并发编程网 – ifeve.com

让天下没有难学的技术

MAY 10 2015

Java8并发教程：Threads和Executors

BlankKelly

JAVA

并发译文

原文地址 原文作者：Benjamin Winterberg 译者：张坤

欢迎阅读我的Java8并发教程的第一部分。这份指南将会以简单易懂的代码示例来教给你如何在Java8中进行并发编程。这是一系列教程中的第一部分。在接下来的15分钟，你将会学会如何通过线程，任务（tasks）和 exector services来并行执行代码。

第一部分：Threads和Executors

第二部分：同步和锁

并发在Java5中首次被引入并在后续的版本中不断得到增强。在这篇文章中介绍的大部分概念同样适用于以前的Java版本。不过我的代码示例聚焦于Java8，大

RECENT POSTS

《TensorFlow官方文档》翻译邀请

《Spring官方文档》17.利用O/X映射器编组XML

《Java NIO教程》Java NIO Path

量使用lambda表达式和其他新特性。如果你对lambda表达式不属性，我推荐你首先阅读我的Java 8 教程。

Threads 和 Runnables

所有的现代操作系统都通过进程和线程来支持并发。进程是通常彼此独立运行的程序的实例，比如，如果你启动了一个Java程序，操作系统产生一个新的进程，与其他程序一起并行执行。在这些进程的内部，我们使用线程并发执行代码，因此，我们可以最大限度的利用CPU可用的核心（core）。

Java从JDK1.0开始执行线程。在开始一个新的线程之前，你必须指定由这个线程执行的代码，通常称为task。这可以通过实现Runnable——一个定义了一个无返回值无参数的run()方法的函数接口，如下面的代码所示：

```
Runnable task = () -> {
    String threadName = Thread.currentThread().getName()
    System.out.println("Hello " + threadName);
};

task.run();

Thread thread = new Thread(task);
thread.start();

System.out.println("Done!");
```

Velocity原理探究

子线程优雅调用父线程RequestScope作用域Bean问题的探究

《Maven官方指南》Maven 配置

高并发编程必备基础

Java中线程池ThreadPoolExecutor原理探究

并发队列-有界阻塞队列ArrayBlockingQueue原理探究

并发队列-无界非阻塞队列ConcurrentLinkedQueue原理探究

《Maven官方指南》模型指南

《Maven官方指南》构建Maven

《Maven官方指南》创建装配

并发队列-无界阻塞队列LinkedBlockingQueue原理探究

并发队列-无界阻塞优先级队列PriorityBlockingQueue原理探究

《Spring 5 官方文档》15.使用JDBC实现数据访问

《Maven官方指南》使用扩展

《Maven官方指南》Maven使用Ant指南

《Maven官方指南》配置代理

《Maven官方指南》生成源文件

《Maven官方指南》权限和发布设置

《Maven官方指南》配置档案插件指南

《Maven官方指南》APT格式

《Maven官方指南》片段宏指南

因为Runnable是一个函数接口，所以我们利用lambda表达式将当前的线程名打印到控制台。首先，在开始一个线程前我们在主线程中直接运行runnable。

控制台输出的结果可能像下面这样：

```
Hello main
Hello Thread-0
Done!
```

或者这样：

```
Hello main
Done!
Hello Thread-0
```

由于我们不能预测这个runnable是在打印'done'前执行还是在之后执行。顺序是不确定的，因此在大的程序中编写并发程序是一个复杂的任务。

我们可以将线程休眠确定的时间。在这篇文章接下来的代码示例中我们可以通过这种方法来模拟长时间运行的任务。

```
Runnable runnable = () -> {
    try {
        String name = Thread.currentThread().getName();
```

Java锁是如何保证数据可见性的

《Maven官方指南》创建一个站点

《Spring官方文档1》17 使用 O/X(Object/XML)映射器对XML进行编组

《Spring 5 官方文档》20. CORS 支持

《Spring 5官方文档》-JMX

《Kafka官方文档》实现

热门文章

Google Guava官方教程（中文版） 449,242 人阅读

Java NIO系列教程（一） Java NIO 概述 288,283 人阅读

Java并发性和多线程介绍目录 194,788 人阅读

Java NIO 系列教程 177,857 人阅读

Java NIO系列教程（十二） Java NIO与IO 159,162 人阅读

Java NIO系列教程（六） Selector 149,633 人阅读

《Storm入门》中文版 145,462 人阅读

Java NIO系列教程（三） Buffer 142,947 人阅读

Java NIO系列教程（二） Channel 142,455 人阅读

Java8初体验（二）Stream语法详解 133,677 人阅读

69道Spring面试题和答案 128,368 人阅读

Netty 5用户指南 117,964 人阅读

```
            System.out.println("Foo " + name);
            TimeUnit.SECONDS.sleep(1);
            System.out.println("Bar " + name);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    };

    Thread thread = new Thread(runnable);
    thread.start();
```

当你运行上面的代码时，你会注意到在第一条打印语句和第二条打印语句之间存在一分钟的延迟。`TimeUnit`在处理单位时间时一个有用的枚举类。你可以通过调用`Thread.sleep(1000)`来达到同样的目的。

使用`Thread`类是很单调的且容易出错。由于并发API在2004年Java5发布的时候才被引入。这些API位于`java.util.concurrent`包下，包含很多处理并发编程的有用的类。自从这些并发API引入以来，在随后的新的Java版本发布过程中得到不断的增强，甚至Java8提供了新的类和方法来处理并发。

接下来，让我们走进并发API中最重要的一部——executor services。

Executors

并发API引入了ExecutorService作为一个在程序中直接使用Thread的高层次的替换方案。Executos支持运行异步任务，通常管理一个线程池，这样一来我们就不需要手动去创建新的线程。在不断地处理任务的过程中，线程池内部线程将会得到复用，因此，在我们可以使用一个executor service来运行和我们想在我们整个程序中执行的一样多的并发任务。

下面是使用executors的第一个代码示例：

```
ExecutorService executor = Executors.newSingleThreadExec
executor.submit(() -> {
String threadName = Thread.currentThread().getName();
System.out.println("Hello " + threadName);
```

});

```
// => Hello pool-1-thread-1
```

Executors类提供了便利的工厂方法来创建不同类型的 executor services。在这个示例中我们使用了一个单线程线程池的 executor。

代码运行的结果类似于上一个示例，但是当运行代码时，你会注意到一个很大的差别：Java进程从没有停止！Executors必须显式的停止-否则它们将持续监听新的任务。

ExecutorService提供了两个方法来达到这个目的——shutdwon()会等待正在执行的任务执行完而shutdownNow()会终止所有正在执行的任务并立即关闭execuotr。

这是我喜欢的通常关闭executors的方式：

```
try {
    System.out.println("attempt to shutdown executor");
    executor.shutdown();
    executor.awaitTermination(5, TimeUnit.SECONDS);
    }
catch (InterruptedException e) {
    System.err.println("tasks interrupted");
}
finally {
    if (!executor.isTerminated()) {
        System.err.println("cancel non-finished tasks");
    }
    executor.shutdownNow();
    System.out.println("shutdown finished");
}
```

executor通过等待指定的时间让当前执行的任务终止来“温柔的”关闭executor。在等待最长5分钟的时间后，execuote最终会通过中断所有的正在执行的任务关

synchronization Synchronized thread tomcat volatile 多线程 并发译文，Java，Maven

ARCHIVES

June 2017 (31)
May 2017 (59)
April 2017 (17)
March 2017 (30)
February 2017 (9)
January 2017 (7)
December 2016 (12)
November 2016 (27)
October 2016 (16)
September 2016 (11)
August 2016 (6)
July 2016 (9)
June 2016 (7)
May 2016 (20)
April 2016 (28)
March 2016 (8)
February 2016 (7)

友情链接

coolshell
Programer. 大猫
一粟的博客
志俊的博客
最代码
点点折
领悟书生

CATEGORIES

akka (20)
Android (3)
C++ (11)
CPU (2)
Framework (52)
GO (6)
groovy (6)

闭。

Callables 和 Futures

除了Runnable，executor还支持另一种类型的任务——Callable。Callables也是类似于runnables的函数接口，不同之处在于，Callable返回一个值。

下面的lambda表达式定义了一个callable：在休眠一分钟后返回一个整数。

```
Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupte
    }
};
```

Callbale也可以像runnbales一样提交给 executor services。但是callables的结果怎么办？因为submit()不会等待任务完成，executor service不能直接返回callable的结果。不过，executor 可以返回一个Future类型的结果，它可以用来在稍后某个时间取出实际的结果。

```
ExecutorService executor = Executors.newFixedThreadPool(
Future<Integer> future = executor.submit(task);

System.out.println("future done? " + future.isDone());

Integer result = future.get();

System.out.println("future done? " + future.isDone());
System.out.print("result: " + result);
```

在将callable提交给exector之后，我们先通过调用`isDone()`来检查这个future是否已经完成执行。我十分确定这会发生什么，因为在返回那个整数之前callable会休眠一分钟、

在调用`get()`方法时，当前线程会阻塞等待，直到callable在返回实际的结果123之前执行完成。现在future执行完毕，我们可以在控制台看到如下的结果：

```
future done? false
future done? true
result: 123
```

Future与底层的executor service紧密的结合在一起。记住，如果你关闭executor，所有的未中止的future都会抛出异常。

February 2014 (38)
January 2014 (19)
December 2013 (10)
November 2013 (4)
October 2013 (20)
September 2013 (38)
August 2013 (48)
July 2013 (26)
June 2013 (16)
May 2013 (9)
April 2013 (17)
March 2013 (41)
February 2013 (25)
January 2013 (57)
December 2012 (9)
October 2012 (1)
August 2012 (1)

本站原创 (82)
架构 (27)
活动 (6)
网络 (6)

```
executor.shutdownNow();
future.get();
```

你可能注意到我们这次创建executor的方式与上一个例子稍有不同。我们使用`newFixedThreadPool(1)`来创建一个单线程线程池的 execuot service。 这等同于使用`newSingleThreadExecutor`不过使用第二种方式我们可以稍后通过简单的传入一个比1大的值来增加线程池的大小。

Timeouts

任何`future.get()`调用都会阻塞，然后等待直到callable中止。在最糟糕的情况下，一个callable持续运行——因此使你的程序将没有响应。我们可以简单的传入一个时长来避免这种情况。

```
ExecutorService executor = Executors.newFixedThreadPool(

    Future<Integer> future = executor.submit(() -> {
    try {
        TimeUnit.SECONDS.sleep(2);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupte
    }
});
```

```
    future.get(1, TimeUnit.SECONDS);
```

运行上面的代码将会产生一个TimeoutException：

```
Exception in thread "main" java.util.concurrent.TimeoutE
    at java.util.concurrent.FutureTask.get(FutureTask.ja
```

你可能已经猜到为什么会抛出这个异常。我们指定的最长等待时间为1分钟，而这个callable在返回结果之前实际需要两分钟。

invokeAll

Executors支持通过invokeAll()一次批量提交多个callable。这个方法结果一个callable的集合，然后返回一个future的列表。

```
ExecutorService executor = Executors.newWorkStealingPool

List<Callable<String>> callables = Arrays.asList(
        () -> "task1",
        () -> "task2",
        () -> "task3");

executor.invokeAll(callables)
    .stream()
    .map(future -> {
```

```
        try {
            return future.get();
        }
        catch (Exception e) {
            throw new IllegalStateException(e);
        }
    })
    .forEach(System.out::println);
```

在这个例子中，我们利用Java8中的函数流（stream）来处理`invokeAll()`调用返回的所有future。我们首先将每一个future映射到它的返回值，然后将每个值打印到控制台。如果你还不属性stream，可以阅读我的Java8 Stream 教程。

invokeAny

批量提交callable的另一种方式就是`invokeAny()`，它的工作方式与`invokeAll()`稍有不同。在等待future对象的过程中，这个方法将会阻塞直到第一个callable中止然后返回这一个callable的结果。

为了测试这种行为，我们利用这个帮助方法来模拟不同执行时间的callable。这个方法返回一个callable，这个callable休眠指定 的时间直到返回给定的结果。

```
Callable<String> callable(String result, long sleepSecon
    return () -> {
        TimeUnit.SECONDS.sleep(sleepSeconds);
```

```
            return result;
        };
    }
```

我们利用这个方法创建一组callable，这些callable拥有不同的执行时间，从1分钟到3分钟。通过`invokeAny()`将这些callable提交给一个executor，返回最快的callable的字符串结果-在这个例子中为任务2：

```
ExecutorService executor = Executors.newWorkStealingPool

List<Callable<String>> callables = Arrays.asList(
callable("task1", 2),
callable("task2", 1),
callable("task3", 3));

String result = executor.invokeAny(callables);
System.out.println(result);

// => task2
```

上面这个例子又使用了另一种方式来创建executor——调用`newWorkStealingPool()`。这个工厂方法是Java8引入的，返回一个

ForkJoinPool类型的 executor，它的工作方法与其他常见的execuotr稍有不同。与使用一个固定大小的线程池不同，ForkJoinPools使用一个并行因子数来创建，默认值为主机CPU的可用核心数。

ForkJoinPools 在Java7时引入，将会在这个系列后面的教程中详细讲解。让我们深入了解一下 scheduled executors 来结束本次教程。

Scheduled Executors

我们已经学习了如何在一个 executor 中提交和运行一次任务。为了持续的多次执行常见的任务，我们可以利用调度线程池。

ScheduledExecutorService支持任务调度，持续执行或者延迟一段时间后执行。

下面的实例，调度一个任务在延迟3分钟后执行：

```
ScheduledExecutorService executor = Executors.newSchedul

Runnable task = () -> System.out.println("Scheduling: "
ScheduledFuture<?> future = executor.schedule(task, 3, T

TimeUnit.MILLISECONDS.sleep(1337);

long remainingDelay = future.getDelay(TimeUnit.MILLISECO
System.out.printf("Remaining Delay: %sms", remainingDela
```

调度一个任务将会产生一个专门的future类型——ScheduleFuture，它除了提供了Future的所有方法之外，他还提供了getDelay()方法来获得剩余的延迟。在延迟消逝后，任务将会并发执行。

为了调度任务持续的执行，executors 提供了两个方法scheduleAtFixedRate()和scheduleWithFixedDelay()。第一个方法用来以固定频率来执行一个任务，比如，下面这个示例中，每分钟一次：

```
ScheduledExecutorService executor =     Executors.newSch

Runnable task = () -> System.out.println("Scheduling: "

int initialDelay = 0;
int period = 1;
executor.scheduleAtFixedRate(task, initialDelay, period,
```

另外，这个方法还接收一个初始化延迟，用来指定这个任务首次被执行等待的时长。

请记住：scheduleAtFixedRate()并不考虑任务的实际用时。所以，如果你指定了一个period为1分钟而任务需要执行2分钟，那么线程池为了性能会更快的执行。

在这种情况下，你应该考虑使用scheduleWithFixedDelay()。这个方法的工作方式与上我们上面描述的类似。不同之处在于等待时间 period 的应用是在一次任务的结束和下一个任务的开始之间。例如：

```
ScheduledExecutorService executor =         Executors.ne

Runnable task = () -> {
    try {
        TimeUnit.SECONDS.sleep(2);
        System.out.println("Scheduling: " + System.nanoT
    }
    catch (InterruptedException e) {
        System.err.println("task interrupted");
    }
};

executor.scheduleWithFixedDelay(task, 0, 1, TimeUnit.SEC
```

这个例子调度了一个任务，并在一次执行的结束和下一次执行的开始之间设置了一个1分钟的固定延迟。初始化延迟为0，任务执行时间为0。所以我们分别在0s,3s,6s,9s等间隔处结束一次执行。如你所见，scheduleWithFixedDelay()在你不能预测调度任务的执行时长时是很有用的。

这是并发系列教程的第以部分。我推荐你亲手实践一下上面的代码示例。你可以从 Github 上找到这篇文章中所有的代码示例，所以欢迎你fork这个repo，给我星星。

我希望你会喜欢这篇文章。如果你有任何的问题都可以在下面评论或者通过 Twitter 给我回复。

原创文章，转载请注明： *转载自并发编程网 – ifeve.com本文链接地址: Java8并发教程：Threads和Executors*

9

About Latest Posts

BlankKelly

实习中...

⭐添加本文到我的收藏

Related Posts:

1. **线程执行者（六）运行多个任务并处理所有结果**

2. 定制并发类（三）实现一个基于优先级的Executor类
3. 定制并发类（二）定制ThreadPoolExecutor类
4. 线程执行者（九）执行者取消一个任务
5. 测试并发应用（三）监控Executor框架
6. Callable和Future
7. 线程执行者（十一）执行者分离任务的启动和结果的处理
8. 线程执行者（十）执行者控制一个任务完成
9. ExecutorService-10个要诀和技巧
10. 定制并发类（六）自定义在计划的线程池内运行的任务
11. 线程执行者（四）执行者执行返回结果的任务
12. CompletableFuture 不能被中断
13. 线程管理（十）线程组
14. 线程执行者（八）执行者周期性地运行一个任务
15. 线程执行者（十二）执行者控制被拒绝的任务

Write comment | Comments RSS | Trackback are closed | Comments (3)

Stefanie
05/11. 2015 11:30am

Log in to Reply | QUOTE

不错，比较基础是嘎嘎嘎嘎 嘎嘎嘎啊啊噶噶嘎嘎

kenny

05/17. 2015 1:40pm

Log in to Reply | QUOTE

second全部翻译成了分钟，至少在本地测试一下是不是分钟吧。

felayman

05/18. 2017 4:12pm

Log in to Reply | QUOTE

second全部翻译成了分钟

You must be logged in to post a comment.

Java8中CAS的增强 Guava官方文档-RateLimiter类