Dubbo 线程池满异常应该是大多数 Dubbo 用户都遇到过的一个问题，本文以 Arthas 3.1.7 版本为例，介绍如何针对该异常进行诊断，主要使用到 dashboard/thread 两个指令。

# Dubbo 线程池满异常介绍

理解线程池满异常需要首先了解 Dubbo 线程模型，官方文档：http://dubbo.apache.org/zh-cn/docs/user/demos/thread-model.html。简单概括下 Dubbo 默认的线程模型：Dubbo 服务端每次接收到一个 Dubbo 请求，便交给一个线程池处理，该线程池默认有 200 个线程，如果 200 个线程都不处于空闲状态，则

注意：这个线程池指的是当前应用所有接口公用，所以不建议一个应用对外提供太多的dubbo接口，只要有一个接口出现阻塞问题，其他的接口都会受到影响。

客户端会报出如下异常：

Caused by: java.util.concurrent.ExecutionException: org.apache.dubbo.remoting.RemotingException: Server side(192.168.1.101,20880) threadpool is exhausted ...

服务端会打印 WARN 级别的日志：

[DUBBO] Thread pool is EXHAUSTED!

引发该异常的原因主要有以下几点：

* 客户端/服务端超时时间设置不合理，导致请求无限等待，耗尽了线程数
* 客户端请求量过大，服务端无法及时处理，耗尽了线程数
* 服务端由于 fullgc 等原因导致处理请求较慢，耗尽了线程数
* 服务端由于数据库、Redis、网络 IO 阻塞问题，耗尽了线程数

原因可能很多，但究其根本，都是因为业务上出了问题，导致 Dubbo 线程池资源耗尽了。所以出现该问题，首先要做的是：

排查业务异常

紧接着针对自己的业务场景对 Dubbo 进行调优：

* 调整接口的超时时间，比如500ms, 3000ms等
* 调整 Provider 端的 dubbo.provider.threads 参数大小，默认 200，可以适当提高。多大算合适？至少 700 不算大；不建议调的太小，容易出现上述问题
* 调整 Consumer 端的 dubbo.consumer.actives 参数，控制消费者调用的速率。这个实践中很少使用，仅仅一提
* 客户端限流
* 服务端扩容
* Dubbo 目前不支持给某个 service 单独配置一个隔离的线程池，用于保护服务，可能在以后的版本中会增加这个特性

另外，不止 Dubbo 如此设计线程模型，绝大多数服务治理框架、 HTTP 服务器都有业务线程池的概念，所以理论上它们都会有线程池满异常的可能，解决方案也类似。

那竟然问题都解释清楚了，我们还需要排查什么呢？一般在线上，有很多运行中的服务，这些服务都是共享一个 Dubbo 服务端线程池，可能因为某个服务的问题，导致整个应用被拖垮，所以需要排查是不是集中出现在某个服务上，再针对排查这个服务的业务逻辑；需要定位到线程堆栈，揪出导致线程池满的元凶。

定位该问题，我的习惯一般是使用 Arthas 的 dashboard 和 thread 命令，而在介绍这两个命令之前，我们先人为的构造一个 Dubbo 线程池满异常的例子。

# 复现 Dubbo 线程池满异常

## 配置服务端线程池大小

dubbo.protocol.threads=10

默认大小是 200，不利于重现该异常

## 模拟服务端阻塞

@Service(version = "1.0.0")

public class DemoServiceImpl implements DemoService {

@Override

public String sayHello(String name) {

sleep();

return "Hello " + name;

}

private void sleep() {

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

sleep 方法模拟了一个耗时操作，主要是为了让服务端线程池耗尽。

## 客户端多线程访问

for (int i = 0; i < 20; i++) {

new Thread(() -> {

while (true){

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

try {

demoService.sayHello("Provider");

} catch (Exception e) {

e.printStackTrace();

}

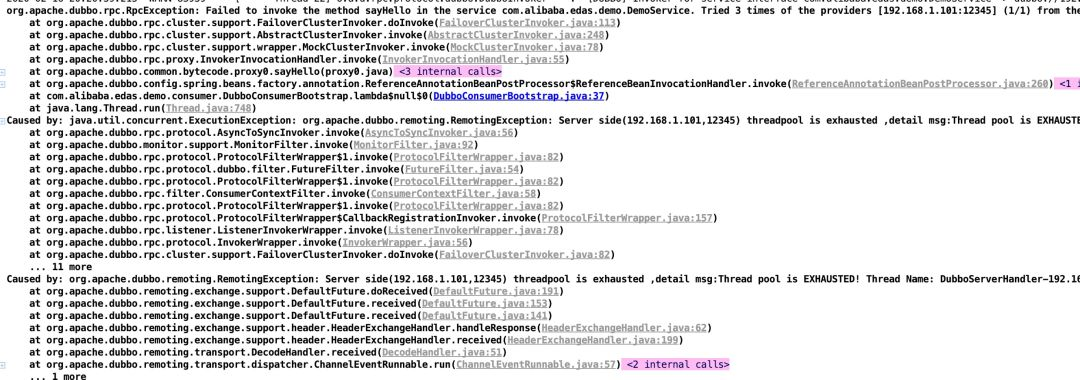
}

}).start();

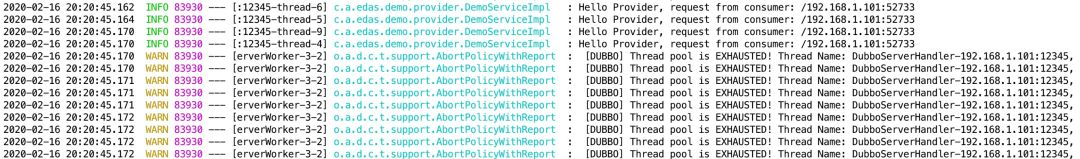
}

## 问题复现

客户端



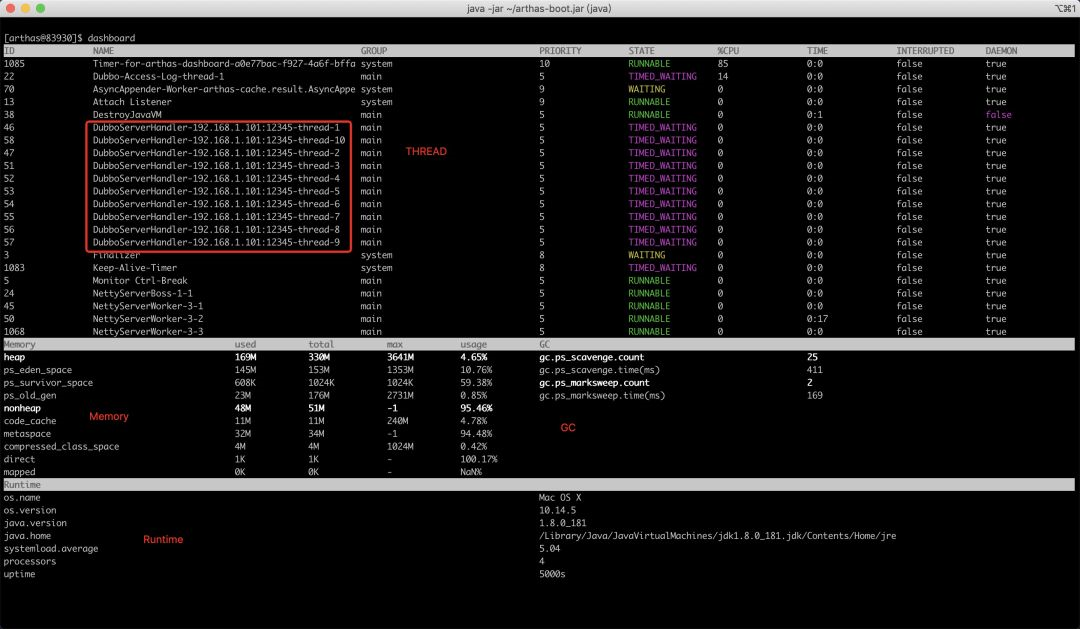
服务端



问题得以复现，保留该现场，并假设我们并不知晓 sleep 的耗时逻辑，使用 Arthas 来进行排查

# dashboard 命令介绍

选择进入dubbo生产者的进程后执行命令： dashboard



可以看到如上所示的面板，显示了一些系统的运行信息，这里主要关注 THREAD 面板，介绍一下各列的含义：

* ID: Java 级别的线程 ID，注意这个 ID 不能跟 jstack 中的 nativeID 一一对应
* NAME: 线程名
* GROUP: 线程组名
* PRIORITY: 线程优先级, 1~10 之间的数字，越大表示优先级越高
* STATE: 线程的状态
* CPU%: 线程消耗的 CPU 占比，采样 100ms，将所有线程在这 100ms 内的 CPU 使用量求和，再算出每个线程的 CPU 使用占比。
* TIME: 线程运行总时间，数据格式为分：秒
* INTERRUPTED: 线程当前的中断位状态
* DAEMON: 是否是 daemon 线程

在空闲状态下线程应该是处于 WAITING 状态，而因为 sleep 的缘故，现在所有的线程均处于 TIME\_WAITING 状态，导致后来的请求被处理时，抛出了线程池满的异常。

在实际排查中，需要抽查一定数量的 Dubbo 线程，记录他们的线程编号，看看它们到底在处理什么服务请求。使用如下命令可以根据线程池名筛选出 Dubbo 服务端线程：

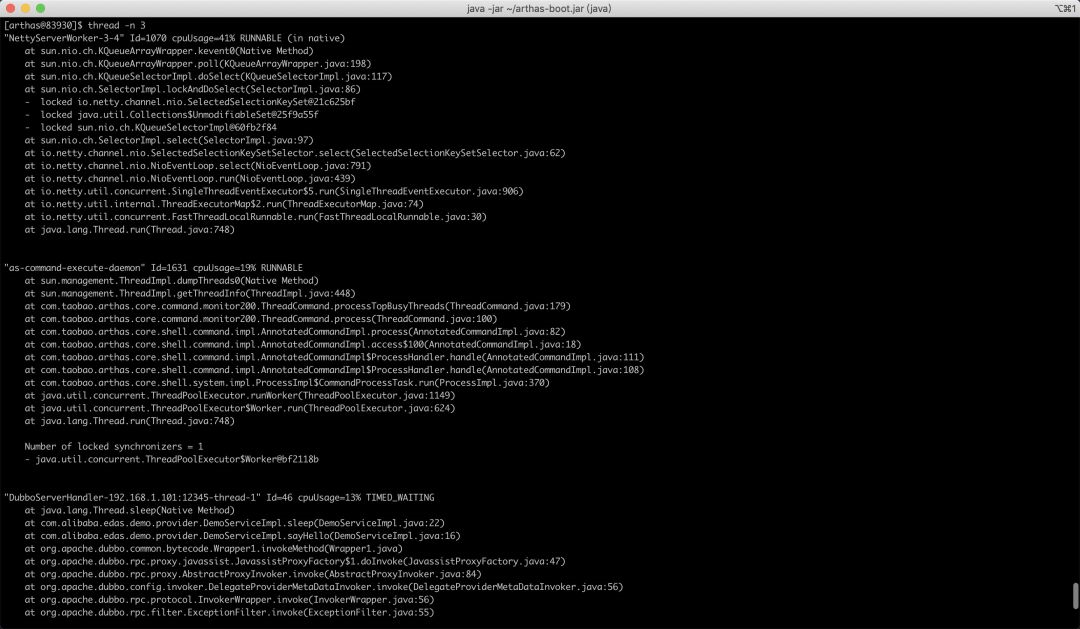
dashboard | grep "DubboServerHandler

# thread 命令介绍

使用 dashboard 筛选出个别线程 id 后，它的使命就完成了，剩下的操作交给 thread 命令来完成。其实，dashboard 中的 thread 模块，就是整合了 thread 命令，但是 dashboard 还可以观察内存和 GC 状态，视角更加全面，所以我个人建议，在排查问题时，先使用 dashboard 纵观全局信息。

## 查看当前最忙的前 n 个线程

$ thread -n 3



## 显示所有线程信息

$ thread

和 dashboard 中显示一致

## 显示当前阻塞其他线程的线程

$ thread -b

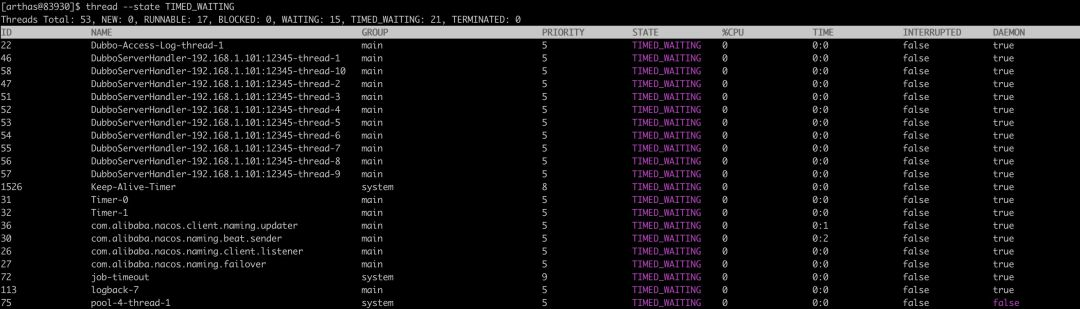
No most blocking thread found!

Affect(row-cnt:0) cost in 22 ms.

这个命令还有待完善，目前只支持找出 synchronized 关键字阻塞住的线程， 如果是 java.util.concurrent.Lock， 目前还不支持

## 显示指定状态的线程

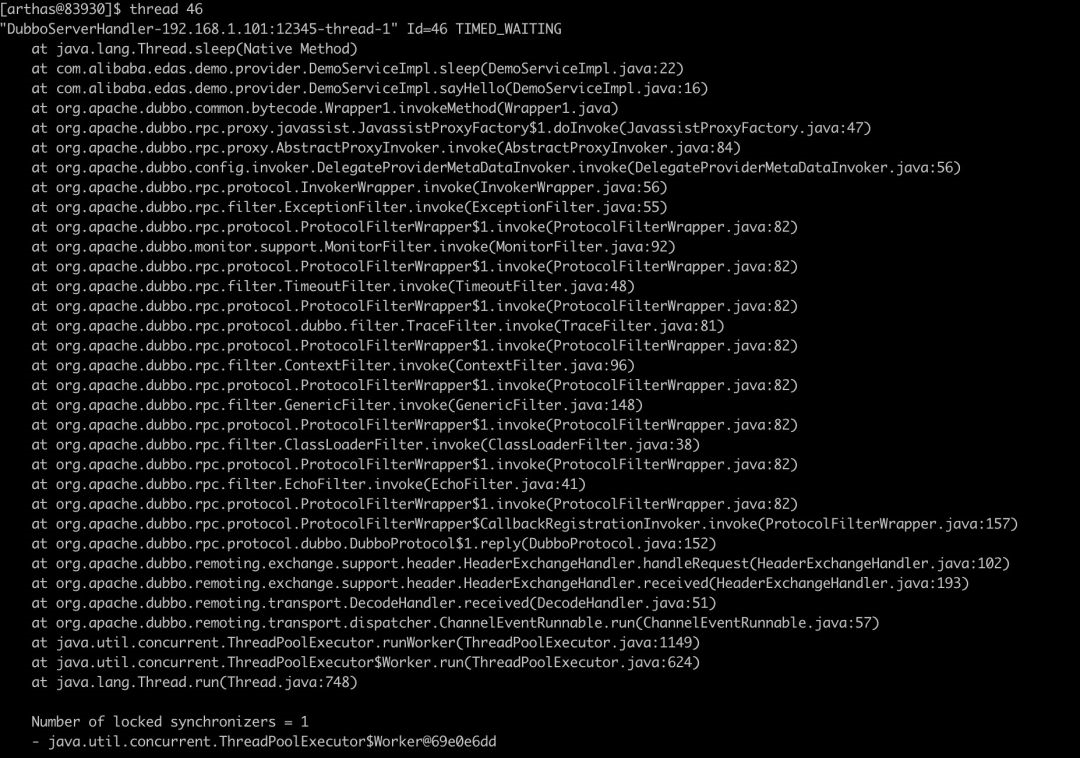
$ thread --state TIMED\_WAITING



线程的状态由6中：new（新建的）, runnable（准备就绪可运行的）,blocked（阻塞的）,waiting（等等待唤醒的）,timed\_waited（超时等待，到了时间自动返回），terminated（结束的）

## 查看指定线程的运行堆栈

$ thread 46



# 问题分析

分析线程池满异常并没有通法，需要灵活变通，我们对下面这些 case 一个个分析：

* 阻塞类问题。例如数据库连接不上导致卡死，运行中的线程基本都应该处于 BLOCKED 或者 TIMED\_WAITING 状态，我们可以借助 thread --state 定位到
* 繁忙类问题。例如 CPU 密集型运算，运行中的线程基本都处于 RUNNABLE 状态，可以借助于 thread -n 来定位出最繁忙的线程
* GC 类问题。很多外部因素会导致该异常，例如 GC 就是其中一个因素，这里就不能仅仅借助于 thread 命令来排查了。
* 定点爆破。还记得在前面我们通过 grep 筛选出了一批 Dubbo 线程，可以通过 thread ${thread\_id} 定向的查看堆栈，如果统计到大量的堆栈都是一个服务时，基本可以断定是该服务出了问题，至于说是该服务请求量突然激增，还是该服务依赖的某个下游服务突然出了问题，还是该服务访问的数据库断了，那就得根据堆栈去判断了。