进击的实时数仓：Flink 在 OPPO 实时计算平台的研发与应用实践

> 为了全面推动数仓实时化，OPPO 基于 Flink 打造的实时计算平台 OStream ，已广泛服务于实时 ETL/ 实时报表 / 实时标签等应用场景。本文主要围绕基于 Flink 打造的实时计算平台 OStream 来展开，分享 OStream 平台的研发之道（包括设计原则、总体架构、Flink 改进优化），业务场景的接入与应用实践，以及平台往智能化方向发展的探索与思考。本文整理自 OPPO 大数据平台研发负责人张俊在 Qcon 全球软件开发大会 2019 广州站的演讲。如果读者正在考虑或者正在建设实时计算平台，希望能给大家带来一些参考。

OPPO 大数据平台介绍

首先介绍一下 OPPO 的业务跟数据规模。OPPO 是一家非常低调的公司，跟互联网、大数据到底有什么关系呢？简单地介绍一下。OPPO 有自己的基于安卓的定制系统 ColorOS，内置很多互联网应用，包括应用商店、浏览器、信息流等一些热门应用。这个系统经过几年的发展，日活现在已超过 2 亿。在业务驱动下，OPPO 的数据量级从 2013 年开始到现在，几乎每年都是 2 到 3 倍的增长速度，所以现在总数据量规模也比较庞大，超过了 100PB，每天新增超过 200TB。

图 1 OPPO 业务与数据规模

在如此大数据量的规模下，不可避免要去构建数据的 Pipeline。我们的主要数据来源是手机端的埋点数据，还有一部分是日志或者是 DB 的数据。我们基于 NiFi 开源的项目构建了整个接入系统，然后基于 HDFS 跟 Hive 做数据的存储跟计算。计算层主要有两部分，第一部分是小时级别的 ETL，做数据清洗跟加工；然后还有一个部分是日级的 Hive 任务，做每天的汇总工作。整个 Pipeline 有一个调度系统，是基于开源 ALflow 做的定制版，我们称之为 OFlow。

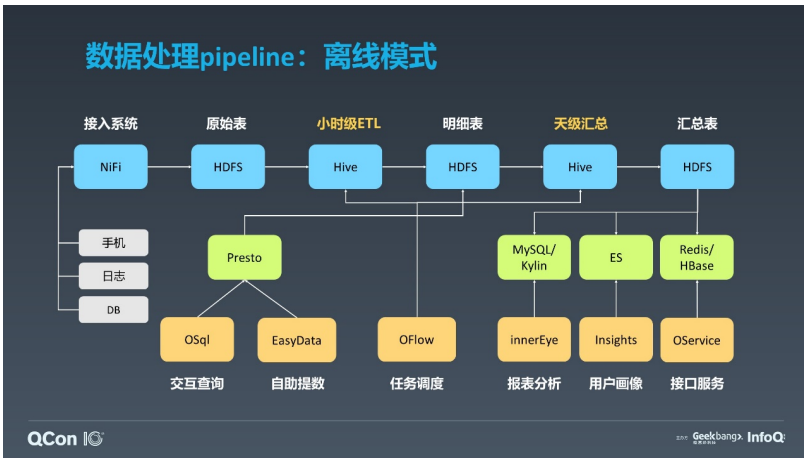

图 2 数据处理 Pipeline：离线模式

接下来是应用层。数据应用主要分三块：报表分析、用户画像、接口服务。也有一些自研产品。每天可以从 Hive 里把数据导到 MySQL/Kylin、ES、Redis/HBase 去支持产品的应用；还会基于 Presto 提供一些交互式查询等服务。这一整个基于离线批处理的 Pipeline，经过 2 到 3 年的沉淀，几乎支撑了所有业务的发展。但是我们发现，最近几年整个互联网的人口红利不断消退，所以我们的业务被迫走向精细化运营的道路。

精细化的一个关键点是及时性，需要捕获用户即时行为跟短期兴趣。最典型的一个应用场景就是实时推荐，相信国内大部分公司都在做实时化推荐的事情。这样的及时性对整个数据处理就会有实时化的诉求。这里所说的实时化，是从过去小时级别、天级别向分钟级别跟秒级别过渡（现在还没有到微秒级别的诉求）。

刚才提到最重要的三块数据的应用——报表、标签、接口，分别列出了实际业务的一些场景。其实除了业务，还有一块容易被我们忽视的，是平台侧对实时化也有诉求。举个典型的例子，OPPO 绝大部分离线调度任务都是在凌晨 0 点启动的。这个可以理解，因为是 T+1 的处理，到了凌晨 0 点就可以处理前一天的数据，所以大家都希望越早处理越好。但这对我们平台来说是一个很大的问题：一到凌晨，整个集群压力就非常大，经常半夜被电话叫起来，处理各种集群问题。对整个平台来说，如果能把批处理变成流式处理，把集中式的集群负载分摊到 24 小时，平台压力就会减少。

另外，如果有实时化流式处理，对整个标签导入，包括质量监控，也是很有帮助的。有了这么一个实时化的诉求之后，各个业务都在尝试搭建自己的实时流 Pipeline。在搭建 Pipeline 时，技术选型最重要的两块，一是计算引擎，二是存储引擎。我们都知道，在开源的生态里，这两块现在是群雄争霸，有很多的选择。对于计算引擎来说，有老牌霸主 Storm，还有 Flink、Spark streaming 这样的新贵；在存储引擎方面，有 Redis、HBase、Elasticsearch 等等。我们要构建一个 Pipeline，计算引擎最终是要对接存储引擎的，很容易就会面临下图这样的局面，像一个很复杂的蜘蛛网。

图 3 实时流处理的乱象

既然是一个数据平台团队，就有责任去构建一个平台收敛这些系统，因此会面临很现实的问题。比如如何说服业务，说现在就需要一个平台，需要你们接入我这个平台？怎么告诉他，我们这个平台是有价值的？站在平台角度，我们会认为这个价值是显而易见的，平台化能产生规模效应。规模效应带来什么呢？研发资源、硬件资源、运营资源的边际成本都可以降低。但反过来站在业务的角度，他不一定认同。他可能会觉得，你讲的是面，但他关注的是点，他会说："我的业务快速发展是当下之急，你的规模效应确实是不错的，但对我业务来说，平台化会给我带来迁移的成本、学习的成本，另外还能给我带来怎样的价值？"

所以对于平台的建设来说，很重要的一点是：从一开始建设这个平台时，就需要站在业务赋能的视角看待这个问题。能不能屏蔽掉底层的一些细节，提高易用性？能不能提供更好的抽象，普适到更多用户？能不能跟业务签订服务等级协议（SLA），给他更好的服务保障？只有说，整个平台化从点到面地推进，先解决某个业务某个点的需求，慢慢把业务接进来，才能产生规模效应，最终，最大化平台的价值。这是我们在平台化推进当中的一个思考。

构建实时计算平台的技术实践

基于这样一个背景，实时计算平台如何推进呢？建设思路是怎样的呢？我们认为，不管构建系统也好，还是说构建一个平台也好，第一优先考虑是顶层设计。顶层设计什么呢？我们认为是两层，上层是 API，下层是 Runtime。"冰山理论"说：你所看到的东西，很可能只是它整体的很小一部分，事实上有更多你所看不到的部分隐藏在水面之下。这理论可以套用在很多地方。

对平台设计来说，我们希望平台的 API 尽可能简化，尽可能抽象，把更多的复杂性留在用户看不到的 Runtime 这一层。所以对 API 考虑是易用性、表达性和灵活性。到 Runtime 这一层，考核的是性能、健壮性和可扩展性。这些都是很复杂的分布式概念。

基于这么一个思路，平台 API 如何选择？既然说这个 API 面对用户，首先要考虑公司的人员分布与使用习惯是什么。

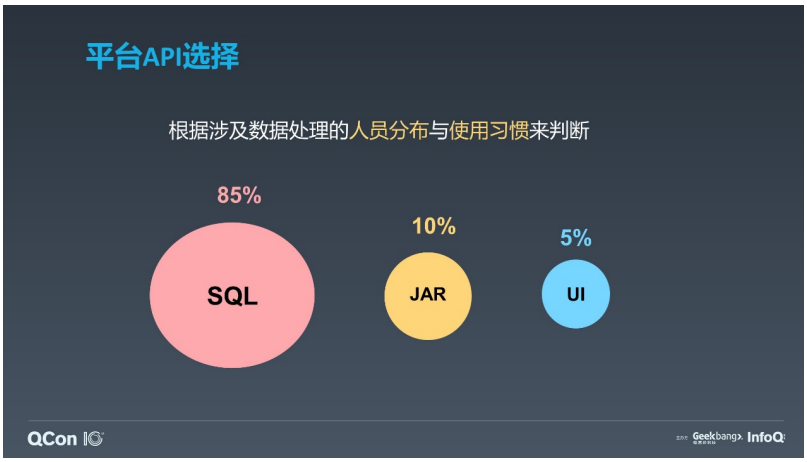

图 4 我司人员使用习惯分布

我们在离线处理时代，几乎都是用 Hive 去做的，所以大部分人员都习惯于写 SQL；可能还有一部分，习惯于写 Scala、Java 等，他们会希望通过 Jar 的方式提交数据处理；还有一部分，可能既不会写 SQL，也不会写程序，会希望有一些简单易用的界面，直接做数据的分析。

可以发现，SQL 是非常重要的平台一等公民。反过来我们再去考核一下 SQL 这个语言，大家都很熟悉，它是一个声明式语言，经过 30、40 年的发展，易用性、灵活性、表达性都很高，可以满足我们刚才提到的设计原则。

再到 Runtime，我们的选择其实非常多。

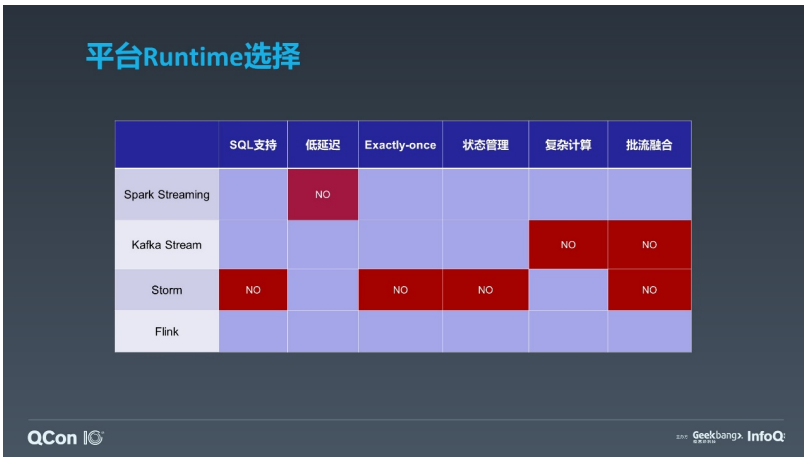

图 5 平台 Runtime 选择

这里仅仅列了几条，把核心诉求列了一个对比矩阵，发现只有 Flink 在各方面都满足需求。其他的引擎，像 Spark Streaming，天然的 micro-batch 方式，没办法达到这种低延时的性能；对 Kafka 来说，它是比较轻量级的框架；对 Storm 来说，现在处于一个英雄垂暮的阶段，很多特性是在它那个时代是没有考虑过的。所以 Flink 相对来说是一个比较好的选择。

可能很多人会说现在 Spark 在发展，它从 Structure streaming 演进过来，也可以支持 Continuous 处理模式，可以支持低延时。但是我们认为，整个技术框架发展，技术最终肯定是趋同的。为什么选择 Flink？还有一个很重要的原因是最近两年 Flink 在国内的发展普及程度。包括像阿里团队，他

们也在大力地宣传跟投入，包括今天 QCon 大会上大沙和云邪两位老师，他们也是阿里团队社区的资深大 V。

回过头来，Flink 这个引擎最核心的优势是什么？哪些对我们来说觉得是一个亮点？首先看 Flink 引擎这一层，它最大的优势是 Runtime，我们希望是高性能的，而它就是低延时、高吞吐。可能很多人会问，低延时跟高吞吐不是矛盾的概念吗？不应该去做一个权衡吗？其实 Flink 也没有去打破这个权衡，其实它内部有所谓的缓冲超时（buffer timeout）的机制，可以向左走，选择非常极端的低延时；也可以向右走，选择极端的高吞吐。那我们怎么看待它的所谓低延时、高吞吐呢？我是这么认为的：第一，不论是极端向左、向右走，最终的性能跟其他框架相比都是有一定优势的；第二，如果你的场景不需要非常极端走两边，可以直接站在中间，它也有比较综合的性能优势。大家可以看一下官网，有一些性能对比资料。

接下来就是端到端的“只说一次（exactly-once）”和高容错的状态管理，满足我们刚才所提到的健壮性要求。再接下来，对我们非常重要的是，基于事件时间和基于晚点数据处理的机制。因为对手机行业来说，手机上的埋点跟网页端埋点最大的区别是，数据的上报需要考虑用户的功耗和网络消耗。所以经常会出现用户的一次行为到这次数据真正的上报之间，有一个很随机的延迟，不能保证实时把数据报过来。所以基于事件时间去处理，包括处理晚点的数据，对我们来说，能够正确分析用户的行为，这很关键。最后，它还可以基于 Yarn。因为我们在过去几年的发展中都是基于 Yarn 去做集群管理，所以我们希望能延续过去的经验。

我们再来看 Flink SQL 提供的能力。

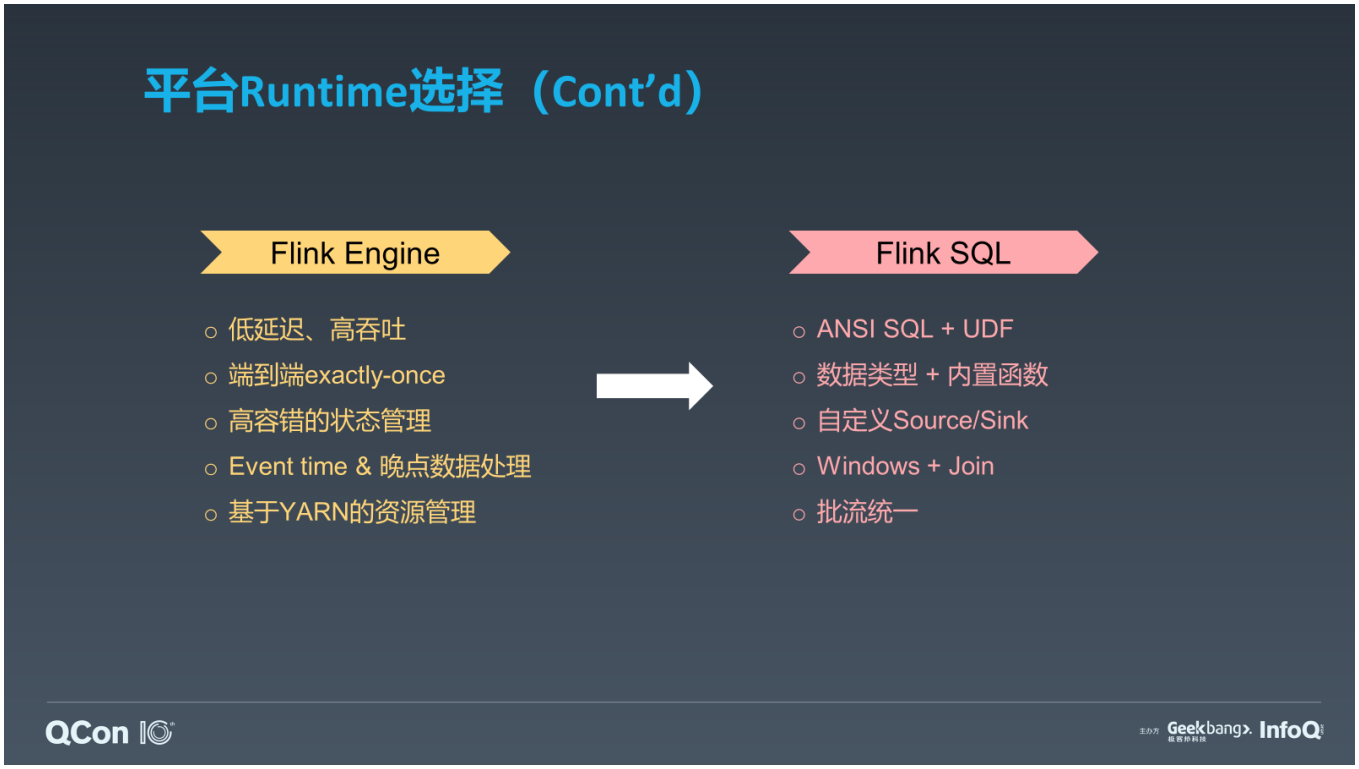

图 6 Flink SQL 提供的能力

前面两条可以不用说，它是支持 ANSI SQL 跟 UDF，包括数据类型、内置函数，这个在 Flink 官网有很详细的介绍。接下来，它可以自定义 Source/Sink，这就满足了刚才说的可扩展性，我们可以不

断扩充它的生态，如果它内置没有包含我们需要的上下游，可以自己做扩展。再接下来，就是之前其他讲师演讲里提到过的，像 windows、join，还有批流统一的能力，这里就不复述了。

我们选择了平台的 API 跟 Runtime 之后会思考：整个平台从离线、批处理到实时的流处理，能不能是平滑的迁移？什么是平滑的迁移呢？前面有提到过，我们希望 API 这一层尽可能抽象跟简化，能不能在 API 这一层经过转变以后，对用户来说尽可能降低学习成本跟迁移的成本？我们发现这是可行的。所以我们会看到，在离线的时代，API 这一层数仓抽象是 table，编程接口是 SQL+UDF；到了实时时代，我们其实可以保持一致，把更多的复杂性放在 Runtime 这一层，可以把计算框架、存储平台迁移到像 Flink、Kafka 这样的引擎上面，但是对于用户来说，API 这一层看到基本的抽象是一致的，这样学习的成本是比较低的。

图 7 离线到实时处理的平滑迁移

基于这么一个思路，最终推导出来整个数据 Pipeline 的实时模式，跟刚才我们说的离线时代几乎是一样的，只不过我们把一些关键性的组件替换了，比如说把 HDFS 替换成了 Kafka，把 Hive 替换成了 Flink，还有还有 OStream 平台，替换离线任务调度。然后其他的产品，包括基于报表、画像、接口这样的产品，都是保持一致的。

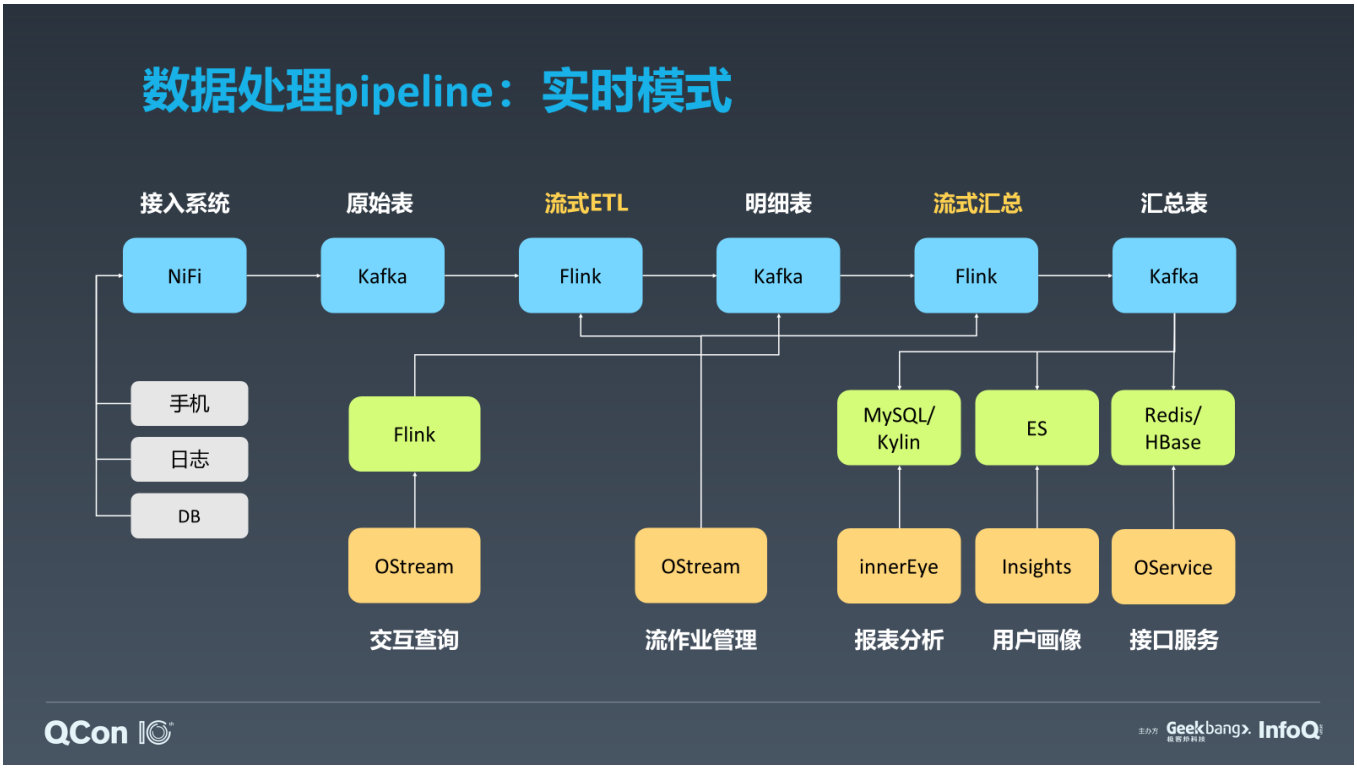

图 8 实时模式与离线模式保持一致

有了这么一个建设思路以后，如何开展整个平台研发工作呢？首先做平台，就要把整体的架构画出来。然后大致分这么几个层次：最下面一层是基础引擎层，这一层大家都很熟悉，无非就是基于 Kafka、Flink、ES 等这些系统，整个集群管理通过 Yarn 来管理，然后 Flink 需要做检查点，会用到 HDFS；平台特性这一层，提供网页版 IDE，会有像元数据管理、流作业管理、日志检索、监控告警等这些能力；再往上就是 API 这一层，编程接口支持 SQL，SQL 是我们的第一等公民，然后还可以支持 Jar 包提交的方式；再往上，我们把它称为“集成工具”。刚才提到，我们会有一些场景，希望用户不用写 SQL，可以通过简单 UI 配置的方式，就能自动生成 SQL。然后我们还会跟内部的 CI、CD 工具自动打通，自动编译好 Jar 包之后，自动把 Jar 包传到我们这个平台，这个正在研发当中。这个就是整体的架构。

图 9 OStream 平台总体架构

前面我提到说 SQL 是整个平台的第一等公民，所以第一步要解决基于 SQL 开发的这么一个框架。刚才提到 API 这一层是基于 Table SQL+UDF 的方式，具体落地到我们的产品，我们需要有一个类似这样的界面（如果大家比较熟悉，这是开源的 Hill 的界面）：左侧有一系列的 table，右边就是 SQL 的编辑框，可以提交 SQL。这个开发界面，具体落到 Runtime 这一层需要怎么支撑呢？核心有两点：一个是元数据的管理，就是如何创建库表，如何上传 UDF；还有一个就是流作业的管理，如何编辑 SQL，如何提交作业，最终提交给 Flink 框架执行。

带着这个问题，我们看一下 Flink SQL 现在 API 的情况。

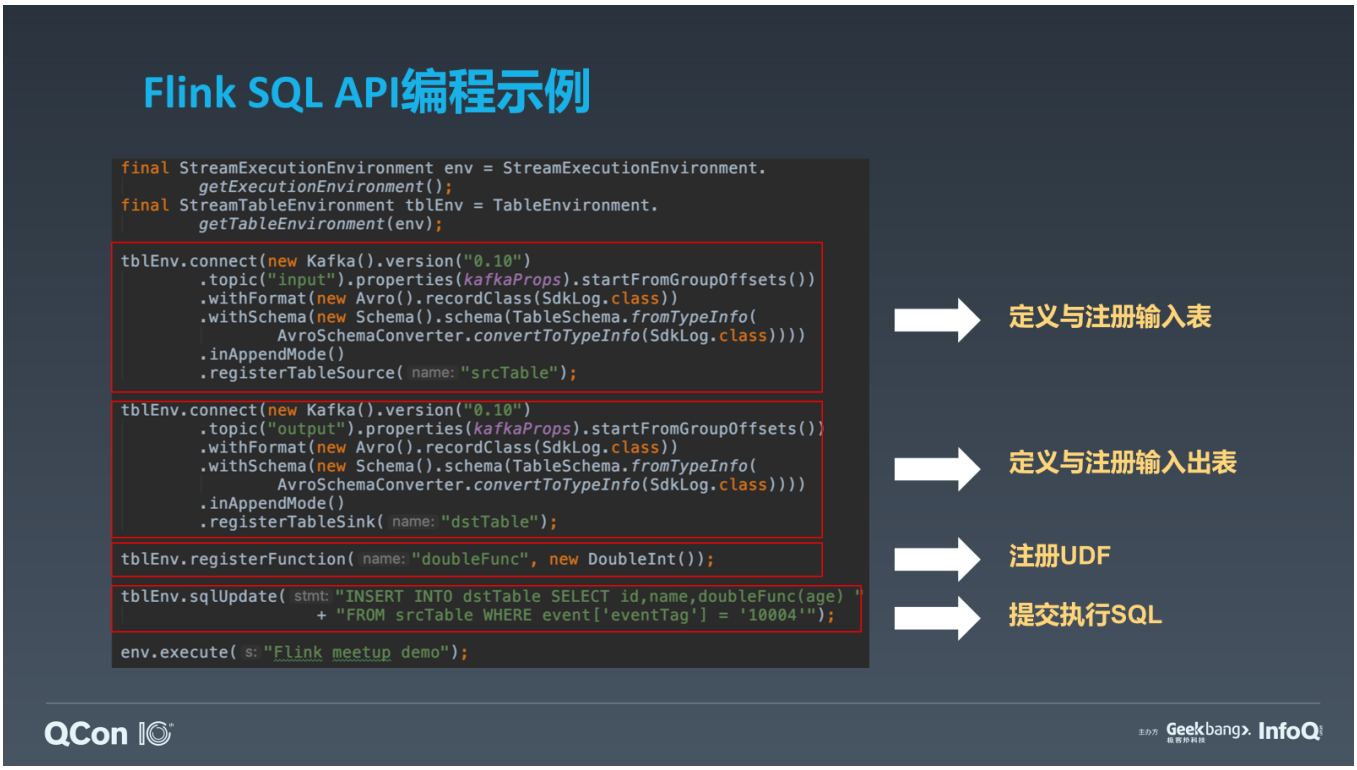

图 10 Flink SQL API 编程示例

这是最简单的 Flink SQL 编程示例，大概就是 20 行代码的样子。刚才提到流作业的管理，SQL 的编译、提交，包括元数据的管理、上传 table、注册 table，都可以通过这个方式创建。但是我们还是不希望这样面向用户。因为用户不希望用编程的方式去实现。所以我们基于 Flink 简单地做了扩展。

整个过程大概是这样子：首先在我们的开发 IDE 上面，用户可以写一个 SQL，写完以后可以提交，提交时创建类似于 job 的概念——我们会把 SQL 跟它所需要的资源、配置等封装起来，变成一个 job，然后保存到 MySQL。这个时候有一个 job store，定期扫描 MySQL，看看有哪些新的 job，然后调用 Flink 里的 TableEnvironment 模块，真正去编译这个 SQL，编译后会生成一个 JobGraph（这是 Flink 认可的一个可执行的单元），最终再向 Yarn 去提交这个任务。

刚才提到过元数据管理——table 创建出来以后，怎么被 Flink 识别？这里面涉及到编译过程中，通过元数据中心创建的 table，如何被它识别到？我们实现了一个框架，用到了 Flink 里面的 TableDescriptor 概念，创建了一个 TableDescriptor，即对表的描述符，也保存到 MySQL 里面。然后通过 Flink 的 ExternalCatalog（刚才云邪老师也有提到过），可以把 table 识别出来，最终通过 TableEnvironment 去注册。这样，Flink 就可以识别出外部的表，这个就是整个开发框架的实现。

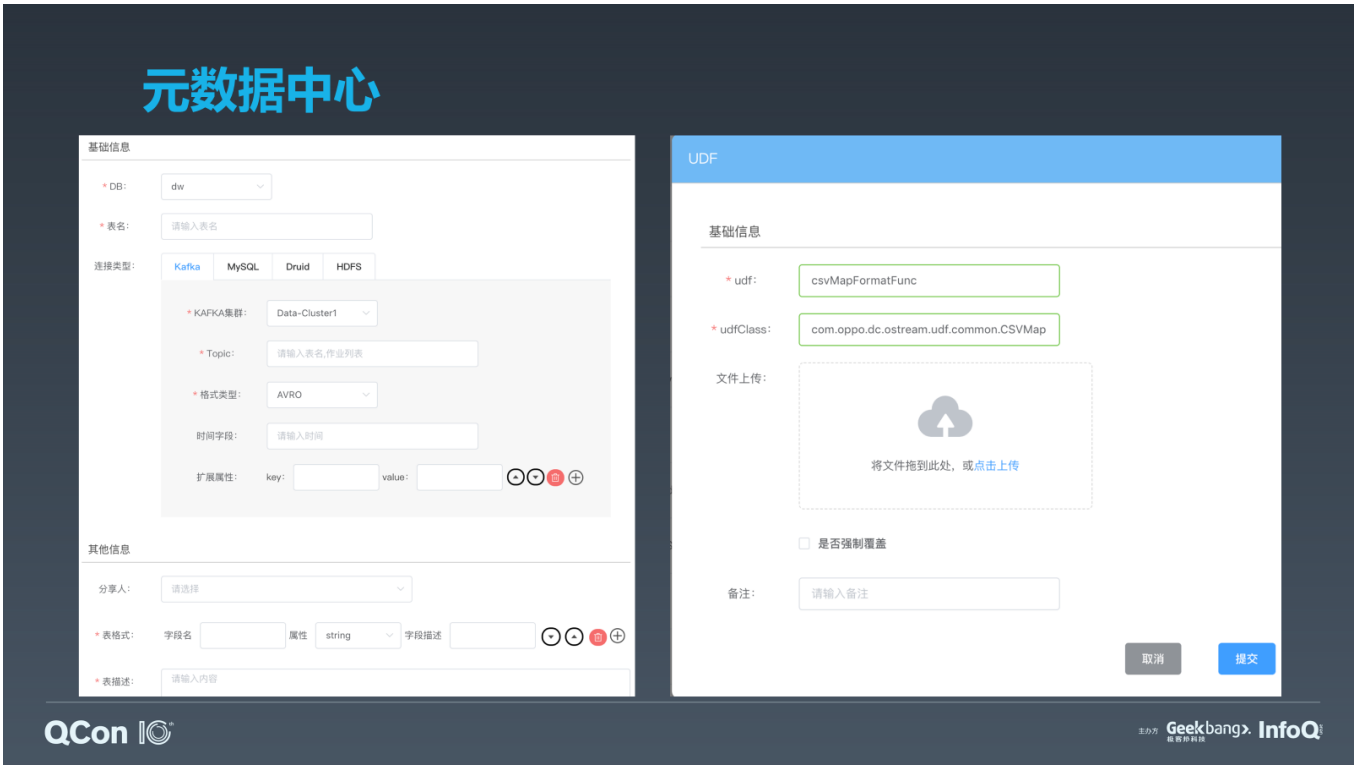

图 11 元数据中心的界面

左边是创建表的界面，我们可以创建像 Kafka、MySQL、Druid、HDFS 等的表。当然，我们其实不希望对用户暴露 DDL，因为对我们的用户来说，他们也不希望去写 create table，他们希望通过 UI 的方式创建表。右边是 UDF 的上传，可以自己写一个 UDF，也可以把一个 Jar 包提交过来，然后指定你的入口类（main class）是什么。

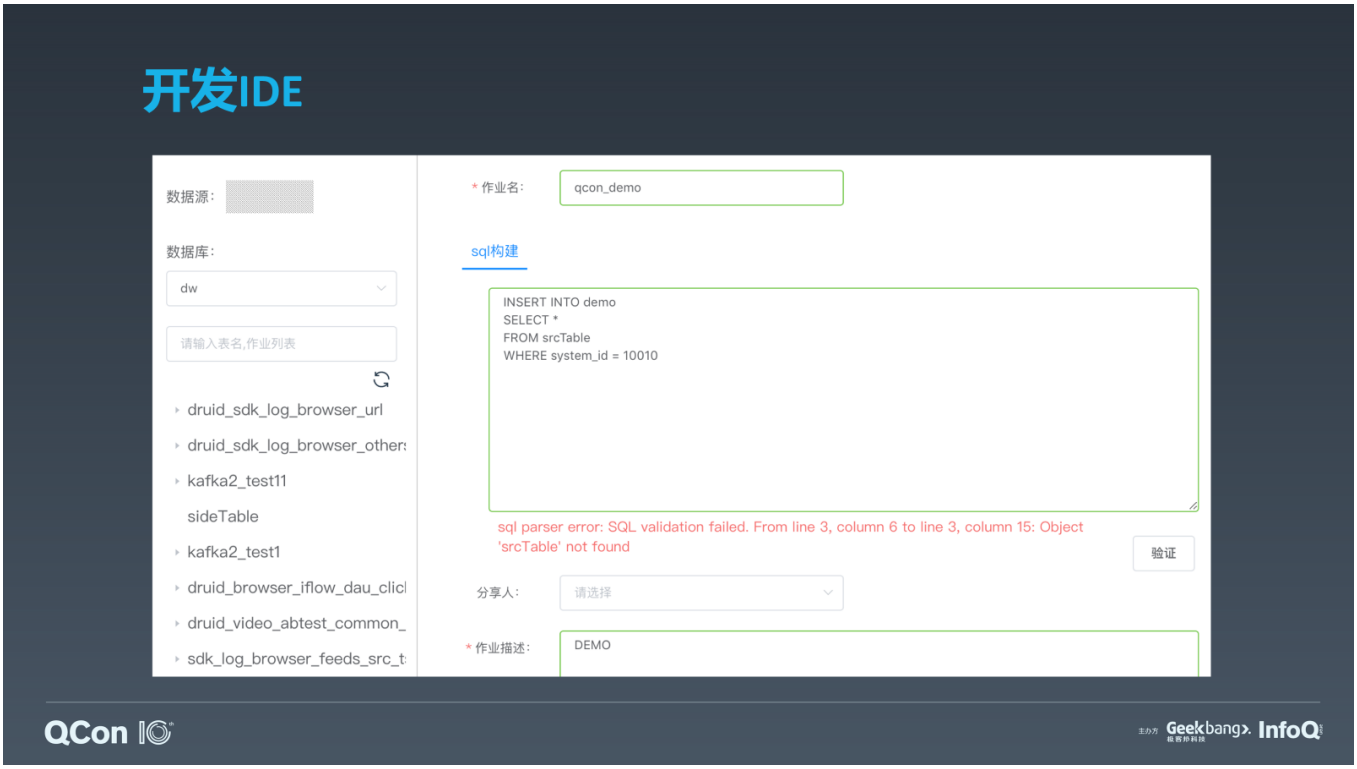

图 12 开发 IDE 界面

还有这个是我们的开发 IDE，也是很简单的界面，其实就是模仿 Hill 的开发界面。

有了这么一个开发 IDE 以后，还要解决什么问题？真的要把它推向用户使用的话，还有两个基本的问题要解决。

一个是 UDF，我们在 Hive 时代，积累了大量内部的 UDF，包括加解密、格式转换，可能会有点问题；还有就是位置、距离的计算等等，其实有大量 Hive 的 UDF。在 Flink 的处理上，大家也希望能够继承下来，直接使用它们。目前来说，Flink 还不支持这一点。云邪老师刚才说，到 1.9 可能支持直接调用 Hive 的 UDF，这个我们是很期待的，但目前来说还没有。我们作为平台方，需要把所有的 UDF 在 Flink 的框架上重新实现一遍。

第二就是维表的 join。有一些业务的维表存在 MySQL、HBase、Hive 里面，我们需要去实现这一点。这在 Flink1.9 也将会实现，但现在的版本还没有，我们怎么做呢？

图 13 维表创建页面

首先，这是我们的维表创建页面，可以创建一个 MySQL 的维表。由于大部分场景下，维表不算太大，可以直接把它 cache 到 Flink 的任务管理容器（task manage container）里面，所以我们这里可以选择 ALL 模式，即把所有的数据全量导入到容器。或者选择 LRU（least recently used，最近最少使用）模式，表示这是可以刷新的。还有一个模式（NONE）是干脆什么都不做，每次都去查外部的数据源。

这是维表的实现，我们现在如何 join 呢？

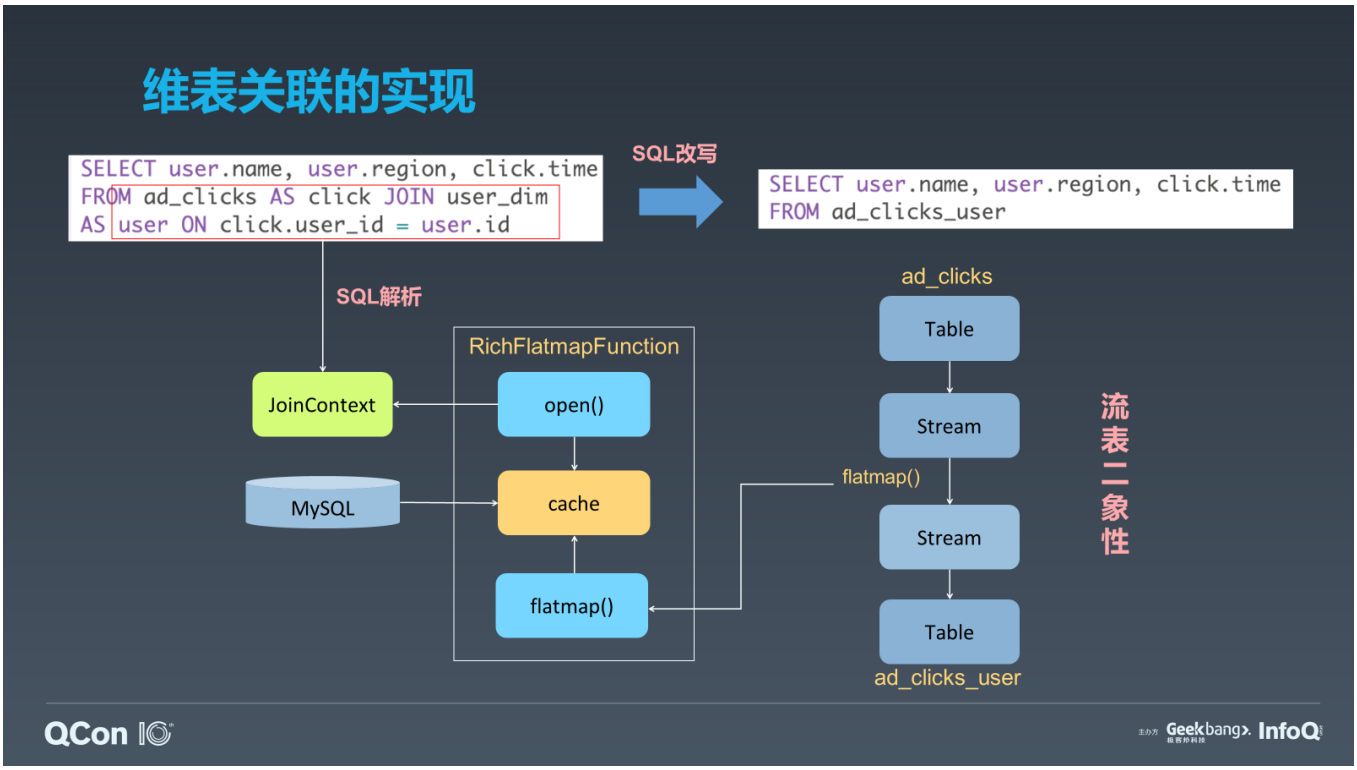

图 14 维表关联的实现

如上图，我们在平台层面会自动改写 SQL。假设用户写了一个 SQL，里面有一个 join 语法，我们在提交之前会加一层 SQL 的解析，解析出这是针对维表的 join（因为我们知道哪个表是维表），把解析出来的上下文封装成一个 JoinContext。有了 JoinContext 之后，我们会把 SQL 解析成更简单的 SQL。我们把 join 的语法替换掉，替换成类似于另外一张表的形式。

背后怎么做呢？实际用到了 Flink 里可以把 table 跟 stream 做无缝转换的功能：把原始的 table 转成 stream，然后在 stream 这一层再用 flatmap 的形式，每次去调用 flatmap 时，都会调用我们自定义的一个 flatmap 函数，这个函数会把刚才我们解析得到的 JoinContext 拿过来。这个 JoinContext 包含什么内容呢？包含维表到底是哪一张表、地址是哪些、关键字是什么。我们拿到这些信息之后，就可以去 MySQL 里面，在初始化阶段，open 阶段，把维表全量加载到 cache 里，然后每次做 flatmap 时，就可以从 cache 里查找，最终通过 stream 跟 table 之间的转换，再转换回一个 table，最终改写成这样的一个 SQL。这就是我们当前的实现，看起来会有点奇怪，所以我们也很期待 Flink 原生对 join 的支持。

接下来，这个平台少不了的，就是日志检索。这也是平台化时需要提供的很常规的功能。我们现在可以对日志提供基于作业名称的、基于 Yarn App ID 的、基于 container ID 的全文检索。我们怎么做的呢？

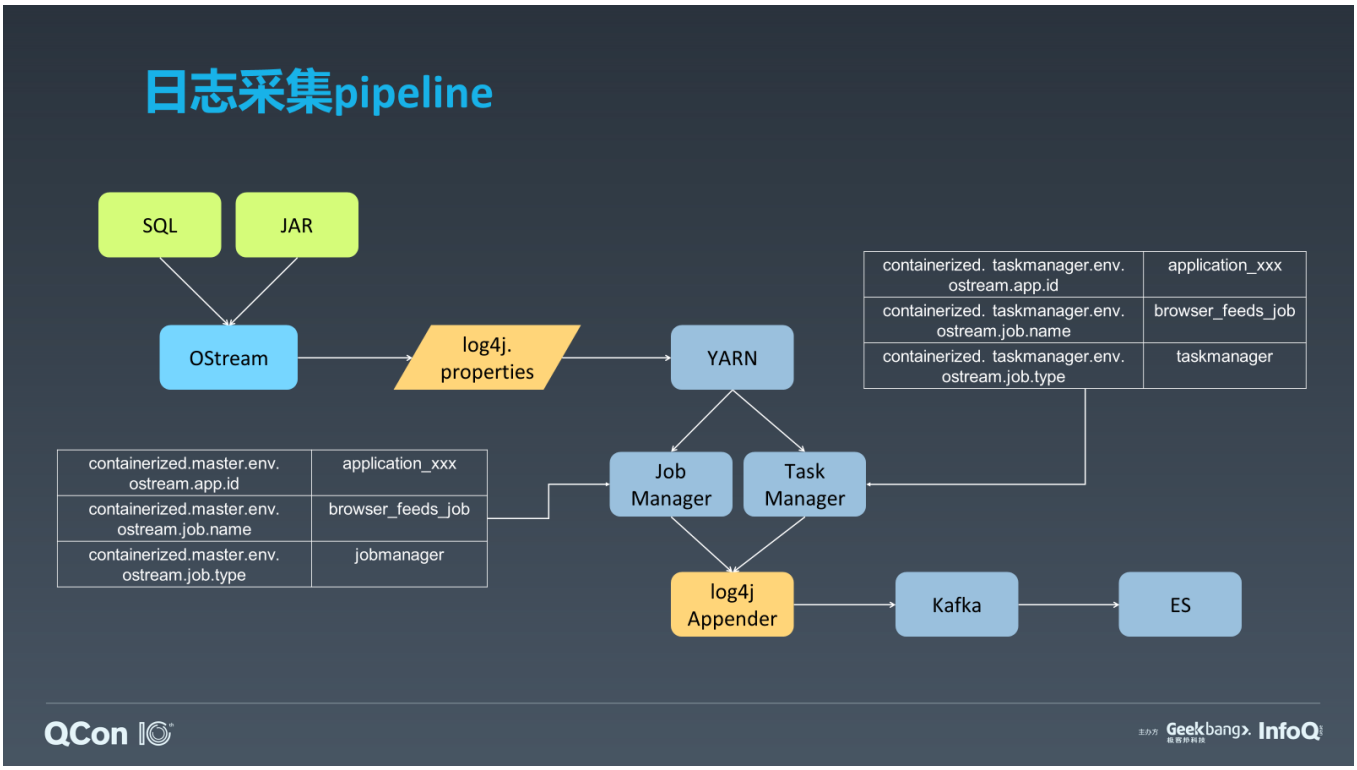

图 15 日志采集 Pipeline

其实也比较简单，无论是 SQL 还是 Jar 包，通过 OStream 提交 Flink 作业时，我们会自动生成这样一个 Log4j 的 properties 配置文件，提供给 Yarn 以后，最终 JobManager 跟 TaskManager 在执行时，就会用到我们自定义的 Log4j Appender，把我们的日志导向统一的 Kafka，最终再把所有的日志导入 ES 做索引，就做到了全文索引。还有一个问题需要解决：所有的日志怎么跟 OStream 的 job 关联起来？比如需要知道每一行日志到底是属于哪一个 job。这个比较好解决，我们在 Flink 里面有配置：在提交 Flink 作业时，可以给出 6 个环境变量，这 6 个环境变量分别注入给 TaskManager 跟 JobManager，我们的 Log4j Appender 会自动识别进程里有没有这样的环境变量，如果有，把它写到 ES 里，每一行日志跟作业名称、App ID 都可以关联起来。

接下来就是指标的监控，这也是平台里面需要提供的一个基本能力。有很多指标，但对于用户来说，最重要的两个指标是什么呢？一个是对 Flink 作业的吞吐，第二个是 Kafka 消费的 lag。这个我们怎么做呢？

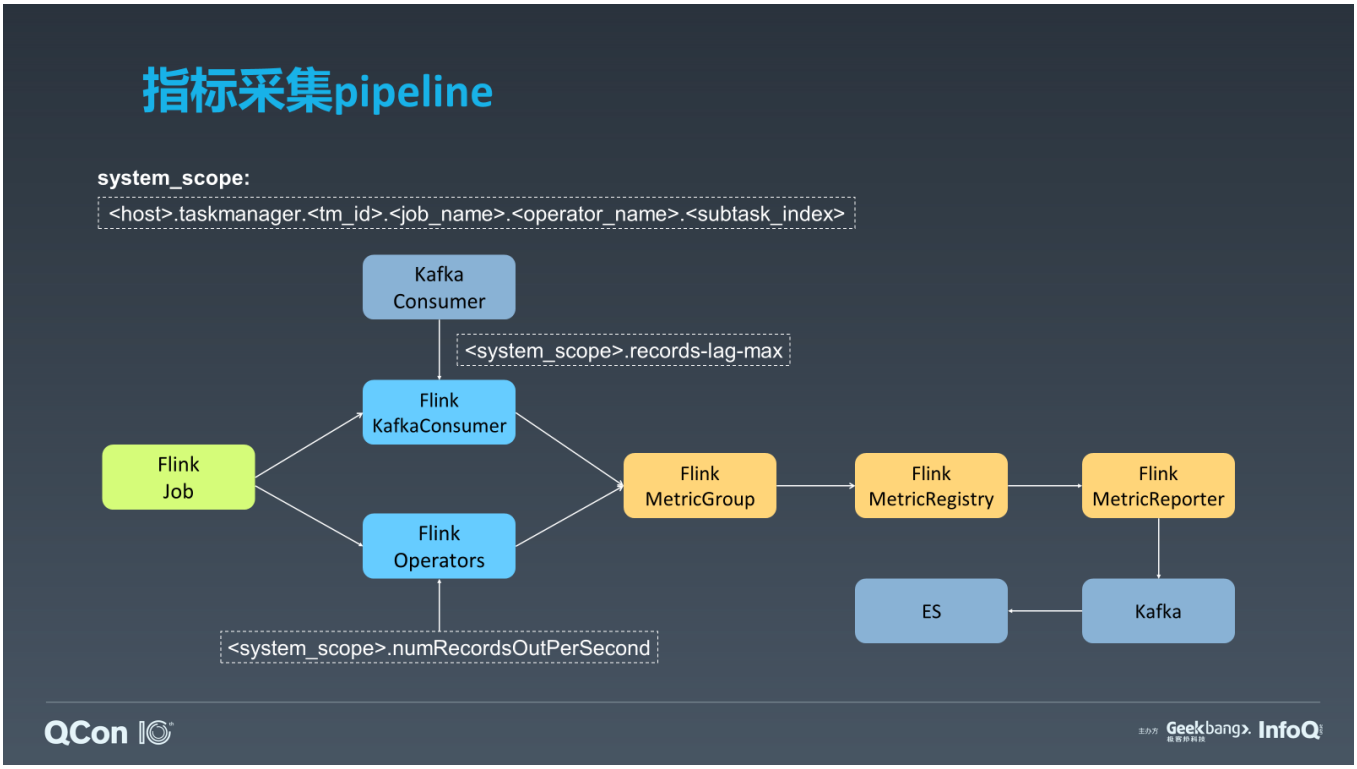

图 16 指标采集 Pipeline

首先，Flink 的 KafkaConsumer 是一个源，对于每一个 Flink job 来说，KafkaConsumer 会自动从 Kafka 的 Consumer 里继承它原生提供的很多指标，比如其中一个指标就是这个 Consumer 消费 Kafka 的 position、消费的延迟是什么。对于所有的 Flink Operators 来说，也会暴露出来这么一个指标：这个 Operator 每秒出去了多少条数据。基于这两种指标，我们可以通过 Flink 原生的内部 Matric 的系统 / 子系统，比如它内部封装的，像 MatricGroup、MatricRegistry，通过我们自己定制的一个 MatricReporter，可以把我们刚才提到的两个指标导入到 Kafka，最终也写到 ES 里去。然后基于这些指标做监控和告警。

图 17 告警规则

这是告警的页面，也是比较简单。基于这个指标可以做同、环比，或者是绝对值的告警。

上面大致介绍了基于 Flink 做的平台化的工作。这里做一个小结，是我们平台研发过程中积累的小小心得，简单分享一下。

第一个心得，对于我们这样的团队来说，最好的策略是维护纯净的分支。这个跟研发投入密不可分，最好的策略是紧跟社区的步伐，尽可能少改动 Flink 本身内核的东西。不然 Flink 整个社区发展很快，慢慢就追不上它的步伐了。怎么做呢？有两个策略。第一，任何的开源框架要打造它的生态，必然要去做很多扩展点，我们可以基于这些扩展点，做一些插件开发。比如刚才提到的元数据管理、如何把外部表对接到 Flink 里面去、如何做日志跟指标的采集，其实都是基于刚才提到的 Catalog、Log4j Appender、Matric Reporter 等等的扩展点。第二点是，如果没有扩展点可以开发插件，我们可以基于 API 去做二次抽象开发，不要改原生的代码。就像刚才提到的 SQL 的作业、维表的关联，其实都是基于原生的 TableEnvironment、TableDescriptor 这些 API 做的开发。

第二个心得，对于小团队来说，研发投入是有限的，大家如果想要去参与开源社区，怎么参加呢？我有些小的建议。首先我们要关注整个社区的动态。最好的方式有两个，第一个是可以关注一下 Flip。每次大的变动在 Flip 都有介绍，从 modification 到 design 再到 example，是一个完整的 story，可以了解这个模块整个发展路径。还有一个就是关注它的 pull request。这可能是最好的途径，因为每个想要往社区提交代码的，都要发一个 pull request，可以关注到每个模块，比如像 table、connect、runtime 这些模块；可以看到作者们在内部的对话，看到社区的进展。第二个可以看作建立一个认知——如果你没有参与过开源的项目，可能对 git branch 的机制、怎么 fork 一个分支、怎么 merge 代码、怎么提交 PR 都没有认知。可以通过一些小的方式，比如说，可以去帮忙修正一些 typo。你会发现 Flink 文档也好，注释也好，有很多语法错误，不管是语法拼写还是语法表达——这些就是 typo。当然，提交这种（对于编程）没什么意义，但如果只是想建立一些基本认知，可以去做一些尝试。最后就是所谓快速调试，什么意思呢？你想去了解 Flink 某一个模块，或者是某一个想法想快速验证一下，不可能说连接外部的 Yarn，然后再搭建一个 Kafka，然后再往 Kafka 里插入一条数据，这样太慢了。最好的方式就是把端到端的集成化测试本地化起来，在自己的电脑上，不需要依赖任何外部数据源，可以直接跑通。还有，自动化起来：整个端到端，比如说往 Kafka 插入一条数据、往 Yarn 提交作业，都可以自动化起来，可以更好地了解 Flink；甚至可以设一些断点，跟踪它的代码，了解 Flink 里面的内部机制。

这里给出了我自己的一个小工程，如果大家感兴趣可以了解一下。我利用很多像 Kafka mini cluster、Yarn mini cluster 的机制，在本地 IDE 可以把整个端到端的流程跑通，而且过程都是自动化的，不需要依赖任何外部的系统。

OStream 平台运营实践经验

介绍了研发工作，最后再介绍一下平台运营方面的实践。第一个实践就是，我们认为最好的方式还是拆分离线跟实时集群。我们最开始的时候，所有的 Flink 都跑在离线集群当中。但我们认为是有问题的，为什么呢？整个离线集群跑的作业都是短平快的作业，资源分配很不确定。

图 18 离线 - 实时计算集群拆分

如果大家对 Yarn 很熟悉的话，看到这张图应该会很亲切。我们可以看到，Yarn 是公平调度机制，每个队列之间，资源分配都是不确定的。不确定性表现在哪里呢？大家可以看到上面有绿色的部分和橙色的部分。虽然可以给每个队列分配资源，但其他的资源随着每个队列的情况在不断动态变化。举个例子，比如说它有一个 steady 的 fair share，这是固定分配的资源份额；还有 instantaneous 的 fair share，表示如果说有一个队列，里面没有任何作业，那就要分享出来给别人去用。还有一块就是队列可以透支资源：比如有一些队列即使里面有程序在跑，但没有消耗完这个份额，这个份额需要被调出来给其他队列分享，其他队列可以透支。那这对我们来说是有问题的，线上集群出现好几次这样的问题。假设说批处理跟实时处理是两个不同的队列，可能我们在最开始的时候，两个队列是 50%、50% 的资源分配，假设实时作业有一些问题导致它要重启，重启的过程中，它的资源会立刻被另外一个队列抢占了，因为另外一个队列可以透支这个资源。所以对于线上队列，我们会把抢占给关闭，这可能是另外一个话题了。如果把抢占开启，其实也有很多的不确定性。

对我们来说，实时队列资源被抢占，导致重启之后就没资源跑起来。所以我们认为，虽然大家都在谈离线跟在线的混步，但我们没有精力研究混步到底怎么做，所以最稳妥的方式如果要提供服务的保障，最稳妥的方还是式把这两个集群做物理拆分。

还有测试，是从我们原先的实时处理模式系统继承过来的。大家知道，我们做数据开发，很重要的一点是，它的测试跟普通程序可能不太一样。很多时候，做数据开发，测试数据的逻辑，需要的不是随便造几条数据就可以了，需要的是全量的数据、生产的数据，不是随便造一些数据。我们以前的方式是这样的：在发布测试作业之前，读写的是测试数据库，而测试库的数据就从生产数据采样而来。但对业务来说，他们觉得这是有问题的。所以平台可以这样做：既然测试时需要依赖生产库，平台可以在提交 SQL 时做 SQL 改写，比如自动把 insert into 改为指向一个测试的库，但是读还是直接从生产库读，可以得到及时的、全量的数据，而不会对生产库造成任何的影响。这是我们的实践经验。

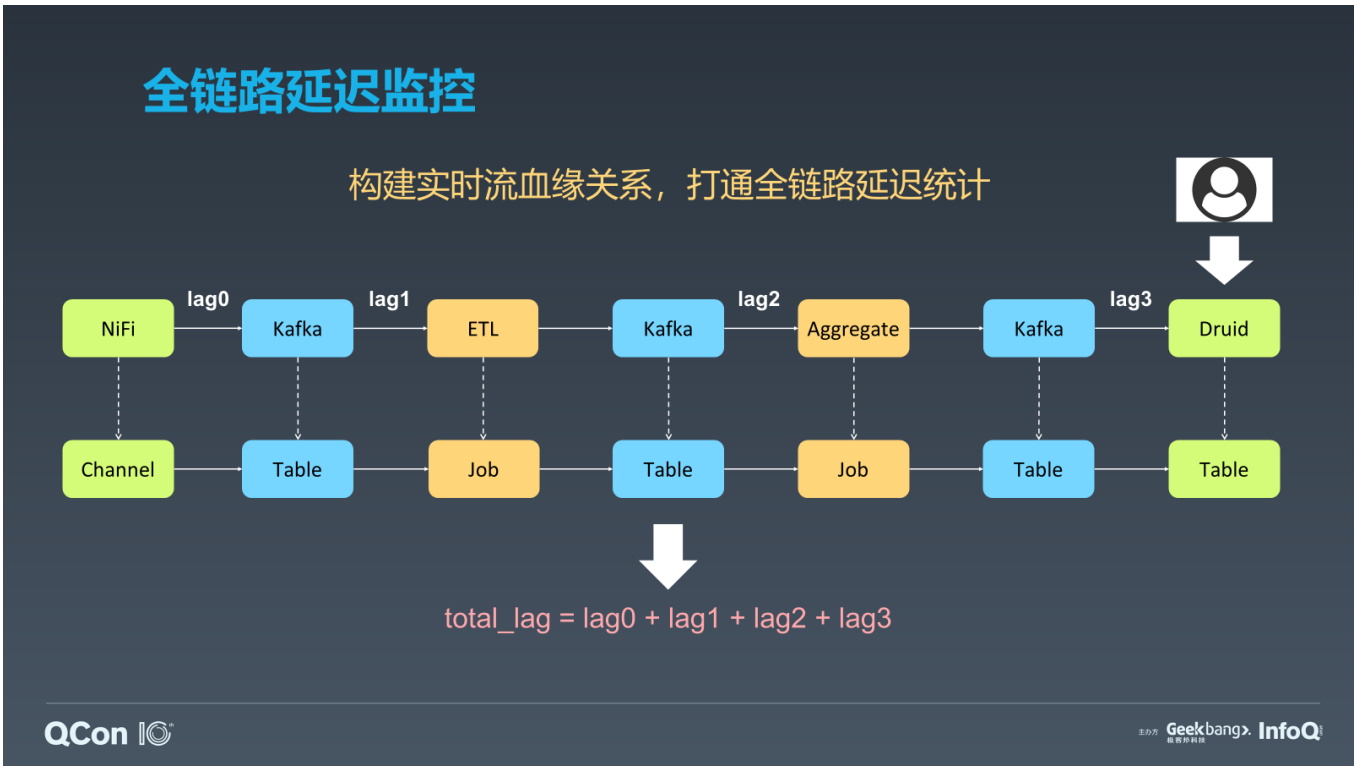

图 19 全链路延迟监控

最后一个实践就是全链路延迟监控。上面这个链路是基于 Flink 的整个实时处理链路。数据从最开始的 NiFi 开始，经过三次 Kafka，最终才真正到 Druid 里，然后才能从 Druid 里做报表。大家可以看到，在每个环节都有 lag，从 NiFi 到 Kafka，从 Kafka 到 Aggregate，再从 Kafka 到 Druid，有 3 次 lag。之前做延迟监控都是针对单个作业，但用户关心的是他们的数据到这个系统之后，什么时候才能在报表中体现出来，关心的是全链路的延迟，单个延迟对他来说没有什么意义。所以我们需要构建血缘关系。

解析 ETL、做 Aggregate，把整个血缘关系解析出来，形成下面这个通路，从接入通道，到中间的 table，再到中间的 job，再到最终的 Druid，这样我们可以把这 4 个 lag 加起来，变成一个统一的 lag，做成一个统一的监控。大家可能还会有一个疑惑：中间 ETL 到 Kafka、Aggregate 到 Kafka 有两处空白，其实也有 lag，为什么没有做监控？我们认为，Flink 有比较好的反压机制。如果 ETL 到 Kafka、Aggregate 到 Kafka 有延迟，Flink 通过反压机制，可以反映到上游的 Kafka 消费的 lag 里，所以我们可以忽略中间这两个 lag，只计算那 4 个 lag。

上面介绍了我们的研发和运营的工作，接下来分享一下案例。

图 20 交互式查询界面

这个界面是以前离线时代提供的交互式查询界面。到了实时数据时代，用户觉得这样的界面还是挺好的，还是需要通过 UI 的方式、拖拽的方式直接分析实时数据。当然，我们会做一些限制：比如说通过拖拽的方式，不可能去查全量的实时数据，可能只能查最新的数据，比如最近一个小时或者几分钟的数据。

下面是实时的 ETL，是我们线上应用最广泛的实时流处理应用。

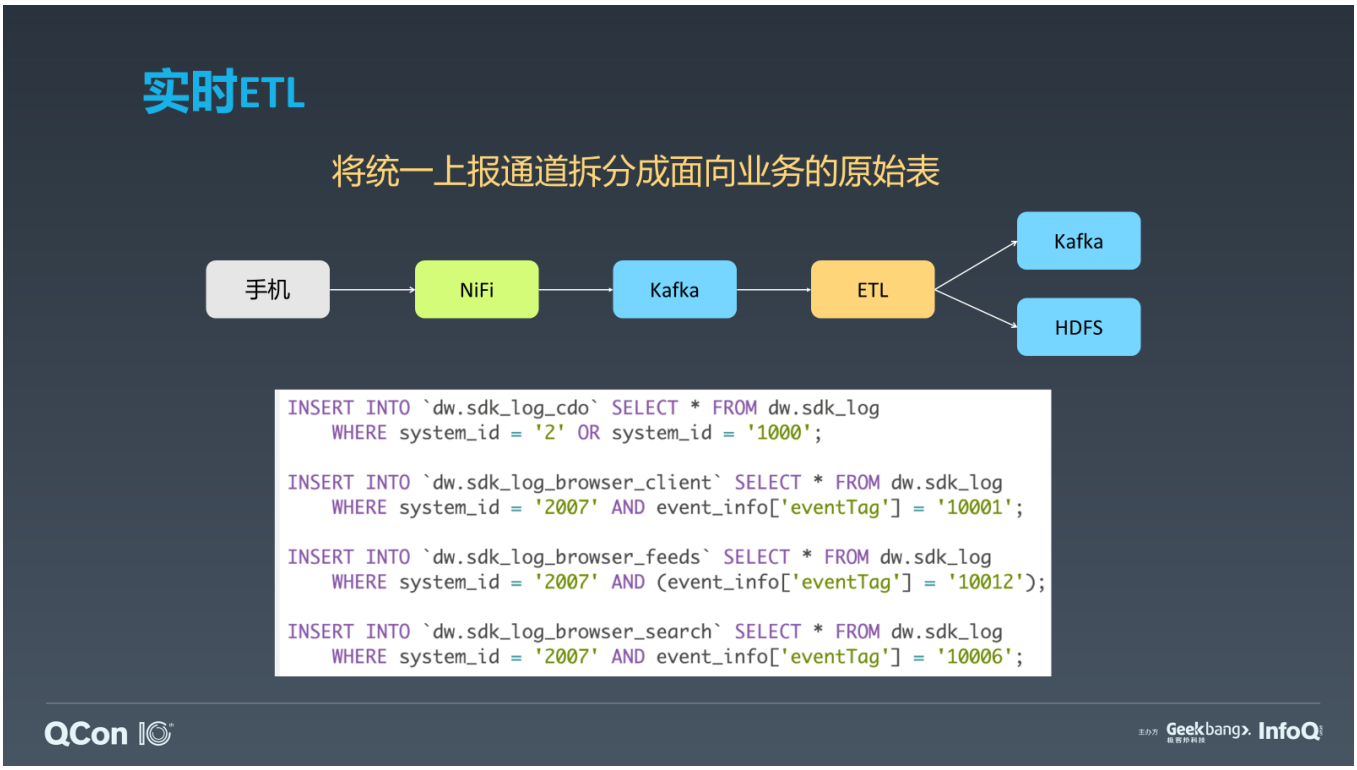

图 21 实时 ETL

所有从手机端过来的数据，都会报到同一个通道里。在录入到数据仓库之前，需要对这些数据做一个拆分，我们用的就是 Flink 的 SQL 方式，比如我们可以写这 4 个 SQL，对同一个 table 的数据，通过不同的条件，拆分到不同的下游表。我们用同样这一套处理逻辑，我们可以同时把数据插入到 Kafka 或者 HDFS，作为后面离线处理的源数据。

还有就是实时标签。标签对我们来说是最重要的数据资产，所以实时标签是很重要的应用。

图 22 实时标签

这个就是我们整个实时标签的通道，从 Kafka 里写一个类似于这样的 SQL，插入一个表，格式做一定的限制——因为这最终要对接到我们的标签系统。大家也可以看到，我们用到了嵌套 SQL，也用到 UDF 的函数（包括窗口）去实现。

未来展望和规划

最后做一些未来的展望：平台应该怎么走？

这可能说得比较大。我们认为**一个平台发展的必然路径，肯定是要从自动化走向智能化**，如果大家比较有信心的话，还会加一个词“智慧化”。大家对这三个名词的定义不太一样，尤其是“智能”，现在各种各样的地方都有智能这个词。

我这里给出我的定义。什么是自动化？我认为它是处理机械的、重复的事情，解放我们人的双手。最典型的自动化应用就是任务调度系统。每天到了一个点，重复性地、不需要人工干涉地自行调度任务，这是自动化。什么是智能化？一定是要做自适应跟自学习。因为不断自学习，所以它的行为应该是人没办法预料到的。可能最典型的应用就是一些 AI 系统，像阿尔法狗，是可以不断学习的。

最后一个“智慧化”，可能比较虚，现实中可能没人见过。以前有部美剧《西部世界》，里面那些 Hosts，就是有自我意识跟自我思维的，具备智慧化的。我们先抛开智慧化不谈，从自动化到智能化，听起来还是比较虚，具体落实到我们平台，我们能做什么事情呢?

比如说对自动化，我们可以做端到端的自动打通（我在后面再说）、通过规则自动生成 SQL、每次提交作业时自动生成告警规则，这些是一些自动化的工作。什么是智能化？我们其实也可以尝试，比如说作业的资源它能不能自动伸缩，因为线上负载一定是曲线的，不是直线。如果说作业资源是直线，难免会有资源的浪费或者不够用的情况，能不能做自动伸缩。还有，作业出现异常能不能自我修复；作业参数这么多，不可能说在提交作业时考虑完善，有一个完美的参数配置，那么能不能在作业运行过程中自动动态调优。

首先看一下端到端的打通。这是一个真实的案例。

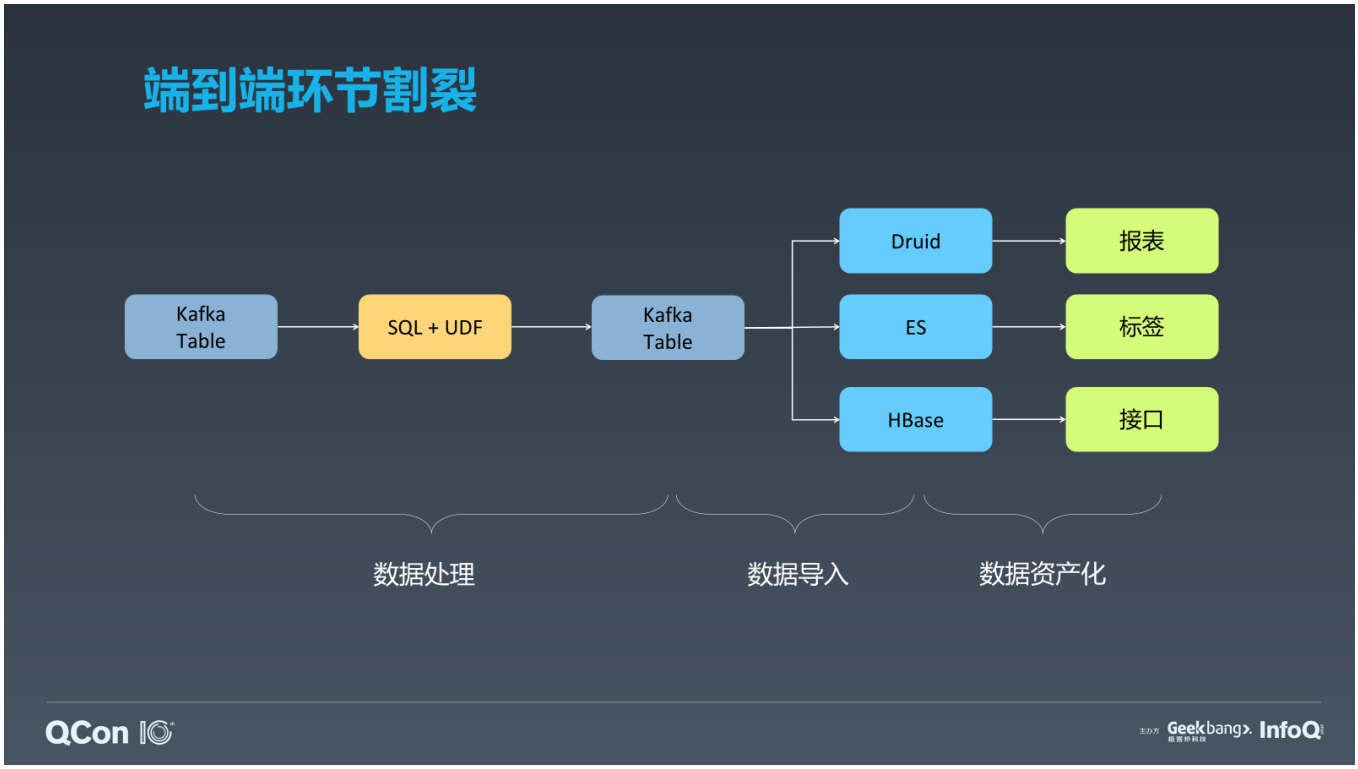

图 23 以前 SQL 的 Pipeline

这是我们以前 SQL 的 Pipeline。数据处理完了，从 Kafka 读、往 Kafka 写；然后通过数据导入的方式导入到存储引擎，这可能是另外一拨维护存储引擎的同学做的；最终把它变成线上资产，可能是数据产品人员做的。这 3 个环节是割裂的，而且需要不同的人去处理。我们考虑能不能把这个过程自动化起来，不要给用户暴露出来说这是个 Kafka table，而是就以展示表、标签表、接口表展现，这才是面向场景的。举例来说，对于展示表，无非有维度字段、指标字段、筛选字段、时间字段等。有这些字段以后，整个数据处理完，写完 SQL 以后，数据就可以自动加载到像 Druid 这样的平台里面去，最终也会自动变成一个报表，整个过程不需要用户再去操心。这就是线上正在做的例子。刚才也提到整个表的创建，未来表的创建是这样的：写了字段名称、描述、类型之后，可能还要判断，如果这是展示的表，则需要指定哪些是维度字段，哪些是指标字段。指定完成之后，当你写 SQL，比如 insert into，自动会跟下游像 Druid、像报表系统打通，甚至可以尝试自动把报表创建出来，对用户来说，这就是端到端的流程。

最后分享一篇论文，是关于刚才说到智能化的。这是 2017 年微软跟 Twitter 发的一篇论文（他们做了一个叫 Dhalion 的系统），很有意思，它的关键字是 self-regulating 的，翻译过来可能是“自我监管”，满足我们刚才所提到的智能化的方向。

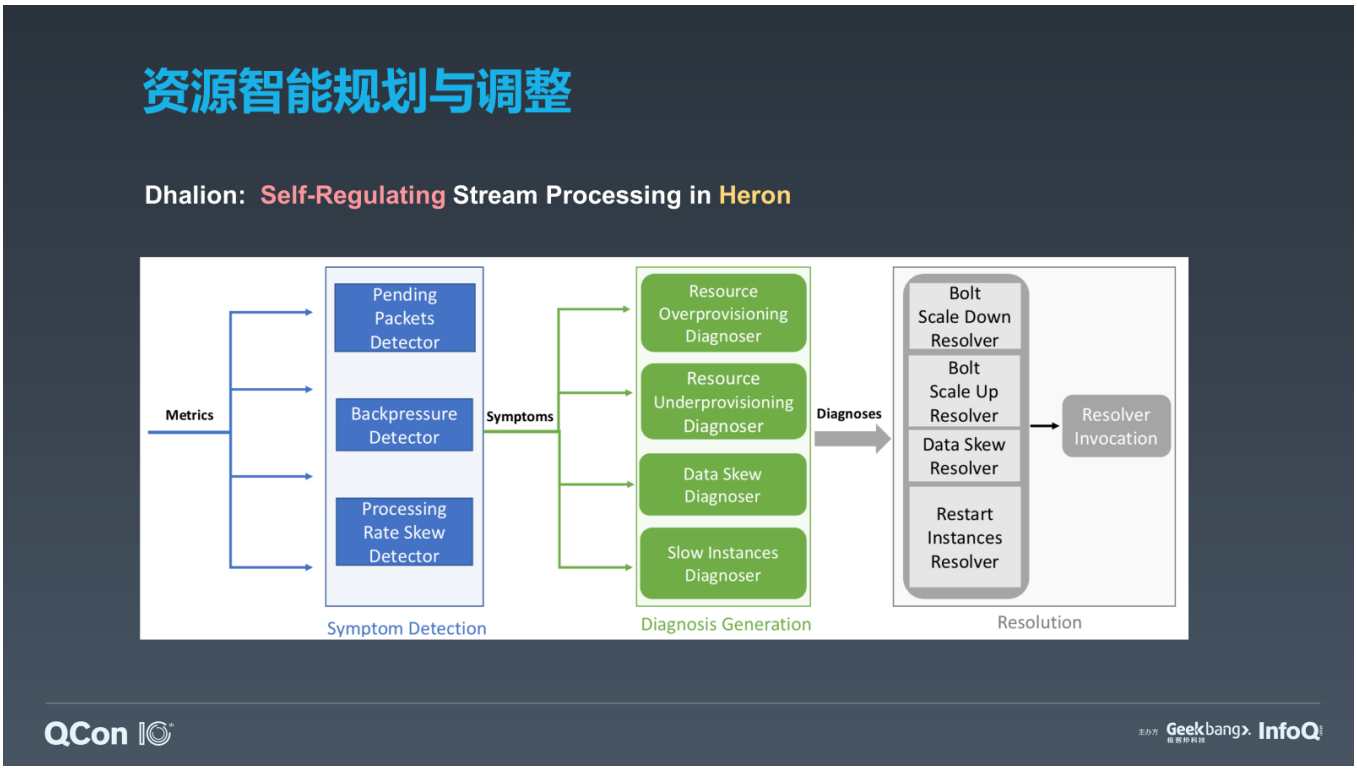

图 24 线上作业智能化自动伸缩

这是从论文里摘出来的一张图，说明线上作业如何实现智能化自动伸缩。它是这么做的：首先是有一些 Detector，像医生去诊断一样，去收集指标，根据指标判断是什么样的症状，比如说是产生反压了，还是线上的速率有倾斜了；获得症状后做一个诊断，通过不同的诊断器，判断到底是数据倾斜呢，还是有些实例因为磁盘变坏而变慢了；经过诊断以后，对症下药，如何解决这个问题。这可能也是自动的，有一些规则说怎么解决数据倾斜的问题、如何解决某个实例变慢的问题。

整个过程，包括中间的诊断，可以做成规则化，也可以做成基于机器学习，不断可以自适应的方式。这个框架基于 Twitter 之前做的 Heron 系统。我们能不能在 Flink 里面做这样的尝试？因为我们发现无论做指标收集也好，反压机制也好，到通过 API 对作业做自动伸缩，这些在 Flink 里都可以实现。我们能不能做这样一套框架，自动地、智能地在线上做资源的动态伸缩，这是挺有意思的一件事情。