> 【有道云笔记】线程池参数动态化
> https://note.youdao.com/s/FBkx8LOR

线程池在业务中的实践

业务背景

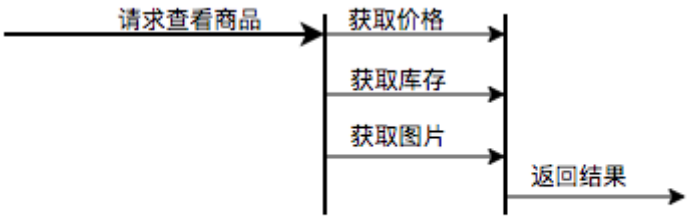
在当今的互联网业界，为了最大程度利用CPU的多核性能，并行运算的能力是不可或缺的。通过线程池管理线程获取并发性是一个非常基础的操作，让我们来看两个典型的使用线程池获取并发性的场景。

场景1：快速响应用户请求

描述：用户发起的实时请求，服务追求响应时间。比如说用户要查看一个商品的信息，那么我们需要将商品维度的一系列信息如商品的价格、优惠、库存、图片等等聚合起来，展示给用户。

分析：从用户体验角度看，这个结果响应的越快越好，如果一个页面半天都刷不出，用户可能就放弃查看这个商品了。而面向用户的功能聚合通常非常复杂，伴随着调用与调用之间的级联、多级级联等情况，业务开发同学往往会选择使用线程池这种简单的方式，将调用封装成任务并行的执行，缩短总体响应时间。另外，使用线程池也是有考量的，这种场景最重要的就是获取最大的响应速度去满足用户，所以应该不设置队列去缓冲并发任务，调高corePoolSize和maxPoolSize去尽可能创造多的线程快速执行任务。

串行执行情况

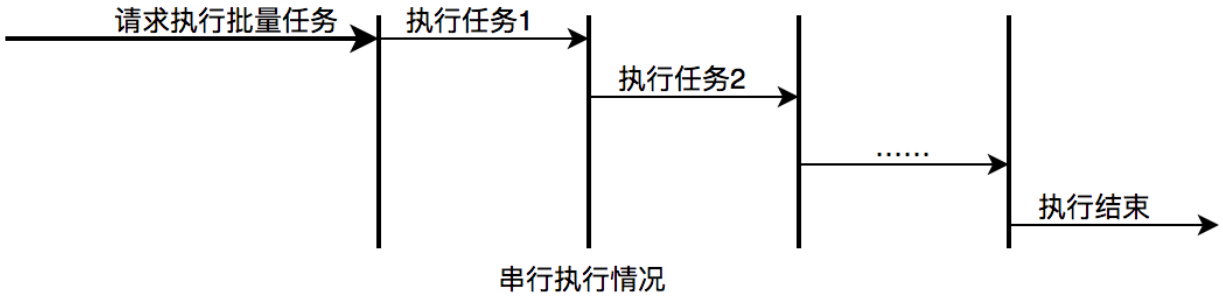


并行执行情况

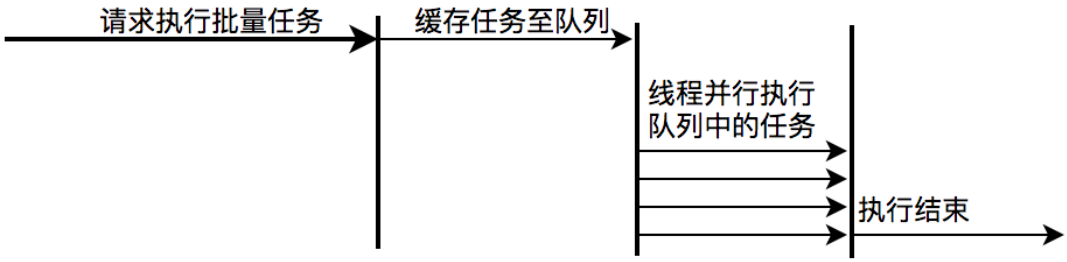
场景2：快速处理批量任务

描述：离线的大量计算任务，需要快速执行。比如说，统计某个报表，需要计算出全国各个门店中有哪些商品有某种属性，用于后续营销策略的分析，那么我们需要查询全国所有门店中的所有商品，并且记录具有某属性的商品，然后快速生成报表。

分析：这种场景需要执行大量的任务，我们也会希望任务执行的越快越好。这种情况下，也应该使用多线程策略，并行计算。但与响应速度优先的场景区别在于，这类场景任务量巨大，并不需要瞬时的完成，而是关注如何使用有限的资源，尽可能在单位时间内处理更多的任务，也就是吞吐量优先的问题。所以应该设置队列去缓冲并发任务，调整合适的corePoolSize去设置处理任务的线程数。在这里，设置的线程数过多可能还会引发线程上下文切换频繁的问题，也会降低处理任务的速度，降低吞吐量。



串行执行情况



并行执行情况

实际问题及方案思考

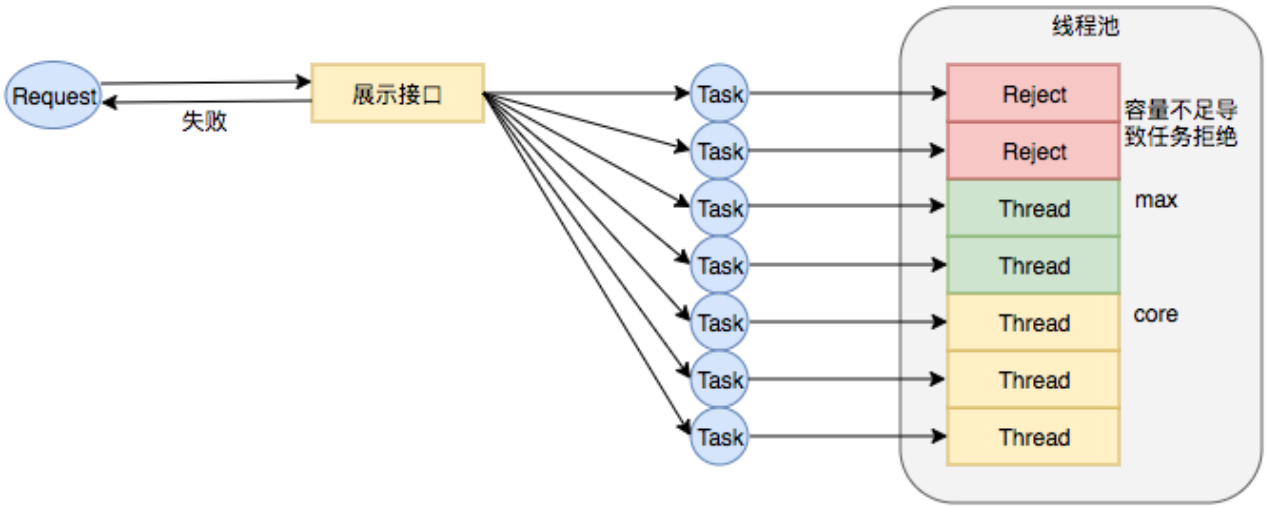
线程池使用面临的核心的问题在于：**线程池的参数并不好配置**。一方面线程池的运行机制不是很好理解，配置合理需要强依赖开发人员的个人经验和知识；另一方面，线程池执行的情况和任务类型相关性较大，IO密集型和CPU密集型的任务运行起来的情况差异非常大，这导致业界并没有一些成熟的经验策略帮助开发人员参考。

关于线程池配置不合理引发的故障，下面举一些例子：

Case1：XX页面展示接口大量调用降级：

事故描述：XX页面展示接口产生大量调用降级，数量级在几十到上百。

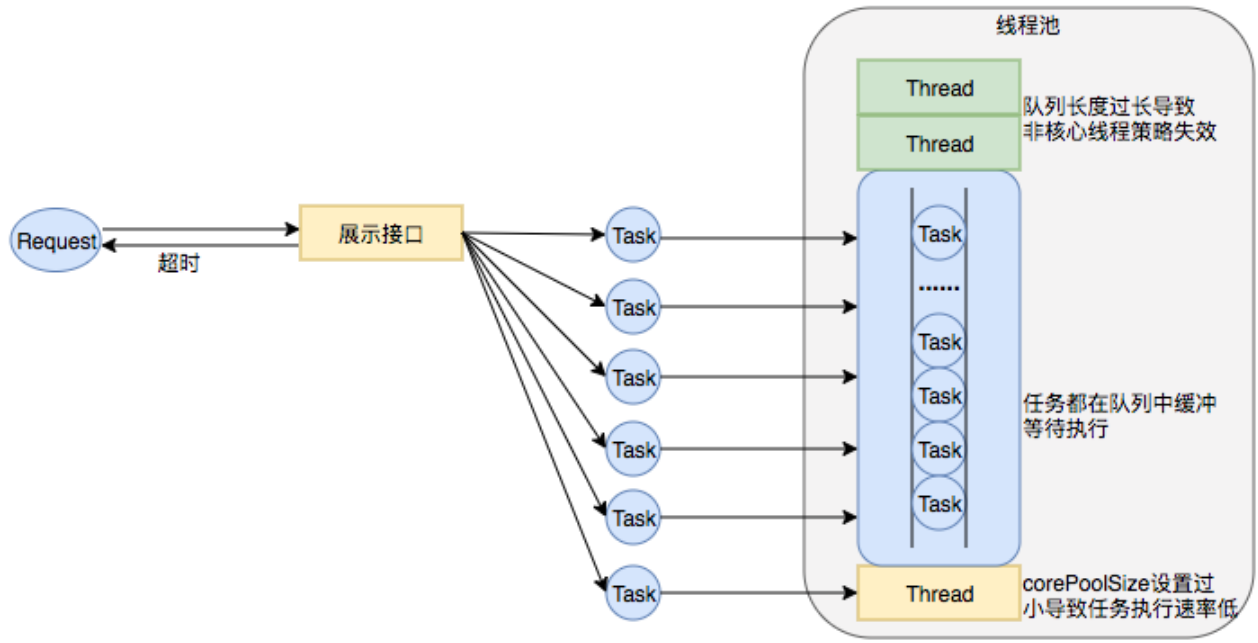
事故原因：该服务展示接口内部逻辑使用线程池做并行计算，由于没有预估好调用的流量，导致最大线程数设置偏小，大量抛出RejectedExecutionException，触发接口降级条件，示意图如下：



Case2：XX业务服务不可用S2级故障

事故描述：XX业务提供的服务执行时间过长，作为上游服务整体超时，大量下游服务调用失败。

事故原因：该服务处理请求内部逻辑使用线程池做资源隔离，由于队列设置过长，最大线程数设置失效，导致请求数量增加时，大量任务堆积在队列中，任务执行时间过长，最终导致下游服务的大量调用超时失败。示意图如下：

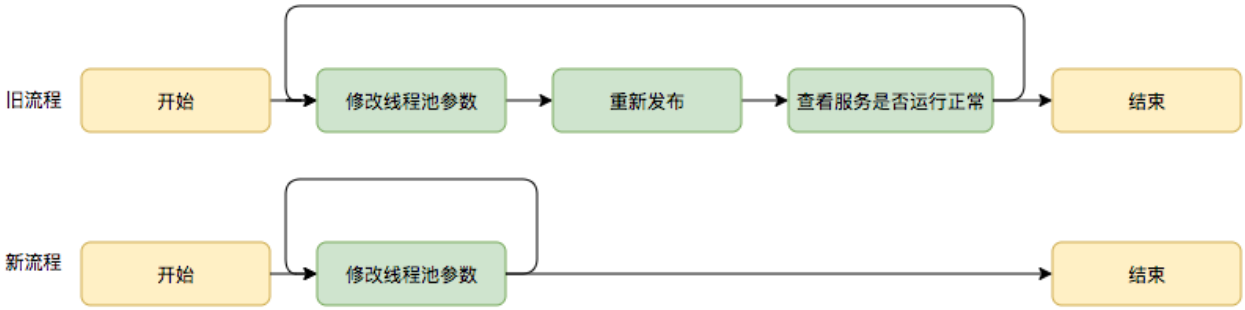


业务中要使用线程池，而使用不当又会导致故障，那么我们怎样才能更好地使用线程池呢？

线程池参数动态化

在日常项目开发中，我们通常会使用线程池来处理一些并发场景，来提高任务处理的效率。但是在使用过程中，无法准确地设置线程池参数，只能在运行过程中，不断去调整参数，然后重启服务。

那如何实现在不重启服务的前提下，动态调整线程池参数呢？

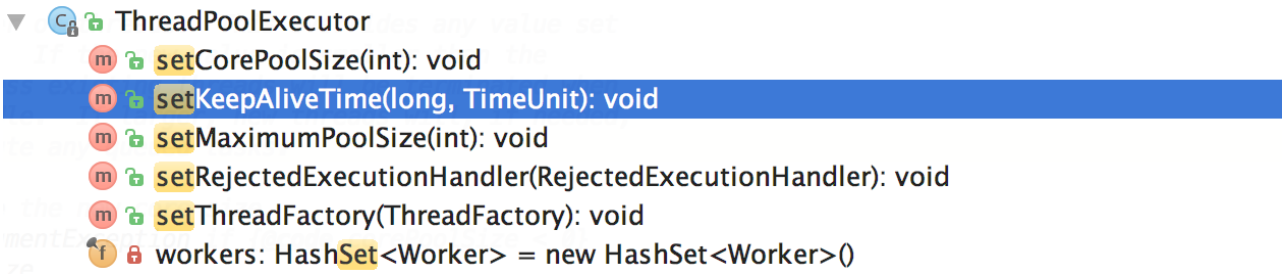


线程池可调整的参数

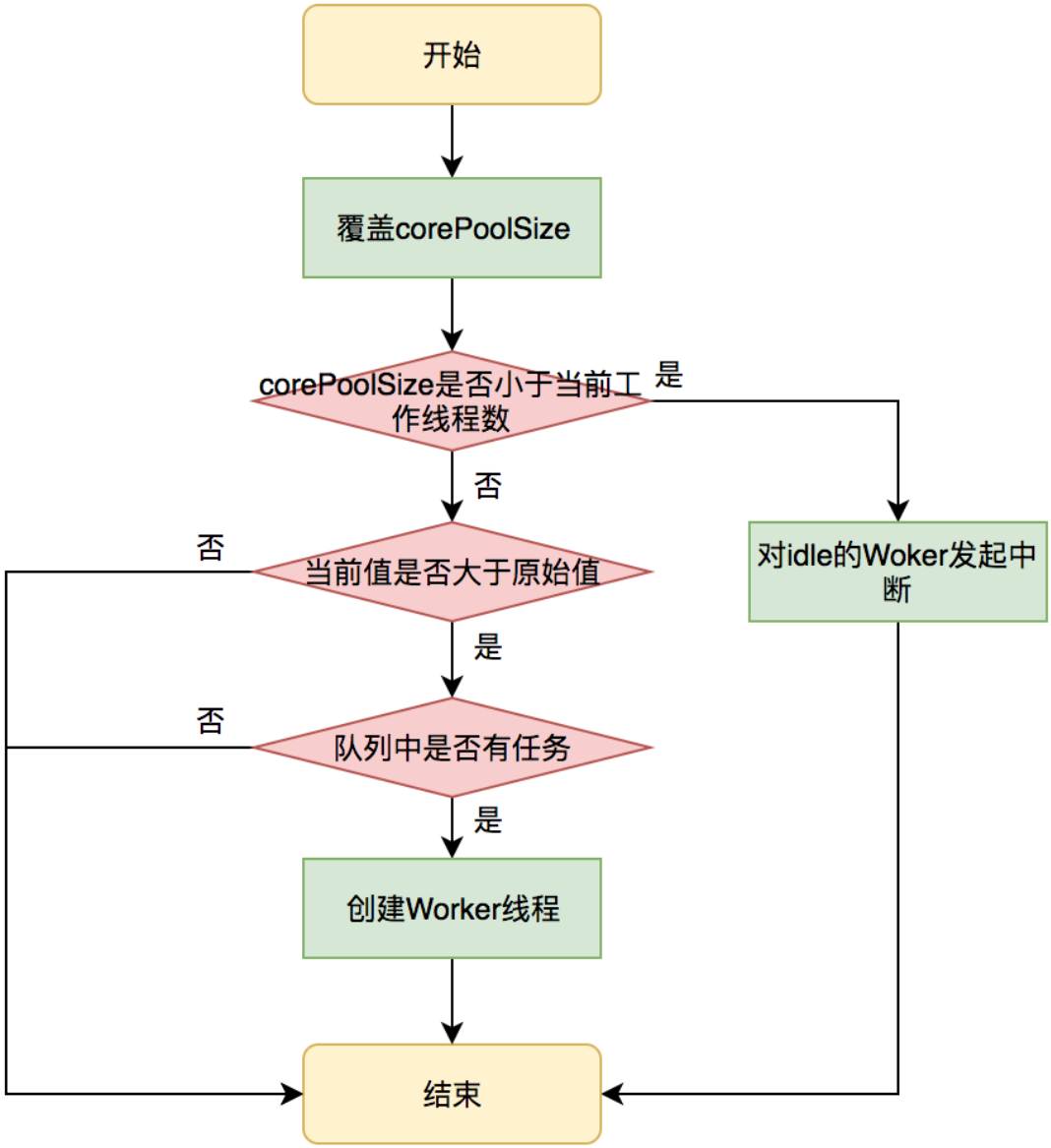
线程池构造参数有8个，但是最核心的是3个：corePoolSize、maximumPoolSize，workQueue，它们最大程度地决定了线程池的任务分配和线程分配策略。考虑到在实际应用中我们获取并发性的场景主要是两种：（1）并行执行子任务，提高响应速度。这种情况下，应该使用同步队列，没有什么任务应该被缓存下来，而是应该立即执行。（2）并行执行大批次任务，提升吞吐量。这种情况下，应该使用有界队列，使用队列去缓冲大批量的任务，队列容量必须声明，防止任务无限制堆积。所以线程池只需要提供这三个关键参数的配置，并且提供两种队列的选择，就可以满足绝大多数的业务需求。

实现思路

JDK原生线程池ThreadPoolExecutor提供了如下几个public的setter方法，如下图所示：



JDK允许线程池使用方通过ThreadPoolExecutor的实例来动态设置线程池的核心策略，以setCorePoolSize为方法例，在运行期线程池使用方调用此方法设置corePoolSize之后，线程池会直接覆盖原来的corePoolSize值，并且基于当前值和原始值的比较结果采取不同的处理策略。对于当前值小于当前工作线程数的情况，说明有多余的worker线程，此时会向当前idle的worker线程发起中断请求以实现回收，多余的worker在下次idel的时候也会被回收；对于当前值大于原始值且当前队列中有待执行任务，则线程池会创建新的worker线程来执行队列任务，setCorePoolSize具体流程如下：


开始
覆盖corePoolSize
corePoolSize是否小于当前工作线程数
是
否
对idle的Woker发起中断
否
当前值是否大于原始值
是
否
队列中是否有任务
是
创建Worker线程
结束

示例代码

```java
public class DynamicThreadPool {
    private ThreadPoolExecutor executor;

    public DynamicThreadPool(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {
        this.executor = new ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
        //executor.allowCoreThreadTimeOut(true);
    }

    public void adjustThreadPool(int newCorePoolSize, int newMaximumPoolSize) {
        this.executor.setCorePoolSize(newCorePoolSize);
        this.executor.setMaximumPoolSize(newMaximumPoolSize);
    }

    public void submitTask(Runnable task) {
        this.executor.execute(task);
    }

    public void print(){
        System.out.println("核心线程数: " + executor.getCorePoolSize()
                + " " +"最大线程数: " + executor.getMaximumPoolSize()
                +" " + "活跃线程数: " + executor.getActiveCount());
    }

    public void shutdown() {
        this.executor.shutdown();
    }

    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<Runnable> workQueue = new LinkedBlockingQueue<>(20);
        DynamicThreadPool dynamicThreadPool = new DynamicThreadPool(10, 10, 10, TimeUnit.SECONDS, workQueue);

        // 提交任务给线程池
        for (int i = 0; i < 30; i++) {
            dynamicThreadPool.submitTask(() -> {
                log.info(Thread.currentThread().getName() + "开始执行任务");
                try {
```

```
                    Thread.sleep(10000);
                    //log.info(Thread.currentThread().getName() + "任务执行完成");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }

        Thread.sleep(1000);
        System.out.println("======修改前=======");
        dynamicThreadPool.print();

        // 动态调整线程池参数
        dynamicThreadPool.adjustThreadPool(5, 8);

        System.out.println("======修改后=======");
        dynamicThreadPool.print();


        Thread.sleep(10000);
        dynamicThreadPool.print();

//        // 关闭线程池
//        dynamicThreadPool.shutdown();
    }
}
```

实现方案

基于Nacos配置中心动态调整线程池参数

我们可以借助Nacos的Listener，在Bean初始化的时候，开启Nacos配置变更的监听，在监听逻辑里面更新线程池参数。

```
nacosConfigManager.getConfigService().addListener("threadPool.yml",
nacosConfigProperties.getGroup(),
                new Listener() {
                    @Override
                    public Executor getExecutor() {
                        return null;
                    }
                    @Override
                    public void receiveConfigInfo(String configInfo) {

                    }
                });
```

核心代码

```java
@Configuration
@Data
public class MyDynamicThreadPool implements InitializingBean {

    @Value("${threadPool.corePoolSize}")
    private int corePoolSize;
    @Value("${threadPool.maxPoolSize}")
    private int maxPoolSize;
    @Value("${threadPool.queueCapacity}")
    private int queueCapacity;
    @Value("${threadPool.keepAliveSeconds}")
    private int keepAliveSeconds;

    private static ThreadPoolTaskExecutor threadPoolTaskExecutor;


    @Autowired
    private NacosConfigManager nacosConfigManager;

    @Autowired
    private NacosConfigProperties nacosConfigProperties;


    @Override
    public void afterPropertiesSet() throws Exception {
        threadPoolTaskExecutor = new ThreadPoolTaskExecutor();
        threadPoolTaskExecutor.setCorePoolSize(corePoolSize);
        threadPoolTaskExecutor.setMaxPoolSize(maxPoolSize);
        threadPoolTaskExecutor.setQueueCapacity(queueCapacity);
        threadPoolTaskExecutor.setKeepAliveSeconds(keepAliveSeconds);
        threadPoolTaskExecutor.setThreadNamePrefix( "Fox--");
        threadPoolTaskExecutor.setRejectedExecutionHandler(
                new RejectedExecutionHandler() {
                    @Override
                    public void rejectedExecution(Runnable r, ThreadPoolExecutor
executor) {
                        System.out.println("队列已满，丢弃任务");
                    }
```

```
                });
        threadPoolTaskExecutor.initialize();
        nacosConfigManager.getConfigService().addListener("threadPool.yml", nacosConfigProperties.getGroup(),
                new Listener() {
                    @Override
                    public Executor getExecutor() {
                        return null;
                    }
                    @Override
                    public void receiveConfigInfo(String configInfo) {
                        System.out.println("动态修改前-->");
                        print();
                        Yaml yaml = new Yaml();
                        InputStream inputStream = new ByteArrayInputStream(configInfo.getBytes());
                        Map<String, Object> dataMap = yaml.load(inputStream);
                        // 将Map转换为JSONObject
                        JSONObject pool =  new JSONObject(dataMap).getJSONObject("threadPool");
threadPoolTaskExecutor.setCorePoolSize(pool.getInteger("corePoolSize"));
threadPoolTaskExecutor.setMaxPoolSize(pool.getInteger("maxPoolSize"));
threadPoolTaskExecutor.setQueueCapacity(pool.getInteger("keepAliveSeconds"));
threadPoolTaskExecutor.setQueueCapacity(pool.getInteger("queueCapacity"));
                        System.out.println("动态修改后-->");
                        print();
                    }
                });

    }
    //执行任务
    public void execute(Runnable runnable){
        threadPoolTaskExecutor.execute(runnable);
    }

    public void print(){
        System.out.println("核心线程数: " + threadPoolTaskExecutor.getThreadPoolExecutor().getCorePoolSize()
```

```
                + " " +"最大线程数: " +
threadPoolTaskExecutor.getThreadPoolExecutor().getMaximumPoolSize()
                +" " + "阻塞队列数: " +
threadPoolTaskExecutor.getThreadPoolExecutor().getQueue().size()
                + "/" + queueCapacity
                +" " + "活跃线程数: " +
threadPoolTaskExecutor.getThreadPoolExecutor().getActiveCount());
    }
}
```

使用DynamicTp——基于配置中心的轻量级动态可监控线程池

DynamicTp 是一个基于配置中心实现的轻量级动态线程池管理工具，主要功能可以总结为动态调参、通知报警、运行监控、三方包线程池管理等几大类。

> 官网地址：https://dynamictp.cn/