# Особенности использования pyspark в разработке

Для выполнения ETL на источниках данных, не являющимися СУБД удобно использовать python как язык общего назначения.  
Базовая идея такого ETL заключается в написании или использования чужих библиотек по извлечению данных из источника и, затем трансформация и загрузка этих данных через spark (здесь и далее имеется в виду **pyspark**) в hadoop.

Технически, весь процесс представляет из себя кастомизированный **shell-action** в oozie, запускающий **python-скрипт**, который инициализирует **spark-сессию** в режиме **yarn-client**,  осуществляет извлечение данных из источника, затем из извлеченных данных собирается **Spark DataFrame**, осуществляется трансформация и, получившийся DataFrame выгружается на hadoop, с представлением в Hive.

Помимо стандартных требований к стеку продуктов, для использования pyspark должны соблюдаться дополнительные условия.

(полный пример pyspark реализован в wf\_jira\_clientfeedback, wf\_ctl\_jira\_daily, crd\_hist\_jira\_clientfeedback, wf\_reg\_jira\_daily. Исходная задача [*DATALAKE-693*](http://jira.moscow.alfaintra.net/browse/DATALAKE-693) *- Загрузка данных из Jira (этап 1) В очереди на тестирование* )

Вот их краткий список:  
- для пользователя, запускающего oozie job, должен быть сгенерирован и доступен с hdfs **kerberos** **keytab**  
- должен быть известен KERBEROS\_PRINCIPAL  
- Путь до keytab и KERBEROS\_PRINCIPAL должны передаваться в python-скрипт  
- Должны быть заданы переменные окружения для spark: JAVA\_HOME, SPARK\_HOME, PYSPARK\_PYTHON  
- Python-скрипт должен уметь принимать на вход аргументы  
- Python-скрипт должен добавлять в PATH нужные библиотеки для обработки данных и SPARK

## Стандартный процесс при использовании pyspark

Извлечение данных осуществляется либо сторонними/стандартными библиотеками python, либо методами фреймворка pyspark.  
Обработка данных производится стандартным функционалом python, либо методами фреймворка pyspark.  
Результат обработки помещается в DataFrame. Из этого DataFrame создается tempview средствами spark. Такой tempview существует, пока активна Spark-сессия и виден только через pyspark.  
Из tempview осуществляется загрузка на hdfs, либо в hive загружаются данные в temp-таблицы.  
Из temp-таблиц осуществялется перегрузка в целевые таблицы.   
Перегрузку можно осуществить как с помощью pyspark, так и обычными методами hive-action.

## Oozie

Для запуска python скрипта используется shell-action.  
Первая особенность - выделение ресурсов памяти, т.к. практически весь процесс будет происходить в рамках ресурсов, аллоцировнных для этого action.  
Для этого, в разделе <configuration> reg- и man- потоков следует указать дополнительные property:

<configuration>

....

<property>

<name>oozie.launcher.yarn.app.mapreduce.am.resource.mb</name>

<value>4096</value>

</property>

<property>

<name>oozie.launcher.mapreduce.map.java.opts</name>

<value>-Xmx4096m</value>

</property>

...

</configuration>

Значение для параметров зависят от размера задачи. В сайзинге стоит учесть и то, что Spark-сессия создается в режиме yarn-client. Драйвером (основным потребителем памяти) в таком случае будет   
наш python-процесс, запускающийся в контейнере из shell-action.

Если предполагается, что spark должен взаимодействовать с hive, в shell-action нужно будет передать hcat-credentials и затем их использовать.

Разберем пример из workflow.xml (расшифровка указана в комментариях к строкам):

...

<credentials>

<credential name="hcat" type="hcat">

<property>

<name>hcat.metastore.uri</name>

<value>${hcat\_metastore\_uri}</value>

</property>

<property>

<name>hcat.metastore.principal</name>

<value>${hive2\_server\_principal}</value>

</property>

</credential>

</credentials>

...

<start to="shell-clientfeedback"/>

<kill name="Kill">

<message>Action failed, error message[${wf:errorMessage(wf:lastErrorNode())}]</message>

</kill>

<!--Run Python Script-->

<action name="shell-clientfeedback" cred="hcat">

<shell xmlns="uri:oozie:shell-action:0.1">

<job-tracker>${jobTracker}</job-tracker>

<name-node>${nameNode}</name-node>

<exec>issues\_and\_comments.py</exec> <!--(обязательно) короткое имя запускаемого python-скрипта-->

<argument>--DATABASE</argument> <!--ключ аргумента, принимаемый python-скриптом для моей специфичной задачи-->

<argument>${DATABASE}</argument> <!--значение из start.sh, принимаемое python-скриптом для моей специфичной задачи-->

<argument>--TMP\_DATABASE</argument> <!--ключ аргумента, принимаемый python-скриптом для моей специфичной задачи-->

<argument>${TMP\_DATABASE}</argument> <!--значение из start.sh, принимаемое python-скриптом для моей специфичной задачи-->

<argument>--JIRA\_USER</argument> <!--ключ аргумента, принимаемый python-скриптом для моей специфичной задачи-->

<argument>${JIRA\_USER}</argument> <!--значение из start.sh, принимаемое python-скриптом для моей специфичной задачи-->

<argument>--JIRA\_PASSWORD</argument> <!--ключ аргумента, принимаемый python-скриптом для моей специфичной задачи-->

<argument>${JIRA\_PASSWORD}</argument> <!--значение из start.sh, принимаемое python-скриптом для моей специфичной задачи-->

<argument>--KERBEROS\_PRINCIPAL</argument> <!--(обязательно) ключ аргумента, принимаемый python-скриптом для Kerberos-->

<argument>${KERBEROS\_PRINCIPAL}</argument> <!--(обязательно) значение аргумента, принимаемый python-скриптом для Kerberos-->

<argument>--INITIAL\_LOAD</argument>

<argument>${INITIAL\_LOAD}</argument> <!--значение из start.sh, принимаемое python-скриптом для моей специфичной задачи-->

<argument>--HIVE\_PRINCIPAL</argument> <!--(обязательно, если нужен hive) ключ аргумента, принимаемый python-скриптом-->

<argument>${hive2\_server\_principal}</argument> <!--(обязательно, если нужен hive) значение из hcat cred -->

<argument>--HIVE\_URI</argument> <!--(обязательно, если нужен hive) ключ аргумента, принимаемый python-скриптом-->

<argument>${hcat\_metastore\_uri}</argument> <!--(обязательно, если нужен hive) значение из hcat cred -->

<env-var>JAVA\_HOME=/usr/java/default</env-var> <!-- (обязательно) переменная, требуемая spark (прописана в конфигурации hadoop) -->

<env-var>SPARK\_HOME=/opt/cloudera/parcels/SPARK2/lib/spark2</env-var> <!-- (обязательно) переменная, требуемая spark (прописана в конфигурации hadoop) -->

<env-var>PYSPARK\_PYTHON=/opt/anaconda/bin/python3</env-var> <!-- (обязательно) переменная, требуемая spark (прописана в конфигурации hadoop) -->

<file>pyspark/issues\_and\_comments.py#issues\_and\_comments.py</file> <!--(обязательно) относительный путь до python-скрипта -->

<file>${KERBEROS\_KEYTAB}#KERBEROS\_KEYTAB</file> <!--(обязательно) путь до keytab (передаю из start.sh) -->

<capture-output/>

</shell>

<ok to="End"/>

<error to="Kill"/>

</action>

### Переменные окружения:

Значения переменных окружения должны совпадать на всех узлах кластера!

JAVA\_HOME - константа, может меняться при обновлении Java или Hadoop  
SPARK\_HOME - константа, может меняться при обновлении Hadoop или Spark  
PYSPARK\_PYTHON - указывает в каком python-интерпретаторе запускается SPARK-сессия. Должен совпадать с требуемой версией python. Это путь должен быть одинаков на всех нодах!

## Python

Сам скрипт должен быть executable.

Первая строка должна содержать ссылку на интерпретатор, в котором запускается этот скрипт. Этот путь аналогичен переменной PYSPARK\_PYTHON  
Для того, чтобы удовлетворить условиям, описанным выше, python скрипт должен быть вида:

#!/opt/anaconda/bin/python3

import argparse # Импорт библиотеки для парсинга входных аргументов

import sys # Импорт библиотеки для добавления в PATH нужных библиотек для обработки данных и SPARK

import os # Импорт библиотеки для добавления в PATH нужных библиотек для обработки данных и SPARK

parser = argparse.ArgumentParser(description="Arguments")

parser.add\_argument('--DATABASE') # парсинг входных аргументов (соответвует именам в тегах <argument> в workflow.xml)

parser.add\_argument('--TMP\_DATABASE') # парсинг входных аргументов (соответвует именам в тегах <argument> в workflow.xml)

parser.add\_argument('--JIRA\_USER') # парсинг входных аргументов (соответвует именам в тегах <argument> в workflow.xml)

parser.add\_argument('--JIRA\_PASSWORD') # парсинг входных аргументов (соответвует именам в тегах <argument> в workflow.xml)

parser.add\_argument('--INITIAL\_LOAD') # парсинг входных аргументов (соответвует именам в тегах <argument> в workflow.xml)

parser.add\_argument('--KERBEROS\_PRINCIPAL') # (обязательно) парсинг входных аргументов (соответвует именам в тегах <argument> в workflow.xml)

parser.add\_argument('--HIVE\_PRINCIPAL') # (если нужна поддержка hive) парсинг входных аргументов (соответвует именам в тегах <argument> в workflow.xml)

parser.add\_argument('--HIVE\_URI') # (если нужна поддержка hive) парсинг входных аргументов (соответвует именам в тегах <argument> в workflow.xml)

args = parser.parse\_args()

# Workaround for spark

# В этом code-сниппете добавляются библиотеки SPARK в PATH (иначе, import spark работать не будет!)

if os.environ.get('SPARK\_HOME'):

PYSPARK\_PYTHON = os.environ['SPARK\_HOME']+'/python' # Должно получиться что-то типа /opt/cloudera/parcels/SPARK2/lib/spark2/python

sys.path.append(PYSPARK\_PYTHON)

sys.path.append(PYSPARK\_PYTHON+'/lib/py4j-0.10.4-src.zip') # Должно получиться что-то типа /opt/cloudera/parcels/SPARK2/lib/spark2/python/lib/py4j-0.10.4-src.zip

# HDFS OOZIE Workaround for local modules

# В этом code-сниппете добавляются уже собственные библиотеки в PATH. Фактически, добавляется текущий каталог выполнения и папка /lib

sys.path.append(os.getcwd()) # atom\_path

sys.path.append(os.getcwd()+'/lib') # atom\_path/lib

## Работа SPARK в python и хитрости

Для работы с pyspark необходимо как-минимум создать сессию.  
Для этого нужен 1 базовый импорт:

from pyspark.sql import SparkSession

И код, инициализирующий сессию:

try:

spark = SparkSession.builder\

.appName('jiraEtl') \

.master('yarn-client') \

.config('spark.yarn.principal', args.KERBEROS\_PRINCIPAL) \

.config('spark.yarn.keytab', 'KERBEROS\_KEYTAB') \

.config('spark.driver.memory', '4096m') \

.config('spark.yarn.executor.memoryOverhead', '16192m') \

.config('hive.metastore.sasl.enabled', 'true') \

.config('hive.security.authorization.enabled', 'false') \

.config('hive.metastore.kerberos.principal', args.HIVE\_PRINCIPAL) \

.config('hive.metastore.uris', args.HIVE\_URI) \

.config('hive.metastore.execute.setugi', 'true') \

.enableHiveSupport() \

.getOrCreate()

except Exception as e:

print("Cannot create session, because of: ", e)

exit(1)

Где   
jiraEtl - имя сессии.   
yarn-client - тип подключения, константа.

метод .config() - принимает два аргумента: проперти и его значение.  
'spark.yarn.principal' - имя kerberos principal (берется по цепочке start.sh->shell-action->argparse)  
'spark.yarn.keytab' - указатель на путь до keytab-файла (описан выше). Т.к. в shell-action указан как '#KERBEROS\_KEYTAB' - можно обращаться по относительному пути.  
'spark.driver.memory' - аналогично проперти [oozie.launcher.yarn.app.mapreduce.am](http://oozie.launcher.yarn.app.mapreduce.am).resource.mb  
'spark.yarn.executor.memoryOverhead' - значение, на сколько можно превысить spark.driver.memory  
Далее следют property, специфичные **для** **Hive**:  
'hive.metastore.sasl.enabled' - константа, для текущего сетапа hadoop - true  
'hive.security.authorization.enabled' - контанта. При kerberos - false  
'hive.metastore.kerberos.principal' - значение берется из hcat в shell-action ${hive2\_server\_principal}  
'hive.metastore.uris' - значение берется из hcat в shell-action ${hcat\_metastore\_uri}  
'hive.metastore.execute.setugi' - константа, true  
.enableHiveSupport() - метод, показывающий, что сессия создается с поддержкой hive.  
.getOrCreate() - метод создает сессию.

Для отладки локально сессия создается следующим образом:

spark = SparkSession.builder\

.appName('testapp') \

.master("local") \

.getOrCreate()

## Извлечение данных из REST-сервиса

При работе с REST-API в python обычно используются библиотеки **requests** и **json**.  
requests - осуществляет подключение к источнику.  
json - парсит ответ от источника  
Фактически, ответ от источника - dict() в python.

Есть несколько специфичных моментов при работе с иточником данных по REST.

Порядок и состав полей могут меняться.

На примере Jira Rest Api - при получении информации о задаче, если в ней нет комментариев,   
поле comments вообще не будет получено. Из чего следует, что нужно проводить какие-либо проверки на качество полученных данных.

В spark.sql при обработке DataFrame есть метод explode() - он разворачивает списки из колонок.  
У него есть особенность - он не разворачивает "пустые" списки, которые могут придти от REST-сервиса.

Фактически, строки, для которых НЕ будет значений, будут просто сброшены из DataFrame.

Для решения этой проблемы есть хук - определить правильный метод explode:

from pyspark.sql.column import \_to\_java\_column

def explode\_outer\_(col):

\_explode\_outer = spark.sparkContext.\_jvm.org.apache.spark.sql.functions.explode\_outer

return Column(\_explode\_outer(\_to\_java\_column(col)))

Функция explode\_outer() заполнит пустые поля nullами.

Еще одной особенностью rest-api является объем передаваемых данных. Т.к. общение происходит по HTTP протоколу хорошим тоном будет:  
- использовать "постраничный" вывод информации;  
- проходить аутентификацию на сервисе единожды, а затем использовать cookie или другой способ хранения AUTH-данных в последующих запросах;  
- реализовывать инкрементальную загрузку на основе дат (например, фильтровать изменения с датой обновления позже или равной предыдущей загрузке);  
- ограничивать количество одновременных и последовательных запросов к источнику (использовать timeout и другие лимиты)