db-patch Documentation

Release 0.1-1

Miguel A. Martínez Pinedo

January 09, 2013

CONTENTS

Contents:

CHAPTER

ONE

USER GUIDE

1.1 How it works

db-patch load the patch; next, it connects with the database engine (see database and connection options) and recovers the list of all the databases. Apply the filter (see the command line options *-d*, *-i* and *-e*) to the databases list, and secuentially applies the patch in each filtered database. When the process ends, it closes all the connections and finish.

For determine the patch type, and to act consequently, *db-patch* uses the file extension; However, it is possible to select the patch to use through the **–patch-type** command line option. Currently *db-patch* support three patch types: *SQL*, *python* and *raw* patches; but it is possible to use custom patches through plugins.

The next table shows the current relation between file extensions and patch types.

Extension	Patch
sql	sql
py	python
raw	raw

1.2 Features

db-patch has two useful features: the dry-run mode and the ability of store the executed SQL sentences.

1.2.1 dry-run mode

Working in this mode (**-t** or **–dry-run** command line option) only the SQL sentences that does not modify the database are executed (and showed); therefore, this mode allow test the patch before of apply it into database.

1.2.2 Store SQL sentences

This feature allow store **exactly** the SQL statements introduced in the database. For manage this feature there are four command line options:

- **-c** or **–clean**. Avoid generate files with the SQL executed.
- **–gen-dir**. Define the directory where the files are stored (by default *sql_sentences*)
- **–gen-file**. Define the file names where the tag HOST will be replaced by the host, the tag NAME for the patch name and the tag TIME a timestamp (by default: HOST_NAME_TIME.sql)
- **-S**. Show the executed SQL in the standard output.

1.3 Patches

1.3.1 SQL patches

A **SQL patch** is a file with *.sql* extension that contains several SQL statements separated by "**;**".

db-patch recovers each SQL statement and execute it into each selected database.

By example the next SQL snippet contains three statements.

```
-- A very simple table creation
CREATE TABLE date (day INT, month CHAR(40), year CHAR(40));

-- Adding two rows
INSERT INTO date VALUES (0, '2012', '12');
INSERT INTO date VALUES (1, '2012', '12');
```

1.3.2 Python patches

A **Python patch** is a python file (with *.py* extension) that contains a class that inherit of *db-patch.patch.python.PythonPatch* and overwrites the *execute* method. Of this way it is possible generate SQL sentences dinamically.

The next snippet shows a very simple Python patch. This patch connect with the database and execute a create sentence, besides it shows messages before and after of apply the patch.

```
from dbpatch.patch.python import PythonPatch

class CreatePatch(PythonPatch):

    CREATE_TABLE = 'CREATE TABLE test (anInt INT, aString CHAR(40))'

    def pre_execute(self):
        print 'Before of execute the patch'

    def execute(self, db):
        connection = self.connect_to(db)
        statement  = self.create_statement(CreatePatch.CREATE_TABLE)
        status = statement.execute()

        return status

    def post_execute(self):
        print 'After of execute the patch'
```

Sometimes could be necessary connect with other database (of the same database engine) for recover information and generate new entries.

The next python patch introduce ten rows in the "test" database, and recover only a few of these rows. Next, it connects with the database "filtered" and introduce the recovered rows.

Note that the method **encode** is used for assure that the strings use the same charset, and the method **set_error** is used for store the error messages.

```
import time
from dbpatch.patch.python import PythonPatch
```

```
class InsertTwoDbsPatch(PythonPatch):

    DB_ORIG = "test"
    DB_DEST = "filtered"
    # The SQL sentences ends with ;
    # of this way the stored SQL can be used as a SQL patch
    INSERT_DATE  = "INSERT INTO date VALUES (?, '?', '?');"
    SELECT_MAYOR = "SELECT * FROM date WHERE day > 5;"

    def execute(self, db):
        if db != InsertTwoDbsPatch.DB_ORIG:
            return

        self.connect_to(InsertTwoDbsPatch.DB_ORIG)

        local = time.localtime()
        for i in range(10):
            values = (i+1, str(local[1]), str(local[0]))
            statement = self.create_statement(InsertTwoDbsPatch.INSERT_DATE,
                                              values)
            status = statement.execute()

            if not status:
                self.set_error('Unable insert [%s] in db [%s]' % \
                               (str(values), InsertTwoDbsPatch.DB_ORIG))
                return False


        select = self.create_statement(InsertTwoDbsPatch.SELECT_MAYOR)
        select.execute()
        rows = select.get_rows()

        self.connect_to(InsertTwoDbsPatch.DB_DEST)

        for row in rows:
            encoded = (row[0], self.encode(row[1]), self.encode(row[2]))
            statement = self.create_statement(InsertTwoDbsPatch.INSERT_DATE,
                                              encoded)
            status = statement.execute()

            if not status:
                self.set_error('Unable insert [%s] in db [%s]' % \
                               (str(values), InsertTwoDbsPatch.DB_DEST))
                return False

        return True
```

PythonPatch class

The **dbpatch.patch.python.PythonPatch** class contains the next methods.

create_statement (*marked_query*, *values=()*, *autocommit=True*)

Create one statement for carry out SQL sentences. This method **always must be used for execute SQL** in any database.

Parameters

- **marked_query** – The query with marks. Examples:

"INSERT INTO date VALUES (?, '?', '?')"

"SELECT * FROM date"

- **values** – The values for populate the marked_query.
- **autocommit** – Flag for enable/disable the autocommit behaviour.

Return type dbpatcher.statement.Statement

encode(*the_string*)
Encode the string passed by parameter taking into account the charset and the charset policy defined for the patch (through command line options).

Parameters **the_string** – The string to encode

Return type The encoded string

connect_to(*db_name*)
Connect with the database passed by parameter.

Parameters **db_name** – The database name to connect

Return type None

get_databases()
Return the list of databases taking into account the include/exclude options

Return type The list of databases (string list)

set_error(*error*)
Setter for error message

Parameters **error** – The string that contains the error message

Return type A string with the error

get_error()
Getter for error message (internally used for to show the error when some is wrong)

Return type A string with the error

get_options()
Getter for options object

Return type dbpatch.options.Options

pre_execute()
This method it will be executed **once** before of execute the patch in all databases. It must return true if the excution is ok, false in other case.

The child classes can overwrite this method.

Return type Boolean

before_database(*db_name*)
This method it will be executed before of execute the patch in each database.

The child classes can overwrite this method.

Parameters **db_name** – The database name. That is, the name of the database where the patch will be applied.

execute(*db_name*)
This method it will be executed in each database. It must return true if the excution is ok, false in other case

The child classes **should** overwrite this method.

Parameters **db_name** – The database name. That is, the name of the database where the patch will be applied.

Return type Boolean

after_database(*db_name*)
This method it will be executed after of execute the patch in each database

The child classes can overwrite this method.

Parameters **db_name** – The database name. That is, the name of the database where the patch has been applied.

post_execute()
This method it will be executed **once** after of execute the patch in all databases. It must return true if the excution is ok, false in other case.

The child classes can overwrite this method.

Return type Boolean

PreparedStatement class

The **dbpatch.patch.PreparedStatement** class contains the next methods.

commit()
Commit the changes. Only if the statement has been created with autocommit=False

rollback()
Discard the changes. Only if the statement has been created with autocommit=False

execute()
Build, trace, and execute the SQL sentence

get_row()
Return the first row obtained of SQL sentence execution.

Return type Tuple

get_rows()
Return all the rows obtained of SQL sentence execution.

Return type Tuple list

1.3.3 Raw patches

A raw patch run a file as a single SQL sentence. This kind of patch it is useful when it is necessary work with complex SQL statements (contains several ";" characters, etc). By example, it could be used for introduce triggers in the databases.

1.4 Invoking db-patch

For a complete and documented option list execute the next command:

```
$> db-patch --help
```

Basic invokations:

```
$> db-patch -u db_user -p db_password -E MySQL -d db2patch -f patch.py --odbc-driver "MySQL"
$> db-patch -u db_user -p db_password -E MySQL -i db2* -f patch.py --odbc-driver "MySQL"
```

The options *-u*/*-p* defines the user and password for connect with the database, and the *-f* option determine the patch file to apply.

The *-E* option defines the database engine, and the *–odbc-driver* defines the odbc driver to use (it can exist several drivers for the same database engine).

The option *-d* define the database names to patch; On the other hand, the option *-i* use a regular expresion (see the re python module) for recover the databases to patch

It is possible delegate the command line options to a file (for avoid bash history issues, etc). For this purpose you can use the option **–args-file** and provide the file name. For the previous examples the file should contain the next line:

```
-u db_user -p db_password -E MySQL -i db2* -f patch.py --odbc-driver "MySQL"
```

and should be invoked with the next command:

```
$> db-patch --args-file file_with_options.txt
```

1.4.1 More ODBC options

Note that it is possible customize the ODBC connection via command line:

- **–odbc-dsn**. Define a DSN for ODBC connections
- **–odbc-extra**. Provide extra options for make the ODBC connection string
- **–odbc-full-connection**. Define the full ODBC connection string.

1.4.2 SQL tracing options

-c, –clean Avoid that all the SQL executed will be stored

–gen-dir=GENERATED_DIR Directory where the SQL sentences for each patch will be stored. By default: sql_sentences

–gen-file=GEN_FILE Determines the name of the file where the generated SQL is stored.The *HOST* flag will be replaced by the hostname passed by command line. The *NAME* flag will be replaced by the patch name without extension. The *TIME* flag will be replaced by a timestamp. By default: “HOST_NAME_TIME.sql”

-S, –show-sql Show the generated SQL sentences in the standard output (stdout)

CHAPTER

TWO

DEVELOPER GUIDE

This guide details how to create plugins for use *db-patch* with other databases, drivers or for create customized patches.

2.1 Adding ODBC connections

Currently only three databases are supported: MySQL, PostgreSQL and SQLite. However, the mechanism for support other databases is simple.

For connect to one database using ODBC only it is necessary define correclty the connection string. However, each database use its keywords for this purpose. Therefore, it is necessary associate each custom keyword with the keyword used by *db-patch*. The next table shows the default keywords used by *db-patch* for create an ODBC connection string.

Keyword	Meaning
USER	Database user
PASSWORD	Database password
SERVER	Hostname of database server
PORT	Port of database server
DRIVER	ODBC driver
CHARSET	Database charset

For instance, PostgreSQL uses the keyword *UID* instance of *USER* and *PWD* instance of *PASSWORD*. Therefore, it is necessary customize the connection string. To do this, it is necessary create a customized OdbcQueryBuilder and overwrite the method *get_customized_keywords* as it is shown in the next code listing.

```
import dbpatch.connection.odbc as odbc

class PostgreSQLQueryBuilder(odbc.OdbcQueryBuilder):

    NAME = 'postgresql'

    def get_internal_database(self):
        return 'postgres'

    def get_all_databases_query(self):
        return "SELECT datname FROM pg_database WHERE datistemplate = false"

    def get_customized_keywords(self):
        return {'USER'     : 'UID',
                'PASSWORD' : 'PWD'}
```

```
def register():
    odbc.OdbcQueryBuilderFactory.register(PostgreSQLQueryBuilder)
```

The previous code listing shows other two methods that should be overwritten:

- **get_internal_database**. Return the name of the internal database used for the database engine. *db-patch* will connect with this database for recover the full list of working databases.
- **get_all_databases_query**. Return the query that will recover the full list of working databases.

Other two aspects should be noted:

- The **NAME** class attribute. Define the name that it will be used by db-patch for register the customized ODBC QueryBuilder.

  Note: It is mandatory define the *NAME* attribute. The *NAME* is provided by the -E (–db-engine) command line option.

- The **register** module method. Register the customized OdbcQueryBuilder for to be used by db-patch

  ---

  Note: If the customized OdbcQueryBuilder is not registered, *db-patch* will not found it.

The method that creates the final ODBC connection string is *OdbcQueryBuilder.get_connection_string*. It is possible overwrite, but previously it is recommended to show the code of this method because it makes checks, and add optional options.

2.2 Adding native connections

It is possible connect with the database engine using specific drivers instead of ODBC. For to do this, it is necessary provide a customized *patcher.connection.Connection* class and to use the *–connection-plugins* command line option.

For drivers compliant with the DB-API 2.0 specification the class *patcher.connection.DbApi2Connection* is provided. A new driver that use this class as parent class only should overwrite a few methods, define the *NAME* class attribute and register the connection (create a module method called register and use the method *db-patch.connection.ConnectionFactory.register_connection*). Next, the methods to overwrite are enumerated.

- **connect**. Connect with one database.
- **get_internal_database**. Return the name of the internal database.
- **get_all_databases**. Recover the full list of databases.

Currently there are three native implementations that can be used as examples:

- For MySQL, using the MySQLdb driver
- For PostgreSQL, using the psycopg2 driver.
- For SQLite, using the sqlite3 python module.

Note: For to use your new native connection it is necessary to provide the next command line options to *db-patch*

- *-N* or *–native*
- *–connection-plugins*

2.3 Creating new patches

For create a new patch only it is necessary inherit of the class *dbpatch.patch.Patch*, provide an implementation of the *execute* method, add the EXTENSION class attribute, register the new patch (creating a module method called register and using the method *dbpatch.patch.PatchFactory.register*), and use the *–patch-plugins* command line option.

As example see the implementation of *raw* and *sql* patches (the code shown below correspond to used for manage *raw* patches).

```
import dbpatch.options
from dbpatch.log import LogFactory
import dbpatch.patch


class RawPatch(dbpatch.patch.Patch):
    '''
    Execute all the content of a file in a single query. Useful by
    example for PL SQL scripts. Be careful, you must introduce
    a single query.

    The keyword 'mydb' will be replaced by each database name
    '''

    EXTENSION = 'raw'

    def __init__(self):
        dbpatch.patch.Patch.__init__(self)
        self._log  = LogFactory().get_logger(RawPatch.__name__)
        patch_file = self._options.get_option(dbpatch.options.PATCH_FILE)
        self._queries = open(patch_file, 'r').read()


    def execute(self, db_name):
        connection = self.connect_to(db_name)
        queries = self._queries.replace('mydb', db_name)
        statement = self.create_statement(queries)
        status = statement.execute()
        if not status:
            return False

        return True


def register():
    dbpatch.patch.PatchFactory.register(RawPatch)
```

2.4 Providing more options

Your new plugins could require more options. For this purpose you can use the **–extra-opts** command line option. This option recover a properties file and add them into the *dbpatch.options.Options* object. That is, for the next file

```
[myopt]
opt1 = first
opt2 = second
```

using the command line option

```
$> db-patch [...] --extra-opts myopt.cfg
```

your plugins can recover this properties from the *Options* object using the keywords *myopt.opt1* and *myopt.opt2* (*section.option_name*)

```
[...]

your_opt1 = Context().options.get_option('myopt.opt1')
your_opt2 = Context().options.get_option('myopt.opt2')

[...]
```

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

INDEX