> 宁静致远，淡泊明志

FlinkAPI之数据类型和UDF函数

支持的数据类型

> Flink流应用程序处理的是**以数据对象表示的事件流**。所以在Flink内部，我们需要能够处理这些对象。它们需要被**序列化和反序列化**，以便通过网络传送它们；或者从**状态后端、检查点和保存点**读取它们。为了有效地做到这一点，Flink需要明确知道应用程序所处理的数据类型。Flink使用类型信息的概念来表示数据类型，并为每个数据类型生成特定的序列化器、反序列化器和比较器。

Flink还具有一个类型提取系统，该系统分析函数的输入和返回类型，以自动获取类型信息，从而获得序列化器和反序列化器。但是，在某些情况下，例如lambda函数或泛型类型，需要显式地提供类型信息，才能使应用程序正常工作或提高其性能。

Flink支持Java和Scala中所有常见数据类型。使用最广泛的类型有以下几种。

基础数据类型

Flink支持所有的Java和Scala基础数据类型，Int, Double, Long, String, ...

```
val numbers: DataStream[Long] = env.fromElements(1L, 2L, 3L, 4L)
numbers.map( n => n + 1 )
```

Java和Scala元组（Tuples）

```
val persons: DataStream[(String, Integer)] = env.fromElements(
("Adam", 17),
("Sarah", 23) )
persons.filter(p => p._2 > 18)
```

Scala样例类（case classes）

```
case class Person(name: String, age: Int)
val persons: DataStream[Person] = env.fromElements(
Person("Adam", 17),
Person("Sarah", 23) )
persons.filter(p => p.age > 18)
```

Java简单对象（POJOs）

```
public class Person {
public String name;
public int age;
  public Person() {}
  public Person(String name, int age) {
this.name = name;
this.age = age;
}
}
DataStream<Person> persons = env.fromElements(
new Person("Alex", 42),
new Person("Wendy", 23));
```

其它数据类型

> Flink对Java和Scala中的一些特殊目的的类型也都是支持的，比如Java的ArrayList，HashMap，Enum等等。

UDF函数

> 和Spark和Hive一样，flink也实现了用户自定义函数

函数类（Function Classes）

> Flink暴露了所有udf函数的接口(实现方式为接口或者抽象类)。例如MapFunction, FilterFunction, ProcessFunction等等。

下面例子实现了FilterFunction接口：

```
class FilterFilter extends FilterFunction[String] {
      override def filter(value: String): Boolean = {
        value.contains("flink")
      }
}
val flinkTweets = tweets.filter(new FlinkFilter)
```

还可以将函数实现成匿名类

```
val flinkTweets = tweets.filter(
new RichFilterFunction[String] {
    override def filter(value: String): Boolean = {
        value.contains("flink")
}
}
)
```

可以将filter的字符串"flink"还可以当作参数传进去。

```
val tweets: DataStream[String] = ...
val flinkTweets = tweets.filter(new KeywordFilter("flink"))

class KeywordFilter(keyWord: String) extends FilterFunction[String] {
    override def filter(value: String): Boolean = {
        value.contains(keyWord)
}
}
```

匿名函数（Lambda Functions）

```
val tweets: DataStream[String] = ...
val flinkTweets = tweets.filter(_.contains("flink"))
```

富函数（Rich Functions）【重点掌握】

> “富函数”是DataStream API提供的一个函数类的接口，所有Flink函数类都有其Rich版本。它与常规函数的不同在于，可以获取运行环境的上下文，并拥有一些生命周期方法，所以可以实现更复杂的功能。

- RichMapFunction
- RichFlatMapFunction
- RichFilterFunction

Rich Function有一个生命周期的概念。典型的生命周期方法有：

- open()方法是rich function的初始化方法，当一个算子例如map或者filter被调用之前open()会被调用。
- close()方法是生命周期中的最后一个调用的方法，做一些清理工作。
- getRuntimeContext()方法提供了函数的RuntimeContext的一些信息，例如函数执行的并行度，任务的名字，以及state状态

```
class MyFlatMap extends RichFlatMapFunction[Int, (Int, Int)] {
var subTaskIndex = 0

override def open(configuration: Configuration): Unit = {
    subTaskIndex = getRuntimeContext.getIndexOfThisSubtask
        // 以下可以做一些初始化工作，例如建立一个和HDFS的连接
}

override def flatMap(in: Int, out: Collector[(Int, Int)]): Unit = {
    if (in % 2 == subTaskIndex) {
        out.collect((subTaskIndex, in))
}
}

override def close(): Unit = {
// 以下做一些清理工作，例如断开和HDFS的连接。
}
}
```