一，并发的发展历史

真空管和穿孔打卡

操作员在机房里面来回调度资源，以及计算机同一个时刻只能运行一个程序，在程序输入的过程中，计算机的计算和处理空闲状态 。而当时的计算机是非常昂贵的，人们为了减少这种资源的浪费。就采用了批处理系统来解决。

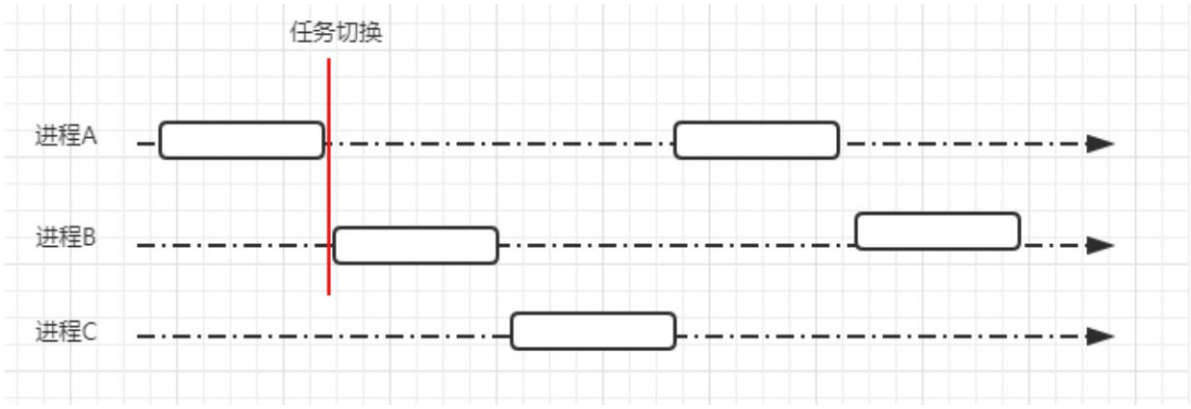
晶体管和批处理系统

批处理操作系统虽然能够解决计算机的空闲问题，但是当某一个作业因为等待磁盘或者其他 I/O 操作而暂停时，那CPU 就只能阻塞直到该 I/O 完成，对于 CPU 操作密集型的程序，I/O 操作相对较少，因此浪费的时间也很少。但是对于 I/O 操作较多的场景来说，CPU 的资源是属于严重浪费的。

集成电路和多道程序设计

多道程序设计的出现解决了这个问题，就是把内存分为几个部分，每一个部分放不同的程序。当一个程序需要等待I/O 操作完成时。那么 CPU 可以切换执行内存中的另外一个程序。如果内存中可以同时存放足够多的程序，那 CPU的利用率可以接近 100%。

进程的本质是一个正在执行的程序，程序运行时系统会创建一个进程，并且给每个进程分配独立的内存地址空间保证每个进程地址不会相互干扰。同时，在 CPU 对进程做时间片的切换时，保证进程切换过程中仍然要从进程切换之前运行的位置出开始执行。所以进程通常还会包括程序计数器、堆栈指针。



有了进程以后，可以让操作系统从宏观层面实现多应用并发。而并发的实现是通过 CPU 时间片不断切换执行的。对于单核 CPU 来说，在任意一个时刻只会有一个进程在被CPU 调度。

线程的出现

有了进程以后，为什么还会发明线程呢?

1，在多核 CPU 中，利用多线程可以实现真正意义上的并行执行

2，在一个应用进程中，会存在多个同时执行的任务，如果其中一个任务被阻塞，将会引起不依赖该任务的任务也被阻塞。通过对不同任务创建不同的线程去处理，可以提升程序处理的实时性

3，线程可以认为是轻量级的进程，所以线程的创建、销毁比进程更快

二，线程的应用

如何应用多线程

在 Java 中，有多种方式来实现多线程。继承 Thread 类、实现 Runnable 接口、使用 ExecutorService、Callable、Future 实现带返回结果的多线程。

继承 Thread 类创建线程

Thread 类本质上是实现了 Runnable 接口的一个实例，代表一个线程的实例。启动线程的唯一方法就是通过 Thread类的 start()实例方法。start0()方法是一个 native 方法，它会启动一个新线程，并执行 run()方法。

```
public class Thread implements Runnable{

    private Runnable target;
    @Override
    public void run() {
        if (target != null) {
            target.run();
        }
    }

    public synchronized void start() {

        if (threadStatus != 0)
            throw new IllegalThreadStateException();

        group.add(this);

        boolean started = false;
        try {
            start0();
            started = true;
        } finally {
            try {
                if (!started) {
                    group.threadStartFailed(this);
                }
            } catch (Throwable ignore) {
                /* do nothing. If start0 threw a Throwable then
                  it will be passed up the call stack */
            }
        }
    }

    private native void start0();
}
```

实现Runnable接口创建线程

```
@FunctionalInterface
public interface Runnable {

    public abstract void run();
}
```

为什么实现Runnable接口可以实现多线程?

```
public class CreateByRunnable implements Runnable{
    @Override
    public void run() {

    }
    public static void main(String[] args) {
        CreateByRunnable runnable = new CreateByRunnable();
        new Thread(runnable).start();
    }
}
```

Thread

```
    public Thread(Runnable target) {
        init(null, target, "Thread-" + nextThreadNum(), 0);
    }

    private void init(ThreadGroup g, Runnable target, String name,
                      long stackSize, AccessControlContext acc) {
        this.target = target;
    }
```

实现Callable接口通过FutureTask包装器来创建Thread线程

有的时候，我们可能需要让一步执行的线程在执行完成以后，提供一个返回值给到当前的主线程，主线程需要依赖这个值进行后续的逻辑处理，那么这个时候，就需要用到带返回值的线程了。Java 中提供了这样的实现方式：

```
public class CreateByCallable implements Callable<String> {
    @Override
    public String call() throws Exception {
        return "null";
    }

    public static void main(String[] args) {
        FutureTask<String> task = new FutureTask<>(new CreateByCallable());
    }
}
```

```
    public void run() {
        if (state != NEW ||
            !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                         null, Thread.currentThread()))
            return;
        try {
            Callable<V> c = callable;
            if (c != null && state == NEW) {
                V result;
                boolean ran;
                try {
                    result = c.call();
                    ran = true;
                } catch (Throwable ex) {
                    result = null;
                    ran = false;
```

```
                    setException(ex);
                }
                if (ran)
                    set(result);
            }
        } finally {
            // runner must be non-null until state is settled to
            // prevent concurrent calls to run()
            runner = null;
            // state must be re-read after nulling runner to prevent
            // leaked interrupts
            int s = state;
            if (s >= INTERRUPTING)
                handlePossibleCancellationInterrupt(s);
        }
    }
```

三，Java 并发编程的基础

线程的生命周期

线程一共有 6 种状态（NEW、RUNNABLE、BLOCKED、WAITING、TIME_WAITING、TERMINATED）

NEW：初始状态，线程被构建，但是还没有调用 start 方法

RUNNABLED：运行状态，JAVA 线程把操作系统中的就绪和运行两种状态统一称为“运行中”

BLOCKED：阻塞状态，表示线程进入等待状态,也就是线程因为某种原因放弃了 CPU 使用权，阻塞也分为几种情况：

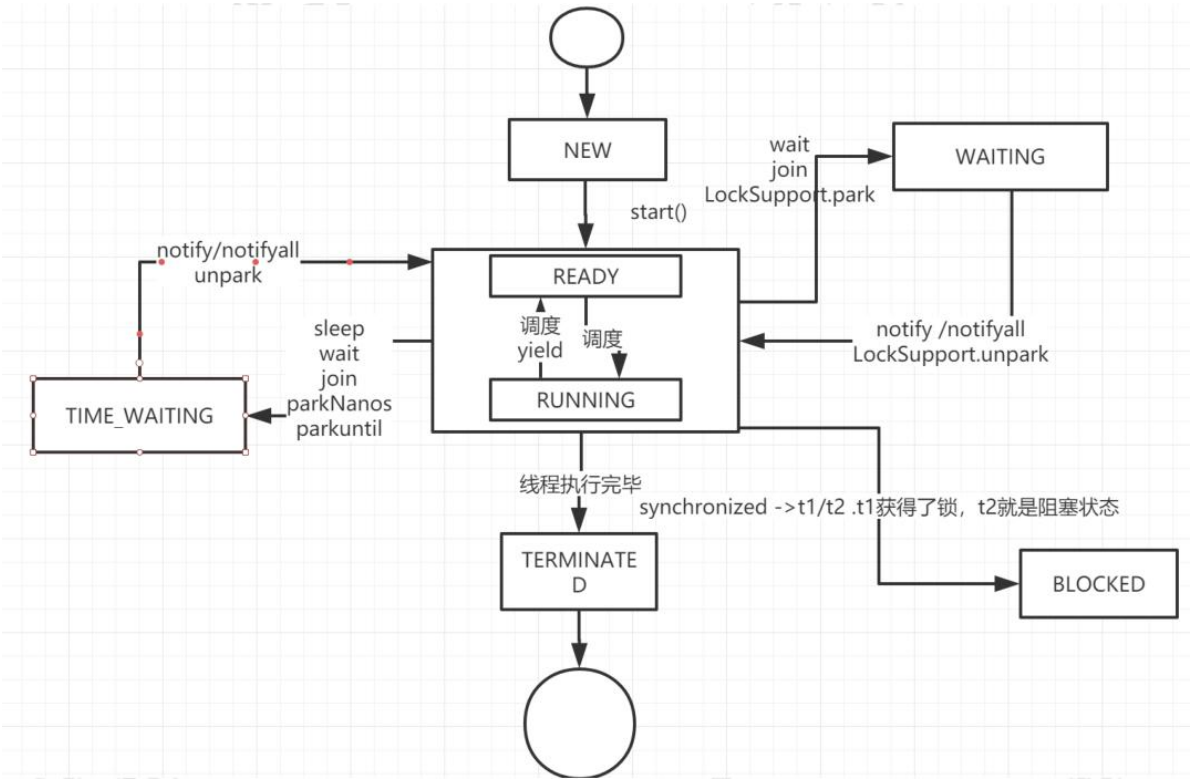
等待阻塞：运行的线程执行 wait 方法，jvm 会把当前线程放入到等待队列

同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被其他线程锁占用了，那么 jvm 会把当前的线程放入到锁池中

其他阻塞：运行的线程执行 Thread.sleep 或者 t.join 方法，或者发出了 I/O 请求时，JVM 会把当前线程设置为阻塞状态，当 sleep 结束、join 线程终止、io 处理完毕则线程恢复

TIME_WAITING：超时等待状态，超时以后自动返回

TERMINATED：终止状态，表示当前线程执行完毕



通过代码演示线程的状态

代码

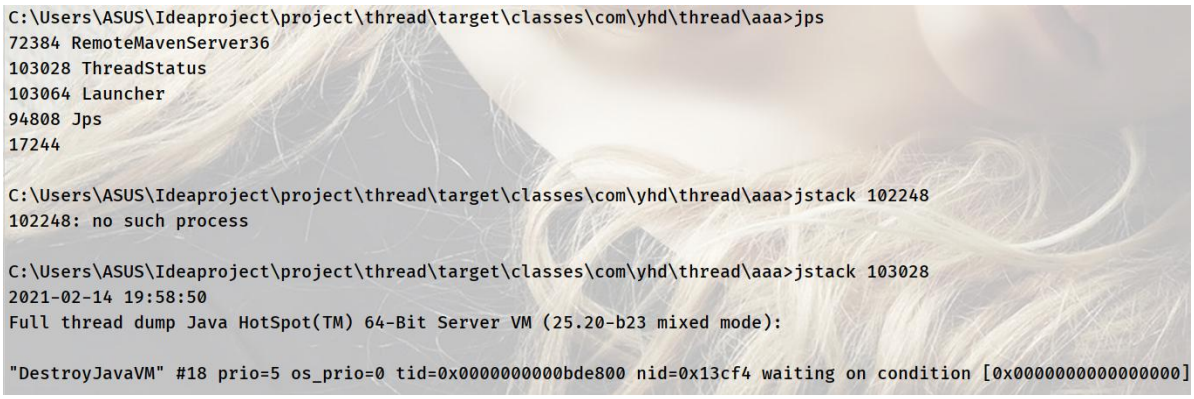
```
public class ThreadStatus {
    public static void main(String[] args) {
        //TIME_WAITING
        new Thread(() -> {
            while (true) {
                try {
                    TimeUnit.SECONDS.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "timewaiting").start();

        //WAITING，线程在 ThreadStatus 类锁上通过 wait 进行等待
        new Thread(() -> {
            while (true) {
                synchronized (ThreadStatus.class) {
                    try {
                        ThreadStatus.class.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }, "Waiting").start();

        //线程在 ThreadStatus 加锁后，不会释放锁
        new Thread(new BlockedDemo(), "BlockDemo-01").start();
        new Thread(new BlockedDemo(), "BlockDemo-02").start();
    }
```

```
    static class BlockedDemo extends Thread {
        @Override
        public void run() {
            synchronized (BlockedDemo.class) {
                while (true) {
                    try {
                        TimeUnit.SECONDS.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

显示线程的状态



jstack是 java 虚拟机自带的一种堆栈跟踪工具。jstack 用于打印出给定的 java 进程 ID 或 core file 或远程调试服务的 Java 堆栈信息

通过上面的分析，我们了解到了线程的生命周期，现在在整个生命周期中并不是固定的处于某个状态，而是随着代码的执行在不同的状态之间进行切换

线程的启动

线程的启动

调用start()方法去启动一个线程，当 run 方法中的代码执行完毕以后，线程的生命周期也将终止。调用 start 方法的语义是当前线程告诉 JVM，启动调用 start 方法的线程。

线程的启动原理

启动一个线程为什么是调用 start 方法，而不是 run 方法?

调用 start 方法实际上是调用一个 native 方法start0()来启动一个线程，首先 start0()这个方法是在 Thread 的静态块中来注册的

registerNatives 的本地方法的定义在文件Thread.c,Thread.c定义了各个操作系统平台要用的关于线程的公共数据和操作

```
#include "jni.h"
#include "jvm.h"
```

```
#include "java_lang_Thread.h"

#define THD "Ljava/lang/Thread;"
#define OBJ "Ljava/lang/Object;"
#define STE "Ljava/lang/StackTraceElement;"

#define ARRAY_LENGTH(a) (sizeof(a)/sizeof(a[0]))

static JNINativeMethod methods[] = {
    {"start0",           "()V",        (void *)&JVM_StartThread},
    {"stop0",            "(" OBJ ")V", (void *)&JVM_StopThread},
    {"isAlive",          "()Z",        (void *)&JVM_IsThreadAlive},
    {"suspend0",         "()V",        (void *)&JVM_SuspendThread},
    {"resume0",          "()V",        (void *)&JVM_ResumeThread},
    {"setPriority0",     "(I)V",       (void *)&JVM_SetThreadPriority},
    {"yield",            "()V",        (void *)&JVM_Yield},
    {"sleep",            "(J)V",       (void *)&JVM_Sleep},
    {"currentThread",    "()" THD,     (void *)&JVM_CurrentThread},
    {"countStackFrames", "()I",        (void *)&JVM_CountStackFrames},
    {"interrupt0",       "()V",        (void *)&JVM_Interrupt},
    {"isInterrupted",    "(Z)Z",       (void *)&JVM_IsInterrupted},
    {"holdsLock",        "(" OBJ ")Z", (void *)&JVM_HoldsLock},
    {"getThreads",        "()[" THD,   (void *)&JVM_GetAllThreads},
    {"dumpThreads",      "([" THD ")[[" STE, (void *)&JVM_DumpThreads},
};

#undef THD
#undef OBJ
#undef STE

JNIEXPORT void JNICALL
Java_java_lang_Thread_registerNatives(JNIEnv *env, jclass cls)
{
    (*env)->RegisterNatives(env, cls, methods, ARRAY_LENGTH(methods));
}
```

从这段代码可以看出， start0()， 实际会执行JVM_StartThread 方法，这个方法是干嘛的呢? 从名字上来看，似乎是在 JVM 层面去启动一个线程，如果真的是这样，那么在 JVM 层面，一定会调用 Java 中定义的 run 方法。那接下来继续去找找答案。我们找到 jvm.cpp 这个文件；这个文件需要下载 hotspot 的源码才能找到.

```
JVM_ENTRY(void, JVM_StartThread(JNIEnv* env, jobject jthread))
    native_thread = new JavaThread(&thread_entry, sz);
```

JVM_ENTRY 是用来定义 JVM_StartThread 函数的，在这个函数里面创建了一个真正和平台有关的本地线程. 本着打破砂锅查到底的原则，继续看看 newJavaThread 做了什么事情,继续寻找 JavaThread 的定义

在 hotspot 的源码中 thread.cpp 文件中 1374 行的位置可以找到如下代码

```
JavaThread::JavaThread(ThreadFunction entry_point, size_t stack_sz) :
JavaThread() {
  _jni_attach_state = _not_attaching_via_jni;
  set_entry_point(entry_point);
  // Create the native thread itself.
  // %note runtime_23
  os::ThreadType thr_type = os::java_thread;
  thr_type = entry_point == &CompilerThread::thread_entry ? os::compiler_thread
:
                                                            os::java_thread;
  os::create_thread(this, thr_type, stack_sz);

}
```

这个方法有两个参数，第一个是函数名称，线程创建成功之后会根据这个函数名称调用对应的函数；第二个是当前进程内已经有的线程数量。最后我们重点关注与一下os::create_thread,实际就是调用平台创建线程的方法来创建线程。

接下来就是线程的启动，会调用 Thread.cpp 文件中的Thread::start(Thread* thread)方法，代码如下

```
void Thread::start(Thread* thread) {

  if (thread->is_Java_thread()) {

    java_lang_Thread::set_thread_status(thread->as_Java_thread()->threadObj(),
                                        JavaThreadStatus::RUNNABLE);
  }
  os::start_thread(thread);
}
```

start 方法中有一个函数调用： os::start_thread(thread);，调用平台启动线程的方法，最终会调用 Thread.cpp 文件中的 JavaThread::run()方法

线程的终止

线程的终止，并不是简单的调用 stop 命令去。虽然 api 仍然可以调用，但是和其他的线程控制方法如 suspend、resume 一样都是过期了的不建议使用，就拿 stop 来说，stop 方法在结束一个线程时并不会保证线程的资源正常释放，因此会导致程序可能出现一些不确定的状态。

要优雅的去中断一个线程，在线程中提供了一个 interrupt方法

interrupt 方法

当其他线程通过调用当前线程的 interrupt 方法，表示向当前线程打个招呼，告诉他可以中断线程的执行了，至于什么时候中断，取决于当前线程自己。

线程通过检查自身是否被中断来进行响应，可以通过isInterrupted()来判断是否被中断。

通过下面这个例子，来实现了线程终止的逻辑

```
public class InterruptTest {

    private static int i;

    public static void main(String[] args)throws Exception {
        Thread thread = new Thread(() -> {
```

```
            while (!Thread.currentThread().isInterrupted()) {
                //默认情况下isInterrupted 返回 false、通过 thread.interrupt 变成了
true
                i++;
            }
            System.out.println("Num:" + i);
        }, "interruptDemo");
        thread.start();
        TimeUnit.SECONDS.sleep(1);
        thread.interrupt(); //加和不加的效果
    }
}
```

这种通过标识位或者中断操作的方式能够使线程在终止时有机会去清理资源，而不是武断地将线程停止，因此这种终止线程的做法显得更加安全和优雅。

Thread.interrupted

上面的案例中，通过 interrupt，设置了一个标识告诉线程可以终止了，线程中还提供了静态方法Thread.interrupted()对设置中断标识的线程复位。比如在上面的案例中，外面的线程调用 thread.interrupt 来设置中断标识，而在线程里面，又通过 Thread.interrupted 把线程的标识又进行了复位。

```
public class InterruptDemo {

    public static void main(String[] args) throws InterruptedException {
        Thread thread = new
                Thread(() -> {
            while (true) {
                if (Thread.currentThread().isInterrupted()) {
                    System.out.println("before:" +
Thread.currentThread().isInterrupted())
                    ;
                    Thread.interrupted(); // 对线程进行复位，由 true 变成 false
                    System.out.println("after:" + Thread
                            .currentThread().isInterrupted());
                }
            }
        }, "interruptDemo");
        thread.start();
        TimeUnit.SECONDS.sleep(1);
        thread.interrupt();
    }
}
```

其他的线程复位

除了通过 Thread.interrupted 方法对线程中断标识进行复位以外，还有一种被动复位的场景，就是对抛出InterruptedException 异常的方法，在InterruptedException 抛出之前，JVM 会先把线程的中断标识位清除，然后才会抛出 InterruptedException，这个时候如果调用 isInterrupted 方法，将会返回 false。

```
public class InterruptDemo {

    private static int i;
```

```
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(() -> {
            while (!Thread.currentThread().isInterrupted()) {
                try {
                    TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            System.out.println("Num:" + i);
        }, "interruptDemo");
        thread.start();
        TimeUnit.SECONDS.sleep(1);
        thread.interrupt();
        System.out.println(thread.isInterrupted());
    }
}
```

为什么要复位

Thread.interrupted()是属于当前线程的，是当前线程对外界中断信号的一个响应，表示自己已经得到了中断信号，但不会立刻中断自己，具体什么时候中断由自己决定，让外界知道在自身中断前，他的中断状态仍然是 false，这就是复位的原因。

线程的终止原理

```
    public void interrupt() {
        if (this != Thread.currentThread())
            checkAccess();

        synchronized (blockerLock) {
            Interruptible b = blocker;
            if (b != null) {
                interrupt0();           // Just to set the interrupt flag
                b.interrupt(this);
                return;
            }
        }
        interrupt0();
    }
```

这个方法里面，调用了 interrupt0()，这个方法在前面分析start 方法的时候见过，是一个 native 方法，同样，找到 jvm.cpp 文件，找到JVM_Interrupt 的定义，这个方法比较简单，直接调用了 Thread::interrupt(thr)这个方法，这个方法的定义在 Thread.cpp 文件中，Thread::interrupt 方法调用了 os::interrupt 方法，这个是调用平台的 interrupt 方法，这个方法的实现是在 os_*.cpp文件中，其中星号代表的是不同平台，因为 jvm 是跨平台的，所以对于不同的操作平台，线程的调度方式都是不一样的。以 os_linux.cpp 文件为例：set_interrupted(true)实际上就是调用 osThread.hpp 中的 set_interrupted()方法，在 osThread 中定义了一个成员属性 volatile jint _interrupted。

通过上面的代码分析可以知道，thread.interrupt()方法实际就是设置一个 interrupted 状态标识为 true、并且通过ParkEvent 的 unpark 方法来唤醒线程。

对于 synchronized 阻塞的线程，被唤醒以后会继续尝试获取锁，如果失败仍然可能被 park

在调用 ParkEvent 的 park 方法之前，会先判断线程的中断状态，如果为 true，会清除当前线程的中断标识

Object.wait 、 Thread.sleep 、 Thread.join 会 抛 出InterruptedException

为什么 Object.wait、Thread.sleep 和 Thread.join 都 会 抛 出InterruptedException?

这几个方法有一个共同点，都是属于阻塞的方法，而阻塞方法的释放会取决于一些外部的事件，但是阻塞方法可能因为等不到外部的触发事件而导致无法终止，所以它允许一个线程请求自己来停止它正在做的事情。当一个方法抛出 InterruptedException 时，它是在告诉调用者如果执行该方法的线程被中断，它会尝试停止正在做的事情并且通过抛出 InterruptedException 表示提前返回。所以，这个异常的意思是表示一个阻塞被其他线程中断了。然后，由于线程调用了 interrupt()中断方法，那么Object.wait、Thread.sleep 等被阻塞的线程被唤醒以后会通过 is_interrupted 方法判断中断标识的状态变化，如果发现中断标识为 true，则先清除中断标识，然后抛出InterruptedException。

InterruptedException 异常的抛出并不意味着线程必须终止，而是提醒当前线程有中断的操作发生，至于接下来怎么处理取决于线程本身，

直接捕获异常不做任何处理

将异常往外抛出

停止当前线程，并打印异常信息

总结：如果当前线程正处于wait，sleep，join状态，然后当前线程被打断，如果当前线程中断标识为true，清除当前线程的中端标识，并且当前线程会收到中断异常，。