Flink 原理与实现：如何处理反压问题

摘要： 流处理系统需要能优雅地处理反压（backpressure）问题。反压通常产生于这样的场景：短时负载高峰导致系统接收数据的速率远高于它处理数据的速率。许多日常问题都会导致反压，例如，垃圾回收停顿可能会导致流入的数据快速堆积，或者遇到大促或秒杀活动导致流量陡增。反压如果不能得到正确的处理，可能会导致资源耗尽甚至系统崩溃。 目前主流的流处理系统 Storm/JStorm/Spark Streami

流处理系统需要能优雅地处理反压（backpressure）问题。反压通常产生于这样的场景：短时负载高峰导致系统接收数据的速率远高于它处理数据的速率。许多日常问题都会导致反压，例如，垃圾回收停顿可能会导致流入的数据快速堆积，或者遇到大促或秒杀活动导致流量陡增。反压如果不能得到正确的处理，可能会导致资源耗尽甚至系统崩溃。

目前主流的流处理系统 Storm/JStorm/Spark Streaming/Flink 都已经提供了反压机制，不过其实现各不相同。

Storm 是通过监控 Bolt 中的接收队列负载情况，如果超过高水位值就会将反压信息写到 Zookeeper ，Zookeeper 上的 watch 会通知该拓扑的所有 Worker 都进入反压状态，最后 Spout 停止发送 tuple。具体实现可以看这个 JIRA STORM-886。

JStorm 认为直接停止 Spout 的发送太过暴力，存在大量问题。当下游出现阻塞时，上游停止发送，下游消除阻塞后，上游又开闸放水，过了一会儿，下游又阻塞，上游又限流，如此反复，整个数据流会一直处在一个颠簸状态。所以 JStorm 是通过逐级降速来进行反压的，效果会较 Storm 更为稳定，但算法也更复杂。另外 JStorm 没有引入 Zookeeper 而是通过 TopologyMaster 来协调拓扑进入反压状态，这降低了 Zookeeper 的负载。

Flink 中的反压

那么 Flink 是怎么处理反压的呢？答案非常简单：Flink 没有使用任何复杂的机制来解决反压问题，因为根本不需要那样的方案！它利用自身作为纯数据流引擎的优势来优雅地响应反压问题。下面我们会深入分析 Flink 是如何在 Task 之间传输数据的，以及数据流如何实现自然降速的。

Flink 在运行时主要由 **operators** 和 **streams** 两大组件构成。每个 operator 会消费中间态的流，

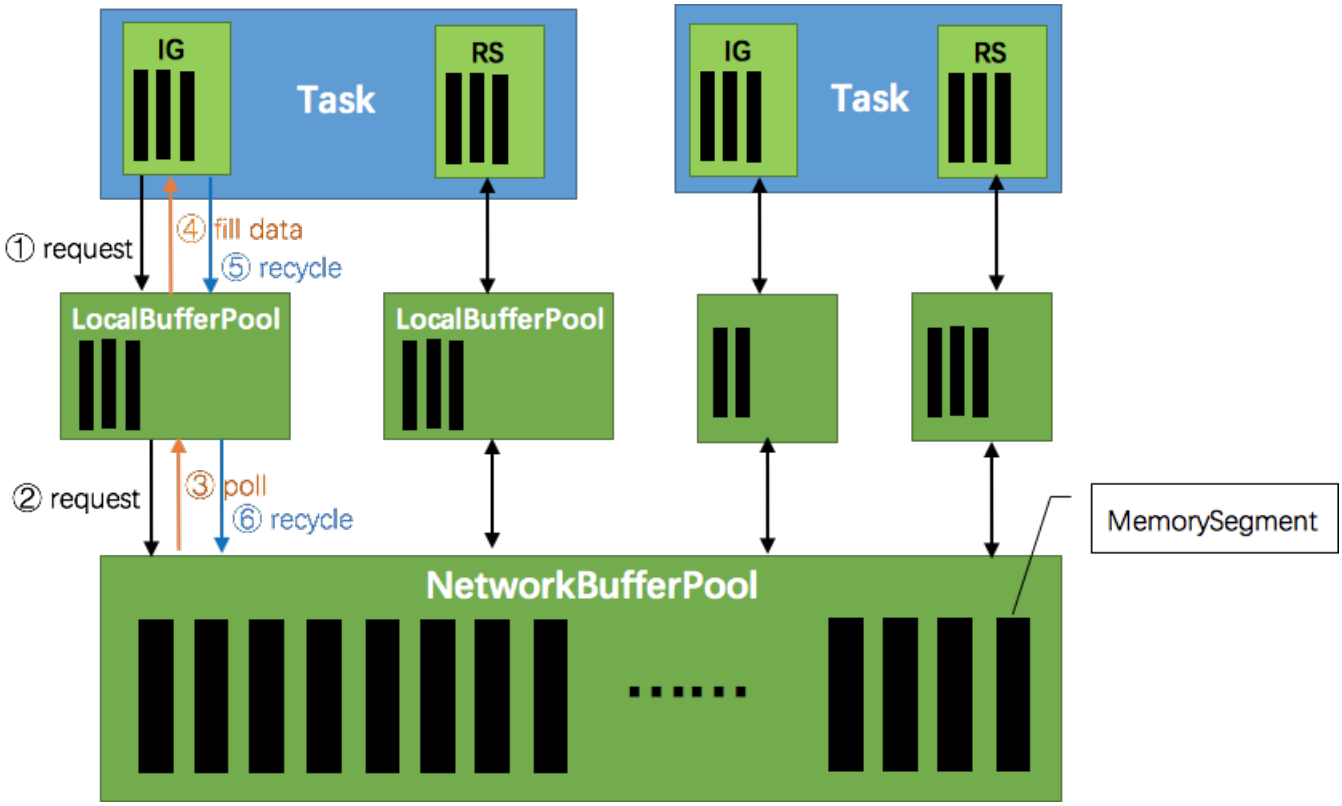
并在流上进行转换，然后生成新的流。对于 Flink 的网络机制一种形象的类比是，Flink 使用了高效有界的分布式阻塞队列，就像 Java 通用的阻塞队列（BlockingQueue）一样。还记得经典的线程间通信案例：生产者消费者模型吗？使用 BlockingQueue 的话，一个较慢的接受者会降低发送者的发送速率，因为一旦队列满了（有界队列）发送者会被阻塞。Flink 解决反压的方案就是这种感觉。

在 Flink 中，这些分布式阻塞队列就是这些逻辑流，而队列容量是通过缓冲池来（`LocalBufferPool`）实现的。每个被生产和被消费的流都会被分配一个缓冲池。缓冲池管理着一组缓冲(`Buffer`)，缓冲在被消费后可以被回收循环利用。这很好理解：你从池子中拿走一个缓冲，填上数据，在数据消费完之后，又把缓冲还给池子，之后你可以再次使用它。

在解释 Flink 的反压原理之前，我们必须先对 Flink 中网络传输的内存管理有个了解。

网络传输中的内存管理

如下图所示展示了 Flink 在网络传输场景下的内存管理。网络上传输的数据会写到 Task 的 InputGate（IG） 中，经过 Task 的处理后，再由 Task 写到 ResultPartition（RS） 中。每个 Task 都包括了输入和输入，输入和输出的数据存在`Buffer`中（都是字节数据）。Buffer 是 MemorySegment 的包装类。



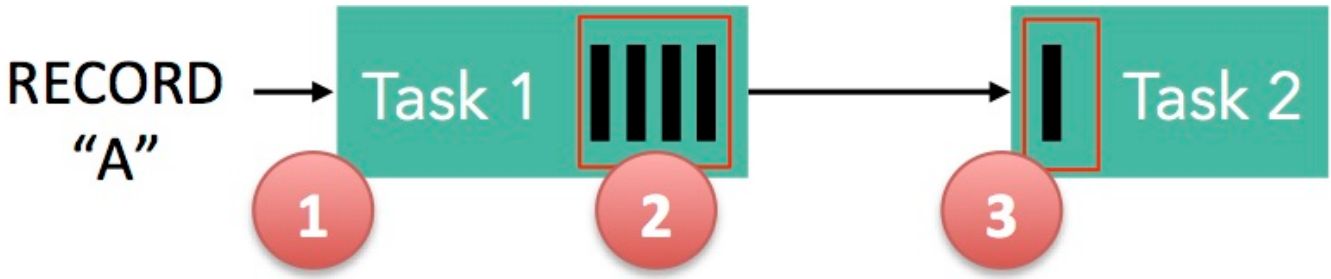
1. TaskManager（TM）在启动时，会先初始化`NetworkEnvironment`对象，TM 中所有与网络相关的东西都由该类来管理（如 Netty 连接），其中就包括`NetworkBufferPool`。根据配置，Flink 会在 NetworkBufferPool 中生成一定数量（默认2048个）的内存块 MemorySegment（关于 Flink 的内存管理，后续文章会详细谈到），内存块的总数量就代表了网络传输中所有可用的内存。NetworkEnvironment 和 NetworkBufferPool 是 Task 之间共

享的，每个 TM 只会实例化一个。

2. Task 线程启动时，会向 NetworkEnvironment 注册，NetworkEnvironment 会为 Task 的 InputGate（IG）和 ResultPartition（RP） 分别创建一个 LocalBufferPool（缓冲池）并设置可申请的 MemorySegment（内存块）数量。IG 对应的缓冲池初始的内存块数量与 IG 中 InputChannel 数量一致，RP 对应的缓冲池初始的内存块数量与 RP 中的 ResultSubpartition 数量一致。不过，每当创建或销毁缓冲池时，NetworkBufferPool 会计算剩余空闲的内存块数量，并平均分配给已创建的缓冲池。注意，这个过程只是指定了缓冲池所能使用的内存块数量，并没有真正分配内存块，只有当需要时才分配。为什么要动态地为缓冲池扩容呢？因为内存越多，意味着系统可以更轻松地应对瞬时压力（如GC），不会频繁地进入反压状态，所以我们要利用起那部分闲置的内存块。
3. 在 Task 线程执行过程中，当 Netty 接收端收到数据时，为了将 Netty 中的数据拷贝到 Task 中，InputChannel（实际是 RemoteInputChannel）会向其对应的缓冲池申请内存块（上图中的①）。如果缓冲池中也没有可用的内存块且已申请的数量还没到池子上限，则会向 NetworkBufferPool 申请内存块（上图中的②）并交给 InputChannel 填上数据（上图中的③和④）。如果缓冲池已申请的数量达到上限了呢？或者 NetworkBufferPool 也没有可用内存块了呢？这时候，Task 的 Netty Channel 会暂停读取，上游的发送端会立即响应停止发送，拓扑会进入反压状态。当 Task 线程写数据到 ResultPartition 时，也会向缓冲池请求内存块，如果没有可用内存块时，会阻塞在请求内存块的地方，达到暂停写入的目的。
4. 当一个内存块被消费完成之后（在输入端是指内存块中的字节被反序列化成对象了，在输出端是指内存块中的字节写入到 Netty Channel 了），会调用 `Buffer.recycle()` 方法，会将内存块还给 LocalBufferPool （上图中的⑤）。如果LocalBufferPool中当前申请的数量超过了池子容量（由于上文提到的动态容量，由于新注册的 Task 导致该池子容量变小），则 LocalBufferPool会将该内存块回收给 NetworkBufferPool（上图中的⑥）。如果没超过池子容量，则会继续留在池子中，减少反复申请的开销。

反压的过程

下面这张图简单展示了两个 Task 之间的数据传输以及 Flink 如何感知到反压的：



1. 记录“A”进入了 Flink 并且被 Task 1 处理。（这里省略了 Netty 接收、反序列化等过程）
2. 记录被序列化到 buffer 中。
3. 该 buffer 被发送到 Task 2，然后 Task 2 从这个 buffer 中读出记录。

不要忘了：记录能被 Flink 处理的前提是，必须有空闲可用的 Buffer。

结合上面两张图看：Task 1 在输出端有一个相关联的 LocalBufferPool（称缓冲池1），Task 2 在输入端也有一个相关联的 LocalBufferPool（称缓冲池2）。如果缓冲池1中有空闲可用的 buffer 来序列化记录“A”，我们就序列化并发送该 buffer。

这里我们需要注意两个场景：

- 本地传输：如果 Task 1 和 Task 2 运行在同一个 worker 节点（TaskManager），该 buffer 可以直接交给下一个 Task。一旦 Task 2 消费了该 buffer，则该 buffer 会被缓冲池1回收。如果 Task 2 的速度比 1 慢，那么 buffer 回收的速度就会赶不上 Task 1 取 buffer 的速度，导致缓冲池1无可用的 buffer，Task 1 等待在可用的 buffer 上。最终形成 Task 1 的降速。
- 远程传输：如果 Task 1 和 Task 2 运行在不同的 worker 节点上，那么 buffer 会在发送到网络（TCP Channel）后被回收。在接收端，会从 LocalBufferPool 中申请 buffer，然后拷贝网络中的数据到 buffer 中。如果没有可用的 buffer，会停止从 TCP 连接中读取数据。在输出端，通过 Netty 的水位值机制来保证不往网络中写入太多数据（后面会说）。如果网络中的数据（Netty输出缓冲中的字节数）超过了高水位值，我们会等到其降到低水位值以下才继续写入数据。这保证了网络中不会有太多的数据。如果接收端停止消费网络中的数据（由于接收端缓冲池没有可用 buffer），网络中的缓冲数据就会堆积，那么发送端也会暂停发送。另外，这会使得发送端的缓冲池得不到回收，writer 阻塞在向 LocalBufferPool 请求 buffer，阻塞了 writer 往 ResultSubPartition 写数据。

这种固定大小缓冲池就像阻塞队列一样，保证了 Flink 有一套健壮的反压机制，使得 Task 生产数据的速度不会快于消费的速度。我们上面描述的这个方案可以从两个 Task 之间的数据传输自然地扩展到更复杂的 pipeline 中，保证反压机制可以扩散到整个 pipeline。

Netty 水位值机制

下方的代码是初始化 NettyServer 时配置的水位值参数。

```
// 默认高水位值为2个buffer大小，当接收端消费速度跟不上，发送端会立即感知到
bootstrap.childOption(ChannelOption.WRITE_BUFFER_LOW_WATER_MARK, config.getMemory
bootstrap.childOption(ChannelOption.WRITE_BUFFER_HIGH_WATER_MARK, 2 * config.getM
```

当输出缓冲中的字节数超过了高水位值，则 Channel.isWritable() 会返回false。当输出缓存中的字节数又掉到了低水位值以下，则 Channel.isWritable() 会重新返回true。Flink 中发送数据的核心代码在 `PartitionRequestQueue` 中，该类是 server channel pipeline 的最后一层。发送数据关键代码如下所示。

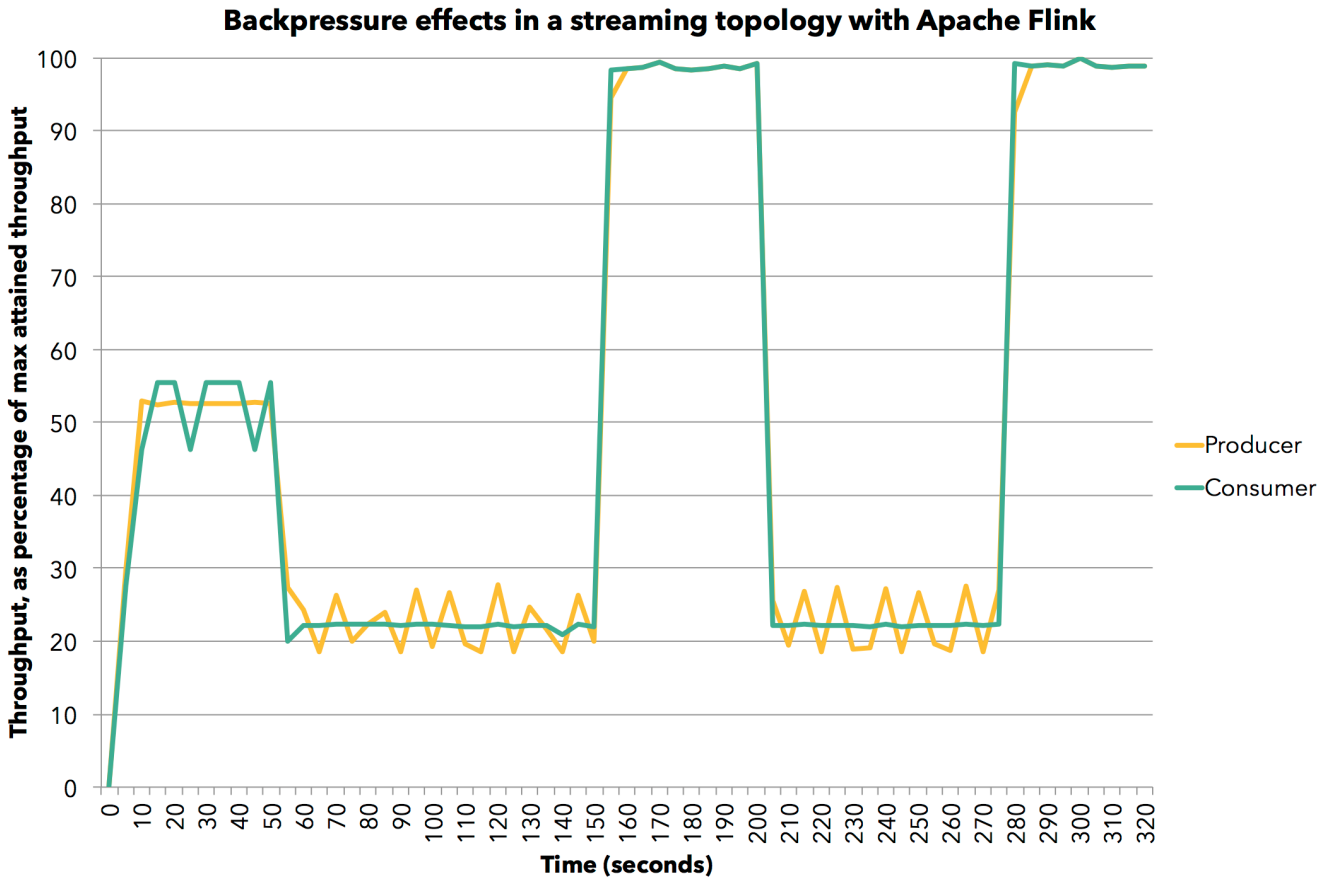
```
private void writeAndFlushNextMessageIfPossible(final Channel channel) throws IOE
    if (fatalError) {
        return;
```

```
    }

    Buffer buffer = null;

    try {
        // channel.isWritable() 配合 WRITE_BUFFER_LOW_WATER_MARK
        // 和 WRITE_BUFFER_HIGH_WATER_MARK 实现发送端的流量控制
        if (channel.isWritable()) {
            // 注意: 一个while循环也就最多只发送一个BufferResponse, 连续发送BufferRespor
            while (true) {
                if (currentPartitionQueue == null && (currentPartitionQueue = que
                    return;
                }

                buffer = currentPartitionQueue.getNextBuffer();

                if (buffer == null) {
                    // 跳过这部分代码
                    ...
                }
                else {
                    // 构造一个response返回给客户端
                    BufferResponse resp = new BufferResponse(buffer, currentParti

                    if (!buffer.isBuffer() &&
                            EventSerializer.fromBuffer(buffer, getClass().getClas
                        // 跳过这部分代码。batch 模式中 subpartition 的数据准备就绪, 通知
                        ...
                    }

                    // 将该response发到netty channel, 当写成功后,
                    // 通过注册的writeListener又会回调进来, 从而不断地消费 queue 中的请求
                    channel.writeAndFlush(resp).addListener(writeListener);

                    return;
                }
            }
        }
    }
    catch (Throwable t) {
        if (buffer != null) {
            buffer.recycle();
        }

        throw new IOException(t.getMessage(), t);
    }
}

// 当水位值降下来后（channel 再次可写），会重新触发发送函数
@Override
public void channelWritabilityChanged(ChannelHandlerContext ctx) throws Exception
    writeAndFlushNextMessageIfPossible(ctx.channel());
}
```

核心发送方法中如果channel不可写，则会跳过发送。当channel再次可写后，Netty 会调用该Handle的 `channelWritabilityChanged` 方法，从而重新触发发送函数。

反压实验

另外，官方博客中为了展示反压的效果，给出了一个简单的实验。下面这张图显示了：随着时间的改变，生产者（黄色线）和消费者（绿色线）每5秒的平均吞吐与最大吞吐（在单一JVM中每秒达到8百万条记录）的百分比。我们通过衡量task每5秒钟处理的记录数来衡量平均吞吐。该实验运行在单 JVM 中，不过使用了完整的 Flink 功能栈。



首先，我们运行生产task到它最大生产速度的60%（我们通过Thread.sleep()来模拟降速）。消费者以同样的速度处理数据。然后，我们将消费task的速度降至其最高速度的30%。你就会看到背压问题产生了，正如我们所见，生产者的速度也自然降至其最高速度的30%。接着，停止消费task的人为降速，之后生产者和消费者task都达到了其最大的吞吐。接下来，我们再次将消费者的速度降至30%，pipeline给出了立即响应：生产者的速度也被自动降至30%。最后，我们再次停止限速，两个task也再次恢复100%的速度。总而言之，我们可以看到：生产者和消费者在 pipeline 中的处理都在跟随彼此的吞吐而进行适当的调整，这就是我们希望看到的反压的效果。

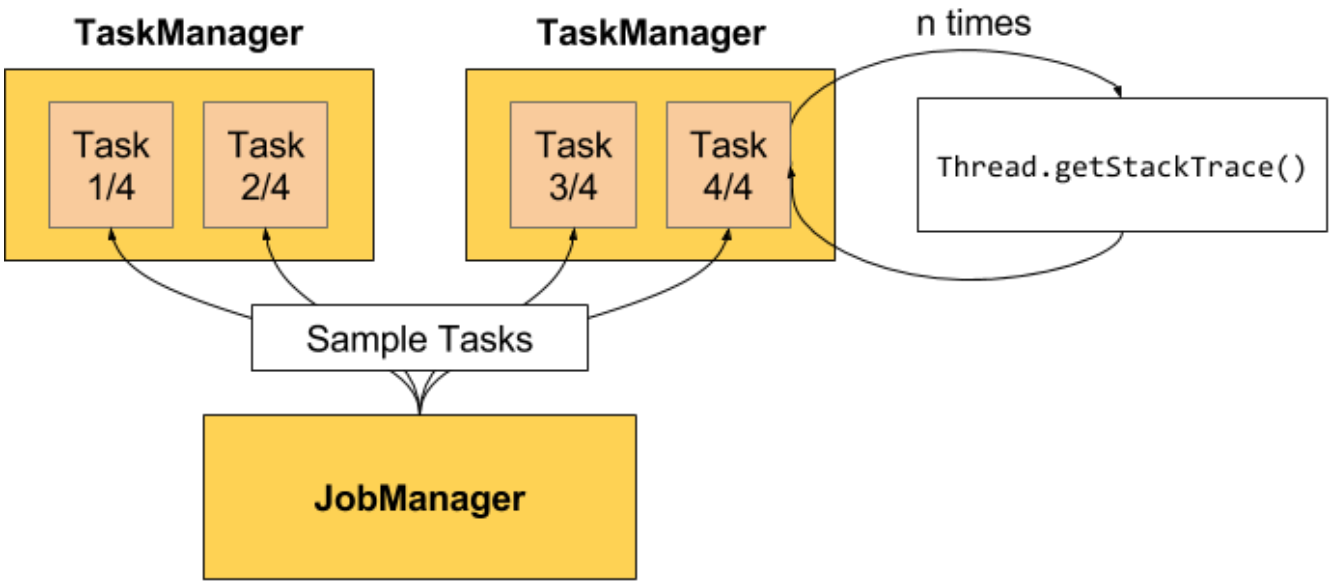
反压监控

在 Storm/JStorm 中，只要监控到队列满了，就可以记录下拓扑进入反压了。但是 Flink 的反压太过于天然了，导致我们无法简单地通过监控队列来监控反压状态。Flink 在这里使用了一个 trick 来

实现对反压的监控。如果一个 Task 因为反压而降速了，那么它会卡在向 `LocalBufferPool` 申请内存块上。那么这时候，该 Task 的 stack trace 就会长下面这样：

```
java.lang.Object.wait(Native Method)
o.a.f.[...].LocalBufferPool.requestBuffer(LocalBufferPool.java:163)
o.a.f.[...].LocalBufferPool.requestBufferBlocking(LocalBufferPool.java:133) <---
[...]
```

那么事情就简单了。通过不断地采样每个 task 的 stack trace 就可以实现反压监控。



Flink 的实现中，只有当 Web 页面切换到某个 Job 的 Backpressure 页面，才会对这个 Job 触发反压检测，因为反压检测还是挺昂贵的。JobManager 会通过 Akka 给每个 TaskManager 发送`TriggerStackTraceSample`消息。默认情况下，TaskManager 会触发100次 stack trace 采样，每次间隔 50ms（也就是说一次反压检测至少要等待5秒钟）。并将这 100 次采样的结果返回给 JobManager，由 JobManager 来计算反压比率（反压出现的次数/采样的次数），最终展现在 UI 上。UI 刷新的默认周期是一分钟，目的是不对 TaskManager 造成太大的负担。

总结

Flink 不需要一种特殊的机制来处理反压，因为 Flink 中的数据传输相当于已经提供了应对反压的机制。因此，Flink 所能获得的最大吞吐量由其 pipeline 中最慢的组件决定。相对于 Storm/JStorm 的实现，Flink 的实现更为简洁优雅，源码中也看不见与反压相关的代码，无需 Zookeeper/TopologyMaster 的参与也降低了系统的负载，也利于对反压更迅速的响应。

参考资料

- How Flink handles backpressure
- Flink: Back Pressure Monitoring