Flink DataSet API编程指南

Flink最大的亮点是实时处理部分，Flink认为批处理是流处理的特殊情况，可以通过一套引擎处理批量和流式数据，而Flink在未来也会重点投入更多的资源到批流融合中。我在Flink DataStream API编程指南中介绍了DataStream API的使用，在本文中将介绍Flink批处理计算的DataSet API的使用。通过本文你可以了解：

- DataSet转换操作(Transformation)
- Source与Sink的使用
- 广播变量的基本概念与使用Demo
- 分布式缓存的概念及使用Demo
- DataSet API的Transformation使用Demo案例

WordCount示例

在开始讲解DataSet API之前，先看一个Word Count的简单示例，来直观感受一下DataSet API的编程模型，具体代码如下：

```
public class WordCount {
    public static void main(String[] args) throws Exception {
        // 用于批处理的执行环境
        ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvi
ronment();

        // 数据源
        DataSource<String> stringDataSource = env.fromElements("hello Fl
ink What is Apache Flink");

        // 转换
        AggregateOperator<Tuple2<String, Integer>> wordCnt = stringDataS
ource
                .flatMap(new FlatMapFunction<String, Tuple2<String, Integ
er>>() {
                    @Override
                    public void flatMap(String value, Collector<Tuple2<St
ring, Integer>> out) throws Exception {
                        String[] split = value.split(" ");
                        for (String word : split) {

```

```
                            out.collect(Tuple2.of(word, 1));
                        }
                    }
                })
                .groupBy(0)
                .sum(1);
        // 输出
        wordCnt.print();
    }
}
```

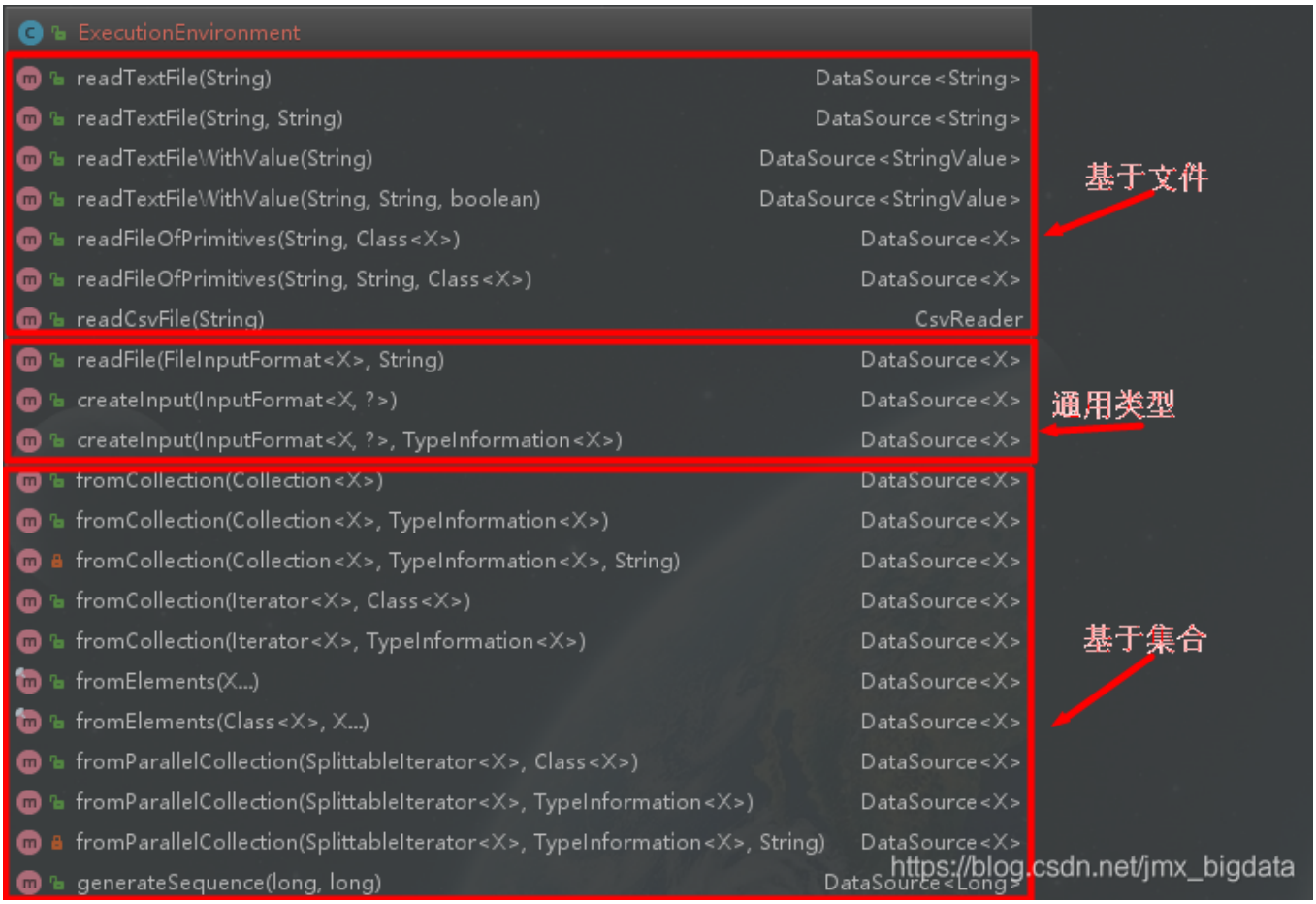
从上面的示例中可以看出，基本的编程模型是：

- 获取批处理的执行环境ExecutionEnvironment
- 加载数据源
- 转换操作
- 数据输出

下面会对数据源、转换操作、数据输出进行一一解读。

Data Source

DataSet API支持从多种数据源中将批量数据集读到Flink系统中，并转换成DataSet数据集。主要包括三种类型：分别是基于文件的、基于集合的及通用类数据源。同时在DataSet API中可以自定义实现InputFormat/RichInputFormat接口，以接入不同数据格式类型的数据源，比如CsvInputFormat、TextInputFormat等。从ExecutionEnvironment类提供的方法中可以看出支持的数据源方法，如下图所示：



基于文件的数据源

readTextFile(path) / TextInputFormat

- 解释

读取文本文件，传递文件路径参数，并将文件内容转换成DataSet类型数据集。

- 使用

```
// 读取本地文件
DataSet<String> localLines = env.readTextFile("file:///path/to/my/textfile");
// 读取HDSF文件
DataSet<String> hdfsLines = env.readTextFile("hdfs://nnHost:nnPort/path/to/my/textfile");
```

readTextFileWithValue(path)/ TextValueInputFormat

- 解释

读取文本文件内容，将文件内容转换成DataSet[StringValue]类型数据集。该方法与readTextFile(String)不同的是，其泛型是StringValue，是一种可变的String类型，通过

StringValue存储文本数据可以有效降低String对象创建数量，减小垃圾回收的压力。

- 使用

```
// 读取本地文件
DataSet<StringValue> localLines = env.readTextFileWithValue("file:///some/local/file");
// 读取HDSF文件
DataSet<StringValue> hdfsLines = env.readTextFileWithValue("hdfs://host:port/file/path");
```

readCsvFile(path)/ CsvInputFormat

- 解释

创建一个CSV的reader，读取逗号分隔(或其他分隔符)的文件。可以直接转换成Tuple类型、POJOs类的DataSet。在方法中可以指定行切割符、列切割符、字段等信息。

- 使用

```
// read a CSV file with five fields, taking only two of them
// 读取一个具有5个字段的CSV文件，只取第一个和第四个字段
DataSet<Tuple2<String, Double>> csvInput = env.readCsvFile("hdfs:///the/CSV/file")
                                    .includeFields("10010")
                                       .types(String.class, Double.class);

// 读取一个有三个字段的CSV文件，将其转为POJO类型
DataSet<Person>> csvInput = env.readCsvFile("hdfs:///the/CSV/file")
                         .pojoType(Person.class, "name", "age", "zipcode");
```

readFileOfPrimitives(path, Class) / PrimitiveInputFormat

- 解释

读取一个原始数据类型(如String,Integer)的文件,返回一个对应的原始类型的DataSet集合

- 使用

```
DataSet<String> Data = env.readFileOfPrimitives("file:///some/local/file", String.class);
```

基于集合的数据源

fromCollection(Collection)

- 解释

从java的集合中创建DataSet数据集，集合中的元素数据类型相同

- 使用

```
DataSet<String> data= env.fromCollection(arrayList);
```

fromElements(T …)

- 解释

从给定数据元素序列中创建DataSet数据集，且所有的数据对象类型必须一致

- 使用

```
 DataSet<String> stringDataSource = env.fromElements("hello Flink What is Apache Flink");
```

generateSequence(from, to)

- 解释

指定from到to范围区间，然后在区间内部生成数字序列数据集,由于是并行处理的，所以最终的顺序不能保证一致。

- 使用

```
DataSet<Long> longDataSource = env.generateSequence(1, 20);
```

通用类型数据源

DataSet API中提供了Inputformat通用的数据接口，以接入不同数据源和格式类型的数据。InputFormat接口主要分为两种类型：一种是基于文件类型，在DataSet API对应readFile()方法；另外一种是基于通用数据类型的接口，例如读取RDBMS或NoSQL数据库中等，在DataSet API中对应createInput()方法。

readFile(inputFormat, path) / FileInputFormat

- 解释

自定义文件类型输入源，将指定格式文件读取并转成DataSet数据集

- 使用

```
env.readFile(new MyInputFormat(), "file:///some/local/file");
```

createInput(inputFormat) / InputFormat

- 解释

自定义通用型数据源，将读取的数据转换为DataSet数据集。如以下实例使用Flink内置的JDBCInputFormat，创建读取mysql数据源的JDBCInput Format，完成从mysql中读取Person表，并转换成DataSet [Row]数据集

- 使用

```
DataSet<Tuple2<String, Integer> dbData =
    env.createInput(
      JDBCInputFormat.buildJDBCInputFormat()
                     .setDrivername("com.mysql.jdbc.Driver")
                     .setDBUrl("jdbc:mysql://localhost/mydb")
                     .setQuery("select name, age from stu")
                     .setRowTypeInfo(new RowTypeInfo(BasicTypeInfo.STRIN
G_TYPE_INFO, BasicTypeInfo.INT_TYPE_INFO))
                     .finish()
    );
```

Data Sink

Flink在DataSet API中的数据输出共分为三种类型。第一种是基于文件实现，对应DataSet的write()方法，实现将DataSet数据输出到文件系统中。第二种是基于通用存储介质实现，对应DataSet的output()方法，例如使用JDBCOutputFormat将数据输出到关系型数据库中。最后一种是客户端输出，直接将DataSet数据从不同的节点收集到Client，并在客户端中输出，例如DataSet的print()方法。

标准的数据输出方法

```
// 文本数据
DataSet<String> textData = // [...]
```

```
3
4  // 将数据写入本地文件
5  textData.writeAsText("file:///my/result/on/localFS");
6
7  // 将数据写入HDFS文件
8  textData.writeAsText("hdfs://nnHost:nnPort/my/result/on/localFS");
9
10 // 写数据到本地文件，如果文件存在则覆盖
11 textData.writeAsText("file:///my/result/on/localFS", WriteMode.OVERWRITE)
12 ;
13
14 // 将数据输出到本地的CSV文件，指定分隔符为"|"
15 DataSet<Tuple3<String, Integer, Double>> values = // [...]
16 values.writeAsCsv("file:///path/to/the/result/file", "\n", "|");
17
18 // 使用自定义的TextFormatter对象
19 values.writeAsFormattedText("file:///path/to/the/result/file",
20     new TextFormatter<Tuple2<Integer, Integer>>() {
21         public String format (Tuple2<Integer, Integer> value) {
22             return value.f1 + " - " + value.f0;
23         }
24     });
```

使用自定义的输出类型

```
1  DataSet<Tuple3<String, Integer, Double>> myResult = [...]
2
3  // 将tuple类型的数据写入关系型数据库
4  myResult.output(
5      // 创建并配置OutputFormat
6      JDBCOutputFormat.buildJDBCOutputFormat()
7                      .setDrivername("com.mysql.jdbc.Driver")
8                      .setDBUrl("jdbc:mysql://localhost/mydb")
9                      .setQuery("insert into persons (name, age, height) v
10 alues (?,?,?)")
11                     .finish()
12     );
```

DataSet转换

转换(transformations)将一个DataSet转成另外一个DataSet，Flink提供了非常丰富的转换操作符。具体使用如下：

Map

一进一出

```
  DataSource<String> source = env.fromElements("I", "like", "flink");
        source.map(new MapFunction<String, String>() {
            @Override
            // 将数据转为大写
            public String map(String value) throws Exception {
                return value.toUpperCase();
            }
        }).print();

```

FlatMap

输入一个元素，产生0个、1个或多个元素

```
stringDataSource
                .flatMap(new FlatMapFunction<String, Tuple2<String, Integ
er>>() {
                    @Override
                    public void flatMap(String value, Collector<Tuple2<St
ring, Integer>> out) throws Exception {
                        String[] split = value.split(" ");
                        for (String word : split) {
                            out.collect(Tuple2.of(word, 1));
                        }
                    }
                })
                .groupBy(0)
                .sum(1);
```

MapPartition

功能和Map函数相似，只是MapPartition操作是在DataSet中基于分区对数据进行处理，函数调用中会按照分区将数据通过Iteator的形式传入，每个分区中的元素数与并行度有关，并返回任意数量的结果值。

```
 source.mapPartition(new MapPartitionFunction<String, Long>() {
            @Override
            public void mapPartition(Iterable<String> values, Collector<
Long> out) throws Exception {
                long c = 0;
                for (String value : values) {
                    c++;
```

```
                }
                //输出每个分区元素个数
                out.collect(c);
            }
        }).print();
```

Filter

过滤数据，如果返回true则保留数据，如果返回false则过滤掉

```
DataSource<Long> source = env.fromElements(1L, 2L, 3L,4L,5L);
        source.filter(new FilterFunction<Long>() {
            @Override
            public boolean filter(Long value) throws Exception {
                return value % 2 == 0;
            }
        }).print();
```

Project

仅能用在Tuple类型的数据集，投影操作，选取Tuple数据的字段的子集

```
  DataSource<Tuple3<Long, Integer, String>> source = env.fromElements(
                Tuple3.of(1L, 20, "tom"),
                Tuple3.of(2L, 25, "jack"),
                Tuple3.of(3L, 22, "bob"));
        // 去第一个和第三个元素
        source.project(0, 2).print();
```

Reduce

通过两两合并，将数据集中的元素合并成一个元素，可以在整个数据集上使用，也可以在分组之后的数据集上使用。

```
DataSource<Tuple2<String, Integer>> source = env.fromElements(
                Tuple2.of("Flink", 1),
                Tuple2.of("Flink", 1),
                Tuple2.of("Hadoop", 1),
                Tuple2.of("Spark", 1),
                Tuple2.of("Flink", 1));
        source
```

```
                .groupBy(0)
                .reduce(new ReduceFunction<Tuple2<String, Integer>>() {
            @Override
            public Tuple2<String, Integer> reduce(Tuple2<String, Integer
> value1, Tuple2<String, Integer> value2) throws Exception {
                return Tuple2.of(value1.f0, value1.f1 + value2.f1);
            }
        }).print();
```

ReduceGroup

将数据集中的元素合并成一个元素，可以在整个数据集上使用，也可以在分组之后的数据集上使用。reduce函数的输入值是一个分组元素的Iterable。

```
DataSource<Tuple2<String, Long>> source = env.fromElements(
                Tuple2.of("Flink", 1L),
                Tuple2.of("Flink", 1L),
                Tuple2.of("Hadoop", 1L),
                Tuple2.of("Spark", 1L),
                Tuple2.of("Flink", 1L));
        source
                .groupBy(0)
                .reduceGroup(new GroupReduceFunction<Tuple2<String,Long>
, Tuple2<String,Long>>() {
                    @Override
                    public void reduce(Iterable<Tuple2<String, Long>> va
lues, Collector<Tuple2<String, Long>> out) throws Exception {
                        Long sum = 0L;
                        String word = "";
                        for(Tuple2<String, Long> value:values){
                            sum += value.f1;
                            word = value.f0;

                        }
                        out.collect(Tuple2.of(word,sum));
                    }
                }).print();
```

Aggregate

通过Aggregate Function将一组元素值合并成单个值，可以在整个DataSet数据集上使用，也可以在分组之后的数据集上使用。仅仅用在Tuple类型的数据集上，主要包括Sum,Min,Max函数

```
DataSource<Tuple2<String, Long>> source = env.fromElements(
                Tuple2.of("Flink", 1L),
                Tuple2.of("Flink", 1L),
                Tuple2.of("Hadoop", 1L),
                Tuple2.of("Spark", 1L),
                Tuple2.of("Flink", 1L));
        source
                .groupBy(0)
                .aggregate(SUM,1)// 按第2个值求和
                 .print();

```

Distinct

DataSet数据集元素去重

```
DataSource<Tuple> source = env.fromElements(Tuple1.of("Flink"),Tuple1.of
("Flink"),Tuple1.of("hadoop"));
        source.distinct(0).print();// 按照tuple的第一个字段去重
// 结果:
(Flink)
(hadoop)
```

Join

默认的join是产生一个Tuple2数据类型的DataSet，关联的key可以通过key表达式、Key-selector函数、字段位置以及CaseClass字段指定。对于两个Tuple类型的数据集可以通过字段位置进行关联，左边数据集的字段通过where方法指定，右边数据集的字段通过equalTo()方法指定。比如：

```
DataSource<Tuple2<Integer,String>> source1 = env.fromElements(
                Tuple2.of(1,"jack"),
                Tuple2.of(2,"tom"),
                Tuple2.of(3,"Bob"));
        DataSource<Tuple2<String, Integer>> source2 = env.fromElements(
                Tuple2.of("order1", 1),
                Tuple2.of("order2", 2),
                Tuple2.of("order3", 3));
        source1.join(source2).where(0).equalTo(1).print();

```

可以在关联的过程中指定自定义Join Funciton, Funciton的入参为左边数据集中的数据元素和右边数据集的中的数据元素所组成的元祖，并返回一个经过计算处理后的数据。

如：

```
// 用户id，购买商品名称，购买商品数量
        DataSource<Tuple3<Integer,String,Integer>> source1 = env.fromEle
ments(
                Tuple3.of(1,"item1",2),
                Tuple3.of(2,"item2",3),
                Tuple3.of(3,"item3",4));
        //商品名称与商品单价
        DataSource<Tuple2<String, Integer>> source2 = env.fromElements(
                Tuple2.of("item1", 10),
                Tuple2.of("item2", 20),
                Tuple2.of("item3", 15));
        source1.join(source2)
                .where(1)
                .equalTo(0)
                .with(new JoinFunction<Tuple3<Integer,String,Integer>, T
uple2<String,Integer>, Tuple3<Integer,String,Double>>() {
                    // 用户每种商品购物总金额
                    @Override
                    public Tuple3<Integer, String, Double> join(Tuple3<I
nteger, String, Integer> first, Tuple2<String, Integer> second) throws Ex
ception {
                        return Tuple3.of(first.f0,first.f1,first.f2 * secon
d.f1.doubleValue());
                    }
                }).print();
```

为了能够更好地引导Flink底层去正确地处理数据集，可以在DataSet数据集关联中，通过Size Hint标记数据集的大小，Flink可以根据用户给定的hint(提示)调整计算策略，例如可以使用joinWithTiny或joinWithHuge提示第二个数据集的大小。示例如下：

```
DataSet<Tuple2<Integer, String>> input1 = // [...]
DataSet<Tuple2<Integer, String>> input2 = // [...]

DataSet<Tuple2<Tuple2<Integer, String>, Tuple2<Integer, String>>>
            result1 =
            // 提示第二个数据集为小数据集
            input1.joinWithTiny(input2)
                  .where(0)
                  .equalTo(0);

DataSet<Tuple2<Tuple2<Integer, String>, Tuple2<Integer, String>>>
            result2 =
            // h提示第二个数据集为大数据集

```

```
            input1.joinWithHuge(input2)
                  .where(0)
                  .equalTo(0);
```

Flink的runtime可以使用多种方式执行join。在不同的情况下，每种可能的方式都会胜过其他方式。系统会尝试自动选择一种合理的方法，但是允许用户手动选择一种策略， 可以让Flink更加灵活且高效地执行Join操作。

```
DataSet<SomeType> input1 = // [...]
DataSet<AnotherType> input2 = // [...]
// 广播第一个输入并从中构建一个哈希表，第二个输入将对其进行探测，适用于第一个数据集非常小的场景
DataSet<Tuple2<SomeType, AnotherType> result =
      input1.join(input2, JoinHint.BROADCAST_HASH_FIRST)
            .where("id").equalTo("key");
// 广播第二个输入并从中构建一个哈希表，第一个输入将对其进行探测，适用于第二个数据集非常小的场景
DataSet<Tuple2<SomeType, AnotherType> result =
      input1.join(input2, JoinHint.BROADCAST_HASH_SECOND)
            .where("id").equalTo("key");
// 将两个数据集重新分区，并将第一个数据集转换成哈希表，适用于第一个数据集比第二个数据集小，但两个数据集都比较大的场景
DataSet<Tuple2<SomeType, AnotherType> result =
      input1.join(input2, JoinHint.REPARTITION_HASH_FIRST)
            .where("id").equalTo("key");
// 将两个数据集重新分区，并将第二个数据集转换成哈希表，适用于第二个数据集比第一个数据集小，但两个数据集都比较大的场景
DataSet<Tuple2<SomeType, AnotherType> result =
      input1.join(input2, JoinHint.REPARTITION_HASH_SECOND)
            .where("id").equalTo("key");
// 将两个数据集重新分区，并将每个分区排序，适用于两个数据集都已经排好序的场景
DataSet<Tuple2<SomeType, AnotherType> result =
      input1.join(input2, JoinHint.REPARTITION_SORT_MERGE)
            .where("id").equalTo("key");
// 相当于不指定，有系统自行处理
DataSet<Tuple2<SomeType, AnotherType> result =
      input1.join(input2, JoinHint.OPTIMIZER_CHOOSES)
            .where("id").equalTo("key");
```

OuterJoin

OuterJoin对两个数据集进行外关联，包含left、right、full outer join三种关联方式，分别对应DataSet API中的leftOuterJoin、rightOuterJoin以及fullOuterJoin方法。注意外连接仅适用于Java 和 Scala DataSet API.

使用方式几乎和join类似：

```
//左外连接
source1.leftOuterJoin(source2).where(1).equalTo(0);
//右外链接
source1.rightOuterJoin(source2).where(1).equalTo(0);

```

此外，外连接也提供了相应的关联算法提示，可以跟据左右数据集的分布情况选择合适的优化策略，提升数据处理的效率。下面代码可以参考上面join的解释。

```
DataSet<SomeType> input1 = // [...]
DataSet<AnotherType> input2 = // [...]
DataSet<Tuple2<SomeType, AnotherType> result1 =
      input1.leftOuterJoin(input2, JoinHint.REPARTITION_SORT_MERGE)
            .where("id").equalTo("key");

DataSet<Tuple2<SomeType, AnotherType> result2 =
      input1.rightOuterJoin(input2, JoinHint.BROADCAST_HASH_FIRST)
            .where("id").equalTo("key");

```

对于外连接的关联算法，与join有所不同。每种外连接只支持部分算法。如下：

- LeftOuterJoin支持：
  - OPTIMIZER_CHOOSES
  - BROADCAST_HASH_SECOND
  - REPARTITION_HASH_SECOND
  - REPARTITION_SORT_MERGE
  - RightOuterJoin支持：
    - OPTIMIZER_CHOOSES
    - BROADCAST_HASH_FIRST
    - REPARTITION_HASH_FIRST
    - REPARTITION_SORT_MERGE
  - FullOuterJoin支持：
    - OPTIMIZER_CHOOSES
    - REPARTITION_SORT_MERGE

CoGroup

CoGroup是对分组之后的DataSet进行join操作，将两个DataSet数据集合并在一起，会先各自对每个DataSet按照key进行分组，然后将分组之后的DataSet传输到用户定义的CoGroupFunction，将两个数据集根据相同的Key记录组合在一起，相同Key的记录会存放在一个Group中，如果指定key仅在一个数据集中有记录，则co-groupFunction会将这个Group与空的Group关联。

```
// 用户id，购买商品名称，购买商品数量
        DataSource<Tuple3<Integer,String,Integer>> source1 = env.fromEle
ments(
                Tuple3.of(1,"item1",2),
                Tuple3.of(2,"item2",3),
                Tuple3.of(3,"item2",4));
        //商品名称与商品单价
        DataSource<Tuple2<String, Integer>> source2 = env.fromElements(
                Tuple2.of("item1", 10),
                Tuple2.of("item2", 20),
                Tuple2.of("item3", 15));

        source1.coGroup(source2)
                .where(1)
                .equalTo(0)
                .with(new CoGroupFunction<Tuple3<Integer,String,Integer>
, Tuple2<String,Integer>, Tuple2<String,Double>>() {
                    // 每个Iterable存储的是分好组的数据，即相同key的数据组织在一
起
                    @Override
                    public void coGroup(Iterable<Tuple3<Integer, String,
 Integer>> first, Iterable<Tuple2<String, Integer>> second, Collector<Tup
le2<String, Double>> out) throws Exception {
                        //存储每种商品购买数量
                        int sum = 0;
                        for(Tuple3<Integer, String, Integer> val1:first){
                        sum += val1.f2;

                    }
                    // 每种商品数量 * 商品单价
                    for(Tuple2<String, Integer> val2:second){
                        out.collect(Tuple2.of(val2.f0,sum * val2.f1.doub
leValue()));

                        }
                    }
                }).print();
```

Cross

将两个数据集合并成一个数据集，返回被连接的两个数据集所有数据行的笛卡儿积，返回的数据行数等于第一个数据集中符合查询条件的数据行数乘以第二个数据集中符合查询条件的数据行数。Cross操作可以通过应用Cross Funciton将关联的数据集合并成目标格式的数据集，如果不指定Cross Funciton则返回Tuple2类型的数据集。Cross操作是计算密集型的算子，建议在使用时加上算法提示，比如
crossWithTiny() and *crossWithHuge()*.

```
//[id,x,y],坐标值
        DataSet<Tuple3<Integer, Integer, Integer>> coords1 = env.fromEle
ments(
                Tuple3.of(1, 20, 18),
                Tuple3.of(2, 15, 20),
                Tuple3.of(3, 25, 10));
        DataSet<Tuple3<Integer, Integer, Integer>> coords2 = env.fromEle
ments(
                Tuple3.of(1, 20, 18),
                Tuple3.of(2, 15, 20),
                Tuple3.of(3, 25, 10));
        // 求任意两点之间的欧氏距离

        coords1.cross(coords2)
                .with(new CrossFunction<Tuple3<Integer, Integer, Integer
>, Tuple3<Integer, Integer, Integer>, Tuple3<Integer, Integer, Double>>(
) {
                    @Override
                    public Tuple3<Integer, Integer, Double> cross(Tuple3
<Integer, Integer, Integer> val1, Tuple3<Integer, Integer, Integer> val2
) throws Exception {
                        // 计算欧式距离
                        double dist = sqrt(pow(val1.f1 - val2.f1, 2) + p
ow(val1.f2 - val2.f2, 2));
                        // 返回两点之间的欧式距离
                        return Tuple3.of(val1.f0,val2.f0,dist);
                    }
                }).print();
```

Union

合并两个DataSet数据集，两个数据集的数据元素格式必须相同，多个数据集可以连续合并.

```
    DataSet<Tuple2<String, Integer>> vals1 = env.fromElements(
                Tuple2.of("jack",20),
```

```
                Tuple2.of("Tom",21));
        DataSet<Tuple2<String, Integer>> vals2 = env.fromElements(
                Tuple2.of("Robin",25),
                Tuple2.of("Bob",30));
        DataSet<Tuple2<String, Integer>> vals3 = env.fromElements(
                Tuple2.of("Jasper",24),
                Tuple2.of("jarry",21));
        DataSet<Tuple2<String, Integer>> unioned = vals1
                .union(vals2)
                .union(vals3);
        unioned.print();

```

Rebalance

对数据集中的数据进行平均分布，使得每个分区上的数据量相同,减轻数据倾斜造成的影响，注意仅仅是`Map-like`类型的算子(比如map，flatMap)才可以用在Rebalance算子之后。

```
DataSet<String> in = // [...]
// rebalance DataSet,然后使用map算子.
DataSet<Tuple2<String, String>> out = in.rebalance()
                                        .map(new Mapper());

```

Hash-Partition

根据给定的Key进行Hash分区，key相同的数据会被放入同一个分区内。可以使用通过元素的位置、元素的名称或者key selector函数指定key。

```
DataSet<Tuple2<String, Integer>> in = // [...]
// 根据第一个值进行hash分区，然后使用 MapPartition转换操作.
DataSet<Tuple2<String, String>> out = in.partitionByHash(0)
                                        .mapPartition(new PartitionMappe
r());

```

Range-Partition

根据给定的Key进行Range分区，key相同的数据会被放入同一个分区内。可以使用通过元素的位置、元素的名称或者key selector函数指定key。

```
DataSet<Tuple2<String, Integer>> in = // [...]
// 根据第一个值进行Range分区，然后使用 MapPartition转换操作.
```

```
DataSet<Tuple2<String, String>> out = in.partitionByRange(0)
                                        .mapPartition(new PartitionMappe
r());
```

Custom Partitioning

除了上面的分区外，还支持自定义分区函数。

```
DataSet<Tuple2<String,Integer>> in = // [...]
DataSet<Integer> result = in.partitionCustom(partitioner, key)
                            .mapPartition(new PartitionMapper());
```

Sort Partition

在本地对DataSet数据集中的所有分区根据指定字段进行重排序，排序方式通过Order.ASCENDING以及Order.DESCENDING关键字指定。支持指定多个字段进行分区排序，如下：

```
DataSet<Tuple2<String, Integer>> in = // [...]
// 按照第一个字段升序排列，第二个字段降序排列.
DataSet<Tuple2<String, String>> out = in.sortPartition(1, Order.ASCENDIN
G)
                                        .sortPartition(0, Order.DESCENDI
NG)
                                        .mapPartition(new PartitionMappe
r());
```

First-n

返回数据集的n条随机结果，可以应用于常规类型数据集、Grouped类型数据集以及排序数据集上。

```
DataSet<Tuple2<String, Integer>> in = // [...]
// 返回数据集中的任意5个元素
DataSet<Tuple2<String, Integer>> out1 = in.first(5);
//返回每个分组内的任意两个元素
DataSet<Tuple2<String, Integer>> out2 = in.groupBy(0)
                                          .first(2);
// 返回每个分组内的前三个元素
// 分组后的数据集按照第二个字段进行升序排序
DataSet<Tuple2<String, Integer>> out3 = in.groupBy(0)
```

```
                                        .sortGroup(1, Order.ASCENDING)
                                        .first(3);

```

MinBy / MaxBy

从数据集中返回指定字段或组合对应最小或最大的记录，如果选择的字段具有多个相同值，则在集合中随机选择一条记录返回。

```
DataSet<Tuple2<String, Integer>> source = env.fromElements(
                Tuple2.of("jack",20),
                Tuple2.of("Tom",21),
                Tuple2.of("Robin",25),
                Tuple2.of("Bob",30));
// 按照第2个元素比较，找出第二个元素为最小值的那个tuple
// 在整个DataSet上使用minBy
ReduceOperator<Tuple2<String, Integer>> tuple2Reduce = source.minBy(1);
tuple2Reduce.print();// 返回(jack,20)

// 也可以在分组的DataSet上使用minBy
source.groupBy(0) // 按照第一个字段进行分组
      .minBy(1)  // 找出每个分组内的按照第二个元素为最小值的那个tuple
      .print();


```

广播变量

基本概念

广播变量是分布式计算框架中经常会用到的一种数据共享方式。其主要作用是将小数据集采用网络传输的方式，在每台机器上维护一个只读的缓存变量，所在的计算节点实例均可以在本地内存中直接读取被广播的数据集，这样能够避免在数据计算过程中多次通过远程的方式从其他节点中读取小数据集，从而提升整体任务的计算性能。

广播变量可以理解为一个公共的共享变量，可以把DataSet广播出去，这样不同的task都可以读取该数据，广播的数据只会在每个节点上存一份。如果不使用广播变量，则会在每个节点上的task中都要复制一份dataset数据集，导致浪费内存。

使用广播变量的基本步骤如下：

```
//第一步创建需要广播的数据集
DataSet<Integer> toBroadcast = env.fromElements(1, 2, 3);

DataSet<String> data = env.fromElements("a", "b");
```

```
5
6  data.map(new RichMapFunction<String, String>() {
7      @Override
8      public void open(Configuration parameters) throws Exception {
9        // 第三步访问集合形式的广播变量数据集
10       Collection<Integer> broadcastSet = getRuntimeContext().getBroadcas
11 tVariable("broadcastSetName");
12     }
13     @Override
14     public String map(String value) throws Exception {
15         ...
16     }
17 }).withBroadcastSet(toBroadcast, "broadcastSetName"); // 第二步广播数据集
```

从上面的代码可以看出，DataSet API支持在RichFunction接口中通过RuntimeContext读取到广播变量。

首先在RichFunction中实现Open()方法，然后调用getRuntimeContext()方法获取应用的RuntimeContext，接着调用getBroadcastVariable()方法通过广播名称获取广播变量。同时Flink直接通过collect操作将数据集转换为本地Collection。需要注意的是，Collection对象的数据类型必须和定义的数据集的类型保持一致，否则会出现类型转换问题。

注意事项：

- 由于广播变量的内容是保存在每个节点的内存中，所以广播变量数据集不易过大。
- 广播变量初始化之后，不支持修改，这样方能保证每个节点的数据都是一样的。
- 如果多个算子都要使用一份数据集，那么需要在多个算子的后面分别注册广播变量。
- 只能在批处理中使用广播变量。

使用Demo

```
1
2  public class BroadcastExample {
3      public static void main(String[] args) throws Exception {
4          ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvi
5 ronment();
6
7          ArrayList<Tuple2<Integer,String>> RawBroadCastData = new ArrayLi
8 st<>();
9
10         RawBroadCastData.add(new Tuple2<>(1,"jack"));
11         RawBroadCastData.add(new Tuple2<>(2,"tom"));
12         RawBroadCastData.add(new Tuple2<>(3,"Bob"));
13
```

```
        // 模拟数据源，[userId,userName]
        DataSource<Tuple2<Integer, String>> userInfoBroadCastData = env.
fromCollection(RawBroadCastData);

        ArrayList<Tuple2<Integer,Double>> rawUserAount = new ArrayList<>
();

        rawUserAount.add(new Tuple2<>(1,1000.00));
        rawUserAount.add(new Tuple2<>(2,500.20));
        rawUserAount.add(new Tuple2<>(3,800.50));

        // 处理数据：用户id，用户购买金额 ，[UserId,amount]
        DataSet<Tuple2<Integer, Double>> userAmount = env.fromCollection
(rawUserAount);

        // 转换为map集合类型的DataSet
        DataSet<HashMap<Integer, String>> userInfoBroadCast = userInfoBr
oadCastData.map(new MapFunction<Tuple2<Integer, String>, HashMap<Integer
, String>>() {

            @Override
            public HashMap<Integer, String> map(Tuple2<Integer, String>
value) throws Exception {
                HashMap<Integer, String> userInfo = new HashMap<>();
                userInfo.put(value.f0, value.f1);
                return userInfo;
            }
        });

       DataSet<String> result = userAmount.map(new RichMapFunction<Tuple
2<Integer, Double>, String>() {
            // 存放广播变量返回的list集合数据
            List<HashMap<String, String>> broadCastList = new ArrayList<
>();
            // 存放广播变量的值
            HashMap<String, String> allMap = new HashMap<>();

            @Override

            public void open(Configuration parameters) throws Exception {
                super.open(parameters);
                //获取广播数据,返回的是一个list集合
                this.broadCastList = getRuntimeContext().getBroadcastVar
iable("userInfo");
                for (HashMap<String, String> value : broadCastList) {
                    allMap.putAll(value);
                }
            }

```

```
                @Override
                public String map(Tuple2<Integer, Double> value) throws Exce
    ption {
                    String userName = allMap.get(value.f0);
                    return "用户id:  " + value.f0 + " | "+ "用户名:  " + userNa
    me + " | " + "购买金额:  " + value.f1;
                }
            }).withBroadcastSet(userInfoBroadCast, "userInfo");

            result.print();
        }

    }
```

分布式缓存

基本概念

Flink提供了一个分布式缓存(distributed cache),类似于Hadoop，以使文件在本地可被用户函数的并行实例访问。分布式缓存的工作机制是为程序注册一个文件或目录(本地或者远程文件系统，如HDFS等)，通过ExecutionEnvironment注册一个缓存文件，并起一个别名。当程序执行的时候，Flink会自动把注册的文件或目录复制到所有TaskManager节点的本地文件系统，用户可以通过注册是起的别名来查找文件或目录，然后在TaskManager节点的本地文件系统访问该文件。

分布式缓存的使用步骤：

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment(
);
// 注册一个HDFS文件
env.registerCachedFile("hdfs:///path/to/your/file", "hdfsFile")
// 注册一个本地文件
env.registerCachedFile("file:///path/to/exec/file", "localExecFile", true)
// 访问数据
getRuntimeContext().getDistributedCache().getFile("hdfsFile");
```

获取缓存文件的方式和广播变量相似，也是实现RichFunction接口，并通过RichFunction接口获得RuntimeContext对象，然后通过RuntimeContext提供的接口获取对应的本地缓存文件。

使用Demo

```
public class DistributeCacheExample {
    public static void main(String[] args) throws Exception {
        //获取运行环境
        ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvi
ronment();
        /**
         *  注册一个本地文件
         *   文件内容为:
         *   1,"jack"
         *   2,"tom"
         *   3,"Bob"
         */
        env.registerCachedFile("file:///E://userinfo.txt", "localFileUser
Info", true);

        ArrayList<Tuple2<Integer,Double>> rawUserAount = new ArrayList<>
();

        rawUserAount.add(new Tuple2<>(1,1000.00));
        rawUserAount.add(new Tuple2<>(2,500.20));
        rawUserAount.add(new Tuple2<>(3,800.50));

        // 处理数据:用户id,用户购买金额 , [UserId,amount]
        DataSet<Tuple2<Integer, Double>> userAmount = env.fromCollection
(rawUserAount);

        DataSet<String> result= userAmount.map(new RichMapFunction<Tuple
2<Integer, Double>, String>() {
            // 保存缓存数据
            HashMap<String, String> allMap = new HashMap<String, String>
();

            @Override
            public void open(Configuration parameters) throws Exception {
                super.open(parameters);
                // 获取分布式缓存的数据
                File userInfoFile = getRuntimeContext().getDistributedCa
che().getFile("localFileUserInfo");
                List<String> userInfo = FileUtils.readLines(userInfoFile
);
                for (String value : userInfo) {

                    String[] split = value.split(",");
                    allMap.put(split[0], split[1]);
                }

            }

```

```
            @Override
            public String map(Tuple2<Integer, Double> value) throws Exce
ption {
                String userName = allMap.get(value.f0);

                return "用户id:  " + value.f0 + " | " + "用户名:  " + userN
ame + " | " + "购买金额:  " + value.f1;
            }
        });

        result.print();

    }
}
```

小结

本文主要讲解了Flink DataSet API的基本使用。首先介绍了一个DataSet API的WordCount案例，接着介绍了DataSet API的数据源与Sink操作，以及基本的使用。然后对每一个转换操作进行了详细的解释，并给出了具体的使用案例。最后讲解了广播变量和分布式缓存的概念，并就如何使用这两种高级功能，提供了完整的Demo案例。