資料庫 - DbClient

原文：ENG-08-1-Database-DbClient.md

DbClient 物件建構

DbClient 物件有兩種建構方式：一是透過 DbClient 類別的靜態方法（見 DbClient.h 標頭檔），如下：

```
#if USE_POSTGRESQL
    static std::shared_ptr<DbClient> newPgClient(const std::string
&connInfo, const size_t connNum);
#endif
#if USE_MYSQL
    static std::shared_ptr<DbClient> newMysqlClient(const std::string
&connInfo, const size_t connNum);
#endif
```

上述介面可取得 DbClient 實作物件的智慧指標。connInfo 為連線字串，採 key=value 參數格式，詳情請見標頭檔註解。connNum 為 DbClient 的連線數，對併發有關鍵影響，請依實際需求設定。

以此方式取得的物件，需自行持有（如存入全域容器）。**不建議建立暫時物件用完即釋放**，原因如下：

- 會浪費連線建立與斷開時間，增加系統延遲；
- 介面本身為非阻塞，取得 DbClient 時其管理的連線尚未建立，且框架不提供連線成功 callback，若要查詢還得 sleep？這違背非同步框架初衷。

因此 DbClient 物件應於程式啟動時建立並全程持有。此工作可完全交由框架處理，drogon 提供第二種建構方式：透過設定檔或 `createDbClient()` 方法建構。設定檔方式詳見db_clients。

需要時可透過框架介面取得 DbClient 智慧指標，介面如下：

```
orm::DbClientPtr getDbClient(const std::string &name = "default");
```

name 參數為設定檔中的 name 選項值，用於區分同一應用的多個 DbClient 物件。DbClient 管理的連線會自動重連，使用者無需關心連線狀態，幾乎隨時可用。**注意**：此方法不可於 app.run() 執行前呼叫，否則僅會取得空的 shared_ptr。

執行介面

DbClient 提供多種執行介面，列舉如下：

```
/// 非同步方法
template <typename FUNCTION1, typename FUNCTION2, typename... Arguments>
void execSqlAsync(const std::string &sql,
                  FUNCTION1 &&rCallback,
```

```
                    FUNCTION2 &&exceptCallback,
                    Arguments &&... args) noexcept;

/// 非同步方法（future）
template <typename... Arguments>
std::future<const Result> execSqlAsyncFuture(const std::string &sql,
                                             Arguments &&... args)
noexcept;

/// 同步方法
template <typename... Arguments>
const Result execSqlSync(const std::string &sql,
                         Arguments &&... args) noexcept(false);

/// 串流型方法
internal::SqlBinder operator<<(const std::string &sql);
```

因綁定參數數量與型別不定，這些方法皆為函式模板。

各方法特性如下表：

方法	同步/非同步	阻塞/非阻塞	例外處理
void execSqlAsync	非同步	非阻塞	不會丟出例外
std::future<const Result> execSqlAsyncFuture	非同步	get() 時阻塞	get() 時可能丟出例外
const Result execSqlSync	同步	阻塞	可能丟出例外
internal::SqlBinder operator<<	非同步	預設非阻塞	不會丟出例外

同步方法通常涉及網路 IO，故為阻塞；非同步方法則為非阻塞。但非同步方法也可阻塞執行，即方法會阻塞至 callback 執行完畢。此時 callback 會在呼叫者執行緒執行，然後方法才回傳。

高併發場景請用非同步非阻塞方法，低併發（如設備管理頁）可用同步方法以求直觀。

execSqlAsync

```
template <typename FUNCTION1, typename FUNCTION2, typename... Arguments>
void execSqlAsync(const std::string &sql,
                  FUNCTION1 &&rCallback,
                  FUNCTION2 &&exceptCallback,
                  Arguments &&... args) noexcept;
```

最常用的非同步介面，採非阻塞模式。

sql 為 SQL 字串，若有綁定參數請用對應資料庫的 placeholder 規則，如 PostgreSQL 用 $1, $2...，MySQL 用 ?。

args 為綁定參數，可為零或多個，數量需與 SQL placeholder 相同，型別可為：

- 整數型：各種長度整數，應與資料庫欄位型別相符；
- 浮點型：`float` 或 `double`，應與欄位型別相符；
- 字串型：`std::string` 或 `const char[]`，對應資料庫字串型或可用字串表示之型別；
- 日期型：`trantor::Date`，對應資料庫 date/datetime/timestamp；
- 二進位型：`std::vector<char>`，對應 PostgreSQL byte 或 MySQL blob；

參數可為左值或右值、變數或常數，皆可自由使用。

rCallback 與 exceptCallback 分別為結果 callback 與例外 callback，定義如下：

- 結果 callback：型別 void (const Result &)，可傳入 std::function、lambda 等；
- 例外 callback：型別 void (const DrogonDbException &)，可傳入相符型別 callable。

SQL 執行成功後，結果以 Result 類別包裝並透過結果 callback 回傳；若執行例外則執行例外 callback，可由 DrogonDbException 物件取得例外資訊。

範例：

```
auto clientPtr = drogon::app().getDbClient();
clientPtr->execSqlAsync("select * from users where org_name=$1",
                        [](const drogon::orm::Result &result) {
                            std::cout << result.size() << " rows
selected!" << std::endl;
                            int i = 0;
                            for (auto row : result)
                            {
                                std::cout << i++ << ": user name is " <<
row["user_name"].as<std::string>() << std::endl;
                            }
                        },
                        [](const DrogonDbException &e) {
                            std::cerr << "error:" << e.base().what() <<
std::endl;
                        },
                        "default");
```

Result 物件為標準容器，支援 iterator，可用 range loop 取得每列物件。Result、Row、Field 物件介面詳見原始碼。

DrogonDbException 為所有資料庫例外基底類別，詳見原始碼註解。

execSqlAsyncFuture

```
template <typename... Arguments>
std::future<const Result> execSqlAsyncFuture(const std::string &sql,
                                             Arguments &&... args)
noexcept;
```

非同步 future 介面省略前述兩個 callback，呼叫後立即回傳 future 物件，需呼叫 get() 取得結果。例外以 try/catch 機制取得，若 get() 未包在 try/catch 內且呼叫堆疊皆無 try/catch，SQL 執行例外時程式會直接結束。

範例：

```
auto f = clientPtr->execSqlAsyncFuture("select * from users where
org_name=$1",
                                       "default");
try
{
    auto result = f.get(); // 阻塞至取得結果或例外
    std::cout << result.size() << " rows selected!" << std::endl;
    int i = 0;
    for (auto row : result)
    {
        std::cout << i++ << ": user name is " <<
row["user_name"].as<std::string>() << std::endl;
    }
}
catch (const DrogonDbException &e)
{
    std::cerr << "error:" << e.base().what() << std::endl;
}
```

execSqlSync

```
template <typename... Arguments>
const Result execSqlSync(const std::string &sql,
                         Arguments &&... args) noexcept(false);
```

同步介面最簡單直觀，輸入 SQL 字串與綁定參數，回傳 Result 物件，呼叫時會阻塞執行緒，錯誤時丟出例外，需用 try/catch 處理。

範例：

```
try
{
    auto result = clientPtr->execSqlSync("update users set user_name=$1
where user_id=$2",
                                         "test",
                                         1); // 阻塞至取得結果或例外
    std::cout << result.affectedRows() << " rows updated!" << std::endl;
}
catch (const DrogonDbException &e)
{
    std::cerr << "error:" << e.base().what() << std::endl;
}
```

operator<<

```
internal::SqlBinder operator<<(const std::string &sql);
```

串流介面較特殊，透過 << 依序輸入 SQL 與參數，並用 >> 指定結果 callback 與例外 callback。例如前述 select 範例，串流介面寫法如下：

```
*clientPtr  << "select * from users where org_name=$1"
            << "default"
            >> [](const drogon::orm::Result &result)
                {
                    std::cout << result.size() << " rows selected!" <<
std::endl;
                    int i = 0;
                    for (auto row : result)
                    {
                        std::cout << i++ << ": user name is " <<
row["user_name"].as<std::string>() << std::endl;
                    }
                }
            >> [](const DrogonDbException &e)
                {
                    std::cerr << "error:" << e.base().what() << std::endl;
                };
```

此用法等同第一個非同步非阻塞介面，選用何種介面可依習慣。若需阻塞模式，可用 << 輸入 Mode::Blocking 參數，本文不詳述。

此外串流介面有特殊用法，若用特殊結果 callback，框架可逐列傳回結果，callback 型別如下：

```
void (bool,Arguments...);
```

第一個 bool 參數為 true 時表示已無資料（所有結果已回傳），即最後一次 callback；後面參數依序對應每列欄位值，框架會自動型別轉換，使用者也需注意型別相符。可用 const 左值參考、右值參考或值型別。

以下用此 callback 改寫前述範例：

```
int i = 0;
*clientPtr  << "select user_name, user_id from users where org_name=$1"
            << "default"
            >> [&i](bool isNull, const std::string &name, int64_t id)
                    {
                    if (!isNull)
                        std::cout << i++ << ": user name is " << name << ",
```

```
    user id is " << id << std::endl;
                    else
                        std::cout << i << " rows selected!" << std::endl;
                    }
                >> [](const DrogonDbException &e)
                    {
                        std::cerr << "error:" << e.base().what() << std::endl;
                    };
```

可見 select 語句的 user_name 與 user_id 欄位值分別賦予 callback 的 name 與 id 變數，無需自行處理型別轉換，十分方便，可彈性運用。

> **注意：非同步程式設計時，需留意上述範例中的 i 變數。必須確保 callback 執行時 i 仍有效，因為是以參考捕獲。callback 會在其他執行緒呼叫，當下 context 可能已失效。通常用智慧指標持有暫時變數並於 callback 捕獲，以確保變數有效。**

小結

每個 DbClient 物件都有一或多個 EventLoop 執行緒控制資料庫連線 IO，透過非同步或同步介面接收請求，並以 callback 回傳結果。

DbClient 的阻塞介面僅阻塞呼叫者執行緒，只要呼叫者非 EventLoop 執行緒，不會影響 EventLoop 正常運作。callback 執行時，程式會在 EventLoop 執行緒運作，因此勿於 callback 內執行阻塞操作，否則會影響資料庫讀寫併發效能。熟悉非阻塞 I/O 程式設計者應能理解此限制。

下一步: Transaction