索信达
Suoxinda Holdings Limited

Flink基本概念及入门实战

项目服务一部/袁海龙

2019-08

索信达
Suoxinda Holdings Limited

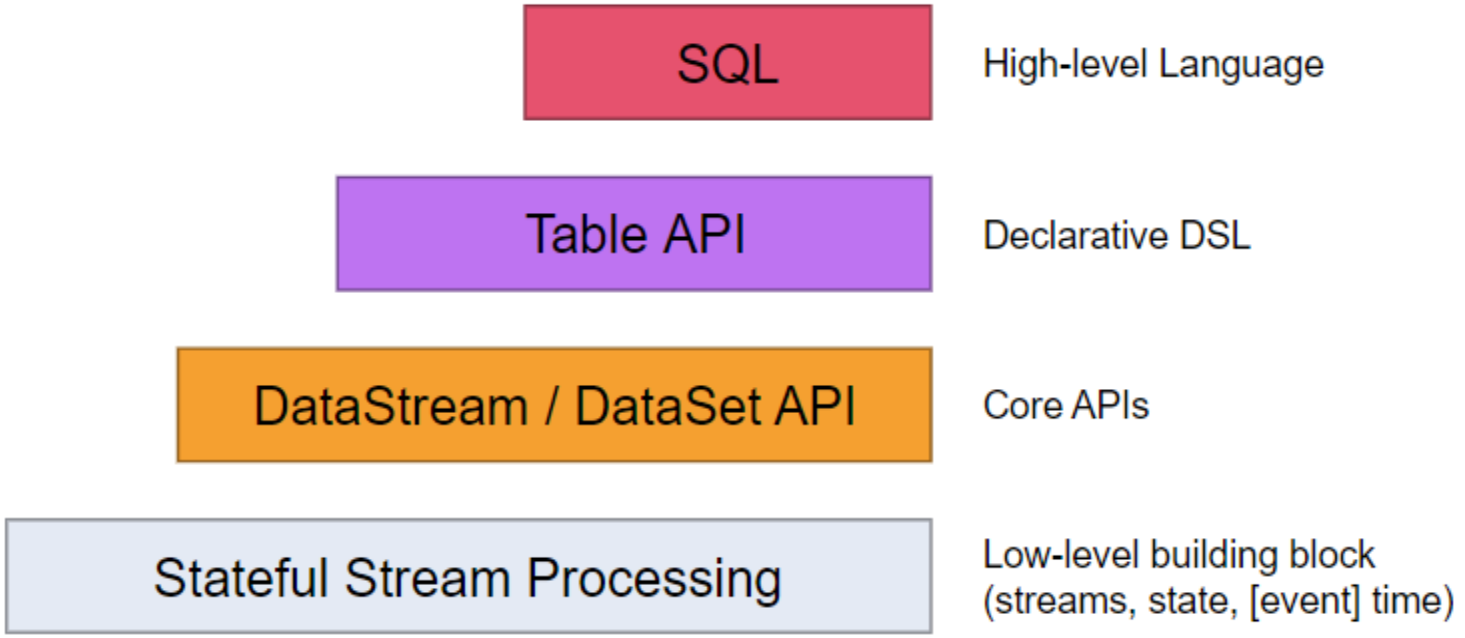
CONTENT

索信达
Suoxinda Holdings Limited

1 Flink编程模型

Flink分层架构

- Stateful Stream Processing
- Core API
- Table
- SQL



https://ci.apache.org/projects/flink/flink-docs-master/concepts/programming-model.html

Stateful Stream Processing

- 提供有状态流最低层的一个简单抽象
- Process Function
- 状态容错
- 灵活度高，但是开发比较复杂
- 能从一个或多个Stream中获取数据

Core API

- DataStream: Stream
- DataSet: Batch

Table & SQL

- Table
  - 动态修改Table（stream）
  - Table Schema
  - 支持各种关系型数据库的操作 如SELECT,JOIN,AGGREGATE等
- SQL
  - 最上层的API
  - 构建在Table上

https://ci.apache.org/projects/flink/flink-docs-master/dev/table/

Flink 构建流程

- 构建计算环境
- 创建Source
- 转换（利用各种算子）
- 结果集进行输出（Sink）

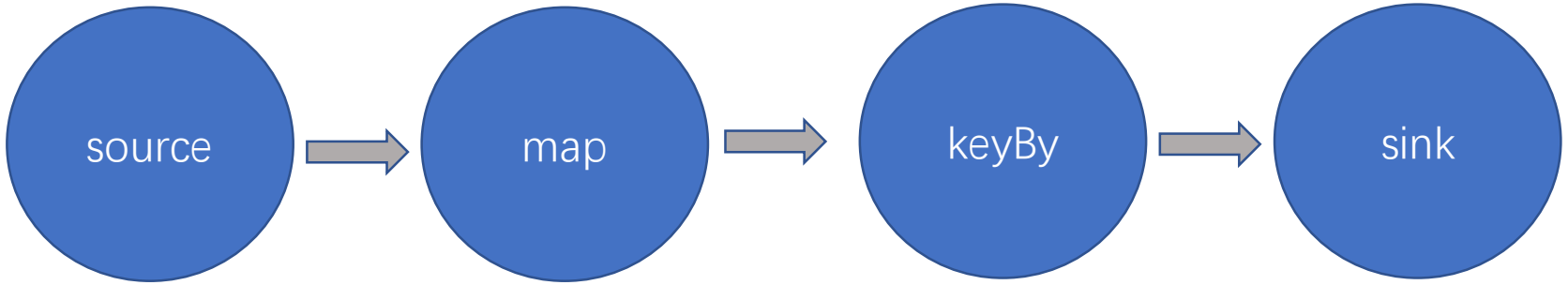
```
DataStream<String> lines = env.addSource(
                                new FlinkKafkaConsumer<>(…));

DataStream<Event> events = lines.map((line) -> parse(line));

DataStream<Statistics> stats = events
        .keyBy("id")
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());

stats.addSink(new RollingSink(path));
```

Source

Transformation

Transformation

Sink



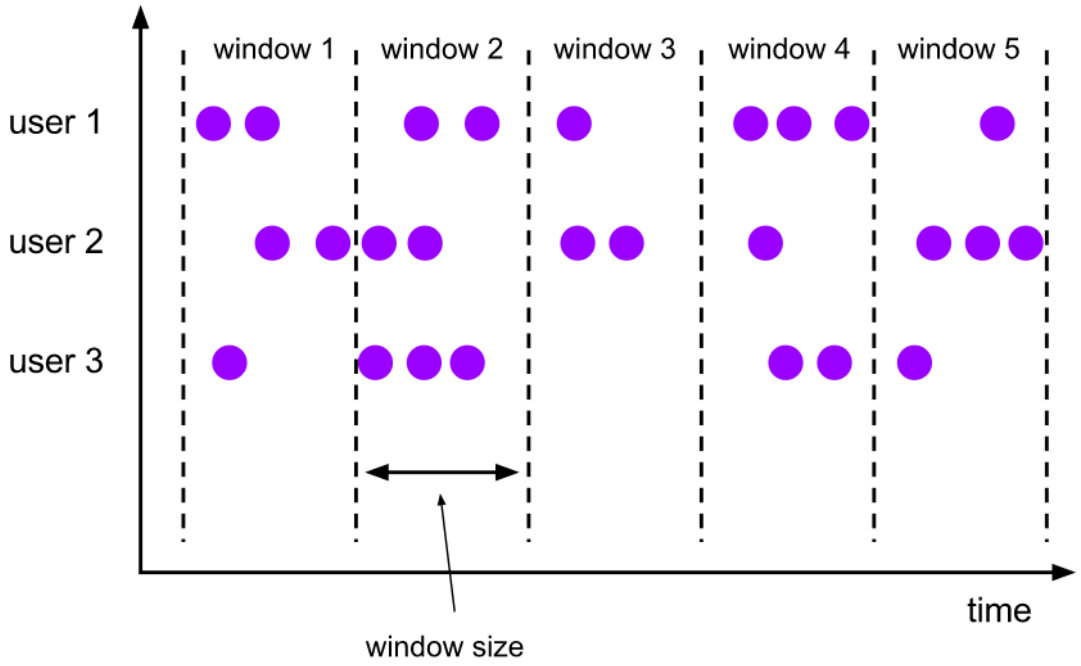
Window

- What
  - It's will infinite stream spilt into "bucket" of finite size for computations
- Window Type
  - Tumbling Windows
  - Sliding Windows
  - Session Windows
  - Globale Windows
- Window Function
- Triggers
- Evictors

https://ci.apache.org/projects/flink/flink-docs-master/dev/stream/operators/windows.html

Tumbing Window

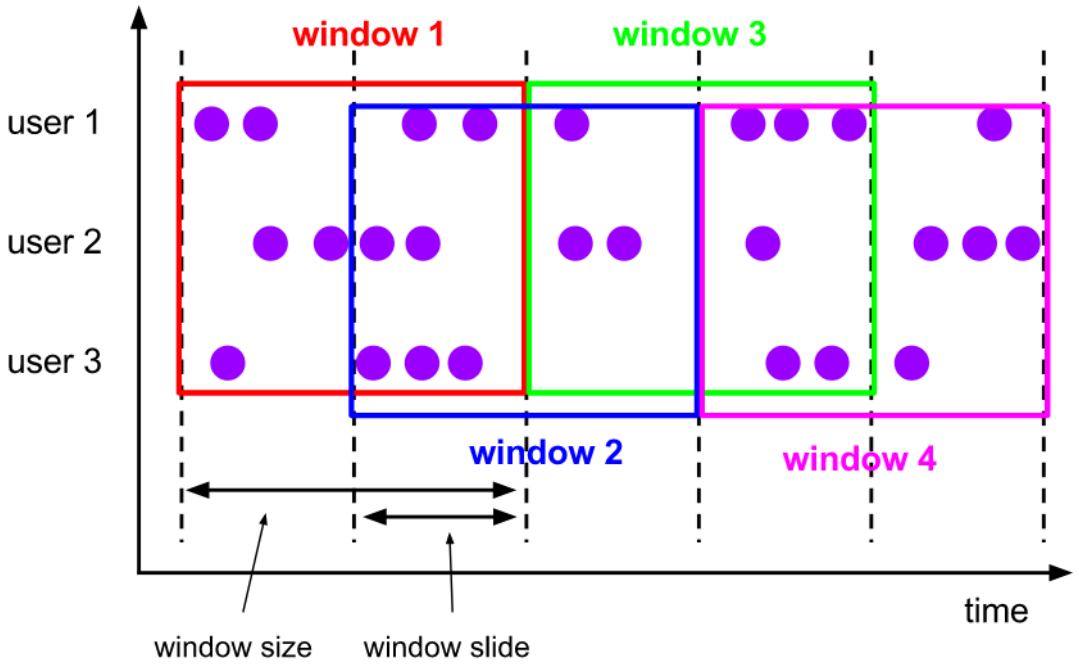
具有固定大小的窗口，
例 每五分钟一个窗口




```scala
val input: DataStream[T] = ...

// tumbling event-time windows
input
    .keyBy(<key selector>)
    .window(TumblingEventTimeWindows.of(Time.seconds(5)))
    .<windowed transformation>(<window function>)

// tumbling processing-time windows
input
    .keyBy(<key selector>)
    .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
    .<windowed transformation>(<window function>)

// daily tumbling event-time windows offset by -8 hours.
input
    .keyBy(<key selector>)
    .window(TumblingEventTimeWindows.of(Time.days(1), Time.hours(-8)))
    .<windowed transformation>(<window function>)
```

Sliding Window

具有固定大小窗口，并且需要指定多久执行一次（如窗口大小为5分钟，每1分钟执行一次）

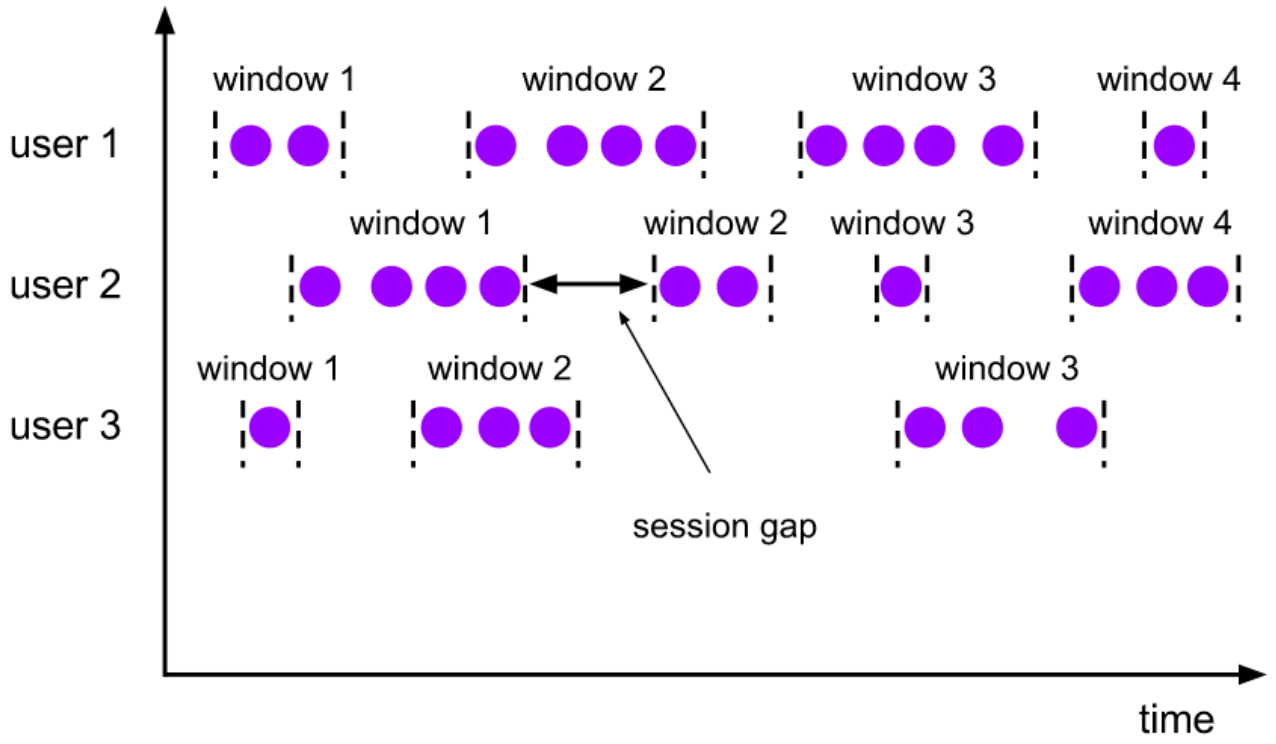



```
val input: DataStream[T] = ...

// sliding event-time windows
input
    .keyBy(<key selector>)
    .window(SlidingEventTimeWindows.of(Time.seconds(10), Time.seconds(5)))
    .<windowed transformation>(<window function>)

// sliding processing-time windows
input
    .keyBy(<key selector>)
    .window(SlidingProcessingTimeWindows.of(Time.seconds(10), Time.seconds(5)))
    .<windowed transformation>(<window function>)

// sliding processing-time windows offset by -8 hours
input
    .keyBy(<key selector>)
    .window(SlidingProcessingTimeWindows.of(Time.hours(12), Time.hours(1), Time.hours(-8)))
    .<windowed transformation>(<window function>)
```

Session Window

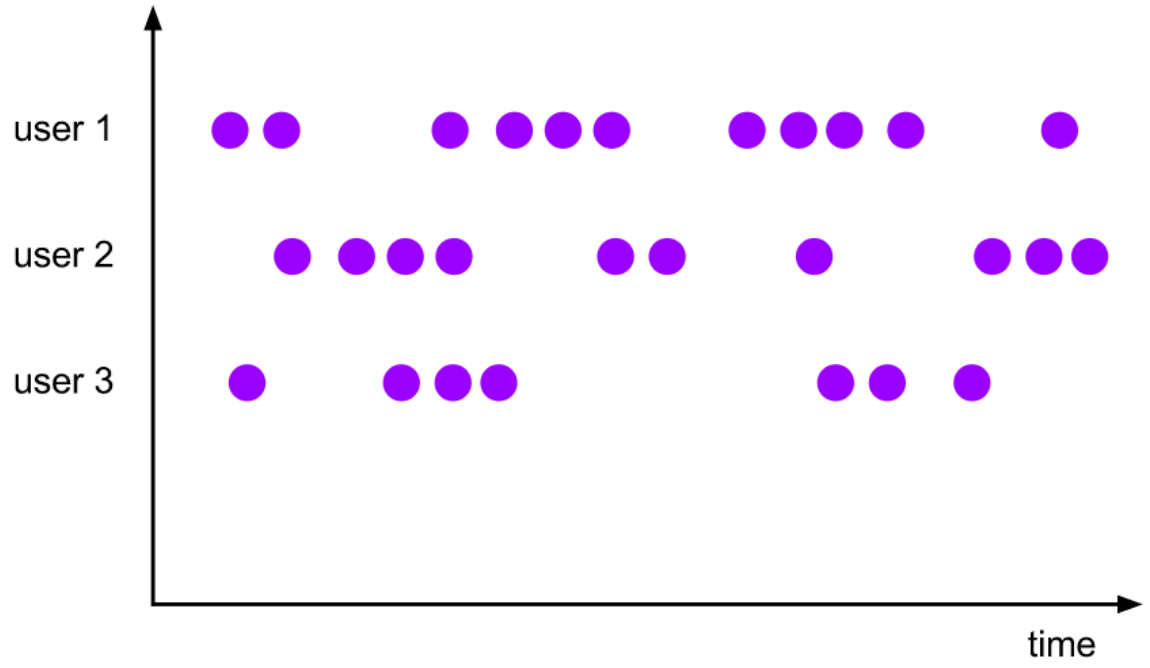
- 无固定的窗口大小
- 没有固定的开始时间和结束时间
- 窗口的关闭以是否能接收到某一个周期内是数据为准
- 超过一定的周期缝隙将开启 一个新的窗口

```
DataStream<T> input = ...;

// event-time session windows with static gap
input
    .keyBy(<key selector>)
    .window(EventTimeSessionWindows.withGap(Time.minutes(10)))
    .<windowed transformation>(<window function>);

// event-time session windows with dynamic gap
input
    .keyBy(<key selector>)
    .window(EventTimeSessionWindows.withDynamicGap((element) -> {
        // determine and return session gap
    }))
    .<windowed transformation>(<window function>);

// processing-time session windows with static gap
input
    .keyBy(<key selector>)
    .window(ProcessingTimeSessionWindows.withGap(Time.minutes(10)))
    .<windowed transformation>(<window function>);

// processing-time session windows with dynamic gap
input
    .keyBy(<key selector>)
    .window(ProcessingTimeSessionWindows.withDynamicGap((element) -> {
        // determine and return session gap
    }))
    .<windowed transformation>(<window function>);
```



Global Window

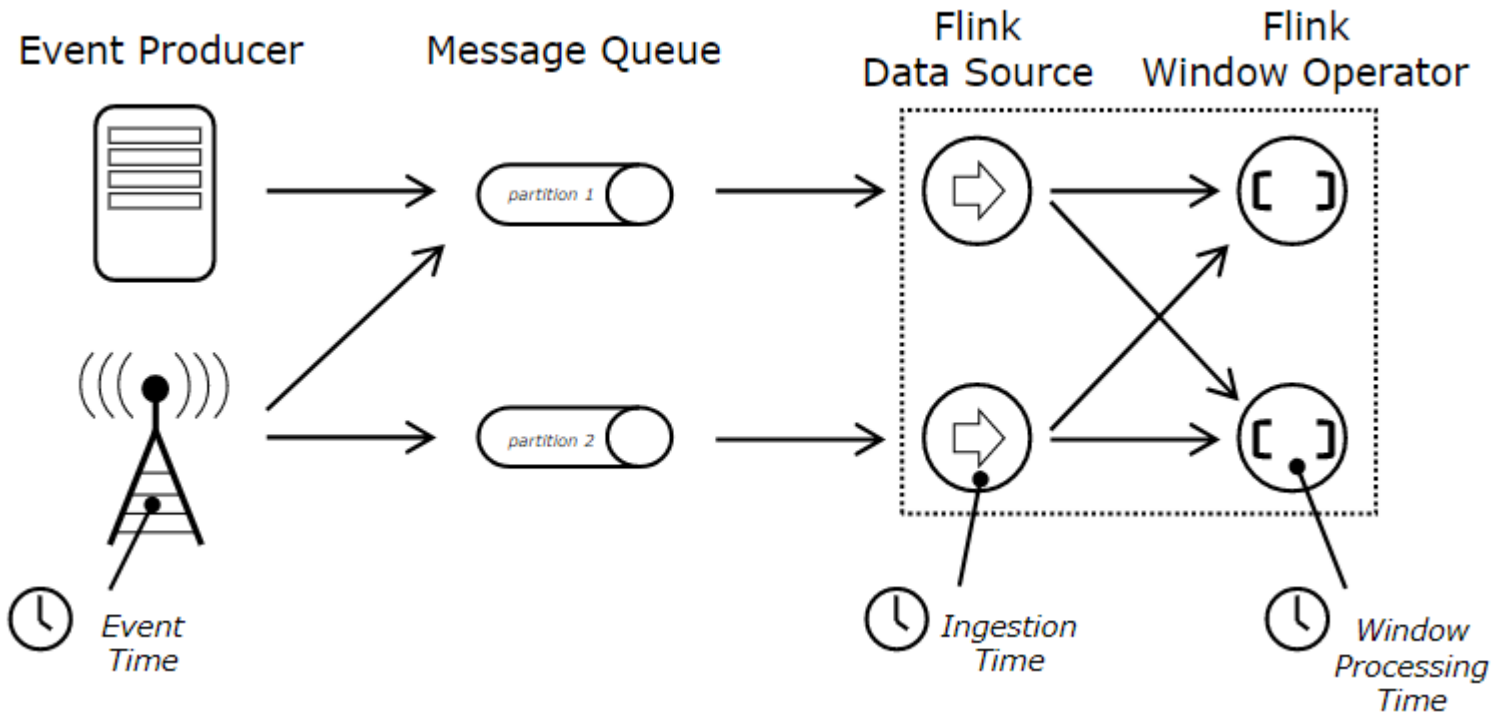
- 分配所有相同的key到相同的单个window
- 必须触发器




```
DataStream<T> input = ...;

input
    .keyBy(<key selector>)
    .window(GlobalWindows.create())
    .<windowed transformation>(<window function>);
```

Time Type

- Event Time
- Processing Time
- Ingestion Time



https://ci.apache.org/projects/flink/flink-docs-master/dev/event_time.html

State

- 什么是状态，状态托管
- Operator State
- Keyed State
- State Backend

https://ci.apache.org/projects/flink/flink-docs-master/dev/stream/state/

State Type

- Operator State
  - 绑定到一个operator上
  - 和Key无关
- 数据结构
  - ListState<T>
  - ValueState<T>
  - …

Manage State Example


```
class BufferSink(threshold:Int) extends SinkFunction[(String,Int)] with CheckpointedFunction{
  @transient
  val bufferElements :ListBuffer[(String, Int)] = ListBuffer[(String, Int)]()
  var checkpointedState:ListState[(String,Int)]=_
  override def invoke(value: (String, Int), context: SinkFunction.Context[_]): Unit = {
    bufferElements += value  //将数据写入内存

    if (bufferElements.size==threshold){
      for(element<-bufferElements){
        //send to sink
      }
      bufferElements.clear()
    }
  }
  override def snapshotState(functionSnapshotContext: FunctionSnapshotContext): Unit = {
    checkpointedState.clear()   //当快照发生时将内存中的数据写入到checkpoint中
    for(element<-bufferElements){
      checkpointedState.add(element)
    }
  }
  override def initializeState(context: FunctionInitializationContext): Unit = {
    val descriptor: ListStateDescriptor[(String, Int)] =new ListStateDescriptor[(String,Int)](
      "buffered-elements",
      TypeInformation.of(new TypeHint[(String,Int)]() {})
    )
    checkpointedState=context.getOperatorStateStore.getListState(descriptor)
    if(context.isRestored){//从checkpoint中恢复状态并写入内存
      for(element<-checkpointedState.get()){
        bufferElements += element
      }
    }
  }
}
```

State Type

- Keyed State
  - 基于key上的一种特殊的Operator State
  - 一般需要通过KeyBy操作来实现
    dataStream.keyBy()

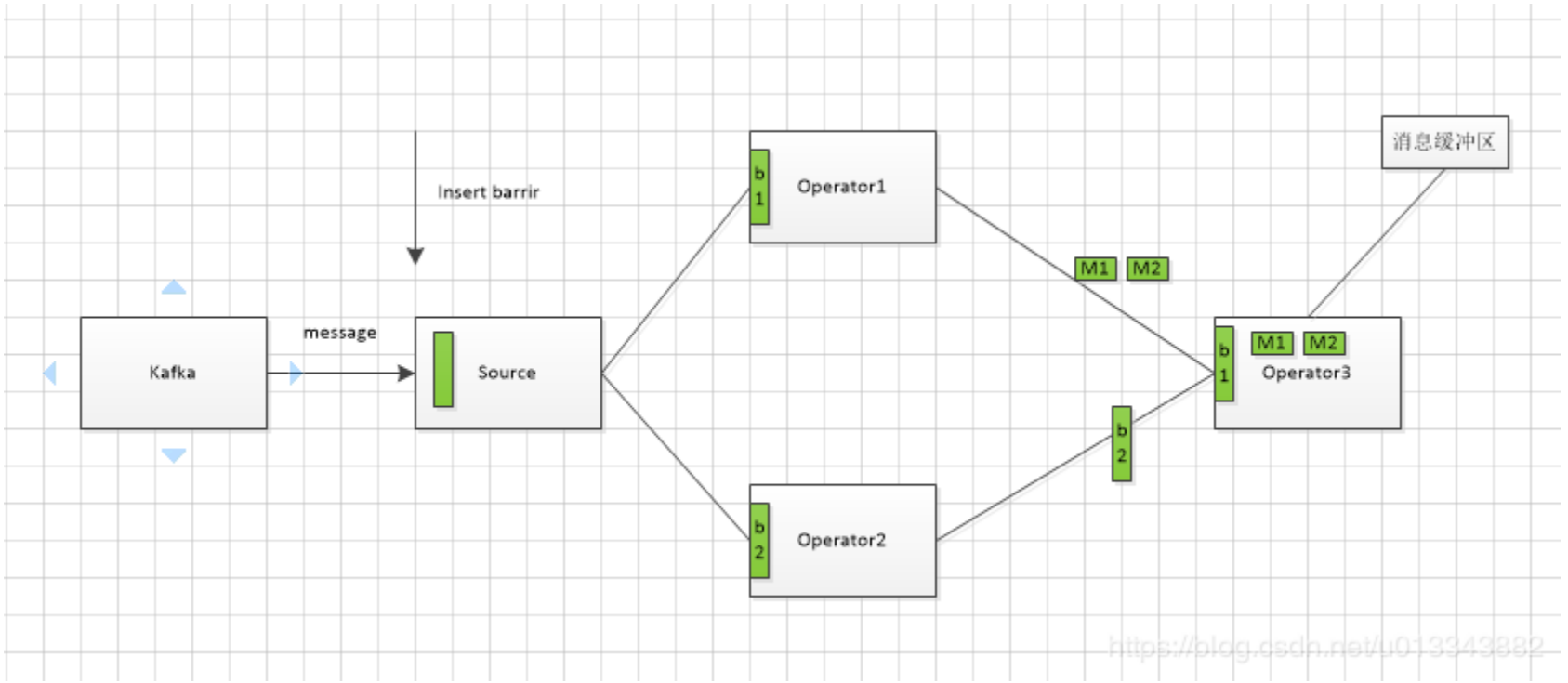
checkpoint

- 为了保证state的容错性，Flink需要对state进行checkpoint
- Checkpoint是Flink实现容错机制最核心的功能，它能够根据配置周期性地基于Stream中各个Operator/task的状态来生成快照，从而将这些状态数据定期持久化存储下来，当Flink程序一旦意外崩溃时，重新运行程序时可以有选择地从这些快照进行恢复，从而修正因为故障带来的程序数据异常
- Flink的checkpoint机制可以与(stream和state)的持久化存储交互的前提
  - 持久化的source，它需要支持在一定时间内重放事件。这种sources的典型例子是持久化的消息队列（比如Apache Kafka，RabbitMQ等）或文件系统（比如HDFS，S3，GFS等
  - 用于state的持久化存储，例如分布式文件系统（比如HDFS，S3，GFS等）

https://ci.apache.org/projects/flink/flink-docs-master/dev/stream/state/checkpointing.html

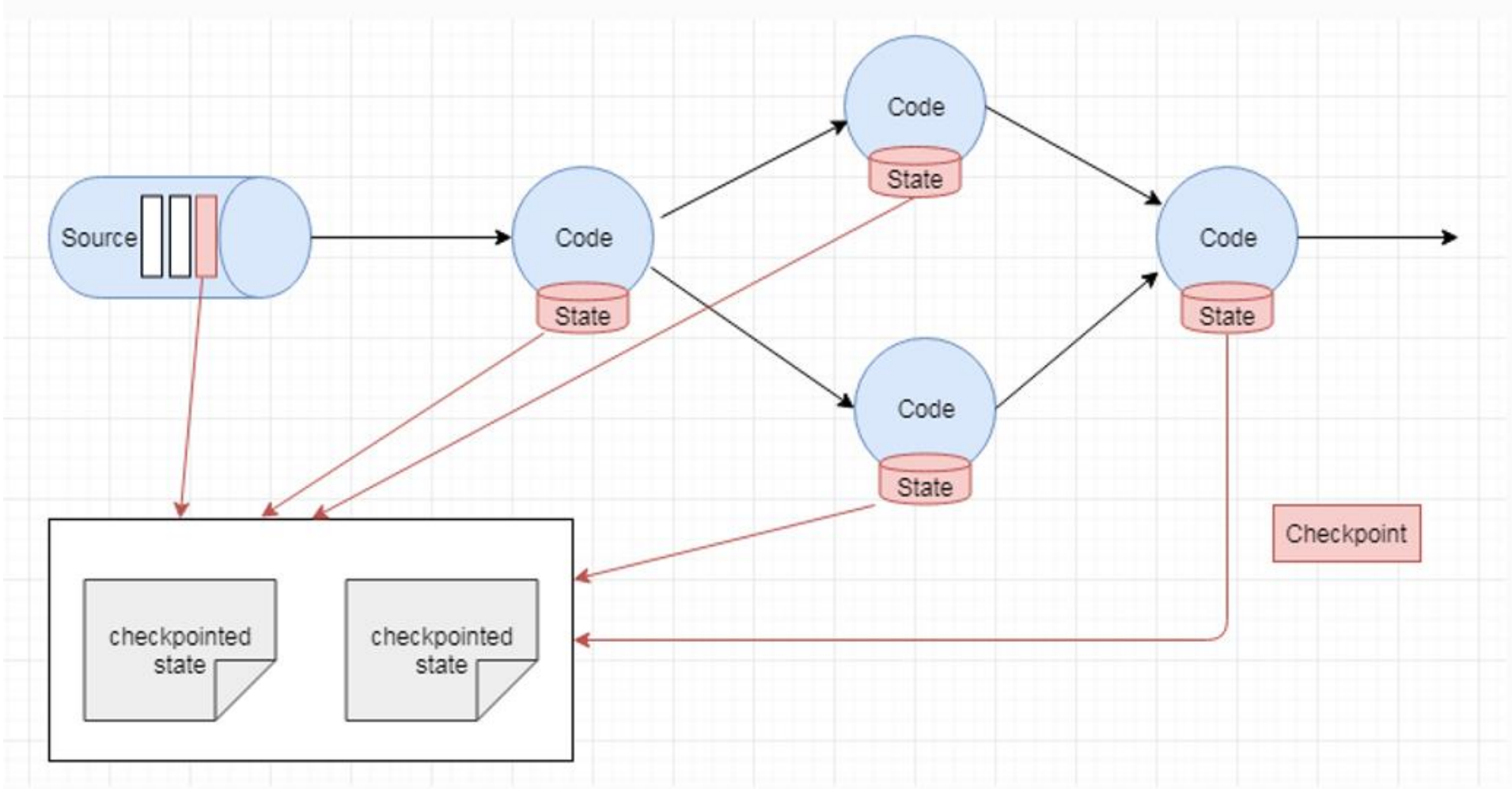
容错

- 依靠CheckPoint机制
- 保证exactly-once（只能保证Flink系统内部）



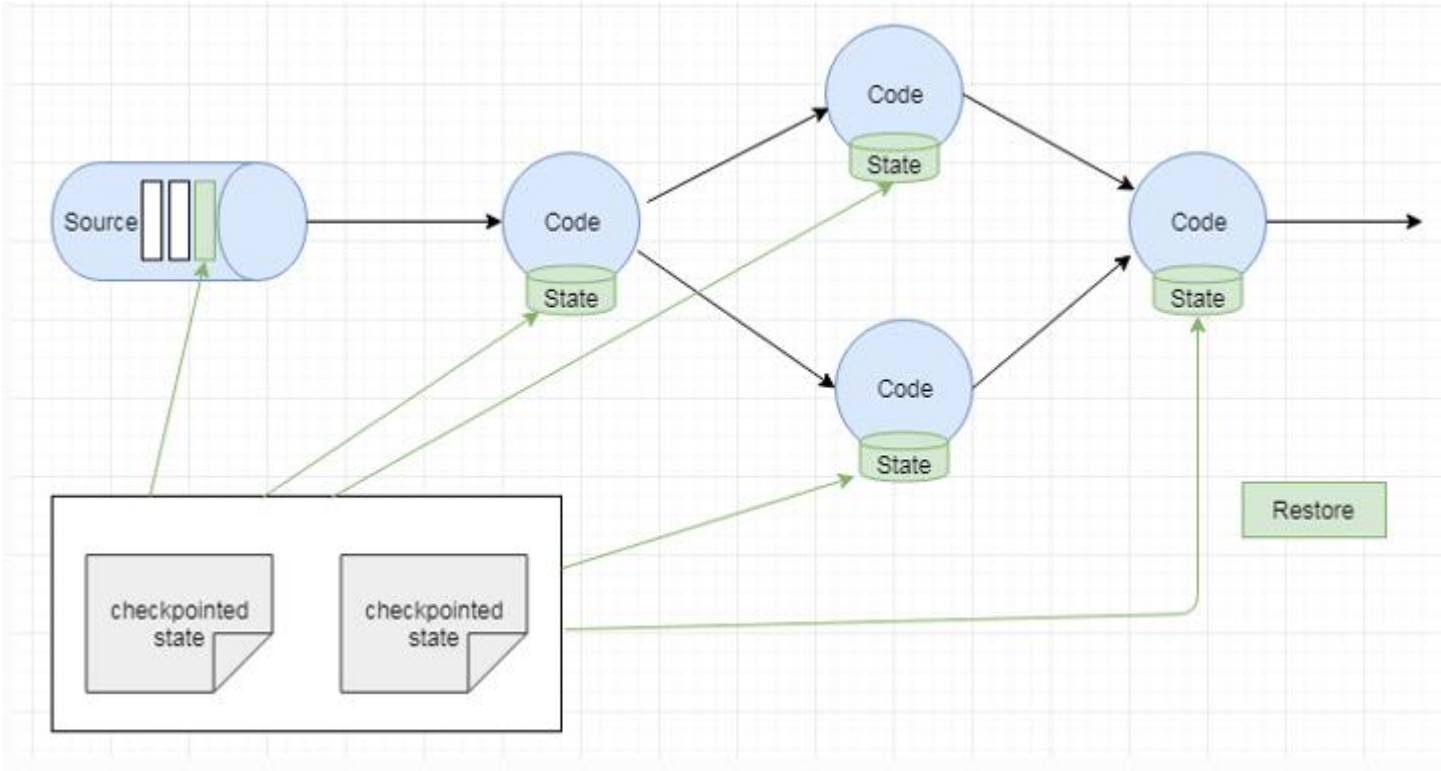
https://blog.csdn.net/u013343882/article/details/98965640

容错


Source
Code
State
Code
State
Code
State
Code
State
checkpointed state
checkpointed state
Checkpoint

容错

- 恢复所有的状态
- 设置Source的位置（如 Kafka的offset）



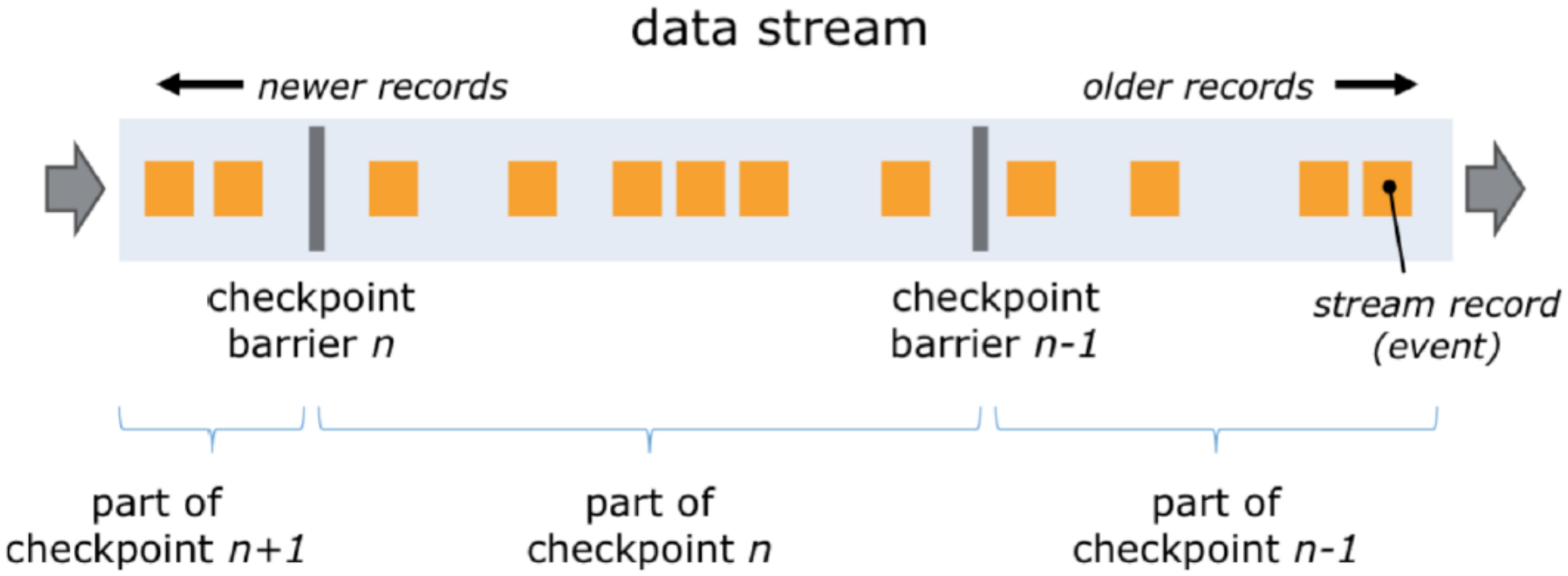
SavePoint

- Flink通过Savepoint功能可以做到程序升级后，继续从升级前的那个点开始执行计算，保证数据不中断
- 全局，一致性快照。可以保存数据源offset，operator操作状态等信息
- 可以从应用在过去任意做了savepoint的时刻开始继续消费

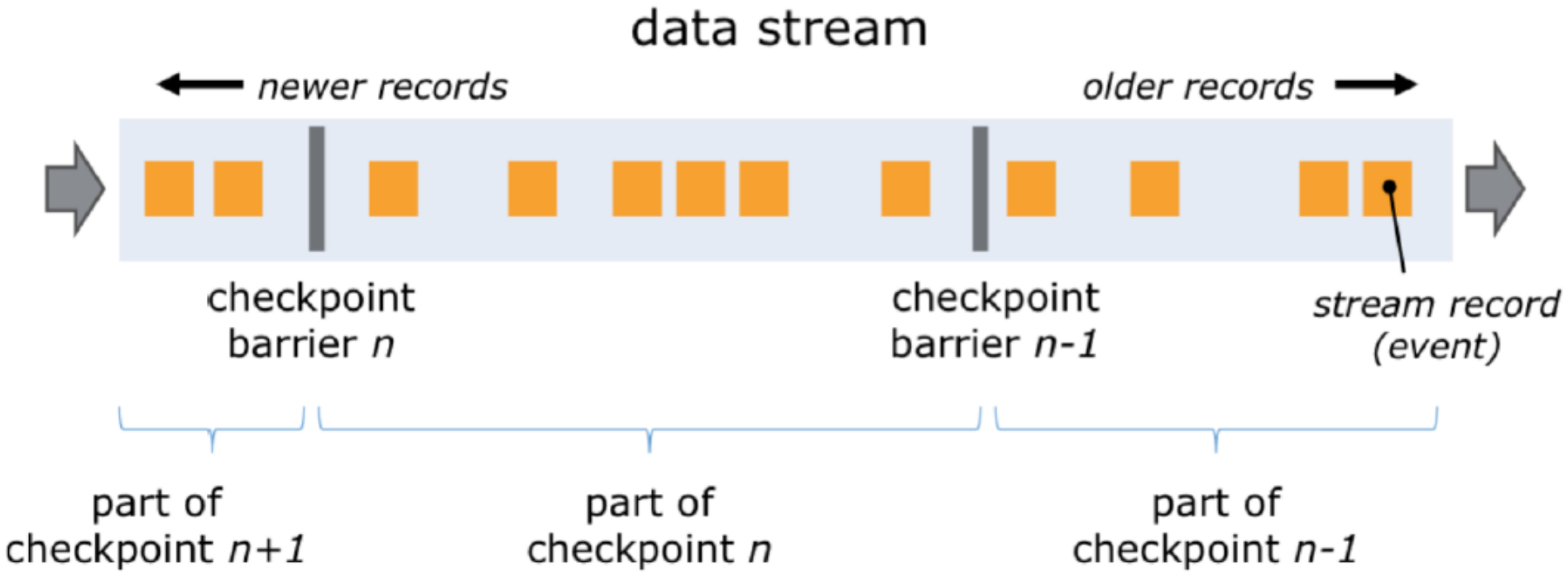
Checkpoint vs SavePoint

- Checkpoint
  - 应用定时触发，用于保存状态，会过期
  - 内部应用失败重启时使用
- Savepoint
  - 用户手动执行，不会过期
  - 在升级情况下使用

Checkpoint barrir



Checkpoint barrir

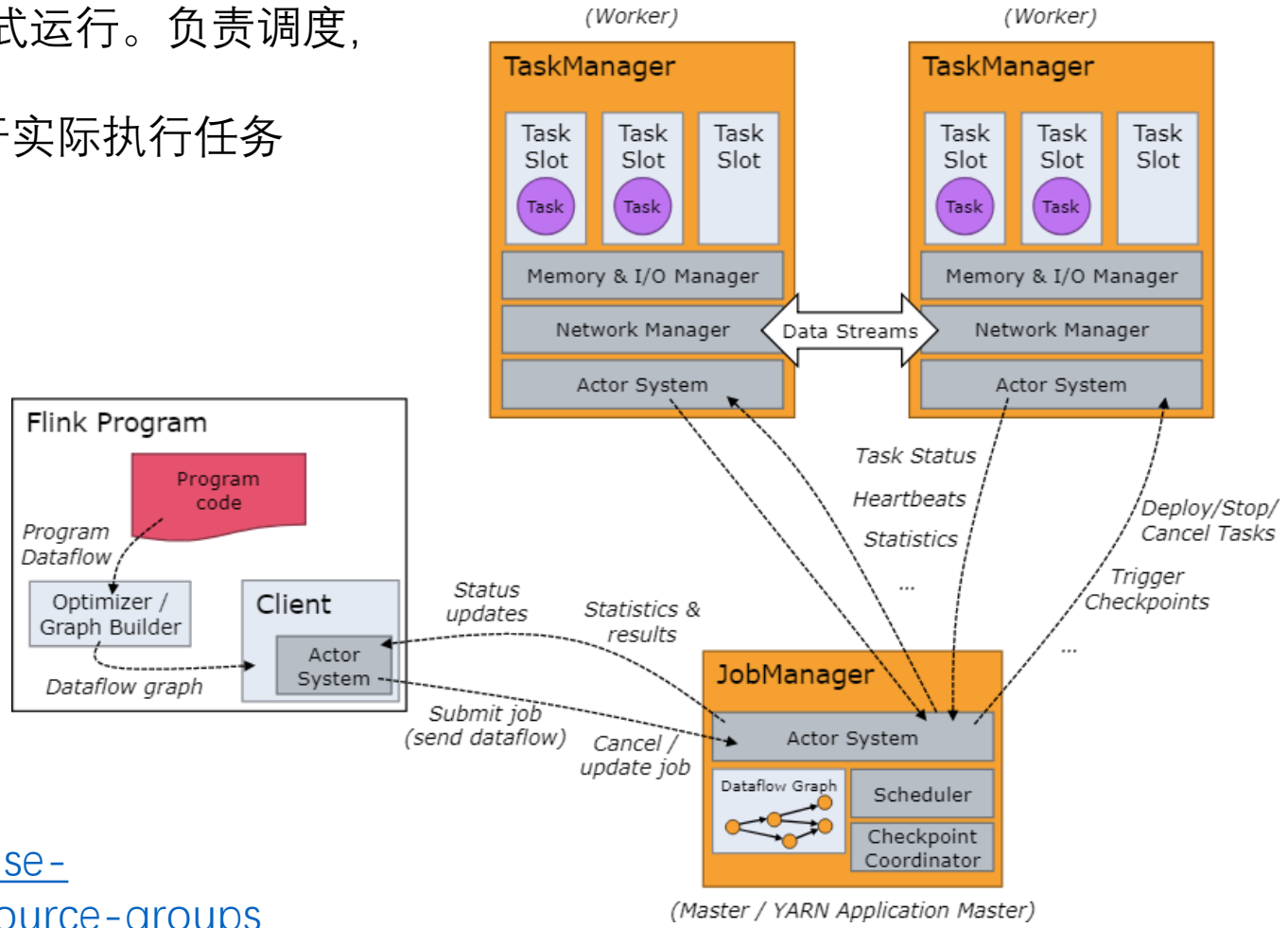


索信达
Suoxinda Holdings Limited

Flink运行时

Flink运行时架构

- Client:用于提交作业
- JobManager:充当Master角色，主要协调分布式运行。负责调度，协调checkpoint，失败恢复等
- TaskManager：有时候也称作Worker,主要用于实际执行任务

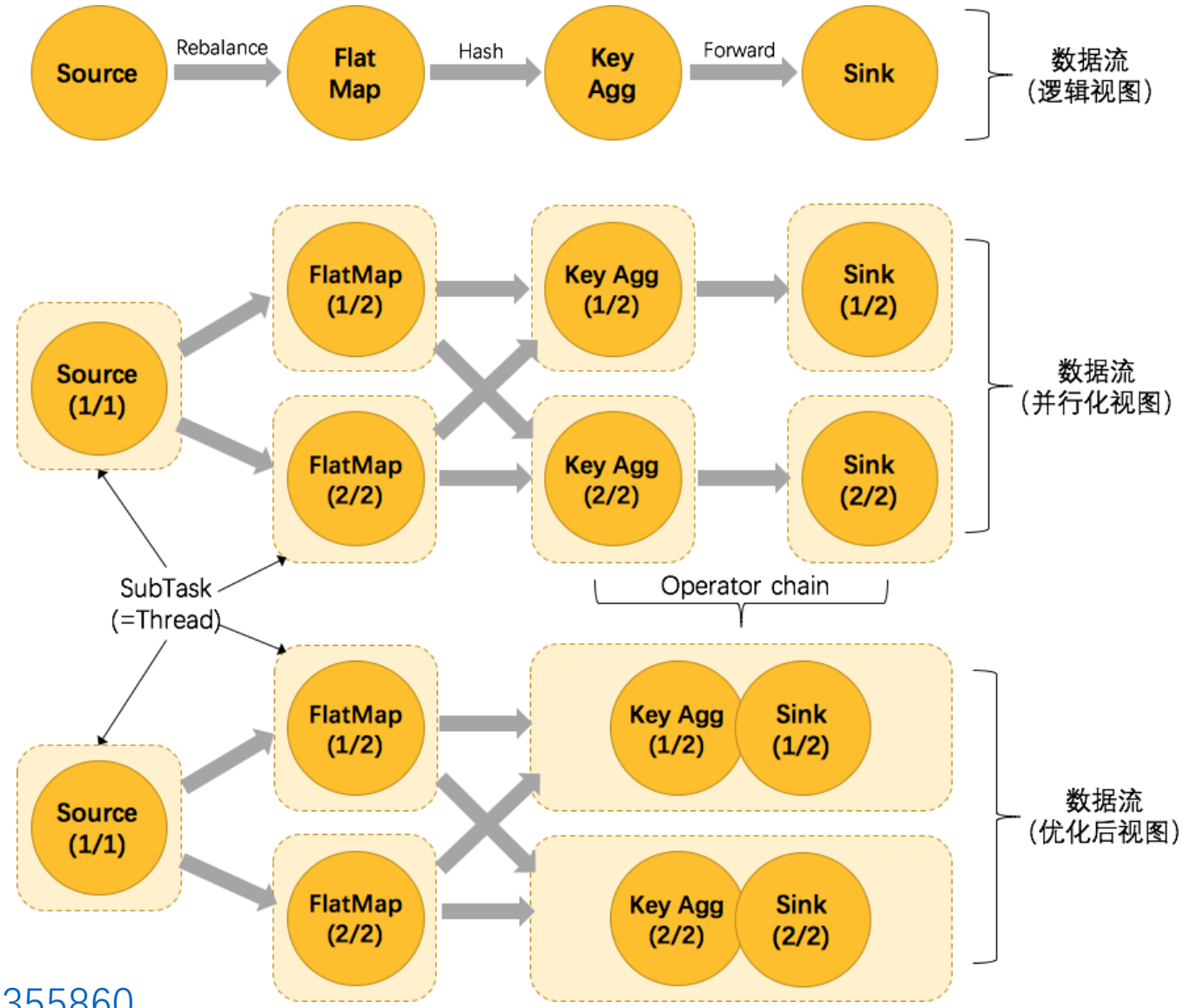


https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/stream/operators/#task-chaining-and-resource-groups

https://blog.csdn.net/u013343882/article/details/82292306

Operator Chains

- 为了更高的分布式执行，Flink会将多个operator一个task在一起执行
- 将operators连接成task有以下好处
  - 减少线程切换
  - 减少序列化/反序列化
  - 减少数据在缓冲区中交换
  - 减少延迟，提高吞吐量

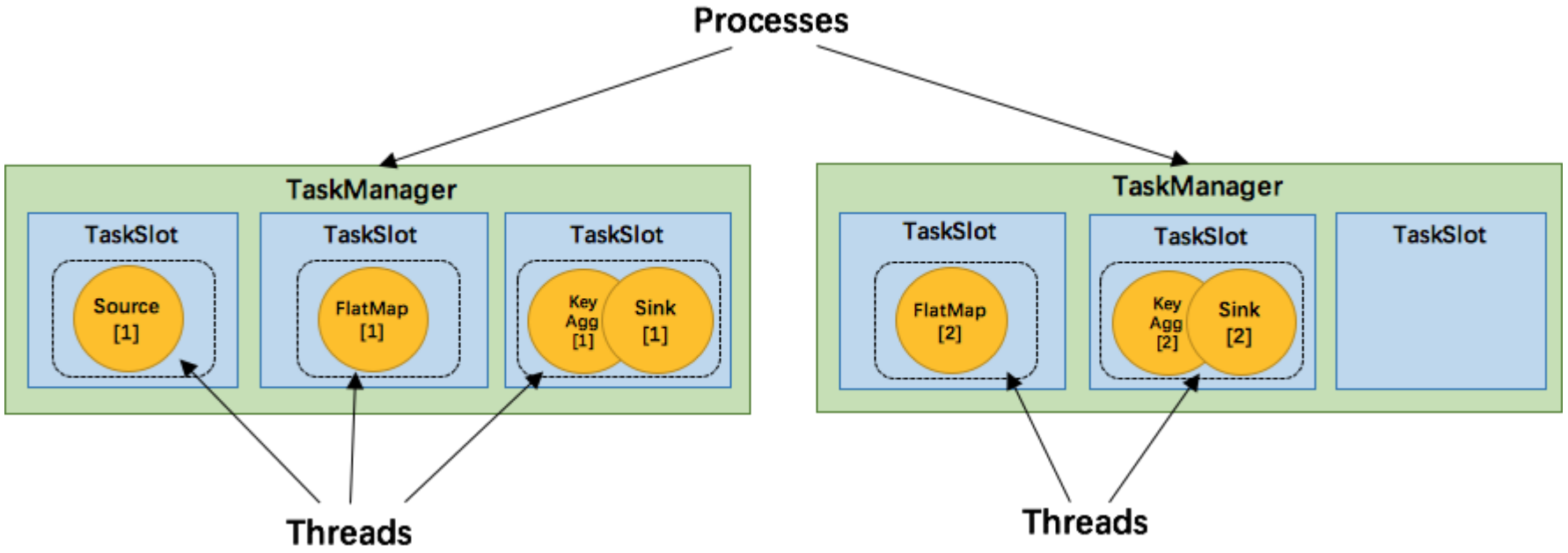


https://blog.csdn.net/u013343882/article/details/82355860

Operator Chains 条件

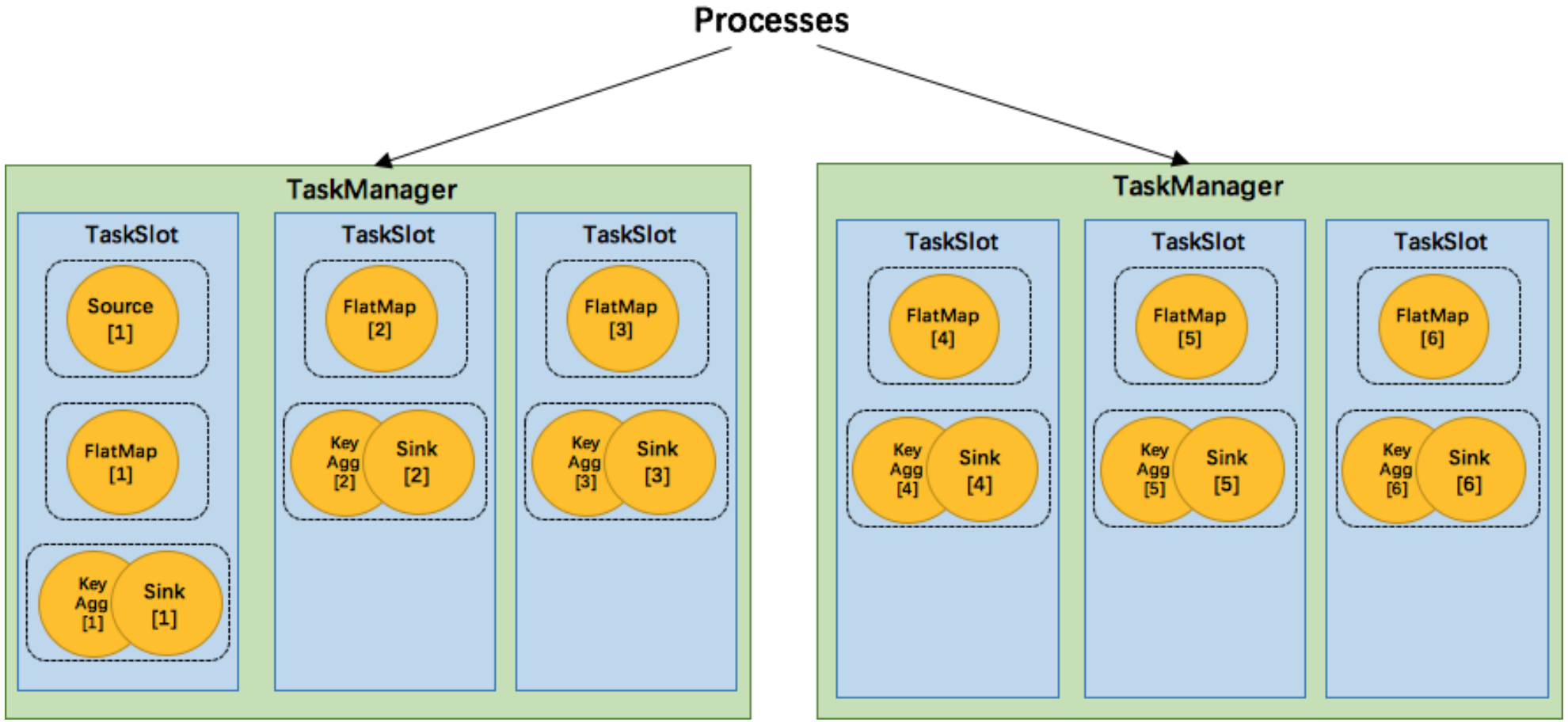
- 上下游并行度一致
- 下游节点入度为1（没有其他节点的输入）
- 上下游节点都在同一个slot group中
- 下游节点的chain策略为ALWAYS(默认)
- 上游节点的chain策略为ALWAYS或HEAD(source默认为Head)
- 两个节点的数据分区方式是forward
- 用户没有禁用chain

TaskManager Slot



SlotSharingGroup & CoLocationGroup

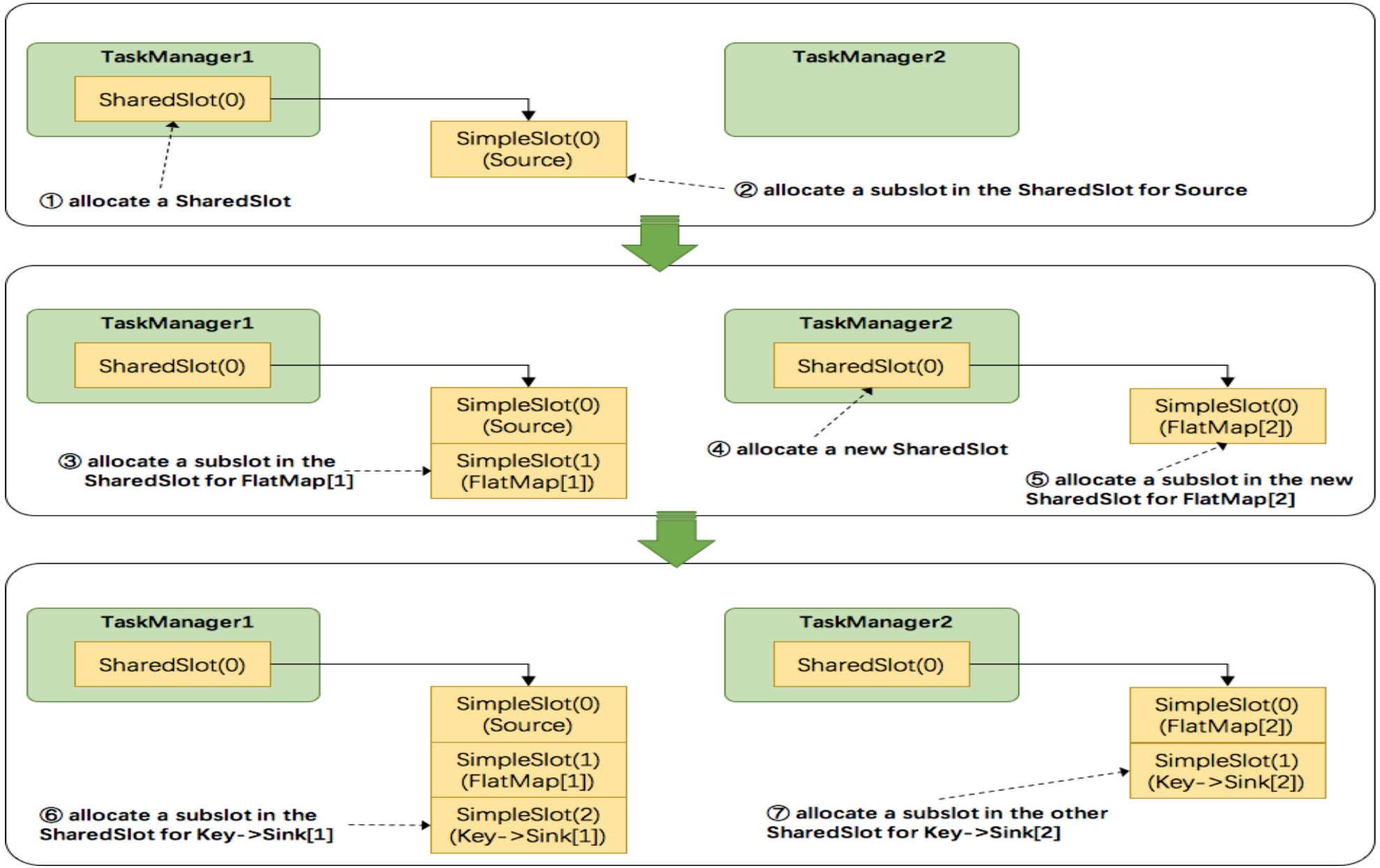
- 允许subtask共享slot （必须是同一个Job不同task的subtask）
  - Task slots和job中最高的并行度一致
  - 资源充分利用



Task共享Slot过程

为简化问题我们假设下面的算子运行在一个具有2个TaskManager并且每个TaskManageer下仅有一个slot

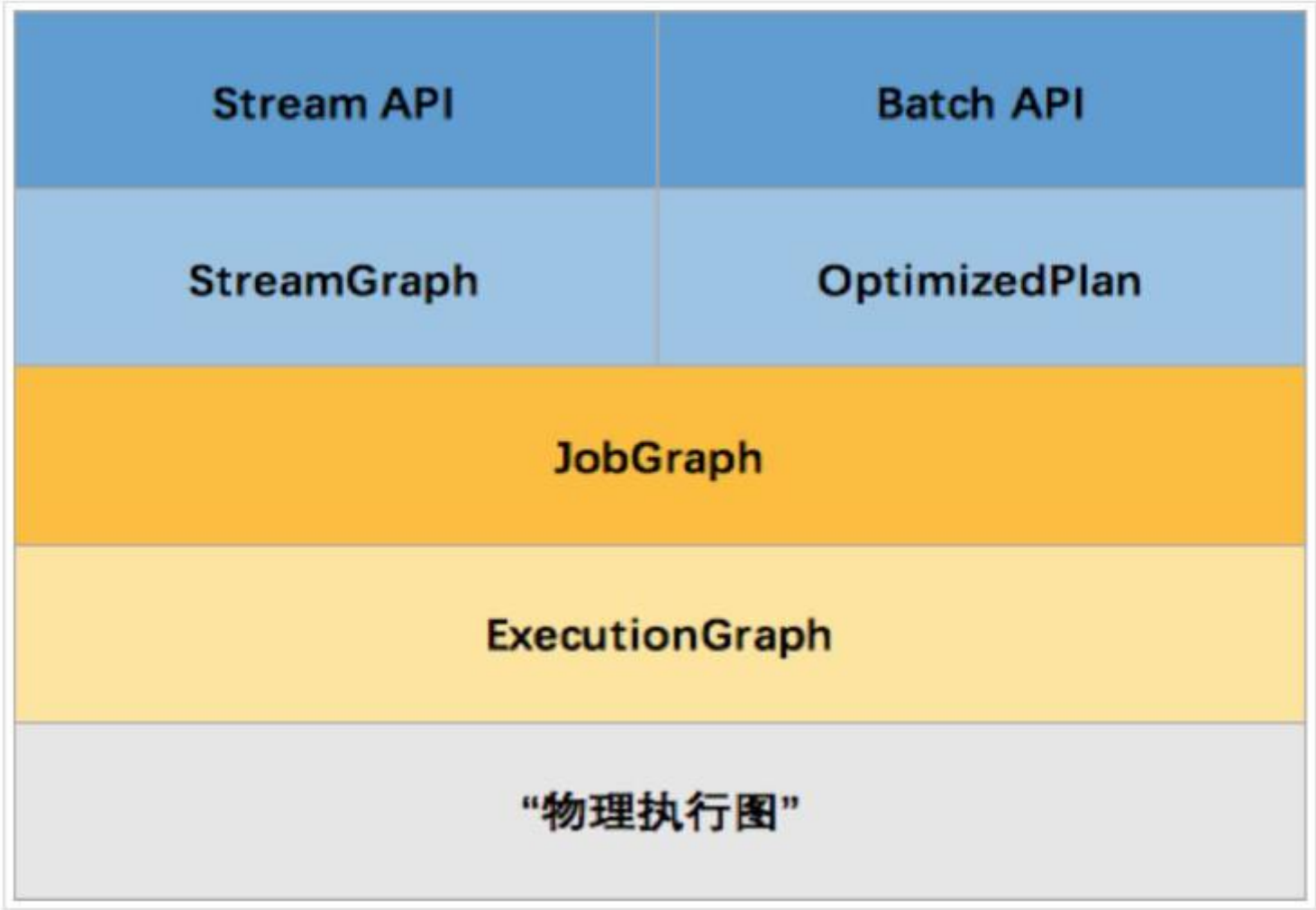
Task共享Slot过程



Flink的几张图

- StreamGraph
- JobGraph
- ExecuteGraph
- 物理执行图

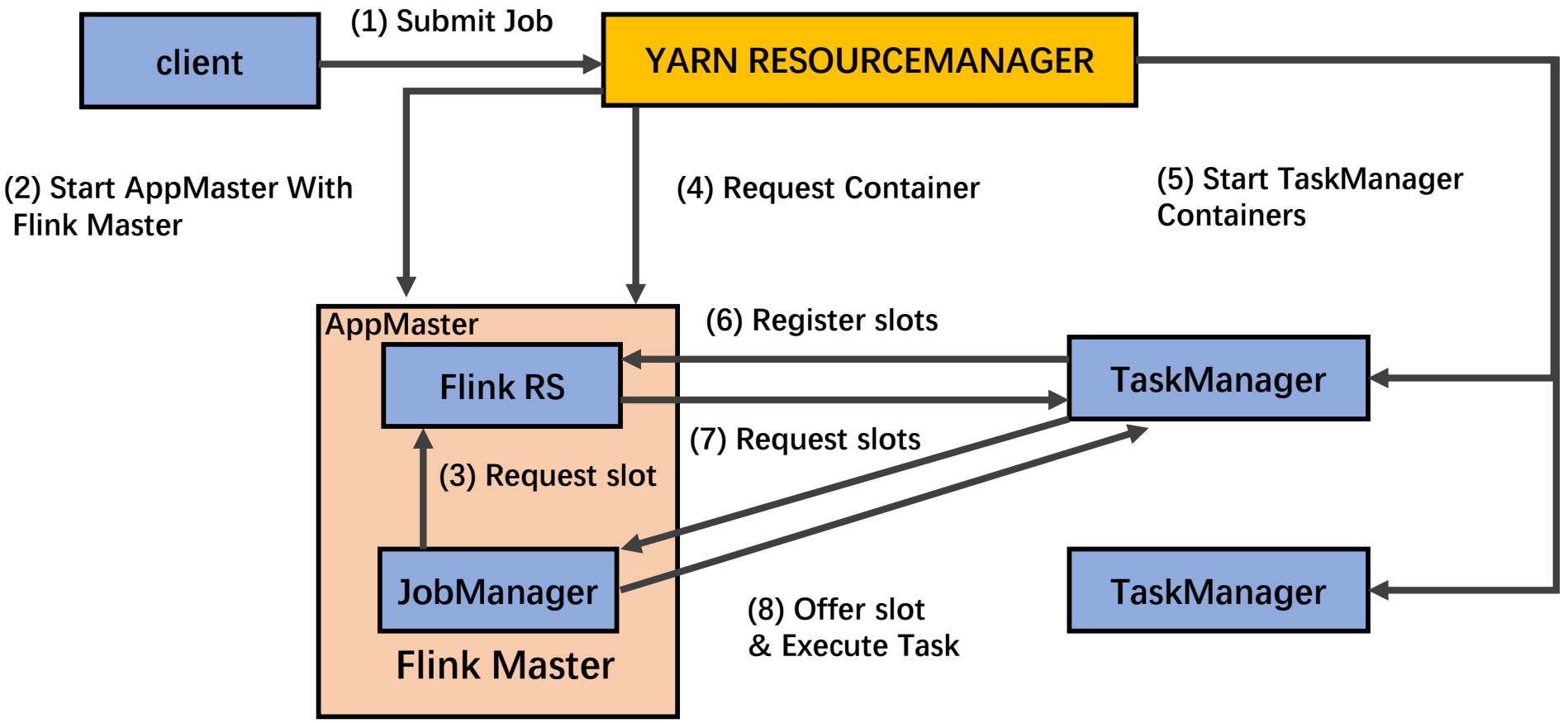
https://blog.csdn.net/u013343882/article/details/82292306



Flink on Yarn

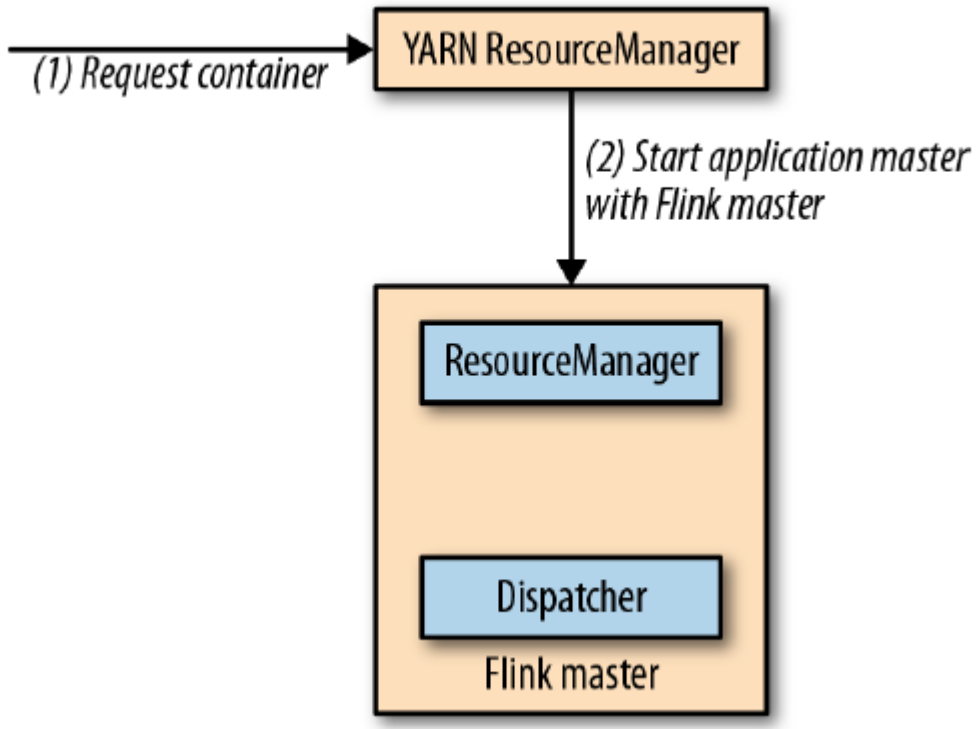
- Flink on Yarn 提交的两种方式
  - Job mode
  - Session mode

Job Mode

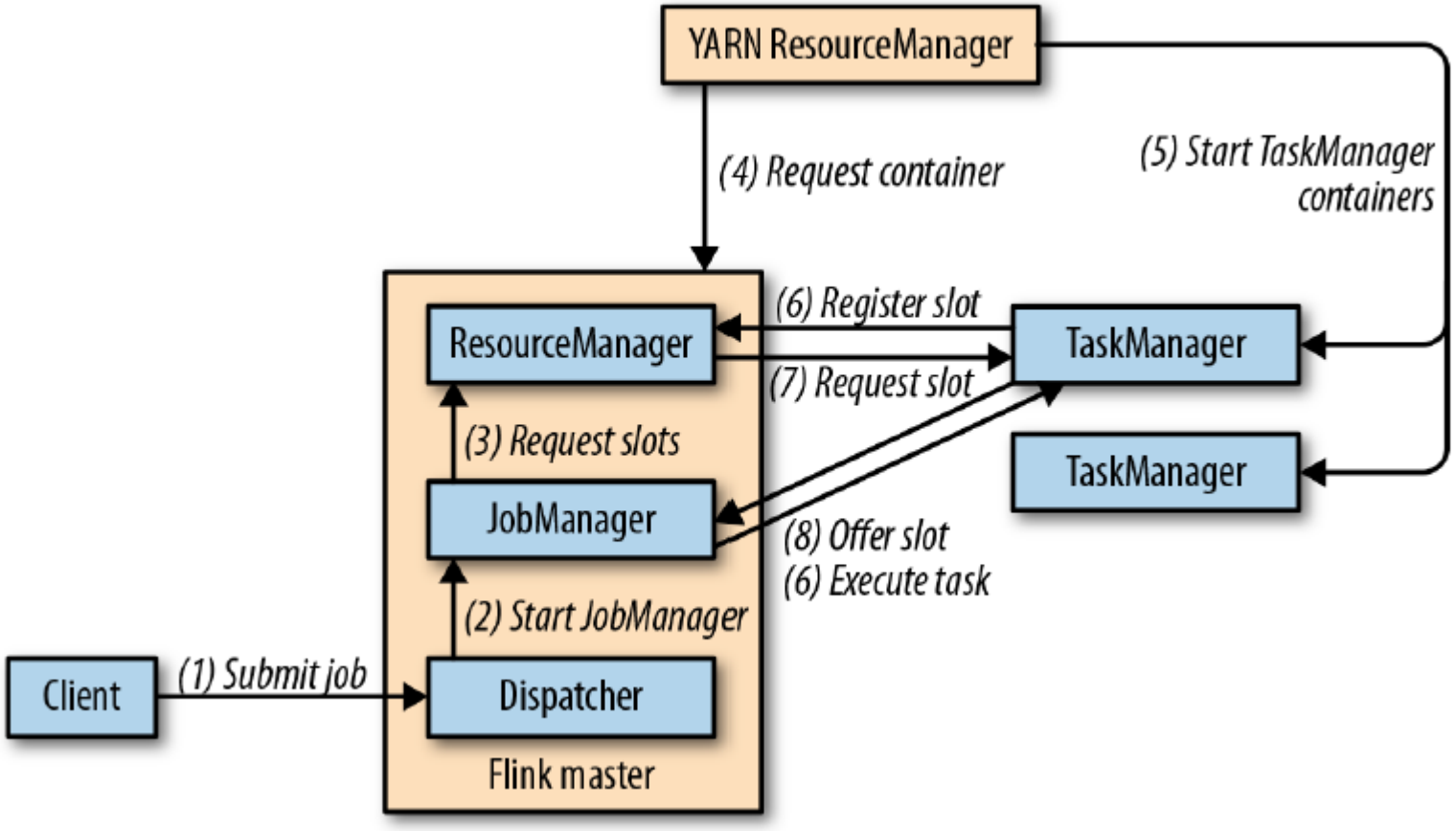

client
(1) Submit Job
YARN RESOURCEMANAGER
(2) Start AppMaster With
Flink Master
(4) Request Container
(5) Start TaskManager
Containers
AppMaster
(6) Register slots
Flink RS
TaskManager
(7) Request slots
(3) Request slot
JobManager
TaskManager
(8) Offer slot
& Execute Task
Flink Master

Session Mode

- Session mode 启动一个长时间运行的Flink Cluster
- 能够运行多个Flink Job



Session Mode



3 Flink Example

Flink 案例代码

https://github.com/314649558/learing/tree/master/flink_stream_demo/poc_parent

索信达
Suoxinda Holdings Limited

科技常在

SCIENCE
EVERYWHERE

THANK
YOU