Apache nifi

开发指南

**版本：V 1.6**

**日期：2018年6月13日**

# Apache Nifi 概念

## NiFi简介

  Apache NiFi 是一个易于使用、功能强大而且可靠的数据拉取、数据处理和分发系统，用于自动化管理系统间的数据流。它支持高度可配置的指示图的数据路由、转换和系统中介逻辑，支持从多种数据源动态拉取数据。NiFi原来是NSA的一个项目，目前已经代码开源，是Apache基金会的顶级项目之一。

NiFi是基于Java的，使用Maven支持包的构建管理。 NiFi基于Web方式工作，后台在服务器上进行调度。用户可以为数据处理定义为一个流程，然后进行处理，后台具有数据处理引擎、任务调度等组件。

## Nifi核心概念

FlowFile：表示通过系统移动的每个对象，包含数据流的基本属性

FlowFile Processor（处理器）：负责实际对数据流执行工作

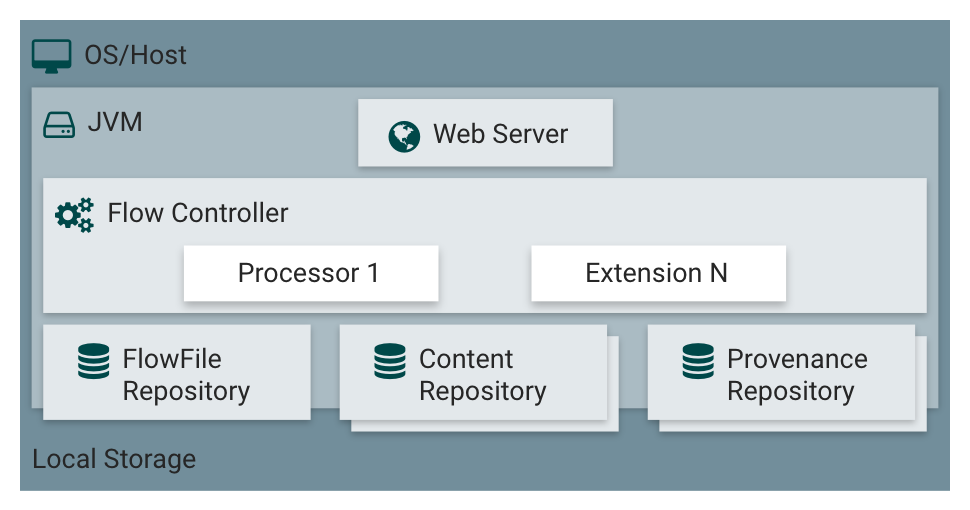
Connection（连接线）：负责不同处理器之间的连接，是数据的有界缓冲区

Flow Controller（流量控制器）：管理进程使用的线程及其分配

Process Group（过程组）：进程组是一组特定的进程及其连接，允许组合其他组件创建新组件

## NIFI架构

NiFi是基于Java的，NiFi的核心部件在JVM里的位置如下图所示：

NiFi在主机操作系统上的JVM内执行。JVM上的NiFi的主要组件如下：

3.1 网络服务器

Web服务器的目的是托管NiFi的基于HTTP的命令和控制API。

3.2 流控制器

流控制器是操作的大脑。它提供用于扩展程序运行的线程，并管理扩展程序接收资源以执行的时间表。

3.3 扩展

有各种类型的NiFi扩展在其他文档中描述。这里的关键是扩展在JVM中运行和执行。

3.4 FlowFile存储库

FlowFile存储库是NiFi跟踪目前在流程中活动的给定FlowFile的知识状态。存储库

实现是可插拔的。默认方法是位于指定磁盘分区上的持久写入前端日志。

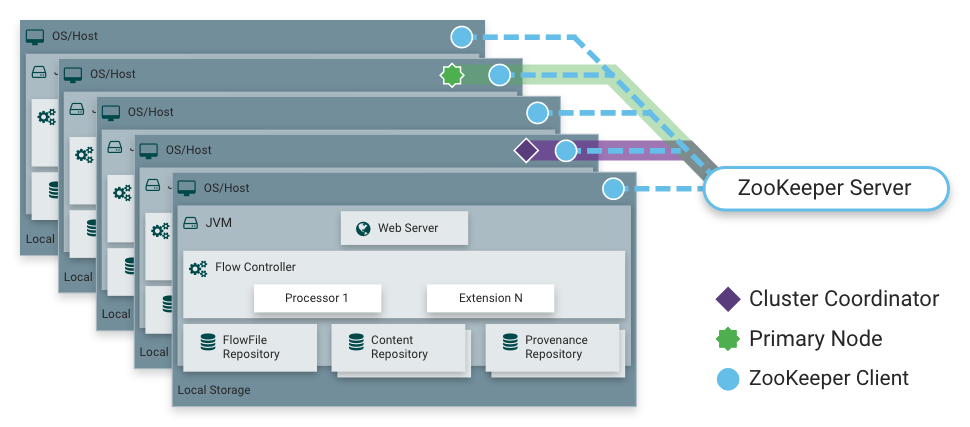
3.5 内容存储库

Content Repository是给定FlowFile的实际内容字节。存储库的实现是可插拔的。默认方法是一个相当简单的机制，它将数据块存储在文件系统中。可以指定多个文件系统存储位置，以便获得不同的物理分区，以减少任何单个卷上的争用。

3.6 源头存储库

Provenance Repository是存储所有来源的事件数据的地方。存储库构造是可插入的，默认实现是使用一个或多个物理磁盘卷。在每个位置内，事件数据被索引和可搜索。

3.7 作为功能强大的数据处理和分发组件，NiFi自然原生支持集群部署方式（推荐部署方式）。NiFi集群部署模式如下图：



集群模式下，NiFi集群中的每个节点对数据执行相同的任务，但是每个节点都在不同的数据集上进行操作。和大部分大数据组件一样，NiFi集群使用Apache ZooKeeper提供协调服务。 Apache ZooKeeper选择一个NiFi节点作为集群协调器，故障转移由ZooKeeper自动处理。 所有集群节点向集群协调器报告心跳和状态信息。集群协调器负责节点的断开和连接。 此外，ZooKeeper会为每个集群选举一个节点作为集群主节点。 作为DataFlow管理器，您可以通过任何节点的用户界面（UI）与NiFi集群进行交互。您所做的任何更改都会同步到集群中的所有节点，从而允许多个入口点。

# NiFi的搭建

## 单机开发环境搭建

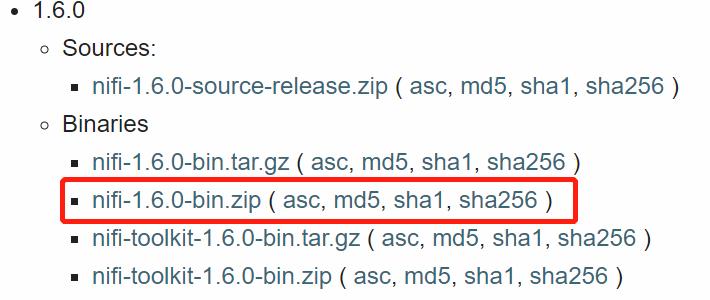
1运行环境准备。

Apache nifi即可运行在Windows平台，也可运行在Linux平台，需要安装jdk（nifi 1.x以上需要jdk8以上，0.x需jdk7以上）和maven（至少3.1.0以上版本）。

2下载

NIFI下载地址：<http://nifi.apache.org/download.html>

下载当前版本的NiFi二进制工程，目前最新的版本为1.6.0。



3支持浏览器：

· Internet Explorer 9+ (see note below)

· Mozilla FireFox 24+

· Google Chrome 36+

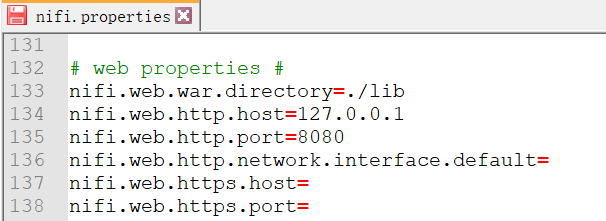
· Safari 8

4修改配置文件。

由于NIFI默认端口为8080，所以需要检查一下8080端口是否被占用，如果被占用可以使用别的未被占用的端口，如9090,9091等。

　检查端口是否被占用命令：netstat -ano|findstr "8080"

　NIFI配置文件：/usr/local/conf/nifi.properties,配置ip（134行nifi.web.http.host）和端口（135行：nifi.web.http.port）



5启动服务。

在linux平台，启动服务使用命令({NIFI ROOT})/bin/nifi.sh start;

在window平台使用命令{NIFI ROOT})\bin\run-nifi.bat。

（双击启动文件：({NIFI ROOT})\bin\run-nifi.bat）

6验证测试

启动服务后过大概3到5分钟，在浏览器中输入：http://localhost/nifi  或者：<http://localhost:8080/nifi>，即可开始使用了。

7基本命令

启动：./nifi.sh start

关闭：./nifi.sh stop

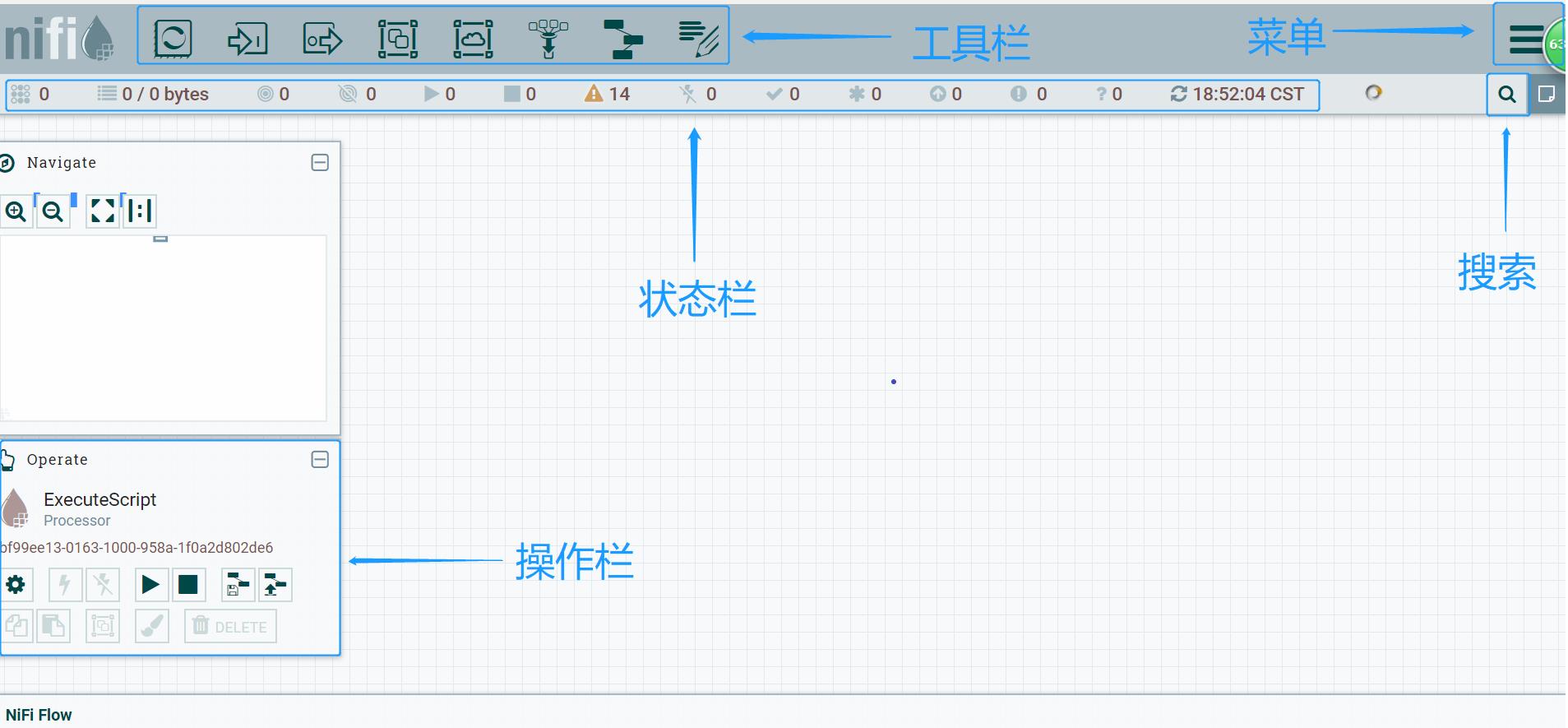
重启：./nifi.sh restart

状态：./nifi.sh status

报表：./nifi-app.log

8、NiFi的操作

（1）UI界面介绍



· 工具栏  
这里主要是构造数据流操作的主要面板。

20171015225641877 添加模块（processor）   
nifi内部会提供各个处理模块，当我们在进行数据处理的过程中，可以选择不同的模块并调整变量进行拼装，从而组合成一个完整的数据流处理的组。

20171015230227849 添加数据流传入点（input-port）   
虽说是数据流输入点，但是并不是整体数据流的起点。它是作为组与组之间的数据流连接的传入点与输出点。

20171015230805490 添加数据流输出点（output-port）   
同理上面的输入点。它是作为组与组之间的数据流连接的传入点与输出点。

20171015231245418 添加组（process-group）   
组相当于系统中的文件夹，作用就是使数据流的各个部分看起来更工整，思路更清晰，不至于从头到尾一条线阅读起来十分不方便。

20171015231805184 添加远端的组（remote process-group）   
根据弹出框进行信息配置，可加入远程的组。

20171015231923316拉取已有的文件（template）   
每当做好一个完整的数据流后，可存储到本地为xml文件，nifi支持本地的template上传，这个按钮就是在上传本地template之后，选择上传过的一个获取到操作画布上。

20171015232528999添加便签（label）   
相当于便签，可放置在画布空白处，写上备注信息。

· Navigate  
这一部分是对区域一这个画布的缩小预览，点击放大缩小可调整视野，蓝框区域就是画布当前的界面，可用鼠标在这部分进行移动从而调整画布的视野。

· 操作栏

20171015233605562开始运行   
选中模块并点击运行按钮，开始进行对数据流的处理。

20171015233740804停止运行   
选中模块并点击停止按钮，则停止了进行对数据流的处理。

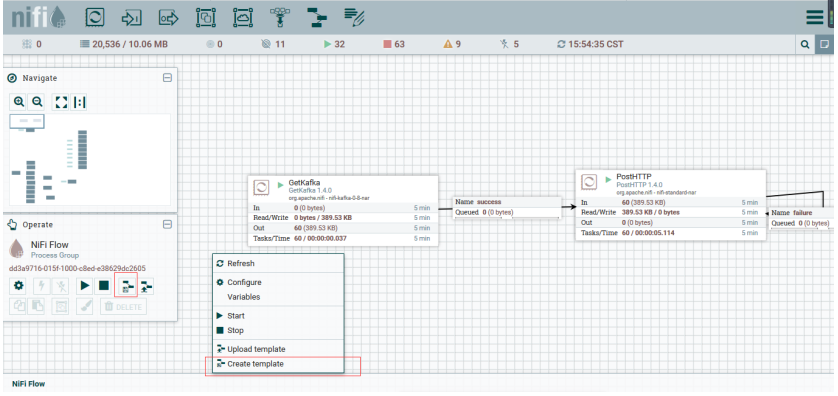
20171015233828318保存template   
选择你要保存的一个template，点击这个保存按钮，可把这个template保存到

nifi系统里（并不是电脑本地，如果想保存到电脑本地，可点击右上角20171015234227928这个按钮，选择Template，弹出的页面上有下载选项）。

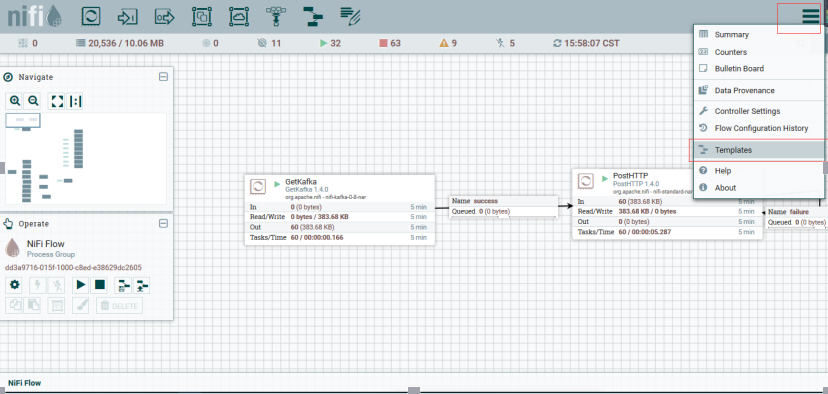
20171015234337591上传template   
可上传本地的template（xml文件）到nifi系统里。

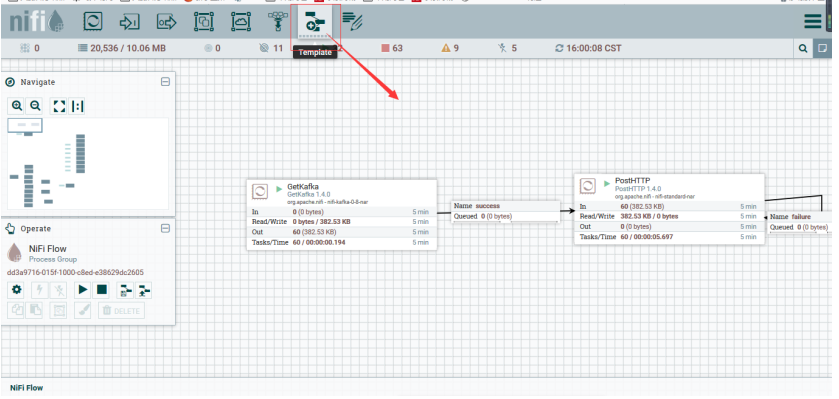
（2）模板

创建模板：在要创建模板的group中点击模板左侧的create template或者鼠标右键空白处。



下载模板：

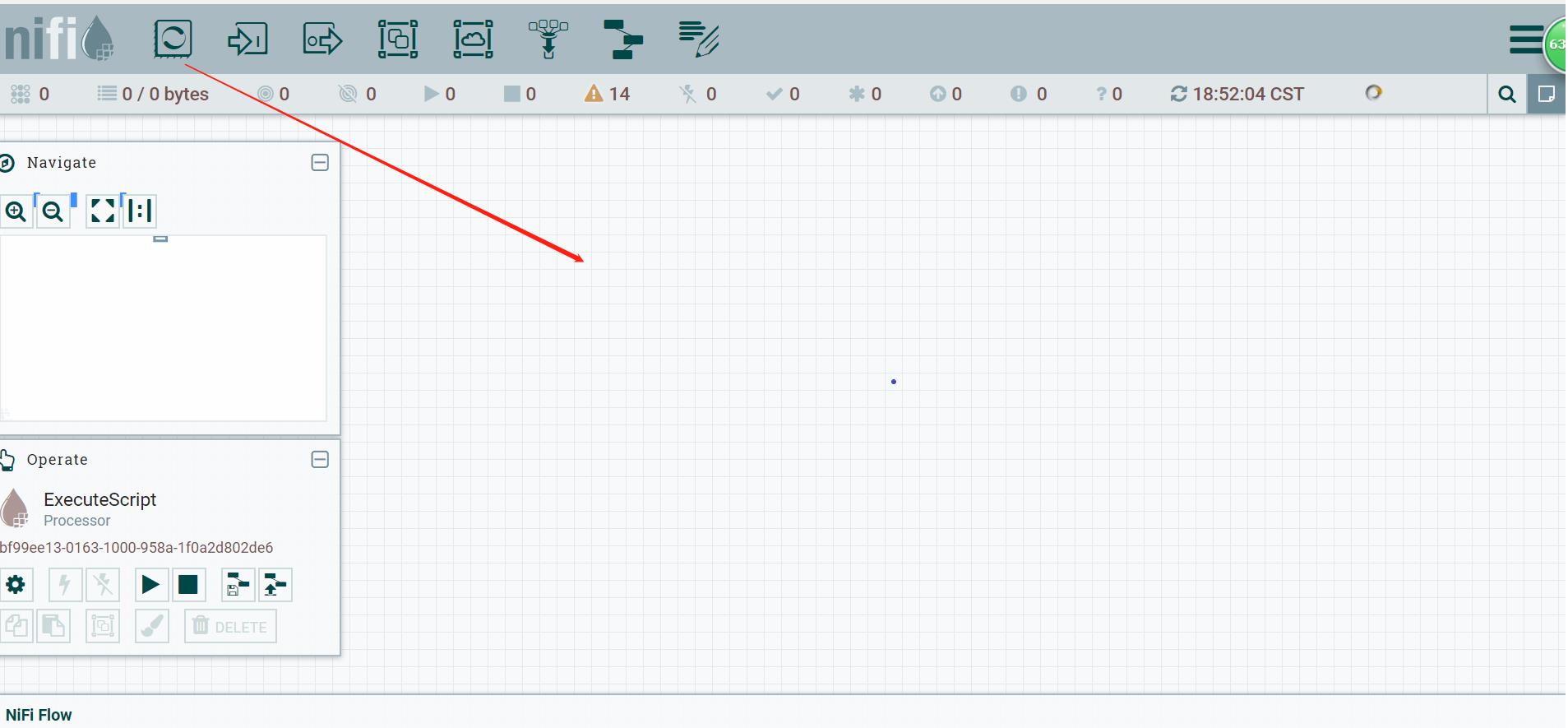
  
使用模板：选择界面上方的template拖放至画布，选择要使用的模板。



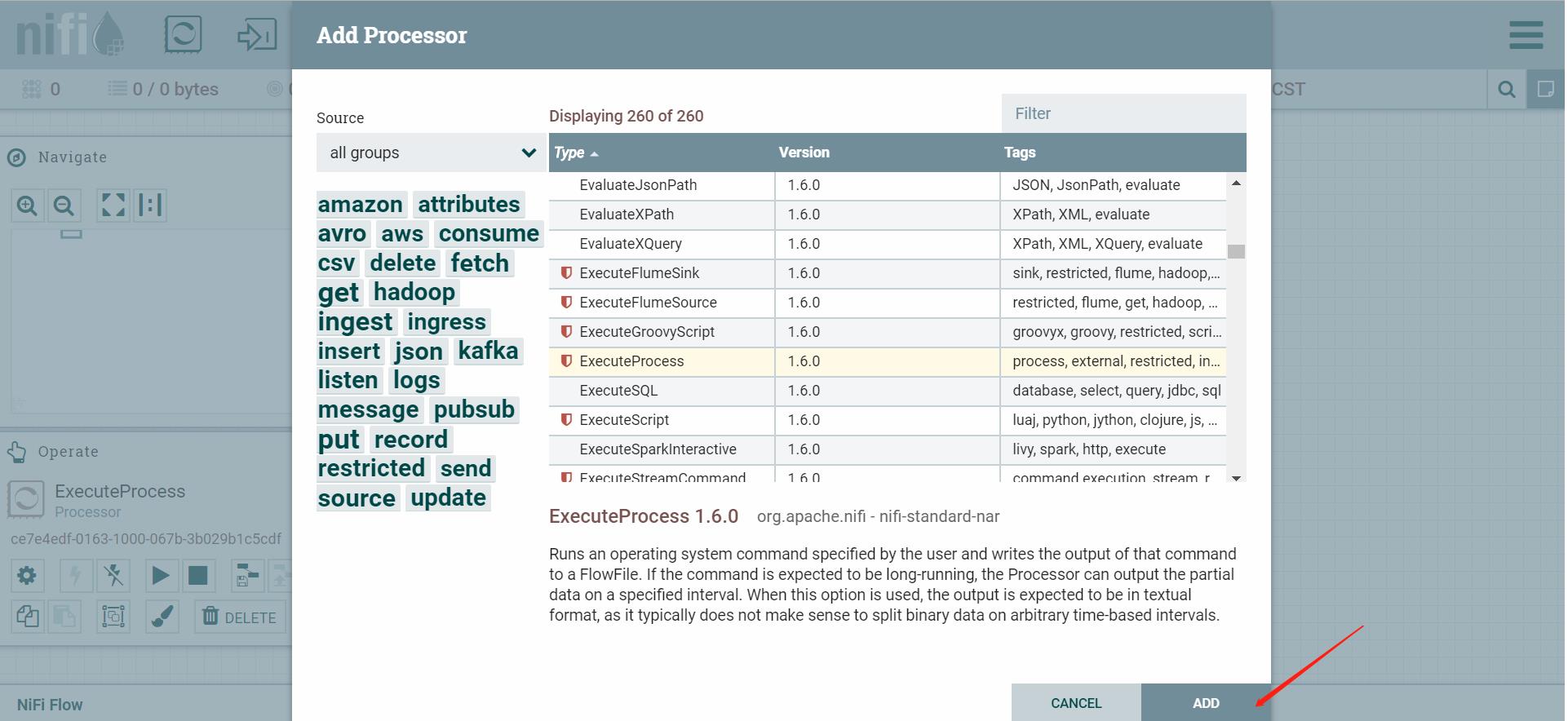
NiFi的模板会保存组中的处理器配置及controller servres 。例如数据库连接，但是不会保存密码。

1. Processor

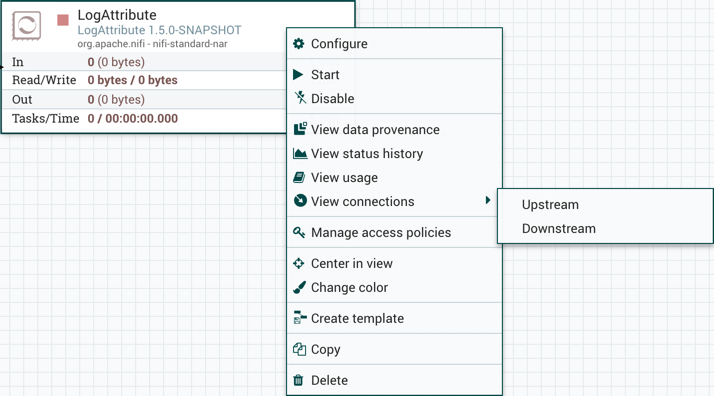
添加处理器：



点击add



将处理器拖到画布上后，可以通过右键单击处理器并从上下文菜单中选择一个选项来与其进行交互。根据分配给您的权限，上下文菜单中可用的选项会有所不同。



虽然上下文菜单中的选项有所不同，但是当您具有使用处理器的完全权限时，通常可以使用以下选项：

· Configure（配置）：此选项允许用户建立或更改处理器的配置。

· Start（启动或停止）：此选项允许用户启动或停止处理器; 该选项可以是Start或Stop，具体取决于处理器的当前状态。

· Disable（启用或禁用）：此选项允许用户启用或启用处理器; 该选项将为“启用”或“禁用”，具体取决于处理器的当前状态。

· View data provenance（查看数据来源）：此选项显示NiFi数据来源表，其中包含有关通过该处理器路由的FlowFiles的数据来源事件的信息。

· View status history（查看状态历史记录）：此选项打开处理器统计信息随时间的图形表示。

· View usage（查看用法）：此选项将用户带到处理器的使用文档。

· View connection → Upstream（查看连接→上游）：此选项允许用户查看和“跳转”

入处理器的上游连接。当处理器连接进出其他进程组时，这尤其有用。

· View connection → Downstream（查看连接→下游）：此选项允许用户查看和“跳转”

到处理器外的下游连接。当处理器连接进出其他进程组时，这尤其有用。

· Centere in view（视图中心）：此选项将画布的视图置于给定的处理器上。

· Change color（更改颜色）：此选项允许用户更改处理器的颜色，这可以使大流量的可视化管理更容易。

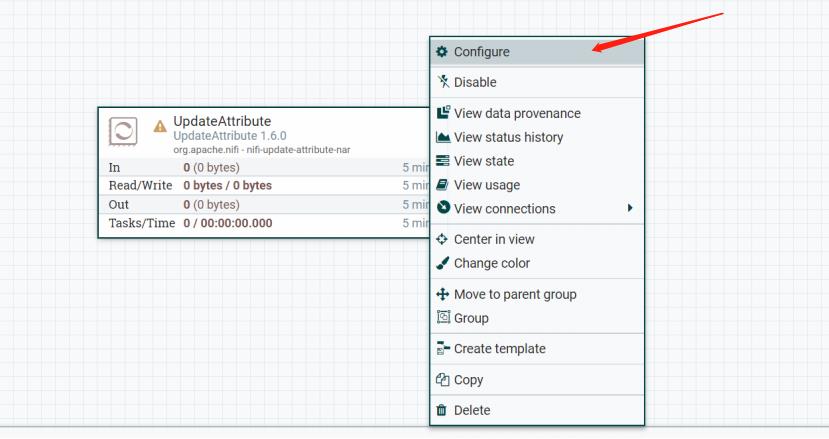
· Create template（创建模板）：此选项允许用户从所选处理器创建模板。

· Copy（复制）：此选项将所选处理器的副本放在剪贴板上，以便可以通过右键单击画布并选择“粘贴”将其粘贴到画布上的其他位置。复制/粘贴操作也可以使用按键Ctrl-C（Command-C）和Ctrl-V（Command-V）完成。

· Delete（删除）：此选项允许从画布中删除处理器。

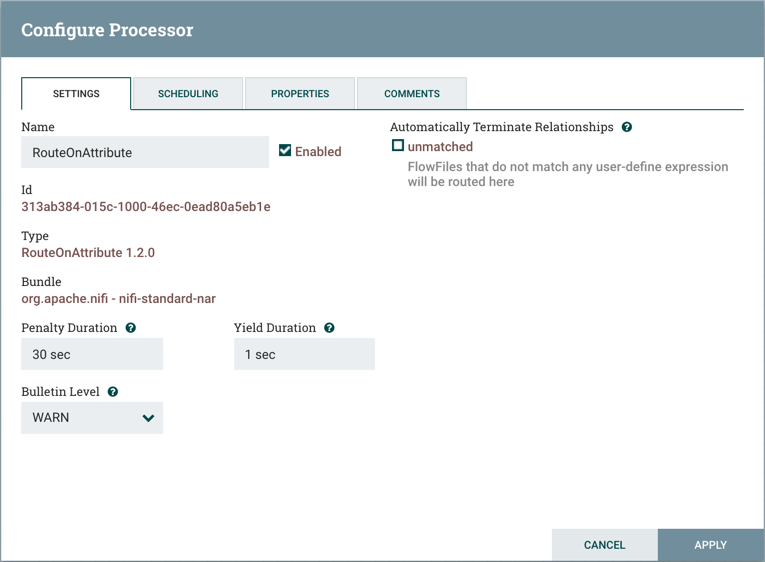
（4）配置processor属性

要配置处理器，请右键单击处理器，然后Configure从上下文菜单中选择该选项。或者，只需双击处理器即可。



#### · 设置选项卡

#### “处理器配置”对话框中的第一个选项卡是“设置”选项卡



Name：Processor名称，默认与处理器类型相同，可以更改。处理器名称旁边是一个复选框，指示处理器是否已启用。

Id：Processor唯一标识符以及Processor的类型和NAR包，无法修改。

Type：Processor类型，无法更改。

Bundle：Processor 的NAR包，无法更改。

Penalty Duration（惩罚持续时间）：在处理一段数据（FlowFile）的正常过程期间，可能发生事件，该事件指示此时不但是数据可以在稍后的时间处理。

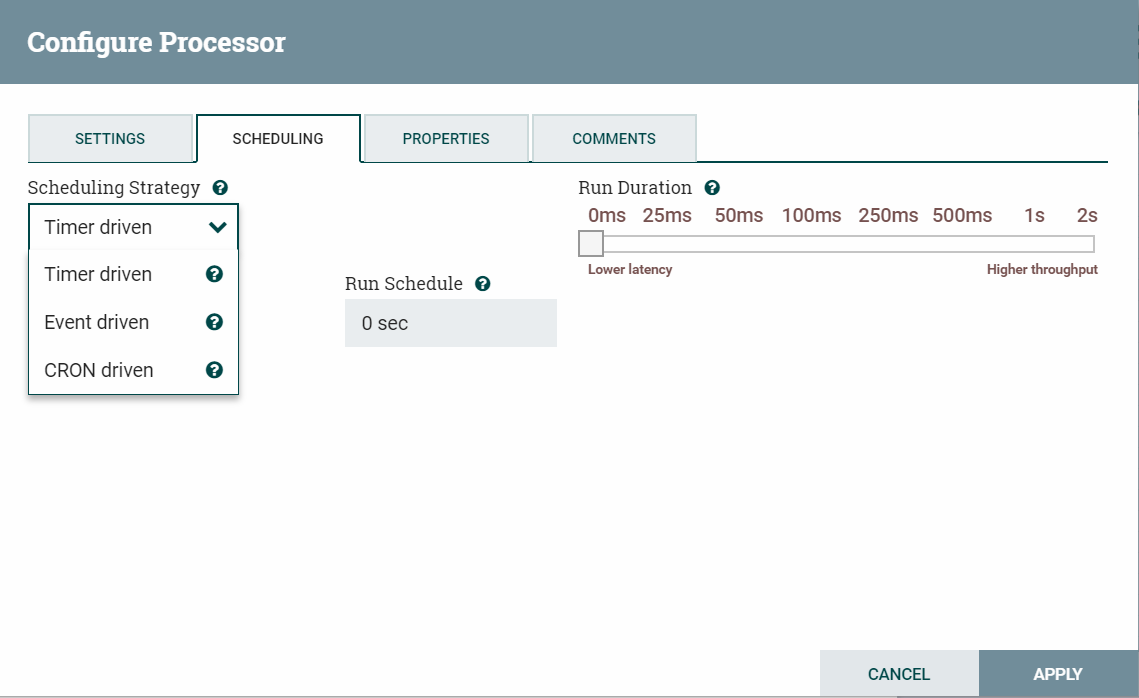
Yield Duration:：处理器可以确定存在某种情况，使得处理器不再能够进行任何进展，而不管其正在处理的数据，这将阻止处理器被安排运行一段时间。

Bulletin Level（公告）：每当处理器写入其日志时，处理器也将生成公告。此设置指示应在用户界面中显示的最低级别的公告。默认情况下，公告级别设置为WARN，这意味着它将显示所有警告和错误级别公告。

Automatically Terminate Relationships（自动终止关系）：为了使处理器被视为有效且能够运行，处理器定义的每个关系必须连接到下游组件或自动终止。

#### · 调度选项卡

“处理器配置”对话框中的第二个选项卡是“计划”选项卡：

NiFi支持三种调度策略，包括Time Driven（时间驱动）、CRON Driven（CRON驱动）和Event Driven（事件驱动，非可选）：

Time Driven：这是默认模式。处理器将安排定期运行。处理器运行的时间间隔由“运行计划”选项定义。

Event Driven：当选择此模式时，处理器将被触发以事件运行，并且当FlowFiles输入连接到此处理器的连接时会发生该事件。此模式目前被认为是实验性的，并且不受所有处理器的支持。选择此模式时，“运行计划”选项不可配置，因为处理器不会触发为定期运行，而是作为事件的结果。此外，这是“并行任务”选项可以设置为0的唯一模式。在这种情况下，线程数量仅受管理员配置的事件驱动线程池大小的限制。

CRON驱动：当使用CRON驱动的调度模式时，处理器被安排定期运行，类似于定时器驱动的调度模式。然而，CRON驱动模式提供了更大的灵活性，但增加了配置的复杂性。CRON驱动的调度值是由六个必填字段和一个可选字段组成的字符串，每个字段由一个空格分隔。

CRON的各参数含义分别代表：秒、分、时、日、月、周、年，需要配合\*、？和L共同执行（\*代表字段的值都有效；?代表对于指定的字段不指定值；L代表长整形）。如：“0 0 13 \* \* ?”代表想要在每天下午1点进行调度执行。根据业务需求进行参数的调度配置。

详情请参阅Quartz文档中的[Chron Trigger教程](http://www.quartz-scheduler.org/documentation/quartz-2.x/tutorials/crontrigger.html" \t "http://nifi.apache.org/docs/nifi-docs/html/_blank)。

<http://www.quartz-scheduler.org/documentation/quartz-2.x/tutorials/crontrigger.html>

· 属性选项卡

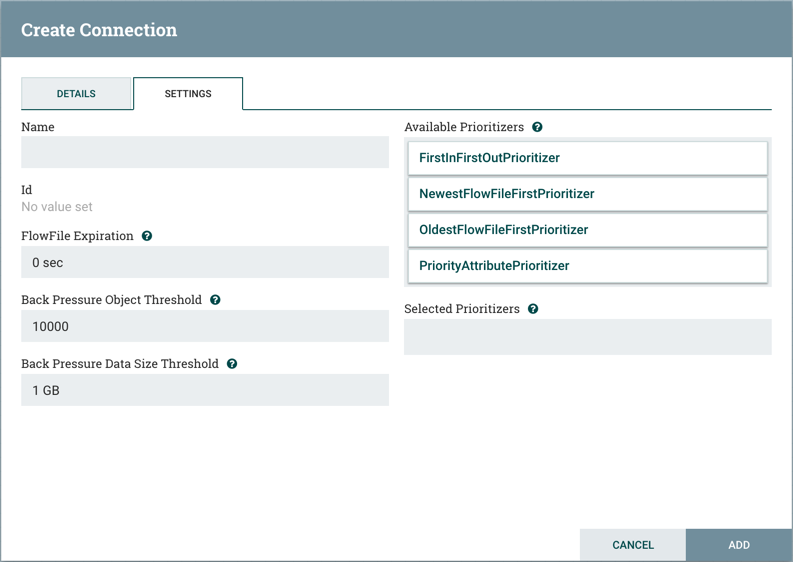
Properties选项卡提供了一种配置特定于Processor的行为的机制。

（5）连接processor

一旦处理器和其他组件被添加到画布中并进行配置，下一步就是将它们彼此连接起来，以便NiFi知道在处理完每个FlowFile后如何处理。这是通过在每个组件之间创建一个连接来完成的。用户将连接气泡从一个组件拖动到另一个组件，直到第二个组件被突出显示。当用户释放鼠标时，会出现一个“创建连接”对话框。必须至少选择一个关系。如果只有一个关系可用，则会自动选择它。

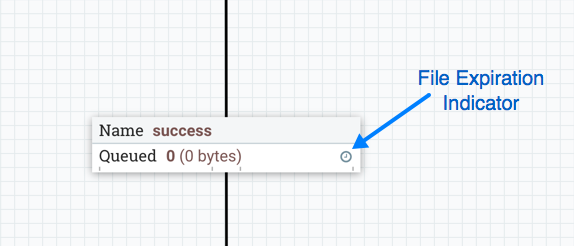
设置

“设置”选项卡提供配置连接名称，FlowFile到期，背压阈值和优先级的功能：



· FlowFlie Expiration

通过FlowFile到期可以自动从流中删除无法及时处理的数据。比如说，如果给定连接上的文件到期时间设置为“1小时”，并且已经在NiFi实例中一小时的文件到达该连接，则该文件将过期。默认值为0 sec表示数据永不过期。当设置了“0秒”以外的文件到期时，连接标签上会出现一个小时钟图标，因此当查看画布上的流时，DFM可以一目了然地看到它。



##### · Back Pressure

NiFi为背压提供两种配置元素。这允许系统避免数据溢出。

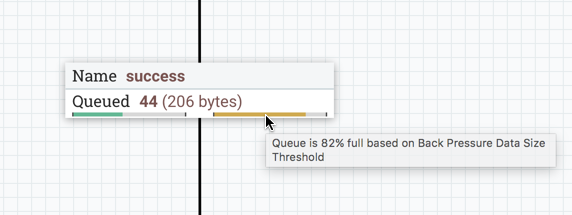
Back pressure object threshold（背压对象阈值）：在应用背压之前可以在队列中的FlowFiles的数量。

Back pressure data size threshold（背压数据大小阈值）：指定了在应用反压之前应排队的最大数据量（大小）。

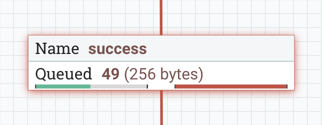
启用背压时，连接标签上会出现小进度条，因此在查看画布上的流时，DFM可以一目了然地看到它。进度条根据队列百分比更改颜色：绿色（0-60％），黄色（61-85％）和红色（86-100％）。



将鼠标悬停在条形图上会显示确切的百分比。



队列完全填满后，Connection将以红色突出显示。



· 优先级

选项卡的右侧提供了对队列中数据进行优先级排序的功能，以便首先处理更高优先级的数据。优先级可以从顶部（'可用的优先级排序器'）拖动到底部（'选择优先级排序器'）。

可以选择多个优先级排序器。位于“所选优先级”列表顶部的优先级排序是最高优先级。如果两个FlowFiles根据此优先级排序器具有相同的值，则第二个优先级排序器将确定首先处理哪个FlowFile，依此类推。如果不再需要优先级排序器，则可以将其从“选定的优先级排序器”列表拖动到“可用的优先级排序器”列表。

可以使用以下优先顺序：

FirstInFirstOutPrioritizer：给定两个FlowFiles，首先处理首先到达连接的FlowFiles。

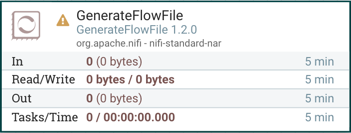
NewestFlowFileFirstPrioritizer：给定两个FlowFiles，将首先处理数据流中最新的FlowFiles。

OldestFlowFileFirstPrioritizer：给定两个FlowFiles，将首先处理数据流中最旧的FlowFiles。

PriorityAttributePrioritizer：给定两个都具有“priority”属性的FlowFile，将首先处理具有最高优先级值的FlowFiles。请注意，应该使用UpdateAttribute处理器将“priority”属性添加到FlowFiles，然后才能到达具有此优先级设置的连接。“优先级”属性的值可以是字母数字，其中“a”是比“z”更高的优先级，“1”是比“9”更高的优先级。

### （6）处理器验证

在尝试启动处理器之前，确保处理器的配置有效非常重要。状态指示器显示在处理器的左上角。如果处理器无效，指示器将显示黄色警告指示器，并带有感叹号，表示存在问题：



在这种情况下，使用鼠标悬停在指示器图标上将提供工具提示，显示处理器的所有验证错误。一旦解决了所有验证错误，状态指示器将变为Stop图标，表示处理器有效并准备启动但当前未运行：



（7）启动processor

为了启动组件，必须满足以下条件：

· 组件的配置必须有效

· 所有为组件定义的关系必须连接到另一个组件或自动终止

· 组件必须停止

· 该组件必须没有活动任务

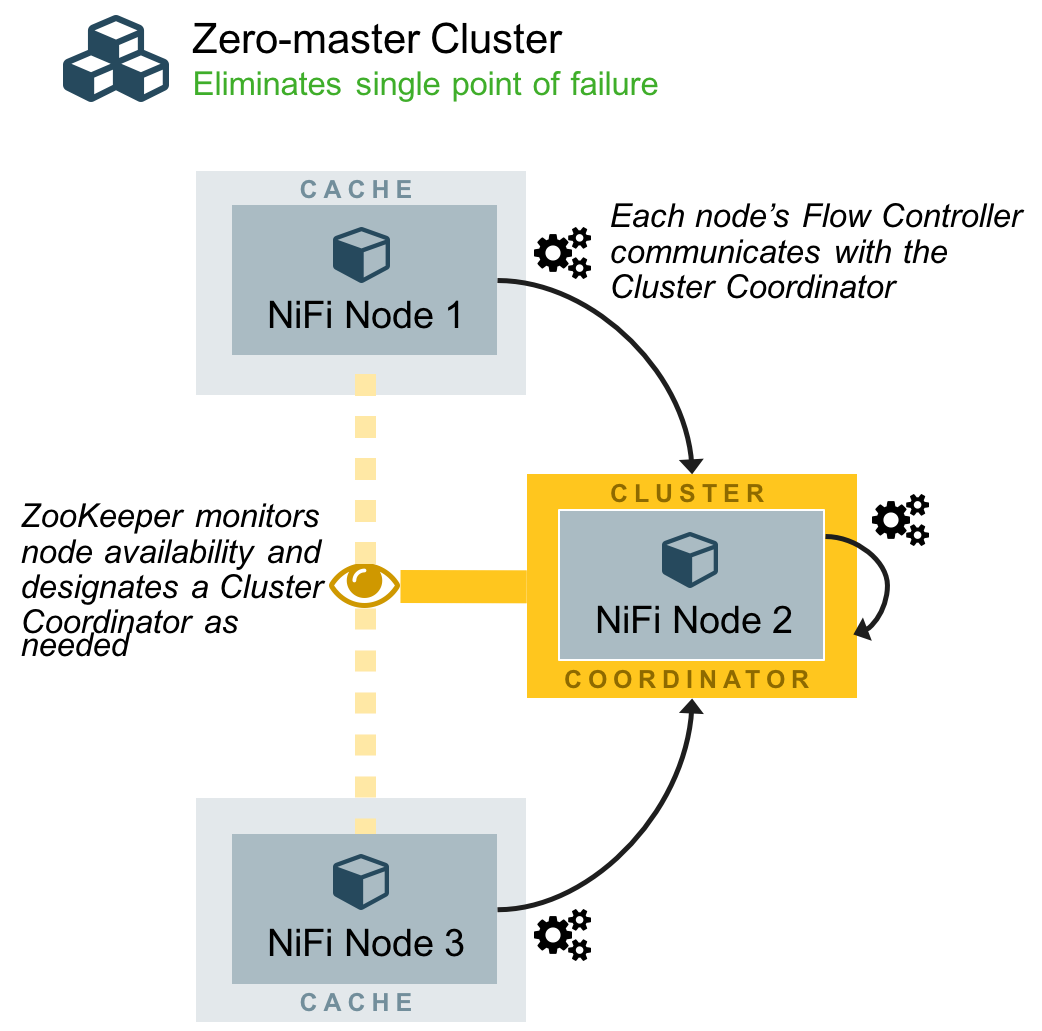
可以通过右键单击一个组件并从上下文菜单中选择Start来启动组件。

如果启动进程组，则该进程组中的所有组件（包括子进程组）都将启动，但那些无效或禁用的组件除外。

一旦启动，处理器的状态指示器将变为播放符号。

## 集群环境搭建

从NiFi 1.0版本开始，NiFi采用Zero-Master聚类范例。NiFi集群中的每个节点都对数据执行相同的任务，但每个节点都运行在不同的数据集上。Apache ZooKeeper选择其中一个节点作为集群协调器，故障转移由ZooKeeper自动处理。所有群集节点都会向群集协调器报告心跳和状态信息。群集协调器负责断开和连接节点。作为DataFlow管理器，您可以通过群集中任何节点的UI与NiFi群集进行交互。您所做的任何更改都会复制到群集中的所有节点，从而允许多个入口点进入群集。



2.1核心模块：   
NiFi Cluster Coordinator（集群协调器）:集群中节点，负责控制任务和管理节点有负载均衡的功能。   
节点:负责实际的数据处理   
主节点:有zookeeper自动选择，此节点上运行隔离处理器   
Isolated Processors（隔离处理器）:不希望在每个节点上运行的任务。独立运行。   
Heartbeats（心跳）:传达节点的运行状态。与集群协调器通信   
特点：采用零主集群范例。每个节点对数据执行相同的任务，但每个节点对不同的数据集进行操作

2.2搭建集群

以一台电脑，两台虚拟机（最小的Centos 7）为例，在三个实例上部署二进制文件并解压缩。现在每个节点上都有一个NiFi目录。

首先要在配置文件“./conf/zookeep.properties”中配置ZK（ZooKeeper）实例的列表:

server.1=node-1:2888:3888

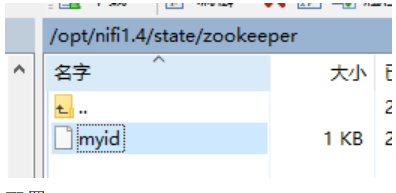
server.2=node-2:2888:3888

server.3=node-3:2888:3888

2.3配置myid

如果多个NiFi节点正在运行嵌入式ZK，则告诉服务器哪一个是重要的。

在nifi目录下创建文件夹/state/zookeeper/并创建文件myid，文件内容与第二步中的server.id一致。



2.4配置state-management.xml:

<property name="Connect String">

node-1:2888,node-2:2888,node-3:2888

</property>

2.5配置nifi节点属性

目录：conf/nifi.properties

指定NiFi必须运行嵌入式ZK实例，并具有以下属性：

nifi.state.management.embedded.zooker.start =true

使用内置zookeeper：  
nifi.zookeeper.connect.string=node-1:2181,node-2:2181,node-3:2181

下面需每个节点单独配，根据节点的IP相应配置，保持集群中节点使用的端口一致

nifi.cluster.is.node=true

nifi.cluster.node.address=node-1

nifi.cluster.node.protocol.port=9999

nifi.remote.input.host=node-1

nifi.remote.input.secure=false

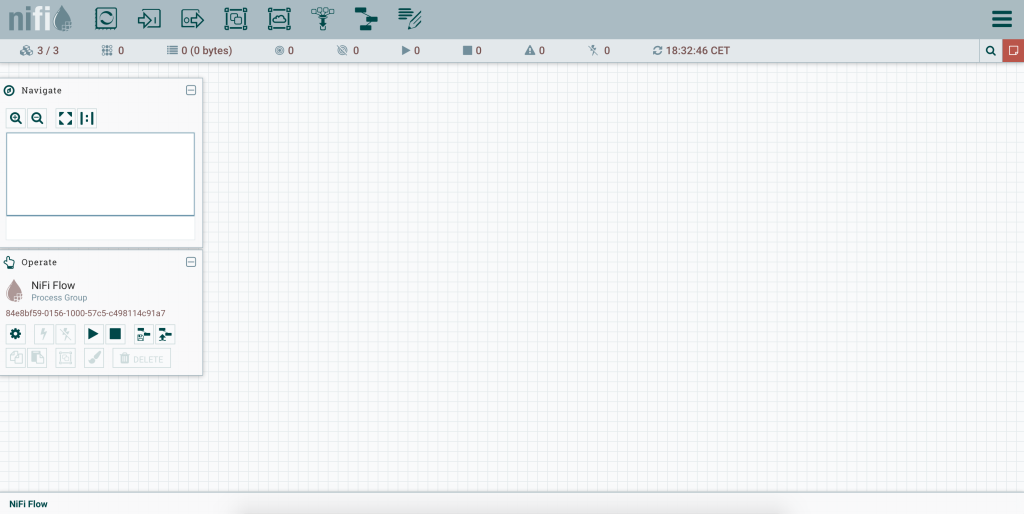
nifi.remote.input.socket.port=9998

nifi.web.http.host =node-1

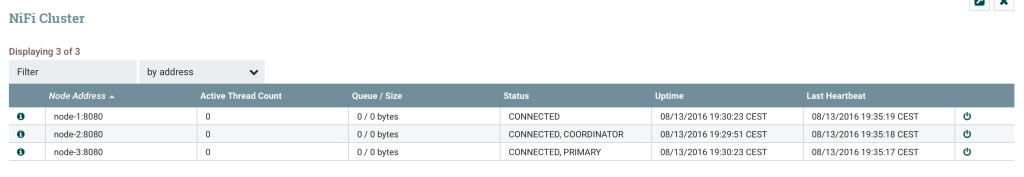
配置完成后就可以此启用节点，集群将选取产生主节点。

2.6测试集群

访问 [http：//node-2：8080 / nifi](http://node-2:8080/nifi)

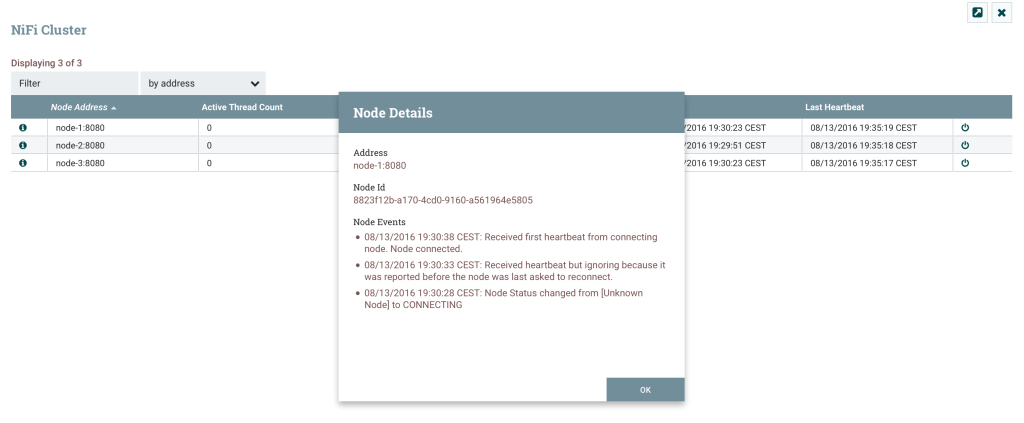


正如在左上角看到的，集群中有3个节点。此外，如果我们进入菜单（右上角的按钮）并选择群集页面，将会出现三个节点的详细信息：



node-2已被选为集群协调器，而node-3则是主节点。这种区别很重要，因为某些处理器必须运行在一个唯一的节点上（为了数据一致性），在这种情况下，我们希望它运行在“主节点上”。

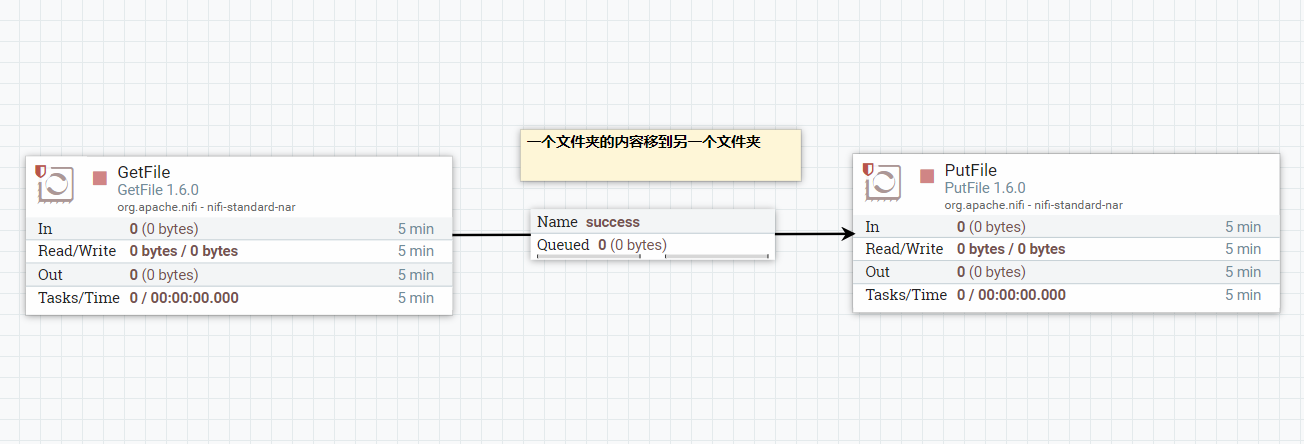
我们可以在特定节点上显示细节（左侧的“信息”图标）：



# 典型技术场景

## GetFile To PutFile

1. 整体流程图

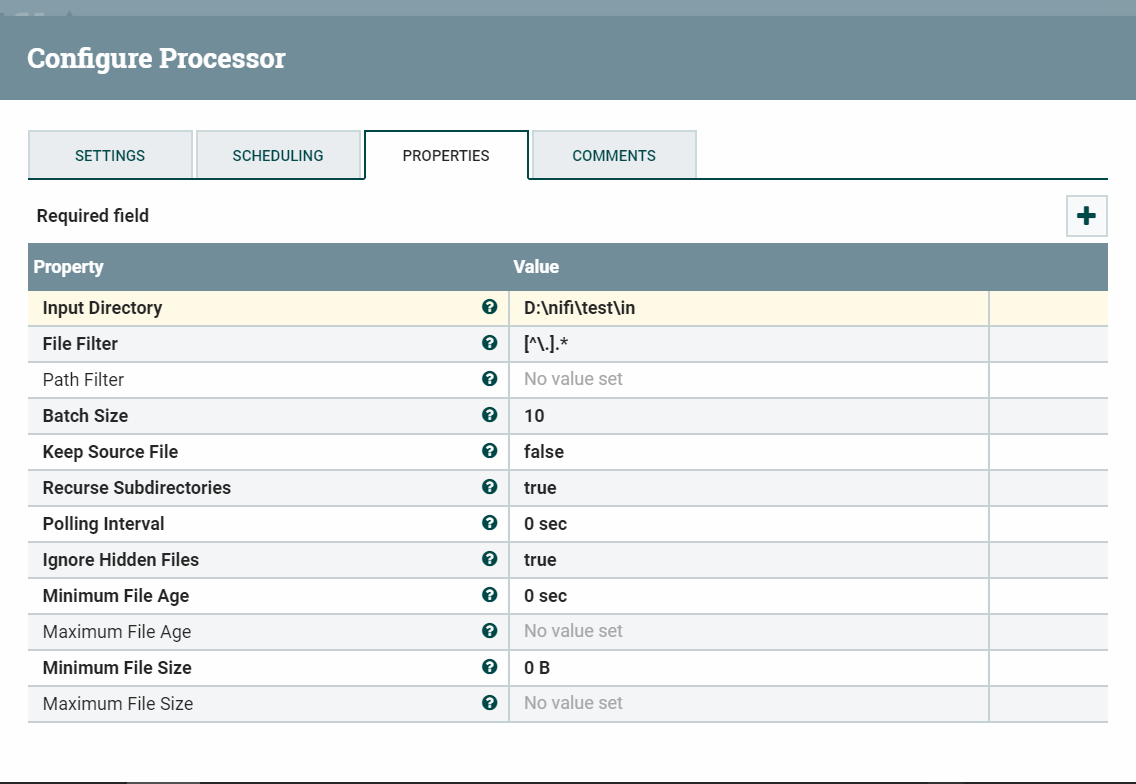


涉及到的处理器以及功能

- GetFile:从指定的路径中读取文件

- PutFile:移动文件到指定位置

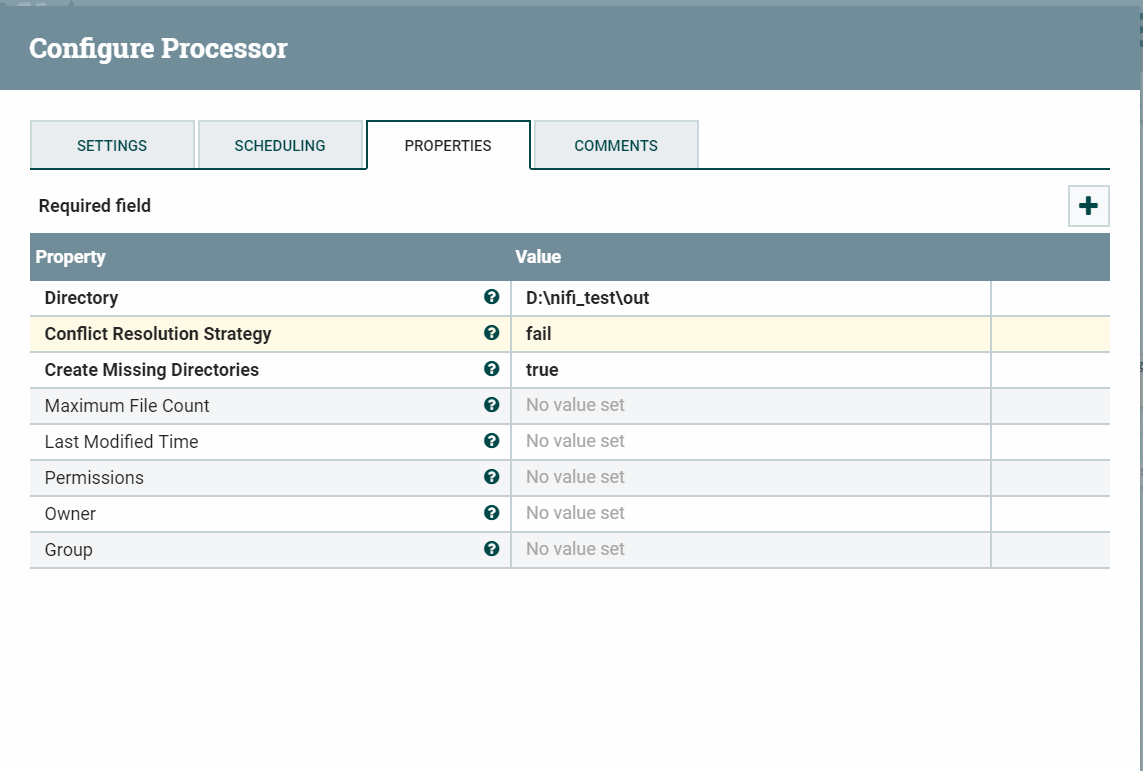
1. 细节说明：
2. GetFile：读取文件



· Input Directory：从中提取文件的输入目录

· File Filter：仅拾取名称与给定正则表达式匹配的文件

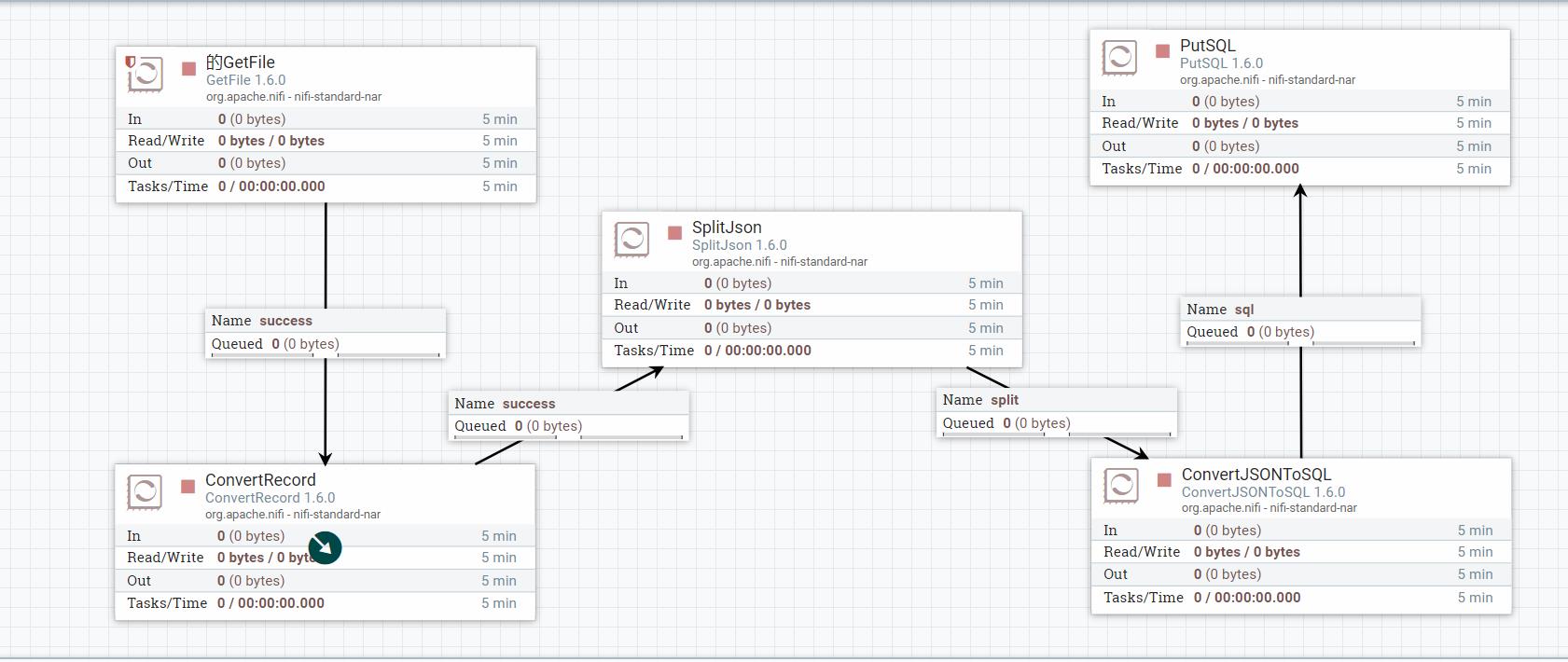
1. PutFile：存放文件



· Directory：文件存放目录

## 从csv到mysql

1. 整体流程图

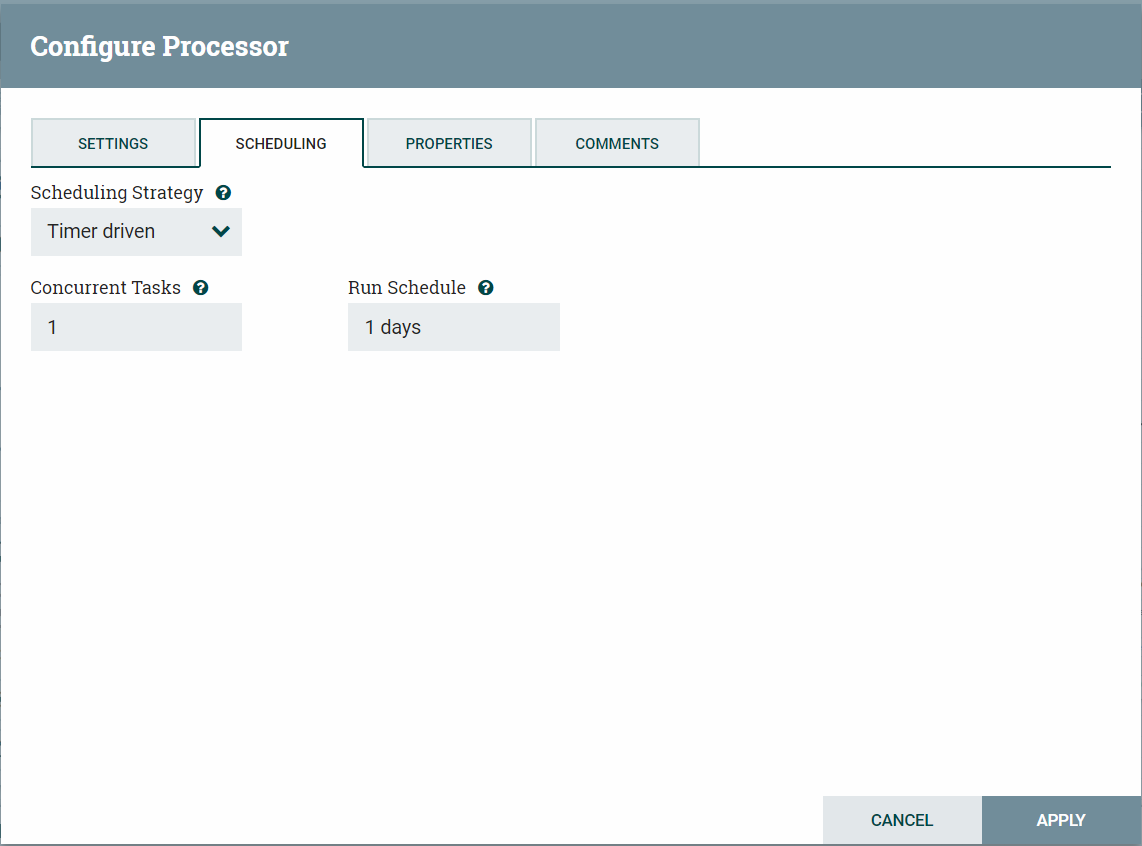


涉及到的处理器以及功能   
· GetFile:从指定的路径中读取文件    
· ConvertRecord:通过指定Reader和Writer的类型，完成文件格式转换

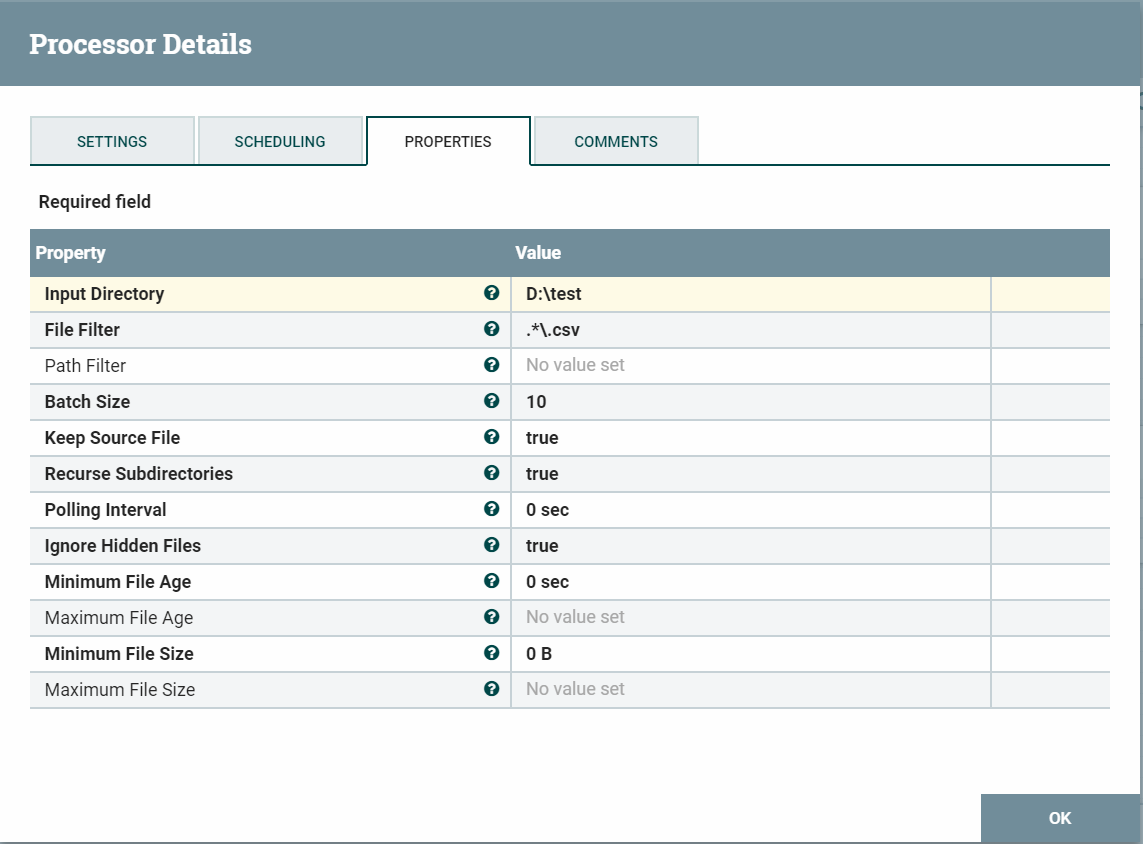
· Splitjson：将JSON文件拆分为多个独立的FlowFiles

· ConvertJSONToSQL：将json中的元素转化为sql中的insert语句  
· PutSQL：执行SQL UPDATE或INSERT命令

1. 细节说明：
2. GetFile：读取文件



设置循环时间为1 days ，防止数据重复插入



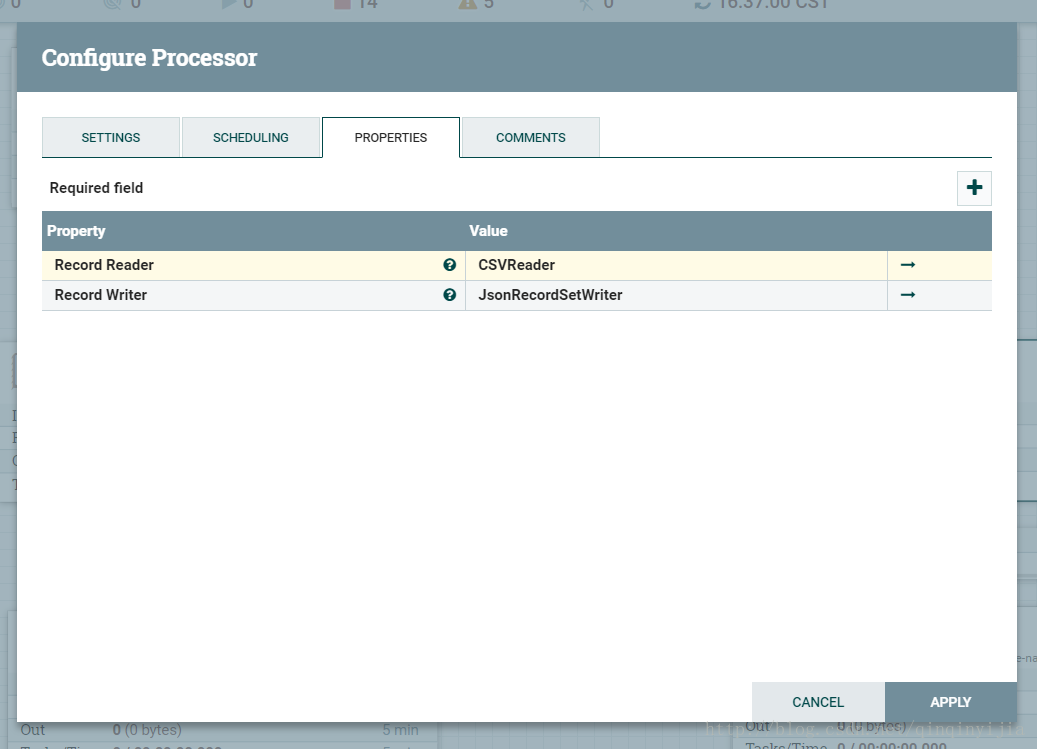
· Input Directory：输入目录，从中提取文件的输入目录

· File Filter：文件过滤器，只有名称与给定正则表达式匹配的文件才会被拾取

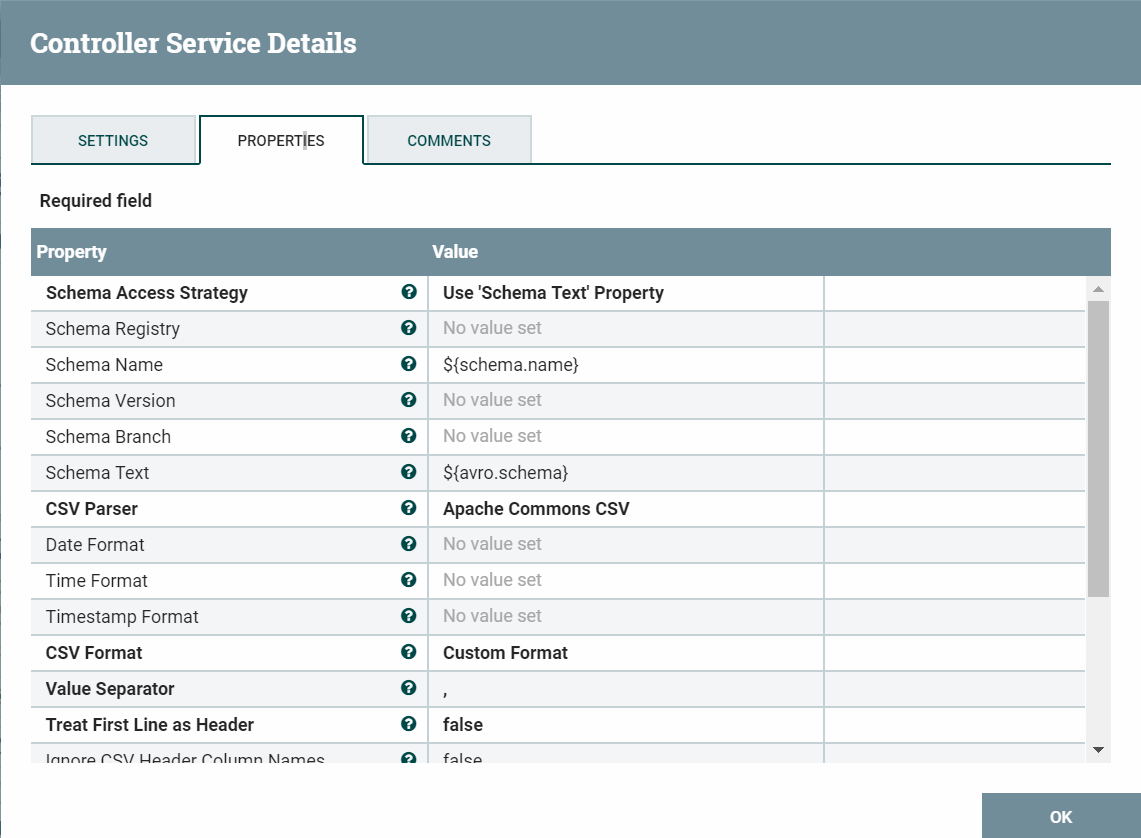
· Keep Source File：默认情况下，会将源文件删除

（2）ConvertRecord:转换文件格式

首先添加一个Record Reader和Record Writer，对于Record Reader，我们选择的是CSVReader，因为我们读取的文件是CSV格式，这个需要根据读取文件的格式选择。对于RecordWriter，我们选择的是JsonRecordSetWriter。

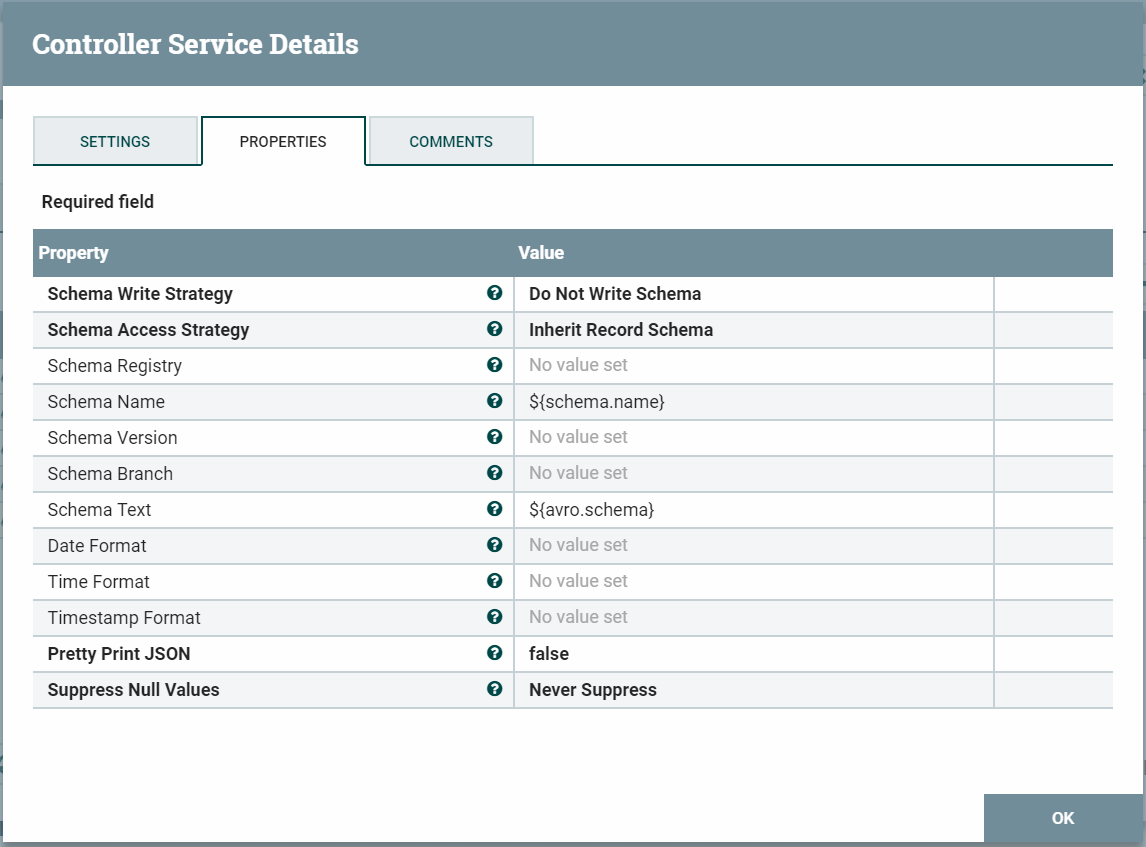


· Record Reader：CSVReader（根据所要读入数据的格式进行设定）,点击右侧的箭头， 对CSVReader的属性进行设定

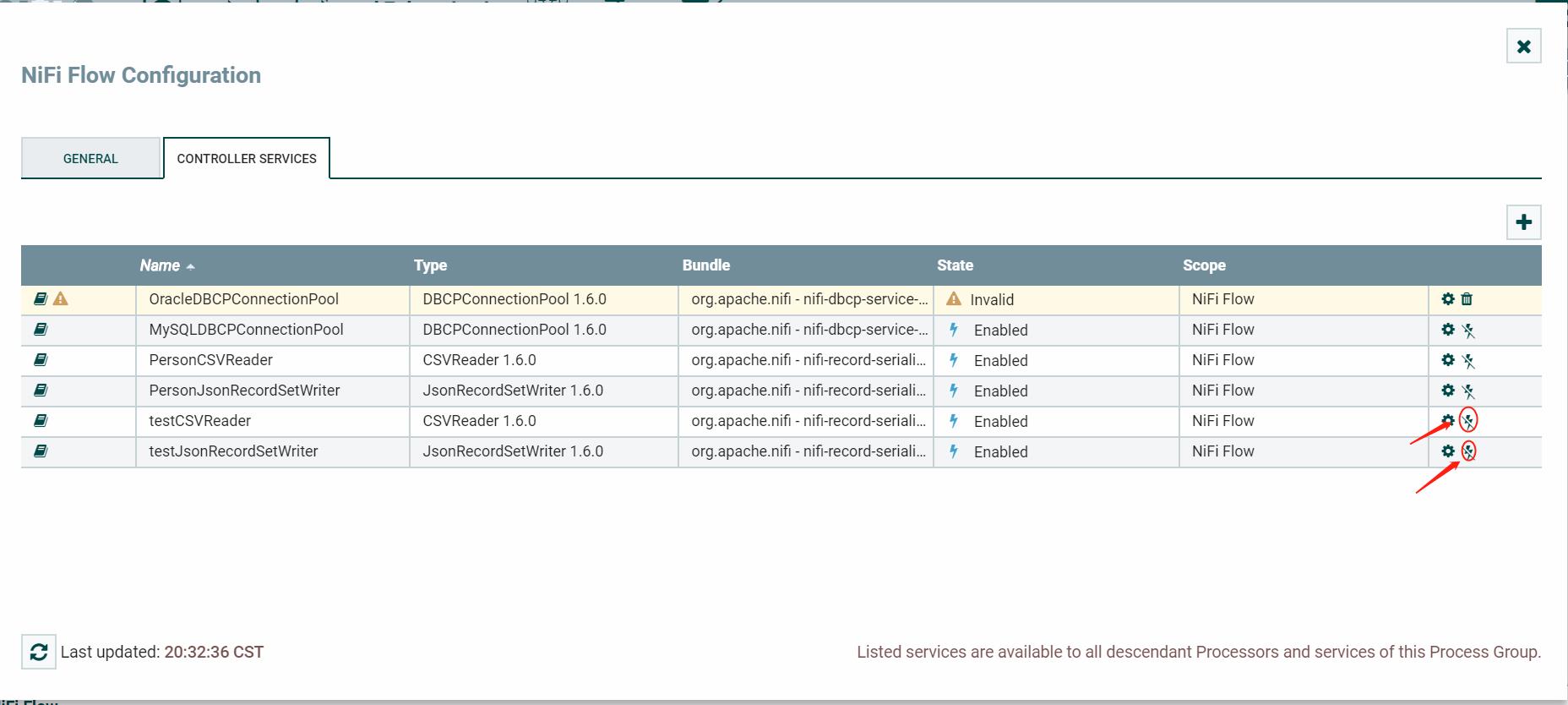


· Schema Access Strategy：这里我们选择通过Schema Test来找到对应的schema   
· Schema Registry：需要选择Scheme Registry的类型，这里选择的是AvroSchemaRegis try，右侧又出现一个小箭头，需要对AvroSchemaRegistry进行设置。

· Record Write进行类似的设置即可。



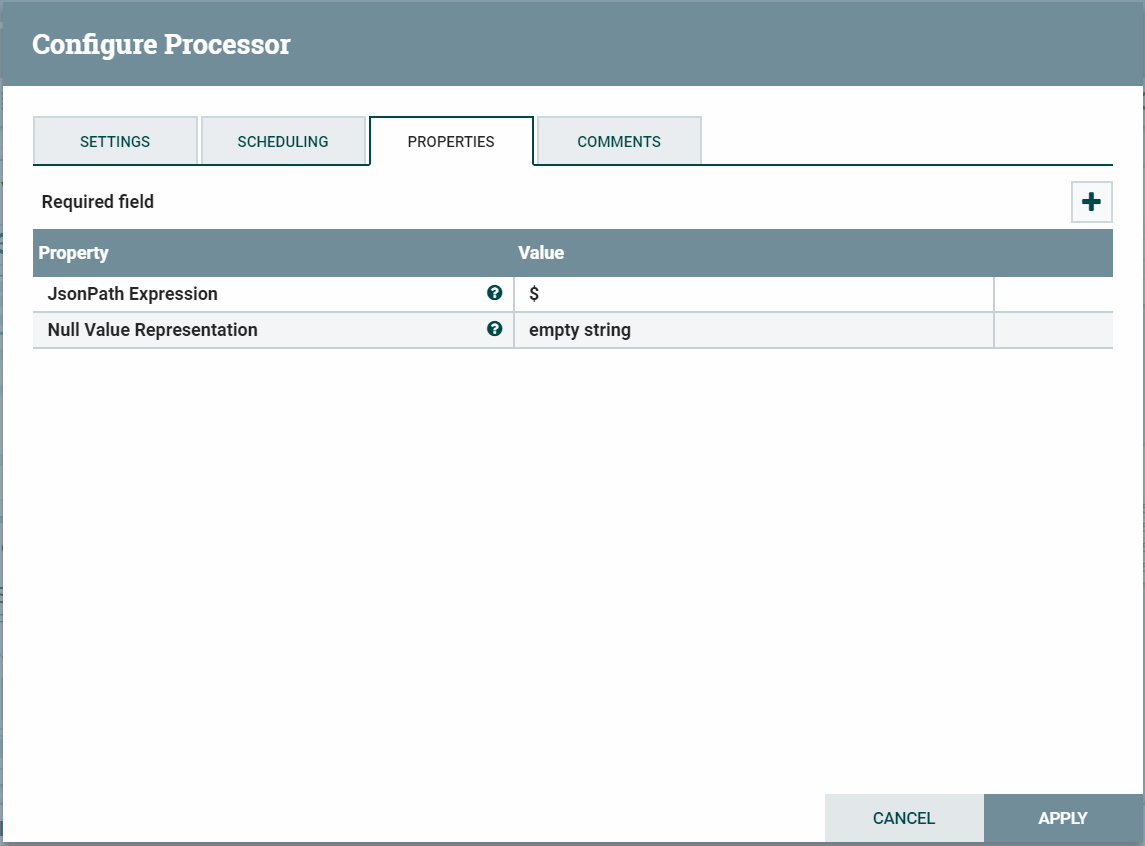
· 启动控制器



（3）SplitJson ：将JSON文件拆分为由JsonPath表达式指定的数组元素的多个独立的

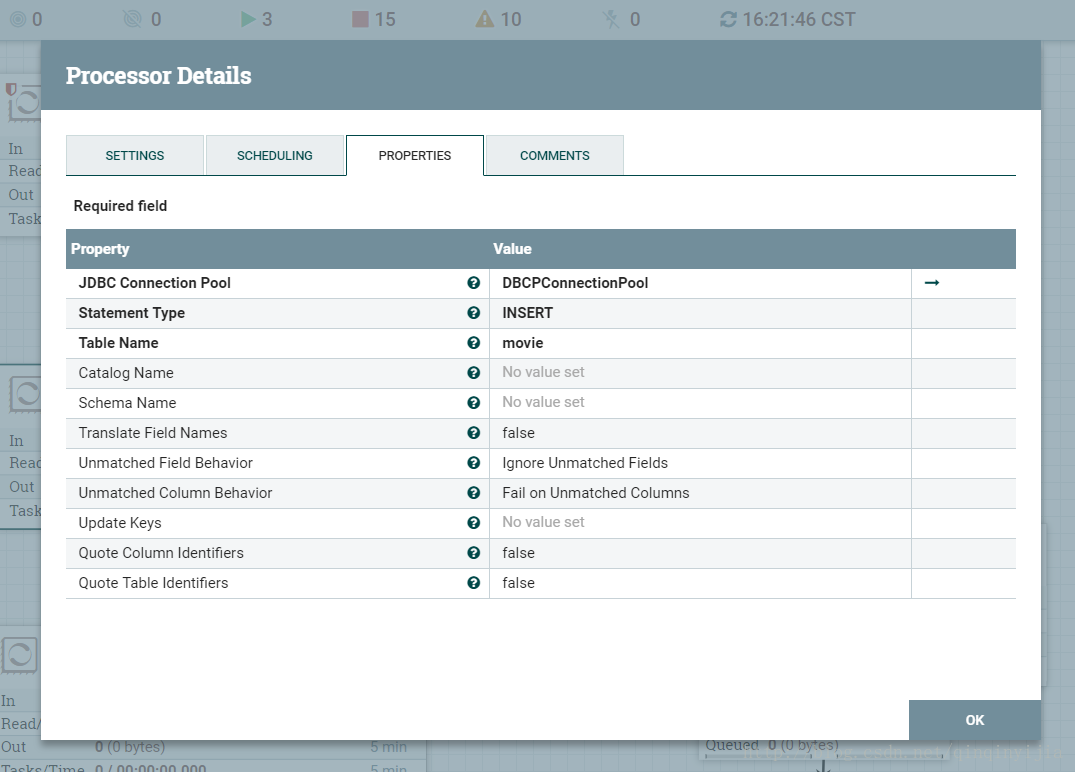
FlowFiles

然后从ConvertAvroToJson拖一条线到SplitJson，关系为success。

  
· JsonPathExpression：一个JsonPath表达式，用于指示要拆分为JSON 的数组元素

（4）ConvertJSONToSQL处理器：将JSON格式的FlowFile转换为SQL语句

【注意】该处理器有一个特性，只能处理flat json，所谓flat是由一个JSON元素组成，每个字段映射到一个简单类型



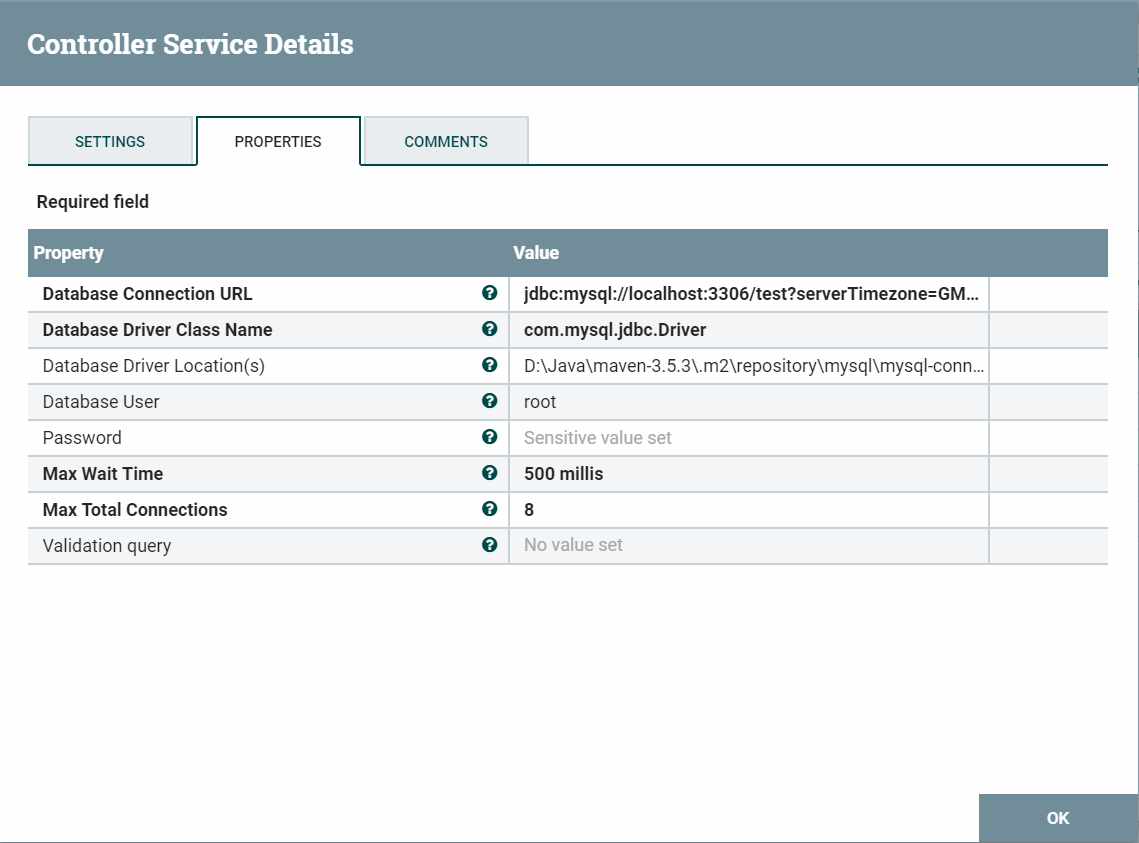
· JDBC Connection Pool：根据要连接的数据库类型选择，我要连接的是mysql数据库，因此选择DBCPConnectionPool

· Statement Type：设置要执行的操作，INSERT和UPDATE等，这里要执行的是插入操作

· Table Name：语句应更新的表的名称

· Translate Field Name: 如果json中元素的属性名称与数据表中的列名称一致，则选择false，否则选择true

· JDBC Connection Pool的属性后面有一个小箭头，点击箭头对此项进行设置：



实际上这个java连接数据的设置是一致的，   
· Database Driver Class Name: 根据要连接的数据库类型选择

jdbc:mysql://localhost:3306/test

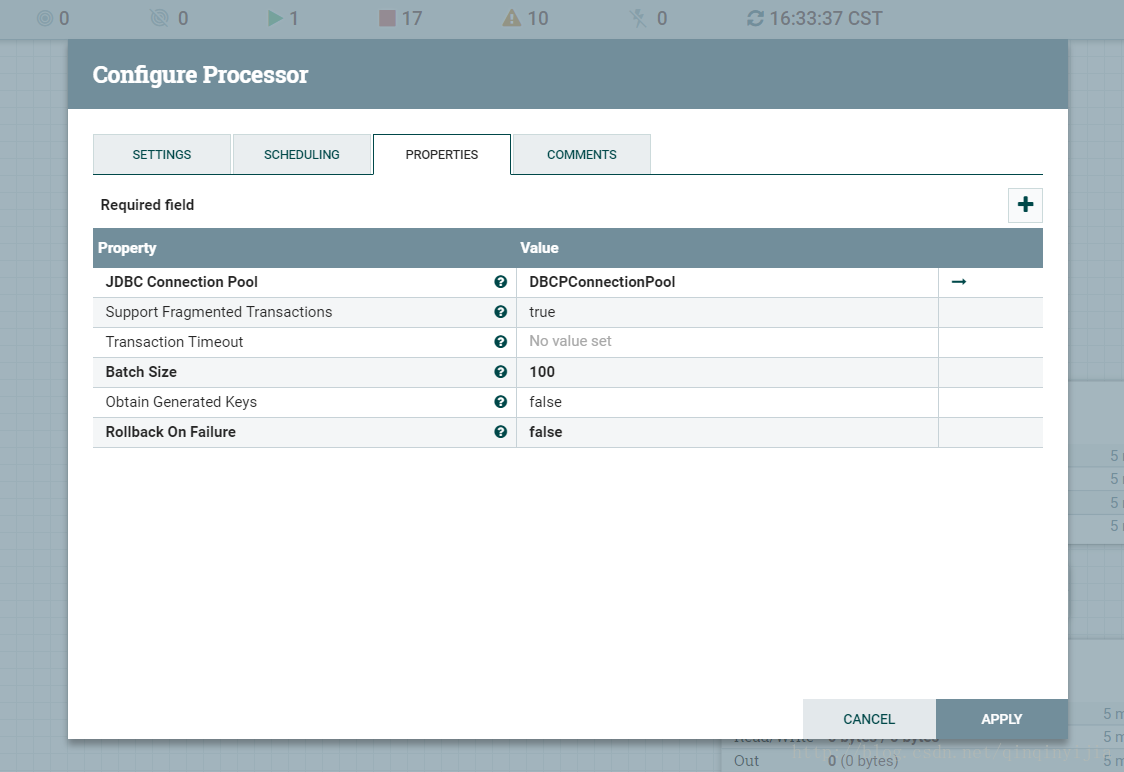
【注意】数据库和系统时区差异问题，在jdbc连接的url后面加上serverTimezone=GMT即可解决问题，如果需要使用gmt+8时区，需要写成GMT%2B8，否则会被解析为空。再一个解决办法就是使用低版本的MySQL jdbc驱动，5.1.28不会存在时区的问题。  
· Database Driver location：选择对应数据库连接jar包的完整路径

【注意】Jar包版本要与MySQL版本相匹配

D:\Java\maven-3.5.3\.m2\repository\mysql\mysql-connector-java\5.1.6

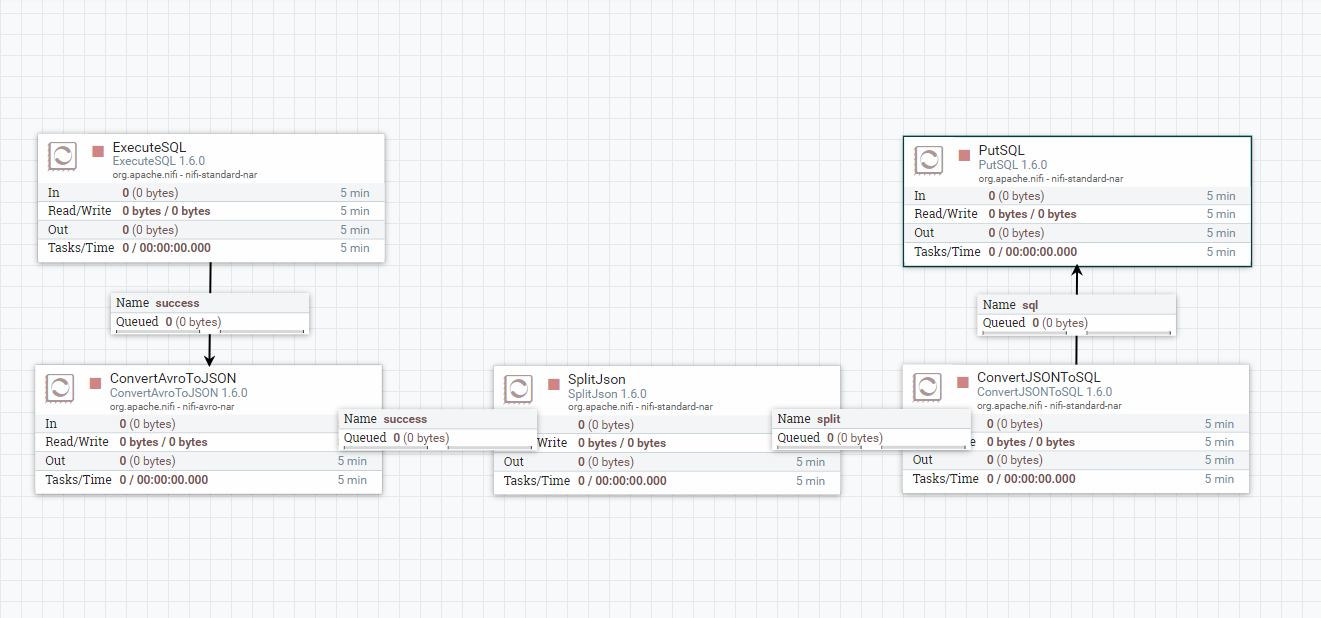
· Database User：登录数据库的用户名   
· Password：用户名对应的密码

（5）PutSQL处理器：这里只设定了 JDBC Connection Pool



## MySQL To Oracle

1. 整体流程图



涉及到的处理器以及功能   
- ExecuteSQL:执行提供的SQL选择查询，查询结果将转换为Avro格式

- ConvertAvroToJson：将avro格式的数据转化成json格式

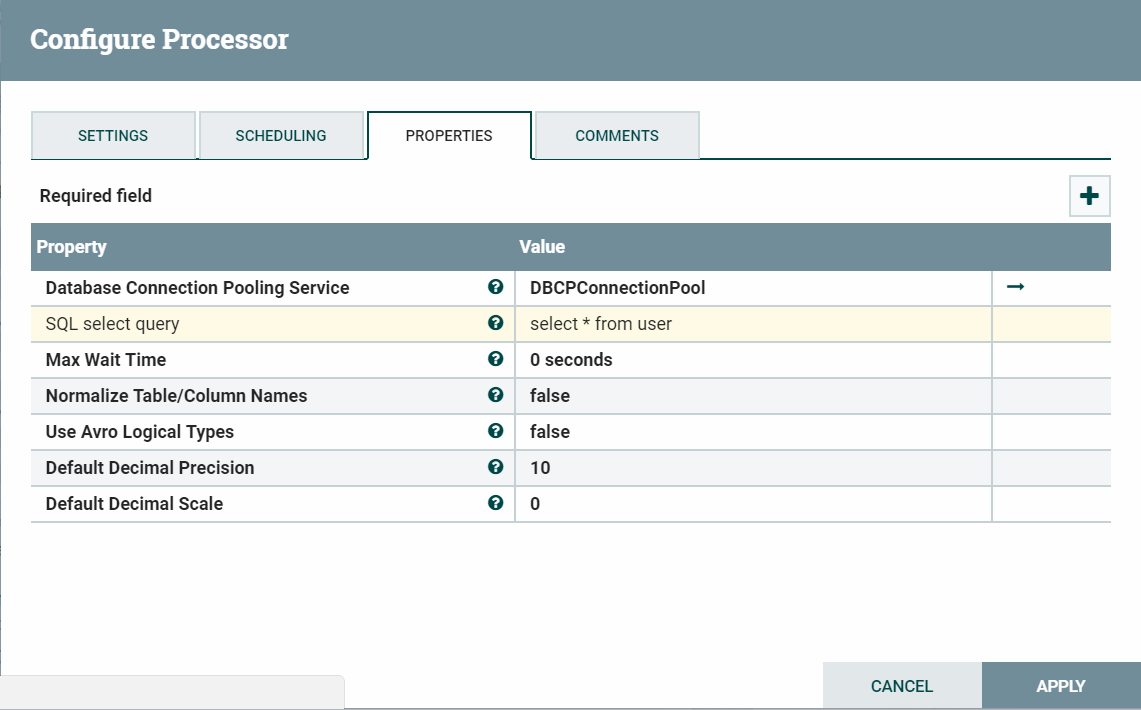
- Splitjson：将JSON文件拆分为多个独立的FlowFiles

- ConvertJSONToSQL：将json中的元素转化为sql中的insert语句

- PutSQL：执行SQL UPDATE或INSERT命令

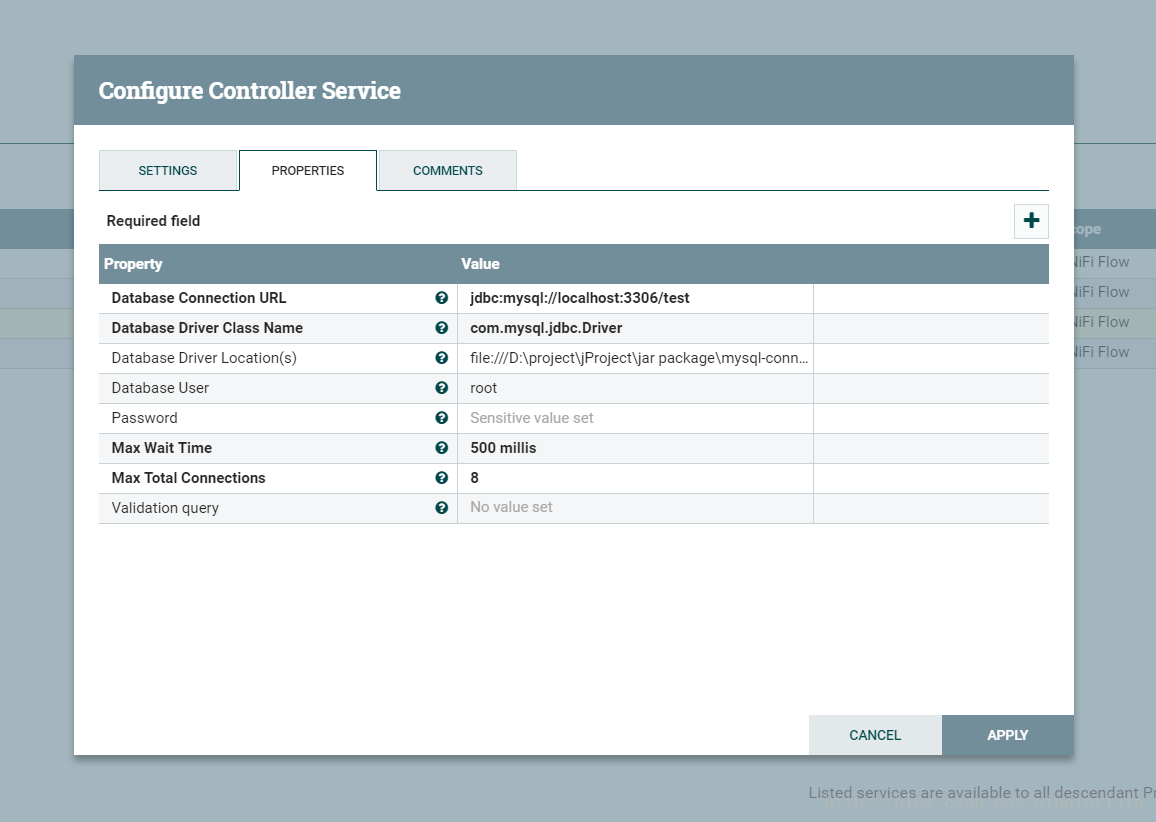
2. 细节说明

1. ExecuteSQL:



· 设置SQL select query为 select \* from user

· service-->DBCPConnectionPool，然后再点击右侧的箭头，配置下一个选项



·Database Connection URL：jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf8&useSSL=true

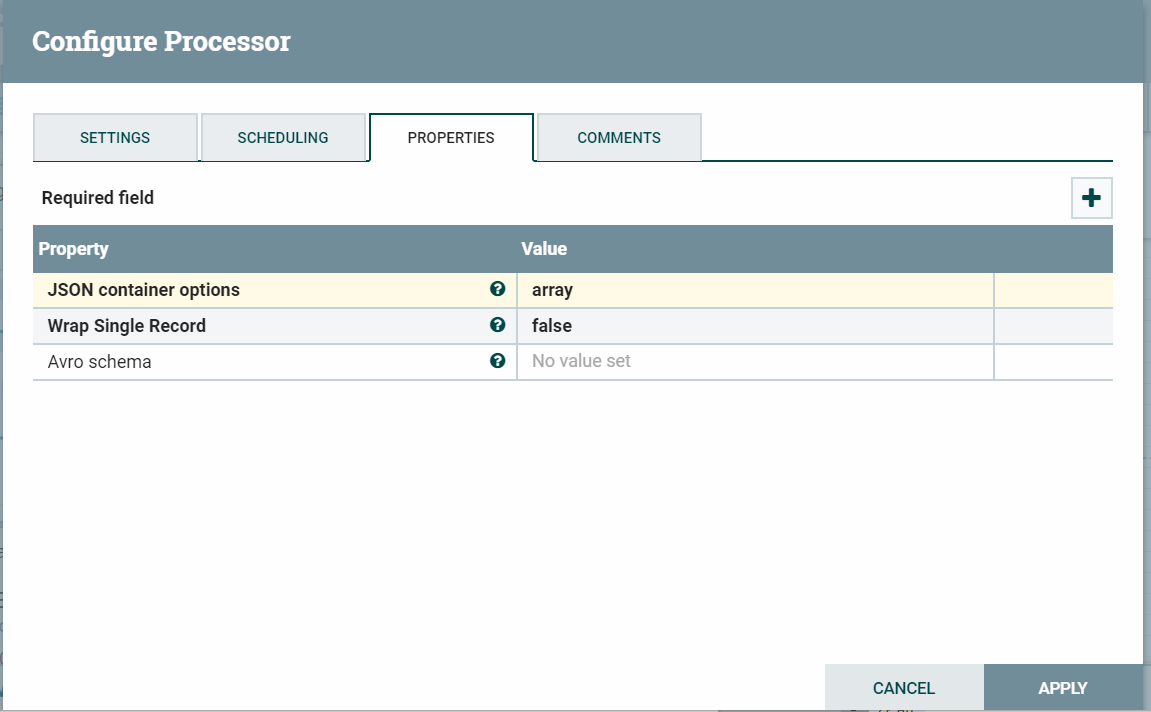
· Database Driver Class Name: com.mysql.jdbc.Driver

· Database Driver location:/home/xxxx/mysql-connector-java-5.1.39.jar

· Database User：登录数据库的用户名

· Password：用户名对应的密码

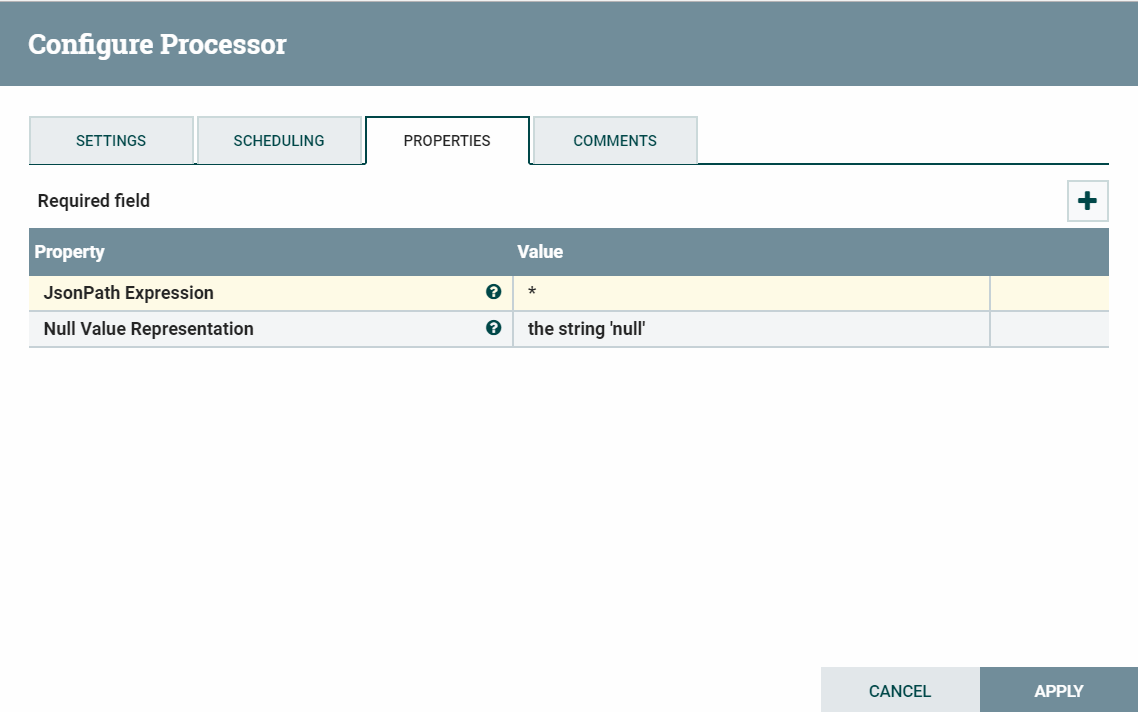
（2）ConvertAvroToJson：将二进制Avro记录转换为JSON对象  
 然后，从ExecuteSQL拖一条线到ConvertAvroToJson，关系为success。



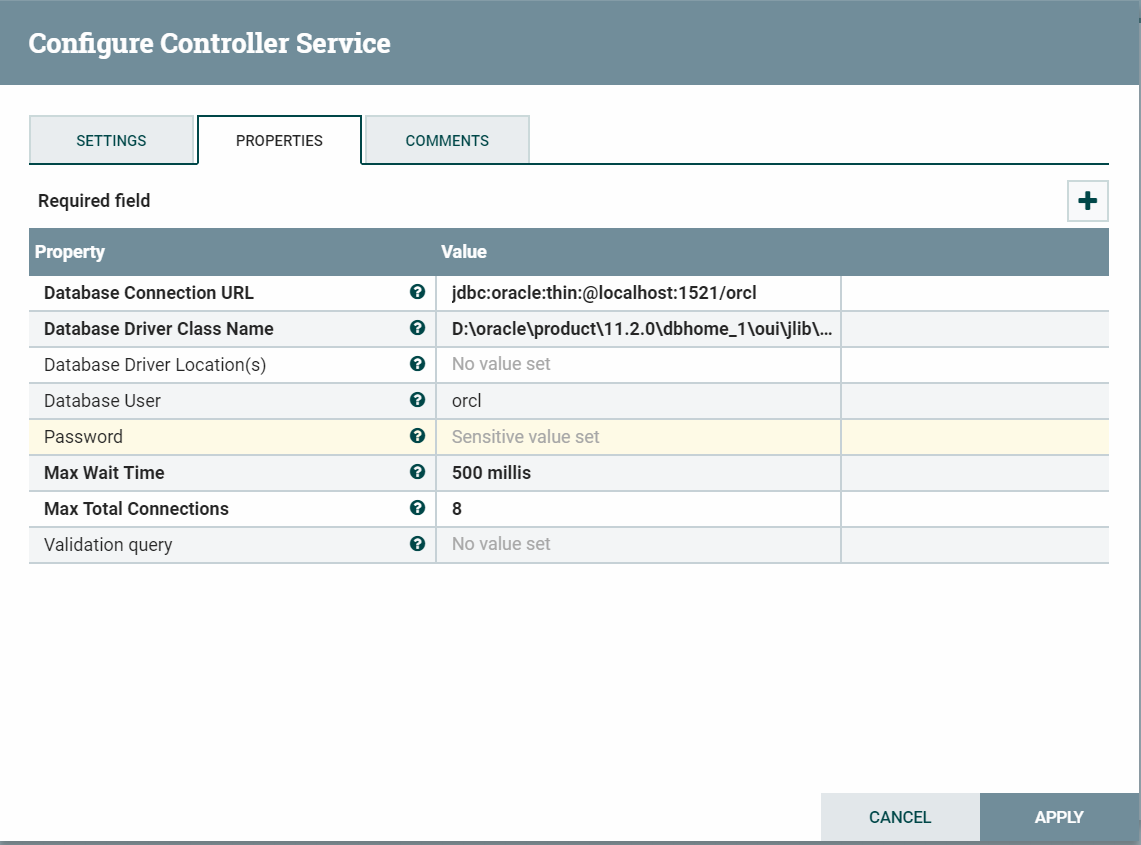
1. SplitJson ：将JSON文件拆分为由JsonPath表达式指定的数组元素的多个独立的

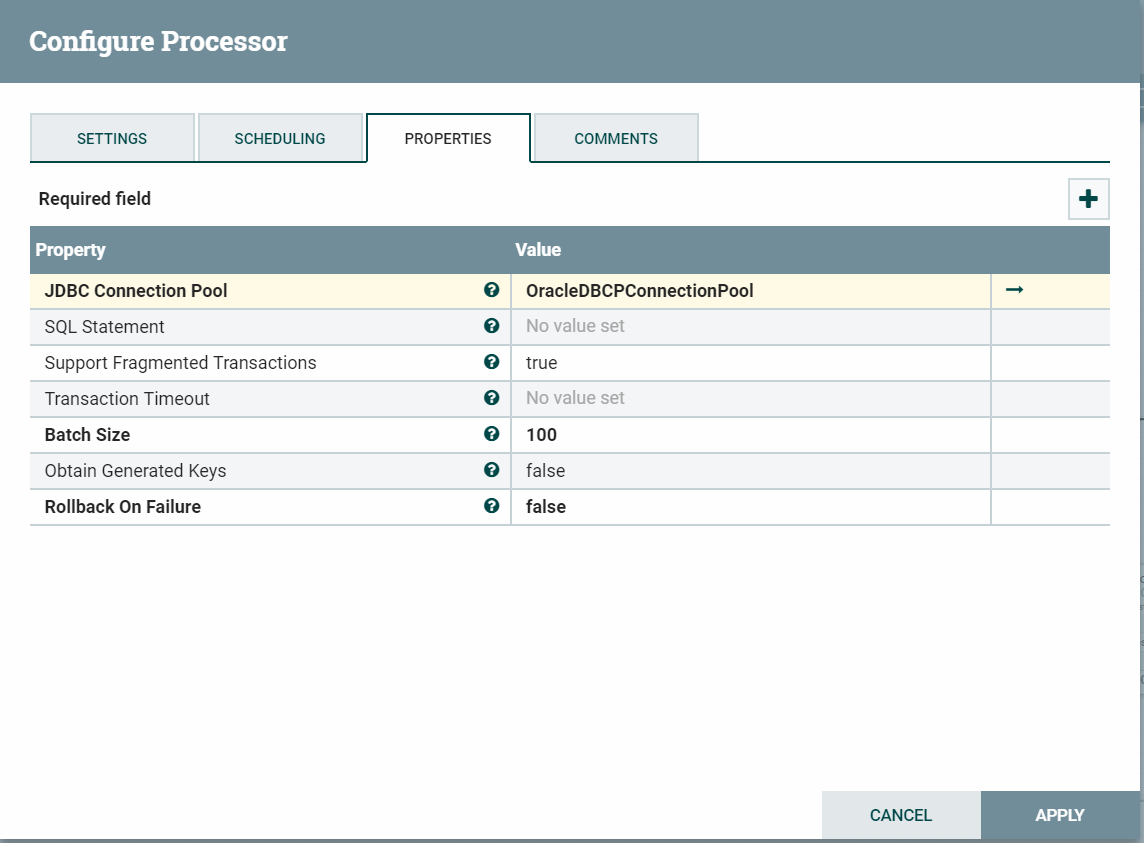
FlowFiles

然后从ConvertAvroToJson拖一条线到SplitJson，关系为success。

  
· JsonPathExpression：一个JsonPath表达式，用于指示要拆分为JSON 的数组元素

1. 添加一个ConvertJSONToSQL到界面，然后配置

 （5）PutSQL



## 执行javaScript脚本

ExecuteScript是一个多功能处理器，允许用户使用编程语言编写自定义逻辑，每次触发

ExecuteScript处理器时都会执行该编程语言。

以下变量绑定被提供给脚本以允许访问NiFi组件：

· session（会话）：这是对分配给处理器的ProcessSession的引用。会话允许您对流文件（如create（），putAttribute（）和transfer（）以及read（）和write（）（）进行操作。

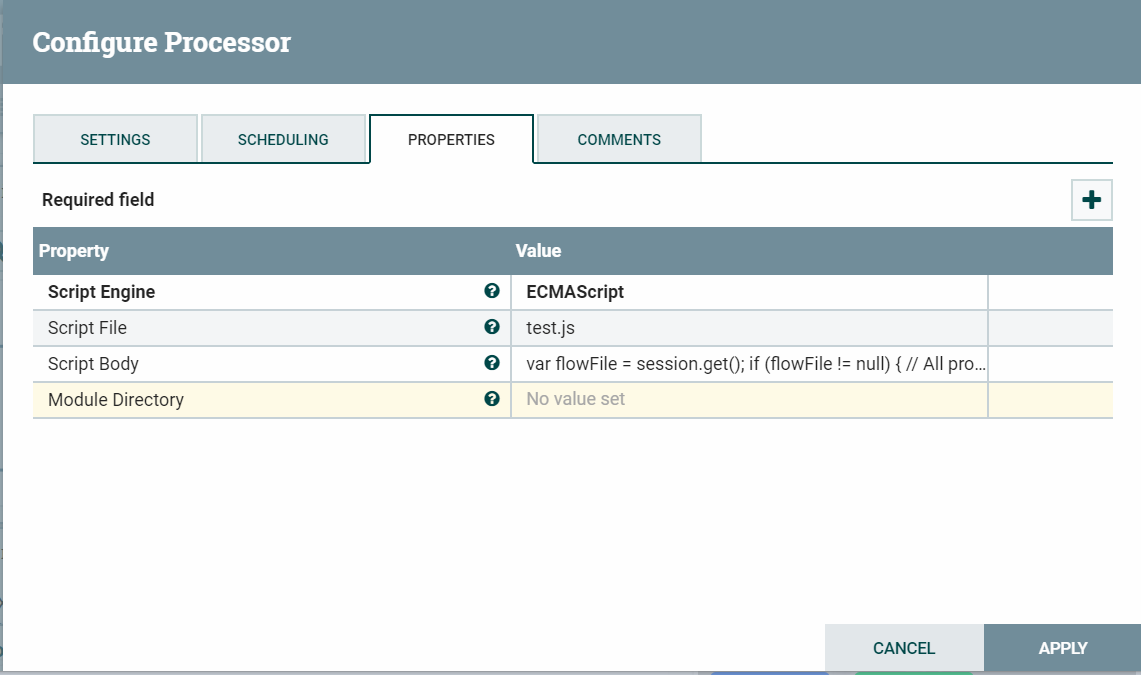
· context（上下文）：这是对处理器的ProcessContext的引用。它可以用来检索处理器属性，关系，Controller服务和StateManager。

· log：这是对处理器ComponentLog的引用。用它来记录消息给NiFi，比如log.info（'Hello world！'）

· REL\_SUCCESS：这是对处理器定义的“成功”关系的引用。它也可以通过引用父类（ExecuteScript）的静态成员来继承，但是一些引擎（如Lua）不允许引用静态成员，所以这是一个方便的变量。这也节省了必须使用关系的完全合格的名称。

· REL\_FAILURE：这是对处理器定义的“失败”关系的引用。和REL\_SUCCESS一样，它也可以通过引用父类（ExecuteScript）的静态成员来继承，但是一些引擎（如Lua）不允许引用静态成员，所以这是一个方便的变量。这也节省了必须使用关系的完全合格的名称。

· Dynamic Properties ： 在ExecuteScript中定义的任何动态属性都将作为设置为与动态属性对应的PropertyValue对象的变量传递给脚本引擎。这允许您获取属性的String值，还可以针对NiFi表达式语言评估该属性，将该值作为适当的数据类型（例如布尔值）等进行转换。由于动态属性名称会成为脚本的变量名称，您必须知道所选脚本引擎的变量命名属性。例如，Groovy不允许在变量名称中使用句点（。），因此如果“my.property”是一个动态属性名称，则会发生错误。



· Script Engine：脚本引擎选择ECMAScript中

· Script File：脚本文件

· Script Body：脚本内容

（1）从会话中获取传入的流文件

方法：使用会话对象中的get（）方法。

var flowFile = session.get();

if (flowFile != null) {

// All processing code goes here

}

#### 从会话中获取多个传入的流文件

方法：使用会话对象中的get（maxResults）方法。

flowFileList = session.get(100) ;

if(!flowFileList.isEmpty()) {

for each (var flowFile in flowFileList) {

// Process each FlowFile here

}

}

1. 从父级FlowFile创建一个新的FlowFile

方法：使用会话对象的create（parentFlowFile）方法。

var flowFile = session.get();

if (flowFile != null) {

var newFlowFile = session.create(flowFile);

// Additional processing here

}

#### 为流文件添加一个属性

方法：使用会话对象中的putAttribute（flowFile，attributeKey，attributeValue）

方法。

var flowFile = session.get();

if (flowFile != null) {

flowFile = session.putAttribute(flowFile, 'myAttr', 'myValue')

}

#### （5）将多个属性添加到流文件

方法：使用会话对象中的putAllAttributes（flowFile，attributeMap）方法。

var number2 = 2;

var attrMap = {'myAttr1':'1', 'myAttr2': number2.toString()}

var flowFile = session.get()

if (flowFile != null) {

flowFile = session.putAllAttributes(flowFile, attrMap)

}

#### （6）从流文件中获取属性

方法：使用FlowFile对象的getAttribute（attributeKey）方法。

var flowFile = session.get();

if (flowFile != null) {

var myAttr = flowFile.getAttribute('filename')

}

（7）从流文件获取所有属性

方法：使用FlowFile对象的getAttributes（）方法。

var flowFile = session.get() if (flowFile != null) {

var attrs = flowFile.getAttributes();

for each (var attrKey in attrs.keySet()) {

// Do something with attrKey (the key) and/or attrs[attrKey] (the value)

}

}

（8）将流文件转移到关系

方法：使用会话对象的transfer（flowFile，relationship）方法。

var flowFile = session.get();

if (flowFile != null) {

// All processing code goes here

if(errorOccurred) {

session.transfer(flowFile, REL\_FAILURE)

} else {

session.transfer(flowFile, REL\_SUCCESS)

}

}

（9）以指定的日志记录级别向日志发送消息

方法：使用带有warn（），trace（），debug（），info（）或error（）方法的log变量。

var ObjectArrayType = Java.type("java.lang.Object[]");

var objArray = new ObjectArrayType(3);

objArray[0] = 'Hello';

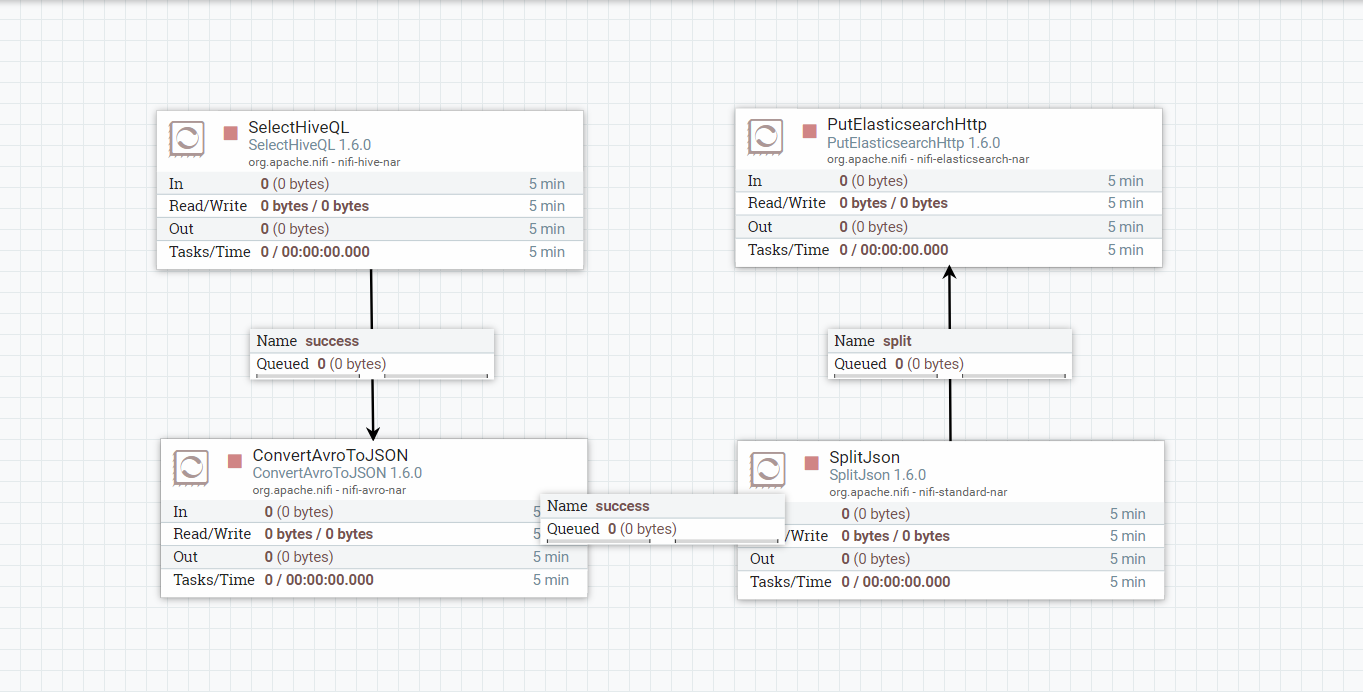
objArray[1] = 1;

objArray[2] = true;

log.info('Found these things: {} {} {}', objArray)

## Hive To Elasticsearch

1. 整体流程图



涉及到的处理器以及功能   
- SelectHiveQL:从Hive库中查取数据

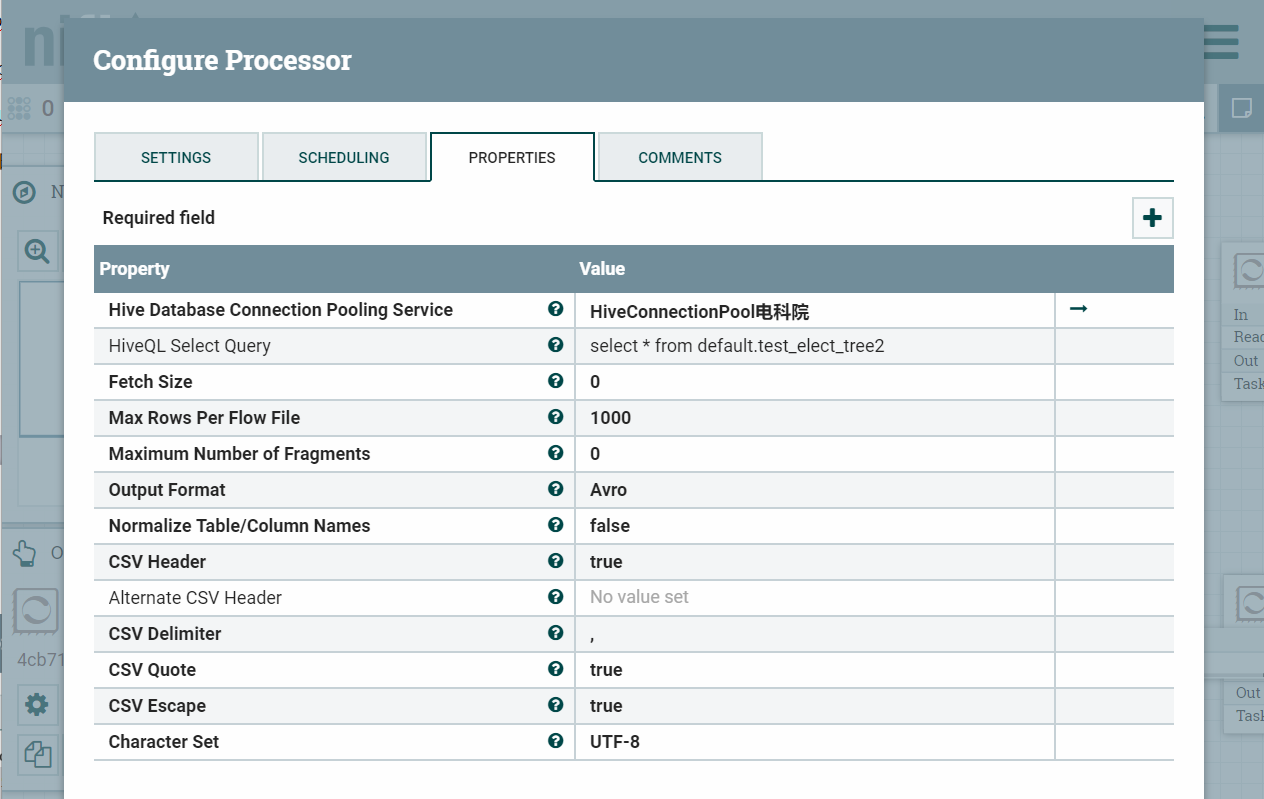
- ConvertAvroToJson：将查出来的数据转换为Json格式

- Splitjson：将Json文件拆分为多个独立的FlowFiles

- PutElasticsearchHttp：将数据插入到ES库中

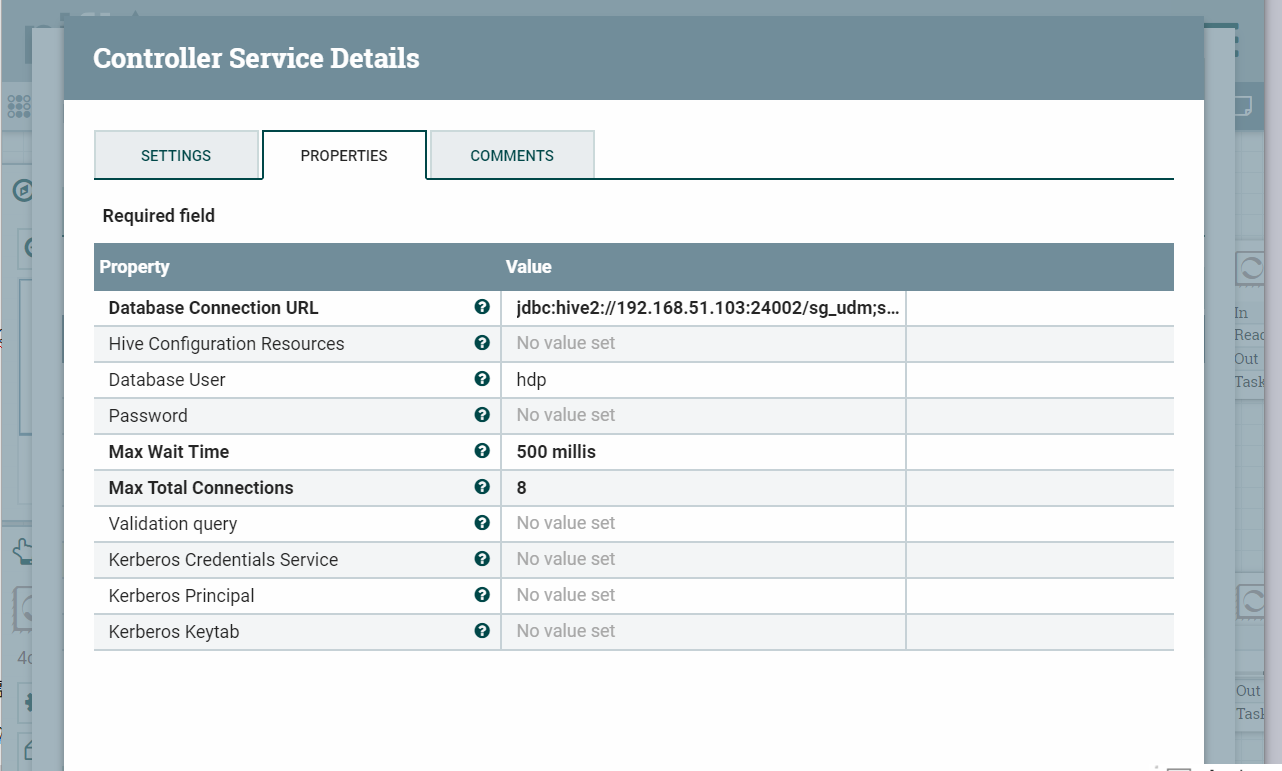
2. 细节说明

（1）SelectHiveQL：读取文件



· HiveQl Select Query ：查询语句

· Hive Database Connection Pooling Service : 点击箭头配置Hive库连接



· Database Connection URL :

jdbc:hive2://192.168.51.103:24002/sg\_udm;serviceDiscoveryMode=zooKeeper;

zooKeeperNamespace=hiveserver2

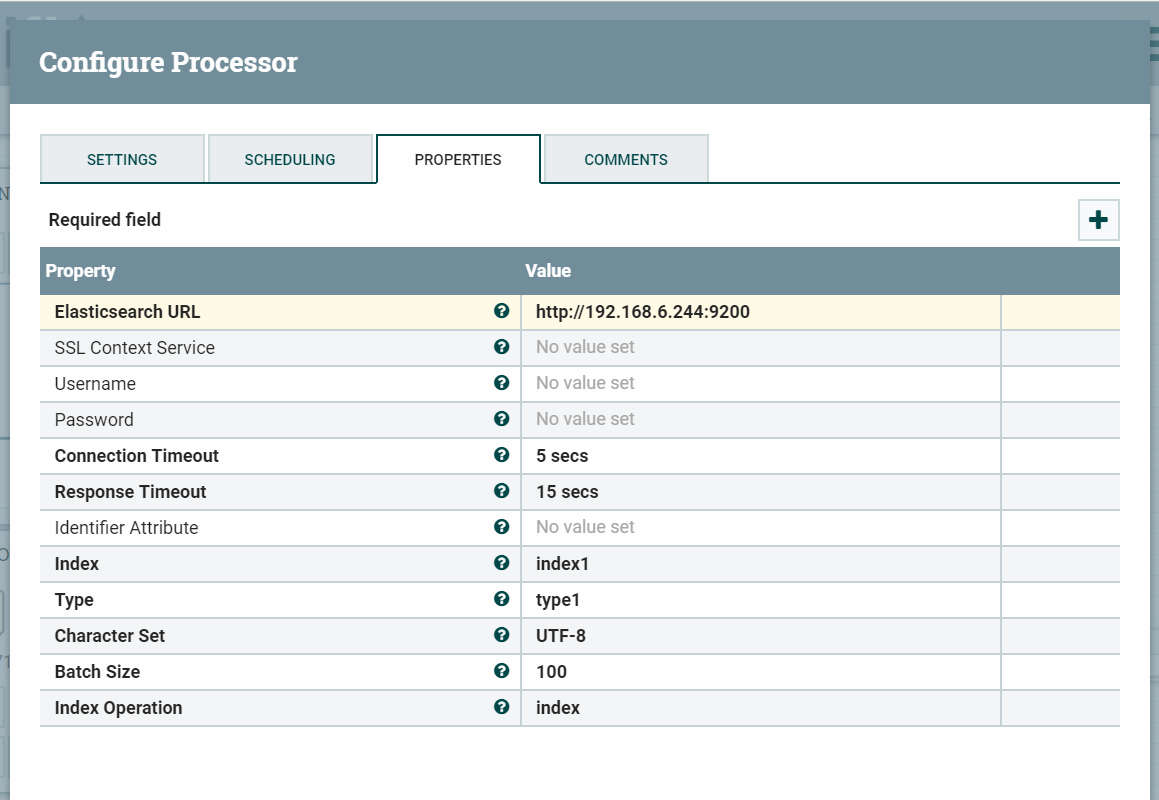
· Database User : 用户名

· Password ：密码

（2）ConvertAvroToJson 默认设置

（3）Splitjson 默认设置

（4）PutElasticsearchHttp



· Elasticsearch URL : 设置ES的ip:端口

注意：此处是http协议

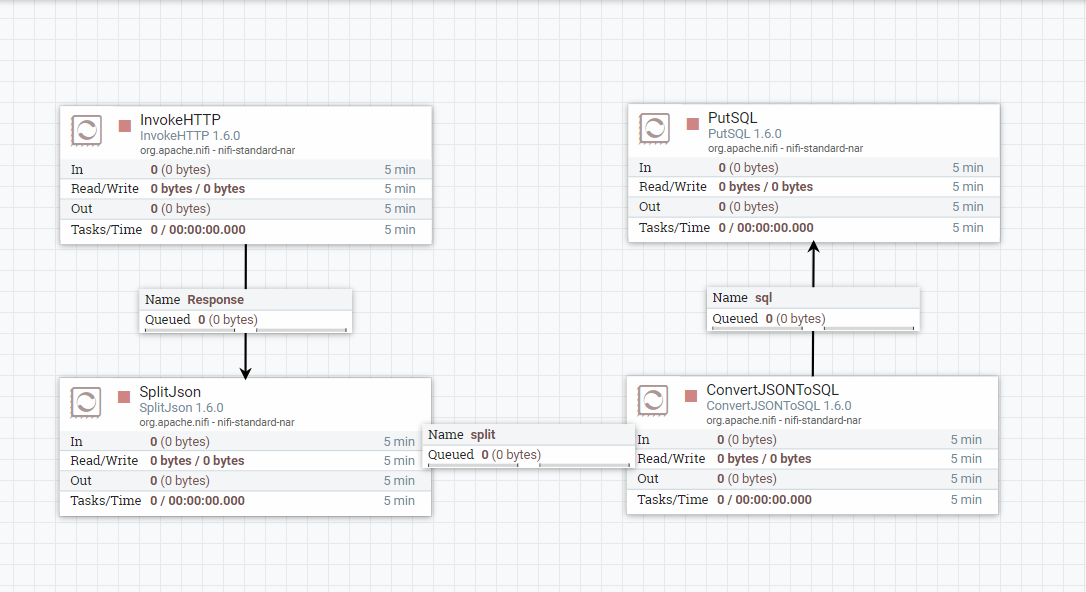
· Index ：设置ES库的Index

· Type ：设置ES库的Type

【查询】http://192.168.6.244:9200/index1/\_search?pretty

## Elasticsearch To MySQL

1. 整体流程图



涉及到的处理器以及功能   
- InvokeHTTP:从ES库中查取数据

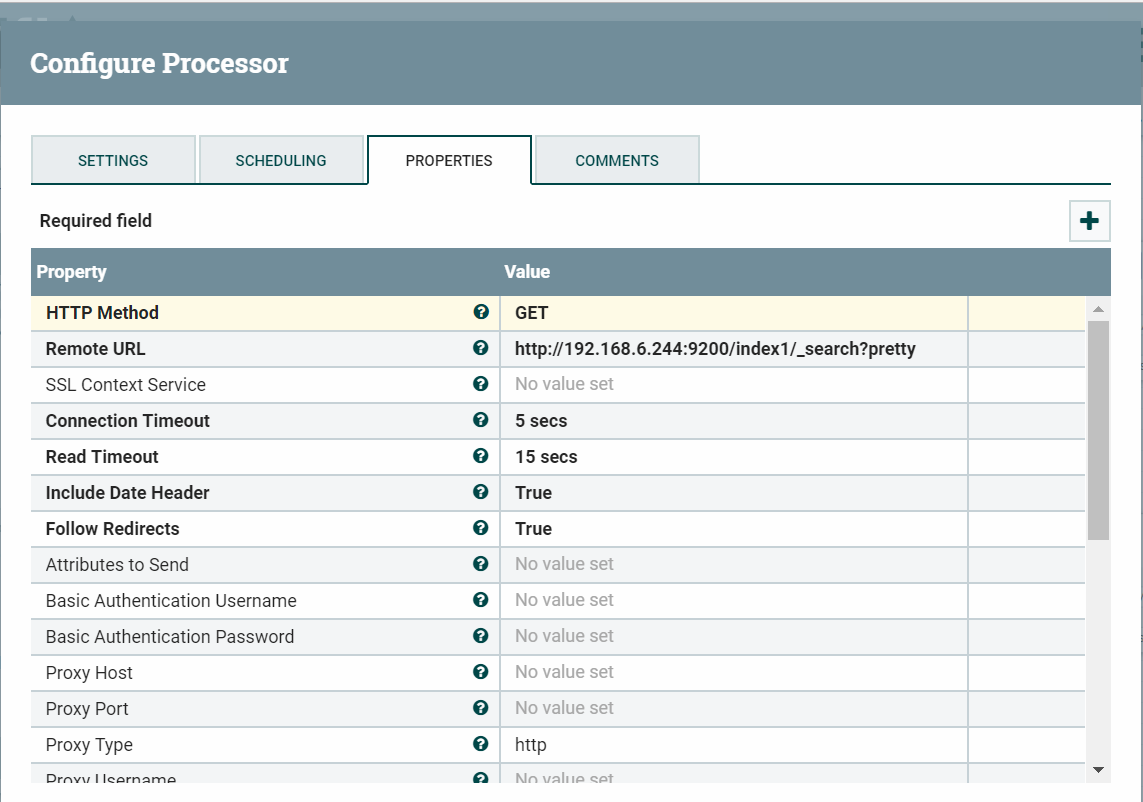
- Splitjson：将Json文件拆分为多个独立的FlowFiles

- ConvertJSONToSQL：将Json格式的数据转换为SQL语句

- PutSQL：将数据插入到MySQL数据库中

2.细节说明：

1. InvokeHTTP：查询数据

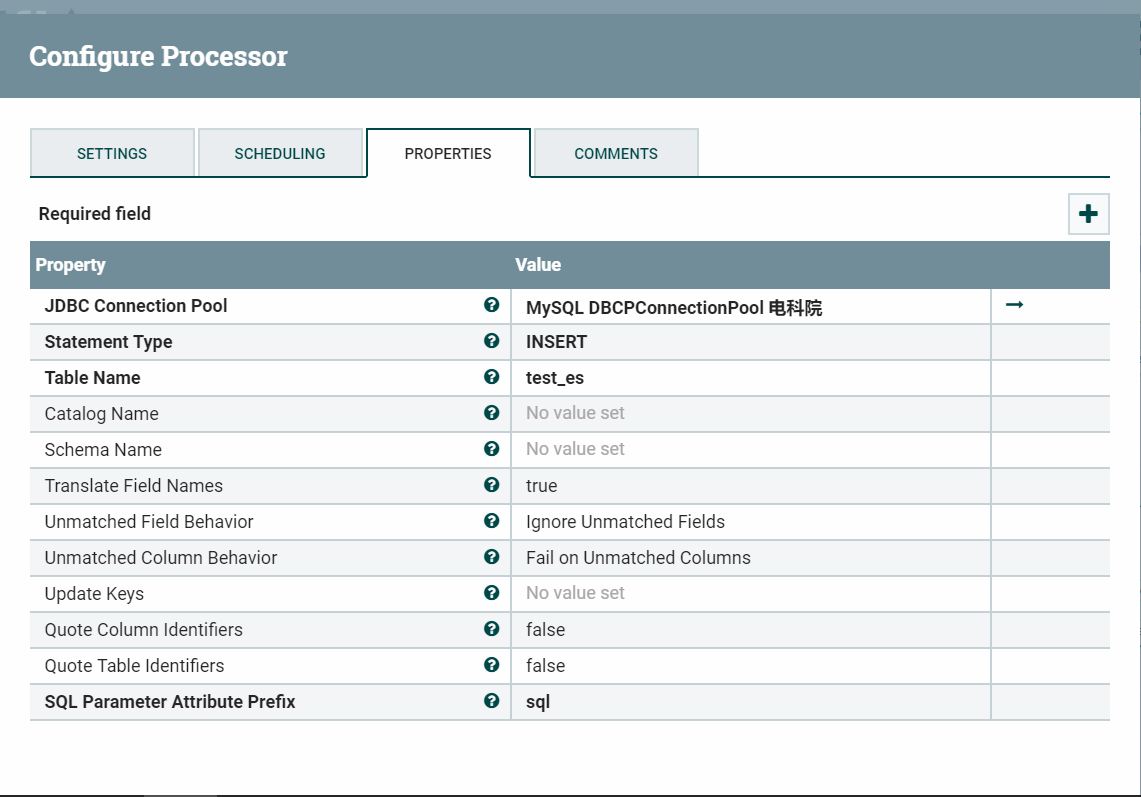


· HTTP Method ：采用GET请求方式

· Remote URL ： ES查询 Rest API

（2）Splitjson：默认设置

（3）ConvertJSONToSQL：

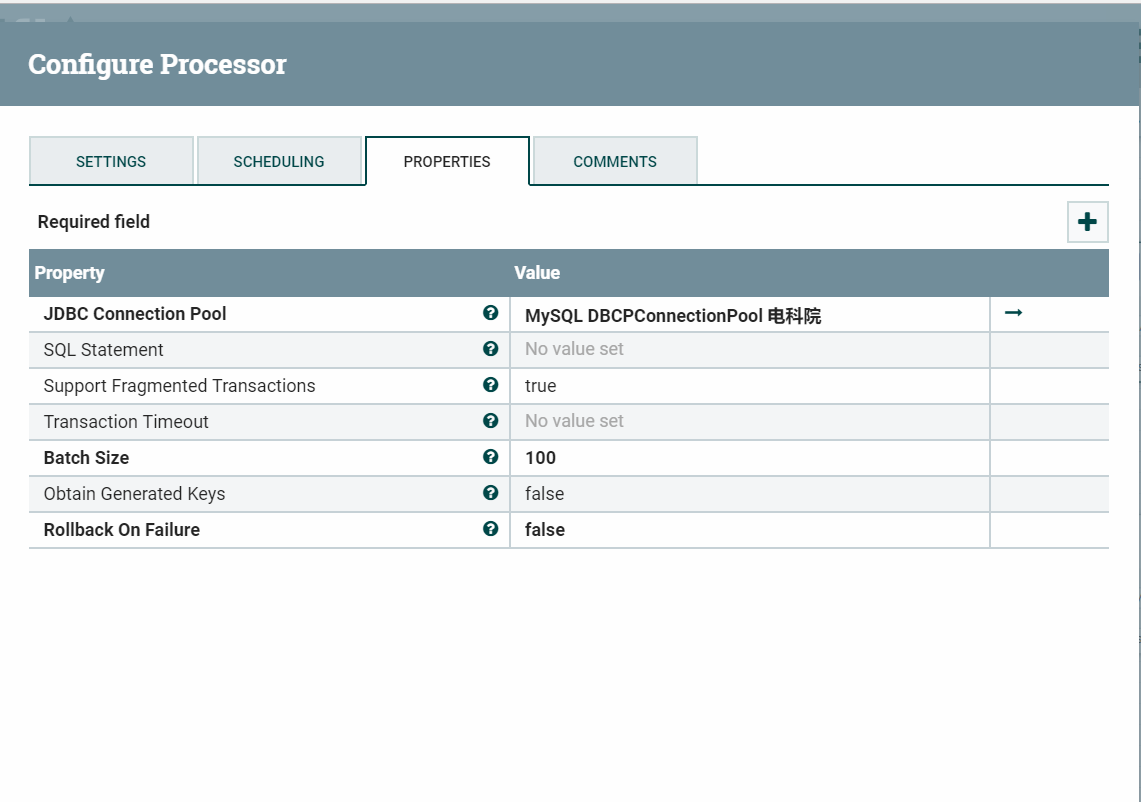


· JDBC Connection Pool ：配置MySQL数据库连接

· Statement Type ：执行INSERT 方式

· Table Name ： 数据库的表名

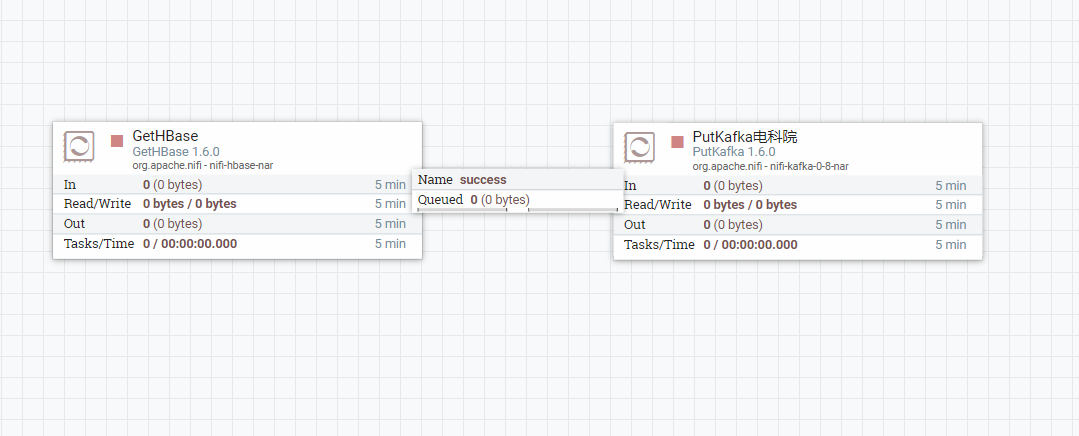
1. PutSQL



· JDBC Connection Pool ：配置MySQL数据库连接

## hbase To Kafka

1. 整体流程图

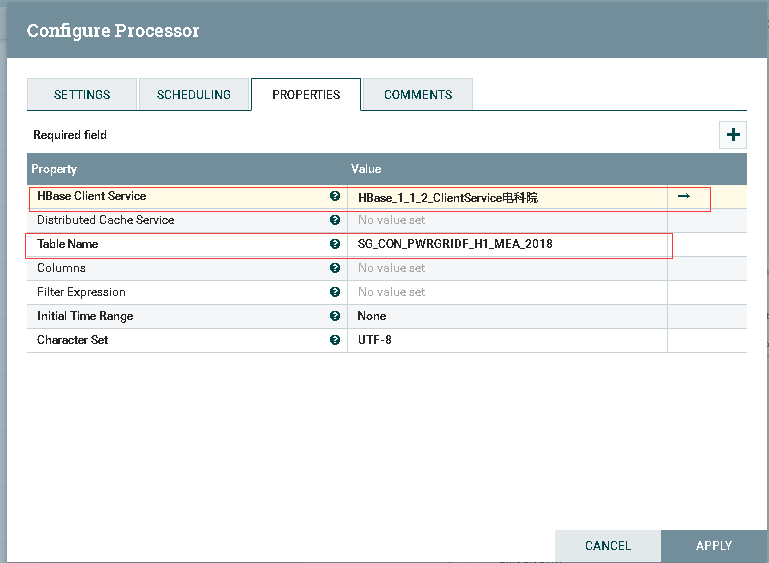


涉及到的处理器以及功能   
- GetHBase:为HBase查询指定表中的任何记录

- PutKafka：将FlowFile的内容作为消息发送到Apache Kafka

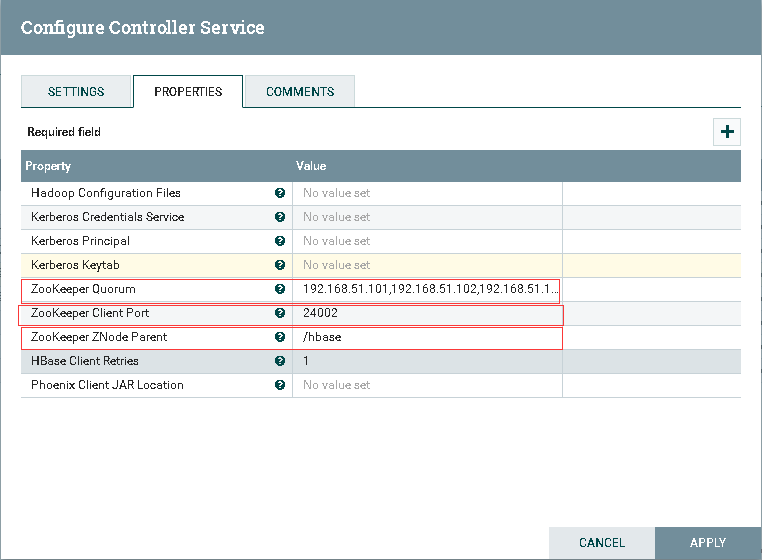
1. 细节说明

（1）GetHBase：查询hbase中的数据



· Table Name是所要查询hbase中的表名

· HBase Client Service用于连接hbase，需要创建一个连接



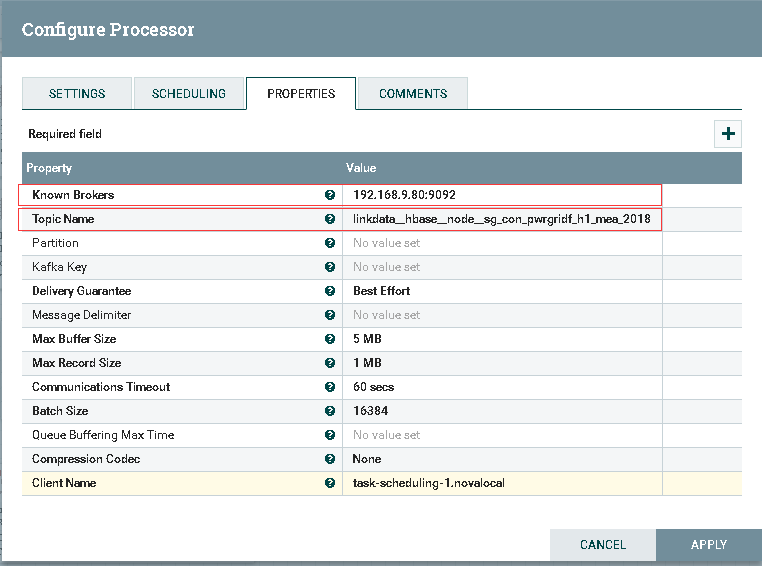
创建hbase的连接配置如下

· ZooKeeper Quorum是zookeeper的ip地址列表

· ZooKeeper Client Port是zookeeper的端口号

· ZooKeeper ZNode Parent是hbase在zookeeper中的节点目录

1. PutKafka：将数据发布到kafka



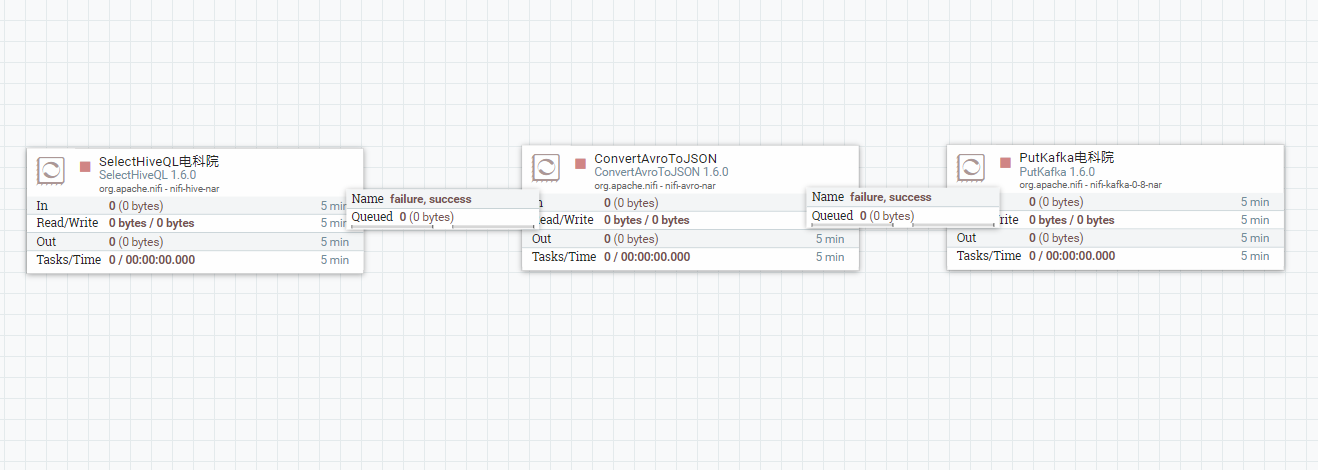
配置方式如下，

· Known Brokers是连接kafka的ip与端口；

· Topic Name是发布到kafka上的topic名称。

## Hive To Kafka

1. 整体流程图



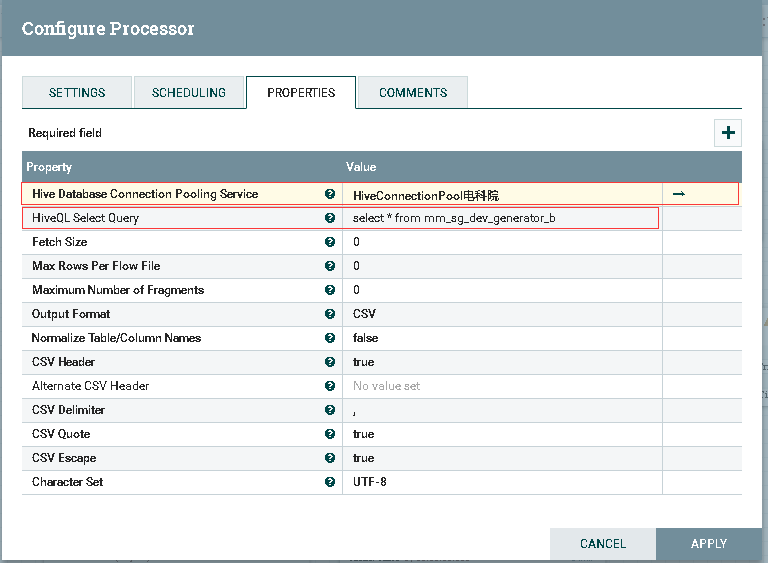
涉及到的处理器以及功能   
- SelectHiveQL:为HBase查询指定表中的任何记录

- ConvertAvroToJson：将查询出来的数据转换成Json格式

- PutKafka：将FlowFile的内容作为消息发送到Apache Kafka

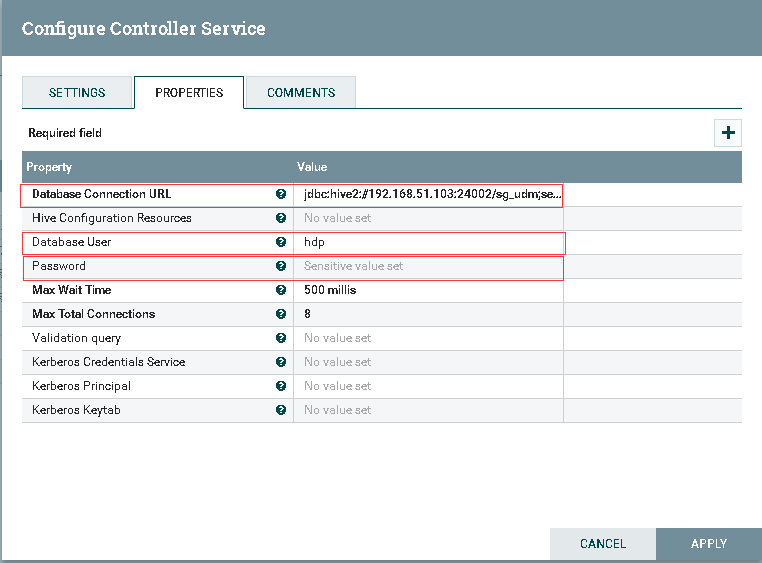
2. 细节说明

(1)SelectHiveQL：查询hive中的数据



· HiveQL Select Query ：查询hive中数据的查询语句。

· Hive Database Connection Pooling Service：用于连接hive，需要创建一个连接



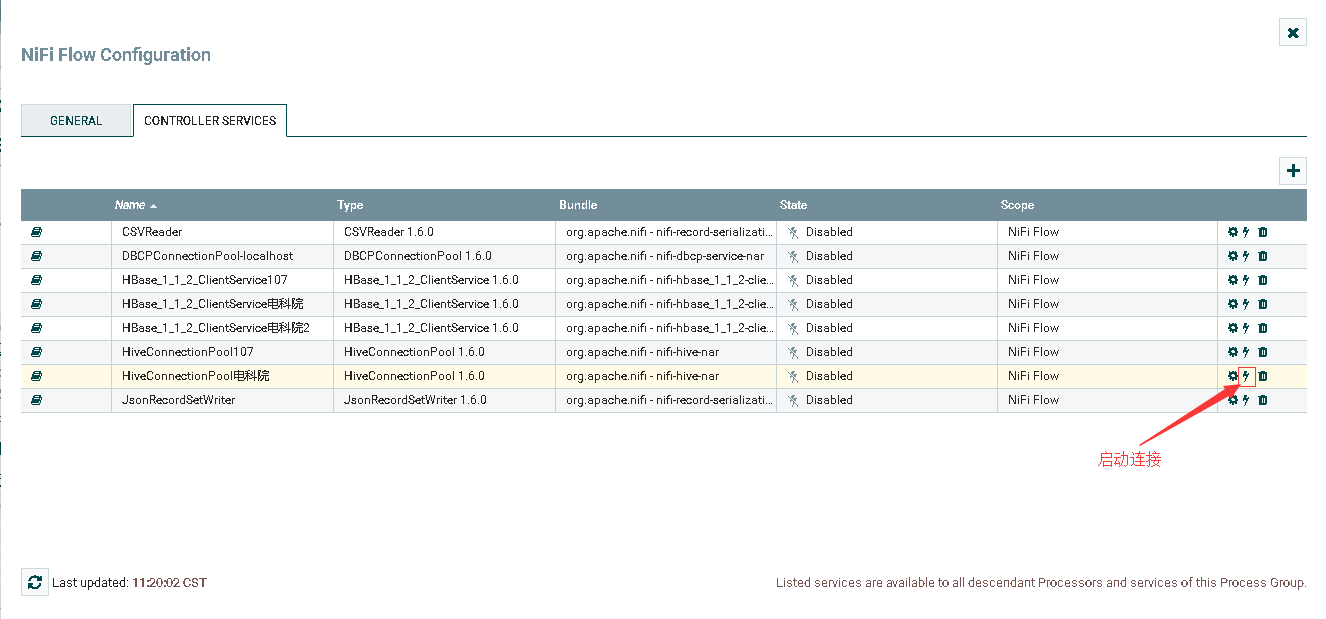
Hive连接配置如下，

· Database Connection URL是连接hive的url

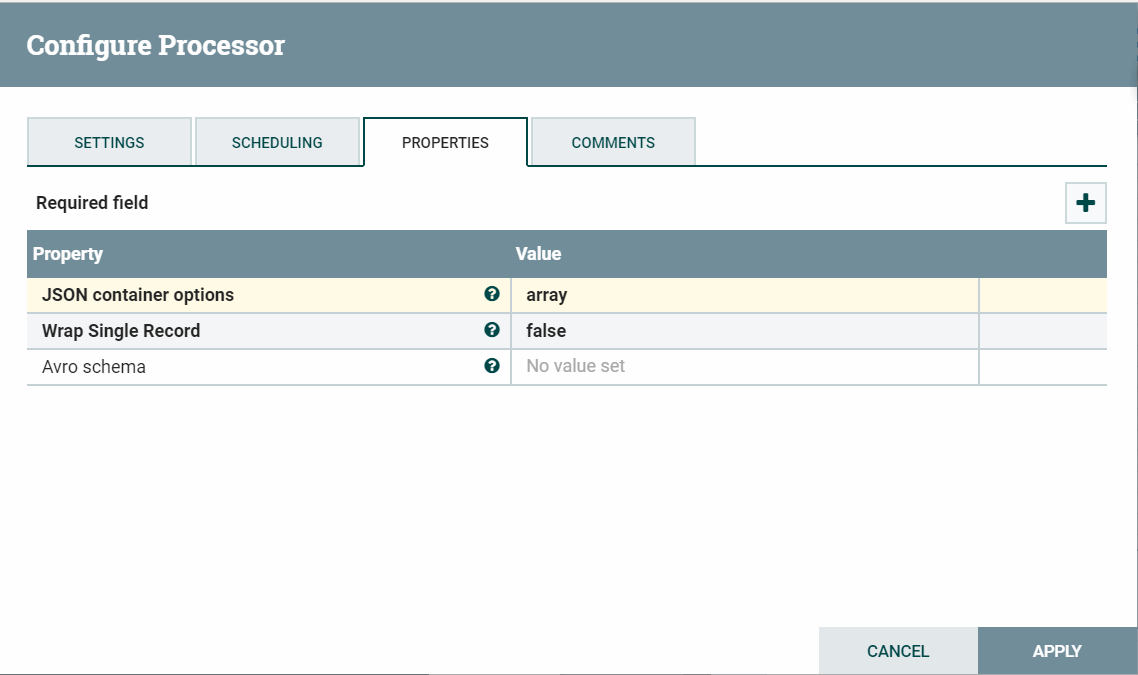
· Database User是连接hive的用户名

· Password是连接hive的密码

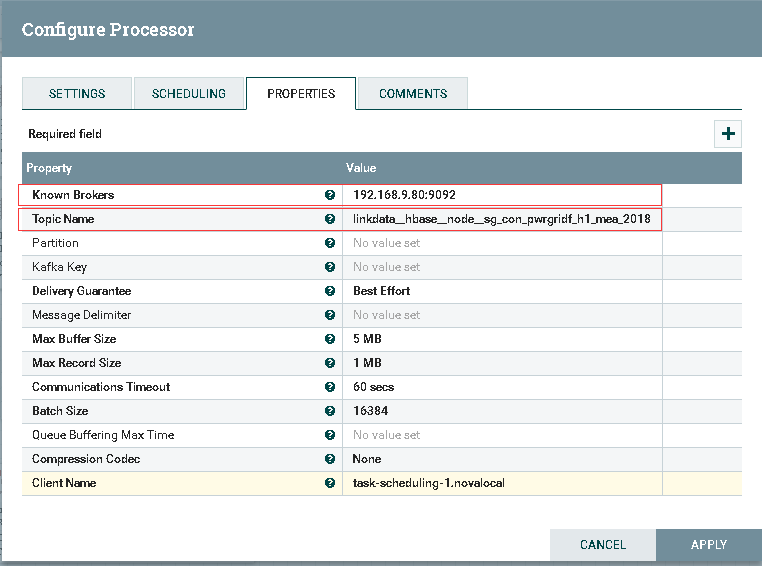
配置完成后需要启动连接，



1. ConvertAvroToJSON



（3）PutKafka：将数据发布到kafka



· Known Brokers：连接kafka的ip与端口

· Topic Name是发布到kafka上的topic名称。

# 组件扩展开发

## 开始

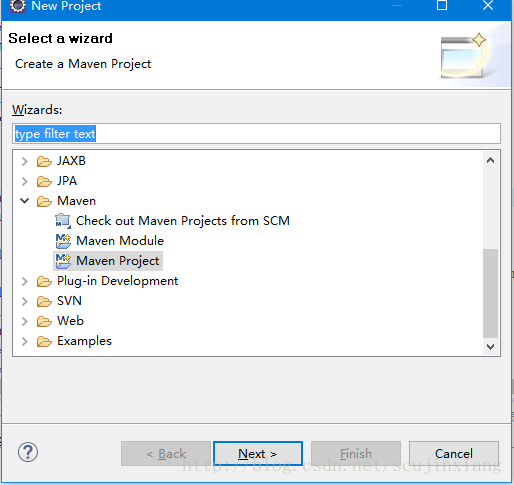
Nifi有很多可用的、文档化的Processor资源，但是某些时候你依然需要去开发属于你自己的Processor，例如从某些特殊的数据库中提取数据、提取不常见的文件格式，或者其他特殊情况。

## 项目依赖

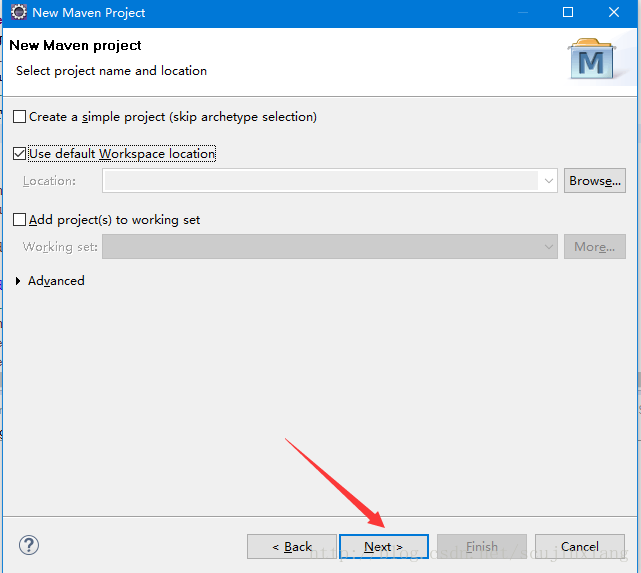
本文以Eclipse开发为例，创建了一个基础的json文件读取Processor，将内容转化为属性值。

（1）安装JDK8，Maven，Eclipse配置Maven管理工具

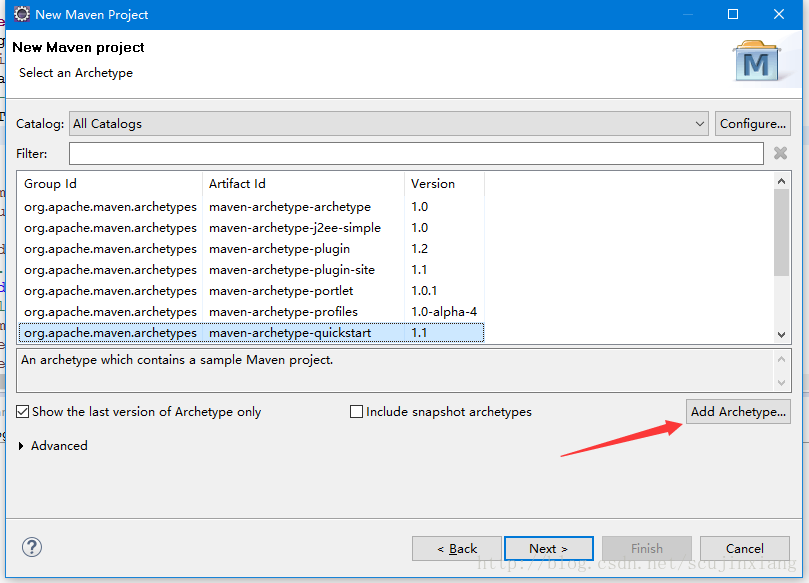
（2）Eclipse新建Maven项目，如下图所示



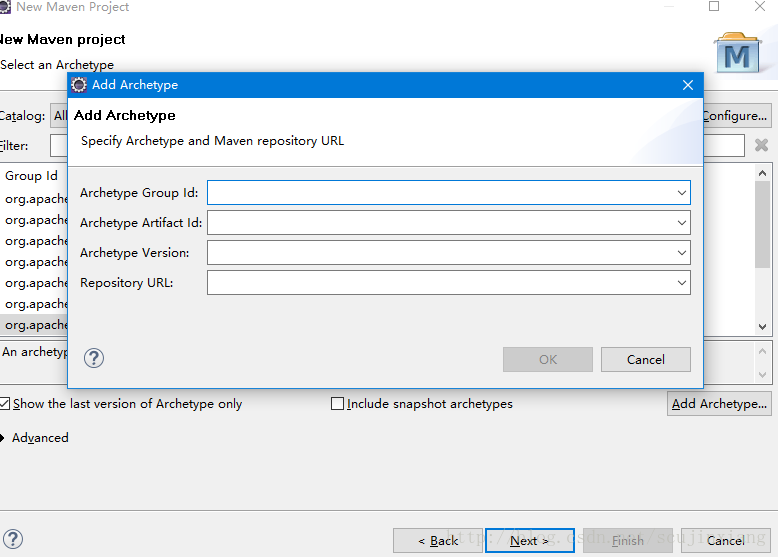
1. 点击next，进入如下图所示页面



1. 默认选项，点击next进入如下页面



1. 点击上图中箭头所指的按钮Add Archetype进入如下所示页面



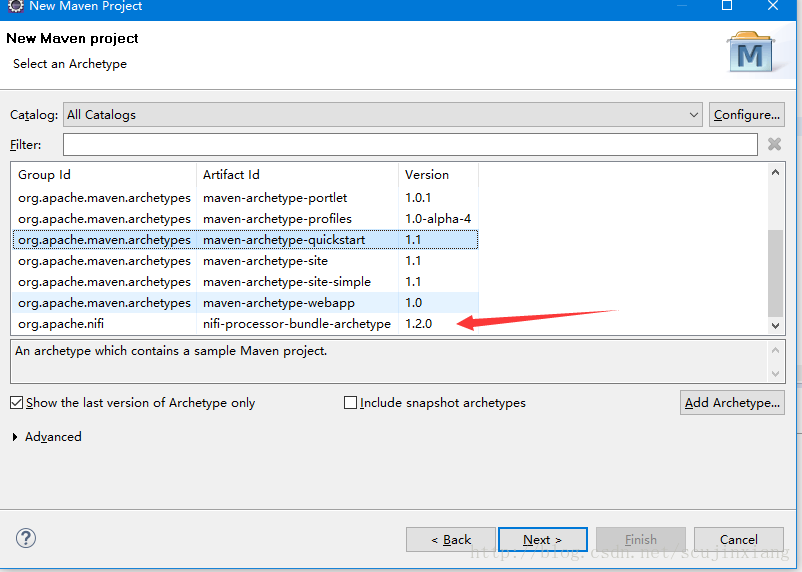
1. 填写：

Archetype Group Id：org.apache.nifi

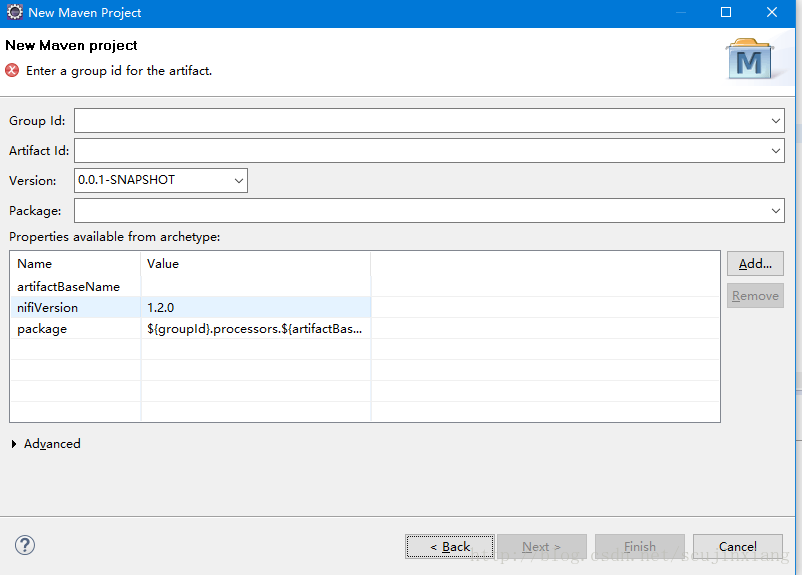
Archetype Artifact Id：nifi-processor-bundle-archetype

Archetype Version：1.2.0

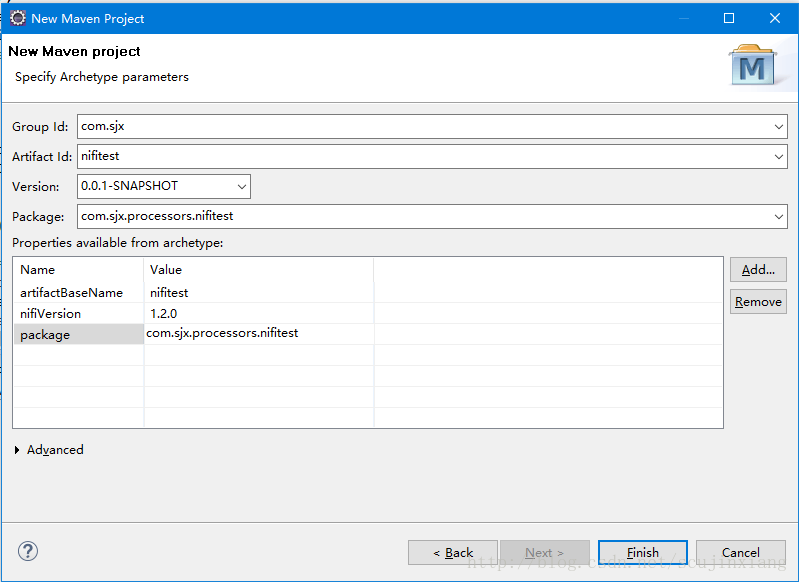
点击OK,可以看到nifi的archetype已经添加到meven中了。



1. 选择org.apache.nifi，点击Next，如下图所示



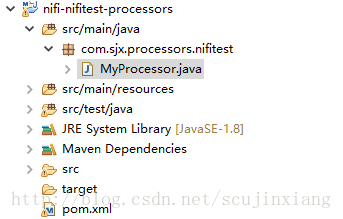
1. 填写项目信息后点击finish完成项目创建



1. 创建完成以后项目下新生成3个目录，我们要开发的东西在nifi-nifitest-processors中完成。

20180116135727395

1. 打开nifi-nifitest-processors目录如下：



## JSON Processor

现在自定义Nifi Processor的前期准备工作都做完了，可以开始构建属于我们自己的Processor了。

（1）在包下新加类起名JsonProcessor使之继承AbstractProcessor

@Tag标签是为了在web GUI中，能够使用搜索的方式快速找到我们自己定义的这个Processor。

@CapabilityDescription内的值会暂时在Processor选择的那个页面中，相当于一个备注。

一般来说只需要继承AbstractProcessor就可以了，但是某些复杂的任务可能需要去继承更底层的AbstractSessionFactoryProcessor这个抽象类。

@Tags({"JSON"})//快速搜索标签

@CapabilityDescription("提取的json文件的属性")

@SeeAlso({})

@ReadsAttributes({@ReadsAttribute(attribute="",description="")})

@WritesAttributes({@WritesAttribute(attribute="",description="")})

public class JsonProcessor extends AbstractProcessor{

（2）新建几个PropertyDescriptor（接受页面配置的参数，如果不需要，可以不进行配置）

public static final PropertyDescriptor JSON\_PATH =

new PropertyDescriptor

.Builder().name("Json Path")

.required(true)

.addValidator(StandardValidators.NON\_EMPTY\_VALIDATOR)

.build();

（3）新建几个Relationship（输出状态，成功或者失败或者其他）

public static final Relationship SUCCESS = new Relationship.Builder()

.name("success").description("SUCCESS")

.build();

public static final Relationship FAILURE = new Relationship.Builder()

.name("failure").description("FAILURE")

.build();

1. 定义两个集合添加上面创建的PropertyDescriptor和Relationship

protected void init(final ProcessorInitializationContext context) {

final List<PropertyDescriptor> descriptors = new ArrayList<PropertyDescriptor>();

descriptors.add(JSON\_PATH);

this.descriptors = Collections.unmodifiableList(descriptors);

final Set<Relationship> relationships = new HashSet<Relationship>();

relationships.add(SUCCESS);

relationships.add(FAILURE);

this.relationships = Collections.unmodifiableSet(relationships);

}

1. 添加新创建属性的get方法

public Set<Relationship> getRelationships(){

return relationships;

}

public final List<PropertyDescriptor> getSupportedPropertyDescriptors(){

return descriptors;

}

如上是初始化Nifi进程，由于Nifi是高度并发条件，所以descriptors和relationship是存储在一个不可变的集合中。

1. onTrigger方法中实现自己的业务，onTrigger方法会在一个flowfile被传入处理器 时调用。

public void onTrigger(ProcessContext context, ProcessSession session) throws ProcessException {

final AtomicReference<String> value = new AtomicReference<>();

//获取flowFile中的内容

FlowFile flowFile = session.get();

session.read(flowFile, in ->{

try {

String json = IOUtils.toString(in);

String result = JsonPath.read(json, "$.hello");

value.set(result);

} catch (Exception e) {

e.printStackTrace();

getLogger().error("Failed to read json string");

}

});

//将读取json数据写入flowFile中

String results = value.get();

if(results != null && !results.isEmpty()) {

flowFile = session.putAttribute(flowFile, "match", results);

}

//将处理结果返回flowFile

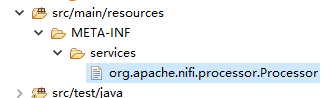
flowFile = session.write(flowFile, out -> out.write(value.get().getBytes()));

session.transfer(flowFile , SUCCESS);

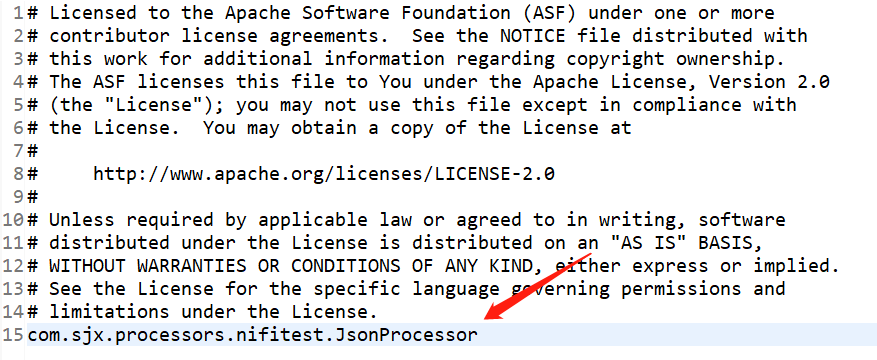
}

## 打包部署

找到文件org.apache.nifi.processor.Processor



在里面添加：包名+类名，将processor暴露出来



Maven运行nifitest，会在项目nifi-nifitest-nar下生成一个nar的包，将包放在nifi目录下的lib目录中，重新启动nifi服务器即可查看到自己添加的processor了。

## 单元测试

Apache Nifi框架的单元测试是基于Junit的Test Runners的，在这一阶段，我们会将单元测试功能加入我们之前创建的JsonProcessor中。

### （1）实例化TestRunner

处理器或控制器服务的大多数单元测试都是通过创建 TestRunner 类的实例来开始的。为了向处理器添加必要的类，我们需要在maven中添加nifi对应的依赖：

<dependency>

<groupId>org.apache.nifi</groupId>

<artifactId>nifi-mock</artifactId>

<version>${nifi version}</version>

</dependency>

（2）在测试中，有几个org.apache.nifi.utils包是需要被import的，比如TestRunner、TestRunners、MockFlowFile这三个类。

（3）在测试方法上添加@Test标签，在添加了这个JUnit 标签后，就可以在方法中去初始化Nifi提供的TestRunner等组件了。

（4）创建一个TestRunner类，然后把自定义的Processor传给它，接着对其的PropertiesDescription进行传值，为了测试可以模拟一个本地的json文件作为资源文件。

（5）当一个test runner创建时，使用runner.setProperties(PropertyDescriptor)以及runner.enqueue(content)进行值赋予。然后使用一些断言进行单元测试，测试结果情况。

public class TestProcessor{

@Test

public void testOnTrigger() throws IOException {

//json文件内容

InputStream content = new ByteArrayInputStream("{\"hello\":\"nifi rocks\"}".getBytes());

//模拟处理器

TestRunner runner = TestRunners.newTestRunner(new JsonProcessor());

//设置JSON\_PATH

runner.setProperty(JsonProcessor.JSON\_PATH, "$.hello");

//向处理器添加内容

runner.enqueue(content);

runner.run(1);

runner.assertQueueEmpty();

List<MockFlowFile> results = runner.getFlowFilesForRelationship(JsonProcessor.SUCCESS);

assertTrue("1 match", results.size() == 1);

MockFlowFile result = results.get(0);

String resultValue = new String(runner.getContentAsByteArray(result));

System.out.println("Match: " + IOUtils.toString(runner.getContentAsByteArray(result)));

//测试属性和内容

result.assertAttributeEquals(JsonProcessor.MATCH\_ATTR, "nifi rocks");

result.assertContentEquals("nifi rocks");

}

}

# Nifi RestAPI

为方便用户使用 NiFi 进行二次开发，NiFi 为开发者提供了 [NIFI RestAPI](http://nifi.apache.org/docs/nifi-docs/)。Rest Api提供实时命令和控制NiFi实例的编程访问。启动和停止处理器，监视队列，查询起源数据等。

## 5.1. 控制器

· 创建一个新公告

/controller/bulletin

· 获取集群的内容

/controller/cluster

· 获取集群中的节点

/controller/cluster/nodes/{id}

· 检索此NiFi控制器的配置

/controller/config

· 创建一个新的控制器服务

/controller/controller-services

· 清除历史

/controller/history

· 获取可用注册表客户端的列表

/controller/registry-clients

· 获取注册表客户端

/controller/registry-clients/{id}

· 创建新的报告任务

/controller/reporting-tasks

## 5.2. 控制器服务

获取控制器的服务

[/controller-services/{id}](http://localhost/controller-services/%7bid%7d)

获取控制器服务属性描述符  
/controller-services/{id}/descriptors

获取控制器服务的状态  
/controller-services/{id}/state

## 5.3. 报告任务

· 获取报告任务

/reporting-tasks/{id}

· 获取报告任务属性描述符

/reporting-tasks/{id}/descriptors

· 获取报告任务的状态

/reporting-tasks/{id}/state

· 清除报告任务的状态

/reporting-tasks/{id}/state/clear-requests

## 5.4. 服务器

· 获取处理器

/processors/{id}

· 获取处理器属性的描述符

/processors/{id}/descriptors

· 获取有关处理器的诊断信息

/processors/{id}/diagnostics

· 获取处理器的状态

/processors/{id}/state

## · 清除处理器的状态

## /processors/{id}/state/clear-requests

· 终止处理器，实质上是“删除”其线程和任何活动任务

/processors/{id}/threads

## 5.5. 连接

· 获取连接  
/connections/{id}

## 5.6. FlowFile 队列

· 创建删除此连接中队列内容的请求。  
/flowfile-queues/{id}/drop-requests

· 获取指定连接的丢弃请求的当前状态  
/flowfile-queues/{id}/drop-requests/{drop-request-id}

· 从Connection获取FlowFile

/flowfile-queues/{id}/flowfiles/{flowfile-uuid}

· 获取Connection中FlowFile的内容

/flowfile-queues/{id}/flowfiles/{flowfile-uuid}/content

· 列出此连接中队列的内容

/flowfile-queues/{id}/listing-requests

· 获取指定连接的列表请求的当前状态

/flowfile-queues/{id}/listing-requests/{listing-request-id}

# 集群压力测试

## 压力测试场景

1.1背景

考虑到大数据管理平台有数据接入量大、数据源多样化、对数据的完整性和容错率要求高、延迟率低等特点，因此计划对Nifi的数据完整性、异常状态下的容错性以及服务器在高负载情况下的性能做一个全面的测试评估，以便于了解nifi的优点和缺陷，从而优化整个大数据管理平台架构。   
1.2测试概要   
（1） 测试环境   
Nifi 1.6.0 集群版(4个节点)

（2）测试目标   
· 数据完整性测试。   
· 异常状态容错机制测试。   
· 不同负载下的响应时间测试。   
· Nifi集群模式下的主从切换测试。

## 压力测试结果

2.1 积压数据量越大，数据处理性能越差，处理时间随着数据量的增加呈指数级增长。   
2.2 数据是否丢失和连接池最大连接数参数以及批量处理SQL的批次条数有关，这个应该是数据处理代码层面的BUG，和Nifi本身无关。Nifi的数据完整性在小数据量下还是可以的。大数据量时候对参数优化要求显得比较严格。   
2.3 数据处理速度和SQL批量处理的批次条数有关，每批处理的越多，处理性能越好。   
2.4 Nifi自身发生错误： Nifi集群的节点如果有宕机情况，会导致整个集群的任务流程无法启动，主节点挂掉会导致nifi的UI界面不可用。如果在任务执行过程中kill掉某个节点进程，会发生丢失数据情况，必须等待节点重新启动后数据会自动恢复。   
2.5 处理模块发生错误：如：mysql挂掉后，数据会自动在入库操作的上游堆积，等待数据库恢复。数据库恢复后，可以完成自动入库，整体数据无丢失。Kafka挂掉后数据流也会进入等待，直至kafka恢复后数据自动流转。

## 性能调优

3.1 Nifi的数据完整性还是有保障的，测试中出现的数据丢失问题主要是由于现在的代码层面对入库失败的数据未做处理造成的。   
3.2 Nifi集群的处理性能和稳定性远高于Nifi单机模式。   
3.3 Nifi集群的处理性能和数据冗余量有直接关系，即nifi处理数据主要依赖磁盘IO。   
3.4 Nifi自身的集群容错率较低，并非传统的主从结构，但对数据处理模块中的组件容错率较强。