mobigen
Make The Best For Smart Business

IRIS 로그분석기 미들웨어

개발자 매뉴얼

Version: 0.0.1
November 2017

Contents

1 개요

1.1 IRIS 로그분석기 미들웨어 version

본 문서는 IRIS 로그분석기 미들웨어 0.8.0 이상의 버전에 적용됩니다.

1.2 About this document

IRIS 로그분석기 미들웨어에 외부 명령어를 개발 및 추가 하는 방법에 대해 기술하고 있습니다.

본 문서의 구성은 다음과 같습니다.

Chapter 1 – 개요
본 문서의 구성에 대하여 개략적으로 소개하였습니다.

Chapter 2 – IRIS 로그분석기 미들웨어의 설치
IRIS 로그분석기 미들웨어 시스템의 외부 명령어 개발을 위해 미들웨어 시스템을 설치하고 실행하는 과정을 설명합니다.

Chapter 3 – 외부 명령어 개발
IRIS 로그분석기 미들웨어 시스템의 외부 명령어 개발과정에 대해 예제를 통해 구현, 실행 및 단위 테스트하는 과정을 설명합니다.

Chapter 4 – 기본 API 상세 소개
IRIS 로그분석기 미들웨어 시스템의 외부 명령어 개발에 참고 할 수 있는 내용을 기술하고 있습니다.

2 IRIS 로그분석기 미들웨어의 설치

IRIS 로그분석기 미들웨어는 **Linux/Unix** 환경에서 구동하도록 제작되었습니다. 따라서 Windows 에서 개발을 지원하지 않습니다. IRIS 로그분석기 미들웨어는 **Python 2.7.x**와 **Spark 2.x**를 기반으로 만들어져 있습니다. Spark의 구동을 위해 **JavaSE 1.8** 이상의 JRE 혹은 JDK가 필요합니다.

2.1 Standalone

IRIS 로그분석기 미들웨어를 Linux/Unix 환경에서 단일 실행하기 위한 설치방법과 환경설정에 대해 설명합니다.

2.1.1 Spark 설치와 PySpark 환경 설정

IRIS 로그분석기 미들웨어의 구동을 위해서는 PySpark의 사용이 가능해야 합니다. PySpark는 Python으로 Spark를 사용할 수 있게 해주는 라이브러리로 Spark에 기본적으로 포함되어 있습니다.

개발 환경 설정에서 사용 할 Spark의 버전을 SPARK_VERSION 변수에 설정 합니다. 2.0.0 이상의 버전을 사용합니다. 해당 변수를 선언 후, wget으로 빌드 된 Spark를 다운 받고 압축을 해제합니다.

```
$ export SPARK_VERSION=2.2.0 &&
wget https://d3kbcqa49mib13.cloudfront.net/spark-$SPARK_VERSION-bin-hadoop2.7.tgz &&
tar xzvf spark-$SPARK_VERSION-bin-hadoop2.7.tgz
```

SPARK_HOME과 PYTHONPATH 를 설정합니다. 해당 환경 변수를 설정하게 되면, PySpark와 같이 실행에 필요한 라이브러리들을 로드 할 수 있게 됩니다.

```
$ export SPARK_HOME=`pwd`/spark-$SPARK_VERSION-bin-hadoop2.7 &&
export PYTHONPATH=$(ZIPS=("$SPARK_HOME/python/lib"/*.zip); IFS=:; echo
"${ZIPS[*]}"):$PYTHONPATH
```

다음과 같이 PySpark 에서 라이브러리 경로가 올바르게 설정되었는지 확인 할 수 있습니다.

```
$ python -c "import pyspark; print pyspark.__version__"
2.2.0
```

2.1.2 IRIS 로그분석기 미들웨어 (a.k.a. angora) 설치

IRIS 로그분석기 미들웨어가 빌드 된 파일, angora.tar.gz 가 미리 로컬에 존재 한다고 가정 합니다. 다음 명령으로 압축을 해제합니다.

```
$ tar xzvf angora.tar.gz
```

압축을 해제하면 다음과 같은 구조의 angora 디렉토리가 생성됩니다.

```
angora
├── bin
│   └── angora
├── conf
│   └── angora.conf.template
├── scripts
│   └── setup.sh
├── examples
│   └── count.py
└── lib
```

bin/angora
IRIS 로그분석기 미들웨어를 실행하거나 종료할 때 사용하는 스크립트입니다.

conf/angora.conf.template
IRIS 로그분석기 미들웨어 설정파일의 템플릿입니다.

scripts/setup.sh
각종 설정을 적용할 때 사용하는 스크립트 입니다.

examples/count.py
IRIS 로그분석기 미들웨어의 외부 명령어 예제 파일입니다.

lib/
IRIS 로그분석기 미들웨어가 사용하는 필수 라이브러리가 위치한 디렉토리입니다.

압축을 해제한 디렉토리를 기준으로 ANGORA_HOME을 설정하고 source scripts/setup.sh를 실행하게 되면, IRIS 로그분석기 미들웨어에 필요한 여러 환경 변수를 설정하게 됩니다.

```
$ export ANGORA_HOME=`pwd`/angora && source angora/scripts/setup.sh
```

설정 파일을 복사합니다. 해당 설정파일은 아이피 및 포트, 로그 경로, 외부 명령어 경로 등 여러 설정 정보를 갖고 있으나, 해당 매뉴얼에서는 일부만 사용합니다.

```
$ cp angora/conf/angora.conf.templete angora/conf/angora.conf
```

angora version show의 명령으로 IRIS 로그분석기 미들웨어의 버전정보가 출력되면, 환경 설정이 올바르게 된 것입니다.

```
$ angora version show
Angora v0.8.0
```

angora engine start 명령어를 사용해 IRIS 로그분석기 미들웨어를 실행합니다.

```
$ angora engine start
```

IRIS 로그분석기 미들웨어가 실행되면, $ANGORA_HOME/logs/angora/angora.log에 로그를 기록합니다. 다음과 같이 로그를 확인하여 미들웨어가 정상적으로 실행되는지 확인합니다.

```
$ tail logs/angora/angora.log
...
2017-10-31 17:41:00,979 - werkzeug - INFO -  * Running on http://0.0.0.0:6036/
(Press CTRL+C to quit)
```

정상 동작이 확인 되었다면, angora engine stop 명령으로 미들웨어를 종료합니다.

```
$ angora engine stop
```

2.1.3 환경 변수 등록

2.1.1 과 **2.1.2** 의 절차를 수행하면 SPARK_HOME, PYTHONPATH 그리고 ANGORA_HOME 의 환경 변수가 정의됩니다. 해당 환경 설정을 ~/.bashrc 혹은 ~/.bash_profile 에 저장해 놓을 수 있습니다.

- bashrc에 등록하는 경우

```
$ echo "export ANGORA_HOME="$ANGORA_HOME >> ~/.bashrc
$ echo "export SPARK_HOME="$SPARK_HOME >> ~/.bashrc
$ echo "export PYTHONPATH="$PYTHONPATH:\$PYTHONPATH >> ~/.bashrc
$ echo "source $ANGORA_HOME/scripts/setup.sh" >> ~/.bashrc
```

- bash_profile에 등록하는 경우

```
$ echo "export ANGORA_HOME="$ANGORA_HOME >> ~/.bash_profile
$ echo "export SPARK_HOME="$SPARK_HOME >> ~/.bash_profile
$ echo "export PYTHONPATH="$PYTHONPATH:\$PYTHONPATH >> ~/.bash_profile
$ echo "source $ANGORA_HOME/scripts/setup.sh" >> ~/.bash_profile
```

위 명령어는 ~/.bashrc 혹은 ~/.bash_profile 에 현재 설정된 환경 설정을 해당 유저 계정 설정에 저장 합니다.

2.2 Docker 버전

IRIS 로그분석기 미들웨어는 IRIS 로그분석기와 함께 Docker 버전으로 배포될 수 있습니다. 본 챕터에서는 Docker 버전으로 배포되는 미들웨어에서 외부명령어를 사용하기 위한 방법을 wordcount 예제를 통해 설명합니다. 아래 예제는 Linux/Unix 환경을 기준으로 작성되었습니다.

2.2.1 Docker 버전 미들웨어 설치

먼저 배포 파일에서 Docker image를 등록하고 실행하는 방법에 대해 설명합니다. Docker 서비스를 사용할 수 있는 환경이어야 합니다.

일반적으로 IRIS 로그분석기 미들웨어의 Docker 버전 배포 파일은 "angora-dist-<version>-<release_date>.tar.gz" 형식으로 제공됩니다. tar.gz 압축을 해제하는 방법은 다음과 같습니다.

```
$ tar xvfz angora-dist-0.8.0-0.0.1-20171127.tar.gz
```

IRIS 로그분석기가 처음 설치되는 환경이라면, IRIS 로그분석기 공통 패키지가 필요합니다. 공통패키지의 Docker 버전 배포 파일은 "common-dist-<ID>-<release_date>.tar.gz" 와 같은 형식입니다. 다음과 같이 IRIS 로그분석기 공통 패키지의 압축을 해제합니다.

```
$ tar xvfz common-dist-46284d2-20171127.tar.gz
```

압축을 해제하면 다음과 같은 구조의 service 디렉토리가 생성됩니다.

```
service
├── bin
│   ├── all
│   │   ├── install-all.sh
│   │   ├── start-all.sh
│   │   └── stop-all.sh
│   ├── angora.sh
│   └── angora-setup-apply.sh
├── common
│   ├── install
│   │   └── install.sh
│   ├── bin
│   │   ├── bin.sh
│   │   └── get-network-conf.sh
```

```
├── conf
│   └── angora
│       ├── docker.conf
│       └── angora.conf.template
├── conf-template
│   └── network.conf
├── images
│   └── angora.tar.gz
├── install
│   ├── common.sh
│   └── angora.sh
└── lib
    └── angora
```

진하게 표시한 항목은 IRIS 로그분석기 미들웨어에서 사용하는 디렉토리와 파일입니다. 아래는 각 항목의 설명입니다.

bin/angora.sh
Docker 버전의 IRIS 로그분석기 미들웨어를 실행하거나 종료할 때 사용하는 스크립트입니다.

bin/angora-setup-apply.sh
각종 설정을 적용할 때 사용하는 스크립트 입니다. angora-docker.sh 와 함께 실행됩니다.

conf/angora.conf.template
IRIS 로그분석기 미들웨어 설정파일의 템플릿입니다.

conf/docker.conf
IRIS 로그분석기 미들웨어의 Docker 환경 설정파일입니다.

conf-template/network.conf
IRIS 로그분석기의 네트워크관련 설정파일입니다. 미들웨어와 관련된 항목만 사용합니다.

images/angora.tar.gz
IRIS 로그분석기 미들웨어의 Docker image 파일입니다.

install/angora.sh
Docker image 파일을 설치할 때 사용하는 스크립트 입니다.

lib/angora
기타 라이브러리를 위한 디렉토리입니다.

2.2.2 Docker 버전 미들웨어 실행

Docker 버전 IRIS 로그분석기 미들웨어를 실행하기 위해서는 Docker 환경에 대해 설정이 필요합니다. Docker 버전 미들웨어는 service/conf 디렉토리를 기준으로 다음 3가지 conf 파일의 설정을 참조합니다.

```
service
└── conf
    ├── network.conf
    └── angora
        ├── docker.conf
        └── angora.conf
```

아래 명령을 통해 service 디렉토리로 이동하여 각 설정 파일을 복사합니다.

```
$ cd service && cp conf/angora/angora.conf.template conf/angora/angora.conf && cp conf-template/network.conf conf/
```

첫 번째로 network.conf에서 다음 항목을 수정해 네트워크 관련 설정을 합니다. 본 예제에서는 angora 항목과 spark 항목을 로컬호스트를 기준으로 설정합니다.

```
$ vi conf/network.conf
...
spark.master.ip=localhost
spark.master.port=7077
...
angora.ip=localhost
angora.port=6036
angora.spark.driver.ip=localhost
angora.spark.driver.port=46036
angora.spark.driver.ui.port=46038
```

두 번째로 Docker image를 Docker 서비스에 등록합니다. 등록이 끝나면 Docker 명령을 이용해 이미지가 제대로 적용되었는지 확인합니다. 이때, 같은 버전의 오래된 미들웨어 이미지가 삭제되는 것에 주의해야 합니다.

```
$ sh install/angora.sh
...
INFO:Succeed to load angora docker image.
```

```
there exists already config file. if you want to copy the template, you may use
option -c.
```

아래 Docker 명령을 통해 이미지가 등록된 것을 확인할 수 있습니다.

```
$ docker images | grep angora
mobigen.com/angora    0.8.0-0.0.1      3632b798e0fc      3 days ago       915MB
mobigen.com/angora    latest           3632b798e0fc      3 days ago       915MB
```

세 번째로 IRIS 로그분석기 미들웨어를 실행합니다.

```
$ bin/angora.sh start
...
[INFO] angora starting done
```

IRIS 로그분석기 미들웨어가 실행되면, service/logs/angora/angora.log에 로그를 기록합니다. 다음과 같이 로그를 확인하여 미들웨어가 정상적으로 실행되는지 확인합니다.

```
$ tail logs/angora/angora.log
...
2017-10-31 17:41:00,979 - werkzeug - INFO -  * Running on http://0.0.0.0:6036/
(Press CTRL+C to quit)
```

미들웨어를 종료하고자 할 때는 다음과 같이 중지 명령을 실행합니다.

```
$ bin/angora.sh stop
...
[INFO] angora stopping done
```

3 외부 명령어 개발

본 챕터에서는 간단한 예제를 통해 외부 명령어를 구현하는 방법에 대해 설명합니다.

3.1 외부 명령어 구현

입력 받은 데이터의 단어 개수를 세는 외부 명령어 wordcount.py를 구현해 보겠습니다. 소스 코드에 대한 내용을 다음과 같이 설명합니다.

예제의 전체 소스 코드는 다음과 같습니다. 아래와 같이 wordcount.py 파일을 생성합니다.

```
$ echo -e "from angora.cmds.base import SimpleCommand


class SimpleWordCount(SimpleCommand):
    def execute(self, sqlCtx, df=None, parsed_args=None, *args, **kwargs):
        # input data
        data = df.collect()

        # word count
        wordcount = {}
        for _row in data:
            for _item in _row :
                for word in _item.split(" ") :
                    count = wordcount.get(word, 0)
                    wordcount[word] = count + 1

        # output data make
        ret = []
        for k, v in wordcount.items():
            ret.append({'word': k, 'count': v})

        # output data
        return sqlCtx.createDataFrame(ret)
" > wordcount.py
```

wordcount.py 소스 코드에 대한 설명은 다음과 같습니다.

먼저 외부 명령어 구현에 필요한 SimpleCommand를 import 합니다.

```
from angora.cmds.base import SimpleCommand
```

다음으로 SimpleCommand를 상속받는 SimpleWordCount 클래스를 정의합니다.

```
class SimpleWordCount(SimpleCommand):
```

다음으로 execute 메소드를 구현합니다.

```
    def execute(self, sqlCtx, df=None, parsed_args=None, *args, **kwargs):
        # input data
        data = df.collect()

        # word count
        wordcount = {}
        for _row in data:
            for _item in _row :
                for word in _item.split(" ") :
                    count = wordcount.get(word, 0)
                    wordcount[word] = count + 1

        # output data make
        ret = []
        for k, v in wordcount.items():
            ret.append({'word': k, 'count': v})

        # output data
        return sqlCtx.createDataFrame(ret)
```

def execute(self, sqlCtx, df=None, parsed_args=None, *args, **kwargs):
execute 메소드는 입력 받은 데이터와 파라미터를 가지고 동작을 수행하는 함수입니다.
execute 메소드의 입력과 출력은 항상 Spark의 DataFrame으로 고정되어있습니다.

메소드의 각 파라미터는 다음과 같습니다.

- **sqlCtx** Spark Context, 활성화 된 Spark Cilent 객체를 의미합니다.
- **df** 입력 데이터인 Spark DataFrame을 의미합니다.
- **parsed_args** list형태의 파라미터를 받을 때 사용하는 옵션입니다. 본 예제에서는 사용하지 않습니다.
- ***args, **kwargs** 검색 명령어 쿼리의 각 단계에 공통적으로 적용되는 positional/keyword argument 입니다. 각 명령어에 사용되는 argument가 다르게 사용되므로 반드시 필요한 옵션입니다.

data = df.collect()
입력 받은 Spark DataFrame형태의 데이터를 로컬로 가져와 list형태로 변환합니다.

- 변환결과: list[Row, Row, ...]

```
wordcount = {}
```
단어와 빈도수를 담기 위한 dict타입 변수를 선언합니다.

```
for _row in data:
    for _item in _row :
        for word in _item.split(" ") :
            count = wordcount.get(word, 0)
            wordcount[word] = count + 1
```
for loop 문을 통해 단어 별 빈도수를 계산합니다. 먼저 데이터에서 row를 하나씩 읽습니다. 해당 row의 item을 하나씩 꺼내 읽은 다음, item에 담긴 문자열 데이터를 하나씩 꺼내 공백을 기준으로 word를 구분합니다. 각 word는 wordcount hash의 key로 저장하고 빈도수를 value에 누적합니다.

- wordcount hash: dict{word: 빈도수, word: 빈도수, ... }

```
ret = []
```
리턴 결과를 담기 위한 list타입 변수를 선언합니다. DataFrame을 생성하는데 사용합니다.

```
for k, v in wordcount.items():
    ret.append({'word': k, 'count': v})
```
빈도수 측정이 끝난 wordcount hash에서 key인 k는 'word'로 분류하고, value인 v는 'count'로 분류해 각각 {'word': k, 'count': v} 형태의 dict로 만들어 ret변수에 append 합니다.

- ret 변수: list[dict{'word': k, 'count': v}, ...]

```
return sqlCtx.createDataFrame(ret)
```
마지막으로 결과를 출력하기 위해 DataFrame으로 만들어 리턴합니다.

- 리턴 DataFrame:

```
  count  word
-------  -------
    ...  ...
```

3.2 분산처리를 위한 외부 명령어 구현

앞서 **3.1 외부 명령어** 구현 의 예제와 동일한 기능을 하는 외부 명령어 wordcount2.py를 Spark 함수를 활용해 구현해 보도록 하겠습니다. 아래와 같이 wordcount2.py 파일을 생성합니다.

```
$ echo -e "from angora.cmds.base import SimpleCommand


class SimpleWordCount2(SimpleCommand):
    def execute(self, sqlCtx, df=None, parsed_args=None, *args, **kwargs):
        return df.rdd.flatMap(lambda row: row) \\
            .flatMap(lambda item: item.split(" ")) \\
            .map(lambda word: (word, 1)) \\
            .reduceByKey(lambda x, y: x + y) \\
            .toDF(['word', 'count'])" > wordcount2.py
```

execute 메소드의 각 단계는 다음과 같습니다.

df.rdd
먼저 입력 받은 데이터(Spark DataFrame)를 Spark RDD로 변환합니다. RDD는 Spark에서 사용하는 특수한 구조의 데이터 객체로서 DataFrame과 상호 변환이 가능합니다.

.flatMap(lambda row: row)
rdd의 flatMap함수를 통해 각 row의 하나의 요소로 갖는 list 구조로 변환합니다. flatMap 함수는 이중 list 구조로 되어있는 각 row를 1줄의 list 구조로 변환하는 Spark RDD 함수입니다.

- 변환결과: list[row, row, ...]

.flatMap(lambda item: item.split(" "))
변환한 list의 각 요소를 space를 기준으로 split(잘라내기)하여 word를 분리합니다. flatMap 함수를 통해 이중 list 구조로 되어있는 각 item 내 단어를 1줄의 list 구조로 변환합니다.

- split 결과: list[item[단어, 단어, ...], item[단어, 단어, ...], ...]
- flatMap 결과: list[단어, 단어, ...]

.map(lambda word: (word, 1))
빈도수 계산을 위해 rdd의 map함수로 list의 요소를 (단어, 1) key-value-pair 형태로 변환합니다. 이때 단어는 key, 출현빈도 1은 value가 됩니다.

- 변환결과: list[(단어, 1), (단어, 1), ...]

.reduceByKey(lambda x, y: x + y)
reduceByKey 함수를 통해 각 단어 별 출현 빈도를 합산합니다.
- 변환결과: list[(단어, 2), (단어, 3), ...]

.toDF(['word', 'count'])
마지막으로 결과를 리턴하기 위해 데이터프레임 형태로 변환합니다.
- 리턴 DataFrame:

```
word       count
-------  -------
...          ...
```

앞서 작성한 **3.1** 의 wordcount 예제와 wordcount2 예제는 동일한 동작을 수행하는 명령어 입니다. 그러나 wordcount 예제는 python 으로만 작성되어 분산 환경을 제대로 사용할 수 없습니다. 분산 환경의 IRIS 로그분석기 미들웨어를 통해 적절한 분산처리를 해야 하는 경우라면 본 챕터의 wordcount2 예제와 같이 Spark 함수를 사용해 명령어를 제작해야 합니다.

3.3 Standalone 버전 미들웨어의 외부명령어 등록

구현한 명령어를 사용하기 위해서는 angora.conf 설정 파일을 수정해 명령어를 등록하는 과정이 필요합니다.

external_command_path는 외부 명령어들의 위치를 나타내는 경로입니다. 아래와 같이 설정하면 $ANGORA_HOME/external 경로를 참조하여 외부 명령어를 찾게 됩니다.

```
$ vi $ANGORA_HOME/conf/angora.conf
...
external_command_path: lib/external
...
```

절대경로인 경우 해당 위치를 그대로 사용하고, 상대경로인 경우(시작 경로가 / 이 아닌 경우) $ANGORA_HOME을 기준으로 명령어의 위치를 설정합니다.

```
angora (== $ANGORA_HOME)
└── lib
    └── external
        ├── bar
        │   └── __init__.py
        ├── foo.py
        ├── wordcount.py
        ├── wordcount2.py
        └── foobar.zip
```

경로 설정이 끝나면 $ANGORA_HOME 디렉토리 아래에 lib/external 디렉토리를 생성하고, **3.1**의 wordcount 예제와 **3.2**의 wordcount2 예제를 해당 디렉토리에 복사합니다.

```
$ mkdir -p $ANGORA_HOME/lib/external &&
cp wordcount.py $ANGORA_HOME/lib/external &&
cp wordcount2.py $ANGORA_HOME/lib/external
```

아래와 같이 import를 통해 해당 명령어가 올바르게 인식되는지 확인 할 수 있습니다.

```
$ cd $ANGORA_HOME/lib/external && python
>>> import wordcount
>>> wordcount.SimpleWordCount
```

```
<class 'wordcount.SimpleWordCount'>
>>> import wordcount2
>>> wordcount.SimpleWordCount2
<class 'wordcount.SimpleWordCount2'>
```

미들웨어를 재시작하면 수정 된 설정이 적용됩니다.

3.4 Docker 버전 미들웨어의 외부명령어 등록

Docker 버전의 IRIS 로그분석기 미들웨어는 호스트 서버의 service 디렉토리를 공유하여 사용합니다. service 디렉토리는 Docker 버전 미들웨어의 /mount 디렉토리와 연결되어 있습니다.

외부 명령어를 적용하기 위해서는 외부 명령어 파일을 호스트 서버의 service 디렉토리에 생성하고, 외부 명령어 경로를 /mount를 기준으로 설정해야 합니다.

먼저, **2.2.1**의 service 디렉토리를 기준으로 lib/external 디렉토리를 생성하고,

```
$ mkdir -p lib/external
```

챕터 **3.1**의 wordcount 예제와 챕터 **3.2**의 wordcount2 예제를 해당 디렉토리에 생성합니다.

- wordcount.py 예제

```
$ echo -e "from angora.cmds.base import SimpleCommand


class SimpleWordCount(SimpleCommand):
    def execute(self, sqlCtx, df=None, parsed_args=None, *args, **kwargs):
        # input data
        data = df.collect()

        # word count
        wordcount = {}
        for _row in data:
            for _item in _row :
                for word in _item.split(" ") :
                    count = wordcount.get(word, 0)
                    wordcount[word] = count + 1

        # output data make
        ret = []
        for k, v in wordcount.items():
            ret.append({'word': k, 'count': v})

        # output data
        return sqlCtx.createDataFrame(ret)
" >> lib/external/wordcount.py
```

- wordcount2.py 예제

```
$ echo -e "from angora.cmds.base import SimpleCommand


class SimpleWordCount2(SimpleCommand):
    def execute(self, sqlCtx, df=None, parsed_args=None, *args, **kwargs):
        return df.rdd.flatMap(lambda row: row)
            .flatMap(lambda item: item.split(" "))
            .map(lambda word: (word, 1))
            .reduceByKey(lambda x, y: x + y)
            .toDF(['word', 'count'])" >> wordcount2.py
```

설정파일의 external_command_path 항목을 수정해 외부 명령어 경로를 설정합니다.

```
$ vi conf/angora/angora.conf
...
external_command_path: /mount/lib/external
```

미들웨어를 재시작하면 수정 된 설정이 적용됩니다.

3.5 미들웨어 외부 명령어 실행

본 챕터에서는 생성한 외부 명령어를 실행하고 결과를 확인하는 방법을 설명합니다. 3.3 또는 3.4 챕터를 참조하여 외부 명령어를 등록합니다. 변경된 설정을 적용하기 위해서는 미들웨어를 다시 시작 해야 합니다.

- Standalone 버전

```
$ angora engine stop && angora engine start
```

- Docker 버전 (service 경로 기준)

```
$ bin/angora.sh stop && bin/angora.sh start
```

IRIS 로그분석기 미들웨어는 RESTful 연결을 위한 API를 제공합니다. 아래는 curl 명령을 사용해 미들웨어의 RESTful API를 사용하는 예 입니다.

먼저, wordcount 예제에 사용할 데이터파일인 tmp.csv를 현재 경로에 생성합니다.

- Standalone 버전

```
$ cd $ANGORA_HOME && echo -e "fieldA,fieldB,fieldC
apple,house,company
house,fruit,car
apple company house,fruit,car
apple pencil,house fruit car,apple company" > tmp.csv
```

- Docker 버전 (service 경로 기준)

```
$ echo -e "fieldA,fieldB,fieldC
apple,house,company
house,fruit,car
apple company house,fruit,car
apple pencil,house fruit car,apple company" > tmp.csv
```

IRIS 로그분석기 미들웨어의 모든 RESTful 요청에는 사용자 계정에 대한 토큰정보가 필요합니다. 외부 명령어 요청을 보내기에 앞서 사용자 계정에 대한 토큰을 발급받아야 합니다.

```
$ curl -X POST "http://localhost:6036/angora/auth" \
-H "Content-Type: application/json" \
-d '{"id" : "test", "password" : "test"}' | grep token
```

다음과 같이 토근정보를 얻을 수 있습니다.

```
"token": "dGVzdC1hOWMwMjkwOC1mZTBmLTQ3ZmItYjdkMC01NWZkYjZhOWIyM2Q="
```

발급 받은 토큰정보를 Authorization: Angora “<token>” 형태로 Request Header에 추가하고, 다시 curl을 통해 외부 명령어 사용을 요청합니다. 만약 사용자 토큰이 만료되어 사용할 수 없게 된 경우 토큰을 재발급 받아야 합니다.

외부 명령어는 미들웨어의 내부 명령어인 angora search 명령을 통해 결과를 확인해 볼 수 있습니다. angora search는 문자열을 명령어 쿼리로 받아들이는 명령어입니다. curl의 -d 옵션으로 Request Body에 아래와 같이 json 형태로 명령어를 전달합니다.

- Standalone 버전

```
$ curl -X GET "http://localhost:6036/angora/search" \
-H "Authorization: Angora dGVzdC1hOWMwMjkwOC1mZTBmLTQ3ZmItYjdkMC01NWZkYjZhOWIyM2Q=" \
-H "Content-Type: application/json" \
-d '{"q" : "hdfs csv read file://'$ANGORA_HOME'/tmp.csv | search * | wordcount"}'
```

- Docker 버전 (service 경로 기준)

```
$ curl -X GET "http://localhost:6036/angora/search" \
-H "Authorization: Angora dGVzdC1hOWMwMjkwOC1mZTBmLTQ3ZmItYjdkMC01NWZkYjZhOWIyM2Q=" \
-H "Content-Type: application/json" \
-d '{"q" : "hdfs csv read file:///mount/tmp.csv | search * | wordcount"}'
```

hdfs csv read <파일경로>
<파일경로>에 해당하는 파일의 내용을 데이터소스로 읽어옵니다.

search *
모든 데이터를 읽어 옵니다.

wordcount
본 매뉴얼에서 구현한 외부 명령어 입니다. 전달 받은 데이터를 통해 wordcount를 수행합니다.

아래는 위에서 설명한 두 번의 curl 명령을 하나로 합친 스크립트 입니다.

- Standalone 버전

```
$ curl -X GET "http://localhost:6036/angora/search" \
-H "Authorization: Angora $(curl -s -X POST "http://localhost:6036/angora/auth" \
-H "Content-Type: application/json" \
-d '{"id" : "test", "password" : "test"}' | \
grep token | awk -F "[\"]" '{printf $4"\n"}')" \
-H "Content-Type: application/json" \
-d '{"q" : "hdfs csv read file://'$ANGORA_HOME'/tmp.csv | search * | wordcount"}'
```

- Docker 버전 (service 경로 기준)

```
$ curl -X GET "http://localhost:6036/angora/search" \
-H "Authorization: Angora \
$(curl -s -X POST "http://localhost:6036/angora/auth" \
-H "Content-Type: application/json" \
-d '{"id" : "test", "password" : "test"}' | \
grep token | awk -F "[\"]" '{printf $4"\n"}')" \
-H "Content-Type: application/json" \
-d '{"q" : "hdfs csv read file:///mount/tmp.csv | search * | wordcount"}'
```

다음과 같이 json 형식으로 결과를 얻을 수 있습니다.

```
{
  "fields": [
    {
      "type": "TEXT",
      "name": "word"
    },
    {
      "type": "LONG",
      "name": "count"
    }
  ],
  "results": [
    [
      "pencil",
      1
    ],
    [
      "apple",
      4
    ],
    [
      "car",
```

```
      3
    ],
    [
      "company",
      3
    ],
    [
      "fruit",
      3
    ],
    [
      "house",
      4
    ]
  ]
}
```

3.6 유닛 테스트

unitest 라이브러리를 이용한 유닛 테스트 (test_wordcount.py) 예제 입니다. 본 매뉴얼에서는 execute 메소드를 테스트 합니다.

```
import unittest

from pyspark import SparkContext, SQLContext

from wordcount import SimpleWordCount


class WordCountTests(unittest.TestCase):
    sqlCtx = None

    @classmethod
    def setUpClass(cls):
        sc = SparkContext('local')
        WordCountTests.sqlCtx = SQLContext(sc)
        WordCountTests.version = str(WordCountTests.sqlCtx._sc.version)

    def setUp(self):
        data = [["apple company"], ["apple money"], ["money pencil car"]]
        self.test_df = WordCountTests.sqlCtx.createDataFrame(data)

    def test_parse(self):
        command = SimpleWordCount()
        self.assertEqual("Test wordcount", command.parse("Test wordcount"))

    def test_execute(self):
        command = SimpleWordCount()
        parsed_args = command.parse("")
        df = command.execute(WordCountTests.sqlCtx, self.test_df, parsed_args)
        df = df.filter(df['word'] == 'money').select(df['count'])
        self.assertEqual(2, df.collect()[0][0])


if __name__ == "__main__":
    unittest.main()
```

def setUpClass(cls):

Spark에서 테스트를 실행시키기 위해, SparkContext 및 SQLContext를 초기화 합니다.

def setUp(self):

해당 단계에서는 SQLContext.range API를 통해 텍스트 데이터를 갖고 있는 Spark DataFrame을 생성합니다. 해당 함수는 매 테스트마다 호출되며, 아래 테스트에서 사용하도록 할 것 입니다. 테스트의 사용되는 텍스트 데이터는 다음과 같습니다.

- 데이터: [["apple company"], ["apple money"], ["money pencil car"]]

def test_parse(self):

해당 메소드 만들어진 wordcount 명령어의 인자를 파싱하는 parse 메서드 부분을 테스트 하고 있습니다. 해당 명령어는 parse 메서드를 구현하지 않았기 때문에 입력 받은 'Test wordcount' 문자열을 그대로 리턴하는 것을 확인하고 있습니다.

def test_execute(self):

해당 메소드 만들어진 wordcount 명령어의 인자 피싱 및 실행 부분을 테스트 하고 있습니다. 전달 받은 테스트 데이터의 wordcount 수행 후 'money' 단어의 빈도수가 2인 것을 확인하고 있습니다.

작성된 유닛 테스트는 아래와 같이 테스트를 실행합니다.

```
$ python -m unittest test_wordcount
```

실행 하게 되면, 아래와 같은 결과가 콘솔에 출력되게 됩니다.

```
..
----------------------------------------------------------------------
Ran 2 tests in 10.402s

OK
```

4 기본 API 상세 소개

4.1 SimpleCommand 클래스

아래는 외부 명령어 작성을 위해 상속받아야 하는 angora.cmds.SimpleCommand의 구현 입니다.

```
class SimpleCommand(BaseCommand):
    def parse(self, raw_args, *args, **kwargs):
        """
        Parse given raw arguments.
        Parameters
        ----------
        raw_args : str
            List of arguments. This is produced by ``shlex.shlex()``. For example,
            a query string, ``search aa bb`` passes ``'aa bb'`` as
            ``raw_args``.
        *args, *kwargs : extra arguments and keyword arguments, optional
        Returns
        -------
        """
        args = # parse `raw_args`.
        return parsed_args

    def execute(self, sqlCtx, df=None, parsed_args=None, *args, **kwargs):
        """
        Executes ``transformation`` in Spark.
        Parameters
        ----------
        sqlCtx : pyspark.sql.SQLContext
            Spark ``SQLContext`` object.
        df : pyspark.sql.DataFrame or None, optional
            Spark ``DataFrame`` object.
        parsed_args : str or None, etc... optional
            Arguments parsed from ``parse()``.
        *args, *kwargs : extra arguments and keyword arguments, optional
        Returns
        -------
        pyspark.sql.DataFrame
        """
        return df
```

외부 명령어를 구현하기 위해서는 angora.cmds.SimpleCommand를 상속 받아 외부 명령어를 정의 합니다. angora.cmds.SimpleCommand를 상속받아 사용하기 위해서는 아래 두 가지 메소드를 재정의 해야 합니다.

```
def parse(self, raw_args, *args, **kwargs):
def execute(self, sqlCtx, df=None, parsed_args=None, *args, **kwargs):
```

아래는 해당 두 가지 메소드에 대한 설명 입니다:

def parse(self, raw_args, *args, **kwargs):

raw_args: 해당 외부 명령어에 들어오는 raw string을 의미합니다.

***args, **kwargs**: 추가적인 positional/keyword argument를 의미합니다.

parse는 해당 외부 명령어에 올 수 있는 파라미터들을 파싱하는 함수입니다. raw_args로 입력 받은 문자열이 들어오게 되는데, 해당 문자열을 execute 함수에서 사용 할 형태로 파싱을 해서 넘겨줘야 합니다. 예를 들어, count 라는 명령어로 필드를 여러 개 받는 parse 함수를 구현한다면, 아래의 경우,

```
$ search * | count fieldA fieldB
```

count 명령어의 parse에서 raw_args로 오는 값은 "fieldA fieldB" 문자열입니다.
해당 메소드에서 return 된 객체가 execute 메소드의 parsed_args인자로 전달 됩니다.
예를 들어, parse에서 argparse.Namespace 타입으로 return 하면, execute에서 parsed_args 인자의 타입은 argparse.Namespace가 됩니다.

def execute(self, sqlCtx, df=None, parsed_args=None, *args, **kwargs):

sqlCtx: SQLContext, 현재 Spark client를 의미합니다.

df: 해당 외부 명령어의 전 단에서 처리된 Spark DataFrame을 의미합니다.

parsed_args: 외부 명령어의 parse 메소드에서 처리된 명령어 인자를 의미합니다. 명령어 쿼리에 이어지는 문자열을 parse에서 정한 형태의 파라미터로 받을 때 사용하는 옵션입니다.

***args, **kwargs** 검색 명령어 쿼리의 각 단계에 공통적으로 적용되는 positional/keyword argument 입니다. 각 명령어에 사용되는 argument가 다르게 사용되므로 반드시 필요한 옵션입니다.

execute 함수의 경우에는, 항상 return을 Spark의 DataFrame으로 반환해야 하는 제약조건이 있습니다.