第14张슬라이드：8.5 容错处理

好，现在我们进入一个新的重要主题——容错处理(Fault Tolerance Processing)。

今天我们会学习两个方面：

- 从检查点恢复故障
- 故障恢复中的限制

在实际的生产环境中，容错处理是非常重要的，因为系统可能会因为各种原因出现故障。

第15张슬라이드：8.5.1 从检查点恢复故障

大家想想，Spark程序一般要长时间运行，对吧？可能运行几天、几周甚至几个月。在这么长的时间里，难免会遇到系统故障或者JVM故障，导致程序退出或系统重启。

所以，Spark程序必须具备恢复能力！

正确配置容错环境包括几个方面：

- 选择容错的输入源
- 记录输入源位置偏移量
- 保存检查点和预写日志(Write-Ahead Log)中间状态
- 使用容错的接收器

好消息是，记录输入源位置偏移量、检查点、预写日志这些都由Spark引擎自动完成。我们作为使用者，只需要提供检查点路径，Spark引擎就会保存恢复所需的所有数据。

还记得8.4.2节中的例子吗？我们在文件接收器内设置了checkpointLocation选项。如果发生故障，由于检查点和预写日志保存了位置偏移量等信息，Spark程序就可以通过这些信息恢复到之前查询的进度和状态。

第16张슬라이드：8.5.2 故障恢复中的限制

虽然Spark设计了分离的输入源和接收器，但是在程序停止运行或者故障发生后，有时候为了解决故障问题，我们可能需要修改部分程序代码并重启查询。

但是，这里有个重要的限制：如果这时仍使用旧的检查点数据来恢复程序运行，那么修改部分代码的做法是不被允许的！或者即使允许，也会导致运行结果可能有无法预期的错误。

这就是为什么我们需要了解故障恢复中的限制。

第17张슬라이드：8.5.2 故障恢复中的限制（具体规则）

现在我给大家详细介绍一下参数修改的规则，这些规则很重要，大家要记住：

输入源的类型和数量的更改：不被允许。比如你原来从Kafka读数据，就不能改成从文件读数据。

输入源参数的更改：部分允许。不会影响检查点状态的参数可以修改，比如Kafka的maxOffsetsPerTrigger等限速参数；但是修改Kafka的主题或者文件路径就不被允许。

接收器类型的更改：有条件允许。File接收器改为Kafka接收器是允许的，但是Kafka只能接收到新的数据；Kafka接收器改为File接收器则不被允许；Kafka接收器和Foreach接收器可以互相替换。

接收器参数的更改：更改File接收器的路径不被允许；Kafka接收器的输出主题允许被更改；Foreach接收器自定义的函数可以被更改。

projection、filter、map等操作的更改：部分允许，比如增加或者删除过滤条件。

有状态操作的更改：不被允许。有些流计算查询操作需要保存状态数据到检查点，这种操作的更改是不被允许的。

课程总结

好，今天的课程就到这里。我们学习了Spark Streaming的输出操作，包括：

1. **启动流计算**：通过DataStreamWriter接口和它的各种配置选项
2. **输出模式**：Append、Complete、Update三种模式的特点和适用场景
3. **输出接收器**：各种内置接收器的特性和使用方法
4. **容错处理**：检查点机制和故障恢复的限制

这些内容在实际项目中都会用到，建议大家课后多练习，特别是File接收器的例子，可以自己动手试试。

下次课我们会继续学习更高级的Spark Streaming主题。有什么问题可以随时问我！

Structured Streaming 中文讲课脚本

第1张슬라이드：8.6 迟到数据处理

同学们好！今天我们要学习的是 Spark Structured Streaming (结构化流处理) 中一个非常重要的概念——迟到数据处理 (Late Data Processing)。

大家请看这张**슬라이드**，我们今天的内容主要分为五个部分：

- 首先我们会讲事件时间 (Event Time) 的概念
- 然后理解什么是迟到数据 (Late Data)
- 接着学习水印 (Watermark) 机制
- 再了解多水印规则 (Multiple Watermark Rules)
- 最后通过一个完整的实例来看迟到数据是如何处理的

这些概念在实时流处理 (Real-time Stream Processing) 中非常关键，因为在真实的分布式系统 (Distributed System) 中，数据不可能总是按时到达。

第2张슬라이드：8.6.1 事件时间

好，我们先来看第一个概念——事件时间（Event Time）。

假设我们有一个数据流（Data Stream），每条数据都包含了数据生成的时间戳（Timestamp）。我们的任务是计算每10分钟内接收到的单词个数，并且每5分钟更新一次结果。

大家看这个图，我来给大家详细解释一下：

上面是输入数据流（Input Data Stream），你们可以看到：

- 在12:02时刻，我们收到了"cat dog"
- 在12:03时刻，收到了"dog dog"
- 在12:07时刻，收到了"owl cat"
- 在12:11时刻，收到了"dog"
- 在12:13时刻，收到了"owl"

下面的时间轴显示的是处理时间（Processing Time），从12:00开始，每5分钟触发一次。

关键在于我们使用的是滑动窗口（Sliding Window）：

- 窗口大小（Window Size）是10分钟
- 滑动间隔（Sliding Interval）是5分钟

所以你们可以看到，每次触发时，我们都会计算多个重叠的时间窗口内的词频统计。这就是事件时间处理的基本概念。

第3张슬라이드：8.6.2 迟到数据

现在我们来看一个更复杂的情况——如果数据迟到了怎么办？

在现实的分布式系统中，由于网络延迟（Network Latency）、系统故障（System Failure）等各种原因，数据可能不会按照事件发生的时间顺序到达。

大家看这个图，我来解释一下发生了什么：

正常情况下：

- 前两条数据在12:00-12:05时间窗口内到达
- 一条数据在12:05-12:10时间窗口内到达
- 两条数据在12:10-12:15时间窗口内到达

但是注意这里有一个问题！有一条数据，它的事件时间（Event Time）是12:04，但是直到12:11才到达系统。这就是我们说的迟到数据（Late Data）。

你们可以看到，由于这条迟到数据的到达，12:00-12:10时间窗口内的dog的计数从3变成了4。这说明迟到数据确实影响了我们的计算结果。

这就引出了一个重要问题：我们应该如何处理这些迟到的数据？

第4张슬라이드：8.6.3 水印（代码部分）

为了解决迟到数据的问题，Spark Structured Streaming 引入了水印 (Watermark) 机制。

大家看这段代码，这是定义水印的标准方法：

```
words = ... # DataFrame流的结构为 { timestamp: Timestamp, word: String }


# 对窗口和字母进行聚合并计算计数
windowedCounts = words ₩
    .withWatermark("timestamp", "10 minutes") ₩
    .groupBy(
        window(words.timestamp, "10 minutes", "5 minutes"),
        words.word) ₩
    .count()
```

关键在于这一行：.withWatermark("timestamp", "10 minutes")

这里我们告诉 Spark：

- 在 timestamp 字段上设置水印
- 允许数据最多迟到10分钟

这意味着，如果一个事件的时间比当前水印时间早超过10分钟，那么这个事件就会被丢弃。

第5张슬라이드：8.6.3 水印（图解）

现在我们通过这个图来理解水印是如何工作的。

这个图很重要，我来逐个解释这些符号：

- 黑色实心圆点：正常到达的数据
- 灰色圆点：成为最大事件时间的数据
- 三角形：迟到到达但在水印范围内的数据
- 空心圆：迟到并且超出水印范围的数据

实线表示水印（Watermark），虚线表示到目前为止能看到的最大事件时间。

水印的计算公式是：当前水印 = 最大事件时间 - 允许迟到时间

比如说，如果我们看到的最大事件时间是12:21，允许迟到10分钟，那么当前水印就是12:11。

你们可以看到，处理时间每5分钟触发一次，从12:05到12:25。每次触发时，Spark 会：

1. 更新水印
2. 丢弃超出水印范围的迟到数据
3. 处理在水印范围内的所有数据
4. 输出更新的结果

这样既能处理一定程度的迟到数据，又能保证系统性能不会因为无限期等待而下降。

第6张슬라이드：8.6.4 多水印规则

在实际应用中，我们经常需要处理来自多个数据源的流。这就涉及到多水印规则（Multiple Watermark Rules）。

让我来解释一下这个复杂的场景：

当一个查询涉及多个输入源的联合（Union）或连接（Join）时：

1. 独**立水印**：每个输入源都可以定义自己独立的水印来跟踪中间状态。单个输入源定义的水印不会影响其他输入源。
2. 独**立跟踪**：Structured Streaming 会独立跟踪每个数据流中的事件时间，并分别计算延迟。
3. **全局水印选择**：如果联合或连接查询存在有状态的操作（Stateful Operations），Structured Streaming 会选择一个全局水印来跟踪所有流计算的中间状态。
4. **保守策略**：默认情况下，为了保证所有流的迟到数据都会被处理，Structured Streaming 会选择最长的迟到水印时间。

举个例子：如果流A允许数据迟到10分钟，流B允许数据迟到20分钟，那么全局水印就会允许保留20分钟的中间状态。

这样的设计确保了即使在多流场景下，我们也不会丢失重要的迟到数据。

第7张슬라이드：8.6.5 处理迟到数据的实例（导入模块）

好，现在我们通过一个完整的实例来看如何处理迟到数据。这是一个 Python 程序，我们先看导入的模块：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# 导入需要用到的模块
import os
import shutil
from functools import partial

from pyspark.sql import SparkSession
from pyspark.sql.functions import window
from pyspark.sql.types import StructType, StructField
from pyspark.sql.types import TimestampType, StringType
```

这里我们导入了：

- os 和 shutil：用于文件系统操作
- SparkSession：创建 Spark 会话
- window 函数：用于定义时间窗口
- 数据类型定义：StructType, StructField, TimestampType, StringType

这些都是构建流处理应用程序的基础组件。

第8张슬라이드：8.6.5 处理迟到数据的实例（环境设置）

接下来我们看环境设置的代码：

```
# 定义CSV文件的路径常量
TEST_DATA_DIR = '/tmp/testdata/'
TEST_DATA_DIR_SPARK = 'file:///tmp/testdata/'

# 搭建测试环境
def test_setUp():
    if os.path.exists(TEST_DATA_DIR):
        shutil.rmtree(TEST_DATA_DIR, ignore_errors=True)
    os.mkdir(TEST_DATA_DIR)

# 恢复测试环境
def test_tearDown():
    if os.path.exists(TEST_DATA_DIR):
        shutil.rmtree(TEST_DATA_DIR, ignore_errors=True)
```

这些函数的作用是：

- test_setUp()：清理并创建测试目录
- test_tearDown()：清理测试环境

注意这里有两个路径：一个是本地文件系统路径，一个是 Spark 使用的 file:// 协议路径。这在分布式环境中是很常见的做法。

第9张슬라이드：8.6.5 处理迟到数据的实例（文件写入函数）

这里我们定义了一个模拟数据输入的函数：

```
def write_to_csv(filename, data):
    with open(TEST_DATA_DIR + filename, "wt", encoding="utf-8") as f:
        f.write(data)
```

这个函数很简单但很重要。在实际的流处理系统中，我们需要模拟数据的实时到达。

注释中提到了一个重要概念：**原子性写入**（Atomic Write）。在生产环境中，如果写入时间较长，我们应该：

1. 先写入临时文件
2. 写入完成后再移动到目标目录

这样可以避免 Spark 读取到不完整的文件。

第10张슬라이드：8.6.5 处理迟到数据的实例（主函数开始）

现在我们进入主函数：

```
if __name__ == "__main__":
    test_setUp()

    # 定义模式
    schema = StructType([
        StructField("word", StringType(), True),
        StructField("eventTime", TimestampType(), True)])

    spark = SparkSession ₩
        .builder ₩
        .appName("StructuredNetworkWordCountWindowedDelay") ₩
        .getOrCreate()

    spark.sparkContext.setLogLevel('WARN')
```

这里我们：

1. 调用 test_setUp() 初始化环境
2. 定义了数据的 Schema：包含 word（字符串）和 eventTime（时间戳）两个字段
3. 创建 SparkSession
4. 设置日志级别为 WARN，这样可以减少控制台输出

Schema 定义很重要，因为 Structured Streaming 需要知道数据的结构才能正确处理。

第11张슬라이드：8.6.5 处理迟到数据的实例（流定义）

现在我们定义数据流和窗口操作：

```
lines = spark ₩
    .readStream ₩
    .format('csv') ₩
    .schema(schema) ₩
    .option("sep", ";") ₩
    .option("header", "false") ₩
    .load(TEST_DATA_DIR_SPARK)

# 定义窗口
windowDuration = '1 hour'

windowedCounts = lines ₩
    .withWatermark("eventTime", "1 hour") ₩
    .groupBy('word', window('eventTime', windowDuration)) ₩
    .count()
```

这里发生了几个关键步骤：

1. **创建流数据源**：从 CSV 文件读取数据，使用分号作为分隔符
2. **设置水印**：允许数据最多迟到1小时
3. **定义窗口操作**：按照1小时的窗口对单词进行计数

注意这里的水印设置：withWatermark("eventTime", "1 hour")，这意味着超过1小时的迟到数据将被丢弃。

第12张슬라이드：8.6.5 处理迟到数据的实例（查询启动）

现在我们启动流查询并写入第一个测试文件：

```
query = windowedCounts ₩
    .writeStream ₩
    .outputMode("update") ₩
    .format("console") ₩
    .option('truncate', 'false') ₩
    .trigger(processingTime="8 seconds") ₩
    .start()


# 写入测试文件file1.csv
write_to_csv('file1.csv', """
正常;2023-10-01 08:00:00
正常;2023-10-01 08:10:00
正常;2023-10-01 08:20:00
""")
```

这里我们：

1. 启动流查询，使用 update 输出模式
2. 每8秒触发一次处理
3. 写入第一批测试数据

第一批数据都是"正常"到达的，时间分别是08:00, 08:10, 08:20。

第13张슬라이드：8.6.5 处理迟到数据的实例（第一次处理）

接下来我们处理第一批数据并添加第二批：

```
# 处理当前数据

query.processAllAvailable()


# 此时事件时间更新到上次看到的最大的2023-10-01 08:20:00


write_to_csv('file2.csv', """

正常;2023-10-01 20:00:00

一小时以内延迟到达;2023-10-01 10:00:00

一小时以内延迟到达;2023-10-01 10:50:00

""")
```

这里很有意思：

1. 第一条数据时间是20:00:00，这会更新我们的最大事件时间
2. 后面两条数据的事件时间是10:00:00和10:50:00，虽然迟到了，但是在1小时的水印范围内
3. 当前水印 = 20:00:00 - 1小时 = 19:00:00，所以这些迟到数据仍然会被处理

第14张슬라이드：8.6.5 处理迟到数据的实例（第二次处理）

继续处理并添加第三批数据：

```
# 处理当前数据
query.processAllAvailable()


# 此时事件时间更新到上次看到的最大的2023-10-01 20:00:00


write_to_csv('file3.csv', """
正常;2023-10-01 20:00:00
一小时以外延迟到达;2023-10-01 10:00:00
一小时以外延迟到达;2023-10-01 10:50:00
一小时以内延迟到达;2023-10-01 19:00:00
""")
```

第三批数据更加有趣：

1. 时间为20:00:00的数据是正常的
2. 时间为10:00:00和10:50:00的数据超出了水印范围（早于19:00:00），会被丢弃
3. 时间为19:00:00的数据恰好在水印边界上，会被处理

这完美展示了水印机制的工作原理！

第15张슬라이드：8.6.5 处理迟到数据的实例（完成）

最后我们完成处理并清理环境：

```
# 处理当前数据
query.processAllAvailable()


query.stop()
test_tearDown()
```

这个完整的例子展示了：

1. 如何设置水印
2. 在水印范围内的迟到数据如何被正确处理
3. 超出水印范围的数据如何被丢弃
4. 流查询的完整生命周期

第16张슬라이드：8.7 查询的管理和监控

好，我们进入下一个重要主题：查询的管理和监控（Query Management and Monitoring）。

在生产环境中，我们不仅要正确地处理数据，还要能够：

- 监控查询的运行状态
- 管理多个并发查询
- 获取性能指标
- 处理异常情况

这一节我们会学习两个内容：

1. 管理和监控的方法
2. 一个具体的监控实例

第17张슬라이드：8.7.1 管理和监控的方法

当我们启动一个流查询时，会返回一个 StreamingQuery 对象。这个对象是我们管理和监控查询的核心工具。

它包含几个重要的属性：

1. **recentProgress**：返回最近多个 StreamingQueryProgress 构成的数组。数组大小可以通过 spark.sql.streaming.numRecentProgressUpdates 配置。
2. **lastProgress**：返回最近的一个 StreamingQueryProgress，包含了查询的详细执行信息。
3. **status**：返回当前查询的状态信息。

这些信息对于理解查询的性能和健康状况非常重要。我们可以通过这些信息来：

- 监控数据处理速度
- 检测性能瓶颈
- 调试问题

第18张슬라이드：8.7.1 管理和监控的方法（续）

在更复杂的场景中，我们可能需要在一个 SparkSession 中运行多个查询。这时候需要用到 StreamingQueryManager。

可以通过 spark.streams 获取 StreamingQueryManager 对象，它提供了几个重要的方法：

1. **awaitAnyTermination()**：等待任意一个查询结束后停止阻塞
2. **active**：返回当前上下文中所有还在活动的查询
3. **resetTerminated()**：忽略已经结束的查询，使 awaitAnyTermination() 可以继续阻塞程序

为什么需要这些方法？因为如果我们启动了多个查询，不能使用单个 query 的 awaitTermination()，那样会导致一个查询结束后程序就退出，其他查询无法继续。

正确的做法是使用循环：

```
while spark.streams.active:
    spark.streams.awaitAnyTermination()
    spark.streams.resetTerminated()
```

第19张슬라이드：8.7.2 一个监控的实例（导入）

现在我们通过一个实际例子来看如何监控流查询。首先看导入的模块：

```
#!/usr/bin/env python3

from pprint import pprint
import time

from pyspark.sql import SparkSession
from pyspark.sql.functions import split
from pyspark.sql.functions import explode
```

这里我们导入了：

- pprint：用于漂亮地打印字典和列表
- time：用于控制监控的时间间隔
- Spark SQL 的相关函数

第20张슬라이드：8.7.2 一个监控的实例（SparkSession设置）

创建 SparkSession 并设置数据源：

```
if __name__ == "__main__":
    spark = SparkSession ₩
        .builder ₩
        .appName("StructuredNetworkWordCount") ₩
        .getOrCreate()

    spark.sparkContext.setLogLevel('WARN')

    lines = spark ₩
        .readStream ₩
        .format("socket") ₩
        .option("host", "localhost") ₩
        .option("port", 9999) ₩
        .load()
```

这里我们使用了 socket 数据源，这是一个很方便的测试方式：

- 监听 localhost 的 9999 端口
- 可以通过 nc -lk 9999 命令模拟数据输入

第21张슬라이드：8.7.2 一个监控的实例（数据处理）

定义数据处理逻辑和查询：

```
words = lines.select(
    explode(
        split(lines.value, " ")
    ).alias("word")
)

wordCounts = words.groupBy("word").count()

query = wordCounts ₩
    .writeStream ₩
    .outputMode("complete") ₩
    .format("console") ₩
    .queryName('write_to_console') ₩
    .trigger(processingTime="8 seconds") ₩
    .start()
```

这里的处理逻辑是经典的词频统计：

1. 按空格分割每行文本
2. 使用 explode 将数组展开为单个单词
3. 按单词分组计数
4. 每8秒触发一次输出

注意 queryName('write_to_console')，这给查询指定了一个名称，便于监控。

第22张슬라이드：8.7.2 一个监控的实例（监控循环）

最核心的监控代码：

```
while True:
    if query.lastProgress:
        if query.lastProgress['numInputRows'] > 0:
            pprint(query.lastProgress)
            pprint(query.status)
    time.sleep(5)
```

这个循环会：

1. 检查是否有最新的进度信息
2. 如果有输入数据（numInputRows > 0），就打印详细信息
3. 每5秒检查一次

这样我们就能实时看到查询的执行情况，包括处理速度、延迟等关键指标。

第23张슬라이드：8.7.2 一个监控的实例（运行方法）

要运行这个监控例子，我们需要准备环境：

首先启动 HDFS：

$ cd /usr/local/hadoop

$ sbin/start-dfs.sh

然后开三个终端：

数据源终端：

$ nc -lk 9999

流计算终端：

$ cd /usr/local/spark/mycode/structuredstreaming/monitor

$ /usr/local/spark/bin/spark-submit StructuredNetworkWordCountWithMonitor.py

这样就建立了一个完整的测试环境。nc 命令会启动一个简单的服务器，我们可以在其中输入文本数据。

第24张슬라이드：8.7.2 一个监控的实例（测试数据）

在数据源终端中，我们可以输入测试数据：

apache spark

apache hadoop

当我们输入这些数据时，nc 程序会将数据发送给 Spark 程序进行处理。这模拟了真实的流数据场景。

我们可以：

1. 连续输入不同的文本行
2. 观察词频统计的实时变化
3. 查看监控信息的更新

这种方式非常适合学习和测试 Structured Streaming 的功能。

第25张슬라이드：8.7.2 一个监控的实例（输出结果）

运行程序后，我们会在流计算终端看到丰富的监控信息：

{'batchId': 0,

'durationMs': {'addBatch': 4107,

'getBatch': 219,

'getOffset': 0,

'queryPlanning': 345,

'triggerExecution': 4736,

'walCommit': 55},

'id': '38846056-5b0a-4c06-894e-fa038a72617f',

'inputRowsPerSecond': 0.25,

'name': 'write_to_console',

'numInputRows': 2,

'processedRowsPerSecond': 0.4222972972972973,

'runId': '57284b41-0800-48c9-9ae8-3b1ecaaa492c',

这些信息告诉我们：

- **batchId**：批次ID
- **durationMs**：各个阶段的耗时（毫秒）
- **inputRowsPerSecond**：输入数据速率
- **processedRowsPerSecond**：处理数据速率
- **numInputRows**：输入行数

通过这些指标，我们可以评估查询的性能和健康状况。

第26张슬라이드：8.8 本章小结

好，同学们，我们今天的课程就要结束了。让我来总结一下本章的主要内容：

1. **Structured Streaming 基础**：我们学习了 Structured Streaming 的基本概念，以及它与 Spark SQL 和 Spark Streaming 的区别。
2. **事件时间处理**：理解了事件时间和处理时间的区别，学会了如何使用时间窗口进行流数据处理。
3. **迟到数据处理**：深入学习了水印机制，知道如何平衡数据完整性和系统性能。
4. **多流处理**：了解了多水印规则，学会处理复杂的多源数据场景。
5. **实际应用**：通过完整的代码示例，掌握了从设计到实现的全过程。
6. **监控和管理**：学会了如何监控流查询的运行状态，这对生产环境非常重要。

Spark Structured Streaming 是一个非常强大的流处理框架，它提供了端到端的精确一次语义保证 (Exactly-Once Semantics)，这在实时数据处理中是非常重要的特性。

希望大家课后能够动手实践这些例子，深入理解流处理的核心概念。下节课我们将继续学习更高级的 Spark 应用。

谢谢大家！