15 | 扩展为集群：如何实现分布式状态存储?

今天我们需要接着讨论有关流计算中状态管理的另外一个重要问题，也就是将状态存储扩展为集群的问题。

为什么说将状态存储扩展为集群会很重要呢?

一方面，这是因为当业务量比较大时，单一的机器节点将不足以处理业务数据洪流，必须通过将流计算系统扩展为集群，来提升系统整体处理能力。

而另一方面，在将流计算系统扩展为集群的过程中，你会发现，难点往往不是增加更多的机器，而是，**如何更加高效地使用所有机器上的内存！**

所以，今天我就专门分享下，如何将状态存储扩展为集群。

扩展为集群

说到将系统扩展为集群，最关键的两点，就是**扩展 CPU 的核心数和内存的容量**。

扩展 CPU 的核心数，对应着**提升系统的计算能力**。在实时流计算系统中，不管是使用诸如 Kafka 消息中间件的分区功能，还是使用像 Flink KeyedStream 这样的流计算框架本身的分区流能力，最终都能轻松方便地实现计算能力的水平扩展。

但是，对于计算中的状态数据来说，就不是件非常容易的事情了。因为状态数据很多时候是需要**共享和同步**的。比如，分别在两个计算节点上被计算的事件，它们可能需要同时访问相同的数据，就算我们先不考虑并发安全的问题，这也意味着相同的数据会被两个不同的节点访问。也就是说，至少有一个节点的跨网络远程访问是不可避免的了。

而根据我们前边在 第 10 以及 11 课时 中的实践经验，在计算时间维度聚合值和分析关联图谱时，我们需要进行大量状态访问。甚至有时候，一次窗口计数的查询会访问几个甚至几十个子窗口的寄存器。如果不能避免或优化这些访问，那么程序的性能势必会严重受累于跨网络的远程状态访问。

所以，我们有必要详细讨论状态的存储和管理问题。下面我将讨论以下三种不同的状态集群方案，将单节点的状态存储扩展为分布式集群的状态存储。

1. 基于 Redis 的状态集群
2. 基于 Apache Ignite 的状态集群
3. 基于分布式文件系统的状态集群

这三种集群方案，分别代表了一种典型的分布式计算架构设计思路，可谓各有千秋。所以，希望你能好好掌握。

基于 Redis 的状态集群

首先，我们还是从基于 Redis 的状态集群方案说起。图 1 展示了用 Redis 集群实现状态分布式存储和管理的原理。

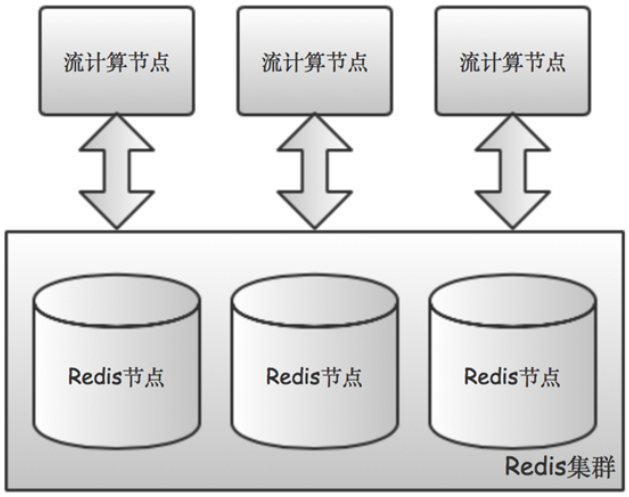

图1 使用Redis集群进行状态存储和管理

在上面的图 1 中，当采用 Redis 集群实现分布式状态存储和管理时，流计算集群和 Redis 集群节点是分离开的。**流计算集群中的每个节点都可以任意访问 Redis 集群中的任何一个节点**。

这样的架构有个非常明显的好处，由于计算和数据是分离开的，所以我们在任何时候，可以任意地新增流计算节点，而不必影响 Redis 集群。反过来，我们也可以任意地新增 Redis 节点，而不必影响流计算集群。

但这样的体系结构也有缺点。以“过去一天同一用户的总交易金额”这个时间维度聚合特征的计算为例。如果我们采用与 10 课时 中相同的算法，就需要先将“1 天”分成 24 个“1 小时”的子窗口。这样，在查询计算时将有 24 次的 Redis GET 操作。

假设这 24 个子窗口的状态数据是分散在 6 台 Redis 上的。如果不做任何优化设计，那么这一个特征计算就需要 24 次 IO 操作，而且要牵涉到与 6 台不同服务器的远程通信，这势必对性能造成极大的影响。

针对以上问题我们该怎么办呢？这里，我们可以根据“局部性原理”和“批次请求处理”的思想来对方案进行优化。

局部性原理

我们先来说说“局部性原理”。局部性原理是指计算单元在访问存储单元时，所访问的存储单元应该趋向于聚集在一个局部的连续区域内。利用局部性原理可以更加充分地提高计算资源的使用效率，从而提高程序的性能。

前面讲到在“过去一天同一用户的总交易金额”的计算中，我们可能需要访问 6 台 Redis 节点上的数据。这是因为默认情况下，Redis Cluster 将数据按照 key 做 hash 后分散各个槽（slot）里，而槽又分布在各个 Redis 节点上。

如果，我们能够让“同一用户”的状态数据保存在相同的槽里，就可以让这批数据存在于相同的 Redis 节点上。正好，Redis 的官方集群方案 Redis Cluster 为我们提供了贴心的标签（tag）功能，允许只使用 key 中的部分字段来计算 hash 值。

具体而言就是，如果 hash_tag 指定为“{}”，那么当 key 中含有“{}”的时候，就不使用整个 key 来计算 hash 值，而只对“{}”包括的部分字段计算 hash 值。比如在使用标签功能后，每个小窗口内记录这个窗口交易总金额的 key 如下所示:

```
$event_type.{$userid}.$window_unit.$window_index
```

经过标签化的 key，相同用户的状态数据会落在相同的 Redis 节点。这样，我们只需要访问一个 Redis 节点了。

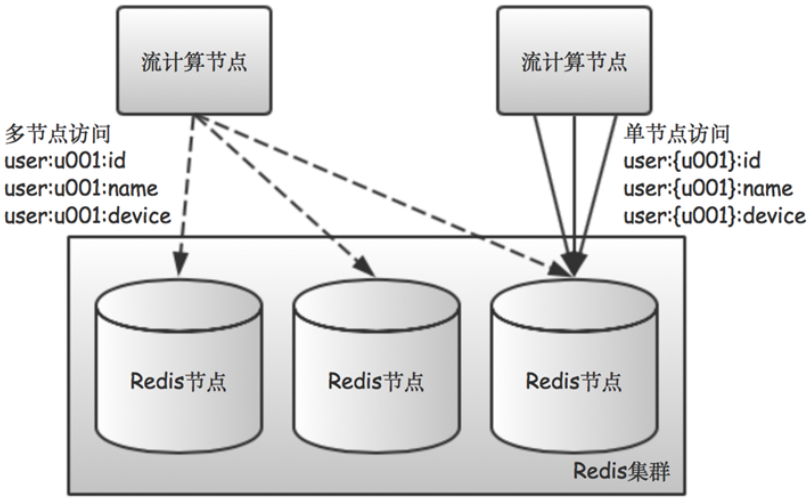

图 2 局部性原理：使用 hash tag 将属于同一用户的记录都分配到相同的Redis节点

现在数据放在同一个节点上了，那这有什么好处呢？好处多着呢。

首先，我们可以放心大胆地使用 Redis 的各种多键指令了，比如 MGET、MSET、SUNION 和 SUNIONSTORE 等。这些指令在操作过程中可以一次访问多个键，从而提高指令执行效率。而如果这些 key 不在同一个 Redis 节点上，这些指令是不能使用的。

其次，我们可以充分发挥 Redis 的 pipeline 功能。通过 Redis 的 pipeline 功能，可以一次性发送多条指令。当执行完后，这些指令的结果一次性返回。通过这种批次传递和执行指令的方式，Redis 减少了平均每条指令执行时不必要的网络开销，提升了执行效率。同样的，如果这些数据不在同一个 Redis 节点上，我们就不能够使用 pipeline 的功能了。

所以说，将相关数据放在相同的节点上，可以给我们留下更多的优化空间。经过上述的优化设计后，原本需要 24 次 IO 操作的特征计算，最优情况下降低为只需要一次 IO 操作。这就是局部性原理的威力所在！

当然，使用局部性也可能出现数据在集群节点上分布不均匀的问题，所以在选择分区标签时，应该尽量分得更细更均匀些，这样可以减小数据倾斜的问题。

批次请求处理

接下来，我们再来看看“批次请求处理”的方法。批次请求处理是指将多个请求收集起来后，一次性成批处理的过程。**批次请求处理可以有效提高 IO 资源的使用效率，并降低消息的平均处理时延。**

比如，Redis 的 pipeline 功能就是一种批次请求处理的技术。但我们不能仅限于 Redis 的Pipeline 功能。实际上，任何与 IO 相关的操作都可以借鉴这种批次处理的思想，比如 RPC（远程过程调用）、REST 请求、数据库查询等。

在实际开发过程中，将请求做批次化处理本身并不是非常复杂的过程，比较麻烦的是应该怎样将分布在程序中各个地方的请求收集起来。针对这个问题，我们可以使用队列和 CompletableFuture 的异步方案，图 3 描述了这个方案的具体实现方法。

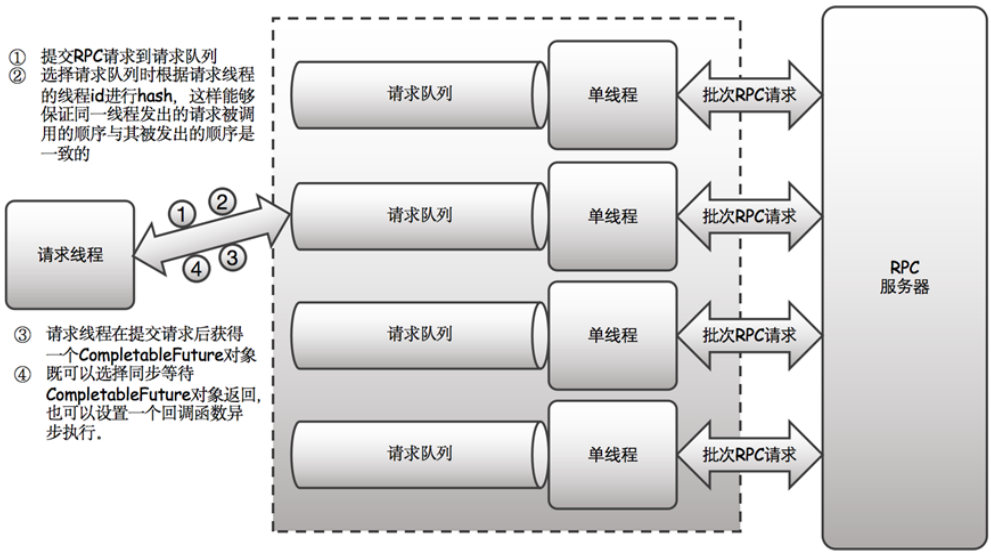

图 3 批次请求实现原理

在上面的图 3 中，当请求发起时，将请求提交给队列后获取一个 CompletableFuture 对象。而另外一个线程，等着从这个队列中取出请求。当该线程取出的请求达到一定数量或者等待超过一定时间时，将取出的这批请求封装成批次请求，发送给请求处理服务器。当批次请求返回后，将批次结果拆解开，再依次使用 CompletableFuture 的 complete 函数将结果交给各个请求发起者。这样就实现了请求的批次化处理。

批次化处理的好处，在于提高了请求处理的吞吐量，降低了每条请求的平均响应时间。但是因为使用了队列和异步的方案，所以也有可能会提高特定某条请求的响应时间。因此在实际开发中，需要你根据具体的使用场景，选择最合适的方案。

基于 Redis 的集群方案到此就介绍完了。现在，我们来看第二种状态集群方案。

基于 Apache Ignite 的状态集群

下面的图 4 是 Apache Ignite 集群用于状态存储和管理的原理图。

图 4 使用 Apache Ignite 集群进行状态存储和管理

从该原理图可以看出，当采用 Apache Ignite 来存储和管理状态时，计算节点和数据节点是耦合在一起的，它们运行在相同的 JVM 内。每个 Apache Ignite 节点会保存全部集群数据中的一部分，流计算节点通过其嵌入的 Apache Ignite 节点来访问状态数据。而 Apache Ignite 作为一种数据网格，其自身的设计和实现机制，会尽可能让计算只需要访问节点本地的数据，从而减少了数据在网络之间的流动。

这种设计方案充分利用了 Apache Ignite 提供的数据网格能力，是一种典型的网格计算架构。

采用 Apache Ignite 数据网格的方案，可以让我们不必过多考虑数据分区问题。Apache Ignite 会自行处理数据局部性以及计算和数据之间亲和性的问题。另外，Apache Ignite 提供的各种计算和查询接口，屏蔽了分布式数据和分布式计算的复杂性，也为我们开发分布式系统带来极大的便利性。网格计算中所有节点都是平等的，当需要水平扩展集群时，只需要将新的节点添加到网格中即可。

不过这种使用数据网格的方案，其成功的地方也是其失败的地方。将计算节点和数据节点耦合在同一个 JVM 后，增加了单一节点的复杂性，同时也使计算资源的分配、管理和监控等变得更加复杂。这点需要你在做方案选型时根据具体场景自行定夺。

基于分布式文件系统的状态集群

第三种状态集群方案，是一种基于分布式文件系统的状态存储和管理集群。这也是一种非常典型的分布式状态存储和管理方案。比如，Flink 的状态存储和管理使用的就是这种方案。下面的图 5 就描述了采用分布式文件系统进行状态存储和管理的方案。

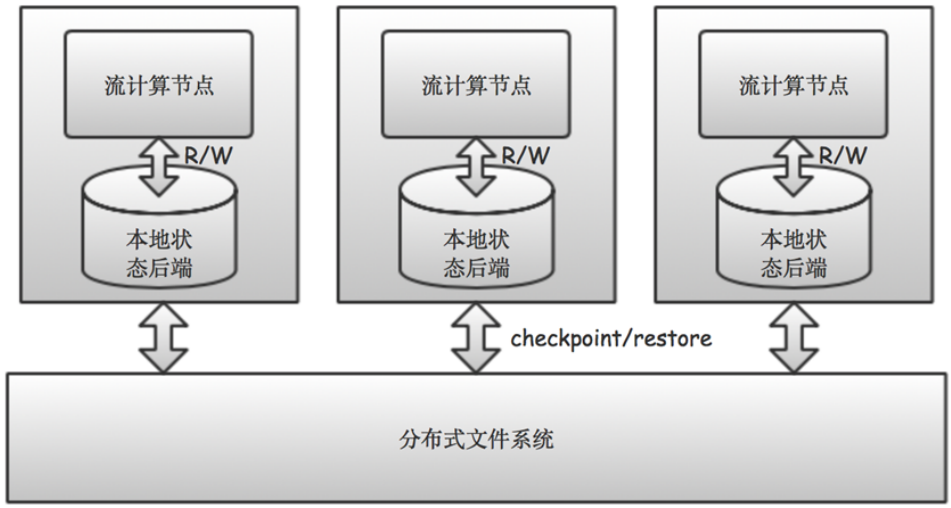

图5 基于分布式文件系统的状态存储和管理集群

在这种分布式状态存储和管理方案中，流计算节点针对状态的操作完全在本地进行，不涉及任何远程操作。但如果只是这样，那当需要扩展或收缩集群的节点数时，怎么保证能够读取到原来的状态信息呢?

因此，在每个节点上，需要有专门的线程定期或在必要的时候（比如任务关闭前），对状态进行 checkpoint。所谓 checkpoint，是指将本地状态后端里的数据做快照（snapshot）之后，保存到分布式文件系统里的过程。当集群在节点数变化后再重启时，各个节点首先从分布式文件系统中读取其所负责数据分片所在的快照，再将快照恢复到状态后端里，这样各个节点就获得重启前的状态数据了，之后的计算又可以完全在本地进行。

这种方案的优势在于，流计算节点对状态的操作在本地完成，不需要任何远程操作。这样本地状态后端的选择可以非常丰富，给性能优化留下极大空间。比如，Flink 目前就已经支持内存、文件系统和 RocksDB 三种状态后端。

不过这种方案也有个缺点，就是不能在运行时动态增加或缩减计算节点数。由于改变计算节点数时，会对状态所在节点进行重新分配，因此需要先做 checkpoint，再从 checkpoint 中恢复状态，这个过程是需要重启流计算应用的，所以也就不能在运行时动态改变计算节点数了。这个问题，对应在 Flink 中，就是当改变算子的并行度（operator parallelism）时，由于相当于改变了计算的节点数，所以需要重启作业。

小结

今天，我们用了三种不同的方式，将流计算中的状态存储扩展为集群。这三种方案代表了不同的分布式系统构建思路。

首先是基于 Redis 的状态存储集群方案。这是一种比较常见的方案，也很容易理解。这种方案代表了一种“计算节点”与“数据节点”分离的分布式系统架构。

然后是基于 Apache Ignite 的状态存储集群方案。这是一种非常经典的方案，但说实话，目前国内用的人较少。这种方案是代表了一种“计算网格”和“数据网格”的分布式系统架构。虽然国内用得少，但是它在整个软件历史上，都是一种非常重要的分布式系统架构方案。

最后是基于分布式文件系统的状态存储集群方案，这是一种非常新颖的方案。而且，它最重要的意义在于，将“内存”的概念扩展到了磁盘。在这种方案中，如果你将本地状态后端视为“内存”的话，那整个系统不就是一种分布式的冯诺伊曼机器吗？在这个分布式的冯诺伊曼机器中，CPU 和内存都是可以无上限的水平扩展的，是不是非常惊艳！

总的来说，这三种方法都是非常经典的分布式系统构建方法。希望你能够掌握它们，以后构建分布式系统时，一定会有所帮助的。

最后留一个小作业，Flink 中的 Keyed State 是怎样分布在 Flink 集群各个计算节点上的呢？将你的想法或问题写在留言区，我看到后会进行分析和解答。

下面是本课时的知识脑图，以便于你理解。