State Backends

当需要对具体的某一种 State 做 Checkpoint 时，此时就需要具体的状态后端存储，刚好 Flink 内置提供了不同的状态后端存储，用于指定状态的存储方式和位置。状态可以存储在 Java 堆内存中或者堆外，在 Flink 安装路径下 conf 目录中的 flink-conf.yaml 配置文件中也有状态后端存储相关的配置，为此在 Flink 源码中还特有一个 CheckpointingOptions 类来控制 state 存储的相关配置，该类中有如下配置：

- state.backend: 用于存储和进行状态 Checkpoint 的状态后端存储方式，无默认值
- state.checkpoints.num-retained: 要保留的已完成 Checkpoint 的最大数量，默认值为 1
- state.backend.async: 状态后端是否使用异步快照方法，默认值为 true
- state.backend.incremental: 状态后端是否创建增量检查点，默认值为 false
- state.backend.local-recovery: 状态后端配置本地恢复，默认情况下，本地恢复被禁用
- taskmanager.state.local.root-dirs: 定义存储本地恢复的基于文件的状态的目录
- state.savepoints.dir: 存储 savepoints 的目录
- state.checkpoints.dir: 存储 Checkpoint 的数据文件和元数据
- state.backend.fs.memory-threshold: 状态数据文件的最小大小，默认值是 1024

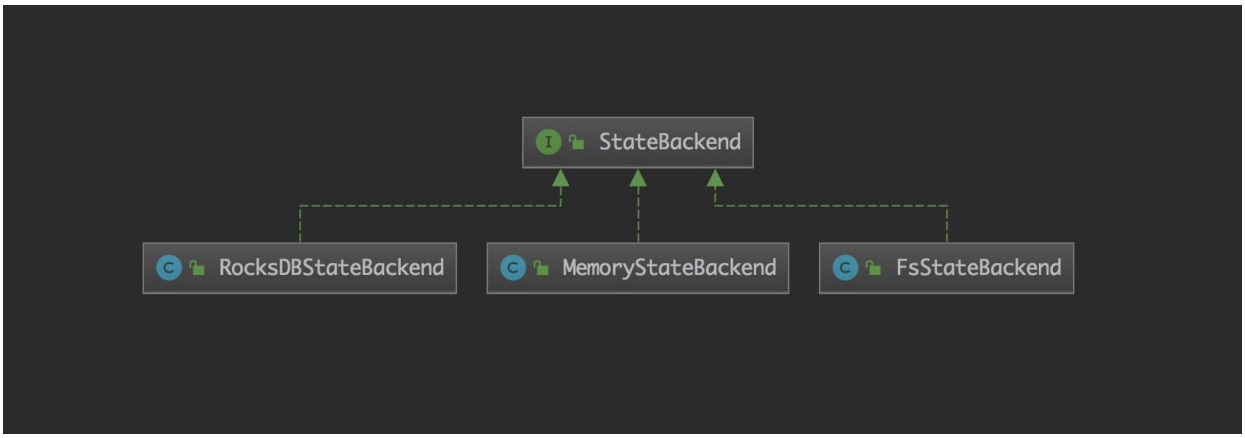
虽然配置这么多，但是，Flink 还支持基于每个 Job 单独设置状态后端存储，方法如下：

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

env.setStateBackend(new MemoryStateBackend());   //设置堆内存存储

//env.setStateBackend(new FsStateBackend(checkpointDir, asyncCheckpoints));
  //设置文件存储

//env.setStateBackend(new RocksDBStateBackend(checkpointDir,
incrementalCheckpoints));  //设置 RocksDB 存储
```



上面三种方式取一种就好了。但是有三种方式，我们该如何去挑选用哪种去存储状态呢？下面讲讲这三种的特点以及该如何选择。

MemoryStateBackend

如果 Job 没有配置指定状态后端存储的话，就会默认采取 MemoryStateBackend 策略。如果你细心的话，可以从你的 Job 中看到类似日志如下：

```
2019-04-28 00:16:41.892 [Sink: zhisheng (1/4)] INFO
org.apache.flink.streaming.runtime.tasks.StreamTask  - No state backend has
been configured, using default (Memory / JobManager) MemoryStateBackend
(data in heap memory / checkpoints to JobManager) (checkpoints: 'null',
savepoints: 'null', asynchronous: TRUE, maxStateSize: 5242880)
```

上面日志的意思就是说如果没有配置任何状态存储，使用默认的 MemoryStateBackend 策略，这种状态后端存储把数据以内部对象的形式保存在 TaskManagers 的内存（JVM 堆）中，当应用程序触发 Checkpoint 时，会将此时的状态进行快照然后存储在 JobManager 的内存中。因为状态是存储在内存中的，所以这种情况会有点限制，比如：

- 不太适合在生产环境中使用，仅用于本地测试的情况较多，主要适用于状态很小的 Job，因为它会将状态最终存储在 JobManager 中，如果状态较大的话，那么会使得 JobManager 的内存比较紧张，从而导致 JobManager 会出现 OOM 等问题，然后造成连锁反应使所有的 Job 都挂掉，所以 Job 的状态与之前的 Checkpoint 的数据所占的内存要小于 JobManager 的内存。
- 每个单独的状态大小不能超过最大的 DEFAULT_MAX_STATE_SIZE(5MB)，可以通过构造 MemoryStateBackend 参数传入不同大小的 maxStateSize。
- Job 的操作符状态和 keyed 状态加起来都不要超过 RPC 系统的默认配置 10 MB，虽然可以修改该配置，但是不建议去修改。

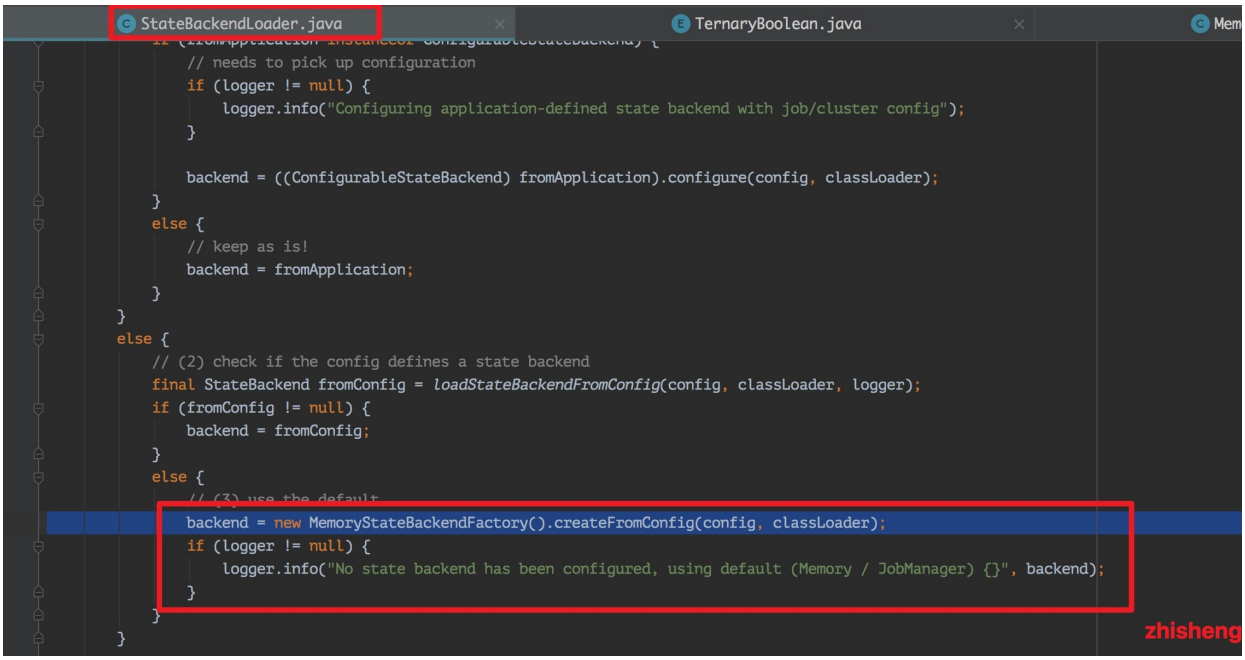
另外就是 MemoryStateBackend 支持配置是否是异步快照还是同步快照，它有一个字段 asynchronousSnapshots 来表示，可选值有：

- TRUE（true 代表使用异步的快照，这样可以避免因快照而导致数据流处理出现阻塞等问题）
- FALSE（同步）
- UNDEFINED（默认值）

在构造 MemoryStateBackend 的默认函数时是使用的 UNDEFINED，而不是异步：

```
public MemoryStateBackend() {
    this(null, null, DEFAULT_MAX_STATE_SIZE, TernaryBoolean.UNDEFINED);//使
用的是 UNDEFINED
}
```

网上有人说默认是异步的，这里给大家解释清楚一下，从上面的那条日志打印的确实也是表示异步，但是前提是你对 State 无任何操作，我跟了下源码，当你没有配置任何的 state 时，它是会在 StateBackendLoader 类中通过 MemoryStateBackendFactory 来创建的 state 的。



继续跟进 MemoryStateBackendFactory 可以发现他这里创建了一个 MemoryStateBackend 实例并通过 configure 方法进行配置，大概流程代码是：

```
//MemoryStateBackendFactory 类
public MemoryStateBackend createFromConfig(Configuration config,
ClassLoader classLoader) {
    return new MemoryStateBackend().configure(config, classLoader);
}

//MemoryStateBackend 类中的 config 方法
public MemoryStateBackend configure(Configuration config, ClassLoader
classLoader) {
    return new MemoryStateBackend(this, config, classLoader);
}

//私有的构造方法
private MemoryStateBackend(MemoryStateBackend original, Configuration
configuration, ClassLoader classLoader) {
    ...
    this.asynchronousSnapshots =
original.asynchronousSnapshots.resolveUndefined(

 configuration.getBoolean(CheckpointingOptions.ASYNC_SNAPSHOTS));
}

//根据 CheckpointingOptions 类中的 ASYNC_SNAPSHOTS 参数进行设置的
public static final ConfigOption<Boolean> ASYNC_SNAPSHOTS = ConfigOptions
        .key("state.backend.async")
        .defaultValue(true) //默认值就是 true，代表异步
        .withDescription(...)
```

可以发现最终是通过读取 `state.backend.async` 参数的默认值（true）来配置是否要异步的进行快照，但是如果你手动配置 MemoryStateBackend 的话，利用无参数的构造方法，那么就不是默认异步，如果想使用异步的话，需要利用下面这个构造函数（需要传入一个 boolean 值，true 代表异步，false 代表同步）：

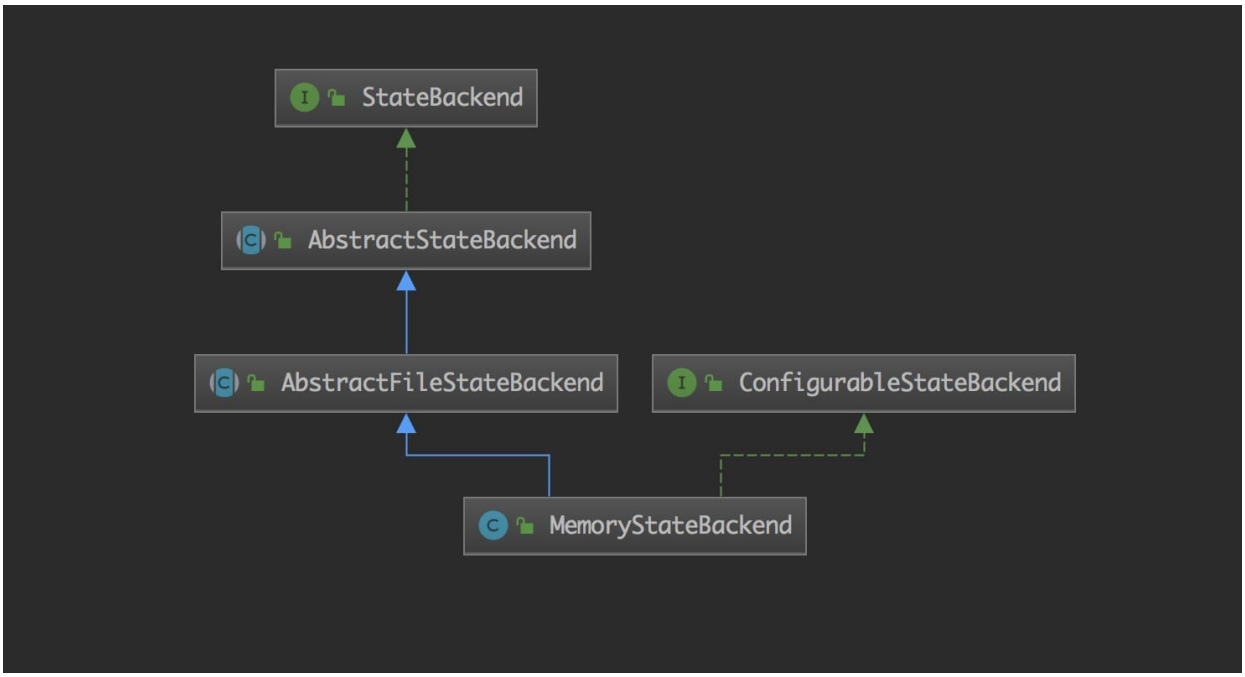
```
public MemoryStateBackend(boolean asynchronousSnapshots) {
    this(null, null, DEFAULT_MAX_STATE_SIZE,
TernaryBoolean.fromBoolean(asynchronousSnapshots));
}
```

如果你再细看了这个 MemoryStateBackend 类的话，那么你可能会发现这个构造函数：

```
public MemoryStateBackend(@Nullable String checkpointPath, @Nullable String
savepointPath) {
    this(checkpointPath, savepointPath, DEFAULT_MAX_STATE_SIZE,
TernaryBoolean.UNDEFINED);//需要你传入 checkpointPath 和 savepointPath
}
```

这个也是用来创建一个 MemoryStateBackend 的，它需要传入的参数是两个路径（checkpointPath、savepointPath），其中 checkpointPath 是写入 checkpoint 元数据的路径，savepointPath 是写入 savepoint 的路径。

这个来看看 MemoryStateBackend 的继承关系图可以更明确的知道它是继承自 AbstractFileStateBackend，然后 AbstractFileStateBackend 这个抽象类就是为了能够将状态存储中的数据或者元数据进行文件存储的。



所以 FsStateBackend 和 MemoryStateBackend 都会继承该类。

FsStateBackend

这种状态后端存储也是将工作状态存储在 TaskManagers 中的内存（JVM 堆）中，但是 Checkpoint 的时候，它和 MemoryStateBackend 不一样，它是将状态存储在文件（可以是本地文件，也可以是 HDFS）中，这个文件具体是哪种需要配置，比如："hdfs://namenode:40010/flink/checkpoints" 或 "file://flink/checkpoints" (通常使用 HDFS 比较多，如果是使用本地文件，可能会造成 Job 恢复的时候找不到之前的 checkkpoint，因为 Job 重启后如果由调度器重新分配在不同的机器的 TaskManager 执行时就会导致这个问题，所以还是建议使用 HDFS 或者其他的分布式文件系统)。

同样 FsStateBackend 也是支持通过 asynchronousSnapshots 字段来控制是使用异步还是同步来进行 Checkpoint 的，异步可以避免在状态 checkpoint 时阻塞数据流的处理，然后还有一点的就是在 FsStateBackend 有个参数 fileStateThreshold，如果状态大小比 MAX_FILE_STATE_THRESHOLD（1MB） 小的话，那么会将状态数据直接存储在 meta data 文件中，而不是存储在配置的文件中（避免出现很小的状态文件），如果该值为 "-1" 表示尚未配置，在这种情况下会使用默认值（1024，该默认值可以通过 `state.backend.fs.memory-threshold` 来配置）。

那么我们该什么时候使用 FsStateBackend 呢?

- 如果你要处理大状态，长窗口等有状态的任务，那么 FsStateBackend 就比较适合
- 使用分布式文件系统，如 HDFS 等，这样 failover 时 Job 的状态可以恢复

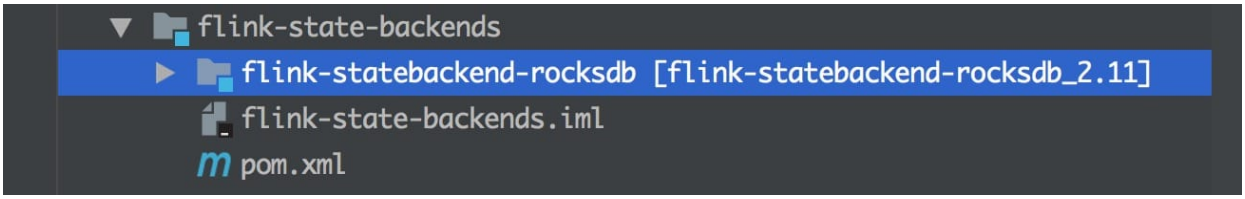
使用 FsStateBackend 需要注意的地方有什么呢?

- 工作状态仍然是存储在 TaskManagers 中的内存中，虽然在 Checkpoint 的时候会存在文件中，所以还是得注意这个状态要保证不超过 TaskManager 的内存

RocksDBStateBackend

RocksDBStateBackend 和上面两种都有点不一样，RocksDB 是一种嵌入式的本地数据库，它会在本地文件系统中维护状态，KeyedStateBackend 等会直接写入本地 RocksDB 中，它还需要配置一个文件系统（一般是 HDFS），比如 `hdfs://namenode:40010/flink/checkpoints` ，当触发 checkpoint 的时候，会把整个 RocksDB 数据库复制到配置的文件系统中去，当 failover 时从文件系统中将数据恢复到本地。

在 Flink 源码中，你也可以看见专门有一个 module 是 flink-statebackend-rocksdb 来放在 flink-state-backends 下面：



足以证明了官方其实也是推荐使用 RocksDB 来作为状态的后端存储，为什么呢：

- state 直接存放在 RocksDB 中，不需要存在内存中，这样就可以减少 TaskManager 的内存压力，如果是存内存的话大状态的情况下会导致 GC 次数比较多，同时还能在 Checkpoint 时将状态持久化到远端的文件系统，那么就比较适合在生产环境中使用
- RocksDB 本身支持 Checkpoint 功能
- RocksDBStateBackend 支持增量的 Checkpoint，在 RocksDBStateBackend 中有一个字段 enableIncrementalCheckpointing 来确认是否开启增量的 checkpoint，默认是不开启的，在 CheckpointingOptions 类中有个 state.backend.incremental 参数来表示，增量 checkpoint 非常使用于超大状态的场景。

讲了这么多 RocksDBStateBackend 的好处，那么该如何去使用呢，可以来看看 RocksDBStateBackend 这个类的相关属性以及构造函数。

属性：

- checkpointStreamBackend：用于创建 Checkpoint 流的状态后端
- localRocksDbDirectories：RocksDB 目录的基本路径，默认是 TaskManager 的临时目录
- enableIncrementalCheckpointing：是否增量 Checkpoint
- numberOfTransferingThreads：用于传输(下载和上传)状态的线程数量，默认为 1
- enableTtlCompactionFilter：是否启用压缩过滤器来清除带有 TTL 的状态

构造函数：

- RocksDBStateBackend(String checkpointDataUri)：单参数，只传入一个路径
- RocksDBStateBackend(String checkpointDataUri, boolean enableIncrementalCheckpointing)：两个参数，传入 Checkpoint 数据目录路径和是否开启增量 Checkpoint
- RocksDBStateBackend(StateBackend checkpointStreamBackend)：传入一种 StateBackend
- RocksDBStateBackend(StateBackend checkpointStreamBackend, TernaryBoolean enableIncrementalCheckpointing)：传入一种 StateBackend 和是否开启增量 checkpoint
- RocksDBStateBackend(RocksDBStateBackend original, Configuration config, ClassLoader classLoader)：私有的构造方法，用于重新配置状态后端

既然知道这么多构造函数了，那么使用就很简单了，根据你的场景考虑使用哪种构造函数创建 RocksDBStateBackend 对象就行了，然后通过 `env.setStateBackend()` 传入对象实例就行，如下所示：

```
//env.setStateBackend(new RocksDBStateBackend(checkpointDir,
incrementalCheckpoints));  //设置 RocksDB 存储
```

那么在使用 RocksDBStateBackend 时该注意什么呢：

- 当使用 RocksDB 时，状态大小将受限于磁盘可用空间的大小
- 状态存储在 RocksDB 中，整个更新和获取状态的操作都是要通过序列化和反序列化才能完成的，跟状态直接存储在内存中，性能可能会略低些
- 如果你应用程序的状态很大，那么使用 RocksDB 无非是最佳的选择

另外在 Flink 源码中有一个专门的 RocksDBOptions 来表示 RocksDB 相关的配置：

- state.backend.rocksdb.localdir：本地目录(在 TaskManager 上)，RocksDB 将其文件放在其中
- state.backend.rocksdb.timer-service.factory：定时器服务实现，默认值是 HEAP
- state.backend.rocksdb.checkpoint.transfer.thread.num：用于在后端传输(下载和上载)文件的线程数，默认是 1
- state.backend.rocksdb.ttl.compaction.filter.enabled：是否启用压缩过滤器来清除带有 TTL 的状态，默认值是 false

小结

通过上面三种 State Backends 的介绍，让大家了解了状态存储有哪些种类，然后对每种状态存储是该如何使用的、它们内部的实现、使用场景、需要注意什么都细讲了一遍，三种存储方式各有特点，可以满足不同场景的需求，通常来说，在开发程序之前，我们要先分析自己 Job 的场景和状态大小的预测，然后根据预测来进行选择何种状态存储，如果拿捏不定的话，建议先在测试环境进行测试，只有选择了正确的状态存储后端，这样才能够保证后面自己的 Job 在生产环境能够稳定的运行。