UCLOUD优刻得

中国第一家公有云科创版上市公司

股票代码：688158

USDP 3.2
大数据平台产品白皮书

优刻得私有云

构建一站式大数据平台基础底座

赋能企业未来

文档更新：2024 年 02 月 20 日

版权信息

关于文档

优得刻科技股份有限公司在编写本文档时已尽最大努力保证其内容准确可靠，但优得刻科技股份有限公司不对本文本中的遗漏、不准确或错误导致的损失和损害承担责任。

由于产品版本升级或其它原因，本文档内容会不定期更新，除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目录

前言

UCloud（优刻得科技股份有限公司）是中立、安全的云计算服务平台，坚持中立，不涉足客户业务领域。公司自主研发 IaaS、PaaS、大数据流通平台、AI 服务平台等一系列云计算产品，并深入了解互联网、传统企业在不同场景下的业务需求，提供公有云、私有云、混合云、专有云在内的综合性行业解决方案。

依托公司在莫斯科、圣保罗、拉各斯、伦敦等全球部署的 32大高效节能绿色数据中心，以及国内北、上、广、深、杭等 11 地线下服务站，UCloud 已为全球上万家企业级客户提供云服务支持，间接服务终端用户数量达到数亿人。UCloud 深耕用户需求，秉持产品快速定制、贴身应需服务的理念，推出适合行业特性的产品与服务，业务已覆盖包含互联网、金融、新零售、制造、教育、政府等在内的诸多行业。

公司核心团队来自腾讯、阿里、百度、华为、VMware 等国内外知名互联网和 IT 企业，同时引进传统金融、医疗、零售、制造业等行业精英人才，目前员工总数超过 1000人。

在云计算之前，企业业务应用上线，需要经历组网规划、容量规划、设备选型、下单、付款、发货、运输、安装、部署、调试的整个完整过程。随着云计算、大数据、人工智能等新技术对各行各业的赋能，以虚拟化和软件定义技术方向构建新一代数据中心为基础，实现业务的集中管理、资源动态调配、业务快速部署及统一运营运维，满足企业关键应用向 x86 及国产化系统迁移时，对资源高性能、高可靠、安全性、高可用及易用性上的要求，同时提高基础架构的自动化管理水平及业务快速交付能力，继而推动企业的数字化转型与业务创新。

1 大数据技术体系

优刻得智能大数据平台构建了丰富且完整的大数据技术体系，通过运用一系列的技术方法实现了更可靠、高可用的平台环境。平台覆盖 Hadoop、Spark、Hive、Alluxio、Trino、HBase、Flink、Kafka、Hudi、Iceberg、Atlas、Ranger、等全栈技术和处理框架，满足各类大数据存储和计算分析的应用场景要求。

1.1 数据采集和预处理

数据采集是大数据技术架构建设的关键一环，它承担着从各种业务系统和运营平台中获取、转换和传输数据的重要任务。随着企业业务的不断扩展和复杂化，各个业务系统和运营环节会持续产生大量的业务数据，这些数据不仅数量庞大，而且涉及多种数据类型，包括结构化数据、非结构化数据等。这些数据来源不断拓宽，不仅包括传统的组织管理数据，如销售、采购、库存、员工等，还包括机器和传感器数据，如呼叫记录、智能仪表、工业设备传感器、设备日志等，以及社交数据，如用户行为记录、反馈数据、用户生成的内容数据等。

企业越来越重视数据资产的积累，数字化转型成为大势所趋。在这个过程中，数据采集的优化和提升显得尤为重要。为了有效地采集和处理这些不断增长的数据，优刻得智能大数据平台产品 USDP 提供了丰富多样的数据采集工具和数据预处理工具。这些工具可以帮助企业快速、准确地采集各种类型的数据，同时，

这些工具还可以对数据进行清洗、去重、转换等预处理操作，进一步提高数据的准确性和可用性。通过这些工具，为企业提供全面、高效、可靠的数据采集和处理解决方案。支持多种数据源的接入，包括结构化数据、非结构化数据、流数据等，并具备强大的数据处理能力。同时，USDP 还提供了丰富的数据预处理功能，如数据清洗、数据转换、数据聚合等，帮助企业将原始数据进行抽取、转换和加载，以满足后续的数据应用需求。

通过使用 USDP 提供的工具，企业可以更加便捷地将各个业务系统和运营平台中的数据进行采集和处理，为后续的数据应用类系统提供更加全面、准确、可靠的数据支持，可满足离线计算、实时计算、集成分发等多种需求，并进行全程状态监控。这不仅可以提高企业的决策效率和经营效益，还可以为企业的长期发展提供强有力的数据支撑。

根据不同业务场景对于数据时效性的不同要求，USDP 提供离线数据采集和实时数据采集两种数据采集类型。离线数据采集主要支持的数据类型为：MySQL、SQL Server、Oracle、MongoDB、HBase、ElasticSearch、离线文件；实时数据采集主要支持的数据类型为：MySQL、日志、HTTP API、JMQ 等，并支持 API 接口实现实时数据上报。

1.1.1 离线数据采集场景

企业离线数据类型和存储形态较多且复杂，常见的数据来源包括数据库系统、业务系统、文件文件存储系统、数据仓库等。

MySQL、Oracle 等关系型数据库是较为常见的结构化数据存储手段之一，以及非关系型数据库如 MongoDB 等。这些数据库当中存储了企业大量的业务数据和日志数据，通过离线数据采集可以将其抓取并存储起来，以便后续分析和决策。USDP 可随着离线数据采集场景的变化，提供符合企业数据采集需求的技术和工具。

企业自己的 ERP、CRM、SCM 等业务系统中，包含了大量的业务数据和交易信息，通过离线数据采集可以将其抓取并存储至大数据平台中，为规模化数据价值挖掘、业务分析和数据价值管理运用提供支撑。

块存储、文件存储等存储介质和系统中存储了大量的文本、图片、视频等半结构化、非结构化数据。离线数据采集可以通过读取文件系统中的文件，将其内容转换为结构化数据或半结构化数据，并存储在大数据平台或本地对象存储系统上，以便后续分析和处理。

部分企业，已逐步随发展需求，建设了适当的的数据仓库平台，这些平台未企业提供了存储和管理数据的能力，它可以对数据进行清洗、整合和汇总。离线数据采集可以将数据从各个系统和数据库中抓取出来，并将其存储在数据仓库中，以便进行数据分析和挖掘。

1.1.2 实时数据采集场景

随着科技的持续进步，企业不断扩大数据规模，同时对数据处理的时效性提出了更高要求。越来越多的业务场景依赖于实时的数据监测和数据分析反馈，以便快速获取数据表现并支持企业进行实时决策和管理控制策略调整。实时数据采集在帮助企业更好地把握市场动态、优化资源配置、提高运营效率、增强竞争力和提升客户满意度方面发挥着重要作用，成为企业在信息化时代取得成功的关键保障。

通常，企业的实时数据来源非常广泛，包括来自各种传感器的数据、实时消息队列、数据库数据、应用程序数据、网络数据以及社交媒体数据等。

- **传感器数据**

传感器可以采集各种类型的数据，例如温度、湿度、压力、速度等等。这些传感器数据通常来自于工业生产、物流运输、环境监测等各个领域。通过实时数据采集系统，企业可以及时获取这些数据，进而对生产过程、货物运输或者环境变化进行实时的监控和管理。

- **实时消息队列**

这类数据通常来自于企业使用的消息中间件，例如 Kafka、RabbitMQ 等。这些消息队列可以存储大量的实时数据，包括用户行为数据、交易数据、日志数据等等。通过实时数据采集系统，企业可以及时获取这些数据，进而进行实时的

数据处理和分析。

- **数据库**

各种类型的数据库，包括关系型数据库和非关系型数据库，都可以通过实时数据采集系统进行实时的监控和数据传输。这些数据库中的数据包括交易数据、库存数据、客户信息等等，对于企业来说具有重要的价值。

- **应用程序数据**

现代企业中使用的各种应用程序可以产生大量的数据，包括日志数据、事件数据、用户行为数据等等。这些数据可以通过实时数据采集系统进行采集和处理，进而进行实时的监控和分析。

- **网络数据**

网络数据包括网络流量、网络日志、网络监控数据等等。这些数据可以通过实时数据采集系统进行采集和处理，进而进行实时的网络监控和管理。

- **社交数据**

社交媒体平台可以产生大量的用户生成内容（UGC），例如聊天记录、话题讨论、点赞、贴吧、评论、用户内容创作等等。这些数据可以通过实时数据采集系统进行采集和处理，进而进行实时的市场洞察和竞争分析。

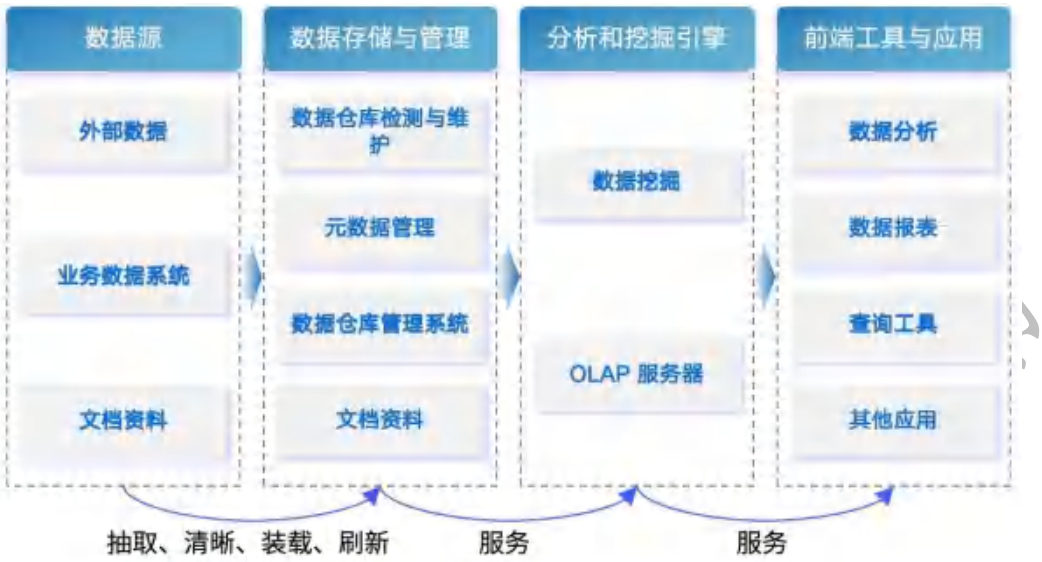
这些实时数据对于企业来说具有重要的价值，可以帮助企业更好地了解市场情况、掌握客户需求、优化资源配置、提高运营效率以及增强市场竞争力等。因此，企业应该重视实时数据的采集和处理，并利用实时数据来辅助决策和控制策略的调整。

1.2 数据仓库

1.2.1 数据仓库概述

数据仓库是决策支持系统 (DSS) 和实时分析应用数据源的结构化数据环境，主要研究和解决从数据库中获取信息的问题。数据仓库是在企业数据库已经大量

部署的前提下，为了进一步挖掘数据价值、支持企业商业决策而产生的，其核心特征是面向主题、集成性、稳定性和时变性。



数据仓库是存放数据的仓库，集成了各个业务系统的数据，如金融业的贷款业务、CRM、存款业务等。它用于企业的数据分析、报告和决策制定，同时也是各业务系统的数据来源。从逻辑上看，数据库和数据仓库没有区别，都是通过数据库软件存放数据的地方，但数据仓库的数据量更大。主要区别在于传统的事务型数据库（如 MySQL）用于在线事务处理（OLTP），如交易事件的发生，而数据仓库主要用于在线分析处理（OLAP），如出报表。虽然初创时期业务量小，用户和数据量少，可以通过业务数据库完成数据分析、出报表等工作，但随着业务增多，用户和数据量庞大，需要跨集群关联多个系统的数据进行报表生成时，数据仓库就变得很有必要。

1.2.2 数据仓库特点

数据仓库并不是所谓的“大型数据库”。数据仓库的方案建设的目的，是为前端查询和分析作为基础，由于有较大的冗余，所以需要的存储也较大。为了更好地为前端应用服务，数据仓库往往有如下几点特点：

- **高效性**：数据仓库必须具有足够高的效率，以便在短时间内处理大量数据。对于日周期的数据分析，客户需要在 24 小时甚至 12 小时内看到昨天的数据分析结果。如果数据仓库的设计不够好，可能会导致分析结果

延迟数日，从而无法满足客户需求。

- **数据质量**：数据仓库应提供准确的数据，但由于数据仓库流程包括多个步骤，如数据清洗、装载、查询和展现等，其中复杂的架构可能会影响数据质量。数据源中的脏数据或代码不严谨都可能导致数据失真，从而为客户带来错误的分析结果和决策，造成损失。

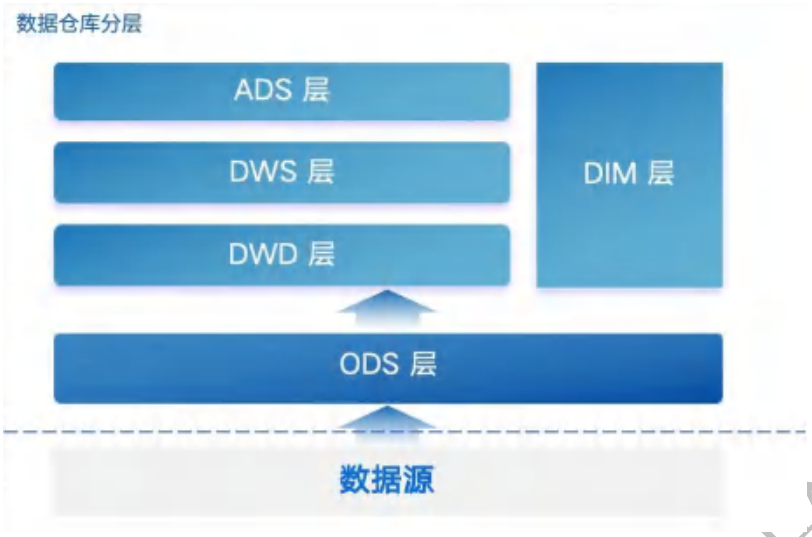
- **扩展性**：为了适应未来的发展，大型数据仓库系统需要具备扩展性。合理的数据建模和中间层的设置可以确保数据仓库在未来能够稳定运行，而不会因为数据量的大幅增加而出现问题。

- **面向主题**：数据仓库中的数据是按照主题域组织的。主题是宏观分析领域的一个抽象概念，是将企业信息系统中的数据综合、归类并进行分析利用的视图。每个主题对应一个特定的分析领域，数据仓库会排除对决策无用的数据，并提供特定主题的简明视图。

1.2.3 数据分层

数据仓库分层是为了更好地组织和管理数据，以提高性能、提升数据处理效率、降低数据冗余、满足业务开发的灵活性和提高数据可维护性，通常情况下，可以根据数据来源、数据主题、数据处理方式等进行划分。

预先分析数仓建设的需求，明确业务需求逻辑和用户需求，了解需要分析的数据类型和频率。通过业务过程分析，深入理解组织的业务流程，识别关键业务指标和数据元素。在数据源分析中，确定操作型数据的来源，包括数据库、日志文件、API 等，并了解数据的结构和变化规律。而后具体设计数据仓库的分层逻辑，以最符合业务需要的分层逻辑为最优选择。



- 操作型数据层（ODS）

ODS 主要任务是保持源数据的原始态，这意味着 ODS 层存储的数据与源系统中的数据保持一致，不进行重大的清洗、转换或汇总。ODS 层的设计旨在提供一个可靠的数据存储，以便快速响应实时业务需求。这一层充当了数据的缓冲区，确保源数据在进入数据仓库后的最初阶段得到保留。

- 维度表层（DIM）

DIM 主要用于存放维度数据，例如客户维度、日期维度等。维度表是数据仓库中的关键组成部分，用于提供上下文和关联信息，使业务用户能够按照不同的维度进行分析。这一层的设计关注于提供一个一致性、易于理解的维度视图，为数据仓库中的事实表提供关联。

- 明细数据层（DWD）

DWD 扮演着存储原始、详细数据的角色。这一层的主要任务是存放大宽表，即包含了多个维度和度量的详细数据表。DWD 层通常采用事实表和维度表的关系模型，以支持复杂的查询和分析需求。这一层的设计关注于提供高度灵活性，确保用户能够以最精细的层次深入数据，进行深度分析。

- 数据汇总层（DWS）

DWS 负责对数据进行轻度汇总，按照一定的维度进行聚合。这一层的设计旨在提供更高层次的数据概览，使用户能够更快速地获取业务洞察。DWS 层的

数据通常已经被聚合为更高层次的度量，以支持决策和业务监控。这一层的存在旨在提高数据查询的性能和效率。

- 应用数据层（ADS）

ADS 主要存放一些个性化数据指标和应用相关的数据。这一层的设计关注于为特定应用程序或用户提供定制的数据视图。ADS 层存储的数据可能包括一些特殊计算的结果、定制的报表和与应用程序密切相关的数据。这为用户提供了更个性化、定制化的数据访问体验。

如上图所示，在整个数据仓库的架构中，这些层次相互配合，形成了一个有机的整体，能够支持各种业务需求和查询场景。通过合理设计和管理这些层次，数据仓库能够提供高性能、灵活性和可维护性，为组织的决策制定和业务智能提供可靠的基础。

数据仓库是构建商业智能的基础，其主要功能是基于数据仓库的数据存储架构，将企业信息系统中联机事务处理产生的时序海量数据进行系统整理和分析，为后续数据挖掘、数据联机分析处理等环节提供支撑，从而帮助企业依托海量数据完成价值提取，以更好地应对外部环境变化并做出正确决策。

1.3 数据湖

数据湖是一种大型数据存储和处理引擎，支持存储结构化数据、半结构化券据和非结构化数据等不同类型数据，同时可实现不同类型海量数据的并行存取处理、分析和传输。

数据湖最核心的特征就是可提供一个容纳所有形式数据的存储集,将不同头型、不同领域的原始数据进行统一的存储,这也是区别于数据仓库的关键。此外,数据湖还具备 PB 级数据存储规模和运算能力，可以支撑多源数据的并行提取交叉分析，并能提供大容量、高速度的数据传输通道。

为了满足日益增长的数据处理和分析需求，以及对数据分析时效性、存储管理的要求，USDP 引入了 Hudi、Iceberg 和 Canal 等组件，进一步丰富了 USDP 的大数据服务生态，为用户提供全面的数据管理解决方案。

Hudi 和 Iceberg 提供了强大的数据管理和存储能力，支持增量数据更新、事务性写入和元数据管理，使数据操作更高效、更可靠。这些组件可以帮助用户构建湖仓一体的数据湖架构，实现数据的统一存储和管理，同时提供灵活、可靠的数据操作和查询。

而 Canal 作为一种流式数据采集工具，能够实时捕获数据库的变更，并将数据发送到目标系统，实现实时的数据采集和传输。

通过引入这些组件，用户可以在现有的大数据集群基础上，构建更加完整和可靠的数据湖架构。无论是数据存储、数据管理，还是数据采集和传输，都能够满足企业的大数据处理和分析需求。

1.4 湖仓一体

湖仓一体是一种新型开放式架构，打通了数据仓库和数据湖，并融合了两种架构的优势，底层支持多种数据类型并存，且实现数据间的相互共享，上层可以通过统一封装的接口进行访问，可同时支持实时查询和分析。参考现有国内外商的产品能力，通常具备:事务一致性、多模数据类型、实时并发、流式数据处理等特性。湖仓一体架构将数据仓库的高性能与管理能力与数据湖的灵活性相互融合，同时也为企业进行数据治理带来更多的便利性。

湖仓一体使得数据入湖后可原地进行数据处理加工，避免数据多份冗余及流动导致的算力、网络及成本开销，可以作为超大型 ODS 存储贴源数据，实现全量数据的实时处理。

2 优刻得智能大数据平台——USDP

2.1 产品概述

智能大数据平台 USDP（UCloud Smart Data Platform）是 UCloud 自主研发的智能化、轻量级的大数据基础服务平台，提供一站式大数据集群管理和运维能力，能够帮用户快速构建起大数据的分析处理能力。凭借其完备得管理功能、丰富的管理工具集、细致的监控体系，可大大简化对大数据基础平台原生且复杂的维护控制工作，提高集群管理效率，提升系统稳定性，降低管理成本，使用户聚焦于数据分析及价值挖掘中，达到降本增效的目的。USDP 多集群部署和集中管理的多个集群能力，可方便快速的管理成百乃至上千台服务器资源在各集群中的使用。该应用程序使安装过程自动化，从而将大数据集群部署时间从数天缩短至数分钟；为您提供集群范围内实时运行的主机和服务的视图；提供单个中央控制台，以在整个群集中实施服务配置管理；结合监控指标采集扩展，将集群及服务状态以及性能表现尽收眼底，为优化集群性能和资源利用率提供强有力的支撑。

2.2 产品架构

2.2.1 USDP 产品架构

USDP 的 Server/Agent 架构设计，以及主服务 HA 特点，使得 USDP 对超大规模大数据平台的管理控制得心应手，Server 为用户提供控制交互，并统管集群等所有服务；Agent 为节点控制端服务，用于轻量化管理、操作所有节点以及节点上的运行大数据服务，智能化的执行管理监控指令、服务指令、修复命令等。

提供丰富的集群管理功能、服务的控制操作等。支持多集群创建与管理，集群监控与自定义监控图表等功能；支持服务操作、配置管理、便捷控制；用户通过 USDP 可轻松完成对平台的管理和维护。

支持集群级监控、服务级监控、以及自定义监控扩展。全面丰富的监控粒度，辅助用户清晰掌握集群及大数据分析业务状况，轻松便捷的完成对整个大数据分析平台业务的管理控制，通过完善的告警通知方式、事件管理机制、活跃告警、日志搜索等功能，轻松定位异常。

USDP 提供了丰富的工具集，如自动化修复工具、异常修复程序，以及异常修复建议等，大大简化了平台/集群的复杂度，降低维护难度，保障平台持续健康稳定的提供服务。

USDP 支持可视化的集群级别的 Kerberos 认证、统一用户配置及管理能力、并结合资源设施层灵活的网络策略，有力的保障海量数据及数据分析及数据服务安全可靠的提供。

USDP 具备可持续迭代升级的能力，用户可通过该功能，对已使用的 USDP 平台及用于生产中的大数据集群，进行升级，获得管理功能及工具的扩展和大数据生态的升级更新，使得业务需求得到技术生态的新功能新特性的支持和服务优化支持，甚至技术架构改进。

2.2.2 大数据平台架构

USDP 扩展支持了数十款大数据生态服务组件/工具，涵盖了大数据业务处理架构所需的各类成熟稳定的服务软件，如数据采集设计的工具、数据仓库服务、批/流处理引擎、消息队列、分布式文件存储系统、分布式可扩展数据库、即系查询引擎、可视化 SQL 级开发工具、数据科学分析工具、MPP 数仓引擎等。

USDP 架构设计中充分考虑了集群中诸多大数据服务的高可用、高可靠性，避免服务的单点架构带来的故障发生，满足企业级大数据平台服务建设要求，使得数据分析业务获得服务的持续、稳定运行，以及优异的性能表现。

基于 USDP 中丰富的大数据生态支持，企业用户可依据建设要求，灵活选择所需的服务及工具，构建适合自身大数据业务分析平台。在使用 USDP 过程中，可通过 USDP 不断的扩展瑜版本迭代，对即有大数据集群实现灵活的服务扩展/集群的持续升级，使大数据业务分析平台满足发展的需求。

USDP 支持的大数据生态服务组件/工具，均来自于开源技术社区，通过集成和优化解决服务/工具之间的兼容性问题和漏洞修复，保障了业务选择的灵活性和安全性。全面兼容开源技术社区，使得用户基于 USDP 构建大数据业务架构时，无需担忧技术平台的捆绑问题，为企业技术发展获得更多自由。

USDP 支持对大数据集群在线横向扩展，具备单集群数百，数千台超大规模集群主机的管理与维护能力。集群性能将随主机规模的扩展呈线性增长，并且

不对主机规格限制，在业务使用中，可灵活根据需求规划不通过规格的主机的用途，如主要用来算力的补充、存储容量的补充，或规划建设某服务的独享的分布式集群，达到充分利用每台主机硬件设施的性能的目的。

2.2.3 存算分离架构

随着互联网、物联网、移动设备等技术的快速发展，数据开始爆炸式增长，对于海量数据的存储分析及治理，已经成为企业和组织面临的重要问题。如何有效地存储、处理、分析和利用这些数据，成为了一个亟待解决的问题。传统的数据架构已经无法满足现代大数据环境下的需求，因此需要一种新的架构来应对这些挑战。

大数据系统存算分离架构是一种新型的数据架构设计范式，旨在解决大数据环境下的一系列挑战。这种架构将数据存储和计算分开部署，通过高速网络进行数据传输，使得它们可以独立地进行扩展和管理，从而提高了系统性能、灵活性和资源利用率，为用途提供全面的大数据解决方案。

2.2.3.1 架构特点

结合 USDP 大数据系统提供的丰富的生态技术，可在实现存算分离架构时，

将数据存储、缓存加速和分析计算业务分层构建：

- 数据存储层

数据存储层负责数据的存储和管理，可以基于分布式文件系统实现，如 Hadoop Distributed File System（HDFS）等。HDFS 文件系统具有高可用性、可扩展性和容错性等特点，支持 PB 级规模数据存储与快速分析计算，并且支持多种数据访问模式。在 USDP 中，存储层主要以集群的形式独立提供数据存储和管理能力，并且允许同一存储集群同时被多个计算集群访问，完成数据处理分析工作。

- 缓存加速层

在存算分离的架构中，通过 Alluxio 可以作为计算集群的数据缓存层，将频繁访问的数据缓存在计算集群节点内存或者固态硬盘（SSD）中，从而加速数据访问速度。Alluxio 为上层应用提供一个统一的文件系统命名空间，整合多种底层存储系统，如 HDFS、S3 等。Alluxio 存储平台提供数据访问的速度保障，同时支持强一致性模型，保持数据的一致性。辅助上层应用及计算任务，实现跨不同存储集群数据访问，以及跨集群的数据共享，方便不同集群或云环境之间的数据迁移和同步。

- 数据分析计算层

数据计算层负责数据的计算和处理，可以采用各种计算框架，如 MapReduce、YARN、TEZ、Spark、Flink 等框架工具工具可以方便地对数据进行查询和分析，实现对多种格式、多种数据来源的海量数据的批量分析处理和实时分析处理任务，满足对数据进行清洗、整合、分析和处理等操作需求，并且具有高效的计算能力的可扩展性。在存算分离的架构中，计算层支持独立地扩展和管理，达到大数据系统管理灵活性扩展和性能提升的目的。

USDP 支持为计算集群中各计算引擎提供容器化部署形态，在数据计算任务提交时，可将计算任务部署在容器中，通过容器编排工具进行调度和任务生命周期管理。容器化的调度能力，对计算任务的打包、封装、部署、迁移，实现效率提升；可根据计算任务的需求动态调整资源，提升资源利用率；通过调度及容器

特性，实现计算任务间的隔离，避免不同任务相互干扰。

2.2.3.2 关键技术

大数据系统存算分离架构的关键技术包括分布式文件系统、计算框架、数据分析工具等方面。

分布式文件系统是实现数据存储层的核心技术之一，可以支持海量的数据存储和管理。其中，HDFS 是最为流行的分布式文件系统之一，它具有高可用性、可扩展性和容错性等特点，可以满足大数据环境下的需求。

计算框架是实现数据计算层的核心技术之一，可以支持海量数据的计算和处理。其中，MapReduce 和 Spark 是两种最为流行的计算框架，它们具有高效的计算能力、可扩展性和容错性等特点，可以满足大数据环境下的需求。

数据分析工具是实现数据分析层的核心技术之一，可以方便地对数据进行查询和分析。其中，Hive 和 Impala 是两种最为流行的数据分析工具，它们支持多种数据格式和来源，可以满足大数据环境下的需求。

2.2.3.3 适用场景

数据分析：存算分离架构可以提供灵活的数据访问和分析能力，可以支持多种数据源和格式的数据查询和分析，从而帮助企业更好地了解市场情况、掌握客户需求，为决策提供数据支持。

机器学习：存算分离架构可以提供高效的数据处理和计算能力，可以支持大规模的机器学习训练和推理任务，从而帮助企业进行数据挖掘、模式识别、预测等应用。

实时数据处理：存算分离架构可以提供实时数据处理的能力，可以支持实时数据采集、处理、分析和反馈等操作，从而帮助企业进行实时决策和控制。

数据仓库：存算分离架构可以支持数据仓库的建设和管理，可以提供高效的数据存储和查询能力，从而帮助企业进行数据整合、管理和分析。

2.3 核心优势

大数据业务系统作为企业信息系统的重要组成部分，近些年来亦成为信创的关注焦点之一。针对私有化部署场景，UCloud 推出的一站式智能大数据平台 USDP，可灵活构建于 IDC 物理服务器、云 IaaS 虚拟化，依托于自研的 USDP Manager 管理工具，实现对多套大数据集群的管理，并可使用户独享大数据集群。支持开源 Hadoop 全生态，进行集群、服务、监控告警、故障诊断等智能化的运维和管理操作，从而协助用户轻松构建和管理大数据业务分析处理能力。

2.3.1 Web 控制台，轻松管理大数据集群

USDP 集中化的管理控制台，在本次版本发布中，增加了集群管理视图功能，该视图使集群管理员对整个集群的服务状态一目了然；并可采取便捷的管理措施调整，保证系统的高可用和稳定性。

在 USDP 中，管理员可通过自动化向导的方式快速取得大数据服务对业务的支持；集中化的管理界面中，企业运营团队可以便捷的控制和调整服务配置和资源分配，以及一键开启/关闭 Kerberos，极大简化配置和管理的复杂性；自动化向导支持快速部署集群、扩展集群主机、给集群添加新的大数据服务，扩展服务实例等操作；结合预制的告警模板和自定义告警，使用户可以清晰掌握集群和集群中所有服务组件的运行状况。

2.3.2 一键开启或关闭 Kerberos 安全模式，为大数据服务的安全保驾护航

数据和服务的安全保障，一直是企业非常重视的问题。USDP 3.0 具备大数据服务安全性保护能力，通过流程化配置，快速开启对集群服务的检测，结合细粒度的权限控制能力，使得大数据集群服务及数据的安全性整体上得到保护。

在 USDP 中，支持了向导化和自动化管理安全模式，通过 Kerberos 的运用，为集群中的用户、服务和主机提供身份认证和授权管理能力，其强大的安全性和跨平台支持特性，确保只有经过身份验证的用户才能访问受保护的资源，为集群免受未经授权的访问和攻击提供了强有力的保护，帮助集群实现高度的安全性和可靠性。集群中的各个组件和服务（如 HDFS、YARN 等）可以通过 Kerberos 进行认证和授权管理，从而保护数据和应用程序的安全性。助力企业提升管理效率和信息安全水平。

2.3.3 大数据集群平滑升级，为需求升级提供有力支撑

集群的持续平滑升级能力，是保障用户基于该平台构建和管理的大数据服务，并获得持续维护和升级的重要支撑，平台及服务的灵活扩展能力，是应对随业务需求不断变化的架构优化的迫切需要。

企业在大数据相关业务方面的不断拓展和深入，数据类型变得更加丰富，数据量级爆发式增长，对数据处理时效的需求不断提高。因此，对于企业前期围绕数据仓库技术构建的大数据平台系统也带来了更高要求，亟需进行系统性升级和技术架构拓展，以满足企业业务持续发展的需要。

USDP 支持集群的持续升级和组件特性等持续拓展。用户可灵活选择大数据平台架构。无论是数仓架构的优化、流式计算的引入，甚至向流批一体架构、湖仓一体等架构演进， 通过 USDP 都能获得支撑。

2.3.4 配置角色组管理，有效提升资源利用率和管理效率

角色组的功能，是将服务配置按实例角色类型分配给相应的角色组，组中各

个角色继承这个组配置，助力大规模分布式服务便捷管理；根据集群主机环境、服务的特殊要求，为不同的主机或服务自定义分配不同的角色组，从而达到资源利用率和管理效率的有效提升。

2.3.5 UDH 更新及发布，多版本可灵活选择

UCloud 大数据组件发行套件，集成了丰富的开源项目，使企业可灵活构建一个功能先进的大数据系统；套件提供了强大的自动部署、管理和监控工具，便于用户操作维护大数据集群；套件中包含了更多的补丁和功能特性，为分布式大数据系统提供稳定性和性能保障。本次 UDH v3.0.0 的发布中，使其囊括的数据采集工具、流批计算引擎、调度系统、存储系统等服务，可全面兼容 Hadoop 3.3.4 版本生态。

2.3.6 宿主环境的修复能力，为运维人员减负

大数据集群宿主环境的修复及初始化能力，大大简化了基础运维在搭建大规模集群前，所需要进行的繁琐的基础设施环境准备工作；自动化修复工具良好的幂等性支持，能有效控制手动运维的出错概率，极大的降低运维人员的工作量和维护成本，保障了系统稳定性、可靠性及安全性。通过工具自动化的能力实现快速部署、配置、升级等复杂操作，使用户更加专注于数据分析业务的推进。

2.3.7 轻量便捷的操作

面对众多大数据服务，USDP 试图做到傻瓜式的“一键操作”，即，一键环境检查，一键环境修复，一键部署。省去用户费力排查环境问题，配置时间同步，系统优化等相关工作。

相比于目前已存在的部分运维工具，USDP 支持极其方便的傻瓜式部署方案，即，用户无需提前四处准备离线包，直接启动 USDP 服务后，通过界面的简单交互，在离线环境中即可完成一切部署过程。并且在部署的过程中，有详细的日志可供查看。

2.3.8 全面的生态支持

与同类产品不同，USDP 以完全中立，无捆绑的立场支持了超过 23 种服务，并将持续支持更多服务与组件的协同工作。

同时，USDP 产品中的服务完全基于 Apache 版本进行集成，并进行大量兼容性测试与压力测试，确保各个服务稳定运行的同时，还能兼顾 Apache 官方频繁的 BUG 修复。因此，用户也无需承担额外的学习成本。

2.4 客户痛点

2.4.1 大数据开源生态技术丰富，但选择成本较高

各开源社区对于处理和分析大规模数据的大数据技术生态，均提供了诸多开源项目，虽然这些项目提供了丰富的功能和灵活性，但在用户使用时可能会面临一些问题。

- 社区大数据生态系统中的项目通常是高度复杂的，需要一定的技术专长才能正确地配置、管理和使用它们。用户可能需要学习各个项目的工作原理、配置参数和最佳实践，这可能需要花费相当长的时间和精力。
- 社区大数据生态系统中的项目通常是独立开发和维护的，因此它们之间的集成和兼容性可能是一个挑战。用户可能需要处理不同版本之间的兼容性问题，以及在不同项目之间传递数据和结果时可能出现的格式和接口不匹配的问题。
- 处理和分析大规模数据需要大量的计算资源和存储资源。用户可能需要投入大量的资金和时间来配置和管理硬件基础设施，以确保系统的性能和可靠性。
- 大数据生态系统通常用于处理海量的原始数据，而这些数据往往需要经过清洗和预处理才能得到有用的信息。用户需要开发适当的数据管道和处理流程，以确保数据的质量和一致性。
- 对于企业来说，处理大规模数据时，安全性和隐私保护是非常重要的考

虑因素。用户需要采取适当的安全措施来保护数据，并确保只有经过授权的人员可以访问和使用数据。

- 社区大数据生态系统中的项目通常在不断演进和更新，发布新版本和补丁。用户需要跟踪这些更新，并评估是否需要升级他们的系统。这可能涉及到配置更改、数据迁移和应用程序代码的兼容性问题等，需要进行仔细的计划和测试，以确保平滑的升级过程。

2.4.2 自主可控的的私有化部署的商业化产品选择困难

为有效解决数据的隐私性及安全性问题，企业通常会考虑私有化建设适合自身业务发展需要的大数据分析处理系统及平台，实现自主可控的目标。而尽管市场中可供选择的能具备自主可控的私有化部署型商业化大数据产品层出不穷，但也存在一些难以选择的困难。

- 私有化部署型商业化大数据产品通常需要企业具备一定的技术实力和专业知识。对于缺乏相关技术团队或经验的企业来说，产品的安装、配置和维护可能是一项具有挑战性的任务。

- 在众多的商业化大数据产品中选择合适的产品可能是一项复杂的任务。企业需要考虑产品的功能、性能、可扩展性、安全性、技术捆绑性等因素，并根据自身需求和预算做出决策。这需要对市场进行详尽的调研和评估，可能需要投入大量的时间和资源。

- 自主可控的私有化部署型商业化大数据产品通常需要企业进行大规模的投资，包括购买许可证、硬件设备和基础设施建设等。需要仔细评估性价比、投资回报和长期成本效益。

- 使用私有化部署型商业化大数据产品需要企业具备足够的技术团队来进行系统管理、监控和维护。这可能需要企业增加技术人员的招聘和培训，增加人力成本和管理负担。

- 私有化部署型商业化大数据产品需要为企业持续提供更新和安全性升级保障，包括安装补丁、修复漏洞和升级系统等。这可能需要企业具备

相关技术能力和资源，并确保能够及时获取供应商的技术支持和更新。

- 在选择新的私有化部署型商业化大数据产品时，企业可能需要将现有的数据迁移到新系统中。这可能涉及数据格式转换、数据清洗和验证等工作，需要投入大量的资源和时间，同时还需要确保新系统与现有系统的兼容性。

- 私有化部署型商业化大数据产品通常在企业自己的内部部署和使用，这可能限制了与外部合作伙伴或其他组织的数据共享和协作。这可能对企业的业务拓展和创新能力造成一定的限制。

- 对于没有大数据处理和管理经验的企业来说，选择和使用商业化大数据产品可能是一个全新的领域。企业需要花费时间和资源来学习和理解相关概念、技术和最佳实践，以充分利用产品的功能和优势。

- 私有化部署型商业化大数据产品通常具有固定的功能和特性，对于某些企业来说可能无法满足特定的业务需求。在这种情况下，企业可能需要进行额外的定制开发或集成其他解决方案，这可能会增加复杂性和成本。

- 使用商业化大数据产品可能涉及到数据隐私、安全性和合规性等方面的风险。企业需要制定相应的风险管理策略和合规措施，确保数据的安全和合法使用，以避免可能的法律和合规风险。

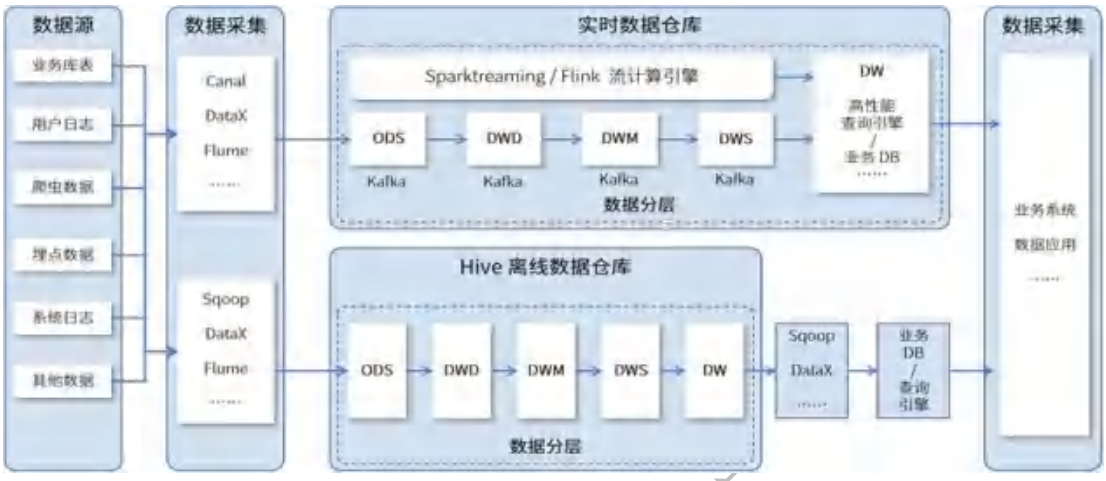
2.5 应用场景

2.5.1 离线/实时数仓架构场景

离线数仓将数据从源系统中抽取出来，经过清洗、转换和加载（ETL）等步骤，使数据按照一定的规则组织到数据仓库中，再通过报表等方式对数据进行分析和挖掘。而离线数仓所擅长处理的大规模数据能力，却不可避免的带来一定的数据延迟性，而实时数仓解决了这一问题。

在实时数仓中，以实时或近乎实时的方式处理数据，将数据通过流式处理引

擎（例如 Kafka、Flink 等）实时抽取、清洗、转换和加载到数据仓库中，再通过可视化工具等方式对数据进行实时监控和分析。基于 USDP 中丰富的大数据技术的选择，企业可根据自身的发展需要，灵活搭建并不断优化整合自己的大数据平台架构，构建适宜的数仓架构场景。



2.5.2 流批一体架构场景

流批一体架构的思想是将流处理和批处理整合在一起，提供更加全面和高效的实时数据分析能力。如使用 Flink 引擎及 Kafka 消息队列等引擎和工具的组合架构，可以在实时数据流中进行复杂的事件驱动处理，并兼顾批处理任务。依托于 Flink 引擎提供的丰富的流处理和批处理 API，和强大的状态管理和容错能力，使得流批一体架构可以更加可靠和高效地处理数据。在减少架构的复杂性和维护成本的同时，使企业达到更加灵活地处理数据的目的，并更好地适应业务需求的变化。而 USDP 中提供了丰富的大数据生态服务、工具和框架的支持，能很好帮助企业构建灵活且复杂的大数据处理架构。

2.5.3 数据湖/湖仓一体架构场景

在将数据湖和数据仓库的优点深度结合的“湖仓一体”数据架构中，以多种格式的数据统一存储为基础，可很好的避免数据冗余和一致性问题；统一高效的数据处理和清洗带给数据质量和可用性有力保障；在该架构的支撑下，企业可开展如批处理、流处理、实时处理等多种模式的数据分析方式，借助 SQL 查询分析、OLAP 分析、数据挖掘、机器学习来满足应用的分析需求；通过该架构，

更有利于企业实现数据的管理和治理，从而提高数据的可靠性和可信度。Hadoop 生态系统、HBase、Hive 等存储框架、Kafka 流处理平台、Presto 查询引擎、Flink/Spark 等流/批处理框架、以及 Hudi、Iceberg 等引擎和存储格式，正是构建湖仓一体架构所需要的，USDP 可以有效解决这些工具、框架和服务的相互兼容问题，以及统一的管理和监控维护，给企业带来丰富且便捷的支持，以满足企业对数据的各种需求。

2.5.4 数据仓库

目前国内最常用的数仓模型为：维度数仓。

其最简单的描述就是按照事实表、维度表来构建数据仓库、数据集市。在维度建模方法体系中，维度是描述事实的角度，如日期、客户、供应商等，事实是要度量的指标，如客户数、销售额等。

通过 USDP 管理平台，用户可以部署构建维度数仓所需的一切服务，快速构建企业数据中台。

2.5.5 机器学习

机器学习是一类算法的总称，这些算法企图从大量历史数据中挖掘出其中隐含的规律，并用于预测或者分类。

在机器学习领域，对运算往往有大量需求，此时通过 USDP 部署 YARN 以及 Spark、Flink 等分布式运算框架，搭配官方算法或自研算法，即可事半功倍的进行机器学习开发。同时，在深度学习领域，建模所需的大量数据，也可以存储于 HDFS，从而真正实现一站式开发。

2.5.6 业务信息检索

USDP 中提供的数仓类服务套件，可以方便用户快速针对 OLTP 系统进行数据读写，也许并非会使用到所有服务，但 USDP 依然支持在成熟的部署方案中，选择独立的服务进行部署。

例如，业务系统需要使用 Solr 存储服务器日志，用户只需要在 USDP 控

制台中选择部署 Solr 即可，一切都是水到渠成。

2.6 交付模式

USDP 支持以纯软件的方式、超融合一体化的方式进行项目交付。

项目中的三个关键阶段：

- **售前阶段：**

从前期的需求沟通入手，逐步明确需求范畴，并围绕需求提供方案设计与建议；当方案能很好的满足项目需求时，即可按照方案进行 POC 验证；输出交付规划材料，为下个阶段提供实施指导。

- **交付实施阶段：**

围绕交付规划要，完成系统的搭建交付工作；协助用户完成首个大数据基础平台的创建；辅助用户对交付环境进行验证，确保符合规划，并且系统及集群运行正常，随机交付给用户进行使用；

- **售后维保阶段：**

在该阶段中，为用户提供 5*8、7*24、金牌服务的三种可选售后标准，用户可根据需求选择其一，进入售后维保状态。三种可选售后标准的具体细则，可参考签订的合约中有关售后维保的具体细则。

3 USDP 平台网络架构

3.1 物理网络架构

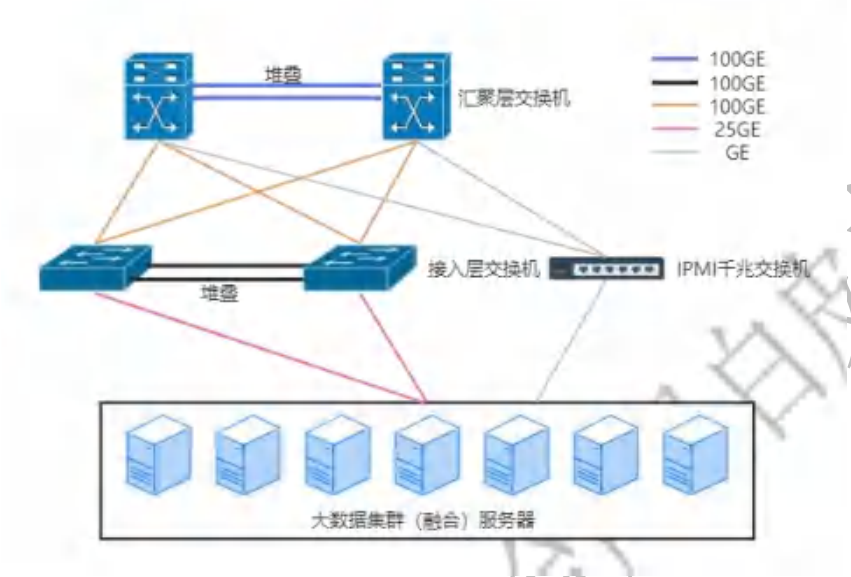


图 2-1 USDP 标准组网架构

说明：

- USDP 管理服务未单独规划服务器（资源占用较小），复用集群的其中一台服务器部署；
- 大数据存储服务和管理服务混合部署模式；
- 大数据集群服务器内网（cluster）与外网（public）无隔离，25GE 网络；
- 接入和汇聚交换机各 2 台做堆叠，保障设备冗余和带宽；
- 大数据集群节点服务器与接入交换机间规划成动态链路聚合（LACP）；
- 接入交换机上联接入到汇聚交换机，静态聚合；
- 节点资源管理通过 MBC 或 IPMI 接入千兆交换机，上联接入汇聚或核心；
- 一组接入层交换机可承载最大服务器数量不超过 N-2 台，N 为单交换

机下联口数量；

3.2 部署规划

为更好的满足业务需求场景，合理规划大数据服务并有效力发挥基础资源运用优势，并通过 USDP 提供的初始化工具在安装部署时结合规划进行初始化操作。

本章节将涉及的名词解释：

- USDP Server：是用户独享的整个大数据系统的管理服务，提供一键安装包、修复工具、可视化的控制台。安装完成后，用户可通过 USDP Server 提供的控制台管理整个大数据系统，包括对多个大数据集群的管理。
- MySQL：是 USDP Server 依赖的管理元数据存储数据库。
- Hadoop Cluster：是通过 USDP 控制台创建并管理的 1-N 个独立的大数据集群。
- 大数据服务：是 Hadoop 集群中各个服务软件，例如：HDFS、Hive、Spark、Hue 等。
- 服务组件：是各个大数据服务自带的服务构成的模块。如 DataNode、NameNode 是 HDFS 服务的两类重要的附件。

请根据自身需求，参考以下三类部署方案。

3.2.1 最小化规模部署

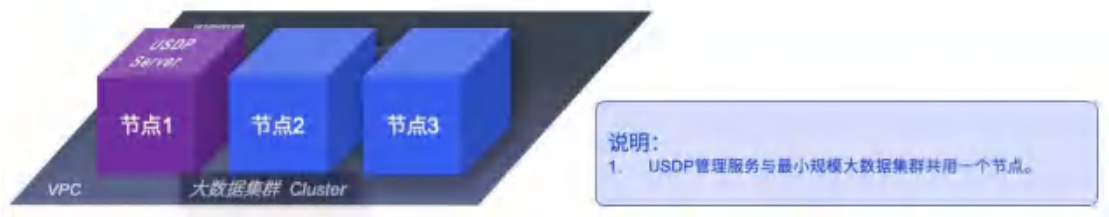


图 2-2 USDP 最小化规模部署示意图

本方案适用于当业务量较小、资源较为紧俏时，以及用户希望搭建一个最小规模的环境时，参考来协助实现智能大数据服务的部署。

因为 HDFS 中 JournalNode 高可用需要部署在 1 个以上的奇数节点上（推荐 3），且数据存储副本为 3，因此最小部署规模为 3 个节点。

节点/服务	最低配置	USDP-Server	MySQL	NTP	Hadoop Cluster	大数据集群内各服务部署规划
节点 1(host01)	16C 32G sys 100GB data 300GB	Y	Y	Y	Cluster1-节点 1	自行规划，数据盘建议按需调整
节点 2(host02)	16C32G sys 100GB data 300GB	-	-	Y	Cluster1-节点 2	自行规划，数据盘建议按需调整
节点 3(host03)	16C32G sys 100GB data 300GB	-	-	Y	Cluster1-节点 3	自行规划，数据盘建议按需调整

表 2-1 USDP 最小化规模部署规划

在该方案中，用户还可将 USDP Server、MySQL、NTP 可以分散到上述 host[01-03] 共三个节点上。

3.2.2 单集群多节点规划

图 2-3 USDP 单集群多节点规划部署示意图

本方案适用于当业务量较小、资源较为紧俏时，以及用户希望搭建一个最小规模的环境，但 USDP Server、NTP 服务器、MySQL 服务器能与大数据集群相对独立的场景。参考本章节内容，来协助实现智能大数据服务的部署参考。

节点/服务	最低配置	USDP Server	MySQL	NTP	Hadoop Cluster	大数据集群内各服务部署规划
USDP Server(host01)	8C 32G sys 100GB data 300GB	Y	Y	Y	USDP Server 节点	自行规划，数据盘建议按需调整
节点 1(host02)	16C32G sys 100GB data 300GB	-	-	Y	Cluster1-节点 1	自行规划，数据盘建议按需调整
节点 2(host03)	16C32G sys 100GB data 300GB	-	-	Y	Cluster1-节点 2	自行规划，数据盘建议按需调整
节点 3(host04)	16C32G sys 100GB data 300GB	-	-	-	Cluster1-节点 3	自行规划，数据盘建议按需调整

表 2-2 单集群多节点集群规划

在该方案中，用户可将 USDP Server、MySQL、NTP 中的其中 1 到 2 个，分散部署到上述 host[01-04] 共四个节点之外的节点去，例如 MySQL 可复用用户现有其他业务系统的 MySQL 数据库。

3.2.3 多集群多节点规划

本方案适用于当业务较复杂、大数据分析业务在不同业务领域有独立隔离的诉求，资源相对较为充裕时，用户希望通过 USDP 创建并管理多个大数据集群环境，并且 USDP Server、NTP 服务器、MySQL 服务器能与大数据集群相对独立的场景。参考本章节内容，来协助实现智能大数据服务的部署参考。

(1) 网络规划

规划至少三个 VPC，示例如下：

VPC 规划	用途	与 VPC-C1 互通	与 VPC-C2 互通
VPC-M(192.168.0.0/27)	管理区	Yes	Yes
VPC-C1(10.0.0.0/16)	分析业务区 1	-	Yes
VPC-C2(172.20.0.0/16)	分析业务区 1	Yes	-

表 2-3 多集群网络规划

若用户网络环境为非 VPC 的其他方式，如 Vlan，则 Vlan 间的划分和互通性配置，与上述 VPC 同理。

规划网络时，建议各 VPC 的 IP 地址段均不重叠，避免日后业务需求调整时，VPC 打通将导致出现地址冲突的网络异常。

VPC 数量可根据需求灵活调整。

VPC 间互通控制，是在 USDP 操控范围之外独立控制，如通过云平台中操作控制。

(2) 准备服务器节点

参考如下表格示例规划：

节点/服务	最低配置	VPC/Vlan	USDP Server	MySQL	NTP	Hadoop Cluster
USDP Server(host01)	8C 32G sys 100GB data 300GB	VPC-M	Y	Y	Y	USDP Server 节点
节点 1(host02)	16C32G sys 100GB data 500GB	VPC-C1	-	-	Y	Cluster1
节点 2(host03	16C32G sys 100GB data 500GB	VPC-C1	-	-	Y	Cluster1

节点/服务	最低配置	VPC/Vlan	USDP Server	MySQL	NTP	Hadoop Cluster
)						
节点3(host04)	16C32G sys 100GB data 500GB	VPC-C1	-	-	-	Cluster1
节点1(host05)	16C32G sys 100GB data 500GB	VPC-C2	-	-	-	Cluster2
节点2(host06)	16C32G sys 100GB data 500GB	VPC-C2	-	-	-	Cluster2
节点3(host07)	16C32G sys 100GB data 500GB	VPC-C2	-	-	-	Cluster2

表 2-4 多集群且集群间网络互通规划

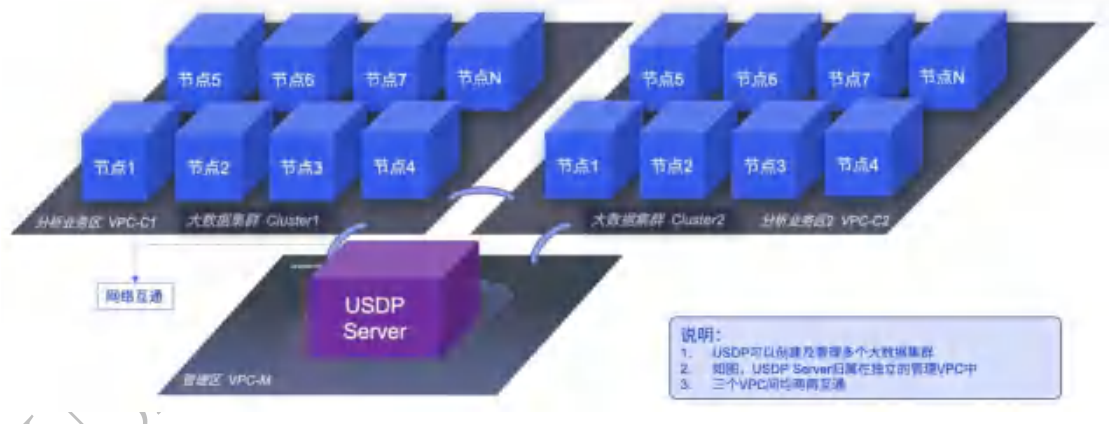


图 2-4 USDP 管理服务管理多个大数据集群且集群间网络互通示意图

如上图所示，因三个 VPC 均两两互通，因此，“大数据集群 Cluster1”集群与“大数据集群 Cluster2 之间”集群可进行跨集群复制数据等操作。

(3) 网络控制调整

若将“网络规划”进行调整，如下表所示：

图 2-5 集群间相互隔离示意图

如上图所示，实现 VPC-M 与 VPC-C1 互通、VPC-M 与 VPC-C2 互通、而 VPC-C1 与 VPC-C2 不互通的目的，因此，“大数据集群 Cluster1”集群与“大数据集群 Cluster2 之间”集群无通讯网络条件；此模式，适合在多个分析系统业务，需要灵活控制、且需要分类管理的场景中，以满足控制需求，达到提供数据分析业务安全性的目的。

3.3 硬件选型

3.3.1 最低配置方案

	测试环境最低配置	生产环境最低配置	备注
CPU	不低于 16 核	不低于 16 核	vCPU
内存	单台主机不低于 32GB	单台主机不低于 32GB	
网卡	1 个 10G 网口（无性能要求）	2 个 25G 网口（做 bond4）	如需网卡级别冗余，2 张 10GB 网卡
系统盘	1 块 SSD 60GB 以上	2 块 SSD 480G（做 raid1）	生产系统盘做 RAID 1
数据盘	根据需要灵活规划数量及容量	无要求，建议按节点使用场景合理规划； 若需要，最少 1 块 SATA 盘（不做 Raid），单盘不低于 500GB，盘数根据需求灵活规划；	推荐 HDFS 三副本； 建议考虑多盘并发读写速率与网卡速率限制匹配问题；
接入交换机	1 台万兆以太网交换机	2 台 25GE 以太网交换机（堆叠）	
服务器数量	三台	三台	

表 2-5 USDP 最低配置方案建议

注意：

- 最低配置建议，只保证平台正常的部署，基本的稳定运行，不包括实际业务数据容量、副本数、性能要求；
- 生产环境，服务器硬件和架构层面必须保证冗余机制；

3.3.2 推荐配置

(4) 服务器推荐配置

机型	配置描述
UTrionC4310-8T-3-1	Factor Form 2U CPU Intel Xeon Silver 4310 Processor(12CORES_2.1GHz_120W_X86) *2 DDR4_32GB_RDIMM_3200MHz *6 OS HDD 480G_SSD_SATA3_512E_2.5"_6Gb/s *2 Data HDD SATA3_HDD_8TB *6 LSI-9311-8I 双口 25G 光口网卡(不含光模块)*1 PSU=800W*2/导轨
UTrionC4314-8T-3-1	Factor Form 2U CPU Intel® Xeon® Silver 4314 Processor(16CORES_2.4GHz_135W_X86) *2 DDR4_32GB_RDIMM_3200MHz *8 OS HDD 480G_SSD_SATA3_512E_2.5"_6Gb/s *2 Data HDD SATA3_HDD_8TB *6 LSI-9311-8I 双口 25G 光口网卡(不含光模块)*1 PSU=800W*2/导轨
UTrionC4314-960G-3-1	Factor Form 2U CPU Intel® Xeon® Silver 4314 Processor(16CORES_2.4GHz_135W_X86) *2 DDR4_32GB_RDIMM_3200MHz *8 OS HDD 480G_SSD_SATA3_512E_2.5"_6Gb/s *2 Data HDD 960G SSD SATA3 _512E_2.5"_6Gb/s *12 LSI-9311-8I 双口 25G 光口网卡(不含光模块)*1 PSU=800W*2/导轨

表 2-6 硬件选型-服务器推荐配置表

(5) 网络设备推荐配置

类型	SKU	产品型号	配置
万兆接入交换机（10GE）	S-RG-6250-3-1	锐捷 RG-S6250-48XS8CQ	48*10G+2*40G+2 个扩展插槽（可再扩展 4*40G），1U，满负荷 234W
	S-H3C-6520-3-1	华三 S6520X-54QC-EI	48*10G+2*40G+2 个扩展插槽（可再扩展 4*40G），1U，满负荷 234W
25GE 接入交换机	S-RG-48VS8CQ-3-1	锐捷 RG-S6510-48VS8CQ(V2.0)	48*25G+8*100G，1U，满负荷 413W
	S-H3C-6850-3-1	华三 S6850-56HF	48*25G+8*100G，1U，满负荷 413W
千兆交换机（IPMI 带外管理）	S-RG-6000C-3-1	锐捷 RG-S6000C-48GT4XS-E	48*1G+4*10G+1 个扩展插槽（可再扩展 8*10G 或者 2*40G），1U，满负荷 93W
	S-H3C-5560-3-1	华三 5560X-54C-EI	48*1G+4*10G+1 个扩展插槽（可再扩展 8*10G 或者 2*40G），1U，满负荷 93W

表 2-7 USDP 硬件选型-网络设备推荐配置表

3.4 平台资源占用

3.4.1 USDP 管理服务架构

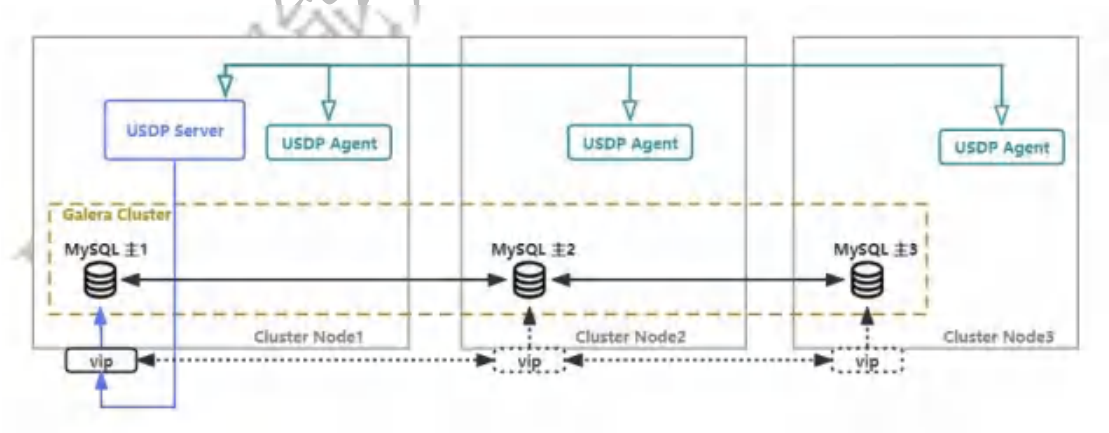


图 2-6 USDP 管理服务逻辑示意图

USDP 采用多主架构 DB（Galera Cluster）实现管理状态数据存储，同时，该 DB 也用做集群大数据服务元数据存储（如 Hive 等），通过该架构，使得

管理数据及大数据服务元数据存储系统得到高可用保障；

3.4.2 USDP 管理服务资源占用说明

服务名称	vCPU	内存（GB）	磁盘（GB）	备注
USDP Server	2	1.5	-	-
USDP Agent	0.2	1~1.5	-	分布于 USDP 所管理的所有主机
DB	2	1~4	100GB（预留）	分布于 USDP 所管理的 3 台主机中

表 2-8 USDP 管理服务资源占用说明参考

3.5 机柜空间规划

注意：

- 单机柜上架的所有服务器及网络设备总耗电，不超过 IDC 机房机柜电流限制（如单机柜 20 安，即该机柜承载的最大功率为 4400W）。

网络设备和服务器的物理机柜空间规划如下图所示：

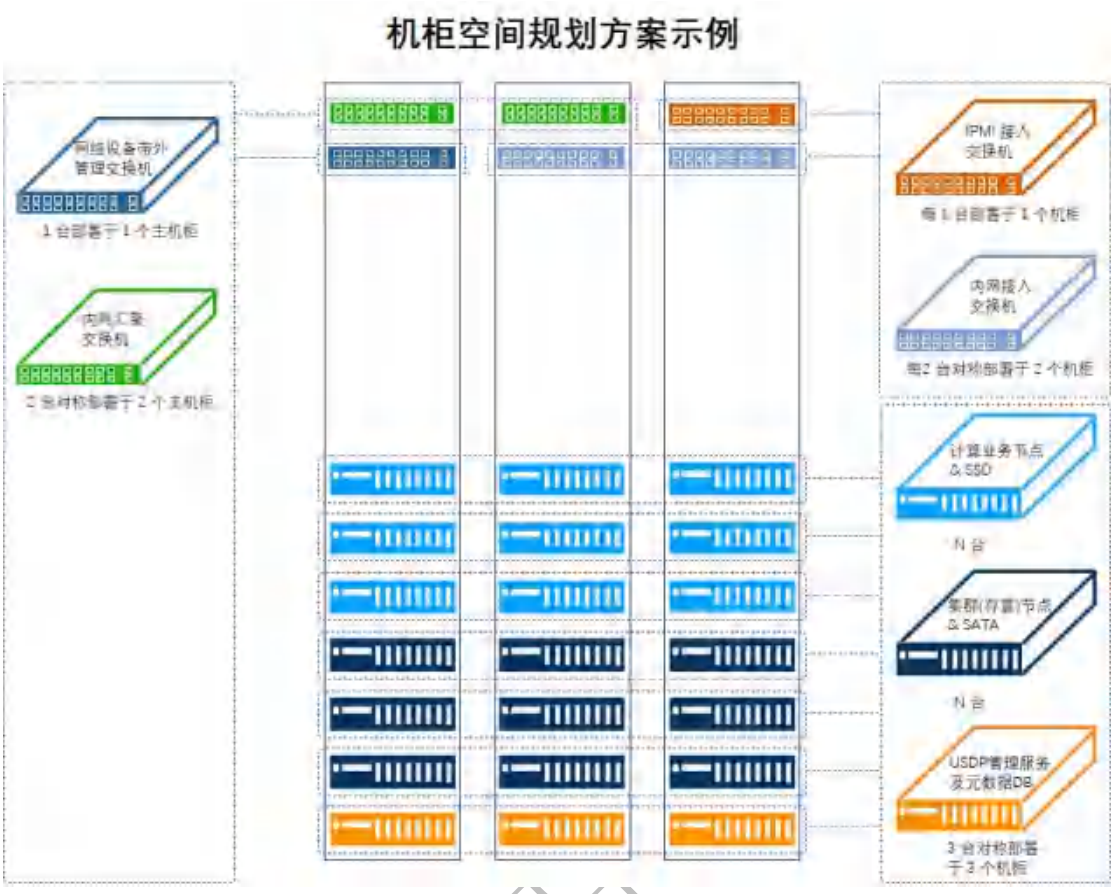


图 2-7 机柜空间规划方案示意图

所有设备在机柜中均匀部署，实现大数据集群各类节点分散于多个机柜，单机柜掉电或故障，故障范围影响相对有限。一个 20 安机柜可支撑约 8 个节点 (推荐配置 1 机型），根据网络架构设计一组接入交换机可支撑约 45 个节点，即一组接入交换机支撑约 6 个机柜。6 个机柜为 1 组，平均 1 组机柜支撑约 45 个节点、1 组内网接入交换机、1 组内网汇聚交换机、1 台 IPMI 接入交换机。

如上图项目案例中的设备包括 4 台业务交换机、1 台运维管理交换机、21 台服务器设备及 3 个机柜：

- 一组内网汇聚交换机对称部署于 2 个机柜，即其中两个机柜各部署 1 台；
- 一组内网接入交换机对称部署于 2 个机柜，即其中两个机柜各部署 1 台；
- 1 台 IPMI 接入交换机和 1 台网络设备带外管理交换机部署于 2 个机柜；
- 3 台管理节点（也用于集群（存算）节点）对称部署于 3 个机柜，即每

个机柜各部署 1 台；

- 9 台集群（存算）&SATA 节点对称部署于 3 个机柜，即每个机柜各部署 3 台；
- 9 台计算业务&SSD 节点对称部署于 3 个机柜，即每个机柜各部署 3 台。

若服务器分集群部署，建议不同各集群的服务器对称部署于多个机柜。

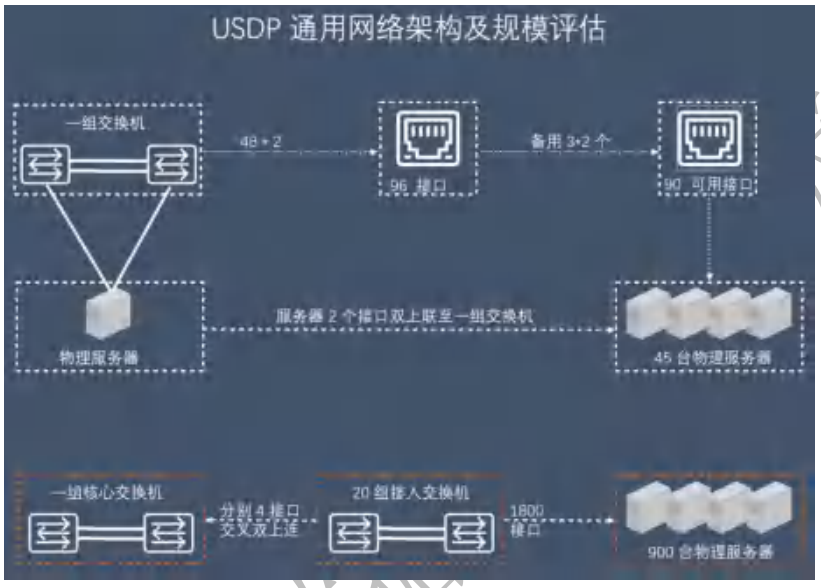


图 2-8 通用网络架构及规模评估建议

4 核心产品服务

4.1 基本概念

4.1.1 集群

大数据集群是指由多台服务器组成的集群，用于处理和存储大规模数据集的分布式计算环境。通过集群，可利用并行计算和分布式存储的能力来处理海量数据。集群的每个节点都可以是一台独立的计算机或服务器。

4.1.2 分布式存储系统

通常使用分布式文件系统（如 Hadoop 的 HDFS）来管理数据的存储和访问。分布式文件系统将数据被分割成多个小块，每个小块分布在不同的节点上。这样可以实现数据的并行处理，每个节点可以独立地处理自己负责的数据块。分布式存储系统提供了高可用性和容错机制，以确保数据的可靠性和持久性。

4.1.3 分布式计算框架

大数据集群通过使用分布式计算框架（如 Apache Spark、Apache Flink 等）来执行复杂的数据处理任务，如数据清洗、数据分析、机器学习等。这些框架提供了高级的编程接口和工具，简化了在分布式环境中进行数据处理的开发和管理难度，并行处理技术，大数据集群可以显著提高数据处理的速度和效率。

4.2 USDP Manager

4.2.1 概述

USDP Server 是 USDP 管理服务模块，负责整个大数据平台的管理控制逻辑，来实现对多套大数据业务集群中的管理 Hive、HBase、Spark、Flink、Presto 等开源的大数据组件，例如进行集群节点、服务配置、监控告警、故障诊断等智能化的运维管理操作，从而协助您轻松构建和管理大数据业务的分析处理能力。

USDP Agent 为 USDP 工作节点控制端服务，用于管理、操作所在节点以

及所在节点上的大数据服务。Server 通过 Restful API 派发相应指令给 Agent，Agent 收到指令后，根据指令类型执行相应指令。在此过程中，会通过接口鉴权的方式，保障平台隐私安全。得益于 Worker 无状态特性，从而保障了 Worker 本身占用资源极少，更加轻量，更易部署、升级、与横向扩展。

USDP Server 需配备一组 Galera Cluster MySQL 多主集群高可用方案存储 USDP 管理平台状态数据以及各大数据集群的相关的元数据信息。

USDP 需要最少 3 个节点，支持上千节点的集群规模，同时，允许 Manager 与 Worker 等相关服务部署在相同的节点上，这样满足大型业务的同时，也尽可能帮助用户使用较小的成本满足小型业务对数据分析的诉求。

4.2.2 USDP 管理平台架构

- USDP Server 通过 HTTP 方式连接 USDP Agent 执行具体的大数据组件安装卸载任务、实例启动停止重启任务、配置文件生成更新任务等。
- USDP Agent 通过 HTTP 方式连接 USDP Server 上报实例进程存活情况、异常实例（逻辑状态运行中但是实际服务进程挂掉的实例）拉起。
- USDP Agent 同时作为一个 prometheus exporter 增加一些服务实例存活状态相关的监控指标。

4.3 USDP Worker

4.3.1 概述

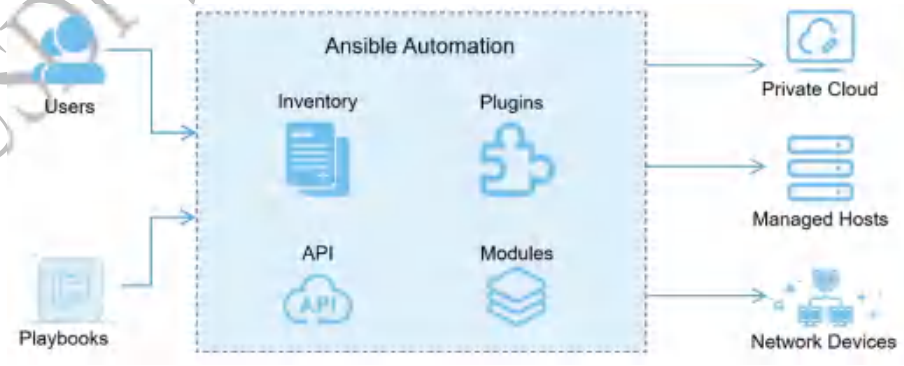
USDP Worker 为 USDP 从节点控制端服务，用于管理、操作所在节点以及所在节点上的大数据服务。Manager 通过 Restful API 派发相应指令给 Worker，Worker 收到指令后，会无状态、幂等性的执行该指令。并在此过程中，会通过接口鉴权的方式，保障平台隐私安全。得益于 Worker 无状态特性，从而保障了 Worker 本身占用资源极少，更加轻量，更易部署、升级、与横向扩展。

4.4 修复工具

4.4.1 概述

修复工具是大数据集群宿主环境的修复及初始化重要工具集，通过修复工具的使用，可大大简化基础运维在搭建和维护大规模集群过程中，所需要进行的繁琐的基础设施环境准备工作；自动化修复工具良好的幂等性支持，能有效控制手动运维的出错概率，极大地降低运维人员的工作量和维护成本，保障了系统稳定性、可靠性及安全性。通过工具自动化的能力实现快速部署、配置、升级等复杂操作，使用户更加专注于数据分析业务的推进。

4.4.2 架构/特性



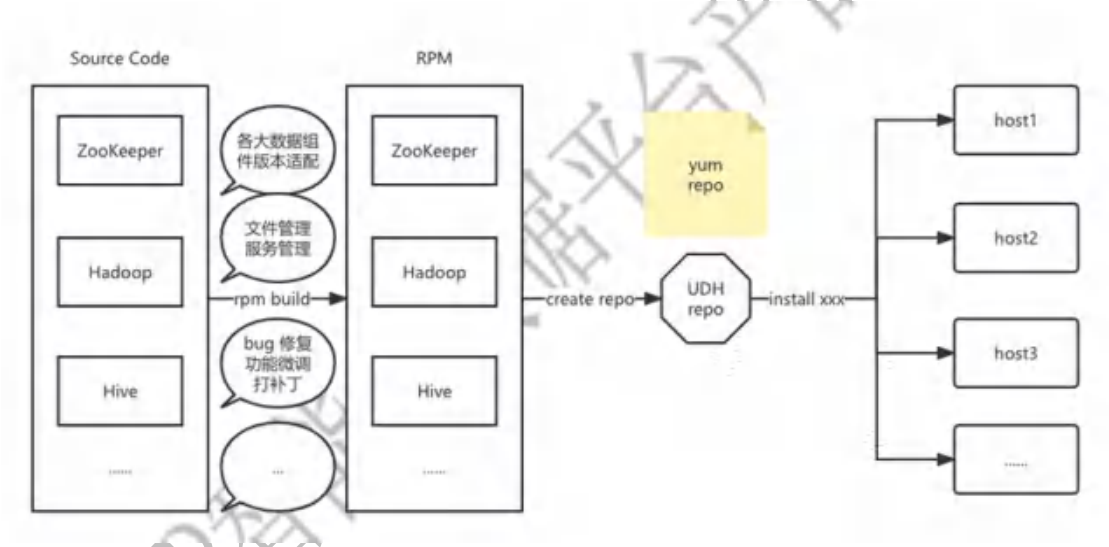
通过 Ansible 框架对相应主机的操作系统基础软件包、防火墙、selinux、交换分区、时间同步、数据库初始化、USDP 管理服务等进行了部署和更新操作。

4.5 UDH 源

4.5.1 概述

UDH 源是 UCloud 大数据组件发行套件，集成了丰富的开源项目，其囊括的数据采集工具、流批计算引擎、调度系统、存储系统等服务，使企业可灵活构建一个功能先进的大数据系统；USDP 提供了强大的自动部署、管理和监控工具，便于用户操作维护大数据集群；套件中包含了更多的补丁和功能特性，为分布式大数据系统提供稳定性和性能保障。UDH 源是 USDP 及已创建使用中的大数据集群得到本次持续更新升级的重要依赖部分。

4.5.2 架构/特性



UDH 是 USDP 根据大数据生态及部分项目使用场景收录开发编译生成的一个大数据组发行版集合。结合 USDP 中对于 UDH 的管理和服务实例部署的管理实现所需大数据组件的安装或卸载等管理。

在各大数据组件发行的源码基础上进行了整体 UDH 发行版本中的大数据组件版本的兼容的适配调整，增加部分 bug 修复和功能调整的补丁。

4.5.3 生态组件列表

当前发行的 UDH v3.2 支持的大数据生态服务组件及工具有：

生态服务	版本	服务说明
Alluxio	2.9.3	统一的数据编排与数据缓存加速层，使应用程序能够以内存级速度与任何存储系统中的数据进行交互
Atlas	2.3.0	元数据管理服务
Canal	1.1.6	数据库增量同步工具，常用于实时数据分析架构的数据采集业务中
DataX	3.0	数据采集与转储服务，提供异构数据源离线同步功能
DolphinScheduler	3.1.8	分布式易扩展的可视化 DAG 工作流任务调度开源系统
EFAK	3.0.1	Kafka 可视化管理服务，包括监控 kafka 集群、可视化消费者线程、偏移量、所有者等。
Flink	1.16.1	分布式处理引擎，用于对无限制和有限制的数据流进行有状态的计算，具有高性能、易用和普遍性等特点
Flume	1.11.0	分布式的海量日志采集、聚合和传输的系统
HBase	2.4.13	高可靠性、高性能、面向列、可伸缩的分布式存储系统
HDFS	3.3.4	分布式文件系统，作为结构化数据的底层存储
Hive	3.1.3	基于 Hadoop 的数据仓库工具，支持通过 Hive SQL 查询方式来分析存储在 Hadoop 分布式文件系统中的数据
Hudi	0.13.0	构建在 Hadoop 文件系统之上的数据湖平台，提供更新数据和删除数据的能力
Hue	4.11.0	提供可视化的运营和开发 Hadoop 应用的管理服务
Iceberg	1.3.0	用于大型分析数据集的高性能开放表格式，其使用类似于 SQL 表的高性能表格式将表添加到

		各种计算引擎中
Impala	4.1.2	分布式计算服务
Kafka	2.8.1	高吞吐量的分布式发布-订阅消息系统
Knox	2.0.0	提供对 Hadoop 集群的统一访问控制应用程序网关，支持 REST API 和 UI 交互方式
Kylin	4.0.3	在 Hadoop/Spark 之上的多维分析（OLAP）巨量数据 SQL 查询的分布式分析引擎
Kyuubi	1.7.1	用于在数据湖上提供 Serverless SQL 的分布式、多租户网关
Oozie	5.2.1	数据处理作业的工作流协调调度服务
Phoenix	5.1.2	是 HBase 的 SQL 驱动，Phoenix 可以使用标准的 JDBC 的 APIs 去代替常规的 Hbase 客户端的 APIs 操作数据表，如插入数据和查询 Hbase 数据
Ranger	2.4.0	集中式的权限管理框架，用于对 Hadoop 生态中的组件进行细粒度的权限访问控制
Solr	8.11.2	高性能的企业级搜索平台
Spark	3.2.3	基于内存的大数据分析引擎,可用于构建大型的、低延迟的数据分析应用程序,
Sqoop	1.4.7	用于在 Hadoop 和关系型数据库之间进行数据转移的工具
StreamPark	2.1.1	流处理极速开发框架，兼容流批一体 & 湖仓一体的应用场景，提供一站式流处理计算平台
TEZ	0.10.2	构建在 YARN 之上的应用程序框架，支持有向无环图（DAG）作业的计算框架来处理数据
Trino	418	分布式计算服务，用于快速运行的 SQL 任务的分布式查询引擎，实现即席查询场景
YARN	3.3.4	资源管理和调度服务，为上层应用提供统一的资源管理和作业调度

Zeppelin	0.10.1	可以启用数据驱动、交互式数据分析和与 SQL，Scala，Python，R 等的协作的 Web 的笔记本
ZKUI	2.0	Zookeeper 可视化管理服务
ZooKeeper	3.5.10	分布式应用程序协调服务，用于统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等

4.6 集群管理视图

4.6.1 概述

集群管理视图功能使集群管理员对整个集群的服务状态一目了然；可对集群中的大数据服务采取便捷的管理措施调整，如集群的安全性开启与操作、集群涉及的各类服务指标的可视化监控等，使管理人员获得快速有效的集群级别操控目的。在 USDP 中，管理员可通过自动化向导的方式快速取得大数据服务对业务的支持；集中化的管理界面中，企业运营团队可以便捷地控制和调整服务配置和资源分配，以及一键开启/关闭 Kerberos，极大简化配置和管理的复杂性；自动化向导支持快速部署集群、扩展集群主机、给集群添加新的大数据服务，扩展服务实例等操作；结合预置的告警模板和自定义告警，使用户可以清晰掌握集群和集群中所有服务组件的运行状况。

4.7 自定义监控图表

监控能力扩展功能为用户提供了更强大和灵活的监控工具，满足不同业务场景下的监控需求。新增的自定义监控图表功能，使用户可以按照业务关注点自主选择关键的监控指标，并以图表化方式展示，实现对集群、主机和大数据服务关键状态反馈的快速捕获。用户可以根据自己的需求进行指标汇总、对比等处理，便捷地获得所关注的监控数据图表化呈现。v3.1 版本中对监控能力的扩展，提供了更灵活、个性化的监控解决方案，帮助用户更好地管理和优化大数据平台的运行情况和性能。

此外，该功能还支持用户自定义业务监控的扩展。用户可以通过扩展采集端的方式，采集平台默认未采集的系统或服务的监控指标。借助 USDP 监控模块，用户可以自定义拓展监控，满足特定业务需求。使得用户能够更全面地监控和评估与业务相关的指标，从而更好地把握系统的运行状况和性能表现。

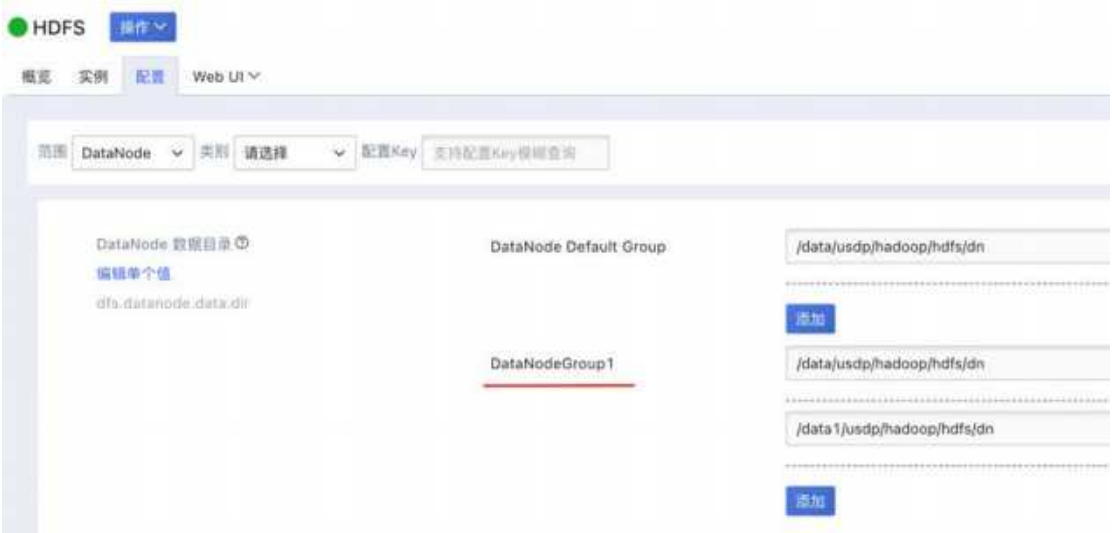
这些监控能力的扩展功能适用于各种应用场景。例如，在大规模数据处理的场景下，用户可以自定义监控图表，以展示关键指标的趋势和变化，帮助监控任务的执行和资源利用；在复杂的服务架构中，用户可以自定义业务监控，以追踪特定服务的性能和可用性。

4.8 角色配置组

4.8.1 概述

角色组的功能，是将服务配置按组件实例角色类型相应的分配同一角色，组中各个角色继承这个组配置，助力大规模分布式服务便捷管理；根据集群主机环境、服务的特殊要求，为不同的主机或服务自定义分配不同的角色组，从而达到资源利用率和管理效率的有效提升。

4.8.2 架构/特性



举例：支持某种磁盘分布的主机归属到某个配置分组，然后个性化配置 DataNode 数据目录。

4.9 服务配置与历史回滚

4.9.1 概述

通过 USDP 控制台 Key / Value 化配置各大数据组件服务的配置，并支持查看配置修改历史和对历史记录中某些配置修改进行回滚。

4.9.2 架构/特性

USDP 支持服务级别的配置“历史记录和回滚”操作，每一次更改某服务的配置，则会被记录至该服务的“配置”管理页面“历史记录和回滚”中。

上图为“HDFS”的服务更改历史列表。

点击列表中某条消息，可查看该历史版本的更改明细；使用者也可根据需求判断，是否将当前配置文件“回滚”至该历史版本更改前的状态，如上图所示，若需要回滚，请点击“版本详细信息”对话框的“还原配置更改”按钮；

如上图所示，此时产生了一条“已还原”的配置更改历史。

4.10 资源池管理

集群资源池管理配置功能为用户提供了便捷的管理工具，可以对 YARN 和 Impala 的集群资源池进行可视化动态配置。对于 YARN，用户可以选择容量调度模式，而对于 Impala，用户可以选择公平调度模式。通过这个工具，为用户提供了更灵活、易优化的资源管理方式，使不同的业务获得整个集群计算资源的合理分配，规避资源争抢等问题，从容地应对复杂的业务需求和资源分配挑战。

集群资源池管理配置功能的应用场景非常广泛。例如，在一个大数据集群中，

可能有不同的业务部门或应用，需要使用不同的计算资源，而资源的分配和管理可能面临挑战。通过集群资源池管理配置功能，用户可以根据实际需求，为各个业务部门或应用分配合适的资源，确保资源的公平分配和有效利用。这样一来，不同的业务可以更高效地运行，避免资源争夺导致的性能下降或系统崩溃。同时，管理员也可以更好地监控和管理集群中资源的使用情况，及时调整资源分配策略，提高集群的整体性能和效率。总之，大数据集群资源池的管理配置功能为用户提供了更灵活和优化的资源管理方式，帮助他们更好地应对复杂的业务需求和资源分配挑战。

4.11 计算任务监控

支持对 YARN 和 Impala 算力集群中运行的应用任务进行实时监控。用户可通过监控便捷地查看任务所属类型、资源池、执行用户、资源消耗、运行时长以及累计占用 CPU 和内存资源等信息。此外，还可以查阅集群任务的执行历史，为管理员提供了更全面的整个系统监控能力。实际上，任何需要对计算资源进行动态分配和管理的大数据环境中，算力集群任务监控均发挥着关键的作用。例如在批处理作业性能优化、实时流处理资源管理等使用场景中，集群计算任务监控在诸多处理场景下都很重要。

批处理作业场景中，企业运行了大量的批处理作业（数据清洗、转换和分析等），通过监控可以实时追踪每个作业的执行情况，包括资源利用、任务进度、任务执行时间等；可以发现哪些作业占用了大量资源，哪些作业运行较慢，进而进行针对性的调优，优化资源分配，提高整体计算效率。譬如通过调整任务的优先级、调度策略，可以更合理地利用计算资源，减少等待时间。

实时流处理场景下，用户基于 Flink 或 Spark Streaming 的应用，任务的资源需求可能会在短时间内快速波动。对于需要快速响应和实时计算的场景，任务监控可以实时了解每个流处理任务的资源占用情况，及时发现异常和瓶颈。譬如某个任务突然消耗了过多的内存，可能是由于数据倾斜或其他问题引起的，监控可以帮助迅速定位问题并采取相应的调整措施，以保证实时流处理系统的稳定性和性能。

4.12 负载均衡

支持基于 Nginx + Keepalived 实现四层/七层代理负载均衡配置管理服务。该功能提供一致性哈希和源地址哈希等六种负载均衡算法，确保流量能够被有效地分散到多个服务器上；用户可以根据实际需求，按照端口自定义负载配置。大数据分析计算任务涉及大量的数据传输和计算资源消耗，通过负载均衡可以避免单一节点负载过重的情况，提升集群资源整体利用率、为整真个集群性能提供保证；此外，负载均衡可以帮助实现大数据服务请求吞吐的灵活扩展，并确保大数据服务具备高可用性和容错性。通过负载均衡器，我们可以集中处理对外的请求，提高安全性。

某客户的大数据应用采用了多层次服务架构，包括基于 HUE 服务的 Web 业务数据开发和基于 IMPALA 的数据查询分析服务。这些服务部署在集群中不同的服务器上，每种服务都有独特的资源需求。为了更好地管理和优化这些服务的性能，我们为用户推荐了负载均衡器的方案。

通过负载均衡器，业务数据开发用户的请求被分发到不同的 HUE 服务器上，使每个用户的 session 得到保持，此外，这种方法还为每个用户提供了稳定的数据处理能力和带宽。对于 IMPALA 数据查询分析服务需求，负载均衡器确保了每个任务可获得足够的计算资源，保证了查询分析的性能和效率。

4.13 日志服务

日志服务通过收集、存储和分析大量的日志数据，可对整个集群所有运行的大数据服务组件的日志进行查看。用户可以根据需要，按照主机名、服务组件、实例、日志等级、自定义时间区间、日志内容关键字或 LogSql 等方式，灵活检索集群的日志数据。日志服务日志服务能够提供丰富的信息和洞察，是故障问题排查诊断、性能监控优化、安全审计监控等场景中很有用，可以很好的帮助企业提高系统稳定性、优化性能、确保安全性、降低成本等方面的重要支撑能力。

在大数据管理平台持续维护中，通常需要通过日志服务排查定位服务组件的

一些问题，包括服务日志错误与错误监控、服务异常分析、服务性能监控优化、用户行为分析等使用场景，集中式的查询分析日志，极大的加速分析问题、定位问题、处理问题的效率。

在大数据服务过程中，服务可能会面临不同级别的警告和错误，例如资源不足、连接超时等问题，日志服务可以实时采集服务产生的警告和错误日志，进行分类和分级，帮助团队快速发现潜在问题并采取及时措施，保障服务的稳定性。服务也可能因为一些未知原因异常退出，并因此导致服务中断、数据处理异常等问题，日志服务追踪服务的异常退出情况，记录退出的时间、原因以及可能的影响。通过分析这些异常退出日志，团队可以识别问题并改进服务，提高系统的可靠性。

4.14 事件

4.14.1 概述

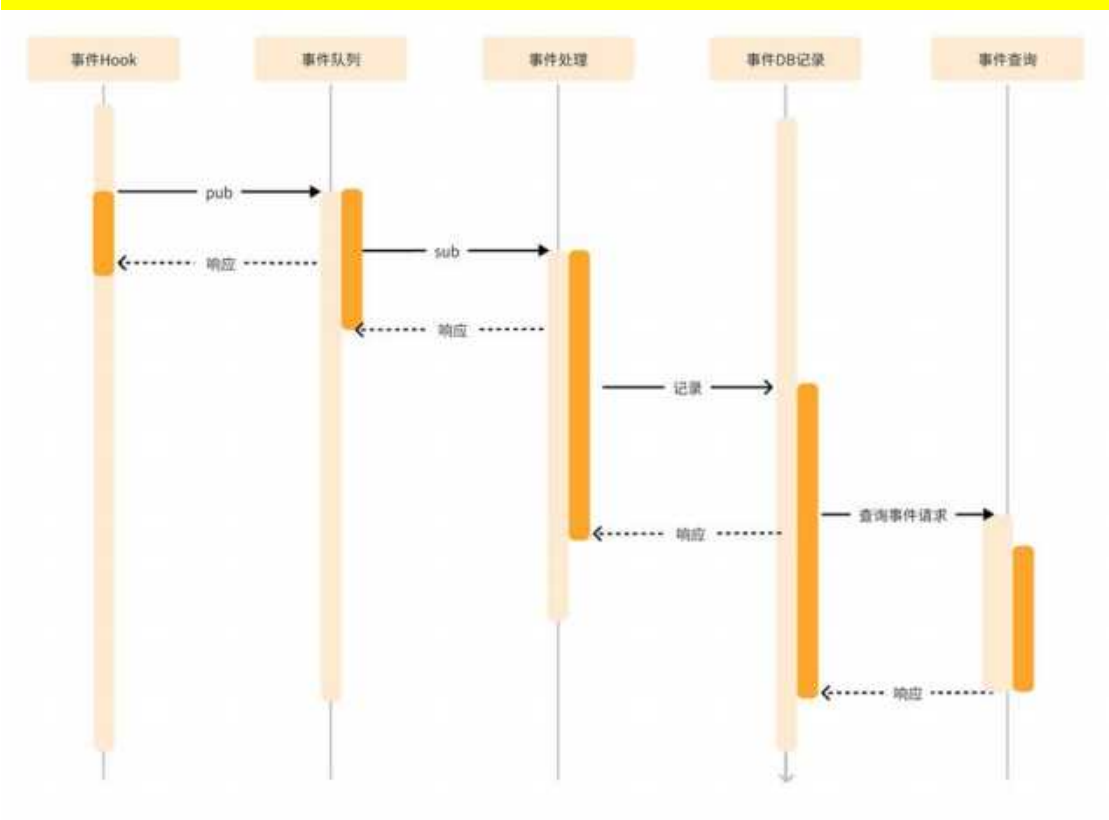
运维审计能力的扩展功能为管理员提供了更强大和全面的审计工具，用于监控和记录平台、集群和大数据服务的关键状态变化。新增的记录和展示功能，使管理员能够快速发现已发生的异常情况，并及时介入诊断和处理这些异常事件，确保系统的稳定和安全。

对 USDP 平台的服务运行过程的发生的关键信息以事件的方式进行记录、展示以及诊断。

在大数据平台的日常运维中，管理员可以通过审计功能追踪和记录平台、集群和各大数据服务的关键信息，如硬件资源异常检测、服务状态异常、集群资源紧缺、错误告警等，以便快速识别和解决潜在问题。操作日志审计功能可以帮助管理员跟踪和审查管理用户操作行为，提高安全性和合规性。

另外，在故障排除和系统故障恢复的场景中，运维审计功能可以提供有关事件发生的详细信息，辅助管理员进行诊断和处理，缩短故障恢复时间。

4.14.2 架构/特性

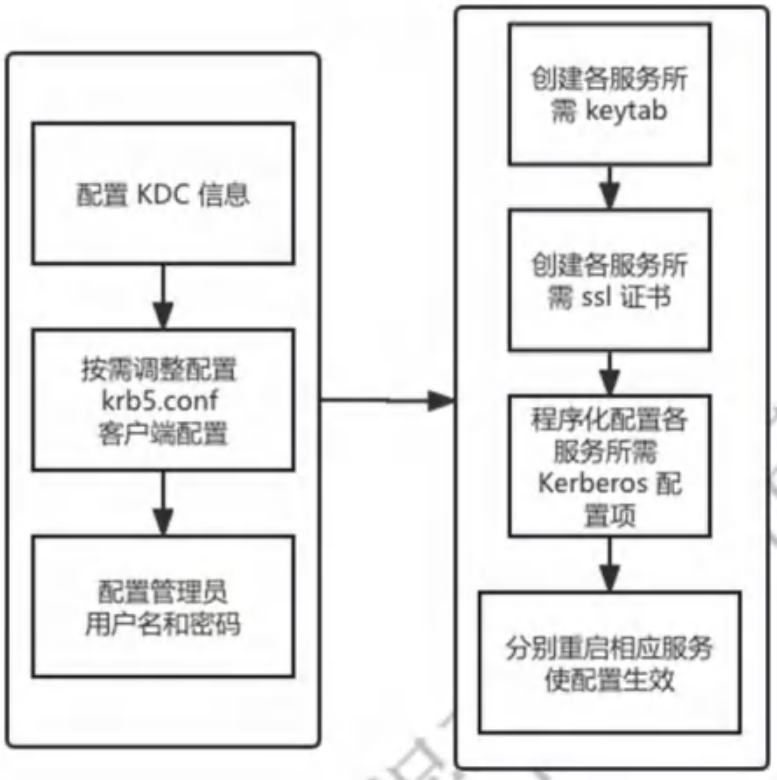


4.15 一键 Kerberos

4.15.1 概述

向导化生成相关 Kerberos 凭证配置和相应大数据组件的 Kerberos 配置。

4.15.2 使用流程

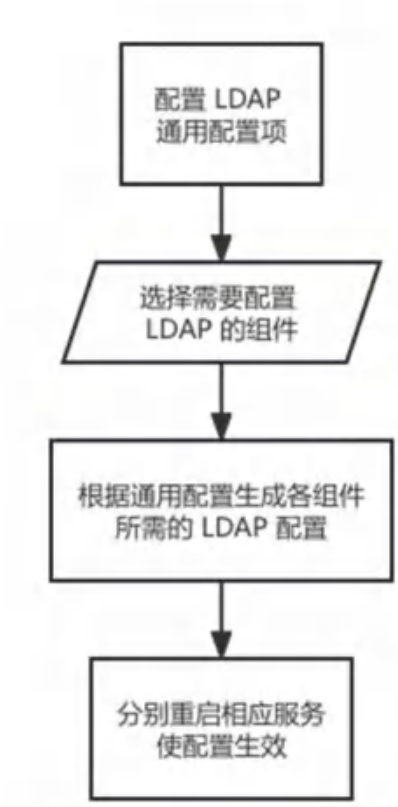


4.16 一键 LDAP

4.16.1 概述

向导化配置相应大数据组件的 LDAP 配置。

4.16.2 使用流程



5 生态服务组件

在当前 USDP UDH 发行版本中，支持的存储服务有 HDFS、Solr、ZooKeeper、Kafka、HBase、Phoenix 等，支持的计算框架有 YARN、Hive、Spark、Flink、Impala、Kylin、TEZ、Trino、等，支持的数据采集和同步工具有 Canal、DataX、Flume、Sqoop 等，支持的调度系统有 DolphinScheduler、Oozie，支持的元数据及安全控制相关服务有 Atlas、Ranger，支持的数据湖相关引擎有 Hudi、Iceberg，支持的可视化开发及管理工具有 Hue、EFAK、ZKUI、Zeppelin、Kerberos 可视化配置、LDAP 一键启用等，以及其他生态技术开发使用的服务 Livy、Kyuubi 等。这些服务的相关详细介绍及特性，将在下文逐一说明。

5.1 HDFS

5.1.1 概述

Hadoop 分布式文件系统 (HDFS) 是一种分布式文件系统。它与现有的分布式文件系统有很多相似之处。然而，与其他分布式文件系统的区别是显着的。HDFS 具有高度容错性，旨在部署在低成本硬件上。HDFS 提供对应用程序数据的高吞吐量访问，适用于具有大型数据集的应用程序。HDFS 放宽了一些 POSIX 要求以启用对文件系统数据的流式访问。HDFS 是 Apache Hadoop Core 项目的一部分。

5.1.2 设计目标

- 硬件故障

硬件故障是常态而不是例外。一个 HDFS 实例可能由成百上千台服务器机器组成，每台机器都存储文件系统数据的一部分。事实上，有大量的组件，而且每个组件都有很大的故障概率，这意味着 HDFS 的某些组件总是无法正常工作。因此，检测故障并从中快速自动恢复是 HDFS 的核心架构目标。

- 流式数据访问

在 HDFS 上运行的应用程序需要对其数据集进行流式访问。它们不是通常

在通用文件系统上运行的通用应用程序。HDFS 更多的是为批处理而不是用户交互使用而设计的。重点是数据访问的高吞吐量而不是数据访问的低延迟。POSIX 强加了许多针对 HDFS 的应用程序不需要的硬性要求。一些关键领域的 POSIX 语义已经被交换以提高数据吞吐率。

- 大数据集

在 HDFS 上运行的应用程序具有大型数据集。HDFS 中的典型文件大小为 GB 到 TB。因此，HDFS 被调整为支持大文件。它应该为单个集群中的数百个节点提供高聚合数据带宽和规模。它应该在单个实例中支持数千万个文件。

- 简单一致性模型

HDFS 应用程序需要一种一次写入多次读取的文件访问模型。文件一旦创建、写入和关闭，除了追加和截断之外不需要更改。支持将内容附加到文件末尾，但不能在任意点更新。这种假设简化了数据一致性问题并实现了高吞吐量数据访问。MapReduce 应用程序或网络爬虫应用程序非常适合此模型。

- 移动计算比移动数据更便宜

如果应用程序请求的计算是在它所操作的数据附近执行的，那么它的效率要高得多。当数据集的规模很大时尤其如此。这最大限度地减少了网络拥塞并增加了系统的整体吞吐量。假设是将计算迁移到更靠近数据所在的位置而不是将数据移动到应用程序运行的位置通常更好。HDFS 为应用程序提供接口，使它们更靠近数据所在的位置。

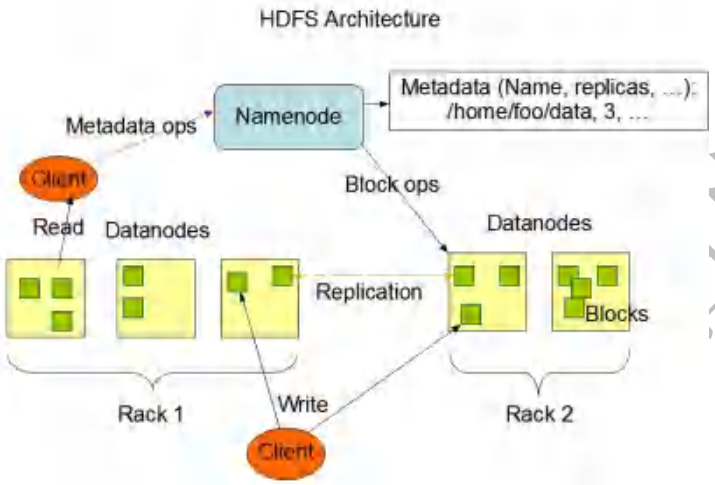
- 跨异构硬件和软件平台的可移植性

HDFS 被设计成可以轻松地从一个平台移植到另一个平台。这有助于广泛采用 HDFS 作为大量应用程序的首选平台。

5.1.3 服务架构

HDFS 具有主/从架构。一个 HDFS 集群由一个 NameNode 组成，一个管理文件系统命名空间并控制客户端对文件访问的主服务器。此外，还有许多数据节点，通常是集群中的每个节点一个，它们管理连接到它们运行的 节点的

存储。HDFS 公开了一个文件系统命名空间，并允许将用户数据存储在文件中。在内部，一个文件被分割成一个或多个块，这些块存储在一组数据节点中。NameNode 执行文件系统命名空间操作，如打开、关闭和重命名文件和目录。它还确定块到数据节点的映射。DataNode 负责处理来自文件系统客户端的读写请求。DataNodes 还执行块创建、删除。



NameNode 和 DataNode 是设计用于在商用机器上运行的软件。这些机器通常运行 GNU/Linux 操作系统 (OS)。HDFS 是使用 Java 语言构建的；任何支持 Java 的机器都可以运行 NameNode 或 DataNode 软件。使用高度可移植的 Java 语言意味着 HDFS 可以部署在各种机器上。典型的部署有一台只运行 NameNode 软件的专用机器。集群中的每台其他机器都运行一个 DataNode 软件实例。该架构不排除在同一台机器上运行多个 DataNode，但在实际部署中这种情况很少见。

集群中单个 NameNode 的存在大大简化了系统的架构。NameNode 是所有 HDFS 元数据的仲裁者和存储库。该系统的设计方式使用户数据永远不会流经 NameNode。

5.1.4 功能特性

- **文件系统命名空间**

HDFS 支持传统的分层文件组织。用户或应用程序可以创建目录并在这些目录中存储文件。文件系统命名空间层次结构与大多数其他现有文件系统相似；

可以创建和删除文件，将文件从一个目录移动到另一个目录，或重命名文件。HDFS 支持用户配额和访问权限。HDFS 不支持硬链接或软链接。但是，HDFS 体系结构并不排除实现这些功能。

虽然 HDFS 遵循 FileSystem 的命名约定，但保留了一些路径和名称（例如/.reserved 和）。透明加密和快照.snapshot 等功能使用保留路径。

NameNode 维护文件系统命名空间。对文件系统命名空间或其属性的任何更改都由 NameNode 记录。应用程序可以指定应由 HDFS 维护的文件的副本数。文件的副本数称为该文件的复制因子。此信息由 NameNode 存储。

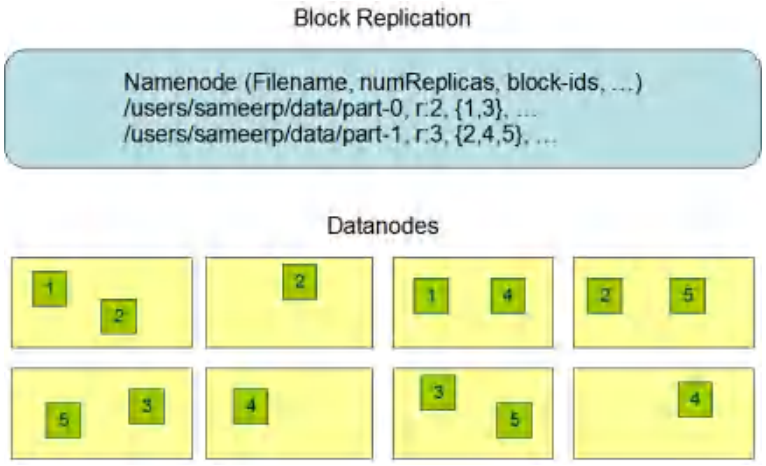
- **数据复制**

HDFS 旨在跨大型集群中的机器可靠地存储非常大的文件。它将每个文件存储为一系列块。复制文件的块以实现容错。每个文件的块大小和复制因子是可配置的。

一个文件中除最后一个块外的所有块都是相同的大小，而在附加和 hsync 中添加了对可变长度块的支持，用户可以在不将最后一个块填充到配置的块大小的情况下开始一个新块。

应用程序可以指定文件的副本数。复制因子可以在文件创建时指定，以后可以更改。HDFS 中的文件是一次性写入的（追加和截断除外）并且任何时候都只有一个写入者。

NameNode 做出关于块复制的所有决定。它定期从集群中的每个 DataNode 接收心跳和块报告。收到心跳意味着 DataNode 正常运行。Blockreport 包含 DataNode 上所有块的列表。



■副本放置：第一步

副本的放置对于 HDFS 的可靠性和性能至关重要。优化副本放置将 HDFS 与大多数其他分布式文件系统区分开来。这是一项需要大量调整和经验的功能。机架感知副本放置策略的目的是提高数据可靠性、可用性和网络带宽利用率。副本放置策略的当前实现是朝着这个方向的第一次努力。实施此策略的短期目标是在生产系统上对其进行验证，了解更多有关其行为的信息，并为测试和研究更复杂的策略奠定基础。

大型 HDFS 实例运行在通常分布在许多机架上的计算机集群上。不同机架中的两个节点之间的通信必须通过交换机。在大多数情况下，同一机架中机器之间的网络带宽大于不同机架中机器之间的网络带宽。

NameNode 通过 Hadoop Rack Awareness 中概述的过程确定每个 DataNode 所属的机架 ID 。一个简单但非最佳的策略是将副本放置在唯一的机架上。这可以防止在整个机架出现故障时丢失数据，并允许在读取数据时使用多个机架的带宽。该策略在集群中均匀分布副本，这使得在组件故障时平衡负载变得容易。但是，这种策略增加了写入的成本，因为一次写入需要将块传输到多个机架。

对于常见情况，当复制因子为 3 时，HDFS 的放置策略是，如果写入者在数据节点上，则将一个副本放在本地机器上，否则在与写入者相同机架的随机数据节点上，另一个副本在不同（远程）机架中的一个节点，以及同一远程机架中不同节点上的最后一个节点。该策略减少了通常提高写入性能的机架间写入流量。机架故障的几率远小于节点故障；此政策不影响数据可靠性和可用性保证。

但是，它不会减少读取数据时使用的聚合网络带宽，因为一个块仅放置在两个唯一的机架中，而不是三个。使用此策略，块的副本不会均匀分布在机架上。两个副本位于一个机架的不同节点上，其余副本位于其他机架之一的节点上。此策略可在不影响数据可靠性或读取性能的情况下提高写入性能。

如果复制因子大于 3，则随机确定第 4 个及后续副本的放置，同时保持每个机架的副本数低于上限（基本上是(replicas - 1) / racks + 2）。

因为 NameNode 不允许 DataNode 拥有同一个 block 的多个副本，所以创建的最大副本数就是当时 DataNode 的总数。

在将存储类型和存储策略的支持添加到 HDFS 之后，NameNode 除了上述的机架感知之外，还会考虑副本放置的策略。NameNode 首先根据机架感知选择节点，然后检查候选节点是否具有与文件关联的策略所需的存储。如果候选节点没有存储类型，NameNode 会寻找另一个节点。如果在第一条路径中找不到足够的节点来放置副本，NameNode 会在第二条路径中寻找具有回退存储类型的节点。

■副本选择

为了最小化全局带宽消耗和读取延迟，HDFS 尝试满足来自最接近读取器的副本的读取请求。如果在与读取节点相同的机架上存在一个副本，则优先选择该副本来满足读取请求。如果 HDFS 集群跨越多个数据中心，则驻留在本地数据中心的副本优于任何远程副本。

■块放置策略

如上所述，当复制因子为 3 时，HDFS 的放置策略是，如果写入者在数据节点上，则将一个副本放在本地机器上，否则在与写入者同一机架的随机数据节点上，另一个副本在节点上在不同的（远程）机架中，最后一个在同一远程机架中的不同节点上。如果复制因子大于 3，则随机确定第 4 个及后续副本的放置，同时保持每个机架的副本数低于上限（基本上是 (replicas - 1) / racks + 2）。除此之外，HDFS 还支持 4 种不同的可插入块放置策略。用户可以根据他们的基础设施和用例选择策略。默认情况下，HDFS 支持 BlockPlacementPolicyDefault。

■安全模式

启动时，NameNode 进入一种称为安全模式的特殊状态。当 NameNode 处于 Safemode 状态时，不会发生数据块的复制。NameNode 从 DataNode 接收 Heartbeat 和 Blockreport 消息。Blockreport 包含 DataNode 托管的数据块列表。每个块都有指定的最小副本数。当该数据块的最小数量的副本已通过 NameNode 签入时，该块被认为是安全复制的。在安全复制的数据块的可配置百分比向 NameNode 登记后（加上额外的 30 秒），NameNode 退出安全模式状态。然后它确定仍然少于指定副本数的数据块列表（如果有的话）。NameNode 然后将这些块复制到其他 DataNode。

HDFS 命名空间由 NameNode 存储。NameNode 使用称为 EditLog 的事务日志来持久记录文件系统元数据发生的每个更改。例如，在 HDFS 中创建一个新文件会导致 NameNode 在 EditLog 中插入一条记录来指示这一点。同样，更改文件的复制因子会导致将新记录插入到 EditLog 中。NameNode 使用其本地主机操作系统文件系统中的一个文件来存储 EditLog。整个文件系统命名空间，包括块到文件的映射和文件系统属性，都存储在一个名为 FsImage 的文件中。FsImage 也作为文件存储在 NameNode 的本地文件系统中。

NameNode 在内存中保存了整个文件系统命名空间和文件 Blockmap 的图像。当 NameNode 启动时，或者检查点由可配置的阈值触发时，它从磁盘读取 FsImage 和 EditLog，将 EditLog 中的所有事务应用到 FsImage 的内存表示，并将这个新版本刷新到一个磁盘上的新 FsImage。然后它可以截断旧的 EditLog，因为它的事务已应用于持久 FsImage。此过程称为检查点。检查点的目的是通过拍摄文件系统元数据的快照并将其保存到 FsImage 来确保 HDFS 具有文件系统元数据的一致视图。尽管读取 FsImage 是高效的，但直接对 FsImage 进行增量编辑并不高效。我们不会为每次编辑修改 FsImage，而是将编辑保存在 Editlog 中。在检查点期间，来自 Editlog 的更改将应用于 FsImage。检查点可以在给定的时间间隔触发 (dfs.namenode.checkpoint.period) 以秒表示，或者在给定数量的文件系统事务累积后（dfs.namenode.checkpoint.txns)。如果设置了这两个属性，则要达到的第一个阈值会触发检查点。

- **文件系统元数据的持久化**

DataNode 将 HDFS 数据存储在其本地文件系统中的文件中。DataNode 不知道 HDFS 文件。它将每个 HDFS 数据块存储在其本地文件系统中的单独文件中。DataNode 不会在同一目录中创建所有文件。相反，它使用启发式方法来确定每个目录的最佳文件数并适当地创建子目录。在同一目录中创建所有本地文件并不是最佳选择，因为本地文件系统可能无法有效地支持单个目录中的大量文件。当 DataNode 启动时，它会扫描其本地文件系统，生成与每个本地文件对应的所有 HDFS 数据块的列表，并将此报告发送给 NameNode。该报告称为块报告。

- **通信协议**

所有 HDFS 通信协议都位于 TCP/IP 协议之上。客户端与 NameNode 机器上的可配置 TCP 端口建立连接。它与 NameNode 对话 ClientProtocol。DataNodes 使用 DataNode 协议与 NameNode 对话。远程过程调用 (RPC) 抽象包装了客户端协议和 DataNode 协议。按照设计，NameNode 从不发起任何 RPC。相反，它只响应 DataNode 或客户端发出的 RPC 请求。

- **鲁棒性**

HDFS 的主要目标是即使在出现故障时也能可靠地存储数据。三种常见的故障类型是 NameNode 故障、DataNode 故障和网络分区。

■数据盘故障、心跳和重新复制

每个 DataNode 周期性地向 NameNode 发送 Heartbeat 消息。网络分区可能导致 DataNode 的子集失去与 NameNode 的连接。NameNode 通过心跳消息的缺失检测到这种情况。NameNode 将最近没有心跳的 DataNode 标记为已死，并且不会向它们转发任何新的 IO 请求。任何注册到死 DataNode 的数据都不再可用于 HDFS。DataNode 死亡可能导致某些块的复制因子低于其指定值。NameNode 不断跟踪哪些块需要复制，并在必要时启动复制。由于多种原因，可能会出现重新复制的必要性：DataNode 可能变得不可用，副本可能损坏，DataNode 上的硬盘可能发生故障，

标记 DataNodes 死亡的超时时间比较保守（默认超过 10 分钟），以避免 DataNodes 状态波动引起的复制风暴。用户可以设置较短的间隔以将 DataNode 标记为陈旧，并通过配置为性能敏感的工作负载避免陈旧节点的读取和/或写入。

■集群再平衡

HDFS 架构与数据重新平衡方案兼容。如果 DataNode 上的可用空间低于某个阈值，方案可能会自动将数据从一个 DataNode 移动到另一个 DataNode。在对特定文件的突然高需求的情况下，方案可能会动态创建额外的副本并重新平衡集群中的其他数据。这些类型的数据重新平衡方案尚未实施。

■数据的完整性

从 DataNode 获取的数据块到达时可能已损坏。由于存储设备故障、网络故障或有缺陷的软件，可能会发生这种损坏。HDFS 客户端软件对 HDFS 文件内容进行校验和校验。当客户端创建 HDFS 文件时，它会计算文件每个块的校验和，并将这些校验和存储在同一 HDFS 命名空间中的单独隐藏文件中。当客户端检索文件内容时，它会验证从每个 DataNode 接收到的数据是否与存储在相关校验和文件中的校验和相匹配。如果不是，则客户端可以选择从另一个具有该块副本的 DataNode 检索该块。

■元数据磁盘故障

FsImage 和 EditLog 是 HDFS 的中心数据结构。这些文件的损坏会导致 HDFS 实例无法运行。为此，NameNode 可以配置为支持维护 FsImage 和 EditLog 的多个副本。对 FsImage 或 EditLog 的任何更新都会导致每个 FsImages 和 EditLogs 同步更新。FsImage 和 EditLog 的多个副本的这种同步更新可能会降低 NameNode 可以支持的每秒命名空间事务的速率。然而，这种降级是可以接受的，因为即使 HDFS 应用程序本质上是数据密集型的，但它们不是元数据密集型的。当一个 NameNode 重启时，它会选择最新一致的 FsImage 和 EditLog 来使用。

提高故障恢复能力的另一种选择是使用多个 NameNode 启用高可用性，或

者使用 NFS 上的共享存储，或者使用分布式编辑日志（称为日志）。后者是推荐的方法。

■快照

快照支持在特定时刻存储数据副本。快照功能的一种用途可能是将损坏的 HDFS 实例回滚到以前已知的良好时间点。

- **数据组织**

■数据块

HDFS 旨在支持非常大的文件。与 HDFS 兼容的应用程序是那些处理大型数据集的应用程序。这些应用程序只写入一次数据，但会读取一次或多次，并要求以流式传输速度满足这些读取。HDFS 支持文件的一次写入多次读取语义。HDFS 使用的典型块大小为 128 MB。因此，一个 HDFS 文件被分割成 128 MB 的块，如果可能，每个块将驻留在不同的 DataNode 上。

■复制流水线

当客户端将数据写入复制因子为 3 的 HDFS 文件时，NameNode 使用复制目标选择算法检索 DataNode 列表。此列表包含将托管该块副本的 DataNode。客户端然后写入第一个 DataNode。第一个 DataNode 开始接收部分数据，将每个部分写入其本地存储库并将该部分传输到列表中的第二个 DataNode。第二个 DataNode 依次开始接收数据块的每个部分，将该部分写入其存储库，然后将该部分刷新到第三个 DataNode。最后，第三个 DataNode 将数据写入其本地存储库。因此，DataNode 可以从管道中的前一个节点接收数据，同时将数据转发到管道中的下一个节点。

- **辅助功能**

可以通过许多不同的方式从应用程序访问 HDFS。HDFS 本身提供了一个 FileSystem Java API 供应用程序使用。此 Java API 和 REST API 的 C 语言包装器也可用。此外，HTTP 浏览器也可用于浏览 HDFS 实例的文件。通过使用 NFS 网关，HDFS 可以作为客户端本地文件系统的一部分进行挂载。

■Shell

HDFS 允许以文件和目录的形式组织用户数据。它提供了一个名为 FS shell 的命令行界面，允许用户与 HDFS 中的数据进行交互。该命令集的语法类似于用户已经熟悉的其他 shell（例如 bash、csh）。FS shell 适用于需要脚本语言与存储数据交互的应用程序。

■DFS 管理员

DFSAdmin 命令集仅供 HDFS 管理员用于管理 HDFS 集群。如将集群将集群置于安全模式、生成 DataNode 列表、重新启用或停用 DataNode(s)等。

■浏览器界面

典型的 HDFS 安装配置 Web 服务器以通过可配置的 TCP 端口公开 HDFS 名称空间。这允许用户导航 HDFS 名称空间并使用 Web 浏览器查看其文件的内容。

- 空间回收

■文件删除和取消删除

如果启用垃圾配置，FS Shell 删除的文件不会立即从 HDFS 中删除。相反，HDFS 将其移动到垃圾目录（每个用户在 下都有自己的垃圾目录 /user/<username>/.Trash）。只要文件保留在垃圾箱中，就可以快速恢复该文件。

最近删除的文件被移动到当前垃圾目录（/user/<username>/.Trash/Current），并且在可配置的时间间隔内，HDFS/user/<username>/.Trash/<date>为当前垃圾目录中的文件创建检查点（在）下，并在旧检查点过期时删除它们。有关垃圾检查点的信息，请参见 FS shell 的 expunge 命令。

在其在垃圾桶中的生命期满后，NameNode 将从 HDFS 命名空间中删除该文件。删除文件会导致与该文件关联的块被释放。请注意，在用户删除文件的时间与 HDFS 中可用空间相应增加的时间之间可能存在明显的时间延迟。

■减少复制因子

当一个文件的复制因子降低时, NameNode 会选择多余的可以删除的副本。下一个 Heartbeat 将此信息传输到 DataNode。DataNode 随后移除相应的块，集群中出现相应的空闲空间。同样，在完成 setReplication API 调用和集群中出现可用空间之间可能存在时间延迟。

5.2 YARN

5.2.1 概述

Hadoop YARN （Yet Another Resource Negotiator）是一种 Hadoop 资源管理器，用户分布式作业提交/执行引擎，它通用资源管理系统和调度平台，允许远程调用者向集群提交任意工作，为上层应用提供统一的资源管理和调度，它的引入为集群在利用率、资源统一管理和数据共享等方面带来了巨大好处。

Hadoop YARN 支持在用户应用容器分配并使用 GPU 资源, 完成图像处理、机器学习和深度学习等计算密集型任务得到加速场景，在部署 YARN 集群时需在每个 NodeManager 所在节点上预装 Nvidia GPU 驱动程序，以确保节点能够使用 GPU 资源，ResourceManager 负责通过容量调度（Capacity Scheduler）的队列模式分配 GPU 资源。当处理用户提交的应用程序时，支持 YARN 应用程序请求特定 GPU 数量，并根据资源可用性自动调度应用程序至拥有所需资源的节点上运行。NodeManager 负责管理节点上 GPU 资源，包括资源的分配和监控。其通过 CGroup 技术实现对 GPU 卡之间的隔离控制，避免资源冲突和应用性能。

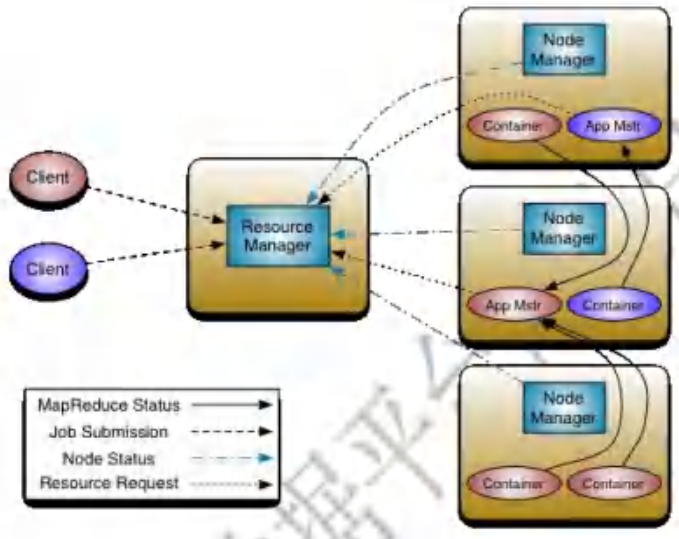
5.2.2 服务架构

YARN 的基本思想是将资源管理和作业调度/监控的功能拆分为单独的守护进程。因此 YARN 拥有一个全局 ResourceManager（RM）和每个应用程序 ApplicationMaster（AM）。应用程序可以是单个作业，也可以是作业的 DAG。

ResourceManager 和 NodeManager 构成了数据计算框架。

ResourceManager 是在系统中所有应用程序之间仲裁资源的最终权威。NodeManager 是每台机器的框架代理，负责容器、监控它们的资源使用情况（cpu、内存、磁盘、网络）并将其报告给 ResourceManager/Scheduler。

每个应用程序的 ApplicationMaster 实际上是一个特定于框架的库，其任务是从 ResourceManager 协商资源并与 NodeManager(s) 一起执行和监视任务。



ResourceManager 有两个主要组件：Scheduler 和 ApplicationsManager。

Scheduler 负责根据熟悉的容量、队列等约束将资源分配给各种正在运行的应用程序。从某种意义上说，Scheduler 是纯粹的调度程序，它不执行应用程序状态的监视或跟踪。此外，它不保证由于应用程序故障或硬件故障而重新启动失败的任务。Scheduler 根据应用程序的资源需求执行其调度功能；它基于资源容器的抽象概念来实现，该容器包含内存、cpu、磁盘、网络等元素。

调度器有一个可插入的策略，负责在各种队列、应用程序等之间划分集群资源。当前的调度器，如 CapacityScheduler 和 FairScheduler 将是插件的一些例子。

ApplicationsManager 负责接受作业提交，协商第一个容器来执行特定于应用程序的 ApplicationMaster，并提供在失败时重新启动 ApplicationMaster 容器的服务。每个应用程序的 ApplicationMaster 负责从调度程序协商适当的资源

容器，跟踪它们的状态并监控进度。

hadoop-2.x 中的 MapReduce 保持与之前稳定版本 (hadoop-1.x) 的 API 兼容性。这意味着所有 MapReduce 作业仍应在 YARN 之上运行，只需重新编译即可。

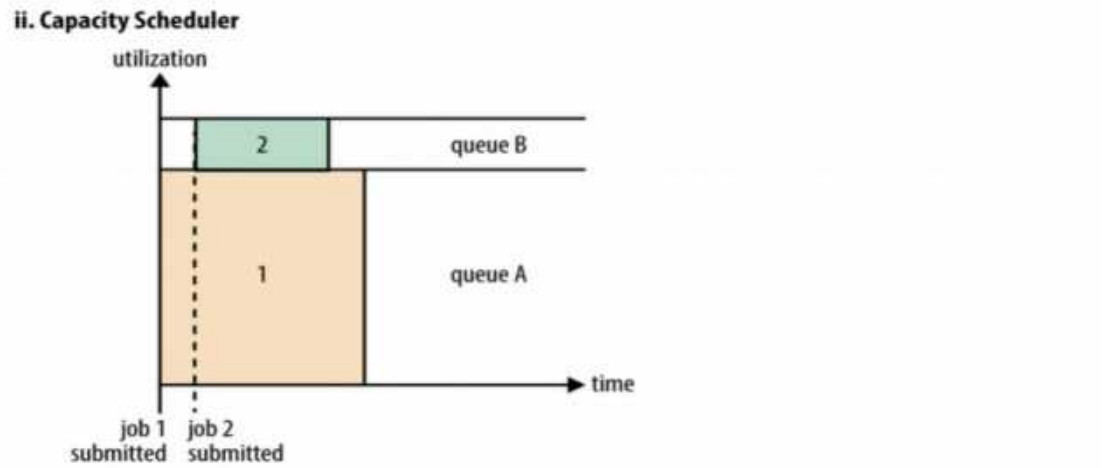
YARN 支持通过 ReservationSystem 进行资源预留的概念，ReservationSystem 是一个组件，允许用户指定资源随时间和时间限制（例如，截止日期）的配置文件，并预留资源以确保重要作业的可预测执行。ReservationSystem 跟踪资源超时，对预订执行准入控制，并动态指示底层调度程序以确保完成预订。

为了将 YARN 扩展到数千个节点之外，YARN 通过 YARN Federation 功能支持 Federation 的概念。Federation 允许透明地将多个 YARN（子）集群连接在一起，并使它们看起来像一个巨大的集群。这可用于实现更大的规模，允许多个独立的集群一起用于非常大的作业，或者用于拥有所有集群容量的租户。

5.2.3 调度器

5.2.3.1 CapacityScheduler

容量调度器 CapacityScheduler 用于 Hadoop 的可插拔调度程序，它允许多租户安全地共享一个大型集群，以便他们的应用程序在分配容量的约束下及时分配资源。同时最大限度地提高集群的吞吐量和利用率。



CapacityScheduler 旨在允许共享大型集群，同时为每个组织提供容量保证。核心思想是实现由多个组织共享 Hadoop 集群中的可用资源，即组织可以访问其他人未使用的任何多余容量。这为用户带来友好的成本效益，以及为组织提供了资源弹性保障。

跨组织共享集群需要对多租户的强大支持，因为每个组织都必须保证容量和安全防护，以确保共享集群不受单个恶意应用程序或用户或其集合的影响。CapacityScheduler 提供了一组严格的限制，以确保单个应用程序或用户或队列不会消耗集群中不成比例的资源量。此外，CapacityScheduler 还对来自单个用户和队列的已初始化和待处理应　　用程序进行了限制，以确保集群的公平性和稳定性。

CapacityScheduler 提供的主要抽象是队列的概念。通常是由管理员配置并管理队列，以反映共享集群的经济性。为了提供对资源共享的进一步控制和可预测性，CapacityScheduler 支持分层队列以确保在允许其他队列使用空闲资源之前在组织的子队列之间共享资源，从而为给定应用程序之间共享空闲资源提供亲和力组织。

CapacityScheduler 支持以下功能：

- 分层队列 - 支持队列的层次结构，以确保在允许其他队列使用空闲资源之前在组织的子队列之间共享资源，从而提供更多的控制和可预测性。
- 容量保证 - 队列被分配了网格容量的一小部分，在某种意义上，它们可以支配一定的资源容量。提交给队列的所有应用程序都可以访问分配给队列的容量。管理员可以对分配给每个队列的容量配置软限制和可选的硬限制。
- 安全性 - 每个队列都有严格的 ACL，控制哪些用户可以将应用程序提交到各个队列。此外，还有一些安全措施可确保用户无法查看和/或修改其他用户的应用程序。此外，还支持每个队列和系统管理员角色。
- 弹性 - 可以将免费资源分配给超出其容量的任何队列。当在未来某个时间点运行低于容量的队列对这些资源有需求时，随着这些资源上调度的

任务完成，它们将被分配给运行低于容量的队列上的应用程序（也支持抢占）。这确保资源以可预测和弹性的方式提供给队列，从而防止集群中的人为资源孤岛，这有助于利用。

- 多租户 - 提供了一套全面的限制，以防止单个应用程序、用户和队列独占队列或整个集群的资源，以确保集群不会不堪重负。

- 可操作性

  ■运行时配置 - 队列定义和属性（如容量、ACL）可以在运行时由管理员以安全的方式更改，以最大限度地减少对用户的干扰。此外，还为用户和管理员提供了一个控制台，用于查看系统中各个队列的当前资源分配情况。管理员可以在运行时添加额外的队列，但不能在运行时删除队列，除非队列已停止并且没有待处理/正在运行的应用程序。

  ■排空应用程序 - 管理员可以在运行时停止队列，以确保在现有应用程序运行完成时，不能提交新的应用程序。如果队列处于 STOPPED 状态，则不能将新应用程序提交给它自己或其任何子队列。现有的应用程序继续完成，因此队列可以优雅地排空。管理员还可以启动已停止的队列。

- 基于资源的调度 - 支持资源密集型应用程序，其中应用程序可以选择指定比默认值更高的资源要求，从而适应具有不同资源要求的应用程序。目前，内存是支持的资源需求。

- 基于默认或用户定义放置规则的队列映射接口 - 此功能允许用户根据一些默认放置规则将作业映射到特定队列。例如基于用户和组，或应用程序名称。用户还可以定义自己的放置规则。

- 优先级调度 - 此功能允许应用程序以不同的优先级提交和调度。较高的整数值表示应用程序的优先级较高。当前仅 FIFO 排序策略支持应用程序优先级。

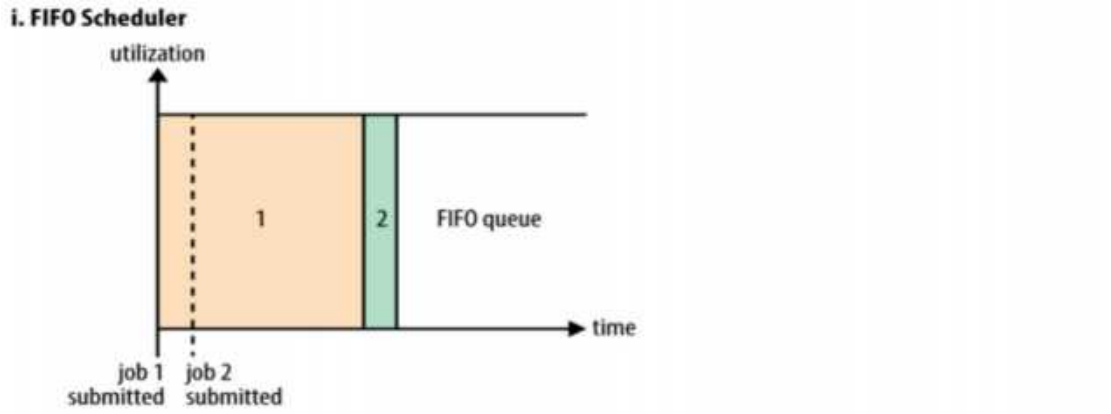
- 绝对资源配置 - 管理员可以为队列指定绝对资源，而不是提供基于百分

比的值。这为管理员提供了更好的控制，可以为给定队列配置所需的资源量。

- 叶队列的动态自动创建和管理 - 此功能支持叶队列的自动创建与队列映射相结合，队列映射目前支持基于用户组的队列映射以将应用程序放置到队列中。调度程序还支持基于在父队列上配置的策略对这些队列进行容量管理。

5.2.3.2 FairScheduler

公平调度器 FairScheduler 用于 Hadoop 的可插拔调度程序，这种资源分配方法允许 YARN 应用程序在大型集群中公平地共享集群资源，使所有应用程序在一段时间内平均获得相等的资源份额。



默认情况下，FairScheduler 仅根据内存做出调度公平性决策，亦可配置为同时使用内存和 CPU 进行调度。当有一个应用程序在运行时，该应用程序会使用整个集群。当提交其他应用程序时，释放的资源将分配给新应用程序，以便每个应用程序最终获得大致相同数量的资源。公平共享还可以与应用程序优先级一起使用 - 优先级用作权重来确定每个应用程序应获得的总资源的比例。

FairScheduler 中所有用户共享一个名为“default”的默认队列，若应用程序在容器资源请求中指定了队列，则该请求将提交到指定队列中，也可以通过配置根据请求中包含的用户名分配队列。在每个队列中，调度策略用于在运行的应用程序之间共享资源，默认是基于内存的公平共享，但也可以配置 FIFO 先进先

出和具有优势资源公平的多资源。队列可以分层排列以划分资源，并配置权重以按特定比例共享集群。

除了提供公平共享外，FairScheduler 还允许为队列分配有保证的最小份额，这将确保某些用户、组或生产应用程序始终获得足够的资源。当对列包含应用程序时，它至少会获得其最小份额，但当队列不需要其完全保证的份额时，超出部分将分配给其他正在运行的应用程序。这让调度程序可以保证队列的容量，同时在这些队列不包含应用程序时有效地利用资源。

FairScheduler 允许所有应用默认运行，但也可以通过配置文件限制每个用户和每个队列运行的应用数量。这在用户必须一次提交数百个应用程序时非常有用，或者如果一次运行太多应用程序，会导致创建过多的中间数据或过多的上下文切换，因此，限制应用数量通常可以达到提高性能的作用。限制应用程序不会导致任何后续提交的应用程序失败，只会在调度程序的队列中等待，直到用户较早的一些应用程序完成。

- **具有可插入策略的分层队列**

公平调度器支持分层队列。所有队列都来自名为“root”的队列。可用资源以典型的公平调度方式分配给根队列的子队列。然后，子队列将分配给他们的资源以同样的方式分配给他们的子队列。应用程序只能安排在叶队列上。通过将队列作为其父队列的子元素放置在公平调度程序分配文件中，也可以将队列指定为其他队列的子队列。

队列的名称以其父级的名称开头，以句点作为分隔符。因此，根队列下名为“queue1”的队列将被称为“root.queue1”，名为“parent1”的队列下名为“queue2”的队列将被称为“root.parent1.queue2”。引用队列时，名称的根部分是可选的，因此 queue1 可以仅称为“queue1”，而 queue2 可以仅称为“parent1.queue2”。

此外，公平调度程序允许为每个队列设置不同的自定义策略，以允许以用户想要的任何方式共享队列的资源。可以通过扩展构建自定义策略 org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.SchedulingP

olicy。FifoPolicy、FairSharePolicy（默认）和 DominantResourceFairnessPolicy 是内置的，可以很容易地使用。

- **自动将应用程序放入队列**

FairScheduler 允许管理员配置策略，自动将提交的应用程序放入适当的队列中。放置可以取决于提交者的用户和组以及应用程序传递的请求队列。策略由一组规则组成，这些规则按顺序应用以对传入的应用程序进行分类。有关如何配置这些策略。

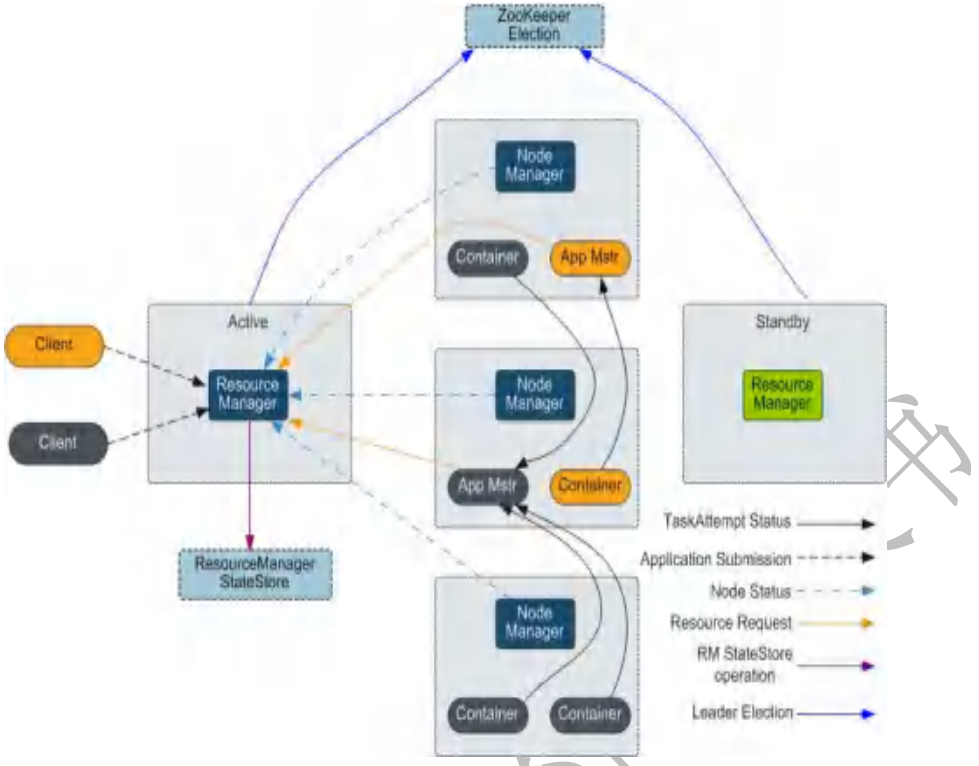
5.3 MapReduce

5.3.1 简介

MapReduce 是一种简化并行计算的编程模型，名字源于该模型中的两项核心操作：Map 和 Reduce。Map 将一个作业分解成为多个任务，Reduce 将分解后多个任务处理的结果汇总起来，得出最终的分析结果。

5.3.2 结构

如下图所示，MapReduce 通过实现 YARN 的 Client 和 ApplicationMaster 接口集成到 YARN 中，利用 YARN 申请计算所需资源。



5.3.3 与组件的关系

MapReduce 和 HDFS 的配合关系

- HDFS 是 Hadoop 分布式文件系统，具有高容错和高吞吐量的特性，可以部署在价格低廉的硬件上，存储应用程序的数据，适合有超大数据集的应用程序。
- 而 MapReduce 是一种编程模型，用于大数据集（大于 1TB）的并行运算。在 MapReduce 程序中计算的数据可以来自多个数据源，如 Local FileSystem、HDFS、数据库等。最常用的是 HDFS，可以利用 HDFS 的高吞吐性能读取大规模的数据进行计算。同时在计算完成后，也可以将数据存储到 HDFS。

MapReduce 和 YARN 的配合关系

MapReduce 是运行在 YARN 之上的一个批处理的计算框架。MRv1 是 Hadoop 1.0 中的 MapReduce 实现，它由编程模型（新旧编程接口）、运行时环境（由 JobTracker 和 TaskTracker 组成）和数据处理引擎（MapTask 和 ReduceTask）三部分组成。该框架在扩展性、容错性（JobTracker 单点）和多

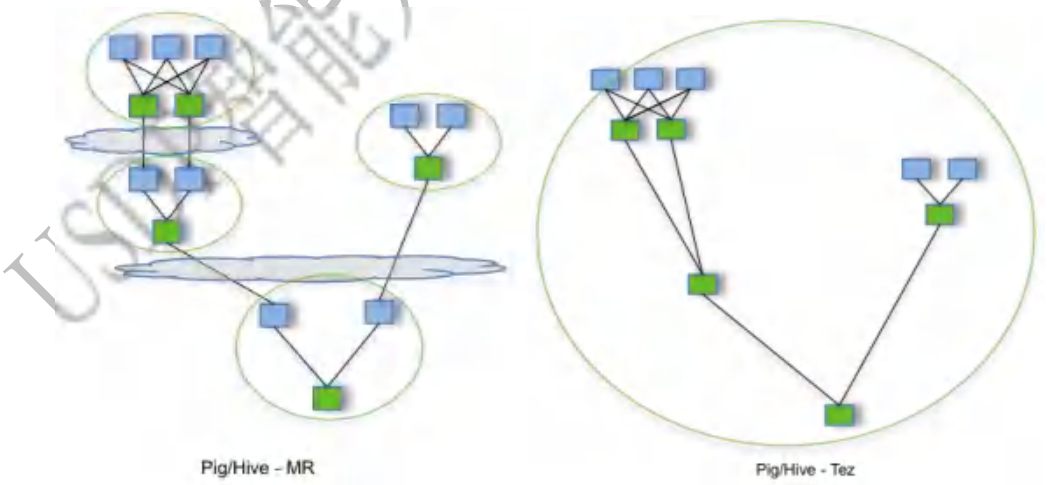
框架支持(仅支持MapReduce一种计算框架)等方面存在不足。MRv2是Hadoop 2.0中的MapReduce实现，它在源码级重用了MRv1的编程模型和数据处理引擎实现，但运行时环境由YARN的ResourceManager和ApplicationMaster组成。其中ResourceManager是一个全新的资源管理系统，而ApplicationMaster则负责MapReduce作业的数据切分、任务划分、资源申请和任务调度与容错等工作。

5.4 TEZ 优化 MapReduce 的 DAG

5.4.1 概述

TEZ是Apache开源的构建在Yarn之上、支持DAG作业的计算程序框架，它直接源于MapReduce框架，核心思想是将Map和Reduce两个操作进一步拆分，即Map被拆分成Input、Processor、Sort、Merge和Output， Reduce被拆分成Input、Shuffle、Sort、Merge、Processor和Output等，这些分解后的元操作可以任意灵活组合，经过控制程序组装后，形成一个复杂的有向无环图（DAG）任务来处理数据。

5.4.2 服务架构



允许 Hive项目运行复杂的 DAG 任务，Tez 可用于高效分析处理数据中，相较于之前需要经过多个 MR 作业，而现在仅需在单个 Tez 作业中即可完成计算。

左侧示意图的流程中包含多个 MR 任务，每个任务运行结束时，都需将中间结果存储到 HDFS 上，而前一任务的 reducer 输出，为下一任务的 mapper 提供数据。频繁读写 HDFS，使整个计算任务变得很低效。

右侧示意图是使用 Tez 作业时的流程，较左侧示意图的任务流程而言，仅在一个任务中就能完成同样的处理过程，子任务之间不需要再频繁访问 HDFS，整体简化了处理流程，降低资源开销的同时，极大的提供了任务运行效率。

5.4.3 服务特性

TEZ 具有以下特点：

- 富有表现力的数据流定义 API
- 灵活的输入-处理器-输出运行时编程模型，对任务模型的灵活性有极大的提升
- 提供 container 复用机制与 Tez Session，减少资源消耗计划在运行时重新配置
- 动态物理数据流决策
- 避免中间数据写回 HDFS，减小任务执行时间

USDP 支持的 Hive 组件，已将 TEZ 为默认计算引擎。用户可根据需求自行调整 Hive 的计算引擎，如 Spark。

5.5 Hbase 分布式 NoSQL 数据库

5.5.1 概述

HBase 是一个高可靠性、高性能、可伸缩、面向列的分布式数据库存储系统，它利用 Hadoop HDFS 作为其文件存储系统，提供对大数据进行随机、实时读/写访问。HBase 不是一个关系型数据库，其设计目标是用来解决关系型数据库在处理海量数据时的理论和实现上的局限性，HBase 可在商用硬件集群上托管非常大的表（数十亿行 x 数百万列）。HBase 从一开始就是为 Terabyte 到

Petabyte 级别的海量数据存储和高速读写而设计，这些数据要求能够被分布在数千台普通服务器上，并且能够被大量并发用户高速访问。

5.5.2 服务架构

5.5.2.1 NoSQL

HBase 是一种“NoSQL”数据库。“NoSQL”是一个通用术语，意思是该数据库不是支持 SQL 作为其主要访问语言的 RDBMS。还有许多 NoSQL 类型的数据库，如 BerkeleyDB 就是本地 NoSQL 数据库的一个示例。而 HBase 是一个分布式数据库。从技术上讲，HBase 实际上更像是一个“数据存储”而不是“数据库”，因为它缺少 RDBMS 中的许多功能，例如类型化列、二级索引、触发器和高级查询语言等。

但 HBase 良好的支持了线性和模块化扩展的特性。HBase 集群通过添加托管在集群服务器上的 RegionServers 来扩展。例如，如果一个集群从 10 个 RegionServer 扩展到 20 个，它在存储和处理能力方面都会翻倍。RDBMS 只能扩展到一个节点，特别是单个数据库服务器的大小的制约性，而为了获得最佳性能，往往需要专门的硬件和存储设备做支撑。

5.5.2.2 HBase 和 Hadoop/HDFS 区别

HDFS 是一种分布式文件系统，非常适合存储大文件。然而它并不是通用文件系统，不提供文件中的快速单个记录查找。HBase 建立在 HDFS 之上，可为大型表提供快速记录查找（和更新）。有时这可能是概念上的混淆点。HBase 在内部将数据放在 HDFS 上存在的索引“StoreFiles”中以进行高速查找。

5.5.3 数据模型

在 HBase 中，数据存储在具有行和列的表中。这与关系数据库 (RDBMS) 的术语重叠，但这个类比并没有什么用。相反，将 HBase 表视为多维映射可能会更好一些。

HBase 有诸多数据模型术语：

Table：HBase 表由多行组成。

- **Row**：HBase 中的一行由一个行键和一个或多个列以及与之关联的值组成。行在存储时按行键的字母顺序排序。为此，行键的设计非常重要。目标是以相关行彼此靠近的方式存储数据。常见的行键模式是网站域。如果您的行键是域，您应该将它们反向存储（org.apache.www、org.apache.mail、org.apache.jira）。这样，所有 Apache 域在表中彼此靠近，而不是根据子域的第一个字母展开。
- **Column**：HBase 中的列由列族和列限定符组成，它们由:（冒号）字符分隔。
- **Column Famil**：通常出于性能原因，列族在物理上将一组列及其值放在一起。每个列族都有一组存储属性，例如它的值是否应该缓存在内存中，它的数据是如何压缩的，它的行键是如何编码的，等等。表中的每一行都有相同的列族，尽管给定的行可能不在给定的列族中存储任何内容。
- **Column Qualifier**：将列限定符添加到列族以提供给定数据片段的索引。给定一个列族 content，一个列限定符可能是 content:html，另一个可能是 content:pdf。尽管列族在创建表时是固定的，但列限定符是可变的，并且行之间可能有很大差异。
- **Cell**：是{row, column, version}的 cell 元组组合，包含一个值和一个时间戳，它表示值的版本。
- **Timestamp**：时间戳写在每个值旁边，是给定版本值的标识符。默认情况下，时间戳表示写入数据时 RegionServer 上的时间，用户也可在数据放入单元格时指定自定义的时间戳值。

5.5.4 应用场景

HBase 并不适合所有问题。

首先，需有大量的数据。如有数亿或数十亿行，那么 HBase 是一个不错的选择。如果数据量相对较小，如有几千/百万行，那么使用传统的 RDBMS 可能是更好的选择，因为所有数据可能会在一个（或两个）节点上结束，而集群的其余部分可能位于闲置的。

其次，在没有 RDBMS 提供的所有额外功能（例如，类型化列、二级索引、事务、高级查询语言等）的情况下能够满足需求。针对 RDBMS 构建的应用程序不能通过简单地更改“移植”到 HBase 例如，JDBC 驱动程序。而是考虑从 RDBMS 种将数据迁移到 HBase，这是业务架构完全重新设计，而非移植。

再次，需要规模化的硬件支持。即使是 HDFS 也不能很好地处理少于 3 个 DataNode（由于诸如 HDFS 块复制之类的默认值为 3）加上 NameNode。

5.5.5 服务特性

- 线性和模块化的可扩展性。
- 强一致性读/写：HBase 不是“最终一致”的 DataStore。这使得它非常适合高速计数器聚合等任务。
- 自动分片：HBase 表通过区域分布在集群上，随着数据的增长，区域会自动拆分和重新分布。
- 自动 RegionServer 故障转移
- Hadoop/HDFS 集成：HBase 开箱即用地支持 HDFS 作为其分布式文件系统。
- MapReduce：HBase 支持通过 MapReduce 进行大规模并行处理，将 HBase 用作源和接收器。
- Java 客户端 API：HBase 支持易于使用的 Java API 进行编程访问。
- Thrift/REST API：HBase 还支持非 Java 前端的 Thrift 和 REST。
- 块缓存和布隆过滤器：HBase 支持块缓存和布隆过滤器以优化大容量查询。

- 运营管理：HBase 为运营洞察力和 JMX 指标提供内置网页。

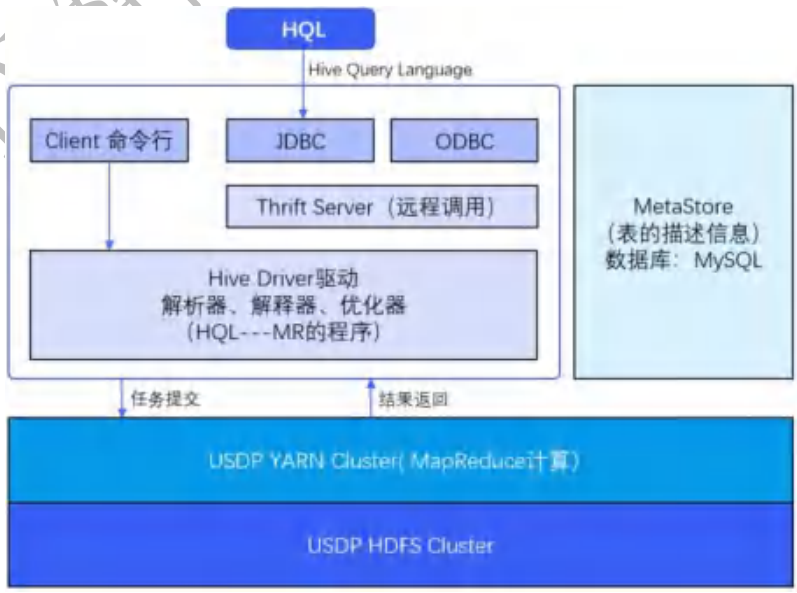
5.6 Hive 分布式数仓

5.6.1 概述

Hive 是建立在 Hadoop 上的一个分布式的、容错的数据仓库系统，它支持大规模数据的分析，并使用 SQL 方便地快速查询、写入和管理驻留在分布式存储中的 PB 级数据。它提供了一系列的工具，可以用来进行数据提取转化加载（ETL），这是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制。Hive 定义了简单的类 SQL 查询语言，称为 HQL，它允许熟悉 SQL 的用户查询数据。同时，这个语言也允许熟悉 MapReduce 开发者的开发自定义的 mapper 和 reducer 来处理内建的 mapper 和 reducer 无法完成的复杂的分析工作。

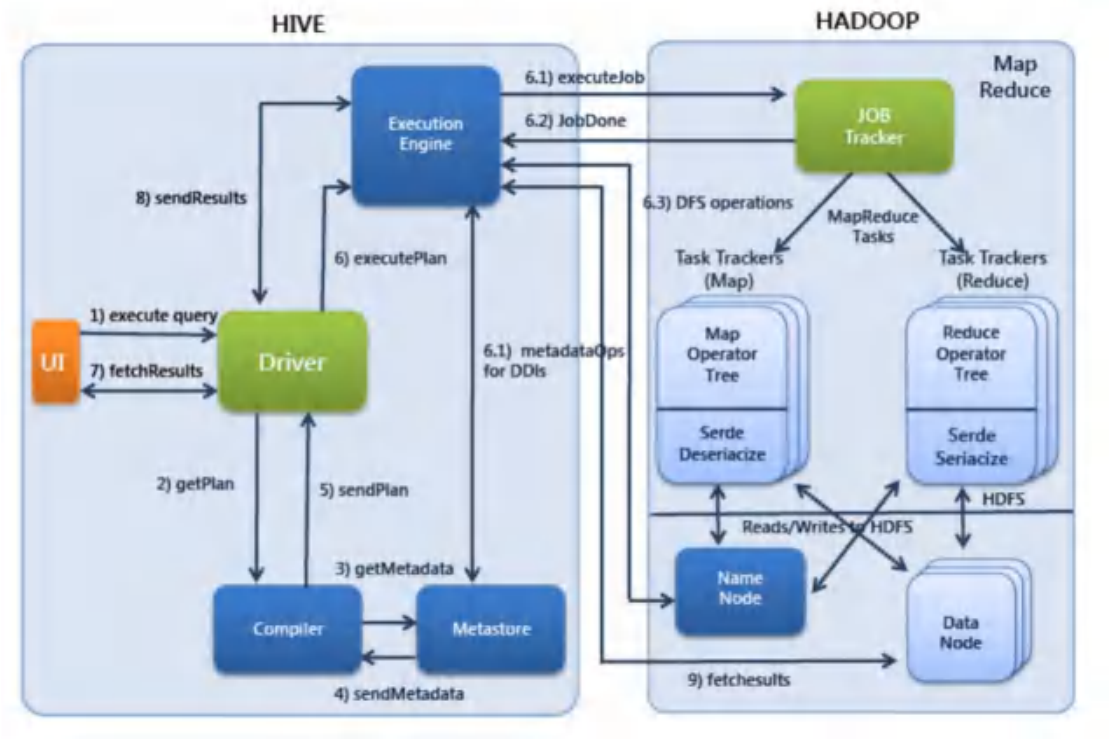
Hive Metastore(HMS)提供了一个元数据的中央存储库，可以很容易地对其进行分析，从而做出明智的、数据驱动的决策，因此它是许多数据湖架构的关键组件。除支持构建在 Hadoop 的 HDFS 文件系统外，Hive 还支持 S3、adls、gs 等存储系统。

5.6.2 服务架构



- HDFS：存储 Hive 仓库的数据文件；
- YARN：完成 Hive 的 HQL 转化的 MR 程序的执行；
- MetaStore：保存管理维护 Hive 的元数据；
- Hive：用来通过 HQL 的执行，转化为 MapReduce 程序的执行，从而对 HDFS 集群中的数据文件进行统计。

5.6.2.1 架构设计



如上图显示出 Hive 的主要组件，以及组件与 Hadoop 的交互逻辑。Hive 的主要组件是：

- **UI** - 用户向系统提交查询和其他操作的用户界面。截至 2011 年，该系统具有命令行界面，并且正在开发基于 Web 的 GUI。
- **驱动程序** - 接收查询的组件。该组件实现了会话句柄的概念，并提供了以 JDBC/ODBC 接口为模型的执行和获取 API。
- **编译器** - 解析查询的组件，对不同的查询块和查询表达式进行语义分析，并最终在从 Metastore 中查找的表和分区元数据的帮助下生成执

行计划。

- **Metastore** - 存储仓库中各种表和分区的所有结构信息的组件，包括列和列类型信息，读写数据所需的序列化器和反序列化器以及存储数据的相应 HDFS 文件。
- **执行引擎** - 执行编译器创建的执行计划的组件。该计划是阶段的 DAG。执行引擎管理计划的这些不同阶段之间的依赖关系，并在适当的系统组件上执行这些阶段。

上图还显示了典型的查询如何在系统中流动。UI 调用驱动程序的执行接口（上图中的步骤 1）。驱动程序为查询创建会话句柄并将查询发送给编译器以生成执行计划（步骤 2）。编译器从 Metastore 获取必要的元数据（步骤 3 和 4）。此元数据用于对查询树中的表达式进行类型检查，以及根据查询谓词修剪分区。编译器生成的计划（第 5 步）是一个阶段的 DAG，每个阶段要么是一个 map/reduce 作业，要么是一个元数据操作，要么是对 HDFS 的操作。对于 map/reduce 阶段，计划包含 map 运算符树（在映射器上执行的运算符树）和 reduce 运算符树（用于需要 reducer 的操作）。执行引擎将这些阶段提交给适当的组件（步骤 6、6.1、6.2 和 6.3）。在每个任务（映射器/缩减器）中，与表或中间输出关联的反序列化器用于从 HDFS 文件中读取行，这些行通过关联的运算符树传递。一旦生成输出，它就会通过序列化程序写入一个临时的 HDFS 文件（这发生在映射器中，以防操作不需要 reduce）。临时文件用于为计划的后续 map/reduce 阶段提供数据。对于 DML 操作，最终的临时文件被移动到表的位置。该方案用于确保不读取脏数据（文件重命名在 HDFS 中是原子操作）。

5.6.3 服务特性

- 支持通过 SQL 轻松访问数据的工具，来支撑数据仓库计算任务，例如提取/转换/加载 (ETL)、报告和数据分析。
- 一种在各种数据格式上施加结构的机制
- 访问直接存储在 HDFS 或其他数据存储系统（如 HBase）中的文件

- 通过 Tez、Spark 或 MapReduce 执行查询
- 使用 HPL-SQL 的过程语言
- 通过 Hive LLAP、Apache YARN 和 Apache Slider 进行亚秒级查询检索。

Hive 提供标准的 SQL 功能，包括许多用于分析的 SQL:2003、SQL:2011 和 SQL:2016 功能。

Hive 的 SQL 还可以通过用户定义的函数 (UDF)、用户定义的聚合 (UDAF) 和用户定义的表函数 (UDTF) 使用用户代码进行扩展。

Hive 并未限定必须使用单一的数据存储格式。Hive 支持带有用逗号和制表符分隔值（CSV/TSV）的文本文件、Parquet、ORC 和其他格式的内置连接器。用户可以使用其他格式的连接器扩展 Hive。

Hive 不是为联机事务处理 (OLTP) 工作负载设计的。它最适用于传统的数据仓库分析任务 (OLAP)。

Hive 旨在最大限度地提高可伸缩性（向外扩展，将更多机器动态添加到 Hadoop 集群）、性能、可扩展性、容错性以及与其输入格式的松散耦合。

5.6.4 主要特征

5.6.4.1 HiveServer2 (HS2)

HiveServer2 (HS2) 是一项使客户端能够对 Hive 执行查询的服务。它支持多客户端并发和认证，为 JDBC 和 ODBC 等开放 API 客户端提供更好的支持。HS2 提供基于 Thrift 的 Hive 服务（TCP 或 HTTP）和用于 Web UI 的 Jetty Web 服务器。

基于 Thrift 的 Hive 服务是 HS2 的核心，负责为 Hive 查询（例如，来自 Beeline）提供服务。Thrift 是一个用于构建跨平台服务的 RPC 框架。它的堆栈由 4 层组成：服务器层、传输层、协议层和处理器层。下面是这些层在 HS2 实现中的使用：

- **服务器层**：HS2 使用 TThreadPoolServer（来自 Thrift）用于 TCP 模式，或使用 Jetty 服务器用于 HTTP 模式。TThreadPoolServer 为每个 TCP 连接分配一个工作线程。每个线程总是与一个连接相关联，即使该连接是空闲的。因此存在大量并发连接导致线程数过多导致的潜在性能问题。

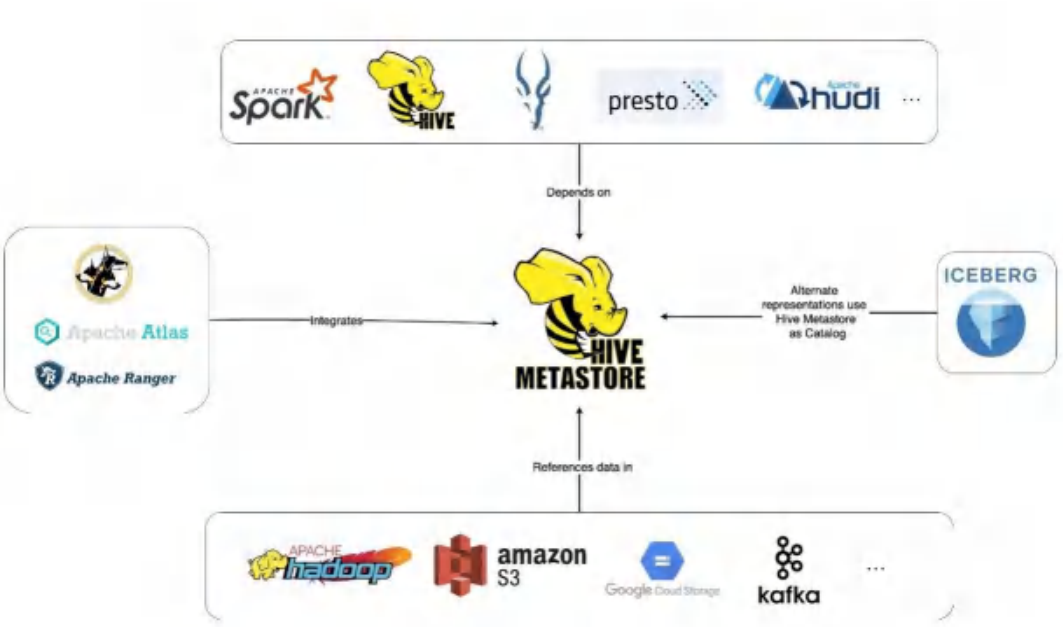
- **运输层**：当客户端和服务器之间需要代理时（例如，出于负载平衡或安全原因），需要 HTTP 模式。这就是支持它以及 TCP 模式的原因。可以通过 Hive 配置属性 hive.server2.transport.mode 指定 Thrift 服务的传输模式。

- **协议层**：协议实现负责序列化和反序列化。HS2 目前使用 TBinaryProtocol 作为其用于序列化的 Thrift 协议。将来可能会考虑基于更多性能评估的其他协议，例如 TCompactProtocol。

- **处理器层**：流程实现是处理请求的应用程序逻辑。例如，ThriftCLIService.ExecuteStatement() 方法实现了编译和执行 Hive 查询的逻辑。

5.6.4.2 Hive Metastore Server (HMS)

Hive Metastore (HMS)是关系数据库中Hive表和分区的元数据中央存储库，并通过 Metastore 服务 API 为客户端(包括 Hive、Impala 和 Spark 等)提供访问这些信息的服务。它已经成为数据湖的构建块之一，数据湖利用了各种开源软件，比如 Spark、Impala 和 Presto 等。在一个较完整的工具生态系统中，几乎都在围绕 Hive Metastore 构建的，如下图所示。



5.6.4.3 Hive ACID

Hive 为 ORC 表 out 和 insert 提供了完整的 ACID 支持。

ACID 代表了数据库事务的四个特征：

- 原子性 - 操作完全成功或失败，不会留下部分数据；
- 一致性 - 一旦应用程序执行操作，该操作的结果在后续的每个操作中都可见；
- 隔离性 - 一个用户的不完整操作不会对其他用户造成意外的副作用；
- 持久性 - 一旦操作完成，即使面对机器或系统故障也会保留该操作。

带有 ACID 语义的事务被添加到 Hive 中，来解决以下使用场景：

- 流式数据摄取。许多用户使用 Apache Flume、Apache Storm 或 Apache Kafka 等工具将数据流式传输到 Hadoop 集群中。虽然这些工具可以以每秒数百行或更多行的速度写入数据，但 Hive 只能每 15 分钟到 1 小时添加分区。更频繁地添加分区会很快导致表中的分区数量过多。这些工具可以将数据流放入现有分区，但这会导致读取器进行脏读取(也就是

说，读取器在开始查询后才会看到写入的数据)，并在目录中留下许多小文件，这会给 NameNode 带来压力。有了这个新功能，这个用例将得到支持，同时允许读者获得一致的数据视图，并避免过多的文件。

- 缓慢变化的维度。在典型的星型模式数据仓库中，维度表随时间变化缓慢。例如，零售商将开设新商店，需要将其添加到商店表中，或者现有商店可能会改变其平方英尺或其他一些跟踪特征。这些更改导致插入单个记录或更新记录(取决于所选择的策略)。从 0.14 开始，Hive 能够支持这一点。
- 数据重述。有时发现收集的数据不正确，需要纠正。或者，数据的第一个实例可能是稍后提供的完整数据的近似值(90%的服务器报告)。或者，业务规则可能要求由于后续交易而对某些交易进行重述(例如，在进行购买之后，客户可能会购买会员资格，从而有权享受折扣价格，包括之前购买的价格)。或者，根据合同，用户可能需要在关系终止时删除其客户的数据。从 Hive 0.14 开始，这些用例可以通过 INSERT、UPDATE 和 DELETE 来支持。
- 使用 SQL MERGE 语句批量更新。

5.6.4.4 Hive 数据压缩

基于查询和基于 MR 的数据压缩能力，是开箱即用的。当使用事务时，ALTER TABLE ... COMPACT 语句可以包含一个 TBLPROPERTIES 子句，用于更改压缩 MapReduce 作业属性或覆盖任何其他 Hive 表属性。

通常情况下，当使用 Hive 事务时，您不需要请求压缩，因为系统会检测到它们的需求并启动压缩。但是，如果关闭了表的压缩，或者您希望在系统不会选择的时间压缩表，则 ALTER TABLE 可以启动压缩。默认情况下，该语句会将压缩请求排入队列并返回。要观察压缩的进度，请使用 SHOW COMPACTIONS。可以指定“AND WAIT”以在压缩完成之前阻止操作。compaction_type 可以是 MAJOR、MINOR 或 REBALANCE。

5.6.4.5 Hive Replication

Hive 支持启动复制和增量复制，用于备份和恢复。Hive Replication 建立在 Metastore 事件和 ExIm 功能之上，为在集群之间复制 Hive 元数据和数据更改提供了一个框架。不要求源集群和副本运行相同的 Hadoop 发行版、Hive 版本或 Metastore RDBMS。复制系统具有相当“轻触”，表现出低耦合度并使用 Hive-metastore Thrift 服务作为集成点。但是，当前的实施并不是“开箱即用”的解决方案。特别是有必要提供某种编排服务，负责请求复制任务并执行它们。

- 潜在用途

  ■灾难恢复集群。

  ■将数据复制到云中以进行外部处理。

5.6.4.6 安全性和可观察性

Hive 支持 kerberos 认证，并与 Ranger 和 Atlas 集成以提高安全性和可观察性。Hive hook 向 Hive 注册以监听创建/更新/删除操作，并通过 Kafka 通知更新 Atlas 中的元数据，以了解 Hive 中的更改。

5.6.4.7 Hive LLAP

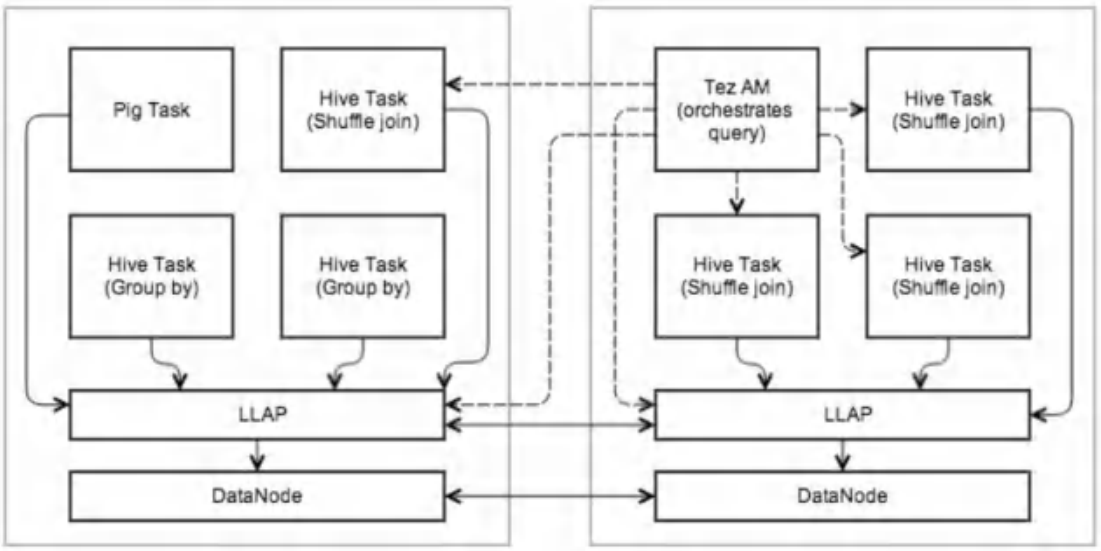
Apache Hive 通过低延迟分析处理（Live Long And Process LLAP）实现交互式和亚秒级 SQL, LLAP 通过使用持久的查询基础设施和优化的数据缓存使 Hive 更快。

LLAP 也称为 Live Long 和 Process，它提供了一种混合执行模型。它由一个长寿命的守护进程组成，它取代了与 HDFS DataNode 的直接交互，以及一个紧密集成的基于 DAG 的框架。

缓存、预取、一些查询处理和访问控制等功能被移入守护进程。小型/短期查询主要由该守护程序直接处理，而任何繁重的工作都将在标准 YARN 容器中执行。

与 DataNode 类似，其他应用程序也可以使用 LLAP 守护进程，尤其是在数据的关系视图优于以文件为中心的处理时。该守护进程还通过可选的 API（例如 InputFormat）开放，其他数据处理框架可以将其用作构建块。

最后但同样重要的是，细粒度的列级访问控制 -- 主流采用 Hive 的关键要求 -- 非常适合这个模型。



上图显示了使用 LLAP 执行的示例。Tez AM 协调整体执行。查询的初始阶段被推送到 LLAP。在 reduce 阶段，大型洗牌在单独的容器中进行。多个查询和应用程序可以同时访问 LLAP。

5.7 Spark 分布式内存计算引擎

5.7.1 Spark 简介

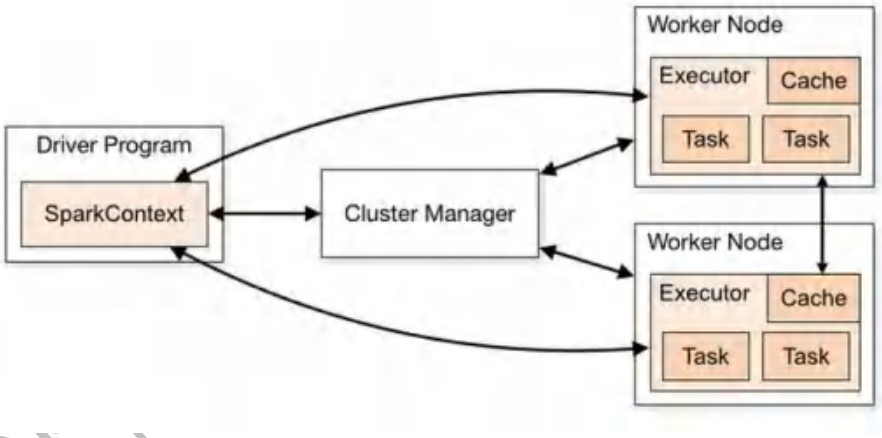
Spark 是基于内存的分布式计算框架。在迭代计算的场景下，数据处理过程中的数据可以存储在内存中，提供了比 MapReduce 高 10 到 100 倍的计算能力。Spark 可以使用 HDFS 作为底层存储，使用户能够快速地从 MapReduce 切换到 Spark 计算平台上去。Spark 提供一站式数据分析能力，包括小批量流式处理、离线批处理、SQL 查询、数据挖掘等，用户可以在同一个应用中无缝结合使用这些能力。

Spark 的特点如下：

- 通过分布式内存计算和 DAG（无回路有向图）执行引擎提升数据处理能力，比 MapReduce 性能高 10 倍到 100 倍。
- 提供多种语言开发接口（Scala/Java/Python），并且提供几十种高度抽象算子，可以很方便构建分布式的数据处理应用。
- 结合 SQL、Streaming、MLlib、GraphX 等形成数据处理栈，提供一站式数据处理能力。
- 完美契合 Hadoop 生态环境, Spark 应用可以运行在 Standalone、Mesos 或者 YARN 上，能够接入 HDFS、HBase、Hive 等多种数据源，支持 MapReduce 程序平滑转接。

5.7.2 Sprak 架构

Spark 的架构如下图所示,



图：Spark 架构

各模块的说明如下表所示。

表：基本概念说明

模块	说明
Cluster Manager	集群管理器，管理集群中的资源。Spark 支持多种集群管理器，Spark 自带的 Standalone 集群管理器、Mesos 或 YARN。Spark 集群默认采用 YARN 模式。
Application	Spark 应用，由一个 Driver Program 和多个 Executor 组成。
Deploy Mode	部署模式，分为 cluster 和 client 模式。cluster 模式下，Driver 会在集群内的节点运行；而在 client 模式下，Driver 在客户端运行（集群外）。
Driver Program	是 Spark 应用程序的主进程，运行 Application 的 main()函数并创建 SparkContext。负责应用程序的解析、生成 Stage 并调度 Task 到 Executor 上。通常 SparkContext 代表 Driver Program。
Executor	在 Work Node 上启动的进程，用来执行 Task，管理并处理应用中使用到的数据。一个 Spark 应用一般包含多个 Executor，每个 Executor 接收 Driver 的命令，并执行一到多个 Task。
Worker Node	集群中负责启动并管理 Executor 以及资源的节点。
Job	一个 Action 算子（比如 collect 算子）对应一个

	Job，由并行计算的多个 Task 组成。
Stage	每个 Job 由多个 Stage 组成，每个 Stage 是一个 Task 集合，由 DAG 分割而成。
Task	承载业务逻辑的运算单元，是 Spark 平台中可执行的最小工作单元。一个应用根据执行计划以及计算量分为多个 Task。

5.7.3 Spark 原理

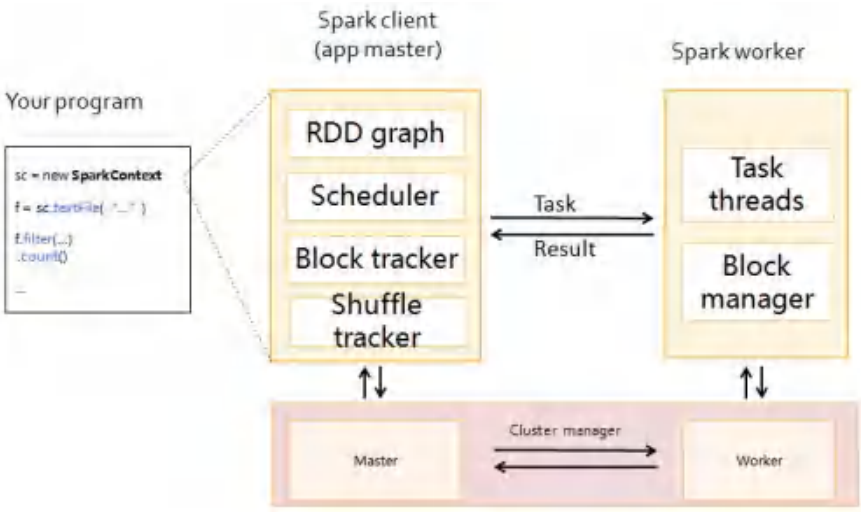
Spark 的应用运行架构如下图所示，运行流程如下所示：

应用程序（Application）是作为一个进程的集合运行在集群上的，由 Driver 进行协调。

在运行一个应用时，Driver 会去连接集群管理器（Standalone、Mesos、YARN）申请运行 Executor 资源，并启动 ExecutorBackend。然后由集群管理器在不同的应用之间调度资源。Driver 同时会启动应用程序 DAG 调度、Stage 划分、Task 生成。

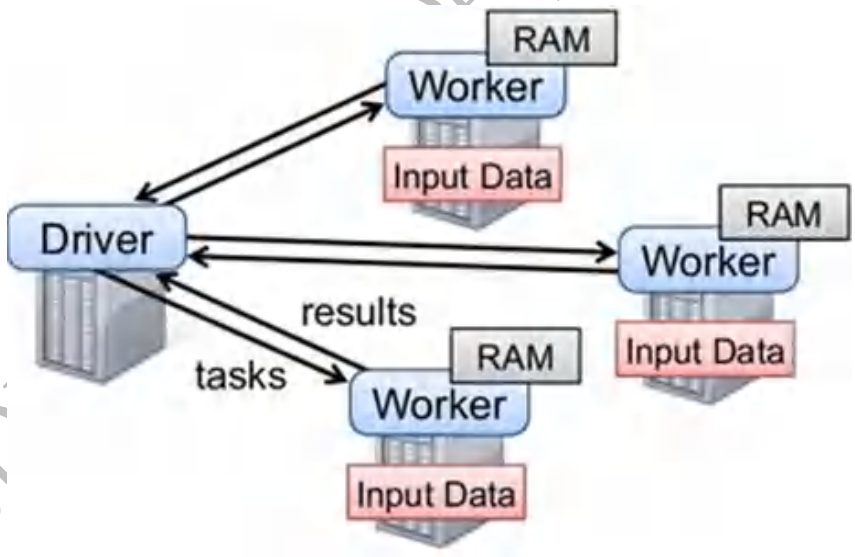
然后 Spark 会把应用的代码（传递给 SparkContext 的 JAR 或者 Python 定义的代码）发送到 Executor 上。

所有的 Task 执行完成后，用户的应用程序运行结束。



图：Spark 应用运行架构

Spark 采用 Master 和 worker 的模式，如下图所示。用户在 Spark 客户端提交应用程序，调度器将 Job 分解为多个 Task 发送到各个 Worker 中执行，各个 Worker 将计算的结果上报给 Driver（即 Master），Driver 聚合结果返回给客户端。



图：Spark 的 Master 和 Worker

在此结构中，有几个说明点：

- 应用之间是独立的。
- 每个应用有自己的 executor 进程，Executor 启动多个线程，并行地执行任务。无论是在调度方面，或者是 executor 方面。各个 Driver 独立调度

自己的任务；不同的应用任务运行在不同的 JVM 上，即不同的 Executor。

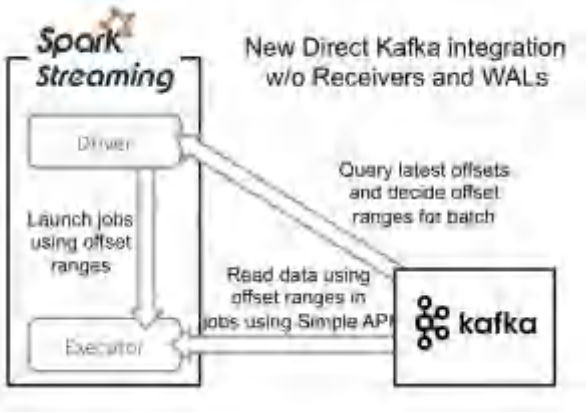
- 不同 Spark 应用之间是不共享数据的，除非把数据存储在外部的存储系统上（比如 HDFS）。
- 因为 Driver 程序在集群上调度任务，所以 Driver 程序最好和 worker 节点比较近，比如在一个相同的局部网络内。
- Spark on YARN 有两种部署模式：
- YARN-Cluster 模式下，Spark 的 Driver 会运行在 YARN 集群内的 ApplicationMaster 进程中，ApplicationMaster 已经启动之后，提交任务的客户端退出也不会影响任务的运行。
- YRAN-Client 模式下，Driver 启动在客户端进程内，ApplicationMaster 进程只用来向 YARN 集群申请资源。

5.7.4 Spark Streaming 原理

Spark Streaming 是一种构建在 Spark 上的实时计算框架，扩展了 Spark 处理大规模流式数据的能力。当前 Spark 支持两种数据处理方式：Direct Streaming 和 Receiver 方式。

5.7.5 Direct Streaming 计算流程

Direct Streaming 方式主要通过采用 Direct API 对数据进行处理。以 Kafka Direct 接口为例，与启动一个 Receiver 来连续不断地从 Kafka 中接收数据并写入到 WAL 中相比，Direct API 简单地给出每个 batch 区间需要读取的偏移量位置。然后，每个 batch 的 Job 被运行，而对应偏移量的数据在 Kafka 中已准备好。这些偏移量信息也被可靠地存储在 checkpoint 文件中，应用失败重启时可以直接读取偏移量信息。



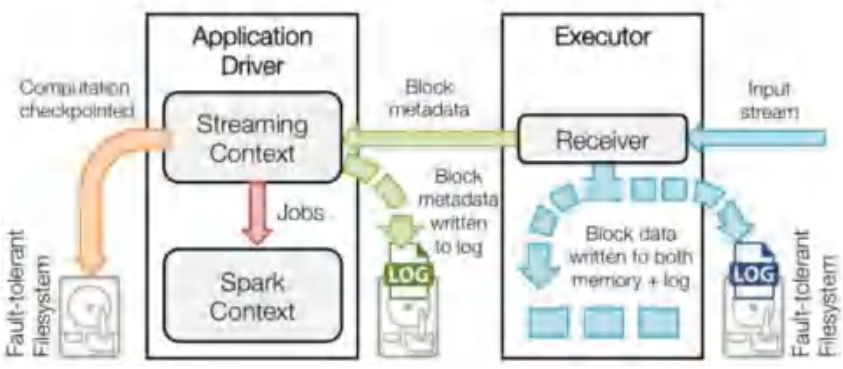
图：Direct Kafka 接口数据传输

需要注意的是，Spark Streaming 可以在失败后重新从 Kafka 中读取并处理数据段。然而，由于语义仅被处理一次，重新处理的结果和没有失败处理的结果是一致的。

因此，Direct API 消除了需要使用 WAL 和 Receivers 的情况，且确保每个 Kafka 记录仅被接收一次，这种接收更加高效。使得 Spark Streaming 和 Kafka 可以很好地整合在一起。总体来说，这些特性使得流处理管道拥有高容错性、高效性及易用性，因此推荐使用 Direct Streaming 方式处理数据。

5.7.6 Receiver 计算流程

在一个 Spark Streaming 应用开始时（也就是 Driver 开始时），相关的 StreamingContext（所有流功能的基础）使用 SparkContext 启动 Receiver 成为长驻运行任务。这些 Receiver 接收并保存流数据到 Spark 内存中以供处理。用户传送数据的生命周期如下图所示：



图：数据传输生命周期

1. 接收数据（蓝色箭头）

   - Receiver 将数据流分成一系列小块，存储到 Executor 内存中。另外，在启用预写日志（Write-ahead Log，简称 WAL）以后，数据同时还写入到容错文件系统的预写日志中。

2. 通知 Driver（绿色箭头）

   - 接收块中的元数据(Metadata)被发送到 Driver 的 StreamingContext。这个元数据包括：
   - 定位其在 Executor 内存中数据位置的块 Reference ID。
   - 若启用了 WAL，还包括块数据在日志中的偏移信息。

3. 处理数据（红色箭头）

   - 对每个批次的数据，StreamingContext 使用 Block 信息产生 RDD 及其 Job。StreamingContext 通过运行任务处理 Executor 内存中的 Block 来执行 Job。

4. 周期性地设置检查点（橙色箭头）

5. 为了容错的需要，StreamingContext 会周期性地设置检查点，并保存到外部文件系统中。

5.7.7 容错性

Spark 及其 RDD 允许无缝地处理集群中任何 Worker 节点的故障。鉴于 Spark Streaming 建立于 Spark 之上，因此其 Worker 节点也具备了同样的容错能力。然而，由于 Spark Streaming 的长正常运行需求，其应用程序必须也具备从 Driver 进程 (协调各个 Worker 的主要应用进程) 故障中恢复的能力。使 Spark Driver 能够容错是件很棘手的事情，因为可能是任意计算模式实现的任意用户程序。不过 Spark Streaming 应用程序在计算上有一个内在的结构：在每批次数据周期性地执行同样的 Spark 计算。这种结构允许把应用的状态(亦称 Checkpoint)周期性地保存到可靠的存储空间中，并在 Driver 重新启动时恢复该状态。

对于文件这样的源数据，这个 Driver 恢复机制足以做到零数据丢失，因为所有的数据都保存在了像 HDFS 或 S3 这样的容错文件系统中。但对于像 Kafka 和 Flume 等其他数据源，有些接收到的数据还只缓存在内存中，尚未被处理，就有可能会丢失。这是由于 Spark 应用的分布操作方式引起的。当 Driver 进程失败时，所有在 Cluster Manager 中运行的 Executor，连同在内存中的所有数据，也同时被终止。为了避免这种数据损失，Spark Streaming 引进了 WAL 功能。

WAL 通常被用于数据库和文件系统中，用来保证任何数据操作的持久性，即先将操作记入一个持久的日志，再对数据施加这个操作。若施加操作的过程中执行失败了，则通过读取日志并重新施加前面预定的操作，系统就得到了恢复。下面介绍了如何利用这样的概念保证接收到的数据的持久性。

Kafka 数据源使用 Receiver 来接收数据，是 Executor 中的长运行任务，负责从数据源接收数据，并且在数据源支持时还负责确认收到数据的结果（收到的数据被保存在 Executor 的内存中，然后 Driver 在 Executor 中运行来处理任务）。

当启用了预写日志以后，所有收到的数据同时还保存到了容错文件系统的日志文件中。此时即使 Spark Streaming 失败，这些接收到的数据也不会丢失。另外，接收数据的正确性只在数据被预写到日志以后 Receiver 才会确认，已经缓存但还没有保存的数据可以在 Driver 重新启动之后由数据源再发送一次。这两个机制确保了零数据丢失，即所有的数据或者从日志中恢复，或者由数据源重发。

如果需要启用预写日志功能，可以通过如下动作实现：

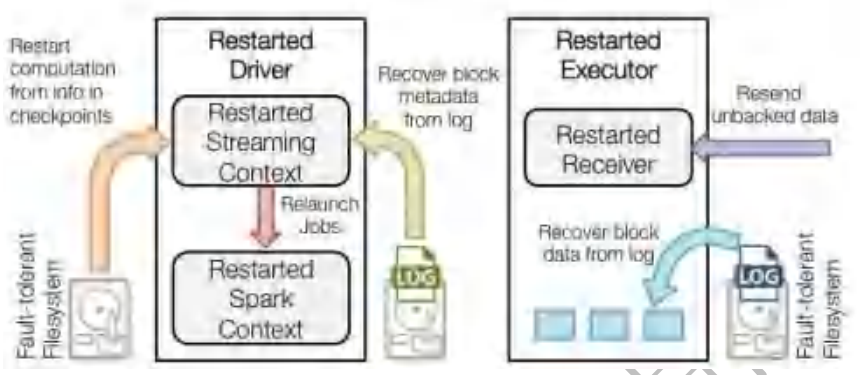
- 通过“streamingContext.checkpoint” (path-to-directory)设置 checkpoint 的目录，这个目录是一个 HDFS 的文件路径，既用作保存流的 checkpoint，又用作保存预写日志。
- 设置 SparkConf 的属性“spark.streaming.receiver.writeAheadLog.enable”为“true”（默认值是“false”）。

在 WAL 被启用以后，所有 Receiver 都获得了能够从可靠收到的数据中恢复的优势。建议缓存 RDD 时不采取多备份选项，因为用于预写日志的容错文件

系统很可能也复制了数据。

5.7.8 恢复流程

当一个失败的 Driver 重启时，按如下流程启动：



图：计算恢复流程

1. 恢复计算（橙色箭头）

使用 checkpoint 信息重启 Driver, 重新构造 SparkContext 并重启 Receiver。

2. 恢复元数据块（绿色箭头）

为了保证能够继续下去所必备的全部元数据块都被恢复。

3. 未完成作业的重新形成（红色箭头）

由于失败而没有处理完成的批处理，将使用恢复的元数据再次产生 RDD 和对应的作业。

4. 读取保存在日志中的块数据（蓝色箭头）

在这些作业执行时，块数据直接从预写日志中读出。这将恢复在日志中可靠地保存的所有必要数据。

5. 重发尚未确认的数据（紫色箭头）

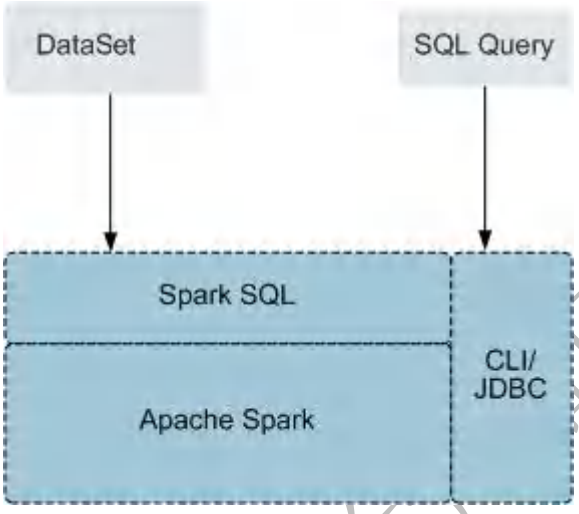
失败时没有保存到日志中的缓存数据将由数据源再次发送。因为 Receiver 尚未对其确认。

因此通过预写日志和可靠的 Receiver，Spark Streaming 就可以保证没有输

入数据会由于 Driver 的失败而丢失。

5.7.9 SparkSQL 和 DataSet 原理

SparkSQL



图：SparkSQL 和 DataSet

Spark SQL 是 Spark 中用于结构化数据处理的模块。在 Spark 应用中，可以无缝的使用 SQL 语句亦或是 DataFrame API 对结构化数据进行查询。

Spark SQL 以及 DataFrame 还提供了一种通用的访问多数据源的方式，可访问的数据源包括 Hive、CSV、Parquet、ORC、JSON 和 JDBC 数据源，这些不同的数据源之间也可以实现互相操作。Spark SQL 复用了 Hive 的前端处理逻辑和元数据处理模块，使用 Spark SQL 可以直接对已有的 Hive 数据进行查询。

另外，SparkSQL 还提供了诸如 API、CLI、JDBC 等诸多接口，对客户端提供多样接入形式。

Spark SQL Native DDL/DML

Spark1.5 将很多 DDL/DML 命令下压到 Hive 执行，造成了与 Hive 的耦合，且在一定程度上不够灵活（比如报错不符合预期、结果与预期不一致等）。

Spark2x 实现了命令的本地化, 使用 Spark SQL Native DDL/DML 取代 Hive 执行 DDL/DML 命令。一方面实现和 Hive 的解耦，另一方面可以对命令进行定制化。

DataSet

DataSet 是一个由特定域的对象组成的强类型集合，可通过功能或关系操作并行转换其中的对象。 每个 Dataset 还有一个非类型视图，即由多个列组成的 DataSet，称为 DataFrame。

DataFrame 是一个由多个列组成的结构化的分布式数据集合，等同于关系数据库中的一张表，或者是 R/Python 中的 data frame。DataFrame 是 Spark SQL 中的最基本的概念，可以通过多种方式创建，例如结构化的数据集、Hive 表、外部数据库或者是 RDD。

可用于 DataSet 的操作分为 Transformation 和 Action。可参见 • Transformation 和 Action。

- Transformation 操作可生成新的 DataSet。

  如 map、filter、select 和 aggregate (groupBy)。

- Action 操作可触发计算及返回记结果。

  如 count、show 或向文件系统写数据。

通常使用两种方法创建一个 DataSet：

- 最常见的方法是通过使用 SparkSession 上的 read 函数将 Spark 指向存储系统上的某些文件。

```
val people = spark.read.parquet("...").as[Person]   // Scala
DataSet<Person> people =
spark.read().parquet("...").as(Encoders.bean(Person.class));//Java
```

- 还可通过已存在的 DataSet 上可用的 transformation 操作来创建数据集。例如，在已存在的 DataSet 上应用 map 操作来创建新的 DataSet：

```
val names = people.map(_.name)  // 使用 Scala 语言，且 names 为一个 Datase

Dataset<String> names = people.map((Person p) -> p.name, Encoders.STRING)); // Java
```

CLI 和 JDBCServer

除了 API 编程接口之外，Spark SQL 还对外提供 CLI/JDBC 接口：

- spark-shell 和 spark-sql 脚本均可以提供 CLI，以便于调试。
- JDBCServer 提供 JDBC 接口，外部可直接通过发送 JDBC 请求来完成结构化数据的计算和解析

5.7.10 SparkSession 原理

SparkSession 是 Spark2x 编程的统一 API，也可看作是读取数据的统一入口。SparkSession 提供了一个统一的入口点来执行以前分散在多个类中的许多操作，并且还为那些较旧的类提供了访问器方法，以实现最大的兼容性。

使用构建器模式创建 SparkSession。如果存在 SparkSession，构建器将自动重用现有的 SparkSession；如果不存在则会创建一个 SparkSession。 在 I/O 期间，在构建器中设置的配置项将自动同步到 Spark 和 Hadoop。

```
import org.apache.spark.sql.SparkSession

val sparkSession = SparkSession.builder

.master("local")

.appName("my-spark-app")

.config("spark.some.config.option", "config-value")

.getOrCreate()
```

- SparkSession 可以用于对数据执行 SQL 查询，将结果返回为 DataFrame。

```
sparkSession.sql("select * from person").show
```

- SparkSession 可以用于设置运行时的配置项，这些配置项可以在 SQL 中使用变量替换。

```
sparkSession.conf.set("spark.some.config", "abcd")
sparkSession.conf.get("spark.some.config")
sparkSession.sql("select ${spark.some.config}")
```

- SparkSession 包括一个“catalog”方法，其中包含使用 Metastore（即数据目录）的方法。方法返回值为数据集，可以使用相同的 Dataset API 来运行。

```
val tables = sparkSession.catalog.listTables()
val columns = sparkSession.catalog.listColumns("myTable")
```

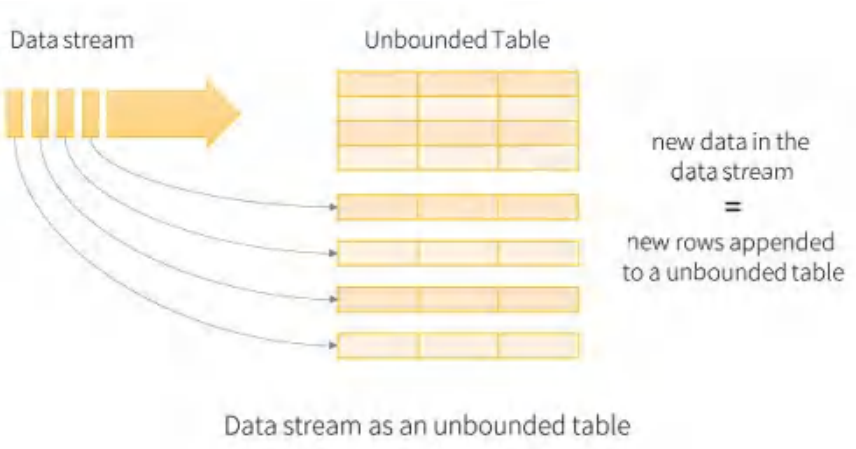
- 底层 SparkContext 可以通过 SparkSession 的 SparkContext API 访问。

```
val sparkContext = sparkSession.sparkContext
```

5.7.11 Structured Streaming 原理

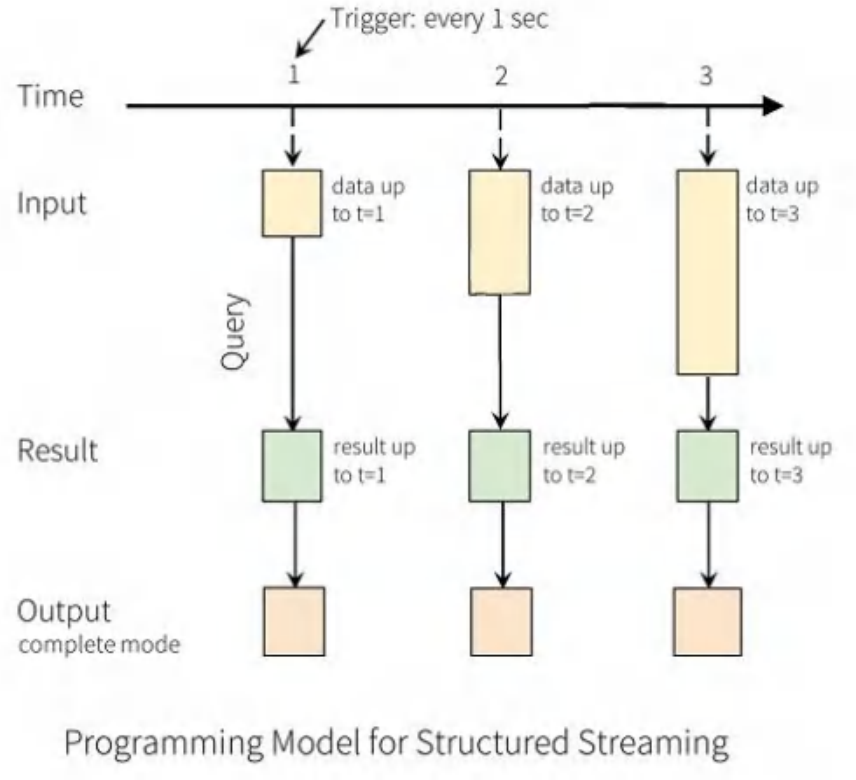
Structured Streaming 是构建在 Spark SQL 引擎上的流式数据处理引擎，使用 Scala 编写，具有容错功能。可以像使用静态 RDD 数据那样编写流式计算过程。当流数据连续不断的产生时，Spark SQL 将会增量的、持续不断的处理这些数据并将结果更新到结果集中。

Structured Streaming 的核心是将流式的数据看成一张不断增加的数据库表，这种流式的数据处理模型类似于数据块处理模型，可以把静态数据库表的一些查询操作应用在流式计算中，Spark 执行标准的 SQL 查询，从不断增加的无边界表中获取数据。



图：Structured Streaming 无边界表

每一条查询的操作都会产生一个结果集 Result Table。每一个触发间隔，当新的数据新增到表中，都会最终更新 Result Table。无论何时结果集发生了更新，都能将变化的结果写入一个外部的存储系统。



图：Structured Streaming 数据处理模型

Structured Streaming 在 OutPut 阶段可以定义不同的存储方式，有如下 3 种：

- Complete Mode：整个更新的结果集都会写入外部存储。整张表的写入操作将由外部存储系统的连接器完成。
- Append Mode：当时间间隔触发时，只有在 Result Table 中新增加的数据行会被写入外部存储。这种方式只适用于结果集中已经存在的内容不希望发生改变的情况下，如果已经存在的数据会被更新，不适合适用此种方式。
- Update Mode：当时间间隔触发时，只有在 Result Table 中被更新的数据才会被写入外部存储系统。注意，和 Complete Mode 方式的不同之处是不更新的结果集不会写入外部存储。

5.7.11.1 基本概念

RDD

即弹性分布数据集（Resilient Distributed Dataset），是 Spark 的核心概念。指的是一个只读的，可分区的分布式数据集，这个数据集的全部或部分可以缓存在内存中，在多次计算间重用。

RDD 的生成：

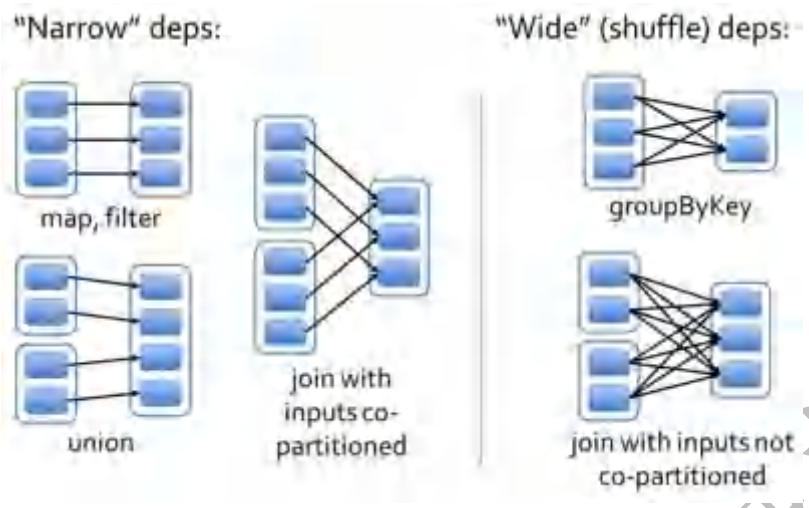
- 从 HDFS 输入创建，或从与 Hadoop 兼容的其他存储系统中输入创建。
- 从父 RDD 转换得到新 RDD。
- 从数据集合转换而来，通过编码实现。

RDD 的存储：

- 用户可以选择不同的存储级别缓存 RDD 以便重用（RDD 有 11 种存储级别）。
- 当前 RDD 默认是存储于内存，但当内存不足时，RDD 会溢出到磁盘中。

Dependency（RDD 的依赖）

RDD 的依赖分别为：窄依赖和宽依赖。



图：RDD 的依赖

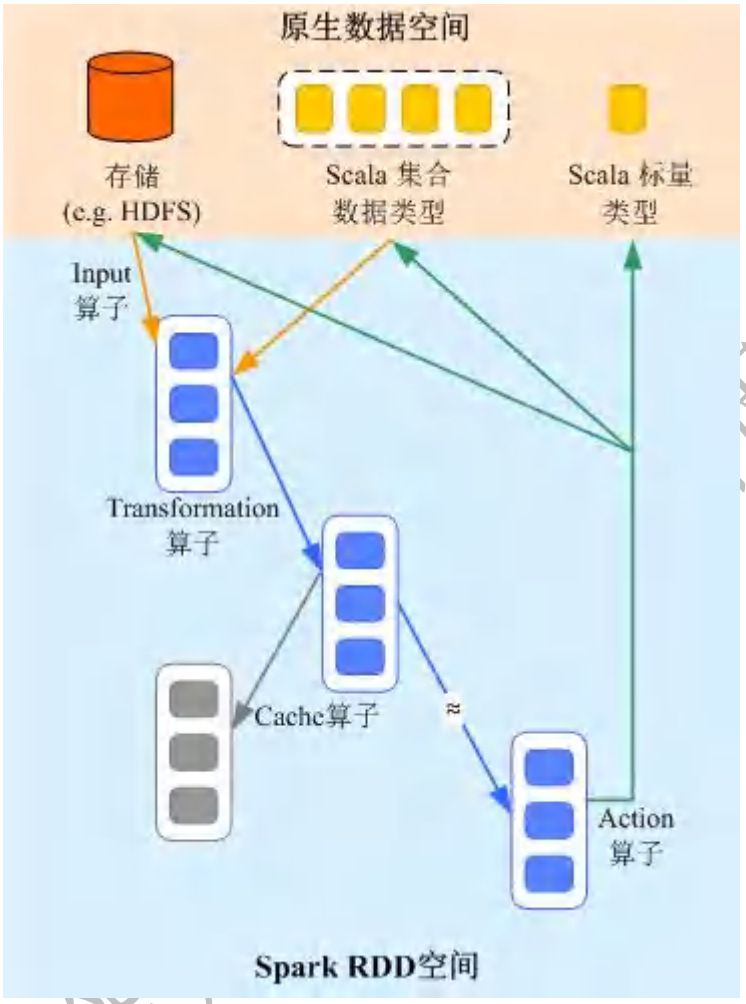
- 窄依赖：指父 RDD 的每一个分区最多被一个子 RDD 的分区所用。
- 宽依赖：指子 RDD 的分区依赖于父 RDD 的所有分区。

窄依赖对优化很有利。逻辑上，每个 RDD 的算子都是一个 fork/join（此 join 非上文的 join 算子，而是指同步多个并行任务的 barrier）：把计算 fork 到每个分区，算完后 join，然后 fork/join 下一个 RDD 的算子。如果直接翻译到物理实现，是很不经济的：一是每一个 RDD（即使是中间结果）都需要物化到内存或存储中，费时费空间；二是 join 作为全局的 barrier，是很昂贵的，会被最慢的那个节点拖死。如果子 RDD 的分区到父 RDD 的分区是窄依赖，就可以实施经典的 fusion 优化，把两个 fork/join 合为一个；如果连续的变换算子序列都是窄依赖，就可以把很多个 fork/join 并为一个，不但减少了大量的全局 barrier，而且无需物化很多中间结果 RDD，这将极大地提升性能。Spark 把这个叫做流水线（pipeline）优化。

Transformation 和 Action（RDD 的操作）

对 RDD 的操作包含 Transformation（返回值还是一个 RDD）和 Action（返回值不是一个 RDD）两种。RDD 的操作流程如图 11 所示。其中 Transformation 操作是 Lazy 的，也就是说从一个 RDD 转换生成另一个 RDD 的操作不是马上执行，Spark 在遇到 Transformations 操作时只会记录需要这样的操作，并不会去执行，需要等到有 Actions 操作的时候才会真正启动计算过程进行计算。Actions

操作会返回结果或把 RDD 数据写到存储系统中。Actions 是触发 Spark 启动计算的动因。



图：RDD 操作示例

RDD 看起来与 Scala 集合类型没有太大差别，但数据和运行模型大相迥异。

```
val file = sc.textFile("hdfs://...")

val errors = file.filter(_.contains("ERROR"))

errors.cache()

errors.count()
```

1. textFile 算子从 HDFS 读取日志文件，返回 file（作为 RDD）。

2. filter 算子筛出带“ERROR”的行，赋给 errors（新 RDD）。filter 算子

是一个 Transformation 操作。

3. cache 算子缓存下来以备未来使用。

4. count 算子返回 errors 的行数。count 算子是一个 Action 操作。

Transformation 操作可以分为如下几种类型：

- 视 RDD 的元素为简单元素。

输入输出一对一，且结果 RDD 的分区结构不变，主要是 map。

输入输出一对多，且结果 RDD 的分区结构不变，如 flatMap（map 后由一个元素变为一个包含多个元素的序列，然后展平为一个个的元素）。

输入输出一对一，但结果 RDD 的分区结构发生了变化，如 union（两个 RDD 合为一个，分区数变为两个 RDD 分区数之和）、coalesce（分区减少）。

从输入中选择部分元素的算子，如 filter、distinct（去除重复元素）、subtract（本 RDD 有、其他 RDD 无的元素留下来）和 sample（采样）。

- 视 RDD 的元素为 Key-Value 对。

对单个 RDD 做一对一运算，如 mapValues（保持源 RDD 的分区方式，这与 map 不同）；

对单个 RDD 重排，如 sort、partitionBy（实现一致性的分区划分，这个对数据本地性优化很重要）；

对单个 RDD 基于 key 进行重组和 reduce，如 groupByKey、reduceByKey；

对两个 RDD 基于 key 进行 join 和重组，如 join、cogroup。

Action 操作可以分为如下几种：

- 生成标量，如 count（返回 RDD 中元素的个数）、reduce、fold/aggregate（返回几个标量）、take（返回前几个元素）。

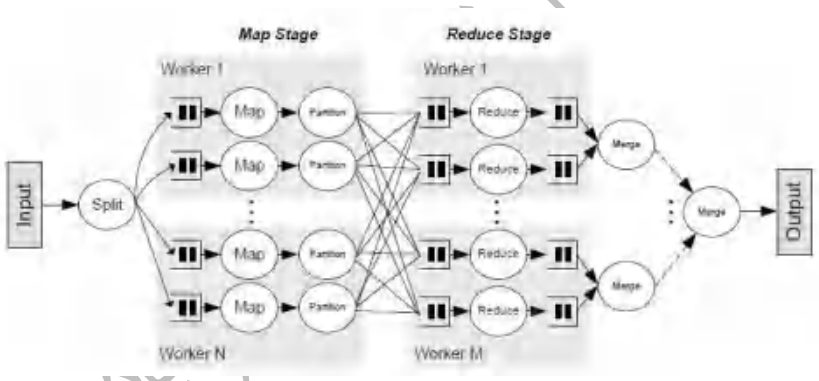
- 生成 Scala 集合类型，如 collect（把 RDD 中的所有元素倒入 Scala 集合类型）、lookup（查找对应 key 的所有值）。

- 写入存储，如与前文 textFile 对应的 saveAsTextFile。
- 还有一个检查点算子 checkpoint。当 Lineage 特别长时（这在图计算中时常发生），出错时重新执行整个序列要很长时间，可以主动调用 checkpoint 把当前数据写入稳定存储，作为检查点。

Shuffle

Shuffle 是 MapReduce 框架中的一个特定的 phase，介于 Map phase 和 Reduce phase 之间，当 Map 的输出结果要被 Reduce 使用时，每一条输出结果需要按 key 哈希，并且分发到对应的 Reducer 上去，这个过程就是 shuffle。由于 shuffle 涉及到了磁盘的读写和网络的传输，因此 shuffle 性能的高低直接影响到了整个程序的运行效率。

下图清晰地描述了 MapReduce 算法的整个流程。



图：算法流程

概念上 shuffle 就是一个沟通数据连接的桥梁，实际上 shuffle 这一部分是如何实现的呢，下面就以 Spark 为例讲一下 shuffle 在 Spark 中的实现。

Shuffle 操作将一个 Spark 的 Job 分成多个 Stage，前面的 stages 会包括一个或多个 ShuffleMapTasks，最后一个 stage 会包括一个或多个 ResultTask。

- Spark Application 的结构

Spark Application 的结构可分为两部分：初始化 SparkContext 和主体程序。

- 初始化 SparkContext：构建 Spark Application 的运行环境。

构建 SparkContext 对象，如：

```
new SparkContext(master, appName, [SparkHome], [jars])
```

参数介绍：

master：连接字符串，连接方式有 local、yarn-cluster、yarn-client 等。

appName：构建的 Application 名称。

SparkHome：集群中安装 Spark 的目录。

jars：应用程序代码和依赖包。

- 主体程序：处理数据

Spark shell 命令

Spark 基本 shell 命令，支持提交 Spark 应用。命令为：

```
./bin/spark-submit \
--class <main-class> \
--master <master-url> \
... # other options
<application-jar> \
[application-arguments]
```

参数解释：

--class：Spark 应用的类名。

--master：Spark 用于所连接的 master，如 yarn-client，yarn-cluster 等。

application-jar：Spark 应用的 jar 包的路径。

application-arguments：提交 Spark 应用的所需要的参数（可以为空）。

Spark JobHistory Server

用于监控正在运行的或者历史的 Spark 作业在 Spark 框架各个阶段的细节以及提供日志显示，帮助用户更细粒度地去开发、配置和调优作业。

5.7.11.2 HA 方案介绍

背景介绍

基于社区已有的 JDBCServer 基础上，采用多主实例模式实现了其高可用性方案。集群中支持同时共存多个 JDBCServer 服务，通过客户端可以随机连接其中的任意一个服务进行业务操作。即使集群中一个或多个 JDBCServer 服务停止工作，也不影响用户通过同一个客户端接口连接其他正常的 JDBCServer 服务。

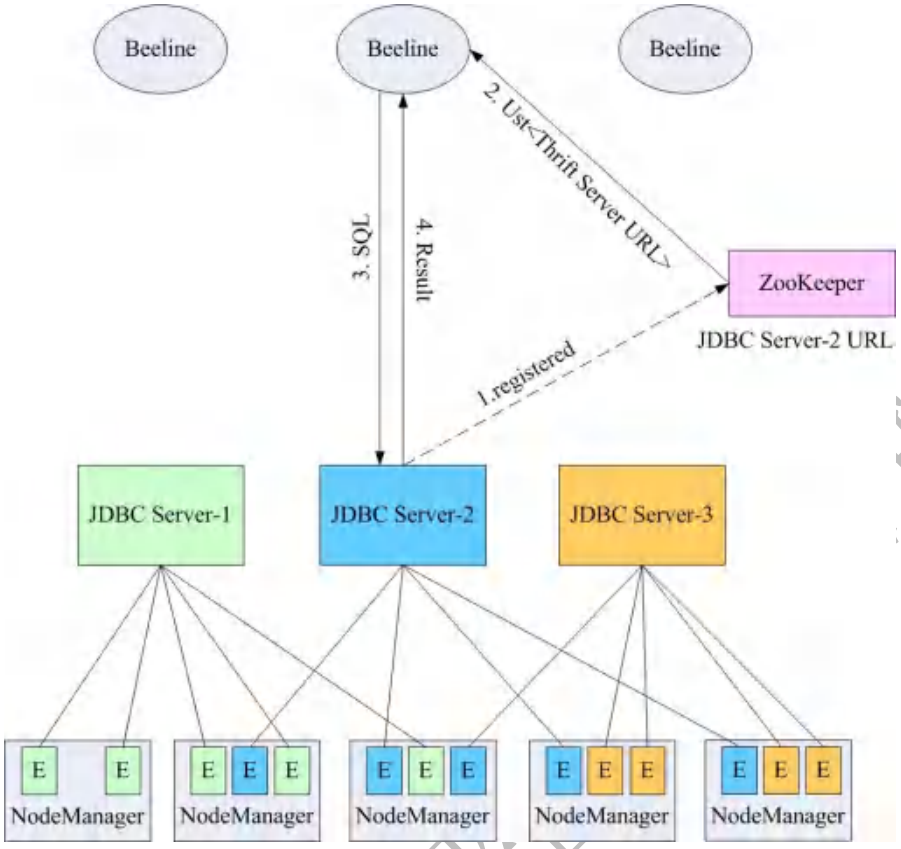
多主实例模式相比主备模式的 HA 方案，优势主要体现在对以下两种场景的改进。

- 主备模式下，当发生主备切换时，会存在一段时间内服务不可用，该时间 JDBCServer 无法控制，取决于 Yarn 服务的资源情况。
- Spark 中通过类似于 HiveServer2 的 Thrift JDBC 提供服务，用户通过 Beeline 以及 JDBC 接口访问。因此 JDBCServer 集群的处理能力取决于主 Server 的单点能力，可扩展性不够。

采用多主实例模式的 HA 方案，不仅可以规避主备切换服务中断的问题，实现服务不中断或少中断，还可以通过横向扩展集群来提高并发能力。

实现方案

多主实例模式的 HA 方案原理如下图所示



图：Spark JDBCServer HA

1. JDBCServer 在启动时，向 ZooKeeper 注册自身消息，在指定目录中写入节点，节点包含了该实例对应的 IP，端口，版本号和序列号等信息（多节点信息之间以逗号隔开）。

示例如下：

```
[serverUri=192.168.169.84:22550;version=V100R002C80SPC200;sequence=0000001244,serverUri=192.168.195.232:22550 ;version=V100R002C80SPC200;sequence=0000001242,serverUri=192.168.81.37:22550 ;version=V100R002C80SPC200;sequence=0000001243]
```

2. 客户端连接 JDBCServer 时，需要指定 Namespace，即访问 ZooKeeper 哪个目录下的 JDBCServer 实例。在连接的时候，会从 Namespace 下随机选择一个实例连接，详细 URL 参见下文连接介绍。

3. 户端成功连接 JDBCServer 服务后，向 JDBCServer 服务发送 SQL 语

句。

4. JDBCServer 服务执行客户端发送的 SQL 语句后，将结果返回给客户端。

在 HA 方案中，每个 JDBCServer 服务（即实例）都是独立且等同的，当其中一个实例在升级或者业务中断时，其他的实例也能接受客户端的连接请求。

多主实例方案遵循以下规则：

- 当一个实例异常退出时，其他实例不会接管此实例上的会话，也不会接管此实例上运行的业务。
- 当 JDBCServer 进程停止时，删除在 ZooKeeper 上的相应节点。
- 由于客户端选择服务端的策略是随机的，可能会出现会话随机分配不均匀的情况，进而可能引起实例间的负载不均衡。
- 实例进入维护模式（即进入此模式后不再接受新的客户端连接）后，当达到退服超时时间，仍在此实例上运行的业务有可能会发生失败。

多主实例模式

多主实例模式的客户端读取 ZooKeeper 节点中的内容，连接对应的 JDBCServer 服务。连接字符串为：

安全模式下：

Kinit 认证方式下的 JDBCURL 如下所示：

```
jdbc:hive2://<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>
,<zkNode3_IP>:<zkNode3_Port>/;serviceDiscoveryMode=zooKeeper;zooKee
perNamespace=sparkthriftserver2x;saslQop=auth-conf;auth=KERBEROS;prin
cipal=spark2x/hadoop.hadoop.com@HADOOP.COM;
```

示例：安全模式下通过 Beeline 客户端连接时执行以下命令：

```
sh CLIENT_HOME/spark/bin/beeline -u
"jdbc:hive2://<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Por
```

t>,<zkNode3_IP>:<zkNode3_Port>/**;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver2x;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.hadoop.com@HADOOP.COM;"**

Keytab 认证方式下的 JDBCURL 如下所示：

```
jdbc:hive2://<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver2x;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.hadoop.com@HADOOP.COM;user.principal=<principal_name>;user.keytab=<path_to_keytab>
```

其中<principal_name>表示用户使用的 Kerberos 用户的 principal，如“test@HADOOP.COM”。<path_to_keytab>表示<principal_name>对应的 keytab 文件路径，如“/opt/auth/test/user.keytab”。

普通模式下：

jdbc:hive2://*<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>*/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=**sparkthriftserver2x**;

示例：普通模式下通过 Beeline 客户端连接时执行以下命令：

```
sh CLIENT_HOME/spark/bin/beeline -u "jdbc:hive2://<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver2x;"
```

非多主实例模式

非多主实例模式的客户端连接的是某个指定 JDBCServer 节点。该模式的连接字符串相比多主实例模式的去掉关于 Zookeeper 的参数项

“serviceDiscoveryMode”和“zooKeeperNamespace”。

示例:安全模式下通过Beeline客户端连接非多主实例模式时执行以下命令:

```
Sh CLIENT_HOME/spark/bin/beeline -u
"jdbc:hive2://<server_IP>:<server_Port>/;user.principal=spark2x/hadoop.hadoop.com@HADOOP.COM;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.hadoop.com@HADOOP.COM;"
```

多主实例模式与非多主实例模式两种模式的 JDBCServer 接口相比,除连接方式不同外其他使用方法相同。由于 Spark JDBCServer 是 Hive 中的 HiveServer2 的另外一个实现.

5.7.11.3 多租户

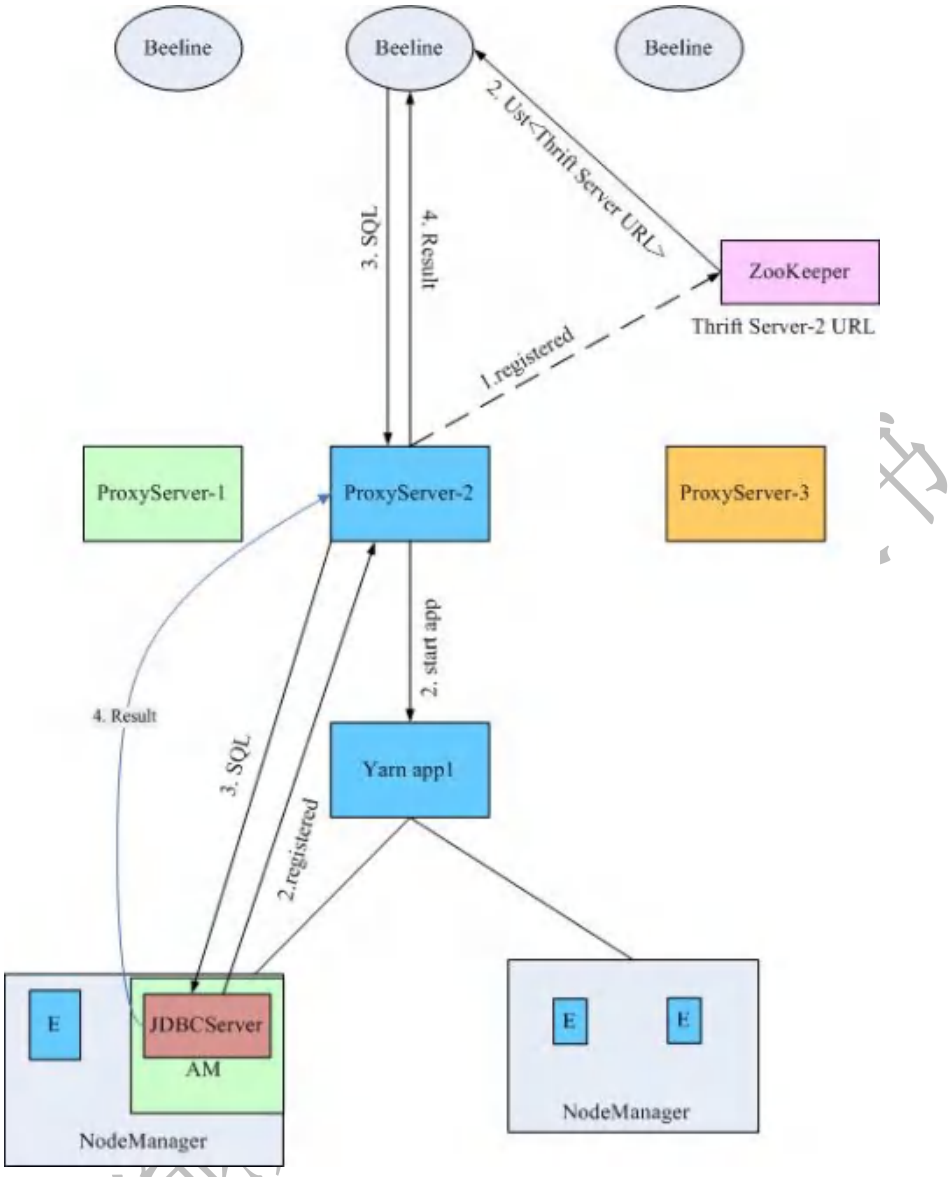
背景介绍

JDBCServer 多主实例方案中，JDBCServer 实现使用 YARN-Client 模式，但 YARN 资源队列只有一个，为了解决这种资源局限的问题，引入了多租户模式。

多租户模式是将 JDBCServer 和租户绑定，每一个租户对应一个或多个 JDBCServer，而一个 JDBCServer 只给一个租户提供服务。不同的租户可以配置不同的 YARN 队列，从而达到资源隔离，且 JDBCServer 根据需求动态启动，可避免浪费资源。

实现方案

多租户模式的 HA 方案原理如下图所示。



图：Spark JDBCServer 多租户

1. ProxyServer 在启动时，向 ZooKeeper 注册自身消息，在指定目录中写入节点信息，节点信息包含了该实例对应的 IP，端口，版本号和序列号等信息（多节点信息之间以逗号隔开）。

示例如下：

```
serverUri=192.168.169.84:22550;version=V100R002C80SPC200;sequence=0000001244,serverUri=192.168.195.232:22550;version=V100R002C80SPC200;sequence=0000001242,serverUri=192.168.81.37:22550;version=V100R002C80SPC200;sequence=0000001243,
```

2. 客户端连接 ProxyServer 时，需要指定 Namespace，即访问 ZooKeeper 哪个目录下的 ProxyServer 实例。在连接的时候，会从 Namespace 下随机选择一个实例连接，详细 URL 参见下文 URL 介绍。

3. 客户端成功连接 ProxyServer 服务，ProxyServer 服务首先确认是否有该租户的 JDBCServer 存在，如果有，直接将 Beeline 连上真正的 JDBCServer；如果没有，则以 YARN-Cluster 模式启动一个新的 JDBCServer。JDBCServer 启动成功后，ProxyServer 会获取 JDBCServer 的地址，并将 Beeline 连上 JDBCServer。

4. 客户端发送 SQL 语句给 ProxyServer，ProxyServer 将语句转交给真正连上的 JDBCServer 处理。最后 JDBCServer 服务将结果返回给 ProxyServer，ProxyServer 再将结果返回给客户端。

在 HA 方案中，每个 ProxyServer 服务（即实例）都是独立且等同的，当其中一个实例在升级或者业务中断时，其他的实例也能接受客户端的连接请求。

多租户模式

多租户模式的客户端读取 ZooKeeper 节点中的内容，连接对应的 ProxyServer 服务。连接字符串为：

安全模式下：

Kinit 认证方式下的客户端 URL 如下所示：

```
jdbc:hive2://<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver2x;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.hadoop.com@HADOOP.COM;
```

示例：安全模式下通过 Beeline 客户端连接时执行以下命令：

```
sh CLIENT_HOME/spark/bin/beeline -u "jdbc:hive2://<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Por
```

```
t>,<zkNode3_IP>:<zkNode3_Port>/;serviceDiscoveryMode=zooKeeper;zoo
KeeperNamespace=sparkthriftserver2x;saslQop=auth-conf;auth=KERBERO
S;principal=spark2x/hadoop.hadoop.com@HADOOP.COM;"
```

Keytab 认证方式下的 URL 如下所示：

```
jdbc:hive2://<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>
,<zkNode3_IP>:<zkNode3_Port>/;serviceDiscoveryMode=zooKeeper;zooKee
perNamespace=sparkthriftserver2x;saslQop=auth-conf;auth=KERBEROS;prin
cipal=spark2x/hadoop.hadoop.com@HADOOP.COM;user.principal=<principa
l_name>;user.keytab=<path_to_keytab>
```

其中<principal_name>表示用户使用的 Kerberos 用户的 principal，如“test@HADOOP.COM”。<path_to_keytab>表示<principal_name>对应的 keytab 文件路径，如“/opt/auth/test/user.keytab”。

普通模式下：

```
jdbc:hive2://<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>
,<zkNode3_IP>:<zkNode3_Port>/;serviceDiscoveryMode=zooKeeper;zooKee
perNamespace=sparkthriftserver2x;
```

示例：普通模式下通过 Beeline 客户端连接时执行以下命令：

```
shCLIENT_HOME/spark/bin/beeline -u
"jdbc:hive2://<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Por
t>,<zkNode3_IP>:<zkNode3_Port>/;serviceDiscoveryMode=zooKeeper;zo
oKeeperNamespace=sparkthriftserver2x;"
```

非多租户模式

非多租户模式的客户端连接的是某个指定 JDBCServer 节点。该模式的连接字符串相比多主实例模式的去掉关于 ZooKeeper 的参数项

“serviceDiscoveryMode”和“zooKeeperNamespace”。

示例：安全模式下通过 Beeline 客户端连接非多租户模式时执行以下命令：

```
sh CLIENT_HOME/spark/bin/beeline -u
"jdbc:hive2://<server_IP>:<server_Port>/;user.principal=spark2x/hadoop.hadoop.com@HADOOP.COM;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.hadoop.com@HADOOP.COM;"
```

多租户模式与非多租户模式两种模式的 JDBCServer 接口相比，除连接方式不同外其他使用方法相同。

指定租户

一般情况下，某用户提交的客户端会连接到该用户默认所属租户的 JDBCServer 上，若需要连接客户端到指定租户的 JDBCServer 上，可以通过添加--hiveconf mapreduce.job.queuename 进行指定。

通过 Beeline 连接的命令示例如下（aaa 为租户名称）：

```
beeline --hiveconf mapreduce.job.queuename=aaa -u
'jdbc:hive2://192.168.39.30:24002,192.168.40.210:24002,192.168.215.97:24002;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver2x;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.hadoop.com@HADOOP.COM;'
```

5.7.11.4 与组件的关系

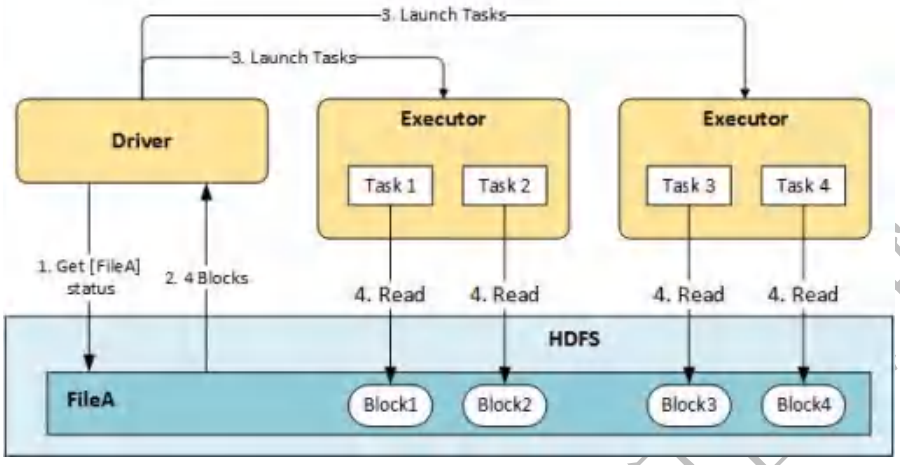
Spark 和 HDFS 的配合关系

通常，Spark 中计算的数据可以来自多个数据源，如 Local File、HDFS 等。最常用的是 HDFS，用户可以一次读取大规模的数据进行并行计算。在计算完成后，也可以将数据存储到 HDFS。

分解来看，Spark 分成控制端(Driver)和执行端（Executor）。控制端负责

任务调度，执行端负责任务执行。

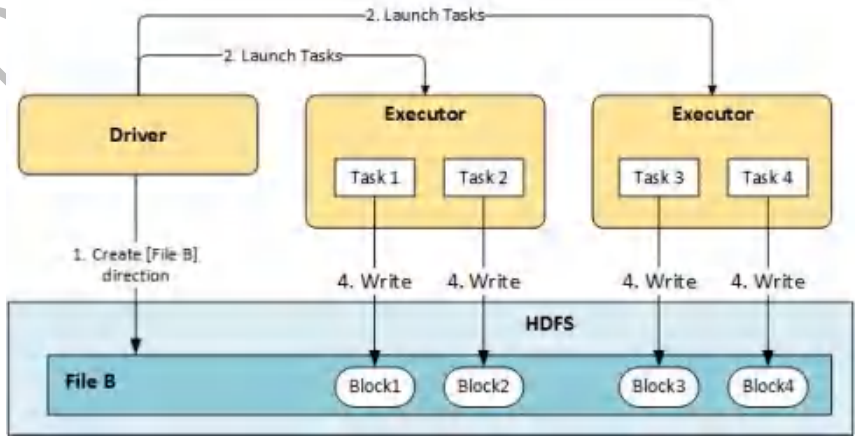
读取文件的过程如图所示。



图：读取文件过程

读取文件步骤的详细描述如下所示：

1. Driver 与 HDFS 交互获取 File A 的文件信息。
2. HDFS 返回该文件具体的 Block 信息。
3. Driver 根据具体的 Block 数据量，决定一个并行度，创建多个 Task 去读取这些文件 Block。
4. 在 Executor 端执行 Task 并读取具体的 Block, 作为 RDD(弹性分布数据集)的一部分。



图：写入文件过程

HDFS 文件写入的详细步骤如下所示：

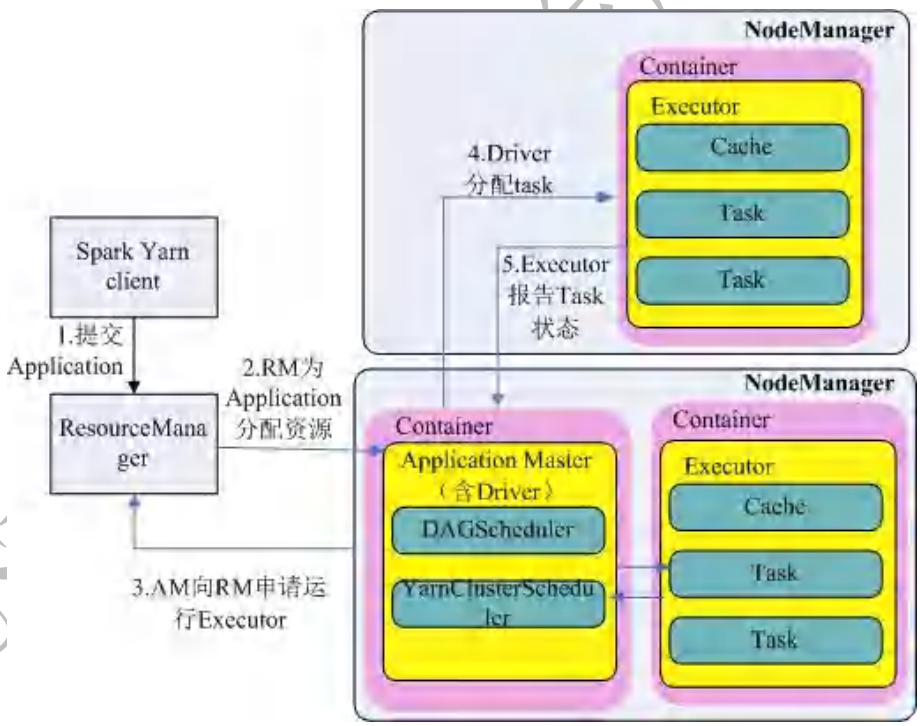
1. Driver 创建要写入文件的目录。
2. 根据 RDD 分区分块情况，计算出写数据的 Task 数，并下发这些任务到 Executor。
3. Executor 执行这些 Task，将具体 RDD 的数据写入到步骤 1 创建的目录下。

Spark 和 YARN 的配合关系

Spark 的计算调度方式，可以通过 YARN 的模式实现。Spark 共享 YARN 集群提供丰富的计算资源，将任务分布式的运行起来。Spark on YARN 分两种模式：YARN Cluster 和 YARN Client。

YARN Cluster 模式

运行框架如下图所示。



图：Spark on yarn-cluster 运行框架

Spark on YARN-Cluster 实现流程：

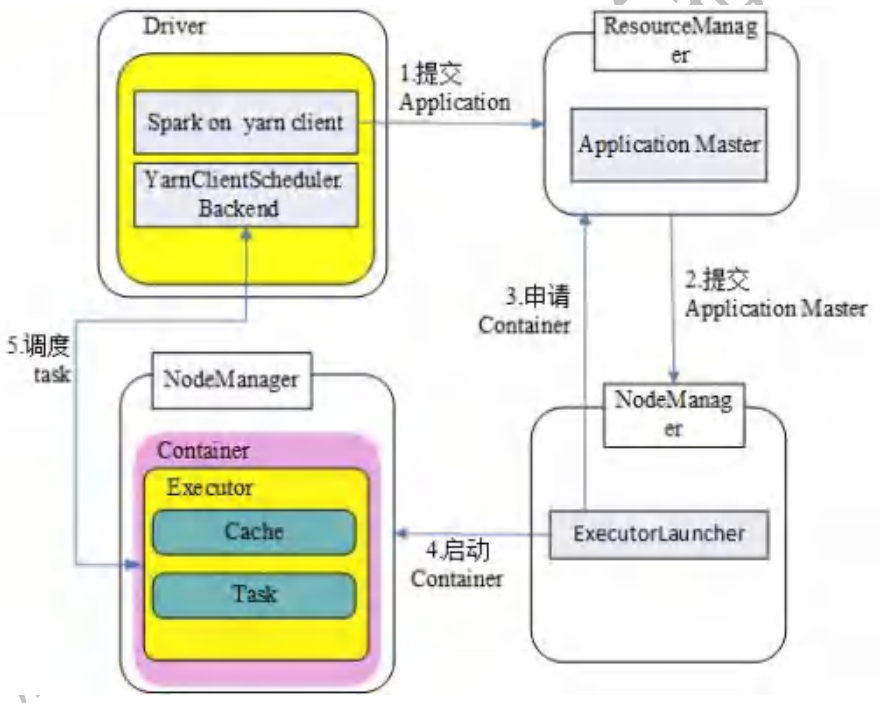
1. 首先由客户端生成 Application 信息，提交给 ResourceManager。
2. ResourceManager 为 Spark Application 分配第一个

Container(ApplicationMaster)，并在该 Container 上启动 Driver。

3. ApplicationMaster 向 ResourceManager 申请资源以运行 Container。
4. ResourceManager 分配 Container 给 ApplicationMaster，ApplicationMaster 和相关的 NodeManager 通讯，在获得的 Container 上启动 Executor，Executor 启动后，开始向 Driver 注册并申请 Task。
5. Driver 分配 Task 给 Executor 执行。
6. Executor 执行 Task 并向 Driver 汇报运行状况。

YARN Client 模式

运行框架如下图所示。



图：Spark on yarn-client 运行框架

Spark on YARN-Client 实现流程：

1. 客户端向 ResourceManager 发送 Spark 应用提交请求，Client 端将启动 ApplicationMaster 所需的所有信息打包，提交给 ResourceManager 上，ResourceManager 为其返回应答，该应答中包含多种信息(如 ApplicationId、可用资源使用上限和下限等)。ResourceManager 收到请求后，会为 ApplicationMaster 寻找合适的节点，并在该节点上启动它。

ApplicationMaster 是 Yarn 中的角色，在 Spark 中进程名字是 ExecutorLauncher。

2. 根据每个任务的资源需求，ApplicationMaster 可向 ResourceManager 申请一系列用于运行任务的 Container。

3. 当 ApplicationMaster (从 ResourceManager 端) 收到新分配的 Container 列表后，会向对应的 NodeManager 发送信息以启动 Container。

4. ResourceManager 分配 Container 给 ApplicationMaster，ApplicationMaster 和相关的 NodeManager 通讯，在获得的 Container 上启动 Executor，Executor 启动后，开始向 Driver 注册并申请 Task。

5. Driver 分配 Task 给 Executor 执行。Executor 执行 Task 并向 Driver 汇报运行状况。

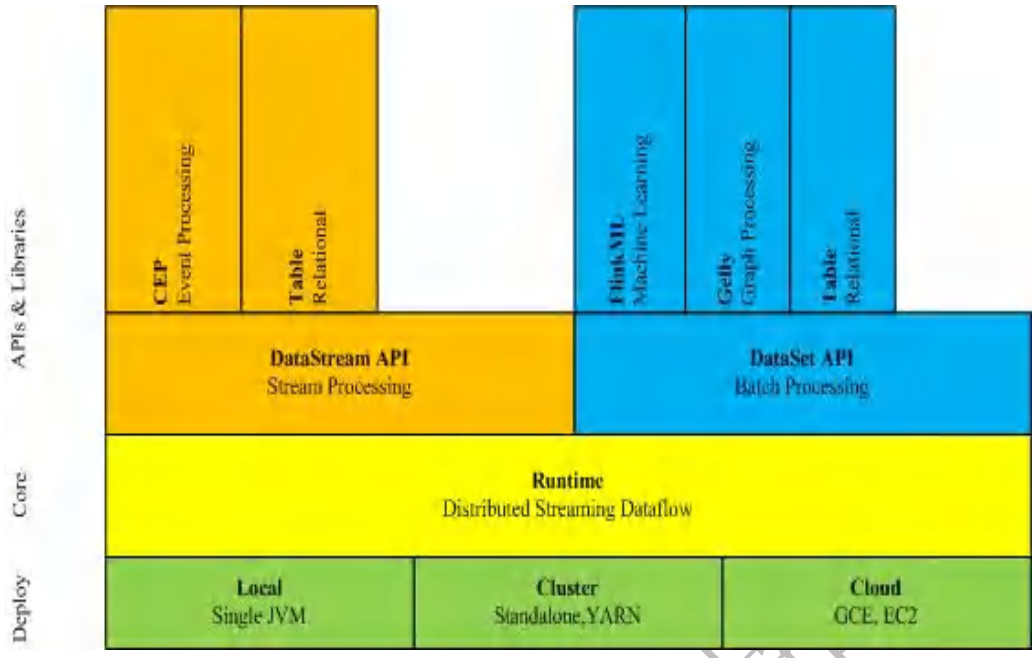
5.8 Flink 流式处理引擎

5.8.1 Flink 简介

Flink 是一个批处理和流处理结合的统一计算框架，其核心是一个提供了数据分发以及并行化计算的流数据处理引擎。它的最大亮点是流处理，是业界最顶级的开源流处理引擎。

Flink 最适合的应用场景是低时延的数据处理（Data Processing）场景：高并发 pipeline 处理数据，时延毫秒级，且兼具可靠性。

Flink 技术栈如下图所示。

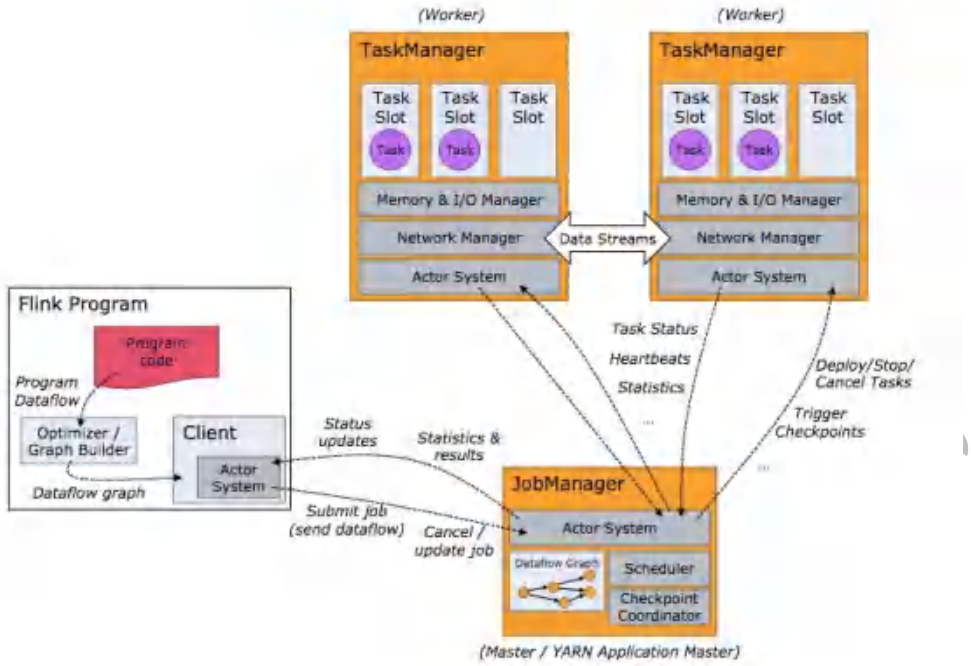


Flink 在当前版本中重点构建如下特性，其他特性继承开源社区，不做增强

- DataStream
- Checkpoint
- 窗口
- Job Pipeline
- 配置表

5.8.2 Flink 架构

Flink 架构如下图所示。



Flink 整个系统包含三个部分：

- Client：Flink Client 主要给用户提供向 Flink 系统提交用户任务（流式作业）的能力。
- TaskManager：Flink 系统的业务执行节点，执行具体的用户任务。TaskManager 可以有多个，各个 TaskManager 都平等。
- JobManager：Flink 系统的管理节点，管理所有的 TaskManager，并决策用户任务在哪些 Taskmanager 执行。JobManager 在 HA 模式下可以有多个，但只有一个主 JobManager。
- Flink 系统提供的关键能力：
- 低时延：提供 ms 级时延的处理能力。
- Exactly Once：提供异步快照机制，保证所有数据真正只处理一次。
- HA：JobManager 支持主备模式，保证无单点故障。
- 水平扩展能力：TaskManager 支持手动水平扩展。

5.8.3 Flink 原理

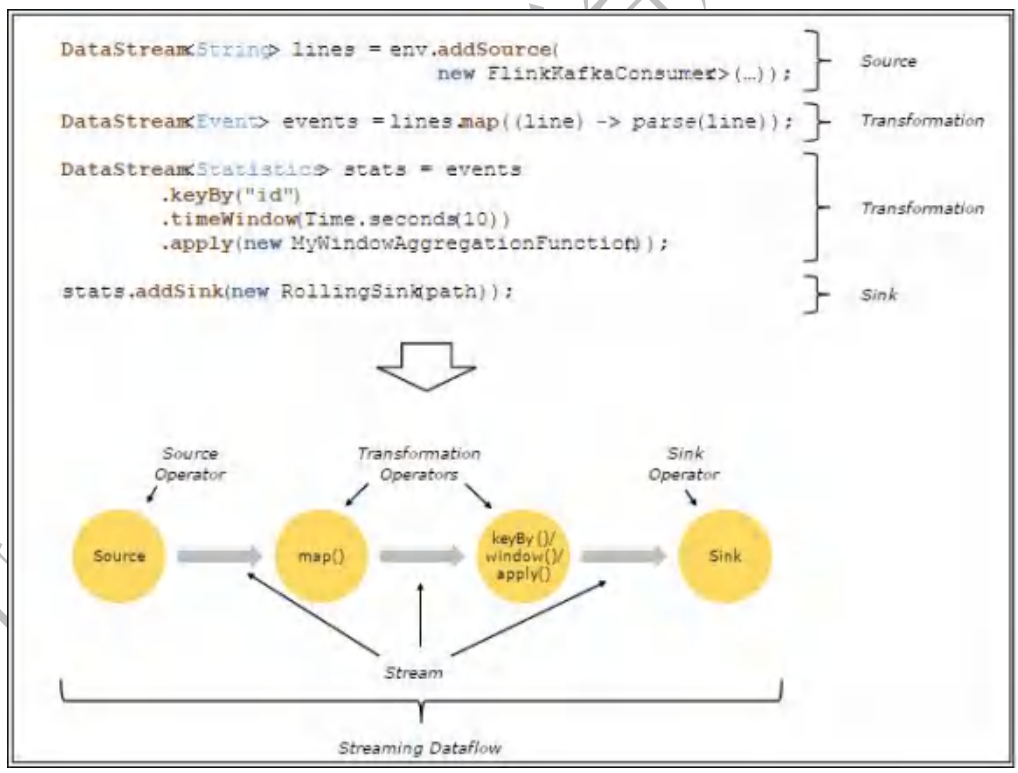
Stream & Transformation & Operator

用户实现的 Flink 程序是由 Stream 和 Transformation 这两个基本构建块组成。

Stream 是一个中间结果数据，而 Transformation 是一个操作，它对一个或多个输入 Stream 进行计算处理，输出一个或多个结果 Stream。

当一个 Flink 程序被执行的时候，它会被映射为 Streaming Dataflow。一个 Streaming Dataflow 是由一组 Stream 和 Transformation Operator 组成，它类似于一个 DAG 图，在启动的时候从一个或多个 Source Operator 开始，结束于一个或多个 Sink Operator。

下图为一个由 Flink 程序映射为 Streaming Dataflow 的示意图。



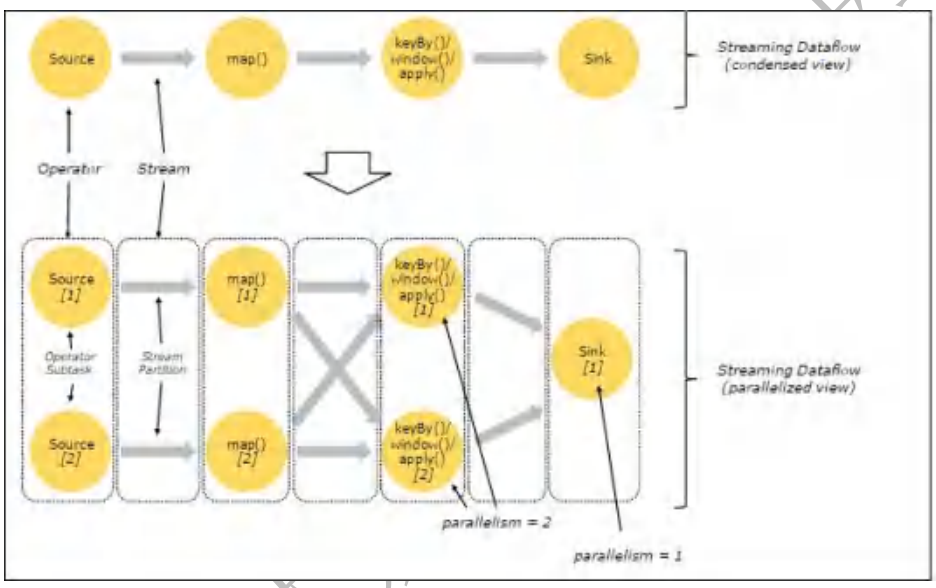
图中“FlinkKafkaConsumer”是一个 Source Operator，Map、KeyBy、TimeWindow、Apply 是 Transformation Operator，RollingSink 是一个 Sink Operator。

Pipeline Dataflow

在 Flink 中，程序是并行和分布式的方式运行。一个 Stream 可以被分成多个 Stream 分区（Stream Partitions），一个 Operator 可以被分成多个 Operator Subtask。

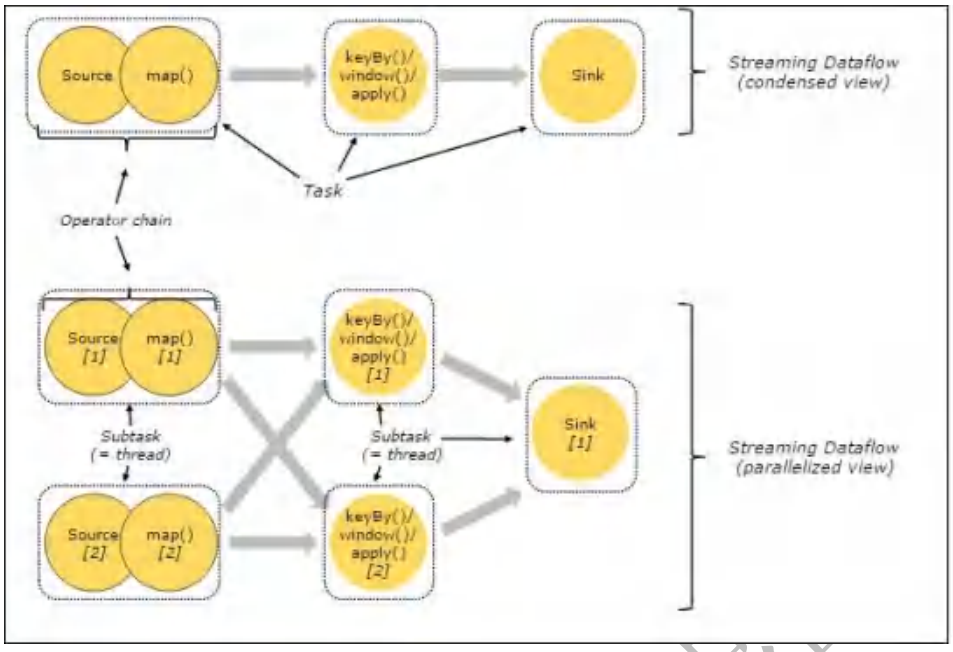
Flink 内部有一个优化的功能，根据上下游算子的紧密程度来进行优化。

紧密度低的算子则不能进行优化，而是将每一个 Operator Subtask 放在不同的线程中独立执行。一个 Operator 的并行度，等于 Operator Subtask 的个数，一个 Stream 的并行度(分区总数)等于生成它的 Operator 的并行度。如下图所示。



Operator Chain

紧密度高的算子可以进行优化，优化后可以将多个 Operator Subtask 串起来组成一个 Operator Chain，实际上就是一个执行链，每个执行链会在 TaskManager 上一个独立的线程中执行，如下图所示。



图中上半部分表示的是将 Source 和 map 两个紧密度高的算子优化后串成一个 Operator Chain，实际上一个 Operator Chain 就是一个大的 Operator 的概念。图中的 Operator Chain 表示一个 Operator，keyBy 表示一个 Operator，Sink 表示一个 Operator，它们通过 Stream 连接，而每个 Operator 在运行时对应一个 Task，也就是说图中的上半部分有 3 个 Operator 对应的是 3 个 Task。

图中下半部分是上半部分的一个并行版本，对每一个 Task 都并行化为多个 Subtask，这里只是演示了 2 个并行度，sink 算子是 1 个并行度。

5.8.4 HA 方案介绍

每个 Flink 集群只有单个 JobManager，存在单点失败的情况。Flink 有 YARN、Standalone 和 Local 三种模式，其中 YARN 和 Standalone 是集群模式，Local 是指单机模式。但 Flink 对于 YARN 模式和 Standalone 模式提供 HA 机制，使集群能够从失败中恢复。本章节主要介绍 YARN 模式下的 HA 方案。

Flink 支持 HA 模式和 Job 的异常恢复。这两项功能高度依赖 ZooKeeper，在使用之前用户需要在“flink-conf.yaml”配置文件中配置 ZooKeeper，配置 ZooKeeper 的参数如下：

```
high-availability:zookeeper
```

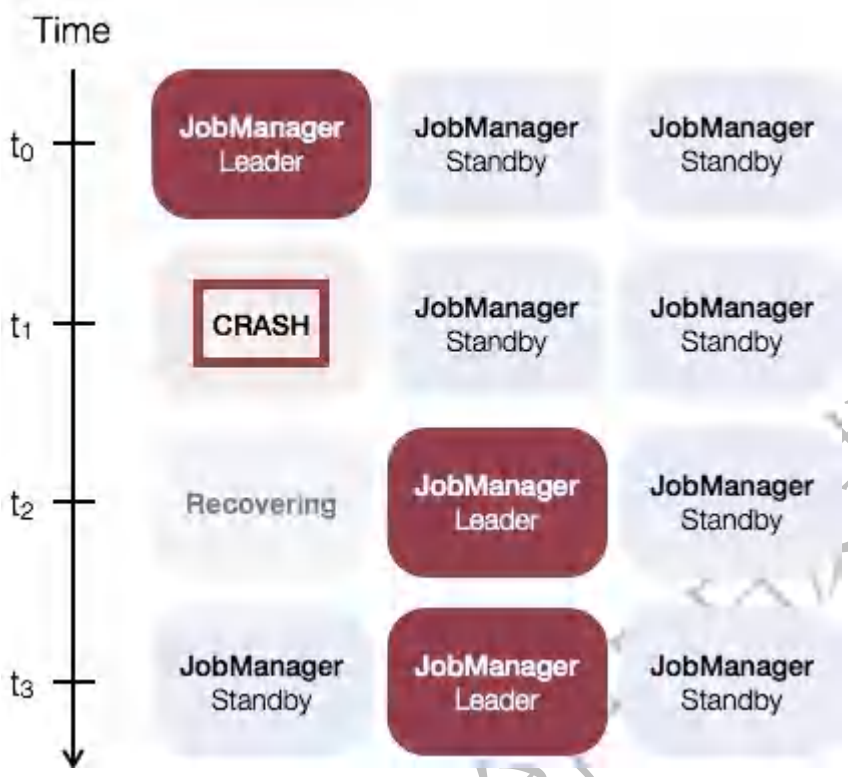
```
high-availability.zookeeper.quorum:localhost:2181

high-availability.storageDir:hdfs:///flink/recovery
```

YARN 模式

Flink 的 JobManager 与 YARN 的 Application Master（简称 AM）是在同一个进程下。YARN 的 ResourceManager 对 AM 有监控，当 AM 异常时，YARN 会将 AM 重新启动，启动后，所有 JobManager 的元数据从 HDFS 恢复。但恢复期间，旧的业务不能运行，新的业务不能提交。ZooKeeper 上还是存有 JobManager 的元数据，比如运行 Job 的信息，会提供给新的 JobManager 使用。对于 TaskManager 的失败，由 JobManager 上 Akka 的 DeathWatch 机制监听处理。当 TaskManager 失败后，重新向 YARN 申请容器，创建 TaskManager。

Standalone 模式

对于 Standalone 模式的集群，可以启动多个 JobManager，然后通过 ZooKeeper 选举出 leader 作为实际使用的 JobManager。该模式下可以配置一个主 JobManager（Leader JobManager）和多个备 JobManager（Standby JobManager），这能够保证当主 JobManager 失败后，备的某个 JobManager 可以承担主的职责。下图为主备 JobManager 的恢复过程。



TaskManager 恢复

对于 TaskManager 的失败，由 JobManager 上 Akka 的 DeathWatch 机制监听处理。当 TaskManager 失败后，由 JobManager 负责创建一个新 TaskManager，并把业务迁移到新的 TaskManager 上。

JobManager 恢复

Flink 的 JobManager 与 YARN 的 Application Master（简称 AM）是在同一个进程下。YARN 的 ResourceManager 对 AM 有监控，当 AM 异常时，YARN 会将 AM 重新启动，启动后，所有 JobManager 的元数据从 HDFS 恢复。但恢复期间，旧的业务不能运行，新的业务不能提交。

Job 恢复

Job 的恢复必须在 Flink 的配置文件中配置重启策略。当前包含三种重启策略：fixed-delay、failure-rate 和 none。只有配置 fixed-delay、failure-rate，job 才可以恢复。另外，如果配置了重启策略为 none，但 job 设置了 checkpoint，默认会将重启策略改为 fixed-delay，且重试次数是配置项“restart-strategy.fixed-delay.attempts”配置为“Integer.MAX_VALUE”。

配置策略的参考如下：

```
restart-strategy: fixed-delay

restart-strategy.fixed-delay.attempts: 3

restart-strategy.fixed-delay.delay: 10 s
```

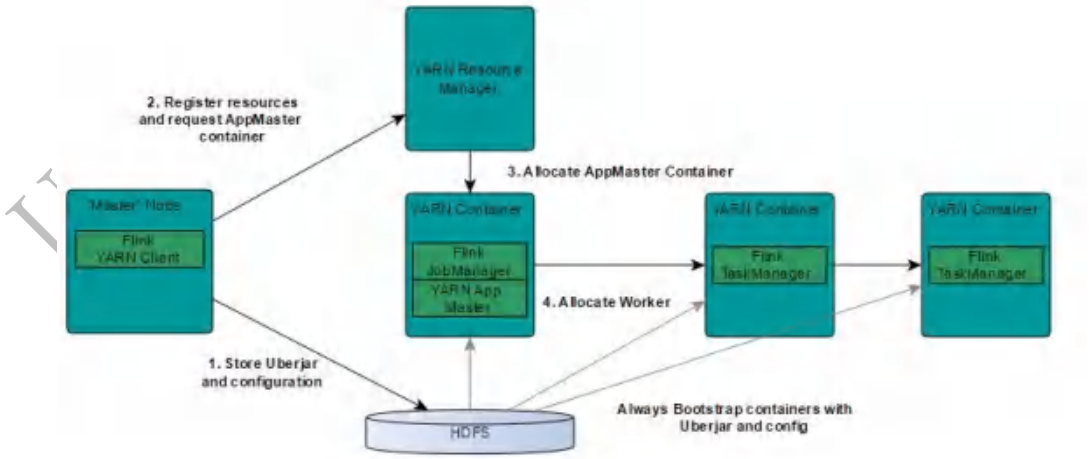
以下场景的异常，都会导致 job 重新恢复：

- 当 JobManager 失败后，所有 Job 会停止，直到新的 JobManager 起来后，所有 Job 恢复。
- 当某一 TaskManager 失败后，这个 TaskManager 上的所有作业都将停止，然后等待有可用资源后重启。
- 当某个 Job 的 Task 失败后，整个 Job 也会重启。

5.8.5 与组件的关系

Flink 支持基于 YARN 管理的集群模式，在该模式下，Flink 作为 YARN 上的一个应用，提交到 YARN 上执行。

Flink 基于 YARN 的集群部署如下图所示。



Flink YARN Client 首先会检验是否有足够的资源来启动 YARN 集群，如果资源足够的话，会将 jar 包、配置文件等上传到 HDFS。

Flink YARN Client 首先与 YARN Resource Manager 进行通信，申请启动 Application Master（以下简称 AM）的 Container，并启动 AM。等所有的 YARN 的 Node Manager 将 HDFS 上的 jar 包、配置文件下载后，则表示 AM 启动成功。

AM 在启动的过程中会和 YARN 的 RM 进行交互，向 RM 申请需要的 Task Manager Container，申请到 Task Manager Container 后，启动 TaskManager 进程。

在 Flink YARN 的集群中，AM 与 Flink JobManager 在同一个 Container 中。AM 会将 JobManager 的 RPC 地址通过 HDFS 共享的方式通知各个 TaskManager，TaskManager 启动成功后，会向 JobManager 注册。

等所有 TaskManager 都向 JobManager 注册成功后，Flink 基于 YARN 的集群启动成功，Flink YARN Client 就可以提交 Flink Job 到 Flink JobManager，并进行后续的映射、调度和计算处理。

5.9 StreamPark

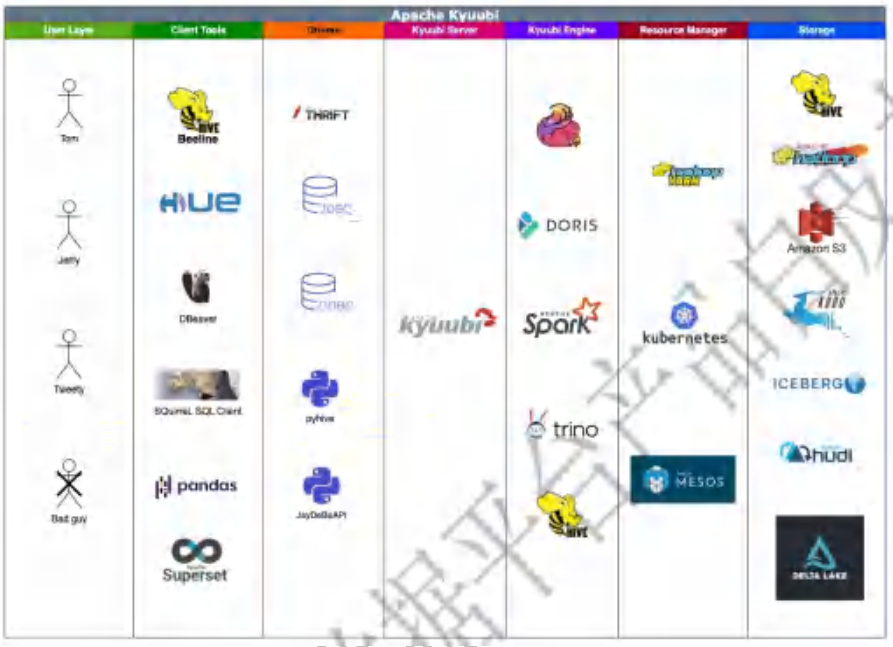
StreamPark 是大数据管理平台扩展的新服务组件，为大数据技术生态提供了实时数据仓库和流批一体的一站式解决方案，简化了 Flink 任务的日常操作和维护。StreamPark 综合了实时数据平台和流式数仓平台的功能，支持低代码的 Flink & Spark 任务托管，融合了诸多最佳实践。此外，它还支持单点登录和不同身份验证系统的集成，提供了集中式的身份验证机制，增强了可定制性和灵活性。

5.10 Kyuubi

5.10.1 Kyuubi 简介

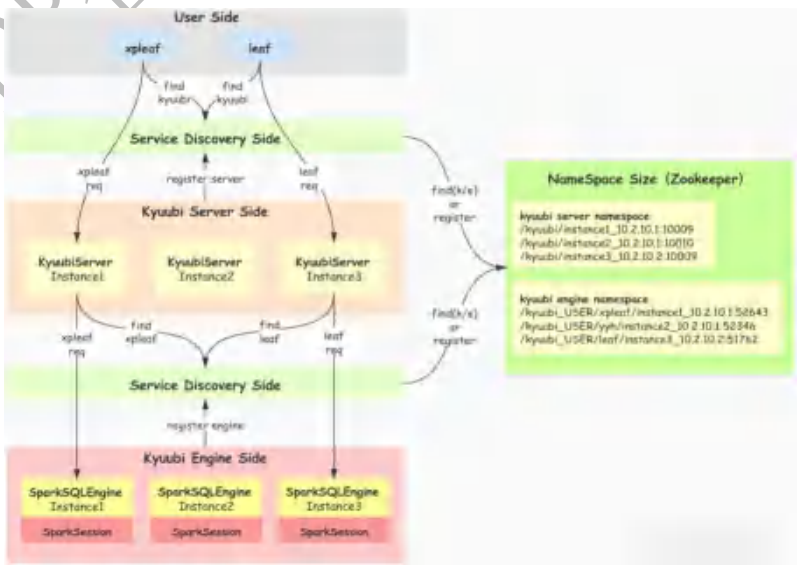
一个开源的高性能的通用 JDBC 和 SQL 执行引擎，Kyuubi 建立在 Spark、Spark、Flink、Hive 和 Trino 等各种现代计算框架之上，以查询来自异构数据源的分布在大量机器上的海量数据集。Kyuubi 弥补了 Spark ThriftServer 在多租户、资源隔离和高可用等方面的不足，可运用于大规模数据分析计算生产级业务场

景。Kyuubi 为最终用户提供简化的数据访问体验，使用户能够使用熟悉的工具专注于业务和数据分析处理，无需关心底层技术和存储细节。同时，Kyuubi 为提供一个易于管理的平台，使管理员能够在不影响用户的情况下升级和维护系统，优化工作负载，并确保集群和数据的安全，包括身份验证、授权和审计功能等。



5.10.2 Kyuubi 架构

下面以 Kyuubi 结合 Spark 构建支持分布式的多租户的数据分析架构，来体现 Kyuubi 的核心架构设计。



用户层包括通过不同方式使用 Kyuubi 的所有用户, 如通过 JDBC 或 beeline 方式使用 Kyuubi 的用户，不同用户及不用客户端均可并行执行数据访问或数据分析业务。

服务发现层依赖于 Zookeeper 实现, 其又分为 Kyuubi Server 层的服务发现和 Kyuubi Engine 层的服务发现。

Kyuubi Server 层主要由多个不同的 Kyuubi Server 实例组成，每个 Kyuubi Server 实例本质上为基于 Apache Thrift 实现的 RPC 服务端，其接收来自用户的请求，但并不会真正执行该请求的相关 SQL 操作，而将以代理转发形式将该请求转发到 Kyuubi Engine 层用户所属的 SparkSQL Engine 实例上。

Kyuubi Engine 层由多个不同的 SparkSQL Engine 实例组成，每个 SparkSQL Engine 实例本质上为基于 Apache Thrift 实现的并且持有一个 SparkSession 实例的 RPC 服务端，其接收来自 Kyuubi Server 实例的请求，并通过 SparkSession 实例来执行。在 Kyuubi 的 USER 共享层级上, 每个 SparkSQL Engine 实例都是租户级别的，即不同的租户会持有不同的 SparkSQL Engine 实例，实现多租户级别的相互资源隔离控制，以及租户任务在不同资源队列中隔离执行。

5.10.3 功能特性

5.10.3.1 统一网关

Kyuubi Server 模块扮演统一网关的角色，通过统一的入口点简化并安全地访问任何集群资源，以部署不同的负载给终端（远程）用户。管理员可以在单个入口点配置、管理和控制对集群的远程访问。Kyuubi 支持多种协议，包括 Hive Thrift 协议、RESTful APIs 和 MySQL，以便用户可以使用兼容这些协议的客户端连接到 Kyuubi Server。

5.10.3.2 多租户及安全控制

Kyuubi 支持端到端多租户。在控制层面，Kyuubi 服务器提供集中式身份验

证层，以降低数据和资源泄露的风险。兼容 LDAP 和 Kerberos，以保护客户端和服务器之间的网络安全。在数据平面上，Kyuubi 引擎使用相同的可信客户端身份来实例化自身。资源获取以及数据和元数据访问都发生在它们自己的引擎内。因此，集群管理者和存储提供商可以轻松保证数据和资源的安全。此外，Kyuubi 还提供引擎授权扩展，将数据安全模型优化到细粒度的行/列级别。

5.10.3.3 高可用性

Kyuubi 采用高可用性（HA）设计方案，确保在指定时间内无故障地持续运行。HA 帮助 Kyuubi 更好的发挥操作性能提升。为了实现在多租户访问的实际生产环境中的高可用性，HA 方案有效地解决了单点故障风险，并帮助实现计划系统维护的零停机时间。Kyuubi 服务器和引擎的故障和系统负载通过指标、日志等方式可见，便于监控和故障检测。

5.10.3.4 高性能

查询性能是实施 Serverless SQL 的关键因素之一。Kyuubi 的查询引擎，支持多种应用以实现高吞吐量，提供可共享的执行运行时以实现低延迟，并进行了服务器端的全局持续优化。Kyuubi 还提供了诸多辅助性能的插件，如 Z-排序、查询优化器等，以进一步提升查询性能。 Serverless SQL 的另一个目标是让最终用户无需或很少关注复杂的性能优化问题，Kyuubi 通过其高性能和高效的设计，使用户能够专注于业务和数据，而不必担心底层性能问题，并为在大规模数据分析查询引擎上提供了可维护性保障。

5.11 Alluxio

为数据分析和 AI 提供统一的数据访问接口，上层计算框架（如 Spark、Flink 等）只需要连接 Alluxio 即可读取和写入存储在底层任意存储系统中的数据；Alluxio 支持数据策略制定，以及数据缓存和预取，提供完整的数据呈现、加速数据访问；通过统一的命名空间，实现跨存储平台、跨数据中心等异构体系敏捷数据集成和编排能力。

5.12 Knox

Knox 是大数据安全代理的开源项目，提供全面的安全和访问控制方案。作为安全代理，它保护集群资源，支持单点登录和身份验证集成，提高用户体验和系统管理效率；而在 API 网关层面，它提供 RESTful API 和协议转换功能，简化应用程序开发和维护；同时还支持 SSL 加密和数据传输安全，保护数据在传输过程中的机密性和完整性；Web 操作界面和管理控制台方便管理员配置和监控安全策略、用户权限和集群访问情况；支持审计、日志记录和多租户配置，满足不同安全需求和环境。

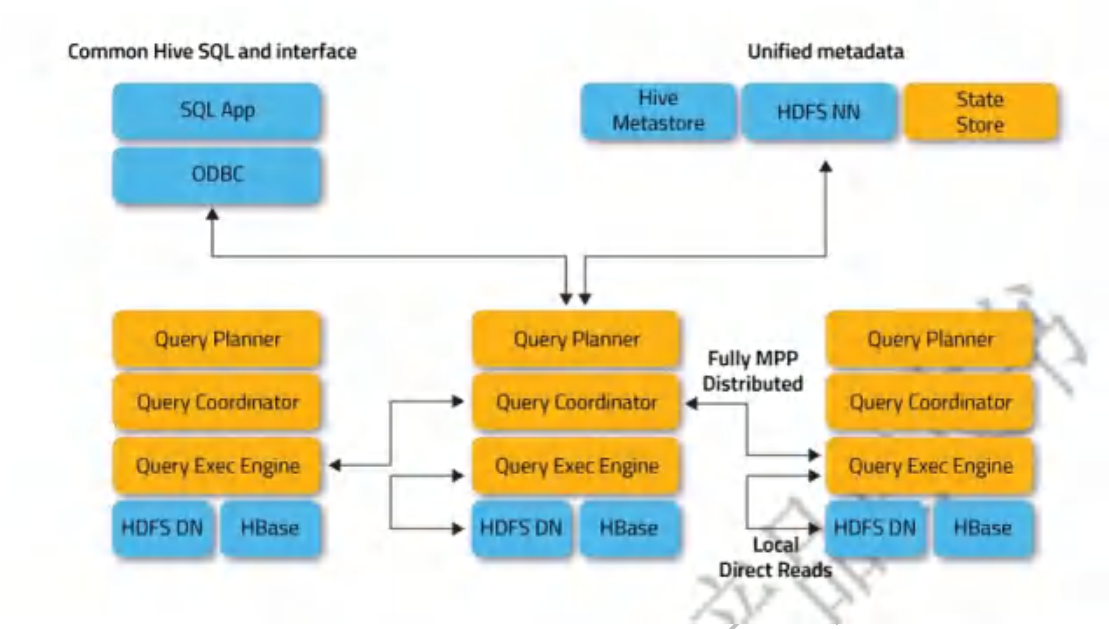
5.13 Impala

5.13.1 概述

Impala 是用于存储在 Hadoop 集群中的数据分布式 MPP SQL 查询引擎。它提高了 Hadoop 上 SQL 查询性能的标准，同时保留了熟悉的用户体验。使用 Impala，可以低延迟和高并发性的 BI/分析读取查询无论是存储在 HDFS 还是 HBase 中的数据，包括 SELECT、JOIN 和聚合函数。此外，Impala 使用与 Hive 相同的元数据、SQL 语法 (Hive SQL)、ODBC 驱动程序和用户界面 (Hue Beeswax)，为面向批处理或实时查询提供熟悉且统一的平台。

目标是将熟悉的 SQL 支持和传统分析数据库的多用户性能与 Hadoop 的可扩展性、灵活性以及安全性和管理扩展相结合。

5.13.2 服务架构



客户端-包括 Hue, ODBC 客户端，JDBC 客户端和 Impala Shell 在内的实体都可以与 Impala 交互。这些接口通常用于发出查询或完成管理任务，例如连接到 Impala、执行某个查询操作。

•Hive Metastore -存储有关 Impala 可用数据的信息。例如，metastore 让 Impala 知道哪些数据库是可用的，以及这些数据库的结构是什么。在通过 Impala SQL 语句创建、删除和修改模式对象、将数据加载到表中等过程中，相关的元数据更改将通过 Impala 1.2 中引入的专用编目服务自动广播到所有 Impala 节点。

•Impala -这个进程，运行在 datanode 上，协调和执行查询。Impala 的每个实例都可以接收、计划和协调来自 Impala 客户端的查询。查询分布在 Impala 节点之间，然后这些节点充当工作人员，执行并行查询片段。

•HBase 和 HDFS—待查询数据的存储。

使用 Impala 执行的查询处理如下:

1. 用户应用程序通过 ODBC 或 JDBC 向 Impala 发送 SQL 查询，它们提供

了标准化的查询接口。用户应用程序可以连接到集群中的任何 impalad。这个 impalad 成为查询的协调器。

2. Impala 解析查询并对其进行分析，以确定跨集群的 impalad 实例需要执行哪些任务。计划执行以获得最佳效率。

3.HDFS、HBase 等服务由本地 impalad 实例访问，提供数据。

4. 每个 impalad 向协调 impalad 返回数据，协调 impalad 将这些结果发送给客户端。

为避免延迟，Impala 实现了一个基于守护进程的分布式架构，守护进程负责查询执行的所有方面，并且与 Hadoop 基础设施的其他部分运行在相同的机器上。结果是性能与商业 MPP 分析 DBMS 相当，甚至超越，具体取决于特定的工作负载。Impala 绕过 MapReduce，通过专门的分布式查询引擎直接访问数据，该引擎与商业并行 RDBMS 中的查询引擎非常相似。结果是性能比 Hive 快一个数量级，具体取决于查询和配置的类型。

Impala 作为一个大规模并行查询执行引擎，可以在现有 Hadoop 集群中的数百台机器上运行。它与底层存储引擎解耦，与传统的关系数据库管理系统不同，在传统的关系数据库管理系统中，查询处理和底层存储引擎是单个紧密耦合系统的组件。

Impala 部署由三个服务组成。Impala 守护进程(impalad)服务负责接受来自客户端进程的查询并在集群中编排它们的执行，以及代表其他 Impala 守护进程执行单个查询片段。当 Impala 守护进程通过管理查询执行在第一个角色中操作时，它被称为该查询的协调器。然而，所有 impalad 守护进程都是对称的；他们可以扮演所有的角色。此属性有助于容错和负载平衡。

集群中的每台机器上都部署了一个 Impala 守护进程，同时也运行着 datanode 进程(底层 HDFS 部署的块服务器)，因此通常每台机器上都有一个 Impala 守护进程。这允许 Impala 利用数据局部性，并且无需使用网络就可以从文件系统中读取块。

Statestore 守护进程(statestored)是 Impala 的元数据发布-订阅服务，它将

集群范围的元数据传播到所有 Impala 进程。Catalog 守护进程(catalogd)作为 Impala 的目录存储库和元数据访问网关。通过目录，Impala 守护进程可以执行 DDL 命令，这些命令反映在外部目录存储(如 Hive Metastore)中。对系统编目的更改通过状态库广播。

5.13.3 功能特性

与查询 Hadoop 数据的其他方法相比，这种方法有很多优点，包括：

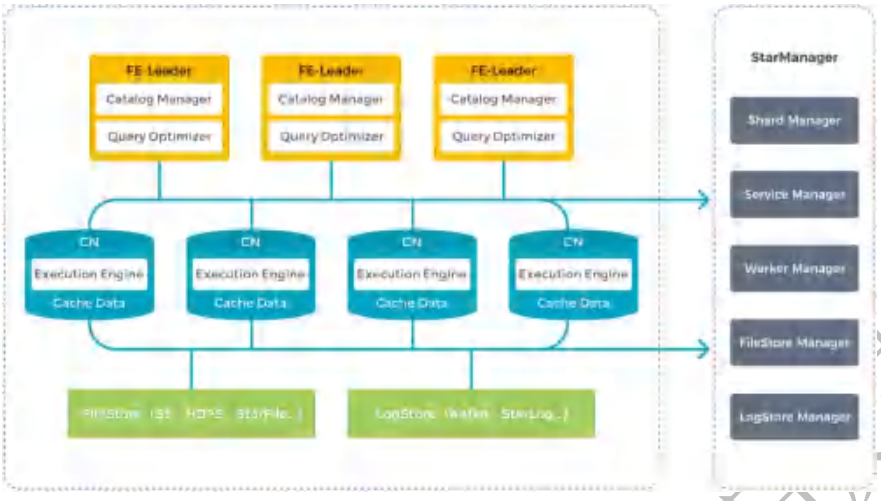
- 由于数据节点上的本地处理，避免了网络瓶颈。
- 可以使用单一、开放和统一的元数据存储。
- 不需要昂贵的数据格式转换，因此不会产生开销。
- 所有数据都可以立即查询，没有 ETL 延迟。
- 所有硬件都用于 Impala 查询和 MapReduce。
- 可横向扩展，实现分布式的查询并发支撑。

5.14 StarRocks

5.14.1 概述

StarRocks 是一款高性能分析型数据仓库，使用向量化、MPP 架构、CBO、智能物化视图、可实时更新的列式存储引擎等技术实现多维、实时、高并发的数据分析。StarRocks 既支持从各类实时和离线的数据源高效导入数据，也支持直接分析数据湖上各种格式的数据。StarRocks 兼容 MySQL 协议，支持标准 SQL 语法，易于对接使用，可使用 MySQL 客户端和常用 BI 工具对接。同时 StarRocks 具备水平扩展、高可用、高可靠、易运维等特性。广泛应用于实时数仓、OLAP 报表、数据湖分析等场景。

5.14.2 系统架构



StarRocks 存算分离新架构简洁，整个系统的核心由 FE（Frontend）、无状态的 CN (Compute Node) 两类进程以及文件系统构成，方便用户部署与维护，支持动态增删计算节点，实现秒级的扩缩容能力。元数据和业务数据都有副本机制，确保整个系统无单点。数据可持久存储在更为可靠和廉价的远程远端对象存储或 HDFS 上，CN 本地磁盘只用于缓存热数据来加速查询。存算分离大幅降低了数据存储成本和扩容成本，有助于实现资源隔离和计算资源的弹性伸缩。

5.14.2.1 节点介绍

FE 是 StarRocks 的前端节点，负责管理元数据、管理客户端连接、进行查询规划、查询调度等工作。每个 FE 节点都会在内存保留一份完整的元数据，这样每个 FE 节点都能够提供无差别的服务。FE 有三种角色：Leader FE，Follower FE 和 Observer FE，区别如下。

FE 角色	元数据读写	Leader 选举
Leader	Leader FE 提供元数据读写服务，Follower 和 Observer 只有读取权限，无写入权限。Follower 和 Observer 将元数据写入请求路由到 Leader，Leader 更新完元数据后，会通过 BDB JE (Berkeley DB Java Edition) 同步给 Follower 和 Observer。必须有半数以上的 Follower 节点同步成功才算作元数据写入成功。	Leader 从 Follower 中自动选出。如果当前 Leader 节点失败，Follower 会发起新一轮选举。
Follower	只有元数据读取权限，无写入权限。通过回放	Follower 参与 Leader 选举，会通过类

FE 角色	元数据读写	Leader 选举
	Leader 的元数据日志来异步同步数据。	Paxos 的 BDBJE 协议自动选举出一个 Leader，必须有半数以上的 Follower 节点存活才能进行选主。
Observer	同 Follower。	Observer 主要用于扩展集群的查询并发能力，可选部署。Observer 不参与选主，不会增加集群的选主压力。

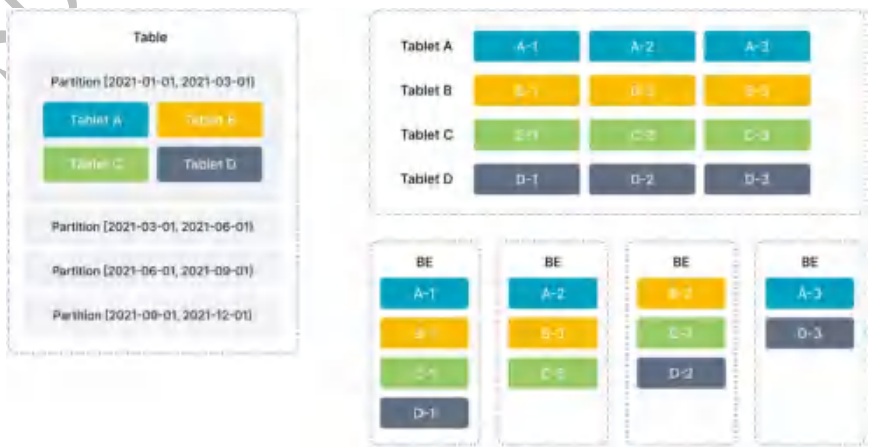
CN 节点负责执行数据导入、查询计算、缓存数据管理等任务，以及负责计算缓存热数据。CN 节点支持动态增删计算节点，实现秒级的扩缩容能力。

5.14.2.2 存储系统

StarRocks 后端存储系统兼容 S3 协议的对象存储系统、Hadoop 分布式文件系统 HDFS，用户可根据需求自由选择。

StarRocks 使用列式存储，采用分区分桶机制进行数据管理。一张表可以被划分成多个分区，一个分区内的数据可以根据一列或者多列进行分桶，将数据切分成多个 Tablet。Tablet 是 StarRocks 中最小的数据管理单元。每个 Tablet 都会以多副本（replica）的形式存储在不同的存储系统中。用户可以自行指定 Tablet 的个数和大小，StarRocks 会管理好每个 Tablet 副本的分布信息。

下图展示了 StarRocks 的数据划分以及 Tablet 多副本机制。表按照日期划分为 4 个分区，第一个分区进一步切分成 4 个 Tablet。每个 Tablet 使用 3 副本进行备份，分布在 3 个不同的存储节点上。



在执行 SQL 语句时，StarRocks 可以对所有 Tablet 实现并发处理，从而充分利用多机、多核提供的计算能力。用户也可以将高并发请求压力分摊到多个物理节点，通过增加物理节点的方式来扩展系统的高并发能力。

Tablet 支持多副本存储（默认三个），多副本能够保证数据存储的高可靠以及服务的高可用。在三副本下，一个节点的异常不会影响服务的可用性，集群的读写服务仍然能够正常进行。增加副本数还有助于提高系统的高并发查询能力。

StarRocks 通过 TabletMeta 描述数据文件在远程存储系统中的存储情况，存储系统节点变化会触发 Tablet 的自动迁移。当节点增加时，一部分 Tablet 会自动均衡到新增的节点，保证数据能够在集群内分布的更加均衡；当节点减少时，待下线机器上的 Tablet 会被自动均衡到其他节点，从而自动保证数据的副本数不变。管理员能够非常容易的实现弹性伸缩，无需手工进行数据的重分布。

5.14.2.3 缓存机制

为了提升存算分离架构的查询性能，StarRocks 构建了分级的数据缓存体系，将最热的数据缓存在内存中，距离计算最近，次热数据则缓存在 CN 节点本地磁盘，冷数据位于远程存储系统中，数据根据访问频率在三级存储中自由流动。

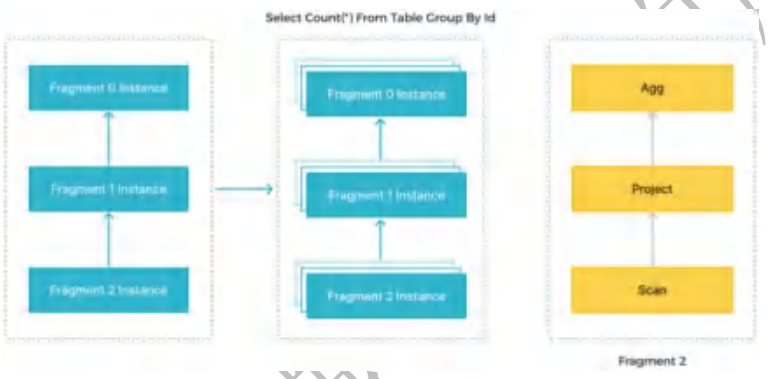
查询时，热数据通常会直接从缓存中命中，而冷数据则需要从远程存储系统中读取并填充至本地缓存，以加速后续访问。通过内存、本地磁盘、远端存储，StarRocks 存算分离构建了一个多层次的数据访问体系，用户可以指定数据冷热规则以更好地满足业务需求，让热数据靠近计算，极大的实现高性能计算和低成本存储。

StarRocks 存算分离的统一缓存允许用户在建表时决定是否开启缓存。如果开启，数据写入时会同步写入本地磁盘以及后端对象存储，查询时，CN 节点会优先从本地磁盘读取数据，如果未命中，再从后端对象存储读取原始数据并同时缓存在本地磁盘。而针对未被缓存的冷数据，StarRocks 可根据应用访问模式，利用数据预读技术、并行扫描技术等手段，减少对于后端对象存储的访问频次，提升查询性能。

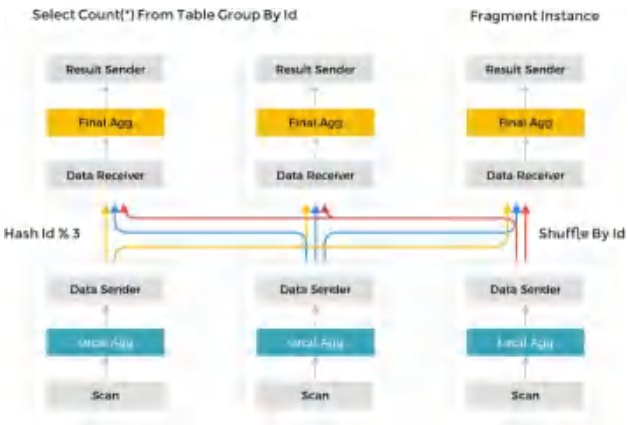
5.14.3 功能特性

5.14.3.1 MPP 分布式执行框架

StarRocks 采用 MPP (Massively Parallel Processing) 分布式执行框架。在 MPP 执行框架中，一条查询请求会被拆分成多个物理计算单元，在多机并行执行。每个执行节点拥有独享的资源（CPU、内存）。MPP 执行框架能够使得单个查询请求可以充分利用所有执行节点的资源，所以单个查询的性能可以随着集群的水平扩展而不断提升。



StarRocks 会将一个查询在逻辑上切分为多个逻辑执行单元（Query Fragment）。按照每个逻辑执行单元需要处理的计算量，每个逻辑执行单元会由一个或者多个物理执行单元来具体实现。物理执行单元是最小的调度单位。一个物理执行单元会被调度到集群某个 CN 节点上执行。一个逻辑执行单元可以包括一个或者多个执行算子，如图中的 Fragment 包括 Scan、Project、Aggregate。每个物理执行单元只处理部分数据。由于每个逻辑执行单元处理的复杂度不一样，所以每个逻辑执行单元的并行度是不一样的，即不同逻辑执行单元可以由不同数目的物理执行单元来具体执行，以提高资源使用率，提升查询速度。



与很多数据分析系统采用的 Scatter-Gather 分布式执行框架不同，MPP 分布式执行框架可以利用更多的资源处理查询请求。在 Scatter-Gather 框架中，只有 Gather 节点能处理最后一级的汇总计算。而在 MPP 框架中，数据会被 Shuffle 到多个节点，并且由多个节点来完成最后的汇总计算。在复杂计算时（比如高基数 Group By，大表 Join 等操作），StarRocks 的 MPP 框架相对于 Scatter-Gather 模式的产品有明显的性能优势。

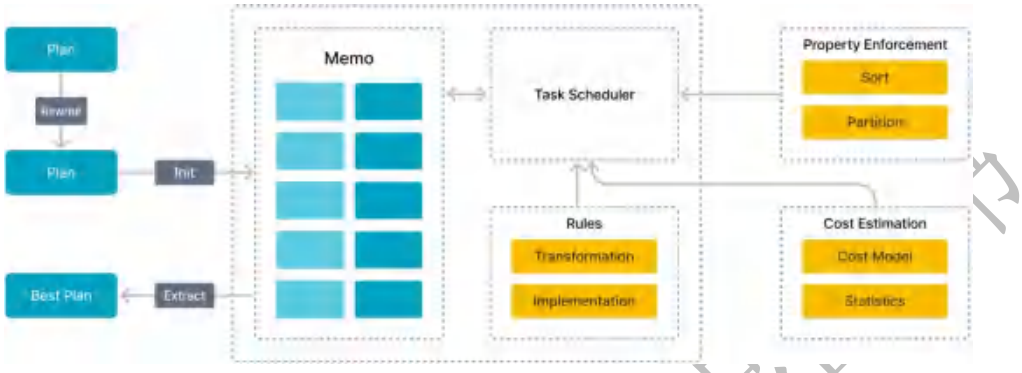
5.14.3.2 全面向量化执行引擎

StarRocks 通过实现全面向量化引擎，充分发挥了 CPU 的处理能力。全面向量化引擎按照列式的方式组织和处理数据。StarRocks 的数据存储、内存中数据的组织方式，以及 SQL 算子的计算方式，都是列式实现的。按列的数据组织也会更加充分的利用 CPU 的 Cache，按列计算会有更少的虚函数调用以及更少的分支判断从而获得更加充分的 CPU 指令流水。另一方面，StarRocks 的全面向量化引擎通过向量化算法充分的利用 CPU 提供的 SIMD（Single Instruction Multiple Data）指令。这样 StarRocks 可以用更少的指令数目，完成更多的数据操作。经过标准测试集的验证，StarRocks 的全面向量化引擎可以将执行算子的性能，整体提升 3~10 倍。

除了使用向量化技术实现所有算子外，StarRocks 还在执行引擎中实现了其他的优化。比如 StarRocks 实现了 Operation on Encoded Data 的技术。对于字符串字段的操作，StarRocks 在无需解码情况下就可以直接基于编码字段完成算子执行，比如实现关联算子、聚合算子、表达式算子计算等。这可以极大的降低

SQL 在执行过程中的计算复杂度。通过这个优化手段，相关查询速度可以提升 2 倍以上。

5.14.3.3 CBO 优化器



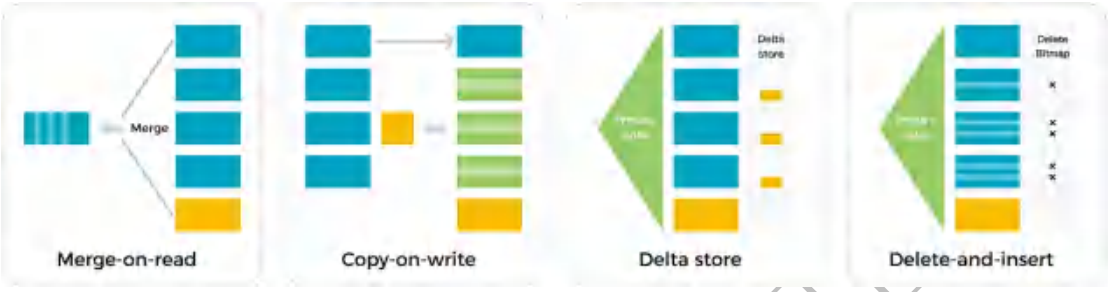
在多表关联查询场景下，仅靠优秀的执行引擎没有办法获得最极致的执行性能。因为这类场景下，不同执行计划的复杂度可能会相差几个数量级。查询中关联表的数目越大，可能的执行计划就越多，在众多的可能中选择一个最优的计划，这是一个 NP-Hard 的问题。只有优秀的查询优化器，才能选择出相对最优的查询计划，从而实现极致的多表分析性能。

StarRocks 设计的基于代价的优化器 CBO（Cost Based Optimizer）是级联（Cascades Like）的，在设计时，针对 StarRocks 的全面向量化执行引擎进行了多项优化和创新。优化器内部实现了公共表达式复用，相关子查询重写，Lateral Join，Join Reorder，Join 分布式执行策略选择，低基数字典优化等重要功能和优化。当前，该优化器已可以完整支持 TPC-DS 99 条 SQL 语句。

5.14.3.4 可实时更新的列式存储引擎

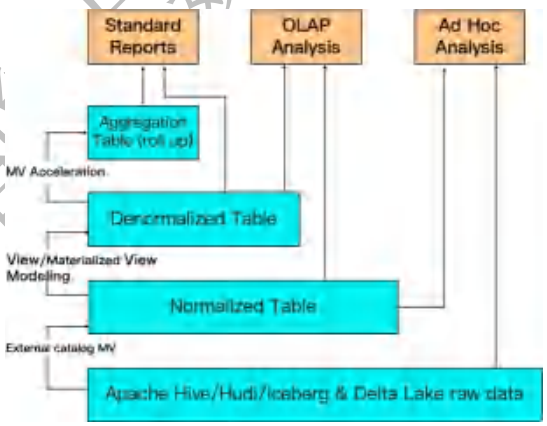
StarRocks 实现了列式存储引擎，数据以按列的方式进行存储。通过这样的方式，相同类型的数据连续存放。一方面，数据可以使用更加高效的编码方式，获得更高的压缩比，降低存储成本。另一方面，也降低了系统读取数据的 I/O 总量，提升了查询性能。此外，在大部分 OLAP 场景中，查询只会涉及部分列。相对于行存，列存只需要读取部分列的数据，能够极大地降低磁盘 I/O 吞吐。

StarRocks 能够支持秒级的导入延迟，提供准实时的服务能力。StarRocks 的存储引擎在数据导入时能够保证每一次操作的 ACID。一个批次的导入数据生效是原子性的，要么全部导入成功，要么全部失败。并发进行的各个事务相互之间互不影响，对外提供 Snapshot Isolation 的事务隔离级别。



StarRocks 存储引擎不仅能够提供高效的 Partial Update 操作，也能高效处理 Upsert 类操作。使用 Delete-and-insert 的实现方式，通过主键索引快速过滤数据，避免读取时的 Sort 和 Merge 操作，同时还可以充分利用其他二级索引，在大量更新的场景下，仍然可以保证查询的极速性能。

5.14.3.5 智能的物化视图



StarRocks 支持用户使用物化视图（materialized view）进行查询加速和数仓分层。不同于一些同类产品的物化视图需要手动和原表做数据同步，StarRocks 的物化视图可以自动根据原始表更新数据。只要原始表数据发生变更，物化视图的更新也同步完成，不需要额外的维护操作就可以保证物化视图能够维持与原表一致。不仅如此，物化视图的选择也是自动进行的。StarRocks 在

进行查询规划时，如果有合适的物化视图能够加速查询，StarRocks 自动进行查询改写(query rewrite)，将查询自动定位到最适合的物化视图上进行查询加速。

StarRocks 的物化视图可以按需灵活创建和删除。用户可以在使用过程中视实际使用情况来判断是否需要创建或删除物化视图。StarRocks 会在后台自动完成物化视图的相关调整。

StarRocks 的物化视图可以替代传统的 ETL 建模流程，用户无需在上游应用处做数据转换，可以在使用物化视图时完成数据转换，简化了数据处理流程。

如上图，最底层 ODS 的湖上数据可以通过 External Catalog MV 来构建 DWD 层的 normalized table；并且可以通过多表关联的物化视图来构建 DWS 层的宽表 (denormalized table)；最上层可以进一步构建实时的物化视图来支撑高并发的查询，提供更加优异的查询性能。

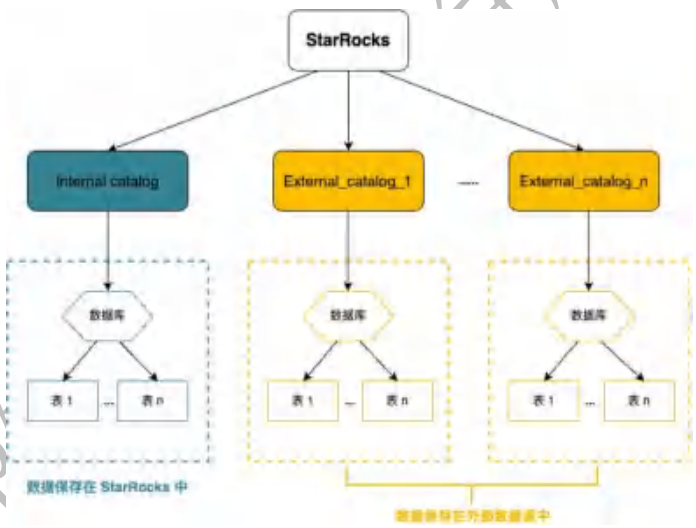
5.14.3.6 数据湖分析

StarRocks 不仅能高效的分析本地存储的数据，也可以作为计算引擎直接分析数据湖中的数据。用户可以通过 StarRocks 提供的 External Catalog，轻松查询存储在 Apache Hive、Apache Iceberg、Apache Hudi、Delta Lake 等数据湖上的数据，无需进行数据迁移。支持的存储系统包括 HDFS、S3、OSS，支持的文件格式包括 Parquet、ORC、CSV。

如上图所示，在数据湖分析场景中，StarRocks 主要负责数据的计算分析，而数据湖则主要负责数据的存储、组织和维护。使用数据湖的优势在于可以使用开放的存储格式和灵活多变的 schema 定义方式，可以让 BI、AI、Adhoc、报表等业务有统一的 single source of truth。而 StarRocks 作为数据湖的计算引擎，可以充分发挥向量化引擎和 CBO 的优势，大大提升了数据湖分析的性能。

5.14.3.7 Catalog 跨数据源数据整合

StarRocks 的目录（Catalog）功能是其数据源管理的一部分，它提供了一个统一的管理界面，用于配置、连接和访问这些不同的数据源，实现跨平台以及跨数据中心的协同计算能力。通过目录，用户可以轻松地整合来自不同来源的数据，进行高效的数据分析和查询。StarRocks 的这一特性特别适合于需要快速、灵活地进行大数据分析的企业和项目。



StarRocks 支持 Catalog（数据目录）功能，实现在一套系统内同时维护内、外部数据，方便用户轻松访问并查询存储在各类外部源的数据。内部数据指保存在 StarRocks (MPPDB) 中的数据；而外部数据则指保存在外部数据源 (如 Hive、Iceberg、Hudi、Delta Lake、HBase、Elasticsearch、JDBC、MPPDB）中的数据。

5.15 Kafka 分布式消息队列

5.15.1 概述

Kafka 是一款高吞吐量、低延迟的分布式消息队列系统。它被设计用于处理大规模的实时数据流，具有可扩展性、可靠性和容错性等特点。Kafka 的主要目标是提供一种高效的方式来传输、存储和处理实时数据流，使数据能够快速而可靠地流动。

Kafka 基于发布-订阅模型，其中消息被发布到一个或多个主题（topic），然后订阅者可以从这些主题订阅消息。它采用分布式的架构，通过分区和复制来提供高可用性和可扩展性。Kafka 的核心设计原则是持久性、高吞吐量和容错性，使其成为处理大规模实时数据的理想选择。

5.15.2 服务架构

Kafka 的服务架构包括以下几个核心组件：

Producer（生产者）：Producer 负责将消息发布到 Kafka 集群。它将消息写入一个或多个主题，然后将消息发送给 Kafka 集群的一个或多个 Broker。Producer 可以选择将消息发送到特定的分区，也可以让 Kafka 根据负载均衡策略自动分配分区。

Broker（代理服务器）：Broker 是 Kafka 集群中的核心组件。它是一个独立的消息服务器，负责接收和存储消息，并处理来自 Producers 和 Consumers 的请求。每个 Broker 都是一个 Kafka 服务器，可以运行在单个机器上或者以集群形式运行。每个 Broker 可以处理多个分区和副本。

Topic（主题）：Topic 是消息的逻辑容器，类似于一个消息队列。Producers 将消息发布到特定的主题，而 Consumers 可以从主题中订阅消息。主题可以分为多个分区，每个分区可以存储一定数量的消息。主题中的消息是有序的，并且可以根据需要保留一段时间。

Partition（分区）：主题可以分为多个分区，每个分区是主题的一个子集。

分区是 Kafka 实现高吞吐量的关键。每个分区在物理上对应一个文件夹，在不同的 Broker 上进行复制。通过将主题划分为多个分区，Kafka 可以实现并行处理和水平扩展。

Consumer（消费者）：Consumer 订阅一个或多个主题，并从 Broker 拉取消息进行消费。Kafka 支持多个消费者组，每个组内可以有多个消费者。Kafka 采用拉模式，即消费者自主拉取消息。消费者可以跟踪自己在每个分区中的读取位置，并以自己的速度消费消息。

ZooKeeper：Kafka 使用 ZooKeeper 来进行协调和配置管理。ZooKeeper 负责管理 Kafka 集群的元数据（如分区分配、Broker 状态等），以及对集群中各个组件的监控和管理。

5.15.3 服务特性

Kafka 具有以下几个重要的服务特性：

高吞吐量：Kafka 通过将数据分区和并行处理来实现高吞吐量。每个分区都可以并行处理，从而提高了整个系统的吞吐量。此外，Kafka 还支持批量写入和读取，进一步提高了数据的处理效率。

低延迟：Kafka 的设计目标之一是提供低延迟的数据传输和处理。它通过使用零拷贝技术、批量处理和索引机制等手段来降低消息的传输和处理延迟。

可靠性：Kafka 具有高度的可靠性和持久性。它使用多个副本机制来保证消息的可靠性。每个分区的消息可以被复制到多个 Broker 上，即使某个 Broker 发生故障，消息仍然可用。Kafka 还支持配置副本的数量和复制因子，以满足不同的可靠性需求。

可扩展性：Kafka 的服务架构是分布式和可扩展的。通过增加 Broker 和分区，可以线性扩展 Kafka 集群的处理能力。Kafka 还支持水平扩展和动态负载均衡，以适应不断增长的数据处理需求。

持久性存储：Kafka 提供持久性的消息存储，它将消息写入磁盘并保留一段时间。这使得 Kafka 能够处理大量的历史数据，并提供回放和再处理的功能。

5.15.4 应用场景

Kafka 在大规模数据处理和实时数据流处理中有广泛的应用场景，包括但不限于以下几个方面：

数据流处理：Kafka 适用于处理大规模的实时数据流。它可以接收和存储来自不同数据源的数据，并使多个消费者能够并行地处理这些数据。Kafka 的高吞吐量和低延迟特性使其成为实时数据处理的理想选择，如实时分析、实时监控、实时报警等。

日志聚合：Kafka 可以作为日志收集和聚合的中间件。它可以接收来自分布式系统的日志数据，并将其存储到集中式存储或分析平台中。通过使用 Kafka，可以方便地将日志数据从多个源集中，进行集中。

消息队列：Kafka 作为分布式消息队列，可以用于解耦和缓冲生产者和消费者之间的通信。生产者将消息发送到 Kafka 的主题中，消费者可以根据需要订阅主题并消费消息。这种异步通信模式能够提高系统的可伸缩性和可靠性，适用于解决生产者和消费者之间的异步通信问题。

数据管道：Kafka 可以用作数据管道，将数据从一个数据源传输到另一个数据目的地。它可以接收来自不同应用程序和系统的数据，并将其传输到目标系统进行处理和存储。通过使用 Kafka 作为数据管道，可以实现数据的实时传输、分发和复制，确保数据的可靠性和一致性。

流式处理：Kafka 可以与流式处理框架（如 Apache Storm、Apache Flink、Apache Samza 等）结合使用，实现实时数据流的处理和分析。Kafka 作为输入和输出的数据源，可以将流式处理的结果发送回 Kafka 主题中，从而构建端到端的实时数据流处理系统。

网络日志收集：Kafka 可以作为网络日志的中转和收集器，接收来自分布式系统中各个节点的日志数据，并将其发送到日志收集系统进行存储和分析。通过使用 Kafka，可以实现高效、可靠的日志收集，支持实时监控和故障排查。

事件驱动架构：Kafka 可以作为事件驱动架构的核心组件，用于处理大规模

的事件流。它可以接收和存储事件，并将其传递给订阅者进行处理。这对于构建实时、可伸缩的事件驱动系统非常有用，如实时数据分析、实时推荐系统、实时广告等。

互联网应用：Kafka 广泛应用于互联网应用中，如用户行为日志收集、消息通知、实时统计分析等。它可以处理大量的实时数据，支持高并发和大规模的用户访问，确保系统的高可用性和稳定性。

5.16 Solr 企业级数据搜索引擎

5.16.1 概述

Solr 是一款开源的企业级数据搜索引擎，基于 Apache Lucene 搜索库。它提供了强大的全文搜索、近实时搜索和分布式搜索的功能，可以帮助组织快速、准确地检索和分析大规模的结构化和非结构化数据。Solr 支持多种数据格式和查询方式，并具有高度可配置和可扩展的特性，使其成为企业级搜索和分析的理想选择。

Solr 的设计目标是提供高性能、可靠性和可扩展性，以满足不同规模和复杂度的搜索需求。它采用分布式架构，支持水平扩展和容错处理，可以处理海量数据和高并发访问。Solr 还提供了丰富的查询语言、强大的过滤和排序功能，以及灵活的插件机制，使用户可以根据自己的需求进行定制和扩展。

5.16.2 服务架构

Solr 的服务架构包括以下几个核心组件：

Core（核心）：Core 是 Solr 的基本单元，包含了索引、配置和查询处理的相关信息。每个 Core 代表了一个独立的索引和一组相关的配置文件。Solr 可以同时管理多个 Core，每个 Core 可以根据需要进行独立的配置和管理。

Solr 服务器：Solr 服务器是运行 Solr 的主机，负责接收和处理来自客户端的请求。它可以运行在单个节点上，也可以以分布式形式运行在多个节点上。Solr

服务器负责协调和管理不同 Core 之间的查询和索引操作。

ZooKeeper：Solr 可以与 ZooKeeper 集成，利用 ZooKeeper 来进行协调和配置管理。ZooKeeper 负责管理 Solr 集群的状态、配置和元数据信息，以及对集群中各个组件的监控和管理。

Client（客户端）：Solr 的客户端负责向 Solr 服务器发送请求，并接收和解析服务器返回的结果。客户端可以使用多种编程语言和协议与 Solr 进行通信，如 HTTP、Java、Python 等。

5.16.3 服务特性

Solr 具有以下几个重要的服务特性：

强大的搜索功能：Solr 提供了丰富的搜索功能，包括全文搜索、字段搜索、范围搜索、模糊搜索等。它支持多种查询语法和查询解析器，可以根据不同的需求和数据类型进行灵活的搜索和过滤。

高性能和近实时搜索：Solr 通过索引和倒排索引的技术，实现了快速的搜索和检索速度。它支持实时索引更新和近实时搜索，使得数据的变更能够几乎立即反映在搜索结果中。

分布式搜索和扩展性：Solr 可以以分布式的方式运行在多个节点上，通过水平扩展来处理大规模的数据和高并发访问。分布式搜索和负载均衡机制使得 Solr 具有高可用性和容错性。

多种数据格式支持：Solr 支持多种数据格式的索引和搜索，包括文本、XML、JSON、CSV 等。它可以处理结构化和非结构化数据，并提供了专门的处理器和过滤器来处理特定的数据类型。

高度可配置和可扩展：Solr 提供了丰富的配置选项和插件机制，使用户可以根据自己的需求进行定制和扩展。用户可以配置索引和查询的行为，添加自定义的处理器和过滤器，以及使用扩展模块和外部组件。

多种部署选项：Solr 支持多种部署选项，包括单机部署、云端部署和容器化部署。它可以与各种大数据平台和工具集成，如 Hadoop、Spark、Hive 等，以便

更好地进行数据分析和搜索。

5.16.4 应用场景

Solr 在企业级搜索和分析中有广泛的应用场景，包括但不限于以下几个方面：

企业搜索：Solr 可以作为企业内部搜索引擎，用于构建和管理企业的内部搜索系统。通过将企业数据索引到 Solr 中，员工可以快速、准确地搜索和访问企业的文档、知识库、产品信息等。

电子商务搜索：Solr 在电子商务领域中得到广泛应用。它可以用于构建产品搜索和推荐系统，使用户能够快速找到所需的产品，并提供相关的推荐和过滤功能。Solr 还支持排序、分页和过滤等功能，满足电子商务网站对搜索的高性能和精确性要求。

日志和事件分析：Solr 可以用于对日志和事件数据进行实时搜索和分析。通过将日志数据索引到 Solr 中，可以快速检索和过滤特定的日志记录，并进行实时的事件监控和分析。这对于故障排查、安全监控和业务分析非常有用。

网站搜索和内容管理：Solr 可以用于构建网站搜索功能，使用户能够快速搜索和访问网站的内容。它可以与内容管理系统（CMS）集成，提供高效的内容搜索和检索功能，以提高网站的用户体验和搜索引擎优化。

数据分析和挖掘：Solr 可以作为数据分析和挖掘的工具，帮助用户从大规模数据集中发现有价值的信息。通过对数据进行索引和搜索，可以执行复杂的查询和聚合操作，进行数据分析和统计。Solr 还支持实时数据流的处理和分析，使用户能够对数据进行实时的探索和挖掘。

地理空间搜索：Solr 提供了对地理空间数据的索引和搜索功能。它支持地理坐标的索引和查询，并提供了丰富的地理空间查询语法和过滤器。这使得 Solr 在地理信息系统（GIS）和位置服务中具有广泛的应用，如地图搜索、位置推荐、路线规划等。

信息检索和推荐系统：Solr 可以用于构建信息检索和推荐系统，帮助用户快

速找到所需的信息。通过将文档、文章、新闻等内容索引到 Solr 中，可以提供准确和相关的搜索结果。Solr 还支持基于内容和用户行为的推荐系统，根据用户的兴趣和偏好推荐相关的内容。

日程安排和会议管理：Solr 可以用于构建日程安排和会议管理系统，帮助用户管理和组织日程、会议和活动。通过将日程和会议信息索引到 Solr 中，可以快速搜索和过滤特定的日程和会议，提供预订、提醒和日程管理的功能。

法律和知识管理：Solr 在法律和知识管理领域有广泛的应用。它可以用于构建法律文档和知识库的搜索和检索系统，帮助用户快速找到相关的法律文件、判例和知识资料。Solr 的全文搜索和高度可配置性使其非常适合处理大量的法律和知识文档。

科学研究和学术搜索：Solr 可以用于科学研究和学术搜索，帮助研究人员和学者快速访问和检索相关的科研论文、期刊和学术资料。通过将科研文献索引到 Solr 中，可以进行高级的全文搜索、引用检索和领域专属搜索。

5.17 HUE 数据开发

5.17.1 概述

HUE（Hadoop User Experience）是一款开源的 Web 界面工具，旨在简化大数据处理和分析的开发体验。HUE 提供了一个用户友好的界面，使用户无需编写复杂的代码即可轻松地进行数据开发、查询和可视化。HUE 支持多种大数据组件和技术，如 Hadoop、Hive、Impala、Spark 等，使用户可以在一个集成的环境中进行各种数据处理操作。

5.17.2 服务架构

HUE 的服务架构通常由以下几个核心组件组成：

- 前端服务器：HUE 的用户界面通过前端服务器提供给用户访问。前端服务器负责处理用户请求、呈现界面、与后端服务通信等。
- 后端服务：HUE 的后端服务负责处理数据开发的请求。它包括各种支

持的数据处理引擎，如 Hive 服务器、Impala、Spark 等。后端服务通过与这些引擎进行通信，执行用户提交的数据开发任务。

- 数据存储：HUE 通常需要与底层的数据存储系统进行交互，如 HDFS（Hadoop 分布式文件系统）、Hive 数据仓库等。这些数据存储系统提供了数据的存储和访问功能。
- 元数据存储：HUE 需要存储和管理与数据开发相关的元数据信息，如表结构、查询历史等。元数据存储通常使用关系型数据库来实现，如 MySQL。

5.17.3 服务特性

HUE 提供了多种数据开发相关的功能和特性，以帮助用户更高效地进行数据处理和分析：

- 交互式查询：HUE 支持用户通过界面轻松地执行交互式查询，如使用 Hive、Impala 等。用户可以编写 SQL 查询语句，提交并立即获取结果，无需编写和运行复杂的命令行代码。
- 工作流程管理：HUE 提供工作流程管理功能，允许用户创建和管理复杂的数据处理工作流。用户可以定义工作流中的任务、依赖关系和执行顺序，以便自动化数据处理流程。
- 数据可视化：HUE 提供直观的数据可视化功能，用户可以通过图表、图形和报表等方式呈现数据结果。这有助于用户更好地理解和分析数据，发现潜在的模式和趋势。
- 安全和权限管理：HUE 支持安全和权限管理功能，用户可以设置访问控制规则，限制用户对数据和功能的访问权限。这有助于保护敏感数据和确保平台的安全性。
- 第三方集成：HUE 支持与第三方工具和服务的集成，如版本控制系统（如 Git）、调度器（如 Oozie）等。这提供了更大的灵活性和扩展性，使用户可以根据自己的需求集成其他工具和服务。

5.17.4 应用场景

HUE 的功能和特性使其在多个数据开发场景中得到广泛应用：

- 数据查询和探索：HUE 提供了交互式查询和数据可视化功能，使用户可以通过简单的界面执行查询操作，探索数据集，发现数据中的模式和关联。
- 数据处理工作流：HUE 的工作流程管理功能使用户可以创建和管理复杂的数据处理工作流。用户可以定义任务和依赖关系，以实现自动化的数据处理和分析流程。
- 数据可视化和报表：HUE 的数据可视化功能允许用户创建直观的图表和报表，以呈现数据的洞察力。这在数据分析和决策支持方面非常有用。
- 数据开发和调试：HUE 提供了易于使用的界面和工具，使数据开发人员能够快速编写、调试和优化数据处理代码。这可以提高开发效率和质量。
- 多用户协作：HUE 支持多用户环境下的协作和权限管理。团队成员可以共享和协作开发任务，并根据需要设置不同的访问权限。

5.17.5 DolphinSchedule 作业编排与调度系统

5.17.6 概述

Apache DolphinScheduler 是一个分布式、易扩展的可视化 DAG 工作流调度平台。致力于解决数据处理流程中错综复杂的依赖关系，敏捷地用低代码创建高性能工作流。

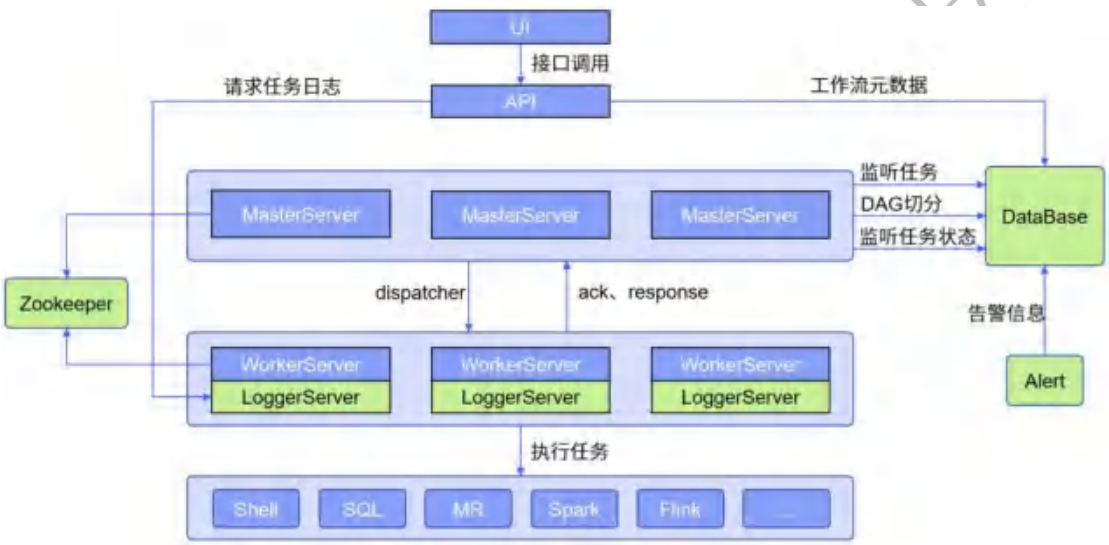
DolphinScheduler 的核心角色包括：

- MasterServer 采用分布式无中心设计理念， MasterServer 主要负责 DAG 任务切分、任务提交、任务 监控，并同时监听其它 MasterServer 和 WorkerServer 的健康状态。

- WorkerServer 也采用分布式无中心设计理念，WorkerServer 主要负责任务的执行和提供日志服务。

  ■ZooKeeper 服务，系统中的 MasterServer 和 WorkerServer 节点都通过 ZooKeeper 来进行集群管理和容错。

- Alert 服务，提供告警相关服务。
- API 接口层，主要负责处理前端 UI 层的请求。
- UI 系统的前端页面，提供系统的各种可视化操作界面。



5.18 Flume 实时数据采集

5.18.1 概述

Flume 是一个可靠、可扩展的分布式系统，用于高效地收集、聚合和移动大量的数据。它通常用于数据同步与采集，将数据从多个源（如日志文件、传感器、消息队列等）收集到目标存储或分析平台（如 Hadoop、Kafka、Hive 等）中。Flume 提供了一个可靠的机制来确保数据的可靠传输和持久性存储，以满足大规模数据处理的需求。

5.18.2 服务架构

Flume 的服务架构通常包括以下几个核心组件：

Source：源组件是数据的起点，负责从不同的数据源收集数据。Flume 提供了多种预置的源组件，如 Avro、Exec、HTTP、Kafka 等，也可以根据需要自定义源组件。

Channel：通道组件是源和目标之间的缓冲区，用于存储临时的事件数据。通道可以将源组件收集到的数据进行缓冲，并提供可靠的数据传输机制。Flume 提供了多种预置的通道组件，如 Memory、JDBC、Kafka 等。

Sink：汇组件是数据的目标，负责将数据发送到指定的存储或分析平台。Flume 提供了多种预置的汇组件，如 HDFS、Hive、Kafka、Elasticsearch 等，也可以根据需要自定义汇组件。

Agent：Agent 是 Flume 的运行实例，包含了源、通道和汇组件。Agent 负责协调源、通道和汇之间的数据传输和流动。

多 Agent 架构：Flume 还支持多 Agent 架构，通过配置多个 Agent，可以实现数据的分布式采集和处理。多 Agent 架构提供了更高的可伸缩性和容错性，以满足大规模数据处理的需求。

5.18.3 服务特性

Flume 具有以下几个重要的服务特性：

可靠性：Flume 提供了可靠的数据传输机制，确保数据从源端到目标端的可靠传输。它使用事务机制和可靠的存储机制来处理故障和失败情况，确保数据不会丢失。

扩展性：Flume 的服务架构支持横向扩展，可以通过增加 Agent 来增加系统的处理能力。它还支持多 Agent 架构，使数据采集和处理可以并行进行，提高了系统的吞吐量和性能。

灵活性：Flume 提供了灵活的配置选项和可扩展性，可以根据需要自定义源、通道和汇组件。这使得 Flume 适用于各种不同的数据源和目标存储，满足各种数据采集和同步的需求。

可定制性：Flume 可以通过编写自定义的插件来扩展其功能和特性。这允许

用户根据自己的需求来定制数据处理逻辑，满足特定的数据处理需求。

5.18.4 应用场景

Flume 在数据同步与采集方面有广泛的应用场景，包括但不限于以下几个方面：

日志收集：Flume 可用于收集和聚合分布式系统中的日志数据。通过配置合适的源组件，可以实时收集分布式环境中的日志，并将其发送到中央存储或分析平台，以进行日志分析、故障排查等。

数据流水线：Flume 可以构建数据流水线，实现数据从源到目标的实时传输和处理。它可以将数据从各种数据源（如传感器、消息队列、API 等）采集到目标存储或分析平台，以便进行后续的数据处理和分析。

数据摄取：Flume 可用于实时采集和传输大量的数据，如网络数据、传感器数据、应用日志等。它提供了高吞吐量和低延迟的数据采集机制，适用于需要实时处理和分析大规模数据的场景。

数据同步：Flume 可以用于数据的异地复制和同步。通过配置合适的源和汇组件，可以实现数据从一个位置到另一个位置的可靠和高效传输，以实现数据备份、灾备和数据同步的需求。

实时数据处理：Flume 与实时数据处理引擎（如 Spark Streaming、Flink 等）的结合，可以实现实时数据的采集、传输和处理。这对于需要实时响应和即时处理的场景非常有用，如实时监控、实时分析等。

5.19 HBase 的 SQL 驱动 Phoenix

Phoenix 是构建在 HBase 上的一个 SQL 层，能让我们用标准的 JDBC APIs 而不是 HBase 客户端 APIs 来创建表，插入数据和对 HBase 数据进行查询。Phoenix 完全使用 Java 编写，作为 HBase 内嵌的 JDBC 驱动。Phoenix 查询引擎会将 SQL 查询转换为一个或多个 HBase 扫描，并编排执行以生成标准的 JDBC 结果集。

5.20 Ranger 权限管理系统

Apache Ranger 提供一个集中式安全管理框架，并解决授权和审计。它可以对 Hadoop 生态的组件如 HDFS、Yarn、Hive、Hbase 等进行细粒度的数据访问控制。通过操作 Ranger 控制台,管理员可以轻松的通过配置策略来控制用户访问权限。

5.21 Zookeeper 分布式协调器

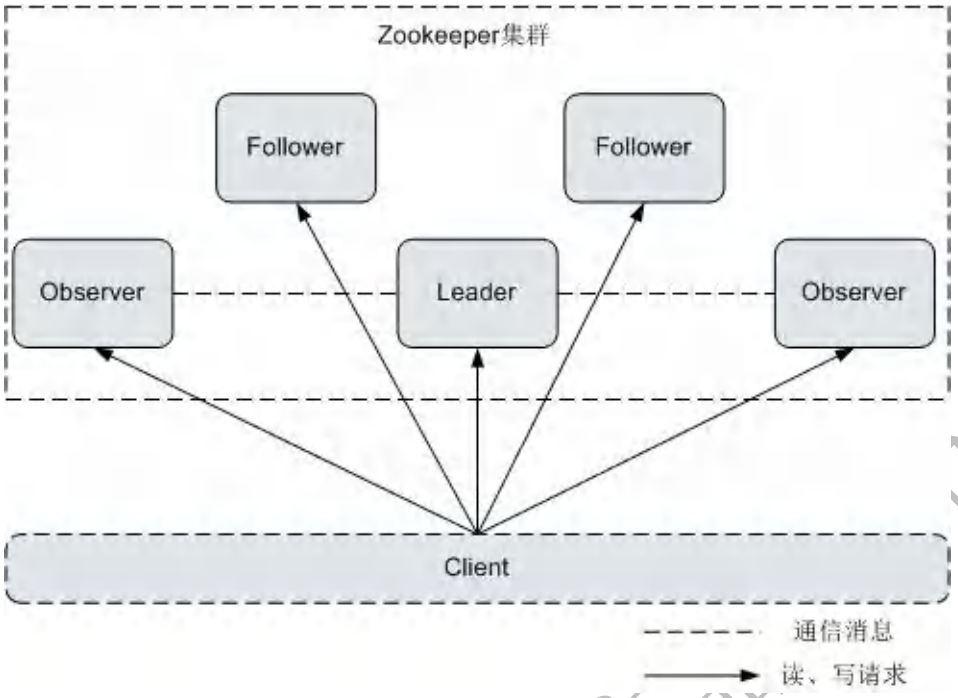
5.21.1 简介

ZooKeeper 是一个分布式、高可用性的协调服务。在大数据产品中主要提供两个功能：

- 帮助系统避免单点故障，建立可靠的应用程序。
- 提供分布式协作服务和维护配置信息。

5.21.2 结构

ZooKeeper 集群中的节点分为三种角色：Leader、Follower 和 Observer，其结构和相互关系如下图所示。通常来说，需要在集群中配置奇数个（2N+1）ZooKeeper 服务，至少（N+1）个投票才能成功的执行写操作。



图：ZooKeeper 结构

图中各部分的功能说明如下表所示。

表：结构图说明

名称	描述
Leader	在 ZooKeeper 集群中只有一个节点作为集群的领导者，由各 Follower 通过 ZooKeeper Atomic Broadcast(ZAB)协议选举产生，主要负责接收和协调所有写请求，并把写入的信息同步到 Follower 和 Observer。
Follower	Follower 的功能有两个： ● 每个 Follower 都作为 Leader 的储备，当 Leader 故障时重新选举 Leader，避免单点故障。 ● 处理读请求，并配合 Leader 一起进行写请求处理。

名称	描述
Observer	Observer 不参与选举和写请求的投票，只负责处理读请求、并向 Leader 转发写请求，避免系统处理能力浪费。
Client	ZooKeeper 集群的客户端，对 ZooKeeper 集群进行读写操作。例如 HBase 可以作为 ZooKeeper 集群的客户端，利用 ZooKeeper 集群的仲裁功能，控制其 HMaster 的“Active”和“Standby”状态。

如果集群启用了安全服务，在连接 ZooKeeper 时需要进行身份认证，认证方式有以下两种：

- keytab 方式：需要从管理员处获取一个“人机”用户，用于登录 Hadoop 集群平台并通过认证，并且获取到该用户的 keytab 文件。
- 票据方式：从管理员处获取一个“人机”用户，用于后续的安全登录，开启 Kerberos 服务的 renewable 和 forwardable 开关并且设置票据刷新周期，开启成功后重启 kerberos 及相关组件。

5.21.3 原理

写请求

1. Follower 或 Observer 接收到写请求后，转发给 Leader。
2. Leader 协调各 Follower，通过投票机制决定是否接受该写请求。
3. 如果超过半数以上的 Leader、Follower 节点返回写入成功，那么 Leader 提交该请求并返回成功，否则返回失败。
4. Follower 或 Observer 返回写请求处理结果。

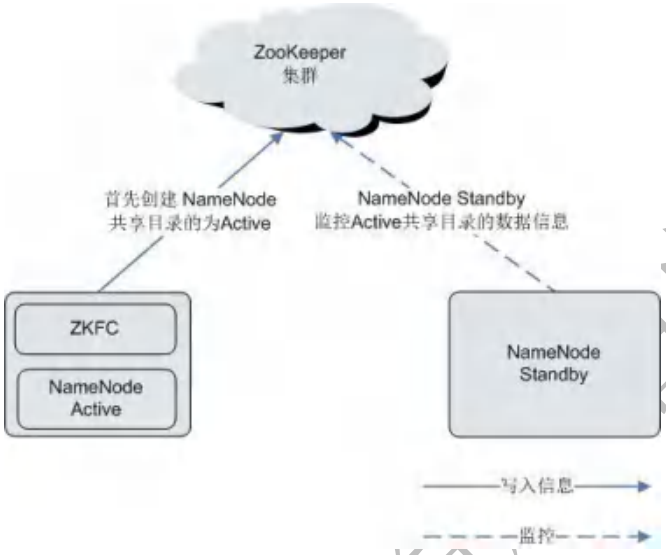
只读请求

客户端直接向 Leader、Follower 或 Observer 读取数据。

5.21.4 与组件的关系

ZooKeeper 和 HDFS 的配合关系

ZooKeeper 与 HDFS 的关系如下图所示。



图：ZooKeeper 和 HDFS 的关系

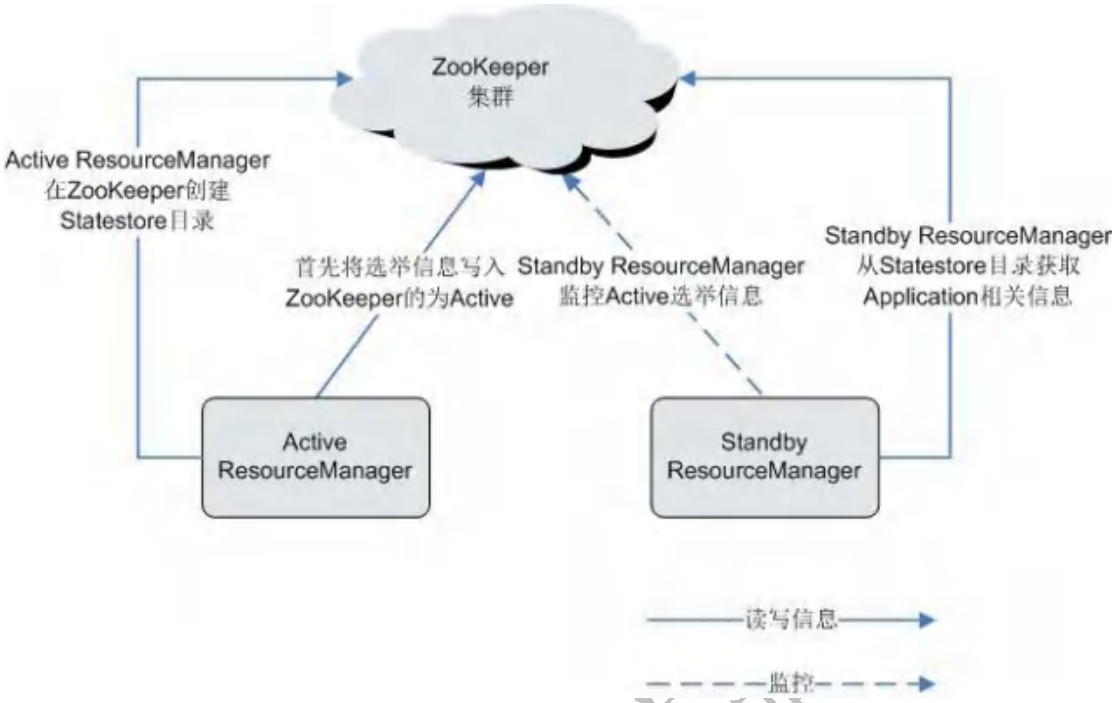
ZKFC（ZKFailoverController）作为一个 ZooKeeper 集群的客户端，用来监控 NameNode 的状态信息。ZKFC 进程仅在部署了 NameNode 的节点中存在。HDFS NameNode 的 Active 和 Standby 节点均部署有 zkfc 进程。

HDFS NameNode 的 ZKFC 连接到 ZooKeeper，把主机名等信息保存到 ZooKeeper 中，即“/hadoop-ha”下的 znode 目录里。先创建 znode 目录的 NameNode 节点为主节点，另一个为备节点。HDFS NameNode Standby 通过 ZooKeeper 定时读取 NameNode 信息。

当主节点进程异常结束时，HDFS NameNode Standby 通过 ZooKeeper 感知“/hadoop-ha”目录下发生了变化，NameNode 会进行主备切换。

ZooKeeper 和 YARN 的配合关系

ZooKeeper 与 YARN 的关系如下图所示。



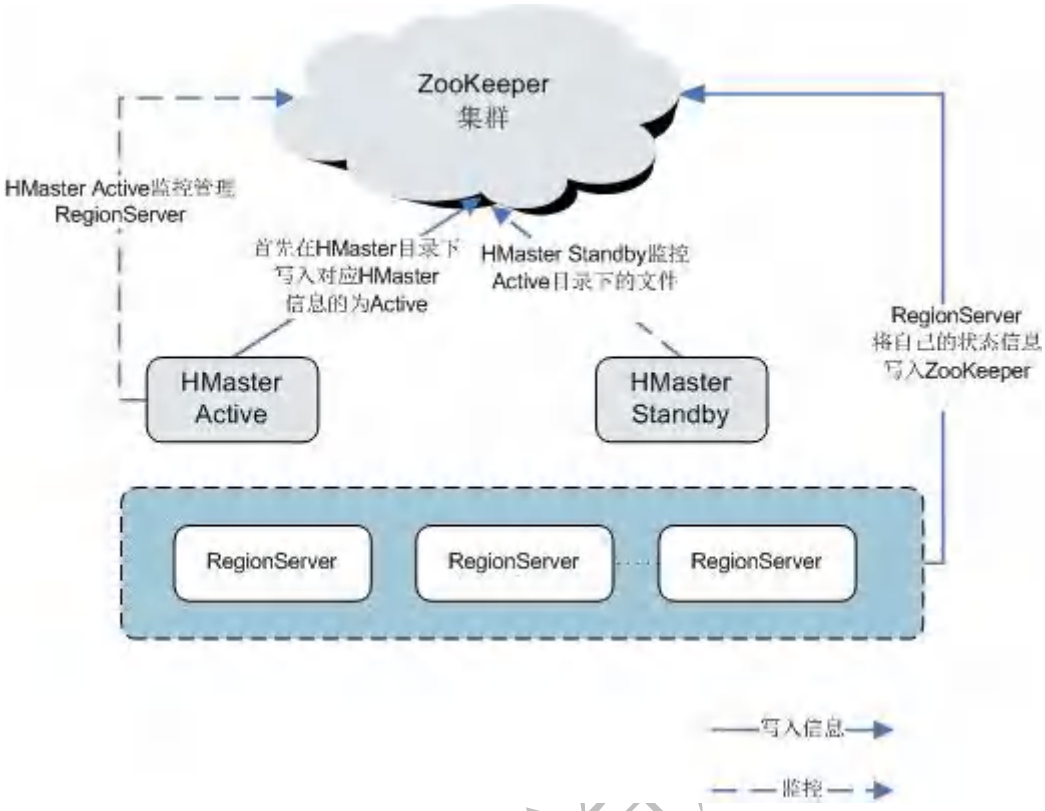
图：ZooKeeper 与 YARN 的关系

在系统启动时，ResourceManager 会尝试把选举信息写入 ZooKeeper，第一个成功把写入 ZooKeeper 的 ResourceManager 被选举为 Active ResourceManager，另一个为 Standby ResourceManager。Standby ResourceManager 定时去 ZooKeeper 监控 Active ResourceManager 选举信息。

Active ResourceManager 还会在 ZooKeeper 中创建 Statestore 目录，存储 Application 相关信息。当 Active ResourceManager 产生故障时，Standby ResourceManager 会从 Statestore 目录获取 Application 相关信息，恢复数据。

ZooKeeper 和 HBase 的配合关系

ZooKeeper 与 HBase 的关系如下图所示。



图：ZooKeeper 和 HBase 的关系

1. HRegionServer 以 Ephemeral node 的方式注册到 ZooKeeper 中。其中 ZooKeeper 存储 HBase 的如下信息：HBase 元数据、HMaster 地址。
2. HMaster 通过 ZooKeeper 随时感知各个 HRegionServer 的健康状况，以便进行控制管理。
3. HBase 也可以部署多个 HMaster，类似 HDFS NameNode，当 HMaster 主节点出现故障时，HMaster 备用节点会通过 ZooKeeper 获取主 HMaster 存储的整个 HBase 集群状态信息。即通过 ZooKeeper 实现避免 HBase 单点故障问题的问题。

5.22 Canal

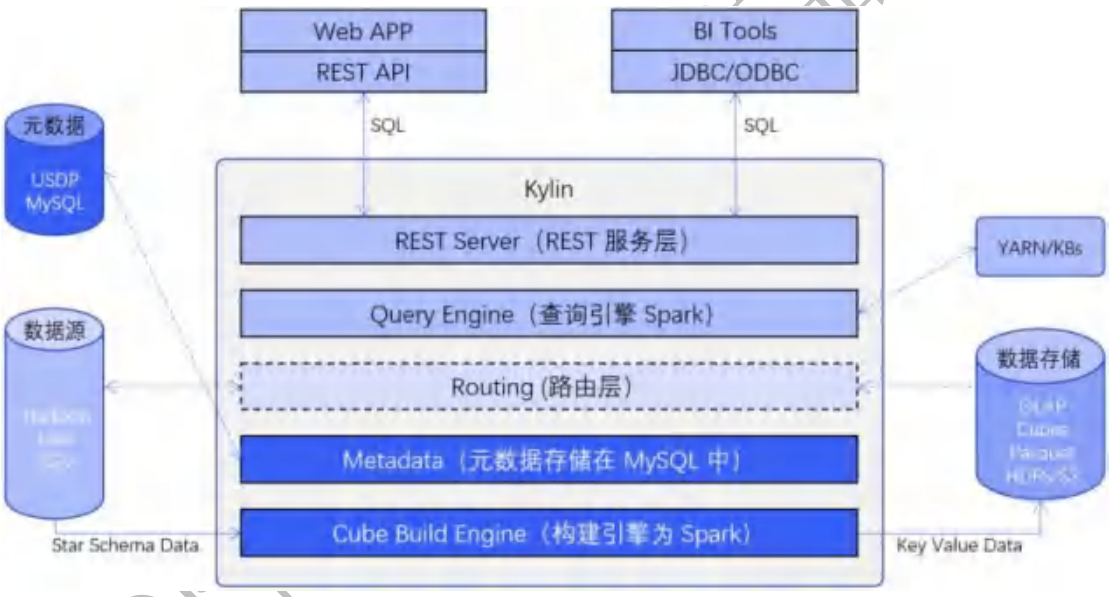
Canal 是用 Java 开发的基于数据库增量日志解析，提供增量数据订阅 &消费的中间件，来解决跨地域数据同步需求，对数据库的日志解析，获取增量变更进行同步。目前 Canal 主要支持了 MySQL 的 Binlog 解析，解析完成后才利用 Canal Client 来处理获得的相关数据。

5.23 Kylin 数仓维度建模服务

Apache Kylin 是一个开源的分布式分析引擎，提供 Hadoop/Spark 之上的 SQL 查询接口及多维分析（OLAP）能力以支持超大规模数据，最初由 eBay Inc 开发并贡献至开源社区。它能在亚秒内查询巨大的 Hive 表。

Kylin 的主要特点包括支持标准 SQL 接口、支持超大规模数据集、亚秒级响应、可伸缩性、高吞吐率、 BI 工具集成等。

Apache Kylin 4 采用了全新的 Spark 构建引擎和 Parquet 作为存储，同时基于 Spark 去做构建和查询，能够充分地利用 Spark 的并行化、向量化和全局动态代码生成等技术，去提高大数据场景下查询的效率。

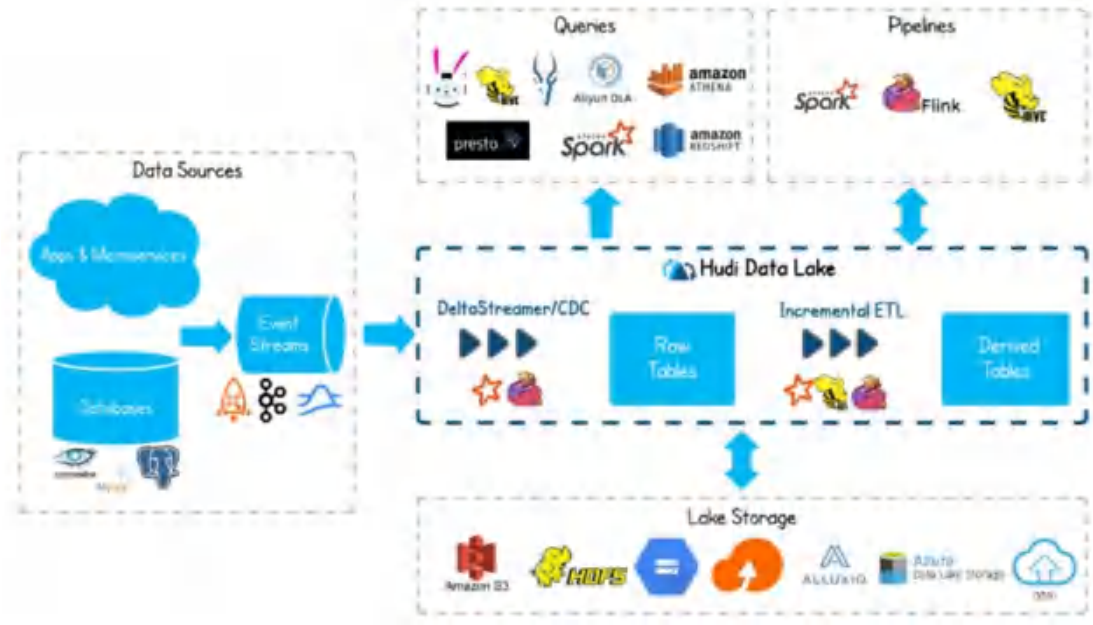


Kylin 是构建大型即时响应数仓的关键服务之一，通过空间换时间的理念，对数据构建若干 Cube，从而实现数据的快速筛查。

5.24 Hudi

Apache Hudi（Hadoop Upserts Delete and Incremental）是下一代流数据湖平台。Hudi 将核心仓库和数据库功能直接引入数据湖，其提供了表、事务、高效的 upserts/delete、高级索引、流摄取服务、数据集群/压缩优化和并发，同时保持数据的开源文件格式（列式文件格式 Parquet 和行式文件格式 Avro 等）。

Hudi 非常适合于流工作负载，而且还允许创建高效的增量批处理管道。Hudi 的高级性能优化，使分析工作负载更快的任何流行的查询引擎，包括 Apache Spark、Flink、Presto、Trino、Hive 等。



Hudi 支持可插拔索引机制支持快速 Upsert/Delete、支持增量拉取表变更以进行处理、支持事务提交和回滚、支持并发控制、支持诸多流批处理和即席查询等引擎的 SQL 读写、自动管理小文件（数据聚簇、压缩、清理）、流式摄入（内置 CDC 源和工具）、内置可扩展存储访问的元数据跟踪、向后兼容的方式实现表结构变更的支持等特性。

5.25 Iceberg

Iceberg 是一个面向海量数据分析场景的开放表格式（Table Format）。表格式（Table Format）可以理解为元数据以及数据文件的一种组织方式，处于计算框架（Flink，Spark...）之下，数据文件之上。

Iceberg 支持数据存储及计算引擎插件化、实时流批一体、数据表演化（Table Evolution）、模式演化（Schema Evolution）、分区演化（Partition Evolution）、列顺序演化（Sort Order Evolution）、隐藏分区（Hidden Partition）、镜像数

据查询（Time Travel）、支持事务（ACID）、基于乐观锁的并发支持、文件级数据剪裁等特性。

5.26 Oozie

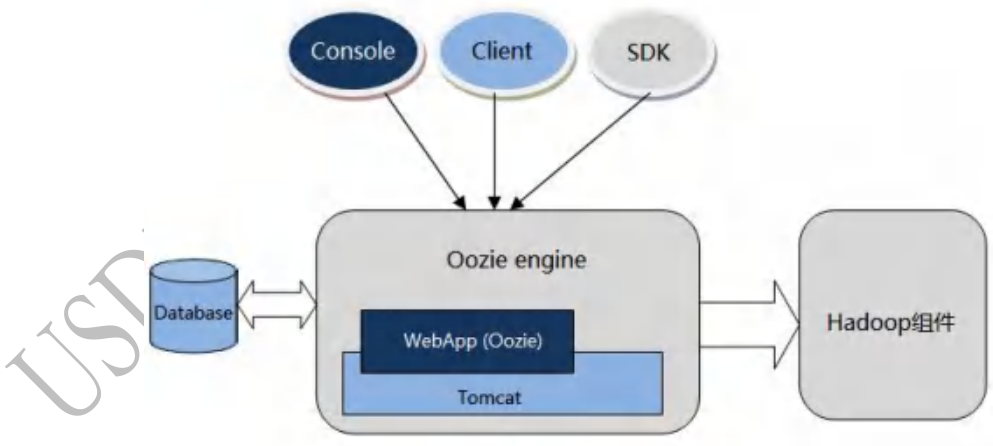
5.26.1 简介

Oozie 是一个基于工作流引擎的开源框架，它能够提供对 Hadoop 作业的任务调度与协调。

5.26.2 结构

Oozie 引擎是一个 Web App 应用，默认集成到 Tomcat 中，采用 pg 数据库。

基于 Ext 提供 WEB Console，该 Console 仅提供对 Oozie 工作流的查看和监控功能。通过 Oozie 对外提 REST 方式的 WS 接口，Oozie client 通过该接口控制（启动、停止等操作）Workflow 流程，从而编排、运行 Hadoop MapReduce 任务，如下图所示。



图中各部分的功能说明如下表所示。

名称	描述
Console	提供对 Oozie 流程的查看和监控功能。

Client	通过接口控制 workflow 流程：可以执行提交流程，启动流程，运行流程，终止流程，恢复流程等操作。
SDK	软件开发工具包 SDK（SoftwareDevelopmentKit）是被软件工程师用于为特定的软件包、软件框架、硬件平台、操作系统等建立应用软件的开发工具的集合。
Database	pg 数据库。
WebApp（Oozie）	webApp（Oozie）即 Oozie server，可以用内置的 Tomcat 容器，也可以用外部的，记录的信息比如日志等放在 pg 数据库中。
Tomcat	Tomcat 服务器是免费的开放源代码的 Web 应用服务器。
Hadoop 组件	底层执行 Oozie 编排流程的各个组件，包括 MapReduce、Hive 等。

5.26.3 原理

Oozie 是一个工作流引擎服务器，用于运行 MapReduce 任务工作流。同时 Oozie 还是一个 Java Web 程序，运行在 Tomcat 容器中。

Oozie 工作流通过 HPDL（一种通过 XML 自定义处理的语言，类似 JBOSS JBPM 的 JPDL）来构造。包含“Control Node”（可控制的工作流节点）、“Action Node”。

- “Control Node”用于控制工作流的编排，如“start”（开始）、“end”（关闭）、“error”（异常场景）、“decision”（选择）、“fork”（并行）、“join”（合并）等。

- Oozie 工作流中拥有多个“Action Node”，如 MapReuce、Java 等。

所有的“Action Node”以有向无环图（DAG Direct Acyclic Graph）的模式部署运行。所以在“Action Node”的运行步骤上是有方向的，当上一个“Action Node”运行完成后才能运行下一个“Action Node”。一旦当前“Action Node”完成，远程服务器将回调 Oozie 的接口，这时 Oozie 又会以同样的方式执行工作流中的下一个“Action Node”，直到工作流中所有“Action Node”都完成（完成包括失败）。

Oozie 工作流提供各种类型的“Action Node”用于支持不同的业务需要，如 MapReduce，HDFS，SSH，Java 以及 Oozie 子流程。

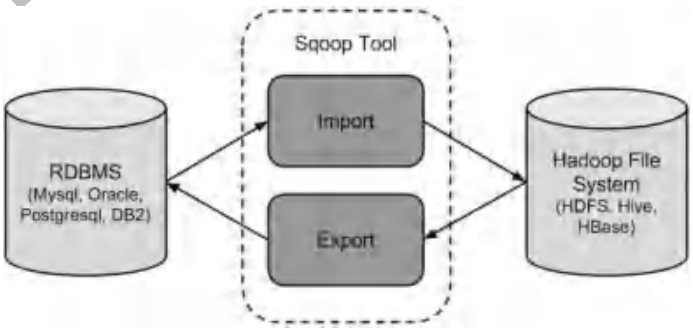
5.27 Sqoop

5.27.1 Sqoop 简介

基于开源 Sqoop 组件进行了功能增强，实现 Hadoop 集群与关系型数据库、文件系统之间交换“数据”、“文件”，同时也可以将数据从关系型数据库或者文件服务器导入到 Hadoop 集群的 HDFS/HBase 中，或者反过来从 HDFS/HBase 导出到关系型数据库或者文件服务器中。

5.27.2 Sqoop 架构

Sqoop 模型主要由 Sqoop Client 和 Sqoop Server 组成，如下图所示：



图：Sqoop 架构

上图中各部分的功能说明如下表所示：

名称	描述
Sqoop Client	Sqoop 的客户端，包括 WebUI 和 CLI 版本两种交互界面。
Sqoop Server	Sqoop 的服务端，主要功能包括：处理客户端操作请求、管理连接器和元数据、提交 MapReduce 作业和监控 MapReduce 作业状态等。
REST API	实现 RESTful（HTTP + JSON）接口，处理来自客户端的操作请求。
Job Scheduler	简单的作业调度模块，支持周期性的执行 Sqoop 作业。
Transform Engine	数据转换处理引擎，支持字段合并、字符串剪切、字符串反序等。
Execution Engine	Sqoop 作业执行引擎，支持以 MapReduce 方式执行 Sqoop 作业。
Submission Engine	Sqoop 作业提交引擎，支持将作业提交给 MapReduce 执行。
Job Manager	管理 Sqoop 作业，包括创建作业、查询作业、更新作业、删除作业、激活作业、去激活作业、启动作业、停止作业。
Metadata Repository	元数据仓库，存储和管理 Sqoop 连接器、转换步骤、作业等数据。
HA Manager	管理 Sqoop Server 进程的主备状态，Sqoop Server 包含 2 个节点，以主备方式部署。

5.27.3 Sqoop 原理

通过 MapReduce 实现并行执行和容错 Sqoop 通过 MapReduce 作业实现并行的导入或者导出作业任务，不同类型的导入导出作业可能只包含 Map 阶段或者同时 Map 和 Reduce 阶段。

Sqoop 同时利用 MapReduce 实现容错，在作业任务执行失败时，可以重新调度。

数据导入到 HBase

1. 在 MapReduce 作业的 Map 阶段中从外部数据源抽取数据。
2. 在 MapReduce 作业的 Reduce 阶段中，按 Region 的个数启动同样个数的 Reduce Task，Reduce Task 从 Map 接收数据，然后按 Region 生成 HFile，存放在 HDFS 临时目录中。
3. 在 MapReduce 作业的提交阶段，将 HFile 从临时目录迁移到 HBase 目录中。

数据导入 HDFS

1. 在 MapReduce 作业的 Map 阶段中从外部数据源抽取数据，并将数据输出到 HDFS 临时目录下（以“输出目录-ldtmp”命名）。
2. 在 MapReduce 作业的提交阶段，将文件从临时目录迁移到输出目录中。
3. 数据导出到关系型数据库
4. 在 MapReduce 作业的 Map 阶段，从 HDFS 或者 HBase 中抽取数据，然后将数据通过 JDBC 接口插入到临时表（Staging Table）中。
5. 在 MapReduce 作业的提交阶段，将数据从临时表迁移到正式表中。

数据导出到文件系统

1. 在 MapReduce 作业的 Map 阶段，从 HDFS 或者 HBase 中抽取数据，然后将数据写入到文件服务器临时目录中。
2. 在 MapReduce 作业的提交阶段，将文件从临时目录迁移到正式目录中。

5.27.4 与组件的关系

与 Sqoop 有交互关系的组件有 HDFS、HBase、Mapreduce 和 Zookeeper。

Sqoop 作为客户端使用这些组件的某些功能，如存储数据到 HDFS 和 HBase，从 HDFS 和 HBase 表读数据，同时 Sqoop 本身也是一个 MR 客户端程序，完成一些数据导入导出任务。

5.28 KrbServer 及 LdapServer

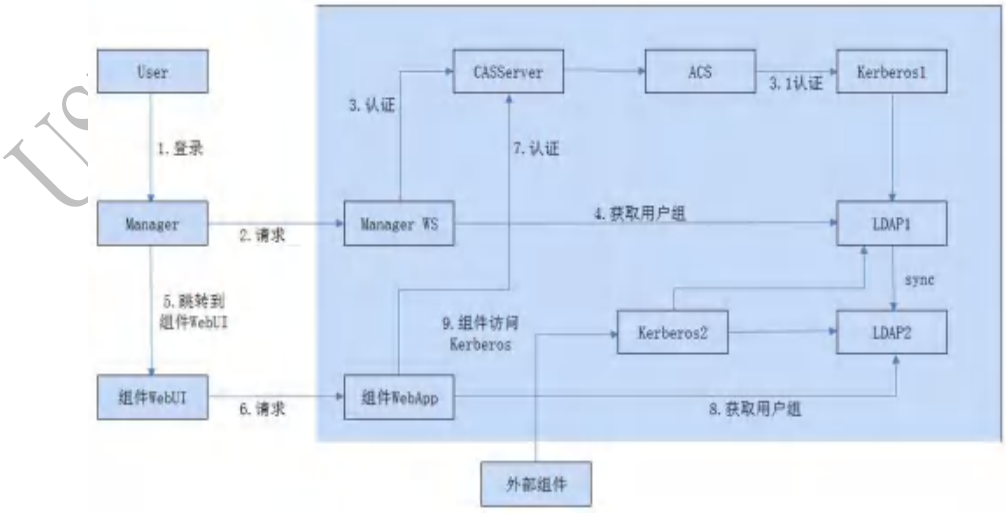
5.28.1 简介

为了管理 Hadoop 集群中数据与资源的访问控制权限，大数据平台推荐以安全模式安装集群。在安全模式下，客户端应用程序在访问 Hadoop 集群中的任意资源之前均需要通过身份认证，建立安全会话链接。Hadoop 集群通过 KrbServer 为所有组件提供 Kerberos 认证功能，实现了可靠的认证机制。

LdapServer 支持轻量目录访问协议（Lightweight Directory Access Protocol，简称为 LDAP），为 Kerberos 认证提供用户和用户组数据保存能力。

5.28.2 结构

Hadoop 集群用户登录时安全认证功能主要依赖于 Kerberos 和 LDAP。



图：安全认证场景架构

上图可分为三类场景：

- 登录 Manager WebUI
  - 认证架构包含步骤 1、2、3、4
- 登录组件 Web UI
  - 认证架构包含步骤 5、6、7、8
- 组件间访问
  - 认证架构为步骤 9

表：关键模块解释

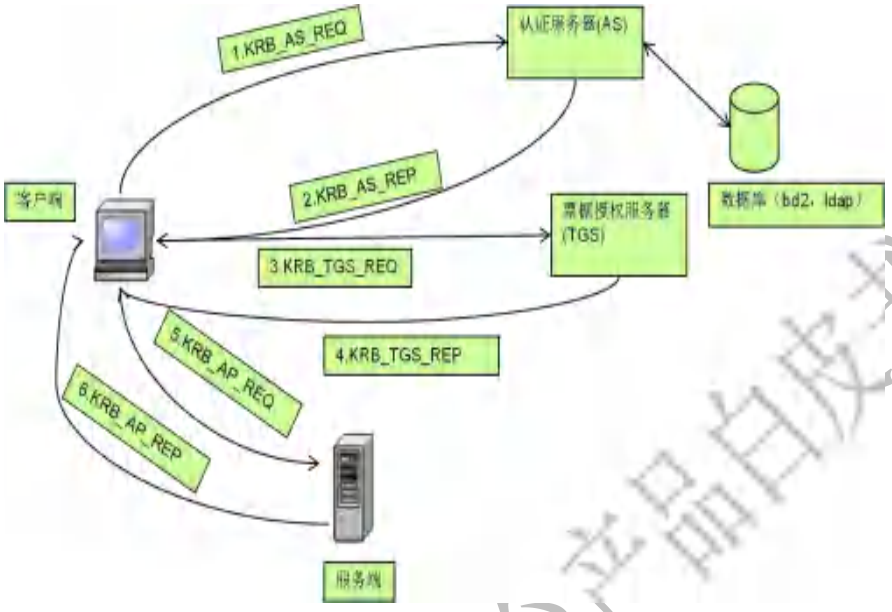
名称	含义
Manager	Hadoop Manager
Manager WS	Hadoop WebBrowser
Kerberos1	部署在 Manager 中的 KrbServer（管理平面）服务，即 OMS Kerberos
Kerberos2	部署在集群中的 KrbServer（业务平面）服务
LDAP1	部署在 Manager 中的 LdapServer（管理平面）服务，即 OMS LDAP
LDAP2	部署在集群中的 LdapServer（业务平面）服务

Kerberos1 访问 LDAP 数据：以负载均衡方式访问主备 LDAP1 两个实例和双备 LDAP2 两个实例。只能在主 LDAP1 主实例上进行数据的写操作，可以在 LDAP1 或者 LDAP2 上进行数据的读操作。

Kerberos2 访问 LDAP 数据：读操作可以访问 LDAP1 和 LDAP2，数据的写操作只能在主 LDAP1 实例进行。

5.28.3 原理

Kerberos 认证



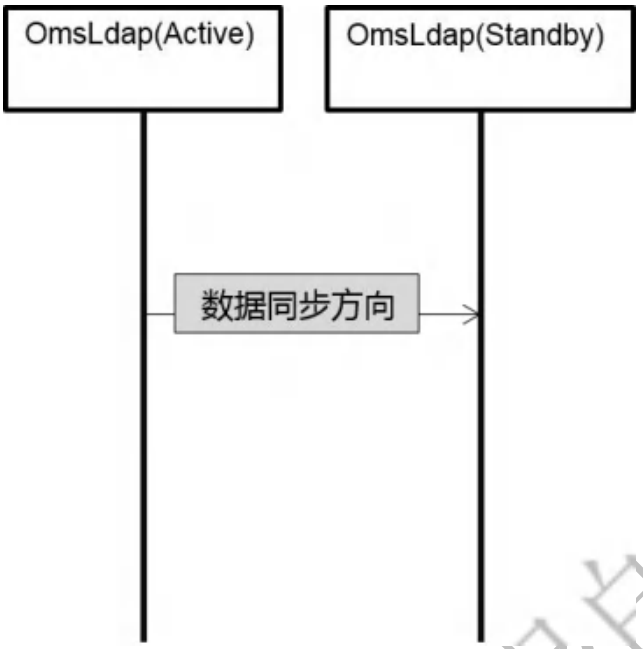
图：认证流程图

LDAP 数据读写

图：数据修改过程

LDAP 数据同步

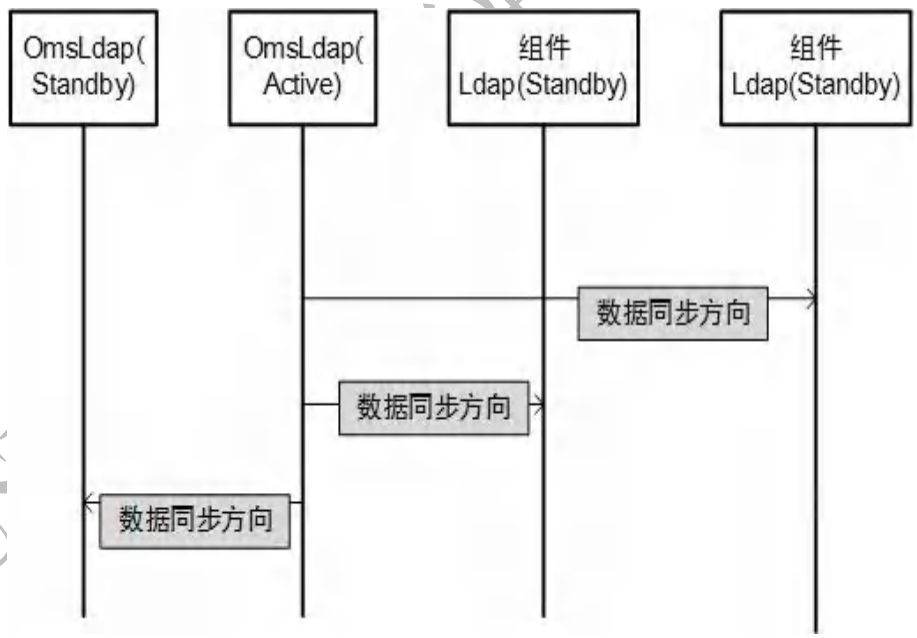
安装集群前 OMS LDAP 数据同步



图：OMS LDAP 数据同步

安装集群前数据同步方向：主 OMS LDAP 同步到备 OMS LDAP。

安装集群后 LDAP 数据同步



图：LDAP 数据同步

安装集群后数据同步方向：主 OMS LDAP 同步到备 OMS LDAP、备组件 LDAP 和备组件 LDAP。

6 运维运营管理

- **统一管理服务**：在智能大数据平台中，统一管理服务指的是集中管理各种运行在平台上的大数据服务。平台提供一个统一的 Web 控制界面，管理员可以通过该界面查看和管理所有的大数据服务，包括对服务的启动、停止、重启、监控等操作。

- **平台管理账号**：为了保证平台的安全性和管理权限，智能大数据平台提供平台管理账号功能。

- **监控告警**：监控告警是智能大数据平台中的核心功能之一。通过监控告警系统，管理员可以实时监测平台中各个大数据服务的运行状态、资源利用率、性能指标等，并设置相应的告警规则。当出现异常或超出预设的阈值时，系统会自动发送告警通知给管理员，以便及时采取措施进行处理。平台提供了丰富的告警规则模板，管理员可便捷的导入并根据需要调整规则，或对规则直接生效运用。

- **通知组**：通知组是一种用于管理和配置告警通知对象范围的功能。管理员可以创建不同的通知组，将相关的用户或团队添加到不同的通知组中。当发生告警事件时，系统可以根据配置的通知组，自动发送告警通知给相应的用户或团队，以便及时响应和解决问题。

- **资源事件**：资源事件功能用于记录和管理平台中发生的各种资源相关事件。这些事件可能包括资源的创建、删除、修改、分配等。通过资源事件功能，管理员可以清楚地了解和跟踪平台中各个资源的操作历史，以便进行资源的管理和优化。

- **服务常用端口**：在智能大数据平台中，各个大数据服务通常需要使用特定的端口进行通信。为了方便管理和配置，平台提供了服务常用端口功能，管理员可以查看和修改各个服务所使用的端口号，确保服务之间的通信正常进行。如为集群某服务的特定端口配置网络策略。若需要调整服务的端口，可直接在相应服务的配置管理页面中进行调整设置。

- **服务部署规则**：大数据服务部署规则指定了在智能大数据平台中部署和

运行服务的规则和约束。这些规则可以包括大数据服务的依赖关系、部署位置、资源限制、数据存储路径等。通过定义和管理服务部署规则，平台可以实现对大数据服务的有效管理和优化资源利用。

- **服务组件启停顺序**：在智能大数据平台中，服务组件启停顺序定义了各个服务组件启动和停止的顺序。由于不同的服务组件之间可能存在依赖关系，需要按照特定的顺序进行启停操作，以确保服务能够正常运行或停止。管理员仅需要参考平台服务启停顺序说明文档，即可方便地管理各个服务组件的启停操作。

- **事件检测与管理功能**：智能大数据平台提供了对平台中所有大数据服务、服务器、存储等对象的各类事件的监测、识别和管理的功能模块。这些事件可以包括系统故障、错误、异常行为、安全漏洞等，通过事件检测与管理功能，管理员可以及时发现并采取相应的措施来应对这些事件，以保证平台的可靠性、稳定性、安全性和性能，为用户提供更好的数据处理和分析服务。

  ■事件监测与采集：平台通过监控和采集系统、服务和应用程序的日志、指标和状态信息，实时获取平台中发生的各种事件数据。

  ■事件分类与识别：通过使用机器学习和人工智能技术，对采集到的事件数据进行分析和处理，识别出不同类型的事件，如性能问题、安全漏洞、异常行为等。

  ■实时告警与通知：一旦发现异常事件或需要管理干预的事件，平台会即时生成告警，并通过通知机制发送给管理员或相关团队，以便及时采取行动。

  ■事件分析与可视化：平台提供事件数据的可视化展示和分析工具，管理员可以通过图表、报表等方式深入了解事件的趋势、频率、严重程度等信息，帮助其更好地进行决策和规划。

  ■事件响应与处理：平台可以提供自动化的事件响应和处理机制，根据预设的规则和策略，对事件进行自动化的处理、修复或调整，以减

少人工干预和降低故障恢复时间。

■事件追踪与审计：平台记录和跟踪事件的历史记录，包括事件的发生时间、触发条件、处理过程等信息，以便进行后续的审计、分析和故障排查。

7 平台安全性

7.1 访问认证

平台安全性是智能大数据平台中至关重要的一方面。访问认证是保障平台安全性的关键措施之一。通过访问认证，平台可以验证用户的身份，确保只有经过授权的用户才能访问和操作平台。

登录密码是最常见的访问认证方式之一。用户需要提供正确的用户名和密码才能登录到平台。为了增强安全性，平台应该要求用户设置强密码，并定期要求用户更改密码。

7.2 统一用户

为了方便管理和维护用户身份，智能大数据平台通常支持统一用户管理。LDAP（轻型目录访问协议）对接是一种常见的统一用户管理解决方案。通过与LDAP 服务器对接，平台可以从 LDAP 中获取用户信息和权限，实现集中管理和统一认证。

LDAP 对接可以实现单点登录（SSO）功能，即用户只需进行一次登录，即可访问平台中的各个服务和应用，无需重复输入用户名和密码。这不仅方便了用户，还提高了平台的安全性，避免了密码被泄露或忘记密码的问题。

7.3 数据中心安全

智能大数据平台通常需要对接数据中心，从中获取数据进行处理和分析。数据中心安全是确保平台运行的重要环节之一。为了保障数据中心安全，平台需要与数据中心的安全设施和策略进行对接。

首先，平台应与数据中心的防火墙和网络安全设备进行对接，以保护数据传输和通信的安全性。这可以通过建立安全的虚拟专用网络（VPN）连接或使用加密协议来实现。

其次，平台需要与数据中心的身份认证和访问控制机制进行对接。这样可以确保只有经过授权的用户和应用程序才能访问数据中心，并限制其权限和访问范围。

另外，平台还应与数据中心的监控系统对接，及时检测和响应异常活动。通过与数据中心安全设施和策略的对接，智能大数据平台能够在保证数据处理效率的同时，提供安全的数据访问和处理环境。

7.4 日志审计

为了确保平台的安全性和合规性，智能大数据平台应该具备日志审计功能。日志审计可以记录和存储平台中发生的操作行为，包括用户登录、服务启停、配置修改等。通过对这些日志进行分析和审计，可以及时发现潜在的安全风险和异常行为。

操作行为审计包括以下几个方面：

登录审计：记录用户登录平台的时间、IP 地址、登录状态等信息，用于追踪和识别异常登录行为。

服务操作审计：记录管理员对服务进行的操作，如启动、停止、重启等，以及对服务配置的修改。这可以用于监控管理员的操作行为，确保其符合规定，并追踪问题的根源。

数据访问审计：记录用户对数据的访问操作，包括读取、写入、修改等。这有助于追溯数据的使用情况，识别异常访问行为和防止数据泄露。

安全事件审计：记录平台中发生的安全事件，如入侵尝试、漏洞扫描、异常访问等。这可以帮助管理员及时发现和应对安全威胁，并采取相应的安全措施。