Flink StateBackend (3) - FsStateBackend

FsStateBackend 可能是大部分公司里最常用的一种 StateBackend 了。

特点及用途

FsStateBackend，简而言之，就是讲状态存储在内存中，用户操作状态，即等于直接操作内存中的对象，没有磁盘开销，没有序列化开销。使用这个 StateBackend 就是一个字：快。当然，这种粗暴的设计必然有其缺陷：

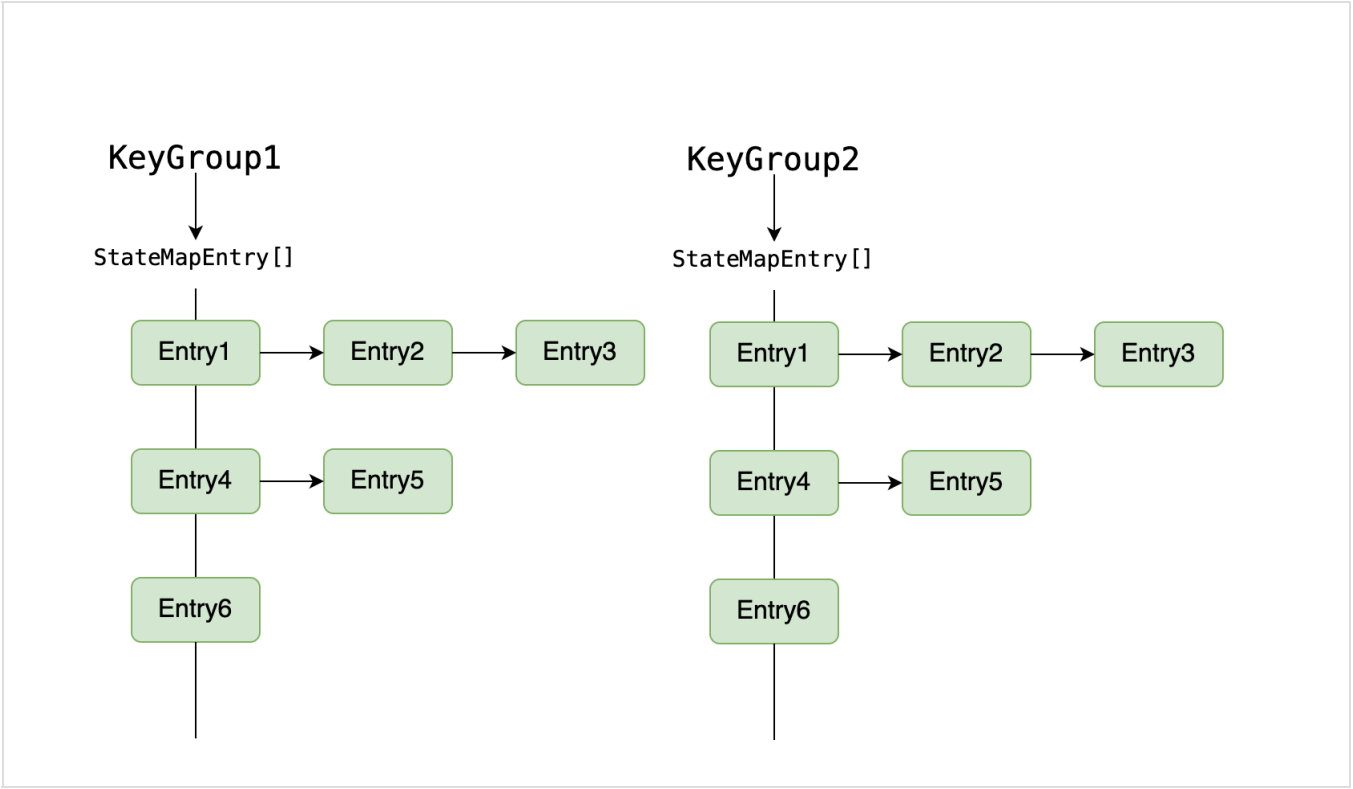
- 状态大小有限制，不能超过 JVM Heap 的上限，一旦有突增流量，立刻 OOM 退出
- GC 的烦恼，如果状态对应的对象不断创建，销毁，那么 GC 问题会让作业基本不可用

实际生产环境中，对于某些稍微复杂一些的场景（比如 Join），如果作业的 TM 内存超过 10G，那么 GC 问题会很明显。那么什么情况下比较适合使用这个 StateBackend 呢?

我推荐一般情况下，只要状态中不存储明细数据，都是可以选用这个 StateBackend 的，比如计算每十分钟的文章阅读数，或者利用 Bitmap 计算每十分钟的 uv 等。一般业务场景下，key 的数量都是可控的，比如文章数、用户数，不太可能出现数量级的暴涨，而对于明细数据，一旦上游出现脏数据、数据倾斜、黑产用户，对于单个 Key 的状态大小都会变成数量级的增长。

状态存储形式

先来一个示意图：



没错，在 Flink 内部采取的结构就是数组 + 链表，和 Java 1.7 中的 HashMap 类似。再看 StateMap 内部提供的接口，也是如下：

```
public abstract S get(K key, N namespace);

public abstract boolean containsKey(K key, N namespace);

public abstract void put(K key, N namespace, S state);

public abstract S putAndGetOld(K key, N namespace, S state);

public abstract void remove(K key, N namespace);

public abstract S removeAndGetOld(K key, N namespace);
```

可以看出，StateMap 以 Key 和 Namespace 作为 Map 的 key，state 作为 Map 的 value。而对于每个 Task 的 StateBackend，采用了 KeyGroup 作为一级索引。

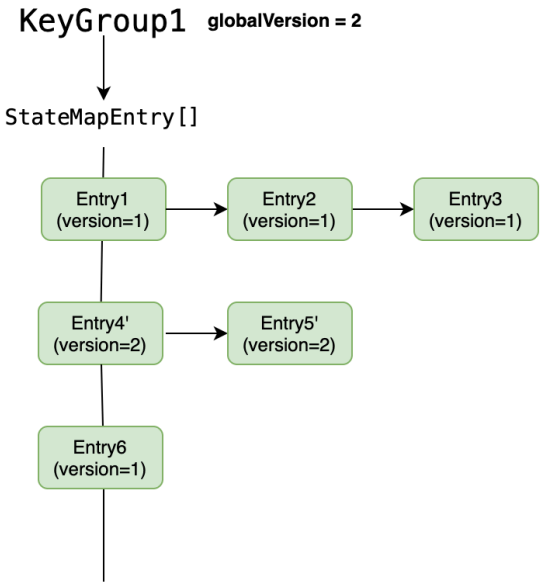
快照机制

对于 FsStateBackend 而言，快照机制分为两种，synchronous 和 asynchronous。synchronous 就是同步进行快照，触发 checkpoint 时将所有 KeyGroups 下的所有 stateMap 给同步到 Hdfs 上。

asynchronous 即异步快照，这里是使用了一个 Copy-On-Write 的机制来保证异步将 stateMap 中的内容同步到 Hdfs 上。Asynchronous 中也分为同步过程和异步过程，其中同步过程主要做两个操作：

1. 对 StateMapEntry[] 数组做一份 snapshot，这里仅仅是对象引用的 snapshot，所以非常轻量。
2. 增加 globalVersion，这样方便后续去区分 Entry 是否是 copy 后的值。

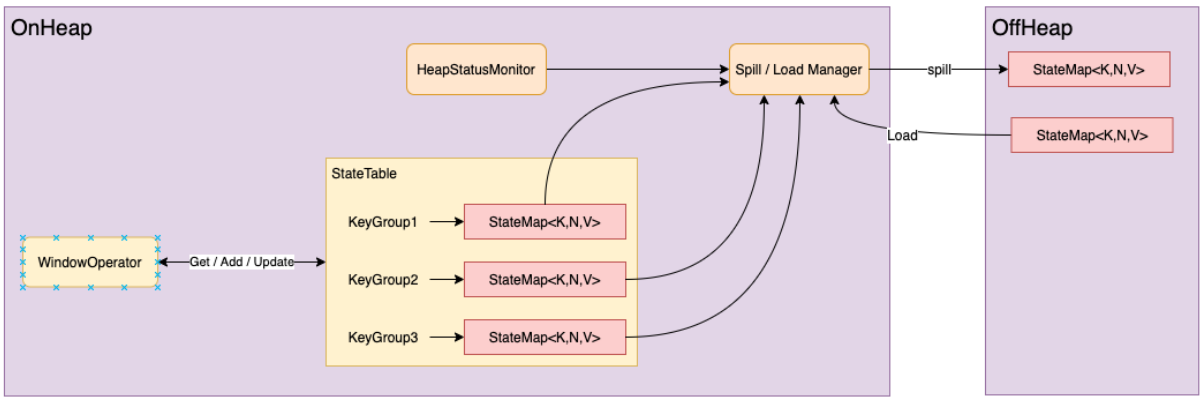
还是以上面的示意图为例，当触发 checkpoint 时，将 globalVersion 从 1 增加至 2（同步操作），然后开始对 stateMap 进行异步写入 Hdfs，如果异步写入的过程中，需要 update Entry5，示意如图：



在这里直接将 Entry5 寻址过程中经过的 Entry 都 copy 了一份，而由于在 asynchronous 已经对 StateMapEntry[] 数组上的对象引用做了 snapshot，所以并不影响我们异步写 Hdfs 的过程。

其他

快照恢复的过程，可以简单等同于将 Hdfs 文件上的内容重新读取出来生成 StateMap。关于这个 StateBackend，因为缺点很明显，后续社区也有改善的方案[1]。方案思路也比较简单，假设所有 KeyGroup 下有些 KeyGroup 属于 hot data，有些属于 cold data，当内存装不下时，将 cold data 持久化到磁盘上。示意图如下：



这里引入了两个组件，HeapStatusMonitor 和 Spill / Load Manager，其中

- HeapStatusMonitor: 由于内存中的对象无法直接预估其大小，所以只能采用近似的方式：
  - 判断 Heap Usage
  - 判断 GC Pause
- Spill / Load Manager:
  - 统计每个 KeyGroup 下的 size 和 request rate
  - 以 KeyGroup 为粒度 Spill / Load StateMap 的数据

个人认为这个设计的主要难点可能是以什么粒度来区分冷热数据，最终使用 KeyGroup 来区分，可能对于默认的 65536 个 KeyGroup 来说，这里的粒度还是有点过粗了，不过可以通过调整 maxParallelism 来辅助调整持久化的粒度，总体来说，是 HeapKeyedStateBackend 的替代版本，完全可以通过参数控制来讲 SpillableHeapKeyedStateBackend 来达到和 HeapKeyedStateBackend 一样的功能。

引用

1. FLIP-50: Spillable Heap State Backend