Flink China

Flink状态管理和容错机制介绍

施晓罡（花名：星罡）
阿里巴巴技术专家
2018.08.11

Flink China

施 晓 罡（花名：星罡）

xiaogang.sxg@alibaba-inc.com

2017-至今　Apache Flink项目　Committer

2016-至今　阿里巴巴计算平台事业部　技术专家

2010-2016　北京大学　博士

2006-2010　北京大学　学士

Flink China

主要内容

- 有状态的流数据处理
- Flink中的状态接口
- 状态管理和容错机制实现
- 阿里相关工作介绍

有状态的流数据处理

Flink China

有状态的计算

(input, state) -> (output, state')

传统流计算系统缺少对于程序状态的有效支持

- ☒ 状态数据的存储和访问
- ☒ 状态数据的备份和恢复
- ☒ 状态数据的划分和动态扩展

Flink China

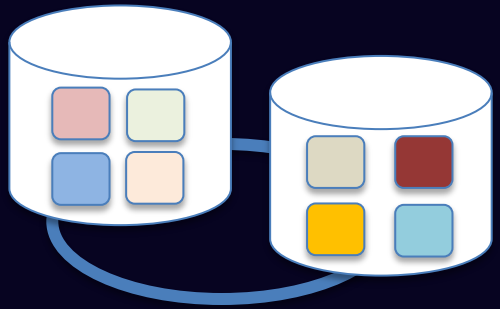
Flink提供了丰富的状态访问接口和高效的容错机制

多种数据类型	多种划分方式	多种存储格式	高效备份和恢复
Value, List, Map, Reducing, Folding, Aggregating	Keyed State Operator State	MemoryStateBackend FsStateBackend RocksDBStateBackend	提供ExactlyOnce保证 增量和异步备份 本地恢复

Flink中的状态管理

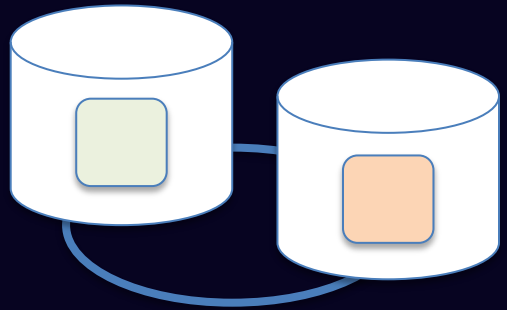
Flink China

Keyed States



记录每个Key对应的状态值
一个Task上可能包含多个Key
不同Task上不会出现相同的Key

Operator States



记录每个Task对应的状态值
数据类型只支持List类型

Flink China

Keyed State的使用

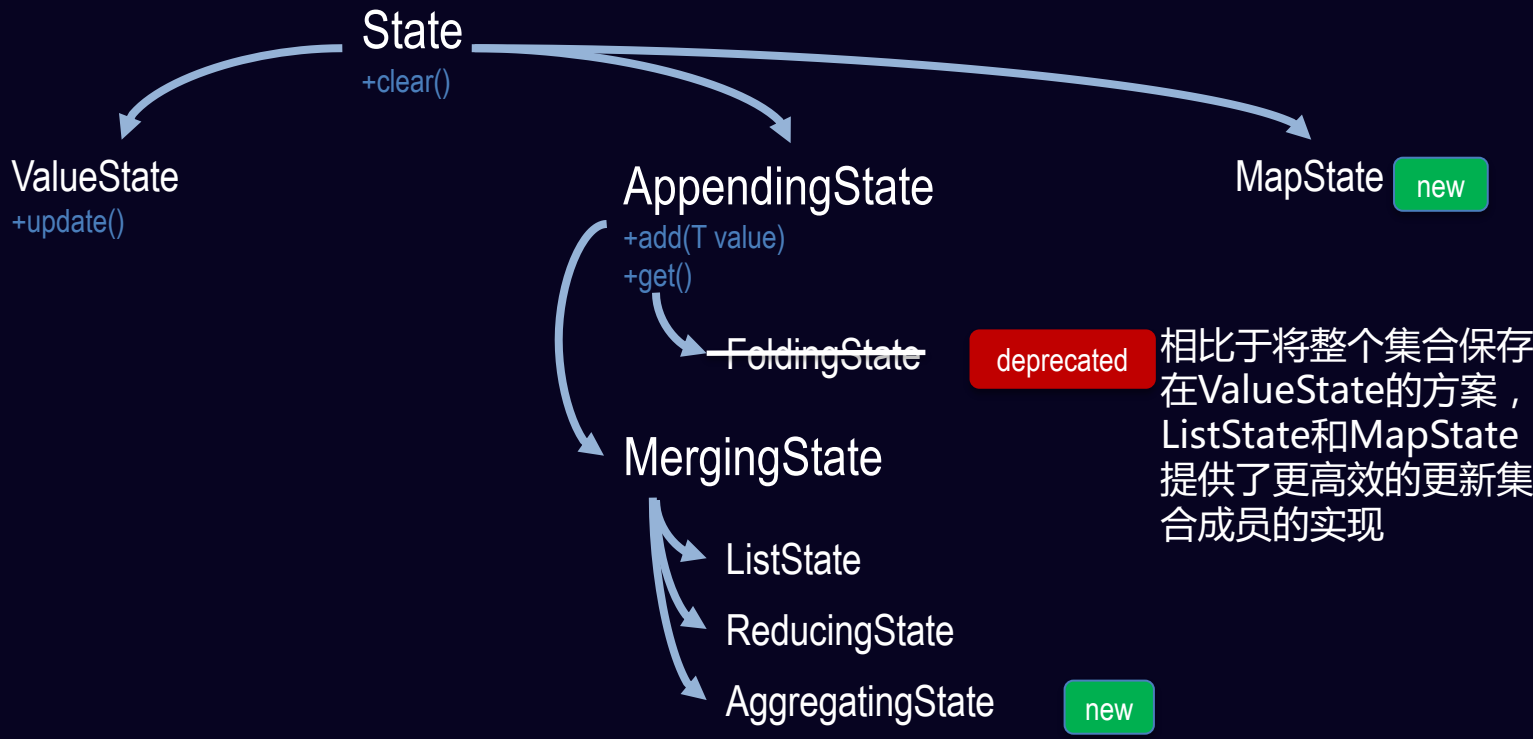
```
public class WordCounter extends RichFlatMapFunction<String, Tuple2<String, Integer>> {
   private transient ValueState<Integer> state;

   @override
   public void open(Configuration configuration) {
      ValueStateDescriptor<Integer> descriptor = new ValueStateDescriptor<>(“counter”, Integer.class, 0);
      state = getRuntimeContext().getState(descriptor);
   }

   @override
   public void flatMap(String word, Collector<Tuple2<String, Integer>> collector) throws Exception {
      int count = state.value();
      count++;
      state.update(count);

      collector.collect(new Tuple2<>(word, count));
   }
}
```

通过RuntimeContext访问State

访问和修改当前Key对应的State值

Flink China

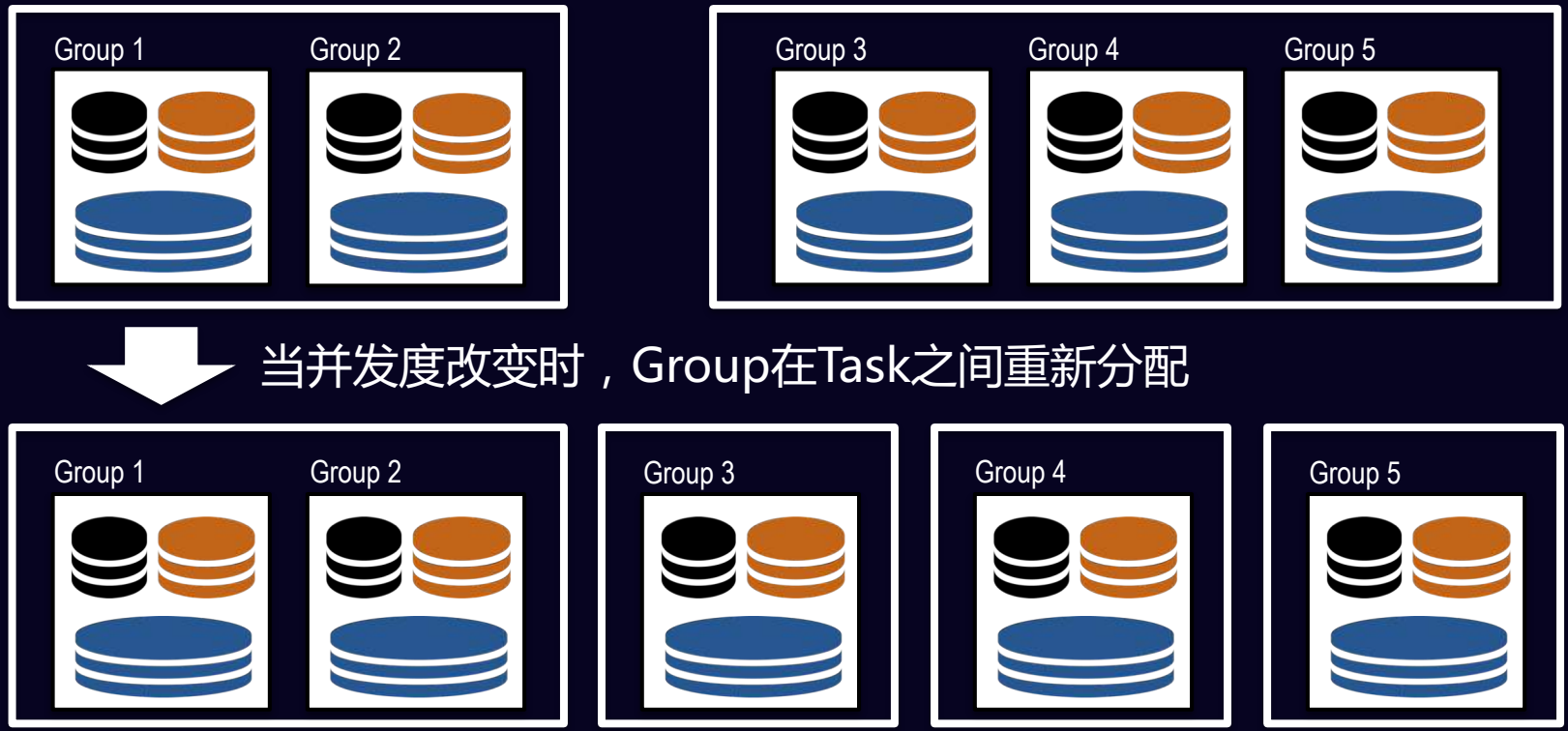
Keyed State提供了多种数据结构



相比于将整个集合保存在ValueState的方案，ListState和MapState提供了更高效的更新集合成员的实现

Flink China

Keyed State的动态扩展



Flink China

Operator State的使用

Operator State可以在任意Stream上使用，但只支持有限的数据结构

```
public class MyFunction<T> extends MapFunction<T, T>, CheckpointedFunction {
   private transient ListState<Long> partitionOffsetsState;
   private transient List<Long> partitionOffsets;

   @override
   public void initializeState(FunctionInitializationContext context) throws Exception {
      ListStateDescriptor<List> descriptor = new ListStateDescriptor<>(“offset”, Long.class);
      partitionOffsetsState = context.getOperatorStateStore().getOperatorState(descriptor);
   }

   @override
   public void snapshotState(FunctionSnapshotContext context) throws Exception {
      partitionOffsetsState.clear();
      partitionOffsetsState.addAll(partitionOffsets);
   }
}
```

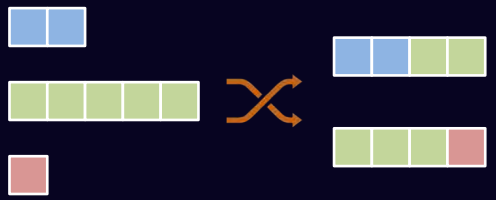
通过FunctionInitializationContext访问State

在Snapshot时将状态数据保存到State中

Flink China

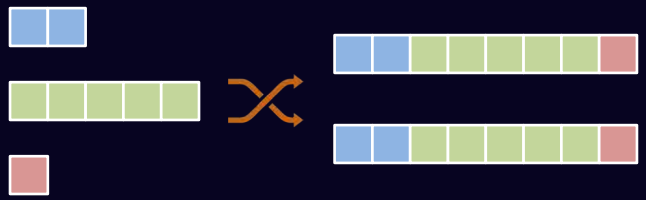
Operator State提供了多种扩展方式

ListState



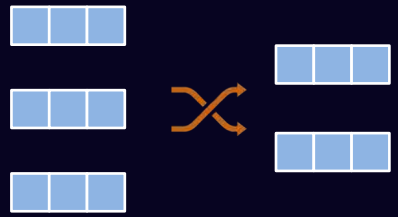
所有State中的元素均匀划分给新的Task

UnionListState



所有State中的元素全部分配给新的Task

BroadcastState(map)



所有Task上的State都是一样的，改变并发的时候，新的Task获得State的一个备份

Flink China

使用Checkpoint提高程序可靠性

Flink按照一定的时间间隔对程序运行状态进行备份

当发生故障时，Flink会将所有Task的状态恢复到最后一次checkpoint中的状态，并从那里开始重新开始执行

Flink提供了多种正确性保障

AT LEAST ONCE： 所有数据至少被处理一次。在发生故障时，数据可能会被处理多次

EXACTLY ONCE： 所有数据正好被处理，无论是否发生异常

Flink China

备份未保存在State中的程序状态数据

```
public class MyFunction<T> extends MapFunction<T, T>, ListCheckpointed<Long> {
   private transient List<Long> partitionOffsets;

   @override
   public void restoreState(List<Long> restoredPartitionOffsets) throws Exception {
      partitionOffsets.addAll(restoredPartitionOffsets);
   }

   @override
   public List<Long> snapshotState(long checkpointId, long checkpointTimestamp) throws Exception {
      return new ArrayList<>(partitionOffsets);
   }
}
```

使用系统的恢复数据来恢复程序运行状态

在Snapshot时将状态数据返回给Flink。这些数据将被保存在系统默认的Operator State中。

对于高级用户，还可以使用StateSnapshotContext中提供的KeyedStateCheckpointOutputStream和OperatorStateCheckpointOutputStream来分别为按key划分的状态数据和按元素数据划分的状态数据进行备份

Flink China

从已停止作业的运行状态中恢复

Savepoint

用户通过命令行触发

数据以标准格式存储，允许作业版本升级或者配置变更

用户在启动命令中提供用于恢复作业状态的savepoint路径

External Checkpoint

在checkpoint完成时将在用户给定的外部存储中保存一份meta数据

当作业以FAILED或者CANCELLED(可配置)状态结束时，外部存储的meta数据将被保留下来

用户在启动命令中提供用于恢复作业状态的checkpoint路径

状态管理和容错的实现

Flink China

State和Checkpoint数据的存储方式

Checkpoint数据直接传递给Master节点

Checkpoint数据写入文件中，而将文件路径传递给Master

Checkpoint数据写入文件中，而将文件路径传递给Master

MemoryStateBackend

FsStateBackend

RocksDBStateBackend

HeapStateBackend

State数据存储在内存中

State数据存储在RocksDB中

Flink China

HeapKeyedStateBackend

State数据被保存在一个由多层Java Map嵌套而成的数据结构中
默认情况下，数据大小不可以超过5MB

State	KeyGroup	Namespace	Key:Value
window-contents	1	Window(10, 20)	K1:V11
			K2:V12
		Window(15, 25)	K1:V21
			K3:{V23, V24, …}
			…
	…	…	…

Flink China

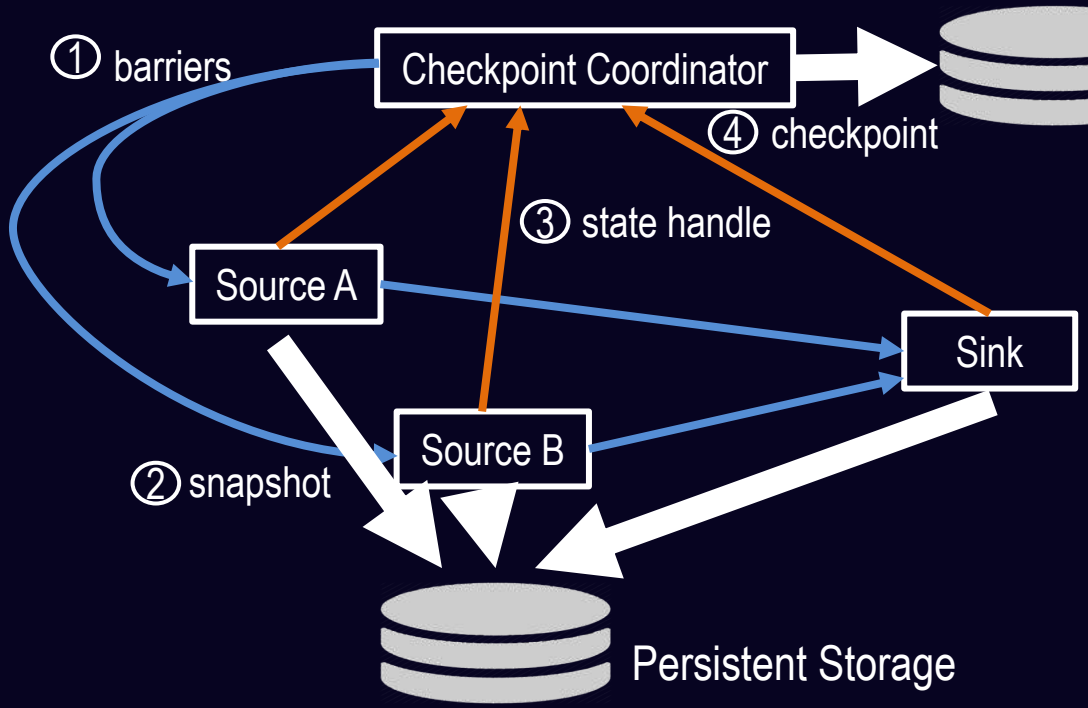
RocksDBKeyedStateBackend

每个State存储在单独一个ColumnFamily中

State	KeyGroup+Key+Namespace	Value
Window-contents	(1, K1, Window(10, 20))	V11
	(1, K1, Window(15, 25))	V21
	(1, K2, Window(10, 20))	V12
	(1, K3, Window(15, 25))	{V23, V24, ...}
...	...	...

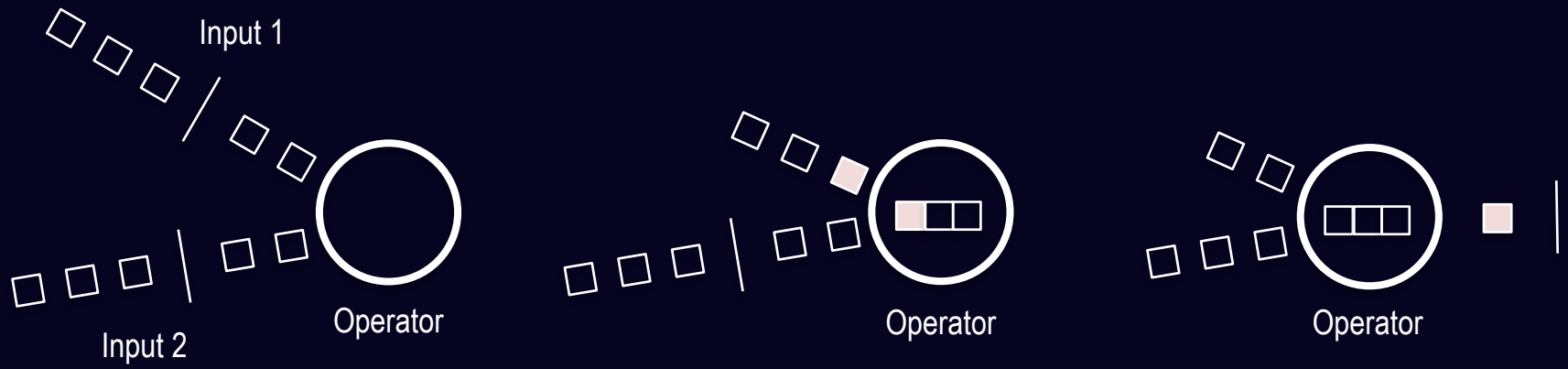
Flink China

Checkpoint执行流程(Chandy-Lamport)



1. Coordinator向所有source节点发送barrier
2. 当Task从输入中收到所有barrier时，将自己的状态写入持久化存储中，并向自己的输出继续传递barrier
3. 当Task完成备份之后，并将备份数据的地址通知Coordinator
4. 当Coordinator收到所有Task完成的消息之后，将这些备份数据的地址写入可持久化存储中，并完成checkpoint

Checkpoint Barrier的对齐



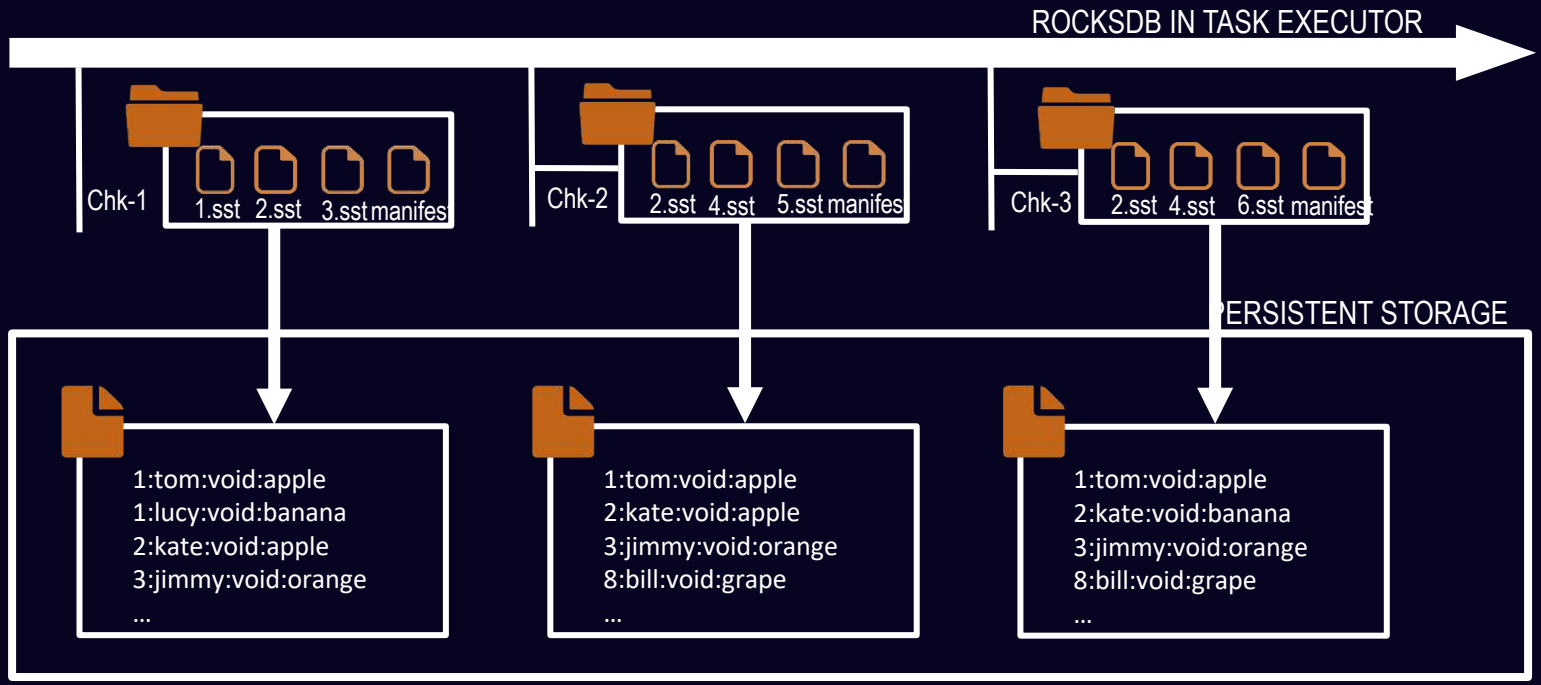
当继续从一个已经收到barrier的输入接收到数据时，如果我们直接处理这个数据，那么这个数据虽然在barrier之后，但其将包含在这次checkpoint备份的状态之中

当发生故障时并从这个checkpoint恢复时，虽然这个数据已经包含在了恢复的状态中，但仍然会被处理。此时的正确性语义是AT LEAST ONCE

为了实现EXACTLY ONCE的语义，Flink通过一个Input Buffer将在对齐阶段收到的数据缓存起来，等到对齐完成之后再进行处理。

全量Checkpoint

存在大量重复数据，并且需要遍历文件内容



Flink China

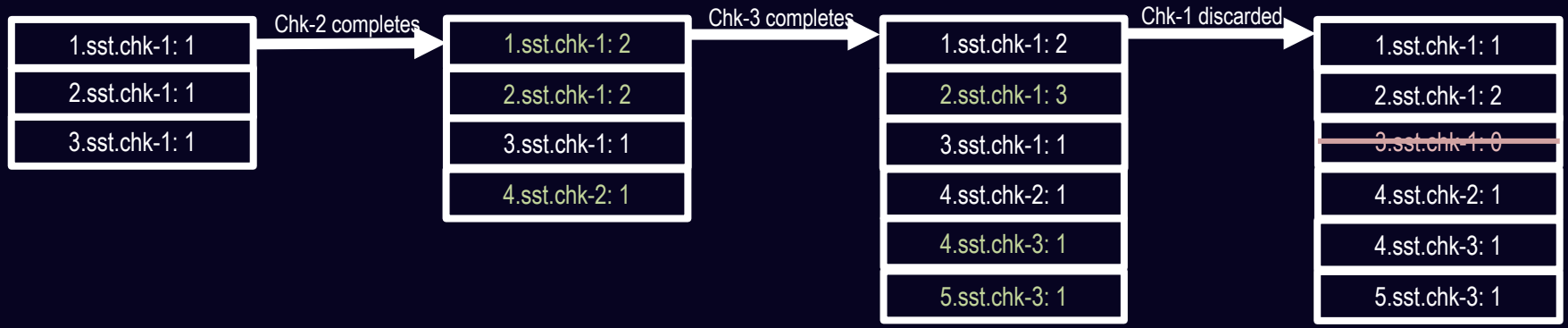
RocksDB的增量Checkpoint

1. 通过硬链创建RocksDB当前文件的备份（同步）
2. 恢复任务执行，在后台线程中和最后一次完成的checkpoint中的文件列表进行对比，将不在文件列表中的文件上传到持久化存储中
3. 所有文件都将checkpoint作为后缀以防止冲突
4. 包含了所有文件（包括新文件和旧文件）的state handle将传递给Coordinator

Checkpoint 2只需要持久化4.sst

Checkpoint 3无法引用4.sst.chk-2，因为当checkpoint 2失败之后，这个文件会被清理掉。

RocksDB的增量Checkpoint



1. Coordinator维护了每个checkpoint文件的引用计数
2. 当一个checkpoint完成之后，其所引用的文件都会被记录下来
3. Flink只维护了一定数据的checkpoint。当checkpoint数目超过一定阈值时，最旧的checkpoint将被删除
4. 当一个checkpoint删除之后，其所引用的文件都会减去对应的引用计数。当一个文件没有任何引用的时候，才会被真正删除。

阿里相关工作简介

Flink China

Flink在阿里的成长路线

- 2015.8 调研Flink
  - 架构最好的流式计算引擎
  - 尚未在大规模生产中验证
- 2015.10 启动Blink
  - 完善Flink架构和稳定性
  - 将Flink在阿里生产上线
- 2016.11 Blink双11上线
  - 搜索推荐全实时链路
  - 实时商品、报表、在线学习
- Blink统一阿里实时流计算
  - 统一服务阿里集团实时计算业务
  - 基于Blink的StreamCompute-2.0

Flink China

阿里在状态管理和容错相关的工作

提供了丰富的State类型（ MapState, SortedMapState)

实现了增量Checkpoint，提升大规模State场景下的性能

实现了Barrier优先读取，提高Checkpoint完成的效率

实现了本地Checkpoint数据缓存，提升大规模作业的恢复性能

基于State重构了InternalTimerService，允许大规模数据流上的Window操作

Flink China

Thanks !