GATEWAY CPP SDK 用户使用手册

V3.2.8-2023/04/06

用户许可协议

版权与所有权声明

INSIGHT 由华泰证券股份有限公司设计、开发。系统及其文档的所有权归属于华泰证券股份有限公司（以下简称“华泰证券”或者“本公司”），并受中华人民共和国国家《著作权法》、《商标法》和国际协约条款的保护。由华泰证券股份有限公司负责系统的更新、维护和销售等活动。用户不得从本系统中删去版权声明，要保证为本系统的拷贝（全部或部分）复制版权声明，并同意制止以任何形式非法拷贝本系统及文档。未经授权擅自复制或散布本数据库的部分或全部内容，将会面对民事起诉。

“INSIGHT”的名称已受到注册商标和其它形式的所有权的保护。

用户许可协议声明

本协议一方为本系统的个人或机构使用者，另一方为华泰证券股份有限公司。用户使用本系统之前，须首先认可本许可协议，如持有异议，请不要使用，并于 30 日内删除安装软件并到本公司办理有关事宜。

本说明书中使用的“本产品”是指计算机软件、数据库、记录媒体、印刷品及在线文档或电子文档，包括收录于本软件产品的所有数据、资料文件、执行文件、附加功能、使用说明书、客户方案书、帮助文件及其它文件，且可能并不只限于以上内容。以下出现的“使用”是指在计算机硬盘或在 RAM、CD-ROM 等其它储存设备上存储、安装、执行或显示本软件产品于屏幕的行为。

许可协议条款

- 本系统仅给您提供唯一使用许可权。用户必须承诺不把本系统提供的全部或部分资料和数据以任何形式转移、出售和公开给任何第三者。
- 用户必须同意并保证，采取必要和合适的措施保护本系统提供的资料和数据的版权和所有权。
- 用户必须通知其所有相关使用者有关本系统的版权声明和本许可协议，并要求所有相关使用者都必须遵循本许可协议的一切条款。
- 用户必须同意在本许可协议终止前，一直承担本协议所要求的一切责任和义务。

许可协议终止

用户若违反本协议的任一条款或条件，华泰证券可以即时终止其使用许可。一旦许可权利被终止，您必须立即销毁本系统及文档的所有拷贝，或将其归还本公司。

适用法律

中华人民共和国《知识产权保护条例》、《著作权法》、《商标法》、《专利法》等。

免责条款

华泰证券尽力为用户提供可信的、准确的资料和数据，但无法完全保证其百分之百的准确和完整。因此，无论在什么情况下，由使用本系统所产生的任何形式的间接或直接的、特别或意外的、必然或偶然的损失和破坏，本公司概不负责。在上述情况发生时，即使本公司事先被告知此类事情有可能发生，本公司亦不对由此导致的任何后果承担责任。

本公司将尽快更新资料数据，但不承担由于使用数据资料延误造成的损失或责任。如果用户发现数据文件中的错误，请立即通知本公司，本公司将尽最大的努力在下一个版本中更正。

联系方式

如果用户对本协议条款有任何疑问，请按照如下方式与本公司联系：

华泰证券股份有限公司

地址：江苏省南京市建邺区江东中路 228 号华泰证券广场

邮编：210019

电话：95597

传真：025-83387416

电子邮件：service@htsc.com

业务联系方式：

电子邮件：insight@htsc.com

运行环境

硬件环境

CPU：64 位双核及以上处理器

内存：2G 及以上

千兆网卡：1000Mb/s

互联网连接带宽不低于 15Mb

操作系统和软件

操作系统：Windows7 及以上 Windows 版本、CentOS 7.3

编译器：Visual Studio 2013/2015/2017、GCC 4.8

网络环境

INSIGHT 服务地址及端口以账号发放时提供的信息为准。

安装与卸载

INSIGHT 系统下载与升级

访问 https://huatech.htsc.com.cn/，进入下载页面获取最新的 INSIGHT C++客户端版本。

SDK 目录结构说明

Gateway 客户端 SDK 目录如下所示：

```
INSIGHT-CPP-BUILD
├──cert    （存放加密认证所需文件，调用客户端需要指定该文件）
├──example    （示例，Linux 下直接为 functional_test）
├──include （包含目录）
|         ├──ACE_Wrappers
```

```
|           ├──mdc_gateway_client
|           |           └──include
|           |           └──protobuf    （数据模型.h 和.cc 文件，需要放入工程中）
|          ├──openssl-1.0.1e
|           └──protobuf-3.1.0
|                        └──include
└──libs    （Windows 静态库）
       ├──vs2013
       ├──vs2015
       └──vs2017
```

组件说明

依赖组件：

- ACE-6.4.3；
- protobuf-3.1
- openssl-1.0.1e
- 在 Linux 平台编译依赖于 GCC 版本，非 4.8 版本需要重新编译动态库，需要请联系华泰提供。

参考示例

参考示例使用说明

functional_test 是一个简单的测试项目，其中包含订阅、查询和回测功能的测试用例，用于测试系统连通性同时给具体开发提供参考。测试 Demo 采用命令行启动的方法运行，命令行参数为：

“user” “password” “ip” port “cert_folder” “export_folder” test_type

其中，user 为用户名，password 为密码，ip 为连接地址，port 为端口号，cert_folder 为 build/cert 文件夹所在路径，export_folder 为指定数据存储路径，test_type 为功能选项，共有五种选择，如表 3- 1 所示：

表 3- 1 功能选项清单

test_type 取值	功能
0	订阅全市场证券数据并通过 ID 剔除部分证券（效率低，不推荐使用）
1	根据证券数据来源订阅行情数据,由行情源、证券类型、数据类型确定行情数据
2	根据证券 ID 来源订阅行情数据
3	查询证券的基础信息/最新状态
4	进行历史行情回测，单次回测的最大数据量限制请查阅用户使用手册
5	盘中回放当天数据，使用需要申请权限

开发指导

Windows 开发引导

- 创建工程

使用 visual studio 新建工程，将 build/include/mdc_gateway_client/protobuf 复制到工程目录下，并将该文件夹下所有文件添加进项目。

- 添加 INCLUDEPATH

在项目的属性-C/C++-常规-附加包含目录中添加：

protobuf;

build/include/protobuf-3.1.0/include;

build/include/mdc_gateway_client/include;

- 添加 LIBPATH

在项目的属性-链接器-常规-附加库目录中添加：

配置	平台	对应 build/libs/ 目录
Debug	Win32	Win32Debug
Debug	x64	WinX64Debug
Release	Win32	Win32Release
Release	x64	WinX64Release

- 添加 LIBS

请在项目的属性-链接器-输入-附加依赖项中添加库。

Debug 版本添加如下 5 个库：

mdc_gateway_clientd.lib

libprotobufd.lib

ACEd.lib

ACE_SSLd.lib

Release 版本添加如下 5 个库：

mdc_gateway_client.lib

libprotobuf.lib

ACE.lib

ACE_SSL.lib

- 添加动态链接库

确认程序的执行目录位置。以 function_test 为例，运行 functional_test.exe 时，functional_

test.exe 所在目录即为执行目录，利用 VS 本地调试时，VC++Project 文件所在路径即为执行目录。

根据前面的项目属性配置，从 build/libs 中拷贝对应版本的动态链接库 DLL 至程序的执行目录下。

Debug 版本添加如下 5 个库：

mdc_gateway_clientd.dll、libeay32.dll、ssleay32.dll、ACEd.dll、ACE_SSLd.dll

Release 版本添加如下 5 个库：

mdc_gateway_client.dll、libeay32.dll、ssleay32.dll、ACE.dll、ACE_SSL.dll

Linux 开发引导

Linux 开发需要修改 CMakeList.txt 的 INCLUDEPATH 和 LIBPATH 等配置，步骤如下：

- 将 include/mdc_gateway_client/protobuf 文件夹复制到工程目录下，并在 txt 中将该文件夹添加到源代码路径及包含路径中；
- 在代码目录下创建 cmake 目录；
- 进入 cmake 目录，运行 cmake ..；
- 运行 make 进行编译。

概况

API 使用限制

在使用 INSIGHT 提供的服务时，需要注意如下限制：

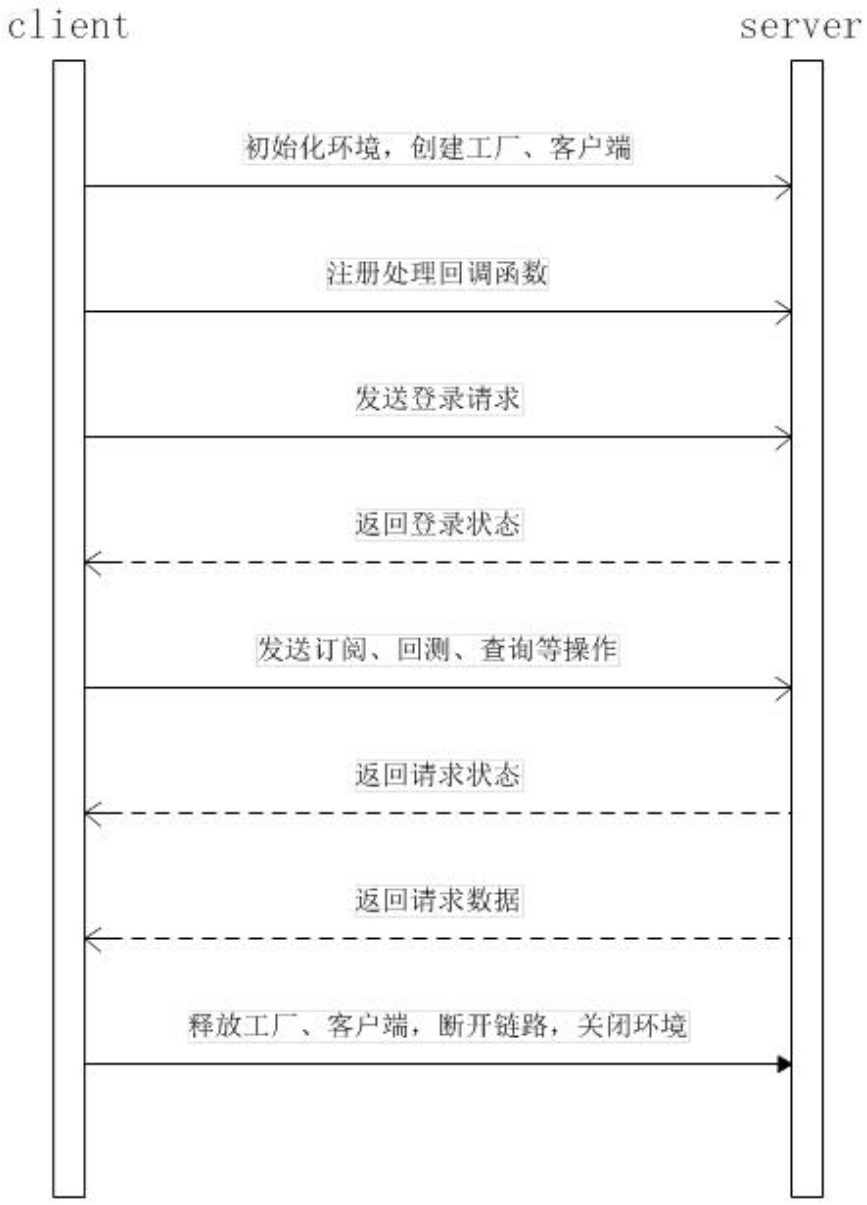
- 应用系统在需要接收实时行情时，需至少具备 15Mb 的专用互联网带宽或到华泰证券机房的专线带宽。若带宽低于此值，则可能造成实时行情会出现延迟、丢失等情况；
- 应用系统在需要进行行情回测时，应当在每日下午 17:00 至次日 7:00 之间进行。为保证交易时段实时行情的传输不受影响，平台会拒绝在上述时间以外的大规模 TICK 行情回测请求，具体使用限制请参阅 7.1；

数据模型说明

INSIGHT 以 Protobuf 定义所提供的行情数据，具体数据模型及字段含义请参阅《INSIGHT 数据字典》。

API 使用流程说明功能列表

工作流程说明



初始化

运行环境初始化

使用 SDK 时**必须先进行环境初始化**，在 main 函数中开始和结束处调用初始化和关闭环境的函数，用于 ACE 环境和日志初始化，示例代码如下：

初始化

```
int main(int argc, char* argv) {
        init_env();                             // 环境初始化
        ...
        fini_env();                             // 关闭环境
        return 0;
}
```

日志配置

SDK 提供控制日志信息的接口，可以配置不同的日志输出模式，接口及释义如下：

日志配置

```
void test_init() {
        open_trace();       // 打开流量日志
        open_file_log();        // 输出日志到文件
    open_heartbeat_trace();//打开 heartbeat_trace，输出心跳日志消息
    open_cout_log();   // 输出日志到控制台
        close_cout_log();       // 关闭流量日志
        close_trace();      // 关闭输出日志到文件
        close_file_log();       // 关闭输出日志到控制台
    open_node_auto();   //服务网关地址根据服务端配置选择
    close_node_auto();  //服务网关地址根据客户端配置选择
    }
```

回复消息处理配置

INSIGHT 提供两种方式处理服务端的回复消息，一种通过函数返回的形式，另一种通过回调函数的形式。通过 response_callback 开关来选择不同的处理方法。

当打开 response_callback 开关时，订阅请求回复、回测请求回复、回测控制请求回复、查询回复均通过相应的回调函数返回。当关闭 response_callback 开关时，程序会自动处理订阅请求回复、回测请求回复、回测控制请求回复，请求结果通过请求的返回值来获

得，查询回复通过参数引用的形式获得。response_callback 开关默认关闭，提供的接口及释义如下：

接口

```
open_response_callback ();                //打开 response_callback 开关，默认关闭
close_response_callback ();              //关闭 response_callback 开关
```

工厂、客户端的创建与释放

说明

创建 client 的函数需要注意 2 个参数：

- bool discovery_service_using_ssl: 外部用户必须设为 true，使用 ssl 加密；
- const char* folder： ssl 证书所在目录，discovery_service_using_ssl 为 true 时生效，不填表示当前运行目录；

接口

接口

```
// 工厂创建，单例
static ClientFactory* Instance();

// 工厂释放，如果生成过客户端，会一同释放
static void Uninstance();

/**
  * 创建客户端，全局唯一，如已创建会自动返回
  * @param[in] discovery_service_using_ssl 是否使用加密
  * @param[in] folder 证书所在的目录，其下具备证书（InsightClientCert.pem）和私钥（InsightClientKeyPkcs8.pem）
  * @return 客户端指针
  */
ClientInterface* CreateClient(bool discovery_service_using_ssl = false, const char* folder = "./cert");
```

示例

示例

```
void test_create_client() {
```

```
        bool ssl = true;
        ClientInterface* client = ClientFactory::Instance()->CreateClient(ssl);
        if (!client) {
                ClientFactory::Uninstance();
                return;
        }
        ClientFactory::Uninstance();
        return;
}
```

说明

注册回调对象后，客户端收到服务端发送的数据时调用用户自定义的实现进行处理，回调实现要求：

- 注册回调必须在客户端 Login 之前；
- 回调对象**必须支持多线程处理**，不要使用线程间共享数据，如需使用需要加锁；

接口

回调对象基类在 message_handle.h 中定义，需要继承此类自定义一个新的 handle 类，并实现如表 5- 1 所列函数：

表 5- 1 回调函数

函数名	功能
OnServiceMessage	发送订阅请求后服务端回复消息，查看是否订阅成功
OnMarketData	处理订阅后推送的实时行情数据
OnPlaybackPayload	处理回测返回的数据
OnPlaybackStatus	处理回测的状态
OnQueryResponse	处理查询请求返回结果
OnLoginSuccess	登录成功回调，本函数在登录成功后回调，为避免影响后续处理逻辑，请不要在此处进行复杂的业务逻辑
OnLoginFailed	登录失败回调，为避免影响后续处理逻辑，请不要在此处进行复杂的业务逻辑
OnNoConnections	处理所有服务器都无法连接的情况
OnReconnect	重连成功时回调，本函数在登录成功后回调，为避免影响后续处理逻

	辑，请不要在此处进行复杂的业务逻辑,建议的使用方式

MessageHandle 接口代码：

接口

```
/**
* 消息处理定义类
* 请根据需求实现下列虚函数，默认方式是不做任何处理
* 要求支持多线程并发，不要使用对象成员
*/
class LIB_EXPORT MessageHandle {
public:
        MessageHandle();
        virtual ~MessageHandle();

public:
        /**
        * 处理 InsightMessage
        * @param[in] header 消息头
        * @param[in] body 消息体
        */
        void ProcessMessage(
                const com::htsc::mdc::insight::model::MessageHeader* header,
                const com::htsc::mdc::insight::model::MessageBody* body);

        //=============================================================================
=========================
        // 以下为业务逻辑处理回调函数
        //=============================================================================
=========================

        /**
        * 处理 MDSecurityRecord
        * @param[in] MarketData 数据体
        */
        virtual void OnMarketData(const
com::htsc::mdc::insight::model::MarketData& data);

        /**
        * 处理回放消息
        * @param[in] PlaybackPayload 回放数据
```

```
        */
        virtual void OnPlaybackPayload(const
com::htsc::mdc::insight::model::PlaybackPayload& payload);

        /**
        * 处理回放状态消息
        * @param[in] PlaybackStatus 回放状态
        */
        virtual void OnPlaybackStatus(const
com::htsc::mdc::insight::model::PlaybackStatus& status);

        /**
        * 处理回放请求返回结果
        * @param[in] PlaybackResponse 回放请求结果
        */
        virtual void OnPlaybackResponse(const
com::htsc::mdc::insight::model::PlaybackResponse& response);

        /**
        * 处理回放控制请求返回结果
        * @param[in] PlaybackControlResponse 回放控制请求结果
        */
        virtual void OnPlaybackControlResponse(const
com::htsc::mdc::insight::model::PlaybackControlResponse& control_response);

        /**
        * 处理证券最新状态返回结果，订阅后返回
        * @param[in] MarketDataStream 状态数据
        */
        virtual void
OnServiceMessage(const ::com::htsc::mdc::insight::model::MarketDataStream&
data_stream);

        /**
        * 处理订阅请求返回结果
        * @param[in] MDSubscribeResponse 订阅请求结果
        */
        virtual void
OnSubscribeResponse(const ::com::htsc::mdc::insight::model::MDSubscribeRespons
e& response);
```

```
        /**
        * 处理查询请求返回结果
        * @param[in] MDQueryResponse 查询请求返回结果
        */
        virtual void
OnQueryResponse(const ::com::htsc::mdc::insight::model::MDQueryResponse&
response);

        /**
        * 处理告警消息
        * 为避免影响后续处理逻辑，请不要在此处进行复杂的业务逻辑
        * @param[in] context 错误内容
        */
        virtual void OnGeneralError(const
com::htsc::mdc::insight::model::InsightErrorContext& context);

        //=============================================================================
=========================
        // 以下为状态信息处理回调函数
        // 主要为影响业务逻辑的重连提供入口
        //=============================================================================
=========================
        /**
        * 登录成功回调，在重连成功时会触发
        * 本函数在登录成功后回调，为避免影响后续处理逻辑，请不要在此处进行复杂
的业务逻辑
        * @param[in] server_name 服务
        */
        virtual void OnLoginSuccess();

        /**
        * 登录失败，在重连时会多次触发
        * 本函数在登录失败后回调，为避免影响后续处理逻辑，请不要在此处进行复杂
的业务逻辑
        * 常见的处理方式是记录下登录错误信息，检查登录验证信息正确性后，在外部
关闭客户端并进行重连
        * @param[in] error_no 错误号
        * @param[in] message 错误消息
        */
        virtual void OnLoginFailed(int error_no, const std::string& message);
```

```
        /**
        * 在重连失败时回调，表示所有服务器都无法连接
        * 仅通知，不要在此处对 Mdc 客户端对象的进行处理
        * 建议的使用方式：在进行回放操作时记录 OnReconnect 的发生时间，如果在回访操作进行时，则本次回放失败。
        */
        virtual void OnNoConnections();

        /**
        * 重连成功时，为避免影响后续处理逻辑，请不要在此处进行复杂的业务逻辑处理
        * 建议的使用方式：在进行回放操作时记录 OnReconnect 发生时间，如果此时正在回放，则回放应失败（回放不支持续传）。
        */
        virtual void OnReconnect();

};
```

示例

注册示例如下，其中自定义 handle 类取名为 NewHandle：

接口

```
void test_regist_handle() {
        bool ssl = true;
        ClientInterface* client = ClientFactory::Instance()->CreateClient(ssl);
        if (!client) {
                return;
        }

        //ip,port,user,password…
        NewHandle* handle = new NewHandle();
        if (!handle) {
                ClientFactory::Uninstance();
                return;
        }

        //注册句柄
        client->RegistHandle(handle);
        //登录服务发现网关…
        //work…
```

```
        ClientFactory::Uninstance();;
        delete handle;
        return;
}
```

- 实时行情推送接口 OnMarketData

OnMarketData 回调接口可在程序请求订阅行情后抛出行情数据，应用系统可在其中添加行情的处理逻辑。

接口

```
/**
* 处理订阅后推送的实时行情数据
* @param[in] data
*/
void OnMarketData(const com::htsc::mdc::insight::model::MarketData& data) {
        static unsigned int count = 0;
        ++count;
        if (count % 10000 == 0) {
                debug_print("==========> NewHandle: process %d MarketData
message", count);
        }
        //获取数据类型
        std::string data_type = get_data_type_name(data.marketdatatype());
        switch (data.marketdatatype()) {
        case MD_TICK:
        {//快照
                if (data.has_mdstock()) {//股票
                        //获取证券类型
                        std::string security_type =
get_security_type_name(data.mdstock().securitytype());
                        //写文件
                        save_debug_string(base_forder_, data_type,
security_type,
                                        data.mdstock().htscsecurityid(),
data.mdstock().ShortDebugString());
                        //访问股票数据部分字段
                        if (false) {
                                MDStock stock = data.mdstock();
                                debug_print("HTSCSecurityID : %s",
stock.htscsecurityid().c_str());
```

```
                                debug_print("MDDate : %d", stock.mddate());
                                //访问队列(queue)数据方式
                                for (int i = 0; i < stock.buypricequeue_size();
i++) {
                                        debug_print("Buy%dPrice : %d", i + 1,
stock.buypricequeue(i));
                                }
                        }
                } else if (data.has_mdbond()) {//债券
                        std::string security_type =
get_security_type_name(data.mdbond().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.mdbond().htscsecurityid(),
data.mdbond().ShortDebugString());
                } else if (data.has_mdindex()) {//指数
                        std::string security_type =
get_security_type_name(data.mdindex().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.mdindex().htscsecurityid(),
data.mdindex().ShortDebugString());
                } else if (data.has_mdfund()) {//基金
                        std::string security_type =
get_security_type_name(data.mdfund().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.mdfund().htscsecurityid(),
data.mdfund().ShortDebugString());
                } else if (data.has_mdoption()) {//期权
                        std::string security_type =
get_security_type_name(data.mdoption().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.mdoption().htscsecurityid(),
data.mdoption().ShortDebugString());
                } else if (data.has_mdfuture()) {//期货
                        std::string security_type =
get_security_type_name(data.mdfuture().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
```

```
                                data.mdfuture().htscsecurityid(),
data.mdfuture().ShortDebugString());
                } else if (data.has_mdforex()) {//外汇
                        std::string security_type =
get_security_type_name(data.mdforex().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.mdforex().htscsecurityid(),
data.mdforex().ShortDebugString());
                } else if (data.has_mdspot()) {//现货
                        std::string security_type =
get_security_type_name(data.mdspot().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.mdspot().htscsecurityid(),
data.mdspot().ShortDebugString());
                } else if (data.has_mdwarrant()) {//权证
                        std::string security_type =
get_security_type_name(data.mdwarrant().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.mdwarrant().htscsecurityid(),
data.mdwarrant().ShortDebugString());
                }
                break;
        }
        case MD_TRANSACTION:
        {//逐笔成交
                if (data.has_mdtransaction()) {
                        std::string security_type =
get_security_type_name(data.mdtransaction().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.mdtransaction().htscsecurityid(),
data.mdtransaction().ShortDebugString());
                }
                break;
        }
        case MD_ORDER:
        {//逐笔委托
                if (data.has_mdorder()) {
```

```
                        std::string security_type =
get_security_type_name(data.mdorder().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.mdorder().htscsecurityid(),
data.mdorder().ShortDebugString());
                }
                break;
        }
        case MD_CONSTANT:
        {//静态信息
                if (data.has_mdconstant()) {
                        std::string security_type =
get_security_type_name(data.mdconstant().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.mdconstant().htscsecurityid(),
data.mdconstant().ShortDebugString());
                }
        }
        case MD_KLINE_15S:
        case MD_KLINE_1MIN:
        case MD_KLINE_5MIN:
        case MD_KLINE_15MIN:
        case MD_KLINE_30MIN:
        case MD_KLINE_60MIN:
        case MD_KLINE_1D:
        {//实时数据只提供 15S 和 1MIN K 线
                if (data.has_mdkline()) {
                        std::string security_type =
get_security_type_name(data.mdkline().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.mdkline().htscsecurityid(),
data.mdkline().ShortDebugString());
                }
                break;
        }
        case MD_TWAP_1MIN:
        {//TWAP 数据
                if (data.has_mdtwap()) {
```

```
                        std::string security_type =
get_security_type_name(data.mdtwap().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.mdtwap().htscsecurityid(),
data.mdtwap().ShortDebugString());
                }
                break;
        }
        case MD_VWAP_1MIN:
        case MD_VWAP_1S:
        {//VWAP 数据
                if (data.has_mdvwap()) {
                        std::string security_type =
get_security_type_name(data.mdvwap().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.mdvwap().htscsecurityid(),
data.mdvwap().ShortDebugString());
                }
                break;
        }
        case AD_FUND_FLOW_ANALYSIS:
        {//资金流向分析数据
                if (data.has_mdfundflowanalysis()) {
                        std::string security_type =
get_security_type_name(data.mdfundflowanalysis().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.mdfundflowanalysis().htscsecurityid(),
data.mdfundflowanalysis().ShortDebugString());
                }
                break;
        }
        case MD_ETF_BASICINFO:
        {//ETF 的基础信息
                if (data.has_mdetfbasicinfo()) {
                        std::string security_type =
get_security_type_name(data.mdetfbasicinfo().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
```

```
                                data.mdetfbasicinfo().htscsecurityid(),
data.mdetfbasicinfo().ShortDebugString());
                }
        }
        case AD_ORDERBOOK_SNAPSHOT:
        {
                if (data.has_orderbooksnapshot()) {
                        std::string security_type =
get_security_type_name(data.orderbooksnapshot().securitytype());
                        save_debug_string(base_forder_, data_type,
security_type,
                                data.orderbooksnapshot().htscsecurityid(),
data.orderbooksnapshot().ShortDebugString());
                }
        }
        default:
                break;
        }
}
```

- 回测数据推送接口 OnPlaybackPayload

OnPlaybackPayload 接口处理回测消息，OnPlaybackStatus 接口处理回测状态消息。

接口

```
/**
* 处理回测请求成功后推送的回测数据,多回放任务并发时存在线程冲突风险
* @param[in] PlaybackPayload 回测数据
*/
void OnPlaybackPayload(const PlaybackPayload& payload) {
        debug_print("------- PARSE message Playback payload, id:%s",
payload.taskid().c_str());
        const MarketDataStream& stream = payload.marketdatastream();
        debug_print("total number=%d, serial=%d, isfinish=%d",
stream.totalnumber(), stream.serial(), stream.isfinished());
        google::protobuf::RepeatedPtrField<MarketData>::const_iterator it
                = stream.marketdatalist().marketdatas().begin();
        while (it != stream.marketdatalist().marketdatas().end()) {
                OnMarketData(*it);
                ++it;
        }
```

```
}
/**
* 处理回测状态
* @param[in] PlaybackStatus 回测状态
*/
void OnPlaybackStatus(const com::htsc::mdc::insight::model::PlaybackStatus&
status) {
        //INITIALIZING = 11            // 初始化中
        //PREPARING = 12               // 准备中
        //PREPARED = 13                        // 准备完成
        //RUNNING = 14                         // 运行中
        //APPENDING = 15               // 队列等待中
        //CANCELED = 16                // 已取消
        //COMPLETED = 17               // 已完成
        //FAILED = 18                  // 任务失败
        service_value_ = status.taskstatus();
        int task_status = status.taskstatus();
        if (task_status == CANCELED || task_status == COMPLETED || task_status
== FAILED)
        {
                task_id_mutex_.lock();
                task_id_status_.erase(status.taskid());
                task_id_mutex_.unlock();
        }
}
```

- 查询数据推送接口 OnQueryResponse

OnQueryResponse 处理服务端返回查询请求返回结果，仅当程序初始化时打开 response_callback 开关时，该回调函数有效。

接口

```
/**
* 处理查询请求返回结果
* @param[in] MDQueryResponse 查询请求返回结果
*/
void OnQueryResponse(const ::com::htsc::mdc::insight::model::MDQueryResponse&
response) {
        if (!response.issuccess()) {
                error_print("query response result: FAIL! ERROR INFO[%d,%s]",
```

```
                        response.errorcontext().errorcode(),
response.errorcontext().message().c_str());
                query_exit_ = true;
                return;
        } else {
                const MarketDataStream& stream = response.marketdatastream();
                debug_print("query response total number=%d, serial=%d,
isfinish=%d",
                        stream.totalnumber(), stream.serial(),
stream.isfinished());
                if (stream.isfinished() == 1) {
                        query_exit_ = true;
                }
                google::protobuf::RepeatedPtrField<MarketData>::const_iterator
it
                        = stream.marketdatalist().marketdatas().begin();
                while (it != stream.marketdatalist().marketdatas().end()) {
                        OnMarketData(*it);
                        ++it;
                }
        }
}
```

用户登录

说明

登录认证过程：

- 将 pem 和 InsightClientKeyPkcs8.pem 文件放到可执行程序所在目录；
- 创建客户端时需要使用 ssl 加密(discovery_service_using_ssl 为 true)，并填写认证文件所在的目录；

接口

```
client = ClientFactory::Instance()->CreateClient(true, cert_folder.c_str());
```

- 外网地址和开放端口请联系华泰证券相关工作人员获取。

接口

用户使用提供的服务网关地址和端口进行登录。当所提供的服务网关地址和端口登录失败后，API 内部会自动使用备选的服务网关地址和端口列表进行登录。接口在 client_interface.h 中定义：

接口

```
 /**
   * 使用服务发现方式，通过网关获取服务列表进行链接，按照增加的先后顺序连接
   * @param[in] ip 服务网关地址
   * @param[in] port 服务网关端口
   * @param[in] user 用户名
   * @param[in] value 登录值（密码或 token)
   * @param[in] password_is_token 是否为 token
   * @param[in] backup_list 服务网关备选地址列表
   * @return 0 成功 < 0 失败，错误信息见错误号
   */
virtual int LoginByServiceDiscovery(const std::string& ip, int port,
       const std::string& user, const std::string& value,
bool password_is_token, std::map<std::string, int>backup_list) = 0;
```

示例

接口

```
bool test_login() {
        client = ClientFactory::Instance()->CreateClient(true,
cert_folder.c_str());
        if (!client) {
                error_print("create client failed!");
                ClientFactory::Uninstance();
                client = NULL;
                return false;
        }
        //注册句柄
        handle = new InsightHandle(export_folder);
        client->set_handle_pool_thread_count(10);
        client->RegistHandle(handle);
        //添加备用发现网关地址
        std::vector<std::string> backup_list;
        backup_list.push_back("221.6.24.39:8262");
        backup_list.push_back("112.4.154.165:8262");
        backup_list.push_back("221.131.138.171:9242");
        //登录
```

```
        int ret = client->LoginByServiceDiscovery(ip, port, user, password,
false, backup_list);
        if (ret != 0) {
                error_print("%s", get_error_code_value(ret).c_str());
                ClientFactory::Uninstance();
                client = NULL;
                DELETE_P(handle);
                return false;
        }
        return true;
}
```

订阅

说明

使用 SDK 订阅证券时首先需要向服务器发送订阅请求，订阅请求接口函数包括两个参数，第一个参数是订阅操作类型：ESubscribeActionType，该数据类型为枚举值，如表 5- 2 所示：

表 5- 2 订阅操作枚举值含义

枚举名称	枚举值	意义
COVERAGE	0	覆盖
ADD	1	增加
DECREASE	2	减少
CANCEL	3	取消

第二个参数是订阅方式，提供三种订阅方式，如表 5- 3 所示：

表 5- 3 订阅方式

订阅方式类名称	意义
SubscribeAll	订阅全市场证券

SubscribeBySourceType	按照市场类型订阅
SubscribeByID	按照证券 ID 订阅

在发出订阅请求时，选择适合的订阅方式，调用对应订阅方式的接口函数，将上述两个参数填入接口函数中即可向服务器发送订阅请求。订阅成功后开市前闭市后的时间段内，每隔 30s 会把订阅对象的最新数据推送出来。

由于交易所转发权限限制，目前可以订阅的证券类型 ESecurityType 如表 5- 4 所示：

表 5- 4 证券类型

证券类型名称	意义
StockType	股票
IndexType	指数
BondType	债券
FundType	基金
OptionType	期权

其中期权只有 level1 数据。

更多数据类型 EMarketDataType 和证券类型 ESecurityType 的定义请参阅《华泰证券行情 INSIGHT 数据模型使用说明》。

全市场订阅说明

根据全市场标的订阅时，需调用 ClientInterface 的 SubscribeAll 接口和 SubscribeAll 类，具体示例代码参见 functional_test/main.cc 的 test_subscribe_by_all_and_decrease 函数。

- 接口

接口

```
/**
* 订阅全市场数据，本函数禁止多线程并发调用
* @input[in] action_type 操作类型
* @input[in] data_types 数据类型列表，由调用方释放
* @return 0 成功 < 0 错误号
```

```
*/
virtual int SubscribeAll(
        ::com::htsc::mdc::insight::model::ESubscribeActionType action_type,
        ::com::htsc::mdc::insight::model::SubscribeAll* subscribe_all) = 0;
```

- 示例

接口

```
/**
* 订阅全市场证券数据并通过 ID 剔除部分证券
* 效率低，不推荐使用
*/
void test_subscribe_by_all_and_decrease() {
        if (client == NULL || handle == NULL) {
                error_print("create client failed!");
                return;
        }
        //订阅 SubscribeAll
        ESubscribeActionType action_type = COVERAGE;
        std::unique_ptr<SubscribeAll> subscribe_all(new SubscribeAll());
        subscribe_all->add_marketdatatypes(MD_TICK);
        subscribe_all->add_marketdatatypes(MD_TRANSACTION);
        subscribe_all->add_marketdatatypes(MD_ORDER);
        if ((client->SubscribeAll(action_type, &(*subscribe_all)) < 0)) {
                return;
        }
        //通过 DECREASE 减少订阅
        action_type = DECREASE;
        std::unique_ptr<SubscribeByID> subscribe_decrease(new SubscribeByID());
        SubscribeByIDDetail* id_detail = subscribe_decrease-
>add_subscribebyiddetails();
        id_detail->set_htscsecurityid("000002.SZ");
        id_detail->add_marketdatatypes(MD_TICK);
        //订阅
        int ret = client->SubscribeByID(action_type, &(*subscribe_decrease));
        if (ret != 0) {
                error_print("%s", get_error_code_value(ret).c_str());
                return;
        }
        std::cout << "input any key to quit...>>";
        char a[4096];
```

```
        std::cin >> a;
        return;
}
```

根据证券类型订阅说明

根据证券类型订阅时，调用 ClientInterface 的 SubscribeBySourceType 接口和 SubscribeBy-

SourceType 类，具体示例代码参见 functional_test/main.cc 的 test_subscribe_by_source_type 函数。

- 接口

接口

```
/**
* 根据证券类型订阅，本函数禁止多线程并发调用
* @input[in] action_type 操作类型
* @input[in] type_details 类型列表，由调用方释放
* @return 0 成功 < 0 错误号
*/
virtual int SubscribeBySourceType(
        ::com::htsc::mdc::insight::model::ESubscribeActionType action_type,
        ::com::htsc::mdc::insight::model::SubscribeBySourceType* source_type)
= 0;
```

- 示例

接口

```
/**
* 根据证券数据来源订阅行情数据,由三部分确定行情数据
* 行情源(SecurityIdSource):XSHG(沪市)|XSHE(深市)|...，不填默认全选
* 证券类型
(SecurityType):BondType(债)|StockType(股)|FundType(基)|IndexType(指)|OptionTyp
e(期权)|...
* 数据类型(MarketDataTypes):MD_TICK(快照)|MD_TRANSACTION(逐笔成交)|MD_ORDER(逐
笔委托)|...
*/
void test_subscribe_by_source_type() {
        if (client == NULL || handle == NULL) {
                error_print("create client failed!");
                return;
```

```
        }
        //订阅 SubscribeBySourceType
        ESubscribeActionType action_type = COVERAGE;
        std::unique_ptr<SubscribeBySourceType> source_type(new
SubscribeBySourceType());

        //////债券
        //SubscribeBySourceTypeDetail* detail_1 = source_type-
>add_subscribebysourcetypedetail();
        //SecuritySourceType* security_source_type_1 = new
SecuritySourceType();
        //security_source_type_1->set_securityidsource(XSHG);
        //security_source_type_1->set_securitytype(BondType);
        //detail_1->set_allocated_securitysourcetypes(security_source_type_1);
        //detail_1->add_marketdatatypes(MD_TICK);

    ......

        //订阅
        int ret = client->SubscribeBySourceType(action_type, &(*source_type));
        if (ret != 0) {
                error_print("%s", get_error_code_value(ret).c_str());
                return;
        }
        std::cout << "input any key to quit...>>";
        char a[4096];
        std::cin >> a;
        return;
}
```

根据证券编号订阅说明

根据证券编号进行订阅时，调用 ClientInterface 的 SubscribeByID 接口和 SubscribeByID 类，具体示例代码参见 functional_test/main.cc 的 test_subscribe_by_id 函数。

- 接口

接口

```
/**
* 根据证券编号订阅，本函数禁止多线程并发调用
```

```
* @input[in] action_type 操作类型
* @input[in] id 订阅 id 指针，由调用方释放
* @return 0 成功 < 0 错误号
*/
virtual int SubscribeByID(
        ::com::htsc::mdc::insight::model::ESubscribeActionType action_type,
        ::com::htsc::mdc::insight::model::SubscribeByID* id) = 0;
```

- 示例

接口

```
/**
* 根据证券 ID 来源订阅行情数据
*/
void test_subscribe_by_id() {
        if (client == NULL || handle == NULL) {
                error_print("create client failed!");
                return;
        }
        //订阅 SubscribeByID
        ESubscribeActionType action_type = COVERAGE;
        std::unique_ptr<SubscribeByID> id(new SubscribeByID());
        //list1 订阅 MD_TRANSACTION、MD_ORDER
        std::vector<std::string> securityIdList1;
        securityIdList1.clear();
        securityIdList1.push_back("A2111.DCE");
        //securityIdList1.push_back("000014.SZ");
        //securityIdList1.push_back("000015.SZ");
        //securityIdList1.push_back("000016.SZ");
        for (unsigned int index = 0; index < securityIdList1.size(); index++)
{
                SubscribeByIDDetail* id_detail = id-
>add_subscribebyiddetails();
                id_detail->set_htscsecurityid(securityIdList1[index]);
                id_detail->add_marketdatatypes(MD_TICK);
                id_detail->add_marketdatatypes(MD_TRANSACTION);
        }
        //list2 订阅 MD_ORDER
        //std::vector<std::string> securityIdList2;
        //securityIdList2.clear();
        //securityIdList2.push_back("000017.SZ");
```

```
        //securityIdList2.push_back("000018.SZ");
        //securityIdList2.push_back("000019.SZ");
        //securityIdList2.push_back("000020.SZ");
        //for (unsigned int index = 0; index < securityIdList2.size(); index++)
{
        //      SubscribeByIDDetail* id_detail = id-
>add_subscribebyiddetails();
        //      id_detail->set_htscsecurityid(securityIdList2[index]);
        //      id_detail->add_marketdatatypes(MD_TICK);
        //      id_detail->add_marketdatatypes(MD_ORDER);
        //}
        //add_globalmarketdatatypes 会覆盖所有的数据类型设置
        //id->add_globalmarketdatatypes(MD_TRANSACTION)

        //订阅
        int ret = client->SubscribeByID(action_type, &(*id));
        if (ret != 0) {
                error_print("%s", get_error_code_value(ret).c_str());
                return;
        }
        std::cout << "input any key to quit...>>";
        char a[4096];
        std::cin >> a;
        return;
}
```

回测

说明

使用 SDK 进行行情回测时，需要向服务器发送回测请求 PlaybackRequest，回测请求需要配置的参数如表 5- 5 所示：

表 5- 5 回测请求参数

回测请求参数	意义
taskId	回测任务编号，必须填写，提供 get_task_id 函数自动生成定制化的 taskId，taskId 命名的具体规则见注 1
htscSecurityIDs	需要回测的证券 id，可以多条，具体数量限制见注 2

replayDataType	需要回测的数据类型(MD_TICK...)，只能选择一种数据类型
startTime	回测数据的起始时间注3
stopTime	回测数据的结束时间注3
exrightsType	除复权类型，具体可设置的复权类型见注4

注 1：taskId 命名规则为：节点名+pid+当前时间（精确到微秒）+自定义内容后缀，如：hostname_13323_20170728T1500.0132_suffix

获取函数为 base_define.h 下的 std::string get_task_id(const std::string& suffix);

示例：

```
1  PlaybackRequest* request = new PlaybackRequest();

2  request->set_taskid(get_task_id (""));
```

注 2：回测功能在开盘和盘后有如下限制：

- 盘后回测：由股票支数和天数的乘积决定，股票支数×回测天数≤450，tick/transaction/order 数据回测天数上限为 30 天。所以回测范围为 15 只股票 30 天以内到 450 支股票一天以内 30 支股票 15 天以内，回测一天最多 450 支。日 K 线最多回测一年，每支证券权重为 005。分钟 K 线最多回测三个月，每支证券权重 0.05.证券支数×证券权重≤450；
- 开盘期间：处理能力为盘后的一半；

注 3：日 K 线回测时，起止时间为 00:00:00-23:59:59；

其余情况，回测数据的起止时间为(starttime,stoptime]。

注 4：可设置的复权类型如表 5- 6 所示：

表 5- 6 复权类型

复权类型	**意义**
NO_EXRIGHTS	不复权
FORWARD_EXRIGHTS	向前复权
BACKWARD_EXRIGHTS	向后复权

根据证券 ID 进行行情回测

- 接口

接口

```
/**
* 请求回放，本函数禁止多线程并发调用
* @param[in] request 已分配好空间的回放请求，由调用方释放
* @return 0 成功 < 0 错误号
*/
virtual int RequestPlayback(::com::htsc::mdc::insight::model::PlaybackRequest*
request) = 0;
```

- 示例

接口

```
/**
* 进行历史行情回测，单次回测的最大数据量限制请查阅用户使用手册
*/
void test_playback() {
        if (client == NULL || handle == NULL) {
                error_print("create client failed!");
                return;
        }
        //request 需要提前分配空间，并由调用方释放
        std::vector<std::string> securityIdList1;
        securityIdList1.clear();
        securityIdList1.push_back("601688.SH");
        securityIdList1.push_back("000014.SZ");
        securityIdList1.push_back("000016.SZ");
        securityIdList1.push_back("000001.SZ");
        securityIdList1.push_back("000002.SZ");
        securityIdList1.push_back("000004.SZ");
        securityIdList1.push_back("000005.SZ");
        securityIdList1.push_back("000006.SZ");
        for (size_t i = 0; i < securityIdList1.size(); i++)
        {
                std::unique_ptr<PlaybackRequest> request(new
PlaybackRequest());
                //设置回放任务号，必须设置
                std::string task_id = get_task_id("");
```

```
                request->set_taskid(task_id);
                handle->task_id_mutex_.lock();
                handle->task_id_status_.insert(std::pair<std::string,
int>(task_id, 0));
                handle->task_id_mutex_.unlock();
                //=================================================
                // 可配置的参数包括证券 id 和证券市场，
                //=================================================
                //填写证券 id
                request->add_htscsecurityids(securityIdList1[i]);

                //设置回放数据类型，只能配置一个
                request->set_replaydatatype(MD_TICK);
                //设置复权类型，不设置默认不做复权处理
                request->set_exrightstype(NO_EXRIGHTS);
                //设置回放的开始结束时间
                request->set_starttime("20200312090000");
                request->set_stoptime("20200312150000");
                //同时在线的回放任务不能超过五个
                while (handle->task_id_status_.size() >= 5)
                {
                        msleep(1000);
                }
                //发送回放请求
                int ret = client->RequestPlayback(&(*request));
                if (ret != 0)
                {
                        error_print("%s", get_error_code_value(ret).c_str());
                        return;
                }
        }
        //等待所有回放任务结束
        while (handle->task_id_status_.size() > 0)
        {
                msleep(1000);
        }
        return;
}
```

查询证券信息

说明

利用 SDK 进行证券信息查询时，需要向服务器发送查询请求 MDQueryRequest，查询请求需要配置的参数如表 5- 7 所示：

表 5- 7 查询请求参数说明

查询请求参数	意义
queryType	本次查询请求的查询类型，具体类型见注 1
htscSecurityIDs	需要查询的证券 id，可以多条
securityIdSource	需要查询的行情源(XSHG...)
securityType	需要查询的证券类型(StockType...)

注 1：查询请求类型定义如下：

查询类型定义

```
QUERY_TYPE_BASE_INFORMATION                    // 查询历史上所有的指定证券的基础信息(MarketDataTypes = MD_CONSTANT)
QUERY_TYPE_LATEST_BASE_INFORMATION    // 查询今日最新的指定证券的基础信息(MarketDataTypes = MD_CONSTANT)
QUERY_TYPE_STATUS                                    // 查询指定证券的最新一条 Tick 数据(MarketDataTypes = MD_TICK)
QUERY_TYPE_ETF_BASE_INFORMATION    //查询 ETF 的基础信息(MarketDataTypes = MD_ETF_BASICINFO)
```

接口

提供两种查询请求接口，一种是接口是关闭 response_callback 开关时，查询回复消息通过查询请求接口中的参数返回，另一种接口是打开 response_callback 开关时，查询回复消息通过查询回复回调函数返回。

查询接口

```
/**
* 行情查询请求，本函数禁止多线程并发调用，回复数据通过引用获得，需要关闭 response_callback 开关
* @param[in] request 已分配好空间的查询请求，由调用方释放
* @param[out] responses 内部分配空间，由调用方调用 ReleaseMdQueryResponses 释放
* @return 0 成功 < 0 错误号
*/
```

```
virtual int RequestMDQuery(::com::htsc::mdc::insight::model::MDQueryRequest*
request,
        std::vector< ::com::htsc::mdc::insight::model::MDQueryResponse* >*&
mdresponses) = 0;

/**
* 行情查询请求，本函数禁止多线程并发调用，回复数据通过回调接口获得，需要打开
response_callback 开关
* @param[in] request 已分配好空间的查询请求，由调用方释放
* @return 0 成功 < 0 错误号
*/
virtual int RequestMDQuery(::com::htsc::mdc::insight::model::MDQueryRequest*
request) = 0;

/**
* 释放行情查询结果
* @param[in] mdresponses 查询结果
*/
virtual void ReleaseQueryResult(
        std::vector< ::com::htsc::mdc::insight::model::MDQueryResponse* >*&
mdresponses);
```

示例

查询回复消息通过查询请求接口中的参数返回示例，本示例仅当在 response_callback 开关关闭时使用。

查询示例

```
/**
* 查询证券的基础信息/最新状态
* 需要关闭 response_callback，response 消息通过查询接口参数返回
*/
void test_query() {
        if (client == NULL || handle == NULL) {
                error_print("create client failed!");
                return;
        }
        //配置查询请求
        std::unique_ptr<MDQueryRequest> request(new MDQueryRequest());
        //设置查询请求类型
        /*
```

```
        QUERY_TYPE_BASE_INFORMATION                    // 查询历史上所有的指定
证券的基础信息(MarketDataTypes = MD_CONSTANT)
        QUERY_TYPE_LATEST_BASE_INFORMATION    // 查询今日最新的指定证券的基础
信息(MarketDataTypes = MD_CONSTANT)
        QUERY_TYPE_STATUS                                        // 查询指定证券
的最新一条 Tick 数据(MarketDataTypes = MD_TICK)
        QUERY_TYPE_ETF_BASE_INFORMATION    //查询 ETF 的基础信息
(MarketDataTypes = MD_ETF_BASICINFO)
        */
        request->set_querytype(QUERY_TYPE_LATEST_BASE_INFORMATION);
        SecuritySourceType* security_source_type = request-
>add_securitysourcetype();
        security_source_type->set_securityidsource(XSHE);
        security_source_type->set_securitytype(IndexType);
        request->add_htscsecurityids("601688.SH");

        //接收查询回复
        std::vector<MDQueryResponse*>* responses;
        //发送查询请求并接收回复消息
        int ret = client->RequestMDQuery(&(*request), responses);
        if (ret != 0) {
                error_print("%s", get_error_code_value(ret).c_str());
                return;
        }

        //访问 responses，成员如下：
        /*
        int32 queryType;                    //查询类型
        bool isSuccess;                    // 请求是否成功
        InsightErrorContext errorContext;  // 错误信息
        MarketDataStream marketDataStream; //行情数据包
        */
        debug_print("query return message count=%d\n\n\n", responses->size());
        //处理查询结果
        for (unsigned int i = 0; i < responses->size(); ++i) {
                if (!responses->at(i)->issuccess()) {
                        continue;
                }
                if (!responses->at(i)->has_marketdatastream()) {
                        continue;
                }
```

```
                //遍历 constants
                google::protobuf::RepeatedPtrField<MarketData>::const_iterator
it
                        = responses->at(i)-
>marketdatastream().marketdatalist().marketdatas().begin();
                google::protobuf::RepeatedPtrField<MarketData>::const_iterator
end
                        = responses->at(i)-
>marketdatastream().marketdatalist().marketdatas().end();
                while (it != end) {
                        handle->OnMarketData(*it);
                        it++;
                }
        }
        //释放空间
        client->ReleaseQueryResult(responses);
        return;
}
```

查询回复消息通过回调函数返回示例，本示例仅当在 response_callback 开关打开时使用

接口

```
/**
* 查询证券的基础信息/最新状态
* 需要打开 response_callback，response 消息通过回调函数返回
*/
void test_query_callback() {
        if (client == NULL || handle == NULL) {
                error_print("create client failed!");
                return;
        }
        //配置查询请求
        std::unique_ptr<MDQueryRequest> request(new MDQueryRequest());
        //设置查询请求类型
        /*
        QUERY_TYPE_BASE_INFORMATION                     // 查询历史上所有的指定
证券的基础信息(MarketDataTypes = MD_CONSTANT)
        QUERY_TYPE_LATEST_BASE_INFORMATION    // 查询今日最新的指定证券的基础
信息(MarketDataTypes = MD_CONSTANT)
```

```
        QUERY_TYPE_STATUS                                       // 查询指定证券
的最新一条 Tick 数据(MarketDataTypes = MD_TICK)
        QUERY_TYPE_ETF_BASE_INFORMATION    //查询 ETF 的基础信息
(MarketDataTypes = MD_ETF_BASICINFO)
        */
        //查询 601688.SH + 深交所全市场股票的静态信息
        request->set_querytype(QUERY_TYPE_LATEST_BASE_INFORMATION);
        request->add_htscsecurityids("510300.SH");
        SecuritySourceType* source_type1 = request->add_securitysourcetype();
        source_type1->set_securitytype(StockType);
        source_type1->set_securityidsource(XSHE);
        SecuritySourceType* source_type2 = request->add_securitysourcetype();
        source_type2->set_securitytype(StockType);
        source_type2->set_securityidsource(XSHG);

        handle->query_exit_ = false;
        //发送查询请求
        int ret = client->RequestMDQuery(&(*request));
        if (ret != 0) {
                error_print("%s", get_error_code_value(ret).c_str());
                return;
        }

        while (true) {
                if (handle->query_exit_) {
                        break;
                }
                else{
                        msleep(1);
                }
        }
}
```

其它

错误号含义获取接口

- 说明

用于解析错误号，返回其含义，如-1000 表示“连接服务器失败，网络不通”。

- 接口

1	std::string get_error_code_value(int code);

设置并发处理线程数

- 说明

当消息的处理逻辑比较复杂，需要更多线程并发处理时，设置处理线程池线程数。必须在调用 LoginByServiceDiscovery 登录服务端之前设置。

- 接口

接口

```
/**
  * 设置处理线程池线程数，在登录之前调用
  * @param[in] count 线程数
  */
virtual void set_handle_pool_thread_count(short count) = 0;

/**
  * 获得处理线程数
  * @return 线程数
  */
virtual short handle_pool_thread_count() const = 0;

//默认值为 5 个处理线程，如果观测程序的 cpu 利用率长期占用超过 300%，且服务器性能足够的，可以配置更多的处理线程。

//处理线程池初始线程数
const int THREAD_POOL_INIT_SIZE = 5;
//处理线程池最大线程数
const int THREAD_POOL_MAX_SIZE = 100;
```

常见日志信息解释

登录日志

- set private_key[../../cert/InsightClientKeyPkcs8.pem] result=0

set certificate[../../cert/InsightClientCert.pem] result=0

解释：密钥认证信息，result=0 成功，否则失败

Tips：失败时主要检查 pem 文件的目录是否正确。

- === try to LoginByServiceDiscovery [1/10] ... ===

begin to login discovery service! ip="289.61.2.39" ,port=9072

ssl connect to server [289.61.2.39:9072] failed! err=10047(Unknown error)

解释：ssl 认证失败

Tips：检查 ip 和 port 是否正确，若均正确且仍登录失败请联系 INSIGHT 官方人员。

- === try to LoginByServiceDiscovery [1/10] ... ===

begin to login discovery service! ip="221.226.112.140" ,port=9242

server reply: login failed, code=400,User authentication failure.

解释：登录时用户认证失败，User authentication failure。

Tips：检查用户名密码是否正确，若均正确且仍登录失败请联系 INSIGHT 官方人员。

- stream recv size=0, recv[0] total[8], system error(10035, Unknown error)!!!

receive message header failed! ret=-1004

=== try to relogin [1/10] ... ===

begin to login data server by TOKEN!

解释：TCP 连接失败，SDK 进行重连

Tips：会有两种情况导致重连：(1)网络中断 (2)网络拥塞后恢复正常

出现重连时先确认网络情况，若 SDK 多次重连失败且网络正常时，请联系 INSIGHT 官方人员。

open_trace()日志信息

由 open_trace()、close_trace()函数控制该部分日志的打开与关闭，这部分的日志信息均 10s 打印一条。

- === work pool thread[0, 156528] tps[153.233221] processed[2000] total[8000] queue size[0]....

解释：work pool 中编号为 0 的线程 pid=156528，线程吞吐量为 153.233221，在 10 秒内处理 2000 条数据，总处理 8000 调数据，当前未处理数据数量为 0

Tips：只需关注 queue size 的大小，如果 queue size 过大，说明回调函数中处理速度过慢，建议大小维持在 100 以内。

- traffic log 信息

```
                                                    ===============
traffic log ===============

                                                    = data             :
164.062500 (KB/s)

                                                    = tps              :
21.349862

                                                    = size/total :
1716/156842 (KB)

                                                    = time cost   :
10164 (ms)

                                                    = no/total     :
217/20958

                                                    = queue size : 0

                                                    ===============
traffic log ===============
```

解释：

data	每秒数据量

tps	吞吐量，每秒数据条数
size/total	time cost 时间内数据大小增量/程序启动后接收数据总量
time cost	距离上一条 traffic log 时间
no/total	time cost 时间内数据条数增量/程序启动后接收数据条数总量
queue size	未分发的数据条数

Tips：主要用于监测网络流量信息，如果每秒数据量在交易时段突然降低，说明网络出现拥堵。

由于订阅数据不同，没有 benchmark。

版本改动说明

1.3.0 版本

重要更新：

- 暂不支持 VS2010

由于采用<chrono>库，暂不支持 vs2010

2.0.0 版本

- 修改数据类型，原有访问数据的方式出现变更，主要包括两处：

I 访问 MarketDataStream 中 MarketData 的方式

例：由原本的

```
1 google::protobuf::RepeatedPtrField<MarketData>::const_iterator it
2 = data_stream.marketdatas().begin();
```

变为：

```
1 google::protobuf::RepeatedPtrField<MarketData>::const_iterator it
2         = data_stream.marketdatalist().marketdatas().begin();
```

II 获取 ESecurityType 的方式

例：由原本的

```
1 ::com::htsc::mdc::insight::model::MarketData marketData;
2 ESecurityType type = marketData.securitytype();
```

变为需要根据具体的数据类型和证券类型来决定：

```
1 ::com::htsc::mdc::insight::model::MarketData marketData;
2 ESecurityType type = marketData.mdstock().securitytype();
```

其他获取 ESecurityType 的方法可以参考 functional_test 中的 get_security_type_name 函数。

- 新增部分日志信息；
- 重连机制优化。

2.0.1 版本

- 新模型适配，新增利率信息数据(MDRate)，基于订单簿推算的行情快照(ADOrderbookSnapshot)，基于订单簿推算的行情快照与原始行情(ADOrderbookSnapshotWithTick)。

2.0.2 版本

- 调整 functional_test 代码结构；
- 完善回测回复数据的部分字段内容，支持回测时判断是否丢失数据；
- 调整部分日志报警级别；
- 更改 protobuf 文件以适配新数据模型。

2.1.0 版本

- 新增 OnPlaybackResponse、OnPlaybackControlResponse、OnSubscribeResponse、OnQueryResponse 四个回调接口用于处理请求回复消息；
- 新增 open_response_callback、close_response_callback、is_response_callback 函数用于控制 response_callback 开关状态；
- 调整配置文件中的配置项；
- 更改 protobuf 文件以适配新数据模型。

2.1.1 版本

- 适配新模型股票新增科创板字段。

2.1.2 版本

- 更改 protobuf 文件以适配新数据模型及字段，请参考数据字典。

3.0.0 版本

- 增加新闻舆情数据，盯盘数据及衍生指标数据。

3.1.0 版本

- 盯盘结果只推送给订阅账号。
- 新增独立数据回放功能，并支持多任务同时回放。
- 修复数据查询分页 bug。
- 更改 protobuf 文件已适配新数据模型及字段，请参考数据字典。
- 提供配置，如果出现 ETIME error 用户可以设置重连。

3.1.2 版本

- 更改 protobuf 文件已适配新数据模型及字段，请参考数据字典。

3.1.3 版本

- 更改 protobuf 文件已适配新数据模型及字段，请参考数据字典。

3.1.5 版本

- 重要更新，更改 protobuf 文件已适配新数据模型及字段适配上交所逐笔委托数据，请参考数据字典。

    - 对上交所委托而言，逐笔委托数据会揭示撮合成交以后的剩余委托数量；
    - 上交所集合竞价及停牌阶段接收到的有效委托不实时发布，在集合竞价或停牌阶段结束后统一发布；
    - 上交所盘后固定价格交易产生的逐笔委托数据不发布；
    - 上交所撤单体现在逐笔委托内，对于撤单而言，OrderType 取 10 深交所在逐笔成交中揭示；
    - 对上交所的逐笔委托来说，OrderNO 代表委托的原始交易系统订单号，在撤单时撤单委托的 OrderNO 即为最初原始报单的 OrderNO；
    - 逐笔成交的 TradeBuyNO 和 TradeSellNO 对应 OrderNO；

3.1.6 版本

- 更改 protobuf 文件已适配新数据模型及字段，请参考数据字典。

3.1.7 版本

- 更改 protobuf 文件已适配新数据模型及字段，请参考数据字典。

3.1.8 版本

- 更改 protobuf 文件已适配新数据模型及字段，请参考数据字典；
- 修复多线程重连的极端情况下，可能出现程序死锁问题；
- 新增站点选择和地址映射开关功能；

3.1.9 版本

- 更改 protobuf 文件以适配新数据模型及字段，请参考数据字典；

    - 新增对上交所新债券交易系统行情支持；
    - 新增对华泰融券通平台日行情和浏览行情的支持；
    - 新增部分静态信息字段；

3.2.0 版本

- 新增客户端端口占用配置功能。

3.2.1 版本

- 更改 protobuf 文件以适配新数据模型及字段，请参考数据字典；

- 新增外汇精度字段；
- 新增部分静态信息字段；
- 优化部分金融数据说明；

3.2.2 版本

- 更改 protobuf 文件以适配新数据模型及字段，请参考数据字典；
  - 新增上交所基金通、深交所基金通数据源；
  - 新增部分静态信息字段；

3.2.3 版本

- 更改 protobuf 文件以适配新数据模型及字段，请参考数据字典；
  - 新增对深交所新债券交易系统支持；

3.2.4 版本

- 更改 protobuf 文件以适配新数据模型及字段，请参考数据字典；
  - 新增美股逐笔成交内部编号；
  - 新增美股最优报价数据；
  - 新增美股期权数据源枚举；
- 融券通交易行情加入外部券行情；

3.2.5 版本

- 更改 protobuf 文件以适配新数据模型及字段，请参考数据字典；
  - 新增 CNEX 成交、报价行情；
  - 新增华泰融券通长期限券行情；

3.2.6 版本

- 更改 protobuf 文件以适配新数据模型及字段，请参考数据字典；
    - 新增美股 OTC 市场的标签，见 ESecurityIDSource；
    - 新增高精度的 IOPV 数据字段：HighAccuracyIOPV 、HighAccuracyPreIOPV，见 MDFund；
    - 新增上交所固收平台固定收益确定报价数据模型，见 MDFIQuote；
    - 新增银行间市场新类型成交数据，在 4-现券买卖市场之上新增银行间市场 9-质押式回购市、10-买断式回购市场、1-同业拆借市场、8-债券借贷市场的成交数据，见 MDTransaction；

3.2.7 版本

- 更改 protobuf 文件以适配新数据模型及字段，请参考数据字典；
    - 新增广期所市场的标签，见 ESecurityIDSource；

3.2.8 版本

- 更改 protobuf 文件以适配新数据模型及字段，请参考数据字典；
    - 新增 EURONEXT、FSI、DBDX、SAO 市场的标签，见 ESecurityIDSource；
    - 新增 MD_CFETS_FOREX 银行间外汇市场行情数据，见 EMarketDataType；
- 更正数据字典部分数据错误；

附录说明

protobuf 编译

编译参考 cmake\README.md

- 配置 cmake（下载安装包加压缩即可）；
- protobuf-3.1.0\cmake 目录下创建 build\solution；
- 启动 cmd，在其中执行 vs 运行环境脚本；
- 进入 solution 目录，通过如下命令生成工程文件（vc 版本根据自己的情况填写）

cmake -G "Visual Studio 12 2013 Win64" -Dprotobuf_BUILD_TESTS=OFF -DCMAKE_INSTALL_PREFIX=../../../../install ../..

- 使用 vc 打开 sln 文件，配置 INCLUDEPATH、LIBPATH、LIBS 等内容；
- 单独编译 protobuf 即可；

修改“配置-常规-平台工具集”

修改目录配置

单独编译 protobuf 工程即可。

华泰证券股份有限公司

HUATAI SECURITIES . CO .,LTD .

业务联系邮箱：insight@htsc.com

业务联系电话：025-83388173

产品官网：insight.htsc.com.cn

公司地址：南京市江东中路 228 号