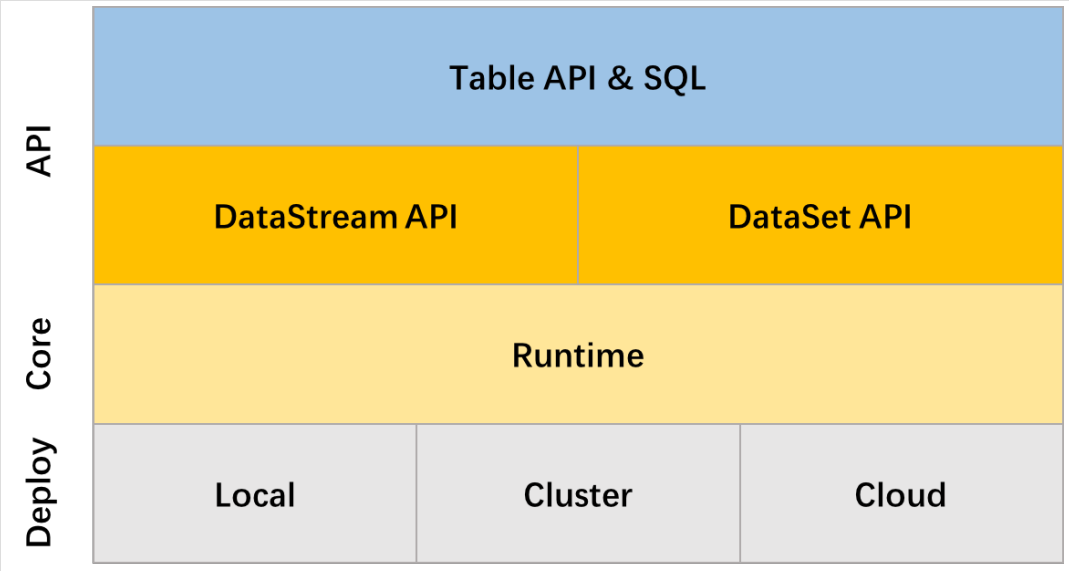
Flink 原理与实现：Table & SQL API

Flink 已经拥有了强大的 DataStream/DataSet API，可以基本满足流计算和批计算中的所有需求。为什么还需要 Table & SQL API 呢?

首先 Table API 是一种关系型API，类 SQL 的API，用户可以像操作表一样地操作数据，非常的直观和方便。用户只需要说需要什么东西，系统就会自动地帮你决定如何最高效地计算它，而不需要像 DataStream 一样写一大堆 Function，优化还得纯靠手工调优。另外，SQL 作为一个“人所皆知”的语言，如果一个引擎提供 SQL，它将很容易被人们接受。这已经是业界很常见的现象了。值得学习的是，Flink 的 Table API 与 SQL API 的实现，有 80% 的代码是共用的。所以当我们讨论 Table API 时，常常是指 Table & SQL API。

Table & SQL API 还有另一个职责，就是流处理和批处理统一的API层。Flink 在runtime层是统一的，因为Flink将批任务看做流的一种特例来执行，这也是 Flink 向外鼓吹的一点。然而在编程模型上，Flink 却为批和流提供了两套API （DataSet 和 DataStream）。为什么 runtime 统一，而编程模型不统一呢？ 在我看来，这是本末倒置的事情。用户才不管你 runtime 层是否统一，用户更关心的是写一套代码。这也是为什么现在 Apache Beam 能这么火的原因。所以 Table & SQL API 就扛起了统一API的大旗，批上的查询会随着输入数据的结束而结束并生成有限结果集，流上的查询会一直运行并生成结果流。Table & SQL API 做到了批与流上的查询具有同样的语法，因此不用改代码就能同时在批和流上跑。



聊聊历史

Table API 始于 Flink 0.9，Flink 0.9 是一个类库百花齐放的版本，众所周知的 Table API, Gelly, FlinkML 都是在这个版本加进去的。Flink 0.9 大概是在2015年6月正式发布的，在 Flink 0.9 发布之前，社区对 SQL 展开过好几次争论，不过当时社区认为应该首先完善 Table API 的功能，再去搞SQL，如果两头一起搞很容易什么都做不好。而且在整个Hadoop生态圈中已经有大量的所谓 “SQL-on-Hadoop” 的解决方案，譬如 Apache Hive, Apache Drill, Apache Impala。”SQL-on-Flink”的事情也可以像 Hadoop 一样丢给其他社区去搞。

不过，随着 Flink 0.9 的发布，意味着抽象语法树、代码生成、运行时函数等都已经成熟，这为SQL的集成铺好了

前进道路。另一方面，用户对 SQL 的呼声越来越高。2015年下半年，Timo 大神也加入了 dataArtisans，于是对 Table API的改造开始了。2016 年初的时候，改造基本上完成了。我们也是在这个时间点发现了 Table API 的潜力，并加入了社区。经过这一年的努力，Flink 已经发展成 Apache 中最火热的项目之一，而 Flink 中最活跃的类库目前非 Table API 莫属。这其中离不开国内公司的支持，Table API 的贡献者绝大多数都来自于阿里巴巴和华为，并且主导着 Table API 的发展方向，这是非常令国人自豪的。而我在社区贡献了一年后，幸运地成为了 Flink Committer。

Table API & SQL 长什么样?

这里不会详细介绍 Table API & SQL 的使用，只是做一个展示。更多使用细节方面的问题请访问官网文档。

下面这个例子展示了如何用 Table API 处理温度传感器数据。计算每天每个以`room`开头的location的平均温度。例子中涉及了如何使用window，event-time等。

```
val sensorData: DataStream[(String, Long, Double)] = ???

// convert DataSet into Table
val sensorTable: Table = sensorData
  .toTable(tableEnv, 'location, 'time, 'tempF)

// define query on Table
val avgTempCTable: Table = sensorTable
  .window(Tumble over 1.day on 'rowtime as 'w)
  .groupBy('location, 'w)
  .select('w.start as 'day, 'location, (('tempF.avg - 32) * 0.556) as 'avgTempC)
  .where('location like "room%")
```

下面的例子是展示了如何用 SQL 来实现。

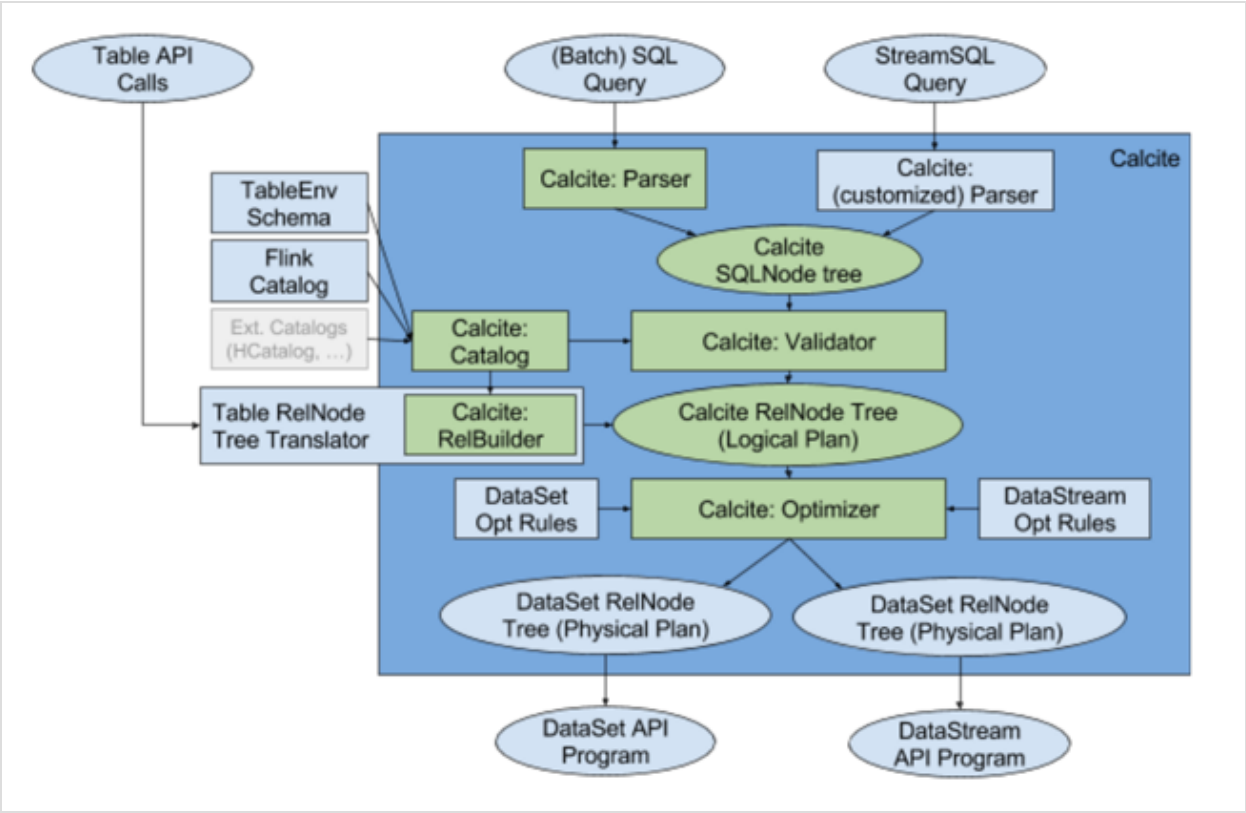
```
val sensorData: DataStream[(String, Long, Double)] = ???

// register DataStream
tableEnv.registerDataStream("sensorData", sensorData, 'location, 'time, 'tempF)

// query registered Table
val avgTempCTable: Table = tableEnv.sql("""
  SELECT FLOOR(rowtime() TO DAY) AS day, location,
    AVG((tempF - 32) * 0.556) AS  avgTempC
  FROM sensorData
  WHERE location LIKE 'room%'
  GROUP BY location, FLOOR(rowtime() TO DAY) """)
```

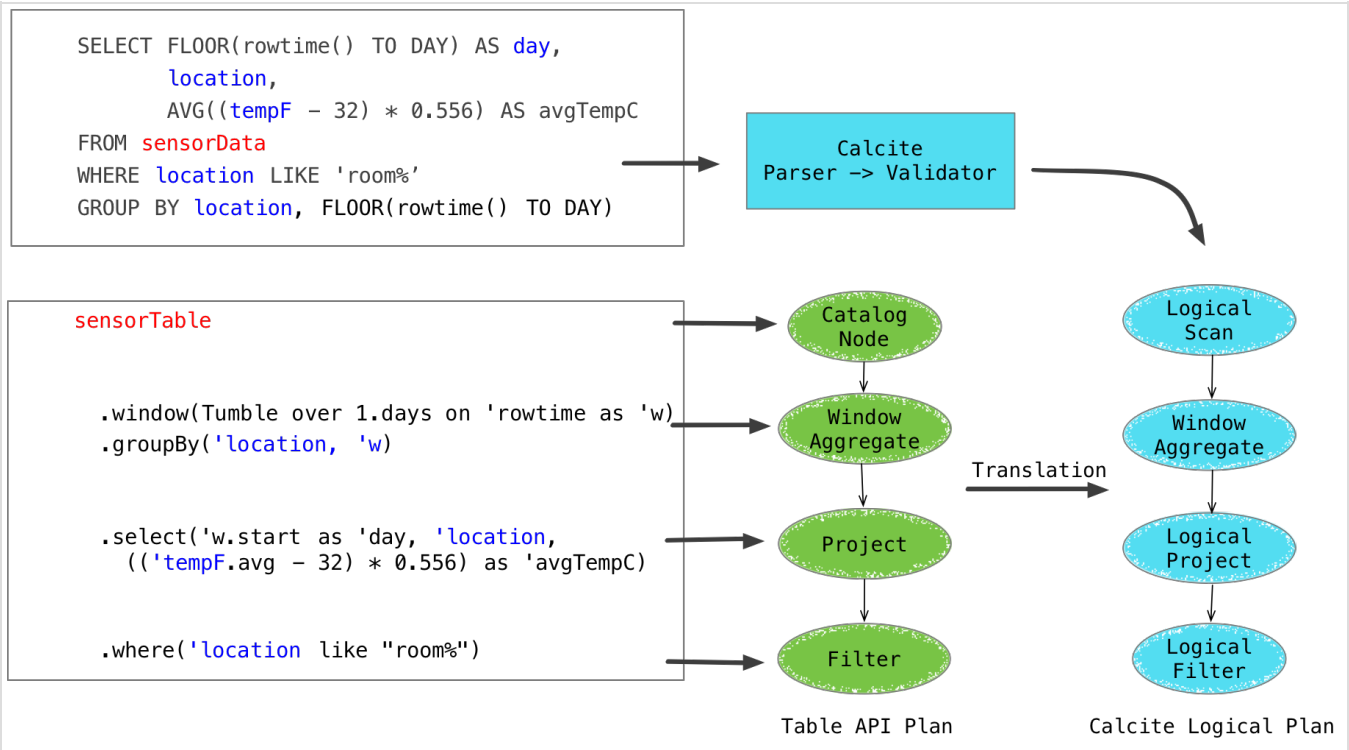
Table API & SQL 原理

Flink 非常明智，没有像Spark那样重复造轮子(Spark Catalyst)，而是将 SQL 校验、SQL 解析以及 SQL 优化交给了 Apache Calcite。Calcite 在其他很多开源项目里也都应用到了，譬如Apache Hive, Apache Drill, Apache Kylin, Cascading。Calcite 在新的架构中处于核心的地位，如下图所示。



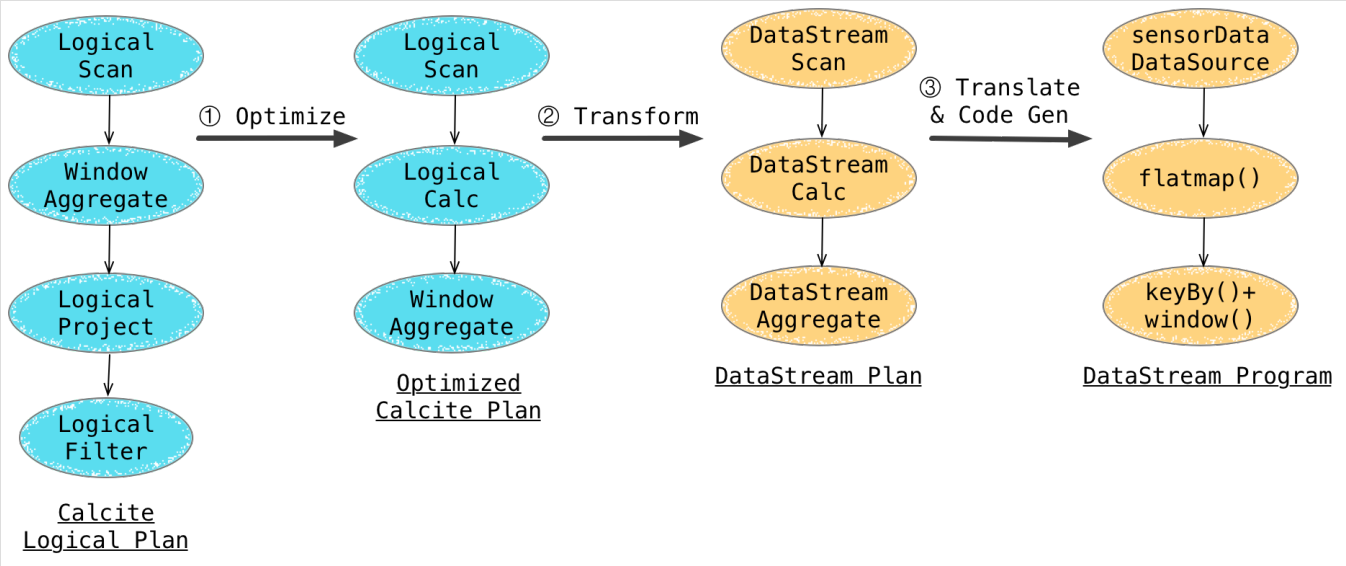
新的架构中，构建抽象语法树的事情全部交给了 Calcite 去做。SQL query 会经过 Calcite 解析器转变成 SQL 节点树，通过验证后构建成 Calcite 的抽象语法树（也就是图中的 Logical Plan）。另一边，Table API 上的调用会构建成 Table API 的抽象语法树，并通过 Calcite 提供的 RelBuilder 转变成 Calcite 的抽象语法树。

以上面的温度计代码为样例，Table API 和 SQL 的转换流程如下，绿色的节点代表 Flink Table Nodes，蓝色的节点代表 Calcite Logical Nodes。最终都转化成了相同的 Logical Plan 表现形式。



之后会进入优化器，Calcite 会基于优化规则来优化这些 Logical Plan，根据运行环境的不同会应用不同的优化规则（Flink提供了批的优化规则，和流的优化规则）。这里的优化规则分为两类，一类是Calcite提供的内置优化规

则（如条件下推，剪枝等），另一类是是将Logical Node转变成 Flink Node 的规则。这两类规则的应用体现为下图中的①和②步骤，这两步骤都属于 Calcite 的优化阶段。得到的 DataStream Plan 封装了如何将节点翻译成对应 DataStream/DataSet 程序的逻辑。步骤③就是将不同的 DataStream/DataSet Node 通过代码生成（CodeGen）翻译成最终可执行的 DataStream/DataSet 程序。



代码生成是 Table API & SQL 中最核心的一块内容。表达式、条件、内置函数等等是需要CodeGen出具体的Function 代码的，这部分跟Spark SQL的结构很相似。CodeGen 出的Function以字符串的形式存在。在提交任务后会分发到各个 TaskManager 中运行，在运行时会使用 Janino 编译器编译代码后运行。

Table API & SQL 现状

目前 Table API 对于批和流都已经支持了基本的Selection, Projection, Union，以及 Window 操作（包括固定窗口、滑动窗口、会话窗口）。SQL 的话由于 Calcite 在最近的版本中才支持 Window 语法，所以目前 Flink SQL 还不支持 Window 的语法。并且 Table API 和 SQL 都支持了UDF,UDTF,UDAF(开发中)。

Table API & SQL 未来

1. Dynamic Tables

   Dynamic Table 就是传统意义上的表，只不过表中的数据是会变化更新的。Flink 提出 Stream <—> Dynamic Table 之间是可以等价转换的。不过这需要引入Retraction机制。有机会的话，我会专门写一篇文章来介绍。

2. Joins

   包括了支持流与流的 Join，以及流与表的 Join。

3. SQL 客户端

   目前 SQL 是需要内嵌到 Java/Scala 代码中运行的，不是纯 SQL 的使用方式。未来需要支持 SQL 客户端执行提交 SQL 纯文本运行任务。

4. 并行度设置

    目前 Table API & SQL 是无法设置并行度的，这使得 Table API 看起来仍像个玩具。

在我看来，Flink 的 Table & SQL API 是走在时代前沿的，在很多方面在做着定义业界标准的事情，比如 SQL 上Window的表达，时间语义的表达，流和批语义的统一等。在我看来，SQL 拥有更天然的流与批统一的特性，并且能够自动帮用户做很多SQL优化（下推、剪枝等），这是 Beam 所做不到的地方。当然，未来如果 Table & SQL API 发展成熟的话，剥离出来作为业界标准的流与批统一的API也不是不可能（叫BeamTable，BeamSQL？），哈哈。这也是我非常看好 Table & SQL API，认为其大有潜力的一个原因。当然就目前来说，需要走的路还很长，Table API 现在还只是个玩具。

参考文献

- Stream Processing for Everyone with SQL and Apache Flink
- From Streams to Tables and Back Again: An Update on Flink's Table & SQL API