发件人：方堃 fangkun119@hotmail.com
主题：
日期：2021年3月4日 下午1:55
收件人：

CH21 多线程(A)：线程创建、封装托管、Join、异常捕获

2016年8月1日 星期一
09:42

全章概要

- 只是一个介绍，本章做不到融会贯通
- Web类库、Servlet等也是基于多线程，理解多线程有助于理解原理
- 速度、设计的可管理性、阻塞、事件驱动编程、进程、线程、抢占式地线程机制（非主动放弃）
- 线程行为，在不同版本的JDK之间，也存在差异，例如

事实上，你可以看到，在某个版本的JDK与下个版本之间，这个简单程序的输出会产生巨大的差异。例如，较早的JDK不会频繁对时间切片，因此线程1可能会首先循环到尽头，然后线程2会经历其所有循环，等等。这实际上与调用一个例程去同时执行所有的循环一样，只是启动所有线程的代价要更加高昂。较晚的JDK看起来会产生更好的时间切片行为，因此每个线程看起来都会获得更加正规的服务。通常，Sun并为提及这些种类的JDK的行为变化，因此你不能依赖于任何线程行为的一致性。最好的方式是在编写使用线程的代码时，尽可能地保守。

本节(A)概要

创建线程的方法

[一、通过实现Runnable接口来创建线程](#) (Web 视图)

1. 实现Runnable接口，交给Thread类

step1	//实现Runnable接口,在run()方法中编写线程的运行逻辑 class MyRunnable implement Runnable {...}
step2	//Runnable交给新创建的Thread Thread myThread = new Thread(new MyRunnable());
step3	//设置myThread的属性（如果需要）并启动线程 myThread.start();e
step4	//不再接受新的线程提交，已经启动的线程还是会一直运行直到完成 executorService .shutdown();

2. 实现Runnable接口，交给Executor托管

step1	//实现Runnable接口（同上）,也可再封装一层ThreadFactory class MyRunnable implement Runnable {...} class MyThreadFactory implement ThreadFactory {...}

step2	//创建ExecutorService，可指定线程池类型，可使用ThreadFactory
	//如果传入ThreadFactory则有机会在newTrhead(Runnable)方法中设置线程属性
	//代码参考线程工厂
	ExecutorService myExeSvc = Executors.newXXXThreadPool();
	ExecutorService myExeSvc = Executors.newXXXThreadPool(myThreadFactory);

step3	//多次调用execute()创建多个线程，使用的execute函数版本与创建ExecutorService时保持一致
	myExeSvc.execute(myRunnable);
	myExeSvc.execute(myThreadFactory);

step4	//不再接受新的线程提交，已经启动的线程还是会一直运行直到完成 executorService .shutdown();

3. 实现Callable接口，交给Executor托管，以获得线程退出时的返回值

step1	//实现Callable<MyReturnType>接口，在call()方法中实现线程逻辑 class MyCallable implement Callable<MyReturnType>;
step2	//创建ExecutorService(也可以在创建时指定Factory？) ExecutorService myExeSvc = Executors.newXXXThreadPool();
step3	//创建线程：多次调用submit()创建多个线程(也可 Future<MyReturnType> future = myExeSvc.submit(new MyCallable());
step4	//通过Future<MyReturnType>.get()来获取线程运行结果，get()函数会被block直到线程执行完毕 MyReturnType ret = future.get();
step5	//不再接受新的线程提交，已经启动的线程还是会一直运行直到完成 executorService .shutdown();

[二、简单情况下创建线程](#)

用内部类直接继承Thread类/自管理Runnable，整个过程封装在单独的函数中

其他知识

1. Thread变量到线程run()/call()结束后才容许会被回收
2. 多种ExecutorService

Executors.newCachedThreadPool()
Executors.newFixedThreadPool(threadNum)
Executors.newSingleThreadPool()
...

3. 休眠

使用TimeUnit的函数例如
TimeUnit.MILLISECONDS.sleep(100)
异常要在run/call中捕捉，跨线程(例如main())无法捕捉

4. getPriority()/setPriority()/yield()

暗示JVM对线程调度设置优先级、或将计算资源让给其他线程

5. 可以将线程设置成后台线程，这样就不用理会后台线程的终结

但是后台线程有自己的特点，参考[后台线程要点](#)

6. 线程管理：

Join一个线程：可以加超时参数、可以被中断
isAlive：检查线程是否还活着（run函数运行中）

另外四个章节概要

[三、Join一个线程、以及线程中断](#)
[四、线程用途演示：](#)
只是一个非常简单说明线程有什么好处的例子
[五、线程组：已废弃不推荐使用、推荐使用Executor替代、如果出于读老代码需要要学习、参考《Java编程思想》第二版](#)
[六、捕获异常：线程内部捕获 & 用Executor捕获线程内部泄露的异常](#)
为每个Executor设置
为全局的Thread设置

一、通过实现Runnable接口来创建线程

Runable

功能	实现Runnable接口，可以在run()函数中封装线程运行时所执行的操作	
使用	需要把Runnable交给Thread对象（main()函数直接调用Runnable接口的run()函数，仅能让任务在主线程中运行） 静态方法Thread.yield()暗示线程调度器，一段任务执行完了，此刻适合切换到其他线程	

例子	线程任务	public class LifeOff implements Runnbale {
		...
		public void run() {
		...
		Thread.yield();
		}
		}

```
public class LiftOff implements Runnable {
  protected int countDown = 10; // Default
  private static int taskCount = 0;
  private final int id = taskCount++;
  public LiftOff() {}
  public LiftOff(int countDown) {
    this.countDown = countDown;
  }
  public String status() {
    return "#" + id + "(" +
      (countDown > 0 ? countDown : "Liftoff!") + "), ";
  }
  public void run() {
    while(countDown-- > 0) {
      System.out.print(status());
      Thread.yield();
    }
```

		} } ///:~	

	main	public class MainThread {
		public static void main(String[] args) {
		LiftOff launch = new LiftOff();
		launch.run();
		}
		} /* Output:
		#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2),
			//main函数直接调用run()函数只能让任务在主线程中运行

	thread	public class BasicThreads {
		public static void main(String[] args) {
		Thread t = new Thread(new LiftOff());
		t.start();
		System.out.println("Waiting for LiftOff");
		}
		} /* Output: (90% match)
		Waiting for LiftOff
		#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2),
		#0(1), #0(Liftoff!),
		*///:~
			Thread thread = new Thread(new MyRunnable());
			//Thread#start()启动线程，在线程中运行MyRunnable#run()
			thread.start();

		public class MoreBasicThreads {
		public static void main(String[] args) {
		for(int i = 0; i < 5; i++)
		new Thread(new LiftOff()).start();
		System.out.println("Waiting for LiftOff");
		}
		} /* Output: (Sample)
		Waiting for LiftOff
		#0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8),
		#3(8), #4(8), #0(7), #1(7), #2(7), #3(7), #4(7), #0(6),
		#1(6), #2(6), #3(6), #4(6), #0(5), #1(5), #2(5), #3(5),
		#4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3),
		#2(3), #3(3), #4(3), #0(2), #1(2), #2(2), #3(2), #4(2),
		#0(1), #1(1), #2(1), #3(1), #4(1), #0(Liftoff!),
		#1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
		*///:~
			//更多地Thread对象，创建更多地线程
			for (int i = 0; i < 5; ++i) {
			new Thread(new MyRunnable()).start();
			}

Thread	功能	在线程中运行Runnable的run()函数（Runnable通过Thread的构造函数传入），例如见上面	
	备注	Thread类对象的回收时间	虽然Thread对象在执行start()之后，就没有引用指向它了 但是Thread其实已经将自己注册了，该对象要等到run()函数执行完毕并死亡之后，才容许被回收

Executor	功能	用线程池对Thread进行管理封装，不需专门写代码来处理Thread声明周期	
	方法	step1	//选择需要的线程池，例如CachedThreadPool, FixedThreadPool, SingleThreadPool ExecutorService executorService = Executor.newXXXThreadPool();
		step2	//线程池中启动一个线程，线程的内容由实现了Runnable接口的类来决定 executorService .execute(new MyRunnable());
		step3	//不再接受新的线程提交，已经启动的线程还是会一直运行直到完成 executorService .shutdown();
	线程池	Executors.newCachedThreadPool();	需要多少线程创建多少，shutdown()时停止创建新线程 是首选方式，发现问题时可考虑改用FixedThreadPool
		Executors.newFixedThreadPool(5);	线程池中的Thread数量是固定的， 在启动线程之前，预先执行代价高昂的线程分配任务

		Executors.newSingleThreadPool();	相当于Executors.newFixedTrheadPool(1)，仅启动一个线程
			如果向SingleThreadPool提交多个execute操作，这些操作会变成串行的
			//: concurrency/SingleThreadExecutor.java
			import java.util.concurrent.*;
			
			public class SingleThreadExecutor {
			public static void main(String[] args) {
			ExecutorService exec =
			Executors.newSingleThreadExecutor();
			for(int i = 0; i < 5; i++)
			exec.execute(new LiftOff());
			exec.shutdown();
			}
			} /* Output:
			#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2),
			#0(1), #0(Liftoff!), #1(9), #1(8), #1(7), #1(6), #1(5),
			#1(4), #1(3), #1(2), #1(1), #1(Liftoff!), #2(9), #2(8),
			#2(7), #2(6), #2(5), #2(4), #2(3), #2(2), #2(1),
			#2(Liftoff!), #3(9), #3(8), #3(7), #3(6), #3(5), #3(4),
			#3(3), #3(2), #3(1), #3(Liftoff!), #4(9), #4(8), #4(7),
			#4(6), #4(5), #4(4), #4(3), #4(2), #4(1), #4(Liftoff!),
			*///:~
			左侧程序输出体现出向SingleThreadPool提交的多个线程是串行运行的

	例子	//: concurrency/CachedThreadPool.java
		import java.util.concurrent.*;
		
		public class CachedThreadPool {
		public static void main(String[] args) {
		ExecutorService exec = Executors.newCachedThreadPool();
		for(int i = 0; i < 5; i++)
		exec.execute(new LiftOff());
		exec.shutdown();
		}
		} /* Output: (Sample)
		#0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8),
		#2(8), #3(8), #4(8), #0(6), #1(7), #2(7), #3(7), #4(7),
		#0(5), #1(6), #2(6), #3(6), #4(6), #0(4), #1(5), #2(5),
		#3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2),
		#1(3), #2(3), #3(3), #4(3), #0(1), #1(2), #2(2), #3(2),
		#4(2), #0(Liftoff!), #1(1), #2(1), #3(1), #4(1),
		#1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
		*///:~
			ExecutorService execSvc = Executors.newCachedThreadPool();
			for (int i = 0; i < 5; ++i) {
			execSvc.execute(new MyRunnable());
			execSvc.shutdown()

		//: concurrency/FixedThreadPool.java
		import java.util.concurrent.*;
		
		public class FixedThreadPool {
		public static void main(String[] args) {
		// Constructor argument is number of threads:
		ExecutorService exec = Executors.newFixedThreadPool(5);
		for(int i = 0; i < 5; i++)
		exec.execute(new LiftOff());
		exec.shutdown();
		}
		} /* Output: (Sample)
		#0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8),
		#2(8), #3(8), #4(8), #0(6), #1(7), #2(7), #3(7), #4(7),
			ExecutorService execSvc = Executors.newFixedThreadPool(5);
			for (int i = 0; i < 5; ++i) {
			execSvc.execute(new MyRunnable());
			}
			execSvc.shutdown();

```
#3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2),
#1(3), #2(3), #3(3), #4(3), #0(1), #1(2), #2(2), #3(2),
#4(2), #0(Liftoff!), #1(1), #2(1), #3(1), #4(1),
#1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
*///:~
```

Callable
ExecutorService.submit()

功能	获得线程运行结束时的返回值

方法

step1	//实现Callable<ReturnType>接口，需要为方法ReturnType call()提供实现
	class MyCallable implements Callable<MyReturnType> {
	MyReturnType call() {...};
	}

step2	//创建线程池,也可以用newCachedThreadPool, newSingleThreadPool ... ExecutorService executorService = Executors.newFixedThreadPool(5);
step3	//启动线程 Future<MyReturnType> future = executorService.submit(new MyCallable());
step4	//获得线程运行结果，调用会被阻塞直到线程运行完毕，函数会抛异常 MyReturnType ret = future.get();

step5	//不再接受新的线程提交
	executorService.shutdown()

例子

```
//: concurrency/CallableDemo.java
import java.util.concurrent.*;
import java.util.*;

class TaskWithResult implements Callable<String> {
  private int id;
  public TaskWithResult(int id) {
    this.id = id;
  }
  public String call() {
    return "result of TaskWithResult " + id;
  }
}
```

```
class MyCallable implements Callable<String> {
 ...
 public String call() {
  return "result of TaskWithResult " + id;
 }
}
```

```
public class CallableDemo {
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    ArrayList<Future<String>> results =
      new ArrayList<Future<String>>();
    for(int i = 0; i < 10; i++)
      results.add(exec.submit(new TaskWithResult(i)));
    for(Future<String> fs : results)
      try {
        // get() blocks until completion:
        System.out.println(fs.get());
      } catch(InterruptedException e) {
        System.out.println(e);
        return;
      } catch(ExecutionException e) {
        System.out.println(e);
      } finally {
        exec.shutdown();
      }
  }
} /* Output:
```

```
//ExecutorService
ExecutorService execSvc = Executors.newCachedThreadPool();
//启动线程
List<Future<String>> futureList = new ArrayList<Future<String>>();
for (int i = 0; i < 10; ++i) {
 futureList.add(execSvc.submit(new MyCallable(i)));
}
//取结果
for (Future<String> future : futureList)
 try {
  //阻塞直到完成
  System.out.println(future.get());
 } catch (InterruptedException e) {
  //因为调用了阻塞函数，所以才需要捕捉InterruptedException
  //捕捉的是主线程的中断，Callable线程的中断要在call()函数中捕捉
  ...
 } catch (ExecutionExeception e) {
  ...
 } finally {
  execSvc.shutdown();
 }
}
```

休眠及中断异常

方法

休眠	老版本Java	Thread.sleep()
	Java SE5/6+	TimeUnit.MILLISECONDS.sleep(100) //可以指定时间单位
捕捉中断异常	需要在run()/call()中捕捉InterruptedException 跨线程（例如main()）无法捕捉	

例子

```
//: concurrency/SleepingTask.java
// Calling sleep() to pause for a while.
import java.util.concurrent.*;

public class SleepingTask extends LiftOff {
  public void run() {
    try {
      while(countDown-- > 0) {
        System.out.print(status());
        // Old-style:
        // Thread.sleep(100);
        // Java SE5/6-style:
        TimeUnit.MILLISECONDS.sleep(100);
      }
    } catch(InterruptedException e) {
      System.err.println("Interrupted");
    }
  }
```

```
//LiftOff是作者在前面例子中写的类、实现了Runnable接口
public class SleepingRunnable implement Runnable {
 public void run() {
  try {
   while(countDown-- > 0) {
    ...
    TimeUnit.MILLISESCONDS.sleep(100); //休眠
   }
  } catch (InterruptedException e) { //中断在Runnable捕捉
   ...
  }
 }
}
```

```
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    for(int i = 0; i < 5; i++)
      exec.execute(new SleepingTask());
    exec.shutdown();
  }
} /* Output:
#0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8),
#3(8), #4(8), #0(7), #1(7), #2(7), #3(7), #4(7), #0(6),
#1(6), #2(6), #3(6), #4(6), #0(5), #1(5), #2(5), #3(5),
#4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3),
#2(3), #3(3), #4(3), #0(2), #1(2), #2(2), #3(2), #4(2),
#0(1), #1(1), #2(1), #3(1), #4(1), #0(Liftoff!),
#1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
*///:~
```

```
ExecutorService exec = Executors.newCachedThreadPool();
for (int i = 0; i < 5; ++i)
 exec.execute(new SleepingRunnable());
exec.shutdown();
```

优先级

方法

函数	getPriority() / setPriority()
效果	不会导致线程死锁，仅仅是被执行的频率降低
取值	尽管JDK有10个优先级，但它与多数操作系统都不能映射得很好。比如，Windows有7个优

		先级且不是固定的，所以这种映射关系也是不确定的。Sun的Solaris有2[31]个优先级。唯一可移植的方法是当调整优先级的时候，只使用MAX_PRIORITY、NORM_PRIORITY和MIN_PRIORITY三种级别。	
	当前线程	Thread.currentThread() 获得当前线程的引用；toString()之后是当前线程的名字	

	例子	//: concurrency/SimplePriorities.java
		// Shows the use of thread priorities.
		import java.util.concurrent.*;
		
		public class SimplePriorities implements Runnable {
		  private int countDown = 5;
		  private volatile double d; // No optimization
		  private int priority;
		  public SimplePriorities(int priority) {
		  this.priority = priority;
		  }
		  public String toString() {
		  return Thread.currentThread() + ": " + countDown;
		  }
		  public void run() {
		  Thread.currentThread().setPriority(priority);
		  while(true) {
		  // An expensive, interruptable operation:
		  for(int i = 1; i < 100000; i++) {
		  d += (Math.PI + Math.E) / (double)i;
		  if(i % 1000 == 0)
		  Thread.yield();
		  }
		  System.out.println(this);
		  if(--countDown == 0) return;
		  }
		}
			public String toString() {
			//打印当前线程名字
			return Thread.currentThread().toString();
			}
			
			public void run() {
			//为当前线程设置优先级
			Thread.currentThread().setPriority(prority);
			...
			//暗示编译器把cpu让给其他线程
			Thread.yield();
			...
			}

		public static void main(String[] args) {
		ExecutorService exec = Executors.newCachedThreadPool();
		for(int i = 0; i < 5; i++)
		exec.execute(
		new SimplePriorities(Thread.MIN_PRIORITY));
		exec.execute(
		new SimplePriorities(Thread.MAX_PRIORITY));
		exec.shutdown();
		}
		} /* Output: (70% match)
		Thread[pool-1-thread-6,10,main]: 5
		Thread[pool-1-thread-6,10,main]: 4
		Thread[pool-1-thread-6,10,main]: 3
		Thread[pool-1-thread-6,10,main]: 2
		Thread[pool-1-thread-6,10,main]: 1
		Thread[pool-1-thread-3,1,main]: 5
		Thread[pool-1-thread-2,1,main]: 5
		Thread[pool-1-thread-1,1,main]: 5
		Thread[pool-1-thread-5,1,main]: 5
		Thread[pool-1-thread-4,1,main]: 5
			ExecutorService execSvc = Executors.newCachedThreadPool();
			execSvc.execute(new MyRunnable());
			execSvc.shutdown();

yield()	如果知道已经完成了在**run()**方法的循环的一次迭代过程中所需的工作，就可以给线程调度机制一个暗示：你的工作已经做得差不多了，可以让别的线程使用CPU了。这个暗示将通过调用**yield()**方法来作出（不过这只是一个暗示，没有任何机制保证它将会被采纳）。当调用**yield()**时，你也是在建议具有相同优先级的其他线程可以运行。 **LiftOff.java**使用**yield()**在各种不同的**LiftOff**任务之间产生分布良好的处理机制。尝试着注释掉**LiftOff.run()**中的**Thread.yield()**，以查看区别。但是，大体上，对于任何重要的控制或在调整应用时，都不能依赖于**yield()**。实际上，**yield()**经常被误用。		
后台线程	内容	后台线程(demon) 功能：提供后台通用服务，辅助前台线程 生命周期：有前台程线程运行时，后台线程也运行 前台线程都结束时，程序就终止，同时也kill所有后台线程	
		创建方法 通过Thread：thread.setDeamon(true); thread.start(); 通过ExecutorService：用Executors.nexXXXThreadPool(...)时，向ThreadPool传入一个线程工厂 线程工厂可以将正在创建的线程设置为后台	
		行为特点 是否是后台：thread.isDeamon()可以知道线程是不是后台线程 后台属性继承：后台线程新创建出的线程都是后台的 不能优雅退出：前台线程终止时，后台线程立刻终止，即使有finally之类的语句，也不会立即执行 只有借助前台Executor来『有序关闭』，才能解决这个问题	
	创建1(Thread)	直接操作线程【myThread.setDeamon(true); myThread.start()】	

		//: concurrency/SimpleDaemons.java
		// Daemon threads don't prevent the program from ending.
		import java.util.concurrent.*;
		import static net.mindview.util.Print.*;
		
		public class SimpleDaemons implements Runnable {
		  public void run() {
		  try {
		  while(true) {
		  TimeUnit.MILLISECONDS.sleep(100);
		  print(Thread.currentThread() + " " + this);
		  }
		  } catch(InterruptedException e) {
		  print("sleep() interrupted");
		  }
		  }
			public void run() {
			try {
			while(true) { //故意阻止线程自己结束，以突出演示效果
			TimeUnit.MILLISECONDS.sleep(100);
			print(...);
			}
			} catch (InterruptedException e) {
			...
			}
			}

		public static void main(String[] args) throws Exception {
		for(int i = 0; i < 10; i++) {
		Thread daemon = new Thread(new SimpleDaemons());
		daemon.setDaemon(true); // Must call before start()
		daemon.start();
		}
		print("All daemons started");
		TimeUnit.MILLISECONDS.sleep(175);
		}
		} /* Output: (Sample)
		All daemons started
		Thread[Thread-0,5,main] SimpleDaemons@530daa
		Thread[Thread-1,5,main] SimpleDaemons@a62fc3
		Thread[Thread-2,5,main] SimpleDaemons@89ae9e
		Thread[Thread-3,5,main] SimpleDaemons@1270b73
			Thread deamon = new Thread(new SimpleDeamons());
			deamon.setDeamon(true); //启动前设置
			deamon.start();
			TimeUnit.MILLISECONDS.sleep(175);
			return; //前台线程main() return时后台线程立即终止

```
Thread[Thread-4,5,main] SimpleDaemons@66aed0
Thread[Thread-5,5,main] SimpleDaemons@16caf43
Thread[Thread-6,5,main] SimpleDaemons@66848c
Thread[Thread-7,5,main] SimpleDaemons@8813f2
Thread[Thread-8,5,main] SimpleDaemons@1d58aae
Thread[Thread-9,5,main] SimpleDaemons@83cc67
...
'///:~
```

创建2 (Executors)

方法	用Executor，ExecutorService托管线程的生命周期
	(1) Executors.newXXXThreadPool(ThreadFactory f); (2) Thread MyThreadFactory#newThread(Runnable r)函数中 new Thread()，并将其setDeamon(true)

```
//: concurrency/DaemonFromFactory.java
// Using a Thread Factory to create daemons.
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DaemonFromFactory implements Runnable {
  public void run() {
    try {
      while(true) {
        TimeUnit.MILLISECONDS.sleep(100);
        print(Thread.currentThread() + " " + this);
      }
    } catch(InterruptedException e) {
      print("Interrupted");
    }
  }
```

```
//Runnable实现
public void run() {
 try {
   while(true) { //故意阻止线程自己结束，以突出演示效果
     TimeUnit.MILLISECONDS.sleep(100);
     print(...);
     }
 } catch (InterruptedException e) {
   ...
 }
}
```

```
//: net/mindview/util/DaemonThreadFactory.java
package net.mindview.util;
import java.util.concurrent.*;

public class DaemonThreadFactory implements ThreadFactory {
  public Thread newThread(Runnable r) {
    Thread t = new Thread(r);
    t.setDaemon(true);
    return t;
  }
} ///:~
```

```
//线程工厂
public class MyThreadFactory implements ThreadFactory {
 public Thread newThread(Runnable runnbale) {
   Thread thread = new Thread(runnable);
   thread.setDeamon(true); //在线程工厂中把线程设为后台
   return thread;
}
```

```
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool(
      new DaemonThreadFactory()); ⟶ 后台线程工厂
    for(int i = 0; i < 10; i++)
      exec.execute(new DaemonFromFactory()); ⟶ Runnable
    print("All daemons started");
    TimeUnit.MILLISECONDS.sleep(500); // Run for a while
  }
} /* (Execute to see output) *///:~
```

```
//使用线程工厂创建线程
ExecutorService execSvc
 = Executors.newCatchedThreadPool(new MyThreadFactory());
for (int i = 0; i < 10; ++i) {
 //参数是ThreadFactory而不再是Runnable
 execSvc.execute(new MyRunnable());
}
```

创建3 (传递)

方法	后台线程创建其他线程，创建出来的也都是后台线程（Thread.isDemaon()的取值默认会传递给子线程）

```
class DaemonSpawn implements Runnable {
  public void run() {
    while(true)
      Thread.yield();
  }
}
```

```
//普通的子线程，不setDeamon()
//isDeamon()属性来自创建这个线程的线程
//后面代码中DaemonSpawn的创建者Daemon是后台线程，因此
DaemonSpawn也是后台线程
class DaemonSpawn implements Runnable
```

```
class Daemon implements Runnable {
  private Thread[] t = new Thread[10];
  public void run() {
    for(int i = 0; i < t.length; i++) {
      t[i] = new Thread(new DaemonSpawn());
      t[i].start();
      printnb("DaemonSpawn " + i + " started, ")
    }
    for(int i = 0; i < t.length; i++)
      printnb("t[" + i + "].isDaemon() = " +
        t[i].isDaemon() + ", ");
    while(true)
      Thread.yield();
  }
}
```

```
class  Daemon implement Runnable {
  ...
  public void run() {
    for (int i = 0; i < threadArray.length; ++i) {
      threadArray[i] = new Thread(new DeamonSpawn());
      threadArray[i].start();
      printnb("threadArray[" + i + "].isDeamon() = " +
        threadArray[i].isDeamon() + ",");
    }
    while (true) //后台线程在前台线程结束时自动终止
      Thread.yield();
}
```

```
public class Daemons {
  public static void main(String[] args) throws Exception {
    Thread d = new Thread(new Daemon());
    d.setDaemon(true);
    d.start();
    printnb("d.isDaemon() = " + d.isDaemon() + ", ");
    // Allow the daemon threads to
    // finish their startup processes:
    TimeUnit.SECONDS.sleep(1);
  }
} /* Output: (Sample)
d.isDaemon() = true, DaemonSpawn 0 started, DaemonSpawn 1
started, DaemonSpawn 2 started, DaemonSpawn 3 started,
DaemonSpawn 4 started, DaemonSpawn 5 started, DaemonSpawn 6
started, DaemonSpawn 7 started, DaemonSpawn 8 started,
DaemonSpawn 9 started, t[0].isDaemon() = true,
t[1].isDaemon() = true, t[2].isDaemon() = true,
t[3].isDaemon() = true, t[4].isDaemon() = true,
t[5].isDaemon() = true, t[6].isDaemon() = true,
t[7].isDaemon() = true, t[8].isDaemon() = true,
t[9].isDaemon() = true,
*///:~
```

```
Thread daemonThread = new Thread(new Deamon());
daemonThread.setDaemon(true);
daemonThread.start();
```

不优雅退出

后台线程不优雅退出	前台线程关闭时，后台线程瞬间退出，无法做到优雅退出 即使是finally语句，也不会被执行，因此靠finally无法保证编写的清理代码被执行
如何有序退出	借助前台Executor来做到有序退出，参考章节末尾的
	这种行为是正确的，即便你基于前面对**finally**给出的承诺，并不希望出现这种行为，但情况仍将如此。当最后一个非后台线程终止时，后台线程会“突然”终止。因此一旦**main()**退出，JVM就会立即关闭所有的后台进程，而不会有任何你希望出现的确认形式。因为你不能以优雅的方式来关闭后台线程，所以它们几乎不是一种好的思想。非后台的**Executor**通常是一种更好的方式，因为**Executor**控制的所有任务可以同时被关闭。正如你将要在本章稍后看到的，在这种情况下，关闭将以有序的方式执行。

```
//: concurrency/DaemonsDontRunFinally.java
// Daemon threads don't run the finally clause
import java.util.concurrent.*;
import static net.mindview.util.Print.*;
```

```
public class DaemonsDontRunFinally {
  public static void main(String[] args) throws Exception {
    Thread t = new Thread(new ADaemon());
    t.setDaemon(true);
```

```
class ADaemon implements Runnable {
  public void run() {
    try {
      print("Starting ADaemon");
      TimeUnit.SECONDS.sleep(1);
    } catch(InterruptedException e) {
      print("Exiting via InterruptedException");
    } finally {
      print("This should always run?");
    }
  }
}
```

```
    t.start();
  }
} /* Output:
Starting ADaemon
*///:~
```

因为前台线程退出时，后台线程立即退出
finally语句中的代码并没有被执行

二、简单情况下创建线程

用途	简单场景下：写一个类、new一个对象（或调用一个函数）就能立刻启动线程

方法

方法			测试驱动
方法1	直接继承Thread （不用runnable就能启动线程） 自管理的Runnable （不用thread就能启动线程）	有构造函数未执行完就启动线程的并发隐患	
方法2	将方法1实现为内部类 封装在外部类构造函数中	有构造函数未执行完就启动线程的并发隐患	
方法3	将方法1实现为内部类 封装在单独函数中【推荐】	参考匿名内部类封装在单独的构造函数中	

```
public class ThreadVariations {
  public static void main(String[] args) {
    new InnerThread1("InnerThread1");
    new InnerThread2("InnerThread2");
    new InnerRunnable1("InnerRunnable1");
    new InnerRunnable2("InnerRunnable2");
    new ThreadMethod("ThreadMethod").runTask();
  }
} /* (Execute to see output) *///:~
```

方法1、2的隐患在于	1. 构造函数还没执行完，线程就启动了 2. 在并发访问的情况下，线程可能会访问到还没处于稳定状态的对象

直接继承Thread

代码

```
public class SimpleThread extends Thread {
  private int countDown = 5;
  private static int threadCount = 0;
  public SimpleThread() {
    // Store the thread name:
    super(Integer.toString(++threadCount));
    start();
  }
  public String toString() {
    return "#" + getName() + "(" + countDown + "), ";
  }
  public void run() {
    while(true) {
      System.out.print(this);
      if(--countDown == 0)
        return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new SimpleThread();
  }
} /* Output:
#1(5), #1(4), #1(3), #1(2), #1(1), #2(5), #2(4), #2(3),
#2(2), #2(1), #3(5), #3(4), #3(3), #3(2), #3(1), #4(5),
#4(4), #4(3), #4(2), #4(1), #5(5), #5(4), #5(3), #5(2),
```

方法（不推荐）

```
//不写Runnbale，直接继承Thread类
public class SimpleThread extends Thread {
 ...
 public SimpleThread() {
  //设置线程名称,可通过getName()函数得到线程的名称
  super(threadName);
  //构造函数未执行完线程就已经开始运行，可能带来风险
  start();
 }
 ...
 //覆盖run()函数
 public void run() {
  ...
 }
}
```

```
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

// Using a named inner class:
class InnerThread1 {
  private int countDown = 5;
  private Inner inner;
  private class Inner extends Thread {
    Inner(String name) {
      super(name);
      start();
    }
    public void run() {
      try {
        while(true) {
          print(this);
          if(--countDown == 0) return;
          sleep(10);
        }
      } catch(InterruptedException e) {
        print("interrupted");
      }
    }
    public String toString() {
      return getName() + ": " + countDown;
    }
  }
  public InnerThread1(String name) {
    inner = new Inner(name);
  }
}
```

```
class Demo {
 //将Thread子类实现为内部类
 private InnerThread innerThread;
 private class InnerThread extends Thread {
  InnerThread (String name) {
   super(name);
   //构造函数未执行完线程就已经开始运行，可能带来风险
   start();
  }
  public void run() {
   ...
  }
 }
 //外部类的构造函数中创建线程
 public Demo(String threadName) {
  innerThread = new InnerThread(name);
 }
}
```

```
// Using an anonymous inner class:
class InnerThread2 {
  private int countDown = 5;
  private Thread t;
  public InnerThread2(String name) {
    t = new Thread(name) {
      public void run() {
        try {
          while(true) {
            print(this);
            if(--countDown == 0) return;
            sleep(10);
          }
        } catch(InterruptedException e) {
          print("sleep() interrupted");
        }
```

```
class Demo {
 private Thread thread;
 //直接在外部类构造函数中实现Thread的子类，用局部匿名内部类的方式
 public Demo(String threadName) {
  thread = new Thread(threadName) {
   public void run() {
    ..
   }
  };
  //构造函数未执行完线程就已经开始运行，可能带来风险
  thread.start();
 }
}
```

```
    }
    public String toString() {
      return getName() + ": " + countDown;
    }
  };
  t.start();
 }
}
```

自管理的Runnable

代码

```
public class SelfManaged implements Runnable {
  private int countDown = 5;
  private Thread t = new Thread(this);
  public SelfManaged() { t.start(); }
  public String toString() {
    return Thread.currentThread().getName() +
      "(" + countDown + "), ";
  }
  public void run() {
    while(true) {
      System.out.print(this);
      if(--countDown == 0)
        return;
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++)
      new SelfManaged();
  }
} /* Output:
Thread-0(5), Thread-0(4), Thread-0(3), Thread-0(2), Thread-
0(1), Thread-1(5), Thread-1(4), Thread-1(3), Thread-1(2),
Thread-1(1), Thread-2(5), Thread-2(4), Thread-2(3), Thread-
2(2), Thread-2(1), Thread-3(5), Thread-3(4), Thread-3(3),
Thread-3(2), Thread-3(1), Thread-4(5), Thread-4(4), Thread-
4(3), Thread-4(2), Thread-4(1),
*///:~
```

方法（不推荐）

```
public class SelfManagedRunnable implements Runnable {
 //封装一个This，将自身传给这个Thread
 private Thread thread = new Thread(this);
 //构造函数未执行完线程就已经开始运行，可能带来风险
 public SelfManagedRunnable() { thread.start(); }
 //run()
 public void run() {
   ...
 }
 ...
}
```

```
// Using a named Runnable implementation:
class InnerRunnable1 {
  private int countDown = 5;
  private Inner inner;
  private class Inner implements Runnable {
    Thread t;
    Inner(String name) {
      t = new Thread(this, name);
      t.start();
    }
    public void run() {
      try {
        while(true) {
          print(this);
          if(--countDown == 0) return;
          TimeUnit.MILLISECONDS.sleep(10);
        }
      } catch(InterruptedException e) {
        print("sleep() interrupted");
      }
    }
    public String toString() {
      return t.getName() + ": " + countDown;
    }
  }
  public InnerRunnable1(String name) {
    inner = new Inner(name);
  }
}
```

```
//将上一个例子中的自管理Runnable实现为内部类
class Demo {
 ...
 private SelfManagedRunnable selfManagedRunnable;
 private class SelfManagedRunnable implements Runnable {
   Thread thread;
   SelfManagedRunnable(String threadName) {
     thread = new Thread(this, threadName);
     thread.start();
   }
   public void run() {
     ...
   }
 }
 ...
 //构造函数未执行完线程就已经开始运行，可能带来风险
 public Demo(String threadName) {
   selfManagedRunnable = new SelfManagedRunnable(threadName);
 }
}
```

```
// Using an anonymous Runnable implementation:
class InnerRunnable2 {
  private int countDown = 5;
  private Thread t;
  public InnerRunnable2(String name) {
    t = new Thread(new Runnable() {
      public void run() {
        try {
          while(true) {
            print(this);
            if(--countDown == 0) return;
            TimeUnit.MILLISECONDS.sleep(10);
          }
        } catch(InterruptedException e) {
          print("sleep() interrupted");
        }
      }
      public String toString() {
        return Thread.currentThread().getName() +
          ": " + countDown;
      }
    }, name);
    t.start();
  }
}
```

```
//将上一个例子中的自管理Runnable实现为局部匿名内部类
class Demo {
 private Thread thread;
 public Demo(String threadName) {
   thread = new Thread(new Runnable() {
     public void run() {
       ...
     }
   }, threadName);
   //构造函数未执行完线程就已经开始运行，可能带来风险
   thread.start();
 }
}
```

风险消除

用途	避免直接继承Thread或自管理的Runnable的例子中，线程运行时构造函数还没有执行完毕所带来的风险
方法	1.线程代码（不需要Runnable的Thead，或自管理Runnable）封装在匿名内部类中 2.匿名内部类对象并没有交给构造函数使用，而是由一个普通函数来调用 这样当myThread.start()时，所有构造函数都已经执行完了

代码

```
// A separate method to run some code as a task:
class ThreadMethod {
  private int countDown = 5;
  private Thread t;
  private String name;
  public ThreadMethod(String name) { this.name = name; }
  public void runTask() {
    if(t == null) {
      t = new Thread(name) {
        public void run() {
          try {
            while(true) {
              print(this);
              if(--countDown == 0) return;
              sleep(10);
            }
          } catch(InterruptedException e) {
            print("sleep() interrupted");
```

```
class Demo {
  private Thread thread;
  //封装在普通函数中
  public void runTask(String threadName) {
    if (null == thread) {
      thread = new Thread(threadName) {
        public void run() {
          ...
        }
      };
      //线程启动时所有的构造函数都已经执行完了，可避免前面所提风险
      thread.start();
```

```
    }
  }
  public String toString() {
    return getName() + ": " + countDown;
  }
 };
 t.start();
 }
 }
}
```

```
 }
 }
}
```

三、Join一个线程，以及线程中断

Join	用途	让一个线程等待另一个线程结束（isAlive()为false时），可添加超时参数
	阻塞	调用join的线程会被阻塞，阻塞状态通过中断也可以结束（不一定是线程终止），这与调用sleep()，Future<ResultType>#get()类似

线程中断

内容：如何中断另一个线程

方法：

线程A中断线程B	线程A的代码中调用 threadB.interrupt();
作用	如果线程B正阻塞在sleep()/join()之类的方法上，可以被唤醒
线程B需要	线程B的阻塞函数会抛出InterruptedException，线程B需要捕获InterruptedException 另外异常无法跨线程传递

替代：java.util.concurrent库的CyclicBarrier也可以处理线程件等待的问题，封装了更强大的处理框架

例子：

线程A（被join）

```
import static net.mindview.util.Print.*;

class Sleeper extends Thread {
  private int duration;
  public Sleeper(String name, int sleepTime) {
    super(name);
    duration = sleepTime;
    start();
  }
  public void run() {
    try {
      sleep(duration);
    } catch(InterruptedException e) {
      print(getName() + " was interrupted. " +
        "isInterrupted(): " + isInterrupted());
      return;
    }
    print(getName() + " has awakened");
  }
}
```

```
// 这个写法参考《二、简单情况下创建线程》
class SleepThread extends Thread {
 public SleepThread (String name, int sleepTime) {
  super(name); //设置线程名称
  ...
 }
 public void run() {
  try {
   //被中断时sleep()会抛出InterruptedException
   sleep(duration);
  } catch (InterruptedException e) {
   ...
   //这个例子选择了结束线程运行
   return;
  }
 }
}
```

线程B（join A）

```
class Joiner extends Thread {
  private Sleeper sleeper;
  public Joiner(String name, Sleeper sleeper) {
    super(name);
    this.sleeper = sleeper;
    start();
  }
  public void run() {
   try {
      sleeper.join();
    } catch(InterruptedException e) {
      print("Interrupted");
    }
    print(getName() + " join completed");
  }
}
```

```
// 这个写法参考《二、简单情况下创建线程》
class JoinerThread extends Thread {
 private SleepThread sleepThread;
 public Joiner(String threadName, SleepThread sleepThread) {
  super(threadName)
  this.sleepThread = sleepThread;
  start();
 }
 public void run() {
  try {
   //join阻塞在等待sleepThread结束时，也可以被其他线程中断
   sleepThread.join();
  } catch (InterruptedException e) {
   print("Interrupted");
  }
 }
}
```

main

```
public class Joining {
  public static void main(String[] args) {
    Sleeper
      sleepy = new Sleeper("Sleepy", 1500),
      grumpy = new Sleeper("Grumpy", 1500);
    Joiner
      dopey = new Joiner("Dopey", sleepy),
      doc = new Joiner("Doc", grumpy);
    grumpy.interrupt();
  }
} /* Output:
Grumpy was interrupted. isInterrupted(): false
Doc join completed
Sleepy has awakened
Dopey join completed
*///:~
```

```
SleepThread sleepThread = new SleepThread("sleep", 1500);
JoinerThread joinThread
  = new JoinerThread("joiner", sleepThread);
sleepThread.interrupt();
```

四、线程用途演示：创建有响应的用户界面

创建有响应界面	只是一个非常简单的演示，说明线程的好处而已

代码

不可响应的界面

```
//: concurrency/ResponsiveUI.java
// User interface responsiveness.
// {RunByHand}

class UnresponsiveUI {
  private volatile double d = 1;
  public UnresponsiveUI() throws Exception {
    while(d > 0)
      d = d + (Math.PI + Math.E) / d;
    System.in.read(); // Never gets here
  }
}
```

可响应的界面：一个线程后台计算、一个线程与用户交互

```
public class ResponsiveUI extends Thread {
  private static volatile double d = 1;
  public ResponsiveUI() {
    setDaemon(true);
    start();
  }
  public void run() {
    while(true) {
      d = d + (Math.PI + Math.E) / d;
    }
  }
  public static void main(String[] args) throws Exception {
    //! new UnresponsiveUI(); // Must kill this process
    new ResponsiveUI();
    System.in.read();
    System.out.println(d); // Shows progress
  }
} ///:~
```

五、线程组：已废弃不推荐使用、推荐使用Executor替代、读老代码参考《Java编程思想》第二版

线程组	“最好把线程组看成是一次不成功的尝试，你只要忽略它就好了。”
	已被Executor替代
	此你就不再需要了解有关线程组的任何知识了（除非要理解遗留代码，请查看可以从www.MindView.net下载的《Thinking in Java（2nd Edition）》，以了解线程组的细节）。

六、捕获异常：线程内部捕获 & 用Executor捕获线程内部泄露的异常

解决问题	异常在线程内部捕获	程序按设计预期正常运行，参考之前的例子，参考线程中断
	异常未能在线程内部捕获	线程（因为无法被其他线程或main函数捕获）会泄露到console，导致程序崩溃

异常传递给console

```
//: concurrency/ExceptionThread.java
// {ThrowsException}
import java.util.concurrent.*;

public class ExceptionThread implements Runnable {
  public void run() {
    throw new RuntimeException();
  }
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool();
    exec.execute(new ExceptionThread());
  }
} ///:~
```

如下（将某些限定符修整为适合显示）：

```
java.lang.RuntimeException
        at ExceptionThread.run(ExceptionThread.java:7)
        at ThreadPoolExecutor$Worker.runTask(Unknown Source)
        at ThreadPoolExecutor$Worker.run(Unknown Source)
        at java.lang.Thread.run(Unknown Source)
```

main函数加try-catch块也没用

```
//: concurrency/NaiveExceptionHandling.java
// {ThrowsException}
import java.util.concurrent.*;
public class NaiveExceptionHandling {
  public static void main(String[] args) {
    try {
      ExecutorService exec =
        Executors.newCachedThreadPool();
      exec.execute(new ExceptionThread());
    } catch(RuntimeException ue) {
      // This statement will NOT execute!
      System.out.println("Exception has been handled!");
    }
  }
} ///:~
```

		需要借助Executor的跨线程异常捕获机制，来捕获线程内部漏捕的异常
方法	JavaSE5	线程组（已废弃）
	JavaSE6	使用Executors捕获异常，步骤如下

Step1	//用Executors创建ExecutorService时传入ThreadFactory
Step2	//ThreadFactory#newThread(Runnable)中为线程设置UncaughtExceptionHandler

Step3	//UncaughtExceptionHandler#uncaughtException(Thread, Throwable)中添加异常处理代码
	class MyUncaughtExceptionHandler implement Thread.UncaughtExceptionHandler {
	public void uncaughtException(Thread t, Throwable e) {...}
	}

例子1（为Thread设置）

线程

```
//: concurrency/CaptureUncaughtException.java
import java.util.concurrent.*;

class ExceptionThread2 implements Runnable {
  public void run() {
    Thread t = Thread.currentThread();
    System.out.println("run() by " + t);
    System.out.println(
      "eh = " + t.getUncaughtExceptionHandler());
    throw new RuntimeException();
  }
}
```

```
class ExceptionThread implements Runnable {
  public void run() {
    ...
    throw new RuntimeException(); //会抛异常
  }
}
```

ThreadFactory

```
class MyUncaughtExceptionHandler implements
Thread.UncaughtExceptionHandler {
  public void uncaughtException(Thread t, Throwable e) {
    System.out.println("caught " + e);
  }
}

class HandlerThreadFactory implements ThreadFactory {
  public Thread newThread(Runnable r) {
    System.out.println(this + " creating new Thread");
    Thread t = new Thread(r);
    System.out.println("created " + t);
    t.setUncaughtExceptionHandler(
      new MyUncaughtExceptionHandler());
    System.out.println(
      "eh = " + t.getUncaughtExceptionHandler());
    return t;
  }
}
```

```
//Thread.UncaughtExceptionHandler
class MyUncaughtExceptionHandler implements
        Thread.UncaughtExceptionHandler {
  public void uncaughtException(Thread t, Throwable e) {
    System.out.println("caught " + e);
  }
}
//ThreadFactory
class MyThreadFactory implements ThreadFactory {
  public Thread newThread(Runnable runnable) {
    Thread thread = new Thread(runnable);
    thread.setUncaughtExceptionHandler(
      new MyUncaughtExceptionHandler());
    return thread;
  }
}
```

创建线程

```
public class CaptureUncaughtException {
  public static void main(String[] args) {
    ExecutorService exec = Executors.newCachedThreadPool(
      new HandlerThreadFactory());
    exec.execute(new ExceptionThread2());
  }
} /* Output: (90% match)
HandlerThreadFactory@de6ced creating new Thread
created Thread[Thread-0,5,main]
eh = MyUncaughtExceptionHandler@1fb8ee3
run() by Thread[Thread-0,5,main]
eh = MyUncaughtExceptionHandler@1fb8ee3
caught java.lang.RuntimeException
*///:~
```

```
ExecutorService execSvc
  = Executors.newCachedThreadPool(new MyThreadFactory());
execSvc.execute(new ExceptionThread);
execSvc.shutdown();
```

例子2（设置全局默认）

解决的问题	上面的方法要为每个Executors逐一设置，如何设置一个缺省的UncaughtExceptionHandler	
方法	函数	Thread.setDefaultUncaughtExceptionHandler(...) //静态函数
	生效场景	没有可供这个Thread使用的UncaughtExceptionHandler时，才使用上面设置的默认handler
	JVM检查顺序	优先用Thread专有版本：如果Executors通过ThreadFactory为Thread设置了专有的handler，使用该专有版本 其次考虑线程租的版本：如果线程组（Java SE5）设置了handler，用线程组设置的handler 最后考虑全局默认版本：如果前两个版本都没有，使用全局默认的handler

代码

```
//: concurrency/SettingDefaultHandler.java
import java.util.concurrent.*;

public class SettingDefaultHandler {
  public static void main(String[] args) {
    Thread.setDefaultUncaughtExceptionHandler(
```

```
        new MyUncaughtExceptionHandler());
      ExecutorService exec = Executors.newCachedThreadPool();
      exec.execute(new ExceptionThread());
    }
} /* Output:
caught java.lang.RuntimeException
*///:~
```

这个处理器只有在不存在线程专有的未捕获异常处理器的情况下才会被调用。系统会检查线程专有版本，如果没有发现，则检查线程组是否有其专有的**uncaughtException()**方法，如果也没有，再调用**defaultUncaughtExceptionHandler**。