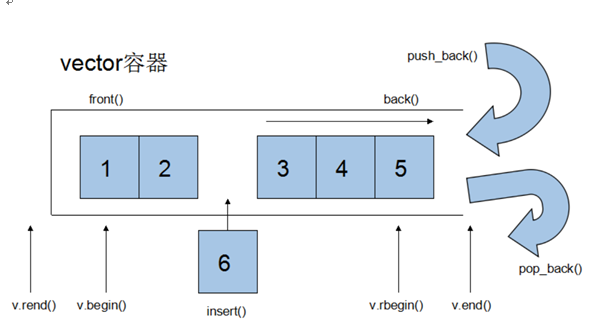
list<Person>::iterator it= p.begin();

|  |  |  |
| --- | --- | --- |
| 输入迭代器 | 提供对数据的只读访问 | 只读，支持++、==、！= |
| 输出迭代器 | 提供对数据的只写访问 | 只写，支持++ |
| 前向迭代器 | 提供读写操作，并能向前推进迭代器 | 读写，支持++、==、！= |
| 双向迭代器 | 提供读写操作，并能向前和向后操作 | 读写，支持++、--， |
| 随机访问迭代器 | 提供读写操作，并能以跳跃的方式访问容器的任意数据，是功能最强的迭代器 | 读写，支持++、--、[n]、-n、<、<=、>、>= |

### 2. 容器

#### 1. vector容器

单端动态数组，连续空间，有未雨绸缪机制。可预留空间，提高效率。头部操作效率低。使用随机迭代器



拷贝构造:

vector<T> v; //采用模板实现类实现，默认构造函数

vector(v.begin(), v.end());//将v[begin(), end())区间中的元素拷贝给本身。

vector(n, elem);//构造函数将n个elem拷贝给本身。

vector(const vector &vec);//拷贝构造函数。

赋值操作

assign(beg, end);//将[beg, end)区间中的数据拷贝赋值给本身。

assign(n, elem);//将n个elem拷贝赋值给本身。

vector& opera2tor=(const vector &vec);//重载等号操作符

swap(vec);// 将vec与本身的元素互换。

vector的大小操作：

size();//返回容器中元素的个数

empty();//判断容器是否为空，判断size

resize(int num);//重新指定容器的长度为num，若容器变长，则以默认值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。

resize(int num, elem);//重新指定容器的长度为num，若容器变长，则以elem值填充新位置。如果容器变短，则末尾超出容器长>度的元素被删除。

capacity();//容器的容量

reserve(int len);//给容器预留容量，避免重复开辟、释放，提高效率

vector数据存取操作：

at(int idx); //返回索引idx所指的数据，如果idx越界，抛出out\_of\_range异常。

operator[];//返回索引idx所指的数据，越界时，运行直接报错

front();//返回容器中第一个数据元素

back();//返回容器中最后一个数据元素

vector插入和删除操作：

insert(const\_iterator pos, int count, ele);//迭代器指向位置pos插入count个元素ele.

push\_back(ele); //尾部插入元素ele

pop\_back();//删除最后一个元素

erase(const\_iterator start, const\_iterator end);//删除迭代器从start到end之间的元素

erase(const\_iterator pos);//删除迭代器指向的元素

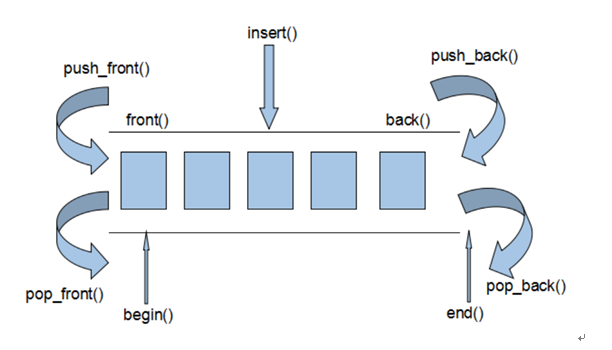
clear();//删除容器中所有元素，删除的是元素个数，空间容量不变

案例：

[src\cpp\STL标准模板库\vector单端动态数组\main.cpp](src/cpp/STL标准模板库/vector单端动态数组/main.cpp)

#### 2.deque容器

双端队列，非连续空间，使用随机迭代器



构造方法：

deque<T> deqT;//默认构造形式

deque(beg, end);//构造函数将[beg, end)区间中的元素拷贝给本身。

deque(n, elem);//构造函数将n个elem拷贝给本身。

deque(const deque &deq);//拷贝构造函数。

deque赋值操作：

assign(beg, end);//将[beg, end)区间中的数据拷贝赋值给本身。

assign(n, elem);//将n个elem拷贝赋值给本身。

deque& operator=(const deque &deq); //重载等号操作符

swap(deq);// 将deq与本身的元素互换

deque大小操作：

deque.size();//返回容器中元素的个数

deque.empty();//判断容器是否为空

deque.resize(num);//重新指定容器的长度为num,若容器变长，则以默认值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。

deque.resize(num, elem); //重新指定容器的长度为num,若容器变长，则以elem值填充新位置,如果容器变短，则末尾超出容器长度的元素被删除。

deque双端插入和删除操作：

push\_back(elem);//在容器尾部添加一个数据

push\_front(elem);//在容器头部插入一个数据

pop\_back();//删除容器最后一个数据

pop\_front();//删除容器第一个数据

deque数据存取：

at(idx);//返回索引idx所指的数据，如果idx越界，抛出out\_of\_range。

operator[];//返回索引idx所指的数据，如果idx越界，不抛出异常，直接出错。

front();//返回第一个数据。

back();//返回最后一个数据

deque插入操作：

insert(pos, elem);//在pos位置插入一个elem元素的拷贝，返回新数据的位置。

insert(pos, n, elem);//在pos位置插入n个elem数据，无返回值。

insert(pos, beg, end);//在pos位置插入[beg,end)区间的数据，无返回值。

deque删除操作：

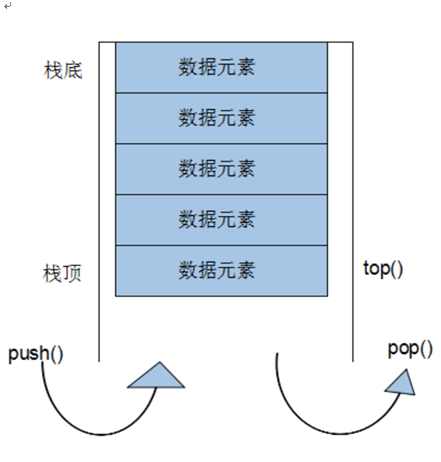
clear();//移除容器的所有数据

erase(beg, end);//删除[beg,end)区间的数据，返回下一个数据的位置。

erase(pos);//删除pos位置的数据，返回下一个数据的位置。

#### 3. stack容器

栈容器，没有迭代器



stack构造函数：

stack<T> stkT;//stack采用模板类实现， stack对象的默认构造形式：

stack(const stack &stk);//拷贝构造函数

stack赋值操作：

stack& operator=(const stack &stk);//重载等号操作符

stack数据存取操作：

push(elem);//向栈顶添加元素

pop();//从栈顶移除第一个元素

top();//返回栈顶元素

stack大小操作：

empty();//判断堆栈是否为空

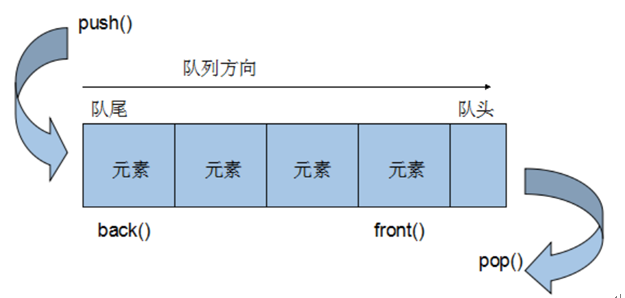
size();//返回堆栈的大小

代码示例：

[src\cpp\STL标准模板库\stack栈容器\main.cpp](src/cpp/STL标准模板库/stack栈容器/main.cpp)

#### 4. queue容器

队列容器，没有迭代器



queue构造函数：

queue<T> queT;//queue采用模板类实现，queue对象的默认构造形式：

queue(const queue &que);//拷贝构造函数

queue存取、插入和删除操作：

push(elem);//往队尾添加元素

pop();//从队头移除第一个元素

back();//返回最后一个元素

front();//返回第一个元素

queue赋值操作：

queue& operator=(const queue &que);//重载等号操作符

queue大小操作：

empty();//判断队列是否为空

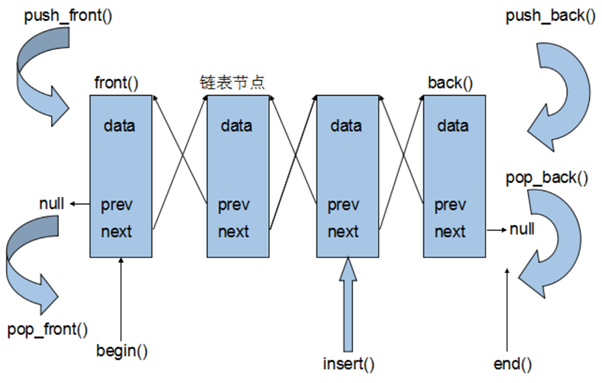
size();//返回队列的大小

代码示例：

[src\cpp\STL标准模板库\queue队列\main.cpp](src/cpp/STL标准模板库/queue队列/main.cpp)

#### 5. list容器

双向链表，使用双向迭代器



list构造函数：

list<T> lstT;//list采用采用模板类实现,对象的默认构造形式：

list(beg, end);//构造函数将[beg, end)区间中的元素拷贝给本身。

list(n, elem);//构造函数将n个elem拷贝给本身。

list(const list &lst);//拷贝构造函数。

list数据元素插入和删除操作：

push\_back(elem);//在容器尾部加入一个元素

pop\_back();//删除容器中最后一个元素

push\_front(elem);//在容器开头插入一个元素

pop\_front();//从容器开头移除第一个元素

insert(pos, elem);//在pos位置插elem元素的拷贝，返回新数据的位置。

insert(pos, n, elem);//在pos位置插入n个elem数据，无返回值。

insert(pos, beg, end);//在pos位置插入[beg,end)区间的数据，无返回值。

clear();//移除容器的所有数据

erase(beg, end);//删除[beg,end)区间的数据，返回下一个数据的位置。

erase(pos);//删除pos位置的数据，返回下一个数据的位置。

remove(elem);//删除容器中所有与elem值匹配的元素。

list大小操作：

size();//返回容器中元素的个数

empty();//判断容器是否为空

resize(num);//重新指定容器的长度为num，

若容器变长，则以默认值填充新位置。

如果容器变短，则末尾超出容器长度的元素被删除。

resize(num, elem);//重新指定容器的长度为num，

若容器变长，则以elem值填充新位置。

如果容器变短，则末尾超出容器长度的元素被删除。

list赋值操作：

assign(beg, end);//将[beg, end)区间中的数据拷贝赋值给本身。

assign(n, elem);//将n个elem拷贝赋值给本身。

list& operator=(const list &lst);//重载等号操作符

swap(lst);//将lst与本身的元素互换。

list数据的存取：

front();//返回第一个元素。

back();//返回最后一个元素。

list反转排序：

reverse();//反转链表，比如lst包含1,3,5元素，运行此方法后，lst就包含5,3,1元素。

sort(); //list排序

代码示例：

[src\cpp\STL标准模板库\list双向链表\main.cpp](src/cpp/STL标准模板库/list双向链表/main.cpp)

#### 6. set/multiset容器

set容器，插入数据会自动排序的容器，不允许元素重复

multiset和set一样但是允许元素重复

使用什么迭代器暂时不清楚待补充

构造函数：

set<T> st;//set默认构造函数：

mulitset<T> mst; //multiset默认构造函数:

set(const set &st);//拷贝构造函数

set赋值操作：

set& operator=(const set &st);//重载等号操作符

swap(st);//交换两个集合容器

set大小操作：

size();//返回容器中元素的数目

empty();//判断容器是否为空

set插入和删除操作：

insert(elem);//在容器中插入元素。

clear();//清除所有元素

erase(pos);//删除pos迭代器所指的元素，返回下一个元素的迭代器。

erase(beg, end);//删除区间[beg,end)的所有元素 ，返回下一个元素的迭代器。

erase(elem);//删除容器中值为elem的元素。

set查找操作：

find(key);//查找键key是否存在,若存在，返回该键的元素的迭代器；若不存在，返回set.end();

count(key);//查找键key的元素个数

lower\_bound(keyElem);//返回第一个key>=keyElem元素的迭代器。

upper\_bound(keyElem);//返回第一个key>keyElem元素的迭代器。

equal\_range(keyElem);//返回容器中key与keyElem相等的上下限的两个迭代器。

代码示例:

[src\cpp\STL标准模板库\set容器\main.cpp](src/cpp/STL标准模板库/set容器/main.cpp)

#### 7. map/multimap容器

map容器可以有一个键，一个值

键和set容器是一样的

map构造函数：

map<T1, T2> mapTT;//map默认构造函数:

map(const map &mp);//拷贝构造函数

map赋值操作：

map& operator=(const map &mp);//重载等号操作符

swap(mp);//交换两个集合容器

map大小操作：

size();//返回容器中元素的数目

empty();//判断容器是否为空

map插入数据元素操作：

map.insert(...); //往容器插入元素，返回pair<iterator,bool>

map<int, string> mapStu;

// 第一种 通过pair的方式插入对象

mapStu.insert(pair<int, string>(3, "小张"));

// 第二种 通过pair的方式插入对象

mapStu.inset(make\_pair(-1, "校长"));

// 第三种 通过value\_type的方式插入对象

mapStu.insert(map<int, string>::value\_type(1, "小李"));

// 第四种 通过数组的方式插入值

mapStu[3] = "小刘";

mapStu[5] = "小王";

map删除操作：

clear();//删除所有元素

erase(pos);//删除pos迭代器所指的元素，返回下一个元素的迭代器。

erase(beg, end);//删除区间[beg,end)的所有元素 ，返回下一个元素的迭代器。

erase(keyElem);//删除容器中key为keyElem的对组。

map查找操作：

find(key);//查找键key是否存在,若存在，返回该键的元素的迭代器；/若不存在，返回map.end();

count(keyElem);//返回容器中key为keyElem的对组个数。对map来说，要么是0，要么是1。对multimap来说，值可能大于1。

lower\_bound(keyElem);//返回第一个key>=keyElem元素的迭代器。

upper\_bound(keyElem);//返回第一个key>keyElem元素的迭代器。

equal\_range(keyElem);//返回容器中key与keyElem相等的上下限的两个迭代器。

代码示例：

[src\cpp\STL标准模板库\map容器\main.cpp](src/cpp/STL标准模板库/map容器/main.cpp)

#### 8. pair<>对组

1. 示例

pair<set<int>::iterator, set<int>::iterator> pit = s1.equal\_range(5);

cout << "下一个元素的迭代器\*(pid.second)="<<\*(pit.second) << endl;//下一个元素的迭代器

cout << "当前元素的迭代器\*(pid.first)="<<\*(pit.first) << endl;;//当前元素的迭代器

2. 示例

pair<map<int, Person>::iterator, map<int, Person>::iterator> dz = p.equal\_range(2);

cout << "ID=\t" << dz.first->first << endl;

cout << "age=\t" << dz.first->second.age << endl;

cout << "name=\t"<<dz.first->second.name << endl;

cout<<endl << endl;

cout << "ID=\t" << dz.second->first << endl;

cout << "age=\t" << dz.second->second.age << endl;

cout << "name=\t" << dz.second->second.name << endl;

### 3. 算法

内建函数

template<class T> bool less<T>//小于 降序排列

template<class T> bool greater<T>//大于 升序排列

#### 1. 算法头文件

1. 算法主要是由头文件<algorithm> <functional> <numeric>组成。

2. <algorithm>是所有STL头文件中最大的一个,其中常用的功能涉及到比较，交换，查找,遍历，复制，修改，反转，排序，合并等...

3. <numeric>体积很小，只包括在几个序列容器上进行的简单运算的模板函数.

4. <functional> 定义了一些模板类,用以声明函数对象。

#### 2. 遍历算法

##### 1. 遍历算法 遍历容器元素

@param beg 开始迭代器

@param end 结束迭代器

@param \_callback 函数回调或者函数对象

@return 函数对象

for\_each(iterator beg, iterator end, \_callback);

##### 2. transform算法 将指定容器区间元素搬运到另一容器中

注意 :目标不知道为什么要用resize指定长度，不然报错

@param beg1 源容器开始迭代器

@param end1 源容器结束迭代器

@param beg2 目标容器开始迭代器

@param \_cakkback 回调函数或者函数对象

@return 返回目标容器迭代器

for\_each(iterator beg1, iterator end1, iterator beg2, \_callbakc)

##### 3. 代码：

[src\cpp\STL标准模板库\遍历算法\main.cpp](src/cpp/STL标准模板库/遍历算法/main.cpp)

#### 3.查找算法

##### find算法 查找元素

@param beg 容器开始迭代器

@param end 容器结束迭代器

@param value 查找的元素

@return 返回第一个找到元素的迭代器

find(iterator beg, iterator end, value)

##### find\_if算法 条件查找

@param beg 容器开始迭代器

@param end 容器结束迭代器

@param callback 回调函数或者谓词(返回bool类型的函数对象)

@return 返回第一个找的元素的迭代器

find\_if(iterator beg, iterator end, \_callback);

##### adjacent\_find算法 查找相邻重复元素

@param beg 容器开始迭代器

@param end 容器结束迭代器

@return 返回相邻元素的第一个位置的迭代器

adjacent\_find(iterator beg, iterator end);

##### binary\_search算法 二分查找法

注意: 在无序序列中不可用

@param beg 容器开始迭代器

@param end 容器结束迭代器

@param value 查找的元素

@return bool 查找返回true 否则false

bool binary\_search(iterator beg, iterator end, value);

##### count算法 统计元素出现次数

@param beg 容器开始迭代器

@param end 容器结束迭代器

@return int返回元素个数

##### count\_if算法 统计元素出现次数

@param beg 容器开始迭代器

@param end 容器结束迭代器

@param callback 回调函数或者谓词(返回bool类型的函数对象)

@return int返回元素个数

count\_if(iterator beg, iterator end, \_callback);

##### 代码示例

[src\cpp\STL标准模板库\查找算法\main.cpp](src/cpp/STL标准模板库/查找算法/main.cpp)

#### 4. 排序算法

##### merge算法 容器元素合并，并存储到另一容器中

注意 : 两个容器必须是有序的

@param beg1 容器1开始迭代器

@param end1 容器1结束迭代器

@param beg2 容器2开始迭代器

@param end2 容器2结束迭代器

@param dest 目标容器开始迭代器

##### sort算法 容器元素排序

@param beg 容器1开始迭代器

@param end 容器1结束迭代器

@param \_callback 回调函数或者谓词(返回bool类型的函数对象)

sort(iterator beg, iterator end, \_callback)

##### random\_shuffle算法 对指定范围内的元素随机调整次序

@param beg 容器开始迭代器

@param end 容器结束迭代器

random\_shuffle(iterator beg, iterator end);

##### reverse算法 反转指定范围的元素

@param beg 容器开始迭代器

@param end 容器结束迭代器

reverse(iterator beg, iterator end)

##### 代码示例：

[src\cpp\STL标准模板库\排序算法\main.cpp](src/cpp/STL标准模板库/排序算法/main.cpp)

#### 5. 拷贝和替换算法

##### copy算法 将容器内指定范围的元素拷贝到另一容器中

@param beg 容器开始迭代器

@param end 容器结束迭代器

@param dest 目标起始迭代器

copy(iterator beg, iterator end, iterator dest)

##### replace算法 将容器内指定范围的旧元素修改为新元素

@param beg 容器开始迭代器

@param end 容器结束迭代器

@param oldvalue 旧元素

@param newvalue 新元素

replace(iterator beg, iterator end, oldvalue, newvalue)

##### replace\_if算法 将容器内指定范围满足条件的元素替换为新元素

@param beg 容器开始迭代器

@param end 容器结束迭代器

@param callback函数回调或者谓词(返回Bool类型的函数对象)

@param oldvalue 新元素

replace\_if(iterator beg, iterator end, \_callback, newvalue);

##### swap算法 互换两个容器的元素

@param c1容器1

@param c2容器2

swap(container c1, container c2);

##### 代码示例：

[src\cpp\STL标准模板库\拷贝和替换算法\main.cpp](src/cpp/STL标准模板库/拷贝和替换算法/main.cpp)

#### 6. 常用算术算法

##### accumulate算法 计算容器元素累计总和

@param beg 容器开始迭代器

@param end 容器结束迭代器

@param value累加值

accumulate(iterator beg, iterator end, value)

##### fill算法 向容器中添加元素

@param beg 容器开始迭代器

@param end 容器结束迭代器

@param value t填充元素

fill(iterator beg, iterator end, value)

##### 代码示例：

[src\cpp\STL标准模板库\常用算数生成算法\main.cpp](src/cpp/STL标准模板库/常用算数生成算法/main.cpp)

#### 7. 常用的集合算法

##### set\_intersection算法 求两个set集合的交集

注意:两个集合必须是有序序列

@param beg1 容器1开始迭代器

@param end1 容器1结束迭代器

@param beg2 容器2开始迭代器

@param end2 容器2结束迭代器

@param dest 目标容器开始迭代器

@return 目标容器的最后一个元素的迭代器地址

##### set\_union算法 求两个set集合的并集

注意:两个集合必须是有序序列

@param beg1 容器1开始迭代器

@param end1 容器1结束迭代器

@param beg2 容器2开始迭代器

@param end2 容器2结束迭代器

@param dest 目标容器开始迭代器

@return 目标容器的最后一个元素的迭代器地址

##### set\_union算法 求两个set集合的并集

注意:两个集合必须是有序序列

@param beg1 容器1开始迭代器

@param end1 容器1结束迭代器

@param beg2 容器2开始迭代器

@param end2 容器2结束迭代器

@param dest 目标容器开始迭代器

@return 目标容器的最后一个元素的迭代器地址

##### 代码示例：

[src\cpp\STL标准模板库\常用集合算法\main.cpp](src/cpp/STL标准模板库/常用集合算法/main.cpp)

# Qt

布局使用wiget

## 创建项目的三个基类

Qwidget：

1.其他两个基类的父类，是一个空窗口

QMainWindow :

1.自带一些控件的窗口，菜单栏，工具栏，铆接部件等

2.可以通过函数返回控制控件的对象

## 对象树

Qt中new出来的对象如果设置了父对象窗口，且父对象继承自QObject该对象就可以不手动释放，在父对象关闭后会自动释放

## 信号与槽

1、绑定信号和槽

connect();

参数1：发送信号的对象

参数2：信号函数的入口地址

参数3：使用槽函数的对象

参数4：槽函数的入口地址，可以使用Lambda表达式

//绑定信号

void(XueSheng:: \*pp)() = &XueSheng::qk;

void(LaoShi:: \*pp1)() = &LaoShi::chifan;

connect(this->ls,pp1,this->xs,pp);

//connect(this->ls,&LaoShi::chifan,this->xs,&XueSheng::qk);

//有参数的信号绑定槽函数，需要使用函数指针

void(XueSheng:: \*p)(QString) = &XueSheng::qk;

void(LaoShi:: \*p1)(QString) = &LaoShi::chifan;

connect(this->ls,p1,this->xs,p);

//Lambda表达式作为槽函数

connect(btn,&QPushButton::clicked,this,[=](){

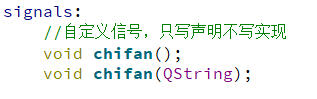
btn->setText("呵呵和");

});

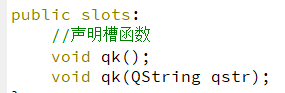
//触发自定义信号,使用关键字 emit

emit this->ls->chifan();

//自定义信号



//自定槽函数的声明，槽函数需要写实现



2、一图了解信号和槽

[src\Qt\信号和槽\Qt信号和槽 .png](src/Qt/信号和槽/Qt信号和槽%20.png)

## Lambda表达式

格式：

[](){}：表示函数入口地址

[](){}():表示函数调用

1.[]：表达式里面使用外部数据，=表示值拷贝，&表示引用传递

2.()：里面可以写参数，一般不写

3.{}：里面写函数体

## 多线程的使用方式2

1、写一个类继承obj类，里面写一个槽函数，槽函数里写要在其他线程执行的代码

2、

实例化自己写的类，要new出来的，不要指定父对象

实例化一个Qthread对象，要new出来的，不要指定父对象

3、把自己的写的类，对象放入Qthread对象中

我的写的类对象->moveToThread(Qthread对象);

4、绑定信号

a、绑定槽函数启动信号：把Qthread对象的started信号，绑定到我写的类的槽函数

b、释放Qthread对象的信号：把 Qthread对象的finished，绑定到Qthread对象 的deleteLater

c、释放自己写的类对象的信号：，在类里面写一个槽函数，delete this，也绑定Qthread对象的finished信号

5、启动线程：

a、调用Qthread对象的start方法，发出信号，开始执行自己写的槽函数

b、如果线程没有回收，再次调用start方法无用

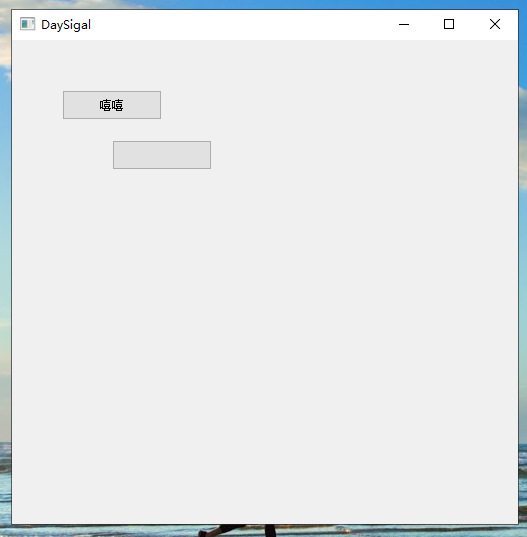
6、回收子线程

a、调用Qthread对象的quit和exit方法

b、调用quit和exit方法后，如果槽函数代码已经运行完毕，会销毁线程，同时发送finished信号，此时可以释放对象，同时也可以，再次调用start启动线程。 如果槽函数还没有运行完毕，等到槽函数运行完毕后才会销毁线程，发送finished信号，可以再次调用start启动线程

## Qt对象（控件）

### Qwidget-->空窗口



//设置窗口名字

this->setWhatsThis("窗口");

// 设置窗口大小,大小可拖拽修改，大小单位应该是分辨率，像素

this->resize(600,400);

// 设置窗口大小,大小不可以拖拽修改

this->setFixedSize(600,400);

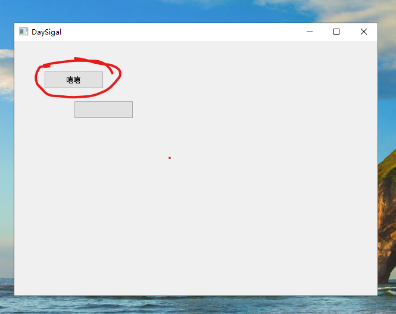
//创建按钮，设置按钮名字，给出按钮归属

QPushButton \*btn =new QPushButton("关闭",this);

//创建按钮，设置按钮名字，给出按钮归属

QPushButton \*btn =new QPushButton("关闭",this);

### QPushButton->按钮



//实例化按钮

QPushButton \*bt1 =new QPushButton();

//显示按钮

bt1->show();

//设置按钮父对象

bt1->setParent(this);

//设置按钮文本

bt1->setText("呵呵");

//移动按钮位置

bt1->move(200,300);

//按钮单击信号

//&QPushButton::clicked单击信号

//&Widget::close窗口关闭槽函数

connect(btn,&QPushButton::clicked,this,&Widget::close);

### QMainWindow ->菜单



#### QmenuBar->菜单栏

//菜单栏 只能有一个

QMenuBar \* bar =menuBar();

#### Qmenu->菜单

//添加菜单

QMenu \* fileMenu = bar->addMenu("文件");

QMenu \* editMenu = bar->addMenu("编辑");

#### Qaction->菜单项

//添加菜单项分割线

fileMenu->addSeparator();

//添加菜单项

QAction \* openActino =fileMenu->addAction("新建");

//菜单项的单击事件

connect(newActino,&QAction::triggered,this,[](){

qDebug()<<"呵呵";

});

//给菜单项添加图片 保存图片

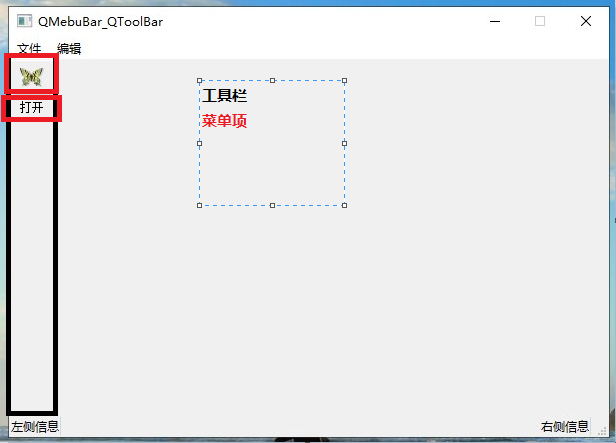
QPixmap pix;

pix.load(":/new/prefix1/Image/butterfly.png");

//添加图片

newActino->setIcon(QIcon(pix));

### QToolBar ->工具栏



//创建工具栏 可以有多个

QToolBar \* toolBar = new QToolBar(this);

//状态栏放入当前窗口

//参数1：设置工具栏默认位置

//Qt::TopToolBarArea 上

//Qt::BottomToolBarArea 下

//Qt::LeftToolBarArea 左

//Qt::RightToolBarArea 右

this->addToolBar(Qt::LeftToolBarArea,toolBar);

//只允许左右停靠

toolBar->setAllowedAreas(Qt::LeftToolBarArea

| Qt::RightToolBarArea );

//不允许浮动在中间 必须停靠在一个地方

toolBar->setFloatable(false);

//不允许拖拽 固定位置

toolBar->setMovable(false);

//给工具栏添加菜单项

toolBar->addAction(newActino);

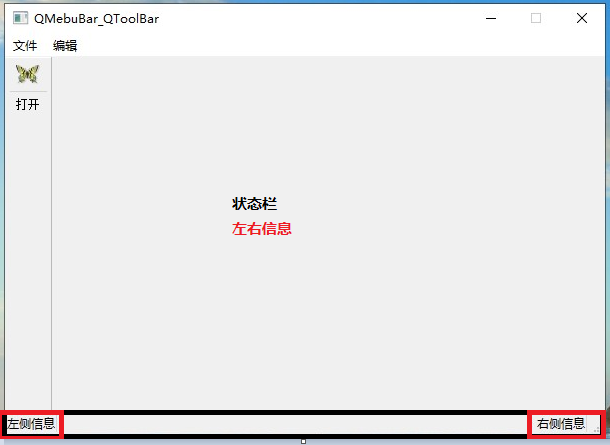
//添加菜单项之间的分割线

toolBar->addSeparator();

//给工具栏添加菜单项

toolBar->addAction(openActino);

### QStatusBar->状态栏



/状态栏 只能有一个

QStatusBar \* stBar = statusBar();

this->setStatusBar(stBar);

//左侧信息

QLabel \*label1 = new QLabel("左侧信息");

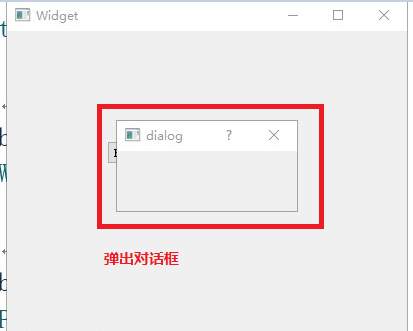
stBar->addWidget(label1);

//右侧信息

QLabel \*label2 = new QLabel("右侧信息");

stBar->addPermanentWidget(label2);

### Qdialog->弹出对话框



//创建对话框（模态）

QDialog dl(this);

//设置对话框大小

dl.resize(150,50);

//已模态、阻塞的显示

dl.*exec*();

//非模态对话框创建

QDialog \*dl1 =new QDialog(this);

//设置对话框大小

dl1->resize(180,60);

//正常显示，不阻塞、非模态

dl1->show();

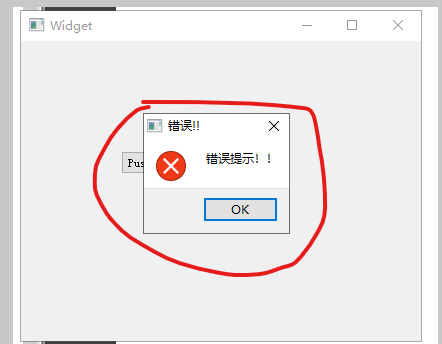
//该对话框关闭后，对象释放

dl1->setAttribute(Qt::WA\_DeleteOnClose);

注意：非模态对话框因为不带阻塞，代码执行出了作用域会被自动释放，所以需要使用NEW出来的对象

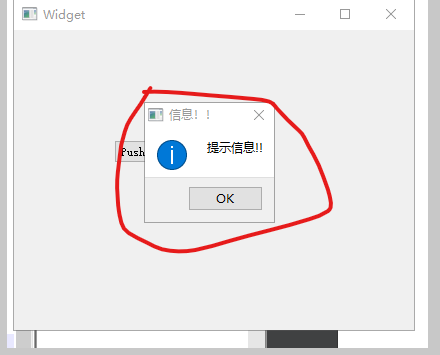
### QmessageBox->消息对话框（模态）

#### 错误对话框



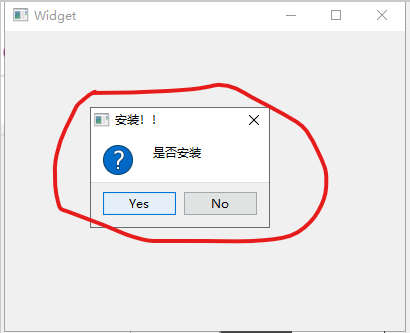
QMessageBox::critical(this,"错误!!","错误提示！！");

#### 提示信息对话框



QMessageBox::information(this,"信息！！","提示信息!!");

#### 询问对话框



QMessageBox::StandardButton ret;

//创建询问对话框

ret = QMessageBox::question(this,"安装！！","是否安装");

//判断询问对话框的选择

if(ret == QMessageBox::Yes)

{

qDebug()<<"yes";

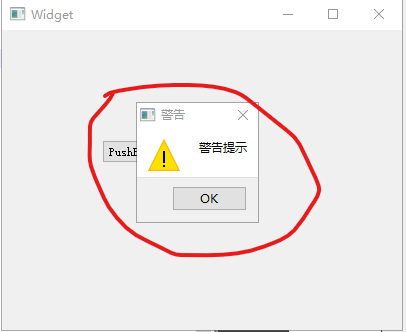
}else

{

qDebug()<<"no";

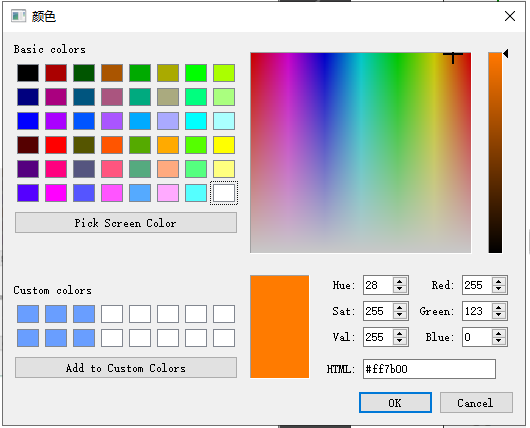
}

#### 警告对话框



QMessageBox::warning(this,"警告","警告提示");

#### 颜色对话框



//创建颜色对话框

参数1：默认颜色

QColor color = QColorDialog::getColor(QColor(255,123,0),this,"颜色");

//通过返回值判断选择的颜色

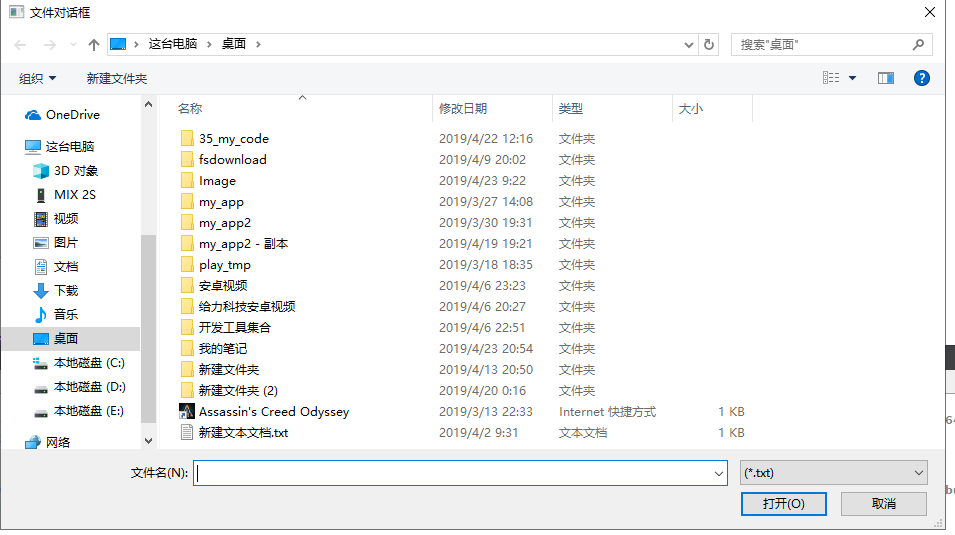
//取消情况下rgb都等于0

qDebug()<<"r="<<color.red()

<<",g="<<color.green()

<<" , b="<<color.blue();

#### 文件选择对话框



//创建文件对话框

//参数1：父窗口

//参数2：对话框标头

//参数3：默认路径

//参数4：筛选特定格式文件

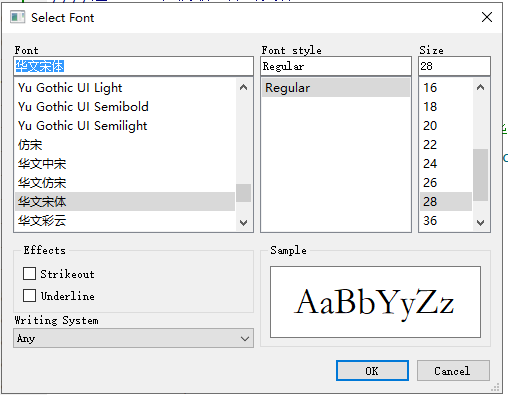
QString fileName= QFileDialog::getOpenFileName(this,"文件对话框","D:\\zm","(\*.txt)");

//通过返回值获取选择的文件

//"D:/zm/新建文本文档.txt"

qDebug() <<fileName;

#### 字体对话框



bool ok;

//创建字体对话框

QFont font= QFontDialog::getFont(&ok,QFont("华文彩云",36));

//通过返回值获取选择的字体

qDebug() << "字体："<< font.family().toUtf8().data()

<< "字号：" << font.pointSize()

<< "是否加粗： "<< font.bold()

<< "是否倾斜："<<font.italic();

//传入参数判断是否确定选择

if(ok)

{

qDebug() << "ok";

}

else

{

qDebug() << "取消选择";

}

### QradioButton->按钮组

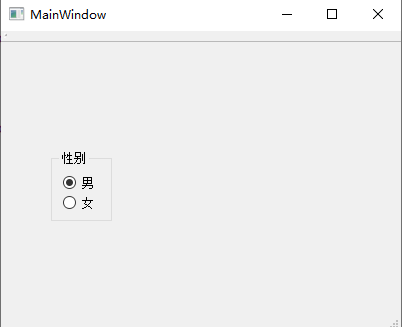
1、分组控件



2、单选按钮



3、效果图



4、代码

//默认选中那个按钮

ui->radioButton->setChecked(true);

//选中信号

connect(ui->radioButton,&QRadioButton::clicked,this,[](){

qDebug()<<"选中男的了";

});

connect(ui->radioButton\_2,&QRadioButton::clicked,this,[](){

qDebug()<<"选中女的了";

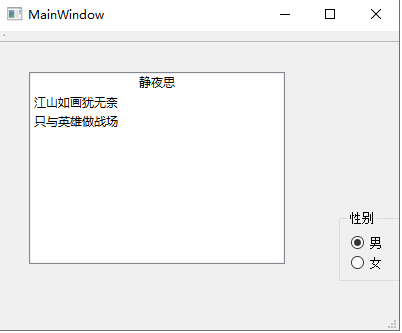
});

### ListWidget->列表控件

1、控件位置



2、效果图





3、代码

//列表头数据

QListWidgetItem \*item =new QListWidgetItem(QString("静夜思"));

//设置列表对齐模式（头？）

item->setTextAlignment(Qt::AlignHCenter);

//加入列表头数据

ui->listWidget->addItem(item);

//列表数据

QStringList list;

list<<"江山如画犹无奈";

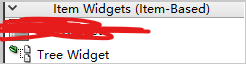
list<<"只与英雄做战场";

//放入列表数据

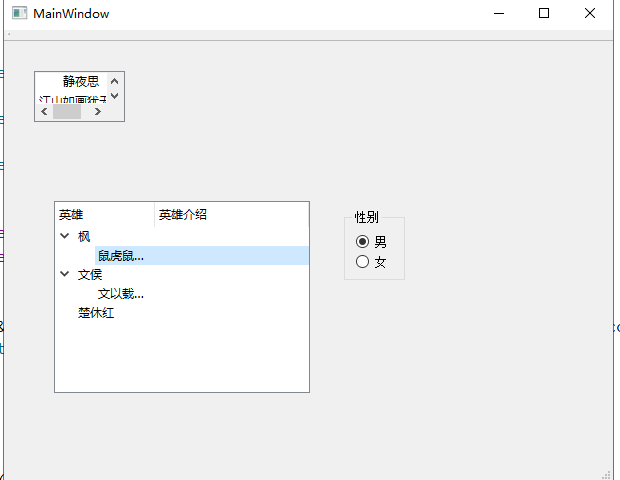
ui->listWidget->addItems(list);

### QtreeWidget->树控件

1、位置



2、效果图



3、代码

//水平头信息

QStringList list2;

list2<<"英雄"<<"英雄介绍";

//水平头信息加入树控件

ui->treeWidget->setHeaderLabels(list2);

//添加顶层项目

QTreeWidgetItem \* hero1 = new QTreeWidgetItem(QStringList()<<"枫");

//顶层项目放入树控件

ui->treeWidget->addTopLevelItem(hero1);

QTreeWidgetItem \* hero2 = new QTreeWidgetItem(QStringList()<<"文侯");

ui->treeWidget->addTopLevelItem(hero2);

QTreeWidgetItem \* hero3 = new QTreeWidgetItem(QStringList()<<"楚休红");

ui->treeWidget->addTopLevelItem(hero3);

//添加子控件

hero1->addChild(new QTreeWidgetItem(QStringList()<<"鼠虎鼠虎鼠虎鼠虎鼠虎鼠虎"));

hero2->addChild(new QTreeWidgetItem(QStringList()<<"文以载道文以载道文以载道"));

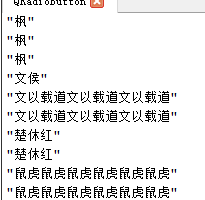
//单击选中事件

connect(ui->treeWidget,&QTreeWidget::itemClicked,this,[=](QTreeWidgetItem \*item,int column){

qDebug()<<item->text(column);

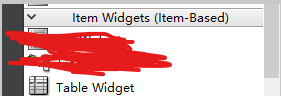
});

4、单击事件

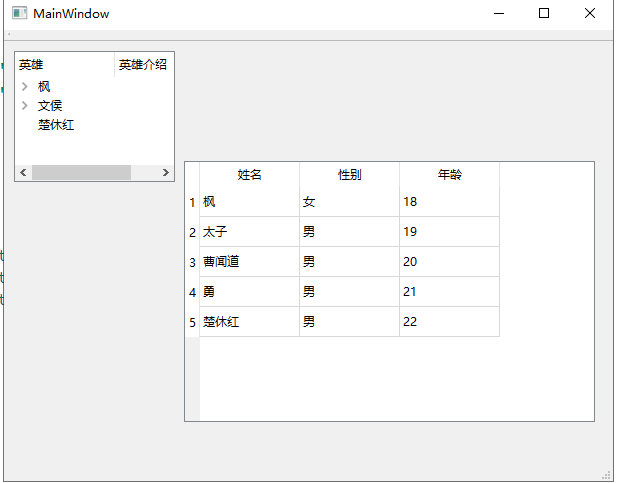


### TableWidget->列表控件

1、位置



2、效果图



3、代码

//设置表格的列数

ui->tableWidget->setColumnCount(3);

//设置表头信息

ui->tableWidget->setHorizontalHeaderLabels(QStringList()<<"姓名"<<"性别"<<"年龄");

//设置表格的行数

ui->tableWidget->setRowCount(5);

//准备表格信息

QStringList name;

QStringList sex;

name<<"枫"<<"太子"<<"曹闻道"<<"勇"<<"楚休红";

sex<<"女"<<"男"<<"男"<<"男"<<"男";

for (int i=0;i<5;++i)

{

//给表格添加数据

//参数1：插入第几行

//参数2：插入第几列

//参数3：需要插入的数据

ui->tableWidget->setItem(i,0,new QTableWidgetItem(name[i]));

ui->tableWidget->setItem(i,1,new QTableWidgetItem(sex[i]));

ui->tableWidget->setItem(i,2,new QTableWidgetItem(QString::number(i+18)));

}

//单击选中事件

connect(ui->tableWidget,&QTableWidget::itemClicked,this,[=](QTableWidgetItem \*item){

qDebug()<<item->text();

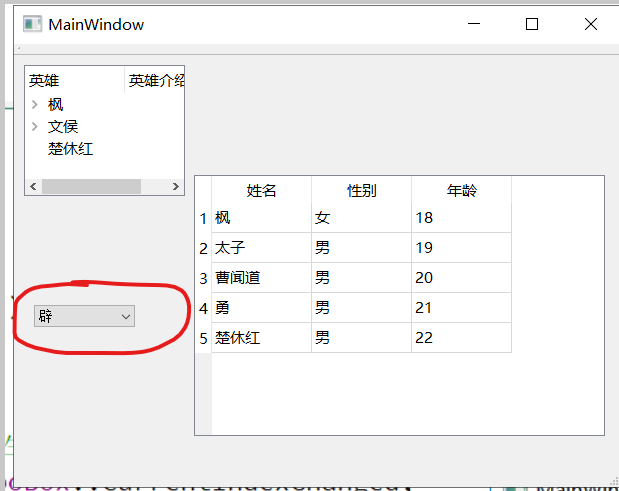
});

### QcomboBox->下拉框

1、位置



2、效果图



3、代码

//给下拉框放入内容

ui->comboBox->addItems(QStringList()<<"唯"<<"刀"<<"百"<<"辟");

//设置下拉框默认显示数据

ui->comboBox->setCurrentIndex(3);

//QComboBox::currentIndexChanged 发生重载 使用函数指针

void (QComboBox:: \*p)(int) = &QComboBox::currentIndexChanged;

//当前选中事件 获取下拉框选中数据

connect(ui->comboBox,p,this,[=](int index){

qDebug()<<ui->comboBox->itemText(index).toUtf8().data();

});

### QLabel->显示图片、gif动画

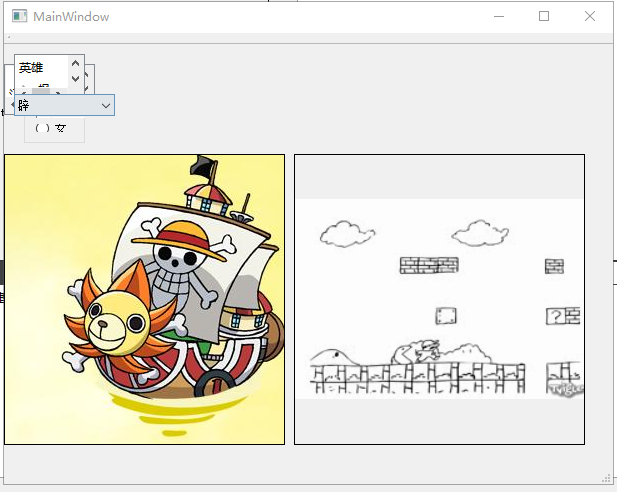
1、位置



2、设置边框



3、效果图



4、代码

//加载图片

QPixmap pix;

pix.load(":/Image/Sunny.jpg");

//图片放入label中

ui->label->setPixmap(pix);

//加载gif图片

QMovie \* movie = new QMovie(":/Image/mario.gif");

//设置gif图片尺寸

movie->setScaledSize(QSize(300,200));

//gif图片默认不播放 设置为播放状态

movie->start();

//gif图片放入labal中

ui->label\_2->setMovie(movie);

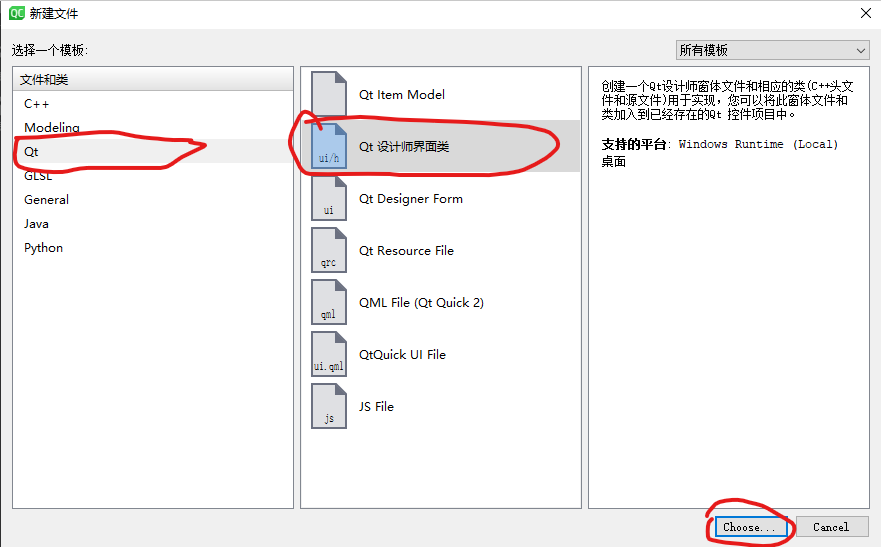
//gif图片可以停止播放

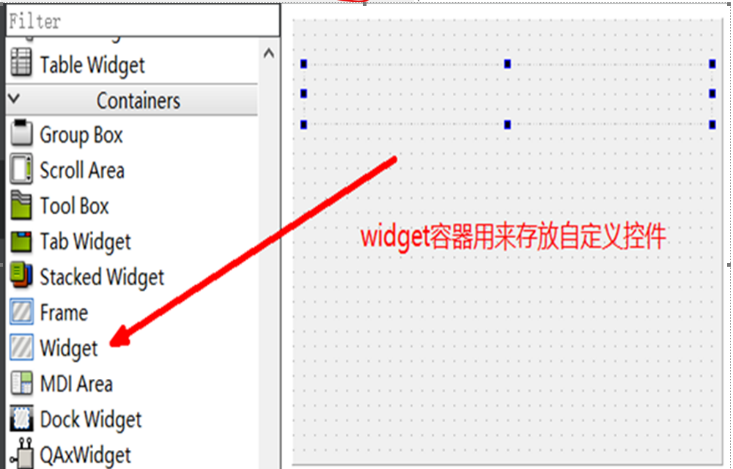
//movie->stop();

## 自定义控件

#### UI组合控件

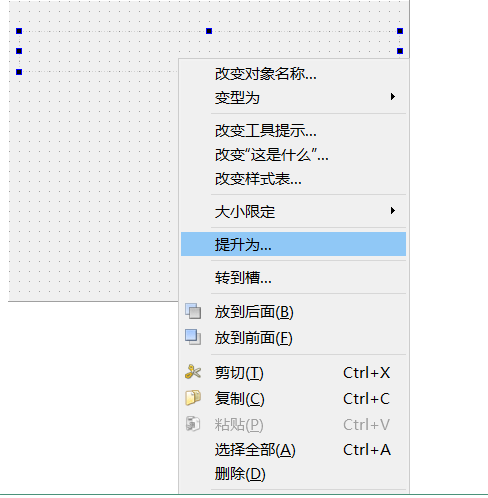
1、创建新文件



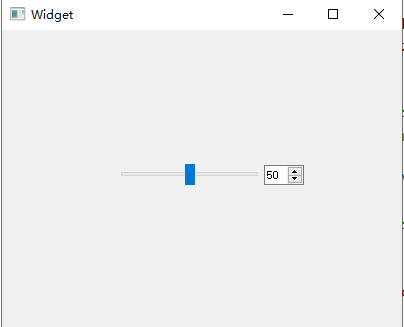


2、使用自定义控件

同一类型提升



3、效果图



4、代码

//QSpinBox::valueChanged 发生重载 使用指针

void(QSpinBox::\*p)(int) = &QSpinBox::valueChanged;

//关联控件两个控件 一个改变发出信号 另外一个调用对应槽函数跟着改变

connect(ui->spinBox,p,ui->horizontalSlider,&QSlider::setValue);

connect(ui->horizontalSlider,&QSlider::valueChanged,ui->spinBox,&QSpinBox::setValue);

//自定义控件 可以提供一些自己需要的方法

void Form::**setData**(int data)

{

ui->spinBox->setValue(data);

}

//自定义控件 可以提供一些自己需要的方法

int Form::**getData**()

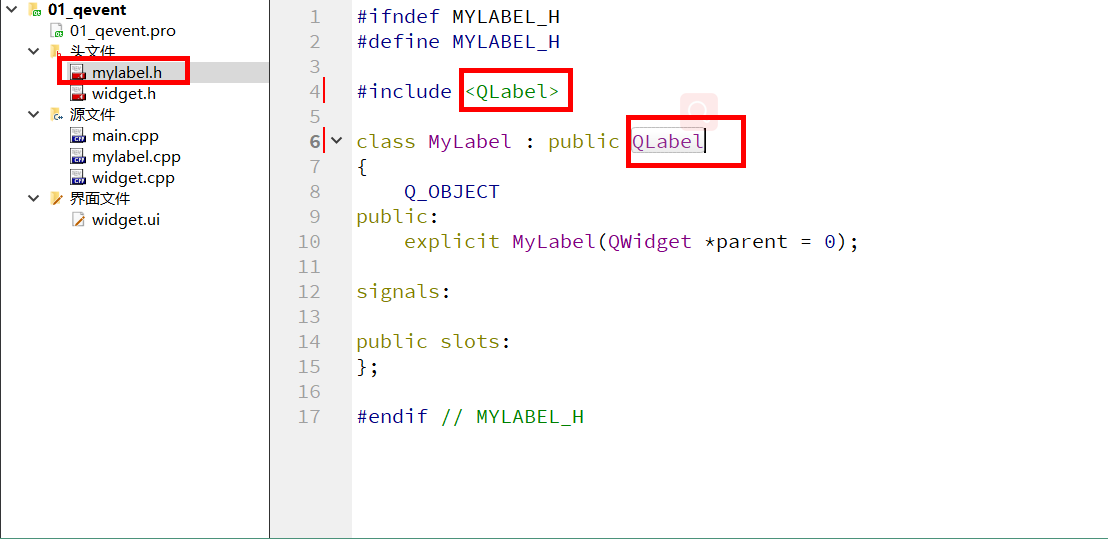
{

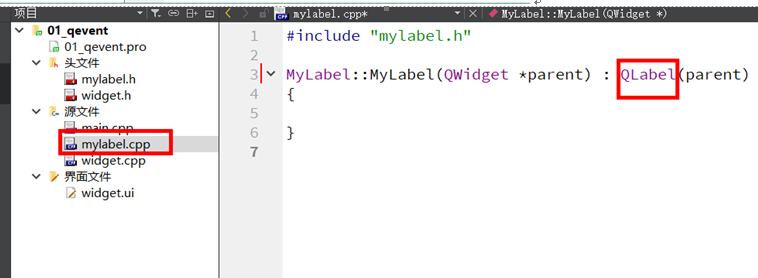
return ui->spinBox->value();

}

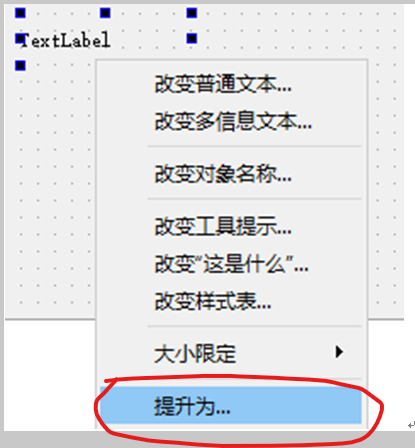
#### 重写控件(事件的理解)

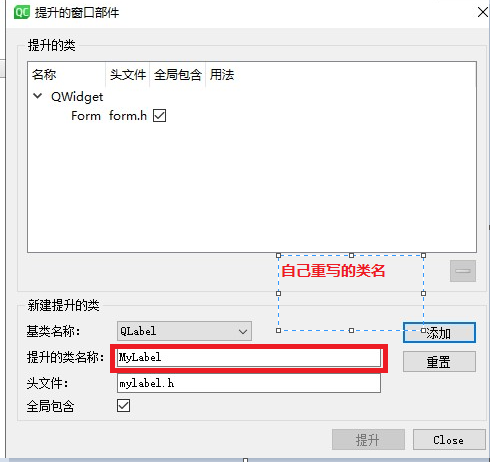
1、创建一个类 继承需要重写控件



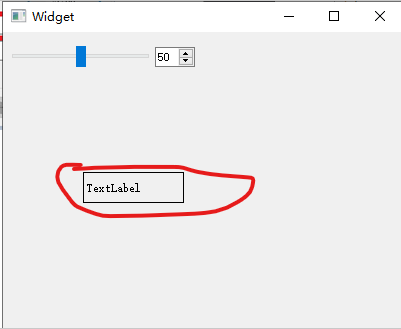


2、同一类型，提升





3、效果图



4、代码

[src\Qt\重写控件QLabel\mylabel.cpp](src/Qt/重写控件QLabel/mylabel.cpp)

[src\Qt\重写控件QLabel\mylabel.h](src/Qt/重写控件QLabel/mylabel.h)

5、事件就是 对象会默认执行的函数

## 定时器Qtimer

1、两种实现方法，通过重写定时器事件，或者使用定时器对象

2、定时器对象 的实现

//给当前对象创建定时器

QTimer \*timer =new QTimer(this);

//开始计时，并且给出每次发送信号的时间间隔

timer->start(500);//0.5s

//定时器发出的信号，捕获

connect(timer,&QTimer::timeout,this,[=](){

static int lrTime=0;

ui->label\_2->setText(QString::number( lrTime++));

});

//可以暂停计时器

//timer->stop();

//简单的延时执行效果，5000表示5秒

QTimer::singleShot(5000,NULL,[=](){

ui->label->setText("呵呵");

});

## 画图

1、画图分为画家、画图引擎（工具）、画图设备

2、代码

//在widget中画图需要重写事件

void Widget::***paintEvent***(QPaintEvent \*)

{

///////////////////////////////基本设置 ///////////////////////////

//创建画家 this表示画图设备 让画家在当前窗口绘图

QPainter painter(this);

//创建一支画笔，同时指定颜色

QPen pen(QColor(255,0,0));

//可以设置笔的风格 虚线

//pen.setStyle(Qt::DotLine);

//可以设置笔的粗细

pen.setWidth(5);

//让画家拿起画笔

painter.setPen(pen);

//画线 给出两个点的坐标 画一条直线

painter.drawLine(QPoint(0,0),QPoint(100,100));

//画圆 参数1：圆心位置 参数2：横向的半径 参数3:纵向的半径

painter.drawEllipse(QPoint(100,100),50,50);

//画矩形（长方形正方形这种）

//第一个数，横向的坐标

//第二个数，纵向的坐标

//第三个数，横向长度

//第四个数，纵向长度

painter.drawRect(QRect(100,0,200,50));

//画文字

painter.drawText(QRect(10,200,60,50),"好好学习，天天向上" );

////////////////////////////高级设置 ///////////////////////////

//添加抗锯齿效果，参数里还可以选择其他的高级抗锯齿效果

painter.setRenderHint(QPainter::Antialiasing);

//画圆

painter.drawEllipse( QPoint(200,100),50,50 );

//移动画家基础坐标

painter.translate(100,0);

//保存当前画家位置

//painter.save();

//去往保存画家位置,如果没有保存的画家位置，则位置不变

//painter.restore();

///////////////////////////画成品图片 ///////////////////////////

//加载图片

QPixmap pix = QPixmap(":/Image/Luffy.png");

//画图片

painter.drawPixmap(0,0,pix);

//利用QImage 修饰像素点

QPainter painter3(this);

//创建画图设备

QImage img2;

//画图设备加入图片

img2.load(":/Image/Luffy.png");

//修改图片中的像素点

for(int i = 50;i<100;i++)

{

for(int j = 50;j<100;j++)

{

//创建像素颜色

QRgb value = qRgb(255,0,0);

//根据坐标修改颜色

img2.setPixel(i,j,value);

}

}

//画图

painter.drawImage(0,0,img2);

}

///////////////////////////widget中也可以手动调用画图事件 /////////////////////////

//手动调用画图事件

this->update();

3、画图设备介绍

//利用QPixmap做绘图设备 会在不同平台上做了显示的优化

//创建画图设备

QPixmap pix(300,300);

//设置画图设备背景颜色

pix.fill(Qt::white);

//创建画家，传入画图设备

QPainter painter(&pix);

//传入画笔，和画笔颜色

painter.setPen(QPen(Qt::red));

//画圆

painter.drawEllipse(QPoint(150,150),100,100);

//画好图后 保存图片的位置

pix.save("D:\\zm\\pix.png");

//利用QImage做绘图设备 可以对像素级做访问

//创建画图设备

QImage img(300,300,QImage::Format\_RGB32);

//设置画图设备背景颜色

img.fill(Qt::white);

//创建画家，传入画图设备

QPainter painter2(&img);

//传入画笔，和画笔颜色

painter2.setPen(QPen(Qt::blue));

//画圆

painter2.drawEllipse(QPoint(150,150),100,100);

//画好图后 保存图片的位置

img.save("D:\\zm\\img.png");

//利用QPicture 做绘图设备 记录和重现画家的绘图指令

//创建画图设备

QPicture pic;

//创建画家

QPainter painter4;

//开始绘图

painter4.begin(&pic);

//笔的颜色

painter4.setPen(QPen(Qt::green));

//画图

painter4.drawEllipse(QPoint(150,150),100,100);

//结束绘图

painter4.end();

//保存画图指令的文件

pic.save("D:\\zm\\pic.lr"); //后缀名没有要求

//重现QPicture绘图指令

QPicture pic;

pic.load("D:\\zm\\pic.lr");

QPainter painter5(this);

//创建一支画笔，同时指定颜色

QPen pen(QColor(255,0,0));

//让画家拿起画笔

painter5.setPen(pen);

painter5.drawPicture(0,0,pic);

## 文件读写

//创建文件操作对象

QFile file("D://zm//35\_my\_code//Qt//untitled13//长恨歌\_gbk.txt");

//打开方式和判断文件是否打开成功

if (!file.*open*(QIODevice::ReadOnly | QIODevice::Text))

{

qDebug()<<"文件打开失败";

return;

}

//读取数据

QByteArray line;

while (!file.*atEnd*())

line += file.readLine();

//使用gbk格式显示

QTextCodec \*codec = QTextCodec::codecForName("gbk");

//显示数据到textEdit控件

ui->textEdit->setText(codec->toUnicode(line));

//默认utf-8显示

//ui->textEdit->setText(line);

//关闭文件操作对象

file.*close*();

//查看文件信息

QString qs("D://zm//35\_my\_code//Qt//untitled13//长恨歌\_utf-8.txt");

QFileInfo info(qs);

qDebug() << "后缀名："<<info.suffix() << "文件名："<<info.fileName()

<< "文件路径："<<info.filePath()<<"文件大小："<<info.size();

//这里需要添加头文件QDateTime

qDebug() <<"创建日期："<<info.created().toString("yyyy-MM-dd hh:mm:ss")

<<"最后修改日期："<<info.lastModified().toString("yyyy/MM/dd hh:mm:ss");

# Linux

## 根目录下的常见目录

/bin: 存放二进制可执行文件(系统命令)

/dev: 存放 硬件设备所对应的文件（鼠标之类）

/etc: 存放 系统配置文件

/home: 家目录

/peoc: 存放系统核心，内存中的数据

/tmp: 存放用户程序工作期间产生的临时文件

/var: 存放系统工作期间产生的 经常变化的文件。（日志、数据库文件）

/root: root用户 家目录

/lib: 存放系统库

usr: 存放用户程序 相关 资源目录

/usr/lib：用户库

/usr/include/ 头文件

## 常用命令

### 查看端口占用

netstat -tnlp | grep :22

### 防火墙

查看防火墙开放的端口

firewall-cmd --list-ports

开放端口

firewall-cmd --zone=public --add-port=8084/tcp --permanent

重启防火墙

firewall-cmd –reload

停止防火墙

systemctl stop firewalld

启动防火墙

systemctl start firewalld

### which 查询命令保存在哪

### df 查看磁盘剩余空间

参数：-h人性化显示

### du 查看当前目录使用空间

参数：-h 人性化显示

注意：可以指定目录

### htop任务管理器

### man手册的使用

向下翻页：空格

上下翻页：PGUP、PGON

上下翻动：上下方向键

回到头部：Home

回到尾部：END

向下搜索：/

向上搜索：？

搜索上一个、下一个：n、N

退出：q

### ls

参数：

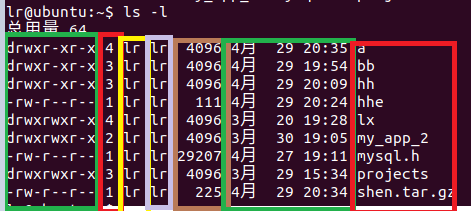
-l: 显示详细信息

-a: 显示所有文件，包括隐藏

-h: 显示人类可读信息，带有 单位 信息

-i: 显示文件的inode节点

命令 ls -l 解读



1、文件类型和权限

a.文件的类型，第一个字符表示

(1). 普通文件 - 占用磁盘存储。

(2). 目录文件 d 占用磁盘存储。

(3). 软链接文件 l 占用磁盘存储。

(4). 块设备文件 b 伪文件，不占用磁盘存储。

(5). 字符设备文件 c 伪文件，不占用磁盘存储。

(6). 管道文件 p 伪文件，不占用磁盘存储。

(7). 套接字 s 伪文件，不占用磁盘存储。

(8). 未知文件

b.文件的权限，后面的字符表示

(1).r表示可读、 w表示可写、 x表示可执行

(2).前三个所有者权限、中三同组权限、后三其他人权限

2、硬连接数

3、文件所有者

4、文件所有组

5、文件大小（单位字节）（4k 的整数倍）

6、时间 文件首次创建的、最后一次修改的时间

7、文件名

### shell 命令使用技巧

history：列出过往执行过的命令

使用 ！+ 命令编号，重新执行该命令

Ctrl+a： 将光标移至最前端。

Ctrl+e： 将光标移至最末端。

Ctrl+u： 清空已输入的命令。

### 创建删除文件、目录

创建文件语法：touch 文件名

创建目录语法：mkdir 目录名

参数：-p 创建多层目录

删除文件、目录语法：rm 文件、目录名

参数：-r 递归删除

-I 询问、交互式删除

### 软、硬链接

硬连接语法：ln 源文件 硬链接文件

硬连接实现原理：源文件和硬链接文件之间有相同的inode节点 指向同一块扇区。

注意：不能给目录创建硬连接。目录的硬连接来自”.”和”..”

软连接语法：ln -s 源文件 软链接文件

注意：

1、软连接可以使用相对路径和绝对路径

2、可以给目录创建软连接

### cp

拷贝语法：cp 源文件 新文件

参数：-r 递归拷贝

### mv

移动语法：mv 源文件 新位置、新文件名

### which命令所在的位置

查找命令语法：which 需要查询的命令

### | 管道

将前一个命令的输出，作为后一个命令的输入。

语法：ps aux | grep xxx 从 系统所有的进程中，过滤出 xxx 进程。

### >、>>重定向

重定向，将前一个命令的输出，重定向到其他地方

>: 文件不存在创建,文件存在覆盖

>>: 文件不存在创建,文件存在追加

### cat、tac、more、head、tail

查看文件语法：cat 文件名

逆序查看文件语法：tac 文件名

-n：每行加编号

查看大文件语法：less 文件名

显示前10行语法：head head -10

显示后10行语法：tail tail -10

### 压缩、解压缩

压缩语法： tar -zcvf 压缩包名.tar.gz 压缩源材料

tar -jcvf 压缩包名.tar.bz2 压缩源材料

解压缩语法：tar -zxvf 压缩包名.tar.gz -C 解压到什么路径

tar -jxvf 压缩包名.tar.bz2

参数含义：

z: 使用gzip格式压缩

j: 使用bzip2 格式压缩

c: 创建压缩包（多个文件、只生成一个压缩包）

x：解压缩

v: 显示过程

f: 指定压缩包的名称。参数通常放在后面

### terr树状态目录显示

语法：terr录名

### ps 命令

语法：ps -aux

参数a：显示终端上的所有进程，包括其他用户的进程

参数u：显示所有进程的详细状态

参数x：显示没有控制终端的进程

使用管道过滤需要的进程：

ps aux | grep xxx

top命令动态进程管理(了解)

### kill发送信号命令

语法：kill -9 进程PID

注意：

1、9号信号的名称SIGKILL

2、查看所有信号：kill -l

### 用户管理

添加用户

语法：adduser 用户名

修改用户密码：

语法：passwd 用户名

给用户管理员权限

chmod -v u+w /etc/sudoers

vim /etc/sudoers

找到 root ALL=(ALL) ALL

下面添加 用户名 ALL=(ALL) ALL

chmod -v u-w /etc/sudoers

### 权限管理

修改文件拥有者

语法：chown 用户名 需要修改的文件、目录名

修改文件所有组

语法：chgrp 组名 需要修改文件/目录名

修改文件权限

语法：chmod 777 文件名

7表示：读(r) 4

写(w) 2

执行(x) 1

第一个数字表示：所有者权限

第二个数字表示：所有组权限

第三个数字表示：其他人权限

权限对于目录和文件的不同作用

r对于文件：可以查看文件内容。cat、less、head

w对于文件：可以修改、删除文件内容。vim

x对于文件：可以被运行。./aout

r对于目录：可以被查看。ls、tree

w对于目录：可以被曾、删、改。mkdir、rm、mv、touch

x对于目录：可以被进入。cd

### find按文件属性查找

按名称找的语法：find 需要查找的目录 -name “查找内容”

通配符：

1、\* 表示零到多个任意字符

2、? 表示一个任意字符

按类型找的语法：find 需要查找的目录 -type “查找的类型”

类型：

(1). 普通文件 f

(2). 目录文件 d

(3). 软链接文件 l

(4). 块设备文件 b

(5). 字符设备文件 c

(6). 管道文件 p

(7). 套接字 s

按文件大小找的语法：find 需要查找的目录 -size “+100M” -size “-1000M”

注意：

1、+100M 表示查找大于100M的文件

2、-1000M 表示查找小于1000M的文件

3、不写单位单位的情况下默认是字节，其他单位k(小写)、M、G

指定搜索层级深度：-maxdepth 1，放在目录后面，数字表示层级

语法：find 需要查找的目录 -maxdepth 1 -name “查找内容”

对查询结果执行命令：-exec，放在结尾

语法：find 需要查找的目录 -name “查找内容” -exec 命令 {} \;

注意：

1、按类型、按文件、按大小、指定搜索深度、对查询结果执行命令，可以同时使用

了解，使用管道对查询结果执行命令

find ./ -maxdepth 1 -type f | xargs ls -l

【问题】：当有一个 含有空格的文件存在时，上述实现报错

find ./ -maxdepth 1 -type f -print0 | xargs -0 ls -l 解决 上述问题。

了解，可扩展：-ctime、-atime、-mtime -cmin -amin -mmin

### grep按文件内容查找

语法：grep -rn “查找的内容” 查找的目录

参数：

n：可以显示查找内容的所在的行号

r：递归查找

注意：

1、find可以和grep结合使用 示例：find ./ -name "\*.c" -exec grep "main" -n {} \;

### ssh文件上传、下载

上传语法：

scp -r 本地文件 目标主机用户名@目标主机IP:绝对存储路径（不能使用相对）

下载语法：

scp -r 目标主机用户名@目标主机IP:待下载文件的绝对存储路径（不能使用相对） 本地存储位置（相对、绝对）

注意：需要输入密码

## vim简单使用

### 命令模式

复制 : yy 复制多行 行数加yy

粘贴：小p粘贴到光标前面，大P粘贴到光标后面

剪切：dd 剪切多行 行数加dd

多行选中：V 加 方向键

撤销：u

反撤销：Ctrl +r

搜索：/向下搜索 ?向上搜索 小写n搜索下一个

分屏显示：vsp 需要打开的文件名

还可以使用vs上下分屏显示

Ctrl+w切换

wqall 一次退出所有分屏

自动对齐：gg=G gg是跳转行首可以省略

### 末行模式

q：退出

q！：强制退出不保存

w：保存

wq：保存退出

1：数字几就跳转第几行

！：执行shell命令

### 基础配置

[src\linux常用命令\vim\.vimrc](file:///C:\Users\lr\AppData\Roaming\Microsoft\Word\src\linux常用命令\vim\.vimrc)

## gcc

### gcc编译的4个步骤

1、预处理：-E

* + - * 生成 .i 文件
* 头文件展开
* 宏替换
* 注释删除

2、C转汇编：-S

* 生成 .s 汇编代码

3、汇编：-c

* 生成 .o 二进制文件

4、链接：

* 链接所有 .o 文件
* 生成可执行程序

### gcc常用参数

| **gcc编译选项** | **选项的意义** |
| --- | --- |
| -E | 预处理指定的源文件，不进行编译 |
| -S | 编译指定的源文件，但是不进行汇编 |
| -c | 编译、汇编指定的源文件，但是不进行链接 |
| -o [file1] [file2] / [file2] -o [file1] | 将文件 file2 编译成可执行文件 file1 |
| -I directory | 指定 include 包含文件的搜索目录 |
| -g | 在编译的时候，生成调试信息，该程序可以被调试器调试 |
| -D | 在程序编译的时候，指定一个宏 |
| -w | 不生成任何警告信息 |
| -Wall | 生成所有警告信息 |
| -On | n的取值范围：0~3。编译器的优化选项的4个级别，-O0表示没有优化，-O1为缺省值，-O3优化级别最高 |
| -l | 在程序编译的时候，指定使用的库 |
| -L | 指定编译的时候，搜索的库的路径。 |
| -fPIC/fpic | 生成与位置无关的代码 |
| -shared | 生成共享目标文件。通常用在建立共享库时 |
| -std | 指定C方言，如:-std=c99，gcc默认的方言是GNU C |

## 库的制作和使用

### 静态库

* 静态库的特点
* 编译阶段加入程序
* 优点
* 静态库被打包到应用程序中加载速度快
* 发布程序无需提供静态库，移植方便
* 缺点
* 消耗系统资源，浪费内存
* 更新、部署、发布麻烦
* 静态库的制作：
* 通过gcc -c 编译源文件获得 .o 二进制文件
* 使用 ar 把 .o 文件打包，生成 .a 库文件
* 示例：ar rcs libcalc.a add.o div.o mult.o sub.o
* ar 命令
* rcs 参数
* libcalc.a
* lib前缀
* .a 后缀
* calc 静态库名称
* 静态库的使用
* 指定头文件位置
* -I 参数 （大写i）
* 指定静态库位置
* -L 参数
* 指定静态库的名称
* -l 参数 （小写L）
* 不要静态库文件的前后缀，只要名称

### 动态库/共享库

* 动态库的特点
* 程序运行时加入程序，多个程序共享一份动态库文件
* 优点
* 可实现进程间资源共享
* 程序升级简单
* 程序员可以控制何时加载动态库
* 缺点
* 加载速度比静态库慢
* 发布程序需要提供依赖的动态库
* 动态库的制作
* 使用gcc 得到 .o 文件
* 示例：gcc -c sub.c add.c div.c mult.c -fpic
* -fpic 动态库需要加-fpic参数，得到于位置无关的代码
* 使用gcc 得到 .so动态库文件
* 示例：gcc -shared add.o sub.o div.o mult.o -o libcalc.so
* lib前缀
* .so 后缀
* calc 动态库名称
* 动态库的使用
* 指定头文件位置
* -I （大写i）
* 指定动态库位置
* -L
* 指定动态库的名称
* -l （小写L）
* 不要动态库文件的前后缀，只要名称
* 找不到动态库文件解决
* ldd 程序名 ，检测程序是否链接到动态库
* 解决方法1：动态库的绝对路径添加到 LD\_LIBRARY\_PATH 中
* 临时生效（当前终端有效）
* 执行该命令：export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:动态库的绝对路径和名称
* 库文件名称不要前后缀
* 用户级配置
* 把该代码添加到 ~ 目录下的 .bashrc 配置文件尾部：export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:动态库的绝对路径和名称
* 库文件名称不要前后缀
* 系统级配置
* 把该代码添加到 /etc/ 目录下的 profile配置文件尾部：export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:动态库的绝对路径
* 库文件名称不要前后缀
* 解决方法2：在 /etc/ld.so.cache 添加动态库路径
* 无法直接修改 /etc/ld.so.cach 需要间接修改
* 在 /etc/ld.so.conf 配置文件中添加库所在的路径，不要名称
* 执行一个命令, 进行更新: sudo ldconfig
* 解决方法3：库文件放入 /use/lib/ 或者 /lib/ 目录中

后期添加：

使用VS时

头文件放入：/usr/local/include/

库文件放入：/usr/lib

## 简单的Makefile编写

### makefile中的规则语法：

目标 … : 依赖 …

命令(shell命令)

* 目标：最终要生成的文件，伪目标不算（.PHINY:）
* …：可以有一个或者多个生成文件或者依赖
* 依赖：生成目标的需要的材料
* 命令：通过命令对依赖操作生成目标，命令前面是tab缩进
* 一个makefile中可以有多条规则，后面的规则一般为前面的规则服务，如果前面的规则找不到依赖的文件，就会执行后面的规则，如果依赖文件足够就不会执行后面的依赖

### makefile中的变量

* 自动变量
* $@: 规则中的目标
* $<: 规则中的第一个依赖
* $^: 规则中的所有依赖

### makefile中的函数

* $(wildcard ./\*.c) 获取文件名，多个目录用空格分开
* $(patsubst %.c,%.o,$(src)) 字符串替换
* 参数：查找参数3字符串中的单词，(单词以“空格”、“tab”、”回车”、“换行”分割),是否符合参数1的单词格式，如果符合使用参数2的单词替换，结果使用变量接收，参数三的数据不会改变
* %号是通配符，表示任意长度的字符串，如果使用两个百分号，那第二个百分号中的内容来源就是第一个百分号中的内容

### 代码示例

[src\linux常用命令\makefile\makefile](file:///C:\Users\lr\AppData\Roaming\Microsoft\Word\src\linux常用命令\makefile\makefile)

### 课堂笔记：动态库、静态库、makefile的编写

[src\linux常用命令\makefile\3-gcc、动态库、静态库、makefile.md](file:///C:\Users\lr\AppData\Roaming\Microsoft\Word\src\linux常用命令\makefile\3-gcc、动态库、静态库、makefile.md)

## ssh别名配置、免密码登录

别名配置

找到 .ssh 目录，在config目录下添加内容

Host 别名

HostName 目标ip

User 登录用户名

IdentitiesOnly yes

远程连接命令： ssh 别名

免密码登录

生成秘钥（默认名称id\_rea）：ssh-keygen -t rsa -C "825422719@qq.com"

找到 .ssh 目录的id\_rsa.pub 拷贝到服务器上

Id\_res 中的秘钥添加到 服务器 家 目录下的 .ssh 目录下的 authorized\_keys 文件中

# Linux系统编程

## 文件IO

### 说明

linux文件IO，没有缓冲区

### 打开或者新建文件

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int open(const char \*pathname, int flags);

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

参数：

pathname:需要打开或者创建的文件

flags：打开方式

必选项（三个选项互斥）：

O\_RDONLY:只读

O\_WRONLY: 只写

O\_RDWR: 读写

可选项:

O\_CREAT: 文件不存在，新建文件，文件存在相当于没写这个选项

O\_APPEND: 在文件末尾追加

mode：

1、若是新建文件需要通过这个参数指定权限，

2、填八进制 如 0777，

3、创建出来的权限需要减去 umask

4、可以通过umask命令查看默认权限 ，open函数生成文件的权限: mode & ~umask

返回值：

成功返回：文件描述符

失败返回：-1，设置errno

注意：

C语言默认打开的三个文件描述符

STDIN\_FILENO：输入  
STDOUT\_FILENO：输出  
STDERR\_FILENO：错误输出

### 关闭文件文件描述符

#include <unistd.h>

int close(int fd);

参数:

fd: open需要关闭的文件描述符

返回值:

成功: 返回0

失败: 返回-1

### 读文件

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t count);

参数：

fd：需要读取的文件描述符

buf：保存读取数据的地方

count：需要读取的字节数

返回值：

成功返回：读取到的字节数字

返回0表示：读到了文件末尾,或者写端关闭

失败返回：-1，设置errno

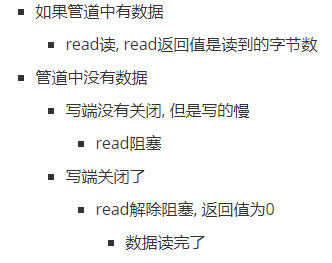
errno = 11，非阻塞情况下读取到缓冲区中没有数据的文件描述符

errno =104，发送数据太快会出现

注意：

1、函数阻塞与否由文件描述符决定

2、读取pipe和mkfifo创建的文件描述符，函数会阻塞



### 写文件

创建一个

#include <unistd.h>

ssize\_t write(int fd, const void \*buf, size\_t count);

参数：

fd：需要写的文件描述符

buf：需要写的数据

count：需要写的字节数，如果写入数据不足，末尾自动补 \0

返回值：

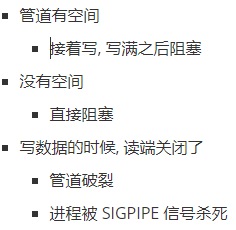
成功返回：写入的字节数

失败返回：-1，设置errno

注意：

1、函数阻塞与否由文件描述符决定

2、对pipe和mkfifo创建的文件描述符进行写操作时，函数会阻塞



### 文件指针操作

#include <sys/types.h>

#include <unistd.h>

off\_t lseek(int fd, off\_t offset, int whence);

参数：

fd：文件描述符

offset：偏移量

whence：指针的位置

SEEK\_SET--文件头

SEEK\_CUR--当前位置

SEEK\_END--末尾

返回值：返回文件指针的位置

示例：

// 1. 移动文件指针到文件头

lseek(fd, 0, SEEK\_SET);

// 2. 获取当前文件指针的位置

lseek(fd, 0, SEEK\_CUR);

// 3. 获取文件长度

lseek(fd, 0, SEEK\_END);

// 4. 拓展文件长度, 拓展100字节

lseek(fd, 100, SEEK\_END);

// 要进行一次写操作

write(fd, " ", 1);

### 操作文件描述符

#include <unistd.h>

int dup(int oldfd);`

功能：复制一个文件描述符

参数：

oldfd：需要被复制的文件描述符

返回值：

成功返回：一个新的文件描述符

失败返回：-1，设置errno

注意：

1、此时有两个文件描述符记得都要关闭

#include <unistd.h>

int dup2(int oldfd, int newfd);

功能：重定向一个文件描述符

参数：

oldfd：不需要不改变指向文件描述符

newfd：需要改变指向的文件描述符

返回值：

成功返回： newfd

失败返回：-1，设置errno

注意：

1、两个文件描述符都需要关闭

2、此时两个文件描述符指向同一个文件

#include <unistd.h>

#include <fcntl.h>

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

功能：该函数可以对文件描述符作一些设置，如修改读写操作，设置文件描述符带阻塞等等

代码示例：

[src\linux系统编程\文件操作\ddq.c](src/linux系统编程/文件操作/ddq.c)

## linux常用系统函数

### 目录操作函数

#### 文件重命名

#include <stdio.h>

int rename(const char \*oldpath, const char \*newpath);

参数：

oldpath：旧的文件名

newpath：新的文件名

返回值：

成功返回：0

失败返回：-1，设置errno

#### 修改进程的工作目录

#include <unistd.h>

int chdir(const char \*path);

参数：

path：切换到那个工作目录

返回值：

成功返回：0

失败返回：-1，设置errno

#### 获取进程的当前工作目录

#include <unistd.h>

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

参数：

buf：传入参数保存进程当前工作目录

size：buf大小

返回值：

成功返回：参数buf的首地址

失败返回：NULL，设置errno

#### 创建目录

#include <sys/stat.h>

#include <sys/types.h>

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

参数：

pathname：目录名称

mode：目录的权限，8进制，mod & ~umask

返回值：

成功返回：0

失败返回：-1

#### 删除目录

#include <unistd.h>

int rmdir(const char \*pathname);

参数：

pathname：目录名

返回值：

成功返回：0

失败返回：-1

注意：

1、相当于 rm -f 只能删除空目录

#### 打开一个目录

#include <sys/types.h>

#include <dirent.h>

DIR \*opendir(const char \*name);

参数:

- name: 要打开的目录名

返回值:

成功返回：DIR

失败返回：NULL

注意：

1、//关闭目录

int closedir(DIR \*dirp);

#### 读取目录中的数据

#include <dirent.h>

struct dirent\* readdir(DIR \*dirp);

参数：

dirp：打开的目录

返回值：

成功返回：dirent结构体,里面保存了一个文件的属性

失败返回：NULL，不设置errno

读取到目录尾返回：NULL，设置errno

注意:

1、目前想不到这个函数可以能会出现失败的情况，所以把NULL当做读到文件尾即可，传入错误参数只会段错误，不会函数调用失败

struct dirent

{

ino\_t d\_ino; // 此目录进入点的inode

ff\_t d\_off; // 目录文件开头至此目录进入点的位移

signed short int d\_reclen; // d\_name 的长度, 不包含NULL 字符

unsigned char d\_type; // d\_name 所指的文件类型

har d\_name[256]; // 文件名

};

d\_type

DT\_BLK - 块设备

DT\_CHR - 字符设备

DT\_DIR - 目录

DT\_LNK - 软连接

DT\_FIFO - 管道

DT\_REG - 普通文件

DT\_SOCK - 套接字

DT\_UNKNOWN - 未知

struct dirent \*readdir(DIR \*dirp);

- 返回一个结构体, 这个对应一个文件

代码案例：

1、小案例

[src\linux系统编程\目录操作\main.c](src/linux系统编程/目录操作/main.c)：

2、演示

[src\linux系统编程\目录操作\demo.c](src/linux系统编程/目录操作/demo.c)

### 文件属性函数

#### 判断文件权限、存在

#include <unistd.h>

函数功能：判断是否有操作某个文件的权限，或者判断文件是否存在

int access(const char \*pathname, int mode);

参数：

- pathname：文件名

- mode：

R\_OK：判断是否有读权限

W\_OK：判断是否有写权限

X\_OK：判断是否有执行权限

F\_OK：判断文件是否存在

返回值：

有操作权限、文件存在返回：0

没有有操作权限、文件存在返回：-1，设置erron

#### 修改文件权限

#include <sys/stat.h>

int chmod(const char \*filename, int mode);

参数：

- filename：文件名

- mode：八进制数设置文件权限 如：0777

返回值：

成功返回：0

失败返回：-1，设置errno

#### 修改文件所有者

#include <unistd.h>

int chown(const char \*path, uid\_t owner, gid\_t group);

参数：

path：要修改的文件名

owner：用户ID

group：组ID

返回值：

成功返回：0

失败返回：-1，设置errno

#### 修改文件大小

#include <unistd.h>

#include <sys/types.h>

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

参数：

path：要操作的文件

length：最终的文件大小

返回值：

成功返回：0

失败返回：-1，谁知errno

### 获取文件属性stat()

功能：获取一个文件的详细属性

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

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

参数：

pathname：文件名

buf: 传出参数, 保存文件的属性信息

返回值:

成功: 返回0

失败: 返回-1, 并设置errno

注意：

stat函数: 获取的是连接文件指向的文件的属性

lstat函数: 获取的是连接文件本身的属性

struct stat {

dev\_t st\_dev; //文件的设备编号

ino\_t st\_ino; //节点

mode\_t st\_mode; //文件的类型和存取的权限

nlink\_t st\_nlink; //连到该文件的硬连接数目，刚建立的文件值为1

uid\_t st\_uid; //用户ID

gid\_t st\_gid; //组ID

dev\_t st\_rdev; //(设备类型)若此文件为设备文件，则为其设备编号

off\_t st\_size; //文件字节数(文件大小)

blksize\_t st\_blksize; //块大小(文件系统的I/O 缓冲区大小)

blkcnt\_t st\_blocks; //块数

time\_t st\_atime; //最后一次访问时间

time\_t st\_mtime; //最后一次修改时间

time\_t st\_ctime; //最后一次改变时间(指属性)

};

关于变量 st\_mode:

- st\_mode -- 16位整数

0-2 bit -- 其他人权限

- S\_IROTH 00004 读权限

- S\_IWOTH 00002 写权限

- S\_IXOTH 00001 执行权限

- S\_IRWXO 00007 掩码, 过滤 st\_mode中除其他人权限以外的信息

3-5 bit -- 所属组权限

- S\_IRGRP 00040 读权限

- S\_IWGRP 00020 写权限

- S\_IXGRP 00010 执行权限

- S\_IRWXG 00070 掩码, 过滤 st\_mode中除所属组权限以外的信息

6-8 bit -- 文件所有者权限

- S\_IRUSR 00400 读权限

- S\_IWUSR 00200 写权限

- S\_IXUSR 00100 执行权限

- S\_IRWXU 00700 掩码, 过滤 st\_mode中除文件所有者权限以外的信息

12-15 bit -- 文件类型

- S\_IFSOCK 0140000 套接字

- S\_IFLNK 0120000 符号链接（软链接）

- S\_IFREG 0100000 普通文件

- S\_IFBLK 0060000 块设备

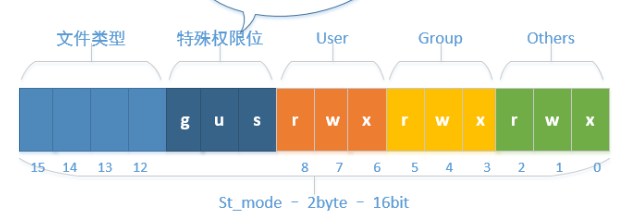
- S\_IFDIR 0040000 目录

- S\_IFCHR 0020000 字符设备

- S\_IFIFO 0010000 管道

- S\_IFMT 0170000 掩码,过滤 st\_mode中除文件类型以外的信息

(st\_mode & S\_IFMT) == S\_IFREG



### exec

功能：使用一个进程拉起另外一个进程，被拉起进程的用户区 替换 拉起进程的用户区，内核区不变，所以不会创建出新的进程

#include<unistd.h>

int execlp(const char \*file,const char \*arg)

参数：

file:命令、程序名称，会在path里面找，一般用来执行命令

arg:参数可用多个，用NULL结束

#include<unistd.h>

int execl (const char \*file,const char \*arg)

参数：

file:命令、程序名称，不会在path里面找，所以一般用于执行程序

arg:参数可用多个，用NULL结束

返回值：

成功：无返回

失败，返回-1，设置errno

## 进程

### 进程说明：

1、每一个进程都拥有独立的虚拟地址空间，进程是linux系统资源的最小分配单位。

2、fork后，只有少量内核中的少量数据不会被拷贝，目前已知PID。

3、孤儿进程，父进程在子进程之前推出了程序，子进程会认PID为1的进程做父进程，由PID为1的进程负责释放

4、僵尸进程，子进程结束后父进程不负责回收，同时父进程还在运行中，子进程就会变成僵尸进程，僵尸进程需要杀死父进程才会死亡

### 创建进程：

#include<unistd.h>

pid\_t fork() (void)

成功返回：

父进程返回：父进程pid

子进程返回：0

失败返回：-1，设置errno

### 获取当前进程的PID和获取父进程的PID

#include <sys/types.h>

#include <unistd.h>

获取当前进程的PID:

pid\_t getpid(void);

获取父进程的PID:

pid\_t getppid(void);

### 进程回收函数

功能：回收一个结束的子进程

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait(int \*wstatus)

参数：wstatus: 传一个int数据进去就行

WIFEXITED(wstatus)值为非0：进程正常结束

WEXITSTATUS(wstatus):

1.子进程的返回值，如果没有返回值，结果可能是随机数

2.值是无符号1字节数据，如果返回值为负数，接收可能不正确

a. -1 == 255，-2==254 ，以此类推

WIFSIGNALED(wstatus): 值为为非0：进程异常终止

WTERMSIG(wstatus):

终止进程的信号编号(整型数据)

返回值:

成功返回：子进程PID

失败返回：-1

注意：

1、函数带阻塞

2、只回收一个进程

3、通过4个宏函数，获取子进程退出状态

可以设置非阻塞和回收指定进程的进程回收函数

pid\_t waitpid(pid\_t pid, int \*wstatus, int options)

参数：

pid:

pid= -1: 所有子进程都可以回收

pid> 0: 需要回收的子进程PID

pid= 0: 没写，不知道

wstatus：用法和上一个函数相同

options：传WNOHANG不阻塞，传0阻塞

返回值:

>0: 回收掉的子进程PID

-1: 无子进程

=0: 参数options为非阻塞，且子进程未结束

### 进程通信pipe (管道)

功能：匿名管道返回两个文件描述符，父子进程之间通信

#include <unistd.h>

int pipe(int pipefd[2]);

参数：

pipefd：传出参数

pipefd[0]:管道的读端

pipefd[1]:管道的写端

返回值：

成功返回：0

失败返回：-1，设置errno

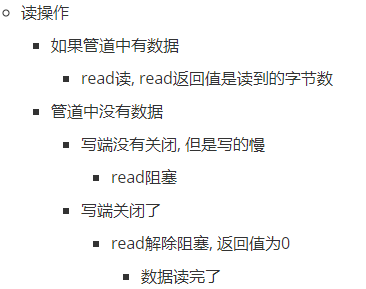
注意：

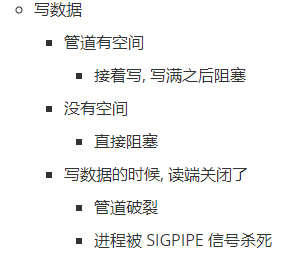
1、管道是单向的，建议读端关闭写端，写端关闭读端

2、管道的默认大小4k 查看ulimit -a

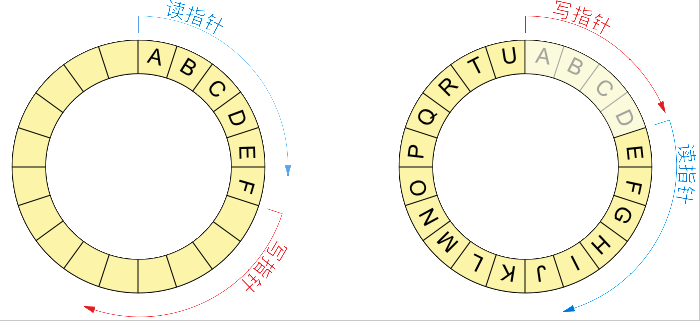
3、管道可以的会动态扩展一部分空间

4、管道文件描述符带阻塞





5、管道的结构，环形队列



### 进程通信mkfifo(管道)

功能：有名管道，创建一个管道文件，进行进程通信

#include <sys/types.h>

#include <sys/stat.h>

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

参数：

pathname：创建的文件名

mode：文件的权限

返回值：

成功返回：0

失败返回：-1，设置ernno

注意：

1、除了管道的大小其他的应该都后上面一样

### 进程通信mmap（内存映射区）

功能：把一个硬盘中的普通文件映射到 程序的内存中 相当于两个进程拥有同一块内存

#include <sys/mman.h>

void \*mmap(void \*addr, size\_t length, int prot, int flags,int fd, off\_t offset);

参数：

addr：传NULL即可，映射到具体的内存位置

length：映射区的大小，一般用文件的大小即可

prot：文件的读写权限

PROT\_READ读

PROT\_WRIFE 写

PROT\_READ | PROT\_WRIFE 读写

flags：映射区的内容是否同步到文件

MAP\_SHARED：同步到文件，进程通信必须设置同步

MAP\_PRIVATE：不同步到文件

fd：被映射文件的文件描述符

offset：映射文件的开始偏移量，必须是4k的整数倍，传0表示从文件其实位置开始映射

返回值：

成功返回：映射区的首地址

失败返回：MAP\_FAILED #define MAP\_FAILED ((void \*) -1)

内存映射区释放：

int munmap(void \*addr, size\_t length);

参数：

addr：映射区的首地址

length：映射区的大小

返回值：

成功返回：0

失败返回：-1，设置errno

代码示例：

1、[src\linux系统编程\进程通信\mmap.c](src/linux系统编程/进程通信/mmap.c)

### 守护进程

守护进程就是后台服务进程

#### 守护进程的特点

1、守护进程一般用d作为结尾

2、守护进程是脱离终端的

3、周期性的执行某些任务

#### 示例：

//父进程fork子进程, 然后退出

pid\_t pid = fork();

if(pid>0 || pid<0)

{

exit(0);

}

//子进程调用setsid函数创建会话

setsid();

//修改当前的工作目录

chdir("/home/itcast/log");

//重设文件掩码

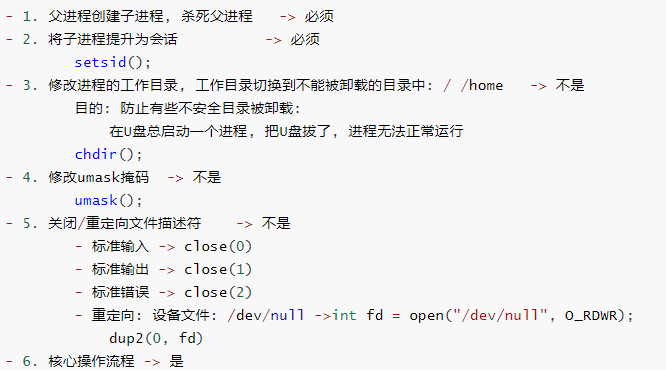
umask(0000);

//关闭文件描述符

close(STDIN\_FILENO);

close(STDOUT\_FILENO);

close(STDERR\_FILENO);



## 信号

### 信号说明:

1、一个进程执行过程中如果收到某个信号，会放下当前执行的代码，去处理这个信号，信号处理完成后在接着执行当前代码

2、信号可以被阻塞、捕获 - SIGKILL, SIGSTOP 这两个信号 不允许被捕捉、阻塞

#### 信号的状态

1、产生

2、未决

- 信号没有被处理

- 阻塞是未决的一种状态

3、递达

- 信号已经被递达，开始处理

- 递达不保证这个信号被正确处理结束，因为在处理一个信号的过程中可能收到另外一个信号，导致当前信号，处理中断。

#### 信号列表

可以通过命令：kill -l 查询当前系统有哪些信号

#### 信号的产生和默认处理动作

通过命令：man 7 signal 查看每个信号详细信息

### 信号处理函数：

typedef void (\*sighandler\_t)(int);

示例：

void函数名(int 处理的信号编号)

void lr\_sig(int signo)

注意：

1、信号处理函数执行中，默认会阻塞当前信号

2、信号处理函数执行中，如果收到其他未被阻塞信号，会执行会去执行其他信号

### 注信号处理函数1：（捕获信号）

功能：捕获某个信号，交给自定义的信号处理函数处理

#include <signal.h>

sighandler\_t signal(int signum, sighandler\_t handler);

函数参数：

signum：信号的编号，需要捕捉的信号

handler：自定义的函数名, 可以是SIG\_IGN--表示忽略, SIG\_DFL--表示行默认动作

函数返回值: 无

注意：

1、在不同的平台下，这个函数可能不同，所以建议使用sigaction函数

2、 SIGKILL, SIGSTOP 这两个信号 不允许被捕捉, 阻塞和忽略

3、处理当前信号过程中会自动阻塞当前信号

### 注册信号处理函数2（捕获信号）

#include <signal.h>

int sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact);

参数：

signum:需要捕捉的信号

act:传入参数,处理方式

oldact:传出参数,旧的处理方式

返回值:

成功返回：0

失败返回：-1，设置errno

所需要结构体：

struct sigaction

{

- void (\*sa\_handler)(int); // 信号处理函数

- void (\*sa\_sigaction)(int, siginfo\_t \*, void \*); //如果需要通过信号传输某些数据，就需要使用这个信号处理函数，可以不给值

- sigset\_t sa\_mask; //当前信号处理函数执行期间需要阻塞的信号，不清楚绑定的信号还是还是信号处理函数，没事过，应该是绑定了信号，一般来说初始化一下这个集合即可

- int sa\_flags; //传0即可，表示使用默认标识

- void (\*sa\_restorer)(void); //已经废弃不在使用，不用给值

};

注意：

1、 SIGKILL, SIGSTOP 这两个信号 不允许被捕捉, 阻塞和忽略

2、处理当前信号过程中会自动阻塞当前信号

### 给指定进程发送信号：

#include <sys/types.h>

#include <signal.h>

int kill(pid\_t pid, int sig);

函数参数：

pid：

pid>0:发送信号给指定程序

pid=0:发送信号给统一进程组的所有进程

pid<-1:不会

pid=-1:发送信号给，进程有权限发送的，系统中所有进程

sig:信号编号，建议使用宏，不同操作系统可以编号不同

返回值:

成功返回：0

失败放回：-1，设置errno

### 指定时间后发送SIGALRM信号

#include <unistd.h>

unsigned int alarm(unsigned int seconds);

参数:

seconds:几秒后发送信号

返回值：

若先前没有调用过alarm函数, 则返回0;

若先前有alarm函数调用, 返回上一次剩余的秒数

注意：

1、使用自然定时法

2、一个进程只有一个时钟, 若多次调用, 最后一次是有效的

3、alarm(0);就是取消时钟的意思

### 指定时间后发送信号

对比上个函数，这个可以重复发送信号

int setitimer(int which,

const struct itimerval \*new\_value,

struct itimerval \*old\_value);

参数：

which:

ITIMER\_REAL：使用自然定时法，真实时间，建议使用，发送的是SIGALRM信号

ITIMER\_VIRTUAL：只计算进程占用cpu的时间，发送的是SIGVTALRM信号

ITIMER\_PROF：计算占用cpu及执行系统调用的时间，发送的是SIGPROF信号

new\_value: 传入参数, 设置时钟触发的时间

old\_value: 传出参数, 先前的设置, 通常传NULL.

返回值：

成功返回：0

失败返回：-1，设置errno

注意：

过程中所需结构体：

struct itimerval

{

struct timeval it\_interval; //以后每次触发的间隔

struct timeval it\_value; //第一次触发的时间

};

struct timeval

{

time\_t tv\_sec; //秒

suseconds\_t tv\_usec; //微秒

};

### 未决信号集和阻塞信号集

说明：

1、信号被发送到程序内核的未决信号集，此时未决信号集会与阻塞信号集进行比较，如果信号未被阻塞，会被送达给程序

2、因为每个信号只占用一个位的空间，只能表示 存在 或者 不存在 两种可能，所以程序收到多个相同未决信号，只有一个会被保存在未决信号集中

3、子进程会拷贝父进程里面的阻塞信号集，未决信号集没测过

4、子进程会拷贝父进程里面的信号捕捉和处理

#include <signal.h>

所需结构体：

typedef struct

{

nsigned long int \_\_val[\_SIGSET\_NWORDS];

}\_\_sigset\_t;

信号集创建

sigset\_t set;

信号集置0,表示不阻塞(初始化操作)

int sigemptyset(sigset\_t \*set);

信号集置1

int sigfillset(sigset\_t \*set);

将某个信号加入信号集

int sigaddset(sigset\_t \*set, int signum);

将某个信号移出信号集

int sigdelset(sigset\_t \*set, int signum);

读取当前进程的未决信号集

int sigpending(sigset\_t \*set);

以上函数返回值：

成功返回：0

失败返回：-1，设置errno

判断某个信号是否在信号集中

int sigismember(const sigset\_t \*set, int signum);

返回值：

在返回：1

不在返回：0

错误返回：-1，设置errno

设置阻塞信号集（生效）

int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset);

参数：

how:处理方式

SIG\_BLOCK: 将用户设置的阻塞信息添加到内核中, 内核中原来的数据不变 mask |= set

SIG\_UNBLOCK: 清除用户在内核中设置的数据, mask &= ~set

SIG\_SETMASK: 用户传入信号集，覆盖内核阻塞信号集

set: 传入参数，是一个自定义信号集合。由参数how来指示如何修改当前信号屏蔽字。

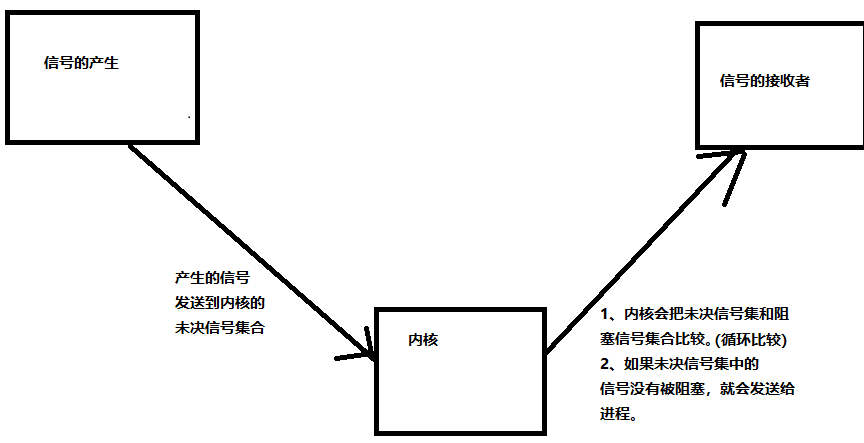
oldset：传出参数，保存旧的 内核阻塞信号集

返回值：

成功返回:0

失败返回:-1,设置erro

图片说明：



### 子进程结束信号SIGCHLD

产生SIGCHLD信号的条件

1 子进程收到SIGSTOP信号

2 子进程收到SIGCONT信号

3 子进程终止

产生SIGCHLD信号的条件

1. 子进程死了, 自杀, 他杀

2. 子进程被暂停了

3. 子进程由暂停状态重新恢复运行

以上三种情况子进程都会给父进程发送信号: SIGCHLD, 父进程默认处理动作是忽略这个信号

通过SIGCHLD信号回收子进程小案例

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<unistd.h>

#include <sys/types.h>

#include <signal.h>

#include <sys/wait.h>

int ii = 0;

//处理信号的函数

void my\_waitpid(int signo)

{

int wpid = -1;

//回收子进程

while (( wpid = waitpid(-1,NULL, WNOHANG)) !=0 && wpid != -1 )

printf("回收了[%d]个进程，进程pid=[%d]\n", ++ii, wpid);

}

int main()

{

int i = 0;

pid\_t pid = 0;

//捕捉子进程死亡信号

signal(SIGCHLD, my\_waitpid);

dzz:

//主进程创建1000个子进程

pid = fork();

if (pid > 0)

{

if (++i < 1000)

goto dzz;

while (ii<1000){}

printf("程序结束所有子进程已回收\n");

}

//子进程做一下打印

if (pid == 0)

printf("第[%d]个子进程产生，PID=[%d]\n",i+1,getpid());

return 0;

}

## 线程

### 线程说明

1、线程没有独立的虚拟地址空间，多个子线程和主线程共享同一个地址空间

2、线程是系统调度执行的最小单位，会抢CPU资源

3、线程不是linux系统的库函数，是第三方库函数

4、线程拥有的一些独立资源

- 线程id

- 处理器现场和栈指针（内核栈）

- 独立的栈空间（用户空间栈）

- errno 变量

- 阻塞信号集

5、多个线程同一时间操作同一个数据时，需要加锁，防止数据异常

6、多个线程间通信 “条件变量”、“信号量”

7、子线程执行中发生异常会导致整个进程死亡

### 创建线程

#include <pthread.h>

int pthread\_create(pthread\_t \*thread,

const pthread\_attr\_t \*attr,

void \*(\*start\_routine) (void \*),

void \*arg);

参数：

thread：传出参数保存线程ID

attr:线程属性，设置线程是否分离状态，传NULL默认就是不分离

//定义线程属性变量

pthread\_attr\_t attr;

//初始化线程属性变量

pthread\_attr\_init(&attr);

//设置线程属性为分离状态

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED);

//释放线程属性

ret = pthread\_attr\_destroy(&attr);

void \*(\*start\_routine) (void \*)：线程执行的函数

arg：需要传入线程的参数，给第三个参数传参

返回值:

成功返回：0

失败返回：错误号, 可以使用strerror函数打印错误原因

### 使当前线程结束

#include <pthread.h>

功能：线程退出时使用这个函数，不要使用return，主线程使用return 会使其他线程也结束

void pthread\_exit(void \*retval);

参数：

retval：退出状态（需要使用全局或者申请的堆空间），不关心退出状态传NULL

注意：

1、是在需要退出的线程处调用

2、主线程退出必须使用这个函数否则其他线程也会死亡

### 获取当前的线程ID

#include <pthread.h>

pthread\_t pthread\_self(void);

返回值：返回线程的ID

### 回收子线程资源

#include <pthread.h>

功能：子线程运行结束后，主线程负责回收子进程的资源，设置线程分离后，不需要主线程回收子资源

int pthread\_join(pthread\_t thread, void \*\*retval);

参数：

thread：需要回收的子线程ID

retval：退出状态 pthread\_exit函数的参数，在这接收，不关心退出状态可以传NULL

返回值：

成功返回：0

失败: 返回错误号, 可以使用strerror函数打印错误原因

注意：

1、分离后的线程无法阻塞等待退出

2、子线程未结束函数会阻塞等待子线程退出

### 设置线程分离

#include <pthread.h>

功能：设置线程分离，子线程退出后不需要主线程回收资源

int pthread\_detach(pthread\_t thread);

参数：

thread：线程ID

返回值:

成功: 返回0

失败: 返回错误号, 可以使用strerror函数打印错误原因

### 取消线程

#include <pthread.h>

功能：在主线程中，取消一个执行中的子线程

int pthread\_cancel(pthread\_t thread);

参数：

thread：需要取消的线程ID

返回值:

成功: 返回0

失败: 返回错误号, 可以使用strerror函数打印错误原因

注意：

1、线程取消不是立即执行的，需要等待线程到达某个取消点(检查点)，可粗略认为一个系统调用(进入内核)即为一个取消点，调用pthread\_testcancel函数设置一个取消点。

### 比较两个线程是否相同

#include <pthread.h>

int pthread\_equal(pthread\_t t1, pthread\_t t2)；

参数：

t1：需要比较的线程ID

t2：需要比较的线程ID

返回值：

两个线程相同返回：非0 值

两个线程不同返回：0

注意：

1、pthread\_t线程ID，在不同的平台下可能封装不同，有些平台下pthread\_t线程ID，是一个结构体，所以比较线程需要使用这个函数

### 线程同步

多个线程同时访问一个数据，会出现数据异常所以需要线程同步

#### 互斥锁

说明：多个线程同时对一个数据操作时，会导致异常，所以需要访问共享数据时，建议加锁，某个线程在访问时，其他线程阻塞等待

1 定义一把锁

pthread\_mutex\_t mutex;

2 初始化锁（第二个参数可以传NULL）

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,

const pthread\_mutexattr\_t \*restrict attr);

参数：

mutex：传入一把锁

attr：设置锁的属性，不设置可以传NULL

返回值：

成功返回：0

失败返回：错误号

3 加锁

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

参数：

mutex：传入一把锁

返回值：

成功返回：0

失败返回：错误号

4 解锁

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

参数：

mutex：传入一把锁

返回值：

成功返回：0

失败返回：错误号

5 释放互斥锁

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

参数：

mutex：传入一把锁

返回值：

成功返回：0

失败返回：错误号

注意：

1、线程对某个数据加锁前，最好释放当前的锁，防止死锁情况出现

2、线程对某个数据加锁前，请留意加锁代码阻塞情况，防止死锁情况出现

代码示例：

[src\linux系统编程\多线程\线程同步\互斥锁\main.c](src/linux系统编程/多线程/线程同步/互斥锁/main.c)

#### 读写锁

说明：读写锁时一种特别的锁，读时可以并行，写时独占

读写锁的使用步骤:

1 定义一把读写锁:

pthread\_rwlock\_t wrlock;

2 初始化读写锁:

int pthread\_rwlock\_init(pthread\_rwlock\_t \*restrict rwlock,

const pthread\_rwlockattr\_t \*restrict attr);

参数:

- rwlock: 读写锁

- attr: 读写锁属性, 使用默认属性, 设置为: NULL

返回值：

成功返回：0

失败返回：错误号

3添加读锁:

int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*rwlock); //

函数功能：加读锁

参数：

rwlock：传入锁

返回值：

成功返回：0

失败返回错误号：错误号

添加写锁:

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

函数功能：加写锁

参数：

rwlock：传入锁

返回值：

成功返回：0

失败返回错误号：错误号

4 解锁

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

参数：

rwlock：传入锁

返回值：

成功返回：0

失败返回错误号：错误号

：

4 释放读写锁:

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

参数：

rwlock：传入锁

返回值：

成功返回：0

失败返回错误号：错误号

注意：

1. 死锁：访问其他锁时建议解开自己的锁
2. 读写锁：读并行，写独占，当读写同时等待锁的时候写的优先级高

3. 适合于对数据结构读的次数远大于写的情况

#### 条件变量

说明：生产者和消费者模型，使用条件变量，生产者生产出商品后通知消费者消费，没有商品时消费者阻塞等待，防止供不应求

条件变量相关函数和使用步骤:

1 定义条件变量:

pthread\_cond\_t cond;

2 初始化条件变量:

int pthread\_cond\_init(pthread\_cond\_t \*restrict cond,

const pthread\_condattr\_t \*restrict attr);

参数:

- cond: 条件变量的地址

- attr: 使用默认属性, 这个值设置为NULL

返回值：

成功返回：0

失败返回：错误号

3阻塞线程等待唤醒

int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond,

pthread\_mutex\_t \*restrict mutex);

函数功能：调用该函数阻塞，同时自动解锁，等待signal 和 broadcast函数唤醒后，自动加锁 并解除阻塞 运行

参数:

- cond: 条件变量

- mutex: 传入锁

返回值：

成功返回：0

失败返回：错误号

注意：

1、唤醒后，执行前在使用while 循环判断一下，防止同时唤醒了多个线程

4 唤醒被wait 函数阻塞的一个或多个线程

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

参数：

- cond: 条件变量

返回值：

成功返回：0

失败返回：错误号

注意：

1、可以使用 pthread\_cond\_broadcast 唤醒所有的阻塞在 pthread\_cond\_wait / pthread\_cond\_timedwait 函数上的线程。

4 释放条件变量

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

参数：

cond：条件变量

返回值：

成功返回：0

失败返回：错误号

代码示例：

[src\linux系统编程\多线程\线程同步\条件变量\main.c](src/linux系统编程/多线程/线程同步/条件变量/main.c)

#### 信号量

说明：生产者和消费者模型，使用信号量，生产出商品后 ++ ，消费后 -- ，防止供不应求，++,--函数自带锁

头文件

#include <semaphore.h>

信号量的使用

1、定义信号量

sem\_t sem;

2、初始化信号量

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

参数:

sem: 信号量

pshared: 传 0 表示用于多线程, 非 0 值表示用于多进程

value: 信号量的初始值，小于等于 0 ，-- 函数阻塞

返回值:

成功: 返回0

失败: 返回-1, 并设置errno

3、++ 函数

int sem\_post(sem\_t \*sem);

功能: 调用该函数一次, 相当于sem++

参数:

sem: 信号量

返回值:

成功: 返回0

失败: 返回-1, 并设置errno

4、-- 函数

int sem\_wait(sem\_t \*sem);

功能: 调用该函数一次, 相当于sem--, 当sem为 0 的时候, 引起阻塞

参数:

sem: 信号量

返回值:

成功: 返回0

失败: 返回-1, 并设置errno

5、释放信号量

int sem\_destroy(sem\_t \*sem);

功能: 释放信号量资源

参数:

sem: 信号量

返回值:

成功: 返回0

失败: 返回-1, 并设置errno

代码示例：

[src\linux系统编程\多线程\线程同步\信号量\main.c](src/linux系统编程/多线程/线程同步/信号量/main.c)

# Linux网络编程

## 辅助函数

### 大小端转换

#include <arpa/inet.h>

uint16\_t htons(uint16\_t hostshort);

功能：小端数据转大端、短整型转换、小端也叫主机字节序，大端也叫网络字节序

参数：

hostshort：传入端口号

返回值：

成功返回：转换之后的数据

失败返回：man没写失败情况

注意：

1、大端数据转小端 ntohs() 短整型，用于端口号

2、atoi()：把字符串转换为整型，返回值没写失败情况

3、uint32\_t htonl(uint32\_t hostlong); 长整型小端转大端，用于IPV4

4、uint32\_t ntohl(uint32\_t netlong); 长整型大端转小端，用于IPV4

6、这是整型转换，字符串配合其他函数使用 或者 使用其他函数

7、需要什么字节序直接转，如果字节序正确那结果不会改变

### 点分制十进制字符串转IP地址

#include <arpa/inet.h>

int inet\_pton(int af, const char \*src, void \*dst);

功能：把点分制十进制字符串，转为网络字节序的整型IP地址

参数：

af：地址族协议

AF\_INET：IPV4

AF\_INET6：IPV6

src：点分制十进制字符串

dst：传出参数，保存转换后的IP地址

返回值：

成功返回：1

失败返回：-1

传入的点分十进制字符串无效返回：0

#include <arpa/inet.h>

const char \*inet\_ntop(int af, const void \*src, char \*dst, socklen\_t size);

功能：把网络字节序转为，点分十进制字符串

参数：

af：地址族协议

AF\_INET：IPV4

AF\_INET6：IPV6

src：网络字节序的整型IP地址

dst：装换成功后的 点分十进制字符串 存储位置

size：第三个参数dst 的内存大小

返回值：

失败返回：NULL

成功返回：非空指针，指向第三个参数的指针

### 利用域名获取IP地址

#include <netdb.h>

extern int h\_errno;

struct hostent \*gethostbyname(const char \*name);

参数：

name：域名字符串

返回值：

成功返回：保存ip信息的结构体

失败返回：NULL

hostent结构体：

struct hostent

{

char \*h\_name; 官方域名(Official domain name)。不懂

char \*\*h\_aliases;多个域名可以访问统一主页，同一IP也可以绑定多个域名，保存除了官方域名外的其他域名。不懂

int h\_addrtype; Ipv4或者Ipv6保存在这里，不懂

int h\_length;保存IP地址的长度，不懂

char \*\*h\_addr\_list; 保存域名对应的IP地址

}

示例代码：

struct hostent \* host = gethostbyname("www.naver.com");

printf("官方域名：%s\n",host->h\_name);

for (int i = 0; host->h\_aliases[i]; i++)

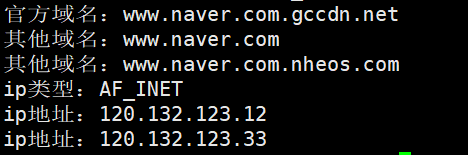
printf("其他域名：%s\n",host->h\_aliases[i]);

printf("ip类型：%s\n",(host->h\_addrtype==AF\_INET)?"AF\_INET":"AF\_INET6");

for (int i = 0; host->h\_addr\_list[i]; i++)

printf("ip地址：%s\n", inet\_ntoa(\*(struct in\_addr\*)host->h\_addr\_list[i]));

运行结果：



### 利用IP获取域名

#include <netdb.h>

#include <sys/socket.h> /\* for AF\_INET \*/

extern int h\_errno;

struct hostent \*gethostbyaddr(const void \*addr,

socklen\_t len,

int type);

功能：传入IP地址，返回域名

参数：

addr：含有IP地址的in\_addr结构体指针。

len：IPV4传4，IPV6传16

type：IPV4传AF\_INET,IPV6传AF\_INET6

返回值：

成功返回：保存ip信息的结构体

失败返回：NULL

注意：

1. 这个函数好像需要某种服务才可以使用，一般情况下函数调用不成功，返回NULL,可以尝试IP地址为127.0.0.1

示例代码：

struct sockaddr\_in addr;

memset(&addr,0,sizeof(addr));

addr.sin\_addr.s\_addr = inet\_addr("220.181.57.216");

struct hostent \* host =

gethostbyaddr((char\*)&addr.sin\_addr.s\_addr,4,AF\_INET);

printf("官方域名：%s\n", host->h\_name);

for (int i = 0; host->h\_aliases[i]; i++)

printf("其他域名：%s\n", host->h\_aliases[i]);

printf("ip类型：%s\n", (host->h\_addrtype == AF\_INET) ? "AF\_INET" : "AF\_INET6");

for (int i = 0; host->h\_addr\_list[i]; i++)

printf("ip地址：%s\n", inet\_ntoa(\*(struct in\_addr\*)host->h\_addr\_list[i]));

### 半关闭函数

#include <sys/socket.h>

int shutdown(int sockfd, int how);

参数：

sockfd：需要关闭的套接字

how：关闭的方式

SHTU\_RD：断开输入流

SHUT\_WR：断开输出流

SHUT\_RDWR：同时断开I/O流

返回值：

成功返回：0

失败返回：-1

### 端口复用

#include <sys/types.h>

#include <sys/socket.h>

// 设置套接字属性

int setsockopt(int sockfd, int level, int optname, const void \*optval, socklen\_t optlen);

功能：这里只介绍端口复用，端口复用需要在bind前调用。

参数：

sockfd: 要操作的套接字

optname: 端口复用的级别(二选一)

SO\_REUSEADDR

SO\_REUSEPORT

optval: 端口复用-> 对应的是整形数

1: 可以复用

0: 不能复用

optlen: optval参数的内存大小

返回值：

成功返回：0

失败返回：-1，设置errno

注意：

1、TCP通信时主动关闭连接方，在四次挥手时，会等待2MSL时间，才会关闭程序，这个时间会占用端口

2、设置端口复用后，只有最后一个bind这个端口的程序能使用这个端口

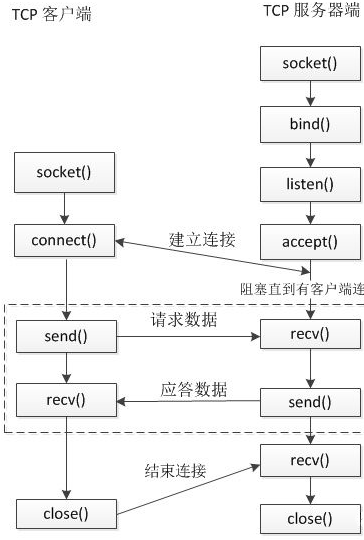
3、



## TCP通信

### tcp通信说明

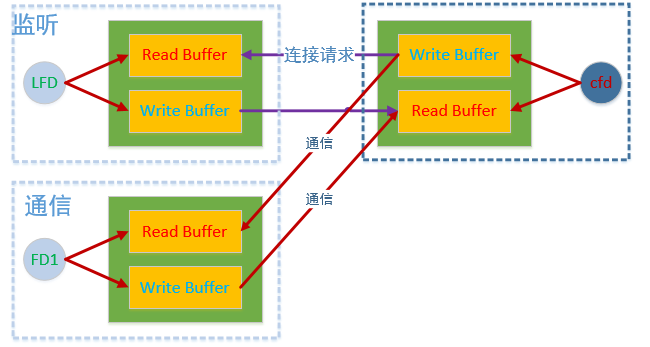
1、TCP通信流程图



2、 TCP套接字缓冲区

write写满时会阻塞

read 读时没有数据也会阻塞



3、TCP通信是面向连接的，安全的，流式传输协议。TCP建立连接时会3次握手，断开连接时会4次挥手

### 创建套接字

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

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

参数：

domain：地址族协议

PF\_INET：ipv4互联网协议族

PF\_INET6：ipv6互联网协议族

PF\_LOCAL：本地通信的UNIX协议族

PF\_PACKET：底层套接字协议族

PF\_IPX：IPX Novell协议族

type：通信过程中使用的协议

SOCK\_STREAM：流式协议（TCP）

SOCK\_DGRAM：报式协议（UDP）

protocol：具体的通信过程中使用的协议

0：一般写0，上个参数中的协议，默认就是TCP和UDP

IPPROTO\_TCP：满足ipv4和面向连接的协议

IPPROTO\_UDP：满足ipv4和面向消息的协议

返回值：

成功返回：文件描述符

失败返回：-1

注意：

1、domain：套接字中的协议族信息，也可以把PF开头换成AF开头

### 给套接字分配（绑定）ip地址和端口

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

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

参数：

sockfd：用于监听的服务器端套接字

addr：存有地址和端口的结构体

一般使用sockaddr\_in结构体，使用时强转

addrlen：第二个结构体的长度

返回值：

成功返回：0

失败返回：-1

所需结构体：

头文件：

#include<arpa/inet.h>

struct sockaddr\_in

{

sa\_family\_t sin\_family; //地址族

AF\_INET：ipv4网络协议中使用的地址族

AF\_INET6：ipv6网络协议中使用的地址族

AF\_LOCAL：本地通信中采用的UNIX协议地址族

uint16\_t sin\_port; //TCP/UDP端口号

htons()：转大端

atoi()：把字符串转换为整型

struct in\_addr sin\_addr; //32位（四个字节）IP地址

char sin\_zero[8]; //不使用，但是必须置0，使用memset

}

struct in\_addr

{

In\_addr\_t s\_addr; //32位ip地址

htonl()：转大端

INADDR\_ANY：自动获取服务器端的IP的地址（整型）

点分制转大端：int inet\_pton(int af, const char \*src, void \*dst);

AF\_INET：

AF\_INET6

成功返回：1

失败返回：-1

传入的点分十进制字符串无效返回：0

}

注意：

1、测试时使用IP地址不是自动获取的情况下，连接时也不能使用127.1

### 进入等待连接请求状态

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

int listen(int sockfd, int backlog);

参数：

sockfd：用于监听的服务器端套接字

backlog：等待处理的连接队列的长度

指定的数值不能大于 /proc/sys/net/core/somaxconn 中存储的数据, 默认为128

返回值：

成功返回：0

失败返回：-1

### 受理客服端连接请求

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

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

参数：

sockfd：服务器端套接字

addr：传入一个sockaddr\_in结构体用于存储客服端的地址信息

addrlen：第二个结构体的内存大小，需要socklen\_t类型的指针，所以请在外面定义，直接sizeof不行

返回值：

成功返回：用于和客服端通信的套接字

失败返回：-1

注意：

1、函数带阻塞

### 客服端请求连接

#include <sys/types.h> /\* See NOTES \*/

#include <sys/socket.h>

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

参数：

sockfd：创建好的客服端套接字

addr：传入保存有服务器IP和端口sockaddr\_in结构体，

addrlen：第二个结构体的长度

返回值：

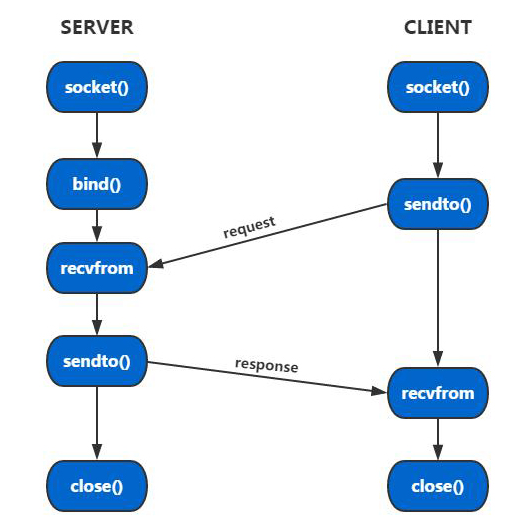
成功返回：0

失败返回：-1,

## UDP通信

### UDP通信说明

1、通信流程图



2、UDP通信时面向无连接的，不安全的，报式传输协议，效率高。

### UDP发送数据

#include <sys/types.h>

#include <sys/socket.h>

ssize\_t sendto(int sockfd,

const void \*buf,

size\_t len,

int flags,

const struct sockaddr \*dest\_addr,

socklen\_t addrlen);

参数：

sockfd：用于传输数据的UDP套接字

buf：需要传输的数据

len：传输多少字节的数据

flags：可选项参数，传0即可

dest\_addr：存有目标地址信息的结构体

addrlen：dest\_addr的大小

返回值：

成功返回：传输的字节数

失败返回：-1

### UDP接收数据

#include <sys/types.h>

#include <sys/socket.h>

ssize\_t recvfrom(int sockfd,

void \*buf,

size\_t len,

int flags,

struct sockaddr \*src\_addr,

socklen\_t \*addrlen);

功能：阻塞等待接收数据

参数：

sockfd：用于接收数据的UDP套接字

buf：接收到的数据

len：接收多少字节的数据

flags：可选项参数，若没有则传递0

src\_addr：保存发送端地址信息的结构体，不需要可以传NULL

addrlen：结构体src\_addr的大小

返回值：

成功返回：接收到的字节数

失败返回：-1

UDP通信示例：

1、服务器

[src\linux网络编程\UDP\UDP通信案例\servers.c](src/linux网络编程/UDP/UDP通信案例/servers.c)

2、客服端

[src\linux网络编程\UDP\UDP通信案例\client.c](src/linux网络编程/UDP/UDP通信案例/client.c)

## 案例：

### 多进程服务器

说明：使用多进程实现并发访问

[src\linux网络编程\多进程服务器\tcp.c](src/linux网络编程/多进程服务器/tcp.c)

### 多线程服务器

说明：使用多线程实现并发访问

[src\linux网络编程\多线程服务器\tcp.c](src/linux网络编程/多线程服务器/tcp.c)

### select IO多路转接

说明：

1、使用select函数监视套接字的变化，实现并发访问。

2、select可以跨平台在windows使用。

3、fd\_set集合限制，最多监视1024个套接字

#include <sys/select.h>

/\* According to earlier standards \*/

#include <sys/time.h>

#include <sys/types.h>

#include <unistd.h>

int select(int nfds,

fd\_set \*readfds,

fd\_set \*writefds,

fd\_set \*exceptfds,

struct timeval \*timeout);

参数：

nfds：需要监听的文件描述符数量，当前创建出来的最大套接字ID+1，或者直接1024

readfds：监听读缓冲区是否有数据，调用时传入需要监听的套接字集合，结束时返回读缓冲区有数据的集合，不关心可以传NULL

writefds：监听写缓冲区是否可以写入数据(缓冲区没满就可以写)，调用时传入需要监听套接字集合，结束时返回可以写入数据的套接字集合，不关心可以传NULL

exceptfds：监听套接字是否有异常，使用参照上面两个参数，不关心可以传NULL

timeval：设置单次监听最长阻塞时间，传NULL一直阻塞监听

返回值：

失败返回：-1

成功返回：发生变化的文件描述符数

返回0：表示本次监听结束没变化的文件描述符

注意：

1、下列函数可以用于用于操作fd\_set类型

void FD\_CLR(int fd, fd\_set \*set); 把某个文件描述符移除

int FD\_ISSET(int fd, fd\_set \*set);判断某个套接字是否存在

存在返回：1

不存在返回：0

void FD\_SET(int fd, fd\_set \*set);把某个文件描述符加入

void FD\_ZERO(fd\_set \*set);初始化，置0

示例代码：

[src\linux网络编程\select\_IO多路转接\select.c](src/linux网络编程/select_IO多路转接/select.c)

[src\linux网络编程\select\_IO多路转接\temp\_select.c](src/linux网络编程/select_IO多路转接/temp_select.c)

### poll

说明：

1、poll是linux下select 的简单改进版

2、使用数组管理需要监听的套接字，可以监听超过1024个套接字

#include <poll.h>

int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);

功能：和select一样，监听套接字

参数：

fds：需要监听的数组

nfds：需要监听多少个套接字

timeout：阻塞时长

0：不阻塞

-1：永久阻塞

>0：阻塞时长，单位毫秒

返回值：

失败返回：-1

成功返回：检测的数组中有多少个文件描述符变化

结构体：

struct pollfd

{

int fd; /\* 委托内核检测的文件描述符 \*/

short events; /\* 委托内核检测文件描述符的什么事件 \*/

POLLIN 读缓冲区是否有数据

POLLOUT 写缓冲区是否能写入

short revents; /\* 文件描述符实际发生的事件 \*/

};

代码示例：

[src\linux网络编程\poll\server.c](src/linux网络编程/poll/server.c)

### epoll

说明：

1、使用epoll 实现高性能并发服务器

2、epoll有两种工作模式，默认使用LT模式：读缓冲区有数据就会通知

3、ET模式高性能模式：读缓冲区发生改变才会通知

- ET模式下一般需要使用while保证数据被读完，所以建议设置套接字为非阻塞模式

- ET模式只需要在上树时加个宏即可使用

#include <unistd.h>

#include <fcntl.h>

// 设置套接字为非阻塞

// 获取套接字flag属性

int flag = fcntl(cfd, F\_GETFL);

// 添加非阻塞属性

flag = flag | O\_NONBLOCK;

// 将新的flag属性设置给cfd

fcntl(cfd, F\_SETFL, flag);

#include <sys/epoll.h>

int epoll\_create(int size);

功能：创建一颗红黑树，维护需要监视的套接字

参数:

size: 没意义, 随便写个数就行，别写0

返回值;

成功返回: 文件描述符, 操作epoll树的根节点

失败返回：-1

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

功能：对epoll 树进行管理：添加、删除节点，修改节点属性

参数:

- epfd: 树的根节点，上一个函数的返回值

- op: 要进行什么样的操作

EPOLL\_CTL\_ADD: 注册新节点, 添加到红黑树上

EPOLL\_CTL\_MOD: 修改套接字的属性

EPOLL\_CTL\_DEL: 从红黑树上删除节点

- fd: 要操作的套接字

- event: 要监视什么操作，删除节点传NULL

返回值：

成功返回：0

失败返回：-1，设置ernno

所需结构体：

struct epoll\_event {

uint32\_t events; 要监视的属性,可以位运算同时使用多个宏

- EPOLLIN 监视读

- EPOLLOUT 监视写

- EPOLLERR 监视错误

- EPOLLET ET触发模式

epoll\_data\_t data; 联合体，在下面

};

typedef union epoll\_data {

void \*ptr; // 复杂，可以使用结构体，给每一个节点一些自定义属性和方法（回调方法），但是第一个属性必须是 fd 保证能正确赋值

int fd; // 简单，套接字

uint32\_t u32;

uint64\_t u64;

} epoll\_data\_t;

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

功能：阻塞等待系统通知，处理有变化的套接字

参数：

epfd：树的根节点，第一个函数的返回值

epoll\_event：传出参数，保存发生变化的套接字，使用数组保存多个

maxevents：第二个参数的数组个数

timeout：阻塞时间

* 0，不阻塞
* -1，一直阻塞
* >0，阻塞时长，毫秒

返回值：

成功返回：变化的套接字个数

失败返回：-1

代码示例：

1、默认LT模式epoll 示例

[src\linux网络编程\epoll\epoll.c](src/linux网络编程/epoll/epoll.c)

2、ET模式代码示例

[src\linux网络编程\epoll\epoll\_et.c](src/linux网络编程/epoll/epoll_et.c)

3、ET模式、prt指针的使用

[src\linux网络编程\epoll\epoll\_et\_prt.c](src/linux网络编程/epoll/epoll_et_prt.c)

### UDP广播

说明：

1、广播强制发送数据，组播可以选择是否接收数据

2、广播就是给局域网内同一网段多个设备发送数据

2、只能在局域网中使用

3、客服端只要绑定了服务器广播使用的端口，就可以收到广播数据

服务器流程：

1、创建通信UDP的套接字

2、设置套接字属性为广播，setsockopt();

3、通信，发送广播的数据，(IP地址自动获取)

4、关闭通信的套接字

客服端：

1、创建通信的UDP套接字

2、绑定广播的端口(IP地址自动获取)

3、通信，接收广播数据

4、关闭套接字

代码示例：

1、服务器

[src\linux网络编程\UDP\UDP广播\servers.c](src/linux网络编程/UDP/UDP广播/servers.c)

2、客服端

[src\linux网络编程\UDP\UDP广播\client.c](file:///D:\zm\我的笔记\src\linux网络编程\UDP\UDP广播\client.c)

网上UDP广播资料

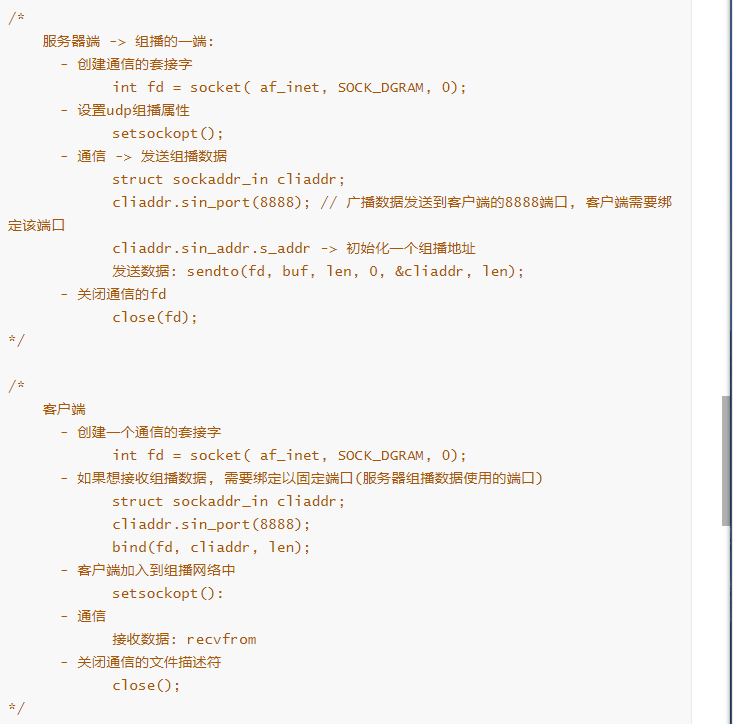
<https://blog.csdn.net/weixin_36750623/article/details/83575412>

### UDP组播

说明：



通信流程：



代码示例：

1、服务器

[src\linux网络编程\UDP\UDP组播\server.c](src/linux网络编程/UDP/UDP组播/server.c)

2、客服端

[src\linux网络编程\UDP\UDP广播\client.c](src/linux网络编程/UDP/UDP广播/client.c)

### TCP本地套接字

说明：

1、用于本机进程间通信

2、一般使用TCP通信

3、使用 套接字文件 替代端口和IP

代码示例：

1、服务器

[src\linux网络编程\本地套接字TCP实现\servers.c](src/linux网络编程/本地套接字TCP实现/servers.c)

2、客服端

[src\linux网络编程\本地套接字TCP实现\client.c](src/linux网络编程/本地套接字TCP实现/client.c)

# SQL

## 增删改查

### select

**select**：结果显示

DISTINCT：去除重复结果

**from**： 表名

**where**：对表的数据进行筛选

1、模糊查询

NOT LIKE：相当于 !=

LIKE：相当于 =

%：表示0到多个字符

\_：表示一个字符

escape '/'：自定义转义字符

2、筛选NULL数据

is：相当于 =

is not：相当于不等于 !=

**group** **by**：分组，分组后select只能显示分组的字段，或者统计函数

**having**：对分组结果进行筛选，能使用统计函数的结果对分组结果进行筛选

**order** **by**：排序，asc升序，desc降序

**select 查询逻辑**

首先从from中查询结果，

然后通过where进行筛选，

然后通过group by进行分组，

然后通过having对分组结果筛选，可以使用统计函数筛选

然后使用order by 进行排序，

最后通过 select 显示

**注意点：**

**1、外连接：**

说明：1、外连接就是多表查询时，保留没有通过条件筛选的数据，

2、可用于多表查询分组时，统计数据为0的组

使用方法：

1、使用jion，on连接表

2、然后在jion前面加left 或者right 或者full

3、加left就是左外连接，保留左边表的数据

4、加right就是右外连接，保留右边表的数据

5、加full就是全外连接，保留两张表的数据

jion的使用：

1、jion替换from中的逗号

2、on 替换 where

3、on 后面可以接着写where

**2、子查询：**

说明：1、一个查询的select语句里边嵌套了另一个select语句

2、select、from、where、having都可以使用子查询

3、子查询返回多行数据使用IN、ANY、ALL判断

IN()：= 符号

IN(子查询)

ANY：集合中的任意一个数据

>= ANY(子查询)

ALL：集合中的所有数据

>=ALL(子查询)

小示例：查询工资大于李磊员工

select \*

from 员工表

where 工资 >

(

select 工资

from 员工表

where ename = '李雷'

)

**3、集合：**

说明：在一个select语句下面在接一个select语句使用集合关键字进行分割

关键字：

并集 union

全并集 union all

交集 intersect

差集 minus

示例：

select \*

from emp

where deptno = 10

**union all**

select \*

from emp

**4、mysql Top**

**limit：**m,n

m可选， 表示跳过前面m条记录

n必填， 只显示n条记录

**5、Oraclet Top**

取前几条数据：

WHERE ROWNUM < 3

由于先判断where 后执行排序所以取Top N，需要使用子查询先排序：

SELECT \*

FROM (

SELECT \*

FROM EMP

ORDER BY DEPTNO asc

)

WHERE ROWNUM < 3

### insert into

语法：

**insert into 表名(列名,列名2.....)**

**values(…………),**

**values(…………),**

**………;**

列名可以忽略

示例：

INSERT INTO gua\_64 (gua\_id,gua\_name,gua\_message)VALUES

(1,'乾为天','元亨利贞。'),

(2,'坤为地','元亨。。'),

### update

语法：

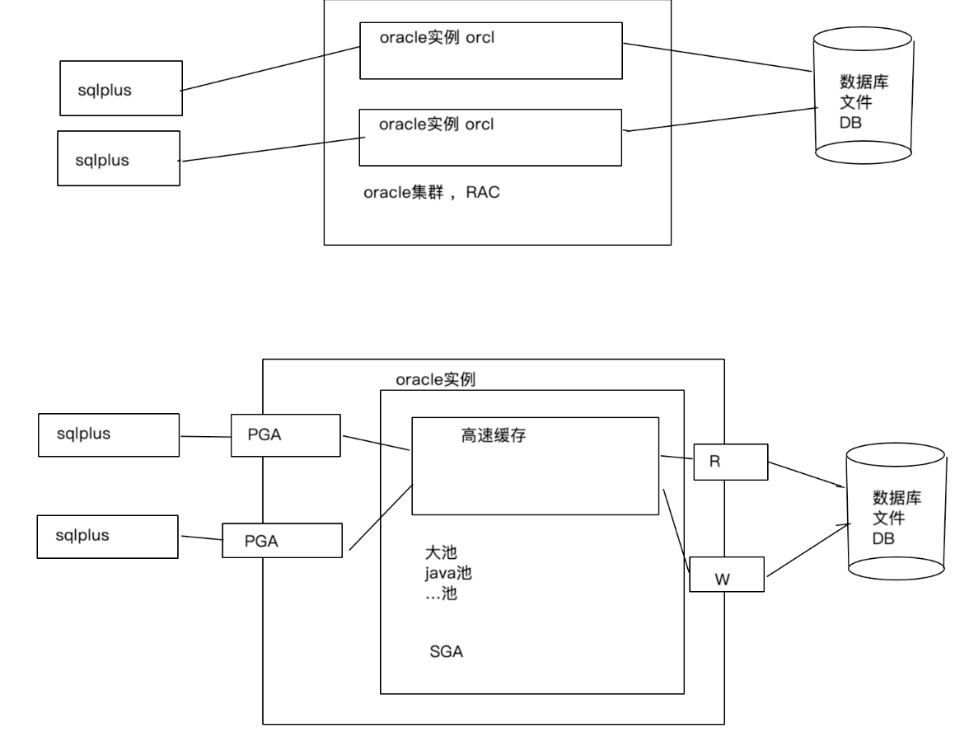
update 表名 set 列名1=值1,列名2=值2..... where 更新条件

### delete

语法：

delete 表名 where 删除条件

## Oracle



### 基本操作

a、管理员登陆： sqlplus / as sysdba

b、启动： startup

c、停止： shutdown

d、解锁用户： alter user 用户名account unlock;

e、修改密码： alter user 用户名identified by "密码";

f、本地登陆： sqlplus 用户名/密码

g、远程登陆： sqlplus 数据库用户名/密码@//IP地址/orcl

h、查看当前用户名称： show user

i、查看当前用户拥有的表： select \* from tab

j、查看当前表结构： desc 表名

k、开启监听服务，开启后允许远程登陆（在linux终端执行）

a、启动： lsnrctl start

b、停止： lsnrctl stop

c、查看状态： lsnrctl status

l、查看用户的表

select \* from all\_tables where owner='用户名';

m、

### 常用函数

**统计函数**

sum --求和

sum(字段)

count --统计个数，NULL的情况不统计

count(字段)

count(distinct 字段)，统计中去重

max/min –返回最大 或 最小值

max(字段)

min(字段)

avg --求平均值，忽略值为NULL的字段

avg(字段)

**字符串处理**

lower 将字符串转小写

lower('hEllo,woRLD')

upper 将字符串转大写

upper('hEllo,woRLD'),

initcap 将首字母变成大写

initcap('hEllo,woRLD')

concat 字符串拼接

concat('hello','world')

只支持两个参数如果需要使用多个，通过返回值继续拼接

concat(concat('hello','world'),'123')

substr 提取子串

substr('helloworld',3) --从第三个字符开始一直提取到最后

substr('helloworld',3,5) --从第三个字符开始取5个字符

instr 查找某个字符串是否另一个字符串的子串

select instr('hello world', 'llo')

是的话返回：3，首字符出现的位置

不是的话返回：0

lpad,rpad 左右填充

lpad('hello','10','\*') from dual;

向左填充，填充后字符串的长度为10

trim 裁剪

trim ( ' Hello world ') --默认裁剪空白字符

trim ( 'H' from 'Hello worldHHHH') --裁剪掉H字符

replace 替换子串

replace('Hello world','l','\*')

第二个参数：被替换的函数

第三个参数：替换的字符串

**数值处理函数**

round --四舍五入

round(45.926,2) 结果：45.93

trunc --截取

trunc(45.926,2) 结果：45.92

ceil --向上取整

ceil(45.926), 结果：46

floor --向下取整

floor(45.926), 结果：45

mod --取模 %

mod(1600,1000) 结果：600

**转换函数**

to\_char --数字转字符串

to\_number --字符串转数字

to\_char --日期转字符串

to\_date --字符串转日期

**日期函数**

sysdate --获取当前系统时间

select sysdate from dual ; oracle使用不在括号

SELECT SYSDATE(); mysql使用加括号

months\_between --计算两个时间相差的月份，自动计算大小月

months\_between(sysdate,sysdate)

last\_day --获取当前月份的最后一天

last\_day(sysdate)

**通用函数（**oracle，特有函数**）**

nvl(字段,val)

如果字段为null，那么返回val，否则返回字段的值

nvl2(字段,val1,val2)

如果字段为null，返回val2 ，否则返回val1

### 表的实现

**创建表：**

create table 表名

(

列名1 类型1,

列名2 类型2,

......

)

**删除表：**

drop table 表名

**约束：**

关键字：

主键 primary key

非空 not null

唯一 unique

检查 check

外键 foreign key

约束使用示例：

create table student

(

sid number constraint pk\_student primary key, --学生Id主键约束

--字段名 字段类型 constraint 约束名 约束关键字

sname varchar2(20) constraint nn\_student\_name not null,--学生姓名非空约束

email varchar2(20) constraint un\_student\_email unique --学生邮件唯一约束

constraint nn\_student\_email not null, --同时邮件可再设非空，没有“，”

age number constraint chk\_student\_age\_min check(age > 10), --学生年龄设置check约束

gender varchar2(6) constraint chk\_student\_gender check(gender in ('男', '女')),

deptno number constraint fk\_student references dept (deptno) ON DELETE SET NULL

)

### 其他知识点

#### 事务

说明：

1、事务是一个不可分割的单元

2、多条SQL语句需要一起执行时，打开事务。如发生问题回滚，没有问题就提交。

事务的提交：

1、显示提交：commit

2、隐示提交

a、使用创表语句create table，会隐式提交事务

b、正常退出，exit/quit

3、显示回滚：rollback

4、隐式回滚：掉电、宕机、非正常退出

#### 视图

说明：

视图就是创建一个虚表，这个表的内容就是一条select语句，每次查看这个虚表其实就是执行这个select语句

视图的创建：

create view 视图名as 子查询

视图的删除：

drop view 视图名

#### 序列

说明：

序列就是一个自动增长种子

创建序列：

create sequence 序列名;

使用序列：

查看序列当前的值：

select 序列名.currval from dual;

插入数据时使用：

字段值填写：序列名.nextval

#### 索引

说明：

索引就是对表的某一列或者多列进行排序，下次查询表的时候先查询索引，然后再取数据，设置多个列为索引，只有第一个列是有序的

创建一个索引

create index 索引名on 表名(字段名)

删除索引

drop index 索引名

#### 同义词

说明：有的表名太长不方便使用，可以创建一个同义词使用，类似于别名

创建同义词：

create synonym 同义词名 for 用户名.表名;

删除同义词

drop synonym 同义词

## MYSQL

### 安装

[src\SQL\MYSQL不同版本第一次使用.docx](src/SQL/MYSQL不同版本第一次使用.docx)

### 基本操作

**数据库启动关闭**

service mysqld stop

service mysqld start

**登陆**

mysql -u用户名 -p

**查看数据库**

show databases

**选择使用哪个数据库**

use 数据库名

**查看表、结构、创表语句**

show tables; --查看表

desc 表名 --查看表结构

show create table 表名 --查看创表语句

### 数据库实现

**约束：**

唯一 unique

非空 not null

主键primary key

自带非空和唯一约束，一个表只能有一个主键

外键 foreign key，

参照其他的列，只能存入参照列中存在的数据

检查约束check

mysql没有check需要其他方法实现，枚举实现

**实现：**

[src\SQL\MYSQL数据库实现\admin\_gl.sql](src/SQL/MYSQL数据库实现/admin_gl.sql)

[src\SQL\MYSQL数据库实现\数据字典表格.xlsx](src/SQL/MYSQL数据库实现/数据字典表格.xlsx)

[src\SQL\MYSQL数据库实现\er.png](src/SQL/MYSQL数据库实现/er.png)

### mysql，C语言接口

**代码示例**

[src\SQL\MYSQL\_C语言\_API\main.c](src/SQL/MYSQL_C语言_API/main.c)

**中文文档**

[src\SQL\MYSQL\_C语言\_API\MySQL 中文完全参考手册5.1.chm](src/SQL/MYSQL_C语言_API/MySQL%20中文完全参考手册5.1.chm)

头文件：

#include<mysql/mysql.h>

**初始化mysql结构体**

MYSQL \*mysql\_init(MYSQL \*mysql)

参数：

mysql：传NULL, 如果mysql是NULL指针，该函数将分配、初始化、并返回新对象。

返回值：

成功返回：mysql指针

失败返回：在内存不足的情况下，返回NULL。

**释放mysql结构体资源**

void mysql\_close(MYSQL \*mysql)

**建立连接**

MYSQL \*mysql\_real\_connect(MYSQL \*mysql,

const char \*host,

const char \*user,

const char \*passwd,

const char \*db,

unsigned int port,

const char \*unix\_socket,

unsigned long client\_flag)

参数：

mysql：分配好资源的mysql结构体

host：服务器的IP

user：访问服务器的用户名

passwd：访问服务器的密码

db：需要访问的数据库

port：服务器的端口，默认3306

unix\_socket：传NULL

client\_flag：传0

返回值：

成功返回：mysql指针

失败返回：NULL。

**设置当前连接的字符集**

int mysql\_set\_character\_set(MYSQL \*mysql, char \*csname)

参数：

mysql：mysql指针

csname：要是用的字符集，如 "UTF8"

返回值：

成功返回：0

失败返回：非0值

**执行SQL语句**

int mysql\_query(MYSQL \*mysql, const char \*query)

函数功能：执行以 \0 为结尾的sql语句，不要以 ; 或者 \g为结尾

参数：

mysql：mysql指针

query：sql语句

返回值：

成功返回：0

失败返回：

CR\_COMMANDS\_OUT\_OF\_SYNC

以不恰当的顺序执行了命令。

CR\_SERVER\_GONE\_ERROR

MySQL服务器不可用。

CR\_SERVER\_LOST

在查询过程中，与服务器的连接丢失。

CR\_UNKNOWN\_ERROR

出现未知错误。

注意：

如果sql语句中包含一些二进制数据（可能包含\0字节数据) ，使用mysql\_real\_query来执行

**执行SQL语句后获取结果集合**

MYSQL\_RES \*mysql\_store\_result(MYSQL \*mysql)

参数：

mysql：mysql指针

返回值：

成功返回：具有多个结果的MYSQL\_RES结果集合。

失败返回：NULL，一些宏看API

**释放结果集**

void mysql\_free\_result(MYSQL\_RES \*result)

参数：

result：获取到的结果集

**获取结果集合有几列**

unsigned int mysql\_num\_fields(MYSQL\_RES \*result)

参数：

result：获取到的结果集

返回值：

成功返回：结果集中的列数

失败返回：无

**获取结果集合中的表头**

MYSQL\_FIELD \*mysql\_fetch\_fields(MYSQL\_RES \*result)

参数：

result：获取到的结果集

返回值：

成功返回：对于结果集，返回所有MYSQL\_FIELD结构的数组。每个结构提供了结果集中1列的字段定义。

失败返回：无

注意：

fields[i].name 通过结果集点name的方式获取表头

**获取结果集合中的每一行数据**

MYSQL\_ROW mysql\_fetch\_row(MYSQL\_RES \*result)

参数：  
 result：获取到的结果集

返回值：

下一行的MYSQL\_ROW结构。如果没有更多要检索的行或出现了错误，返回NULL。

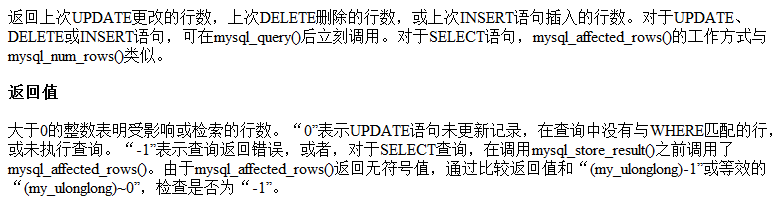
**获取受影响的行**

my\_ulonglong mysql\_affected\_rows(MYSQL \*mysql)

参数：

mysql：mysql指针

返回值：



**C语言打开事物的接口：**

my\_bool mysql\_autocommit(MYSQL \*mysql, my\_bool mode)

函数功能：打开和关闭mysql事物

参数：

mysql：mysql句柄

mode：传0打开事物，传1关闭事物

返回值：

成功返回：0

失败返回：非0值

### mysql中文乱码问题

**1、连接服务器查询编码集**

show variables like 'character%';

**2、查询结果解析**

一、character\_set\_client

主要用来设置客户端使用的字符集。

二、character\_set\_connection

主要用来设置连接数据库时的字符集，如果程序中没有指明连接数据库使用的字符集类型则按照这个字符集设置。

三、character\_set\_database

主要用来设置默认创建数据库的编码格式，如果在创建数据库时没有设置编码格式，就按照这个格式设置。

四、character\_set\_filesystem --不用关系

　　 文件系统的编码格式，把操作系统上的文件名转化成此字符集，即把 character\_set\_client转换character\_set\_filesystem， 默认binary是不做任何转换的。

五、character\_set\_results

　　 数据库给客户端返回时使用的编码格式，如果没有指明，使用服务器默认的编码格式。

六、character\_set\_server

　　 服务器安装时指定的默认编码格式，这个变量建议由系统自己管理，不要人为定义。

七、character\_set\_system --不用关系

　　 数据库系统使用的编码格式，这个值一直是utf8，不需要设置，它是为存储系统元数据的编码格式。

八、character\_sets\_dir --不用关系

　　 这个变量是字符集安装的目录。

**3、字符集装换**

一、mysql Server收到请求时将请求数据从 character\_set\_client 转换为 character\_set\_connection

二、进行内部操作前将请求数据从 character\_set\_connection 转换为内部操作字符集,步骤如下

A. 使用每个数据字段的 CHARACTER SET 设定值；

　　 B. 若上述值不存在，则使用对应数据表的字符集设定值

　　 C. 若上述值不存在，则使用对应数据库的字符集设定值；

　　 D. 若上述值不存在，则使用 character\_set\_server 设定值。

三、最后将操作结果从内部操作字符集转换为 character\_set\_results

**4、LINUX的编码集问题**

统一设置为中文UTF-8编码

shell命令：export LC\_ALL="zh\_CN.UTF-8"

**5、终端软件编码集问题**

如Xshell、CMD、等等

### 其他知识点

#### 事物

**说明：**

mysql默认不打开事物，每执行一条数据都会自动提交相当于没有事物

**mysql客服端查看事物是否打开:**

select @@autocommit; --0为事物打开。1 为事物关闭

**mysql客服端打开事物:**

set autocommit=0; --设置为1，关闭事物

**事物的提交和回滚：**

ROLLBACK

COMMIT

预处理

存储过程