8. Kafka Connect

原文链接 : http://kafka.apache.org/documentation/#connect

译文链接 :8. Kafka Connect（修改该链接为 ApacheCN 对应的译文链接）

贡献者 : @陈留锁，ApacheCN，Apache中文网

8.1 概述

Kafka Connect 是一个可扩展、可靠的在 Kafka 和其他系统之间流传输的数据工具。它可以通过 Connectors（连接器）简单、快速的将大集合数据导入和导出 Kafka。Kafka Connect 可以接收整个数据库或收集来自所有的应用程序的消息到Kafka Topic。使这些数据可用于低延迟流处理。导出可以把 topic 的数据发送到 secondary storage（辅助存储也叫二级存储）也可以发送到查询系统或批处理系统做离线分析。Kafka Connect 功能包括：

- Kafka连接器通用框架：Kafka Connect 规范了 Kafka 与其他数据系统集成，简化了 Connector 的开发、部署和管理。
- 分布式和单机模式：扩展到大型支持整个 organization 的集中管理服务，也可缩小到开发，测试和小规模生产部署。
- REST 接口：通过REST API来提交（和管理） Connector 到 Kafka Connect 集群
- offset自动化管理：从 Connector 获取少量的信息，Kafka Connect 来管理 offset 的提交，所以 Connector 的开发者不需要担心这个容易出错的部分。
- 分布式和默认扩展：Kafka Connect 建立在现有的组管理协议上。更多的工作可以添加扩展到 Kafka Connect 集群。
- 流/批量集成：利用 Kafka 现有的能力，Kafka Connect 是一个桥接流和批量数据系统的理想解决方案。

8.2 用户指南

快速指南提供了一个如何运行单机版 Kafka Connect 的例子。本节主要对如何配置、运行和管理 Kafka Connect 的问题进行详细的阐述。

运行 Kafka Connect

Kafka
Connect目前支持两种执行模式：独立（单进程）和分布式。在独立模式下，所有的工作都在一个单进程中进行的。这样易于配置，在一些情况下，只有一个在工作是好的（例如，收集日志文件），但它不会从kafka
Connection的功能受益，如容错。通过下面的命令开始一个单进程的例子：

```
> bin/connect-standalone.sh config/connect-standalone.properties
Connector1.properties [Connector2.properties ...]
```

第一个参数是 worker 的配置，这包括 Kafka 连接的参数设置，序列化格式，以及频繁地提交 offset（偏移量）。本节提供的例子用的是默认的配置 conf/server.properties。其余的参数是 Connector（连接器）配置文件。你可以配置你需要的，但是所有的执行都在同一个进程（在

不同的线程）。分布式的模式会自动平衡。允许你动态的扩展（或缩减），并在执行任务期间和配置、偏移量提交中提供容错保障，非常类似于独立模式：

```
bin/connect-distributed.sh config/connect-distributed.properties
```

在不同的类中，配置参数定义了 Kafka Connect 如何处理，哪里存储配置，如何分配 work，哪里存储 offset 和任务状态。在分布式模式中，Kafka Connect 在 topic 中存储 offset，配置和任务状态。建议手动创建 offset 的 topic，可以自己来定义需要的分区数和副本数。如果启动 Kafka Connect 时还没有创建 topic，那么 topic 将自动创建（使用默认的分区和副本），这可能不是最合适的（因为 Kafka 可不知道业务需要，只能根据默认参数创建）。特别是以下配置参数尤为关键，启动集群之前设置：

- group.id (默认 connect-cluster)：Connect cluster group 使用唯一的名称；注意这不能和 consumer group ID（消费者组）冲突。
- config.storage.topic (默认 connect-configs)：topic 用于存储 Connector 和任务配置；注意，这应该是一个单个的 partition，多副本的 topic。你需要手动创建这个 topic，以确保是单个 partition（自动创建的可能会有多个partition）。
- offset.storage.topic (默认 connect-offsets) ：topic 用于存储 offsets；这个topic应该配置多个 partition 和副本。
- status.storage.topic (默认 connect-status)：topic 用于存储状态；这个 topic 可以有多个 partitions 和副本

注意，在分布式模式中，Connector（连接器）配置不能使用命令行。要使用下面介绍的 REST API 来创建，修改和销毁 Connector。

配置连接器

Connector 的配置是简单的 key-value 映射。对于独立模式，这些都是在属性文件中定义，并通过在命令行上的 Connect 处理。在分布式模式，JSON 负载 Connector 的创建（或修改）请求。大多数配置都是依赖的 Connector，有几个常见的选项：

- name： 连接器唯一的名称，不能重复。
- Connector.calss：连接器的Java类。
- tasks.max：连接器创建任务的最大数。
- Connector.class 配置支持多种格式：全名或连接器类的别名。比如连接器是 org.apache.kafka.connect.file.FileStreamSinkConnector，你可以指定全名，也可以使用 FileStreamSink或FileStreamSinkConnector。Sink Connector 也有一个额外的选项来控制它们的输入：
- topics：作为连接器的输入的 topic 列表。

对于其他的选项，你可以查看连接器相关文档。

REST API

由于 Kafka Connect 的目的是作为一个服务运行，提供了一个用于管理 Connector 的 REST API。默认情况下，此服务的端口是8083。以下是当前支持的终端入口：

- GET /Connectors：返回活跃的 Connector 列表
- POST /Connectors：创建一个新的 Connector；请求的主体是一个包含字符串name字段和对象 config 字段（Connector 的配置参数）的 JSON 对象。
- GET /Connectors/{name}：获取指定 Connector 的信息
- GET /Connectors/{name}/config：获取指定 Connector 的配置参数
- PUT /Connectors/{name}/config：更新指定 Connector 的配置参数
- GET /Connectors/{name}/status：获取 Connector 的当前状态，包括它是否正在运行，失败，暂停等。
- GET /Connectors/{name}/tasks：获取当前正在运行的 Connector 的任务列表。
- GET /Connectors/{name}/tasks/{taskid}/status：获取任务的当前状态，包括是否是运行中的，失败的，暂停的等，
- PUT /Connectors/{name}/pause：暂停连接器和它的任务，停止消息处理，直到 Connector 恢复。
- PUT /Connectors/{name}/resume：恢复暂停的 Connector（如果 Connector 没有暂停，则什么都不做）
- POST /Connectors/{name}/restart：重启 Connector（Connector 已故障）
- POST /Connectors/{name}/tasks/{taskId}/restart：重启单个任务 (通常这个任务已失败)
- DELETE /Connectors/{name}：删除 Connector, 停止所有的任务并删除其配置

Kafka Connector 还提供了获取有关 Connector plugins 信息的 REST API：

- GET /Connector-plugins：返回已在 Kafka Connect 集群安装的 Connector plugin 列表。请注意，API 仅验证处理请求的 worker 的 Connector。这以为着你可能看不不一致的结果，特别是在滚动升级的时候（添加新的 Connector jar）
- PUT /Connector-plugins/{Connector-type}/config/validate ：对提供的配置值进行验证，执行对每个配置验证，返回验证的建议值和错误信息。

8.3 Connector开发指南

本章节介绍了开发者怎么样编写新的Connector，用于kafka和其他系统之间的数据移动。简要回顾几个关键的概念，然后介绍如何创建一个简单的Connector。

核心概念和API

在Kafka和其他系统之间复制数据，用户创建自定义的从系统中pull数据或push数据到系统的Connector（连接器）。Connector有两种形式：SourceConnectors从其他系统导入数据（如：JDBCSourceConnector将导入一个关系型数据库到Kafka）和SinkConnectors导出数据（如

：HDFSSinkConnector将kafka主题的内容导出到HDFS文件）。Connector不会执行任何复制自己的数据：它们的配置展示了要复制的数据，而Connector是负责打破这一工作变成一组可以分配worker的任务。这些任务也有两种相对应的形式：SourceTask 和 SinkTask。在手里的任务，每个任务必须复制其子集的数据或Kafka的。在Kafka Connect，这些任务作为一组具有一致性模式的记录（消息）组成的输出和输入流。有时，这种映射是明显的：在一组日志文件，每个文件可以被视为一个流，每个分析的行形成一个记录，使用相同的模式和offset存储在文件中的字节偏移。在其他的情况下可能需要更多的努力来映射到该模型：一个JDBC连接器可以将每张表映射到一个流，但offset是不太清楚的。一种可能的映射使用时间戳列来生成查询递增返回新的数据，上次查询时间戳可被用作offset。

Streams and Records（流和记录）

每个流都应该有一个key-value的记录序列。key和value可以具有复杂的结构 — 提供了许多原始类型，但数组、对象和嵌套的数据结构也可以。运行时，数据格式不承担任何特定的序列化格式，这种转换是由框架内部处理的。除了key和value，记录（由源和传递到sink产生的）关联的流ID和offset。这些都是使用了框架。定期提交的offset的数据（已处理的），以便在发生故障时，处理可以从最后一个提交的偏移量恢复，避免不必要的重复处理。

Dynamic Connectors（动态连接器）

并非所有的工作都是静态的，Connector（连接器）的实现还负责监控外部系统（根据外部系统的变化可能需要重新配置）。例如，在JDBCSourceConnector的例子中，Connector可分配一组表到每个任务。当创建一个新的表，它必须要发现这个新表，并更新到配置把新的表分配到任务中。当注意到一个变化，需要重新配置（或任务数量的变化），它通知框架更新相应的任务。

Connector（开发一个简单的连接器）

开发一个连接器只需要实现两个接口，Connector 和 Task。在 Kafka 源代码里 file 包下有一个简单的例子。该 Connector 是用于独立模式，SourceConnector/SourceTask 实现文件每行读取，并作为记录（消息）用SinkConnector/SinkTask 把每个记录写到一个文件。本节的其余部分将通过一些代码来演示创建一个连接器的关键步骤，但开发者也应参考完整的例子的源代码，大部分的细节都略为简单。

Connector例子

我们拿 SourceConnector 作为一个简单的例子。SinkConnector 的实现也非常类似。通过创建一个继承 SourceConnector 的类开始，增加一个字段存储解析的配置信息（文件名读取和发送数据到topic）：

```
public class FileStreamSourceConnector extends SourceConnector {
    private String filename;
    private String topic;
```

最简单的方法是 getTaskClass()，它定义了在工作进程中实例化的实际读取数据的类：

```
@Override
public Class<? extends Task> getTaskClass() {
    return FileStreamSourceTask.class;
}
```

定义 FileStreamSourceTask 类，接下来，我们增加一些标准的生命周期的方法，start()和stop()：

```
@Override
public void start(Map<String, String> props) {
    // The complete version includes error handling as well.
    filename = props.get(FILE_CONFIG);
    topic = props.get(TOPIC_CONFIG);
}

@Override
public void stop() {
    // Nothing to do since no background monitoring is required.
}
```

最后，是实现真正核心的 getTaskConfigs()。在这种情况下，我们只处理一个文件，这样即使我们生成更多的任务（根据 maxTask 参数），我们返回一个列表，只有一个入口：

```
@Override
public List<Map<String, String>> getTaskConfigs(int maxTasks) {
    ArrayList>Map<String, String>> configs = new ArrayList<>();
    // Only one input stream makes sense.
    Map<String, String> config = new Map<>();
    if (filename != null)
        config.put(FILE_CONFIG, filename);
    config.put(TOPIC_CONFIG, topic);
    configs.add(config);
    return configs;
}
```

虽然在本例中未使用，SourceTask也提供了两个API来提交源系统的offset：commit 和 commitRecord。提供了有消息确认机制的源系统API。重写这些方法，允许source Connector（源连接器）在源系统应答消息。一旦他们写入到Kafka，无论消息是成批的还是单独。commit API在源系统存储offset，由poll返回offset。这个API的实现是阻塞的，直到提交完成。commitRecord API为在源系统中的每个写入到Kafka之后的SourceRecord保存offset，Kafka Connect自动记录offset。SourceTasks不需要实现。在Connector需要确认在源系统acknowledge（应答）消息的情况下，即使有多个任务，这种方法实现通常是非常简单的，只需要一个API。它只确定输入任务的数量，这可能需要它从远程服务提取数据。然后瓜分数据。由于一些模式之间分配work（工作）非常普遍，有些实用工具提供了ConnectorUtils来简化这些情况，注意，这个简单的例子不包括动态输入。详见在下一节讨论如何触发更新任务配置。

Task例子 - Source Task

接下来我们将介绍对应的SourceTask的实现。我们将使用伪代码来展示大部分的实现，你可以参考完整的示例的源代码。和连接器一样，我们需要创建一个类（继承基于Task的类）。它也有一些标准的生命周期方法：

```
public class FileStreamSourceTask extends SourceTask<Object, Object> {
    String filename;
    InputStream stream;
    String topic;

    public void start(Map<String, String> props) {
        filename = props.get(FileStreamSourceConnector.FILE_CONFIG);
        stream = openOrThrowError(filename);
        topic = props.get(FileStreamSourceConnector.TOPIC_CONFIG);
    }

    @Override
    public synchronized void stop() {
        stream.close();
    }
```

稍微简化了一下，说明这些方法是比较简单的，仅分配或释放资源。有两个点需要注意。首先，start()方法还未处理以前offset的恢复，这将在后面的部分讨论，其次，stop()方法是同步的。SourceTasks提供了专门的线程，可以无限期的阻塞。所以需要从别的Worker线程来停止。接下来，我们实现任务的主要功能，poll()方法。它从输入系统获取时间并返回一个List。

```
@Override
public List<SourceRecord> poll() throws InterruptedException {
    try {
        ArrayList<SourceRecord> records = new ArrayList<>();
        while (streamValid(stream) && records.isEmpty()) {
            LineAndOffset line = readToNextLine(stream);
            if (line != null) {
                Map<String, Object> sourcePartition =
Collections.singletonMap("filename", filename);
                Map<String, Object> sourceOffset =
Collections.singletonMap("position", streamOffset);
                records.add(new SourceRecord(sourcePartition,
sourceOffset, topic, Schema.STRING_SCHEMA, line));
            } else {
                Thread.sleep(1);
            }
        }
        return records;
    } catch (IOException e) {
        // Underlying stream was killed, probably as a result of calling
stop. Allow to return
        // null, and driving thread will handle any shutdown if
necessary.
    }
    return null;
}
```

同样，我们省略了一些细节，我们可以看到重要的步骤：poll()方法反复的调用，并每次调用都会尝试从文件中读取记录（消息）。读取每一行，也跟踪文件的offset。它使用该信息来创建一个输出SourceRecord和四条信息：源分区（只有1个，读取单个文件），源offset（在文件中的字节offset），输出topic的name，和输出value（行，包括一个模式，表示value始终是一个string）。SourceRecord构造函数的其他实现也

包括一个指定的输出分区和key。请注意，此实现使用正常的Java InputStream接口，如果数据不可用则可以sleep（休眠）。这个可以接受，因为Kafka Connect为每个任务提供了一个专用的线程。而任务实现必须基于poll()接口，这样有跟多的灵活性（自己实现）。在这种情况下，基于NIO的实现会更有效，但方法简单，快速实现，并兼容老版本（Java）。

Sink Tasks

前面已经介绍了如何实现一个简单的SourceTast。不像SourceConnector 和 SinkConnector, SourceTask 和 SinkTask有很多不同的接口，因为SourceTask使用pull接口，SinkTask使用push接口。两者都有共同的生命周期的方法，但是SinkTask接口是完全不同的：

```
public abstract class SinkTask implements Task {
    public void initialize(SinkTaskContext context) {
        this.context = context;
    }
    public abstract void put(Collection<SinkRecord> records);
    public abstract void flush(Map<TopicPartition, Long> offsets);
```

SinkTask文档有全部的细节，但接口和SourceTask一样简单。put()方法包含大部分的实现，接收集合SinkRecords，执行转换，并存储到目标系统。这个方法不需要确保返回之前数据完全写入到目标系统。事实上，在大部分情况下，内存缓冲是有用的，这样记录可以按一个整批次一次发送，从而减少插入事件进入downstream（下游）数据存储的开销。SinkRecord作为SourceRecords包含相同的信息：Kafka topic，partition，offset和事件key和value。flush()方法在offset提交过程期间，它允许任务从故障中恢复，并从安全点恢复（这样就没有事件会被错过）。该方法应该将任何未完成的数据push到目标系统，然后阻塞，直到写入已得到确认。通常offset参数可以忽略，但在某些情况下，想要实现存储offset信息到目标系统以提供正好一次的语义。例如，HDFS Connector（连接器）可以做到这一点，使用原子移动操作来确保flush()的原子性，确保提交数据和offset到最终的位置（HDFS）。

从之前的offset恢复（Resuming from Previous Offsets）

SourceTask包含一个流ID（输入的文件名）和每个记录的offset（文件中的位置）。框架使用了定时提交offset，所以在故障的情况下，任务恢复并减少再处理和可能重复的事件数（如果Kafka Connect正常的停止，可从最近的offset恢复，例如在独立模式或重新加载配置）。提交处理是完全自动化的，但只有Connector知道如何返回到正确的位置，从该位置恢复。正确的恢复后，任务可以使用SourceContext传递其initialize()方法来访问offset数据。在initialize()中，我们会添加一些代码来读取offset（如果存在），并找到它的位置。

```
stream = new FileInputStream(filename);
    Map<String, Object> offset =
context.offsetStorageReader().offset(Collections.singletonMap(FILENAME_F
IELD, filename));
    if (offset != null) {
        Long lastRecordedOffset = (Long) offset.get("position");
        if (lastRecordedOffset != null)
            seekToOffset(stream, lastRecordedOffset);
    }
```

当然，你可能需要为每个输入流读取大量的key。OffsetStorageReader接口也允许批量读取（有效的负载所有的offset），然后找出每个输入流到合适的位置。

动态的输入/输出流 (Dynamic Input/Output Streams)

Kafka Connect的工作被定义为拷贝大量数据。如拷贝一个完整的数据库，而不是创建多个job来分别复制每一张表。这种设计的后果是，一个Connector的输入或输出流集合可以随着时间的推移而变化。Source Connector需要监听源系统的改变。例如：数据库表的增加/删除。当发现改变，通过ConnectorContext对象通知框架，来重新加载。例如，在SourceConnector：

```
if (inputsChanged())
        this.context.requestTaskReconfiguration();
```

该框架将立即请求新配置并更新任务，在重新加载配置之前优雅的提交自己的进度。注意，SourceConnector检测目前留给Connector实现，如果需要一个额外的线程执行此监控。那么Connector必须分配它自己。理想的情况下，监控变更代码将会隔离Connector和任务，不需担心

。然而，变更也可能影响任务，最常见的是，当其中一个输入流在输入系统销毁了。例如：如果一张表从数据库中删除。如果任务在Connector之前遇到问题，如果Connector需要poll（轮询）变更，则任务将需要处理随后的错误。这些都是常见的问题。值得庆幸的是，这可以通过简单catch和处理相应的异常。SinkConnectors通常

只能处理流的增加，它可以转换输出新的entires（例如，一个新的数据库表）。该框架管理对Kafka的输入的任何变更。例如当输入的topic集变化（由一个正则表达式的订阅）。
SinkTasks等待新的输入流，它可能需要在下游系统创建新的资源。比如数据库中的新表，最棘手的情况是在这种情况产生的冲突（多个SinkTask看到一个新的输入流并同时尝试去创建新的资源）。SinkConnectors，另一方面，一般不需要特殊的代码来处理一组动态的流。

连接配置验证（Connect Configuration Validation）

Kafka
Connect允许你在提交要执行的Connector之前来验证Connector的配置，并可以提供故障和推荐值的反馈。利用这个优势，Connector开发者需要提供一个config()的实现来暴露配置给框架。下面的代码在FileStreamSourceConnector定义配置和暴露给框架。

```
private static final ConfigDef CONFIG_DEF = new ConfigDef()
        .define(FILE_CONFIG, Type.STRING, Importance.HIGH, "Source
filename.")
        .define(TOPIC_CONFIG, Type.STRING, Importance.HIGH, "The topic
to publish data to");

    public ConfigDef config() {
        return CONFIG_DEF;
    }
```

ConfigDef类用于指定预期的配置集，对于每个配置，你可以指定name，type，默认值，描述，group信息，group中的顺序，配置值的宽和适于在UI显示的名称。另外，你可以通过重写Validator类来指定的验证逻辑用于单个配置验证。此外，由于配置之间有可能存在依赖关系。例如，配置中的vaild和visibillty的值可能会根据其他的配置的值而变化。为了解决这个问题，ConfigDef允许你指定一个配置依赖，并提供推荐系统的实现来获取valid的值并设置visibillty得到的当前配置值。此外，Connector的validate()方法提供了一个默认验证实现，返回一个列表（返回允许配置的列表，每个配置的配置错误和推荐值）。然后，它不适用配置验证的推荐值。你可以提供一个自定义的配置验证覆盖的默认实现，这可能会使用建议的值。

Working with Schemas

FileStream Connector是很好的例子，因为它很简单，但是也有很普通的结构化数据 —
每行只有一个字符串（string），实际Connector都需要更复杂的数据格式模式，要创建更复杂的数据，你需要使用Kafka Connect data API。除了原始类型的影响，大多数记录的结构需要2个类：Schema和Struct。API文档有完整参考。这里是一个简单的例子，创建一个Schema和Struct：

```
Schema schema = SchemaBuilder.struct().name(NAME)
    .field("name", Schema.STRING_SCHEMA)
    .field("age", Schema.INT_SCHEMA)
    .field("admin", new
SchemaBuilder.boolean().defaultValue(false).build())
    .build();

Struct struct = new Struct(schema)
    .put("name", "Barbara Liskov")
    .put("age", 75)
    .build();
```

如果你实现一个source
Connector，你需要决定何时以及如何创建schema。如果可能的话，你应该尽量避免重复计算。例如，如果你的Connector保证有一个固定的schema，用静态和使用单例，然而，大部分Connector有动态的schema。一个简单的例子，一个数据库Connector。甚至只考虑一张表，这个schema不会预定义整个Connector（因为它表到表的变化），但它也不会固定为单表生命周期中，因为用户可能会ALTER
TABLE（修改表）。Connector必须能够检测这些变化并作出反应，Sink
Connector之所以简单，是因为它们消费数据，不需要创建shema。但是，它们应该同样的去关心验证它们收到的schema的格式是预期的。当schema不匹配 — 通常表示上游的生产者生产无效的数据不能被正确的转换到目标系统 — sink Connectors抛出一个异常给系统。

Kafka Connect管理（Kafka Connect Administration）

Kafka
Connect的REST层提供了一组API来管理集群。这包括查看Connector的配置和任务的状态，以及改变其当前的行为（例如改变配置和重新启动任务）。
当一个Connector第一次被提交到集群，worker重新平衡集群中全部的Connector和它们的任务。使每个worker具有大致相同的工作量。当Connector递增和减少任务数，或Connector配置发生变化时，也使用了同样的重新平衡程序。你可以使用REST
API查看Connector当前的状态和任务，包括每个分配worker的id。例如，查询一个源文件的状态（使用 GET
/Connectors/file-source/status）可能会产生如下的输入：

```
{
  "name": "file-source",
  "Connector": {
    "state": "RUNNING",
    "worker_id": "192.168.1.208:8083"
  },
  "tasks": [
    {
      "id": 0,
      "state": "RUNNING",
      "worker_id": "192.168.1.209:8083"
    }
  ]
}
```

Connector和它们的任务发布状态状态更新到共享topic（配置status.storage.topic）集群监控中的所有的worker。因为woker异步消费这个topic，在一个状态改变之前，有一个典型的（短）延迟是可见的（通过状态API）。下列的状态可能是Connector或是其任务之一：

- UNASSIGNED: Connector/task 还未分配给worker.
- RUNNING: Connector/task 正在运行.
- PAUSED: The Connector/task has been administratively paused.
- FAILED: Connector/tast故障（通常是抛出一个异常，状态输出报告）。

在多数情况下，Connector和任务状态将匹配，尽管他们可能短时间不同（当发生变化或任务故障）。例如，当一个Connector刚启动时，Connector和其任务转换到运行状态之前，可能有明显的延时。当任务故障（因为Connect不会自动重启故障的任务）
状态还会出现分歧。手动的重启Connector/任务时，可以使用上面列出的重启API。注意，如果你尝试去重启任务（这个任务正在rebalance），Connect将会返回一个409（冲突）状态代码。你可以在rebalance完成之后重试，但是没有必要，因为rebalance有效地重新启动集群中的所有Connector和任务。

有时可以暂时的停止Connector的消息处理。例如，如果远程系统正在维护，最好source
Connector停止poll，而不是一直报错误的异常日志刷屏。对于这个用例，Connect提供了一个暂停/恢复（pause/resume）的API。虽然source Connector暂停了，Connect将停止poll额外的记录。当sink
Connector被暂停时，Connect将停止向它推送新的消息。暂停状态是持久性的。所以即使你重新启动集群，Connector也不会再次启动消费处理，直到任务恢复。注意，Connector的任务转换到PAUSED状态时有可能会有延迟。因为它需要在暂停的期间来完成所有处理。另外，失败的任务不会转换到PAUSED状态，直到它们重新启动。