

有赞实时任务优化：Flink Checkpoint 异常解析与应用实践

作者：沈磊（有赞大数据）

有赞实时任务主要以 Flink 为主，为了保证实时任务的容错恢复以及停止重启时的状态恢复，几乎所有的实时任务都会开启 Checkpoint 或者触发 Savepoint 进行状态保存。由于 Savepoint 底层原理的实现和 Checkpoint 几乎一致，本文结合 Flink 1.9 版本，重点讲述 Flink Checkpoint 原理流程以及常见原因分析，让用户能够更好的理解 Flink Checkpoint，从而开发出更健壮的实时任务。

一、什么是 Flink Checkpoint 和状态

1.1 Flink Checkpoint 是什么

Flink Checkpoint 是一种容错恢复机制。这种机制保证了实时程序运行时，即使突然遇到异常或者机器问题时也能够进行自我恢复。Flink Checkpoint 对于用户层面来说，是透明的，用户会感觉实时任务一直在运行。

Flink Checkpoint 是 Flink 自身的系统行为，用户无法对其进行交互，用户可以在程序启动之前，设置好实时任务 Checkpoint 相关的参数，当任务启动之后，剩下的就全交给 Flink 自行管理。

1.2 为什么要开启 Checkpoint

实时任务不同于批处理任务，除非用户主动停止，一般会一直运行，运行的过程中可能存在机器故障、网络问题、外界存储问题等等，要想实时任务一直能够稳定运行，实时任务要有自动容错恢复的功能。而批处理任务在遇到异常情况时，在重新计算一遍即可。实时任务因为会一直运行的特性，如果在从头开始计算，成本会很大，尤其是对于那种运行时间很久的实时任务来说。

实时任务开启 Checkpoint 功能，也能够减少容错恢复的时间。因为每次都是从最新的 Checkpoint 点位开始状态恢复，而不是从程序启动的状态开始恢复。举个例子，如果你有一个运行一年的实时任务，如果容错恢复是从一年前启动时的状态恢复，实时任务可能需要运行很久才能恢复到现在状态，这一般是业务方所不允许的。

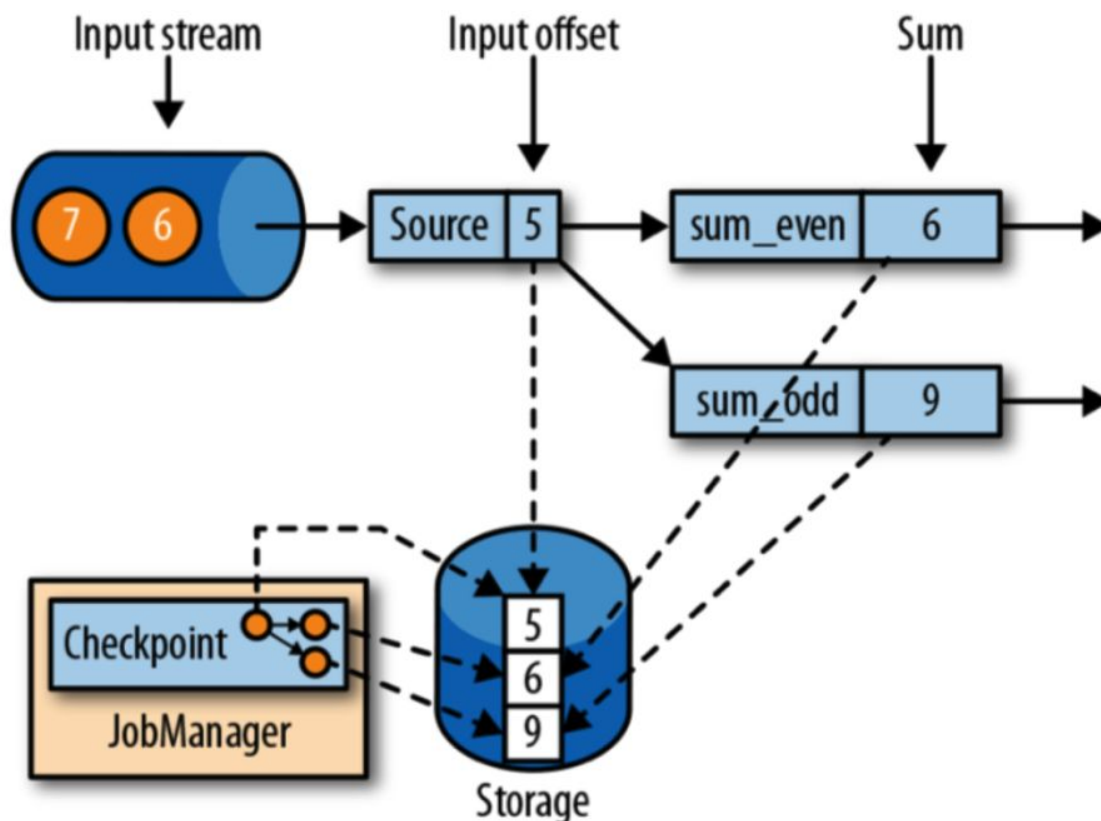
1.3 Flink 任务状态是什么

Flink Checkpoint 会将实时任务的状态存储到远端存储，比如 HDFS，亚马逊的 S3 等等。Flink 任务状态可以理解为实时任务计算过程中，中间产生的数据结果，同时这些计算结果会在后续实时任务处理时，能够继续进行使用。实时任务的状态可以是一个聚合结果值，比如 WordCount 统计的每个单词的数量，也可以是消息流中的明细数据。

Flink 任务状态整体可以划分两种：Operator 状态和 KeyedState。常见的 Operator 状态，比如 Kafka Topic 每

个分区的偏移量。KeyedState 是基于 KeyedStream 来使用的，所以在使用前，你需要对你的流通过 keyby 来进行分区，常见的状态比如有 MapState、ListState、ValueState 等等。

下面是一个实时计算奇数和偶数的任务的示例：



在上图中，假如输入的流来自于 Kafka，那么 Kafka Topic 分区的偏移量是状态，所有奇数的和、所有偶数的和也都是状态。

二、Flink Checkpoint 流程和原理

2.1 开启 Checkpoint 功能

想要使用 Flink Checkpoint 功能，首先是要在实时任务开启 Checkpoint。Flink 默认情况下是关闭 Checkpoint 功能，下面代码是开启 Checkpoint：

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  
/** 开启 checkpoint 功能 */  
env.enableCheckpointing(interval: 3000, CheckpointingMode.EXACTLY_ONCE);  
/** 使用RocksDB 进行状态存储 */  
env.setStateBackend(new RocksDBStateBackend(checkpointDataUri: "hdfsPath"));
```

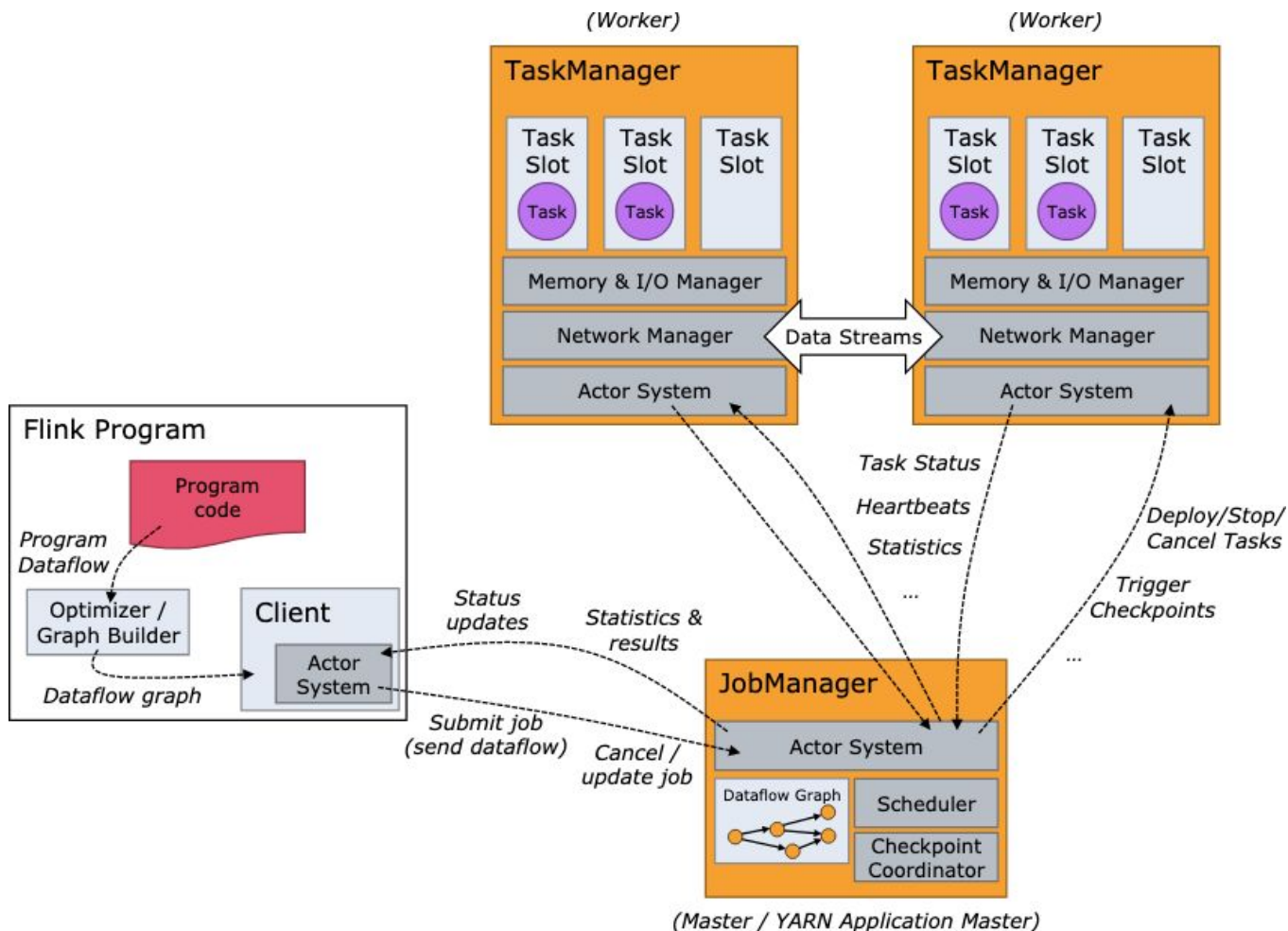
上述代码中，设置了 Flink Checkpoint 的间隔 3 秒，设置的 Checkpoint 的语义为 EXACTLY_ONCE。Flink 默认的 Checkpoint 语义为 EXACTLY_ONCE。上述代码也使用 RocksDBStateBackend 进行状态存储。用户也可以自己设置 Flink Checkpoint 的参数，通过 CheckpointConfig 这个类进行设置，代码如下：

```
CheckpointConfig chkConfig = env.getCheckpointConfig();
```

```
/** 调用 CheckpointConfig 各种 set 方法 */
chkConfig.setX
```

2.2 Flink 一次 Checkpoint 的参与者

Flink 整体作业采用主从架构，Master 为 JobManager，Slave 为 TaskManager，Client 则是负责提交用户实时任务的代码逻辑，Flink 整体框架图如下图所示：



JobManager 主要负责实时任务的调度以及对 Checkpoint 的触发，TaskManager 负责真正用户的代码执行逻辑，具体表现形式则是 Task 在 TaskManager 上面进行运行，一个 Task 对应一个线程，它可能运行一个算子的 SubTask，也可能是运行多个 Chain 起来的算子的 SubTask。

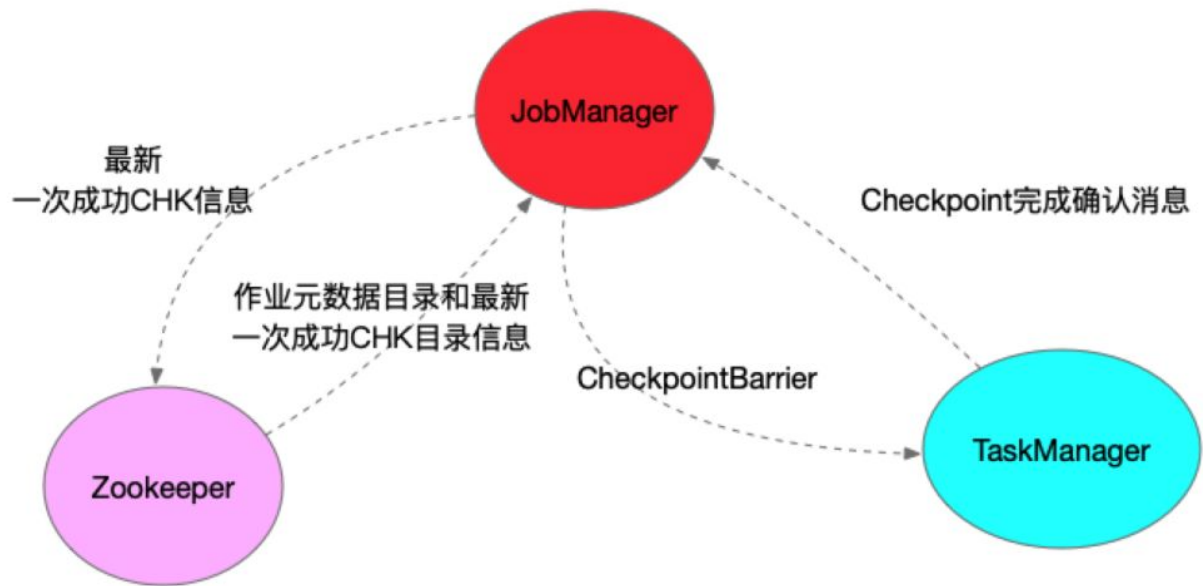
Flink 实时任务一次 Checkpoint 的参与者主要包括三块：JobManager、TaskManager 以及 Zookeeper。JobManager 定时会触发执行 Checkpoint，具体则是在 JobManager 中运行的 CheckpointCoordinator 中触发所有 Source 的 SubTask 向下游广播 CheckpointBarrier。

TaskManager 收到 CheckpointBarrier 后，根据 Checkpoint 的语义，决定是否在进行 CheckpointBarrier 对齐时，缓冲后续的数据记录，当收到所有上游输入的 CheckpointBarrier 后，开始做 Checkpoint。TaskManager Checkpoint 完成后，会向 JobManager 发送确认完成的消息。只有当所有 Sink 算子完成 Checkpoint 且发送确认消息后，该次 Checkpoint 才算完成。

在高可用模式下，ZooKeeper 主要存储最新一次 Checkpoint 成功的目录，当 Flink 任务容错恢复时，会从最新成功的 Checkpoint 恢复。Zookeeper 同时也存储着 Flink 作业的元数据信息。比如在高可用模式下，Flink 会将

































































JobGraph 以及相关 Jar 包存储在 HDFS 上面，Zookeeper 记录着该信息。再次容错重启时，读取这些信息，进行任务启动。

下图是一次 Checkpoint 的参与者：



2.3 Checkpoint 协调者：CheckpointCoordinator

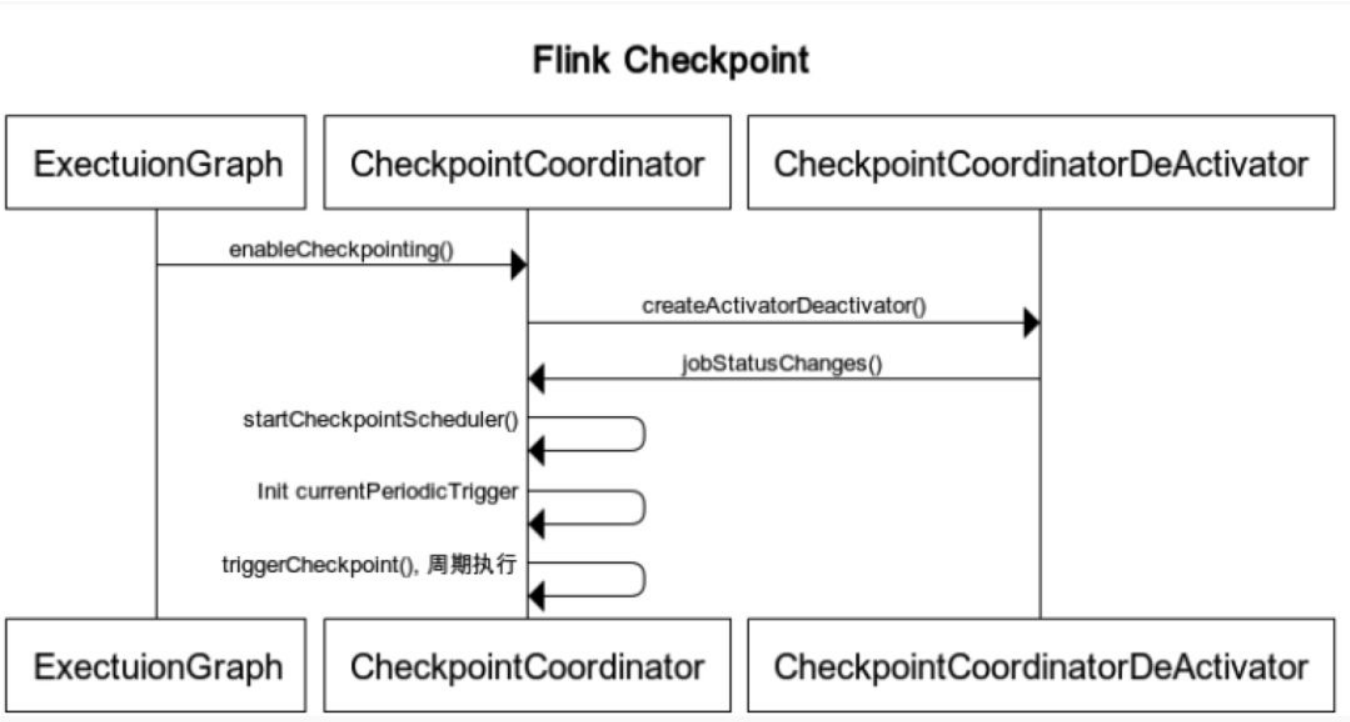
CheckpointCoordinator,是 Checkpoint 中最重要的类，协调着实时任务整个 Checkpoint 的执行。下图是 CheckpointCoordinator 中的方法：

CheckpointCoordinator		
 	addMasterHook(MasterTriggerRestoreHook<?>)	boolean
 	getNumberOfRegisteredMasterHooks()	int
 	setCheckpointStatsTracker(CheckpointStatsTracker)	void
 	shutdown(JobStatus)	void
 	isShutdown()	boolean
 	triggerSavepoint(long, String)	CompletableFuture<CompletedCheckpoint>
 	triggerCheckpoint(long, boolean)	boolean
 	triggerCheckpoint(long, CheckpointProperties, String, boolean)	CheckpointTriggerResult
 	receiveDeclineMessage(DeclineCheckpoint)	void
 	receiveAcknowledgeMessage(AcknowledgeCheckpoint)	boolean
 	completePendingCheckpoint(PendingCheckpoint)	void
 	failUnacknowledgedPendingCheckpointsFor(ExecutionAttemptID, Throwable)	void
 	rememberRecentCheckpointId(long)	void
 	dropSubsumedCheckpoints(long)	void
 	triggerQueuedRequests()	void
 	getNumScheduledTasks()	int
 	restoreLatestCheckpointedState(Map<JobVertexID, ExecutionJobVertex>, boolean, boolean)	boolean
 	restoreSavepoint(String, boolean, Map<JobVertexID, ExecutionJobVertex>, ClassLoader)	boolean
 	getNumberOfPendingCheckpoints()	int
 	getNumberOfRetainedSuccessfulCheckpoints()	int
 	getPendingCheckpoints()	Map<Long, PendingCheckpoint>
 	getSuccessfulCheckpoints()	List<CompletedCheckpoint>
 	getCheckpointStorage()	CheckpointStorage
 	getCheckpointStore()	CompletedCheckpointStore
 	getCheckpointIdCounter()	CheckpointIDCounter
 	getCheckpointTimeout()	long
 	isPeriodicCheckpointingConfigured()	boolean
 	startCheckpointScheduler()	void
 	stopCheckpointScheduler()	void
 	createActivatorDeactivator()	JobStatusListener
 	discardCheckpoint(PendingCheckpoint, Throwable)	void
 	discardSubtaskState(JobID, ExecutionAttemptID, long, TaskStateSnapshot)	void

Flink CheckpointCoordinator 中有几个比较重要的方法：

1. triggerCheckpoint, 触发 Flink 任务进行 Checkpoint 的方法
2. triggerSavepoint, 触发 Flink 任务 Savepoint 的方法
3. restoreSavepoint, Flink 任务从 Savepoint 状态恢复
4. restoreLatestCheckpointedState, 从最新一次 Checkpoint 点位状态恢复
5. receiveAcknowledgeMessage, 接受 Operator SubTask Checkpoint 完成的消息并处理

Flink CheckpointCoordinator 类是在 ExecutionGraph 形成时进行初始化的，具体则是在 ExecutionGraph 创建之后，调用 enableCheckpointing 方法，然后在该方法中，CheckpointCoordinator 进行创建。以下是 Flink Checkpoint 触发的时序图：



当 Flink 作业状态由创建到运行时，CheckpointCoordinator 中的 ScheduledThreadPoolExecutor 会定时执行 ScheduledTrigger 中的逻辑。ScheduledTrigger 本质就是一个 Runnable，run 方法中执行 triggerCheckpoint 方法。

2.4 Flink Checkpoint 流程与原理

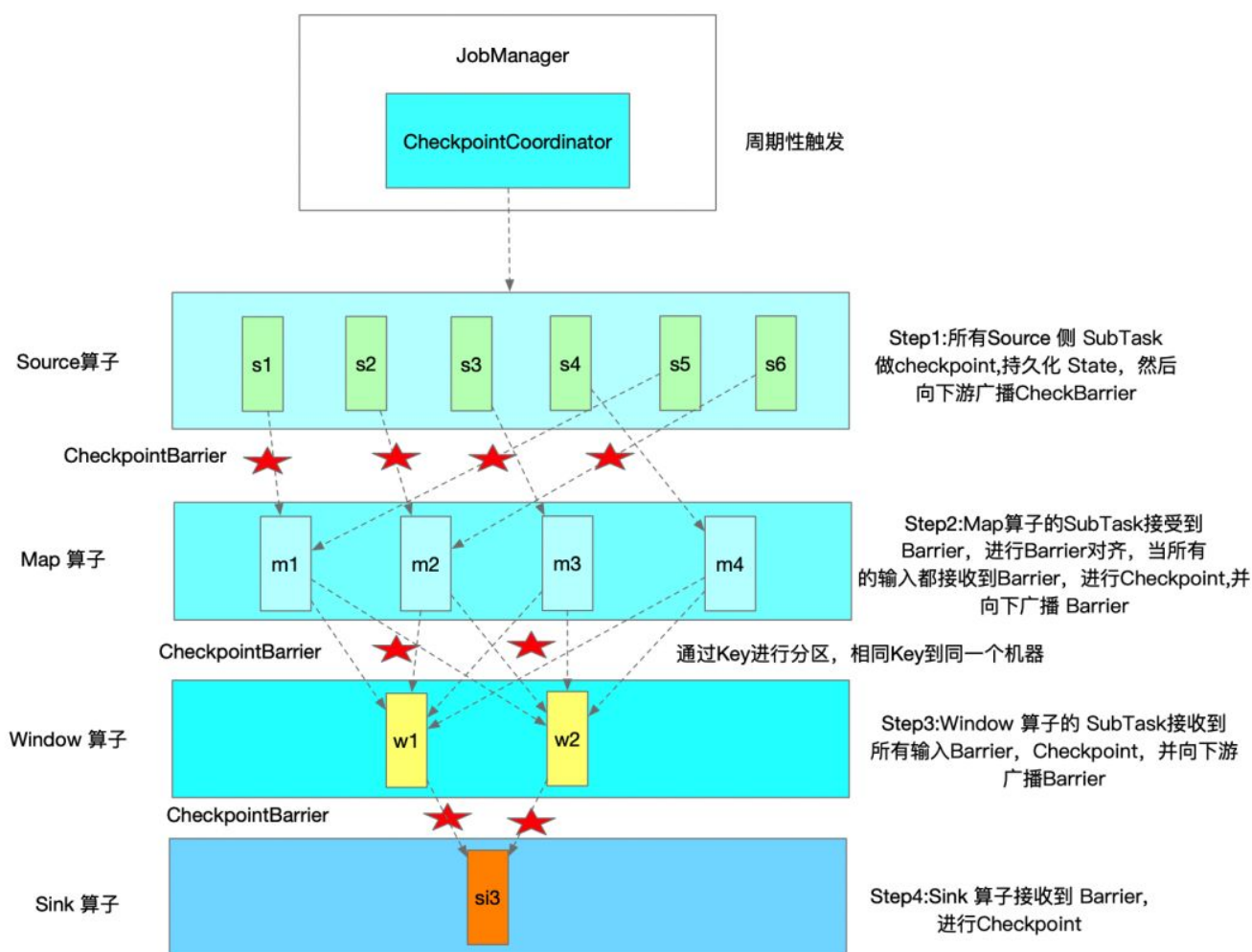
一次 Flink Checkpoint 的流程是从 CheckpointCoordinator 的 triggerCheckpoint 方法开始，下面来看看一次 Flink Checkpoint 涉及到的主要内容：

1. Checkpoint 开始之前先进行预检查，比如检查最大并发的 Checkpoint 数，最小的 Checkpoint 之间的时间间隔。默认情况下，最大并发的 Checkpoint 数为 1，最小的 Checkpoint 之间的时间间隔为 0。
2. 判断所有 Source 算子的 Subtask (Execution) 是否都处于运行状态，有则直接报错。同时检查所有待确认的算子的 SubTask(Execution)是否是运行状态，有则直接报错。
3. 创建 PendingCheckpoint，同时为该次 Checkpoint 创建一个 Runnable，即超时取消线程，默认 Checkpoint 十分钟超时。
4. 循环遍历所有 Source 算子的 Subtask(Execution),最底层调用 Task 的triggerCheckpointBarrier, 广播 CheckBarrier 到下游，同时 Checkpoint 其状态。
5. 下游的输入中有 CheckpointBarrierHandler 类来处理 CheckpoinBarrier，然后会调用 notifyCheckpoint 方法，通知 Operator SubTask 进行 Checkpoint。
6. 每当 Operator SubTask 完成 Checkpoint 时，都会向 CheckpointCoordoritor 发送确认消息。Checkpoint Coordinator 的 receiveAcknowledgeMessage 方法会进行处理。
7. 在一次 Checkpoint 过程中，当所有从 Source 端到 Sink 端的算子 SubTask 都完成之后，CheckpointCoor

driver 会通知算子进行 notifyCheckpointCompleted 方法，前提是算子的函数实现 CheckpointListener 接口。

Flink 会定时在任务的 Source 算子的 SubTask 触发 CheckpointBarrier，CheckpointBarrier 是一种特殊的消息事件，会随着消息通道流入到下游的算子中。只有当最后 Sink 端的算子接收到 CheckpointBarrier 并确认该次 Checkpoint 完成时，该次 Checkpoint 才算完成。所以在某些算子的 Task 有多个输入时，会存在 Barrier 对齐时间，我们可以在 Flink Web UI 上面看到各个 Task 的 CheckpointBarrier 对齐时间。

下图是一次 Flink Checkpoint 实例流程示意图：



Flink Checkpoint 保存的任务状态在程序取消停止时，默认会进行清除。Checkpoint 状态保留策略主要有两种：

DELETE_ON_CANCELLATION, RETAIN_ON_CANCELLATION

DELETE_ON_CANCELLATION 表示当程序取消时，删除 Checkpoint 存储的状态文件。RETAIN_ON_CANCELLATION 表示当程序取消时，保存之前的 Checkpoint 存储的状态文件 用户可以结合业务情况，设置 Checkpoint 保留模式：

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  
/** 开启 checkpoint */
```

```
env.enableCheckpointing(10000);  
/** 设置 checkpoint 保留策略,取消程序时,保留 checkpoint 状态文件 */  
env.getCheckpointConfig.enableExternalizedCheckpoints(ExternalizedCheckpointClean
```

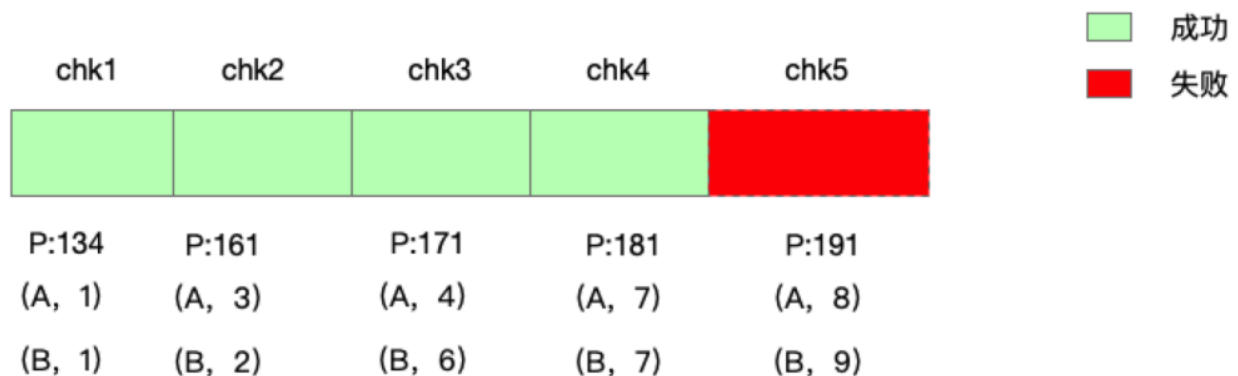
2.5 Flink Checkpoint 语义

Flink Checkpoint 支持两种语义：Exactly_Once 和 At_least_Once，默认的 Checkpoint 语义是 Exactly_Once。具体语义含义如下：

Exactly_Once 含义是：保证每条数据对于 Flink 任务的状态结果只影响一次。打个比方，比如 WordCount 程序，目前实时统计的 "hello" 这个单词数为 5，同时这个结果在这次 Checkpoint 成功后，保存在了 HDFS。在下次 Checkpoint 之前，又来 2 个 "hello" 单词，突然程序遇到外部异常自动容错恢复，会从最近的 Checkpoint 点开始恢复，那么会从单词数为 5 的这个状态点开始恢复，Kafka 消费的数据点位也是状态为 5 这个点位开始计算，所以即使程序遇到外部异常自动恢复时，也不会影响到 Flink 状态的结果计算。

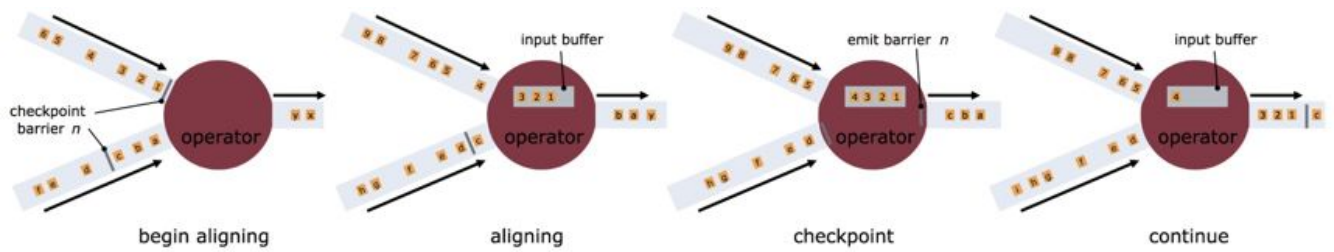
At_Least_Once 含义是：每条数据对于 Flink 任务的状态计算至少影响一次。比如在 WordCount 程序中，你统计到的某个单词的单词数可能会比真实的单词数要大，因为同一条消息，当 Flink 任务容错恢复后，可能将其计算多次。

Flink 中 Exactly_Once 和 At_Least_Once 具体是针对 Flink 任务状态而言的，并不是 Flink 程序对消息记录只处理一次。举个例子，当前 Flink 任务正在做 Checkpoint，该次 Checkpoint 还没有完成，这次 Checkpoint 时间段的数据其实已经进入 Flink 程序处理，只是程序状态没有最终存储到远程存储。当程序突然遇到异常，进行容错恢复时，那么就会从最新的 Checkpoint 进行状态恢复重启，上一次 Checkpoint 成功到这次 Checkpoint 失败的数据还会进入 Flink 系统重新处理，具体实例如下图：



上图中表示一个 WordCount 实时任务的 Checkpoint，在进行 chk-5 Checkpoint 时，突然遇到程序异常，那么实时任务会从 chk-4 进行恢复，那么之前 chk-5 处理的数据，Flink 系统会再次进行处理。不过这些数据的状态没有 Checkpoint 成功，所以 Flink 任务容错恢复再次运行时，对于状态的影响还是只有一次。

Exactly_Once 和 At_Least_Once 具体在底层实现大致相同，具体差异表现在 CheckpointBarrier 对齐方式的处理：



如果是 Exactly_Once 模式，某个算子的 Task 有多个输入通道时，当其中一个输入通道收到 CheckpointBarrier 时，Flink Task 会阻塞该通道，其不会处理该通道后续数据，但是会将这些数据缓存起来，一旦完成了所有输入通道的 CheckpointBarrier 对齐，才会继续对这些数据进行消费处理。

对于 At_least_Once，同样针对某个算子的 Task 有多个输入通道的情况下，当某个输入通道接收到 Checkpoint Barrier 时，它不同于 Exactly Once，即使没有完成所有输入通道 CheckpointBarrier 对齐，At Least Once 也会继续处理后续接收到的数据。所以使用 At Least Once 不能保证数据对于状态计算只有一次的计算影响。

三、Flink Checkpoint 常见失败原因和注意点

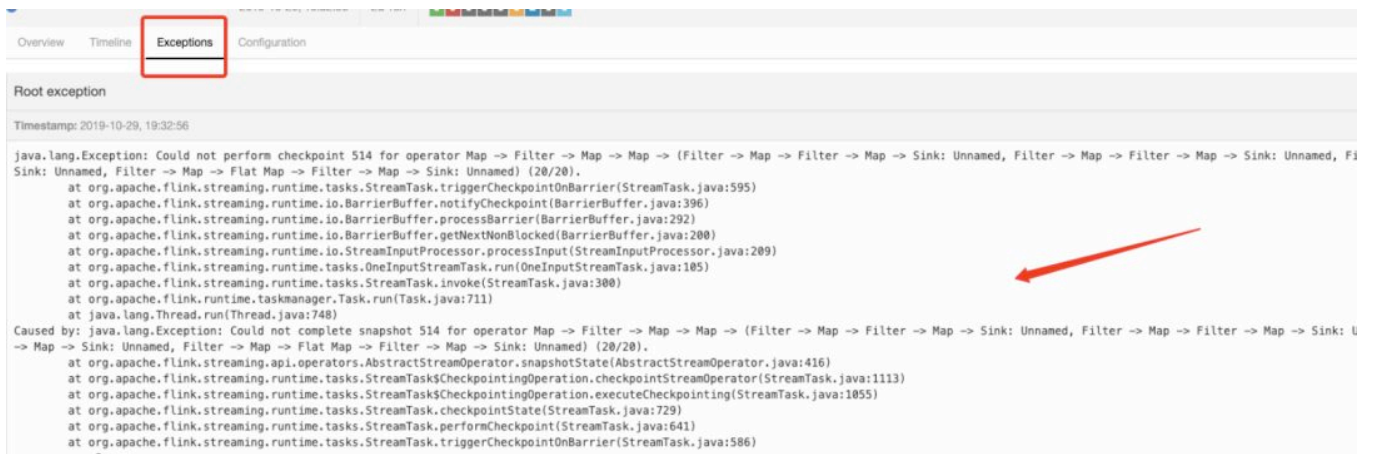
3.1 Flink Checkpoint 常见失败原因分析

Flink Checkpoint 失败有很多原因，常见的失败原因如下：

1. 用户代码逻辑没有对于异常处理，让其直接在运行中抛出。比如解析 Json 异常，没有捕获，导致 Checkpoint 失败，或者调用 Dubbo 超时异常等等。
2. 依赖外部存储系统，在进行数据交互时，出错，异常没有处理。比如输出数据到 Kafka、Redis、HBase 等，客户端抛出了超时异常，没有进行捕获，Flink 任务容错机制会再次重启。
3. 内存不足，频繁 GC，超出了 GC 负载的限制。比如 OOM 异常
4. 网络问题、机器不可用问题等等。

从目前的具体实践情况来看，Flink Checkpoint 异常大多数还是用户代码逻辑的问题，对于程序异常没有正确的处理导致。所以在编写 Flink 实时任务时，一定要注意处理程序可能出现的各种异常。这样，也会让实时任务的逻辑更加的健壮。

当自己的 Flink 实时任务 Checkpoint 失败时，用户可以先通过 Flink Web UI 进行快速定位 Checkpoint 失败的原因，如果在 Flink Web UI 上面没有看到异常信息，可以去看任务的具体日志进行定位，如下是 Flink Web UI 查看错误原因示意图：

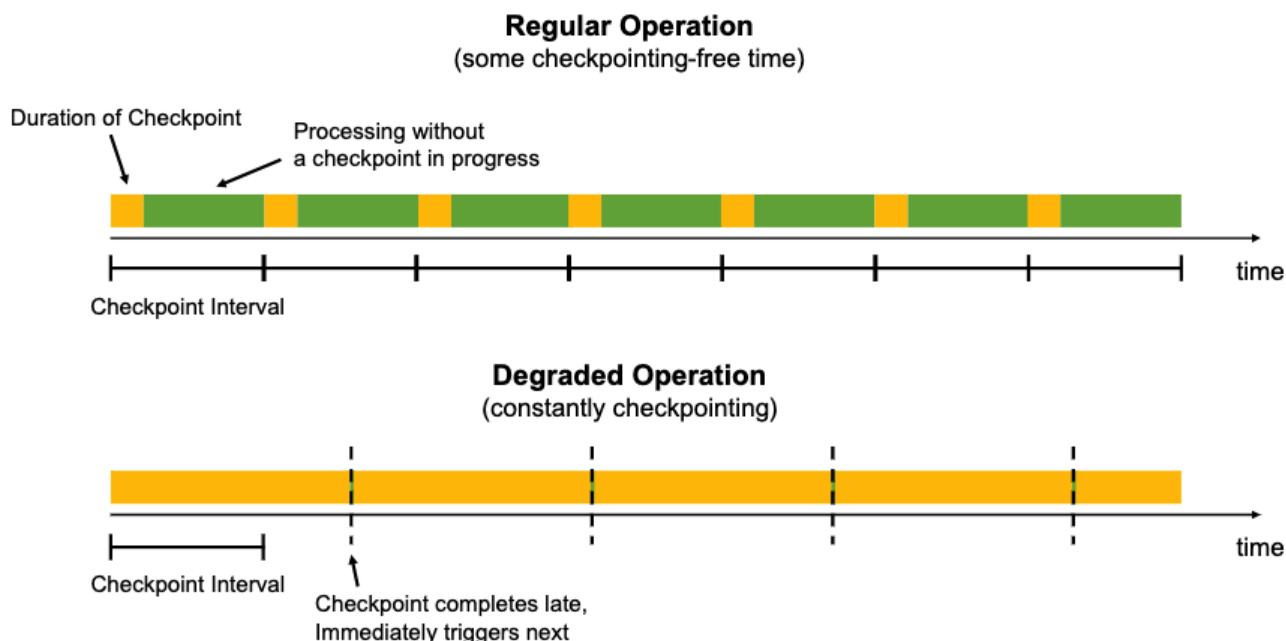


3.2 Flink Checkpoint 参数配置及注意点

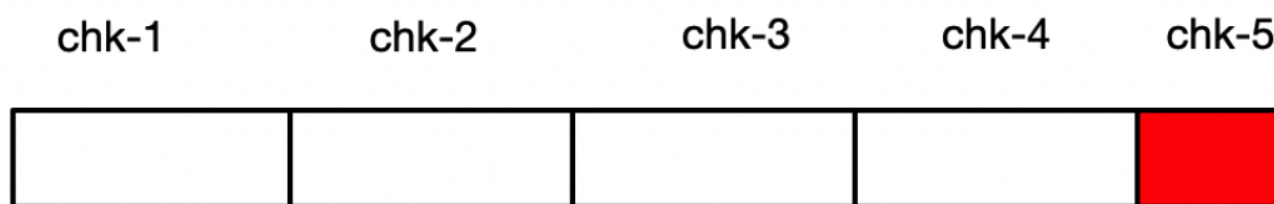
下面是设置 Flink Checkpoint 参数配置的建议及注意点：

1. 当 Checkpoint 时间比设置的 Checkpoint 间隔时间要长时，可以设置 Checkpoint 间最小时间间隔。这样在上次 Checkpoint 完成时，不会立马进行下一次 Checkpoint，而是会等待一个最小时间间隔，之后再进行检查点。否则，每次 Checkpoint 完成时，就会立马开始下一次 Checkpoint，系统会有很多资源消耗在 Checkpoint 方面，而真正任务计算的资源就会变少。
2. 如果 Flink 状态很大，在进行恢复时，需要从远程存储上读取状态进行恢复，如果状态文件过大，此时可能导致任务恢复很慢，大量的时间浪费在网络传输方面。此时可以设置 Flink Task 本地状态恢复，任务状态本地恢复默认没有开启，可以设置参数 `state.backend.local-recovery` 值为 `true` 进行激活。
3. Checkpoint 保存数，Checkpoint 保存数默认是 1，也就是只保存最新的 Checkpoint 的状态文件，当进行状态恢复时，如果最新的 Checkpoint 文件不可用时(比如 HDFS 文件所有副本都损坏或者其他原因)，那么状态恢复就会失败，如果设置 Checkpoint 保存数 2，即使最新的 Checkpoint 恢复失败，那么 Flink 会回滚到之前那一次 Checkpoint 的状态文件进行恢复。考虑到这种情况，用户可以增加 Checkpoint 保存数。
4. 建议设置的 Checkpoint 的间隔时间最好大于 Checkpoint 的完成时间。

下图是不设置 Checkpoint 最小时间间隔示例图，可以看到，系统一致在进行 Checkpoint，大量的资源使用在 Flink Checkpoint 上，可能对运行的任务产生一定影响：



还有一种特殊的情况，Flink 端到端 Sink 的 EXACTLYONCE 的问题，也就是数据从 Flink 端到外部消息系统的消息一致性。打个比方，Flink 输出数据到 Kafka 消息系统中，如果使用 Kafka 0.10 的版本，Flink 不支持端到端的 EXACTLYONCE，可能存在消息重复输入到 Kafka。



如上图所示，当做 chk-5 Checkpoint 的时候，chk-5 失败，然后从 chk-4 来进行恢复，但是 chk-5 的部分数据在 Checkpoint 失败之前就有部分进入到 Kafka 消息系统，再次恢复时，该部分数据可能再次重放到 Kafka 消息系统中。

Flink 中解决端到端的一致性有两种方法：做幂等以及事务写，幂等的话，可以使用 KV 存储系统来做幂等，因为 KV 存储系统的多次操作结果都是相同的。Flink 内部目前支持二阶段事务提交，Kafka 0.11 以上版本支持事务写，所以支持 Flink 端到 Kafka 端的 EXACTLY_ONCE。

四、有赞的优化实践

有赞实时计算对于 Flink 任务的 Checkpoint 和 Savepoint 做了两个方面工作，第一个工作对于 Flink Checkpoint 失败的情况，如果 Checkpoint 失败过于频繁，同时 Flink Checkpoint 失败次数如果达到平台默认的失败阈值，平台会及时给用户报警提示。我们会每 5 分钟检查一次实时任务，统计实时任务近 15 分钟内，Flink Checkpoint 失败次数的最大值和最小值的差值达到平台默认的阈值，则会立马给用户报警，让用户能够及时的处理问题。

当然，并不是所有的 Flink 实时任务 Checkpoint 失败平台都能发现，因为 Checkpoint 失败次数的检查，首先与用户配置的 Checkpoint 的时间间隔有关。举个例子，如果用户配置的 Checkpoint 间隔为 1 小时，其实平台

默认 Checkpoint 逻辑检查根本就无法发现实时任务 Checkpoint 失败。

针对这种情况，实时平台也支持用户自定义设置 Checkpoint 失败阈值，目前支持两种 Checkpoint 失败逻辑检查，一个是实时任务的 Checkpoint 失败次数的总和达到阈值，另一个则是近 10 分钟内，Flink Checkpoint 次数的最大值和最小值的差值的计算逻辑，用户可以根据实时任务的敏感度，设置具体的参数。

第二个方面则是针对 Flink 任务的状态恢复，为了防止实时任务的状态丢失，实时计算平台会定期的对实时任务进行 Savepoint 触发，当任务由于外界因素导致任务失败时，这种失败是任务直接挂掉，Yarn 任务的状态直接为 Killed，这种情况下，如果用户开启自动拉起功能，实时平台自动拉起实时任务，同时从最新的 Savepoint 进行状态恢复，以至于状态不丢失。同时，实时计算平台也支持用户停止任务时，触发 Savepoint，再次重启实时任务时，还是从停止时的任务状态进行恢复。

五、总结

目前，有赞在实时计算方面，还有很长的路要走。在满足业务的同时，可能也会有很多的坑需要踩。后面有赞实时计算会重点在实时数仓方面进行投入，同时会基于 Flink SQL 进行功能扩展和开发。

为了用户开发实时任务的便利性，后面有赞会开始进行在线实时计算平台的设计开发。未来也会将实时任务迁移到 K8S 上面，这样在大促场景下，能够更方便的进行资源的扩容和缩容。

未来，有赞实时计算平台会为用户带来更好的开发体验，降低用户开发实时任务的难度，让我们一起拭目以待。