# 認識 Drogon 的執行緒模型

原文：ENG-FAQ-1-Understanding-drogon-threading-model.md

Drogon 是一個高速的 C++ 網頁應用框架。它的高效部分原因在於不對底層執行緒模型做過度抽象，但這也常常造成一些困惑。常見的問題包括：為什麼某些阻塞呼叫後才送出回應、為什麼在同一個事件迴圈呼叫阻塞式網路函式會造成死結等。這篇文章旨在說明造成這些現象的條件，以及如何避免。

## 事件迴圈與執行緒

Drogon 以執行緒池運作，每個執行緒都有自己的事件迴圈（Event Loop）。事件迴圈是 Drogon 的核心。每個 Drogon 應用至少有兩個事件迴圈：主迴圈和工作迴圈。只要你沒做奇怪的事，主迴圈永遠在主執行緒（啟動 `main` 的執行緒）上執行，負責啟動所有工作迴圈。以 Hello World 範例來說，`app().run()` 會在主執行緒啟動主迴圈，然後產生三個工作執行緒/迴圈。

```
#include <drogon/drogon.h>
using namespace drogon;

int main()
{
    app().registerHandler("/", [](const HttpRequest& req
        , std::function<void (const HttpResponsePtr &)> &&callback) {
        auto resp = HttpResponse::newHttpResponse();
        resp->setBody("Hello wrold");
        callback(resp);
    });
    app().addListener("0.0.0.0", 80800);
    app().setNumThreads(3);
    app().run();
}
```

執行緒結構如下：

```
            .--------------.
            | app().run() | <main thread>
            :--------------:
                  |
            .-----v------.
            | MAIN LOOP  |
            :------------:
                  | Spawns
      .-----------+--------------.
      |           |              |
.-----v----.   .--v-------.  .---v----.
| Worker 1 |   | Worker 2 |  | etc... |
:----------:   :----------:  :--------:
 <thread 1>     <thread 2>   <thread ...>
```

工作迴圈數量取決於多個因素，像是 HTTP 伺服器指定的執行緒數、非 fast DB 及 NoSQL 連線數等（fast 與非 fast 連線後述）。總之，Drogon 不只 HTTP 伺服器執行緒。每個事件迴圈本質上就是一個任務佇列，具備以下功能：

- 從任務佇列讀取並執行任務。你可以從任何執行緒提交任務到事件迴圈。提交任務完全無鎖（lock-free），在所有情況下不會造成資料競爭。事件迴圈一次只處理一個任務，因此任務有明確的執行順序。但如果排在長時間任務後，其他任務就會延遲。
- 監聽並分派它管理的網路事件
- 執行定時器（通常由使用者建立）

當上述都沒發生時，事件迴圈/執行緒就會阻塞等待。

```
// 在主迴圈排入兩個任務
trantor::EventLoop* loop = app().getLoop();
loop->queueInLoop([]{
    std::cout << "task1: 我要等 5 秒\n";
    std::this_thread::sleep_for(5s);
    std::cout << "task1: 哈囉！\n";
});
loop->queueInLoop([]{
    std::cout << "task2: 世界！\n";
});
```

所以執行上會先看到 `task1: 我要等 5 秒`，暫停 5 秒後才會同時出現 `task1: 哈囉` 和 `task2: 世界`。

> 技巧 1：不要在事件迴圈裡呼叫阻塞式 IO，其他任務都得等它完成。

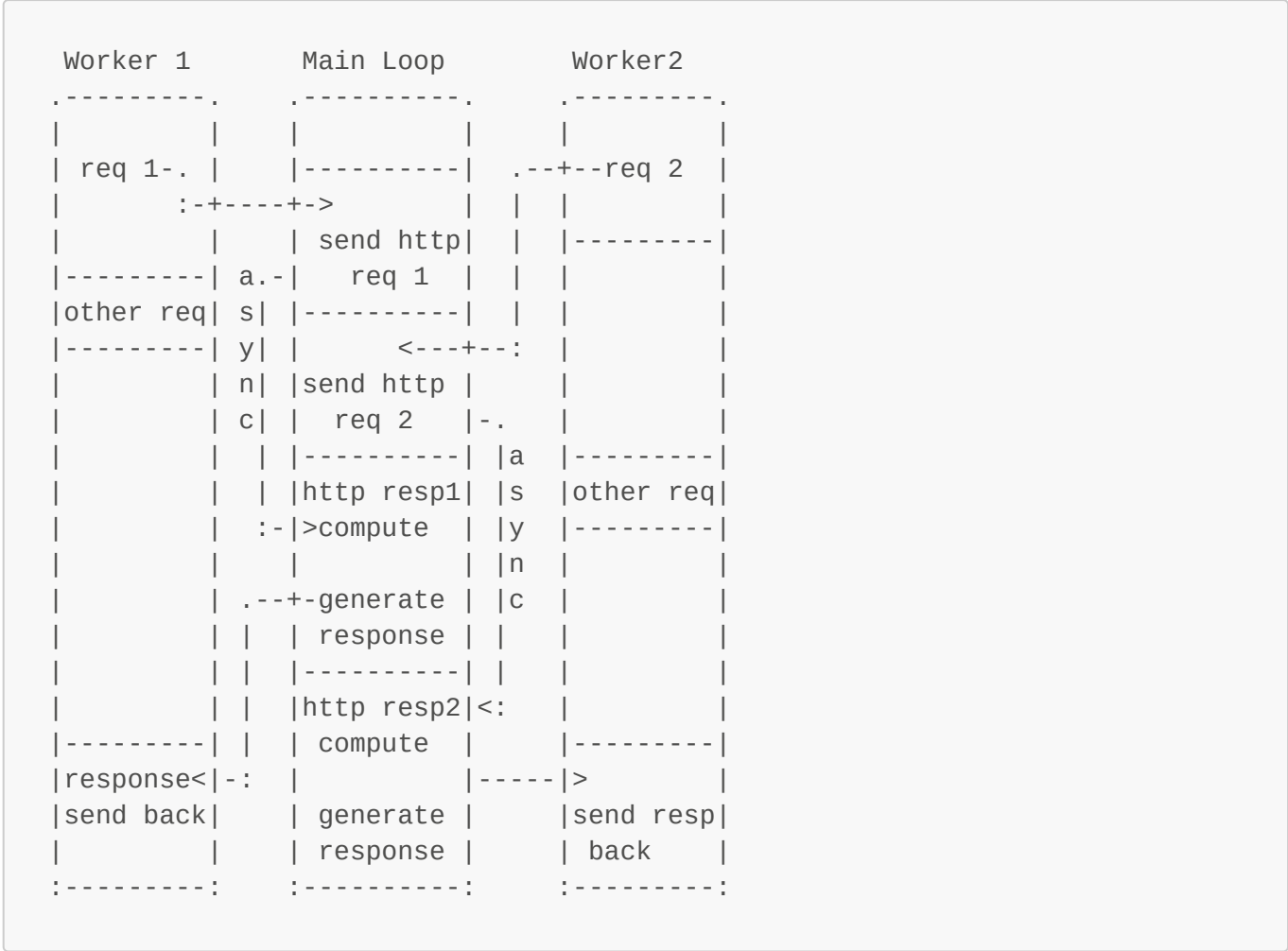
## 實務上的網路 IO

Drogon 幾乎所有東西都跟事件迴圈綁定，包括 TCP 連線、HTTP client、DB client、資料快取等。為了避免競爭，所有 IO 都在所屬事件迴圈執行。如果從其他執行緒呼叫 IO，參數會被存起來並以任務方式提交到正確的事件迴圈。這有些影響，例如在 HTTP handler 裡呼叫遠端端點或 DB 查詢時，client 的 callback 不一定（通常不會）在 handler 的同一執行緒執行。

```
app().registerHandler("/send_req", [](const HttpRequest& req
    , std::function<void (const HttpResponsePtr &)> &&callback) {
    // 這個 handler 會在某個 HTTP 伺服器執行緒執行

    // 建立在主迴圈運作的 HTTP client
    auto client = HttpClient::newHttpClient("https://drogon.org",
app().getLoop());
    auto request = HttpRequest::newHttpRequest();
    client->sendRequest(request, [](ReqResult result, const HttpResponse&
resp) {
        // 這個 callback 會在主執行緒執行
    });
});
```

所以如果你不注意，可能會塞爆事件迴圈，例如在主迴圈建立大量 HTTP client 並送出所有請求，或是在 DB callback 裡執行大量運算，導致其他 DB 查詢被延遲。




HTTP 伺服器也是一樣。如果回應是在其他執行緒產生（例如在 DB callback 裡），回應會排入所屬執行緒等候送出，而不是立即送出。

> 技巧 2：注意你的運算放在哪裡，不小心會拖慢整體效能。

## 事件迴圈死結

了解 Drogon 的設計後，應該不難理解如何讓事件迴圈死結。只要在同一迴圈提交遠端 IO 請求並等待結果（同步 API 就是這樣），它會提交 IO 請求然後等 callback。

```
app().registerHandler("/dead_lock", [](const HttpRequest& req
    , std::function<void (const HttpResponsePtr &)> &&callback) {
    auto currentLoop = app().getIOLoops()[app().getCurrentThreadIndex()];
    auto client = HttpClient::newHttpClient("https://drogon.org",
currentLoop);
    auto request = HttpRequest::newHttpRequest();
    auto resp = client->sendRequest(resp); // 死結！呼叫同步介面
});
```

可以這樣想：

```
      Some loop
    .------------.
    | new client |
    | new request|
    |send request|
  .->WAIT resp---+-.
  | |  ....      | |
?| |            | |
?| |------------| |
?| |            | |
?| |            | |
?| |            | |
?| |            | |
  | |            | |
  | |            | |
  | |            | |
  | |            | |
  | |            | |
  | |            | |
  | |------------| |
  | | read resp  | |
  :-+-         <-+-:
    | oops       |
    | deadlock   |
    :------------:
```

其他像 DB、NoSQL callback 也一樣。幸好非 fast DB client 會在自己的執行緒運作，每個 client 都有自己的執行緒，所以在 HTTP handler 裡用同步查詢是安全的。但千萬不要在同一 client 的 callback 裡用同步查詢，否則還是會死結。

> 技巧 3：同步 API 既慢又危險，能不用就不用。如果真的要用，請確保 client 在不同執行緒。

### Fast DB client

Drogon 以效能為優先，易用性其次。Fast DB client 會共用 HTTP 伺服器執行緒，這樣可以省去提交任務到其他執行緒的開銷，也避免 OS 的 context switch。但因此不能用同步查詢，否則會死結事件迴圈。

## 協程救援

Drogon 開發時遇到的兩難是：非同步 API 雖然高效但難用，同步 API 雖然好寫但容易出問題且慢。Lambda 宣告又冗長，語法也不美觀，程式碼不是自上而下而是充滿 callback。同步 API 雖然乾淨，但效能差。

```cpp
// drogon 的非同步 DB API
auto db = app().getDbClient();
db->execSqlAsync("INSERT......", [db, callback](auto result){
    db->execSqlAsync("UPDATE .......", [callback](auto result){
        // 成功處理
    },
    [callback](const DbException& e) {
        // 失敗處理
```

```
    })
},
[callback](const DbException& e){
    // 失敗處理
})
```

對比：

```
// drogon 的同步 API，例外可自動由框架處理
db->execSqlSync("INSERT.....");
db->execSqlSync("UPDATE.....");
```

有沒有辦法兩者兼得？C++20 協程（coroutine）就是答案。它本質上是編譯器支援的 callback 包裝器，讓你的程式碼看起來像同步，但其實全程非同步。以下是協程寫法：

```
co_await db->execSqlCoro("INSERT.....");
co_await db->execSqlCoro("UPDATE.....");
```

看起來就像同步 API，但其實更好。你能享受非同步效能又能用同步語法。協程的原理超出本文範圍，維護者強烈建議能用協程就用（GCC >= 11，MSVC >= 16.25）。但協程不是萬靈丹，還是要注意事件迴圈塞爆和競爭問題。不過用協程寫非同步程式碼更好 debug、也更容易理解。

技巧 4：能用協程就用

## 小結

- 優先用 C++20 協程和 Fast DB 連線
- 同步 API 會拖慢甚至死結事件迴圈
- 如果一定要用同步 API，請確保 client 在不同執行緒