# 

**Spark Streaming的分析与应用**

**需求规格说明书**

Version 1.0.1

小组成员：

陈少杰

姜鑫

蒲彦均

邹嘉欣

**版本变更记录**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本 | 变更时间 | 修改人 | 审核人 | 备注 |
| 1.0 | 2017/03/22 | 邹嘉欣、陈少杰 | 姜鑫、蒲彦均 | 初稿 |
| 1.0.1 | 2017/03/23 | 陈少杰 | 邹嘉欣 | 增加用例图 |
|  |  |  |  |  |

目录

[图片目录 4](#_Toc477973223)

[1. 前言 5](#_Toc477973224)

[1.1 目的 5](#_Toc477973225)

[1.2 系统概述 5](#_Toc477973226)

[1.3 文档概述 5](#_Toc477973227)

[1.4 术语和缩略语 6](#_Toc477973228)

[2. 引用文档 6](#_Toc477973229)

[3. 功能性需求 7](#_Toc477973230)

[3.1 数据的产生与输入 7](#_Toc477973231)

[3.1.1输入流接收器 7](#_Toc477973232)

[3.1.2 数据源输入流 7](#_Toc477973233)

[3.2 数据流抽象DStream 7](#_Toc477973234)

[3.2.1 DStream的离散化 7](#_Toc477973235)

[3.2.2 DStream转换及构建DStream Gragh 7](#_Toc477973236)

[3.3 作业调度 7](#_Toc477973237)

[3.3.1 Spark应用之间的资源调度 7](#_Toc477973238)

[3.3.2 同一应用程序内的资源调度 8](#_Toc477973239)

[3.4 长时间容错 9](#_Toc477973240)

[3.5 窗口支持 9](#_Toc477973241)

[4. 非功能性需求 9](#_Toc477973242)

[4.1性能调优 9](#_Toc477973243)

[4.1.1合理的批处理时间 9](#_Toc477973244)

[4.1.2 增加并行化 9](#_Toc477973245)

[4.1.3 及时清除过期数据 10](#_Toc477973246)

图片目录

图 1-1 Spark生态系统 6

# 前言

## 目的

以Spark Streaming开源项目及相关资料为输入，分析软件设计需求，结合软件工程综合实验具体要求，输出软件需求规格说明书，作为设计开发的依据并指导后续的开发工作。

## 系统概述

Apache Spark是一个围绕速度、易用性和复杂分析构建的大数据处理框架。最初在2009年由加州大学伯克利分校的AMP-Lab开发，并于2010年成为Apache的开源项目之一。

Spark是一个类似于Hadoop的MapReduce的分布式计算框架，其核心是弹性分布式数据集，提供了比MapReduce更丰富的模型，可以在快速在内存中对数据集进行多次迭代，以支持复杂的数据挖掘算法和图形计算算法。

Spark Streaming是建立在Spark上的实时计算框架，通过它提供的丰富的API、基于内存的高速执行引擎，用户可以结合流式、批处理和交互式查询应用，扩展了Spark处理大规模流式数据的能力。图1-1展示了Spark与其配套的Spark SQL, Spark Streaming, MLib和GraphX等上层模块构成的生态系统。



图 1-1 Spark生态系统

## 文档概述

文档用途：本文档主要是介绍Spark Streaming系统需求及规格说明。

主要内容：

* 以用例图、状态图的形式给出Spark Streaming系统功能需求的分解结构，并对用例模型中的参与者和用例进行详细的描述，其中主要包括软件系统的用例模型、系统的核心流程等；
* 使用 RUCM 模型对功能需求进行建模；
* 描述了与此次系统实施相关的硬件环境的一些要求；
* 描述了与此系统实施相关的软件环境的要求。

## 术语和缩略语

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 术语 | 英文 | 说明 |
| 1 | UCM | UCM | 用例建模 |
| 2 | RUCM | RUCM | 限制性用例模型 |
| 3 | SparkConf | SparkConf | 包含了集群配置的一系列参数，包括master节点，app名称等。 |
| 4 | SparkContext | SparkContext | 所有Spark程序的入口，相当于main函数。可以完成变量广播，RDD的操作等。SparkContext的构建需要SparkConf。 |
| 5 | DStream | DStream | DStream是Spark的一个基本抽象概念，它代表了一个连续的数据流，数据流按照时间间隔进行切分，会产生一系列的RDD。所以，DStream也可以看作是RDD的序列。 |
| 6 | RDD | RDD | RDD(Resilient Distributed Datasets)弹性分布式数据集，是分布式内存的一个抽象概念，本质上是一个只读的分区记录集合。 |
| 7 | filter | filter | Spark Streaming主要适用于流式的数据处理，而输入流中的数据并不是全部都需要处理，这时便可以使用filter进行数据过滤，减少集群计算压力。 |
| 8 | map/mapPartition/flatMap | map/mapPartition/flatMap | map系列的函数是基于一个RDD中的元素来产生一个新的RDD的方式。其具体的用法，区别，性能等也是本项目研究的方向之一。 |

# 引用文档

* 《Spark Streaming的分析与应用项目计划》

# 功能性需求

功能性的需求可以分为数据的产生与输入、数据流抽象、作业调度、长时间容错和窗口支持五部分。图3.1为系统用例图。



图3.1 系统用例图

## 3.1 数据的产生与输入

### 3.1.1输入流接收器

输入接收器Receiver用来持续产生或持续接收系统外的数据，比如 TwitterReceiver 可以实时爬取 twitter 数据。为了支持多种数据源，需要定义多种Receiver，对不同的数据源进行数据获取。

每个ReceiverSupervisor启动后将马上生成一个用户提供的 Receiver 实现的实例，并在 Receiver 实例生成后开始持续不断地接收外界数据，并持续交给ReceiverSupervisor。ReceiverSupervisor将数据成块存储。存储完成后，ReceiverSupervisor及时上报数据的mata信息给driver 端的ReceiverTracker。

### 3.1.2 数据源输入流

在driver端，数据源输入流（ReceiverInputDStream）在每个batch检查ReceiverTracker 收到的数据块的meta信息，界定哪些新数据需要在本batch内处理，然后生成相应的 RDD 实例去处理这些块数据。

## 3.2 数据流抽象DStream

有了数据的来源，就需要有数据的处理与存储。DStream是Spark Streaming中所有流数据的抽象。

### 3.2.1 DStream的离散化

DStream（Discretized Stream）是Spark Streaming对内部持续的实时数据流的抽象描述。而要处理的实时数据流，在Spark Streaming中对应于一个DStream 实例；另外，RDD（Resilient Distributed Datasets）是Spark最基本也是最根本的数据抽象，是只读的分区记录集合。 DStream 代表了一系列连续的RDD，DStream中每个RDD包含特定时间间隔的数据。



图 3.2 DStream的离散化

对DStream应用的任何操作都会转换为RDD上的操作。例如，在将文本流转换为词流的示例中，flatMap操作应用于文本的DStream中的每个RDD，以生成词的DStream的RDD 。



图 3.3 DStream与RDD

### 3.2.2 DStream的操作

|  |  |
| --- | --- |
| **用例** | **描述** |
| **map**(*func*) | 源DStream的每个元素通过函数func返回一个新的DStream。 |
| **flatMap**(*func*) | 类似与map操作，不同的是每个输入元素可以被映射出0或者更多的输出元素。 |
| **filter**(*func*) | 在源DSTREAM上选择Func函数返回仅为true的元素,最终返回一个新的DSTREAM 。 |
| **union**(*otherStream*) | 返回一个包含源DStream与其他DStream的元素合并后的新DStream。 |
| **count**() | 对源DStream内部的所含有的RDD的元素数量进行计数，返回一个内部的RDD只包含一个元素的DStream。 |
| **reduce**(*func*) | 使用函数func（有两个参数并返回一个结果）将源DStream 中每个RDD的元素进行聚 合操作,返回一个内部所包含的RDD只有一个元素的新DStream。 |
| **countByValue**() | 计算DStream中每个RDD内的元素出现的频次并返回新的DStream[(K,Long)]，其中K是RDD中元素的类型，Long是元素出现的频次。 |
| **reduceByKey**(*func*, [*numTasks*]) | 当一个类型为（K，V）键值对的DStream被调用的时候,返回类型为类型为（K，V）键值对的新 DStream,其中每个键的值V都是使用聚合函数func汇总。注意：默认情况下，使用 Spark的默认并行度提交任务（本地模式下并行度为2，集群模式下位8），可以通过配置numTasks设置不同的并行任务数。 |
| **join**(*otherStream*, [*numTasks*]) | 当被调用类型分别为（K，V）和（K，W）键值对的2个DStream时，返回类型为（K，（V，W））键值对的一个新 DSTREAM。 |
| **cogroup**(*otherStream*, [*numTasks*]) | 当被调用的两个DStream分别含有(K, V) 和(K, W)键值对时,返回一个(K, Seq[V], Seq[W])类型的新的DStream。 |
| **transform**(*func*) | 通过对源DStream的每RDD应用RDD-to-RDD函数返回一个新的DStream，这可以用来在DStream做任意RDD操作。 |
| **updateStateByKey**(*func*) | 返回一个新状态的DStream,其中每个键的状态是根据键的前一个状态和键的新值应用给定函数func后的更新。这个方法可以被用来维持每个键的任何状态数据。 |
| **print()** | 在Driver中打印出DStream中数据的前10个元素。 |
| **saveAsTextFiles(prefix, [suffix])** | 将DStream中的内容以文本的形式保存为文本文件，其中每次批处理间隔内产生的文件以prefix-TIME\_IN\_MS[.suffix]的方式命名。 |
| **saveAsObjectFiles(prefix, [suffix])** | 将DStream中的内容按对象序列化并且以SequenceFile的格式保存。其中每次批处理间隔内产生的文件以prefix-TIME\_IN\_MS[.suffix]的方式命名。 |
| **saveAsHadoopFiles(prefix, [suffix])** | 将DStream中的内容以文本的形式保存为Hadoop文件，其中每次批处理间隔内产生的文件以prefix-TIME\_IN\_MS[.suffix]的方式命名。 |
| **foreachRDD(func)** | 最基本的输出操作，将func函数应用于DStream中的RDD上，这个操作会输出数据到外部系统，比如保存RDD到文件或者网络数据库等。需要注意的是func函数是在运行该streaming应用的Driver进程里执行的。 |

## 3.3 作业调度

在每个Spark应用（包含一个SparkContext实例）中运行了一些其独占的执行器（executor）进程。集群管理器提供了Spark应用之间的资源调度。在各个Spark应用内部，各个线程可能并发地通过action操作提交多个Spark作业。在Spark应用内部（对应同一个SparkContext）各个作业之间，Spark默认FIFO调度，同时也可以支持公平调度。下面将分别介绍Spark应用之间的资源调度以及Spark应用内部的资源调度。

### 3.3.1 Spark应用之间的资源调度

如果在集群上运行，每个Spark应用都会获得一批独占的执行器JVM，来运行其任务并存储数据。如果有多个用户共享集群，那么会有很多资源分配相关的选项，如何设置还取决于具体的集群管理器。

对Spark所支持的各个集群管理器而言，最简单的资源分配，就是对资源静态划分。这种方式就意味着，每个Spark应用都是设定一个最大可用资源总量，并且该应用在整个生命周期内都会占住这些资源。这种方式在Spark独立部署（standalone）和YARN调度，以及Mesos粗粒度模式（coarse-grained Mesos mode）下都可用。

* Standalone mode: 默认情况下，Spark应用在独立部署的集群中都会以FIFO（first-in-first-out）模式顺序提交运行，并且每个Spark应用都会占用集群中所有可用节点。不过可以通过设置spark.cores.max或者spark.deploy.defaultCores 来限制单个应用所占用的节点个数。最后，除了可以控制对CPU的使用数量之外，还可以通过 spark.executor.memory 来控制各个应用的内存占用量。
* Mesos: 在Mesos中要使用静态划分的话，需要将 spark.mesos.coarse 设为true，同样，也需要设置 spark.cores.max 来控制各个应用的CPU总数，以及 spark.executor.memory 来控制各个应用的内存占用。
* YARN: 在YARN中需要使用 –num-executors 选项来控制Spark应用在集群中分配的执行器的个数，对于单个执行器（executor）所占用的资源，可以使用 –executor-memory 和 –executor-cores 来控制。

### 3.3.2 同一应用程序内的资源调度

在Spark应用程序（即SparkContext实例）内部，如果多个作业分别由单独的线程提交，那么它们就可以同时并行执行。所谓的作业指的是Spark action（如save、collect等）以及需要执行该action的任何任务。Spark调度器是完全线程安全的，而且支持这种特性能让应用程序响应多个服务请求（比如为多个用户提供查询服务）。

默认情况下，spark调度器以FIFO方式运行作业。每个作业被分成若干个“stage”（比如map和reduce阶段），同时，当第一个作业中的stage有任务要启动时，那么该作业优先获得所有可用资源，接着是第二个作业获得优先权，依次类推。如果队列首位的作业不需要占用集群上的所有资源，那么后续的作业就可以立即启动，但是如果队列首位的作业规模很大，那么后续作业的执行会显著地延迟。

从Spark 0.8版开始，通过配置可以让各个作业公平地共享资源。在公平资源共享模式下，spark以轮转方式在各个作业之间分配任务，以便所有作业都能大致平等地共享集群资源。这就意味着，在长期作业正在运行时提交的短期作业可以立即获得资源并得到良好的响应时间，而无需等待长期作业执行完毕。该模式对于多用户环境是最佳的。

## 3.4 长时间容错

由于Spark的数据导入导出、数据抽象、作业调度3个模块都是分布式的并且长时间运行的，所以需要一个错误响应与处理模块来应对前3个模块运行过程中出现的错误。其中保障数据的产生与输入模块需要进行数据的备份。保障数据流抽象和作业调度这两个模块需要对这两个模块的过程数据设置检查点，在出错时读取检查点信息然后恢复即可。

### 3.4.1 数据导入导出模块长时间容错

在数据导入导出模块，数据接收器如果出错只需要直接重启就可以了。核心是需要保障接收到的数据的安全，保障了源数据，就可以保障后续工作的重现。在这里我们可以按照需求场景的不同来灵活设定对源数据的保障。

* 热备份**：**热备份是指在存储块数据时，将其存储到本执行器、并同时复制到另外一个执行器上去。这样在一个执行器失效后，可以立刻无感知切换到另一份 执行器进行计算。实现方式是，在实现自己的接收器时，设置存储级别。将块数据存储在多个执行器上。
* 冷备份：冷备份是每次存储块数据前，可以先把块数据作为 log 写出到log日志里，再存储到所在的执行器。执行器失效时，就由另外的执行器去读日志信息，再重做 log 来恢复块数据。日志信息通常写到可靠存储如 HDFS上，所以恢复时可能需要花费一段重现时间。
* 数据重发：如果源数据的发出者支持数据重放，例如Apache Kafka，那就可以选择不在数据接收端来备份数据而是出现问题时让源数据的发出者进行数据重发即可。

### 3.4.2 数据抽象模块及作业调度模块的长时间容错

由于数据抽象模块及作业调度模块都是对输入信息进行处理，产生块数据进行执行，所以可以采用上一节提到的冷备份的方式进行备份。当出现问题时，读取到log日志然后重现操作即可。此外，由于这两个模块会产生大量的过程数据。所以可以对整个流程处理设置检查点，来记录两个模块的完成情况。这样可以在出现问题后选择最近一次的检查点进行恢复，最大化的减少损失时间。

## 3.5 窗口支持

为了更有效率的再一组RDD进行计算，spark streaming提供了窗口计算，可以转换滑动窗口内的数据，即可以更灵活的控制DStream的大小（时间间隔大小、数据元素个数）。在设置好窗口的长度，滑动的间隔长度后，窗口在源DStream上滑动。每次滑动，窗口内的RDD将被合并生成窗口DStream内的RDD。

# 4. 非功能性需求

## 4.1性能调优

### 4.1.1合理的批处理时间

在Spark Streaming中需要设置批处理的时间间隔，每隔此时间间隔会提交一个Job。Job之间可能存在依赖关系，后面的Job必须确保前面的Job执行结束才能提交。若前面Job执行时间超过了批处理时间间隔，后面的Job无法按时提交，就会进一步拖延接下来的Job，造成后续Job的阻塞。所以必须设置合理的批处理时间间隔。

### 4.1.2 增加并行化

数据处理并行化，有助于充分利用资源，以提高性能。增加并行化程度，可以通过增加Job的并行度，或者从接收端来增加并行度。

### 4.1.3 及时清除过期数据

Spark Streaming会将接收到的数据全部存储，随着时间推移，有些数据不需要了，这些数据的存储会浪费宝贵的内存资源。所以需要定期清理超时的无用数据。