03 | 反向压力：如何避免异步系统中的 OOM 问题?

在第 02 课时，我们使用了 Netty 并配合 Java 8 中的 CompletableFuture 类，构建了一个完全异步执行的数据采集服务器。经过这种改造，CPU 和 IO 的使用效率被充分发挥出来，显著提高了服务器在高并发场景下的性能。

但是，关于异步的问题我们还并没有彻底解决。上面的改造还存在一个致命的缺陷，也就是今天我们要讨论的，在异步系统中流量控制和反向压力的问题。

异步系统中的 OOM 问题

回想下 02 课时中，基于 Netty 和 CompletableFuture 类的数据采集服务器，关键是下面这部分代码（请参见完整代码）：

```
public static ExecutorService createExecutor(int nThreads, String threadNamePrefix) {
    return Executors.newFixedThreadPool(nThreads, threadNameThreadFactory(threadNamePrefix));
}
final private Executor decoderExecutor = createExecutor(2, "decoder");
final private Executor ectExecutor = createExecutor(8, "ect");
final private Executor senderExecutor = createExecutor(2, "sender");
@Override
protected void channelRead0(ChannelHandlerContext ctx, HttpRequest req) {
    CompletableFuture
            .supplyAsync(() -> this.decode(ctx, req), this.decoderExecutor)
            .thenApplyAsync(e -> this.doExtractCleanTransform(ctx, req, e), this.ectExecutor)
            .thenApplyAsync(e -> this.send(ctx, req, e), this.senderExecutor);
}
```

从上面的代码可以看出，我们在进行请求处理时，采用了 CompletableFuture 类提供的异步执行框架。在整个执行过程中，请求的处理逻辑都是提交给每个步骤各自的执行器，来进行处理，比如 decoderExecutor、ectExecutor 和 senderExecutor。

仔细分析下这些执行器你就会发现，在上面异步执行的过程中，没有任何阻塞的地方。只不过每个步骤都将它要处理的任务，存放在了执行器的任务队列中。每个执行器，如果它处理得足够快，那么任务队列里的任务都会被及时处理。这种情况下不存在什么问题。

但是，一旦有某个步骤处理的速度比较慢，比如在图 1 中，process 的速度比不上 decode 的速度，那么，消息就会在 process 的输入队列中积压。而由于执行器的任务队列，默认是非阻塞且不限容量的。这样，任务队列里积压的任务，就会越来越多。终有一刻，JVM 的内存会被耗尽，然后抛出 OOM 异常，程序就退出了。

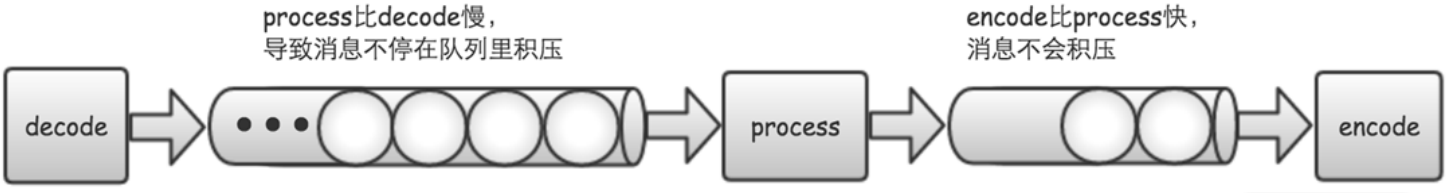

图 1 任务在各个Executor队列中积压

所以，为了避免 OOM 的问题，我们必须对上游输出给下游的速度做流量控制。那怎么进行流量控制呢?

一种方式，是**严格控制上游的发送速度**。比如，控制上游每秒钟只能发送 1000 条消息。这种方法是可行的，但是非常低效。如果实际下游每秒钟能够处理 2000 条消息，那么，上游每秒钟发送 1000 条消息，就会使得下游一半的性能没有发挥出来。如果下游因为某种原因，性能降级为每秒钟只能处理 500 条消息，那么在一段时间后，同样会发生 OOM 问题。

所以，我们该如何进行流量控制呢？这里有一种更优雅的方法，也就是**反向压力**。

反向压力原理

在反向压力的方案中，上游能够根据下游的处理能力，动态地调整输出速度。当下游处理不过来时，上游就减慢发送速度，当下游处理能力提高时，上游就加快发送速度。

反向压力的思想，已经成为流计算领域的共识，并且形成了反向压力相关的标准，也就是Reactive Streams 。

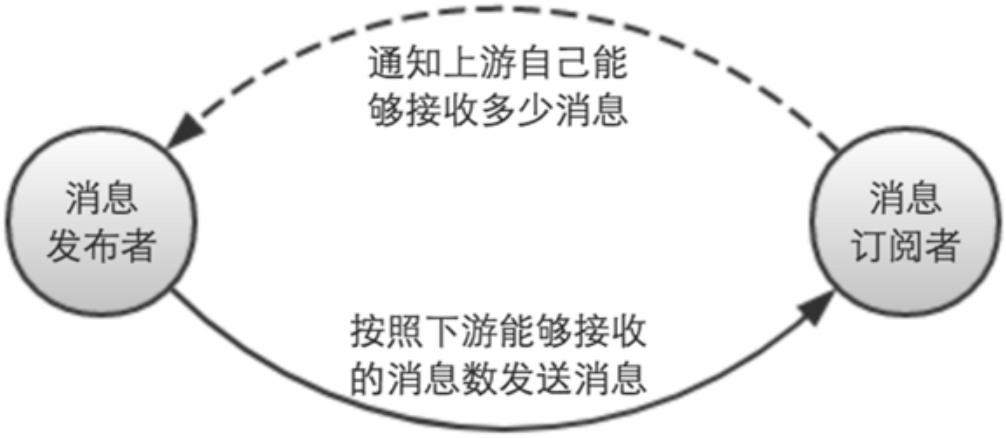

图2 Reactive Streams工作原理

上面的图 2 描述了 Reactive Streams 的工作原理。当下游的消息订阅者，从上游的消息发布者接收消息前，会先通知消息发布者自己能够接收多少消息。然后消息发布者就按照这个数量，向下游的消息订阅者发送消息。这样，整个消息传递的过程都是量力而行的，就不存在上下游之间因为处理速度不匹配，而造成的 OOM 问题了。

目前，一些主流的异步框架都开始支持 Reactive Streams 标准，比如 RxJava、Reactor、Akka Streams、Vert.x 等。这足以说明，OOM 和反向压力问题在异步系统中是多么重要！

实现反向压力

现在，我们回到 Netty 数据采集服务器。那究竟该怎样为这个服务器加上反向压力的功能呢?

前面我们分析了异步执行的过程，之所以会出现 OOM 问题，主要还是因为，接收线程在接收到新的请求后，触发了一系列任务。这些任务都会被存放在任务队列中，并且这些任务队列，都是非阻塞且不限容量的。

因此，要实现反向压力的功能，只需要从两个方面来进行控制。

1. 其一是，执行器的任务队列，它的容量必须是有限的。
2. 其二是，当执行器的任务队列已经满了时，就阻止上游继续提交新的任务，直到任务队列，重新有新的空间可用为止。

按照上面这种思路，我们就可以很容易地实现反向压力。下面的图 3 就展示了，使用容量有限的阻塞队列，实现反向压力的过程。

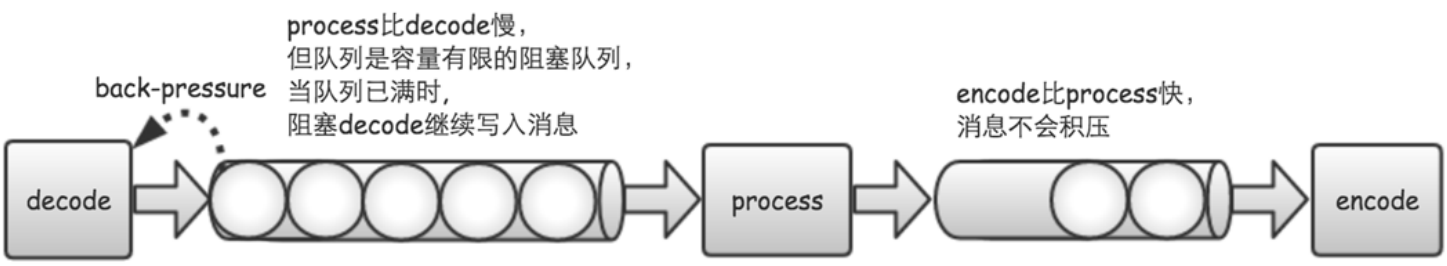

图 3 使用容量有限的阻塞队列实现反向压力

当 process 比 decode 慢时，运行一段时间后，位于 process 前的任务队列就会被填满。当 decode 继续往里面提交任务时，就会被阻塞，直到 process 从这个任务队列中取走任务为止。

以上说的都是实现原理。那具体用代码该怎样实现呢？下面就是这样一个具备反向压力能力的 ExecutorService 的具体实现。

```
private final List<ExecutorService> executors;
private final Partitioner partitioner;
private Long rejectSleepMills = 1L;
public BackPressureExecutor(String name, int executorNumber, int coreSize, int maxSize, int capac
    this.rejectSleepMills = rejectSleepMills;
    this.executors = new ArrayList<>(executorNumber);
    for (int i = 0; i < executorNumber; i++) {
        ArrayBlockingQueue<Runnable> queue = new ArrayBlockingQueue<>(capacity);
        this.executors.add(new ThreadPoolExecutor(
                coreSize, maxSize, 0L, TimeUnit.MILLISECONDS,
                queue,
                new ThreadFactoryBuilder().setNameFormat(name + "-" + i + "-%d").build(),
                new ThreadPoolExecutor.AbortPolicy()));
    }
    this.partitioner = new RoundRobinPartitionSelector(executorNumber);
}
@Override
public void execute(Runnable command) {
    boolean rejected;
    do {
        try {
           rejected = false;
           executors.get(partitioner.getPartition()).execute(command);
        } catch (RejectedExecutionException e) {
            rejected = true;
            try {
                TimeUnit.MILLISECONDS.sleep(rejectSleepMills);
            } catch (InterruptedException e1) {
                logger.warn("Reject sleep has been interrupted.", e1);
            }
        }
    } while (rejected);
}
```

在上面的代码中，BackPressureExecutor 类在初始化时，新建一个或多个 ThreadPoolExecutor 对象，作为执行任务的线程池。这里面的关键点有两个。

- 第一个是，在创建 ThreadPoolExecutor 对象时，采用 ArrayBlockingQueue。这是一个容量有限的阻塞队列。因此，当任务队列已经满了时，就会停止继续往队列里添加新的任务，从而避免内存无限大，造成 OOM 问题。
- 第二个是，将 ThreadPoolExecutor 拒绝任务时，采用的策略设置为 AbortPolicy。这就意味着，在任务队列已经满了的时候，如果再向任务队列提交任务，就会抛出 RejectedExecutionException 异常。之后，我们再通过一个 while 循环，在循环体内，捕获 RejectedExecutionException 异常，并不断尝试，重新提交任务，直到成功为止。

这样，经过上面的改造，当下游的步骤执行较慢时，它的任务队列就会占满。这个时候，如果上游继续往下游提交任务，它就会不停重试。这样，自然而然地降低了上游步骤的处理速度，从而起到了流量控制的作用。

接下来，我们就可以在数据接收服务器中，使用这个带有反向压力功能的 BackPressureExecutor 了（请参见 完整代码）。

```
final private Executor decoderExecutor = new BackPressureExecutor("decoderExecutor",
        1, 2, 1024, 1024, 1);
final private Executor ectExecutor = new BackPressureExecutor("ectExecutor",
        1, 8, 1024, 1024, 1);
final private Executor senderExecutor = new BackPressureExecutor("senderExecutor",
        1, 2, 1024, 1024, 1);
@Override
protected void channelRead0(ChannelHandlerContext ctx, HttpRequest req) {
    CompletableFuture
            .supplyAsync(() -> this.decode(ctx, req), this.decoderExecutor)
            .thenApplyAsync(e -> this.doExtractCleanTransform(ctx, req, e), this.ectExecutor)
            .thenApplyAsync(e -> this.send(ctx, req, e), this.senderExecutor);
}
```

从上面的代码可以看出，我们只需把 decode、doExtractCleanTransform 和 send 等每一个步骤用到的执行器，都替换成 BackPressureExecutor 即可。这样，就实现了反向压力功能，其他部分的代码，不需要做任何改变！

最后，还需要说明下的是，在 BackPressureExecutor 的实现中，为什么需要封装多个执行器呢？这是因为，使用 M * N 个线程，有三种不同的方法：

- 第一种是，每个执行器使用 1 个线程，然后使用个 M * N 执行器；
- 第二种是，每个执行器使用 M * N 个线程，然后使用 1 个执行器；
- 第三种是，每个执行器使用 M 个线程，然后使用 N 个执行器。

在不同场景下，三种使用方式的性能表现会有所不同。根据我的经验，主要是因为，队列的生产者之间，存在着相互竞争，然后队列的消费者之间，也存在着相互竞争。所以，如果你要使用这个类的话，还是需要根据实际的使用场景，分配合适的队列数和线程数，避免对同一个队列的竞争，过于激烈。这样，有利于提升程序的性能。

小结

今天，我用反向压力的功能进行流量控制，解决了异步系统中的 OOM 问题。对于一个能够在生产环境上稳定运行的系统来说，任何使用了异步技术的地方，都需要尤其注意OOM 问题。

其实，解决异步系统 OOM 问题的方法，并不限于反向压力。比如，我们在使用线程池时，设置线程的数量，这也是一种保护措施。但是，我们今天着重强调的是反向压力的方法。这是因为，反向压力在流计算系统中，有着非常重要的地位。像目前的流计算框架，比如 Flink、Spark Streaming 等，都支持反向压力。可以说，如果没有反向压力的功能，任何一个流计算系统，都会时时刻刻有着 OOM 崩溃的风险。

在今天的讨论中，我们已经多次用到了上游、下游，甚至是 Reactive Streams 这种，直接与“流”相关的字眼。我们已经隐隐约约感受到，“流”与“异步”之间，有着千丝万缕的关系。在接下来的课程中，我们还会专门讨论到，它们之间的关联关系。

相信通过今天的课程，你在以后使用异步编程时，一定会注意到系统的 OOM 问题。你在以往的编程中，有没有遇到过 OOM 问题呢？有的话，可以在评论区留言，我看到后会和你一起分析解决！

本课时精华：

点击此链接查看本课程所有课时的源码

PB 级企业大数据项目实战 + 拉勾硬核内推，5 个月全面掌握大数据核心技能。点击链接，全面赋能！