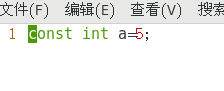
1. Makefile

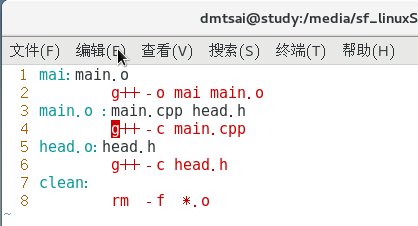
Head.h



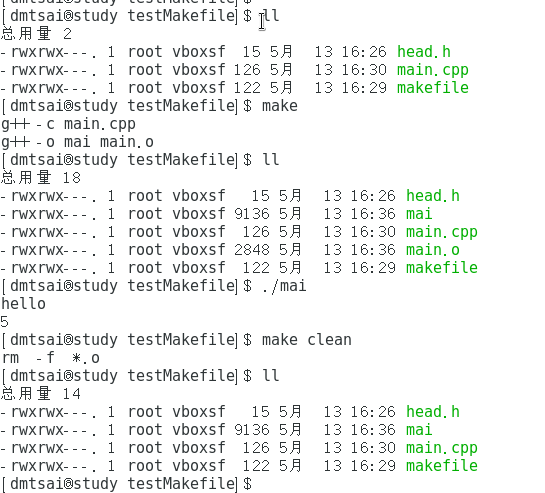
Main.cpp



Makefile



编译



1. gcc 参数

[https://www.cnblogs.com/zjiaxing/p/5557549.html](网页/gcc_g++%20实战之编译的四个过程%20.html)

预处理

g++ -E main.cpp > main.i

转汇编语言 ,生成main.s

g++ -S main.cpp

转机器代码（二进制）,生成main.o

g++ -c main.cpp

链接机器代码，生成可执行程序a.out

g++ main.o –o

g++ main.o –o main

单文件

g++ main.cpp

多文件,生成可执行程序main

g++ main.o head.o –o main

* + 1. .h.gch文件

g++ -c test.h会生成test.h.gch,预编译头文件，能够加快编译速度

* 1. 静态库，动态库

[https://blog.csdn.net/LLZK\_/article/details/55519242](网页/Linux下的静态库与动态库%20.html)

静态库

静态库文件名的命名方式是“libxxx.a”,库名前加”lib”，后缀用”.a”，“xxx”为静态库名。

链接时间： 静态库的代码是在编译过程中被载入程序中。

链接方式：静态库的链接是将整个函数库的所有数据都整合进了目标代码。这样做优点是在编译后的执行程序不在需要外部的函数库支持，因为所使用的函数都已经被编进去了。缺点是，如果所使用的静态库发生更新改变，你的程序必须重新编译。

动态库

动态库的命名方式与静态库类似，前缀相同，为“lib”，后缀变为“.so”。所以为“libmytime.so”

链接时间：动态库在编译的时候并没有被编译进目标代码，而是当你的程序执行到相关函数时才调用该函数库里的相应函数。这样做缺点是因为函数库并没有整合进程序，所以程序的运行环境必须提供相应的库。优点是动态库的改变并不影响你的程序，所以动态函数库升级比较方便。

编译静态库：

编译time.o文件：g++ -c time.cpp

生成静态库文件libmytime.a: ar cr libmytime.a time.o

使用静态库： g++ -o main main.cpp –static –lmytime –L.

-static :表示程序的静态链接

-lmytime：链接静态库mytime

-L. :链接时需指明静态库所存在的路径，‘.’表示当前路径

编译动态库：

编译time.o文件：g++ -fPIC -c time.cpp

生成静态库文件libmytime.a: g++ -shared –fPIC –o libmytime.so time.o

使用静态库： g++ -o main main.cpp –lmytime –L.

1. pthread 编译参数 –lpthread

[https://blog.csdn.net/llzk\_/article/details/55670172](网页/Linux多线程编程（一）%20.html)

[网页\Linux多线程编程（二）.html](网页/Linux多线程编程（二）.html)

[网页\Linux多线程编程（三）.html](网页/Linux多线程编程（三）.html)

pthread 库不是 Linux 系统默认的库，连接时需要使用静态库 libpthread.a，所以在使用pthread\_create()创建线程，以及调用 pthread\_atfork()函数建立fork处理程序时，需要链接该库.

在编译中要加 -lpthread参数  
gcc thread.c -o thread -lpthread  
thread.c为你些的源文件，不要忘了加上头文件#include.

eclipse  
解决方法如下：  
Project->Properties->C/C++ Build->Settings->GCC C++ Linker->Libraries  
在Libraries(-l)中添加pthread即可  
在Libraries search path(-L)中添加crypto即可

* 1. pthread\_create函数，创建线程

int pthread\_create(pthread\_t \*tid, const pthread\_attr\_t \*attr, void \*(\*func)(void \*), void \*arg);

pthread\_t \*tid：一个进程内的各个线程是由线程ID标识的，如果新线程创建成功，返回tid指针。

const pthread\_attr\_t \*attr：每个线程有多个属性，包括优先级、初始栈大小、是否是一个守护线程等等。

void \*(\*func)(void \*)：线程启动函数，线程从调用这个函数开始，或显示结束（调用pthread\_exit()），或隐式结束（让该函数返回）。

void \*arg：线程执行func函数的传递参数

* 1. pthread\_join函数，等待一个线程终止

int pthread\_join(pthread\_t \*tid, void \*\*status);

void \*\*status：二级指针，如果status指针非空，那么所等待线程的返回值将存放在status指向的位置。

pthread\_join作用：

用于等待其他线程结束：当调用 pthread\_join() 时，当前线程会处于阻塞状态，直到被调用的线程结束后，当前线程才会重新开始执行。

对线程的资源进行回收：如果一个线程是非分离的（默认情况下创建的线程都是非分离）并且没有对该线程使用 pthread\_join() 的话，该线程结束后并不会释放其内存空间，这会导致该线程变成了“僵尸线程”。

例子：

void\* fun(void\*)

{

static int num = 5;

return (void\*)&num;

}

void\* args;

pthread\_join(thread, &args);

*cout* << \*(int\*)h << *endl*;

* 1. 互斥锁mutex

互斥锁是最基本的同步方式，它用来保护一个“临界区”，保证任何时刻只由一个线程在执行其中的代码。这个“临界区”通常是线程的共享数据。

下面三个函数给一个互斥锁上锁和解锁：

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

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

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

　　假设线程2要给已经被线程1锁住的互斥锁（mutex）上锁（即执行pthread\_mutex\_lock(mutex)），那么它将一直阻塞直到到线程1解锁为止（即释放mutex）。

如果互斥锁变量时静态分配的，通常初始化为常值PTHREAD\_MUTEX\_INITIALIZER，如果互斥锁是动态分配的，那么在运行时调用pthread\_mutex\_init函数来初始化。

* 1. 条件变量cond

互斥锁用于上锁，而条件变量则用于等待，通常它都会跟互斥锁一起使用。

int pthread\_cond\_wait(pthread\_cond\_t \*cptr,pthread\_mutex\_t \*mptr);

先锁住mptr，再调用pthread\_cond\_wait；

调用pthread\_cond\_wait时先阻塞本线程，然后释放mptr，

唤醒时先竞争mptr，若竞争到锁，则加锁返回，否则下次竞争mptr；

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

　　通常pthread\_cond\_signal只唤醒等待在相应条件变量上的一个线程，若有多个线程需要被唤醒呢，这就要使用下面的函数了：

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

1. thread

<https://segmentfault.com/a/1190000016186741>

<https://www.cnblogs.com/KillerAery/p/9574410.html>

* 1. thread构造函数

std::thread类的构造函数是使用可变参数模板实现的，第一个参数是线程的入口函数，而后面的若干个参数是该函数的参数。

第一参数的类型并不是c语言中的函数指针（c语言传递函数都是使用函数指针），在c++11中，增加了可调用对象(Callable Objects)的概念，总的来说，可调用对象可以是以下几种情况：

函数指针

重载了operator()运算符的类对象，即仿函数

lambda表达式（匿名函数）

std::function

传值还是传引用？传值。int a;传引用需要用std::ref(a)，不能使用&a；

std::thread类，内部也有若干个变量，当使用构造函数创建对象的时候，是将参数先赋值给这些变量，所以这些变量只是个副本，然后在线程启动并调用线程入口函数时，传递的参数只是这些副本，所以内部怎么操作都是改变副本，而不影响外面的变量。

线程对象之间是不能复制的，只能移动，移动的意思是，将线程的所有权在std::thread实例间进行转移。

void some\_function();

void some\_other\_function();

std::thread t1(some\_function);

// std::thread t2 = t1; // 编译错误

std::thread t2 = std::move(t1); //只能移动 t1内部已经没有线程了

t1 = std::thread(some\_other\_function); // 临时对象赋值 默认就是移动操作

std::thread t3;

t3 = std::move(t2); // t2内部已经没有线程了

t1 = std::move(t3); // 程序将会终止，因为t1内部已经有一个线程在管理了

* 1. 常用方法

#include<thread>

std::thread t(func); //启动新线程

t.join() //阻塞当前线程，等待新线程完成

t.joinable()

t.detach() //分离线程，继续执行

#include<mutex>

std::mutex mu;

mu.lock();

mu.unlock();

析构时解锁：

std::lock\_guard<std::mutex> guard(mu);

std::lock\_guard<std::mutex> guard2(mu,std::adopt\_lock);//初始构造不上锁

std::unique\_lock<std::mutex> guard3(mu);//可临时解锁

std::unique\_lock<std::mutex> guard4(mu , std::defer\_lock); //初始构造不上锁

guard3.lock();

guard3.unlock();

同时上几个锁：

std::lock(mu1,mu2);

#include <condition\_variable>

std::condition\_variable cond;

cond.wait(mu);

cond.wait(mu,if); //if为判断条件

cond.notify\_one();

cond.notify\_all();

* 1. 线程返回值 std::future

#include<future>

线程1 future.get()阻塞

线程2 promise. set\_value()唤醒future

void initiazer(std::promise<int> &promiseObj){

std::cout << "Inside thread: " << std::this\_thread::get\_id() << std::endl;

std::this\_thread::sleep\_for(std::chrono::seconds(1));

promiseObj.set\_value(35);

}

int main(){

std::promise<int> promiseObj;

std::future<int> futureObj = promiseObj.get\_future();

std::thread th(initiazer, std::ref(promiseObj));

std::cout << futureObj.get() << std::endl;

th.join();

return 0;

}

1. 信号

[https://blog.csdn.net/llzk\_/article/details/56679445](网页/Linux信号（Signal）.html)

#include<stdio.h>

#include<signal.h>

#include<unistd.h>

void sig\_run(int signo)

{

//DO NOTOING

}

int mysleep(int secs)

{

struct sigaction sig,osig;

//设置sigaction的属性

sig.sa\_handler = sig\_run;

sigemptyset(&sig.sa\_mask);

sig.sa\_flags = 0;

sigaction(SIGALRM,&sig,&osig);

alarm(secs);

pause();

int ret = alarm(0);

sigaction(SIGALRM,&osig,NULL);

return ret;

}

int main()

{

while(1)

{

mysleep(1);

printf("hello everyone!\n");

}

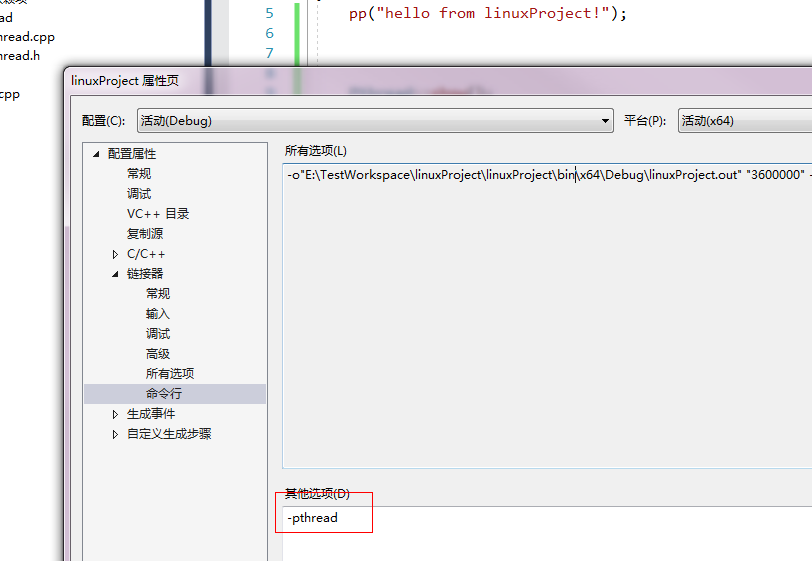
return 0;

}

1. vs2017 开发linux

[https://www.jianshu.com/p/1aa5a3761074](网页/VS2017创建并调试LINUX程序%20.html)

加编译参数：



1. lambda

<https://www.cnblogs.com/DswCnblog/p/5629165.html>

格式：

[capture list] (params list) mutable exception-> return type { function body }

各项具体含义如下：

capture list：捕获外部变量列表

params list：形参列表

mutable指示符：用来说用是否可以修改捕获的变量，默认不可改变值捕获

exception：异常设定

return type：返回类型

function body：函数体

捕获列表和函数体不可省略。

简单示例：

升序排序

vector<int> kk = { 5,6,9,2 };

sort(kk.begin(), kk.end(),[](int a,int b){return a<b;});

值捕获：

声明，创建lambda时拷贝，之后更改无效

引用捕获：sort(kk.begin(), kk.end(),[&kk](int a,int b){return a<b;});

隐式值捕获：[=](int a,int b){return a<b;}

隐式引用捕获：[&](int a,int b){return a<b;}

1. 数字头文件limits.h

#include <limits.h>:

INT\_MAX int最大值

#include <float.h>

1. 多线程读写文件

<https://blog.csdn.net/daiyudong2020/article/details/52015817>

一个进程打开一个文件，有一个文件描述符，记录当前文件偏移量

两个进程或线程打开同一文件，有两个文件描述符，有两个文件偏移量，可并行读写文件

多进程（线程）共享一个文件描述符时，只有一个文件偏移量，不可并行读写文件

1. qt 元对象 反射

int methodIndex = this->metaObject()->indexOfMethod("exitSystem()");

   QMetaMethod method = this->metaObject()->method(methodIndex);

   method.invoke(this, Qt::QueuedConnection);