大数据ETL实现

学习计划：

- 了解 Spark 的基本原理和性质
- 掌握 Spark 的安装装置和 ETL 实现
- 了解 Spark 的交互模式
- 掌握 Hadoop 平台的搭建
- 了解 Hive 的安装方法和基本性质
- 了解 Sqoop 的基本应用

在大数据时代，人们需要处理和使用的数据越来越多。对于企业来说，数据已经成为企业的生存基础，能否利用好自己的数据对企业的发展至关重要。数据库技术为企业分析海量数据提供了有效方案，而在数据仓库的构建过程中，ETL 往往是整个过程中最耗时和复杂的阶段。日益增长的数据处理量对 ETL 技术提出了更高的性能要求，也带来了更大的挑战。

5.1 Spark 的分布式 ETL 实现

为了应对海量数据的 ETL 处理需求，用分布式并行技术实现 ETL 很有必要。尽管当前基于 MapReduce（Hadoop 的一个子项目）范型实现的分布式 ETL 方案能够实现海量数据的高效处理，但是由于 MapReduce 编程模型的限制，即 MapReduce 只有两种处理方式，以及多步的处理过程中存在的高 I/O（输入/输出）开销，使其在 ETL 的转换过程中存在一些性能问题，在处理效率和处理速度方面还有许多优化空间。针对大数据的“海量”特征，以及基于 MapReduce 范型实现的分布式 ETL 方案的局限性，结合数据仓库理论知识和分布式处理技术，基于 Spark 对分布式并行 ETL 技术进行研究，近年来又提出了一种分布式 ETL 的设计方案，重点研究数据转换过程中转换处理的并行实现，并根据不同的转换处理类型给出了适用的解决方法。针对前期非聚集操作，如基本的数据清洗、数据格式标准化操作，提出了基于分区的并行管道处理算法，以分区为单位处理数据，从而提高数据转换效率；对于聚集操作，如事实表的数值数据的聚合操作，采用了分区预聚合方法，以减小数据传输频率。结果显示，提出的方法能够明显加速大数据量的转换处理，进而提高分布式 ETL 的性能和处理效率。本章对基于 Spark 的数据处理流程进行了

性能优化讲解。详细分析了 Spark 在处理中的常见数据倾斜问题，根据不同场景下的数据倾斜情况，分别给出了对应的并行优化策略。最后，通过开发一个实际的决策支持系统，阐述了基于 Spark 的分布式 ETL 的设计与应用情况，包括与传统 ETL 开发方案的比较分析，分析结果证明了基于 Spark 的分布式 ETL 方案的有效性和高可扩展性。

5.1.1 Spark 概述

ApacheSpark 是专为大规模数据处理而设计的快速、通用的计算引擎。Spark 是加州大学伯克利分校的 AMP 实验室开源的类 Hadoop 与 MapReduce 的通用并行框架，Spark 拥有 Hadoop MapReduce 的优点，但不同于 MapReduce 的是，Job 的中间输出结果可以保存在内存中，从而不再需要读写 HDFS，因此，Spark 能更好地适用于数据挖掘与机器学习等需要迭代的 MapReduce 的算法。

Spark 是一种与 Hadoop 相似的开源集群计算环境，但是两者之间还存在一些不同，这些不同使 Spark 在某些工作负载方面表现得更加优越。换句话说，Spark 启用了内存分布数据集，除了能够提供交互式查询外，还可以优化迭代工作负载。

Spark 是使用 Scala 语言实现的，它将 Scala 用作其应用程序框架。与 Hadoop 不同，Spark 和 Scala 能够紧密集成，Scala 可以像操作本地集合对象一样轻松地操作分布式数据集。

尽管创建 Spark 是为了支持分布式数据集上的迭代作业，但是实际上它是对 Hadoop 的补充，可以在 Hadoop 文件系统中并行运行。通过名为 Mesos 的第三方集群框架可以支持此行为。

5.1.2 Spark 数据模型——RDD

弹性分布数据集（Resilient Distributed Dataset，RDD）是 Spark 的最基本抽象，是对分布式内存的抽象使用，实现了以操作本地集合的方式来操作分布式数据集的抽象实现。RDD 是 Spark 最核心的部分，它表示已被分区，不可变的并能够被并行操作的数据集合，不同的数据集格式对应不同的 RDD 实现。RDD 必须是可序列化的。RDD 可以 cache 到内存中，每次对 RDD 数据集操作之后的结果，都可以存放到内存中，下一个操作可以直接从内存中输入，省去了 MapReduce 的大量磁盘 I/O 操作。这对于迭代运算比较常见的机器学习算法和交互式数据挖掘来说，效率提升比较大。

可以将 RDD 理解为一个大的集合，将所有数据都加载到内存中，方便多次重用。第一，它是分布式的，可以分布在多台机器上进行计算；第二，它是弹性的，在计算处理过程中，机器的内存不够时，它会和硬盘进行数据交换，虽然从某种程度上会降低性能，但是可以确保计算得以继续进行。

RDD 是分布式只读且已分区的集合对象。这些集合是弹性的，如果一部分数据集丢失，则可以对它们进行重建，具有自动容错、位置感知调度和可伸缩性，而容错性是最难实现的。大多数分布式数据集的容错性有两种方式：数据检查点和记录数据的更新。对于大规模数据分析系统，数据检查点操作成本很高，主要原因是大规模数据在服务器之间传输带来的各方面问题，相比记录数据的更新，RDD 也只支持粗粒度的转换，也就是记录如

何从其他 RDD 转换而来(即 Lineage),以便恢复丢失的分区。

RDD 的特性为:数据存储结构不可变;支持跨集群的分布式数据操作;可对数据记录按 key 分区;提供了粗粒度的转换操作;数据存储在内存中,保证了低延迟性。

5.1.3 Spark 的安装配置

Spark 安装可以分为三步:安装 JDK 和 Spark、配置 Spark 和启动 Spark。

1. 安装 JDK 和 Spark

在安装 JDK 和 Spark 时,需要部署虚拟机、下载 Spark 和 JDK 安装包,详见如下步骤。

(1) 机器部署:准备一台以 Linux 为操作系统的服务器,安装好 JDK 1.7。

(2) 如图 5-1 所示,下载 Spark 安装包。

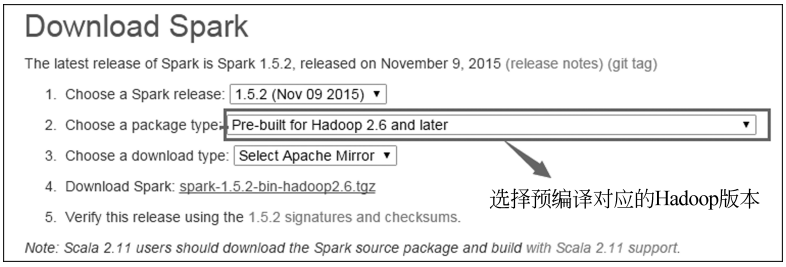

图 5-1 下载 Spark 安装包

(3) 上传解压安装包。上传 spark-1.5.2-bin-hadoop 2.6.tgz 安装包到 Linux 上。

(4) 解压安装包到指定位置。打开 Linux 系统命令行,输入“tar -zxvf spark-1.5.2-bin-hadoop2.6.tgz -C /home/hadoop/app”。

2. 配置 Spark

配置 Spark 需要修改“spark-env.sh.template”“spark-env.sh”和“slaves.template”文件,详细步骤如下。

(1) 进入 Spark 安装目录。

```
cd /home/hadoop/app/spark-1.5.2-bin-hadoop2.6
```

进入 conf 目录重命名并修改“spark-env.sh.template”文件。

```
cd conf/
mv spark-env.sh.template spark-env.sh
```

(2) 在“vi spark-env.sh”配置文件中添加如下配置。

```
export JAVA_HOME=/home/hadoop/app/jdk1.7.0_65
export SPARK_MASTER_IP=weekend110
export SPARK_MASTER_PORT=7077
```

(3) 保存并退出。

(4) 重命名并修改“slaves.template”文件。

```
mv slaves.template slaves
```

(5) 在“vi slaves”文件中添加子节点所在的位置(Worker 节点)weekend110。

(6) 保存并退出。

(7) Spark 集群配置完毕,目前是 1 个 Master 和 1 个 Worker,在 weekend110 上启动 Spark 集群。

```
/home/hadoop/app/spark-1.5.2-bin-hadoop2.6/sbin/start-all.sh
```

(8) 启动后执行“jps”命令,主节点上有 Master 进程,其他子节点上有 Worker 进程,如图 5-2 所示,登录 Spark 管理界面查看集群状态(主节点)。

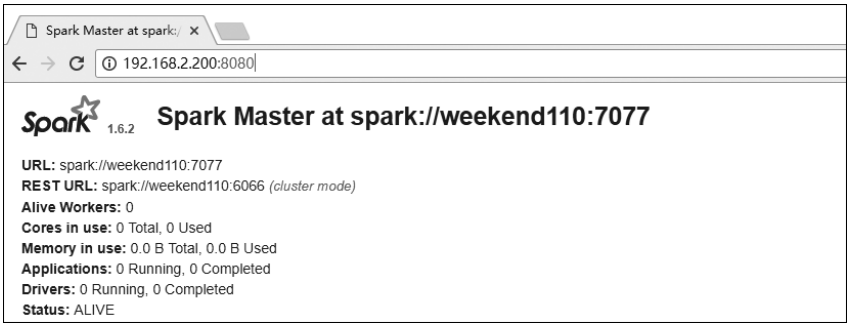

图 5-2 登录 Spark

至此,单节点 Spark 集群安装完毕,但是 Master 节点存在单点故障,要解决此问题,需借助 ZooKeeper,并且至少启动两个 Master 节点来实现高度可行,配置方式比较简单,如下所示。

Spark 集群规划:node1、node2 是 Master;node3、node4、node5 是 Worker。

(1) 在 node1 节点上修改 slaves 配置文件内容,指定 Worker 节点。

(2) 在 node1 上执行 sbin/start-all.sh 脚本,然后在 node2 上执行 sbin/start-master.sh,启动第二个 Master。

3. 执行第一个 Spark 程序

```
spark-submit --class org.apache.spark.examples.SparkPi --master spark://
weekend110:7077 --executor-memory 500m --total-executor-cores 2/home/
hadoop/app/park-1.6.2-bin-hadoop2.6/lib/spark-examples-1.6.2-hadoop2.6.0.
jar 10
```

该算法是利用蒙特卡罗算法求 PI,参数说明如下。

(1) “--master spark://weekend200:7077”指定了 Master 的地址。

(2) "--executor-memory 1G"指定了每个 Worker 可用内存为 1GB。

(3) "--total-executor-cores 2"指定了整个集群使用的 CUP 核数为 2。

spark-shell 是 Spark 自带的交互式 Shell 程序,方便用户进行交互式编程,用户可以在该命令行下用 Scala 编写 Spark 程序。

5.1.4 分布式 ETL 总体架构

第 2 章介绍了基于 Hadoop 平台的 ETL 实现过程。在基于 MapReduce 模型实现的 ETL 中,在转换阶段的多步数据处理过程中存在一些性能限制。而在 Spark 的多步数据处理过程中,可以将中间数据缓存到内存中,从而多次重用已缓存数据,直到全部数据处理完后保存到目标结果中。因而使用 Spark 进行 ETL,能极大减少使用 Hadoop 平台进行 ETL 过程中的磁盘 I/O 次数,提高整体过程的处理效率和处理性能。图 5-3 所示是基于 Spark 的 ETL 过程中的数据流向图。

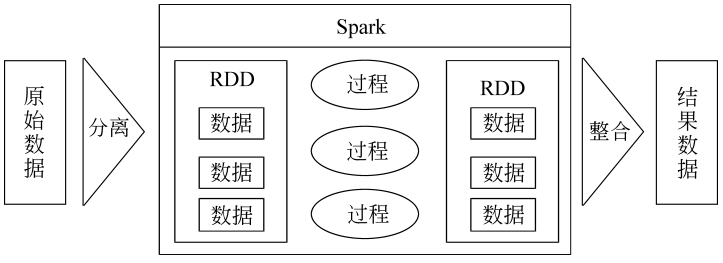

图 5-3 基于 Spark 的 ETL 数据流向图

从图 5-3 中可以看出,原始数据经过分离(spilt)成若干部分,交予 Spark 进行处理,最终将处理后的数据整合(merge)成结果数据。相比于 Hadoop 平台,基于 Spark 的 ETL 过程将主要的数据转换处理"封装"在 Spark 中,基于 Spark 的内存处理单元 RDD 完成复杂和耗时的转换处理,利用 RDD 的可缓存性,尽量减少转换过程中的 I/O 开销,从而提高 ETL 的整体性能。

基于 Spark 的分布式 ETL 架构如图 5-4 所示。首先从数据源抽取出需要的细粒度原始数据,暂存到 HDFS 中,抽取的数据源可以是关系数据库、文本文件等,抽取实现则可以根据数据源格式选择可用的 Sqoop 工具或者使用 HDFS 的上传功能;之后进入正式的转换阶段,此阶段基于 Spark 集群,通过从 HDFS 中读取各数据块创建 RDD 来完成一系列的转换处理,其中转换处理分为前期的格式化处理和之后的整合处理,包括数据清洗、数据过滤、数据转换、数据聚合等转换操作;最后将转换完成的结果数据加载到目标数据仓库中,加载实现可以直接通过 JDBC 中间件将数据写到目标(DW)中,或者先将数据保存为指定格式(CSV、Parquet)的文件,再进行后续加载。

本章主要介绍基于 Spark 的并行 ETL 的过程,重点关注数据转换过程中的并行实现。从图 5-4 所示的架构中也可以看出,数据转换过程是 ETL 中相对复杂和耗时的阶段,它需要对抽取到的源数据进行一系列连续的转换操作,以得到符合要求的目标数据。

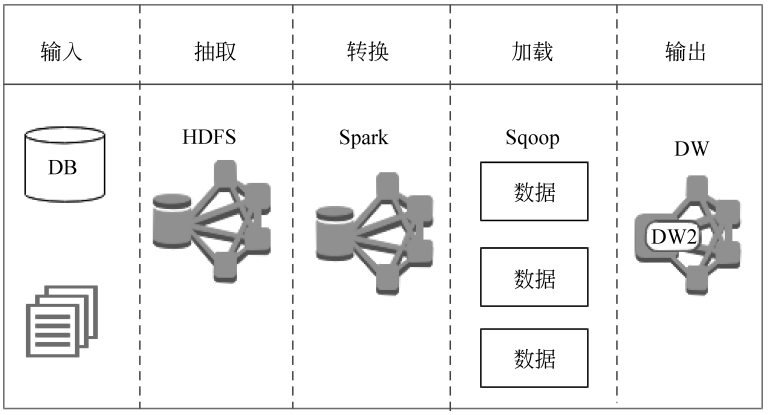

图 5-4　基于 Spark 的分布式 ETL 架构

5.1.5　分布式转换引擎的实现

用户通过分布式转换引擎对数据源进行操作，它是基于 Spark 框架的分布式 ETL 工具的中枢，其功能是从 ETL 脚本中读取任务流程相关节点信息，再对脚本解析后，根据相关信息调用不同的流程节点执行。

分布式转换引擎包括流程解析模块、流程执行模块、Spark 流程节点和用户扩展模块。流程解析模块读取命令行传递的 ETL 脚本，并对 ETL 脚本进行解析，根据任务执行序列和流程节点依赖关系调用相应的 Spark 流程节点或者用户扩展模块，动态编译加载用户自定义的函数。用户扩展模块则读取用户自定义的 Java 源代码，将 Java 源代码进行动态编译，或者对 Java Class 文件进行加载运行。

采用解析脚本的方式实现分布式转换引擎，主要是从扩展性和灵活性两方面考虑，增加分布式 ETL 工具的适应面。同时还在分布式转换引擎中添加用户扩展模块，通过使用向用户提供的 Java 源代码进行动态编译加载的方式，给用户自定义转换节点功能的空间。

分布式转换引擎从 ETL 脚本中读取任务流程节点类型信息，根据流程节点类型的不同，分为普通流程节点、Java 源代码、外部类三种情况。当类型为普通流程节点时，直接调用分布式 ETL 工具提供的相应 Spark 流程节点。当类型为 Java 源代码时，会根据源代码创建 JavaFileManagerImpl 对象，然后通过 JavaSourceCompiler 类中的 JavaCompileTask 对象调用 JavaCompilerClassLoader 对象，最终执行 JavaCompiler 的 getTask 函数进行动态编译。当类型为 class 时，会根据 ETL 脚本参数中的类名执行 JavaCompilerClassLoader 类中的 class.forname()函数进行加载，但是必须将外部类文件放置到指定目录，或者打成 Jar 包放置到 classpath 中。具体执行流程如图 5-5 所示。

1. 流程解析模块

流程解析模块的功能为解析 ETL 脚本的内容，读取命令行传递的 ETL 脚本，并对 ETL 脚本进行解析，生成任务执行序列和流程节点依赖关系。本书规定 ETL 脚本的格式如表 5-1 所示。

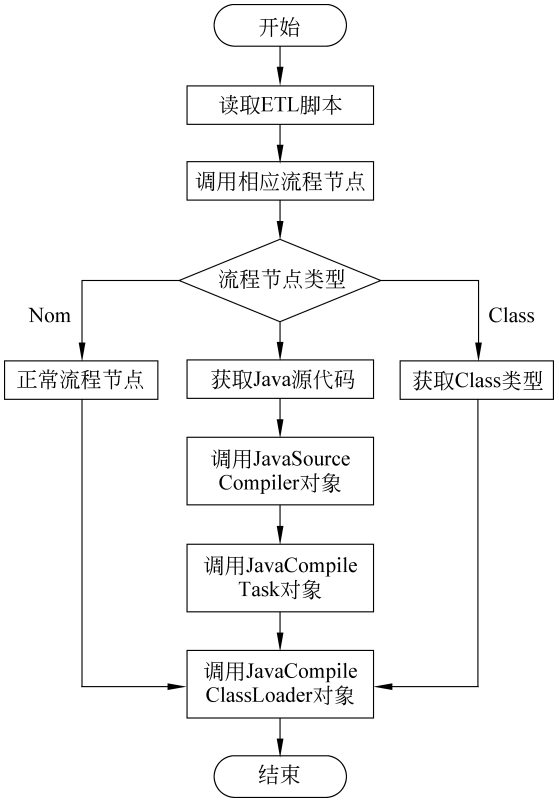

图 5-5 分布式转换引擎工作流程

表 5-1 ETL 脚本格式

内 容	解 析
流程节点序号	当前流程节点的序号(开始节点序号为 0,结束节点序号为 1)
前驱节点序号	为其前驱节点的序号,若前驱节点为多个,则以逗号分隔
流程节点名称	当前流程节点的名称
流程节点类型	有 5 种类型:start、extract、transform、load 和 end
流程节点参数	流程节点的参数设置

流程解析模块就是要对这种格式的 ETL 进行解析,使基于 Spark 的分布式 ETL 工具能灵活地执行多种用途的 ETL 任务。在 5 种类型的流程节点中,extract、transform、load 分别代表 ETL 流程中的抽取、转换、加载 3 种类型节点。值得注意的是,在 ETL 流程中增加了 start 和 end 两种类型的节点,在整个 ETL 流程中,start 节点主要负责初始化 Spark 集群,根据脚本提供的参数创建 SparkContext。end 节点表示 ETL 脚本结束,只有添加 end 节点后,分布式转换引擎才会将 Spark 任务提交到集群中执行。另外,在 ETL 流程中有且仅有一个 start 节点和一个 end 节点。

2. Spark 流程节点

常规的 ETL 工具将流程节点分为抽取、转换、加载 3 种类型，但是 Spark 程序在执行真正工作代码前，必须设置参数对集群进行初始化创建 SparkContext。另外需要注意的是，每个 Spark 程序有且只有一个 SparkContext，不同 SparkContext 之间无法共享 RDD。虽然可以将集群参数初始化放到抽取节点中进行，但是由于 ETL 流程可能存在多个抽取节点，所以在基于 Spark 框架的分布式 ETL 工具增加了开始节点，负责对 Spark 集群进行初始化创建 SparkContext。同时在 ETL 流程最后增加结束节点，负责将 Spark 程序提交到 Spark 集群中运行。

对于常规的抽取、转换、加载节点，可以使用不同的 Spark 处理模型实现。例如，对于常规批处理任务，可以使用 Spark 实现；对于实时性业务，可以使用 SparkStreaming 实现；对于海量数据查询业务，可以用 SparkSQL 实现等。需要注意的是，在 Spark 框架中，各个模块的 Context 类型并不相同，如普通 Spark 程序为 SparkContext，SparkStreaming 程序为 StreamingContext 类型，Spark SQL 程序为 SQLContext 类型等，并不能随意转换。不过由于 SQLContext 等类型都是基本 SparkContext 类型的子类，可以通过 SparkContext 转换。所以，在实现中，常规流程节点会根据需要对开始节点中生成的 SparkContext 进行转换，并在流程节点结束时再转换为 SparkContext 传给下一个流程节点。

(1) 基于 Spark 的抽取节点。

在 Spark 框架中，RDD 容错机制是基于 Lineage 的。Lineage 是由不同 RDD 的转换关系构成的计算链，可以把这个计算链认为是 RDD 之间演化的“血统”。在由于部分 Spark 集群节点故障导致 RDD 计算结果丢失时，只需要根据 Lineage 重新计算该 RDD 即可，整个 Spark 程序并不会失败。

这就会带来一个问题，如果数据源数量巨大，则全量抽取过程可能会持续时间较长，若中间有集群节点出现故障导致 RDD 结果丢失，就需要重新执行整个抽取操作。所以，为了减少 Spark 集群中间 RDD 计算结果丢失后重新抽取的开销，将每个 RDD 的抽取数量限制到一定大小，比如每个 RDD 大小设置为 20 000 条数据，这样当某一个 RDD 出错后，重新计算的开销仅仅为抽取这个 20 000 条数据的开销，而不是重新抽取源数据库中的所有数据，会大大减少重新抽取的开销，当数据量较大时将会带来性能的提升。

所以，基于 Spark 的全量抽取节点在对源数据库进行抽取时，首先获取表中的数据总条数、主键值的最大值和最小值，之后根据单个 RDD 限制数量的大小，计算出将要划分的 RDD 数，创建多个连接并行抽取源数据库中的数据。

(2) 基于 Spark 的转换节点。

在抽取数据后，在将数据加载到目的数据库之前，经常需要处理某些数据。例如，常见的数据迁移情况是数据源和数据目的表结构不一致，列名不相同，这时就需要转换原始数据的列名；或者在迁移时只需要处理部分数据，这时可以直接将不需要处理的数据传递为加载节点，将需要处理的数据进行转换处理后再加载，这个过程如图 5-6 所示。

经过抽取节点处理后，数据流分为两部分：发往转换节点的转换数据流和直接加载

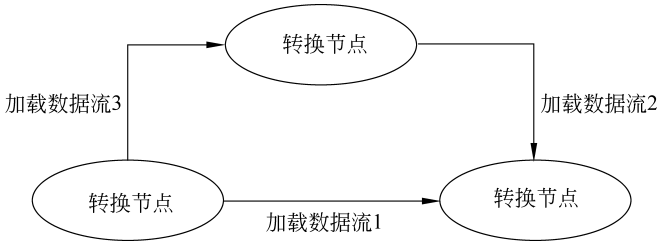

图 5-6 有转换节点的数据迁移模型

的加载数据流 1，转换数据流在转换节点处理后，生成加载数据流 2 发往加载节点。

因为在对数据进行转换处理时，抽取节点的元数据结构以及处理需求都不相同，很多都依赖于具体问题，为了向用户提供灵活处理抽取数据的能力，从两方面向用户提供提交针对具体问题的 Java 源代码的功能。

① 通过向 Spark 操作算子提供函数。

② 通过用户扩展模块动态编译加载执行码。

两种方式在形式上最主要的区别在 ETL 脚本中参数的类型上，第①种方式不需要特殊说明类型，直接放置在流程节点的参数列表中，由流程节点直接运行；第②种方式则需要标明为用户自定义功能 Java 源代码，由用户扩展模块对 Java 源代码进行动态编译加载执行。

(3) 基于 Spark 的加载节点。

加载节点是 ETL 流程的结束，通过 Spark 的 action 算子实现，业务逻辑相对比较简单，只需要将转换节点处理后的数据持久化到目的数据库即可。简单的做法是直接通过 foreach 算子执行 SQL 语句，将数据写入到目的数据库中，但这会使每个记录都向数据库服务器申请创建一个连接对象，创建和销毁每个数据的连接对象将会造成不必要的时间和资源开销。因为 Spark 框架的 RDD 是一个分散式内存数据集，RDD 中的数据存放在多个集群节点中，所以不能直接以 RDD 为单位向数据库服务器申请创建连接对象执行 SQL 语句，这会导致某些集群节点由于未创建连接而不能执行 SQL 操作。

因为 RDD 在集群的不同节点都存在一个分区，所以需要使 ForeachPartition 算子为每个分区创建一个单独的连接对象，使用该连接执行所有在该 RDD 分区的 SQL 语句，充分利用 Spark 集群多节点的并发性。

5.1.6 SparkStreaming 的实时同步实现

SparkStreaming 实现同步，主要利用增量抽取机制进行，下面介绍增量抽取机制和 SparkStreaming 的增量抽取机制。

1. 增量抽取机制

数据抽取节点是 ETL 流程的开始节点，直接影响数据转换节点和数据加载节点时的处理速度，如果数据抽取节点效率低，那么即便数据转换节点和数据加载节点使用分布式处理框架，也不能提高整个 ETL 流程的处理效率。可以从多主机并行抽取和增量抽取两

方面提高数据抽取效率，增量抽取机制要比全量抽取更加复杂，要求在一定时间段内精确迅速地抽取改变的数据，同时不能增加太大的负荷，影响源数据库服务器上业务的正常运行。

2. SparkStreaming 的增量抽取机制

SparkStreaming 是 Spark 核心 API 的一种扩展，通过它可以实现对实时流数据的高吞吐量、低容错率处理。SparkStreaming 可以很好地支持流数据格式，如很好地支持 Kafka、Flume、Kinesis、Twitter、Zero MQ、MQTT 等工具生成的流数据，这是当前多数公司收集日志或者数据常用的方法，具有广泛的适应性。

SparkStreaming 的内部实现原理是接收实时输入数据流并将数据流划分为微批次，然后由底层的 Spark 框架引擎分批处理，生成最终的结果流。SparkStreaming 将原始流数据抽象为 DStream 的离散数据流。DStream 是 SparkStreaming 提供的基本抽象，代表了一个连续的数据流，这个数据流可以是从数据源接收的，也可以是对输入流进行处理后的数据流。因为在内部，它由多个 RDD 连续序列表示，所以也是不可改变的数据集抽象，DStream 中的每个 RDD 都包含数据流上某个时间间隔的数据集。基于 SparkStreaming 的程序运行周期如下。

(1) 设置 Spark Conf，并生成 JavaStreamingContext 对象。

(2) 通过创建输入 DStream 定义输入数据源。

(3) 通过对 DStream 应用转换算子和输出操作定义数据流的计算流程。

(4) 使用 StreamingContext.start()函数开始接收流数据并根据第(2)步的计算流程进行运算。

(5) 使用 streamingContext.awaitTermination()函数等待数据流运算结束。

在 Spark 程序中有两种方法可以生成 SparkStreaming 运行时需要的 JavaStreamingContext：直接使用 SparkConf 对象生成，以及从现有 Spark 通用的 JavaSparkContext 对象转换。第一种方法直接生成 JavaStreamingContext，但是之后就不能使用 Spark SQL 等其他库的函数(各自都有自己格式的 Context)；而第二种方法在 Java StreamingContext 关闭后，可以继续使用 JavaSparkContext 转换为其他格式的 Context，因此选择第二种方式生成 StreamingContext。

5.2 Spark 完成在 ETL 时的相关技术

本节主要讲述 Spark 的架构，分析 Spark 执行 Application 的实现逻辑，为后续扩展为 Web Service 提供理论支持。通过分析 Spark 的 Master 模块，可以深入理解 Spark 集群的交互调用模式，从而设计出合适的 SparkWebService。

为了后续描述更为准确，便于理解，介绍后续用到的部分 Spark 组件。

- Application：是指用户提交的 Spark 程序，运行时由 Driver 与 Executor 组成。
- Driver：用户运行 Application 时执行 main()函数，创建 SparkContext 的过程。
- Executor：在 Worker 节点上由 Application 启动的，执行分配的 Tasks 的进程。