# Tại sao lại cần Connection Pool

Bình thường với mỗi client request sẽ mở 1 kết nối tới database server và rồi lại đóng lại.

Việc tạo và hủy các kết nối tới DB sẽ ảnh hưởng rất nhiều tới hiệu năng của ứng dụng. Điển hình là database server sẽ mất từ 1 đến 3s để thiết lập kết nối tới DB

Connection Pooling là kĩ thuật để khắc chế việc này

## Connection Pooling

Cơ chế hoạt động của nó như sau: khi một connection (một kết nối) được tạo, nó sẽ được đưa vào pool và sử dụng lại cho các yêu cầu kết nối tiếp theo và chỉ bị đóng khi hết thời gian timeout.

Ví dụ, max pool size = 10 (số lượng tối đa connection trong pool là 10).

Bây giờ user kết nối tới database (truy vấn database), hệ thống sẽ kiểm tra trong connection pool có kết nối nào đang rảnh không?

* Trường hợp chưa có kết nối nào trong connection pool hoặc tất cả các kết nối đều bận (đang được sử dụng bởi user khác) và số lượng connection trong connection < 10 thì sẽ tạo một connection mới tới database để kết nối tới database đồng thời kết nối đó sẽ được đưa vào connection pool.
* Trường hợp tất cả các kết nối đang bận và số lượng connection trong connection pool = 10 thì người dùng phải đợi cho các user dùng xong để được dùng.

Sau khi một kết nối được tạo và sử dụng xong nó sẽ không đóng lại mà sẽ duy trì trong connection pool để dùng lại cho lần sau và chỉ thực sự bị đóng khi hết thời gian timeout (lâu quá không dùng đến nữa)

MySQL2 là thư viện cho phép sử dụng MySQL-Native

# Install

npm install --save mysql2

# Create Connection

const mysql = require('mysql2');

const connection = mysql.createConnection({

  host: 'localhost',

  user: 'root',

  database: 'test'

});

# Create Pool

const pool = mysql.createPool({

  host: 'localhost',

  user: 'root',

  database: 'test',

  waitForConnections: true,

  connectionLimit: 10,

  queueLimit: 0

});

Sử dụng Pool giống như Connection :

2 methods :

* pool.query()
* pool.execute()

## Auto Release

pool.query("SELECT field FROM atable", function(err, rows, fields) {

*// Connection is automatically released when query resolves*

})

## Manually Release

pool.getConnection(function(err, conn) {

*// Do something with the connection*

   conn.query(*/\* ... \*/*);

*// Don't forget to release the connection when finished!*

   pool.releaseConnection(conn);

})

# Query

// simple query

connection.query(

  'SELECT \* FROM `table` WHERE `name` = "Page" AND `age` > 45',

  function(err, results, fields) {

    console.log(results); // results contains rows returned by server

    console.log(fields); // fields contains extra meta data about results, if available

  }

);

// with placeholder

connection.query(

  'SELECT \* FROM `table` WHERE `name` = ? AND `age` > ?',

  ['Page', 45],

  function(err, results) {

    console.log(results);

  }

);

# Prepare Statement

Auto : tự prepare & query statement dùng execute()

* Sẽ create, cache & re-used by connection
* Nếu như execute nhiều lần thì performance tốt hơn

connection.execute('select 1 + ? + ? as result', [5, 6], (err, rows, fields) => {

});

// close cached statement for 'select 1 + ? + ? as result'. noop if not in cache

connection.unprepare('select 1 + ? + ? as result');

* **rows** : chứa kết quả return
* **fields** : chứa extra metadata about result

Manually :

connection.prepare('select ? + ? as tests', (err, statement) => {

// statement.parameters - array of column definitions, length === number of params, here 2

// statement.columns - array of result column definitions. Can be empty if result schema is dynamic / not known

// statement.id

// statement.query

statement.execute([1, 2], (err, rows, columns) => {

// -> [ { tests: 3 } ]

});

// note that there is no callback here. There is no statement close ack at protocol level.

statement.close();

});

# Use Promise Wrapper

## Connection

const mysql = require("mysql2/promise");

// or require('mysql2').createConnectionPromise

const connection = mysql.createConnection({

        host: "localhost",

        user: "root",

        database: "test",

    })

    .then((conn) => conn.query("select foo from bar"))

    .then(([rows, fields]) => console.log(rows[0].foo));

Or async await

async function example1() {

    const mysql = require("mysql2/promise");

    const conn = await mysql.createConnection({ database: test });

    const [rows, fields] = await conn.execute("select ?+? as sum", [2, 2]);

    await conn.end();

}

## Pool

const pool = require("mysql2/promise").createPool({});

// or require('mysql2').createPoolPromise({})

// or require('mysql2').createPool({}).promise()

pool.getConnection()

    .then((conn) => {

        const res = conn.query("select foo from bar");

        conn.release();

        return res;

    })

    .then((result) => {

        console.log(result[0][0].foo);

    })

    .catch((err) => {

        console.log(err); // any of connection time or query time errors from above

    });

Or async await

async function example2() {

    const mysql = require("mysql2/promise");

    const pool = mysql.createPool({ database: test });

    // execute in parallel, next console.log in 3 seconds

    await Promise.all([

pool.query("select sleep(2)"),

pool.query("select sleep(3)")

]);

    console.log("3 seconds after");

    await pool.end();

}

## Name placeholder

connection.execute("select :x + :y as z",

    { x: 1, y: 2 },

    (err, rows) => {});

// unnamed

connection.query('select ? + ? as z',

    [1, 1],

    (err, rows) => {});

## Row as array

const options = { sql: 'select A,B,C,D from foo', rowsAsArray: true };

connection.query(options, (err, results) => {

  /\* results will be an array of arrays like this now:

  [[

     'field A value',

     'field B value',

     'field C value',

     'field D value',

  ], ...]

  \*/

});