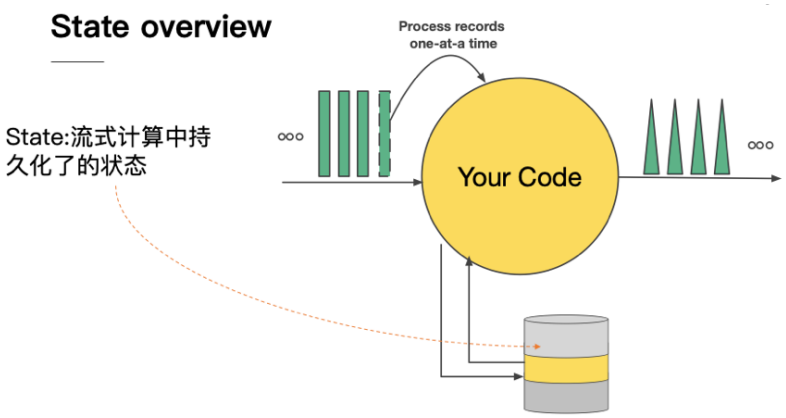
Flink State 最佳实践

本文主要分享与交流 Flink 状态使用过程中的一些经验与心得，当然标题取了“最佳实践”之名，希望文章内容能给读者带去一些干货。本文内容首先是回顾 state 相关概念，并认识和区别不同的 state backend；之后将分别对 state 使用访问以及 checkpoint 容错相关内容进行详细讲解，分享一些经验和心得。

State 概念回顾

我们先回顾一下到底什么是 state，流式计算的数据往往是转瞬即逝， 当然，真实业务场景不可能说所有的数据都是进来之后就走掉，没有任何东西留下来，那么留下来的东西其实就是称之为 state，中文可以翻译成状态。

在下面这个图中，我们的所有的原始数据进入用户代码之后再输出到下游，如果中间涉及到 state 的读写，这些状态会存储在本地的 state backend（可以对标成嵌入式本地 kv 存储）当中。

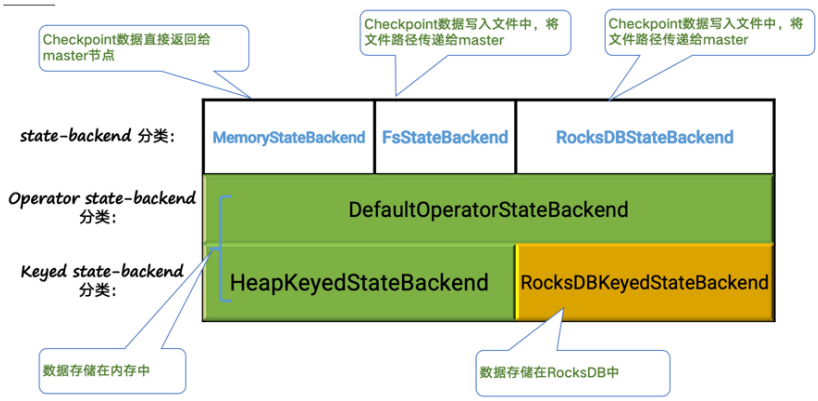


接下来我们会在四个维度来区分两种不同的 state：operator state 以及 keyed state。

1. 是否存在当前处理的 key（current key）：operator state 是没有当前 key 的概念，而 keyed state 的数值总是与一个 current key 对应。
2. 存储对象是否 on heap: 目前 operator state backend 仅有一种 on-heap 的实现；而 keyed state backend 有 on-heap 和 off-heap（RocksDB）的多种实现。
3. 是否需要手动声明快照（snapshot）和恢复 (restore) 方法：operator state 需要手动实现 snapshot 和 restore 方法；而 keyed state 则由 backend 自行实现，对用户透明。
4. 数据大小：一般而言，我们认为 operator state 的数据规模是比较小的；认为 keyed state 规模是相对比较大的。需要注意的是，这是一个经验判断，不是一个绝对的判断区分标准。

StateBackend 的分类

下面这张图对目前广泛使用的三类 state backend 做了区分，其中绿色表示所创建的operator/keyed state backend 是 on-heap 的，黄色则表示是 off-heap 的。



一般而言，在生产中，我们会在 FsStateBackend 和 RocksDBStateBackend 间选择：

- **FsStateBackend**：性能更好；日常存储是在堆内存中，面临着 OOM 的风险，不支持增量 checkpoint。
- **RocksDBStateBackend**：无需担心 OOM 风险，是大部分时候的选择。

RocksDB StateBackend 概览和相关配置讨论

RocksDB 是 Facebook 开源的 LSM 的键值存储数据库，被广泛应用于大数据系统的单机组件中。Flink 的 keyed state 本质上来说就是一个键值对，所以与 RocksDB 的数据模型是吻合的。下图分别是 “window state” 和 “value state” 在 RocksDB 中的存储格式，所有存储的 key，value 均被序列化成 bytes 进行存储。

Window state		Value state	
KeyGroup + Key + Namespace	value	KeyGroup + Key + Namespace	value
(1, K1, Window(10, 20))	v1	(2, K2, VoidNameSpace)	v2
(1, K3, Window(10, 20))	v3	(2, K4, VoidNameSpace)	v4
(1, K5, Window(10, 20))	v5	(2, K6, VoidNameSpace)	v6
...	...	...	...

在 RocksDB 中，每个 state 独享一个 Column Family，而每个 Column family 使用各自独享的 write buffer 和 block cache，上图中的 window state 和 value state实际上分属不同的 column family。

下面介绍一些对 RocksDB 性能比较有影响的参数，并整理了一些相关的推荐配置，至于其他配置项，可以参阅社区相关文档。

状态	建议
state.backend.rocksdb.thread.num	后台 flush 和 compaction 的线程数. 默认值 ‘1‘. 建议调大
state.backend.rocksdb.writebuffer.count	每个 column family 的 write buffer 数目，默认值 ‘2‘. 如果有需要可以适当调大
state.backend.rocksdb.writebuffer.size	每个 write buffer 的 size，默认值‘64MB‘. 对于写频繁的场景，建议调大

state.backend.rocksdb.block.cache-size	每个 column family 的 block cache大小，默认值‘8MB’，如果存在重复读的场景，建议调大

State best practice：一些使用 state 的心得

Operator state 使用建议

■ 慎重使用长 list

下图展示的是目前 task 端 operator state 在执行完 checkpoint 返回给 job master 端的 StateMetaInfo 的代码片段。

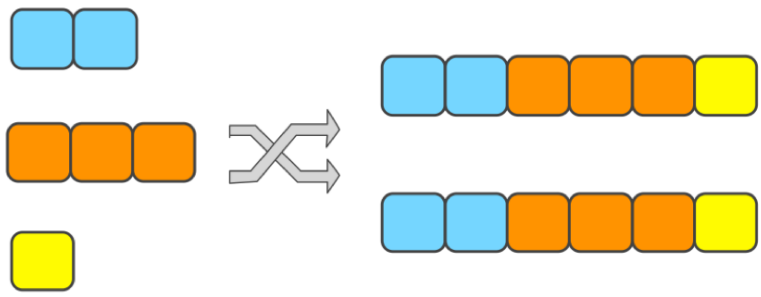
```
/**
 * Meta information about the operator state handle.
 */
class StateMetaInfo implements Serializable {

    private static final long serialVersionUID = 3593817615858941166L;

    private final long[] offsets;
    private final Mode distributionMode;

    public StateMetaInfo(long[] offsets, Mode distributionMode) {
        this.offsets = Preconditions.checkNotNull(offsets);
        this.distributionMode = Preconditions.checkNotNull(distributionMode);
    }
```

由于 operator state 没有 key group 的概念，所以为了实现改并发恢复的功能，需要对 operator state 中的每一个序列化后的元素存储一个位置偏移 offset，也就是构成了上图红框中的 offset 数组。

那么如果你的 operator state 中的 list 长度达到一定规模时，这个 offset 数组就可能会有几十 MB 的规模，关键这个数组是会返回给 job master，当 operator 的并发数目很大时，很容易触发 job master 的内存超用问题。我们遇到过用户把 operator state 当做黑名单存储，结果这个黑名单规模很大，导致一旦开始执行 checkpoint，job master 就会因为收到 task 发来的“巨大”的 offset 数组，而内存不断增长直到超用无法正常响应。

■ 正确使用 UnionListState

union list state 目前被广泛使用在 kafka connector 中，不过可能用户日常开发中较少遇到，他的语义是从检查点恢复之后每个并发 task 内拿到的是原先所有operator 上的 state，如下图所示：



kafka connector 使用该功能，为的是从检查点恢复时，可以拿到之前的全局信息，如果用户需要使用该功能，需要切记恢复的 task 只取其中的一部分进行处理和用于下一次 snapshot，否则有可能随着作业不断的重启而导致 state 规模不断增长。

Keyed state 使用建议

■ 如何正确清空当前的 state

state.clear() 实际上只能清理当前 key 对应的 value 值，如果想要清空整个 state，需要借助于 applyToAllKeys 方法，具体代码片段如下：

```
// clear state via applyToAllKeys().
backend.applyToAllKeys(VoidNamespace.INSTANCE, VoidNamespaceSerializer.INSTANCE, listStateDescriptor,
    new KeyedStateFunction<Integer, ListState<String>>() {
        @Override
        public void process(Integer key, ListState<String> state) throws Exception {
            state.clear();
        }
    });
```

如果你的需求中只是对 state 有过期需求，借助于 state TTL 功能来清理会是一个性能更好的方案。

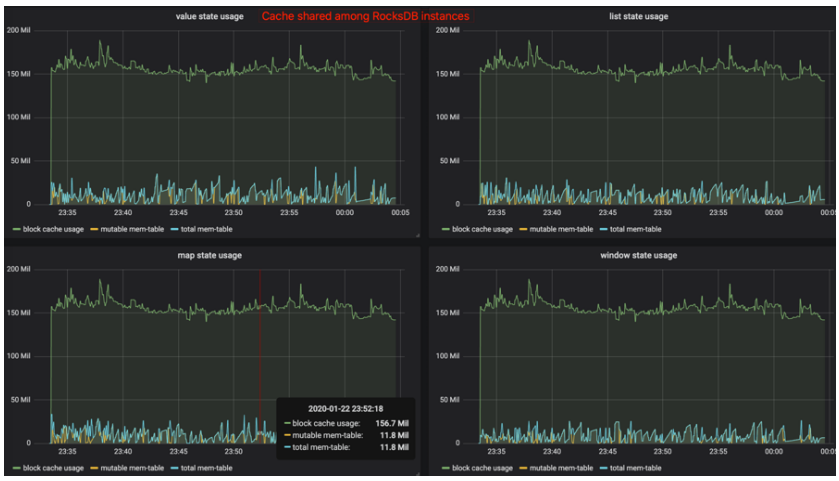
■ RocksDB 中考虑 value 值很大的极限场景

受限于 JNI bridge API 的限制，单个 value 只支持 2^31 bytes 大小，如果存在很极限的情况，可以考虑使用 MapState 来替代 ListState 或者 ValueState，因为RocksDB 的 map state 并不是将整个 map 作为 value 进行存储，而是将 map 中的一个条目作为键值对进行存储。

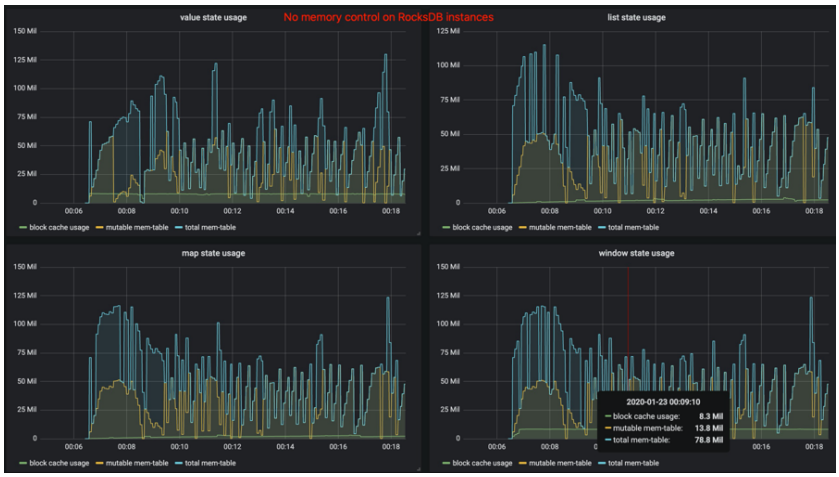
■ 如何知道当前 RocksDB 的运行情况

比较直观的方式是打开 RocksDB 的 native metrics ，在默认使用 Flink managed memory 方式的情况下，state.backend.rocksdb.metrics.block-cache-usage ，state.backend.rocksdb.metrics.mem-table-flush-pending，state.backend.rocksdb.metrics.num-running-compactions 以及 state.backend.rocksdb.metrics.num-running-flushes 是比较重要的相关 metrics。

下面这张图是 Flink-1.10 之后，打开相关 metrics 的示例图：



而下面这张是 Flink-1.10 之前或者关闭 state.backend.rocksdb.memory.managed 的效果：



■ 容器内运行的 RocksDB 的内存超用问题

在 Flink-1.10 之前，由于一个 state 独占若干 write buffer 和一块 block cache，所以我们会建议用户不要在一个 operator 内创建过多的 state，否则需要考虑到相应的额外内存使用量，否则容易造成在容器内运行时，相关进程被容器环境所杀。对于用户来说，需要考虑一个 slot 内有多少 RocksDB 实例在运行，一个 RocksDB 中有多少 state，整体的计算规则就很复杂，很难真得落地实施。

Flink-1.10 之后，由于引入了 RocksDB 的内存托管机制，在绝大部分情况下， RocksDB 的这一部分 native 内存是可控的，不过受限于 RocksDB 的相关 cache 实现限制（这里暂不展开，后续会有文章讨论），在某些场景下，无法做到完美控制，这时候建议打开上文提到的 native metrics，观察相关 block cache 内存使用是否存在超用情况，可以将相关内存添加到 taskmanager.memory.task.off-heap.size 中，使得 Flink 有更多的空间给 native 内存使用。

一些使用 checkpoint 的使用建议

■ Checkpoint 间隔不要太短

虽然理论上 Flink 支持很短的 checkpoint 间隔，但是在实际生产中，过短的间隔对于底层分布式文件系统而言，会带来很大的压力。另一方面，由于检查点的语义，所以实际上 Flink 作业处理 record 与执行 checkpoint 存在互斥锁，过于频繁的 checkpoint，可能会影响整体的性能。当然，这个建议的出发点是底层分布式文件系统的压力考虑。

■ 合理设置超时时间

默认的超时时间是 10min，如果 state 规模大，则需要合理配置。最坏情况是分布式地创建速度大于单点（job master 端）的删除速度，导致整体存储集群可用空间压力较大。建议当检查点频繁因为超时而失败时，增大超时时间。