

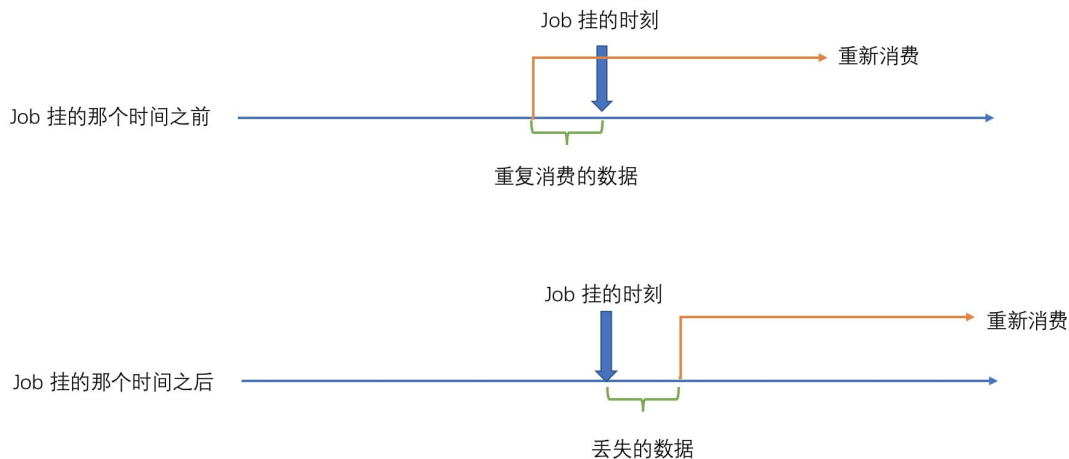
前言

在基础篇中的 [2一文让你彻底了解大数据实时计算框架 Flink](#) 一文中介绍了 Flink 是一款有状态的流处理框架。那么大家可能有点疑问，这个状态是什么意思？拿 [6通过 WordCount 程序教你快速入门上手 Flink](#) 文章中的案例来做解释吧，在 Flink 最简单的 Word Count 程序中，需要不断的对 word 出现的个数进行结果统计，那么后一个结果就需要利用前一个的结果然后再做 +1 的操作，这样前一个计算就需要将 word 出现的次数 count 进行存着（这个 count 那么就是一个状态）然后后面才可以进行累加。

为什么需要 state?

对于流处理系统，数据是一条一条被处理的，如果没有对数据处理的进度进行记录，那么如果这个处理数据的 Job 因为机器问题或者其他问题而导致重启，那么它是不知道上一次处理数据是到哪个地方了，这样的情况下如果是批数据，倒是可以很好的解决（重新将这份固定的数据再执行一遍），但是流数据那就麻烦了，你根本不知道什么在 Job 挂的那个时刻数据消费到哪里了？那么你重启的话该从哪里开始重新消费呢？你可以有以下选择（因为你可能也不确定 Job 挂的具体时间）：

- Job 挂的那个时间之前：如果是从 Job 挂之前开始重新消费的话，那么会导致部分数据（从新消费的时间点到之前 Job 挂的那个时间点之前的数据）重复消费
- Job 挂的那个时间之后：如果是从 Job 挂之后开始消费的话，那么会导致部分数据（从 Job 挂的那个时间点到新消费的时间点产生的数据）丢失，没有消费



zhisheng

为了解决上面两种情况（数据重复消费或者数据没有消费）的发生，那么是不是就得需要个什么东西做个记录将这种数据消费状态，Flink state 就这样诞生了，state 中存储着每条数据消费后数据的消费点（生产环境需要持久化这些状态），当 Job 因为某种错误或者其他原因导致重启时，就能够从 checkpoint（定时将 state 做一个全局快照，在 Flink 中，为了能够让 Job 在运行的过程中保证容错性，才会对这些 state 做一个快照，在下一篇文章 [2Flink Checkpoint 和 Savepoint 区别及其如何配置使用?](#) 会详细讲）中的 state 数据进行恢复。

State 的种类

在 Flink 中有两个基本的 state：Keyed state 和 Operator state，下面来分别介绍一下这两种 State：

Keyed State

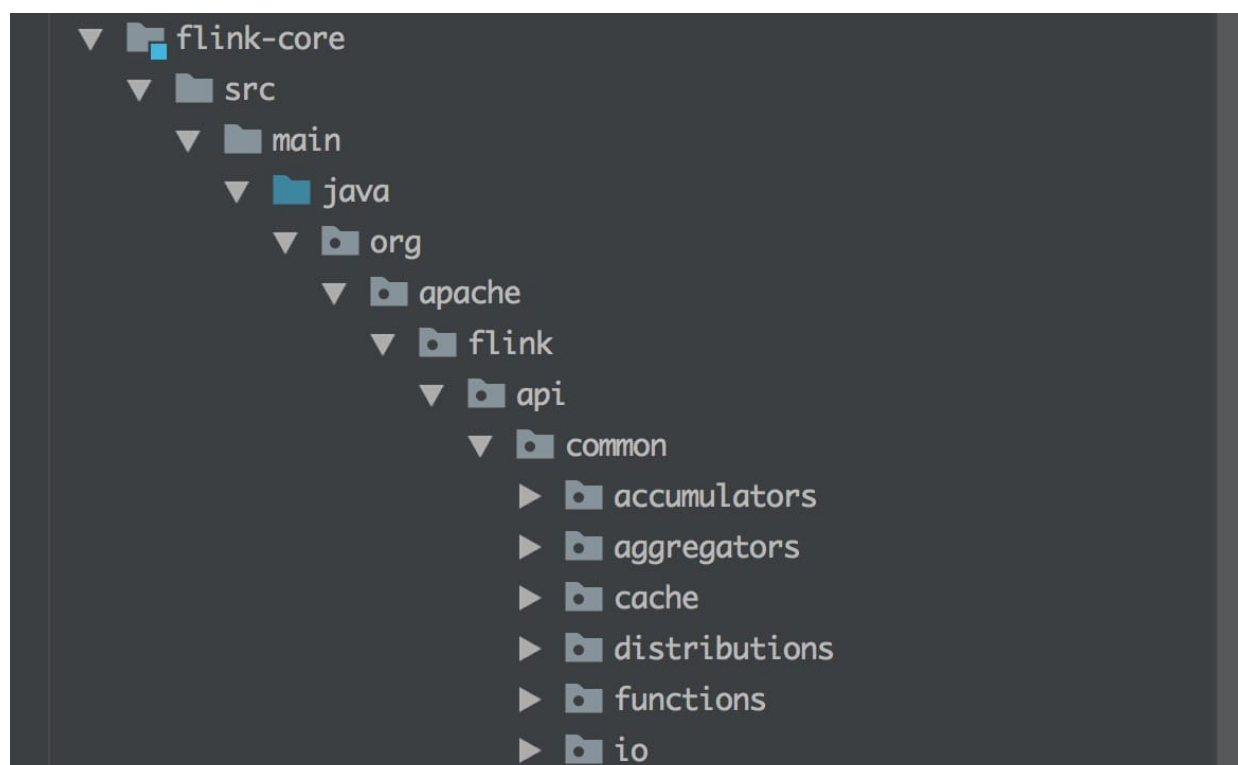
Keyed State 总是和具体的 key 相关联，也只能在 KeyedStream 的 function 和 operator 上使用。你可以将 Keyed State 当作是 Operator State 的一种特例，但是它是被分区或分片的。每个 Keyed State 分区对应一个 key 的 Operator State，对于某个 key 在某个分区上有唯一的状态。逻辑上，Keyed State 总是对应着一个 `<parallel-operator-instance, key>` 二元组，在某种程度上，因为每个具体的 key 总是属于唯一一个具体的 parallel-operator-instance（并行操作实例），这种情况下，那么就可以简化认为是 `<operator, key>`。Keyed State 可以进一步组织成 Key Group，Key Group 是 Flink 重新分配 Keyed State 的最小单元，所以有多少个并行，就会有多少个 Key Group。在执行过程中，每个 keyed operator 的并行实例会处理来自不同 key 的不同 Key Group。

Operator State

对 Operator State 而言，每个 operator state 都对应着一个并行实例。Kafka Connector 就是一个很好的例子。每个 Kafka consumer 的并行实例都会持有一份 topic partition 和 offset 的 map，这个 map 就是它的 Operator State。

当并行度发生变化时，Operator State 可以将状态在所有的并行实例中进行重分配，并且提供了多种方式进行重分配。

在 Flink 源码中，在 flink-core module 下的 `org.apache.flink.api.common.state` 中可以看到 Flink 中所有和 State 相关的类。



```
▶ operators
▶ resources
▶ restartstrategy
▶ serialization
▼ state
  I AggregatingState
  C AggregatingStateDescriptor
  I AppendingState
  I BroadcastState
  I FoldingState
  C FoldingStateDescriptor
  I KeyedStateStore
  I ListState
  C ListStateDescriptor
  I MapState
  C MapStateDescriptor
  I MergingState
  I OperatorStateStore
  I ReadOnlyBroadcastState
  I ReducingState
  C ReducingStateDescriptor
  I State
  C StateDescriptor
  C StateTtlConfig
  I ValueState
  C ValueStateDescriptor
```

zhisheng

Raw and Managed State

Keyed State 和 Operator State 都有两种存在形式，即 Raw State（原始状态）和 Managed State（托管状态）。

原始状态是 Operator（算子）保存它们自己的数据结构中的 state，当 checkpoint 时，原始状态会以字节流的形式写入进 checkpoint 中。Flink 并不知道 State 的数据结构长啥样，仅能看到原生的字节数组。

托管状态可以使用 Flink runtime 提供的数据结构来表示，例如内部哈希表或者 RocksDB。具体有 ValueState，ListState 等。Flink runtime 会对这些状态进行编码然后将它们写入到 checkpoint 中。

DataStream 的所有 function 都可以使用托管状态，但是原生状态只能在实现 operator 的时候使用。相对于原生状态，推荐使用托管状态，因为如果使用托管状态，当并行度发生改变时，Flink 可以自动的帮你重分配 state，同时还可以更好的管理内存。

注意：如果你的托管状态需要特殊的序列化，目前 Flink 还不支持。

使用托管 Keyed State

托管的 Keyed State 接口提供对不同类型状态（这些状态的范围都是当前输入元素的 key）的访问，这意味着这种状态只能在通过 stream.keyBy() 创建的 KeyedStream 上使用。

我们首先来看一下有哪些可以使用的状态，然后再来看看它们在程序中是如何使用的：

- ValueState: 保存一个可以更新和获取的值（每个 Key 一个 value），可以用 update(T) 来更新 value，可以用 value() 来获取 value。
- ListState: 保存一个值的列表，用 add(T) 或者 addAll(List) 来添加，用 Iterable get() 来获取。
- ReducingState: 保存一个值，这个值是状态的很多值的聚合结果，接口和 ListState 类似，但是可以用相应的 ReduceFunction 来聚合。
- AggregatingState<IN, OUT>: 保存很多值的聚合结果的单一值，与 ReducingState 相比，不同点在于聚合类型可以和元素类型不同，提供 AggregateFunction 来实现聚合。
- FoldingState<T, ACC>: 与 AggregatingState 类似，除了使用 FoldFunction 进行聚合。
- MapState<UK, UV>: 保存一组映射，可以将 kv 放进这个状态，使用 put(UK, UV) 或者 putAll(Map<UK, UV>) 添加，或者使用 get(UK) 获取。

所有类型的状态都有一个 clear() 方法来清除当前的状态。

注意：FoldingState 已经不推荐使用，可以用 AggregatingState 来代替。

需要注意，上面的这些状态对象仅用来和状态打交道，状态不一定保存在内存中，也可以存储在磁盘或者其他地方。另外，你获取到的状态的值是取决于输入元素的 key，因此如果 key 不同，那么在一次调用用户函数中获得的值可能与另一次调用的值不同。

要使用一个状态对象，需要先创建一个 StateDescriptor，它包含了状态的名字（你可以创建若干个 state，但是它们必须要有唯一的值以便能够引用它们），状态的值的类型，或许还有一个用户定义的函数，比如 ReduceFunction。根据你想要使用的 state 类型，你可以创建 ValueStateDescriptor、ListStateDescriptor、ReducingStateDescriptor、FoldingStateDescriptor 或者 MapStateDescriptor。

状态只能通过 RuntimeContext 来获取，所以只能在 RichFunction 里面使用。RichFunction 中你可以通过 RuntimeContext 用下述方法获取状态：

- ValueState getState(ValueStateDescriptor)
- ReducingState getReducingState(ReducingStateDescriptor)
- ListState getListState(ListStateDescriptor)

- `AggregatingState<IN, OUT> getAggregatingState(AggregatingState<IN, OUT>)`
- `FoldingState<T, ACC> getFoldingState(FoldingStateDescriptor<T, ACC>)`
- `MapState<UK, UV> getMapState(MapStateDescriptor<UK, UV>)`

上面讲了这么多概念，那么来一个例子来看看如何使用状态：

```

1  public class CountWindowAverage extends RichFlatMapFunction<Tuple2<Long,
2  Long>, Tuple2<Long, Long>> {
3
4      //ValueState 使用方式，第一个字段是 count，第二个字段是运行的和
5      private transient ValueState<Tuple2<Long, Long>> sum;
6
7      @Override
8      public void flatMap(Tuple2<Long, Long> input, Collector<Tuple2<Long,
9  Long>> out) throws Exception {
10
11          //访问状态的 value 值
12          Tuple2<Long, Long> currentSum = sum.value();
13
14          //更新 count
15          currentSum.f0 += 1;
16
17          //更新 sum
18          currentSum.f1 += input.f1;
19
20          //更新状态
21          sum.update(currentSum);
22
23          //如果 count 等于 2，发出平均值并清除状态
24          if (currentSum.f0 >= 2) {
25              out.collect(new Tuple2<>(input.f0, currentSum.f1 /
26  currentSum.f0));
27              sum.clear();
28          }
29
30      @Override
31      public void open(Configuration config) {
32          ValueStateDescriptor<Tuple2<Long, Long>> descriptor =
33              new ValueStateDescriptor<>(
34                  "average", //状态名称
35                  TypeInformation.of(new TypeHint<Tuple2<Long,
36  Long>>() {}), //类型信息
37                  Tuple2.of(0L, 0L)); //状态的默认值
38          sum = getRuntimeContext().getState(descriptor); //获取状态
39      }
40
41      env.fromElements(Tuple2.of(1L, 3L), Tuple2.of(1L, 5L), Tuple2.of(1L, 7L),
42  Tuple2.of(1L, 4L), Tuple2.of(1L, 2L))
43          .keyBy(0)
44          .flatMap(new CountWindowAverage())
45          .print();
46
47      //结果会打印出 (1,4) 和 (1,5)

```

这个例子实现了一个简单的计数器，我们使用元组的第一个字段来进行分组(这个例子中，所有的 key 都是 1)，这个 CountWindowAverage 函数将计数和运行时总和保存在一个 ValueState 中，一旦计数等于 2，就会发出平均值并清理 state，因此又从 0 开始。请注意，如果在第一个字段中具有不同值的元组，则这将为每个不同的输入 key 保存不同的 state 值。

State 存活时间 (TTL)

TTL 可以分配给任何类型的 Keyed state，如果一个状态设置了 TTL，那么当状态过期时，那么之前存储的状态值会被清除。所有的状态集合类型都支持单个入口的 TTL，这意味着 List 集合元素和 Map 集合都支持独立到期。为了使用状态 TTL，首先必须要构建 StateTtlConfig 配置对象，然后可以通过传递配置在 State descriptor 中启用 TTL 功能：

```
1 import org.apache.flink.api.common.state.StateTtlConfig;
2 import org.apache.flink.api.common.state.ValueStateDescriptor;
3 import org.apache.flink.api.common.time.Time;
4
5 StateTtlConfig ttlConfig = StateTtlConfig
6     .newBuilder(Time.seconds(1))
7     .setUpdateType(StateTtlConfig.UpdateType.OnCreateAndWrite)
8     .setStateVisibility(StateTtlConfig.StateVisibility.NeverReturnExpired)
9     .build();
10
11 ValueStateDescriptor<String> stateDescriptor = new ValueStateDescriptor<>
12     ("zhisheng", String.class);
13 stateDescriptor.enableTimeToLive(ttlConfig);    //开启 ttl
```

上面配置中有几个选项需要注意：

1、newBuilder 方法的第一个参数是必需的，它代表着状态存活时间。

2、UpdateType 配置状态 TTL 更新时（默认为 OnCreateAndWrite）：

- StateTtlConfig.UpdateType.OnCreateAndWrite: 仅限创建和写入访问时更新
- StateTtlConfig.UpdateType.OnReadAndWrite: 除了创建和写入访问，还支持在读取时更新

3、StateVisibility 配置是否在读取访问时返回过期值（如果尚未清除），默认是 NeverReturnExpired：

- StateTtlConfig.StateVisibility.NeverReturnExpired: 永远不会返回过期值
- StateTtlConfig.StateVisibility.ReturnExpiredIfNotCleanedUp: 如果仍然可用则返回

在 NeverReturnExpired 的情况下，过期状态表现得好像它不再存在，即使它仍然必须被删除。该选项对于在 TTL 之后必须严格用于读取访问的数据的用例是有用的，例如，应用程序使用隐私敏感数据。

另一个选项 ReturnExpiredIfNotCleanedUp 允许在清理之前返回过期状态。

注意：

- 状态后端会存储上次修改的时间戳以及对应的值，这意味着启用此功能会增加状态存储的消

耗，堆状态后端存储一个额外的 Java 对象，其中包含对用户状态对象的引用和内存中原始的 long 值。RocksDB 状态后端存储为每个存储值、List、Map 都添加 8 个字节。

- 目前仅支持参考 processing time 的 TTL
- 使用启用 TTL 的描述符去尝试恢复先前未使用 TTL 配置的状态可能会导致兼容性失败或者 StateMigrationException 异常。
- TTL 配置并不是 Checkpoint 和 Savepoint 的一部分，而是 Flink 如何在当前运行的 Job 中处理它的方式。
- 只有当用户值序列化器可以处理 null 值时，具体 TTL 的 Map 状态当前才支持 null 值，如果序列化器不支持 null 值，则可以使用 NullableSerializer 来包装它（代价是需要一个额外的字节）。

清除过期 state

默认情况下，过期值只有在显式读出时才会被删除，例如通过调用 ValueState.value()。

注意：这意味着默认情况下，如果未读取过期状态，则不会删除它，这可能导致状态不断增长，这个特性在 Flink 未来的版本可能会发生变化。

此外，你可以在获取完整状态快照时激活清理状态，这样就可以减少状态的大小。在当前实现下不清除本地状态，但是在从上一个快照恢复的情况下，它不会包括已删除的过期状态，你可以在 StateTtlConfig 中这样配置：

```
1 import org.apache.flink.api.common.state.StateTtlConfig;
2 import org.apache.flink.api.common.time.Time;
3
4 StateTtlConfig ttlConfig = StateTtlConfig
5     .newBuilder(Time.seconds(1))
6     .cleanupFullSnapshot()
7     .build();
```

此配置不适用于 RocksDB 状态后端中的增量 checkpoint。对于现有的 Job，可以在 StateTtlConfig 中随时激活或停用此清理策略，例如，从保存点重启后。

除了在完整快照中清理外，你还可以在后台激活清理。如果使用的后端支持以下选项，则会激活 StateTtlConfig 中的默认后台清理：

```
1 import org.apache.flink.api.common.state.StateTtlConfig;
2 StateTtlConfig ttlConfig = StateTtlConfig
3     .newBuilder(Time.seconds(1))
4     .cleanupInBackground()
5     .build();
```

要在后台对某些特殊清理进行更精细的控制，可以按照下面的说明单独配置它。目前，堆状态后端依赖于增量清理，RocksDB 后端使用压缩过滤器进行后台清理。

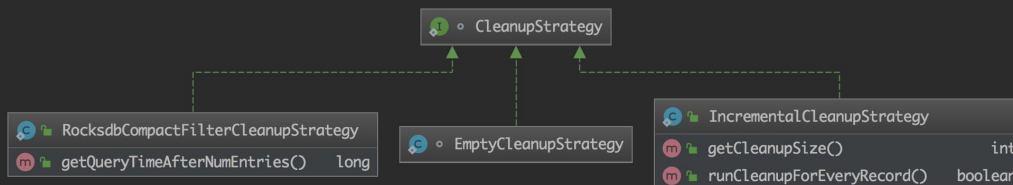
我们再来看看 TTL 对应着的类 StateTtlConfig 类中的具体实现，这样我们才能更加的理解其使用方式。

在该类中的属性有如下：

StateTtlConfig		
serialVersionUID		long
DISABLED		StateTtlConfig
updateType		UpdateType
stateVisibility		StateVisibility
timeCharacteristic		TimeCharacteristic
ttl		Time
cleanupStrategies		CleanupStrategies

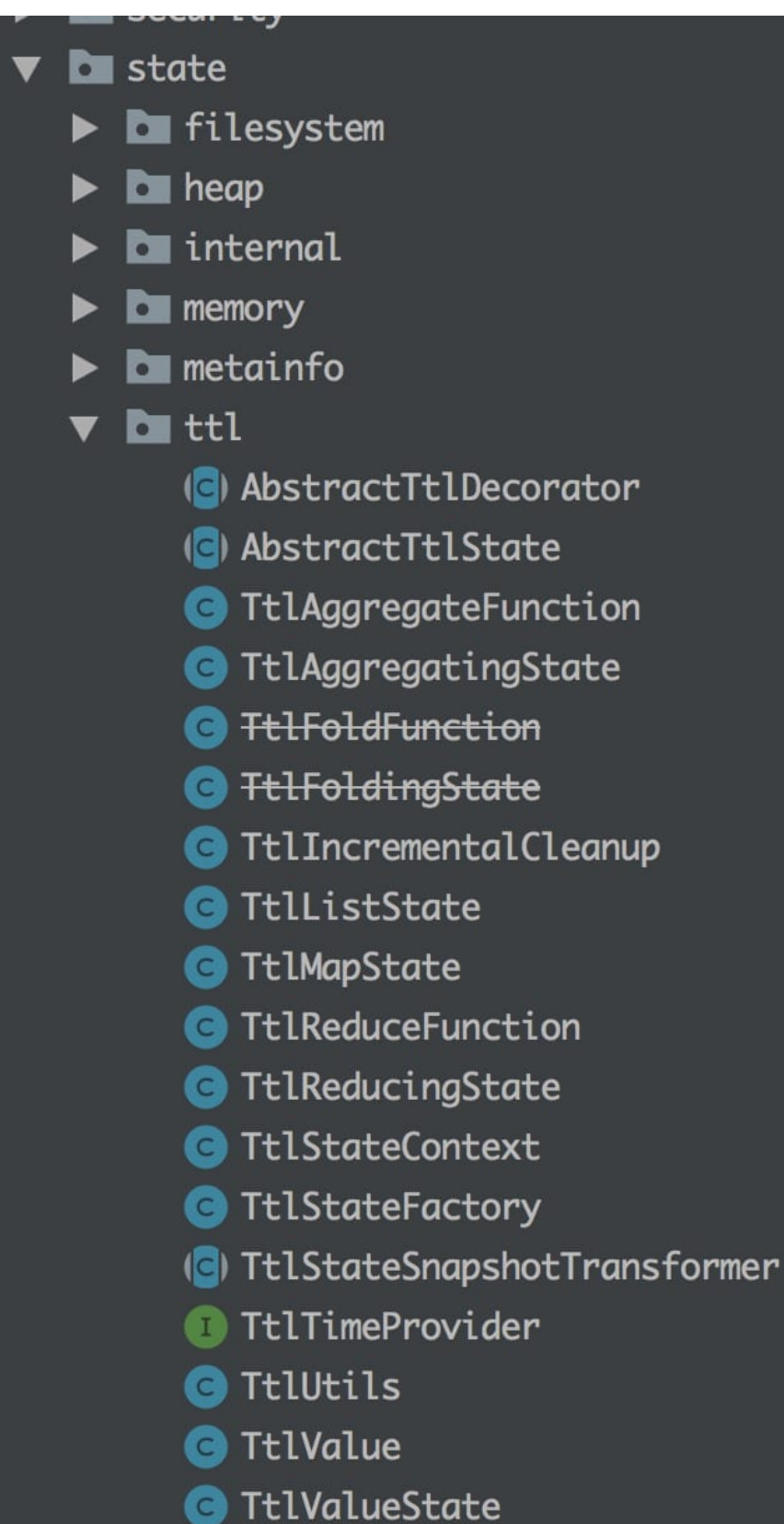
zhisheng

- DISABLED：它默认创建了一个 UpdateType 为 Disabled 的 StateTtlConfig
- UpdateType：这个是一个枚举，包含 Disabled（代表 TTL 是禁用的，状态不会过期）、OnCreateAndWrite、OnReadAndWrite 可选
- StateVisibility：这也是一个枚举，包含了 ReturnExpiredIfNotCleanedUp、NeverReturnExpired
- TimeCharacteristic：这是时间特征，其实是只有 ProcessingTime 可选
- Time：设置 TTL 的时间，这里有两个参数 unit 和 size
- CleanupStrategies：TTL 清理策略，在该类中又有字段 isCleanupInBackground（是否在后台清理）和相关的清理 strategies（包含 FULL_STATE_SCAN_SNAPSHOT、INCREMENTAL_CLEANUP 和 ROCKSDB_COMPACTION_FILTER），同时该类中还有 CleanupStrategy 接口，它的实现类有 EmptyCleanupStrategy（不清理，为空）、IncrementalCleanupStrategy（增量的清除）、RocksdbCompactFilterCleanupStrategy（在 RocksDB 中自定义压缩过滤器）。



zhisheng

如果对 State TTL 还有不清楚的可以看看 Flink 源码 flink-runtime module 中的 state ttl 相关的实现：



使用托管 Operator State

为了使用托管的 Operator State，必须要有一个有状态的函数，这个函数可以实现 CheckpointedFunction 或者 ListCheckpointed 接口。

下面分别讲一下如何使用：

CheckpointedFunction

如果是实现 CheckpointedFunction 接口的话，那么我们先来看下这个接口里面有什么方法呢：

```
1 //当请求 checkpoint 快照时，将调用此方法
2 void snapshotState(FunctionSnapshotContext context) throws Exception;
3
4 //在分布式执行期间创建并行功能实例时，将调用此方法。 函数通常在此方法中设置其状态存储数据
  结构
5 void initializeState(FunctionInitializationContext context) throws
  Exception;
```

当有请求执行 checkpoint 的时候，snapshotState() 方法就会被调用，initializeState() 方法会在每次初始化用户定义的函数时或者从更早的 checkpoint 恢复的时候被调用，因此 initializeState() 不仅是不同类型的状态被初始化的地方，而且还是 state 恢复逻辑的地方。

目前，List 类型的托管状态是支持的，状态被期望是一个可序列化的对象的 List，彼此独立，这样便于重分配，换句话说，这些对象是可以重新分配的 non-keyed state 的最小粒度，根据状态的访问方法，定义了重新分配的方案：

- Even-split redistribution：每个算子会返回一个状态元素列表，整个状态在逻辑上是所有列表的连接。在重新分配或者恢复的时候，这个状态元素列表会被按照并行度分为子列表，每个算子会得到一个子列表。这个子列表可能为空，或包含一个或多个元素。举个例子，如果使用并行性 1，算子的检查点状态包含元素 element1 和 element2，当将并行性增加到 2 时，element1 可能最终在算子实例 0 中，而 element2 将转到算子实例 1 中。
- Union redistribution：每个算子会返回一个状态元素列表，整个状态在逻辑上是所有列表的连接。在重新分配或恢复的时候，每个算子都会获得完整的状态元素列表。

如下示例是一个有状态的 SinkFunction 使用 CheckpointedFunction 来发送到外部之前缓存数据，使用了 Even-split 策略。

下面是一个有状态的 SinkFunction 的示例，它使用 CheckpointedFunction 来缓存数据，然后再将这些数据发送到外部系统，使用了 Even-split 策略：

```
1 public class BufferingSink implements SinkFunction<Tuple2<String,
  Integer>>, CheckpointedFunction {
2
3     private final int threshold;
4
5     private transient ListState<Tuple2<String, Integer>>
  checkpointedState;
6
7     private List<Tuple2<String, Integer>> bufferedElements;
8
9     public BufferingSink(int threshold) {
10         this.threshold = threshold;
11         this.bufferedElements = new ArrayList<>();
12     }
13
14     @Override
```

```

15     public void invoke(Tuple2<String, Integer> value, Context context)
throws Exception {
16         bufferedElements.add(value);
17         if (bufferedElements.size() == threshold) {
18             for (Tuple2<String, Integer> element: bufferedElements) {
19                 //将数据发到外部系统
20             }
21             bufferedElements.clear();
22         }
23     }
24
25     @Override
26     public void snapshotState(FunctionSnapshotContext context) throws
Exception {
27         checkpointedState.clear();
28         for (Tuple2<String, Integer> element : bufferedElements) {
29             checkpointedState.add(element);
30         }
31     }
32
33     @Override
34     public void initializeState(FunctionInitializationContext context)
throws Exception {
35         ListStateDescriptor<Tuple2<String, Integer>> descriptor =
36             new ListStateDescriptor<> (
37                 "buffered-elements",
38                 TypeInformation.of(new TypeHint<Tuple2<String, Integer>>()
{}));
39
40         checkpointedState =
context.getOperatorStateStore().getListState(descriptor);
41
42         if (context.isRestored()) {
43             for (Tuple2<String, Integer> element :
checkpointedState.get()) {
44                 bufferedElements.add(element);
45             }
46         }
47     }
48 }

```

initializeState 方法将 FunctionInitializationContext 作为参数，它用来初始化 non-keyed 状态。注意状态是如何初始化的，类似于 Keyed state，StateDescriptor 包含状态名称和有关状态值的类型的信息：

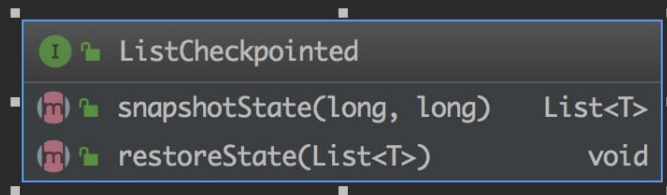
```




1 ListStateDescriptor<Tuple2<String, Integer>> descriptor =
2     new ListStateDescriptor<> (
3         "buffered-elements",
4         TypeInformation.of(new TypeHint<Tuple2<Long, Long>>() {}));
5
6 checkpointedState =
context.getOperatorStateStore().getListState(descriptor);

```

ListCheckpointed

是一种受限的 CheckpointedFunction，只支持 List 风格的状态和 even-split 的重分配策略。该接口里面的方法有：



	ListCheckpointed	
	snapshotState(long, long)	List<T>
	restoreState(List<T>)	void

- snapshotState(): 获取函数的当前状态。状态必须返回此函数先前所有的调用结果。
- restoreState(): 将函数或算子的状态恢复到先前 checkpoint 的状态。此方法在故障恢复后执行函数时调用。如果函数的特定并行实例无法恢复到任何状态，则状态列表可能为空。

Stateful Source Functions

与其他算子相比，有状态的 source 函数需要注意的地方更多，比如为了保证状态的更新和结果的输出原子性，用户必须在 source 的 context 上加锁。

```
1 public static class CounterSource extends RichParallelSourceFunction<Long>
  implements ListCheckpointed<Long> {
2
3     //一次语义的当前偏移量
4     private Long offset = 0L;
5
6     //作业取消标志
7     private volatile boolean isRunning = true;
8
9     @Override
10    public void run(SourceContext<Long> ctx) {
11        final Object lock = ctx.getCheckpointLock();
12
13        while (isRunning) {
14            //输出和状态更新是原子性的
15            synchronized (lock) {
16                ctx.collect(offset);
17                offset += 1;
18            }
19        }
20    }
21
22    @Override
23    public void cancel() {
24        isRunning = false;
25    }
26
27    @Override
```

```

28     public List<Long> snapshotState(long checkpointId, long
checkpointTimestamp) {
29         return Collections.singletonList(offset);
30     }
31
32     @Override
33     public void restoreState(List<Long> state) {
34         for (Long s : state)
35             offset = s;
36     }
37 }

```

或许有些算子想知道什么时候 checkpoint 全部做完了，可以参考使用 `org.apache.flink.runtime.state.CheckpointListener` 接口来实现，在该接口里面有 `notifyCheckpointComplete` 方法。

Broadcast State

https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/stream/state/broadcast_state.html

前面提到了两种 Operator state 支持的动态扩展方法：even-split redistribution 和 union redistribution。Broadcast State 是 Flink 支持的另一种扩展方式，它用来支持将某一个流的数据广播到下游所有的 Task 中，数据都会存储在下流 Task 内存中，接收到广播的数据流后就可以在操作中利用这些数据，一般我们会将一些规则数据进行这样广播下去，然后其他的 Task 也都能根据这些规则数据做配置，更常见的就是规则动态的更新，然后下游还能够动态的感知。

Broadcast state 的特点是：

- 使用 Map 类型的数据结构
- 仅适用于同时具有广播流和非广播流作为数据输入的特定算子
- 可以具有多个不同名称的 Broadcast state

那么我们该如何使用 Broadcast State 呢？下面通过一个例子来讲解一下，在这个例子中，我要广播的数据是监警告警的通知策略规则，然后下游拿到我这个告警通知策略去判断哪种类型的告警发到哪里去，该使用哪种方式来发，静默时间多长等。

第一个数据流是要处理的数据源，流中的对象具有告警或者恢复的事件，其中用一个 `type` 字段来标识哪个事件是告警，哪个事件是恢复，然后还有其他的字段标明是哪个集群的或者哪个项目的，简单代码如下：

```

1 | DataSource<AlertEvent> alertData = env.addSource(new
FlinkKafkaConsumer011<>("alert",
2 |     new AlertEventSchema(),
3 |     parameterTool.getProperties()));

```

然后第二个数据流是要广播的数据流，它是告警通知策略数据（定时从 MySQL 中读取的规则表），简单代码如下：

```

1 | DataStreamSource<Rule> alarmdata = env.addSource(new
   | GetAlarmNotifyData());
2 |
3 | // MapState 中保存 (RuleName, Rule) , 在描述类中指定 State name
4 | MapStateDescriptor<String, Rule> ruleStateDescriptor = new
   | MapStateDescriptor<> (
5 |     "RulesBroadcastState",
6 |     BasicTypeInfo.STRING_TYPE_INFO,
7 |     TypeInformation.of(new TypeHint<Rule>() {}));
8 |
9 | // alarmdata 使用 MapStateDescriptor 作为参数广播, 得到广播流
10 | BroadcastStream<Rule> ruleBroadcastStream =
   | alarmdata.broadcast(ruleStateDescriptor);

```

然后你要做的是将两个数据流进行连接，连接后再根据告警规则数据流的规则数据进行处理（这个告警的逻辑很复杂，我们这里就不再深入讲），伪代码大概如下：

```

1 | alertData.connect(ruleBroadcastStream)
2 |     .process(
3 |         new KeyedBroadcastProcessFunction<AlertEvent, Rule>() {
4 |             //根据告警规则的数据进行处理告警事件
5 |         }
6 |     )
7 |     //可能还有更多的操作

```

`alertData.connect(ruleBroadcastStream)` 该 `connect` 方法将两个流连接起来后返回一个 `BroadcastConnectedStream` 对象，如果对 `BroadcastConnectedStream` 不太清楚的可以回看[文章 4如何使用 DataStream API 来处理数据?](#) 再次复习一下。`BroadcastConnectedStream` 调用 `process()` 方法执行处理逻辑，需要指定一个逻辑实现类作为参数，具体是哪种实现类取决于非广播流的类型：

- 如果非广播流是 keyed stream，需要实现 `KeyedBroadcastProcessFunction`
- 如果非广播流是 non-keyed stream，需要实现 `BroadcastProcessFunction`

那么该怎么获取这个 Broadcast state 呢，它需要通过上下文来获取：

```

1 | ctx.getBroadcastState(ruleStateDescriptor)

```

BroadcastProcessFunction 和 KeyedBroadcastProcessFunction

这两个抽象函数有两个相同的需要实现的接口：

- `processBroadcastElement()`：处理广播流中接收的数据元
- `processElement()`：处理非广播流数据的方法

用于处理非广播流是 non-keyed stream 的情况：


```

1 public abstract class BroadcastProcessFunction<IN1, IN2, OUT> extends
  BaseBroadcastProcessFunction {
2
3     public abstract void processElement(IN1 value, ReadOnlyContext ctx,
      Collector<OUT> out) throws Exception;
4
5     public abstract void processBroadcastElement(IN2 value, Context ctx,
      Collector<OUT> out) throws Exception;
6 }

```

用于处理非广播流是 keyed stream 的情况

```

1 public abstract class KeyedBroadcastProcessFunction<KS, IN1, IN2, OUT> {
2
3     public abstract void processElement(IN1 value, ReadOnlyContext ctx,
      Collector<OUT> out) throws Exception;
4
5     public abstract void processBroadcastElement(IN2 value, Context ctx,
      Collector<OUT> out) throws Exception;
6
7     public void onTimer(long timestamp, OnTimerContext ctx, Collector<OUT>
      out) throws Exception;
8 }

```

可以看到这两个接口提供的上下文对象有所不同。非广播方（processElement）使用 ReadOnlyContext，而广播方（processBroadcastElement）使用 Context。这两个上下文对象（简称 ctx）通用的方法接口有：

- 访问 Broadcast state：ctx.getBroadcastState(MapStateDescriptor<K, V> stateDescriptor)
- 查询数据元的时间戳：ctx.timestamp()
- 获取当前水印：ctx.currentWatermark()
- 获取当前处理时间：ctx.currentProcessingTime()
- 向旁侧输出（side-outputs）发送数据：ctx.output(OutputTag outputTag, X value)

这两者不同之处在于对 Broadcast state 的访问限制：广播方对其具有读和写的权限（read-write），非广播方只有读的权限（read-only），为什么要这么设计呢，主要是为了保证 Broadcast state 在算子的所有并行实例中是相同的。由于 Flink 中没有跨任务的通信机制，在一个任务实例中的修改不能在并行任务间传递，而广播端在所有并行任务中都能看到相同的数据元，只对广播端提供可写的权限。同时要求在广播端的每个并行任务中，对接收数据的处理是相同的。如果忽略此规则会破坏 State 的一致性保证，从而导致不一致且难以诊断的结果。也就是说，processBroadcast() 的实现逻辑必须在所有并行实例中具有相同的确定性行为。

使用 Broadcast state 需要注意

前面介绍了 Broadcast state，并将 BroadcastProcessFunction 和 KeyedBroadcastProcessFunction 做了个对比，那么接下来强调一下使用 Broadcast state 时需要注意的事项：

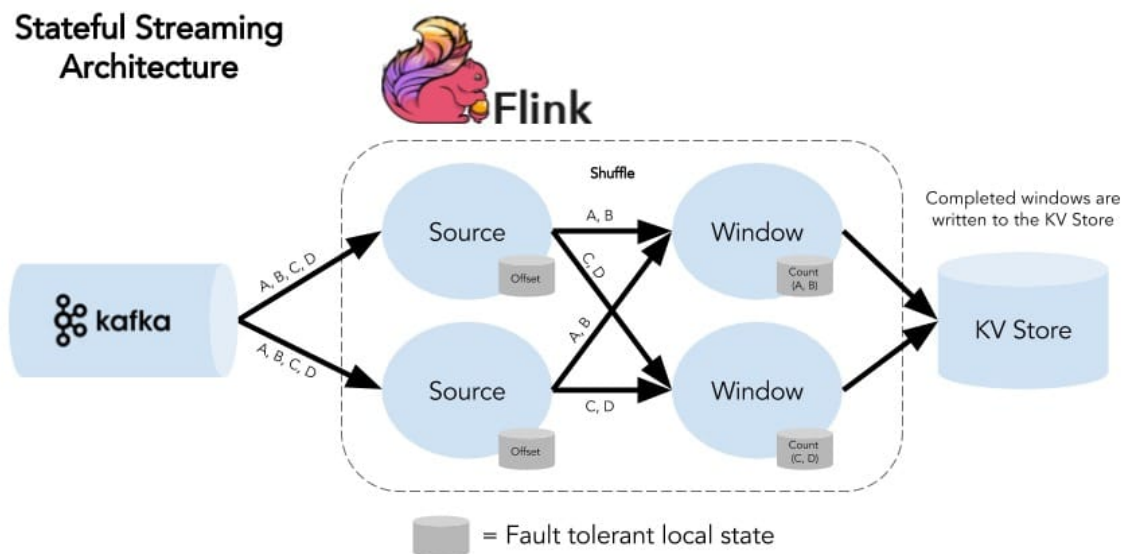
- 没有跨任务的通信，这就是为什么只有广播方可以修改 Broadcast state 的原因。
- 用户必须确保所有任务以相同的方式为每个传入的数据元更新 Broadcast state，否则可能导致

结果不一致。

- 跨任务的 Broadcast state 中的事件顺序可能不同，虽然广播的元素可以保证所有元素都将转到所有下游任务，但元素到达的顺序可能不一致。因此，Broadcast state 更新不能依赖于传入事件的顺序。
- 所有任务都会把 Broadcast state 存入 checkpoint，虽然 checkpoint 发生时所有任务都具有相同的 Broadcast state。这是为了避免在恢复期间所有任务从同一文件中进行恢复（避免热点），然而代价是 state 在 checkpoint 时的大小成倍数（并行度数量）增加。
- Flink 确保在恢复或改变并行度时不会有重复数据，也不会丢失数据。在具有相同或改小并行度后恢复的情况下，每个任务读取其状态 checkpoint。在并行度增大时，原先的每个任务都会读取自己的状态，新增的任务以循环方式读取前面任务的检查点。
- 不支持 RocksDB state backend，Broadcast state 在运行时保存在内存中。

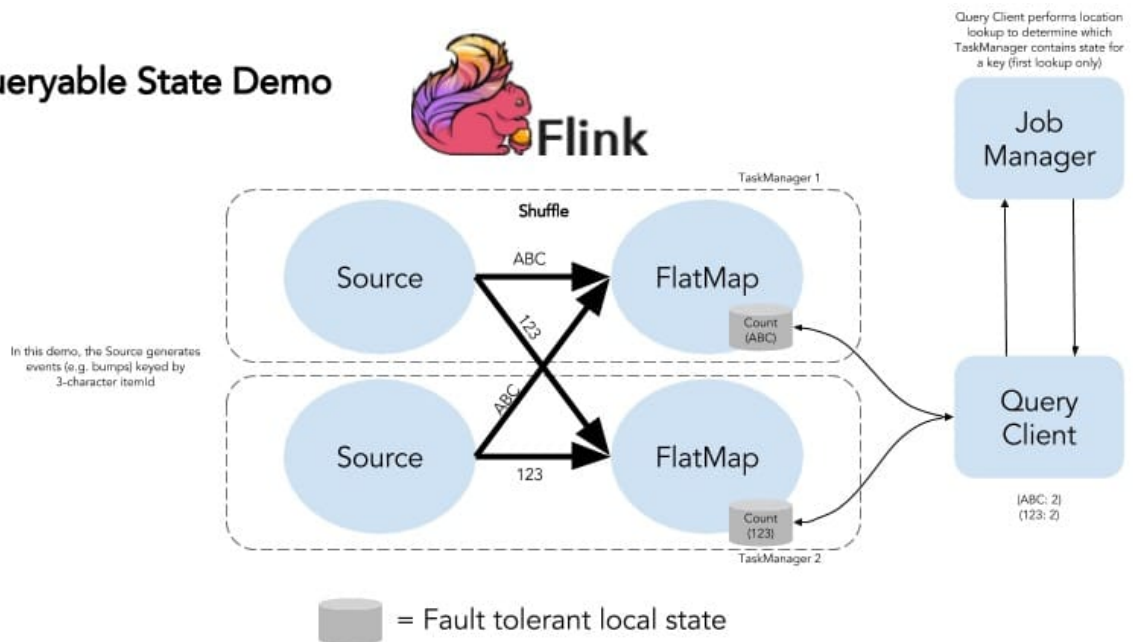
Queryable State

Queryable State，顾名思义，就是可查询的状态。



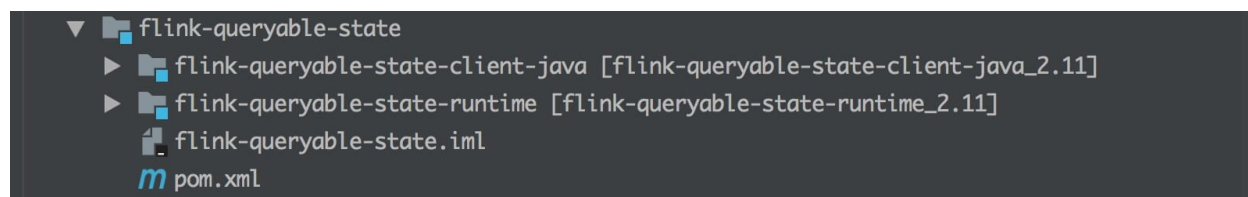
传统管理这些状态的方式是通过将计算后的状态结果存储在第三方 KV 存储中，然后由第三方应用去获取这些 KV 状态，但是在 Flink 中，现在有了 Queryable State，意味着允许用户对流的内部状态进行实时查询。

Queryable State Demo



那么就不再像其他流计算框架，需要将结果存储到其他外部存储系统才能够被查询到，这样我们就可以不再需要等待状态写入外部存储（这块可能是其他系统的主要瓶颈之一），甚至可以做到无需任何数据库就可以让用户直接查询到数据，这使得数据获取到的时间会更短，更及时，如果你有这块的需求（需要将某些状态数据进行展示，比如数字大屏），那么就强烈推荐使用 Queryable State。目前可查询的 state 主要针对可分区的 state，如 keyed state 等。

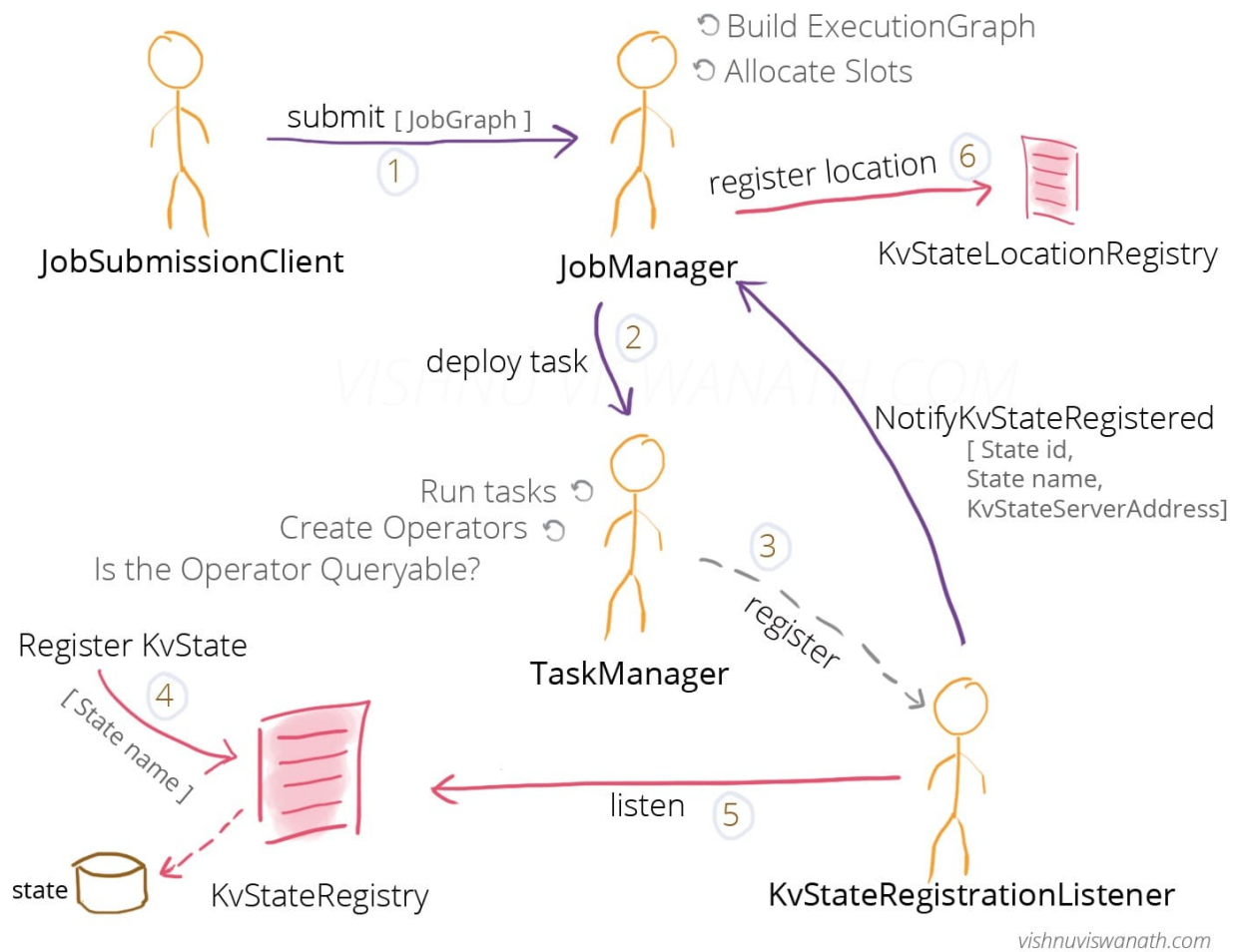
在 Flink 源码中，为此还专门有一个 module 来讲 Queryable State 呢！



那么我们该如何使用 Queryable State 呢？有如下两种方式：

- `QueryableStateStream`, 将 `KeyedStream` 转换为 `QueryableStateStream`, 类似于 Sink, 后续不能进行任何转换操作
- `StateDescriptor#setQueryable(String queryableStateName)`, 将 Keyed State 设置为可查询的 (不支持 Operator State)

外部应用在查询 Flink 应用程序内部状态的时候要使用 `QueryableStateClient`, 提交异步查询请求来获取状态。如何使状态可查询呢，假如已经创建了一个状态可查询的 Job, 并通过 `JobClient` 提交 Job, 那么它在 Flink 内部的具体实现如下图（图片来自 [Queryable States in Apache Flink - How it works](#)）所示：



上面讲解了让 State 可查询的原理，如果要在 Flink 集群中使用的话，首先得将 Flink 安装目录下 opt 里面的 `flink-queryable-state-runtime_2.11-1.8.0.jar` 复制到 lib 目录下，默认 lib 目录是不包含这个 jar 的。

```
drwxr-xr-x@ 22 zhisheng staff 704B 4 4 14:06 .opt
zhisheng@zhisheng: /usr/local/flink-1.8.0$ ll opt
total 455448
-rw-r--r--@ 1 zhisheng staff 52K 4 4 14:05 flink-cep-scala_2.11-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 180K 4 4 14:01 flink-cep_2.11-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 746K 4 4 14:04 flink-gelly-scala_2.11-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 625K 4 4 14:04 flink-gelly_2.11-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 511K 4 4 14:04 flink-metrics-datadog-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 159K 4 4 14:04 flink-metrics-graphite-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 900K 4 4 14:04 flink-metrics-influxdb-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 101K 4 4 14:04 flink-metrics-prometheus-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 10K 4 4 14:04 flink-metrics-slf4j-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 12K 4 4 14:04 flink-metrics-statsd-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 31M 4 4 14:05 flink-ml_2.11-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 36M 4 4 13:59 flink-oss-fs-hadoop-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 139K 4 4 14:04 flink-python_2.11-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 22K 4 4 14:03 flink-queryable-state-runtime_2.11-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 20M 4 4 13:59 flink-s3-fs-hadoop-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 38M 4 4 13:59 flink-s3-fs-presto-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 15M 4 4 14:05 flink-sql-client_2.11-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 40M 4 4 14:03 flink-streaming-python_2.11-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 25M 4 4 13:59 flink-swift-fs-hadoop-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 15M 4 4 14:05 flink-table_2.11-1.8.0.jar
zhisheng@zhisheng: /usr/local/flink-1.8.0$ ll lib
total 274744
-rw-r--r--@ 1 zhisheng staff 92M 4 4 14:06 flink-dist_2.11-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 41M 4 27 10:48 flink-shaded-hadoop2-uber-2.8.3-1.8.0.jar
-rw-r--r--@ 1 zhisheng staff 478K 1 7 19:38 log4j-1.2.17.jar
-rw-r--r--@ 1 zhisheng staff 9.7K 1 7 19:38 slf4j-log4j12-1.7.15.jar
zhisheng@zhisheng: /usr/local/flink-1.8.0$
```

然后你可以像下面这样操作让状态可查询：

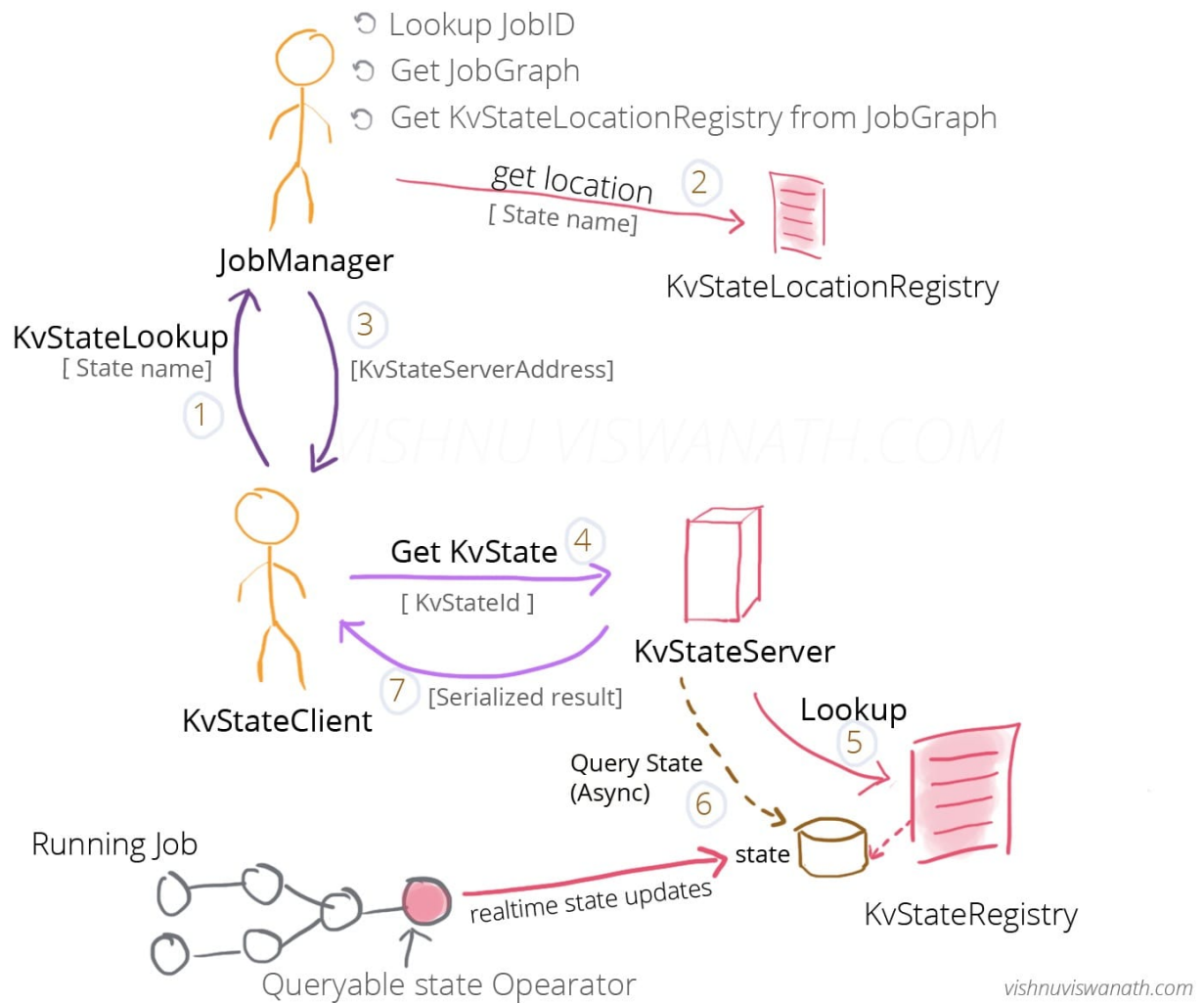
```

1 // Reducing state
2 ReducingStateDescriptor<Tuple2<Integer, Long>> reducingState = new
ReducingStateDescriptor<> (
3     "zhisheng",
4     new SumReduce(),
5     source.getType());
6
7 final String queryName = "zhisheng";
8
9 final QueryableStateStream<Integer, Tuple2<Integer, Long>> queryableState
=
10     dataStream.keyBy(new KeySelector<Tuple2<Integer, Long>, Integer>()
{
11         private static final long serialVersionUID =
-4126824763829132959L;
12         @Override
13         public Integer getKey(Tuple2<Integer, Long> value) {
14             return value.f0;
15         }
16     }).asQueryableState(queryName, reducingState);

```

除了上面的 Reducing，你还可以使用 ValueState、FoldingState，还可以直接通过 asQueryableState(queryName)，注意不支持 ListState，调用 asQueryableState 方法后会返回 QueryableStateStream，接着无需再做其他操作。

那么用户如果定义了 Queryable State 的话，该怎么来查询对应的状态呢？下面来看看具体逻辑：



简单来说，当用户在 Job 中定义了 queryable state 之后，就可以在外部通过 QueryableStateClient 来查询对应的状态实时值，你可以创建如下方法：

```

1 //创建 Queryable State Client
2 QueryableStateClient client = new QueryableStateClient(host, port);
3
4 public QueryableStateClient(final InetAddress remoteAddress, final int
  remotePort) {
5     ...
6     this.client = new Client<>(
7         "Queryable State Client", 1,
8         messageSerializer, new DisabledKvStateRequestStats());
9 }

```

在 QueryableStateClient 中有几个不同参数的 `getKvState` 方法，参数可有 JobID、queryableStateName、key、namespace、keyTypeInfo、namespaceTypeInfo、StateDescriptor，其实内部最后调用的是一个私有的 `getKvState` 方法：


```

1 private CompletableFuture<KvStateResponse> getKvState(
2     final JobID jobId, final String queryableStateName,
3     final int keyHashCode, final byte[] serializedKeyAndNamespace) {
4     ...
5     //构造 KV state 查询的请求
6     KvStateRequest request = new KvStateRequest(jobId, queryableStateName,
7         keyHashCode, serializedKeyAndNamespace);
8     //这个 client 是在构造 QueryableStateClient 中赋值的, 这个 client 是
9     Client<KvStateRequest, KvStateResponse>, 发送请求后会返回
10    CompletableFuture<KvStateResponse>
11    return client.sendRequest(remoteAddress, request);
12    ...
13 }

```

在 Flink 源码中专门有一个 `QueryableStateOptions` 类来设置可查询状态相关的配置，有如下这些配置。

服务器端：

- `queryable-state.proxy.ports`：可查询状态代理的服务器端口范围的配置参数，默认是 9069
- `queryable-state.proxy.network-threads`：客户端代理的网络线程数，默认是 0
- `queryable-state.proxy.query-threads`：客户端代理的异步查询线程数，默认是 0
- `queryable-state.server.ports`：可查询状态服务器的端口范围，默认是 9067
- `queryable-state.server.network-threads`：KvState 服务器的网络线程数
- `queryable-state.server.query-threads`：KvStateServerHandler 的异步查询线程数
- `queryable-state.enable`：是否启用可查询状态代理和服务端

客户端：

- `queryable-state.client.network-threads`：KvState 客户端的网络线程数

注意：

可查询状态的生命周期受限于 Job 的生命周期，例如，任务在启动时注册可查询状态，在清理的时候会注销它。在未来的版本中，可能会将其解耦，以便在任务完成后仍可以允许查询到任务的状态。

总结

本文对 Flink 的 State 做了一个很详尽的讲解，不管是从使用方面，还从原理进行深度分析，涉及的有 State 的分类如 Keyed State、Operator State、Raw State、Managed State、Broadcast State 等。还讲了如何让 State 进行可查询的配置，State 的过期，目的就是让大家对 State 彻底的了解使用方式和原理实现。

下面一图来看看 State 在 Flink 中的整体结构：

