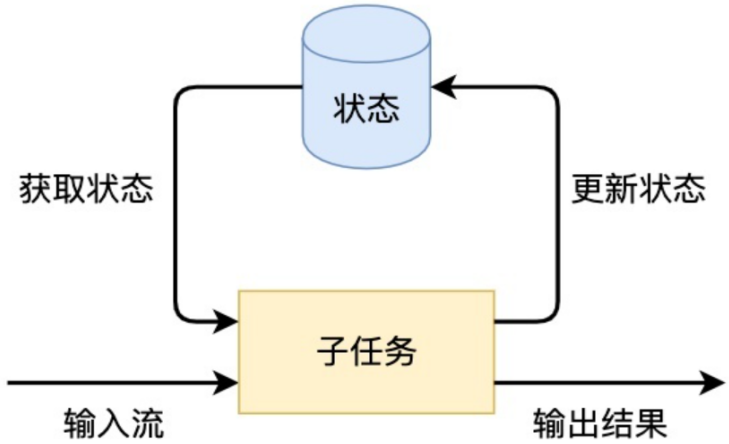
为什么要管理状态

- 稍复杂的流处理场景都需要记录状态，然后在新流入数据的基础上不断更新状态



1.1 什么是状态

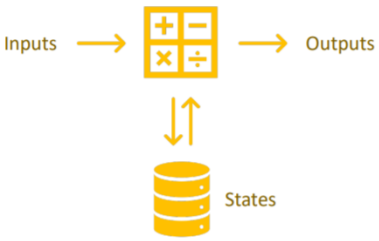
有状态计算的例子：访问量统计

- 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



状态分类

- Flink有两种基本类型的状态：托管状态（**Managed State**）和原生状态（**Raw State**）。对Managed State继续细分，它又有两种类型：**Keyed State**和**Operator State**。

	Managed State	Raw State
状态管理方式	Flink Runtime托管，自动存储、自动恢复、自动伸缩	用户自己管理
状态数据结构	Flink提供的常用数据结构，如ListState、MapState等	字节数组：byte[]
使用场景	绝大多数Flink算子	用户自定义算子

以上两者的具体区别是

- 从状态管理的方式上来说，Managed State由Flink Runtime托管，状态是自动存储、自动恢复的，Flink在存储管理和持久化上做了一些优化。当我们横向伸缩，或者说我们修改Flink应用的并行度时，状态也能自动重新分布到多个并行实例上。Raw State是用户自定义的状态
- 从状态的数据结构上来说，Managed State支持了一系列常见的数据结构，如ValueState、ListState、MapState等。Raw State只支持字节，任何上层数据结构需要序列化为字节数组。使用时，需要用户自己

序列化，以非常底层的字节数组形式存储，Flink并不知道存储的是什么样的数据结构

- 从具体使用场景来说，绝大多数的算子都可以通过继承Rich函数类或其他提供好的接口类，在里面使用Managed State。Raw State是在已有算子和Managed State不够用时，用户自定义算子时使用

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

- OperatorState 没有current key 概念
- KeyedState 的数值总是与一个current key对应的

current key

- OperatorState 只有堆内存一种实现
- KeyedState 有堆内存和RocksDB两种实现

heap

- OperatorState 需要手动实现snapshot和restore方法
- KeyedState由backend实现，对用户透明

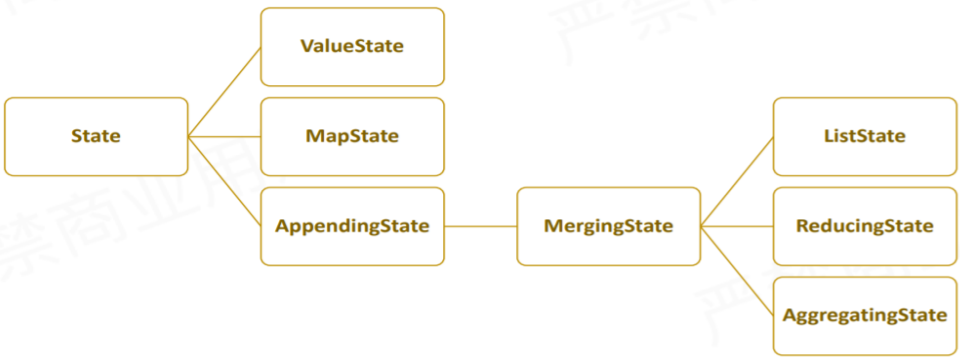
snapshot

- OperatorState 一般被认为是规模比较小的
- KeyedState 一般是相对规模较大的

Size

2.3 Keyed State 使用示例

- 几种 Keyed State 之间的关系



- 几种 Keyed State 之间的差异

	状态数据类型	访问接口
ValueState	单个值	• update(T) / T value()
MapState	Map	• put(UK key, UV value) / putAll(Map<UK,UV> map) • remove(UK key) • boolean contains(UK key) / UV get(UK key) • Iterable<Map.Entry> entries() / Iterator<Map.Entry> iterator() • Iterable<UK> keys() / Iterable<UV> values()
ListState	List	• add(T) / addAll(List<T>) • update(List<T>) / Iterable<T> get()
ReducingState	单个值	• add(T) / addAll(List<T>) • update(List<T>) / T get()
AggregatingState	单个值	• add(IN) / OUT get()

状态如何保存与恢复

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);
```

	Checkpoint	Savepoint
触发管理方式	• 由 Flink 自动触发并管理	• 由用户手动触发并管理

主要用途	• 在 Task 发生异常时快速恢复
	  • 例如网络抖动导致的超时异常
		• 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

特点	• 轻量 • 自动从故障中恢复 • 在作业停止后默认清除	• 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

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**

- **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

- **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