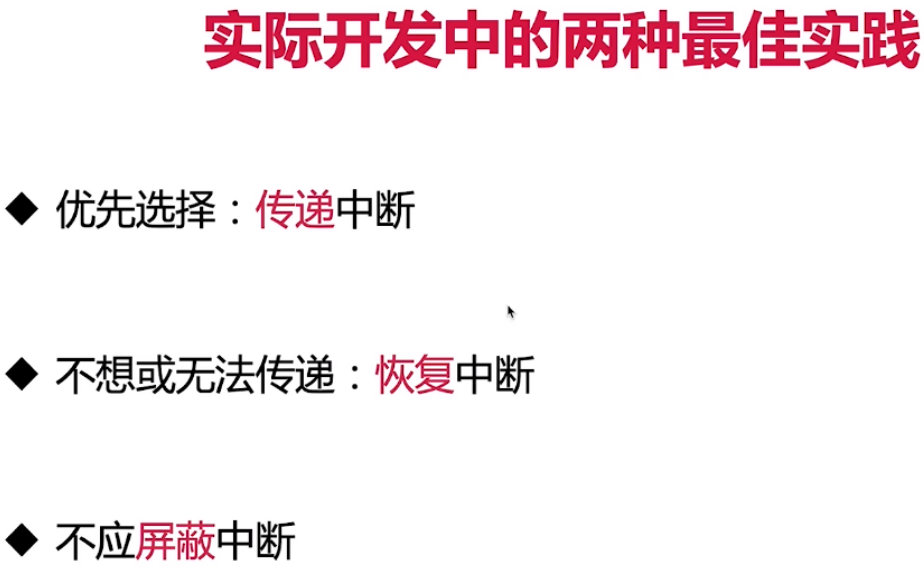
中断线程的两种最佳实践



1. run 方法中 调用的 子方法或者业务中 有中断出现的情况，在 子方法中出现的中断应该 抛出由 run 函数进行捕获处理。注：run方法中只能 try / catch 捕获异常，不能再 抛出异常

```
/**
 *  最佳实践：  在 catch 了 InterruptedException 之后的优先选择是 在方法签名中 抛出异常
 *  那么在run 方法中 会 强制 try / catch
 */
public class RightWayStopThreadInProd  implements Runnable{
    @Override
    public void run() {
        while (true){
            System.out.println("go");
            try {
                throwInMethod();  // 调用子方法。 子方法抛出异常后，进行异常捕获，run方法只能 使用try / catch捕获
            } catch (InterruptedException e) {
                e.printStackTrace();
                // 做出 异常处理的方法， 比如保存日志，停止程序
                System.out.println("保存日志");

            }
        }
    }

    private void throwInMethod() throws InterruptedException {
        /**
         *  考虑 出现的 异常是否需要 暴露给上一层的调用者
         */
//        try {
//            System.out.println(Thread.currentThread().getName());
//            Thread.sleep(2000);
//        } catch (InterruptedException e) {
```

```
//            e.printStackTrace();
//        }

        // 抛出异常
         Thread.sleep(2000);

    }

    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new RightWayStopThreadInProd());
        thread.start();

        Thread.sleep(1000);

        thread.interrupt(); // 中断
    }
}
```

2. 不想传递 或者 无法传递的情况下： 恢复中断，让上层接收到中断信号进行处理

```
/**
 *  最佳实践2： 在 catch 子语句中 调用 Thread。currentThread().interrupt() 来恢复设置中断状态
 *  以便于在后续的执行中，依然能够检查到刚才发生了中断，回到 RightWayStopThreadInProd 补上中断，让它跳出
 *
 */
public class RightWayStopThreadInProd2 implements Runnable{
    @Override
    public void run() {
        while (true){

            // 判断是否有中断异常的信号
            if(Thread.currentThread().isInterrupted()){
                System.out.println("接收到中断信号");
                break;
            }
            System.out.println("go");
            reInterrupt();  // 调用子方法，子方法中 没有 抛出异常，所以不能 try / catch

        }
    }

    private void reInterrupt(){
        // 抛出异常
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            // 捕获到 中断异常后 再进行 抛出中断， 以便于上层 检查是否出现了 中断信号
            Thread.currentThread().interrupt();  // 如果注释该语句，则上层无法接收到 中断信号
            e.printStackTrace();
        }

    }
```

```
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new RightWayStopThreadInProd2());
        thread.start();

        Thread.sleep(1000);

        thread.interrupt(); // 中断
    }
}
```

3. 响应中断的方法总结：

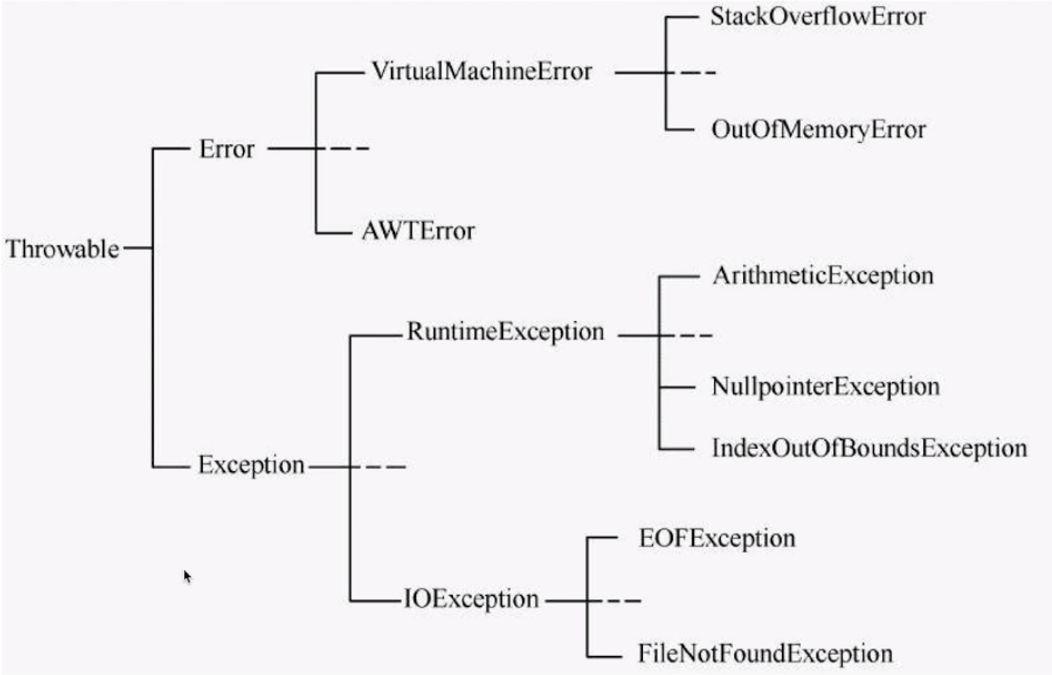
响应中断的方法总结列表

- Object. wait()/ wait(long)/ wait(long, int)
- Thread. sleep(long) /sleep(long, int)
- Thread. join()/ join(long)/ join(long, int)
- java. util. concurrent. BlockingQueue. take() /put(E)
- java. util. concurrent. locks. Lock. lockInterruptibly()

- java. util. concurrent. CountDownLatch. await()
- java. util. concurrent. CyclicBarrier. await()
- java. util. concurrent. Exchanger. exchange(V)
- java.nio.channels.InterruptibleChannel相关方法
- java.nio.channels.Selector的相关方法

彩蛋： Java 异常体系：

彩蛋：Java异常体系



5. 错误的停止方法

- 被弃用的stop, suspend和resume方法
- 用volatile设置boolean标记位

1. Stop

```
/**
 * 错误的停止方法：  用stop（）来停止线程，
 *       会导致线程运行到一半突然停止，没有办法完成一个基本单位的操作，
 *       造成脏数据
 *       （模拟军队领取军资）
 */

public class StopThread implements Runnable {

    @Override
    public void run() {
        // 模拟指挥军队，一共 5 个连队，每个连队 10 人，以连队为单位发送军资，叫到的士兵去发送武器
        for (int i = 0; i < 5; i++) {
            System.out.println("连队 " + i + "开始领取武器" );
            for (int j = 0; j < 10; j++) {
                System.out.println(j);
                try {
                    Thread.sleep(30);
                } catch (InterruptedException e) {
```

```
                    e.printStackTrace();
                }
            }
            System.out.println("连队 " + i + "已经领取完毕");
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new StopThread());
        thread.start();
        Thread.sleep(1000);

        // 停止线程,出现 单位 （连队） 被终止了，导致没有完成 基本单位的操作
        thread.stop();
    }
}
```

2. suspend 方法 调用后 不会 释放锁资源，会导致 出现 死锁情况
3. 使用 volatile 设置 Boolean 标志位

   代码1： 可以正确的运行

```
/**
 * 演示 volatile的局限性， part1 看似可行
 */
public class WrongWayVolatile implements Runnable {

    private volatile boolean canceled = false;

    @Override
    public void run() {
        int num = 0;
        try {
//              volatile 具有可见性，
            while (num <= 100000 && !canceled){
                if (num % 100 == 0){
                    System.out.println(num + "是 100 的倍数");
                }
                num ++;
                Thread.sleep(1);  // 注释掉该语句，程序便不能 被 volatile 设置的标志位暂停
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
    }

    public static void main(String[] args) throws InterruptedException {
        WrongWayVolatile wrongWayVolatile = new WrongWayVolatile();
        Thread thread = new Thread(wrongWayVolatile);
        thread.start();

        Thread.sleep(5000);

        // 使用 volatile 修饰的变量 控制线程的结束
        wrongWayVolatile.canceled = true;
    }
```

```
}
```

代码二：

```
/**
 * 演示 volatile的局限性，当当前线程陷入 阻塞时， volatile 是无法 结束线程
 *
 * 该例子中生产者的生产速度很快，消费者消费速度慢，所以会出现阻塞队列满，生产者会阻塞，等待消费者进一步消费
 *
 *
 */
public class WrongWayVolatileCantStop {

    // 使用 生产者 与 消费者
    public static void main(String[] args) throws InterruptedException {
        // 1. 共同的仓库
        ArrayBlockingQueue storage = new ArrayBlockingQueue(10);

        // 2. 创建生产者
        Producer producer = new Producer(storage);

        // 3. 创建生产者线程
        Thread producerThread = new Thread(producer);
        producerThread.start();
        // 暂停主线程，让队列满
        Thread.sleep(1000);

        // 4. 创建消费者
        Consumer consumer = new Consumer(storage);
        // 5. 生产者进行消费
        while (consumer.needMoreNums()){
            System.out.println(consumer.storage.take() + " 被消费了");
            Thread.sleep(100);
        }

        // 6. 消费完成后，需要停止生产者
        System.out.println("消费者不需要更多数据了。");
        // 消费者不需要更多数据的情况下，通过 设置的 boolean 通知生产者停止生产
        producer.canceled = true;
        System.out.println(producer.canceled);

    }


}

// 生产者
class Producer implements Runnable{
    BlockingQueue storage;
    public volatile boolean canceled = false;

    public Producer(BlockingQueue storage) {
        this.storage = storage;
    }

    @Override
```

```
    public void run() {
        int num = 0;
        try {
            // volatile 具有可见性,
            while (num <= 100000 && !canceled){
                if (num % 100 == 0){
                    System.out.println(num + "是 100 的倍数, 被放到了仓库中--");
                    storage.put(num);
                }
                num ++;
//                Thread.sleep(1); // 不能进行睡眠,,------ 如果没有被注释掉, 该程序会被正常执行完
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }finally {
            System.out.println("生产者停止运行! ");
        }
    }
}

// 消费者
class Consumer{
    BlockingQueue storage;

    public Consumer(BlockingQueue storage) {
        this.storage = storage;
    }

    // 判断是否需要更多
    public boolean needMoreNums(){
        if (Math.random() > 0.95){
            return false;
        }else {
            return true;
        }
    }
}
```

4. 代码3： 使用 interrupt解决不能解决中断

```
/**
 * 使用 interrupt 进行通知线程中断
 */
public class WrongWayVolatileFixed {

    // 使用 生产者 与 消费者
    public static void main(String[] args) throws InterruptedException {

        WrongWayVolatileFixed wrongWayVolatileFixed = new WrongWayVolatileFixed();

        // 1. 共同的仓库
        ArrayBlockingQueue storage = new ArrayBlockingQueue(10);

        // 2. 创建生产者
```

```
        Producer producer = wrongWayVolatileFixed.new Producer(storage);

        // 3. 创建生产者线程
        Thread producerThread = new Thread(producer);
        producerThread.start();
        // 暂停主线程，让队列满
        Thread.sleep(1000);

        // 4. 创建消费者
        Consumer consumer = wrongWayVolatileFixed.new Consumer(storage);
        // 5. 生产者进行消费
        while (consumer.needMoreNums()){
            System.out.println(consumer.storage.take() + " 被消费了");
            Thread.sleep(100);
        }

        // 6. 消费完成后，需要停止生产者
        System.out.println("消费者不需要更多数据了。");
        // 消费者不需要更多数据的情况下，通过 设置的 boolean 通知生产者停止生产

        producerThread.interrupt();  // 开启中断通知

    }


    // 生产者
    class Producer implements Runnable{
        BlockingQueue storage;

        public Producer(BlockingQueue storage) {
            this.storage = storage;
        }

        @Override
        public void run() {
            int num = 0;
            try {
                // volatile 具有可见性，
                while (num <= 100000 &&
!Thread.currentThread().isInterrupted()){
                    if (num % 100 == 0){
                        System.out.println(num + "是 100 的倍数，被放到了仓库中--");
                        storage.put(num);
                    }
                    num ++;
//                  Thread.sleep(1); // 注释 与 不注释 程序都可以正常执行
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }finally {
                System.out.println("生产者停止运行！");
            }
        }
    }

    // 消费者
    class Consumer{
```

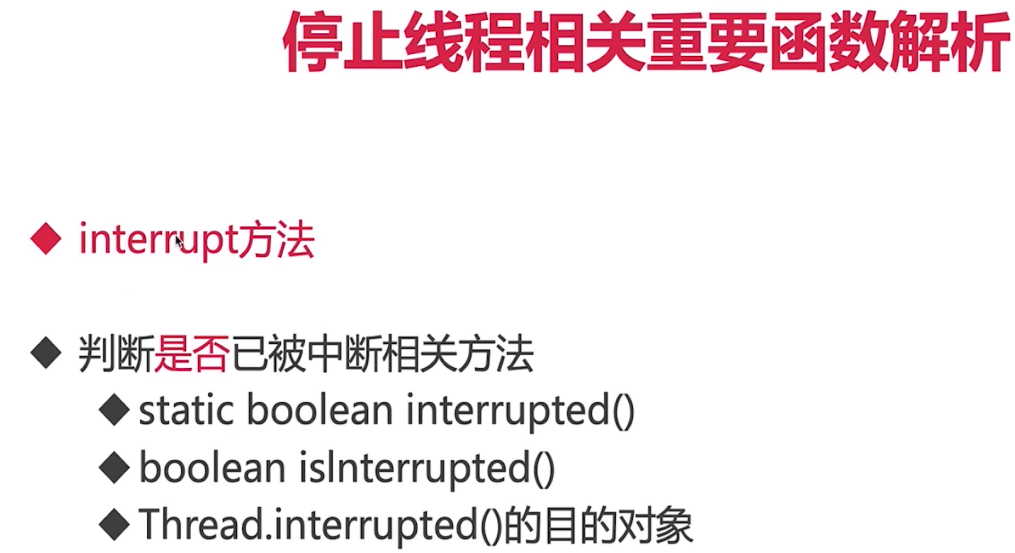
```
        BlockingQueue storage;

        public Consumer(BlockingQueue storage) {
            this.storage = storage;
        }

        // 判断是否需要更多
        public boolean needMoreNums(){
            if (Math.random() > 0.95){
                return false;
            }else {
                return true;
            }
        }
    }

}
```

解析 interrupt方法：




```
public static boolean interrupted() {
    return currentThread().isInterrupted(true);
}
```

该静态方法的执行对象是 当前线程对象，也就是说 使用其他的线程对象在 主线程中调用该方法，执行的对象是**主线程**， 而且该方法会清除 中断标志位

停止线程的常见面试问题：

1.

停止线程——面试常见问题

- 如何停止线程
  1. 原理：用interrupt来请求、好处
  2. 想停止线程，要请求方、被停止方、子方法被调用方相互配合
  3. 最后再说错误的方法：stop/suspend已废弃，volatile的boolean无法处理长时间阻塞的情况

2.

停止线程——面试常见问题

- 如何处理不可中断的阻塞