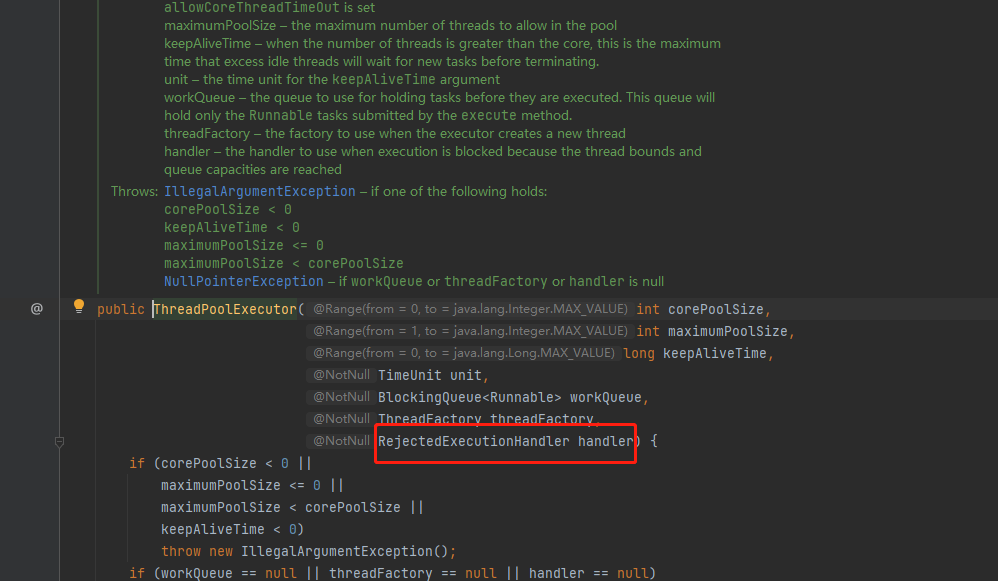
**一、了解线程池**



（重要）最后一个参数 有以下决绝策略

构造方法中的字段含义如下

corePoolSize：核心线程数量，当有新任务在execute()方法提交时，会执行以下判断：

如果运行的线程少于 corePoolSize，则创建新线程来处理任务，即使线程池中的其他线程是空闲的；

如果线程池中的线程数量大于等于 corePoolSize 且小于 maximumPoolSize，则只有当workQueue满时才创建新的线程去处理任务；

如果设置的corePoolSize 和 maximumPoolSize相同，则创建的线程池的大小是固定的，这时如果有新任务提交，若workQueue未满，则将请求放入workQueue中，等待有空闲的线程去从workQueue中取任务并处理；

如果运行的线程数量大于等于maximumPoolSize，这时如果workQueue已经满了，则通过handler所指定的策略来处理任务

所以，任务提交时，判断的顺序为 corePoolSize –> workQueue –> maximumPoolSize。

maximumPoolSize：最大线程数量

workQueue：等待队列，当任务提交时，如果线程池中的线程数量大于等于corePoolSize的时候，把该任务封装成一个Worker对象放入等待队列；

keepAliveTime：线程池维护线程所允许的空闲时间。当线程池中的线程数量大于corePoolSize的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了keepAliveTime；

threadFactory：它是ThreadFactory类型的变量，用来创建新线程。默认使用Executors.defaultThreadFactory() 来创建线程。使用默认的ThreadFactory来创建线程时，会使新创建的线程具有相同的NORM\_PRIORITY优先级并且是非守护线程，同时也设置了线程的名称。

handler：它是RejectedExecutionHandler类型的变量，表示线程池的饱和策略。如果阻塞队列满了并且没有空闲的线程，这时如果继续提交任务，就需要采取一种策略处理该任务。线程池提供了4种策略：

1 AbortPolicy：直接抛出异常，这是默认策略；

2CallerRunsPolicy：用调用者所在的线程来执行任务；

3DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；

4DiscardPolicy：直接丢弃任务；

简单来说，在执行execute()方法时如果状态一直是RUNNING时，的执行过程如下：

1如果workerCount < corePoolSize，则创建并启动一个线程来执行新提交的任务；

2如果workerCount >= corePoolSize，且线程池内的阻塞队列未满，则将任务添加到该阻塞队列中；

3如果workerCount >= corePoolSize && workerCount < maximumPoolSize，且线程池内的阻塞队列已满，则创建并启动一个线程来执行新提交的任务；

4如果workerCount >= maximumPoolSize，并且线程池内的阻塞队列已满, 则根据拒绝策略来处理该任务, 默认的处理方式是直接抛异常。

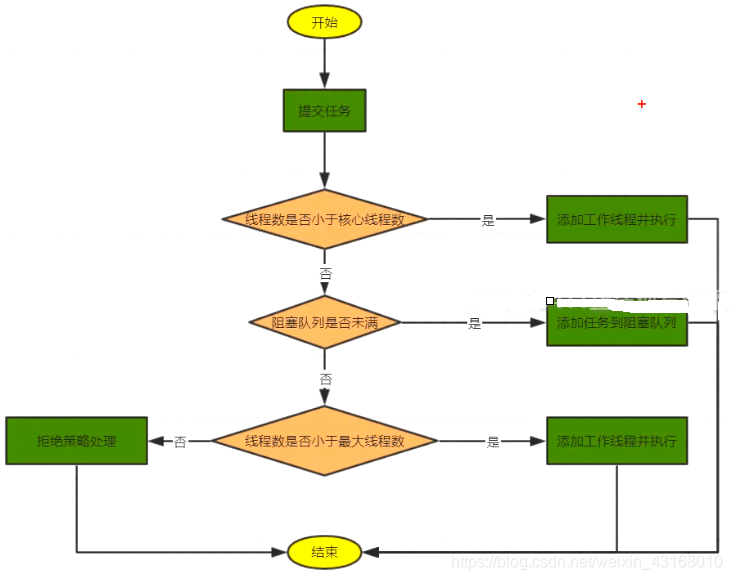
处理流程

1.查看核心线程池是否已满，不满就创建一条线程执行任务，否则执行第二步。

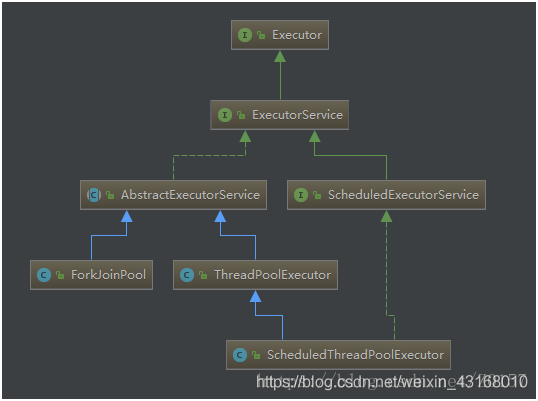
2.查看任务队列是否已满，不满就将任务存储在任务队列中，否则执行第三步。

3.查看线程池是否已满，即就是是否达到最大线程池数，不满就创建一条线程执行任务，否则就按照策略处理无法执行的任务。

流程图如下



JDK线程接口层级



一、线程池接口：ExecutorService为线程池接口，提供了线程池生命周期方法，继承自Executor接口，ThreadPoolExecutor为线程池实现类，提供了线程池的维护操作等相关方法，继承自AbstractExecutorService，AbstractExecutorService实现了ExecutorService接口。

二、线程池的体系结构：

java.util.concurrent.Executor 负责线程的使用和调度的根接口

|--ExecutorService 子接口： 线程池的主要接口

|--ThreadPoolExecutor 线程池的实现类

|--ScheduledExceutorService 子接口： 负责线程的调度

|--ScheduledThreadPoolExecutor : 继承ThreadPoolExecutor，实现了ScheduledExecutorService

**二、JDK内置现成工具类Executors**

工具类 ： Executors

Executors为线程迟工具类，相当于一个工厂类，用来创建合适的线程池，返回ExecutorService类型的线程池。有人如下方法。

ExecutorService newFixedThreadPool() : 创建固定大小的线程池

ExecutorService newCachedThreadPool() : 缓存线程池，线程池的数量不固定，可以根据需求自动的更改数量。

ExecutorService newSingleThreadExecutor() : 创建单个线程池。 线程池中只有一个线程

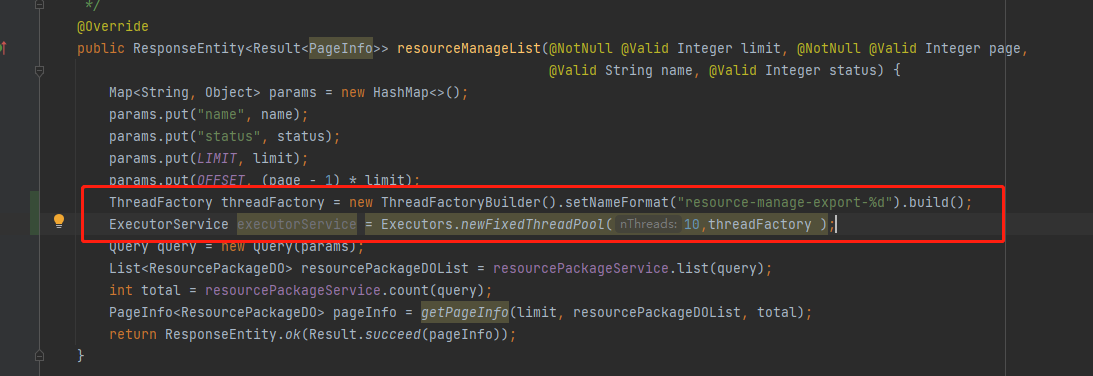
ScheduledExecutorService newScheduledThreadPool() : 创建固定大小的线程，可以延迟或定时的执行任务

其中AbstractExecutorService是他的抽象父类，继承自ExecutorService,ExecutorService 接口扩展Executor接口，增加了生命周期方法。

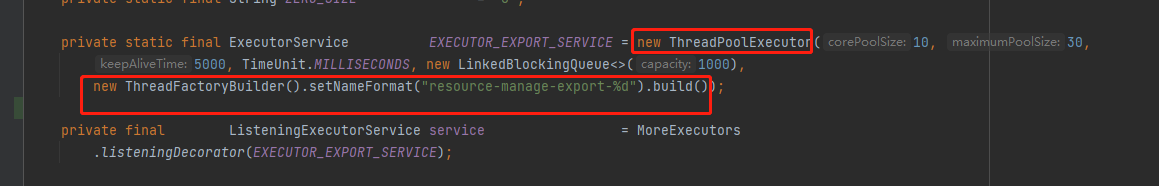
实际应用中我一般都比较喜欢使用Exectuors工厂类来创建线程池，里面有五个方法，分别创建不同的线程池，如上，创建一个制定大小的线程池，Exectuors工厂实际上就是调用的ExectuorPoolService的构造方法，传入默认参数。

**三、举例创建方式 new实现类 /工具类创建**

Executors.*newFixedThreadPool*



new ThreadPoolExecutor



**四、回调函数使用 两种方式 1.谷歌包封装调用jdk线程 2.spring配置ThreadPoolTaskExecutor类**

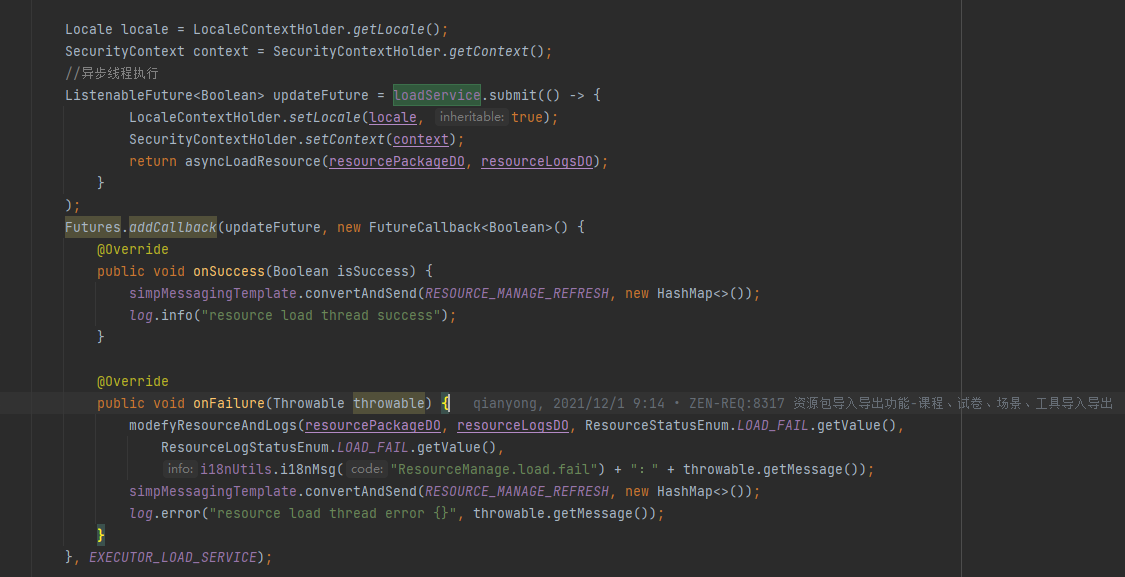
**1.全程导入谷歌封装包和JDK 自带Executor线程接口**

private static final ExecutorService EXECUTOR\_LOAD\_SERVICE = new ThreadPoolExecutor(10, 30,

5000, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<>(1000),

new ThreadFactoryBuilder().setNameFormat("resource-manage-load-%d").build());

private final ListeningExecutorService loadService = MoreExecutors.listeningDecorator(EXECUTOR\_LOAD\_SERVICE);



Locale locale = LocaleContextHolder.getLocale();

SecurityContext context = SecurityContextHolder.getContext();

//异步线程执行

ListenableFuture<Boolean> updateFuture = loadService.submit(() -> {

LocaleContextHolder.setLocale(locale, true);

SecurityContextHolder.setContext(context);

return asyncLoadResource(resourcePackageDO, resourceLogsDO);

}

);

Futures.addCallback(updateFuture, new FutureCallback<Boolean>() {

@Override

public void onSuccess(Boolean isSuccess) {

simpMessagingTemplate.convertAndSend(RESOURCE\_MANAGE\_REFRESH, new HashMap<>());

log.info("resource load thread success");

}

@Override

public void onFailure(Throwable throwable) {

modefyResourceAndLogs(resourcePackageDO, resourceLogsDO, ResourceStatusEnum.LOAD\_FAIL.getValue(),

ResourceLogStatusEnum.LOAD\_FAIL.getValue(),

i18nUtils.i18nMsg("ResourceManage.load.fail") + "：" + throwable.getMessage());

simpMessagingTemplate.convertAndSend(RESOURCE\_MANAGE\_REFRESH, new HashMap<>());

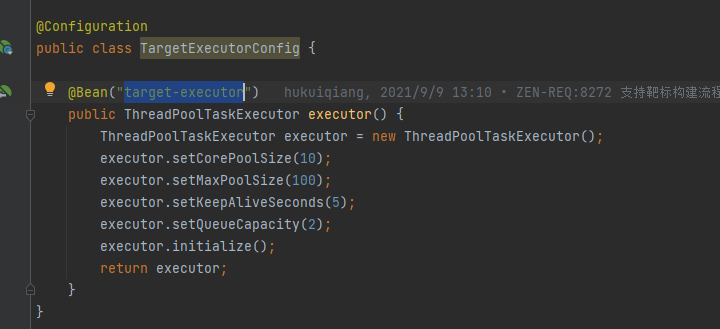
log.error("resource load thread error {}", throwable.getMessage());

}

}, EXECUTOR\_LOAD\_SERVICE);

============================================================================

**2.通过spring自带的ThreadPoolTaskExecutor类配置后回调**



@Configuration

public class TargetExecutorConfig {

@Bean("target-executor")

public ThreadPoolTaskExecutor executor() {

ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();

executor.setCorePoolSize(10);

executor.setMaxPoolSize(100);

executor.setKeepAliveSeconds(5);

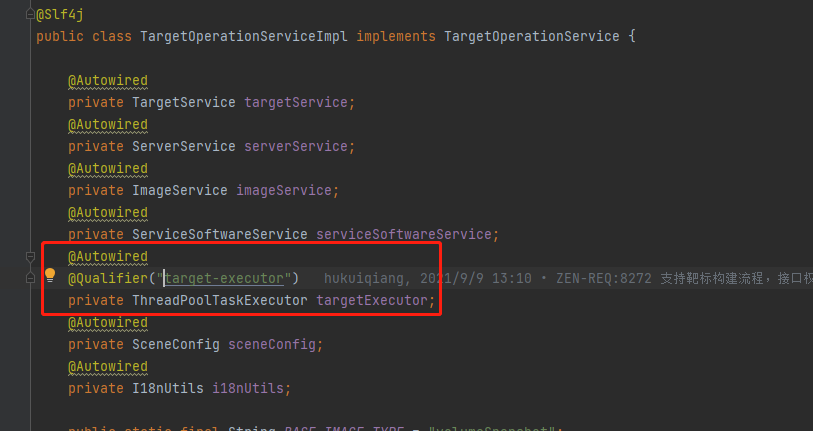
executor.setQueueCapacity(2);

executor.initialize();

return executor;

}

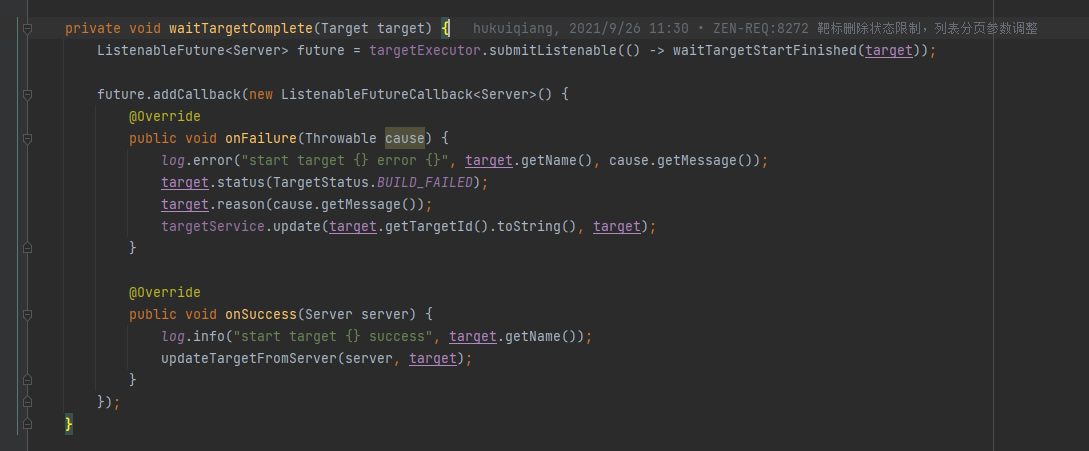
}



@Autowired

@Qualifier("target-executor")

private ThreadPoolTaskExecutor targetExecutor;



private void waitTargetComplete(Target target) {

ListenableFuture<Server> future = targetExecutor.submitListenable(() -> waitTargetStartFinished(target));

future.addCallback(new ListenableFutureCallback<Server>() {

@Override

public void onFailure(Throwable cause) {

log.error("start target {} error {}", target.getName(), cause.getMessage());

target.status(TargetStatus.BUILD\_FAILED);

target.reason(cause.getMessage());

targetService.update(target.getTargetId().toString(), target);

}

@Override

public void onSuccess(Server server) {

log.info("start target {} success", target.getName());

updateTargetFromServer(server, target);

}

});

}