第 2 章 系统开发约定

2.1 开发能力要求

如果只是希望了解证券交易系统主要结构，不参与具体开发工作，那么最基本的要求是学过 C 语言，并了解面向对象编程，这样就能读懂大部分代码并理解系统架构原理。

如果要进行具体开发工作，则需要掌握 C++或 Java 编程，C#也可以考虑但不是很推荐，并具有项目开发经验。对于 Python 或 JavaScript 编程不做硬性要求，可以现学现用。对数据结构、常用算法、网络通信、设计模式、多线程编程、数据库编程等知识技能则是多多益善。本书讲解的项目实例以 C++和 Python 为主要开发语言。最好对 Windows 系统和 Linux 系统有一定了解，会使用 Visual Studio、Visual Studio Code、Eclipse、CMake、SVN、GIT 等开发环境和管理工具。

基本的证券投资知识也是需要的，最好有实际证券交易经验或有空考一下证券期货基金从业资格。如果已有一些适合用程序来做交易的投资策略，带着这样的需求和目的，对于尝试开发一套投资者交易终端，定会有事半功倍的效果。

2.2 基本开发原则

尽量写简单的大家都能看懂的代码。

少用语法糖，降低复杂性，尽量不用“旁门左道”和“独门秘技”。大型系统需要多人协作共同开发维护，不需要在代码上展现个人才艺，对于实时交易系统，更需要在发生异常时能争分夺秒地进行应急修复。晦涩的代码不容易发现缺陷，不利于及时修正，会极大地增加资金损失的风险。

从需求出发对系统各模块进行适当解耦。

就像风险和收益一样，时间与空间、通用与个性、兼容与效率，永远是需要根据需求做权衡取舍而无法兼得的。比如开发投资者交易终端，如果需求是日级到秒级的交易，那就可以在效率上做最大妥协，尽可能增加系统通用性和兼容性，使用网络通讯和服务进程的方式，最大化解耦系统模块；如果是毫秒级交易，最好将各系统模块合并到一个进程，考虑取消多用户模式，适当减少多策略和多账户产生的性能开销，适当降低策略计算复杂

度；如果是纳秒必争的微秒级交易，那就需要对所有模块做最大精简和优化，高度集成高度耦合，并对操作系统、服务器软硬件、网络传输延迟等做全面彻底优化，专用专享，只为极致。

低耦合重在系统架构，高耦合重在系统优化，不考虑优化的话，做减法总归比做加法容易，所以对于需求多样的投资者交易终端，本书先用低耦合通用型架构做全面讲解，然后再探讨高耦合专用型架构下的性能优化，对于券商柜台系统和交易所交易系统，本书中的实现则优先追求效率和必要功能，其次考虑通用和附加扩展。代码复用上，尽量争取各系统模块像积木一样，简单修改以后可以在不同架构模式下使用。

根据应用场景灵活使用合适的开发语言。

“PHP 是世界上最好的语言”，这句话应该耳熟能详。在各类技术社群和论坛里，各种编程语言孰优孰劣也像中医和西医一样每次都能群起而争之，以至于上升到哲学和信仰的高度。但一个经验丰富的开发人员应该对当前主流的编程语言及其特性有较为全面和深入的了解，只有入行不久的新人才会觉得自己掌握了一门语言很了不起并抓着不放。

就像做一张桌子是用木头还是塑料还是不锈钢一样，用什么语言开发项目，也是根据需求场景来选择的。C、Java、Python、C++、C#、JavaScript、PHP、SQL、R、Swift、Go、Matlab 等这些语言之所以能同时成为主流，就是因为各自有不同的特性可以满足不同方向的需求，社区生态、可用类库、运行速度、并行能力、开发效率、跨平台性、封装层次、文档示例等等，合适的才是最好的，能融会贯通灵活使用才是最棒的。框架选型、架构选型、模式选型，也都是如此，懂得拿捏取舍方可为领悟。

做与不做是个挺重要的问题。

人生短短几十年，做任何一件事情都是有机会成本的，一个时间段内只能选择做一件事，同时要放弃其他千千万万件可以选择的事。假设出生条件和周围环境都相同，那么一个人之所以能够发展得比同样条件的其他人更有优势，我想主要在于这个人在人生的时间段里选择做了更有价值更有助于成长的事。

选择大公司还是小公司，两者所接触到的工作内容肯定是不一样的。为公司为团队还是为个人开发交易系统，其功能需求和可靠级别也是不一样的。一个投资团队是选择长期专注于一个投资领域还是打一枪换一个地方只要有机会就上，对团队对个人的收益和成长也肯定会不同。

很多用惯 C++的人在使用 Python 以后都会有种人生苦短相见恨晚的感叹，毕竟大家的时间精力都有限。构建系统的时候，是满足普罗大众的基础投资需求，还是致力于更高层次的专业投资技术；个人闲暇的时候，是接外包私活赚钱养家糊口，还是多学习多研究给自己充电。每个人的条件和发展不尽相同，做与不做，是个挺重要的问题。

2.3 代码风格约定

2.3.1 通用代码风格

统一的缩进和换行，统一的字符编码。
代码的注释说明尽量详尽清晰，方便他人维护。
在整个项目解决方案中，使用一致的代码风格编写代码。
做好异常处理，消除编译警告，测试环境和生产环境不混淆。
明确变量生命周期，注意变量赋值类型，就近定义，减少内存拷贝。
变量、函数、类、模块等尽量使用描述性命名，能够望词生意，便于理解。
有利于避免变量空值、访问越界、意外改值、线程竞争、条件遗漏等隐蔽问题。

2.3.2 C++ 代码风格

---------- demo_class.h ----------

```
/* copyright notice */

#ifndef PROJECT_MODULE_DEMO_CLASS_H
#define PROJECT_MODULE_DEMO_CLASS_H

#include <string> // 1 标准库头文件

#include <mysql.h> // 2 第三方库头文件

#include <common/sysdef.h> // 3 开发者关联项目头文件

#include "global/define.h" // 4 本项目其他模块头文件

#include "trader.h" // 5 本模块其他头文件

#ifdef __OS_WINDOWS__
#define PROJECT_MODULE_EXPIMP __declspec(dllexport)
#endif

#ifdef __OS_LINUX__
#define PROJECT_MODULE_EXPIMP __attribute__((visibility("default")))
#endif
```

```
namespace project {

    #define DEF_FLAG “flag_xyz”
    #define DEF_SIZE 1234567890

    typedef std::shared_ptr<std::thread> thread_ptr;

#pragma pack( push )
#pragma pack( 1 )

    struct DemoStruct {
        int32_t m_year;
        int64_t m_large;
        std::string m_name;

        DemoStruct( int32_t year, int64_t large, std::string name = “” )
            : m_year( year )
            , m_large( large )
            , m_name( name ) {
        }
    };

#pragma pack( pop )

    class PROJECT_MODULE_EXPIMP Client_X {
    public:
        Client_X() {
        }

        virtual ~Client_X() {
        }

    public:
        virtual void OnClientInfo( int32_t info_type ) = 0;
    };

    class OtherClass; // declare
```

```
    class PROJECT_MODULE_EXPIMP DemoClass : public Client_X {
     public:
          DemoClass();
          ~DemoClass();

     public:
          static DemoClass* GetInstance();

     public:
          bool ReadConfig( std::string file_path );

     public:
          void OnClientInfo( int32_t info_type ) override;

     public:
         bool m_connect_ok;
         std::vector<DemoStruct> m_vec_demo_struct;
         std::map<size_t, DemoStruct*> m_map_demo_struct;

     private:
          OtherClass* m_other_class;
          static DemoClass* m_instance;
     };

} // namespace project

#endif // PROJECT_MODULE_DEMO_CLASS_H
```

---------- demo_class.cpp ----------

```
/* copyright notice */

#include <other_head_files.h>

#include "demo_class.h"

namespace project {
```

```
DemoClass* DemoClass::m_instance = nullptr;

DemoClass::DemoClass()
    : Client_X()
    , m_connect_ok( false )
    , m_other_class( nullptr ) {
    try {
        m_other_class = new OtherClass();
    }
    catch( ... ) {
    }
    m_instance = this;
}

DemoClass::~DemoClass() {
    m_vec_demo_struct.clear();
    for( auto it_ds = m_map_demo_struct.begin();
        it_ds != m_map_demo_struct.end(); it_ds++ ) {
        delete it_ds->second;
    }
    m_map_demo_struct.clear();
    if( m_other_class != nullptr ) {
        delete m_other_class;
        m_other_class = nullptr;
    }
}

DemoClass* DemoClass::GetInstance() {
    return m_instance;
}

bool DemoClass::ReadConfig( std::string file_path ) {
    // TODO: read config file
    for( size_t i = 0; i < 10; i++ ) {
        DemoStruct* demo_struct = new DemoStruct( (int32_t)i, 123, “name”);
        m_map_demo_struct[i] = demo_struct;
        m_vec_demo_struct.push_back( *demo_struct );
```

```
        }
        return true;
    }

    void DemoClass::OnClientInfo( int32_t info_type ) {
        // TODO: deal with client info
    }

} // namespace project
```

2.3.3 Python 代码风格

---------- trader.py ----------

```
# -*- coding: utf-8 -*-

/* copyright notice */

from datetime import datetime # 1 standard library

from pubsub import pub # 2 third party library

import logger # 3 other module

class Trader():
    class Order(object):
        def __init__(self, **kwargs):
            self.order_id = "" # 委托编号
            self.instrument = kwargs.get("instrument", "") # 合约代码

        def ToString(self):
            return "order_id: %s, " % self.order_id + \
                   "instrument: %s" % self.instrument

    def __init__(self, parent, now_time):
        self.parent = parent
        self.now_time = now_time
        self.started = False
        self.task_dict = {}
```

```
        self.logger = logger.Logger()
        order = self.Order(instrument = "IF2006")
        print("order: ", order.ToString())

    def __del__(self):
        self.started = False

    def IsTraderReady(self):
        return self.started == True

def GetTime():
    return datetime.now().strftime("%H:%M:%S")

if __name__ == "__main__":
    import sys
    from PyQt5.QtWidgets import QApplication
    app = QApplication(sys.argv)
    trader = Trader(None, GetTime())
    print("now time: ", trader.now_time)
    sys.exit(app.exec_())
```

2.3.4 其他语言风格

对于 Java、C#、JavaScript、SQL 等开发语言，这里不再详细列举，可以根据上文所示的 C++、Python 代码风格做一些调整，也可以参考 Google[01]、Mozilla[02]等大公司以及一些知名开源项目[03][04][05]使用的代码规范。个人项目依从自己的习惯保持一种风格，团队项目与其他成员协商确定整体风格，原则上做到统一、清晰、明确、方便阅读，易于维护即可。

2.4 开发环境准备

如果准备参与证券交易系统开发，那么最好能掌握以下开发环境的使用。

操作系统推荐使用 64 位 Windows 10 专业版或企业版、Windows Server 2019 标准版或数据中心版、64 位 CentOS 7 或 8。其他 Linux 发行版本如 RHEL、Ubuntu、SUSE、Debian、Fedora 等也都可以，看个人喜好和熟悉程度。因为不能适配 Unicode 字符编码等问题，不再使用 Windows 7 和 Windows Server 2008 甚至更古老的 Windows 操作系统，也不再使用 32 位操作系统。后期分发部署可以考虑 Docker 等容器环境。

C++开发在 Windows 下主要使用 Visual Studio 2017 或 2019，在 Linux 下主要使用 Visual Studio Code，当然 Windows 下也可以使用 Visual Studio Code，只是功能和便捷方面 Visual Studio 更胜一筹。有了 Visual Studio Code 那么 Linux 下就不推荐 Qt Creator[06]、CodeLite[07]、CodeBlocks[08]、Eclipse[09]等 IDE 来做 C++开发了，Vim 和 Emacs 就更不提了。

Visual Studio 推荐安装 Qt Visual Studio Tools、Indent Guides、VSColorOutput、ForceUTF8(No BOM)、Productivity Power Tools 2017/2019、DevExpress、Visual Assist 等扩展。Visual Studio Code 推荐安装 C/C++、C#、CMake、Code Runner、Guides、Output Colorizer、Python、Vetur、vscode-icons 等插件。

C++的跨平台项目构建使用 CMake[10]，需要掌握 CMake 的语法和 CMakeLists.txt 文件的编写，要会使用 OPTION、IF-ELSE-ENDIF、FILE、INCLUDE、MESSAGE、FIND_PACKAGE、INCLUDE_DIRECTORIES、LINK_DIRECTORIES、SET、ADD_SUBDIRECTORY、SOURCE_GROUP、ADD_EXECUTABLE、TARGET_LINK_LIBRARIES、MAKE_DIRECTORY、INSTALL 等常用关键字，能够设置各类路径，构建复杂项目结构、判断编译环境、编译器优化配置、外部链接库引用、可执行程序生成、动态静态链接库生成、查找配置诸如 Boost[11]和 Qt[06]等功能库、目标程序和关联文件的安装部署等。

C++常用的第三方库，像组件库 Boost，压缩解压 ZLib、LzmaLib，界面库 Qt、DuiLib、wxWidgets，序列化 RapidJson、JsonCpp、ProtoBuf、Thrift、MsgPack 等，数据库接口 MariaDB、MySQL、SQLite、Mongo、LevelDB、BerkeleyDB 等，测试框架 GoogleTest，加密库 OpenSSL、CryptoPP，脚本嵌入 Node、V8、Lua，金融相关 QuantLib、QuickFix，并发优化 Intel TBB，等等，可以先编译好，在开发时调用。因为有些行情或交易接口只有 32 位没有 64 位，所以第三方库仍需 32 位和 64 位各编译一份，Linux 下有时为了部署方便会使用全静态编译，所以第三方库也是动态和静态各编译一份。

Python 开发主要使用 Eclipse 加 PyDev[12]插件，有时也会用 Visual Studio Code 或自带的 IDLE 做简单的单文件编写调试，PyCharm[13]只是在做数值计算开发时用一下。最好能根据自己的开发需求确定所需的功能库，逐一通过 pip 命令或直接下载安装包的方式部署 Python 环境，而不是使用 Anaconda[14]这种整合式安装包。目前 2.7 版本的 Python 已不再使用，要求 3.6 及以上版本，日常使用中两者的差别主要体现在 print 函数、字符串编码、除法小数位等方面，而 3.X 各版本的选择主要根据所需最新特性和功能库是否支持。对 Python 语言未来发展感兴趣的可以关注 Python 的 PEP 文档[15]。

使用 lxml、xlrd、xlwt 等读写数据，使用 urllib、requests、beautifulsoup 等编写简单的网络爬虫，使用 NumPy、SciPy、Pandas、Matplotlib 等进行数据分析，使用 PyQt 开发用户界面，使用 PyMySQL、PyMsSQL、PyMongo 等访问数据库，这些在日常开发中会经常用到。掌握 ScikitLearn、TensorFlow、PyTorch 等机器学习库对数据研究会有帮助。

JavaScript 开发使用 Visual Studio Code 编写和调试代码，前端使用 Vue[16]框架，后端使用 Node.js[17]引擎。JavaScript 主要用来开发通过 Web 浏览器进行多用户访问的各类管理后台，也可以通过 Google 的 V8 引擎嵌入到 C++交易终端中作为脚本语言开发证券交易策略。

前端开发中，目前 React、Vue、Angular 三大主流框架，React 和 Angular 相比 Vue 都更加庞大，学习曲线陡峭，考虑主要用来开发管理后台，简单易用即可，不准备投入太多精力进行深度定制，故选择使用 Vue 框架，掌握 Vue 项目构建、Element UI 控件使用、webpack 配置、账户登录验证和权限控制、基本的 HTML 和 CSS 编写等即可。后端开发用到的类库和技术主要有 koa 路由和解析前端请求、mysql 和 sequelize 访问数据库、async 异步调用、net 网络通信等。

C#开发使用 Visual Studio 编写代码，使用 DevExpress 编写用户界面。Java 开发使用 Eclipse 编写代码。两者既可以作为核心语言开发整套交易系统，也可以作为辅助仅用于编写部分功能模块，具体使用依开发者偏好来定。本书使用 C++作为核心开发语言，C#和 Java 仅用于开发部分周边项目或提供一些开发示例，少量开发环境要素在用到的时候再作讲解。

数值计算方面目前也是以 Python 为主，Matlab、R、Octave 等已很少使用，Julia 也尚未流行，有时会有一些与 C++等混合编程实现相互调用的需求，总之等有机会用到的时候再说。

日常使用的数据库主要有 MySQL 或 MariaDB、SQL Server、MongoDB、PostgreSQL、SQLite 这几个关系型数据库，KDB+[18]、DolphinDB[19]、InfluxDB[20]等内存时间序列数据库在对数据存取性能有要求时会用到，LevelDB、Redis、BerkeleyDB、Memcached、Tokyo Tyrant/Cabinet、HBase 等 key-value 型 NoSQL 缓存数据库也会用到一些。要求掌握安装、配置，通过数据库 API 接口进行连接、登录、增、删、改、查、嵌套查询、查询优化等操作。

本地代码和文档版本管理使用 SVN，开源代码和文档版本管理使用 GIT。通过[add]增加、[del]删除、[fix]修复、[doc]文档、[new]新功能、[test]测试用例、[style]样式修改等标签对版本日志进行标记。

使用新的技术手段往往能获得效能提升，但也存在较高的试错成本，新技术更新迭代太快，自身架构不够稳定，经常会有无法向后兼容的调整发生，不妨先静观其变，等到稳定、流行、普及的时候再予应用。以上罗列的开发环境以当前主流成熟的方案为主，本小节内容也会随着各种技术、语言、类库的不断发展进化而作更新。

注释说明

<01>. https://github.com/google/styleguide
<02>. https://firefox-source-docs.mozilla.org/code-quality/coding-style/
<03>. https://github.com/vuejs/vue
<04>. https://github.com/mono/mono
<05>. https://github.com/elastic/elasticsearch
<06>. https://www.qt.io
<07>. https://codelite.org
<08>. https://www.codeblocks.org
<09>. https://www.eclipse.org
<10>. https://cmake.org
<11>. https://www.boost.org
<12>. https://www.pydev.org
<13>. https://www.jetbrains.com/pycharm/
<14>. https://www.anaconda.com
<15>. https://www.python.org/dev/peps/
<16>. https://vuejs.org
<17>. https://nodejs.org
<18>. https://kx.com
<19>. https://dolphindb.com
<20>. https://www.influxdata.com

参考文献

[01].