IoTDB v0.1.2

用户手册

V0.1.5

2017.9.26

目录

第1章 IoTDB 概述

IoTDB 是针对时间序列数据收集、存储与分析一体化的数据管理引擎。它具有体量轻、性能高、易使用的特点，完美对接 Hadoop 与 Spark 生态，适用于工业物联网应用中海量时间序列数据高速写入和复杂分析查询的需求。

1.1 IoTDB 应用场景

IoTDB 套件由若干个组件构成，共同形成“数据收集-数据写入-数据存储-数据查询-数据可视化-数据分析”等一系列功能。图 1.1 展示了使用 IoTDB 套件全部组件后形成的整体应用架构。下文称所有组件形成 IoTDB 套件，而 IoTDB 特指其中的时间序列数据库组件。

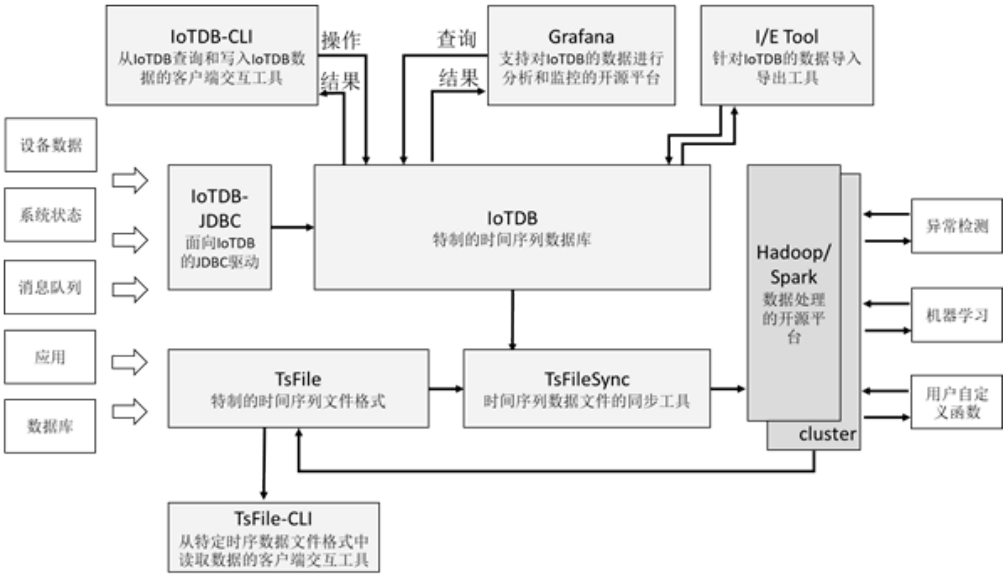

图 1.1 IoTDB 整体应用架构

在图 1.2 中，用户可以通过 JDBC 将来自设备上传感器采集的时序数据、服务器负载和 CPU 内存等系统状态数据、消息队列中的时序数据、应用程序的时序数据或者其他数据库中的时序数据导入到本地或者远程的 IoTDB 中。用户还可以将上述数据直接写成本地（或位于 HDFS 上）的 TsFile 文件。

对于写入到 IoTDB 的数据以及本地的 TsFile 文件，可以通过同步工具 TsFileSync 将数据文件同步到 HDFS 上，进而实现在 Hadoop 或 Spark 的数据处理平台上的诸如异常检测、机器学习等数据处理任务。对于分析的结果，可以写回成 TsFile 文件。

IoTDB 和 TsFile 还提供了相应的客户端工具，满足用户查看和写入数据的

SQL 形式、脚本形式和图形化形式等多种需求。

1.1.1 场景 1

某公司采用表面贴装技术（SMT）生产芯片：需要首先在芯片上的焊接点处印刷（即涂抹）锡膏，然后将元器件放置在锡膏上，进而通过加热熔化锡膏并冷却，使得元器件被焊接在芯片上。上述流程采用自动化生产线。为了确保产品质量合格，在印刷锡膏后，需要通过光学设备对锡膏印刷的质量进行评估：采用三维锡膏印刷检测（SPI）设备对每个焊接点上的锡膏的体积（v）、高度（h）、面积（a）、水平偏移（px）、竖直偏移（py）进行度量。

为了提升印刷质量，该公司有必要将各个芯片上焊接点的度量值进行存储，以便后续基于这些数据进行分析。

此时可以采用 IoTDB 套件中的 TsFile 组件、TsFileSync 工具和 Hadoop/Spark 集成组件对数据进行存储：每新印刷一个芯片，就在 SPI 设备上使用 SDK 写一条数据，这些数据最终形成一个 TsFile 文件。通过 TsFileSync 工具，生成的 TsFile 文件将按一定规则（如每天）被同步到 Hadoop 数据中心，并由数据分析人员对其进行分析。

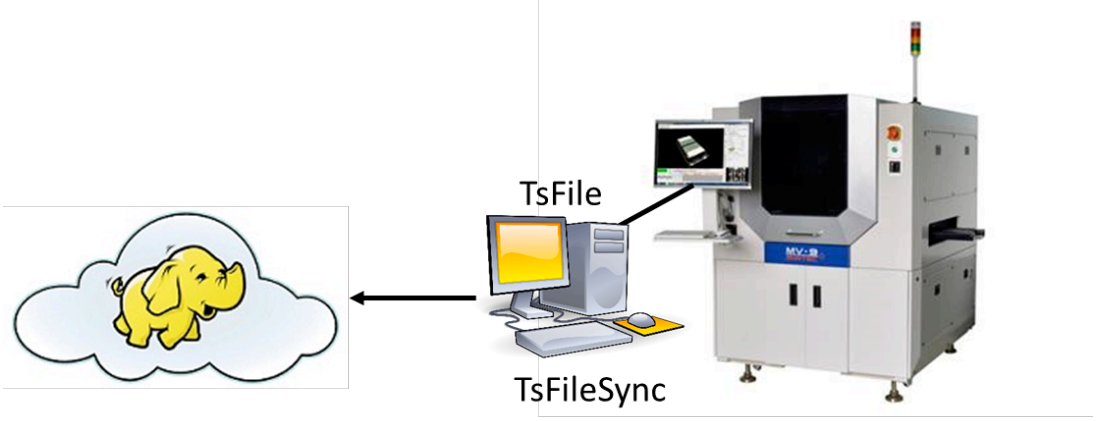

图 1.2 场景 1 示例

在场景 1 中，仅需要 TsFile、TsFileSync 部署在一台 PC 上，此外还需要 Hadoop/Spark 集群。其示意图如图 1.2 所示。图 1.3 展示了此时的应用架构。

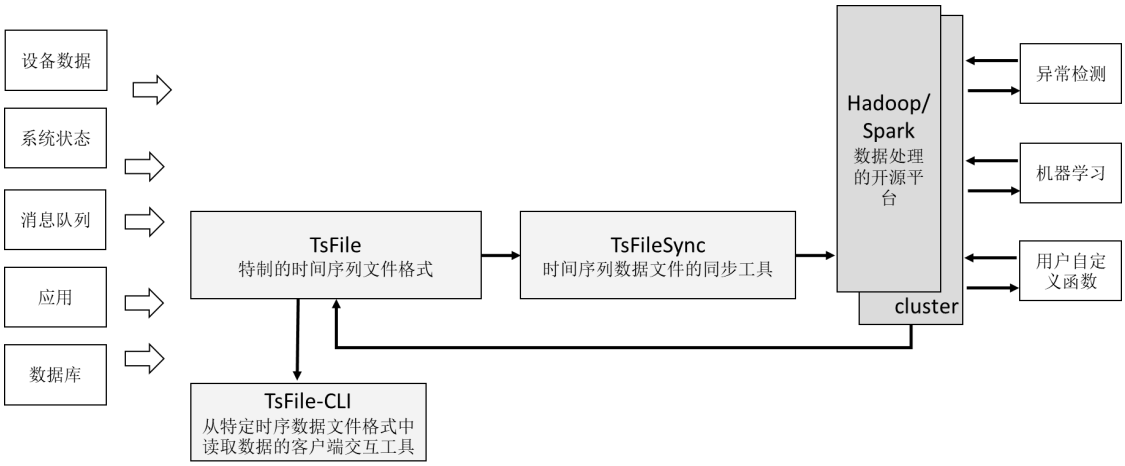

图 1.3 场景 1 的应用架构

1.1.2 场景 2

某公司拥有多座风力发电机，公司在每个发电机上安装了上百种传感器，分别采集该发电机的工作状态、工作环境中的风速等信息。

为了保证发电机的正常运转并对发电机及时监控和分析，公司需要收集这些传感器信息，在发电机工作环境中进行部分计算和分析，还需要将收集的原始信息上传到数据中心。

此时可以采用 IoTDB 套件中的 IoTDB、TsFileSync 工具和 Hadoop/Spark 集成组件等。需要部署一个场控 PC 机，其上安装 IoTDB 和 TsFileSync 工具，用于支持读写数据、本地计算和分析以及上传数据到数据中心。此外还需要部署 Hadoop/Spark 集群用于数据中心端的数据存储和分析。如图 1.4 所示。

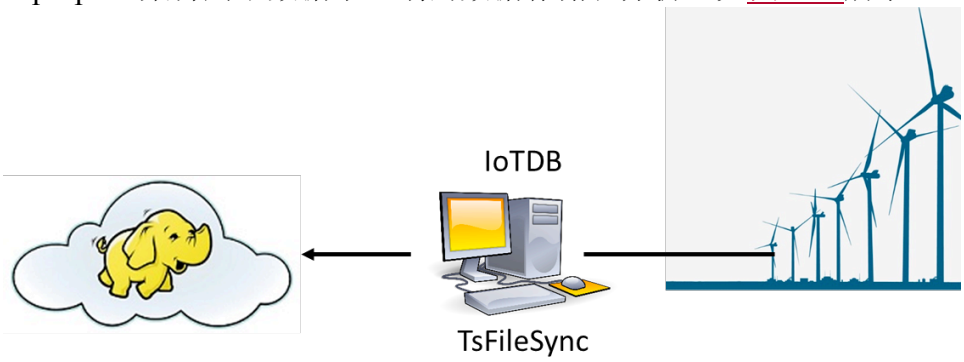

图 1.4 场景 2 的使用示意

图 1.5 给出了此时的应用架构。

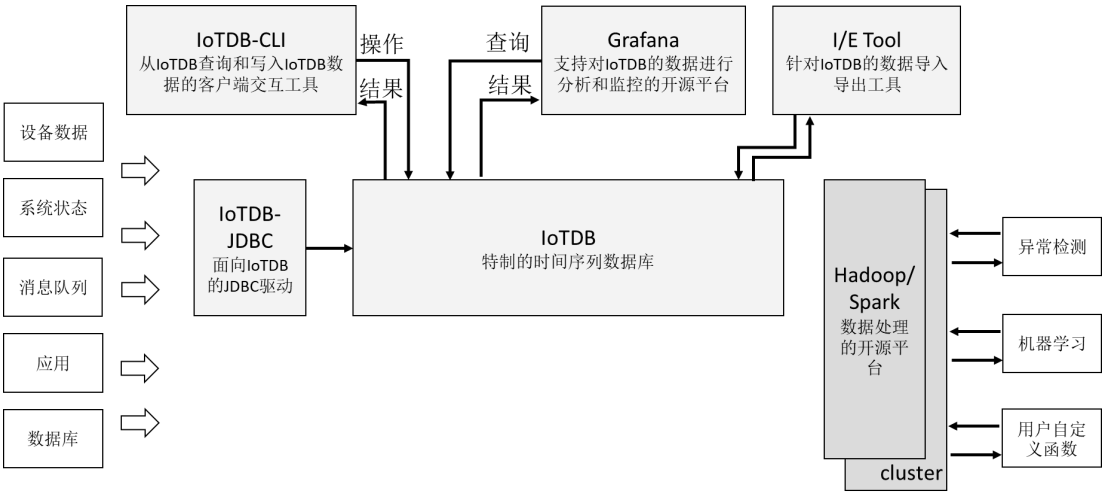

图 1.5 场景 2 的应用架构

1.2 主要功能与特点

IoTDB 具有以下特点：

- 灵活的部署方式
  - 云端一键部署
  - 终端解压即用
  - 终端-云端无缝连接（数据云端同步工具）
- 低硬件成本的存储解决方案
  - 高压缩比的磁盘存储（10 亿数据点硬盘成本低于 1.4 元）
- 目录结构的时间序列组织管理方式
  - 支持复杂结构的智能网联设备的时间序列组织
  - 支持大量同类物联网设备的时间序列组织
  - 可用模糊方式对海量复杂的时间序列目录结构进行检索
- 高通量的时间序列数据读写
  - 支持百万级低功耗强连接设备数据接入（海量）
  - 支持智能网联设备数据高速读写（高速）
  - 以及同时具备上述特点的混合负载
- 面向时间序列的丰富查询语义
  - 跨设备、跨传感器的时间序列时间对齐
  - 面向时序数据特征的计算（频域变换，当前版本不支持）
  - 提供面向时间维度的丰富聚合函数支持
- 极低的学习门槛

    - 支持类 SQL 的数据操作
    - 提供 JDBC 的编程接口
    - 完善的导入导出工具（当前版本不支持）
- 完美对接开源生态环境（当前版本不支持）
    - 支持开源数据分析生态系统：Hadoop、Spark
    - 支持开源可视化工具对接：Grafana

1.3 手册介绍目标及读者人群

手册仅涉及 IoTDB 和相关的组件，不介绍 TsFile 和 TsFile-CLI 和基于 TsFile 文件的 Hadoop、Spark 生态对接方法。

手册针对的读者是使用 IoTDB 的数据库管理员（DBA）和普通开发者。

第2章 IoTDB 基本概念

IoTDB 中涉及如下基本概念：

- 设备（device）
- 传感器（sensor）
- 路径（path）
- 时间序列（timeseries path）
- 前缀路径（prefix path）
- 带*路径（path with star）
- 数据的列（column）
- 存储组（storage group）
- 数据点(point)
- 值（value）
- 时间戳（timestamp）

关于以上的基本概念的详细解释，请在本章第 2.1-2.10 小节查看。

2.1 设备

2.2 传感器

传感器是指在实际场景中的一种检测装置，它能感受到被测量的信息，并能将感受到的信息按一定规律变换成为电信号或其他所需形式的信息输出并发送给 IoTDB。在 IoTDB 当中，存储的所有的数据及路径，都是以传感器为单位进行组织。

2.3 路径

在 IoTDB 中，路径是指符合以下约束的表达式：

path: LayerName (DOT LayerName)+

LayerName: Identifier | STAR

其中 STAR 为“*”，DOT 为“.”。

我们称一个路径中在两个“.”中间的部分叫做一个层级，则 root.a.b.c 为一个层级为 4 的路径。

值得说明的是，在路径中，root 为一个保留字符，它只允许出现在下文提到

的时间序列的开头，若其他层级出现 root，则无法解析，提示报错（当前版本不提示报错，但不建议其他层级出现 root 关键字）。

2.4 时间序列

时间序列是 IoTDB 中的核心概念。时间序列可以被看作产生时序数据的传感器的所在完整路径，在 IoTDB 中所有的时间序列必须以 root 开始、以传感器作为结尾。一个时间序列也可称为一个全路径。

例如，vehicle 种类的 device1 有名为 sensor1 的传感器，则它的时间序列可以表示为：root.vehicle.device1.sensor1。

注意：当前 IoTDB 支持的时间序列必须大于等于四层（之后会更改为两层）。

2.5 前缀路径

前缀路径是指一个时间序列的前缀所在的路径，一个前缀路径包含以该路径为前缀的所有时间序列。例如当前我们有 root.vehicle.device1.sensor1、root.vehicle.device1.sensor2、root.vehicle.device2.sensor1 三个传感器，则 root.vehicle.device1 前缀路径包含 root.vehicle.device1.sensor1、root.vehicle.device1.sensor2 两个时间序列，而不包含 root.vehicle.device2.sensor1。

2.6 带*路径

为了使得在表达多个时间序列或表达前缀路径的时候更加方便快捷，IoTDB 为用户提供带*路径。“*”可以出现在路径中的任何层。按照“*”出现的位置，带*路径可以分为两种：

“*”出现在路径的结尾；

“*”出现在路径的中间；

当“*”出现在路径的结尾时，其代表的是（“*”）+，即为一层或多层“*”。例如 root.vehicle.device1.*代表的是 root.vehicle.device1.*, root.vehicle.device1.*.*, root.vehicle.device1.*.*.*等所有以 root.vehicle.device1 为前缀路径的大于等于 4 层的路径。

当“*”出现在路径的中间，其代表的是“*”本身，即为一层。例如 root.vehicle.*.sensor1 代表的是以 root.vehicle 为前缀，以 sensor1 为后缀，层次等于 4 层的路径。

注意：*不能放在路径开头。

注意：*放在末尾时与前缀路径表意相同，例如 root.vehicle.*与 root.vehicle

为相同含义。

2.7 数据的列

一个时间序列也称为一个数据的列。

2.8 存储组

存储组是用于让用户定义如何在磁盘上组织和隔离不同的时间序列数据的。属于同一个存储组的时间序列将被不断地写入相应文件夹的同一文件中，该文件可能因用户命令或系统策略而关闭，这些传感器接下来到来的数据会存入同一文件夹的新文件中；属于不同存储组的时间序列则被存储于不同的文件夹。

用户可以将任意前缀路径设置成存储组。如有 4 条时间序列 root.vehicle.d1.s1, root.vehicle.d1.s2, root.vehicle.d2.s1, root.vehicle.d2.s2，路径 root.vehicle 下的两个设备 d1,d2 可能属于同一个业主，或者同一个厂商，因此关系紧密。这时候就可以将前缀路径 root.vehicle 指定为一个存储组，这将使得 IoTDB 将其下的所有设备的数据存储在同一个文件夹下。未来 root.vehicle 下增加了新的设备，也将属于该存储组。

注意：不允许将一个完整路径(如上例的 root.vehicle.d1.s1)设置成存储组。

设置合理数量的存储组可以带来性能的提升：既不会因为产生过多的存储文件（夹）导致频繁切换 IO 降低系统速度（并且会占用大量内存且出现频繁的内存-文件切换），也不会因为过少的存储文件夹（降低了并发度从而）导致写入命令阻塞。

用户应根据自己的数据规模和使用场景，平衡存储文件的存储组设置，以达到更好的系统性能。

注意：一个时间序列其前缀必须属于某个存储组。在创建时间序列后，用户必须设定该序列属于哪个存储组（Storage Group）。只有设置了存储组的时间序列才可以被持久化在磁盘上。

一个前缀路径一旦被设定成存储组后就不可以再更改这个存储组的设置。

一个存储组设定后，其对应的前缀路径的所有父层级也不允许再设置存储组（如，root.vehicle.d0 设置存储组后，root.vehicle 层级不允许被设置为存储组）。

2.9 值

一个时间序列的值是由实际中的传感器向 IoTDB 发送的数值。这个值可以按照数据类型被 IoTDB 存储，同时用户也可以针对这个值的数据类型选择压缩方

式，以及对应的编码方式。数据类型与对应编码的详细信息请参见本文第 5.2 节。

2.10 时间戳

时间戳是一个数据到来的时间点。在 IoTDB 中每一个值都与时间戳绑定为一个时间-值对(timestamp, value)。IoTDB 时间戳分为两种类型，一种为 LONG 类型，一种为 DATETIME 类型（包含 DATETIME-INPUT, DATETIME-DISPLAY 两个小类）。

在用户在输入时间戳时，可以使用 LONG 类型的时间戳或 DATETIME-INPUT 类型的时间戳，其中 DATETIME-INPUT 类型的时间戳支持格式见表格 2.1。

yyyy-MM-dd HH:mm:ss
yyyy/MM/dd HH:mm:ss
yyyy.MM.dd HH:mm:ss
yyyy-MM-dd'T'HH:mm:ss
yyyy/MM/dd'T'HH:mm:ss
yyyy.MM.dd'T'HH:mm:ss
yyyy-MM-dd HH:mm:ssZZ
yyyy/MM/dd HH:mm:ssZZ
yyyy.MM.dd HH:mm:ssZZ
yyyy-MM-dd'T'HH:mm:ssZZ
yyyy/MM/dd'T'HH:mm:ssZZ
yyyy.MM.dd'T'HH:mm:ssZZ
yyyy/MM/dd HH:mm:ss.SSS
yyyy-MM-dd HH:mm:ss.SSS
yyyy.MM.dd HH:mm:ss.SSS
yyyy/MM/dd'T'HH:mm:ss.SSS
yyyy-MM-dd'T'HH:mm:ss.SSS
yyyy.MM.dd'T'HH:mm:ss.SSS
yyyy-MM-dd HH:mm:ss.SSSZZ
yyyy/MM/dd HH:mm:ss.SSSZZ
yyyy.MM.dd HH:mm:ss.SSSZZ
yyyy-MM-dd'T'HH:mm:ss.SSSZZ
yyyy/MM/dd'T'HH:mm:ss.SSSZZ
yyyy.MM.dd'T'HH:mm:ss.SSSZZ
ISO8601 标准时间格式

表格 2.1 DATETIME-INPUT 类型支持格式

IoTDB 在显示时间戳时可以支持 LONG 类型以及 DATETIME-DISPLAY 类

型，其中 DATETIME-DISPLAY 类型可以支持用户自定义时间格式。自定义时间格式的语法见表格 2.2。

Symbol	Meaning	Presentation	Examples
G	era	text	AD
C	century of era (>=0)	number	20
Y	year of era (>=0)	year	1996
x	weekyear	year	1996
w	week of weekyear	number	27
e	day of week	number	2
E	day of week	text	Tuesday; Tue
y	year	year	1996
D	day of year	number	189
M	month of year	month	July; Jul; 07
d	day of month	number	10
a	halfday of day	text	PM
K	hour of halfday (0~11)	number	0
h	clockhour of halfday (1~12)	number	12
H	hour of day (0~23)	number	0
k	clockhour of day (1~24)	number	24
m	minute of hour	number	30
s	second of minute	number	55
S	fraction of second	millis	978
z	time zone	text	Pacific Standard Time; PST
Z	time zone offset/id	zone	-0800; -08:00; America/Los_Angeles
'	escape for text	delimiter	
''	single quote	literal	'

表格 2.2 DATETIME-DISPLAY 自定义时间格式的语法

2.11 数据点

一个时间序列是由若干个数据点按时间顺序排序组成的。一个数据点是一个时间戳-值对(timestamp, value)。

第3章 IoTDB 安装方法

3.1 安装包结构介绍

IoTDB 安装包结构遵循以下目录：

/bin 可执行文件目录

start-client.sh 客户端启动脚本（适用于 Linux 系统及 MacOS 系统）

start-client.bat 客户端启动脚本（适用于 Windows 系统）

start-server.sh 服务器启动脚本（适用于 Linux 系统及 MacOS 系统）

start-server.bat 服务器启动脚本（适用于 Windows 系统）

stop-server.sh 服务器停止脚本（适用于 Linux 系统及 MacOS 系统）

stop-server.bat 服务器停止脚本（适用于 Windows 系统）

import-csv.sh csv 数据文件导入脚本（适用于 Linux 系统及 MacOS 系统，当前版本不支持）

import-csv.bat csv 数据文件导入脚本（适用于 Windows 系统，当前版本不支持）

export-csv.sh csv 数据文件导出脚本（适用于 Linux 系统及 MacOS 系统，当前版本不支持）

export-csv.bat csv 数据文件导出脚本（适用于 Windows 系统，当前版本不支持）

/conf

logback.xml 日志配置文件

iotdb-env.sh 环境变量配置文件（Linux 系统及 MacOS 系统）

iotdb-env.bat 环境变量配置文件（Windows 系统）

tsfile-format.properties IoTDB 文件格式配置文件

iotdb-engine.properties IoTDB 引擎配置文件

/lib 项目依赖目录

/LICENSE 代码 LICENSE（当前版本没有此文件）

3.2 安装环境要求

安装前需要保证电脑上配有 JDK>=1.8 的运行环境。

如需从源码进行编译，还需要 Maven>=3.0 的运行环境。

3.3 安装过程

IoTDB 支持多种安装途径。用户可以使用三种方式对 IoTDB 进行安装——下载二进制可运行程序、使用云端版本、使用源码编译 。

3.3.1 从官网下载二进制可运行程序

用户可以从 http://tsfile.org/download 上下载已经编译好的可执行程序，当前发布的可运行程序为 alpha 测试版，该压缩包包含了 IoTDB 系统运行所需的所有必要组件。

下载后，用户可使用以下命令对 IoTDB 的压缩包进行解压:

（Linux 下和 MacOS 下）

Shell > uzip iotdb-alpha.zip

（Windows 下）

使用解压缩工具解压 iotdb-alpha.zip

解压后文件夹内容见图 3.1。

图 3.1 解压后文件夹内容

3.3.2 使用云端版本

用户可以使用已经制作好的 IoTDB 云端进行对 IoTDB 进行一键化配置。该云端版本部署在腾讯云上，网址是 https://www.qcloud.com/。（注：当前 IoTDB 自定义镜像需要好友分享才可使用，如需使用云端版本，请联系 IoTDB）操作步骤如下：

步骤一：购买云服务器（见图 3.2）

图 3.2 购买云服务器

步骤二：选择自定义配置（见图 3.3），根据实际需求选择云服务器所需的

CPU、内存等硬件配置（建议：内存至少选择 1GB）。

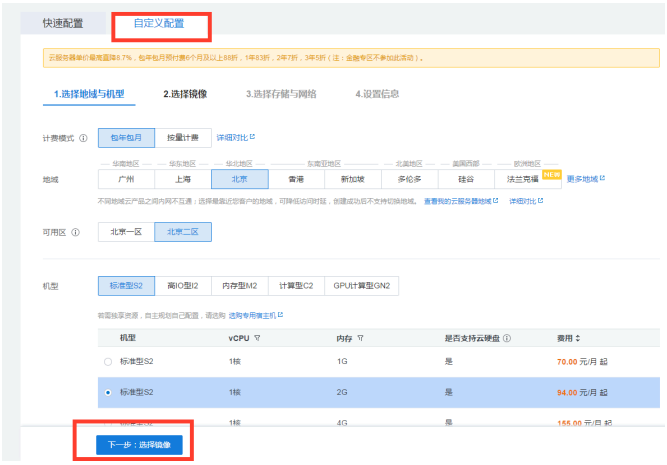

图 3.3 选择自定义配置

步骤三：选择自定义镜像 IoTDB，在“自定义镜像”一栏中点选“IoTDB”（见图 3.4）

图 3.4 选择自定义镜像 IoTDB

步骤四：选择存储与网络（见图 3.5，注意：带宽至少选择 1Mbps）

图 3.5 选择存储与网络

步骤五：设置信息并购买（见图 3.6），购买成功之后，服务器会在几分钟之后自动启动。这个时候 IoTDB 也已经启动起来了。

图 3.6 设置信息并购买

此时，用户可以远程连接服务器使用 IoTDB 的客户端对 IoTDB Server 进行操作，也可以使用该配置好的腾讯云的 3000 端口直接查看 Grafana 工具为用户呈现的可视化数据（当前版本不支持）。Grafana 的详细使用方法请参见本文第 7.3 节。

3.3.3 使用源码编译

除以上两种方式外，用户还可以使用从 Git 仓库克隆源码进行编译的方式安装 IoTDB。使用源码编译 IoTDB 时需要首先用 mvn 编译 TsFile 项目，具体操作请参见：https://github.com/thulab/tsfile/wiki/Installation。同时还需要用 mvn 编译 iotdb-jdbc 项目，具体操作请参见：https://github.com/thulab/iotdb-jdbc。

TsFile 项目编译完成后，执行以下命令，即可获取 IoTDB 源码：

（安装 git 命令行工具后）

Shell > git clone https://github.com/thulab/iotdb.git

（当前为未发布版本，需要获取权限后才可克隆源码）

源码克隆后，进入到源码文件夹目录下，使用以下命令进行编译：

Shell > mvn clean package -Dmaven.test.skip=true

编译过程如图 3.7。

```
$ mvn clean package -Dmaven.test.skip=true
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building tsfiledb 0.0.1
[INFO] ------------------------------------------------------------------------
Downloading: https://repo.maven.apache.org/maven2/org/antlr/antlr3-maven
3.5.2/antlr3-maven-plugin-3.5.2.pom
Downloaded: https://repo.maven.apache.org/maven2/org/antlr/antlr3-maven-
.5.2/antlr3-maven-plugin-3.5.2.pom (8.3 kB at 4.6 kB/s)
Downloading: https://repo.maven.apache.org/maven2/org/antlr/antlr-master
ntlr-master-3.5.2.pom
Downloaded: https://repo.maven.apache.org/maven2/org/antlr/antlr-master/
tlr-master-3.5.2.pom (12 kB at 30 kB/s)
Downloading: https://repo.maven.apache.org/maven2/org/antlr/antlr3-maven
3.5.2/antlr3-maven-plugin-3.5.2.jar
Downloaded: https://repo.maven.apache.org/maven2/org/antlr/antlr3-maven-
.5.2/antlr3-maven-plugin-3.5.2.jar (13 kB at 31 kB/s)
```

图 3.7 编译过程截图

当出现以下输出时，即为编译成功，如图 3.8。

```
[INFO] Executing tasks

main:
     [copy] Copying 1 file to D:\CodeRespository\tsfiledb\tsfiledb\lib
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------------
-------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
-------
[INFO] Total time: 20.838 s
[INFO] Finished at: 2017-08-29T16:09:58+08:00
[INFO] Final Memory: 37M/464M
[INFO] ------------------------------------------------------------------------
-------
```

图 3.8 编译成功截图

3.4 测试 IoTDB 安装状态

用户可以根据以下操作对 IoTDB 进行简单的试用，若以下操作均无误，则说明 IoTDB 安装成功。

在后文中，记$IOTDB_HOME 表示 IoTDB 的安装目录路径。

3.4.1 启动服务器

用户可以使用$IOTDB_HOME/bin 文件夹下的 start-server 脚本启动 IoTDB 服务器。

Linux 系统与 MacOS 系统启动命令如下：

Shell > ./bin/start-server.sh

Windows 系统启动命令如下：

Shell > bin/start-server.bat

启动后出现如图 3.9 提示即为启动成功。

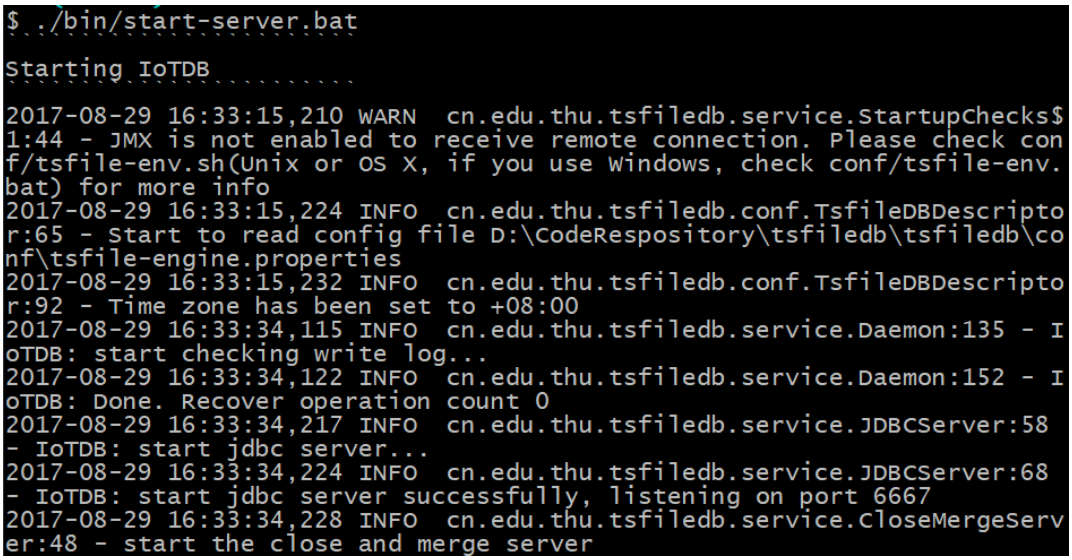

图 3.9 系统启动成功状态

当服务器输出 log 中包含 ERROR 输出时，服务器启动不成功。相关错误可查询《IoTDB 故障排除及错误信息文档》（暂缺）。

3.4.2 测试客户端

初始安装后的 IoTDB 中有一个默认用户：root，默认密码为 root。用户可以使用该用户尝试运行 IoTDB 客户端以测试服务器是否正常启动。客户端启动脚本为$IOTDB_HOME/bin 文件夹下的 start-client 脚本。启动脚本时需要指定运行 IP 和 PORT。以下为服务器在本机运行，且用户未更改运行端口号的示例。若用户尝试连接远程服务器或更改了服务器运行的端口号，请在-h 和-p 项处使用自己的 IP 和 PORT。

Linux 系统与 MacOS 系统启动命令如下：

Shell > ./bin/start-client.sh -h 127.0.0.1 -p 6667 -u root -pw

Windows 系统启动命令如下：

Shell > bin/start-client.bat -h 127.0.0.1 -p 6667 -u root –pw

回车后输入 root 用户的密码，即可成功启动客户端。启动后出现如图 3.10 提示即为启动成功。

图 3.10 客户端启动成功状态

连接成功后，用户可使用 Client 输入命令操作 IoTDB Server。

输入 quit 或 exit 可退出 Client 结束本次会话，Client 输出 quit normally 表示退出成功，如图 3.11。

```
IoTDB> login successfully
IoTDB> quit
quit normally
```

图 3.11 客户端退出成功状态

3.4.3 停止服务器

服务器停止脚本为$IOTDB_HOME/bin 文件夹下的 stop-server 脚本。

Linux 系统与 MacOS 系统停止命令如下：

Shell > ./bin/stop-server.sh

Windows 系统停止命令如下：

Shell > bin/stop-server.bat

当显示如图 3.12 状态时即为服务器停止成功。

```
➜  tsfiledb git:(master) ./bin/stop-server.sh
close IoTDB
```

图 3.12 服务器停止成功状态

第4章 IoTDB Server 管理与配置概述

4.1 配置选项

为方便 IoTDB Server 的配置与管理，IoTDB Server 为用户提供三种配置项，使得用户可以在启动服务器时对其进行配置。这些配置选项及变量分别为：环境配置项、系统配置项、日志配置项。三种配置项的配置文件均位于$IOTDB_HOME/conf 文件下。

4.1.1 环境配置项

环境配置项主要用于对 IoTDB Server 运行的 Java 环境相关参数进行配置，如 JVM 相关配置。IoTDB Server 启动时，此部分配置会被传给 JVM。用户可以通过查看 iotdb-env.sh(或 iotdb-env.bat)文件查看环境配置项内容。详细配置项说明请看本文第 4.4 节。

4.1.2 系统配置项

系统配置项是 IoTDB Server 运行的核心配置，它主要用于设置 IoTDB Server 文件层和引擎层的参数，便于用户根据自身需求调整 Server 的相关配置，以达到较好的性能表现。系统配置项可分为两大模块：文件层配置项和引擎层配置项。用户可以通过查看 tsfile-format.properties, iotdb-engine.properties,文件查看和修改两种配置项的内容。详细内容可以参见本文第 4.4 节。

4.1.3 日志配置项

日志配置项是 IoTDB Server 运行时的日志相关配置。IoTDB 将根据日志配置项输出相应的系统日志，默认日志级别为 INFO 级别。IoTDB Server 启动时，此部分配置会被加载。

默认的日志输出路径为$IOTDB_HOME/logs/文件夹。日志记录采用分级记录，级别与日志文件名相对应，不同级别的日志信息记录到不同的日志文件中。日志文件按日期记录，同一天内，若日志文件大小超过设定的最大文件大小，则按 0、1、2.....顺序命名。

用户可以通过查看 logback.xml 文件查看和修改日志配置项内容。详细内容可以参见本文 4.4 节。

4.2 配置方式

当前 IoTDB 可以修改配置文件相关配置项对环境配置项、系统配置项和日志配置项进行设置：

IoTDB 配置文件的默认位置位于$IOTDB_HOME/conf 文件夹下，其中涉及 server 配置的共有 4 个文件，分别为：iotdb-env.sh, tsfile-format.properties, iotdb-engine.properties, logback.xml。用户可以通过更改其中的配置项对系统运行的相关配置项进行配置。

4 个文件的说明如下：

- **iotdb-env.sh：** 环境配置项的默认配置文件。用户可以在文件中配置 JAVA-JVM 的相关系统配置项。详细配置项说明请看本文 4.4 章。
- **tsfile-format.properties：**IoTDB 文件层配置项的默认配置文件。用户可以在文件中配置 IoTDB 存储时 TsFile 文件的相关信息，如每次将内存中的数据写入到磁盘时的数据大小(group_size_in_byte)，内存中每个列打一次包的大小(page_size_in_byte)等。该文件的格式如图 4.1 所示。详细配置项说明请看本文 4.4 章。

```
# Memory size threshold for flushing to disk or HDFS, default value is 128MB
group_size_in_byte=134217728

# The memory size for each series writer to pack page, default value is 1MB
page_size_in_byte=1048576

# The maximum number of data points in a page, defalut 1024*1024
max_number_of_points_in_page=1048576
```

图 4.1 iotdb-format.properties 文件格式

- **iotdb-engine.properties：**IoTDB 引擎层配置项的默认配置文件。用户可以在文件中配置 IoTDB 引擎运行时的相关参数，如 JDBC 服务监听端口(rpc_port)、overflow 数据文件存储目录(overflow_data_dir)等。该文件的格式如图 4.2 所示。详细配置项说明请看本文 4.4 章。

```
# port which JDBC server listens to
rpc_port=6667

# Write ahead log configuration

# is write ahead log enable
enable_wal=true

# When the total number of write ahead log in the file and memory reaches the specified size, all
# Increase this value, it will lead to short write pause. Decrease this value, it will increase IO
wal_cleanup_threshold=500000
```

图 4.2 iotdb-engine.properties 文件格式

- **logback.xml：**日志配置项的默认配置文件。用户可以在文件中配置 IoTDB

的日志配置项。详细配置项说明请看本文第 4.4 章。

4.3 配置项列表

表格 4.1 中提供了 IoTDB Server 中涉及的全部配置项。其中列出了配置项是否可以通过配置文件进行配置（Conf-File）以及变量生效的时间（Effective）。其中生效时间（Effective）栏中，1 代表配置在重启服务器后生效，2 代表该配置项对已完成的写操作无效，对后续的写操作有效。

Name	Conf-File	Effective
compressor	Yes	2
data_dir	Yes	1
enable_wal	Yes	1
fetch_size	Yes	1
float_precision	Yes	2
flush_wal_period_in_ms	Yes	1
flush_wal_threshold	Yes	1
gourp_size_in_byte	Yes	2
jmx_local	Yes	1
jmx_port	Yes	1
max_number_of_points_in_page	Yes	2
max_opened_folder	Yes	1
max_string_length	Yes	2
merge_concurrent_threads	Yes	1
page_size_in_byte	Yes	2
period_time_for_flush_in_second	Yes	1
period_time_for_merge_in_second	Yes	1
rpc_port	Yes	1
time_series_data_type	Yes	2
time_series_encoder	Yes	2
value_encoder	Yes	2
wal_cleanup_threshold	Yes	1

表格 4.1 配置项及配置项属性

4.4 配置项详细说明

名字 compressor	
描述	数据压缩方法，可选 UNCOMPRESSED，SNAPPY 两种，默认 UNCOMPRESSED，即不压缩
所属配置项	系统配置项-文件层
类型	String

默认值	UNCOMPRESSED

名字 data_dir	
描述	IoTDB 数据存储路径
所属配置项	系统配置项-引擎层
类型	String
默认值	data

名字 enable_wal	
描述	是否开启写前日志，默认值为 true 表示开启，配置成 false 表示关闭
所属配置项	系统配置项-引擎层
类型	Bool
默认值	true

名字 fetch_size	
描述	批量读取数据的时候，每一次读取数据的数量。某次会话中，用户可以在使用时自己设定，此时仅在该次会话中生效
所属配置项	系统配置项-引擎层
类型	Int
默认值	10000

名字 float_precision	
描述	浮点数精度
所属配置项	系统配置项-文件层
类型	Int
默认值	2

名字 flush_wal_period_in_ms	
描述	写前日志定期刷新到磁盘的周期，单位毫秒，有可能丢失至多 flush_wal_period_in_ms 毫秒的操作
所属配置项	系统配置项-引擎层
类型	Int
默认值	10

名字 flush_wal_threshold	
描述	写前日志达到一定数量之后，刷新到磁盘，有可能丢失至多 flush_wal_threshold 个操作
所属配置项	系统配置项-引擎层
类型	Int
默认值	10000

名字 group_size_in_byte	
描述	每次将内存中的数据写入到磁盘时的大小，默认 128MB
所属配置项	系统配置项-文件层
类型	Int
默认值	134217728

名字 JMX_ LOCAL	
描述	JMX 监控模式，配置为 yes 表示仅允许本地监控，设置为 no 的时候表示允许远程监控
所属配置项	环境配置项
类型	String
默认值	yes

名字 JMX_PORT	
描述	JMX 监听端口
所属配置项	环境配置项
类型	String
默认值	31999

名字 max_number_of_points_in_page	
描述	一个页中最多包含的数据点数量，默认 1KB
所属配置项	系统配置项-文件层
类型	Int
默认值	1048576

名字 max_opened_folder	
描述	最大同时打开的文件夹数量值变大，占用内存变多，IO 随机读写变小，文件分块(即 group)更加整齐值越小，占用内存越少，IO 随机读写变多，文件块大小不足 group 的概率变大 group_size_in_byte * max_opened_folder = 内存的最大占用量理论值 对于一个应用，folder 的总量等于 SQL 中设置的 storage_group 的数量
所属配置项	系统配置项-引擎层
类型	Int
默认值	100

名字 max_string_length	
描述	针对字符串类型的数据，单个字符串最大长度
所属配置项	系统配置项-文件层
类型	Int

默认值	128

名字 merge_concurrent_threads	
描述	overflow 数据进行合并的时候最多可以起多少个线程来 merge 值越大，对 IO 和 CPU 消耗越多。值越小，当 overflow 数据过多时，磁盘占用量越大，读取会变慢
所属配置项	系统配置项-引擎层
类型	Int
默认值	10

名字 page_size_in_byte	
描述	内存中每个列打一次包的大小，默认 1MB
所属配置项	系统配置项-文件层
类型	Int
默认值	134217728

名字 period_time_for_flush_in_second	
描述	IoTDB 定时 close 文件的时长设定。每隔设置的时间，系统会自动将内存中的数据刷入磁盘，并将当前打开的所有文件流封口
所属配置项	系统配置项-引擎层
类型	Int
默认值	3600

名字 period_time_for_merge_in_second	
描述	IoTDB 在运行时有两部分数据存在于内存：overflow 和 bufferwrite。设定此项后，系统会自动每隔一段时间合并两部分数据
所属配置项	系统配置项-引擎层
类型	Int
默认值	7200

名字 rpc_port	
描述	jdbc 服务监听端口
所属配置项	系统配置项-引擎层
类型	Int
默认值	6667

名字 time_series_data_type	
描述	时间戳数据类型，可选 INT32, INT64
所属配置项	系统配置项-文件层

类型	String
默认值	INT64

名字 time_series_encoder	
描述	时间列编码方式，默认二阶差分(TS_2DIFF)，可选 PLAIN, RLE(游程编码，run-length encoding)
所属配置项	系统配置项-文件层
类型	String
默认值	TS_2DIFF

名字 value_encoder	
描述	值列编码方式，默认 PLAIN，对于 int, long, float, double 类型的数据，还可以选 RLE(游程编码，run-length encoding)和 TS_2DIFF(二阶差分)
所属配置项	系统配置项-文件层
类型	String
默认值	PLAIN

名字 wal_cleanup_threshold	
描述	当文件中和内存中的日志总数量达到指定大小后，对所有日志进行压缩并去掉无用的记录日志。该值过大会导致短暂的写入暂停，过小会增加 IO 和 CPU 消耗
所属配置项	系统配置项-引擎层
类型	Int
默认值	500000

4.5 系统服务器部署与启动

IoTDB Server 的启动脚本是位于$IOTDB_HOME/bin 目录下的 start-server 文件。在启动前若需要配置 IoTDB 可以更改$IOTDB_HOME/conf 文件夹下的配置文件，详细说明及配置方法参见本文第 4 章。

完成配置后，可以启动 IoTDB Server。

Linux 系统与 MacOS 系统启动命令如下：

Shell > ./bin/start-server.sh

Windows 系统启动命令如下：

Shell > bin/start-server.bat

启动后出现如图 4.3 提示即为启动成功。

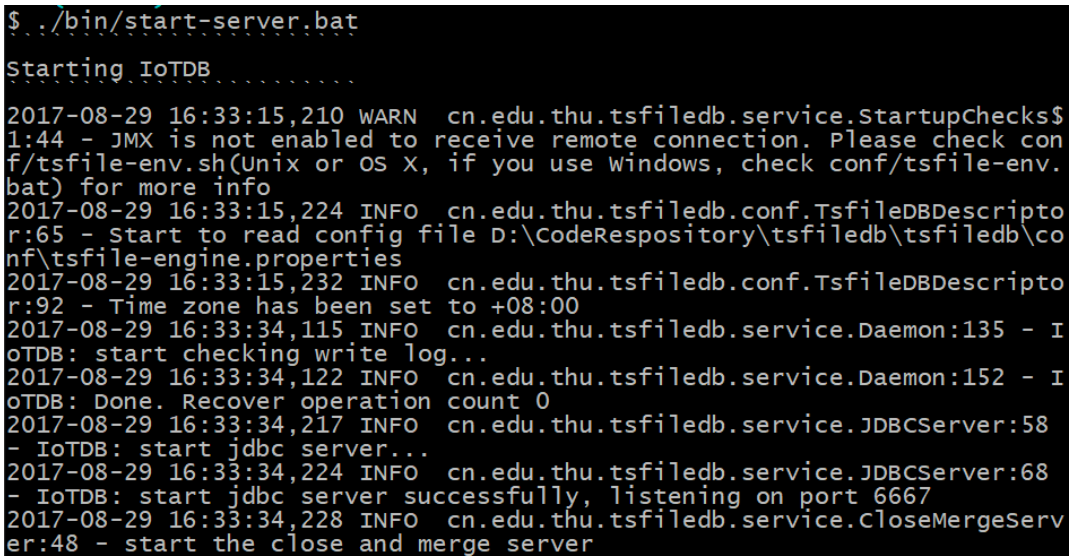

图 4.3 系统启动成功状态

当服务器输出 log 中包含 ERROR 输出时，服务器启动不成功。相关错误可查询《IoTDB 故障排除及错误信息文档》（暂无）。

第5章 IoTDB 详细使用说明

5.1 支持的主要功能

IoTDB 提供包含 DDL(Data Definition Language)、DML(Data Management Language)、AUTH(Authentation Management)、SESSION(Session Management)的四大类操作。

- DDL 操作主要包含时间序列的相关管理操作，方便使用者对时间序列的属性进行设定与管理；
- DML 操作主要包含用户对时序数据的插入、更新、删除、查询操作，为使用者提供针对时间序列数据的高效管理接口；
- AUTH 操作主要包含用户管理以及权限管理操作，方便使用者对 IoTDB 进行多用户管理及多权限管理；
- SESSION 操作主要包含会话级别的操作，方便使用者对于每个客户端与服务器建立的连接单独设定参数。

本章第 5.2 节将介绍 IoTDB 支持的数据类型及编码，第 5.3 节将介绍 DDL 操作相关功能及用法，第 5.4 节将介绍 DML 操作相关功能及用法，第 5.5 节将介绍 AUTH 操作相关功能及用法，第 5.6 节将介绍 SESSION 操作相关功能及用法。

除此以外，IoTDB 还为用户提供一些辅助工具，包括

- 可视化的结果查询工具(基于 Grafana);
- 数据云端上传工具；
- 数据导入工具；
- 数据导出工具；

使用以上工具请参考本文第 6 章。

5.2 支持的数据类型及编码

5.2.1 数据类型

IoTDB 支持 BOOLEAN（布尔值）, INT32（整型）, INT64（长整型）, FLOAT（单精度浮点数）, DOUBLE（双精度浮点数）, TEXT（字符串）, ENUMS（枚举值，0.1.2 版本不支持枚举值）一共 7 种数据类型。

其中 FLOAT 与 DOUBLE 类型可以指定 MAX_POINT_NUMBER，该项为浮

点数的小数点后位数，不指定则根据配置文件 tsfile-format.properties 文件中的 float_precision 项配置（配置方法参见本文第 4 章）。ENUM 类型必须指定 ENUM_VALUES，该项为该枚举类型的所有值。具体指定方式请参见本文第 8.2.1 节。

当系统中用户输入的数据类型与该时间序列的数据类型不对应时，系统会提醒类型错误。

5.2.2 编码类型

为了提高数据的存储效率，需要在数据写入的过程中对数据进行编码，从而减少磁盘空间的使用量。在写数据以及读数据的过程中都能够减少 I/O 操作的数据量从而提高性能。IoTDB 支持七种针对不同类型的数据的编码方法：

- PLAIN 编码（PLAIN）
- 二阶差分编码（TS_2DIFF）
- 游程编码（RLE）
- DFT 编码（DFT）
- GORILLA 编码（GORILLA）
- BITMAP 编码（BITMAP）

5.2.2.1 PLAIN 编码

PLAIN 编码，默认的编码方式，即不编码，支持多种数据类型，但是存储效率较低。

5.2.2.2 二阶差分编码

二阶差分编码，比较适合编码单调递增或者递减的序列数据，不适合编码波动较大的数据。

二阶差分编码也可用于对浮点数进行编码，比较适合存储某些浮点数值连续出现、单调调递增或者递减的序列数据，不适合存储对小数点后精度要求较高以及前后波动较大的序列数据。

5.2.2.3 游程编码

游程编码，比较适合存储某些整数值连续出现的序列，不适合编码大部分情况下前后值不一样的序列数据。

游程编码也可用于对浮点数进行编码，比较适合存储某些浮点数值连续出现的序列数据，不适合存储对小数点后精度要求较高以及前后波动较大的序列数据。

5.2.2.4 DFT 编码

DFT 编码，比较适合允许有损压缩的序列数据，希望对数据进行无损编码的时候不建议采用 DFT。

注意：v0.1.2 版本上暂时没有该编码。

5.2.2.5 GORILLA 编码

GORILLA 编码，比较适合编码前后值比较接近的浮点数序列，不适合编码前后波动较大的数据。

5.2.2.6 BITMAP 编码

BITMAP 编码，只适合编码枚举类型的数据，而且枚举值的数量较少。

5.2.3 数据类型与编码的对应关系

本文第 5.2.2 节中介绍的七种编码适用于不同的数据类型，若对应关系错误，则无法正确创建时间序列（当前该约束不生效）。数据类型与支持其编码的编码方式对应关系如表格 5.1。

数据类型	支持的编码
BOOLEAN	PLAIN
INT32	PLAIN, RLE, TS_2DIFF
INT64	PLAIN, RLE, TS_2DIFF
FLOAT	PLAIN, RLE, TS_2DIFF, GORILLA
DOUBLE	PLAIN, RLE, TS_2DIFF, GORILLA
TEXT	PLAIN
ENUMS	BITMAP

表格 5.1 数据类型与支持其编码的对应关系

5.3 DDL 相关功能及用法

DDL 主要包含时间序列的相关管理操作。时间序列是 IoTDB 中的核心概念。时间序列可以被看作产生时序数据的传感器的所在路径，在 IoTDB 中所有的时间序列均位于 root 根路径下。例如，vehicle 种类的 device1 有名为 sensor1 的传感器，则它的时间序列可以表示为：root.vehicle.device1.sensor1。

5.3.1 创建时间序列

使用 CREATE TIMESERIES 语句可以创建时间序列，详细语法参见本文第 8.2.1 节。示例代码如下：

IoTDB > CREATE TIMESERIES root.vehicle.d0.s0 WITH DATATYPE=INT32, ENCODING=RLE;

以上示例代码创建了路径为 vehicle.d0.s0 的传感器 s0 的时间序列，其中 s0 的数据类型为 INT32，编码方式为 RLE。

执行成功后会出现 execute successfully 的提示，代表时间序列创建已经完成。如图 5.1。

```
IoTDB> CREATE TIMESERIES root.vehicle.d0.s0 WITH DATATYPE=INT32, ENCODING=RL
E
execute successfully.
```

图 5.1 create timeseries 语句执行成功结果

值得说明的是，在创建时间序列时，DATATYPE 与 ENCODING 需要满足对应关系，即 DATATYPE 需要选择支持其类型的编码方式，否则系统会报出错误无法创建（当前未实现错误提示）。对应关系参见本文第 5.2.3 节。

除此之外，时间序列不可重复创建，若创建的时间序列路径已经存在，系统会给出提示，如图 5.2。

```
IoTDB> CREATE TIMESERIES root.vehicle.d0.s0 WITH DATATYPE=INT32, ENCODING=RL
E
statement error: Timeseries root.vehicle.d0.s0 already exist
```

图 5.2 create timeseries 语句重复创建报错提示

5.3.2 删除时间序列

使用 DELETE TIMESERIES 语句可以删除已经创建的时间序列，详细语法参见本文第 8.2.2 节。示例代码如下：

IoTDB > DELETE TIMESERIES root.vehicle.d0.s0;

以上示例代码删除了路径为 root.vehicle.d0.s0 的传感器 s0 的时间序列。

执行成功后会出现 execute successfully 的提示，代表时间序列删除已经完成。如图 5.3。

```
IoTDB> DELETE TIMESERIES root.vehicle.d0.s0
execute successfully.
```

图 5.3 delete timeseries 语句执行成功结果

当所需删除的时间序列不存在时，系统会返回 ERROR 告知该时间序列不存在，如图 5.4。

```
IoTDB> DELETE TIMESERIES root.vehicle.d0.s1
statement error: Timeseries does not exist and cannot be delete its metadata
 and data
```

图 5.4 delete timeseries 语句 timeseries 不存在报错提示

除此之外，DELETE TIMESERIES 语句也支持对时间序列的前缀路径以及带星路径（即，root.vehicle.d0.*等，关于带*路径的定义，请参见本文第 2.6 节）进行删除，执行后，该前缀路径下的所有传感器均被删除。示例代码如下：

IoTDB > DELETE TIMESERIES root.vehicle.d0;

以上示例代码删除了路径为 root.vehicle.d0 下的全部时间序列。

5.3.3 查询时间序列

使用 SHOW TIMESERIES 语句可以查看已经创建的时间序列，该命令会以 JSON 格式为用户展示已经创建的时间序列的数据类型、编码方式、时间序列存储组等一系列参数。详细语法参见本文第 8.2.3 节。示例代码如下：

IoTDB > SHOW TIMESERIES;

执行结果如图 5.5。

```
IoTDB> show timeseries
===  Timeseries Tree  ===

root:{
    vehicle:{
        d0:{
            s0:{
                DataType: INT32,
                Encoding: RLE,
                args: {},
                FileName: null
            }
        }
    }
}
```

图 5.5 show timeseries 语句执行成功结果

此结果表明此时 IoTDB 中有一个路径为 root.vehicle.d0.s0 的时间序列，数据类型为 INT，编码方式为 RLE。

5.3.4 设置存储组

在创建时间序列后，用户必须设定存储组（Storage Group）。关于存储组的详细解释参见本文第 2.8 节。

用户可以使用 SET STORAGE GROUP 语句设置存储组，详细语法请参见本文第 8.2.4 节。示例代码如下：

IoTDB > SET STORAGE GROUP TO root.vehicle;

以上示例代码将存储组设置为 root.vehicle 路径，位于该路径下的所有数据都将被存储在一个路径下。

执行成功后会出现 execute successfully 的提示，代表时间序列创建已经完成。

如图 5.6。

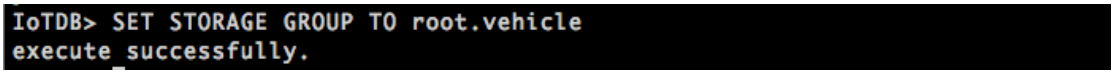

图 5.6 设置存储组语句执行成功结果

当指定的存储组路径不存在时，系统会返回 ERROR 告知该路径不存在，如图 5.7。

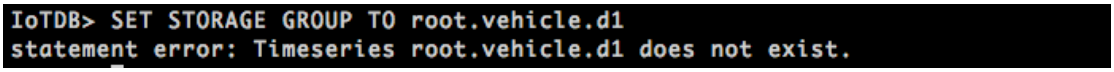

图 5.7 设置存储组不存在报错提示

当指定的存储组已经被设置过，系统会返回 ERROR 告知该存储组已经被设置过，如下图 5.8。

图 5.8 设置存储组已存在报错提示

5.4 DML 相关功能及用法

DML 操作主要包含用户对数据的增删改查功能。

5.4.1 插入数据

使用 INSERT 语句可以向指定的已经创建的时间序列中插入数据。在插入数据时，用户可以根据自身需求选择使用长整型的时间戳、DATETIME-INPUT 类型的时间戳、now()函数，在 INSERT、UPDATE 等语句中均可支持以上类型的时间戳模式。指定时间格式请见本文第 2.10 节。INSERT 语句详细语法请参见本文第 8.3.1 节。示例代码如下：

IoTDB > INSERT INTO root.vehicle.d0(timestamp,s0) VALUES (1,101);

以上示例代码将长整型的 timestamp 以及数值为 101 的传感器数据插入到已经创建的时间序列 root.vehicle.d0.s0 中。执行成功后会出现 execute successfully 的提示，代表数据插入已完成。如图 5.9。

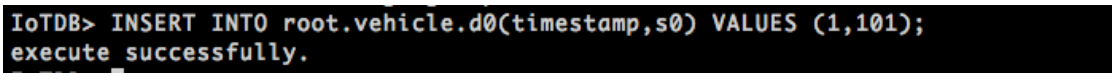

图 5.9 insert 单列数据语句执行成功结果

INSERT 语句还可以支持在同一个时间点下多列数据的插入，若有名为 root.vehicle.d0.s0 与 root.vehicle.d0.s1 两个时间序列，分别为 INT 和 TEXT 类型，同时向 102 时间点插入这两个时间序列的值，示例代码如下：

IoTDB > INSERT INTO root.vehicle.d0(timestamp, s0, s1) VALUES (102, 80,

'PASS');

执行成功后会出现 execute successfully 的提示，代表数据插入已完成，如图 5.10。

```
IoTDB> INSERT INTO root.vehicle.d0(timestamp, s0, s1) VALUES (102, 80, 'PASS
');
execute successfully.
```

图 5.10 insert 多列数据语句执行成功结果

若用户向一个不存在的时间序列中插入数据，系统将会返回 ERROR 告知该 Timeseries 路径不存在。如图 5.11。

```
IoTDB> INSERT INTO root.vehicle.d0(timestamp,s3) VALUES (1,101);
statement error: Timeseries root.vehicle.d0.s3 does not exist.
```

图 5.11 insert 语句插入时间序列不存在报错提示

若用户向一个未设定存储组的时间序列中插入数据，系统将会返回 ERROR 告知该时间序列未设置存储组。如图 5.12。

```
IoTDB> INSERT INTO root.vehicle.d0(timestamp,s0) VALUES (1,101);
statement error: Timeseries root.vehicle.d0.s0 does not set storage group, p
lease set storage group first and then do the operation
```

图 5.12 insert 语句插入时间序列未设置存储组报错提示

若用户插入的数据类型与该 Timeseries 对应的数据类型不一致，系统将会返回 ERROR 告知数据类型有误。如图 5.13。

```
IoTDB> insert into root.laptop.d3(timestamp,s1) values(1970-01-01T08:08:19.910,'aaaaa');
statement error: For input string: "'aaaaa'"
```

图 5.13 insert 语句插入时间序列数据类型不一致报错提示

5.4.2 删除数据

使用 DELETE 语句可以删除指定的时间序列中符合时间删除条件的数据，详细语法参见本文第 8.3.3 节。在删除数据时，用户可以选择需要删除的一个或多个时间序列、时间序列的前缀、时间序列带*路径对某时间之前的数据进行删除。示例代码如下：

IoTDB > DELETE FROM root.vehicle.d0.s0 WHERE time < 104;

以上示例代码删除了 root.vehicle.d0.s0 时间序列下所有时间戳值小于 104 的数据。

IoTDB > DELETE FROM root.vehicle.d0.s0, root.vehicle.d0.s1 WHERE time < 104;

以上示例代码删除了 root.vehicle.d0.s0 和 root.vehicle.d0.s1 时间序列下所有时间戳值小于 104 的数据。

IoTDB > DELETE FROM root.vehicle.d0 WHERE time < 104;

以上示例代码删除了 root.vehicle.d0 前缀路径下所有时间序列时间戳值小于 104 的数据。

IoTDB > DELETE FROM root.vehicle.*.s0 WHERE time < 104;

以上示例代码删除了前缀为 root.vehicle，后缀为 s0，路径层次为 4 层的路径下所有时间序列时间戳值小于 104 的数据。

所有的 DELETE 语句执行成功后均会出现 execute successfully 的提示。如图 5.14。

```
IoTDB> DELETE FROM root.vehicle.d0.s0, root.vehicle.d0.s1 WHERE time < 104
execute successfully.
```

图 5.14 delete 数据语句执行成功结果

若用户从一个不存在的时间序列中删除数据，系统将会返回 ERROR 告知该 Timeseries 路径不存在，如图 5.15。

```
IoTDB> DELETE FROM root.vehicle.d0.s2 WHERE time < 104
statement error: Timeseries does not exist and cannot be delete data
```

图 5.15 delete 语句时间序列不存在报错提示

5.4.3 更新数据

使用 UPDATE 语句可以更新指定的时间序列中符合时间更新条件的数据，详细语法参见本文第 8.3.2 节。在更新数据时，用户可以选择需要更新的一个或多个时间序列、时间序列前缀（当前不支持）并指定更新某个时间点或时间段的数据。示例代码如下：

IoTDB > UPDATE root.vehicle.d0.s0 SET VALUE = 100000 WHERE time < 104 and time > 100;

以上示例代码将 root.vehicle.d0.s0 时间序列下时间戳值在(100,104)区间的数据修改为 100000。

执行成功后会出现 execute successfully 的提示。如图 5.16。

```
IoTDB> UPDATE root.vehicle.d0.s0 SET VALUE = 100000 WHERE time < 104 and tim
e > 100
execute successfully.
```

图 5.16 update 语句执行成功结果

若用户更新一个不存在的时间序列中的数据，系统将会返回 ERROR 告知该 Timeseries 路径不存在。如图 5.17。

```
IoTDB> UPDATE root.vehicle.d0.s1 SET VALUE = 100000 WHERE time < 104 and tim
e > 100
statement error: Timeseries root.vehicle.d0.s1 does not exist.
```

图 5.17 update 语句时间序列不存在报错提示

5.4.4 查询数据

使用 SELECT 语句可以按照用户需求对 IoTDB 中的数据进行多种查询。用户可以按设备、传感器、时间、值对指定的时间序列、时间序列前缀、时间序列带*路径进行相应的查询，也可以进行聚合查询和索引查询（还未提供接口）。详细语法参见本文第 8.3.4 节。示例代码如下：

IoTDB > SELECT * FROM root.vehicle;

以上代码查询了 root.vehicle 路径下的全部数据。执行成功后会出现 execute successfully 的提示，并显示查询结果的数据数量。如图 5.18。

```
IoTDB> SELECT * FROM root.vehicle
+-----------------------+------------------+
|                   Time|root.vehicle.d0.s0|
+-----------------------+------------------+
|1970-01-01T08:00:00.001|               101|
+-----------------------+------------------+
record number = 1
execute successfully.
```

图 5.18 select 语句执行成功结果

除此之外，用户还可以进行跨列、聚合查询，并按照时间、设备值选择相应的数据。

在此给出一个聚合查询的示例代码：

IoTDB > SELECT COUNT(s0) FROM root.vehicle.d0;

以上代码查询了 root.vehicle.d0 路径下 s0 设备的数据数目，执行成功后会出现 execute successfully 的提示，并显示查询结果的数据数量。同时 IoTDB 可以支持多个聚合查询，在此给出多聚合查询的示例代码：

IoTDB > SELECT COUNT(s0), MAX_VALUE(s0) FROM root.vehicle.d0;

若用户从一个不存在的时间序列中查询数据，系统将会返回 ERROR 告知该时间序列不存在，如图 5.19。

```
IoTDB> SELECT COUNT(s0) FROM root.vehicle.d1
statement error: Timeseries does not exist.
```

图 5.19 select 语句时间序列不存在报错提示

5.5 AUTH 相关功能及用法

AUTH 操作主要包含用户及角色的管理的功能。为了便于数据库的权限管理，IoTDB 提供了用户、角色、权限的概念。IoTDB 中有若干个默认权限，可以创建多个用户和多个角色。一个用户可以拥有多个角色，一个角色可以对应多

个用户。角色可以拥有多个权限，一个权限也可以对应多个角色。使用时，操作者可以向某一个角色指定权限，然后向用户指定该角色。

（当前 IoTDB 还未实现 AUTH 操作的接口。）

5.6 SESSION 相关功能及用法

SESSION 操作主要包含客户端会话级别的操作，如设置客户端时间戳显示格式、客户端时间戳显示时区、强制合并文件、强制刷新数据到磁盘。

5.6.1 时区设置

IoTDB 提供了对服务器端的时区配置：IoTDB Server 启动时会根据配置文件设置服务器时区。用户对数据进行读写时若没有指定数据的时区，则数据的时间戳根据服务器端时区进行解析。同时，用户也可以设置会话级别的时区，设置后在该次会话中会话设置时区优先。

用户可以使用 SHOW 语句查看当前设置的时区：

SHOW TIME_ZONE

设置时区的语句实例为：

SET TIME_ZONE = +08:00

该语句将该次会话的写入和读取时区设定为+8 时区。时区的设置范围为：[-24:00, +24:00]。

详细语法参见本文第 8.5.2 节。

5.6.2 时间格式设置

IoTDB 当前支持多种时间格式。根据用户输入的时间，IoTDB 会自动解析为一个数字类型的时间戳进行处理。在用户使用 Client 工具进行数据查询的时候，如果用户没有指定查询结果中时间列的格式，则按照默认的格式(即 ISO8601)输出。如果用户指定了查询时间的显示格式，则按照用户设置的时间格式显示。

用户在 Client 工具中可以使用 SHOW 语句查看当前显示的时间格式：

SHOW TIME_DISPLAY_TYPE

设置显示的时间格式的语句实例为：

SET TIME_DISPLAY_TYPE = yyyy-MM-dd HH:mm:ss

详细语法参见本文第 8.5.1 节。显示的时间格式标准参见本文第 2.10 节。

5.6.3 强制合并文件

由于在 IoTDB 写入数据的过程中会将数据存在内存中的 overflow 和

butterwrite 两部分中，系统会定时对这两部分数据进行合并（merge）操作，定时的时间间隔会根据 iotdb-engine.properties 配置文件中的配置项 period_time_for_merge_in_second 进行设置，设置方式请参见本文第 4 章。同时，系统也为对于系统比较熟悉的用户，留有手动 merge 数据的接口。实例代码如下：

IoTDB > merge

以上语句会将 overflow 和 bufferwrite 中的数据进行合并，并提示 execute successfully。

5.6.4 强制刷新数据到磁盘

由于在 IoTDB 写入数据的过程中会先将数据放在内存中，当到达设定的 Row Group 大小或到定时的时间后，系统会对将数据写入磁盘（flush 操作）。RowGroup 大小为 tsfile-fomat.properties 文件中的 gourp_size_in_byte 项，设置方式请参见本文第 4 章。定时的时间间隔会根据 iotdb-engine.properties 配置文件中的配置项 period_time_for_flush_in_second 进行设置，设置方式请参见本文第 4 章。同时，系统也为对于系统比较熟悉的用户，留有手动 flush 文件（刷新数据到磁盘）的接口。实例代码如下：

IoTDB > flush

以上语句会将内存中的数据写入磁盘，并提示 execute successfully。

第6章 IoTDB 的 SDK 说明

6.1 JDBC

JDBC 是一套用于连接数据库的 JAVA 标准接口。为了便于开发者在 JAVA 客户端应用中使用 IoTDB，我们实现了 JDBC 标准的 java.sql 接口，为用户开发了一套针对 IoTDB 自身的 JDBC Driver，提供包括注册、连接、使用等多种 JDBC 操作，所有的接口都符合 JDBC 标准 API。

本章将从以下方面介绍 IoTDB JDBC 的使用：

- IoTDB-JDBC 版本介绍（第 6.1.1 节）
- IoTDB-JDBC 安装说明（第 6.1.2 节）
- IoTDB-JDBC 详细说明（第 6.1.3 节）
- IoTDB-JDBC 使用示例（第 6.1.4 节）
- IoTDB-JDBC 常见问题（第 6.1.5 节）

6.1.1 IoTDB-JDBC 版本介绍

IoTDB JDBC Version	JDBC Version	IoTDB Version	Status
0.0.1		0.0.1	Recommand

6.1.2 IoTDB-JDBC 安装说明

6.1.2.1 安装依赖

安装前需要保证电脑上配有 JDK>=1.8 的运行环境。

如需从源码进行编译，还需要 Maven>=3.0 的运行环境。

6.1.2.2 使用 JAR 包安装

在 IoTDB 官网上下载 IoTDB 的 JDBC 包，（官网下载页面链接：http://tsfile.org/download#jdbc）。下载成功后，将 JAR 包加入到 classpath 中即可使用。

如果想要在 pom.xml 中引用 JDBC 包，可以执行以下命令将 JAR 包安装到 mvn 仓库（其中{FilePath}为下载的 JDBC 包路径）：

```
> mvn install:install-file -DgroupId=cn.edu.tsinghua -DartifactId=iotdb-jdbc -Dversion=0.1.2
-Dpackaging=jar -Dfile={FilePath}
```

安装成功后，即可在 pom.xml 中引入 JDBC 包：

```
<dependencies>
    <dependency>
      <groupId>cn.edu.tsinghua</groupId>
      <artifactId>iotdb-jdbc</artifactId>
      <version>0.1.2</version>
    </dependency>
</dependencies>
```

6.1.2.3 使用源码编译

除从 IoTDB 官网直接下载 JAR 包的方式外，用户还可以通过克隆 IoTDB-JDBC 项目源码，然后使用 mvn 编译的方式获取 IoTDB-JDBC 的 JAR 包。

（安装 git 命令行工具后）

Shell > git clone https://github.com/thulab/iotdb-jdbc.git

（当前为未发布版本，需要获取权限后才可克隆源码）

源码克隆后，进入到源码文件夹目录下，使用以下命令进行编译：

Shell > mvn clean package -Dmaven.test.skip=true

编译过程如图 6.1。

```
$ mvn clean package -Dmaven.test.skip=true
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building IoTDB Jdbc 0.1.2
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ iotdb-jdbc ---
[INFO]
[INFO] --- maven-thrift-plugin:0.1.11:compile (thrift-sources) @ iotdb-jdbc ---
[INFO] D:\CodeRespository\iotdb-jdbc\src\main\thrift does not exist. Review the configuration or consider disabling the plugin.
[INFO]
[INFO] --- build-helper-maven-plugin:3.0.0:add-source (default) @ iotdb-jdbc ---
[INFO] Source directory: D:\CodeRespository\iotdb-jdbc\interface\thrift added.
```

图 6.1 IoTDB-JDBC 编译过程截图

当出现以下输出时，即为编译成功，如图 6.2。

```
[INFO]
[INFO] --- maven-assembly-plugin:2.5.5:single (make-assembly) @ iotdb-jdbc ---
[INFO] Building jar: D:\CodeRespository\iotdb-jdbc\target\iotdb-jdbc-0.1.2-jar-with-dependencies.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 6.601 s
[INFO] Finished at: 2017-09-26T14:17:09+08:00
[INFO] Final Memory: 37M/567M
[INFO] ------------------------------------------------------------------------
```

图 6.2 编译成功截图

编译后，需要使用 mvn 命令将其安装到 mvn 本地仓库，安装命令如下：

Shell > mvn clean install -Dmaven.test.skip=true

安装过程如图 6.3。

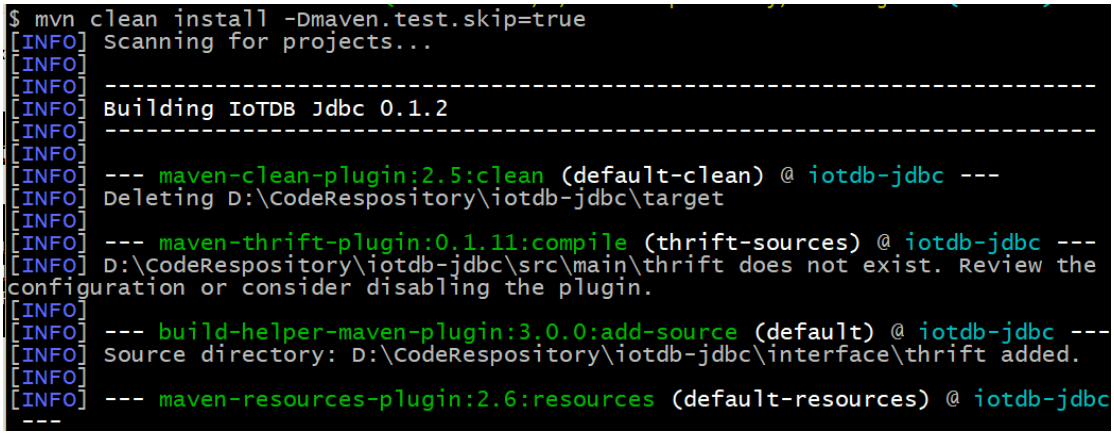

图 6.3 IoTDB-JDBC 安装过程截图

当出现以下输出时，即为编译成功，如图 6.4。

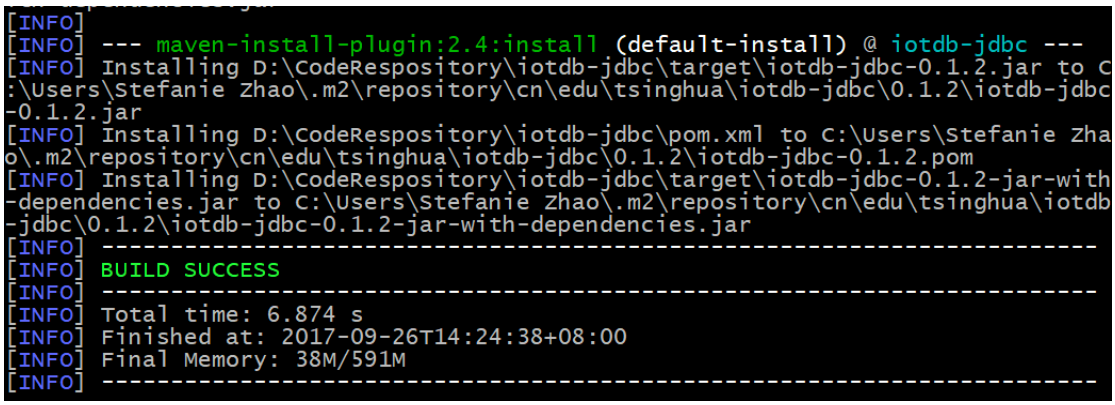

图 6.4 安装成功截图

安装成功后，即可在 pom.xml 中引入 JDBC 包：

```
<dependencies>
    <dependency>
      <groupId>cn.edu.tsinghua</groupId>
      <artifactId>iotdb-jdbc</artifactId>
      <version>0.1.2</version>
    </dependency>
</dependencies>
```

6.1.3 IoTDB-JDBC 详细说明

6.1.3.1 Java Class

```
java.sql.Connection;
java.sql.DatabaseMetaData;
java.sql.DriverManager;
java.sql.ResultSet;
java.sql.SQLException;
java.sql.Statement
```

6.1.3.2 数据类型的对应关系

本节给出在IoTDB中的数据类型与JDBC返回的JAVA数据类型的对应关系。

Java Type	IoTDB Supported Type	描述
int	INT32	整形，取值范围[-2^{31}, 2^{31}-1]
long	INT64	长整型，取值范围[-2^{63}, 2^{63}-1]
float	FLAOT	单精度浮点数，按四字节存储
double	DOUBLE	双精度浮点数，按八字节存储
boolean	BOOLEAN	布尔变量，取值为 false 或者 true
String	TEXT	字符串
enum	ENUM	枚举值（0.1.2 版本不支持）

6.1.3.3 DriverManager

```
DriverManager{
    // String url ： JDBC 要连接的地址
    // String username ： 用户名，默认为 root
    // String password ： 密码，默认为 root
    getConnection(String url, String username, String password);
}
```

其中，URL 格式为 "jdbc:tsfile://host:port/"，host 默认为 localhost，port 默认为 6667。

6.1.3.4 Connection

```
Connection {
    // 创建当前连接的 statement 用以执行 SQL 语句
    public Statement createStatement()
    // 获取数据库的元数据信息
    public DatabaseMetaData getMetaData()
    // 关闭 jdbc 客户端与数据库之间的连接
    public void close()
}
```

6.1.3.5 Statement

```
Statement {
    // 将要批量执行的 SQL 语句暂存
    public void addBatch(String sql)
    // 清除保存的要批量执行的语句
    public void clearBatch()
    // 结束本次操作
    public void close()
    // 执行输入的 SQL 语句，如果是查询操作，返回 true，否则返回 false
    public boolean execute(String sql)
    // 批量执行 SQL 语句，将每一条语句执行的结果保存在数组中
    public int[] executeBatch()
    // 执行查询操作
    public ResultSet executeQuery(String sql)
    // 执行更新操作
    public int executeUpdate(String sql)
    // 获取当前客户端和数据库的连接
    public Connection getConnection()
    // 获取每次客户端向数据库请求数据的最大条数
    public int getFetchSize()
    // 获取结果集中保存的最大结果数量
    public int getMaxRows()
    // 获得当前 SQL 语句执行后的结果集
```

```
    public ResultSet getResultSet()
    // 判断当前操作是否结束
    public boolean isClosed()
    // 设置每次客户端向数据库请求数据的最大条数
    public void setFetchSize(int fetchSize)
    // 设置结果集中保存的最大结果数量
    public void setMaxRows(int num)
    // 设置查询超时时间
    public void setQueryTimeout(int seconds)
}
```

6.1.3.6 ResultSetMetadata

```
ResultSetMetaData {
    // 获取列数
    public int getColumnCount()
    // 根据列数得到列名，并将列名转化为要打印的名字
    public String getColumnLabel(int column)
    // 根据列数返回列名
    public String getColumnName(int column)
}
```

6.1.3.7 ResultSet

```
ResultSet {
    // 结束当前操作
    public void close()
    // 根据列名称 columnName 返回它在第几列
    public int findColumn(String columnName)
    // 将字段第 columnIndex 列的数据转化为 BigDecimal 并返回
    public BigDecimal getBigDecimal(int columnIndex)
    // 将字段名称为 columnName 的数据转化为 BigDecimal 并返回
    public BigDecimal getBigDecimal(String columnName)
    // 将字段第 columnIndex 列的数据转化为 BigDecimal，设精度为 scale 并返回
    public BigDecimal getBigDecimal(int columnIndex, int scale)
    // 将字段名称为 columnName 的数据转化为 BigDecimal，设精度为 scale 并返回
```

```
public BigDecimal getBigDecimal(String columnName, int scale)
// 将字段第 columnIndex 列的数据转化为 boolean 并返回
public boolean getBoolean(int columnIndex)
// 将字段名称为 columnName 的数据转化为 boolean 并返回
public boolean getBoolean(String columnName)
// 将字段第 columnIndex 列的数据转化为 Date 并返回
public Date getDate(int columnIndex)
// 将字段名称为 columnName 的数据转化为 Date 并返回
public Date getDate(String columnName)
// 将字段第 columnIndex 列的数据转化为 double 并返回
public double getDouble(int columnIndex)
// 将字段名称为 columnName 的数据转化为 double 并返回
public double getDouble(String columnName)
// 将字段第 columnIndex 列的数据转化为 float 并返回
public float getFloat(int columnIndex)
// 将字段名称为 columnName 的数据转化为 float 并返回
public float getFloat(String columnName)
// 将字段第 columnIndex 列的数据转化为 int 并返回
public int getInt(int columnIndex)
// 将字段名称为 columnName 的数据转化为 int 并返回
public int getInt(String columnName)
// 将字段第 columnIndex 列的数据转化为 long 并返回
public long getLong(int columnIndex)
// 将字段名称为 columnName 的数据转化为 long 并返回
public long getLong(String columnName)
// 获取结果集的 metadata 信息
public ResultSetMetaData getMetaData()
// 将字段第 columnIndex 列的数据转化为 Object 并返回
public Object getObject(int columnIndex)
// 将字段名称为 columnName 的数据转化为 Object 并返回
public Object getObject(String columnName)
// 将字段第 columnIndex 列的数据转化为 short 并返回
public short getShort(int columnIndex)
```

```
    // 将字段名称为 columnName 的数据转化为 short 并返回
    public short getShort(String columnName)
    // 获得当前的 Statement
    public Statement getStatement()
    // 将字段第 columnIndex 列的数据转化为 String 并返回
    public String getString(int columnIndex)
    // 将字段名称为 columnName 的数据转化为 String 并返回
    public String getString(String columnName)
    // 将字段第 columnIndex 列的数据转化为 Time 并返回
    public Time getTime(int columnIndex)
    // 将字段名称为 columnName 的数据转化为 Time 并返回
    public Time getTime(String columnName)
    // 将字段第 columnIndex 列的数据转化为 Timestamp 并返回
    public Timestamp getTimestamp(int columnIndex)
    // 将字段名称为 columnName 的数据转化为 Timestamp 并返回
    public Timestamp getTimestamp(String columnName)
    // 判断当前 SQL 操作是否结束
    public boolean isClosed()
    // 指向下一行数据并返回 true
    // 如果所有的数据已经遍历完或者超过了该 ResultSet 所包含最大行数(由 maxRows
控制)，返回 false
    public boolean next()
}
```

6.1.3.8 DatabaseMetaData

```
DatabaseMetaData {
    // 获取表结构信息(注:在 iotdb-jdbc 中该函数与标准 JDBC 所定义的用法不同,现阶
段只能支持通过该函数获取所有的列名，或者单列的数据类型,参见本文第 6.1.4.4 节。)
    // 当 columnPattern 为"root.*"的时候，可以获得所有的列名
    // 当 columnPattern 为某一列(如"root.a.b.c")的时候，可以获得该列的数据类型
    public ResultSet getColumns(String catalog, String schemaPattern, String
columnPattern, String deltaObjectPattern)
```

```
    // 获得当前客户端与数据库的连接
    public Connection getConnection()
    // 获取数据库 metadata 信息,包含所有列的名称以及它的数据类型,编码方式等
    public String toString()
}
```

6.1.4 IoTDB-JDBC 使用示例

6.1.4.1 使用 DriverManager 建立连接

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

Connection conn = null;

try {
    //注册 IoTDB 的 JDBC 驱动
    Class.forName("cn.edu.tsinghua.iotdb.jdbc.TsfileDriver");
    //使用 DriverManager 连接 IoTDB
    connection = DriverManager.getConnection("jdbc:tsfile://localhost:6667/", "root",
"root");
    // Do something with the Connection
   ...
} catch (SQLException ex) {
    // handle any errors
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
} finally {
    if(connection != null) connection.close();
}
```

6.1.4.2 使用 IoTDB-JDBC 执行 SQL 语句

```
import java.sql.Connection;
```

```
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;

// 建立 JDBC 连接，详细操作请见 6.1.4.1 节例子

Statement statement = null;
ResultSet resultSet = null;

try {
        statement = connection.createStatement();
        statement.execute("select s1 from root.laptop.d1");
        resultSet = statement.getResultSet();
        // Do something with the ResultSet ....
}
catch (SQLException ex){
        // handle any errors
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
}
finally {
       if(statement != null) statement.close();
        if(connection != null) connection.close();
}
```

6.1.4.3 使用 IoTDB-JDBC 获取数据库 MetaData 信息

如果想看数据中的各个字段的详细信息（类似 JDBC 标准的"show metadata"），可以通过 getMetaData 的方式获取。详细操作如下：

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.DatabaseMetaData;
```

```
import java.sql.SQLException;

// 建立 JDBC 连接，详细操作请见 6.1.4.1 节例子

ResultSet resultSet = null;
try {
        DatabaseMetaData databaseMetaData = connection.getMetaData();
        String metadata = databaseMetaData.toString();
        // Do something with the metadata…
        // Note that metadata is returned as string format,
        // user can convert it to json format
}
catch (Exception e){
        // handle any errors
        e.printStackTrace();
}
finally {
        if(connection != null) connection.close();
}
```

6.1.4.4 使用 IoTDB-JDBC 获取表结构信息

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.DatabaseMetaData;

// 建立 JDBC 连接，详细操作请见 6.1.4.1 节例子

ResultSet resultSet = null;

try {
        DatabaseMetaData databaseMetaData = connection.getMetaData();
        String metadata = databaseMetaData.toString();
```

```
        // get all columns
        ResultSet resultSet = databaseMetaData.getColumns(null, null, "root.*", null);
        while(resultSet.next()){
            System.out.println(String.format("column %s", resultSet.getString(0)));

        // get data type for one column
        resultSet = databaseMetaData.getColumns(null, null, "root.a.b.c", null);
        while(resultSet.next()){
            System.out.println(String.format("column %s, type %s", resultSet.getString(0),
resultSet.getString(1)));
        }
}
catch (Exception e){
        // handle any errors
        e.printStackTrace();
}
finally {
        if(connection != null) connection.close();
}
```

6.1.5 IoTDB-JDBC 常见问题

第7章 IoTDB 相关工具使用说明

7.1 导入工具

7.2 导出工具

7.3 可视化工具（Grafana）

7.4 云端同步工具

第8章 IoTDB SQL 语言

8.1 基本定义

8.1.1 关键字

关键字		
	SQL 语句的保留关键字	具有特殊含义的关键字
IoTDB V0.1	CLOSE, COMPRESSOR, CREATE, DATATYPE, DELETE, DROP, ENCODING, FROM, GROUP, INSERT, INTO, MAX_POINT_NUMBER, MERGE, ON, QUIT, SELECT, SET, SHOW, STORAGE, TIME，ENUM_VALUES, TIMESERIES, TIMESTAMP, TO, UPDATE, VALUES, WHERE, WITH, ROOT	数据类型：BOOLEAN, DOUBLE, ENUMS, FLOAT, INT32, INT64, TEXT 编码方法：BITMAP, DFT, GORILLA, PLAIN, RLE, TS_2DIFF 压缩方法：UNCOMPRESSOR, SNAPPY 逻辑符号：AND, OR, NOT, !, TRUE, FALSE

注意：关键字是不区分大小写的，保留关键字在 SQL 语句中不能被用作值。

8.1.2 基本单元

QUOTE := '\'';
DOT := '.';
COLON : ':' ;
COMMA := ',' ;
SEMICOLON := ';' ;
LPAREN := '(' ;
RPAREN := ')' ;
EQUAL := '=' | '==';
NOTEQUAL := '<>' | '!=';
LESSTHANOREQUALTO := '<=';
LESSTHAN := '<';
GREATERTHANOREQUALTO := '>=';
GREATERTHAN := '>';
EQUAL := '=' | '==';
DIVIDE := '/';
PLUS := '+';

MINUS := '-';
STAR := '*';
Letter := 'a'..'z' | 'A'..'Z';
HexDigit := 'a'..'f' | 'A'..'F';
Digit := '0'..'9';
StringLiteral := ('\'' (~('\''))* '\'' | '\"' (~('\"'))* '\"'); eg. ‘abc’ eg. “abc”
Integer := ('-' | '+')? Digit+; eg. 123 eg. -222
Float := ('-' | '+')? Digit+ DOT Digit+ (('e' | 'E') ('-' | '+')? Digit+)?; eg. 3.1415 eg. 1.2E10
Boolean := TRUE(大小写不敏感) | FALSE(大小写不敏感) | 0 | 1
Identifier := (Letter | '_') (Letter | Digit | '_' | MINUS)*; eg. a123 eg. _abc123

8.1.3 IoTDB 基本词法

PointValue : Integer | Float | StringLiteral | Boolean
TimeValue : Integer | DateTime | ISO8601 | NOW()
DateTime : 支持的详细类型请参见本文第 2.10 节 eg. 2016-11-16T16:22:33+08:00 eg. 2016-11-16 16:22:33+08:00 eg. 2016-11-16T16:22:33.000+08:00 eg. 2016-11-16 16:22:33.000+08:00
PrecedenceEqualOperator : EQUAL | NOTEQUAL | LESSTHANOREQUALTO | LESSTHAN | GREATERTHANOREQUALTO | GREATERTHAN
Timeseries : ROOT DOT <Path> DOT <SensorName> Path : <LayerName> (DOT <LayerName>)$^{+}$ LayerName : Identifier SensorName : Identifier eg. root.laptop.d1.s1 eg. root.beijing.laptop.d1.s1 说明：Timeseries 对应为 IoTDB 中的时间序列，详情请参见本文第 2.4 节。
PrefixPath : ROOT (DOT <LayerName>)$^{+}$ LayerName : Identifier | STAR eg. root.laptop eg. root.*.s2

SuffixPath : <LayerName> (DOT <layerName>)+ layerName : Identifier | STAR eg. d0.s1 eg. *.s2

8.2 数据定义语句

query = <statement> [; <statement>]*

statement = <create_timeseries_statement>
<delete_record_statement>
<delete_timeseires_statement>
<insert_record_statement>
<select_record_statement>
<set_storage_group_statement>
<set_time_display_statement>
<set_time_zone_statement>
<update_record_statement>
<show_timeseries_statement>

8.2.1 创建时间序列

create_timeseries_statement

CREATE TIMESERIES <Timeseries> WITH <AttributeClauses>
AttributeClauses : DATATYPE=<DataTypeValue> COMMA ENCODING=<EncodingValue> [COMMA <ExtraAttributeClause>]*
DataTypeValue: BOOLEAN | DOUBLE | FLOAT | INT32 | INT64 | TEXT | ENUMS(当前版本不支持)
EncodingValue: BITMAP | DFT(当前版本不支持) | GORILLA | PLAIN | RLE | TS_2DIFF
ExtraAttributeClause: {
COMPRESSOR = <CompressorValue>
MAX_POINT_NUMBER = Integer
ENUM_VALUES=Identifier (COMMA Identifier)*(当前版本不支持)
}
CompressorValue: UNCOMPRESSOR | SNAPPY

Eg: IoTDB > CREATE TIMESERIES root.laptop.d1.s1 WITH DATATYPE=INT32, ENCODING=RLE
注：数据类型 DATATYPE 与编码类型 ENCODING 需要对应，关于数据类型与编码介绍请参见本文第 5.2.1 节与第 5.2.2 节，对应关系请参见第 5.2.3 节。

8.2.2 删除时间序列

delete_timeseries_statement
DELETE TIMESERIES <PrefixPath> [COMMA <PrefixPath>]*
Eg: IoTDB > DELETE TIMESERIES root.laptop.d1.s1 Eg: IoTDB > DELETE TIMESERIES root.laptop.d1, root.vehicle.* Eg: IoTDB > DELETE TIMESERIES root.laptop.*

8.2.3 显示时间序列

show_timeseries_statement
SHOW TIMESERIES
Eg: IoTDB > SHOW TIMESERIES

8.2.4 设置存储组

set_storage_group_statement
SET STORAGE GROUP TO <PrefixPath>
Eg: IoTDB > SET STORAGE GROUP TO root.laptop.d1

8.3 数据操作语句

8.3.1 指定时间点插入（多条）记录

insert_record_statement
INSERT INTO <PrefixPath> LPAREN TIMESTAMP COMMA <Sensor> [COMMA <Sensor>]* RPAREN VALUES LPAREN <TimeValue>, <PointValue> [COMMA <PointValue>]* RPAREN Sensor : Identifier
Eg: IoTDB > INSERT INTO root.laptop.d1(timestamp, s1) VALUES (12345, -2000) Eg: IoTDB > INSERT INTO root.laptop.d1(timestamp, s1) VALUES (NOW(), 20.1323) Eg: IoTDB > INSERT INTO root.laptop.d1(timestamp, s1) VALUES (2000-01-01T08:00:00+08:00, 2000) Eg: IoTDB > INSERT INTO root.laptop.d0(timestamp, s0, s1) VALUES (102, 80, ‘PASS’);
注：PrefixPath 与 Sensor 需要满足约束 PrefixPath + Sensor = Timeseries 注：Sensor 与 PointValue 顺序与个数需要一一对应

8.3.2 根据指定时间范围更新（多条）记录

update_record_statement
UPDATE <UpdateClause> SET VALUE = <PointValue> WHERE <WhereClause> UpdateClause: <Timeseries> WhereClause : <Condition> [(AND | OR) <Condition>]*

Condition : <Expression> [(AND | OR) <Expression>]* Expression : [NOT | !]? TIME PrecedenceEqualOperator <TimeValue>
Eg: IoTDB > UPDATE root.laptop.d0.s0 SET VALUE = 3 WHERE time < NOW() and time > 1000 Eg: IoTDB > UPDATE root.laptop.d0.s0 SET VALUE = 3 WHERE time <= 100 and time >= 1 Eg: IoTDB > UPDATE root.laptop.d0.s0, root.laptop.d0.s1 SET VALUE = 3 WHERE time >= 1
注：更新列的类型需要与 PointValue 类型

8.3.3 删除指定时间之前的（多条）记录

delete_record_statement
DELETE FROM <PrefixPath> [COMMA <PrefixPath>]* WHERE TIME LESSTHAN <TimeValue>
Eg: DELETE FROM root.laptop.d1.s1 WHERE time < 2016-11-16T16:22:33+08:00 Eg: DELETE FROM root.vehicle.d0.s0, root.vehicle.d1.s1 WHERE time < NOW(); Eg: DELETE FROM root.laptop.* WHERE time < 1234567

8.3.4 在给定条件下查询（多条）记录

select_record_statement
SELECT <SelectClause> FROM <FromClause> [WHERE <WhereClause>]? SelectClause : <SelectPath> (COMMA <SelectPath>)* SelectPath : <FUNCTION> LPAREN <SuffixPath> RPAREN | <SuffixPath> FUNCTION : 'COUNT' , 'MIN_TIME', 'MAX_TIME', 'MIN_VALUE', 'MAX_VALUE' FromClause : <PrefixPath> WhereClause : <Condition> [(AND | OR) <Condition>]* Condition : <Expression> [(AND | OR) <Expression>]* Expression : [NOT | !]? <TimeExpr> | [NOT | !]? <SensorExpr> TimeExpr : TIME PrecedenceEqualOperator <TimeValue> SensorExpr : (<Timeseries> | <SuffixPath>) PrecedenceEqualOperator <PointValue>
Eg: IoTDB > SELECT d1.s1, d2.s2 FROM root.laptop WHERE root.laptop.d1.s1 < 1000 and time > 2017-6-1 8:00:00 Eg. IoTDB > SELECT * FROM root Eg. IoTDB > SELECT COUNT(s1) FROM root.laptop.d1 WHERE root.laptop.d1.s1 < 2000 Eg. IoTDB > SELECT MIN_TIME(s1) FROM root.laptop.d1 WHERE root.laptop.device_1.s1< 2000 Eg. IoTDB > SELECT MAX_TIME(s1) FROM root.laptop.d1 WHERE root.laptop.device_1.s1< 2000 Eg. IoTDB > SELECT MIN_TIME(s1) FROM root.laptop.d1 WHERE root.laptop.device_1.s1< 2000

Eg. IoTDB > SELECT MAX_VALUE(s1) FROM root.laptop.d1 WHERE root.laptop.device_1.s1< 2000
注 1：SelectClause 中的 SuffixPath 和 FromClause 中的 PrefixPath 需要满足 PrefixPath+SuffixPath=Timeseries 约束； 注 2：WhereClause 中 SensorExpr 如果是以 SuffixPath 开头，需要和 FromClause 中的 PrefixPath 满足 PrefixPath+SuffixPath=Timeseries 约束

8.4 数据库管理语句

8.4.1 账户管理语句

8.4.2 修改用户

8.4.3 创建用户

8.4.4 删除用户

8.4.5 创建角色

8.4.6 删除角色

8.4.7 授权用户

8.4.8 授权角色

8.4.9 撤销用户

8.4.10 撤销角色

8.4.11 授权角色给用户

8.4.12 撤销用户的角色

8.4.13 更新密码

8.5 SET 语句

8.5.1 设置时间格式

set_time_display_statement
SET TIME_DISPLAY_TYPE=<TimeValue>

Eg: IoTDB > SET TIME_DISPLAY_TYPE=ISO8601 Eg: IoTDB > SET TIME_DISPLAY_TYPE=DEFAULT（注：DEFAULT 格式与 ISO8601 格式相同） Eg: IoTDB > SET TIME_DISPLAY_TYPE=LONG Eg: IoTDB > SET TIME_DISPLAY_TYPE=NUMBER Eg: IoTDB > SET TIME_DISPLAY_TYPE=yyyy-MM-dd KK:mm:ss
注：TIME_DISPLAY_TYPE 设置方法参见本文第 2.10 节

8.5.2 设置时区

set_time_zone_statement
SET TIME_ZONE=<TimeZone> TimeZone : [PLUS | MINUS]? Digit Digit COLON Digit Digit
Eg: IoTDB > SET TIME_ZONE=+08:00 Eg: IoTDB > SET TIME_ ZONE=-01:00

8.6 函数

8.6.1 AVG 函数（待添加）

8.6.2 COUNT 函数

COUNT 函数返回被 SELECT 语句选取的 timeseries（一个或多个）非空值的个数。返回结果是带符号的 64 位整型数。如果没有匹配的行，COUNT()返回 0。

SELECT COUNT(SuffixPath) FROM <FromClause> [WHERE <WhereClause>]?
Eg. SELECT COUNT(s1) FROM root.laptop.d1 WHERE root.laptop.d1.s1 < 2000
注 1: 关于< FromClause >和< WhereClause >的使用，见本文第 8.3.4 节 注 2：SuffixPath 和 FromClause 中的 PrefixPath 需要满足 PrefixPath+SuffixPath=Timeseries 约束； 注 3：当前版本不支持带 Where 过滤条件的聚合操作

8.6.3 MAX_TIME 函数

MAX_TIME 函数返回该列标识符最大时间戳。返回结果是带符号的 64 位整型数，且大于 0。

SELECT MAX_TIME (SuffixPath) FROM <FromClause> [WHERE <WhereClause>]?
Eg. SELECT MAX_TIME(s1) FROM root.laptop.d1 WHERE root.laptop.d1.s1< 2000
注 1: 关于< FromClause >和< WhereClause >的使用，见本文第 8.3.4 节 注 2：SuffixPath 和 FromClause 中的 PrefixPath 需要满足 PrefixPath+SuffixPath=Timeseries 约束； 注 3：当前版本不支持带 Where 过滤条件的聚合操作

8.6.4 MAX_VALUE 函数

MAX_VALUE 函数返回的是该列按字典序排序的最大值。

SELECT MAX_VALUE (SuffixPath) FROM <FromClause> [WHERE <WhereClause>]?
Eg.SELECT MAX_VALUE(s1) FROM root.laptop.d1 WHERE root.laptop.d1.s1< 200
注 1: 关于< FromClause >和< WhereClause >的使用，见本文第 8.3.4 节 注 2：SuffixPath 和 FromClause 中的 PrefixPath 需要满足 PrefixPath+SuffixPath=Timeseries 约束； 注 3：当前版本不支持带 Where 过滤条件的聚合操作

8.6.5 MIN_TIME 函数

MIN_TIME 函数返回该列标识符最小时间戳。返回结果是带符号的 64 位整型数，且大于 0。

SELECT MIN_TIME (SuffixPath) FROM <FromClause> [WHERE <WhereClause>]?
Eg. SELECT MIN_TIME(s1) FROM root.laptop.d1 WHERE root.laptop.d1.s1< 2000
注 1: 关于< FromClause >和< WhereClause >的使用，见本文第 8.3.4 节 注 2：SuffixPath 和 FromClause 中的 PrefixPath 需要满足 PrefixPath+SuffixPath=Timeseries 约束； 注 3：当前版本不支持带 Where 过滤条件的聚合操作

8.6.6 MIN_VALUE 函数

MIN_VALUE 函数返回的是该列按字典序排序的最小值。

SELECT MIN_VALUE (SuffixPath) FROM <FromClause> [WHERE <WhereClause>]?
Eg. SELECT MIN_TIME(s1) FROM root.laptop.d1 WHERE root.laptop.d1.s1< 2000
注 1: 关于< FromClause >和< WhereClause >的使用，见本文第 8.3.4 节 注 2：SuffixPath 和 FromClause 中的 PrefixPath 需要满足 PrefixPath+SuffixPath=Timeseries 约束； 注 3：当前版本不支持带 Where 过滤条件的聚合操作

8.6.7 NOW 函数

NOW 函数返回带符号的 64 位整型数的当前时间戳，它可以被用于本文第 8.3 节的数据操作语句中。

NOW()
Eg. INSERT INTO root.laptop.d0 (timestamp, s0, s1) VALUES(NOW() , -1011.666, 'On') Eg. UPDATE root. laptop.d0.s0 SET VALUE = -33000 WHERE time = NOW() Eg. DELETE FROM root.laptop.d0.s0 WHERE time < NOW() Eg. SELECT * FROM root.laptop WHERE time < NOW() Eg. SELECT COUNT(s0) FROM root.laptop.d0 WHERE root.laptop.d0.s0 < 2000 OR time

<= NOW()