



状态管理与容错机制

孙梦瑶 · 美团点评/研发工程师

Apache Flink Community China



Apache Flink

CONTENT

目录 >>

01 /

状态管理的基本概念

02 /

状态的类型与使用示例

03 /

容错机制与故障恢复

01

状态管理的基本概念

为什么要管理状态

状态管理的基本概念

1.1 / 什么是状态

1.2 / 为什么要管理状态

1.3 / 理想的状态管理

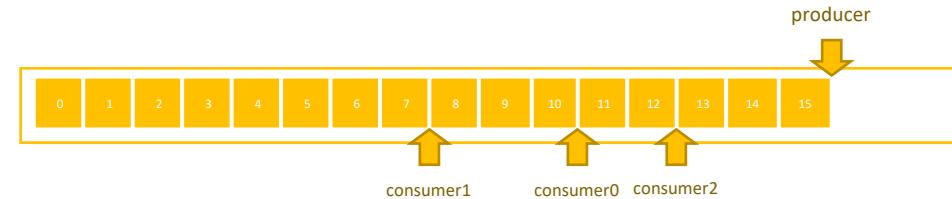


1.1 什么是状态

无状态计算的例子：消费延迟计算

- 消息队列
 - 一个生产者持续写入
 - 多个消费组分别读取
- 如何实时统计每个消费者落后多少条数据
- 输入输出：

```
{  
    "timestamp": 1555516800,  
    "offset":  
    {  
        "producer": 15,  
        "consumer0": 10,  
        "consumer1": 7,  
        "consumer2": 12  
    }  
}  
  
{  
    "timestamp": 1555516800,  
    "lag":  
    {  
        "consumer0": 5,  
        "consumer1": 8,  
        "consumer2": 3  
    }  
}
```



- 单条输入包含所需的所有信息
- 相同输入可以得到相同输出



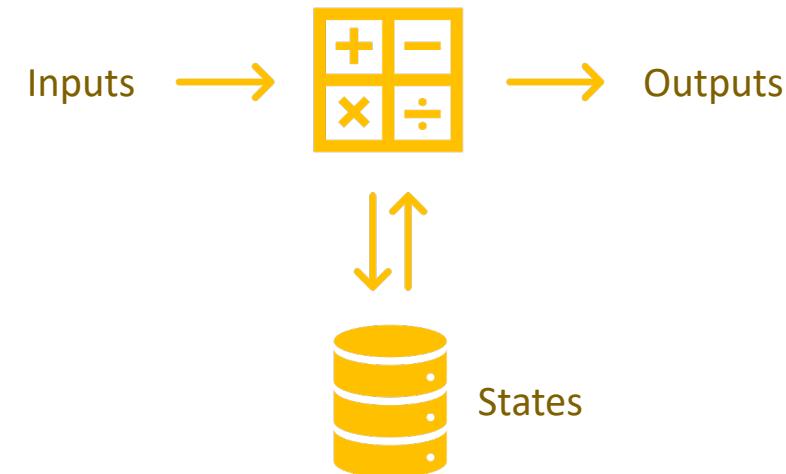
1.1 什么是状态

有状态计算的例子：访问量统计

- Nginx 访问日志
 - 每个请求访问一个URL地址
- 如何实时统计每个地址总共被访问了多少次
- 输入输出：

```
[{  
    "@timestamp": "18/Apr/2019:00:00:00",  
    "remote_addr": "127.0.0.1",  
    "request": "GET",  
    "url": "/api/a"  
, {  
    "@timestamp": "18/Apr/2019:00:00:01",  
    "remote_addr": "127.0.0.1",  
    "request": "POST",  
    "url": "/api/b"  
, {  
    "@timestamp": "18/Apr/2019:00:00:00",  
    "remote_addr": "127.0.0.1",  
    "request": "GET",  
    "url": "/api/a"  
}]  
  
[  
    {"url": "/api/a",  
     "count": 1  
, {  
        "url": "/api/b",  
        "count": 1  
, {  
            "url": "/api/a",  
            "count": 2  
        }]  
]
```

- 单条输入仅包含所需的部分信息
 - 当前请求信息
- 相同输入可能得到不同输出
 - 当前请求之前的累计访问量





1.1 什么是状态

- 需要使用状态的场景举例



去重

记录所有的主键



窗口计算

已进入的未触发的数据



机器学习/深度学习

训练的模型及参数



访问历史数据

需要与昨日进行对比



1.2 为什么要管理状态

- 最直接的方式——内存



存储容量限制



备份与恢复



横向扩展

- 对流式作业的要求



7*24小时运行，高可靠



数据不丢不重，恰好计算一次



数据实时产出，不延迟



1.3 理想的状态管理



易用

丰富的数据结构
多样的组织形式
简洁的扩展接口



高效

读写快, 恢复快
可以方便地横向扩展
备份不影响处理性能



可靠

持久化
不丢不重
具备容错能力

02

状态的类型与使用示例

状态如何为我所用

状态的类型与使用示例

2.1/ Managed State & Raw State

2.2/ Keyed State & Operator State

2.3/ Keyed State 使用示例



2.1 Managed State & Raw State

状态管理方式

Managed State

- Flink Runtime 管理
 - 自动存储, 自动恢复
 - 内存管理上有优化

状态数据结构

已知的数据结构

- value, list, map ...

推荐使用场景

大多数情况下均可使用

Raw State

- 用户自己管理
 - 需要自己序列化

字节数组

- byte[]

自定义 Operator 时可使用



2.2 Keyed State & Operator State

- **Keyed State**

- 只能用在 KeyedStream 上的算子中
- 每个 Key 对应一个 State
 - 一个 Operator 实例处理多个 Key, 访问相应的多个 State
- 并发改变, State 随着 Key 在实例间迁移
- 通过 RuntimeContext 访问
 - Rich Function
- 支持的数据结构
 - ValueState
 - ListState
 - ReducingState
 - AggregatingState
 - MapState

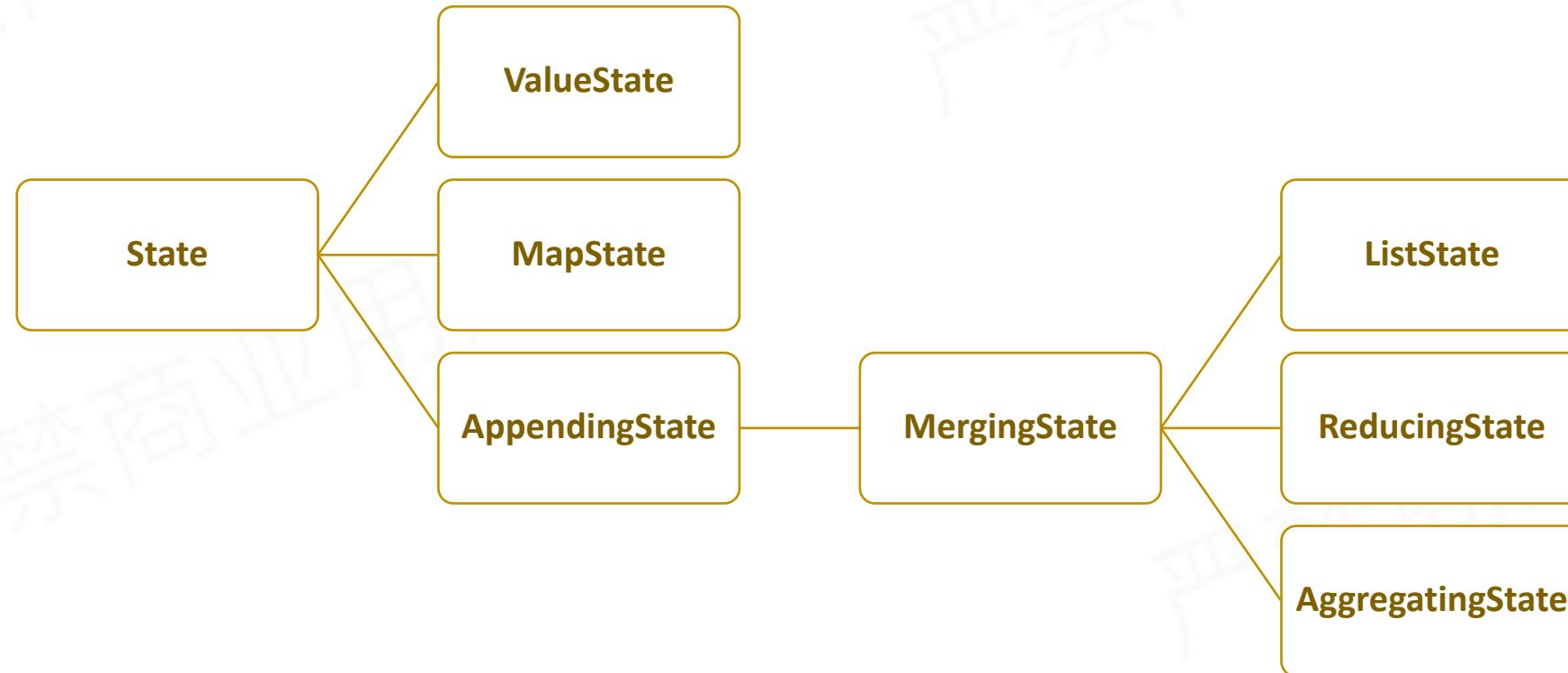
- **Operator State**

- 可以用于所有算子
 - 常用于 source, 例如 FlinkKafkaConsumer
- 一个 Operator 实例对应一个 State
- 并发改变时有多种重新分配方式可选
 - 均匀分配
 - 合并后每个得到全量
- 实现 CheckpointedFunction 或 ListCheckpointed 接口
- 支持的数据结构
 - ListState



2.3 Keyed State 使用示例

- 几种 Keyed State 之间的关系



2.3 Keyed State 使用示例

- 几种 Keyed State 之间的差异

	状态数据类型	访问接口
ValueState	单个值	<ul style="list-style-type: none">• update(T) / T value()
MapState	Map	<ul style="list-style-type: none">• put(UK key, UV value) / putAll(Map<UK,UV> map)• remove(UK key)
ListState	List	<ul style="list-style-type: none">• boolean contains(UK key) / UV get(UK key)• Iterable<Map.Entry> entries() / Iterator<Map.Entry> iterator()• Iterable<UK> keys() / Iterable<UV> values()
ReducingState	单个值	<ul style="list-style-type: none">• add(T) / addAll(List<T>)• update(List<T>) / Iterable<T> get()
AggregatingState	单个值	<ul style="list-style-type: none">• add(IN) / OUT get()

2.3 Keyed State 使用示例

- ValueState 使用示例
- 简单状态机

- <https://github.com/apache/flink/blob/master/flink-examples/flink-examples-streaming/src/main/java/org/apache/flink/streaming/examples/statemachine/StateMachineExample.java>

```
static class StateMachineMapper extends RichFlatMapFunction<Event, Alert> {
    private ValueState<State> currentState;
    public void open(Configuration conf) {
        currentState = getRuntimeContext().getState(new ValueStateDescriptor<>("state", State.class));
    }
    public void flatMap(Event evt, Collector<Alert> out) throws Exception {
        State state = currentState.value();
        if (state == null) {
            state = State.Initial;
        }
        State nextState = state.transition(evt.type());
        if (nextState == State.InvalidTransition) {
            out.collect(new Alert(evt.sourceAddress(), state, evt.type()));
        } else if (nextState.isTerminal()) {
            currentState.clear();
        } else {
            currentState.update(nextState);
        }
    }
}
```

03

容错机制与故障恢复

如何确保不丢不重

容错机制与故障恢复

3.1 / 状态如何保存及恢复

3.2 / 可选的状态存储方式

3.1 状态如何保存及恢复

- **Checkpoint**
- 定时制作分布式快照，对程序中的状态进行备份
- 发生故障时
 - 将整个作业的所有Task都回滚到最后一次成功Checkpoint中的状态，然后从那个点开始继续处理
- 必要条件
 - 数据源支持重发
- 一致性语义
 - 恰好一次
 - 至少一次

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.enableCheckpointing(1000);
env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);
env.getCheckpointConfig().setMinPauseBetweenCheckpoints(500);
env.getCheckpointConfig().setCheckpointTimeout(60000);
env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);
env.getCheckpointConfig().enableExternalizedCheckpoints(ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);
```



3.1 状态如何保存及恢复

Checkpoint

- 由 Flink 自动触发并管理

触发管理方式

- 在 Task 发生异常时快速恢复
 - 例如网络抖动导致的超时异常

主要用途

- 轻量
- 自动从故障中恢复
- 在作业停止后默认清除

特点

Savepoint

- 由用户手动触发并管理

- 有计划地进行备份，使作业能停止后再恢复
 - 例如修改代码、调整并发
- 持久
- 以标准格式存储，允许代码或配置发生改变
- 手动触发从Savepoint的恢复



3.2 可选的状态存储方式

- **MemoryStateBackend**
- 构造方法
 - MemoryStateBackend(int maxStateSize, boolean asynchronousSnapshots)
- 存储方式
 - State: TaskManager 内存
 - Checkpoint: JobManager 内存
- 容量限制
 - 单个 State maxStateSize 默认 5M
 - maxStateSize <= akka.framesize 默认 10M
 - 总大小不超过 JobManager 的内存
- 推荐使用的场景
 - 本地测试；几乎无状态的作业，比如ETL；JobManager 不容易挂，或挂掉影响不大的情况
 - **不推荐在生产场景使用**



3.2 可选的状态存储方式

- **FsStateBackend**
- 构造方法
 - FsStateBackend(URI checkpointDataUri, boolean asynchronousSnapshots)
- 存储方式
 - State: TaskManager 内存
 - Checkpoint: 外部文件系统 (本地或HDFS)
- 容量限制
 - 单 TaskManager 上 State 总量不超过它的内存
 - 总大小不超过配置的文件系统容量
- 推荐使用的场景
 - 常规使用状态的作业, 例如分钟级窗口聚合、join; 需要开启 HA 的作业
 - 可以在生产场景使用



3.2 可选的状态存储方式

- **RocksDBStateBackend**
- 构造方法
 - RocksDBStateBackend(URI checkpointDataUri, boolean enableIncrementalCheckpointing)
- 存储方式
 - State: TaskManager 上的 KV 数据库 (实际使用内存+磁盘)
 - Checkpoint: 外部文件系统 (本地或HDFS)
- 容量限制
 - 单 TaskManager 上 State 总量不超过它的内存+磁盘
 - 单 Key 最大 2G
 - 总大小不超过配置的文件系统容量
- 推荐使用的场景
 - 超大状态的作业, 例如天级窗口聚合; 需要开启 HA 的作业; 对状态读写性能要求不高的作业
 - 可以在生产场景使用



总结

为什么要使用状态?

数据之间有关联，需要通过状态满足业务逻辑

为什么要管理状态?

实时计算作业需要7*24运行，需要应对不可靠因素带来的影响

如何选择状态的类型和存储方式?

分析自己的业务场景，比对各方案的利弊，选择合适的，够用即可



Apache Flink

THANKS

