Flink StateBackend (4) - RocksDBStateBackend

RocksDBStateBackend 是 Flink 中用来存储大状态的 StateBackend。

特点及用途

上一篇讲到，如果状态过大，使用 FsStateBackend 很容易受到 GC 的影响，对于这种场景，使用 RocksDBStateBackend 就是一种更明智的选择。不过 RocksDB 的缺陷也非常明显：

- 相比 FsStateBackend 需要更多的开销
  - Java 对象需要序列化和反序列化
  - 磁盘 IO
  - RocksDB 的 compaction
- 对于 Read-Modify-Write 的场景不够友好（下面会细讲一下）

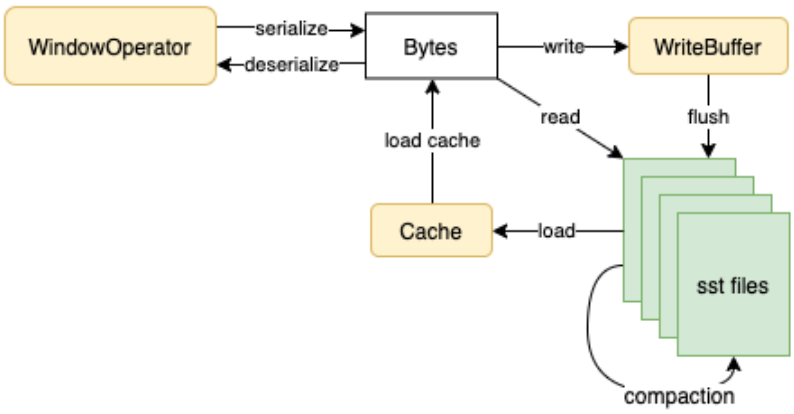
优势就是可以存储更大的状态，比较常见的场景是广告业务中 click 和 impression 的匹配，因为需要存储明细数据，所以状态相比存储聚合值的状态要大上好几个数量级。通常，使用 RocksDBStateBackend 对磁盘性能要求比较高，如果没有 SSD，建议还是不要使用 RocksDBStateBackend 了。（使用 HDD 和 RocksDBStateBackend 的性能可以差上百倍）

状态存储形式

RocksDBStateBackend 底层使用 RocksDB 来存储状态，Task 在初始化时，会在本地磁盘上创建一个 RocksDB 实例。在 RocksDB 实例中，Key 是任务中 Record 的 Key + Namespace，Column Family 是 state 的唯一标识符。假如统计每个用户的 click 和 impression 两个状态，在 RocksDB 中的形式就是这样。（在这里忽略了 namespace）

Key	click (Column Family)	impression (Column Family)
user1	[1,2,3,4,5]	[10,11]
user2	[1,2]	[3,4]
user3	[4,5]	[8,9]

所有对 State 的操作可以简单理解为对 RocksDB 实例的操作，刚才讲到 RocksDBStateBackend 对于 Read-Modify-Write 操作不太友好，这也是由 RocksDB 本身的 append-only 特性导致的。如下图所示，数据在读取后，经过更新写入 write-buffer，然后通过 flush 持久化到磁盘上，下次读取时又重新从磁盘中 load 起来，这样相当于一直在刷 rocksdb 的 cache。



优化

鉴于 RocksDBStateBackend 的性能比较堪忧，从发布以来也进行了各种的优化，在这里列举几个我们经常用到的。

ListState 使用 RocksDB 的 merge 特性

RocksDB 在中引入了 Merge Operation 来避免 Read-Modify-Write 的开销，其原理是给 Read-Modify-Write 的操作提供一个回调的函数，然后直接将回调操作写入到 RocksDB 中，等最终 Read 操作时，合并原始值 + 后续所有的回调操作，返回结果。假设我们需要在 List 中进行 add 操作，不做优化的伪代码如下：

```
public void add(T element) {
    List<T> list = read from rocksdb and deserialize
    list.add(element)
    serialize list and write into rocksdb
}
```

这就是一个典型的 Read-Modify-Write 的操作，而实际优化后的代码只需要调用 merge 函数接口就行了。（因为 merge 操作并没有开放 java api，所以 Flink 的做法是调用 JNI 来实现这个功能）

利用 RocksDB compaction 清理过期数据

Flink 本身提供了三种策略来清理过期数据：

- FULL_STATE_SCAN_SNAPSHOT: 在 full snapshot 时，因为做全量快照时本身需要遍历一遍所有数据，将状态中中的过期数据清除。因为 RocksDBStateBackend 基本都是用增量，所以在这里不适用。
- INCREMENTAL_CLEANUP: 在 touch 状态时，每次增量读取若干条数据，如果 expire 就清理。
- ROCKSDB_COMPACTION_FILTER: 这里也是充分利用了 RocksDB 的 compaction filter 特性，因为 RocksDB 内部本身就要做 compaction，遍历所有数据，Flink 在这里用 JNI 实现了这一功能。

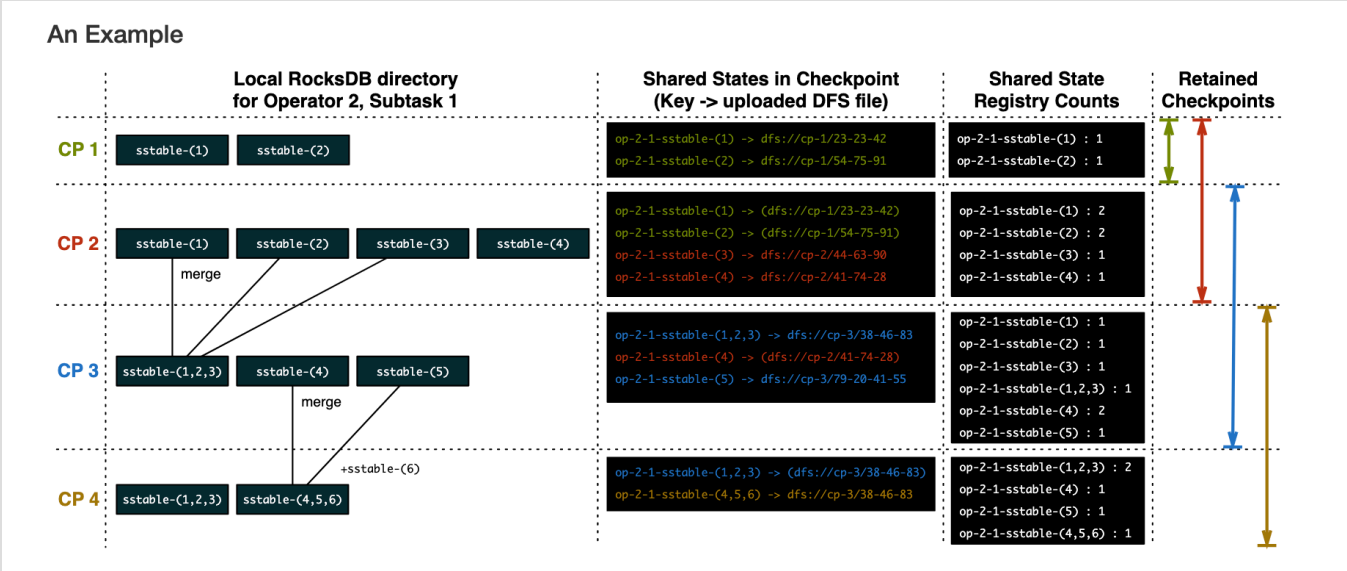
其他

其他的优化有增量快照，MapState 的 cache 优化等，在这里不一一列举了。

快照和恢复

增量快照

因为 RocksDB 本身是 append-only 的，所以对 RocksDB 的操作会被持久化到一系列的 sst 文件中。根据这个特性，我们可以在每次快照时，将 write-buffer 中的数据持久化到磁盘上并记录一个最新的 sst 文件标号，等到下次快照时就知道相对于上次快照新增了哪些文件。这里存在一个问题，就是当发生 compaction 时，历史的文件如何清除的问题。详细原理见：incremental-checkpointing。



实际上，Flink 采用了引用计数法来做历史文件的清除，如上图所示，展示了一个 retained-checkpoints=2 的示例：

- CP1 完成后，sst-1 和 sst-2 的计数都是 1
- CP2 完成后，sst-1 和 sst-2 的计数为 2，sst-3 和 sst-4 由于是新增的文件，所以是 1
- CP3 完成后，sst-1 到 sst-3 发生 compaction 产生了新文件，由于最多只保留两个 checkpoint，所以 CP1 被丢弃，对应的 sst-1 和 sst-2 解除引用，sst-1 和 sst-2 计数减 1，sst-3 因为在这次 checkpoint 没有出现，所以计数不变
- CP4 完成后，CP2 被丢弃，sst-1 和 sst-2 的计数再次减 1，说明没有 checkpoint 再依赖 sst-1 和 sst-2 文件，可以清除。

恢复

和 FsStateBackend 恢复时将所有 state 直接 load 到内存中的 eager 模式不同，RocksDBStateBackend 的恢复策略是 lazy 模式，先将 sst 文件从 Hdfs 端拷贝回来，然后在任务运行时去加载状态。在这里涉及到一个扩缩容后 sst 文件重新分配的问题，在这里也没有什么好办法，因为 KeyGroupRange 发生了变化，比如以前 KeyGroupRange 是 (4-6) 的现在 KeyGroupRange 是 (3-7)，所以除了自己之前的 sst 文件，我们还需要读取另外两个 Task 的 RocksDB 数据才能把状态恢复回来。

Flink 这里的做法是：

1. 生成一个临时目录 tmp_dir，生成一个新的 RocksDB 实例 newDB
2. 总共需要读取三个 Task 的 RocksDB 文件，在这里生成三个 RocksDB 实例，采用 rocksdb 的 range 操作读取 KeyGroupRange 在 (3–7) 之间的数据，并采用 newDB 写入到临时目录下
3. 将临时目录的数据移入到正式目录中，作为运行时的状态

可以看到，如果出现扩缩容，RocksDB 的恢复操作也是很昂贵的。

其他

后续社区内关于 RocksDBStateBackend 的改进还有很多，包括：

1. sst 小文件过多，采用合并策略上传到 Hdfs
2. 在 RocksDBStateBackend 上加 cache 来解决 Read–Modify–Write 场景下的问题

引用

- https://flink.apache.org/features/2018/01/30/incremental-checkpointing.html
- https://github.com/facebook/rocksdb