Python 使用 MySQL 数据库完整指南

Python 提供了多种方式连接和操作 MySQL 数据库，以下是常用的方法和最佳实践。

1. 安装 MySQL 驱动

常用驱动选择

```
# 最流行的纯 Python驱动
pip install mysql-connector-python

# 更快的 C扩展驱动 (推荐)
pip install pymysql

# MySQL官方驱动
pip install mysql-connector-python

# 另一个流行选择
pip install MySQL-python  # 注意: 仅支持 Python 2.x和旧版 3.x
```

2. 连接 MySQL 数据库

使用 PyMySQL (推荐)

```
import pymysql

# 建立连接
connection = pymysql.connect(
```

```
    host='localhost',
    user='username',
    password='password',
    database='dbname',
    charset='utf8mb4',
    cursorclass=pymysql.cursors.DictCursor  # 返回字典格式结果
)

try:
    with connection.cursor() as cursor:
        # 执行  SQL查询
        sql = "SELECT * FROM `users` WHERE `email`=%s"
        cursor.execute(sql, ('user@example.com',))

        # 获取结果
        result = cursor.fetchone()
        print(result)
finally:
    connection.close()
```

使用 mysql-connector-python（官方驱动 ）

```
import mysql.connector

config = {
    'user': 'username',
    'password': 'password',
    'host': 'localhost',
    'database': 'dbname',
    'raise_on_warnings': True
}

cnx = mysql.connector.connect(**config)
cursor = cnx.cursor(dictionary=True)  # 返回字典格式结果

query = "SELECT * FROM employees WHERE hire_date > %s"
hire_start = datetime.date(1999, 1, 1)
cursor.execute(query, (hire_start,))
```

```
for row in cursor:
    print(row)

cursor.close()
cnx.close()
```

3. CRUD □□□

□□□

```
def create_table():
    conn = pymysql.connect(host='localhost', user='root', password='', database='test')
    try:
        with conn.cursor() as cursor:
            sql = """
            CREATE TABLE IF NOT EXISTS `users` (
                `id` INT AUTO_INCREMENT PRIMARY KEY,
                `name` VARCHAR(255) NOT NULL,
                `email` VARCHAR(255) NOT NULL UNIQUE,
                `created_at` TIMESTAMP DEFAULT CURRENT_TIMESTAMP
            ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
            """
            cursor.execute(sql)
        conn.commit()
    finally:
        conn.close()
```

□□□

```
def insert_user(name, email):
    conn = pymysql.connect(host='localhost', user='root', password='', database='test')
    try:
        with conn.cursor() as cursor:
            sql = "INSERT INTO `users` (`name`, `email`) VALUES (%s, %s)"
```

```
            cursor.execute(sql, (name, email))
        conn.commit()
        return cursor.lastrowid
    except pymysql.err.IntegrityError:
        print("Email already exists")
        return None
    finally:
        conn.close()
```

```
def get_users(page=1, per_page=10):
    conn = pymysql.connect(host='localhost', user='root', password='', database='test')
    try:
        with conn.cursor() as cursor:
            offset = (page - 1) * per_page
            sql = "SELECT * FROM `users` LIMIT %s OFFSET %s"
            cursor.execute(sql, (per_page, offset))
            return cursor.fetchall()
    finally:
        conn.close()
```

```
def update_user(user_id, name=None, email=None):
    conn = pymysql.connect(host='localhost', user='root', password='', database='test')
    try:
        with conn.cursor() as cursor:
            updates = []
            params = []
            if name:
                updates.append("`name` = %s")
                params.append(name)
            if email:
                updates.append("`email` = %s")
                params.append(email)
```

```
            if updates:
                params.append(user_id)
                sql = f"UPDATE `users` SET {', '.join(updates)} WHERE `id` = %s"
                cursor.execute(sql, params)
                conn.commit()
                return cursor.rowcount
            return 0
    except pymysql.err.IntegrityError:
        print("Email already exists")
        return None
    finally:
        conn.close()
```

□□□

```
def delete_user(user_id):
    conn = pymysql.connect(host='localhost', user='root', password='', database='test')
    try:
        with conn.cursor() as cursor:
            sql = "DELETE FROM `users` WHERE `id` = %s"
            cursor.execute(sql, (user_id,))
            conn.commit()
            return cursor.rowcount
    finally:
        conn.close()
```

4. □□ ORM (SQLAlchemy)

```
from sqlalchemy import create_engine, Column, Integer, String, DateTime
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from datetime import datetime

# □□□□□□□ : mysql+pymysql://<username>:<password>@<host>/<dbname>[?<options>]
engine = create_engine('mysql+pymysql://root:@localhost/test?charset=utf8mb4')
Base = declarative_base()
```

```
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String(255), nullable=False)
    email = Column(String(255), unique=True, nullable=False)
    created_at = Column(DateTime, default=datetime.now)

# 创建表
Base.metadata.create_all(engine)

# 创建会话
Session = sessionmaker(bind=engine)
session = Session()

# 添加新用户
new_user = User(name="John Doe", email="john@example.com")
session.add(new_user)
session.commit()

# 查询用户
user = session.query(User).filter_by(email="john@example.com").first()
print(user.name, user.email)

session.close()
```

5. 连接池管理

使用 DBUtils 创建连接池

```
from dbutils.pooled_db import PooledDB
import pymysql

pool = PooledDB(
    creator=pymysql,
```

```
    maxconnections=10,
    mincached=2,
    host='localhost',
    user='root',
    password='',
    database='test',
    charset='utf8mb4'
)

def get_users():
    conn = pool.connection()
    try:
        with conn.cursor() as cursor:
            cursor.execute("SELECT * FROM users")
            return cursor.fetchall()
    finally:
        conn.close()
```

6. 最佳实践

1. 始终使用参数化查询 - 防止 SQL 注入

```
# 错误做法
cursor.execute(f"SELECT * FROM users WHERE name = '{name}'")

# 正确做法
cursor.execute("SELECT * FROM users WHERE name = %s", (name,))
```

2. 使用上下文管理器 - 自动关闭连接

```
with pymysql.connect(...) as conn:
    with conn.cursor() as cursor:
        cursor.execute(...)
```

3. 处理事务 - 确保数据一致性

```
try:
    conn.begin()
    # 执行多个操作
```

```
    conn.commit()
except:
    conn.rollback()
    raise
```

4. 设置合适的字符集 - 推荐使用 `utf8mb4` 支持完整 Unicode
5. 连接管理 - 生产环境使用连接池
6. 错误处理 - 捕获特定异常

```
try:
    cursor.execute(...)
except pymysql.err.IntegrityError as e:
    print("Duplicate entry:", e)
except pymysql.err.ProgrammingError as e:
    print("SQL syntax error:", e)
```

7. 性能优化

1. 使用索引 - 确保查询字段有适当索引
2. 批量操作 - 减少数据库往返

```
# 批量插入
sql = "INSERT INTO users (name, email) VALUES (%s, %s)"
cursor.executemany(sql, [('a','a@a.com'), ('b','b@b.com')])
```

3. 只查询需要的列 - 避免 `SELECT *`
4. 使用连接池 - 减少连接创建开销
5. 合理设置缓存 - 对不常变的数据使用缓存

通过以上方法，你可以在 Python 中高效安全地使用 MySQL 数据库。

Revision #1
Created 20 May 2025 11:15:08 by Admin
Updated 20 May 2025 11:18:07 by Admin