360数据处理平台的架构演进及优化实践

360技术 2018-07-13

奇技指南

Titan2.0大数据处理平台是360大数据中心提供的数据集成平台，基于第三代计算引擎，提供批流合一的数据集成、数据同步，数据计算，数据分析等功能或服务，帮助用户和业务快速构建数据体系。本次要分享的是360大数据中心数据处理平台Titan的架构演进，以及一些具体的实践过程。
本文来自作者在DBAplus社群第153期的线上分享。

背景介绍

在当今的大数据时代，大数据计算引擎已经从原先最早的Hadoop生态系统演变到了第三代甚至是第四代计算引擎，比如Spark以及Flink等；存储引擎也是呈现多样化的发展，如支持MPP的关系型存储、分布式存储、时序数据库等。大数据生态的多样性给大数据开发的学习成本造成了很大程度的提高。

然而，当前我们的主流开发模式还是基于脚本开发，业务人员无法参与到我们的计算中来，这种开发模式在很大程度上依赖开发人员的效率，数据的时效性会难以保证。同时，数据开发过程中的数据流转全程黑盒，这给开发维护人员带来了很大的维护困难。最后，缺乏统一的资源调度导致资源长时间被占用不释放，从而引起资源浪费。以上是大数据开发人员所面临的问题和挑战。

从平台的角度来看，以360公司为例，公司产品形态多样化，包括了PC产品、WEB产品、移动端产品等等，多样化的产品意味着数据处理平台面临着繁杂的数据类型，同时也必然面临了多样化的存储引擎及存储格式。

此外，在大数据时代，数据处理的场景已经不再停留在简单的ETL过程，不仅仅包括简单的指标计算，还需要涵盖数据解析、数据分析、机器挖掘等等。随着数据对产品的指导作用越来越重要，业务对数据的时效、规则生效的时效以及需求响应的时效有了非常高的要求。

从大数据开发人员和平台开发的角度分析了当前面临的问题和挑战之后，接下来我们来看看要如何通过平台来解决这些问题。

平台演进

Titan大数据处理平台是360集团内部的一站式大数据处理应用平台，提供了数据集成、 数据同步、数据计算、数据分析以及流式数据处理等大数据处理应用场景的功能。

既然是演进，那就要从前世讲到今生了。我们的平台化进程基本可以分为三个阶段：

第一阶段：Titan前

这个阶段的架构图如下图1所示：

图1 Titan前架构

这个时期分布式计算兴起，从传统单机计算过渡到了分布式计算，在分布式计算引擎的基础上抽象了各类脚本模板，基于此，我们的工作模式从纯手工劳作转变到了脚本模板的开发。

开发模式的转变使得我们的计算效率和开发效率得到了很大程度的提升。但随着产品爆发式的增长、场景的增多，脚本模板无法提供灵活的方式，依然需要铺大量的人力解决。

第二阶段：Titan 1.0

这个阶段的架构图如下图2所示：

图2 Titan 1.0架构

在这个架构里，基于分布式引擎及计算场景，抽象了各类型的模板库，通过模板库引用到我们的上层交互，以系统的方式将数据开发一定程度上交给业务。同时可以看到，这个阶段里，数据源相对丰富了，产品也比较丰富了，有计算平台、报表平台、在线查询系统以及经营分析平台。

一方面把数据运维交给业务，另一方面也一定程度上把数据开发交给业务，让业务具有自主开发的能力。

但随着产品的发展，这个架构也很快暴露出了一些问题：一方面没有实时流的接入；另一方面模板库从一定程度上也是定制开发，依然无法满足个性化场景；任务运维开放的不够，在一定程度上依赖平台开发人员，运维成了瓶颈。

第三阶段：Titan 2.0

这个阶段的架构图如下图3所示：

图3 Titan 2.0架构

Titan 2.0采用了第三代计算引擎。

众所周知，第三代计算引擎主要特点是支持JOB内部DAG以及强调实时计算。在第三代计算引擎基础上，自主研发了DITTO组件框架和规则引擎，基于组件及计算的抽象提供丰富的组件库，采用图元拖拽的方式实现数据处理的无码化操作。同时通过调度监控以及权限管理，实现系统的自助运维以及数据安全的保障，值得一提的是，支持多种数据源接入，极大程度上满足各种数据类型及存储类型。

Tian 2.0功能模块

一级功能

从功能上，Tian 2.0划分了以下几个功能模块，一级功能视图如下图4所示：

图4 一级功能视图

二级功能

数据源管理实现了多数据源归一化处理，同时也提供了对数据安全和质量约束的保证。抽取和加载通过对各个存储引擎处理的抽象，实现了同时支持流式数据抽取和加载以及批数据抽取和加载。

任务管理部分通过图源配置，达到给用户提供灵活配置的需求，同时辅以我们运维管理模块的规则配置、任务调测以及数据流查看，实现了可视化运维和实例运维的效果。

调度引擎在提供即时调度、指定周期调度、历史任务调度的同时，还支持了实时调度和监控的功能。此外，权限管理不仅实现了角色权限、操作权限、菜单权限，还实现了数据源权限管理，为数据提供安全保障。如图5视图所示：

图5 二级功能视图

Titan 2.0从技术架构上来讲，采用的是第三代计算引擎，实现了跨引擎以及批处理和流处理的统一，这对上层用户是透明的；其次，它采用了DAG优化，提供了整个系统的处理能力以及计算效率，Titan 2.0与数据资产系统结合实现了全链路的数据质量监控以及数据的安全性保障。

业务架构上的优势可以总结为4个词：

- 多源：包含了对多应用的支持，以及对各类异构数据源存储系统柜的支持；
- 易用：系统的易用性突出在无码化上，数据计算，比如一个DAU计算只需要拖拖拽拽就可以了；
- 自助、灵活：自助化和灵活性体现在任务的配置修改、监控调度完全交给用户自己，不依赖于开发人员，想怎么玩就怎么玩，让用户真实的体验到玩转大数据。

实践案例

接下来分享一下平台的几个典型的实践：

首先是DITTO组件框架，DITTO组件框架的设计是为了解决多数据源、多存储源以及满足多种计算场景的需求。我们采用统一入口的方式，实现了对异构异源数据的支持以及离线计算与实时计算的统一，并采用组件组装DAG后计算满足各类场景需求。

下图6为DITTO组件框架的基本架构图，从架构图中可以看到，DITTO是搭建在Spark、Flink这类计算引擎之上的，上层可以支持离线计算、实时计算、机器学习、在线查询等图元化应用。

图6 组件框架架构图

DITTO采用了三层任务结构设计，如图7所示，分别是Application、Job、Task，它们之间的关系是一对多的关系，也就是一个App对应多个Job，一个Job对应多个Task，App在一次运行中负责了所有层次的初始化及所有事件的提交，DAG会在Job层次进行编排，最后Task这里是真正的执行单元。

图7 DITTO组件框架设计

在DITTO中，组件是最小的执行单元，DITTO的抽象大致可以分为组件计算逻辑抽象，组件计算引擎抽象和组件运行环境抽象这几个部分。

这里分享DITTO设计过程中如何对计算、组件、context和规则引擎进行抽象：

- 首先计算抽象。不论是数据处理、数据分析还是数据挖掘，最终都是一个数据的输入到一个数据的输出，那么首先就会有一个数据源的概念。而数据处理又分为业务的处理逻辑以及计算单元，这样就产生了业务逻辑的抽象和计算逻辑的抽象。当然，在计算的时候，要根据业务场景或者需求需要来选择不同的计算引擎，所以基于这四个方面计算可以进行动态的拼装，随意拼接去形成一次数据处理的计划，满足一定场景，这就是我们计算抽象的过程。

图8 计算抽象

- 其次是组件抽象，组件的执行需要做一个统一的入口或者说统一的标准，这个标准抽象为生命周期。标准定下来之后，数据如何进入到组件中，这里就会有一个数据采集。接下来数据在组件之间的流转需要定义一个元数据，元数据包括了数据类型和数据字段。对于数据依赖其实就是拓扑关系的维护。

图9 计算组件抽象

下图10是关于组件的类图，通过一个高层次的抽象来达到计算引擎的无关性，同时从这个类图中也可以看到我们的组件周期大致分为了prepare、execute、declare、clearup等几个阶段，数据的依赖通过dependencies来维系。

图10 组件的类图

- 而后是context，在DITTO中context主要负责初始化过程及上下文传输，在一个application的初始阶段，首先由context进行初始化。它负责了组件的初始化、计算引擎的初始化、时间的初始化还有DittoEnv和调度器的初始化。上下文传输部分的话，由于对于组件来说只关注自己本身，context来负责维系各个组件之间的数据流转。

图11 context

图12 context初始化流程图

- 最后是规则引擎，规则引擎其实比较好理解，它最重要的功能就是实现业务规则从应用程序代码中分离，简单来说就是实现了集成代码与业务无关，也就是将字符串或者是代码块传到我们的解释器后直接给出结果。在Titan 2.0里，基于规则引擎主要抽象出了四个部分，分别包括了逻辑运算、内置运算、四则运算、文本处理。

图13 规则引擎

在做DITTO初期，规则引擎是以一种DSL的方式接入到系统中的，随着场景的发展以及我们平台的扩展，规则引擎目前以独立的产品形态来提供服务。

图14为当前我们规则引擎的整体架构，从架构图中可以看到在提供常规规则处理的同时，我们规则引擎也提供了规则库和函数库的管理，方便我们业务以更加自主自助的方式来试用我们的规则引擎。

图14 规则引擎架构

另外在实际场景中，我们还从组件粒度上对性能和场景进行了优化，包括防止数据倾斜处理、防止内存溢出处理、数据缓存处理小文件合并等等。

心得体会

最后分享一下个人的一些心得体会。

我个人认为，在设计产品或者技术架构上，首先应该遵循化繁为简的原则，一定要避免过度设计，这个我们早期确实也踩过不少的坑；再者就是要一起从业务场景出发，做好需求分析，了解用户的核心痛点才能做到设计直击痛点、方便用户，因为平台的发展离不开业务的滋养，也正是业务场景的不断变化才带来了平台的不断进步；最后，平台的稳定是一切的基础，一切性能优化都需要找到那个平衡点。