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，GFEX）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， 例如：xele-futures-trader-api-linux-4.1.630-f0989ea.tar.gz。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
lib/libxele_futures_trader_api.so	Linux 64位版本的API动态库
config/xtf_trader_api.config	配置文件模板
example/ExampleXX.cpp	示例程序的源代码文件
example/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**出现异常崩溃，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-2022-09-11-16_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；
2022/09/29	V4.1.465	增加错误码查询接口，支持错误消息内容的中英文版本； makeXTFApi()接口允许配置文件路径为空，启动API之前，通过API的setConfig()接口来设置参数； 优化资金和仓位计算；
2022/10/13	V4.1.486	优化报单插入错误回报处理； 优化UDP预热报单； 优化部分结构体字段；
2022/11/22	V4.1.586	新增版本兼容性检测； 账户字段扩展为20字节； 增加交易流水的日志记录功能； 修复已知的冻结手续费计算错误问题；
2022/12/20	V4.1.629	提供资金同步接口； 新增创建预热报单接口； 中金期权功能优化； XTFAccount结构体增加交割保证金、昨日余额字段；
2023/01/18	V4.1.664	优化使用手册文档，规范错误码的定义； 发送TCP数据时增加多线程同步机制，以防数据交替发送造成异常； 修复部分资金计算与柜台不一致的问题； 报撤单接口增加多线程工作模式，在配置文件增加配置选项 `ORDER_MT_ENABLED=true` 可以启用多线程报单功能。默认为单线程报撤单，不启用多线程模式；
2023/03/22	V4.1.749	增加商品期权功能； 增加行权和对冲接口； 报单状态增加 `XTF_OS_Received` 状态表示通过柜台风控； 修复部分资金与柜台不一致的问题；
2023/06/01	V4.1.861	增加询价接口； 增加回报过滤接口； 增加申报费字段和接口； 增加本地单号撤单接口； 报撤单对象增加用户自定义字段userRef；

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( )

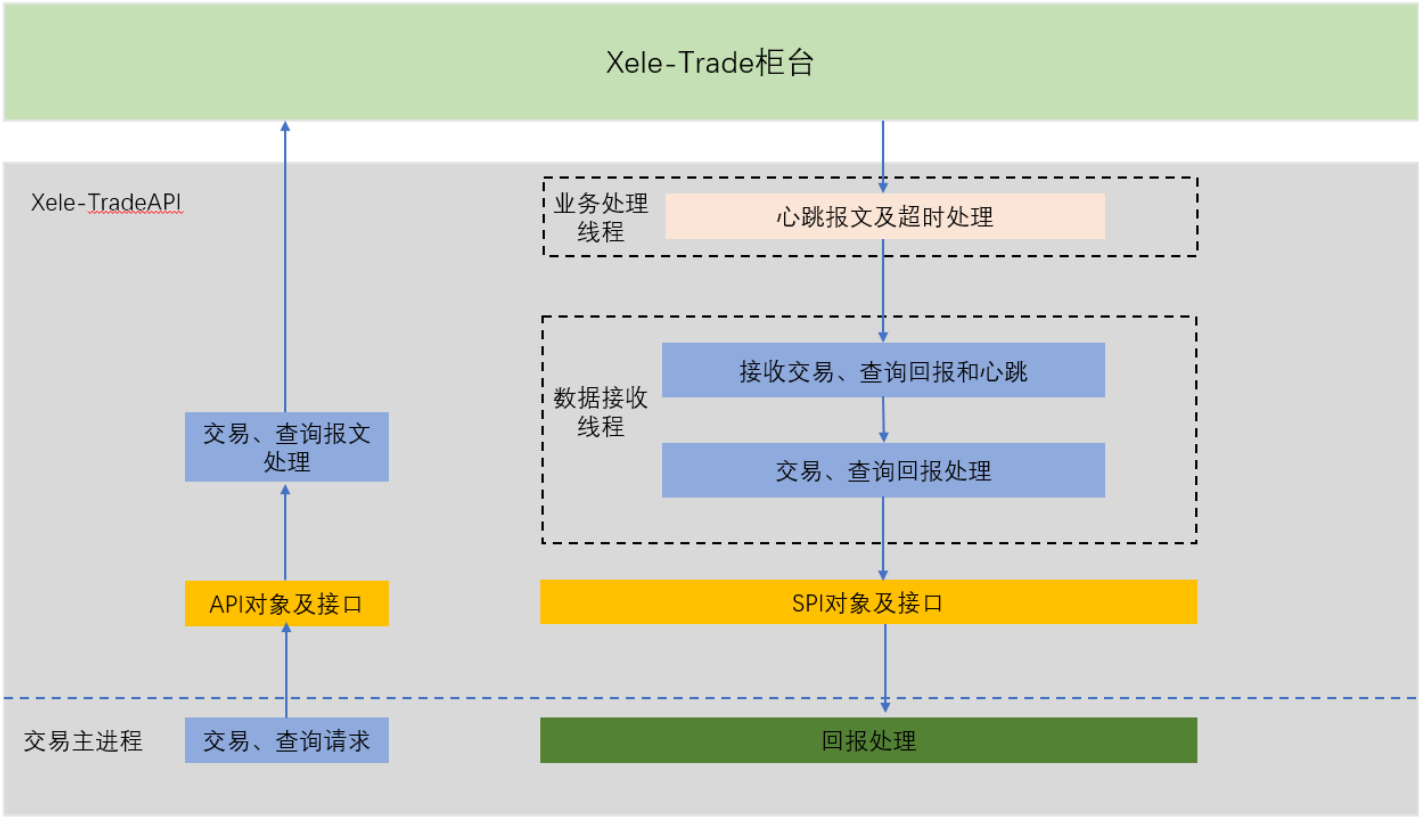
回应：

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报单预热线程：主要是处理交易预热功能，建议与用户报单线程绑到同一颗CPU物理核。此线程只有启用预热报单时才会运行，否则不会运行；
API业务处理线程：主要是处理心跳、重连等公共业务逻辑；
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，也不能在多线程同时发送报单请求。
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：

```
############################################################
## 创建API实例时，请使用独立的配置文件。
## 多个账号请创建多个API实例，每个API实例使用各自不同的配置文件。
############################################################

## 资金账号
ACCOUNT_ID=account_1

## 账号密码
ACCOUNT_PWD=password

## 看穿式监管需要的APP_ID
APP_ID=app-client-id

## 看穿式监管需要的授权码
AUTH_CODE=user-auth-code

## 查询使用的地址和端口（TCP）
QUERY_SERVER_IP=127.0.0.1
QUERY_SERVER_PORT=33333

## 交易使用的地址和端口（UDP）
TRADE_SERVER_IP=127.0.0.1
TRADE_SERVER_PORT=62000

## 回报数据处理线程绑核，默认不绑核，该线程以busy loop模式运行
## 回报数据处理线程的绑核，可以加快数据的接收和处理，
## 建议为每个API实例绑定一个隔离的CPU核。
#TCP_WORKER_CORE_ID=3

## 回报数据处理线程是否启用busy loop模式运行，默认为true
## 不建议启用此配置项，会增加回报处理的时延。
#TCP_WORKER_BUSY_LOOP_ENABLED=true

## UDP预热处理时间间隔，单位：毫秒，取值范围：[10,50]，默认为20
## 不建议启用此配置项，保持默认值。
#WARM_INTERVAL=20

## UDP预热处理线程绑核，默认不绑核，该线程不是以busy loop模式运行
## 建议将预热线程与用户策略线程绑在同一个物理CPU上。
#TRADE_WORKER_CORE_ID=4

## 公共处理线程绑核，默认不绑核，该线程不是以busy loop模式运行
## 公共处理线程主要用于处理通用任务，例如：
## - 慢速数据处理；
## - 心跳超时维护；
## - 超时重连处理；
## - 其他任务，等；
## 说明：
## - 负数表示不绑核；
## - 公共处理线程是多个API实例共用的，只需要在不同的API实例配置文件中设置一次即可，以第一个API实例创建的配置生效；
## 不建议使用此配置项，保持默认值。
#TASK_WORKER_CORE_ID=5
```

注意：用户也可以通过setConfig()接口设置自定义的临时参数，以便在需要的地方使用 getConfig() 接口进行获取。此功能可以用作临时的参数传递；

心跳机制

Xele-Trade-Futures柜台系统在登录成功后，柜台每隔一定时间发送心跳报文给客户端来维持连接；API也会向柜台定时发送心跳报文来维持链接，默认心跳间隔为6秒。超过60秒钟，未收到柜台心跳，API会主动断开与柜台的连接，并等待下一次重连。API和柜台系统之间的心跳自动维护，无需用户干预。

重连机制

API登录Xele-Trade-Futures柜台系统后，如果出现异常中断，会自动发起重连。首次重连的间隔是100ms，随后的重连时间间隔为200ms、400ms、……，直至最大间隔32s。最大重连次数为9600次。如果API在中断后，尝试9600次依然无法成功建立连接，则停止重连，并通过XTFSpi::onError()接口通知用户。

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example中"Example01.cpp"）：

```
class Example_01_Trader : public ExampleTrader {
public:
    Example_01_Trader() = default;
    ... // 其余代码参见example/Example01.cpp

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

    ... // 其余代码参见example/Example01.cpp
}

/**
 * @brief 一个简单的报撤单功能演示，API登录柜台后，报一手多头开仓单，等待3秒后，尝试撤单。
 *
 * @param config
 * @param instrumentId
 */
void runExample(const std::string &configPath, const std::string &instrumentId, double price, int volume) {
    printf("start example 01.\n");

    Example_01_Trader trader;
    trader.setConfigPath(configPath);
    trader.setInstrumentID(instrumentId);
    trader.setPrice(price);
    trader.setVolume(volume);

    trader.start();
    while (!trader.isStarted())
```

```
        trader.wait(1, "wait for trader started");
    ... // 其余代码参见example/Example01.cpp
}

int main(int argc, const char *argv[]) {
    printf("api version: %s.\n", getXTFVersion());

    // 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";
    double price = 301.50;
    int volume = 1;
    runExample(configPath, instrumentId, price, volume);
    return 0;
}
```

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。

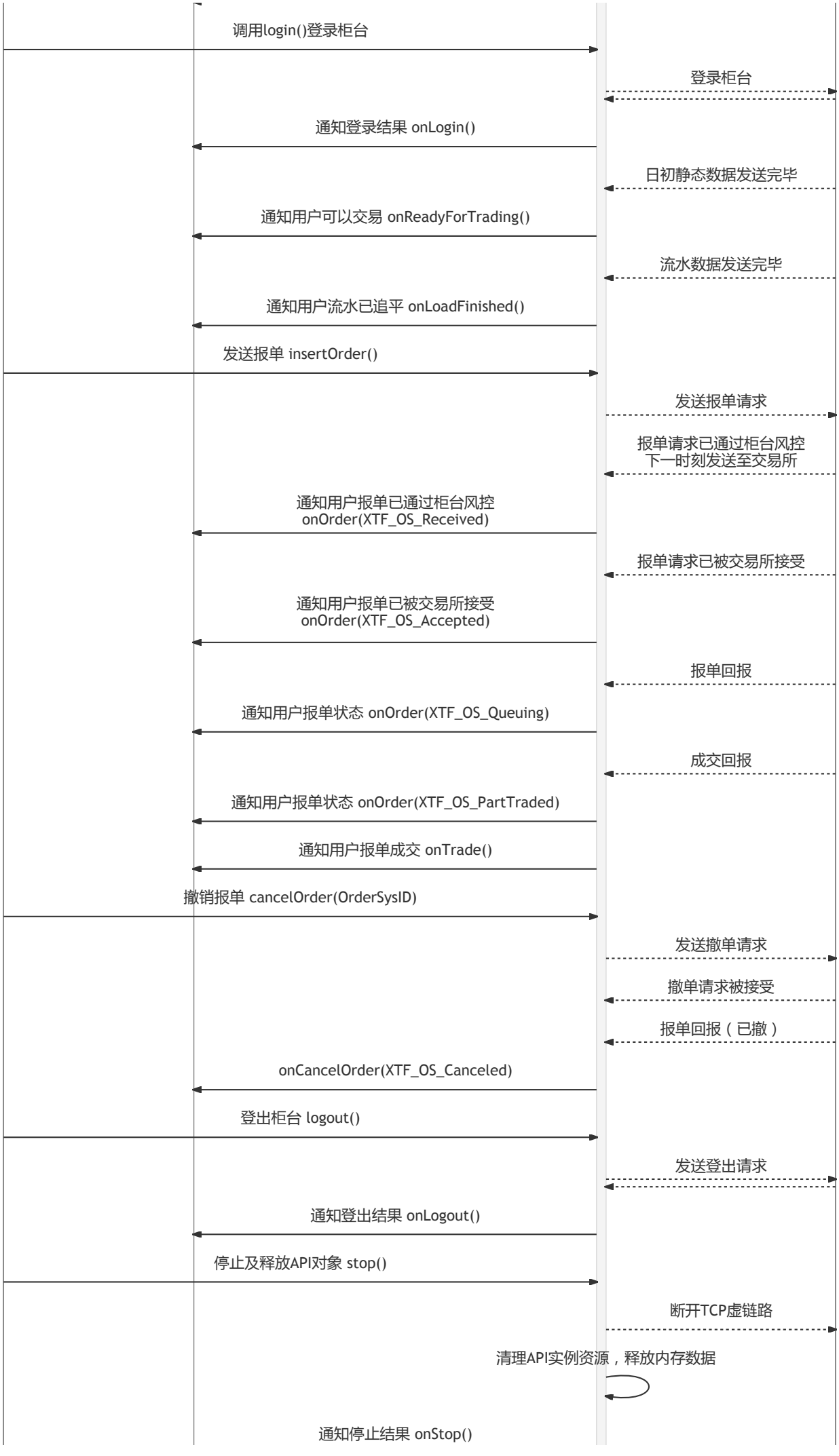
示例代码：功能调用及回调函数（具体代码详见example中“Example01.cpp”）

```
// 客户端程序和柜台建立连接后，调用报单操作接口。
trader.start();
... // 其余代码参见example/Example01.cpp
trader.insertOrder();
... // 其余代码参见example/Example01.cpp
trader.stop();
```

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。

事件时序图

下图是API实例在生命期内调用接口及触发的事件时序图：


调用login()登录柜台
登录柜台
通知登录结果 onLogin()
日初静态数据发送完毕
通知用户可以交易 onReadyForTrading()
流水数据发送完毕
通知用户流水已追平 onLoadFinished()
发送报单 insertOrder()
发送报单请求
报单请求已通过柜台风控
下一时刻发送至交易所
通知用户报单已通过柜台风控
onOrder(XTF_OS_Received)
报单请求已被交易所接受
通知用户报单已被交易所接受
onOrder(XTF_OS_Accepted)
报单回报
通知用户报单状态 onOrder(XTF_OS_Queuing)
成交回报
通知用户报单状态 onOrder(XTF_OS_PartTraded)
通知用户报单成交 onTrade()
撤销报单 cancelOrder(OrderSysID)
发送撤单请求
撤单请求被接受
报单回报（已撤）
onCancelOrder(XTF_OS_Canceled)
登出柜台 logout()
发送登出请求
通知登出结果 onLogout()
停止及释放API对象 stop()
断开TCP虚链路
清理API实例资源，释放内存数据
通知停止结果 onStop()

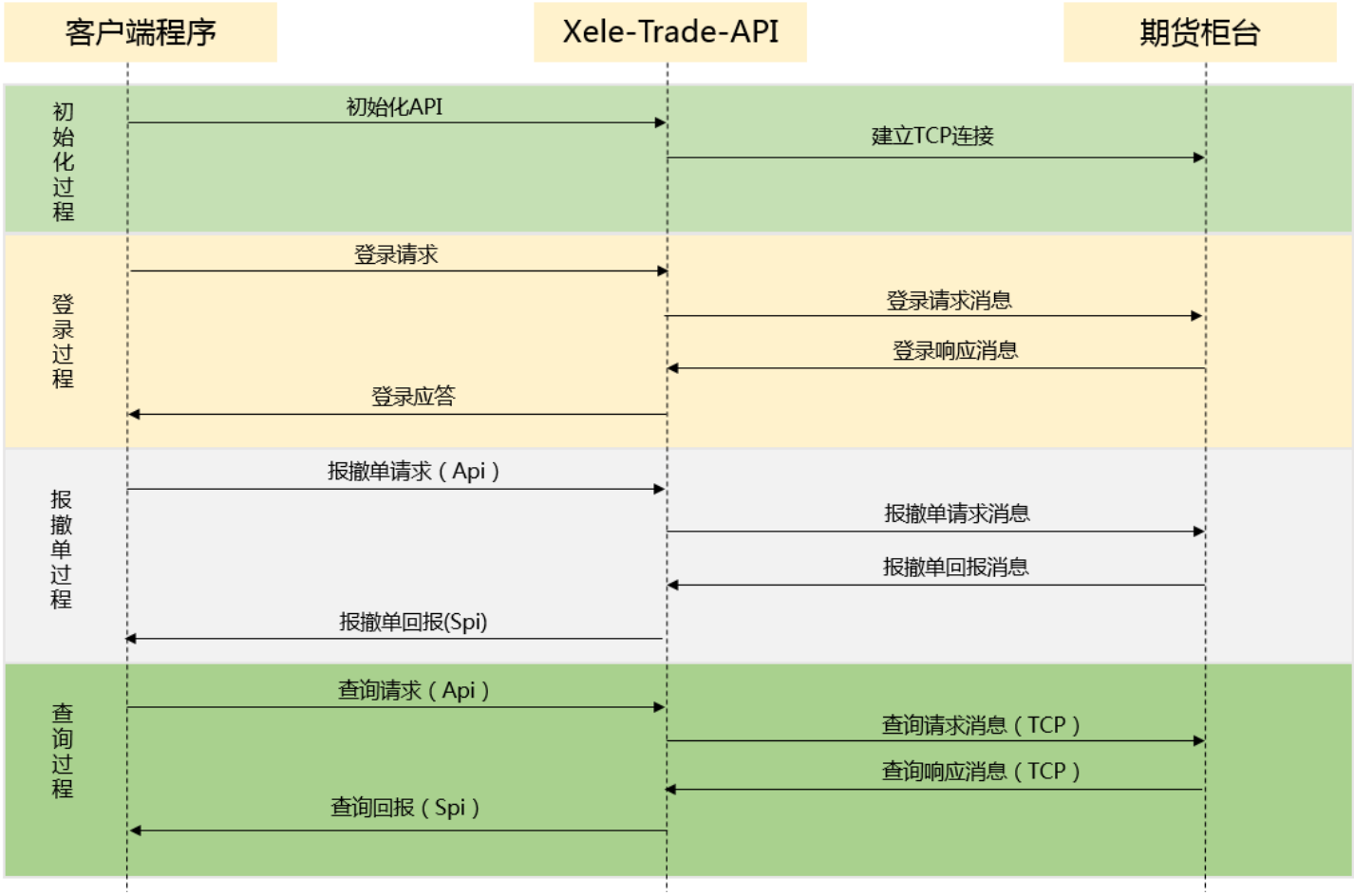
TraderAPI基本操作说明

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响应结果功能；
交易服务：主要是处理报单、撤单以及接收交易回报等功能；

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

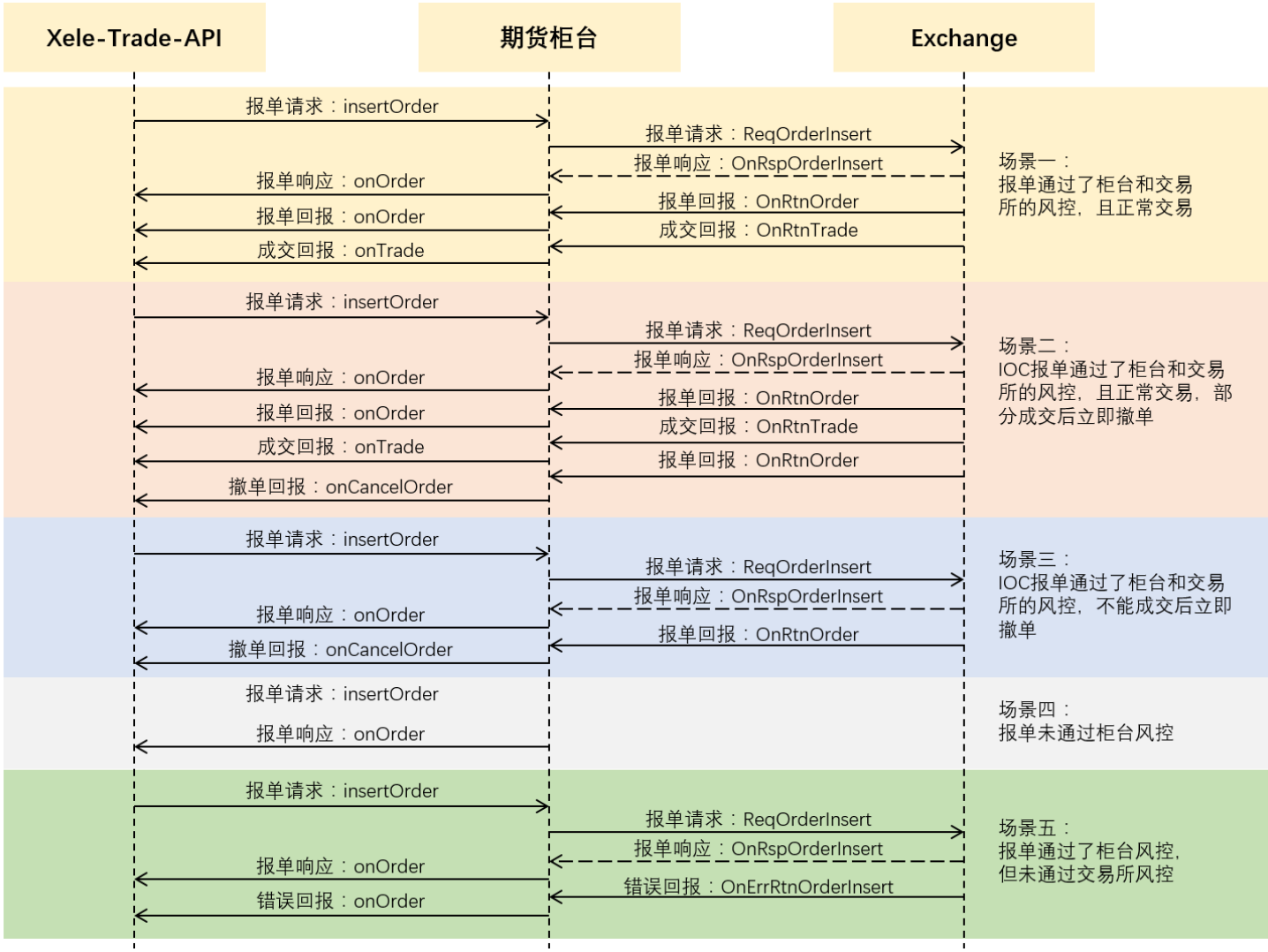
初始化网络连接后，就可以通过login()接口发起登录请求了，在onLogin()回调接口处理登录响应，只有登录成功后才能进行业务处理。Xele-Trade-Futures柜台系统支持用户**多点登录**（默认最多登录5次），同一个用户每次可创建多个API实例，初始化完成后即可以进行登录。多点登录有以下特点：

- 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连接；
- 交易数据流的下行报文是发送给该用户的所有登录点的连接；
- 每个登录点的查询数据流或交易数据流上下行连接如果断开，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几种场景。

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和onCancelOrder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；如果报单状态为`XTF_OS_Canceled`则通过onCancelOrder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；

示例代码：报单示例代码如下（具体代码详见example中“Example02.cpp”）：

```
int insertOrder() { // 报单
   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 = 1000.0;
   order.volume = volume;
   order.channelSelectionType = XTF_CS_Auto;
   order.orderLocalID = ++mOrderLocalId;
   return mApi->insertOrder(order);
}
```

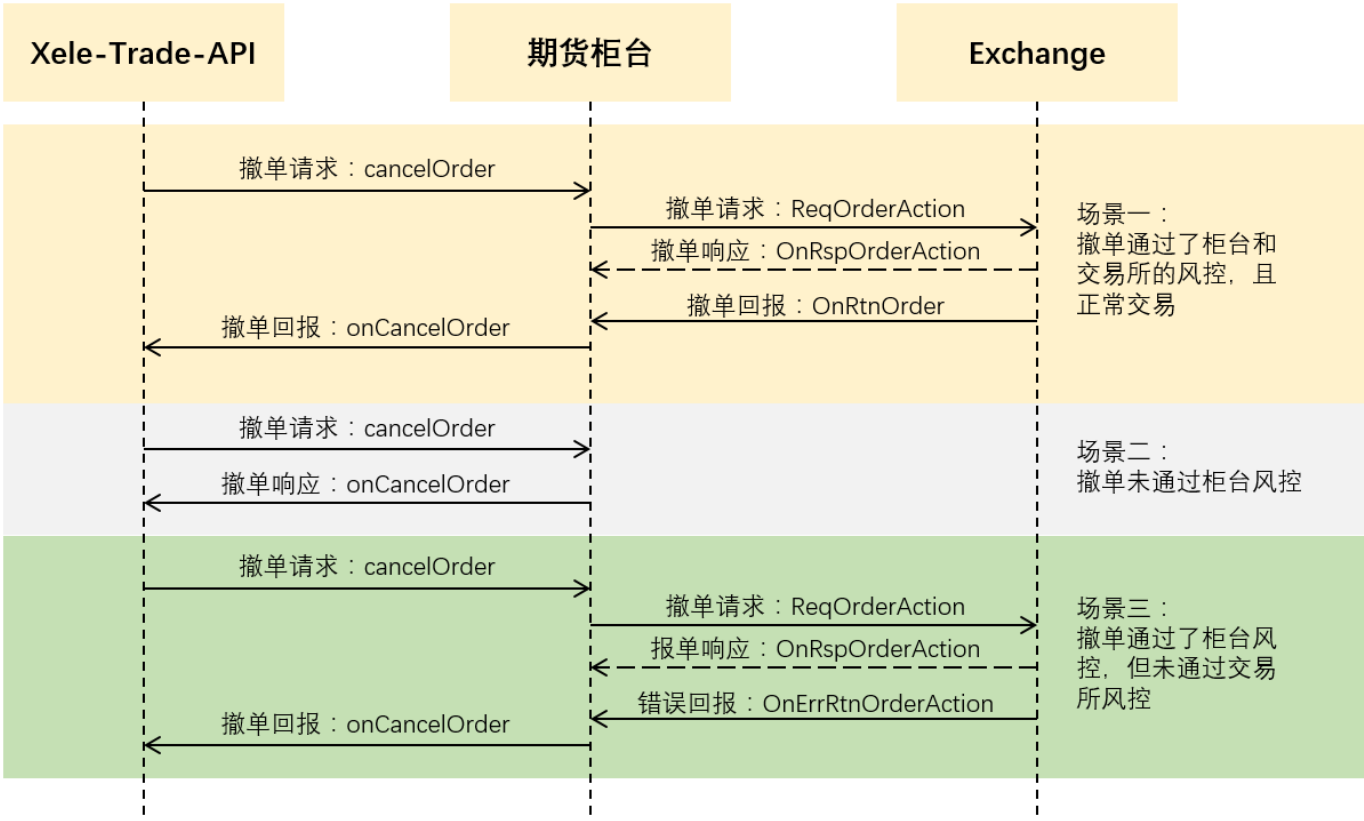
```
void onOrder(int errorCode, const XTFOrder *order) override {
    // 报单失败。根据报单错误码判断是柜台拒单，还是交易所拒单。
    if (errorCode != 0) {
        switch (order->actionType) {
            case XTF_OA_Insert:
                printf("insert order failed, error: %d\n", errorCode);
                break;
            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。根据报单状态判断处理逻辑。
    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_Queuing:
            printf("order queuing.\n");
            break;
        case XTF_OS_Rejected:
            printf("order rejected.\n");
            break;
        default:
            break;
    }
}

void onCancelOrder(int errorCode, const XTFOrder *cancelOrder) override {
    // 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 XTFTrade *trade) override {
    // 收到交易回报
    printf("recv trade, sys-order-id: %d, trade-id: %ld.\n", trade->order->sysOrderID, trade->tradeID);
}
```

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onCancelOrder接口通知用户。如果撤单时已有部分成交，则会通过onOrder和onTrade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onCancelOrder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：撤单示例代码如下（具体代码详见example中"Example02.cpp"）：

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

void onCancelOrder(int errorCode, const XTFOrder *cancelOrder) override {
    // 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);
}
```

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 流水数据已加载完毕，可进行资金和仓位操作	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 XTFApi::insertOrders XTFSpi::onOrder
交易接口	撤单及订单状态改变通知接口	XTFApi::cancelOrder XTFApi::cancelOrders XTFSpi::onCancelOrder
交易接口	成交回报通知接口	XTFSpi::onTrade
行权对冲接口	报撤单、回报及订单状态改变通知接口	XTFApi::insertExecOrder XTFApi::cancelExecOrder XTFSpi::onExecOrder XTFSpi::OnCancelExecOrder
做市接口	询价、应答接口	XTFApi::demandQuote XTFSpi::onDemandQuote
数据变化通知接口	账户出入金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	XTFSpi::onExchange
数据变化通知接口	合约发生变化时回调该接口	XTFSpi::onInstrument
外接行情接口	行情发生变化时回调该接口	XTFSpi::onBookUpdate
外接行情接口	订阅行情接口 取消订阅行情	XTFApi::subscribe XTFApi::unsubscribe
外接行情接口	更新合约行情	XTFApi::updateBook
查询管理接口	同步资金仓位信息，同步接口 调用成功后自动更新XTFAccount中的资金信息	XTFApi::syncFunds
查询管理接口	获取当前账户信息	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
查询管理接口	查找成交信息	XTFApi::findTrades
参数配置管理接口	设置配置参数	XTFApi::setConfig
参数配置管理接口	查询配置参数	XTFApi::getConfig

业务类型	业务	请求接口 / 响应接口
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	XTFSpi::onEvent
其他通用接口	错误通知接口	XTFSpi::onError
辅助接口	构造预热报单	XTFApi::buildWarmOrder

SPI类管理接口

生命期管理接口

onStart方法

接口功能

API对象启动通知接口

接口原型

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

接口参数

- errorCode：启动结果。0-表示启动成功，可以调用 login() 接口登录柜台；其他值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

接口功能

API对象停止通知接口

接口原型

```
void onStop(int reason);
```

接口参数

- reason：表示停止的原因；

```
ERRXTFAPI_ClosedByUser          : 客户调用stop接口主动关闭
ERRXTFAPI_ClosedByTimeout       : 心跳超时主动关闭
ERRXTFAPI_ClosedBySendError     : 发送数据时，检测到套接字异常，主动关闭
ERRXTFAPI_ClosedByRecvError     : 检测到对端关闭TCP连接
ERRXTFAPI_ClosedByLoginError    : 检测到登录应答协议异常后主动关闭
ERRXTFAPI_ClosedByLogoutError   : 检测到登出应答协议异常后主动关闭
ERRXTFAPI_ClosedByZipStreamError: 检测到流水推送数据异常后主动关闭
```

返回值

无

onServerReboot方法

接口功能

柜台在交易时段发生过重启的通知接口。

- 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。

接口原型

```
void onServerReboot();
```

接口参数

无

返回值

无

会话管理接口

onLogin方法

接口功能

登录结果通知接口

接口原型

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

接口参数

- errorCode：登录结果。0-表示登录成功；其他值表示登录失败返回的错误码；
- exchangeCount：交易所数量，登录成功后返回交易所的可用数量；此时`不能查询交易席位`信息。需要收到 onReadyForTrading() 或 onLoadFinished() 事件通知后，才能查询交易所的交易席位信息。

返回值

无

onLogout 方法

接口功能

登出结果通知接口。

接口原型

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

接口参数

- errorCode：登出结果。0-表示登录成功；其他值表示登出失败返回的错误码；

返回值

无

onChangePassword方法

接口功能

修改密码结果通知接口。

接口原型

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

接口参数

- errorCode：修改结果。0-表示修改成功；其他值表示密码修改失败返回的错误码；

返回值

无

onReadyForTrading 方法

接口功能

日初静态数据已加载完毕，可进行报单操作。

初静态数据已经加载完毕，用户可以根据需要查询合约对象，并进行报单；但是，`报单回报`会在日内历史流水追平之后，才会到达。

注意，至此阶段日内历史流水数据并未加载完成，暂时无法计算资金和仓位，如果需要根据资金和仓位进行报单，需要等待流水追平之后，再发送报单。

接口原型

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

接口参数

- account ：当前登录的账户信息，参见XTFAccount结构体；在API实例生命周期内，此对象指针一直有效，可以保存使用；该对象指针与getAccount()接口查询的一致；

返回值

无

onLoadFinished方法

接口功能

交易日内所有流水数据已加载完毕，用户可以根据流水数据计算资金和仓位，并据此进行报单。

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；

接口原型

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

接口参数

- account ：当前登录的账户信息，参见XTFAccount结构体；

返回值

无

交易接口

onOrder方法

接口功能

报单回报及订单状态改变通知接口。

接口原型

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

接口参数

- errorCode：报单操作错误码
- order：报单回报对象，参见XTFOrder结构体；

返回值

无

onCancelOrder方法

接口功能

撤单回报状态改变通知接口。

主动撤单成功或者失败、IOC订单被交易所撤单都会产生该事件，撤单成功时该订单orderStatus为XTF_OS_Canceled。

说明：

1. 该事件在流水重传的时候也会产生，此时isHistory为true;
2. 流水数据支持断点续传功能；
3. 如果是新创建的API对象，流水数据从头开始推送；
4. 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；

接口原型

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

接口参数

- errorCode：撤单操作错误码；
- order：报单回报对象，参见XTFOrder结构体；

返回值

无

onTrade方法

接口功能

成交回报通知接口。

1. 该事件在流水重传的时候也会产生,此时isHistory为true;
2. 流水数据支持断点续传功能；
3. 如果是新创建的API对象，流水数据从头开始推送；
4. 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；

成交回报（XTFTrade）对象被创建后，在API实例的生命周期内是一直有效的（如果柜台发生重启，则对象失效）。

虽然XTFTrade对象在创建后，其字段不再更新，但是成交回报关联的XTFOrder对象和成交对应的仓位明细会发生变化。

如果需要对成交回报进行异步处理，建议将XTFTrade的字段拷贝到用户管理的内存之后，再做下一步处理。

如果用户不关心报单状态、已成交数量及成交明细，那么也可以直接使用XTFTrade指针。

接口原型

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

接口参数

- trade：成交回报对象，参见XTFTrade结构体；

返回值

无

onExecOrder方法

接口功能

行权或对冲报单回报及订单状态改变通知接口。

行权或对冲报单回报，没有以下几个状态：

- XTF_OS_Queuing
- XTF_OS_PartTraded
- XTF_OS_AllTraded

接口原型

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

接口参数

- errorCode：报单操作错误码
- order：报单回报对象，参见XTFOrder结构体；

返回值

无

onCancelExecOrder方法

接口功能

行权或对冲撤单回报状态改变通知接口。

主动撤单成功或者失败会产生该事件，撤单成功时该订单orderStatus为XTF_OS_Canceled

说明：

1. 该事件在流水重传的时候也会产生，此时isHistory为true;
2. 流水数据支持断点续传功能；
3. 如果是新创建的API对象，流水数据从头开始推送；
4. 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；

接口原型

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

接口参数

- errorCode：撤单操作错误码；
- order：报单回报对象，参见XTFOrder结构体；

返回值

无

onPositionCombEvent方法

接口功能

持仓组合回报事件通知接口。

如果启用自动组合功能，当持仓被组合或解锁组合时，会产生组合回报事件，通过此接口通知用户。

接口原型

```
void onPositionCombEvent(int errorCode, const XTFPositionCombEvent &combEvent);
```

接口参数

- errorCode：错误码，作为保留；
- combEvent：组合事件对象，参见XTFPositionCombEvent结构体；

返回值

无

onDemandQuote方法

接口功能

询价应答接口。

询价请求成功或者失败都会产生该事件，即调用 demandQuote() 方法后，询价结果由此接口通知用户。

接口原型

```
void onDemandQuote(int errorCode, const XTFInputQuoteDemand &inputQuoteDemand);
```

接口参数

- errorCode：询价操作错误码；
- inputQuoteDemand：询价录入对象，用于返回本次询价请求的输入参数。该对象只能在本接口内部使用，如果在接口之外使用，可以创建对象副本后，对副本做进一步处理。详细定义参见XTFInputQuoteDemand结构体。

返回值

无

数据变化通知接口

onAccount方法

接口功能

账户出入金发生变化时回调该接口。

接口原型

```
void onAccount(int event, int action, const XTFAccount *account);
```

接口参数

- event：账户变化事件类型，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，参见XTFAccount结构体；

返回值

无

onExchange方法

接口功能

交易所前置状态发生变化时回调该接口。

接口原型

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

接口参数

- 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

接口功能

合约发生变化时回调该接口。

接口原型

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

接口参数

- 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，参见XTFInstrument结构体；

返回值

无

外接行情接口

onBookUpdate方法

接口功能

行情发生变化时回调该接口。

默认只通知用户subscribe()指定的合约行情，参见subscribe()接口。

接口原型

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

接口参数

- marketData：行情信息对象，参见XTFMarketData结构体；

返回值

无

其他通用接口

onEvent方法

接口功能

事件通知接口。

接口原型

```
void onEvent(const XTFEvent &event);
```

接口参数

- event：通知的事件对象，参见XTFEvent结构体；

返回值

无

onError方法

接口功能

错误通知接口。

接口原型

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

接口参数

- errorCode：错误码，ERRXTFAPI_RetryMaxCount：TCP重连失败且已达最大次数。
- data：附带的错误数据，无数据则为nullptr；
- size：附带的错误数据大小，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

接口功能

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();
```

接口原型

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

接口参数

- 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。

stop方法

接口功能

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

接口原型

```
int stop();
```

接口参数

无

返回值

接口调用成功返回0，否则返回错误码。

login方法

接口功能

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；

接口原型

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

接口参数

- 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

接口功能

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。

接口原型

```
int logout();
```

接口参数

无

返回值

接口调用成功返回0，否则返回错误码。

changePassword方法

接口功能

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。

接口原型

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

接口参数

- 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，密码修改成功与否的结果在 onChangePassword(errorCode) 接口中通知用户；

交易接口

insertOrder方法

接口功能

发送报单。

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

重要提示：

> 1. 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；
> 2. 该接口默认是线程不安全的，不能在多线程调用同一个API实例对象的报单接口；如果需要启用线程安全模式，使用setConfig("ORDER_MT_ENABLED", "true")进行配置即可；

接口原型

```
int insertOrder(const XTFInputOrder &inputOrder);
int insertOrders(const XTFInputOrder inputOrders[], size_t orderCount);
```

接口参数

- inputOrder：待插入的报单参数，参见XTFInputOrder结构体；
- inputOrders：待批量插入的报单参数数组；
- orderCount：批量插入数量，不能大于16个。如果超过16，则批量报单插入失败；

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。

cancelOrder方法

接口功能

发送撤单。

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

重要提示：

> 1. 该接口默认是线程不安全的，不能在多线程调用同一个API实例对象的报单接口；如果需要启用线程安全模式，使用setConfig("ORDER_MT_ENABLED", "true")进行配置即可；
> 2. 如果使用XTFOrder报单对象直接撤单，务必使用API返回的XTFOrder对象指针，用户不能从外部构造一个XTFOrder来撤单；
> 3. 不建议使用交易所单号撤单；

接口原型

```
int cancelOrder(const XTFOrder *order);
int cancelOrders(const XTFOrder *orders, size_t orderCount); // 批量撤单接口，最大数量为16个
int cancelOrder(XTFOrderIDType orderIDType, long orderID); // 如果是本地单号撤单，需要用户具备本地单号撤单的权限
int cancelOrders(XTFOrderIDType orderIDType, long orderIDs[], size_t orderCount); // 批量撤单接口，最大数量为16个
```

接口参数

- order：待撤销的报单对象，参见XTFOrder结构体；
- orders：待批量撤销的报单对象数组；
- orderCount：批量撤单数量，不能大于16个。如果超过16，则批量撤单失败；
- orderIDType：报单编号类型，目前支持柜台流水号、本地报单编号和交易所单号撤单；
- 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。

关于本地单号撤单的说明

为了确保本地单号撤单可以正常使用，有如下的要求：

1. 需要账号支持本地单号撤单功能，可以通过 XTFAccount::isSupportCancelOrderByLocalID()接口查看当前账号是否支持本地单号撤单；
2. 本地单号在[0, X]范围之内，X的值默认是5000000，柜台可以根据需要进行配置，用户可以通过XTFAccount::getMaxAllowedLocalOrderID()接口查询该值；
3. 本地单号保持单调递增，普通单、报价单、行权对冲单，都不能冲突；

insertExecOrder方法

接口功能

发送行权或对冲报单。

在 XTFSpi::onExecOrder()或XTFSpi::onCancelExecOrder() 接口中通知用户行权或对冲报单的结果和状态。

重要提示：

> 1. 行权或对冲的本地报单编号与普通订单的本地报单编号，不能冲突，需要进行统一编号；
> 2. 该接口默认是线程不安全的，不能在多线程调用同一个API实例对象的报单接口；如果需要启用线程安全模式，使用setConfig("ORDER_MT_ENABLED", "true")进行配置即可；

接口原型

```
int insertExecOrder(const XTFExecOrder &execOrder);
```

接口参数

- execOrder：待发送的行权或对冲报单参数，参见XTFExecOrder结构体；

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。

行权或对冲报单字段说明

通过XTFOrderFlag、XTFOrderType、XTFDirection三个字段区分是请求行权、放弃行权、请求对冲、请求不对冲。下面的表格给出了不同报单对应的字段值：

报单类型	XTFOrderFlag	XTFDirection	XTFOrderType
请求行权（且行权后自对冲期权）	XTF_ODF_OptionsExecute	XTF_D_Buy	XTF_ODT_SelfClose
请求行权（且行权后不对冲期权）	XTF_ODF_OptionsExecute	XTF_D_Buy	XTF_ODT_NotSelfClose
放弃行权	XTF_ODF_OptionsExecute	**XTF_D_Sell**	—
请求期权对冲	XTF_ODF_OptionsSelfClose	XTF_D_Buy	XTF_ODT_SelfCloseOptions
请求履约对冲	XTF_ODF_OptionsSelfClose	XTF_D_Buy	XTF_ODT_SelfCloseFutures
请求期权**不**对冲	XTF_ODF_OptionsSelfClose	**XTF_D_Sell**	XTF_ODT_SelfCloseOptions
请求履约**不**对冲	XTF_ODF_OptionsSelfClose	**XTF_D_Sell**	XTF_ODT_SelfCloseFutures

按照上述表格构造行权或对冲报单后，通过`insertExecOrder(const XTFExecOrder &execOrder)`接口即可实现报单请求；对应的撤单请求使用`cancelExecOrder(const XTFOrder *order)`接口。

cancelExecOrder方法

接口功能

发送行权/自对冲撤单。

在XTFSpi::onCancelExecOrder()接口中通知撤单的结果和报单状态。

重要提示：

> 1. 该接口默认是线程不安全的，不能在多线程调用同一个API实例对象的报单接口；如果需要启用线程安全模式，使用setConfig("ORDER_MT_ENABLED", "true")进行配置即可；
> 2. 如果使用XTFOrder报单对象直接撤单，务必使用API返回的XTFOrder对象指针，用户不能从外部构造一个XTFOrder来撤单；
> 3. 不建议使用交易所单号撤单；

接口原型

```
int cancelExecOrder(const XTFOrder *order);
int cancelExecOrder(XTFOrderIDType orderIDType, long orderID);
```

接口参数

- order：待撤销的报单对象，参见XTFOrder结构体；
- orderIDType：报单编号类型，目前支持柜台流水号、本地报单编号和交易所单号撤单；
- 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。

关于本地单号撤单的说明

与普通撤单相同，参考cancelOrder的本地单号撤单说明。

demandQuote方法

接口功能

发送询价请求。

在XTFSpi::onDemandQuote()接口中通知询价的结果。

重要提示：

> 该接口默认是线程不安全的，不能在多线程调用同一个API实例对象的报单接口；如果需要启用线程安全模式，使用setConfig("ORDER_MT_ENABLED", "true")进行配置即可。

接口原型

```
int demandQuote(const XTFInstrument *instrument, int localID = 0);
int demandQuote(const XTFInputQuoteDemand &inputQuoteDemand);
```

接口参数

- instrument：待询价的合约对象。必须传入一个有效的合约对象，不可传入空对象；
- localID：询价请求的本地编号，无规则约束，由用户自定义。可选参数，默认为0；
- inputQuoteDemand：询价请求对象。该对象提供了询价更丰富的输入参数，如：可以指定询价的发送席位编号、用户自定义数据等。详细信息参见XTFInputQuoteDemand结构体。

返回值

接口调用成功返回0，否则返回错误码。

调用成功仅表示本次询价请求发送成功，并不表示询价操作是成功的，询价结果在 XTFSpi::onDemandQuote() 接口中返回。

外接行情接口

subscribe方法

接口功能

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。

说明：

1. API自身是不带有行情接入功能，但是提供了外部行情更新接口updateBook()，通过此接口更新API内部存储的最新行情数据；
2. 如果没有订阅合约行情，当updateBook()导致的行情变化时，不会回调onBookUpdate()接口；
3. 如果订阅了合约行情，当updateBook()导致合约行情变化时，会回调onBookUpdate()接口；

接口原型

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

接口参数

- instrument：待订阅的合约对象，参见XTFInstrument结构体；

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

接口功能

取消订阅行情，与subscribe()接口功能相反。

接口原型

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

接口参数

- instrument：待取消订阅的合约对象，参见XTFInstrument结构体；

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

接口功能

更新合约行情。
暂时仅支持一档行情数据。
更新行情后，API会自动处理以下逻辑：

1. 根据合约仓位计算持仓盈亏；
2. 如果用户调用subscribe()接口订阅行情数据，会通过 XTFSpi::onBookUpdate() 接口通知用户；

接口原型

```
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。
);
```

接口参数

- instrument：合约对象，参见XTFInstrument结构体；
- 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

接口功能

获取当前资金账户信息。

接口原型

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

接口参数

无

返回值

返回资金账户对象指针。

getExchangeCount方法

接口功能

获取交易所数量信息。

接口原型

```
int getExchangeCount();
```

接口参数

无

返回值

返回交易所的数量，负数表示错误码。

getExchange方法

接口功能

获取指定位置的交易所信息。

接口原型

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

接口参数

- pos：交易所对象的位置下标，从0开始计算，最大值为 getExchangeCount() - 1；

返回值

返回指定位置的交易所对象，参见XTFExchange结构体。

getInstrumentCount方法

接口功能

获取合约数量信息。

接口原型

```
int getInstrumentCount();
```

接口参数

无

返回值

- 大于等于0：表示合约数量；
- 小于0：表示查询发生的错误码；

getInstrument方法

接口功能

根据合约位置查询合约信息。

接口原型

```
const XTFInstrument* getInstrument(int pos);
```

接口参数

- pos：合约的位置下标值，取值范围：[0, getInstrumentCount() - 1]；

返回值

- 合约对象指针，参见XTFInstrument结构体；
- 如果参数异常，或者查询发生错误，返回空指针；

getInstrumentByID方法

接口功能

根据合约编号字符串查询合约信息。

接口原型

```
const XTFInstrument* getInstrumentByID(const char *instrumentID);
```

接口参数

- instrumentID：合约编号字符串，例如：`sc2305`、`cu2301`等；

返回值

- 合约对象指针，参见XTFInstrument结构体；
- 如果合约不存在，或者查询发生错误，返回空指针；

getCombInstrumentCount方法

接口功能

获取组合合约数量信息。

接口原型

```
int getCombInstrumentCount();
```

接口参数
无

返回值

- 大于等于0：表示组合合约数量；
- 小于0：表示查询发生的错误码；

getCombInstrument方法

接口功能

根据位置查询组合合约信息。

接口原型

```
const XTFCombInstrument* getCombInstrument(int pos);
```

接口参数

- pos：位置下标值，取值范围：[0, getCombInstrumentCount() - 1]；

返回值

- 组合合约对象指针，参见XTFCombInstrument结构体；
- 如果参数异常，或者查询发生错误，返回空指针；

getCombInstrumentByID方法

接口功能

根据组合合约编号字符串查询组合合约信息。

接口原型

```
const XTFCombInstrument* getCombInstrumentByID(const char *combInstrumentID);
```

接口参数

- combInstrumentID：组合合约编号字符串，例如：`sc2305,-sc2305`、`-cu2301,cu2302`等；

返回值

- 组合合约对象指针，参见XTFCombInstrument结构体；
- 如果合约不存在，或者查询发生错误，返回空指针；

getProductCount方法

接口功能

获取品种数量信息。

接口原型

```
int getProductCount();
```

接口参数

无

返回值

- 大于等于0：表示品种数量；
- 小于0：表示查询发生的错误码；

getProduct方法

接口功能

根据位置查询品种信息。

接口原型

```
const XTFProduct* getProduct(int pos);
```

接口参数

- pos：位置下标值，取值范围：[0, getProductCount() - 1]；

返回值

- 品种对象指针，参见XTFProduct结构体；
- 如果参数异常，或者查询发生错误，返回空指针；

getProductByID方法

接口功能

根据品种编号字符串查询品种信息。

接口原型

```
const XTFProduct* getProductByID(const char *productID);
```

接口参数

- productID：合约编号字符串，例如：`sc`、`cu`等；

返回值

- 品种对象指针，参见XTFProduct结构体；
- 如果品种不存在，或者查询发生错误，返回空指针；

findOrders方法

接口功能

根据指定的过滤条件查询报单信息。

接口原型

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

接口参数

- 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

返回实际查询到的报单数量。
如果查询的实际报单数大于指定的数量count，则取前面的count个报单，并返回count；
如果查询的实际报单数小于指定的数量count，则表示查到的数量少于传入的数量，返回实际的数量；

findTrades方法

接口功能

根据指定的过滤条件查询成交信息。

接口原型

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

接口参数

- 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；
- trades：外部传入的数组，用于存储查询结果；

返回值

返回实际查询到的成交数量。
如果查询的实际成交数大于指定的数量count，则取前面的count个成交，并返回count；
如果查询的实际成交数小于指定的数量count，则表示查到的数量少于传入的数量，返回实际的数量；

参数配置管理接口

enableAutoCombinePosition方法

接口功能

启停仓位自动组合功能。
创建API实例后、启动API前，调用 enableAutoCombinePosition(true) 接口设置启用仓位自动组合功能。启用组合之后，API登录会发送自动组合标识到柜台。柜台会根据用户的当前持仓，按照组合规则进行自动组合。
应在启动会话之前设置自动组合功能，启动之后设置不会生效。默认为不启用仓位自动组合功能。

仓位自动组合说明：

1. 无论是否启用自动组合功能，上个交易日的历史仓位，柜台都会将满足组合规则的仓位全部组合；
2. 日内交易的新开仓位，如果用户登录时启用了仓位自动组合，那么剩余未组合的历史仓位（单腿）和日内新开仓位，柜台会将满足组合规则的仓位全部组合；
3. 如果日内有多次登录，且最近一次没有启用仓位自动组合，那么上一次登录时，已组合的仓位不再变化，本次会话过程中的新开仓位，柜台不会自动组合。

接口原型

```
void enableAutoCombinePosition(bool enabled);
```

接口参数

- enabled：是否启用自动组合功能，true-启用 false-不启用；

返回值

无

setConfig方法

接口功能

设置配置参数，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。

配置参数名称列表：

```
"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
"TRADE_SERVER_IP"：交易服务地址
"TRADE_SERVER_PORT"：交易服务端口
"QUERY_SERVER_IP"：查询服务地址
"QUERY_SERVER_PORT"：查询服务端口
"HEARTBEAT_INTERVAL"：心跳间隔时间，单位：毫秒
"HEARTBEAT_TIMEOUT"：心跳超时时间，单位：毫秒
"TCP_RECONNECT_ENABLED"：TCP断开后是否重连
"TCP_RETRY_INTERVAL"：TCP重连最小间隔时间，单位：毫秒
"TCP_RETRY_INTERVAL_MAX"：TCP重连最大间隔时间，单位：毫秒
"TCP_RETRY_COUNT"：TCP重连次数
"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
"TRADE_WORKER_CORE_ID"：报单处理线程绑核
"TASK_WORKER_CORE_ID"：通用任务线程绑核
"POSITION_CALC_ENABLED"：是否计算仓位
"MONEY_CALC_ENABLED"：是否计算资金，如果启用资金计算，会默认启用仓位计算
"HISTORY_ONLY_CALC_ENABLED"：是否仅计算历史流水的资金和仓位，如果启用，那么历史流水追平后，将不再计算资金和仓位
"WARM_INTERVAL"：预热时间间隔，取值范围：[10,50]，单位：毫秒
"ORDER_MT_ENABLED"：是否启用多线程报单功能，默认不启用多线程报单功能

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。会话内部不使用用户自定义的参数，仅对其临时存储。

接口原型

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

接口参数

- name：参数名称
- value：参数值

返回值

0表示参数配置成功，非0值表示失败。

getConfig方法

接口功能

查询配置参数。

接口原型

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

接口参数

- name：参数名称，参见`setConfig()`；

返回值

返回配置参数值字符串。
返回的字符串为临时字符串对象，如有需要请保存字符串值，再做后续的处理。

setReportFilter方法

接口功能

设置回报过滤规则（白名单）。

回报过滤规则分成两种条件：1）按品种类型过滤；2）按合约编号模糊匹配；

品种类型过滤目前仅区分：期货和期权两个类型。设置此过滤类型，是因为：若仅按合约编号进行模糊查找，可能同时匹配期货合约与期权合约，增加品种类型过滤，可以区分这两大类合约。

合约编号过滤规则，目前`通配符*`仅支持`后缀模式`，多个表达式使用逗号分割。例如：`au*,au23*`。

无效的合约过滤规则表达式如下：

```
a*2310  // 通配符在中间，不支持
*u2310  // 通配符在前面，不支持
a*23*2  // 多个通配符，且通配符不是后缀模式，不支持
```

有效的合约过滤规则表达式，但是会被改写成等价的规范形式：

```
au23**  // 有效，但会被改写成 au23*
```

约束条件：

- 回报过滤规则必须在login()接口被调用之前进行设置；
- 如果API已经登录，那么调用setReportFilter设置回报过滤，不会生效；重新登录后生效；

接口原型

```
int setReportFilter(const XTFReportFilter &filter);
```

接口参数

- filter：回报过滤规则；

返回值

0表示设置成功，非0表示本次操作失败对应的错误码。

getReportFilter方法

接口功能

查询生效的回报过滤规则（白名单）。

设置回报过滤规则后，可以通过此接口查询具体生效的回报过滤规则。设置与实际生效的规则，可能会有形式上的不同。

约束条件：

- 必须在会话登录期间的调用，才是有效的；
- 会话退出时，会自动清除所有规则；

接口原型

```
int getReportFilter(XTFReportFilter &filter)
```

接口参数

- filter：回报过滤规则；

返回值

0表示查询成功，非0表示本次查询失败对应的错误码。

getVersion方法

接口功能

获取API版本字符串。

接口原型

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

接口参数

无

返回值

版本字符串

辅助接口

syncFunds方法

接口功能

本接口使用同步方式向柜台查询资金。

调用约束：

- 本接口不能并发调用，上一次同步请求结束后，才能进行下一次调用；
- 调用接口会阻塞调用者线程，直到应答返回或超时返回；

使用建议：

- 如果没有外部行情接入，调用此接口可以同步柜台和本地计算的资金差异；
- 如果有外部极速行情，建议使用 updateBook() 的方式，在本地计算资金；
- 接入外部行情和调用资金同步接口，两种方式不要混用，以免出现资金错误；

接口原型

```
int syncFunds(int msTimeout = 45);
```

接口参数

- msTimeout：超时时间，单位：毫秒；默认为45毫秒超时，<=0 表示不允许超时；

返回值

- 0：表示资金同步成功，可以访问XTFAccount对象查看最新的资金数据；
- ETIMEDOUT：表示请求超时；
- <0：表示内部发生错误，可以联系技术支持查看具体错误信息；

buildWarmOrder方法

接口功能

传入合约参数，创建一个预热报单，写入用户提供的缓冲区之中。

默认创建的预热报单没有合约信息，如果需要携带合约信息，传入合约参数即可。合约可以通过getInstrumentByID()接口查询获得。

约束说明：

- 发送至柜台的预热报单每秒限制不超过50个，建议发送间隔为：20ms~50ms；
- 预热报单和真实报单建议使用同一个线程发送；

接口原型

```
int buildWarmOrder(void *buf, size_t size, const XTFInstrument *instrument = nullptr);
```

接口参数

- buf：用户提供的预热报单缓冲区；
- size：用户提供的预热报单缓冲区大小，不小于64字节。如果大于64字节，那么仅头部的64字节有效；
- instrument：合约指针，默认为空；

返回值

0-表示成功，非0表示对应的错误码。

结构体说明

XTFUserData结构体

说明：客户自定义数据对象类

```
class XTFAccount {
public:
    void           *userPtr;
    double          userDouble1;
    double          userDouble2;
    int             userInt1;
    int             userInt2;

    XTFUserData() {
        userPtr = nullptr;
        userDouble1 = 0.0;
        userDouble2 = 0.0;
        userInt1 = 0;
        userInt2 = 0;
    }
};
```

XTFAccount结构体

说明：客户/账户资金信息对象类

```
class XTFAccount {
public:
    XTFAccountID    accountID;                  ///< 账户编码
    double          preBalance;                 ///< 日初资金
    double          staticBalance;              ///< 静态权益
    double          deliveryMargin;             ///< 交割保证金
    double          deposit;                    ///< 今日入金
    double          withdraw;                   ///< 今日出金
    double          margin;                     ///< 占用保证金，期权空头保证金目前支持昨结算价或报单价计算。
    double          frozenMargin;               ///< 冻结保证金
    double          premium;                    ///< 权利金
    double          frozenPremium;              ///< 冻结权利金
    double          commission;                 ///< 手续费
```

```
    double          frozenCommission;               ///< 冻结手续费
    double          orderCommission;                ///< 申报费
    double          balance;                        ///< 动态权益：静态权益+持仓亏损+平仓盈亏-手续费-申报费+权利金
                                                    ///< 其中：
                                                    ///< - 持仓盈亏出现亏损时为负数，计入动态权益与可用资金；
                                                    ///< - 持仓盈亏盈利时为正数，不计入动态权益与可用。
    double          available;                      ///< 可用资金：动态权益-占用保证金-冻结保证金-交割保证金-冻结权利金-冻结手续
费
                                                    ///< 其中：占用保证金包含期权与期货
    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。
    mutable XTFUserData userData;                   ///< 保留给用户使用的数据对象

    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 XTFCashInOut* getCashInOut(int pos) const;///< 按位置索引查询出入金对象，从0开始计算
    int             getCombPositionCount() const;   ///< 查询所有组合持仓的数量
    const XTFCombPosition* getCombPosition(int pos) const; ///< 按位置索引查询组合持仓对象

    bool isSupportCancelOrderByLocalID() const;     ///< 是否支持本地报单编号撤单。
                                                    ///< 如果不支持，那么以本地报单编号撤单时，将返回不支持此功能的错误码
                                                    ///< 如果支持本地单号撤单，对本地单号有上限要求，具体上限值根据
getMaxAllowedLocalOrderID() 查询获得
    XTFLocalOrderID getMaxAllowedLocalOrderID() const; ///< 本地单号撤单时允许的最大本地单号，0表示不支持本地单号撤单
};
```

XTFExchange结构体

说明：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，INE，DCE，GFEX，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
    mutable XTFUserData userData;                   ///< 保留给用户使用的数据对象

    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
};
```

XTFProductGroup结构体

说明：品种组信息对象类

```
class XTFProductGroup {
public:
    XTFProductGroupID   productGroupID;                 ///< 品种组代码
    mutable XTFUserData userData;                       ///< 保留给用户使用的数据对象
    int                 getProductCount() const;        ///< 查询品种组包含的品种数量
    const XTFProduct*   getProduct(int pos) const;      ///< 按位置索引查询品种对象
    const XTFExchange*  getExchange() const;            ///< 查询品种组所属的交易所对象
};
```

XTFProduct结构体

说明：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
    mutable XTFUserData userData;                       ///< 保留给用户使用的数据对象
    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

说明：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
    mutable XTFUserData     userData;                       ///< 保留给用户使用的数据对象

    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
    int                     getOrderCommissionRatioAmountLevelCount() const; ///< 查询信息量的档位数
    int                     getOrderCommissionRatioOTRLevelCount() const; ///< 查询OTR的档位数
    const XTFOrderCommissionRatio* getOrderCommissionRatio(int amountLevelPos, int otrLevelPos) const; ///< 查询指定信息量档位、OTR档位对应的申报费率
    double                  getOrderCommission() const;     ///< 查询该合约的累计申报费
};
```

XTFPrePosition结构体

说明：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
    mutable XTFUserData     userData;               ///< 保留给用户使用的数据对象
    const XTFInstrument*    getInstrument() const;  ///< 查询昨持仓所属合约XTFInstrument的指针
};
```

XTFTradeDetail结构体

说明：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

说明：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
};
```

XTFPosition结构体

说明：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 - 已平历史持仓 + 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，表示持仓的占用保证金。不考虑单向大边或组合持仓的保证金减免逻
辑。
    double                  paidMargin;             ///< 实付保证金，表示实际支付的占用保证金。在计算单向大边或启用组合时，实付保
证金可能会小于占用保证金。
    double                  frozenMargin;           ///< 冻结保证金
    double                  frozenCommission;       ///< 冻结手续费
    double                  totalOpenPrice;         ///< 总开仓金额：昨仓使用昨结算价，今仓使用成交价计算
    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）
    mutable XTFUserData     userData;               ///< 保留给用户使用的数据对象

    double          getOpenPrice() const;           ///< 持仓均价
    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（历史持仓总量 - 已平历史持仓量）
    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
};
```

XTFInputOrder结构体

说明：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。
                                                    ///< 注意不能与柜台保留的几个特殊ID冲突：
                                                    ///< 1. 非本柜台报单固定为0x88888888；
                                                    ///< 2. 柜台清流启动后的历史报单固定为0xd8888888；
                                                    ///< 3. 柜台平仓报单固定为0xe8888888；
                                                    ///< 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。当报单类型为FAK时，
                                                    ///< 如果 minVolume > 1，那么API默认使用最小成交数量进行报单；
                                                    ///< 如果 minVolume ≤ 0，那么API默认使用任意成交数量进行报单；
    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）
    XTFUserRef              userRef;                ///< 用户自定义数据，发送到柜台后，柜台不作处理，保留原值返回给用户

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

XTFExecOrder结构体

说明：行权对冲报单请求对象类

```
class XTFExecOrder {
public:
    XTFLocalOrderID         localOrderID;           ///< 本地报单编号，需要由用户保证唯一性，用于本地存储索引。
                                                    ///< 不能和XTFInputOrder本地报单编号发送冲突，应与XTFInputOrder的本地报单
编号统一处理。
    XTFOrderFlag            orderFlag;              ///< 报单标志，用于区分是行权还是自对冲
    XTFOrderType            orderType;              ///< 报单类型：
                                                    ///< - 行权：XTF_ODT_SelfClose | XTF_ODT_NotSelfClose
                                                    ///< - 对冲：XTF_ODT_SelfCloseOptions | XTF_ODT_SelfCloseFutures
    XTFOffsetFlag           offsetFlag;             ///< 开平仓标志：XTF_OF_Close | XTF_OF_CloseToday |
XTF_OF_CloseYesterday
    XTFDirection            direction;              ///< 行权和对冲方向：
                                                    ///< - XTF_D_Buy：请求行权、请求对冲；
                                                    ///< - XTF_D_Sell：放弃行权、请求不对冲；
    XTFHedgeFlag            hedgeFlag;              ///< 投机套保标志
    double                  minProfit;              ///< 行权最小利润
    uint16_t                volume;                 ///< 行权数量
    XTFChannelSelectionType channelSelectionType;   ///< 席位编号选择类型
    uint8_t                 channelID;              ///< 席位编号
    XTFUserRef              userRef;                ///< 用户自定义数据，发送到柜台后，柜台不作处理，保留原值返回给用户
    XTF_CONST XTFInstrument *instrument;            ///< 报单合约对象
};
```

XTFOrder结构体

说明：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
                                                    ///< 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
                                                    ///< 保留的特殊本地单号：
                                                    ///< 1. 非本柜台报单固定为0x88888888；
                                                    ///< 2. 柜台清流启动后的历史报单固定为0xd8888888；
                                                    ///< 3. 柜台平仓报单固定为0xe8888888；
    XTFExchangeOrderID      exchangeOrderID;        ///< 交易所报单编号
                                                    ///< 报单状态为 XTF_OS_Received 及之前状态时，此字段无效；
                                                    ///< 报单状态为 XTF_OS_Accepted 及之后状态时，此字段有效；
    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。
    XTFUserRef              userRef;                ///< 用户自定义数据，发送到柜台后，柜台不作处理，保留原值返回给用户
    mutable XTFUserData     userData;               ///< 保留给用户使用的数据对象
    int                     getTradeCount() const;  ///< 查询报单对应的成交明细数量
    const XTFTrade*         getTrade(int pos) const;///< 按位置索引查询报单对应的成交明细

    XTFHedgeFlag            getHedgeFlag() const;
    double                  getOptionsExecMinProfit() const;
    XTFOptionsExecResult    getOptionsExecResult() const;
};
```

XTFTrade结构体

说明：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;                 ///< 该字段已废弃
    double                  commission;             ///< 本次回报已成交手数产生的手续费
    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
    XTFUserRef              userRef;                ///< 用户自定义数据，与关联的报单对象userRef相同
    mutable XTFUserData     userData;               ///< 保留给用户使用的数据对象

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

XTFMarginRatio结构体

说明：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。
    mutable XTFUserData     userData;                   ///< 保留给用户使用的数据对象
    const XTFInstrument*    getInstrument() const;      ///< 所属XTFInstrument的指针
};
```

XTFCommissionRatio结构体

说明：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
    mutable XTFUserData     userData;                   ///< 保留给用户使用的数据对象
    const XTFInstrument*    getInstrument() const;      ///< 所属XTFInstrument的指针
};
```

XTFOrderCommissionRatio结构体

说明：申报费率对象类

```
class XTFOrderCommissionRatio {
public:
    uint32_t                amountBegin;                ///< 信息量范围起始值
    uint32_t                amountEnd;                  ///< 信息量范围结束值
    double                  otrBegin;                   ///< OTR范围起始值
    double                  otrEnd;                     ///< OTR范围结束值
    double                  rate;                       ///< 申报费率

    ///< 用于方便计算申报费的固定值：针对相同的OTR范围，可以通过此固定值加快计算，计算方式如下：
    ///<           | OTR=[0,5)                                          | OTR=[5,10) | OTR=[10, MAX)
    ///< [  0,100) | rate1=0.1, fixValue1=0.0                          |            |
    ///< [100,200) | rate2=0.2, fixValue2=fixValue1 + 0.1*(100-0)      |            |
    ///< [200,MAX) | rate3=0.3, fixValue3=fixValue2 + 0.2*(200-100)    |            |
    double                  fixValue;
};
```

XTFMarketData结构体

说明：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）
    mutable XTFUserData userData;               ///< 保留给用户使用的数据对象

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

XTFCashInOut结构体

说明：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
    mutable XTFUserData     userData;               ///< 保留给用户使用的数据对象
};
```

XTFEvent结构体

说明：通知事件类

```
class XTFEvent {
public:
    char                tradingDay[9];                  ///< 通知事件日期(交易日)
    char                eventTime[9];                   ///< 通知事件时间
    XTFEventType        eventType;                      ///< 通知事件类型
    XTFEventID          eventID;                        ///< 通知事件ID
    int                 eventLen;                       ///< 通知事件数据长度
    char                eventData[256];                 ///< 通知事件的数据
    char                reserve[2];                     ///< 预留字段
};

/// 投资者风控事件通知结构体
class XTFPrivateEventInvestorPrc {
public:
    XTFPrcID            prcID;                          ///< 风控的具体ID类型
    char                investorID[13];                 ///< 投资者名称
    int                 prcValue;                       ///< 投资者风控值
    char                reserve[11];                    ///< 预留字段
};

/// 投资者合约风控事件通知结构体
class XTFPrivateEventInstrumentPrc {
public:
    XTFPrcID            prcID;                          ///< 风控的具体ID类型
    char                investorID[13];                 ///< 投资者名称
    char                instrumentID[31];               ///< 合约名称
    int                 prcValue;                       ///< 合约风控值
    char                reserve[12];                    ///< 预留字段
};
```

XTFOrderFilter结构体

说明：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，支持同时查询四种状态
    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

    XTFOrderFilter() {
        startTime[0] = '\0';
        endTime[0] = '\0';
        direction = XTF_D_Invalid;
        offsetFlag = XTF_OF_Invalid;
        orderFlag = XTF_ODF_Invalid;
        orderType = XTF_ODT_Invalid;
        orderStatus[0] = XTF_OS_Invalid;
        orderStatus[1] = XTF_OS_Invalid;
        orderStatus[2] = XTF_OS_Invalid;
        orderStatus[3] = XTF_OS_Invalid;
        instrument = nullptr;
        product = nullptr;
        exchange = nullptr;
    }
};
```

XTFTradeFilter结构体

说明：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

    XTFTradeFilter() {
        startTime[0] = '\0';
        endTime[0] = '\0';
        direction = XTF_D_Invalid;
        offsetFlag = XTF_OF_Invalid;
        instrument = nullptr;
        product = nullptr;
        exchange = nullptr;
    }
};
```

XTFCombInstrument结构体

说明：组合合约类

```
class XTFCombInstrument {
public:
    XTFInstrumentID         combInstrumentID;               ///< 组合合约编号
    uint32_t                combInstrumentIndex;            ///< 组合合约序号
    XTFCombType             combType;                       ///< 组合类型
    XTFCombDirection        combDirection;                  ///< 组合合约方向
    mutable XTFUserData     userData;                       ///< 保留给用户使用的数据对象

    const XTFInstrument*    getLeftInstrument() const;      ///< 左腿合约对象指针
    const XTFInstrument*    getRightInstrument() const;     ///< 右腿合约对象指针
    const XTFCombPosition*  getCombPosition(XTFCombHedgeFlag combHedgeFlag = XTF_COMB_HF_SpecSpec) const; ///< 查询组合持仓对象
    long                    getCombPriority(XTFCombHedgeFlag combHedgeFlag = XTF_COMB_HF_SpecSpec) const; ///< 查询组合优先级
    const XTFExchange*      getExchange() const;            ///< 合约所属交易所
};
```

XTFCombPositionDetail结构体

说明：组合持仓明细类

```
class XTFCombPositionDetail {
public:
    int                     volume;                     ///< 组合持仓数量
    double                  margin;                     ///< 占用保证金总额（左腿保证金+右腿保证金）
    double                  paidMargin;                 ///< 实付保证金总额（优惠后的实付保证金，左腿保证金或者右腿保证金）
    XTFTradeID              leftTradeID;                ///< 左腿合约的成交编号
    XTFTradeID              rightTradeID;               ///< 右腿腿合约的成交编号
    mutable XTFUserData     userData;                   ///< 保留给用户使用的数据对象
};
```

XTFCombPosition结构体

说明：组合持仓类

```
class XTFCombPosition {
public:
    int                     position;                   ///< 组合仓位总数量
    double                  margin;                     ///< 占用保证金总额，所有组合明细的左腿保证金+右腿保证金之和
    double                  paidMargin;                 ///< 实付保证金总额，所有组合明细的实付保证金之和
    XTFCombHedgeFlag        combHedgeFlag;              ///< 组合投机套保标志
    mutable XTFUserData     userData;                   ///< 保留给用户使用的数据对象
    const XTFCombInstrument* getCombInstrument() const;     ///< 关联的组合合约
    int                     getCombPositionDetailCount() const; ///< 组合明细数量
    const XTFCombPositionDetail& getCombPositionDetail(int pos) const; ///< 组合明细
};
```

XTFPositionCombEvent结构体

说明：组合通知事件类

```
class XTFPositionCombEvent {
public:
    XTFSysOrderID           sysOrderID;                 ///< 柜台流水号
    XTFLocalOrderID         localOrderID;               ///< 用户填写的本地编号
    XTFExchangeOrderID      exchangeOrderID;            ///< 交易所报单编号
    XTFCombHedgeFlag        combHedgeFlag;              ///< 组合投保标志
    XTFCombAction           combAction;                 ///< 组合行为类型(组合/解锁)
    uint32_t                combVolume;                 ///< 数量
    XTFTime                 combTime;                   ///< 组合时间
    const XTFCombInstrument*combInstrument;             ///< 组合合约
};
```

XTFInputQuoteDemand结构体

说明：询价请求对象类

```
class XTFInputQuoteDemand {
public:
    XTFLocalOrderID         localOrderID;               ///< 本地报单编号，无特殊要求，用户自行定义即可
    XTFChannelSelectionType channelSelectionType;       ///< 席位编号选择类型
    uint8_t                 channelID;                  ///< 席位编号
    XTFUserRef              userRef;                    ///< 用户自定义数据，发送到柜台后，柜台不作处理，保留原值返回给用户
    XTF_CONST XTFInstrument *instrument;                ///< 合约对象
};
```

XTFReportFilter结构体

说明：回报过滤器对象类（白名单）

```
class XTFReportFilter {
public:
    XTFProductFilter        productFilter;          ///< FUTURES ||(&&) OPTIONS || ALL
    char                    instrumentFilter[320]; ///< 过滤表达式字符串。通配符*仅支持后缀模式，多个表达式使用逗号分割。例如：
au*,au23*
};
```

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[16]	品种数据类型
XTFProductGroupID	char[16]	品种组数据类型
XTFInstrumentID	char[32]	合约ID数据类型
XTFAccountID	char[20]	账号ID数据类型
XTFDate	char[9]	日期数据类型
XTFTime	char[9]	时间数据类型
XTFExchangeOrderID	int64_t	交易所报单编号数据类型
XTFSysOrderID	int32_t	柜台流水号数据类型
XTFLocalOrderID	int32_t	本地报单编号数据类型（用户定义）
XTFLocalActionID	int32_t	本地撤单编号数据类型（用户定义）
XTFTradeID	int64_t	成交编号数据类型

枚举类型

报单编号类型（XTFOrderIDType）

类型：uint8_t

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

品种类型（XTFProductClass）

类型：uint8_t

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（XTFOptionsType）

类型：uint8_t

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（XTFDirection）

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（XTFPositionDirection）

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

投机套保标志（XTFHedgeFlag）

类型：uint8_t，当前仅支持投机

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（XTFCashDirection）

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（XTFOffsetFlag）

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（暂不支持）
XTF_OF_CloseToday	平今	3
XTF_OF_CloseYesterday	平昨	4
XTF_OF_Invalid	无效	9

报单标志（XTFOrderFlag）

类型：uint8_t

枚举类型	说明	枚举值
XTF_ODF_Normal	普通报单	0
XTF_ODF_CombinePosition	组合持仓	1
XTF_ODF_OptionsExecute	行权报单	2
XTF_ODF_OptionsSelfClose	对冲报单	3
XTF_ODF_Warm	预热报单	254
XTF_ODF_Invalid	无效值	255

报单类型（XTFOrderType）

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
XTF_ODT_SelfClose	期权对冲（仅行权有效）	21
XTF_ODT_NotSelfClose	期权不对冲（仅行权有效）	22
XTF_ODT_SelfCloseOptions	期权对冲（仅对冲有效）	31
XTF_ODT_SelfCloseFutures	履约对冲（期权期货对冲卖方履约后的期货仓位，仅对冲有效）	32

报单状态（XTFOrderStatus）

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_Received	报单发送到柜台，柜台已接收此报单，并通过柜台风控，下一时刻将发往交易所。 此状态的报单，柜台流水号是有效的，但交易所单号无效。可以根据此流水号进行撤单操作。	7
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_Queuing | XTF_OS_Canceled | XTF_OS_PartTraded | XTF_OS_AllTraded | XTF_OS_Rejected	1
XTF_OS_Queuing	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

席位选择（XTFChannelSelectionType）

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
XTF_CS_Unknown	历史报单回报，无法确认报单通道选择类型	9

保证金计算价格类型（XTFMarginPriceType）

类型：uint8_t

枚举类型	说明	枚举值
XTF_MPT_MaxLastSettlementOrLastPrice	最新价和昨结算价之间的较大值（期权空头暂不支持）	0
XTF_MPT_LastSettlementPrice	昨结算价计算保证金（期权空头默认）	1
XTF_MPT_OrderPrice	报单价计算保证金（期权空头有效）	2
XTF_MPT_OpenPrice	开仓价计算保证金（期货默认）	3
XTF_MPT_LastPrice	最新价（暂未使用）	4
XTF_MPT_AverageOpenPrice	开仓均价（暂未使用）	5
XTF_MPT_AveragePrice	市场平均成交价（暂未使用）	6
XTF_MPT_LimitPrice	涨跌停价（暂未使用）	7

合约状态（XTFInstrumentStatus）

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（XTFOrderActionType）

类型：uint8_t

枚举类型	说明	枚举值
XTF_OA_Invalid	无效的报单操作	255
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

事件通知编号（XTFEventID）

类型：int

枚举类型	说明	枚举值
XTF_EVT_AccountCashInOut	账户出入金发生变化通知	0x1001
XTF_EVT_ExchangeChannelConnected	交易所交易通道已连接通知	0x1011
XTF_EVT_ExchangeChannelDisconnected	交易所交易通道已断开通知	0x1012
XTF_EVT_InstrumentStatusChanged	合约状态发生变化通知	0x1021

组合类型（XTFCombType）

类型：uint8_t

枚举类型	说明	枚举值
XTF_COMB_SPL	期货对锁	0
XTF_COMB_OPL	期权对锁	1
XTF_COMB_SP	跨期套利	2
XTF_COMB_SPC	跨品种套利	3
XTF_COMB_BLS	买入垂直价差	4
XTF_COMB_BES	卖出垂直价差	5
XTF_COMB_CAS	期权日历价差	6
XTF_COMB_STD	期权跨式	7
XTF_COMB_STG	期权宽跨式	8
XTF_COMB_BFO	买入期货期权	9
XTF_COMB_SFO	卖出期货期权	10

组合方向类型（XTFCombDirection）

类型：uint8_t

枚举类型	说明	枚举值
XTF_COMB_D_LongShort	多-空	0
XTF_COMB_D_ShortLong	空-多	1

组合投机套保标志（XTFCombHedgeFlag）

类型：uint8_t，当前仅支持投机-投机

枚举类型	说明	枚举值
XTF_COMB_HF_SpecSpec	投机-投机	1
XTF_COMB_HF_SpecHedge	投机-保值	2
XTF_COMB_HF_HedgeHedge	保值-保值	3
XTF_COMB_HF_HedgeSpec	保值-投机	4
XTF_COMB_HF_MaxCount	保留内部使用	5

组合动作类型（XTFCombAction）

类型：uint8_t

枚举类型	说明	枚举值
XTF_COMB_AT_Combine	组合	1
XTF_COMB_AT_Uncombine	拆组合	2

期权行权/对冲执行结果（XTFOptionsExecResult）

类型：uint8_t

枚举类型	说明	枚举值
XTF_OER_Error	执行失败	'e'
XTF_OER_NoExec	没有执行	'n'
XTF_OER_Canceled	已经取消	'c'
XTF_OER_Success	执行成功	'0'
XTF_OER_NoPosition	期权持仓不够	'1'
XTF_OER_NoDeposit	资金不够	'2'
XTF_OER_NoParticipant	会员不存在	'3'
XTF_OER_NoClient	客户不存在	'4'
XTF_OER_NoInstrument	合约不存在	'6'
XTF_OER_NoRight	没有执行权限	'7'
XTF_OER_InvalidVolume	不合理的数量	'8'
XTF_OER_NoEnoughHistoryTrade	没有足够的历史成交	'9'

回报品种过滤选项（XTFProductFilter）

类型：uint16_t

枚举类型	说明	枚举值
XTF_PCF_Futures	表示仅处理期货类型的数据	0x0001
XTF_PCF_Options	表示仅处理期权类型的数据	0x0002
XTF_PCF_All	表示处理所有品种类型的数据	0xFFFF

常用代码示例

查询合约的申报费率

```
// 查询指定合约所有档位配置的申报费率
void queryInstrumentOrderCommissionRatio(const XTFInstrument &instrument) {
    auto getAmountString = [](uint32_t amount) -> std::string {
        if (amount < UINT32_MAX) return std::to_string(amount);
        return "MAX";
    };
    auto getOTRString = [](double otr) -> std::string {
        if (otr < UINT32_MAX) return std::to_string(otr);
        return "MAX";
    };
    auto amountLevels = instrument.getOrderCommissionRatioAmountLevelCount(); // 查询信息量档位数量
    auto otrLevels = instrument.getOrderCommissionRatioOTRLevelCount(); // 查询OTR档位数量
    for (auto j = 0; j < otrLevels; ++j) {
        for (auto i = 0; i < amountLevels; ++i) {
            auto rate = instrument.getOrderCommissionRatio(i, j);
```

```
            if (!rate) continue;
            printf("instrument-id=%s, "
                   "amount=[%s, %s), ort=[%s, %s), rate=%.4f, fixValue=%.4f\n",
                   instrument.instrumentID,
                   getAmountString(rate->amountBegin).c_str(),
                   getAmountString(rate->amountEnd).c_str(),
                   getOTRString(rate->otrBegin).c_str(),
                   getOTRString(rate->otrEnd).c_str(),
                   rate->rate, rate->fixValue);
        }
    }
}
```

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_Queuing:
```

```
            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 XTFTrade *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;
}
```