libCMS61850 二次开发手册

本文针对 libCMS61850 二次开发方法进行说明，帮助用户更好更快的上手，开发出自己的国产 61850。本库基于 C++11 进行开发，需要开发者掌握一定的 c++11 知识，面向对象，回调，智能指针等。

一、环境准备

本代码理论上可以基于任何平台编译，目前仅针对 linux 平台进行编译讲解。需要准备一个装有 ubuntu18.04 及以上（或同类 linux 相应版本）。

1、 安装编译工具链

若基于 x86 linux 运行，需安装 gcc/g++编译器。指令如下：

sudo apt install gcc

sudo apt install g++

若是交叉编译编译链，则需向开发板厂商索取交叉编译链，若是基于 aarch64，也可直接命令安装

sudo apt install gcc-aarch64-linux-gnu

sudo apt install g++-aarch64-linux-gnu

代码基于 cmake 管理编译，还需进行 cmake 的安装

sudo apt install cmake

确认 g++版本，需保证在 4.8.5 及以上，以保证 c++11 的全功能支持

```
pnc@pnc-virtual-machine:~/bc/code/OtherComponent/QtUi$ aarch64-linux-gnu-g++ -v
Using built-in specs.
COLLECT_GCC=aarch64-linux-gnu-g++
COLLECT_LTO_WRAPPER=/usr/lib/gcc-cross/aarch64-linux-gnu/7/lto-wrapper
Target: aarch64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 7.5.0-3ubuntu1~18.0
only --program-suffix=-7 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib -
-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable
disable-werror --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu
Thread model: posix
gcc version 7.5.0 (Ubuntu/Linaro 7.5.0-3ubuntu1~18.04)
```

2、 代码编译

将 libCMS61850 解压至任意目录，通过终端切换至解压目录，通过执行./build.sh 来进行编译，目前支持 x86 及 aarch64 平台，若需要其它平台编译，通过修改 CMakeLists 及 build.sh 脚本增加。以 aarch64 平台为例，执行./build.sh aarch64，动态库生成会在./out/aarch64/Debug 目录。也可生成 Release 库，可通过修改脚本实现

二、接口开发

2.1、代码框架

库代码采用组件化开发的思想，所以不用纠结为啥找不到 main 函数，对于每个组件库来说入口函数为组件相应的构造函数，然后为 init start 函数。以下介绍以下基本的组件运行流程

IUnknown： 组件基类，系统组件及业务组件最终继承的类。必须包含头文件 Component/IUnknown.h

SIMPLE_DEF_I： 宏定义，省去一些组件注册过程中重复代码的书写。第一个参数填写去掉 I 的命名，第二个参数是改组件的 clssid，唯一确定组件的名称。调用相应组件的实例时会用到。一般保持与第一个参数一致

固有函数：init，start，stop，destroy，分别表示组件的初始化，启动，停止及销毁。用户编写相应组件的时候，可以重写相应的函数。

对于 CMS61850 来说，组件接口文件为 ICMS61850.h，部分内容如下

```cpp
namespace cms {

class ICMS61850 : public base::IUnknown {
SIMPLE_DEF_I(CMS61850, "CMS51850")

/// 选择回调，返回值void，第一个参数节点名称，第二个为值。下同
using SelectFunc = TFunction<void, const std::string&, int>;
using OperFunc = TFunction<void, const std::string&, int>;
using SettingFunc = TFunction<void, const std::string&, int>;

public:
    virtual bool init() { return true; }
    virtual bool start() { return true; }
    virtual bool stop() { return true; }
    virtual bool destroy() { return true; }

public:
    /**
     * @brief 更新数据接口，考虑到用户只需要关注数据部分，内部的quality及time自动更新
     * @note 注意此接口与国标版61850接口型式一样，但具体参数有区别
     * @param domName 域名，iedName + ldName 如(KHPDFMONT)
     * @param varName 对象名 prefix + lnClass + lnInst + doName 如(airGGIO6.tmp)
     * @param attr 节点属性 DA 如(mag.f)
     * @param data 更新数据地址
     * @return true 更新成功
     * @return false 更新失败
     */
    virtual bool updateData(const std::string &domName, const std::string &varName, const std::string &attr, void *data){return true;}
    /**
```

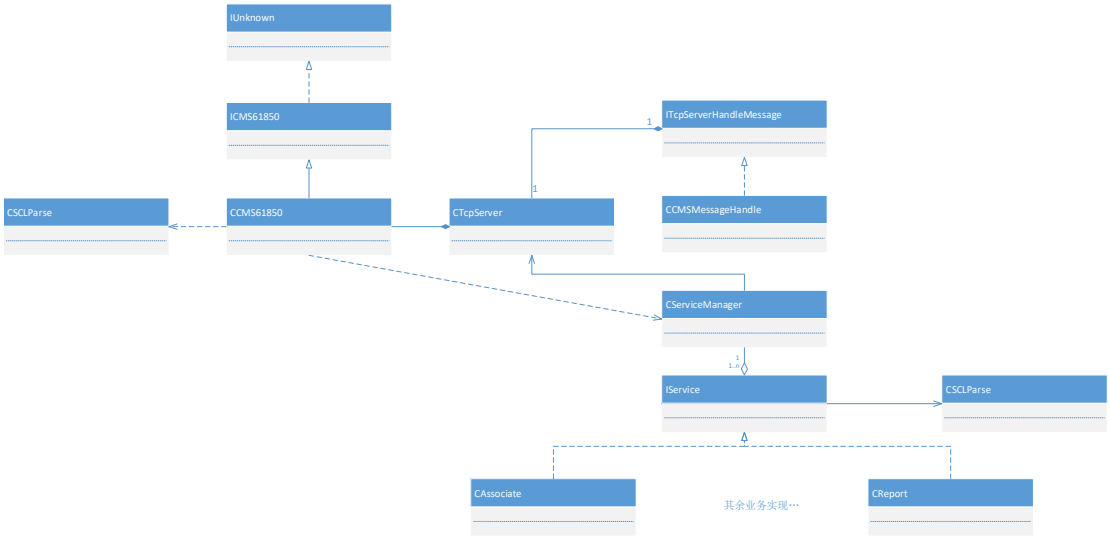
其集成了 base::IUnknown 基类组件。

组件实现文件分别为 CMS61850.h 及 CMS61850.cpp

```cpp
class CCMS61850 : public ICMS61850 {
SIMPLE_DEF_C(CMS61850, "CMS61850")

public:
    virtual bool init();
    virtual bool start();

public:
    virtual bool updateData(const std::string &domName, const std::string &varName, const std::string &attr, void *data);
```

程序运行时，首先执行 CCMS61850 的构造函数，然后会依次执行 init start 函数，在程序结束时会依次执行 stop destroy 函数，用户可根据自己需要进行相应接口继承重写。

核心业务部分类图如下所示：

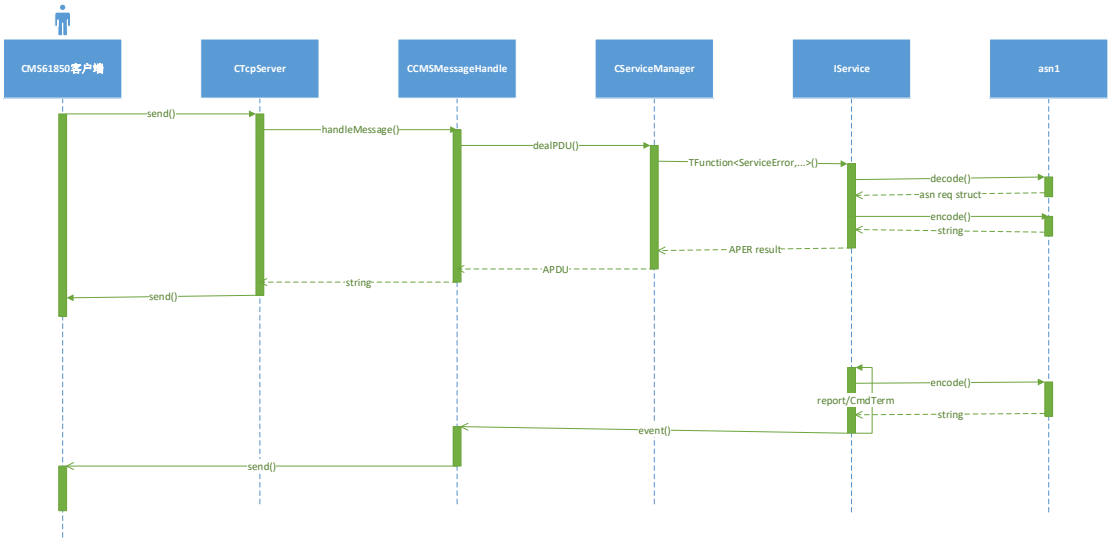


ITcpServerHandleMessage：tcp 消息处理基类
CCMSMessageHandle：具体的 CMS 消息处理类
CMS61850：组件的具体实现
CTcpServer：tcp 服务端，主要实现 tcp 的信息交互，其创建时需传入相应的 messageHandle 类。此处传入 CCMSMesageHandle。具体代码如下：

```
m_pHandleMsg.reset(new CCMSMessageHandle());
m_pTcpServer.reset(new base::CTcpServer("0.0.0.0", port, m_pHandleMsg));
```

CSCLParse：scl 文件的解析类
CServerManager：服务管理类，管理继承于 IService 的各项具体服务
IService：服务基类
CAssociate：具体的业务处理类，继承于 IService，比如此类主要实现协商相关的内容

主要部分交互时序图如下：



2.2、代码实现

此部分主要讲解一些重点代码，方便用户进一步理解代码架构及相应二次开发。

2.2.1 如何获取及使用组件

cms61850 使用到了框架种的一些组件功能（可闭源免费提供），若用户未购买框架且本身公司有相应框架的话，可以根据功能来进行相应替换。以下主要以使用到的两种组件进行解释。

```
    m_pConfig = base::CComponentManager::instance()->getComponent<base::IConfigManager>("ConfigManager");
    assert(m_pConfig);
```

此代码表示获取配置的相关组件，该组件主要完成了 json 配置的解析，读取，下发等功能。组件获取成功后，怎么使用呢？每个组件提供都是以接口文件的形式，可以很好的做到代码隔离，相应的接口，看其相应的接口文件。比如配置的接口文件为 IConfigManager.h

```
    SIMPLE_DEF_I(ConfigManager, "ConfigManager")
public:
    virtual bool init(){return true;}
    virtual bool start(){return true;}
    virtual bool stop(){return true;}
    virtual bool destroy(){return true;}

    /**
    *@brief 下发配置
    *将配置下发至配置中心管理
    *@param key 配置名
    *@param cfg 待配置
    *@return 下发是否成功
    */
    virtual bool setConfig(const std::string &key, const Json::Value &cfg) = 0;
```

其中的 setConfig 就可以直接进行使用，使用时传入相应参数就可以。

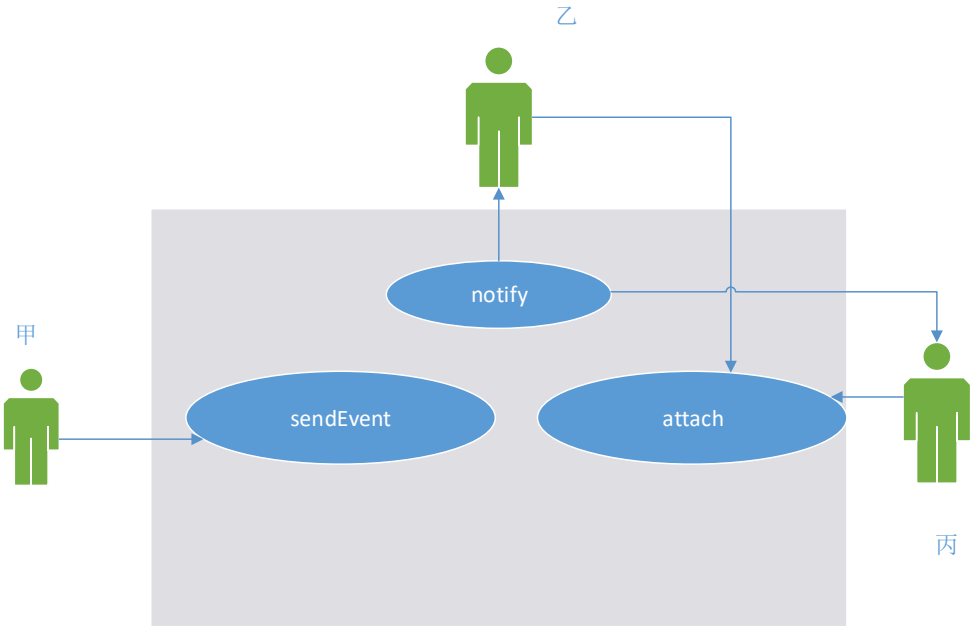
我们本身的 cms61850 也是一个组件，可同样的方法获取其组件指针，然后进行相应的接口调用，完成额外的业务。

auto *pCMS61850 = base::CComponentManager::instance()->getComponent<cms::ICMS61850>("CMS61850");

```
class ICMS61850 : public base::IUnknown {
SIMPLE_DEF_I(CMS61850, "CMS51850")

/// 选择回调，返回值void，第一个参数节点名称，第二个为值。下同
using SelectFunc = TFunction<void, const std::string&, int>;
using OperFunc = TFunction<void, const std::string&, int>;
using SettingFunc = TFunction<void, const std::string&, int>;

public:
    virtual bool init() { return true; }
    virtual bool start() { return true; }
    virtual bool stop() { return true; }
    virtual bool destroy() { return true; }

public:
    /**
    * @brief 更新数据接口，考虑到用户只需要关注数据部分，内部的quality及time自动更新
    * @note 注意此接口与国标版61850接口型式一样，但具体参数有区别
    * @param domName 域名，iedName + ldName 如(KHPDFMONT)
    * @param varName 对象名 prefix + lnClass + lnInst + doName 如(airGGIO6.tmp)
    * @param attr 节点属性 DA 如(mag.f)
    * @param data 更新数据地址
    * @return true 更新成功
    * @return false 更新失败
    */
    virtual bool updateData(const std::string &domName, const std::string &varName, const std::string &attr, void *data){return true;}
```

这样我们就可以在自己的业务中调用相应的接口，比如更新数据

第二个使用到的是事件组件 IEventManager，这个其实也是一个典型的订阅发布模式，交互图如下：



此模式可好的解耦相关代码，使用方法与前述一致，都是先获取组件指针，然后调用相应的收发事件接口即可

2.2.2 如何使用回调

代码中，为了更好的符合开闭原则，使用了很多回调函数，方便用户注册以实现自己特定的功能。回调只要使用了 TFunction 模板类，可以绑定任意型式的函数，与 c++11 的 std::function 相似，但使用起来更为方便。使用方法如下：

示例：

```
void test(int);
TFunction<void, int> tmp(&test);
class CTest{
public:
    void test(int);
};
CTest a;
TFunction<void, int> tmp1(&CTest::test, &a);
tmp(1);
tmp1(10)
```

模板实例化的第一个参数为函数返回类型，第二个为参数类型。

针对配置中心的回调函数类推，可以传入的函数型构为返回值 void,参数为 const Json::Value&的任意函数类型。

需注意的是当传入是成员函数时，TFunction 的第二个参数需要传入相应类的实例指针，且保持有效，否则，在调用时会出现死机问题。

当然每次显示实例化绑定一个函数书写起来会比较繁琐，可以直接包含 Function/Bind.h 利用 base::function 直接绑定即可，省去实例化的过程

```
m_configManager->attachChangedConfig("VideoServ", base::function(&CFFmpeg::attachRecConfig, this));
```

使用 TFunction 的特点，可以使任意函数类型的调用方式都如 C 函数调用方式一致。

TFunctionWrap 是 TFunction 的绑定参数版，也就是可以将函数的参数也都绑定进来，在调用时可以直接调用，不需要再传参。接上面的例子

TFunctionWrap<void, int> wrap(tmp1, 10);

wrap();

这个调用方式与 tmp1(10)效果一致。

优点是函数调用的方式得到进一步的统一，可以统一标识成 func();

同样可以调用 base::bind 的函数进行绑定，取代手动实例化。bind 除了可以返回具体的实例化类型，也可以返回 TFunction<void>的类型，我们这里称为闭包 Clouse

2.2.3 生成 APER 代码

APER 基于 asn1c 进行修改后封装，asn 语法不再详细，可另行百度学习，仅以使用来说。首先根据 CMS61850 的协议文档描述，进行语法定义的摘抄（本代码已全部生成相应代码，用户可自行测试），形成 asn 文件。以协商为例，协议中描述如下：

关联协商服务的语法定义如下：

```
AssociateNegotiate-RequestPDU ::= SEQUENCE {
    apduSize            [0] IMPLICIT INT16U,
    asduSize            [1] IMPLICIT INT32U,
    protocolVersion     [2] IMPLICIT INT32U
}

AssociateNegotiate-ResponsePDU ::= SEQUENCE {
    apduSize            [0] IMPLICIT INT16U,
    asduSize            [1] IMPLICIT INT32U,
    protocolVersion     [2] IMPLICIT INT32U,
    modelVersion        [3] IMPLICIT VisibleString
}
```

asn 文件如下：

```
AssociateNegotiate-RequestPDU ::= SEQUENCE {
apduSize [0] IMPLICIT INT16U,
asduSize [1] IMPLICIT INT32U,
protocolVersion [2] IMPLICIT INT32U
}
AssociateNegotiate-ResponsePDU ::= SEQUENCE {
apduSize [0] IMPLICIT INT16U,
asduSize [1] IMPLICIT INT32U,
protocolVersion [2] IMPLICIT INT32U,
modelVersion [3] IMPLICIT VisibleString
}
```

基本就只需要照抄，生成的文件如下：

```c
/*
 * Generated by asn1c-0.9.29 (http://lionet.info/asn1c)
 * From ASN.1 module "CMS61850Module"
 *  found in "./61850CMS.asn"
 *  `asn1c -D ./out -gen-APER -no-gen-BER -no-gen-OER -no-gen-XER -no-gen-JER -no-gen-example -fcompound-names`
 */

#ifndef _Associate_RequestPDU_H_
#define _Associate_RequestPDU_H_


#include <asn_application.h>

/* Including external dependencies */
#include "VisibleString129.h"
#include <OCTET_STRING.h>
#include "UtcTime.h"
#include <constr_SEQUENCE.h>

#ifdef __cplusplus
extern "C" {
#endif

/* Associate-RequestPDU */
typedef struct Associate_RequestPDU {
    VisibleString129_t  *serverAccessPointReference;    /* OPTIONAL */
    struct Associate_RequestPDU__authenticationParameter {
        OCTET_STRING_t   signatureCertificate;
        UtcTime_t    signedTime;
        OCTET_STRING_t   signedValue;

        /* Context for parsing across buffer boundaries */
        asn_struct_ctx_t _asn_ctx;
    } *authenticationParameter;

    /* Context for parsing across buffer boundaries */
    asn_struct_ctx_t _asn_ctx;
} Associate_RequestPDU_t;

/* Implementation */
extern asn_TYPE_descriptor_t asn_DEF_Associate_RequestPDU;

#ifdef __cplusplus
}
#endif

#endif  /* _Associate_RequestPDU_H_ */
#include <asn_internal.h>
```

```c
/*
 * Generated by asn1c-0.9.29 (http://lionet.info/asn1c)
 * From ASN.1 module "CMS61850Module"
 *  found in "./61850CMS.asn"
 *  `asn1c -D ./out -gen-APER -no-gen-BER -no-gen-OER -no-gen-XER -no-gen-JER -no-gen-example -fcompound-names`
 */

#ifndef _Associate_ResponsePDU_H_
#define _Associate_ResponsePDU_H_


#include <asn_application.h>

/* Including external dependencies */
#include <OCTET_STRING.h>
#include "ServiceError.h"
#include "UtcTime.h"
#include <constr_SEQUENCE.h>

#ifdef __cplusplus
extern "C" {
#endif

/* Associate-ResponsePDU */
typedef struct Associate_ResponsePDU {
    OCTET_STRING_t    associationId;
    ServiceError_t    serviceError;
    struct Associate_ResponsePDU__authenticationParameter {
        OCTET_STRING_t    signatureCertificate;
        UtcTime_t    signedTime;
        OCTET_STRING_t    signedValue;

        /* Context for parsing across buffer boundaries */
        asn_struct_ctx_t _asn_ctx;
    } *authenticationParameter;

    /* Context for parsing across buffer boundaries */
    asn_struct_ctx_t _asn_ctx;
} Associate_ResponsePDU_t;

/* Implementation */
extern asn_TYPE_descriptor_t asn_DEF_Associate_ResponsePDU;

#ifdef __cplusplus
}
#endif

#endif  /* _Associate_ResponsePDU_H_ */
#include <asn_internal.h>
```

实际使用时，只需要对这些结构体进行操作即可。

注意：在语法摘抄至 asn 文件时，生成时可能回报如下错误

FATAL: value tagged in IMPLICIT mode but must be EXPLICIT at line 658 in ./61850CMS.asn

将相应行的 IMPLICIT 改成 EXPLICIT 即可

2.2.4 业务代码

流程：

以协商代码为例：

编写 CAssociate 类（处理协商业务的），需继承 IService

```c
class CAssociate : public IService {
```

在 cpp 中定义相应的 static 变量

```c
static CAssociate s_ass;
```

此句的作用，是在程序启动之前，就会进行对象的构造。CAssociate 构造函数如下：

```
CAssociate::CAssociate()
    : m_stationId("linuxzq")
    , m_threadRes("AssociateManage")
{
    CServiceManager::instance()->attachService("Associate", this);
}
```

构造中将自己的指针注册进了 CServiceManager 中，其它的业务类都是同样的设计。
紧接着程序运行起来后，进入 CMS61850 组件的 init 函数

```
bool CCMS61850::init()
{
    setPrintLevel(getClisd(), DEBUG);
    m_pConfig = base::CComponentManager::instance()->getComponent<base::IConfigManag
    m_pConfig->getConfig("CMS61850", m_cfgValue);
    int connectNum = m_cfgValue["connectNum"].asInt();
    int port = m_cfgValue["port"].asInt();
    m_pHandleMsg.reset(new CCMSMessageHandle());
    m_pTcpServer.reset(new base::CTcpServer("0.0.0.0", port, m_pHandleMsg));
    m_pTcpServer->setConnectNum(connectNum);
    if (!CSCLParse::instance()->init())
    {
        return false;
    }
    if (!CServiceManager::instance()->init())
    {
        return false;
    }
```

调用到了 CServiceManager 的 init 函数

```
bool CServiceManager::init()
{
    for (const auto &iter : m_mapSrv)
    {
        if (!iter.second->init())
        {
            errorf("%s service init failed\n", iter.first.c_str());
            return false;
        }
    }
    base::IConfigManager *pConfig = base::CComponentManager::instance()->getComponent<base::IConfigManager>("ConfigManager");
    Json::Value cfgValue;
    pConfig->getConfig("CMS61850", cfgValue);
    m_errNum = cfgValue["errorNum"].asInt();
    m_apduSize = cfgValue["associate"]["apduSize"].asInt();
    m_asduSize = cfgValue["associate"]["asduSize"].asInt();
    return true;
}
```

这个函数将 m_mapSrv 遍历执行其相应的 init 函数，m_mapSrv 中存放的指针实际就是具体的业务指针，如前述的 CAssociate。

```
bool CAssociate::init()
{
    CServiceManager::instance()->attachFunc(1, "Associate", base::function(&CAssociate::associate, this));
    CServiceManager::instance()->attachFunc(2, "Abort", base::function(&CAssociate::abort, this));
    CServiceManager::instance()->attachFunc(3, "Release", base::function(&CAssociate::release, this));
    CServiceManager::instance()->attachFunc(153, "Test", base::function(&CAssociate::test, this));
    CServiceManager::instance()->attachFunc(154, "AssociateNegotiate", base::function(&CAssociate::associateNegotiate, this));
```

此函数中将具体的服务码与处理函数进行了绑定。这样可根据前面的时序图，当收到相应的服务码报文，则会进入相应的处理函数。比如服务码 2 终止的函数实现如下：

```
ServiceError CAssociate::abort(const std::string &funcName, const NetMessage &message, std::string &response)
{
    CSafeStruct<Abort_RequestPDU> reqPtr;
    if (!Decode(reqPtr, message.buf, message.len))
    {
        errorf("decode %s failed\n", funcName.c_str());
        return ServiceError_decode_error;
    }
    if ((const char *)reqPtr->associationId.buf != m_stationId)
    {
        errorf("abort station[%s] failed, not find\n", reqPtr->associationId.buf);
        return ServiceError_other;
    }
    infof("%s Abort, code[%ld]\n", m_stationId.c_str(), reqPtr->reason);
    auto *pEvent = base::CComponentManager::instance()->getComponent<base::IEventManager>("EventManager");
    Json::Value cfgValue;
    cfgValue["state"] = "disconnect";
    cfgValue["clientId"] = message.clientId;
    pEvent->sendEvent("cmsSocket", cfgValue);
    CServiceManager::instance()->closeClient(message.clientId);
    return ServiceError_no_error;
}
```

使用接口：

为使客户尽少的了解编解码相关的底层细节以及接口，对其进行了进一步的封装，形成自己的接口及类 CSafeStruct。

```
#define Decode(ptr, buf, len) decode(ptr.getTypePtr(), (void **)(ptr.getRef()), buf, len)
#define Encode(ptr, response) encode(ptr.getTypePtr(), ptr.get(), response)
```

Decode 及 Encode 接口分别是 APER 的编解码，Decode 第一个参数 ptr 为 CSafeStruct 的指针，buf 为待解码的内容，len 为长度。解码后的数据将填充到 ptr 内部的 asn 结构体，以协商请求为例，就是会将 buf 解码后，填充到 Associate_RequestPDU 结构体中
Encode 第一个参数为 CSafeStruct 的指针，第二个参数为解码后的结果。

CSafeStruct 是一个模板类，为了更好的管理资源，减少一些重复语句的编写。

```
template<typename T>
class CSafeStruct;

#define SafeDef(type) \
template<>\
class CSafeStruct<type> {\
using realType = DEFType(type);\
public:\
    explicit CSafeStruct() : m_bRelese(true) {m_ptr = CallocPtr(realType); m_stuPtr = StructPtr(type);}\
    CSafeStruct(realType *ptr) : m_bRelese(false) { m_ptr = ptr; m_stuPtr = StructPtr(type);}\
    ~CSafeStruct() {if (m_bRelese) FreeStruct(type, m_ptr);}\
public:\
    realType *get() {return m_ptr;}\
    realType **getRef(){return &m_ptr;}\
    realType *operator->() {return m_ptr;}\
    realType& operator*() {return *m_ptr;}\
    asn_TYPE_descriptor_t *getTypePtr() {return m_stuPtr;}\
private:\
    realType *m_ptr;\
    asn_TYPE_descriptor_t *m_stuPtr;\
    bool m_bRelese;\
};
```

使用方式，需先调用 SafeDef 定义，然后再使用相应模板

```
SafeDef(AssociateNegotiate_RequestPDU)
```

```
CSafeStruct<AssociateNegotiate_RequestPDU> reqPtr;
```

在使用时注意，若结构体下的元素需要申请内存，只管分配，不需要也不能进行手动释放。内存释放在 CSafeStruct 变量的声明周期结束后会自动释放。

在 Common.h 中，包含了对 OCT_STRING、BIT_STRING、内存、时间方面的处理，具体可见其实现。
在 SCLParse 中，包含了对 SCL 文件的解析，并且对各节点按照其类型进行了相应的内存分配，可方便读取修改相关值

```
struct DataInfo {
    struct LeafInfo {
        std::string name;
        std::string valueType;
        std::string fc;
        DataMem data;
        std::string initValue;
        std::string cdcType;
        /// 子节点
        std::vector<LeafInfo> vecInfo;
    };
    struct DomInfo {
        std::string domName;
        std::vector<LeafInfo> leafInfo;
    };
    /* key1 domName
       key2 lnName
    */
    // std::map<std::string, std::map<std::string, LeafInfo>> mapLeafInfo;
    std::vector<DomInfo> vecLeafName;
};

public:
    bool init();
    SclInfo &getSCLInfo() {return m_info;}
    DataInfo &getDataInfo() {return m_dInfo;}

public:
    /// 查找scl里的信息
    LdInfo *getLdInfo(const std::string &domName);
    LnInfo *getLnInfo(LdInfo *ldInfo, const std::string &lnName);
    LnInfo *getLnInfo(const std::string &ldName, const std::string &lnName);
    Report *getReportInfo(LnInfo *lnInfo, const std::string &rpName);
    DataSet *getDataSetInfo(LnInfo *lnInfo, const std::string &dsName);

public:
    /// 查找响应节点信息
    /// domName iedName + ldName
    /// leafName ln.do.name
    DataInfo::DomInfo *getDomInfo(const std::string &domName);
    DataInfo::LeafInfo *getLeafInfo(const std::string &domName, const std::string &leafName);
```

以上的接口和结构体为用户主要使用的，尤其 getLeafInfo 使用较多，该接口返回了具体节点下的信息，包括类型，约束，值，以及包含的子节点。可根据具体业务使用。

三、如何快速使用

libCMS61850 已基本覆盖了全 CMS 功能，大部分客户可能不需要对内部进行修改。更多需要快速投入实际生产使用。比如采集的某传感器的值，怎么通过 cms61850 转发出去。用户可参见 ICMS61850.h 接口文件，已经开放了更新数据，定值及遥控的接口。

```
public:
    /**
     * @brief 更新数据接口，考虑到用户只需要关注数据部分，内部的quality及time自动更新
     * @note 注意此接口与国标版61850接口型式一样，但具体参数有区别
     * @param domName 域名，iedName + ldName 如(KHPDFMONT)
     * @param varName 对象名 prefix + lnClass + lnInst + doName 如(airGGIO6.tmp)
     * @param attr 节点属性 DA 如(mag.f)
     * @param data 更新数据地址
     * @return true 更新成功
     * @return false 更新失败
     */
    virtual bool updateData(const std::string &domName, const std::string &varName, const std::string &attr, void *data){return true;}
    /**
     * @brief 更新数据接口，可更新任意数据，如quality time DC等。适合高度自由定制更新的客户
     * @note 此接口用户务必保证数据及长度正确，否则会使程序产生异常
     * @param domName 域名 iedName + ldName
     * @param leafName 叶节点名称 如(airGGIO6.tmp.mag.f)
     * @param data 节点数据地址
     * @param len 节点数据长度
     * @return true 更新成功
     * @return false 更新失败
     */
    virtual bool updateData(const std::string &domName, const std::string &leafName, void *data, int len){return true;}
    /**
     * @brief 注册选择回调函数，当选择动作（仅selectWithValue）发生，会触发
     * @param func 选择回调函数
     *
     */
    virtual bool attachSelectFunc(SelectFunc func){return true;}
    /**
     * @brief 注册操作回调函数，当操作动作发生，会触发
     * @param func 操作回调函数
     *
     */
    virtual bool attachOperFunc(SelectFunc func){return true;}
    /**
     * @brief 注册定值回调函数，当定值动作发生，会触发
     * @param func 选择回调函数
     *
     */
    virtual bool attachSettingFunc(SelectFunc func){return true;}
```

用户在获取 ICMS61850 的组件指针后，可直接使用相应接口，比如，我们的温度对应 icd 文件的 KHPDFMONT/airGGIO6.tmp 节点，需要更新值 20.5，示例代码如下：

```
auto *pCMS61850 = base::CComponentManager::instance()->getComponent<cms::ICMS61850>("CMS61850");
float tmp = 20.5;
pCMS61850->updata("KHPDFMONT", "airGGIO6.tmp", "mag.f", &tmp);
```

对于遥控遥调，只需要注册相应回调即可实现业务逻辑。

关于 cms61850 更多介绍，可进入以下主页学习

https://blog.csdn.net/z5201314100?spm=1019.2139.3001.5343