**Flink状态管理**

**(can for FaultTorellent + re-scale)**

对于持续生成新数据的场景，采用流计算显然是有利的。数据源源不断的产生，流计算[系统](https://www.2cto.com/os/)理论上就要不间断的提供数据计算（可以停机维护的场景不在本文的讨论范围）。那么假如遇到下面的几种情况，流计算是如何保证数据的一致性的呢？

1、应用程序bug修复，即功能的修改

2、应用程序增加、删除新的功能

3、流计算框架版本的升级

4、突发的大量数据的到来

以上列出的几种情况，我相信在大多数的流数据场景下，都可能遇到。对于以上的这些情况，我们在流计算系统中，分别会产生什么问题呢？

1、 bug修复--》停止job，重新发布新的jar包后，之前每个operator的状态数据，

在重启job时还能继续用么？

2、 功能修复--》停止job，重新发布新的jar包后，新的或已删除的operator的状态，怎么办呢？

3、框架版本升级--》相同的jar包，在升级前后，状态能兼容么？

4、突发大量数据--》势必会导致严重的背压（backpressure），临时增加集群规模（扩容）

，状态还能正确恢复么？

上述这几种情况所面临的问题，相信大多数的流计算程序员或者架构人员都要考虑。幸好，**Apache Flink的savepoint机制**，让这一切变得简单而高效！

**1、Flink中的savepoint（保存点）**

savepoint是做什么的，有什么作用？简单而言，它是检查点的一个指针，提供了让“时间倒流”的功能，可以让Flink流计算程序重新处理过去的数据。

这种能力需要设置几个条件：

1、激活检查点

2、使用可重发的数据源

3、状态要被Flink管理

4、合适的state backend

**2、reprocessing的思考**

Flink通过savepoint机制，可以让流处理程序比较优雅的处理bug修复，程序升级、集群扩容等需求。实际上，我们只需要获取以下3种数据即可：

1、应用程序的jar包

2、保存点对应的快照（实际是检查点产生的）

3、可访问的保存点和检查点路径

其余的[组件](https://www.2cto.com/kf/all/zujian/)，都可以认为是临时性的。

## 3、Flink管理的状态

有人问过我流计算中，状态是指什么？

这个问题很好回答，我举个例子：  
假如输入数据为 e = {event\_id:int, event\_value:int}。  
如果输出仅仅是event\_value，那就用个map即可，这就是无状态的流计算。  
如果输出是最大的event\_value,那就需要在map函数中，记住之前最大的event\_value,然后再与当前数据的event\_value比较，去输出最终最大的event\_value。

然后有人就问，这个用一个HashMap不也行么？或者把更复杂的临时数据，存到redis等也可以达到相同的目的啊。  
考虑一下，你如果存到HashMap中，程序一旦失败，自动恢复后，此时HashMap中的数据还有么？显然是无法拿到的。

Flink本身就是一个有状态的分布式流计算系统。在提交job时，Flink逻辑上将所有的operator分解成job graph，物理上分解为并行的execution graph。每个并行的slot都是一个独立的task或subtask，同一个operator的不同的并行slot之间，并不共享数据。整个的DAG图中，数据只是从上游的operator流向下游的operator。

就数据本地性而言，Flink中的状态数据总是绑定到特定的task上。基于这种设计，一个task的状态总是本地的，在tasks之间没有通信。

Flink中有两种类型的状态：

1、operator state

2、keyed state

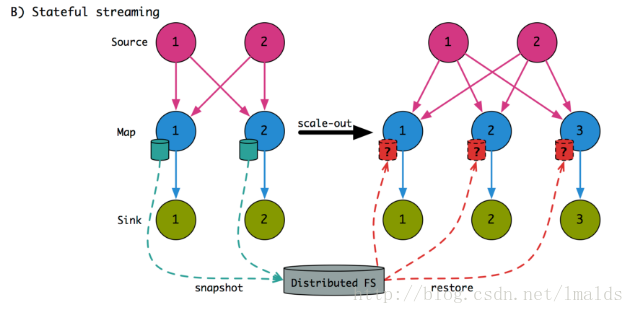
每个operator state一定会绑定到一个特定的operator，其是属于一个operator的，例如实现了ListCheckpointed接口的operator。比如kafka就是一个很好的operator的例子。

keyed state就相当于分组后的operator state。其通常要基于keyedStream。

参见官方文档

## 4、扩容时的思考

考虑下假如有非常大的数据到来，我们想要扩容来应对这些增加的数据，例如下图：



当map的并行读由2变为3后，我们可以将之前的map\_1和map\_2的状态对应到新的map\_1和map\_2,留给map\_3一个空的状态。

依赖于状态的类型以及具体的操作，这种方法很低效可能导致结果不正确。

那么Flink中，是如何做到真正的扩容的呢？

首先，Flink根据状态的分类，分别使用不同的扩容方法。

我们先看下对于operator state的方法：

## 4.1、扩容时operator state上的状态的重分配

FlinkKafkaConsumer是一个典型的operator state的例子。Flink kafka consumer，维护着消费的每个partition的offset信息。

当检查点激活时，系统会定期对kafka consumer做快照，调用snapshotState()方法。在Flink 1.1的版本中，这个方法返回的是一个HashMap，即每个并行的实例返回的都是自己的slot处理的partition的offset信息，代码如下：

<code>@Override

public HashMap<kafkatopicpartition, long=""> snapshotState(long checkpointId, long checkpointTimestamp)

throws Exception {

if (!running) {

LOG.debug("snapshotState() called on closed source");

return null;

}

final AbstractFetcher<!--?, ?--> fetcher = this.kafkaFetcher;

if (fetcher == null) {

// the fetcher has not yet been initialized, which means we need to return the

// originally restored offsets

return restoreToOffset;

}

HashMap<kafkatopicpartition, long=""> currentOffsets = fetcher.snapshotCurrentState();

if (LOG.isDebugEnabled()) {

LOG.debug("Snapshotting state. Offsets: {}, checkpoint id: {}, timestamp: {}",

KafkaTopicPartition.toString(currentOffsets), checkpointId, checkpointTimestamp);

}

// the map cannot be asynchronously updated, because only one checkpoint call can happen

// on this function at a time: either snapshotState() or notifyCheckpointComplete()

pendingCheckpoints.put(checkpointId, currentOffsets);

// truncate the map, to prevent infinite growth

while (pendingCheckpoints.size() > MAX\_NUM\_PENDING\_CHECKPOINTS) {

pendingCheckpoints.remove(0);

}

return currentOffsets;

}</kafkatopicpartition,></kafkatopicpartition,></code>

这样，假如进行扩容，扩大到20个并行度，那么恢复时，kafka source中的快照中，只有10个HashMap对象，此时无法分配给20个slot来处理，也就是说增加的10个并行度，根本没有参与运算。

在Flink 1.2+的版本中，Flink对检查点做了轻微的修改，引入了一个全局的ListState变量：

<code><code>private transient ListState<tuple2<kafkatopicpartition, long="">> offsetsStateForCheckpoint;</tuple2<kafkatopicpartition,></code></code>

即每个并行的实例会把对儿放入全局的状态变量offsetsStateForCheckpoint中，这样在恢复时，就可以对这个ListState遍历，重新分配到不同的slot中，从而实现扩容后的重分配了，具体代码如下：

<code><code><code>public class FlinkKafkaConsumer<t> extends

RichParallelSourceFunction<t> implements CheckpointedFunction {

// ...

private transient ListState<tuple2<kafkatopicpartition, long="">> offsetsOperatorState;

@Override

public void initializeState(FunctionInitializationContext context) throws Exception {

OperatorStateStore stateStore = context.getOperatorStateStore();

// register the state with the backend

this.offsetsOperatorState = stateStore.getSerializableListState("kafka-offsets");

// if the job was restarted, we set the restored offsets

if (context.isRestored()) {

for (Tuple2<kafkatopicpartition, long=""> kafkaOffset :

offsetsOperatorState.get()) {

// ... restore logic

}

}

}

@Override

public void snapshotState(FunctionSnapshotContext context) throws Exception {

this.offsetsOperatorState.clear();

// write the partition offsets to the list of operator states

for (Map.Entry<kafkatopicpartition, long=""> partition :

this.subscribedPartitionOffsets.entrySet()) {

this.offsetsOperatorState.add(Tuple2.of(partition.getKey(), partition.getValue()));

}

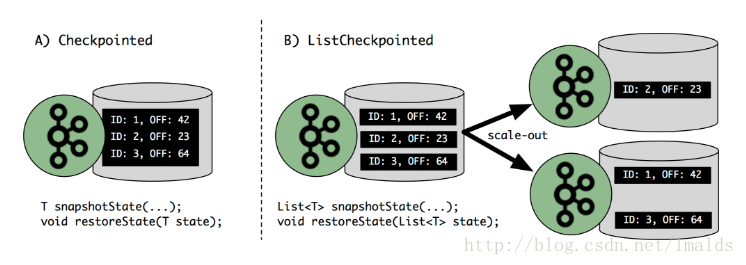
}

// ...

}

</kafkatopicpartition,></kafkatopicpartition,></tuple2<kafkatopicpartition,></t></t>

</code></code></code>



上图描述了针对kafka consumer，原来的Checkpointed接口与现在的ListCheckpointed接口的区别。

通过这个例子我们可以看出，Flink 1.2之后的版本，在扩容后，依然可以做到状态的重分配的。<喎�"/kf/ware/vc/" target="\_blank" class="keylink">vcD4NCjxoMiBpZD0="42扩容时keyed-state上的状态的重分配">4.2、扩容时keyed state上的状态的重分配

keyed State仅仅对于keyedStream是可用的，一般是通过keyBy()实现。相同的key，会通过hash方法，对这个operator的并行度取模操作，对应到相同的operator实例中。

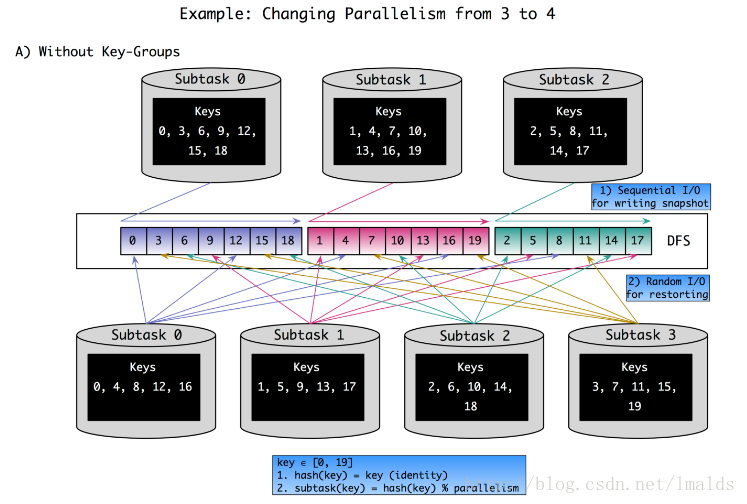
对于keyed State扩容后的状态重分配，可以有这么几种解决办法：

## 4.2.1、对key进行hash取模重分配

正是由于这个操作hash(key) mod parallelism(operator)，因此，在扩容时的状态重分配上，keyed state比operator state有一个明显的优势，keyed state很容易实现状态的重分配。

然而，这种简单的重分配方案，在扩容后也存在一个问题：hash(key) mod parallelism(operator)这个操作，很容易导致新分配的subtask，其处理的state，并不是之前本地操作的状态。

我们举个例子：有20个不同的key，并行度由3变为4，我们看看通过hash(key) mod parallelism(operator)这种方法导致的状态在重分配时的变化情况，如下图：



首先，我们解释下图中两个等式的含义：

<code><code><code>key∈[0,19]

1、hash(key) = key(identity)-->简单的理解为把key本身的值赋给key的hash值

2、subtask(key) = hash(key) % parallelism -->hash值除以并行度取余，即取模操作</code></code></code>

这样算下来，当并行度是3时，每个并行的operator实例所包含的key为：

<code><code><code>subtask 0：0,3，6,9,12,15,18

subtask 1：1,4,7,10,13,16,19

subtask 2：2,5,8,11,14,17 </code></code></code>

当并行度变为4，此时每个并行的operator实例包含的key为：

<code><code><code>subtask 0：0,4,8,12,16

subtask 1：1,5,9,13,17

subtask 2：2,6,10,14,18

subtask 3:3,7,11,15,19 </code></code></code>

可以看到，原来的subtask0,1,2在扩容后，所处理的key，大部分都不是扩容前自己实例本地处理过的key，即读到的大部分都是不相关的数据，这样导致新的subtask在恢复状态时，效率比较低。

因此，我们来看第二种解决办法。

## 4.2.2、对key进行标记并跟踪location

这个方法就是在检查点时，对于每个key监理一个索引号来跟踪本operator实例生成的状态中有哪些key，这样在扩容后恢复时，就可以有选择性的读取本地实例生成的key的状态。

这个方法虽然可以避免读取很多不相关的状态数据，但是其也有2个明显的缺点：

<code><code><code>1、key到index的映射，可能会增长的非常大。

2、这种方法会产生巨大的随机IO操作。</code></code></code>

因此，这种解决方法的性能非常的差。

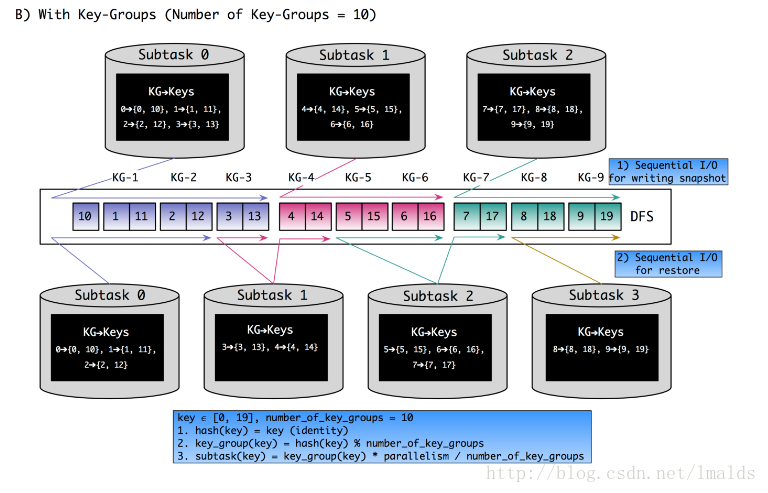
## 4.2.3、Flink中的key-groups组

Flink对于keyed state的扩容后状态重分配的解决办法介于两者之间，其引入了key-group的概念。key-group是状态分配的原子单位。

首先，key-group的数量在job启动前必须是确定的且运行中不能改变。由于key-group是state分配的原子单位，而每个operator并行实例至少包含一个key-group，因此operator的最大并行度不能超过设定的key-group的个数。

总而言之，key-group在扩容时的灵活性与恢复时的负载之间提供了一种解决办法。每个key-group是以key的范围来组织，这就使得我们在恢复时不仅可以顺序读取key-group，而且可以跨多个key-group读取。

这里边说的太抽象，我们还是以一个例子来说明，并行度从3到4，有10个key-group的情况：



首先，我们还是解释下图中三个等式的含义：

首先，我们看下并行度是3时，每个并行的operator实例所包含的key-group与key为：

<code><code><code>key∈[0,19] ， key-group的数量： 10

1、hash(key) = key(identity)-->简单的理解为把key本身的值赋给key的hash值

2、key\_group(key) = hash(key) % number\_of\_key\_groups-->hash值除以10取余，先把key分到对应的key-group中

3、subtask(key) = key\_group(key) \* parallelism / number\_of\_key\_groups-->这一步决定每个key-group会分到哪一个并行实例中，用组号\*并行度3，再除以10取整。例如key = 15，则key-group = 5；其subtask = 5 \* 3 / 10 = 1,即key为15的数据，落到了subtask是1的实例上。</code></code></code>

当并行度变为4时，我们看看key被分到的不同的subtask，变化不大，例如原来的key-group只有3被分出去了，从subtask 0到了subtask 1。但是大部分key在扩容后，其状态依然是本地的operator实例上。

这就很好的平衡了最开始的2种分配方法，既做到了扩容后，状态尽量本地化，同时也做到了恢复时读取state的高性能。

**state backend**

MemoryStateBackend,FsStateBackend, 和RocksDBStateBackend。其中MemoryStateBackend主要用于调试开发中使用，后面2者适合于生产环境中使用。这三种实现均继承自AbstractStateBackend类。在StreamTask的初始化过程中会初始化OperatorChain中所有的operator，而AbstractKeyedBackend也是在这个过程中初始化的，而且一个StreamTask中只有一个，其实这个也符合常理，因为多个key by操作产生的operator必然在不同的线程中。

**AbstractStateBackend**

AbstractStateBackend的定义比较简单，它要求子类实现三个接口：

createStreamFactory: 为某个job的某个operator创建CheckpointStreamFactory，实际上只有FsStateBackend实现了这个接口，RocksDBStateBackend的实现需要传入一个AbstractStateBackend，通常情况下是FsStateBackend。

createKeyedStateBackend: 创建一个keyed state backend用来管理keyed state

createOperatorStateBackend: 创建一个OperateStateBackend，AbstractStateBackend提供了一个实现，就是在内存中的一个Map，key是state名字，value就是list state。

**FsStateBackend**

FsStateBackend会在checkpoint的时候将state存储到一个持久化的存储中，比如hdfs。对于keyed state，FsStateBackend简单地将其放置在内存中，因此对于比较大的state，FsStateBackend有可能会**引起比较严重的GC**。而且snapshot的过程是一个同步的过程，也就是说将state序列化并写入文件系统的过程是一个**同步**的过程，过大的state同样可能引起阻塞。

**RocksDBStateBackend**

RocksDBStateBackend与FsStateBackend不同，它将key state存储在rocksdb中。这种做法有2个好处：首先，比较大的state**不会引起GC**；其次，由于rocksdb支持snapshot操作，因此snapshot的过程是一个**异步**的过程，不会阻塞。但是rocksdb实现的state也会有几个可能的缺点：首先，state的update和get操作都会有一个序列化和反序列化的过程，因此效率会比直接在内存中低；其次，rocksdb使用LSM-Tree作为存储结构，compaction过程需要大量的读写磁盘，因此也有可能引起阻塞，对于这个问题一个可能优化是使用memory filesystem，将所有的存储放在内存中；最后，rocksdb的tuning比较复杂，在普通的SATA硬盘上表现如何还需要确认。