Other languages: 繁體中文

Understanding Drogon's threading model

Drogon is a fast C++ web app framework. It's fast in part by not abstracting the underlying threading model away. However, it also causes some confusions. It's not uncommon to see issues and discussions about why responses are only sent after some blocking call, why calling a blocking networking function on the same event loop blocks causes a deadlock, etc.. This page aims to explain the exact condition that causes them and how to avoid them.

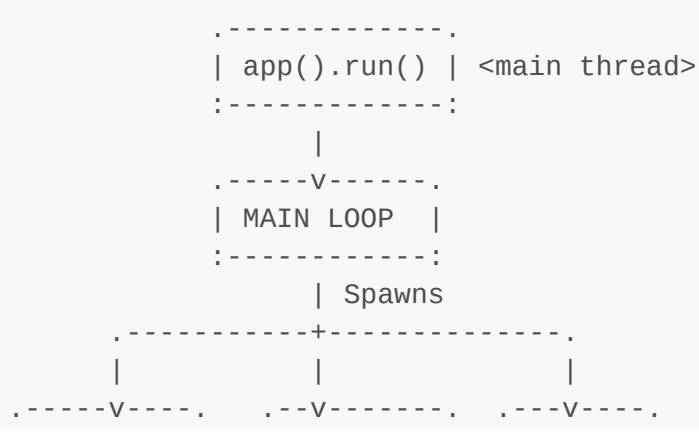
Event loops and threads

Drogon runs on a thread pool where each thread has it's own event loop. And event loops are at the core of Drogon. Every drogon application has at least 2 event loops. A main loop and a worker loop. As long as you are not doing anything funny. The main loop always runs on the main thread (the thread that started `main`). And it is responsible for starting all worker loops. Take the hello world app for example. The line `app().run()` starts the main loop on the main thread. Which in turn spawns 3 worker thread/loop.

```
#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();
}
```

The thread hierarchy looks like this



```
| Worker 1 |   | Worker 2 |  | etc... |
:----------:   :----------:  :--------:
 <thread 1>     <thread 2>   <thread ...>
```

The number of worker loops depends on numerous variables. Namely, how many threads are specified for the HTTP server, how many non-fast DB and NoSQL connections are created - we'll get to fast vs non-fast connections later. Just know that drogon has more threads than just the HTTP server threads. Each event loop is essentially a task queue with the following functionality.

- Reads tasks from a task queue and execute them. You can submit task to run on the a loop from any other threads. Task submitting is totally lock free (thanks to lock free data structure!) and won't cause data race in all circumstances. Event loops process tasks one-by-one. Thus tasks have a well-defined order of execution. But also, tasks that's queued after a huge, long running task gets delayed.
- Listen to and dispatch network events that it manages
- Execute timers when they timeout (usually created by the user)

When non of the above is happening. The event loop/thread blocks and waits for them.

```
// queuing two tasks on the main loop
trantor::EventLoop* loop = app().getLoop();
loop->queueInLoop([]{
    std::cout << "task1: I'm gonna wait for 5s\n";
    std::this_thread::sleep_for(5s);
    std::cout << "task1: hello!\n";
});
loop->queueInLoop([]{
    std::cout << "task2: world!\n";
});
```

Hopefully is's clear why running the above snippet will result in `task1: I'm gonna wait for 5s` appear immediately. Pauses for 5 seconds and then both `task1: hello` and `task2: world!` showing up.

> So tip 1: Don't call blocking IO in the event loop. Other tasks has to wait for that IO.

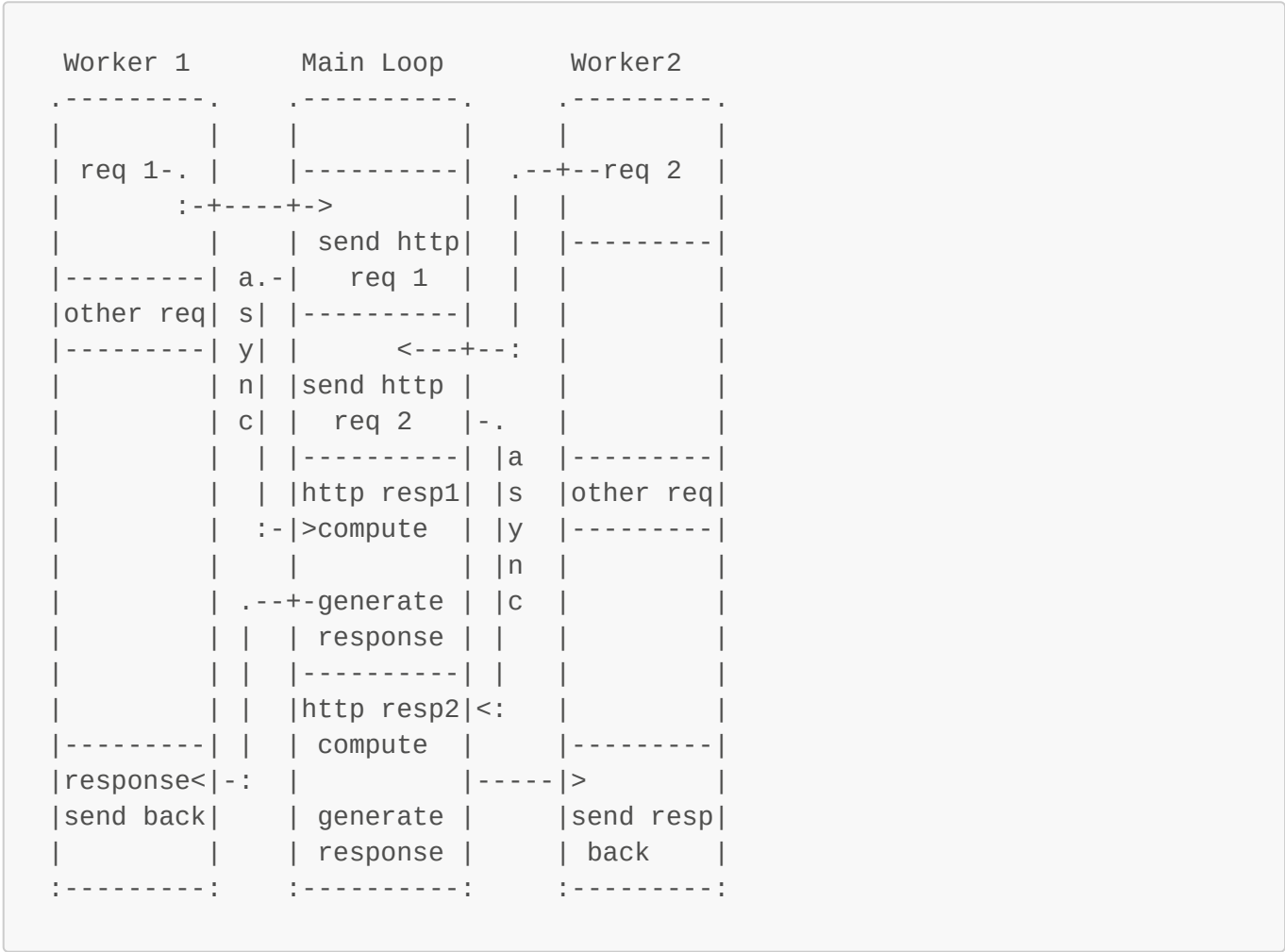
Network IO in practice

Almost everything in drogon is associated with an event loop. This includes the TCP stream/connection, HTTP client, DB clients and data cache. To avoid race conditions, all IO are done in the associated event loop. If an IO call is made from another thread, then parameters are stored and submitted as a task to the appropriate event loop. This has some implications. For example when calling a remote endpoint from within a HTTP handler or making a DB call. The callback from the client may not necessarily (in fact, commonly does not) runs on the same thread as the handler.

```
app().registerHandler("/send_req", [](const HttpRequest& req
    , std::function<void (const HttpResponsePtr &)> &&callback) {
    // This handler will run on one of the HTTP server threads
```

```
    // Create a HTTP client that runs on the main loop
    auto client = HttpClient::newHttpClient("https://drogon.org",
app().getLoop());
    auto request = HttpRequest::newHttpRequest();
    client->sendRequest(request, [](ReqResult result, const HttpResponse&
resp) {
        // This callback runs on the main thread
    });
});
```

Therefor, it is possible to clog up an event loop if you are not aware of what your code is actually doing. Like creating a ton of HTTP clients on the main loop and sending all out-going requests. Or running compute-heavy functions in a DB callback. Stopping other DB queries being processed.



The same principle is also true for the HTTP server. If a response is generated from a separate thread (ex: in a DB callback). Then the response is queue on associated thread for sending instead of sending immediately.

> Tip 2: Be aware where you place your computations. They can also harm throughput if not careful.

Deadlocking the event loop

With the understanding on how Drogon is designed. It shouldn't be hard to see how to deadlock an event loop. You simply submit a remote IO request and wait on it on the same loop - The sync API is exactly that. It submits an IO request and waits for the 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); // DEADLOCK! calling the sync
interface
});
```

Which could be visualized as

The same is true for everything else too. DB, NoSQL callbacks, you name it. Fortunately non-fast DB clients runs on their own threads; each client gets their own thread. Thus it's safe to make synchronous DB queries from a HTTP handler. However you should not run sync DB queries inside the callback of the same client. Otherwise the same thing happens.

> Tip 3: Sync APIs are evil for both performance and safety. Avoid them like the plague. If you have to, make sure you run the client on a separate thread.

Fast DB clients

Drogon is designed for performance first, ease of use kinda second. Fast DB clients are DB clients that shares the HTTP server threads. Theses improves performance by eliminating the need to submit requests to another thread and avoids content switch by the OS. However due to that fact. You cannot make sync queries on them. It'll deadlock the event loop.

Coroutines to the rescue

A dilemma in Drogon's development and use has being that, async APIs are more efficient but annoying to use. While sync APIs can be problematic and slow. But they are easy to program for. Lambda declaration can be long and tedious. And the syntax is just not fun to look at. Besides the code doesn't run top-to-bottom; it's full of callbacks. The sync API is much cleaner than the async once. At the cost of performance.

```
// drogon's async DB API
auto db = app().getDbClient();
db->execSqlAsync("INSERT......", [db, callback](auto result){
    db->execSqlAsync("UPDATE .......", [callback](auto result){
        // Handle success
    },
    [callback](const DbException& e) {
        // handle failure
    })
},
[callback](const DbException& e){
    // handle failure
})
```

versus

```
// drogon's sync API. Exception can be handled automatically by the
framework
db->execSqlSync("INSERT.....");
db->execSqlSync("UPDATE.....");
```

There must be a way to have the cake an eat it too right? Enter C++20's coroutines. Essentially they are a first-class, compiler supported wrapper around callbacks to make your code look like it's synchronous. But in fact async all the way. Here's the same code in coroutines.

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

It's exactly like the sync API! But better in almost every way. You get all the benefit of async but keep using the sync interface. How it actually works if out of the scope of this post. The maintainers urge to use coroutines when you can (GCC >= 11. MSVC >= 16.25). It is however, not quite magic. It does not solve clogging event loops and race conditions for you. However it's easier to debug and understand async code with coroutines.

Tip 4: Use coroutines when you can

Summary

- Use C++20 coroutines and `Fast DB` connections when you can
- Synchronous API can slow down or event deadlock the event loop
- If you have to use a synchronous API. Make sure they are associated with a different thread