PANDORA 交易系统

系 统 介 绍

PandoraTrader
量化交易系统

文档标识

文档名称	PandoraTrader 系统介绍
版本号	<V1.0>
状况	◉ 草案 ○ 评审过的 ○ 更新过的 ○ 定为基线的

文档修订历史

版本	日期	描述	修订者
V0.9	<2019-2-1>	1. 草稿	吴长盛
V1.0	<2022-2-4>	2. 初稿	吴长盛

正式核准

姓名	签字	日期

分发控制

副本	接受人	机构

目 录

Contents

1. 第一章 介绍

1.1 Pandora Trader 名称由来

据百度百科介绍，Pandora 是希腊神话中赫菲斯托斯用粘土做成的第一个女人。众神赠予使她拥有更诱人的魅力的礼物：火神赫菲斯托斯给她做了华丽的金长袍；爱神维纳斯赋予她妩媚与诱惑男人的力量；众神使者赫耳墨斯教会了她言语的技能。神灵们每人给她一件礼物，但唯独智慧女神雅典娜拒绝给她智慧。

古希腊语中，潘是所有的意思，多拉则是礼物。“潘多拉”即为“拥有一切天赋的女人”；而 PandoraTrader 则寓意为拥有各种功能的交易软件。

我们设计这样一个“开箱即用的”交易平台拥有设计者赋予各种技能，唯独不携带智慧；这个智慧是属于策略设计者的，期待策略设计者设计优秀的策略给予交易软件足够的智慧，能够在飘荡的市场上乘风破浪，挂云帆济沧海。

1.2 概述

Pandora Trader 是一个基于 C++ 的高频量化解决方案。该解决方案，是作者根据自己的高频量化交易策略需求，而逐步形成的。目前代码已经久经考验，从 2017 年年初实盘稳定运行至今，能够满足作者自身的量化交易需求。作者自己也在做量化策略，期待和各位量化大牛深入探讨量化交易策略和量化交易系统。欢迎各位大牛与作者联系（pandoratrader@163.com）。

获取地址：https://github.com/pegasusTrader/PandoraTrader

Pandora Trader 由交易核心（Trading core)，策略模块，虚拟交易所等模块组成。交易核心通过对接交易 API, 将策略需要的信息维护在本地，方便策略快速查询获取。策略模块是提供了基础的策略调用接口，采用 c++这样通用的语言环境，策略设计者可以放飞创意，不受限制引入各种第三方库来辅助策略。虚拟交易所支持 Tick 级别的排队撮合校验，撮合判定有很高准确度。

2. 第二章 体系结构

2.1 目录结构

```
PANDORATRADER
        │  PandoraTrader.sln
        │  README.md
        │
        ├─Doc----------------------------------------文档说明
        ├─Interface
        │  ├─lib--------------------------------------平台支持库文件
        │  │  ├─Debug
        │  │  │
        │  │  ├─Release
        │  │  │
        │  │  ├─Linux
        │  │  │
        │  │  ├─Ubuntu
        │  │  │
        │  │  └─X64
        │  │      ├─Debug
        │  │      │
        │  │      └─Release
        │  │
        │  ├─include----------------------------------平台公共头文件
        │  │
        │  ├─CTPTradeApi64----------------------------X64 CTP API
        │  │
        │  ├─CTPTradeApi32----------------------------Win32 CTP API
        │  │
        │  └─CTPTradeApiLinux-------------------------Linux CTP API
        │
        ├─PandoraTrade--------------------------------实盘交易程序
        │      ReadMe.txt
        │      PandoraTrader.vcxproj
        │      stdafx.cpp
        │      stdafx.h
        │      targetver.h
        │      PandoraTrader.vcxproj.user
```

```
        │      PandoraTraderConfig.xml------------------策略交易配置文件，负责配置行情（前置地址，用户，密码等），交易（前置地址，用户，密码，授权等），策略配置文件等
        │      PandoraDemoStrategyTrader.cpp------------策略交易平台主程序，负责实例化策略，行情和交易，并初始化他们
        │
        ├─PandoraSimulator-----------------------------回测验证程序
        │      PandoraSimulator.vcxproj
        │      PandoraSimulator.vcxproj.user
        │      PandoraSimulator.vcxproj.filters
        │      PandoraSimulator.cpp---------------------回测平台主程序，负责实例化回测系统，包括策略，模拟交易模块和模拟撮合等
        │      HisMarketDataIndex.xml-------------------回测历史数据文件表
        │      PandoraSimulatorConfig.xml---------------回测使用配置文件
        │
        └─PandoraStrategy------------------------------用户策略库
          │  PandoraStrategy.vcxproj
          │  PandoraStrategy.vcxproj.user
          │  PandoraStrategy.vcxproj.filters
          │  cwEmptyStrategy.cpp--------------------------空策略 cpp
          │  cwEmptyStrategy.h----------------------------空策略(啥操作都不执行，用于检验连接登录等）的头文件
          │  cwStrategyDemo.h-----------------------------演示策略头文件
          └─ cwStrategyDemo.cpp--------------------------演示策略 cpp
```

PandoraTrader 整个项目主要由 3 个工程组成，PandoraStrategy，PandoraTrade 和 PandoraSimulator 组成。

其中 PandoraStrategy 是系统提供 demo 策略或者用户自编策略管理库，编译后将生产静态策略库，策略库将作为回测和实盘程序编译的输入，实现策略的统一管理。

PandoraTrade 工程则是将策略库中的策略编译成实盘程序用于交易。该工程将行情模块，交易模块和策略组合在一起，形成策略为主体，交易和行情模块为支撑的交易程序，详细见 2.2 实盘程序框架。

PandoraSimulator 工程师将策略库中的策略编译成用于回测的程序。该工程将模拟行情模块，模拟交易模块，模拟撮合系统以及策略组合在一起，尽可能拟合实际交易的回测系统，详细见 2.3 策略回测框架。

系统主要是将各个模块进行组合搭建，其中各个模块可以自行开发替换。

2.2 实盘交易程序框架

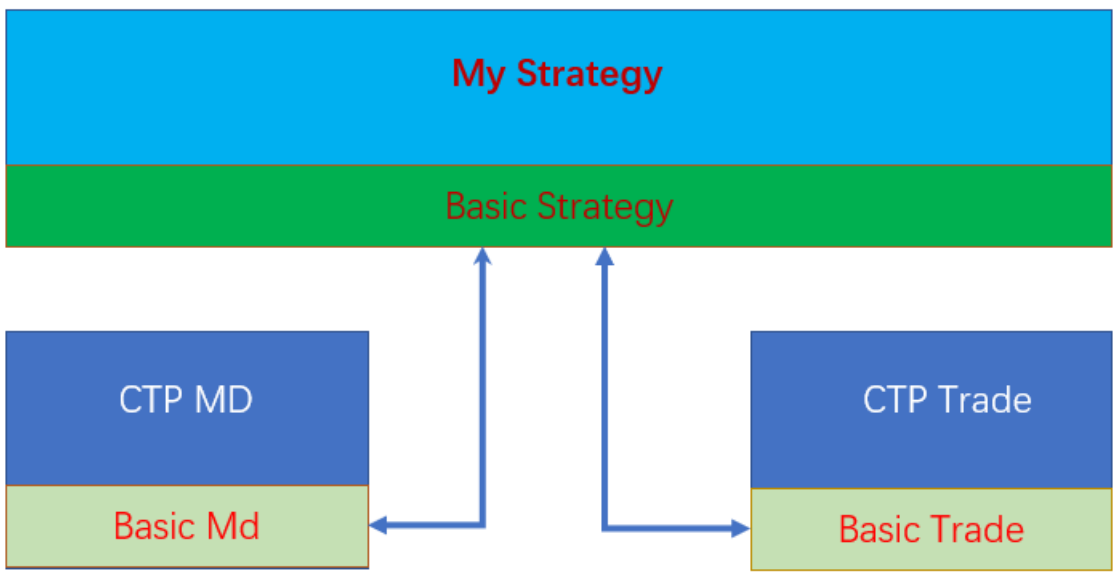
实盘交易程序的编译框架已经提供，该工程主要任务是将行情模块，交易模块和策略模块组合在一起，成为有机的整体。要求行情模块从 cwBasicMdSpi 中派生，交易模块从 cwBasicTradeSpi

中派生而来，而策略则由 cwBasicStrategy 派生而来。通过相互注册工作，组合在一起。其中行情模块，交易模块和策略模块只要满足前面提到到从对应基类中派生出来，各个模块可以自由被替换；从而实现了，同一策略，无需改动任何策略代码，实现交易柜体和行情接口的无缝切换。
具体代码如下：

```
    m_TradeChannel.RegisterBasicStrategy(dynamic_cast<cwBasicStrategy*>(&m_cwStategy));

    m_mdCollector.RegisterTradeSPI(dynamic_cast<cwBasicTradeSpi*>(&m_TradeChannel));
    m_mdCollector.RegisterStrategy(dynamic_cast<cwBasicStrategy*>(&m_cwStategy));
```

各模块间相互关系如下图：



将各个模块组合起来之后，将初始化行情模块和交易模块，代码如下：

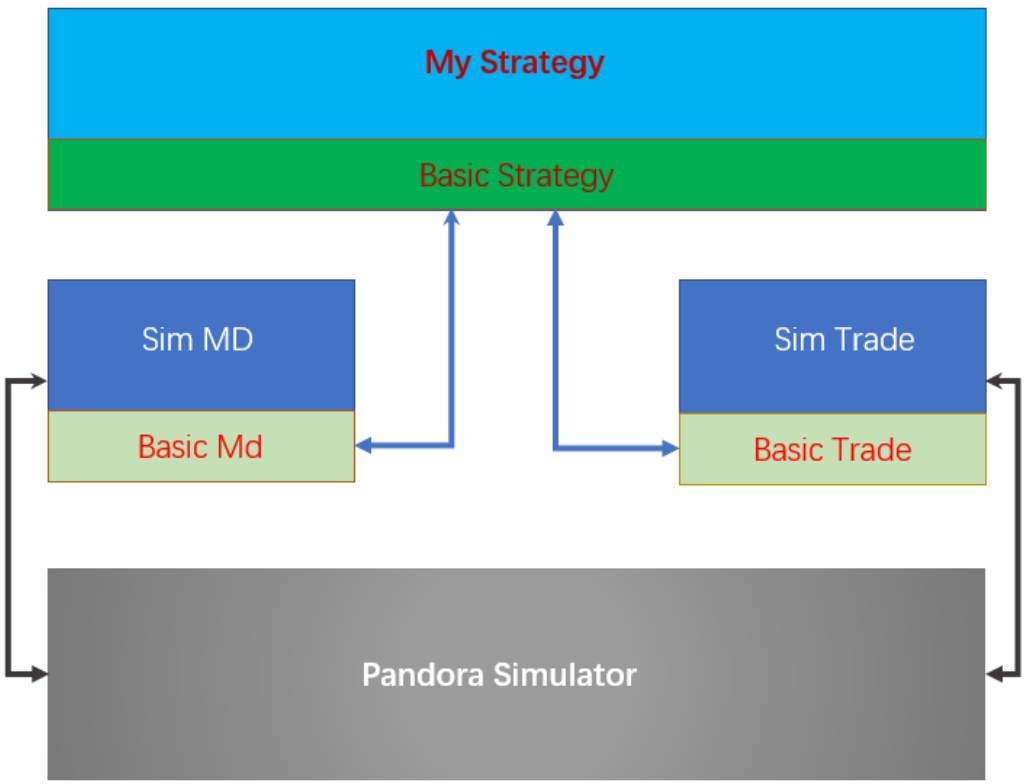
```
    std::thread m_PriceServerThread = std::thread(PriceServerThread);

    std::thread m_TradeServerThread = std::thread(TradeServerThread);
```

这样就完成实盘交易程序初始化，行情和交易模块独立工作，又将相关事件通知到策略模块，实现以策略为主体，各个模块协同工作的实盘交易策略。

2.3 策略回测程序框架

策略回测程序，主要是将模拟行情模块，模拟交易模块，模拟撮合系统以及策略组合在一起，形成由模拟撮合系统发出行情，支持报单撤单，成交撮合等功能，将通过模拟行情模块，模拟交易模块与策略进行交互，支持策略开展回测。此时除了策略可以通过调用函数获取是否处于回测之外，策略处理过程和流程和实盘交易完全一致，实现了尽可能模拟实盘交易。
各个模块组合相互关系图如下：



首先是模拟撮合系统（PandoraSimulator）完成初始化，初始化之后，将模拟行情模块和模拟交易模块与模拟撮合系统（Pandora Simulator）进行组合。

```
    m_PegasusSimulator.InitialSimulator(strFullPath.c_str());

#ifdef REALTIME_QUOTE
    m_PegasusSimulator.SetMdSpi((void*)dynamic_cast<cwBasicMdSpi*>(&m_mdColle
ctor));
#else
    m_PegasusSimulator.SetMdSpi((void*)(&m_mdCollector));
#endif // REALTIME_QUOTE

    m_PegasusSimulator.SetTradeSpi((void*)&m_TradeChannel);
```

其次，和实盘交易初始化一致，将从 cwBasicMdSpi 中派生的行情模块，从 cwBasicTradeSpi 中派生而来的交易模块，及由 cwBasicStrategy 派生而来策略，通过相互注册组合在一起。

```
    m_TradeChannel.RegisterBasicStrategy(dynamic_cast<cwBasicStrategy*>(&m_St
rategy));

    m_mdCollector.RegisterTradeSPI(dynamic_cast<cwBasicTradeSpi*>(&m_TradeCha
nnel));
    m_mdCollector.RegisterStrategy(dynamic_cast<cwBasicStrategy*>(&m_Strategy
));
```

将各个模块组合起来之后，将启动各模块的工作,代码如下：

```
    std::thread m_PriceServerThread = std::thread(PriceServerThread);
```

在启动的最后，开始通知模拟器（PandoraSimulator）开始回测。
开始回测之后，模拟器（PandoraSimulator）读取行情数据，并通过 Sim Md 通知到策略，策略根据自身的逻辑，调用下单函数，通过 Sim Trade 对下单、撤单操作传递给模拟器，模拟器将根据行情情况判定挂单和成交情况，并将模拟相应的回报通过 Sim Trade 返回，通知策略。实现了模拟实盘环境的回测操作。

3. 第三章 快速上手

为方便您能够快速上手使用该量化平台，特做此说明：

3.1 Windows 平台

Windows 下使用 PandoraTrader,建议使用 visual studio 来配置管理工程项目。从 Github 或者 gitee 上下载后，按配置好的工程项目，就可以开始 Pandora 量化之旅。

3.1.1 下载安装 visual studio。

在微软官方可以下载 visual studio（https://visualstudio.microsoft.com/zh-hans/vs/）。
目前 visual studio 社区版对个人客户免费使用。
为更好兼容和使用 PandoraTrader，请选择 visual studio 2015 或更新版本来适配。

3.1.2 获取 PandoraTrader

可以直接访问 Github 地址下载
Github 地址：https://github.com/pegasusTrader/PandoraTrader

也可以从 gitee 上获取：
Getee 地址：https://gitee.com/wuchangsheng/PandoraTrader

在 github 上，有两个分支：
Master：这是目前经过有效测试后，实盘运行的稳定版本。
develop：新增很多新功能，会持续更新，已经完成模块测试，实盘还需进一步进行测试。

用 git 命令获取 master 默认分支：
git clone https://github.com/pegasusTrader/PandoraTrader.git

用 git 命令获取 develop_6_5_1 分支如下：
git clone -b develop_6_5_1 https://github.com/pegasusTrader/PandoraTrader.git
对于 6.5.1 的版本，可以从 develop_6_5_1 分支获取。

3.2 Centos

Centos Linux 环境下，请从 develop_6_5_1 分支获取最新的内容，使用 CMAKE,G++来编译工程。

用 git 命令获取 develop_6_5_1 分支如下：
git clone -b develop_6_5_1 https://github.com/pegasusTrader/PandoraTrader.git

进入目录后：

在 PandoraTrade 的 CmakeLists.txt 中：
set_property(TARGET CTPTradeLIB PROPERTY IMPORTED_LOCATION "/root/projects/PandoraTrader/Interface/CTPTradeApiLinux/thosttraderapi_se.so")
set_property(TARGET CTPMdLIB PROPERTY IMPORTED_LOCATION "/root/projects/PandoraTrader/Interface/CTPTradeApiLinux/thostmduserapi_se.so")
要将其中的路径修改为本地文件的实际全路径。

编译 debug:

```
mkdir builddebug
cd builddebug
cmake -DCMAKE_BUILD_TYPE=DEBUG ..
```

```
make
```

编译 Release:

```
mkdir buildrelease
cd buildrelease
cmake -DCMAKE_BUILD_TYPE=RELEASE ..
make
```

3.3 Ubuntu

Centos Linux 环境下，请从 develop 分支获取最新的内容，使用 CMAKE,G++来编译工程。
库和编译方式与 Centos 的一致，但库的内容，需要从 Ubuntu 替换到 Linux 文件夹中。

3.4 策略编写

用 visual studio 打开 PandoraTrader.sln，可以打开已经配置好的 PandoraTrader 工程,其中有三个工程：
PandoraStrategy：是策略库工程，用来管理自己的策略
目前这个工程提供了三个 demo 策略，可以在该工程新建自己的新策略。
cwEmptyStrategy：一个 demo 策略，无逻辑实现，可以测试 PandoraTrader 和柜台的连通性等。
cwMarketDataReceiver：一个行情接受的 demo 策略。
cwPandoraPairTrading：套利，配对交易的 demo 策略，演示了 Agent 的使用和配对交易的实现方法。
cwStrategyDemo：一个接口调用的 demo 策略。
PandoraSimulator：选定需要回测的策略，并将其编译成一个用于回测的程序。执行该程序，配置相关信息，就可以实现策略的回测功能。编译需依赖 PandoraStrategy 工程。
PandoraTrader：选定需要实盘交易的策略，并将其编译成一个实盘交易使用的程序。执行该程序可以实现对应策略的实盘交易。编译需依赖 PandoraStrategy 工程。

3.5 进行回测

配置 PandoraSimulatorConfig.xml 文件，配置回测模式，数据源，回测速度，合约信息，初始资金。

```
<?xml version="1.0" encoding="utf-8"?>
<Config Generated_at="2016-12-26 23:21:47">
<!-- Configs for PegasusTrader -->
<User type="2">
    <SimulatorServer        Front="F:\HisData\HisMarketDataIndex.xml"        Interval="0"
```

Instrument="F:\HisData\Instrument.xml"/> //设置数据源的地址
<Account PreBalance="0"/> //初始的资金
</User>
<Subscription>
<!--
<Instrument ID="j1909"/>
-->
</Subscription>
<StrategyConfigFile>
.\StrategyTest.xml //策略配置文件，可以通过 initialStrategy 传给策略，但也可以不存在策略配置文件
</StrategyConfigFile>
<Result>
<TotalResult bSave="1" Interval="10" />
<InsResult ID="j1909"bSave="0" Interval="1"/>
</Result>
</Config> //TotalResult：配置的是整个回测结果。Interval 代表回测盈亏曲线结果，代表曲线的采样周期。
InsResult：分合约配置回测结果。ID 为合约，bSave 配置是否保存为文件，Interval 配置采样间隔。

//回测模式，数据源，回测速度，合约信息，初始资金等配置

```
<User type="2">
    <SimulatorServer Front="F:\HisData\HisMarketDataIndex.xml" Interval="0" Instrument="F:\HisData\Instrument.xml"/>
    <Account PreBalance="0"/>
</User>
```

Type：回测模式配置，目前支持的回测数据源类型如下

```
type_CSV_file = 0,                  //CSV文件
type_BIN_file=1,                    //bin二进制文件
type_CSV_List_file=2,               //CSV文件序列
type_BIN_List_file=3,               //bin二进制文件序列
type_DB=4,                          //数据库（暂不支持）
type_REAL_Time_Quote=5,             //实时行情
type_Custom_Quote=6                 //用户自定义数据
```

Front：
对于 type_CSV_file 和 type_BIN_file 两种模式的时候，应该配置行情文件的全路径。
对于 type_CSV_List_file 和 type_BIN_List_file 两种模式的时候，应该配置行情文件序列的文件全路径。配置的行情文件序列的文件数据格式如下，回测时，将按 DateIndexId 顺序来读取行情文件来进行回测。

```
<?xml version="1.0" ?>
<HisMDFiles>
    <MDFile DateIndexId="202011230" FilePath="F:\dat\Log\MarketData_20201123_111144.csv" />
    <MDFile DateIndexId="202011231" FilePath="F:\dat\Log\MarketData_20201123_205940.csv" />
    <MDFile DateIndexId="202011240" FilePath="F:\dat\Log\MarketData_20201124_082456.csv" />
    <MDFile DateIndexId="202011241" FilePath="F:\dat\Log\MarketData_20201124_204537.csv" />
</HisMDFiles>
```

对于 type_REAL_Time_Quote 模式，模拟器接入的行情应该是实盘的行情，并将行情接口如 cwFtdMdSpi 的 m_bNoUseBasicMdUpdate 设置为 true(默认值应该为 false).其余按实盘正常初始化即可。在该模式下，可以用实盘的行情和速度来进行回测，并记录回测的结果。

对于 type_Custom_Quote 模式，在回测模拟器中，需要自行读取行情数据，并调用 cwPegasusSimulator 的 AddCustomData 函数送给模拟器，模拟器会将收到的行情数据，按正常的回测逻辑进行撮合进行回测。

Interval：行情播放时间间隔

Instrument：配置合约信息的全路径。合约信息在接入 simnow 或者实盘，即用交易接口接入 cwFtdTradeSpi 后会自行从交易柜台获取合约信息，并保存。

```
  <Subscription>
<!--
    <Instrument ID="j1909"/>
-->
  </Subscription>
```

//订阅合约信息

<Instrument ID="j1909"/>中输入所需测试的期货合约（可以配置多个）即可，也可以在策略中调用 cwBasicStrategy 来订阅合约，此处配置的合约和策略调用方法来订阅的并集是整个策略订阅合约的集合。

```
    <StrategyConfigFile>
      .\StrategyTest.xml
    </StrategyConfigFile>
```

在 PandoraDemoStrategyTrader 已经实现功能读取策略配置文件路径，并调用策略 InitialStrategy 将该字段传给策略。策略可以根据这个配置文件的路径，自主在配置文件中读取策略需要的配置参数信息。

```
 <Result>
    <TotalResult bSave="1" Interval="10" />
    <InsResult ID="IC1907" bSave="0" Interval="1"/>
 </Result>
```

回测结果分析数据，

TotalResult：配置的是整个回测结果

InsResult：分合约配置回测结果。ID 为合约，bSave 配置是否保存为文件，Interval 配置采样间隔。

3.6 实盘交易的配置

在 PandoraTrade 的文件夹下有个 PandoraTraderConfig.xml 配置文件，请将该文件拷贝到可执行文件所在文件夹下（用 visual studio 来编译的，编译成功后，如果可执行文件夹下没有该文件，会自动将该配置文件拷贝到可执行文件目录下）。

//行情配置信息
<MarketDataServer Front="tcp://180.168.146.187:10110" BrokerID="9999" UserID="" PassWord=""/>
将交易的账号信息（前置地址、BrokerID、UserID 及密码）

//交易配置信息
<TradeServer Front="tcp://180.168.146.187:10100" BrokerID="9999" UserID="" PassWord="" ProductInfo="Pandora" AppID="Pandora" AuthCode="Pandora"/>
将交易的账号信息（前置地址、BrokerID、UserID 及密码，AppID 和授权码）

//订阅合约信息
<Instrument ID="j1909"/>中输入所需测试的期货合约（可以配置多个）即可，也可以在策略中调用 cwBasicStrategy 来订阅合约，此处配置的合约和策略调用方法来订阅的并集是整个策略订阅合约的集合。

//策略配置文件路径
<StrategyConfigFile>
K:\Release\StrategyDemo.xml
</StrategyConfigFile>
在 PandoraDemoStrategyTrader 已经实现功能读取策略配置文件路径，并调用策略 InitialStrategy 将该字段传给策略。策略可以根据这个配置文件的路径，自主在配置文件中读取策略需要的配置参数信息。

4. 第四章 开发使用

4.1 策略视角初始化流程

(1)InitialStrategy

这个函数将在最开始被调用，被调用时，还未开始行情模块，交易模块的初始化操作。建议比较耗时的策略初始化在此处进行，（如果需要合约信息，建议在目录中准备 Instrument

文件，并调用 InitalInstrumentData 函数获取，后将正常获取相关合约信息）。其他的函数回调，都是在运行过程中，建议不要处理耗时的操作，如有较为耗时操作，可以采用异步方式。

(2) OnReady

这个函数将在交易初始化完成之后，将会被调用。从策略层面订阅合约，策略参数初始化等。

建议策略在收到 OnReady 之后，才开始相应的策略逻辑。

（3）根据事件驱动，回调行情， 订单状态，成交等信息，触发策略开展相应策略逻辑。

（4）建议利用行情时间，通过 4.2.35GetTradeTimeSpace 获取相应的交易时间，判断交易状态，离收盘多少，做好相关收盘准备（特别是日内策略，避免隔夜等）。

4.2 cwBasicStrategy

cwBasicStrategy 是策略的基础类，该类是为 cwBasicKindleStrategy 服务，cwBasicKindleStrategy 是从 cwBasicStrategy 派生出来的。用户策略不得从该策略派生。用户策略应该以 cwBasicKindleStrategy 来派生；其中，用户策略可以调用 cwBasicStrategy 提供的函数功能。用户策略不可直接从 cwBasicStrategy 派生策略，否则提供回调函数无法被系统调用。函数调用如果返回的类型是 shared_ptr,需要判定其返回的指针（get()）是否为 NULL。如果为 NULL，则返回信息无效。

本文档对于函数参数[in],[out]说明，对于系统传给用户的信息，都表明为[out]，对于用户传给系统的信息标记为[in].

4.2.1 PriceUpdate 方法

回调函数，当订阅的行情数据更新，该回调函数会被系统调用，用于通知行情更新。

函数原型：

```
virtual void PriceUpdate(cwMarketDataPtr pPriceData) = 0;
```

参数：

[out]pPriceData:行情数据

返回值:无

4.2.2 OnRtnTrade 方法

回调函数，当有成交时，该回调函数会被系统调用，用于通知账户有成交。

函数原型：

```
virtual void OnRtnTrade(cwTradePtr pTrade) = 0;
```

参数：

[out]pTrade:成交信息

返回值:无

4.2.3 OnRtnOrder 方法

回调函数，当订单有更新时，该回调函数会被系统调用，用于通知订单更新

函数原型：

```
virtual void OnRtnOrder(cwOrderPtr pOrder, cwOrderPtr pOriginOrder = cwOrderPtr()) = 0;
```

参数：

[out]pOrder:最新报单;

[out]pOriginOrder:上一次更新报单结构体，有可能为NULL（即第一次收到订单更新）

返回值:无

4.2.4 OnOrderCanceled 方法

回调函数，当订单撤单成功，该回调函数会被系统调用，用于通知订单撤单成功。

函数原型：

```
virtual void OnOrderCanceled(cwOrderPtr pOrder) = 0;
```

参数：

[out]pOrder:已经撤单的报单。

返回值:无

4.2.5 OnRspOrderInsert 方法

回调函数，下单后，柜台会给相应的报单响应，该回调函数会被系统调用，用于通知柜台的报单响应。

函数原型：

```
virtual void OnRspOrderInsert(cwOrderPtr pOrder, cwRspInfoPtr pRspInfo) {};
```

参数：

[out]pOrder:柜台响应的订单的信息;

[out]pRspInfo:相应的柜台回应信息。

返回值:无

4.2.6 OnRspOrderCancel 方法

回调函数，撤单后，柜台会给相应的撤单相应，该回调函数会被系统调用，用于通知柜台的撤单相

应。

函数原型：

```
virtual void OnRspOrderCancel(cwOrderPtr pOrder, cwRspInfoPtr pRspInfo) {};
```

参数：

[out]pOrder:柜台响应的订单的信息；

[out]pRspInfo:相应的柜台回应信息。

返回值:无

4.2.7 OnStrategyTimer 方法

回调函数，当策略定时器超时，该回调函数会被系统调用。策略定时器可以通过SetTimer函数来设定；取消定时器可以通过RemoveTimer来实现。用iTimerId来标识哪个定时器触发。

函数原型：

```
virtual void OnStrategyTimer(int iTimerId, const char * szInstrumentID) {};
```

参数：

[out]iTimerId:定时器ID,用于标识是哪个定时器触发。

返回值:无

4.2.8 OnReady 方法

回调函数，当策略交易初始化完成时，该回调函数会被系统调用，可以在此函数做策略的初始化操作。

函数原型：

```
virtual void OnReady() {};
```

参数：无

返回值:无

4.2.9 OnSimulationPartEnd 方法

回调函数，只有回测该函数才能被系统调用。回测部分结束（夜盘结束和日盘结束）将被调用。type_CSV_List_file和type_BIN_List_file两种回测模式下，每个文件行情回放结束后，该函数被系统回调。

函数原型：

```
virtual void OnSimulationPartEnd() {};
```

参数：无

返回值:无

4.2.10 OnSimulationFinished 方法

回调函数，只有回测该函数才能被系统调用。整个回测结束将被调用

函数原型：

```
virtual void OnSimulationFinished() {};
```

参数：无

返回值:无

4.2.11 InitialStrategy 方法

策略初始化函数，此处可以在主流程中调用，用于对策略的初始化。用户策略可以重写（override）该函数，实现初始化功能。在BasciStrategy中,该函数判断就是初始化策略配置文件变量m_strConfigFileFullPath的。如果参数不为空且字符长度大于零，则将参数赋值给m_strConfigFileFullPath；否则m_strConfigFileFullPath则赋值为程序所在目录的路径加BasicStrategyConfig.xml（即认为BasicStrategyConfig.xml存在，且在程序所在目录的路径下；m_strConfigFileFullPath是该文件的全路径）。

函数原型：

```
virtual void InitialStrategy(const char * pConfigFilePath);
```

参数：

[in] pConfigFilePath:策略配置文件，可传入空值；如果传入空值，数BasicStrategyConfig

返回值:无

4.2.12 InitalInstrumentData 方法

在Trade SPI准备就绪前，策略需要用到合约信息，可以利用该函数先从文件中获取合约信息；参数为nullptr时，默认和程序在同一路径下的Instrument.xml。

Instrument.xml文件格式参照如下：

```
<?xml version="1.0" ?>
<Instruments>
<Instrument ExchangeID="SHFE" InstrumentID="au2012" InstrumentName="黄金2012"
ProductID="au" ProductClass="1" CreateDate="20191016" OpenDate="20191118"
ExpireDate="20201215" Currency="1" OptionsType="" StartDelivDate="20201216"
EndDelivDate="20201216" PositionType="2" MaxMarginSideAlgorithm="0"
UnderlyingInstrID="au" UnderlyingMultiple="0" DeliveryYear="2020"
DeliveryMonth="12" MaxMarketOrderVolume="30" MinMarketOrderVolume="3"
MaxLimitOrderVolume="500" MinLimitOrderVolume="3" IsTrading="1"
VolumeMultiple="1000" PriceTick="0.02" StrikePrice="0" />
    <Instrument ExchangeID="CZCE" InstrumentID="CF105" InstrumentName="一号棉花
105" ProductID="CF" ProductClass="1" CreateDate="20200520" OpenDate="20200520"
```

```
ExpireDate="20210514" Currency="1" OptionsType="" StartDelivDate="20210514"
EndDelivDate="20210514" PositionType="2" MaxMarginSideAlgorithm="0"
UnderlyingInstrID="" UnderlyingMultiple="1" DeliveryYear="2021" DeliveryMonth="5"
MaxMarketOrderVolume="1000" MinMarketOrderVolume="1" MaxLimitOrderVolume="1000"
MinLimitOrderVolume="1" IsTrading="1" VolumeMultiple="5" PriceTick="5"
StrikePrice="0" />
    <Instrument ExchangeID="DCE" InstrumentID="l2104-C-9200"
InstrumentName="l2104-C-9200" ProductID="l_o" ProductClass="2"
CreateDate="20201215" OpenDate="20201215" ExpireDate="20210305" Currency="1"
OptionsType="1" StartDelivDate="20210305" EndDelivDate="20210305" PositionType="2"
MaxMarginSideAlgorithm="0" UnderlyingInstrID="l2104" UnderlyingMultiple="1"
DeliveryYear="2021" DeliveryMonth="4" MaxMarketOrderVolume="1000"
MinMarketOrderVolume="1" MaxLimitOrderVolume="1000" MinLimitOrderVolume="1"
IsTrading="1" VolumeMultiple="5" PriceTick="0.5" StrikePrice="9200" />
       </Instruments>
```

函数原型：

```
virtual void InitalInstrumentData(const char * pInstrumentDataFilePath = NULL);
```

参数：

[in]pInstrumentDataFilePath:参数合约信息的文件全路径。

返回值:无

4.2.13 GetStrategyName 方法

用于获取策略名称。用户策略可以重写（override）该函数，重写时函数名，参数列表，返回值类型，必须同基类中被该函数一致；函数实现可不一致。在调用时，会调用用户策略重写，而不会调用基类的函数。如果用户没有重写该函数，在调用时，应返回基类的函数，返回值为“BasicStrategy”。

函数原型：

```
virtual std::string  GetStrategyName() { return "BasicStrategy"; }
```

参数：无

返回值:返回策略名称

4.2.14 GetLastestMarketData 方法

获取有订阅合约的最新的行情。平台会记录有订阅合约的最新行情数据（一条）；可通过该函数获取，如果传入的合约无效或者未订阅，则返回的cwMarketDataPtr对应指针为NULL。

函数原型：

```
cwMarketDataPtr GetLastestMarketData(std::string InstrumentID);
```

参数：

[in]InstrumentID:需要获取的合约

返回值：订阅合约的最新的行情

4.2.15 GetAccount 方法

获取账户信息

函数原型：

```
cwAccountPtr GetAccount();
```

参数：无

返回值:账户信息，包括总资金，可以资金，保证金，盈亏，手续费等信息。

4.2.16 GetActiveOrders 方法

获取挂单列表。

函数原型：

```
bool GetActiveOrders(std::map<std::string, cwOrderPtr>& ActiveOrders);
```

参数：

[out]ActiveOrders:订单列表的Map,用于存放放回的报单信息，key为本地报单引用（OrderRef）。

返回值：在交易接口尚未初始化或者策略平台未接入则返回false,正常返回则为true。

4.2.17 GetActiveOrders 方法

获取传入合约的挂单列表。

函数原型：

```
bool GetActiveOrders(std::string InstrumentID, std::map<std::string, cwOrderPtr>& ActiveOrders);
```

参数：

[in]InstrumentID:获取挂单列表的指定合约；

[out] ActiveOrders:订单列表的Map,用于存放放回的报单信息，key为本地报单引用（OrderRef）。

返回值：在交易接口尚未初始化或者策略平台未接入则返回false,正常返回则为true。

4.2.18 GetActiveOrdersLong 方法

获取多头挂单数量（手数），不论开平等。

函数原型：

```
int GetActiveOrdersLong(std::string InstrumentID);
```

参数：

[in]InstrumentID:获取挂单列表的指定合约

返回值：指定合约的多头挂单手数

4.2.19 GetActiveOrdersShort 方法

获取空头挂单数量（手数），不论开平等。

函数原型：

```
int GetActiveOrdersShort(std::string InstrumentID);
```

参数：

[in]InstrumentID:获取挂单列表的指定合约

返回值：指定合约的空头挂单手数

4.2.20 GetAllOrders 方法

获取所有报单列表，传入map用于返回报单信息，交易所报单编号(sysOrderID)为Key

函数原型：

```
bool GetAllOrders(std::map<std::string, cwOrderPtr>& Orders);
```

参数：

[in]InstrumentID:获取挂单列表的指定合约；订单列表的Map,用于存放放回的报单信息，key为系统报单编号（交易所报单编号sysOrderID）。

返回值：在交易接口尚未初始化或者策略平台未接入则返回false,正常返回则为true。

4.2.21 GetTrades 方法

获取所有成交列表，传入map用于返回信息，交易所成交编号(TradeID)为Key

函数原型：

```
bool GetTrades(std::map<std::string, cwTradePtr>& trades);                    ///Key TradeID
```

参数：

[out]trades:成交列表的Map,用于存放放回的成交信息，key为交易所成交编号（TradeID）。

返回值：在交易接口尚未初始化或者策略平台未接入则返回false,正常返回则为true。

4.2.22 GetPositions 方法

获取持仓列表，传入map用于返回信息，合约ID为Key

函数原型：

```
bool GetPositions(std::map<std::string, cwPositionPtr>& PositionMap);
```

参数：

[out]PositionMap:持仓列表的Map,用于存放放回的报单信息，key为合约ID（InstrumentID）。

返回值：在交易接口尚未初始化或者策略平台未接入则返回false,正常返回则为true。可通过

m_bStrategyReady来判断，初始化完成为true，否则为false。

4.2.23 GetNetPosition 方法

获取合约净头寸。净头寸定义为多头持仓减去空头持仓，当净头寸为正值N，则说明多头持仓比空头持仓多N手；当净头寸为零时，说明多头持仓和空头持仓相等（可能两个都为零）；当净头寸为负值（-N,N为正数），则说明多头持仓比空头少N手。

函数原型：

```
int  GetNetPosition(std::string InstrumentID);
```

参数：

[in]InstrumentID:要查询净头寸的合约。

返回值：合约的净头寸，在交易接口尚未初始化或者策略平台未接入时该函数返回0。交易接口是否初始化，能否有效获取，可通过m_bStrategyReady来判断，初始化完成为true，否则为false。

4.2.24 GetPositionsAndActiveOrders 方法

获取持仓和挂单列表。该函数应更为常用，可获取一个时间截面下的持仓和挂单列表；该函数可避免在调用获取持仓（GetPositions）后调用获取挂单列表（GetActiveOrders）之间时间差，挂单和持仓列表没能一致。举例说明：

在原始持仓为0手，有1手挂单多头；当调用GetPosition时获取持仓为0，这个时间差内挂单被成交，此时持仓为1手，没有挂单；再次调用获取挂单列表时，返回没有挂单。程序获取的信息为当前没有持仓，亦没有挂单。

故提供此函数来解决，需要同时获取持仓和挂单列表的信息，避免信息变动造成组合后的信息失真。

函数原型：

```
bool GetPositionsAndActiveOrders(std::map<std::string, cwPositionPtr>& PositionMap,
        std::map<std::string, cwOrderPtr>& ActiveOrders);
```

参数：

[out]PositionMap:持仓列表的Map,用于存放放回的报单信息，key为合约ID（InstrumentID）

[out]ActiveOrders:订单列表的Map,用于存放放回的报单信息，key为本地报单引用（OrderRef）

返回值：在交易接口尚未初始化或者策略平台未接入则返回false,正常返回则为true。可通过m_bStrategyReady来判断，初始化完成为true，否则为false。

4.2.25 GetPositionsAndActiveOrders 方法

获取指定合约持仓和挂单列表，此函数功能和用法与4.2.24描述类似，但该函数对合约进行筛选，即只返回传入的合约的持仓和挂单列表信息。

函数原型：

```
bool GetPositionsAndActiveOrders(std::string InstrumentID, cwPositionPtr&
```

```
pPosition, std::map<std::string, cwOrderPtr>& ActiveOrders);
```

参数：

[in]InstrumentID:要查询净头寸的合约。

[out]PositionMap:持仓列表的Map,用于存放放回的报单信息，key为合约ID（InstrumentID）。

[out]ActiveOrders:订单列表的Map,用于存放放回的报单信息，key为本地报单引用（OrderRef）

返回值：在交易接口尚未初始化或者策略平台未接入则返回false,正常返回则为true。可通过m_bStrategyReady来判断，初始化完成为true，否则为false。

4.2.26 GetNetPositionAndActiveOrders 方法

获取指定合约净持仓和挂单列表，此函数功能用法和4.2.24描述类似，但该函数，持仓仅返回指定合约净头寸和挂单列表信息。

函数原型：

```
bool GetNetPositionAndActiveOrders(std::string InstrumentID, int & iPosition,
std::map<std::string, cwOrderPtr> & ActiveOrders);
```

参数：

[in]InstrumentID:要查询净头寸的合约。

[out]iPosition:指定合约的净头寸。

[out]ActiveOrders:订单列表的Map,用于存放放回的报单信息，key为本地报单引用（OrderRef）

返回值：在交易接口尚未初始化或者策略平台未接入则返回false,正常返回则为true。可通过m_bStrategyReady来判断，初始化完成为true，否则为false。

4.2.27 GetInstrumentData 方法

获取合约信息，主要包括交易所，产品信息，最小变动，合约乘数，最大（小）限价单手数，最大（小）市价单手数等信息。

函数原型：

```
cwInstrumentDataPtr GetInstrumentData(std::string InstrumentID);
```

参数：

[in]InstrumentID:要查询的合约。

返回值：合约信息

4.2.28 SubScribePrice 方法

订阅合约

函数原型：

```
void   SubScribePrice(std::vector<std::string>& SubscribeInstrument);
```

参数：

[in]SubscribeInstrument:要订阅的合约序列。建议一次订阅在50个以内。

返回值：无

4.2.29 UnSubScribePrice 方法

取消订阅合约

函数原型：

```
void   UnSubScribePrice(std::vector<std::string>& UnSubscribeInstrument);
```

参数：

[in]UnSubscribeInstrument:要取消订阅的合约序列。

返回值：无

4.2.30 GetTickSize 方法

获取合约价格最小变动，如果获取失败返回-1。

对有特殊需求或者无法从接口中获取合约信息，可以通过4.2.12InitalInstrumentData函数初始化合约信息。初始化后，可以通过该函数获取。

函数原型：

```
double    GetTickSize(const char * szInstrumentID);
```

参数：

[in]szInstrumentID:合约ID,如au2012。

返回值：合约的最小价格变动，如果获取失败，则返回-1。

4.2.31 GetMultiplier 方法

获取合约乘数，如果获取失败返回-1

对有特殊需求或者无法从接口中获取合约信息，可以通过4.2.12InitalInstrumentData函数初始化合约信息。初始化后，可以通过该函数获取。

函数原型：

```
double    GetMultiplier(const char * szInstrumentID);
```

参数：

[in]szInstrumentID:合约ID,如au2012。

返回值：合约乘数，如果获取失败，则返回-1。

4.2.32 GetMarginRate 方法

获取保证金率，如果获取失败返回空指针，需检查返回值有效性

第一次获取保证金率时，会返回默认值，保证金为1，即百分之百占用，并立即向柜台查询，并保存在本地，再次查询时，将返回最新值或者默认值（仍未查询到相应信息，依然返回默认值）

函数原型：

```
cwMarginRateDataPtr          GetMarginRate(const char * szInstrumentID);
```

参数：

[in]szInstrumentID:合约ID,如au2012。

返回值：保证金率相关结构体，如果获取失败，则返回空指针。

4.2.33 GetCommissionRate 方法

获取手续费率，如果获取失败返回空指针，需检查返回值有效性

第一次获取手续费率时，会返回默认值，手续费率为0，即不收手续费，并立即向柜台查询，并保存在本地，再次查询时，将返回最新值或者默认值（仍未查询到相应信息，依然返回默认值）

函数原型：

```
cwCommissionRateDataPtr      GetCommissionRate(const char * szInstrumentID);
```

参数：

[in]szInstrumentID:合约ID,如au2012。

返回值：手续费率相关结构体，如果获取失败，则返回空指针。

4.2.34 GetProductID 方法

获取产品ID ,如获取失败则返回空指针

对有特殊需求或者无法从接口中获取合约信息，可以通过4.2.12InitalInstrumentData函数初始化合约信息。初始化后，可以通过该函数获取。

函数原型：

```
char *    GetProductID(const char * szInstrumentID);
```

参数：

[in]szInstrumentID:合约ID,如au2012。

返回值：合约乘数，如果获取失败，则返回nullptr。

4.2.35 GetTradeTimeSpace 方法

获取交易时间段，距开盘多少秒和距收盘多少秒。

函数原型：

```
bool  GetTradeTimeSpace(const char * szInstrumentID, const char * updatetime,
        cwProductTradeTime::cwTradeTimeSpace& iTradeIndex, int& iOpen, int& iClose);
```

参数：

[in]szInstrumentID:合约ID,如au2012。

[in]updatetime:当前时间，字符串，格式为“hh:mm:ss”,如“10:23:45”。

[out]iTradeIndex：返回当前交易时段的枚举。枚举定义如下：

```
    NoTrading = 0                                   //非交易时段
    , NightPartOne=1                                //夜盘
```

```
        , AMPartOne=2                                   //上午第一阶段
        , AMPartTwo=3                                   //上午第二阶段
        , PMPartOne=4                                   //下午第一阶段
        , CallAuctionOrderingOpen=5                         //集合竞价报单（开盘）
        , CallAuctionMatchOpen=6                        //集合竞价撮合（开盘）
        , CallAuctionOrderingClose=7                        //集合竞价报单（收盘）
        , CallAuctionMatchClose=8                       //集合竞价撮合（收盘）
```

[out]iOpen:该交易时段已经开盘多久（秒数）

[out]iClose:该交易时段还有多久收盘（秒数）

返回值：通过传入的信息无法查找到系统中内置的品种交易时段，则返回false。

4.2.36 GetPreTimeSpaceInterval 方法

获取前一个交易时段到当前交易时段开盘时间间隔

函数原型：

```
int  GetPreTimeSpaceInterval(const char * szInstrumentID,
cwProductTradeTime::cwTradeTimeSpace iTradeIndex);
```

参数：

[in]szInstrumentID:合约ID,如au2012。

[in]iTradeIndex：返回当前交易时段的枚举。枚举定义见4.2.35 GetTradeTimeSpace。

返回值：通过传入的信息无法查找到系统中内置的品种交易时段，则返回false。

4.2.37 GetTradeTime 方法

获取指定交易时段

函数原型：

```
cwProductTradeTime::TradeTimePtr GetTradeTime(const char * szInstrumentID,
cwProductTradeTime::cwTradeTimeSpace iTradeIndex);
```

参数：

[in]szInstrumentID:合约ID,如au2012。

[in]iTradeIndex：返回当前交易时段的枚举。枚举定义见4.2.35 GetTradeTimeSpace。

返回值：返回交易时段定义信息。

交易时段定义如下：

```
typedef struct tagProductTradeTime
{
    cwTradeTimeSpace TradeTimeSpace;
    cwRangeOpenClose RangeOpenClose;

    cwMarketTimePtr BeginTime;
    cwMarketTimePtr EndTime;
```

```
}ProductTradeTime;
```

其中：

TradeTimeSpace：枚举定义见4.2.35 GetTradeTimeSpace。

RangeOpenClose：定义了开闭区间内容，枚举定义如下：

```
enum cwRangeOpenClose
{
    cwLeftOpenRightOpen = 0,                    //(a,b) ：  左开右开区间
    cwLeftOpenRightClose,                       //(a,b] ：  左开右闭区间
    cwLeftCloseRightOpen,                       //[a,b) ：  左闭右开区间
    cwLeftCloseRightClose                       //[a,b] ：  左闭右闭区间
};
```

BeginTime：时段开始时间

EndTime：时段结束时间

4.2.38 GetTradingDay 方法

获取当前交易日

函数原型：

```
cwPandoraTrader::cwDate  GetTradingDay();
```

参数：

无。

返回值：返回当前交易日信息，返回格式为cwDate形式。

4.2.39 GetInstrumentDateSpace 方法

获取合约的交易状态（按交易时间估计，非精确处理，仅供参考）

交易日历：系统内置交易日历，根据交易所更新交易日历信息，定期更新。

处理方法：按交割月前一天为最后交易日。

函数原型：

```
bool    GetInstrumentDateSpace(const char* szInstrumentID,
cwPandoraTrader::cwDate date,
    cwInstrumentTradeDateSpace& iTradeDateSpace, int &iRemain);
```

参数：

[in]szInstrumentID:需要查询的合约ID,如au2012。

[in]date:需要查询的日期。

[out]iTradeDateSpace:合约交易状态枚举：

```
enum cwInstrumentTradeDateSpace:int
{
    cwFinishDeliver = 0,                    //完成交割
```

```
    cwDeliverMonth = 1,                    //交割月
    cwFirstMonthBeforeDeliverMonth,        //交割月前第一个月
    cwSecondMonthBeforeDeliverMonth,       //交割月前第二个月
    cwRegularTradingDateSpace              //普通交易日期时间段
    };
```

[out]iRemain:剩余天数

返回值：获取成功返回true，否则返回false。

4.2.40 GetBuisnessDayRemain 方法

以当前交易日计，获取合约剩余交易日（按交易时间估计，非精确处理，仅供参考）

处理方法：非中金所（CFFEX），按交割月前一天为最后交易日，中金所按结束日期为最后交易日。

函数原型：

```
bool    GetBuisnessDayRemain(const char* szInstrumentID,
    cwInstrumentTradeDateSpace& iTradeDateSpace, int& iRemain);
```

参数：

[in]szInstrumentID:需要查询的合约ID,如au2012。

[in]date:需要查询的日期。

[out]iTradeDateSpace:合约交易状态枚举，定义见4.2.39 GetInstrumentDateSpace。

[out]iRemain:剩余天数

返回值：获取成功返回true，否则返回false。

4.2.41 GetInstrumentCancelCount 方法

获取合约当前撤单次数

函数原型：

```
int  GetInstrumentCancelCount(std::string InstrumentID);
```

参数：

[in] InstrumentID:合约ID,如au2012。

返回值：该合约的撤单次数，包括下单冻结的部分。如果交易尚未初始化完成，返回值为0。如果接口不支持，返回-1；

4.2.42 GetInstrumentDeclarationMsgCount 方法

获取合约当前信息量

函数原型：

```
int  GetInstrumentDeclarationMsgCount (std::string InstrumentID);
```

参数：

[in] InstrumentID:合约ID,如au2012。

返回值：该合约的信息量，包括下单冻结的部分。如果交易尚未初始化完成，返回值为0。如果接口不支持，返回-1。

4.2.43 IsThisStrategySubScribed 方法

获取合约是否是订阅

函数原型：

```
bool  IsThisStrategySubScribed(std::string InstrumentID);
```

参数：

[in] InstrumentID:合约ID,如au2012。

返回值：返回该函数是否通过4.2.28 SubScribePrice方法订阅过。如果还在订阅中，则返回true，如果未订阅或者是通过4.2.29 UnSubScribePrice取消订阅后，则返回false.

4.2.44 GetIsSimulation 方法

获取当前状态是否为回测模拟情况

函数原型：

```
inline bool GetIsSimulation() { return m_bIsSimulation; }
```

参数：无

返回值：该函数可以返回当前是否运行在回测模式.

4.2.45 SetTimer 方法

设置定时器 iTimerId定时器id，在OnTimer回调依据此id判定是哪个定时器触发，iElapse 触发间隔（毫秒），时间精度为1ms。目前仅支持100个定时器。对同个TimerId进行多次设置，触发间隔将会被覆盖。

函数原型：

```
bool  SetTimer(int iTimerId, int iElapse, const char * szInstrumentID = nullptr);
```

参数：

[in] iTimerId:定时器ID,任意整数都可，如1。

[in] iElapse: 触发间隔（毫秒）,如定时1秒触发一次，此参数为1000

[in] szInstrumentID:该定时器触发的ID

返回值：当定时器数量大于等于100个时，设置定时器会失败，并返回false.否则返回true.

4.2.46 RmoveTimer 方法

移除定时器

函数原型：

```
void  RemoveTimer(int iTimerId);
```

参数：

[in] iTimerId:定时器ID。

返回值：无.

4.3 cwBasicKindleStrategy

cwBasicKindleStrategy 是从 cwBasicStrategy 中派生的类，用户策略可以直接从这个类派生开发自己的策略。与 cwBasicStrategy 一样，函数调用如果返回的类型是 shared_ptr,需要判定其返回的指针（get()）是否为 NULL。如果为 NULL，则返回信息无效。

这个类在 cwBasicStrategy 的基础上做了一些封装方便开发策略。主要提供的功能有：

i. 提供 K 线功能以及回调

用户策略可以直接订阅 k 线，在平台中，会根据订阅的 k 线和周期，合成 k 线，并在适当的时候，通过 OnBar 函数回调通知用户策略 k 线完结。由于 PandoraTrader 不依赖于别的平台，故此处的 K 线没有历史数据，数据是根据订阅的合约和周期，通过依据实时行情进行计算生成的。如果需要历史数据，可以在启动程序或者在策略 OnReady 回调中将历史数据载入对应容器中，就可以正常使用了。

ii. 提供投资组合工作区

平台提供了投资组合管理的功能，对于一个投资组合里的合约所有的回调都是同个线程，不同投资组合是不同线程。这样处理能够降低策略开发者对于线程的把控，不需要考虑策略中线程安全的问题；同时不同投资组合之间互不影响，相互隔离。如果没有指定投资组合，则所有的合约都在默认投资组合中。

iii. 提供代理人（Agent）功能

平台提供了代理人的功能。何谓代理人呢，就是对策略提供特别的功能的类，这个类可以正常收取行情，收取相应的回报；可以提供类似子策略的功能，方便策略控制管理。举个具体的例子，比如策略主要负责策略信号，可以直接调用一个仓位管理的代理人，由其处理所有的报单和持仓；这样策略可以专注于信号的开发，将仓位管理这块相对一致性的代码做整合。这样可以简化策略开发，实现代码的快速复用。

具体功能函数如何使用这些功能，参见后续函数功能说明。

4.3.1 PriceUpdate 方法

回调函数，当订阅的行情数据更新，该回调函数会被系统调用，用于通知行情更新。

如果有订阅K线数据，行情更新后会同时触发PriceUpdate和OnBar的时候，PriceUpdate会先于OnBar。

函数原型：

```
virtual void        PriceUpdate(cwMarketDataPtr pPriceData) {};
```

参数：

[out]pPriceData:行情数据

返回值:无

4.3.2 OnBar 方法

回调函数，当新形成K线的时候，该回调函数会被系统调用，用于通知k线更新。

函数原型：

```
virtual void            OnBar(std::string    InstrumentID,    int    iTimeScale,
cwBasicKindleStrategy::cwKindleSeriesPtr pKindle) {};
```

参数：

[out]InstrumentID:k线更新的合约

[out]iTimeScale:形成新k线的对应的周期

[out]pKindle:k线管理类，里面有k线序列，以及一些k线序列处理函数。这个管理类下是同个合约同个时间周期的k线数据。如果要获取别的合约或周期的k线数据，可以调用GetKindleSeries来获取。

返回值:无

4.3.3 OnRtnTrade 方法

回调函数，当有成交时，该回调函数会被系统调用，用于通知账户有成交。

函数原型：

```
virtual void        OnRtnTrade(cwTradePtr pTrade) {};
```

参数：

[out]pTrade:成交信息

返回值:无

4.3.4 OnRtnOrder 方法

回调函数，当订单有更新时，该回调函数会被系统调用，用于通知订单更新

函数原型：

```
virtual void        OnRtnOrder(cwOrderPtr pOrder, cwOrderPtr pOriginOrder =
cwOrderPtr()) {};
```

参数：

[out]pOrder:最新报单;

[out]pOriginOrder:上一次更新报单结构体，有可能为NULL（即第一次收到订单更新）

返回值:无

4.3.5 OnOrderCanceled 方法

回调函数，当订单撤单成功，该回调函数会被系统调用，用于通知订单撤单成功。

函数原型：

```
virtual void        OnOrderCanceled(cwOrderPtr pOrder) {};
```

参数：

[out]pOrder:已经撤单的报单。

返回值:无

4.3.6 OnRspOrderInsert 方法

回调函数，下单后，柜台会给相应的报单响应，该回调函数会被系统调用，用于通知柜台的报单响应。

函数原型：

```
virtual void        OnRspOrderInsert(cwOrderPtr pOrder, cwRspInfoPtr pRspInfo) {};
```

参数：

[out]pOrder:柜台响应的订单的信息;

[out]pRspInfo:相应的柜台回应信息。

返回值:无

4.3.7 OnRspOrderCancel 方法

回调函数，撤单后，柜台会给相应的撤单相应，该回调函数会被系统调用，用于通知柜台的撤单相应。

函数原型：

```
virtual void OnRspOrderCancel(cwOrderPtr pOrder, cwRspInfoPtr pRspInfo) {};
```

参数：

[out]pOrder:柜台响应的订单的信息;

[out]pRspInfo:相应的柜台回应信息。

返回值:无

4.3.8 OnStrategyTimer 方法

回调函数，当策略定时器超时，该回调函数会被系统调用。策略定时器可以通过SetTimer函数来设定；取消定时器可以通过RemoveTimer来实现。用iTimerId来标识哪个定时器触发。

函数原型：

```
virtual void OnStrategyTimer(int iTimerId, const char * szInstrumentID) {};
```

参数：

[out]iTimerId:定时器ID,用于标识是哪个定时器触发。

返回值:无

4.3.9 OnReady 方法

回调函数，当策略交易初始化完成时，该回调函数会被系统调用，可以在此函数做策略的初始化操作。

函数原型：

```
virtual void OnReady() {};
```

参数：无

返回值:无

4.3.10 SubcribeKindle 方法

订阅 k 线， 订阅 k 线后，对应的 Tick 行情会自动被订阅，并进行回调。

函数原型：

```
cwKindleSeriesPtr        SubcribeKindle(const char * szInstrumentID, int iTimeScale, int HisKindleCount = 0);
```

参数：

[in]szInstrumentID:要订阅的合约k线合约。

[in]iTimeScale:k线周期，以秒为单位，如1分钟k则填60,5分钟则填300，可以直接引用cwKINDLE_TIMESCALE定义的常见k线周期。

[in]HisKindleCount：这个是预留参数项，目前没用。

返回值：k线序列管理类。

4.3.11 SubcribeDailyKindle 方法

订阅日线 K 线， 订阅 k 线后，对应的 Tick 行情会自动被订阅，并进行回调。日线 K 线周期都默认为 86400 秒，实际处理的时候，按照对应品种的实际交易时间，合成对应的 k 线。在没有历史 k 线加载的情况，日线的 k 线应该是从开始收行情到当前的行情合成的 k 线。

函数原型：

```
cwKindleSeriesPtr        SubcribeDailyKindle(const char * szInstrumentID);
```

参数：

[in]szInstrumentID:要订阅的合约k线合约。

返回值：k线序列管理类。

4.3.12 GetKindleSeries 方法

获取已经订阅的 K 线，只要订阅过的 K 线，就可以通过该函数来获取。如果获取返回的值为 NULL，则代表

函数原型：

```
cwKindleSeriesPtr       GetKindleSeries(const char * szInstrumentID, int iTimeScale);
```

参数：

[in]szInstrumentID:要获取k线的合约。

[in]iTimeScale:要获取k线的周期

返回值：k 线序列管理类。

4.3.13 InputLimitOrder 方法

报单函数--限价单

函数原型：

```
cwOrderPtr              InputLimitOrder(const char * szInstrumentID,
cwFtdcDirectionType direction, cwOpenClose openclose, int volume, double
price);
```

参数：

[in]szInstrumentID:下单合约。

[in]direction:下单方向，可以填入：

买入：CW_FTDC_D_Buy（或者字符'0'）

卖出：CW_FTDC_D_Sell（或者字符'1'）

[in]openclose:订单开平模式，枚举定义如下：

```
enum  cwOpenClose
{
    cwOpen = 0          //开仓
    , cwClose           //平仓（平昨）
    , cwCloseToday      //平今
 };
```

[in]volume:下单数量，应为正整数。

[in]price:下单价格，应在对应合约的最小变动价位上。

返回值：下单的订单，订单引用（OrderRef）可以用来标记该订单。注：因为该订单还未收到柜台和交易所的返回信息，相关字段将处于无效状态。

4.3.14 InputFAKOrder 方法

报单函数--FAK 单（Filled And Kill 立即成交剩余自动撤销指令），该订单能成交多少，成交多少，不能成交部分将自动撤单。该订单不需要进行撤单操作，另外在当前交易所风控要求下，FAK 订单和 FOK 订单不纳入撤单数量限制，可以灵活配合使用。请注意不同交易所对该交易指令的支持情况。

函数原型：

```
cwOrderPtr              InputFAKOrder(const char * szInstrumentID,
cwFtdcDirectionType direction, cwOpenClose openclose, int volume, double price);
```

参数：

[in]szInstrumentID:下单合约。

[in]direction:下单方向，可以填入：

买入：CW_FTDC_D_Buy（或者字符'0'）

卖出：CW_FTDC_D_Sell（或者字符'1'）

[in]openclose:订单开平模式，枚举定义如下：

```
enum  cwOpenClose
{
    cwOpen = 0          //开仓
    , cwClose           //平仓（平昨）
    , cwCloseToday      //平今
 };
```

[in]volume:下单数量，应为正整数。

[in]price:下单价格，应在对应合约的最小变动价位上。

返回值：下单的订单，订单引用（OrderRef）可以用来标记该订单。注：因为该订单还未收到柜台和交易所的返回信息，相关字段将处于无效状态。

4.3.15 InputFOKOrder 方法

报单函数--FOK 单（Filled Or Kill 立即全部成交否则自动撤销指令），该订单要么全部成交，如不能成交将自动撤单。该订单不需要进行撤单操作，另外在当前交易所风控要求下，FAK 订单和 FOK 订单不纳入撤单数量限制，可以灵活配合使用。请注意不同交易所对该交易指令的支持情况。

函数原型：

```
cwOrderPtr              InputFOKOrder(const char * szInstrumentID,
cwFtdcDirectionType direction, cwOpenClose openclose, int volume, double price);
```

参数：

[in]szInstrumentID:下单合约。

[in]direction:下单方向，可以填入：

买入：CW_FTDC_D_Buy（或者字符'0'）

卖出：CW_FTDC_D_Sell（或者字符'1'）

[in]openclose:订单开平模式，枚举定义如下：

```
enum  cwOpenClose
{
    cwOpen = 0          //开仓
    , cwClose           //平仓（平昨）
    , cwCloseToday      //平今
 };
```

[in]volume:下单数量，应为正整数。

[in]price:下单价格，应在对应合约的最小变动价位上。

返回值：下单的订单，订单引用（OrderRef）可以用来标记该订单。注：因为该订单还未收到柜台和交易所的返回信息，相关字段将处于无效状态。

4.3.16 EasyInputOrder 方法

简化报单函数

函数原型：

```
cwOrderPtr              EasyInputOrder(const char * szInstrumentID, int volume, double price,
        cwOpenCloseMode openclosemode = cwOpenCloseMode::CloseTodayThenYd,
        cwInsertOrderType insertordertype = cwInsertOrderType::cwInsertLimitOrder);
```

参数：

[in]szInstrumentID:下单合约。

[in]volume:下单数量，非零整数。正数则做多，负数则做空

[in]openclosemode:简化开平模式，枚举定义如下：

```
enum cwOpenCloseMode :int
{
    CloseTodayThenYd = 0,       //先平今，再平昨,可开，用于平今免（或者无所谓）的品种
    OpenOnly = 1,               //只开
    CloseOnly = 2,              //只平 只会报出平仓部分报单，如果报单量大于持仓数，也只报出持仓数
    CloseYdThenOpen = 3,        //先平昨，后开仓，不平今，用于平今很贵的品种，弊病是要等全部平完再开仓,可能下的手数会小于实际报单数
    CloseYdOrOpen = 4,          //暂不支持，优先平昨，可开仓，开仓后不再平仓，用于平今很贵的品种，又不耽误开仓，弊病是有一点昨仓可能没平
    CloseYdThenToday = 5        //暂不支持，先平昨，再平今,可开，用于平昨便宜，平今和开仓差不多的品种
};
```

[in]insertordertype:订单类型，默认为限价单。枚举定义如下：

```
{
cwInsertLimitOrder = 0,    //限价单
cwInsertFAKOrder = 1,      //FAK Filled And Kill 立即成交剩余自动撤销指令
cwInsertFOKOrder = 2,      //FOK Filled Or Kill 立即全部成交否则自动撤销指令
}
```

返回值：下单的订单，订单引用（OrderRef）可以用来标记该订单。注：因为该订单还未收到柜台和交易所的返回信息，相关字段将处于无效状态。

4.3.17 EasyInputMultiOrder 方法

简化报单函数，该报单函数和 4.4.16 EasyInputOrder 的区别是，EasyInputOrder 每次调用只会下一个订单，这个函数可能会根据逻辑进行拆单，报出多个订单。

举个例子：如果为开仓 1500 手，限价单最大手数为 500,则会自动拆单为 3 笔下出。另一种情况：如 openclosemode 设置为 CloseTodayThenYd，买入 10 手，此时有 3 手空头持仓，会将报单拆成两个，一个 3 手平仓，一个 7 手开仓。以此类推。

函数原型：

```
std::deque<cwOrderPtr> EasyInputMultiOrder(const char * szInstrumentID, int volume, double price,
        cwOpenCloseMode openclosemode = cwOpenCloseMode::CloseTodayThenYd,
        cwInsertOrderType insertordertype = cwInsertOrderType::cwInsertLimitOrder);
```

参数：

[in]szInstrumentID:下单合约。

[in]volume:下单数量，非零整数。正数则做多，负数则做空

[in]openclosemode:简化开平模式，定义见4.3.16 EasyInputOrder

[in]insertordertype:订单类型，默认为限价单。枚举定义如下：

```
{
cwInsertLimitOrder = 0,    //限价单
cwInsertFAKOrder = 1,      //FAK Filled And Kill 立即成交剩余自动撤销指令
cwInsertFOKOrder = 2,      //FOK Filled Or Kill 立即全部成交否则自动撤销指令
}
```

返回值：下单的订单，订单引用（OrderRef）可以用来标记该订单。注：因为该订单还未收到柜台和交易所的返回信息，相关字段将处于无效状态。

4.3.18 CancelOrder 方法

撤单函数

函数原型：

```
bool                    CancelOrder(cwOrderPtr pOrder);
```

参数：

[in]pOrder:需要撤单的订单。建议从系统中获取的订单或者是下单后返回的订单；不建议自行构造。

返回值：成功返回 true,否则返回 false。注：成功失败，只作为发出撤单请求，撤单是否成功，无法通过该函数返回值判断，订单状态请根据订单回报来处理。

4.3.19 CancelAll 方法

对该账户下所有的处于挂单状态的订单发起撤单。

函数原型：

```
int                     CancelAll();
```

参数：

无

返回值：成功发起撤单的订单数量。注：成功失败，只作为发出撤单请求，撤单是否成功，无法通过该函数返回值判断，订单状态请根据订单回报来处理。

4.3.20 CancelAll 方法

对该账户下指定合约所有的处于挂单状态的订单发起撤单。

函数原型：

```
int                     CancelAll(const char * szInstrumentID);
```

参数：

[in]szInstrumentID:需要发起撤单合约。

返回值：成功发起撤单的订单数量。注：成功失败，只作为发出撤单请求，撤单是否成功，无法通过该函数返回值判断，订单状态请根据订单回报来处理。

4.3.21 CancelAll 方法

对该账户下指定合约和方向的所有的处于挂单状态的订单发起撤单。

函数原型：

```
int                     CancelAll(const char * szInstrumentID,
cwFtdcDirectionType direction);
```

参数：

[in]szInstrumentID:需要发起撤单合约。

[in]direction:下单方向，可以填入：

买入：CW_FTDC_D_Buy（或者字符'0'）

卖出：CW_FTDC_D_Sell（或者字符'1'）

返回值：成功发起撤单的订单数量。注：成功失败，只作为发出撤单请求，撤单是否成功，无法通过该函数返回值判断，订单状态请根据订单回报来处理。

5. 第五章 附录

5.1 合约信息文件

Instrument.xml

5.2 联系方式

感谢您对 PandoraTrader 的关注，如有任何疑问和建议，请不要迟疑，马上和作者取得联系。

QQ 群：615093081

Email: pandoratrader@163.com

6. 第六章 鸣谢

1. 感谢李日旺先生对项目参与及文档编写工作
2. 感谢江伟博士对于回测相关的文档贡献。