并发的发展历史

真空管和穿孔打卡

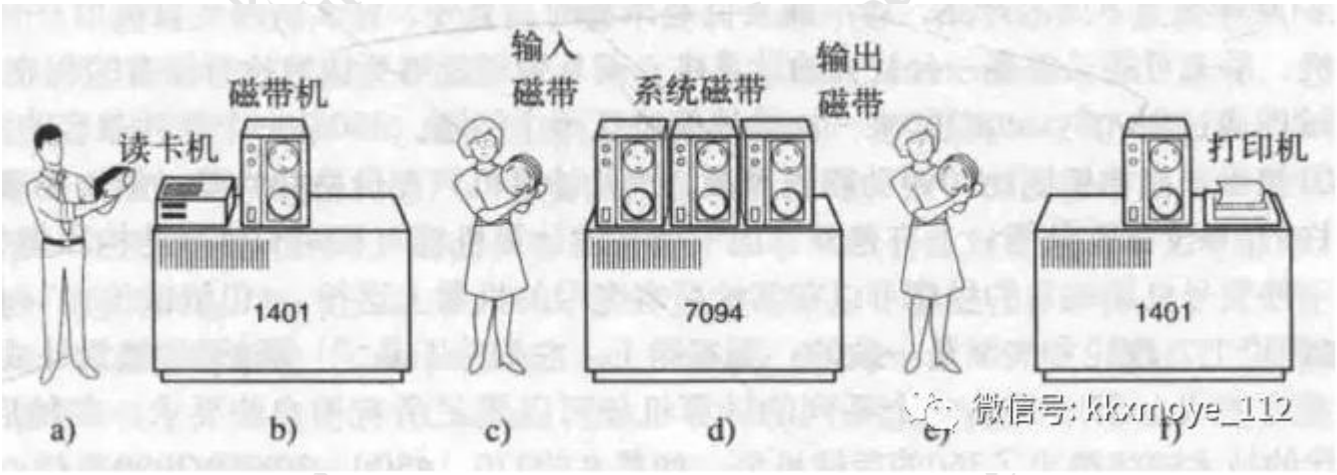
最早的计算机只能解决简单的数学运算问题，比如正弦、余弦等。运行方式：程序员首先把程序写到纸上，然后穿孔成卡片，再把卡片盒带入到专门的输入室。输入室会有专门的操作员将卡片的程序输入到计算机上。计算机运行完当前的任务以后，把计算结果从打印机上进行输出，操作员再把打印出来的结果送入到输出室，程序员就可以从输出室取到结果。然后，操作员再继续从已经送入到输入室的卡片盒中读入另一个任务重复上述的步骤。

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，计算机计算机和处理空闲状态 。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

批处理操作系统的运行方式：在输入室收集全部的作业，然后用一台比较便宜的计算机把它们读取到磁带上。然后把磁带输入到计算机，计算机通过读取磁带的指令来进行运算，最后把结果输出磁带上。批处理操作系统的好处在于，计算机会一直处于运算状态，合理的利用了计算机资源。(运行流程如下图所示)



a: 程序员把卡片拿到 1401 机

b: 1401 机把批处理作业读到磁带上

c: 操作员把输入磁带送到熬 7094 机

d: 7094 机进行计算

e: 操作员把输出磁带送到 1401 机

f:1401 机打印输出

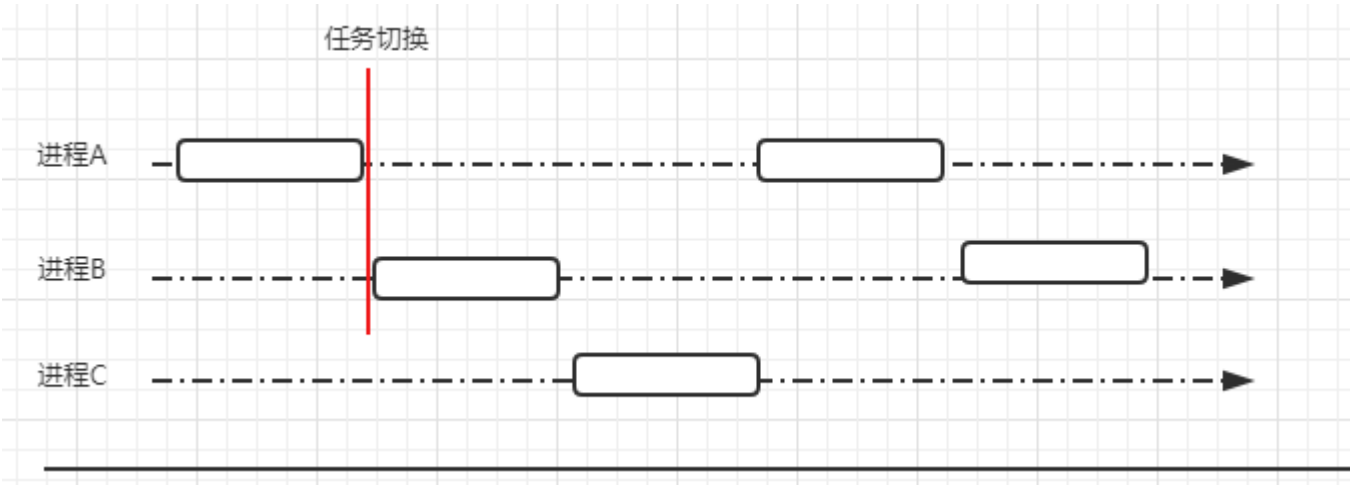
批处理操作系统虽然能够解决计算机的空闲问题，但是当

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%。

在这个时候，引入了第一个概念- 进程，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 时间片不端切换执行的。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

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。start()方法是一个 native 方法，它会启动一个新线程，并执行 run()方法。这种方式实现多线程很简单，通过自己的类直接 extend Thread，并复写 run()方法，就可以启动新线程并执行自己定义的 run()方法。

```
public class MyThread extends Thread {
      public void run() {
        System.out.println("MyThread.run()");
      }
}
MyThread myThread1 = new MyThread();
MyThread myThread2 = new MyThread();
myThread1.start();
```

```
myThread2.start();
```

实现 Runnable 接口创建线程

如果自己的类已经 extends 另一个类，就无法直接 extends Thread，此时，可以实现一个 Runnable 接口

```
public class MyThread extends OtherClass implements Runnable {
    public void run() {
      System.out.println("MyThread.run()");
    }
}
```

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 CallableDemo implements Callable<String> {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        ExecutorService executorService= Executors.newFixedThreadPool(1);
```

```
        CallableDemo callableDemo=new CallableDemo();
        Future<String>
future=executorService.submit(callableDemo);
        System.out.println(future.get());
        executorService.shutdown();
    }
    @Override
    public String call() throws Exception {
        int a=1;
        int b=2;
        System.out.println(a+b);
        return "执行结果:"+(a+b);
    }
}
```

多线程的实际应用场景

其实大家在工作中应该很少有场景能够应用多线程了，因为基于业务开发来说，很多使用异步的场景我们都通过分布式消息队列来做了。但并不是说多线程就不会被用到，你们如果有看一些框架的源码，会发现线程的使用无处不在

之前我应用得比较多的场景是在做文件跑批，每天会有一

些比如收益文件、对账文件，我们会有一个定时任务去拿到数据然后通过线程去处理

之前看 zookeeper 源码的时候看到一个比较有意思的异步责任链模式

Request

```
public class Request {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Request{" +
                "name='" + name + '\'' +
```

```
                '}';
    }
}
```

RequestProcessor

```
public interface RequestProcessor {

    void processRequest(Request request);
}
```

PrintProcessor

```
public class PrintProcessor extends Thread implements
RequestProcessor{

    LinkedBlockingQueue<Request> requests = new
LinkedBlockingQueue<Request>();

    private final RequestProcessor nextProcessor;

    public PrintProcessor(RequestProcessor nextProcessor) {
        this.nextProcessor = nextProcessor;
    }
```

```
    @Override
    public void run() {
        while (true) {
            try {
                Request request=requests.take();
                System.out.println("print
data:"+request.getName());
                nextProcessor.processRequest(request);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    //处理请求
    public void processRequest(Request request) {
        requests.add(request);
    }
}
```

SaveProcessor

```
public class SaveProcessor extends Thread implements
RequestProcessor{

    LinkedBlockingQueue<Request> requests = new
LinkedBlockingQueue<Request>();

    @Override
    public void run() {
        while (true) {
            try {
                Request request=requests.take();
                System.out.println("begin save request
info:"+request);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    //处理请求
    public void processRequest(Request request) {
        requests.add(request);
```

```
    }

}
```

Main

```
public class Main {

    PrintProcessor printProcessor;

    protected Main(){
        SaveProcessor saveProcessor=new SaveProcessor();
        saveProcessor.start();
        printProcessor=new PrintProcessor(saveProcessor);
        printProcessor.start();
    }

    private void doTest(Request request){
        printProcessor.processRequest(request);
    }

    public static void main(String[] args) {
        Request request=new Request();
```

```
        request.setName("Mic");
        new Main().doTest(request);
    }
}
```

Java 并发编程的基础

基本应用搞清楚以后，我们再来基于 Java 线程的基础切入，来逐步去深入挖掘线程的整体模型。

线程的生命周期

Java 线程既然能够创建，那么也势必会被销毁，所以线程是存在生命周期的，那么我们接下来从线程的生命周期开始去了解线程。

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”

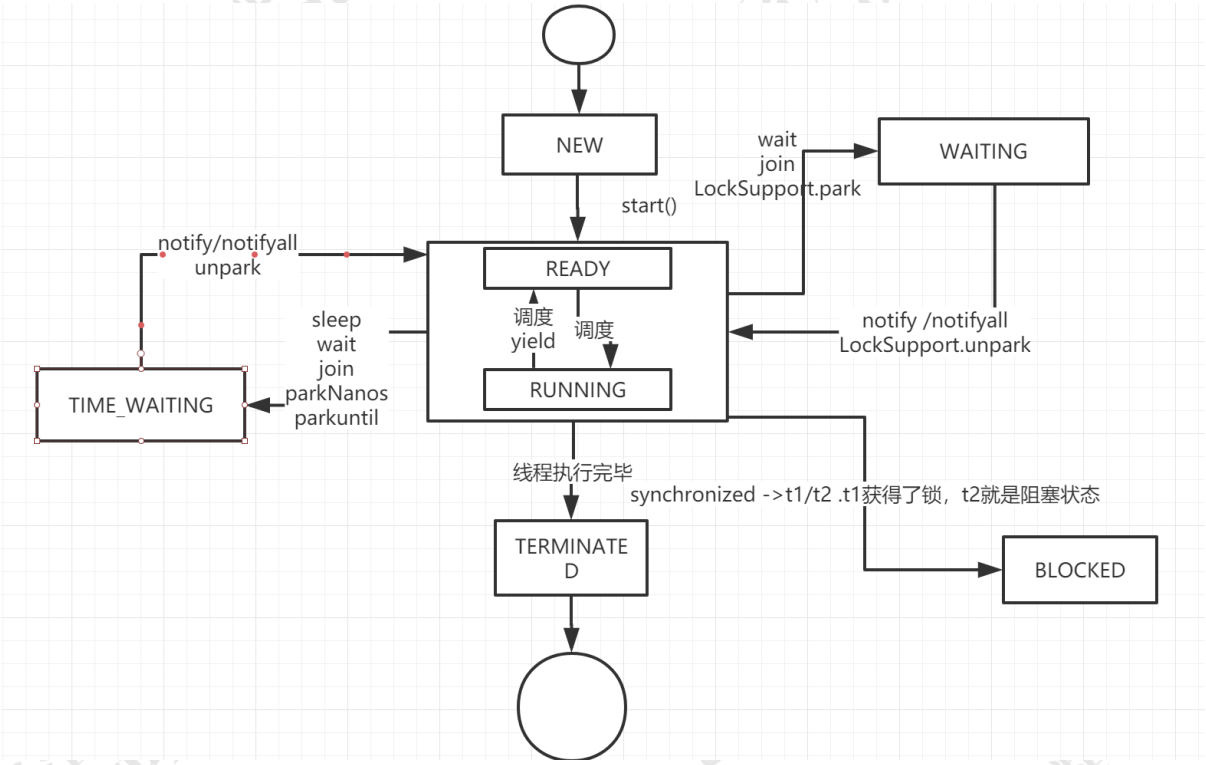
BLOCKED：阻塞状态，表示线程进入等待状态,也就是线程因为某种原因放弃了 CPU 使用权，阻塞也分为几种情况

- 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

编写案例代码

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

```
                }
            }
        }
}
```

启动一个线程前，最好为这个线程设置线程名称，因为这样在使用 jstack 分析程序或者进行问题排查时，就会给开发人员提供一些提示

显示线程的状态

- 运行该示例，打开终端或者命令提示符，键入“jps”，（JDK1.5 提供的一个显示当前所有 java 进程 pid 的命令）
- 根据上一步骤获得的 pid，继续输入 jstack pid（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

前面我们通过一些案例演示了线程的启动，也就是调用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

很多同学最早学习线程的时候会比较疑惑，启动一个线程为什么是调用 start 方法，而不是 run 方法，这做一个简单的分析，先简单看一下 start 方法的定义

```
public class Thread implements Runnable {
...
public synchronized void start() {
        /**
         * This method is not invoked for the main method thread or "system"
         * group threads created/set up by the VM. Any new functionality added
         * to this method in the future may have to also be added to the VM.
         *
         * A zero status value corresponds to state "NEW".
         */
        if (threadStatus != 0)
            throw new IllegalThreadStateException();
        /* Notify the group that this thread is about to be started
         * so that it can be added to the group's list of threads
         * and the group's unstarted count can be decremented. */
        group.add(this);
        boolean started = false;
        try {
            start0(); //注意这里
            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();//注意这里
...
```

我们看到调用 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，代码如下

```
public class Thread implements Runnable {
    /* Make sure registerNatives is the first thing <clinit> does. */
    private static native void registerNatives();
    static {
        registerNatives();
    }
```

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，以下是 Thread.c 的全部内容 http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/00cd9dc3c2b5/src/share/native/java/lang/Thread.c

```
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},
    {"setNativeName",    "(" STR ")V", (void *)&JVM_SetNativeThreadName},
};
#undef THD
#undef OBJ
#undef STE
#undef STR
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))
  JVMWrapper("JVM_StartThread");
...
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 1558 行的位置可以找到如下代码

```
JavaThread::JavaThread(ThreadFunction entry_point, size_t stack_sz) :
  Thread()
#if INCLUDE_ALL_GCS
  , _satb_mark_queue(&_satb_mark_queue_set),
  _dirty_card_queue(&_dirty_card_queue_set)
#endif // INCLUDE_ALL_GCS
{
  if (TraceThreadEvents) {
    tty->print_cr("creating thread %p", this);
  }
  initialize();
  _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 == &compiler_thread_entry ? os::compiler_thread :
                                                     os::java_thread;
  os::create_thread(this, thr_type, stack_sz);
  _safepoint_visible = false;
  // The _osthread may be NULL here because we ran out of memory (too many threads act
  // We need to throw and OutOfMemoryError - however we cannot do this here because th
  // may hold a lock and all locks must be unlocked before throwing the exception (thr
  // the exception consists of creating the exception object & initializing it, initia
  // will leave the VM via a JavaCall and then all locks must be unlocked).
  //
  // The thread is still suspended when we reach here. Thread must be explicit started
  // by creator! Furthermore, the thread must also explicitly be added to the Threads
  // by calling Threads:add. The reason why this is not done here, is because the thre
  // object must be fully initialized (take a look at JVM_Start)
}
```

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) {
  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);
  }
}
```

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

线程的启动过程大家都非常熟悉，但是如何终止一个线程呢？ 这是面试过程中针对 3 年左右的人喜欢问到的一个题目。

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。

线程通过检查资深是否被中断来进行相应，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 InterruptDemo {
    private static int i;
    public static void main(String[] args) throws InterruptedException {
        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 {
    private static int i;
    public static void
main(String[] args) throws
InterruptedException {
        Thread thread=new
Thread(()->{
            while(true){
```

```
if(Thread.currentThread().isInterr
upted()){

System.out.println("before:"+Threa
d.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

分别通过下面两个 demo 来演示复位的效果

```
public class InterruptDemo {
    private static int i;
    public static void main(String[] args) throws InterruptedException {
        Thread thread=new Thread(()->{
```

```
public class InterruptDemo {
    private static int i;
    public static void main(String[] args) throws InterruptedException {
        Thread thread=new Thread(()->{
```

```
while(!Thread.currentThread().isInterrupted()){
                i++;
            }

System.out.println("Num:"+i);
        },"interruptDemo");

thread.start();

TimeUnit.SECONDS.sleep(1);

thread.interrupt();

System.out.println(thread.isInterrupted());
```

```
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

我们来看一下 thread.interrupt()方法做了什么事情

```
public class Thread implements Runnable {
...
    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，这里就不再重复贴代码了，同样，我们找到 jvm.cpp 文件，找到 JVM_Interrupt 的定义

```
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 operate
  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 the
  // acquire of the lock
  JavaThread* thr = java_lang_Thread::thread(JNIHandles::resolve_non_null(jthread));
  if (thr != NULL) {
    Thread::interrupt(thr);
  }
JVM_END
```

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，代码如下

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

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。我们以 os_linux.cpp 文件为例

```
void os::interrupt(Thread* thread) {
  assert(Thread::current() == thread || Threads_lock->owned_by_self(),
    "possibility of dangling Thread pointer");
  //获取本地线程对象
  OSThread* osthread = thread->osthread();
  if (!osthread->interrupted()) {//判断本地线程对象是否为中断
    osthread->set_interrupted(true);//设置中断状态为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().
    //这里是内存屏障，这块在后续的文章中会剖析；内存屏障的目的是使得interrupted状态对其他线程立即
    OrderAccess::fence();
    //_SleepEvent相当于Thread.sleep，表示如果线程调用了sleep方法，则通过unpark唤醒
    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用于synchronized同步块和Object.wait()，这里相当于也是通过unpark进行唤醒
  ParkEvent * ev = thread->_ParkEvent ;
  if (ev != NULL) ev->unpark() ;
}
```

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。

1. 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
2. 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
3. Object.wait 、 Thread.sleep 、 Thread.join 会抛出 InterruptedException

这里给大家普及一个知识点，为什么 Object.wait、Thread.sleep 和 Thread.join 都会抛出 InterruptedException？你会发现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

需要注意的是，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，比如

1. 直接捕获异常不做任何处理
2. 将异常往外抛出
3. 停止当前线程，并打印异常信息

为了让大家能够更好的理解上面这段话，我们以 Thread.sleep 为例直接从 jdk 的源码中找到中断标识的清除以及异常抛出的方法代码

找到 is_interrupted() 方法，linux 平台中的实现在 os_linux.cpp 文件中，代码如下

```
bool os::is_interrupted(Thread* thread, bool clear_interrupted) {
  assert(Thread::current() == thread || Threads_lock->owned_by_self(),
    "possibility of dangling Thread pointer");
  OSThread* osthread = thread->osthread();
  bool interrupted = osthread->interrupted(); //获取线程的中断标识
  if (interrupted && clear_interrupted) {//如果中断标识为true
    osthread->set_interrupted(false);//设置中断标识为false
    // consider thread->_SleepEvent->reset() ... optional optimization
  }
  return interrupted;
}
```

找到 Thread.sleep 这个操作在 jdk 中的源码体现，怎么找?

相信如果前面大家有认真看的话，应该能很快找到，代码在 jvm.cpp 文件中

```
JVM_ENTRY(void, JVM_Sleep(JNIEnv* env, jclass threadClass, jlong millis))
  JVMWrapper("JVM_Sleep");
  if (millis < 0) {
    THROW_MSG(vmSymbols::java_lang_IllegalArgumentException(), "timeout value is negat
  }
  //判断并清除线程中断状态，如果中断状态为true,抛出中断异常
  if (Thread::is_interrupted (THREAD, true) && !HAS_PENDING_EXCEPTION) {
    THROW_MSG(vmSymbols::java_lang_InterruptedException(), "sleep interrupted");
  }
  // Save current thread state and restore it at the end of this block.
  // And set new thread state to SLEEPING.
  JavaThreadSleepState jtss(thread);
...
```

注意上面加了中文注释的地方的代码，先判断 is_interrupted 的状态，然后抛出一个 InterruptedException 异常。到此为止，我们就已经分析清楚了中断的整个流程。