一、概要

1. 简介创建线程四种方式；
2. 介绍创建线程池几种方式，及线程池工作原理及参数介绍；
3. Springboot中如何自定义线程池；
4. Springboot 线程池项目Demo；
5. Q\A
6. 内容

什么是线程和进程？

进程：

进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。

在java中，当我们启动main函数时就是启动了一个JVM的进程，而main函数所在的线程就是这个进程中的一个线程，也称主线程。

线程：

线程和进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行过程中可以产生多个线程。

在java中，当我们启动main函数时其实就是启动了一个JVM的进程，而main函数所在的线程就是在这个进程的一个线程，也称为主线程。

1. **简介创建线程四种方式；**

Java使用Thread类来表示线程，所有的线程都是Thread类或者是他的子类。Java有四种方式来创建线程。

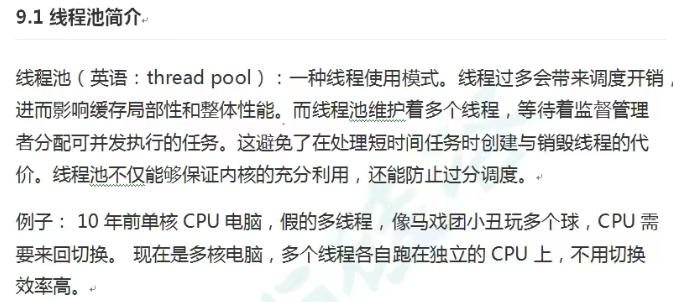
（1）继承Thread类创建线程（****重写run方法。****）

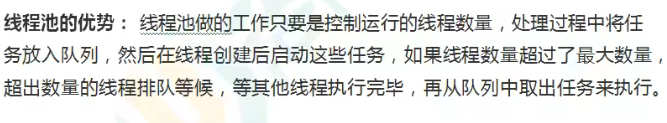
（2）实现Runnable接口创建线程（****重写run方法，不抛异常，无返回值。****）

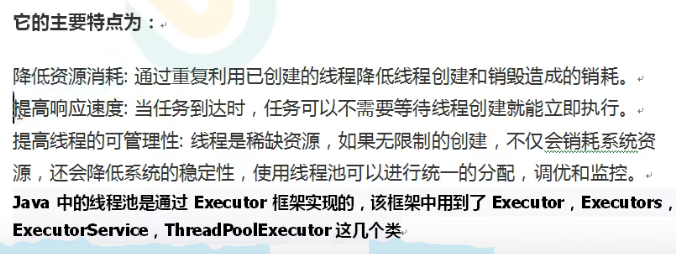
（3）使用Callable（****重写call方法，可抛异常，有返回值。****）和Future创建线程

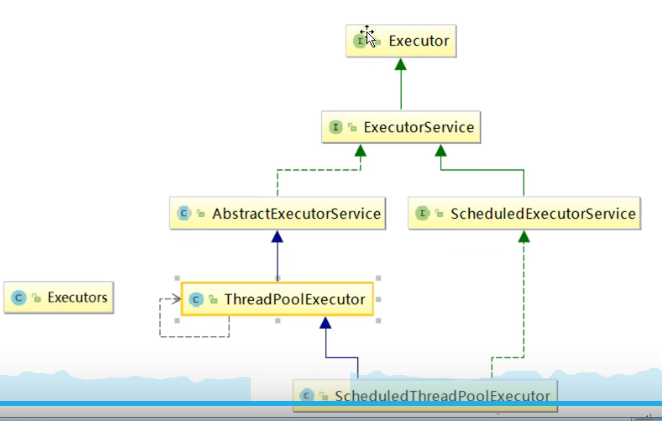
（4）使用线程池创建线程，例如使用Executor框架

1. **介绍创建线程池几种方式；**









Java通过Executors提供四种线程池，分别为：

》newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

》newFixedThreadPool  创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

》newScheduledThreadPool 创建一个可定期或者延时执行任务的定长线程池，支持定时及周期性任务执行。

》newCachedThreadPool 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

四种线程池本质都是创建ThreadPoolExecutor类。

**线程池的几个主要参数的作用**

corePoolSize: 规定线程池有几个线程(worker)在运行。

maximumPoolSize: 当workQueue满了,不能添加任务的时候，这个参数才会生效。规定线程池最多只能有多少个线程(worker)在执行。

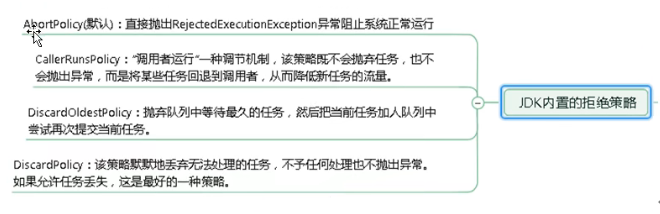
keepAliveTime: 超出corePoolSize大小的那些线程的生存时间,这些线程如果长时间没有执行任务并且超过了keepAliveTime设定的时间，就会消亡。

unit: 生存时间对于的单位

workQueue: 存放任务的队列

threadFactory: 创建线程的工厂

handler: 当workQueue已经满了，并且线程池线程数已经达到maximumPoolSize，将执行拒绝策略。



线程池任务执行流程：

》当线程池小于corePoolSize时，新提交任务将创建一个新线程执行任务，即使此时线程池中存在空闲线程。

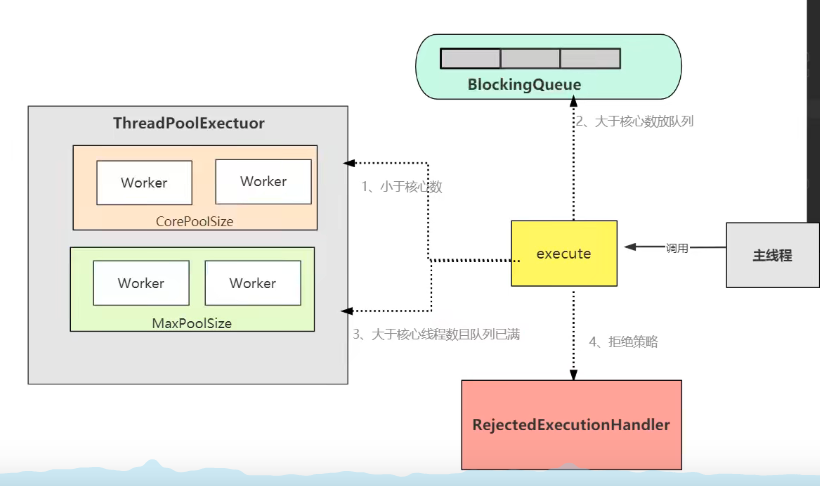
》当线程池达到corePoolSize时，新提交任务将被放入workQueue中，等待线程池中任务调度执行

》当workQueue已满，且maximumPoolSize>corePoolSize时，新提交任务会创建新线程执行任务

》当提交任务数超过maximumPoolSize时，新提交任务由RejectedExecutionHandler处理

》当线程池中超过corePoolSize线程，空闲时间达到keepAliveTime时，释放空闲线程

》当设置allowCoreThreadTimeOut(true)时，该参数默认false，线程池中corePoolSize线程空闲时间达到keepAliveTime也将关闭



【强制】线程池不允许使用Executors创建，建议通过ThreadPoolExecutor的方式创建，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明：Executors返回的线程池对象的弊端如下：

1.FixedThreadPool和SingleThreadPool:

允许的请求队列长度为Integet.MAX\_VALUE,可能会堆积大量的请求从而导致OOM;

2.CachedThreadPool:

允许创建线程数量为Integet.MAX\_VALUE,可能会创建大量的线程，从而导致OOM.

**如何自定义线程池**

**public class ThreadPoolDemo {**

**public static void main(String[] args) {**

**ThreadPoolExecutor threadPool = new ThreadPoolExecutor(**

**2,**

**5,**

**2L,**

**TimeUnit.SECONDS,**

**new ArrayBlockingQueue<>(90),**

**Executors.defaultThreadFactory(),**

**new ThreadPoolExecutor.AbortPolicy()**

**);**

**try {**

**for (int i = 0; i < 10; i++) {**

**threadPool.execute(() -> {**

**System.out.println(Thread.currentThread().getName() + " 办理业务！");**

**});**

**}**

**} catch (Exception e) {**

**e.printStackTrace();**

**} finally {**

**threadPool.shutdown();**

**}**

**}**

**}**

**Springboot 线程池项目Demo**

**四种线程池demo**

**package com.adam.thread;**

**import java.util.concurrent.ExecutorService;**

**import java.util.concurrent.Executors;**

**import java.util.concurrent.ScheduledExecutorService;**

**import java.util.concurrent.TimeUnit;**

**import static java.lang.System.out;**

**public class FourThreadPoolsDemo {**

**public static void main(String[] args) {**

**/\*\***

**\* 1.newCachedThreadPool：**

**\***

**\* 底层：返回ThreadPoolExecutor实例，**

**\* corePoolSize为0；maximumPoolSize为Integer.MAX\_VALUE；**

**\* keepAliveTime为60L；时间单位TimeUnit.SECONDS；**

**\* workQueue为SynchronousQueue(同步队列)**

**\* 通俗：当有新任务到来，则插入到SynchronousQueue中，由于SynchronousQueue是同步队列，**

**\* 因此会在池中寻找可用线程来执行，若有可以线程则执行，若没有可用线程则创建一个线程来执行该任务；**

**\* 若池中线程空闲时间超过指定时间，则该线程会被销毁。**

**\* 适用：执行很多短期的异步任务**

**\***

**\* 1.创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程<br>**

**\* 2.当任务数增加时，此线程池又可以智能的添加新线程来处理任务<br>**

**\* 3.此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小<br>**

**\*/**

**// ExecutorService cachedThreadPool = Executors.newCachedThreadPool();**

**// for (int i = 1; i <= 10; i++) {**

**// final int ii = i;**

**// try {**

**// Thread.sleep(ii \* 1);**

**// } catch (InterruptedException e) {**

**// e.printStackTrace();**

**// }**

**// cachedThreadPool.execute(() -> out.println("线程名称：" + Thread.currentThread().getName() + "，执行" + ii));**

**// }**

**/\*\***

**\***

**\* 2.newFixedThreadPool：**

**\***

**\* 底层：返回ThreadPoolExecutor实例，接收参数为所设定线程数量n，**

**\* corePoolSize和maximumPoolSize均为n；**

**\* keepAliveTime为0L；时间单位TimeUnit.MILLISECONDS；**

**\* WorkQueue为：new LinkedBlockingQueue<Runnable>() 无界阻塞队列**

**\* 通俗：创建可容纳固定数量线程的池子，每个线程的存活时间是无限的，当池子满了就不再添加线程了；**

**\* 如果池中的所有线程均在繁忙状态，对于新任务会进入阻塞队列中(无界的阻塞队列)**

**\* 适用：执行长期任务**

**\***

**\* 1.创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小<br>**

**\* 2.线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程<br>**

**\* 3.因为线程池大小为3，每个任务输出index后sleep 2秒，所以每两秒打印3个数字，和线程名称<br>**

**\*/**

**// ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);**

**// for (int i = 0; i < 10; i++) {**

**// final int ii = i;**

**// fixedThreadPool.execute(() -> {**

**// out.println("线程名称：" + Thread.currentThread().getName() + "，执行" + ii);**

**// try {**

**// Thread.sleep(2000);**

**// } catch (InterruptedException e) {**

**// e.printStackTrace();**

**// }**

**// });**

**// }**

**/\*\***

**\* 3.newSingleThreadExecutor:**

**\***

**\* 底层：FinalizableDelegatedExecutorService包装的ThreadPoolExecutor实例，**

**\* corePoolSize为1；maximumPoolSize为1；**

**\* keepAliveTime为0L；时间单位TimeUnit.MILLISECONDS；**

**\* workQueue为：new LinkedBlockingQueue<Runnable>() 无解阻塞队列**

**\* 通俗：创建只有一个线程的线程池，当该线程正繁忙时，对于新任务会进入阻塞队列中(无界的阻塞队列)**

**\* 适用：按顺序执行任务的场景**

**\***

**\*创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行**

**\*/**

**// ExecutorService pool = Executors.newSingleThreadExecutor();**

**// for (int i = 0; i < 10; i++) {**

**// final int ii = i;**

**// pool.execute(() -> out.println(Thread.currentThread().getName() + "=>" + ii));**

**// }**

**/\*\***

**\* 4.NewScheduledThreadPool:**

**\***

**\* 底层：创建ScheduledThreadPoolExecutor实例，该对象继承了ThreadPoolExecutor，**

**\* corePoolSize为传递来的参数，**

**\* maximumPoolSize为Integer.MAX\_VALUE；**

**\* keepAliveTime为0；时间单位TimeUnit.NANOSECONDS；**

**\* workQueue为：new DelayedWorkQueue() 一个按超时时间升序排序的队列**

**\* 通俗：创建一个固定大小的线程池，线程池内线程存活时间无限制，线程池可以支持定时及周期性任务执行，**

**\* 如果所有线程均处于繁忙状态，对于新任务会进入DelayedWorkQueue队列中，这是一种按照超时时间排序的队列结构**

**\* 适用：执行周期性任务**

**\***

**\* 创建一个定长线程池，支持定时及周期性任务执行。延迟执行**

**\*/**

**ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);**

**Runnable r1 = () -> out.println("线程名称：" + Thread.currentThread().getName() + "，执行:3秒后执行");**

**scheduledThreadPool.schedule(r1, 3, TimeUnit.SECONDS);**

**Runnable r2 = () -> out.println("线程名称：" + Thread.currentThread().getName() + "，执行:延迟2秒后每3秒执行一次");**

**scheduledThreadPool.scheduleAtFixedRate(r2, 2, 3, TimeUnit.SECONDS);**

**Runnable r3 = () -> out.println("线程名称：" + Thread.currentThread().getName() + "，执行:普通任务");**

**for (int i = 0; i < 5; i++) {**

**scheduledThreadPool.execute(r3);**

**}**

**}**

**}**

Springboot 线程池项目

Pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.adam.thread</groupId>

<artifactId>demo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>demo</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>1.8</java.version>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<spring-boot.version>2.1.17.RELEASE</spring-boot.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>

<version>${spring-boot.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<mainClass>com.adam.thread.DemoApplication</mainClass>

</configuration>

<executions>

<execution>

<id>repackage</id>

<goals>

<goal>repackage</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

package com.adam.thread;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.scheduling.annotation.EnableAsync;  
  
@EnableAsync  
@SpringBootApplication  
public class DemoApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(DemoApplication.class, args);  
 }  
  
}

package com.adam.thread.config.dto;  
  
import lombok.Data;  
import org.springframework.boot.context.properties.ConfigurationProperties;  
import org.springframework.stereotype.Component;  
  
*/\*\*  
 \* 线程池配置属性类  
 \*  
 \** ***@author*** *YuXD  
 \*/*@Data  
@Component  
@ConfigurationProperties(prefix = "task.pool")  
public class TaskThreadPoolConfig {  
  
 */\*\*  
 \* 核心线程数  
 \*/* private int corePoolSize;  
  
 */\*\*  
 \* 最大线程数  
 \*/* private int maxPoolSize;  
  
 */\*\*  
 \* 线程空闲时间  
 \*/* private int keepAliveSeconds;  
  
 */\*\*  
 \* 任务队列容量（阻塞队列）  
 \*/* private int queueCapacity;  
}

package com.adam.thread.config;  
  
import com.adam.thread.config.dto.TaskThreadPoolConfig;  
import org.springframework.beans.factory.annotation.Autowired;  
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.Executor;  
import java.util.concurrent.ThreadPoolExecutor;  
  
*/\*\*  
 \* 自定义下城  
 \*  
 \** ***@author*** *： YuXD  
 \*/*@Configuration  
@EnableAsync  
public class CustomizeThreadPoolConfig {  
  
 @Autowired  
 private TaskThreadPoolConfig config;  
  
 @Bean("customizeThreadPool")  
 public Executor doConfigCustomizeThreadPool() {  
 ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();  
 //核心线程池大小  
 executor.setCorePoolSize(config.getCorePoolSize());  
 //最大线程数  
 executor.setMaxPoolSize(config.getMaxPoolSize());  
 //队列容量  
 executor.setQueueCapacity(config.getQueueCapacity());  
 //活跃时间  
 executor.setKeepAliveSeconds(config.getKeepAliveSeconds());  
 //线程名字前缀  
 executor.setThreadNamePrefix("customize-thread-");  
 /\*  
 当poolSize已达到maxPoolSize，如何处理新任务（是拒绝还是交由其它线程处理）  
 CallerRunsPolicy：不在新线程中执行任务，而是由调用者所在的线程来执行  
 \*/  
 executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());  
 executor.initialize();  
 return executor;  
 }  
  
}

package com.adam.thread.config;  
  
import com.adam.thread.config.dto.TaskThreadPoolConfig;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.aop.interceptor.AsyncUncaughtExceptionHandler;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.scheduling.annotation.AsyncConfigurer;  
import org.springframework.scheduling.annotation.EnableAsync;  
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;  
  
import java.util.concurrent.Executor;  
import java.util.concurrent.ThreadPoolExecutor;  
  
*/\*\*  
 \* 重写默认线程池配置  
 \*  
 \** ***@author*** *YuXD  
 \*/*@Slf4j  
@Configuration  
@EnableAsync  
public class OverrideDefaultThreadPoolConfig implements AsyncConfigurer {  
  
 @Autowired  
 private TaskThreadPoolConfig config;  
  
 @Override  
 public Executor getAsyncExecutor() {  
 ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();  
 //核心线程池大小  
 executor.setCorePoolSize(config.getCorePoolSize());  
 //最大线程数  
 executor.setMaxPoolSize(config.getMaxPoolSize());  
 //队列容量  
 executor.setQueueCapacity(config.getQueueCapacity());  
 //活跃时间  
 executor.setKeepAliveSeconds(config.getKeepAliveSeconds());  
 //线程名字前缀  
 executor.setThreadNamePrefix("default-thread-");  
 /\*  
 当poolSize已达到maxPoolSize，如何处理新任务（是拒绝还是交由其它线程处理）  
 CallerRunsPolicy：不在新线程中执行任务，而是由调用者所在的线程来执行  
 \*/  
 executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());  
 executor.initialize();  
 return executor;  
 }  
  
 */\*\*  
 \* 异步任务中异常处理  
 \*  
 \** ***@return*** *\*/* @Override  
 public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {  
 return (ex, method, params) -> {  
 *log*.error("==========================" + ex.getMessage() + "=======================", ex);  
 *log*.error("exception method:" + method.getName());  
 };  
 }  
}

package com.adam.thread.controller;  
  
import com.adam.thread.service.IStatusAnalyseService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.util.List;  
  
@RestController  
public class ThreadDemoController {  
  
 @Autowired  
 private List<IStatusAnalyseService> statusAnalyseServiceList;  
  
  
 @GetMapping("/sayHelloAsync")  
 public String sayHelloAsync() {  
 for (IStatusAnalyseService statusAnalyseService : statusAnalyseServiceList) {  
 // 采用自定义线程池  
 statusAnalyseService.doStatusAnalyseHandle(null, null);  
 // 采用默认线程池  
 statusAnalyseService.doStatusAnalyseHandle(null);  
 }  
 return "Hello, Async!";  
 }  
}

package com.adam.thread.service.impl;  
  
import com.adam.thread.service.IStatusAnalyseService;  
import org.springframework.scheduling.annotation.Async;  
  
import java.util.Random;  
  
*/\*\*  
 \* 抽象的设备数据分析Manager  
 \*  
 \** ***@author*** *YuXD  
 \** ***@since*** *2020-06-18 22:47  
 \*/*public abstract class AbstractDeviceDataAnalyseManager implements IStatusAnalyseService {  
  
 @Async("customizeThreadPool")  
 @Override  
 public void doStatusAnalyseHandle(String start, String end) {  
 int sleepSeconds = new Random().nextInt(3) + 1;  
 try {  
 Thread.*sleep*(sleepSeconds \* 1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println(getDataType() + "在自定义线程" + Thread.*currentThread*().getName() + "执行了" + sleepSeconds + "秒");  
 }  
  
 @Async  
 @Override  
 public void doStatusAnalyseHandle(String end) {  
 int sleepSeconds = new Random().nextInt(3) + 1;  
 try {  
 Thread.*sleep*(sleepSeconds \* 1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println(getDataType() + "在默认线程" + Thread.*currentThread*().getName() + "执行了" + sleepSeconds + "秒");  
 }  
}

package com.adam.thread.service.impl;  
  
import org.springframework.stereotype.Service;  
  
*/\*\*  
 \** ***@description:*** *机加设备数据分析Service实现类  
 \** ***@author:*** *YuXD  
 \** ***@create:*** *2021-03-15 20:17  
 \*\*/*@Service("JJ")  
public class JJDeviceDataAnalyseManager extends AbstractDeviceDataAnalyseManager {  
  
 @Override  
 public String getDataType() {  
 return "机加";  
 }  
  
}

package com.adam.thread.service.impl;  
  
import org.springframework.stereotype.Service;  
  
*/\*\*  
 \** ***@description:*** *铸造设备数据分析Service实现类  
 \** ***@author:*** *YuXD  
 \** ***@create:*** *2020-06-18 22:56  
 \*\*/*@Service("ZHU")  
public class ZHUDeviceDataAnalyseManager extends AbstractDeviceDataAnalyseManager {  
  
 @Override  
 public String getDataType() {  
 return "铸造";  
 }  
  
}

package com.adam.thread.service.impl;  
  
import org.springframework.stereotype.Service;  
  
*/\*\*  
 \** ***@description:*** *总装设备数据分析Service实现类  
 \** ***@author:*** *YuXD  
 \** ***@create:*** *2020-06-18 22:56  
 \*\*/*@Service("ZZ")  
public class ZZDeviceDataAnalyseManager extends AbstractDeviceDataAnalyseManager {  
  
 @Override  
 public String getDataType() {  
 return "总装";  
 }  
  
}

package com.adam.thread.service;  
  
*/\*\*  
 \* 参数分析基础Service,所有需要进行参数分析的都需要实现该接口  
 \*  
 \** ***@author*** *YuXD  
 \*/*public interface IStatusAnalyseService {  
  
 */\*\*  
 \* 设备状态解析处理  
 \*  
 \** ***@param*** *start 开始时间  
 \** ***@param*** *end 截止时间  
 \*/* void doStatusAnalyseHandle(String start, String end);  
  
 */\*\*  
 \* 设备状态解析处理  
 \*  
 \** ***@param*** *end 截止时间  
 \*/* void doStatusAnalyseHandle(String end);  
  
 */\*\*  
 \* 获取数据类别  
 \*  
 \** ***@return*** *\*/* String getDataType();  
  
}

application.yml

task:  
 pool:  
 corePoolSize: 10  
 maxPoolSize: 20  
 keepAliveSeconds: 300  
 queueCapacity: 50