資料庫 - 交易（Transaction）

原文：ENG-08-2-Database-Transaction.md

> **交易（Transaction）**是關聯式資料庫的重要功能，Drogon 以 `Transaction` 類別提供交易支援。

`Transaction` 物件由 `DbClient` 建立，許多交易相關操作會自動執行：

- 在建立 `Transaction` 物件時，自動執行 begin 指令以「啟動」交易；
- 當 `Transaction` 物件被解構時，自動執行 commit 指令以結束交易；
- 若發生例外導致交易失敗，則自動執行 rollback 指令以回滾交易；
- 若交易已回滾，則 SQL 指令會回傳例外（丟出例外或執行例外 callback）。

交易建立

由 `DbClient` 提供交易建立方法如下：

```
std::shared_ptr<Transaction> newTransaction(const std::function<void(bool)>
&commitCallback = std::function<void(bool)>())
```

此介面非常簡單，回傳 `Transaction` 物件的智慧指標。顯然，當智慧指標失去所有持有者並解構交易物件時，交易即結束。`commitCallback` 參數用於回傳交易提交是否成功。需注意，此 callback 僅用於指示 `commit` 指令是否成功。若交易在執行過程中自動或手動回滾，則不會執行此 callback。一般來說，`commit` 指令會成功，callback 的 bool 參數為 true。僅在特殊情況（如提交過程中連線中斷）才會通知提交失敗，此時伺服器端交易狀態不確定，使用者需特別處理。當然，考量此情況極少發生，非關鍵服務可選擇忽略此事件，建立交易時不傳入 `commitCallback`（預設為空 callback）。

交易必須獨佔資料庫連線。因此，建立交易時，`DbClient` 需從自身連線池選擇一個閒置連線交由交易物件管理。這會有一個問題：若 `DbClient` 所有連線都在執行 SQL 或其他交易，介面會阻塞直到有閒置連線。

框架也提供非同步交易建立介面如下：

```
void newTransactionAsync(const std::function<void(const
std::shared_ptr<Transaction> &)> &callback);
```

此介面透過 callback 回傳交易物件，不會阻塞目前執行緒，確保應用高併發。使用者可依實際情況選用同步或非同步版本。

交易介面

`Transaction` 介面幾乎與 `DbClient` 相同，僅有以下兩點差異：

- `Transaction` 提供 `rollback()` 介面，允許使用者在任何情況下回滾交易。有時交易已自動回滾，再呼叫 `rollback()` 也不會有負面影響，故明確呼叫 rollback() 是良好策略，至少可確保不會誤提交。
- 使用者不可呼叫交易物件的 `newTransaction()` 介面，這很容易理解。雖然資料庫有子交易概念，框架目前尚未支援。

事實上，`Transaction` 設計為 `DbClient` 的子類別，目的是維持介面一致性，同時也方便 ORM 使用。

目前框架未提供交易隔離層級控制介面，即隔離層級為資料庫服務預設值。

交易生命週期

交易物件的智慧指標由使用者持有。當有未執行的 SQL 時，框架會持有該物件，因此不必擔心交易物件在尚有未執行 SQL 時被解構。此外，交易物件的智慧指標常在其介面的結果 callback 中捕獲並使用。這是正常用法，不必擔心循環引用導致物件無法銷毀，框架會自動協助打破循環引用。

範例

以最簡單的例子說明，假設有一個任務表，使用者選取未處理任務並將其狀態改為處理中。為避免併發競爭，使用 `Transaction` 類別，程式如下：

```
{
    auto transPtr = clientPtr->newTransaction();
    transPtr->execSqlAsync( "select * from tasks where status=$1 for update
order by time",
                            "none",
                            [=](const Result &r) {
                                if (r.size() > 0)
                                {
                                    std::cout << "Got a task!" <<
std::endl;
                                    *transPtr << "update tasks set
status=$1 where task_id=$2"
                                              << "handling"
                                              << r[0]
["task_id"].as<int64_t>()
                                              >> [](const Result &r)
                                                 {
                                                    std::cout <<
"Updated!";
                                                    // ... 處理任務 ...
                                                 }
                                              >> [](const DrogonDbException
&e)
                                                 {
                                                    std::cerr << "err:" <<
e.base().what() << std::endl;
                                                 };
                                }
                                else
                                {
                                    std::cout << "No new tasks found!" <<
std::endl;
                                }
                            },
                            [](const DrogonDbException &e) {
                                std::cerr << "err:" << e.base().what() <<
std::endl;
```

```
                                });
    }
```

此例中，select for update 用於避免併發修改。update 語句在 select 結果 callback 中完成。最外層大括號用於限制 transPtr 生命週期，確保 SQL 執行完畢後能及時解構並結束交易。

下一步: ORM