Java 中如何应用线程：

- 实现 Runnable 接口
  - Java 多重实现，用接口比较方便
- Thread 类（本质上是对 Runnable 接口的实现）
- Callable / Future 带返回值的线程
- ThreadPool

作用：

- 线程可以合理利用多核心 CPU 资源，提高对计算机资源的利用，提高程序的吞吐量
- 如果是单线程应用，就只能利用单核心，不能发挥多核心的优势，来对程序执行效率进行优化

实际应用中，用得比较多的都是线程池

- 单独 new Thread 会有上下文切换、死锁等问题
- 而且单独 new Thread 资源不可控，ThreadPool 在资源可控方面会好很多

多线程的应用：

- 文件跑批：收益文件、对账文件。
  - 跟基金、银行、信托公司对账，一般 T + 1 日提供对账文件
  - 将对账文件读取解析，先存到数据库里，然后再通过定时任务执行对账跑批
- BIO 应用时，使用多线程解决阻塞问题
  - new Thread(new Handler(socket)).start()
- ZooKeeper 源码中的异步责任链模式
  - 第一个责任链节点对象会先将 Request 加入到 LinkedBlockingQueue<Request> 中
  - 然后启动的时候就会启动线程去不断的从 LinkedBlockingQueue 中 take() ，有 Request 则处理并传递给责任链中的下一个处理节点
  - 没有 Request 则会阻塞
- 将程序改造为异步处理
  - 异步消息队列 MQ 中会使用大量的多线程
  - 我们使用的定时任务也可以将程序改为异步处理
  - 支付流程中涉及异步处理，所以要做幂等

所有和阻塞相关的方法，都会抛出一个异常 InterruptedException ？

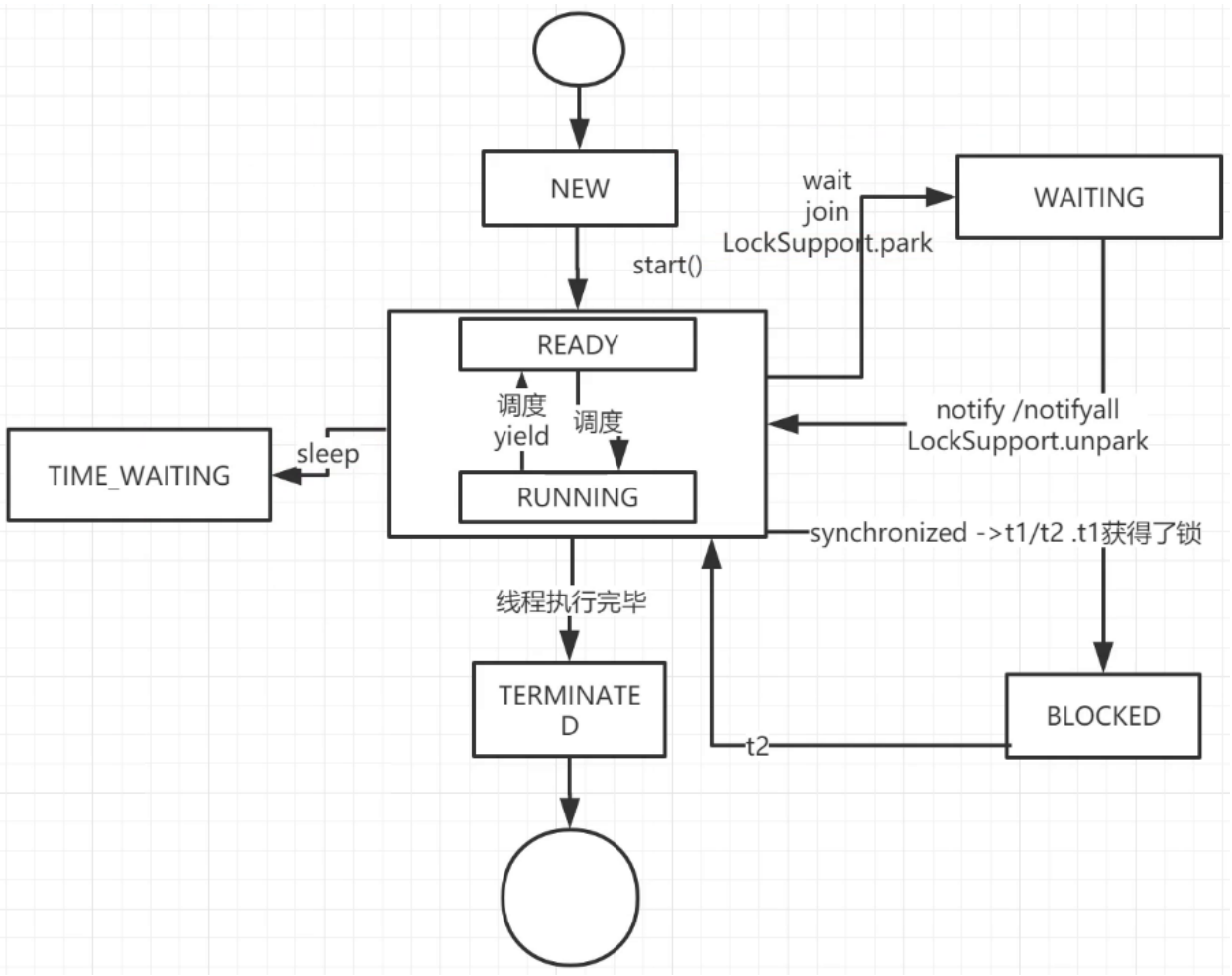
- 这是一个检查异常 checked exception ，所以必须被处理，swallow 或者仅仅是记录都是不推荐的处理方式
  - catch 到这个异常，应该做相应的处理，因为被 interrupted 的线程处于阻塞状态，表示该线程并没有正常执行完
  - 那么当前线程是继续执行还是取消执行，还是设置自己的 interrupted 状态，还是向上层抛出异常
- sleep 、join 、wait 等处于阻塞状态的线程被 interrupt 收到中断请求的时候，就会抛出这个异常
- 正常执行的线程被 interrupt ，只会记录一个状态 isInterrupted ，然后线程检查到这个状态，再进行相应的处理后中断执行
  - 这样可以保证线程执行的数据和完成的工作保证完整性和稳定性
  - 如果当前线程无法处理 InterruptedException ，那么也至少要把自己的 isInterrupted 设置为 true ，好让更上层的调用者知道

```
while(!Thread.currentThread().isInterrupted()){
    try {
        Thread.sleep(1000);
    } catch(InterruptedException ex) {
        Thread.interrupt()
    }
}
```

- 以前可以用 Thread.stop() 让一个线程去停止另一个线程，但是这种方法太暴力，突然停止其他线程会导致被停止的线程无法完成一些清理工作，所以 Thread.stop() 已经被抛弃了
    - Java 线程的终止操作最初是直接暴露给用户的，java.lang.Thread 类提供了 stop() 方法，允许用户暴力的终止一个线程并退出临界区（释放所有锁，并在当前调用栈抛出 ThreadDeath Exception）。 同样的，Thread.suspend() 和 Thread.resume() 方法允许用户灵活的暂停和恢复线程。然而这些看似简便的 API 在 JDK 1.2 就被 deprecate 掉了，原因是 stop() 方法本质上是不安全的，它会强制释放掉线程持有的锁，这样临界区的数据中间状态就会遗留出来，从而造成不可预知的后果。
- 参考：https://www.jianshu.com/p/e2b22c6bcd22
    - Java 线程中止，唯一的也是最好的办法就是让线程从 run() 方法返回
    - Thread.interrupt 的作用其实不是中断线程，而是通知线程应该中断了，给这个线程发一个信号，告诉它，它应该结束了，设置一个停止标志， 具体到底中断还是运行，应该由被通知的线程自己处理
    - 具体来说，对一个线程，调用 interrupt() 时：
        - 如果一个线程处于阻塞状态（如线程调用了 Thread.sleep() 、thread.join() 、Object.wait() 、1.5 中的 Condition.await() 、以及可中断的通道上的 I/O 操作方法后可进入阻塞状态），则在线程在检查中断标示时如果发现中断标示为 true ，则会在这些阻塞方法（Thread.sleep() 、thread.join() 、Object.wait() 、1.5 中的 Condition.await() 及可中断的通道上的 I/O 操作方法）调用处抛出 InterruptedException 异常

回到**并发基础**：

- 生命周期
    - 线程创建到销毁，中间会有很多过程
    - Java 线程的生命周期，我猜应该是 Thread 里定义的六个：NEW 、RUNNABLE 、BLOCKED 、WAITING 、 TIMED_WAITING 、TERMINATED



Thread.sleep() 和 Object.wait() 的区别?

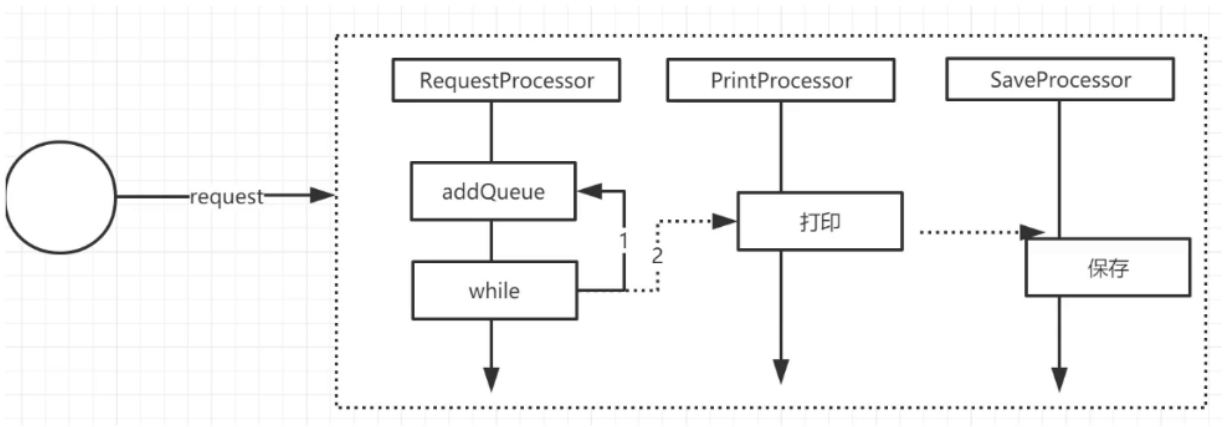
- Thread.sleep() 会让当前线程暂停执行指定的时间，让出 CPU 给其他线程
    - 但是 Monitor 状态依然保持，线程不会释放对象锁
    - 到达指定的时间后，又会自动恢复运行状态

- Object.wait() 方法调用，线程会释放对象锁，进入等待此对象的等待锁定池
    - 只有针对此对象调用 notify() 方法后，本线程才进入对象锁定池准备，获取对象锁进入运行状态

异步责任链模式的好处：

- 如果责任链中有一个任务比较耗时，后续的任务都会阻塞等待，在责任链的处理引入阻塞队列和异步处理的情况下，我认为好处有：
    - 提高吞吐量，为什么呢？因为任务提交很简单，耗时很短，就是把任务加入到队列，就返回成功
    - 提高处理能力，可以多线程处理任务队列中提交的任务，利用线程池的优势，没有任务时就阻塞，释放资源
    - 解耦，提交任务和处理任务分离，可以更好的优化和提高性能

异步责任链一条链上的处理是串行的，但是整个任务的提交和处理是异步化的



线程的使用目的： 可以提高程序性能

线程的基本体系：

- 线程的生命周期：6 个状态
- 线程创建
    - Thread.java 官方注释里写的 2 种
    - 实际中我们创建可以用 4 种
        - extends Thread
        - implements Runnalbe
        - Future / Callable 带返回值的线程
        - 线程池 ThreadPool
- 线程的使用

几个问题：

- 线程的启动为什么是 start ?
    - 看源码可以看到线程只能启动一次，会检查 NEW 状态等
    - hg.openjdk.java.net/jdk8/jdk8/jdk/file/00cd9dc3c2b5/src/share/native/java/lang/Thread.c 里面可以看到 start0 绑定的是 JVM_StartThread
    - http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/87ee5ee27509/src/share/vm/prims/jvm.cpp 里面可以找到 JVM_StartThread
        - 可以看到其中比较关键的是：

```
native_thread = new JavaThread(&thread_entry, sz);
……
Thread::start(native_thread);
```

        - 在 http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/095e60e7fc8c/src/share/vm/runtime/thread.cpp

中看 new JavaThread(&thread_entry, sz) 执行了什么

```
JavaThread::JavaThread(ThreadFunction entry_point, size_t stack_sz) :
  Thread()
```

- 其中，最关键的是

```
os::create_thread(this, thr_type, stack_sz);
```

- 调用 os 操作系统的方法，去创建线程
- 另外可以看到 start 相关源码

```
void Thread::start(Thread* thread) {
  trace("start", thread);
  // Start is different from resume in that its safety is guaranteed by context or
  // being called from a Java method synchronized on the Thread object.
  if (!DisableStartThread) {
    if (thread->is_Java_thread()) {
      // Initialize the thread state to RUNNABLE before starting this thread.
      // Can not set it after the thread started because we do not know the
      // exact thread state at that time. It could be in MONITOR_WAIT or
       // in SLEEPING or some other state.
       java_lang_Thread::set_thread_status(((JavaThread*)thread)->threadObj(),
                                           java_lang_Thread::RUNNABLE);
    }
    os::start_thread(thread);
  }
}
```

- 可以看到 Thread 的状态被设置为 RUNNABLE ，然后调用 os 的方法 start_thread ，启动后会回调 Thread 里的 run 方法

- 线程的终止?
  - 线程的终止，stop 方法已经被 deprecated 了，还有 suspend 挂起和 resume 恢复方法也是 deprecated
    - stop 方法相当于 kill -9 去强行终止一个线程，这是一种不安全的操作
      - 因为当前调用 stop 的线程不知道被 stop 的线程的状态
        - 可能只运行了一半，就把被 stop 的线程关闭了，会造成一些问题，比如临界区数据结构的完整性和稳定性、一些应该做的清理工作（导致临界区数据中间状态遗留，从而造成不可预知的后果）
    - 以前可以用 Thread.stop() 让一个线程去停止另一个线程，但是这种方法太暴力，突然停止其他线程会导致被停止的线程无法完成一些清理工作，所以 Thread.stop() 已经被抛弃了
      - Java 线程的终止操作最初是直接暴露给用户的，java.lang.Thread 类提供了 stop() 方法，允许用户暴力的终止一个线程并退出临界区（释放所有锁，并在当前调用栈抛出 ThreadDeath Exception）。同样的，Thread.suspend() 和 Thread.resume() 方法允许用户灵活的暂停和恢复线程。然而这些看似简便的 API 在 JDK 1.2 就被 deprecate 掉了，原因是 stop() 方法本质上是不安全的，它会强制释放掉线程持有的锁，这样临界区的数据中间状态就会遗留出来，从而造成不可预知的后果。
    - 所以 stop 不建议使用了
  - 如何正确的关闭线程?
    - Thread.interrupt()
      - 参考：https://www.jianshu.com/p/e2b22c6bcd22

    - Java 线程中止，唯一的也是最好的办法就是让线程从 run() 方法返回
    - Thread.interrupt 的作用其实不是中断线程，而是通知线程应该中断了，给这个线程发一个信号，告诉它，它应该结束了，设置一个停止标志， 具体到底中断还是运行，应该由被通知的线程自己处理
    - 具体来说，对一个线程，调用 interrupt() 时：
        - 如果一个线程处于阻塞状态（如线程调用了 Thread.sleep() 、thread.join() 、Object.wait() 、1.5 中的 Condition.await() 、以及可中断的通道上的 I/O 操作方法后可进入阻塞状态），则在线程在检查中断标示时如果发现中断标示为 true ，则会在这些阻塞方法调用处抛出 InterruptedException 异常
- 可以理解为 Thread 中有一个 volatile int isInterrupted ，为 0 表示 false ，为 1 表示 true
    - volatile 修饰以后，加了一个内存屏障
        - 内存屏障（Memory Barrier，或有时叫做内存栅栏，Memory Fence）是一种 CPU 指令，用于控制特定条件下的重排序和内存可见性问题。Java 编译器也会根据内存屏障的规则禁止重排序。
        - 内存屏障可以被分为以下几种类型：
            - LoadLoad 屏障：对于这样的语句 Load1; LoadLoad; Load2 ，在 Load2 及后续读取操作要读取的数据被访问前，保证 Load1 要读取的数据被读取完毕。
            - StoreStore 屏障：对于这样的语句 Store1; StoreStore; Store2 ，在 Store2 及后续写入操作执行前，保证 Store1 的写入操作对其它处理器可见。
            - LoadStore 屏障：对于这样的语句 Load1; LoadStore; Store2 ，在 Store2 及后续写入操作被刷出前，保证 Load1 要读取的数据被读取完毕。
            - StoreLoad 屏障：对于这样的语句 Store1; StoreLoad; Load2 ，在 Load2 及后续所有读取操作执行前，保证 Store1 的写入对所有处理器可见。它的开销是四种屏障中最大的。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能。
        - 有的处理器的重排序规则较严，无需内存屏障也能很好的工作，Java 编译器会在这种情况下不放置内存屏障。
        - 参考：https://www.sohu.com/a/313178496 100111562
- Thread.interrupt() 里面是怎么实现的?
    - hg.openjdk.java.net/jdk8/jdk8/jdk/file/00cd9dc3c2b5/src/share/native/java/lang/Thread.c 可以看到 interrupt0 对应的 native 方法是 JVM_Interrupt
    - 在 http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/87ee5ee27509/src/share/vm/prims/jvm.cpp 中找到 JVM_Interrupt

```
// Consider: A better way to implement JVM_Interrupt() is to acquire
// Threads_lock to resolve the jthread into a Thread pointer, fetch
// Thread->platformevent, Thread->native_thr, Thread->parker, etc.,
// drop Threads_lock, and the perform the unpark() and thr_kill() operations
// outside the critical section.  Threads_lock is hot so we want to minimize
// the hold-time.  A cleaner interface would be to decompose interrupt into
// two steps.  The 1st phase, performed under Threads_lock, would return
// a closure that'd be invoked after Threads_lock was dropped.
// This tactic is safe as PlatformEvent and Parkers are type-stable (TSM) and
 // admit spurious wakeups.

```

```
JVM_ENTRY(void, JVM_Interrupt(JNIEnv* env, jobject jthread))
  JVMWrapper("JVM_Interrupt");

  // Ensure that the C++ Thread and OSThread structures aren't freed before we op
  oop java_thread = JNIHandles::resolve_non_null(jthread);
  MutexLockerEx ml(thread->threadObj() == java_thread ? NULL : Threads_lock);
  // We need to re-resolve the java_thread, since a GC might have happened during
  // acquire of the lock
  JavaThread* thr = java_lang_Thread::thread(JNIHandles::resolve_non_null(jthread
  if (thr != NULL) {
    Thread::interrupt(thr);
  }
JVM_END
```

- 注意这里调用了 Thread::interrupt(thr) ，然后我们进入 http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/095e60e7fc8c/src/share/vm/runtime/thread.cpp 可以看到

```
void Thread::interrupt(Thread* thread) {
  trace("interrupt", thread);
  debug_only(check_for_dangling_thread_pointer(thread);)
  os::interrupt(thread);
}
```

- 最后调用了 os 的 interrupt 方法，而 os 相关的 cpp 文件针对不同平台有很多个，都是针对不同平台的实现，所以 Java 同样的代码可以执行在不同平台
  - 打开 os_linux.cpp ，可以看到 OrderAccess::fence() 是内存屏障，关键代码 osthread->set_interrupted(true) 设置属性

```
void os::interrupt(Thread* thread) {
  assert(Thread::current() == thread || Threads_lock->owned_by_self(),
    "possibility of dangling Thread pointer");

  OSThread* osthread = thread->osthread();

  if (!osthread->interrupted()) {
    osthread->set_interrupted(true);
    // More than one thread can get here with the same value of osthread,
     // resulting in multiple notifications.  We do, however, want the store
     // to interrupted() to be visible to other threads before we execute unpark()
     OrderAccess::fence();
     ParkEvent * const slp = thread->_SleepEvent ;
     if (slp != NULL) slp->unpark() ;
  }

  // For JSR166. Unpark even if interrupt status already was set
  if (thread->is_Java_thread())
    ((JavaThread*)thread)->parker()->unpark();
```

```
  ParkEvent * ev = thread->_ParkEvent ;
  if (ev != NULL) ev->unpark() ;

}
```

- 打开 osThread.hpp ，可以看到 set_interrupted 方法

```
void set_interrupted(bool z)                    { _interrupted = z ? 1 : 0; }
```

- 可以看到 _interrupted 是定义的一个 volatile 的属性

```
volatile jint _interrupted;      // Thread.isInterrupted state
```

线程的中断和线程的复位?

- InterruptedException
  - 试想，一个线程处于阻塞状态，如何及时的通知它中断以及采取相应的行动？答案是 InterruptedException ，让它及时 catch 到 InterruptedException ，然后进行相应的处理，最后结束掉自己的运行
  - 不然，处于阻塞状态的线程被中断以后还得等阻塞结束，就不能及时进行相应的处理
- 线程被中断以后如何复位?
  - 复位是指对中断的线程进行标识的复位，方法有以下两种：
    - Thread.interrupted()
    - InterruptedException
      - 中断一个处于阻塞状态的线程会抛出异常
        - Thread.sleep() 、thread.join() 、object.wait() 、blockingQueue.take() ，以及其他的一些阻塞，如可中断 IO 阻塞等
      - 抛出 InterruptedException 异常后，isInterrupted 会复位为 false
      - 原因：
        - 中断之前会先去复位再抛出这样一个异常
        - 可以看源码中

```
// Start: jvm.cpp 中 JVM_ENTRY(void, JVM_Sleep(JNIEnv* env, jclass threadClass, jlo
if (Thread::is_interrupted (THREAD, true) && !HAS_PENDING_EXCEPTION) {
  THROW_MSG(vmSymbols::java_lang_InterruptedException(), "sleep interrupted");
}
// End  : jvm.cpp 中 JVM_ENTRY(void, JVM_Sleep(JNIEnv* env, jclass threadClass, jlo
// Start: objectMonitor.cpp 中 void ObjectMonitor::wait(jlong millis, bool interrup
// check for a pending interrupt
if (interruptible && Thread::is_interrupted(Self, true) && !HAS_PENDING_EXCEPTION)
  // post monitor waited event.  Note that this is past-tense, we are done waiting
  if (JvmtiExport::should_post_monitor_waited()) {
     // Note: 'false' parameter is passed here because the
     // wait was not timed out due to thread interrupt.
     JvmtiExport::post_monitor_waited(jt, this, false);
  }
  if (event.should_commit()) {
    post_monitor_wait_event(&event, 0, millis, false);
  }
  TEVENT (Wait - Throw IEX) ;
  THROW(vmSymbols::java_lang_InterruptedException());
```

```
20    return ;
21  }
22  // 以及
23  // check if the notification happened
24  if (!WasNotified) {
25    // no, it could be timeout or Thread.interrupt() or both
26    // check for interrupt event, otherwise it is timeout
27    if (interruptible && Thread::is_interrupted(Self, true) && !HAS_PENDING_EXCEPTI(
28      TEVENT (Wait - throw IEX from epilog) ;
29      THROW(vmSymbols::java_lang_InterruptedException());
30    }
31  }
32  // End  : objectMonitor.cpp 中 void ObjectMonitor::wait(jlong millis, bool interru
```

- 都有 Thread::is_interrupted(Self, true) / Thread::is_interrupted (THREAD, true)，还是在 thread.cpp 中，我们可以看到后面为 true 的都是这个 clear_interrupted

```
1  bool Thread::is_interrupted(Thread* thread, bool clear_interrupted) {
2    trace("is_interrupted", thread);
3    debug_only(check_for_dangling_thread_pointer(thread);)
4    // Note:  If clear_interrupted==false, this simply fetches and
5    // returns the value of the field osthread()->interrupted().
6    return os::is_interrupted(thread, clear_interrupted);
7  }
```

- 最后调用具体 os 的 is_interrupted，在 os_linux.cpp 中调用的事以下方法

```
1  bool os::is_interrupted(Thread* thread, bool clear_interrupted) {
2    assert(Thread::current() == thread || Threads_lock->owned_by_self(),
3      "possibility of dangling Thread pointer");
4
5    OSThread* osthread = thread->osthread();
6
7    bool interrupted = osthread->interrupted();
8
9    if (interrupted && clear_interrupted) {
10     osthread->set_interrupted(false);
11     // consider thread->_SleepEvent->reset() ... optional optimization
12   }
13
14   return interrupted;
15 }
```

- 最后返回的 interrupted 为 ture 并且 osthread->set_interrupted(false) 将 _interrupted 设置为 false
- 所以说，中断之前会先去复位（Thread::is_interrupted(Self, true) / Thread::is_interrupted (THREAD, true) 的 clear_interrupted 为 true，最终会把底层的 _interrupted 设置为 false）再抛出这样一个异常（THROW_MSG(vmSymbols::java_lang_InterruptedException(), "sleep interrupted") / THROW(vmSymbols::java_lang_InterruptedException())）

- 为什么需要复位?

        - Thread.interrupted() 是属于当前线程的操作，是当前线程对外界中断信号的一种回应（从英语语法上也可以体会 interrupted 表示我已经知道被中断了），可能我并不一定会马上进行处理完成处理，什么时候完成中断由自己决定
        - 让外界知道我现在还不能中断，所以先置为 false 复位，之后可能会再次被中断置为 true

InterruptedException 所有被阻塞的线程被中断都会抛出这个异常，为什么?

- thread.join() 、Thread.sleep() 、object.wait() 、blockingQueue.take() 等阻塞方法及其 TIMED_WAITING 的阻塞方法，相应的释放
    - object.wait() 的释放 object.notify()
    - Thread.sleep() 取决于设置的时间
    - thread.join() 等到 thread 执行完，或者 thread 执行时间超过 join(long millis) 中设置的时间，当前调用 thread.join() 的线程才重新进入 Runnable 状态，得以继续执行
        - 参考 https://my.oschina.net/payzheng/blog/690059
            - join 的本质是使用 synchronized 先获取当前 Thread 对象的锁
            - 然后在方法中调用 wait(0) / wait(delay) （前提是当前线程 isAlive()），此时 wait() 释放了前面的当前 Thread 对象的锁，等待被唤醒
            - After run() finishes, notify() is called by the Thread subsystem. 当线程运行结束的时候，notify 是被线程的子系统调用的
            - 当使用线程对象作为锁的时候，如果锁对象执行完毕了，wait 就会被线程的子系统调用 notify 唤醒，停止等待继续执行
- 如果一个线程阻塞，无法释放，就需要其他线程来停止该线程正在做的事情，解除阻塞。被阻塞的线程被中断会抛出 InterruptedException
    - 它会中断当前阻塞的被中断线程，并抛出 InterruptedException 异常，告诉它应该处理中断
- 如何处理 InterruptedException ?
    - 抛出 InterruptedException 只是表达了一个信号，如果没有合适的处理，线程在异常之后，还是会不断的运行之后的循环，因为此时 isInterrupted 为 false 被复位了
    - 需要被中断的线程本身去处理，完成当前该完成的处理，然后执行完毕退出
        - 捕获异常，Swallow （不推荐）
        - 线程执行完当前任务，结束运行
        - 把异常继续往上层抛

总结:

- 线程的启动，是 start 方法，基于不同的操作系统在 JDK 源码中进行相应实现，实现了不同的线程创建和启动指令
- interrupt 线程中断，可以达到终止线程的目的
    - 可以通过 thread.interrupted() 对 JDK 源码中的 _interrupted 进行复位重置
    - 阻塞的线程被中断，可以通过在 InterruptedException 的处理中进行重置
- 复位重置 _interrupted 的目的:
    - 表示当前我已经收到了中断信号，我可能不会立刻中断自己
    - 并且也需要让外部调用者知道，我在中断之前，线程的中断状态依然是 false
- JDK 源码中有一个标记 _interrupted ，以及进行相应处理的代码

```
volatile jint _interrupted;      // Thread.isInterrupted state
```

- JDK 中 interrupt 源码

```
void os::interrupt(Thread* thread) {
  assert(Thread::current() == thread || Threads_lock->owned_by_self(),
    "possibility of dangling Thread pointer");

  OSThread* osthread = thread->osthread();
```

```
6
7   if (!osthread->interrupted()) {
8     osthread->set_interrupted(true);
9     // More than one thread can get here with the same value of osthread,
10     // resulting in multiple notifications.  We do, however, want the store
11     // to interrupted() to be visible to other threads before we execute unpark()
12     OrderAccess::fence();
13     ParkEvent * const slp = thread->_SleepEvent ;
14     if (slp != NULL) slp->unpark() ;
15   }
16
17   // For JSR166. Unpark even if interrupt status already was set
18   if (thread->is_Java_thread())
19     ((JavaThread*)thread)->parker()->unpark();
20
21   ParkEvent * ev = thread->_ParkEvent ;
22   if (ev != NULL) ev->unpark() ;
23
24 }
```

- 这里不仅 osthread->set_interrupted(true) 将 _interrupted 设置为 true （在 _interrupted 为 false 的情况下 !osthread->interrupted()）
  - 还看到 OrderAccess::fence() 内存屏障，以保证 _interrupted 的 store 对后续的其他 threads 在我们执行 unpark() 之前都是可见的
    - 因为这里不止一个 thread 能执行进来，并且是以相同的 osthread 值，会导致 multiple notifications 多重通知
- ParkEvent 与 LockSupport.park() 、LockSupport.unpark() 相关
  - 这里 unpark() 会去唤醒阻塞的线程，唤醒以后现在 _interrupted 已经为 true 了，然后被唤醒的线程再去继续处理中断
    - 非阻塞的线程就直接自己处理中断
    - 阻塞的会抛出 InterruptedException ，也是线程自己处理
- 下面这一段应该跟同步锁相关

```
1 if (thread->is_Java_thread())
2     ((JavaThread*)thread)->parker()->unpark();
```

问题：

- 多线程单核能否提高效率?
  - 可以提高效率，因为线程比进程更轻量，切换线程比切换进程更快
- 并行和并发的区别
  - 并行
    - 就像有一条 4 车道的马路，当前可以并行 4 辆车
  - 并发
    - 这条马路能支撑的车辆数，可能是 40 辆车，但肯定多于并行的 4 辆车
    - 吞吐量可以支撑 40 辆车，如果超过 40 辆，处理不过来，就会堵车