Artículo
Jose-Tomas Salvador · 7 jul, 2021 Lectura de 8 min

Conexión con JDBC desde Python a la base de datos de IRIS - segunda nota rápida

Palabras clave PyODBC, unixODBC, IRIS, IntegratedML, Jupyter Notebook, Python 3

Propósito

Hace unos meses traté el tema de la "conexión con JDBC desde Python a la base de datos de IRIS", y desde entonces utilicé ese artículo con más frecuencia que mi propia nota oculta en mi PC. Por eso, traigo aquí otra nota de 5 minutos sobre cómo hacer una "conexión con JDBC desde Python a la base de datos de IRIS". ODBC y PyODBC parecen bastante fáciles de configurar en un cliente de Windows, sin embargo, siempre me atasco un poco en la configuración de un cliente unixODBC y PyODBC en un servidor de estilo Linux/Unix. ¿Existe un enfoque tan sencillo y consistente como se supone que debe ser para hacer que el trabajo de instalación de PyODBC/unixODBC funcione en un cliente linux estándar sin ninguna instalación de IRIS, contra un servidor IRIS remoto?

Alcance

Hace poco tuve que dedicar un tiempo a hacer funcionar desde cero una demo de PyODBC dentro de Jupyter Notebook en un entorno Docker en Linux. De ahí me surgió la idea de realizar esta nota detallada, para futuras consultas.

En el alcance:

En esta nota vamos a tratar los siguientes componentes:

- **PyODBC** sobre **unixODBC**
- El servidor de **Jupyter Notebook** con Tensorflow 2.2 y Python 3
- Servidor **IRIS2020.3** CE con IntegratedML, incluyendo datos de prueba de ejemplo.
- dentro de este entorno
- El motor Docker con **Docker-compose** sobre AWS **Ubuntu** 16.04
- Docker Desktop para **MacOS**, y la **Docker Toolbos for Windows** 10 también están probadas

Fuera del alcance:

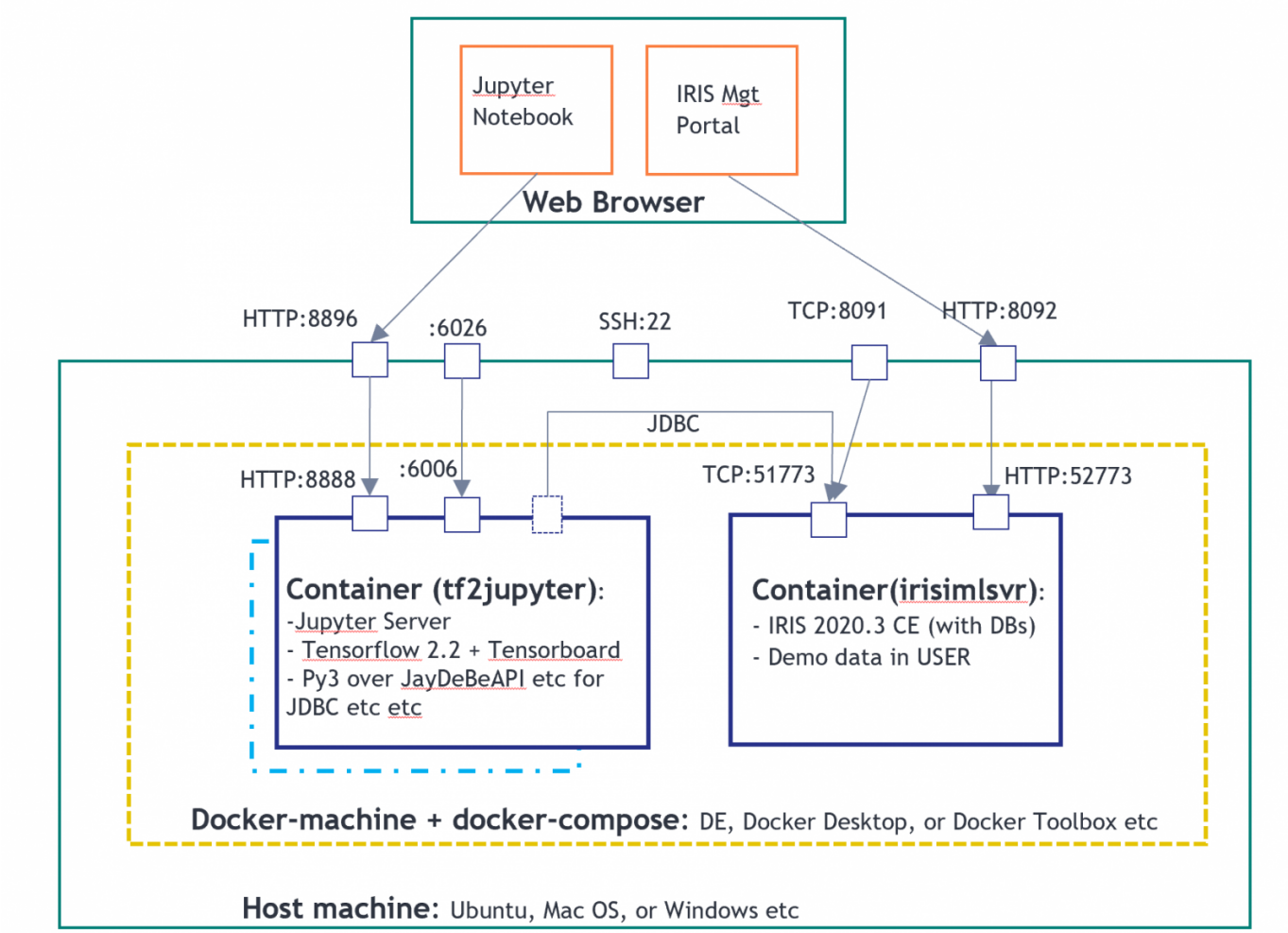
De nuevo, los aspectos no funcionales no se evalúan en este entorno de demo. Son importantes y pueden tratarse por separado, como:

- Seguridad y auditoría de extremo a extremo.
- Rendimiento y escalabilidad
- Licencias y soporte, etc.

Entorno

Se puede usar cualquier imagen en Docker para Linux en las configuraciones y los pasos de prueba que se indican a continuación, pero una forma sencilla de configurar un entorno de este tipo en 5 minutos es:

1. **Clonar** en Git esta plantilla de demostración
2. Ejecutar "**docker-compose up -d**" en el directorio clonado que contiene el archivo docker-compose.yml. Simplemente creará un entorno de demostración, como se muestra en la siguiente imagen, de 2 contenedores. Uno para el servidor de Jupyter Notebook como cliente PyODBC, y otro para un servidor IRIS2020.3 CE.



En el entorno anterior, el tf2jupyter solo contiene una configuración de cliente "Python sobre JDBC", todavía no contiene ninguna configuración de cliente ODBC o PyODBC. Así que ejecutaremos los siguientes pasos para configurarlo, directamente desde Jupyter Notebook para que sea autoexplicativo.

Pasos

Ejecuté las siguientes configuraciones y pruebas en un servidor AWS Ubuntu 16.04. Mi colega @Thomas Dyar los ejecutó en MacOS. También lo probé brevemente en Docker Toolbox for Windows. Pero haznos saber si encuentras algún problema. Los siguientes pasos se pueden automatizar de forma sencilla en tu Dockerfile. Lo grabé manualmente aquí en caso de que se me olvide cómo se hacía después de unos meses.

1. Documentación oficial:

Soporte de ODBC para IRIS
Cómo definir la fuente de datos ODBC en Unix
Soporte PyODBC para IRIS

2. Conectarse al servidor Jupyter

Utilicé el SSH tunneling de Putty en forma local en el puerto 22 de AWS Ubuntu remoto, después mapeé al puerto 8896 como en la imagen anterior. (Por ejemplo, en el entorno local de Docker también se puede utilizar http

directamente en la IP:8896 del equipo Docker).

3. Ejecutar la instalación de ODBC desde Jupyter Notebook

Ejecuta la siguiente línea de comando directamente desde una celda en Jupyter:

```
!apt-get update
!apt-get install gcc
!apt-get install -y tdsodbc unixodbc-dev
!apt install unixodbc-bin -y
!apt-get clean -y
```

Instalará el compilador gcc (incluyendo g++), FreeTDS, unixODBC y unixodbc-dev , que son necesarios para volver a compilar el *driver* PyODBC en el siguiente paso. Este paso no es necesario en un servidor nativo de Windows o PC para la instalación de PyODBC.

4. Ejecutar la instalación de PyODBC desde Jupyter

```
!pip install pyodbc

Collecting pyodbc
  Downloading pyodbc-4.0.30.tar.gz (266 kB)
     |????????????????????????????????| 266 kB 11.3 MB/s eta 0:00:01
Building wheels for collected packages: pyodbc
  Building wheel for pyodbc (setup.py) ... done
  Created wheel for pyodbc: filename=pyodbc-4.0.30-cp36-cp36m-linux_x86_64.whl size=2
73453 sha256=b794c35f41e440441f2e79a95fead36d3aebfa74c0832a92647bb90c934688b3
  Stored in directory: /root/.cache/pip/wheels/e3/3f/16/e11367542166d4f8a252c031ac3a4
163d3b901b251ec71e905
Successfully built pyodbc
Installing collected packages: pyodbc
Successfully installed pyodbc-4.0.30
```

Lo anterior es la instalación mínima de PIP para esta demostración de Docker. En la documentación oficial, para la "Instalación de MacOS X", se proporcionan instrucciones para una instalación de PIP más detallada.

5. Volver a configurar los archivos y enlaces ODBC INI en Linux:

Ejecuta las siguientes líneas de comando para volver a crear los enlaces **odbcinst.ini** y **odbc.ini**
!rm /etc/odbcinst.ini !rm /etc/odbc.ini
!ln -s /tf/odbcinst.ini /etc/odbcinst.ini !ln -s /tf/odbc.ini /etc/odbc.ini

Nota: La razón de realizar lo anterior es que **los pasos 3 y 4 normalmente crearían 2 archivos ODBC en blanco (por lo tanto, no válidos) en el directorio \etc.** A diferencia de la instalación de Windows, estos archivos ini en blanco causan problemas, por lo tanto, debemos eliminarlos y luego simplemente volver a crear un enlace a los archivos ini reales proporcionados en un volumen Docker mapeado: /tf/odbcinst.ini, y /tf/odbc.ini Revisemos estos 2 archivos ini - en este caso, son la forma más simple para las configuraciones ODBC de Linux:
!cat /tf/odbcinst.ini

```
[InterSystems ODBC35]
UsageCount=1
Driver=/tf/libirisodbcu35.so
```

```
Setup=/tf/libirisodbcu35.so
SQLLevel=1
FileUsage=0
DriverODBCVer=02.10
ConnectFunctions=YYN
APILevel=1
DEBUG=1
CPTimeout=<not Pooled>
```

!cat /tf/odbc.ini

```
[IRIS PyODBC Demo]
Driver=InterSystems ODBC35
Protocol=TCP
Host=irisimlsvr
Port=51773
Namespace=USER
UID=SUPERUSER
Password=SYS
Description=Sample namespace
Query Timeout=0
Static Cursors=0
```

Los archivos anteriores están pre-configurados y se proporcionan en la unidad mapeada. Esto se refiere al driver **libirisodbcu35.so,** que también se puede conseguir en la instancia del contenedor del servidor IRIS (en su directorio {iris-installation}/bin). Por lo tanto, para que la instalación ODBC anterior funcione, **deben existir estos 3 archivos** en la unidad mapeada (o en cualquier unidad de Linux) **con los permisos de los archivos adecuados**:

libirisodbcu35.so
odbcinst.ini
odbc.ini

6. Verificar la instalación de PyODBC

!odbcinst -j

```
unixODBC 2.3.4
DRIVERS............: /etc/odbcinst.ini
SYSTEM DATA SOURCES: /etc/odbc.ini
FILE DATA SOURCES..: /etc/ODBCDataSources
USER DATA SOURCES..: /root/.odbc.ini
SQLULEN Size.......: 8
SQLLEN Size........: 8
SQLSETPOSIROW Size.: 8
```

import pyodbc print(pyodbc.drivers())

```
['InterSystems ODBC35']
```

Por ahora, las salidas anteriores indicarán que el *driver* ODBC tiene los enlaces válidos. Deberíamos poder ejecutar alguna prueba de ODBC de Python en Jupyter Notebook.

7. Ejecutar la conexión ODBC de Python en las muestras de IRIS:

import pyodbc import time
1. Get an ODBC connection

```
#input("Hit any key to start")
dsn = 'IRIS PyODBC Demo'
server = 'irisimlsvr' #IRIS server container or the docker machine's IP
port = '51773' # or 8091 if docker machine IP is used
database = 'USER'
username = 'SUPERUSER'
password = 'SYS'
#cnxn = pyodbc.connect('DSN='+dsn+';') # use the user DSN defined in odbc.ini, or use the connection string below
cnxn = pyodbc.connect('DRIVER={InterSystems ODBC35};SERVER='+server+';PORT='+port+';DATABASE='+database+';UID='+username+';PWD='+ password)
###ensure it reads strings correctly.
cnxn.setdecoding(pyodbc.SQL_CHAR, encoding='utf8')
cnxn.setdecoding(pyodbc.SQL_WCHAR, encoding='utf8')
cnxn.setencoding(encoding='utf8')
### 2. Get a cursor; start the timer
cursor = cnxn.cursor()
start= time.clock()
### 3. specify the training data, and give a model name
dataTable = 'DataMining.IrisDataset'
dataTablePredict = 'Result12'
dataColumn = 'Species'
dataColumnPredict = "PredictedSpecies"
modelName = "Flower12" #chose a name - must be unique in server end
### 4. Train and predict
#cursor.execute("CREATE MODEL %s PREDICTING (%s) FROM %s" % (modelName, dataColumn, dataTable))
#cursor.execute("TRAIN MODEL %s FROM %s" % (modelName, dataTable))
#cursor.execute("Create Table %s (%s VARCHAR(100), %s VARCHAR(100))" % (dataTablePredict, dataColumnPredict, dataColumn))
#cursor.execute("INSERT INTO %s SELECT TOP 20 PREDICT(%s) AS %s, %s FROM %s" % (dataTablePredict, modelName, dataColumnPredict, dataColumn, dataTable))
#cnxn.commit()
### 5. show the predict result
cursor.execute("SELECT * from %s ORDER BY ID" % dataTable) #or use dataTablePredict result by IntegratedML if you run step 4 above
row = cursor.fetchone()
while row:
  print(row)
  row = cursor.fetchone()
### 6. CLose and clean
cnxn.close()
end= time.clock()
print ("Total elapsed time: ")
print (end-start)
```

```
(1, 1.4, 0.2, 5.1, 3.5, 'Iris-setosa')
(2, 1.4, 0.2, 4.9, 3.0, 'Iris-setosa')
(3, 1.3, 0.2, 4.7, 3.2, 'Iris-setosa')
(4, 1.5, 0.2, 4.6, 3.1, 'Iris-setosa')
(5, 1.4, 0.2, 5.0, 3.6, 'Iris-setosa')
... ...
... ...
... ...
(146, 5.2, 2.3, 6.7, 3.0, 'Iris-virginica')
(147, 5.0, 1.9, 6.3, 2.5, 'Iris-virginica')
(148, 5.2, 2.0, 6.5, 3.0, 'Iris-virginica')
(149, 5.4, 2.3, 6.2, 3.4, 'Iris-virginica')
(150, 5.1, 1.8, 5.9, 3.0, 'Iris-virginica')
```

```
Total elapsed time:
0.023873000000000033
```

Un par de consejos:

1. **cnxn = pyodbc.connect()** - en el entorno Linux, la cadena de conexión expresada en esta llamada debe ser literalmente correcta y sin espacios.
2. Establece la codificación de la conexión correctamente con, por ejemplo, utf8. En este caso, el valor predeterminado no funcionaría para las cadenas.
3. **libirisodbcu35.so** - lo ideal es que este driver esté estrechamente alineado con la versión del servidor IRIS remoto.

Siguiente

Ahora tenemos un entorno Docker con un Jupyter notebook que incluye Python3 y Tensorflow2.2 (sin GPU) a través de una conexión PyODBC (así como JDBC) en un servidor IRIS remoto. Deben funcionar para todas las sintaxis SQL personalizadas, como las que son propiedad de IntegratedML en IRIS, así que ¿por qué no explorar un poco más en IntegratedML y ser creativo con su forma SQL de conducir ciclos de vida ML?
Además, me gustaría que pudiéramos volver a tratar o recapitular el enfoque más sencillo sobre IRIS nativo o incluso Magic SQL en el entorno Python, que se conectará con el servidor IRIS. Y, el extraordinario Python Gateway está disponible ahora, así que incluso podemos intentar invocar las aplicaciones y servicios externos de Python ML, directamente desde el servidor IRIS. Me gustaría que pudiéramos probar más sobre eso también.

Anexo

El archivo del bloc de notas anterior se revisará en este repositorio de Github, y también en Open Exchange.

#Analítica #Artificial Intelligence (AI) #Machine Learning (ML) #InterSystems IRIS