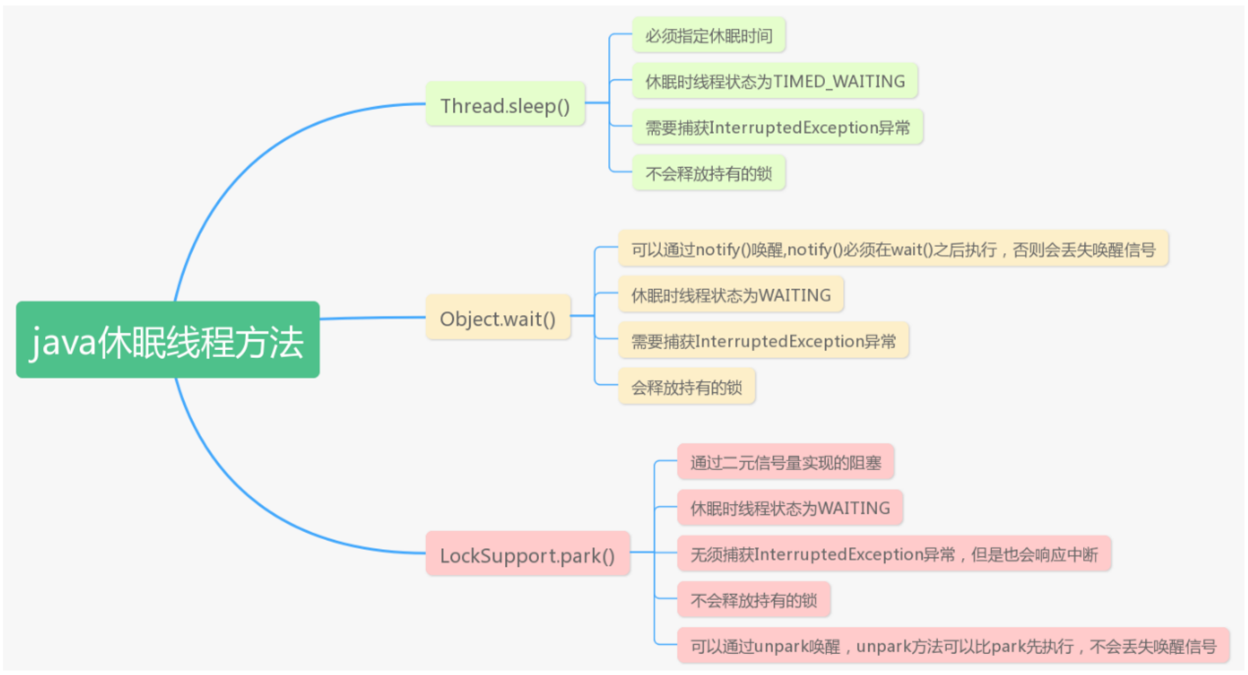
主讲老师：Fox


java休眠线程方法
Thread.sleep()
必须指定休眠时间
休眠时线程状态为TIMED_WAITING
需要捕获InterruptedException异常
不会释放持有的锁
Object.wait()
可以通过notify()唤醒,notify()必须在wait()之后执行，否则会丢失唤醒信号
休眠时线程状态为WAITING
需要捕获InterruptedException异常
会释放持有的锁
LockSupport.park()
通过二元信号量实现的阻塞
休眠时线程状态为WAITING
无须捕获InterruptedException异常，但是也会响应中断
不会释放持有的锁
可以通过unpark唤醒，unpark方法可以比park先执行，不会丢失唤醒信号

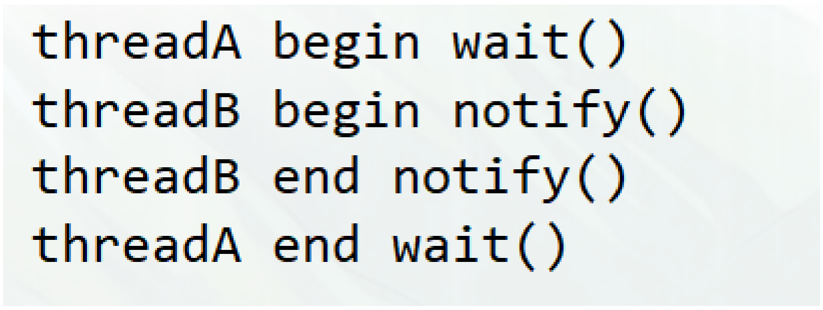
使用wait/notify阻塞唤醒线程

```java
public class WaitTest {

 public void testWaitMethod(Object lock) {
 try {
 synchronized (lock) {
 System.out.println(Thread.currentThread().getName()+" begin wait()");
 // wait() 释放锁资源
 Thread.sleep(5000);

 lock.wait();
 System.out.println(Thread.currentThread().getName()+" end wait()");
 }
 } catch (Exception e) {
 e.printStackTrace();
 }
 }

 public void testNotifyMethod(Object lock) {
 synchronized (lock) {
 System.out.println(Thread.currentThread().getName()+" begin notify()");
```

结果

```

lock.notify();

System.out.println(Thread.currentThread().getName()+" end notify()");
}
}

public static void main(String[] args) {
Object lock = new Object();
WaitTest test = new WaitTest();

Thread t1 = new Thread(new Runnable() {
@Override
public void run() {
test.testWaitMethod(lock);
}
},"threadA");
t1.start();

Thread t2 = new Thread(new Runnable() {
@Override
public void run() {
test.testNotifyMethod(lock);
}
},"threadB");

t2.start();
try {
Thread.sleep(500000);
} catch (InterruptedException e) {
e.printStackTrace();
}

}
}
```

结果



```
threadA begin wait()
threadB begin notify()
threadB end notify()
threadA end wait()
```

使用wait，notify来实现等待唤醒功能至少有两个缺点：

- wait和notify都是Object中的方法,在调用这两个方法前必须先获得锁对象，这限制了其使用场合:只能在同步代码块中。
- 当对象的等待队列中有多个线程时，notify只能随机选择一个线程唤醒，无法唤醒指定的线程

使用LockSupport的话，我们可以在任何场合使线程阻塞，同时也可以指定要唤醒的线程，相当的方便。

使用park/unpark阻塞唤醒线程

LockSupport是JDK中用来实现线程阻塞和唤醒的工具。使用它可以在任何场合使线程阻塞，可以指定任何线程进行唤醒，并且不用担心阻塞和唤醒操作的顺序，但要注意连续多次唤醒的效果和一次唤醒是一样的。JDK并发包下的锁和其他同步工具的底层实现中大量使用了LockSupport进行线程的阻塞和唤醒，掌握它的用法和原理可以让我们更好的理解锁和其它同步工具的底层实现。

```java
public class LockSupportTest {

 public static void main(String[] args) {
 Thread parkThread = new Thread(new ParkThread());
 parkThread.start();

 System.out.println("唤醒parkThread");
 LockSupport.unpark(parkThread);
 }

  static class ParkThread implements Runnable{

  @Override
  public void run() {
  System.out.println("ParkThread开始执行");
  LockSupport.park();
System.out.println("ParkThread执行完成");
 }
 }
}
```

LockSupport阻塞和唤醒线程原理

每个线程都有Parker实例

```
class Parker : public os::PlatformParker {
private:
 volatile int _counter ;
 ...
public:
 void park(bool isAbsolute, jlong time);
 void unpark();
 ...
}
class PlatformParker : public CHeapObj<mtInternal> {
 protected:
 pthread_mutex_t _mutex [1] ;
 pthread_cond_t _cond [1] ;
 ...
}
```

LockSupport就是通过控制变量`_counter`来对线程阻塞唤醒进行控制的。原理有点类似于信号量机制。

- 当调用`park()`方法时，会将_counter置为0，同时判断前值，小于1说明前面被`unpark`过,则直接退出，否则将使该线程阻塞。
- 当调用`unpark()`方法时，会将_counter置为1，同时判断前值，小于1会进行线程唤醒，否则直接退出。

形象的理解，线程阻塞需要消耗凭证(permit)，这个凭证最多只有1个。当调用park方法时，如果有凭证，则会直接消耗掉这个凭证然后正常退出；但是如果没有凭证，就必须阻塞等待凭证可用；而unpark则相反，它会增加一个凭证，但凭证最多只能有1个。

- 为什么可以先唤醒线程后阻塞线程?

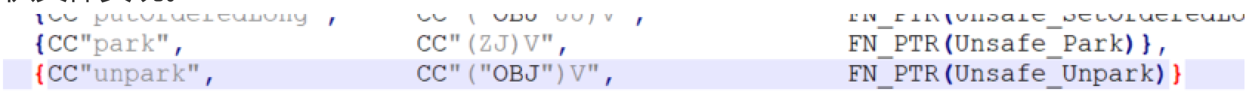
因为unpark获得了一个凭证,之后调用park因为有凭证消费，故不会阻塞。

- 为什么唤醒两次后阻塞两次会阻塞线程。

因为凭证的数量最多为1，连续调用两次unpark和调用一次unpark效果一样，只会增加一个凭证；而调用两次park却需要消费两个凭证。

park&unpark源码分析

在 Hotspot 源码中，unsafe.cpp 文件专门用于为 Java Unsafe 类中的各种 native 方法提供具体实现。



```
{CC"park",          CC"(ZJ)V",          FN_PTR(Unsafe_Park)},
{CC"unpark",        CC"("OBJ")V",       FN_PTR(Unsafe_Unpark)}
```

park和unpark的实现代码如下

```
UNSAFE_ENTRY(void, Unsafe_Park(JNIEnv *env, jobject unsafe, jboolean isAbsolute, jlong time))
  UnsafeWrapper("Unsafe_Park");
  EventThreadPark event;
#ifndef USDT2
  HS_DTRACE_PROBE3(hotspot, thread__park__begin, thread->parker(), (int) isAbsolute, time);
#else /* USDT2 */
  HOTSPOT_THREAD_PARK_BEGIN(
  (uintptr_t) thread->parker(), (int) isAbsolute, time);
#endif /* USDT2 */
  JavaThreadParkedState jtps(thread, time != 0);
  thread->parker()->park(isAbsolute != 0, time);
#ifndef USDT2
  HS_DTRACE_PROBE1(hotspot, thread__park__end, thread->parker());
#else /* USDT2 */
  HOTSPOT_THREAD_PARK_END(
  (uintptr_t) thread->parker());
#endif /* USDT2 */
  if (event.should_commit()) {
  oop obj = thread->current_park_blocker();
  event.set_klass((obj != NULL) ? obj->klass() : NULL);
  event.set_timeout(time);
  event.set_address((obj != NULL) ? (TYPE_ADDRESS) cast_from_oop<uintptr_t>(obj) : 0);
  event.commit();
  }
UNSAFE_END

UNSAFE_ENTRY(void, Unsafe_Unpark(JNIEnv *env, jobject unsafe, jobject jthread))
```

```
UnsafeWrapper("Unsafe_Unpark");
Parker* p = NULL;
if (jthread != NULL) {
oop java_thread = JNIHandles::resolve_non_null(jthread);
if (java_thread != NULL) {
jlong lp = java_lang_Thread::park_event(java_thread);
if (lp != 0) {
// This cast is OK even though the jlong might have been read
// non-atomically on 32bit systems, since there, one word will
// always be zero anyway and the value set is always the same
p = (Parker*)addr_from_java(lp);
} else {
// Grab lock if apparently null or using older version of library
MutexLocker mu(Threads_lock);
java_thread = JNIHandles::resolve_non_null(jthread);
if (java_thread != NULL) {
JavaThread* thr = java_lang_Thread::thread(java_thread);
if (thr != NULL) {
p = thr->parker();
if (p != NULL) { // Bind to Java thread for next time.
java_lang_Thread::set_park_event(java_thread, addr_to_java(p));
}
}
}
}
}
}
if (p != NULL) {
#ifndef USDT2
HS_DTRACE_PROBE1(hotspot, thread__unpark, p);
#else /* USDT2 */
HOTSPOT_THREAD_UNPARK(
(uintptr_t) p);
#endif /* USDT2 */
p->unpark();
}
UNSAFE_END
```

每个线程对象都有一个 Parker 实例 （源码文件：thread.hpp）

```
// JSR166 per-thread parker
private:
Parker* _parker;
public:
Parker* parker() { return _parker; }
```

parker 类的定义如下 （源码文件 park.hpp）

- Parker 类继承 os::PlatformParker。 针对不同操作系统进行适配
- 有一个 _counter 属性，_counter字段 > 0时，可以通行，即park方法会直接返回，同时_counter会被赋值为0，否则，当前线程陷入条件等待。unpark方法可以将_counter置为1，并且唤醒当前等待的线程。
- 提供了公开的 park 和 unpark 方法

```
// Parker 类继承 os::PlatformParker。 针对不同操作系统进行适配，比如linux在os_linux.cpp中实现了park，unpark方法
class Parker : public os::PlatformParker {
private:
 volatile int _counter ; // 通行的许可证，当 _count > 0 时park方法直接返回。unpark会将_counter置为1
 Parker * FreeNext ;
 JavaThread * AssociatedWith ; // Current association

public:
 Parker() : PlatformParker() {
  _counter = 0 ;
  FreeNext = NULL ;
  AssociatedWith = NULL ;
  }
 protected:
  ~Parker() { ShouldNotReachHere(); }
 public:
  // For simplicity of interface with Java, all forms of park (indefinite,
  // relative, and absolute) are multiplexed into one call.
  void park(bool isAbsolute, jlong time);
  void unpark();

  // Lifecycle operators
  static Parker * Allocate (JavaThread * t) ;
  static void Release (Parker * e) ;
 private:
  static Parker * volatile FreeList ;
  static volatile int ListLock ;

};
```

park源码分析

具体park方法实现在父类PlatformParker中。 我们可以看下linux系统下Packer的park方法实现过程 （源码文件 os_linux.cpp）

1. 判断是否需要阻塞等待，如果已经是 _counter >0, 不需要等待，将 _counter = 0 ， 返回
2. 如果 1 不成立，构造当前线程的 ThreadBlockInVM ， 检查 _counter > 0 是否成立，成立则将 _counter 设置为 0， unlock mutex 返回；
3. 如果 2 不成立，当前线程需要根据时间进行不同的条件等待，如果条件满足正确返回，则将 _counter 设置为0， unlock mutex ， park 调用成功。

```
void Parker::park(bool isAbsolute, jlong time) {
 // Ideally we'd do something useful while spinning, such
 // as calling unpackTime().

 // Optional fast-path check:
 // Return immediately if a permit is available.
 // We depend on Atomic::xchg() having full barrier semantics
 // since we are doing a lock-free update to _counter.
 // 使用 xchg 指令修改为0,返回原值，原值大于0说明有通行证，直接返回
  if (Atomic::xchg(0, &_counter) > 0) return;

  Thread* thread = Thread::current();
  assert(thread->is_Java_thread(), "Must be JavaThread");
  JavaThread *jt = (JavaThread *)thread;

  // Optional optimization -- avoid state transitions if there's an interrupt pending.
  // Check interrupt before trying to wait
  // 如果当前线程的中断标志位为true，直接返回，注意：不会清除中断标志位
  if (Thread::is_interrupted(thread, false)) {
  return;
  }

  // Next, demultiplex/decode time arguments
  timespec absTime;
  if (time < 0 || (isAbsolute && time == 0) ) { // don't wait at all
  return;
  }
  if (time > 0) {
  unpackTime(&absTime, isAbsolute, time);
  }


  // Enter safepoint region
  // Beware of deadlocks such as 6317397.
  // The per-thread Parker:: mutex is a classic leaf-lock.
  // In particular a thread must never block on the Threads_lock while
  // holding the Parker:: mutex. If safepoints are pending both the
  // the ThreadBlockInVM() CTOR and DTOR may grab Threads_lock.
  // 构造当前线程的 ThreadBlockInVM，为了防止死锁等特殊场景
  ThreadBlockInVM tbivm(jt);

  // Don't wait if cannot get lock since interference arises from
  // unblocking. Also. check interrupt before trying wait
  // 再次判断线程是否存在中断状态，如果存在，尝试获取互斥锁，如果获取失败，直接返回
  if (Thread::is_interrupted(thread, false) || pthread_mutex_trylock(_mutex) != 0) {
  return;
  }

  int status ;
  // 如果_counter > 0, 不需要等待
  if (_counter > 0) { // no wait needed
  _counter = 0; // _counter重置为0
  status = pthread_mutex_unlock(_mutex); //释放互斥锁
  assert (status == 0, "invariant") ;
  // Paranoia to ensure our locked and lock-free paths interact
  // correctly with each other and Java-level accesses.
  // 插入写屏障
  OrderAccess::fence();
  return;
  }

 #ifdef ASSERT
  // Don't catch signals while blocked; let the running threads have the signals.
  // (This allows a debugger to break into the running thread.)
```

```
sigset_t oldsigs;
sigset_t* allowdebug_blocked = os::Linux::allowdebug_blocked_signals();
pthread_sigmask(SIG_BLOCK, allowdebug_blocked, &oldsigs);
#endif

OSThreadWaitState osts(thread->osthread(), false /* not Object.wait() */);
jt->set_suspend_equivalent();
// cleared by handle_special_suspend_equivalent_condition() or java_suspend_self()

assert(_cur_index == -1, "invariant");
if (time == 0) {
_cur_index = REL_INDEX; // arbitrary choice when not timed
// pthread_cond_wait用于阻塞当前线程，等待别的线程使用 pthread_cond_signal或pthread_cond_broadcast来唤醒它
// pthread_cond_wait内部实现：先释放mutex，然后加入waiter队列等待signal，如果有signal唤醒了该线程则后续执行重新上锁
status = pthread_cond_wait (&_cond[_cur_index], _mutex) ;
} else {
_cur_index = isAbsolute ? ABS_INDEX : REL_INDEX;
// 线程进入有超时时间的等待，内部实现调用了pthread_cond_timedwait
status = os::Linux::safe_cond_timedwait (&_cond[_cur_index], _mutex, &absTime) ;
if (status != 0 && WorkAroundNPTLTimedWaitHang) {
pthread_cond_destroy (&_cond[_cur_index]) ;
pthread_cond_init (&_cond[_cur_index], isAbsolute ? NULL : os::Linux::condAttr());
}
}
_cur_index = -1;
assert_status(status == 0 || status == EINTR ||
status == ETIME || status == ETIMEDOUT,
status, "cond_timedwait");

#ifdef ASSERT
pthread_sigmask(SIG_SETMASK, &oldsigs, NULL);
#endif
// _counter重新设置为0
_counter = 0 ;
// 释放互斥锁
status = pthread_mutex_unlock(_mutex) ;
```

```
assert_status(status == 0, status, "invariant") ;
// Paranoia to ensure our locked and lock-free paths interact
// correctly with each other and Java-level accesses.
// 插入写屏障
OrderAccess::fence();

// If externally suspended while waiting, re-suspend
if (jt->handle_special_suspend_equivalent_condition()) {
jt->java_suspend_self();
}
}
```

unpark源码分析

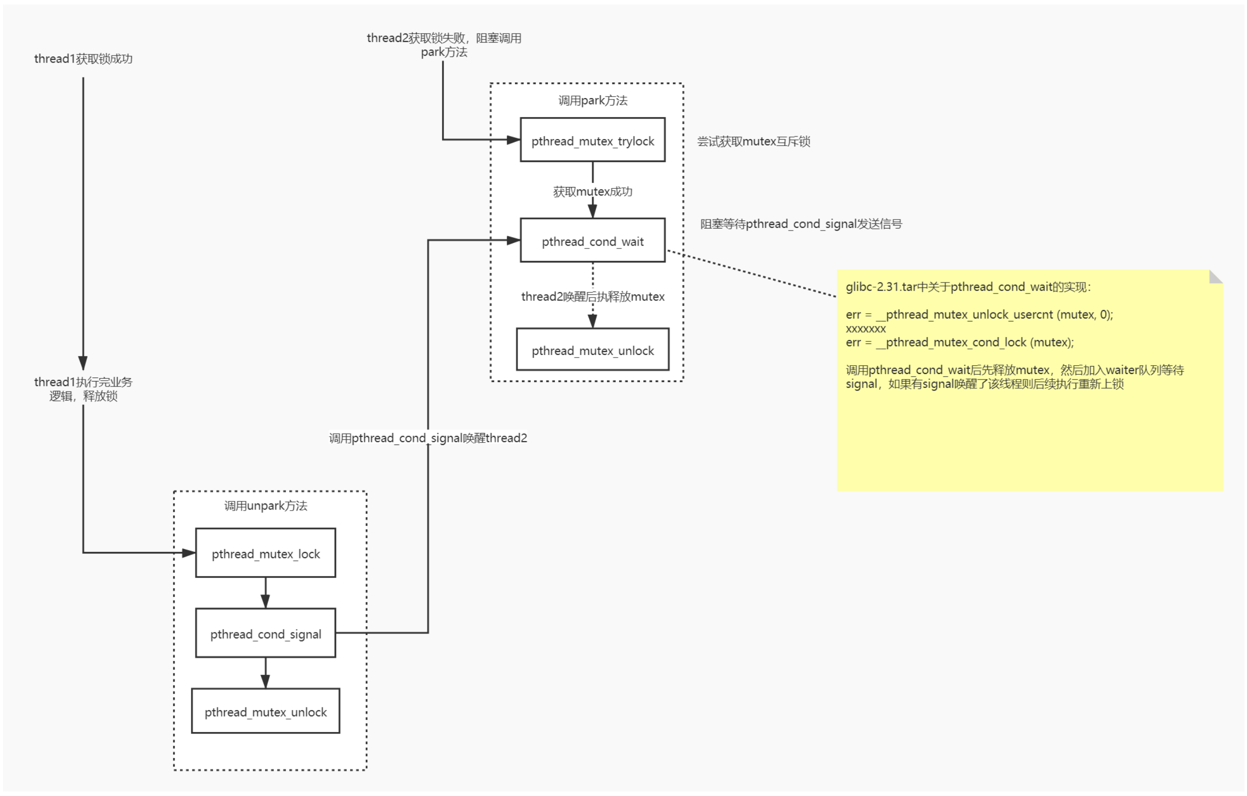
我们再来看看unpark方法的实现流程（源码文件 os_linux.cpp）

1. pthread_mutex_lock 获取互斥锁
2. _counter 设置为 1
3. 判断 _counter 的旧值：

- 小于 1 时，调用 pthread_cond_signal 唤醒在 park 阻塞的线程；
- 等于 1 时，释放互斥锁

```
void Parker::unpark() {
 int s, status ;
 // 获取互斥锁
 status = pthread_mutex_lock(_mutex);
 assert (status == 0, "invariant") ;
 s = _counter;
 // 将_counter置为1
 _counter = 1;
 // s记录的是unpark之前的_counter旧值，如果s < 1，说明有可能该线程在等待状态，
需要唤醒。
  if (s < 1) {
  // thread might be parked
  if (_cur_index != -1) {
  // thread is definitely parked
  if (WorkAroundNPTLTimedWaitHang) {
  // pthread_cond_signal的作用： 发送一个信号给另外一个正在处于阻塞等待状态的
线程,使其脱离阻塞状态,继续执行.
  status = pthread_cond_signal (&_cond[_cur_index]);
  assert (status == 0, "invariant");
  status = pthread_mutex_unlock(_mutex);
  assert (status == 0, "invariant");
  } else {
  status = pthread_mutex_unlock(_mutex);
  assert (status == 0, "invariant");
  status = pthread_cond_signal (&_cond[_cur_index]);
  assert (status == 0, "invariant");
  }
  } else {
  //_cur_index==-1 释放互斥锁
  pthread_mutex_unlock(_mutex);
  assert (status == 0, "invariant") ;
  }
  } else {
  // s==1 释放互斥锁
  pthread_mutex_unlock(_mutex);
  assert (status == 0, "invariant") ;
  }
 }
```

https://www.processon.com/view/link/6139797c63768906a22cb4e2



glibc-2.31.tar中关于pthread_cond_wait的实现：

```
err = __pthread_mutex_unlock_usercnt (mutex, 0);
xxxxxxx
err = __pthread_mutex_cond_lock (mutex);
```

调用pthread_cond_wait后先释放mutex，然后加入waiter队列等待signal，如果有signal唤醒了该线程则后续执行重新上锁

pthread_cond_wait

pthread_cond_wait()函数等待条件变量变为真的。它需要两个参数，第一个参数就是条件变量，而第二个参数mutex是保护条件变量的互斥量。也就是说这个函数在使用的时候需要配合pthread_mutex_lock()一起使用。

```
pthread_mutex_lock(&mutex);
pthread_cond_wait(&cond,&mutex);
```

pthread_cond_wait() 用于阻塞当前线程，等待别的线程使用 pthread_cond_signal() 或pthread_cond_broadcast来唤醒它 。不同之处在于，pthread_cond_signal()可以唤醒至少一个线程；而pthread_cond_broadcast()则是唤醒等待该条件满足的所有线程。在使用的时候需要注意，一定是在改变了条件状态以后再给线程发信号。 pthread_cond_wait() 必须与pthread_mutex 配套使用。pthread_cond_wait() 函数一进入wait状态就会自动release mutex。当其他线程通过 pthread_cond_signal() 或pthread_cond_broadcast ，把该线程唤醒，使 pthread_cond_wait()通过（返回）时，该线程又自动获得该mutex 。

pthread_cond_signal

pthread_cond_signal函数的作用是发送一个信号给另外一个正在处于阻塞等待状态的线程,使其脱离阻塞状态,继续执行。如果没有线程处在阻塞等待状态,pthread_cond_signal也会成功返回。