Модуль MySQLdb в Python, клиент БД MySQL

Кл… базы данных MySQL для языка Python

Мод… …бертка Python для клиента MySQLdb._mysql написанного на языке C, которая делает его совместимым с и… …ерсии 2). Из соображений быстродействия и эффективности значительная часть кода, реализующего Pyt… …-249), находится в подмодуле MySQLdb._mysql.

При разработке интерфейса доступа к базам данных, основным руководством была спецификация PEP-249.

Содержание:

Установка модуля MySQLdb в OS Linux.

Так как MySQLdb написана на Python с использованием языка C и непосредственно использует API MySQL, то первым шагом необходимо установить заголовки и библиотеки для разработки Python 3 и MySQL следующим образом:

```
# для Debian / Ubuntu
$ sudo apt-get install python3-dev default-libmysqlclient-dev build-essential
# Red Hat / CentOS
% sudo yum install python3-devel mysql-devel
```

Обратите внимание, что это основной шаг для успешной установки модуля в Linux системах. Если на этом этапе возникает какая-то ошибка, то исправить ее нужно самостоятельно или обратиться за поддержкой на какой-нибудь форум пользователей.

Затем можно установить модуль MySQLdb в виртуальное окружение через менеджер пакетов pip:

```
# создаем виртуальное окружение, если нет
$ python3 -m venv .venv --prompt VirtualEnv
# активируем виртуальное окружение
$ source .venv/bin/activate
# обновляем `pip`
(VirtualEnv):~$ python3 -m pip install -U pip
# ставим модуль `MySQLdb`
(VirtualEnv):~$ python3 -m pip install -U mysqlclient
```

Это не опечатка, модуль MySQLdb зарегистрирован под именем mysqlclient.

Та… …OS Linux, модуль MySQLdb можно установить из заранее скомпилированных репозиториев (более простой вариант, но версия MySQLdb может быть ниже последней стабильной):

```
$ sudo apt-get install python3-mysqldb
```

Эта команда установит модуль MySQLdb для всей операционной системы Linux.

Установка модуля MySQLdb в OS Windows.

Установка модуля MySQLdb в OS Windows очень сложное. Но есть некоторые бинарные исходники, которые можно легко установить. Если для установленной в OS версии Python не существует бинарных исходников, то можно попробовать выполнить сборку из исходного кода при помощи pip, но это может не сработать.

Чтобы выполнить сборку из исходного кода, загрузите коннектор C MariaDB и установите его. Он должен быть установлен в директорию по умолчанию (обычно это "C:\Program Files\MariaDB\MariaDB Connector C" или "C:\Program Files (x86)\MariaDB\MariaDB Connector C" для 32-разрядной версии). Если установить коннектор в другое место, то перед запуском pip install mysqlclient, необходимо установить переменную среды MYSQLCLIENT_CONNECTOR. Затем, после установки коннектора и соответствующей версии Visual Studio (необходимо для компиляции исходного кода модуля) для версии Python, запустите установку MySQLdb: c:/pip install mysqlclient.

Особенности модуля MySQLdb.

По сути, модуль MySQLdb предоставляет пользователям два разных API для связи с базой данных MySQL:

- "*MySQL C API translation*" - официальный API MySQL для языка программирования C (расположен в подмодуле MySQLdb._mysql);
- "*Python Database API*" - официальная спецификация Python для унификации доступа к различным базам данных (PEP 249). Представляет собой тонкую обертку Python вокруг подмодуля MySQLdb._mysql, которая не теряет скорость работы с базой данных.

Если нужно писать приложения, переносимые между базами данных (MySQL <=> Postgree <=> SQLite), то используйте "*Python Database API*" и избегайте прямого использования "*MySQL C API translation*". MySQLdb._mysql предоставляет интерфейс, который в основном реализует MySQL C API. Описание подмодуля MySQLdb._mysql будет поверхностным, так как для доступа к MySQL необходимо использовать API более высокого уровня. Если по каким-то причинам необходимо работать с MySQLdb._mysql, то используйте стандартную документацию "*MySQL C API Developer Guide*".

Алгоритм использования MySQLdb.

Высокоуровневый "*Python Database API*" хорошо справляется с обеспечением достаточно переносимого интерфейса, но некоторые дополнительные методы (которых нет в PEP 249) и аргументы не переносимы. В частности, некоторые аргументы, принимаемые функцией MySQLdb.connect(), полностью зависят от реализации.

Прежде чем выполнять запросы к имеющейся базе данных MySQL, необходимо импортировать модуль MySQLdb и создать подключение к базе:

```
>>> import MySQLdb
# база данных `thangs` должна быть уже создана
>>> db = MySQLdb.connect(passwd="******", db="thangs")
```

Для выполнения запроса к базе данных MySQL создается курсор, а потом уже можно в нем выполнять запросы:

```
>>> cursor = db.cursor()
>>> max_price = 5
# таблица `breakfast` должна иметь соответствующие записи
>>> cursor.execute("""SELECT spam, eggs, sausage FROM breakfast
          WHERE price < %s""", (max_price,))
```

В этом примере max_price = 5 (уже известен), зачем тогда использовать литерал %s в строке? Потому что MySQLdb преобразует его в буквальное значение SQL, которое представляет собой строку '5'. Когда преобразование будет завершено, то запрос фактически выполнит: "... WHERE price < 5".

Почему подстановка выполняется в виде кортежа (max_price,) ? Потому что API БД требует, чтобы передавались любые параметры в виде последовательности. Из-за конструкции синтаксического анализатора (max_price) интерпретируется как использование алгебраической группировки и просто как max_price, а не как кортеж. Добавление последней запятой внутри скобок, то есть (max_price,), заставляет его создать кортеж.

А теперь результаты:

```
>>> cursor.fetchone()
# (3L, 2L, 0L)
```

Ме[Вверх]sor.fetchone() возвращает один кортеж, это значение является строкой таблицы БД, и значения по умолчанию пр[illegible]тся правильно.

Что это за буквы 'L'? В то время как столбец INTEGER в MySQL отлично преобразуется в целое число Python, целое число без знака UNSIGNED INTEGER может переполниться, и поэтому, эти значения преобразуются в длинные целые числа Python.

Если необходимо получение большего количества строк из таблицы MySQL, то можно использовать методы курсора cursor.fetchmany(n) или cursor.fetchall(). Они делают именно то, как называются. В cursor.fetchmany(n), значение n является необязательным и по умолчанию равно свойству курсора cursor.arraysize, которое обычно равно 1. Оба этих метода возвращают последовательность строк или если строк больше нет, то пустую последовательность. Если использовать другой класс курсора, например cursors.DictCursor, то сами строки таблицы будут иметь вид словаря, где в качестве ключей будут значения имен столбцов таблицы.

Обратите внимание, что в отличие от сказанного выше, метод cursor.fetchone() возвращает None, когда больше нет строк для выборки.

Единственный метод, отличный от представленных выше, который еще будете использовать, это метод cursor.executemany(), который позволяет выполнить множественный запрос:

```
cursor.executemany(
      """INSERT INTO breakfast (name, spam, eggs, sausage, price)
      VALUES (%s, %s, %s, %s, %s)""",
      [
      ("Тарелка для любителей спама и сосисок", 5, 1, 8, 7.95 ),
      ("Не так много спама", 3, 2, 0, 3.95 ),
      ("Не ждите НИКАКОГО СПАМА!", 0, 4, 3, 5.95 )
      ] )
```

Здесь вставляется три строки по пять значений. Обратите внимание, что вставляется смесь типов (строки, целые числа, числа с плавающей запятой), но по-прежнему используем только литерал %s. Также обратите внимание, что строки формата используется только для одной строки. Модуль MySQLdb выбирает их и дублирует для каждой строки.

Освобождение ресурсов, занятых MySQLdb.

Повторим лучшею практику для всех, кто сталкивается с SQL-соединением с использованием MySQLdb или любого другого пакета для подключения python3 к базам данных. Это необходимо знать.

Процесс извлечения данных из БД должен состоять из 7 шагов:

- Создать соединение;
- Создать курсор;
- Создать строку запроса;
- Выполнить запрос;
- Подтвердить запрос;
- Закрыть курсор;
- Закрыть соединение;

Общий случай cursor.close() и connect.close().

(Следующий пример предполагает, что есть таблица с именем tablename. У нее есть 4 столбца/поля с именами field1, field2, field3, field4). Если соединение локальное (та же машина), то это '127.0.0.1', также известный как 'localhost'.

```
import MySQLdb
# создаем соединение
mydb = MySQLdb.connect(host=host, user=user, passwd=passwd, db=database, charset="utf8")
# создаем курсор
cursor = mydb.cursor()
# создаем строку запроса с использованием подстановок
query = """INSERT INTO tablename (text_for_field1, text_for_field2, text_for_field3, text_for_field4)
         VALUES (%s, %s, %s, %s)"""
# выполняем запрос
cursor.execute(query, (field1, field2, field3, field4))
# фиксируем/подтверждаем выполнение запроса на вставку
mydb.commit()
# закрываем курсор
cursor.close()
# закрываем соединение (если больше не оно не нужно)
mydb.close()
```

Соединение с базой данных и объект курсора - разные вещи. Соединение находится на уровне SQL сервера, а курсор можно рассматривать как элемент данных. Можно иметь **несколько курсоров** для одних и тех же данных **в рамках одного соединения.** Не Вверх подключений к одной и той же базе данных с одного клиента/компьютера - редкое явление.

> *Правильное понимание курсора описано в этой цитате: - "Парадигма курсора не специфична для Python, но является частой структурой данных в самих базах данных". Кстати, в API C MySQL нет курсоров и подстановок. Модуль MySQLdb ("Python Database API") эмулирует курсор, это требование спецификация PEP 249 для переносимости/совместимости кода Python с другими базами данных.*

В зависимости от базовой реализации, можно создать несколько курсоров, использующих одно и то же соединение с базой данных. Закрытие курсора должно освободить ресурсы, связанные с запросом (освободить оперативную память), включая любые результаты, которые никогда не извлекались из БД (или извлекались, но не использовались), но не устраняло соединение с самой базой данных. Поэтому можно в рамках одного соединения MySQLdb.connect получить новый курсор в той же базе данных. без необходимости повторной аутентификации.

Полный пример работы с модулем MySQLdb:

Модуль MySQLdb не имеет встроенных менеджеров контекста для закрытия курсора или соединения. Так как же удобнее и лучше использовать cursor.close() и connect.close()? Наша команда, для закрытия курсора/очистки ОЗУ использует функцию стандартной библиотеки contextlib.closing, а соединение с MySQL обычно закрываем в конце кода.

Алгоритм использования:

```
from MySQLdb import connect, cursors, Error
from contextlib import closing

# конфигурация соединения с базой данных
MYSQLCONF = {
    'host': 'localhost', # хост базы данных
    'user': '******', # имя пользователя базы данных
    'password': '******', # пароль пользователя базы данных
    'db': 'test_db', # имя базы данных
    'charset': 'utf8', # используемая кодировка базы данных
    'autocommit': True, # автоматический cursor.commit()
    # извлекаемые строки из БД принимают вид словаря
    'cursorclass': cursors.DictCursor
}

# поднимаем соединение с базой данных
db = connect(**MYSQLCONF)

with closing(db.cursor()) as cursor:
    # создаем строку с запросом с использованием подстановок
    query = """INSERT INTO table (field1, field2, field3)
               VALUES (%s, %s, %s)"""
    try:
        # выполняем запрос с необходимыми параметрами
        cursor.execute(query, (param1, param2, param3,))
        # здесь должна быть строка с командой `cursor.commit()`
        # но ее нет, т.к. соединение настроено с 'autocommit': True
    except Error as err:
        # Класс `Error` выводит ошибки, связанные с работой базы данных
        # и не обязательно находящихся под контролем программиста
        print(err)
    # если нужно, то можно получить `id` вставленной записи
    print('lastrowid =>', cursor.lastrowid)

# далее какой-то код программы
...

# если программа от запроса к запросу выполняется со
# значительными задержками, то лучше проверить наличие
# открытого соединения, т.к. сервер MySQL может разорвать
# его по таймауту (определен в конфигурации MySQL сервера)
if not db:
    db = connect(**MYSQLCONF)

# обратите внимание, что если сервер MySQL не разорвал соединение с
# клиентом, то будет использоваться старое соединение с базой данных
with closing(db.cursor()) as cursor:
    query = """SELECT id, field1, field3 FROM table
               WHERE field2 = %s
               ORDER BY field2 DESC"""
    try:
        cursor.execute(query, (param2,))
    except Error as err:
```

Вверх

```
        print(err)

    data_table = cursor.fetchall()

# если запросов к базе данных в коде далее больше не будет,
# то закрываем соединение, но предварительно проверим, вдруг
# соединение разорвал сервер или сборщик мусора Python...
if db:
    db.close()

# обрабатываем данные запроса к таблице
for row in data_table:
    rowid = row['id']
    field1 = row['field1']
    field3 = row['field3']
    # далее что-то делаем с подученными
    # данными из таблицы
    ...
```

Содержание раздела:

Вверх