8.3 输入源

SL-01 슬라이드 1: 课程目录

今天我们来学习Spark Structured Streaming（结构化流处理）的输入源（Input Sources）部分。请大家看屏幕，我们今天主要会学习四种不同的输入源。

首先是File源（File Source），这是我们今天重点要讲的内容；然后是Kafka源（Kafka Source），这个在大数据实时处理中非常重要；接下来是Socket源（Socket Source），主要用于简单的网络数据接收；最后是Rate源（Rate Source），这个主要用于测试和学习。

- **屏幕 → píng mù**
- 工作重点 (gōng zuò zhòng diǎn)

SL-02슬라이드 2: File源概述

好，现在我们开始讲File源。

大家注意看，File源，或者我们也可以叫它"文件源"，它的作用是什么呢？

它是以文件流（File Stream）的形式来读取某个目录中的文件。

支持的文件格式很多，比如说 csv、json、orc(Optimized Row Columnar) 、parquet、text 等等，

这些都是我们在大数据处理中经常遇到的格式。 **yù dào**

这里有一个非常重要的注意事项，大家一定要记住！ **zhùyì shìxiàng, jìzhù.**

文件放置到给定目录的操作应当是原子性的（Atomic,automicity）。

什么意思呢？就是说你不能长时间在给定目录内打开文件写入内容，而是应该采取大部分操作系统都支持的方式，就是先写入到临时文件，然后移动文件到给定目录。

为什么要这样做呢？

因为如果你直接在目标目录里写文件，

Spark可能会读到一个还没写完的文件，这就会出问题。

SL-03 슬라이드 3: File源选项详解（1/4)

接下来我们看看File源的选项(Options）。

这些选项很重要，因为它们决定了File源如何工作。

第一个是path选项，这个是输入路径的目录，所有文件格式都通用。path支持glob通配符路径，什么是glob呢？比如说你可以用星号（*）来匹配多个文件。但是要注意，目录或glob通配符路径的格式不支持以多个逗号分隔的形式。(*，?，[a,b,c])

第二个是maxFilesPerTrigger，这个选项控制每个触发器（Trigger）中要处理的最大新文件数。默认是没有最大值的，也就是说会处理所有的新文件。

第三个是latestFirst，这个问的是否优先处理最新的文件。当有大量文件积压时，设置为True可以优先处理新文件，默认是False。这个选项在实际应用中很有用，特别是当你的数据处理跟不上数据产生速度的时候。

SL-04 슬라이드 4: File源选项详解（2/4)

现在我们看第四个选项，fileNameOnly。

这个选项问的是，是否仅根据文件名而不是完整路径来检查新文件，默认为False。

如果设置为True，那么下面这些文件将被视为（shìwéi）相同的文件，因为它们的文件名"dataset.txt"相同。大家看屏幕上的例子：

- "file:///dataset.txt"
- "s3://a/dataset.txt"
- "s3n://a/b/dataset.txt"
- "s3a://a/b/c/dataset.txt"

虽然它们在不同的路径，甚至在不同的存储系统上，但是因为文件名相同，如果fileNameOnly设置为True，Spark就会认为它们是同一个文件。

这个选项什么时候有用呢？比如说你有一个文件从本地文件系统移动到了S3，如果你不想重复处理，就可以设置这个选项为True。(不覆盖现有文件，防止对同一文件)

SL-05 슬라이드 5: File源选项详解（3/4)

第五个选项是maxFileAge，这个是目录内文件的最大生存时间。

如果文件的生存时间超过这个值，则忽略这个文件。

注意几个要点：首次批处理运行时，所有文件都被认为是有效的。

如果latestFirst被设置为True，则这个参数会被认为无效，为什么呢？因为如果这个参数有效，有可能会忽略有效的旧文件。

还有一个重要的点，这个生存时间设置是基于最新的文件的，

而不是当前的系统时间，默认值为一星期。

这个选项的作用是什么呢？

它可以帮助你避免处理太旧的文件，比如说你只想处理最近一周的数据。

SL-06 슬라이드 6: File源选项详解（4/4)

最后一个重要的选项是cleanSource，这个是清理已完成文件的方法。

有三个选项：归档（archive）、删除（delete）和关闭（off）。

如果未提供这个选项，默认值为关闭。

如果选择了归档，则必须再提供sourceArchiveDir选项。

这里有一个很重要的注意事项：为了防止(fángzhǐ)归档后的文件又被PATH通配符匹配成功导致文件被重复处理，sourceArchiveDir的值必须在PATH通配符外面。

我给大家举个例子，比如使用"/hello?/spark/*"作为PATH通配符，则"/hello1/spark/archive/dir"和"/hello1/spark"都不能作为sourceArchiveDir的值，而"/helloarchive/spark"可以。

Spark在归档时会保留文件的原始路径，比如原始文件目录为"/a/b/dataset.txt"，归档目录为"/helloarchive/spark"，则原始文件被归档后的路径为"/helloarchive/spark/a/b/dataset.txt"。

归档和删除操作既有好处也有坏处。好处是会减少列出待处理的文件的系统开销，坏处是移动和删除均(都)会带来一定的系统开销。

所以，必须仔细评估这个操作对系统的影响再决定是否开启。

还有两个重要的注意事项：

第一，移动和删除可能失败，虽然失败但是不会影响查询；

第二，因为存在移动和删除，所以不要在多个不同的查询中使用，否则多次未预期的移动和删除会导致数据混乱。

SL-07 슬라이드 7: 特定文件格式选项

除了刚才讲的通用选项之外，特定的文件格式也有一些其他特定的选项，

具体可以参阅Spark手册内DataStreamReader中的相关说明。

https://spark.apache.org/docs/latest/streaming/apis-on-dataframes-and-datasets.html#creating-streaming-dataframes-and-streaming-datasets

我给大家举个例子，以.csv文件源为例，看看这段代码：

```
csvDF = spark \
    .readStream \
    .format("csv") \
    .option("seq", ";") \
    .load("SOME_DIR")
```

这里，seq选项指定了.csv的间隔符号。默认情况下，csv文件用逗号分隔，但是如果你的文件用分号或者其他符号分隔，就需要通过这个选项来指定。

SL-08 슬라이드 8: 实际例子介绍

好，理论讲完了，现在我们来看一个实际的例子。

我们要通过一个JSON格式文件的处理来演示File源的使用方法。

整个例子主要包括以下两个步骤：

第一步，创建程序生成JSON格式的File源测试数据；

第二步，创建程序对数据进行统计。

这样的设计让我们可以完整地看到从数据生成到数据处理的整个流程。

SI-09 슬라이드 9: 测试数据生成程序（1/5）

现在我们开始第一步，创建程序生成JSON格式的File源测试数据。

为了演示JSON格式文件的处理，这里我们随机生成一些JSON格式的文件来进行测试。

代码文件叫spark_ss_filesource_generate.py。

这段程序首先建立测试环境，清空测试数据所在的目录，接着使用for循环一千次来生成一千个文件，文件名为"e-mall-数字.json"，文件内容是不超过100行的随机JSON行。

每行的格式是这样的：

{"eventTime": 1546939167, "action": "logout", "district": "fujian"}

这个JSON包含三个字段：eventTime是事件时间，action是动作（比如登录、登出、购买），district是地区。

让我们先看看需要导入的模块：

#!/usr/bin/env python3

-*- coding: utf-8 -*-

导入需要用到的模块

```
import os
import shutil
import random
import time
```

SL-10 슬라이드 10: 测试数据生成程序 (2/5)

接下来我们定义一些常量和模式。大家看这里：

```
TEST_DATA_TEMP_DIR = '/tmp/'
TEST_DATA_DIR = '/tmp/testdata/'
ACTION_DEF = ['login', 'logout', 'purchase']
DISTRICT_DEF = ['fujian', 'beijing', 'shanghai', 'guangzhou']
JSON_LINE_PATTERN = '{{"eventTime": {}, "action": "{}", "district": "{}"}}\n'
```

这里我们定义了临时目录、测试数据目录，还有动作的定义和地区的定义。

JSON_LINE_PATTERN是我们生成JSON行的模板(múbǎn Template)。

然后是测试环境搭建的函数：

```
def test_setUp():
    if os.path.exists(TEST_DATA_DIR):
        shutil.rmtree(TEST_DATA_DIR, ignore_errors=True)
    os.mkdir(TEST_DATA_DIR)
```

这个函数的作用是判断(pànduàn)文件夹是否存在，如果存在则删除旧数据，并建立文件夹。

这样确保我们每次运行都是从一个干净的环境开始。

SL-11 슬라이드 11: 测试数据生成程序（3/5)

然后我们有测试环境的恢复函数和文件生成函数：

```
def test_tearDown():
    if os.path.exists(TEST_DATA_DIR):
        shutil.rmtree(TEST_DATA_DIR, ignore_errors=True)

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

这里要特别注意write_and_move函数，它体现了我们之前说的原子性操作。

我们先把文件写到临时目录，

然后再移动到目标目录。

这样确保Spark读取的时候文件已经完整写入了。

SL-12 슬라이드 12: 测试数据生成程序（4/5)

现在我们看主程序部分：

```
if __name__ == "__main__":
    test_setUp()
    for i in range(1000):
        filename = 'e-mall-{}.json'.format(i)
        content = ''
        rndcount = list(range(100))
        random.shuffle(rndcount)
        for _ in rndcount:
            content += JSON_LINE_PATTERN.format(
                str(int(time.time())),
                random.choice(ACTION_DEF),
                random.choice(DISTRICT_DEF))
        write_and_move(filename, content)
        time.sleep(1)
    test_tearDown()
```

这个主程序做什么呢？

它生成1000个文件，每个文件包含随机数量（不超过100行）的JSON数据。

每行数据包含当前时间戳、随机的动作和随机的地区。

注意这里有个time.sleep(1)，这是为了模拟(추상은 mó, 구체는 mú)文件是逐渐(zhújiàn 점차)产生的，而不是一次性全部产生的。

```
#导入需要用到的模块
import os
import shutil
import random
import time

TEST_DATA_TEMP_DIR='/tmp/'
TEST_DATA_DIR='/tmp/testdata/'

ACTION_DEF=['login','logout','purchase']
DISTRICT_DEF=['fujian','beijing','shanghai','guangzhou']
JSON_LINE_PATTERN='{{"eventTime":{},"action":"{}","district":"{}"}}₩n'

#测试环境搭建,判断文件夹是否存在,如果存在则删除旧数据,并建立文件夹
def test_setUp():
    if os.path.exists(TEST_DATA_DIR):
        shutil.rmtree(TEST_DATA_DIR,ignore_errors=True)
    os.mkdir(TEST_DATA_DIR)

# 测试环境的恢复, 对文件夹进行清理
def test_tearDown():
    if os.path.exists(TEST_DATA_DIR):
        shutil.rmtree(TEST_DATA_DIR,ignore_errors=True)
# 生成测试文件
def write_and_move(filename,data):
    with open(TEST_DATA_TEMP_DIR+filename,
        "wt",encoding="utf-8") as f:
        f.write(data)
    shutil.move(TEST_DATA_TEMP_DIR+filename,
                TEST_DATA_DIR+filename)

if __name__=='__main__':
    test_setUp()
    for i in range(1000):
        filename='e-mail-{}.json'.format(i)
        content=''
        rndcount=list(range(1000))
        random.shuffle(rndcount)
```

```
        for _ in rndcount:
            content+=JSON_LINE_PATTERN.format(
                str(int(time.time())),
                random.choice(ACTION_DEF),
                random.choice(DISTRICT_DEF))
        write_and_move(filename,content)

        time.sleep(1)

    test_tearDown()
```

SL-13 슬라이드 13: 数据统计程序（1/4)

现在我们看第二步，创建程序对数据进行统计。这个文件叫spark_ss_filesource.py。

首先是导入需要的模块：

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

导入需要用到的模块

```
import os
import shutil
from pprint import pprint

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

定义JSON文件的路径常量

```
TEST_DATA_DIR_SPARK = 'file:///tmp/testdata/'
```

这里我们导入了PySpark相关的模块，

包括SparkSession、

window函数、

以及数据类型定义。

SL-14 슬라이드 14: 数据统计程序（2/4）

接下来是程序的主要部分，首先是定义模式（Schema）和创建SparkSession：

```
if __name__ == "__main__":
    # 定义模式，为时间戳类型的eventTime、字符串类型的操作和省份组成
    schema = StructType([
        StructField("eventTime", TimestampType(), True),
        StructField("action", StringType(), True),
        StructField("district", StringType(), True)])

    spark = SparkSession \
        .builder \
        .appName("StructuredEMallPurchaseCount") \
        .getOrCreate()

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

这里我们定义了一个模式，指定eventTime是时间戳类型，action和district是字符串类型。然后创建了一个SparkSession，应用名叫"StructuredEMallPurchaseCount"。

SL-15 슬라이드 15: 数据统计程序（3/4）

然后我们创建数据流和窗口统计：

```
lines = spark \
    .readStream \
    .format("json") \
    .schema(schema) \
    .option("maxFilesPerTrigger", 100) \
    .load(TEST_DATA_DIR_SPARK)

# 定义窗口
windowDuration = '1 minutes'

windowedCounts = lines \
    .filter("action = 'purchase'") \
    .groupBy('district', window('eventTime', windowDuration)) \
    .count() \
    .sort(asc('window'))
```

这里我们用readStream读取JSON格式的文件，设置schema和maxFilesPerTrigger选项。然后定义一个1分钟的窗口，过滤出购买动作，按地区和时间窗口分组，计算购买次数。

SL-16 슬라이드 16: 数据统计程序（4/4）

最后是启动查询：

```
query = windowedCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .option('truncate', 'false') \
    .trigger(processingTime="10 seconds") \
    .start()

query.awaitTermination()
```

这里我们用writeStream把结果输出到控制台，设置输出模式为"complete"，每10秒触发一次处理，然后等待查询终止。

SL-17 슬라이드 17: 测试运行程序

现在我们来看如何运行这个程序。

程序运行过程需要访问HDFS（Hadoop Distributed File System），因此，需要启动HDFS，命令如下：

```
$ cd /usr/local/hadoop
$ sbin/start-dfs.sh
```

然后新建一个终端，执行如下命令生成测试数据：

```
$ cd /usr/local/spark/mycode/structuredstreaming/file
$ python3 spark_ss_filesource_generate.py
```

再新建一个终端，执行如下命令运行数据统计程序：

```
$ cd /usr/local/spark/mycode/structuredstreaming/file
$ /usr/local/spark/bin/spark-submit spark_ss_filesource.py
```

这样我们就有三个终端在运行：一个生成数据，一个处理数据，还有一个是HDFS服务。

SL-18 슬라이드 18: 运行结果展示

运行程序以后，可以看到类似如下的输出结果：

```
-------------------------------------------
Batch: 0
-------------------------------------------
+---------+-----------------------------------------+-----+
|district |window                                   |count|
+---------+-----------------------------------------+-----+
|guangzhou|{2023-01-08 17:19:00, 2023-01-08 17:20:00}|283  |
|shanghai |{2023-01-08 17:19:00, 2023-01-08 17:20:00}|251  |
|fujian   |{2023-01-08 17:19:00, 2023-01-08 17:20:00}|258  |
|beijing  |{2023-01-08 17:19:00, 2023-01-08 17:20:00}|258  |
|guangzhou|{2023-01-08 17:20:00, 2023-01-08 17:21:00}|492  |
|beijing  |{2023-01-08 17:20:00, 2023-01-08 17:21:00}|499  |
|fujian   |{2023-01-08 17:20:00, 2023-01-08 17:21:00}|513  |
|shanghai |{2023-01-08 17:20:00, 2023-01-08 17:21:00}|503  |
|guangzhou|{2023-01-08 17:21:00, 2023-01-08 17:22:00}|71   |
|fujian   |{2023-01-08 17:21:00, 2023-01-08 17:22:00}|74   |
|shanghai |{2023-01-08 17:21:00, 2023-01-08 17:22:00}|66   |
|beijing  |{2023-01-08 17:21:00, 2023-01-08 17:22:00}|52   |
+---------+-----------------------------------------+-----+
```

大家可以看到，这个结果显示了每个地区在每个1分钟时间窗口内的购买次数。比如广州在17:19到17:20这一分钟内有283次购买，上海有251次，等等。

随着时间的推移，你会看到新的批次数据不断出现，这就是Structured Streaming的实时处理效果。

好，今天的File源就讲到这里。大家有什么问题吗？

```
# 导入需要用到的模块
import os
import shutil
from pprint import pprint

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

#定义 JSON格式文件的路径常量
TEST_DATA_DIR_SPARK='file:///tmp/testdata/'

if __name__=='__main__':
    #定义模式,其由时间戳类型的eventTime以及字符串类型的action 和distric组成
```

```
schema=StructType([
    StructField("eventTime",TimestampType(),True),
    StructField("action",StringType(),True),
    StructField("district",StringType(),True)
])

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

spark.sparkContext.setLogLevel("WARN")

lines=spark ₩
    .readStream ₩
    .format("json") ₩
    .schema(schema) ₩
    .option("maxFilesPerTrigger",100) ₩
    .load(TEST_DATA_DIR_SPARK)

#定义窗口
windowDuration='1 minutes'
windowedCounts=lines ₩
    .filter("action='purchase'") ₩
    .groupBy('district', window('eventTime',windowDuration)) ₩
    .count() ₩
    .sort(asc('window'))

query=windowedCounts ₩
    .writeStream ₩
    .outputMode("complete") ₩
    .format("console") ₩
    .option('truncate','false') ₩
    .trigger(processingTime="10 seconds") ₩
    .start()
query.awaitTermination()
```

실행을 할때는 Compound 를 만들어서 실행시키는 방법 : 조심히 할 것

5) PyCharm 에서 "Compound Run" 만드는 실제 순서

5-1. Run Config A: FileProducer ([Run]-[Edit Configurations] 실행)

- Type: **Python (pop-up 창에서 왼쪽의 [Python]선택**
- [Add new run configuration...]클릭후에 이름 작성 [FileProducer]하고 Script path 에 p191... 파일 선택해 줌 그리고 나서 [Apply] 클릭
- Script path: PycharmProjects/Ch08StructuredStreamin/p195_file_data_source.py
- Working directory: 프로젝트 루트

5-2. Run Config B: FileStreamJob (spark-submit)

- Type: **Python (pop-up 창에서 왼쪽의 [Python]선택**
- [Add new run configuration...]클릭후에 이름 작성 [FileStreamJob]하고 Script path 에 p195... 파일 선택해 줌 그리고 나서 [Apply] 클릭
- Script path: StructuredStreamin/p196_spark_ss_filesource.py
- Working directory: 프로젝트 루트

5-3. Run Config C: FileCompoundPipeline

- Type: Compound **(pop-up 창에서 왼쪽의 [Compound]선택)**
- [Add new run configuration...]클릭후에 이름 작성 [FileCompoundPipeline]하고 Python 에서 FileProducer 와 FileStreamJob 를 선택하고 나서 [Apply] 클릭
- 실행 순서:
  - FileStreamJob **먼저**(스트림 대기 상태)
  - 그 다음 FileProducer (파일을 떨어뜨리기 시작)

5-4. 실행

- PyCharm 에서 [Run]-[FileCompoundPipeline] 실행 버튼 클릭
- [한참 기다린다.- 1 분 30 초 후에 응답 나옴.] batch 0 .. 이후 계속 나옴
- 종료 후에는 [Run] 메뉴에서 파일 생성도 추가로 중지 시킴. [Stop FileProducer]

SL-19 슬라이드 8.3.2 Kafka源

今天我们来学习 Spark Structured Streaming（结构化流处理）的数据源- Kafka源。

同学们，请看这张**슬라이드**。

大家知道，在流处理（Stream Processing）中，数据源的选择非常重要。

那么为什么说 Kafka源是流处理最理想的输入源呢？

主要有两个原因：zhǔ yào

第一， 它可以保证实时性（Real-time）；
第二， 它具有很好的容错性（Fault Tolerance）。

接下来我们看看 Kafka源都有哪些配置选项。

这些选项（Options）对我们实际使用非常重要：

首先是 assign，它用来指定我们要消费的 Kafka主题（Topic）和分区（Partition）。

然后是 subscribe，这个是订阅 Kafka主题，可以是多个主题，用逗号分隔。

第三个是 subscribePattern，这是一个正则表达式，可以匹配多个主题，很灵活。

kafka.bootstrap.servers 是 Kafka服务器的列表，格式是 "host:port"。

startingOffsets 和 endingOffsets 分别是起始和结束位置的偏移量（Offset）。

最后一个 failOnDataLoss 是个布尔值，

表示当 Kafka数据可能丢失时是否触发流计算失败。

一般我们建议禁止，避免误报。

SL-20 슬라이드 8.3.2 Kafka源示例代码

好，现在我们看一个简单的示例代码。

这个代码非常简洁，但是包含了使用 Kafka源的核心要素。

```
df = spark \
    .readstream() \
    .format('kafka') \
    .option('subscribe', 'input') \
    .load()
```

大家看，我们首先调用 spark.readStream()，注意这里是 readStream，

不是 read，因为我们处理的是流数据（Streaming Data）。

然后用 format('kafka') 指定我们使用的是 Kafka格式。

接着用 option('subscribe', 'input') 订阅名为 'input' 的主题。

最后调用 load() 加载数据。

就这么简单！

- **调用 (diào yòng): 프로그램이 함수(function), 서브루틴(subroutine), 메소드(method), 명령 등을 실행하기 위해 불러오는 행위를 말합니다.**
- **订阅报纸/杂志 *Dìngyuè bàozhǐ / zázhì* 신문/잡지를 구독하다.**

SL-21 슬라이드 실例演示介绍

好，接下来我们要做一个完整的实例演示。

这个实例会帮助大家更好地理解 Kafka源的工作原理。**bāng zhù**

在这个实例中，我们要做什么呢？

首先，我们会写一个生产者程序（Producer Program），

这个程序每0.1秒生成一个包含2个字母的单词，

然后把这些单词写入到 Kafka的一个主题中，这个主题叫做 "wordcount-topic"。

然后，我们写一个 Spark消费者程序（Consumer Program），它会订阅 wordcount-topic，源源不断地收到单词。

每隔8秒钟，它会对收到的单词进行一次词频统计（Word Count），把统计结果输出到另一个 Kafka主题 "wordcount-result-topic" 中。

同时，我们还会开启2个监控程序来检查 Spark处理的输入和输出结果。

在开始之前，我需要提醒大家，为了顺利完成这个实验，你们需要安装比较新的 Kafka版本。建议到 Kafka官网下载 Kafka_**2.12-3.5.1.**tgz。

这里我假设大家的 Kafka安装目录是 "/usr/local/kafka"。

源源不断 (yuán yuán bù duàn)

- **뜻**: 계속 끊이지 않는 모양을形容하여, 마치 물줄기처럼 끝이 없는 것을 강조합니다. **지속적이고 양이 많음**을 강조합니다.
- **한국어**: 끊임없이, 연속적으로, 줄줄이

SL-22 슬라이드 1. 启动 Kafka

现在我们开始实际操作。

首先要启动 Kafka。

大家要注意，Kafka依赖于 Zookeeper，所以我们要先启动 Zookeeper。

请大家在 Linux系统中新建一个终端， zhōng duān

我们叫它 "Zookeeper终端"。

在这个终端中输入下面的命令：

```
# ZooKeeper 기본 포트: 2181
netstat -tulpn | grep :2181
확인 후에 다음 사항은 Pycharm에서 설정함
$ cd /usr/local/kafka
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

[1] [Run]-[Edit configurations...]를 선택한다.

[2] 왼쪽의 [Shell Script]선택 하고, [Add new run configurations...]를 클릭한다.

[3] Name칸에 ZookeeperServer2025로 입력하고

Script Path: /usr/local/kafka/bin/zookeeper-server-start.sh

Script options: config/zookeeper.properties

Working directory: /usr/local/kafka

차례대로 입력한 후에 오른쪽 아래에서 [Apply]를 클릭한다.

大家看，首先进入 Kafka目录，然后运行 zookeeper-server-start.sh 脚本。

特别注意：不要关闭这个终端窗口！

一旦关闭，Zookeeper服务就停止了。

接下来，请大家再打开一个终端，

我们叫它 "Kafka终端"。

在这个终端中输入：

```
$ cd /usr/local/kafka
$ bin/kafka-server-start.sh config/server.properties
# Kafka Broker 기본 포트: 9092
netstat -tulpn | grep :9092
```

[1] [Run]-[Edit configurations...]를 선택한다.

[2] 왼쪽의 [Shell Script]선택 하고, [Add new run configurations...]를 클릭한다.

[3] Name칸에 KafkaServerServer2025로 입력하고

Script Path: /usr/local/kafka/bin/kafka-server-start.sh

Script options: config/server.properties

Working directory: /usr/local/kafka

차례대로 입력한 후에 오른쪽 아래에서 [Apply]를 클릭한다.

同样，这个终端也不能关闭，否则 Kafka服务就停止了。

SL-23 设置监控终端

为了方便观察我们程序的运行效果，我们需要设置两个监控终端。

首先，再新开一个终端，我们叫它 "监控输入终端"。

在这个终端中执行下面的命令：

```
$ cd /usr/local/kafka
$ bin/kafka-console-consumer.sh \
> --bootstrap-server localhost:9092 --topic wordcount-topic
```

[1] [Run]-[Edit configurations...]를 선택한다.

[2] 왼쪽의 [Shell Script]선택 하고, [Add new run configurations...]를 클릭한다.

[3] Name칸에 KafkaInputMonitor2025로 입력하고

Script Path: /usr/local/kafka/bin/kafka-console-consumer.sh

Script options: -server localhost:9092 --topic wordcount-topic

Working directory: /usr/local/kafka

차례대로 입력한 후에 오른쪽 아래에서 [Apply]를 클릭한다.

这个命令会监控 Kafka收到的文本，也就是我们生产者程序发送的单词。

然后，再新开一个终端，叫 "监控输出终端"，执行这个命令：

```
$ cd /usr/local/kafka
$ bin/kafka-console-consumer.sh \
> --bootstrap-server localhost:9092 --topic wordcount-result-topic
```

[1] [Run]-[Edit configurations...]를 선택한다.

[2] 왼쪽의 [Shell Script]선택 하고, [Add new run configurations...]를 클릭한다.

[3] Name칸에 KafkaInputMonitor2025로 입력하고

Script Path: /usr/local/kafka/bin/kafka-console-consumer.sh

Script options: -server localhost:9092 --topic wordcount-result-topic

Working directory: /usr/local/kafka

차례대로 입력한 후에 오른쪽 아래에서 [Apply]를 클릭한다.

这个命令会监控输出的结果，也就是词频统计的结果。

现在我们有四个终端了：Zookeeper终端、Kafka终端、监控输入终端、监控输出终端。

SL-24 2. 编写生产者程序

好，现在我们来编写生产者程序。我们创建一个文件叫 spark_ss_kafka_producer.py。

大家看代码的开头部分：

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

import string
import random
import time
from kafka import KafkaProducer
```

这里我们导入了几个必要的模块：

string用于生成字符，

random用于随机选择，

time用于控制时间间隔，

KafkaProducer是 Kafka的生产者类（Producer Class）。

SL-25 生产者程序主逻辑

现在看生产者程序的主要逻辑：

```
if __name__ == "__main__":
    producer = KafkaProducer(bootstrap_servers=['localhost:9092'])

    while True:
        s2 = (random.choice(string.ascii_lowercase) for _ in range(2))
        word = ''.join(s2)
        value = bytearray(word, 'utf-8')
        producer.send('wordcount-topic', value=value) \
            .get(timeout=10)
        time.sleep(0.1)
```

我们来一行一行分析：

首先创建一个 KafkaProducer对象，连接到本地的9092端口。

然后进入无限循环。

每次循环中：

- 随机生成2个小写字母
- 把它们连接成一个单词
- 转换成字节数组（ByteArray）
- 发送到 'wordcount-topic' 主题
- 等待0.1秒

这样我们就能每0.1秒生成一个2字母的随机单词。

```
import string
import random
import time
from kafka import KafkaProducer
from kafka.errors import KafkaError

if __name__ == '__main__':
    try:
        producer = KafkaProducer(bootstrap_servers=['localhost:9092'])

        while True:
            # 2글자 랜덤 단어 생성
            word = ''.join(random.choices(string.ascii_lowercase, k=2))
            value = word.encode('utf-8')

            try:
                # 메시지 전송
                future = producer.send('wordcount-topic', value=value)
                record_metadata = future.get(timeout=1.0)
            #     print(f"Sent: {word} to partition {record_metadata.partition} at offset {record_metadata.of
fset}")
            except KafkaError as e:
                print(f"Failed to send message: {e}")

            time.sleep(0.1)

    except Exception as e:
        print(f"Error initializing producer: {e}")
    finally:
        if 'producer' in locals():
            producer.close()
```

SL-26 Python Kafka模块安装及运行(여기는 이미 설치했으므로 설치는 통과하고, 아랫부분 실행부분만)

在运行生产者程序之前，我们需要确保安装了 Python的 Kafka支持。

如果还没有安装，请按照以下步骤：

第一步，确认是否安装了 pip3：

```
$ sudo apt-get install pip3
```

第二步，安装 kafka-python模块：

```
$ sudo pip3 install kafka-python
```

安装完成后，在终端中执行生产者程序：

```
$ cd /usr/local/spark/mycode/structuredstreaming/kafka/
$ python3 spark_ss_kafka_producer.py
```

生产者程序运行后，大家应该能在 "监控输入终端" 看到持续输出的2字母单词，比如 "ab", "cd", "ef" 等等。

SL-27 3. 编写消费者程序

现在我们来编写消费者程序。这个程序使用 Spark Structured Streaming 来处理从 Kafka接收的数据。

文件名是 spark_ss_kafka_consumer.py，开头部分：

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

from pyspark.sql import SparkSession

if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .appName("StructuredKafkaWordCount") \
        .getOrCreate()

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

这里我们导入 SparkSession，

创建一个 Spark应用，应用名称是 "StructuredKafkaWordCount"。

注意我们把日志级别设置为 'WARN'，这样可以减少不必要的输出信息。

编写	**biān xiě**
消费者	**xiāo fèi zhě**
程序	**chéng xù**

日志	**rì zhì**	**4 声, 4 声**	**로그(Log)**
级别	**jí bié**	**2 声, 2 声**	**레벨, 등급**
设置	**shè zhì**	**4 声, 4 声**	**설정(하다)**

SL-28 消费者程序 - 数据读取

接下来是数据读取部分：

```
lines = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", 'wordcount-topic') \
    .load() \
    .selectExpr("CAST(value AS STRING)")

wordCounts = lines.groupBy("value").count()
```

让我详细解释一下：

首先，我们使用 spark.readStream 创建一个流数据读取器（Stream Reader）。

format("kafka") 指定我们从 Kafka读取数据。

然后设置 Kafka服务器地址和订阅的主题。

load() 加载数据后，

我们用 selectExpr("CAST(value AS STRING)") 把接收到的字节数据转换成字符串。

最后，lines.groupBy("value").count() 对单词进行分组并计数，

这就是我们的词频统计（Word Count）逻辑。

SL-29 消费者程序 - 输出设置

现在是输出设置部分：

```
query = wordCounts \
    .selectExpr("CAST(value AS STRING) as key",
                "CONCAT(CAST(value AS STRING), ':', CAST(count AS STRING)) as
value") \
    .writeStream \
    .outputMode("complete") \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("topic", "wordcount-result-topic") \
    .option("checkpointLocation", "file:///tmp/kafka-sink-cp") \
    .trigger(processingTime="8 seconds") \
    .start()

query.awaitTermination()
```

这段代码比较复杂，我们一步步来看：

首先，selectExpr 把计数结果格式化成 "单词:次数" 的格式。

writeStream 创建输出流（Output Stream）。

outputMode("complete") 表示每次输出完整的结果。

format("kafka") 表示输出到 Kafka。

然后设置输出的 Kafka主题为 "wordcount-result-topic"。

checkpointLocation 设置检查点位置，这对容错很重要。

trigger(processingTime="8 seconds") 设置每8秒处理一次。

最后 start() 启动查询，awaitTermination() 等待程序结束。

```
from pyspark.sql import SparkSession

if __name__ == '__main__':
    spark = SparkSession ₩
        .builder ₩
        .appName("StructuredKafkaWordCount") ₩
        .getOrCreate()

    spark.sparkContext.setLogLevel("WARN")

    lines = spark ₩
        .readStream ₩
        .format("kafka") ₩
        .option("kafka.bootstrap.servers", "localhost:9092") ₩
        .option("subscribe", "wordcount-topic") ₩
        .option("startingOffsets", "latest") ₩
        .load() ₩
        .selectExpr("CAST(value AS STRING)")

    wordCounts=lines.groupBy("value").count()

    query=wordCounts ₩
        .selectExpr("CAST(value AS STRING) as key", "CONCAT(CAST(value AS STRING),':',CAST(count AS STRING)) as value") ₩
        .writeStream ₩
        .outputMode("complete") ₩
        .format("kafka") ₩
        .option("kafka.bootstrap.servers", "localhost:9092") ₩
        .option("topic", "wordcount-result-topic") ₩
        .option("checkpointLocation", "file:///tmp/kafka-sink-cp") ₩
        .trigger(processingTime="8 second") ₩
        .start()

    query.awaitTermination()
    spark.stop()
```

SL-30 消费者程序执行及结果

现在我们来运行消费者程序。在终端中执行：

```
$ cd /usr/local/spark/mycode/structuredstreaming/kafka/
$ /usr/local/spark/bin/spark-submit \
> --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.4.0 \
> spark_ss_kafka_consumer.py
```

注意这里我们使用 spark-submit 命令，而不是直接运行 python。

还需要用 --packages 参数指定 Kafka连接器的依赖包。

消费者程序运行起来后，大家可以在 "监控输出终端" 看到类似这样的输出结果：

sq:3

bl:6

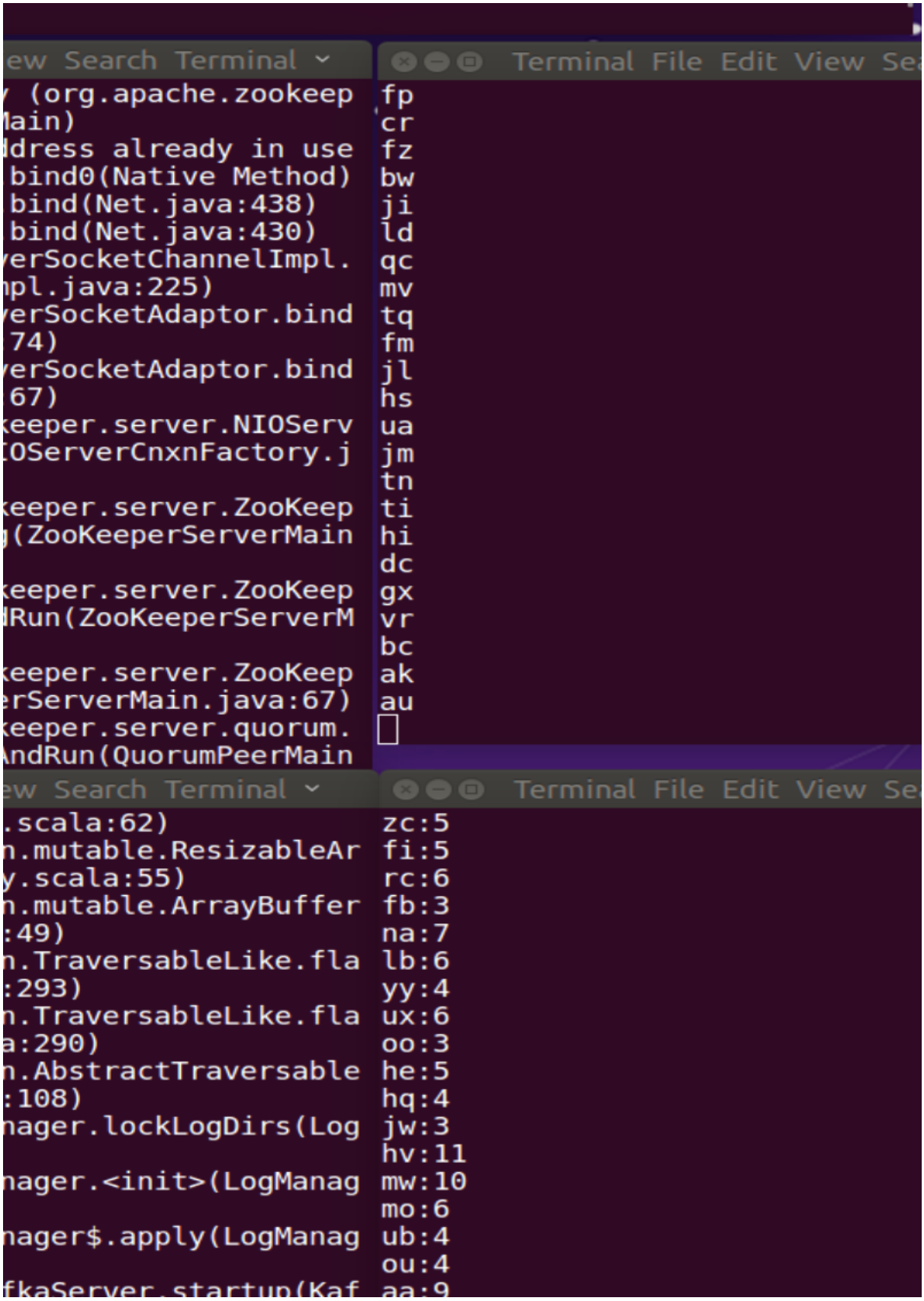
lo:8

...

这表示单词 "sq" 出现了3次，"bl" 출현了6次，"lo" 출현了8次，等等。

- Compound 만드는 방법은 앞에서와 동일함

종합적으로 내가 해보니까 터머널을 compound에 넣는 것 까지 하는 것은 보기가 나쁘고, 그냥 터미널을 4개 열어서 놓고 Python file 2개(consumer, producer)를 compound로 만들어서 실행하는 것이 좋음.
처음 실행하면 Kafka 입력만 표시되고 출력은 모니터링되지 않음.
한참 기다리면 출력도 자연스럽게 나옴.(시간을 가지고 기다릴 것)


ew Search Terminal
(org.apache.zookeep
Main)
ddress already in use
bind0(Native Method)
bind(Net.java:438)
bind(Net.java:430)
verSocketChannelImpl.
mpl.java:225)
verSocketAdaptor.bind
74)
verSocketAdaptor.bind
67)
keeper.server.NIOServ
IOServerCnxnFactory.j
keeper.server.ZooKeep
g(ZooKeeperServerMain
keeper.server.ZooKeep
dRun(ZooKeeperServerM
keeper.server.ZooKeep
erServerMain.java:67)
keeper.server.quorum.
AndRun(QuorumPeerMain
Terminal File Edit View Se
fp
cr
fz
bw
ji
ld
qc
mv
tq
fm
jl
hs
ua
jm
tn
ti
hi
dc
gx
vr
bc
ak
au
ew Search Terminal
.scala:62)
n.mutable.ResizableAr
y.scala:55)
n.mutable.ArrayBuffer
:49)
n.TraversableLike.fla
:293)
n.TraversableLike.fla
a:290)
n.AbstractTraversable
:108)
nager.lockLogDirs(Log
nager.<init>(LogManag
nager$.apply(LogManag
fkaServer.startup(Kaf
Terminal File Edit View Se
zc:5
fi:5
rc:6
fb:3
na:7
lb:6
yy:4
ux:6
oo:3
he:5
hq:4
jw:3
hv:11
mw:10
mo:6
ub:4
ou:4
aa:9

SL-31 8.3.3 Socket源

好，现在我们来学习第二种数据源 - Socket源（Socket Source）。

Socket源从一个本地或远程主机的某个端口服务上读取数据，数据的编码为 UTF8。

但是大家要注意，Socket源有一些限制：

第一，它使用内存保存读取到的所有数据。

第二，远端服务不能保证数据在出错后可以使用检查点（Checkpoint）或者指定当前已处理的偏移量（Offset）来重放数据。

因此，Socket源无法提供端到端的容错保障（End-to-End Fault Tolerance）。

所以，Socket源一般仅用于测试或学习用途，不建议在生产环境中使用。

Socket源的配置选项很简单：

host：主机 IP地址或者域名，这是必须设置的。

port：端口号，也是必须设置的。

includeTimestamp：是否在数据行内包含时间戳（Timestamp）。使用时间戳可以用来测试基于时间聚合（Time-based Aggregation）的功能。

关于 Socket源的实例，大家可以参考8.2节的 StructuredNetworkWordCount.py。

여기서 실습은 앞에서 한 것(8.2.2)과 유사함

먼저 터미널을 하나 만들어서

nc -lk 9999

한 후에 프로그램을 실행한다.

그 다음에 text를 입력한다.

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

if __name__ == "__main__":
    # 创建 Spark 会话(session)
    spark = SparkSession.builder ₩
        .appName("StructuredNetworkWordCount") ₩
        .getOrCreate()

    # 设置日志等级为 WARN，减少干扰信息
    spark.sparkContext.setLogLevel("WARN")

    # 创建流式数据源，从 socket 接收数据
    lines = spark.readStream ₩
        .format("socket") ₩
        .option("host", "localhost") ₩
        .option("port", 9999) ₩
        .load()

    # 拆分单词并计数
    words = lines.select(
        explode(split(lines.value, " ")).alias("word")
    )
    wordCounts = words.groupBy("word").count()

    # 控制台输出结果(console)
    query = wordCounts.writeStream ₩
        .outputMode("complete") ₩
        .format("console") ₩
        .option("truncate", False) ₩
        .trigger(processingTime="8 seconds") ₩
        .start()

    query.awaitTermination()
```

SL-32 8.3.4 Rate源

最后我们来学习第三种数据源 - Rate源（Rate Source）。

Rate源是一个很有趣的数据源，它可以每秒生成特定个数的数据行。每个数据行包括时间戳（Timestamp）和值（Value）字段。

时间戳是消息发送的时间，值是从开始到当前消息发送的总个数，从0开始计数。

Rate源主要用来做什么呢？一般用作调试（Debugging）或性能基准测试（Performance Benchmarking）。

现在我们看看 Rate源的代码实现，文件名是 spark_ss_rate.py：

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

from pyspark.sql import SparkSession

if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .appName("TestRateStreamSource") \
        .getOrCreate()

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

这部分和之前的消费者程序很相似，只是应用名称改成了 "TestRateStreamSource"。

SL-33 Rate源实现代码

现在看 Rate源的核心实现：

```
lines = spark \
    .readStream \
    .format("rate") \
    .option('rowsPerSecond', 5) \
    .load()

print(lines.schema)

query = lines \
    .writeStream \
    .outputMode("update") \
    .format("console") \
    .option('truncate', 'false') \
    .start()
query.awaitTermination()
```

让我解释一下这段代码：

首先，format("rate") 指定我们使用 Rate源。

option('rowsPerSecond', 5) 设置每秒生成5行数据。

print(lines.schema) 打印数据的结构（Schema）。

然后我们把数据输出到控制台（Console），outputMode("update") 表示只输出新增的数据。

option('truncate', 'false') 表示不截断长文本。

SL-34 Rate源执行命令及初始结果

现在我们来运行 Rate源程序。在 Linux终端中执行：

```
$ cd /usr/local/spark/mycode/structuredstreaming/rate/
$ /usr/local/spark/bin/spark-submit spark_ss_rate.py
```

执行后，首先会看到数据结构的输出：

```
StructType(List(StructField(timestamp,TimestampType,true),StructField(value,LongType,true)))
```

这告诉我们数据有两个字段：timestamp（时间戳类型）和 value（长整型）。

然后是第一个批次（Batch 0）的输出，通常是空的：

```
-------------------------------------------
Batch: 0
```

```
----------------------------------------
+---------+-----+
|timestamp|value|
+---------+-----+
+---------+-----+
```

SL-35 Rate源执行结果 - 批次1

接下来我们看到第一个有数据的批次（Batch 1）：

```
-------------------------------------------
Batch: 1
-------------------------------------------
+-----------------------+-----+
|timestamp              |value|
+-----------------------+-----+
|2023-10-01 15:42:38.595|0    |
|2023-10-01 15:42:38.795|1    |
|2023-10-01 15:42:38.995|2    |
|2023-10-01 15:42:39.195|3    |
|2023-10-01 15:42:39.395|4    |
+-----------------------+-----+
```

大家看，这里有5行数据，正好符合我们设置的 rowsPerSecond=5。

时间戳大约每200毫秒增加一次（因为5行/秒 = 200毫秒/行）。

值从0开始递增：0, 1, 2, 3, 4。

这就是 Rate源的工作方式，它按照设定的频率生成带有时间戳和递增值的数据，非常适合用来测试流处理程序的性能和正确性。

总结

好，今天我们学习了 Spark Structured Streaming 的三种主要数据源：

1. **Kafka源**：最适合生产环境的流数据源，提供可靠性和实时性
2. **Socket源**：简单的测试用数据源，不适合生产环境
3. **Rate源**：用于性能测试和调试的合成数据源

每种数据源都有其特定的用途和适用场景。在实际项目中，我们要根据需求选择合适的数据源。

大家有什么问题吗？