# SpringBoot用线程池ThreadPoolTaskExecutor异步处理百万级数据

Original 程序猿羊 程序猿小杨 2023-06-13 07:50 Posted on 上海

收录于合集

#多线程 2 #线程池 4 #springboot 5

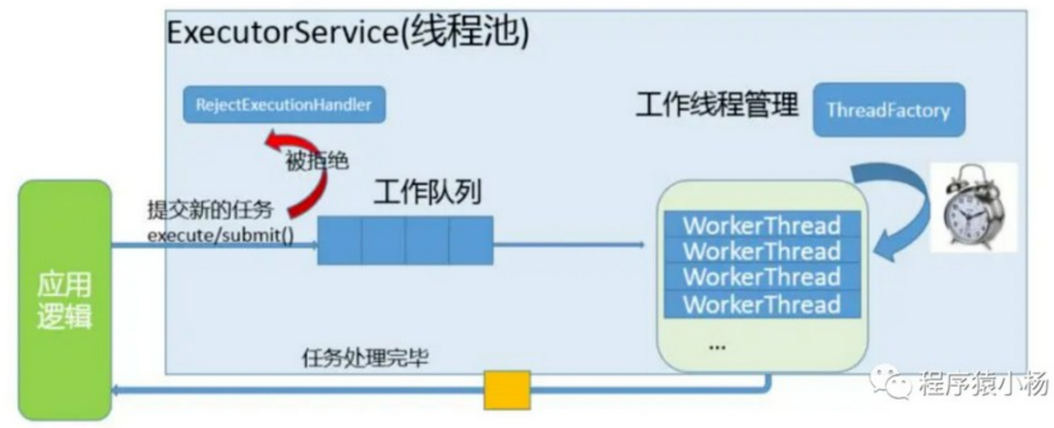


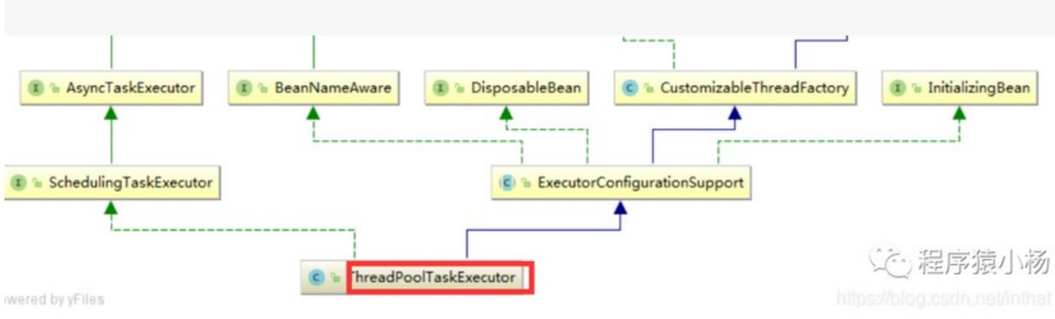
👍推荐大家关注一个公众号👍



## 一、背景：

利用ThreadPoolTaskExecutor多线程异步批量插入，提高百万级数据插入效率。ThreadPoolTaskExecutor是对ThreadPoolExecutor进行了封装处理。ThreadPoolTaskExecutor是ThreadPoolExecutor的封装，所以，性能更加优秀，推荐ThreadPoolTaskExecutor。




## 二、具体细节：

### 2.1、配置application.yml

```yaml
# 异步线程配置 自定义使用参数
async:
  executor:
    thread:
      core_pool_size:  10  # 配置核心线程数 默认8个 核数*2+2
      max_pool_size:  100   # 配置最大线程数
      queue_capacity:  99988  # 配置队列大小
      keep_alive_seconds:  20  #设置线程空闲等待时间秒s
      name:
        prefix: async-thread-  # 配置线程池中的线程的名称前缀
```

### 2.2、ThreadPoolConfig配置注入Bean

```java
package com.wonders.common.config;
import cn.hutool.core.thread.ThreadFactoryBuilder;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

/**
 * @Description: TODO:利用ThreadPoolTaskExecutor多线程批量执行相关配置
 * 自定义线程池
 * 发现不是线程数越多越好，具体多少合适，网上有一个不成文的算法：CPU核心数量*2 +2 个线
 * @Author: yyalin
 * @CreateDate: 2022/11/6 11:56
 * @Version: V1.0
 */
@Configuration
@EnableAsync
@Slf4j
public class ThreadPoolConfig {
    //自定义使用参数
    @Value("${async.executor.thread.core_pool_size}")
    private int corePoolSize;   //配置核心线程数
    @Value("${async.executor.thread.max_pool_size}")
    private int maxPoolSize;    //配置最大线程数
    @Value("${async.executor.thread.queue_capacity}")
    private int queueCapacity;
    @Value("${async.executor.thread.name.prefix}")
    private String namePrefix;
    @Value("${async.executor.thread.keep_alive_seconds}")
    private int keepAliveSeconds;

    //1、自定义asyncServiceExecutor线程池
    @Bean(name = "asyncServiceExecutor")
    public ThreadPoolTaskExecutor asyncServiceExecutor() {
        log.info("start asyncServiceExecutor......");
        //在这里修改
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        //配置核心线程数
        executor.setCorePoolSize(corePoolSize);
        //配置最大线程数
        executor.setMaxPoolSize(maxPoolSize);
        //设置线程空闲等待时间 s
        executor.setKeepAliveSeconds(keepAliveSeconds);
        //配置队列大小 设置任务等待队列的大小
        executor.setQueueCapacity(queueCapacity);
        //配置线程池中的线程的名称前缀
        //设置线程池内线程名称的前缀-------阿里编码规约推荐--方便出错后进行调试
        executor.setThreadNamePrefix(namePrefix);
        // rejection-policy：当pool已经达到max size的时候，如何处理新任务
        // CALLER_RUNS：不在新线程中执行任务，而是有调用者所在的线程来执行
        executor.setRejectedExecutionHandler(new ThreadPoolExecutor.DiscardPo
        //执行初始化
        executor.initialize();
        return executor;
    }
    /**
     * 2、公共线程池,利用系统availableProcessors线程数量进行计算
     */
    @Bean(name = "commonThreadPoolTaskExecutor")
    public ThreadPoolTaskExecutor commonThreadPoolTaskExecutor() {
        ThreadPoolTaskExecutor pool = new ThreadPoolTaskExecutor();
        int processNum = Runtime.getRuntime().availableProcessors(); // 返回可
        int corePoolSize = (int) (processNum / (1 - 0.2));
        int maxPoolSize = (int) (processNum / (1 - 0.5));
        pool.setCorePoolSize(corePoolSize); // 核心池大小
        pool.setMaxPoolSize(maxPoolSize); // 最大线程数
        pool.setQueueCapacity(maxPoolSize * 1000); // 队列程度
        pool.setThreadPriority(Thread.MAX_PRIORITY);
        pool.setDaemon(false);
        pool.setKeepAliveSeconds(300);// 线程空闲时间
        return pool;
    }
   //3自定义defaultThreadPoolExecutor线程池
    @Bean(name = "defaultThreadPoolExecutor", destroyMethod = "shutdown")
    public ThreadPoolExecutor systemCheckPoolExecutorService() {
```

```
        int maxNumPool=Runtime.getRuntime().availableProcessors();
        return new ThreadPoolExecutor(3,
                maxNumPool,
                60,
                TimeUnit.SECONDS,
                new LinkedBlockingQueue<Runnable>(10000),
                //置线程名前缀，例如设置前缀为hutool-thread-，则线程名为hutool-th
                new ThreadFactoryBuilder().setNamePrefix("default-executor-th
                (r, executor) -> log.error("system pool is full! "));
    }

}

```

### 2.3、创建异步线程，业务类

```
//1、自定义asyncServiceExecutor线程池
    @Override
    @Async("asyncServiceExecutor")
    public void executeAsync(List<Student> students,
                             StudentService studentService,
                             CountDownLatch countDownLatch) {
        try{
            log.info("start executeAsync");
            //异步线程要做的事情
            studentService.saveBatch(students);
            log.info("end executeAsync");
        }finally {
            countDownLatch.countDown();// 很关键, 无论上面程序是否异常必须执行cou
        }
    }
```

### 2.4、拆分集合工具类

```
package com.wonders.threads;

import com.google.common.collect.Lists;
import org.springframework.util.CollectionUtils;

import java.util.ArrayList;
import java.util.List;

/**
 * @Description: TODO: 拆分工具类
 * 1、获取需要进行批量更新的大集合A，对大集合进行拆分操作，分成N个小集合A-1 ~ A-N;
 * 2、开启线程池，针对集合的大小进行调参，对小集合进行批量更新操作;
 * 3、对流程进行控制，控制线程执行顺序。按照指定大小拆分集合的工具类
 * @Author: yyalin
 * @CreateDate: 2022/5/6 14:43
 * @Version: V1.0
 */
public class SplitListUtils {
    /**
     * 功能描述:拆分集合
     * @param <T> 泛型对象
     * @MethodName: split
     * @MethodParam: [resList:需要拆分的集合, subListLength:每个子集合的元素个数
     * @Return: java.util.List<java.util.List<T>>:返回拆分后的各个集合组成的列表
     * 代码里面用到了guava和common的结合工具类
     * @Author: yyalin
     * @CreateDate: 2022/5/6 14:44
     */
    public static <T> List<List<T>> split(List<T> resList, int subListLength
        if (CollectionUtils.isEmpty(resList) || subListLength <= 0) {
            return Lists.newArrayList();
        }
        List<List<T>> ret = Lists.newArrayList();
        int size = resList.size();
        if (size <= subListLength) {
            // 数据量不足 subListLength 指定的大小
            ret.add(resList);
        } else {
            int pre = size / subListLength;
            int last = size % subListLength;
            // 前面pre个集合，每个大小都是 subListLength 个元素
            for (int i = 0; i < pre; i++) {
                List<T> itemList = Lists.newArrayList();
                for (int j = 0; j < subListLength; j++) {
                    itemList.add(resList.get(i * subListLength + j));
                }
```

```
                ret.add(itemList);
            }
            // last的进行处理
            if (last > 0) {
                List<T> itemList = Lists.newArrayList();
                for (int i = 0; i < last; i++) {
                    itemList.add(resList.get(pre * subListLength + i));
                }
                ret.add(itemList);
            }
        }
        return ret;
    }

    /**
     * 功能描述:方法二：集合切割类，就是把一个大集合切割成多个指定条数的小集合，方便
     * 推荐使用
     * @MethodName: pagingList
     * @MethodParam:[resList:需要拆分的集合, subListLength:每个子集合的元素个数
     * @Return: java.util.List<java.util.List<T>>：返回拆分后的各个集合组成的列
     * @Author: yyalin
     * @CreateDate: 2022/5/6 15:15
     */
    public static <T> List<List<T>> pagingList(List<T> resList, int pageSize
        //判断是否为空
        if (CollectionUtils.isEmpty(resList) || pageSize <= 0) {
            return Lists.newArrayList();
        }
        int length = resList.size();
        int num = (length+pageSize-1)/pageSize;
        List<List<T>> newList =  new ArrayList<>();
        for(int i=0;i<num;i++){
            int fromIndex = i*pageSize;
            int toIndex = (i+1)*pageSize<length?(i+1)*pageSize:length;
            newList.add(resList.subList(fromIndex,toIndex));
        }
        return newList;
    }

    // 运行测试代码 可以按顺序拆分为11个集合
    public static void main(String[] args) {
        //初始化数据
        List<String> list = Lists.newArrayList();
        int size = 19;
        for (int i = 0; i < size; i++) {
            list.add("hello-" + i);
        }
        // 大集合里面包含多个小集合
        List<List<String>> temps = pagingList(list, 100);
        int j = 0;
        // 对大集合里面的每一个小集合进行操作
        for (List<String> obj : temps) {
            System.out.println(String.format("row:%s -> size:%s,data:%s", ++
        }
    }

}
```

### 2.5、造数据，多线程异步插入

```java
public int batchInsertWay() throws Exception {
        log.info("开始批量操作.........");
        Random rand = new Random();
        List<Student> list = new ArrayList<>();
        //造100万条数据
        for (int i = 0; i < 1000003; i++) {
            Student student=new Student();
            student.setStudentName("大明: "+i);
            student.setAddr("上海:"+rand.nextInt(9) * 1000);
            student.setAge(rand.nextInt(1000));
            student.setPhone("134"+rand.nextInt(9) * 1000);
            list.add(student);
        }
        //2、开始多线程异步批量导入
        long startTime = System.currentTimeMillis(); // 开始时间
        //boolean a=studentService.batchInsert(list);
        List<List<Student>> list1=SplitListUtils.pagingList(list,100);  //拆分
        CountDownLatch countDownLatch = new CountDownLatch(list1.size());
        for (List<Student> list2 : list1) {
            asyncService.executeAsync(list2,studentService,countDownLatch);
        }
        try {
            countDownLatch.await(); //保证之前的所有的线程都执行完成，才会走下面的
            long endTime = System.currentTimeMillis(); //结束时间
            log.info("一共耗时time: " + (endTime - startTime) / 1000 + " s");
            // 这样就可以在下面拿到所有线程执行完的集合结果
        } catch (Exception e) {
            log.error("阻塞异常:"+e.getMessage());
        }
        return list.size();

    }
```

### 2.6、测试结果



**10个核心线程：**

```
==> Parameters: 大明: 999999(String), 426(Integer), 1347000(String), 上海:1000(String)
2023-06-10 13:50:53 INFO  async-thread-10 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 13:50:53 INFO  async-thread-5 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 13:50:53 INFO  async-thread-6 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 13:50:53 INFO  async-thread-3 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 13:50:53 INFO  async-thread-4 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 13:50:53 INFO  http-nio-8081-exec-10 com.wonders.controller.StudentController 一共耗时time: 31 s
```



**20个核心线程**

```
==> Parameters: 大明: 999998(String), 532(Integer), 1344000(String), 上海:6000(String)
==> Parameters: 大明: 999999(String), 850(Integer), 1340(String), 上海:3000(String)
2023-06-10 14:08:16 INFO  async-thread-5 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 14:08:16 INFO  async-thread-3 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 14:08:16 INFO  async-thread-13 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 14:08:16 INFO  async-thread-15 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 14:08:16 INFO  async-thread-11 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 14:08:16 INFO  http-nio-8081-exec-1 com.wonders.controller.StudentController 一共耗时time: 28 s
```



**50个核心线程：**

```
==> Parameters: 大明: 999698(String), 859(Integer), 1341000(String), 上海:8000(String)
==> Parameters: 大明: 999699(String), 84(Integer), 1343000(String), 上海:2000(String)
2023-06-10 14:11:06 INFO  async-thread-30 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 14:11:06 INFO  async-thread-27 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 14:11:06 INFO  async-thread-49 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 14:11:06 INFO  async-thread-20 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 14:11:06 INFO  async-thread-43 com.wonders.threads.AsyncServiceImpl end executeAsync
2023-06-10 14:11:06 INFO  http-nio-8081-exec-1 com.wonders.controller.StudentController 一共耗时time: 27 s
```



**汇总结果：**

| 序号 | 核心线程(core_pool_size) | 插入数据(万) | 耗时（秒） |
| --- | --- | --- | --- |
| 1 | 10 | 100w | 31s |
| 2 | 15 | 100w | 28s |
| 3 | 50 | 100w | 27s |

**结论**：对不同线程数的测试，发现不是线程数越多越好，具体多少合适，网上有一个不成文的算法：CPU核心数量*2 +2 个线程。

**个人推荐配置：**

```
int processNum = Runtime.getRuntime().availableProcessors(); // 返回可用处理器的
int corePoolSize = (int) (processNum / (1 - 0.2));
int maxPoolSize = (int) (processNum / (1 - 0.5));
```

**更多优秀文章，请扫码关注个人微信公众号或搜索“程序猿小杨”添加。**

**推荐文章：**

1、SpringBoot使用@Async实现多线程异步;

收录于合集 #多线程 2

〈 上一篇 · SpringBoot用线程池ThreadPoolExecutor处理百万级数据