

ERP定时任务，多线程管理

简述：erp项目中，涉及多个第三方平台的订单拉取任务，订单状态同步任务……，这些都需要写一个定时任务，按照一定时间频率来进行任务的执行，在erp中，通过配置统一的线程池来管理，现做简要总结

一，编写通用线程池

erp中有两个通用线程池，第一个：CommonThreadPool，用于管理一般不需要定时的任务代码如下：

```
1  /**
2   *
3   */
4  package com.samton.erp.api.orders.thread.pool;
5
6  import java.util.concurrent.BlockingQueue;
7  import java.util.concurrent.LinkedBlockingDeque;
8  import java.util.concurrent.ThreadPoolExecutor;
9  import java.util.concurrent.TimeUnit;
10
11 import org.apache.log4j.Logger;
12
13 import com.samton.erp.api.orders.thread.NamedThreadFactory;
14
15 /**
16  *
17  * @Description:通用线程池
18  * Copyright (c) 2016, Sutu. All rights reserved
19  */
20 public class CommonThreadPool {
21
22     private static final Logger logger =
23         Logger.getLogger(CommonThreadPool.class);
24
25     /** 提供一个通用的线程池，可以直接使用 */
26     private static ThreadPoolExecutor threadPool;
27
28     private static final int processorCount =
29         Runtime.getRuntime().availableProcessors();
30 }
```

```

29     static{
30         BlockingQueue<Runnable> workQueue = new
LinkedBlockingDeque<Runnable>(10000);
31
32         threadPool = new ThreadPoolExecutor(processorCount * 20,
processorCount * 1000, 1, TimeUnit.MINUTES, workQueue, new
NamedThreadFactory("sutu-thread-pool"));
33     }
34
35     public static ThreadPoolExecutor getThreadPool(String why){
36         if(logger.isInfoEnabled()){
37             logger.debug("准备申请线程资源执行[" + why + "]");
38         }
39         return threadPool;
40     }
41
42     /**
43      *
44      * @Title:      shutdown
45      * @Description: 关闭通用线程池
46      * @param:
47      * @return:      void
48      * @author      李建洲
49      * @Date        2016年10月15日 下午7:54:52
50      */
51     public static void shutdown(){
52         logger.debug("通用线程池准备关闭");
53         threadPool.shutdown();
54     }
55 }
56

```

线程池方面，选择了ThreadPoolExecutor这个线程池实现类作为统一线程池：

<https://flowus.cn/1b88509c-39fd-4f73-9bd1-ee20d53f4f8c>

里面有一个参数是线程工厂，用于规定新线程的构造方式：

<https://flowus.cn/368c0980-e3f1-49a1-87c2-1836c215d82d>

在ERP中，自定义了一个线程工程来自动生成指定的线程名称，方便管理和调试

```

▼
1  /**
2   *
3   */
4  package com.samton.erp.api.orders.thread;
5
6  import java.util.concurrent.ThreadFactory;

```

```

7 import java.util.concurrent.atomic.AtomicInteger;
8
9 /**
10  *
11  * @Description:线程的Factory类，以便自定义线程名
12  * @author:      lijianzhou
13  * @date:        2016年3月27日
14  * Copyright (c) 2016, Sutu. All rights reserved
15  */
16 public class NamedThreadFactory implements ThreadFactory {
17
18     static final AtomicInteger poolNumber = new AtomicInteger(1);
19
20     final AtomicInteger threadNumber = new AtomicInteger(1);
21
22     final ThreadGroup group;
23     final String namePrefix;
24     final boolean isDaemon;
25
26     public NamedThreadFactory() {
27         this("pool");
28     }
29
30     public NamedThreadFactory(String name) {
31         this(name, false);
32     }
33
34     public NamedThreadFactory(String prefix, boolean daemon) {
35         SecurityManager s = System.getSecurityManager();
36         group = (s != null) ? s.getThreadGroup() :
Thread.currentThread().getThreadGroup();
37         namePrefix = prefix + "-" + poolNumber.getAndIncrement() + "-
thread-";
38         isDaemon = daemon;
39     }
40
41
42     @Override
43     public Thread newThread(Runnable r) {
44         Thread t = new Thread(group, r, namePrefix +
threadNumber.getAndIncrement(), 0);
45         t.setDaemon(isDaemon);
46         if(t.getPriority() != Thread.NORM_PRIORITY){
47             t.setPriority(Thread.NORM_PRIORITY);
48         }
49         return t;
50     }
51
52 }

```

```
static final AtomicInteger poolNumber = new AtomicInteger(1);
```

```
final AtomicInteger threadNumber = new AtomicInteger(1);
```

这两个是原子类，保证高并发环境下数值唯一，具体方法在JUC当中

二，CommonScheduledThreadPool,专门用来管理定时任务

```
1  /**
2   *
3   */
4  package com.samton.erp.api.orders.thread.pool;
5
6  import java.util.concurrent.ScheduledThreadPoolExecutor;
7
8  import org.apache.log4j.Logger;
9
10 import com.samton.erp.api.orders.thread.NamedThreadFactory;
11
12 /**
13  *
14  * @Description:通用调度任务线程池
15  * Copyright (c) 2016, Sutu. All rights reserved
16  */
17 public class CommonScheduledThreadPool {
18
19     private static final Logger logger =
20         Logger.getLogger(CommonScheduledThreadPool.class);
21
22     // 2*core
23     private static final int processorCount =
24         Runtime.getRuntime().availableProcessors() * 20;
25
26     private static ScheduledThreadPoolExecutor threadPool;
27
28     static{
29         threadPool = new ScheduledThreadPoolExecutor(processorCount, new
30             NamedThreadFactory("sutu-common-schedule-pool"));
31     }
32
33     public static ScheduledThreadPoolExecutor getScheduledThreadPool(String
34         why){
35         if(logger.isInfoEnabled()){
36             logger.debug("准备申请线程资源执行[" + why + "]");
37         }
38     }
39 }
```

```

33     }
34     return threadPool;
35 }
36
37 public static void shutdown(){
38     logger.debug("通用调度线程池准备关闭");
39     threadPool.shutdown();
40 }
41 }
42

```

这里的线程池实现类使用了ScheduledThreadPoolExecutor，有很多可以创建定时线程任务的方法：<https://flowus.cn/7b020d43-9a6d-4d3c-8000-f994fe4c92c3>

这里有一个很重要的点，也是我觉得的erp定时任务性能瓶颈的原因之一

创建线程最关键的的因素之一是线程池的核心线程数，erp中使用`private static final int processorCount = Runtime.getRuntime().availableProcessors() * 20`;来规定核心线程数：

`Runtime.getRuntime().availableProcessors()` 是 Java 中获取当前计算机上可用处理器核心数量的一种方式。这个方法返回的是一个整数，表示系统的逻辑处理器的数量。在现代多核处理器架构中，每个物理 CPU 可能包含多个核心，而每个核心可能支持多个线程（如 Intel 的 Hyper-Threading 技术），所以实际可用的逻辑处理器数量可能会比物理核心数更多。

这个方法似乎是获取了当前计算机可承载的最大线程数，但是这个类是静态的，是随类初始化一起加载的不可变的，一旦加载值就是固定的，猜测会有更好的方法

Java 中的 `ThreadPoolExecutor` 提供了多种方法来动态调整线程池的大小。以下是一些常用的方法和步骤来实现动态调整线程池大小：

- 1，修改核心线程数：setCorePoolSize
- 2，修改最大线程数：setMaximumPoolSize

或许有方法可以动态获取当前虚拟机有关性能的信息，然后动态更改核心线程数？如下示例：

```

1 long availableMemory = Runtime.getRuntime().maxMemory(); // 获取最大可用内存
2 long threadStackSize = 512 * 1024; // 每个线程栈大小为 512KB
3 long maxThreads = availableMemory / threadStackSize; // 估算最大线程数

```

三，编写抽象类

erp复用性较高的是定时任务的线程池，这里是为定时任务写的
目的是为了管理线程任务，统一规范，易于编写（我猜的）
代码如下：

一，先编写一个接口

```
1 /**
2  *
3  */
4 package com.samton.erp.api.orders.thread.schedule;
5
6 import java.util.concurrent.TimeUnit;
7
8 /**
9  *
10 * @Description:调度任务
11 * Copyright (c) 2016, Sutu. All rights reserved
12 */
13 public interface ScheduleTask extends Runnable {
14
15     /**
16      *
17      * @Title:      init
18      * @Description: 初始化
19      * @param:
20      * @return:      void
21      * @Date        2016年10月15日 下午7:33:03
22      */
23     void init();
24
25     /**
26      *
27      * @Title:      close
28      * @Description: 关闭
29      * @param:
30      * @return:      void
31      * @Date        2016年10月15日 下午7:33:14
32      */
33     void close();
34
35     /**
36      *
37      * @Title:      getWhy
38      * @Description: 申请调度线程的目的
39      * @param:
40      * @return
```

```

40     * @return:      String
41     * @Date        2016年10月15日 下午7:33:24
42     */
43     String getWhy();
44
45     /**
46     *
47     * @Title:        addSchedule
48     * @Description:   添加一个调度。默认添加固定延迟500ms钟的定时任务
49     * @param:
50     * @return:       void
51     * @Date        2016年10月15日 下午7:33:39
52     */
53     void addSchedule();
54
55     /**
56     *
57     * @Title:        addSchedule
58     * @Description:   添加一个调度。默认添加固定延迟1s钟的定时任务
59     *                注：建议只加入同一个类型的调度任务
60     * @param:        @param name 调度任务名称
61     * @param:        @param scheduleTask
62     * @return:       void
63     * @Date        2016年10月15日 下午7:33:47
64     */
65     void addSchedule(String name, ScheduleTask scheduleTask);
66
67     /**
68     *
69     * @Title:        addScheduleWithFixedDelay
70     * @Description:   添加一个固定延时的调度任务
71     * @param:        @param initialDelay
72     * @param:        @param delay
73     * @param:        @param unit
74     * @return:       void
75     * @Date        2016年10月15日 下午7:37:27
76     */
77     void addScheduleWithFixedDelay(long initialDelay, long delay,
78                                     TimeUnit unit);
79
80     /**
81     *
82     * @Title:        addScheduleAtFixedRate
83     * @Description:   添加一个固定频率的调度任务
84     * @param:        @param initialDelay
85     * @param:        @param period
86     * @param:        @param unit
87     * @return:       void

```

```
87      * @Date          2016年10月15日 下午7:37:41
88      */
89      void addScheduleAtFixedRate(long initialDelay, long period,
TimeUnit unit);
90
91      /**
92      *
93      * @Title:          removeSchedule
94      * @Description:    移除一个调度
95      * @param:
96      * @return:         void
97      * @Date           2016年10月15日 下午7:37:52
98      */
99      void removeSchedule();
100
101      /**
102      *
103      * @Title:          removeSchedule
104      * @Description:    移除一个调度
105      * @param:         @param name    调度任务名称
106      * @return:         void
107      * @Date           2016年10月15日 下午7:50:47
108      */
109      void removeSchedule(String name);
110
111      /**
112      *
113      * @Title:          getScheduleTaskSize
114      * @Description:    获取调度任务的个数
115      * @param:         @return
116      * @return:         int
117      * @Date           2016年10月15日 下午7:51:03
118      */
119      int getScheduleTaskSize();
120
121      /**
122      *
123      * @Title:          getScheduleTaskSize
124      * @Description:    指定调度任务的个数
125      * @param:         @param name
126      * @param:         @return
127      * @return:         int
128      * @Date           2016年10月15日 下午7:51:14
129      */
130      int getScheduleTaskSize(String name);
131  }
132
```


感觉这个设计很繁琐，明明就一个实现类为什么还要专门写一个接口呢？

二，编写抽象实现类

代码如下：

```
1  /**
2   *
3   */
4  package com.samton.erp.api.orders.thread.schedule;
5
6  import java.util.ArrayList;
7  import java.util.HashMap;
8  import java.util.List;
9  import java.util.Map;
10 import java.util.concurrent.ScheduledFuture;
11 import java.util.concurrent.TimeUnit;
12
13 import org.apache.log4j.Logger;
14
15 import com.samton.erp.api.orders.thread.pool.CommonScheduledThreadPool;
16
17 /**
18  *
19  * @Description:定时线程任务调度类
20  * @date:      2016年3月27日
21  * Copyright (c) 2016, Sutu. All rights reserved
22  */
23 public abstract class AbstractScheduleTask implements ScheduleTask {
24
25     private final Logger logger = Logger.getLogger(this.getClass());
26
27     private final Map<String, List<ScheduledFuture<?>>> scheduleTaskMap
28     = new HashMap<String, List<ScheduledFuture<?>>>();
29
30     private final String DEFAULT_SCHEDULE_TASK =
31     "default_schedule_task";
32
33     /** 缺省初始延迟10ms */
34     private final int DEFAULT_INITAIL_DELAY = 10;
35
36     /** 缺省的延迟500ms */
37     private final int DEFAULT_DELAY = 500;
38
39     /** 当前线程继续执行还是放弃本次执行等待下次调度*/
```

```

38     private final ThreadLocal<Boolean> continueExeThreadLocal = new
ThreadLocal<Boolean>();
39
40     private final Object lock = new Object();
41     @Override
42     public void run() {
43         try{
44             resetExecute();//恢复执行状态
45             long begin = System.currentTimeMillis();
46             logger.debug(getWhy() + "线程调度进行");
47             //快速响应容器关闭
48             while(isContinueExecute()){
49                 // 设置为不连续执行
50                 continueExecute(false);
51                 // 子类执行方法
52                 execute();
53                 //累计执行超过10s则放弃本次调度
54                 if((System.currentTimeMillis() - begin) > 10000){
55                     break;
56                 }
57             }
58             }catch(Throwable e){
59                 e.printStackTrace();
60             }
61         }
62
63         /**
64          *
65          * @Title:      init
66          * @Description: 初始化
67          * @param:
68          * @return:      void
69          * @Date        2016年10月15日 下午7:33:03
70          */
71         @Override
72         public void init() {
73             List<ScheduledFuture<?>> scheduledFutures = new
ArrayList<ScheduledFuture<?>>();
74             scheduleTaskMap.put(DEFAULT_SCHEDULE_TASK, scheduledFutures);
75         }
76
77         /**
78          *
79          * @Title:      close
80          * @Description: 关闭
81          * @param:
82          * @return:      void
83          * @Date        2016年10月15日 下午7:33:14

```

```

84     */
85     @Override
86     public void close() {
87         synchronized (lock) {
88             for (Map.Entry<String, List<ScheduledFuture<?>>> entry :
scheduleTaskMap.entrySet()) {
89                 List<ScheduledFuture<?>> scheduleFutures =
entry.getValue();
90                 for(ScheduledFuture<?> scheduleFuture :
scheduleFutures){
91                     scheduleFuture.cancel(false);
92                 }
93             }
94         }
95     }
96
97     /**
98     *
99     * @Title:         addSchedule
100    * @Description:    添加一个调度。默认添加固定延迟500ms的定时任务
101    * @param:
102    * @return:         void
103    * @Date           2016年10月15日 下午7:33:39
104    */
105    @Override
106    public void addSchedule() {
107        synchronized (lock) {
108            ScheduledFuture<?> scheduledFuture =
CommonScheduledThreadPool.getScheduledThreadPool(getWhy())
109                .scheduleWithFixedDelay(this,
DEFAULT_INITAIL_DELAY, DEFAULT_DELAY, TimeUnit.MILLISECONDS);
110            doAddScheduleFuture(DEFAULT_SCHEDULE_TASK,
scheduledFuture);
111        }
112    }
113
114    /**
115    *
116    * @Title:         addSchedule
117    * @Description:    添加一个调度。默认添加固定延迟1s的定时任务
118    *                  注：建议只加入同一个类型的调度任务
119    * @param:         @param name 调度任务名称
120    * @param:         @param scheduleTask
121    * @return:         void
122    * @Date           2016年10月15日 下午7:33:47
123    */
124    @Override
125    public void addSchedule(String name, ScheduleTask scheduleTask) {

```

```

126         synchronized (lock) {
127             ScheduledFuture<?> scheduledFuture =
CommonScheduledThreadPool.getScheduledThreadPool(getWhy())
128                 .scheduleWithFixedDelay(scheduleTask,
DEFAULT_INITAIL_DELAY, DEFAULT_DELAY, TimeUnit.MILLISECONDS);
129             doAddScheduleFuture(DEFAULT_SCHEDULE_TASK,
scheduledFuture);
130         }
131     }
132
133     /**
134     *
135     * @Title:         addScheduleWithFixedDelay
136     * @Description:    添加一个固定延时的调度任务
137     * @param:         @param initialDelay
138     * @param:         @param delay
139     * @param:         @param unit
140     * @return:        void
141     * @Date           2016年10月15日 下午7:37:27
142     */
143     @Override
144     public void addScheduleWithFixedDelay(long initialDelay, long
delay, TimeUnit unit) {
145         synchronized (lock) {
146             ScheduledFuture<?> scheduledFuture =
CommonScheduledThreadPool.getScheduledThreadPool(getWhy())
147                 .scheduleWithFixedDelay(this, initialDelay, delay,
unit);
148             doAddScheduleFuture(DEFAULT_SCHEDULE_TASK,
scheduledFuture);
149         }
150     }
151
152     /**
153     *
154     * @Title:         addScheduleAtFixedRate
155     * @Description:    添加一个固定频率的调度任务
156     * @param:         @param initialDelay
157     * @param:         @param period
158     * @param:         @param unit
159     * @return:        void
160     * @Date           2016年10月15日 下午7:37:41
161     */
162     @Override
163     public void addScheduleAtFixedRate(long initialDelay, long period,
TimeUnit unit) {
164         ScheduledFuture<?> scheduledFuture =
CommonScheduledThreadPool.getScheduledThreadPool(getWhy())

```

```

165         .scheduleAtFixedRate(this, initialDelay, period, unit);
166         doAddScheduleFuture(DEFAULT_SCHEDULE_TASK, scheduledFuture);
167     }
168
169     private void doAddScheduleFuture(String name, ScheduledFuture<?>
scheduledFuture){
170         List<ScheduledFuture<?>> scheduledFutures =
scheduleTaskMap.get(name);
171         if(scheduledFutures == null){
172             scheduledFutures = new ArrayList<ScheduledFuture<?>>();
173             scheduleTaskMap.put(DEFAULT_SCHEDULE_TASK,
scheduledFutures);
174         }
175
176         scheduledFutures.add(scheduledFuture);
177     }
178
179     /**
180      *
181      * @Title:         removeSchedule
182      * @Description:    移除一个调度
183      * @param:
184      * @return:         void
185      * @Date           2016年10月15日 下午7:37:52
186      */
187     @Override
188     public void removeSchedule() {
189         synchronized (lock) {
190             removeSchedule(DEFAULT_SCHEDULE_TASK);
191         }
192     }
193
194     /**
195      *
196      * @Title:         removeSchedule
197      * @Description:    移除一个调度
198      * @param:         @param name    调度任务名称
199      * @return:         void
200      * @Date           2016年10月15日 下午7:50:47
201      */
202     @Override
203     public void removeSchedule(String name) {
204         synchronized (lock) {
205             List<ScheduledFuture<?>> scheduledFutures =
scheduleTaskMap.get(name);
206             if(scheduledFutures == null){
207                 return;
208             }

```

```

209
210         if(scheduledFutures.isEmpty()){
211             scheduleTaskMap.remove(name);
212         }else{
213             //移除最后一个
214             ScheduledFuture<?> scheduledFuture =
scheduledFutures.remove(scheduledFutures
215                 .size() - 1);
216             scheduledFuture.cancel(false);
217         }
218     }
219 }
220
221 /**
222  *
223  * @Title:         getScheduleTaskSize
224  * @Description:    获取调度任务的个数
225  * @param:         @return
226  * @return:         int
227  * @Date           2016年10月15日 下午7:51:03
228  */
229 @Override
230 public int getScheduleTaskSize() {
231     return getScheduleTaskSize(DEFAULT_SCHEDULE_TASK);
232 }
233
234 /**
235  *
236  * @Title:         getScheduleTaskSize
237  * @Description:    指定调度任务的个数
238  * @param:         @param name
239  * @param:         @return
240  * @return:         int
241  * @Date           2016年10月15日 下午7:51:14
242  */
243 @Override
244 public int getScheduleTaskSize(String name) {
245     List<ScheduledFuture<?>> scheduledFutures =
scheduleTaskMap.get(name);
246     if(scheduledFutures == null){
247         return 0;
248     }else{
249         return scheduledFutures.size();
250     }
251 }
252
253 /**
254  * 子类应该实现的执行实际操作的方法.<br>

```

```

255     * 返回true继续执行<br>
256     * 返回false本次调度已执行完<br>
257     * 如果在一次调度中做批量操作,子类不建议采用while(true)独占调度线程实现方式,而
    应该采用小批量间歇执行,如while(i<=N)的方式.<br>
258     */
259     public abstract void execute();
260
261     protected void continueExecute(boolean conitueExe){
262         continueExeThreadLocal.set(conitueExe);
263     }
264
265     protected boolean isContinueExecute(){
266         return continueExeThreadLocal.get();
267     }
268
269     private void resetExecute(){
270         continueExecute(true);
271     }
272 }
273

```

这个抽象类实现了大部分接口的方法,但是没有实现execute()方法,应为不同的子类具体要做的逻辑不一样,所以以抽象方法的方式等待子类写具体的逻辑

execute()方法在run()方法内被调用,当调用scheduleAtFixedRate这种线程池方法时,会传入this参数,执行run()方法,间接调用execute方法

四, 编写具体实现逻辑

我写的一个示例:

```

▼
1 @Service("syncNaverOrder")
2 public class TestTask extends AbstractScheduleTask {
3     private final Logger logger = Logger.getLogger(this.getClass());
4
5     @Override
6     public void init() {
7         super.init();
8         addScheduleWithFixedDelay(1,6, TimeUnit.HOURS);
9     }
10    @Override
11    public void execute() {
12        for(int i = 0;i<10;i++){
13            CommonThreadPool.getThreadPool("furryLoveShou").execute(new
                TestTask().printLoveClass());
14        }

```

```

15
16 public printLoveClass implements Runnable{
17     @Override
18     public void run(){
19         sout("furryLoveShuo");
20     }
21
22 }
23

```

可以看到，这里CommonThreadPool又回来了，为什么这里要再调用一个线程池形成嵌套呢？

查了一下，这种模式有时被称为“任务链”或“级联任务”，好处

是：<https://flowus.cn/eb549d81-754c-436f-a26a-b6c23f588351>

如何启动想执行的任务呢？

1，以上的实现是通过调用init()方法初始化任务，设计定时间隔，先super()初始化父类线程列表，在添加一个定时任务，那么如何调用init()方法呢？

1，写init在erp中的目的是服务一启动就调用，erp的处理方式是通过编写.xml的方式调用的，编写了一个spring.xml，和web.xml，这年头谁还用啊？

2，不就是初始化bean指定初始化方法吗？使用配置类和注解

例：假设现在有一个类叫"TestTask"，里面有一个方法叫"init()"

```

1 import org.springframework.context.annotation.Bean;
2 import org.springframework.context.annotation.Configuration;
3
4 @Configuration
5 public class AppConfig {
6
7     @Bean(initMethod = "init")
8     public TestTask testTask() {
9         return new TestTask();
10    }
11 }

```

这样，服务启动就会执行init方法

3,还有一种方法是调用spring的ApplicationListener，没仔细研究，感兴趣可以了解一下

```

1 import org.springframework.context.ApplicationListener;

```



```

2 import org.springframework.context.event.ContextRefreshedEvent;
3
4 public class StartupInitializer implements
    ApplicationListener<ContextRefreshedEvent> {
5
6     @Override
7     public void onApplicationEvent(ContextRefreshedEvent event) {
8         // 初始化逻辑
9         System.out.println("Initializing during application
startup...");
10    }
11 }

```

2, 还可以再写一个方法, 绕过init, 手动调用执行execute方法:

```

1 public void submit() {
2     CommonThreadPool.getThreadPool("furryLoveShuo").execute(new
TestTask().printLoveClass());
3 }


```

这个方法跳过了定时任务, 而是手动调用submit方法新建一个线程执行一遍要执行的方法
 结束, 如果有对juc java并发感兴趣的可以进一步研究, 如果对上面提到的性能瓶颈有想法的欢迎
 讨论

 [Thread构造](#)

 [ThreadPoolExecutor](#)

 [线程工厂、级联任务](#)

 [任务链好处](#)

 [ScheduledThreadPoolExecutor](#)