BetterQuant 量化交易系统的设计和实现

作者：byrnexu（老许） 邮箱：byrnexu@qq.com

一、 前言

近两年来，量化私募领域迎来了蓬勃发展，越来越多的工程师涌入这一领域。在这个行业中，许多初涉者希望了解量化交易系统的设计与实现以及整体架构。由于对于程序的运行的稳定性可扩展性和性能等方面都有着比较高的要求，因此量化交易系统是金融领域属于相对比较复杂的系统。

相较于国外，在国内该领域的起步较晚。量化交易系统的知识相对分散，因此，大部分知识都只能通过在工作中摸索与积累来获取。即便一些同行在量化私募从事研发工作，所涉及的可能也仅限于某个特定模块或者子系统，例如行情接入、交易执行、风险控制、资产核算或者算法交易等等。如果不是公司的核心人员，不大有机会了解系统的全貌。

自 2000 年起，笔者便投身于交易系统的研发工作。曾在国内头部金融 IT 软件和金融数据服务供应商任职，十多年前开始转向量化私募领域。在此期间，主要担任量化私募首席架构师和技术总监等职务，工作范围涵盖量化交易系统及回测平台的研发。由于量化私募公司的 IT 团队通常规模较小，除了部门管理职责外，也需亲力亲为参与一线研发工作。在工作过程中，有时候因为对业务场景的不熟悉，笔者也曾经走了些弯路，同时也积累了一定的经验。相信无论你是刚进入量化行业的新手，还是已经具备一定经验的老手，本文都会对你有一定的帮助。

二、 开源项目 BetterQuant

BetterQuant 是作者独立研发的一套开源的量化交易系统，项目托管地址：https://github.com/byrnexu/betterquant2，由于系统相对比较复杂，所有代码加起来接近八万行，因此自从项目上传到 Github 之后，很多朋友给我的反馈是想了解整个系统却无从着手，基于这个原因，笔者下定决心要写一本量化交易相关的文章，所以这篇文章同时也是 BetterQuant 的设计和实现的说明，反过来 BetterQuant 也是这篇文章对应的项目实践。

以我目前对私募量化这个行业的了解，有些同行的系统仅仅是接入了行情和交易，至于风控和核算，都是业务部门来一个需求就增加一个相应的模块或者功能，可以说都是业务驱动的，笔者刚入的时候就是如此，时间长了，就产生了各种各样的问题，比如整个系统就产生了大量冗余的代码，系统的复杂度增加、扩展性变差、迭代效率低下等等。

这篇文章和 BetterQuant 是笔者交易系统多年研发经验的积累。如果说笔者之前研发的系统是不断的被业务和需求追赶的话，那么 BetterQuant 不再被业务需求驱动和追赶，而是主动覆盖大部分业务场景的系统。

当然这篇文章不仅仅告诉你量化交易系统的设计和实现，通过阅读这篇文章，你还可以学会在开发高性能服务器的时候如何合理的组织服务端代码，使得代码更具可读性和扩展性。

三、 量化交易系统的子系统和模块组成

要了解量化交易系统的子系统和模块组成，我们可以先从需求出发，而需求我们可以从用户的角度出发，量化交易系统的用户主要包括编写策略的金融工程师（以下简称金工）、交易员、风控管理和资产核算人员等等。所以量化交易系统不仅仅是行情和交易的接入和接口，还包括更加复杂和重要的算法交易、风险控制和资产核算等等，因此我设计的量化交易系统也就是 BetterQuant 主要包括以下子系统和模块：

- **公共模块**
  存放公共函数或者公共类的地方。

- **行情网关**
  接收交易所或者交易柜台发送的行情，并转化为 BetterQuant 内部统一格式。

- **交易网关**

  向交易柜台或者交易所发起报单，接收报单响应和回报，将报单转换为交易所格式将回报转换为 BetterQuant 内部统一格式。

- **交易核心**

  系统内单边持仓处理，事前风控等功能。

- **风控子系统**

  事中风控处理，如已实现和未实现盈亏计算等。

- **策略引擎**

  编写交易策略的库，提供了 c++和 python 两种语言的接口。

- **算法交易模块**

  提供算法交易接口，模块内部实现算法交易功能。

- **订单管理模块**

  主要是负责未完结也就是在途订单信息的维护。

- **仓位管理模块**

  最小粒度的仓位信息，通过对最小粒度的仓位信息的汇总，你可以的都各个层面的仓位信息。

以上只是各个子系统和模块的简单说明，后续会对每一个模块结合 BetterQuant 项目展开来详细说明

四、 技术选型

因为我对 c 语言和 c++相对熟悉，所以在去年年底 BetterQuant 写下第一行代码之前，我一直在考虑到底是用 c 语言还是 c++来实现这个系统，c 语言是一门简单且优雅的语言，大名鼎鼎的 linux 内核主要就是 c 语言编写，编译效率高，但是第三方库相对较少，可能要重新造不少轮子。c++语言本身语法比较复杂，很多代码通过模板在编译期展开，编译效率慢，但是相比 c 语言来说第三方库较为丰富。所以虽然 c 语言是一个很好的选择，考虑到开发效率方面的因素，最后还是决定用 c++来开发这套系统。

1. 关于 c++的三个建议

世界上最出色的 C++专家之一，C 缺陷与陷阱、C++沉思录的作者 Andrew Koenig 曾经给中国读者使用 c++的三个建议：

- 避免使用裸指针
- 提倡使用程序库

- 使用类来表示概念

裸指针是 c 语言的东西，相对于 c 语言 c++有更好的选择，那就是智能指针，可能有人会说量化交易系统这种追求微秒级甚至纳秒级的系统里使用智能指针会不会影响运行效率，这个问题可以从几个方面来回答：

第一，BetterQuant 包含完善且通用的事前风控逻辑，所以他不是一个纳秒级而是一个微秒级的 ticker to order 系统，使用智能指针不会造成微秒级的额外开销。

第二，BetterQuant 大部分子系统和模块运行在多线程环境下，所以在很多场景你会发现系统使用 shared_ptr 在线程间传递资源，因为使用 shared_ptr 在多线程环境中传递和共享资源是效率最高的方式之一。

第三：BetterQuant 通过智能指针创建的资源基本上遵循在线程内传引用，在线程间传递指针的拷贝或者移动构造，所以基本上都不存在额外的开销，即使用 c 语言开发多线程应用，如果要实现同样的逻辑，也要手动实现智能指针的引用计数或者释资源等相关的代码逻辑。

第四：个人认为 RAII 是 c++相对 c 的最大优势之一，可以避免很多不小心导致的资源泄露或者死锁等问题。

2. 第三方库

Andrew Koenig 给我们的另一个建议是尽可能的使用第三方库，同时为了避免重复造轮子，BetterQuant 用到了大量的第三方库，主要如下：

- **boost**

  作为 c++的准标准库，boost 中有大量的优秀的框架、数据结构和算法，比如 asio、interprocess、multi-index、algorithm 等等。

- **spdlog**

  日志处理。

- **yaml-cpp**

  配置文件处理。

- **concurrent_queue**

  无锁的异步队列，主要用户在线程间传递数据。

- **fmt**

  字符串格式化库。

- **gflags**

  google 的命令行参数处理库。

- **gtest**

  google 的单元测试框架。

- **magic_enum**
  枚举和字符串转换库。

- **rapidjson**
  json 字符串处理库。

- **xxHash**
  一个轻量且高效的 hash 算法库。

- **unordered_dense**
  基于 robin-hood 和高性能的 hashmap 和 hashset 的实现。

- **iceoryx**
  基于共享内存的进程间传递数据的第三方库，对于 BetterQuant 相对来说重了一些，以后有时间的话会考虑开发一个轻量级的 ipc 库将其替换，新的库会通过配置决定通过共享内存、tcp/ip 或者 domain socket 传递数据。

- **drogon**
  BetterQuant 的 web 服务是通过 drogon 库实现的。

第一次编译系统，BetterQuant 会自动帮你下载并编译第三方库。

3. 数据保存

交易系统在运行过程中会产生各种各样的数据，主要包括行情数据、交易数据、风控的状态数据、策略数据、基础数据等等：

- **行情数据**

为了回测或者其他目的，交易所推送的行情需要保存下来，由于历史行情数据比较大，因此刚入行的时候笔者曾经尝试过自己定义一些数据格式保存历史行情，后来随着时序数据库的普及，历史行情的保存有了更好的选择，比如 influxDB 等等，但是 influxDB 的分布式版本是收费的，相比较而言 TDengine 的分布式版本是免费且开源的，或许是一个更好的选择。

有一点需要注意的是，最好把交易所或者交易柜台收到的行情的原始数据也就是为转换为内部统一格式之前的数据也保存下来，这样即使转换程序有 bug，也可以通过原始数据重新生成内部统一格式的数据。

- **交易数据**

交易数据主要包括委托信息和成交明细，这些数据经常需要统计和汇总，因此关系型数据库就是一个更好的选择，BetterQuant 将这些数据存放在 mysql 中。

在 BetterQuant 中 mysql 仅仅是一个存放数据的地方，整个交易过程都是在内存或者共享内存中进行的，委托和成交信息的入库也是无锁的异步的过程，因此交易数据的保存对于整体性能没有任何影响。

- **风控的状态数据**

风控的状态数据主要包括在风控过程中产生的中间数据，举个例子：在流控中需要统计每分钟的报单次数，那么最近一分钟的已报单数就需要记录下来，另外即使系统重启，风控的状态也要恢复，并且由于流控属于事前风控，也就是发生在报单前，因此风控状态的保存不能影响报单的效率。基于上述原因，基本上所有的风控状态都保存在共享内存中，使用共享内存可以做到即使服务关闭只要不掉电，共享内存中的数据仍然存在。

- **策略数据**

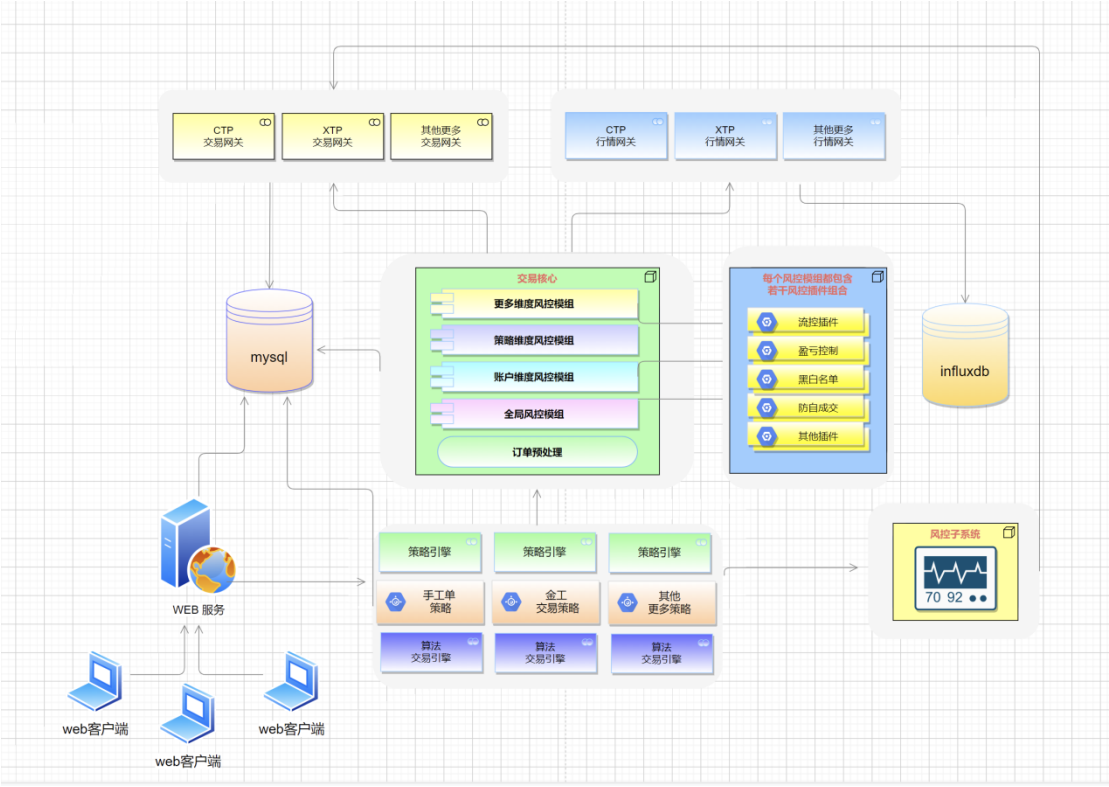
每个策略运行过程中，有时候会需要保存一些自己的数据在下次启动的时候重新加载，这样就需要把这些中间状态的数据保存下来。金工和系统交互的接口也就是策略引擎的 api 提供了这一功能。在 linux 下，可以通过配置文件中指定 tmpfs 挂载的目录保存这些数据，通过 tmpfs 直接和共享内存交互并保存数据，建议使用 json 格式。

- **基础数据**

基础数据主要包括用户信息、账户信息、产品信息、策略组合信息、策略信息、费率信息等等，这些数据同样存放在关系型数据库 mysql 中。

五、 系统架构

1. 系统架构图



2. 订单的生命周期

BetterQuant 的整体架构较为复杂，只是上面这张图肯定没法清晰的表达系统的整体架构是什么样的，最后我想了一个办法就是通过一张订单的生命周期说明下系统是怎么设计的：

1) 交易策略启动后，会根据金工调用的接口订阅各种行情，同时订阅请求会被发往相应的行情网关。

2) 行情网关在收到订阅请求之后，假如行情网关处于全市场订阅的模式（配置中指定），那么不再向交易柜台或者交易所发起订阅，如果不是处于全市场订阅模式，那么会向交易所发起相应标的行情类型的订阅请求，同时行情网关会记录订阅者和订阅的行情信息。

3) 行情网关在收到行情之后，会向订阅该行情的所有交易策略或者子系统（比如风控子系统）推送该行情，交易策略在收到行情之后，触发策略引擎的行情回调函数。同时该消息会通过无锁的异步队列进入行情存储模块，如果行情网关的配置中指定了要保存当前类型的行情，那么该行情会被保存到时序数据库中。

4) 触发策略引擎的行情回调函数之后，开始进入交易策略的内部逻辑，经过交易策略内部逻辑之后，如果确定不需要下单，那么整个流程结束，如果需要下单，那么交易策略会调用下单接口或者算法单接口，同时这张新的订单会被异步入库。

5) 策略调用下单接口之后订单会被发往交易核心，交易核心收到报单请求之后，如果发现是合约报单，会先检查该订单是开仓还是平仓指令，如果发现订单没有指定开平，那么会对该订单做一个预处理的动作，也就是会先检查当前使用的资金账户上是否已经有反向头寸，如果有反向头寸，那么将报单请求设定为平仓动作，这么做的好处是，对于那些不支持单向大边保证金制度或者单边持仓的交易所，可以有效的降低保证金的占用。

6) 随后订单会进入交易核心的事前风控模块，这里会进行诸如流控之类的风控处理，如果订单未通过流控，那么会返回一个废单信息及相应的错误码给交易策略。

7) 如果订单通过了事前风控，交易核心会根据资金账号将该订单发往相应的交易网关。

8) 交易网关在收到下单请求之后，会将下单请求转换为交易柜台或者交易所所可以接受的格式，同时将该请求转发至交易柜台或者交易所。

9) 如果交易所返回报单出错，那么该应答会原路径返回值交易策略，如果没有出错，那么继续等待委托回报，收到委托回报之后，该回报同样会原路径返回至交易策略，同时该订单信息会异步入库，从而更新数据库中该订单的信息。

10) 继续等待回报，直至订单完结。

六、 服务端程序的基本框架

1. 基本框架

假设某一个服务包含模块 ModuleA、ModuleB，其中 ModuleA 又包含 ModuleA1，那么如何组织代码使得系统简洁明了且容易扩展。

通常我会编写一个 MainSvc（实际工作中，这个模块的名称通常是根据服务的功能来命名，比如 TradeSvc 等等），MainSvc 包含两个成员变量也就是 ModuleA 和 ModuleB 的实例 moduleA_和 moduleB_，因为 ModuleA1 是 ModuleA 的成员，所以 ModuleA 有一个 moduleA1_ 的成员。ModuleA 和 ModuleB 分别都包含一个指向 MainSvc 的指针（在 ModuleA 和 ModuleB 的构造函数中初始化），这样通过这个指针，ModuleA 可以调用 MainSvc 的 getModuleB 函数获取 ModuleB，而 ModuleB 可以通过 MainSvc 的 getModuleA 函数获取 ModuleA，MainSvc 就像一个模块之间互相访问的“中介”，所有的模块只需要和它打交道即可和其他模块互动。

首先我们来看看 MainSvc 的代码：

MainSvc.hpp

```
1. #pragma once
2.
3. #include <memory>
4.
5. namespace bq {
6.
7. class ModuleA;
8. using ModuleAUPtr = std::unique_ptr<ModuleA>;
9. class ModuleB;
10.using ModuleBUPtr = std::unique_ptr<ModuleB>;
11.
12.class MainSvc {
13.public:
14.  MainSvc();
15.
16.public:
17.  void run();
18.
19.public:
20.  ModuleAUPtr& getModuleA();
21.  ModuleBUPtr& getModuleB();
22.
23.private:
24.  ModuleAUPtr moduleA_{nullptr};
25.  ModuleBUPtr moduleB_{nullptr};
```

```
};

}
```

MainSvc.cpp

```
#include "MainSvc.hpp"

#include "ModuleB.hpp"
#include "ModuleA.hpp"
#include "ModuleA1.hpp"

namespace bq {

MainSvc::MainSvc() {
  moduleA_ = std::make_unique<ModuleA>(this);
  moduleB_ = std::make_unique<ModuleB>(this);
}

void MainSvc::run() {
  // 获取ModuleA
  auto& moduleA = getModuleA();

  // 获取ModuleB
  auto& moduleB = getModuleB();

  // 获取ModuleA1
  auto& moduleA1 = getModuleA()->getModuleA1();
}

ModuleAUPtr& MainSvc::getModuleA() {
  return moduleA_;
}

ModuleBUPtr& MainSvc::getModuleB() {
  return moduleB_;
}

}
```

ModuleA.hpp

```
#pragma once

#include <memory>
```

```
4.
5. namespace bq {
6.
7. class MainSvc;
8.
9. class ModuleA1;
10.using ModuleA1UPtr = std::unique_ptr<ModuleA1>;
11.
12.class ModuleA {
13.public:
14.  explicit ModuleA(MainSvc* const mainSvc);
15.
16.public:
17.  void start();
18.  void stop();
19.
20.public:
21.  ModuleA1UPtr& getModuleA1();
22.
23.private:
24.  MainSvc* const mainSvc_{nullptr};
25.  ModuleA1UPtr moduleA1_{nullptr};
26.};
27.
28.}
```

ModuleA.cpp

```
1. #include "ModuleA.hpp"
2.
3. #include "ModuleA1.hpp"
4. #include "MainSvc.hpp"
5.
6. namespace bq {
7.
8. ModuleA::ModuleA(MainSvc* const mainSvc)
9.   : mainSvc_(mainSvc)
10.{}
11.
12.void ModuleA::start() {
13.  // 在moduleA中获取moduleB的引用
14.  auto& moduleB = mainSvc_->getModuleB();
15.}
16.
17.void ModuleA::stop() {}
```

```
18.
19.ModuleA1UPtr& ModuleA::getModuleA1() {
20.  return moduleA1_;
21.}
22.
23.}
```

ModuleB.hpp

```
1. #pragma once
2.
3. #include <memory>
4.
5. namespace bq {
6.
7. class MainSvc;
8.
9. class ModuleB {
10.public:
11.  explicit ModuleB(MainSvc* const mainSvc);
12.
13.public:
14.  void start();
15.  void stop();
16.
17.private:
18.  MainSvc* const mainSvc_{nullptr};
19.};
20.
21.}
```

ModuleB.cpp

```
1. #include "ModuleB.hpp"
2.
3. #include "ModuleA.hpp"
4. #include "ModuleA1.hpp"
5. #include "MainSvc.hpp"
6.
7. namespace bq {
8.
9. ModuleB::ModuleB(MainSvc* const mainSvc)
10.  : mainSvc_(mainSvc)
11.{}
12.
```

```
void ModuleB::start() {
  // 在moduleB中获取moduleA的引用
  auto& moduleB = mainSvc_->getModuleA();

  // 在moduleB中获取moduleA1的引用
  auto& moduleA = mainSvc_->getModuleA()->getModuleA1();
}

void ModuleB::stop() {
}

}
```

ModuleA1.hpp

```
#pragma once

namespace bq {
class ModuleA1 {};
}
```

注：你也可以在创建 ModuleA1 实例的时候传入一个 MainSvc 的指针，使得 ModuleA1 也可以通过 MainSvc 和其他模块直接交互。

Main.cpp

```
#include "MainSvc.hpp"
#include "ModuleA.hpp"
#include "ModuleA1.hpp"
#include "ModuleB.hpp"

#include <cstdlib>
#include <cstddef>

int main() {
  auto mainSvc = std::make_unique<bq::MainSvc>();
  mainSvc->run();

  return EXIT_SUCCESS;
}
```

2. 信号处理

上一节的程序只是个示范程序，程序运行进入 run 函数，然后整个进程就直接退出了，作为一个服务端进程，必须能够接受类似 kill -SIGINT pid 这样的命令优雅的退出。这里就需要提一下 Boost 中的 Asio 库了，Boost 的 Asio 是一个网络库，同时也是一个异步任务处理框架，它同时还很好的封装了各个平台的信号处理功能。BetterQuant 利用 Asio 库实现了一个信号处理类 SignalHandler，这样通过 SignalHandler 方便的将信号处理和子模块的退出函数联系起来。源码可参考：

```
1. pub/inc/util/SignalHandler.hpp
2. pub/src/util/SignalHandler.cpp
```

SignalHandler 的使用很简单，public 暴露的接口函数就 4 个其中我们会用到的基本上就是构造和 run 函数：

```
1. class SignalHandler {
2.   using CBSignalHandler =
3.       std::function<void(const boost::system::error_code&, int)>;
4.
5.  public:
6.   SignalHandler(const std::string& moduleName,
7.                 const CBSignalHandler& exitHandler) noexcept;
8.   void run();
```

构造函数就两个入参，moduleName 输出日志时使用，exitHandler 时一个包含两个参数的回调函数，一个参数时错误码，另一个是收到的 signal 编号。SignalHandler 的使用也很简单只需对 MainSvc 这个模块做如下修改即可（灰色底纹部分为修改的代码）：

MainSvc.hpp

```
1. #pragma once
2.
3. #include <memory>
4.
5. namespace bq {
6.
7. class ModuleA;
8. using ModuleAUPtr = std::unique_ptr<ModuleA>;
9. class ModuleB;
10.using ModuleBUPtr = std::unique_ptr<ModuleB>;
11.
12.class SignalHandler;
13.using SignalHandlerUPtr = std::unique_ptr<SignalHandler>;
14.
```

```
15.class MainSvc {
16.public:
17.  MainSvc();
18.
19.public:
20.  void run();
21.
22.private:
23.  void onExit(const boost::system::error_code& e,
24.              int signalNumber);
25.
26.public:
27.  ModuleAUPtr& getModuleA();
28.  ModuleBUPtr& getModuleB();
29.
30.private:
31.  ModuleAUPtr moduleA_{nullptr};
32.  ModuleBUPtr moduleB_{nullptr};
33.
34.  // 增加一个 SignalHandler 对象
35.  SignalHandlerUPtr signalHandler_{nullptr};
36.};
37.
38.}
```

MainSvc.cpp

```
1. #include "MainSvc.hpp"
2.
3. #include "ModuleB.hpp"
4. #include "ModuleA.hpp"
5. #include "ModuleA1.hpp"
6.
7. #include "util/SignalHandler.hpp"
8.
9. namespace bq {
10.
11.MainSvc::MainSvc() {
12.  moduleA_ = std::make_unique<ModuleA>(this);
13.  moduleB_ = std::make_unique<ModuleB>(this);
14.}
15.
16.void MainSvc::run() {
17.  // 获取 ModuleA
18.  auto& moduleA = getModuleA();
```

```
19.
20.   // 获取ModuleB
21.   auto& moduleB = getModuleB();
22.
23.   // 获取ModuleA1
24.   auto& moduleA1 = getModuleA()->getModuleA1();
25.
26.   // 构造signalHandler
27.   const auto exitHandler =
28.       [this] (const boost::system::error_code& e,
29.               int signalNumber) {
30.         onExit(e, signalNumber);
31.       };
32.   signalHandler_ = std::make_unique<SignalHandler>(
33.       "MainSvc", exitHandler);
34.
35.   // 堵塞，直到收到退出信号
36.   signalHandler_->run();
37. }
38.
39. // 收到退出信号后的处理函数
40. void MainSvc::onExit(const boost::system::error_code& e,
41.                       int signalNumber) {
42.   // 退出 moduleA 和 moduleB，执行清理动作
43.   getModuleA()->stop();
44.   getModuleB()->stop();
45. }
46.
47. ModuleAUPtr& MainSvc::getModuleA() {
48.   return moduleA_;
49. }
50.
51. ModuleBUPtr& MainSvc::getModuleB() {
52.   return moduleB_;
53. }
54.
55. }
```

3. SvcBase

因为每个服务启动后都会有一些基本的动作，比如加载配置，初始化日志模块，初始化随机数种子，创建信号处理模块等等，BetterQuant 将所有这些动作都封装在 SvcBase 类中，

实现一个子系统的时候，只需要从 SvcBase 继承即可获得相应的功能。代码参考：

```
1. pub/inc/util/SvcBase.hpp
2. pub/src/util/SvcBase.cpp
```

因为有一些库也用到 SvcBase，这些库不需要处理信号，所以 SvcBase 的构造中有一个枚举类型的参数 InstallSignalHandler 决定是否构造 SignalHandler。另外我个人不怎么喜欢在 public 里暴露虚函数，所以 BetterQuant 中大部分需要子类实现的虚函数都用 template method 模式处理了一下。

七、 公共模块

1. 公共模块的划分

关于公共模块，有的公共模块和业务无关但是整个模块都会用到，也就是可以复用到其他项目，比如日期和浮点数的处理、压缩算法、随机数生成、异步任务分发等等，所有的这些代码都放在公共模块 pub 下。

有一些模块或者定义和业务相关且整个项目都会用到或者说它不属于风控也不属于行情和交易，比如仓位、订单和盈亏这些结构的定义等等，所有的这些模块都放在公共模块 bqpub 下。

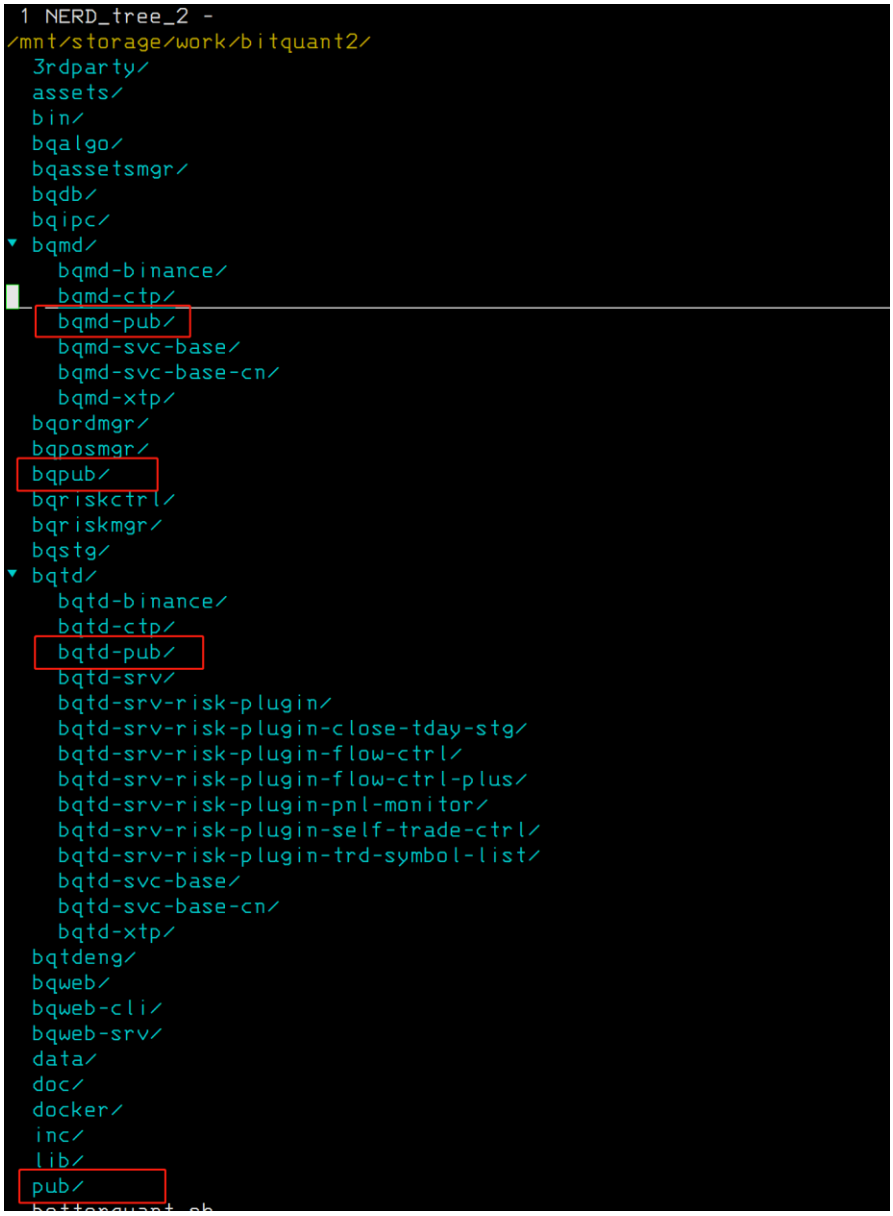
有一些模块或者定义只有在交易网关、交易核心等交易相关的子系统或者模块中用到，他们就会被放在 bqtd-pub 目录下。

有一些模块或者定义只有在行情网关等行情相关的子系统或者模块中用到，他们就会被放在 bqmd-pub 目录下。

有一些模块或者定义只会在当前子系统中用到，那么我就会放在当前子系统的一些公共模块中，以策略引擎为例，视其分类我会放在 StgEngDef.hpp、StgEngConst.hpp 或者 StgEngUtil.hpp 中。

最后还有一种情况就是，假如我有一个字符串算法，开始只是在交易网关中被用到，最初他会被放在 TDSvcUtil.hpp 中，随着项目的进行，后来发现他在交易相关的模块中被用到，那么这个字符串算法就会被“提升”到 bqtd-pub 中，当然如果后来被发现这个算法在行情模块中也会被用到，那么他就会被继续“提升”到 bqpub 中，当然如果是一个通用的算法，其他项目也可以使用，那么它还可以继续被被“提升”到 pub 模块中，这就是目前 BetterQuant 对公共模块的组织方式。

下图就可以看到 BetterQuant 几个基础的公共模块目录的组织方式。



2. 异步任务分发

在高性能服务开发中，我们经常会遇到这样的问题，就是收到一条消息之后会进行一些复杂的业务处理，由于业务处理的时间较长，因此会造成发送方堵塞等问题，这就需要在接收方收到一个消息之后，马上将该消息丢给另一个线程处理，这样就不会对发送和接收的过程产生影响。

解决上述问题的一个办法就是使用异步队列，intel 的 tbb、boost 库中都有相应的实现，笔者将 github 上的 cameron314/concurrentqueue 封装了下实现了一个 TaskDispatcher 类专门用于处理异步任务分发，代码参考：

1. pub/inc/util/TaskDispatcher.hpp

```
54 template <typename Task, BlockType blockType>
55 class TaskDispatcher {
56  using AsyncQueueType = typename std::conditional<
57      blockType == BlockType::Block,
58      moodycamel::BlockingConcurrentQueue<AsyncTaskSPtr<Task>>,
59      moodycamel::ConcurrentQueue<AsyncTaskSPtr<Task>>>::type;
60
61  public:
62   TaskDispatcher(const TaskDispatcher&) = delete;
63   TaskDispatcher& operator=(const TaskDispatcher&) = delete;
64   TaskDispatcher(const TaskDispatcher&&) = delete;
65   TaskDispatcher& operator=(const TaskDispatcher&&) = delete;
66
67   TaskDispatcher(const TaskDispatcherParamSPtr& taskDispatcherParam,
68                  const CBMsgParser<Task>& cbMsgParser,
69                  const CBGetThreadNoForTask<Task>& cbGetThreadNoForTask,
70                  const CBHandleAsyncTask<Task>& cbHandleAsyncTask,
71                  const CBOnThreadStart cbOnThreadStart = nullptr,
72                  const CBOnThreadExit cbOnThreadExit = nullptr)
73       : taskDispatcherParam_(taskDispatcherParam),
74         cbMsgParser_(cbMsgParser),
75         cbGetThreadNoForTask_(cbGetThreadNoForTask),
76         cbHandleAsyncTask_(cbHandleAsyncTask),
77         cbOnThreadStart_(cbOnThreadStart),
78         cbOnThreadExit_(cbOnThreadExit) {}
79
```

TaskDispatcher 是一个模板类，有两个模板参数 Task 和另一个枚举类型的 BlockType，因为不同的业务场景处理的异步处理的任务不一样的，所以这里有一个 Task 类型的模板参数决定了处理何种类型的任务。concurrentqueue 这个库异步处理队列有两种，阻塞的和非阻塞的，BlockType 决定了 TaskDispatcher 内部使用的是阻塞的还是非阻塞的 queue。

另外 TaskDispatcher 的构造函数有 6 个参数，先来看看第一个参数 TaskDispatcherParam：

1. TaskDispatcherParam

1. pub/inc/util/TaskDispatcherParam.hpp

```
20 const static std::string DEFAULT_TASK_DISPATCHER_PARAM =
21     "moduleName=TaskDispatcher; numOfUnprocessedTaskAlert=100; "
22     "maxBulkRecvTaskNumEveryTime=1; timeDurOfWaitForTask=500; "
23     "taskRandAssignedThreadPoolSize=1; taskSpecificThreadPoolSize=4; "
24     "preCreateTaskSpecificThreadPool=0";
25
26 struct TaskDispatcherParam;
27 using TaskDispatcherParamSPtr = std::shared_ptr<TaskDispatcherParam>;
28
29 struct TaskDispatcherParam {
30   TaskDispatcherParam() = default;
31   TaskDispatcherParam(const std::string& moduleName,
32                       std::uint32_t taskRandAssignedThreadPoolSize = 1,
33                       std::uint32_t taskSpecificThreadPoolSize = 4,
34                       std::uint32_t numOfUnprocessedTaskAlert = 100,
35                       std::uint32_t maxBulkRecvTaskNumEveryTime = 1,
36                       std::uint32_t timeDurOfWaitForTask = 1000,
37                       bool preCreateTaskSpecificThreadPool = false);
38
39   std::string moduleName_;
40   std::uint32_t taskRandAssignedThreadPoolSize_{1};
41   std::uint32_t taskSpecificThreadPoolSize_{4};
42   std::uint32_t numOfUnprocessedTaskAlert_{100};
43   std::uint32_t maxBulkRecvTaskNumEveryTime_{1};
44   std::uint32_t timeDurOfWaitForTask_{1000};
45   bool preCreateTaskSpecificThreadPool_{false};
46 };
```

我们可以看到 TaskDispatcherParam 一共有 7 个成员，它每个成员的值是由配置中的字符串覆盖默认的字符串 DEFAULT_TASK_DISPATCHER_PARAM 之后转换而来：

- **moduleName**

有的子系统可能同时会有多个 TaskDispatcher 存在，为了确定打印出来的日志来自于哪个 TaskDispatcher，所以这里传入了一个 moduleName，用于日志输出。

- **taskRandAssignedThreadPoolSize**

  TaskDispatcher 可以有两个线程池，有的任务分发的时候并不需要确保其有序，只需要将其投递到任意一线程中，使得其不阻塞当前线程从而能够被快速的处理，参数 taskRandAssignedThreadPoolSize 就是设定这个异步队列处理线程池的大小。如果将其设置为 0，那么表示不开启这种类型的线程池。

- **taskSpecificThreadPoolSize**

  上面提到了 TaskDispatcher 可以有两个线程池，这是因为有的任务分发的时候会根据任务的一些特征需要其保持有序，比如行情推送的时候同一品种的逐笔成交不能丢到两个线程中处理，否则订阅方收到的可能就是乱序的消息，那么就得将同品种的逐笔成交丢到异步队列的同一个处理线程，taskSpecificThreadPoolSize 就是设定这个异步队列处理线程池的大小。同样如果将其设置为 0，那么表示不开启这种类型的线程池。

- **numOfUnprocessedTaskAlert**

  内存泄露一直是 c/c++的一个问题，当然在经验丰富的工程师眼里这都已经不是问题，但是有另一种比较隐蔽的情形，就是队列堆积导致的内存增长经常被人忽视，numOfUnprocessedTaskAlert 这个参数的就是每当任务堆积是这个参数的整数倍的时候，就会输出一条 warn 级别的日志。

- **maxBulkRecvTaskNumEveryTime**

  常规情况下异步队列接收端接受任务的时候是一个一个接收的，但是当堆积的任务比较多的时候，批量接收效率更好，maxBulkRecvTaskNumEveryTime 就是在接收端也就是 consumer 端每次接收的消息的数量。

- **timeDurOfWaitForTask**

  这个参数决定每隔多久检测一次是否退出异步任务处理。

- **preCreateTaskSpecificThreadPool**

  根据 taskSpecificThreadPoolSize 创建的线程池中的每个线程是动态创建的，如果将 preCreateTaskSpecificThreadPool 置为 true，那么表示在 TaskDispatcher 启动的时候马上创建所有的 taskSpecificThreadPoolSize 数量的线程。

了解了 TaskDispatacherParam 再来看看其他几个参数：

```
28 template <typename Task>
29 using CBMsgParser =
30     std::function<std::tuple<int, AsyncTaskSPtr<Task>>(const Task&)>;
31
32 template <typename Task>
33 using CBGetThreadNoForTask =
34     std::function<std::uint32_t(const AsyncTaskSPtr<Task>&, std::uint32_t)>;
35
36 template <typename Task>
37 using CBHandleAsyncTask = std::function<void(AsyncTaskSPtr<Task>&)>;
38
39 using CBOnThreadStart = std::function<void(std::uint32_t threadNo)>;
40 using CBOnThreadExit = std::function<void(std::uint32_t threadNo)>;
41
```

可以看到剩下的几个参数都是回调函数：

2. cbMsgParser

这个回调函数的返回包含了一个AsyncTaskSPtr<Task>也就是AsyncTask<Task>的智能指针，

```
35 template <typename Task>
36 struct AsyncTask {
37   AsyncTask() = default;
38   explicit AsyncTask(const Task& task, const std::any& arg = std::any())
39       : task_(task), arg_(arg) {}
40   Task task_;
41   std::any arg_;
42 };
43 template <typename Task>
44 using AsyncTaskSPtr = std::shared_ptr<AsyncTask<Task>>;
45
```

可以看到 AsyncTask 是一个包含了 Task 类型和 std::any 类型的成员，cbMsgParser 的主要作用就是需要将 Task 丢到指定线程的时候需要将 Task 的一些特征提取出来，放在一个 std::any 类型的 arg_中，使得后续的 cbGetThreadNoForTask 可以根据 arg 决定需要将其投递到哪个线程。

参数名叫 arg 是有因为受 posix 的函数 pthread_create 启发，c 语言不像 c++可以通过 this 指针和成员变量来传递信息，所以在使用 c 语言编写一些异步接口的时候最好传入一个 void* 参数，使得异步过程触发的时候可以得到一些调用方的信息，当然这是题外话了，因为我们用的是 c++所以这个 arg 的类型就改成了 std::any。

```
2
3 int pthread_create(pthread_t *tidp,
4                    const pthread_attr_t *attr,
5                    void *(*start_rtn)(void*),
6                    void *arg);
7
8
```

3. cbGetThreadNoForTask

前面也提到了这个回调时根据 AsyncTask 中的 arg 决定将 Task 投递到哪个异步处理线程的（也就是 consumer 线程）。

4. cbHandleAsyncTask

处理 AsyncTask 的回调函数。

5. cbOnThreadStart、cbOnThreadExit

启动异步处理线程前和退出异步处理线程前的回调函数。

最后说明下 TaskDispatcher 是通过 dispatch 函数投递消息的：

```
226  public:
227   int dispatch(Task& task) {
```

TaskDispatch 是系统中比较重要的一个模块，几乎所有的异步处理过程都是通过这个模块来处理的。

3. 定时器

通过使用 Boost::Asio，BetterQuant 封装了一个定时器类 Scheduler。

1. pub/inc/util/Scheduler.hpp

```
20 class Scheduler {
21   using Callback = std::function<void()>;
22
23  public:
24   Scheduler(const std::string& moduleName, const Callback& callback,
25             std::uint32_t interval, bool avgInterval = false,
26             std::uint32_t countMax = 0);
27
28  public:
29   int start(const std::string& startTime = "");
30   int stop();
```

可以看到 Scheduler 的构造有四个入参，模块名称（和 TaskDispatcher 中模块名称的作用一样），定时器触发时候的回调函数，定时器触发间隔，avgInterval 表示回调函数的执行时间计算在触发间隔内，也就是确保定时器每隔 interval 毫秒触发一次，countMax 表示定时器触发的总次数。

由于每个 Scheduler 启动时都会创建一个线程，因此如果有比较多的定时任务的时候，就会有线程数过多来的问题，为此 BetterQuant 又实现了一个 SchedulerTaskBundle 用于将多个定时任务整合在一个 Scheduer 中。

2. pub/inc/util/SchedulerTask.hpp
3. pub/src/util/SchedulerTask.cpp

4. 其他

公共模块还包括日志处理、定时器、浮点数处理，并不那么复杂，这里就不一一介绍了。

八、 行情网关

行情网关目前能提供的主流的行情种类包括订单簿（Books）、逐笔成交（Trades）、逐笔委托（Orders）、Tickers 等等，但是需要注意的是并不是每个行情网关都提供上述所有的行情类型。

除了上面常见的行情类型之外，行情网关还提供了 LastPrice 和 Bid1Ask1 两种类型的行情：

提供 LastPrice 的原因是因为有时候只需要用最新价来计算一些东西，但是有的市场最新价来自 Tickers 而有的市场最新价来自 Trades，如果没有 LastPrice，那么为了获得最新价就需要订阅多种类型的行情，在订阅端处理起来比较麻烦，因此就有了 LastPrice，可以通过配置项 createLastPriceByTickers 决定最新价是否通过 Tickers 创建。

另外由于 Books 在很多市场甚至于同一市场他们的档数不一样，还有和 LastPrice 类似，来源也不一样，而很多场景往往只需要一档行情就够了，因此行情网关还提供了仅包含一档行情的 Bid1Ask1 类型的行情，可以通过配置 createBid1Ask1ByTickers 项决定 Bid1Ask1 是否通过 Tickers 创建。

行情网关的结构基本上就和前面提到的服务端程序的基本框架类似，有一点不同的是其公共部分都被剥离出来放在基础模块 bqmd-svc-base-cn 中了，其中-cn 表示是中国大陆市场，因为这个系统设计的目的就是准备在未来会支持更加多样化的市场，比如港股甚至数字货币等其他市场。

1. bqmd-svc-base-cn

服务端程序的基本框架有 MainSvc、ModuleA、ModuleA1 和 ModuleB 这些模块，类似的 bqmd-svc-base-cn 包括以下模块：

- MDSvcOfCN

MDSvcOfCN 相当于前面提到的服务端程序基本框架中的 MainSvc，它的主要作用是封装了每个行情网关初始化、启动和停止都要做的事情，比如加载配置、初始化日志系统、初始化数据库系统、初始化行情订阅管理器、初始化基于共享内存的 IPC 服务端，如果是行情回放模式，那么初始化行情回放相关模块，另外如果不是全订阅模式，那么根据订阅请求向柜台或者交易所发起订阅，否则的话初始化历史行情存储模块，另外还通过 template method 提供了一些在初始化、启动和停止可以做的一些额外的其他事情的接口供子类扩展。

- SHMSrvMsgHandler

目前这个模块主要负责同步过来的订阅信息，根据这个订阅信息更新本地订阅信息。

- RawMDHandler

这个模块主要负责将不同种类的行情分发的不同的线程，这就用到了前面提到的 TaskDispatcher，子类也就是具体的网关在收到原始的行情之后可以调用 RawMDHandler 的 dispatch 将行情交给 TaskDispatcher::dispatch 处理：

```
protected:
 MDSvcOfCN* mdSvc_{nullptr};

 Topic2LastTsGroupSPtr topic2LastExchTsGroup_{nullptr};
 mutable std::mutex mtxTopic2LastExchTsGroup_;

 TaskDispatcherSPtr<RawMDSPtr, BlockType::Block> taskDispatcher_{nullptr};
```

同时在异步处理线程（consumer 线程）中，提供了供扩展的接口在子类中可以通过这些接口将行情统一转换为内部格式：

```
private:
 void handle(RawMDAsyncTaskSPtr& asyncTask);

 virtual void handleNewSymbol(RawMDAsyncTaskSPtr& asyncTask) {}
 virtual bool handleMDTickers(RawMDAsyncTaskSPtr& asyncTask) { return true; }
 virtual bool handleMDTrades(RawMDAsyncTaskSPtr& asyncTask) { return true; }
 virtual bool handleMDOrders(RawMDAsyncTaskSPtr& asyncTask) { return true; }
 virtual bool handleMDBooks(RawMDAsyncTaskSPtr& asyncTask) { return true; }
```

- MDStorageSvc

这个模块将行情写入时序数据库，可以通过配置项 saveTickers、saveTrades、saveBooks、saveOrders 决定是否将特定类型的行情写入数据库。

- MDCache 和 MDPlayback

这两个行情模块分别用于回放，其中 MDCache 用于从时序数据库中读取定量的历史行情，MDPlaryback 从 MDCache 中获取行情并回放，回放的时候可以指定回放的速度。

2. 行情网关的具体实现

目前系统中实现了国内现货（也就是股票）的 xtp 和期货的 ctp 两个接口，分别位于以下两个目录：

1. bqmd/bqmd-xtp
2. bqmd/bqmd-ctp

以 bqmd-xtp 为例，主要包含以下模块：

- MDSvcOfXTP

MDSvcOfCN 的子类，因为 MDSvcOfCN 处理了大部分行情网关在初始化、启动和停止需要做的一些事情，因此 MDSvcOfXTP 基本上就是做了连接 xtp 行情接口的一些初始化、启动和停止以及向行情服务发起和取消订阅请求的实现。

- BQQuoteSpi

继承自 xtp 的 QuoteSpi，主要用于处理收到 xtp 服务端的行情的处理，收到行情之后会将其封装为 RawMD 并交由行情网关基类 RawMDHandler 处理，RawMDHandler 内部会交由 TaskDispatcher::dispatch 处理。

- RawMDHandlerOfXTP

这个模块继承自 RawMDHandler，主要的作用是重载了各种行情处理函数，将行情的格式转换为内部的统一格式。

```
26 class RawMDHandlerOfXTP : public RawMDHandler {
27  public:
28   RawMDHandlerOfXTP(const RawMDHandlerOfXTP&) = delete;
29   RawMDHandlerOfXTP& operator=(const RawMDHandlerOfXTP&) = delete;
30   RawMDHandlerOfXTP(const RawMDHandlerOfXTP&&) = delete;
31   RawMDHandlerOfXTP& operator=(const RawMDHandlerOfXTP&&) = delete;
32
33   using RawMDHandler::RawMDHandler;
34
35  private:
36   std::tuple<int, RawMDAsyncTaskSPtr> makeAsyncTask(
37       const RawMDSPtr& task) final;
38
39  private:
40   void handleNewSymbol(RawMDAsyncTaskSPtr& asyncTask) final;
41   bool handleMDTickers(RawMDAsyncTaskSPtr& asyncTask) final;
42   bool handleMDTrades(RawMDAsyncTaskSPtr& asyncTask) final;
43   bool handleMDOrders(RawMDAsyncTaskSPtr& asyncTask) final;
44   bool handleMDBooks(RawMDAsyncTaskSPtr& asyncTask) final;
45 };
```

ctp 网关的接入类似，这里就不再介绍了。

九、 交易网关

和行情网关类似交易网关的结构也和前面提到的服务端程序的基本框架类似，同样其公共部分被剥离出来放在基础模块 bqtd-svc-base-cn 中了，-cn 表示是中国大陆市场。

1. bqtd-svc-base-cn

bqtd-svc-base-cn 包括以下模块：

- TDSvcOfCN

TDSvcOfCN 同样相当于前面提到的服务端程序基本框架中的 MainSvc，它的主要作用是封装了每个交易网关初始化、启动和停止都要做的事情，比如加载配置、初始化日志系统、初始化数据库系统、初始化行情订阅管理器、初始化基于共享内存的 IPC 服务端、初始化代码表、外部状态码信息、当前账号仓位信息（资金账号和网关是一对一关系）、手续费信息等等。

- TDSrvTaskHandler

前面提到过，一个订单从交易策略出发、经过交易核心再到交易网关，TDSrvTaskHandler 就是交易网关处理来自交易核心的消息，目前处理的消息主要包括下单请求、撤单请求等等。

另外这里有一个比较特殊的消息，就是 MSG_ID_SYNC_UNCLOSED_ORDER_INFO 也就是同步未完结订单消息，我们看代码会发现这个消息不是交易核心发过来的，而是交易网关内部自己产生的，那么这个同步未完结订单之所以不是通过定时器定时发送到交易柜台而是交由 TDSrvTaskHandler 处理，这是为了统一处理流控。

还有，如果交易网关是模拟成交的模式，那么所有的订单信息会交由模拟成交模块 SimedOrderInfoHandler 处理。

- SimedOrderInfoHandler

这是模拟成交模块，这个模块的主要作用是根据模拟下单请求中的扩展字段 SimedTDInfo 生成相应的委托回报。

- TDGateway

这是一个接口类，主要定义了具体的交易网关也就是子类中需要实现的一些接口，比如网关的初始化、下单、撤单和未完结订单的同步等等。

2. 交易网关的具体实现

目前系统中实现了国内现货（也就是股票）的 xtp 和期货的 ctp 两个接口，分别位于以下两个目录：

```
3. bqtd/bqtd-xtp
4. bqtd/bqtd-ctp
```

以 bqtd-xtp 为例，主要包含以下模块：

- TDSvcOfXTP

TDSvcOfCN 的子类，因为 TDSvcOfCN 处理了大部分交易网关在初始化、启动和停止需要做的一些事情，因此 TDSvcOfXTP 就是创建了一个 TDGatewayOfXTP，并将其赋予成员变量 tdGateway_。

- TDGatewayOfXTP

TDGateway 的子类，这个模块主要是负责交易网关的初始化以及下单、撤单和同步未完结订单接口的实现。

- BQTraderSpi

xtp 接口 TraderSpi 的子类，这里主要负责成交回报、委托回报等信息的处理。

十、 策略引擎

策略引擎目前以库的形式提供，包括一系列可调用的接口，用户可以通过调用策略引擎提供的接口创建交易策略，当然你也可以利用这些接口撰写非交易策略，比如一些 pnl 监控策略，所以说策略引擎不仅仅是策略的引擎，同时也是一个功能强大的异步任务并行处理框架。策略引擎除了行情订阅、取消订阅、下单等一系列常见的功能之外，还有很多金工在撰写策略过程中可能会用到的实用接口，具体的接口清单可以参考后面的 策略引擎接口。

对于策略而言，通常都包含若干个参数，每个参数有若干个取值，比如最常见的网格策

略，有价格范围、网格数量、等差 or 等比等几个参数，上述的每个参数也可能会有若干个取值，因此在策略启动之后，肯定会有几套参数同时运行的场景，运行的过程中也可能会对这些参数进行调整，或者说增加一套参数组合，另外对于那些表现不好的参数，也可能会考虑将其关闭。

基于上述理由，BetterQuant 将策略设计为一个策略包含若干个策略实例的形式，每个策略实例对应一套运行参数，为了最大程度的减少策略实例之间对于系统硬件资源的竞争，这些策略实例运行在一个线程池中，线程池的大小可以在策略的配置文件中设置。

策略引擎主要包含以下子模块：

模块名称	模块功能
bqstgengimpl	策略引擎内部业务逻辑真正实现部分
bqstgeng-cxx	用 pimpl 封装的策略引擎的 cxx 的接口
bqstgeng-c	用 pimpl 封装的策略引擎的 c 语言的接口（TODO）
bqstgeng-py	策略引擎 Python 接口

1. bqstgengimpl

bqstgengimpl 包括以下模块：

- StgEngImpl

StgEngImpl 同样相当于前面提到的服务端程序基本框架中的 MainSvc，主要作用是策略引擎的初始化、启动和停止，比如配置的加载、日志系统的初始化、数据库引擎的初始化和启停、策略实例的初始化、在途订单管理模块的初始化、仓位管理模块的初始化、算法单管理模块的初始化和启停、动态 Candle 生成模块的初始化、交易服务 IPC 客户端、风控子系统 IPC 客户端、web 服务 IPC 客户端的创建和连接等等，另外提供了一系列对外接口的实现，如下图所示:

策略引擎接口

- 下单
- 撤单
- 撤销策略所有挂单
- 撤销策略实例所有挂单
- 撤销一个算法单下所有订单
- 创建一个算法单
- 撤销一个算法单
- 订阅
- 取消订阅
- 安装一个固定时间的定时器
- 安装一个间隔执行的定时器
- 移除一个定时器
- 获取订单信息
- 获取策略实例所有未完结订单
- 获取策略实例仓位信息
- 查两个时间点之间历史行情
- 查某时间点之前的历史行情
- 查某时间点之后的历史行情
- 查两个时间点之间历史行情
- 查某时间点之前的历史行情
- 查某时间点之后的历史行情
- 保存策略实例数据
- 加载策略实例数据

回调接口：

策略引擎回调

- 获取策略引擎对象
- 人工干预指令处理
- TOPIC推送
- 委托回报
- 撤单应答
- 逐笔成交
- 逐笔委托
- 订单簿
- Candle
- Tickers
- 策略启动事件
- 策略实例启动事件
- 新增策略实例事件
- 移除策略实例事件
- 策略实例参数变化事件
- 策略实例定时器触发事件
- 账户层面仓位变动信息
- 账户层面仓位快照信息
- 策略层面仓位变动信息
- 策略层面仓位快照信息
- 策略实例层面仓位变动信息
- 策略实例层面仓位快照信息

可以看到 StgEngImpl 是一个相当大的模块。

- StgInstTaskHandlerImpl

  这个模块主要处理外部系统发送过来或者内部定时器等模块发送的各种消息，比如各种行情、报撤单应答、委托回报、策略参数增删改、定时器消息等等。

- DynCandleSvc

  动态 Candle 生成模块，有时候我们需要那种不在整点触发的分钟的整数倍的 Candle，动态 Candle 模块就是用来来生产这样的 Candle，通过指定在什么时间点生成和周期就可以生成你需要的 Candle，这个模块就是实现这样的功能的。

- WebSrvTaskHandler

  来自 web 服务的消息的处理，这里主要处理来自 web 客户端的手工报撤单等请求。

2. bqstgeng-cxx

这是用 pimpl 封装之后的策略引擎的 cxx 的接口，凡是用 cxx 编写交易策略的都使用这个接口。

3. bqstgeng-py

这是利用 boost::python 实现的策略引擎的 python 接口。

十一、 算法交易模块

算法交易模块位于以下目录：

```
1. bqalgo
```

算法交易模块也可以说是策略引擎的一部分，维护通过策略引擎的算法单接口创建的算法单，以下时策略引擎的算法单接口：

```
244
245  public:
246   //! algoName仅仅是一个助记词
247   std::tuple<int, AlgoId> algoOrder(const StgInstInfoSPtr& stgInstInfo,
248                                     const std::string& algoType,
249                                     const std::string& algoName,
250                                     std::uint32_t lifetime,
251                                     const std::string& algoParamsInJsonFmt);
252   int cancelAlgoOrder(AlgoId algoId);
253   std::string getProgressOfAlgoOrder(AlgoId algoId);
254
```

算法交易模块主要包含以下子模块：

- AlgoMgr

AlgoMgr 负责维护系统中创建的算法单，并将策略引擎传递过来的行情和回报投递给相应的算法单。

- AlgoOrder

算法单基类，封装了所有的算法单需要的通用业务逻辑和接口。

- AlgoTWAP

TWAP 算法，很多刚进入量化的同行肯定会觉得类似 TWAP 这样的拆单算法很复杂，其实很多问题本身确实是很复杂的，但是因为已经有很多成熟的解决方案了，所以这些问题也就不能说复杂了。就好比字符串匹配 kmp 算法，第一个实现这个算法的人很厉害，但是后面的人知道原理，再去写一遍也就并不是那么复杂了，个人觉得交易系统中最复杂的还是核

算和风控。

解决类似 TWAP 的拆单算法最简单明了的办法就是用有限状态机，Boost 库中就有现成的高质量的有限状态机实现，以下是 TWAP 算法的状态机定义：

```
// clang-format off
struct transition_table
    : public boost::mpl::vector<
      // ------+-----------------------+-------------------------+-----------------------+-------------------+-------------------+
      //       + Start                 + Event                   + Target                + Action            + Guard             +
      // ------+-----------------------+-------------------------+-----------------------+-------------------+-------------------+
      msmf::Row<StateOfNewOrder         , EventOfOrderSuccess       , StateOfOrderInProc      >,
      msmf::Row<StateOfNewOrder         , EventOfOrderFailed        , StateOfError            >,
      // ------+-----------------------+-------------------------+-----------------------+-------------------+-------------------+
      msmf::Row<StateOfOrderInProc      , EventOfCheckIfCancelOrder , StateOfCancelOrderInProc , ActionOfCancelOrder, GuardOfCancelOrder>,
      msmf::Row<StateOfOrderInProc      , EventOfCancelOrderFailed  , StateOfOrderInProc      >,
      msmf::Row<StateOfOrderInProc      , EventOfOrderRetFilled     , StateOfSuccess          >,
      msmf::Row<StateOfOrderInProc      , EventOfOrderRetFailed     , StateOfError            >,
      // ------+-----------------------+-------------------------+-----------------------+-------------------+-------------------+
      msmf::Row<StateOfCancelOrderInProc, EventOfCancelOrderRetFailed , StateOfOrderInProc      >,
      msmf::Row<StateOfCancelOrderInProc, EventOfOrderRetFilled     , StateOfSuccess          >,
      msmf::Row<StateOfCancelOrderInProc, EventOfOrderRetFailed     , StateOfError            >,
      msmf::Row<StateOfCancelOrderInProc, EventOfOrderRetCanceled   , StateOfRetryOrder       >,  // 人工撤单会触发此事件从而导致 StateOfOrd
      msmf::Row<StateOfCancelOrderInProc, EventOfOrderSizeLTTickerSize, StateOfSuccess        >,  // 人工撤单会触发此事件从而导致 StateOfOrd
      // ------+-----------------------+-------------------------+-----------------------+-------------------+-------------------+
      msmf::Row<StateOfRetryOrder       , EventOfOrderSuccess       , StateOfOrderInProc      >,
      msmf::Row<StateOfRetryOrder       , EventOfOrderFailed        , StateOfError            >,
      // ------+-----------------------+-------------------------+-----------------------+-------------------+-------------------+
      msmf::Row<StateOfError            , EventOfCheckIfRestoreOrder  , StateOfRestoreOrder     , ActionOfRestoreOrder, GuardOfRestoreOrder>,
      // ------+-----------------------+-------------------------+-----------------------+-------------------+-------------------+
      msmf::Row<StateOfRestoreOrder     , EventOfOrderSuccess       , StateOfOrderInProc      >,
      msmf::Row<StateOfRestoreOrder     , EventOfOrderFailed        , StateOfError            >
      // ------+-----------------------+-------------------------+-----------------------+-------------------+-------------------+
      > {};
// clang-format on
```

- AlgoSmartOrder
  SmartOrder 算法。

十二、 交易核心

交易核心这个模块位于：

```
1. bqtd/bqtd-srv
```

交易核心的功能是对订单进行预处理，关于预处理，前面也解释过了，就是对于那些交易期货没有指定开平的报单的请求，先检查这个报单请求使用的资金账号上是否有反向头寸，如果有反向头寸的话，直接将其设定为平仓方向的指令，这么做的主要作用是，对于那些不支持单边持仓或者单向大边保证金制度的交易所，可以有效的降低保证金占用。

除了订单预处理之外，交易核心的最重要的功能事前风控的处理，关于事前风控，这里就需要提一个叫维度的概念，举个例子：假如一个风控是一个策略在单位时间内的报单次数，那么这里的维度就是策略层面，还有一些维度可能会包含多个字段，比如说一个策略下某一个资金账号每分钟的报单次数，那么这里的维度就是就是策略编号加上资金账号，BetterQuant 一共有多达十多个字段可供选择，包括：产品组合、产品、用户、账号组合、账号、交易账号、策略组合、策略、策略实例、算法单、市场、代码类型、代码、买卖、多空等等，这些字段或者字段的组合就构成了维度。

除了维度之外，作为事前风控还有各种不同的风控指标，比如说交易的黑白名单、盈亏监控、自成交监控、流控等等，维度加上风控指标就构成了各种各样的风控。比如策略层面的黑白名单，或者说策略下某个资金账号交易时的黑白名单。

在进入事前风控以后，为了避免竞争，通常的做法是会加入一把全局锁锁定整个过程，但是这样风控过程就会退化为单线程，但是有的风控，比如控制每个资金账号每分钟的报单

次数，这种情况如果有多个资金账号，那么这个过程就可以以资金账号为单位放在线程池里并发进行，这样就会提升风控的效率，当然这个例子中流控处理的业务不是很复杂，有可能单线程处理更快，这里只是举个例子。

接下来我们再来介绍一个概念，就是风控模组，维度加上一个或者多个风控指标就构成了风控模组，因为每个风控模组的维度是一样的，所以每个风控模组运行在一个线程池中，具备同样的线程粒度，在一个系统同时会存在多个模组， BetterQuant 将风控指标设计为一个个动态链接库的插件，这样每个风控模组就像一个个容器，在配置中定义好容器的维度，然后放入你需要的风控指标也就是插件即可，通过在配置中配置一个或者多个风控模组，这样整个事前风控就通过配置动态生成，实现了高性能且灵活的事前风控。

交易核心主要包含以下模块：

- TDSrv

  TDSrv 模块主要负责在启动时加载配置、初始化数据库引擎、从数据库获取风控需要的一些配置、比如黑白名单、自成交控制范围等等、另外根据配置初始化各个维度的风控模组，创建和初始化订单预处理模块等等。

- OrderPreProc

  这个模块主要负责前面提到的订单的预处理。

- RiskCtrlModule

  这是风控模组模块，主要负责启动的时候根据配置创建和初始化风控模组，包括加载各种风控插件，初始化各个维度的风控状态数据等等。

- RiskCtrlConfMonitor

  这个模块主要负责监控风控插件相关配置的变化。

- StgEngTaskHandler

  这个模块主要负责来自策略引擎的消息的处理，比如下单和撤单等等。

- TDGWTaskHandler

  这个模块主要负责交易网关的消息的处理，比如下单应答、撤单应答和委托回报等等。

- PosMgrRestore

  这个模块主要负责在盘中交易核心关闭重启后的仓位重建的动作。

十三、 风控子系统

在 BetterQuant 中，风控子系统是处理事中风控的，因为事前风控必须在报单前发起，所以事前风控只能放在报单过程中，也就是只能在策略引擎、交易核心或者交易网关中，由于很多风控都是跨策略和账号的，所以事前风控就放在了交易核心的风控模组中，事中风控也可以放在交易核心中，但是为了不影响保单效率，BetterQuant 将事中风控都剥离出来，

放在了一个独立的风控子系统中。

风控子系统主要有以下模块：

- RiskMgr
  就如例子中的 MainSvc 一样，RiskMgr 主要负责各种模块的初始化和启停。

- StgEngTaskHandler
  这个模块主要负责来自策略引擎的消息的处理，比如下单和撤单等等。

- TDGWTaskHandler
  这个模块主要负责交易网关的消息的处理，比如下单应答、撤单应答和委托回报等等。

- PubSvc
  目前的主要功能是仓位快照的计算和发布，其他子系统可以通过订阅接口订阅仓位快照信息，通过仓位快照信息可以计算出盈亏情况。

十四、 订单管理

从业务层面来说，订单管理是非常重要的一个模块，设计合理的订单管理模块，使得我们可以准确且高效的计算被冻结的资金和仓位，这些数据可以为风控、报单和回报等功能提供强有力的支撑。订单管理模块位于目录：

```
1. bqordmgr
```

因为订单管理模块被应用在策略引擎、交易核心、交易网关的报单过程中，所以对仓位管理模块的性能有一定的要求，仓位管理模块是用 boost::multi-index 实现的，boost::multi-index 是一个支持多个索引的数据结构，他甚至可以和 boost::interprocess 结合，在共享内存上创建一个支持多个索引的数据结构。

由于订单管理模块在多个子系统中会被用到，每个子系统需要的索引都不一样，为了确保性能，BetterQuant 用变长模板参数这个语言特性，使得你可以在使用的时候动态指定需要的索引：

```
87 //! StgInstId 层面索引
88 struct TagStgInstIdOfOM {};
89 struct KeyStgInstIdOfOM
90     : boost::multi_index::composite_key<
91           OrderInfo, MIDX_MEMBER(OrderInfo, ProductId, productId_),
92           MIDX_MEMBER(OrderInfo, UserId, userId_),
93           MIDX_MEMBER(OrderInfo, StgId, stgId_),
94           MIDX_MEMBER(OrderInfo, StgInstId, stgInstId_)> {};
95 using MIdxStgInstIdOfOM = boost::multi_index::ordered_non_unique<
96     boost::multi_index::tag<TagStgInstIdOfOM>, KeyStgInstIdOfOM,
97     boost::multi_index::composite_key_result_less<
98         KeyStgInstIdOfOM ::result_type>>;
99
100 //! ClosedTime索引
101 struct TagClosedTimeOfOM {};
102 struct KeyClosedTimeOfOM
103     : boost::multi_index::composite_key<
104           OrderInfo, MIDX_MEMBER(OrderInfo, std::uint64_t, closedTime_)> {};
105 using MIdxClosedTimeOfOM = boost::multi_index::ordered_non_unique<
106     boost::multi_index::tag<TagClosedTimeOfOM>, KeyClosedTimeOfOM,
107     boost::multi_index::composite_key_result_less<
108         KeyClosedTimeOfOM ::result_type>>;
109
110 //! AlgoId索引
111 struct TagAlgoIdOfOM {};
112 struct KeyAlgoIdOfOM : boost::multi_index::composite_key<
113                            OrderInfo, MIDX_MEMBER(OrderInfo, AlgoId, algoId_)> {
114 };
115 using MIdxAlgoIdOfOM = boost::multi_index::ordered_non_unique<
116     boost::multi_index::tag<TagAlgoIdOfOM>, KeyAlgoIdOfOM,
117     boost::multi_index::composite_key_result_less<KeyAlgoIdOfOM ::result_type>>;
118
119 template <typename... IndexTypes>
120 class OrdMgr {
121   //! 用于存放在途订单的OrderInfoGroup
122   using OrderInfoGroup = boost::multi_index::multi_index_container<
```

比如在策略引擎中指定了四个索引：

```
19 namespace bq {
20
21 using StgOrdMgr = OrdMgr<MIdxOrderIdOfOM, MIdxMarketCodeExchOrderIdOfOM,
22                           MIdxStgInstIdOfOM, MIdxAlgoIdOfOM>;
23 using StgOrdMgrSPtr = std::shared_ptr<StgOrdMgr>;
24
25 using StgPosMgr = PosMgr<MIdxMainOfPM, MIdxStgInstIdOfPM>;
26 using StgPosMgrSPtr = std::shared_ptr<StgPosMgr>;
27
28 }  // namespace bq
29
30 namespace bq::stg {
```

订单管理的主要功能是负责未完结订单和部分最近完结订单的维护、修改和删除、订单信息查询等功能。

十五、 仓位管理

仓位管理同样是非常重要的一个模块，风控、报单和回报等功能都需要详细的仓位信息：

```
2. bqposmgr
```

因为仓位管理模块被应用在策略引擎、交易核心、交易网关的报单过程中，所以对仓位管理模块的性能同样有一定的要求，和订单管理一样仓位管理模块也是用 boost::multi-index 实现的。

仓位管理模块同样在多个子系统中会被用到，每个子系统需要的索引都不一样，为了确保性能，BetterQuant 用变长模板参数这个语言特性，使得你可以在使用的时候动态指定需要的索引，比如在策略引擎中指定了两个索引：

```
namespace bq {

using StgOrdMgr = OrdMgr<MIdxOrderIdOfOM, MIdxMarketCodeExchOrderIdOfOM,
                         MIdxStgInstIdOfOM, MIdxAlgoIdOfOM>;
using StgOrdMgrSPtr = std::shared_ptr<StgOrdMgr>;

using StgPosMgr = PosMgr<MIdxMainOfPM, MIdxStgInstIdOfPM>;
using StgPosMgrSPtr = std::shared_ptr<StgPosMgr>;

}  // namespace bq
```

仓位管理的主要功能是负责仓位数量的维护、持仓均价、已实现盈亏的计算等等。

十六、 其他（TODO）

1. 账户体系

2. 仓位重建

3. 算法交易和策略引擎的交互

4. 处理外部（交易所/柜台）数据异常

5. 统一错误处理

6. 编写缓存友好的程序

7. IPC 模块的设计

8. 订单状态处理

9. 已实现和未实现盈亏的计算

10. 风控插件的设计