Other languages: 繁體中文

Redis

Drogon supports Redis, a very fast, in-memory data store. Which could be used as a database cache or a message broker. Like everything in Drogon, Redis connections are asynchronous. Which ensures Drogon running with very high concurrency even under heavy load.

Redis support depends on the `hiredis` library. Redis support won't be available if hiredis is not available when building Drogon.

Creating a client

Redis clients can be created and retrieve pragmatically through `app()`.

```
app().createRedisClient("127.0.0.1", 6379);
...
// After app.run()
RedisClientPtr redisClient = app().getRedisClient();
```

Redis clients can also be created via the configuration file.

```
    "redis_clients": [
        {
            //name: Name of the client,'default' by default
            //"name":"",
            //host: Server IP, 127.0.0.1 by default
            "host": "127.0.0.1",
            //port: Server port, 6379 by default
            "port": 6379,
            //passwd: '' by default
            "passwd": "",
            //db index: 0 by default
            "db": 0,
            //is_fast: false by default, if it is true, the client is
faster but user can't call any synchronous interface of it and can't use it
outside of the IO threads and the main thread.
            "is_fast": false,
            //number_of_connections: 1 by default, if the 'is_fast' is
true, the number is the number of connections per IO thread, otherwise it
is the total number of all connections.
            "number_of_connections": 1,
            //timeout: -1.0 by default, in seconds, the timeout for
executing a command.
            //zero or negative value means no timeout.
            "timeout": -1.0
        }
    ]
```

Using Redis

execCommandAsync executes Redis commands in an asynchronous manner. It takes at least 3 parameters, the first and second are callback whom will be called when the Redis command succeed or failed. The third being the command it self. The command could be a C-style format string. And the rests are arguments for the format string. For example, to set the key `name` to `drogon`:

```
redisClient->execCommandAsync(
    [](const drogon::nosql::RedisResult &r) {},
    [](const std::exception &err) {
        LOG_ERROR << "something failed!!! " << err.what();
    },
    "set name drogon");
```

Or set `myid` to `587d-4709-86e4`

```
redisClient->execCommandAsync(
    [](const drogon::nosql::RedisResult &r) {},
    [](const std::exception &err) {
        LOG_ERROR << "something failed!!! " << err.what();
    },
    "set myid %s", "587d-4709-86e4");
```

The same execCommandAsync can also retrieve data from Redis.

```
redisClient->execCommandAsync(
    [](const drogon::nosql::RedisResult &r) {
        if (r.type() == RedisResultType::kNil)
            LOG_INFO << "Cannot find variable associated with the key
'name'";
        else
            LOG_INFO << "Name is " << r.asString();
    },
    [](const std::exception &err) {
        LOG_ERROR << "something failed!!! " << err.what();
    },
    "get name");
```

Transaction

Redis transaction allows multiple commands to be executed in a single step. All commands within a transaction are executed in order, no commands by other clients can be executed **in middle** of a transaction. Note that a transaction is not atomic. This means that after receiving the exec command, the transaction will

be executed, If any command in the transaction fails to execute, the rest of the commands will still be executed. redis transactions do not support rollback operations.

The newTransactionAsync method creates a new transaction. Then the transaction could be used just like a normal RedisClient. Finally, the RedisTransaction::execute method executes said transaction.

```
redisClient->newTransactionAsync([](const RedisTransactionPtr &transPtr) {
    transPtr->execCommandAsync(
        [](const drogon::nosql::RedisResult &r) { /* this command works */
},
        [](const std::exception &err) { /* this command failed */ },
    "set name drogon");

    transPtr->execute(
        [](const drogon::nosql::RedisResult &r) { /* transaction worked */
},
        [](const std::exception &err) { /* transaction failed */ });
});
```

Coroutines

Redis clients support coroutines. One should use the GCC 11 or a newer compiler and use `cmake -DCMAKE_CXX_FLAGS="-std=c++20"` to enable it. See the coroutine section for more information.

```
try
{
    auto transaction = co_await redisClient->newTransactionCoro();
    co_await transaction->execCommandCoro("set zzz 123");
    co_await transaction->execCommandCoro("set mening 42");
    co_await transaction->executeCoro();
}
catch(const std::exception& e)
{
    LOG_ERROR << "Redis failed: " << e.what();
}
```

Next: Testing Framework