Apache Flink 概念介绍：有状态流式处理引擎的基石

戴资力 · Ververica / Software Engineer / Apache Flink PMC

Apache Flink 直播教程 - 2019年03月14日

CONTENT

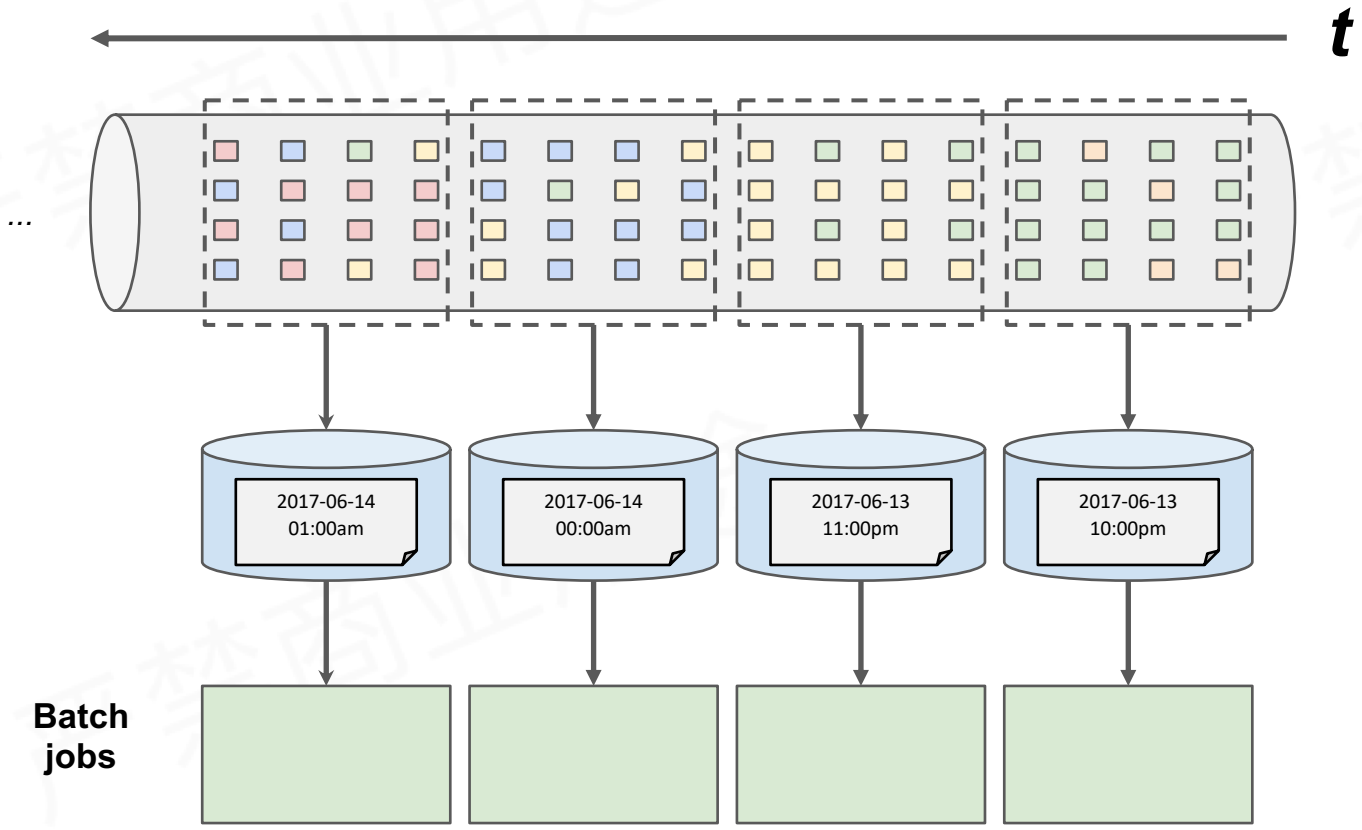
目录 >>

01

何谓「有状态流式处理」？

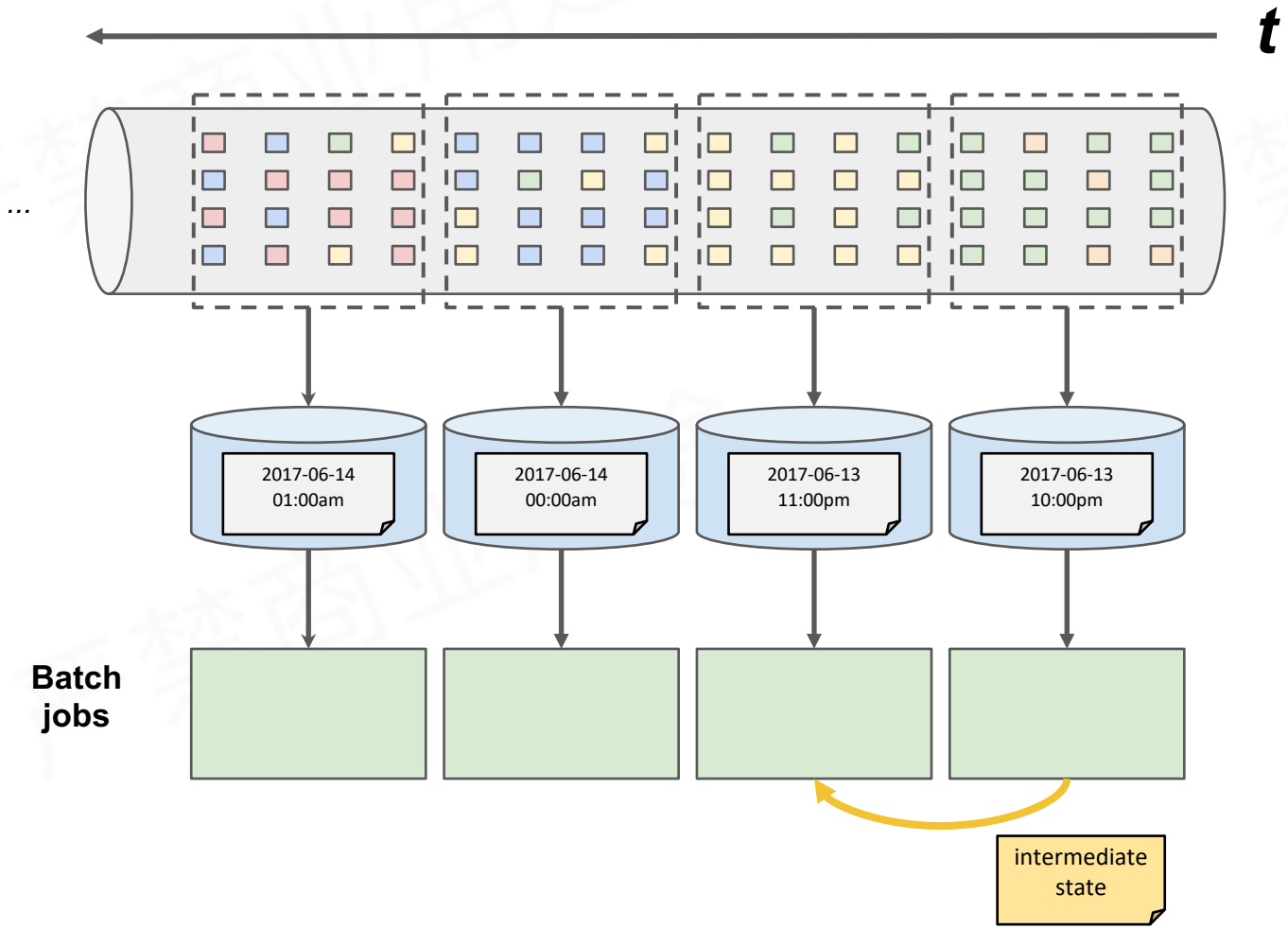
What is stateful stream processing?

传统批次处理方法



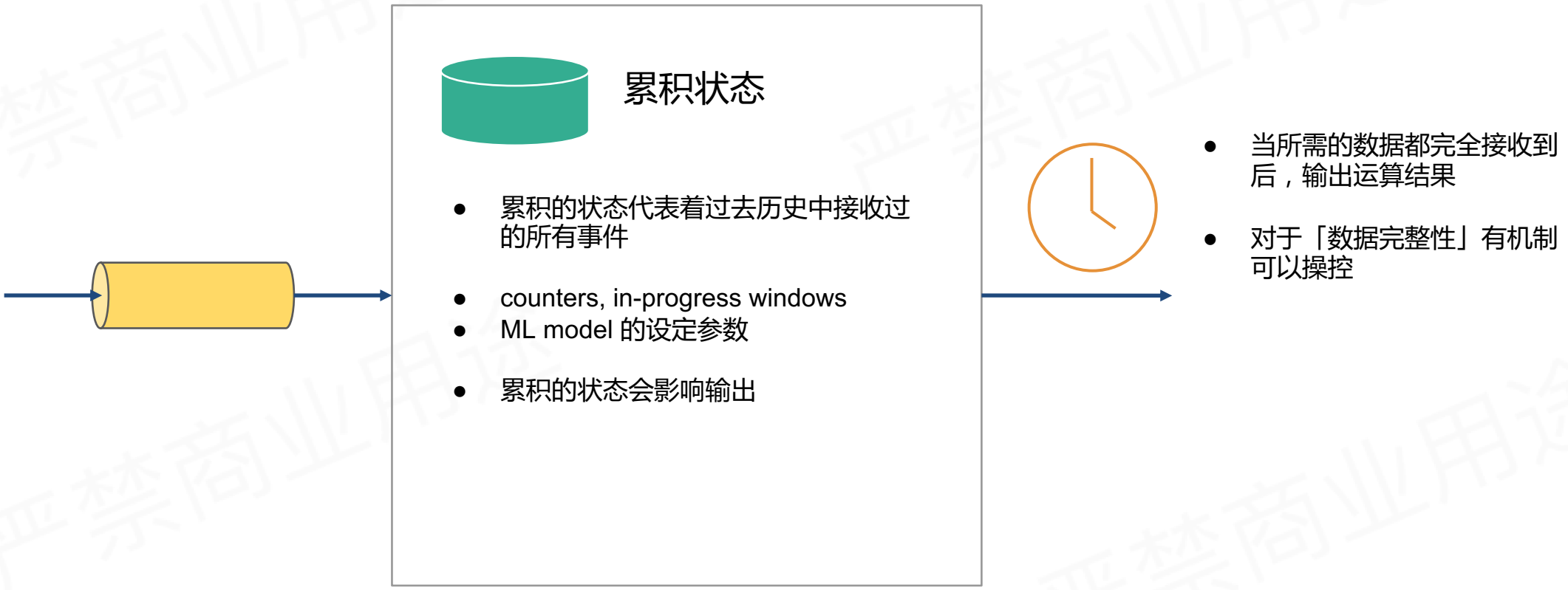
- 持续收取数据
- 以时间作为划分数个批次档案的依据
- 周期性执行批次运算

传统批次处理方法

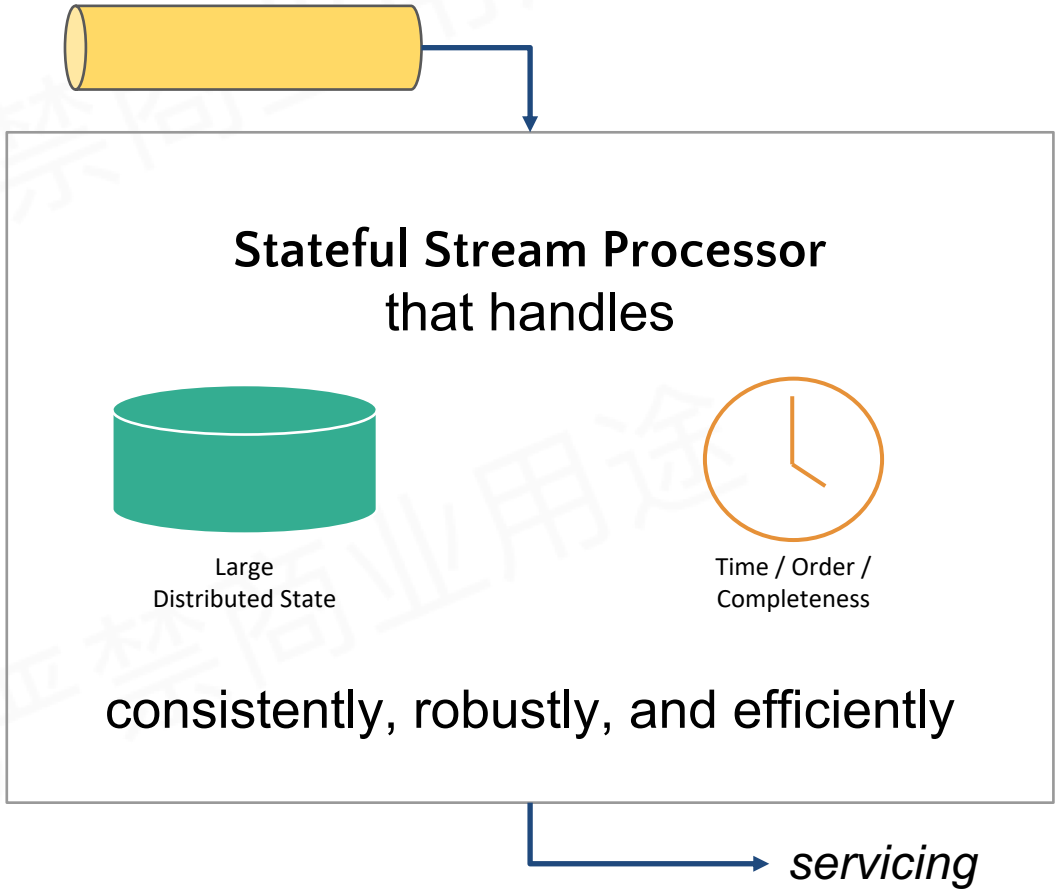


- 假设计算每小时出现特定事件转换的次数
  (# of A → B per hour)
- 如果事件转换跨越了所定义的时间划分?
  → 将中介运算结果(intermediate result) 带到下一个批次运算
- 如果接收到事件的顺序颠倒?

理想方法

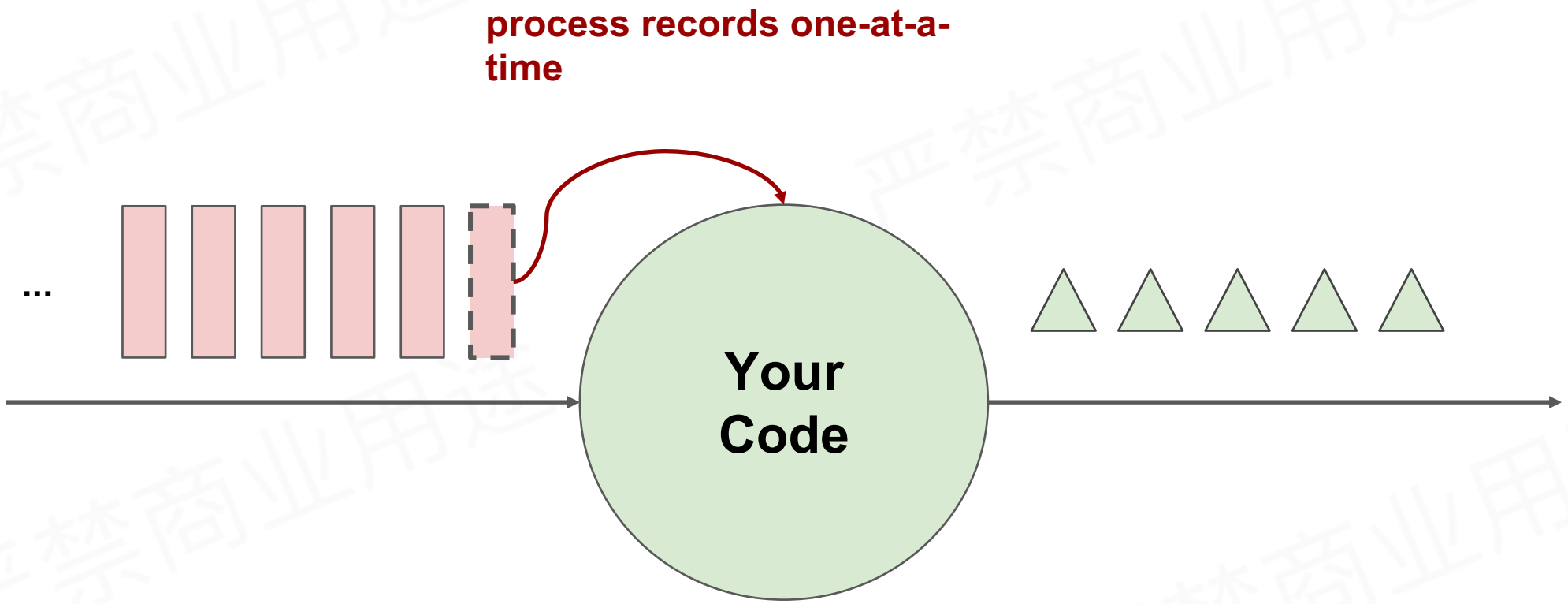


理想方法



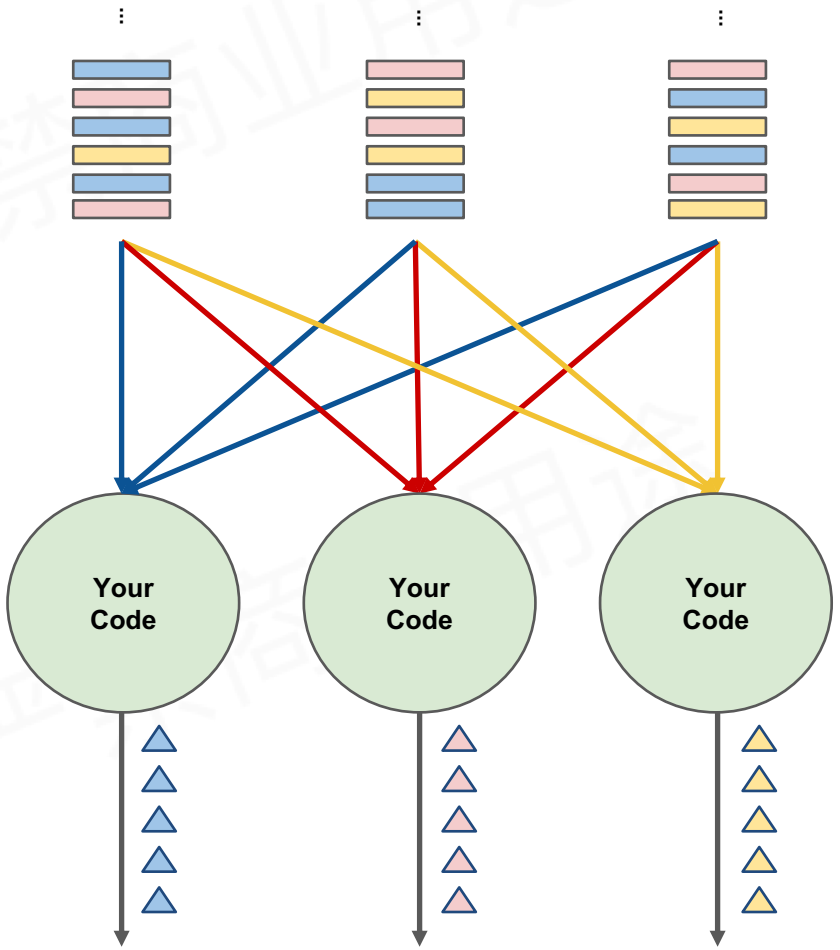
- Stateful stream processing as a **new paradigm** to **continuously process continuous data**
- Produce **accurate** results
- Results available in real-time is only a natural consequence of the model

流式处理



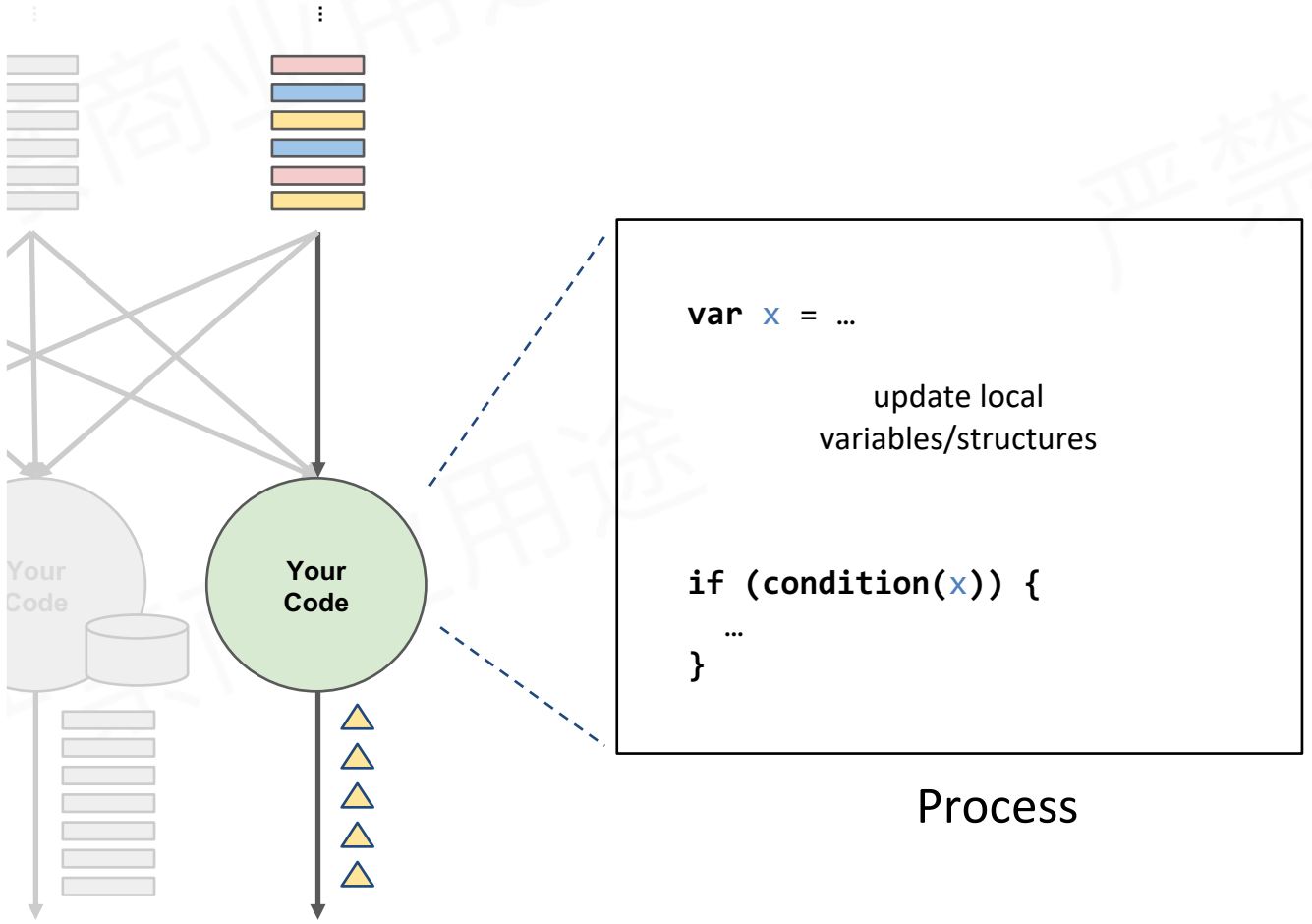
Long running computation, on an endless stream of input

分散式流式处理

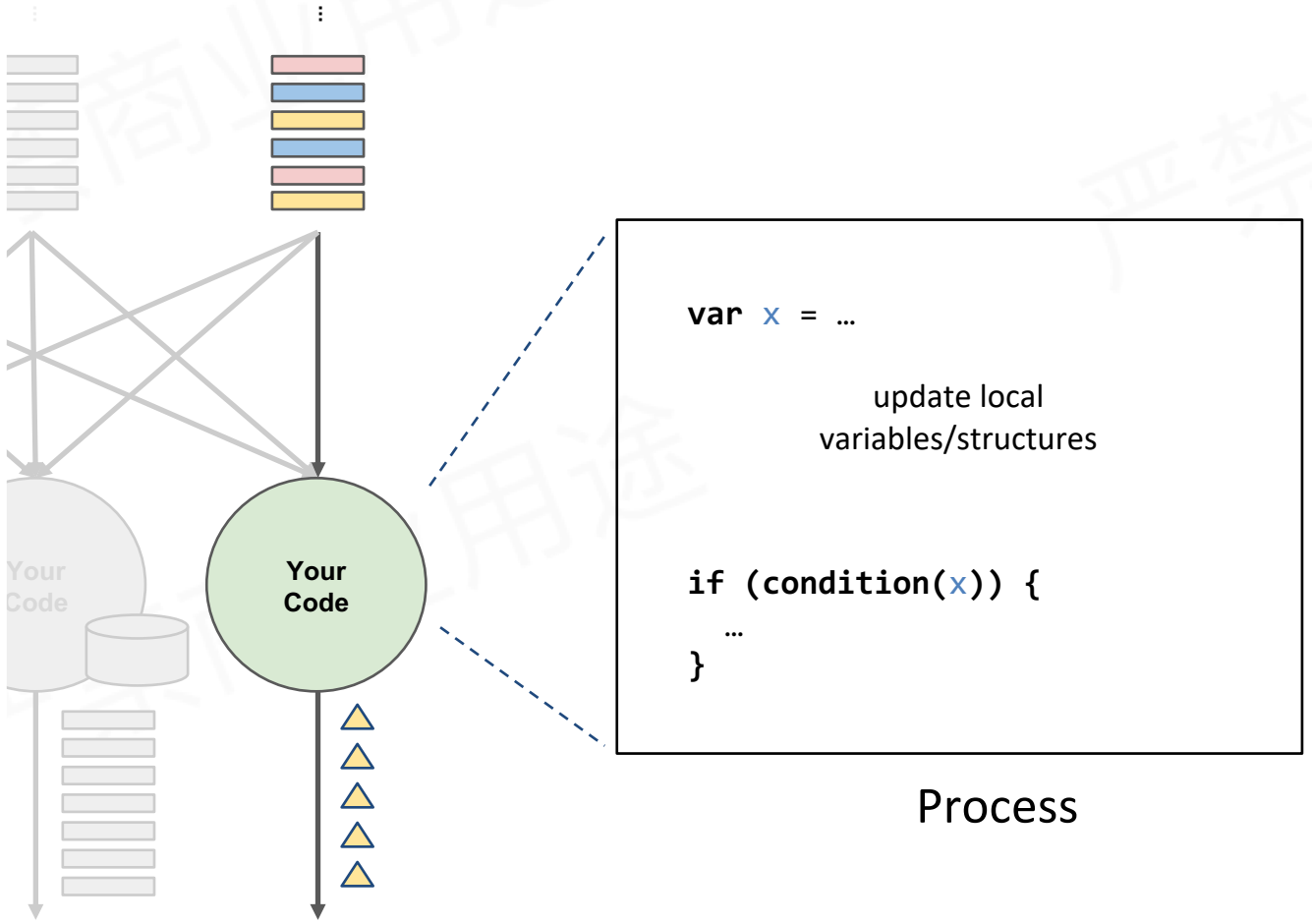


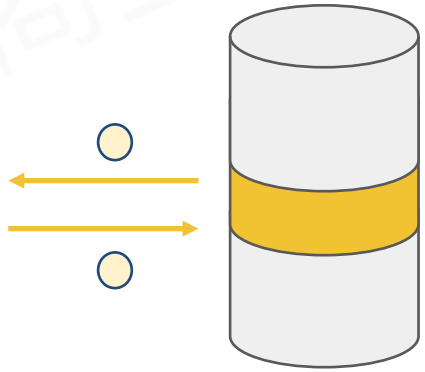
- partitions input streams by some key in the data
- distributes computation across multiple instances
- Each instance is responsible for some key range

有状态分散式流式处理



有状态分散式流式处理





- embedded local state backend
- State co-partitioned with the input stream by key

02

有状态流式处理的挑战

Challenges for a stateful stream processor

2.1 / 状态容错（State Fault Tolerance）

状态维护（State Management）

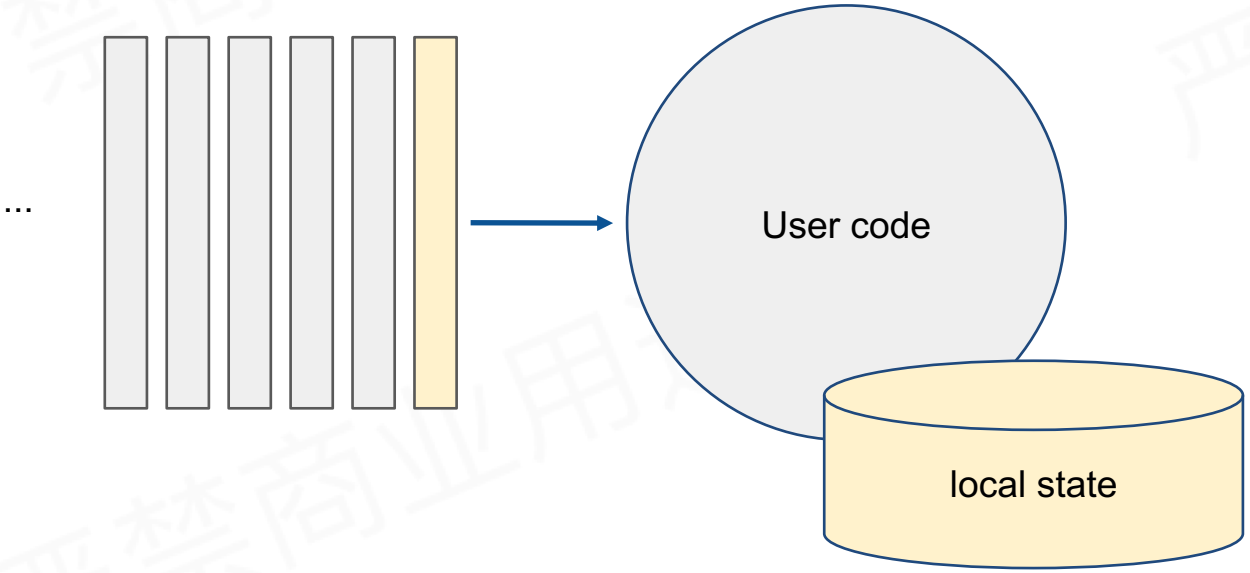
Event-time 处理（Event-time processing）

状态保存与迁移（Savepoints and Job Migration）

状态容错

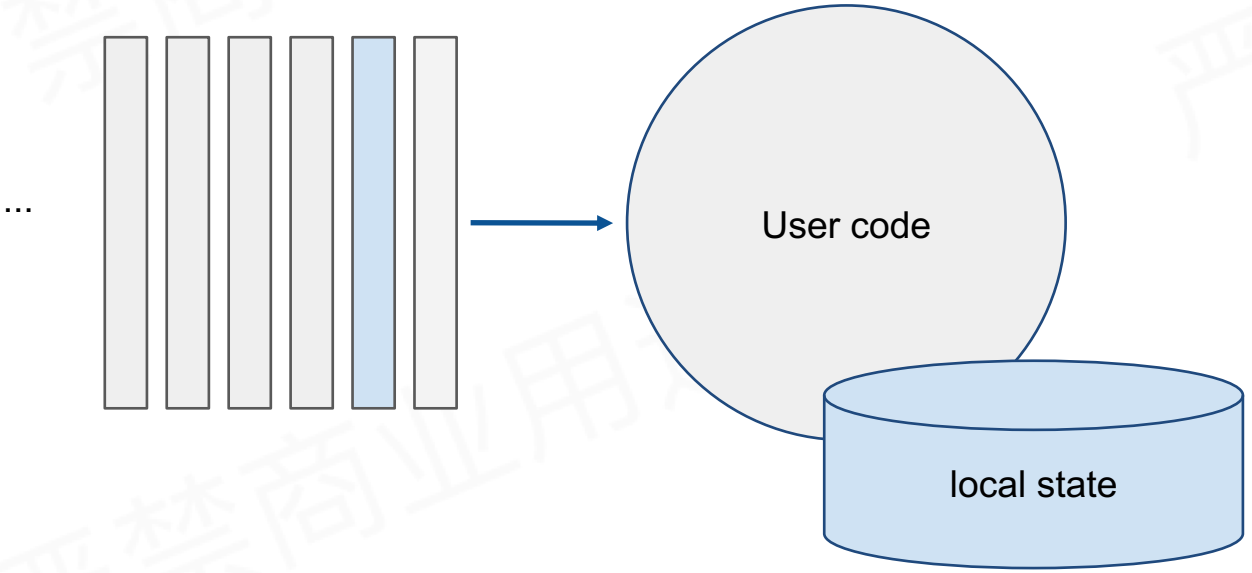
- 如何确保状态拥有**精确一次（exactly-once guarantee）**的容错保证？

简易场景的精确一次容错方法



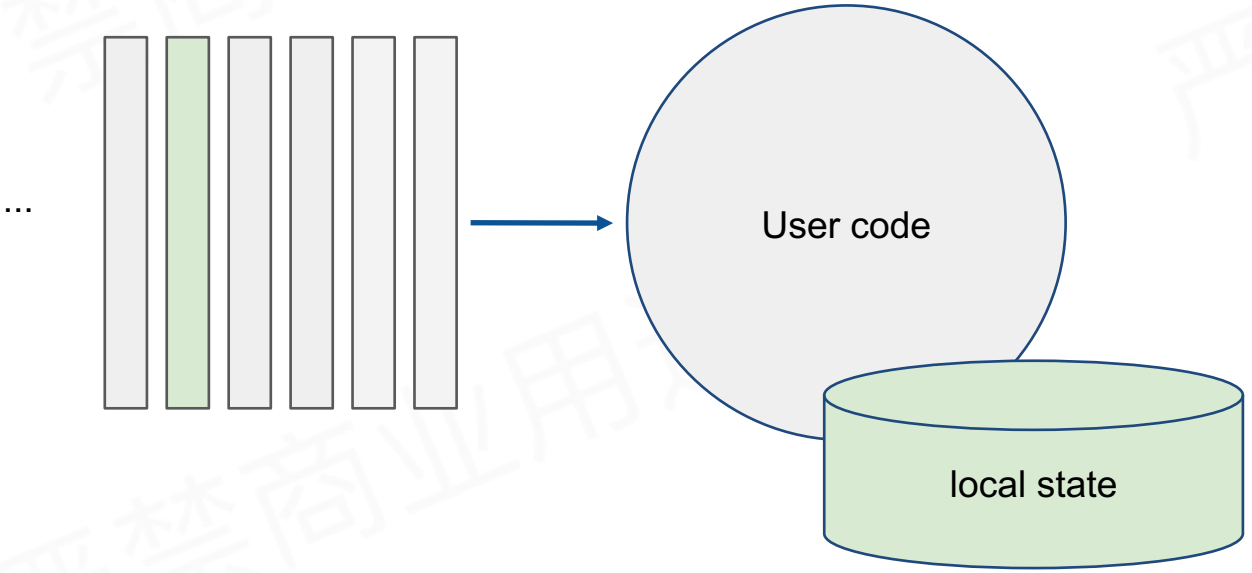
periodically snapshot the state + event log position

简易场景的精确一次容错方法



位置	状态
X	*State @ X*

简易场景的精确一次容错方法



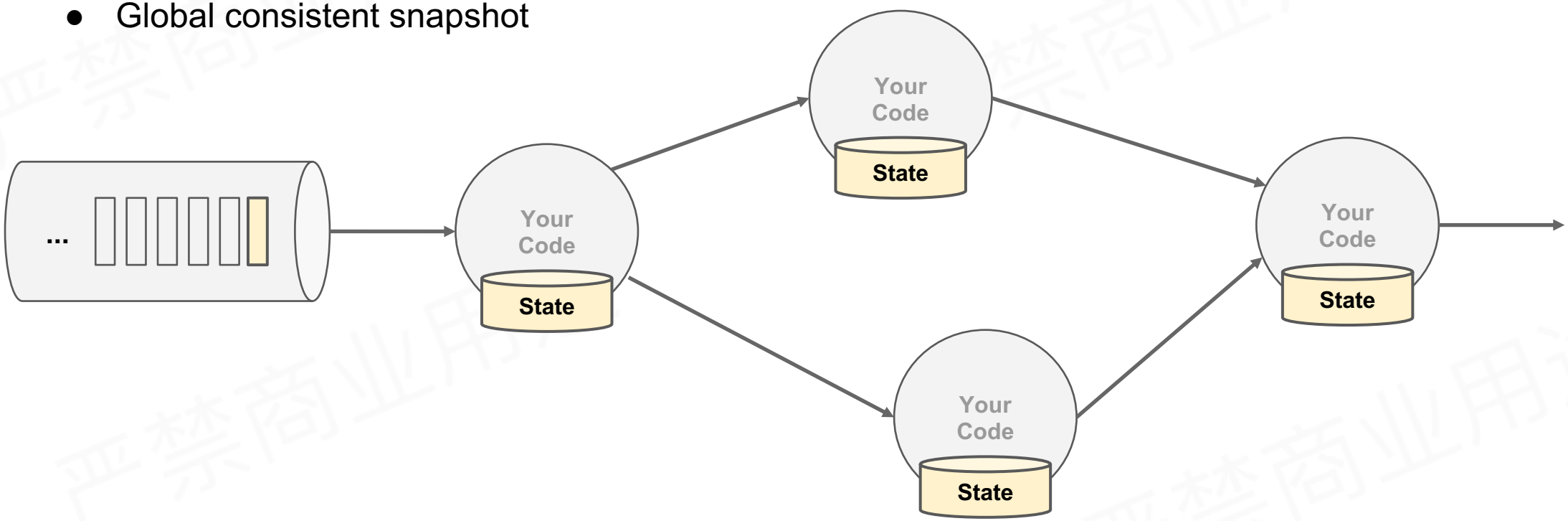
位置	状态
Y	*State @ Y*

状态容错

- 如何确保状态拥有**精确一次（exactly-once guarantee）**的容错保证？
- 如何在分散式场景下替多个拥有本地状态的运算子产生一个**全域一致的快照（global consistent snapshot）**？
- 更重要的是，如何在**不中断运算**的前提下产生快照？

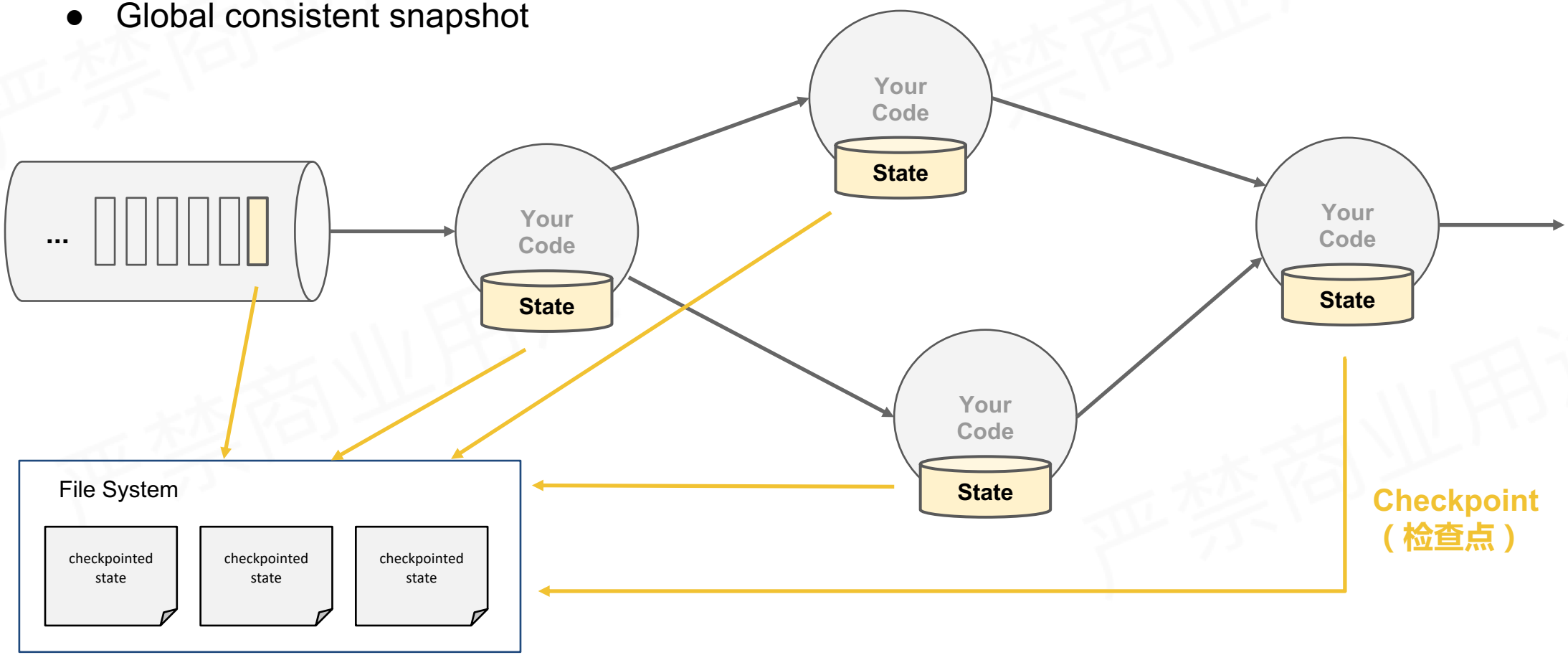
分散式状态容错

- Global consistent snapshot



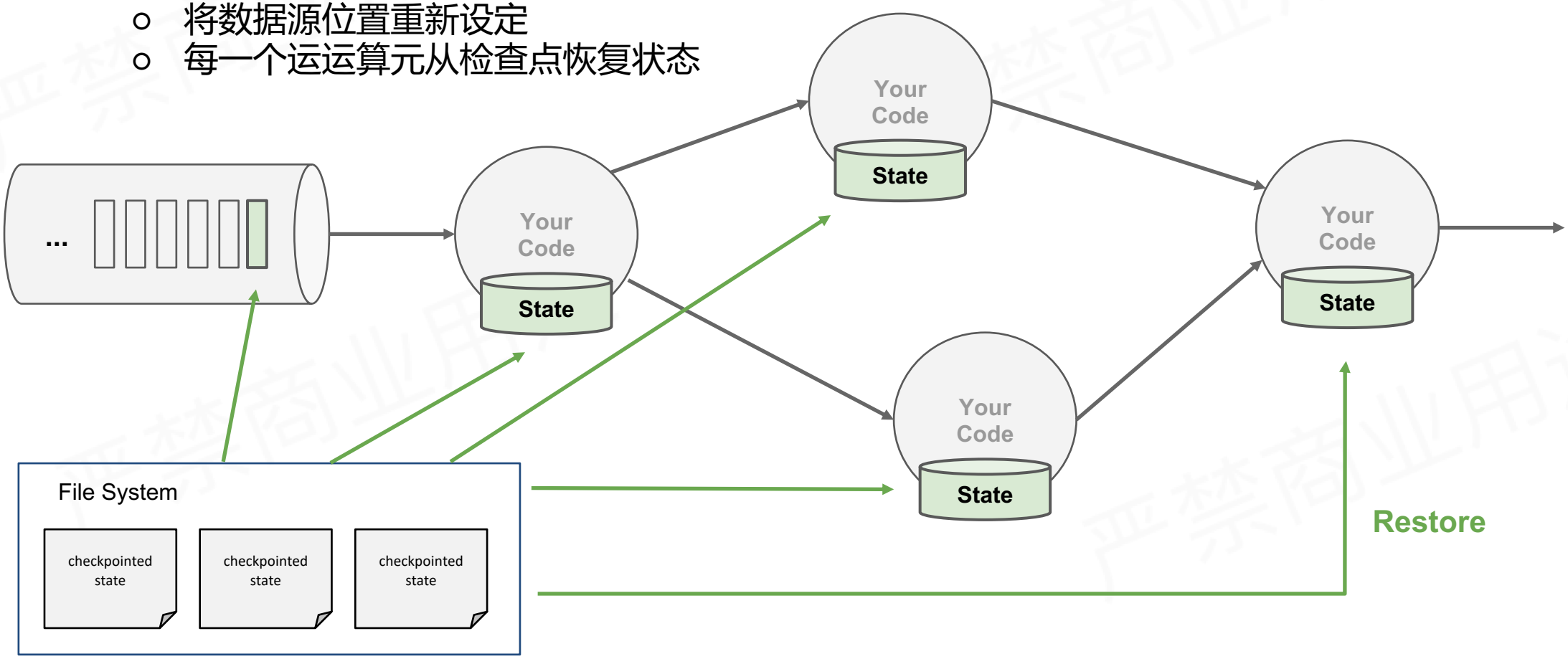
分散式状态容错

- Global consistent snapshot

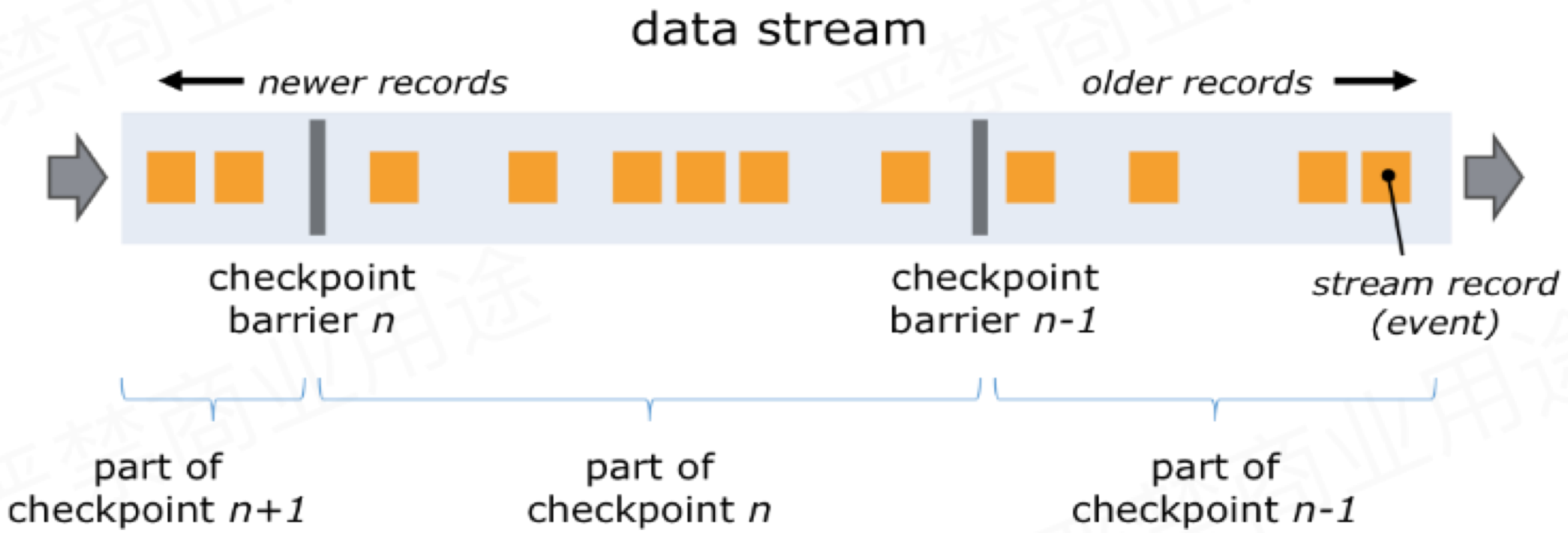


分散式状态容错

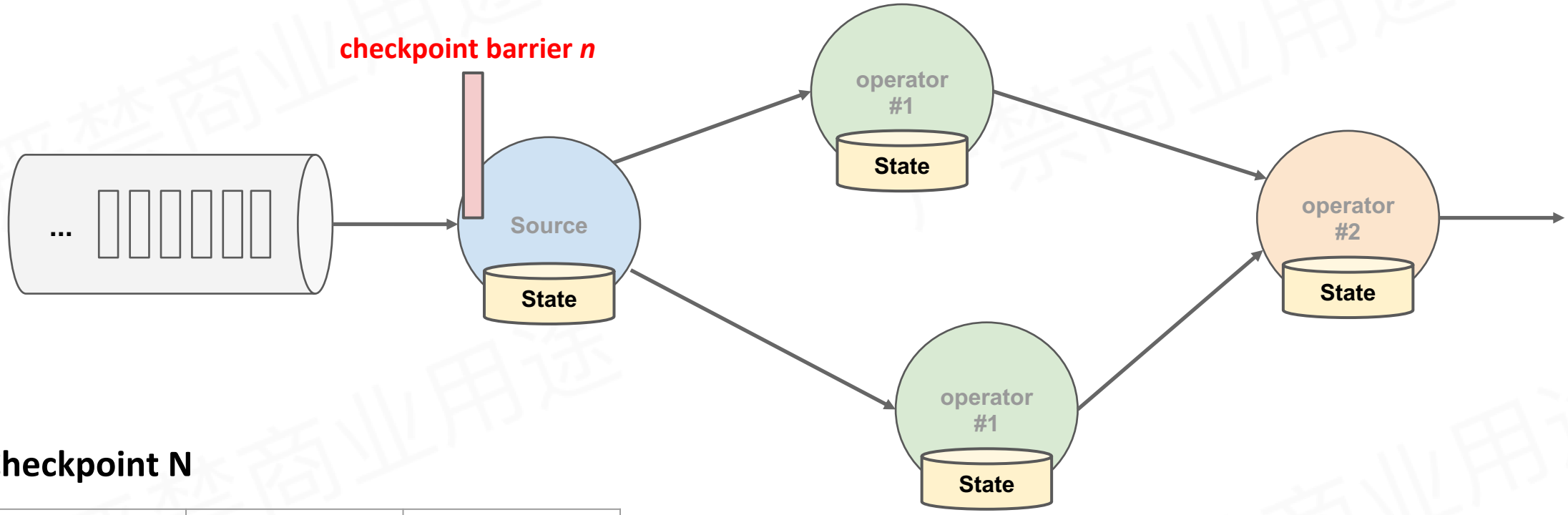
- 容错恢复：
  - 将数据源位置重新设定
  - 每一个运运算元从检查点恢复状态



分散式快照（Distributed Snapshots）方法



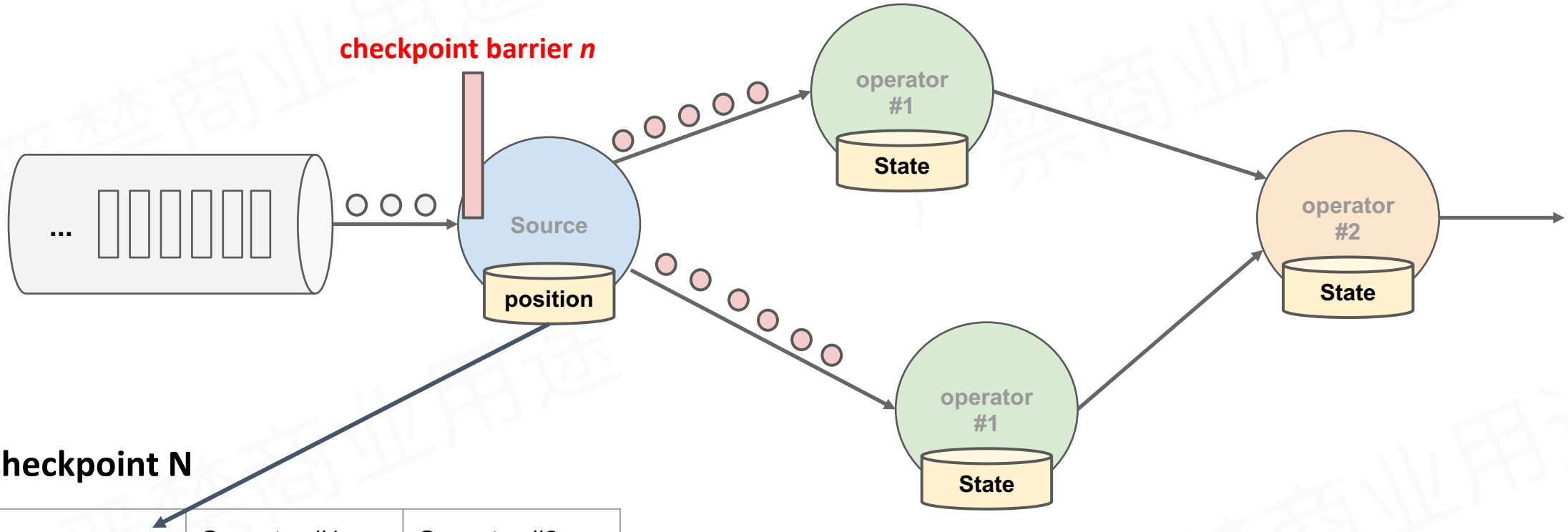
分散式快照（Distributed Snapshots）



Checkpoint N

Source	Operator #1	Operator #2

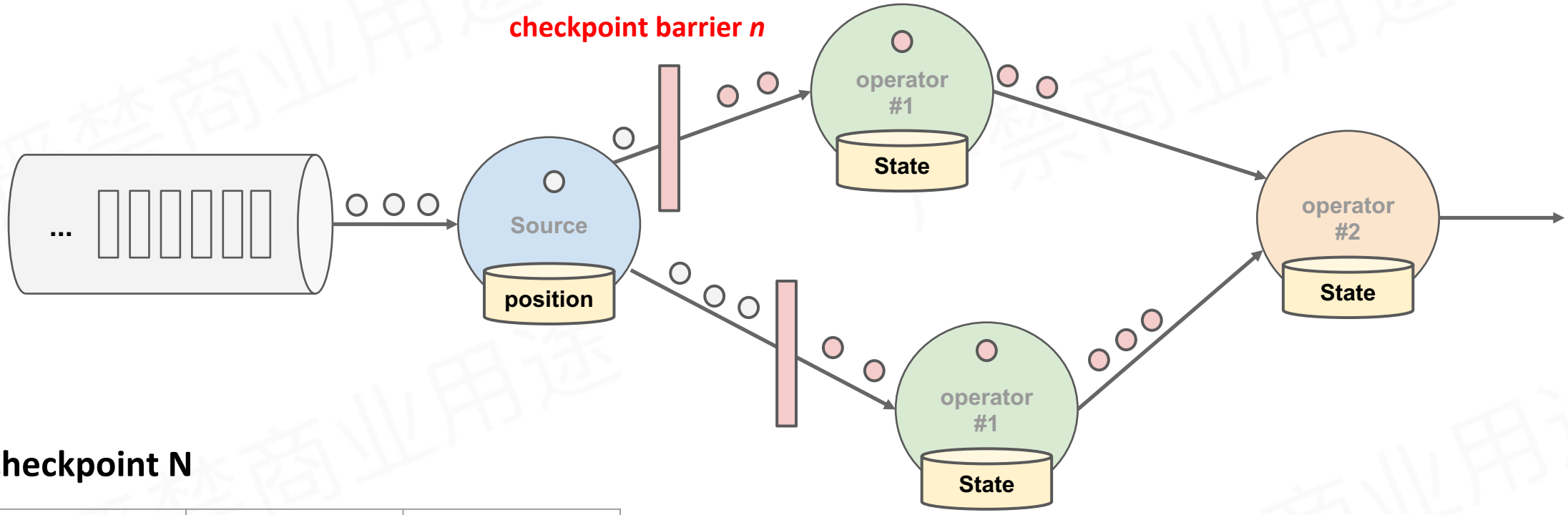
分散式快照（Distributed Snapshots）



Checkpoint N

Source	Operator #1	Operator #2
#partition offset		

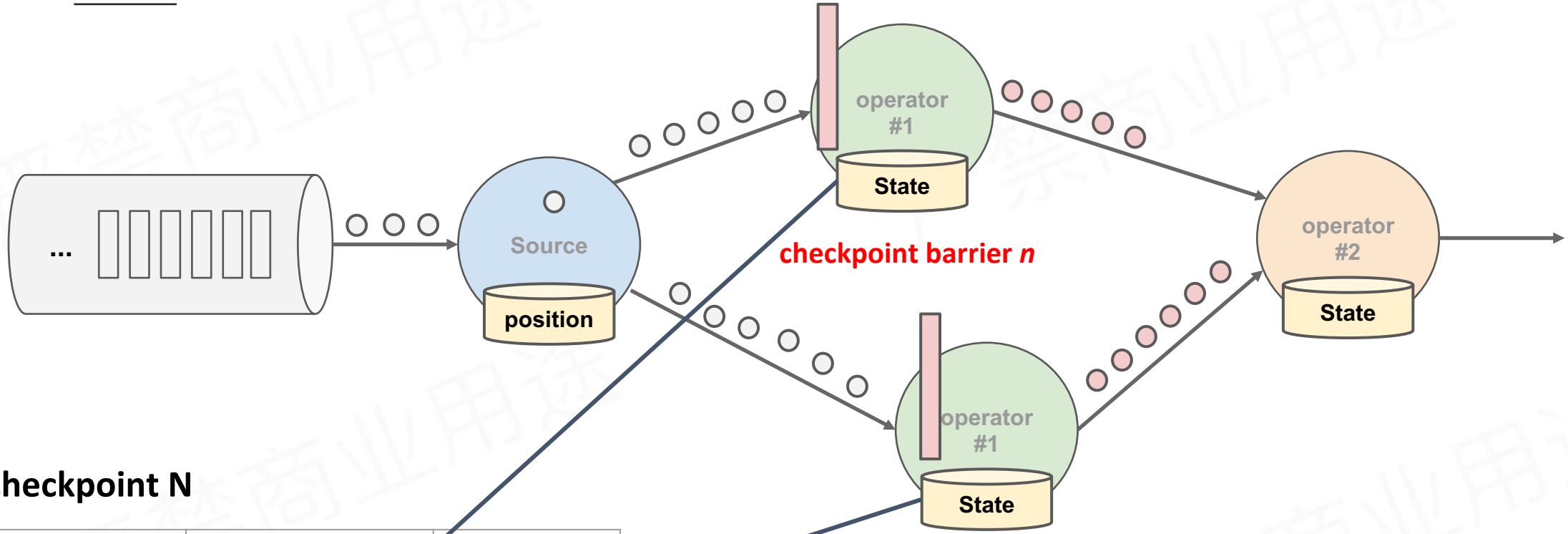
分散式快照（Distributed Snapshots）



Checkpoint N

Source	Operator #1	Operator #2
#partition offset		

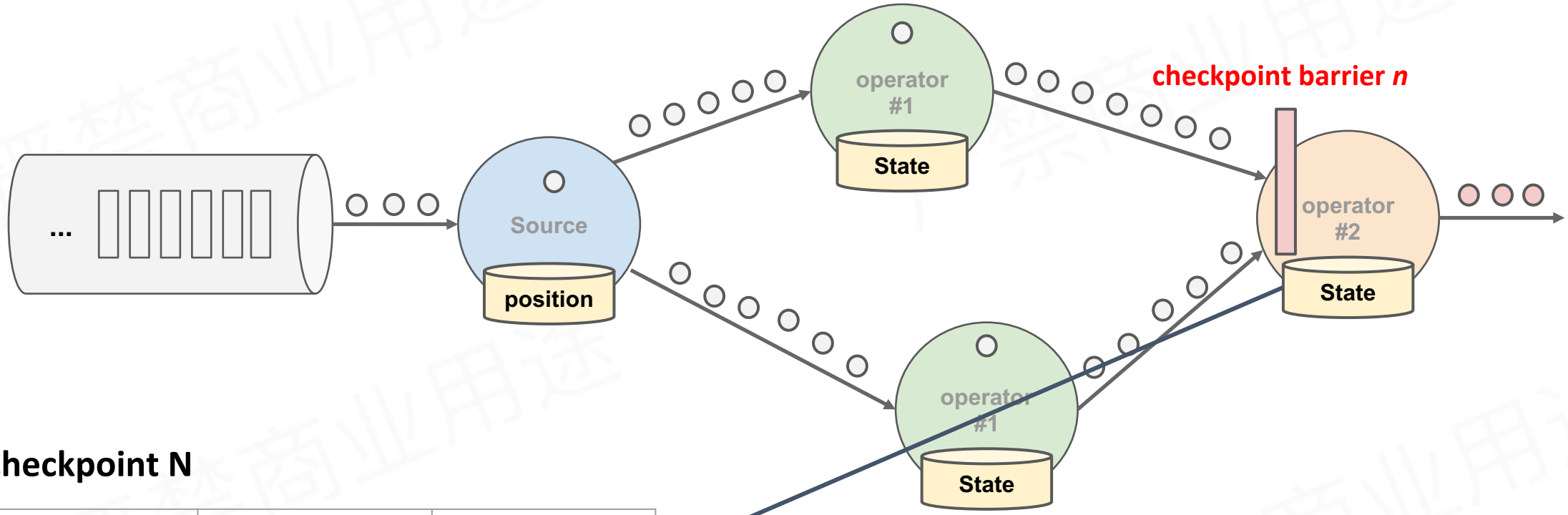
分散式快照（Distributed Snapshots）



Checkpoint N

Source	Operator #1	Operator #2
#partition offset	subtask #1 state	
	subtask #2 state	

分散式快照（Distributed Snapshots）



Checkpoint N

Source	Operator #1	Operator #2
#partition offset	subtask #1 state	subtask #1 state
	subtask #2 state	

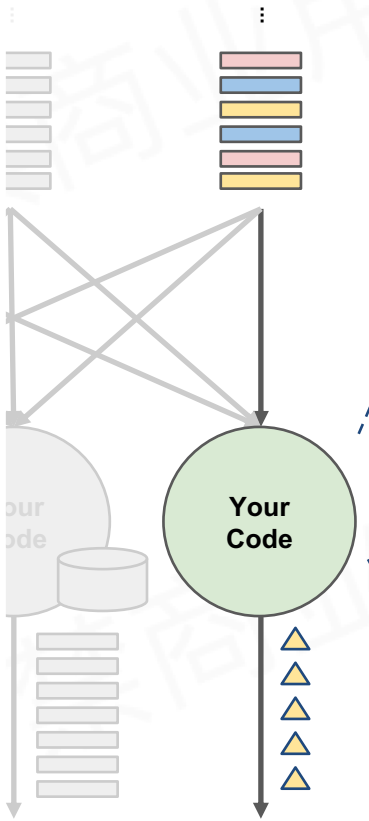
状态容错（State Fault Tolerance）

2.2 / 状态维护（State Management）

Event-time 处理（Event-time processing）

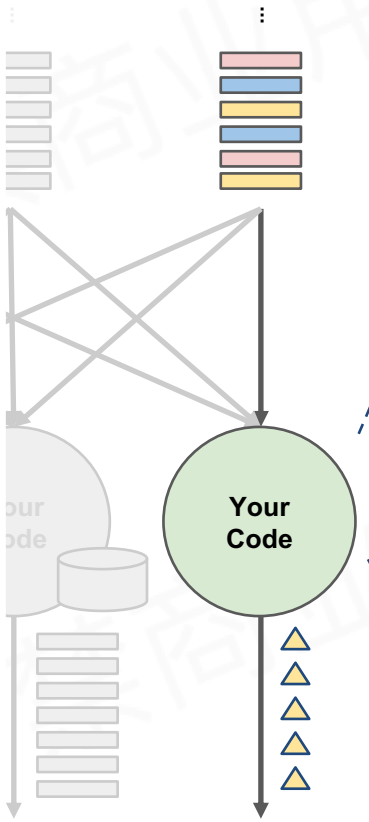
状态保存与迁移（Savepoints and Job Migration）

状态维护




```
val desc =
    new ValueStateDescriptor[MyPojo](
        “my-value-state”,
        createTypeInformation[MyPojo]
    )

val state =
    getRuntimeContext().getState(desc);

val p = state.value();
...

state.update(...)
...
```

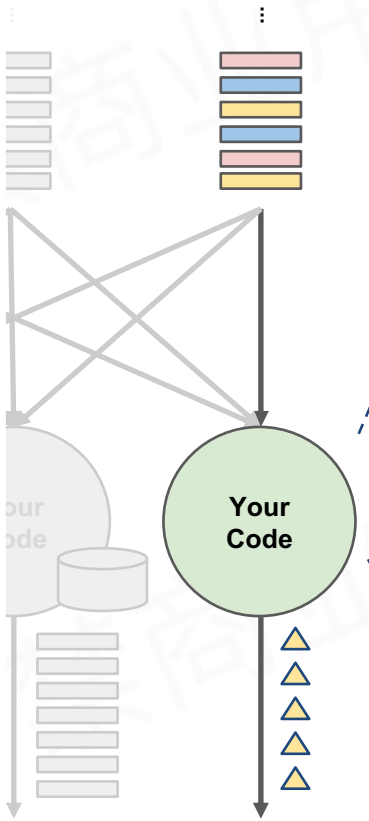
状态维护




```
val desc =
    new ValueStateDescriptor[MyPojo](
        “my-value-state”,
        createTypeInformation[MyPojo]
    )

val state =
    getRuntimeContext().getState(desc);

val p = state.value();
...

state.update(...)
...
```

状态识别 ID

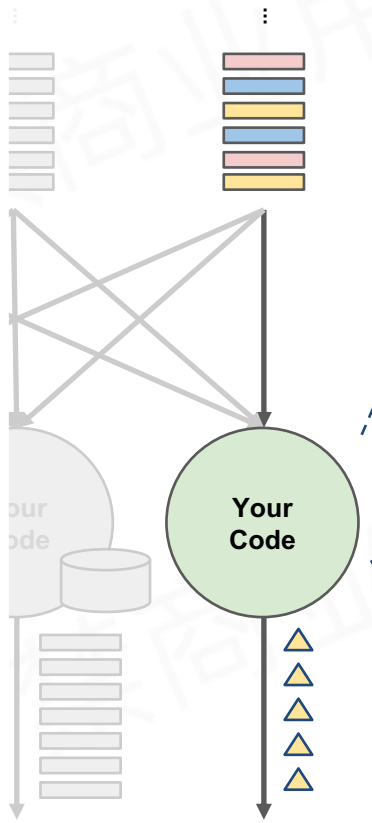
状态维护




```
val desc =
    new ValueStateDescriptor[MyPojo](
        “my-value-state”,
        createTypeInformation[MyPojo]
    )

val state =
    getRuntimeContext().getState(desc);

val p = state.value();
...

state.update(...)
...
```

状态数据型态资讯

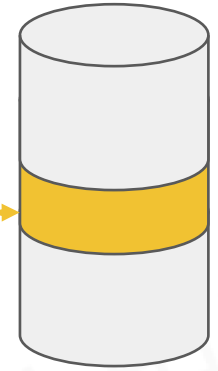
状态维护



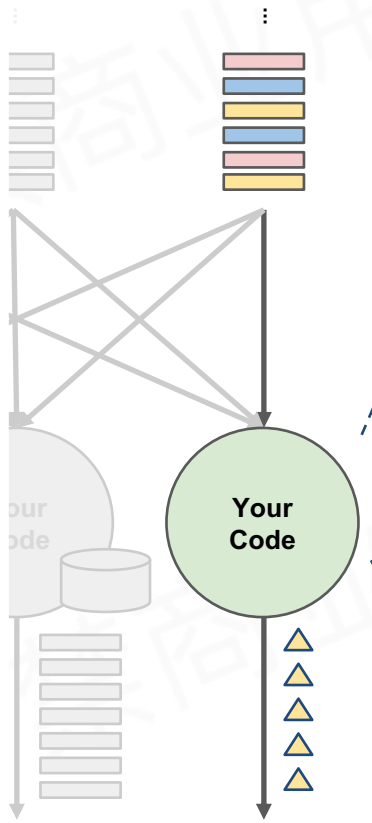

```
val desc =
    new ValueStateDescriptor[MyPojo](
        “my-value-state”,
        createTypeInformation[MyPojo]
    )

val state =
    getRuntimeContext().getState(desc);

val p = state.value();
...

state.update(...)
...
```

注册状态

本地状态后端

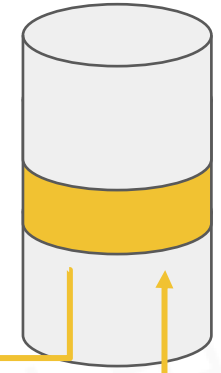


状态维护




```
val desc =
    new ValueStateDescriptor[MyPojo](
        “my-value-state”,
        createTypeInformation[MyPojo]
    )

val state =
    getRuntimeContext().getState(desc);

val p = state.value();
...

state.update(...)
...
```

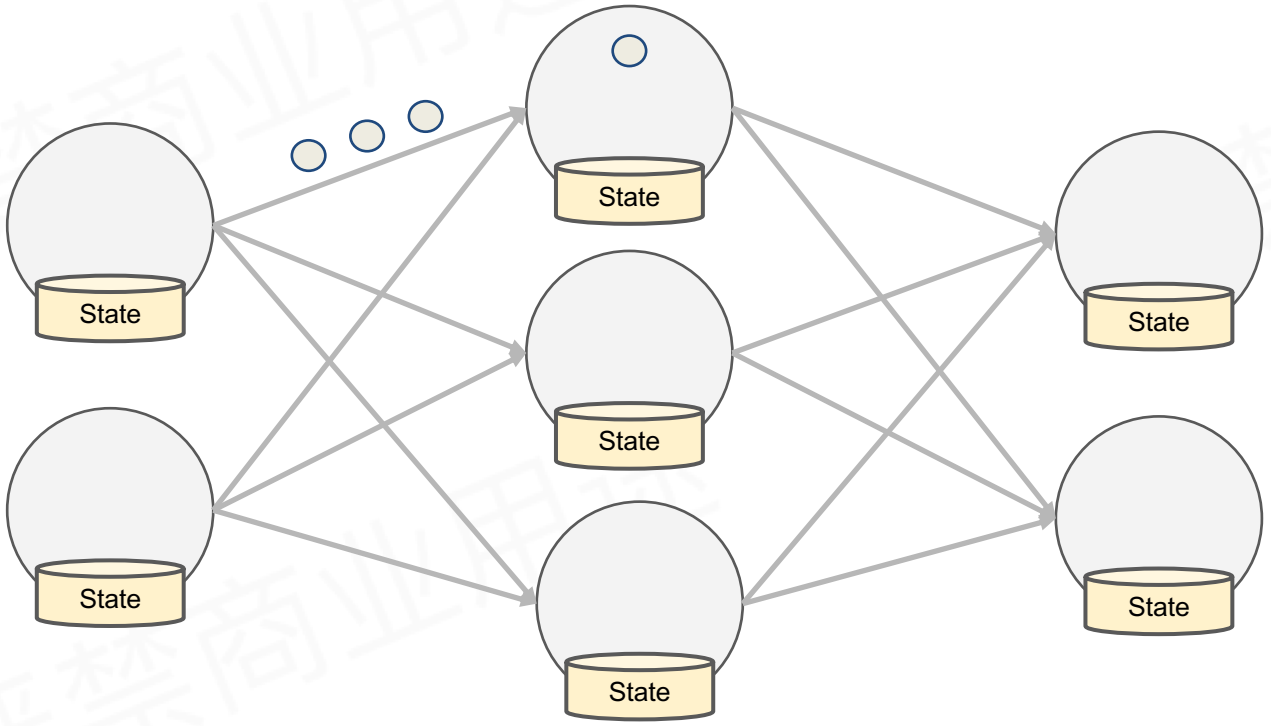
本地状态后端

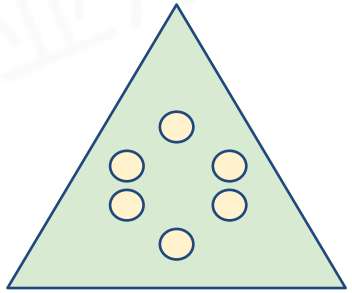


read

write

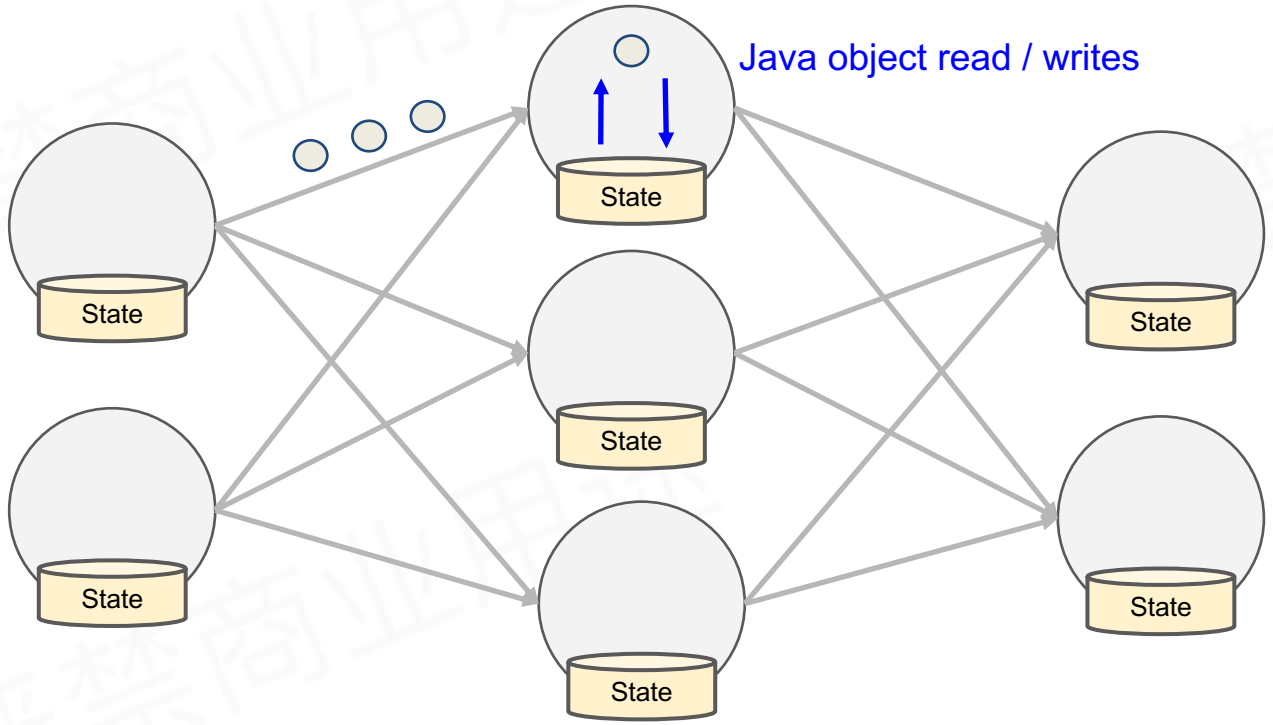
状态维护

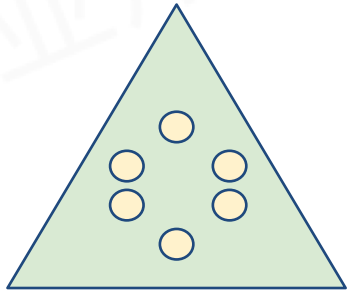




JVM Heap **状态后端**
(`MemoryStateBackend`, `FsStateBackend`)

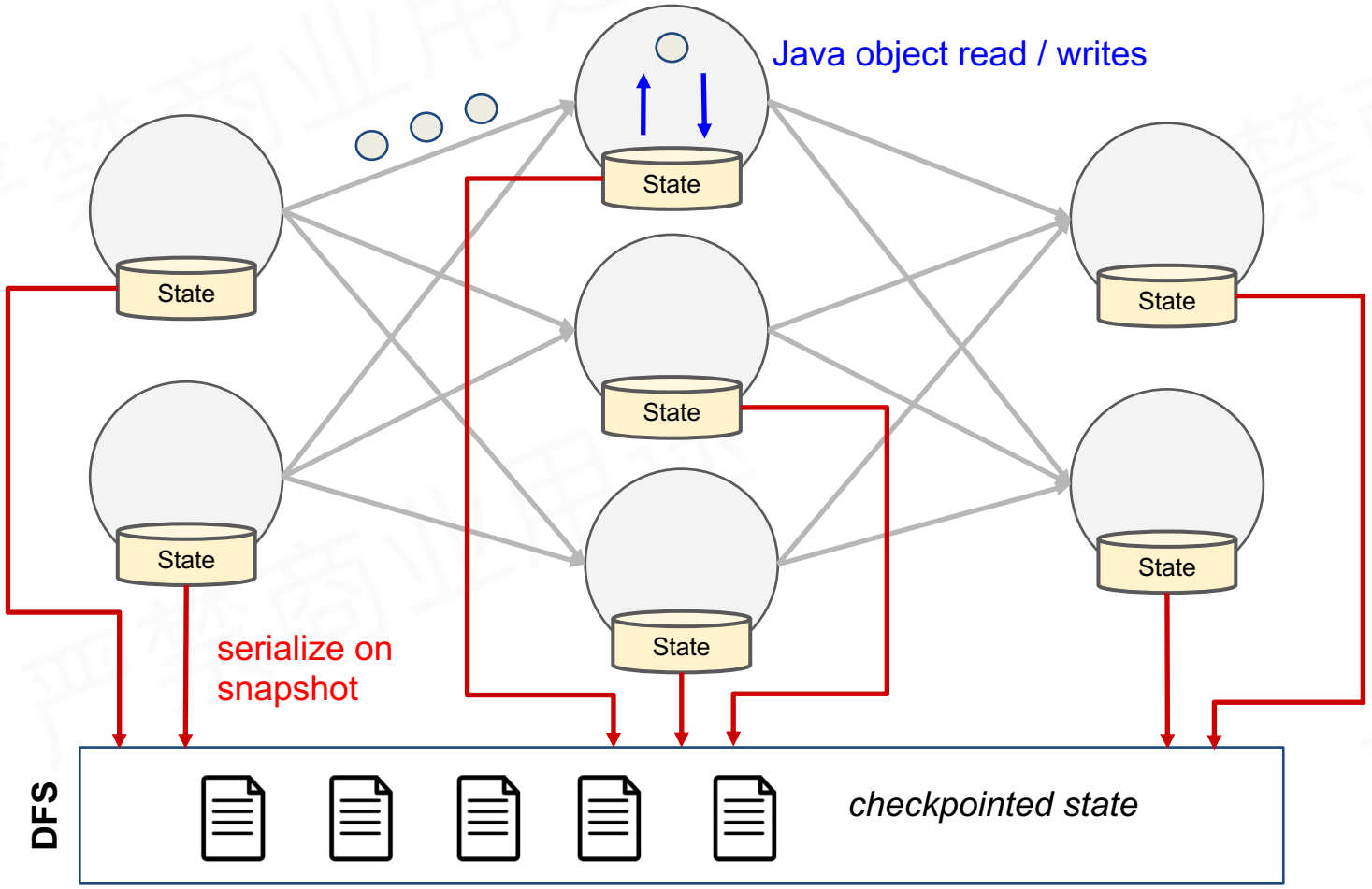
状态维护

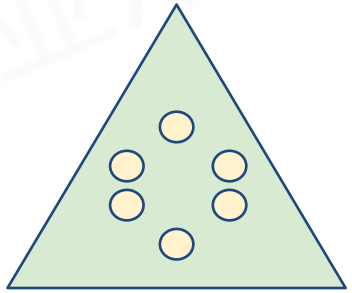




JVM Heap **状态后端**
(`MemoryStateBackend`, `FsStateBackend`)

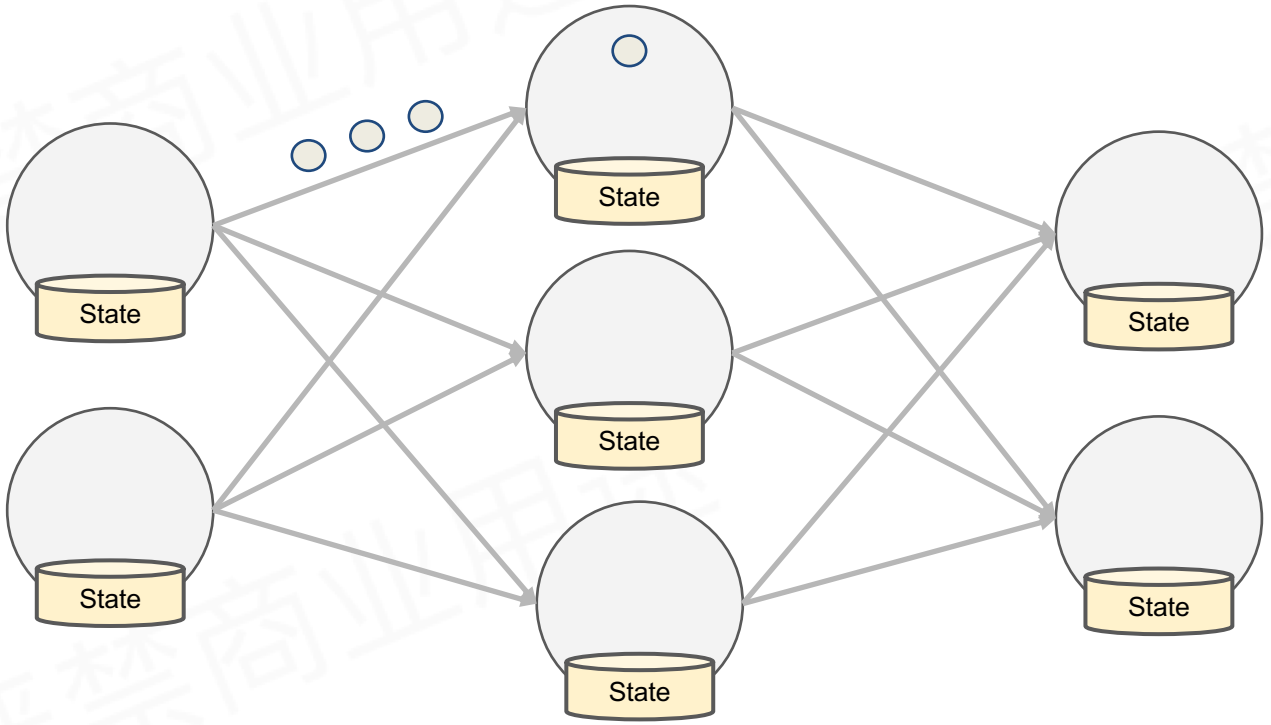
状态维护





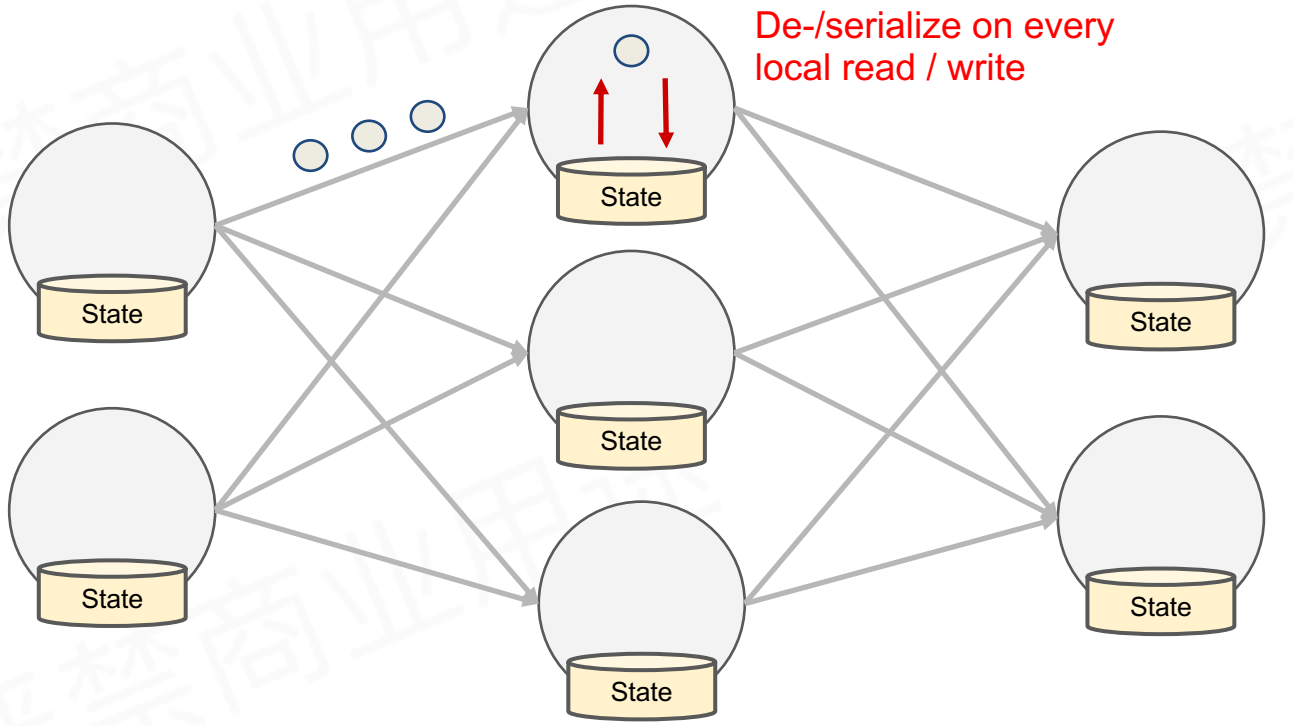
JVM Heap **状态后端**
(`MemoryStateBackend`, `FsStateBackend`)

状态维护



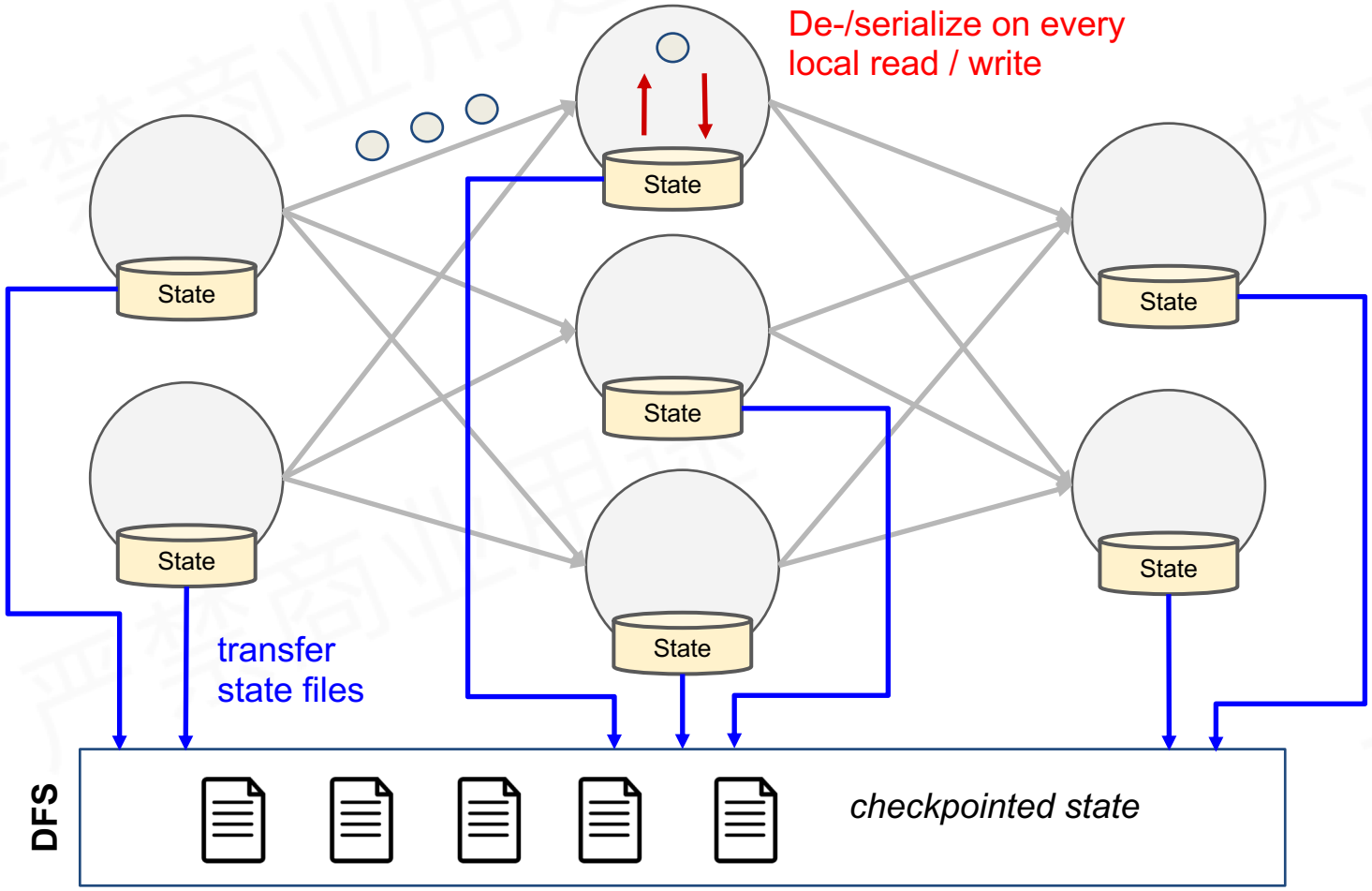
RocksDB **状态后端**

状态维护



RocksDB **状态后端**

状态维护



RocksDB **状态后端**

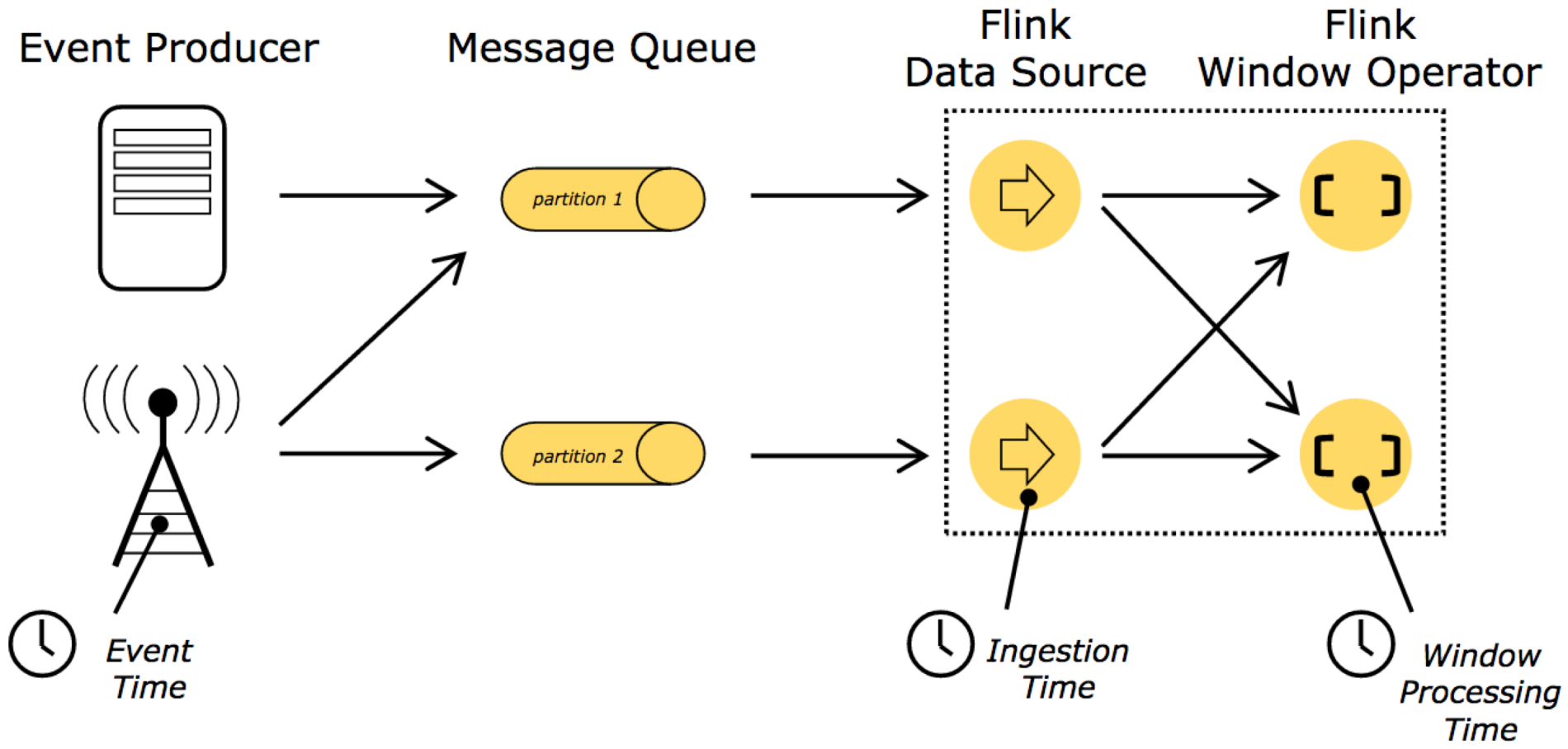
状态容错（State Fault Tolerance）

状态维护（State Management）

2.3 / Event-time 处理（Event-time processing）

状态保存与迁移（Savepoints and Job Migration）

不同时间种类

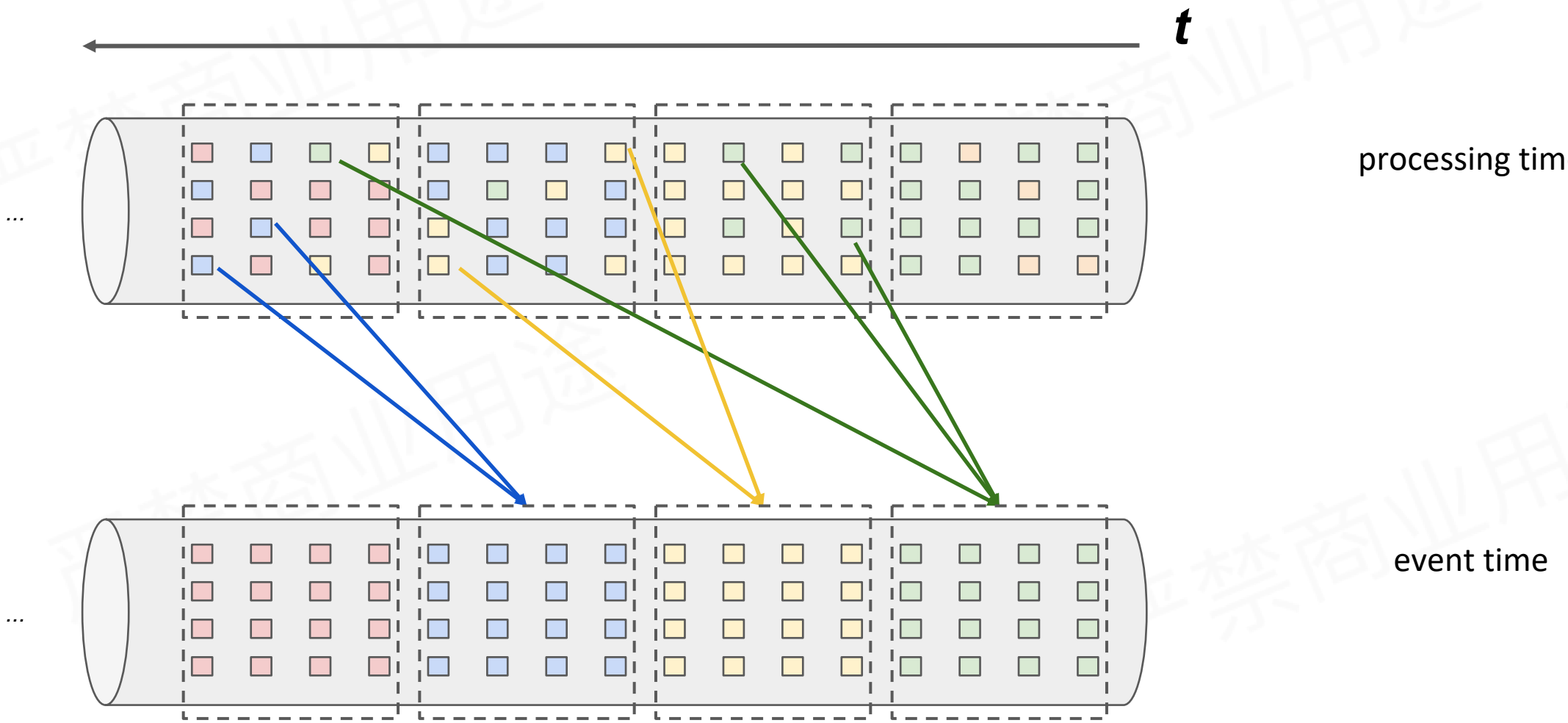


Event-Time 处理

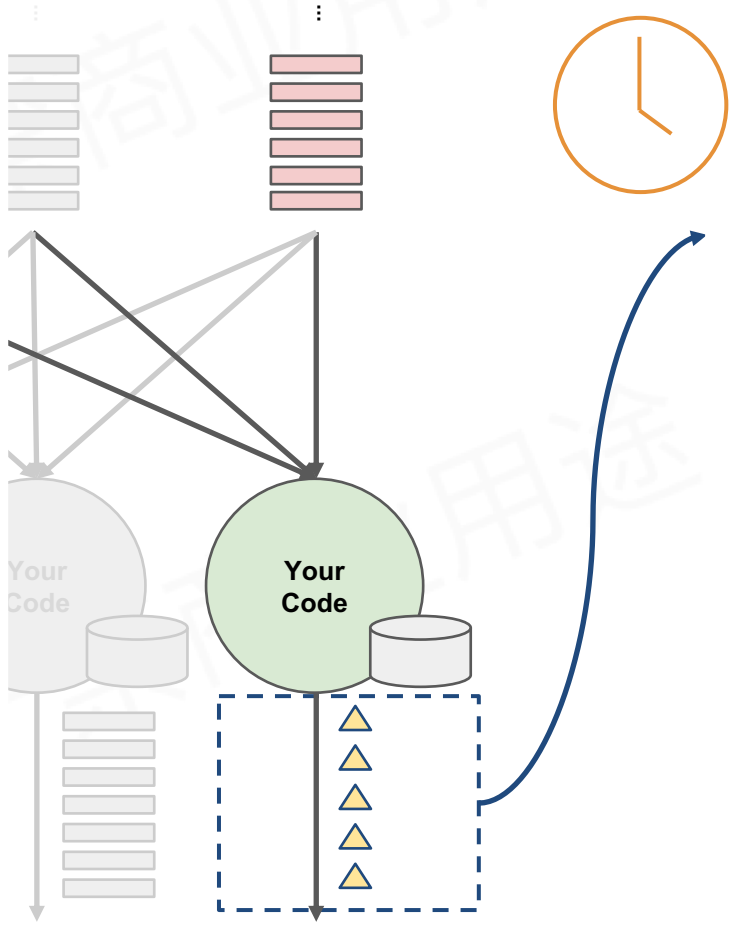
processing time

...

Event-Time 处理



Event-Time 处理

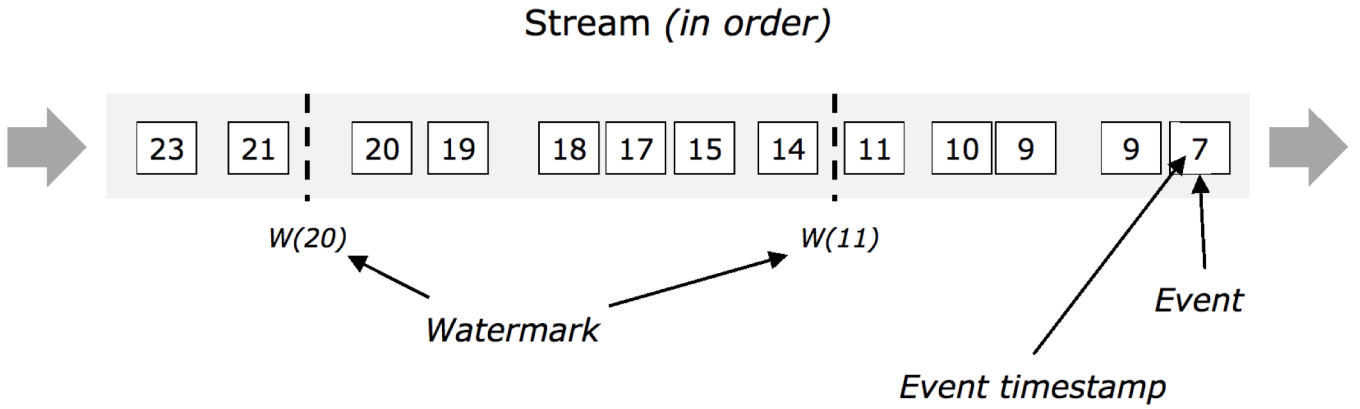


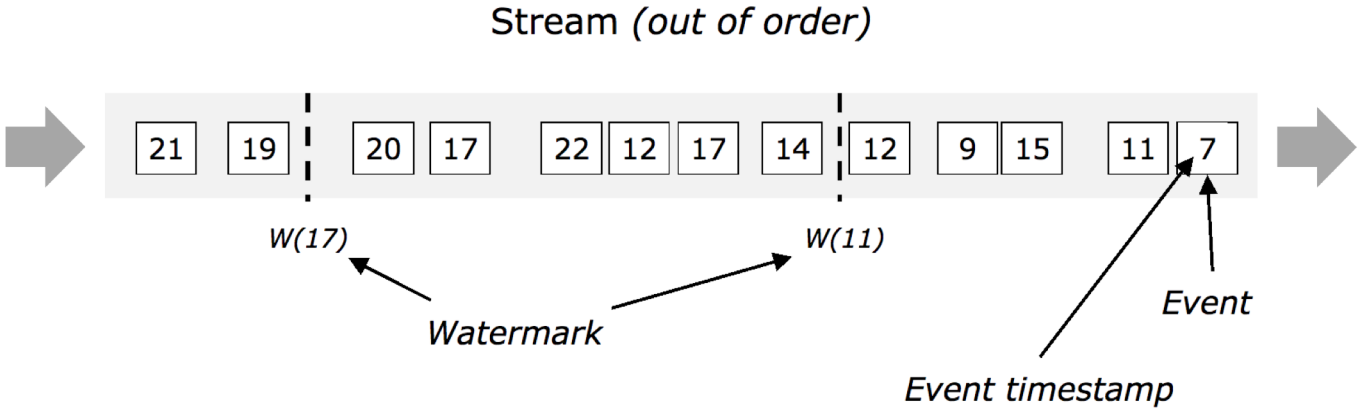
何时才能输出运算结果?

- 是否已经完整收到运算所需的所有数据？
- 这个问题通常都跟时间有关 e.g. 是否已经收到 3 - 4 pm 之间的所有数据?
- **要回答这个问题必须让处理引擎有 event-time 的认知**

Watermarks

- Watermarks 也是 Flink 中的特殊事件
- 一个带有时间戳 *t* 的 watermark 会让运运算元判定不会再收到任何时间戳 < *t* 的事件





状态容错（State Fault Tolerance）

状态维护（State Management）

Event-time 处理（Event-time processing）

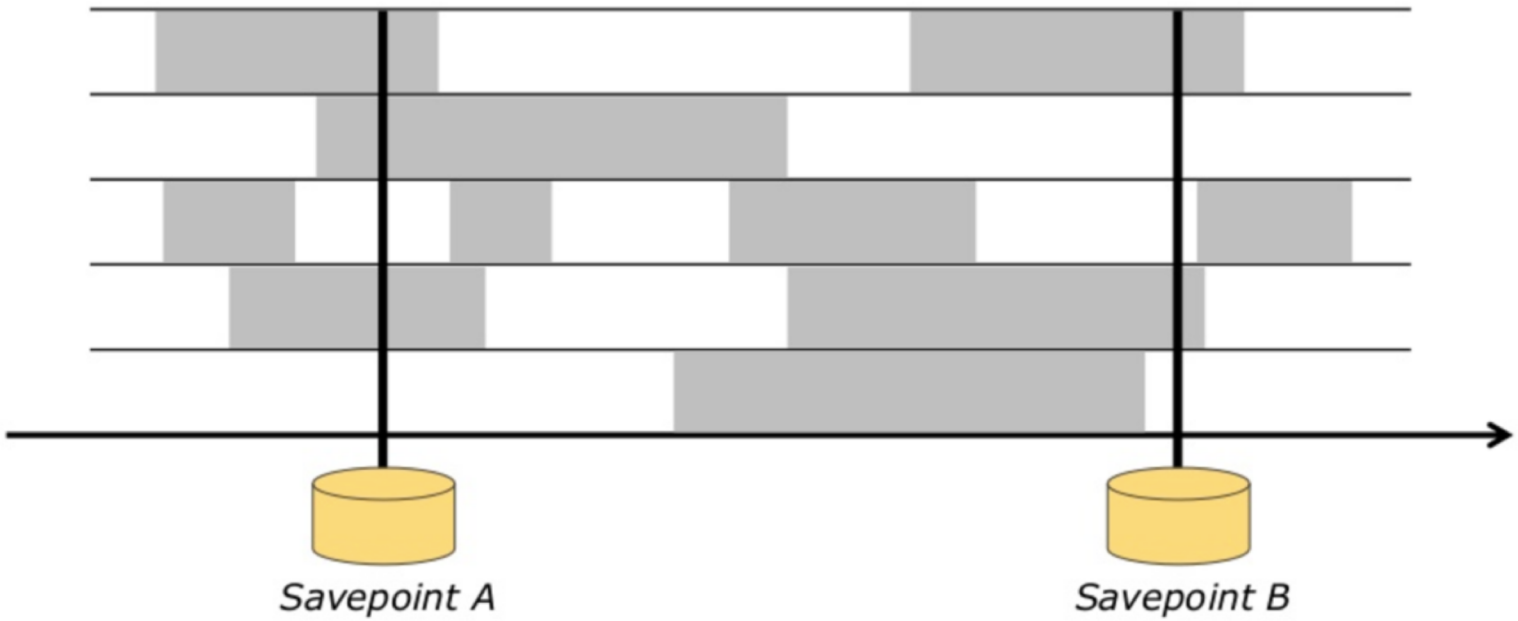
2.4 / 状态保存与迁移（Savepoints and Job Migration）

状态保存与迁移

- 流式处理应用是无时无刻在运行，维运上有几个重要考量：
  - 更改應用邏輯 / 修 bug 等，如何將前一執行的狀態遷移到新的執行？
  - 如何重新定义运行的平行化程度？
  - 如何升级运算丛集的版本号？

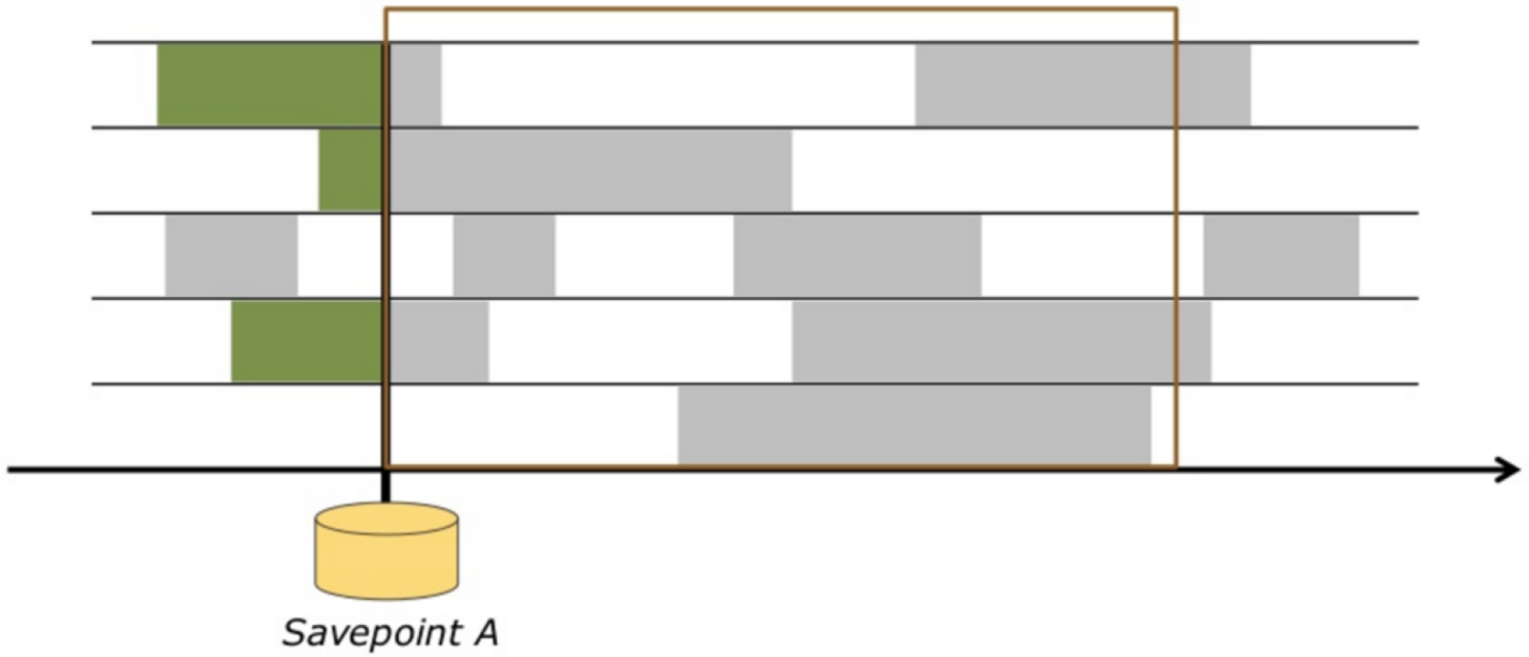
保存点 (Savepoint)

- 可以想成：一个手动产生的检查点（Checkpoint）
  - 保存点记录着流式应用中所有运运算元的状态



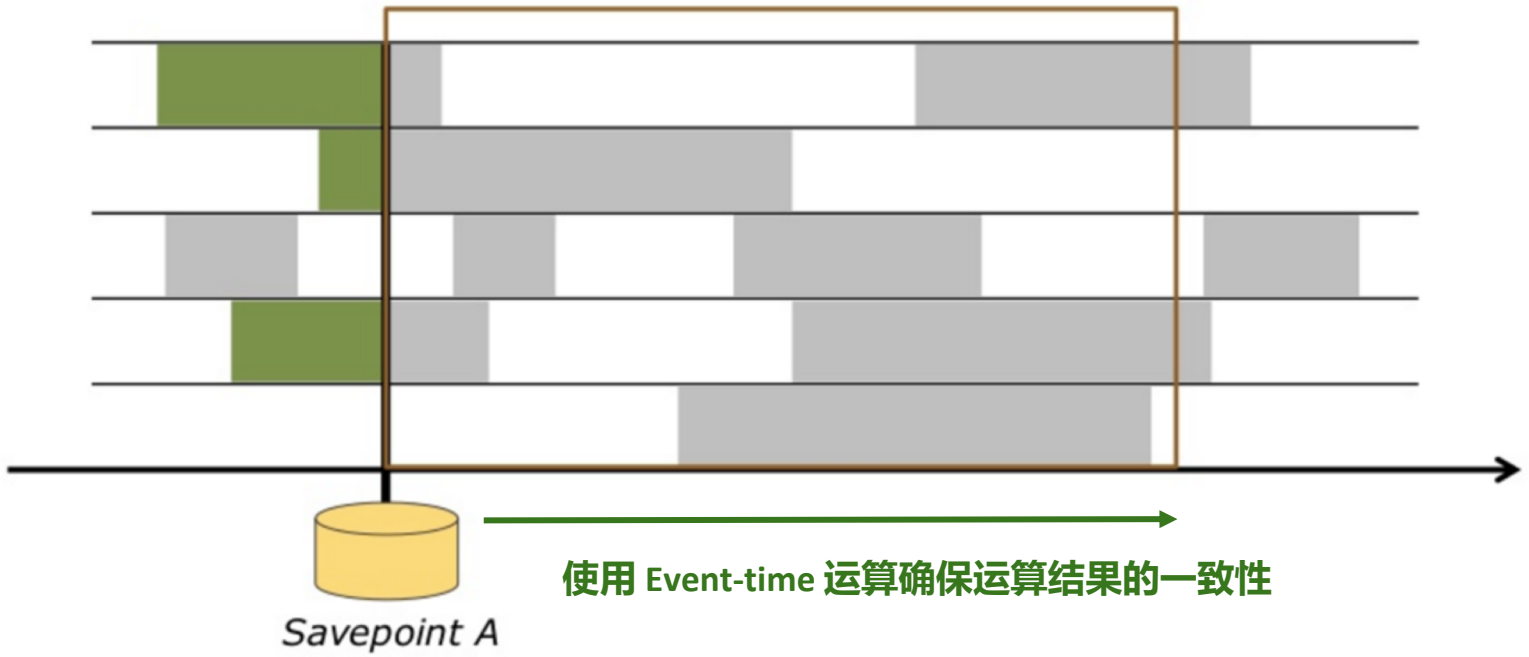
保存点 (Savepoint)

- 执行停止之前，产生一个保存点



保存点 (Savepoint)

- 从保存点恢复新的执行，并且利用 Event-time 处理赶上最新的数据



03

总结

Apache Flink

- **状态容错**: 精确一次保证，分布式快照（ Distributed Snapshots ）
- **可应付极大的状态量 (TB+ scale)**: out-of-core 状态后端, asynchronous 快照
- **状态迁移**: 在应用重新平行化 / 更动应用代码的状况下仍能恢复历史状态
- Event-time **处理**: 用以定义何时接收完毕所需数据

Apache Flink

- **状态容错**: 精确一次保证，分布式快照（ Distributed Snapshots ）
- **可应付极大的状态量 (TB+ scale)**: out-of-core 状态后端, asynchronous 快照
- **状态迁移**: 在应用重新平行化 / 更动应用代码的状况下仍能恢复历史状态
- Event-time **处理**: 用以定义何时接收完毕所需数据

THANKS