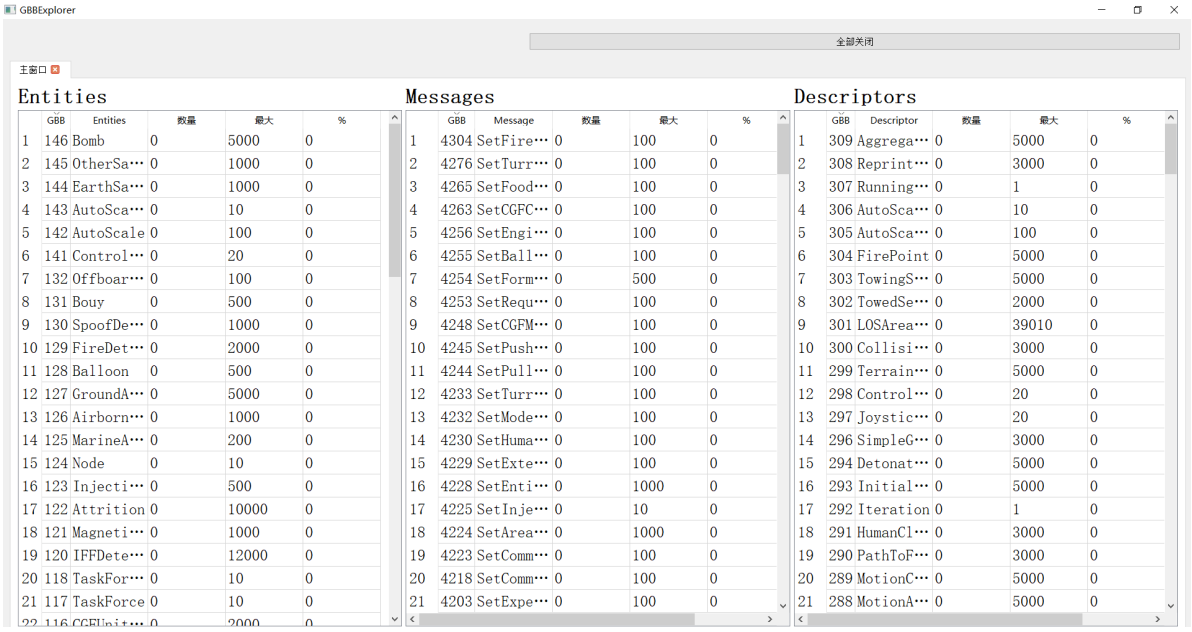
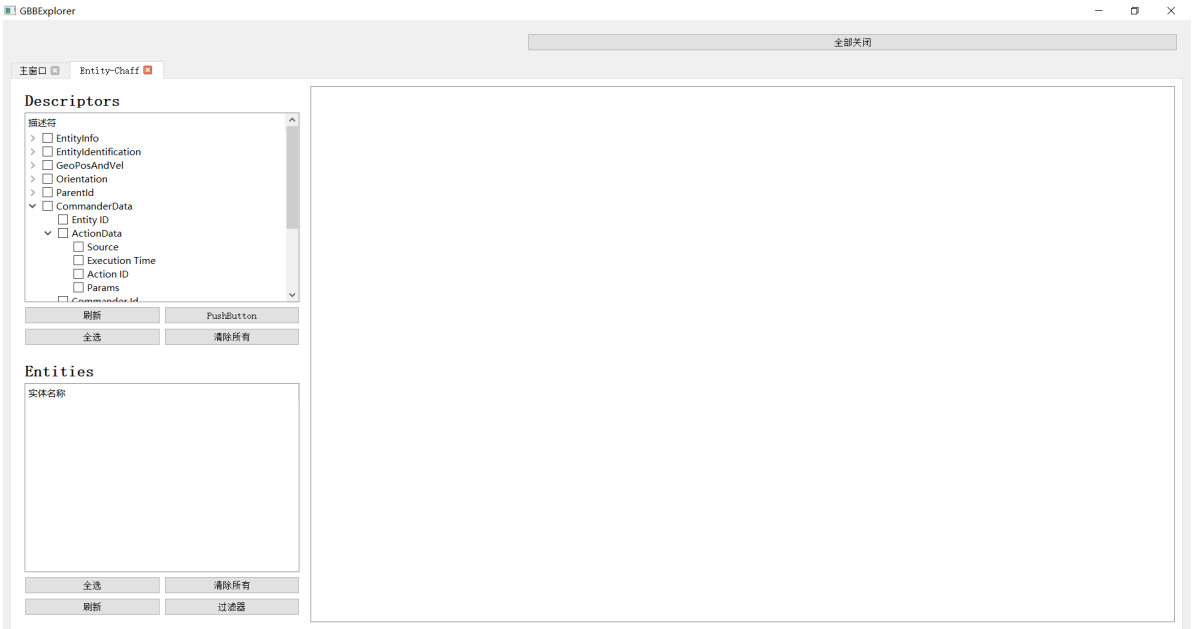
GBBExplorer记录阅读文档

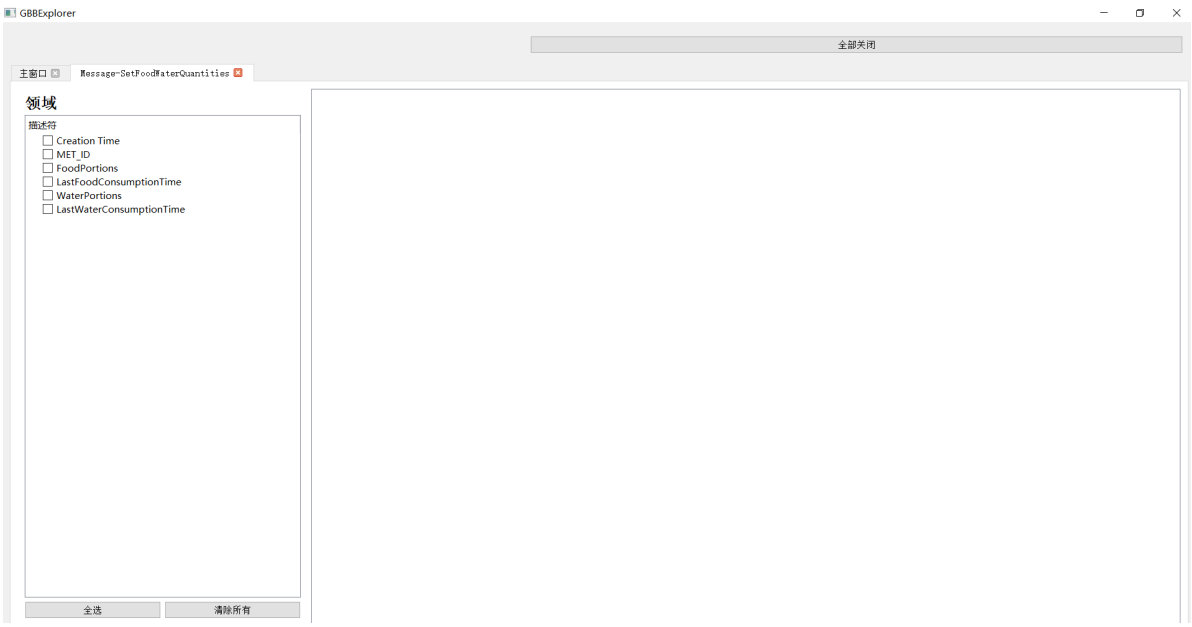
Widget类，主要是绘制界面的类，界面展示如下：



detail类，实体跳转界面的具体展示



detailmessage类，消息跳转界面展示



在Widget.h中定义的函数和槽函数如下，实现在Widget.cpp里

具体的一些声明和注释如下

```
private:
    void initForm(); //初始化主窗口


private slots:
    void on_tableViewdoubleClicked(const QModelIndex &index); //双击主页tableview上的名称跳转显示详情
    void on_removetabbtn(int index); //删除标签
    void on_pushButton_8_clicked(); //详情页全选子项目按钮的实现
    void on_pushButton_7_clicked();//清除按钮功能实现
    void on_treeWidget_2_clicked(QTreeWidgetItem *item);//treeWidget_2选中/不选中触发事件，模拟GBBexplorer中选择与取消
    void on_treeWidget_clicked(QTreeWidgetItem *item);//treewidget选中进行全部的行显示
```

StaticData类，主要是初始化静态数据

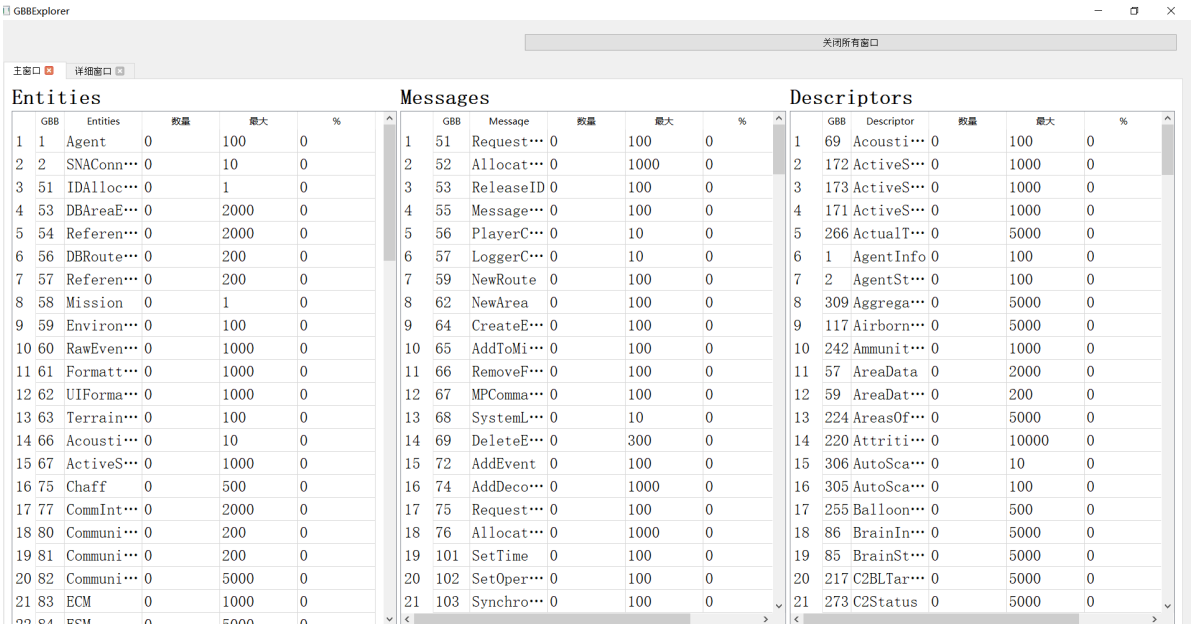
主要参照实现了CHSim-TKE_GBBExplorer\Infra\GBBExplorer\GBBExplorer文件夹中staticData.cs和entity.cs等定义的结构，从SerializedBuffer中获取静态数据

```
   void InitStructures();     // Create the Structures Static List
   void InitDescriptors();    // Create the Descriptors Static List
   void InitEntities();       // Create the Entities Static List
   void InitMessages();      // Create the Messages Static List
   int SetStringFromPtr(char* CurrentIntPtr, std::string &StringName);
```

main函数中

```
//用于日志记录和链接初始化GBB平台
LoggerUtil::Init();
    if (theConfigManager.Load("UI"))
    {
        theConfigManager.SetApplicationArgs(argc, argv);
        if (theProcessHelper->startNotificationEngine() == SUCCESS
            &&theProcessHelper->waitForBlackboardToStart() == SUCCESS
            &&theMonitorManager.Init())
```

已进行静态页面的展示



	GBB	Entities	数量	最大	%
1	1	Agent	0	100	0
2	2	SNAConn…	0	10	0
3	51	IDAlloc…	0	1	0
4	53	DBAreaE…	0	2000	0
5	54	Referen…	0	2000	0
6	56	DBRoute…	0	200	0
7	57	Referen…	0	200	0
8	58	Mission	0	1	0
9	59	Environ…	0	100	0
10	60	RawEven…	0	1000	0
11	61	Formatt…	0	1000	0
12	62	UIForma…	0	1000	0
13	63	Terrain…	0	100	0
14	66	Acousti…	0	10	0
15	67	ActiveS…	0	1000	0
16	75	Chaff	0	500	0
17	77	CommInt…	0	2000	0
18	80	Communi…	0	200	0
19	81	Communi…	0	200	0
20	82	Communi…	0	5000	0
21	83	ECM	0	1000	0

	GBB	Message	数量	最大	%
1	51	Request…	0	100	0
2	52	Allocat…	0	1000	0
3	53	ReleaseID	0	100	0
4	55	Message…	0	100	0
5	56	PlayerC…	0	10	0
6	57	LoggerC…	0	10	0
7	59	NewRoute	0	100	0
8	62	NewArea	0	100	0
9	64	CreateE…	0	100	0
10	65	AddToMi…	0	100	0
11	66	RemoveF…	0	100	0
12	67	MPComma…	0	100	0
13	68	SystemL…	0	10	0
14	69	DeleteE…	0	300	0
15	72	AddEvent	0	100	0
16	74	AddDeco…	0	1000	0
17	75	Request…	0	100	0
18	76	Allocat…	0	1000	0
19	101	SetTime	0	100	0
20	102	SetOper…	0	100	0
21	103	Synchro…	0	100	0

	GBB	Descriptor	数量	最大	%
1	69	Acousti…	0	100	0
2	172	ActiveS…	0	1000	0
3	173	ActiveS…	0	1000	0
4	171	ActiveS…	0	1000	0
5	266	ActualT…	0	5000	0
6	1	AgentInfo	0	100	0
7	2	AgentSt…	0	100	0
8	309	Aggrega…	0	5000	0
9	117	Airborn…	0	5000	0
10	242	Ammunit…	0	1000	0
11	57	AreaData	0	2000	0
12	59	AreaDat…	0	200	0
13	224	AreasOf…	0	5000	0
14	220	Attriti…	0	10000	0
15	306	AutoSca…	0	10	0
16	305	AutoSca…	0	100	0
17	255	Balloon…	0	500	0
18	86	BrainIn…	0	5000	0
19	85	BrainSt…	0	5000	0
20	217	C2BLTar…	0	5000	0
21	273	C2Status	0	5000	0

在widget.cpp里对获得的数组vecinfo进行迭代展示代码如下，以entity为例

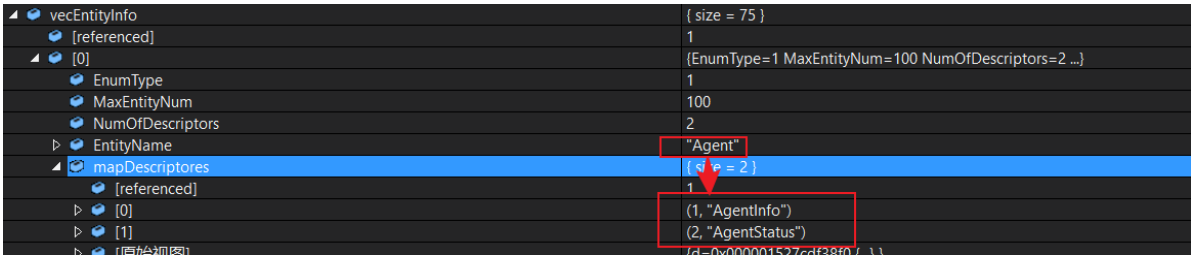
```
//初始化实例
        StaticData staticdata;

        staticdata.InitStructures();

        //读取静态实体数据显示
        staticdata.InitEntities();
        for (int i = 0; i < staticdata.vecEntityInfo.size(); i++)
        {
            QString EnumType =
QString::number(staticdata.vecEntityInfo[i].EnumType);
            QString EntityName =
QString::fromStdString(staticdata.vecEntityInfo[i].EntityName);
            QString MaxEntityNum =
QString::number(staticdata.vecEntityInfo[i].MaxEntityNum);
            model->setItem(i, 0, new QStandardItem(EnumType));
            model->setItem(i, 1, new QStandardItem(EntityName));
            model->setItem(i, 2, new QStandardItem("0"));
            model->setItem(i, 3, new QStandardItem(MaxEntityNum));
            //double rate = 0;
            model->setItem(i, 4, new QStandardItem("0"));
        }
```

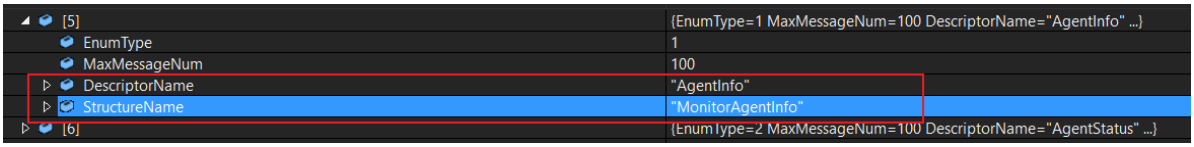
各个数据关联情况

- Entity点击详情，以agent为例：

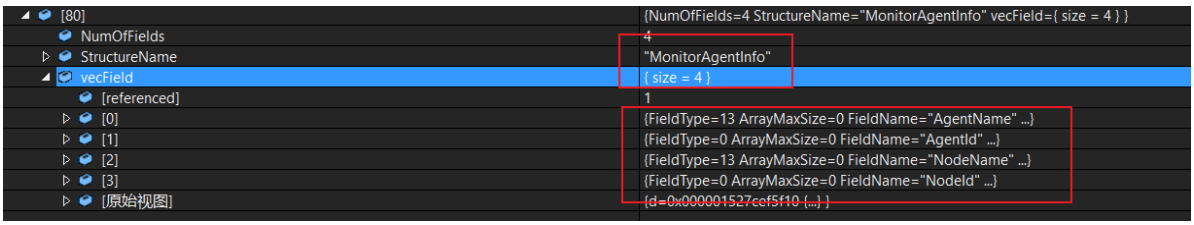


mapdescriptors是对应的2个根节点：

再去vecdescriptors中找到对应name的structure



根据`structname`在vecstruct中找子节点



- message详情，以requestnewid为例：

根据消息名称，在M_MessageInfo结构中找到descriptoname，再和上面一样去vecdescriptors中找到对应name的structure，做多重展示

注意，Creation time是手动添加的，逻辑在MessageView.cs中，MET_ID是根据m_bIsDescAsMessage添加的

```
1 个引用
private void InitStructFields()
{
    // Insert Creating TimeColumn
    m_lstAllFields.Add(new Field("Creation Time", "", FieldType.Time, m_cGeneralFunction.TimeDisplayState));
    m_ctlFieldsTree.Nodes.Add("Creation Time");

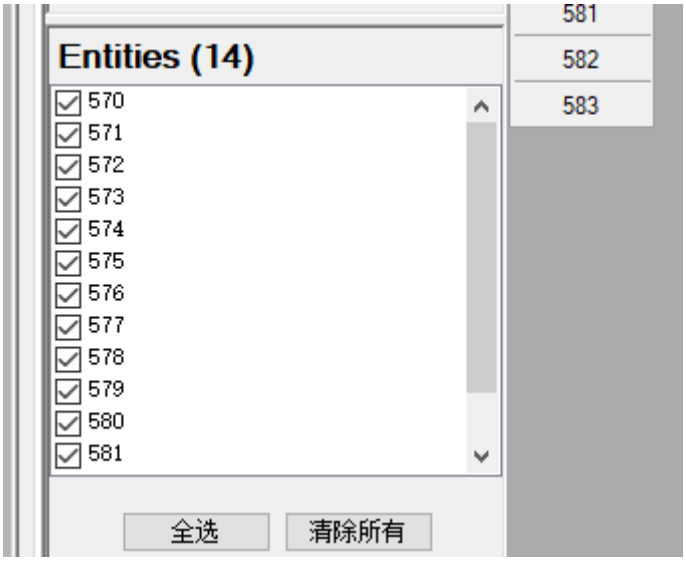
    // Insert "DescAsMessageColumn
    if (m_cCurrentMessage.IsDescAsMessage)
    {
        m_lstAllFields.Add(new Field("MET_ID", "", FieldType.MET_ID, 0));
        m_ctlFieldsTree.Nodes.Add("MET_ID");
    }
```

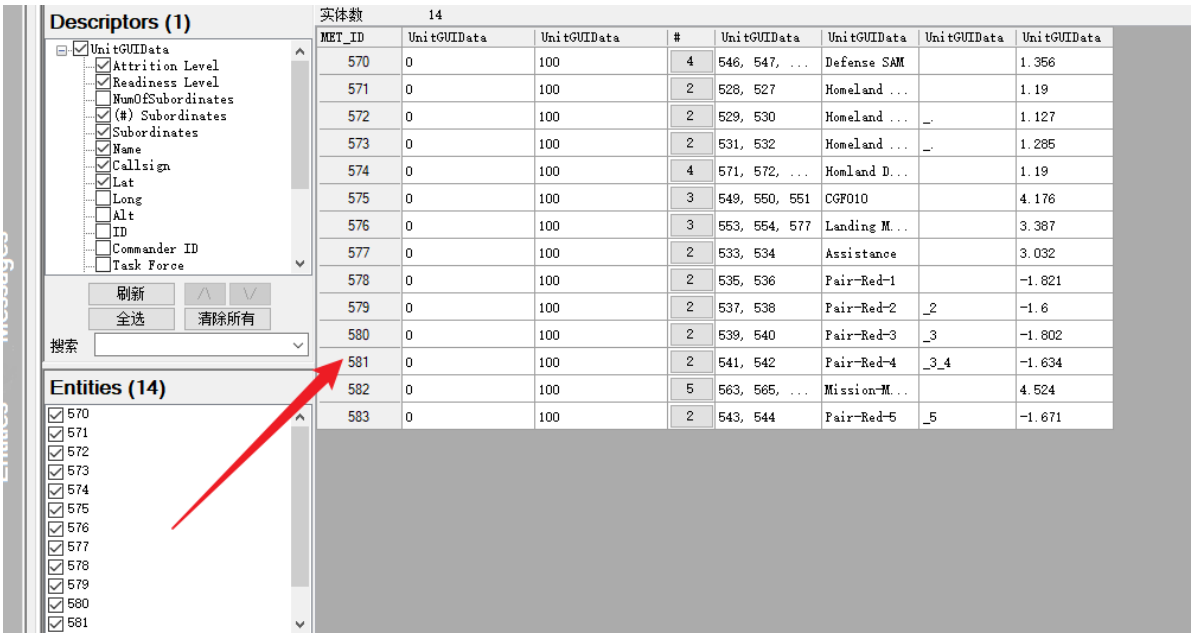
动态数据获取，在DynamicData类里

`bool GetEntitiesIDs(enum_t eEntityType);`每个周期，获取实体的**METID**

`bool GetEntityCount(enum_t eEntityType);`每个周期，动态获取实体数量

用于获取entity里的详情，如下图





`bool GetEntityDynamicData(id_t idEntity);`用于根据METID获取对应id的动态Entity数据

`bool GetEntityDynamicData(id_t idEntity, void* pDesNumbersVoid);`这个pDesnumber不清楚是做什么的，应该传什么参呢

调用情况：

dynamicdata.cs中：

```
216             if (CurrentDataTable.Rows[ElementIndex].Tag != null) // If this is ORANGE row (deleted entity)
217             {
218                 continue;
219             }
220             // Send to GBBMonitorManager.dll -
221             // 1.List of Entities need to read
222             // 2.List of descriptors number to read
223             // and get pointer from to the Data
224             try
225             {
226                 DataBufferPtr = ImportFunction.GetEntityData(EntityID[ElementIndex], m_pDescriptorsIntPtr);
227             }
228             catch
229             {
230                 // Write to error to log
231                 string ErrorString = "";
232                 for (int i = 0; i < Descriptores.Count; ++i)
233                 {
234                     ErrorString += Descriptores[i].m_sName + ",";
235                 }
```

补充源码里的cpp定义

```
42
43  extern "C" __declspec(dllexport) void* GetEntityData(int nEntityID, void* pDescritorsEnumType)
44  {
45      if (theMonitorManager.GetEntityDynamicData(nEntityID,pDescritorsEnumType))
46      {
47          return (void*)theMonitorManager.GetSerializedBuffer()->GetBuffer();
48      }
49
50      return NULL;
51  }
```

gbbmonitorwrapper.cpp里的定义

```
Widget.cpp | GBBMonitorWrapper.cpp | GBBMonitorManag...rnFunctions.cpp | Widget.h | StaticData.h | StaticData.cpp | main.cpp
杂项文件 | (全局范围) | GetEntityData(int nEntityID, void * pDescritorsEnumType)
191     return buffer;
192 }
193
194 extern "C" __declspec(dllexport) void* GetEntityData(int nEntityID,void* pDescritorsEnumType)
195 {
196     unsigned char* buffer = new unsigned char[200];
197     int* pEntity1 = (int*)buffer;
198     *pEntity1 = rand() % 6;
199
200     int* pEntity2 = (int*)(pEntity1+1);
201     *pEntity2= rand() % 6;
202
203     int* pEntity3 = (int*)(pEntity2+1);
204     *pEntity3= rand() % 6;
150 %
```

`bool GetEntityEnumType(long pEntityMet_ID);`获取enumtype

已经在DynamicData类里实现

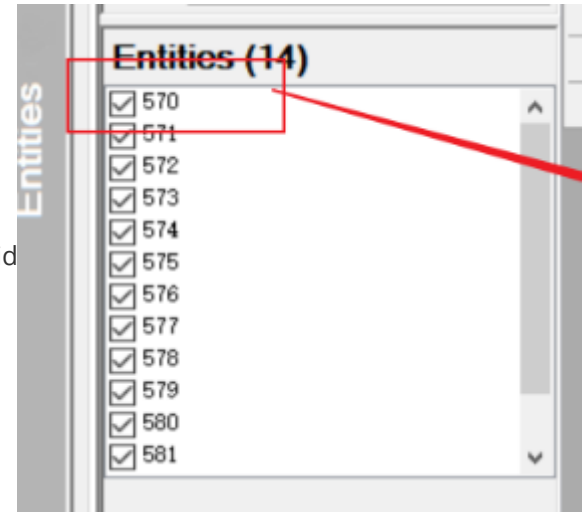
```
33
34      public:
35          DynamicData();
36          //DynamicData(QString configPath);
37          int GetEntityCount(int eEntityType);//每个周期，动态获取某个实体数量
38          void GetEntitiesIDs(int eEntityType);//每个周期，获取实体的所有metid
39          int GetEntityEnumType(long pEntityMet_ID);//每个周期，根据metid获取属于哪个实体enum
40          int GetMessageCount(int eMessageType); //每个周期获取某个消息的数量
41          int GetDescriptorCount(int eDescriptorType); //每个周期获取某个描述符的数量
42          //void GetEntityDynamicData(int nEntityID); //根据id获取动态数据
43          void GetEntityDynamicData(int eEntityType);//获取某个实体的全部动态数据
44          //bool ReadFieldFromPtr(char* allTheTablePtr, StaticData::M_FieldInfo currentField, )
45          ~DynamicData();
46      };
```

void GetEntitiesIDs(int eEntityType);//每个周期，获取实体的所有metid

比如：点击

▶	116	CGFUnitGUIData	14	2000

获取这些id

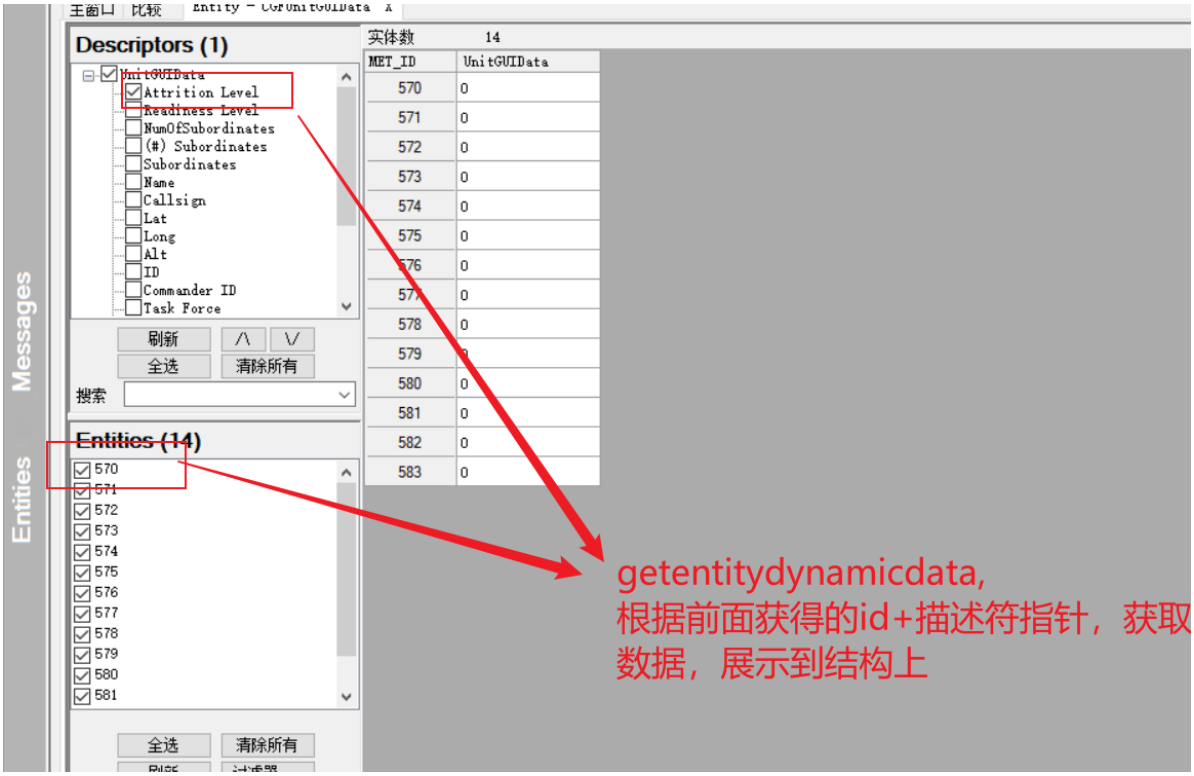


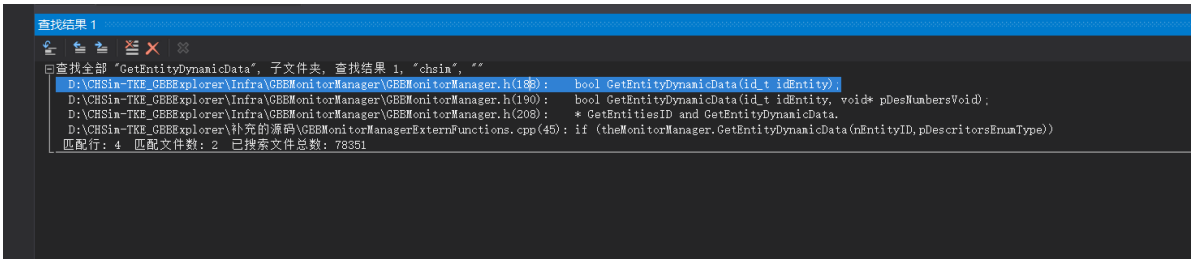
GetEntityDynamicData的结构

```
DynamicData                                    GetEntityDynamicData(int eEn
    int num = staticdata.vecDescriptorsInfo.size();
    int m = 0;
    //为每个描述符在分配的空间中占位，以enumtype
    for (int i = 0; i < num; ++i) {
        *(int*)(m_descriptorPtr + m) = staticdata.vecDescriptorsInfo[i].EnumType;
        m += sizeof(int);
    }
    *(int*)(m_descriptorPtr + m) = -1;

    GetEntitiesIDs(eEntityType);//每个周期，获取对应id实体的所有metid
    int num2 = EntitiesId.size();
    for (int i = 0; i < num2; i++) {
        if (theMonitorManager.GetEntityDynamicData(EntitiesId[i], m_descriptorPtr)) {
            GBBMonitor::SerializedBuffer* p = theMonitorManager.GetSerializedBuffer();
            char* ptr = (char*)p->GetBuffer();
```

可以理解成下面，其中描述符指针按enumtype进行占位（空间由xml中读取）





实现里的重要函数理解`ReadRowFromIntPtr`

```
        int NumOfBuffer = *(int*)(ptr1); ptr1 += sizeof(int);//buffer长度
        //TODO dynamicdata.cs中 GetEntityTableData等函数使用到的 ReadRowFromIntPtr函数功能
        char a[10000];
        for (int i = 0; i < 10000; i++) {
            a[i] = *ptr1;
            ptr1++;
        }
        qDebug() << "hello";
```

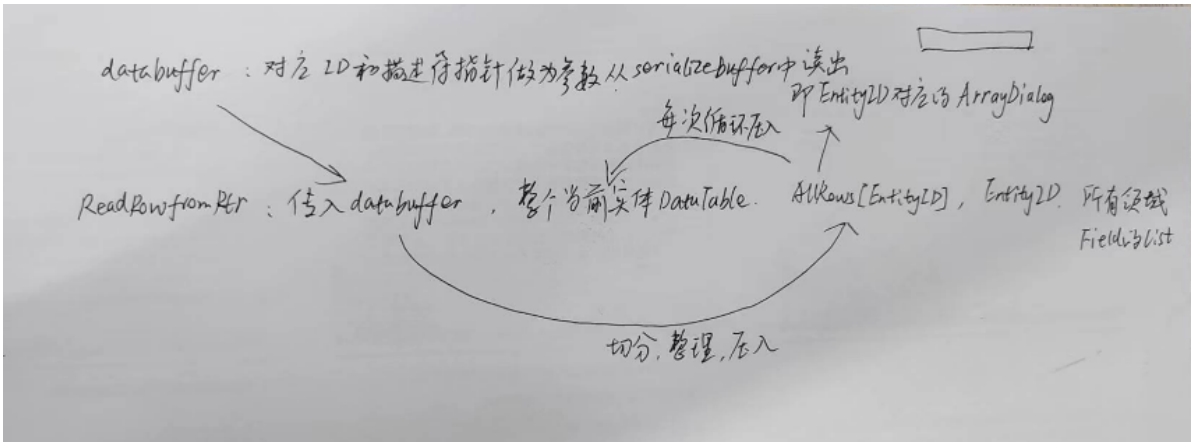
核心逻辑

```
    }

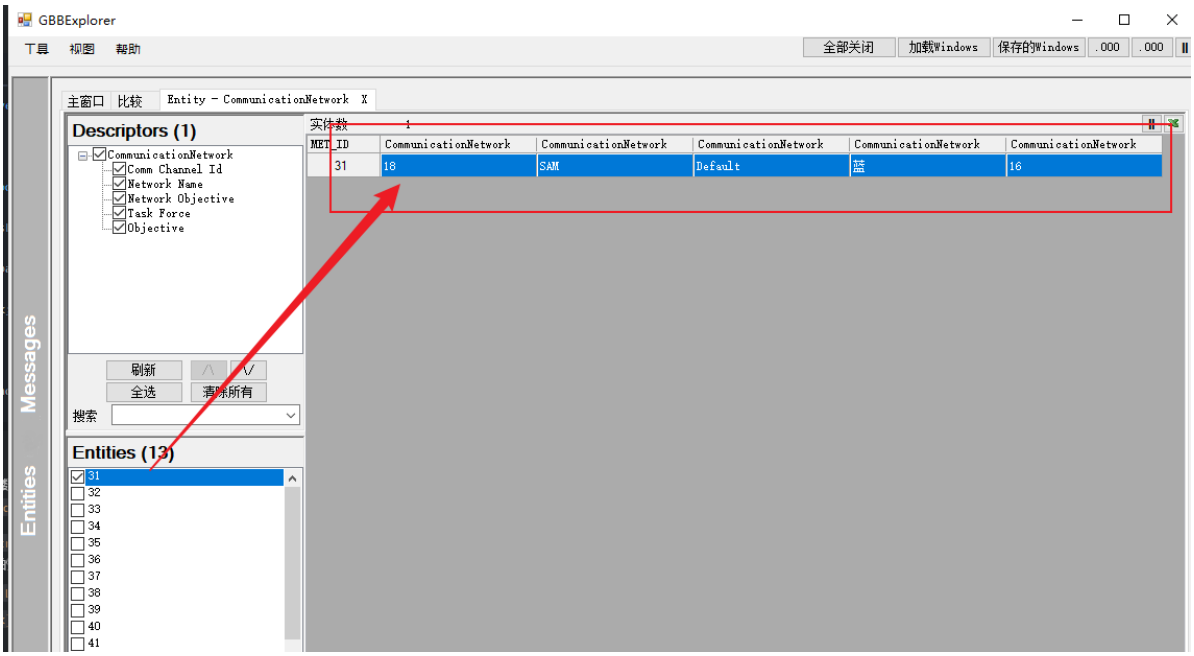
    m_nCurrentPos = 0;
    m_nBufferLength = Get32Bit(ref DataBufferPtr);

    ReadRowFromIntPtr(ref DataBufferPtr, ref CurrentDataTable, AllRowsArrayColumnIndex[ElementIndex], ElementIndex, F


    //Byte[] temp = new byte[10000];
    //int index = 0;
```

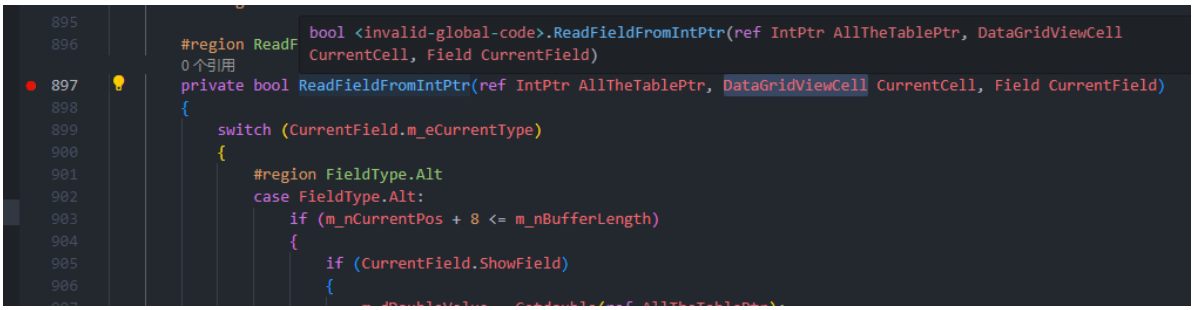


实际上是实现了切分获得下面这一行数据的功能

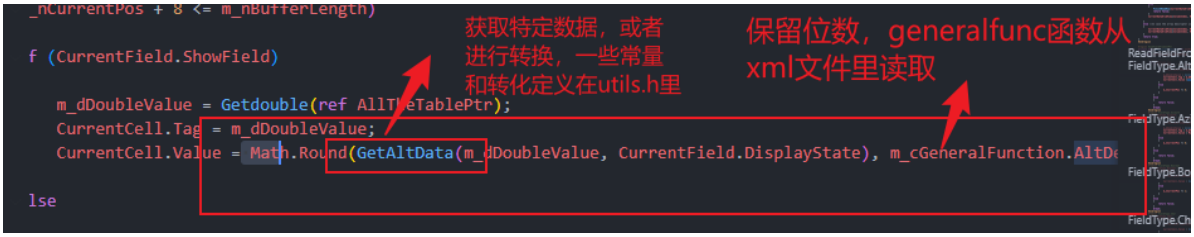


`ReadRowFromIntPtr`中

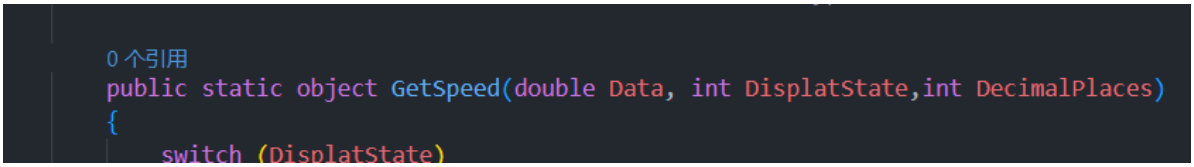
对于常规field，readfieldFromPtr函数根据fieldtype进行切割划分，得到值，以enum=81为例，选中id=31的实体，得到一个bufferlength长度的buffer（每一行都是这样），第一个字节是bool值，标志是否是开始描述符，接着按照不同类型切分



普通field的解析(switch case)



utils.h中用c++重写conversions.cs里面的所有转换，两者等同



```
0 个引用
public static object GetSpeed(double Data, int DisplatState,int DecimalPlaces)
{
    switch (DisplatState)
```

这里conversion.cs中c#的object类型，返回值可以有多种，查阅了一下资料，c++貌似没有什么好替代的，qt中倒是有类似的qvarient,正在试试看

```
class Conversions
{
public:
    static double convertFromMStoknot(double mps) { /*...*/ }
    static double convertFromMStoKph(double mps) { /*...*/ }
};

Q_DECLARE_METATYPE(double)

QVariant getSpeed(double data, int displayState, int decimalPlaces)
{
    switch (displayState)
    {
    case 1: // convert from m/s to kts
        return QVariant::fromValue(std::round(Conversions::convertFromM
    case 2: // convert from m/s to kph
        return QVariant::fromValue(std::round(Conversions::convertFromM
    default:
        return QVariant::fromValue(std::round(data * pow(10, decimalPla
    }
}
```

这里，我们使用 `QVariant::fromValue()` 将转换后的结果包装成 `QVariant` 对象。由于 `QVariant` 默认不支持 `double` 类型，我们需要使用 `Q_DECLARE_METATYPE(double)` 声明 `double` 类型以便 `QVariant` 能处理。同时，我们使用 `std::round` 进行四舍五入，而不是 `Math.Round`，因为这是C++的标准库函数。注意，C++中没有内置的舍入到指定小数位的方法，所以这里使用了乘以10的幂来实现四舍五入后再除以相同的幂。

重新生成

在Qt中，qvariant类是用于存储和传递各种不同数据类型的通用容器。qvariant可以存储多种基本数据类型,包括整型(int)、浮点型(double)、字符串(QString)等。

对于double类型,您可以使用以下方法进行操作:

1. 将double值赋给qvariant:

$$qvariantv = 3.14;$$

2. 从qvariant中获取double值:

$$doubled = v.toDouble();$$

3. 检查qvariant是否包含double类型:

$$if(v.type() == QVariant :: Double)//dosomething$$

转换回来

```
QVariant variantValue;

// 假设variantValue已经被赋值

// 转换为int
int intValue = variantValue.toInt();

// 转换为double
double doubleValue = variantValue.toDouble();

// 转换为QString
QString stringValue = variantValue.toString();

// 转换为bool
bool boolValue = variantValue.toBool();
```

精度decimalpalaces在xml中获取：

```
  <Data>
    <DescriptorBufferSize>10000000</DescriptorBufferSize>
  </Data>
  <Display>
    <WindowMode>0</WindowMode>
    <DescriptorsMode>0</DescriptorsMode>
    <MainWindow>
      <RowsColor Red="100" Orange="75" Yellow="50" />
    </MainWindow>
    <DecimalPlaces>
      <DecimalPlace Type="Double" NumberDecimalPlaces="3" />
      <DecimalPlace Type="Alt" NumberDecimalPlaces="2" />
      <DecimalPlace Type="Azimuth" NumberDecimalPlaces="5" />
      <DecimalPlace Type="LatLong" NumberDecimalPlaces="3" />
      <DecimalPlace Type="Speed" NumberDecimalPlaces="3" />
      <DecimalPlace Type="Range" NumberDecimalPlaces="3" />
    </DecimalPlaces>
    <FieldsStates>
      <Field Name="EnumToString" DefaultState="1" />
      <Field Name="Boolean" DefaultState="1" />
      <Field Name="Time" DefaultState="1" />
      <Field Name="Azimuth" DefaultState="1" />
      <Field Name="Altitude" DefaultState="1" />
      <Field Name="LatLong" DefaultState="1" />
      <Field Name="Range" DefaultState="1" />
    </FieldsStates>
  </Display>
  <OldWindows>
    <Windows />
  </OldWindows>
</GBBExplorer>
```

TODO：`showArrayField函数解析`（arrayfield情况）

界面展示

在qt中，用tablewidget取代c#里的DataGridViewCell,将单元格内数据设置其中。在 Qt 的 QTableWidget 中,使用 setData() 方法可以将自定义数据存储在单元格的 UserRole 中。这种方式可以帮助您在单元格中存储额外的信息,以便在需要时进行访问和操作。

以下是一个具体的例子:

```
1  // 创建 QTableWidget 并设置表头
2  QTableWidget* tableWidget = new QTableWidget(this);
3  tableWidget->setColumnCount(3);
4  tableWidget->setHorizontalHeaderLabels(QStringList() << "Name" << "Age" <<
   "ID");
5
6  // 添加数据行
7  QTableWidgetItem* nameItem = new QTableWidgetItem("John");
8  QTableWidgetItem* ageItem = new QTableWidgetItem("30");
9  QTableWidgetItem* idItem = new QTableWidgetItem();
10 idItem->setData(Qt::UserRole, 123); // 将 ID 存储在 UserRole 中
11
12 tableWidget->setItem(0, 0, nameItem);
13 tableWidget->setItem(0, 1, ageItem);
14 tableWidget->setItem(0, 2, idItem);
```

在这个例子中,我们创建了一个 QTableWidget,并添加了三列:Name、Age 和 ID。对于 ID 列,我们使用 setData() 方法将 ID 值存储在单元格的 UserRole 中。

通过这种方式,您可以在单元格中存储任何类型的自定义数据,而不仅仅是显示在单元格中的文本或图像。这样做的好处包括:

1. **数据分离**:将单元格显示的内容与实际的数据分离,可以更灵活地控制单元格的表现形式。
2. **数据访问**:在需要访问该数据时,可以通过 data() 方法获取存储在 UserRole 中的值。这在单元格事件处理中非常有用。
3. **扩展性**:您可以根据需要在单元格中存储更多的自定义数据,并在代码中进行相应的处理。

例如,在单元格单击事件中,您可以这样访问存储在 UserRole 中的 ID 值:

```
void MainWindow::on_tableWidget_cellClicked(int row, int column)
{
    QTableWidgetItem* item = tableWidget->item(row, column);
    if (item)
    {
        int id = item->data(Qt::UserRole).toInt();
        qDebug() << "Clicked cell ID:" << id;
    }
}
```

Qt 提供了一些预定义的 ItemDataRole,您可以根据需求选择合适的角色。例如,可以使用以下方式来存储和访问双精度浮点数:

```
const int DoubleValueRole = Qt::UserRole + 1; // 定义一个新的 ItemDataRole

// 设置双精度浮点数
double m_dDoubleValue = 3.14159;
item->setData(DoubleValueRole, QVariant::fromValue(m_dDoubleValue));

// 获取双精度浮点数
double doubleValue = item->data(DoubleValueRole).toDouble();
```

在这个例子中,我们定义了一个新的 ItemDataRole `DoubleValueRole`,并使用它来存储和访问双精度浮点数。您也可以为其他类型的数据定义自己的 ItemDataRole,如字符串、整数、对象等。

使用自定义 ItemDataRole 的好处包括:

1. **可扩展性**:您可以根据需求定义任意数量的自定义 ItemDataRole,以存储各种类型的数据。
2. **代码可读性**:使用自定义的 ItemDataRole 名称可以让代码更加易读和易于维护。
3. **避免命名冲突**:使用自定义的 ItemDataRole 可以避免与 Qt 预定义的角色发生名称冲突。

除了使用自定义的 ItemDataRole,您还可以考虑使用 QVariant 来存储复杂的数据结构,例如:

```
// 存储一个自定义对象
MyCustomObject obj;
item->setData(Qt::UserRole, QVariant::fromValue(obj));

// 获取自定义对象
MyCustomObject retrievedObj = item->data(Qt::UserRole).value<MyCustomObject>
();
```

总之,在 Qt 的 QTableWidget 中,您可以灵活地使用 ItemDataRole 来存储各种类型的自定义数据。

deshandler里定义了各个field的结构描述和添加

GBBexplorer的动态展示/刷新逻辑

代码在GBBform.cs中。

静态初始化页面：

```
480     #region InitMainTab
        1 个引用
481     private void InitMainTab()
482     {
483         // Insert entity Type column
484         List<Entity> AllEntities = m_cBL.GetAllStaticEntities();
485         EntitiesTable.RowCount = AllEntities.Count;
486         for (i = 0; i < AllEntities.Count; ++i)
487         {
488             OriginalEntityOrderMainTab.Add(AllEntities[i].m_sName, i);
489             EntitiesTable[0, i].Value = AllEntities[i].EnumType;
490             EntitiesTable[1, i].Value = AllEntities[i].m_sName;
491             EntitiesTable[3, i].Value = AllEntities[i].MaxEntityNum;
492         }
493         // Insert Message Type column
494         List<Message> AllMessages = m_cBL.GetAllStaticMessages();
495         MessagesTable.RowCount = AllMessages.Count;
496         for (i = 0; i < AllMessages.Count; ++i)
497         {
498             OriginalMessageOrderMainTab.Add(AllMessages[i].m_sName, i);
499             MessagesTable[0, i].Value = AllMessages[i].EnumType;
500             MessagesTable[1, i].Value = AllMessages[i].m_sName;
501             MessagesTable[3, i].Value = AllMessages[i].MaxMessageNum;
502
```

使用timer,隔一段时间更新entity视图和message视图

```
#region startTimer
1 个引用
private void startTimer()
{
    UpdateTimer = new System.Windows.Forms.Timer();
    UpdateTimer.Interval = UpdateRateMilliSconds;
    UpdateTimer.Tick += new EventHandler(UpdateTimer_Tick);
    UpdateTimer.Start();
}
#endregion
#region UpdateTimer_Tick
1 个引用
public void UpdateTimer_Tick(object sender, EventArgs e)
{

    UpdateAckMessges();

    foreach (EntityView CurrentEntityView in AllEntityViews)
    {
        CurrentEntityView.UpdateEntity(RemoveDeletedEntitiesAfterSeconds);
    }
    foreach (MessageView CurrentMessageView in AllMessageViews)
    {
        CurrentMessageView.UpdateMessages();
    }

    UpdateCompareData();

    UpdateMainTab();
    if (ExportEveryCycle) nowToolStripMenuItem_Click(null, null);
```

这其中使用了在BL.cs里封装了各种写好的类

updatemaintab就是更新主页面上数量（动态数据获取）

```
#endregion
#region UpdateMainTab
2 个引用
private void UpdateMainTab()
{
    #region Entities
    //  Update Entities
    for (i = 0; i < EntitiesTable.RowCount; ++i)
    {
        // EntitiesTable[2, i].Value = getEntityQuantity(EntitiesEnumDictionary[EntitiesTab
        EntitiesTable[2, i].Value = getEntityQuantity(EntitiesTable[1, i].Value.ToString())

        SetPercentAndRowColor(EntitiesTable, i);
    }
    m_cBL.Sorting(EntitiesTable);
    #endregion

    #region Messages
    //   Update Messages
    for (i = 0; i < MessagesTable.RowCount; ++i)
    {
        // MessagesTable[2, i].Value = getMessageQuantity(MessagesEnumDictionary[MessagesTal
        MessagesTable[2, i].Value = getMessageQuantity(MessagesTable[1, i].Value.ToString()
```

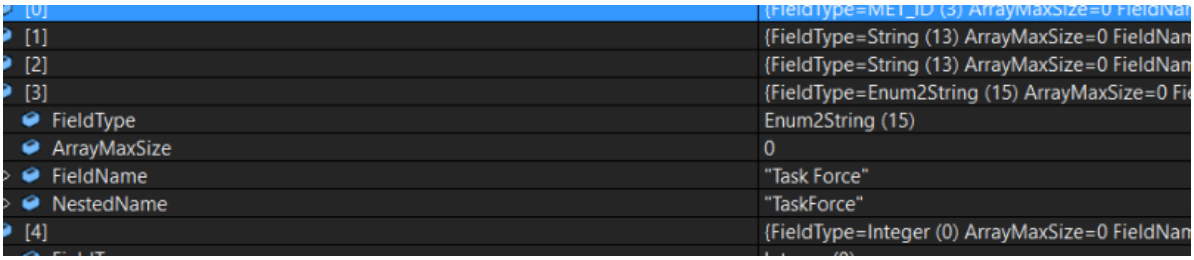
使用thread/timer多线程维护数据更新

- 主页面：各个实体/消息的数量的更新（qtimer）
- entity详情界面，暂时用thread进行展示（包括id，数量）
- 6.3：已完成大部分conversion.cs的函数转化，非array领域，正在编写。需要做的工作；array领域以及`showArrayField`函数对照源代码思路，部分还原
- 6.4新增用于处理多线程和定时器的类，并对需要进行的函数进行加锁，防止线程出错。
- 6.9新增对实体detail页面的描述符按字典序排序

如何将写的GetEntityDynamicData函数展示entity动态数据

- 发出：tableview
- 槽函数：GetEntityDynamicData
- 信号：doubleclick
- receiver：新建页面的tablewidget
- 传入一个detail类，通过cast获取到tablewidget对象，通过行列拿到item，再进行操作
- 刷新问题：使用Qtimer和Qthread进行操作

一些遇到的问题



在点击81号实体的时候，获取实体所有的field，与动态数据对应起来，其中task force的nestedname非空，但是其对应的taskforce找不到这样的结构体，且GBBExplorer中它也没有更多层。

```
for each(StaticData::M_FieldInfo fieldInfo in structInfo.vecField)
{
    //FieldsList.push_back(fieldInfo);
    if (fieldInfo.NestedName.empty())
    {
        FieldsList.push_back(fieldInfo);//这使得构建fieldlist的时候出错
    }
    else
    {
        for each(StaticData::M_StructuresInfo structInfo2 in staticdata.vecStructuresInfo)
        {
            if (structInfo2.StructureName == fieldInfo.NestedName) {
                for each(StaticData::M_FieldInfo fieldInfo2 in structInfo2.vecField)
                {
                    FieldsList.push_back(fieldInfo2);
                }
            }
        }
    }
}
```

Messge页面动态数据获取

`GetMessageDynamicData(enum_t eMessageType, HT::HT_TIME tSince);`

传入message的Enumtype和一个时间戳

- 6.18已完成动态获取Messge消息内容编写，这里涉及到array（待加入），跟原来c#的区别依然是传入一个detailmessage类替代原来的datagridview，用cast获取tablewidget控件进行操作，主要是单元格，一些区别已经写在注释中，比如qt中没有一次性获取某一行的datagridviewRow，这里使用用某行的第一列单元格替代，如有需要，可以此获得该行其他单元格

```
新增：
void GetMessageWithAckTableData(enum_t eMessageType, detailMessage *
MessageGridView, QVector<StaticData::M_FieldInfo> FieldsList, HT::HT_TIME
& requireTime);//获取有ack响应的message,比如时间等
void GetMessageDynamicData(enum_t eMessageType, detailMessage *
MessageGridView, QVector<StaticData::M_FieldInfo> FieldsList, HT::HT_TIME
& requireTime, QVector<CreationTime*> lst_LastCreationTime, int&
NextCreationTimeIndex, bool WithAck);//获取动态消息message
bool ReadAckRowFromIntPtr(char * ptr, QTableWidget*& tableWidget,
QVector<StaticData::M_FieldInfo> FieldsList, int nRowIndex, int
bufferLength);
```

- 在`ReadAckRowFromIntPtr`中，因为CreationTime（每个消息都有，buffer流中有但是不在读出的field里）和MET_ID（只有描述符消息有（m_bIsDescAsMessage==true）,buffer流中有但是不在读出的field里）在第一列和第二列，我进行了特判

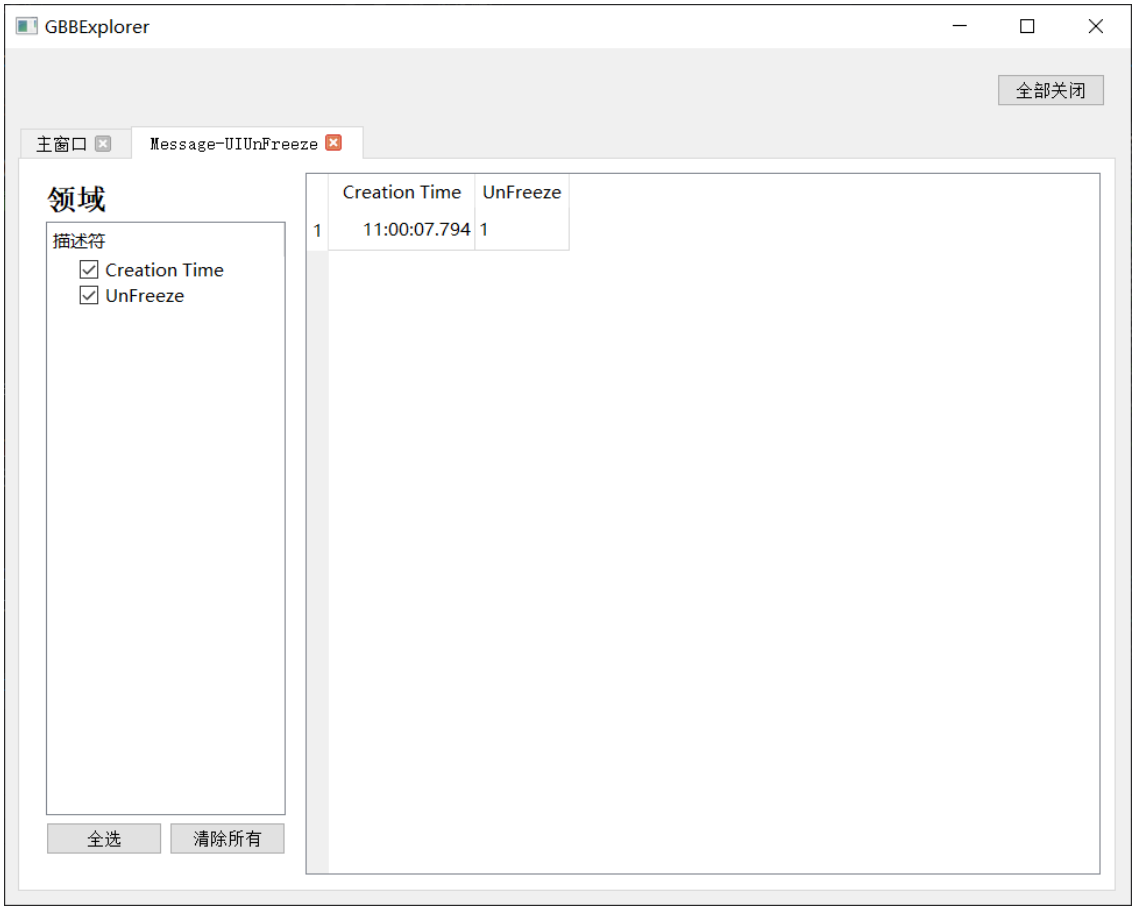
```
table->setRowCount(NumberOfNewMessage);
int columnCount = m_cCurrentMessage.m_bIsDescAsMessage ?
FieldsList.size() + 2 : FieldsList.size() + 1;//列数，如果是描述符消息，会多
2列MET_ID和creation time；如果不是，只增加一列creation time
    table->setColumnCount(columnCount);
    //特判,前两位
    int n = columnCount - FieldsList.size();
    if (n == 1)//不是消息描述符,只读Creation Time
    {
        StaticData::M_FieldInfo fieldInfo;
        fieldInfo.FieldType = StaticData::FieldType::Time;
        int n = tableWidget->rowCount();
        int m = tableWidget->columnCount();
        QTableWidgetItem* item = new QTableWidgetItem();
        tableWidget->setItem(nRowIndex, 0, item);
        bool flag = ReadFieldFromPtr(ptr, item, fieldInfo,
bufferLength);
    }
    else //是消息描述符,读Creation Time 和MET_ID
    {
        StaticData::M_FieldInfo fieldInfo1;
        fieldInfo1.FieldType = StaticData::FieldType::Time;
        QTableWidgetItem* item = new QTableWidgetItem();
        tableWidget->setItem(nRowIndex, 0, item);
        bool flag = ReadFieldFromPtr(ptr, item, fieldInfo1,
bufferLength);

        StaticData::M_FieldInfo fieldInfo2;
        fieldInfo1.FieldType = StaticData::FieldType::MET_ID;
        QTableWidgetItem* item1 = new QTableWidgetItem();
        tableWidget->setItem(nRowIndex, 1, item);
        bool flag2 = ReadFieldFromPtr(ptr, item1, fieldInfo2,
bufferLength);
    }
```

测试效果：

```
long long value = *(long long*)(fieldPtr);
//item->setData(Tag, value);
QVariant val = GetTime(value, 1);
//item->se                    val  19:04:35.697   e, val);
fieldPtr += 8;  已用时间 <= 1ms
m_nCurrentPos += 8;

e
```



- TODO:对array的解析
  - ```
    321                 //CreationTime timeRow(rowPosition);
    322                 lst_LastCreationTime.push_back(new CreationTime(rowPosition));
    323
    324                 // Create all the ArrayDialogs for the new row TODO
    325                 //CurrentMessageView.CreateNewArrayDialog();
    326
    ```
    ```