

介绍

Xele-Trade-Futures介绍

Xele-Trade-Futures是业内领先的基于FPGA、微秒级极速交易系统，独创完整流程的FPGA数据传输系统，拥有纳秒级响应速度，提供最快速准确的资讯通道；是为证券、期货高端人士及机构、基金类专业投资机构及产业巨头量身打造的高性能交易系统。

该文档是极速交易系统Xele-Trade-Futures的投资者使用手册，它提供API功能及其使用说明，展示投资者开发客户端程序(Client Program)的通用步骤。

TraderAPI简介

TraderAPI用于与Xele-Trade-Futures进行通信。通过API，投资者可以向多个交易所（SHFE，CFFEX，INE，CZCE，DCE）发送交易指令，获得相应的回复和交易状态回报。

TraderAPI 是一个基于 C++ 的类库，通过使用和扩展类库提供的接口来实现全部的交易功能。发布给用户的文件包以**xele-futures-trader-api-OS-**版本号.tar.gz**** 命名，包含的文件如下：

文件名	说明
include/XTFApi.h	API头文件, 定义了目前支持的接口
include/XTFDataStruct.h	定义了函数接口的参数结构体类型
include/XTFDataType.h	定义了使用到的字段的类型及常量
libxele_futures_trader_api.so	Linux 64位版本的API动态库
ExampleXX.cpp	示例程序的源代码文件
xtf_trader_api.config	示例的配置文件
Makefile	示例程序的编译文件

TraderAPI发行平台

目前发布的主要基于**Linux 64位**操作系统的版本，包括动态库.so文件和.h头文件；

备注说明: 如果客户程序使用**API**出现异常奔溃，这并不表示一定是**API**自身异常，但**API**会捕捉该异常信号，并记录异常调用栈信息协助定位，生成的异常文件一般在客户程序所在目录，命名格式为**xeleapi-year-month-day-hour-minute-second.coredump**（例如**xeleapi-2020-09-11-16_46_02.coredump**），生成的**codrdump**文件大小限制最大为**8G**；

TraderAPI修改历史

日期	API版本	API主要变更说明
2022/07/18	V4.1.200	API接口、结构体全新改版
2022/09/06	V4.1.394	增加查询报单和查询成交的接口，调整部分数据结构
2022/09/13	V4.1.413	调整报单通知接口onOrder； 增加撤单通知接口onCancelOrder； 修改onBookUpdate接口的参数类型；
2022/09/15	V4.1.425	XTFOrder启用insertTime字段； XTFOrderFilter和XTFTradeFilter增加时间字段含义的说明；
2022/09/19	V4.1.437	XTFAccount增加lastLocalOrderID字段； XTFExchange增加tradingDay字段
2022/09/20	V4.1.441	增加柜台重启通知接口onServerReboot；
2022/09/21	V4.1.445	XTFInputOrder增加minVolume字段，XTFOrder增加orderMinVolume字段，用于实现最小数量成交的FAK报单；
2022/09/26	V4.1.454	增加回报数据处理线程BusyLoop启用开关配置项； 增加交易通道IP地址查询功能； XTFExchange结构体增加hasChannelIP字段用于标记是否支持IP地址查询； XTFAccount结构体增加lastLocalActionID字段，表示用户定义的最后一次本地撤单编号；

备注说明：

1. 新版本的**API**接口不是向前兼容的，如有特殊情况，详见发布说明；

TraderAPI运行机制

TraderAPI接口命名

TraderAPI提供了2类接口，分别为**XTFSpi**和**XTFApi**。

请求

int XTFApi::XXX()

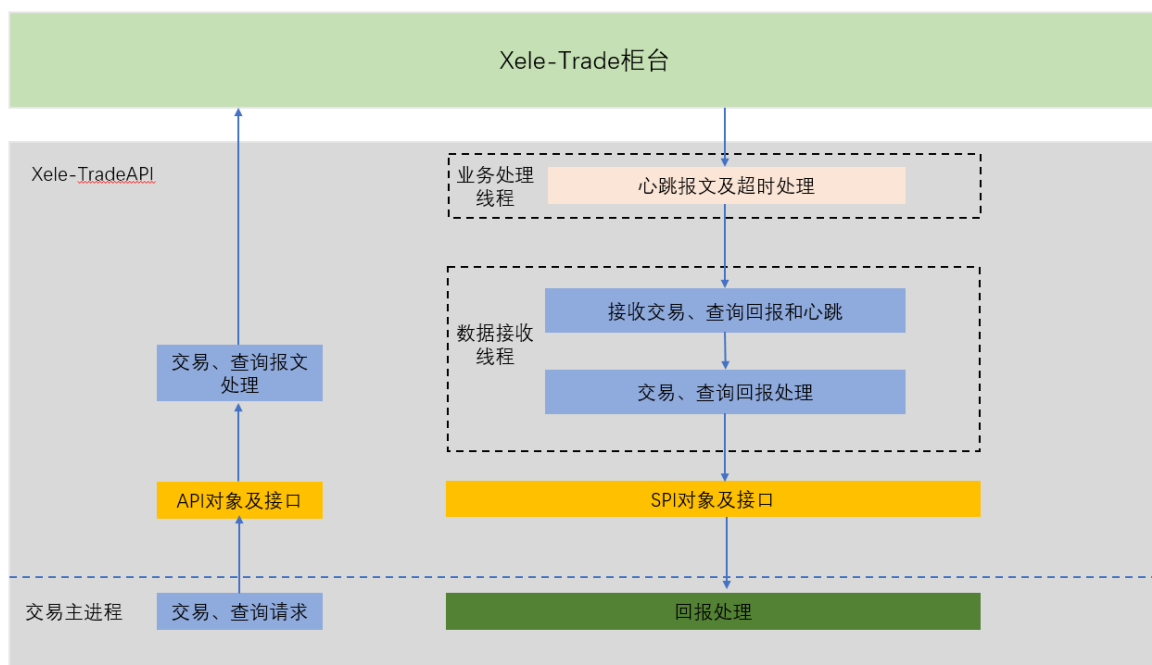
回应：

void XTFSpi::onXXX ()

该模式的相关接口以XXX，onXXX来命名。

工作线程

TraderAPI主要提供API 请求接口供用户发送请求命令，提供SPI回调接口供用户接收回报消息。API逻辑结构如下图所示：



用户客户端通过API的请求接口可以向柜台发送相关请求，通过继承API的SPI回调函数，可以接收到相关响应和回报。

客户端程序和交易前端之间的通信是由API工作线程驱动的，客户端程序至少存在3个线程组成：

客户端主线程：主要是客户端发送请求和处理接收到的API的回报信息；

API数据接收线程：主要是接收和处理柜台系统的交易回报和查询响应；

API报单处理线程：主要是处理交易预热功能，建议与用户报单线程绑在同一CPU核；

API业务处理线程：主要是处理公共业务、心跳等其他业务逻辑；

所有线程均支持绑核操作。

备注说明：

1. 由XTFApi和XTFSpi提供的接口**不是线程安全的**。多个线程同时为单个XTFApi实例发送请求是不允许的。
2. 用户注册的SPI回调函数需要尽快处理相关数据，如果SPI回调接口阻塞了API的查询回报/交易回报处理线程，**将会影响该处理线程正常接收后续的数据**。
3. 支持创建多个XTFApi实例，每个实例分别使用独立的SPI对象处理。建议同一进程内，API实例数量不超过4个。

API配置

API的配置需要在启动之前进行参数的设置，启动之后设置的参数不会生效

参数设置如下：

ACCOUNT_ID：资金账户ID

ACCOUNT_PWD：资金账户密码

APP_ID：应用程序ID

AUTH_CODE：认证授权码

QUERY_SERVER_IP：查询服务地址

QUERY_SERVER_PORT：查询服务端口

TRADE_SERVER_IP：交易服务地址

TRADE_SERVER_PORT：交易服务端口

WARM_INTERVAL：预热处理间隔，取值范围[10, 50]，单位：毫秒

参考配置: WARM_INTERVAL=20

TCP_WORKER_CORE_ID：TCP数据处理线程绑核，默认不绑核。建议用户绑核

参考配置: TCP_WORKER_CORE_ID=2

TRADE_WORKER_CORE_ID：预热处理线程绑核，默认不绑核。建议将预热线程与用户策略线程绑在同一个物理核

参考配置: TRADE_WORKER_CORE_ID=4

TASK_WORKER_CORE_ID：公共处理线程绑核，默认不绑核。建议可不绑核。

参考配置: TASK_WORKER_CORE_ID=6

注意：用户也可以设置自己的参数，以便在需要的地方使用 getConfig() 接口进行获取。

心跳机制

Xele-Trade-Futures柜台系统在登录成功后，每隔一定时间发送心跳报文给客户端来维持连接，API也会定时发送心跳报文来维持链接，默认间隔为3s，超时时间可配（API配置文件），API和柜台系统之间的心跳自动维护，无需用户干预。

工作流程

客户端和柜台交易系统的交互过程分为2个阶段：初始化阶段和功能调用阶段。

初始化阶段

在初始化阶段，Xele-Trade-Futures交易系统的程序必须完成如下步骤：

start():调用该接口，API会自动向交易柜台发起连接。

交易柜台的配置信息，默认从创建API对象的配置文件中读取。

也可以通过 setConfig() 接口配置。

示例代码：API初始化流程（具体代码详见demo中“userdemo.cpp”的620-627行）：

```
void start() {
    if (mApi) {
        printError("trader has been started.");
        return;
    }

    mOrderLocalId = 0;
    mApi = makeXTFApi(mConfigPath.c_str());
    if (mApi == nullptr) {
        printError("create xtf api failed.");
        return;
    }

    printInfo("api version: %s.", getXTFVersion());
    int ret = mApi->start(this);
    if (ret != 0) {
        printf("api start failed, error code: %d\n", ret);
    }
}

void startExample1(const std::string &configPath,
                  const std::string &username,
                  const std::string &password,
                  const std::string &instrumentId) {
    printInfo("start example 01.");
    Example_01_Trader trader;

    /* 设置相关API配置 */
    trader.setConfigPath(configPath);
    trader.setUsername(username);
    trader.setPassword(password);
    trader.setAppID("xtf-example-1.0");
    trader.setAuthCode("ABCDEFGHijklmnOPQrstUVWxyz");
```

```

trader.setInstrumentID(instrumentId);

/* 调用API接口，注意每个API接口的调用都应该检查其返回值，非0值表示存在错误。 */
trader.start();
sleep(10);
trader.insertOrder();
sleep(10);
trader.stop();
}

```

功能调用阶段

在功能调用阶段，客户端程序可以任意调用交易接口中的请求方法，如：insertOrder、cancelOrder、findOrders等，同时用户需要继承SPI回调函数以获取响应回报信息。

示例代码：功能调用及回调函数（具体代码详见example中“Example01.cpp”）

```

/* 客户端程序和柜台建立链接后，调用报单操作接口*/
/* 注意每个API接口的调用都应该检查其返回值，非0值表示存在错误。 */
trader.start();
sleep(1);
...
trader.insertOrder();
...
trader.stop();

```

TraderAPI基本操作说明

Xele-Trade-Futures柜台系统包括**数据服务**和**交易服务**：

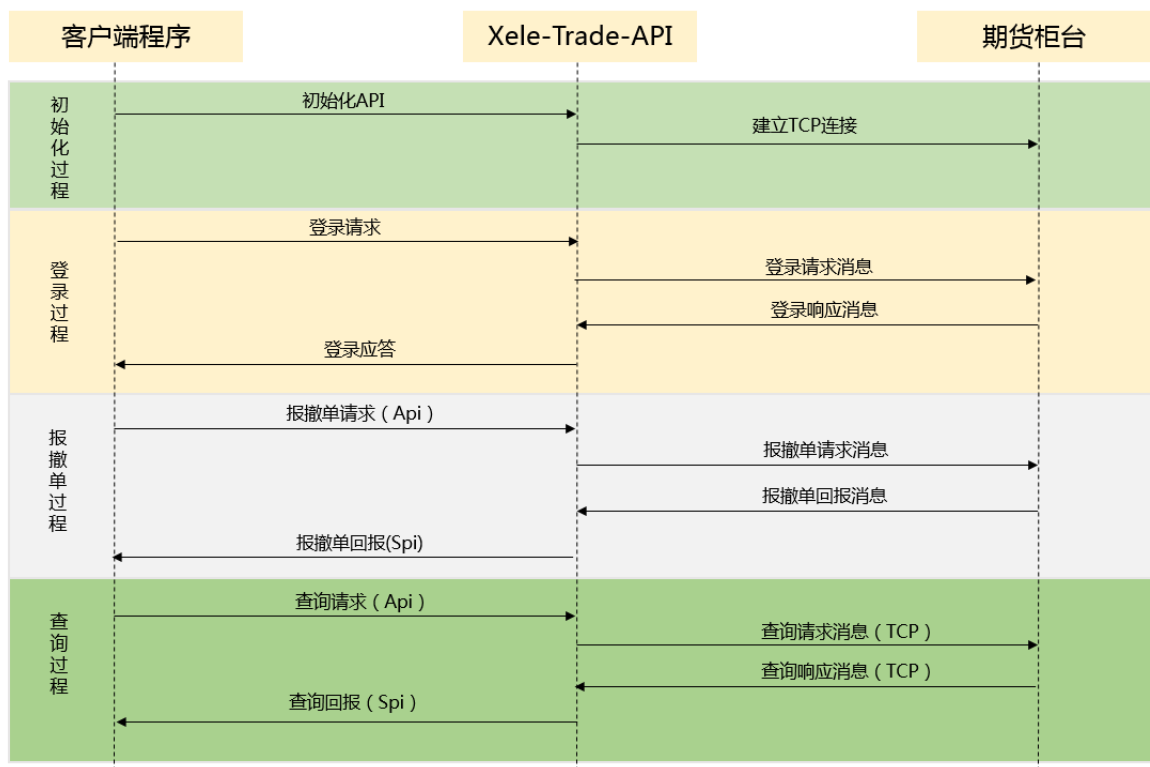
数据服务：主要是提供相关报单、合约、资金等查询以及接收相关回报功能；

交易服务：主要是处理报、撤单等功能。

对于客户端来讲，用户需要通过API的login接口登录柜台，登录成功后才可以调用insertOrder或者cancelOrder接口进行报撤单操作。

TraderAPI和Xele-Trade-Futures柜台系统的信息交互主要包括：登录登出、报撤单操作、查询操作。

TraderAPI和柜台系统的上述信息交互流程如下图所示：



登录

初始化网络连接后，就可以通过查询流接口login进行登录请求了，调用onLogin回调接口处理登录响应，只有登录成功后才能进行业务处理。

Xele-Trade-Futures柜台系统支持每个用户**多点登录**（默认最多是5个子账户登录），同一个用户每次可创建多个API实例，初始化完成后即可以进行登录。多点登录有以下特点：

- 每个登录点都是独立的；且都需要登录成功后，才可以进行报撤单以及查询操作；
- 查询数据流的下行报文只发送给发起请求的登录点的链接；
- 交易数据流的下行报文是发送给该用户的所有登录点的链接；
- 每个登录点的查询数据流或交易数据流上下行链接如果断开，API均会自动重连；
- 如果用户希望重新开始操作，则需要退出所有登录点，然后再进行登录操作；

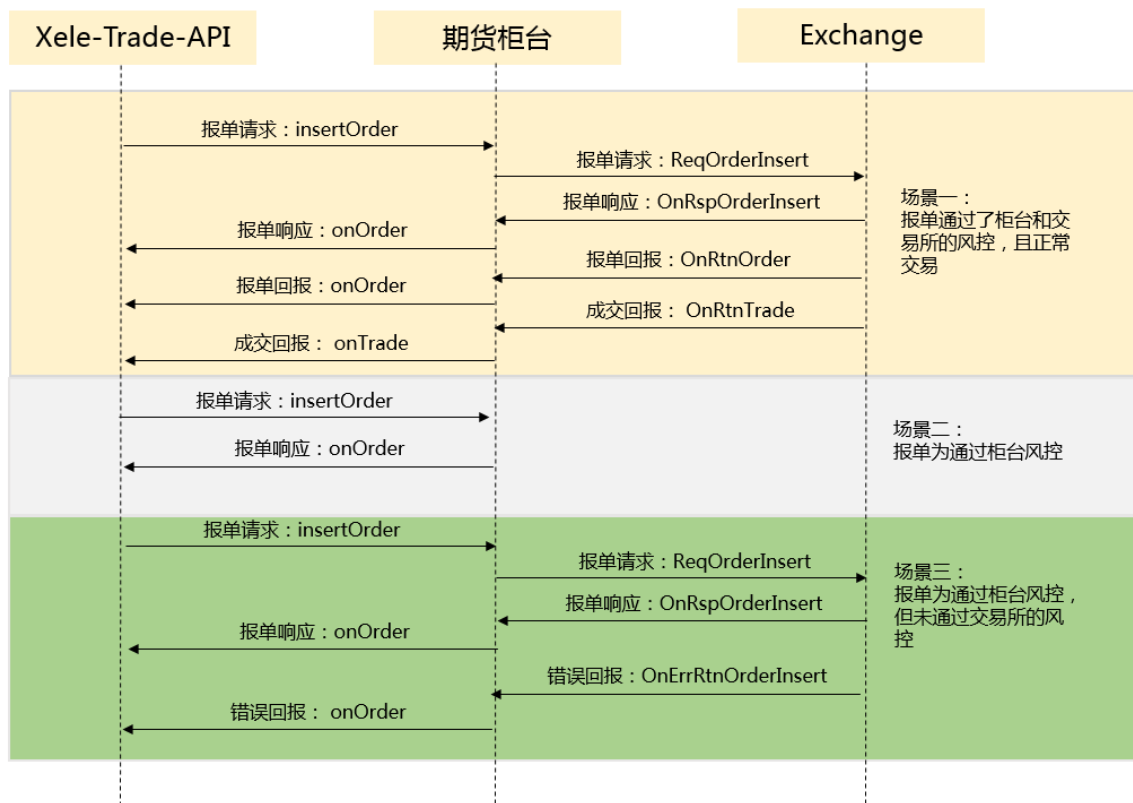
示例代码：用户登录（具体代码详见example中“Example01.cpp”）：

```
void onStart(int errorCode, bool isFirstTime) override {
    if (errorCode == 0) {
        if (isFirstTime) {
            // TODO: init something if needed.
        }
        /* 调用登录接口 */
        int ret = mApi->login(mUsername.c_str(), mPassword.c_str(),
mAppID.c_str(), mAuthCode.c_str());
        if (ret != 0) {
            printError("api login failed, error code: %d", ret);
        }
    } else {
        printError("api start failed, error code: %d.", errorCode);
    }
}

void onLogin(int errorCode, int exchangeCount) override {
    if (errorCode != 0) {
        printError("login failed, error code: %d.", errorCode);
        return;
    }
}
```

```
printInfo("login success.");
}
```

报单

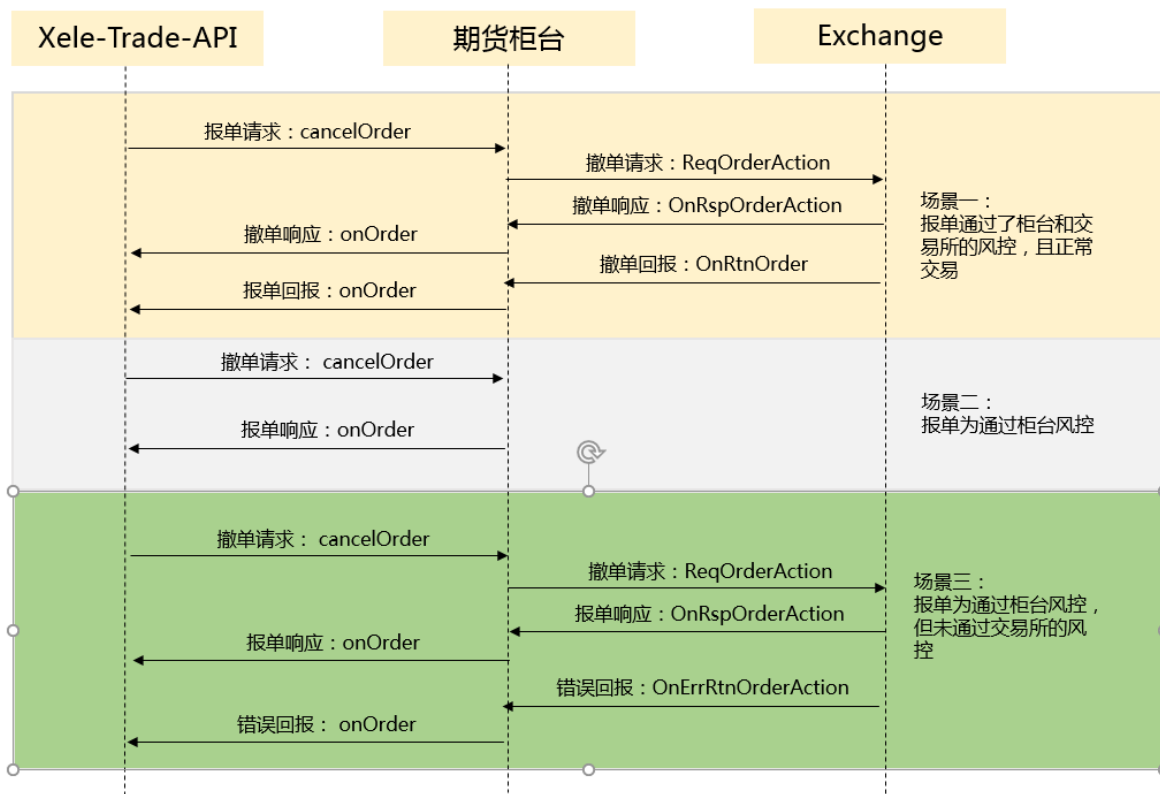


如上图所示，报单的流程主要有上述三种场景。

报单的报文说明：

- 不管哪种场景，柜台系统都会发送报单响应，API通过onOrder接口通知用户；
- 当报单成功进入交易所后，报单发生的任何状态变化，柜台系统都会发送报单回报，API通过onOrder接口通知用户；
- 当报单成功进入交易所后，报单发生的任何成交，柜台都会发送成交回报，API通过onTrade接口通知用户；

撤单



如上图所示，撤单的流程主要有上述三种场景。

撤单的报文说明：

- 不管哪种场景，柜台系统都会发送撤单响应，API通过onOrder或onCancelOrder接口通知用户；
- 当撤单未通过柜台系统或者交易所风控时，柜台系统都会发送错误回报，API通过onOrder接口通知用户；
- 当撤单成功进入交易所后，撤单发生的任何状态变化，柜台系统都会发送回报。当报单被撤销后，API通过onCancelOrder接口通知用户；

示例代码：报撤单示例代码如下

```

void onReadyForTrading(const XTFAccount *account) override {
    if (mApi == nullptr) {
        return;
    }

    // 静态数据初始化完毕后回调，传递对象指针，是为了用户保存指针对象，方便使用。
    // 这里可以开始做交易。
    mInstrument = mApi->getInstrumentByID(mInstrumentID.c_str());
    if (mInstrument == nullptr) {
        logError("instrument not found: %s.", mInstrumentID.c_str());
    }
}

void onLoadFinished(const XTFAccount *account) override {
    // 流水数据追平后回调
}

void onOrder(int errorCode, const XTFOder *order) override {
    // 报撤单失败。根据报单错误码判断是柜台拒单，还是交易所拒单。
    if (errorCode != 0) {
        switch (order->actionType) {
            case XTF_OA_Insert:
                printError("insert order failed.");
                break;
            case XTF_OA_Cancel:

```



```

        printError("cancel order failed.");
        break;
    default:
        printError("order action(%d) failed, error code: %d", order-
>actionType, errorCode);
        break;
    }
    return;
}

// 收到报单回报。根据报单状态判断是报单还是撤单。
switch (order->orderStatus) {
    case XTF_OS_Accepted:
        printInfo("order accepted.");
        break;
    case XTF_OS_AllTraded:
        printInfo("order all traded.");
        break;
    case XTF_OS_Canceled:
        printInfo("order canceled.");
        break;
    case XTF_OS_Queueing:
        printInfo("order queueing.");
        break;
    case XTF_OS_Rejected:
        printInfo("order rejected.");
        break;
    default:
        break;
}
}

void onTrade(const XTFTTrade *trade) override {
    // 收到交易回报
}

//撤单
int cancelOrder(int sysOrderID) {
    int ret = mApi->cancelOrder(XTF_OIDT_System, sysOrderID);
    if (ret != 0) {
        printError("cancel order failed, sysOrderID: %d, error code: %d",
sysOrderID, ret);
    }
    return ret;
}

//报单
int insertOrder() {
    int ret = openLong(1);
    if (ret != 0) {
        printError("open long order failed, error code: %d", ret);
    }
    return ret;
}

int openLong(int volume) {
    if (mApi == nullptr) return -1;
    XTFInputOrder order;

```

```
memset(&order, 0, sizeof(order));
order.instrument = mInstrument;
order.direction = XTF_D_Buy;
order.offsetFlag = XTF_OF_Open;
order.orderType = XTF_ODT_FOK;
order.price = 1000;
order.volume = volume;
order.channelSelectionType = XTF_CS_Auto;
order.orderLocalID = ++mOrderLocalId;
return mApi->insertOrder(order);
}
```

TraderAPI接口参考说明

TraderAPI接口

业务类型	业务	请求接口 / 响应接口
生命期管理接口	API对象启动通知接口	XTFApi::start XTFSpi::onStart
生命期管理接口	API对象停止通知接口	XTFApi::stop XTFSpi::onStop
会话管理接口	登录结果通知接口	XTFApi::login XTFSpi::onLogin
会话管理接口	登出结果通知接口	XTFApi::logout XTFSpi::onLogout
会话管理接口	修改密码结果通知接口	XTFApi::changePassword XTFSpi::onChangePassword
会话管理接口	日初静态数据已加载完毕，可进行报单操作。	XTFSpi::onReadyForTrading XTFSpi::onLoadFinished
交易接口	报单回报及订单状态改变通知接口	XTFApi::insertOrder XTFSpi::onOrder
交易接口	撤单及订单状态改变通知接口	XTFApi::cancelOrder XTFSpi::onCancelOrder
交易接口	成交回报通知接口	XTFSpi::onTrade
数据变化通知接口	账户发生变化时回调该接口	XTFSpi::onAccount
数据变化通知接口	交易所前置状态发生变化时回调该接口（暂未使用）	XTFSpi::onExchange
数据变化通知接口	合约发生变化时回调该接口（暂未使用）	XTFSpi::onInstrument
外接行情接口	行情发生变化时回调该接口	XTFApi::subscribe XTFSpi::onBookUpdate
外接行情接口	取消订阅行情	XTFApi::unsubscribe
外接行情接口	更新合约行情	XTFApi::updateBook
查询管理接口	获取当前资金账户信息	XTFApi::getAccount
查询管理接口	获取交易所信息	XTFApi::getExchangeCount
查询管理接口	获取合约信息	XTFApi::getInstrumentCount
查询管理接口	查找报单信息	XTFApi::findOrders

业务类型	业务	请求接口 / 响应接口
查询管理接口	查找成交信息	XTFApi::findTrades
参数配置管理接口	设置会话参数	XTFApi::setConfig
参数配置管理接口	查询会话参数	XTFApi::getConfig
参数配置管理接口	获取API版本	XTFApi::getVersion
其他通用接口	事件通知接口（暂未使用）	XTFSpi::onEvent
其他通用接口	其他错误通知接口（暂未使用）	XTFSpi::onError

SPI类管理接口

生命期管理接口

onStart方法

功能：API对象启动通知接口

函数原形：

```
void onStart(int errorCode, bool isFirstTime);
```

参数：

- errorCode：启动结果。
0-表示启动成功，可以调用 login() 接口登录柜台；负数表示启动失败对应的错误码
- isFirstTime: 是否初次打开。

可以据此标志位，判断在初次打开时，进行数据的初始化等操作。

返回值：空

onStop方法

功能：API对象停止通知接口

函数原形：

```
void onStop(int errorCode) {}
```

参数：0-表示停止成功；X-表示停止失败对应的错误码；

返回值：无

onServerReboot方法

功能：柜台在交易时段发生重启的通知接口

函数原形：

```
void onServerReboot();
```

参数：0-表示停止成功；X-表示停止失败对应的错误码；

返回值：无

说明：

- 柜台在交易时段如果发生重启，API中断后会自动重连；
- 重新连接后，API会清空上一次登录的所有数据，并重新从柜台加载数据；
- 由于API的本地数据发生了变化，因此所有外部使用的指针数据会失效，用户需要在收到onServerReboot事件后，清理上一次的所有数据指针，此刻这些数据指针依然有效；当onServerReboot事件处理之后，数据指针将会失效。

会话管理接口

onLogin方法

功能：登录结果通知接口

函数原形：

```
void onLogin(int errorCode, int exchangeCount);
```

参数：

errorCode：登录结果。0-表示登录成功；X-表示登录失败返回的错误码；

exchangeCount：交易所数量，用户可以根据该值，获取交易所的信息。

返回值：无

onLogout 方法

功能：登出结果通知接口

函数原形：

```
void onLogout(int errorCode);
```

参数：errorCode：登出结果。0-表示登录成功；X-表示登录失败返回的错误码；

返回值：无

onChangePassword方法

功能：修改密码结果通知接口

函数原形：

```
void onChangePassword(int errorCode)
```

参数：errorCode：修改结果。0-表示修改成功；X-表示修改失败返回的错误码；

返回值：无；

onReadyForTrading 方法

功能：日初静态数据已加载完毕，可进行报单操作。

函数原形：

```
void onReadyForTrading(const XTFAccount *account);
```

参数：account：当前登录的账户信息

XTFAccount结构体：客户/账户资金信息对象类

```
class XTFAccount {
public:
    XTFAccountID    accountID;           ///< 账户编码
    double          preBalance;          ///< 昨日余额
    double          deposit;             ///< 今日入金
    double          withdraw;            ///< 今日出金
    double          commission;          ///< 手续费
    double          frozenCommission;     ///< 冻结的手续费
    double          margin;              ///< 保证金
    double          frozenMargin;        ///< 冻结的保证金
    double          premium;             ///< 权利金
    double          frozenPremium;       ///< 冻结的权利金
    double          balance;             ///< 资金余额
    double          available;           ///< 可用资金
    double          availableRatio;      ///< 资金可用限度
    double          positionProfit;      ///< 持仓盈亏，所有合约的持仓盈亏
    之和
    double          closeProfit;         ///< 平仓盈亏，所有合约的平仓盈亏
    之和
    XTFLocalOrderID lastLocalOrderID;    ///< 用户最后一次报单的本地编号
    XTFLocalActionID lastLocalActionID;  ///< 用户最后一次撤单的本地编号
    如果使用API接口进行撤单，
    API内部会从(0xF0000001)开始逐步自增生成本地撤单编号。
    用户可以通过判断最高位来确定
    是否为自定义的本地撤单编号。
};
```

返回值：无；

onLoadFinished方法

功能：所有流水数据已加载完毕，可进行资金和持仓计算。

函数原形：

```
void onLoadFinished(const XTFAccount *account)
```

参数：account：当前登录的账户信息

XTFAccount结构体：客户/账户资金信息对象类，同上。

返回值：无；

说明：

- 1.流水数据支持断点续传功能；
- 2.如果是新创建的API对象，流水数据从头开始推送；
- 3.如果API对象已存在，连接断开后重连，流水数据从断开处续传推送

交易接口

onOrder方法

功能：报撤单回报及订单状态改变通知接口

函数原形：

```
void onOrder(int errorCode, const XTFOOrder *order);
```

参数：

- errorCode：报单操作错误码
 - order：报单回报对象
- XTFOOrder结构体：报单回报对象类

```
class XTFOOrder {
public:
    XTFSysOrderID      sysOrderID;          ///< 柜台流水号
    XTFLocalOrderID    localOrderID;        ///< 用户填写的本地报单号，必须保证唯一性，否则会产生回报错误
                                                    ///< 柜台清流启动后的历史报单固定为0xd8888888，非本柜台报单固定为0x88888888，
                                                    ///< 错误为1172“请求中的报单编号不存在”时，返回0，其他情况返回订单真实编号
    XTFXExchangeOrderID exchangeOrderID;    ///< 交易所报单编号
    XTFXDirection      direction;           ///< 买卖方向
    XTFXOffsetFlag      offsetFlag;         ///< 开平仓标志
    double              orderPrice;         ///< 报单价格
    uint32_t             orderVolume;       ///< 报单数量
    uint32_t             orderMinVolume;    ///< 最小成交数量
    uint32_t             totalTradedVolume; ///< 报单累计已成交数量
    XTFOOrderType        orderType;         ///< 限价/FAK/市价类型
    XTFOOrderFlag        orderFlag;         ///< 报单标志
    XTFXChannelSelectionType channelSelectionType; ///< 席位连接选择
    uint8_t              channelID;         ///< 席位连接编号，0xFF表示无效值
    uint8_t              realChannelID;     ///< 实际席位连接编号，由柜台返回，0xFF表示无效值
    XTFOOrderStatus      orderStatus;       ///< 报单状态
    XTFXDate             insertDate;        ///< 报单插入日期，字符串格式：20220915
    XTFXTime             insertTime;        ///< 报单插入时间，字符串格式：10:20:30
    XTFXTime             updateTime;        ///< 报单更新时间，字符串格式：10:20:30
    XTFXTime             cancelTime;        ///< 报单撤单时间，字符串格式：10:20:30
    bool                 isHistory;         ///< 回报链路断开重连后或者程序重启后，客户端API会自动进行流水重构，
                                                    ///< 在追平服务器流水之前收到的报单回报，该字段为true。追平流水之后，该字段为false。
                                                    ///< 如对流水重构的回报不需要特殊处理，可不用处理该字段。
    bool                 isSuspended;       ///< 报单是否挂起（暂未使用）
    XTFX_CONST XTFXInstrument *instrument;  ///< 所属XTFXInstrument的指针。
    如果报单传入的合约不存在，那么合约对象指针可能为空。
```

```
XTFOrderActionType      actionType;          ///< 报单对象对应的操作类型（比如：报单、撤单、挂起、激活等），只读字段，外部调用不需要设置该字段。};
```

返回值：无；

onCancelOrder方法

功能：撤单回报状态改变通知接口

函数原形：

```
void onCancelOrder(int errorCode, const XTFOrder *order);
```

参数：

- errorCode：报单操作错误码
- order：报单回报对象

XTFOrder结构体：报单回报对象类，与onOrder方法相同。

API接收到报单回报，如果报单状态是 被撤销(XTF_OS_Canceled) 时，都通过onCancelOrder()接口通知用户，不再通过onOrder()接口通知。

可以通过重载该接口，添加报单被撤后的处理逻辑。

返回值：无；

onTrade方法

功能：成交回报通知接口

函数原形：

```
void onTrade(const XTFTTrade *trade);
```

参数：

- trade : 成交回报对象

XTFTrade结构体：成交回报对象类

```

class XTFTTrade {
public:
    XTFTTradeID            tradeID;           ///< 交易所成交编码
    double                 tradePrice;        ///< 成交价格
    uint32_t               tradeVolume;      ///< 本次回报已成交数量
    double                 margin;           ///< 保证金
    double                 commission;       ///< 手续费
    XTFTTime               tradeTime;        ///< 报单成交时间，字符串格式：
10:20:30
    XTFDirection           direction;        ///< 买卖方向,详情参考
    XTFDDataType.h         XTFOffsetFlag     ///< 开平仓标志,详情参考
    XTFDDataType.h         bool             isHistory;      ///< 回报链路断开重连后或者程序重
    启后，客户端API会自动进行流水重构，
    ///< 在追平服务器流水之前收到的成
    交回报，该字段为true。追平流水之后，该字段为false。
    ///< 如对流水重构的回报不需要特殊
    处理，可不用处理该字段。
}

```



```

        XTF_CONST XTFOrder      *order;          ///< 所属XTFOrder的指针
        XTF_CONST XTFTrade      *matchTrade;      ///< 对手成交回报，如果不为空则表明自成交

        bool                    isSelfTraded() const { return matchTrade != nullptr; }
    } ///< 判断是否为自成交
};

```

返回值：无；

数据变化通知接口

onAccount方法

功能：账户资金发生变化时回调该接口

函数原形：

```

void onAccount(int event, int action, const XTFAccount *account) {}

```

参数：

- event：账户变化事件类型（详见枚举类型章节）
XTF_EVT_AccountCashInOut 账户资金发生出入金流水变化

- Action:账户变化的动作

XTF_CASH_In:入金

XTF_Cash_Out:出金

- account：账户对象

XTFAccount结构体：客户/账户资金信息对象类

```

class XTFAccount {
public:
    XTFAccountID    accountID;          ///< 账户编码
    double          preBalance;         ///< 昨日余额
    double          deposit;            ///< 今日入金
    double          withdraw;           ///< 今日出金
    double          commission;         ///< 手续费
    double          frozenCommission;   ///< 冻结的手续费
    double          margin;             ///< 保证金
    double          frozenMargin;       ///< 冻结的保证金
    double          premium;            ///< 权利金
    double          frozenPremium;      ///< 冻结的权利金
    double          balance;            ///< 资金余额
    double          available;          ///< 可用资金
    double          availableRatio;     ///< 资金可用限度
    double          positionProfit;     ///< 持仓盈亏，所有合约的持仓盈亏之和
    double          closeProfit;        ///< 平仓盈亏，所有合约的平仓盈亏之和
    XTFLocalOrderID lastLocalOrderID;   ///< 用户最后一次报单的本地编号
    XTFLocalActionID lastLocalActionID;  ///< 用户最后一次撤单的本地编号
    ///< 如果使用API接口进行撤单，
    ///< 用户可以通过判断最高位来确定
    ///< 是否为自定义的本地撤单编号。
};

```

返回值：无；

onExchange方法

功能：交易所前置状态发生变化时回调该接口

函数原形：

```
void onExchange(int event, int channelId, const XTFExchange *exchange);
```

参数：

- **event**：账户变化事件类型（详见枚举类型章节）
XTF_EVT_ExchangeChannelConnected：交易所交易通道已连接通知
XTF_EVT_ExchangeChannelDisconnected：交易所交易通道已断开通知
- **channelID**：通道号
- **exchange**：交易所对象，请参考[XTFExchange结构体](#)部分。

返回值：无；

onInstrument方法

功能：合约发生变化时回调该接口

函数原形：

```
void onInstrument(int event, const XTFInstrument *instrument);
```

参数：

- **event**：账户变化事件类型（详见枚举类型章节）
XTF_EVT_InstrumentStatusChanged：合约状态发生变化通知
- **instrument**：合约对象

XTFInstrument结构体：合约对象类

```
class XTFInstrument {
public:
    XTFInstrumentID instrumentID;    ///< 合约代码
    uint32_t instrumentIndex;    ///< 合约序号
    int deliveryYear;    ///< 交割年份
    int deliveryMonth;    ///< 交割月份
    int maxMarketOrderVolume;    ///< 市价报单的最大报单量
    int minMarketOrderVolume;    ///< 市价报单的最小报单量
    int maxLimitOrderVolume;    ///< 限价报单的最大报单量
    int minLimitOrderVolume;    ///< 限价报单的最小报单量
    double priceTick;    ///< 最小变动价位
    int multiple;    ///< 合约单位
    XTFDate expireDate;    ///< 合约到期日，字符串格式：20220915
    bool singleSideMargin;    ///< 是否单边计算保证金
    XTFInstrumentStatus status;    ///< 当前状态
};
```

返回值：无；

外接行情接口

onBookUpdate方法

功能：行情发生变化时回调该接口

函数原形：

```
void onBookUpdate(const XTFMarketData &marketData);
```

参数：

- **marketData**：行情信息对象

XTFMarketData结构体：行情信息对象类

```
class XTFMarketData {
public:
    int      tradingDay;      ///< 交易日
    double    preSettlementPrice; ///< 前结算价
    double    preClosePrice;   ///< 前收盘价
    double    preOpenInterest;  ///< 前持仓量
    double    upperLimitPrice;  ///< 涨停板价
    double    lowerLimitPrice;  ///< 跌停板价
    double    lastPrice;        ///< 最新价
    double    bidPrice;         ///< 买入价。为零代表无买入价。
    int       bidVolume;        ///< 买入量。为零代表无买入价。
    double    askPrice;         ///< 卖出价。为零代表无卖出价。
    int       askVolume;        ///< 卖出量。为零代表无卖出价。
    double    turnover;         ///< 成交金额。
    double    openInterest;     ///< 持仓量。
    int       volume;           ///< 成交量。
    int       timeStamp;        ///< 时间戳，是整数，表示从该交易日开始到此时间的毫秒数。每个交易日
    从17点开始，即17:00:00对应时间戳0。周一的时间和周六的不区分。
    double    averagePrice;     ///< 行情均价
    CONST XTFInstrument *instrument; ///< 所属XTFInstrument的指针
};
```

返回值：无；

其他通用接口

onEvent方法

功能：事件通知接口

函数原形：

```
void onEvent(int type, void *data, size_t size);
```

参数：

- **data**：附带的事件数据，无数据则为nullptr
- **size**：事件数据大小，无数据则为0

返回值：无；

onError方法（暂未使用）

功能：错误通知接口

函数原形：

```
void onError(int errorCode, void *data, size_t size);
```

参数：

- **data**：附带的事件数据，无数据则为nullptr
- **size**：事件数据大小，无数据则为0

返回值：无；

API类管理接口

会话管理接口

start方法

功能：启动API。

调用该接口，API会自动向交易柜台发起连接

交易柜台的配置信息，默认从创建API对象的配置文件中读取。也可以通过 setConfig() 接口配置。

如果接口调用成功，将回调 XTFListener::onStart() 接口通知启动的结果。

如果API停止后再重新启动，可以传入新的对象，以处理新的业务逻辑。

例如：

```
XTFSpiA *a = new XTFSpiA()
api->start(a);
... // do something.
api->stop();

XTFSpiB *b = new XTFSpiB()
api->start(b); // ok.
... // do something.
api->stop();
```

函数原形：

```
int start(XTFSpi *spi);
```

参数：

- **spi**：回调事件处理对象

返回值：接口调用成功返回0，否则返回错误码

说明：不建议传入不同的XTFSpi对象，从逻辑处理上存在一定风险

stop方法

功能：停止API接口

调用该接口，API会断开与交易柜台的连接。

如果接口调用成功，将回调 XTFSpi::onStop() 接口通知停止的结果。

停止后的API接口，可以重新启动。此时，可以传入新的Listener对象，处理不同的逻辑

函数原形：

```
int stop()
```

参数：无

返回值：接口调用成功返回0，否则返回错误码

login方法

功能：登录交易柜台

接口调用成功后，将回调 onLogin() 接口通知登录结果。

登录接口有三种不同形式的重载：

- 1.不带任何参数的接口，默认使用配置文件中的设置，或者通过 setConfig() 接口的信息；
- 2.仅带有账户和密码的接口，AppID和AuthCode默认使用配置文件的设置。或者通过setConfig() 接口设置的信息；
- 3.带有全部参数的接口，将会自动覆盖配置文件或 setConfig() 接口设置的信息；

函数原形：

```
int login()  
int login(const char *accountID, const char *password)  
int login(const char *accountID, const char *password, const char *appID, const char *authCode)
```

参数：

- accountID: 资金账户编码
- password: 资金账户密码
- appID: 应用程序ID
- authCode: 认证授权码

返回值：接口调用成功返回0，否则返回错误码

logout方法

功能：登出交易柜台

该接口调用成功后，将回调 onLogout() 接口通知登录结果。

函数原形：

```
int logout()
```

参数：无

返回值：接口调用成功返回0，否则返回错误码

changePassword方法

功能：修改账户密码

接口调用成功后，将回调 onChangePassword() 接口通知修改密码结果。

函数原形：

```
int changePassword(const char *oldPassword, const char *newPassword)
```

参数：

- oldPassword：旧密码
- newPassword：新密码

返回值：接口调用成功返回0，否则返回错误码，调用成功并不表示密码修改成功

交易接口

insertOrder方法

功能：发送报单

在 XTFSpi::onOrder() 接口中通知用户插入报单的结果和报单状态。

函数原形：

int insertOrder(const XTFinputOrder &inputOrder);

int insertOrders(const XTFinputOrder *inputOrder,size_t orderCount);

参数：

- inputOrder：报单参数

XTFinputOrder结构体：报单请求对象类

```
class XTFinputOrder {
public:
    XTFLocalOrderID        localOrderID;           ///< 本地报单编号（用户）
                                                    ///< 本地报单编号需要由用户保证唯一性，用于本地存储索引。
                                                    ///< 为保证报单性能，API不做本地报单编号重复的校验。
                                                    ///< 如果API发生了断线重连，在历史流水追平之后，请继续保持后续本地报单编号与历史报单编号的唯一性。
                                                    ///< 如果不能保证本地报单编号的唯一性，请不要使用API的订单管理功能。
                                                    ///< API允许客户端使用同一用户名/口令多次登录，但客户端需要使用某种机制确保本地报单编号不发生重复。
                                                    ///< 比如：（奇偶交替、分割号段）+单向递增。

    XTFDirection            direction;              ///< 买卖方向
    XTFOffsetFlag           offsetFlag;             ///< 开平仓标志
    XTFOrderType            orderType;              ///< 报单类型：限价(GFD)/市价/FAK/FOK
    double                  price;                  ///< 报单价格
    uint32_t                volume;                 ///< 报单数量
    uint32_t                minVolume;              ///< 最小成交数量
    XTFFChannelSelectionType channelSelectionType;  ///< 席位编号选择类型
    uint8_t                 channelID;               ///< 席位编号
    XTFOrderFlag            orderFlag;              ///< 报单标志（不使用，默认都是普通报单）

    XTF_CONST XTFinstrument *instrument;           ///< 报单合约对象
};
```

返回值：接口调用成功返回0，否则返回错误码，调用成功并不表示报单操作的结果。

重要说明：API允许客户端使用同一用户名/口令多次登录，但客户端需要使用某种机制确保本地报单编号不发生重复。比如：（奇偶交替、分割号段）+单向递增。

cancelOrder方法

功能：取消报单

在 XTFSpi::onOrder() 和XTFSpi::onCancelOrder()接口中通知取消报单的结果和报单状态。

函数原形：

```
int cancelOrder(const XTFOrder *order);
int cancelOrders(const XTFOrder *orders, size_t orderCount);
int cancelOrder(XTFOrderIDType orderIDType, long orderID);
int cancelOrders(XTFOrderIDType orderIDType, long orderIDs[], size_t orderCount);
```

参数：

- order：撤单请求对象指针
- orderIDType：报单编号类型，目前支持柜台流水号和本地报单编号撤单；
- orderID：柜台流水号或本地报单编号，如果是本地报单编号，要求本地报单编号具备唯一性；

返回值：接口调用成功返回0，否则返回错误码，调用成功并不表示报单操作的结果。

外接行情接口

subscribe方法

功能：订阅行情

行情订阅成功后，当收到对应合约的行情数据后，会回调 onBookUpdate() 接口。

函数原形：

```
int subscribe(const XTFInstrument *instrument)
```

参数：

- instrument：合约对象

XTFInstrument结构体：合约对象类

```
class XTFInstrument {
public:
    XTFInstrumentID instrumentID;    ///< 合约代码
    uint32_t instrumentIndex;    ///< 合约序号
    int deliveryYear;    ///< 交割年份
    int deliveryMonth;    ///< 交割月份
    int maxMarketOrderVolume;    ///< 市价报单的最大报单量
    int minMarketOrderVolume;    ///< 市价报单的最小报单量
    int maxLimitOrderVolume;    ///< 限价报单的最大报单量
    int minLimitOrderVolume;    ///< 限价报单的最小报单量
    double priceTick;    ///< 最小变动价位
    int multiple;    ///< 合约单位
    XTFDate expireDate;    ///< 合约到期日，字符串格式：20220915
    bool singleSideMargin;    ///< 是否单边计算保证金
    XTFInstrumentStatus status;    ///< 当前状态
};
```

返回值：接口调用成功或失败，调用成功表示行情订阅成功

unsubscribe方法

功能：取消订阅行情

函数原形：

```
int subscribe(const XTFInstrument *instrument)
```

参数：

- instrument：合约对象，nullptr表示取消当前所有已订阅的行情。

XTFInstrument结构体：合约对象类

```
class XTFInstrument {
public:
    XTFInstrumentID instrumentID;    ///< 合约代码
    uint32_t instrumentIndex;    ///< 合约序号
    int deliveryYear;    ///< 交割年份
    int deliveryMonth;    ///< 交割月份
    int maxMarketOrderVolume;    ///< 市价报单的最大报单量
    int minMarketOrderVolume;    ///< 市价报单的最小报单量
    int maxLimitOrderVolume;    ///< 限价报单的最大报单量
    int minLimitOrderVolume;    ///< 限价报单的最小报单量
    double priceTick;    ///< 最小变动价位
    int multiple;    ///< 合约单位
    XTFDate expireDate;    ///< 合约到期日，字符串格式：20220915
    bool singleSideMargin;    ///< 是否单边计算保证金
    XTFInstrumentStatus status;    ///< 当前状态
};
```

返回值：接口调用成功或失败，调用成功表示行情订阅取消成功。

updateBook方法

功能：更新合约行情

暂时仅支持一档行情数据。

行情发生变化后，将回调 XTFSpi::onMarketData() 通知接口。

函数原形：

```
int updateBook(const XTFInstrument *instrument,    ///< 合约对象
    double lastPrice,    ///< 最新价
    double bidPrice,    ///< 买入价。为零代表无买入价。
    int bidVolume,    ///< 买入量。为零代表无买入价。
    double askPrice,    ///< 卖出价。为零代表无卖出价。
    int askVolume    ///< 卖出量。为零代表无卖出价。
);
```

参数：

- instrument：合约对象

XTFInstrument结构体：合约对象类

```
class XTFInstrument {
public:
    XTFInstrumentID instrumentID;    ///< 合约代码
    uint32_t instrumentIndex;    ///< 合约序号
    int deliveryYear;    ///< 交割年份
    int deliveryMonth;    ///< 交割月份
    int maxMarketOrderVolume;    ///< 市价报单的最大报单量
    int minMarketOrderVolume;    ///< 市价报单的最小报单量
    int maxLimitOrderVolume;    ///< 限价报单的最大报单量
    int minLimitOrderVolume;    ///< 限价报单的最小报单量
    double priceTick;    ///< 最小变动价位
    int multiple;    ///< 合约单位
    XTFDate expireDate;    ///< 合约到期日，字符串格式：20220915
```



```
bool        singleSideMargin;    ///< 是否单边计算保证金
XTFInstrumentStatus status;      ///< 当前状态
};
```

- lastPrice : 最新价
- bidPrice : 买入价。为零代表无买入价
- bidVolume : 买入量。为零代表无买入价
- askPrice : 卖出价。为零代表无卖出价
- askVolume : 卖出量。为零代表无卖出价

返回值：接口调用成功或失败，调用成功表示行情更新成功。

查询管理接口

getAccount方法

功能：获取当前资金账户信息

函数原形：

```
const XTFAccount*   getAccount()
```

参数：无

返回值：返回资金账户对象指针

getExchangeCount方法

功能：获取交易所信息

函数原形：

```
int        getExchangeCount();
const XTFExchange* getExchange(int pos);
```

参数：

- pos : 交易所的相对位置

返回值：无参数-第几个交易所，有参数-交易所信息

getInstrumentCount方法

功能：获取合约信息

函数原形：

```
int        getInstrumentCount();
const XTFInstrument* getInstrument(int pos);
const XTFInstrument* getInstrumentByID(const char *instrumentID);
```

参数：

- pos : 交易所的相对位置

返回值：无参数-第几个交易所，有参数-合约信息

findOrders方法

功能：查找报单信息

函数原形：

```
int findOrders(const XTOrderFilter &filter,
               unsigned int count,
               const XTOrder *orders[]);
```

参数：

- filter：报单查找条件；
- count：外部传入的数组长度参数，最大查询该数量的报单；
- orders：外部传入的数组，用于存储查询结果；

返回值：实际查询到的报单数量，如果大于count，则表示取前面的count个报单；如果小于count，则表示查到的数量少于传入的数量。

findTrades方法

功能：查找成交信息

函数原形：

```
int findTrades(const XTTradeFilter &filter
               unsigned int count,
               const XTTrade *trades[]);
```

参数：

- filter：成交明细查找条件；
- count：外部传入的数组长度参数，最大查询该数量的成交明细；
- orders：外部传入的数组，用于存储查询结果；

返回值：实际查询到的成交数量，如果大于count，则表示取前面的count个成交；如果小于count，则表示查到的数量少于传入的数量。

参数配置管理接口

setConfig方法

功能：设置会话参数，应在启动会话之前设置参数，启动之后设置的参数不会生效。

函数原形：

```
int setConfig(const char *name, const char *value)
```

参数：

- Name：参数名称
- Value：参数值

返回值：参数配置成功或失败

说明：会话参数名称列表：

- "ACCOUNT_ID"：资金账户ID
- "ACCOUNT_PWD"：资金账户密码
- "APP_ID"：应用程序ID
- "AUTH_CODE"：认证授权码
- "QUERY_SERVER_IP"：查询服务地址
- "QUERY_SERVER_PORT"：查询服务端口

- "TRADE_SERVER_IP"：交易服务地址
 - "TRADE_SERVER_PORT"：交易服务端口
 - "TCP_WORKER_CORE_ID"：数据收发线程绑核
 - "TCP_WORKER_BUSY_LOOP_ENABLED"：数据收发线程是否启用BusyLoop模式运行，默认为"true"，表示BusyLoop模式
 - "TRADE_WORKER_CORE_ID"：报单处理线程绑核
 - "TASK_WORKER_CORE_ID"：通用任务线程绑核
 - "WARM_INTERVAL"：预热时间间隔，取值范围：[10,50]，单位：毫秒
- 用户也可以设置自己的参数，以便在需要的地方使用 getConfig() 接口进行获取。会话内部不使用这部分用户参数，仅对其临时存储。

getConfig方法

功能：查询会话参数

函数原形：

```
string getConfig(const char *name);
```

参数：

- Name：参数名称

返回值：参数值

getVersion方法

功能：获取API版本

函数原形：

```
const char *getVersion();
```

参数：无

返回值：版本字符串

结构体说明

XTFExchange结构体

功能：交易所信息对象类

```
class XTFExchange {
public:
    XTFExchangeID      exchangeID;          ///< 交易所编码，字符串。比如：CFFEX, SHFE, DCE, CZCE等
    uint8_t            clientIndex;          ///< 裸协议报单需要使用该字段，每个交易所对应的值不同。
    uint32_t            clientToken;          ///< 裸协议报单需要使用该字段，每个交易所对应的值不同。
    XTFDate              tradingDay;          ///< 交易日，整数字符串。比如：20220915
    bool                hasChannelIP;          ///< 是否支持席位编号IP地址查询，true表示支持IP地址查询
    int                 getChannelCount() const;  ///< 查询所有席位编号数量
```

```

uint8_t          getChannel(int pos) const;    ///< 按位置索引查询席位编号。

                                                    ///< 位置索引pos参数取值范围：[0, getChannelCount() - 1]

                                                    ///< 如果使用API报单直接使用查询的席位编号值即可；

                                                    ///< 如果是裸协议报单，则需要在查询结果的基础上+10作为席位编号。

const char*      getChannelIP(int pos) const;  ///< 按位置索引查询席位编号IP地址。

                                                    ///< 如果hasChannelIP为false，返回nullptr；

                                                    ///< 否则返回对应的IP地址字符串指针

const char*      getChannelIPByID(uint8_t id) const; ///< 按席位编号查询IP地址，返回值同getChannelIP()

int              getProductGroupCount() const;  ///< 查询品种组数量

const XTFProductGroup* getProductGroup(int pos) const; ///< 按位置索引查找品种组对象
};

```

XTFAccount结构体

功能：客户/账户资金信息对象类

```

class XTFAccount {
public:
    XTFAccountID    accountID;                ///< 账户编码
    double          preBalance;                ///< 昨日余额
    double          deposit;                  ///< 今日入金
    double          withdraw;                 ///< 今日出金
    double          commission;               ///< 手续费
    double          frozenCommission;          ///< 冻结的手续费
    double          margin;                   ///< 保证金
    double          frozenMargin;              ///< 冻结的保证金
    double          premium;                  ///< 权利金
    double          frozenPremium;             ///< 冻结的权利金
    double          balance;                  ///< 资金余额
    double          available;                ///< 可用资金
    double          availableRatio;            ///< 资金可用限度
    double          positionProfit;            ///< 持仓盈亏，所有合约的持仓盈亏之和

    double          closeProfit;               ///< 平仓盈亏，所有合约的平仓盈亏之和

    XTFLocalOrderID lastLocalOrderID;          ///< 用户最后一次报单的本地编号
    XTFLocalActionID lastLocalActionID;         ///< 用户最后一次撤单的本地编号
                                                    ///< 如果使用API接口进行撤单，API内部会从(0xF0000001)开始逐步自增生成本地撤单编号。

                                                    ///< 用户可以通过判断最高位来确定是否为自定义的本地撤单编号。

    int             getOrderCount() const;      ///< 查询所有报单数量，用于遍历查询所有报单列表

    const XTFOrder* getOrder(int pos) const;    ///< 按位置索引查询报单对象，从0开始计算

    int             getPrePositionCount() const; ///< 查询所有昨持仓数量，用于遍历查询所有昨持仓列表
};

```

```

    const XTFPrePosition* getPrePosition(int pos) const; ///< 按位置索引查询昨持仓对象，从0开始计算
    int getPositionCount() const; ///< 查询所有仓位数量，用于遍历查询所有持仓数据，同一合约的多头和空头是两个不同持仓对象
    const XTFPosition* getPosition(int pos) const; ///< 按位置索引查询持仓对象，从0开始计算。已平仓合约无法通过此接口查询
    int getTradeCount() const; ///< 查询所有成交数量，用于遍历查询所有成交列表
    const XTFTrade* getTrade(int pos) const; ///< 按位置索引查询成交对象，从0开始计算
    int getCashInOutCount() const; ///< 查询所有出入金记录数量，用于遍历查询所有出入金列表
    const XTCashInOut* getCashInOut(int pos) const; ///< 按位置索引查询出入金对象，从0开始计算
};

```

XTFPrePosition结构体

功能：客户昨持仓对象类

```

class XTFPrePosition {
public:
    int preLongPosition; ///< 昨多头持仓量
    int preShortPosition; ///< 昨空头持仓量
    double preSettlementPrice; ///< 昨结算价
    const XTFInstrument* getInstrument() const; ///< 查询昨持仓所属合约
    XTFInstrument的指针
};

```

XTFPosition结构体

功能：客户持仓对象类

```

class XTFPosition {
public:
    XTFPositionDirection direction; ///< 持仓方向：多头、空头
    int position; ///< 持仓总量：历史持仓 - 已平仓
    史持仓 + 今仓 - 已平今仓，包含了平仓冻结量
    int todayPosition; ///< 今持仓量：今仓 - 已平今仓，
    包含了平仓冻结量（今仓部分）
    int combinedPosition; ///< 已组合的持仓数量
    int openFrozen; ///< 开仓冻结量
    int closeFrozen; ///< 平仓冻结量
    double margin; ///< 占用保证金
    double paidMargin; ///< 实付保证金
    double frozenMargin; ///< 冻结保证金
    double frozenCommission; ///< 冻结手续费
    double totalOpenPrice; ///< 总开仓金额
    double positionProfit; ///< 持仓盈亏 本合约剩余仓位的持仓
    盈亏（通过现价计算的动态值）
    double closeProfit; ///< 平仓盈亏 本合约所有今日已平
    仓位计算的总盈亏（静态值）

    double getOpenPrice() const; ///< 开仓均价
    int getAvailablePosition() const; ///< 获取可用仓位（持仓总量 - 平
    仓冻结量）

```

```

int            getYesterdayPosition() const;    ///< 获取剩余的历史持仓
int            getPositionDetailCount() const;  ///< 查询仓位明细数量
const XTFPositionDetail& getPositionDetail(int pos) const; ///< 查询仓位明细，
包括开仓成交和平仓成交明细
const XTFInstrument* getInstrument() const;    ///< 所属XTFInstrument的指针
};

```

XTFProduct结构体

功能：产品信息对象类

```

class XTFProduct {
public:
    XTFProductID        productID;                ///< 品种代码
    XTFProductClass      productClass;             ///< 品种类型
    int                  multiple;                 ///< 合约数量乘数
    double               priceTick;                ///< 最小变动价位
    int                  maxMarketOrderVolume;     ///< 市价报单的最大报单量
    int                  minMarketOrderVolume;     ///< 市价报单的最小报单量
    int                  maxLimitOrderVolume;      ///< 限价报单的最大报单量
    int                  minLimitOrderVolume;      ///< 限价报单的最小报单量
    int                  getInstrumentCount() const; ///< 查询品种包含的合约数量
    const XTFInstrument* getInstrument(int pos) const; ///< 按位置索引查询合约对象
    const XTFProductGroup* getProductGroup() const; ///< 查询品种所属的品种组对象
};

```

XTFInstrument结构体

功能：合约对象类

```

class XTFInstrument {
public:
    XTFInstrumentID      instrumentID;              ///< 合约代码
    uint32_t             instrumentIndex;           ///< 合约序号
    int                  deliveryYear;              ///< 交割年份
    int                  deliveryMonth;             ///< 交割月份
    int                  maxMarketOrderVolume;     ///< 市价报单的最大报单量
    int                  minMarketOrderVolume;     ///< 市价报单的最小报单量
    int                  maxLimitOrderVolume;      ///< 限价报单的最大报单量
    int                  minLimitOrderVolume;      ///< 限价报单的最小报单量
    double               priceTick;                ///< 最小变动价位
    int                  multiple;                 ///< 合约数量乘数
    XTFOptionsType       optionsType;              ///< 期权类型
    XTFDate              expireDate;               ///< 合约到期日，字符串
    格式: 20220915
    bool                singleSideMargin;          ///< 是否单边计算保证金
    XTFInstrumentStatus  status;                   ///< 当前状态

    const XTFExchange*   getExchange() const;      ///< 查询合约所属的
    XTFExchange指针

```

```

    const XTFProduct*      getProduct() const;          ///< 查询合约所属的
XTFProduct指针
    const XTFMarginRatio*  getMarginRatio(XTFHedgeFlag hedgeFlag =
XTF_HF_Speculation) const; ///< 保证金率
    const XTFCommissionRatio* getCommissionRatio(XTFHedgeFlag hedgeFlag =
XTF_HF_Speculation) const; ///< 手续费率
    const XTFPrePosition*  getPrePosition(XTFHedgeFlag hedgeFlag =
XTF_HF_Speculation) const; ///< 历史持仓，来自日初数据，交易日内不会变化
    const XTFPosition*     getLongPosition() const;      ///< 查询合约当前多头持
仓，包含历史持仓数据
    const XTFPosition*     getShortPosition() const;     ///< 查询合约当前空头持
仓，包含历史持仓数据
    const XTFMarketData*   getMarketData() const;       ///< 查询合约的行情
XTFMarketData指针
    const XTFInstrument*   getUnderlyingInstrument() const; ///< 如果是期权合约，表
示对应的基础合约指针
};

```

XTFMarginRatio结构体

功能：保证金率对象类

```

class XTFMarginRatio {
public:
    double          longMarginRatioByMoney;    ///< 按照金额计算的多头保证
金率。
    double          longMarginRatioByVolume;   ///< 按照数量计算的多头保证
金率。
    double          shortMarginRatioByMoney;   ///< 按照金额计算的空头保证
金率。
    double          shortMarginRatioByVolume;  ///< 按照数量计算的空头保证
金率。
    const XTFInstrument* getInstrument() const;  ///< 所属XTFInstrument的指
针
};

```

XTFCommissionRatio结构体

功能：手续费率对象类

```

class XTFCommissionRatio {
public:
    double          openRatioByMoney;          ///< 按金额计算的开仓手续费
率
    double          openRatioByVolume;         ///< 按数量计算的开仓手续费
率
    double          closeRatioByMoney;         ///< 按金额计算的平昨手续费
率
    double          closeRatioByVolume;        ///< 按数量计算的平昨手续费
率
    double          closeTodayRatioByMoney;    ///< 按金额计算的平今手续费
率
    double          closeTodayRatioByVolume;   ///< 按数量计算的平今手续费
率
    const XTFInstrument* getInstrument() const;  ///< 所属XTFInstrument的指
针
};

```

XTFInputOrder结构体

功能：报单请求对象类

```

class XTFInputOrder {
public:
    XTFLocalOrderID    localOrderID;          ///< 本地报单编号（用户）
                                                    ///< 本地报单编号需要由用户保证唯
一性，用于本地存储索引。
                                                    ///< 为保证报单性能，API不做本地
报单编号重复的校验。
                                                    ///< 如果API发生了断线重连，在历
史流水追平之后，请继续保持后续本地报单编号与历史报单编号的唯一性。
                                                    ///< 如果不能保证本地报单编号的唯
一性，请不要使用API的订单管理功能。
                                                    ///< API允许客户端使用同一用户
名/口令多次登录，但客户端需要使用某种机制确保本地报单编号不发生重复。
                                                    ///< 比如：（奇偶交替、分割号段）
+单向递增。

    XTFDirection       direction;              ///< 买卖方向
    XTFOffsetFlag       offsetFlag;            ///< 开平仓标志
    XTFOrderType        orderType;             ///< 报单类型：限价(GFD)/市
价/FAK/FOK
    double              price;                 ///< 报单价格
    uint32_t            volume;                ///< 报单数量
    uint32_t            minVolume;             ///< 最小成交数量
    XTFChannelSelectionType channelSelectionType; ///< 席位编号选择类型
    uint8_t             channelID;             ///< 席位编号
    XTFOrderFlag        orderFlag;            ///< 报单标志（不使用，默认都是普
通报单）

    XTF_CONST XTFInstrument *instrument;        ///< 报单合约对象
};

```

XTFOrder结构体

功能：报单回报对象类


```

class XTForder {
public:
    XTFSysOrderID        sysOrderID;          ///< 柜台流水号
    XTFLocalOrderID      localOrderID;        ///< 用户填写的本地报单号，必须保
证唯一性，否则会产生回报错误

    ///< 柜台清流启动后的历史报单固定
    为0xd8888888，非本柜台报单固定为0x88888888，

    ///< 错误为1172“请求中的报单编号
    不存在”时，返回0，其他情况返回订单真实编号

    XTFExchangeOrderID   exchangeOrderID;    ///< 交易所报单编号
    XTFDirection         direction;          ///< 买卖方向
    XTFOffsetFlag        offsetFlag;         ///< 开平仓标志
    double               orderPrice;         ///< 报单价格
    uint32_t             orderVolume;        ///< 报单数量
    uint32_t             orderMinVolume;     ///< 最小成交数量
    uint32_t             totalTradedVolume;  ///< 报单累计已成交数量
    XTFOrderType         orderType;          ///< 限价/FAK/市价类型
    XTFOrderFlag         orderFlag;          ///< 报单标志
    XTFChannelSelectionType channelSelectionType; ///< 席位连接选择
    uint8_t              channelId;          ///< 席位连接编号，0xFF表示无效
值
    uint8_t              realChannelID;      ///< 实际席位连接编号，由柜台返
回，0xFF表示无效值
    XTFOrderStatus       orderStatus;        ///< 报单状态
    XTFDate              insertDate;         ///< 报单插入日期，字符串格式：
20220915
    XTFTime              insertTime;         ///< 报单插入时间，字符串格式：
10:20:30
    XTFTime              updateTime;         ///< 报单更新时间，字符串格式：
10:20:30
    XTFTime              cancelTime;         ///< 报单撤单时间，字符串格式：
10:20:30
    bool                isHistory;           ///< 回报链路断开重连后或者程序重
启后，客户端API会自动进行流水重构，

    ///< 在追平服务器流水之前收到的报
    单回报，该字段为true。追平流水之后，该字段为false。

    ///< 如对流水重构的回报不需要特殊
    处理，可不用处理该字段。
    bool                isSuspended;        ///< 报单是否挂起（暂未使用）
    XTF_CONST XTFInstrument *instrument;     ///< 所属XTFInstrument的指针。
    如果报单传入的合约不存在，那么合约对象指针可能为空。
    XTFOrderActionType   actionType;         ///< 报单对象对应的操作类型（比
如：报单、撤单、挂起、激活等），只读字段，外部调用不需要设置该字段。

    int                 getTradeCount() const; ///< 查询报单对应的成交数量
    const XTFTrade*      getTrade(int pos) const; ///< 按位置索引查询报单对应的成交
明细
};

```

XTFTrade结构体

功能：成交回报对象类

```

class XTFTrade {
public:
    XTFTradeID          tradeID;             ///< 交易所成交编码

```

```

double          tradePrice;          ///< 成交价格
uint32_t        tradeVolume;         ///< 本次回报已成交数量
double          margin;              ///< 保证金
double          commission;          ///< 手续费
XTFTime         tradeTime;           ///< 报单成交时间，字符串格式：
10:20:30
XTFDirection    direction;           ///< 买卖方向,详情参考
XTFDataType.h   XTFOffsetFlag        offsetFlag;           ///< 开平仓标志,详情参考
XTFDataType.h   bool                isHistory;           ///< 回报链路断开重连后或者程序重
启后，客户端API会自动进行流水重构，
                                                    ///< 在追平服务器流水之前收到的成
交回报，该字段为true。追平流水之后，该字段为false。
                                                    ///< 如对流水重构的回报不需要特殊
处理，可不用处理该字段。

XTF_CONST XTFOrder *order;           ///< 所属XTFOrder的指针
XTF_CONST XTFTrade *matchTrade;      ///< 对手成交回报，如果不为空则表
明自成交

bool            isSelfTraded() const { return matchTrade != nullptr;
} ///< 判断是否为自成交
};

```

XTFMarketData结构体

功能：行情信息对象类

```

class XTFMarketData {
public:
    int          tradingDay;           ///< 交易日
    double        preSettlementPrice;  ///< 前结算价
    double        preClosePrice;       ///< 前收盘价
    double        preOpenInterest;     ///< 前持仓量（暂未使用）
    double        upperLimitPrice;     ///< 涨停价
    double        lowerLimitPrice;     ///< 跌停价
    double        lastPrice;           ///< 最新价（接入外部行情后有效）
    double        bidPrice;            ///< 买入价。为零代表无买入价（接
入外部行情后有效）
    int          bidVolume;            ///< 买入量。为零代表无买入价（接
入外部行情后有效）
    double        askPrice;            ///< 卖出价。为零代表无卖出价（接
入外部行情后有效）
    int          askVolume;            ///< 卖出量。为零代表无卖出价（接
入外部行情后有效）
    int          volume;               ///< 成交量（暂未使用）
    int          snapTime;             ///< 时间戳（暂未使用）
    double        turnover;            ///< 成交金额（暂未使用）
    double        openInterest;        ///< 持仓量（暂未使用）
    double        averagePrice;        ///< 行情均价（暂未使用）

    const XTFInstrument* getInstrument() const; ///< 所属XTFInstrument的指针
};

```

XTFCashInOut结构体

功能：出入金对象类

```
class XTFCashInOut {
public:
    XTFCashDirection    direction;          ///< 出入金方向
    double               amount;            ///< 出入金金额
    XTFTIME              time;              ///< 出入金时间
};
```

XTFTradeDetail结构体

功能：交易明细对象类

```
class XTFTradeDetail {
public:
    XTF_CONST XTFTrade    *trade;          ///< 成交对象
    int                 volume;            ///< 成交量
};
```

XTFPositionDetail结构体

功能：定义持仓明细结构体

```
class XTFPositionDetail {
public:
    XTF_CONST XTFTrade    *openTrade;      ///< 开仓成交回报，当等于
    nullptr时，代表昨仓，昨仓只会出现在持仓明细链表的头部
    XTFTradeID            openTradeID;      ///< 开仓成交编码
    double                openPrice;        ///< 开仓成交价格，与成交回报中的
    价格相同。如果是历史持仓则表示昨结算价。
    int                   openVolume;       ///< 持仓明细开仓数量
    int                   remainingVolume;   ///< 持仓明细中剩余仓位总数量
    int                   getCloseTradeCount() const;    ///< 平仓成交回报数
    量
    const XTFTradeDetail& getCloseTradeDetail(int pos) const; ///< 平仓成交回报明
    细
    bool                  isHistory() const { return openTrade == nullptr; }
    ///< 判断是否为历史持仓
    int                   getCombPositionDetailCount() const;    ///< 持仓
    明细关联的组合仓位明细列表
    const XTFCombPositionDetail& getCombPositionDetail(int pos) const; ///< 持仓
    明细
    int                   getCombinedVolume() const;    ///< 持仓
    明细中被组合占用的数量
};
```

XTFPositionFilter结构体（暂未使用）

功能：查询持仓信息的过滤器对象类（暂未使用）

```

class XTFPositionFilter {
public:
    int                positionDate;           ///< 持仓日期
    XTFPositionDirection positionDirection;    ///< 持仓方向
    XTFHedgeFlag        hedgeFlag;            ///< 套保标志，-1表示所有
    XTF_CONST XTFExchange *exchange;          ///< 指向交易所结构的指针，空指针表示所有
    XTF_CONST XTFProduct *product;            ///< 指向品种结构的指针，空指针表示所有
};

```

XTFOrderFilter结构体

功能：查询报单信息的过滤器对象类

```

class XTFOrderFilter {
public:
    XTFTime            startTime;              ///< 开始时间，字符串格式：10:20:30，空字符串表示所有
    XTFTime            endTime;                ///< 结束时间，字符串格式：10:20:30，空字符串表示所有
    XTFDirection       direction;              ///< 指定报单买卖方向，无效值表示所有
    XTFOffsetFlag      offsetFlag;             ///< 指定报单开平标志，无效值表示所有
    XTFOrderFlag       orderFlag;              ///< 指定报单标志，无效值表示所有
    XTFOrderType       orderType;              ///< 指定报单指令类型，无效值表示所有
    XTFOrderStatus     orderStatus[4];         ///< 指定报单状态，全部为无效值时表示所有
    XTF_CONST XTFInstrument *instrument;        ///< 指向合约结构的指针，空指针表示所有
    XTF_CONST XTFProduct *product;             ///< 指向品种结构的指针，空指针表示所有
    XTF_CONST XTFExchange *exchange;           ///< 指向交易所结构的指针，空指针表示所有
};

```

XTFTradeFilter结构体

功能：查询报单信息的过滤器对象类

```

class XTFTradeFilter {
public:
    XTFTime            startTime;              ///< 开始时间，字符串格式：10:20:30，空字符串表示所有
    XTFTime            endTime;                ///< 结束时间，字符串格式：10:20:30，空字符串表示所有
    XTFDirection       direction;              ///< 指定报单买卖方向，无效值表示所有
    XTFOffsetFlag      offsetFlag;             ///< 指定报单开平标志，无效值表示所有
    XTF_CONST XTFInstrument *instrument;        ///< 指向合约结构的指针，空指针表示所有
    XTF_CONST XTFProduct *product;             ///< 指向品种结构的指针，空指针表示所有
    XTF_CONST XTFExchange *exchange;           ///< 指向交易所结构的指针，空指针表示所有
};

```

TraderAPI开发示例

下列是我们打包发给您的Example02.cpp代码，供您参考使用，具体详见发布包的example目录。

```
/**
```

```

* @brief 一个简单的策略功能演示
*
* 说明：本演示代码需要接入外部行情，仅用于API功能演示用途，不能用于生产环境。
*
* 策略：根据行情数据选择买卖方向，每个方向的仓位最多持仓1手。
*
* 根据实时行情判断：
* 1. 假如卖出量小于买入量，或卖出量大于买入量不足10手：
*   1.1 如果持有空头仓位，那么以当前卖价平空头仓位；
*   1.2 如果没有多头仓位，那么以当前买价建多头仓位；
*
* 2. 假如卖出量大于买入量超过10手：
*   2.1 如果持有多头仓位，那么以当前买价平多头仓位；
*   2.2 如果没有空头仓位，那么以当前卖价建空头仓位；
*
* 演示功能：
* 1. 通过配置文件创建API实例；
* 2. 启动API，启动成功后，调用login接口登录柜台；
* 3. 等数据加载完毕后，实时接入外部行情；
* 4. 根据行情和交易策略，进行报单；
* 5. 经过1000次行情处理后，退出交易；
*/

#include <map>
#include "ExampleTrader.h"

class Example_02_Trader : public ExampleTrader {
public:
    Example_02_Trader() = default;

    ~Example_02_Trader() override {
        // release api.
        if (mApi) {
            mApi->stop();
            delete mApi;
            mApi = nullptr;
        }
    };

    void start() {
        if (mApi) {
            printf("error: trader has been started.\n");
            return;
        }

        mOrderLocalId = 0;
        mApi = makeXTFApi(mConfigPath.c_str());
        if (mApi == nullptr) {
            printf("error: create xtf api failed, please check config: %s.\n",
mConfigPath.c_str());
            exit(0);
        }

        printf("api version: %s.\n", getXTFVersion());
        int ret = mApi->start(this);
        if (ret != 0) {
            printf("start failed, error code: %d\n", ret);
            exit(0);
        }
    }
};

```

```

    }
}

void stop() {
    if (!mApi) {
        printf("error: trader is not started.\n");
        return;
    }

    int ret = mApi->stop();
    if (ret == 0) {
        delete mApi;
        mApi = nullptr;
    } else {
        printf("api stop failed, error code: %d\n", ret);
    }
}

void onStart(int errorCode, bool isFirstTime) override {
    ExampleTrader::onStart(errorCode, isFirstTime);

    if (errorCode == 0) {
        if (isFirstTime) {
            // TODO: init something if needed.
        }

        //mApi->login(mUsername.c_str(), mPassword.c_str(), mAppID.c_str(),
mAuthCode.c_str());
        int errorCode = mApi->login();
        if (errorCode != 0) {
            printf("api logging in failed, error code: %d\n", errorCode);
        }
    } else {
        printf("error: api start failed, error code: %d.\n", errorCode);
    }
}

void onStop(int errorCode) override {
    ExampleTrader::onStop(errorCode);

    if (errorCode == 0) {
        printf("api stop ok.\n");
    } else {
        printf("error: api stop failed, error code: %d.\n", errorCode);
    }
}

void onLogin(int errorCode, int exchangeCount) override {
    ExampleTrader::onLogin(errorCode, exchangeCount);

    if (errorCode != 0) {
        printf("error: login failed, error code: %d.\n", errorCode);
        return;
    }

    printf("login success.\n");
}

```

```

void onLogout(int errorCode) override {
    ExampleTrader::onLogout(errorCode);

    if (errorCode != 0) {
        printf("error: logout failed, error code: %d.\n", errorCode);
        return;
    }

    printf("logout success.\n");
}

void onChangePassword(int errorCode) override {
    if (errorCode != 0) {
        printf("error: change password failed, error code: %d.\n",
errorCode);
        return;
    }

    printf("change password success.\n");
}

void onReadyForTrading(const XTFAccount *account) override {
    ExampleTrader::onReadyForTrading(account);

    if (!mApi) return;
    mOrderLocalId = account->lastLocalOrderID;

    // 静态数据初始化完毕后回调，传递对象指针，是为了用户保存指针对象，方便使用。
    // 这里可以开始做交易。
    mInstrument = mApi->getInstrumentByID(mInstrumentID.c_str());
    if (!mInstrument) {
        printf("error: instrument not found: %s.\n", mInstrumentID.c_str());
        return;
    }

    int ret = mApi->subscribe(mInstrument);
    if (ret != 0) {
        printf("subscribe instrument %s failed, error code: %d\n",
mInstrumentID.c_str(), ret);
    }
}

void onLoadFinished(const XTFAccount *account) override {
    ExampleTrader::onLoadFinished(account);

    // 流水数据追平后回调
    printf("info: data load finished.\n");
}

void onOrder(int errorCode, const XTFOrder *order) override {
    // 报撤单失败。根据报单错误码判断是柜台拒单，还是交易所拒单。
    if (errorCode != 0) {
        switch (order->actionType) {
            case XTF_OA_Insert:
                printf("insert order failed, error: %d\n", errorCode);
                break;
            case XTF_OA_Return:
                printf("return order failed, error: %d\n", errorCode);

```

```

        break;
    default:
        printf("order action(%d) failed, error: %d.\n",
            order->actionType, errorCode);
        break;
    }
    return;
}

// 收到报单回报。根据报单状态判断是报单还是撤单。
switch (order->orderStatus) {
    case XTF_OS_Accepted:
        printf("order accepted.\n");
        mOrders[order->localOrderID] = order; // 保存报单对象
        break;
    case XTF_OS_AllTraded:
        printf("order all traded.\n");
        break;
    case XTF_OS_Queueing:
        printf("order queuing.\n");
        break;
    case XTF_OS_Rejected:
        printf("order rejected.\n");
        break;
    default:
        break;
}

}

void onCancelOrder(int errorCode, const XTFOrder *cancelOrder) override {
    // 报撤单失败。根据报单错误码判断是柜台拒单，还是交易所拒单。
    if (errorCode != 0) {
        printf("error: cancel order failed, sys-id: %d.\n", cancelOrder-
>sysOrderID);
        return;
    }
    printf("order canceled, sys-id: %d.\n", cancelOrder->sysOrderID);
}

void onTrade(const XTFTTrade *trade) override {
    // 收到交易回报
    printf("recv trade, sys-order-id: %d, trade-id: %ld.\n", trade->order-
>sysOrderID, trade->tradeID);
}

void onAccount(int event, int action, const XTFAccount *account) override {
    // 账户信息发生变化时回调该接口，如：出入金变化
    if (event == XTF_EVT_AccountCashInOut) {
        if (action == XTF_CASH_In) printf("cash in.\n");
        if (action == XTF_CASH_Out) printf("cash out.\n");
    }
}

void onExchange(int event, int channelID, const XTFExchange *exchange)
override {
    // 交易所信息发生变化时回调该接口，如：交易所前置变化
    printf("exchange is changed.\n");
}

```



```

void onInstrument(int event, const XTFInstrument *instrument) override {
    // 合约属性发生变化时回调该接口，如：状态变化
    if (event == XTF_EVT_InstrumentStatusChanged) {
        printf("instrument status changed: %s %d.\n",
            instrument->instrumentID, instrument->status);
    }
}

void onBookUpdate(const XTFMarketData *marketData) override {
    // 行情回调接口，根据行情触发交易策略
    doTrade(*marketData);
}

void onEvent(const XTFEvent &event) override {
    printf("recv event: %d.\n", event.eventID);
}

void onError(int errorCode, void *data, size_t size) override {
    printf("something is wrong, error code: %d.\n", errorCode);
}

void updateBook(const char *instrumentID,
                double lastPrice,
                double bidPrice,
                int bidVolume,
                double askPrice,
                int askVolume) {
    if (!mApi) {
        printf("error: api is not started.\n");
        return;
    }

    const XTFInstrument *instrument = mApi->getInstrumentByID(instrumentID);
    if (!instrument) {
        printf("error: instrument is not found: %s.\n", instrumentID);
        return;
    }

    int ret = mApi->updateBook(instrument, lastPrice, bidPrice, bidVolume,
askPrice, askVolume);
    if (ret != 0) {
        printf("error: update market data failed, error code: %d\n", ret);
    }
}

private:
void doTrade(const XTFMarketData &marketData) {
    if (marketData.getInstrument() != mInstrument) {
        return;
    }

    double askPrice = marketData.askPrice;
    int askVolume = marketData.askVolume;
    double bidPrice = marketData.bidPrice;
    int bidVolume = marketData.bidVolume;
    if (askVolume - bidVolume <= 10) { // buy
        if (mInstrument->getShortPosition()->position >= 1) {

```

```

        closeShort(askPrice, 1);
        printf("close short position: 1.\n");
        return;
    }

    if (mInstrument->getLongPosition()->position <= 0) {
        openLong(askPrice, 1);
        printf("open long position: 1.\n");
        return;
    }
}

if (askVolume - bidVolume >= 10) { // sell
    if (mInstrument->getLongPosition()->position >= 1) {
        closeLong(bidPrice, 1);
        printf("close long position: 1.\n");
        return;
    }

    if (mInstrument->getShortPosition()->position <= 0) {
        openShort(bidPrice, 1);
        printf("open short position: 1.\n");
        return;
    }
}

}

int openLong(double price, uint32_t volume) {
    if (!mApi) return -1;
    XTFInputOrder order{};
    memset(&order, 0, sizeof(order));
    order.instrument = mInstrument;
    order.direction = XTF_D_Buy;
    order.offsetFlag = XTF_OF_Open;
    order.orderType = XTF_ODT_FOK;
    order.price = price;
    order.volume = volume;
    order.channelSelectionType = XTF_CS_Auto;
    order.localOrderID = ++mOrderLocalId;
    return mApi->insertOrder(order);
}

int closeLong(double price, uint32_t volume) {
    if (!mApi) return -1;
    XTFInputOrder order{};
    memset(&order, 0, sizeof(order));
    order.instrument = mInstrument;
    order.direction = XTF_D_Buy;
    order.offsetFlag = XTF_OF_Close;
    order.orderType = XTF_ODT_FOK;
    order.price = price;
    order.volume = volume;
    order.channelSelectionType = XTF_CS_Auto;
    order.localOrderID = ++mOrderLocalId;
    return mApi->insertOrder(order);
}

int openShort(double price, uint32_t volume) {

```

```

        if (!mApi) return -1;
        XTFInputOrder order{};
        memset(&order, 0, sizeof(order));
        order.instrument = mInstrument;
        order.direction = XTF_D_Sell;
        order.offsetFlag = XTF_OF_Open;
        order.orderType = XTF_ODT_FOK;
        order.price = price;
        order.volume = volume;
        order.channelSelectionType = XTF_CS_Auto;
        order.localOrderID = ++mOrderLocalId;
        return mApi->insertOrder(order);
    }

    int closeShort(double price, uint32_t volume) {
        if (!mApi) return -1;
        XTFInputOrder order{};
        memset(&order, 0, sizeof(order));
        order.instrument = mInstrument;
        order.direction = XTF_D_Sell;
        order.offsetFlag = XTF_OF_Close;
        order.orderType = XTF_ODT_FOK;
        order.price = price;
        order.volume = volume;
        order.channelSelectionType = XTF_CS_Auto;
        order.localOrderID = ++mOrderLocalId;
        return mApi->insertOrder(order);
    }

    int cancelOrder(int orderLocalId) {
        auto iter = mOrders.find(orderLocalId);
        if (iter == mOrders.end()) {
            printf("order not found: %d\n", orderLocalId);
            return -1;
        }
        const XTForder *order = iter->second;
        return mApi->cancelOrder(order);
    }

private:
    const XTFInstrument *mInstrument;
    std::map<int, const XTForder *> mOrders;
    int mOrderLocalId;
};

void runExample(const std::string &configPath, const std::string &instrumentId)
{
    printf("start example 02.\n");

    Example_02_Trader trader;
    trader.setConfigPath(configPath);
    trader.setInstrumentID(instrumentId);

    trader.start();

    bool isStop = false;
    int tickCount = 0;
    while (!isStop) {

```

```

        if (trader.isLoadFinished()) {
            // TODO: 修改下面的代码，接入外部行情，通过外部行情，驱动策略进行报单。
            double bidPrice = 300.50;
            double askPrice = 301.50;
            double lasPrice = 301.00;
            int bidVolume = 24;
            int askVolume = 38;
            trader.updateBook(instrumentId.c_str(),
                              lasPrice,
                              bidPrice,
                              bidVolume,
                              askPrice,
                              askVolume);
        }

        printf("sleep 500ms.\n");
        usleep(500000); // sleep 500ms
        if (tickCount++ > 1000) {
            isStop = true;
        }
    }

    trader.stop();
}

int main(int argc, const char *argv[]) {
    // TODO: 解析传入参数，提取相关的配置
    std::string configPath = "../config/xtf_trader_api.config";
    std::string instrumentId = "au2212";
    runExample(configPath, instrumentId);
    return 0;
}

```

字段类型

字段类型总表

基础数据类型如下：

数据类型名	数据类型	数据类型说明
XTFExchangeID	char[12]	交易所ID数据类型
XTFProductID	char[32]	品种数据类型
XTFInstrumentID	char[32]	合约ID数据类型
XTFAccountID	char[16]	账号ID数据类型
XTFPassword	char[16]	账号密码数据类型
XTFTime	char[16]	时间数据类型
XTFString	char[32]	字符串数据类型
XTFExchangeOrderID	int	交易所报单编号数据类型
XTFSysOrderID	int	柜台唯一的报单编号数据类型
XTFLocalOrderID	int	本地委托号数据类型（用户定义）
XTFTradeID	long long	成交编号数据类型

枚举类型

报单编号类型

类型：int

合约状态类型	说明	枚举值(可显示字符)
XTF_OIDT_Local	用户维护的本地编号	0
XTF_OIDT_System	柜台维护的系统编号	1
XTF_OIDT_Exchange	市场唯一的交易所编号	2

产品类型

类型：int

枚举类型	说明	枚举值
XTF_PC_Futures	期货类型	1
XTF_PC_Options	期货期权类型	2
XTF_PC_Combination	组合类型	3
XTF_PC_SpotOptions	现货期权类型	6

期权类型

类型：int

枚举类型	说明	枚举值
XTF_OT_NotOption	非期权	0
XTF_OT_CallOption	看涨	1
XTF_OT_PutOption	看跌	2

买卖方向

类型：uint8_t

枚举类型	说明	枚举值
XTF_D_Buy	买	1
XTF_D_Sell	卖	2

持仓多空方向

类型：uint8_t

枚举类型	说明	枚举值
XTF_PD_Long	多头	1
XTF_PD_Short	空头	2

持仓日期

类型：uint8_t

枚举类型	说明	枚举值
XTF_PSD_Today	今仓	1
XTF_PSD_History	昨仓	2

投机套保标志，当前仅支持投机

类型：uint8_t

枚举类型	说明	枚举值
XTF_HF_Invalid	无效	0
XTF_HF_Speculation	投机	1
XTF_HF_Arbitrage	套利	2
XTF_HF_Hedge	保值	3
XTF_HF_MaxCount	保留内部使用	4

出入金方向

类型：uint8_t

枚举类型	说明	枚举值
XTF_CASH_IN	入金	1
XTF_CASH_OUT	出金	2

开平标志

类型：字符枚举

枚举类型	说明	枚举值
XTF_OF_Open	开仓	0
XTF_OF_Close	平仓	1
XTF_OF_ForceClose	强平	2
XTF_OF_CloseToday	平今	3
XTF_OF_CloseYesterday	平昨	4
XTF_OF_Invalid	无效	9

指令类型

类型：uint8_t

枚举类型	说明	枚举值
XTF_ODT_Limit	GFD报单（限价）	0
XTF_ODT_FAK	FAK报单（限价）	1
XTF_ODT_FOK	FOK报单（限价）	2
XTF_ODT_Market	市价报单（暂不支持）	3

报单状态

类型：uint8_t

枚举类型	说明	枚举值
XTF_OS_Created	报单已创建，为报单的初始状态。 报单发送成功后，API会立即创建报单对象并设置为初始状态，此时报单的柜台流水号、交易所编号等字段都是无效的。当收到柜台返回的应答和回报时，会切换为其它状态， 用户可以根据此状态查询哪些报单还没有收到应答和回报。	0
XTF_OS_Accepted	报单已通过柜台校验并送往交易所，且已收到交易所的报单录入结果（报单应答消息）。 此状态为临时状态，在报单应答和报单回报乱序的场景下，会跳过此状态，直接进入后续状态。 后续状态包括：XTF_OS_Queueing XTF_OS_Canceled XTF_OS_PartTraded XTF_OS_AllTraded XTF_OS_Rejected	1
XTF_OS_Queueing	报单未成交，已被交易所确认且有效	2
XTF_OS_Canceled	报单已被撤销，可能是客户主动撤销，也可能是FAK类或市价类报单被交易所系统自动撤销	3
XTF_OS_PartTraded	部分成交	4
XTF_OS_AllTraded	完全成交	5
XTF_OS_Rejected	报单被柜台或交易所拒绝，拒绝原因放在InputOrder的ErrorNo字段中	6
XTF_OS_Invalid	无效的值	9

报单标志

类型：uint8_t

枚举类型	说明	枚举值
XTF_ODF_Normal	普通报单	0

席位选择

类型: uint8_t

枚举类型	说明	枚举值
XTF_CS_Auto	不指定交易所席位链接	0
XTF_CS_Fixed	指定交易所席位链接	1

API事件

类型：uint8_t

枚举类型	说明	枚举值
XTF_AE_ServerRestarted	发现服务器重启，客户端退出前推送的事件	1

保证金计算价格类型

类型：int

枚举类型	说明	枚举值
XTF_CBT_PreSettlementPrice	昨结算价	0
XTF_CBT_OpenPrice	开仓价	1
XTF_CBT_LastPrice	最新价	2
XTF_CBT_MarketAveragePrice	市场均价	3
XTF_CBT_MaxLastPreSettlementPrice	昨结算和最新价的大值	4
XTF_CBT_OrderPrice	报单价格	5
XTF_CBT_None	不纳入计算	6

合约状态

类型：字符枚举

枚举类型	说明	枚举值
XTF_IS_BeforeTrading	开盘前	0
XTF_IS_NoTrading	非交易	1
XTF_IS_Continuous	连续交易	2
XTF_IS_AuctionOrdering	集合竞价报单	3
XTF_IS_AuctionBalance	集合竞价价格平衡	4
XTF_IS_AuctionMatch	集合竞价撮合	5
XTF_IS_Closed	收盘	6
XTF_IS_TransactionProcessing	交易业务处理	7

报单操作类型

类型：uint8_t

枚举类型	说明	枚举值
XTF_OA_Invalid	无效的报单操作	0
XTF_OA_Insert	插入报单	1
XTF_OA_Cancel	撤销报单	2
XTF_OA_Suspend	挂起报单（暂未使用）	3
XTF_OA_Resume	恢复报单（暂未使用）	4
XTF_OA_Update	更新报单（暂未使用）	5
XTF_OA_Return	报单回报	9

事件通知类型

类型：uint8_t

枚举类型	说明	枚举值
XTF_EVT_AccountCashInOut	账户出入金发生变化通知	1
XTF_EVT_ExchangeChannelConnected	交易所交易通道已连接通知	11
XTF_EVT_ExchangeChannelDisconnected	交易所交易通道已断开通知	12
XTF_EVT_InstrumentStatusChanged	合约状态发生变化通知	21

10 附录