线程池运用不当的一次线上事故

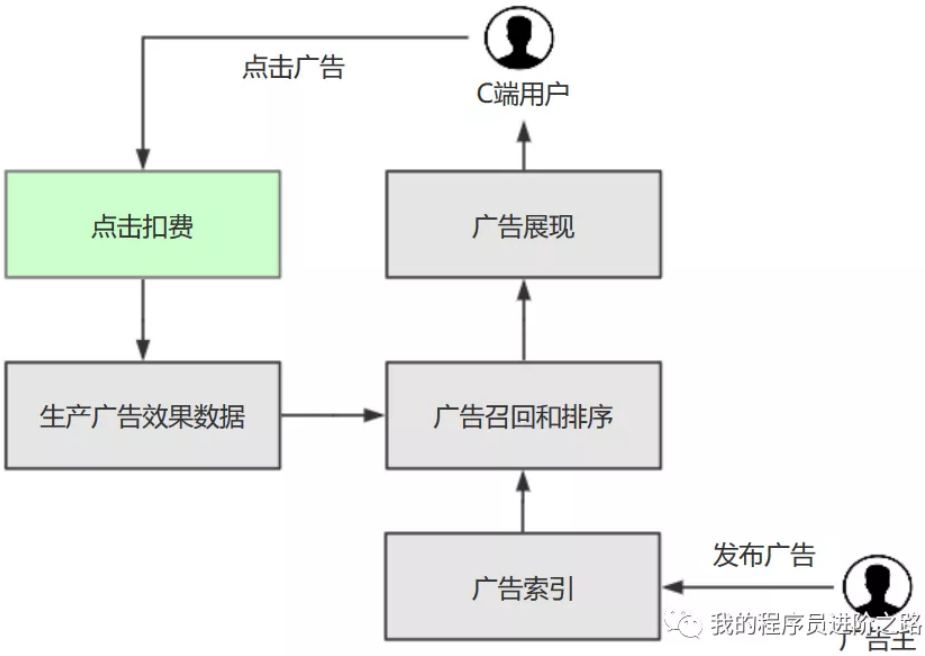
Rockets java 线程池 4月前

在高并发、异步化等场景，线程池的运用可以说无处不在。线程池从本质上来讲，即通过空间换取时间，因为线程的创建和销毁都是要消耗资源和时间的，对于大量使用线程的场景，使用池化管理可以延迟线程的销毁，大大提高单个线程的复用能力，进一步提升整体性能。

今天遇到了一个比较典型的线上问题，刚好和线程池有关，另外涉及到死锁、jstack命令的使用、JDK不同线程池的适合场景等知识点，同时整个调查思路可以借鉴，特此记录和分享一下。

业务背景描述

该线上问题发生在广告系统的核心扣费服务，首先简单交代下大致的业务流程，方便理解问题。



绿框部分即扣费服务在广告召回扣费流程中所处的位置，简单理解：当用户点击一个广告后，会从C端发起一次实时扣费请求(CPC，按点击扣费模式)，扣费服务则承接了该动作的核心业务逻辑：包括执行反作弊策略、创建扣费记录、click日志埋点等。

问题现象和业务影响

12月2号晚上11点左右，我们收到了一个线上告警通知：扣费服务的线程池任务队列大小远远超出了设定阈值，而且队列大小随着时间推移还在持续变大。详细告警内容如下：

相应的，我们的广告指标：点击数、收入等也出现了非常明显的下滑，几乎同时发出了业务告警通知。其中，点击数指标对应的曲线表现如下：

关于作者

Rockets

0 关注 36 粉丝 3 文章

热门文章

1 PerfMa社区上线一周年，... 小稳_PerfMa

2 ThreadLocalRandom 安全吗 技术小哥哥

3 讨论在 Linux Control Grou... 涤生

4 G1垃圾回收源码分析(一) 小蓝鲸

5 解决服务器进程退出问题 landon30

6 大量类加载器创建导致诡... CoderMeng

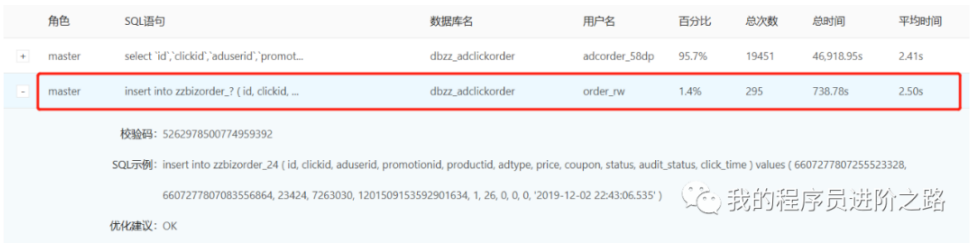
该线上故障发生在流量高峰期，持续了将近30分钟后才恢复正常。

问题调查和事故解决过程

下面详细说下整个事故的调查和分析过程。

第1步：收到线程池任务队列的告警后，我们第一时间查看了扣费服务各个维度的实时数据：包括服务调用量、超时量、错误日志、JVM监控，均未发现异常。
第2步：然后进一步排查了扣费服务依赖的存储资源（mysql、redis、mq），外部服务，发现了事故期间存在大量的数据库慢查询。

上述慢查询来自于事故期间一个刚上线的大数据抽取任务，从扣费服务的mysql数据库中大批量并发抽取数据到hive表。因为扣费流程也涉及到写mysql，猜测这个时候mysql的所有读写性能都受到了影响，果然进一步发现insert操作的耗时也远远大于正常时期。



第3步：我们猜测数据库慢查询影响了扣费流程的性能，从而造成了任务队列的积压，所以决定立马暂定大数据抽取任务。但是很奇怪：停止抽取任务后，数据库的insert性能恢复到正常水平了，但是阻塞队列大小仍然还在持续增大，告警并未消失。

第4步：考虑广告收入还在持续大幅度下跌，进一步分析代码需要比较长的时间，所以决定立即重启服务看看有没有效果。为了保留事故现场，我们保留了一台服务器未做重启，只是把这台机器从服务管理平台摘掉了，这样它不会接收到新的扣费请求。

果然重启服务的杀手锏很管用，各项业务指标都恢复正常了，告警也没有再出现。至此，整个线上故障得到解决，持续了大概30分钟。

问题根本原因的分析过程

下面再详细说下事故根本原因的分析过程。

第1步：第二天上班后，我们猜测那台保留了事故现场的服务器，队列中积压的任务应该都被线程池处理掉了，所以尝试把这台服务器再次挂载上去验证下我们的猜测，结果和预期完全相反，积压的任务仍然都在，而且随着新请求进来，系统告警立刻再次出现了，所以又马上把这台服务器摘了下来。

第2步：线程池积压的几千个任务，经过1个晚上都没被线程池处理掉，我们猜测应该存在死锁情况。所以打算通过jstack命令dump线程快照做下详细分析。

```
#找到扣费服务的进程号
$ jstack pid > /tmp/stack.txt

# 通过进程号dump线程快照，输出到文件中
$ jstack pid > /tmp/stack.txt
```

在jstack的日志文件中，立马发现了：用于扣费的业务线程池的所有线程都处于waiting状态，线程全部卡在了截图中红框部分对应的代码行上，这行代码调用了countDownLatch的await()方法，即等待计数器变为0后释放共享锁。

第3步：找到上述异常后，距离找到根本原因就很接近了，我们回到代码中继续调查，首先看了下业务代码中使用了newFixedThreadPool线程池，核心线程数设置为25。针对newFixedThreadPool，JDK文档的说明如下：

创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。如果在所有线程处于活跃状态时提交新任务，则在有可用线程之前，新任务将在队列中等待。

关于newFixedThreadPool，核心包括两点：
1、最大线程数 = 核心线程数，当所有核心线程都在处理任务时，新进来的任务会提交到任务队列中等待；

2、使用了无界队列：提交给线程池的任务队列是不限制大小的，如果任务被阻塞或者处理变慢，那么显然队列会越来越大。

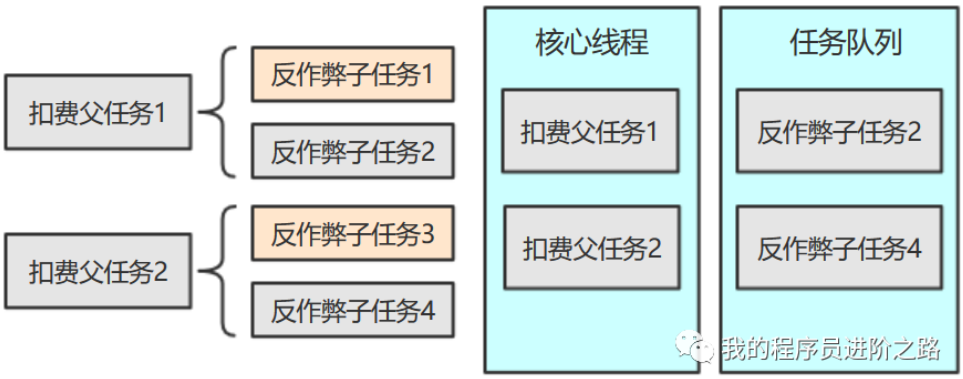
所以，进一步结论是：核心线程全部死锁，新进的任务不对涌入无界队列，导致任务队列不断增加。

第4步：到底是什么原因导致的死锁，我们再次回到jstack日志文件中提示的那行代码做进一步分析。下面是我简化过后的示例代码：

```
/**
 * 执行扣费任务
 */
public Result<Integer> executeDeduct(ChargeInputDTO chargeInput) {
  ChargeTask chargeTask = new ChargeTask(chargeInput);
  bizThreadPool.execute(() -> chargeTaskBll.execute(chargeTask ));
  return Result.success();
}

/*
 * 扣费任务的具体业务逻辑
 */
public class ChargeTaskBll implements Runnable {

  public void execute(ChargeTask chargeTask) {
     // 第一步：参数校验
     verifyInputParam(chargeTask);

     // 第二步：执行反作弊子任务
     executeUserSpam(SpamHelper.userConfigs);

     // 第三步：执行扣费
     handlePay(chargeTask);

     // 其他步骤：点击埋点等
     ...
  }
}

/**
 * 执行反作弊子任务
 */
public void executeUserSpam(List<SpamUserConfigDO> configs) {
  if (CollectionUtils.isEmpty(configs)) {
    return;
```

```
    }

    try {
      CountDownLatch latch = new CountDownLatch(configs.size());
      for (SpamUserConfigDO config : configs) {
        UserSpamTask task = new UserSpamTask(config,latch);
        bizThreadPool.execute(task);
      }
      latch.await();
    } catch (Exception ex) {
      logger.error("", ex);
    }
  }
```

通过上述代码，大家能否发现死锁是怎么发生的呢？根本原因在于：一次扣费行为属于父任务，同时它又包含了多次子任务：子任务用于并行执行反作弊策略，而父任务和子任务使用的是同一个业务线程池。当线程池中全部都是执行中的父任务时，并且所有父任务都存在子任务未执行完，这样就会发生死锁。下面通过1张图再来直观地看下死锁的情况：



假设核心线程数是2，目前正在执行扣费父任务1和2。另外，反作弊子任务1和3都执行完了，反作弊子任务2和4都积压在任务队列中等待被调度。因为反作弊子任务2和4没执行完，所以扣费父任务1和2都不可能执行完成，这样就发生了死锁，核心线程永远不可能释放，从而造成任务队列不断增大，直到程序OOM crash。

死锁原因清楚后，还有个疑问：上述代码在线上运行很长时间了，为什么现在才暴露出问题呢？另外跟数据库慢查询到底有没有直接关联呢？

暂时我们还没有复现证实，但是可以推断出：上述代码一定存在死锁的概率，尤其在高并发或者任务处理变慢的情况下，概率会大大增加。数据库慢查询应该就是导致此次事故出现的导火索。

解决方案

弄清楚根本原因后，最简单的解决方案就是：增加一个新的业务线程池，用来隔离父子任务，现有的线程池只用来处理扣费任务，新的线程池用来处理反作弊任务。这样就可以彻底避免死锁的情况了。

问题总结

回顾事故的解决过程以及扣费的技术方案，存在以下几点待继续优化：

1. 使用固定线程数的线程池存在OOM风险，在阿里巴巴Java开发手册中也明确指出，而且用的词是『不允许』使用Executors创建线程池。 而是通过ThreadPoolExecutor去创建，这样让写的同学能更加明确线程池的运行规则和核心参数设置，规避资源耗尽的风险。
2. 广告的扣费场景是一个异步过程，通过线程池或者MQ来实现异步化处理都是可选的方案。另外，极个别的点击请求丢失不扣费从业务上是允许的，但是大批量的请求丢弃不处理且没有补偿方案是不允许的。后续采用有界队列后，拒绝策略可以考虑发送MQ做重试处理。

请先登录，再评论

评论列表

手机用户700083 1月前

很赞，这个case碰巧前不久就遇到了。只不过是在测试阶段发现的，并没有影响到线上。

回复

贝壳 2月前

很赞👍

回复

spring_430960 3月前

精彩

回复

秋风画扇 4月前

父任务依赖子任务执行，且放在同一个线程池中执行本身就存在问题。

如果用户点击并发很高，很容易造成父任务占满资源池的现象；猜测系统并发不是太高，一般提交/生成的父子任务都能快速结束。

当insert变慢的时候，导致任务耗时增加，先前子任务尚未完成又提交了新的父任务占用了过多的线程，最终可用线程<子任务所需线程造成了这种尴尬的局面。

回复

CringKong 4月前

这个案例很有意思，有依赖关系的任务在这种情况下确实会出现一种逻辑上的死锁，以前阿里巴巴规范要求必须自己手动指定线程池参数，还不以为然，现在来看是很有必要的。

回复

黄金键盘 4月前

案例很有意思, 也很精彩 👍

回复

喵的沉思 4月前

请教楼主，文中的SQL监控是使用的开源的吗

回复

哇咔咔_601491 4月前

感谢分享，受益良多！

回复

金hb.Ryan 冷空氣駕到 4月前

慢查询导致线程响应慢，这个时候线程池可用线程少于反作弊的子任务的时候大概率就会导致任务的资源互相占用，且占用了了为数不多的可用线程，极端情况，扣费请求数量等于线程数量，所有线程被wait住，所以也可以理解成死锁吧

回复

残月@诗雨_393862 4月前

这不是死锁吧，就只是子任务没执行完，造成父任务也执行不完，应该检查子任务为什么没执行完吧

回复

肖波 4月前

回复 残月@诗雨_393862：

这就是死锁，子任务都在等待线程池资源，所以执行不完。

回复

< 1 2 >