Executor 端长时容错详解

[酷玩 Spark] Spark Streaming 源码解析系列，返回目录请 猛戳这里

「腾讯广告」技术团队（原腾讯广点通技术团队）荣誉出品

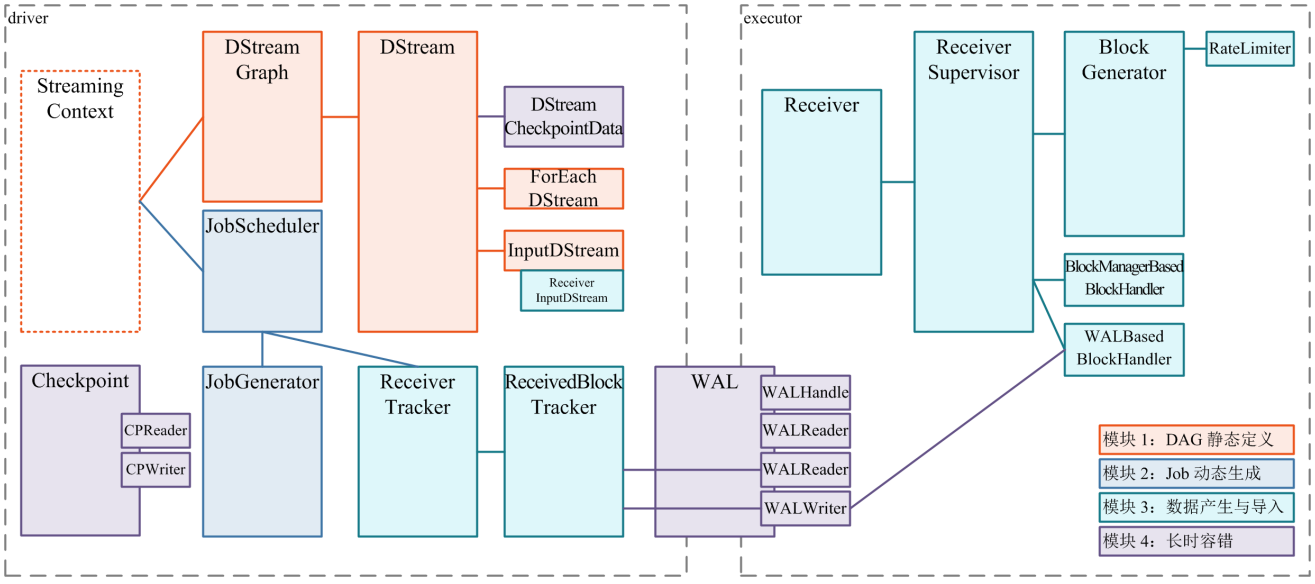
```
本系列内容适用范围：

* 2018.11.02 update, Spark 2.4 全系列 √ (已发布：2.4.0)
* 2018.02.28 update, Spark 2.3 全系列 √ (已发布：2.3.0 ~ 2.3.2)
* 2017.07.11 update, Spark 2.2 全系列 √ (已发布：2.2.0 ~ 2.2.3)
```

阅读本文前，请一定先阅读 Spark Streaming 实现思路与模块概述 一文，其中概述了 Spark Streaming 的 4 大模块的基本作用，有了全局概念后再看本文对 `模块 4：长时容错` 细节的解释。

引言

之前的详解我们详解了完成 Spark Streamimg 基于 Spark Core 所新增功能的 3 个模块，接下来我们看一看第 4 个模块将如何保障 Spark Streaming 的长时运行 —— 也就是，如何与前 3 个模块结合，保障前 3 个模块的长时运行。

通过前 3 个模块的关键类的分析，我们可以知道，保障模块 1 和 2 需要在 driver 端完成，保障模块 3 需要在 executor 端和 driver 端完成。



本文我们详解 executor 端的保障。

在 executor 端，`ReceiverSupervisor` 和 `Receiver` 失效后直接重启就 OK 了，关键是保障收到的块数据的安全。保障了源头块数据，就能够保障 RDD DAG （Spark Core 的 lineage）重做。

Spark Streaming 对源头块数据的保障，分为 4 个层次，全面、相互补充，又可根据不同场景灵活设置：

- (1) 热备
- (2) 冷备
- (3) 重放
- (4) 忽略

(1) 热备

热备是指在存储块数据时，将其存储到本 executor、并同时 replicate 到另外一个 executor 上去。这样在一个 replica 失效后，可以立刻无感知切换到另一份 replica 进行计算。

实现方式是，在实现自己的 `Receiver` 时，即指定一下 `StorageLevel` 为 `MEMORY_ONLY_2` 或 `MEMORY_AND_DISK_2` 就可以了。

比如这样：

```
class MyReceiver extends Receiver(StorageLevel.MEMORY_ONLY_2) {
  override def onStart(): Unit = {}
  override def onStop(): Unit = {}
}
```

这样，`Receiver` 在将数据 `store()` 给 `ReceiverSupervisorImpl` 的时候，将同时指明此 `storageLevel`。`ReceiverSupervisorImpl` 也将根据此 `storageLevel` 将块数据具体的存储给 `BlockManager`。

然后就是依靠 `BlockManager` 进行热备。具体的 —— 我们以 `ReceiverSupervisorImpl` 向 `BlockManager` 存储一个 `byteBuffer` 为例 —— `BlockManager` 在收到 `putBytes(byteBuffer)` 时，实际是直接调用 `doPut(byteBuffer)` 的。 那么我们看 `doPut(...)` 方法（友情提醒，主要看代码里的注释）：

```
private def doPut(blockId: BlockId, data: BlockValues, level: StorageLevel, ...)
  : Seq[(BlockId, BlockStatus)] = {
  ...
  //【如果  putLevel.replication > 1 的话，就定义这个 future，复制数据到另外的 executor 上】
  val replicationFuture = data match {
    case b: ByteBufferValues if putLevel.replication > 1 =>
      val bufferView = b.buffer.duplicate()
      Future {
        //【这里非常重要，会在 future 启动时去实际调用 replicate() 方法，复制数据到另外的 executor 上】
        replicate(blockId, bufferView, putLevel)
```

```
      }(futureExecutionContext)
    case _ => null
  }

  putBlockInfo.synchronized {
    ...
    // 【存储到本机 blockManager 的 blockStore 里】
    val result = data match {
      case IteratorValues(iterator) =>
        blockStore.putIterator(blockId, iterator, putLevel, returnValues)
      case ArrayValues(array) =>
        blockStore.putArray(blockId, array, putLevel, returnValues)
      case ByteBufferValues(bytes) =>
        bytes.rewind()
        blockStore.putBytes(blockId, bytes, putLevel)
    }
  }

  //【再次判断  putLevel.replication > 1】
  if (putLevel.replication > 1) {
    data match {
      case ByteBufferValues(bytes) =>
        //【如果之前启动了 replicate 的 future，那么这里就同步地等这个 future 结束】
        if (replicationFuture != null) {
          Await.ready(replicationFuture, Duration.Inf)
        }
      case _ =>
        val remoteStartTime = System.currentTimeMillis
        if (bytesAfterPut == null) {
          if (valuesAfterPut == null) {
            throw new SparkException(
              "Underlying put returned neither an Iterator nor bytes! This
shouldn't happen.")
          }
          bytesAfterPut = dataSerialize(blockId, valuesAfterPut)
        }
        //【否则之前没有启动 replicate 的 future，那么这里就同步地调用 replicate() 方法，
复制数据到另外的 executor 上】
        replicate(blockId, bytesAfterPut, putLevel)
        logDebug("Put block %s remotely took %s"
          .format(blockId, Utils.getUsedTimeMs(remoteStartTime)))
    }
  }

  ...
}
```

所以，可以看到，`BlockManager` 的 `putBytes()` 语义就是承诺了，如果指定需要 replicate，那么当 `putBytes()` 方法返回时，就一定是存储到本机、并且一定 replicate 到另外的 executor 上了。对于 `BlockManager` 的 `putIterator()` 也是同样的语义，因为 `BlockManager` 的 `putIterator()` 和 `BlockManager` 的 `putBytes()` 一样，都是基于 `BlockManager` 的 `doPut()` 来实现的。

简单总结本小节的解析，`Receiver` 收到的数据，通过 `ReceiverSupervisorImpl`，将数据交给 `BlockManager` 存储；而 `BlockManager` 本身支持将数据 `replicate()` 到另外的 executor 上，这样就完成了 `Receiver` 源头数据的热备过程。

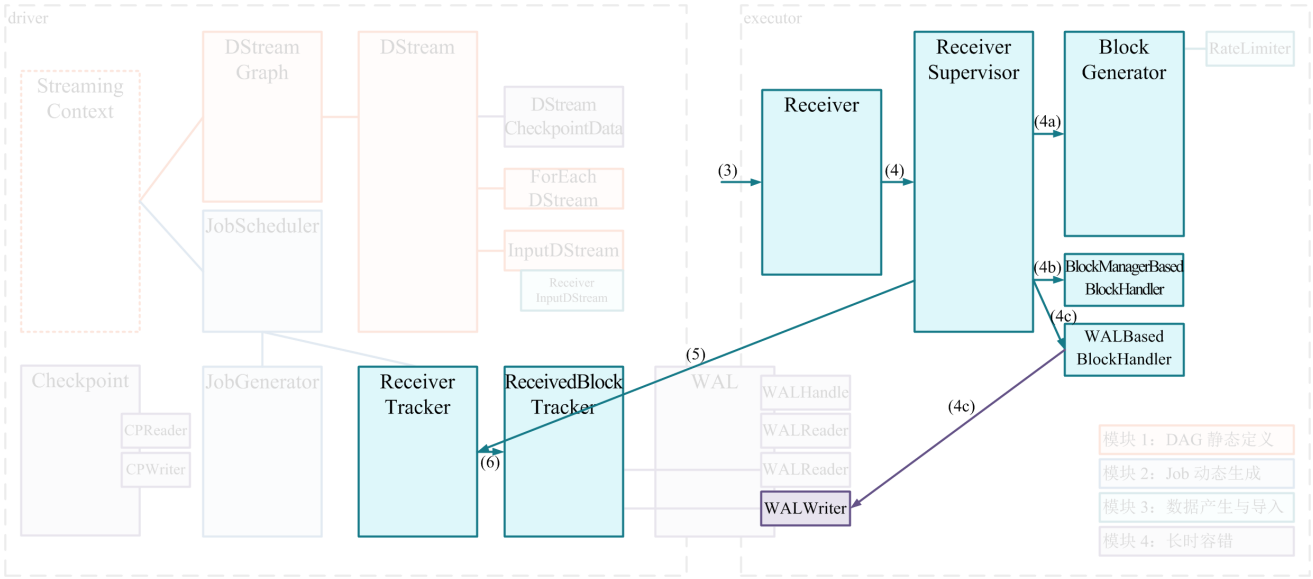
而在计算时，计算任务首先将获取需要的块数据，这时如果一个 executor 失效导致一份数据丢失，那么计算任务将转而向另一个 executor 上的同一份数据获取数据。因为另一份块数据是现成的、不需要像冷备那样重新读取的，所以这里不会有 recovery time。

(2) 冷备

!!! 需要同时修改

冷备是每次存储块数据时，除了存储到本 executor，还会把块数据作为 log 写出到 WriteAheadLog 里作为冷备。这样当 executor 失效时，就由另外的 executor 去读 WAL，再重做 log 来恢复块数据。WAL 通常写到可靠存储如 HDFS 上，所以恢复时可能需要一段 recover time。

冷备的写出过程如下图 4(c) 过程所示：



这里我们需要插播一下详解 `WriteAheadLog` 框架。

WriteAheadLog 框架

`WriteAheadLog` 的方式在单机 RDBMS、NoSQL/NewSQL 中都有广泛应用，前者比如记录 transaction log 时，后者比如 HBase 插入数据可以先写到 HLog 里。

`WriteAheadLog` 的特点是顺序写入，所以在做数据备份时效率较高，但在需要恢复数据时又需要顺序读取，所以需要一定 recovery time。

不过对于 Spark Streaming 的块数据冷备来讲，在恢复时也非常方便。这是因为，对某个块数据的操作只有一次（即新增块数据），而没有后续对块数据的追加、修改、删除操作，这就使得在 WAL 里只会有一条此块数据的 log entry。所以，我们在恢复时只要 seek 到这条 log entry 并读取就可以了，而不需要顺序读取整个 WAL。

也就是，Spark Streaming 基于 WAL 冷备进行恢复，需要的 recovery time 只是 seek 到并读一条 log entry 的时间，而不是读取整个 WAL 的时间，这个是个非常大的节省。

Spark Streaming 里的 WAL 框架，由一组抽象类，和一组基于文件的具体实现组成。其类结构关系如下：

WriteAheadLog, WriteAheadLogRecordHandle

`WriteAheadLog` 是多条 log 的集合，每条具体的 log 的引用就是一个 `LogRecordHandle`。这两个 abstract 的接口定义如下：

```
//  来自 WriteAheadLog

@org.apache.spark.annotation.DeveloperApi
public abstract class WriteAheadLog {
  //  【写方法：写入一条 log，将返回一个指向这条 log 的句柄引用】
  abstract public WriteAheadLogRecordHandle write(ByteBuffer record, long time);

  //  【读方法：给定一条 log 的句柄引用，读出这条 log】
  abstract public ByteBuffer read(WriteAheadLogRecordHandle handle);

  //  【读方法：读取全部 log】
  abstract public Iterator<ByteBuffer> readAll();

  //  【清理过时的 log 条目】
  abstract public void clean(long threshTime, boolean waitForCompletion);

  //  【关闭方法】
  abstract public void close();
}

//  来自 WriteAheadLogRecordHandle
```

```
@org.apache.spark.annotation.DeveloperApi
public abstract class WriteAheadLogRecordHandle implements java.io.Serializable {
  //  【Handle 则是一个空接口，需要具体的子类定义真正的内容】
}
```

这里 `WriteAheadLog` 基于文件的具体实现是 `FileBasedWriteAheadLog`，`WriteAheadLogRecordHandle` 基于文件的具体实现是 `FileBasedWriteAheadLogSegment`，下面我们详细看看这两个具体的类。

FileBasedWriteAheadLogSegment

`FileBasedWriteAheadLog` 有 3 个重要的配置项或成员：

- rolling 配置项
  - `FileBasedWriteAheadLog` 的实现把 log 写到一个文件里（一般是 HDFS 等可靠存储上的文件），然后每隔一段时间就关闭已有文件，产生一些新文件继续写，也就是 rolling 写的方式
  - rolling 写的好处是单个文件不会太大，而且删除不用的旧数据特别方便
  - 这里 rolling 的间隔是由参数 `spark.streaming.receiver.writeAheadLog.rollingIntervalSecs（默认 = 60 秒）` 控制的
- WAL 存放的目录：`{checkpointDir}/receivedData/{receiverId}`
  - `{checkpointDir}` 在 `ssc.checkpoint(checkpointDir)` 指定的
  - `{receiverId}` 是 `Receiver` 的 id
  - 在这个 WAL 目录里，不同的 rolling log 文件的命名规则是 `log-{startTime}-{stopTime}`
- 然后就是 `FileBasedWriteAheadLog.currentLogWriter`
  - 一个 `LogWriter` 对应一个 log file，而且 log 文件本身是 rolling 的，那么前一个 log 文件写完成后，对应的 writer 就可以 `close()` 了，而由新的 writer 负责写新的文件
  - 这里最新的 `LogWriter` 就由 `currentLogWriter` 来指向

接下来就是 `FileBasedWriteAheadLog` 的读写方法了：

- `write(byteBuffer: ByteBuffer, time: Long)`
  - 最重要的是先调用 `getCurrentWriter()`，获取当前的 currentWriter
  - 注意这里，如果 log file 需要 rolling 成新的了，那么 currentWriter 也需要随之更新；上面 `getCurrentWriter()` 会完成这个按需更新 `currentWriter` 的过程
  - 然后就可以调用 `writer.write(byteBuffer)` 就可以了
- `read(segment: WriteAheadLogRecordHandle): ByteBuffer`
  - 直接调用 `reader.read(fileSegment)`
  - 在 reader 的实现里，因为给定了 `segment` —— 也就是 `WriteAheadLogRecordHandle`，而 `segment` 里包含了具体的 log file 和 offset，就可以直接 seek 到这条 log，读出数据并返回

所以总结下可以看到，`FileBasedWriteAheadLog` 主要是进行 rolling file 的管理，然后将具体的写方法、读方法是由具体的 `LogWriter` 和 `LogReader` 来做的。

WriteAheadLogRecordHandle

前面我们刚说，`WriteAheadLogRecordHandle` 是一个 log 句柄的空实现，需要子类指定具体的 log 句柄内容。

然后在基于的 file 的子类实现 `WriteAheadLogRecordHandle` 里，就记录了 3 方面内容：

```
// 来自 FileBasedWriteAheadLogSegment

private[streaming] case class FileBasedWriteAheadLogSegment(path: String, offset:
Long, length: Int)
  extends WriteAheadLogRecordHandle
```

- `path: String`
- `offset: Long`
- `length: Int`

这 3 方面内容就非常直观了，给定文件、偏移和长度，就可以唯一确定一条 log。

FileBasedWriteAheadLogWriter

`FileBasedWriteAheadLogWriter` 的实现，就是给定一个文件、给定一个块数据，将数据写到文件里面去。

然后在完成的时候，记录一下文件 path、offset 和 length，封装为一个 `FileBasedWriteAheadLogSegment` 返回。

这里需要注意下的是，在具体的写 HDFS 数据块的时候，需要判断一下具体用的方法，优先使用 `hflush()`，没有的话就使用 `sync()`：

```
// 来自 FileBasedWriteAheadLogWriter

private lazy val hadoopFlushMethod = {
  // Use reflection to get the right flush operation
  val cls = classOf[FSDataOutputStream]
  Try(cls.getMethod("hflush")).orElse(Try(cls.getMethod("sync"))).toOption
}
```

FileBasedWriteAheadLogRandomReader

`FileBasedWriteAheadLogRandomReader` 的主要方法是 `read(segment: FileBasedWriteAheadLogSegment): ByteBuffer`，即给定一个 log 句柄，返回一条具体的 log。

这里主要代码如下，注意到其中最关键的是 `seek(segment.offset)`！

```
// 来自 FileBasedWriteAheadLogRandomReader
```

```
def read(segment: FileBasedWriteAheadLogSegment): ByteBuffer = synchronized {
  assertOpen()
  // 【seek 到这条 log 所在的 offset】
  instream.seek(segment.offset)
  // 【读一下 length】
  val nextLength = instream.readInt()
  HdfsUtils.checkState(nextLength == segment.length,
    s"Expected message length to be ${segment.length}, but was $nextLength")
  val buffer = new Array[Byte](nextLength)
  // 【读一下具体的内容】
  instream.readFully(buffer)
  // 【以 ByteBuffer 的形式，返回具体的内容】
  ByteBuffer.wrap(buffer)
}
```

FileBasedWriteAheadLogReader

`FileBasedWriteAheadLogReader` 实现跟 `FileBasedWriteAheadLogRandomReader` 差不多，不过是不需要给定 log 的句柄，而是迭代遍历所有 log：

```
// 来自 FileBasedWriteAheadLogReader

// 【迭代方法：hasNext()】
override def hasNext: Boolean = synchronized {
  if (closed) {
    // 【如果已关闭，就肯定不 hasNext 了】
    return false
  }

  if (nextItem.isDefined) {
    true
  } else {
    try {
      // 【读出来下一条，如果有，就说明还确实 hasNext】
      val length = instream.readInt()
      val buffer = new Array[Byte](length)
      instream.readFully(buffer)
      nextItem = Some(ByteBuffer.wrap(buffer))
      logTrace("Read next item " + nextItem.get)
      true
    } catch {
     ...
    }
  }
```

```
}

// 【迭代方法：next()】
override def next(): ByteBuffer = synchronized {
  // 【直接返回在 hasNext() 方法里实际读出来的数据】
  val data = nextItem.getOrElse {
    close()
    throw new IllegalStateException(
      "next called without calling hasNext or after hasNext returned false")
  }
  nextItem = None // Ensure the next hasNext call loads new data.
  data
}
```

WAL 总结

通过上面几个小节，我们看到，Spark Streaming 有一套基于 rolling file 的 WAL 实现，提供一个写方法，两个读方法：

- `WriteAheadLogRecordHandle write(ByteBuffer record, long time)`
  - 由 `FileBasedWriteAheadLogWriter` 具体实现
- ByteBuffer read(WriteAheadLogRecordHandle handle)`
  - 由 `FileBasedWriteAheadLogRandomReader` 具体实现
- `Iterator<ByteBuffer> readAll()`
  - 由 `FileBasedWriteAheadLogReader` 具体实现

(3) 重放

如果上游支持重放，比如 Apache Kafka，那么就可以选择不用热备或者冷备来另外存储数据了，而是在失效时换一个 executor 进行数据重放即可。

具体的，Spark Streaming 从 Kafka 读取方式有两种：

- 基于 `Receiver` 的
  - 这种是将 Kafka Consumer 的偏移管理交给 Kafka —— 将存在 ZooKeeper 里，失效后由 Kafka 去基于 offset 进行重放
  - 这样可能的问题是，Kafka 将同一个 offset 的数据，重放给两个 batch 实例 —— 从而只能保证 at least once 的语义
- Direct 方式，不基于 `Receiver`
  - 由 Spark Streaming 直接管理 offset —— 可以给定 offset 范围，直接去 Kafka 的硬盘上读数据，使用 Spark Streaming 自身的均衡来代替 Kafka 做的均衡
  - 这样可以保证，每个 offset 范围属于且只属于一个 batch，从而保证 exactly-once

这里我们以 Direct 方式为例，详解一下 Spark Streaming 在源头数据实效后，是如果从上游重放数据的。

这里的实现分为两个层面：

- `DirectKafkaInputDStream`：负责侦测最新 offset，并将 offset 分配至唯一个 batch
  - 会在每次 batch 生成时，依靠 `latestLeaderOffsets()` 方法去侦测最新的 offset
  - 然后与上一个 batch 侦测到的 offset 相减，就能得到一个 offset 的范围 `offsetRange`
  - 把这个 offset 范围内的数据，唯一分配到本 batch 来处理
- `KafkaRDD`：负责去读指定 offset 范围内的数据，并基于此数据进行计算
  - 会生成一个 Kafka 的 `SimpleConsumer` —— `SimpleConsumer` 是 Kafka 最底层、直接对着 Kafka 硬盘上的文件读数据的类
  - 如果 `Task` 失败，导致任务重新下发，那么 offset 范围仍然维持不变，将直接重新生成一个 Kafka 的 `SimpleConsumer` 去读数据

所以看 Direct 的方式，归根结底是由 Spark Streaming 框架来负责整个 offset 的侦测、batch 分配、实际读取数据；并且这些分 batch 的信息都是 checkpoint 到可靠存储（一般是 HDFS）了。这就没有用到 Kafka 使用 ZooKeeper 来均衡 consumer 和记录 offset 的功能，而是把 Kafka 直接当成一个底层的文件系统来使用了。

当然，我们讲上游重放并不只局限于 Kafka，而是说凡是支持消息重放的上游都可以 —— 比如，HDFS 也可以看做一个支持重放的可靠上游 —— FileInputDStream 就是利用重放的方式，保证了 executor 失效后的源头数据的可读性。

(4) 忽略

最后，如果应用的实时性需求大于准确性，那么一块数据丢失后我们也可以选择忽略、不恢复失效的源头数据。

假设我们有 r1, r2, r3 这三个 `Receiver`，而且每 5 秒产生一个 Block，每 15 秒产生一个 batch。那么，每个 batch 有 `15 s ÷ 5 block/s/receiver × 3 receiver = 9 block`。现在假设 r1 失效，随之也丢失了 3 个 block。

那么上层应用如何进行忽略？有两种粒度的做法。

粗粒度忽略

粗粒度的做法是，如果计算任务试图读取丢失的源头数据时出错，会导致部分 task 计算失败，会进一步导致整个 batch 的 job 失败，最终在 driver 端以 `SparkException` 的形式报出来 —— 此时我们 catch 住这个 `SparkException`，就能够屏蔽这个 batch 的 job 失败了。

粗粒度的这个做法实现起来非常简单，问题是会忽略掉整个 batch 的计算结果。虽然我们还有 6 个 block 是好的，但所有 9 个的数据都会被忽略。

细粒度忽略

细粒度的做法是，只将忽略部分局限在丢失的 3 个 block 上，其它部分 6 部分继续保留。目前原生的 Spark Streaming 还不能完全做到，但我们对 Spark Streaming 稍作修改，就可以做到了。

细粒度基本思路是，在一个计算的 task 发现作为源数据的 block 失效后，不是直接报错，而是另外生成一个空集合作为“修正”了的源头数据，然后继续 task 的计算，并将成功。

如此一来，仅局限在发生数据丢失的 3 个块数据才会进行“忽略”的过程，6 个好的块数据将正常进行计算。最后整个 job 是成功的。

当然这里对 Spark Streaming 本身的改动，还需要考虑一些细节，比如只在 Spark Streaming 里生效、不要影响到 Spark Core、SparkSQL，再比如 task 通常都是会失效重试的，我们希望前几次现场重试，只在最后一次重试仍不成功的时候再进行忽略。

我们把修改的代码，以及使用方法放在这里了，请随用随取。

总结

我们上面分四个小节介绍了 Spark Streaming 对源头数据的高可用的保障方式，我们用一个表格来总结一下：

	图示	优点	缺点
(1) 热备		无 recover time	需要占用双倍资源
(2) 冷备		十分可靠	存在 recover time
(3) 重放		不占用额外资源	存在 recover time
(4) 忽略		无 recover time	准确性有损失

（本文完，参与本文的讨论请 猛戳这里，返回目录请 猛戳这里）