[Spring中线程池的应用](http://blog.csdn.net/shimiso/article/details/8964527)

2013-05-23 11:54 16557人阅读 [评论](http://blog.csdn.net/shimiso/article/details/8964527#comments)(3) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/shimiso/article/details/8964527#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

j2ee相关（66） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg java基础（29） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

多线程并发处理起来通常比较麻烦，如果你使用spring容器来管理业务bean，事情就好办了多了。spring封装了java的多线程的实现，你只需要关注于并发事物的流程以及一些并发负载量等特性，具体来说如何使用spring来处理并发事务：

1.了解 TaskExecutor接口

Spring的TaskExecutor接口等同于java.util.concurrent.Executor接口。 实际上，它存在的主要原因是为了在使用线程池的时候，将对Java 5的依赖抽象出来。 这个接口只有一个方法execute(Runnable task)，它根据线程池的语义和配置，来接受一个执行任务。最初创建TaskExecutor是为了在需要时给其他Spring组件提供一个线程池的抽象。 例如ApplicationEventMulticaster组件、JMS的 AbstractMessageListenerContainer和对Quartz的整合都使用了TaskExecutor抽象来提供线程池。 当然，如果你的bean需要线程池行为，你也可以使用这个抽象层。

2. TaskExecutor接口的实现类

(1)SimpleAsyncTaskExecutor 类

这个实现不重用任何线程，或者说它每次调用都启动一个新线程。但是，它还是支持对并发总数设限，当超过线程并发总数限制时，阻塞新的调用，直到有位置被释放。如果你需要真正的池，请继续往下看。

(2)SyncTaskExecutor类

这个实现不会异步执行。相反，每次调用都在发起调用的线程中执行。它的主要用处是在不需要多线程的时候，比如简单的test case。

(3)ConcurrentTaskExecutor 类

这个实现是对Java 5 java.util.concurrent.Executor类的包装。有另一个备选, ThreadPoolTaskExecutor类，它暴露了Executor的配置参数作为bean属性。很少需要使用ConcurrentTaskExecutor, 但是如果ThreadPoolTaskExecutor不敷所需，ConcurrentTaskExecutor是另外一个备选。

(4)SimpleThreadPoolTaskExecutor 类

这个实现实际上是Quartz的SimpleThreadPool类的子类，它会监听Spring的生命周期回调。当你有线程池，需要在Quartz和非Quartz组件中共用时，这是它的典型用处。

(5)ThreadPoolTaskExecutor 类

它不支持任何对java.util.concurrent包的替换或者下行移植。Doug Lea和Dawid Kurzyniec对java.util.concurrent的实现都采用了不同的包结构，导致它们无法正确运行。 这个实现只能在Java 5环境中使用，但是却是这个环境中最常用的。它暴露的bean properties可以用来配置一个java.util.concurrent.ThreadPoolExecutor，把它包装到一个TaskExecutor中。如果你需要更加先进的类，比如ScheduledThreadPoolExecutor,我们建议你使用ConcurrentTaskExecutor来替代。

(6)TimerTaskExecutor类

这个实现使用一个TimerTask作为其背后的实现。它和SyncTaskExecutor的不同在于，方法调用是在一个独立的线程中进行的，虽然在那个线程中是同步的。

(7)WorkManagerTaskExecutor类

这个实现使用了CommonJ WorkManager作为其底层实现，是在Spring context中配置CommonJ WorkManager应用的最重要的类。和SimpleThreadPoolTaskExecutor类似，这个类实现了WorkManager接口，因此可以直接作为WorkManager使用。

3.线程池Demo之 ThreadPoolTaskExecutor

(1)编写测试类

**[java]** [view plain](http://blog.csdn.net/shimiso/article/details/8964527) [copy](http://blog.csdn.net/shimiso/article/details/8964527)

1. **import** org.springframework.core.task.TaskExecutor;
3. **public** **class** MainExecutor {
4. **private** TaskExecutor taskExecutor;
5. **public** MainExecutor (TaskExecutor taskExecutor) {
6. **this**.taskExecutor = taskExecutor;
7. }
8. **public** **void** printMessages() {
9. **for**(**int** i = 0; i < 25; i++) {
10. taskExecutor.execute(**new** MessagePrinterTask("Message" + i));
11. }
12. }

15. **private** **class** MessagePrinterTask **implements** Runnable {
16. **private** String message;
17. **public** MessagePrinterTask(String message) {
18. **this**.message = message;
19. }
20. **public** **void** run() {
21. System.out.println(message);
22. }
23. }
24. }

在业务代码中，通常以for循环的方式执行多个事务  
for(int k = 0; k < n; k++) {      
        taskExecutor.execute(new ThreadTransCode());      
    }  
其它繁琐的线程管理的事情就交给执行器去管理。  
值得注意的事有两点  
1, taskExecutor.execute(new ThreadTransCode());  激活的线程都是守护线程，主线程结束，守护线程就会放弃执行，这个在业务中式符合逻辑的，在单元测试中为了看到执行效果，需要自行阻塞主线程。  
2, taskExecutor.execute(new ThreadTransCode());    的执行也不是完全安全的，在执行的过程中可能会因为需要的线程查过了线程队列的容量而抛出运行时异常，如有必要需要捕获。

(2)spring的配置

**[html]** [view plain](http://blog.csdn.net/shimiso/article/details/8964527) [copy](http://blog.csdn.net/shimiso/article/details/8964527)

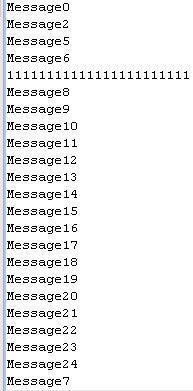
1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "/spring-beans.dtd"**>**
3. **<beans>**
5. <!-- 异步线程池 -->
6. **<bean** id="threadPool" class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor"**>**
7. <!-- 核心线程数  -->
8. **<property** name="corePoolSize" value="10" **/>**
9. <!-- 最大线程数 -->
10. **<property** name="maxPoolSize" value="50" **/>**
11. <!-- 队列最大长度 >=mainExecutor.maxSize -->
12. **<property** name="queueCapacity" value="1000" **/>**
13. <!-- 线程池维护线程所允许的空闲时间 -->
14. **<property** name="keepAliveSeconds" value="300" **/>**
15. <!-- 线程池对拒绝任务(无线程可用)的处理策略 -->
16. **<property** name="rejectedExecutionHandler"**>**
17. **<bean** class="java.util.concurrent.ThreadPoolExecutor$CallerRunsPolicy" **/>**
18. **</property>**
19. **</bean>**
21. **<bean** id="mainExecutor" class="supben.MainExecutor"**>**
22. **<property** name="threadPool" ref="threadPool" **/>**
23. **</bean>**
25. **<bean** id="springScheduleExecutorTask" class="org.springframework.scheduling.concurrent.ScheduledExecutorTask"**>**
26. **<property** name="runnable" ref="mainExecutor" **/>**
27. <!-- 容器加载10秒后开始执行 -->
28. **<property** name="delay" value="10000" **/>**
29. <!-- 每次任务间隔 5秒-->
30. **<property** name="period" value="5000" **/>**
31. **</bean>**
33. **<bean** id="springScheduledExecutorFactoryBean" class="org.springframework.scheduling.concurrent.ScheduledExecutorFactoryBean"**>**
34. **<property** name="scheduledExecutorTasks"**>**
35. **<list>**
36. **<ref** bean="springScheduleExecutorTask" **/>**
37. **</list>**
38. **</property>**
39. **</bean>**
41. **</beans>**

(3)调用

**[java]** [view plain](http://blog.csdn.net/shimiso/article/details/8964527) [copy](http://blog.csdn.net/shimiso/article/details/8964527)

1. ApplicationContext appContext = **new** ClassPathXmlApplicationContext("applicationContext.xml");
2. MainExecutor te = (MainExecutor)appContext.getBean("taskExecutorExample");
3. te.printMessages();
4. System.out.println("11111111111111111111111");

(4)效果



[[置顶] SPRING中的线程池ThreadPoolTaskExecutor](http://blog.csdn.net/qinpeng100423/article/details/9307181)

标签： [Spring](http://www.csdn.net/tag/Spring)[线程池](http://www.csdn.net/tag/%e7%ba%bf%e7%a8%8b%e6%b1%a0)[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)

2013-07-12 10:36 44175人阅读 [评论](http://blog.csdn.net/qinpeng100423/article/details/9307181#comments)(11) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/qinpeng100423/article/details/9307181#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

JAVA（19） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg Spring（7） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

一、初始化

1，直接调用

**[java]** [view plain](http://blog.csdn.net/qinpeng100423/article/details/9307181) [copy](http://blog.csdn.net/qinpeng100423/article/details/9307181)

[print](http://blog.csdn.net/qinpeng100423/article/details/9307181)[?](http://blog.csdn.net/qinpeng100423/article/details/9307181)

1. ThreadPoolTaskExecutor poolTaskExecutor = **new** ThreadPoolTaskExecutor();
2. //线程池所使用的缓冲队列
3. poolTaskExecutor.setQueueCapacity(200);
4. //线程池维护线程的最少数量
5. poolTaskExecutor.setCorePoolSize(5);
6. //线程池维护线程的最大数量
7. poolTaskExecutor.setMaxPoolSize(1000);
8. //线程池维护线程所允许的空闲时间
9. poolTaskExecutor.setKeepAliveSeconds(30000);
10. poolTaskExecutor.initialize();

2、配置文件

**[plain]** [view plain](http://blog.csdn.net/qinpeng100423/article/details/9307181) [copy](http://blog.csdn.net/qinpeng100423/article/details/9307181)

[print](http://blog.csdn.net/qinpeng100423/article/details/9307181)[?](http://blog.csdn.net/qinpeng100423/article/details/9307181)

1. <!-- 配置线程池 -->
2. <bean id ="taskExecutor"  class ="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor" >
3. <!-- 线程池维护线程的最少数量 -->
4. <span style="white-space:pre">  </span><property name ="corePoolSize" value ="5" />
5. <!-- 线程池维护线程所允许的空闲时间 -->
6. <span style="white-space:pre">  </span><property name ="keepAliveSeconds" value ="30000" />
7. <!-- 线程池维护线程的最大数量 -->
8. <span style="white-space:pre">  </span><property name ="maxPoolSize" value ="1000" />
9. <!-- 线程池所使用的缓冲队列 -->
10. <span style="white-space:pre">  </span><property name ="queueCapacity" value ="200" />
11. </bean>

程序里面获取：

ApplicationContext ctx =  new ClassPathXmlApplicationContext("applicationContext.xml");  
ThreadPoolTaskExecutor poolTaskExecutor = (ThreadPoolTaskExecutor)ctx.getBean("taskExecutor");  
  
  
二、利用线程池启动线程  
Thread udpThread = new Thread(udp);  
poolTaskExecutor.execute(udpThread);  
获取当前线程池活动的线程数：  
int count = poolTaskExecutor.getActiveCount();  
logger.debug("[x] - now threadpool active threads totalNum : " +count);  
  
  
  
  
三、配置解释  
当一个任务通过execute(Runnable)方法欲添加到线程池时：  
1、 如果此时线程池中的数量小于corePoolSize，即使线程池中的线程都处于空闲状态，也要创建新的线程来处理被添加的任务。  
2、 如果此时线程池中的数量等于 corePoolSize，但是缓冲队列 workQueue未满，那么任务被放入缓冲队列。  
3、如果此时线程池中的数量大于corePoolSize，缓冲队列workQueue满，并且线程池中的数量小于maximumPoolSize，建新的线程来处理被添加的任务。  
4、 如果此时线程池中的数量大于corePoolSize，缓冲队列workQueue满，并且线程池中的数量等于maximumPoolSize，那么通过 handler所指定的策略来处理此任务。也就是：处理任务的优先级为：核心线程corePoolSize、任务队列workQueue、最大线程 maximumPoolSize，如果三者都满了，使用handler处理被拒绝的任务。  
5、 当线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止。这样，线程池可以动态的调整池中的线程数。  
  
  
四、其它线程池  
JDK的ThreadPoolExecutor  
一个 ExecutorService，它使用可能的几个池线程之一执行每个提交的任务，通常使用Executors 工厂方法配置。  
线程池可以解决两个不同问题：由于减少了每个任务调用的开销，它们通常可以在执行大量异步任务时提供增强的性能，并且还可以提供绑定和管理资源（包括执行集合任务时使用的线程）的方法。每个ThreadPoolExecutor 还维护着一些基本的统计数据，如完成的任务数。  
为了便于跨大量上下文使用，此类提供了很多可调整的参数和扩展挂钩。但是，强烈建议程序员使用较为方便的 Executors 工厂方法 Executors.newCachedThreadPool()（无界线程池，可以进行自动线程回收）、Executors.newFixedThreadPool(int)（固定大小线程池）和Executors.newSingleThreadExecutor()（单个后台线程），它们均为大多数使用场景预定义了设置。否则，在手动配置和调整此类时，使用以下指导：  
1、核心和最大池大小   
ThreadPoolExecutor 将根据 corePoolSize（参见 getCorePoolSize()）和 maximumPoolSize（参见getMaximumPoolSize()）设置的边界自动调整池大小。当新任务在方法execute(java.lang.Runnable) 中提交时，如果运行的线程少于 corePoolSize，则创建新线程来处理请求，即使其他辅助线程是空闲的。如果运行的线程多于 corePoolSize 而少于 maximumPoolSize，则仅当队列满时才创建新线程。如果设置的 corePoolSize 和 maximumPoolSize 相同，则创建了固定大小的线程池。如果将 maximumPoolSize 设置为基本的无界值（如Integer.MAX\_VALUE），则允许池适应任意数量的并发任务。在大多数情况下，核心和最大池大小仅基于构造来设置，不过也可以使用setCorePoolSize(int) 和 setMaximumPoolSize(int) 进行动态更改。  
2、按需构造   
默认情况下，即使核心线程最初只是在新任务需要时才创建和启动的，也可以使用方法 prestartCoreThread() 或prestartAllCoreThreads() 对其进行动态重写。  
3、创建新线程   
使用 ThreadFactory 创建新线程。如果没有另外说明，则在同一个ThreadGroup 中一律使用Executors.defaultThreadFactory() 创建线程，并且这些线程具有相同的NORM\_PRIORITY 优先级和非守护进程状态。通过提供不同的 ThreadFactory，可以改变线程的名称、线程组、优先级、守护进程状态，等等。如果从newThread 返回 null 时ThreadFactory 未能创建线程，则执行程序将继续运行，但不能执行任何任务。  
4、保持活动时间   
如果池中当前有多于 corePoolSize 的线程，则这些多出的线程在空闲时间超过 keepAliveTime 时将会终止（参见 getKeepAliveTime(java.util.concurrent.TimeUnit)）。这提供了当池处于非活动状态时减少资源消耗的方法。如果池后来变得更为活动，则可以创建新的线程。也可以使用方法setKeepAliveTime(long, java.util.concurrent.TimeUnit) 动态地更改此参数。使用Long.MAX\_VALUETimeUnit.NANOSECONDS 的值在关闭前有效地从以前的终止状态禁用空闲线程。  
5、排队   
所有 BlockingQueue 都可用于传输和保持提交的任务。可以使用此队列与池大小进行交互：  
如果运行的线程少于 corePoolSize，则 Executor 始终首选添加新的线程，而不进行排队。   
如果运行的线程等于或多于 corePoolSize，则 Executor 始终首选将请求加入队列，而不添加新的线程。   
如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 maximumPoolSize，在这种情况下，任务将被拒绝。   
排队有三种通用策略：  
a、直接提交。工作队列的默认选项是 SynchronousQueue，它将任务直接提交给线程而不保持它们。在此，如果不存在可用于立即运行任务的线程，则试图把任务加入队列将失败，因此会构造一个新的线程。此策略可以避免在处理可能具有内部依赖性的请求集合时出现锁定。直接提交通常要求无界 maximumPoolSizes 以避免拒绝新提交的任务。当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。  
b、无界队列。使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）将导致在所有 corePoolSize 线程都忙的情况下将新任务加入队列。这样，创建的线程就不会超过 corePoolSize。（因此，maximumPoolSize 的值也就无效了。）当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列；例如，在 Web 页服务器中。这种排队可用于处理瞬态突发请求，当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。  
c、有界队列。当使用有限的 maximumPoolSizes 时，有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制。队列大小和最大池大小可能需要相互折衷：使用大型队列和小型池可以最大限度地降低 CPU 使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量。如果任务频繁阻塞（例如，如果它们是 I/O 边界），则系统可能为超过您许可的更多线程安排时间。使用小型队列通常要求较大的池大小，CPU 使用率较高，但是可能遇到不可接受的调度开销，这样也会降低吞吐量。  
6、被拒绝的任务   
当 Executor 已经关闭，并且 Executor 将有限边界用于最大线程和工作队列容量，且已经饱和时，在方法 execute(java.lang.Runnable) 中提交的新任务将被拒绝。在以上两种情况下，execute 方法都将调用其RejectedExecutionHandler 的RejectedExecutionHandler.rejectedExecution(java.lang.Runnable, java.util.concurrent.ThreadPoolExecutor) 方法。下面提供了四种预定义的处理程序策略：  
在默认的 ThreadPoolExecutor.AbortPolicy 中，处理程序遭到拒绝将抛出运行时RejectedExecutionException。  
在 ThreadPoolExecutor.CallerRunsPolicy 中，线程调用运行该任务的execute 本身。此策略提供简单的反馈控制机制，能够减缓新任务的提交速度。  
在 ThreadPoolExecutor.DiscardPolicy 中，不能执行的任务将被删除。  
在 ThreadPoolExecutor.DiscardOldestPolicy 中，如果执行程序尚未关闭，则位于工作队列头部的任务将被删除，然后重试执行程序（如果再次失败，则重复此过程）。  
定义和使用其他种类的 RejectedExecutionHandler 类也是可能的，但这样做需要非常小心，尤其是当策略仅用于特定容量或排队策略时。  
7、挂钩方法   
此类提供 protected 可重写的 beforeExecute(java.lang.Thread, java.lang.Runnable) 和afterExecute(java.lang.Runnable, java.lang.Throwable) 方法，这两种方法分别在执行每个任务之前和之后调用。它们可用于操纵执行环境；例如，重新初始化 ThreadLocal、搜集统计信息或添加日志条目。此外，还可以重写方法terminated() 来执行 Executor 完全终止后需要完成的所有特殊处理。  
如果挂钩或回调方法抛出异常，则内部辅助线程将依次失败并突然终止。   
8、队列维护   
方法 getQueue() 允许出于监控和调试目的而访问工作队列。强烈反对出于其他任何目的而使用此方法。remove(java.lang.Runnable) 和purge() 这两种方法可用于在取消大量已排队任务时帮助进行存储回收。

# [Spring线程池开发实战](http://blog.csdn.net/chszs/article/details/8219189)

标签： [java](http://www.csdn.net/tag/java)[Java](http://www.csdn.net/tag/Java)[JAVA](http://www.csdn.net/tag/JAVA)[spring](http://www.csdn.net/tag/spring)[Spring](http://www.csdn.net/tag/Spring)[thread](http://www.csdn.net/tag/thread)[Thread](http://www.csdn.net/tag/Thread)[线程](http://www.csdn.net/tag/%e7%ba%bf%e7%a8%8b)[线程池](http://www.csdn.net/tag/%e7%ba%bf%e7%a8%8b%e6%b1%a0)

2012-11-24 10:46 17473人阅读 [评论](http://blog.csdn.net/chszs/article/details/8219189#comments)(2) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/chszs/article/details/8219189#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

JavaEE开发（121） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg Java&算法（86） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg 并行&多线程（1） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主chszs的原创文章，未经博主允许不得转载。

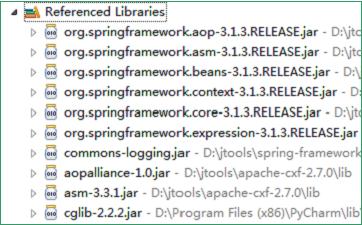
目录[(?)[+]](http://blog.csdn.net/chszs/article/details/8219189)

# ****Spring线程池开发实战****

**作者：chszs，转载需注明。**

**作者博客主页：http://blog.csdn.net/chszs**  
  
本文提供了三个Spring多线程开发的例子，由浅入深，由于例子一目了然，所以并未做过多的解释。诸位一看便知。

## 前提条件：

1）在Eclipse创建一个Java项目，我取名为SpringThreadDemo。  
2）项目所需的JAR包如图所示：  
 

下面开始。

**注：项目源码已经托管到GitHub，地址：https://github.com/chszs/SpringThreadDemo**

## 例子1：Spring结合Java线程。

通过继承Thread创建一个简单的Java线程，然后使用@Component让Spring容器管理此线程，Bean的范围必须是prototype，因此每个请求都会返回一个新实例，运行每个单独的线程。

### PrintThread.java

**[java]** [view plain](http://blog.csdn.net/chszs/article/details/8219189) [copy](http://blog.csdn.net/chszs/article/details/8219189)

[print](http://blog.csdn.net/chszs/article/details/8219189)[?](http://blog.csdn.net/chszs/article/details/8219189)

1. **package** com.chszs.thread;
3. **import** org.springframework.stereotype.Component;
4. **import** org.springframework.context.annotation.Scope;
6. @Component
7. @Scope("prototype")
8. **public** **class** PrintThread **extends** Thread{
9. @Override
10. **public** **void** run(){
11. System.out.println(getName() + " is running.");
12. **try**{
13. Thread.sleep(5000);
14. }**catch**(InterruptedException e){
15. e.printStackTrace();
16. }
17. System.out.println(getName() + " is running again.");
18. }
19. }

### AppConfig.java

**[java]** [view plain](http://blog.csdn.net/chszs/article/details/8219189) [copy](http://blog.csdn.net/chszs/article/details/8219189)

[print](http://blog.csdn.net/chszs/article/details/8219189)[?](http://blog.csdn.net/chszs/article/details/8219189)

1. **package** com.chszs.config;
3. **import** org.springframework.context.annotation.ComponentScan;
4. **import** org.springframework.context.annotation.Configuration;
6. @Configuration
7. @ComponentScan(basePackages="com.chszs.thread")
8. **public** **class** AppConfig {
9. }

### App.java

**[java]** [view plain](http://blog.csdn.net/chszs/article/details/8219189) [copy](http://blog.csdn.net/chszs/article/details/8219189)

[print](http://blog.csdn.net/chszs/article/details/8219189)[?](http://blog.csdn.net/chszs/article/details/8219189)

1. **package** com.chszs;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.annotation.AnnotationConfigApplicationContext;
5. **import** com.chszs.config.AppConfig;
6. **import** com.chszs.thread.PrintThread;
8. **public** **class** App {
9. **public** **static** **void** main(String[] args){
10. ApplicationContext ctx =
11. **new** AnnotationConfigApplicationContext(AppConfig.**class**);
12. PrintThread printThread1 = (PrintThread)ctx.getBean("printThread");
13. printThread1.setName("Thread 1");
15. PrintThread printThread2 = (PrintThread)ctx.getBean("printThread");
16. printThread2.setName("Thread 2");
18. PrintThread printThread3 = (PrintThread)ctx.getBean("printThread");
19. printThread3.setName("Thread 3");
21. PrintThread printThread4 = (PrintThread)ctx.getBean("printThread");
22. printThread4.setName("Thread 4");
24. PrintThread printThread5 = (PrintThread)ctx.getBean("printThread");
25. printThread5.setName("Thread 5");
27. printThread1.start();
28. printThread2.start();
29. printThread3.start();
30. printThread4.start();
31. printThread5.start();
32. }
33. }

### 输出：

Thread 1 is running.  
Thread 2 is running.  
Thread 4 is running.  
Thread 5 is running.  
Thread 3 is running.  
Thread 2 is running again.  
Thread 1 is running again.  
Thread 5 is running again.  
Thread 4 is running again.  
Thread 3 is running again.

## 例子2：Spring线程池结合非Spring托管Bean。

使用Spring的ThreadPoolTaskExecutor类创建一个线程池。执行线程无需受Spring容器的管理。

### PrintTask.java

**[java]** [view plain](http://blog.csdn.net/chszs/article/details/8219189) [copy](http://blog.csdn.net/chszs/article/details/8219189)

[print](http://blog.csdn.net/chszs/article/details/8219189)[?](http://blog.csdn.net/chszs/article/details/8219189)

1. **package** com.chszs.thread;
3. **public** **class** PrintTask **implements** Runnable{
4. String name;
5. **public** PrintTask(String name){
6. **this**.name = name;
7. }
8. @Override
9. **public** **void** run() {
10. System.out.println(name + " is running.");
11. **try**{
12. Thread.sleep(5000);
13. }**catch**(InterruptedException e){
14. e.printStackTrace();
15. }
16. System.out.println(name + " is running again.");
17. }
19. }

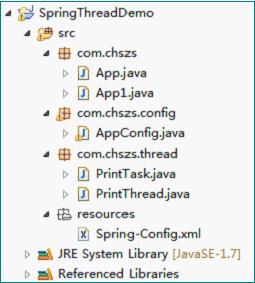
### Spring-Config.xml

**[html]** [view plain](http://blog.csdn.net/chszs/article/details/8219189) [copy](http://blog.csdn.net/chszs/article/details/8219189)

[print](http://blog.csdn.net/chszs/article/details/8219189)[?](http://blog.csdn.net/chszs/article/details/8219189)

1. **<beans** xmlns="http://www.springframework.org/schema/beans"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xmlns:context="http://www.springframework.org/schema/context"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
6. http://www.springframework.org/schema/context
7. http://www.springframework.org/schema/context/spring-context-3.1.xsd"**>**
9. **<bean** id="taskExecutor"
10. class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor"**>**
11. **<property** name="corePoolSize" value="5" **/>**
12. **<property** name="maxPoolSize" value="10" **/>**
13. **<property** name="WaitForTasksToCompleteOnShutdown" value="true" **/>**
14. **</bean>**
15. **</beans>**

注意这个Spring配置文件的位置，如图所示：



### App1.java

**[java]** [view plain](http://blog.csdn.net/chszs/article/details/8219189) [copy](http://blog.csdn.net/chszs/article/details/8219189)

[print](http://blog.csdn.net/chszs/article/details/8219189)[?](http://blog.csdn.net/chszs/article/details/8219189)

1. **package** com.chszs;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **import** org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;
7. **import** com.chszs.thread.PrintTask;
9. **public** **class** App1 {
11. **public** **static** **void** main(String[] args) {
12. ApplicationContext ctx =
13. **new** ClassPathXmlApplicationContext("resources/Spring-Config.xml");
14. ThreadPoolTaskExecutor taskExecutor =
15. (ThreadPoolTaskExecutor)ctx.getBean("taskExecutor");
16. taskExecutor.execute(**new** PrintTask("Thread 1"));
17. taskExecutor.execute(**new** PrintTask("Thread 2"));
18. taskExecutor.execute(**new** PrintTask("Thread 3"));
19. taskExecutor.execute(**new** PrintTask("Thread 4"));
20. taskExecutor.execute(**new** PrintTask("Thread 5"));
21. // 检查活动的线程，如果活动线程数为0则关闭线程池
22. **for**(;;){
23. **int** count = taskExecutor.getActiveCount();
24. System.out.println("Active Threads : " + count);
25. **try**{
26. Thread.sleep(1000);
27. }**catch**(InterruptedException e){
28. e.printStackTrace();
29. }
30. **if**(count==0){
31. taskExecutor.shutdown();
32. **break**;
33. }
34. }
35. }
37. }

### 输出：

Thread 1 is running.  
Thread 2 is running.  
Thread 3 is running.  
Thread 4 is running.  
Active Threads : 4  
Thread 5 is running.  
Active Threads : 5  
Active Threads : 5  
Active Threads : 5  
Active Threads : 5  
Active Threads : 5  
Thread 4 is running again.  
Thread 2 is running again.  
Thread 3 is running again.  
Thread 1 is running again.  
Thread 5 is running again.  
Active Threads : 0  
  
**作者：chszs，转载需注明。博客主页：http://blog.csdn.net/chszs**

## 例子3：Spring线程池结合Spring托管Bean。

本例仍然使用ThreadPoolTaskExecutor类，并使用@Component注释声明Spring的托管Bean。  
下面的例子PrintTask2是Spring的托管Bean，使用@Autowired注释简化代码。

### PrintTask2.java

**[java]** [view plain](http://blog.csdn.net/chszs/article/details/8219189) [copy](http://blog.csdn.net/chszs/article/details/8219189)

[print](http://blog.csdn.net/chszs/article/details/8219189)[?](http://blog.csdn.net/chszs/article/details/8219189)

1. **package** com.chszs.thread;
3. **import** org.springframework.context.annotation.Scope;
4. **import** org.springframework.stereotype.Component;
6. @Component
7. @Scope("prototype")
8. **public** **class** PrintTask2 **implements** Runnable {
9. String name;
11. **public** **void** setName(String name) {
12. **this**.name = name;
13. }
15. @Override
16. **public** **void** run(){
17. System.out.println(name + " is running.");
18. **try**{
19. Thread.sleep(5000);
20. }**catch**(InterruptedException e){
21. e.printStackTrace();
22. }
23. System.out.println(name + " is running again.");
24. }
25. }

### AppConfig.java

**[java]** [view plain](http://blog.csdn.net/chszs/article/details/8219189) [copy](http://blog.csdn.net/chszs/article/details/8219189)

[print](http://blog.csdn.net/chszs/article/details/8219189)[?](http://blog.csdn.net/chszs/article/details/8219189)

1. **package** com.chszs.config;
3. **import** org.springframework.context.annotation.Bean;
4. **import** org.springframework.context.annotation.ComponentScan;
5. **import** org.springframework.context.annotation.Configuration;
6. **import** org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;
8. @Configuration
9. @ComponentScan(basePackages="com.chszs.thread")
10. **public** **class** AppConfig {
11. @Bean
12. **public** ThreadPoolTaskExecutor taskExecutor(){
13. ThreadPoolTaskExecutor pool = **new** ThreadPoolTaskExecutor();
14. pool.setCorePoolSize(5);
15. pool.setMaxPoolSize(10);
16. pool.setWaitForTasksToCompleteOnShutdown(**true**);
17. **return** pool;
18. }
19. }

### App2.java

**[java]** [view plain](http://blog.csdn.net/chszs/article/details/8219189) [copy](http://blog.csdn.net/chszs/article/details/8219189)

[print](http://blog.csdn.net/chszs/article/details/8219189)[?](http://blog.csdn.net/chszs/article/details/8219189)

1. **package** com.chszs;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.annotation.AnnotationConfigApplicationContext;
5. **import** org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;
7. **import** com.chszs.config.AppConfig;
8. **import** com.chszs.thread.PrintTask2;
10. **public** **class** App2 {
11. **public** **static** **void** main(String[] args) {
12. ApplicationContext ctx =
13. **new** AnnotationConfigApplicationContext(AppConfig.**class**);
14. ThreadPoolTaskExecutor taskExecutor =
15. (ThreadPoolTaskExecutor)ctx.getBean("taskExecutor");
17. PrintTask2 printTask1 = (PrintTask2)ctx.getBean("printTask2");
18. printTask1.setName("Thread 1");
19. taskExecutor.execute(printTask1);
21. PrintTask2 printTask2 = (PrintTask2)ctx.getBean("printTask2");
22. printTask2.setName("Thread 2");
23. taskExecutor.execute(printTask2);
25. PrintTask2 printTask3 = (PrintTask2)ctx.getBean("printTask2");
26. printTask3.setName("Thread 3");
27. taskExecutor.execute(printTask3);
29. **for**(;;){
30. **int** count = taskExecutor.getActiveCount();
31. System.out.println("Active Threads : " + count);
32. **try**{
33. Thread.sleep(1000);
34. }**catch**(InterruptedException e){
35. e.printStackTrace();
36. }
37. **if**(count==0){
38. taskExecutor.shutdown();
39. **break**;
40. }
41. }
42. }
44. }

### 输出：

Thread 1 is running.  
Thread 2 is running.  
Active Threads : 2  
Thread 3 is running.  
Active Threads : 3  
Active Threads : 3  
Active Threads : 3  
Active Threads : 3  
Thread 1 is running again.  
Thread 2 is running again.  
Thread 3 is running again.  
Active Threads : 1  
Active Threads : 0  
  
**从这三个简单的实例中，你是不是发现了Spring框架在多线程方面的强大之处！！**