大数据技术基础

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

第五章 大数据处理
——流计算框架

鄂海红 计算机学院 教授
ehaihong@bupt.edu.cn 微信: 87837001 QQ: 3027581960

大数据处理—流计算框架

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Contents

目 录

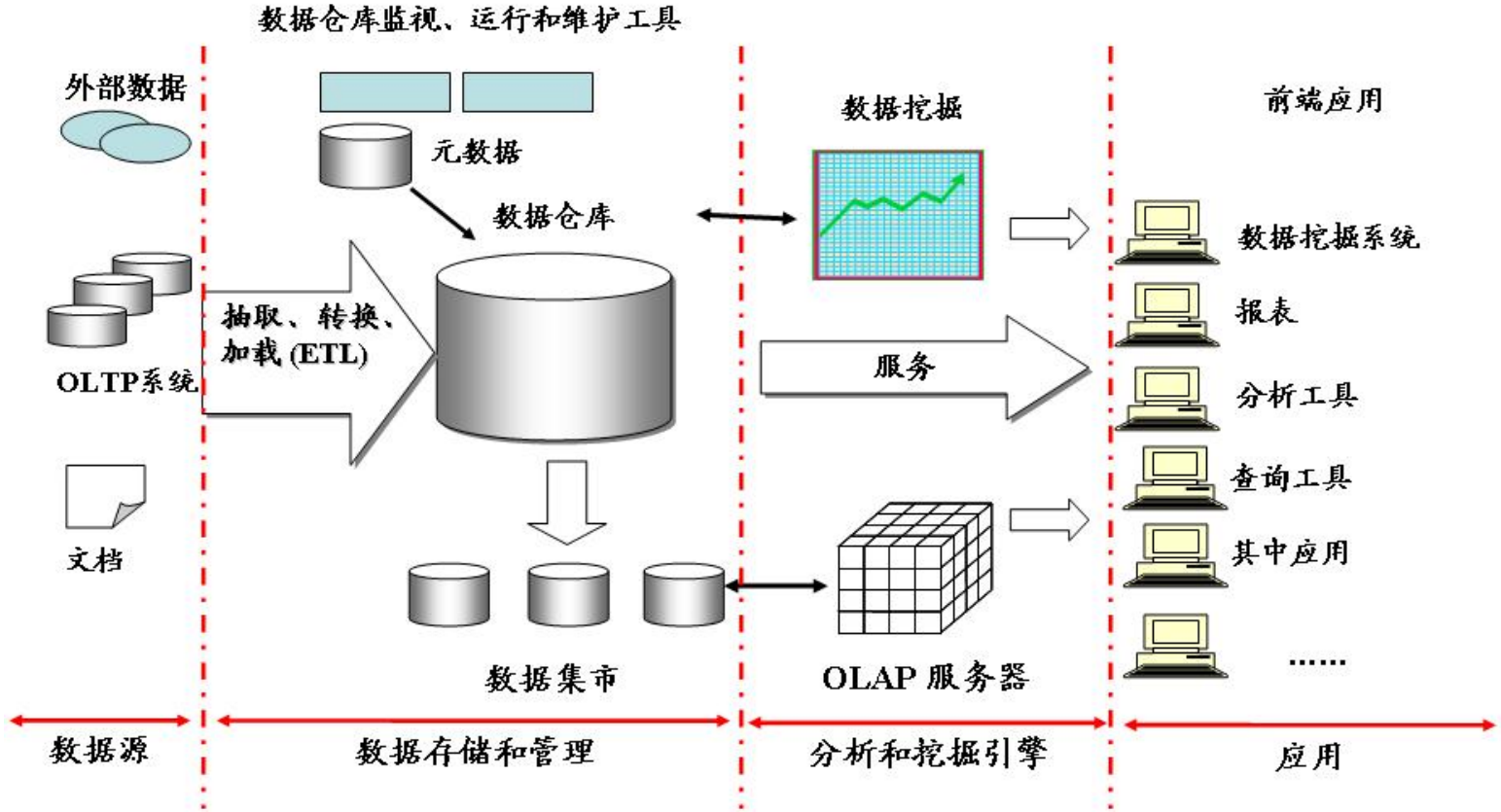
大数据处理—流计算

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

流计算概述

静态数据和流数据

- 很多企业为了支持决策分析而构建的数据仓库系统，其中存放的大量历史数据就是静态数据
- 技术人员可以利用数据挖掘和OLAP（On-Line Analytical Processing）分析工具从静态数据中找到对企业有价值的信息



静态数据和流数据

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

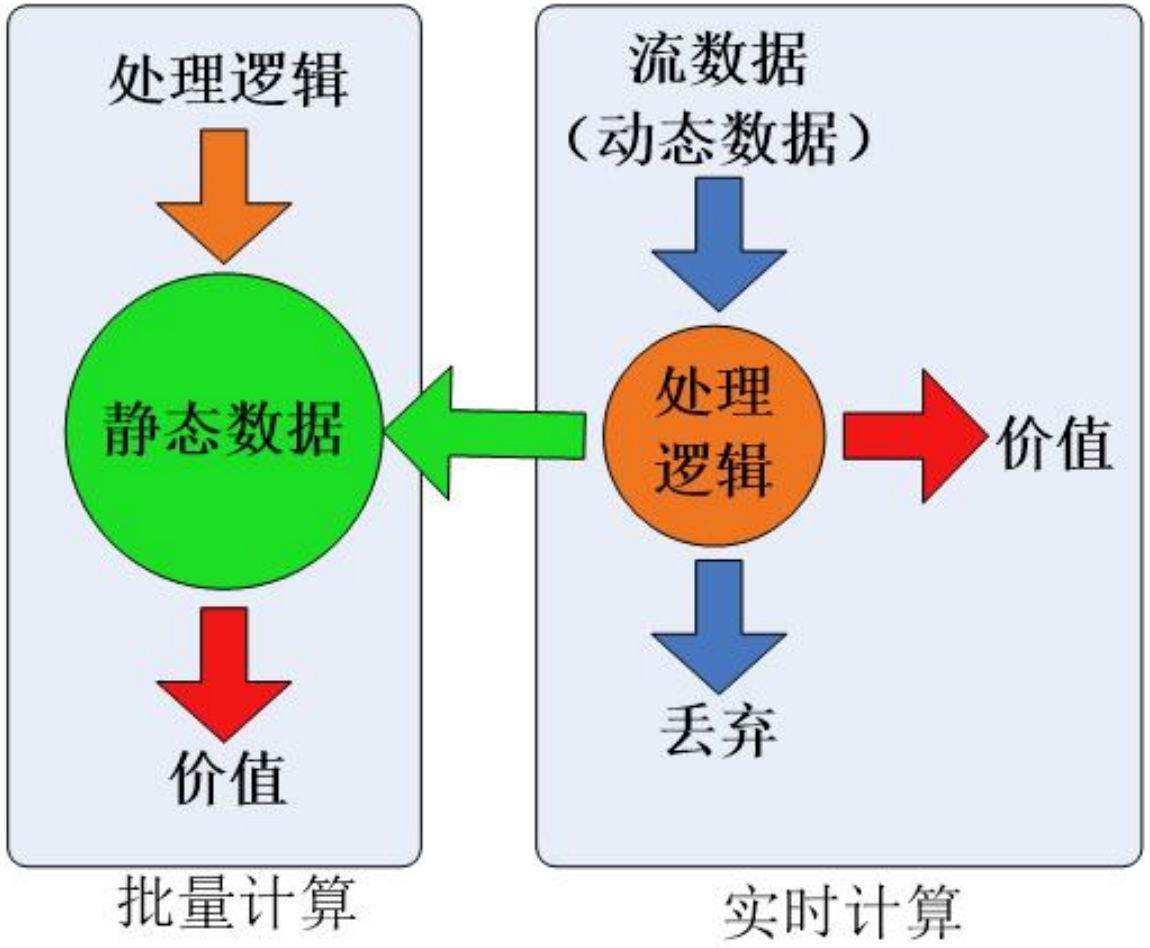
- 近年来，在Web应用、网络监控、传感监测等领域，兴起了一种新的数据密集型应用——流数据，即数据以大量、快速、时变的流形式持续到达
- 实例：PM2.5检测、电子商务网站用户点击流

- 流数据具有如下特征：
  - 数据快速持续到达，潜在大小也许是无穷无尽的
  - 数据来源众多，格式复杂
  - 数据量大，但是不十分关注存储，一旦经过处理，要么被丢弃，要么被归档存储
  - 注重数据的整体价值，不过分关注个别数据
  - 数据顺序颠倒，或者不完整，系统无法控制将要处理的新到达的数据元素的顺序

批量计算和实时计算

- 对静态数据和流数据的处理，对应着两种截然不同的计算模式：批量计算和实时计算

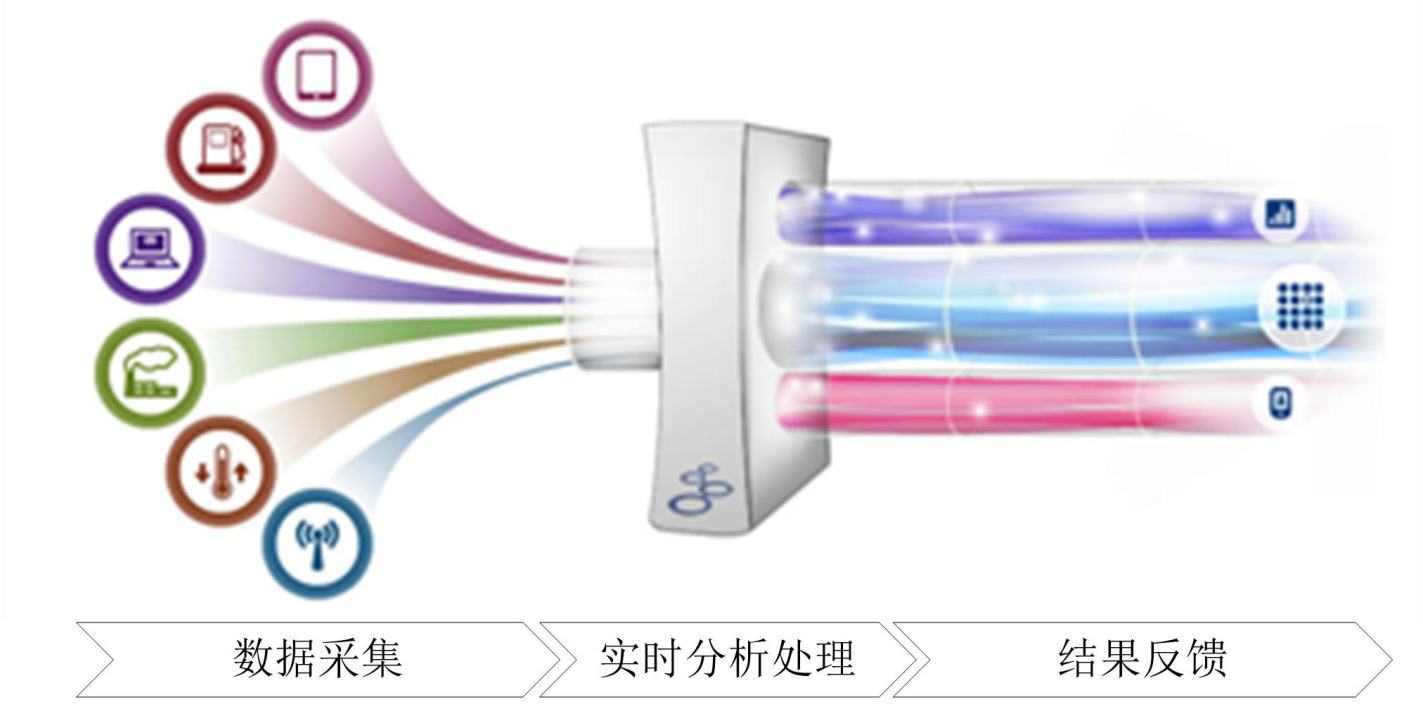
- 批量计算：充裕时间处理静态数据，如Hadoop
- 流数据不适合采用批量计算，因为流数据不适合用传统的关系模型建模
- 流数据必须采用实时计算，响应时间为秒级
- 数据量少时，不是问题；但是，在大数据时代，数据格式复杂、来源众多、数据量巨大，对实时计算提出了很大的挑战
- 因此，针对流数据的实时计算——流计算，应运而生



数据的两种处理模型

流计算概念

- 流计算：实时获取来自不同数据源的海量数据，经过实时分析处理，获得有价值的信息



流计算示意图

流计算概念

- 流计算秉承一个基本理念，即**数据的价值随着时间的流逝而降低**，如用户点击流
- 因此，当事件出现时就应该立即进行处理，而不是缓存起来进行批量处理。为了及时处理流数据，就需要一个低延迟、可扩展、高可靠的处理引擎

对于一个流计算系统来说，它应达到如下需求：

- 高性能：处理大数据的基本要求，如每秒处理几十万条数据
- 海量式：支持TB级甚至是PB级的数据规模
- 实时性：保证较低的延迟时间，达到秒级别，甚至是毫秒级别
- 分布式：支持大数据的基本架构，必须能够平滑扩展
- 易用性：能够快速进行开发和部署
- 可靠性：能可靠地处理流数据

流计算框架

- 当前业界诞生了许多专门的流数据实时计算系统来满足各自需求
- 常见的是开源流计算框架：
  - Storm：免费、开源的分布式实时计算系统，可简单、高效、可靠地处理大量的流数据
  - SparkStreaming：核心Spark API的扩展，不像Storm那样一次处理一个数据流。相反，它在处理数据流之前，会按照时间间隔对数据流进行分段切分。
  - Flink：针对流数据+批数据的计算框架。把批数据看作流数据的一种特例，延迟性较低(毫秒级)，且能够保证消息传输不丢失不重复。

传统的数据处理流程 VS 流计算处理流程

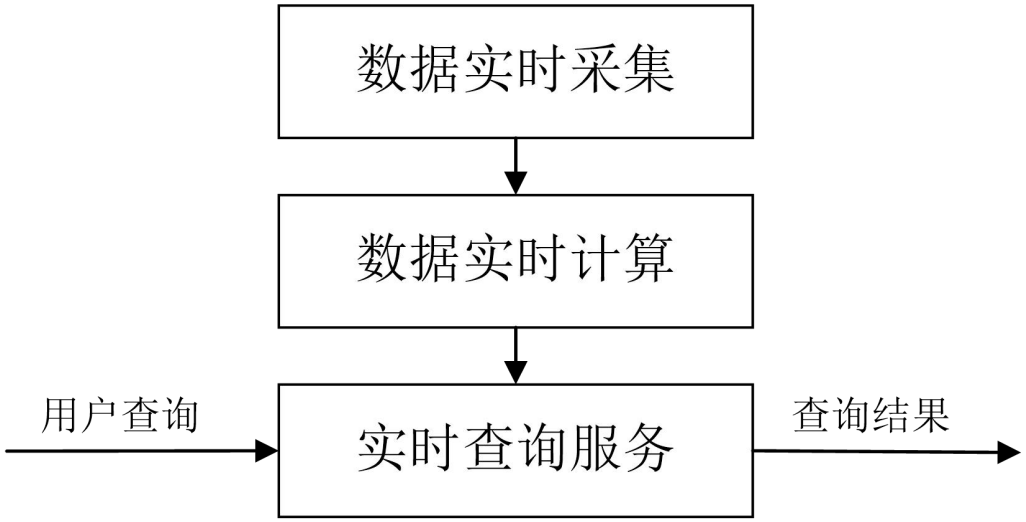
- 传统的数据处理流程，需要先采集数据并存储在关系数据库等数据管理系统中，之后由用户通过查询操作和数据管理系统进行交互

传统的数据处理流程示意图

- 传统的数据处理流程隐含了两个前提：
  - **存储的数据是旧的**。存储的静态数据是过去某一时刻的快照，这些数据在查询时可能已不具备时效性了
  - **需要用户主动发出查询来获取结果**

传统的数据处理流程 VS 流计算处理流程

- 流计算的处理流程一般包含三个阶段：数据实时采集、数据实时计算、实时查询服务



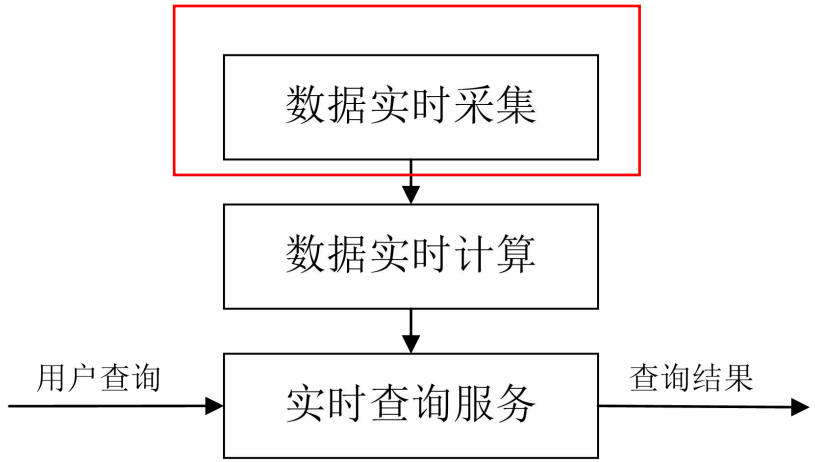
流计算处理流程示意图

流计算处理流程

1. 数据实时采集

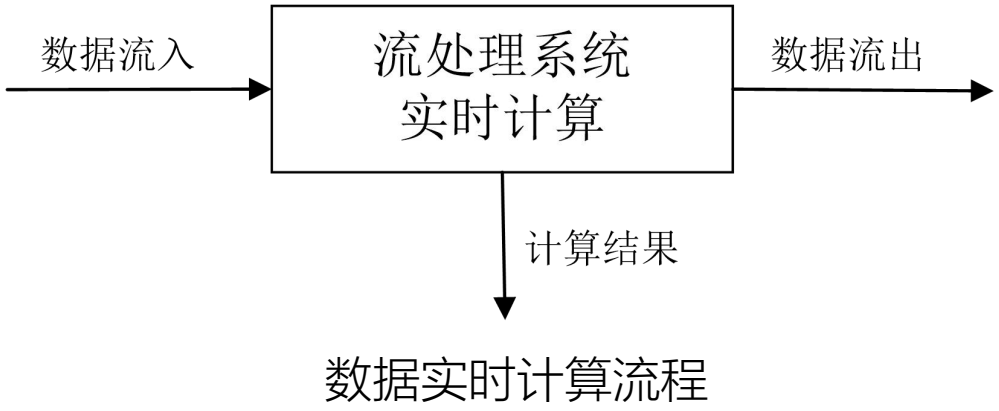
北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- 数据实时采集阶段通常采集多个数据源的海量数据，需要保证实时性、低延迟与稳定可靠
- 以日志数据为例，由于分布式集群的广泛应用，数据分散存储在不同的机器上，因此需要实时汇总来自不同机器上的日志数据
- 目前有许多互联网公司发布的开源分布式日志采集系统均可满足每秒数百MB的数据采集和传输需求，如：
  - Facebook的Scribe
  - LinkedIn的Kafka
  - 淘宝的Time Tunnel
  - 基于Hadoop的Chukwa和Flume

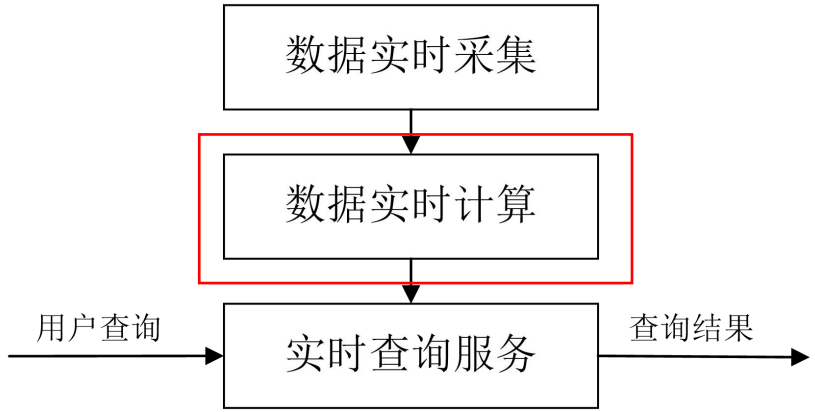


流计算处理流程

- 数据实时计算阶段对采集的数据进行实时的分析和计算，并反馈实时结果
- 经流处理系统处理后的数据，可视情况进行存储，以便之后再进行分析计算。在时效性要求较高的场景中，处理之后的数据也可以直接丢弃



数据实时计算流程

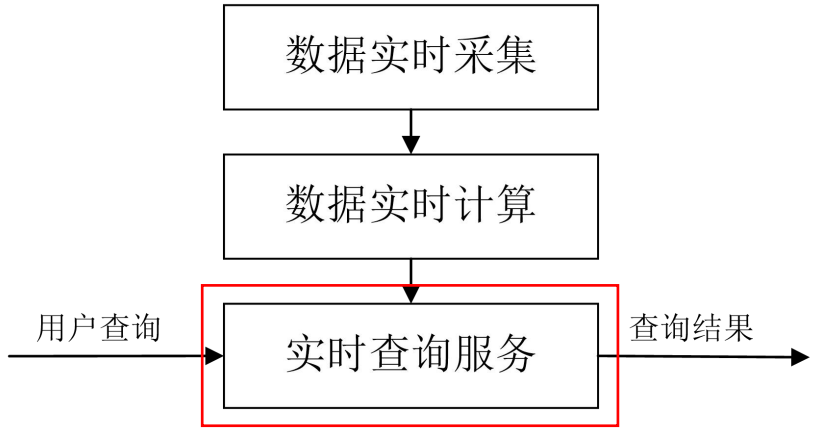


流计算处理流程

3.实时查询服务

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- 实时查询服务：经由流计算框架得出的结果可供用户进行实时查询、展示或储存
- 传统的数据处理流程，用户需要主动发出查询才能获得想要的结果
- 而在流处理流程中，实时查询服务可以不断更新结果，并将用户所需的结果实时推送给用户
- 虽然通过对传统的数据处理系统进行**定时**查询，也可以实现不断地更新结果和结果推送，但通过这样的方式获取的结果，仍然是根据过去某一时刻的数据得到的结果，与实时结果有着本质的区别



流计算处理流程

- 可见，流处理系统与传统的数据处理系统有如下不同：
  - 流处理系统处理的是实时的数据，而传统的数据处理系统处理的是预先存储好的静态数据
  - 用户通过流处理系统获取的是实时结果，而通过传统的数据处理系统，获取的是过去某一时刻的结果
  - 流处理系统无需用户主动发出查询，实时查询服务可以主动将实时结果推送给用户

大数据处理—流计算

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Spark Streaming

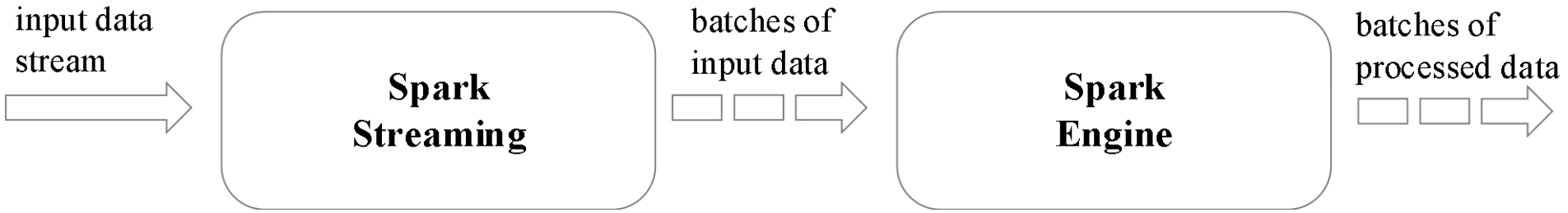
Spark Streaming设计

- Spark Streaming可整合多种输入数据源，如Kafka、Flume、HDFS，甚至是普通的TCP socket套接字
- 经处理后的数据可存储至文件系统、数据库，或显示在仪表盘里

图 Spark Streaming支持的输入、输出数据源

Spark Streaming设计

- Spark Streaming的基本原理是将实时输入数据流以时间片（秒级，通常在0.5秒~2秒之间）为单位进行拆分，然后经Spark引擎以类似批处理的方式处理每个时间片数据



Spark Streaming执行流程

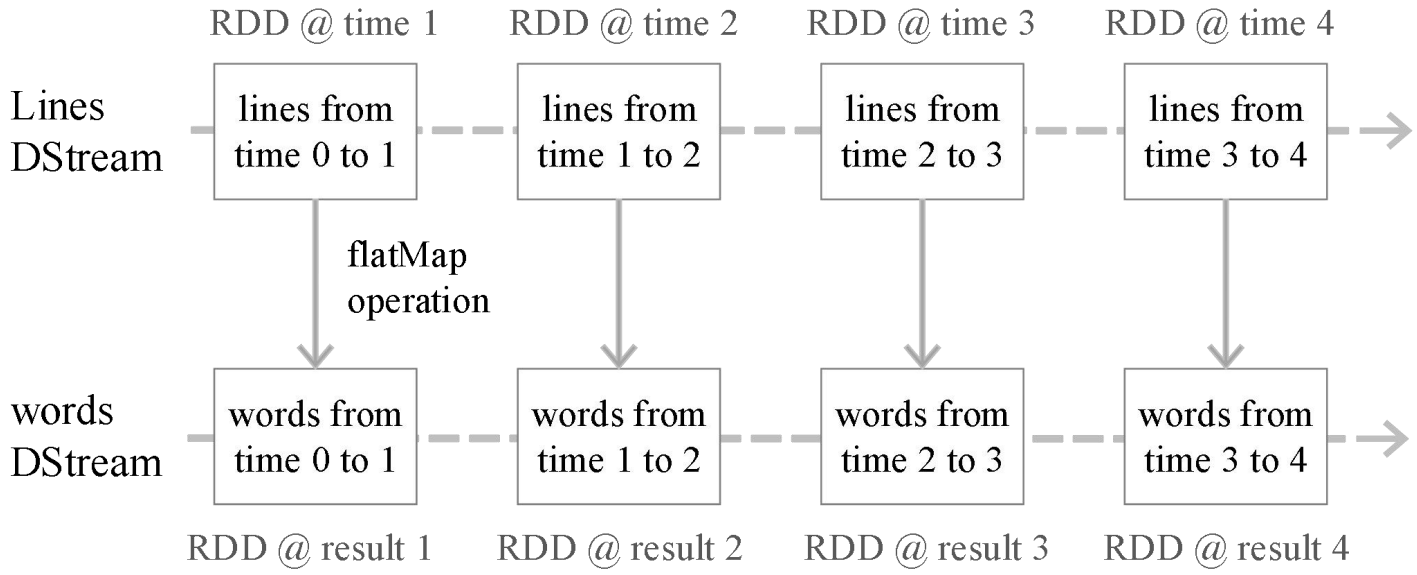
Spark Streaming设计

- Spark Streaming最主要的抽象是DStream（Discretized Stream，离散化数据流），表示连续不断的数据流
- 在内部实现上，Spark Streaming的输入数据按照时间片（如1秒）分成一段一段，每一段数据转换为Spark中的RDD，这些分段就是DStream
- 对DStream的操作都最终转变为对相应的RDD的操作

DStream举例

- 在进行单词的词频统计时，一个又一个句子会像流水一样源源不断到达，Spark Streaming会把数据流切分成一段一段，每段形成一个RDD，即RDD@time 1、RDD@time2、RDD@time 3和RDD@time 4等，每个RDD里面都包含了一些句子，这些RDD就构成了一个 DStream
- 对这个DStream执行flatMap操作时，实际上会被转换成针对每个RDD的flatMap 操作，转换得到的每个新的RDD中都包含了一些单词，这些新的RDD(即RDD@result1、RDD@result2、RDD@result 3、RDD@result 4 等) 又构成一个新的 DStream
- 整个流式计算可根据业务的需求对这些中间的结果进一步处理，或者存储到外部设备中

DStream操作示意图



Spark Streaming与Storm的对比

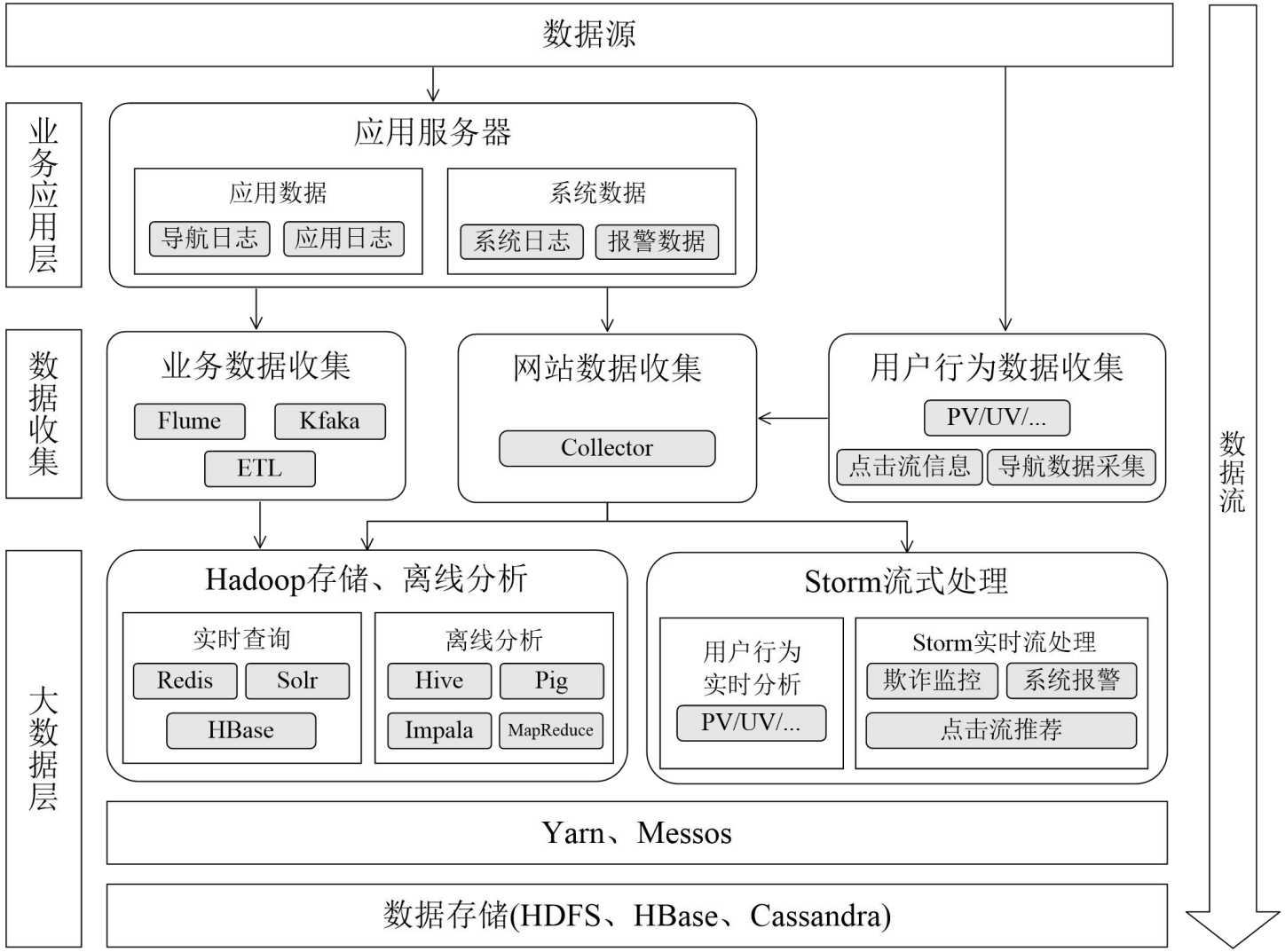
- Spark Streaming和Storm最大的区别在于，Spark Streaming无法实现毫秒级的流计算，而Storm可以实现毫秒级响应
- Spark Streaming构建在Spark上，一方面是因为Spark的低延迟执行引擎（100ms+）可以用于实时计算，另一方面，相比于Storm，RDD数据集更容易做高效的容错处理
- Spark Streaming采用的小批量处理的方式使得它可以同时兼容批量和实时数据处理的逻辑和算法，因此，方便了一些需要历史数据和实时数据联合分析的特定应用场合

从“Hadoop+Storm”架构转向Spark架构

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- 为了能同时进行批处理与流处理，企业应用中通常会采Hadoop+Storm的架构
- 在这种部署架构中，Hadoop和Storm框架部署在资源管理框架YARN (或Mesos)之上，接受统一的资源管理和调度
- 并共享底层的数据存储（HDFS、HBase等)
- Hadoop负责对批量历史数据的实时查询和离线分析，而Storm则负责对流数据的实时处理

采用Hadoop+Storm部署方式的一个案例

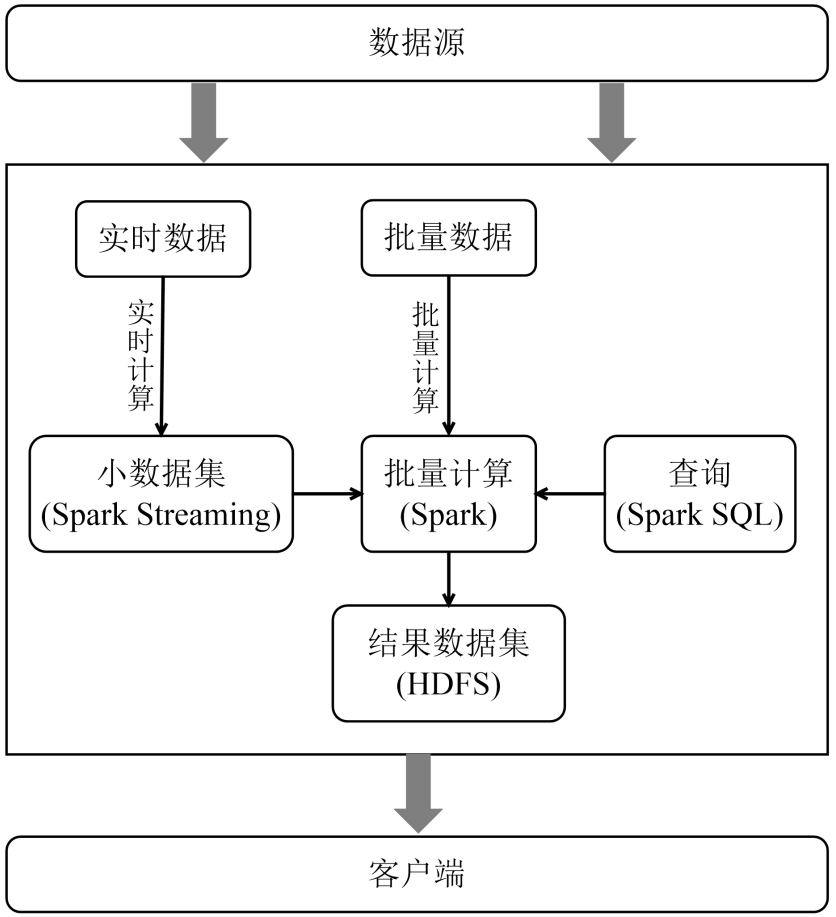


从“Hadoop+Storm”架构转向Spark架构

采用Spark架构具有如下优点：

- 实现一键式安装和配置、线程级别的任务监控和告警
- 降低硬件集群、软件维护、任务监控和应用开发的难度
- 便于做成统一的硬件、计算平台资源池

用Spark架构满足批处理和流处理需求

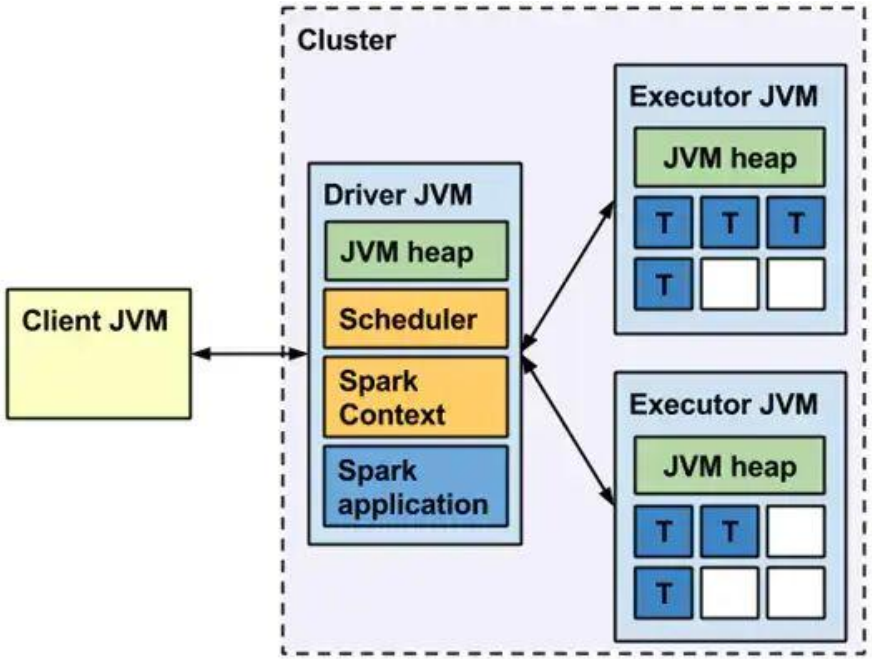


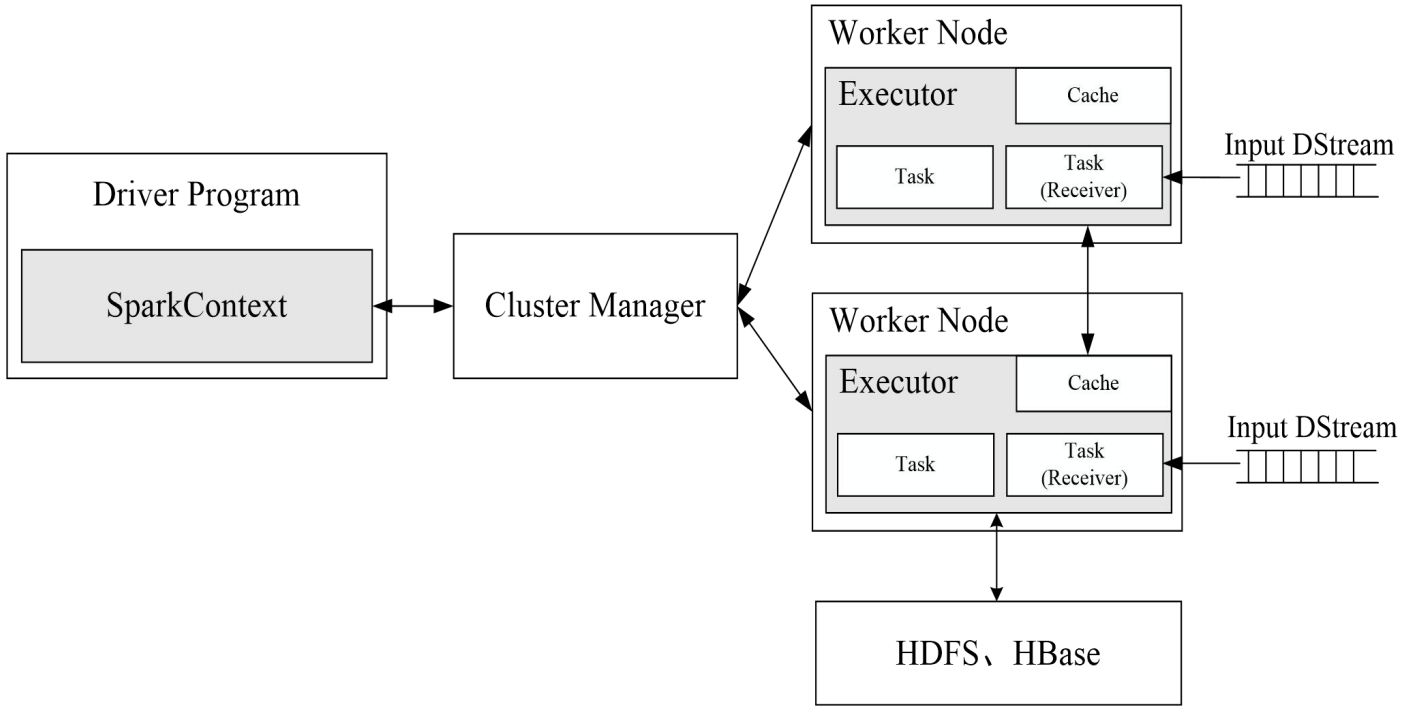
注：对于需要毫秒级实时响应的场景，还是需要采用其他的流计算框架（如Storm、Flink ）

DStream操作流程

Spark Streaming工作机制

- 在Spark Streaming中，会有一个组件Receiver，作为一个长期运行的task跑在一个Executor上
- 每个Receiver都会负责一个input DStream（比如从文件中读取数据的文件流，比如套接字流，或者从Kafka中读取的一个输入流等等）
- Spark Streaming通过input DStream与外部数据源进行连接，读取相关数据





Spark Streaming程序的基本步骤

编写Spark Streaming程序的基本步骤是：

- 1.通过创建输入DStream来定义输入源
- 2.通过对DStream应用转换操作和输出操作来定义流计算
- 3.用streamingContext.start()来开始接收数据和处理流程
- 4.通过streamingContext.awaitTermination()方法来等待处理结束（手动结束或因为错误而结束）
- 5.可以通过streamingContext.stop()来手动结束流计算进程

创建StreamingContext对象

- 如果要运行一个Spark Streaming程序，就需要首先生成一个StreamingContext对象，它是Spark Streaming程序的主入口
  1. 登录Linux系统后，启动spark-shell
  2. 进入spark-shell后，就已获得了一个默认的SparkContext对象，也就是sc
  3. 因此，可以采用如下方式来创建StreamingContext对象：

```
scala> import org.apache.spark.streaming._
scala> val ssc = new StreamingContext(sc, Seconds(1))
```

创建StreamingContext对象

- 如果是编写一个独立的Spark Streaming程序，而不是在spark-shell中运行，则需要通过如下方式创建StreamingContext对象：
- 先创建SparkConf对象

```
import org.apache.spark._
import org.apache.spark.streaming._
val conf = new
SparkConf().setAppName("TestDStream").setMaster("local[2]")
val ssc = new StreamingContext(conf, Seconds(1))
```

大数据处理—流计算

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

数据源管理

输入源

基本输入源

- 文件流
- 套接字流
- RDD队列流

高级输入源

- Kafka

文件流

1.在spark-shell中创建文件流

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

[实战目标]：编写一个Spark Streaming 程序，一直对文件系统中的某个目录进行监听，一旦发现有新的文件生成，Spark Streaming就会自动把文件内容读取过来，使用用户自定义的处理逻辑进行处理

- 首先在Linux系统中打开第一个终端（这里称为“数据源终端”），创建一个logfile目录

```
$ cd /usr/local/spark/mycode
$ mkdir streaming
$ cd streaming
$ mkdir logfile
$ cd logfile
```

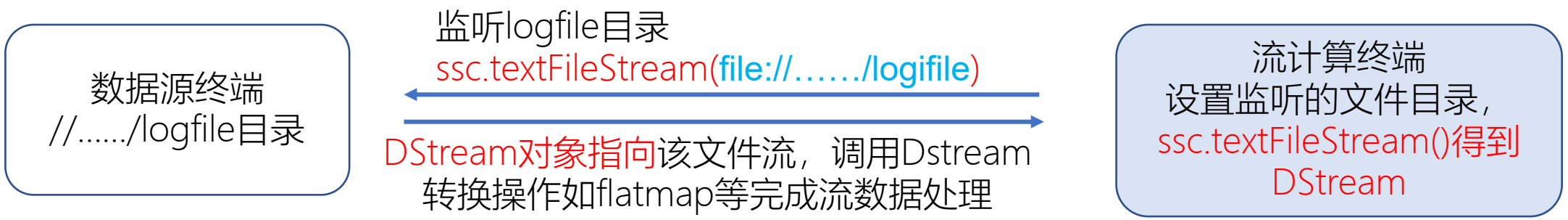
数据源终端
//...../logfile目录

文件流

1.在spark-shell中创建文件流

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- 在Linux系统中打开另外一个终端窗口（称为“流计算终端”），启动进入spark-shell
- ssc.textFileStream()语句用于创建一个“文件流类型的输入源，构建出DStream




```
scala> import org.apache.spark.streaming._
scala> val ssc = new StreamingContext(sc, Seconds(20))
scala> val lines =
ssc.textFileStream("file:///usr/local/spark/mycode/streaming/logfile")
```

文件流

1.在spark-shell中创建文件流

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

```
scala> val words = lines.flatMap(_.split(" "))
scala> val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
scala> wordCounts.print()
scala> ssc.start()
```

- 接下来的 lines.flatMap()、words.map(和wordCounts.print()是流计算处理过程，负责对文件流中发送过来的文件内容进行词频统计
- ssc.start()语句用于启动流计算过程，回车后Spark Streaming就开始进行循环监听
- 可以使用Ctrl+C手动停止这个流计算过程

文件流

1.在spark-shell中创建文件流

数据源终端
//...../logfile目录

监听logfile目录
ssc.textFileStream(file://...../logifile)

DStream对象指向该文件流，调用Dstream转换操作如flatmap等完成流数据处理

流计算终端
设置监听的文件目录，
ssc.textFileStream()得到DStream

1. 切换到第一个Linux终端（数据源终端）在".....streaming/logfile"目录下新建一个log.txt文件，在文件中输入一些英文语句后保存退出文本编辑器
2. 切换到第二个Linux终端（流计算终端），就可以在监听窗口中显示词频统计结果

进入循环监听状态，屏幕上会显示一堆的信息

```
//这里省略若干屏幕信息
-------------------------------------------
Time: 1479431100000 ms
-------------------------------------------
//这里省略若干屏幕信息
-------------------------------------------
Time: 1479431120000 ms
-------------------------------------------
//这里省略若干屏幕信息
-------------------------------------------
Time: 1479431140000 ms
-------------------------------------------
```

文件流

2. 采用独立应用程序方式创建文件流

- 首先，创建代码目录和代码文件TestStreaming.scala
- 在Linux系统中，关闭之前打开的所有Linux 终端，重新打开一个终端（“流计算终端”），执行如下命令：

```
$ cd /usr/local/spark/mycode
$ mkdir streaming
$ cd streaming
$ mkdir -p src/main/scala
$ cd src/main/scala
$ vim TestStreaming.scala
```

文件流

- 用vim编辑器新建一个TestStreaming.scala代码文件，请在里面输入以下代码：

```
import org.apache.spark._
import org.apache.spark.streaming._
object WordCountStreaming {
  def main(args: Array[String]) {
    val sparkConf = new
SparkConf().setAppName("WordCountStreaming").setMaster("local[2]")
//设置为本地运行模式，2个线程，一个监听，另一个处理数据
    val ssc = new StreamingContext(sparkConf, Seconds(2))// 时间间隔为2秒
    val lines = ssc.textFileStream("file:///usr/local/spark/mycode/streaming/logfile")
 //这里采用本地文件，当然也可以采用HDFS文件
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
    wordCounts.print()
    ssc.start()
    ssc.awaitTermination()
  }
}
```

文件流

- 在/usr/local/spark/mycode/streaming 目录下创建simple.bat文件

```
$ cd /usr/local/spark/mycode/streaming
$ vim simple.sbt
```

在simple.sbt文件中输入以下代码：

```
name := "Simple Project"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" % "spark-streaming_2.11" % "2.1.0"
```

执行sbt打包编译的命令如下：

```
$ cd /usr/local/spark/mycode/streaming
$ /usr/local/sbt/sbt package
```

注：SBT(Simple Build Tool)是Scala的项目构建工具，拥有依赖管理，构建过程管理和打包等功能

文件流

- 打包成功以后，就可以输入以下命令启动这个程序：

```
$ cd /usr/local/spark/mycode/streaming
$ /usr/local/spark/bin/spark-submit --class “WordCountStreaming”
/usr/local/spark/mycode/streaming/target/scala-2.11/simple-
project_2.11-1.0.jar
```

- 执行上面命令后，就进入了监听状态（称为“监听窗口”）
- 切换到另外一个Shell窗口，在"/usr/local/spark/mycode/streaming/logfile"目录下再新建一个log2.txt文件，文件里面随便输入一些单词，保存好文件退出vim编辑器
- 再次切换回“监听窗口”，等待20秒以后，按键盘Ctrl+C或者Ctrl+D停止监听程序，就可以看到监听窗口的屏幕上会打印出单词统计信息

DStream数据源管理——套接字流

套接字流

1.Socket工作原理

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- Spark Streaming可以通过Socket端口监听并接收数据，然后进行相应处理

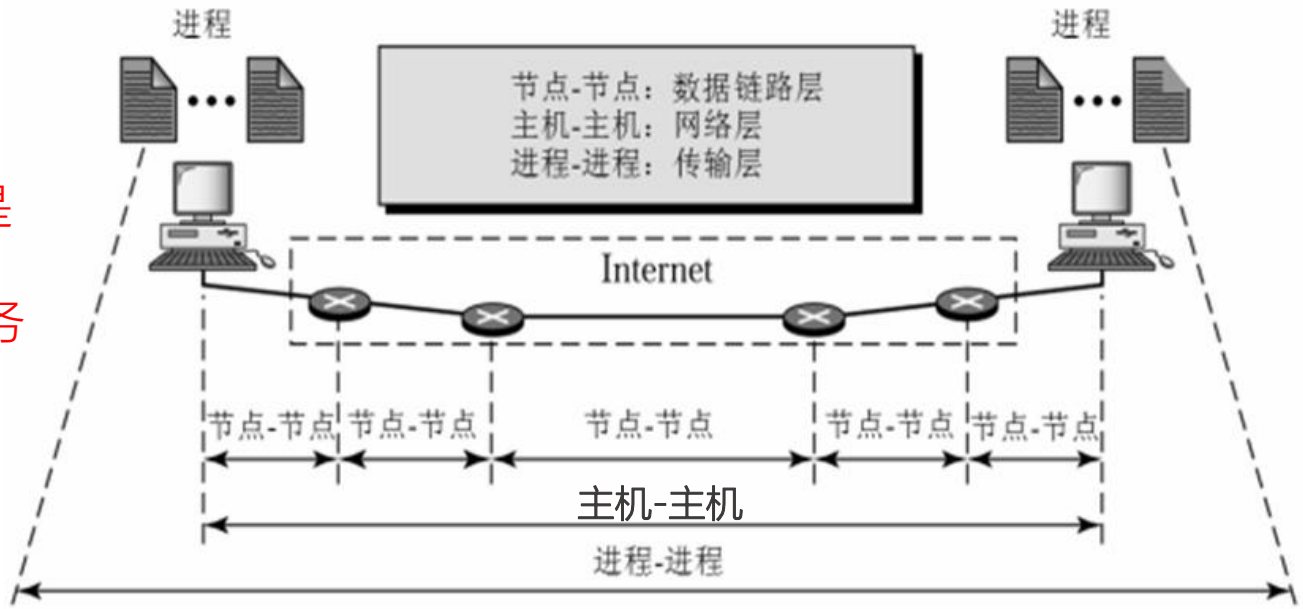
【知识点复习】Java网络编程可以在三个层次上进行：

1. URL层次，即最高级层次，基于应用层通信协议，利用URL直接进行Internet上的资源访问和数据传输
2. Socket层次,即传统网络编程经常采用的流式套接字方式，通过在Client/Server（客户机/服务器）结构的应用程序之间建立Socket套接字连接，然后在连接之上进行数据通信
3. Datagram数据包层次，即最低级层次，采用一种无连接的数据包套接字传输方法，是用户数据报 (UDP) 协议的通信方式

套接字流-补充-Socket通信原理

- TCP通过流控（滑动窗口机制）、顺序编码、应答和计时器， 保证将数据按序、正确地从某个主机中的一个进程传递到另一台主机的一个进程
- TCP在一个不可靠的互联网络中为应用程序提供可靠的端点间的字节流服务
- 所有TCP 连接都是全双工和点对点的
  - 全双工是指数据可在连接的两个方向上同时传输
  - **点对点意味 着每条TCP连接只有两个端点**

TCP协议是端-端的数据传输服务



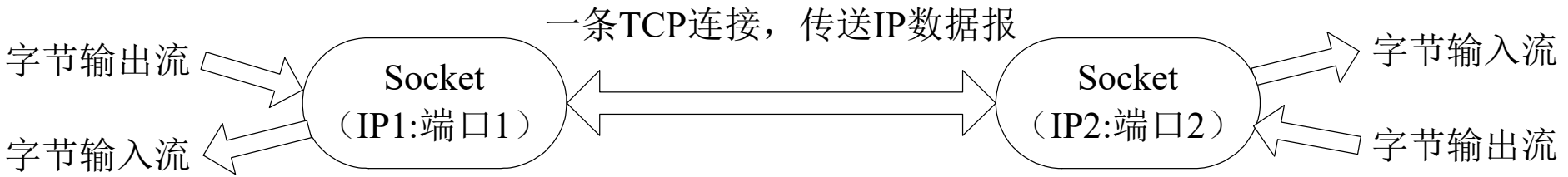
IP 协议并不保证一定把数据包送达目标主机，在发送过程中，会因为数据包结束生命周期，或者找不到路由而丢弃数据包

套接字流-补充-Socket通信原理

- 一条“TCP连接”连接的两端是Internet上分别在两台主机中运行的**两个进程**
- 一个是 发送进程，一个是接收进程
- 每个进程需要用（**一个IP地址+一个端口号**）唯一确定

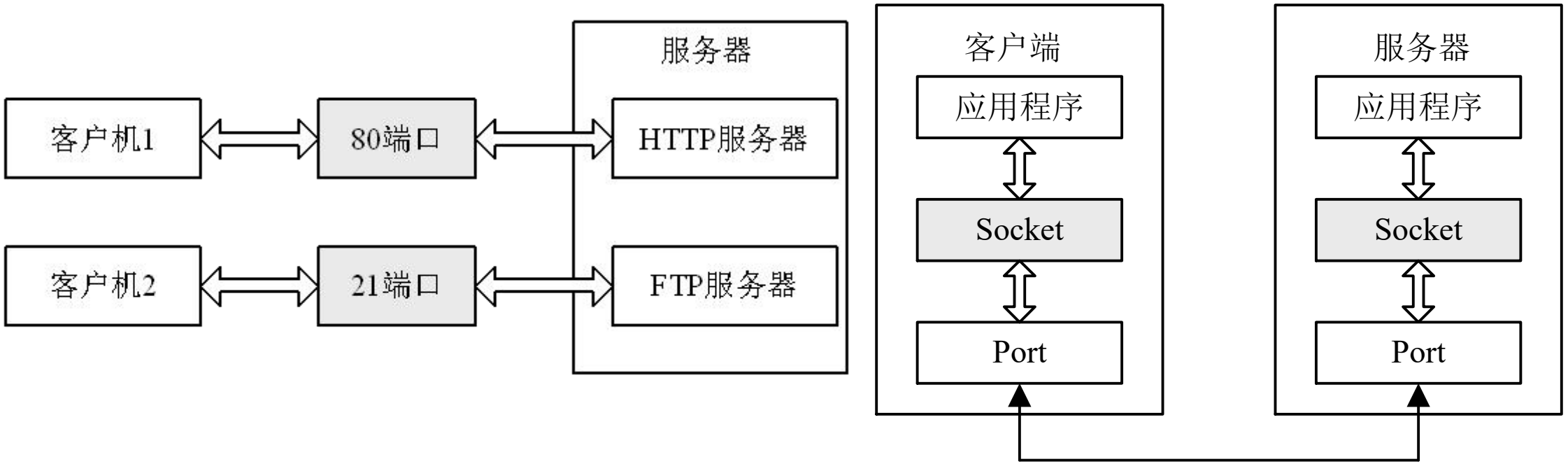
套接字(Socket)

- 一 条TCP连接包含一个源端口号和一个宿端口号，分别用来识别**发送进程**和**接收进程**
- 一个48位的套接字(Socket)，用来确定一个通信的端点： 取值=一个端口号+所在的主机IP地址
- 一对套接字就可以在互联网络中唯一标识一条TCP连接



套接字流-补充-Socket的理解

- 网络程序中套接字（Socket）用于将应用程序与端口连接起来。
- 套接字是一个假想的连接装置，就像插插头的设备“插座”，用于连接电器与电线，如下图所示。Java将套接字抽象化为类，程序设计者只需创建Socket类对象，即可使用套接字。



套接字流-补充-Java的Socket通信实现

Socket通信流程说明如下:

1. 服务端创建一个ServerSocket对象，指定端口号
2. ServerSocket对象等候客户端的连接请求
3. 服务端接收到客户端的连接请求，**建立一条TCP连接**，再创建一个服务端Socket对象与客户端的Socket对象进行通信
4. 服务端和客户端分别建立**字节输入/输出流**，进行数据传输准备。
5. 服务端和客户端通过各自的字节输入流获得对方发来的数据，通过字节输出流向对方发送数据
6. 一方决定结束通信，向对方发送结束信息，另一方接收到结束信息后，双方分别关闭 各自的TCP连接
7. ServerSocket对象停止等候客户端的连接请求

套接字流-补充-Java的Socket通信实现

- Java在包java.net中提供了两个类Socket和ServerSocket
  - 用于创建ServerSocket对象
  - 用于创建服务端Socket对象与客户端的Socket对象

```
public class ServerSocket extends Object
{   public ServerSocket(int port) throws  IOException //构造方法，指定端口号
   //连接队列的最大长度是50，当连接队列已满，又有客户端发起连接请求时，服务器将拒绝该连接请求
   //连接队列是指已完成TCP三次握手但还没有被accept()取走的TCP连接
    public ServerSocket(int port,int backlog)throws IOException
    //构造方法，backlog（指定队列最大长度）,创建服务器套接字并将其绑定到指定的本地端口号
    public ServerSocket(int port,int backlog,InetAddress bindAddr)throws IOException
      //构造方法，使用指定的端口、侦听backlog和要绑定到的本地 IP 地址创建服务器套接字
       //bindAddr表示要将服务器绑定到的InetAddress地址
      //InetAddress类 表示Internet上一台计算机的主机名和IP地址
    public Socket accept() throws IOException
      //等待接收客户端的连接请求，连接成功后返回一个已连接的Socket对象
    public void close() throws IOException      //停止等候客户端的连接请求
}
```

套接字流-补充-Java的Socket通信实现


```
public class Socket extends Object
{
    public Socket(String host, int port) throws UnknownHostException, IOException
                                        //构造Socket对象方法，指定主机名和端口号
    public Socket(InetAddress address, int port) throws UnknownHostException,
   IOException//构造方法，指定IP和端口号
    public Socket(String host,int port,InetAddress localAddr,int localPort)throws
   IOException//构造方法，指定远程主机名，远程端口号，本地地址，本地端口
   public Socket(InetAddress address,int port,InetAddress localAddr,int localPort) throws
   IOException//构造方法，指定远程IP，远程端口号，本地地址，本地端口

   public InputStream getInputStream() throws IOException              //返回TCP连接提供的
   字节输入流
    public OutputStream getOutputStream() throws IOException          //返回TCP连接提供的
   字节输出流
    public synchronized void close() throws IOException        //关闭TCP连接
}
```

```
import java.io.*;      import java.net.*;
    public class SimpleSocketClient {
    public static void main(String[] args) {
    Socket socket = null;
    InputStream is = null;
    OutputStream os = null;
    String serverIP = "127.0.0.1";     //服务器端IP地址
    int port = 10000;       //服务器端端口号
    String data = "Hello";  //发送内容
    try {          socket = new Socket(serverIP,port);  //建立连接
    os = socket.getOutputStream();  //发送数据
    os.write(data.getBytes());
    is = socket.getInputStream();  //接收数据
    byte[] b = new byte[1024];
    int n = is.read(b);
    System.out.println("服务器反馈：" + new String(b,0,n));  //输出反馈数据
    } catch (Exception e) {          e.printStackTrace(); //打印异常信息
    }finally{
       try {      is.close();  //关闭流和连接
                  os.close();
                  socket.close();
    } catch (Exception e2) {}      }      }      }
```

```
/**
 * 简单的Socket客户端
 * 功能为：发送字符串"Hello"到服务器端，并打印出服务器端的反馈
 */
```

套接字流-补充-Java的Socket通信实现

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

TCP Socket网络编程-Socket类的常用方法

方　法	功　能
InetAddress getLocalAddress()	返回本地Socket中的IP的InetAddress对象
int getLocalPort()	返回本地Socket中的端口号
InetAddress getInetAddress()	返回对方Socket中的IP地址
int getPort()	返回对方Socket中的端口号
void close() throws IOException	关闭Socket，释放资源
InputStream getInputStream()throws IOException	获取与Socket相关联的字节输入流，用于从Socket中读数据
OutputStream getOutputStream()throws IOException	获取与Socket相关联的字节输出流，用于向Socket中写数据

套接字流-补充-Java的Socket通信实现

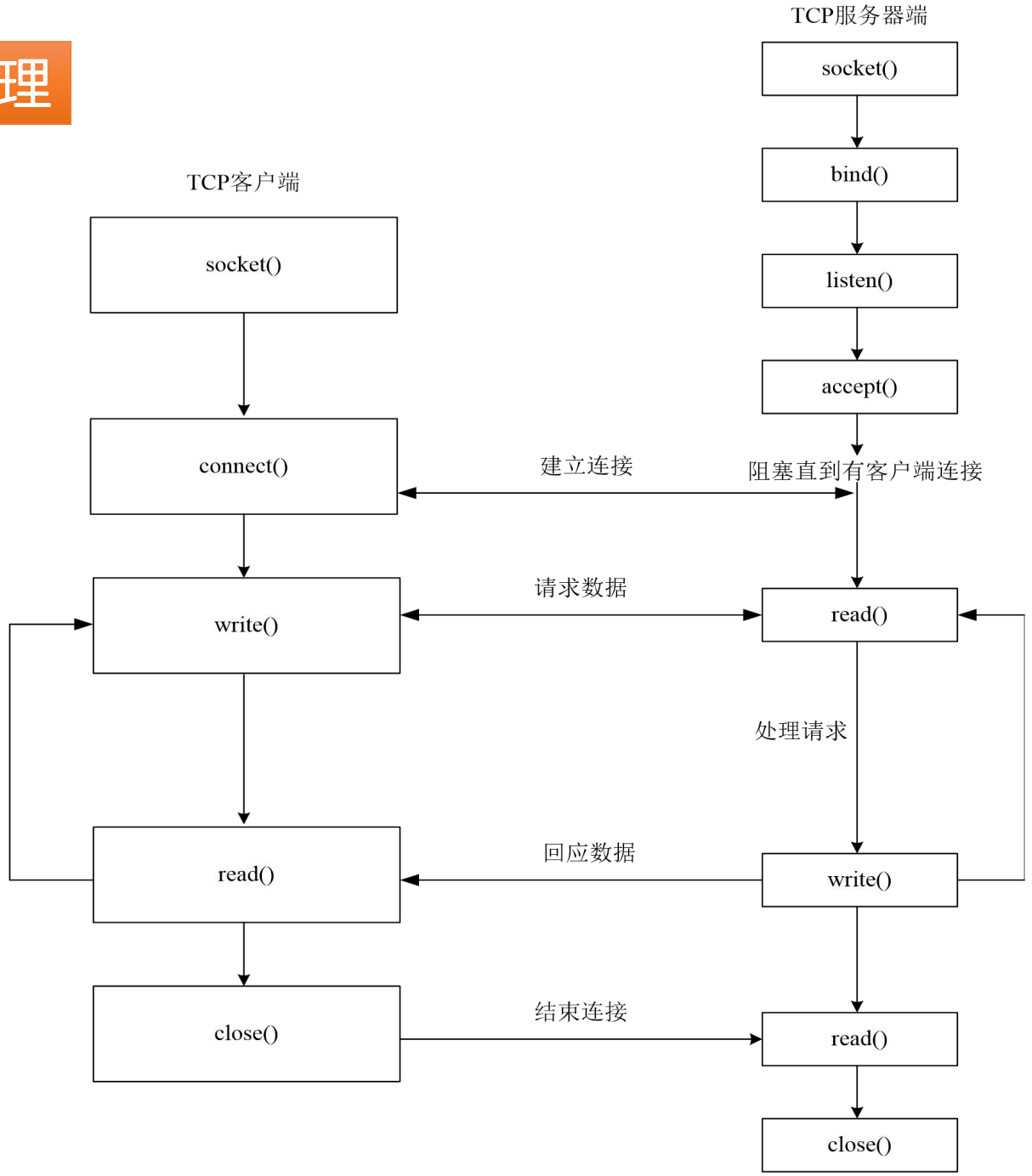
服务端代码

```
ServerSocket server=null; //声明服务端ServerSocket对象
        try {   server=new ServerSocket(4700);
                  //创建一个ServerSocket在端口4700监听客户请求
        }catch(IOException e){System.out.println("can not listen to :"+e);}

        Socket socket=null; //声明服务端Socket对象
        try {
                socket=server.accept();
//服务端Socket对象接收客户端Socket对象的连接请求
//accept()是一个阻塞的方法，一旦有客户请求，它就会返回一个Socket对象用于同、客户进行交互

        }catch(IOException e){System.out.println("Error:"+e);}
```

套接字流

1.Socket工作原理

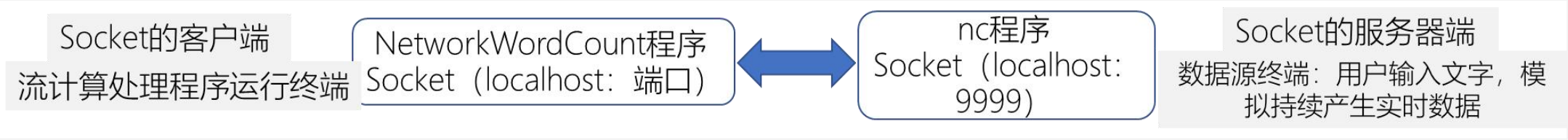
- 在套接字流作为数据源的应用场景中，Spark Streaming程序就是Socket通信的客户端
- 它通过Socket方式请求数据，获取数据以后启动流计算过程进行处理



套接字流

2.使用套接字流作为数据源

首先创建代码目录和代码文件NetworkWordCount.scala，该程序启动在流计算终端，作为Socket客户端，用于去连接参数所指定的服务器端，并读取数据流



套接字流

2.使用套接字流作为数据源

首先创建代码目录和代码文件NetworkWordCount.scala，该程序启动在流计算终端，作为Socket客户端，用于去连接参数所指定的服务器端，并读取数据流

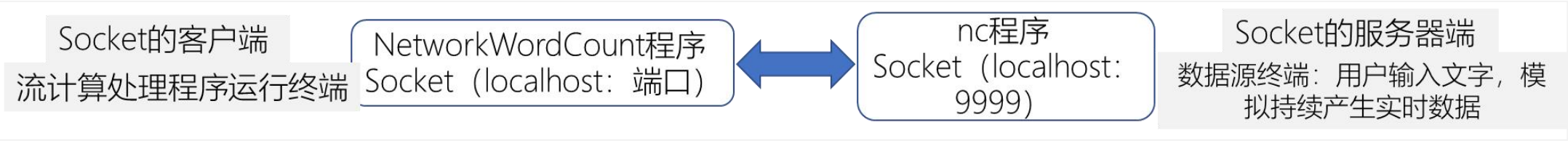
- 新建一个Linux终端（作为"流计算终端"），在终端里执行如下命令：

```
$ cd /usr/local/spark/mycode
$ mkdir streaming #如果已经存在该目录，则不用创建
$ mkdir -p /src/main/scala #如果已经存在该目录，则不用创建
$ cd /usr/local/spark/mycode/streaming/src/main/scala
$ vim NetworkWordCount.scala
```

请在NetworkWordCount.scala文件中输入如下内容：

```
package org.apache.spark.examples.streaming
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.storage.StorageLevel
```

剩余代码在下一页



套接字流的创建 ssc.socketTextStream()

- ssc.socketTextStream()用于**创建一个"套接字流"类型的输入流**，有3个输入参数
- args(0)提供了主机地址，
- args(1).toInt提供了通信端口号
- Socket客户端使用该主机地址和端口号与服务器端建立通信
- StorageLevel.MEMORY_AND_DISK_SER表示Spark Streaming作为客户端，在接收到来自服务器端的数据以后，采用的存储方式为MEMORY_AND_DISK_SER，使用内存和磁盘作为存储介质

```
object NetworkWordCount {
  def main(args: Array[String]) {
    if (args.length < 2) {
      System.err.println("Usage: NetworkWordCount <hostname> <port>")
      System.exit(1)
    }
    StreamingExamples.setStreamingLogLevels() //设置log4j日志级别
    val sparkConf = new
SparkConf().setAppName("NetworkWordCount").setMaster("local[2]")
    val ssc = new StreamingContext(sparkConf, Seconds(1))
    val lines = ssc.socketTextStream(args(0), args(1).toInt,
StorageLevel.MEMORY_AND_DISK_SER)
//主机地址；端口号；数据的存储方式
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
    wordCounts.print()
    ssc.start()
    ssc.awaitTermination()
  }
}
```

套接字流-日志

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- 主要用于在程序运行中，wordCounts.print()语句的打印信息能够得到正确显示
- 在相同目录下再新建另外一个代码文件StreamingExamples.scala
- 定义了StreamingExamples类，继承了Logging类
- 定义了setStreamingLogLevels()方法，将log4j日志的级别设置为Level.WARN

```
package org.apache.spark.examples.streaming
import org.apache.spark.internal.Logging
import org.apache.log4j.{Level, Logger}
/** Utility functions for Spark Streaming examples. */
object StreamingExamples extends Logging {
  /** Set reasonable logging levels for streaming if the user has not configured log4j. */
  def setStreamingLogLevels() {
    val log4jInitialized = Logger.getRootLogger.getAllAppenders.hasMoreElements
    if (!log4jInitialized) {
      // We first log something to initialize Spark's default logging, then we override the
      // logging level.
      logInfo("Setting log level to [WARN] for streaming example." +
        " To override add a custom log4j.properties to the classpath.")
      Logger.getRootLogger.setLevel(Level.WARN)
    }
  }
}
```

套接字流-编译打包读取套接字流的客户端程序

- 使用sbt工具对代码进行编译打包

创建sbt文simple.sbt

```
$ cd /usr/local/spark/mycode/streaming/
$ vim simple.sbt
```

sbt文件内容

```
name := "Simple Project"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" % "spark-streaming_2.11" % "2.1.0"
```

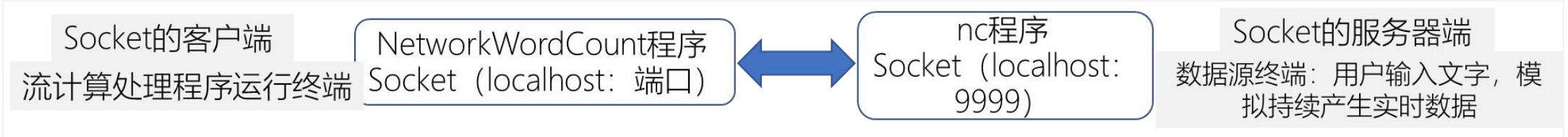
进行编译打包

```
$ cd /usr/local/spark/mycode/streaming
$ /usr/local/sbt/sbt package
```

套接字流-启动客户端程序

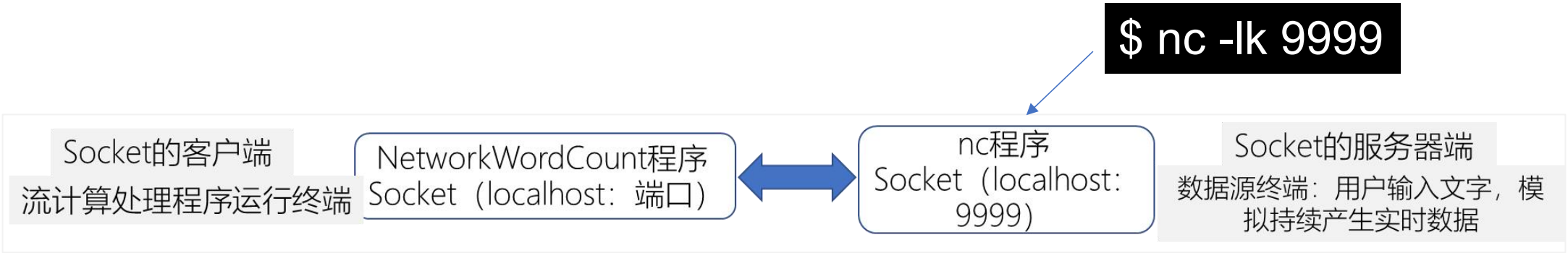
打包成功以后，输入命令启动程序：
设置所连接的服务器地址和IP localhost 9999

```
$ cd /usr/local/spark/mycode/streaming
$ /usr/local/spark/bin/spark-submit --class "
org.apache.spark.examples.streaming.NetworkWordCount"
/usr/local/spark/mycode/streaming/target/scala-2.11/simple-
project_2.11-1.0.jar localhost 9999
```



套接字流-设置一个服务器端，发送数据

- 新打开一个Linux窗口作为“数据源终端”，使用nc网络工具（在这里作为Socket服务器端），启动监听并接受连接，监听的端口号是9999，-l启动本地监听，-k让服务端保持连接，不断开
- 由于已经在上一页步骤里，“流计算窗口”启动运行了NetworkWordCount程序，该程序作为socket的客户端程序，会向localhost 9999端口的发送连接请求
- nc程序会监听到本地localhost 9999端口来自客户端程序NetworkWordCount的连接请求，于是就会建立双方的连接通道

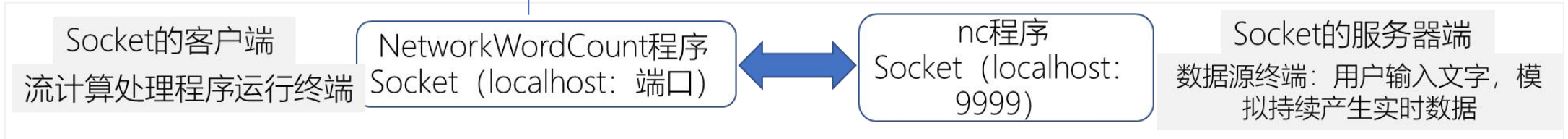


套接字流-设置一个服务器端，发送数据

- 连接通道建立好后，在“数据源终端”窗口输入的内容，由nc程序发送给“流计算终端”的NetworkWordCount程序处理
- 可以在nc窗口中随意输入一些单词，“流计算终端”的程序就会自动获得单词数据流信息，并在窗口每隔1秒就会打印出词频统计信息，大概会在屏幕上出现类似右侧的结果：

在流计算终端的输出效果

```
-------------------------------------------
Time: 1479431100000 ms
-------------------------------------------
(hello,1)
(world,1)
-------------------------------------------
Time: 1479431120000 ms
-------------------------------------------
(hadoop,1)
-------------------------------------------
Time: 1479431140000 ms
-------------------------------------------
(spark,1) -
```



套接字流

3. 使用Socket编程实现自定义数据源

- 上面的示例，是采用了nc程序作为数据源
- 下面再进一步，把数据源的产生方式修改一下，不使用nc程序，而是采用自己编写的程序产生Socket数据源
- 新建一个Linux终端（作为数据源终端），在该终端里执行如下命令新建一个代码文件DataSourceSocket.scala

```
$ cd /usr/local/spark/mycode/streaming/src/main/scala
$ vim DataSourceSocket.scala
```

```
package org.apache.spark.examples.streaming
Import  java.io.{PrintWriter}
Import  java.net.ServerSocket
Import  scala.io.Source
```

剩余代码见下一页

Socket的客户端

流计算处理程序运行终端

NetworkWordCount程序
Socket（localhost：端口）

DataSourceSocket程序
Socket（localhost：9999）

Socket的服务器端

数据源终端：用户输入文字，模拟持续产生实时数据

套接字流-编程实现自定义数据源

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

3. 使用Socket编程实现自定义数据源

- 从一个文件中读取内容，把文件的每一行作为一个字符串，每次随机选择文件中的一行，源源不断发送给客户端NetworkWordCount程序
- DataSourceSocket程序在运行时，需要为该程序提供3个参数，即<filename> <port> <millisecond>，其中<filename> 表示作为数据源头的文件的路径；<port>表示Socket通信的端口号；<millisecond>表示Socket服务器端每隔多长时间向客户端发送一次数据

```
object DataSourceSocket {
  def index(length: Int) = {
    val rdm = new java.util.Random
    rdm.nextInt(length) // 生成一个 0到length的随机数
  } //随机获取0-length 之间的一个数
  def main(args: Array[String]) {
    if (args.length != 3) {
      System.err.println("Usage: <filename> <port> <millisecond>")
      System.exit(1)
    }
    val fileName = args(0)
    val lines = Source.fromFile(fileName).getLines.toList
//读取文件的每一行数据到list里面
    val rowCount = lines.length //获得数据行数
    val listener = new ServerSocket(args(1).toInt)
//服务器端使用ServerSocket类创建一个ServerSocket对象listener
```

剩余代码见下一页

套接字流-编程实现自定义数据源

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- ServerSocket对象listener的accept方法，接收客户端的连接请求，成功后创建Socket对象
- new Thread启动一个新的线程对象，并重写线程对象的run方法
- run方法中使用socket的getOutputStream()方法获得输出流，并用该输出流创建了标准输入流PrintWriter对象out
- 每隔指定的间隔时间，out输出流对象向网络写一行文件file里的数据
- 完成输出流后，关闭socket对象

```
    while (true) {
      val socket = listener.accept()
      new Thread() {
        override def run = {
          println("Got client connected from: " + socket.getInetAddress)
          val out = new PrintWriter(socket.getOutputStream(), true)
          while (true) {
            Thread.sleep(args(2).toLong)
            val content = lines(index(rowCount))
            println(content)
            out.write(content + '\n')
            out.flush()
          }
          socket.close()
        }
      }.start()// Thread类的start方法，启动线程
    }
  }
}
```

套接字流-编译启动DataSourceSocket程序

- 执行sbt打包编译：

```
$ cd /usr/local/spark/mycode/streaming
$ /usr/local/sbt/sbt package
```

- DataSourceSocket程序需要把一个文本文件作为输入参数，所以，在启动这个程序之前，需要首先创建一个文本文件word.txt并随便输入几行内容：

  /usr/local/spark/mycode/streaming/word.txt

- 启动DataSourceSocket程序，在9999端口进行监听来自客户端的连接：

```
$ /usr/local/spark/bin/spark-submit   \
> --class "org.apache.spark.examples.streaming.DataSourceSocket"   \
>/usr/local/spark/mycode/streaming/target/scala-2.11/simple-project_2.11-1.0.jar  \
> /usr/local/spark/mycode/streaming/word.txt  9999 1000
```

- 每隔1000毫秒（1秒）向客户端发送word.txt的一行内容

套接字流-启动流计算程序连接socket数据源

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- 在Linux流计算终端，启动NetworkWordCount程序，作为Socket客户端去连接9999端口的服务器：

```
$ /usr/local/spark/bin/spark-submit --class
"org.apache.spark.examples.streaming.NetworkWordCount"
/usr/local/spark/mycode/streaming/target/scala-2.11/simple-project_2.11-1.0.jar localhost 9999
```

- 启动成功后， NetworkWordCount程序会不断的去收取来自服务器端发来的数据，并进行自定义程序的处理（本示例中是对数据进行词频统计），并在屏幕上不断打印出词频统计信息

大数据处理—流计算

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

DStream数据源管理
——RDD队列流

RDD队列流

- 在编写Spark Streaming应用程序的时候，可以调用StreamingContext的queueStream()创建基于RDD队列的DStream
- 如val queueStream = ssc.queueStream(rddQueue)
- queueStream 的入口参数是一个RDD队列
- 新建一个TestRDDQueueStream.scala代码文件，功能是：每隔1秒创建一个RDD，Streaming每隔2秒就对数据进行处理

```
package org.apache.spark.examples.streaming
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.{Seconds,
StreamingContext}
```

剩余代码在下一页

RDD队列流

- ssc.queueStream(rddQueue)语句创建了一个RDD队列流类型的数据源
- sparkStreaming会每隔2秒从rddQueue这个队列中取出数据（即若干个RDD）进行处理
- queueStream.map(r => (r % 10, 1))语句会把其中每个RDD元素进行转换，得到每个数的余数，处理为（余数,1）
- reduceByKey(_ + _)统计每个余数出现的次数
- ssc.start()启动流计算
- For循环不断的向rddQueue中加入新生成的RDD
- ssc.sparkContext.makeRDD(1 to 100,2)的功能是创建一个RDD这个RDD被分为2个分区，RDD包含1,2,3.....99,100

```
object QueueStream {
  def main(args: Array[String]) {
    val sparkConf = new
SparkConf().setAppName("TestRDDQueue").setMaster("local[2]")
    val ssc = new StreamingContext(sparkConf, Seconds(2))
    // 创建StreamingContext对象ssc每隔2秒就对数据进行处理
    val rddQueue =new
scala.collection.mutable.SynchronizedQueue[RDD[Int]]()
    val queueStream = ssc.queueStream(rddQueue)
//ssc调用queueStream方法，用RDD队列创建 RDD队列流
    val mappedStream = queueStream.map(r => (r % 10, 1))
    val reducedStream = mappedStream.reduceByKey(_ + _)
    reducedStream.print()
    ssc.start()
    for (i <- 1 to 10){
        rddQueue += ssc.sparkContext.makeRDD(1 to 100,2)
        Thread.sleep(1000)//每隔1秒执行一次循环，创建一个RDD
    }//通过for循环10次，连续创建RDD对象，形成一个RDD队列
    ssc.stop()
  }
}
```

剩余代码见下一页

RDD队列流

- sbt打包成功后，执行下面命令运行程序：

```
$ cd /usr/local/spark/mycode/streaming
$ /usr/local/spark/bin/spark-submit   \
>--class "org.apache.spark.examples.streaming.QueueStream"   \
>/usr/local/spark/mycode/streaming/target/scala-2.11/simple-project_2.11-1.0.jar
```

执行上面命令以后，程序就开始运行，就可以看到类似下面的结果：

```
-------------------------------------------
Time: 1479522100000 ms        //第i个输出
-------------------------------------------
(4,10)
(0,10)
(6,10)
(8,10)
(2,10)
(1,10)
(3,10)
(7,10)
(9,10)
(5,10)
```

注意，streaming会输出多次的输出，
每个输出是对每个流的统计
每个输出应该是这样的

RDD队列流

- 如果更换为循环30次，产生有30个RDD的RDD队列
- 每个队列被streaming处理后，分别输出统计结果

```
object QueueStream {
  def main(args: Array[String]) {
    val conf = new SparkConf().setMaster("local[2]").setAppName("queueStream")
    //每1秒对数据进行处理
    val ssc = new StreamingContext(conf,Seconds(1))
    //创建一个能够push到QueueInputDStream的RDDs队列
    val rddQueue = new mutable.SynchronizedQueue[RDD[Int]]()
    //基于一个RDD队列创建一个输入源
    val inputStream = ssc.queueStream(rddQueue)
    val mappedStream = inputStream.map(x => (x % 10,1))
    val reduceStream = mappedStream.reduceByKey(_ + _)
    reduceStream.print
    ssc.start()
    for(i <- 1 to 30){
      rddQueue += ssc.sparkContext.makeRDD(1 to 100, 2)   //创建RDD
      Thread.sleep(1000)
    }
    ssc.stop()
  }
}
```

```
-------------------------------------------
Time: 1459595433000 ms //第1个输出
-------------------------------------------
(4,10)
(0,10)
(6,10)
(8,10)
(2,10)
(1,10)
(3,10)
(7,10)
(9,10)
(5,10)
.....
.....
-------------------------------------------
Time: 1459595463000 ms //第30个输出
-------------------------------------------
(4,10)
(0,10)
(6,10)
(8,10)
(2,10)
(1,10)
(3,10)
(7,10)
(9,10)
(5,10)
```

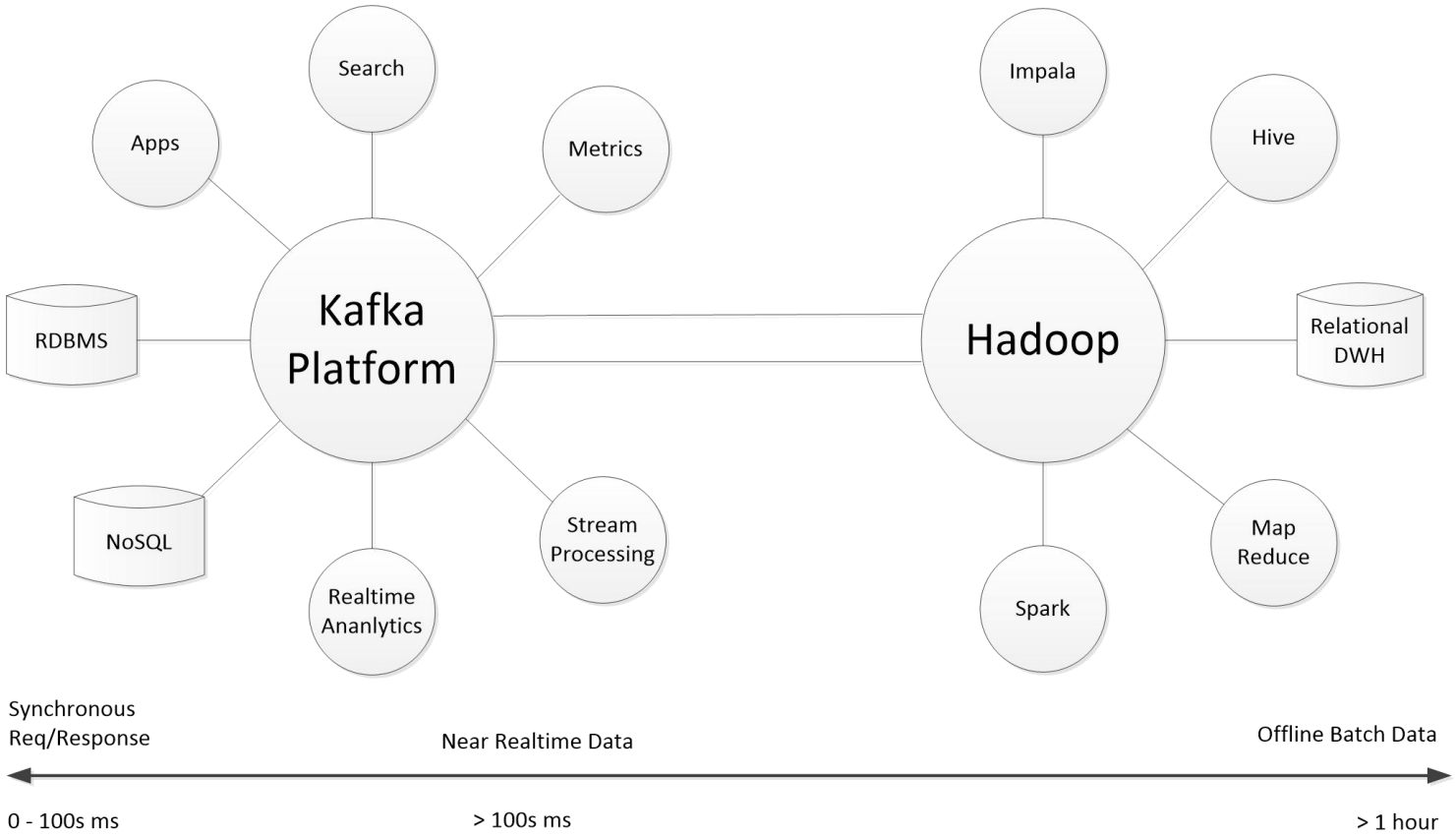
大数据处理—流计算

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

数据源管理——高级数据源之Kafka流

Kafka简介

- Kafka是一种高吞吐量的分布式发布订阅消息系统，用户通过Kafka系统可以发布大量的消息，同时也能实时订阅消费消息
- Kafka可以同时满足在线实时处理和批量离线处理
- 在公司的大数据生态系统中，可以把Kafka作为数据交换枢纽，不同类型的分布式系统（关系数据库、NoSQL数据库、流处理系统、批处理系统等），可以统一接入到Kafka，实现和Hadoop各个组件之间的不同类型数据的实时高效交换



Kafka作为数据交换枢纽

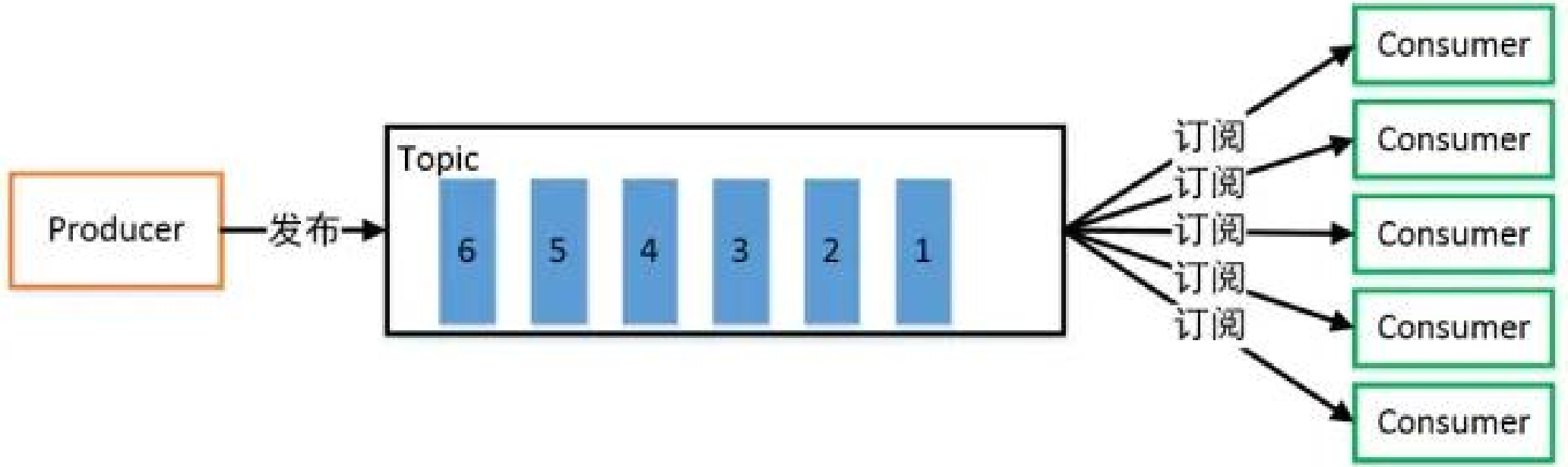
消息队列Message Queue

Kafka 本质上是一个 MQ（Message Queue）

- **解耦**：允许应用系统/大数据系统独立的扩展或修改队列两边的处理过程
- **可恢复性**：即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理
- **缓冲**：有助于解决生产消息和消费消息的处理速度不一致的情况
- **灵活性&峰值处理能力**：不会因为突发的超负荷的请求而完全崩溃，消息队列能够使关键组件顶住突发的访问压力
- **异步通信**：消息队列允许用户把消息放入队列但不立即处理它

发布/订阅模式

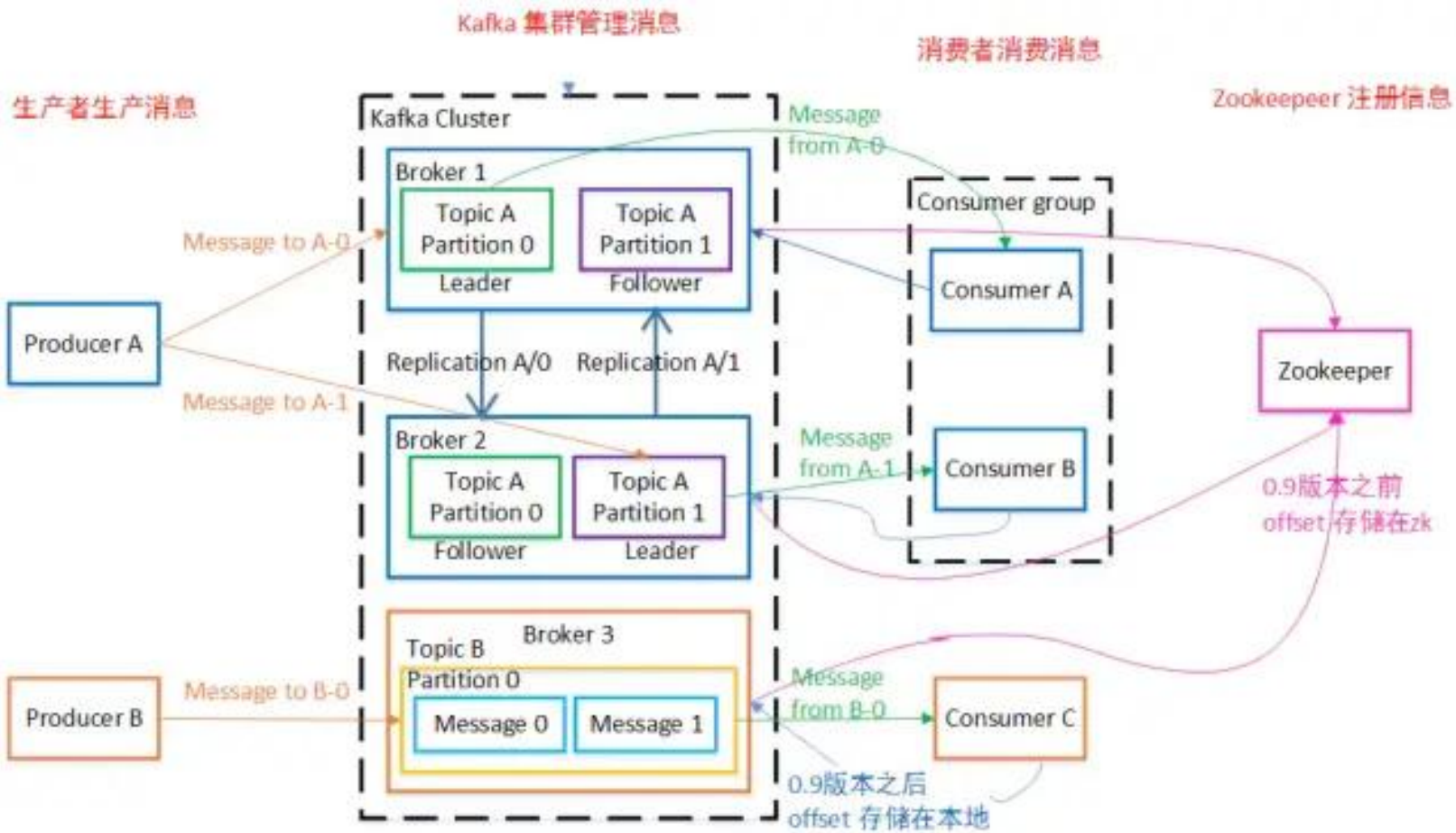
- 一对多模式
- 生产者将消息发布到 Topic 中，有多个消费者订阅该主题
- 发布到 Topic 的消息会被所有订阅者消费，被消费的数据不会立即从 Topic 清除



Kafka架构

- Kafka 存储的消息来自多种方式生成数据的进程，称为 Producer 生产者的进程
- 数据从而可以被发布到不同的 Topic 主题下的不同 Partition 分区
- 在一个分区内，这些消息被索引并连同时间戳存储在一起；其它被称为 Consumer 消费者的进程可以从分区订阅消息
- Kafka 运行在一个由一台或多台服务器组成的集群上，并且分区以跨集群结点分布，一台kafka机器是一个Broker
- Leader主副本、Follower从副本

一个 Topic由多个 Partition 组成
生产者/消费者可以以 Partition 为单位读写



offset：监控数据消费到什么位置

Kafka术语-1

- Producer： 消息生产者，向 Kafka Broker 发消息的客户端
- Consumer：消息消费者，从 Kafka Broker 取消息的客户端
- Consumer Group：消费者组（CG），消费者组内每个消费者负责消费不同分区的数据，提高消费能力。**一个分区只能由组内一个消费者消费，消费者组之间互不影响**。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者
- Broker：一台 Kafka 机器就是一个 Broker；一个集群由多个 Broker 组成；一个 Broker 可以容纳多个 Topic
- Topic：可以理解为一个队列；Topic 将消息分类，生产者和消费者面向的是同一个 Topic；每条发布到Kafka集群的消息都有一个类别——称为Topic；物理上不同Topic的消息分开存储，逻辑上一个Topic的消息虽然保存于一个或多个broker上，但用户只需指定消息的Topic即可，生产或消费数据而不必关心数据存于何处

Kafka术语-2

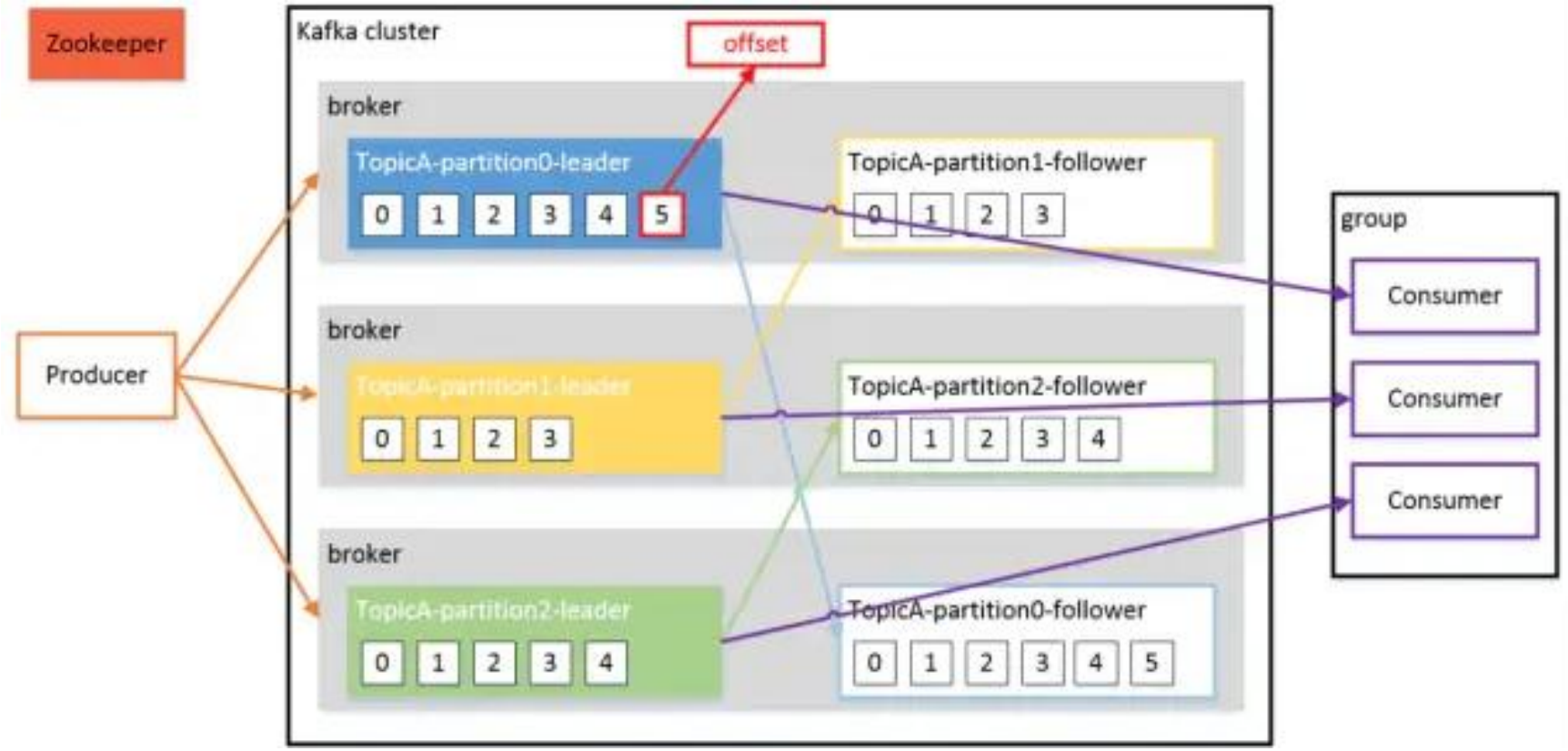
北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- Partition：为了实现扩展性，提高并发能力，一个非常大的 Topic 可以分布到多个 Broker （即服务器）上，一个 Topic 可以分为多个 Partition，每个 Partition 是一个有序的队列
- Replica：副本，为实现备份的功能，保证集群中的某个节点发生故障时，该节点上的 Partition 数据不丢失，且 Kafka 仍然能够继续工作，Kafka 提供了副本机制，一个 Topic 的每个分区都有若干个副本，一个 Leader 和若干个 Follower
  - Leader：每个分区多个副本的“主”副本，生产者发送数据的对象，以及消费者消费数据的对象，都是 Leader
  - Follower：每个分区多个副本的“从”副本，实时从 Leader 中同步数据，保持和 Leader 数据的同步；Leader 发生故障时，某个 Follower 还会成为新的 Leader
- Offset：消费者消费的位置信息，监控数据消费到什么位置，当消费者挂掉再重新恢复的时候，可以从消费位置继续消费
- Zookeeper：Kafka 集群的正常工作依赖于 Zookeeper帮助存储和管理集群信息

Kafka工作流程

- Kafka将 Record 流存储在称为 Topic 的类别中，每个记录由一个键、一个值和一个时间戳组成
- Kafka中消息是以Topic 进行分类，生产者生产消息，消费者消费消息，面向的都是同一个Topic
- Topic 是逻辑上的概念，而Partition 是物理上的概念，每个 Partition 对应于一个 log 文件，该 log 文件中存储的就是 Producer 生产的数据

- Producer生产的数据会不断追加到该 log 文件末端，且每条数据都有自己的 Offset
- 消费者组中的每个消费者，都会实时记录自己消费到了哪个 Offset，以便出错恢复时，从上次的位置继续消费

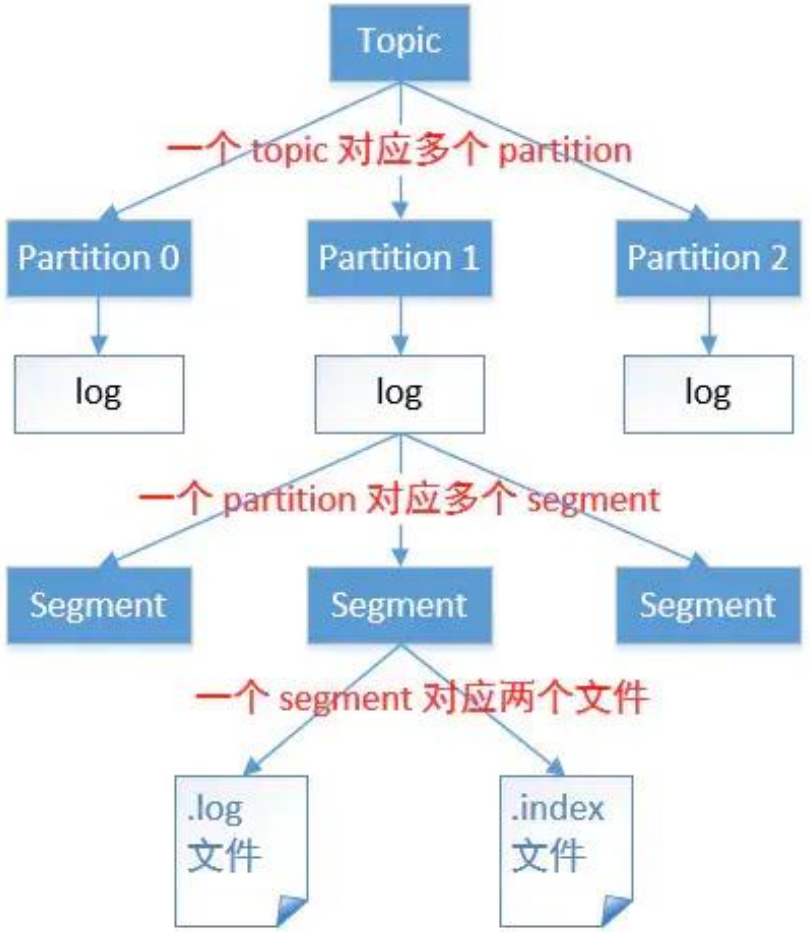


Kafka存储机制

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- 由于生产者生产的消息会不断追加到 log 文件末尾，为防止 log 文件过大导致数据定位效率低下，Kafka 采取了分片和索引机制
- 它将每个 Partition 分为多个 Segment，每个 Segment 对应两个文件："index" 索引文件和 "log" 数据文件
- 这些文件位于同一文件下，该文件夹的命名规则为：topic 名-分区号
- 例如，first 这个 topic 有三分分区，则其对应的文件夹为 first-0，first-1，first-2
- index 和 log 文件以当前 Segment 的第一条消息的 Offset 命名

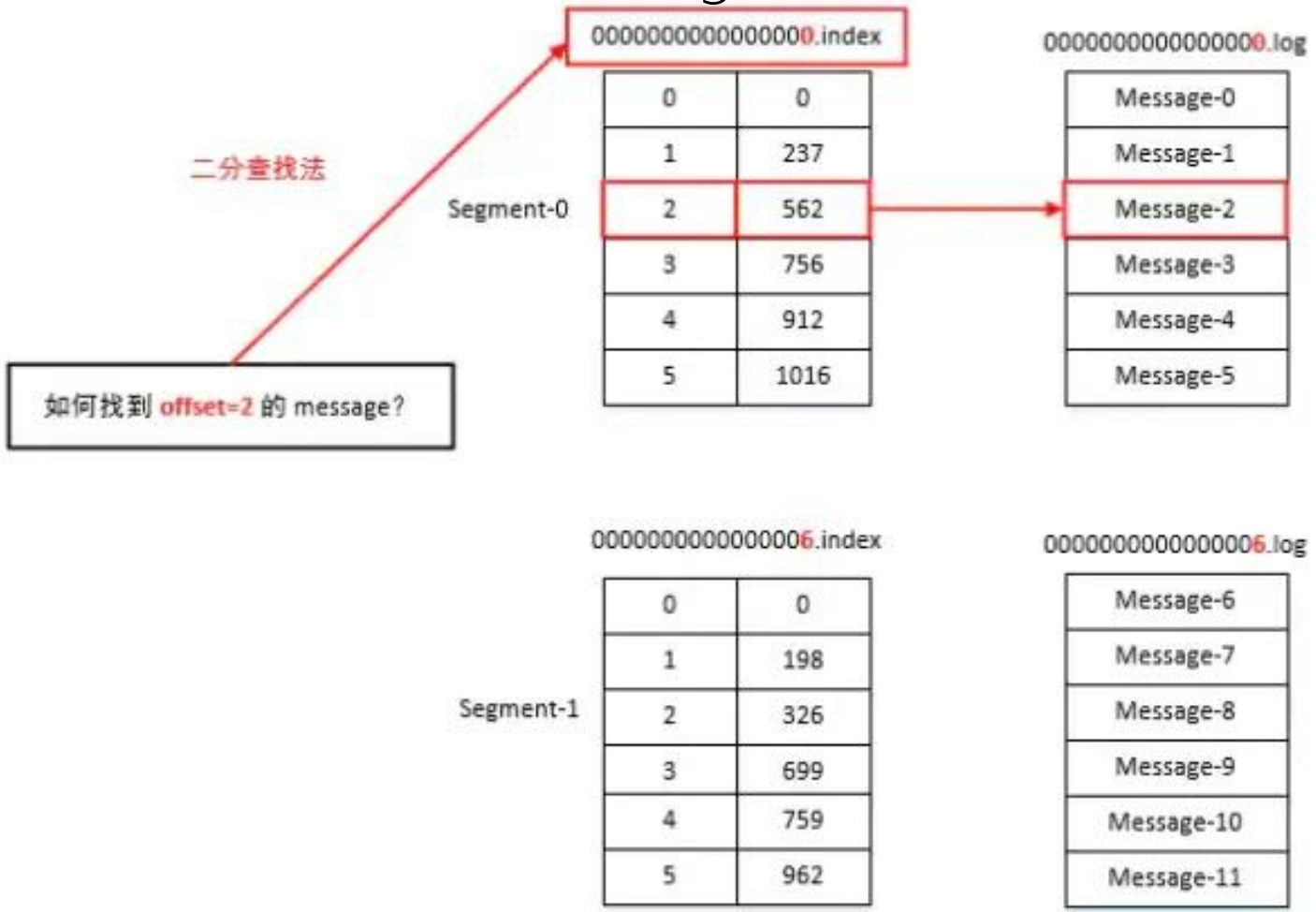
```
# ls /root/data/kafka/first-0
00000000000000009014.index
00000000000000009014.log
00000000000000009014.timeindex
00000000000000009014.snapshot
leader-epoch-checkpoint
```



Kafka存储机制

- ".index" 文件存储大量的索引信息
- ".log" 文件存储大量的数据
- 索引文件中的元数据指向对应数据文件中 Message 的物理偏移量

index 文件和 log 文件的结构示意图



Kafka生产者-分区策略

分区原因：

- 方便在集群中扩展，每个 Partition 可以通过调整以适应它所在的机器
- 一个 Topic可以有多个 Partition 组成，因此可以以 Partition 为单位读写

分区原则：

- 需要将 Producer 发送的数据封装成一个 ProducerRecord 对象

该对象需要指定一些参数：

- **topic**：string 类型，NotNull
- **partition**：int 类型，可选
- **timestamp**：long 类型，可选
- **key**：string 类型，可选
- **value**：string 类型，可选
- **headers**：array 类型，Nullable

1. 指明 Partition 的情况下，直接将给定的 Value 作为 Partition 的值
2. 没有指明 Partition 但有 Key 的情况下，将 Key 的 Hash 值与分区数取余得到 Partition 值
3. 既没有 Partition 有没有 Key 的情况下，第一次调用时随机生成一个整数（后面每次调用都在这个整数上自增），将这个值与可用的分区数取余，得到 Partition 值，也就是常说的 Round-Robin 轮询算法

Kafka消费者-消费方式 pull 和 push

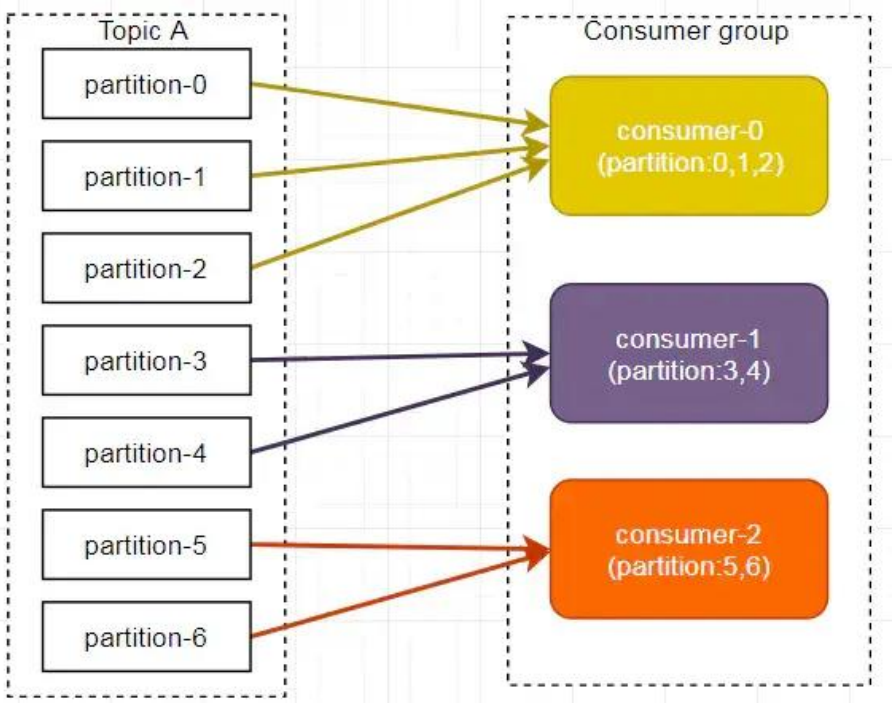
- push方式：由消息中间件主动地将消息推送给消费者
  - 优点：优点是不需要消费者额外开启线程监控中间件，节省开销
  - 缺点：无法适应消费速率不相同的消费者；因为消息的发送速率是broker决定的，而消费者的处理速度又不尽相同，所以容易造成部分消费者空闲，部分消费者堆积，造成缓冲区溢出
- pull方式：由消费者主动向消息中间件拉取消息
  - 优点：消费端可以按处理能力进行拉取
  - 缺点：消费端需要另开线程监控中间件，有性能开销
- 对于Kafka而言，pull模式更合适
- pull模式可简化broker的设计，Consumer可自主控制消费消息的速率，同时Consumer可以自己控制消费方式，既可批量消费也可逐条消费，同时还能选择不同的提交方式从而实现不同的传输语义

Kafka消费者-分区分配策略

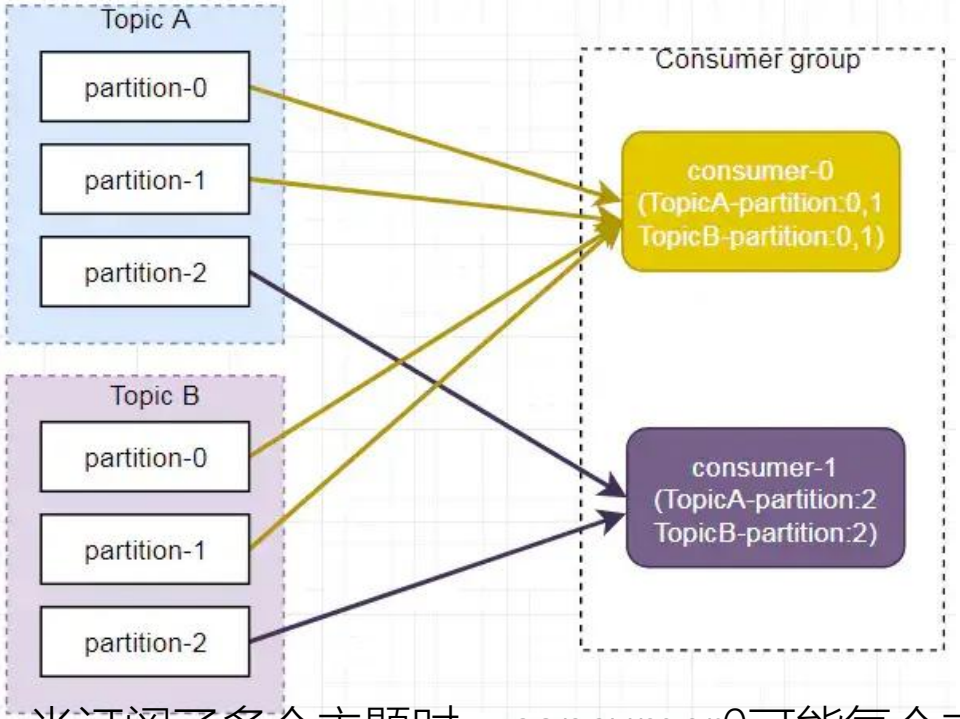
- 一个 Consumer Group 中有多个 Consumer，一个 Topic 有多个 Partition，所以必然会涉及到 Partition 的分配问题，即确定哪个 Partition 由哪个 Consumer 来消费
- Kafka 有两种分配策略，一个是 RoundRobin，一个是 Range，默认为Range，当消费者组内消费者发生变化时，会触发分区分配策略（方法重新分配）
  - Range策略会将消费组内所有订阅这个topic的消费者按照名称的字典序排序，然后为每个消费者划分固定的分区范围，如果不够平均分配，那么字典序靠前的消费者会被多分配一个分区
  - RoundRobin策略的原理是将消费组内所有消费者以及消费者所订阅的所有topic的partition按照字典序排序，然后通过轮询方式逐个将分区以此分配给每个消费者

Kafka消费者-分区分配策略 Range 方式

- Range 方式针对于每个topic，各个topic之间分配时没有任何关联
- 但是，如右侧图所示，Consumer0、Consumer1 同时订阅了主题 A 和 B，可能造成消息分配不对等问题，当消费者组内订阅的主题越多，分区分配可能越不均衡



当只订阅一个主题时，consumer0多分到一个分区消费

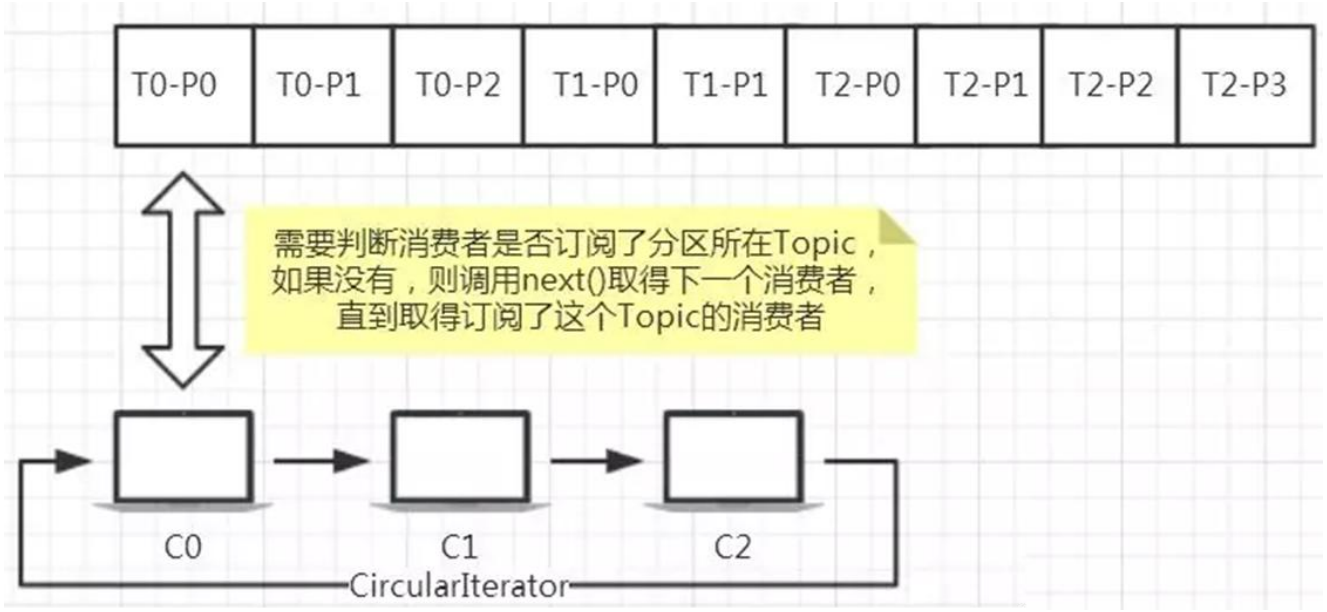


当订阅了多个主题时，consumer0可能每个主题都被多分到一个分区消费，上面consumer0比consumer1多了2个分区要消费

Kafka消费者-分区分配策略

RoundRobin 轮询方式

- RoundRobin策略针对于全局所有的topic和消费者，分配步骤如下：
- 消费者按照字典排序，例如C0, C1, C2... ...，并构造环形迭代器
- topic名称按照字典排序，并得到每个topic的所有分区，从而得到所有分区集合
- 遍历第2步所有分区集合，同时轮询消费者
- 如果轮询到的消费者订阅的topic不包括当前遍历的分区所属topic，则跳过；否则分配给当前消费者，并继续第3步



- 3个Topic：T0（3个分区0, 1, 2），T1（两个分区0, 1），T2（4个分区0, 1, 2, 3）；
- 3个Consumer: C0订阅了[T0, T1]， C1订阅了[T1, T2]， C2订阅了[T2, T0]；
- RoundRobin结果分配结果如下：
  T0-P0分配给C0，T0-P1分配给C2，T0-P2分配给C0，
  T1-P0分配给C1，T1-P1分配给C0，
  T2-P0分配给C1，T2-P1分配给C2，T2-P2分配给C1，T0-P3分配给C2；

Kafka消费者-offset 的维护

- 由于 Consumer 在消费过程中可能会出现断电宕机等故障，Consumer 恢复后，需要从故障前的位置继续消费
- 所以 Consumer 需要实时记录自己消费到了哪个offset，以便故障恢复后继续消费
- Kafka 0.9 版本之前，Consumer 默认将 offset 保存在 Zookeeper 中，从 0.9 版本开始，Consumer 默认将 offset 保存在 Kafka 一个内置的 Topic 中，该 Topic 为 __consumer_offsets

Kafka准备工作

1.启动Kafka

- 假设已经成功安装Kafka到“/usr/local/kafka”目录下
- 打开一个终端，输入下面命令启动Zookeeper服务：

```
$ cd /usr/local/kafka
$ ./bin/zookeeper-server-start.sh
config/zookeeper.properties
```

- 不要关闭这个终端窗口，一旦关闭，Zookeeper服务就停止了

Kafka准备工作

- 打开第二个终端，然后输入下面命令启动Kafka服务：

```
$ cd /usr/local/kafka
$ bin/kafka-server-start.sh config/server.properties
```

- 不要关闭这个终端窗口，一旦关闭，Kafka服务就停止了

Kafka准备工作

2.测试Kafka是否正常工作

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- 再打开第三个终端，然后输入下面命令创建一个自定义名称为“wordsendertest”的Topic：

```
$ cd  /usr/local/kafka
$ ./bin/kafka-topics.sh  --create  --zookeeper  localhost:2181  \
>--replication-factor  1  --partitions  1  --topic  wordsendertest
#可以用list列出所有创建的Topic，验证是否创建成功
$ ./bin/kafka-topics.sh  --list  --zookeeper  localhost:2181
```

Kafka准备工作

- 下面用生产者（Producer）来产生一些数据，请在当前终端内继续输入下面命令：

```
$ ./bin/kafka-console-producer.sh  --broker-list  localhost:9092 \
>  --topic  wordsendertest
```

- 上面命令执行后，就可以在当前终端内用键盘输入一些英文单词，比如可以输入：

```
hello hadoop
hello spark
```

Kafka准备工作

- 现在可以启动一个消费者，来查看刚才生产者产生的数据。请另外打开第四个终端，输入下面命令：

```
$ cd /usr/local/kafka
$./bin/kafka-console-consumer.sh  --zookeeper  localhost:2181  \
> --topic  wordsendertest  --from-beginning
```

- 可以看到，屏幕上会显示出如下结果，也就是刚才在另外一个终端里面输入的内容：

```
hello hadoop
hello spark
```

Spark准备工作

- 1.添加相关jar包
- 2.启动spark-shell

Spark准备工作

1.添加相关jar包

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- Kafka和Flume等高级输入源，需要依赖独立的库（jar文件）
- 没有添加对应jar包前，在spark-shell中执行下面import语句进行测试：

```
scala> import org.apache.spark.streaming.kafka._
<console>:25: error: object kafka is not a member of
package org.apache.spark.streaming
       import org.apache.spark.streaming.kafka._
                                          ^
```

- 对于Spark2.1.0版本，如果要使用Kafka，则需要下载spark-streaming-kafka-0-8_2.11相关jar包，下载地址：
- http://mvnrepository.com/artifact/org.apache.spark/spark-streaming-kafka-0-8_2.11/2.1.0

2.11是Scala的版本，2.1.0是spark版本

Spark准备工作

- 把jar文件复制到Spark目录的jars目录下

```
$ cd /usr/local/spark/jars
$ mkdir kafka
$ cd ~
$ cd 下载
$ cp ./spark-streaming-kafka-0-8_2.11-2.1.0.jar
/usr/local/spark/jars/kafka
```

- 继续把Kafka安装目录的libs目录下的所有jar文件复制到“/usr/local/spark/jars/kafka”目录下，请在终端中执行下面命令：

```
$ cd /usr/local/kafka/libs
$ cp ./* /usr/local/spark/jars/kafka
```

Spark准备工作

2.启动spark-shell

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- 执行如下命令启动spark-shell：

```
$ cd  /usr/local/spark
$ ./bin/spark-shell  \
>--jars /usr/local/spark/jars/*:/usr/local/spark/jars/kafka/*
```

- 启动成功后，再次执行如下命令：

```
scala> import  org.apache.spark.streaming.kafka._
//会显示下面信息
import org.apache.spark.streaming.kafka._
```

编写Spark Streaming程序使用Kafka数据源

- 1.编写生产者（producer）程序
- 2.编写消费者（consumer）程序
- 3.编写日志格式设置程序
- 4.编译打包程序
- 5.运行程序

1.编写生产者（producer）程序

- 编写KafkaWordProducer程序。执行命令创建代码目录：

```
$ cd /usr/local/spark/mycode
$ mkdir kafka
$ cd kafka
$ mkdir -p src/main/scala
$ cd src/main/scala
$ vim KafkaWordProducer.scala
```

- 在KafkaWordProducer.scala中输入以下代码：

```
package org.apache.spark.examples.streaming
import java.util.HashMap
import org.apache.kafka.clients.producer.{KafkaProducer, ProducerConfig,
ProducerRecord}
import org.apache.spark.SparkConf
import org.apache.spark.streaming._
import org.apache.spark.streaming.kafka._
```

剩余代码见下一页

1.编写生产者（producer）程序

```
$ /usr/local/spark/bin/spark-submit  \
>--driver-class-path /usr/local/spark/jars/*:/usr/local/spark/jars/kafka/*  \
>--class "org.apache.spark.examples.streaming.KafkaWordProducer"  \
>/usr/local/spark/mycode/kafka/target/scala-2.11/simple-project_2.11-1.0.jar  \
> localhost:9092 wordsender 3 5
```

- KafkaWordProducer程序模拟产生一系列字符串（产生随机的整数序列）
- metadataBrokerList参数指定Kafka的Broker的地址
- topic参数指定topic的名称
- messagesPerSec参数指定每秒发送几条消息
- wordsPerMessage参数指定每条消息包含几个单词（这里程序生成的message是整数序列，也就是指定一次发几个整数）

```
object KafkaWordProducer {
  def main(args: Array[String]) {
    if (args.length < 4) {
      System.err.println("Usage: KafkaWordCountProducer <metadataBrokerList>
<topic> " +
        "<messagesPerSec> <wordsPerMessage>")
      System.exit(1)
    }
    val Array(brokers, topic, messagesPerSec, wordsPerMessage) = args
    // Zookeeper connection properties
    val props = new HashMap[String, Object]()
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, brokers)
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
      "org.apache.kafka.common.serialization.StringSerializer")
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
      "org.apache.kafka.common.serialization.StringSerializer")
    val producer = new KafkaProducer[String, String](props)
```

创建producer对象

剩余代码见下一页

1.编写生产者（producer）程序

北京郵電大學 BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

- 根据messagesPerSec（每秒发送几条消息）输入参数来准备message
- 根据wordsPerMessage（每条消息发送几个单词）输入参数，为每条消息产生几个随机数，构成str字符串
- ProducerRecord类用str字符串来创建名称为topic的数据message
- 通过producer对象的send方法发送出message

```
// Send some messages
    while(true) {
      (1 to messagesPerSec.toInt).foreach { messageNum =>
        val str = (1 to wordsPerMessage.toInt).map(x =>
scala.util.Random.nextInt(10).toString)
          .mkString(" ")
                print(str)
                println()
        val message = new ProducerRecord[String, String](topic, null, str)
        producer.send(message)
      }
     Thread.sleep(1000)
    }
  }
}
```

producer对象的send方法，发送数据

```
7 5 0 7 3
2 8 2 1 3
0 1 2 9 2
8 0 9 0 9
9 0 0 6 8
6 6 1 6 5
8 3 6 7 7
.....
```

2.编写消费者（consumer）程序

- 继续在当前目录下创建KafkaWordCount.scala代码文件，它会把KafkaWordProducer发送过来的单词进行词频统计，代码内容如下：

```
package org.apache.spark.examples.streaming
import org.apache.spark._
import org.apache.spark.SparkConf
import org.apache.spark.streaming._
import org.apache.spark.streaming.kafka._
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.kafka.KafkaUtils
```

剩余代码见下一页

2.编写消费者（consumer）程序

- 在SparkStreaming中，如果是文件流类型的数据源，Spark自身的容错机制可以保证数据不会发生丢失
- 对于Kafka等数据源，当数据源源不断到达时，会首先被放入到缓存中，尚未被处理，可能会发生丢失
- 为避免系统失败时发生数据丢失，可以通过ssc.checkpoint()创建检查点

```
object KafkaWordCount{
def main(args:Array[String]){
StreamingExamples.setStreamingLogLevels()
val sc = new SparkConf().setAppName("KafkaWordCount").setMaster("local[2]")
val ssc = new StreamingContext(sc,Seconds(10))
ssc.checkpoint("file:///usr/local/spark/mycode/kafka/checkpoint") //设置检查点，
//若存放在HDFS上，则写成类似ssc.checkpoint("/user/hadoop/checkpoint")这种形式，但是，要启动hadoop
val zkQuorum = "localhost:2181" //Zookeeper服务器地址
val group = "1"
 //topic所在的group
//可以设置为自己想要的名称，如 val group = "test-consumer-group"
```

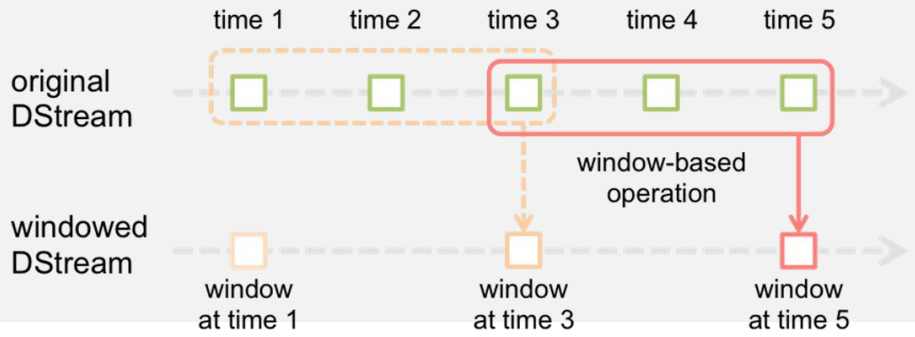
剩余代码见下一页

- 注意：设置检查点之后的数据仍然可能发生丢失，如果要保证数据不发生丢失，可以开启SparkStreaming的预写式日志（WAL:Write Ahead Logs)功能。当采用预写式日志以后，接收数据的正确性只在数据被预写到日志以后Receiver才会确认。
- 当系统发生失败导致缓存中数据丢失时，就可以从日志中恢复丢失的数据。预写式日志需要额外的开销，因此默认WAL功能是关闭的。设置SparkConf的属性spark.streaming. receiver.writeAheadLog.enable为true，开启该功能

2.编写消费者（consumer）程序

- val topics = “wordsender” 这里在consumer的程序里，写死了要去消费的数据topic名称
- val topics = “wordsender”这里如果存在多个topic，可以用逗号隔开
- 对接收到的topic数据做词频统计

```
val topics = "wordsender"  //topics的名称
val numThreads = 1  //每个topic的分区数
val topicMap =topics.split(",").map((_,numThreads.toInt)).toMap
val lineMap = KafkaUtils.createStream(ssc,zkQuorum,group,topicMap)
val lines = lineMap.map(_._2)
//前面producer发送过来的键值对的key是null，这里只要取出value的值
val words = lines.flatMap(_.split(" "))
val pair = words.map(x => (x,1))
val wordCounts = pair.reduceByKeyAndWindow(_ + _,_ -_,Minutes(2),Seconds(10),2)
 //采用滑动窗口方式，进行增量计算，统计收到的实时数据的词频
wordCounts.print
ssc.start
ssc.awaitTermination
}
}
```



reduceByKeyAndWindow(func,invFunc,windowLength, slideInterval,[numTasks])	先对滑动窗口中新的时间间隔内的数据进行增量聚合，再移去最早的同等时间间隔内的数据统计量。

3.编写日志格式设置程序

- 继续在当前目录下创建StreamingExamples.scala代码文件，，用于设置log4j：

```
package org.apache.spark.examples.streaming
import org.apache.spark.Logging
import org.apache.log4j.{Level, Logger}
/** Utility functions for Spark Streaming examples. */
object StreamingExamples extends Logging {
  /** Set reasonable logging levels for streaming if the user has not configured log4j.
*/
  def setStreamingLogLevels() {
    val log4jInitialized = Logger.getRootLogger.getAllAppenders.hasMoreElements
    if (!log4jInitialized) {
      // We first log something to initialize Spark's default logging, then we override the
      // logging level.
      logInfo("Setting log level to [WARN] for streaming example." +
        " To override add a custom log4j.properties to the classpath.")
      Logger.getRootLogger.setLevel(Level.WARN)
    }
  }
}
```

4.编译打包程序

- 创建simple.sbt文件：

```
$ cd /usr/local/spark/mycode/kafka/
$ vim simple.sbt
```

- 在simple.sbt中输入以下代码：

```
name := "Simple Project"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.1.0"
libraryDependencies += "org.apache.spark" % "spark-streaming_2.11" % "2.1.0"
libraryDependencies += "org.apache.spark" % "spark-streaming-kafka-0-8_2.11" % "2.1.0"
```

- 进行打包编译：

```
$ cd /usr/local/spark/mycode/kafka/
$ /usr/local/sbt/sbt package
```

5.运行程序-运行KafkaWordProducer程序

- 打开一个终端，执行如下命令，运行“KafkaWordProducer”程序，生成一些单词（是一堆整数形式的单词）：

```
$ cd /usr/local/spark
$ /usr/local/spark/bin/spark-submit  \
>--driver-class-path /usr/local/spark/jars/*:/usr/local/spark/jars/kafka/*  \
>--class "org.apache.spark.examples.streaming.KafkaWordProducer"  \
>/usr/local/spark/mycode/kafka/target/scala-2.11/simple-project_2.11-1.0.jar  \
> localhost:9092 wordsender 3 5
```

指定broker是localhost:9092
指定生产者producer的名字
指定每秒发3条消息
指定每条消息发5个单词

5.运行程序-运行KafkaWordProducer程序

- 执行上面命令后，屏幕上会不断滚动出现新的单词，如下：

```
7 5 0 7 3
2 8 2 1 3
0 1 2 9 2
8 0 9 0 9
9 0 0 6 8
6 6 1 6 5
8 3 6 7 7
.....
```

- 不要关闭这个终端窗口，就让它一直不断发送单词

5.运行程序-运行KafkaWordCount程序

- 请新打开一个终端，执行下面命令，运行KafkaWordCount程序，执行词频统计：

```
$ cd /usr/local/spark
$ /usr/local/spark/bin/spark-submit   \
>--driver-class-path /usr/local/spark/jars/*:/usr/local/spark/jars/kafka/* \
>--class "org.apache.spark.examples.streaming.KafkaWordCount"   \
>/usr/local/spark/mycode/kafka/target/scala-2.11/simple-project_2.11-1.0.jar
```

5.运行程序-运行KafkaWordCount程序

- 运行上面命令以后，就启动了词频统计功能，屏幕上就会显示如下类似信息：

```
-------------------------------------------
Time: 1488156500000 ms
-------------------------------------------
(4,5)
(8,12)
(6,14)
(0,19)
(2,11)
(7,20)
(5,10)
(9,9)
(3,9)
(1,11)
```

北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

谢谢聆听
批评指正

ehaihong@bupt.edu.cn