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。

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；
2023/07/31	V4.1.931	增加通过单号和交易所进行撤单的接口； 增加极简模式配置，此模式下API不再保存任何的交易（订单和成交）信息，由用户自行管理； 优化报撤单接口的性能，支持Solarflare和Exablaze网卡的加速处理；

日期	API版本	API主要变更说明
2023/08/29	V4.1.990	增加全局管理接口的说明； 增加获取高精度纳秒级时间戳的接口 `getXTFNanoTimestamp()` ； 增加Onload版本自动检测功能：如果版本小于7.x或大于7.x，提示相应的告警日志； 增加低延迟网卡功能开关配置：配置项为 `XNIC_ENABLED` ，true表示启用，false表示禁用；
2023/09/14	V4.1.1019	增加报单分组编号功能；
2023/10/17	V4.1.1053	增加前置席位优选类型FixedThenAuto，用户如果指定的前置席位不合法，则使用自动优选模式自动选择合法的前置席位；
2023/11/27	V4.1.1068	makeXTFApi接口增加用户自定义信号处理函数的说明；
2023/12/15	V4.1.1072	增加配置选项 `COMBINATION_ENABLED` 和 `COMBINATION_TYPES` ，用于设置仓位自动组合的类型； 增加配置选项 `HISTORICAL_FINISHED_ORDER_FILTER_ENABLED` ，用于滤除做市柜台的历史完结订单回报流水；
2024/03/15	V4.1.1226	增加指定席位优先级接口；
2024/04/15	V4.1.1260	增加报单类型的枚举值,支持GIS指令； XTFAccount 结构体新增 getMarginType() 接口，支持交易客户查>询柜台的保证金算法类型；
2024/05/22	V4.1.1314	调整行权冻结保证金计算公式，与 `CTP` 保持一致； 申报费查询 `XTFInstrument.getOrderCommissionRatio(int, int)` 接口增加下标溢出保护；
2024/06/19	V4.1.1350	低延迟网卡功能支持配置网卡类型和网卡名称； 支持指定前置撤单； `XTFInstrument` 支持信息量查询接口； 增加配置选项 `CHECK_VERSION_ENABLED` ，用于控制头文件和库文件的版本一致性检测。默认为启用检测；
2024/06/28	V4.1.1360	支持接入柜台的一档行情功能； 支持合约级别的申报费计算开关功能； `XTFMarketData` 结构体字段调整：删除averagePrice字段，增加openPrice字段，调整snapTime字段顺序；
2024/08/14	V4.1.1375	增加看穿式监管信息的说明；
2024/11/19	V4.1.1380	增加TCP方式报撤单功能； 增加保证金计算说明；
2025/03/12	V4.1.1385	广期所增加对商品指数合约的支持； 增加sendWarmOrder接口，用来发送指定合约的预热单； UDP 预热单行为调整，修改为默认不发送； 期权合约信息量计算调整：上期能源、郑商、广期：期权合约的询价计入信息量。其他均不计入； 支持指数类期权合约新增标的指数价格 `indexPrice` 字段； 修复资金计算相关的问题；
2025/05/16	V4.1.1390	大商行权/对冲单增加 `execLevel` 字段，支持按合约、系列、品种及投资者级别的对冲，支持期货持仓对冲； 大商广期卖出垂直价差组合保证金计算公式调整，解决平仓空头期权时不足以支付权利金导致的穿仓的风险； 大商广期支持按照日初组合明细数据计算组合持仓的场景； 郑商支持日初套利组合持仓的计算处理，暂不支持盘中套利开仓操作；
2025/06/26	V4.1.1395	大商所/广期所"卖出期权垂直价差组合"保证金根据柜台的配置来选择计算方法；
2025/10/15	V4.1.1400	空头期权平仓时不冻结权利金； 大商所启用RULE保证金算法时，支持月均价合约也参与保证金优惠计算； UDP报撤单支持中科驭数网卡
2026/02/12	V4.1.1415	询价结构体 `XTFInputQuoteDemand` , 新增柜台系统单号 `systemOrderNo` 和询价时间 `demandTime` 字段;
2026/05/21	V4.1.1420	合约结构体 `XTFInstrument` 增加 `strikePrice` 字段，表示期权合约的执行价，期货合约该字段无效； 大商所支持期权日历价差组合类型；
2026/09/04	V4.1.1425	郑商所增加盘中套利相关接口；
2026/09/09	V4.1.1426	新增交易席位代码查询接口： `XTFExchange::getSeatCode & getSeatCodeByID` ；

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( )

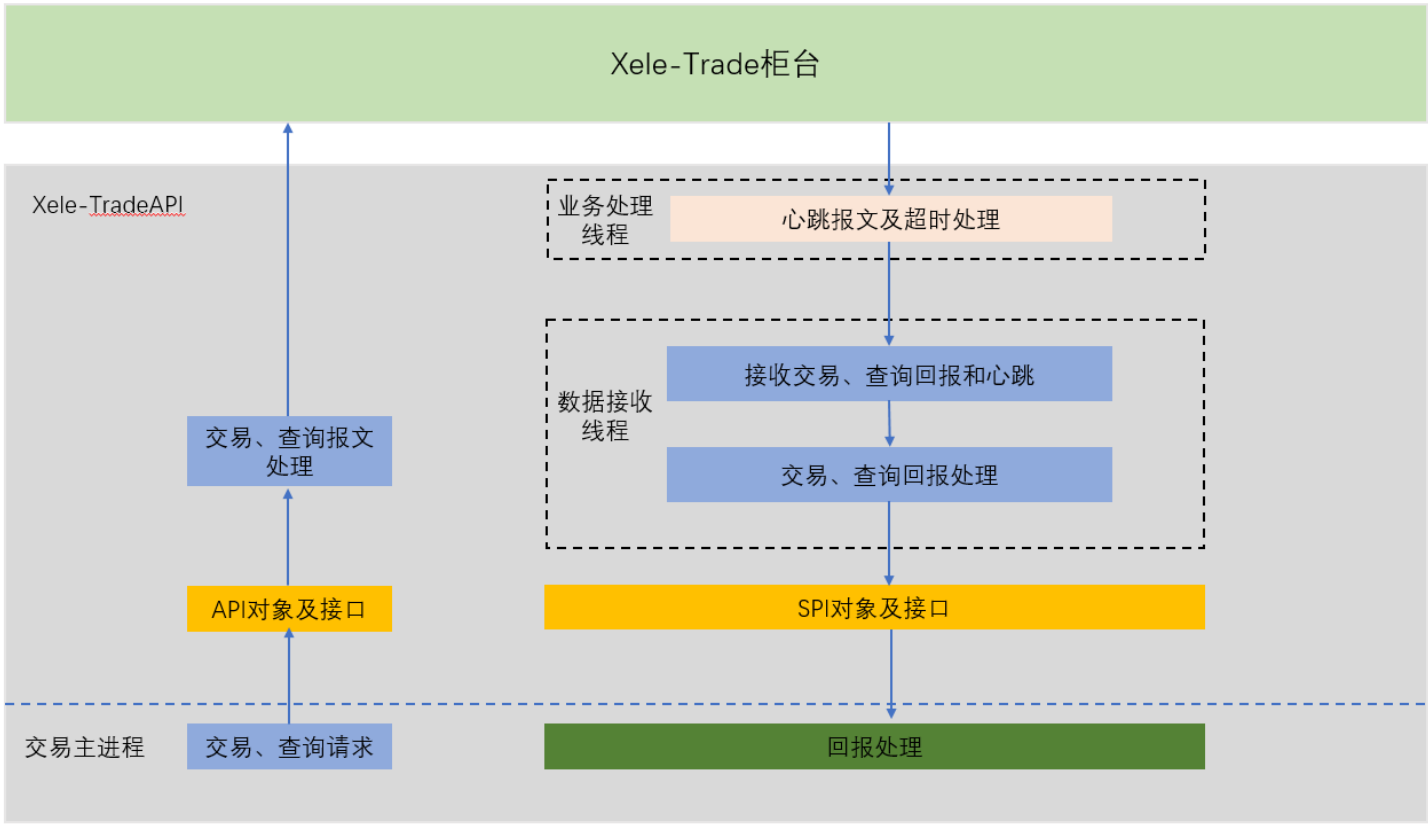
回应：

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）
# 当 TRADING_PROTOCOL=udp 时，配置选项生效。
TRADE_SERVER_IP=127.0.0.1
TRADE_SERVER_PORT=62000

# @since 4.1.1380
# 交易使用的传输层协议类型，支持UDP和TCP两种方式。
# 当使用TCP方式时，TRADE_SERVER_IP 和 TRADE_SERVER_PORT 不生效。
# 取值范围：[TCP, UDP]，默认是UDP方式。
# 此参数只能从配置文件设置，API的setConfig()接口不生效；
#TRADING_PROTOCOL=TCP

# @since 4.1.1360
# 指定是否启用柜台的TCP行情服务。
# true 表示启用行情服务，false 表示不启用。
# 当启用行情服务时，API登录成功且静态数据推送完成之后，会自动发起行情服务连接。
# 连接成功后，开始接收柜台定时推送的行情数据。默认情况下，不会回调通知行情数据。
# 用户调用了 subscribe 接口订阅成功后，API会通过 onBookUpdated 接口通知行情数据。
# 如果连接失败，API会记录错误信息到日志文件，并无主动通知的接口。
# 如果用户关注行情且长时间没有收到行情数据，可以检查API日志，以确认行情的错误原因。
# 默认不启用行情。
#MD_ENABLED=true

# @since 4.1.1360
# 指定柜台的TCP行情服务IP地址和端口
# 仅当 MD_ENABLED=true 时，配置生效。
#MD_SERVER_IP=127.0.0.1
#MD_SERVER_PORT=33000

# 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，可以加快交易回报数据的接收和处理，
# 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，取值范围：[true,false]，默认为true
# 不建议此配置项设置为false，会增加回报处理的时延。
#TCP_WORKER_BUSY_LOOP_ENABLED=true

# 报单是否启用线程安全模式，取值范围：[true,false]，默认为false，线程不安全模式。
# 启用线程安全模式，单个API实例可以支持多个线程同时报单；但是性能略有下降，会略微增加报单延时。
#ORDER_MT_ENABLED=false

# UDP预热处理是否启用，取值范围：[true,false]，默认为false：表示不启用
# 当TRADING_PROTOCOL=TCP时，此配置不生效。不支持TCP预热报单
#WARM_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

# 是否启用完结报单内存资源自动回收功能。true表示启用，false表示不启用，默认为不启用。
# 启用此功能需要非常谨慎。如果报单量非常大，且无成交的撤单占据大部分，那么可以启用此功能。其他场景下，建议不要启用。
# 启用后对于所有的错单、拒单和撤单（无成交），其分配的内存资源会被自动回收。
# 如果启用此功能，报单内存可能会被回收利用，用户需要根据报单回报消息和报单状态变化，来判断XTFOrder对象指针是否有效，
# 只有状态是已接收、正在队列中、部分成交和全部成交的报单，内存资源会持久保留、不会回收，这些报单的指针是有效的。
# 其他状态的报单指针是无效的，需要丢弃。
#RELEASE_FINISHED_ORDER_ENABLED=false

# 是否启用历史完结订单过滤功能（仅做市商柜台支持此功能）。true表示启用，false表示不启用，默认为不启用。
# 如果启用此过滤功能，客户端登录时，对所有的历史回报流水做以下过滤处理：
# 1、普通单：错单和撤单（无成交）的回报流水，不再推送给客户端；
# 2、报价单：状态为错单、两腿衍生单都是撤单且无成交的回报流水，不再推送给客户端；
# 3、衍生单：所有报价产生的衍生单，错单和撤单（无成交）的回报流水，不再推送给客户端；
# 如果启用此过滤功能，客户端可以收到的历史回报流水数据包括：
# 1、普通单：状态是已接收、正在队列中、部分成交、全部成交和部分成交的撤单的回报流水；
# 2、衍生单及其原生报价单：状态是已接收、正在队列中、部分成交、全部成交和部分成交的撤单的回报流水及其相关的报价单流水；
# 3、其他不在上述过滤条件中的回报流水；
# 此功能不影响登录后的回报流水，登录后的所有报撤单流水，全量返回给客户端。
# 极简模式此配置项暂不生效。
#HISTORICAL_FINISHED_ORDER_FILTER_ENABLED=false

# 设置API的运行模式，0-表示正常模式，1-表示极简模式，默认为正常模式
# 正常模式下，支持所有交易数据的本地存储和查询，可以支持持仓和资金计算；
# 极简模式下，不支持交易数据的本地存储和查询，不支持持仓和资金计算；
# 此参数只能从配置文件设置，API的setConfig()接口不生效；
#RUN_MODE=0

# 是否自动注册API内部的信号处理函数，true表示注册，false表示不注册。默认为true
# 此参数是全局参数，以第一次创建API实例对象时传入的为准；并且只能从配置文件设置，API的setConfig()接口不生效；
#REGISTER_SIGNAL_ENABLED=true

# 是否启用仓位的自动组合功能
# - true表示启用
# - false表示不启用
# 默认为false，不启用
# 此选项在4.1.1225版本及之后生效
#COMBINATION_ENABLED=true

# 指定自动组合的类型列表。默认表示自动组合所有的类型
# 组合类型取值范围请参考XTFDataTypes.h文件中XTFCombType的枚举定义
# 组合类型列表的每个元素，以逗号','分割。
# 例如：针对大商和广期所，可以按照如下配置自动组合的类型：
# - 自动组合期货对锁、期权对锁：COMBINATION_TYPES=0,1
# - 自动组合期货对锁、跨期套利和跨品种套利：COMBINATION_TYPES=0,2,3
# 注意：此配置选项需要与自动组合开关 COMBINE_ENABLED 选项配合使用
# 此选项在4.1.1225版本及之后生效
#COMBINATION_TYPES=0,1,2

# 是否启用低延迟网卡功能。
# 如果启用此配置项，那么API启动时会自动检测当前是否有可用的低延迟网卡，如果有则使用低延迟网卡快速通道进行报撤单。
# 如果禁用此配置项，那么API启动时不会检测是否有可用的低延迟网卡，默认走系统协议栈通道进行报撤单。
# 若启用该功能，建议使用onload-7.1.x版本或exanic2.7.x版本的驱动。
# 默认为启用。
#XNIC_ENABLED=false

# @since 4.1.1350
# 指定启用的低延迟网卡名称。
# 如果启用此配置项，则使用指定的网卡；反之，则自动选择一个可用的网卡。
# 建议使用默认配置，由API自动检测可用网卡。
#XNIC_NAME=eth0

# @since 4.1.1350
# 指定启用的低延迟网卡类型。
# 如果启用此配置项，则使用指定类型的驱动；反之，则自动选择匹配的网卡驱动。
# 建议使用默认配置，由API自动检测可用网卡类型。
# 支持类型范围：[sfc, exanic, swiftn]
#XNIC_TYPE=sfc

# @since 4.1.1350
# 是否启用头文件和库的版本一致性检测。
```

```
# 如果启用此配置项，那么API启动时会自动检测当前使用的头文件版本和库版本是否一致。如果不一致则启动失败，并返回相应的错误码。
# 如果禁用此配置项，那么API启动时不会检测头文件版本和库版本是否一致。
# 默认为启用。
#CHECK_VERSION_ENABLED=true
```

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。

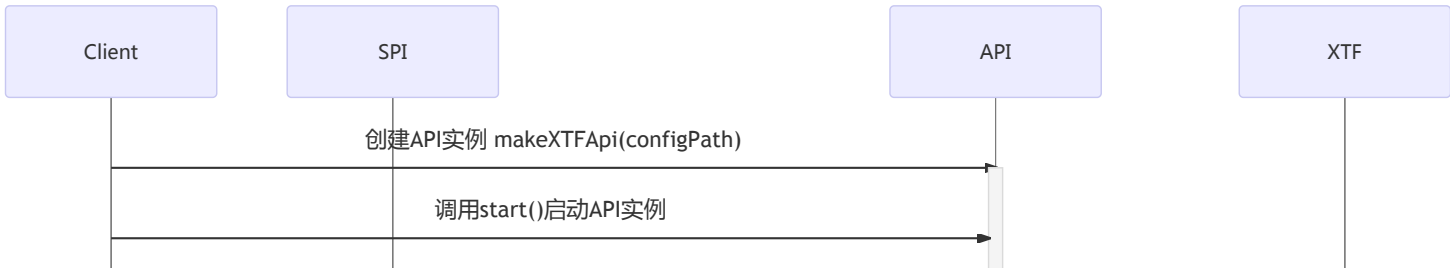
示例代码：功能调用及回调函数（具体代码详见example中“Example01.cpp”）

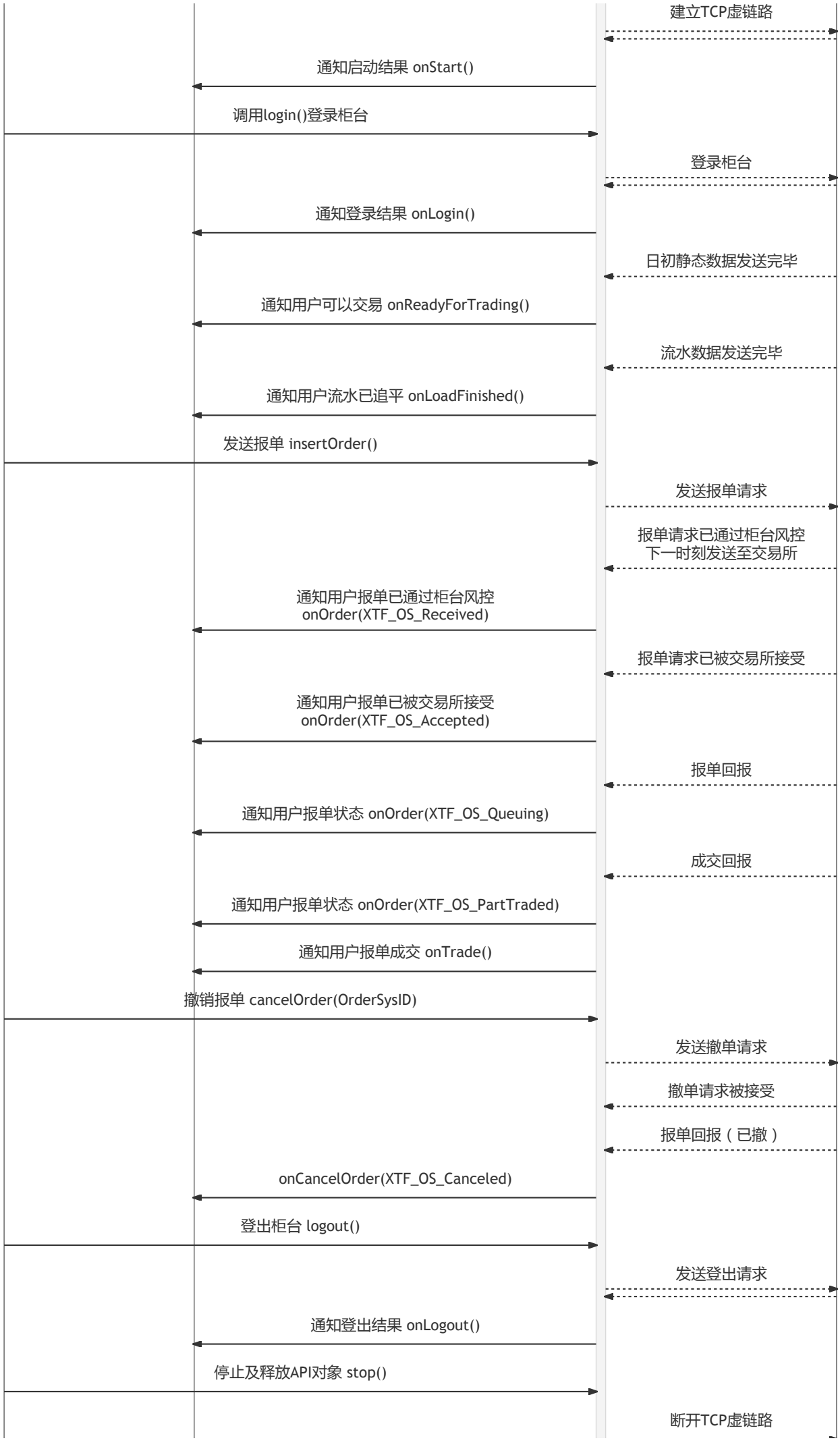
```
// 客户端程序和柜台建立连接后，调用报单操作接口。
trader.start();
... // 其余代码参见example/Example01.cpp
trader.insertOrder();
... // 其余代码参见example/Example01.cpp
trader.stop();
```

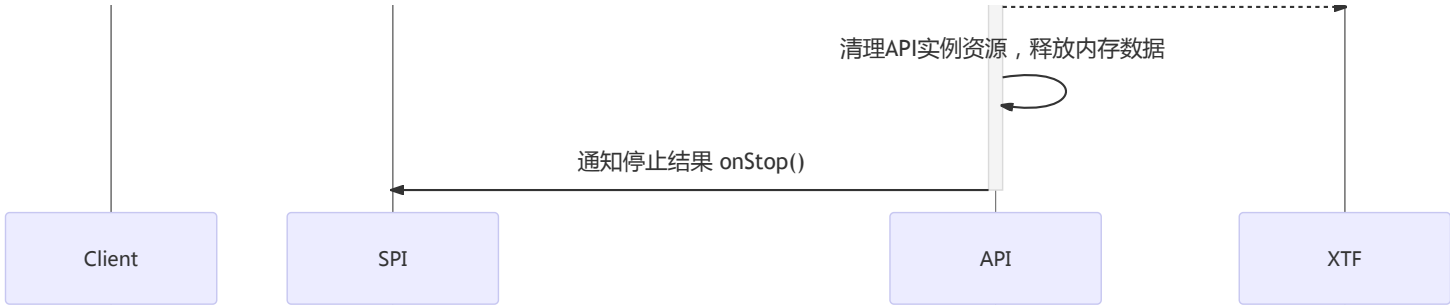
注意：每个API接口的调用都应该**检查其返回值**，非0值表示存在错误。

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：




建立TCP虚链路
通知启动结果 onStart()
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



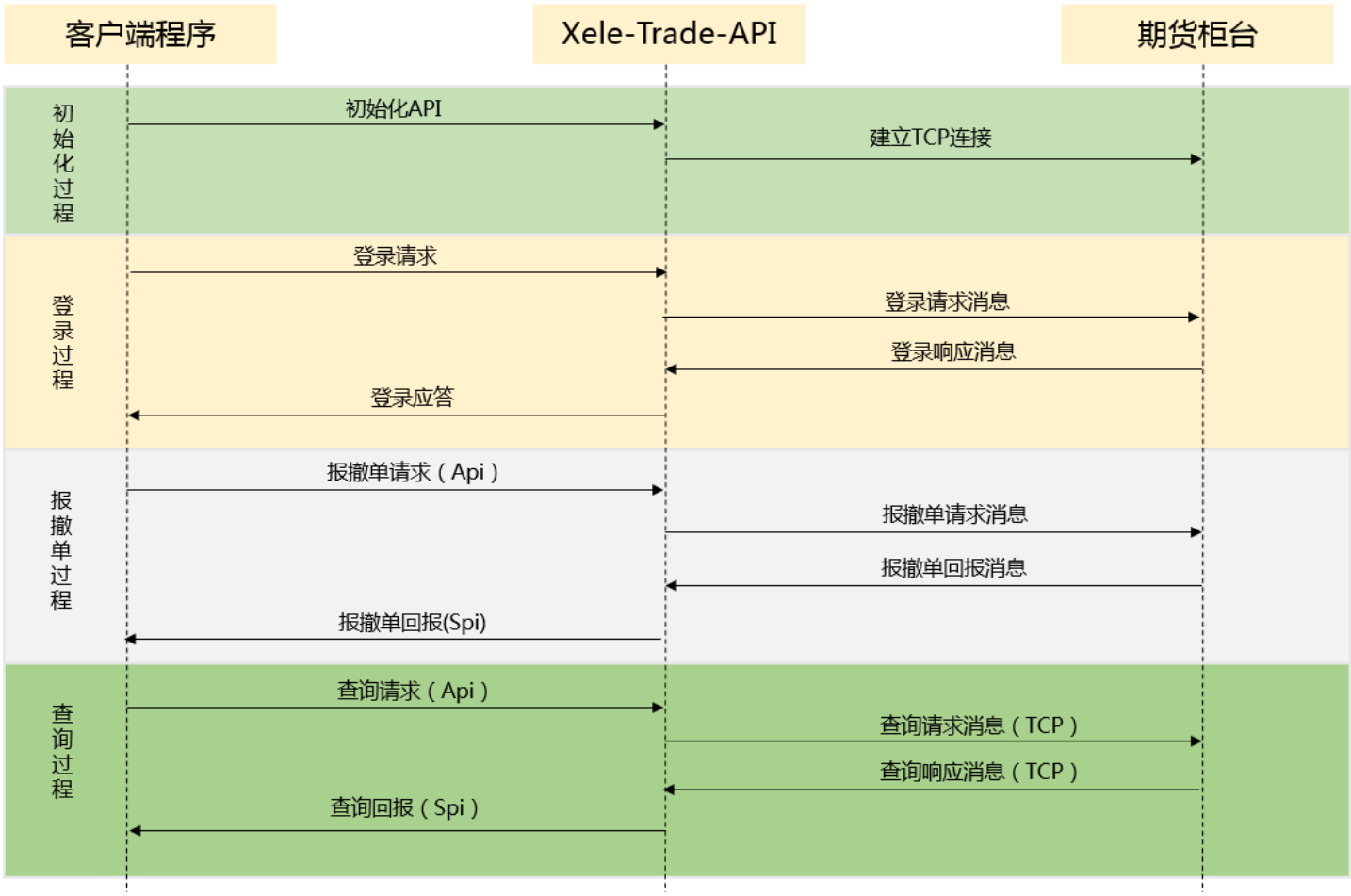
TraderAPI基本操作说明

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.");
}
```

看穿式监管

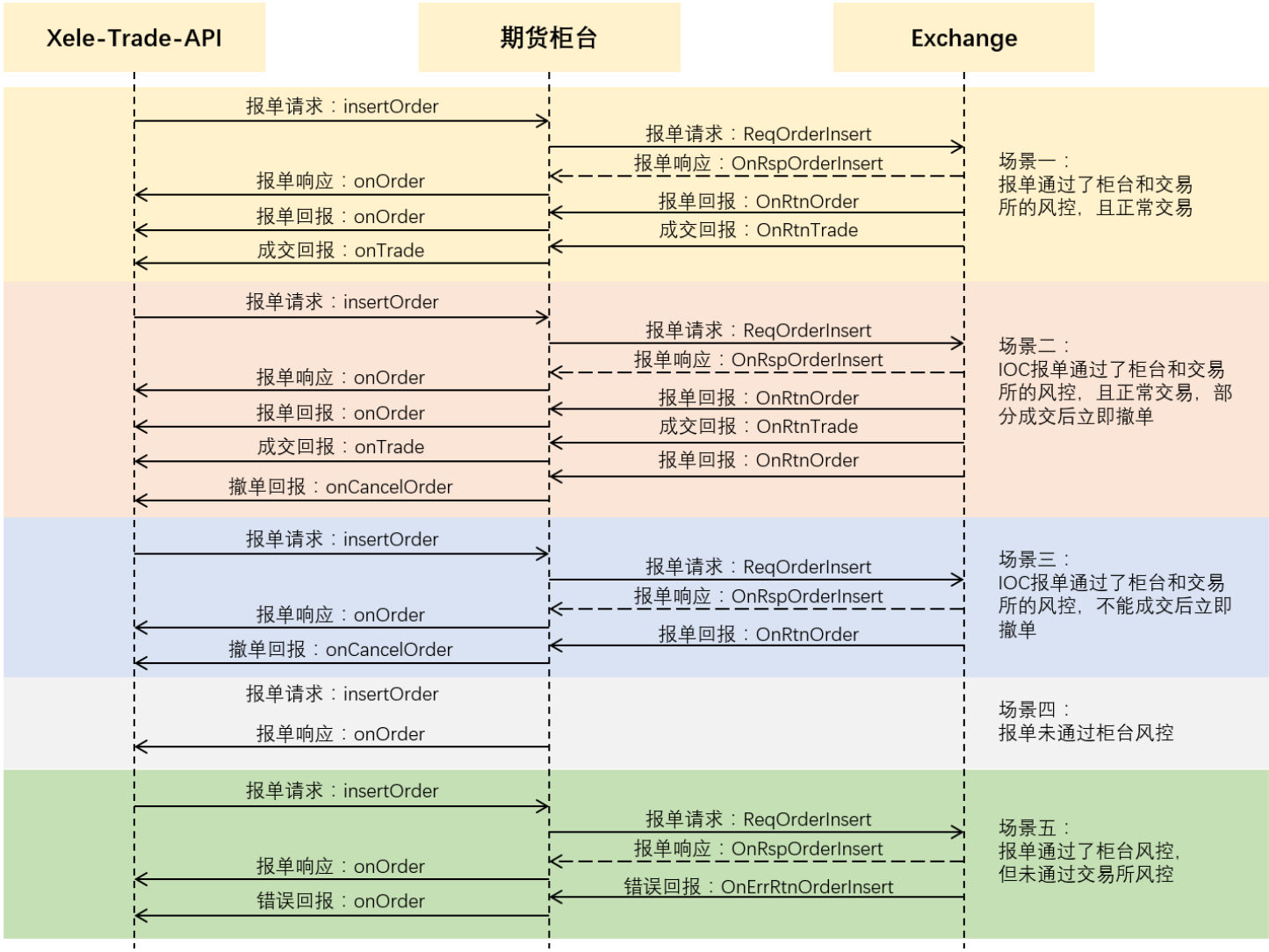
由于期货市场监控中心规定需要采集用户侧的看穿式监管信息，API在登录柜台之前，会自动采集客户端服务器的部分信息，加密后发送至柜台，由期货公司上传给期货市场监控中心。规定要求采集的看穿式监管信息包括：

1. 终端类型：Windows系统默认为1，Linux系统默认为2；
2. 系统时间：采集信息时当前的系统时间；
3. 网络IP地址；最多2个；
4. 网卡MAC地址：最多2个；
5. 主机名：Hostname；
6. 系统版本：操作系统的版本号；
7. 硬盘序列号；
8. CPU序列号；
9. BIOS序列号；

以下是两种操作系统的看穿式监管信息采集样例，用户登录柜台后，可以在日志文件中看到此信息的具体内容：

```
1@2024-08-09 09:37:03@192.168.2.200@192.168.2.200@fefcfeca72a1@fefcfeca72a1@win7-SOFT@6.1@VDI_DISK@FB060000FFFB8B1F@serial1
2@2024-08-09 10:01:31@172.17.0.1@172.18.0.1@0242CC45DE99@0242FE299D1C@cd215@3.10.@Z1W5OYDK@54060500FFFBEBBF@6CU741VH5F
```

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；

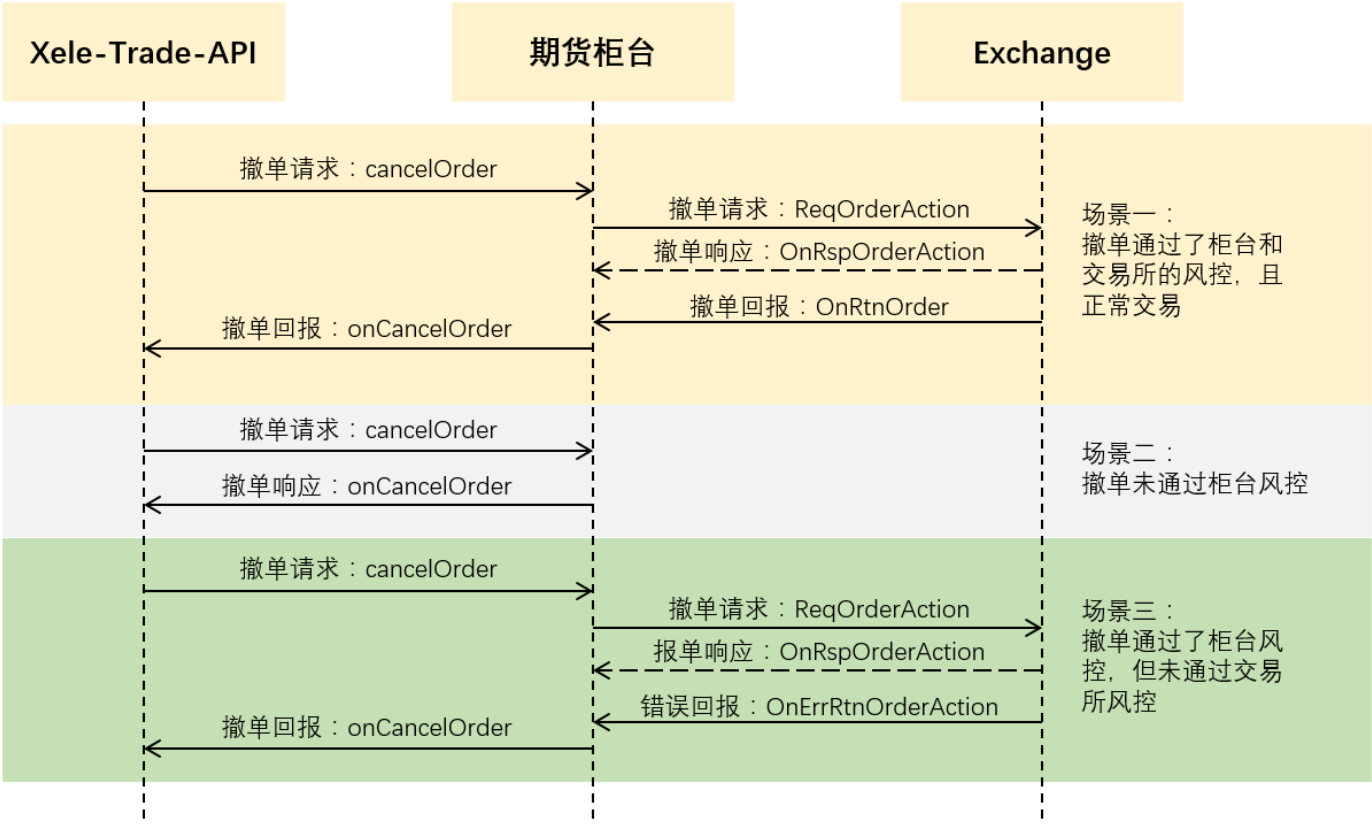
- 当报单成功进入交易所后，报单发生的任何成交，柜台都会发送成交回报，API通过onTrade接口通知用户；

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
套利接口	套利报撤机回报处理接口	XTFApi::insertArbiOrder XTFApi::cancelArbiOrder XTFSpi::onArbiOrder XTFSpi::onCancelArbiOrder
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

业务类型	业务	请求接口 / 响应接口
查询管理接口	查找套利单信息	XTFApi::findArbiOrders
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
参数配置管理接口	指定席位优先级接口	XTFApi::setChannelPriorities
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

全局管理接口

makeXTFApi方法

接口功能

创建API实例对象。

有两种方式创建API实例：

1. 配置文件方式，需要传入配置文件路径。如果传入的路径打开失败，那么创建API实例失败，返回空指针；
2. 参数设置方式，不需要传入配置文件路径。调用函数时配置文件路径参数传入空指针（nullptr）即可。创建成功后，需要通过API->setConfig()接口设置以下必选参数：
   "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
   详细内容请参考 `XTFApi::setConfig()` 接口注释说明。

接口原型

```
XTFApi* makeXTFApi(const char *configPath);
```

接口参数

- configPath：configPath 参数配置文件路径；

返回值

- 如果传入的路径是有效的，那么创建API实例成功，返回API实例指针；
- 如果传入的路径是非空的，但文件打开失败，那么创建API实例失败，返回空指针；
- 如果不传入路径，默认为nullptr，那么创建API实例成功，返回API实例指针。此时，需要通过setConfig接口设置实例的配置参数，否则无法使用；

关于信号处理函数的说明

创建API实例时，如果是首次调用该接口，API默认会注册内部的信号处理函数（Signal Handler）用于调试用途。如果在调用makeXTFApi()之前，APP已经注册了信号处理函数，那么API会覆盖已注册信号处理函数，导致信号无法传递给用户定义的信号处理函数。

可以按照以下方法处理此问题：

1. 如果API的版本低于4.1.1068版本，可以先调用makeXTFApi()接口，之后再注册用户自己的信号处理函数。调换注册的顺序，可以解决此问题；
2. 如果API的版本等于或高于4.1.1068版本，除了上面的方法之外，还可以通过在配置文件中增加配置项 `REGISTER_SIGNAL_ENABLED=false`，来关闭makeXTFApi()接口内部自动注册信号函数而导致覆盖的问题。默认API会自动注册内部信号处理函数，此选项需要显式指定为false，才能生效；

getXTFVersion方法

接口功能

查询API版本号。

版本字符串示例：`Lnx64 Xele-Trade TraderAPI v4.1.852-811d2dd`

接口原型

```
const char* getXTFVersion();
```

接口参数

无

返回值

API版本字符串。

getXTFErrorMessage方法

接口功能

根据错误码查询错误消息描述字符串。支持中英文两种语言的错误描述字符串。

接口原型

```
const char* getXTFErrorMessage(int errorCode, int language);
```

接口参数

- errorCode：待查询的错误码；
- language: 语言类型；0-中文，1-英文。

返回值

错误消息描述字符串。

getXTFNanoTimestamp方法

接口功能

按纳秒获取当前时间戳。

接口原型

```
unsigned long long getXTFNanoTimestamp();
```

接口参数

无

返回值

纳秒计数的时间戳。

setXTFLogEnabled方法

接口功能

是否启用日志功能。

启用日志功能，API会在应用程序当前执行目录下，生成 `xtf-api-log` 目录，存储日志文件。每次启动会按照时间戳生成新的日志文件。

禁用日志功能，API不再创建日志文件和记录日志。

日志文件可以记录API在运行过程中出现的简要过程和告警与错误，在发生异常时可以帮助错误发生原因的排查和定位。启用日志并不会给API的运行性能带来下降，目前API默认启用了日志功能。

接口原型

```
void setXTFLogEnabled(bool enabled);
```

接口参数

- enabled：启用或禁用日志，true-启用，false-禁用。

返回值

无

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

无

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

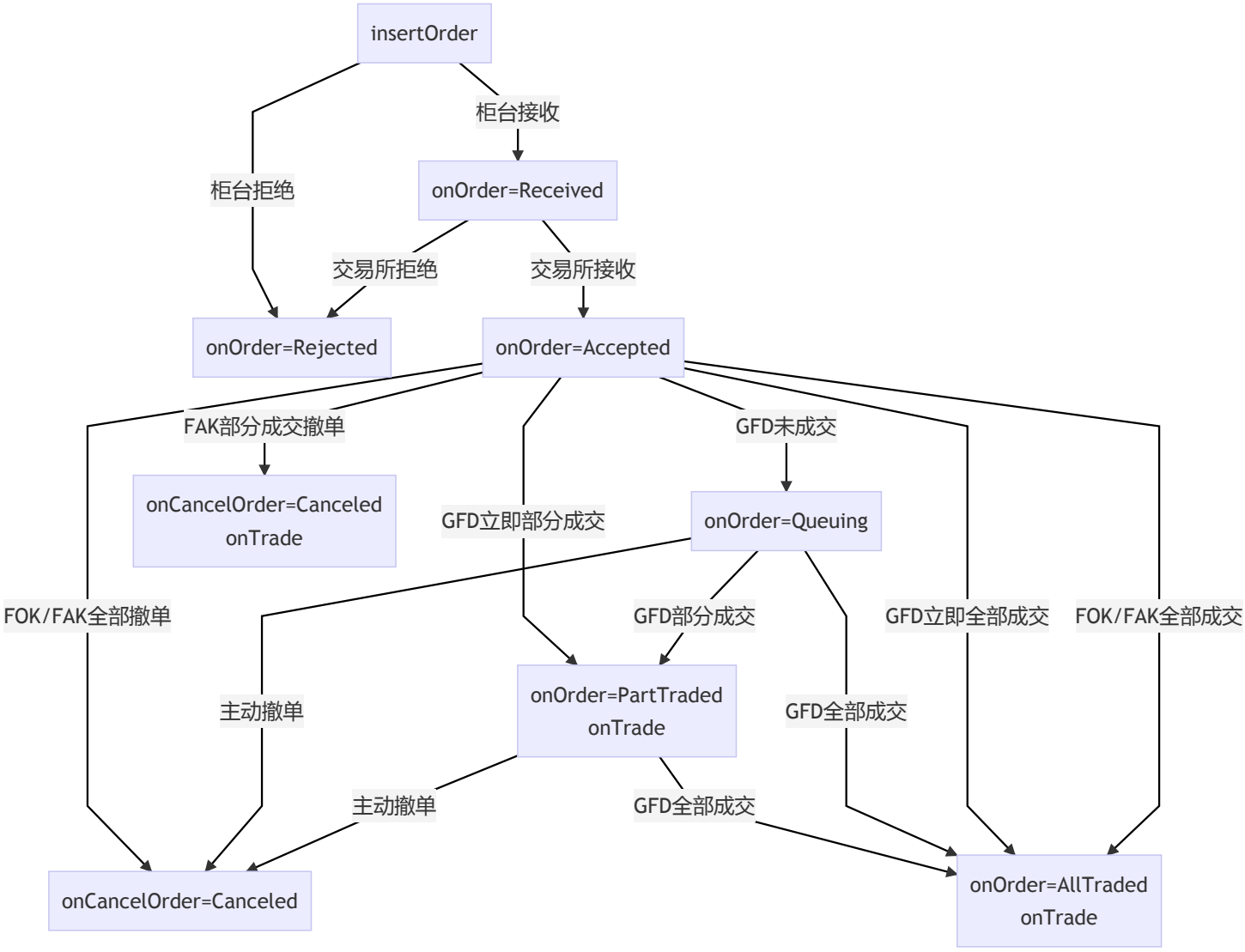
无

交易接口

交易类回调接口，主要是提供交易的结果和状态通知。

API提供了三类交易接口：普通订单、行权对冲单、套利订单、询价。

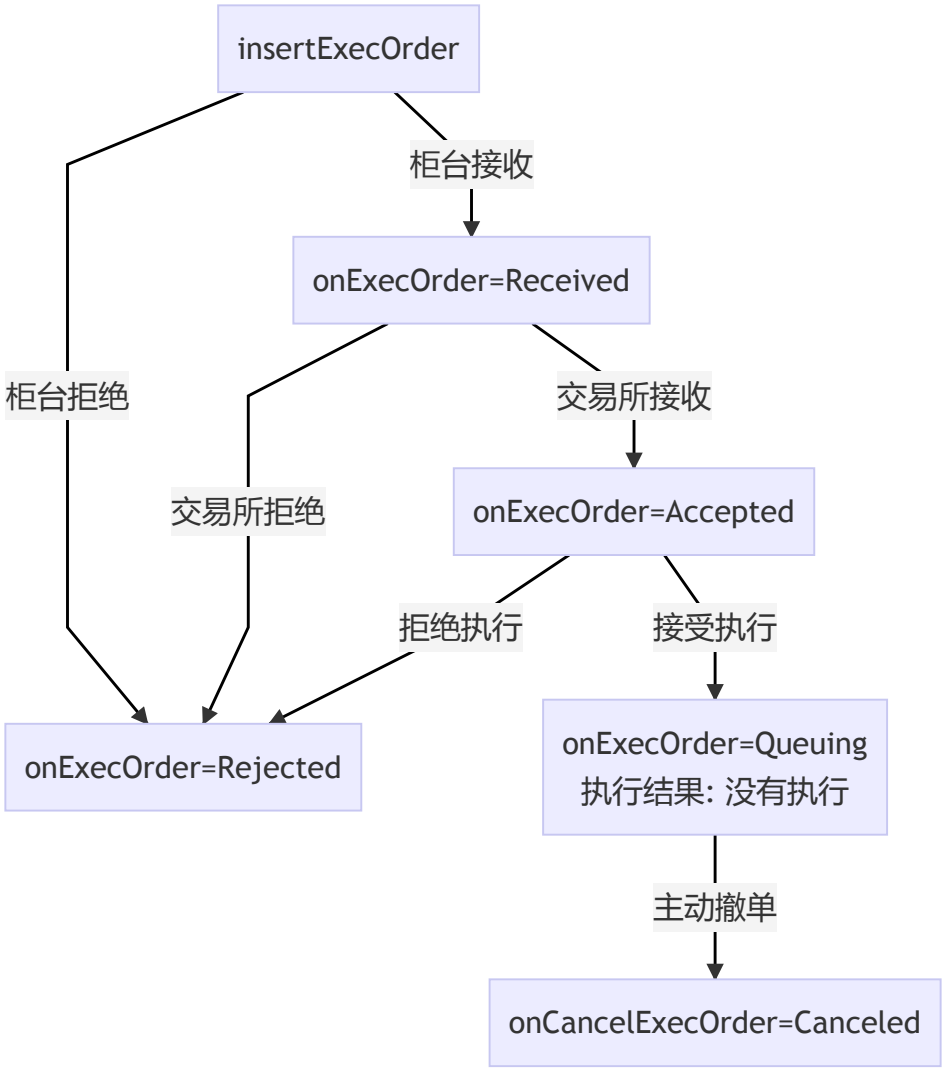
普通订单的回调流程图如下：



普通订单的回调接口包括：

- onOrder(): 主要处理发送订单的结果和订单状态变更的通知（如果是已撤状态走onCancelOrder，其他场景的状态变更走onOrder）；
- onCancelOrder(): 主要处理撤销订单的结果。撤单成功时，当订单状态变成已撤回调该接口；撤单失败时，回调该接口通知撤单失败；
- onTrade()：主要处理订单成交的通知；

行权对冲单的回调流程图如下：



行权对冲单的回调接口包括：

- onExecOrder()：主要处理发送订单的结果和订单状态变更的通知（如果是已撤状态走onCancelExecOrder，其他场景的状态变更走onExecOrder）；
- onCancelExecOrder()：主要处理撤销订单的结果。撤单成功时，当订单状态变成已撤回调该接口；撤单失败时，回调该接口通知撤单失败；

询价回调接口包括：

- onDemandQuote()：处理询价的结果；

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_PartTraded
- XTF_OS_AllTraded

> 【@since 4.1.1390】
> 自版本 4.1.1390 起，行权/对冲单增加了级别字段 execLevel，可以按合约、系列、品种及投资者级别来进行操作，当前仅大商所对冲有效。当用户收到 onExecOrder() 事件后，需要先判断 execLevel 是哪种类型，再根据业务逻辑进行相应的处理。

示例代码：

```
void XxxSpi::onExecOrder(int errorCode, const XTFOrder *order) {
    switch (order->getExecLevel()) {
        case XTF_EXL_Instrument: {
                auto instrument = order->instrument;
                ... // do something with instrument object.
            }
            break;
        case XTF_EXL_Series: {
                auto series = order->instrument->getSeries();
                ... // do something with series object.
            }
            break;
        case XTF_EXL_Product: {
                auto product = order->instrument->getProduct();
                ... // do something with product object.
            }
            break;
        case XTF_EXL_Investor: {
                ... // do something for current investor.
            }
            break;
        defalut: {
                auto instrument = order->instrument;
                ... // do something with instrument object by default.
            }
            break;
    }
}
```

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

onArbiOrder方法

接口功能

套利单回报及订单状态改变通知接口。

接口原型

```
void onArbiOrder(int errorCode, const XTFArbiOrder *arbiOrder);
```

接口参数

- errorCode：报单操作错误码
- arbiOrder：套利单回报对象，参见XTFArbiOrder结构体；

返回值

无

onCancelArbiOrder方法

接口功能

套利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 onCancelArbiOrder(int errorCode, const XTFArbiOrder *arbiOrder);
```

接口参数

- errorCode：撤单操作错误码；
- arbiOrder：被撤套利单对象。详情参见XTFArbiOrder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 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 cancelOrder(const XTFInputAction &inputAction, const XTFExchange *exchange); // 按撤单录入和交易所撤单
int cancelOrder(XTFOrderIDType orderIDType, long orderID, const XTFExchange *exchange); // 按单号和交易所撤单
int cancelOrders(const XTFOrder *orders, size_t orderCount); // 批量撤单接口，最大数量为16个
int cancelOrders(XTFOrderIDType orderIDType, long orderIDs[], size_t orderCount); // 批量撤单接口，最大数量为16个
int cancelOrders(const XTFInputAction inputActions[], size_t actionCount, const XTFExchange *exchange); // 按撤单录入和交易所批量撤单，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；
- orderIDs：批量撤单的柜台流水号、本地报单编号或交易所单号数组，如果是本地报单编号，要求本地报单编号具备唯一性；
- inputAction：撤单录入对象，提供了更多撤单参数；详情参考XTFInputAction结构体；
- inputActions：撤单录入对象，提供了更多撤单参数；详情参考XTFInputAction结构体；
- exchange：交易所对象指针；

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);
int cancelExecOrder(XTFOrderIDType orderIDType, long orderID, const XTFExchange *exchange);
```

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；
- exchange：交易所对象；

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。

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。

指定前置撤普通单方法

接口功能

指定前置 撤普通单。

在 `XTFSpi::onCancelOrder()` 接口中通知撤单的结果和状态。

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；

接口原型

```
int cancelOrder(const XTFInputAction &action, const XTFExchange *exchange);
int cancelOrders(const XTFInputAction actions[], size_t actionCount, const XTFExchange *exchange);
```

接口参数

- action：撤单录入信息；
- actions：撤单录入信息数组；
- actionCount：撤单录入信息数组大小，不能超过16个，否则会返回ERRXTFAPI_OrderCountExceeded错误；
- exchange：订单所属的交易所对象，每次调用只能撤销单个交易所的订单，如果是多交易所的场景，需要明确指定所属交易所。
  如果只有一个交易所，可以传入nullptr，默认为当前交易所对象；

返回值

接口调用成功返回0，否则返回错误码，调用成功并不表示撤单操作的结果。

用法示例

```
XTFExchange *exchange = api->getExchange(1); // 批量订单编号所属的交易所对象，这里选择第二个交易所；
XTFInputAction action{};
action.actionType = XTF_OA_Cancel;
action.channelSelectionType = XTF_CS_Fixed;
action.channelID = exchange->getChannel(2); // 指定交易席位进行撤单，比如：使用第三个交易席位；
action.orderIDType = XTF_OIDT_System; // 使用柜台流水号进行撤单
action.orderID = 100; // 柜台流水号
action.localActionID = ++lastLocalActionID; // 用户本地撤单编号
int ret = api->cancelOrder(action, exchange); // 发送数据到柜台进行撤单
if (ret != 0) {
    printf("cancel order failed, ret=%d.\n", ret);
} else {
    printf("cancel order request ok.\n");
}
```

insertArbiOrder方法

接口功能

发送套利报单。

在 XTFSpi::onArbiOrder()或XTFSpi::onCancelArbiOrder() 接口中通知用户插入套利单的结果和订单状态。

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 insertArbiOrder(const XTFInputArbiOrder &inputArbiOrder);
```

接口参数

- inputArbiOrder：待插入的套利单参数，参见XTFInputArbiOrder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。

cancelArbiOrder方法

接口功能

发送套利撤单。

在XTFSpi::onCancelArbiOrder()接口中通知撤销的结果和订单状态。

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XTFArbiOrder对象直接撤单，务必使用API返回的XTFArbiOrder对象指针，用户不能从外部构造一个XTFArbiOrder来撤单；
> 3. 不建议使用交易所单号撤单；

接口原型

```
int cancelArbiOrder(const XTFArbiOrder *arbiOrder);
int cancelArbiOrder(const XTFInputAction &action, const XTFExchange *exchange);
```

接口参数

- arbiOrder：待撤销的套利单对象，参见XTFArbiOrder结构体；
- action：撤单录入对象，参见XTFInputAction结构体；
- exchange：交易所对象指针；

返回值

接口调用成功返回0，否则返回错误码，调用成功并不表示撤单操作的结果。

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；空指针表示订阅所有合约对象。

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 unsubscribe(const XTFInstrument *instrument);
```

接口参数

- instrument：待取消订阅的合约对象，参见XTFInstrument结构体；空指针表示取消订阅所有合约对象。

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；

getArbiInstrumentCount方法

接口功能

获取套利合约数量信息。

接口原型

```
int getArbiInstrumentCount();
```

接口参数

无

返回值

- 大于等于0：表示套利合约数量；

- 小于0：表示查询发生的错误码；

getArbiInstrument方法

接口功能

根据位置查询套利合约信息。

接口原型

```
const XTFArbiInstrument* getArbiInstrument(int pos);
```

接口参数

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

返回值

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

getArbiInstrumentByID方法

接口功能

根据套利合约编号字符串查询套利合约信息。

接口原型

```
const XTFArbiInstrument* getArbiInstrumentByID(const char *ArbiInstrumentID);
```

接口参数

- ArbiInstrumentID：套利合约编号字符串，例如：`SPD TA612&TA706`等；

返回值

- 套利合约对象指针，参见XTFArbiInstrument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；

findArbiOrders方法

接口功能

根据指定的过滤条件查询套利单信息。

接口原型

```
int findArbiOrders(const XTFArbiOrderFilter &filter, unsigned int count, const XTFArbiOrder *orders[]);
```

接口参数

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

返回值

返回实际查询到的套利单数量。
如果查询的实际套利单数大于指定的数量count，则取前面的count个报价，并返回count；
如果查询的实际套利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；

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
"WARM_ENABLED"：
    预热报单功能是否启用，取值范围：[true,false]，默认为false：表示不启用；
    UDP报单方式下生效，TCP报单时不支持预热报单；
"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
"TRADE_LOG_ENABLED"：是否启用交易明细日志功能。true表示启用，false表示不启用，默认为不启用。
```

启用后会在xtf-api-log日志目录下生成xtf-trade-xxx.log日志文件，记录了上下行交易明细数据，可用于调试。
启用该功能会对下行处理速度有影响。生产环境下，不建议启用。
"RELEASE_FINISHED_ORDER_ENABLED"：是否启用完结报单内存资源自动回收功能。true表示启用，false表示不启用，默认为不启用。
启用此功能需要非常谨慎。如果报单量非常大，且无成交的撤单占据大部分，那么可以启用此功能。其他场景下，建议不要启用。
启用后对于所有的错单、拒单和撤单（无成交），其分配的内存资源会被自动回收。
如果启用此功能，报单内存可能会被回收利用，用户需要根据报单回报消息和报单状态变化，来判断XTFOrder对象指针是否有效，
只有状态是已接收、正在队列中、部分成交和全部成交的报单，内存资源会持久保留、不会回收，这些报单的指针是有效的。
其他状态的报单指针是无效的，需要丢弃。
"HISTORICAL_FINISHED_ORDER_FILTER_ENABLED"：是否启用历史完结订单过滤功能（仅做市商柜台支持此功能）
1、普通单：错单和撤单（无成交）的回报流水，不再推送给客户端；
2、报价单：状态为错单、两腿衍生单都是撤单且无成交的回报流水，不再推送给客户端；
3、衍生单：所有报价产生的衍生单，错单和撤单（无成交）的回报流水，不再推送给客户端；
如果启用此过滤功能，客户端可以收到的历史回报流水数据包括：
1、普通单：状态是已接收、正在队列中、部分成交、全部成交和部分成交的撤单的回报流水；
2、衍生单及其原生报价单：状态是已接收、正在队列中、部分成交、全部成交和部分成交的撤单的回报流水及其相关的报价单流水；
3、其他不在上述过滤条件中的回报流水；
此功能不影响登录后的回报流水，登录后的所有报撤单流水，全量返回给客户端。
"XNIC_ENABLED"：是否启用低延迟网卡功能。
如果启用此配置项，那么API启动时会自动检测当前是否有可用的低延迟网卡，如果有则使用低延迟网卡快速通道进行报撤单。
如果禁用此配置项，那么API启动时不会检测是否有可用的低延迟网卡，默认走系统协议栈通道进行报撤单。
若启用该功能，建议使用onload-7.1.x版本或exanic2.7.x版本的驱动。
默认为启用。
"XNIC_NAME"：指定启用的低延迟网卡名称。
如果启用此配置项，则使用指定的网卡；反之，则自动选择一个可用的网卡。
建议使用默认配置，由API自动检测可用网卡。
@since 4.1.1350
"XNIC_TYPE"：指定启用的低延迟网卡类型。
如果启用此配置项，则使用指定类型的驱动；反之，则自动选择匹配的网卡驱动。
支持类型范围：[sfc, exanic, swiftn]，建议使用默认配置，由API自动检测可用网卡类型。
@since 4.1.1350
"ORDER_GROUP_ID"：配置用户报单分组编号，由用户自定义。取值范围：[0, 128]，默认为0。
"COMBINATION_ENABLED"：是否启用仓位的自动组合功能。
true表示启用，false表示不启用，默认为不启用。
@since 4.1.1225
"COMBINATION_TYPES"：指定自动组合的类型列表。默认表示自动组合所有的类型。
组合类型取值范围请参考XTFDataTypes.h文件中XTFCombType的枚举定义。
组合类型列表的每个元素，以逗号','分割。
例如：针对大商和广期所，可以按照如下配置自动组合的类型：
- 自动组合期货对锁、期权对锁：COMBINATION_TYPES=0,1
- 自动组合期货对锁、跨期套利和跨品种套利：COMBINATION_TYPES=0,2,3
注意：此配置选项需要与自动组合开关 COMBINATION_ENABLED 选项配合使用
@since 4.1.1225
"CHECK_VERSION_ENABLED"：是否启用头文件和库的版本一致性检测。
如果启用此配置项，那么API启动时会自动检测当前使用的头文件版本和库版本是否一致。如果不一致则启动失败，并返回相应的错误码。
如果禁用此配置项，那么API启动时不会检测头文件版本和库版本是否一致。
默认为启用。如需要禁用，请设置此选项为 CHECK_VERSION_ENABLED=false。
@since 4.1.1350

"MD_ENABLED"
指定是否启用柜台的TCP行情服务。
true 表示启用行情服务，false 表示不启用。
当启用行情服务时，API登录成功且静态数据推送完成之后，会自动发起行情服务连接。
连接成功后，开始接收柜台定时推送的行情数据。默认情况下，不会回调通知行情数据。
用户调用了 subscribe 接口订阅成功后，API会通过 onBookUpdated 接口通知行情数据。
如果连接失败，API会记录错误信息到日志文件，并无主动通知的接口。
如果用户关注行情且长时间没有收到行情数据，可以检查API日志，以确认行情的错误原因。
默认不启用行情。
@since 4.1.1360

"MD_SERVER_IP"
指定柜台的TCP行情服务IP地址
仅当 MD_ENABLED=true 时，配置生效。
@since 4.1.1360

"MD_SERVER_PORT"
指定柜台的TCP行情服务端口
仅当 MD_ENABLED=true 时，配置生效。
@since 4.1.1360

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）。

回报过滤规则分成三种条件：1）按品种类型过滤；2）按合约编号模糊匹配；3）按报单组编号过滤；

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*
```

按报单组编号过滤，只需要启用`orderGroupIdEnabled`字段即可。

- true表示仅收到本报单分组的回报；
- false表示收到全部回报；

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，参见XTFReportFilter结构体；

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，参见XTFReportFilter结构体；

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。

setChannelPriorities方法

接口功能

投资者指定席位优先级。

约束说明：

- 投资者需要配置权限后才能指定席位优先级；
- 每次设置优先级后，会有一个生效时段，超过该时段后，席位优先级自动恢复原来的配置。具体的时段配置，参考柜台配置说明；
- 该接口为同步接口；

接口原型

```
int setChannelPriorities(const XTFExchange *exchange, uint32_t *priorities, size_t prioritiesCount,
XTFChannelSelectionStrategy strategy);
```

接口参数

- exchange：待设置的交易所对象。如果柜台是单交易所，该参数可以传入null表示默认的交易所对象；
- priorities：席位优先级数组，传递每个席位的优先级权重值；
- prioritiesCount：席位优先级数组长度，用来和交易所席位数进行比较。
  1）如果数组长度大于实际的席位数，则只取前面的席位数量对应的优先级；
  2）如果数组长度小于实际的席位数，则后面默认的席位优先级为0；
  3）prioritiesCount的最大值为16；
- strategy：1表示优先发送策略，2表示概率发送策略。默认为1；

返回值

0表示设置成功，非0表示本次设置失败对应的错误码。

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用户线程发送；

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。

sendWarmOrder方法

接口功能

传入合约参数，API自动向柜台发送一个携带该合约信息的预热单。

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用户线程发送；

接口原型

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

接口参数

- instrument：合约指针，不能为空；

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 XTFUserData {
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。需等待流水数据推送完成之后生效
    const XTFOrder* getOrder(int pos) const;    ///< 按位置索引查询报单对象，从0开始计算
    int             getArbiOrderCount() const;  ///< 查询所有套利报单数量，用于遍历查询所有套利报单列表。需等待流水数据推送完成之后
生效
    const XTFArbiOrder* getArbiOrder(int pos) const;///< 按位置索引查询套利报单对象，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。需等待日初数据推送完成之后生效
    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。需等待流水数据推送完成之后生效
    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。需等待流水数据推送完成之后生效
    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。需等待流水数据推送完成之后
生效
    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。需等待流水数据推送完成之后生效
    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; ///< 本地单号撤单时柜台允许的最大本地单号，0表示不支持本地单号撤单
    XTFMarginType   getMarginType() const;          ///< 查询柜台的保证金计算类型
};
```

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()
    const char*         getSeatCode(int pos) const; ///< 按位置索引查询席位代码。席位代码交易所范围唯一，但会根据配置而变化
    const char*         getSeatCodeByID(uint8_t id) const; ///< 按席位编号查询席位代码，返回值同getSeatCode()
    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
    const char*         getClientID() const;        ///< 获取该交易所对应的客户代码
};
```

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：品种信息对象类

```
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
};
```

XTFSeries结构体

说明：系列信息对象类

```
class XTFSeries {
public:
    XTFSeriesID         seriesID;                   ///< 系列代码
};
```

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
    double                  strikePrice;                ///< 执行价（仅期权有效）
    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
    int                     tradingSegmentSN;           ///< 交易阶段编号（大商所和广期所暂未使用此字段）
    XTFInstrumentStatusEnterReason enterReason;         ///< 进入本状态原因（大商所和广期所暂未使用此字段）
    XTFTime                 enterTime;                  ///< 进入本状态时间
    XTFDate                 enterDate;                  ///< 进入本状态日期（大商所和广期所暂未使用此字段）
    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
    int                     getMessageAmount(uint32_t &total, uint32_t &traded) const; ///< 查询当前实时统计的信息量，返回值为0表示查询成功，其他表示错误码
    double                  getNextOrderCommissionRate() const;///< 根据合约当前实时统计的申报量和OTR，查询下一笔报单需要的申报费率。如果数据错误或者查找失败，则返回DBL_MAX
    const XTFSeries*        getSeries() const;              ///< 查询合约所属的XTFSeries指针，部分交易所可能没有所属系列的概念，此时返回为空指针
};
```

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 ≤ 1，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
    XTFExecLevel            execLevel;              ///< 行权/对冲级别。大商所该字段若为0按照合约级别处理，若为异常值柜台拒单；非大商所
该字段无效，不做校验
};
```

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；
    XTFOrderGroupID         orderGroupID;           ///< 报单分组编号（启动API之前由用户自定义，取值范围[0~128]）
    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;
    XTFExecLevel            getExecLevel() const;   ///< 如果是行权或对冲，可以使用此接口查询execLevel字段的值
};
```

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。

    XTFOrderGroupID         orderGroupID;           ///< 报单分组编号
    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
    XTFLocalOrderID         getLocalOrderID() const;///< 成交回报关联的订单本地单号
    XTFSysOrderID           getSysOrderID() const;  ///< 成交回报关联的订单流水号
    bool                    isArbitrage() const;    ///< 区分是普通或报价的成交还是套利成交回报
    const XTFArbiOrder*     getArbiOrder() const;   ///< 所属XTFArbiOrder的指针
};
```

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。通过updateBook()接口更新时，此字段值无效
    int             snapTime;                   ///< 时间戳。通过updateBook()接口更新时，此字段值无效
    double          preSettlementPrice;         ///< 前结算价
    double          preClosePrice;              ///< 前收盘价
    double          preOpenInterest;            ///< 前持仓量。通过updateBook()接口更新时，此字段值无效
    double          upperLimitPrice;            ///< 涨停价
    double          lowerLimitPrice;            ///< 跌停价
    double          openPrice;                  ///< 开盘价。通过updateBook()接口更新时，此字段值无效
    double          lastPrice;                  ///< 最新价
    double          bidPrice;                   ///< 买一价
    int             bidVolume;                  ///< 买一量
    double          askPrice;                   ///< 卖一价
    int             askVolume;                  ///< 卖一量
    int             volume;                     ///< 成交量。通过updateBook()接口更新时，此字段值无效
    double          turnover;                   ///< 成交金额。通过updateBook()接口更新时，此字段值无效
    double          openInterest;               ///< 持仓量。通过updateBook()接口更新时，此字段值无效
    double          indexPrice;                 ///< 标的指数价，指数类期权合约有效
    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:
    XTFCombInstrumentID     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
    XTFSysOrderID           sysOrderNo;
    XTFTime                 demandTime;
};
```

XTFReportFilter结构体

说明：回报过滤器对象类（白名单）

```
class XTFReportFilter {
public:
    XTFProductFilter        productFilter;                  ///< FUTURES ||(&&) OPTIONS
    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*
    bool                    orderGroupIdEnabled;            ///< 是否通过报单分组编号过滤回报 true表示仅收到报单分组的回报，false表示
收到全部回报；
};
```

XTFInputAction结构体

说明：撤单录入对象类

```
class XTFInputAction {
public:
    XTFOrderActionType      actionType;                     ///< 目前仅支持撤销报单
    XTFChannelSelectionType channelSelectionType;           ///< 席位编号选择类型
    XTFChannelID            channelID;                      ///< 席位编号
    XTFOrderIDType          orderIDType;                    ///< 通过报单编号撤单时，编号的类型，支持：本地单号、柜台流水号、交易所单号
    XTFOrderID              orderID;                        ///< 通过报单编号撤单时，待撤销的报单编号
    XTFLocalActionID        localActionID;                  ///< 客户本地撤单编号
};
```

XTFArbiInstrument结构体

说明：套利合约类

```
class XTFArbiInstrument {
public:
    XTFArbiInstrumentID     arbiInstrumentID;               ///< 套利合约编号
    uint32_t                arbiInstrumentIndex;            ///< 套利合约序号
    XTFArbiType             arbiType;                       ///< 套利组合类型
    XTFArbiFlag             arbiFlag;                       ///< 套利合约标志
    XTFCombHedgeFlag        combHedgeFlag;                  ///< 投机套保标志，当前仅支持投机-投机
    XTFArbiLegNum           legNum;                         ///< 套利合约腿数目
    XTFArbiLegRatio         legRatio;                       ///< 套利合约腿比例
    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
    double                  upperLimitPrice;                ///< 价格上限
    double                  lowerLimitPrice;                ///< 价格下限
    mutable XTFUserData     userData;                       ///< 保留给用户使用的数据对象

    const XTFExchange*      getExchange() const;            ///< 套利合约所属交易所
    const XTFInstrument*    getInstrumentLeg1() const;      ///< 腿1合约对象指针
    const XTFInstrument*    getInstrumentLeg2() const;      ///< 腿2合约对象指针
    const XTFCombPosition*  getCombPosition(XTFDirection direction) const; ///< 查询套利组合持仓对象
};
```

XTFInputArbiOrder结构体

说明：套利报单请求对象类

```
class XTFInputArbiOrder {
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。
    XTFOrderType            orderType;          ///< 报单类型：限价(GFD)/市价/FAK/FOK/GIS
    XTFOrderFlag            orderFlag;          ///< 报单标志，当前默认为XTF_ODF_Arbitrage
    XTFCombHedgeFlag        combHedgeFlag;      ///< 投机套保标记，当前仅支持投机-投机
    XTFCombOffsetFlag       combOffsetFlag;     ///< 开平仓标志，有效位数与XTFArbiInstrument.legNum保持一致
    XTFDirection            direction;          ///< 买卖方向
    double                  price;              ///< 报单价格，可以为负数
    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 ≤ 1，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（当指定席位报单时，应从XTFAccount::getChannel(pos)接口查询席位编
号后填入此处，否则可能是无效的席位编号）
    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 XTFArbiInstrument *arbiInstrument;    ///< 报单合约对象
};
```

XTFArbiOrder结构体

说明：套利单回报对象类

```
class XTFArbiOrder {
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；
    XTFOrderGroupID         orderGroupID;       ///< 报单分组编号（启动API之前由用户自定义，取值范围[0~128]）
    XTFArbiType             arbiType;           ///< 套利组合类型
    XTFArbiLegNum           legNum;             ///< 套利合约腿数目，目前最大支持的腿数为3
    XTFArbiLegRatio         legRatio;           ///< 套利合约腿比例
    XTFCombHedgeFlag        combHedgeFlag;      ///< 投机套保标记，当前仅支持投机-投机
    XTFDirection            direction;          ///< 买卖方向
    XTFCombOffsetFlag       combOffsetFlag;     ///< 开平仓标志，有效位数与legNum保持一致
    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
    XTFOrderStatus          orderStatus;        ///< 报单状态
    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
    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。
```

```
    XTF_CONST XTFArbiInstrument *arbiInstrument;    ///< 所属XTFArbiInstrument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
};
```

XTFArbiOrderFilter结构体

说明：查询套利单信息的过滤器对象类

```
class XTFArbiOrderFilter {
public:
    XTFTime             startTime;          ///< 开始时间，字符串格式：dd:dd:dd，例如：09:20:30，空字符串表示所有
    XTFTime             endTime;            ///< 结束时间，字符串格式：dd:dd:dd，例如：14:20:30，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
    XTFOrderType        orderType;          ///< 指定报单指令类型，仅支持FAK/FOK/GIS/GFD(Limit)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
    XTFArbiType         arbiType;           ///< 指定套利类型，无效值时表示所有
    XTFCombOffsetFlag   combOffsetFlag;     ///< 指定组合开平标志，无效值时表示所有
    XTF_CONST XTFArbiInstrument *instrument;///< 指向套利合约结构的指针，空指针表示所有

    XTFArbiOrderFilter() {
        startTime[0] = '\0'; startTime[sizeof(XTFTime)-1] = '\0';
        endTime[0] = '\0'; endTime[sizeof(XTFTime)-1] = '\0';
        direction = XTF_D_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;
        arbiType = XTF_ARBI_Invalid;
        combOffsetFlag[0] = XTF_OF_Invalid;
        combOffsetFlag[1] = XTF_OF_Invalid;
        combOffsetFlag[2] = XTF_OF_Invalid;
        combOffsetFlag[3] = XTF_OF_Invalid;
        instrument = nullptr;
    }
};
```

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
XTFCombInstrumentID	char[36]	组合合约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
XTFSeriesID	char[19]	系列数据类型
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
XTFQuoteDemandID	int64_t	询价编号
XTFUserRef	int32_t	用户自定义的数据，会发送到柜台，由柜台原值返回

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	期权对冲报单	3
XTF_ODF_DerivedFromQuote	报价衍生单	4
XTF_ODF_FuturesSelfClose	期货对冲报单（目前仅大商所支持）	5
XTF_ODF_Arbitrage	套利报单	6
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_GIS	GIS报单（限价）	4
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
XTF_ODT_Invalid	无效值	255

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	255

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。报单时自动优选席位，撤单时使用报单席位；	0
XTF_CS_Fixed	指定交易所席位。如果指定的席位不可用或不存在，则返回错误；	1
XTF_CS_FixedThenAuto	指定交易所席位，如果指定的席位不可用或不存在，则报单时自动优选席位，撤单时使用报单席位； 此功能需要柜台不低于R004-P1版本	2
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_MaxPreSettlementOrLastPrice	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_PreSettlementPrice	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

席位优选策略（XTFChannelSelectionStrategy）

类型：uint8_t

枚举类型	说明	枚举值
XTF_CSS_Priority	优先策略：按照设置的优先级进行遍历 如果席位队列未达上限，则加入此席位的发送队列； 如果席位队列已经达到上限，则选择优先级较低的下一个席位。	1
XTF_CSS_Probability	概率策略：按照设置的优先级作为概率分布，以概率的方式选择席位。	2

保证金计算类型（XTFMarginType）

类型：uint8_t

枚举类型	说明	枚举值
XTF_MT_DefaultMargin	默认保证金算法，各交易所有所不同，参考交易所的说明	0
XTF_MT_ShfeSpmmMargin	上期/能源所 SPMM 保证金算法	1
XTF_MT_CffexRcamsMargin	中金所 RCAMS 保证金算法	2
XTF_MT_CzceSpbmMargin	郑商所 SPBM 保证金算法	3
XTF_MT_DceRuleMargin	大商所 DCE 保证金算法	4

报价顶单类型（XTFQuoteReplaceType）

类型：uint8_t

枚举类型	说明	枚举值
XTF_QRT_ReplaceNone	不顶单	1
XTF_QRT_ReplaceLast	顶最近一笔报价单	2
XTF_QRT_ReplaceSpecific	顶指定的报价单	3

行权/对冲级别（XTFExecLevel）

类型：uint8_t

枚举类型	说明	枚举值
XTF_EXL_Instrument	按合约级别行权/对冲	1
XTF_EXL_Series	按系列级别行权/对冲	2
XTF_EXL_Product	按品种级别行权/对冲	3
XTF_EXL_Investor	按投资者级别行权/对冲	4

套利合约类型（XTFArbiType）

类型：uint8_t

枚举类型	说明	枚举值
XTF_ARBI_CzceSPD	郑商所 SPD 期货跨期	1
XTF_ARBI_CzceIPS	郑商所 IPS 期货跨品种	2
XTF_ARBI_CzceSTD	郑商所 STD 期权跨式	3
XTF_ARBI_CzceSTG	郑商所 STG 期权宽跨式	4
XTF_ARBI_CzcePRT	郑商所 PRT 备兑（暂未使用）	5
XTF_ARBI_Invalid	无效值	255

套利合约标志（XTFArbiFlag）

类型：uint8_t

枚举类型	说明	枚举值
XTF_ARBF_Standard	标准套利合约	1
XTF_ARBF_DceUDS	大商所用户自定义期权套利合约	2
XTF_ARBI_Invalid	无效值	255

套利合约腿比例（XTFArbiLegRatio）

类型：uint8_t

枚举类型	说明	枚举值
XTF_ARBI_LegRatio_1v1	套利组合合约腿比例, 默认是1:1	1

保证金计算说明

在创建API对象时，如果设置了启用资金计算和仓位计算的配置参数，那么API会根据回报数据实时计算账户的资金和仓位。

保证金计算配置

涉及的配置参数：

```
# 设置API的运行模式，0-表示正常模式，1-表示极简模式，默认为正常模式
# 正常模式下，支持所有交易数据的本地存储和查询，可以支持持仓和资金计算；
# 极简模式下，不支持交易数据的本地存储和查询，不支持持仓和资金计算；
```

```
# 此参数只能从配置文件设置，API的setConfig()接口不生效；
RUN_MODE=0

# 是否启用仓位计算（RUN_MODE必须等于0时，生效）
# - true：启用
# - false：不启用
POSITION_CALC_ENABLED=true

# 是否启用资金计算（RUN_MODE必须等于0时，生效）
# 如果启用资金计算，默认自动启用仓位计算，否则无法计算资金
# - true：启用
# - false：不启用
MONEY_CALC_ENABLED=true
```

> 说明：默认情况下，API是自动启用资金和仓位计算的。

保证金查询方法

启用了资金仓位计算后，API登录交易柜台，会自动根据柜台推送的流水数据计算仓位和资金，在收到 `onLoadFinished()` 回调之后，表示数据追平，且已经完成了资金和仓位的计算。可以通过下面的方法查看对应的数据：

```
XTFApi *api;
... // 调用 makeXTFApi() 成功创建，且登录成功

auto account = api->getAccount();

// 查询占用保证金（组合保证金算法场景下，表示所有持仓的组合保证金总和）
auto paidMargin = account->margin;

// 查询冻结保证金（组合保证金算法场景下，表示所有委托的冻结保证金总和）
auto frozenMargin = account->frozenMargin;
```

保证金优惠说明

中金/上期的单向大边保证金

中金所期货单向大边优惠是针对同一个品种组的保证金优惠，一个品种组内包含多个相关品种（如权益类期货，IC、IF、IH品种归属于一个品种组）。 同一个投资者的同一个品种组中内，参与单向大边优惠的合约先按照多、空方向分别计算出传统保证金，再分别汇总累加到品种组级别上的多、空总保证金，两者中的较大者为该投资者在该品种组上的实收保证金。

$$投资者在某品种组i的保证金 = max(\sum参与单向大边的各合约j的多头保证金, \sum参与单向大边的各合约j的空头保证金) + \sum不参与单向大边的各合约k的(多头保证金 + 空头保$$

$$投资者总保证金 = \sum投资者在各品种组i的保证金$$

上期所期货单向大边优惠是针对同品种内的保证金优惠，即同一个投资者的同一品种中，参与单向大边优惠的合约先按照多、空方向分别计算出传统保证金，再分别汇总累加到品种级别上的多、空总保证金，两者中较大者为该投资者在该品种上的实收保证金。

$$投资者在某品种i的保证金 = max(\sum参与单向大边的各合约j的多头保证金, \sum参与单向大边的各合约j的空头保证金) + \sum不参与单向大边的各合约k的(多头保证金 + 空头保证$$

$$投资者总保证金 = \sum投资者在各品种i的保证金$$

期权合约只收卖方的保证金，因此不涉及单向大边优惠。

大商/广期的组合策略优惠保证金

交易所提供套利订单和组合申请两种方式让投资者享受组合保证金优惠。柜台目前支持的是第二种，盘中按照交易所规定的组合策略优先级，定时将投资者在各合约上的单腿持仓进行组合，实时计算优惠后的组合保证金，剩余未组合的单腿持仓仍按传统保证金算法计收。

注意，在平仓时，系统释放保证金遵循先平单腿持仓，后平优惠组合持仓的原则。当平仓需打破优惠组合时，按照优惠组合持仓的优先级从低到高进行打破，根据组合拆分后的结果，柜台追收另一腿的传统保证金。

$$投资者总保证金 = \sum该投资者在各组合策略上的组合保证金 + \sum该投资者在各合约上的剩余单腿计收的传统保证金$$

期货

- 对锁：$组合保证金 = max(腿1合约价格 * 组合手数 * 合约乘数, 腿2合约价格 * 组合手数 * 合约乘数)$
- 跨期：$组合保证金 = max(腿1合约价格 * 组合手数 * 合约乘数, 腿2合约价格 * 组合手数 * 合约乘数)$
- 跨品种：$组合保证金 = max(腿1合约价格 * 组合手数 * 合约乘数, 腿2合约价格 * 组合手数 * 合约乘数)$

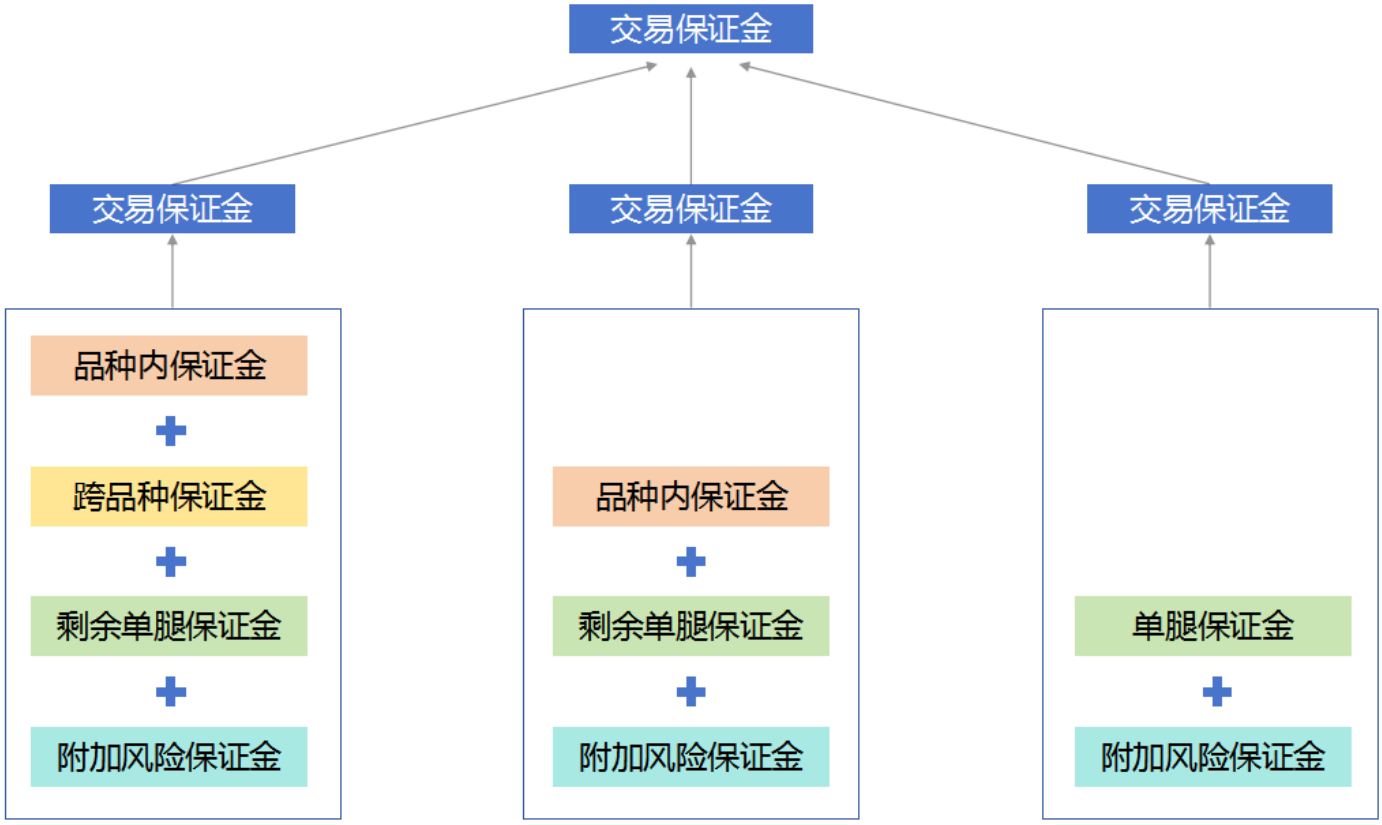
期权

- 卖出期权期货组合：$组合保证金 = 期货合约价格 * 组合手数 * 合约乘数 + 期权权利金 * 组合手数$
- 期权跨式：$组合保证金 = max(看涨期权传统保证金, 看跌期权传统保证金) + 另一方权利金 * 组合手数$。其中，$看涨期权传统保证金 = 看跌期权传统保证金$时，$组合保证金 = max(看涨期权传统保证金, 看跌期权传统保证金) + max(看涨期权权利金 * 组合手数, 看跌期权权利金 * 组合手数)$
- 期权宽跨式：$组合保证金 = max(看涨期权传统保证金, 看跌期权传统保证金) + 另一方权利金 * 组合手数$。其中，$看涨期权传统保证金 = 看跌期权传统保证金$时，$组合保证金 = max(看涨期权传统保证金, 看跌期权传统保证金) + max(看涨期权权利金 * 组合手数, 看跌期权权利金 * 组合手数)$
- 期权对锁：$组合保证金 = 卖期权传统保证金 * 优惠系数(0.2)$
- 买入垂直价差：$组合保证金 = 卖期权传统保证金 * 优惠系数(0.2)$
- 卖出垂直价差：$组合保证金 = min(执行价格之差 * 组合手数, 卖期权传统保证金)$
- 买入期权期货组合：$组合保证金 = 期货合约价格 * 组合手数 * 合约乘数 * 优惠系数(0.8)$

大商RuLe组合保证金

RuLe组合保证金是大商所面向全市场投资者开展的新型组合保证金优惠业务，柜台先按照不同合约类型分别计算优惠后的保证金，汇总后再按会员单位设置的投资者保证金系数计算实收保证金。

$$投资者总保证金 = (\sum 该投资者在各个一般月份合约i的保证金 + \sum 该投资者在各个临近交割合约j的保证金 + \sum 该投资者在各个非组合合约k的保证金) * 投资者保证金系数$$



郑商SPBM组合保证金

SPBM组合保证金是郑商所面向全体做市商开展的新型组合保证金优惠业务，柜台先按品种组（期货品种和以其为标的的期权品种，属于同一个品种组）分别计算合约内对锁保证金、品种内合约间对锁保证金、品种间对冲保证金、品种裸保证金以及附加总保证金后进行汇总累加，再和卖方期权最低风险取较大值，扣减买期权权利金超出的部分，最终得到品种组级别的非交割月份合约保证金。对于品种组内的交割月份合约，不参与风险对冲，直接按照单向大边计收交割保证金。

$$投资者在某品种组i的保证金 = max(合约对冲保证金 + 品种内对冲保证金 + 品种间对冲保证金 + 裸保证金 + 附加总保证金, 卖方期权最低风险) + 交割保证金 - 价值冲抵$$

。其中，$附加总保证金 = 合约内对锁附加保证金 + 品种内合约间附加保证金 + 品种间对冲附加保证金 + 品种裸附加保证金$

最后，将投资者在各个品种组的保证金进行汇总累加，按照投资者的保证金系数计算实收保证金。

$$投资者总保证金 = \sum 投资者在各品种组i的保证金 * 投资者保证金系数$$

上期SPMM组合保证金

SPMM组合保证金是上期所预计优先面向做市商开展的新型组合保证金优惠业务，组保业务参数是按照普通客户、做市商不同客户类型进行区分。组保业务参数是指期权纯粹价格风险比例、跨期优惠系数、跨品种优惠系数以及最小保证金比例。其中，EC品种的合约不参与优惠，仍按照传统保证金算法计收。其他参与优化的合约，是在现有期货合约和期权合约传统保证金计算方式的基础上，对同品种组内、跨品种组间进行线性价格风险对冲。

首先，按品种群（在一个品种群内，包含多个相关性的品种组）计算应收保证金、品种组内持仓与报单的优惠保证金、跨品种优惠以及最小保证金。

$$投资者在某品种群i的保证金 = max（品种群i的应收保证金 - 品种群i的跨期优惠保证金、品种群i的报单优惠保证金 - 品种群i的跨品种优惠保证金，品种群i最小保证金）$$

其中，品种群内的各个合约是按照标准算法、组保算法两种保证金算法区分处理，归属组保算法的合约会进行跨期优惠、报单优惠以及跨品种优惠计算，最后和最小保证金取较大值。归属标准算法的合约直接按应收保证金累加计收。

最后，将投资者在各个品种群的保证金进行汇总累加，按照投资者的保证金系数计算实收保证金。

$$投资者总保证金 = \sum 投资者在各品种群i的保证金 * 投资者保证金系数$$

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;
}
```