

# Java 并发编程之美：线程相关的基础知识

## 前言

借用 Java 并发编程实践中的话：编写正确的程序并不容易，而编写正常的并发程序就更难了；相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却都还离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

本 Chat 作为 Java 并发编程之美系列的开篇，首先通过通俗易懂的方式先来和大家聊聊多线程并发编程线程有关基础知识（本文结合示例进行讲解，定会让你耳目一新），具体内容如下：

- 什么是线程？线程和进程的关系。
- 线程创建与运行。创建一个线程有那几种方式？有何区别？
- 线程通知与等待，多线程同步的基础设施。
- 线程的虚假唤醒，以及如何避免。
- 等待线程执行终止的 join 方法。想让主线程在子线程执行完毕后在做一点事情？
- 让线程睡眠的 sleep 方法，sleep 的线程会释放持有的锁？
- 线程中断。中断一个线程，被中断的线程会自己终止？
- 理解线程上下文切换。线程多了一定好？
- 线程死锁，以及如何避免。
- 守护线程与用户线程。当 main 函数执行完毕，但是还有用户线程存在的时候，JVM 进程会退出？

## 什么是线程

在讨论什么是线程前有必要先说下什么是进程，因为线程是进程中的一个实体，线程本身是不会独立存在的。进程是代码在数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，线程则是进程的一个执行路径，一个进程至少有一个线程，进程中的多个线程是共享进程的资源。

操作系统在分配资源时候是把资源分配给进程的，但是 CPU 资源就比较特殊，它是分派到线程的，因为真正要占用 CPU 运行的是线程，所以说线程是 CPU 分配的基本单位。

Java 中当我们启动 main 函数时候其实就启动了一个 JVM 的进程，而 main 函数所在线程就是这个进程中的一个线程，也叫做主线程。



如图一个进程中有多个线程，多个线程共享进程的堆和方法区资源，但是每个线程有自己的程序计数器，栈区域。

其中程序计数器是一块内存区域，用来记录线程当前要执行的指令地址，那么程序计数器为何要设计为线程私有的呢？前面说了线程是占用 CPU 执行的基本单位，而 CPU 一般是使用时间片轮转方式让线程轮询占用的，所以当前线程 CPU 时间片用完后，要让出 CPU，等下次轮到自己时候在执行，那么如何知道之前程序执行到哪里了？其实程序计数器就是为了记录该线程让出 CPU 时候的执行地址，待再次分配到时间片时候就可以从自己私有的计数器指定地址继续执行了。

另外每个线程有自己的栈资源，用于存储该线程的局部变量，这些局部变量是该线程私有的，其它线程是访问不了的，另外栈还用来存放线程的调用栈帧。

堆是一个进程中最大的一块内存，堆是被进程中的所有线程共享的，是进程创建时候分配的，堆里面主要存放使用 new 操作创建的对象实例。

方法区则是用来存放进程中的代码片段的，是线程共享的。

## 线程创建与运行

Java 中有三种线程创建方法，分别为实现 Runnable 接口的 run 方法、继承 Thread 类并重写 run 方法、使用 FutureTask 方式。

首先看下继承 Thread 方法的实现：

```

public class ThreadTest {

    //继承Thread类并重写run方法
    public static class MyThread extends Thread {

        @Override
        public void run() {

            System.out.println("I am a child thread");

        }

    }

    public static void main(String[] args) {

        // 创建线程
        MyThread thread = new MyThread();

        // 启动线程
        thread.start();

    }

}

```

如上代码 MyThread 类继承了 Thread 类，并重写了 run 方法，然后调用了线程的 start 方法启动了线程，当创建完 thread 对象后该线程并没有被启动执行。

当调用了 start 方法后才是真正启动了线程。其实当调用了 start 方法后线程并没有马上执行而是处于就绪状态，这个就绪状态是指该线程已经获取了除 CPU 资源外的其它资源，等获取 CPU 资源后才会真正处于运行状态。

当 run 方法执行完毕，该线程就处于终止状态了。使用继承方式好处是 run 方法内获取当前线程直接使用 this 就可以，无须使用 Thread.currentThread() 方法，不好的地方是 Java 不支持多继承，如果继承了 Thread 类那么就不能再继承其它类，另外任务与代码没有分离，当多个线程执行一样的任务时候需要多份任务代码，而 Runnable 则没有这个限制，下面看下实现 Runnable 接口的 run 方法方式：

```

public static class RunnableTask implements Runnable{

    @Override
    public void run() {
        System.out.println("I am a child thread");
    }

}

public static void main(String[] args) throws
InterruptedException{

    RunnableTask task = new RunnableTask();
}

```

```

        new Thread(task).start();
        new Thread(task).start();
    }

```

如上面代码，两个线程公用一个 task 代码逻辑，需要的话 RunnableTask 可以添加参数进行任务区分，另外 RunnableTask 可以继承其他类，但是上面两种方法都有一个缺点就是任务没有返回值，下面看最后一种是使用 FutureTask：

```

//创任务类，类似Runnable
public static class CallerTask implements Callable<String>{

    @Override
    public String call() throws Exception {

        return "hello";
    }

}

public static void main(String[] args) throws
InterruptedException {
    // 创建异步任务
    FutureTask<String> futureTask = new FutureTask<>(new
CallerTask());
    //启动线程
    new Thread(futureTask).start();
    try {
        //等待任务执行完毕，并返回结果
        String result = futureTask.get();
        System.out.println(result);
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}

```

**注：**每种方式都有自己的优缺点，应该根据实际场景进行选择。

## 线程通知与等待

Java 中 Object 类是所有类的父类，鉴于继承机制，Java 把所有类都需要的方法放到了 Object 类里面，其中就包含本节要讲的通知等待系列函数,这些通知等待函数是组成并发包中线程同步组件的基础。

下面讲解下 Object 中关于线程同步的通知等待函数。

void wait() 方法

首先谈下什么是共享资源，所谓共享资源是说该资源被多个线程共享，多个线程都可以去访问或者修改的资源。另外本文当讲到的共享对象就是共享资源。

当一个线程调用一个共享对象的 wait() 方法时候，调用线程会被阻塞挂起，直到下面几个事情之一发生才返回：

1. 其它线程调用了该共享对象的 notify() 或者 notifyAll() 方法；
2. 其它线程调用了该线程的 interrupt() 方法设置了该线程的中断标志，该线程会抛出 InterruptedException 异常返回。

另外需要注意的是如果调用 wait() 方法的线程没有事先获取到该对象的监视器锁，则调用 wait() 方法时候调用线程会抛出 IllegalMonitorStateException 异常。

那么一个线程如何获取到一个共享变量的监视器那？

- (1) 执行使用 synchronized 同步代码块时候，使用该共享变量作为参数：

```
synchronized (共享变量) {  
    //doSomething  
}
```

- (2) 调用该共享变量的方法，并且该方法使用了 synchronized 修饰：

```
synchronized void add(int a,int b){  
    //doSomething  
}
```

另外需要注意的是一个线程可以从挂起状态变为可以运行状态（也就是被唤醒）即使该线程没有被其它线程调用 notify(), notifyAll() 进行通知，或者被中断，或者等待超时，这就是所谓的虚假唤醒。

虽然虚假唤醒在应用实践中很少发生，但是还是需要防范于未然的，做法就是不停的去测试该线程被唤醒的条件是否满足，不满足则继续等待，也就是说在一个循环中去调用 wait() 方法进行防范，退出循环的条件是条件满足了唤醒该线程。

```
synchronized (obj) {  
    while (条件不满足){  
        obj.wait();  
    }  
}
```

如上代码是经典的调用共享变量 wait() 方法的实例，首先通过同步块获取 obj 上面的监视器锁，然后通过 while 循环内调用 obj 的 wait() 方法。

下面从生产者消费者例子来加深理解，如下面代码是一个生产者的例子，其中 queue 为共享变量，生产者线程在调用 queue 的 wait 方法前，通过使用 synchronized 关键字拿到了该共享变量 queue 的监视器，所以调用 wait() 方法才不会抛出 IllegalMonitorStateException 异常，如果当前队列没有空闲容量则会调用 queue 的 wait() 挂起当前线程，这里使用循环就是为了避免上面说的虚假唤醒问题，这里假如当前线程虚假唤醒了，但是队列还是没有空余容量的话，当前线程还是会调用 wait() 把自己挂起。

```
//生产线程
synchronized (queue) {

    //消费队列满，则等待队列空闲
    while (queue.size() == MAX_SIZE) {
        try {
            //挂起当前线程，并释放通过同步块获取的queue上面的锁，让消费线
            程可以获取该锁，然后获取队列里面元素
            queue.wait();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    //空闲则生成元素，并通知消费线程
    queue.add(ele);
    queue.notifyAll();

}
}
```

```
//消费线程
synchronized (queue) {

    //消费队列为空
    while (queue.size() == 0) {
        try
            //挂起当前线程，并释放通过同步块获取的queue上面的锁，让生产
            线可以获取该锁，生产元素放入队列
            queue.wait();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    //消费元素，并通知唤醒生产线程
    queue.take();
    queue.notifyAll();

}
}
```

另外当一个线程调用了共享变量的 wait() 方法后该线程会被挂起，同时该线程会暂时释放对该共享变量监视器的持有，直到另外一个线程调用了共享变量的 notify() 或者 notifyAll() 方法才有可能重新获取到该共享变量的监视器的持有权（这里说有可能，是因为考虑到多个线程第一次都调用了 wait() 方法，所以多个线程会竞争持有该共享变量的监视器）。

借用上面这个例子来讲解下调用共享变量 wait() 方法后当前线程会释放持有的共享变量的锁的理解。

如上代码假如生产线程 A 首先通过 synchronized 获取到了 queue 上的锁，那么其它生产线程和所有消费线程都会被阻塞，线程 A 获取锁后发现当前队列已满会调用 queue.wait() 方法阻塞自己，然后会释放获取的 queue 上面的锁，这里考虑下为何要释放该锁？如果不释放，由于其它生产线程和所有消费线程已经被阻塞挂起，而线程 A 也被挂起，这就处于了死锁状态。这里线程 A 挂起自己后释放共享变量上面的锁就是为了打破死锁必要条件之一的持有并等待原则。关于死锁下面章节会有讲到，线程 A 释放锁后其它生产线程和所有消费线程中会有一个线程获取 queue 上的锁进而进入同步块，这就打破了死锁。

最后再举一个例子说明当一个线程调用共享对象的 wait() 方法被阻塞挂起后，如果其它线程中断了该线程，则该线程会抛出 InterruptedException 异常后返回：

```
public class WaitNotifyInterrupt {

    static Object obj = new Object();

    public static void main(String[] args) throws
    InterruptedException {

        //创建线程
        Thread threadA = new Thread(new Runnable() {
            public void run() {
                try {
                    System.out.println("---begin---");
                    //阻塞当前线程
                    obj.wait();
                    System.out.println("---end---");

                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        threadA.start();

        Thread.sleep(1000);
```

```

        System.out.println("---begin interrupt threadA---");
        threadA.interrupt();
        System.out.println("---end interrupt threadA---");
    }
}

```

运行上面代码输出为：

```

<terminated> WaitNotifyInterupt [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (2017年12月12日 14:12:12)
---begin---
Exception in thread "Thread-0" java.lang.IllegalMonitorStateException
    at java.lang.Object.wait(Native Method)
    at java.lang.Object.wait(Object.java:502)
    at com.zlx.con.program.example.WaitNotifyInterupt$1.run(WaitNotifyInterupt.java:16)
    at java.lang.Thread.run(Thread.java:745)
---begin interrupt threadA---
---end interrupt threadA---
|

```

如上代码 threadA 调用了共享对 obj 的 wait () 方法后阻塞挂起了自己，然后主线程在休眠 1s 后中断了 threadA 线程，可知中断后 threadA 在 obj.wait() 处抛出了 java.lang.IllegalMonitorStateException 异常后返回后终止。

### void wait(long timeout) 方法

该方法相比 wait() 方法多一个超时参数，不同在于如果一个线程调用了共享对象的该方法挂起后，如果没有在指定的 timeout ms 时间内被其它线程调用该共享变量的 notify() 或者 notifyAll() 方法唤醒，那么该函数还是会因为超时而返回。

需要注意的是如果在调用该函数时候 timeout 传递了负数会抛出 IllegalArgumentException 异常。

### void wait(long timeout, int nanos) 方法

内部是调用 wait(long timeout)，如下代码：只是当 nanos>0 时候让参数一递增1。

```

    public final void wait(long timeout, int nanos) throws
        InterruptedException {
        if (timeout < 0) {
            throw new IllegalArgumentException("timeout value is
negative");
        }

        if (nanos < 0 || nanos > 999999) {
            throw new IllegalArgumentException(
                "nanosecond timeout value out of
range");
        }
    }

```



```

        if (nanos > 0) {
            timeout++;
        }

        wait(timeout);
    }

```

## void notify() 方法

一个线程调用共享对象的 notify() 方法后，会唤醒一个在该共享变量上调用 wait 系列方法后被挂起的线程，一个共享变量上可能会有多个线程在等待，具体唤醒哪一个等待的线程是随机的。

另外被唤醒的线程不能马上从 wait 返回继续执行，它必须获取了共享对象的监视器后才可以返回，也就是唤醒它的线程释放了共享变量上面的监视器锁后，被唤醒它的线程也不一定会获取到共享对象的监视器，这是因为该线程还需要和其它线程一块竞争该锁，只有该线程竞争到了该共享变量的监视器后才可以继续执行。

类似 wait 系列方法，只有当前线程已经获取到了该共享变量的监视器锁后，才可以调用该共享变量的 notify() 方法，否则会抛出 IllegalMonitorStateException 异常。

## void notifyAll() 方法

不同于 notify() 方法在共享变量上调用一次就会唤醒在该共享变量上调用 wait 系列方法被挂起的一个线程，notifyAll() 则会唤醒所有在该共享变量上由于调用 wait 系列方法而被挂起的线程。

最后本小节最后讲一个例子来说明 notify() 和 notifyAll() 的具体含义和一些需要注意的地方，代码实例如下：

```

private static volatile Object resourceA = new Object();

public static void main(String[] args) throws
InterruptedException {

    // 创建线程
    Thread threadA = new Thread(new Runnable() {
        public void run() {

            // 获取resourceA共享资源的监视器锁
            synchronized (resourceA) {

                System.out.println("threadA get resourceA lock");
                try {

                    System.out.println("threadA begin wait");

```

```

        resourceA.wait();
        System.out.println("threadA end wait");

    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

});

// 创建线程
Thread threadB = new Thread(new Runnable() {
    public void run() {

        synchronized (resourceA) {
            System.out.println("threadB get resourceA lock");
            try {

                System.out.println("threadB begin wait");
                resourceA.wait();
                System.out.println("threadB end wait");

            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
});

// 创建线程
Thread threadC = new Thread(new Runnable() {
    public void run() {

        synchronized (resourceA) {

            System.out.println("threadC begin notify");
            resourceA.notifyAll();
        }
    }
});

// 启动线程
threadA.start();
threadB.start();

Thread.sleep(1000);
threadC.start();

// 等待线程结束

```

```

        threadA.join();
        threadB.join();
        threadC.join();
        System.out.println("main over");
    }

```

输出结果：

```

WaitNotifyAllTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home
threadA get resourceA lock
threadA begin wait
threadB get resourceA lock
threadB begin wait
threadC begin notify
threadA end wait

```

如上代码开启了三个线程，其中线程 A 和 B 分别调用了共享资源 resourceA 的 wait() 方法，线程 C 则调用了 notify() 方法。

这里启动线程 C 前首先调用 sleep 方法让主线程休眠 1s，目的是让线程 A 和 B 全部执行到调用 wait 方法后在调用线程 C 的 notify 方法。

这个例子企图希望在线程 A 和线程 B 都因调用共享资源 resourceA 的 wait() 方法而被阻塞后，线程 C 在调用 resourceA 的 notify() 方法，希望可以唤醒线程 A 和线程 B，但是从执行结果看只有一个线程 A 被唤醒了，线程 B 没有被唤醒，

从结果看线程调度器这次先调度了线程 A 占用 CPU 来运行，线程 A 首先获取 resourceA 上面的锁，然后调用 resourceA 的 wait() 方法挂起当前线程并释放获取到的锁，然后线程 B 获取到 resourceA 上面的锁并调用了 resourceA 的 wait()，此时线程 B 也被阻塞挂起并释放了 resourceA 上的锁。

线程 C 休眠结束后在共享资源 resourceA 上调用了 notify() 方法，则会激活 resourceA 的阻塞集合里面的一个线程，这里激活了线程 A，所以线程 A 调用的 wait() 方法返回了，线程 A 执行完毕。而线程 B 还处于阻塞状态。

如果把线程 C 里面调用的 notify() 改为调用 notifyAll() 而执行结果如下：

```

<terminated> WaitNotifyAllTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/t
threadA get resourceA lock
threadA begin wait
threadB get resourceA lock
threadB begin wait
threadC begin notify
threadB end wait
threadA end wait
main over

```

可知线程 A 和线程 B 被挂起后，线程 C 调用 notifyAll() 函数会唤醒在 resourceA 等待的所有线程，这里线程 A 和线程 B 都会被唤醒，只是线程 B 先获取到 resourceA 上面的锁然后

从 wait() 方法返回，等线程 B 执行完毕后，线程 A 又获取了 resourceA 上面的锁，然后从 wait() 方返回，当线程 A 执行完毕，主线程就返回后，然后打印输出。

**注：**在调用具体共享对象的 wait 或者 notify 系列函数前要先获取共享对象的锁；另外通知和等待是实现线程同步的原生方法，理解它们的协作功能很有必要；最后由于线程虚假唤醒的存在，一定要使用循环检查的方式。

## 等待线程执行终止的 join 方法

在项目实践时候经常会遇到一个场景，就是需要等待某几件事情完成后才能继续往下执行，比如多个线程去加载资源，当多个线程全部加载完毕后在汇总处理，Thread 类中有个静态的 join 方法就可以做这个事情，前面介绍的等待通知方法是属于 Object 类的，而 join 方法则是直接在 Thread 类里面提供的，join 是无参，返回值为 void 的方法。下面看一个简单的例子来介绍 join 的使用：

```
public static void main(String[] args) throws
InterruptedException {
    Thread threadOne = new Thread(new Runnable() {

        @Override
        public void run() {

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println("child threadOne over!");

        }
    });

    Thread threadTwo = new Thread(new Runnable() {

        @Override
        public void run() {

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println("child threadTwo over!");
```

```

        }
    });

    //启动子线程
    threadOne.start();
    threadTwo.start();

    System.out.println("wait all child thread over!");

    //等待子线程执行完毕，返回
    threadOne.join();
    threadTwo.join();

    System.out.println("all child thread over!");

}

```

如代码主线程里面启动了两个子线程，然后在分别调用了它们的 join() 方法，那么主线程首先会阻塞到 threadOne.join() 方法，等 threadOne 执行完毕后返回，threadOne 执行完毕后 threadOne.join() 就会返回，然后主线程调用 threadTwo.join() 后再次被阻塞，等 threadTwo 执行完毕后主线程也就返回了。这里只是为了演示 join 的作用，对应这类需求后面会讲的 CountdownLatch 是不错选择。

另外线程 A 调用线程 B 的 join 方法后会被阻塞，当其它线程调用了线程 B 的 interrupt() 方法中断了线程 B 时候，线程 B 会抛出 InterruptedException 异常而返回，下面通过一个例子来加深理解：

```

    public static void main(String[] args) throws
    InterruptedException {

        //线程one
        Thread threadOne = new Thread(new Runnable() {

            @Override
            public void run() {

                System.out.println("threadOne begin run!");
                for (;;) {

                }

            }

        });

        //获取主线程
        final Thread mainThread = Thread.currentThread();

        //线程two
        Thread threadTwo = new Thread(new Runnable() {

```

```

        @Override
        public void run() {
            //休眠1s
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //中断主线程
            mainThread.interrupt();
        }
    });

    // 启动子线程
    threadOne.start();

    //延迟1s启动线程
    threadTwo.start();

    try{//等待线程one执行结束
        threadOne.join();

    }catch(InterruptedException e){
        System.out.println("main thread:" + e);
    }

}

```

输出结果：

```

JoinInterruptedExceptionTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/C
threadOne begin run!
main thread: java.lang.InterruptedException

```

如上代码 threadOne 线程里面执行死循环，主线程调用 threadOne 的 join 方法阻塞自己等待线程 threadOne 执行完毕，待 threadTwo 休眠 1s 后会调用主线程的 interrupt() 方法设置主线程的中断标志。

从结果看主线程中 threadOne.join() 处会抛出 InterruptedException 异常而返回。这里需要注意的是 threadTwo 里面调用的是主线程的 interrupt(), 而不是线程 threadOne 的。

**注：**由于 CountDownLatch 功能比 join 更丰富，所以项目实践中一般使用 CountDownLatch，关于 CountDownLatch，后面 Chat 《Java 并发编程之美高级篇》会有具体讲解。

## 让线程睡眠的 sleep 方法

Thread 类中有一个静态的 sleep 方法，当一个执行中的线程调用了 Thread 的 sleep 方法后，调用线程会暂时让出指定时间的执行权，也就是这期间不参与 CPU 的调度，但是该线程所拥有的监视器资源，比如锁还是持有不让出的。当指定的睡眠时间到了该函数会正常返回，线程就处于就绪状态，然后参与 CPU 的调度，当获取到了 CPU 资源就可以继续运行了。如果在睡眠期间其它线程调用了该线程的 interrupt() 方法中断了该线程，该线程会在调用 sleep 的地方抛出 InterruptedException 异常返回。

首先看一个例子来说明线程在睡眠时候拥有的监视器资源不会被释放是什么意思：

```
public class SleepTest2 {

    // 创建一个独占锁
    private static final Lock lock = new ReentrantLock();

    public static void main(String[] args) throws
    InterruptedException {

        // 创建线程A
        Thread threadA = new Thread(new Runnable() {
            public void run() {
                // 获取独占锁
                lock.lock();
                try {
                    System.out.println("child threadA is in
sleep");

                    Thread.sleep(10000);

                    System.out.println("child threadA is in
awaked");

                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    // 释放锁
                    lock.unlock();
                }
            }
        });

        // 创建线程B
        Thread threadB = new Thread(new Runnable() {
            public void run() {
                // 获取独占锁
                lock.lock();
                try {
                    System.out.println("child threadB is in
sleep");

                    Thread.sleep(10000);
```

```

        System.out.println("child threadB is in
awaked");

        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            // 释放锁
            lock.unlock();
        }
    }
});

// 启动线程
threadA.start();
threadB.start();

}

}

```

执行结果：

```

<terminated> SleepTest2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/b
child threadA is in sleep
child threadA is in awaked
child threadB is in sleep
child threadB is in awaked

```

---

如上代码首先创建了一个独占锁，然后创建了两个线程，每个线程内部先获取锁，然后睡眠，睡眠结束后会释放锁。

首先无论你执行多少遍上面的代码都是先输出线程 A 的打印或者先输出线程 B 的打印，不会存在线程 A 和线程 B 交叉打印的情况。

从执行结果看线程 A 先获取了锁，那么线程 A 会先打印一行，然后调用 sleep 让自己沉睡 10s，在线程 A 沉睡的这 10s 内那个独占锁 lock 还是线程 A 自己持有的，线程 B 会一直阻塞直到线程 A 醒过来后执行 unlock 释放锁。

下面在来看下当一个线程处于睡眠时候如果另外一个线程中断了它，会不会在调用 sleep 处抛出异常。

```

public static void main(String[] args) throws
InterruptedException {

    //创建线程
    Thread thread = new Thread(new Runnable() {
        public void run() {

            try {

```



```

        System.out.println("child thread is in
sleep");

        Thread.sleep(10000);
        System.out.println("child thread is in
awaked");

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

});

//启动线程
thread.start();

//主线程休眠2s
Thread.sleep(2000);

//主线程中断子线程
thread.interrupt();
}

```

执行结果：

```

<terminated> SleepTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/jav
child thread is in sleep
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at com.zlx.con.program.example.SleepTest$1.run(SleepTest.java:14)
    at java.lang.Thread.run(Thread.java:745)

```

如上代码在子线程睡眠期间主线程中断了它，所以子线程在调用 sleep 处抛出了 InterruptedException 异常。

**注：**sleep 方法只是会让调用线程暂时让出指定时间的 CPU 执行权，但是该线程所拥有的监视器资源，比如锁还是持有不让出的。

## 线程中断

Java 中线程中断是一种线程间协作模式，通过设置线程的中断标志并不能直接终止该线程的执行，而是需要被中断的线程根据中断状态自行处理。

- void interrupt() 方法

中断线程，例如当线程 A 运行时，线程 B 可以调用线程 A 的 interrupt() 方法来设置线程 A 的中断标志为 true 并立即返回。设置标志仅仅是设置标志，线程 A 并没有实际被中断，

会继续往下执行的。如果线程 A 因为调用了 wait 系列函数或者 join 方法或者 sleep 函数而被阻塞挂起，这时候线程 B 调用了线程 A 的 interrupt() 方法，线程 A 会在调用这些方法的地方抛出 InterruptedException 异常而返回。

- boolean isInterrupted()

检测当前线程是否被中断，如果是返回 true，否者返回 false。

```
public boolean isInterrupted() {  
    //传递false，说明不清除中断标志  
    return isInterrupted(false);  
}
```

- boolean interrupted()

检测当前线程是否被中断，如果是返回 true，否者返回 false，与 isInterrupted 不同的是该方法如果发现当前线程被中断后会清除中断标志，并且该函数是 static 方法，可以通过 Thread 类直接调用。另外从下面代码可以知道 interrupted() 内部是获取当前调用线程的中断标志而不是调用 interrupted() 方法的实例对象的中断标志。

```
public static boolean interrupted() {  
    //清除中断标志  
    return currentThread().isInterrupted(true);  
}
```

下面看一个线程使用 Interrupted 优雅退出的经典使用例子，代码如下：

```
public void run() {  
    try {  
        ....  
        //线程退出条件  
        while (!Thread.currentThread().isInterrupted() && more  
work to do) {  
            // do more work;  
        }  
    } catch (InterruptedException e) {  
        // thread was interrupted during sleep or wait  
    }  
    finally {  
        // cleanup, if required  
    }  
}
```

下面看一个根据中断标志判断线程是否终止的例子：

```

public static void main(String[] args) throws
InterruptedException {

    Thread thread = new Thread(new Runnable() {

        @Override
        public void run() {

            //如果当前线程被中断则退出循环
            while (!Thread.currentThread().isInterrupted())

                System.out.println(Thread.currentThread() + "
hello");
        }
    });

    //启动子线程
    thread.start();

    //主线程休眠1s，以便中断前让子线程输出点东西
    Thread.sleep(1);

    //中断子线程
    System.out.println("main thread interrupt thread");
    thread.interrupt();

    //等待子线程执行完毕
    thread.join();
    System.out.println("main is over");

}

```

输出结果：

```

<terminated> MyThread [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.
Thread[Thread-0,5,main] hello
main thread interrupt thread
Thread[Thread-0,5,main] hello
main is over

```

如上代码子线程 thread 通过检查当前线程中断标志来控制是否退出循环，主线程在休眠 1s 后调用 thread 的 interrupt() 方法设置了中断标志，所以线程 thread 退出了循环。

**注：**中断一个线程仅仅是设置了该线程的中断标志，也就是设置了线程里面的一个变量的值，本身是不能终止当前线程运行的，一般程序里面是检查这个标志的状态来判断是否需要终止当前线程。

## 理解线程上下文切换

在多线程编程中，线程个数一般都大于 CPU 个数，而每个 CPU 同一时刻只能被一个线程使用，为了让用户感觉多个线程是在同时执行，CPU 资源的分配采用了时间片轮转的策略，也就是给每个线程分配一个时间片，在时间片内占用 CPU 执行任务。当前线程的时间片使用完毕后当前就会处于就绪状态并让出 CPU 让其它线程占用，这就是上下文切换，从当前线程的上下文切换到了其它线程。

那么就有一个问题让出 CPU 的线程等下次轮到自己占有 CPU 时候如何知道之前运行到哪里了？所以在切换线程上下文时候需要保存当前线程的执行现场，当再次执行时候根据保存的执行现场信息恢复执行现场。

线程上下文切换时机：

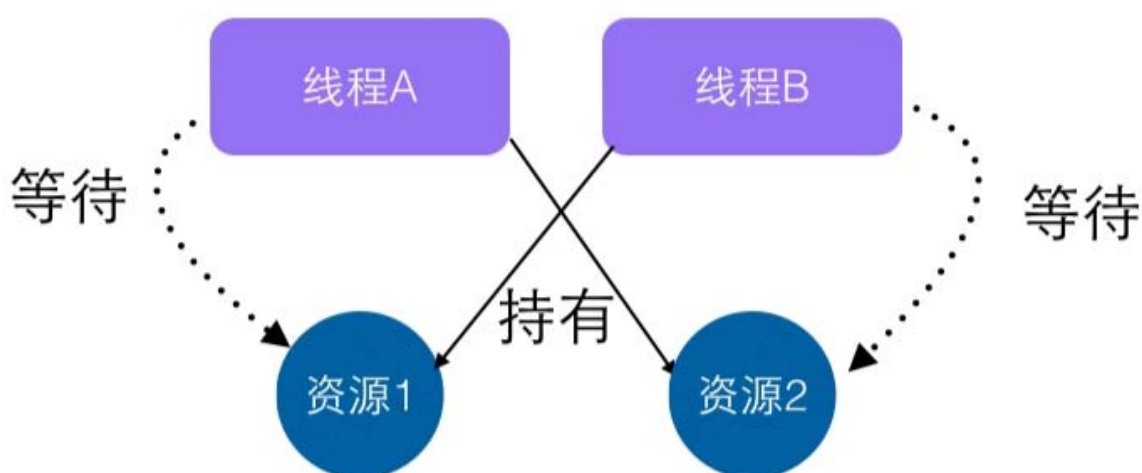
- 当前线程的 CPU 时间片使用完毕处于就绪状态时候；
- 当前线程被其它线程中断时候。

**注：**由于线程切换是有开销的，所以并不是开的线程越多越好，比如如果机器是4核心的，你开启了100个线程，那么同时执行的只有4个线程，这100个线程会来回切换线程上下文来共享这四个 CPU。

## 线程死锁

什么是线程死锁

死锁是指两个或两个以上的线程在执行过程中，因争夺资源而造成的互相等待的现象，在无外力作用的情况下，这些线程会一直相互等待而无法继续运行下去。



如上图，线程 A 已经持有了资源1的同时还想要资源2，线程 B 在持有资源2的时候还想要资源1，所以线程1和线程2就相互等待对方已经持有的资源，就进入了死锁状态。

那么产生死锁的原因都有哪些，学过操作系统的应该都知道死锁的产生必须具备以下四个必要条件。

- 互斥条件：指线程对已经获取到的资源进行排它性使用，即该资源同时只由一个线程占用。如果此时还有其它进行请求获取该资源，则请求者只能等待，直至占有资源的线程用毕释放。
- 请求并持有条件：指一个线程已经持有了至少一个资源，但又提出了新的资源请求，而新资源已被其其它线程占有，所以当前线程会被阻塞，但阻塞的同时并不释放自己已经获取的资源。
- 不可剥夺条件：指线程获取到的资源在自己使用完之前不能被其它线程抢占，只有在自己使用完毕后由自己释放。
- 环路等待条件：指在发生死锁时，必然存在一个线程——资源的环形链，即线程集合{T0, T1, T2, ..., Tn}中的T0正在等待一个T1占用的资源；T1正在等待T2占用的资源，.....Tn正在等待已被T0占用的资源。

下面通过一个案例来说明线程死锁：

```
public class DeadLockTest2 {

    // 创建资源
    private static Object resourceA = new Object();
    private static Object resourceB = new Object();

    public static void main(String[] args) {

        // 创建线程A
        Thread threadA = new Thread(new Runnable() {
            public void run() {
                synchronized (resourceA) {
                    System.out.println(Thread.currentThread() + "
get ResourceA");

                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    System.out.println(Thread.currentThread() +
"waiting get ResourceB");
                    synchronized (resourceB) {
                        System.out.println(Thread.currentThread()
+ "get ResourceB");
                    }
                }
            }
        });

        // 创建线程B
        Thread threadB = new Thread(new Runnable() {
            public void run() {
                synchronized (resourceB) {
```

```

        System.out.println(Thread.currentThread() + "
get ResourceB");

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(Thread.currentThread() +
"waiting get ResourceA");
        synchronized (resourceA) {
            System.out.println(Thread.currentThread()
+ "get ResourceA");
        }
    };
}
});

// 启动线程
threadA.start();
threadB.start();
}
}

```

输出结果：

```

DeadLockTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Hom
Thread[Thread-0,5,main] get ResourceA
Thread[Thread-1,5,main] get ResourceB
Thread[Thread-0,5,main]waiting get ResourceB
Thread[Thread-1,5,main]waiting get ResourceA

```

下面分析下代码和结果，其中 Thread-0 是线程 A，Thread-1 是线程 B，代码首先创建了两个资源，并创建了两个线程。

从输出结果可以知道线程调度器先调度了线程 A，也就是把 CPU 资源让给了线程 A，线程 A 调用了 getResourceA() 方法，方法里面使用 synchronized(resourceA) 方法获取到了 resourceA 的监视器锁，然后调用 sleep 函数休眠 1s，休眠 1s 是为了保证线程 A 在执行 getResourceB 方法前让线程 B 抢占到 CPU 执行 getResourceB 方法。

线程 A 调用了 sleep 期间，线程 B 会执行 getResourceB 方法里面的 synchronized(resourceB)，代表线程 B 获取到了 objectB 对象的监视器锁资源，然后调用 sleep 函数休眠 1S。

好了，到了这里线程 A 获取到了 objectA 的资源，线程 B 获取到了 objectB 的资源。线程 A 休眠结束后会调用 getResourceB 方法企图获取到 objectB 的资源，而 ObjectB 资源被线程 B 所持有，所以线程 A 会被阻塞而等待。而同时线程 B 休眠结束后会调用 getResourceA 方法企图获取到 objectA 上的资源，而资源 objectA 已经被线程 A 持有，所以线程 A 和 B 就陷入了相互等待的状态也就产生了死锁。

下面从产生死锁的四个条件来谈谈本案例如何满足了四个条件。

首先资源 resourceA 和 resourceB 都是互斥资源，当线程 A 调用 synchronized(resourceA) 获取到 resourceA 上的监视器锁后释放前，线程 B 在调用 synchronized(resourceA) 尝试获取该资源会被阻塞，只有线程 A 主动释放该锁，线程 B 才能获得，这满足了资源互斥条件。

线程 A 首先通过 synchronized(resourceA) 获取到 resourceA 上的监视器锁资源，然后通过 synchronized(resourceB) 等待获取到 resourceB 上的监视器锁资源，这就构造了持有并等待。

线程 A 在获取 resourceA 上的监视器锁资源后，不会被线程 B 掠夺走，只有线程 A 自己主动释放 resourceA 的资源时候，才会放弃对该资源的持有权，这构造了资源的不可剥夺条件。

线程 A 持有 objectA 资源并等待获取 objectB 资源，而线程 B 持有 objectB 资源并等待 objectA 资源，这构成了循环等待条件。

所以线程 A 和 B 就形成了死锁状态。

## 如何避免线程死锁

要想避免死锁，需要破坏构造死锁必要条件的至少一个即可，但是学过操作系统童鞋应该都知道目前只有持有并等待和循环等待是可以被破坏的。

造成死锁的原因其实和申请资源的顺序有很大关系，使用资源申请的有序性原则就可以避免死锁，那么什么是资源的有序性呢，先看一下对上面代码的修改：

```
// 创建线程B
Thread threadB = new Thread(new Runnable() {
    public void run() {
        synchronized (resourceA) {
            System.out.println(Thread.currentThread() + "
get ResourceB");

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println(Thread.currentThread() +
"waiting get ResourceA");
            synchronized (resourceB) {
                System.out.println(Thread.currentThread()
+ "get ResourceA");
            }
        }
    }
});
```



```
    }  
});
```

输出结果：

```
<terminated> DeadLockTest2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java  
Thread[Thread-0,5,main] get ResourceA  
Thread[Thread-0,5,main]waiting get ResourceB  
Thread[Thread-0,5,main]get ResourceB  
Thread[Thread-1,5,main] get ResourceB  
Thread[Thread-1,5,main]waiting get ResourceA  
Thread[Thread-1,5,main]get ResourceA  
,
```

如上代码可知修改了线程 B 中获取资源的顺序和线程 A 中获取资源顺序一致，其实资源分配有序性就是指假如线程 A 和 B 都需要资源 1, 2, 3.....n 时候，对资源进行排序，线程 A 和 B 只有在获取到资源 n-1 时候才能去获取资源 n。

**注：**编写并发程序，多个线程进行共享多个资源时候要注意采用资源有序分配法避免死锁的产生。

## 守护线程与用户线程

Java 中线程分为两类，分别为 Daemon 线程（守护线程）和 User 线程（用户线程），在 JVM 启动时候会调用 main 函数，main 函数所在的线程是一个用户线程，这个是我们可以看到线程，其实 JVM 内部同时还启动了好多守护线程，比如垃圾回收线程（严格说属于 JVM 线程）。

那么守护线程和用户线程有什么区别那？区别之一是当最后一个非守护线程结束时候，JVM 会正常退出，而不管当前是否有守护线程；也就是说守护线程是否结束并不影响 JVM 的退出。言外之意是只要有一个用户线程还没结束正常情况下 JVM 就不会退出。

那么 Java 中如何创建一个守护线程呢？代码如下：

```
public static void main(String[] args) {  
  
    Thread daemonThread = new Thread(new Runnable() {  
        public void run() {  
  
        }  
    });  
  
    // 设置为守护线程  
    daemonThread.setDaemon(true);  
    daemonThread.start();  
  
}
```



可知只需要设置线程的 daemon 参数为 true 即可。

下面通过例子来加深用户线程与守护线程的区别的理解，首先看下面代码：

```
public static void main(String[] args) {  
  
    Thread thread = new Thread(new Runnable() {  
        public void run() {  
            for(;;){}  
        }  
    });  
  
    //启动子线  
    thread.start();  
  
    System.out.print("main thread is over");  
}
```

结果输出为：



```
testDaemonThread [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (2017年9月28日 下午10:05:43)  
main thread is over
```

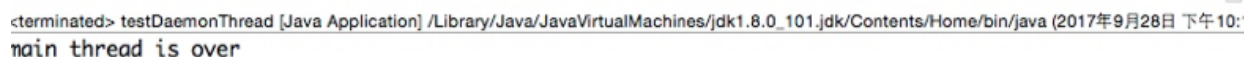
如上代码在 main 线程中创建了一个 thread 线程，thread 线程里面是无限循环，运行代码从结果看 main 线程已经运行结束了，那么 JVM 进程已经退出了？从 IDE 的输出结果右侧上的红色方块说明 JVM 进程并没有退出，另外 Mac 上执行 `ps -eaf | grep java` 会输出结果，也可以证明这个结论。

这个结果说明了当父线程结束后，子线程还是可以继续存在的，也就是子线程的生命周期并不受父线程的影响。也说明了当用户线程还存在的情况下 JVM 进程并不会终止。

那么我们把上面的 thread 线程设置为守护线程后在运行看看会有什么效果：

```
//设置为守护线程  
thread.setDaemon(true);  
//启动子线  
thread.start();
```

执行结果为：



```
<terminated> testDaemonThread [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (2017年9月28日 下午10:05:43)  
main thread is over
```

如上在启动线程前设置线程为守护线程，从输出结果可知 JVM 进程已经终止了，执行 `ps -eaf | grep java` 也看不到 JVM 进程了。这个例子里面 main 函数是唯一的用户线

程，thread 线程是守护线程，当 main 线程运行结束后，JVM 发现当前已经没有用户线程了，就会终止 JVM 进程。

Java 中在 main 线程运行结束后，JVM 会自动启动一个叫做 DestroyJavaVM 线程，该线程会等待所有用户线程结束后终止 JVM 进程，下面通过简单的 JVM 代码来证明这个结论：

翻开 JVM 的代码，最终会调用到 JavaMain 这个函数：

```
int JNICALL
JavaMain(void * _args)
{
    ...
    //执行Java中的main函数
    (*env)->CallStaticVoidMethod(env, mainClass, mainID,
    mainArgs);

    //main函数返回值
    ret = (*env)->ExceptionOccurred(env) == NULL ? 0 : 1;

    //等待所有非守护线程结束，然后销毁JVM进程
    LEAVE();
}
```

LEAVE 是 C 语言里面的一个宏定义，定义如下：

```
#define LEAVE() \
    do { \
        if ((*vm)->DetachCurrentThread(vm) != JNI_OK) { \
            JLI_ReportErrorMessage(JVM_ERROR2); \
            ret = 1; \
        } \
        if (JNI_TRUE) { \
            (*vm)->DestroyJavaVM(vm); \
            return ret; \
        } \
    } while (JNI_FALSE)
```

上面宏的作用实际是创建了一个名字叫做 DestroyJavaVM 的线程来等待所有用户线程结束。

在 Tomcat 的 NIO 实现 NioEndpoint 中会开启一组接受线程用来接受用户的链接请求和一组处理线程负责具体处理用户请求，那么这些线程是用户线程还是守护线程呢？下面我们看下 NioEndpoint 的 startInternal 方法：

```
public void startInternal() throws Exception {

    if (!running) {
```

```

        running = true;
        paused = false;

        ...

        //创建处理线程
        pollers = new Poller[getPollerThreadCount()];
        for (int i=0; i<pollers.length; i++) {
            pollers[i] = new Poller();
            Thread pollerThread = new Thread(pollers[i],
getName() + "-ClientPoller-" + i);
            pollerThread.setPriority(threadPriority);
            pollerThread.setDaemon(true); //声明为守护线程
            pollerThread.start();
        }
        //启动接受线程
        startAcceptorThreads();
    }

    protected final void startAcceptorThreads() {
        int count = getAcceptorThreadCount();
        acceptors = new Acceptor[count];

        for (int i = 0; i < count; i++) {
            acceptors[i] = createAcceptor();
            String threadName = getName() + "-Acceptor-" + i;
            acceptors[i].setThreadName(threadName);
            Thread t = new Thread(acceptors[i], threadName);
            t.setPriority(getAcceptorThreadPriority());
            t.setDaemon(getDaemon()); //设置是否为守护线程，默认为守护
线程
            t.start();
        }
    }

    private boolean daemon = true;
    public void setDaemon(boolean b) { daemon = b; }
    public boolean getDaemon() { return daemon; }

```

如上代码也就是说默认情况下接受线程和处理线程都是守护线程，这意味着当 Tomcat 收到 shutdown 命令后 Tomcat 进程会马上消亡，而不会等处理线程处理完当前的请求。

**注：**如果你想在主线程结束后 JVM 进程马上结束，那么创建线程的时候可以设置线程为守护线程，否则如果希望主线程结束后子线程继续工作，等子线程结束后在让 JVM 进程结束那么就设置子线程为用户线程。

## 总结

本 Chat 作为 Java 并发编程之美系列的开篇，讲解了多线程并发编程线程有关基础知识，有了这些基础，为后面研究并发编程的高级知识打下基础。