Apache Flink 漫谈系列 - State

实际问题

在流计算场景中，数据会源源不断的流入Apache Flink系统，每条数据进入Apache Flink系统都会触发计算。如果我们想进行一个Count聚合计算，那么每次触发计算是将历史上所有流入的数据重新计算一次，还是每次计算都是在上一次计算结果之上进行增量计算呢?答案是肯定的，Apache Flink是基于上一次的计算结果进行增量计算的。那么问题来了: "上一次的计算结果保存在哪里，保存在内存可以吗?"，答案是否定的，如果保存在内存，在由于网络，硬件等原因造成某个计算节点失败的情况下，上一次计算结果会丢失，在节点恢复的时候，就需要将历史上所有数据(可能十几天，上百天的数据)重新计算一次，所以为了避免这种灾难性的问题发生，Apache Flink 会利用State存储计算结果。本篇将会为大家介绍Apache Flink State的相关内容。

什么是State

这个问题似乎有些"弱智"？不管问题的答案是否显而易见，但我还是想简单说一下在Flink里面什么是State？State是指流计算过程中计算节点的中间计算结果或元数据属性，比如 在aggregation过程中要在state中记录中间聚合结果，比如 Apache Kafka 作为数据源时候，我们也要记录已经读取记录的offset，这些State数据在计算过程中会进行持久化(插入或更新)。所以Flink中的State就是与时间相关的，Flink任务的内部数据（计算数据和元数据属性）的快照。

为什么需要State

与批计算相比，State是流计算特有的，批计算没有failover机制，要么成功，要么重新计算。流计算在 大多数场景 下是增量计算，数据逐条处理（大多数场景)，每次计算是在上一次计算结果之上进行处理的，这样的机制势必要将上一次的计算结果进行存储（生产模式要持久化），另外由于 机器，网络，脏数据等原因导致的程序错误，在重启job时候需要从成功的检查点(checkpoint，后面篇章会专门介绍)进行state的恢复。增量计算，Failover这些机制都需要state的支撑。

State 存储实现

Flink内部有三种state的存储实现，具体如下:

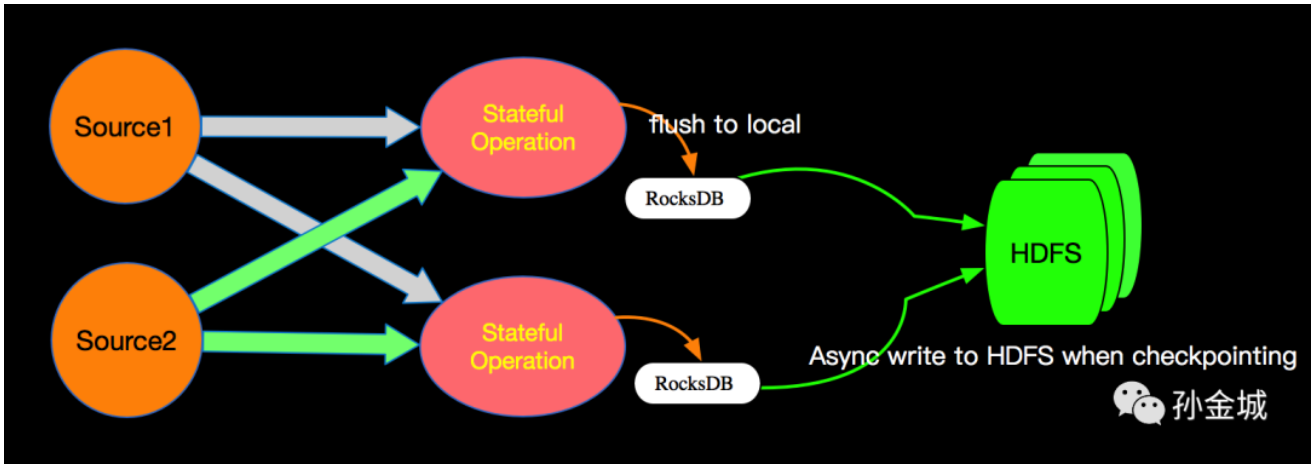
- 基于内存的HeapStateBackend - 在debug模式使用，不 建议在生产模式下应用;

- 基于HDFS的FsStateBackend - 分布式文件持久化，每次读写都操作内存，同需考虑OOM问题；

- 基于RocksDB的RocksDBStateBackend - 本地文件+异步HDFS持久化；

State存储的架构

Apache Flink 默认是RocksDB+HDFS的方式进行State的存储，State存储分两个阶段，首先本地存储到RocksDB，然后异步的同步到远程的HDFS。这样的而设计既消除了HeapStateBackend的局限（内存大小，机器坏掉丢失等），也减少了纯分布式存储的网络IO开销。



State 分类

- KeyedState - 这里面的key是我们在SQL语句中对应的GroupBy/PartitioneBy里面的字段，key的值就是groupby/PartitionBy字段组成的Row的字节数组，每一个key都有一个属于自己的State，key与key之间的State是不可见的；

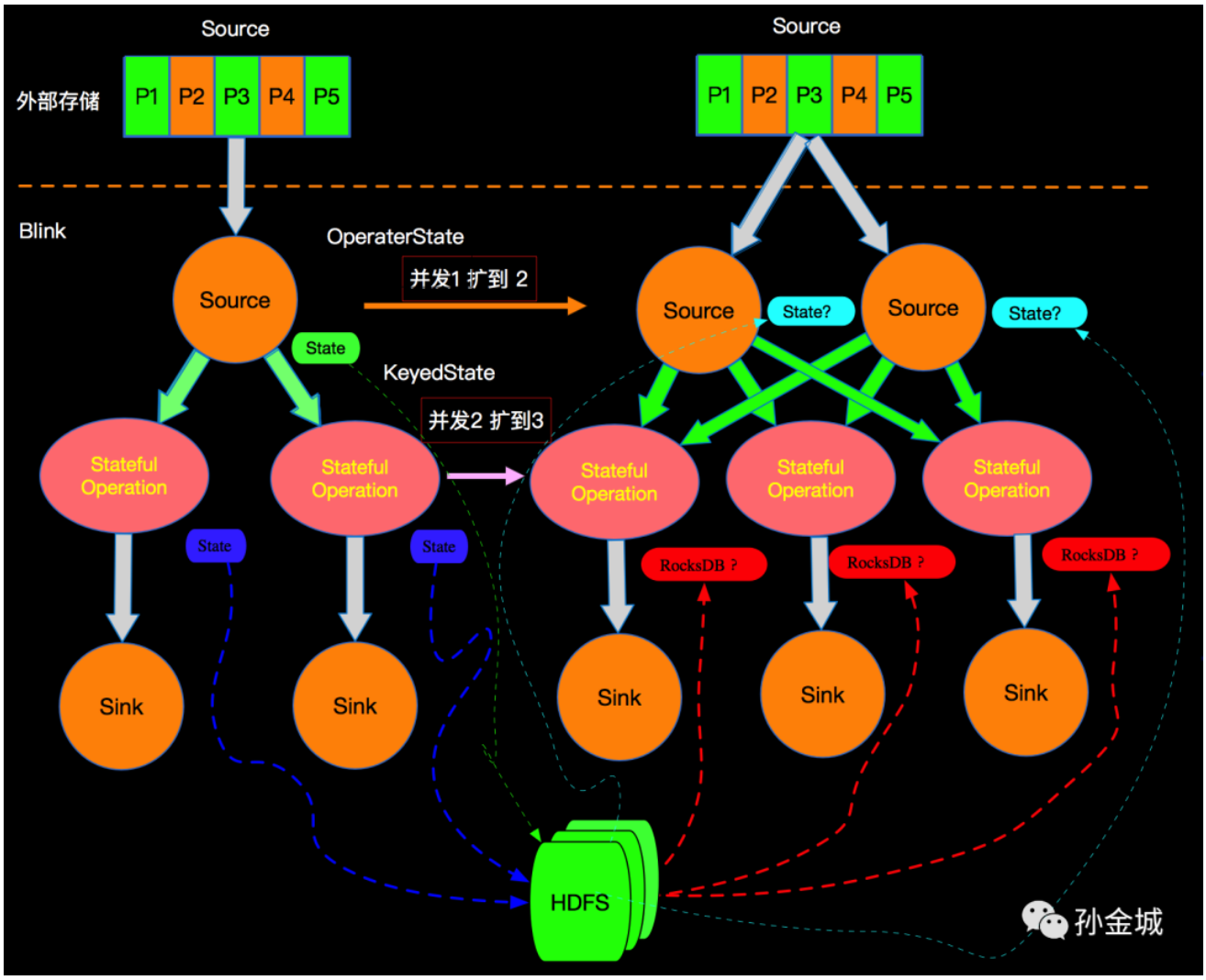
- OperatorState - Flink内部的Source Connector的实现中就会用OperatorState来记录source数据读取的offset。

State在扩容时候的重新分配

Flink是一个大规模并行分布式系统，允许大规模的有状态流处理。为了可伸缩性，Flink作业在逻辑上被分解成operator graph，并且每个operator的执行被物理地分解成多个并行运算符实例。从概念上讲，Flink中的每个并行运算符实例都是一个独立的任务，可以在自己的机器上调度到网络连接的其他机器运行。

Flink的DAG图中只有边相连的节点有网络通信，也就整个DAG在垂直方向有网络IO，在水平方向如下图的stateful节点之间没有网络通信，这种模型也保证了每个operator实例维护一份自己的state，并且保存在本地磁盘（远程异步同步）。通过这种设计，任务的所有状态数据都是本地的，并且状态访问不需要任务之间的网络通信。避免这种流量对于像Flink这样的大规模并行分布式系统的可扩展性至关重要。

如上我们知道Flink中State有OperatorState和KeyedState，那么在进行扩容时候（增加并发）State如何分配呢？比如：外部Source有5个partition，在Flink上面由Source的1个并发扩容到2个并发，中间Stateful Operation 节点由2个并发并扩容的3个并发，如下图所示:



在Flink中对不同类型的State有不同的扩容方法，接下来我们分别介绍。

OperatorState对扩容的处理

我们选取Flink中某个具体Connector实现实例进行介绍，以MetaQ为例，MetaQ以topic方式订阅数据，每个topic会有N>0个分区，以上图为例，假设我们订阅的MetaQ的topic有5个分区，那么当我们source由1个并发调整为2个并发时候，State是怎么恢复的呢？

State 恢复的方式与Source中OperatorState的存储结构有必然关系，我们先看MetaQSource的实现是如何存储State的。首先MetaQSource 实现了ListCheckpointed<T extends Serializable>，其中的

T是Tuple2<InputSplit,Long>，我们在看ListCheckpointed接口的内部定义如下：

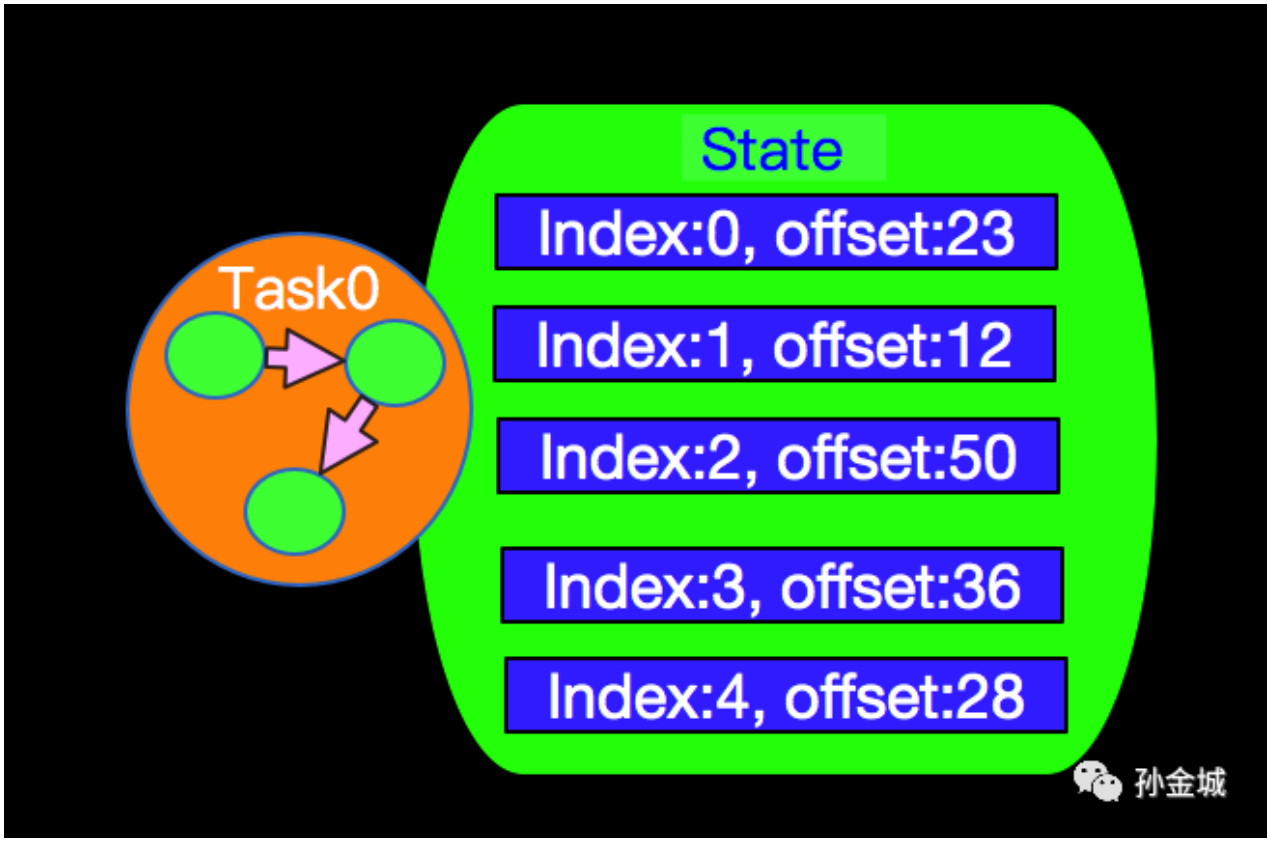
```
public interface ListCheckpointed<T extends Serializable> {
    List<T> snapshotState(long var1, long var3) throws Exception;

    void restoreState(List<T> var1) throws Exception;
}
```

我们发现 snapshotState 方法的返回值是一个List<T>,T是Tuple2<InputSplit,Long>，也就是snapshotState方法返回List<Tuple2<InputSplit,Long>>,这个类型说明state的存储是一个包含partiton和offset信息的列表，InputSplit代表一个分区，Long代表当前partition读取的offset。InputSplit有一个方法如下：

```
public interface InputSplit extends Serializable {
    int getSplitNumber();
}
```

也就是说，InputSplit我们可以理解为是一个Partition索引，有了这个数据结构我们在看看上面图所示的case是如何工作的？当Source的并行度是1的时候，所有打partition数据都在同一个线程中读取，所有partition的state也在同一个state中维护，State存储信息格式如下：

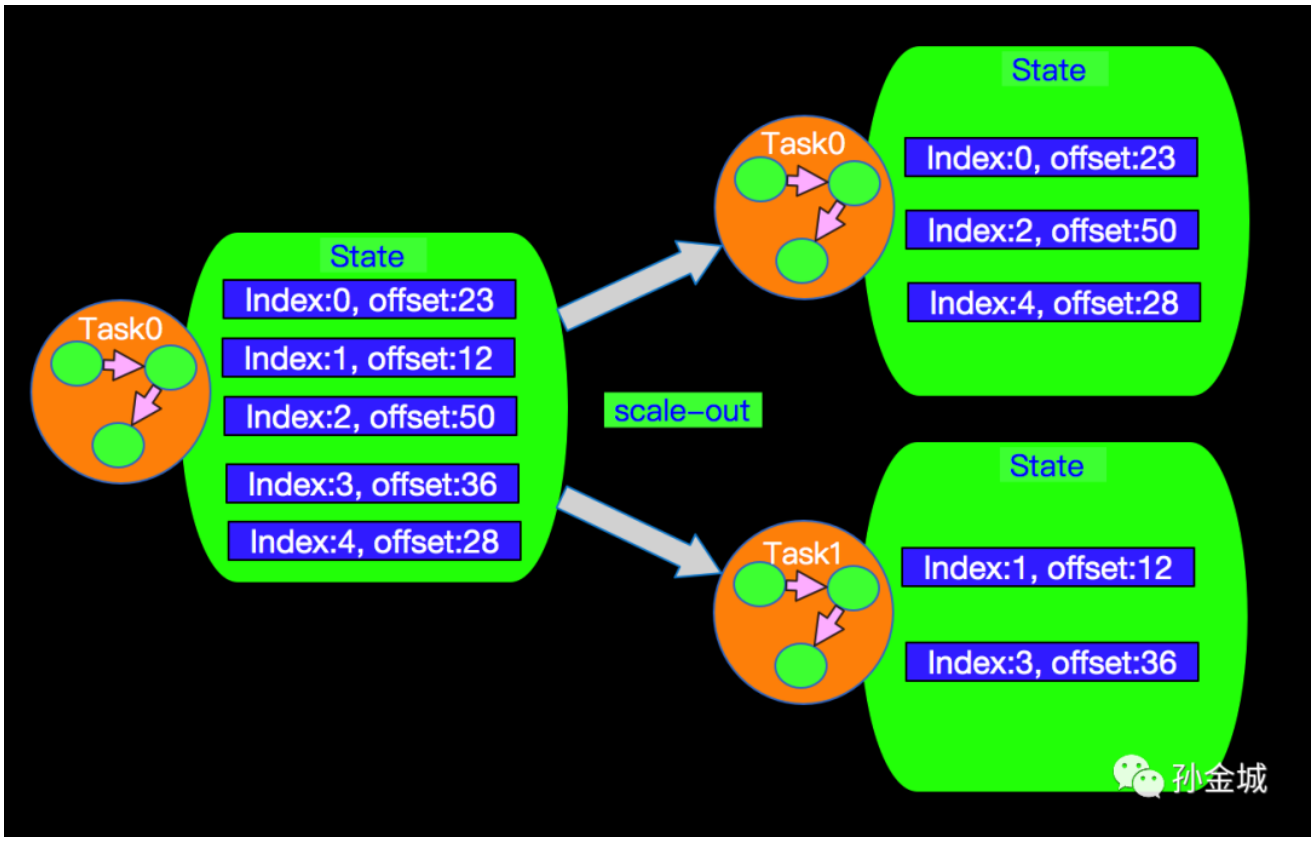


如果我们现在将并发调整为2，那么我们5个分区的State将会在2个独立的任务（线程）中进行维

护，在内部实现中我们有如下算法进行分配每个Task所处理和维护partition的State信息，如下：

```
List<Integer> assignedPartitions = new LinkedList<>();
for (int i = 0; i < partitions; i++) {
  if (i % consumerCount == consumerIndex) {
    assignedPartitions.add(i);
  }
}
```

这个求mod的算法，决定了每个并发所处理和维护partition的State信息，针对我们当前的case具体的存储情况如下：



那么到现在我们发现上面扩容后State得以很好的分配得益于OperatorState采用了List<T>的数据结构的设计。另外大家注意一个问题，相信大家已经发现上面分配partition的算法有一个限制，那就是Source的扩容（并发数）是否可以超过Source物理存储的partition数量呢？答案是否定的，不能。目前Flink的做法是提前报错，即使不报错也是资源的浪费，因为超过partition数量的并发永远分配不到待管理的partition。

KeyedState对扩容的处理

对于KeyedState最容易想到的是hash(key) mod parallelism(operator)方式分配state，就和OperatorState一样，这种分配方式大多是情况是恢复的state不是本地已有的state，需要一次网络

拷贝，这种效率比较低，OperatorState采用这种简单的方式进行处理是因为OperatorState的state一般都比较小，网络拉取的成本很小，对于KeyedState往往很大，我们会有更好的选择，在Flink中采用的是Key-Groups方式进行分配。

什么是Key-Groups

Key-Groups 是Flink中对keyed state按照key进行分组分组的方式，每个key-group中会包含N>0个key，一个key-group是State分配的原子单位。在Flink中关于Key-Group的对象是 KeyGroupRange，如下：

```
public class KeyGroupRange implements KeyGroupsList, Serializable {
        ...
        ...
  private final int startKeyGroup;
  private final int endKeyGroup;
        ...
        ...
}
```

KeyGroupRange两个重要的属性就是 startKeyGroup和endKeyGroup，定义了startKeyGroup和endKeyGroup属性后Operator上面的Key-Group的个数也就确定了；

什么决定Key-Groups的个数

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，那么在Flink的内部实现上key-group的数量就是最大并行度的值。

```
GroupRange.of(0, maxParallelism)
```

如何决定key属于哪个Key-Group

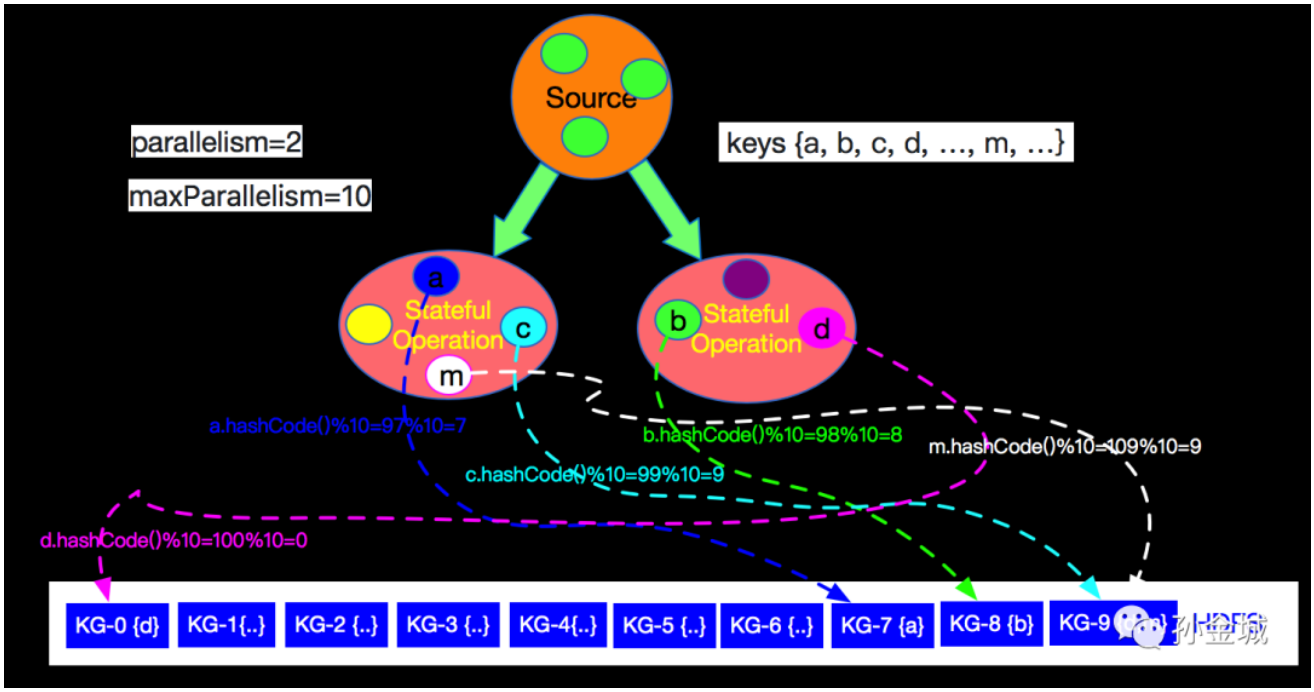
确定好GroupRange之后，如何决定每个Key属于哪个Key-Group呢？我们采取的是取mod的方式，在KeyGroupRangeAssignment中的assignToKeyGroup方法会将key划分到指定的key-group中，如下：

```
public static int assignToKeyGroup(Object key, int maxParallelism) {
      return computeKeyGroupForKeyHash(key.hashCode(), maxParallelism);
}

public static int computeKeyGroupForKeyHash(int keyHash, int maxParallelism
      return HashPartitioner.INSTANCE.partition(keyHash, maxParallelism);
}

@Override
public int partition(T key, int numPartitions) {
      return MathUtils.murmurHash(Objects.hashCode(key)) % numPartitions;
}
```

如上实现我们了解到分配Key到指定的key-group的逻辑是利用key的hashCode和maxParallelism取余操作的来分配的。如下图当parallelism=2,maxParallelism=10的情况下，流上key与key-group的对应关系如下图所示：



如上图key(a)的hashCode是97，与最大并发10取余后是7，被分配到了KG-7中，流上每个event都会分配到KG-0至KG-9其中一个Key-Group中。

每个Operator实例如何获取Key-Groups，了解了Key-Groups概念和如何分配每个Key到指定的Key-Groups之后，我们看看如何计算每个Operator实例所处理的Key-Groups。在KeyGroupRangeAssignment的computeKeyGroupRangeForOperatorIndex方法描述了分配算法：

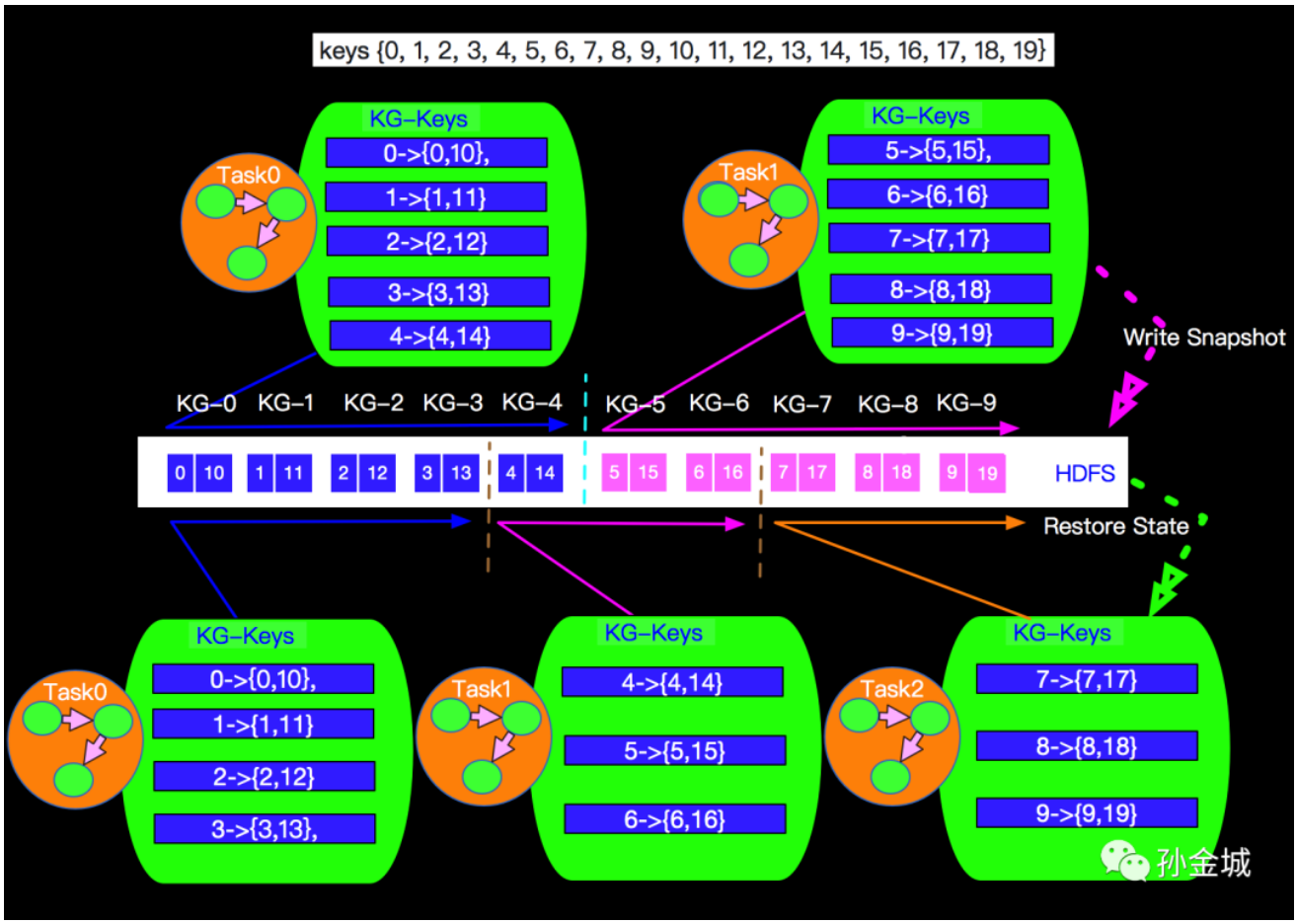
```
public static KeyGroupRange computeKeyGroupRangeForOperatorIndex(
      int maxParallelism,
      int parallelism,
      int operatorIndex) {
  GroupRange splitRange = GroupRange.of(0, maxParallelism).getSplitRange(pa
  int startGroup = splitRange.getStartGroup();
  int endGroup = splitRange.getEndGroup();
   return new KeyGroupRange(startGroup, endGroup - 1);
}

public GroupRange getSplitRange(int numSplits, int splitIndex) {
        ...
        final int numGroupsPerSplit = getNumGroups() / numSplits;
  final int numFatSplits = getNumGroups() % numSplits;

  int startGroupForThisSplit;
  int endGroupForThisSplit;
  if (splitIndex < numFatSplits) {
      startGroupForThisSplit = getStartGroup() + splitIndex * (numGroupsPer
      endGroupForThisSplit =   startGroupForThisSplit + numGroupsPerSplit +
        } else {
      startGroupForThisSplit = getStartGroup() + splitIndex * numGroupsPerS
      endGroupForThisSplit =  startGroupForThisSplit + numGroupsPerSplit;
  }
  if (startGroupForThisSplit >= endGroupForThisSplit) {
    return GroupRange.emptyGroupRange();
  } else {
    return new GroupRange(startGroupForThisSplit, endGroupForThisSplit);
  }
}
```

上面代码的核心逻辑是先计算每个Operator实例至少分配的Key-Group个数，将不能整除的部分N个，平均分给前N个实例。最终每个Operator实例管理的Key-Groups会在GroupRange中表示，本质是一个区间值；下面我们就上图的case，说明一下如何进行分配以及扩容后如何重新分配。

假设上面的Stateful Operation节点的最大并行度maxParallelism的值是10，也就是我们一共有10个Key-Group，当我们并发是2的时候和并发是3的时候分配的情况如下图：



如上算法我们发现在进行扩容时候，大部分state还是落到本地的，如Task0只有KG-4被分出去，其他的还是保持在本地。同时我们也发现，一个job如果修改了maxParallelism的值那么会直接影响到Key-Groups的数量和key的分配，也会打乱所有的Key-Group的分配，目前在Flink系统中统一将maxParallelism的默认值调整到4096，最大程度的避免无法扩容的情况发生。

小结

本篇简单介绍了Flink中State的概念，并重点介绍了OperatorState和KeyedState在扩容时候的处理方式。Flink State是支撑Flink中failover，增量计算，Window等重要机制和功能的核心设施。后续介绍failover，增量计算，Window等相关篇章中也会涉及State的利用，当涉及到本篇没有覆盖的内容时候再补充介绍。

订阅号&知识星球【免费】

分享是最好的享受，予人成功是最大的成功，一个人最大的开心不源于自己会什么，而源于能让别人擅长什么，无欲无求，但予人所求！

More about Me...

我坚信：

"致虚极，守静笃。万物并作，吾以观其复”。“虚”和“静”是心灵的本初的状态，也应该是一种常态，看到新芽不惊，看到落叶不哀，静观万物的循环往复，通晓自然之理，体悟自然之道。"