第一章 线程管理 Java 7并发编程实战手册读书笔记

第一章 线程管理

1.进程与线程

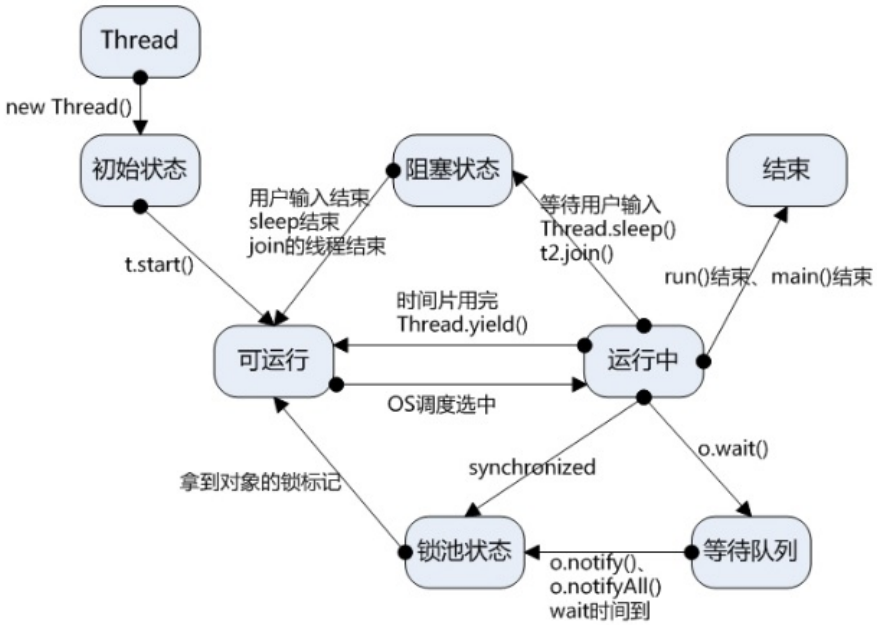
a.进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.

b.线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.

c.Java VM（java虚拟机）在运行时启动了一个进程---java.exe,该进程在执行时，至少有一个线程在控制着java程序的运行，并且这个线程存在于Java的main函数中，该线程称之为**Java的主线程**。

d.扩展：在JVM运行时，除了main函数中的线程在运行外，还有JVM中负责Java垃圾回收的线程在运行。因此，Java不是单线程运行程序。

e.线程间的状态



2.并发与并行

a.并行性包含同时性和并发性，前者是指两个或多个事件在**同一时刻**发生，后者是指两个或多个事件在**同一时间段内**发生。

b.计算机操作系统中把并行性和并发性明显区分开，主要是从微观的角度来说的，具体是指进程的并行性（多处理机的情况下，多个进程同时运行）和并发性（单处理机的情况下，多个进程在同一时间间隔运行的）。

3.创建线程的两种方式

(1).两种方法

a.继承Thread类，并且覆盖run()方法

```
class Calculator extends Thread {
 private int number;
 public Calculator(int number) {
  this.number=number;
 }
 @Override
 public void run() {
  for (int i=1; i<=10; i++){
   System.out.printf("%s: %d * %d = %d\n",Thread.currentThread().getName(),number,i,i*number);
  }
 }
}

public class Main {
 public static void main(String[] args) {
```

```
 for (int i=1; i<=10; i++){
  Calculator calculator=new Calculator(i);
  calculator.start();
 }
 }
}
```

结果

```
<terminated> Main (1) [Java Application] D:\Java\jre\bin\javaw.exe (2016年11月19日
Thread-6: 7 * 2 = 14
Thread-6: 7 * 3 = 21
Thread-6: 7 * 4 = 28
Thread-6: 7 * 5 = 35
Thread-6: 7 * 6 = 42
Thread-6: 7 * 7 = 49
Thread-6: 7 * 8 = 56
Thread-6: 7 * 9 = 63
Thread-6: 7 * 10 = 70
Thread-4: 5 * 1 = 5
Thread-4: 5 * 2 = 10
Thread-4: 5 * 3 = 15
Thread-4: 5 * 4 = 20
Thread-4: 5 * 5 = 25
Thread-4: 5 * 6 = 30
Thread-4: 5 * 7 = 35
Thread-4: 5 * 8 = 40
Thread-4: 5 * 9 = 45
Thread-4: 5 * 10 = 50
```

b.实现Runable接口

```
public class Calculator implements Runnable {
 private int number;
 public Calculator(int number) {
  this.number=number;
 }
 @Override
 public void run() {
  for (int i=1; i<=10; i++){
   System.out.printf("%s: %d * %d = %d\n",Thread.currentThread().getName(),number,i,i*number);
  }
 }
}

//主类
public class Main {
 public static void main(String[] args) {
  for (int i=1; i<=10; i++){
   Calculator calculator=new Calculator(i);
   Thread thread=new Thread(calculator);
   thread.start();
  }
 }
}
```

结果

```
<terminated> Main [Java Application] D:\Java\jre\bin\javaw.exe (2016年11月19日 上午11:28:22)
Thread-1: 2 * 1 = 2
Thread-0: 1 * 2 = 2
Thread-0: 1 * 3 = 3
Thread-0: 1 * 4 = 4
Thread-0: 1 * 5 = 5
Thread-0: 1 * 6 = 6
Thread-0: 1 * 7 = 7
Thread-0: 1 * 8 = 8
Thread-0: 1 * 9 = 9
Thread-0: 1 * 10 = 10
Thread-1: 2 * 2 = 4
Thread-1: 2 * 3 = 6
Thread-1: 2 * 4 = 8
Thread-1: 2 * 5 = 10
Thread-1: 2 * 6 = 12
Thread-1: 2 * 7 = 14
Thread-1: 2 * 8 = 16
Thread-1: 2 * 9 = 18
Thread-1: 2 * 10 = 20
```

结果分析

start()方法的调用后并不是立即执行多线程代码，而是使得该线程变为可运行态（Runnable），什么时候运行是由操作系统决定的。从程序运行的结果可以发现，多线程程序是乱序执行。

(2).这两种方法哪种更好？

实现Runnable接口比继承Thread类所具有的优势，所以一般都用实现Runnable的方法创建线程

1）适合多个相同的程序代码的线程去处理同一个资源

2）可以避免java中的单继承的限制

3）增加程序的健壮性，代码可以被多个线程共享，代码和数据独立

4）线程池只能放入实现Runable或callable类线程，不能直接放入继承Thread的类

4.线程信息的获取和设置

ID：该属性表示每个线程的唯一标识；

Name：该属性存储每个线程的名称；

Priority：该属性存储每个Thread对象的优先级。线程优先级分1到10十个级别，1表示最低优先级，10表示最高优先级。并不推荐修改线程的优先级，但是如果确实有这方面的需求，也可以尝试一下。

Status: 该属性存储线程的状态。线程共有六种不同的状态：新建（new）、运行（runnable）、阻塞（blocked）、等待（waiting）、限时等待（time waiting）或者终止（terminated）。线程的状态必定是其中一种。

5.线程的中断

(1).通过调用isInterrupted()方法判断一个线程是否已经中断

```
public class PrimeGenerator extends Thread{
      @Override
 public void run() {
  long number=1L;
  while (true) {
   if (isPrime(number)) {
    System.out.printf("Number %d is Prime\n",number);
   }

   if (isInterrupted()) {
    System.out.printf("The Prime Generator has been Interrupted\n");
    return;
   }
   number++;
  }
 }
 private boolean isPrime(long number) {
  if (number <=2) {
   return true;
  }
  for (long i=2; i<number; i++){
   if ((number % i)==0) {
    return false;
   }
  }
  return true;
 }
}

public class Main {
 public static void main(String[] args) {
  Thread task=new PrimeGenerator();
  task.start();

  // Wait 5 seconds
  try {
   TimeUnit.SECONDS.sleep(5);
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  // Interrupt the prime number generator
  task.interrupt();
```

```
}
}
```

结果

```
<terminated> Main (3) [Java Application] D:\Java\jre\bin\javaw.exe (2016年11月19日
Number 50101 is Prime
Number 50111 is Prime
Number 50119 is Prime
Number 50123 is Prime
Number 50129 is Prime
Number 50131 is Prime
Number 50147 is Prime
Number 50153 is Prime
Number 50159 is Prime
Number 50177 is Prime
Number 50207 is Prime
Number 50221 is Prime
Number 50227 is Prime
Number 50231 is Prime
Number 50261 is Prime
Number 50263 is Prime
Number 50273 is Prime
Number 50287 is Prime
The Prime Generator has been Interrupted
```

(2).Java提供了InterruptedException异常，当检查到线程中断的时候，就会抛出这个异常，因此，通过捕获这个异常，就可以控制线程中断（当线程里实现了复杂的算法，或者线程里有递归调用时，可以使用这种方式）。

```
public class FileSearch implements Runnable {

 private String fileName;
 public FileSearch(String initPath, String fileName) {
  this.initPath = initPath;
  this.fileName = fileName;
 }
 @Override
 public void run() {
  File file = new File(initPath);
  if (file.isDirectory()) {
   try {
    directoryProcess(file);
   } catch (InterruptedException e) {
    System.out.printf("%s: The search has been interrupted",Thread.currentThread().getName());
    cleanResources();
   }
  }
 }

     private void cleanResources() {

 }

 private void directoryProcess(File file) throws InterruptedException {

  // Get the content of the directory
  File list[] = file.listFiles();
  if (list != null) {
   for (int i = 0; i < list.length; i++) {
    if (list[i].isDirectory()) {
     // If is a directory, process it
     directoryProcess(list[i]);
    } else {
     // If is a file, process it
     fileProcess(list[i]);
    }
   }
  }
  // Check the interruption
  if (Thread.interrupted()) {
```

```
   throw new InterruptedException();
  }
 }

 private void fileProcess(File file) throws InterruptedException {

  if (file.getName().equals(fileName)) {
   System.out.printf("%s : %s\n",Thread.currentThread().getName() ,file.getAbsolutePath());
  }

  // Check the interruption
  if (Thread.interrupted()) {
   throw new InterruptedException();
  }
 }
}
```

6.线程休眠和恢复

(1)线程休眠的两种方式

a.Thread.sleep()

1）线程的sleep()方法表示线程被挂起多少毫秒，此时，**当前线程不会释放锁。**

2）sleep()的另一种使用方式是通过TimeUtil枚举元素进行调用，但是此方法接收的参数是**秒**，最后转换成毫秒，如TimeUnit.SECONDS.sleep(1)表示线程休眠1秒

b.Object.wait()

当线程执行到wait()方法上，当前线程会**释放锁**，此时其他线程可以占有该锁，一旦wait()方法执行完成，当前线程又继续持有该锁，直到执行完该锁的作用域.

wait()退出的条件：

1）达到了wait(long timout)指定的时间

2）其他线程调用了notify()或者notifyAll()

如果在同步代码块之外调用wait()，JVM将会抛出IllegalMonitorStateException

```
public class FileClock implements Runnable {

 @Override
 public void run() {
  for (int i = 0; i < 10; i++) {
   System.out.printf("%s\n", new Date());
   try {
    // Sleep during one second
    TimeUnit.SECONDS.sleep(1);
   } catch (InterruptedException e) {
    System.out.printf("The FileClock has been interrupted");
   }
  }
 }
}

public class Main {

 public static void main(String[] args) {
  FileClock clock=new FileClock();
  Thread thread=new Thread(clock);
  thread.start();
  try {
   // Waits five seconds
   TimeUnit.SECONDS.sleep(5);
  } catch (InterruptedException e) {
   e.printStackTrace();
  };
  // Interrupts the Thread
  thread.interrupt();
 }
}
```

结果

```
<terminated> Main (5) [Java Application] D:\Java\jre\bin\javaw.exe (2016年11月19日
Sat Nov 19 16:20:13 CST 2016
Sat Nov 19 16:20:15 CST 2016
Sat Nov 19 16:20:16 CST 2016
Sat Nov 19 16:20:17 CST 2016
Sat Nov 19 16:20:18 CST 2016
The FileClock has been interruptedSat Nov 19 16:20:18 CST 2016
Sat Nov 19 16:20:19 CST 2016
Sat Nov 19 16:20:20 CST 2016
Sat Nov 19 16:20:21 CST 2016
Sat Nov 19 16:20:22 CST 2016
```

7.等待线程的终止

1) join():指等待t线程终止

2) join()的作用是： “等待该线程终止” ，这里需要理解的就是该线程是指的主线程等待子线程的终止。也就是在子线程调用了join()方法后面的代码，只有等到子线程结束了才能执行。

3) 在很多情况下，主线程生成并起动了子线程，如果子线程里要进行大量的耗时的运算，主线程往往将于子线程之前结束，但是如果主线程处理完其他的事务后，需要用到子线程的处理结果，也就是主线程需要等待子线程执行完成之后再结束，这个时候就要用到join()方法了。

Demo1---主线程使用join()方法

```
public class DataSourcesLoader implements Runnable {

 @Override
 public void run() {

  // Writes a messsage
  System.out.printf("Begining data sources loading: %s\n",new Date());
  // Sleeps four seconds
  try {
   TimeUnit.SECONDS.sleep(4);
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  // Writes a message
  System.out.printf("Data sources loading has finished: %s\n",new Date());
 }
}
public class NetworkConnectionsLoader implements Runnable {

 @Override
 public void run() {
  // Writes a message
  System.out.printf("Begining network connections loading: %s\n",new Date());
  // Sleep six seconds
  try {
   TimeUnit.SECONDS.sleep(6);
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  // Writes a message
  System.out.printf("Network connections loading has finished: %s\n",new Date());
 }
}

public class Main {

 public static void main(String[] args) {

  // Creates and starts a DataSourceLoader runnable object
  DataSourcesLoader dsLoader = new DataSourcesLoader();
  Thread thread1 = new Thread(dsLoader,"DataSourceThread");
  thread1.start();
```

```
 // Creates and starts a NetworkConnectionsLoader runnable object
 NetworkConnectionsLoader ncLoader = new NetworkConnectionsLoader();
 Thread thread2 = new Thread(ncLoader,"NetworkConnectionLoader");
 thread2.start();

 // Wait for the finalization of the two threads
 try {
  thread1.join();
  thread2.join();
 } catch (InterruptedException e) {
  e.printStackTrace();
 }

 // Waits a message
 System.out.printf("Main: Configuration has been loaded: %s\n",new Date());
}
}
```

```
<terminated> Main (6) [Java Application] D:\Java\jre\bin\javaw.exe (2016年11月19日 下
Begining data sources loading: Sat Nov 19 16:41:39 CST 2016
Begining network connections loading: Sat Nov 19 16:41:39 CST 2016
Data sources loading has finished: Sat Nov 19 16:41:43 CST 2016
Network connections loading has finished: Sat Nov 19 16:41:45 CST 2016
Main: Configuration has been loaded: Sat Nov 19 16:41:45 CST 2016
```

Demo2---主线程不适用jion()方法，将Main的main()函数换成下面的代码，其他两个线程类不变

```
public class Main {

 public static void main(String[] args) {

 // Creates and starts a DataSourceLoader runnable object
 DataSourcesLoader dsLoader = new DataSourcesLoader();
 Thread thread1 = new Thread(dsLoader,"DataSourceThread");
 thread1.start();

 // Creates and starts a NetworkConnectionsLoader runnable object
 NetworkConnectionsLoader ncLoader = new NetworkConnectionsLoader();
 Thread thread2 = new Thread(ncLoader,"NetworkConnectionLoader");
 thread2.start();

          //此处没有使用join()方法
 // Waits a message
 System.out.printf("Main: Configuration has been loaded: %s\n",new Date());
 }
}
```

此时结果为：可以看到：Main: Configuration has been loaded 还没有等待两个线程结束就已经打印出来了，这是因为主线程（Main类main函数中的线程是主线程）没有使用join()方法等待两个子线程结束（这段代码中子线程是thread1 和thread2）

```
<terminated> Main (6) [Java Application] D:\Java\jre\bin\javaw.exe (2016年11月19日 下
Main: Configuration has been loaded: Sat Nov 19 16:44:19 CST 2016
Begining data sources loading: Sat Nov 19 16:44:19 CST 2016
Begining network connections loading: Sat Nov 19 16:44:19 CST 2016
Data sources loading has finished: Sat Nov 19 16:44:23 CST 2016
Network connections loading has finished: Sat Nov 19 16:44:25 CST 2016
```

8.守护线程的创建和运行

1）在Java中有两类线程：User Thread(用户线程)、Daemon Thread(守护线程)。用个比较通俗的比喻，任何一个守护线程都是整个JVM中所有非守护线程的保姆：

2）只要当前JVM实例中尚存在任何一个非守护线程没有结束，守护线程就全部工作；只有当最后一个非守护线程结束时，守护线程随着JVM一同结束工作。

3）Daemon的作用是为其他线程的运行提供便利服务，守护线程最典型的应用就是 GC (垃圾回收器)，它就是一个很称职的守护者。

4）User和Daemon两者几乎没有区别，唯一的不同之处就在于虚拟机的离开：如果 User Thread已经全部退出运行了，只剩下Daemon Thread存在了，虚拟机也就退出了。 因为没有了被守护者，Daemon也就没有工作可做了，也就没有继续运行程序的必要了。

5）守护线程并非只有虚拟机内部提供，用户在编写程序时也可以自己设置守护线程。

6）通过setDaemon()方法设置线程为守护线程，但是只能在改线程的start()方法调用之前设置，一旦线程开始，讲不能再修改守护状

态。一般在构造函数里面设置线程为守护线程。
7）isDaemon()方法被用来检测一个线程是不是守护线程，返回true表示该线程是守护线程。

Demo

```
public class CleanerTask extends Thread {
 private Deque<Event> deque;

 /**
  * Constructor of the class
  * @param deque data structure that stores events
  */
 public CleanerTask(Deque<Event> deque) {
  this.deque = deque;
  // Establish that this is a Daemon Thread
  setDaemon(true);  //设置为守护线程，在构造函数中设置，可以保证这个类被实例化时，就设置为守护线程，所以不论怎样，都会在改线程的start()方法之前.
 }

 @Override
 public void run() {
  while (true) {
   Date date = new Date();
   clean(date);
  }
 }

 private void clean(Date date) {
  //TO DO
}
```

9.线程中不可控异常的处理

(1)Java异常的种类

a.Exception

一般分为Checked异常和Runtime异常，所有RuntimeException类及其子类的实例被称为Runtime异常，不属于该范畴的异常则被称为CheckedException。

1）Checked异常

只有java语言提供了Checked异常，Java认为Checked异常都是可以被处理的异常，所以Java程序必须显示处理Checked异常。如果程序没有处理Checked异常，该程序在编译时就会发生错误无法编译。对Checked异常处理方法有两种

1 当前方法知道如何处理该异常，则用try...catch块来处理该异常。

2 当前方法不知道如何处理，则在定义该方法是声明抛出该异常。

2）RuntimeException

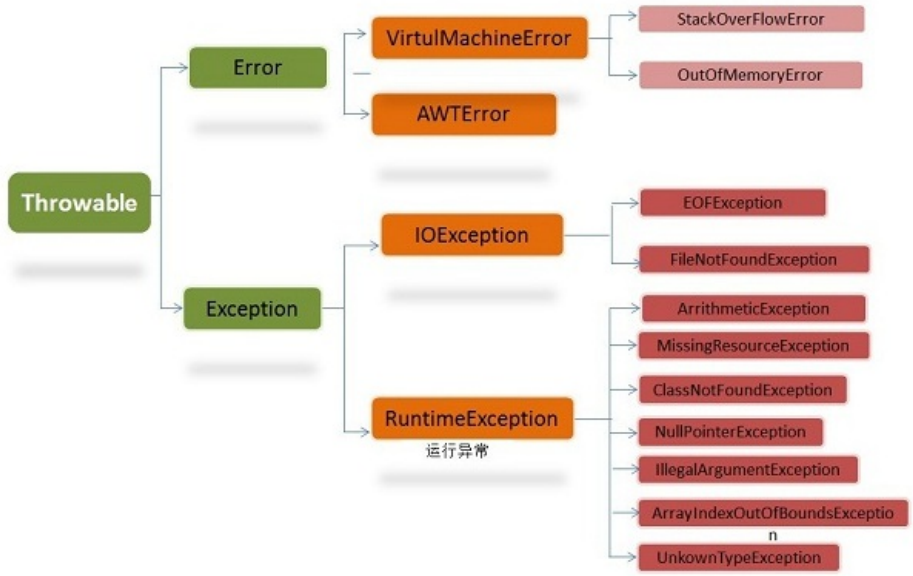
Runtime如除数是0和数组下标越界等，其产生频繁，处理麻烦，若显示申明或者捕获将会对程序的可读性和运行效率影响很大。所以由系统自动检测并将它们交给缺省的异常处理程序。

b.Error

当程序发生不可控的错误时，通常做法是通知用户并中止程序的执行。与异常不同的是Error及其子类的对象不应被抛出。

Error是throwable的子类，代表编译时间和系统错误，用于指示合理的应用程序不应该试图捕获的严重问题。

Error由Java虚拟机生成并抛出，包括动态链接失败，虚拟机错误等。程序对其不做处理。



(2)线程中对异常的处理

a.非运行时异常（Unchecked Exception）

必须捕获并处理，因为run()方法并不支持throws语句

b.运行时异常（Runtime Exception）

1）默认在控制台输出异常信息

2）设置UncaughtException异常处理器来自定义处理操作（实现Thread.UncaughtExceptionHandler接口并且实现UncaughtException方法）

```
public class ExceptionHandler implements UncaughtExceptionHandler {

 @Override
 public void uncaughtException(Thread t, Throwable e) {
  System.out.printf("An exception has been captured\n");
  System.out.printf("Thread: %s\n",t.getId());
  System.out.printf("Exception: %s: %s\n",e.getClass().getName(),e.getMessage());
  System.out.printf("Stack Trace: \n");
  e.printStackTrace(System.out);
  System.out.printf("Thread status: %s\n",t.getState());
 }
}
public class Task implements Runnable {
 @Override
 public void run() {
  // The next instruction always throws and exception
  int numero=Integer.parseInt("TTT");
 }
}

public class Main {

 public static void main(String[] args) {
  Task task=new Task();
  Thread thread=new Thread(task);
  thread.setUncaughtExceptionHandler(new ExceptionHandler());
  thread.start();
  try {
   thread.join();
  } catch (InterruptedException e) {
   e.printStackTrace();
  }

  System.out.printf("Thread has finished\n");

 }
}
```

结果

```
<terminated> Main (8) [Java Application] D:\Java\jre\bin\javaw.exe (2016年11月19日 下午7:25:20)
An exception has been captured
Thread: 9
Exception: java.lang.NumberFormatException: For input string: "TTT"
Stack Trace:
java.lang.NumberFormatException: For input string: "TTT"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at com.packtpub.java7.concurrency.chapter1.recipe8.task.Task.run(Task.java
        at java.lang.Thread.run(Unknown Source)
Thread status: RUNNABLE
Thread has finished
```

(3)线程中对异常处理器的查找过程

首先，查找线程对象的未捕获异常处理器，如果找不到，JVM继续查找线程对象所在线程组的未捕获异常处理器，如果还是找不到，则继续查找默认的未捕获异常处理器（即java.lang.Thread.setDefaultUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler()）

10.线程局部变量的使用

如果创建的对象是实现了Runnable接口的类的实例，用它作为参数创建了多个线程，并启动这些线程,那么所有的线程将共享相同的属性，也就是说，如果一个线程改变了某个属性，所有的线程都会被这个改变影响。

在这种情况下，如果这个对象的属性不需要被所有的线程共享，则可以使用线程局部变量(Thread Local Variable）。

Demo1--不使用线程局部变量

```
public class UnsafeTask implements Runnable{

 @Override
 public void run() {
  startDate=new Date();
  System.out.printf("Starting Thread: %s : %s\n",Thread.currentThread().getId(),startDate);
  try {
   TimeUnit.SECONDS.sleep((int)Math.rint(Math.random()*10));
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  System.out.printf("Thread Finished: %s %s\n",Thread.currentThread().getId(),startDate);
 }
}
public class Main {

 public static void main(String[] args) {

  UnsafeTask task=new UnsafeTask();
  for (int i=0; i<3; i++){
   Thread thread=new Thread(task);
   thread.start();
   try {
    TimeUnit.SECONDS.sleep(2);
   } catch (InterruptedException e) {
    e.printStackTrace();
   }
  }
 }
}
```

结果

```
<terminated> Main (9) [Java Application] D:\Java\jre\bin\javaw.exe (2016年11月
Starting Thread: 9 : Sat Nov 19 19:52:17 CST 2016
Starting Thread: 10 : Sat Nov 19 19:52:19 CST 2016
Thread Finished: 10 : Sat Nov 19 19:52:19 CST 2016
Starting Thread: 11 : Sat Nov 19 19:52:21 CST 2016
Thread Finished: 9 : Sat Nov 19 19:52:21 CST 2016
Thread Finished: 11 : Sat Nov 19 19:52:21 CST 2016
```

结果分析：从结果可以看到线程9的结束时间和线程11的结束时间相同，这是因为当线程启动时设置了startDate的值，而此时线程9也调

用了startDate，因为变量被所有的线程共享，所以此线程结束的时间和线程11的开始时间一致。

Demo2--使用线程局部变量

```
public class SafeTask implements Runnable {

 private static ThreadLocal<Date> startDate= new ThreadLocal<Date>() {
  protected Date initialValue(){
   return new Date();
  }
 };
 @Override
 public void run() {
  // Writes the start date
  System.out.printf("Starting Thread: %s : %s\n",Thread.currentThread().getId(),startDate.get());
  try {
   TimeUnit.SECONDS.sleep((int)Math.rint(Math.random()*10));
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
  // Writes the start date
  System.out.printf("Thread Finished: %s : %s\n",Thread.currentThread().getId(),startDate.get());
 }
}

public class SafeMain {

 public static void main(String[] args) {

  SafeTask task=new SafeTask();
  for (int i=0; i<3; i++){
   Thread thread=new Thread(task);
   try {
    TimeUnit.SECONDS.sleep(1);
   } catch (InterruptedException e) {
    e.printStackTrace();
   }
   thread.start();
  }

 }
}
```

结果

```
<terminated> SafeMain [Java Application] D:\Java\jre\bin\javaw.exe (2016年11月19
Starting Thread: 9 : Sat Nov 19 20:02:07 CST 2016
Starting Thread: 10 : Sat Nov 19 20:02:08 CST 2016
Thread Finished: 9 : Sat Nov 19 20:02:07 CST 2016
Starting Thread: 11 : Sat Nov 19 20:02:09 CST 2016
Thread Finished: 11 : Sat Nov 19 20:02:09 CST 2016
Thread Finished: 10 : Sat Nov 19 20:02:08 CST 2016
```

结果分析：此Demo的startDate是线程局部变量，所以每个线程的开始和结束时间是一致的

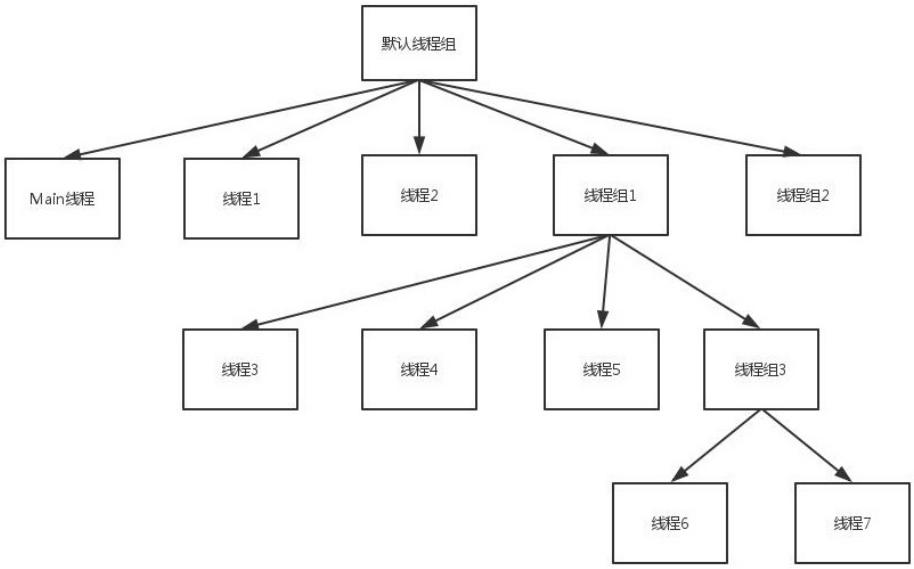
11.线程的分组

(1)Java中使用ThreadGroup来表示线程组，它可以对一批线程进行分类管理。对线程组的控管理，即同时控制线程组里面的这一批线程。

(2)用户创建的所有线程都属于指定线程组，如果没有显示指定属于哪个线程组，那么该线程就属于默认线程组（即main线程组）。默认情况下，子线程和父线程处于同一个线程组。

(3)只有在创建线程时才能指定其所在的线程组，线程运行中途不能改变它所属的线程组，也就是说线程一旦指定所在的线程组，就直到该线程结束。

(4)线程组与线程之间结构类似于树形的结构：




```
public class Result {

 private String name;
 public String getName() {
  return name;
 }
 public void setName(String name) {
  this.name = name;
 }
}
public class SearchTask implements Runnable {

 private Result result;
 public SearchTask(Result result) {
  this.result=result;
 }

 @Override
 public void run() {
  String name=Thread.currentThread().getName();
  System.out.printf("Thread %s: Start\n",name);
  try {
   doTask();
   result.setName(name);
  } catch (InterruptedException e) {
   System.out.printf("Thread %s: Interrupted\n",name);
   return;
  }
  System.out.printf("Thread %s: End\n",name);
 }

 private void doTask() throws InterruptedException {
  Random random=new Random((new Date()).getTime());
  int value=(int)(random.nextDouble()*100);
  System.out.printf("Thread %s: %d\n",Thread.currentThread().getName(),value);
  TimeUnit.SECONDS.sleep(value);
 }
}

public class Main {

 public static void main(String[] args) {

  ThreadGroup threadGroup = new ThreadGroup("Searcher");
  Result result=new Result();
```

```
 SearchTask searchTask=new SearchTask(result);
 for (int i=0; i<5; i++) {
  Thread thread=new Thread(threadGroup, searchTask); //创建线程组
  thread.start();
  try {
   TimeUnit.SECONDS.sleep(1);
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
 }

 System.out.printf("Number of Threads: %d\n",threadGroup.activeCount());
 System.out.printf("Information about the Thread Group\n");
 threadGroup.list();

 // Write information about the status of the Thread objects to the console
 Thread[] threads=new Thread[threadGroup.activeCount()];
 threadGroup.enumerate(threads);
 for (int i=0; i<threadGroup.activeCount(); i++) {
  System.out.printf("Thread %s: %s\n",threads[i].getName(),threads[i].getState());
 }
 waitFinish(threadGroup);
 threadGroup.interrupt();
}
private static void waitFinish(ThreadGroup threadGroup) {
 while (threadGroup.activeCount()>9) {
  try {
   TimeUnit.SECONDS.sleep(1);
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
 }
}
}
```

结果

```
<terminated> Main (10) [Java Application] D:\Java\jre\bin\javaw.exe (2016年11月19日
Thread Thread-0: Start
Thread Thread-0: 32
Thread Thread-1: Start
Thread Thread-1: 31
Thread Thread-2: Start
Thread Thread-2: 58
Thread Thread-3: Start
Thread Thread-3: 49
Thread Thread-4: Start
Thread Thread-4: 77
Number of Threads: 5
Information about the Thread Group
java.lang.ThreadGroup[name=Searcher,maxpri=10]
    Thread[Thread-0,5,Searcher]
    Thread[Thread-1,5,Searcher]
    Thread[Thread-2,5,Searcher]
    Thread[Thread-3,5,Searcher]
    Thread[Thread-4,5,Searcher]
Thread Thread-0: TIMED_WAITING
Thread Thread-1: TIMED_WAITING
Thread Thread-2: TIMED_WAITING
Thread Thread-3: TIMED_WAITING
Thread Thread-4: TIMED_WAITING
Thread Thread-1: Interrupted
Thread Thread-3: Interrupted
Thread Thread-4: Interrupted
Thread Thread-0: Interrupted
Thread Thread-2: Interrupted
```

12.线程组中不可控异常的处理

(1)线程中对异常处理器的查找过程

首先，查找线程对象的未捕获异常处理器，如果找不到，JVM继续查找线程对象所在线程组的未捕获异常处理器，如果还是找不到，则继续查找默认的未捕获异常处理器（即java.lang.Thread.setDefaultUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler()）

(2)线程组的未捕获异常处理器

继承ThreadGroup并覆盖UncaughtException

```
public class MyThreadGroup extends ThreadGroup {

 public MyThreadGroup(String name) {    //这一步是必须的
  super(name);
 }

 @Override
 public void uncaughtException(Thread t, Throwable e) {
  // Prints the name of the Thread
  System.out.printf("The thread %s has thrown an Exception\n",t.getId());
  // Print the stack trace of the exception
  e.printStackTrace(System.out);
  // Interrupt the rest of the threads of the thread group
  System.out.printf("Terminating the rest of the Threads\n");
  interrupt();
 }
}
public class Task implements Runnable {

 @Override
 public void run() {
  int result;
  // Create a random number generator
  Random random=new Random(Thread.currentThread().getId());
  while (true) {
   // Generate a random number a calculate 1000 divide by that random number
   result=1000/((int)(random.nextDouble()*1000));
   System.out.printf("%s : %f\n",Thread.currentThread().getId(),result);
   // Check if the Thread has been interrupted
   if (Thread.currentThread().isInterrupted()) {
    System.out.printf("%d : Interrupted\n",Thread.currentThread().getId());
    return;
   }
  }
 }
}
public class Main {

 public static void main(String[] args) {

  MyThreadGroup threadGroup=new MyThreadGroup("MyThreadGroup");
  Task task=new Task();
  for (int i=0; i<2; i++){
   Thread t=new Thread(threadGroup,task);
   t.start();
  }
 }
}
```

结果

```
<terminated> Main (11) [Java Application] D:\Java\jre\bin\javaw.exe (2016年11月19日 下午9:00:02)
9 : The thread 9 has thrown an Exception
java.util.IllegalFormatConversionException: f != java.lang.Integer
        at java.util.Formatter$FormatSpecifier.failConversion(Unknown Source)
        at java.util.Formatter$FormatSpecifier.printFloat(Unknown Source)
        at java.util.Formatter$FormatSpecifier.print(Unknown Source)
        at java.util.Formatter.format(Unknown Source)
        at java.io.PrintStream.format(Unknown Source)
        at java.io.PrintStream.printf(Unknown Source)
        at com.packtpub.java7.concurrency.chapter1.recipe11.task.Task.run(Task.jav
        at java.lang.Thread.run(Unknown Source)
Terminating the rest of the Threads
10 : The thread 10 has thrown an Exception
java.util.IllegalFormatConversionException: f != java.lang.Integer
        at java.util.Formatter$FormatSpecifier.failConversion(Unknown Source)
        at java.util.Formatter$FormatSpecifier.printFloat(Unknown Source)
        at java.util.Formatter$FormatSpecifier.print(Unknown Source)
        at java.util.Formatter.format(Unknown Source)
        at java.io.PrintStream.format(Unknown Source)
        at java.io.PrintStream.printf(Unknown Source)
        at com.packtpub.java7.concurrency.chapter1.recipe11.task.Task.run(Task.jav
        at java.lang.Thread.run(Unknown Source)
Terminating the rest of the Threads
```

13.使用工厂类创建线程

Java提供一个接口，ThreadFactory接口实现一个线程对象工厂

```
public class MyThreadFactory implements ThreadFactory {

 private int counter;
 private String name;
 private List<String> stats;

 public MyThreadFactory(String name){
  counter=0;
  this.name=name;
  stats=new ArrayList<String>();
 }

 @Override
 public Thread newThread(Runnable r) {
  // Create the new Thread object
  Thread t=new Thread(r,name+"-Thread_"+counter);
  counter++;
  // Actualize the statistics of the factory
  stats.add(String.format("Created thread %d with name %s on %s\n",t.getId(),t.getName(),new Date()));
  return t;
 }

 public String getStats(){
  StringBuffer buffer=new StringBuffer();
  Iterator<String> it=stats.iterator();

  while (it.hasNext()) {
   buffer.append(it.next());
  }

  return buffer.toString();
 }
}

public class Task implements Runnable {

 @Override
 public void run() {
  try {
   TimeUnit.SECONDS.sleep(1);
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
```

```
 }
}
public class Main {

 public static void main(String[] args) {

  MyThreadFactory factory=new MyThreadFactory("MyThreadFactory");
  Task task=new Task();
  Thread thread;

  System.out.printf("Starting the Threads\n");
  for (int i=0; i<10; i++){
   thread=factory.newThread(task);
   thread.start();
  }

  System.out.printf("Factory stats:\n");
  System.out.printf("%s\n",factory.getStats());
 }
}
```

结果

```
<terminated> Main (12) [Java Application] D:\Java\jre\bin\javaw.exe (2016年11月19日 下午9:19:46)
Starting the Threads
Factory stats:
Created thread 9 with name MyThreadFactory-Thread_0 on Sat Nov 19 21:19:46 CST 2016
Created thread 10 with name MyThreadFactory-Thread_1 on Sat Nov 19 21:19:46 CST 2016
Created thread 11 with name MyThreadFactory-Thread_2 on Sat Nov 19 21:19:46 CST 2016
Created thread 12 with name MyThreadFactory-Thread_3 on Sat Nov 19 21:19:46 CST 2016
Created thread 13 with name MyThreadFactory-Thread_4 on Sat Nov 19 21:19:46 CST 2016
Created thread 14 with name MyThreadFactory-Thread_5 on Sat Nov 19 21:19:46 CST 2016
Created thread 15 with name MyThreadFactory-Thread_6 on Sat Nov 19 21:19:46 CST 2016
Created thread 16 with name MyThreadFactory-Thread_7 on Sat Nov 19 21:19:46 CST 2016
Created thread 17 with name MyThreadFactory-Thread_8 on Sat Nov 19 21:19:46 CST 2016
Created thread 18 with name MyThreadFactory-Thread_9 on Sat Nov 19 21:19:46 CST 2016
```