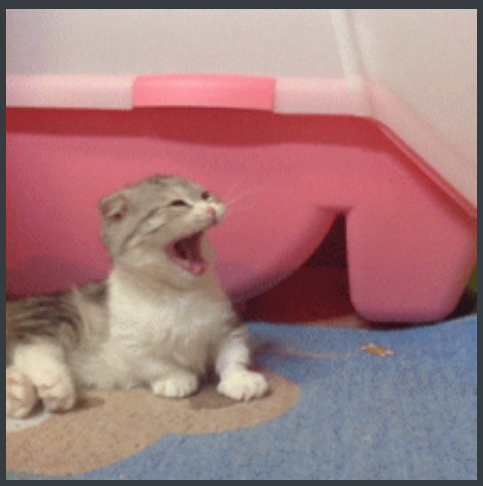
> 点赞再看，养成习惯，微信搜索【**敖丙**】关注这个互联网苟且偷生的工具人。
>
> 本文 **GitHub** https://github.com/JavaFamily 已收录，有一线大厂面试完整考点、资料以及我的系列文章。

背景

敖丙之前在工作中遇到一个问题，我定义了一个线程池来执行任务，但是程序执行结束后任务没有全部执行完，当时心态就差点崩了。



业务场景是这样的：由于统计业务需要，订单信息需要从主库中经过统计业务代码写入统计库（中间需要逻辑处理所以不能走binlog）。

由于代码质量及历史原因，目前的重新统计接口是单线程的，粗略算了算一共有100万条订单信息，每100条的处理大约是10秒，所以理论上处理完全部信息需要28个小时，这还不算因为 mysql 中 limit 分页导致的后期查询时间以及可能出现的内存溢出导致中止统计的情况。

基于上述的原因，以及最重要的一点：统计业务是根据订单所属的中心进行的，各个中心同时统计不会导致脏数据。

所以，我计划使用线程池，为每一个中心分配一条线程去执行统计业务。

业务实现

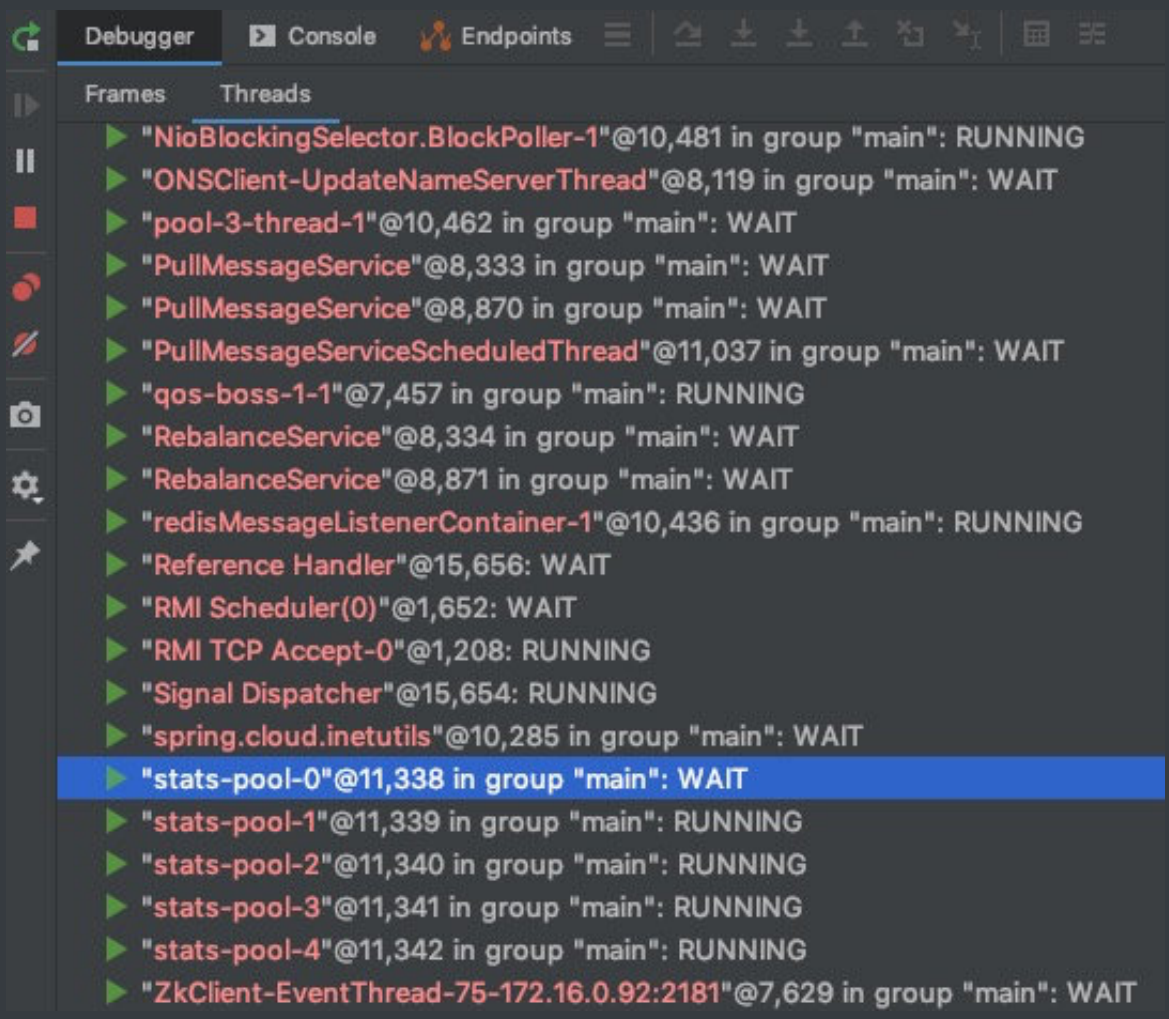
```
// 线程工厂，用于为线程池中的每条线程命名
ThreadFactory namedThreadFactory = new
ThreadFactoryBuilder() .setNameFormat("stats-pool-%d").build();

// 创建线程池，使用有界阻塞队列防止内存溢出
ExecutorService statsThreadPool = new ThreadPoolExecutor(5, 10,
                0L, TimeUnit.MILLISECONDS,
                new LinkedBlockingQueue<>(100), namedThreadFactory);
// 遍历所有中心，为每一个centerId提交一条任务到线程池
statsThreadPool.submit(new StatsJob(centerId));
```

在创建完线程池后，为每一个 centerId 提交一条任务到线程池，在我的预想中，由于线程池的核心线程数为5，最多5个中心同时进行统计业务，将大大缩短100万条数据的总统计时间，于是万分兴奋的我开始执行重新统计业务了。

问题

在跑了很久之后，当我查看统计进度时，我发现了一个十分诡异的问题（如下图）。

蓝框标出的这条线程是 WAIT 状态，表明这条线程是空闲状态，但是从日志中我看到这条线程并没有完成它的任务，因为这个中心的数据有10万条，但是日志显示它只跑到了一半，之后就再无关于此中心的日志了。



这是什么原因?

我当场就想到了三歪，肯定是三歪今天早上上班左脚先迈进公司的，导致代码水土不服，一定是这样，我去找他去。

调试及原因

咳咳三歪是开玩笑的，我们还是需要找到真实原因。

可以想到的是，这条线程因为某些原因被阻塞了，并且没有继续进行下去，但是日志又没有任何异常信息...

可能有经验的工程师已经知道了原因...

由于个人水平的线程，暂时没有找到原因的我只能放弃使用线程池，乖乖用单线程跑...

幸运的是，单线程跑的任务竟然抛错了（为什么要说幸运？），于是马上想到，之前那条 WAIT 状态的线程可能是因为同样的抛错所以被中断了，导致任务没有继续进行下去。

> 为什么说幸运？因为如果单线程的任务没有抛错的话，我可能很久都想不到是这个原因。

深入探究线程池的异常处理

工作上的问题到这里就找到原因了，之后的解决过程也十分简单，这里就不提了。

但是疑问又来了，为什么使用线程池的时候，线程因异常被中断却没有抛出任何信息呢？还有平时如果是在 main 函数里面的异常也会被抛出来，而不是像线程池这样被吞掉。

如果子线程抛出了异常，线程池会如何进行处理呢？

> 我提交任务到线程池的方式是：`threadPoolExecutor.submit(Runnbale task);` ，后面了解到使用 execute() 方式提交任务会把异常日志给打出来，这里研究一下为什么使用 submit 提交任务，在任务中的异常会被“吞掉”。

对于 submit() 形式提交的任务，我们直接看源码：

```
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
    // 被包装成 RunnableFuture 对象，然后准备添加到工作队列
    RunnableFuture<Void> ftask = newTaskFor(task, null);
    execute(ftask);
    return ftask;
}
```

它会被线程池包装成 RunnableFuture 对象，而最终它其实是一个 FutureTask 对象，在被添加到线程池的工作队列，然后调用 start() 方法后， FutureTask 对象的 run() 方法开始运行，即本任务开始执行。

```
public void run() {
    if (state != NEW ||
!UNSAFE.compareAndSwapObject(this,runnerOffset,null,
Thread.currentThread()))
        return;
    try {
        Callable<V> c = callable;
        if (c != null && state == NEW) {
            V result;
            boolean ran;
            try {
                result = c.call();
                ran = true;
            } catch (Throwable ex) {
                // 捕获子任务中的异常
                result = null;
                ran = false;
                setException(ex);
            }
            if (ran)
                set(result);
        }
    } finally {
        runner = null;
        int s = state;
        if (s >= INTERRUPTING)
            handlePossibleCancellationInterrupt(s);
    }
}
```

在 FutureTask 对象的 run() 方法中，该任务抛出的异常被捕获，然后在setException(ex); 方法中，抛出的异常会被放到 outcome 对象中，这个对象就是 submit() 方法会返回的 FutureTask 对象执行 get() 方法得到的结果。

但是在线程池中，并没有获取执行子线程的结果，所以异常也就没有被抛出来，即被“吞掉”了。

这就是线程池的 submit() 方法提交任务没有异常抛出的原因。

线程池自定义异常处理方法

在定义 ThreadFactory 的时候调用 `setUncaughtExceptionHandler` 方法，自定义异常处理方法。 例如：

```
    ThreadFactory namedThreadFactory = new ThreadFactoryBuilder()
                    .setNameFormat("judge-pool-%d")
                    .setUncaughtExceptionHandler((thread, throwable)->
    logger.error("ThreadPool {} got exception", thread,throwable))
                    .build();
```

这样，对于线程池中每条线程抛出的异常都会打下 error 日志，就不会看不到了。

后续

在修复了单个线程任务的异常之后，我继续使用线程池进行重新统计业务，终于跑完了，也终于完成了这个任务。

事后我也叫三歪以后进公司一定要先迈出右脚进来，不然对写代码的风水影响很大。

小结：丙这个事故也给大家一个警示，使用线程池时需要注意，子线程的异常，如果没有被捕获就会丢失，可能会导致后期根据日志调试时无法找到原因。

我是敖丙，一个在互联网苟且偷生的程序员。

你知道的越多，你不知道的越多，人才们的 **【三连】** 就是丙丙创作的最大动力，我们下期见！

注：如果本篇博客有任何错误和建议，欢迎人才们留言！

> 文章持续更新，可以微信搜索「 **敖丙** 」第一时间阅读，回复【**资料**】有我准备的一线大厂面试资料和简历模板，本文 **GitHub** https://github.com/JavaFamily 已经收录，有大厂面试完整考点，欢迎Star。


三太子教育
微信搜索