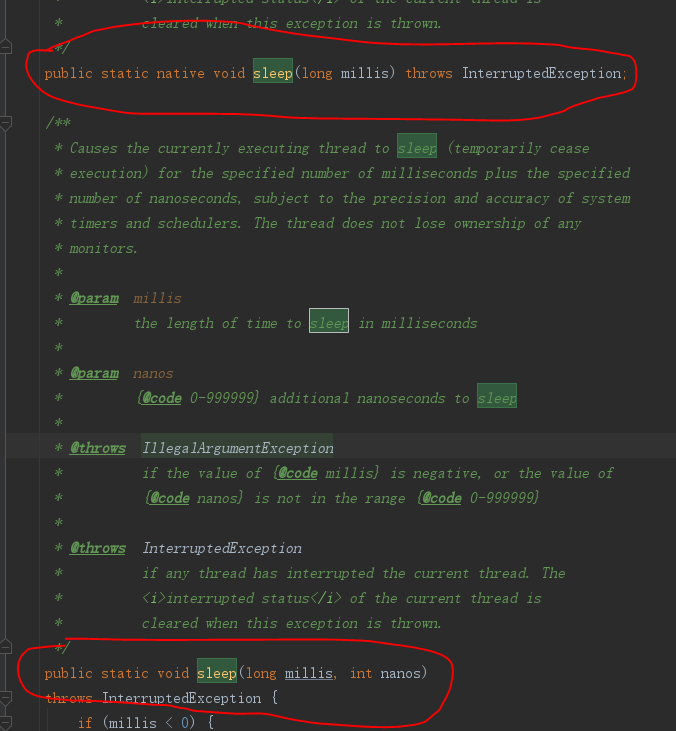
# Thread API的使用和介绍

1. **Sleep方法**



Sleep方法会使当前线程进入指定毫秒数的休眠，暂停执行，虽然给定了一个休眠的时间，但是最终要以系统的定时器和调度器的精度为准，休眠有一个非常重要的特性，那就是其不会放弃monitor锁的所有权

Demo：

package com.lagoon.test;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/1 11:11  
 \** ***@Description*** *简单的demo  
 \* currentTimeMillis();获取当前时间的方法  
 \*/*public class ThreadSleep{  
 public static void main(String[] args) {  
 /\*  
 开启的额外线程  
 \*/  
 new Thread(()->{  
  
 long startTime=System.currentTimeMillis();  
 sleep(2\_000L);  
 long endTime=System.currentTimeMillis();  
 System.out.println(String.format("额外的线程休眠了 %d ms",(endTime-startTime)));  
 }).start();  
  
 /\*  
 主线程  
 \*/  
 long starttime=System.*currentTimeMillis*();  
 *sleep*(3\_000L);  
 long endTime=System.*currentTimeMillis*();  
 System.*out*.println(String.*format*("主线程休眠了 %d ms",(endTime-starttime)));  
 }  
  
 private static void sleep(long ms){  
 try {  
 Thread.*sleep*(ms);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

建议使用TimeUnit替代Thread.sleep

在JDK1.5以后，JDK引入了一个枚举TimeUnit，其对sleep方法提供了很好的封装，使用它可以省去时间单位的换算步骤，比如想让线程休眠3小时24分17秒88毫秒，使用TimeUnit来实现就特别方便

Thread.*sleep*(12257088L);

等价于

TimeUnit.*HOURS*.sleep(3);  
TimeUnit.*MINUTES*.sleep(24);  
TimeUnit.*SECONDS*.sleep(17);  
TimeUnit.*MILLISECONDS*.sleep(88);

**2．yield方法**

yield方法属于一种启发式方法，其会提醒调度器我愿意放弃当前的额CPU资源，如果CPU资源不紧张，则会忽略这种提醒

调用yield方法会使当前线程从RUNNING状态切换到RUNNABLE状态一般这个方法不常用

yield和sleep方法的本质区别：

* Sleep会导致当前线程的暂停指定的时间，没有CPU时间时间片的消耗
* Yield只是对CPU调度器的一个提示，如果CPU调度器没有忽视这个提示，它会导致线程上下文的切换
* Sleep会使线程短暂的block，会在给定的时间内释放CPU资源
* Yield会使RUNNING状态的Thread进入RUNNABLE状态（如果CPU调度器没有忽视这个提示的话）
* Sleep几乎百分百的完成了给定时间的休眠，而yield的提示并不能一定担保
* 一个线程sleep另一个线程调用interrupt会捕捉到中断信号，而yield则不会

1. **设置线程的优先级方法**



线程的优先级介绍：

进程有进程的优先级，线程同样也有优先级，理论上是优先级更高的线程会获取优先被CPU调度的机会。

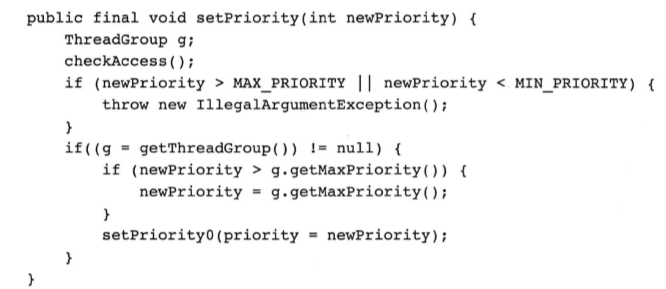
如果CPU比较忙，设置优先级可能会获得更多的CPU时间片，但是闲时优先级的高低几乎不会有任何作用

设置优先级线程demo：

package com.lagoon.test;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/1 12:16  
 \** ***@Description*** *设置线程优先级简单demo  
 \*/*public class ThreadPriority {  
  
 public static void main(String[] args) {  
 Thread t1=new Thread(()->{  
 while (true){  
 System.out.println("t1");  
 }  
 });  
 t1.setPriority(3);  
  
 Thread t2=new Thread(()->{  
 while (true){  
 System.out.println("t2");  
 }  
 });  
  
 t2.setPriority(10);  
 t1.start();  
 t2.start();  
 }  
}

运行结果发现t2的输出频率明显很高

setPriority方法源码

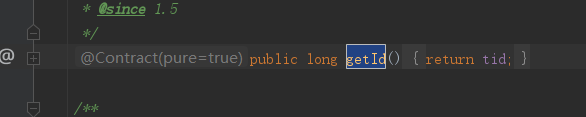


通过源码可以看出，线程的优先级不能小于1也不能大于10，如果指定的线程优先级大于线程所在group的优先级，那么指定的优先级将会失效，取而代之的是group的最大优先级

关于优先级的一些总结：

一般情况下，不会对线程的优先级进行设定，更不会让某些业务严重地依赖线程的优先级别，比如权重，借助优先级设定某个任务的权重，这种方式是可取的，一般定义线程的时候用线程默认的优先级就好了，线程默认的优先级和他的父类保持一致，一般情况下都是5，因为main线程的优先级就是5，所以它派生出来的线程都是5

1. **获取线程ID**

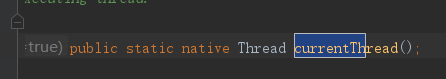


这个方法，获取线程唯一的ID，线程的ID在整个JVM进程中都会是唯一的，并且是从0开始逐次递增

一个JVM进程启动时，实际上是开辟了很多个线程，自增序列已经有了一定的消耗，因此我们自己创建的线程绝非第0号线程

1. **获取当前线程**

获取当前线程的方法：

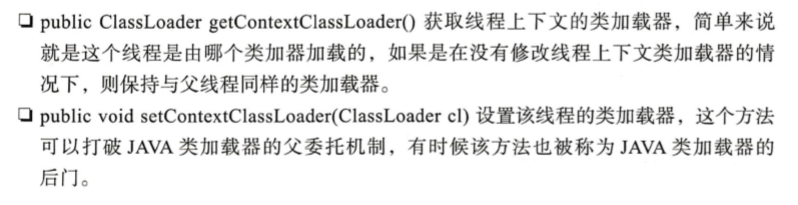


方法用于返回当前执行线程的引用，这个方法使用非常广泛

Demo：

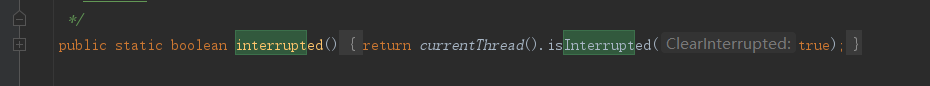
package com.lagoon.test;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/1 12:39  
 \** ***@Description*** *获取当前线程demo  
 \*/*public class CurrentThread {  
 public static void main(String[] args) {  
 Thread thread=new Thread(){  
 @Override  
 public void run() {  
 System.out.println(Thread.currentThread().getName());  
 }  
 };  
 thread.start();  
  
 String name=Thread.*currentThread*().getName();  
 System.*out*.println("主线程"+name);  
 }  
}

1. **设置线程上下文类加载器**



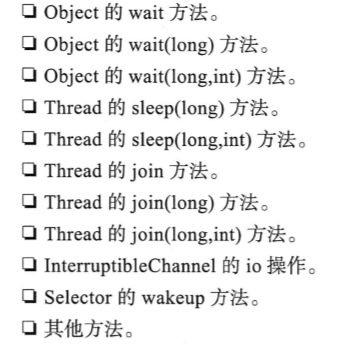
1. **线程interrupt**







1. 第一个方法interrupt



以上的方法的调用会使得当前线程进入阻塞状态，而调用当前线程的interrupt方法，就可以打断阻塞

若另外一个线程调用被阻塞线程的interrupt方法，则会打断这种阻塞因此这种方法有时会被称为可中断方法

但是打断一个线程并不等于该线程的生命周期结束，仅仅是打断了当前线程的阻塞状态

一旦线程在阻塞的情况下被打断，都会抛出一个InterruptedException的异常，这个异常就像一个signal信号一样通知当前线程被打断了

Demo：

package com.lagoon.test;  
  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/1 12:58  
 \** ***@Description*** *打断线程阻塞demo  
 \*/*public class ThreadInterrupt {  
 public static void main(String[] args) throws InterruptedException {  
 Thread thread=new Thread(()->{  
 try {  
 TimeUnit.MINUTES.sleep(1);  
 } catch (InterruptedException e) {  
 System.out.println("我正在休眠，但是被打断了...");  
 }  
 });  
 thread.start();  
  
 TimeUnit.*MILLISECONDS*.sleep(2);  
 thread.interrupt();  
  
 }  
}

上面的代码创建了一个线程，并且企图休眠一分钟，不过大约在2毫秒之后就被主线程调用interrupt方法打断，程序的执行结果就是“我正在休眠，但是被打断了…”

在一个线程内部存在着名为interrupt flag的标识，如果一个线程被interrupt，那么它的flag将被设置，但是如果当前线程正在执行一可中断方法被阻塞时，调用interrupt方法将其中断，反而会导致flag清除

另外，如果一个线程已经是死亡状态，那么尝试对其interrupt会直接被忽略

1. 第二个方法 isIntertupted

isIntertupted是Thread的一个成员方法，它主要判断当前线程是否被中断，该方法仅仅是对interrupt标识的一个判断，并不会影响标识发生任何改变

demo：

package com.lagoon.test;  
  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/1 13:19  
 \** ***@Description*** *\*/*public class ThreadisInterrupted {  
 public static void main(String[] args) throws InterruptedException {  
 Thread thread=new Thread(){  
 @Override  
 public void run() {  
 while (true){  
  
 }  
 }  
 };  
  
 thread.start();  
 System.*out*.println("线程被中断了吗？"+thread.isInterrupted());  
 /\*  
 被main线程调用而导致中断  
 \*/  
 thread.interrupt();  
 System.*out*.println("线程被中断了吗？"+thread.isInterrupted());  
 }  
}

sleep方法是可中断方法，会捕捉到中断信号，也就是捕获Interrupted异常，之后就会擦掉interrupt标识

package com.lagoon.test;  
  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/1 17:48  
 \** ***@Description*** *\*/*public class ThreadisInterrupted2 {  
  
 public static void main(String[] args) throws InterruptedException {  
 Thread thread=new Thread(){  
 @Override  
 public void run() {  
 while (true){  
 try {  
 TimeUnit.MINUTES.sleep(1);  
 } catch (InterruptedException e) {  
 System.out.printf("我被中断了吗？"+isInterrupted());  
 }  
 }  
 }  
 };  
  
 thread.setDaemon(true);  
 thread.start();  
 TimeUnit.*MILLISECONDS*.sleep(2);  
 System.*out*.println("线程被中断了吗？"+thread.isInterrupted());  
 thread.interrupt();  
 TimeUnit.*MILLISECONDS*.sleep(2);  
 System.*out*.println("线程被中断了吗？"+thread.isInterrupted());  
 }  
}

由于在run方法中使用了sleep这个可中断方法，它会捕捉到中断信号，并且擦除interrupt标识，因此程序的运行结果都会是false

不难理解，可中断方法捕获到了中断信号之后，为了不影响线程中其他方法的执行，将线程的interrupt标识复位是一种很合理的设计

第3种方法interupted

Interrupted是一个静态方法，虽然其也用于判断当前线程是否被中断，但是它和成员方法isInterrupted还是有很大的区别的，调用该方法会直接擦除掉线程的Interrupted标识，需要注意的是，如果当前线程被打断了，那么第一次调用interrupted方法会返回true，并且立即擦除了interrupted标识，第二次包括以后的调用都会返回false，除非在此期间线程有一次被打断

Demo：  
package com.lagoon.test;  
  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/1 19:13  
 \** ***@Description*** *interrupted方法demo  
 \*/*public class Threadinterrupted {  
 public static void main(String[] args) throws InterruptedException {  
 Thread thread=new Thread(){  
 @Override  
 public void run() {  
 while (true){  
 System.out.println(Thread.interrupted());  
 }  
 }  
 };  
 thread.setDaemon(true);  
 thread.start();  
  
 TimeUnit.*MILLISECONDS*.sleep(2);  
 thread.interrupt();  
 }  
}

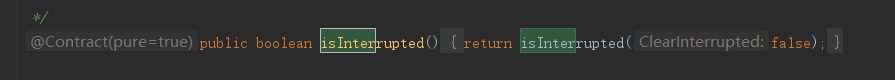
IsInterrupted和interrupted方法源码分析

两者都调用了同一个本地方法

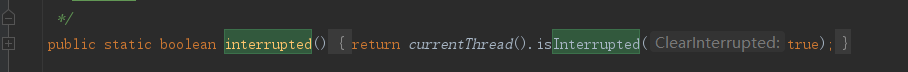


其中参数ClearInterrupted主要是用来控制是否擦除线程interrupt的标识

IsInterrupted方法的源码中该参数为false，表示不想擦除

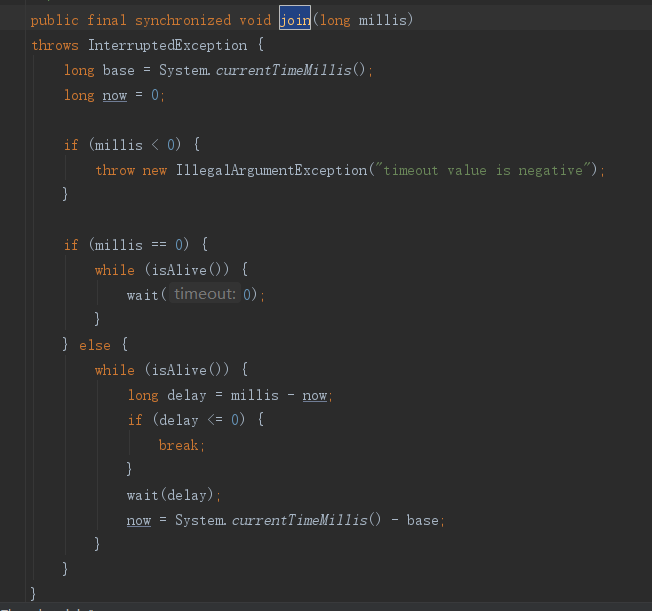


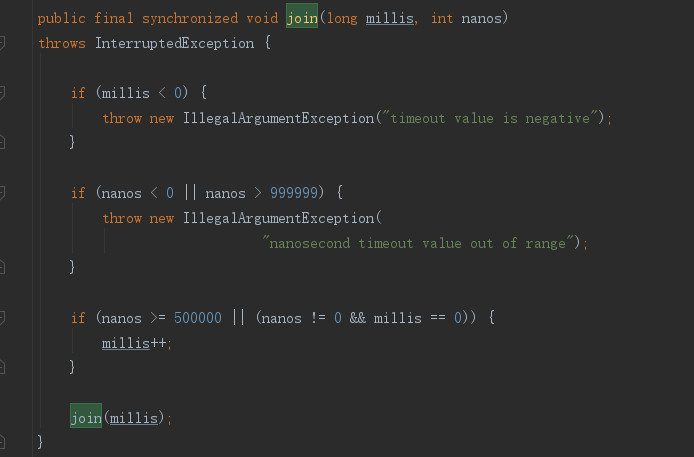
而interrupted静态方法中该参数为true，表示想要擦除

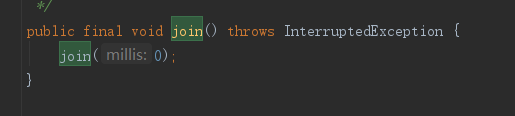


1. **线程join**

Join方法同样是一个非常重要的方法，使用它的特性可以实现很多强大的功能，与sleep一样它也是一个可中断方法，也就是说，如果其他线程执行了对当前线程的interrupt操作，它也会捕获到中断信号，并且擦除线程的interrupt标识，Thread的API为我们提供三个不同的join方法







Join线程方法详解

Join某个线程A，会使当前线程B进入等待，直到线程A结束生命周期，或者达到给定的时间，那么在此期间B线程是处于BLOCKED的，而不是线程A

Demo：  
package com.lagoon.test;  
  
import java.util.List;  
import java.util.concurrent.TimeUnit;  
import java.util.stream.IntStream;  
  
import static java.util.stream.Collectors.*toList*;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/1 19:32  
 \** ***@Description*** *\*/*public class ThreadJoin {  
 public static void main(String[] args) throws InterruptedException {  
 //定义两个线程，并保存到threads中  
 List<Thread> threads= IntStream.*range*(1,3).mapToObj(ThreadJoin::*create*).collect(*toList*());  
  
 //启动这两个线程  
 threads.forEach(Thread::start);  
  
 //执行这两个线程的join方法  
 for (Thread thread:threads){  
 thread.join();  
 }  
  
 //main线程循环输出  
 for (int i=0;i<10;i++){  
 System.*out*.println(Thread.*currentThread*().getName()+"#"+i);  
 *shortSleep*();  
 }  
 }  
  
 //构造一个简单的线程，每个线程只是简单的循环输出  
 private static Thread create(int seq){  
 return new Thread(()->{  
 for (int i=0;i<10;i++){  
 System.out.println(Thread.currentThread().getName()+"#"+i);  
 shortSleep();  
 }  
 },String.valueOf(seq));  
 }  
  
 private static void shortSleep(){  
 try {  
 TimeUnit.*SECONDS*.sleep(1);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

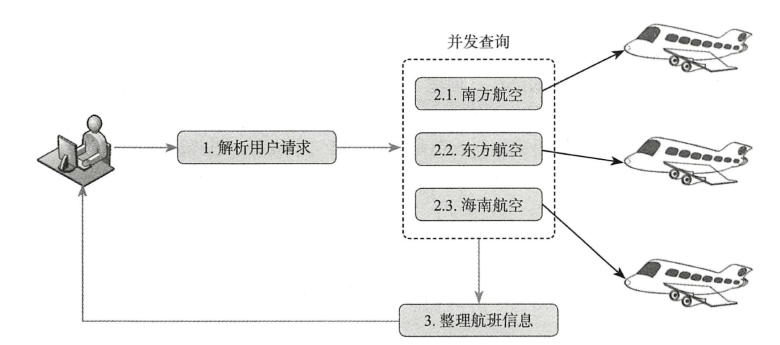
运行上面的程序，会发现线程一和线程二交替的输出直到结束它们的生命周期，main线程才会开始运行

但是如果将join方法那段全部注释掉再运行的话，会发现三个线程交替地输出

**Join方法实战**

**应用场景：**

假设你有一个APP，主要用于查询航班信息，你的APP是没有实时数据的，当用户发起查询请求时，需要到各大航空公司的接口获取信息，最后统一整理加工返回到APP客户端，如图所示



该例子是典型的串行任务局部并行化处理，用户在APP客户端输入出发地“北京”和目的地“上海”，服务器在接收到这个请求之后，先来验证用户的信息，然后到各大航空公司的接口查询信息，最后经过整理加工返回给客户端，每一个航空公司的接口不会都一样，获取的数据格式也不一样，查询的速度也存在着差异，如果再跟航空公司进行串行化交互（逐个地查询），很明显客户端需要等待很长的时间，这样的话，用户体验就会非常的差。如果将每一个航空公司的查询都交给一个线程去工作，然后在它们结束工作之后统一对数据进行整理，这样就可以极大的节约时间，从而提高用户体验效果

查询接口

package com.lagoon.concurrent;  
  
import java.util.List;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/1 19:57  
 \** ***@Description*** *查询接口  
 \*/*public interface FightQuery {  
 List<String> get();  
}

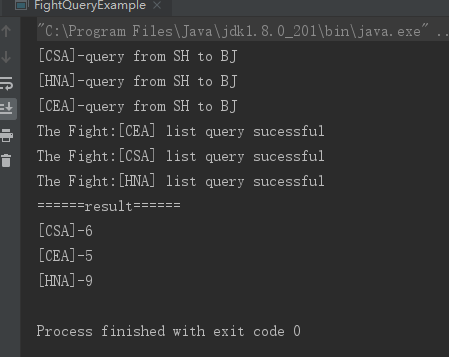
查询Fight的task：其实就是一个线程的子类，主要用于到各大航空公司获取数据

package com.lagoon.concurrent;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.concurrent.ThreadLocalRandom;  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/1 20:00  
 \** ***@Description*** *查询Fight的task：其实就是一个线程的子类，主要用于到各大航空公司获取数据  
 \*/*public class FightQueryTask extends Thread implements FightQuery{  
  
  
 private final String origin;  
 private final String destination;  
  
 private final List<String> fightList=new ArrayList<>();  
  
 public FightQueryTask(String airline,String origin, String destination) {  
 super("["+airline+"]");  
 this.origin = origin;  
 this.destination = destination;  
 }  
  
 @Override  
 public void run() {  
 System.*out*.printf("%s-query from %s to %s \n",getName(),origin,destination);  
 int randomVal= ThreadLocalRandom.*current*().nextInt(10);  
 try {  
 TimeUnit.*SECONDS*.sleep(randomVal);  
 this.fightList.add(getName()+"-"+randomVal);  
 System.*out*.printf("The Fight:%s list query sucessful\n",getName());  
 } catch (InterruptedException e) {  
   
 }  
 }  
  
 @Override  
 public List<String> get() {  
 return this.fightList;  
 }  
}

航班查询实现：

package com.lagoon.concurrent;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/1 20:09  
 \** ***@Description*** *航班查询实现  
 \*/*public class FightQueryExample {  
 //合作的各大航空公司  
 private static List<String> *fightCompany* = Arrays.*asList*("CSA","CEA","HNA");  
  
 public static void main(String[] args) {  
 List<String> results=*search*("SH","BJ");  
 System.*out*.println("======result======");  
 results.forEach(System.*out*::println);  
 }  
  
 private static List<String> search(String orginal,String dest){  
 final List<String> result=new ArrayList<>();  
 //创建查询航班信息的线程列表  
 List<FightQueryTask> tasks=*fightCompany*.stream().map(f->*createSearchTask*(f,orginal,dest)).collect(Collectors.*toList*());  
  
 //分别启动这几个线程  
 tasks.forEach(Thread::start);  
  
 //分别调用每一个线程的join方法，阻塞当前线程  
 tasks.forEach(t->{  
 try {  
 t.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
  
 //在此之前，当前线程会被阻塞住，获取每一个查询线程的结果，并且加入到result中  
 tasks.stream().map(FightQuery::get).forEach(result::addAll);  
 return result;  
 }  
  
 private static FightQueryTask createSearchTask(String fight,String orignal,String dest){  
 return new FightQueryTask(fight,orignal,dest);  
 }  
}

运行结果



1. **关闭线程**

* 线程结束生命周期正常关闭，线程运行结束以后，完成了自己的使命，就会正常退出，如果线程中的任务耗时比较短，或者时间可控，那么放任其结束即可
* 捕获中断信号关闭线程

Demo：

package com.lagoon.test;  
  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/2 13:00  
 \** ***@Description*** *捕获中断信号来关闭线程demo  
 \*/*public class InterruptThreadExit {  
  
 public static void main(String[] args) throws InterruptedException {  
 Thread t=new Thread(){  
 @Override  
 public void run() {  
 System.out.println("我即将启动...");  
 while (!isInterrupted()){  
  
 }  
 System.out.println("我即将退出...");  
 }  
 };  
  
 t.start();  
 TimeUnit.*MINUTES*.sleep(1);  
 System.*out*.println("程序即将关闭...");  
 t.interrupt();  
 }  
}

上面的demo是通过检查线程的interrupt的标识来决定是否退出的，如果在线程中执行某个可中断的方法，则可以通过捕获中断信号来决定是否退出

* 使用volatile开关控制

由于线程的interrupt标识很有可能被擦除，或者逻辑单元中不会调用任何可中断方法，所以使用volatile修饰的开关flag关闭线程也是一种常用的方法

Demo：

package com.lagoon.test;  
  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \** ***@Author*** *WinkiLee  
 \** ***@Date*** *2019/5/2 13:08  
 \** ***@Description*** *使用volatile开关控制来关闭线程  
 \*/*public class FlagThreadExit {  
 static class MyTask extends Thread{  
 private volatile boolean closed=false;  
  
 @Override  
 public void run() {  
 System.*out*.println("我即将启动...");  
 while (!closed && !isInterrupted()){  
  
 }  
 System.*out*.println("我即将退出...");  
 }  
  
 public void close(){  
 this.closed=true;  
 this.interrupt();  
 }  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
 MyTask t=new MyTask();  
 t.start();  
 TimeUnit.*SECONDS*.sleep(20);  
 System.*out*.println("程序即将被关闭");  
 t.close();  
 }  
}

上面的demo定义了一个closed开关变量，并且是使用volatile关键字修饰。

* 异常退出

在一个线程的执行单元中，是不允许抛出checked异常的，不论Thread中的run方法，还是Runnable中的run方法，如果线程在运行过程中需要捕获checked异常并且判断是否还有运行下去的必要，那么此时可以将checked异常封装成unchecked异常抛出进而结束线程的生命周期

* 进程假死