**Apache Crunch**

|  |  |
| --- | --- |
| **审核人** |  |
| **重要性** | 中 |
| **紧迫性** | 中 |
| **拟制人** | 张包峰 |
| **提交日期** |  |

**作者：张包峰**

(版权所有,翻版必究)

**修改记录**

|  |  |  |
| --- | --- | --- |
| **更新时间** | **变更内容** | **变更理由** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

[**作者：张包峰** 1](#_Toc395220368)

[1.1 Crunch设计 4](#_Toc395220369)

[1.1.1 数据模型、操作符、基础类 4](#_Toc395220370)

[1.1.2 DoFn处理数据 5](#_Toc395220371)

[1.1.3 利用PTypes序列化数据 6](#_Toc395220372)

[1.1.4 数据读写 7](#_Toc395220373)

[1.1.5 数据处理原语 9](#_Toc395220374)

[1.2 计算相关 11](#_Toc395220375)

[1.2.1 Aggregators实现 11](#_Toc395220376)

[1.2.2 Join实现 12](#_Toc395220377)

[1.3 数据相关 12](#_Toc395220378)

[1.3.1 Source实现 12](#_Toc395220379)

[1.4 pipeline实现 12](#_Toc395220380)

[1.4.1 MRPipeline 13](#_Toc395220381)

[1.4.2 SparkPipeline 13](#_Toc395220382)

[1.4.3 MemPipeline 13](#_Toc395220383)

[1.5 With HBase 13](#_Toc395220384)

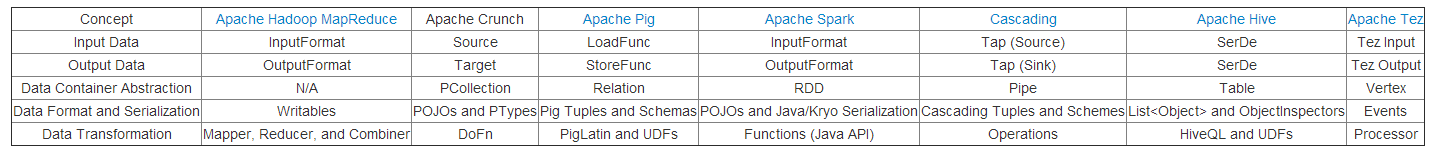
[1.6 Spark Support 13](#_Toc395220385)

[1.7 scrunch 14](#_Toc395220386)

## Crunch设计

按照<http://crunch.apache.org/user-guide.html>梳理一遍

七种Hadoop之上的计算表示层对比：

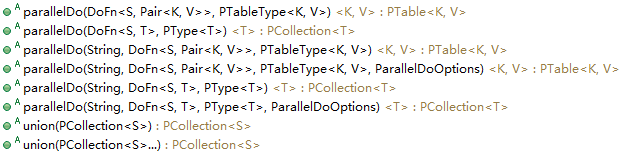


### 数据模型、操作符、基础类

三种分布式数据集的抽象接口：PCollection，PTable，PGroupedTable

* PCollection<T>代表分布式、不可变的数据集，提供parallelDo和union方法，触发对每个元素进行DoFn操作，返回新的PCollection<U>
* PTable<K, V>是PCollection<Pair<K, V>>实现，代表分布式、未排序的multimap。除了继承自PCollection 的parallelDo，还复写了union方法，提供了groupByKey方法。groupByKey方法对应MapReduce job里的排序阶段。在groupByKey操作里，开发者可以在shuffle过程里(参见GroupingOptions类)做细粒度的reducer数目、分区策略、分组策略以及排序策略控制
* PGroupedTable<K, V>是groupByKey操作的结果，代表分布式、排过序的map，具备迭代器，其实现是PCollection<Pair<K, Iterable<V>>>。除了继承自PCollection 的parallelDo、union，提供combineValues方法，允许在shuffle的map端或reduce端使用满足交换律和结合律的聚合算子(参见Aggregator类)作用于PGroupedTable实例的values上

PCollection里的两种基本原语接口：



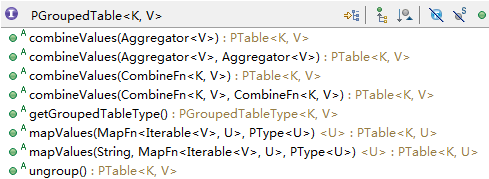
org.apache.crunch.lib里面的其他数据转换操作都来自于上面四种原语。

PCollection提供的其他方法：

count(), min(), max(), aggregate(Aggregator)

filter(), cache()

PTable提供的方法：



PObject<T>，同FlumeJava设计，用于存储Java对象，物化过了之后可以使用getValue()方法获得PObject的值。

数据从Source流入，经过pipeline处理，最后从Target输出。

提供了三种pipeline，分别是MRPipeline，MemPipeline，SparkPipeline

### DoFn处理数据

hadoop的MapReduce job，通过配置job.xml，set本次job的map和reduce class，反射出具体类。Crunch的做法是使用java的序列化把DoFn序列化(DoFn实现了java.io.Serializable)，在pipeline里传输到task上并被调用。使用的时候要注意可序列化(某些情况下使用transient，static等方式)，特别是在MRPipeline和SparkPipeline环境下。

DoFn允许访问TaskInputOutputContext(Hadoop里task的一个上下文类)里的内容，且DoFn可以是在map里，也可以在reduce里。在执行的时候，首先触发initialize方法，类似Mapper, Reducer里的setup方法，比如如果使用了第三方非序列化的类，就可以在此处先实例化出来(声明为transient)。然后是process方法，结果被Emitter发射出去，比如传递给下一个DoFn。最后当所有输入被处理后，执行cleanup，一方面可以最后把一些状态传递给下一个stage，另一方面用于释放资源。

DoFn还有一些和hadoop mr比较类似的地方，比如increment，context和configuration的set/get，还有scaleFactor这个设置，用于估计处理完后数据量的大小，可以影响任务执行的优化(比如判断Reduce个数、I/O量)。默认scaleFactor是0.99，子类会复写这个值，在下面会提到。

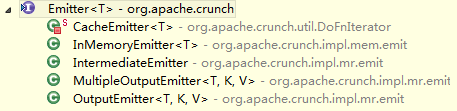
#### FilterFn，MapFn，CombineFn子类

常见的DoFn有FilterFn，MapFn，CombineFn，比较方便使用和测试，几个基础抽象类里都有使用到。

DoFn<S, T>的Process方法做实际的执行逻辑，

**public** **abstract** **void** process(**S** input, Emitter<**T**> emitter);

Emitter对应输出，子类体系如下



FilterFn<T>继承DoFn<T, T>，需要实现其accept(T input)方法，返回boolean，它的process方法会调用accept来判断是否输出。

**public** **void** process(**T** input, Emitter<**T**> emitter) {

**if** (accept(input)) {

emitter.emit(input);

}

}

PCollection的Filter方法就是传入一个FilterFn的实现。FilterFn有and，or，not等子类，定义在FilterFns里。scaleFactor为0.5，很好理解。

MapFn<T>继承DoFn<S, T>，需要实现其map(S input)方法，返回T，process方法里调用如下：

**public** **void** process(**S** input, Emitter<**T**> emitter) {

emitter.emit(map(input));

}

scaleFactor为1.0。

CombineFn继承DoFn<Pair<S, Iterable<T>>, Pair<S, T>>，用于在reduce执行前处理map输出，减少shuffle过程的网络开销，与PGroupedTable里的combineValues()绑定使用。CombineFn常常和Aggregator的实现子类结合使用。

### 利用PTypes序列化数据

本节内容对应的package为org.apache.crunch.types，都是和类型相关的类。

PType<T>定义了数据的序列化和反序列化方式，在PCollection的parallelDo里面使用，如最简单的：

<**T**> PCollection<**T**> parallelDo(DoFn<**S**, **T**> doFn, PType<**T**> type);

由于PCollection<T>范型的设计，T被类型擦除(type erasure)了，以上的output类型需要和指定的PType相符，类似于这样：

PCollection<String> lines = ...;

lines.parallelDo(new DoFn<String, Integer>() { ... }, Writables.ints());

Crunch设置了两种PTypeFamily，一种是hadoop的writable，另一种是Avro。 Crunch还是比较靠拢Apache Hadoop的MR的(至少在Spark出现之前，也只能为Hadoop MR做pipeline吧)。

PTypeFamily提供一些基础的类型是这样的，

PType<Void> nulls();

PType<String> strings();

PType<Long> longs();

PType<Integer> ints();

PType<Float> floats();

PType<Double> doubles();

PType<Boolean> booleans();

PType<ByteBuffer> bytes();

为了适应PTable，PType有另一个子类体系，PTableType<K, V>，继承PType<Pair<K, V>>。

PType，PTableType的Avro、Writable类型的构造、扩展方式就不说明了。

### 数据读写

大部分数据读写格式是hadoop inputFormat/outputFormat那套，简单介绍下主要类和类型。

本节内容对应的package为org.apache.crunch.io，都是和数据读写相关的类。

#### Source

Source<T>和TableSource<K, V>代表数据源，分别对应PCollection和PTable。

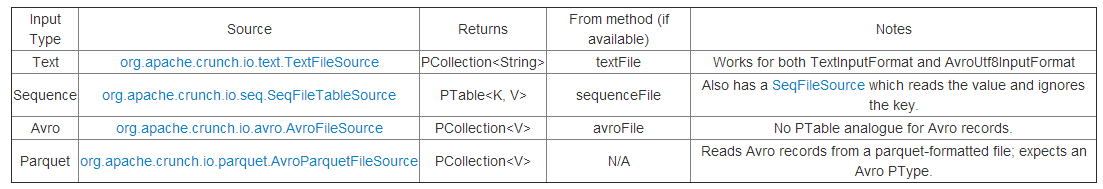
在Pipelie的read方法里使用。

<**T**> PCollection<**T**> read(Source<**T**> source);

<**K**, **V**> PTable<**K**, **V**> read(TableSource<**K**, **V**> tableSource);

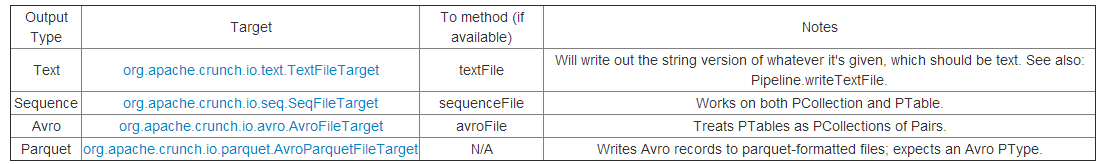
有一个org.apache.crunch.io.From类，定义了一些静态方法，用于读取数据源的时候指定数据格式和类型，比如Writable，然后返回Source或TableSource。

比较常用的Source对应的Input类型如下：



#### Target

Target的定义和Source类似，主要是在Pipeline的write方法里使用，常用的类型如下：



Target具备一些不同的WriteMode，是个枚举类，如下例子：

PCollection<String> lines = ...;

// The default option is to fail if the output path already exists.

lines.write(At.textFile("/user/crunch/out"), WriteMode.DEFAULT);

// Delete the output path if it already exists.

lines.write(At.textFile("/user/crunch/out"), WriteMode.OVERWRITE);

// Add the output of the given PCollection to the data in the path

// if it already exists.

lines.write(At.textFile("/user/crunch/out"), WriteMode.APPEND);

// Use this directory as a checkpoint location, which requires that this

// be a SourceTarget, not just a Target:

lines.write(At.textFile("/user/crunch/out"), WriteMode.CHECKPOINT);

有一个SourceTarget<T>类比较特殊，同时继承了Source<T>和Target，既可充当输入源，又可充当输出地。

#### 数据的物化

PCollection有一个物化的方法，

/\*\*

\* Returns a reference to the data set represented by this PCollection that

\* may be used by the client to read the data locally.

\*/

Iterable<**S**> materialize();

是延迟触发的。

### 数据处理原语

本节介绍org.apache.crunch.lib包下的数据处理模型类，算是advanced原语。

#### groupByKey

PTable的三个groupByKey方法控制了数据的shuffle和处理过程，

PGroupedTable<**K**, **V**> groupByKey();

PGroupedTable<**K**, **V**> groupByKey(**int** numPartitions);

PGroupedTable<**K**, **V**> groupByKey(GroupingOptions options);

第一个是最简单的shuffle，输出的paritition数目会由planner估计数据大小而设置。

第三个方法里的GroupingOptions对groupByKey提供了更多细粒度的控制，包括数据如何分区、如何排序、如何分组。

如果下面执行引擎是hadoop，那么会使用hadoop的Partitiner、RawComparator来做分区和排序。

GroupingOptions是不可变的，通过GroupingOptions.Builder构建出来使用：

GroupingOptions opts = GroupingOptions.builder()

.groupingComparatorClass(MyGroupingComparator.class)

.sortComparatorClass(MySortingComparator.class)

.partitionerClass(MyPartitioner.class)

.numReducers(N)

.conf("key", "value")

.conf("other key", "other value")

.build();

PTable<String, Long> kv = ...;

PGroupedTable<String, Long> kv.groupByKey(opts);

#### combineValues

PTable通过groupByKey得到PGroupedTable，它的combineValues可以让planner控制在shuffle的前后对数据做一些聚合函数的处理。

利用Aggregators的静态方法，使用简单聚合函数的实现类：

PTable<String, Double> data = ...;

// Sum the values of the doubles for each key.

PTable<String, Double> sums =

data.groupByKey().combineValues(Aggregators.SUM\_DOUBLES());

// Find the ten largest values for each key.

PTable<String, Double> maxes =

data.groupByKey().combineValues(Aggregators.MAX\_DOUBLES(10));

PTable<String, String> text = ...;

// Get a random sample of 100 unique elements for each key.

PTable<String, String> samp =

text.groupByKey().combineValues(Aggregators.SAMPLE\_UNIQUE\_ELEMENTS(100));

#### simple aggregations

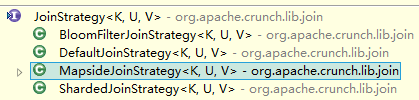
参考Aggregator的实现类。

#### join

支持inner join, left outer join, right outer join, full outer join。定义在枚举类JoinType里。由JoinStrategy执行join动作，

PTable<**K**, Pair<**U**,**V**>> join(PTable<**K**, **U**> left, PTable<**K**, **V**> right, JoinType joinType);

JoinStrategy实现类有



**Reduce-side Joins**

对应DefaultStrategy类，是hadoop里比较简单和鲁棒的join，来自两处input的处理后的数据都shuffle到同一个reducer上，比较小的那份数据被收集起来，与流进来的比较大的那份数据进行join。

**Map-side Joins**

对应MapsideJoinStrategy类，比较小的那份数据需要load到内存里，需要保证比较小的那个table能够被缓存在各task的内存里。

**Sharded Joins**

对应ShardedJoinStrategy类，允许把相同key的数据，分区到多个reducer上，避免某些reducer上数据量过大，因为很多分布式join会有数据倾斜的问题，导致某些reducer会出现内存不够的情况。

**Bloom Filter Joins**

对应BloomFilterStrategy类，适合左侧table数据量太大，但仍远小于右侧table数据量，且右侧table的大多数key无法匹配左侧table数据的情况。

#### cogroups

Crunch的cogroup与Pig里的cogroup类似，接受多份PTable，根据相同的key，输出一个个bag。cogroup处理类似join的第一步。

#### sorting

略。

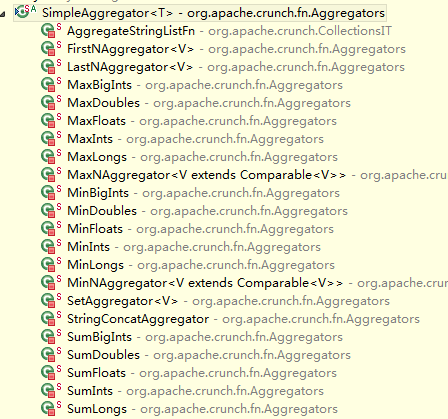
#### others

Cartisian、Coalescing、Distinct、Sampling、Set Operations等。

## 计算相关

### Aggregators实现

不需要初始化的简单聚合函数实现体系：SimpleAggregator



实现方式很简单，复写update方法的时候，做一些比较、求和等操作，记录到全局变量里。或者把结果记录在一个SortedSet里，返回的时候可以取topN。

### Join实现

## 数据相关

### Source实现

## 跟代码

## pipeline实现

Pipeline的read、write、cache、materlize方法

Pipiline run()触发执行，返回PipelineResult，包含本次任务stages的信息。

### MRPipeline

run()依赖runAsync()

new一个MSCRPlanner来生成和优化执行计划

MSCRPlanner是MRPipeline的

### SparkPipeline

### MemPipeline

## With HBase

## Spark Support

利用JavaSparkContext

执行在SparkRuntime里

在一些小的点上做了些额外的事情

1. 添加了Writable和Avro的serde(见org.apache.crunch.impl.spark.serde)
2. 多路输入
3. 多路输出
4. checkpiont

## scrunch