Flink (//ci.apache.org/projects/flink/flink-docs-release-1.10/zh/) v1.10

DataStream API

Apache Flink 提供了 DataStream API 来实现稳定可靠的、有状态的流处理应用程序。 Flink 支持对状态和时间的细粒度控制，以此来实现复杂的事件驱动数据处理系统。 这个入门指导手册讲述了如何通过 Flink DataStream API 来实现一个有状态流处理程序。

你要搭建一个什么系统

在当今数字时代，信用卡欺诈行为越来越被重视。 罪犯可以通过诈骗或者入侵安全级别较低系统来盗窃信用卡卡号。用盗得的信用卡进行很小额度的例如一美元或者更小额度的消费进行测试。 如果测试消费成功，那么他们就会用这个信用卡进行大笔消费，来购买一些他们希望得到的，或者可以倒卖的财物。

在这个教程中，你将会建立一个针对可疑信用卡交易行为的反欺诈检测系统。 通过使用一组简单的规则，你将了解到 Flink 如何为我们实现复杂业务逻辑并实时执行。

准备条件

这个代码练习假定你对 Java 或 Scala 有一定的了解，当然，如果你之前使用的是其他开发语言，你也应该能够跟随本教程进行学习。

困难求助

如果遇到困难，可以参考 社区支持资源 (https://flink.apache.org/zh/gettinghelp.html)。 当然也可以在邮件列表提问，Flink 的 用户邮件列表 (https://flink.apache.org/zh/community.html#mailing-lists) 一直被评为所有Apache项目中最活跃的一个，这也是快速获得帮助的好方法。

怎样跟着教程练习

首先，你需要在你的电脑上准备以下环境：

- Java 8 or 11
- Maven

一个准备好的 Flink Maven Archetype 能够快速创建一个包含了必要依赖的 Flink 程序骨架，基于此，你可以把精力集中在编写业务逻辑上即可。 这些已包含的依赖包括 `flink-streaming-java`、`flink-walkthrough-common` 等，他们分别是 Flink 应用程序的核心依赖项和这个代码练习需要的数据生成器，当然还包括其他本代码练习所依赖的类。

> **说明:** 为简洁起见，本练习中的代码块中可能不包含完整的类路径。完整的类路径可以在文档底部 链接 中找到。

Java **Scala**

```
$ mvn archetype:generate \
    -DarchetypeGroupId=org.apache.flink \
    -DarchetypeArtifactId=flink-walkthrough-datastream-java \
    -DarchetypeVersion=1.10.0 \
    -DgroupId=frauddetection \
    -DartifactId=frauddetection \
    -Dversion=0.1 \
    -Dpackage=spendreport \
    -DinteractiveMode=false
```

你可以根据自己的情况修改 `groupId`、 `artifactId` 和 `package`。通过这三个参数， Maven 将会创建一个名为 `frauddetection` 的文件夹，包含了所有依赖的整个工程项目将会位于该文件夹下。 将工程目录导入到你的开发环境之后，你可以找到 `FraudDetectionJob.java` （或 `FraudDetectionJob.scala`） 代码文件，文件中的代码如下所示。你可以在 IDE 中直接运行这个文件。 同时，你可以试着在数据流中设置一些断点或者以 DEBUG 模式来运行程序，体验 Flink 是如何运行的。

Java **Scala**

FraudDetectionJob.java

```
package spendreport;

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.walkthrough.common.sink.AlertSink;
import org.apache.flink.walkthrough.common.entity.Alert;
import org.apache.flink.walkthrough.common.entity.Transaction;
import org.apache.flink.walkthrough.common.source.TransactionSource;

public class FraudDetectionJob {

    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        DataStream<Transaction> transactions = env
            .addSource(new TransactionSource())
            .name("transactions");

        DataStream<Alert> alerts = transactions
            .keyBy(Transaction::getAccountId)
            .process(new FraudDetector())
            .name("fraud-detector");

        alerts
            .addSink(new AlertSink())
            .name("send-alerts");

        env.execute("Fraud Detection");
    }
}
```

FraudDetector.java

```
package spendreport;

import org.apache.flink.streaming.api.functions.KeyedProcessFunction;
import org.apache.flink.util.Collector;
import org.apache.flink.walkthrough.common.entity.Alert;
import org.apache.flink.walkthrough.common.entity.Transaction;

public class FraudDetector extends KeyedProcessFunction<Long, Transaction, Alert> {

    private static final long serialVersionUID = 1L;

    private static final double SMALL_AMOUNT = 1.00;
    private static final double LARGE_AMOUNT = 500.00;
    private static final long ONE_MINUTE = 60 * 1000;

    @Override
    public void processElement(
            Transaction transaction,
            Context context,
            Collector<Alert> collector) throws Exception {

        Alert alert = new Alert();
        alert.setId(transaction.getAccountId());

        collector.collect(alert);
    }
}
```

代码分析

让我们一步步地来分析一下这两个代码文件。FraudDetectionJob 类定义了程序的数据流，而 FraudDetector 类定义了欺诈交易检测的业务逻辑。

下面我们开始讲解整个 Job 是如何组装到 FraudDetectionJob 类的 main 函数中的。

执行环境

第一行的 StreamExecutionEnvironment 用于设置你的执行环境。 任务执行环境用于定义任务的属性、创建数据源以及最终启动任务的执行。

Java | Scala

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
```

创建数据源

数据源从外部系统例如 Apache Kafka、Rabbit MQ 或者 Apache Pulsar 接收数据，然后将数据送到 Flink 程序中。 这个代码练习使用的是一个能够无限循环生成信用卡模拟交易数据的数据源。 每条交易数据包括了信用卡 ID （accountId），交易发生的时间 （timestamp） 以及交易的金额（amount）。 绑定到数据源上的 name 属性是为了调试方便，如果发生一些异常，我们能够通过它快速定位问题发生在哪里。

Java **Scala**

```
DataStream<Transaction> transactions = env
    .addSource(new TransactionSource())
    .name("transactions");
```

对事件分区 & 欺诈检测

transactions 这个数据流包含了大量的用户交易数据，需要被划分到多个并发上进行欺诈检测处理。由于欺诈行为的发生是基于某一个账户的，所以，必须要要保证同一个账户的所有交易行为数据要被同一个并发的 task 进行处理。

为了保证同一个 task 处理同一个 key 的所有数据，你可以使用 DataStream#keyBy 对流进行分区。 process() 函数对流绑定了一个操作，这个操作将会对流上的每一个消息调用所定义好的函数。 通常，一个操作会紧跟着 keyBy 被调用，在这个例子中，这个操作是FraudDetector，该操作是在一个 *keyed context* 上执行的。

Java **Scala**

```
DataStream<Alert> alerts = transactions
    .keyBy(Transaction::getAccountId)
    .process(new FraudDetector())
    .name("fraud-detector");
```

输出结果

sink 会将 DataStream 写出到外部系统，例如 Apache Kafka、Cassandra 或者 AWS Kinesis 等。 AlertSink 使用 **INFO** 的日志级别打印每一个 Alert 的数据记录，而不是将其写入持久存储，以便你可以方便地查看结果。

Java **Scala**

```
alerts.addSink(new AlertSink());
```

运行作业

Flink 程序是懒加载的，并且只有在完全搭建好之后，才能够发布到集群上执行。 调用 StreamExecutionEnvironment#execute 时给任务传递一个任务名参数，就可以开始运行任务。

Java **Scala**

```
env.execute("Fraud Detection");
```

欺诈检测器

欺诈检查类 `FraudDetector` 是 `KeyedProcessFunction` 接口的一个实现。 他的方法 `KeyedProcessFunction#processElement` 将会在每个交易事件上被调用。 这个程序里边会对每笔交易发出警报，有人可能会说这做报过于保守了。

本教程的后续步骤将指导你对这个欺诈检测器进行更有意义的业务逻辑扩展。

Java **Scala**

```
public class FraudDetector extends KeyedProcessFunction<Long, Transaction, Alert> {

    private static final double SMALL_AMOUNT = 1.00;
    private static final double LARGE_AMOUNT = 500.00;
    private static final long ONE_MINUTE = 60 * 1000;

    @Override
    public void processElement(
            Transaction transaction,
            Context context,
            Collector<Alert> collector) throws Exception {

        Alert alert = new Alert();
        alert.setId(transaction.getAccountId());

        collector.collect(alert);
    }
}
```

实现一个真正的应用程序

我们先实现第一版报警程序，对于一个账户，如果出现小于 $1 美元的交易后紧跟着一个大于 $500 的交易，就输出一个报警信息。

假设你的欺诈检测器所处理的交易数据如下：

Txn 1	Txn 2	Txn 3	Txn 4	Txn 5	Txn 6	Txn 7	Txn 8	Txn 9	Txn 10
$13.01	$25.00	$0.09	$510.00	$102.62	$91.50	$0.02	$30.01	$701.83	$31.92

交易 3 和交易 4 应该被标记为欺诈行为，因为交易 3 是一个 $0.09 的小额交易，而紧随着的交易 4 是一个 $510 的大额交易。 另外，交易 7、8 和 交易 9 就不属于欺诈交易了，因为在交易 7 这个 $0.02 的小额交易之后，并没有跟随一个大额交易，而是一个金额适中的交易，这使得交易 7 到 交易 9 不属于欺诈行为。

欺诈检测器需要在多个交易事件之间记住一些信息。仅当一个大额的交易紧随一个小额交易的情况发生时，这个大额交易才被认为是欺诈交易。 在多个事件之间存储信息就需要使用到 状态 (//ci.apache.org/projects/flink/flink-docs-release-1.10/zh/concepts/glossary.html#managed-state)，这也是我们选择使用 KeyedProcessFunction (//ci.apache.org/projects/flink/flink-docs-release-1.10/zh/dev/stream/operators/process_function.html) 的原因。 它能够同时提供对状态和时间的细粒度操作，这使得我们能够在接下来的代码练习中实现更复杂的算法。

最直接的实现方式是使用一个 boolean 型的标记状态来表示是否刚处理过一个小额交易。 当处理到该账户的一个大额交易时，你只需要检查这个标记状态来确认上一个交易是是否小额交易即可。

然而，仅使用一个标记作为 `FraudDetector` 的类成员来记录账户的上一个交易状态是不准确的。 Flink 会在同一个 `FraudDetector` 的并发实例中处理多个账户的交易数据，假设，当账户 A 和账户 B 的数据被分发的同一个并发实例上处理时，账户 A 的小额交易行为可能会将标记状态设置为真，随后账户 B 的大额交易可能会被误判为欺诈交易。 当然，我们可以使用如 `Map` 这样的数据结构来保存每一个账户的状态，但是常规的类成员变量是无法做到容错处理的，当任务失败重启后，之前的状态信息将会丢失。 这样的话，如果程序曾出现过失败重启的情况，将会漏掉一些欺诈报警。

为了应对这个问题，Flink 提供了一套支持容错状态的原语，这些原语几乎与常规成员变量一样易于使用。

Flink 中最基础的状态类型是 ValueState (//ci.apache.org/projects/flink/flink-docs-release-1.10/zh/dev/stream/state/state.html#using-managed-keyed-state)，这是一种能够为被其封装的变量添加容错能力的类型。 `ValueState` 是一种 *keyed state*，也就是说它只能被用于 *keyed context* 提供的 operator 中，即所有能够紧随 `DataStream#keyBy` 之后被调用的operator。 一个 operator 中的 *keyed state* 的作用域默认是属于它所属的 key 的。 这个例子中，key 就是当前正在处理的交易行为所属的信用卡账户（key 传入 keyBy() 函数调用），而 `FraudDetector` 维护了每个帐户的标记状态。 `ValueState` 需要使用 `ValueStateDescriptor` 来创建，`ValueStateDescriptor` 包含了 Flink 如何管理变量的一些元数据信息。状态在使用之前需要先被注册。 状态需要使用 `open()` 函数来注册状态。

Java **Scala**

```
public class FraudDetector extends KeyedProcessFunction<Long, Transaction, Alert> {

    private static final long serialVersionUID = 1L;

    private transient ValueState<Boolean> flagState;

    @Override
    public void open(Configuration parameters) {
        ValueStateDescriptor<Boolean> flagDescriptor = new ValueStateDescriptor<>(
                "flag",
                Types.BOOLEAN);
        flagState = getRuntimeContext().getState(flagDescriptor);
    }
```

`ValueState` 是一个包装类，类似于 Java 标准库里边的 `AtomicReference` 和 `AtomicLong`。 它提供了三个用于交互的方法。`update` 用于更新状态，`value` 用于获取状态值，还有 `clear` 用于清空状态。 如果一个 key 还没有状态，例如当程序刚启动或者调用过 `ValueState#clear` 方法时，`ValueState#value` 将会返回 `null`。 如果需要更新状态，需要调用 `ValueState#update` 方法，直接更改 `ValueState#value` 的返回值可能不会被系统识别。 容错处理将在 Flink 后台自动管理，你可以像与常规变量那样与状态变量进行交互。

下边的示例，说明了如何使用标记状态来追踪可能的欺诈交易行为。

Java **Scala**

```
@Override
public void processElement(
        Transaction transaction,
        Context context,
        Collector<Alert> collector) throws Exception {

    // Get the current state for the current key
    Boolean lastTransactionWasSmall = flagState.value();

    // Check if the flag is set
    if (lastTransactionWasSmall != null) {
        if (transaction.getAmount() > LARGE_AMOUNT) {
            // Output an alert downstream
            Alert alert = new Alert();
            alert.setId(transaction.getAccountId());

            collector.collect(alert);
        }

        // Clean up our state
        flagState.clear();
    }

    if (transaction.getAmount() < SMALL_AMOUNT) {
        // Set the flag to true
        flagState.update(true);
    }
}
```

对于每笔交易，欺诈检测器都会检查该帐户的标记状态。 请记住，`ValueState` 的作用域始终限于当前的 key，即信用卡帐户。 如果标记状态不为空，则该帐户的上一笔交易是小额的，因此，如果当前这笔交易的金额很大，那么检测程序将输出报警信息。

在检查之后，不论是什么状态，都需要被清空。 不管是当前交易触发了欺诈报警而造成模式的结束，还是当前交易没有触发报警而造成模式的中断，都需要重新开始新的模式检测。

最后，检查当前交易的金额是否属于小额交易。 如果是，那么需要设置标记状态，以便可以在下一个事件中对其进行检查。 注意，`ValueState<Boolean>` 实际上有 3 种状态：unset (`null`)，`true`，和 `false`，`ValueState` 是允许空值的。 我们的程序只使用了 unset (`null`) 和 `true` 两种来判断标记状态被设置了与否。

欺诈检测器 v2：状态 + 时间 = ❤️

骗子们在小额交易后不会等很久就进行大额消费，这样可以降低小额测试交易被发现的几率。 比如，假设你为欺诈检测器设置了一分钟的超时，对于上边的例子，交易 3 和 交易 4 只有间隔在一分钟之内才被认为是欺诈交易。 Flink 中的 `KeyedProcessFunction` 允许您设置计时器，该计时器在将来的某个时间点执行回调函数。

让我们看看如何修改程序以符合我们的新要求：

- 当标记状态被设置为 `true` 时，设置一个在当前时间一分钟后触发的定时器。
- 当定时器被触发时，重置标记状态。
- 当标记状态被重置时，删除定时器。

要删除一个定时器，你需要记录这个定时器的触发时间，这同样需要状态来实现，所以你需要在标记状态后也创建一个记录定时器时间的状态。

Java **Scala**

```
    private transient ValueState<Boolean> flagState;
    private transient ValueState<Long> timerState;

    @Override
    public void open(Configuration parameters) {
        ValueStateDescriptor<Boolean> flagDescriptor = new ValueStateDescriptor<>(
                "flag",
                Types.BOOLEAN);
        flagState = getRuntimeContext().getState(flagDescriptor);

        ValueStateDescriptor<Long> timerDescriptor = new ValueStateDescriptor<>(
                "timer-state",
                Types.LONG);
        timerState = getRuntimeContext().getState(timerDescriptor);
    }
```

`KeyedProcessFunction#processElement` 需要使用提供了定时器服务的 `Context` 来调用。 定时器服务可以用于查询当前时间、注册定时器和删除定时器。 使用它，你可以在标记状态被设置时，也设置一个当前时间一分钟后触发的定时器，同时，将触发时间保存到 `timerState` 状态中。

Java **Scala**

```
if (transaction.getAmount() < SMALL_AMOUNT) {
    // set the flag to true
    flagState.update(true);

    // set the timer and timer state
    long timer = context.timerService().currentProcessingTime() + ONE_MINUTE;
    context.timerService().registerProcessingTimeTimer(timer);
    timerState.update(timer);
}
```

处理时间是本地时钟时间，这是由运行任务的服务器的系统时间来决定的。

当定时器触发时，将会调用 `KeyedProcessFunction#onTimer` 方法。 通过重写这个方法来实现一个你自己的重置状态的回调逻辑。

Java **Scala**

```
@Override
public void onTimer(long timestamp, OnTimerContext ctx, Collector<Alert> out) {
    // remove flag after 1 minute
    timerState.clear();
    flagState.clear();
}
```

最后，如果要取消定时器，你需要删除已经注册的定时器，并同时清空保存定时器的状态。 你可以把这些逻辑封装到一个助手函数中，而不是直接调用 `flagState.clear()`。

Java **Scala**

```
private void cleanUp(Context ctx) throws Exception {
    // delete timer
    Long timer = timerState.value();
    ctx.timerService().deleteProcessingTimeTimer(timer);

    // clean up all state
    timerState.clear();
    flagState.clear();
}
```

这就是一个功能完备的，有状态的分布式流处理程序了。

完整的程序

Java **Scala**

```
package spendreport;

import org.apache.flink.api.common.state.ValueState;
import org.apache.flink.api.common.state.ValueStateDescriptor;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.api.functions.KeyedProcessFunction;
import org.apache.flink.util.Collector;
import org.apache.flink.walkthrough.common.entity.Alert;
import org.apache.flink.walkthrough.common.entity.Transaction;

public class FraudDetector extends KeyedProcessFunction<Long, Transaction, Alert> {

    private static final long serialVersionUID = 1L;

    private static final double SMALL_AMOUNT = 1.00;
    private static final double LARGE_AMOUNT = 500.00;
    private static final long ONE_MINUTE = 60 * 1000;

    private transient ValueState<Boolean> flagState;
    private transient ValueState<Long> timerState;

    @Override
    public void open(Configuration parameters) {
        ValueStateDescriptor<Boolean> flagDescriptor = new ValueStateDescriptor<>(
                "flag",
                Types.BOOLEAN);
        flagState = getRuntimeContext().getState(flagDescriptor);

        ValueStateDescriptor<Long> timerDescriptor = new ValueStateDescriptor<>(
                "timer-state",
                Types.LONG);
        timerState = getRuntimeContext().getState(timerDescriptor);
    }

    @Override
    public void processElement(
            Transaction transaction,
            Context context,
            Collector<Alert> collector) throws Exception {

        // Get the current state for the current key
        Boolean lastTransactionWasSmall = flagState.value();

        // Check if the flag is set
        if (lastTransactionWasSmall != null) {
            if (transaction.getAmount() > LARGE_AMOUNT) {
                //Output an alert downstream
                Alert alert = new Alert();
                alert.setId(transaction.getAccountId());

                collector.collect(alert);
```

```
            }
            // Clean up our state
            cleanUp(context);
        }

        if (transaction.getAmount() < SMALL_AMOUNT) {
            // set the flag to true
            flagState.update(true);

            long timer = context.timerService().currentProcessingTime() + ONE_MINUTE;
            context.timerService().registerProcessingTimeTimer(timer);

            timerState.update(timer);
        }
    }

    @Override
    public void onTimer(long timestamp, OnTimerContext ctx, Collector<Alert> out) {
        // remove flag after 1 minute
        timerState.clear();
        flagState.clear();
    }

    private void cleanUp(Context ctx) throws Exception {
        // delete timer
        Long timer = timerState.value();
        ctx.timerService().deleteProcessingTimeTimer(timer);

        // clean up all state
        timerState.clear();
        flagState.clear();
    }
}
```

期望的结果

使用已准备好的 `TransactionSource` 数据源运行这个代码，将会检测到账户 3 的欺诈行为，并输出报警信息。 你将能够在你的 task manager 的日志中看到下边输出：

```
2019-08-19 14:22:06,220 INFO  org.apache.flink.walkthrough.common.sink.AlertSink            - Ale
rt{id=3}
2019-08-19 14:22:11,383 INFO  org.apache.flink.walkthrough.common.sink.AlertSink            - Ale
rt{id=3}
2019-08-19 14:22:16,551 INFO  org.apache.flink.walkthrough.common.sink.AlertSink            - Ale
rt{id=3}
2019-08-19 14:22:21,723 INFO  org.apache.flink.walkthrough.common.sink.AlertSink            - Ale
rt{id=3}
2019-08-19 14:22:26,896 INFO  org.apache.flink.walkthrough.common.sink.AlertSink            - Ale
rt{id=3}
```

想参与贡献翻译？ (https://cwiki.apache.org/confluence/display/FLINK/Flink+Translation+Specifications)