|  |  |
| --- | --- |
| 渐变_logo_2中 |  |
|  |  |
|  |  |
|  |  |
|  |  |

金融基础件2.0接入

T2SDK开发手册

|  |  |
| --- | --- |
| 文档版本 | V2.0 |
| 发布日期 | 2013-06-27 |

**修改记录**

|  |  |  |  |
| --- | --- | --- | --- |
| 版本号 | 日期 | 修改者 | 修改说明 |
| V2.0 | 20130627 | 许欣芃 | 创建 |

目 录

[目 录 3](#_Toc369512871)

[前 言 4](#_Toc369512872)

[产品简介 4](#_Toc369512873)

[读者对象 4](#_Toc369512874)

[手册概况 4](#_Toc369512875)

[缩略语/术语 4](#_Toc369512876)

[1. 开发包简介 6](#_Toc369512877)

[2. 开发流程 7](#_Toc369512878)

[2.1 同步开发流程 7](#_Toc369512879)

[2.2 异步开发流程 7](#_Toc369512880)

[3. 开发接口 9](#_Toc369512881)

[3.1 T2SDK引出函数 9](#_Toc369512882)

[3.1.1 取开发包版本信息（GetVersionInfo） 9](#_Toc369512883)

[3.1.2 创建配置接口（NewConfig） 9](#_Toc369512884)

[3.1.3 创建连接接口（NewConnection） 10](#_Toc369512885)

[3.1.4 创建打包器接口（NewPacker） 10](#_Toc369512886)

[3.1.5 创建解包器接口（NewUnPacker） 10](#_Toc369512887)

[3.1.6 加密接口（Encode） 11](#_Toc369512888)

[3.1.7 密码加密接口（EncodeEx） 11](#_Toc369512889)

[3.1.8 创建过滤接口（NewFilter） 11](#_Toc369512890)

[3.1.9 创建订阅参数接口（NewSubscribeParam） 12](#_Toc369512891)

[3.1.10 创建业务消息接口（NewBizMessage） 12](#_Toc369512892)

[3.2 配置接口【CConfigInterface】 13](#_Toc369512893)

[3.2.1 读取配置文件（Load） 13](#_Toc369512894)

[3.2.2 保存配置文件（Save） 15](#_Toc369512895)

[3.2.3 获取字符型配置项值（GetString） 15](#_Toc369512896)

[3.2.4 获取整型配置项值（GetInt） 16](#_Toc369512897)

[3.2.5 设置字符型配置项值（SetString） 16](#_Toc369512898)

[3.2.6 设置整型配置项值（SetInt） 16](#_Toc369512899)

[3.3 连接回调接口【CCallbackInterface】 17](#_Toc369512900)

[3.3.1 TCP连接成功回调（OnConnect） 17](#_Toc369512901)

[3.3.2 安全连接成功回调（OnSafeConnect） 17](#_Toc369512902)

[3.3.3 注册成功回调（OnRegister） 18](#_Toc369512903)

[3.3.4 连接断开回调（OnClose） 18](#_Toc369512904)

[3.3.5 数据发送回调（OnSent） 18](#_Toc369512905)

[3.3.6 收到消息回调（OnReceivedBiz） 19](#_Toc369512906)

[3.3.7 收到消息扩展回调（OnReceivedBizEx） 19](#_Toc369512907)

[3.3.8 收到业务消息回调（OnReceivedBizMsg） 20](#_Toc369512908)

[3.4 业务消息接口【IBizMessage】 21](#_Toc369512909)

[3.4.1 设置功能号（SetFunction） 21](#_Toc369512910)

[3.4.2 获取功能号（GetFunction） 21](#_Toc369512911)

[3.4.3 设置包类型（SetPacketType） 22](#_Toc369512912)

[3.4.4 获取包类型（GetPacketType） 22](#_Toc369512913)

[3.4.5 设置营业部号（SetBranchNo） 22](#_Toc369512914)

[3.4.6 获取营业部号（GetBranchNo） 23](#_Toc369512915)

[3.4.7 设置系统号（SetSystemNo） 23](#_Toc369512916)

[3.4.8 获取系统号（GetSystemNo） 23](#_Toc369512917)

[3.4.9 设置子系统号（SetSubSystemNo） 23](#_Toc369512918)

[3.4.10 获取子系统号（GetSubSystemNo） 24](#_Toc369512919)

[3.4.11 设置发送者编号（SetSenderId） 24](#_Toc369512920)

[3.4.12 获取发送者编号（GetSenderId） 24](#_Toc369512921)

[3.4.13 设置包序号（SetPacketId） 25](#_Toc369512922)

[3.4.14 获取包序号（GetPacketId） 25](#_Toc369512923)

[3.4.15 设置目的地路由（SetTargetInfo） 25](#_Toc369512924)

[3.4.16 获取目的地路由（GetTargetInfo） 26](#_Toc369512925)

[3.4.17 设置发送者路由（SetSendInfo） 26](#_Toc369512926)

[3.4.18 获取发送者路由（GetSendInfo） 26](#_Toc369512927)

[3.4.19 设置错误号（SetErrorNo） 27](#_Toc369512928)

[3.4.20 获取错误号（GetErrorNo） 27](#_Toc369512929)

[3.4.21 设置错误信息（SetErrorInfo） 27](#_Toc369512930)

[3.4.22 获取错误信息（GetErrorInfo） 28](#_Toc369512931)

[3.4.23 设置返回码（SetReturnCode） 28](#_Toc369512932)

[3.4.24 获取返回码（GetReturnCode） 28](#_Toc369512933)

[3.4.25 设置业务内容（SetContent） 28](#_Toc369512934)

[3.4.26 获取业务内容（GetContent） 29](#_Toc369512935)

[3.4.27 设置订阅类型（SetIssueType） 29](#_Toc369512936)

[3.4.28 获取订阅类型（GetIssueType） 30](#_Toc369512937)

[3.4.29 设置订阅序号（SetSequeceNo） 30](#_Toc369512938)

[3.4.30 获取订阅序号（GetSequeceNo） 31](#_Toc369512939)

[3.4.31 设置订阅关键字段（SetKeyInfo） 31](#_Toc369512940)

[3.4.32 获取订阅关键字段（GetKeyInfo） 32](#_Toc369512941)

[3.4.33 设置附加数据（SetAppData） 32](#_Toc369512942)

[3.4.34 获取附加数据（GetAppData） 32](#_Toc369512943)

[3.4.35 请求转换成应答（ChangeReq2AnsMessage） 33](#_Toc369512944)

[3.4.36 消息转换成二进制流（GetBuff） 33](#_Toc369512945)

[3.4.37 二进制流转换成消息（SetBuff） 33](#_Toc369512946)

[3.5 打包器接口【IF2Packer】 34](#_Toc369512947)

[3.5.1 设置缓存区空间（SetBuffer） 34](#_Toc369512948)

[3.5.2 开始打包（BeginPack） 34](#_Toc369512949)

[3.5.3 构造新结果集（NewDataset） 34](#_Toc369512950)

[3.5.4 添加新字段（AddField） 35](#_Toc369512951)

[3.5.5 添加string型数据（AddStr） 35](#_Toc369512952)

[3.5.6 添加int型数据（AddInt） 35](#_Toc369512953)

[3.5.7 添加double型数据（AddDouble） 36](#_Toc369512954)

[3.5.8 添加char型数据（AddChar） 36](#_Toc369512955)

[3.5.9 添加二进制数据（AddRaw） 36](#_Toc369512956)

[3.5.10 结束打包（EndPack） 37](#_Toc369512957)

[3.5.11 获取打包结果指针（GetPackBuf） 37](#_Toc369512958)

[3.5.12 获取打包结果长度（GetPackLen） 37](#_Toc369512959)

[3.5.13 获取打包缓存长度（GetPackBufSize） 37](#_Toc369512960)

[3.5.14 获取打包版本（GetVersion） 38](#_Toc369512961)

[3.5.15 设置返回码（SetReturnCode） 38](#_Toc369512962)

[3.5.16 获取解包接口（UnPack） 38](#_Toc369512963)

[3.5.17 释放打包内存（FreeMem） 39](#_Toc369512964)

[3.5.18 清空最后一个结果集数据（ClearValue） 39](#_Toc369512965)

[3.5.19 开始打包扩展接口（BeginPackEx） 39](#_Toc369512966)

[3.5.20 清空当前结果集（ClearDataSet） 39](#_Toc369512967)

[3.6 结果集接口【IF2ResultSet】 40](#_Toc369512968)

[3.6.1 获取列个数（GetColCount） 40](#_Toc369512969)

[3.6.2 根据下标获取列名字（GetColName） 40](#_Toc369512970)

[3.6.3 获取列类型（GetColType） 40](#_Toc369512971)

[3.6.4 获取列精度（GetColScale） 41](#_Toc369512972)

[3.6.5 获取列最大长度（GetColWidth） 41](#_Toc369512973)

[3.6.6 根据列名字找到列下标（FindColIndex） 41](#_Toc369512974)

[3.6.7 根据列下标获取string类型值（GetStrByIndex） 42](#_Toc369512975)

[3.6.8 根据列名字获取string类型值（GetStr） 42](#_Toc369512976)

[3.6.9 根据列下标获取char类型值（GetCharByIndex） 42](#_Toc369512977)

[3.6.10 根据列名字获取char类型值（GetChar） 42](#_Toc369512978)

[3.6.11 根据列下标获取double类型值（GetDoubleByIndex） 43](#_Toc369512979)

[3.6.12 根据列名字获取double类型值（GetDouble） 43](#_Toc369512980)

[3.6.13 根据列下标获取int类型值（GetIntByIndex） 43](#_Toc369512981)

[3.6.14 根据列名字获取int类型值（GetInt） 43](#_Toc369512982)

[3.6.15 根据列下标获取二进制类型值（GetRawByIndex） 44](#_Toc369512983)

[3.6.16 根据列名字获取二进制类型值（GetRaw） 44](#_Toc369512984)

[3.6.17 最后一次取得值是否为空（WasNull） 44](#_Toc369512985)

[3.6.18 结果集下一条记录（Next） 45](#_Toc369512986)

[3.6.19 结果集是否到结尾（IsEOF） 45](#_Toc369512987)

[3.6.20 结果集是否为空（IsEmpty） 45](#_Toc369512988)

[3.6.21 结果集释放（Destroy） 46](#_Toc369512989)

[3.7 解包器接口【IF2UnPacker】 46](#_Toc369512990)

[3.7.1 获取解包器版本（GetVersion） 46](#_Toc369512991)

[3.7.2 二进制转换为解包器（Open） 46](#_Toc369512992)

[3.7.3 获取结果集个数（GetDatasetCount） 47](#_Toc369512993)

[3.7.4 根据下标设置当前结果集（SetCurrentDatasetByIndex） 47](#_Toc369512994)

[3.7.5 根据结果集名字设置当前结果集（SetCurrentDataset）、 47](#_Toc369512995)

[3.7.6 获取解包器缓存指针（GetPackBuf） 48](#_Toc369512996)

[3.7.7 获取解包器缓存长度（GetPackLen） 48](#_Toc369512997)

[3.7.8 获取当前结果集行数（GetRowCount） 48](#_Toc369512998)

[3.7.9 指向结果集第一行记录（First） 48](#_Toc369512999)

[3.7.10 指向结果集最后一行记录（Last） 49](#_Toc369513000)

[3.7.11 跳转到结果集任意行（Go） 49](#_Toc369513001)

[3.8 过滤器接口【CFilterInterface】 49](#_Toc369513002)

[3.8.1 根据下标获取条件名字（GetFilterNameByIndex） 49](#_Toc369513003)

[3.8.2 根据下标获取条件值（GetFilterValueByIndex） 50](#_Toc369513004)

[3.8.3 根据条件名字好获取条件值（GetFilterValue） 50](#_Toc369513005)

[3.8.4 获取过滤条件个数（GetCount） 50](#_Toc369513006)

[3.8.5 设置过滤条件名字和值（SetFilter） 51](#_Toc369513007)

[3.9 订阅参数接口【CSubscribeParamInterface】 51](#_Toc369513008)

[3.9.1 设置主题名字（SetTopicName） 51](#_Toc369513009)

[3.9.2 设置附加数据（SetAppData） 51](#_Toc369513010)

[3.9.3 设置过滤条件（SetFilter） 52](#_Toc369513011)

[3.9.4 设置返回字段（SetReturnFiled） 52](#_Toc369513012)

[3.9.5 设置补缺标志（SetFromNow） 52](#_Toc369513013)

[3.9.6 设置覆盖标志（SetReplace） 53](#_Toc369513014)

[3.9.7 设置发送间隔（SetSendInterval） 53](#_Toc369513015)

[3.9.8 获取主题名字（GetTopicName） 53](#_Toc369513016)

[3.9.9 获取附加数据（GetAppData） 54](#_Toc369513017)

[3.9.10 根据下标获取过滤字段名字（GetFilterNameByIndex） 54](#_Toc369513018)

[3.9.11 根据下标获取过滤字段值（GetFilterValueByIndex） 54](#_Toc369513019)

[3.9.12 根据过滤名字获取值（GetFilterValue） 55](#_Toc369513020)

[3.9.13 获取过滤条件个数（GetFilterCount） 55](#_Toc369513021)

[3.9.14 获取返回字段（GetReturnFiled） 55](#_Toc369513022)

[3.9.15 获取补缺标志（GetFromNow） 55](#_Toc369513023)

[3.9.16 获取覆盖标志（GetReplace） 56](#_Toc369513024)

[3.9.17 获取发送间隔（GetSendInterval） 56](#_Toc369513025)

[3.10 订阅回调接口【CSubCallbackInterface】 56](#_Toc369513026)

[3.10.1 收到发布消息回调（OnReceived） 56](#_Toc369513027)

[3.10.2 收到剔除订阅回调（OnRecvTickMsg） 57](#_Toc369513028)

[3.11 订阅接口【CSubscribeInterface】 58](#_Toc369513029)

[3.11.1 订阅主题 58](#_Toc369513030)

[3.11.2 取消订阅 58](#_Toc369513031)

[3.11.3 取消订阅扩展接口 59](#_Toc369513032)

[3.11.4 获取已经订阅的信息 59](#_Toc369513033)

[3.12 发布接口【CPublishInterface】 60](#_Toc369513034)

[3.12.1 业务发包格式发送 60](#_Toc369513035)

[3.12.2 二进制数据发送 61](#_Toc369513036)

[3.13 连接接口【CConnectionInterface】 61](#_Toc369513037)

[3.13.1 初始化连接对象（Create） 62](#_Toc369513038)

[3.13.2 开始连接/注册（Connect） 62](#_Toc369513039)

[3.13.3 断开连接（Close） 62](#_Toc369513040)

[3.13.4 取连接服务器地址（GetServerAddress） 63](#_Toc369513041)

[3.13.5 取连接状态（GetStatus） 63](#_Toc369513042)

[3.13.6 取服务器负载（GetServerLoad） 64](#_Toc369513043)

[3.13.7 取错误码对应的错误信息（GetErrorMsg） 64](#_Toc369513044)

[3.13.8 取连接错误号（GetConnectError） 64](#_Toc369513045)

[3.13.9 发送业务数据（SendBiz） 65](#_Toc369513046)

[3.13.10 接收业务数据（RecvBiz） 65](#_Toc369513047)

[3.13.11 发送业务数据扩展接口（SendBizEx） 66](#_Toc369513048)

[3.13.12 接收业务数据扩展接口（RecvBizEx） 67](#_Toc369513049)

[3.13.13 连接初始化的扩展接口（CreateEx） 68](#_Toc369513050)

[3.13.14 获取服务端上连接的IP和端口（GetRealAddress） 68](#_Toc369513051)

[3.13.15 获取本地连接使用的IP和端口（GetSelfAddress） 68](#_Toc369513052)

[3.13.16 获取连接使用的MAC地址（GetSelfMac） 69](#_Toc369513053)

[3.13.17 创建订阅者（NewSubscriber） 69](#_Toc369513054)

[3.13.18 获取发布者（GetPublisher） 70](#_Toc369513055)

[3.13.19 获取主题信息（GetTopic） 70](#_Toc369513056)

[3.13.20 获取消息中心的最后错误（GetMCLastError） 71](#_Toc369513057)

[3.13.21 连接用IBizMessage接口初始化（Create2BizMsg） 72](#_Toc369513058)

[3.13.22 连接发送IBizMessage业务消息（SendBizMsg） 72](#_Toc369513059)

[3.13.23 连接接收IBizMessage业务消息（RecvBizMsg） 72](#_Toc369513060)

[4. 注意事项 73](#_Toc369513061)

[4.1 相关限制 73](#_Toc369513062)

[4.2 调用限制 74](#_Toc369513063)

[4.3 回调线程 74](#_Toc369513064)

[4.4 编程建议 74](#_Toc369513065)

[5. 示例代码 75](#_Toc369513066)

[5.1 同步发送接收 75](#_Toc369513067)

[5.2 异步发送接收 79](#_Toc369513068)

[5.3 订阅 83](#_Toc369513069)

[5.4 发布 89](#_Toc369513070)

[6. 附录 93](#_Toc369513071)

[6.1 T2sdk错误码 93](#_Toc369513072)

前 言

产品简介

英文全称TCP 2 Software Development Kit。T2SDK是接入金融基础件2.0的客户端开发包，因为该开发包底层采用TCP协议，取名为“T2SDK”。T2SDK以dll（Windows下）或so（Linux下）的形式提供给开发者，通过使用T2SDK开发程序可以轻松实现和CRES2.0（金融基础件2.0）平台的对接，使用T2\_SDK开发的程序在本文档中也称为“客户端”。

读者对象

本指南主要适用于以下人员：

* 开发人员，仅限于C++开发，对接各种在CRES2.0平台基础上搭建的系统，譬如：UFX，UF2.0等等。

手册概况

本手册各章节内容如下表所示。

|  |  |
| --- | --- |
| 章节 | 内容 |
| 1开发包简介 | 介绍开发包的支持系统，获取方式 |
| 2开发流程 | 介绍开发的步骤，异步同步的开发模式 |
| 3 开发接口 | 具体的开发接口的函数说明 |
| 4示例代码 | 开发的demo的实例代码 |
| 5 使用注意事项 | 介绍接口调用时禁止的方式 |

缩略语/术语

下面列出了本手册中出现的缩略语和术语。

|  |  |  |
| --- | --- | --- |
| **C** |  |  |
| CRES2.0 | 金融基础件2.0 | 恒生的新一代金融基础件平台，本文档介绍的开发包就是接入此平台的C++版本的开发包。下面说的服务端不特殊说明都是指CRES2.0. |
| **U** |  |  |
| UFX | 恒生统一接入系统 | 客户端的统一接入控制系统，是基于CRES2.0平台之上搭建，用T2SDK可以接入。 |
| UF2.0 | 恒生经济运营平台 | 恒生的证券经纪运营平台系统，是基于CRES2.0平台之上搭建，用T2SDK可以接入。 |
| M |  |  |
| MC2.0 | 消息中心2.0 | 本文档提到的订阅发布接口都是针对消息中心2.0,1.0的订阅发布不能用这个接口，需要自己开发代码 |
| MC1.0 | 消息中心1.0 | 消息中心1.0需要通过业务消息中增加字段来订阅，没有现成的接口调用。 |

# 开发包简介

名称：t2sdk.dll

开发语言：C++

支持操作系统：windows/linux

**版本信息获取：**

* WINDOWS：右键->属性->详细信息，里面有版本信息，格式是“1.0.0.\*”。
* LINUX：通过命令“strings libt2sdk.so |grep version”，如下所示：

[integration@fbase56 lib]$ strings libt2sdk.so |grep version

lzo\_version

\_lzo\_version\_string

\_lzo\_version\_date

\_Z10conversionc

\_Z10conversionPc

**version Jun 27 2013 14:47:40 这就是版本信息**

SSL handshake failed, maybe client and server SSL version mismatch.

version

**获取方式：**

* 恒生内部的员工，以项目组的形式向研发中心客服申请使用。
* 恒生外部开发商，向对应的项目的恒生接口人获取开发包。

# 开发流程

本章节主要介绍两种开发的模式，同步、异步。

* 1. 同步开发流程



* 1. 异步开发流程



# 开发接口

本章主要介绍开发包提供的所有接口，函数参数说明。没有特殊说明，接口都是线程安全的。

* 1. T2SDK引出函数

### 取开发包版本信息（GetVersionInfo）

**函数原型：**

int FUNCTION\_CALL\_MODE GetVersionInfo();

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回发布包的版本信息，返回类型是一个INT型数据。

**用法说明：**

返回的INT数据，高位的4bit表示开发包的类型，始终为1，后面28bit表示版本信息，版本信息值越大，版本越高。

### 创建配置接口（NewConfig）

**函数原型：**

CConfigInterface\* FUNCTION\_CALL\_MODE NewConfig();

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

新的连接配置项接口。

**用法说明：**

通过返回的新的连接配置项接口，设置相应的连接属性，具体的接口说明参照下面的CConfigInterface接口说明。

注意

NewConfig()返回的接口，必须调用AddRef，否则调用Release接口就无法释放

### 创建连接接口（NewConnection）

**函数原型：**

CConnectionInterface\* FUNCTION\_CALL\_MODE NewConnection(CConfigInterface \*lpConfig);

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CConfigInterface \*lpConfig | 配置对象，通过NewConfig()返回 | 输入参数 |

**返回：**

新的连接接口，所有发包收包都是建立在连接接口之上。

**用法说明：**

通过返回的连接接口指针，就可以访问服务端，发功能号，收应答，所有操作都是基于连接。

注意

NewConnection ()返回的接口，必须调用AddRef，否则调用Release接口就无法释放

### 创建打包器接口（NewPacker）

**函数原型：**

IF2Packer \* FUNCTION\_CALL\_MODE NewPacker(int iVersion);

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int iVersion | 打包器的版本，支持1,2两种 | 输入参数 |

**返回：**

新的打包器接口，用于业务数据的组装，传递业务接口需要的参数

**用法说明：**

打包器需要什么版本，根据业务接口的说明来决定。

注意

NewPacker返回的接口，必须调用AddRef，否则调用Release接口就无法释放

### 创建解包器接口（NewUnPacker）

**函数原型：**

IF2UnPacker \* FUNCTION\_CALL\_MODE NewUnPacker(void \* lpBuffer, unsigned int iLen);

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| void \* lpBuffer | 二进制数据的首指针 | 输入参数 |
| unsigned int iLen | 二进制数据的长度 | 输入参数 |

**返回：**

根据传入的二进制数据进行解包，生成新的解包器，数据不符合解包器格式，就放那会NULL。

**用法说明：**

解包器一般用于收到应答时，对业务应答的解析，获取自己需要的内容。

注意

NewUnPacker返回的接口，必须调用AddRef，否则调用Release接口就无法释放

### 加密接口（Encode）

**函数原型：**

char \* FUNCTION\_CALL\_MODE Encode(char \*EncodePass, const char\* Password, int nReserve = 0 );

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| char \*EncodePass | 传出的加密结果，长度不超过16（包括'\0'） | 输出参数 |
| const char\* Password | 传入的待加密的密码 | 输入参数 |
| int nReserve | 保留输入参数，采用默认值即可 | 输入参数 |

**返回：**

对密码加密处理，返回，输出参数的空间是外面申请。

**用法说明：**

用于客户端传递密码时，加密处理，防止密码以明文形式传递。**不过此接口已经不用，由下面的EncodeEx取代。**

### 密码加密接口（EncodeEx）

**函数原型：**

int FUNCTION\_CALL\_MODE EncodeEx(const char \*pIn, char \*pOut);

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \*pIn | 传入的待加密的字符串 | 输入参数 |
| char \*pOut | 长度和传入的字符串大小一样，由外部申请 | 输出参数 |

**返回：**

对密码加密处理，返回，输出参数的空间是外面申请。

**用法说明：**

用于客户端传递密码时，加密处理，防止密码以明文形式传递。

### 创建过滤接口（NewFilter）

**函数原型：**

CFilterInterface\* FUNCTION\_CALL\_MODE NewFilter();

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

新的过滤接口，用于订阅发布的时候添加过滤属性用

**用法说明：**

订阅和发布的时候需要根据订阅主题，指定相关的过滤参数，根据过滤条件获取对应的发布消息。

注意

NewFilter返回的接口，必须调用AddRef，否则调用Release接口就无法释放

### 创建订阅参数接口（NewSubscribeParam）

**函数原型：**

CSubscribeParamInterface\* FUNCTION\_CALL\_MODE NewSubscribeParam();

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

新的订阅参数接口，用于订阅时候添加订阅属性用

**用法说明：**

订阅时候用来添加订阅属性，包括主题信息，过滤信息，发送频度等等。

注意

NewSubscribeParam返回的接口，必须调用AddRef，否则调用Release接口就无法释放

### 创建业务消息接口（NewBizMessage）

**函数原型：**

IBizMessage\* FUNCTION\_CALL\_MODE NewBizMessage();

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

新的业务消息接口

**用法说明：**

通过这个接口设置业务相关的属性，底层根据这个消息构造二进制发送。

注意

NewBizMessage返回的接口，必须调用AddRef，否则调用Release接口就无法释放

* 1. 配置接口【CConfigInterface】

### 读取配置文件（Load）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE Load(const char \*szFileName) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \*szFileName | 连接属性的配置文件 | 输入参数 |

**返回：**

加载成功返回0，其他值表示失败

**用法说明：**

这里加载的文件是ini格式的文件，下面是文件的简单实例：

[t2sdk]

;许可证文件

license\_file=

;服务器地址

servers=192.168.194.56:6002;192.168.194.56:8001

;接受缓存初始化大小

init\_recv\_buf\_size=512

;发送缓存初始化大小

init\_send\_buf\_size=512

;发送队列

send\_queue\_size=1000

[safe]

;安全级别

safe\_level=none

;客户号

client\_id=123456

;客户密码

comm\_pwd=888888

;ssl证书

cert\_file=xxx.pem

;ssl证书密码

cert\_pwd=xxxxxxxx

每个标签以及属性都有自己的含义，下面是一些通用标签和含义的解释：

|  |  |  |
| --- | --- | --- |
| 标签名 | 属性名 | 含义 |
| [t2sdk] |  | 基本参数 |
|  | servers | 客户端要连接的服务端的IP和端口，可以配置一个或多个，中间以;分割，第一次连接时，会从这串服务端地址中，随即选择一个地址来尝试建立连接； 再次连接时，会取当前地址的后一个地址来尝试建立连接，以此类推 |
|  | init\_recv\_buf\_size | 接收缓存的初始大小，单位字节，实际接收到服务端的数据时，可能会扩大（如果需要） |
|  | init\_send\_buf\_size | 每块发送缓存的初始大小，单位字节，该大小也会根据需要动态扩大 |
|  | send\_queue\_size | 发送队列的大小，该大小不会动态变化，若该配置项很小，且连接发包很频繁，则可能因为发送队列满而造成发送失败 |
|  | license\_file | 此配置项配置许可证文件的路径，如果没有配置，则默认为当前目录下的license.dat。该许可证需向恒生电子股份有限公司申请。 |
|  | heartbeat\_time | 客户端给服务端发送心跳的间隔时间，单位为秒，最小值为5秒，不配或者配<=0，表示不开启客户端心跳 |
|  | if\_error\_log | 是否记录错误日志的参数，1表示记录，0表示不记录，默认不记录。 |
|  | logdir | 记录日志的目录，默认在程序运行的当前目录，只有开启记录日志，才会记录 |
|  | if\_sendRecv\_log | 记录发包和收包的日志，用于客户端查找问题，1表示记录，0表示不记录，默认不记录 |
|  | login\_name | 有名客户端的名字，默认是匿名 |
|  | license\_pwd | 许可证的加密密码，用于UFX接入时，默认没有密码 |
| [safe] |  | 连接相关 |
|  | safe\_level | 连接的安全模式，可以选择明文（none），通信密码（pwd），SSL（ssl）,注意大小写敏感 |
|  | client\_id | 当连接的安全模式为pwd时，client\_id配置项才生效,客户号，一个客户号对应一个密码 |
|  | comm\_pwd | 在pwd模式下，当client\_id为空时，comm\_pwd为默认的密钥; 注意当client\_id为空时，comm\_pwd必须和服务端的配置一致方可正常使用 |
|  | ca\_file | 指定客户端的证书（根证书，校验服务端用） |
|  | ca\_pwd | 若证书为\*.pfx，则需要配ca\_pwd，若为\*.pem，则可配可不配 |
|  | cert\_file | cert\_file配置被服务端校验的证书路径和密码 |
|  | cert\_pwd | 不管格式如何，cert\_pwd必配 |
|  | check\_server\_cert | 是否校验服务端，可不配，若不配，默认校验服务端，0表示不校验。1表示校验 |
| [proxy] |  | 代理参数 |
|  | proxy\_type | 可配置采用何种代理配置http/socks4/socks5，空表示不采用代理 |
|  | ip | 若proxy\_type不为空，下面配置生效; 代理服务器的IP地址 |
|  | port | 代理服务器的端口 |
|  | user\_name | 登陆代理服务器的用户名 |
|  | password | 登陆代理服务器的密码 |
| [mc] |  | 发布订阅标签，需要发布订阅就必须配置 |
|  | client\_name | 客户端名字 |

### 保存配置文件（Save）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE Save(const char \*szFileName) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \*szFileName | 需要保存属性的配置文件 | 输入参数 |

**返回：**

保存成功返回0，其他值表示失败

**用法说明：**

这个函数是把属性内存落地成文件，文件格式如上面加载的文件。

### 获取字符型配置项值（GetString）

**函数原型：**

virtual const char \* FUNCTION\_CALL\_MODE GetString(const char \*szSection, const char \*szEntry, const char \*szDefault) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \*szSection | 节名 | 输入参数 |
| const char \*szEntry | 变量名 | 输入参数 |
| const char \*szDefault | 默认值 | 输入参数 |

**返回：**

获取ini配置里面相应的节点名和变量名的属性值，用字符串形式返回。

**用法说明：**

[t2sdk]

license\_file=

上面t2sdk就是节点名，license\_file就是属性名，如果配置中没有节点名和属性名，用第三个参数作为返回值。

### 获取整型配置项值（GetInt）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetInt(const char \*szSection, const char \*szEntry, int iDefault) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \*szSection | 节名 | 输入参数 |
| const char \*szEntry | 变量名 | 输入参数 |
| int iDefault | 默认值 | 输入参数 |

**返回：**

获取ini配置里面相应的节点名和变量名的属性值，用整型形式返回。

**用法说明：**

节点名和属性名定义如上面所示，同样的，如果不存在这个属性，就返回第三个参数。

### 设置字符型配置项值（SetString）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE SetString(const char \*szSection, const char \*szEntry, const char \*szValue) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \*szSection | 节名 | 输入参数 |
| const char \*szEntry | 变量名 | 输入参数 |
| const char \*szValue | 需要设置的值 | 输入参数 |

**返回：**

设置字符串型属性值到对应的节点和变量中。返回0表示设置成功，其他表示失败。

**用法说明：**

通过代码设置属性到配置项中，可以不需要读取配置文件。

### 设置整型配置项值（SetInt）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE SetInt(const char \*szSection, const char \*szEntry, int iValue) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \*szSection | 节名 | 输入参数 |
| const char \*szEntry | 变量名 | 输入参数 |
| int iValue | 需要设置的值 | 输入参数 |

**返回：**

设置整型属性值到对应的节点和变量中。返回0表示设置成功，其他表示失败。

**用法说明：**

下面代码展示如果通过代码设置连接属性，不需要读取配置文件，当然读取文件之后，也是可以再设置属性，属性设置是覆盖的：

CConfigInterface \* lpConfig = NewConfig();

lpConfig->AddRef();

lpConfig->SetString("t2sdk", "servers", "192.168.94.30:9999");

lpConfig->SetString("t2sdk", "license\_file", "license.dat");

lpConfig->SetInt("t2sdk", "send\_queue\_size", 100);

lpConfig->SetString("safe", "safe\_level", "none");

* 1. 连接回调接口【CCallbackInterface】

如果是异步的模式，需要完成的回调接口定义。上层业务代码需要定义类继承这个接口，然后实现相应的功能。

### TCP连接成功回调（OnConnect）

这个函数在TCP连接成功之后就会回调。

**函数原型：**

virtual void FUNCTION\_CALL\_MODE OnConnect(CConnectionInterface \*lpConnection) = 0;

**回调参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CConnectionInterface \*lpConnection | 当前回调产生的连接指针 | 回调 |

**用法说明：**

在SOCKET套接字建立成功之后，回调这个函数，这个回调里面一般也就是简单的展示连接信息，无法进行任何操作。

### 安全连接成功回调（OnSafeConnect）

这个函数在安全连接成功之后就会回调，安全连接就是指，密码安全模式下，密码交换成功之后；或者SSL连接建立成功之后。

**函数原型：**

virtual void FUNCTION\_CALL\_MODE OnSafeConnect(CConnectionInterface \*lpConnection) = 0;

**回调参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CConnectionInterface \*lpConnection | 当前回调产生的连接指针 | 回调 |

**用法说明：**

在安全连接建立成功之后，回调这个函数，这个回调里面一般也就是简单的展示连接信息，无法进行任何操作。

### 注册成功回调（OnRegister）

这个函数在成功收到服务端注册应答之后就会回调，注册动作时发生在安全连接之后。

**函数原型：**

virtual void FUNCTION\_CALL\_MODE OnRegister(CConnectionInterface \*lpConnection) = 0;

**回调参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CConnectionInterface \*lpConnection | 当前回调产生的连接指针 | 回调 |

**用法说明：**

在注册成功之后，回调这个函数，这个回调里面可以通知其他线程已经可以开始干活了，连接已经建立，可以发包收包做业务操作了。

### 连接断开回调（OnClose）

这个函数在连接断开之后，会进入这个回调，连接断开可能是服务器断开，或者客户端检测到连接断开，等等。

**函数原型：**

virtual void FUNCTION\_CALL\_MODE OnClose(CConnectionInterface \*lpConnection) = 0;

**回调参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CConnectionInterface \*lpConnection | 当前回调产生的连接指针 | 回调 |

**用法说明：**

连接断开回调里面，可以自己重新连接，但是不可以在这个回调函数里面直接调用连接的connect函数，需要通过事件通知，其他线程进行重连，当然也可以不进行重连，开发包底层会在下一次发包之前判断连接状态，然后就会进行重连。

### 数据发送回调（OnSent）

这个函数在连接上有数据发送成功之后，就会回调这个函数。

**函数原型：**

virtual void FUNCTION\_CALL\_MODE OnSent(CConnectionInterface \*lpConnection, int hSend, void \*reserved1, void \*reserved2, int nQueuingData) = 0;

**回调参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CConnectionInterface \*lpConnection | 当前回调产生的连接指针 | 回调 |
| int hSend | 发送的句柄，连接的发送函数成功之后返回的句柄 | 回调 |
| void \*reserved1 | 保留字段1 | 回调 |
| void \*reserved2 | 保留字段2 | 回调 |
| int nQueuingData | 剩余发送队列长度 | 回调 |

**用法说明：**

这个回调是在连接成功发送数据之后回调，主要是为了告诉上层应用需要发送的数据已经成功发送。是通过参数hsend来匹配是哪部分数据发送成功。这个回调里面一般不建议做业务操作，最多就展示。

### 收到消息回调（OnReceivedBiz）

这个函数在连接上有数据接收成功之后，就会回调这个函数。下面介绍的两个也是同样的回调，只是返回的内容多少有所不同，回调接口只会调用其中一个，具体调用哪一个回调，是由连接的Creat函数决定的。这个回调是调用连接接口的Create函数。

**函数原型：**

virtual void FUNCTION\_CALL\_MODE OnReceivedBiz(CConnectionInterface \*lpConnection, int hSend, const void \*lpUnPackerOrStr, int nResult) = 0;

**回调参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CConnectionInterface \*lpConnection | 当前回调产生的连接指针 | 回调 |
| int hSend | 发送的句柄，这表示接收到的这个数据是针对哪个请求的应答，通过这个句柄进行匹配 | 回调 |
| const void \*lpUnPackerOrStr | 业务详细内容，根据下一个参数的值不一样而不同 | 回调 |
| int nResult | 返回值，决定上一个参数的表达形式 | 回调 |

**用法说明：**

* 如果nResult等于0，表示业务数据接收成功，并且业务操作成功，lpUnpackerOrStr指向一个解包器，此时应首先将该指针转换为IF2UnPacker \*。
* 如果nResult等于1，表示业务数据接收成功，但业务操作失败了，lpUnpackerOrStr指向一个解包器，此时应首先将该指针转换为IF2UnPacker \*。
* 如果nResult等于2，表示收到非业务错误信息，lpUnpackerOrStr指向一个可读的字符串错误信息。
* 如果nResult等于3，表示业务包解包失败。lpUnpackerOrStr指向NULL。

这个回调主要就是把异步推送过来的消息交给上层处理，可能是之前发送的请求的应答， 也可以是服务端主推的消息，根据hsedn来匹配。在这个函数中，建议不要做太多的业务操作，否则会阻塞回调的线程，建议直接拷贝数据，然后入队列，其他线程从队列中获取数据处理。

### 收到消息扩展回调（OnReceivedBizEx）

这个函数和上面的函数一样，只是增加了一些返回的内容。触发这个回调，是调用连接接口的CreateEx函数。

**函数原型：**

virtual void FUNCTION\_CALL\_MODE OnReceivedBizEx(CConnectionInterface \*lpConnection, int hSend, LPRET\_DATA lpRetData, const void \*lpUnpackerOrStr, int nResult) = 0;

**回调参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CConnectionInterface \*lpConnection | 当前回调产生的连接指针 | 回调 |
| int hSend | 发送的句柄，这表示接收到的这个数据是针对哪个请求的应答，通过这个句柄进行匹配 | 回调 |
| LPRET\_DATA lpRetData | 回调的应答结构体，消息中部分内容。 | 回调 |
| const void \*lpUnPackerOrStr | 业务详细内容，根据下一个参数的值不一样而不同 | 回调 |
| int nResult | 返回值，决定上一个参数的表达形式 | 回调 |

**用法说明：**

nResult的说明跟前面的接口一样，不再重复，这里介绍一下额外返回的业务体结构：

typedef struct tagRespondData

{

int functionID;//返回的功能号

int returnCode;//返回码

int errorNo;//错误号

char\* errorInfo;//错误信息

int issueType;//MC1.0的订阅类型

void\* lpKeyInfo;//MC1.0的消息关键字段，这是一个pack的二进制格式

int keyInfoLen;//MC1.0的关键字段长度

Route\_Info sendInfo;//20110302 xuxp 应答里面增加发送者信息

}RET\_DATA, \*LPRET\_DATA;

通过这个结构体返回一些之前的OnReceivedBiz没有返回的信息，便于业务层的操作，尤其是需要消息中心1.0的订阅和发布的功能的时候，就必须使用这个回调接口了。

### 收到业务消息回调（OnReceivedBizMsg）

这个函数和上面的函数一样，返回了一个完整的业务消息，所有的应答内容都会在这个消息中包含。触发这个回调，是调用连接接口的Create2BizMsg函数。建议使用这个回调接口，因为包含所有需要的消息，比上面两个接口返回的字段都全。

**函数原型：**

virtual void FUNCTION\_CALL\_MODE OnReceivedBizMsg(CConnectionInterface \*lpConnection, int hSend, IBizMessage\* lpMsg) = 0;

**回调参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CConnectionInterface \*lpConnection | 当前回调产生的连接指针 | 回调 |
| int hSend | 发送的句柄，这表示接收到的这个数据是针对哪个请求的应答，通过这个句柄进行匹配 | 回调 |
| IBizMessage\* lpMsg | 返回的业务消息接口，通过接口可以访问需要关心的内容。 | 回调 |

**用法说明：**

这里返回消息接口的定义，下面有详细的介绍。如果程序是一个服务端，通过异步接收请求，然后处理完之后转成应答返回，可以调用消息接口的请求转应答操作。建议，所有的开发，以后都采用这个回调接口，方便业务的操作。

* 1. 业务消息接口【IBizMessage】

业务消息接口，主要是返回消息中心各种需要关心的字段，当然发送消息也可以通过设置相关字段，发送请求。

### 设置功能号（SetFunction）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetFunction(const int nFUnctionNo) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const int nFUnctionNo | 功能号 | 输入参数 |

**返回：**

没有返回值

**用法说明：**

为消息设置一个功能号，所有消息都必须有的字段，后台根据功能号选择相应的业务逻辑处理。

### 获取功能号（GetFunction）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetFunction() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回消息中对应的功能号

**用法说明：**

一般用于从应答消息中获取对应的功能号。

### 设置包类型（SetPacketType）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetPacketType(const int nPacketType) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const int nPacketType | 消息类型，0表示请求，1表示应答。没有其他值。 | 输入 |

**返回：**

没有返回值

**用法说明：**

为消息设置一个消息类型，所有消息都必须有的字段，客户端一般是发请求，如果作为服务处理端，那就是接收请求，返回应答。

### 获取包类型（GetPacketType）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetPacketType() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回消息中对应的消息类型，0表示请求，1表示应答。

**用法说明：**

一般用于客户端收到消息之后，根据消息类型，来选择不同的处理逻辑。

### 设置营业部号（SetBranchNo）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetBranchNo(const int nBranchNo) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const int nBranchNo | 营业部号 | 输入 |

**返回：**

没有返回值

**用法说明：**

为消息设置一个营业部号，根据业务接口需要，来决定是否需要设置营业部号。

### 获取营业部号（GetBranchNo）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetBranchNo() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回消息中对应的营业部号。

**用法说明：**

### 设置系统号（SetSystemNo）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetSystemNo(const int nSystemNo) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const int nSystemNo | 系统号 | 输入 |

**返回：**

没有返回值

**用法说明：**

为消息设置一个系统号，根据业务接口需要，来决定是否需要设置系统号。系统号是指消息应该属于哪个系统处理，譬如：融资融券系统、集中交易系统等等。后台都会对系统编号，前台就根据后台规定传入规定的编号。这样才能保证业务能被后台系统处理。

### 获取系统号（GetSystemNo）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetSystemNo() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回消息中对应的系统号。

**用法说明：**

### 设置子系统号（SetSubSystemNo）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetSubSystemNo(const int nSubSystemNo) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const int nSubSystemNo | 子系统号 | 输入 |

**返回：**

没有返回值

**用法说明：**

为消息设置一个子系统号，根据业务接口需要，来决定是否需要设置子系统号。子系统号是系统号下面的详细分类，譬如：集中交易系统下面会分为，账户管理、交易、资金、历史、查询等等各个子系统。这些子系统也是由后台统一划分，前台根据接口说明传入对应的编号。

### 获取子系统号（GetSubSystemNo）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetSubSystemNo() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回消息中对应的子系统号。

**用法说明：**

### 设置发送者编号（SetSenderId）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetSenderId(const int nSenderId) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const int nSenderId | 发送者编号 | 输入 |

**返回：**

没有返回值

**用法说明：**

为消息设置一个发送者编号，发送者编号用于上层业务在异步模式下，匹配请求和应答，这个字段在请求中有上层业务系统自己送入，后台应答会原样返回。 也可以用在选择哪个线程处理，获取其他的应用中都可以。

### 获取发送者编号（GetSenderId）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetSenderId() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回消息中对应的发送者编号。

**用法说明：**

根据这个编号可以来匹配请求，也可以选择相应线程来处理。

### 设置包序号（SetPacketId）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetPacketId(const int nPacketId) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const int nPacketId | 包序号 | 输入 |

**返回：**

没有返回值

**用法说明：**

为消息设置一个包序号，每个消息都有一个自己序号，上层应用可以自己设置这个包序号，然后应答根据包序号来匹配请求。如果上层应用没有设置包序号，开发包底层业务设置这个序号，底层设置的序号是连续递增的。一般如果没有特殊需要，这个包序号是不需要上层业务自己添加的。

### 获取包序号（GetPacketId）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetPacketId() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回消息中对应的包序号。

**用法说明：**

### 设置目的地路由（SetTargetInfo）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetTargetInfo(const BIZROUTE\_INFO targetInfo) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const BIZROUTE\_INFO targetInfo | 目的地路由信息 | 输入 |

**返回：**

没有返回值

**用法说明：**

如果知道这条消息应该去那个中间件节点处理，可以调用这个函数设置中间件的路由信息，路由信息结构体中具体的字段定义如下：

typedef struct tagBizRouteInfo

{

char ospfName[ID\_STR\_LEN+1];//路由目标节点中间件名字

char nbrName[ID\_STR\_LEN+1];//中间件节点的邻居名字

char svrName[SVRINSTANCE\_NAME\_LENGTH+1];//中间件的进程名字

char pluginID[PLUGIN\_NAME\_LENGTH+1];//中间件插件名

int connectID;//连接号

int memberNO;//成员编号

}BIZROUTE\_INFO;

### 获取目的地路由（GetTargetInfo）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE GetTargetInfo(BIZROUTE\_INFO& targetInfo) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| BIZROUTE\_INFO& targetInfo | 消息中的目标路由信息 | 输出 |

**返回：**

返回消息中对应的目标路由。

**用法说明：**

一般用于客户端在获取应答的目标路由时使用。

### 设置发送者路由（SetSendInfo）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetSendInfo(const BIZROUTE\_INFO sendInfo) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const BIZROUTE\_INFO sendInfo | 发送者路由信息 | 输入 |

**返回：**

没有返回值

**用法说明：**

这个函数传入参数和设置路由信息的参数是一致，主要用于客户端在收到请求之后，发送应答时可以设置自己的路由信息作为发送者信息，便于服务端知道应答是由哪个客户端发起的。

### 获取发送者路由（GetSendInfo）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE GetSendInfo(BIZROUTE\_INFO& sendInfo) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| BIZROUTE\_INFO& sendInfo | 消息中的发送者路由信息 | 输出 |

**返回：**

返回消息中对应的发送者路由。

**用法说明：**

一般用于客户端在获取应答的发送者路由时使用。

### 设置错误号（SetErrorNo）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetErrorNo(const int nErrorNo) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const int nErrorNo | 具体的错误号 | 输入 |

**返回：**

没有返回值

**用法说明：**

这个函数用于客户端处理了请求之后，返回相应的错误号信息，便于请求发起方知道具体的错误原因，错误号：0表示成功，1，-1这两个错误号被中间件占用，上层的业务如果需要设置错误号，不可以使用1，-1。

### 获取错误号（GetErrorNo）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetErrorNo() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回消息中对应的错误号信息。

**用法说明：**

一般用于客户端在收到应答之后，根据错误号来判断之前的请求是否正确处理。

### 设置错误信息（SetErrorInfo）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetErrorInfo(const char\* strErrorInfo) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char\* strErrorInfo | 具体的错误信息 | 输入 |

**返回：**

没有返回值

**用法说明：**

这个函数用于客户端处理了请求之后，根据错误号设置相应的错误信息，便于对方收到错误号的时候，还可以显示相应的错误信息。

### 获取错误信息（GetErrorInfo）

**函数原型：**

virtual const char\* FUNCTION\_CALL\_MODE GetErrorInfo() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回消息中对应的错误信息。

**用法说明：**

一般用于客户端在收到应答之后，获取错误号对应的错误信息。

### 设置返回码（SetReturnCode）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetReturnCode(const int nReturnCode) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const int nReturnCode | 具体的返回码 | 输入 |

**返回：**

没有返回值

**用法说明：**

这个函数用于客户端处理了请求之后，返回相应的返回码，一般与错误码一致。

### 获取返回码（GetReturnCode）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetReturnCode() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回消息中对应的返回码信息。

**用法说明：**

一般用于客户端在收到应答之后，获取对应返回码，做不同的处理。

### 设置业务内容（SetContent）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetContent(void\* lpContent,int iLen) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| void\* lpContent | 具体的业务二进制内容 | 输入 |
| int iLen | 上面二进制内容的长度 | 输入 |

**返回：**

没有返回值

**用法说明：**

这个函数用于设置业务二进制的内容，一般业务请求或者应答都是打包解包的结构。下面是该函数的使用例子：

IF2Packer\* pack = NewPacker(2);

pack->AddRef();

pack->BeginPack();

pack->AddField("plugin\_id");

pack->AddField("function\_id");

pack->AddStr("ospf");

pack->AddInt(3);

pack->EndPack();

//把业务包打入业务消息 lpBizMessage->SetContent(pack->GetPackBuf(),pack->GetPackLen());

### 获取业务内容（GetContent）

**函数原型：**

virtual const void\* FUNCTION\_CALL\_MODE GetContent(int& iLen) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int& iLen | 获取业务内容的长度 | 输出 |

**返回：**

返回消息中对应的业务内容的指针，长度由输出参数返回。

**用法说明：**

一般用于客户端在收到应答之后，从应答中获取对应的业务内容，用解包器解包。如下所示：

int iLen = 0;

const void \* lpBuffer = lpBizMessageRecv->GetContent(iLen);

IF2UnPacker \* lpUnPacker = NewUnPacker((void \*)lpBuffer,iLen);

### 设置订阅类型（SetIssueType）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetIssueType(const int nIssueType) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const int nIssueType | 订阅类型 | 输入 |

**返回：**

没有返回值

**用法说明：**

这个函数用于向消息中心1.0发起订阅，设置订阅类型，原来消息中心1.0的订阅类型是用数字表示的，具体什么类型对应哪个数字，可以参考消息中心1.0的配置，如下所示：

<issuetype note="允许发布类型">

<type **value="0"** superior="0" note="文件更新">

<key name="update\_type"/>

</type>

<type **value="1"** superior="0" credibility="0" note="系统公告">

<key name="branch\_no"/>

<key name="operator\_no"/>

</type>

。。。。。。。。。。。。。。。。。。。。。。。。。

### 获取订阅类型（GetIssueType）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetIssueType() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回消息中对应的订阅类型。

**用法说明：**

一般用于客户端在收到订阅应答或者发布消息的时候，获取对应消息的订阅类型。

### 设置订阅序号（SetSequeceNo）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetSequeceNo(const int nSequeceNo) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const int nSequeceNo | 订阅序号 | 输入 |

**返回：**

没有返回值

**用法说明：**

这个函数用于向消息中心1.0发起订阅，设置序号，不同的序号表示不同的客户端，一般没有特殊要求，都不需要设置。

### 获取订阅序号（GetSequeceNo）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetSequeceNo() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回消息中对应的订阅序号。

**用法说明：**

一般用于客户端在收到订阅应答或者发布消息的时候，获取对应消息的订阅序号。

### 设置订阅关键字段（SetKeyInfo）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetKeyInfo(void\* lpKeyData,int iLen) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| void\* lpKeyData | 关键字段的二进制内容 | 输入 |
| int iLen | 二进制对应的长度 | 输入 |

**返回：**

没有返回值

**用法说明：**

这个函数用于向消息中心1.0发起订阅，设置相应的关键字段，服务端根据这个关键字段，向客户端推消息中和关键字段相符的消息。关键字段的一般用打包器的方式打包，一个订阅类型中包含哪些关键字段也是由消息中心1.0配置文件决定，订阅相应的类型的时候，必须指定所有关键字段的取值。如下所示：

pack->BeginPack();

pack->AddField("branch\_no", 'S');

pack->AddField("fund\_account", 'S');

pack->AddStr("8888");

pack->AddStr("20130107");

pack->EndPack();

lpBizMessage-> SetKeyInfo(pack->GetPackBuf(),pack->GetPackLen());

### 获取订阅关键字段（GetKeyInfo）

**函数原型：**

virtual const void\* FUNCTION\_CALL\_MODE GetKeyInfo(int& iLen) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int& iLen | 返回二进制的长度 | 输出 |

**返回：**

返回消息中对应的关键字段的二进制内容。

**用法说明：**

一般用于客户端在收到订阅应答或者发布消息的时候，获取对应消息的关键字段，一般用解包器进行解包，获取相应的关键字段的取值。

### 设置附加数据（SetAppData）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetAppData(const void\* lpAppdata,int nAppLen) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const void\* lpAppdata | 附加数据的二进制内容 | 输入 |
| int nAppLen | 二进制对应的长度 | 输入 |

**返回：**

没有返回值

**用法说明：**

这个函数用于向消息中心1.0发起订阅，设置相应的附加数据，附加数据在消息推送的时候会原样返回给客户端，一般附加数据存放的是客户端的窗口句柄，用于区分不同窗口的订阅。附加数据的格式没有规定形式，可以任意形式存放。

### 获取附加数据（GetAppData）

**函数原型：**

virtual const void\* FUNCTION\_CALL\_MODE GetAppData(int& nAppLen) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int& nAppLen | 返回二进制的长度 | 输出 |

**返回：**

返回消息中对应的附加数据的二进制内容。

**用法说明：**

一般用于客户端在收到发布消息的时候，获取对应消息的附加数据字段，用来匹配相应的窗口，调用不同的窗口处理发布消息。

### 请求转换成应答（ChangeReq2AnsMessage）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE ChangeReq2AnsMessage() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

转换成功返回0，失败返回其他值。

**用法说明：**

这个函数是从请求转成应答，用于客户端收到请求之后，转换成应答，返回给请求方。

### 消息转换成二进制流（GetBuff）

**函数原型：**

virtual void\* FUNCTION\_CALL\_MODE GetBuff(int& nBuffLen) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int& nBuffLen | 返回二进制的长度 | 输出 |

**返回：**

返回二进制指针。

**用法说明：**

这个函数用于把消息转换成二进制数据流，用于网络上的传输。

### 二进制流转换成消息（SetBuff）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE SetBuff(const void\* lpBuff,int nBuffLen) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const void\* lpBuff | 二进制数据指针 | 输入 |
| int nBuffLen | 数据长度 | 输入 |

**返回：**

成功返回0，失败返回其他值。

**用法说明：**

这个函数把二进制数据流转换成消息对象，便于消息对象的接口调用获取需要的内容。

* 1. 打包器接口【IF2Packer】

打包器版本一共分为两种，V1.0和V2.0。V1.0是针对原来老版本的打包器，兼容原有的协议。V2.0是后来新使用的。客户端传入何种版本的打包器，是由后台服务决定。

打包器由多个结果集组成，一个结果集里面有自己的返回值，名字，列，行等数据，每个结果集里面会有多条记录。V1.0的打包器不支持多结果，只支持单结果集。

### 设置缓存区空间（SetBuffer）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetBuffer(void \* pBuf,int iBufSize,int iDataLen=0 )=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| void \* pBuf | 缓冲区地址 | 输入 |
| int iBufSize | 缓冲区空间 | 输入 |
| int iDataLen=0 | 已有数据长度，新增数据加在已有数据之后（只对V1.0格式的包有效） | 输入 |

**返回：**

没有返回值

**用法说明：**

本函数主要用设置打包器的空间，由外部传入一个指定的空间。一般情况下，都是内容分配空间。如果传入外部空间，那么这个空间由外部维护，最后不可以调用FreeMem来释放空间。第三个参数只针对V1.0的打包器有效。

### 开始打包（BeginPack）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE BeginPack()=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

没有返回值

**用法说明：**

开始打包，在打包之前一定要先调用这个函数，表明下面是开始打字段和值。同时这个函数会清空打包缓存，之前的打包数据会全部清空。

### 构造新结果集（NewDataset）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE NewDataset(const char \*szDatasetName, int iReturnCode = 0)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \*szDatasetName | 结果的名字 | 输入 |
| int iReturnCode = 0 | 结果集的返回值 | 输入 |

**返回：**

成功返回0，失败返回其他值。

**用法说明：**

在打单结果集的包时，可以不调用本方法,均取默认值，打多结果集的时候，需要先调用这个函数。这个函数只有V2.0打包器支持。

### 添加新字段（AddField）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE AddField(const char \*szFieldName,char cFieldType ='S',int iFieldWidth=255,int iFieldScale=4)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \*szFieldName | 字段名字 | 输入 |
| char cFieldType ='S' | 字段类型 | 输入 |
| int iFieldWidth=255 | 字段长度 | 输入 |
| int iFieldScale=4 | 字段精度 | 输入 |

**返回：**

负数表示失败，否则为目前包的长度

**用法说明：**

此函数用于在结果集里面增加一个列，列类型支持五种：I整数，F浮点数，C字符，S字符串，R任意二进制数据；字段长度是指当前字段最大长度，在添加值的时候，超出最大长度，就会截断；字段精度只用于类型为F的时候。

### 添加string型数据（AddStr）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE AddStr(const char \*szValue)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \*szValue | 字符串数据 | 输入 |

**返回：**

负数表示失败，否则为目前包的长度

**用法说明：**

此函数用于添加字符串数据，针对的是字符串类型的列。

### 添加int型数据（AddInt）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE AddInt(int iValue)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int iValue | 整型数据 | 输入 |

**返回：**

负数表示失败，否则为目前包的长度

**用法说明：**

此函数用于添加整型数据，针对的是整数类型的列。

### 添加double型数据（AddDouble）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE AddDouble(double fValue)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| double fValue | 浮点数数据 | 输入 |

**返回：**

负数表示失败，否则为目前包的长度

**用法说明：**

此函数用于添加浮点数数据，针对的是浮点数类型的列。

### 添加char型数据（AddChar）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE AddChar(char cValue)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| char cValue | 字符数据 | 输入 |

**返回：**

负数表示失败，否则为目前包的长度

**用法说明：**

此函数用于添加字符数据，针对的是字符类型的列。

### 添加二进制数据（AddRaw）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE AddRaw(void \* lpBuff,int iLen)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| void \* lpBuff | 二进制数据 | 输入 |
| int iLen | 数据长度 | 输入 |

**返回：**

负数表示失败，否则为目前包的长度

**用法说明：**

此函数用于添加二进制数据，针对的是二进制类型的列。只有V2.0版本打包器才支持二进制列。

### 结束打包（EndPack）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE EndPack()=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

没有返回值

**用法说明：**

此函数用于最后结束打包，打包完之后必须调用此函数。

### 获取打包结果指针（GetPackBuf）

**函数原型：**

virtual void \* FUNCTION\_CALL\_MODE GetPackBuf() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

二进制数据首地址

**用法说明：**

此函数是返回打包器的二进制的数据流首地址。把打包器结构转换为二进制数据。

### 获取打包结果长度（GetPackLen）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetPackLen() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

有效数据的长度

**用法说明：**

此函数返回打包器转换为二进制之后的有效数据长度。配合上面的GetPackBuf使用。

### 获取打包缓存长度（GetPackBufSize）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetPackBufSize() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

二进制数据的缓存区长度

**用法说明：**

此函数返回打包器转换为二进制之后的缓存区长度。配合上面的GetPackBuf使用。

### 获取打包版本（GetVersion）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetVersion(void) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回版本信息，1表示版本v1.0，其他值表示为v2.0

**用法说明：**

获取打包器版本信息。

### 设置返回码（SetReturnCode）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetReturnCode(unsigned long dwRetCode) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| unsigned long dwRetCode | 返回码 | 输入 |

**返回：**

没有返回值

**用法说明：**

此函数用于为当前结果集设置返回码，返回码取缺省值0，则不设置，如果设置，则必须在EndPack()之前调用。

### 获取解包接口（UnPack）

**函数原型：**

virtual IF2UnPacker \* FUNCTION\_CALL\_MODE UnPack() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

当前打包器的解包接口，失败返回NULL指针

**用法说明：**

必须在EndPack()之后才能调用,在打包器释放时相应的解包器实例也释放。

### 释放打包内存（FreeMem）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE FreeMem(void\* lpBuf) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| void\* lpBuf | 缓存指针 | 输入 |

**返回：**

没有返回值。

**用法说明：**

该函数用于释放内部的空间，需要显示调用这个函数。如下所示：

pack->FreeMem(pack->GetPackBuf());

### 清空最后一个结果集数据（ClearValue）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE ClearValue() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

没有返回值。

**用法说明：**

该函数用于删除当前打包器的最后一个结果集，清除最后一个结果集的数据，这样再次添加数据而不必添加列信息，以便提高性能。

### 开始打包扩展接口（BeginPackEx）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE BeginPackEx(char\* szName = NULL) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| char\* szName = NULL | 结果集名字 | 输入 |

**返回：**

没有返回值。

**用法说明：**

该函数用于为第一个默认结果增加名字，是BeginPack的扩展函数。

### 清空当前结果集（ClearDataSet）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE ClearDataSet() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

没有返回值。

**用法说明：**

清空当前结果集，可以重新打入新的字段，新的值。

* 1. 结果集接口【IF2ResultSet】

这里结果集是一个行列结果的数据结构，类似数据库的表。

### 获取列个数（GetColCount）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetColCount()=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回结果集中列的个数。

**用法说明：**

用于获取结果集中列的个数，可以根据列来遍历结果集数据。

### 根据下标获取列名字（GetColName）

**函数原型：**

virtual const char \* FUNCTION\_CALL\_MODE GetColName(int column)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int column | 列下标 | 输入 |

**返回：**

返回列的名字。下标越界返回NULL。

**用法说明：**

根据传入的列下标获取对应的列名字，可以结合GetColCount函数来遍历所有的列名字。

### 获取列类型（GetColType）

**函数原型：**

virtual char FUNCTION\_CALL\_MODE GetColType(int column)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int column | 列下标 | 输入 |

**返回：**

返回列的类型。下标越界默认返回S。

**用法说明：**

根据传入的列下标获取对应的列类型，列类型返回值就五种：I整数，F浮点数，C字符，S字符串，R任意二进制数据。

### 获取列精度（GetColScale）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetColScale(int column)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int column | 列下标 | 输入 |

**返回：**

返回列的精度。下标越界默认返回0。

**用法说明：**

double类型的列才有精度，其他类型的列精度设置了也没有用。

### 获取列最大长度（GetColWidth）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetColWidth(int column) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int column | 列下标 | 输入 |

**返回：**

返回列的最大长度。下标越界默认返回1。

**用法说明：**

### 根据列名字找到列下标（FindColIndex）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE FindColIndex(const char \* columnName)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \* columnName | 列名字 | 输入 |

**返回：**

返回列的下标。不存在返回-1

**用法说明：**

根据列的名字获取对应的下标，然后可以根据下标获取对应的值。也可以用这个函数判断是否存在对应的列。

### 根据列下标获取string类型值（GetStrByIndex）

**函数原型：**

virtual const char \* FUNCTION\_CALL\_MODE GetStrByIndex(int column)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int column | 列下标 | 输入 |

**返回：**

返回当前行记录中对应的列下标的值，以string形式返回。下标越界返回NULL。

**用法说明：**

请确保当前列类型为’S’，才用这个函数获取，否则数据可能不准确。

### 根据列名字获取string类型值（GetStr）

**函数原型：**

virtual const char \* FUNCTION\_CALL\_MODE GetStr(const char \* columnName)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \* columnName | 列名 | 输入 |

**返回：**

返回当前行记录中对应列名的列值，以string形式返回。名字不存在返回NULL。

**用法说明：**

请确保当前列类型为’S’，才用这个函数获取，否则数据可能不准确。

### 根据列下标获取char类型值（GetCharByIndex）

**函数原型：**

virtual char FUNCTION\_CALL\_MODE GetCharByIndex(int column)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int column | 列下标 | 输入 |

**返回：**

返回当前行记录中对应列下标的值，以char形式返回。下标越界返回字符’\0’。

**用法说明：**

请确保当前列类型为’C’，才用这个函数获取，否则数据可能不准确。

### 根据列名字获取char类型值（GetChar）

**函数原型：**

virtual char FUNCTION\_CALL\_MODE GetChar(const char \* columnName)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \* columnName | 列名 | 输入 |

**返回：**

返回当前行记录中对应列名的列值，以char形式返回。名字不存在返回’\0’。

**用法说明：**

请确保当前列类型为’C’，才用这个函数获取，否则数据可能不准确。

### 根据列下标获取double类型值（GetDoubleByIndex）

**函数原型：**

virtual double FUNCTION\_CALL\_MODE GetDoubleByIndex( int column)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int column | 列下标 | 输入 |

**返回：**

返回当前行记录中对应列下标的值，以double形式返回。下标越界返回0。

**用法说明：**

请确保当前列类型为’F’，才用这个函数获取，否则数据可能不准确。

### 根据列名字获取double类型值（GetDouble）

**函数原型：**

virtual double FUNCTION\_CALL\_MODE GetDouble(const char \* columnName)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \* columnName | 列名 | 输入 |

**返回：**

返回当前行记录中对应列名的列值，以double形式返回。名字不存在返回0。

**用法说明：**

请确保当前列类型为’F’，才用这个函数获取，否则数据可能不准确。

### 根据列下标获取int类型值（GetIntByIndex）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetIntByIndex(int column)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int column | 列下标 | 输入 |

**返回：**

返回当前行记录中对应列下标的值，以int形式返回。下标越界返回0。

**用法说明：**

请确保当前列类型为’I’，才用这个函数获取，否则数据可能不准确。

### 根据列名字获取int类型值（GetInt）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetInt(const char \* columnName)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \* columnName | 列名 | 输入 |

**返回：**

返回当前行记录中对应列名的列值，以int形式返回。名字不存在返回0。

**用法说明：**

请确保当前列类型为’I’，才用这个函数获取，否则数据可能不准确。

### 根据列下标获取二进制类型值（GetRawByIndex）

**函数原型：**

virtual void \* FUNCTION\_CALL\_MODE GetRawByIndex(int column,int \* lpRawLen) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int column | 列下标 | 输入 |
| int \* lpRawLen | 二进制数据长度 | 输出 |

**返回：**

返回当前行记录中对应列下标的值，以二进制形式返回。下标越界返回NULL。

**用法说明：**

请确保当前列类型为’R’，才用这个函数获取，其他类型的列调用这个函数获取都是NULL。

### 根据列名字获取二进制类型值（GetRaw）

**函数原型：**

virtual void \* FUNCTION\_CALL\_MODE GetRaw(const char \* columnName,int \* lpRawLen) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \* columnName | 列名 | 输入 |
| int \* lpRawLen | 二进制数据长度 | 输出 |

**返回：**

返回当前行记录中对应列名的列值，以二进制形式返回。名字不存在返回NULL。

**用法说明：**

请确保当前列类型为’R’，才用这个函数获取，其他类型的列调用这个函数获取都是NULL。

### 最后一次取得值是否为空（WasNull）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE WasNull()=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

0表示是空，1表示不是空。

**用法说明：**

用来判断最后一次取得列值是不是空。可以结合前面的函数，一起判断当前列获取默认值是原来的值， 还是空值。

### 结果集下一条记录（Next）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE Next()=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

没有返回值

**用法说明：**

用于遍历使用，当前记录所有列的值都获取完了，就可以移到一条记录。

### 结果集是否到结尾（IsEOF）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE IsEOF()=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

0表示没有到结果集结尾；1表示到了结果集结尾。

**用法说明：**

在遍历的时候，判断当前结果集的所有记录是不是已经遍历完了，结合Next函数一起使用。遍历代码如下：

while(!unpack-> IsEOF())

{

……..//读列值

unpack->Next();

}

### 结果集是否为空（IsEmpty）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE IsEmpty()=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

0表示结果集记录数不为0；1表示结果集记录数为0。

**用法说明：**

在遍历之前可以先做记录数个数判断，为空就不需要遍历了。

### 结果集释放（Destroy）

**函数原型：**

virtual void \* FUNCTION\_CALL\_MODE Destroy() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

无返回值

**用法说明：**

结果集释放函数，一般都是调用Release接口释放，不使用这个函数。这个函数功能和Release接口一样。

* 1. 解包器接口【IF2UnPacker】

此接口是继承自上面结果集接口，实现所有结果集接口之外，支持多结果集解析。

### 获取解包器版本（GetVersion）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetVersion() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回解包器版本，1表示是V1.0版本，33表示V2.0版本。

**用法说明：**

### 二进制转换为解包器（Open）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE Open(void \* lpBuffer,unsigned int iLen) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| void \* lpBuffer | 二进制数据指针 | 输入 |
| unsigned int iLen | 长度 | 输入 |

**返回：**

0表示解析成功，其他表示失败。

**用法说明：**

这个函数用客户端收到二进制数据之后，用解包器来解包，转换成解包器结构之后，便于获取行列数据。

### 获取结果集个数（GetDatasetCount）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetDatasetCount()=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回结果集个数。

**用法说明：**

V2.0的版本支持多结果集，如果是v1.0解包器，用于返回1。通过这个函数可以知道当前解包器有几个结果集，用于遍历所有结果集，获取相关数据。

### 根据下标设置当前结果集（SetCurrentDatasetByIndex）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE SetCurrentDatasetByIndex(int nIndex)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int nIndex | 结果集序号 | 输入 |

**返回：**

0表示成功，否则失败。设置的序号不可以超过上面函数GetDatasetCount值。

**用法说明：**

这个函数是把解包器设置到指定序号的结果集，然后就可以获取当前结果集的数据。结合上面的GetDatasetCount就可以遍历所有结果集数据。

### 根据结果集名字设置当前结果集（SetCurrentDataset）、

**函数原型：**

virtual int FUNCTION\_CALL\_MODE SetCurrentDataset(const char \*szDatasetName)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| const char \*szDatasetName | 结果集名字 | 输入 |

**返回：**

0表示成功，否则失败。如果对应的名字的结果集不存在，返回失败。

**用法说明：**

这个函数是把解包器设置到指定名字的结果集，如果结果集有名字，就可以用这个函数。

### 获取解包器缓存指针（GetPackBuf）

**函数原型：**

virtual void \* FUNCTION\_CALL\_MODE GetPackBuf() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回解包器内部的缓存指针。

**用法说明：**

这个缓存一般就是Open函数传入的指针。

### 获取解包器缓存长度（GetPackLen）

**函数原型：**

virtual unsigned int FUNCTION\_CALL\_MODE GetPackLen() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回解包器内部的缓存长度。

**用法说明：**

这个缓存一般就是Open函数传入的长度。

### 获取当前结果集行数（GetRowCount）

**函数原型：**

virtual unsigned int FUNCTION\_CALL\_MODE GetRowCount() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回当前结果集记录行数。

**用法说明：**

根据这个函数返回的行数，可以用下面Go函数直接指定行访问数据。

### 指向结果集第一行记录（First）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE First() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

无返回值。

**用法说明：**

指向当前结果集的首条记录。用于一个结果集记录的遍历开始。

### 指向结果集最后一行记录（Last）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE Last() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

无返回值。

**用法说明：**

指向当前结果集的最后一条记录。

### 跳转到结果集任意行（Go）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE Go(int nRow) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int nRow | 要跳转到的记录行 | 输入 |

**返回：**

无返回值。

**用法说明：**

取结果集的第n条记录，取值范围[1, GetRowCount()]，方便任意访问结果集中的具体某行。

* 1. 过滤器接口【CFilterInterface】

过滤条件接口用于向消息中心2.0订阅的时候，传入的过滤信息。用于上层程序根据自己需要做相关订阅。

### 根据下标获取条件名字（GetFilterNameByIndex）

**函数原型：**

virtual char\* FUNCTION\_CALL\_MODE GetFilterNameByIndex(int index) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int index | 下标 | 输入 |

**返回：**

返回对应下标的过滤条件名字，没有就返回NULL。

**用法说明：**

根据下标获取过滤条件名字，下标的取值范围在0，GetCount之间。

### 根据下标获取条件值（GetFilterValueByIndex）

**函数原型：**

virtual char\* FUNCTION\_CALL\_MODE GetFilterValueByIndex(int index)= 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int index | 下标 | 输入 |

**返回：**

返回对应下标的过滤条件值，没有就返回NULL。

**用法说明：**

根据下标获取过滤条件值，下标的取值范围在0，GetCount之间。

### 根据条件名字好获取条件值（GetFilterValue）

**函数原型：**

virtual char\* FUNCTION\_CALL\_MODE GetFilterValue(char\* fileName)= 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| char\* fileName | 过滤条件名字 | 输入 |

**返回：**

根据过滤条件名字返回对应的值，不存在就返回NULL。

**用法说明：**

如果知道过滤条件的名字就可以根据这个函数获取值。

### 获取过滤条件个数（GetCount）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetCount() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回过滤条件的个数。

**用法说明：**

这个函数可以获取过滤条件的个数，根据个数，按照上面下标获取方式可以遍历所有的过滤条件名字和值。

### 设置过滤条件名字和值（SetFilter）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetFilter(char\* filterName,char\* filterValue) =0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| char\* filterName | 过滤条件名字 | 输入 |
| char\* filterValue | 过滤条件值 | 输入 |

**返回：**

无

**用法说明：**

这个函数是往过滤条件结构中添加一个过滤条件名字和值对应关系。用于订阅的时候设置过滤条件的取值。

* 1. 订阅参数接口【CSubscribeParamInterface】

订阅参数接口，用于向消息中心2.0订阅的时候，传入所有和订阅相关的参数。

### 设置主题名字（SetTopicName）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetTopicName(char\* szName) =0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| char\* szName | 主题名字 | 输入 |

**返回：**

无

**用法说明：**

订阅必须存在主题，如果订阅参数没有主题，订阅就会失败。主体必须是后台消息中心必须存在的主题，否则会报主题不存在的错误。

### 设置附加数据（SetAppData）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetAppData(void\* lpData,int iLen)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| void\* lpData | 二进制数据首地址 | 输入 |
| int iLen | 数据长度 | 输入 |

**返回：**

无

**用法说明：**

附加数据是订阅的时候传入，发布消息的时候，服务端会原样返回，一般用于传递窗口句柄，便于发布消息回来的时候找到对应的窗口处理。

### 设置过滤条件（SetFilter）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetFilter(char\* filterName,char\* filterValue)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| char\* filterName | 过滤条件名字 | 输入 |
| char\* filterValue | 过滤条件值 | 输入 |

**返回：**

无

**用法说明：**

设置订阅的过滤条件，和前面的过滤条件接口是一样，只不过在订阅参数里面集成了，方便上层调用。

### 设置返回字段（SetReturnFiled）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetReturnFiled(char\* filedName)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| char\* filedName | 返回字段的名字 | 输入 |

**返回：**

无

**用法说明：**

订阅的时候，可以先设置好需要返回的字段，譬如，只关心发布消息中的“branchno,fundaccut”两个字段，那么就可以在订阅时候，调用下面的代码：

lpSubscribeParam->SetReturnFiled(“branchno”);

lpSubscribeParam->SetReturnFiled(“fundaccut”);

这样订阅之后，收到推送过来的消息就只有这两个字段的值。

### 设置补缺标志（SetFromNow）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetFromNow(bool bFromNow)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| bool bFromNow | False表示不补缺，默认值  True表示需要补缺 | 输入 |

**返回：**

无

**用法说明：**

补缺，就是是否需要不在线期间这个主题发布的消息，当然前提是这个主题的可靠级别是文件级才行。主要用于某些重要的通知消息，如果订阅端没有上线，就不能实时接收到，等下次客户端订阅的时候，可以通过补缺来接收之前的消息，防止消息丢失。

### 设置覆盖标志（SetReplace）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetReplace(bool bReplace)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| bool bReplace | False表示不覆盖，默认值  True表示需要覆盖 | 输入 |

**返回：**

无

**用法说明：**

覆盖，就是覆盖之前所有的订阅，譬如，一个客户端之前做了很多订阅，从某一个时刻开始，只要当前订阅，之前订阅都不需要了，那么就可以在当前订阅发起的时候设置这个参数，就无需取消之前所有的订阅，方便操作。

### 设置发送间隔（SetSendInterval）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE SetSendInterval(int nSendInterval)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int nSendInterval | 发送间隔，单位秒 | 输入 |

**返回：**

无

**用法说明：**

发送间隔，就是服务端推送消息给客户端的频率。譬如：客户端订阅了行情，但是又不想要每一笔行情，只要5秒能收到一笔就可以，这样就可以设置这个参数，那么客户端就会每5秒接收到一笔行情。

### 获取主题名字（GetTopicName）

**函数原型：**

virtual char\* FUNCTION\_CALL\_MODE GetTopicName() =0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回订阅参数中的主题名字。

**用法说明：**

### 获取附加数据（GetAppData）

**函数原型：**

virtual void\* FUNCTION\_CALL\_MODE GetAppData(int \*iLen) =0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int \*iLen | 附加数据长度 | 输出 |

**返回：**

返回订阅参数中的附加数据指针，没有返回NULL。

**用法说明：**

### 根据下标获取过滤字段名字（GetFilterNameByIndex）

**函数原型：**

virtual char\* FUNCTION\_CALL\_MODE GetFilterNameByIndex(int index) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int index | 过滤条件下标 | 输入 |

**返回：**

返回订阅参数中对应下标的过滤字段名字。

**用法说明：**

这个函数类似于前面的过滤条件的接口。

### 根据下标获取过滤字段值（GetFilterValueByIndex）

**函数原型：**

virtual char\* FUNCTION\_CALL\_MODE GetFilterValueByIndex(int index)= 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int index | 过滤条件下标 | 输入 |

**返回：**

返回订阅参数中对应下标的过滤字段值。

**用法说明：**

这个函数类似于前面的过滤条件的接口。

### 根据过滤名字获取值（GetFilterValue）

**函数原型：**

virtual char\* FUNCTION\_CALL\_MODE GetFilterValue(char\* fileName)= 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| char\* fileName | 过滤条件名字 | 输入 |

**返回：**

返回对应过滤条件名字的值。

**用法说明：**

这个函数类似于前面的过滤条件的接口。

### 获取过滤条件个数（GetFilterCount）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetFilterCount() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回对应过滤条件个数。

**用法说明：**

这个函数类似于前面的过滤条件的接口。

### 获取返回字段（GetReturnFiled）

**函数原型：**

virtual char\* FUNCTION\_CALL\_MODE GetReturnFiled()=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回订阅参数的返回字段。没有返回NULL。

**用法说明：**

多个返回字段，用’\2’符号隔开，譬如” branchno\2fundaccut”

### 获取补缺标志（GetFromNow）

**函数原型：**

virtual bool FUNCTION\_CALL\_MODE GetFromNow()=0 ;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回补缺标志。

**用法说明：**

### 获取覆盖标志（GetReplace）

**函数原型：**

virtual bool FUNCTION\_CALL\_MODE GetReplace() =0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回覆盖标志。

**用法说明：**

### 获取发送间隔（GetSendInterval）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetSendInterval()=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回发布间隔。

**用法说明：**

* 1. 订阅回调接口【CSubCallbackInterface】

订阅回调的接口，一般是上层实现回调接口的函数，在接收到发布的消息之后，会回调接口给上层应用，这是上层应用程序唯一获取发布消息的接口。

### 收到发布消息回调（OnReceived）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE OnReceived(CSubscribeInterface \*lpSub,int subscribeIndex, const void \*lpData, int nLength,LPSUBSCRIBE\_RECVDATA lpRecvData) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CSubscribeInterface \*lpSub | 当前回调订阅接口指针 | 回调 |
| int subscribeIndex | 当前回调对应订阅标识 | 回调 |
| const void \*lpData | 接收到发布内容指针 | 回调 |
| int nLength | 发布内容的长度 | 回调 |
| LPSUBSCRIBE\_RECVDATA lpRecvData | 订阅相关信息 | 回调 |

**返回：**

无

**用法说明：**

回调的结构体包含内容如下：

typedef struct tagSubscribeRecvData

{

char\* lpFilterData; /\*\*< 过滤字段的数据头指针，用解包器解包 \*/

int iFilterDataLen; /\*\*< 过滤字段的数据长度 \*/

char\* lpAppData; /\*\*< 附加数据的数据头指针 \*/

int iAppDataLen; /\*\*< 附加数据的长度 \*/

char szTopicName[260]; /\*\*< 主题名字 \*/

}SUBSCRIBE\_RECVDATA, \*LPSUBSCRIBE\_RECVDATA;

这个回调会在收到发布消息之后回调进去，上层应用在这个函数里面做业务处理。回调参数里面，还支持当前消息属于哪个订阅，以及过滤条件是什么，附加数据是什么，是什么主题等信息，都返回给了上层函数实现者。

### 收到剔除订阅回调（OnRecvTickMsg）

**函数原型：**

virtual void FUNCTION\_CALL\_MODE OnRecvTickMsg(CSubscribeInterface \*lpSub,int subscribeIndex,const char\* TickMsgInfo) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CSubscribeInterface \*lpSub | 当前回调的订阅接口指针 | 回调 |
| int subscribeIndex | 当前回调的对应订阅标识 | 回调 |
| const char\* TickMsgInfo | 剔除订阅的原因 | 回调 |

**返回：**

无

**用法说明：**

此回调会在订阅被踢出的时候调用，什么时候订阅会被服务端踢出，在主题设置了踢出前面订阅者的踢人策略的时候，前面已经订阅的订阅者就会收到这个回调。

* 1. 订阅接口【CSubscribeInterface】

这个接口主要负责订阅，取消订阅，接收发布消息回调给上层应用，接口实例化有连接接口里面的函数实现。

### 订阅主题

**函数原型：**

virtual int FUNCTION\_CALL\_MODE SubscribeTopic(CSubscribeParamInterface\* lpSubscribeParamInter ,unsigned int uiTimeout,IF2UnPacker\*\* lppBizUnPack=NULL,IF2Packer\* lpBizPack=NULL) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CSubscribeParamInterface\* lpSubscribeParamInter | 订阅传入的参数 | 输入 |
| unsigned int uiTimeout | 订阅超时时间，单位毫秒 | 输入 |
| IF2UnPacker\*\* lppBizUnPack=NULL | 业务校验失败返回解包器 | 输出 |
| IF2Packer\* lpBizPack=NULL | 业务校验需要传入信息 | 输入 |

**返回：**

返回订阅标识，大于0表示有效的订阅标识，否则其他表示错误。

**用法说明：**

订阅有时候需要业务校验，这层校验是服务端上进行了，除了校验订阅的主题和过滤条件之外，还可以进行业务层校验，需要业务人员在服务端上进行开发。业务校验需要的额外的参数就需要用lpBizPack这个参数传递进去；如果业务校验出错，具体的错误信息就会在lppBizUnPack返回，这里lppBizUnPack用法如下所示：

IF2UnPacker\* lpBizUnPack =NULL;

SubscribeTopic(...,&lpBizUnPack);

//最后根据返回值，如果是失败的就判断 lpBizUnPack 是不是NULL.

if(lpBizUnPack)

{

//最后错误信息获取完之后,释放

lpBizUnPack->Release();

}

### 取消订阅

**函数原型：**

virtual int FUNCTION\_CALL\_MODE CancelSubscribeTopic(int subscribeIndex) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int subscribeIndex | 订阅标识 | 输入 |

**返回：**

返回0表示取消订阅成功，返回其他值，根据错误号获取错误信息.

**用法说明：**

这里输入的订阅标识就是SubscribeTopic返回的值。

### 取消订阅扩展接口

**函数原型：**

virtual int FUNCTION\_CALL\_MODE CancelSubscribeTopicEx(char\* topicName,CFilterInterface\* lpFilterInterface) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| char\* topicName | 主题名字 | 输入 |
| CFilterInterface\* lpFilterInterface | 过滤条件 | 输入 |

**返回：**

返回0表示取消订阅成功，返回其他值，根据错误号获取错误信息.

**用法说明：**

这里输入的是主题名字和过滤条件，根据这个属性可以唯一确定一个订阅标识，如果没有记录之前订阅成功的标识的话，可以用这个接口取消订阅。最好是采用前面的CancelSubscribeTopic函数，效率会高。

### 获取已经订阅的信息

**函数原型：**

virtual void FUNCTION\_CALL\_MODE GetSubcribeTopic(IF2Packer\* lpPack)=0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| IF2Packer\* lpPack | 所有订阅信息打包器，外面申请，函数内部添加字段和值 | 输入 |

**返回：**

无返回值.

**用法说明：**

函数内部返回的打包器字段如下所示：

- SubcribeIndex 订阅标识，也就是SubscribeTopic函数返回值

- IsBornTopic 先天订阅标识，0表示不是先天订阅

- TopicName 主题名字

- TopicNo 主题编号

- FilterRaw 过滤字段信息

- Appdata 附加数据

- SendInterval 发送间隔

- ReturnFileds 返回字段

- isReplace 覆盖标识

- isFromNow 补缺标识

根据上面的字段获取相关的订阅信息。

* 1. 发布接口【CPublishInterface】

这个接口主要用于发布消息到消息中心2.0，然后推送给关心的客户端。发布的消息分两种，一种是采用打包器的行列方式的数据，另一种就是随意的二进制。

### 业务发包格式发送

**函数原型：**

virtual int FUNCTION\_CALL\_MODE PubMsgByPacker(char\* topicName ,IF2UnPacker\* lpUnPacker,int iTimeOut=-1,

IF2UnPacker\*\* lppBizUnPack=NULL,bool bAddTimeStamp=false) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| char\* topicName | 主题名字 | 输入 |
| IF2UnPacker\* lpUnPacker | 发布的业务包 | 输入 |
| int iTimeOut=-1 | 超时时间 | 输入 |
| IF2UnPacker\*\* lppBizUnPack=NULL | 业务校验失败返回 | 输出 |
| bool bAddTimeStamp=false | 是否添加时间戳 | 输入 |

**返回：**

返回0表示成功，返回其他值，根据错误号获取错误信息.

**用法说明：**

参数中解包器，是发布内容先用pack打包，然后调用unpack接口传入，代码如下：

IF2Packer\* lpOnePack = NewPacker(2);

lpOnePack->AddRef();

lpOnePack->BeginPack();

lpOnePack->AddField("branchno");

lpOnePack->AddStr("1000");

lpOnePack->EndPack();

IF2UnPacker\* lpUnPack = lpOnePack->UnPack();

int iRet = lpPublish->PubMsgByPacker(topicName,lpUnPack,5000);

注意的是，打的pack包里面必须要有这个主题的所有过滤字段信息，否则就会发布出错。

业务校验和订阅类似，不过不需要单独成传业务参数，只要接收错误信息即可，错误信息的处理和上面订阅接口一样。如下所示：

IF2UnPacker\* lpBizUnPack =NULL;

PubMsgByPacker(...,&lpBizUnPack);

//最后根据返回值，如果是失败的就判断 lpBizUnPack 是不是NULL.

if(lpBizUnPack)

{

//最后错误信息获取完之后,释放

lpBizUnPack->Release();

}

### 二进制数据发送

**函数原型：**

virtual int FUNCTION\_CALL\_MODE PubMsg(char\* topicName, CFilterInterface\* lpFilterInterface, const void \*lpData, int nLength,int iTimeOut=-1,

IF2UnPacker\*\* lppBizUnPack=NULL,bool bAddTimeStamp=false) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| char\* topicName | 主题名字 | 输入 |
| CFilterInterface\* lpFilterInterface | 过滤信息 | 输入 |
| const void \*lpData | 二进制数据首地址 | 输入 |
| int nLength | 二进制数据长度 | 输入 |
| int iTimeOut=-1 | 超时时间 | 输入 |
| IF2UnPacker\*\* lppBizUnPack=NULL | 业务校验失败返回 | 输出 |
| bool bAddTimeStamp=false | 是否添加时间戳 | 输入 |

**返回：**

返回0表示成功，返回其他值，根据错误号获取错误信息.

**用法说明：**

这个函数唯一不同就是，传入的数据不是pack结构的数据，是随意的二进制数据，指针和长度传入即可。因为不是pack的形式数据，所以需要单独指定过滤字段，当然过滤字段包含当前发布主题的所有过滤字段，表明当前消息属于什么过滤信息，这样才可以推送给关心这个消息的订阅端。

* 1. 连接接口【CConnectionInterface】

这个接口是主要的操作接口，上层通过这个接口收发数据，建立连接，进行发布订阅等等操作。

### 初始化连接对象（Create）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE Create(CCallbackInterface \*lpCallback) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CCallbackInterface \*lpCallback | 回调接口实现类指针 | 输入 |

**返回：**

返回0表示成功，否则表示失败，通过调用GetErrorMsg可以获取详细错误信息

**用法说明：**

如果应用不需要任何回调方法，则可向该方法传递NULL，而不必自定义回调类和对象。如果有回调实例，这个初始化的数据接收函数只会调用OnReceivedBiz函数。

### 开始连接/注册（Connect）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE Connect(unsigned int uiTimeout) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| unsigned int uiTimeout | 连接同步等待的超时时间 | 输入 |

**返回：**

返回0表示成功，否则表示失败，通过调用GetErrorMsg可以获取详细错误信息

**用法说明：**

此函数用于向后台进行建TCP连接和注册的过程，如果是密码级别的连接，还包括了密钥的交互，SSL级别的连接就包括了SSL的连接。不同级别连接，建立的时间都不一样，上层根据实际传入不同的连接超时时间。如果超时时间传入0，那么连接就会是异步的模式，上层就需要调用GetStatus函数判断连接是否已经注册成功，否则发包都会包“连接未注册”的错误。

### 断开连接（Close）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE Close() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回0表示成功，否则表示失败，通过调用GetErrorMsg可以获取详细错误信息

**用法说明：**

此函数用于主动断开和服务端的连接。

### 取连接服务器地址（GetServerAddress）

**函数原型：**

virtual const char \* FUNCTION\_CALL\_MODE GetServerAddress(int \*lpPort) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int \*lpPort | 服务端的端口 | 输出 |

**返回：**

返回服务器IP地址，参数返回是端口。

**用法说明：**

这个函数返回当前连接对应的服务端的IP地址和端口，在服务端地址有多个情况下，可以确定当前连接连的是哪个服务器。

### 取连接状态（GetStatus）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetStatus() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回服务器的连接状态。

**用法说明：**

连接的状态有以下几种：

///连接状态，可组合

enum ConnectionStatus

{

Disconnected = 0x0000, /\*\*< 未连接 \*/

Connecting = 0x0001, /\*\*< socket正在连接 \*/

Connected = 0x0002, /\*\*< socket已连接 \*/

SafeConnecting = 0x0004, /\*\*< 正在建立安全连接 \*/

SafeConnected = 0x0008, /\*\*< 已建立安全连接 \*/

Registering = 0x0010, /\*\*< 正注册 \*/

Registered = 0x0020, /\*\*< 已注册 \*/

Rejected = 0x0040 /\*\*< 被拒绝，将被关闭 \*/

};

连接状态是组合，如果需要判断一个连接的状态是否已经注册，代码如下所示：

if (lpConnection->m\_Status & Registered)

{

//连接已经注册

}

### 取服务器负载（GetServerLoad）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetServerLoad() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回服务器负载（非负数），越大表示越繁忙，否则表示失败，通过调用GetErrorMsg可以获取详细错误信息

**用法说明：**

使用者可以同时创建多个连接实例同时连接不同的服务器，根据完成连接的时间以及负载决定使用哪个服务器，负载指的是后台业务处理队列的积压情况。

### 取错误码对应的错误信息（GetErrorMsg）

**函数原型：**

virtual const char \* FUNCTION\_CALL\_MODE GetErrorMsg(int nErrorCode) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int nErrorCode | 错误码 | 输入 |

**返回：**

返回错误信息

**用法说明：**

根据错误码，返回错误，错误信息返回支持中文和英文两种格式。可以连接配置参数中设置语言。

### 取连接错误号（GetConnectError）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE GetConnectError() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回连接建立和注册过程出现的错误

**用法说明：**

当连接无法与服务端完成注册时，既可通过Connect的返回值获取错误号，也可通过调用本方法来获取错误号，然后用该错误号调用GetErrorMsg可获取可读的错误信息。

### 发送业务数据（SendBiz）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE SendBiz(int iFunID, IF2Packer \*lpPacker, int nAsy = 0, int iSystemNo = 0, int nCompressID = 1) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int iFunID | 功能号 | 输入 |
| IF2Packer \*lpPacker | 业务请求数据 | 输入 |
| int nAsy = 0 | 是否异步发送 | 输入 |
| int iSystemNo = 0 | 系统号 | 输入 |
| int nCompressID = 1 | 是否压缩 | 输入 |

**返回：**

返回发送句柄（正数），否则表示失败，通过调用GetErrorMsg可以获取详细错误信息。

**用法说明：**

当异步发送标志为0，就表示同步发送，同步发送的包，通过调用RecvBiz来接收；如果异步发送标志位1，就表示异步发包，当收到应答包后，自动触发回调函数OnReceivedBiz。这个函数需要配合Create这个初始化函数一起使用。

### 接收业务数据（RecvBiz）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE RecvBiz(int hSend, void \*\*lppUnPackerOrStr, unsigned uiTimeout = 1000, unsigned uiFlag = 0) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int hSend | 发送句柄，就是SendBiz返回值 | 输入 |
| void \*\*lppUnPackerOrStr | 返回的数据，根据函数返回值不一样，意义不一样 | 输出 |
| unsigned uiTimeout = 1000 | 接收数据的超时时间 | 输入 |
| unsigned uiFlag = 0 | 接收选项，接收超时之后是否删除对应发送句柄的数据。 | 输入 |

**返回：**

小于0表示RecvBiz操作本身失败，通过调用GetErrorMsg可以获取详细错误信息

**用法说明：**

下面针对lppUnPackerOrStr输出参数做详细说明：

* 如果返回值等于0，表示业务数据接收成功，并且业务操作成功，\*lppUnPackerOrStr指向一个解包器，此时应首先将该指针转换为IF2UnPacker \*。
* 如果返回值等于1，表示业务数据接收成功，但业务操作失败了，\*lppUnPackerOrStr指向一个解包器，此时应首先将该指针转换为IF2UnPacker \*。
* 如果返回值等于2，表示收到非业务错误信息，\*lppUnPackerOrStr指向一个可读的字符串错误信息。
* 如果返回值等于3，表示业务包解包失败。\*lppUnPackerOrStr原先所指向的内容不会被改变。

接收选项参数说明如下：

* + - * 0表示接收超时后仍可继续调用RecvBiz来接收，
      * 1表示当接收超时后，把hSend相关数据删除

最后注意：外部指针所指向的解包器的内存由SDK内部管理，外部切勿释放！这个函数也是配合SendBiz，Create使用。

### 发送业务数据扩展接口（SendBizEx）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE SendBizEx(int iFunID, IF2Packer \*lpPacker,char\* svrName, int nAsy = 0, int iSystemNo = 0, int nCompressID = 1,int branchNo=0,LPREQ\_DATA lpRequest=NULL) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int iFunID | 功能号 | 输入 |
| IF2Packer \*lpPacker | 业务请求数据 | 输入 |
| char\* svrName | 指定中间件的节点 | 输入 |
| int nAsy = 0 | 是否异步发送 | 输入 |
| int iSystemNo = 0 | 系统号 | 输入 |
| int nCompressID = 1 | 是否压缩 | 输入 |
| int branchNo=0 | 营业部号 | 输入 |
| LPREQ\_DATA lpRequest=NULL | 其他相关的请求参数结构体指针 | 输入 |

**返回：**

返回发送句柄（正数），否则表示失败，通过调用GetErrorMsg可以获取详细错误信息。

**用法说明：**

当异步发送标志为0，就表示同步发送，同步发送的包，通过调用RecvBizEx来接收；当异步发送标志为1，就表示异步发送，异步发送的包，当收到应答包后，自动触发回调函数OnReceivedBizEx。这个函数需要配合CreateEx这个初始化函数一起使用。

参数结构LPREQ\_DATA定义如下所示：

typedef struct tagRequestData

{

int sequeceNo;//消息中心1.0的序号

int issueType;//消息中心1.0发布类型

void\* lpKeyInfo; //消息中心1.0关键字段，一般是Pack二进制形式

int keyInfoLen; //消息中心1.0关键字段长度

void\* lpFileHead;// 消息中心1.0附加数据

int fileHeadLen;// 消息中心1.0附加数据长度

int packetType;//包类型

Route\_Info routeInfo;//请求里面增加路由信息

int iSubSystemNo;//参数中增加子系统号传入

}REQ\_DATA, \*LPREQ\_DATA;

### 接收业务数据扩展接口（RecvBizEx）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE RecvBizEx(int hSend, void \*\*lppUnpackerOrStr, LPRET\_DATA\* lpRetData, unsigned uiTimeout = 1000, unsigned uiFlag = 0) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int hSend | 发送句柄，就是SendBiz返回值 | 输入 |
| void \*\*lppUnPackerOrStr | 返回的数据，根据函数返回值不一样，意义不一样 | 输出 |
| LPRET\_DATA\* lpRetData | 返回的其他数据 | 输出 |
| unsigned uiTimeout = 1000 | 接收数据的超时时间 | 输入 |
| unsigned uiFlag = 0 | 接收选项，接收超时之后是否删除对应发送句柄的数据。 | 输入 |

**返回：**

小于0表示RecvBizEx操作本身失败，通过调用GetErrorMsg可以获取详细错误信息。

**用法说明：**

下面针对lppUnPackerOrStr输出参数做详细说明：

* 如果返回值等于0，表示业务数据接收成功，并且业务操作成功，\*lppUnPackerOrStr指向一个解包器，此时应首先将该指针转换为IF2UnPacker \*。
* 如果返回值等于1，表示业务数据接收成功，但业务操作失败了，\*lppUnPackerOrStr指向一个解包器，此时应首先将该指针转换为IF2UnPacker \*。
* 如果返回值等于2，表示收到非业务错误信息，\*lppUnPackerOrStr指向一个可读的字符串错误信息。
* 如果返回值等于3，表示业务包解包失败。\*lppUnPackerOrStr原先所指向的内容不会被改变。

接收选项参数说明如下：

* + - * 0表示接收超时后仍可继续调用RecvBizEx来接收，
      * 1表示当接收超时后，把hSend相关数据删除

最后注意：

* 外部指针所指向的解包器的内存由SDK内部管理，外部切勿释放！
* 外部指针所指向的LPRET\_DATA的内存由SDK内部管理，外部切勿释放！
* 这个函数也是配合SendBizEx，CreateEx使用。

其他需要返回的字段的结构体定义如下：

typedef struct tagRespondData

{

int functionID;//功能号

int returnCode;//返回码

int errorNo;//错误号

char\* errorInfo;//错误信息

int issueType;//消息中心1.0发布类型

void\* lpKeyInfo; //消息中心1.0关键字段，一般是Pack二进制形式

int keyInfoLen; //消息中心1.0关键字段长度

Route\_Info sendInfo;//应答里面增加发送者信息

}RET\_DATA, \*LPRET\_DATA;

### 连接初始化的扩展接口（CreateEx）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE CreateEx(CCallbackInterface \*lpCallback) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CCallbackInterface \*lpCallback | 回调接口实现类指针 | 输入 |

**返回：**

返回0表示成功，否则表示失败，通过调用GetErrorMsg可以获取详细错误信息

**用法说明：**

如果应用不需要任何回调方法，则可向该方法传递NULL，而不必自定义回调类和对象。如果有回调实例，这个初始化的数据接收函数只会调用OnReceivedBizEx函数。

### 获取服务端上连接的IP和端口（GetRealAddress）

**函数原型：**

virtual const char\* FUNCTION\_CALL\_MODE GetRealAddress() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

获取ip地址和端口，格式：ip地址+端口

**用法说明：**

这个函数返回当前连接在服务端显示的IP地址和端口信息。

### 获取本地连接使用的IP和端口（GetSelfAddress）

**函数原型：**

virtual const char\* FUNCTION\_CALL\_MODE GetSelfAddress() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

获取ip地址和端口，格式：ip地址+端口

**用法说明：**

在多网卡的机器上，这个接口可以返回当前连接使用的IP地址和端口信息。

### 获取连接使用的MAC地址（GetSelfMac）

**函数原型：**

virtual const char\* FUNCTION\_CALL\_MODE GetSelfMac() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

MAC的地址字符串格式，类似“D067E5556D83”,中间没有分隔符

**用法说明：**

在多网卡的机器上，这个接口可以返回当前连接使用网卡的MAC地址。

### 创建订阅者（NewSubscriber）

**函数原型：**

virtual CSubscribeInterface\* FUNCTION\_CALL\_MODE NewSubscriber(CSubCallbackInterface \*lpCallback,char\* SubScribeName,int iTimeOut,

int iInitRecvQLen=INIT\_RECVQ\_LEN,int iStepRecvQLen=STEP\_RECVQ\_LEN) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CSubCallbackInterface \*lpCallback | 订阅者接收消息的回调类指针 | 输入 |
| char\* SubScribeName | 订阅者名字 | 输入 |
| int iTimeOut | 超时时间 | 输入 |
| int iInitRecvQLen=INIT\_RECVQ\_LEN | 初始化接收队列长度 | 输入 |
| int iStepRecvQLen=STEP\_RECVQ\_LEN | 接收队列扩大步长 | 输入 |

**返回：**

返回订阅接口实例，失败返回NULL，错误信息通过GetMCLastError获取。

**用法说明：**

要使用订阅发布的功能必须在连接里面增加相应的参数。对应的ini配置形式如下：

[mc]

;如果需要消息中心的订阅发布功能，这个配置项必须配置

client\_name=xuxp

连接属性里面没有这个配置，初始化就会失败。一个连接对应多个订阅，每个订阅者的都有自己的名字，名字不可以一样，是通过参数传进来的。

订阅端会有数据需要落地的，落地的目录可以配置，对应的ini配置形式如下：

[t2sdk]

logdir=D:\\mc

如果没有配置，就会在程序运行的当前目录下，建立一个目录，目录名字为mc开头，后面跟着client\_name配置的名字。

### 获取发布者（GetPublisher）

**函数原型：**

virtual CPublishInterface\* FUNCTION\_CALL\_MODE GetPublisher(int msgCount,int iTimeOut,bool bResetNo = false) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int msgCount | 本地缓存消息个数 | 输入 |
| int iTimeOut | 第一次初始化发布者超时时间 | 输入 |
| bool bResetNo = false | 是否重置序号 | 输入 |

**返回：**

返回发送接口实例，这是一个单例，如果已经new了，就直接返回对应的指针，失败返回NULL，错误信息通过GetMCLastError获取。

**用法说明：**

一个连接对应一个发布者，每个发布者的都有自己的名字，这个发布者的名字是在连接参数中获取的。对应的ini配置形式如下：

[mc]

;如果需要消息中心的订阅发布功能，这个配置项必须配置

client\_name=xuxp

连接属性里面没有这个配置，初始化就会失败。

本地缓存的消息个数是针对文件级，或者系统级的主题而言，这两种主题发布，会在本地落地，落地的目录可以配置，对应的ini配置形式如下：

[t2sdk]

logdir=D:\\mc

如果没有配置，就会在程序运行的当前目录下，建立一个目录，目录名字为mc开头，后面跟着client\_name配置的名字。

### 获取主题信息（GetTopic）

**函数原型：**

virtual IF2UnPacker\* FUNCTION\_CALL\_MODE GetTopic(bool byForce,int iTimeOut) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| bool byForce | 是否强制从服务端获取主题 | 输入 |
| int iTimeOut | 获取主题的超时时间 | 输入 |

**返回：**

返回消息中心的主题信息，用解包器的形式返回。

**用法说明：**

解包器外面需要调用release接口进行释放。解包器的返回字段有：

- TopicName

- TopicNo

- ReliableLevel

- IssuePriority

- MsgLifetime

- Stutas

- TickStrategy

- BusinessVerify

- Local

- FilterField1

- FilterField2

- FilterField3

- FilterField4

- FilterField5

- FilterField6

- FilterField7

- FilterField8

- FilterField9

- FilterField10

- FilterField11

- FilterField12

- FilterField13

- FilterField14

- FilterField15

- FilterField16

- SubscribeStr

- PublishStr

### 获取消息中心的最后错误（GetMCLastError）

**函数原型：**

virtual const char\* FUNCTION\_CALL\_MODE GetMCLastError() = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| 无 |  |  |

**返回：**

返回创建订阅者和获取发布者过程中的最后的错误信息

**用法说明：**

### 连接用IBizMessage接口初始化（Create2BizMsg）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE Create2BizMsg(CCallbackInterface \*lpCallback) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| CCallbackInterface \*lpCallback | 回调接口实现类指针 | 输入 |

**返回：**

返回0表示成功，否则表示失败，通过调用GetErrorMsg可以获取详细错误信息

**用法说明：**

如果应用不需要任何回调方法，则可向该方法传递NULL，而不必自定义回调类和对象。如果有回调实例，这个初始化的数据接收函数只会调用OnReceivedBizMsg函数。

### 连接发送IBizMessage业务消息（SendBizMsg）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE SendBizMsg(IBizMessage\* lpMsg,int nAsy = 0) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| IBizMessage\* lpMsg | 业务请求消息 | 输入 |
| int nAsy = 0 | 异步发送标志 | 输入 |

**返回：**

返回发送句柄（正数），否则表示失败，通过调用GetErrorMsg可以获取详细错误信息

**用法说明：**

当异步发送标志为0，就表示同步发送，同步发送的包，通过调用IBizMessage来接收；如果异步发送标志位1，就表示异步发包，当收到应答包后，自动触发回调函数OnReceivedBizMsg。这个函数需要配合Create2BizMsg这个初始化函数一起使用。

### 连接接收IBizMessage业务消息（RecvBizMsg）

**函数原型：**

virtual int FUNCTION\_CALL\_MODE RecvBizMsg(int hSend, IBizMessage\*\* lpMsg, unsigned uiTimeout = 1000, unsigned uiFlag = 0) = 0;

**输入参数：**

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数说明 | 用法 |
| int hSend | 发送句柄，就是SendBiz返回值 | 输入 |
| IBizMessage\*\* lpMsg | 返回的消息 | 输出 |
| unsigned uiTimeout = 1000 | 接收数据的超时时间 | 输入 |
| unsigned uiFlag = 0 | 接收选项，接收超时之后是否删除对应发送句柄的数据。 | 输入 |

**返回：**

返回0表示成功，否则表示失败，通过调用GetErrorMsg可以获取详细错误信息

**用法说明：**

接收选项参数说明如下：

* 0表示接收超时后仍可继续调用RecvBizMsg来接收，
* 1表示当接收超时后，把hSend相关数据删除

最后注意：外部指针所指向的消息指针的内存由SDK内部管理，外部切勿释放！这个函数也是配合SendBizMsg，Create2BizMsg使用。

# 注意事项

* 1. 相关限制
* 通过T2\_SDK的引出函数NewXXXX返回的对象，需要调用对象的Release方法释放，而不能直接用delete，因为t2sdk.dll（t2sdk.so）和调用程序可能是由不同的编译器、编译模式生成，delete可能会导致异常。为了适合Delphi等使用（Delphi对接口自动调用AddRef和Release方法），用C开发的代码，需要在NewXXXX之后调用一下AddRef以保证引用计数正确。
* 开发者在进行同步SendBiz，RecvBiz时，一般的调用流程为一次SendBiz后，马上RecvBiz。然而，如果开发者先进行N次SendBiz操作，再进行N次RecvBiz，此时，T2\_SDK可能会出现RecvBiz操作很慢，而实际上要接收的数据早已到达的情况。为了避免这种情况的发送，开发者可在配置文件中配置event\_count，使其数量等于N。即使event\_count配置的数量达不到N，通过扩大event\_count的数量也可以降低出现上述糟糕情况的概率。但请注意，event\_count的数量如果配置太大，将占用大量的系统资源。
* 使用T2\_SDK发送的包的最大长度为16M-1字节。
* Windows下，单进程使用T2\_SDK最多可以和服务端建立1024个连接，可通过多进程线性扩展。
* Linux下，单进程使用T2\_SDK最多可以和服务端建立（1024 减去 该进程打开的文件数）个连接，可通过多进程线性扩展。
* 通过GetVersionInfo函数获取的版本信息请用16进制方式查看，目前版本为0x01000002，表示1.0.0.2。
* 由于T2\_SDK考虑到多连接应用，所以SDK底层实现不支持断开重连。另外，设法在OnClose内部调用Connect来实现断开重连也是不可行的。那调用者该如何实现断开重连功能呢？一种可行的解决方案是在OnClose内部触发一个事件，而由wait该事件的其他线程去执行断开重连操作，具体请参考t2sdk\_demo目录下的“T2\_SDK异步发包、收包”例子。
* 在使用T2\_SDK时，对同一个连接，请勿多次Create。
* 用Delphi调用T2\_SDK时，务必要将IsMultiThread全局变量设置成True。
  1. 调用限制

1. OnClose回调中不能调用Connect、release、NewConnection
2. OnSent回调中不能调用Connect、Close、release、NewConnection
3. OnReceivedBiz回调中不能调用Connect、release、NewConnection
   1. 回调线程

t2sdk本身有两个线程：接收线程，发送线程，加上调用线程，总共3种线程

CCallbackInterface的回调函数，需要注意线程安全问题。

|  |  |
| --- | --- |
| 函数 | 线程 |
| OnConnect | 调用线程，发送线程 |
| OnSafeConnect | 调用线程，发送线程，接收线程 |
| OnRegister | 接收线程 |
| OnClose | 调用线程，发送线程，接收线程 |
| OnSent | 发送线程 |
| OnReceivedBiz | 接收线程 |

* 1. 编程建议
* 回调函数中，外部调用者切勿做大量耗时的操作，否则会引起数据发送/接收阻塞。
* 回调函数中，建议不要做类似同步的操作，否则可能会产生死锁。比如OnReceivedBiz回调中有段代码是m\_lstInfo.AddString(info)，其中m\_lstInfo是一个MFC文本编辑框，每次做一次AddString操作，线程会同步通知（SendMessageA）主线程刷新页面。注意，这是一个同步操作，假如当时主线程正在处理业务，而且下一步刚好是NewConnection之类的限制操作，那么就会产生死锁。

# 示例代码

本章主要是代码的示例，分成下面四个开发模式。展示代码不是很完整，完整代码请参考我们发布包的例子。

* 1. 同步发送接收

int main()

{

//通过T2SDK的引出函数，来获取一个新的CConfig对象

//此对象在创建连接对象时被传递，用于配置所创建的连接对象的各种属性（比如服务器IP地址、安全模式等）

//值得注意的是，在向配置对象设置配置信息时，配置信息既可以从ini文件中载入，

//也可以在程序代码中设定，或者是2者的混合，如果对同一个配置项设不同的值，则以最近一次设置为准

CConfigInterface \* lpConfig = NewConfig();

//通过T2SDK的引出函数NewXXXX返回的对象，需要调用对象的Release方法释放，而不能直接用delete

//因为t2sdk.dll和调用程序可能是由不同的编译器、编译模式生成，delete可能会导致异常

//为了适合Delphi等使用（Delphi对接口自动调用AddRef方法），用C/C++开发的代码，需要在NewXXXX之后调用一下AddRef

//以保证引用计数正确

lpConfig->AddRef();

//[t2sdk] servers指定需要连接的IP地址及端口，可配置多个，中间以“;”间隔

lpConfig->SetString("t2sdk", "servers", "127.0.0.1:9004");

//[t2sdk] license\_file指定许可证文件路径

lpConfig->SetString("t2sdk", "license\_file", "license.dat");

//[t2sdk] lang指定错误信息的语言号（缺省为简体中文2052），1033为英文

lpConfig->SetString("t2sdk", "lang", "1033");

//[t2sdk] send\_queue\_size指定T2\_SDK的发送队列大小

lpConfig->SetString("t2sdk", "send\_queue\_size", "100");

//[safe] safe\_level指定连接的安全模式，需要和T2通道的安全模式一致，否则连接失败

lpConfig->SetString("safe", "safe\_level", "none");

//通过T2SDK的引出函数，来获取一个新的CConnection对象指针

CConnectionInterface \*lpConnection = NewConnection(lpConfig);

lpConnection->AddRef();

int ret = 0;

//初始化连接对象，返回0表示初始化成功，注意此时并没开始连接服务器

if (0 == (ret = lpConnection->Create2BizMsg(NULL)))

{

//正式开始连接，参数1000为超时参数，单位毫秒

if (ret = lpConnection->Connect(1000))

{

puts(lpConnection->GetErrorMsg(ret));

}

else

{

//获取业务消息，结束后要释放此消息

IBizMessage\* lpBizMessage = NewBizMessage();

lpBizMessage->AddRef();

//功能号

lpBizMessage->SetFunction(76);

//请求类型

lpBizMessage->SetPacketType(REQUEST\_PACKET);

//设置子系统号

lpBizMessage->SetSubSystemNo(1);

//设置营业部号

lpBizMessage->SetBranchNo(100);

//获取打包器

IF2Packer\* pack = NewPacker(2);

pack->AddRef();

pack->BeginPack();

pack->AddField("plugin\_id");

pack->AddField("function\_id");

pack->AddStr("ospf");

pack->AddInt(3);

pack->EndPack();

//把业务包打入业务消息

lpBizMessage->SetContent(pack->GetPackBuf(),pack->GetPackLen());

//指定路由

//BIZROUTE\_INFO targInfo;

//strcpy(targInfo.ospfName,"ls\_mcup");

//strcpy(targInfo.pluginID,"mproxy");

//lpBizMessage->SetTargetInfo(targInfo);

//发送业务消息

if ((ret = lpConnection->SendBizMsg(lpBizMessage)) < 0)

{

puts(lpConnection->GetErrorMsg(ret));

}

else

{

//应答业务消息

IBizMessage\* lpBizMessageRecv = NULL;

//默认超时时间为1秒

ret = lpConnection->RecvBizMsg(ret,&lpBizMessageRecv);

if (ret == 0)

{

//成功,应用程序不能释放lpBizMessageRecv消息

if (lpBizMessageRecv->GetErrorNo() ==0)

{

//如果要把消息放到其他线程处理，必须自行拷贝，操作如下：

//int iMsgLen = 0;

//void \* lpMsgBuffer = lpBizMessageRecv->GetBuff(iMsgLen);

//将lpMsgBuffer拷贝走，然后在其他线程中恢复成消息可进行如下操作：

//lpBizMessageRecv->SetBuff(lpMsgBuffer,iMsgLen);

//没有错误信息

puts("业务操作成功");

int iLen = 0;

const void \* lpBuffer = lpBizMessageRecv->GetContent(iLen);

IF2UnPacker \* lpUnPacker = NewUnPacker((void \*)lpBuffer,iLen);

ShowPacket(lpUnPacker);

}

else

{

//有错误信息

puts(lpBizMessageRecv->GetErrorInfo());

}

}

else

{

//失败打印错误信息

puts(lpConnection->GetErrorMsg(ret));

}

}

//释放资源

pack->FreeMem(pack->GetPackBuf());

pack->Release();

lpBizMessage->Release();

}

}

else

{

puts(lpConnection->GetErrorMsg(ret));

}

//通过getchar阻塞线程，等待服务端应答包到达

getchar();

lpConnection->Release();

lpConfig->Release();

return 0;

}

* 1. 异步发送接收

void CCallback::OnReceivedBizMsg(CConnectionInterface \*lpConnection, int hSend, IBizMessage\* lpMsg)

{

if (lpMsg!=NULL)

{

//成功,应用程序不能释放lpBizMessageRecv消息

if (lpMsg->GetErrorNo() ==0)

{

//如果要把消息放到其他线程处理，必须自行拷贝，操作如下：

//int iMsgLen = 0;

//void \* lpMsgBuffer = lpBizMessageRecv->GetBuff(iMsgLen);

//将lpMsgBuffer拷贝走，然后在其他线程中恢复成消息可进行如下操作：

//lpBizMessageRecv->SetBuff(lpMsgBuffer,iMsgLen);

//没有错误信息

puts("业务操作成功");

int iLen = 0;

const void \* lpBuffer = lpMsg->GetContent(iLen);

IF2UnPacker \* lpUnPacker = NewUnPacker((void \*)lpBuffer,iLen);

ShowPacket(lpUnPacker);

}

else

{

//有错误信息

puts(lpMsg->GetErrorInfo());

}

}

}

int main()

{

// 通过T2SDK的引出函数，来获取一个新的CConfig对象

// 此对象在创建连接对象时被传递，用于配置所创建的连接对象的各种属性（比如服务器IP地址、安全模式）

CConfigInterface \* lpConfig = NewConfig();

// 通过T2SDK的引出函数NewXXXX返回的对象，需要调用对象的Release方法释放，而不能直接用delete

// 因为t2sdk.dll和调用程序可能是由不同的编译器、编译模式生成，delete可能会导致异常

// 为了适应Delphi等使用（Delphi对接口自动调用AddRef方法），用C/C++开发的代码，需要在NewXXXX之后调用一下AddRef

// 以保证引用计数正确

lpConfig->AddRef();

// [t2sdk] servers指定需要连接的IP地址及端口

lpConfig->SetString("t2sdk", "servers", "127.0.0.1:9004");

// [t2sdk] license\_file指定许可证文件

lpConfig->SetString("t2sdk", "license\_file", "license.dat");

// [t2sdk] send\_queue\_size指定T2\_SDK的发送队列大小

lpConfig->SetString("t2sdk", "send\_queue\_size", "100");

// 通过T2SDK的引出函数，来获取一个新的CConnection对象

g\_Connection = NewConnection(lpConfig);

g\_Connection->AddRef();

// 创建自定义类CCallback的对象（在初始化连接对象时需传递此对象，请看下面代码）

CCallback callback;

int ret = 0;

// 初始化连接对象，返回0表示初始化成功，注意此时并没开始连接服务器,这里必须用Create2BizMsg，否则回调不成功

if (0 == (ret = g\_Connection->Create2BizMsg(&callback)))

{

// 开启断开重连线程，由于在非断开情况下，该线程处于Wait状态，故对应用性能影响甚微

HANDLE thread = CreateThread(NULL, 0, AutoConnect, NULL, 0, NULL);

CloseHandle(thread);

// 正式开始连接注册，参数1000为超时参数，单位是ms

if (ret = g\_Connection->Connect(1000))

{

// 若连接/注册失败，打印失败原因

puts(g\_Connection->GetErrorMsg(ret));

}

else

{

//获取业务消息，结束后要释放此消息

IBizMessage\* lpBizMessage = NewBizMessage();

lpBizMessage->AddRef();

//功能号

lpBizMessage->SetFunction(76);

//请求类型

lpBizMessage->SetPacketType(REQUEST\_PACKET);

//设置子系统号

//lpBizMessage->SetSubSystemNo(1);

//设置营业部号

//lpBizMessage->SetBranchNo(100);

//获取打包器

IF2Packer\* lpPacker = NewPacker(2);

lpPacker->AddRef();

lpPacker->BeginPack();

lpPacker->AddField("plugin\_id");

lpPacker->AddField("function\_id");

lpPacker->AddStr("ospf");

lpPacker->AddInt(3);

lpPacker->EndPack();

//把业务包打入业务消息

lpBizMessage->SetContent(lpPacker->GetPackBuf(),lpPacker->GetPackLen());

//指定路由

BIZROUTE\_INFO targInfo;

strcpy(targInfo.ospfName,"ar");

strcpy(targInfo.nbrName,"xuxp");

////strcpy(targInfo.pluginID,"mproxy");

lpBizMessage->SetTargetInfo(targInfo);

// 异步模式收发包，第三个参数填1

g\_Connection->SendBizMsg(lpBizMessage, 1);

// 释放打包器，不释放会引起内存泄露，后果严重

lpPacker->FreeMem(lpPacker->GetPackBuf());

lpPacker->Release();

//释放消息

lpBizMessage->Release();

}

}

else

{

puts(g\_Connection->GetErrorMsg(ret));

}

// 通过getchar阻塞线程，等待服务端应答包到达

// 演示断开重连时，可在此时关闭服务器，然后再恢复

getchar();

// 释放资源

g\_Connection->Release();

lpConfig->Release();

return 0;

}

* 1. 订阅

这里订阅是针对服务端为消息中心2.0的代码。

void CSubCallback::OnReceived(CSubscribeInterface \*lpSub,int subscribeIndex, const void \*lpData, int nLength,

LPSUBSCRIBE\_RECVDATA lpRecvData)

{

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

PrintSub(subscribeIndex,lpRecvData);

IF2UnPacker\* lpUnPack = NewUnPacker((void\*)lpData,nLength);

if (lpUnPack)

{

lpUnPack->AddRef();

PrintUnPack(lpUnPack);

lpUnPack->Release();

}

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

void CSubCallback::OnRecvTickMsg(CSubscribeInterface \*lpSub,int subscribeIndex,const char\* TickMsgInfo)

{

}

int main()

{

//通过T2SDK的引出函数，来获取一个新的CConfig对象指针

//此对象在创建连接对象时被传递，用于配置所创建的连接对象的各种属性（比如服务器IP地址、安全模式等）

//值得注意的是，在向配置对象设置配置信息时，配置信息既可以从ini文件中载入，

//也可以在程序代码中设定，或者是2者的混合，如果对同一个配置项设不同的值，则以最近一次设置为准

CConfigInterface \* lpConfig = NewConfig();

lpConfig->AddRef();

lpConfig->Load("subscriber.ini");

//如果需要使用发布订阅功能，必须配置配置mc标签下面的client\_name项，配置文件里面有了，不需要下面这句代码添加

//lpConfig->SetString("mc","client\_name","xuxp");

//通过T2SDK的引出函数，来获取一个新的CConnection对象指针

g\_lpConnection = NewConnection(lpConfig);

g\_lpConnection->AddRef();

//创建自定义类CCallback的对象（在创建连接时需传递此对象，请看下面代码）

CCallback callback;

int ret = 0;

//初始化连接对象，返回0表示初始化成功，注意此时并没开始连接服务器

if (0 == (ret = g\_lpConnection->Create2BizMsg(&callback)))

{

//正式开始连接，参数1000为超时参数，单位是ms

if (ret = g\_lpConnection->Connect(1000))

{

puts(g\_lpConnection->GetErrorMsg(ret));

}

else

{

CSubCallback subscriberCallback;

char\* bizName = (char\*)lpConfig->GetString("subcribe","biz\_name","");

//如果需要使用发布订阅功能，必须配置配置mc标签下面的client\_name项

CSubscribeInterface\* lpSub = g\_lpConnection->NewSubscriber(&subscriberCallback,bizName,5000);

if (!lpSub)

{

printf("NewSubscribe Error: %s\n",g\_lpConnection->GetMCLastError());

return -1;

}

lpSub->AddRef();

//订阅参数获取

CSubscribeParamInterface\* lpSubscribeParam = NewSubscribeParam();

lpSubscribeParam->AddRef();

char\* topicName = (char\*)lpConfig->GetString("subcribe","topic\_name","");//主题名字

lpSubscribeParam->SetTopicName(topicName);

char\* isFromNow = (char\*)lpConfig->GetString("subcribe","is\_rebulid","");//是否补缺

if (strcmp(isFromNow,"true")==0)

{

lpSubscribeParam->SetFromNow(true);

}

else

{

lpSubscribeParam->SetFromNow(false);

}

char\* isReplace = (char\*)lpConfig->GetString("subcribe","is\_replace","");//是否覆盖

if (strcmp(isReplace,"true")==0)

{

lpSubscribeParam->SetReplace(true);

}

else

{

lpSubscribeParam->SetReplace(false);

}

char\* lpApp = "xuxinpeng";

lpSubscribeParam->SetAppData(lpApp,9);//添加附加数据

//添加过滤字段

int nCount = lpConfig->GetInt("subcribe","filter\_count",0);

for (int i=1;i<=nCount;i++)

{

char lName[128]={0};

sprintf(lName,"filter\_name%d",i);

char\* filterName = (char\*)lpConfig->GetString("subcribe",lName,"");

char lValue[128]={0};

sprintf(lValue,"filter\_value%d",i);

char\* filterValue = (char\*)lpConfig->GetString("subcribe",lValue,"");

lpSubscribeParam->SetFilter(filterName,filterValue);

}

//添加发送频率

lpSubscribeParam->SetSendInterval(lpConfig->GetInt("subcribe","send\_interval",0));

//添加返回字段

nCount = lpConfig->GetInt("subcribe","return\_count",0);

for (int k=1;k<=nCount;k++)

{

char lName[128]={0};

sprintf(lName,"return\_filed%d",k);

char\* filedName = (char\*)lpConfig->GetString("subcribe",lName,"");

lpSubscribeParam->SetReturnFiled(filedName);

}

int subscribeIndex = 0;

printf("开始订阅\n");

int iRet = lpSub->SubscribeTopic(lpSubscribeParam,5000,NULL);

if(iRet>0)

{

subscribeIndex = iRet;

printf("SubscribeTopic info:[%d] 成功\n",iRet);

g\_allSubscribeParam[subscribeIndex] = lpSubscribeParam;//保存到map中，用于以后的取消订阅

}

else

{

printf("SubscribeTopic info:[%d] %s\n",iRet,g\_lpConnection->GetErrorMsg(iRet));

return-1;

}

//打印已经订阅的主题信息

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

IF2Packer\* lpPack = NewPacker(2);

lpPack->AddRef();

lpSub->GetSubcribeTopic(lpPack);

if (lpPack)

{

PrintUnPack(lpPack->UnPack());

}

lpPack->FreeMem(lpPack->GetPackBuf());

lpPack->Release();

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("输入任意字符取消订阅\n");

getchar();

//添加过滤字段

iRet = lpSub->CancelSubscribeTopic(subscribeIndex);

printf("CancelSubscribeTopic:%d %s\n",iRet,g\_lpConnection->GetErrorMsg(iRet));

printf("输入任意字符退出\n");

getchar();

//释放订阅端

lpSub->Release();

printf("退出\n");

}

}

else

{

puts(g\_lpConnection->GetErrorMsg(ret));

}

//通过getchar阻塞线程，等待服务端应答包到达

getchar();

g\_lpConnection->Release();

lpConfig->Release();

return 0;

}

* 1. 发布

这里的发布代码也是针对服务器为消息中心2.0的代码。

int main()

{

//通过T2SDK的引出函数，来获取一个新的CConfig对象指针

//此对象在创建连接对象时被传递，用于配置所创建的连接对象的各种属性（比如服务器IP地址、安全模式等）

//值得注意的是，在向配置对象设置配置信息时，配置信息既可以从ini文件中载入，

//也可以在程序代码中设定，或者是2者的混合，如果对同一个配置项设不同的值，则以最近一次设置为准

CConfigInterface \* lpConfig = NewConfig();

lpConfig->AddRef();

lpConfig->Load("publisher.ini");

//如果需要使用发布订阅功能，必须配置配置mc标签下面的client\_name项，配置文件里面有了，不需要下面这句代码添加

//lpConfig->SetString("mc","client\_name","xuxp");

//通过T2SDK的引出函数，来获取一个新的CConnection对象指针

g\_lpConnection = NewConnection(lpConfig);

g\_lpConnection->AddRef();

//创建自定义类CCallback的对象（在创建连接时需传递此对象，请看下面代码）

CCallback callback;

int ret = 0;

//初始化连接对象，返回0表示初始化成功，注意此时并没开始连接服务器

if (0 == (ret = g\_lpConnection->Create2BizMsg(&callback)))

{

//正式开始连接，参数1000为超时参数，单位是ms

if (ret = g\_lpConnection->Connect(1000))

{

puts(g\_lpConnection->GetErrorMsg(ret));

}

else

{

char szFilterName[6][100]={0};

char szFilterValue[6][100]={0};

char\* topicName = (char\*)lpConfig->GetString("publish","topic\_name","");//获取发布的主题名

//获取过滤字段名字和值

int nCount = lpConfig->GetInt("publish","filter\_count",0);

for (int i=1;i<=nCount;i++)

{

char lName[128]={0};

sprintf(lName,"filter\_name%d",i);

char\* filterName = (char\*)lpConfig->GetString("publish",lName,"");

char lValue[128]={0};

sprintf(lValue,"filter\_value%d",i);

char\* filterValue = (char\*)lpConfig->GetString("publish",lValue,"");

strncpy(szFilterName[i-1],filterName,99);

strncpy(szFilterValue[i-1],filterValue,99);

}

printf("构造发布消息\n");

//如果需要使用发布订阅功能，必须配置配置mc标签下面的client\_name项

CPublishInterface\* lpPublish = g\_lpConnection->GetPublisher(200,5000);

if (!lpPublish)

{

printf("NewPublish Error: %s\n",g\_lpConnection->GetMCLastError());

return -1;

}

printf("开始发布\n");

//构造发布的业务包内容

IF2Packer\* lpOnePack = NewPacker(2);

lpOnePack->AddRef();

lpOnePack->BeginPack();

for (int j=0;j<nCount;j++)

{

lpOnePack->AddField(szFilterName[j]);

}

for (int k=0;k<nCount;k++)

{

lpOnePack->AddStr(szFilterValue[k]);

}

lpOnePack->EndPack();

IF2UnPacker\* lpUnPack = lpOnePack->UnPack();

while (1)

{

//业务包构造完毕

//调用业务的发送接口进行发布

int iRet = lpPublish->PubMsgByPacker(topicName,lpUnPack,5000,NULL/\*,true,&uRecordTime\*/);

//打印错误信息

printf("Next %d,%s,MsgID:%d\n",iRet,g\_lpConnection->GetErrorMsg(iRet),lpPublish->GetMsgNoByTopicName(topicName));

printf("是否退出，[y]退出，[其他字符]继续\n");

char ch=getchar();

if (ch=='y')

{

break;

}

}

lpOnePack->FreeMem(lpOnePack->GetPackBuf());

lpOnePack->Release();

//释放发布端

lpPublish->Release();

printf("退出\n");

}

}

else

{

puts(g\_lpConnection->GetErrorMsg(ret));

}

//通过getchar阻塞线程，等待服务端应答包到达

getchar();

g\_lpConnection->Release();

lpConfig->Release();

return 0;

}

# 附录

* 1. T2sdk错误码

-1套接字的值超过FD\_SETSIZE  
-2不存在指定编号的服务器端口  
-3包ID无效  
-4内存申请失败  
-5接收超时  
-6文件打开失败  
-7配置对象行数超过限制  
-8配置对象未指定服务器地址  
-9代理连接失败  
-10连接失败（非阻塞错误）  
-11同步连接超时或发生错误  
-12连接还未注册  
-13参数无效  
-14待发送数据长度异常  
-15发送队列满  
-16连接重连失败  
-17接收超时  
-18超出许可证的发送包数限制  
-19超出许可证的发送字节数限制  
-20注册超时  
-21SSL握手失败，可能客户端和服务端SSL版本不一致  
-22客户端被服务端校验的证书不存在，是否配置有误？  
-23该许可证允许接入客户端数已满  
-24不允许有名注册  
-25许可证不适用于T2  
-26许可证无效  
-27非注册包  
-28客户端通信密码错  
-29交换数据密码包中业务数据无效  
-30交换数据密码包中无业务数据  
-31非交换数据密码包，可能安全模式不匹配  
-32服务端在指定端口上没有进程在监听  
-33客户端被服务端校验的的证书密码错误（本地证书保护密码错），是否配置有误？  
-34客户端被服务端校验的证书过期，需要向服务端重新申请  
-35客户端校验服务端的证书不存在，证书名字是否配错？  
-36客户端校验服务端的证书密码错误  
-37服务端不支持SSL或者证书错误  
-38没有一个有效的证书（被服务端校验的证书）  
-39没有一个有效的根证书  
-40没有一个可用事件用来发送报文  
-41可用事件已经存在。