<https://www.cnblogs.com/wujing-hubei/p/5222013.html>

**多线程**

使用多线程好处：

一、通过为每种事件类型的处理单独分配线程，可以简化处理**异步事件**的代码，线程处理事件可以采用**同步编程模式**，启闭异步编程模式简单

二、方便的通信和数据交换

     由于进程之间具有独立的数据空间，多进程必须使用操作系统提供的复杂机制才能实现内存和文件描述符的共享，导致上下文切换的开销比较大。而线程之间共享进程的所有资源，所以，多线程可以访问相同的存储空间和文件描述符。

三、对一些不具有互斥型的问题，可以将其分解为从而改善程序的吞吐量。对于进程，在完成多个任务时，实际上需要将任务串行化。对于多线程，相互独立而且想不不依赖的任务可以交叉运行，只需要为每个任务分配一个线程即可。

四、交互的程序可以通过使用多线程实现响应时间的改善，多线程可以把程序中用于处理用户输入输出的部分与其他部分分开。

线程包含了进程内执行环境必须的信息，包括进程ID，一组寄存器值，栈，调度优先级和策略，信号屏蔽字，errno变量以及线程私有数据。

线程ID使用pthread\_t数据类型来表示，类似于一种结构体，可以使用 pthread\_self 函数和 pthread\_equal 函数来通过线程ID识别线程。

原型：

#include <pthread.h>

int pthread\_eaual(pthread\_t tid1,pthread\_t tid2);   //判断两个线程ID是否相等

                      相等返回非0，不相等返回0

#include <pthread.h>

pthread\_t pthread\_self(void);   //获取自身线程ID

    返回调用线程的ID

**线程创建**

#include <pthread.h>

int pthread\_creat(pthread\_t \*restrict tidp,const pthread\_attr\_t \*attr,void\*(\*start\_rtn)(void),void \*restrict arg);   //创建线程

                                              成功返回0，否则返回错误编号

tidp 指向的内存单元被设置为新创建的现成的线程ID

attr 用于定制线程的线程属性，设置为NULL时，则使用默认属性

新创建的函数从strat\_rtn函数的地址开始运行，该函数只有一个无类型指针参数arg.**若要想向strat\_rtn传递不知一个参数，可以将多个参数放在一个结构体中，然后把结构体的地址作为arg参数传入。**

**线程终止**

 如果进程中的任一线程调用了exit，\_Exit或\_exit，则整个进程会终止。同样，如果信号的默认动作是终止进程，那么，把信号发送到进程会终止整个进程。

单个进程的退出方式有三种：

1、线程只是从启动的例程中退出，返回值是线程的终止码；

2,、线程可以被统一进程中的其他线程取消；

3、线程调用pthread\_exit；

#include<pthread.h>

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

线程通过调用pthread\_exit函数终止执行，就如同进程在结束时调用exit函数一样。这个函数的作用是，终止调用它的线程并返回一个指向某个对象的[指针](http://baike.baidu.com/view/159417.htm)。

rval\_ptr是一个无类型的指针，与传递给启动例程的单个参数类似，可以通过进程中的其他线程调用pthread\_join函数访问这个指针。

#include<pthread.h>

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

               成功返回0；否则返回错误编号

pthread\_join()函数，以阻塞的方式等待thread指定的线程调用pthread\_exit、冲启动例程中返回或者被取消。当函数返回时，被等待线程的资源被收回。

如果线程已经结束，那么该函数会立即返回。并且thread指定的线程必须是joinable的。

 thread: 线程[标识符](http://baike.baidu.com/view/390932.htm)，即线程ID，标识唯一线程。rval\_ptr: 用户定义的指针，用来存储被等待线程的返回值。

如果线程是从启动例程中返回，

rval\_ptr将包含返回码，如果线程被取消，由rval\_ptr指向的内存单元被置为PTHREAD\_CANCELED。

**线程同步**

线程同步机制包括**互斥**，**读写锁**以及**条件变量**

**互斥**

可以把互斥变量之置为常量PTHREAD\_MUTEX\_INITIALIZER(针对静态分配的互斥量)，或调用pthread\_mutex\_init函数进行初始化。如果动态的分配互斥量（如调用malloc函数），那么在释放内存前需要调用pthread\_mutex\_destory。

#include<pthread.h>

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

int  pthread\_mutex\_destory(pthread\_mutex\_t  mutex);

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

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

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

                    成功返回0，否则返回错误编号

如果线程不希望被堵塞，可以调用pthread\_mutex\_trylock尝试对互斥量进行加锁。如果调用pthread\_mutex\_trylock时互斥量处于未锁住状态，则pthread\_mutex\_trylock将互斥量锁住，不会出现阻塞并返回0，否则失败并返回EBUSY。

[**读写锁**](http://blog.csdn.net/dsg333/article/details/22057063)

初始化

#include<pthread.h>

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

int pthread\_rwlock\_destory(pthread\_rwlock\_t  \*restrict rwlock);   //销毁读写锁，

                                      成功返回0，错误返回错误编号；

锁定读写锁与解锁

#include<pthread.h>

int pthread\_rwlock\_rdlock(pthread\_rwlock\_t  rwlock);   //在读模式下锁定读写锁

int pthread\_rwlock\_wrlock(pthread\_rwlock\_t  rwlock);   //在写模式下锁定读写锁

int pthread\_rwlock\_unlock(pthread\_rwlock\_t  rwlock);   //解锁

                                      成功返回0，错误返回错误编号；

**1、初始化读写锁**

　　int pthread\_rwlock\_init(pthread\_rwlock\_t \*rwlock, const pthread\_rwlockattr\_t \*attr); 与 pthread\_rwlock\_t rwlock = PTHREAD\_RWLOCK\_INITIALIZER;作用相同

　　如果 attr 为 NULL，则使用缺省的读写锁属性，其作用与传递缺省读写锁属性对象的地址相同。初始化读写锁之后，该锁可以使用任意次数，而无需重新初始化。成功初始化之后，读写锁的状态会变为已初始化和未锁定。如果调用pthread\_rwlock\_init() 来指定已初始化的读写锁，则结果是不确定的。如果读写锁在使用之前未初始化，则结果是不确定的。对于 Solaris 线程，请参见rwlock\_init语法。

　　如果缺省的读写锁属性适用，则 PTHREAD\_RWLOCK\_INITIALIZER 宏可初始化以静态方式分配的读写锁，其作用与通过调用pthread\_rwlock\_init() 并将参数attr 指定为 NULL 进行动态初始化等效，区别在于不会执行错误检查。

pthread\_rwlock\_init 返回值

　　如果成功，pthread\_rwlock\_init() 会返回零。否则，将返回用于指明错误的错误号。

　　如果 pthread\_rwlock\_init() 失败，将不会初始化rwlock，并且rwlock 的内容是不确定的。

**2、获取读写锁中的读锁**

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

　　如果写入器未持有读锁，并且没有任何写入器基于该锁阻塞，则调用线程会获取读锁。如果写入器未持有读锁，但有多个写入器正在等待该锁时，调用线程是否能获取该锁是不确定的。如果某个写入器持有读锁，则调用线程无法获取该锁。如果调用线程未获取读锁，则它将阻塞。调用线程必须获取该锁之后，才能从pthread\_rwlock\_rdlock() 返回。如果在进行调用时，调用线程持有rwlock 中的写锁，则结果是不确定的。

　　为避免写入器资源匮乏，允许在多个实现中使写入器的优先级高于读取器。例如，Solaris 线程实现中写入器的优先级高于读取器。 请参见rw\_rdlock 语法。

　　一个线程可以在 rwlock 中持有多个并发的读锁，该线程可以成功调用pthread\_rwlock\_rdlock()n 次。该线程必须调用pthread\_rwlock\_unlock()n 次才能执行匹配的解除锁定操作。

　　如果针对未初始化的读写锁调用 pthread\_rwlock\_rdlock()，则结果是不确定的。

　　线程信号处理程序可以处理传送给等待读写锁的线程的信号。从信号处理程序返回后，线程将继续等待读写锁以执行读取，就好像线程未中断一样。

**pthread\_rwlock\_rdlock 返回值**

如果成功，pthread\_rwlock\_rdlock() 会返回零。否则，将返回用于指明错误的错误号。

**3、读取非阻塞读写锁中的锁**

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

**pthread\_rwlock\_tryrdlock 返回值**

如果获取了用于在 rwlock 所引用的读写锁对象中执行读取的锁，则pthread\_rwlock\_tryrdlock() 将返回零。如果没有获取该锁，则返回用于指明错误的错误号--EBUSY。

**4、写入读写锁中的锁**

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

　　如果没有其他读取器线程或写入器线程持有读写锁 rwlock，则调用线程将获取写锁。否则，调用线程将阻塞。调用线程必须获取该锁之后，才能从pthread\_rwlock\_wrlock() 调用返回。如果在进行调用时，调用线程持有读写锁（读锁或写锁），则结果是不确定的。

　　为避免写入器资源匮乏，允许在多个实现中使写入器的优先级高于读取器。（例如，Solaris 线程实现允许写入器的优先级高于读取器。请参见rw\_wrlock 语法。）

　　如果针对未初始化的读写锁调用 pthread\_rwlock\_wrlock()，则结果是不确定的。

　　线程信号处理程序可以处理传送给等待读写锁以执行写入的线程的信号。从信号处理程序返回后，线程将继续等待读写锁以执行写入，就好像线程未中断一样。

**pthread\_rwlock\_wrlock 返回值**

如果获取了用于在 rwlock 所引用的读写锁对象中执行写入的锁，则pthread\_rwlock\_rwlock() 将返回零。如果没有获取该锁，则返回用于指明错误的错误号

**5、写入非阻塞读写锁中的锁**

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

　　如果针对未初始化的读写锁调用 pthread\_rwlock\_trywrlock()，则结果是不确定的。

　　线程信号处理程序可以处理传送给等待读写锁以执行写入的线程的信号。从信号处理程序返回后，线程将继续等待读写锁以执行写入，就好像线程未中断一样。

**pthread\_rwlock\_trywrlock 返回值**

如果获取了用于在 rwlock 引用的读写锁对象中执行写入的锁，则pthread\_rwlock\_trywrlock() 将返回零。否则，将返回用于指明错误的错误号--EBUSY。

**6、解除锁定读写锁**

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

　　如果调用线程未持有读写锁 rwlock，则结果是不确定的。对于 Solaris 线程，请参见rw\_unlock 语法。

　　如果通过调用 pthread\_rwlock\_unlock() 来释放读写锁对象中的读锁，并且其他读锁当前由该锁对象持有，则该对象会保持读取锁定状态。如果pthread\_rwlock\_unlock() 释放了调用线程在该读写锁对象中的最后一个读锁，则调用线程不再是该对象的属主。如果pthread\_rwlock\_unlock() 释放了该读写锁对象的最后一个读锁，则该读写锁对象将处于无属主、解除锁定状态。

　　如果通过调用 pthread\_rwlock\_unlock() 释放了该读写锁对象的最后一个写锁，则该读写锁对象将处于无属主、解除锁定状态。

　　如果 pthread\_rwlock\_unlock() 解除锁定该读写锁对象，并且多个线程正在等待获取该对象以执行写入，则通过调度策略可确定获取该对象以执行写入的线程。如果多个线程正在等待获取读写锁对象以执行读取，则通过调度策略可确定等待线程获取该对象以执行写入的顺序。如果多个线程基于rwlock 中的读锁和写锁阻塞，则无法确定读取器和写入器谁先获得该锁。

　　如果针对未初始化的读写锁调用 pthread\_rwlock\_unlock()，则结果是不确定的。

**7、pthread\_rwlock\_destroy**

int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock);与pthread\_rwlock\_t rwlock = PTHREAD\_RWLOCK\_INITIALIZER;相同

　　在再次调用 pthread\_rwlock\_init() 重新初始化该锁之前，使用该锁所产生的影响是不确定的。实现可能会导致pthread\_rwlock\_destroy() 将rwlock 所引用的对象设置为无效值。如果在任意线程持有 rwlock 时调用 pthread\_rwlock\_destroy()，则结果是不确定的。尝试销毁未初始化的读写锁会产生不确定的行为。已销毁的读写锁对象可以使用pthread\_rwlock\_init() 来重新初始化。销毁读写锁对象之后，如果以其他方式引用该对象，则结果是不确定的。对于 Solaris 线程，请参见rwlock\_destroy语法。

**pthread\_rwlock\_destroy 返回值**

如果成功，pthread\_rwlock\_destroy() 会返回零。否则，将返回用于指明错误的错误号--EINVAL。

**条件变量**

和互斥量一样，可以把条件变量之置为常量PTHREAD\_COND\_INITIALIZER(针对静态分配的条件变量)，或调用pthread\_cond\_init函数进行初始化。在释放内存前需要调用pthread\_cond\_destory函数对变量进行去除初始化。

#include<pthread.h>

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

int  pthread\_cond\_destory(pthread\_cond\_t cond);

                                       成功返回0，否则返回错误编号

条件变量是与条件测试一起使用的，通常线程会对一个条件进行测试，如果条件不满足调用条件等待函数就等待条件满足。

#include<pthread.h>

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

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

                                       成功返回0，否则返回错误编号

cond是一个指向条件变量的指针，mutex是一个指向互斥量的指针，线程在调用前应该拥有这个互斥量。传递给pthread\_cond\_wait的互斥量对条件进行保护，调用者吧锁住的互斥量传递给函数，函数把调用线程防盗等待条件的线程列表上，然后互斥量解锁，pthread\_cond\_wait返回时，互斥量再次被锁住。

pthread\_cond\_timedwait与pthread\_cond\_wait工作方式相似，只是多了一个timeout。timeout是一个**绝对值**而不是相对值。其为需要等待的时间加上当前时间，即未来某一时刻。

timeout通过timespec结构指定：

struct timespec{

          time\_t    tv\_sec;

          time\_t    tv\_nsec;

};

可以使用gettimeofday获取用timeval表示的当前时间，然后把等待时间转化成timespec结构。

可以使用一下函数得到timeout值得绝对时间

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

{

   struct timeval now;

   gettimeofday(&now); //获取当前时间

   tsp->tv\_sec=now.tv\_sec;

   tsp->tv\_nsec=now.tv\_usec\*1000;   //usec转换成nsec

   tsp->tv\_sec+=minutes\*60;

}

pthread\_cond\_signal和pthread\_cond\_broadcast两个函数用来通知线程条件已经满足。

pthread\_cond\_signal在条件满足后唤醒等待条件的某一个线程，而pthread\_cond\_broadcast将唤醒所有线程。

#include<pthread.h>

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

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

                                       成功返回0，否则返回错误编号

**注意：**一定要在条件状态改变以后再给线程发信号