

Apache Flink

十大技术难点实战

深度解析 Flink 生产环境常见难题与优化实战

- 集群规划的 **科学算法**
- Flink SQL应用的 **demo 演示**
- 生产环境常见问题 **排查与解法**
- Flink 生态应用 **原理与实践**



扫一扫二维码图案，关注我吧



开发者社区



阿里云实时计算



实时计算交流钉钉群



Flink 社区微信公众号

| 目录

102 万行代码，1270 个问题，Flink 新版发布了什么？	4
从开发到生产上线，如何确定集群规划大小？	11
Demo：基于 Flink SQL 构建流式应用	22
Flink Checkpoint 问题排查实用指南	37
如何分析及处理 Flink 反压？	48
Flink on YARN（上）：一张图轻松掌握基础架构与启动流程	56
Flink on YARN（下）：常见问题与排查思路	64
Apache Flink 与 Apache Hive 的集成	72
Flink Batch SQL 1.10 实践	83
如何在 PyFlink 1.10 中自定义 Python UDF？	90
Flink 1.10 Native Kubernetes 原理与实践	107

102 万行代码，1270 个问题， Flink 新版发布了什么？

作者：李钰（绝顶） | 阿里巴巴高级技术专家

导读：Apache Flink 是公认的新一代开源大数据计算引擎，可以支持流处理、批处理和机器学习等多种计算形态，也是 Apache 软件基金会和 GitHub 社区最为活跃的项目之一。

2019 年 1 月，阿里巴巴实时计算团队宣布将经过双十一历练和集团内部业务打磨的 Blink 引擎进行开源并向 Apache Flink 贡献代码，此后的一年中，阿里巴巴实时计算团队与 Apache Flink 社区密切合作，持续推进 Flink 对 Blink 的整合。

2 月 12 日，Apache Flink 1.10.0 正式发布，在 Flink 的第一个双位数版本中正式完成了 Blink 向 Flink 的合并。在此基础之上，Flink 1.10 版本在生产可用性、功能、性能上都有大幅提升。本文将详细为大家介绍该版本的重大变更与新增特性。

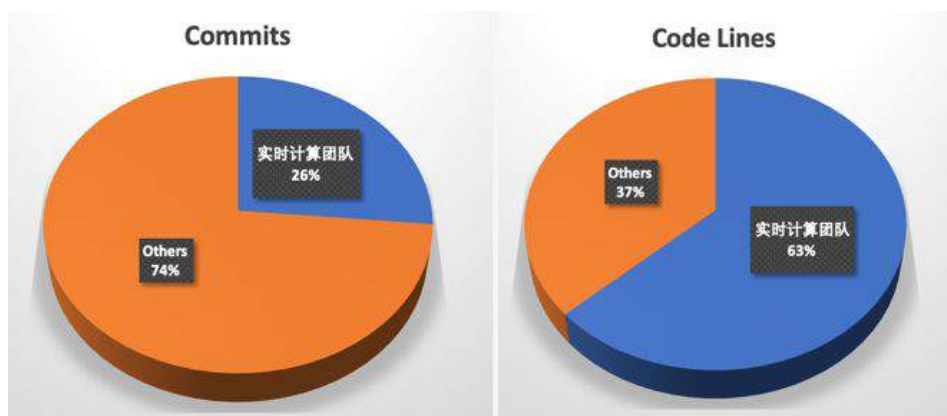
Flink 1.10 是迄今为止规模最大的一次版本升级，除标志着 Blink 的合并完成外，还实现了 Flink 作业的整体性能及稳定性的显著优化、对原生 Kubernetes 的初步集成以及对 Python 支持 (PyFlink) 的重大优化等。

综述

Flink 1.10.0 版本一共有 218 名贡献者，解决了 1270 个 JIRA issue，经由 2661 个 commit 总共提交了超过 102 万行代码，多项数据对比之前的几个版本都有所提升，印证着 Flink 开源社区的蓬勃发展。

	1.7.0版本	1.8.0版本	1.9.0版本	1.10.0版本
解决的问题数量	428	422	977	1270
代码提交次数	969	1094	1964	2661
贡献者人数	112	140	190	218

其中阿里巴巴实时计算团队共提交 64.5 万行代码，超过总代码量的 60%，做出了突出的贡献。

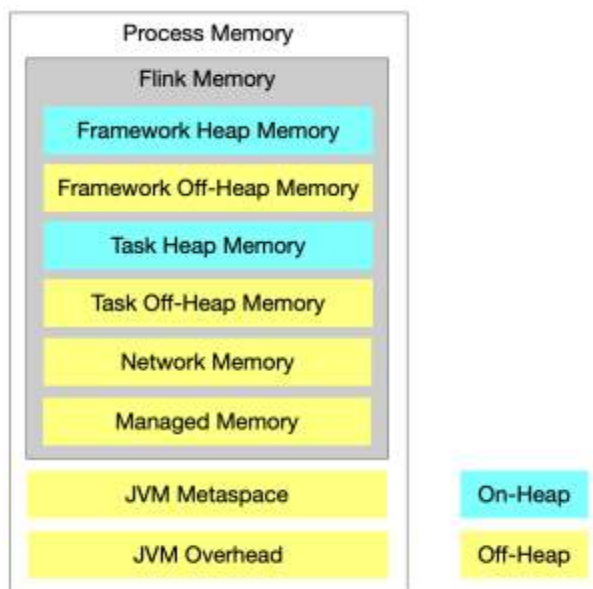


在该版本中，Flink 对 SQL 的 DDL 进行了增强，并实现了生产级别的 Batch 支持和 Hive 兼容，其中 TPC-DS 10T 的性能更是达到了 Hive 3.0 的 7 倍之多。在内核方面，对内存管理进行了优化。在生态方面，增加了 Python UDF 和原生 Kubernetes 集成的支持。后续章节将在这些方面分别进行详细介绍。

内存管理优化

在旧版本的 Flink 中，流处理和批处理的内存配置是割裂的，并且当流式作业配置使用 RocksDB 存储状态数据时，很难限制其内存使用，从而在容器环境下经常出现内存超用被杀的情况。

在 1.10.0 中，我们对 Task Executor 的内存模型，尤其是受管理内存 (Managed Memory) 进行了大幅度的改进 (FLIP-49)，使得内存配置对用户更加清晰：



此外, 我们还将 RocksDB state backend 使用的内存纳入了托管范畴, 同时可以通过简单的配置来指定其能使用的内存上限和读写缓存比例 (FLINK-7289)。如下图所示, 在实际测试当中受控前后的内存使用差别非常明显。



受控前的内存使用情况 (share-slot)



受控后的内存使用情况 (share-slot)

Batch 兼容 Hive 且生产可用

Flink 从 1.9.0 版本开始支持 Hive 集成，但并未完全兼容。在 1.10.0 中我们对 Hive 兼容性做了进一步的增强，使其达到生产可用的标准。具体来说，Flink 1.10.0 中支持：

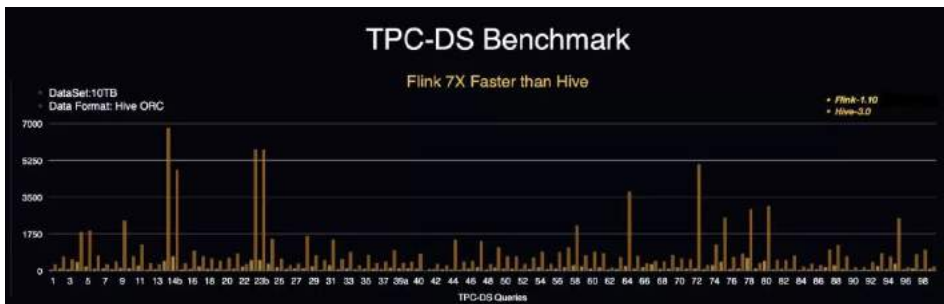
- **Meta 兼容** – 支持直接读取 Hive catalog，覆盖 Hive 1.x/2.x/3.x 全部版本
- **数据格式兼容** – 支持直接读取 Hive 表，同时也支持写成 Hive 表的格式；支持分区表
- **UDF 兼容** – 支持在 Flink SQL 内直接调用 Hive 的 UDF，UDTF 和 UDAF

与此同时，1.10.0 版本中对 batch 执行进行了进一步的优化 (FLINK-14133)，主要包括：

- 向量化读取 ORC (FLINK-14135)
- 基于比例的弹性内存分配 (FLIP-53)
- Shuffle 的压缩 (FLINK-14845)

- 基于新调度框架的优化 (FLINK-14735)

在此基础上将 Flink 作为计算引擎访问 Hive 的 meta 和数据，在 TPC-DS 10T benchmark 下性能达到 Hive 3.0 的 7 倍以上。



SQL DDL 增强

Flink 1.10.0 支持在 SQL 建表语句中定义 watermark 和计算列，以 watermark 为例：

```
CREATE TABLE table_name (  
    WATERMARK FOR columnName AS <watermark_strategy_expression>  
) WITH (  
    ...  
)
```

除此之外，Flink 1.10.0 还在 SQL 中对临时函数 / 永久函数以及系统 / 目录函数进行了明确区分，并支持创建目录函数、临时函数以及临时系统函数：

```
CREATE [TEMPORARY|TEMPORARY SYSTEM] FUNCTION  
[IF NOT EXISTS] [catalog_name.] [db_name.] function_name  
AS identifier [LANGUAGE JAVA|SCALA]
```

Python UDF 支持

Flink 从 1.9.0 版本开始增加了对 Python 的支持 (PyFlink)，但用户只能使用

Java 开发的 User-defined-function (UDF)，具有一定的局限性。在 1.10.0 中我们为 PyFlink 增加了原生 UDF 支持 (FLIP-58)，用户现在可以在 Table API/SQL 中注册并使用自定义函数，如下图所示：

```
* if os.path.exists(sink_path):
*     if os.path.isfile(sink_path):
*         os.remove(sink_path)
*     else:
*         shutil.rmtree(sink_path)
* s_env.set_parallelism(1)
* t = st_env.from_elements([(1, 'hi', 'hello'), (2, 'hi', 'hello')], ['a', 'b', 'c'])
)
* st_env.connect(FileSystem().path(sink_path)) \
*     .with_format(OldCsv()
*         .field_delimiter(',')
*         .field("a", DataTypes.BIGINT())
*         .field("b", DataTypes.STRING())
*         .field("c", DataTypes.STRING())) \
*     .with_schema(Schema()
*         .field("a", DataTypes.BIGINT())
*         .field("b", DataTypes.STRING())
*         .field("c", DataTypes.STRING())) \
*     .register_table_sink("stream_sink")
* t.select("a + 1, b, c").insert_into("stream_sink")
* st_env.execute("stream_job")
>>>
```

同时也可以方便的通过 pip 安装 PyFlink：

```
pip install apache-flink
```

更多详细介绍，请参考：

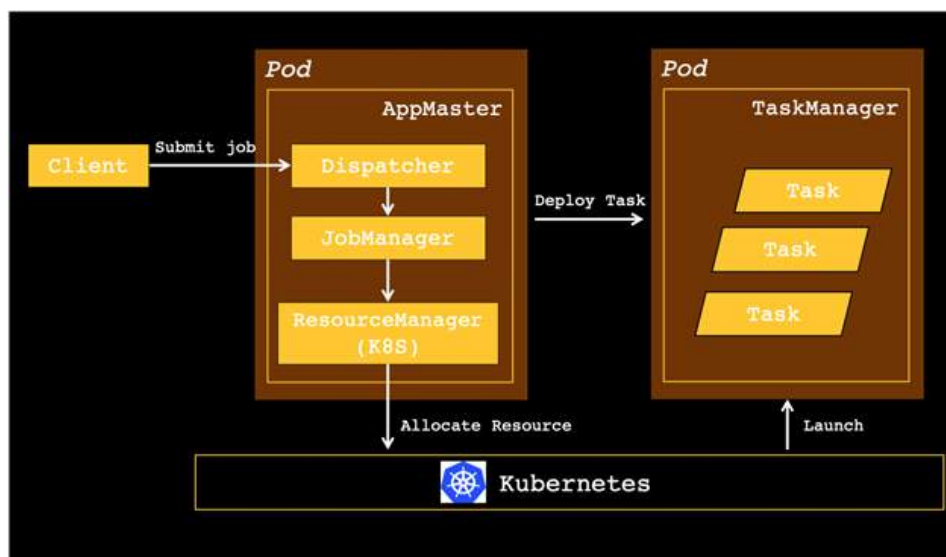
<https://enjoyment.cool/2020/02/19/Deep-dive-how-to-support-Python-UDF-in-Apache-Flink-1-10/>

原生 Kubernetes 集成

Kubernetes (K8S) 是目前最为流行的容器编排系统，也是目前最流行的容器化应用发布平台。在旧版本当中，想要在 K8S 上部署和管理一个 Flink 集群比较复杂，需要对容器、算子及 kubectl 等 K8S 命令有所了解。

在 Flink 1.10 中，我们推出了对 K8S 环境原生支持 (FLINK-9953)，Flink

的资源管理器会主动和 Kubernetes 通信，按需申请 pod，从而可以在多租户环境中以较少的资源开销启动 Flink，使用起来也更加的方便。



更多内容，参考 1.10.0 版本发布日志：

<https://ci.apache.org/projects/flink/flink-docs-stable/release-notes/flink-1.10.html>

结语

2019 年 1 月，阿里巴巴实时计算团队宣布 Blink 开源。整整一年之后，Flink 1.10.0 版本的发布宣告 Flink 和 Blink 的整合正式完成。我们践行着自己的诺言，开放源码，更相信社区的力量，相信社区是开源协作精神与创新的摇篮。我们也衷心希望有更多志同道合的小伙伴加入我们，一起把 Apache Flink 做的越来越好！

从开发到生产上线，如何确定集群规划大小？

作者：Robert Metzger 翻译：毛家琦 校对：秦江杰

在 Flink 社区中，最常被问到的问题之一是：在从开发到生产上线的过程中如何确定集群的大小。这个问题的标准答案显然是“视情况而定”，但这并非一个有用的答案。本文概述了一系列的相关问题，通过回答这些问题，或许你能得出一些数字作为指导和参考。

计算并建立一个基线

第一步是仔细考虑应用程序的运维指标，以达到所需资源的基线。

需要考虑的关键指标是：

- 每秒记录数和每条记录的大小
- 已有的不同键 (key) 的数量和每个键对应的状态大小
- 状态更新的次数和状态后端的访问模式

最后，一个更实际的问题是与客户之间围绕停机时间、延迟和最大吞吐量的服务级别协议 (sla)，因为这些直接影响容量规划。

接下来，根据预算，看看有什么可用的资源。例如：

- **网络容量**，同时把使用网络的外部服务也纳入考虑，如 Kafka、HDFS 等。
- **磁盘带宽**，如果您依赖于基于磁盘的状态后端，如 RocksDB (并考虑其他磁盘使用，如 Kafka 或 HDFS)
- **可用的机器数量、CPU 和内存**

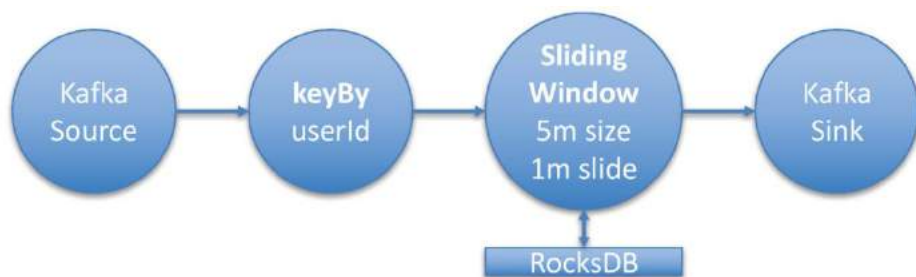
基于所有这些因素，现在可以为正常运行构建一个基线，外加一个资源缓冲量用于恢复追赶或处理负载尖峰。建议您在建立基线时也考虑检查点期间 (checkpoint-

ing) 使用的资源情况。

示例：数据说明

当前在假设的集群上计划作业部署，将建立资源使用基线的过程可视化。这些数字是粗略的值，它们并不全面——在文章的最后将进一步说明在进行计算过程中遗漏的部分。

Flink 流计算作业和硬件示例



Flink 流计算作业拓扑示例

在本案例中，我将部署一个典型的 Flink 流处理作业，该作业使用 Flink 的 Kafka 数据消费者从 Kafka 消息源中读取数据。然后使用带键的总计窗口运算符 (window operator) 进行转换运算。窗口运算符在时间窗口 5 分钟执行聚合。由于总是有新的数据，故将把窗口配置为 1 分钟的滑动窗口 (sliding window)。

这意味着将在每分钟更新过去 5 分钟的聚合量。流计算作业为每个用户 id 创建一个合计量。从 Kafka 消息源消费的每条消息大小 (平均) 为 2 kb。

假设吞吐量为每秒 100 万条消息。要了解窗口运算符 (window operator) 的状态大小，需要知道不同键的数目。在本例中，键 (keys) 是用户 id 的数量，即 500000000 个不同的用户。对于每个用户，需要计算四个数字，存储为长整数 (8 字节)。

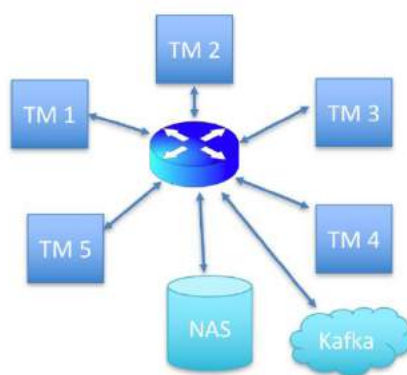
总结一下工作的关键指标：

- 消息大小：2 KB
- 吞吐量：1000000 msg/ 秒
- 不同键数量：500000000（窗口聚合：每个键 4 个长整形）
- Checkpointing：每分钟一次。

■ Hardware:

- 5 machines
- 10 gigabit Ethernet
- Each machine running a Flink TaskManager
- Disks are attached via the network

■ Kafka is separate



假定的硬件设置

如上图所示，共有五台机器在运行作业，每台机器运行一个 Flink 任务管理器（Flink 的工作节点）。磁盘是通过网络相互连接的（这在云设置中很常见），从主交换机到运行 TaskManager 的每台计算机都由一个 10 千兆位以太网连接。Kafka 缓存代理（brokers）在不同的机器上分开运行。

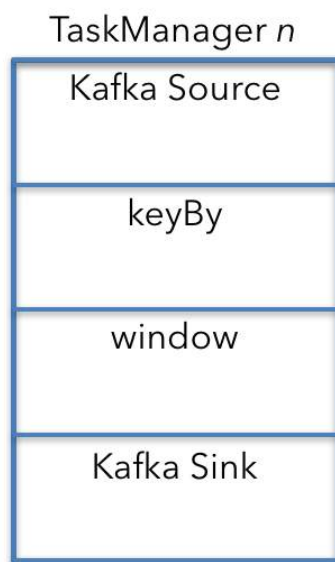
每台机器有 16 个 CPU 核。为了简化处理，不考虑 CPU 和内存需求。但实际情况中，根据应用程序逻辑和正在使用的状态后端，我们需要注意内存。这个例子使用了一个基于 RocksDB 的状态后端，它稳定并且内存需求很低。

从单独的一台机器的视角

要了解整个作业部署的资源需求，最容易的方法是先关注一台计算机和一个 TaskManager 中的操作。然后，可以使用一台计算机的数字来计算总体资源需求量。

默认情况下（如果所有运算符具有相同的并行度并且没有特殊的调度限制），流作业的所有运算符都在每一台计算机上运行。

在这种情况下，Kafka 源（或消息消费者）、窗口运算符和 Kafka 发送端（或消息生产者）都在这五台机器上运行。



机器视图图 - TaskManager n

从上图来看，keyBy 是一个单独运算符，因此计算资源需求更容易。实际上，keyBy 是一个 API 构造，并转换为 Kafka source 和窗口运算符（window operator）之间连接的配置属性。

以下将自上而下地分析（上图）这些运算符，了解他们的网络资源需求。

The Kafka source

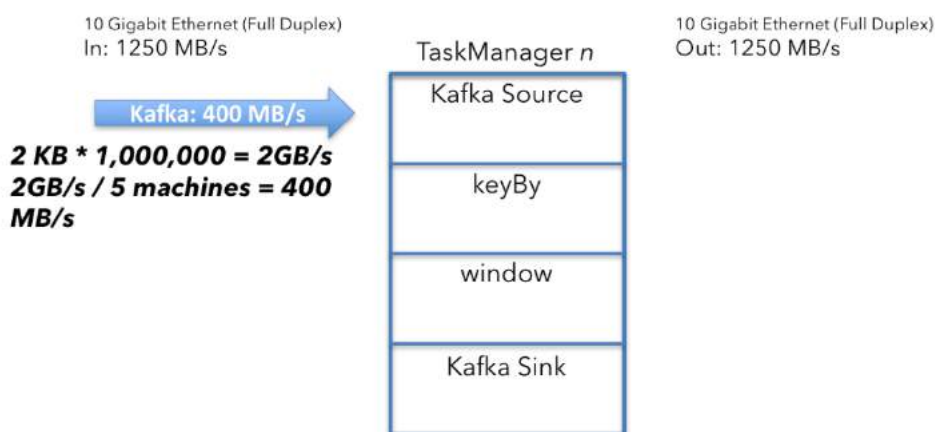
要计算单个 Kafka 源（source）接收的数据量，我们首先计算 Kafka 的合计输入。这些 source 每秒接收 1000000 条消息，每条消息大小为 2 KB。

$$2 \text{ KB} \times 1,000,000/\text{s} = 2 \text{ GB/s}$$

将 2 GB/s 除以机器数 (5) 得到以下结果：

$$2 \text{ GB/s} \div 5 \text{ 台机器} = 400 \text{ MB/s}$$

群集中运行的 5 个 Kafka 源中的每一个都接收平均吞吐量为 400 MB/s 的数据结果。



Kafka source 的计算过程

The Shuffle / keyBy

接下来，需要确保具有相同键（在本例中为用户 id）的所有事件都在同一台计算机上结束。正在读取的 Kafka 消息源的数据（在 Kafka 中）可能会根据不同的分区方案进行分区。

Shuffle 过程将具有相同键的所有数据发送到一台计算机，因此需要将来自 Kafka 的 400 MB/s 数据流拆分为一个 user id 分区流：

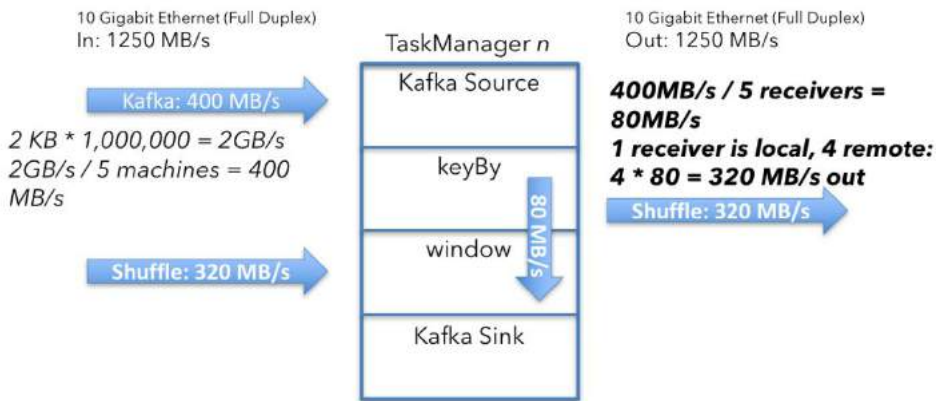
$$400 \text{ MB/s} \div 5 \text{ 台机器} = 80 \text{ MB/s}$$

平均而言，我们必须向每台计算机发送 80 MB/s 的数据。此分析是从一台机器

的角度进行的，这意味着某些数据已经在指定的目标机器运行了，因此减去 80 MB/s 即可：

$$400 \text{ MB/s} - 80 \text{ MB} = 320 \text{ MB/s}$$

可以得到结果：每台机器以 320 MB/s 的速率接收和发送用户数据。



The shuffle 的计算过程

- Window 窗口输出和 Kafka 发送

下一个要问的问题是窗口运算符发出多少数据并发送到 Kafka 接收器。答案是 67 MB/s，我们来解释一下我们是怎么得到这个数字的。

窗口运算符为每个键 (key) 保留 4 个数字 (表示为长整形) 的聚合值。运算符每分钟发出一次当前聚合总值。每个键从聚合中发出 2 个整形 (user_id, window_ts) 和 4 个长整形：

$$(2 \times 4 \text{ 字节}) + (4 \times 8 \text{ 字节}) = \text{每个键 } 40 \text{ 字节}$$

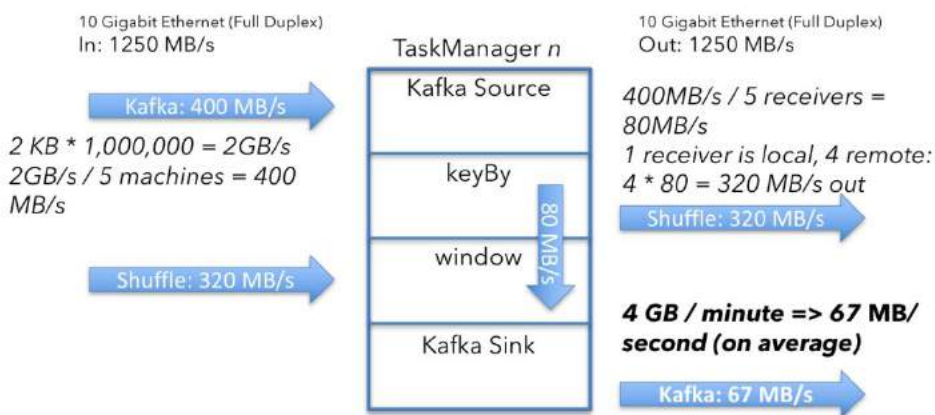
然后将键的总数 (500000000 除以机器数量) 计算在内：

$$100000000 \text{ 个 keys} \times 40 \text{ 个字节} = 4 \text{ GB (从每台机器来看)}$$

然后计算每秒大小：

$4 \text{ GB} / \text{分钟} \div 60 = 67 \text{ MB} / \text{秒}$ (由每个任务管理器发出)

这意味着每个任务管理器平均从窗口运算符发出 67 MB/s 的用户数据。由于每个任务管理器上都有一个 Kafka 发送端 (和窗口运算符在同一个任务管理器中)，并且没有进一步的重新分区，所以这得到的是 Flink 向 Kafka 发送的数据量。

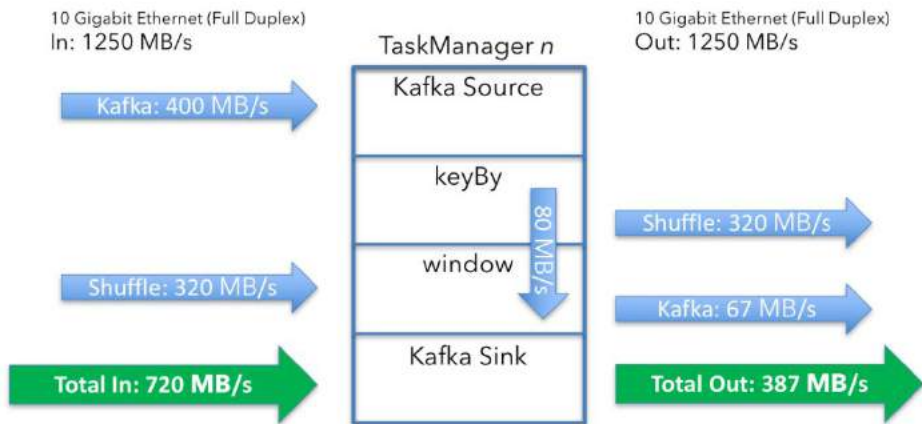


用户数据：从 Kafka，分发到窗口运算符并返回到 Kafka

窗口运算符的数据发射预计将是“突发”的，因为它们每分钟发送一次数据。实际上，运算符不会以 67 mb/s 的恒定速率给客户发送数据，而是每分钟内将可用带宽最大化几秒钟。

这些总计为：

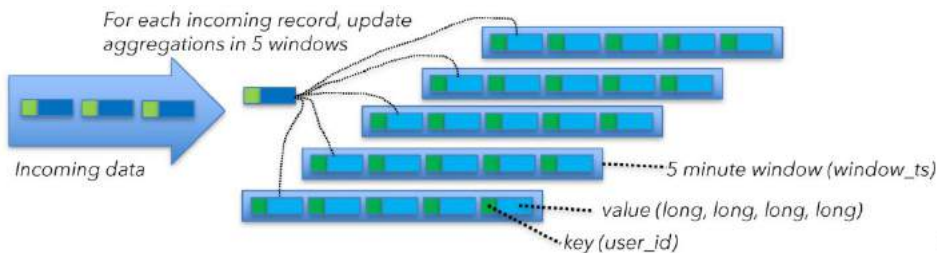
- **数据输入：** 每台机器 720 MB/s (400+320)
- **数据输出：** 每台机器 387 MB/s (320+67)



- 状态访问和检查点

这不是全部的（内容）。到目前为止，我只查看了 Flink 正在处理的用户数据。在实际情况中需要计入从磁盘访问的开销，包括到 RocksDB 的存储状态和检查点。要了解磁盘访问成本，请查看窗口运算符（window operator）如何访问状态。Kafka 源也保持一定的状态，但与窗口运算符相比，它可以忽略不计。

要了解窗口运算符（window operator）的状态大小，需要从不同的角度进行查看。Flink 正在用 1 分钟的滑动窗口计算 5 分钟的窗口量。Flink 通过维护五个窗口来实现滑动窗口，每次滑动都对应一个 1 分钟的窗口。如前所述，当使用窗口实现即时聚合时，将为每个窗口中的每个键（key）维护 40 字节的状态。对于每个传入事件，首先需要从磁盘检索当前聚合值（读取 40 字节），更新聚合值，然后将新值写回（写入 40 字节）。



窗口状态

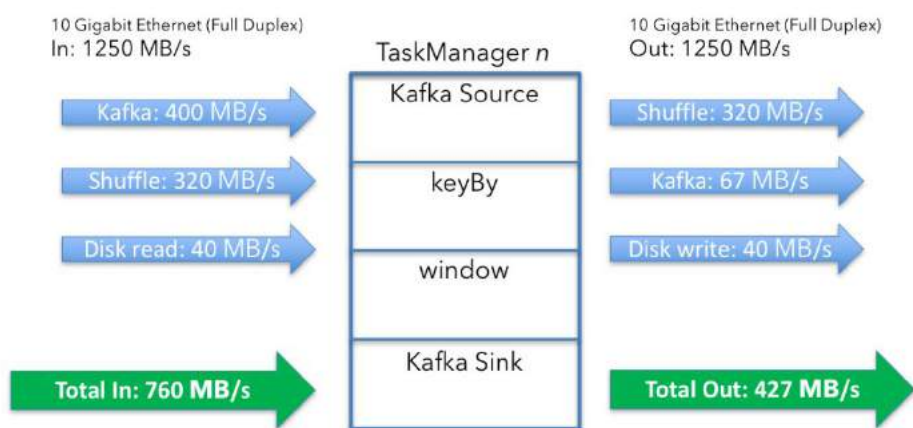
这意味着：

40 字节状态 x 5 个窗口 x 每台计算机 $200000 \text{ msg/s} = 40 \text{ MB/s}$

即需要的每台计算机的读或写磁盘访问权限。如前所述，磁盘是网络相互连接的，因此需要将这些数字添加到总吞吐量计算中。

现在总数是：

- **数据输入：** 760 MB/s (400 MB/s 数据输入 + 320 MB/s 随机播放 + 40 MB/s 状态)
- **数据输出：** 427 MB/s (320 MB/s 随机播放 + 67 MB/s 数据输出 + 40 MB/s 状态)



上述考虑是针对状态访问的，当新事件到达窗口运算符时，状态访问会持续进行，还需要容错启用检查点。如果机器或其他部分出现故障，需要恢复窗口内容并继续处理。

检查点设置为每分钟一个检查点，每个检查点将作业的整体状态复制到网络连接的文件系统中。

让我们一起来看看每台计算机上的整个状态有多大：

40 字节状态 x 5 个窗口 x 100000000 个 keys = 20 GB

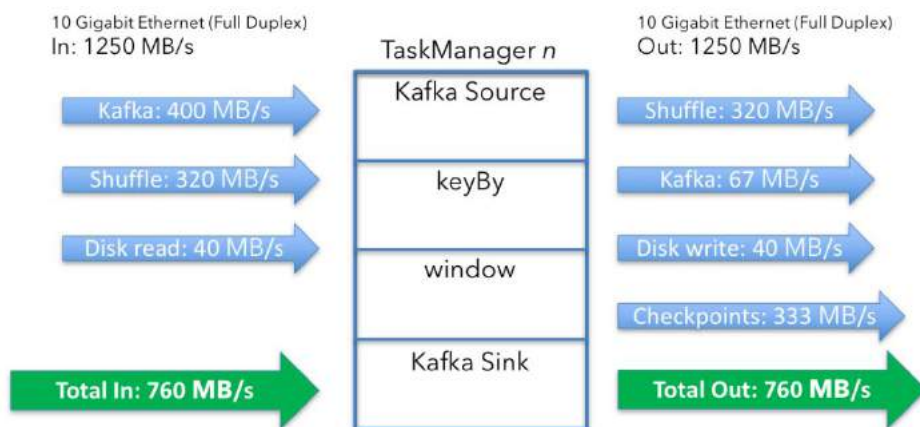
并且，要获得每秒的值：

$20 \text{ GB} \div 60 = 333 \text{ MB/秒}$

与窗口运算类似，检查点是突发的，每分钟一次，它都试图将数据全速发送到外部存储器。Checkpointing 引发对 RocksDB 的额外状态访问（在本案例中，RocksDB 位于网络连接的磁盘上）。自 Flink 1.3 版本以来，RocksDB 状态后端支持增量 checkpoint，概念上通过仅发送自上一个 checkpoint 以来的变化量，减少了每个 checkpoint 上所需的网络传输，但本例中不使用此功能。

这会将总数更新为：

- 数据输入：760 MB/s (400+320+40)
- 数据输出：760 MB/s (320+67+40+333)



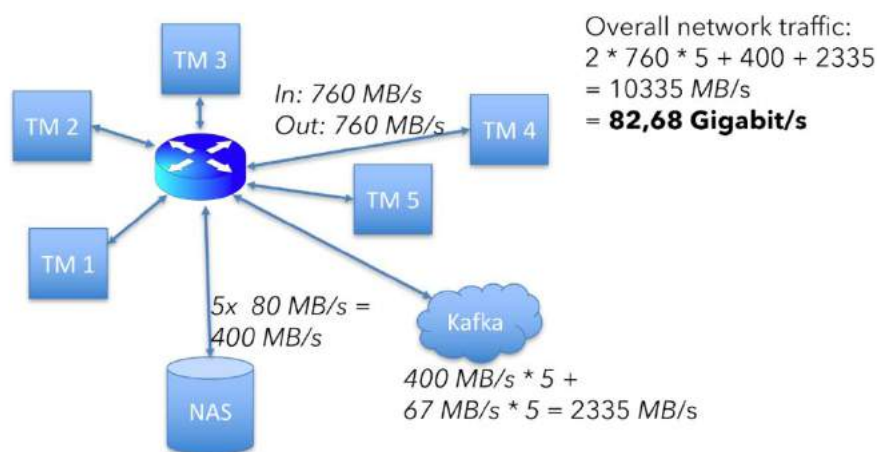
这意味着整个网络流量为：

$760 + 760 \times 5 + 400 + 2335 = 10335 \text{ MB/秒}$

400 是 5 台机器上 80 MB 状态访问（读写）进程的总和，2335 是集群上 Kafka

输入和输出进程的总和。

这大概是上图所示硬件设置中可用网络容量的一半以上。



联网要求

补充一点，这些计算都不包括协议开销，例如来自 Flink、Kafka 或文件系统的 TCP、Ethernet 和 RPC 调用。但这仍然是一个很好的出发点，可以帮助您了解工作所需的硬件类型，以及性能指标。

扩展方法

基于以上分析，这个例子，在一个 5 节点集群的典型运行中，每台机器都需要处理 760 个 Mb/s 的数据，无论是输入还是输出，从 1250 Mb/s 的总容量来看，它保留了大约 40% 的网络容量因为部分被主观所简化的复杂因素，例如网络协议开销、从检查点恢复事件重放期间的重载，以及由数据歪斜引起的跨集群的负载不平衡。

对于 40% 的净空是否合适，没有一个一刀切的答案，但是这个算法应该是一个很好的起点。尝试上面的计算，更换机器数量、键 (keys) 的数量或每秒的消息数，选择要考虑的运维指标，然后将其与您的预算和运维因素相平衡。

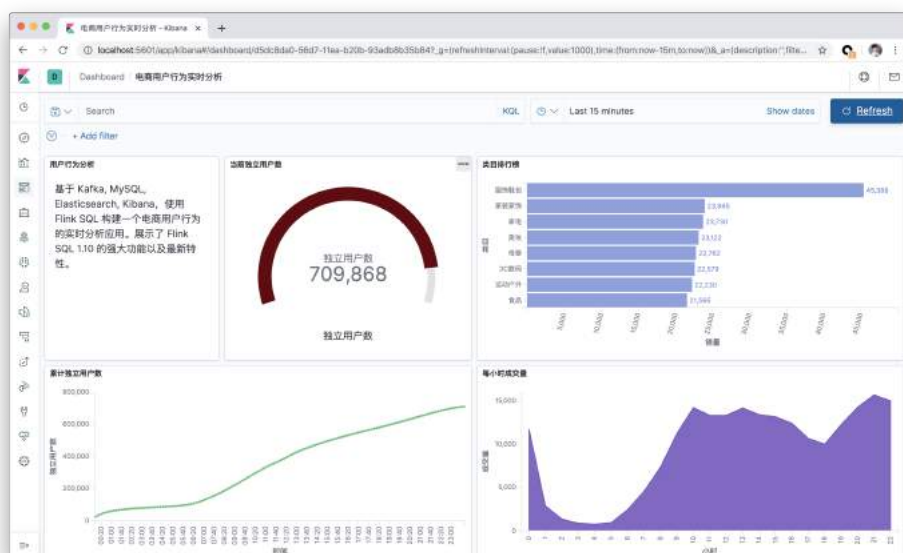
| Demo: 基于 Flink SQL 构建流式应用

作者: 伍翀 (云邪) | Apache Flink PMC, 阿里巴巴技术专家

上周四在 Flink 中文社区钉钉群中直播分享了《Demo: 基于 Flink SQL 构建流式应用》, 直播内容偏向实战演示。这篇文章是对直播内容的一个总结, 并且改善了部分内容, 比如除 Flink 外其他组件全部采用 Docker Compose 安装, 简化准备流程。读者也可以结合视频和本文一起学习。完整分享可以观看视频回顾: <https://www.bilibili.com/video/av90560012>

Flink 1.10.0 于近期刚发布, 释放了许多令人激动的新特性。尤其是 Flink SQL 模块, 发展速度非常快, 因此本文特意从实践的角度出发, 带领大家一起探索使用 Flink SQL 如何快速构建流式应用。

本文将基于 Kafka, MySQL, Elasticsearch, Kibana, 使用 Flink SQL 构建一个电商用户行为的实时分析应用。本文所有的实战演练都将在 Flink SQL CLI 上执行, 全程只涉及 SQL 纯文本, 无需一行 Java/Scala 代码, 无需安装 IDE。本实战演练的最终效果图:



准备

一台装有 Docker 和 Java8 的 Linux 或 MacOS 计算机。

使用 Docker Compose 启动容器

本实战演示所依赖的组件全都编排到了容器中，因此可以通过 `docker-compose` 一键启动。你可以通过 `wget` 命令自动下载该 `docker-compose.yml` 文件，也可以手动下载。

```
mkdir flink-demo; cd flink-demo;
wget https://raw.githubusercontent.com/wuchong/flink-sql-demo/master/docker-
compose.yml
```

该 Docker Compose 中包含的容器有：

- **DataGen**: 数据生成器。容器启动后会自动开始生成用户行为数据，并发送到 Kafka 集群中。默认每秒生成 1000 条数据，持续生成约 3 小时。也可以更改

`docker-compose.yml` 中 datagen 的 `speedup` 参数来调整生成速率 (重启 docker compose 才能生效)。

- **MySQL**: 集成了 MySQL 5.7 , 以及预先创建好了类目表 (`category`), 预先填入了子类目与顶级类目的映射关系, 后续作为维表使用。
- **Kafka**: 主要用作数据源。DataGen 组件会自动将数据灌入这个容器中。
- **Zookeeper**: Kafka 容器依赖。
- **Elasticsearch**: 主要存储 Flink SQL 产出的数据。
- **Kibana**: 可视化 Elasticsearch 中的数据。

在启动容器前, 建议修改 Docker 的配置, 将资源调整到 4GB 以及 4 核。启动所有的容器, 只需要在 `docker-compose.yml` 所在目录下运行如下命令。

```
docker-compose up -d
```

该命令会以 detached 模式自动启动 Docker Compose 配置中定义的所有容器。你可以通过 `docker ps` 来观察上述的五个容器是否正常启动了。也可以访问 <http://localhost:5601/> 来查看 Kibana 是否运行正常。

另外可以通过如下命令停止所有的容器:

```
docker-compose down
```

下载安装 Flink 本地集群

我们推荐用户手动下载安装 Flink, 而不是通过 Docker 自动启动 Flink。因为这样可以更直观地理解 Flink 的各个组件、依赖、和脚本。

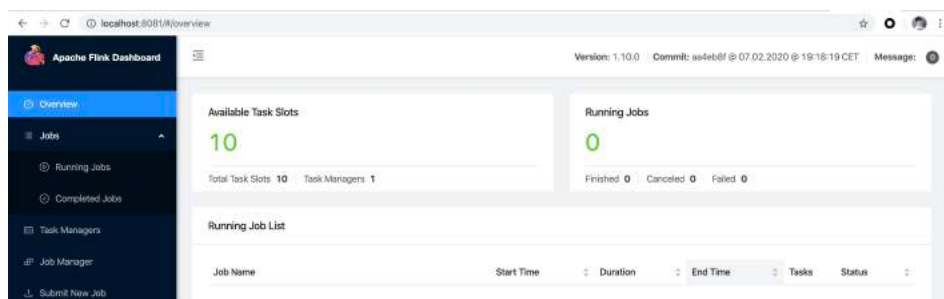
1. 下载 Flink 1.10.0 安装包并解压 (解压目录 `flink-1.10.0`): https://www.apache.org/dist/flink/flink-1.10.0/flink-1.10.0-bin-scala_2.11.tgz
2. 进入 `flink-1.10.0` 目录: `cd.flink-1.10.0`
3. 通过如下命令下载依赖 jar 包, 并拷贝到 `lib/` 目录下, 也可手动下载和拷

贝。因为我们运行时需要依赖各个 connector 实现。

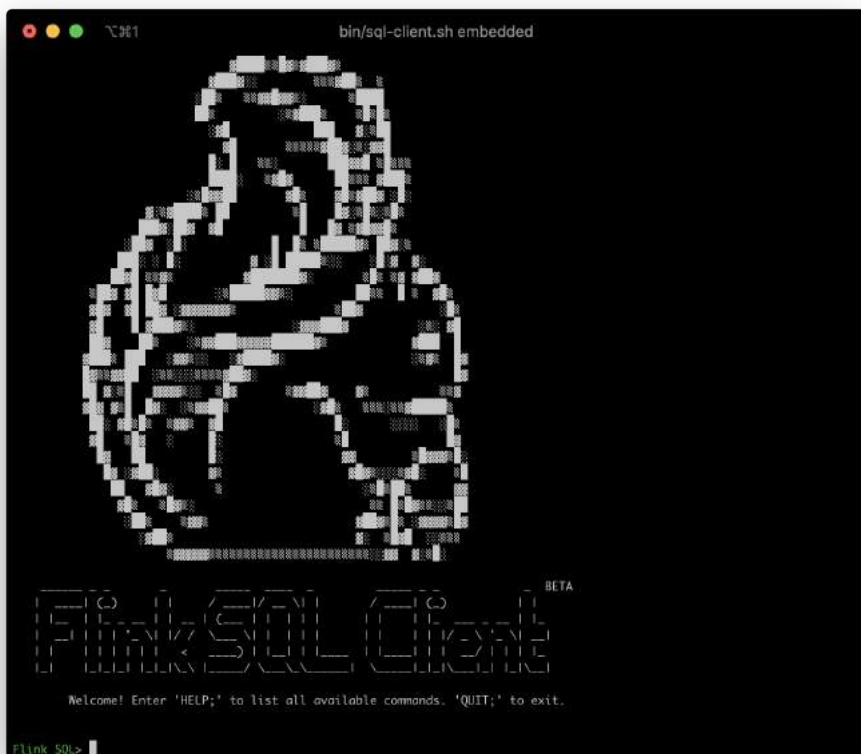
```
wget -P ./lib/ https://repo1.maven.org/maven2/org/apache/flink/flink-  
json/1.10.0/flink-json-1.10.0.jar | \  
wget -P ./lib/ https://repo1.maven.org/maven2/org/apache/flink/flink-  
sql-connector-kafka_2.11/1.10.0/flink-sql-connector-  
kafka_2.11-1.10.0.jar | \  
wget -P ./lib/ https://repo1.maven.org/maven2/org/apache/flink/flink-  
sql-connector-elasticsearch6_2.11/1.10.0/flink-sql-  
connector-elasticsearch6_2.11-1.10.0.jar | \  
wget -P ./lib/ https://repo1.maven.org/maven2/org/apache/flink/flink-  
jdbc_2.11/1.10.0/flink-jdbc_2.11-1.10.0.jar | \  
wget -P ./lib/ https://repo1.maven.org/maven2/mysql/mysql-connector-  
java/5.1.48/mysql-connector-java-5.1.48.jar
```

4. 将 `conf/flink-conf.yaml` 中的 `taskmanager.numberOfTaskSlots` 修改成 10，因为我们会同时运行多个任务。
5. 执行 `./bin/start-cluster.sh`，启动集群。

运行成功的话，可以在 <http://localhost:8081> 访问到 Flink Web UI。并且可以看到可用 Slots 数为 10 个。



6. 执行 `bin/sql-client.sh embedded` 启动 SQL CLI。便会看到如下的松鼠欢迎界面。



使用 DDL 创建 Kafka 表

Datagen 容器在启动后会往 Kafka 的 `user_behavior` topic 中持续不断地写入数据。数据包含了 2017 年 11 月 27 日一天的用户行为（行为包括点击、购买、加购、喜欢），每一行表示一条用户行为，以 JSON 的格式由用户 ID、商品 ID、商品类目 ID、行为类型和时间组成。该原始数据集来自[阿里云天池公开数据集](#)，特此鸣谢。

我们可以在 `docker-compose.yml` 所在目录下运行如下命令，查看 Kafka 集群中生成的前 10 条数据。


```
docker-compose exec kafka bash -c 'kafka-console-consumer.sh --topic user_
behavior --bootstrap-server kafka:9094 --from-beginning --max-messages 10'
```

```
{"user_id": "952483", "item_id": "310884", "category_id": "4580532",
  "behavior": "pv", "ts":
"2017-11-27T00:00:00Z"}
{"user_id": "794777", "item_id": "5119439", "category_id": "982926",
  "behavior": "pv", "ts":
"2017-11-27T00:00:00Z"}
...
```

有了数据源后，我们就可以用 DDL 去创建并连接这个 Kafka 中的 topic 了。在 Flink SQL CLI 中执行该 DDL。

```
CREATE TABLE user_behavior (
  user_id BIGINT,
  item_id BIGINT,
  category_id BIGINT,
  behavior STRING,
  ts TIMESTAMP(3),
  proctime as PROCTIME(), -- 通过计算列产生一个处理时间列
  WATERMARK FOR ts as ts - INTERVAL '5' SECOND -- 在 ts 上定义 watermark, ts 成
为事件时间列
) WITH (
  'connector.type' = 'kafka', -- 使用 kafka connector
  'connector.version' = 'universal', -- kafka 版本, universal 支持 0.11 以上
的版本
  'connector.topic' = 'user_behavior', -- kafka topic
  'connector.startup-mode' = 'earliest-offset', -- 从起始 offset 开始读取
  'connector.properties.zookeeper.connect' = 'localhost:2181', --
zookeeper 地址
  'connector.properties.bootstrap.servers' = 'localhost:9092', -- kafka
broker 地址
  'format.type' = 'json' -- 数据源格式为 json
);
```

如上我们按照数据的格式声明了 5 个字段，除此之外，我们还通过计算列语法和 PROCTIME() 内置函数声明了一个产生处理时间的虚拟列。我们还通过 WATERMARK 语法，在 ts 字段上声明了 watermark 策略（容忍 5 秒乱序），ts 字段因此也成了事件时间列。关于时间属性以及 DDL 语法可以阅读官方文档了解更多：

- 时间属性: https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/table/streaming/time_attributes.html
- DDL: <https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/table/sql/create.html#create-table>

在 SQL CLI 中成功创建 Kafka 表后, 可以通过 `show tables;` 和 `describe user_behavior;` 来查看目前已注册的表, 以及表的详细信息。我们也可以直接在 SQL CLI 中运行 `SELECT * FROM user_behavior;` 预览下数据 (按 `q` 退出)。

接下来, 我们会通过三个实战场景来更深入地了解 Flink SQL。

统计每小时的成交量

使用 DDL 创建 Elasticsearch 表

我们先在 SQL CLI 中创建一个 ES 结果表, 根据场景需求主要需要保存两个数据: 小时、成交量。

```
CREATE TABLE buy_cnt_per_hour (
    hour_of_day BIGINT,
    buy_cnt BIGINT
) WITH (
    'connector.type' = 'elasticsearch', -- 使用 elasticsearch connector
    'connector.version' = '6', -- elasticsearch 版本, 6 能支持 es 6+ 以及 7+ 的版本
    'connector.hosts' = 'http://localhost:9200', -- elasticsearch 地址
    'connector.index' = 'buy_cnt_per_hour', -- elasticsearch 索引名, 相当于数据库的表名
    'connector.document-type' = 'user_behavior', -- elasticsearch 的 type, 相当于数据库的库名
    'connector.bulk-flush.max-actions' = '1', -- 每条数据都刷新
    'format.type' = 'json', -- 输出数据格式 json
    'update-mode' = 'append'
);
```

我们不需要在 Elasticsearch 中事先创建 `buy_cnt_per_hour` 索引, Flink Job 会自动创建该索引。

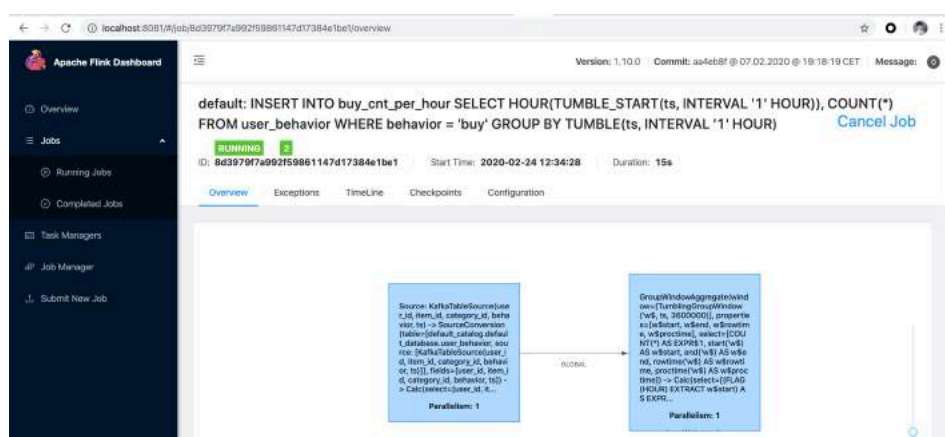
提交 Query

统计每小时的成交量就是每小时共有多少“buy”的用户行为。因此会需要用到 TUMBLE 窗口函数，按照一小时切窗。然后每个窗口分别统计“buy”的个数，这可以通过先过滤出“buy”的数据，然后 `COUNT(*)` 实现。

```
INSERT INTO buy_cnt_per_hour
SELECT HOUR(TUMBLE_START(ts, INTERVAL '1' HOUR)), COUNT(*)
FROM user_behavior
WHERE behavior = 'buy'
GROUP BY TUMBLE(ts, INTERVAL '1' HOUR);
```

这里我们使用 `HOUR` 内置函数，从一个 `TIMESTAMP` 列中提取出一天中第几个小时的值。使用了 `INSERT INTO` 将 query 的结果持续不断地插入到上文定义的 `es` 结果表中（可以将 `es` 结果表理解成 query 的物化视图）。另外可以阅读该文档了解更多关于窗口聚合的内容：<https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/table/sql/queries.html#group-windows>

在 Flink SQL CLI 中运行上述查询后，在 Flink Web UI 中就能看到提交的任任务，该任务是一个流式任务，因此会一直运行。



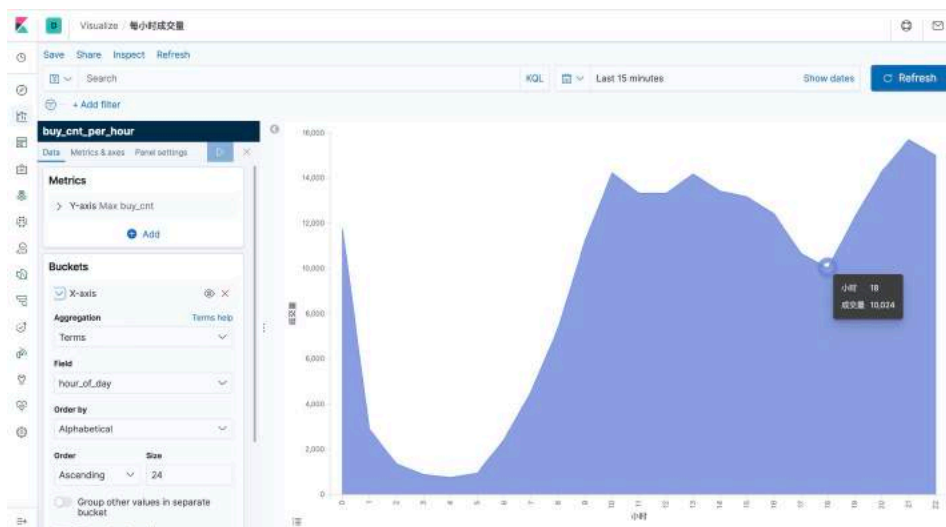
使用 Kibana 可视化结果

我们已经通过 Docker Compose 启动了 Kibana 容器，可以通过 <http://localhost:5601> 访问 Kibana。首先我们需要先配置一个 index pattern。点击左侧工具栏的“Management”，就能找到“Index Patterns”。点击“Create Index Pattern”，然后通过输入完整的索引名“buy_cnt_per_hour”创建 index pattern。创建完成后，Kibana 就知道了我们的索引，我们就可以开始探索数据了。

先点击左侧工具栏的“Discovery”按钮，Kibana 就会列出刚刚创建的索引中的内容。



接下来，我们先创建一个 Dashboard 用来展示各个可视化的视图。点击页面左侧的“Dashboard”，创建一个名为“用户行为日志分析”的 Dashboard。然后点击“Create New”创建一个新的视图，选择“Area”面积图，选择“buy_cnt_per_hour”索引，按照如下截图中的配置（左侧）画出成交量面积图，并保存为“每小时成交量”。



可以看到凌晨是一天中成交量的低谷。

统计一天每 10 分钟累计独立用户数

另一个有意思的可视化是统计一天中每一刻的累计独立用户数 (uv)，也就是每一刻的 uv 数都代表从 0 点到当前时刻为止的总计 uv 数，因此该曲线肯定是单调递增的。

我们仍然先在 SQL CLI 中创建一个 Elasticsearch 表，用于存储结果汇总数据。主要有两个字段：时间和累积 uv 数。

```
CREATE TABLE cumulative_uv (
  time_str STRING,
  uv BIGINT
) WITH (
  'connector.type' = 'elasticsearch',
  'connector.version' = '6',
  'connector.hosts' = 'http://localhost:9200',
  'connector.index' = 'cumulative_uv',
  'connector.document-type' = 'user_behavior',
  'format.type' = 'json',
  'update-mode' = 'upsert'
);
```

为了实现该曲线，我们可以先通过 OVER WINDOW 计算出每条数据的当前分钟，以及当前累计 uv（从 0 点开始到当前行为止的独立用户数）。uv 的统计我们通过内置的 COUNT(DISTINCT user_id) 来完成，Flink SQL 内部对 COUNT DISTINCT 做了非常多的优化，因此可以放心使用。

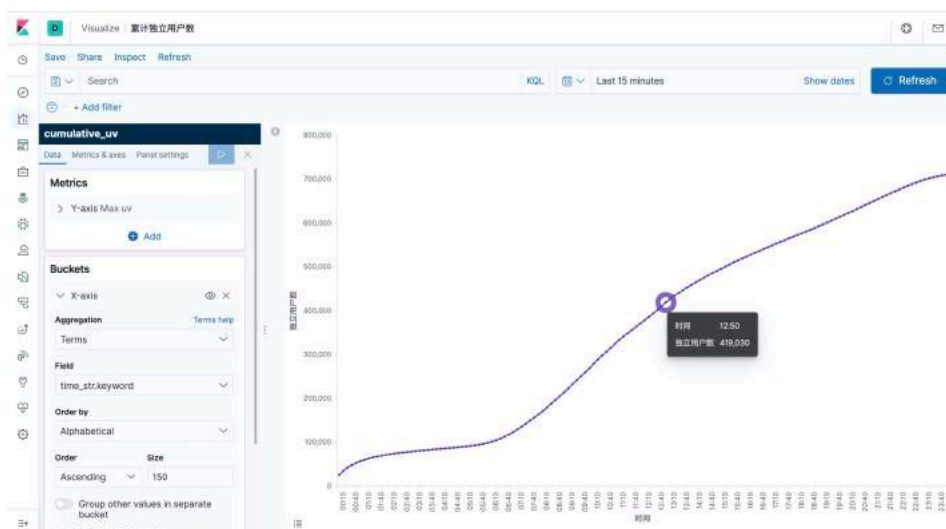
```
CREATE VIEW uv_per_10min AS
SELECT
  MAX(SUBSTR(DATE_FORMAT(ts, 'HH:mm'),1,4) || '0') OVER w AS time_str,
  COUNT(DISTINCT user_id) OVER w AS uv
FROM user_behavior
WINDOW w AS (ORDER BY proctime ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW);
```

这里我们使用 SUBSTR 和 DATE_FORMAT 还有 || 内置函数，将一个 TIME-STAMP 字段转换成了 10 分钟单位的时间字符串，如：12:10, 12:20。关于 OVER WINDOW 的更多内容可以参考文档：<https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/table/sql/queries.html#aggregations>

我们还使用了 CREATE VIEW 语法将 query 注册成了一个逻辑视图，可以方便地在后续查询中对该 query 进行引用，这有利于拆解复杂 query。注意，创建逻辑视图不会触发作业的执行，视图的结果也不会落地，因此使用起来非常轻量，没有额外开销。由于 uv_per_10min 每条输入数据都产生一条输出数据，因此对于存储压力较大。我们可以基于 uv_per_10min 再根据分钟时间进行一次聚合，这样每 10 分钟只有一个点会存储在 Elasticsearch 中，对于 Elasticsearch 和 Kibana 可视化渲染的压力会小很多。

```
INSERT INTO cumulative_uv
SELECT time_str, MAX(uv)
FROM uv_per_10min
GROUP BY time_str;
```

提交上述查询后，在 Kibana 中创建 cumulative_uv 的 index pattern，然后在 Dashboard 中创建一个“Line”折线图，选择 cumulative_uv 索引，按照如下截图中的配置（左侧）画出累计独立用户数曲线，并保存。



顶级类目排行榜

最后一个有意思的可视化是类目排行榜，从而了解哪些类目是支柱类目。不过由于源数据中的类目分类太细（约 5000 个类目），对于排行榜意义不大，因此我们希望能将其归约到顶级类目。所以笔者在 mysql 容器中预先准备了子类目与顶级类目的映射数据，用作维表。

在 SQL CLI 中创建 MySQL 表，后续用作维表查询。

```
CREATE TABLE category_dim (
  sub_category_id BIGINT, -- 子类目
  parent_category_id BIGINT -- 顶级类目
) WITH (
  'connector.type' = 'jdbc',
  'connector.url' = 'jdbc:mysql://localhost:3306/flink',
  'connector.table' = 'category',
  'connector.driver' = 'com.mysql.jdbc.Driver',
  'connector.username' = 'root',
  'connector.password' = '123456',
  'connector.lookup.cache.max-rows' = '5000',
  'connector.lookup.cache.ttl' = '10min'
);
```

同时我们再创建一个 Elasticsearch 表，用于存储类目统计结果。

```
CREATE TABLE top_category (
  category_name STRING, -- 类目名称
  buy_cnt BIGINT -- 销量
) WITH (
  'connector.type' = 'elasticsearch',
  'connector.version' = '6',
  'connector.hosts' = 'http://localhost:9200',
  'connector.index' = 'top_category',
  'connector.document-type' = 'user_behavior',
  'format.type' = 'json',
  'update-mode' = 'upsert'
);
```

第一步我们通过维表关联，补全类目名称。我们仍然使用 CREATE VIEW 将该查询注册成一个视图，简化逻辑。维表关联使用 temporal join 语法，可以查看文档了解更多：<https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/table/streaming/joins.html#join-with-a-temporal-table>

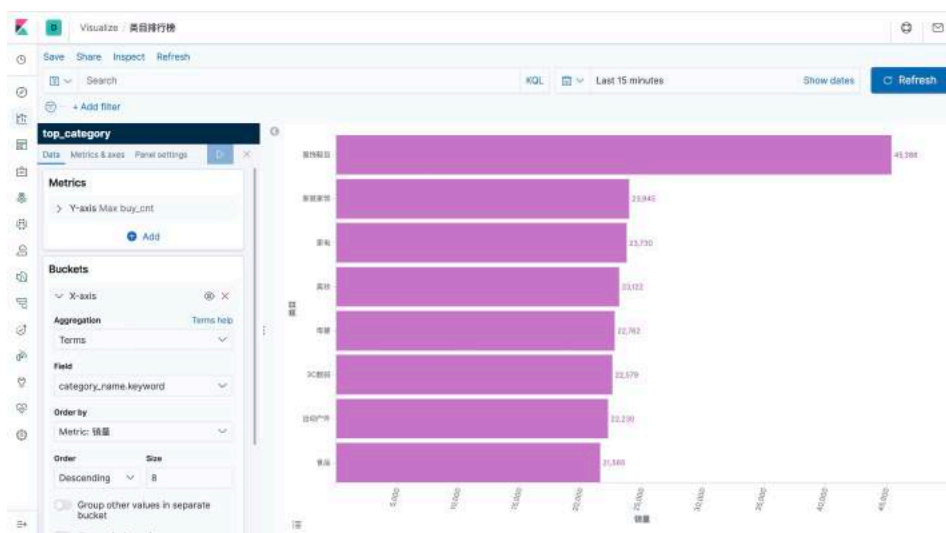
```
CREATE VIEW rich_user_behavior AS
SELECT U.user_id, U.item_id, U.behavior,
CASE C.parent_category_id
  WHEN 1 THEN '服饰鞋包'
  WHEN 2 THEN '家装家饰'
  WHEN 3 THEN '家电'
  WHEN 4 THEN '美妆'
  WHEN 5 THEN '母婴'
  WHEN 6 THEN '3C 数码'
  WHEN 7 THEN '运动户外'
  WHEN 8 THEN '食品'
  ELSE '其他'
END AS category_name
FROM user_behavior AS U LEFT JOIN category_dim FOR SYSTEM_TIME AS OF
U.proctime AS C
ON U.category_id = C.sub_category_id;
```

最后根据 类目名称分组，统计出 buy 的事件数，并写入 Elasticsearch 中。

```
INSERT INTO top_category
SELECT category_name, COUNT(*) buy_cnt
FROM rich_user_behavior
WHERE behavior = 'buy'
```

```
GROUP BY category_name;
```

提交上述查询后，在 Kibana 中创建 `top_category` 的 index pattern，然后在 Dashboard 中创建一个”Horizontal Bar”条形图，选择 `top_category` 索引，按照如下截图中的配置（左侧）画出类目排行榜，并保存。



可以看到服饰鞋包的成交量远远领先其他类目。

到目前为止，我们已经完成了三个实战案例及其可视化视图。现在可以回到 Dashboard 页面，对各个视图进行拖拽编排，让我们的 Dashboard 看上去更加正式、直观（如本文开篇效果图）。当然，Kibana 还提供了非常丰富的图形和可视化选项，而用户行为数据中也有很多有意思的信息值得挖掘，感兴趣的读者可以用 Flink SQL 对数据进行更多维度的分析，并使用 Kibana 展示更多可视化图，并观测图形数据的实时变化。

结尾

在本文中，我们展示了如何使用 Flink SQL 集成 Kafka, MySQL, Elastic-

search 以及 Kibana 来快速搭建一个实时分析应用。整个过程无需一行 Java/Scala 代码，使用 SQL 纯文本即可完成。期望通过本文，可以让读者了解到 Flink SQL 的易用和强大，包括轻松连接各种外部系统、对事件时间和乱序数据处理的原生支持、维表关联、丰富的内置函数等等。希望你能喜欢我们的实战演练，并从中获得乐趣和知识！

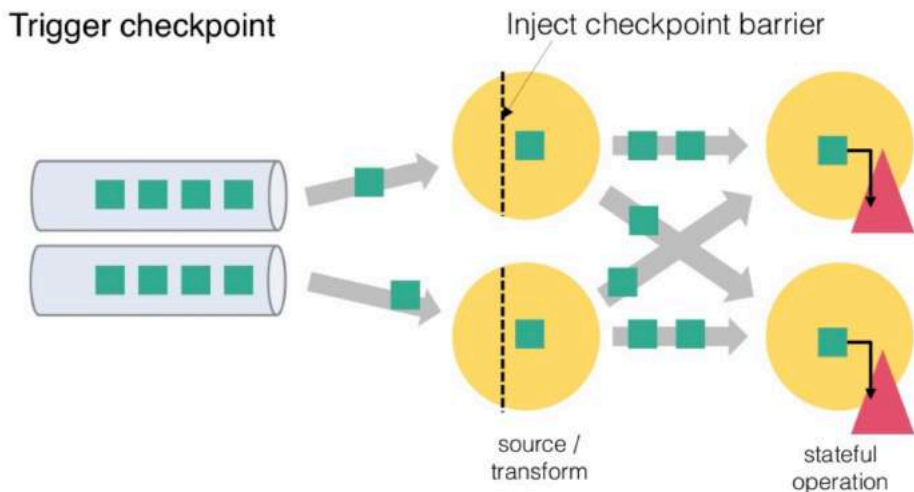
Flink Checkpoint 问题排查实用指南

作者：邱从贤（山智） | 阿里巴巴高级开发工程师

在 Flink 中，状态可靠性保证由 Checkpoint 支持，当作业出现 failover 的情况下，Flink 会从最近成功的 Checkpoint 恢复。在实际情况中，我们可能会遇到 Checkpoint 失败，或者 Checkpoint 慢的情况，本文会统一聊一聊 Flink 中 Checkpoint 异常的情况（包括失败和慢），以及可能的原因和排查思路。

1. Checkpoint 流程简介

首先我们需要了解 Flink 中 Checkpoint 的整个流程是怎样的，在了解整个流程之后，我们才能在出问题的时候，更好的进行定位分析。



从上图我们可以知道，Flink 的 Checkpoint 包括如下几个部分：

- JM trigger checkpoint
- Source 收到 trigger checkpoint 的 PRC，自己开始做 snapshot，并往下游发送 barrier
- 下游接收 barrier（需要 barrier 都到齐才会开始做 checkpoint）
- Task 开始同步阶段 snapshot
- Task 开始异步阶段 snapshot
- Task snapshot 完成，汇报给 JM

上面的任何一个步骤不成功，整个 checkpoint 都会失败。

2. Checkpoint 异常情况排查

2.1 Checkpoint 失败

可以在 Checkpoint 界面看到如下图所示，下图中 Checkpoint 10423 失败了。

10428	IN PROGRESS	7/15 (47%)	18:35:14	18:40:59	5min 45s	77.62 GB	0 B
10427	COMPLETED	15/15	18:26:50	18:35:11	8min 12s	12.00 GB	\$2.81 MB
10426	COMPLETED	15/15	18:18:59	18:26:57	7min 58s	16.27 GB	\$2.81 MB
10425	COMPLETED	15/15	18:12:29	18:16:58	6min 29s	15.34 GB	\$2.72 MB
10424	COMPLETED	15/15	18:06:15	18:12:27	6min 11s	26.23 GB	\$2.81 MB
10423	FAILED	14/15	17:56:15	18:06:05	10min	8.82 GB	\$6.16 MB
10422	COMPLETED	15/15	17:47:57	17:56:13	8min 16s	17.17 GB	\$2.72 MB
10421	COMPLETED	15/15	17:38:10	17:47:06	9min 37s	16.01 GB	\$2.79 MB
10420	COMPLETED	15/15	17:28:27	17:38:17	9min 50s	16.96 GB	\$2.81 MB
10419	COMPLETED	15/15	17:19:12	17:28:25	9min 13s	14.47 GB	\$2.81 MB

点击 Checkpoint 10423 的详情，我们可以看到类下图所示的表格（下图中将 operator 名字截取掉了）。

Acknowledged	Latest Acknowledgment	End to End Duration	State Size	Buffered During Alignment
5/5 (100%)	18:05:07	8min 51s	55.44 KB	0 B
5/5 (100%)	18:04:32	8min 16s	55.44 KB	0 B
4/5 (80%)	18:06:05	9min 49s	8.82 GB	66.16 MB

上图中我们看到三行，表示三个 operator，其中每一列的含义分别如下：

- 其中 **Acknowledged** 一列表示有多少个 subtask 对这个 Checkpoint 进行了 ack，从图中我们可以知道第三个 operator 总共有 5 个 subtask，但是只有 4 个进行了 ack；
- 第二列 **Latest Acknowledgment** 表示该 operator 的所有 subtask 最后 ack 的时间；
- **End to End Duration** 表示整个 operator 的所有 subtask 中完成 snapshot 的最长时间；
- **State Size** 表示当前 Checkpoint 的 state 大小 -- 主要这里如果是增量 checkpoint 的话，则表示增量大小；

- `Buffered During Alignment` 表示在 barrier 对齐阶段积攒了多少数据, 如果这个数据过大也间接表示对齐比较慢);

Checkpoint 失败大致分为两种情况: Checkpoint Decline 和 Checkpoint Expire。

2.1.1 Checkpoint Decline

我们能从 `jobmanager.log` 中看到类似下面的日志

`Decline checkpoint 10423 by task 0b60f08bf8984085b59f8d9bc74ce2e1` of job `85d268e6fbc19411185f7e4868a44178`. 其中 10423 是 checkpointID, `0b60f08bf8984085b59f8d9bc74ce2e1` 是 execution id, `85d268e6fbc-19411185f7e4868a44178` 是 job id, 我们可以在 `jobmanager.log` 中查找 execution id, 找到被调度到哪个 taskmanager 上, 类似如下所示:

```
2019-09-02 16:26:20,972 INFO [jobmanager-future-thread-61] org.apache.flink.runtime.executiongraph.ExecutionGraph - XXXXXXXXXXXX (100/289)
(87b751b1fd90e32af55f02bb2f9a9892)
switched from SCHEDULED to DEPLOYING.
2019-09-02 16:26:20,972 INFO [jobmanager-future-thread-61] org.apache.flink.runtime.executiongraph.ExecutionGraph - Deploying XXXXXXXXXXXX (100/289)
(attempt #0) to slot
container_e24_1566836790522_8088_04_013155_1 on hostnameABCDE
```

从上面的日志我们知道该 execution 被调度到 `hostnameABCDE` 的 `container_e24_1566836790522_8088_04_013155_1` slot 上, 接下来我们就可以到 `container_e24_1566836790522_8088_04_013155` 的 `taskmanager.log` 中查找 Checkpoint 失败的具体原因了。

另外对于 Checkpoint Decline 的情况, 有一种情况我们在这里单独抽取出来进行介绍: Checkpoint Cancel。

当前 Flink 中如果较小的 Checkpoint 还没有对齐的情况下, 收到了更大的

Checkpoint，则会把较小的 Checkpoint 给取消掉。我们可以看到类似下面的日志：

```
$taskNameWithSubTaskAndID: Received checkpoint barrier for checkpoint 20
before completing
current checkpoint 19. Skipping current checkpoint.
```

这个日志表示，当前 Checkpoint 19 还在对齐阶段，我们收到了 Checkpoint 20 的 barrier。然后会逐级通知到下游的 task checkpoint 19 被取消了，同时也会通知 JM 当前 Checkpoint 被 decline 掉了。

在下游 task 收到被 cancelBarrier 的时候，会打印类似如下的日志：

```
DEBUG
$taskNameWithSubTaskAndID: Checkpoint 19 canceled, aborting alignment.

或者

DEBUG
$taskNameWithSubTaskAndID: Checkpoint 19 canceled, skipping alignment.

或者

WARN
$taskNameWithSubTaskAndID: Received cancellation barrier for checkpoint 20
before completing
current checkpoint 19. Skipping current checkpoint.
```

上面三种日志都表示当前 task 接收到上游发送过来的 barrierCancel 消息，从而取消了对应的 Checkpoint。

2.1.2 Checkpoint Expire

如果 Checkpoint 做的非常慢，超过了 timeout 还没有完成，则整个 Checkpoint 也会失败。当一个 Checkpoint 由于超时而失败是，会在 `jobmanager.log` 中看到如下的日志：

```
Checkpoint 1 of job 85d268e6fbc19411185f7e4868a44178 expired before completing.
```

表示 Checkpoint 1 由于超时而失败，这个时候可以看看这个日志后面是否有

类似下面的日志：

```
Received late message for now expired checkpoint attempt 1 from
0b60f08bf8984085b59f8d9bc74ce2e1 of job 85d268e6fbc19411185f7e4868a44178.
```

可以按照 2.1.1 中的方法找到对应的 taskmanager.log 查看具体信息。

下面的日志如果是 DEBUG 的话，我们会在开始处标记 `DEBUG`

我们按照下面的日志把 TM 端的 snapshot 分为三个阶段，开始做 snapshot 前，同步阶段，异步阶段：

```
DEBUG
Starting checkpoint (6751) CHECKPOINT on task taskNameWithSubtasks (4/4)
```

这个日志表示 TM 端 barrier 对齐后，准备开始做 Checkpoint。

```
DEBUG
2019-08-06 13:43:02,613 DEBUG org.apache.flink.runtime.state.
AbstractSnapshotStrategy -
DefaultOperatorStateBackend snapshot (FsCheckpointStorageLocation
{fileSystem=org.
apache.flink.core.fs.SafetyNetWrapperFileSystem@70442baf,
checkpointDirectory=xxxxxxx,
sharedStateDirectory=xxxxxxx, taskOwnedStateDirectory=xxxxxx,
metadataFilePath=xxxxxx,
reference=(default), fileSizeSizeThreshold=1024}, synchronous part) in
thread Thread[Async calls on
Source: xxxxxx
_source -> Filter (27/70),5,Flink Task Threads] took 0 ms.
```

上面的日志表示当前这个 backend 的同步阶段完成，共使用了 0 ms。

```
DEBUG
DefaultOperatorStateBackend snapshot (FsCheckpointStorageLocation
{fileSystem=org.
apache.flink.core.fs.SafetyNetWrapperFileSystem@7908affe,
checkpointDirectory=xxxxxx,
sharedStateDirectory=xxxxxx, taskOwnedStateDirectory=xxxxxx,
metadataFilePath=xxxxxx,
reference=(default), fileSizeSizeThreshold=1024}, asynchronous part) in
thread Thread[pool-48-
```

```
thread-14,5,Flink Task Threads] took 369 ms
```

上面的日志表示异步阶段完成，异步阶段使用了 369 ms

在现有的日志情况下，我们通过上面三个日志，定位 snapshot 是开始晚，同步阶段做的慢，还是异步阶段做的慢。然后再按照情况继续进一步排查问题。

2.2 Checkpoint 慢

在 2.1 节中，我们介绍了 Checkpoint 失败的排查思路，本节会分情况介绍 Checkpoint 慢的情况。

Checkpoint 慢的情况如下：比如 Checkpoint interval 1 分钟，超时 10 分钟，Checkpoint 经常需要做 9 分钟（我们希望 1 分钟左右就能够做完），而且我们预期 state size 不是非常大。

对于 Checkpoint 慢的情况，我们可以按照下面的顺序逐一检查。

2.2.1 Source Trigger Checkpoint 慢

这个一般发生较少，但是也有可能，因为 source 做 snapshot 并往下游发送 barrier 的时候，需要抢锁（这个现在社区正在进行用 mailbox 的方式替代当前抢锁的方式，详情参考^[1]）。如果一直抢不到锁的话，则可能导致 Checkpoint 一直得不到机会进行。如果在 Source 所在的 `taskmanager.log` 中找不到开始做 Checkpoint 的 log，则可以考虑是否属于这种情况，可以通过 `jstack` 进行进一步确认锁的持有情况。

2.2.2 使用增量 Checkpoint

现在 Flink 中 Checkpoint 有两种模式，全量 Checkpoint 和 增量 Checkpoint，其中全量 Checkpoint 会把当前的 state 全部备份一次到持久化存储，而增量 Checkpoint，则只备份上一次 Checkpoint 中不存在的 state，因此增量 Checkpoint 每次上传的内容会相对更好，在速度上会有更大的优势。

现在 Flink 中仅在 RocksDBStateBackend 中支持增量 Checkpoint，如果你已经使用 RocksDBStateBackend，可以通过开启增量 Checkpoint 来加速，具体的可以参考 [2]。

2.2.3 作业存在反压或者数据倾斜

我们知道 task 仅在接收到所有的 barrier 之后才会进行 snapshot，如果作业存在反压，或者有数据倾斜，则会导致全部的 channel 或者某些 channel 的 barrier 发送慢，从而整体影响 Checkpoint 的时间，这两个可以通过如下的页面进行检查：

Detail	SubTasks	TaskManagers	Watermarks	Accumulators	BackPressure	Metrics
Measurement: 2s ago Back Pressure Status: ● high						
SubTask	Ratio	Status				
1	0.96	● high				
2	0.85	● high				
3	0.89	● high				
4	0.97	● high				
5	0.86	● high				

上图中我们选择了一个 task，查看所有 subtask 的反压情况，发现都是 high，表示反压情况严重，这种情况下会导致下游接收 barrier 比较晚。

Detail	SubTasks	TaskManagers	Watermarks	Accumulators	BackPres
ID	Delay(ms)	Bytes Received	Records Received	Bytes Sent	
LOG 207	-	87.65 MB	144,356	79.44 MB	
LOG 137	-	56.21 MB	92,742	50.65 MB	
LOG 73	-	60.07 MB	91,836	49.13 MB	
LOG 40	-	41.67 MB	69,513	35.64 MB	
LOG 156	-	31.82 MB	52,267	28.67 MB	
LOG 389	-	26.92 MB	45,034	24.75 MB	
LOG 53	-	24.71 MB	41,635	22.19 MB	
LOG 101	-	20.84 MB	35,946	18.68 MB	

上图中我们选择其中一个 operator，点击所有的 subtask，然后按照 Records Received/Bytes Received/TPS 从大到小进行排序，能看到前面几个 subtask 会比其他的 subtask 要处理的数据多。

如果存在反压或者数据倾斜的情况，我们需要首先解决反压或者数据倾斜问题之后，再查看 Checkpoint 的时间是否符合预期。

2.2.4 Barrier 对齐慢

从前面我们知道 Checkpoint 在 task 端分为 barrier 对齐（收齐所有上游发送过来的 barrier），然后开始同步阶段，再做异步阶段。如果 barrier 一直对不齐的话，就不会开始做 snapshot。

barrier 对齐之后会有如下日志打印：

```
DEBUG
Starting checkpoint (6751) CHECKPOINT on task taskNameWithSubtasks (4/4)
```

如果 `taskmanager.log` 中没有这个日志，则表示 barrier 一直没有对齐，接下来我们需要了解哪些上游的 barrier 没有发送下来，如果你使用 At Least Once 的话，可以观察下面的日志：

```
DEBUG
Received barrier for checkpoint 96508 from channel 5
```

表示该 task 收到了 channel 5 来的 barrier，然后看对应 Checkpoint，再查看还剩哪些上游的 barrier 没有接受到，对于 ExactlyOnce 暂时没有类似的日志，可以考虑自己添加，或者 jmap 查看。

2.2.5 主线程太忙，导致没机会做 snapshot

在 task 端，所有的处理都是单线程的，数据处理和 barrier 处理都由主线程处理，如果主线程在处理太慢（比如使用 RocksDBBackend，state 操作慢导致整体处理慢），导致 barrier 处理的慢，也会影响整体 Checkpoint 的进度，在这一步我们

需要能够查看某个 PID 对应 hotmethod，这里推荐两个方法：

1. 多次连续 jstack，查看一直处于 `RUNNABLE` 状态的线程有哪些；
2. 使用工具 [AsyncProfile](#) dump 一份火焰图，查看占用 CPU 最多的栈；

如果有其他更方便的方法当然更好，也欢迎推荐。

2.2.6 同步阶段做的慢

同步阶段一般不会太慢，但是如果我们通过日志发现同步阶段比较慢的话，对于非 RocksDBBackend 我们可以考虑查看是否开启了异步 snapshot，如果开启了异步 snapshot 还是慢，需要看整个 JVM 在干嘛，也可以使用前一节中的工具。对于 RocksDBBackend 来说，我们可以用 `iostate` 查看磁盘的压力如何，另外可以查看 tm 端 RocksDB 的 log 的日志如何，查看其中 SNAPSHOT 的时间总共开销多少。

RocksDB 开始 snapshot 的日志如下：

```
2019/09/10-14:22:55.734684 7fef66ffd700 [utilities/checkpoint/checkpoint_
impl.cc:83] Started
the snapshot process -- creating snapshot in directory /tmp/flink-io-
87c360ce-0b98-48f4-9629-
2cf0528d5d53/XXXXXXXXXX/chk-92729
```

snapshot 结束的日志如下：

```
2019/09/10-14:22:56.001275 7fef66ffd700 [utilities/checkpoint/checkpoint_
impl.cc:145] Snapshot
DONE. All is good
```

2.2.7 异步阶段做的慢

对于异步阶段来说，tm 端主要将 state 备份到持久化存储上，对于非 RocksDB-Backend 来说，主要瓶颈来自于网络，这个阶段可以考虑观察网络的 metric，或者对应机器上能够观察到网络流量的情况（比如 `iftop`）。

对于 RocksDB 来说，则需要从本地读取文件，写入到远程的持久化存储上，

所以不仅需要考虑网络的瓶颈，还需要考虑本地磁盘的性能。另外对于 RocksDB-Backend 来说，如果觉得网络流量不是瓶颈，但是上传比较慢的话，还可以尝试考虑开启多线程上传功能^[3]。

3. 总结

在第二部分内容中，我们介绍了官方编译的包的情况下排查一些 Checkpoint 异常情况的主要场景，以及相应的排查方法，如果排查了上面所有的情况，还是没有发现瓶颈所在，则可以考虑添加更详细的日志，逐步将范围缩小，然后最终定位原因。

上文提到的一些 DEBUG 日志，如果 flink dist 包是自己编译的话，则建议将 Checkpoint 整个步骤内的一些 DEBUG 改为 INFO，能够通过日志了解整个 Checkpoint 的整体阶段，什么时候完成了什么阶段，也在 Checkpoint 异常的时候，快速知道每个阶段都消耗了多少时间。

参考内容

[1] Change threading-model in StreamTask to a mailbox-based approach

[2] 增量 checkpoint 原理介绍

[3] RocksDBStateBackend 多线程上传 State

如何分析及处理 Flink 反压？

作者：林小铂 | 网易游戏高级开发工程师

反压 (backpressure) 是实时计算应用开发中，特别是流式计算中，十分常见的问题。反压意味着数据管道中某个节点成为瓶颈，处理速率跟不上上游发送数据的速率，而需要对上游进行限速。由于实时计算应用通常使用消息队列来进行生产端和消费端的解耦，消费端数据源是 pull-based 的，所以反压通常是从某个节点传导至数据源并降低数据源 (比如 Kafka consumer) 的摄入速率。

关于 Flink 的反压机制，网上已经有不少博客介绍，中文博客推荐这两篇 [1](#)。简单来说，Flink 拓扑中每个节点 (Task) 间的数据都以阻塞队列的方式传输，下游来不及消费导致队列被占满后，上游的生产也会被阻塞，最终导致数据源的摄入被阻塞。而本文将着重结合官方的博客 [\[4\]](#) 分享笔者在实践中分析和处理 Flink 反压的经验。

反压的影响

反压并不会直接影响作业的可用性，它表明作业处于亚健康的状态，有潜在的性能瓶颈并可能导致更大的数据处理延迟。通常来说，对于一些对延迟要求不太高或者数据量比较小的应用来说，反压的影响可能并不明显，然而对于规模比较大的 Flink 作业来说反压可能会导致严重的问题。

这是因为 Flink 的 checkpoint 机制，反压还会影响到两项指标：checkpoint 时长和 state 大小。

- 前者是因为 checkpoint barrier 是不会越过普通数据的，数据处理被阻塞也会导致 checkpoint barrier 流经整个数据管道的时长变长，因而 checkpoint 总体时间 (End to End Duration) 变长。

- 后者是因为为保证 EOS (Exactly-Once-Semantics, 准确一次), 对于有两个以上输入管道的 Operator, checkpoint barrier 需要对齐 (Alignment), 接受到较快的输入管道的 barrier 后, 它后面数据会被缓存起来但不处理, 直到较慢的输入管道的 barrier 也到达, 这些被缓存的数据会被放到 state 里面, 导致 checkpoint 变大。

这两个影响对于生产环境的作业来说是十分危险的, 因为 checkpoint 是保证数据一致性的关键, checkpoint 时间变长有可能导致 checkpoint 超时失败, 而 state 大小同样可能拖慢 checkpoint 甚至导致 OOM (使用 Heap-based StateBackend) 或者物理内存使用超出容器资源 (使用 RocksDBStateBackend) 的稳定性问题。

因此, 我们在生产上要尽量避免出现反压的情况 (顺带一提, 为了缓解反压给 checkpoint 造成的压力, 社区提出了 FLIP-76: Unaligned Checkpoints[4] 来解耦反压和 checkpoint)。

定位反压节点

要解决反压首先要做的是定位到造成反压的节点, 这主要有两种办法:

1. 通过 Flink Web UI 自带的反压监控面板;
2. 通过 Flink Task Metrics。

前者比较容易上手, 适合简单分析, 后者则提供了更加丰富的信息, 适合用于监控系统。因为反压会向上游传导, 这两种方式都要求我们从 Source 节点到 Sink 的逐一排查, 直到找到造成反压的根源原因 [4]。下面分别介绍这两种办法。

反压监控面板

Flink Web UI 的反压监控提供了 SubTask 级别的反压监控, 原理是通过周期性对 Task 线程的栈信息采样, 得到线程被阻塞在请求 Buffer (意味着被下游队列阻塞) 的频率来判断该节点是否处于反压状态。默认配置下, 这个频率在 0.1 以下则为

OK, 0.1 至 0.5 为 LOW, 而超过 0.5 则为 HIGH。

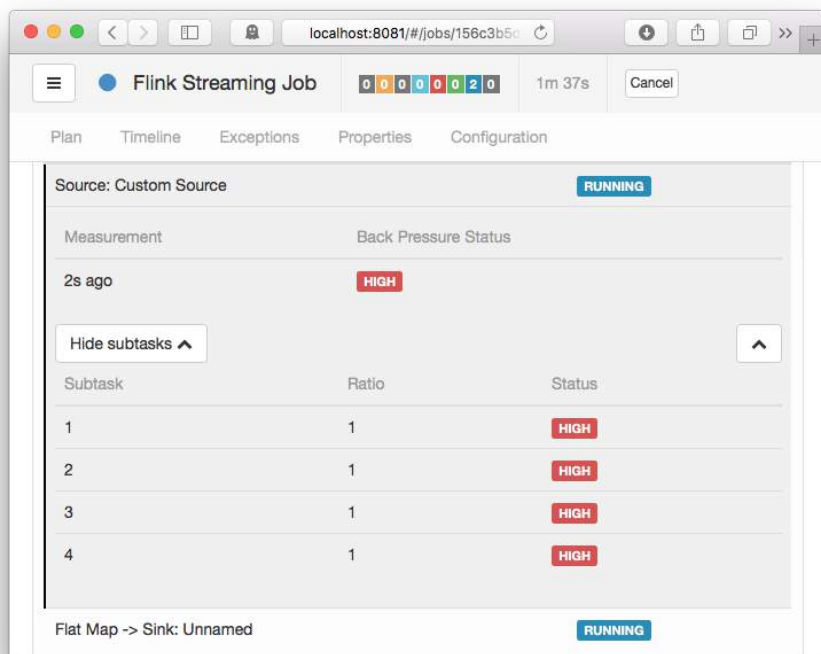


图 1 Flink 1.8 的 Web UI 反压面板 (来自官方博客)

如果处于反压状态, 那么有两种可能性:

1. 该节点的发送速率跟不上它的产生数据速率。这一般会发生在一条输入多条输出的 Operator (比如 flatmap)。
2. 下游的节点接受速率较慢, 通过反压机制限制了该节点的发送速率。

如果是第一种状况, 那么该节点则为反压的根源节点, 它是从 Source Task 到 Sink Task 的第一个出现反压的节点。如果是第二种情况, 则需要继续排查下游节点。

值得注意的是, 反压的根源节点并不一定会在反压面板体现出高反压, 因为反压

面板监控的是发送端，如果某个节点是性能瓶颈并不会导致它本身出现高压，而是导致它的上游出现高压。总体来看，如果我们找到第一个出现反压的节点，那么反压根源要么是就这个节点，要么是它紧接着的下游节点。

那么如果区分这两种状态呢？很遗憾只通过反压面板是无法直接判断的，我们还需要结合 Metrics 或者其他监控手段来定位。此外如果作业的节点数很多或者并行度很大，由于要采集所有 Task 的栈信息，反压面板的压力也会很大甚至不可用。

Task Metrics

Flink 提供的 Task Metrics 是更好的反压监控手段，但也要求更加丰富的背景知识。

首先我们简单回顾下 Flink 1.5 以后的网路栈，熟悉的读者可以直接跳过。

TaskManager 传输数据时，不同的 TaskManager 上的两个 Subtask 间通常根据 key 的数量有多个 Channel，这些 Channel 会复用同一个 TaskManager 级别的 TCP 链接，并且共享接收端 Subtask 级别的 Buffer Pool。

在接收端，每个 Channel 在初始阶段会被分配固定数量的 Exclusive Buffer，这些 Buffer 会被用于存储接受到的数据，交给 Operator 使用后再被释放。Channel 接收端空闲的 Buffer 数量称为 Credit，Credit 会被定时同步给发送端被后者用于决定发送多少个 Buffer 的数据。

在流量较大时，Channel 的 Exclusive Buffer 可能会被写满，此时 Flink 会向 Buffer Pool 申请剩余的 Floating Buffer。这些 Floating Buffer 属于备用 Buffer，哪个 Channel 需要就去哪里。而在 Channel 发送端，一个 Subtask 所有的 Channel 会共享同一个 Buffer Pool，这边就没有区分 Exclusive Buffer 和 Floating Buffer。

CREDIT-BASED FLOW CONTROL (FLINK 1.5+)

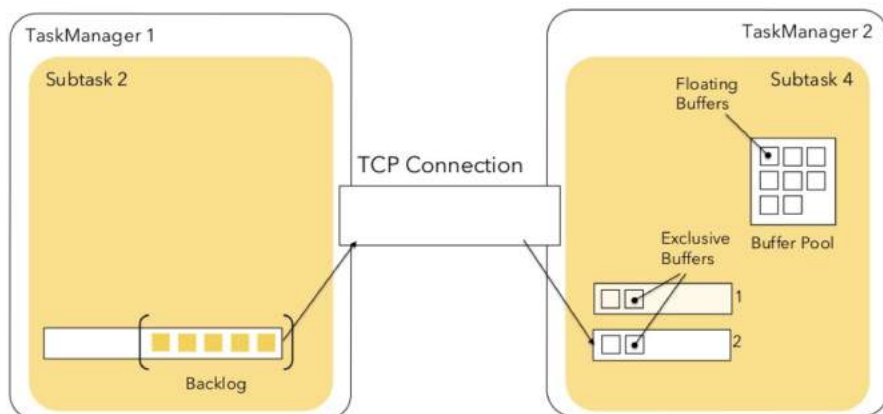


图 2 Flink Credit-Based 网络

我们在监控反压时会用到的 Metrics 主要和 Channel 接受端的 Buffer 使用率有关，最为有用的是以下几个 Metrics:

Metris	描述
outPoolUsage	发送端 Buffer 的使用率
inPoolUsage	接收端 Buffer 的使用率
floatingBuffersUsage (1.9 以上)	接收端 Floating Buffer 的使用率
exclusiveBuffersUsage (1.9 以上)	接收端 Exclusive Buffer 的使用率

其中 inPoolUsage 等于 floatingBuffersUsage 与 exclusiveBuffersUsage 的总和。

分析反压的大致思路是：如果一个 Subtask 的发送端 Buffer 占用率很高，则表明它被下游反压限速了；如果一个 Subtask 的接受端 Buffer 占用很高，则表明它将反压传导至上游。反压情况可以根据以下表格进行对号入座（图片来自官网）：












	outPoolUsage low	outPoolUsage high
inPoolUsage low		 (backpressured, temporary situation: upstream is not backpressured yet or not anymore)
inPoolUsage high (Flink 1.9+)	if all upstream tasks' outPoolUsage are low:  (may eventually cause backpressure)	 (backpressured by downstream task(s) or network, probably forwarding backpressure upstream)
	if any upstream task's outPoolUsage is high:  (may exercise backpressure upstream and may be the source of backpressure)	

图 3 反压分析表

outPoolUsage 和 inPoolUsage 同为低或同为高分别表明当前 Subtask 正常或处于被下游反压，这应该没有太多疑问。而比较有趣的是当 outPoolUsage 和 inPoolUsage 表现不同时，这可能是出于反压传导的中间状态或者表明该 Subtask 就是反压的根源。

如果一个 Subtask 的 outPoolUsage 是高，通常是被下游 Task 所影响，所以可以排查它本身是反压根源的可能性。如果一个 Subtask 的 outPoolUsage 是低，但其 inPoolUsage 是高，则表明它有可能是反压的根源。因为通常反压会传导至其上游，导致上游某些 Subtask 的 outPoolUsage 为高，我们可以根据这点来进一步判断。值得注意的是，反压有时是短暂的且影响不大，比如来自某个 Channel 的短暂网络延迟或者 TaskManager 的正常 GC，这种情况下我们可以不用处理。

对于 Flink 1.9 及以上版本，除了上述的表格，我们还可以根据 floatingBufferUsage/exclusiveBuffersUsage 以及其上游 Task 的 outPoolUsage 来进行进一步的分析一个 Subtask 和其上游 Subtask 的数据传输。

	exclusiveBuffersUsage low	exclusiveBuffersUsage high
floatingBuffersUsage low + all upstream outPoolUsage low		._3
floatingBuffersUsage low + any upstream outPoolUsage high	 (potential network bottleneck)	._3
floatingBuffersUsage high + all upstream outPoolUsage low	 (backpressure eventually appears on only some of the input channels)	 (backpressure eventually appears on most or all of the input channels)
floatingBuffersUsage high + any upstream outPoolUsage high	 (backpressure on only some of the input channels)	 (backpressure on most or all of the input channels)

³ this should not happen

图 4 Flink 1.9 反压分析表

通常来说, floatingBuffersUsage 为高则表明反压正在传导至上游, 而 exclusiveBuffersUsage 则表明了反压是否存在倾斜 (floatingBuffersUsage 高、exclusiveBuffersUsage 低为有倾斜, 因为少数 channel 占用了大部分的 Floating Buffer)。

至此, 我们已经有比较丰富的手段定位反压的根源是出现在哪个节点, 但是具体的原因还没有办法找到。另外基于网络的反压 metrics 并不能定位到具体的 Operator, 只能定位到 Task。特别是 embarrassingly parallel (易并行) 的作业 (所有的 Operator 会被放入一个 Task, 因此只有一个节点), 反压 metrics 则派不上用场。

分析具体原因及处理

定位到反压节点后, 分析造成原因的办法和我们分析一个普通程序的性能瓶颈的办法是十分类似的, 可能还要更简单一点, 因为我们要观察的主要是 Task Thread。

在实践中, 很多情况下的反压是由于数据倾斜造成的, 这点我们可以通过 Web UI 各个 SubTask 的 Records Sent 和 Record Received 来确认, 另外 Check-point detail 里不同 SubTask 的 State size 也是一个分析数据倾斜的有用指标。

此外，最常见的问题可能是用户代码的执行效率问题（频繁被阻塞或者性能问题）。最有用的办法就是对 TaskManager 进行 CPU profile，从中我们可以分析到 Task Thread 是否跑满一个 CPU 核：如果是的话要分析 CPU 主要花费在哪些函数里面，比如我们生产环境中就偶尔遇到卡在 Regex 的用户函数（ReDoS）；如果不是的话要看 Task Thread 阻塞在哪里，可能是用户函数本身有些同步的调用，可能是 checkpoint 或者 GC 等系统活动导致的暂时系统暂停。

当然，性能分析的结果也可能是正常的，只是作业申请的资源不足而导致了反压，这就通常要求拓展并行度。值得一提的，在未来的版本 Flink 将会直接在 WebUI 提供 JVM 的 CPU 火焰图 [5]，这将大大简化性能瓶颈的分析。

另外 TaskManager 的内存以及 GC 问题也可能会导致反压，包括 TaskManager JVM 各区内存不合理导致的频繁 Full GC 甚至失联。推荐可以通过给 TaskManager 启用 G1 垃圾回收器来优化 GC，并加上 `-XX:+PrintGCDetails` 来打印 GC 日志的方式来观察 GC 的问题。

总结

反压是 Flink 应用运维中常见的问题，它不仅意味着性能瓶颈还可能导致作业的不稳定性。定位反压可以从 Web UI 的反压监控面板和 Task Metric 两者入手，前者方便简单分析，后者适合深入挖掘。定位到反压节点后我们可以通过数据分布、CPU Profile 和 GC 指标日志等手段来进一步分析反压背后的具体原因并进行针对性的优化。

参考文献

1. [Flink 原理与实现：如何处理反压问题](#)
2. [一文彻底搞懂 Flink 网络流控与反压机制](#)
3. [Flink 轻量级异步快照 ABS 实现原理](#)
4. [Flink Network Stack Vol. 2: Monitoring, Metrics, and that Backpressure Thing](#)
5. [Support for CPU FlameGraphs in new web UI](#)

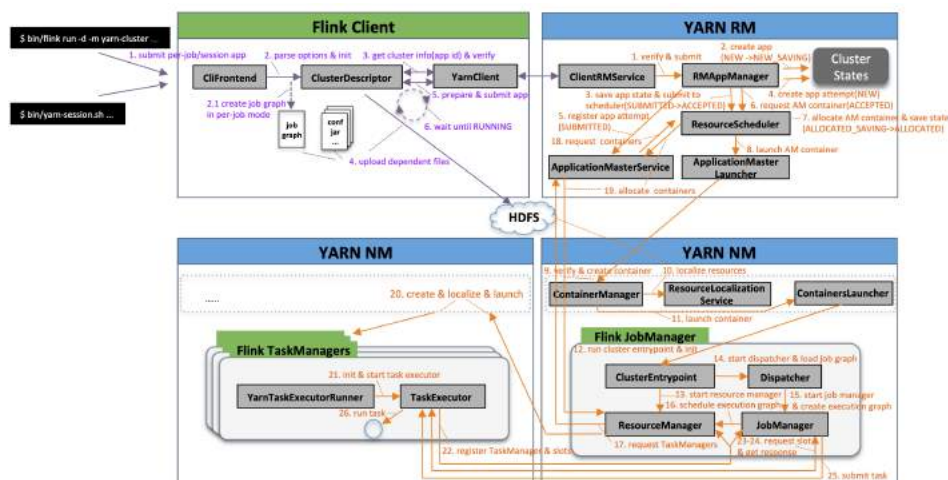
Flink on YARN (上): 一张图轻松掌握基础架构与启动流程

作者：杨弢 | 阿里巴巴技术专家

Flink 支持 Standalone 独立部署和 YARN、Kubernetes、Mesos 等集群部署模式，其中 YARN 集群部署模式在国内的应用越来越广泛。Flink 社区将推出 Flink on YARN 应用解读系列文章，分为上、下两篇。本文基于 FLIP-6 重构后的资源调度模型将介绍 Flink on YARN 应用启动全流程，并进行详细步骤解析。下篇将根据社区大群反馈，解答客户端和 Flink Cluster 的常见问题，分享相关问题的排查思路。

Flink on YARN 流程图

Flink on YARN 集群部署模式涉及 YARN 和 Flink 两大开源框架，应用启动流程的很多环节交织在一起，为了便于大家理解，在一张图上画出了 Flink on YARN 基础架构和应用启动全流程，并对关键角色和流程进行了介绍说明，整个启动流程又被划分成客户端提交（流程标注为紫色）、Flink Cluster 启动和 Job 提交运行（流程标注为橙色）两个阶段分别阐述，由于分支和细节太多，本文会忽略掉一些，只介绍关键流程（基于 Flink 开源 1.9 版本源码整理）。



客户端提交流程

1. 执行命令 `:bin/flink run -d -m yarn-cluster ...` 或 `bin/yarn-session.sh ...` 来提交 per-job 运行模式或 session 运行模式的应用；
2. 解析命令参数项并初始化，启动指定运行模式，如果是 per-job 运行模式将根据命令行参数指定的 Job 主类创建 job graph；
 - 如果可以从命令行参数 (`-yid`) 或 YARN properties 临时文件 (`${java.io.tmp-dir}/.yarn-properties-${user.name}`) 中获取应用 ID，向指定的应用提交 Job；
 - 否则当命令行参数中包含 `-d` (表示 detached 模式) 和 `-m yarn-cluster` (表示指定 YARN 集群模式)，启动 per-job 运行模式；
 - 否则当命令行参数项不包含 `-yq` (表示查询 YARN 集群可用资源) 时，启动 session 运行模式；
3. 获取 YARN 集群信息、新应用 ID 并启动运行前检查；
 - 通过 YarnClient 向 YARN ResourceManager(下文缩写为: YARN RM, YARN Master 节点, 负责整个集群资源的管理和调度) 请求创建一个新应用 (YARN RM 收到创建应用请求后生成新应用 ID 和 container 申请的

资源上限后返回), 并且获取 YARN Slave 节点报告 (YARN RM 返回全部 slave 节点的 ID、状态、rack、http 地址、总资源、已使用资源等信息);

- **运行前检查:** (1) 简单验证 YARN 集群能否访问; (2) 最大 node 资源能否满足 flink JobManager/TaskManager vcores 资源申请需求; (3) 指定 queue 是否存在 (不存在也只是打印 WARN 信息, 后续向 YARN 提交时排除异常并退出); (4) 当预期应用申请的 Container 资源会超出 YARN 资源限制时抛出异常并退出; (5) 当预期应用申请不能被满足时 (例如总资源超出 YARN 集群可用资源总量、Container 申请资源超出 NM 可用资源最大值等) 提供一些参考信息。

4. **将应用配置** (flink-conf.yaml、logback.xml、log4j.properties) 和相关文件 (flink jars、ship files、user jars、job graph 等) 上传至分布式存储 (例如 HDFS) 的应用暂存目录 (/user/\${user.name}/.flink/);

5. **准备应用提交上下文** (ApplicationSubmissionContext, 包括应用的名、类型、队列、标签等信息和应用 Master 的 container 的环境变量、classpath、资源大小等), 注册处理部署失败的 shutdown hook (清理应用对应的 HDFS 目录), 然后通过 YarnClient 向 YARN RM 提交应用;

6. **循环等待直到应用状态为 RUNNING, 包含两个阶段:**

- **循环等待应用提交成功 (SUBMITTED):** 默认每隔 200ms 通过 YarnClient 获取应用报告, 如果应用状态不是 NEW 和 NEW_SAVING 则认为提交成功并退出循环, 每循环 10 次会将当前的应用状态输出至日志: "Application submission is not finished, submitted application is still in ", 提交成功后输出日志: "Submitted application "

- **循环等待应用正常运行 (RUNNING):** 每隔 250ms 通过 YarnClient 获取应用报告, 每轮循环也会将当前的应用状态输出至日志: "Deploying cluster, current state ". 应用状态成功变为 RUNNING 后将输出日志 "YARN application has been deployed successfully." 并退出循环, 如果等到的是非预期状态如 FAILED/FINISHED/KILLED, 就会在输出 YARN 返回的诊断信息 ("The

YARN application unexpectedly switched to state during deployment. Diagnostics from YARN: ...) 之后抛出异常并退出。

Flink Cluster 启动流程

1. **YARN RM 中的 ClientRMService** (为普通用户提供的 RPC 服务组件, 处理来自客户端的各种 RPC 请求, 比如查询 YARN 集群信息, 提交、终止应用等) **接收到应用提交请求, 简单校验后将请求转交给 RMAppManager** (YARN RM 内部管理应用生命周期的组件);
2. **RMAppManager 根据应用提交上下文内容创建初始状态为 NEW 的应用**, 将应用状态持久化到 RM 状态存储服务 (例如 ZooKeeper 集群, RM 状态存储服务用来保证 RM 重启、HA 切换或发生故障后集群应用能够正常恢复, 后续流程中的涉及状态存储时不再赘述), 应用状态变为 NEW_SAVING;
3. **应用状态存储完成后, 应用状态变为 SUBMITTED**; RMAppManager 开始向 ResourceScheduler (YARN RM 可拔插资源调度器, YARN 自带三种调度器 FifoSchedular/FairSchedular/CapacityScheduler, 其中 CapacityScheduler 支持功能最多使用最广泛, FifoSchedular 功能最简单基本不可用, 今年社区已明确不再继续支持 FairSchedular, 建议已有用户迁至 CapacityScheduler) 提交应用, 如果无法正常提交 (例如队列不存在、不是叶子队列、队列已停用、超出队列最大应用数限制等) 则抛出拒绝该应用, 应用状态先变为 FINAL_SAVING 触发应用状态存储流程并在完成后变为 FAILED; 如果提交成功, 应用状态变为 ACCEPTED;
4. **开始创建应用运行实例** (ApplicationAttempt, 由于一次运行实例中最重要的组件是 ApplicationMaster, 下文简称 AM, 它的状态代表了 ApplicationAttempt 的当前状态, 所以 ApplicationAttempt 实际也代表了 AM), 初始状态为 NEW;
5. **初始化应用运行实例信息**, 并向 ApplicationMasterService (AM&RM 协

议接口服务, 处理来自 AM 的请求, 主要包括注册和心跳) 注册, 应用实例状态变为 SUBMITTED;

6. **RMAppManager 维护的应用实例开始初始化 AM 资源申请信息并重新校验队列**, 然后向 ResourceScheduler 申请 AM Container (Container 是 YARN 中资源的抽象, 包含了内存、CPU 等多维度资源), 应用实例状态变为 ACCEPTED;
7. **ResourceScheduler 会根据优先级 (队列 / 应用 / 请求每个维度都有优先级配置) 从根队列开始层层递进**, 先后选择当前优先级最高的子队列、应用直至具体某个请求, 然后结合集群资源分布等情况作出分配决策, AM Container 分配成功后, 应用实例状态变为 ALLOCATED_SAVING, 并触发应用实例状态存储流程, 存储成功后应用实例状态变为 ALLOCATED;
8. **RMAppManager 维护的应用实例开始通知 ApplicationMasterLauncher** (AM 生命周期管理服务, 负责启动或清理 AM container) 启动 AM container, ApplicationMasterLauncher 与 YARN NodeManager (下文简称 YARN NM, 与 YARN RM 保持通信, 负责管理单个节点上的全部资源、Container 生命周期、附属服务等, 监控节点健康状况和 Container 资源使用) 建立通信并请求启动 AM container;
9. **ContainerManager** (YARN NM 核心组件, 管理所有 Container 的生命周期) **接收到 AM container 启动请求**, YARN NM 开始校验 Container Token 及资源文件, 创建应用实例和 Container 实例并存储至本地, 结果返回后应用实例状态变为 LAUNCHED;
10. **ResourceLocalizationService** (资源本地化服务, 负责 Container 所需资源的本地化。它能够按照描述从 HDFS 上下载 Container 所需的文件资源, 并尽量将它们分摊到各个磁盘上以防止出现访问热点) **初始化各种服务组件**、创建工作目录、从 HDFS 下载运行所需的各种资源至 Container 工作目录 (路径为: `${yarn.nodemanager.local-dirs}/usercache/${user}/appcache//`);

11. **ContainersLauncher** (负责 container 的具体操作, 包括启动、重启、恢复和清理等) 将待运行 **Container** 所需的环境变量和运行命令写到 **Container 工作目录**下的 `launch_container.sh` 脚本中, 然后运行该脚本启动 Container;

12. **Container 进程加载并运行 ClusterEntrypoint**(Flink JobManager 入口类, 每种集群部署模式和应用运行模式都有相应的实现, 例如在 YARN 集群部署模式下, per-job 应用运行模式实现类是 `YarnJobClusterEntrypoint`, session 应用运行模式实现类是 `YarnSessionClusterEntrypoint`), 首先初始化相关运行环境:

- 输出各软件版本及运行环境信息、命令行参数项、classpath 等信息;
- 注册处理各种 SIGNAL 的 handler: 记录到日志
- 注册 JVM 关闭保障的 shutdown hook: 避免 JVM 退出时被其他 shutdown hook 阻塞
- 打印 YARN 运行环境信息: 用户名
- 从运行目录中加载 flink conf
- 初始化文件系统
- 创建并启动各类内部服务 (包括 `RpcService`、`HAService`、`BlobServer`、`HeartbeatServices`、`MetricRegistry`、`ExecutionGraphStore` 等)
- 将 RPC address 和 port 更新到 flink conf 配置

13. **启动 ResourceManager** (Flink 资源管理核心组件, 包含 `YarnResourceManager` 和 `SlotManager` 两个子组件, `YarnResourceManager` 负责外部资源管理, 与 YARN RM 建立通信并保持心跳, 申请或释放 `TaskManager` 资源, 注销应用等; `SlotManager` 则负责内部资源管理, 维护全部 Slot 信息和状态) 及相关服务, 创建异步 `AMRMClient`, 开始注册 AM, 注册成功后每隔一段时间 (心跳间隔配置项: `$(yarn.heartbeat.interval)`, 默认 5s) 向 YARN RM 发送心跳来发送资源更新请求和接受资源变更结果。YARN RM 内部该应用和应用运行实例的状态都变为 `RUNNING`,

并通知 AMLivelinessMonitor 服务监控 AM 是否存活状态, 当心跳超过一定时间 (默认 10 分钟) 触发 AM failover 流程;

14. **启动 Dispatcher** (负责接收用户提供的作业, 并且负责为这个新提交的作业拉起一个新的 JobManager) 及相关服务 (包括 REST endpoint 等), 在 per-job 运行模式下, Dispatcher 将直接从 Container 工作目录加载 JobGraph 文件; 在 session 运行模式下, Dispatcher 将在接收客户端提交的 Job (通过 BlockServer 接收 job graph 文件) 后再进行后续流程;
15. **根据 JobGraph 启动 JobManager** (负责作业调度、管理 Job 和 Task 的生命周期), 构建 ExecutionGraph (JobGraph 的并行化版本, 调度层最核心的数据结构);
16. **JobManager 开始执行 ExecutionGraph**, 向 ResourceManager 申请资源;
17. **ResourceManager 将资源请求加入等待请求队列**, 并通过心跳向 YARN RM 申请新的 Container 资源来启动 TaskManager 进程; 后续流程如果有空闲 Slot 资源, SlotManager 将其分配给等待请求队列中匹配的请求, 不用再通过 18. YarnResourceManager 申请新的 TaskManager;
18. **YARN ApplicationMasterService 接收到资源请求后**, 解析出新的资源请求并更新应用请求信息;
19. **YARN ResourceScheduler 成功为该应用分配资源后更新应用信息**, ApplicationMasterService 接收到 Flink JobManager 的下一跳心跳时返回新分配资源信息;
20. **Flink ResourceManager 接收到新分配的 Container 资源后**, 准备好 TaskManager 启动上下文 (ContainerLauncherContext, 生成 TaskManager 配置并上传至分布式存储, 配置其他依赖和环境变量等), 然后向 YARN NM 申请启动 TaskManager 进程, YARN NM 启动 Container 的流程与 AM Container 启动流程基本类似, 区别在于应用实例在 NM 上已存在并未 RUNNING 状态时则跳过应用实例初始化流程, 这里不再赘述;

21. **TaskManager 进程加载并运行 YarnTaskExecutorRunner** (Flink TaskManager 入口类), 初始化流程完成后启动 TaskExecutor (负责执行 Task 相关操作);
22. **TaskExecutor 启动后先向 ResourceManager 注册**, 成功后再向 SlotManager 汇报自己的 Slot 资源与状态; SlotManager 接收到 Slot 空闲资源后主动触发 Slot 分配, 从等待请求队列中选出合适的资源请求后, 向 TaskManager 请求该 Slot 资源
23. **TaskManager 收到请求后检查该 Slot 是否可分配** (不存在则返回异常信息)、Job 是否已注册 (没有则先注册再分配 Slot), 检查通过后将 Slot 分配给 JobManager;
24. **JobManager 检查 Slot 分配是否重复**, 通过后通知 Execution 执行部署 task 流程, 向 TaskExecutor 提交 task; TaskExecutor 启动新的线程运行 Task。

参考资料

[Flink Release-1.9 SourceCode](#)

[Flink Release-1.9 Documents](#)

[\[FLIP-6 – Flink Deployment and Process Model – Standalone, Yarn, Mesos, Kubernetes, etc.\]](#)

[YARN 3.2 SourceCode](#)

[YARN 3.2.0 Documents](#)

| Flink on YARN (下): 常见问题与排查思路

作者: 杨弢 | 阿里巴巴技术专家

Flink 支持 Standalone 独立部署和 YARN、Kubernetes、Mesos 等集群部署模式, 其中 YARN 集群部署模式在国内的应用越来越广泛。Flink 社区将推出 Flink on YARN 应用解读系列文章, 分为上、下两篇。上篇分享了基于 FLIP-6 重构后的资源调度模型介绍 Flink on YARN 应用启动全流程, 本文将根据社区大群反馈, 解答客户端和 Flink Cluster 的常见问题, 分享相关问题的排查思路。

客户端常见问题与排查思路

应用提交控制台异常信息: Could not build the program from JAR file.

这个问题的迷惑性较大, 很多时候并非指定运行的 JAR 文件问题, 而是提交过程中发生了异常, 需要根据日志信息进一步排查。最常见原因是未将依赖的 Hadoop JAR 文件加到 CLASSPATH, 找不到依赖类 (例如: `ClassNotFoundException: org.apache.hadoop.yarn.exceptions.YarnException`) 导致加载客户端入口类 (`FlinkYarnSessionCli`) 失败。

Flink on YARN 应用提交时如何关联到指定 YARN 集群?

Flink on YARN 客户端通常需配置 `HADOOP_CONF_DIR` 和 `HADOOP_CLASSPATH` 两个环境变量来让客户端能加载到 Hadoop 配置和依赖 JAR 文件。示例 (已有环境变量 `HADOOP_HOME` 指定 Hadoop 部署目录):

```
export HADOOP_CONF_DIR=${HADOOP_HOME}/etc/hadoop
export HADOOP_CLASSPATH=`${HADOOP_HOME}/bin/hadoop classpath`
```


客户端日志在哪里，如何配置？

客户端日志通常在 Flink 部署目录的 log 文件夹下: \${FLINK_HOME}/log/flink-\${USER}-client-.log, 使用 log4j 配置: \${FLINK_HOME}/conf/log4j-cli.properties。

有的客户端环境比较复杂, 难以定位日志位置和配置时, 可以通过以下环境变量配置打开 log4j 的 DEBUG 日志, 跟踪 log4j 的初始化和详细加载流程: export JVM_ARGS="-Dlog4j.debug=true"

客户端疑难问题排查思路

当客户端日志无法正常定位时, 可以修改 log4j 配置文件将日志级别由 INFO 改为 DEBUG 后重新运行, 看是否有 DEBUG 日志可以帮助排查问题。对于一些没有日志或日志信息不完整的问题, 可能需要开展代码级调试, 修改源码重新打包替换的方式太过繁琐, 推荐使用 Java 字节码注入工具 Byteman (详细语法说明请参考: Byteman Document), 使用示例:

(1) 编写调试脚本, 例如打印 Flink 实际使用的 Client 类, 以下脚本表示在 CliFrontend#getActiveCustomCommandLine 函数退出时打印其返回值;

```
RULE test
CLASS org.apache.flink.client.cli.CliFrontend
METHOD getActiveCustomCommandLine
AT EXIT
IF TRUE
DO traceIn("----->CliFrontend#getActiveCustomCommandLine return: "+$!);
ENDRULE
```

(2) 设置环境变量, 使用 byteman javaagent:

```
export BYTEMAN_HOME=/path/to/byte-home
export TRACE_SCRIPT=/path/to/script
export JVM_ARGS="-javaagent:${BYTEMAN_HOME}/lib/byteman.jar=script:${TRACE_SCRIPT}"
```

(3) 运行测试命令 `bin/flink run -m yarn-cluster -p 1 ./examples/streaming/WordCount.jar`，控制台将输出内容：

```
----->CliFrontend#getActiveCustomCommandLine return: org.apache.flink.yarn.cli.  
FlinkYarnSessionCli@25ce9dc4
```

Flink Cluster 常见问题与排查思路

用户应用和框架 JAR 包版本冲突问题

该问题通常会抛出 `NoSuchMethodError/ClassNotFoundException/IncompatibleClassChangeError` 等异常，要解决此类问题：

1. **首先需要根据异常类定位依赖库**，然后可以在项目中执行 `mvn dependency:tree` 以树形结构展示全部依赖链，再从中定位冲突的依赖库，也可以增加参数 `-Dincludes` 指定要显示的包，格式为 `[groupId]:[artifactId]:[type]:[version]`，支持匹配，多个用逗号分隔，例如：`mvn dependency:tree -Dincludes=power,javaassist`；
2. **定位冲突包后就要考虑如何排包**，简单的方案是用 `exclusion` 来排除掉其从他依赖项目中传递过来的依赖，不过有的应用场景需要多版本共存，不同组件依赖不同版本，就要考虑用 `Maven Shade` 插件来解决，详情请参考 `Maven Shade Plugin`。

依赖库有多版本 JAR 包共存时如何确定某类的具体来源？

很多应用运行 `CLASSPATH` 中存在相同依赖库的多个版本 JAR 包，导致实际使用的版本跟加载顺序有关，排查问题时经常需要确定某个类的来源 JAR，Flink 支持给 JM/TM 进程配置 JVM 参数，因此可以通过下面三个配置项来打印加载类及其来源（输出在 `.out` 日志），根据具体需要选择其中之一即可：

```
env.java.opts=-verbose:class // 配置 JobManager&TaskManager  
env.java.opts.jobmanager=-verbose:class // 配置 JobManager
```

```
env.java.opts.taskmanager=-verbose:class // 配置 TaskManager
```

Flink 应用的完整日志如何查看?

Flink 应用运行中的 JM/TM 日志可以在 WebUI 上查看, 但是查问题时通常需结合完整日志来分析排查, 因此就需要了解 YARN 的日志保存机制, YARN 上 Container 日志保存位置跟应用状态有关:

1. **如果应用还没有结束, Container 日志会一直保留在其运行所在的节点上**, 即使 Container 已经运行完成仍然可以在所在节点的配置目录下找到: `${yarn.nodemanager.log-dirs}`, 也可以直接从 WebUI 访问: `http://node/containerlogs/`
2. **如果应用已结束并且集群启用了日志收集** (`yarn.log-aggregation-enable=true`), 则通常应用结束后 (也有配置可以增量上传) NM 会将其全部日志上传至分布式存储 (通常是 HDFS) 并删除本地文件, 我们可以通过 `yarn` 命令 `yarn logs -applicationId -appOwner` 查看应用的全部日志, 还可以增加参数项 `-containerId -nodeAddress` 来查看某 container 的日志, 也可以直接访问分布式存储目录: `${yarn.nodemanager.remote-app-log-dir}/${user}/${yarn.nodemanager.remote-app-log-dir-suffix}/`

Flink 应用资源分配问题排查思路

如果 Flink 应用不能正常启动达到 RUNNING 状态, 可以按以下步骤进行排查:

1. **需要先检查应用当前状态**, 根据上述对启动流程的说明, 我们知道:
 - **处于 NEW_SAVING 状态**时正在进行应用信息持久化, 如果持续处于这个状态我们需要检查 RM 状态存储服务 (通常是 ZooKeeper 集群) 是否正常;
 - **如果处于 SUBMITTED 状态**, 可能是 RM 内部发生一些 hold 读写锁的耗时操作导致事件堆积, 需要根据 YARN 集群日志进一步定位;

- 如果处于 **ACCEPTED** 状态, 需要先检查 AM 是否正常, 跳转到步骤 2;
 - 如果已经是 **RUNNING** 状态, 但是资源没有全部拿到导致 JOB 无法正常运行, 跳转到步骤 3;
2. **检查 AM 是否正常**, 可以从 YARN 应用展示界面 (<http://cluster/app/>) 或 YARN 应用 REST API (<http://ws/v1/cluster/apps/>) 查看 diagnostics 信息, 根据关键字信息明确问题原因与解决方案:
- Queue's AM resource limit exceeded. 原因是达到了队列 AM 可用资源上限, 即队列的 AM 已使用资源和 AM 新申请资源之和超出了队列的 AM 资源上限, 可以适当调整队列 AM 可用资源百分比的配置项: `yarn.scheduler.capacity..maximum-am-resource-percent`。
 - User's AM resource limit exceeded. 原因是达到了应用所属用户在该队列的 AM 可用资源上限, 即应用所属用户在该队列的 AM 已使用资源和 AM 新申请资源之和超出了应用所属用户在该队列的 AM 资源上限, 可以适当提高用户可用 AM 资源比例来解决该问题, 相关配置项: `yarn.scheduler.capacity..user-limit-factor` 与 `yarn.scheduler.capacity..minimum-user-limit-percent`。
 - AM container is launched, waiting for AM container to Register with RM. 大致原因是 AM 已启动, 但内部初始化未完成, 可能有 ZK 连接超时等问题, 具体原因需排查 AM 日志, 根据具体问题来解决。
 - Application is Activated, waiting for resources to be assigned for AM. 该信息表示应用 AM 检查已经通过, 正在等待调度器分配, 此时需要进行调度器层面的资源检查, 跳转到步骤 4。
3. **确认应用确实有 YARN 未能满足的资源请求**: 从应用列表页点击问题应用 ID 进入应用页面, 再点击下方列表的应用实例 ID 进入应用实例页面, 看 Total Outstanding Resource Requests 列表中是否有 Pending 资源, 如果没有, 说明 YARN 已分配完毕, 退出该检查流程, 转去检查 AM; 如果有, 说明调度器未能完成分配, 跳转到步骤 4;

4. **调度器分配问题排查**, YARN-9050 支持在 WebUI 上或通过 REST API 自动诊断应用问题, 将在 Hadoop3.3.0 发布, 之前的版本仍需进行人工排查:

- **检查集群或 queue 资源**, scheduler 页面树状图叶子队列展开查看资源信息: Effective Max Resource、Used Resources: (1) 检查集群资源或所在队列资源或其父队列资源是否已用完; (2) 检查叶子队列某维度资源是否接近或达到上限;
- **检查是否存在资源碎片**: (1) 检查集群 Used 资源和 Reserved 资源之和占总资源的比例, 当集群资源接近用满时 (例如 90% 以上), 可能存在资源碎片的情况, 应用的分配速度就会受影响变慢, 因为大部分机器都没有资源了, 机器可用资源不足会被 reserve, reserved 资源达到一定规模后可能导致大部分机器资源被锁定, 后续分配可能就会变慢; (2) 检查 NM 可用资源分布情况, 即使集群资源使用率不高, 也有可能是因为各维度资源分布不同造成, 例如 1/2 节点上的内存资源接近用满 CPU 资源剩余较多, 1/2 节点上的 CPU 资源接近用满内存资源剩余较多, 申请资源中某一维度资源值配置过大也可能造成无法申请到资源;
- **检查是否有高优先级的问题应用频繁申请并立即释放资源的问题**, 这种情况会造成调度器忙于满足这一个应用的资源请求而无暇顾及及其他应用;
- **检查是否存在 Container 启动失败或刚启动就自动退出的情况**, 可以查看 Container 日志 (包括 localize 日志、launch 日志等)、YARN NM 日志或 YARN RM 日志进行排查。

TaskManager 启动异常:

```
org.apache.hadoop.yarn.exceptions.YarnException: Unauthorized request to start container. This token is expired. current time is ... found ...
```

该异常在 Flink AM 向 YARN NM 申请启动 token 已超时的 Container 时抛出, 通常原因是 Flink AM 从 YARN RM 收到这个 Container 很久之后 (超过了 Container 有效时间, 默认 10 分钟, 该 Container 已经被释放) 才去启动它, 进一

步原因是 Flink 内部在收到 YARN RM 返回的 Container 资源后串行启动。

当待启动的 Container 数量较多且分布式文件存储如 HDFS 性能较慢(启动前需上传 TaskManager 配置)时 Container 启动请求容易堆积在内部, FLINK-13184 对这个问题进行了优化, 一是在启动前增加了有效性检查, 避免了无意义的配置上传流程, 二是进行了异步多线程优化, 加快启动速度。

Failover 异常 1:

java.util.concurrent.TimeoutException: Slot allocation request timed out for ...

异常原因是申请的 TaskManager 资源无法正常分配, 可以按 Flink 应用资源分配问题排查思路的步骤 4 排查问题。

Failover 异常 2:

java.util.concurrent.TimeoutException: Heartbeat of TaskManager with id timed out.

异常直接原因是 TaskManager 心跳超时, 进一步原因可能有:

- **进程已退出**, 可能自身发生错误, 或者受到 YARN RM 或 NM 上抢占机制影响, 需要进一步追查 TaskManager 日志或 YARN RM/NM 日志;
- **进程仍在运行**, 集群网络问题造成失联, 连接超时时会自行退出, JobManager 在该异常后会 Failover 自行恢复(重新申请资源并启动新的 TaskManager);
- **进程 GC 时间过长**, 可能是内存泄露或内存资源配置不合理造成, 需根据日志或分析内存进一步定位具体原因。

Failover 异常 3:

java.lang.Exception: Container released on a lost node

异常原因是 Container 运行所在节点在 YARN 集群中被标记为 LOST，该节点上的所有 Container 都将被 YARN RM 主动释放并通知 AM，JobManager 收到此异常后会 Failover 自行恢复（重新申请资源并启动新的 TaskManager），遗留的 TaskManager 进程可在超时后自行退出。

Flink Cluster 疑难问题排查思路

首先根据 JobManager/TaskManager 日志分析定位问题，完整日志请参考“Flink 应用的完整日志如何查看”，如果想获取 DEBUG 信息，需修改 JobManager/TaskManager 的 log4j 配置（\${FLINK_HOME}/conf/log4j.properties）后重新提交运行，对于仍在运行的进程，推荐使用 Java 字节码注入工具 Byteman 来一窥进程内部的相关状态，详细说明请参考：How Do I Install The Agent Into A Running Program?

参考资料

文中橙色字体部分均有跳转，详细参考资料请见下方链接：

[Byteman Documents](#)

[Maven Shade Plugin](#)

[YARN-9050](#)

[FLINK-13184](#)

[How Do I Install The Agent Into A Running Program?](#)

Flink on YARN 上、下两篇文章对 Flink on YARN 应用启动全流程进行梳理，并对客户端和 Flink Cluster 的常见问题提供了排查思路，供大家参考，希望在实际应用中能够对大家有所帮助。

| Apache Flink 与 Apache Hive 的集成

作者：李锐、王刚

李锐 (天离), Apache Hive PMC, 阿里巴巴技术专家

王刚 (乔燃), 阿里巴巴高级开发工程师

导读：随着 Flink 在流式计算的应用场景逐渐成熟和流行，如果 Flink 能同时把批量计算的应用场景处理好，就能减少用户在使用 Flink 时开发和维护的成本，并且能够丰富 Flink 的生态。SQL 是批计算中比较常用的工具，所以 Flink 针对于批计算也以 SQL 为主要接口。本次分享主要介绍 Flink 对批处理的设计与 Hive 的集成。

主要分为下面三点展开：

- 设计架构
- 项目进展
- 性能测试

设计架构

首先和大家分享一下 Flink 批处理的设计架构。

背景



Flink 提升批处理的主要原因是为了减少客户的维护成本和更新成本，以及更好地完善 Flink 的生态环境。又因为 SQL 是批计算场景中一个非常重要的工具，所以我们希望以 SQL 作为批计算场景中的主要接口。为此我们着重优化了 Flink SQL 的功能。目前 Flink SQL 主要有以下几点需要优化：

- 需要完整的元数据管理体制
- 缺少对 DDL(数据定义语言 DDL 用来创建数据库中的各种对象：表、视图、索引、同义词、聚簇等) 的支持
- 与外部系统进行对接不是很方便，尤其是 Hive。Hive 是大数据领域最早的 SQL 引擎，用户基础非常广泛，新的一些 SQL 工具，如 Spark SQL、Impala 都提供了与 Hive 对接的功能，这样用户才能更好地将其应用从 Hive 迁移过来，所以与 Hive 对接对 Flink SQL 而言也十分重要。

目标



目标

Goals

统一的Catalog接口
Unified Catalog API

提供基于内存的和可持久化的Catalog实现
Provide both in-memory and persistent catalog implementations

支持与Hive的互操作:
Interoperability with Hive:

访问Hive元数据
Access to Hive metadata

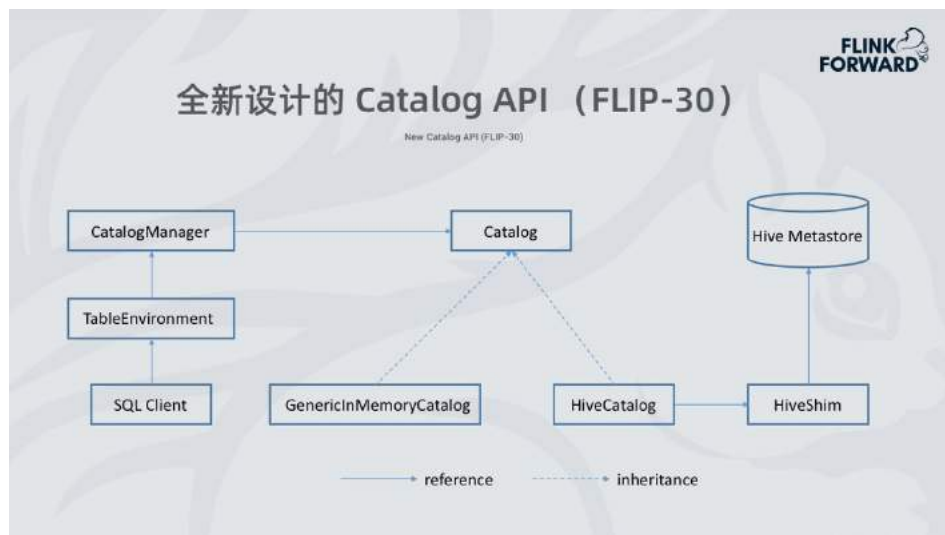
读写Hive表
Read/write Hive tables

支持Flink作为Hive的引擎(长期目标)
Add Flink as a Hive execution engine (long-term plan)

所以我们要完成以下目标:

- **定义统一的 Catalog 接口**, 这个是 Flink SQL 更方便与外部对接的前提条件。如果大家用过 Flink 的 TableSources 和 TableSink 来对接外部的系统的表, 会发现不管是通过写程序还是配置 yaml 文件会跟传统的 SQL 使用方式会有些不同。我们希望改善 Hive 的用户迁移 Flink SQL 需要通过定义 TableSources 和 TableSink 的方式来与 Hive 进行交互, 为此社区提供了一套新的 Catalog 接口以一种更接近传统 SQL 的方式与 Hive 进行交互。
- **提供基于内存和可持久化的实现**。基于内存是 Flink 原有的方式, 用户所有的元数据的生命周期是跟其 Session (会话) 绑定的, Session (会话) 结束之后所有的元数据也都没有了。因为要跟 Hive 交互所以肯定还要提供一个持久化的 Catalog。
- **支持与 Hive 的互操作**。有了 Catalog 之后用户就可以通过 Catalog 访问 Hive 的元数据, 提供 Data Connector 让用户能通过 Flink 读写 Hive 的实际数据, 实现 Flink 与 Hive 的交互。
- **支持 Flink 作为 Hive 的计算引擎 (长期目标)**, 像 Hive On Spark, Hive On Tez。

全新设计的 Catalog API (FLIP-30)



用户通过 SQL Client 或者 Table API 提交请求，Flink 会创建 TableEnvironment，TableEnvironment 再创建 CatalogManager 加载并配置 Catalog 实例，并且 Catalog 支持多种元数据类型 table、database、function、view、partition 等，在 1.9.0 的版本当中 Catalog 会有两个实现：

- 一个是基于内存的 GenericInMemoryCatalog。
- 另一个是 HiveCatalog。HiveCatalog 通过 HiveShim 与 Hive Metastore 交互来操作 Hive 元数据，HiveShim 的作用是处理 Hive 在大版本中 Hive Metastore 不兼容的问题。

从这种实现的方式可以看出，用户可以创建多个 Catalog，也可以访问多个 Hive Metastore，来达到跨 Catalog 查询的操作。

读写 Hive 数据



有了元数据之后就可以实现 Flink SQL 的 Data Connector 来真正的读写 Hive 实际数据。Flink SQL 写入的数据必须要兼容 Hive 的数据格式，也就是 Hive 可以正常读取 Flink 写入的数据，反过来也是一样的。为了实现这一点我们大量复用 Hive 原有的 Input/Output Format、SerDe 等 API，一是为了减少代码冗余，二是尽可能的保持兼容性。

在 Data Connect 中读取 Hive 表数据具体实现类为：HiveTableSource、HiveTableInputFormat。写 Hive 表的具体实现类为：HiveTableSink、HiveTableOutputFormat。

项目进展

再跟大家分享一下 Flink 1.9.0 的现状和 1.10.0 中的新特性以及未来规划。

Flink 1.9.0 的现状



Flink SQL 在 1.9.0 版本中是作为试用功能发布的，功能还不是很完善，具体表现在：

- 支持的数据类型还不全。(1.9.0 中带参数的数据类型基本上都不支持：如 DECIMAL, CHAR 等)
- 对分区表的支持不完善，只能读取分区表，不能写分区表。
- 不支持表的 INSERT OVERWRITE。

Flink 1.10.0 中的新特性



Flink SQL 在 1.10.0 版本里我们做了比较多的进一步开发，与 Hive 集成的功能更加完整。

- 支持读写静态分区和动态分区表。
- 在表级别和分区级别都支持 INSERT OVERWRITE。
- 支持了更多数据类型。（除 UNION 类型都支持）
- 支持更多的 DDL。（CREATE TABLE/DATABASE）
- 支持在 Flink 中调用 Hive 的内置函数。（Hive 大约 200 多个内置函数）
- 支持了更多的 Hive 版本。（Hive 的 1.0.0~3.1.1）
- 做了很多性能优化如，Project/Predicate Pushdown，向量的读取 ORC 数据等。

Module 接口



为了能让用户调用 Flink SQL 时调用 Hive 的内置函数，我们在 Flink 1.10 中引入了一个 Module 接口。这个 Module 是为了让用户能够方便的把外部系统的内置函数接入到系统当中。

- 其使用方式和 Catalog 类似，用户可以通过 Table API 或 Yaml 文件来配置 Module。
- Module 可以同时加载多个，Flink 解析函数的时候通过 Module 的加载顺序在多个 Module 中查找函数的解析。也就是如果两个 Module 包含名字相同的 Function，先加载的 Module 会提供 Function 的定义。
- 目前 Module 有两个实现，CoreModule 提供了 Flink 原生的内置函数，HiveModule 提供了 Hive 的内置函数。

未来工作



未来的工作主要是先做功能的补全，其中包括：

- View 的支持（有可能在 1.11 中完成）
- 持续改进 SQL CLI 的易用性，现在支持翻页显示查询结果，后续支持滚动显示。并支持 Hive 的 `-e -f` 这种非交互式的使用方式
- 支持所有的 Hive 常用 DDL，例如 `CREATE TABLE AS`。
- 兼容 Hive 的语法，让原来在 Hive 上的工程在 Flink 的顺滑的迁移过来
- 支持 SQL CLI 的远程模式，类似 HiveServer2 的远程连接模式。
- 支持流式的写入 Hive 数据。

性能测试

下面是 Flink 在批处理作业下与 HiveMR 对比测试的测试环境和结果。

测试环境



首先我们的测试环境使用了 21 个节点的物理机群，一个 Master 节点和 20 个 Slave 节点。节点的硬件配置是 32 核，64 个线程，256 内存，网络做了端口聚合，每个机器是 12 块的 HDD 硬盘。

测试工具



测试工具使用了 Hortonworks 的 hive-testbench，在 github 中是一个开源工具。我们使用这个工具生成了 10TB 的 TPC-DS 测试数据集，然后分别通过 Flink SQL 和 Hive 对该数据集进行 TPC-DS 的测试。一方面我们对比了 Flink 和 Hive 的性能，另一方面我们验证了 Flink SQL 能够很好的访问 Hive 的数据。测试用到了 Hive 版本是 3.1.1，Flink 用到的是 Master 分支代码。

测试结果



测试结果 Flink SQL 对比 Hive On MapReduce 取得了大约 7 倍的性能提升。这得益于 Flink SQL 所做的一系列优化，比如在调度方面的优化，以及执行计划的优化等。总体来说如果用的是 Hive On MapReduce，迁移到 Flink SQL 会有很大的性能上的提升。

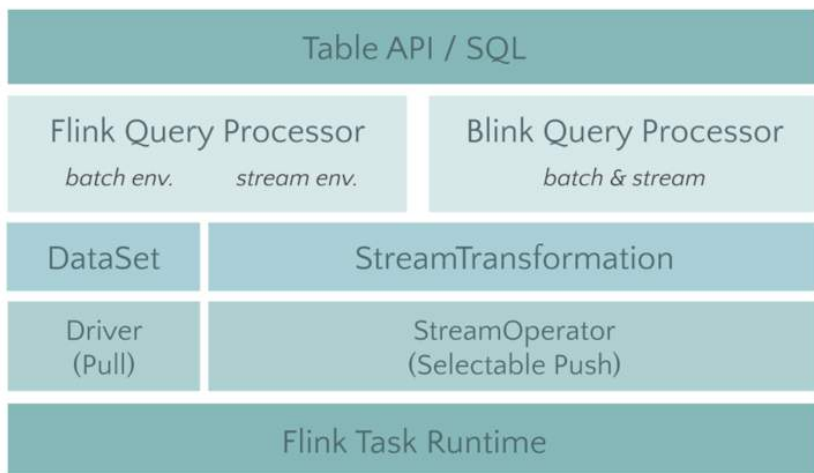
| Flink Batch SQL 1.10 实践

作者：李劲松（之信） | Apache Flink Committer，阿里巴巴技术专家

Flink 作为流批统一的计算框架，在 1.10 中完成了大量 batch 相关的增强与改进。1.10 可以说是第一个成熟的生产可用的 Flink Batch SQL 版本，它一扫之前 Dataset 的羸弱，从功能和性能上都有大幅改进，以下我从架构、外部系统集成、实践三个方面进行阐述。

架构

Stack

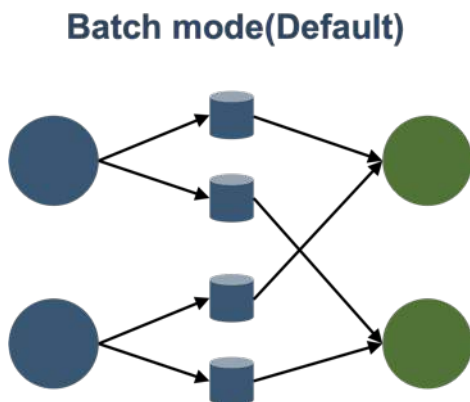


首先来看下 stack，在新的 Blink planner 中，batch 也是架设在 Transformation 上的，这就意味着我们和 Dataset 完全没有关系了：

1. 我们可以尽可能的和 streaming 复用组件，复用代码，有同一套行为。

2. 如果想要 Table/SQL 的 toDataset 或者 fromDataset，那就完全没戏了。
尽可能的在 Table 的层面来处理吧。
3. 后续我们正在考虑在 DataStream 上构建 BoundedStream，给 DataStream 带来批处理的功能。

网络模型



Batch 模式就是在中间结果落盘，这个模式和典型的 Batch 处理是一致的，比如 MapReduce/Spark/Tez。

Flink 以前的网络模型也分为 Batch 和 Pipeline 两种，但是 Batch 模式只是支持上下游隔断执行，也就是说资源用量可以不用同时满足上下游共同的并发。但是另外一个关键点是 Failover 没有对接好，1.9 和 1.10 在这方面进行了改进，支持了单点的 Failover。

建议在 Batch 时打开：

```
jobmanager.execution.failover-strategy = region
```

为了避免重启过于频繁导致 JobMaster 太忙了，可以把重启间隔提高：

```
restart-strategy.fixed-delay.delay = 30 s
```

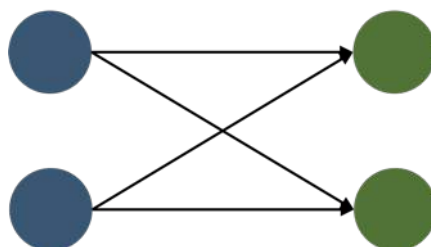
Batch 模式的好处有：

- 容错好，可以单点恢复
- 调度好，不管多少资源都可以运行
- 性能差，中间数据需要落盘，强烈建议开启压缩

`taskmanager.network.blocking-shuffle.compression.enabled = true`

Batch 模式比较稳，适合传统 Batch 作业，大作业。

Pipeline mode



Pipeline 模式是 Flink 的传统模式，它完全和 Streaming 作业用的是同一套代码，其实社区里 Impala 和 Presto 也是类似的模式，纯走网络，需要处理反压，不落盘，它主要的优缺点是：

- 容错差，只能全局重来
- 调度差，你得保证有足够的资源
- 性能好，Pipeline 执行，完全复用 Stream，复用流控反压等功能。

有条件可以考虑开启 Pipeline 模式。

调度模型

Flink on Yarn 支持两种模式，Session 模式和 Per job 模式，现在已经在调度层次高度统一了。

1. Session 模式没有最大进程限制，当有 Job 需要资源时，它就会去 Yarn 申请新资源，当 Session 有空闲资源时，它就会给 Job 复用，所以它的模型和 PerJob 是基本一样的。
2. 唯一的不同只是：Session 模式可以跨作业复用进程。

另外，如果想要更好的复用进程，可以考虑加大 TaskManager 的超时释放：
`resourcemanager.taskmanager-timeout = 900000`

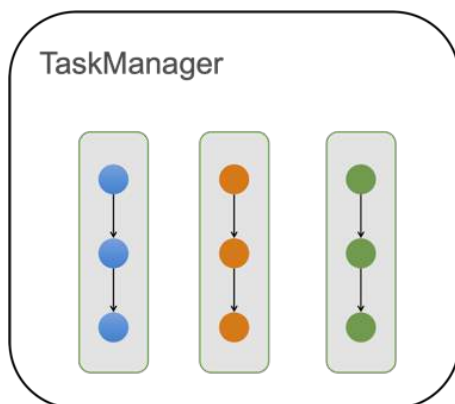
资源模型

先说说并发：

1. 对 Source 来说：目前 Hive 的 table 是根据 InputSplit 来定需要多少并发的，它之后能 Chain 起来的 Operators 自然都是和 source 相同的并发。
2. 对下游网络传输过后的 Operators(Tasks) 来说：除了一定需要单并发的 Task 来说，其它 Task 全部统一并发，由 `table.exec.resource.default-parallelism` 统一控制。

我们在 Blink 内部实现了基于统计信息来推断并发的功能，但是其实以上的策略在大部分场景就够用了。

Manage 内存



目前一个 TaskManager 里面含有多个 Slot，在 Batch 作业中，一个 Slot 里只能运行一个 Task（关闭 SlotShare）。

对内存来说，单个 TM 会把 Manage 内存切分成 Slot 粒度，如果 1 个 TM 中有 n 个 Slot，也就是 Task 能拿到 $1/n$ 的 manage 内存。

我们在 1.10 做了重大的一个改进就是：Task 中 chain 起来的各个 operators 按照比例来瓜分内存，所以现在配置的算子内存都是一个比例值，实际拿到的还要根据 Slot 的内存来瓜分。

这样做的一个重要好处是：

1. 不管当前 Slot 有多少内存，作业能都 run 起来，这大大提高了开箱即用。
2. 不管当前 Slot 有多少内存，Operators 都会把内存瓜分干净，不会存在浪费的可能。

当然，为了运行的效率，我们一般建议单个 Slot 的 manage 内存应该大于 500MB。

另一个事情，在 1.10 后，我们去除了 OnHeap 的 manage 内存，所以只有 off-heap 的 manage 内存。

外部系统集成

Hive

强烈推荐 Hive Catalog + Hive，这也是目前批处理最成熟的架构。在 1.10 中，除了对以前功能的完善以外，其它做了几件事：

1. 多版本支持，支持 Hive 1.X 2.X 3.X
2. 完善了分区的支持，包括分区读，动态 / 静态分区写，分区统计信息的支持。
3. 集成 Hive 内置函数，可以通过以下方式 load：

```
a)TableEnvironment.loadModule(“hiveModule”,new HiveModule(“hiveVersion”))
```

4. 优化了 ORC 的性能读，使用向量化的读取方式，但是目前只支持 Hive 2+ 版本，且要求列没有复杂类型。有没有进行过优化差距在 5 倍量级。

兼容 Streaming Connectors

得益于流批统一的架构，目前的流 Connectors 也能在 batch 上使用，比如 HBase 的 Lookup 和 Sink、JDBC 的 Lookup 和 Sink、Elasticsearch 的 Sink，都可以在 Batch 无缝对接使用起来。

实践

SQL-CLI

在 1.10 中，SQL-CLI 也做了大量的改动，比如把 SQL-CLI 做了 stateful，里面也支持了 DDL，还支持了大量的 DDL 命令，给 SQL-CLI 暴露了很多 TableEnvironment 的能力，这让用户可以方便得多。后续，我们也需要对接 JDBC 的客户端，让用户可以更好的对接外部工具。但是 SQL-CLI 仍然待继续改进，比如目前仍然只支持 Session 模式，不支持 Per Job 模式。

编程方式

```
TableEnvironment tEnv = TableEnvironment.create(EnvironmentSettings
    .newInstance()
    .useBlinkPlanner()
    .inBatchMode()
    .build());
```

老的 BatchTableEnv 因为绑定了 Dataset，而且区分 Java 和 Scala，是不干净的设计方式，所以 Blink planner 只支持新的 TableEnv。

TableEnv 注册的 source, sink, connector, functions，都是 temporary 的，重启之后即失效了。如果需要持久化的 object，考虑使用 HiveCatalog。


```
tEnv.registerCatalog("hive", hiveCatalog);
tEnv.useCatalog("hive");
```

可以通过 `tEnv.sqlQuery` 来执行 DML，这样可以获得一个 `Table`，我们也通过 `collect` 来获得小量的数据：

```
Table table = tEnv.sqlQuery("SELECT COUNT(*) FROM MyTable");
List<Row> results = TableUtils.collectToList(table);
System.out.println(results);
```

可以通过 `tEnv.sqlUpdate` 来执行 DDL，但是目前并不支持创建 `hive` 的 `table`，只能创建 `Flink` 类型的 `table`：

```
tEnv.sqlUpdate(
    "CREATE TABLE myResult (" +
        " cnt BIGINT"
    ") WITH (" +
        " 'connector.type'='jdbc', "
        " ....."
    ")");
```

可以通过 `tEnv.sqlUpdate` 来执行 `insert` 语句，`Insert` 到临时表或者 `Catalog` 表中，比如 `insert` 到上面创建的临时 `JDBC` 表中：

```
tEnv.sqlUpdate("INSERT INTO myResult SELECT COUNT(*) FROM MyTable");
tEnv.execute("MyJob");
```

当结果表是 `Hive` 表时，可以使用 `Overwrite` 语法，也可以使用静态 `Partition` 的语法，这需要打开 `Hive` 的方言：

```
tEnv.getConfig().setSqlDialect(SqlDialect.HIVE);
```

结语

目前 `Flink batch SQL` 仍然在高速发展中，但是 `1.10` 已经是一个可用的版本了，它在功能上、性能上都有很大的提升，后续还有很多有意思的 `features`，等待着大家一起去挖掘。

如何在 PyFlink 1.10 中自定义 Python UDF ?

作者：孙金城（金竹） | Apache Flink PMC，阿里巴巴高级技术专家

我们知道 PyFlink 是在 Apache Flink 1.9 版新增的，那么在 Apache Flink 1.10 中 Python UDF 功能支持的速度是否能够满足用户的急切需求呢？



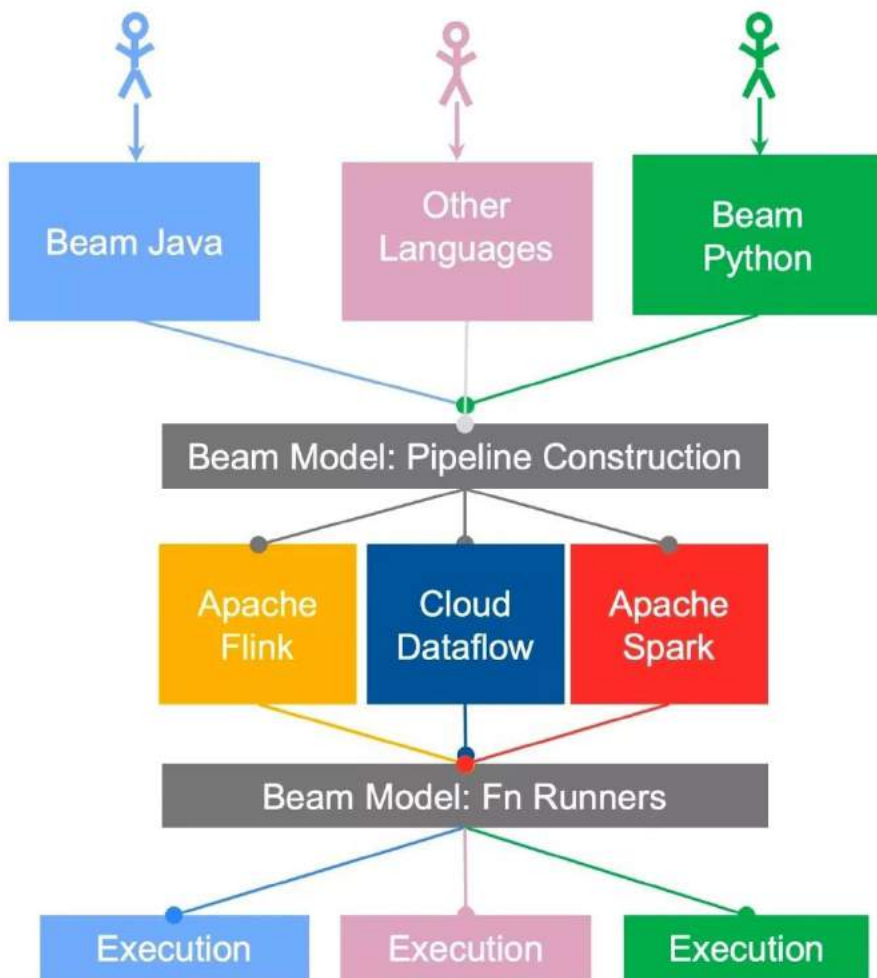
Python UDF 的发展趋势

直观的判断，PyFlink Python UDF 的功能也可以如上图一样能够迅速从幼苗变成大树，为啥有此判断，请继续往下看…

Flink on Beam

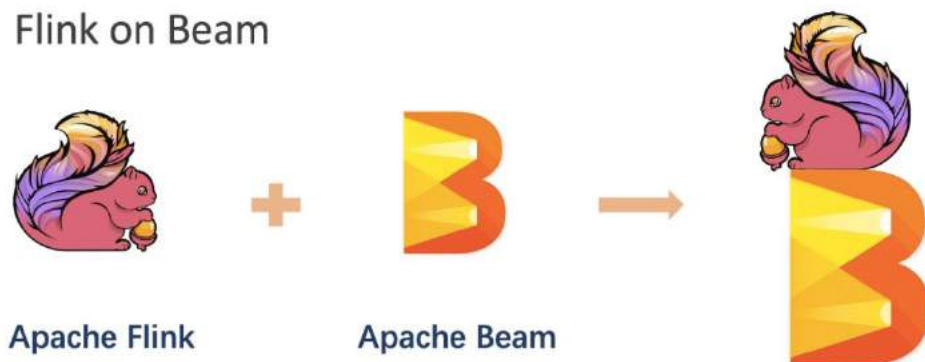
我们都知道有 Beam on Flink 的场景，就是 Beam 支持多种 Runner，也就是

说 Beam SDK 编写的 Job 可以运行在 Flink 之上。如下图所示：

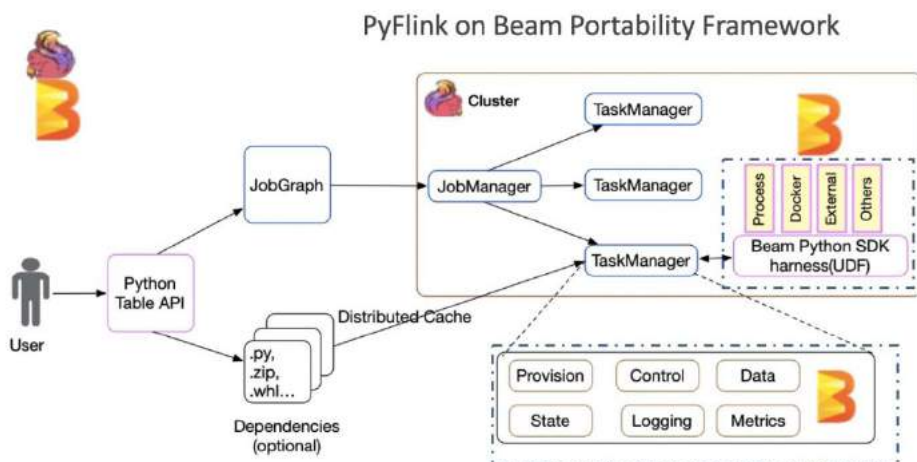


上面这图是 Beam Portability Framework 的架构图，他描述了 Beam 如何支持多语言，如何支持多 Runner，单独说 Apache Flink 的时候我们就可以说是 Beam on Flink，那么怎么解释 Flink on Beam 呢？

Flink on Beam



在 Apache Flink 1.10 中我们所说的 Flink on Beam 更精确的说是 PyFlink on Beam Portability Framework。我们看一下简单的架构图，如下：



Beam Portability Framework 是一个成熟的多语言支持框架，框架高度抽象了语言之间的通信协议 (gRPC), 定义了数据的传输格式 (Protobuf), 并且根据通用流计算框架所需要的组件，抽象个各种服务，比如 DataService, StateService, MetricsService 等。在这样一个成熟的框架下，PyFlink 可以快速的构建自己的 Python 算子，同时重用 Apache Beam Portability Framework 中现有 SDK

harness 组件，可以支持多种 Python 运行模式，如：Process，Docker，etc.，这使得 PyFlink 对 Python UDF 的支持变得非常容易，在 Apache Flink 1.10 中的功能也非常的稳定和完整。那么为啥说是 Apache Flink 和 Apache Beam 共同打造呢，是因为我发现目前 Apache Beam Portability Framework 的框架也存在很多优化的空间，所以我在 Beam 社区进行了[优化讨论](#)，并且在 Beam 社区也贡献了[20+ 的优化补丁](#)。

概要了解了 Apache Flink 1.10 中 Python UDF 的架构之后，我们还是切入的代码部分，看看如何开发和使用 Python UDF。

如何定义 Python UDF

在 Apache Flink 1.10 中我们有多种方式进行 UDF 的定义，比如：

- Extend ScalarFunction, e.g.:

```
class HashCodeMean(ScalarFunction):
    def eval(self, i, j):
        return (hash(i) + hash(j)) / 2
```

- Lambda Function

```
lambda i, j: (hash(i) + hash(j)) / 2
```

- Named Function

```
def hash_code_mean(i, j):
    return (hash(i) + hash(j)) / 2
```

- Callable Function

```
class CallableHashCodeMean(object):
    def __call__(self, i, j):
        return (hash(i) + hash(j)) / 2
```

我们发现上面定义函数除了第一个扩展 ScalaFunction 的方式是 PyFlink 特有的，其他方式都是 Python 语言本身就支持的，也就是说，在 Apache Flink 1.10 中

PyFlink 允许以任何 Python 语言所支持的方式定义 UDF。

如何使用 Python UDF

那么定义完 UDF 我们应该怎样使用呢？Apache Flink 1.10 中提供了 2 种 Decorators，如下：

- Decorators – udf(), e.g. :

```
udf(lambda i, j: (hash(i) + hash(j)) / 2,  
    [for input types], [for result types])
```

```
Decorators - @udf, e.g. :  
@udf(input_types=..., result_type=...)  
    def hash_code_mean(...):  
        return ...
```

然后在使用之前进行注册，如下：

```
st_env.register_function("hash_code", hash_code_mean)
```

接下来就可以在 Table API/SQL 中进行使用了，如下：

```
my_table.select("hash_code_mean(a, b)").insert_into("Results")
```

目前为止，我们已经完成了 Python UDF 的定义，声明和注册了。接下来我们还是看一个完整的示例吧：)

案例描述

- 需求

假设苹果公司要统计该公司产品在双 11 期间各城市的销售数量和销售金额分布情况。

- 数据格式

每一笔订单是一个字符串，字段用逗号分隔，例如：

```

ItemName, OrderCount, Price, City
-----
iPhone 11, 30, 5499, Beijing\n
iPhone 11 Pro, 20, 8699, Guangzhou\n

```

案例分析

根据案例的需求和数据结构分析，我们需要对原始字符串进行结构化解析，那么需要一个按“,”号分隔的 UDF(split) 和一个能够将各个列信息展平的 DUF(get)。同时我们需要根据城市进行分组统计。

核心实现

UDF 定义

- Split UDF

```

@udf(input_types=[DataTypes.STRING()],
      result_type=DataTypes.ARRAY(DataTypes.STRING()))
def split(line):
    return line.split(",")

```

- Get UDF

```

@udf(input_types=[DataTypes.ARRAY(DataTypes.STRING()), DataTypes.INT()],
      result_type=DataTypes.STRING())
def get(array, index):
    return array[index]

```

注册 UDF

- 注册 Split UDF

```
t_env.register_function("split", split)
```

- 注册 Get UDF

```
t_env.register_function("get", get)
```

核心实现逻辑

如下代码我们发现核心实现逻辑非常简单，只需要对数据进行解析和对数据进行集合计算：

```
t_env.from_table_source(SocketTableSource(port=9999))\
    .alias("line")\
    .select("split(line) as str_array")\
    .select("get(str_array, 3) as city, "
           "get(str_array, 1).cast(LONG) as count, "
           "get(str_array, 2).cast(LONG) as unit_price")\
    .select("city, count, count * unit_price as total_price")\
    .group_by("city")\
    .select("city, sum(count) as sales_volume, sum(total_price) as sales")\
    .insert_into("sink")
t_env.execute("Sales Statistic")
```

上面的代码我们假设是一个 Socket 的 Source，Sink 是一个 Chart Sink，那么最终运行效果图，如下：



我总是认为在博客中只是文本描述而不能让读者真正的在自己的机器上运行起来的博客，不是好博客，所以接下来我们看看按照我们下面的操作，是否能在你的机器上也运行起来？ :)

环境

因为目前 PyFlink 还没有部署到 PyPI 上面，在 Apache Flink 1.10 发布之

前，我们需要通过构建 Flink 的 master 分支源码来构建运行我们 Python UDF 的 PyFlink 版本。

源代码编译

在进行编译代码之前，我们需要你已经安装了 [JDK8](#) 和 [Maven3x](#)。

- 下载解压

```
tar -xvf apache-maven-3.6.1-bin.tar.gz
mv -rf apache-maven-3.6.1 /usr/local/
```

- 修改环境变量 (~/.bashrc)

```
MAVEN_HOME=/usr/local/apache-maven-3.6.1
export MAVEN_HOME
export PATH=${PATH}:${MAVEN_HOME}/bin
```

除了 JDK 和 MAVEN 完整的环境依赖性如下：

- JDK 1.8+ (1.8.0_211)
- Maven 3.x (3.2.5)
- Scala 2.11+ (2.12.0)
- Python 3.6+ (3.7.3)
- Git 2.20+ (2.20.1)
- Pip3 19+ (19.1.1)

我们看到基础环境安装比较简单，我这里就不每一个都贴出来了。如果大家有问题欢迎邮件或者博客留言。

- 下载 Flink 源代码：

```
git clone https://github.com/apache/flink.git
```

- 编译

```
cd flink
mvn clean install -DskipTests
...
[INFO] flink-walkthrough-datastream-scala ..... SUCCESS [ 0.192 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 18:34 min
[INFO] Finished at: 2019-12-04T23:03:25+08:00
[INFO] -----
```

- 构建 PyFlink 发布包

```
cd flink-python; python3 setup.py sdist bdist_wheel
...
adding 'apache_flink-1.10.dev0.dist-info/WHEEL'
adding 'apache_flink-1.10.dev0.dist-info/top_level.txt'
adding 'apache_flink-1.10.dev0.dist-info/RECORD'
removing build/bdist.macosx-10.14-x86_64/wheel
```

- 安装 PyFlink(PyFlink 1.10 需要 Python3.6+)

```
pip3 install dist/*.tar.gz
...
Successfully installed apache-beam-2.15.0 apache-flink-1.10.dev0 avro-
python3-1.9.1
cloudpickle-1.2.2 crcmod-1.7 dill-0.2.9 docopt-0.6.2 fastavro-0.21.24
future-0.18.2 grpcio-1.25.0
hdfs-2.5.8 httplib2-0.12.0 mock-2.0.0 numpy-1.17.4 oauth2client-3.0.0 pbr-
5.4.4 protobuf-3.11.1
pyarrow-0.14.1 pyasn1-0.4.8 pyasn1-modules-0.2.7 pydot-1.4.1 pymongo-3.9.0
pyyaml-3.13 rsa-4.0
```

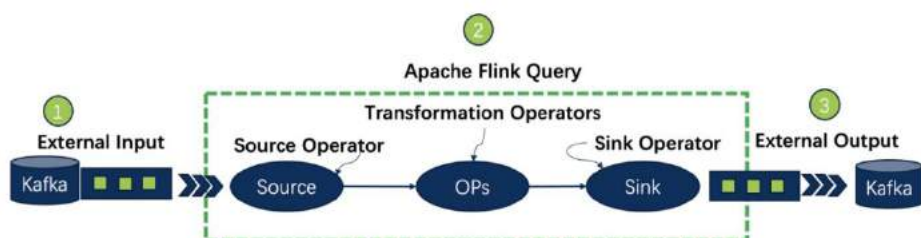
也可以查看一下，我们核心需要 apache-beam 和 apache-flink，如下命令：

```
jincheng:flink-python jincheng.sunjc$ pip3 list
Package                                Version
-----
alabaster                              0.7.12
```

apache-beam	2.15.0
apache-flink	1.10.dev0
atomicwrites	1.3.0

如上信息证明你我们所需的 Python 依赖已经没问题了，接下来回过头来在看看如何进行业务需求的开发。

PyFlink 的 Job 结构



一个完成的 PyFlink 的 Job 需要有外部数据源的定义，有业务逻辑的定义和最终计算结果输出的定义。也就是 Source connector, Transformations, Sink connector，接下来我们根据这个三个部分进行介绍来完成我们的需求。

Source Connector

我们需要实现一个 Socket Connector，首先要实现一个 StreamTableSource，核心代码是实现 getDataStream，代码如下：

```
@Override
public DataStream<Row> getDataStream(StreamExecutionEnvironment env) {
    return env.socketTextStream(hostname, port, lineDelimiter, MAX_RETRY)
        .flatMap(new Splitter(fieldNames.length, fieldDelimiter, appendProctime))
        .returns(getReturnType());
}
```

上面代码利用了 StreamExecutionEnvironment 中现有 socketTextStream 方法接收数据，然后将业务订单数据传个一个 FlatMapFunction, FlatMapFunction

主要实现将数据类型封装为 Row, 详细代码查阅 [Splitter](#)。

同时, 我们还需要在 Python 封装一个 SocketTableSource, 详情查阅 [socket_table_source.py](#)。

Sink Connector

我们预期要得到的一个效果是能够将结果数据进行图形化展示, 简单的思路是将数据写到一个本地的文件, 然后在写一个 HTML 页面, 使其能够自动更新结果文件, 并展示结果。所以我们还需要自定义一个 Sink 来完成该功能, 我们的需求计算结果是会不断的更新的, 也就是涉及到 Retraction (如果大家不理解这个概念, 可以查阅我以前的博客), 目前在 Flink 里面还没有默认支持 Retract 的 Sink, 所以我们需要自定义一个 RetractSink, 比如我们实现一下 CsvRetractTableSink。

CsvRetractTableSink 的核心逻辑是缓冲计算结果, 每次更新进行一次全量 (这是个纯 demo, 不能用于生产环境) 文件输出。源代码查阅 [CsvRetractTableSink](#)。

同时我们还需要利用 Python 进行封装, 详见 chart_table_sink.py。

在 chart_table_sink.py 我们封装了一个 http server, 这样我们可以在浏览器中查阅我们的统计结果。

业务逻辑

完成自定义的 Source 和 Sink 之后我们终于可以进行业务逻辑的开发了, 其实整个过程自定义 Source 和 Sink 是最麻烦的, 核心计算逻辑似乎要简单的多。

- 设置 Python 版本 (很重要)

如果你本地环境 python 命令版本是 2.x, 那么需要对 Python 版本进行设置, 如下:

```
t_env.get_config().set_python_executable("python3")
```

PyFlink 1.10 之后支持 Python 3.6+ 版本。

- 读取数据源

PyFlink 读取数据源非常简单，如下：

```
...
...
t_env.from_table_source(SocketTableSource(port=9999)).alias("line")
```

上面这一行代码定义了监听端口 9999 的数据源，同时结构化 Table 只有一个名为 line 的列。

- 解析原始数据

我们需要对上面列进行分析，为了演示 Python UDF，我们在 SocketTableSource 中并没有对数据进行预处理，所以我们利用上面 UDF 定义一节定义的 UDF，来对原始数据进行预处理。

```
...
...
.select("split(line) as str_array")
.select("get(str_array, 3) as city, " "get(str_array, 1).cast(LONG) as count,
       " "get(str_array, 2).cast(LONG)
       as unit_price")
.select("city, count, count * unit_price as total_price")
```

- 统计分析

核心的统计逻辑是根据 city 进行分组，然后对 销售数量和销售金额进行求和，如下：

```
...
...
.group_by("city")
.select("city, sum(count) as sales_volume, sum(total_price)
       as sales")\
```

- 计算结果输出

计算结果写入到我们自定义的 Sink 中，如下：

```
...
...
.insert_into("sink")
```

- 完整的代码 (blog_demo.py)

```
from pyflink.datastream import StreamExecutionEnvironment
from pyflink.demo import ChartConnector, SocketTableSource
from pyflink.table import StreamTableEnvironment, EnvironmentSettings,
DataTypes
from pyflink.table.descriptors import Schema
from pyflink.table.udf import udf

env = StreamExecutionEnvironment.get_execution_environment()
t_env = StreamTableEnvironment.create(
    env,
    environment_settings=EnvironmentSettings.new_instance().use_blink_
planner().build())
t_env.connect(ChartConnector())\
    .with_schema(Schema()\
        .field("city", DataTypes.STRING())\
        .field("sales_volume", DataTypes.BIGINT())\
        .field("sales", DataTypes.BIGINT()))\
    .register_table_sink("sink")

@udf(input_types=[DataTypes.STRING()],
    result_type=DataTypes.ARRAY(DataTypes.STRING()))
def split(line):
    return line.split(",")

@udf(input_types=[DataTypes.ARRAY(DataTypes.STRING()), DataTypes.INT()],
    result_type=DataTypes.STRING())
def get(array, index):
    return array[index]

t_env.get_config().set_python_executable("python3")

t_env.register_function("split", split)
t_env.register_function("get", get)
t_env.from_table_source(SocketTableSource(port=6666))\
    .alias("line")\
    .select("split(line) as str_array")\
    .select("get(str_array, 3) as city, "
        "get(str_array, 1).cast(LONG) as count, "
        "get(str_array, 2).cast(LONG) as unit_price")\
    .select("city, count, count * unit_price as total_price")\
```

```
.group_by("city")\  
.select("city, "  
        "sum(count) as sales_volume, "  
        "sum(total_price) as sales")\  
.insert_into("sink")  
  
t_env.execute("Sales Statistic")
```

上面代码中大家会发现一个陌生的部分，就是 `from pyflink.demo import ChartConnector, SocketTableSource`。其中 `pyflink.demo` 是哪来的呢？其实就是包含了上面我们介绍的 自定义 Source/Sink (Java&Python)。下面我们来介绍如何增加这个 `pyflink.demo` 模块。

安装 pyflink.demo

为了大家方便我把自定义 Source/Sink (Java&Python) 的源代码放到了这里，大家可以进行如下操作：

- 下载源码

```
git clone https://github.com/sunjincheng121/enjoyment.code.git
```

- 编译源码

```
cd enjoyment.code/PyUDFDemoConnector/; mvn clean install
```

- 构建发布包

```
python3 setup.py sdist bdist_wheel  
...  
...  
adding 'pyflink_demo_connector-0.1.dist-info/WHEEL'  
adding 'pyflink_demo_connector-0.1.dist-info/top_level.txt'  
adding 'pyflink_demo_connector-0.1.dist-info/RECORD'  
removing build/bdist.macosx-10.14-x86_64/wheel
```

- 安装 Pyflink.demo

```
pip3 install dist/pyflink-demo-connector-0.1.tar.gz
...
...
Successfully built pyflink-demo-connector
Installing collected packages: pyflink-demo-connector
Successfully installed pyflink-demo-connector-0.1
```

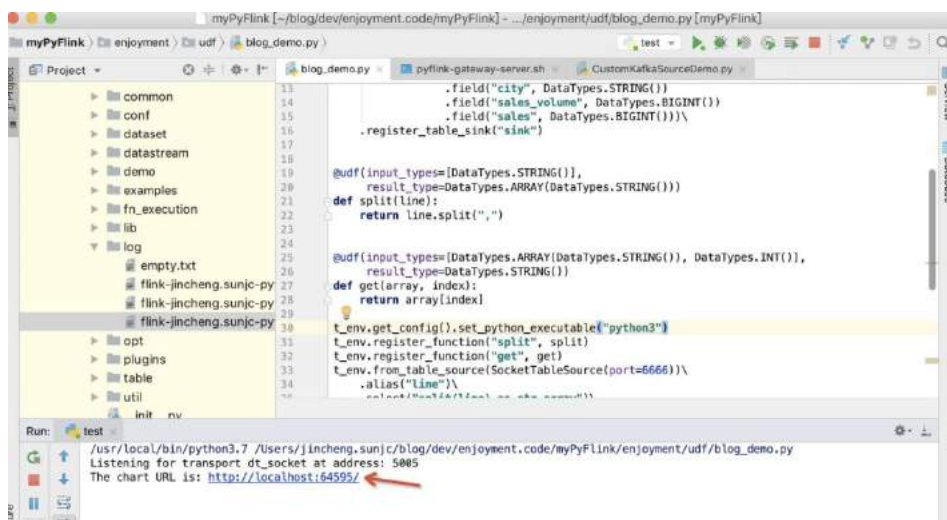
出现上面信息证明已经将 PyFlink.demo 模块成功安装。接下来我们可以运行我们的示例了:)

运行示例

示例的代码在上面下载的源代码里面已经包含了, 为了简单, 我们利用 PyCharm 打开 enjoyment.code/myPyFlink。同时在 Terminal 启动一个端口:

```
nc -l 6666
```

启动 blog_demo, 如果一切顺利, 启动之后, 控制台会输出一个 web 地址, 如下所示:



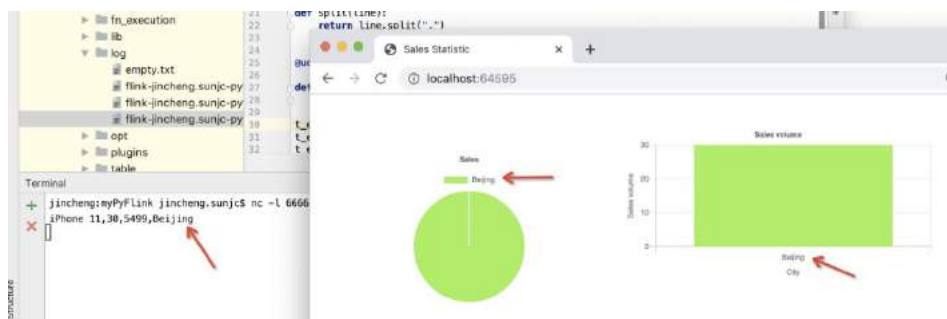
我们打开这个页面，开始是一个空白页面，如下：



我们尝试将下面的数据，一条，一条的发送给 Source Connector：

```
iPhone 11,30,5499,Beijing
iPhone 11 Pro,20,8699,Guangzhou
MacBook Pro,10,9999,Beijing
AirPods Pro,50,1999,Beijing
MacBook Pro,10,11499,Shanghai
iPhone 11,30,5999,Shanghai
iPhone 11 Pro,20,9999,Shenzhen
MacBook Pro,10,13899,Hangzhou
iPhone 11,10,6799,Beijing
MacBook Pro,10,18999,Beijing
iPhone 11 Pro,10,11799,Shenzhen
MacBook Pro,10,22199,Shanghai
AirPods Pro,40,1999,Shanghai
```

当输入第一条订单 iPhone 11,30,5499,Beijing, 之后，页面变化如下：



随之订单数据的不断输入，统计图不断变化。一个完整的 GIF 演示如下：



小结

本篇从架构到 UDF 接口定义，再到具体的实例，向大家介绍了在 Apache Flink 1.10 发布之后，如何利用 PyFlink 进行业务开发，其中 用户自定义 Source 和 Sink 部分比较复杂，这也是目前社区需要进行改进的部分 (Java/Scala)。真正的核心逻辑部分其实比较简单，为了大家按照本篇进行实战操作有些成就感，所以我增加了自定义 Source/Sink 和图形化部分。但如果大家想简化实例的实现也可以利用 Kafka 作为 Source 和 Sink，这样就可以省去自定义的部分，做起来也会简单一些。

| Flink 1.10 Native Kubernetes 原理与实践

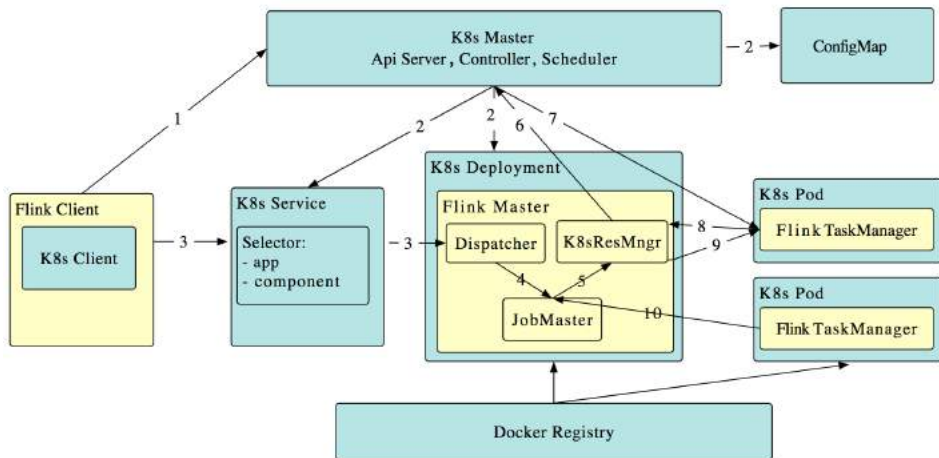
作者：周凯波（宝牛） | 阿里巴巴技术专家

千呼万唤始出来，在 Kubernetes 如火如荼的今天，Flink 社区终于在 1.10 版本提供了对 Kubernetes 的原生支持，也就是 [Native Kubernetes Integration](#)。不过还只是 Beta 版本，预计会在 1.11 版本里面提供完整的支持。

我们知道，在 Flink 1.9 以及之前的版本里面，如果要在 Kubernetes 上运行 Flink 任务是需要事先指定好需要的 TaskManager(TM) 的个数以及 CPU 和内存的。这样的问题是：大多数情况下，你在任务启动前根本无法精确的预估这个任务需要多少个 TM。如果指定的 TM 多了，会导致资源浪费；如果指定的 TM 个数少了，会导致任务调度不起来。本质原因是在 Kubernetes 上运行的 Flink 任务并没有直接向 Kubernetes 集群去申请资源。

Flink 在 1.10 版本完成了 `Active Kubernetes Integration` 的第一阶段，支持了 session clusters。后续的第二阶段会提供更完整的支持，如支持 per-job 任务提交，以及基于原生 Kubernetes API 的高可用，支持更多的 Kubernetes 参数如 toleration, label 和 node selector 等。`Active Kubernetes Integration` 中的 `Active` 意味着 Flink 的 ResourceManager (KubernetesResourceManager) 可以直接和 Kubernetes 通信，按需申请新的 Pod，类似于 Flink 中对 Yarn 和 Mesos 的集成所做的那样。在多租户环境中，用户可以利用 Kubernetes 里面的 namespace 做资源隔离启动不同的 Flink 集群。当然，Kubernetes 集群中的用户帐号和赋权是需要提前准备好的。

原理



工作原理如下（段首的序号对应图中箭头所示的数字）：

1. Flink 客户端首先连接 Kubernetes API Server，提交 Flink 集群的资源描述文件，包括 configmap，job manager service，job manager deployment 和 [Owner Reference](#)。
2. Kubernetes Master 就会根据这些资源描述文件去创建对应的 Kubernetes 实体。以我们最关心的 job manager deployment 为例，Kubernetes 集群中的某个节点收到请求后，Kubelet 进程会从中央仓库下载 Flink 镜像，准备和挂载 volume，然后执行启动命令。在 flink master 的 pod 启动后，Dispatcher 和 KubernetesResourceManager 也都启动了。

前面两步完成后，整个 Flink session cluster 就启动好了，可以接受提交任务请求了。

1. 用户可以通过 Flink 命令行即 flink client 往这个 session cluster 提交任务。此时 job graph 会在 flink client 端生成，然后和用户 jar 包一起通过 RestClnet 上传。

2. 一旦 job 提交成功, JobSubmitHandler 收到请求就会提交 job 给 Dispatcher。接着就会生成一个 job master。
3. JobMaster 向 KubernetesResourceManager 请求 slots。
4. KubernetesResourceManager 从 Kubernetes 集群分配 TaskManager。每个 TaskManager 都是具有唯一表示的 Pod。KubernetesResourceManager 会为 TaskManager 生成一份新的配置文件, 里面有 Flink Master 的 service name 作为地址。这样在 Flink Master failover 之后, TaskManager 仍然可以重新连上。
5. Kubernetes 集群分配一个新的 Pod 后, 在上面启动 TaskManager。
6. TaskManager 启动后注册到 SlotManager。
7. SlotManager 向 TaskManager 请求 slots。
8. TaskManager 提供 slots 给 JobMaster。然后任务就会被分配到这个 slots 上运行。

实践

Flink 的[文档](#)上对如何使用已经写的比较详细了, 不过刚开始总会踩到一些坑。如果对 Kubernetes 不熟, 可能会花点时间。

(1) 首先得有个 Kubernetes 集群, 会有个 `~/.kube/config` 文件。尝试执行 `kubectl get nodes` 看下集群是否正常。

如果没有这个 `~/.kube/config` 文件, 会报错:

```
2020-02-17 22:27:17,253 WARN io.fabric8.kubernetes.client.Config
    - Error reading service
    account token from: [/var/run/secrets/kubernetes.io/serviceaccount/token].
    Ignoring.
2020-02-17 22:27:17,437 ERROR org.apache.flink.kubernetes.cli.
KubernetesSessionCli    - Error while
running the Flink session.
io.fabric8.kubernetes.client.KubernetesClientException: Operation: [get] for
kind: [Service] with
name: [flink-cluster-81832d75-662e-40fd-8564-cd5a902b243c] in namespace:
```

```
[default] failed.
    at io.fabric8.kubernetes.client.KubernetesClientException.
laundryThrowable(KubernetesClientException.java:64)
    at io.fabric8.kubernetes.client.KubernetesClientException.
laundryThrowable(KubernetesClientException.java:72)
    at io.fabric8.kubernetes.client.dsl.base.BaseOperation.
getMandatory(BaseOperation.java:231)
    at io.fabric8.kubernetes.client.dsl.base.BaseOperation.get(BaseOperation.
java:164)
    at org.apache.flink.kubernetes.kubeclient.Fabric8FlinkKubeClient.
getService(Fabric8FlinkKubeClient.java:334)
    at org.apache.flink.kubernetes.kubeclient.Fabric8FlinkKubeClient.
getInternalService(Fabric8FlinkKubeClient.java:246)
    at org.apache.flink.kubernetes.cli.KubernetesSessionCli.
run(KubernetesSessionCli.java:104)
    at org.apache.flink.kubernetes.cli.KubernetesSessionCli.
lambda$main$0(KubernetesSessionCli.java:185)
    at org.apache.flink.runtime.security.NoOpSecurityContext.
runSecured(NoOpSecurityContext.java:30)
    at org.apache.flink.kubernetes.cli.KubernetesSessionCli.
main(KubernetesSessionCli.java:185)
Caused by: java.net.UnknownHostException: kubernetes.default.svc: nodename
nor servname provided, or not known
```

(2) 提前创建好用户和赋权 (RBAC)

```
kubectl create serviceaccount flink
kubectl create clusterrolebinding flink-role-binding-flink --clusterrole=edit
--serviceaccount=default:flink
如果没有创建用户，使用默认的用户去提交，会报错：
Caused by: io.fabric8.kubernetes.client.KubernetesClientException: Failure
executing: GET at:
https://10.10.0.1/api/v1/namespaces/default/pods?labelSelector=app%3Dkaibo-
test%2Ccomponent
%3Dtaskmanager%2Ctype%3Dflink-native-kubernetes.

Message: Forbidden!Configured service account doesn't have access.
Service account may have been revoked. pods is forbidden:
User "system:serviceaccount:default:default" cannot list resource "pods" in
API group "" in the
namespace "default".
```

(3) 这一步是可选的。默认情况下，JobManager 和 TaskManager 只会将 log 写到各自 pod 的 /opt/flink/log。如果想通过 kubectl logs 看到日志，需要将 log 输出到控制台。要做如下修改 FLINK_HOME/conf 目录下的 log4j.properties 文件。

```
log4j.rootLogger=INFO, file, console

# Log all infos to the console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS}
%-5p %-60c %x - %m%n
```

然后启动 session cluster 的命令行需要带上参数：

```
-Dkubernetes.container-start-command-template="%java% %classpath% %jvmem%
%jvmopts% %logging% %class% %args%"
```

(4) 终于可以开始启动 session cluster 了。如下命令是启动一个每个 Task-Manager 是 4G 内存，2 个 CPU，4 个 slot 的 session cluster。

```
bin/kubernetes-session.sh -Dkubernetes.container-start-command-
template="%java% %classpath%
%jvmem% %jvmopts% %logging% %class% %args%" -Dkubernetes.cluster-id=kaibo-
test
-Dtaskmanager.memory.process.size=4096m -Dkubernetes.taskmanager.cpu=2
-Dtaskmanager.numberOfTaskSlots=4
```

更多的参数详见文档：<https://ci.apache.org/projects/flink/flink-docs-release-1.10/ops/config.html#kubernetes>

使用 `kubect1 logs kaibo-test-6f7dffcbcf-c2p7g -f` 就能看到日志了。

如果出现大量的这种日志（目前遇到是云厂商的 LoadBalance liveness 探测导致）：

```
2020-02-17 14:58:56,323 WARN org.apache.flink.runtime.dispatcher.
DispatcherRestEndpoint -
Unhandled exception
java.io.IOException: Connection reset by peer
    at sun.nio.ch.FileDispatcherImpl.read0(Native Method)
    at sun.nio.ch.SocketDispatcher.read(SocketDispatcher.java:39)
    at sun.nio.ch.IOUtil.readIntoNativeBuffer(IOUtil.java:223)
    at sun.nio.ch.IOUtil.read(IOUtil.java:192)
```

```
at sun.nio.ch.SocketChannelImpl.read(SocketChannelImpl.java:377)
at org.apache.flink.shaded.netty4.io.netty.buffer.PooledByteBuf.setBytes(PooledByteBuf.java:247)
```

可以暂时在 log4j.properties 里面配置上:

```
log4j.logger.org.apache.flink.runtime.dispatcher.
DispatcherRestEndpoint=ERROR, file
```

这个日志太多会导致 WebUI 上打开 jobmanger log 是空白, 因为文件太大了前端无法显示。

如果前面第 (1) 和第 (2) 步没有做, 会出现各种异常, 通过 kubectl logs 就能很方便的看到日志了。

Session cluster 启动后可以通过 kubectl get pods,svc 来看是否正常。

通过端口转发来查看 Web UI:

```
kubectl port-forward service/kaibo-test 8081
```

打开 <http://127.0.0.1:8001> 就能看到 Flink 的 WebUI 了。

(5) 提交任务

```
./bin/flink run -d -e kubernetes-session -Dkubernetes.cluster-id=kaibo-test
examples/streaming/
TopSpeedWindowing.jar
```

我们从 Flink WebUI 页面上可以看到, 刚开始启动时, UI 上显示 Total/Available Task Slots 为 0, Task Managers 也是 0。随着任务的提交, 资源会动态增加。任务停止后, 资源就会释放掉。

在提交任务后, 通过 kubectl get pods 能够看到 Flink 为 TaskManager 分配了新的 Pod。


```
(* kubernetes-admin-c5dab8dc5fff44f428cc6d6a2704cb974:default) vvp-services git:(master) kgp
```

NAME	READY	STATUS	RESTARTS	AGE
kaibo-test-9c8c74845-6xldc	1/1	Running	0	10m
kaibo-test-taskmanager-1-1	1/1	Running	0	5m20s
kaibo-test-taskmanager-1-2	1/1	Running	0	2m42s
kaibo-test-taskmanager-1-3	1/1	Running	0	53s

(6) 停止 session cluster

```
echo 'stop' | ./bin/kubernetes-session.sh -Dkubernetes.cluster-id=kaibo-test  
-Dexecution.  
attached=true
```

也可以手工删除资源：

```
kubect1 delete service/<ClusterID>
```

总结

可以看到，Flink 1.10 版本对和 Kubernetes 的集成做了很好的尝试。期待社区后续的 1.11 版本能对 per-job 提供支持，以及和 Kubernetes 的深度集成，例如基于原生 Kubernetes API 的高可用。最新进展请关注 [FLINK-14460](#)。



扫一扫二维码图案，关注我吧



开发者社区



阿里云实时计算



实时计算交流钉钉群



Flink 社区微信公众号