彩蛋 2 | 穷途末路的选择：Lambda 架构

在前面的课时中，我们都是在讨论实时计算的问题。但真实的世界里，很多事情都不尽人愿。有时候，因为算法复杂度过高、数据量过大，我们并不能通过直接的实时计算，获得想要的结果。比如，二度关联图谱计算以及一些复杂的统计学习模型或机器学习模型训练等。在这种情况下，我们该如何制定出一个可以真实落地的系统架构方案呢?

这个时候，我们就需要用到压箱底的 Lambda 架构了。

Lambda 架构

什么是 Lambda 架构呢？下面的图 1 进行了说明。

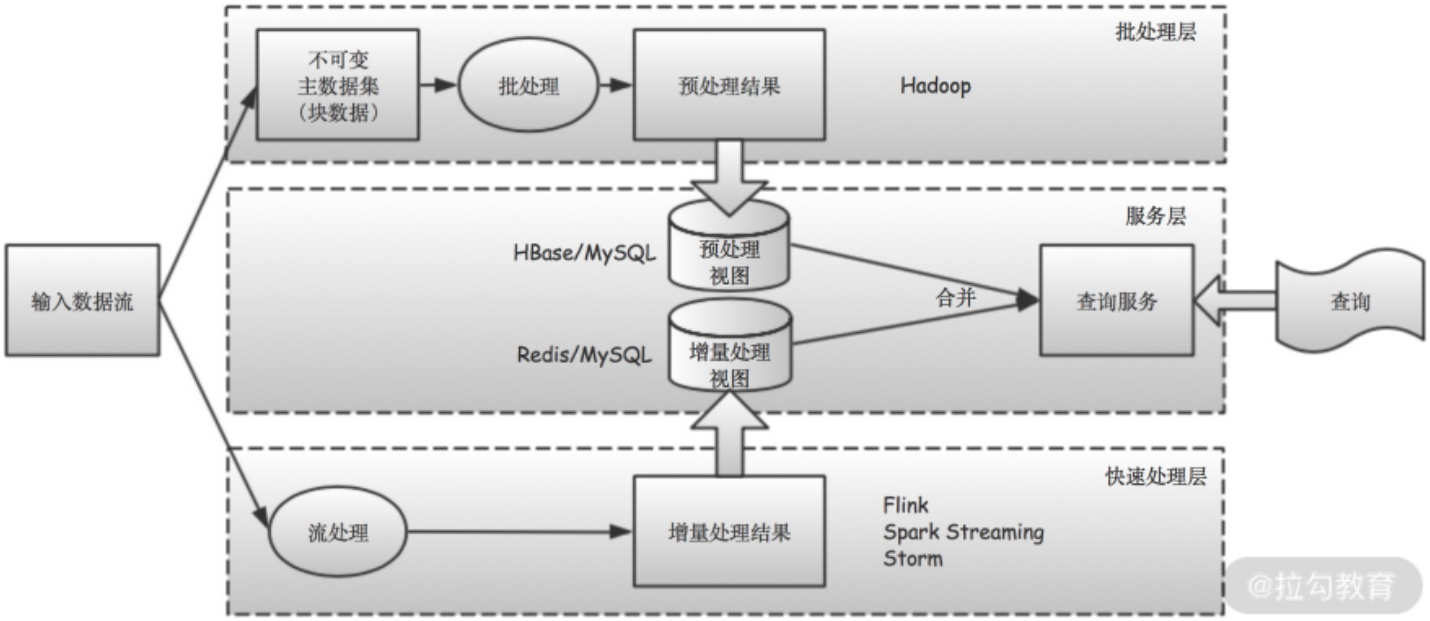

图 1 Lambda 系统架构图

从上面的图 1 可以看出，Lambda 架构总体上分为三层：批处理层（batch layer）、快速处理层（speed layer）和服务层（serving layer），其中：

- 批处理层负责处理主数据集（也就是历史全量数据）；
- 快速处理层负责处理增量数据（也就是新进入系统的数据）；
- 服务层用于将批处理层和快速处理层的结果合并起来，给用户或应用程序提供查询服务。

Lambda 架构是一种架构设计思路，针对每一层的技术组件选型并没有严格限定。我们可以根据自己公司和项目的实际情况，选择相应的技术方案。

对于批处理层，数据存储可以选择 HDFS、S3 等大数据存储系统，而计算工具则可以选择 MapReduce、Hive、Spark 等大数据处理框架。批处理层的计算结果（比如数据库表或者视图），由于需要被服务层或快速处理层快速访问，所以可以存放在诸如 MySQL、HBase 等能够快速响应查询请求的数据库中。

对于快速处理层，这就是各种流计算框架的用武之地了，比如 Flink、Spark Streaming 和 Storm 等。快速处理层由于对性能要求更加严苛，它们的计算结果可以存入像 Redis 这样具有超高性能表现的内存数据库中。不过有时候为了查询方便，也可以将计算结果

存放在 MySQL 等传统数据库中，毕竟这些数据库配合缓存一起使用的话，性能也是非常棒的。

对于服务层，当其接收到查询请求时，就可以分别从存储批处理层和快速处理层计算结果的数据库中，取出相应的计算结果并做合并，就能得到最终的查询结果了。

不过，虽然 Lambda 架构实现了间接的实时计算，但它也存在一些问题。其中最主要的就是，对于同一个查询目标，需要分别为批处理层和快速处理层开发不同的算法实现。也就是说，对于相同的逻辑，需要开发两种不同的代码，并使用两种不同的计算框架（比如同时使用 Storm 和 Spark），这对开发、测试和运维，都带来一定的复杂性和额外工作。

所以，Lambda 架构的改进版本，也就是 Kappa 架构应运而生。

Kappa 架构

下面的图 2 展示了 Kappa 架构的工作原理。

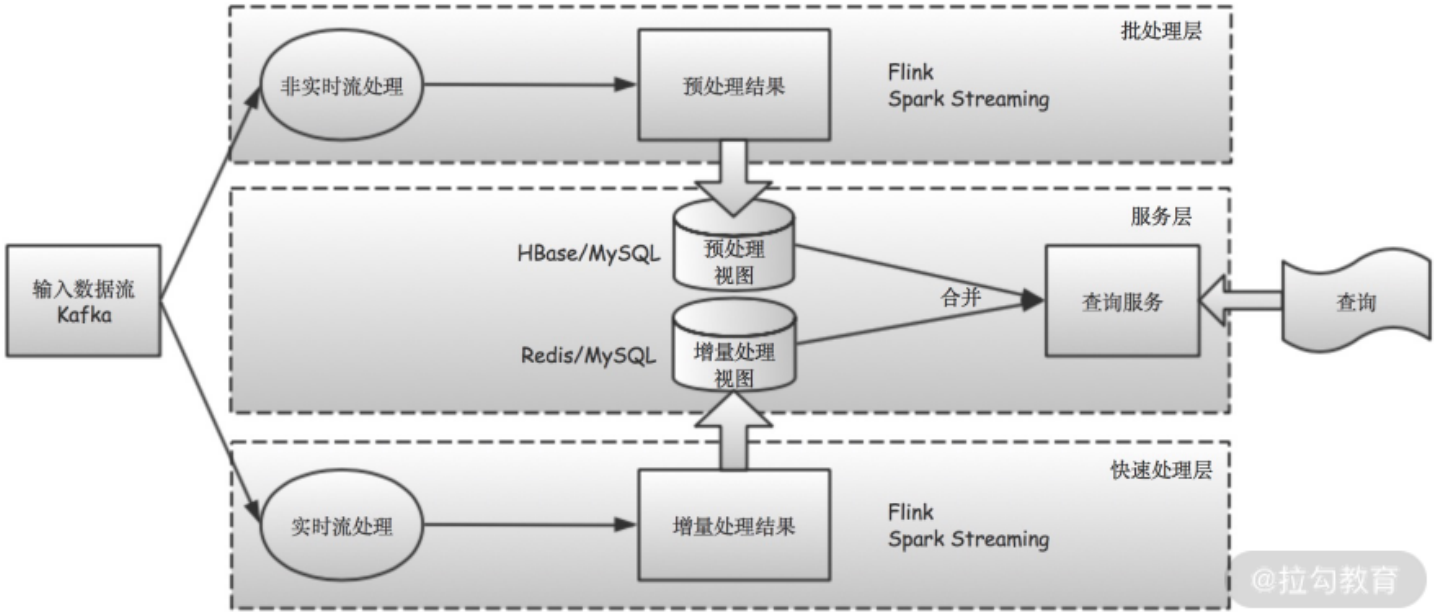

图 2 Kappa 系统架构图

从上面的图 2 可以看出，Kappa 架构相比 Lambda 架构的最大改进，就是将批处理层也用快速处理层的流计算技术所取代。这样一来，批处理层和快速处理层**使用相同的流计算逻辑，并有更统一的计算框架**，从而降低了开发、测试和运维的成本。

另外，由于 Kappa 架构完全使用“流计算”来处理数据，这就让我们在“存储”方面也可以作出调整。我们不必再像在 Lambda 架构中，将离线数据转储到 HDFS、S3 这样的“块数据”存储系统。而只需要将数据按照“流”的方式，存储在 Kafka 这样的“流数据”存储系统里即可。这既减少了数据存储的空间，也避免了不必要的数据转储，同时还降低了系统的复杂程度。

所以说，在 Flink 和 Spark Streaming 等新一代**流批一体计算框架**，以及诸如 Kafka 和 Pulsar 等新一代**流式大数据存储系统**的**双重加持**下，使用 Kappa 架构处理大数据，已经成为一种非常自然的选择。

使用 Flink 实现 Kappa 架构

正所谓光说不练假把式，下面我们就使用 Flink 来演示如何实现 Kappa 架构。

假设现在我们需要统计“最近 3 天每种商品的销售量”。根据 Kappa 架构的思路，我们将这个计算任务，分为离线处理层和快速处理层。

其中离线处理层的实现如下（完整代码参考这里）：

```
DataStream counts = stream
    // 将字符串的数据解析为JSON对象
    .map(new MapFunction<String, Event>() {
        @Override
        public Event map(String s) throws Exception {
            return JSONObject.parseObject(s, Event.class);
        }
    })
    // 提取出每个事件中的商品，转化为商品计数事件
    .map(new MapFunction<Event, CountedEvent>() {
        @Override
        public CountedEvent map(Event event) throws Exception {
            return new CountedEvent(event.product, 1, event.timestamp);
        }
    })
    .assignTimestampsAndWatermarks(new EventTimestampPeriodicWatermarks())
    .keyBy("product")
    // 对于批处理层，使用滑动窗口SlidingEventTimeWindows
.timeWindow(Time.days(3), Time.minutes(30))
// 最后是批处理窗口内的聚合计算
    .reduce((e1, e2) -> {
        CountedEvent countedEvent = new CountedEvent();
        countedEvent.product = e1.product;
        countedEvent.timestamp = e1.timestamp;
        countedEvent.count = e1.count + e2.count;
        countedEvent.minTimestamp = Math.min(e1.minTimestamp, e2.minTimestamp);
        countedEvent.maxTimestamp = Math.max(e1.maxTimestamp, e2.maxTimestamp);
        return countedEvent;
    });
```

在上面的代码中，我们采用了长度为 3 天，步长为 30 分钟的滑动窗口。也就是说，每三十分钟会计算一次三天内各个商品的销售量。

接下来是快速处理层的实现（完整代码参考这里）：

```
DataStream counts = stream
    // 将字符串的数据解析为JSON对象
    .map(new MapFunction<String, Event>() {
        @Override
        public Event map(String s) throws Exception {
            return JSONObject.parseObject(s, Event.class);
        }
    })
    // 提取出每个事件中的商品，转化为商品计数事件
    .map(new MapFunction<Event, CountedEvent>() {
        @Override
        public CountedEvent map(Event event) throws Exception {
            return new CountedEvent(event.product, 1, event.timestamp);
        }
    })
    .assignTimestampsAndWatermarks(new EventTimestampPeriodicWatermarks())
    .keyBy(x -> x.product)
    // 对于批处理层，使用翻转窗口TumblingEventTimeWindows
    .window(TumblingEventTimeWindows.of(Time.seconds(15)))
// 最后是批处理窗口内的聚合计算
    .reduce((e1, e2) -> {
        CountedEvent countedEvent = new CountedEvent();
        countedEvent.product = e1.product;
        countedEvent.timestamp = e1.timestamp;
        countedEvent.count = e1.count + e2.count;
        countedEvent.minTimestamp = Math.min(e1.minTimestamp, e2.minTimestamp);
        countedEvent.maxTimestamp = Math.max(e1.maxTimestamp, e2.maxTimestamp);
        return countedEvent;
    });
```

在上面的代码中，我们采用了长度为 15 秒的翻滚窗口。也就是说，每 15 秒钟会计算一次 15 秒内各个商品的销售量。

从上面两部分的代码中，我们就可以体会到 Kappa 架构的优势所在了。因为，在上面**批处理层**和**快速处理层**的实现中，**除了两个窗口的类型不一样以外，其他的代码完全相同**！是不是非常惊艳？！

接下来，在批处理层和快速处理层各自计算出结果后，需要将计算结果存入数据库，具体如下（完整代码参考这里）：

```
public class JdbcWriter extends RichSinkFunction<CountedEvent> {
    // 将每个窗口内的计算结果保存到数据库中
    private String inset_sql = "INSERT INTO table_counts(id,start,end,product,v_count,layer) VALU
            "ON DUPLICATE KEY UPDATE start=?,end=?,product=?,v_count=?,layer=?;";
    private long slideMS = 0;
    private long slideNumberInWindow = 0;
    private String layer = null;
    public JdbcWriter(long slideMS, long slideNumberInWindow, String layer) {
        this.slideMS = slideMS;
        this.slideNumberInWindow = slideNumberInWindow;
        this.layer = layer;
    }
    @Override
    public void invoke(CountedEvent value, Context context) throws Exception {
        // 通过对滑动或翻滚的步长取整，以对齐时间窗口，从而方便后续合并离线部分和实时部分的计算结果
        long start = value.minTimestamp / slideMS;
        long end = value.minTimestamp / slideMS + slideNumberInWindow;
        String product = value.product;
        int v_count = value.count;
        String layer = this.layer;
        String id = DigestUtils.md5Hex(Joiner.on("&").join(Lists.newArrayList(start, end, product
        preparedStatement.setString(1, id);
        preparedStatement.setLong(2, start);
        preparedStatement.setLong(3, end);
        preparedStatement.setString(4, product);
        preparedStatement.setInt(5, v_count);
        preparedStatement.setString(6, layer);
        preparedStatement.setLong(7, start);
        preparedStatement.setLong(8, end);
        preparedStatement.setString(9, product);
        preparedStatement.setInt(10, v_count);
        preparedStatement.setString(11, layer);
        preparedStatement.executeUpdate();
    }
}
```

在上面的代码中，我们将批处理层和快速处理层的结果都存入了数据库。

最后，服务层只需要使用一条简单的 SQL 语句，就能将批处理层和快速处理层的计算结果合并起来，具体如下（完整代码参考这里）：

```
SELECT product, sum(v_count) as s_count from
(
        SELECT * FROM table_counts WHERE start=? AND end=? AND layer='batch'
        UNION
        SELECT * FROM table_counts WHERE start>=? AND end<=? AND layer='fast'
) as union_table GROUP BY product;
```

在上面的代码中，我们使用 UNION 操作，将批处理层和快速处理层的结果合并起来。然后，在这个合并的表上，通过分组聚合计算，就能非常方便地计算出“最近 3 天每种商品的销售量”这个计算目标了。

小结

今天，我们讨论了采用 Lambda 架构和 Kappa 架构，间接实现实时计算的方法。

总的来说，Lambda 架构是一种通用的架构思想，它告诉我们，**当不能直接做到实时计算时，不妨尝试采用离线和实时相结合的折中计算方案**。而从 Lambda 架构上改进而来的 Kappa 架构，通过“流”来统一“编程界面”，降低了系统、开发和运维的复杂程度。

但是，这并不意味着 Kappa 架构就能够取代 Lambda 架构了。

因为，在实际项目开发过程中，并不是所有的任务都适合用流计算的方式来完成。目前为止，采用批处理方式实现的算法，比采用流处理方式实现的算法，不管是在丰富度、成熟度、还是可用第三方工具库方面，都要优越很多。

另外，是选择将离线计算和实时计算框架统一起来，还是将数据人员（他们已经有很多好用且熟悉的数据分析工具，比如 R、Python、Spark 等）和开发人员各自的生产力和创造力发挥出来，还有待商榷。

所以，我们还是需要根据具体的业务场景、已有技术积累、团队研发能力等多方面因素，设计出最终能够真实落地的方案。

最后，你在实际工作中有没有碰到过，不能够直接实现实时计算的场景呢？如果使用 Lambda 架构或 Kappa 架构的话，你会怎么做？可以将你的想法或问题写在留言区。

下面是本课时的知识脑图。