反应式流

回压的原理与实现

By 王石冲

个人简介

- Scala程序员
- 杭州数云信息技术有限公司架构师
- 《反应式设计模式》译者
- 第四届阿里中间件性能挑战赛优胜奖

内容

- 什么是反应式流
- 反应式流要处理的问题
- 回压的原理和实现
- 丢弃而不是崩溃

什么是反应式流

- RxJava, RxJs, Rx.Net…?
- 不是某种具体实现
- 反应式流只是一种倡议，它提倡一种处理异步数据流的标准，即，使用非阻塞的回压机制。
- 反应式流瞄准的场景不仅包括运行时环境（JVM和JavaScript），也包括网络协议 （rsocket）

流处理面临的问题

- 无法确定大小和速度的活跃数据流
- 数据跨越异步边界（从独立的生产者到独立的消费者）
- 过快的生产者可能会淹没消费者（推模式）
- 消费者可能会需要耗费资源来轮询生产者，以获取数据（拉模式）

回压如何解决这些问题

什么是回压?

- 英文是Back pressure. 在反应式流里面，翻译成回压比背压合适。
- 流的下游向上游发出的信号
- 当下游的处理能力跟不上上游的发送能力时，就发出回压信息，让上游减慢速度，或者丢弃数据
- 回压必须是异步的，否则会抵消回压带来的好处

回压的原理

传统的生产者消费者解决方案

- 信号量
- 异步队列（阻塞回压）
- JCtools

反应式流的方案原理

- 跨学科理论的结合（控制理论在计算机科学里面的应用）
- 排队论
- 由Roland Khun博士在反应式流的初始倡议中提出，后被采纳作为标准

场景实例

场景一：速率可控的生产者

- 假设利用Akka的Actor来计算调和级数：1 - 1/2 + 1/3 - 1/4 … (向2的自然对数收敛)。在达到指定精度之后停止计算
- 由管理者Actor负责分发计算任务和符号
- 由工作者Actor来负责计算浮点数

拉取模式

- 让消费者向生产者对数据的批量大小提出要求。

流程

- 由工作者Actor在启动的时候发起工作请求，每次发10个
- 管理者Actor根据工作请求发回给工作者
- 当工作者请求的工作数量小于5个的时候，就再次发送10个请求
- 这样持续往复，直到达到指定精度，或者完成所有工作数量

工作者代码

```
class Worker(manager: ActorRef) extends Actor {
  private val mc = new MathContext( setPrecision = 100, RoundingMode.HALF_EVEN)
  private val plus = BigDecimal(1, mc)
  private val minus = BigDecimal(-1, mc)

  private var requested = 0

  def request(): Unit =
    if (requested < 5) {
      manager ! WorkRequest(self, 10)
      requested += 10
    }

  request()

  def receive: Receive = {
    case Job(id, data, replyTo) ⇒
      requested -= 1
      request()
      val sign = if ((data & 1) == 1) plus else minus
      val result = sign / data
      replyTo ! JobResult(id, result)
  }
}
```

管理者代码

```
class Manager extends Actor {

  private val workStream: Iterator[Job] =
    Iterator.range(1, 1000000).map(x => Job(x, x, self))

  private val aggregator: (BigDecimal, BigDecimal) ⇒ BigDecimal = (x: BigDecimal, y: BigDecimal) ⇒ x + y
  private val mc = new MathContext( setPrecision = 10000, RoundingMode.HALF_EVEN)
  private var approximation: BigDecimal = BigDecimal(0, mc)

  private var outstandingWork: Int = 0

  (1 to 8) foreach (_ ⇒ context.actorOf(Props(new Worker(self))))

  def receive: Receive = {
    case WorkRequest(worker, items) ⇒
      workStream.take(items).foreach { job ⇒
        worker ! job
        outstandingWork += 1
      }
    case JobResult(id, report) ⇒
      approximation = aggregator(approximation, report)
      outstandingWork -= 1
      if (outstandingWork == 0 && workStream.isEmpty) {
        println(s"final result: $approximation")
        context.system.terminate()
      }
  }
}
```

单个任意快的生产者和多个较慢的消费者

- 如果生产者立即分布任务给消费者，可能会耗尽内存（无界缓冲区），或者生产者阻塞（阻塞队列）
- 预先决定的平均分布最终得不到平均的执行情况
- 如果某个工作者失败了，那么所有分配给它的任务都会丢失

拉取模式的特点

- 当生产者快于消费者，生产者最终会缺少需求（拉取状态）
- 当生产者慢于消费者，消费者最终会有未被满足的需求（推送状态）
- 在负载一直变化的时候，这个机制将自动地在签署两种模式中切换，而不需要任何额外的协调工作（动态推拉状态）

思考

- 这个过程，回压体现在哪里?
- 如果生产者只是中间节点，回压能继续向上传播吗?
- 回压可能的缺点?

场景二：任务来自于外界，速率不可控

- 管理者要缓存任务，平滑系统，减少摩擦
- 在任务产生速度过快时，要拒绝任务，保护系统

托管队列模式

- 管理一条显式的输入队列，并对其填充级别予以反应。

加入队列以后的管理者

```
class Manager extends Actor {

  private var workQueue: Queue[Job] = Queue.empty[Job]
  private var requestQueue: Queue[WorkRequest] = Queue.empty[WorkRequest]

  (1 to 8) foreach (_ ⇒ context.actorOf(Props(new Worker(self))))

  def receive: Receive = {
    case job@Job(id, _, replyTo) ⇒
      if (requestQueue.isEmpty) {
        if (workQueue.size < 1000) workQueue :+= job
        else replyTo ! JobRejected(id)
      } else {
        val WorkRequest(worker, items) = requestQueue.head
        worker ! job
        if (items > 1) {
          worker ! DummyWork(items - 1)
        }
        requestQueue = requestQueue.drop(1)
      }
    case wr@WorkRequest(worker, items) ⇒
      if (workQueue.isEmpty) {
        requestQueue :+= wr
      } else {
        workQueue.iterator.take(items).foreach(job ⇒ worker ! job)
        val sent = Math.min(workQueue.size, items)
        if (sent < items) {
          worker ! DummyWork(items - sent)
        }
        workQueue = workQueue.drop(items)
      }
  }
}
```

此时的工作者Actor

```
class Worker(manager: ActorRef) extends Actor {
  val mc = new MathContext( setPrecision = 100, RoundingMode.HALF_EVEN)
  private val plus = BigDecimal(1, mc)
  private val minus = BigDecimal(-1, mc)

  private var requested = 0

  def request(): Unit =
    if (requested < 5) {
      manager ! WorkRequest(self, 10)
      requested += 10
    }

  request()

  def receive: Receive = {
    case Job(id, data, replyTo) ⇒
      requested -= 1
      request()
      val sign = if ((data & 1) == 1) plus else minus
      val result = sign / data
      replyTo ! JobResult(id, result)
    case DummyWork(count) ⇒
      requested -= count
      request()
  }
}
```

加入托管队列，我们得到了啥?

- 平滑消费过程
- 可以通过观测队列填充程度来获得更多信息：
  - 缓冲队列过长，可以启动新的工作者
  - requestQueue过长，可以移除工作者
- 或者观察生产者和消费者的速率差
  - 速率持续增长，扩大工作者池子
  - 速率持续下降，缩小工作者池子

场景三：如何在场景二下最大化利用系统

- 外界速率远远大于系统处理能力
- 此时消息不仅仅会填满管理者内部维护的队列，还会塞满其Mailbox

丢弃模式

- 丢弃请求，比不受控制地失败更加可取

方法

- 改写管理者Actor，当流入速率超过工作者处理能力的8倍的时候，丢弃任务，且不发回任何响应

管理者Actor

```
private val queueThreshold = 1000
private val dropThreshold = 1384

def random: ThreadLocalRandom = ThreadLocalRandom.current

def shallEnqueue(atSize: Int): Boolean =
  (atSize < queueThreshold) || {
    val dropFactor = (atSize - queueThreshold) >> 6
    random.nextInt(dropFactor + 2) == 0
  }
```

```
case job @ Job(id, _, replyTo) ⇒
  if (requestQueue.isEmpty) {
    val atSize = workQueue.size
    if (shallEnqueue(atSize)) {
      workQueue :+= job
    } else if (atSize < dropThreshold) {
      replyTo ! JobRejected(id)
    }
  } else {
```

丢弃模式

- 服务降级功能只能在给定的点下才能生效
- 当这个点被突破，完全不提供任何功能，比功能降级，代价更低
- 在严重过载下，这是服务保持对资源控制的唯一手段
- 此时，在强脉冲流量下，仍可按照特定比例入列，而不是全部拒绝。

内容和源码来源

- 《反应式设计模式》第16章——流量控制
- https://rdp.reactiveplatform.xyz/chapter-16/index.html

强烈推荐《反应式设计模式》一书！

广告时间

WayneWang12的技术博客

Q & A