**版图基本框架描述**

[总体描述 2](#_Toc492244741)

[基础数据结构模块 3](#_Toc492244742)

[版图数据类结构描述图样 3](#_Toc492244743)

[版图数据结构简单描述 3](#_Toc492244744)

[版图文件读写模块 4](#_Toc492244745)

[文件读写模块类结构描述图样 4](#_Toc492244746)

[注意点描述 4](#_Toc492244747)

[功能CheckList 6](#_Toc492244748)

[功能描述 6](#_Toc492244749)

# 总体描述

架构示意图如下:



版图数据：

定义版图中的基本元素、层、版图场景的结构，一个版图就是一个版图场景对象。其他任何模块都是针对版图场景数据操作。

文件读写模块：

把各种格式的版图文件读入到通用的版图数据结构，或者把通用的版图数据结构写入到指定的格式文件。

运算模块：

根据需要把版图数据中的某个元素，或者某个层，甚至整个版图数据进行运算，诸如旋转平缩放等。

视图和UI交互：

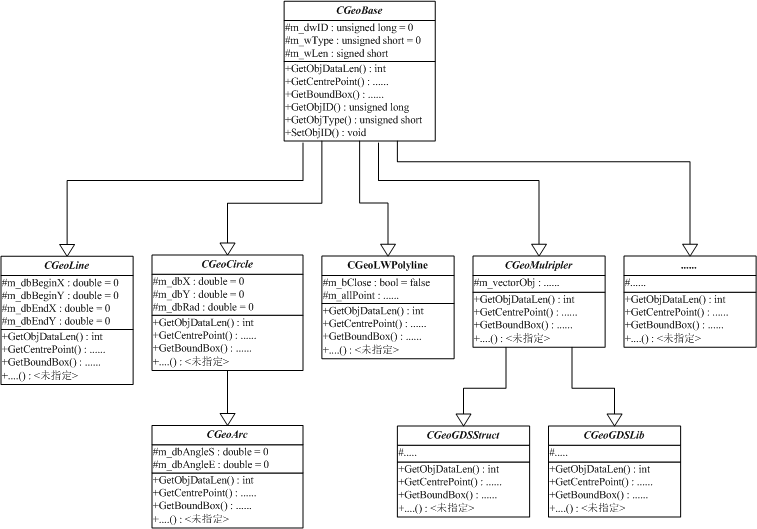
显示版图场景，绘制版图中的各种元素如直线，圆等。包括各种控制的操作。

目前存在的问题：

1. 并没有一个通用的版图数据结构，数据只是简单实用QGraphicsItem子对象来表示并保存，这样明显是没法实际操作的。以后的一些运算仿真就会比较麻烦甚至没法做。
2. 对于版图中各种元素绘制，也是直接采用QGraphicsItem子对象来表示图元。这种方式的优点节省了代码量，很多细节不需要关心。缺点是资源耗费增大，有些特殊应用可能比较难以实现，例如也许修改会比较麻烦。我还要评估一下这样的效率和资源耗费。以及坐标转换后是否导致精度损失。
3. 目前实现的操作好像都是针对单个图元，不能针对一组甚至全部数据。例如undo。
4. 源代码目录结构不要随意变更

# 基础数据结构模块

## 版图数据类结构描述图样



## 版图数据结构简单描述

1. 版图数据场景结构为CGeoScene，它包含两个列表，一个为节点的层列表，一个为Struct节点列表(也就是Cell列表)。对非GDS数据(如dxf数据)可能没有Struct节点，则Struct节点列表为空。通过层列表和Struct节点列表都能遍历所有的节点，因为它们最后都指向具体的数据节点。
2. 对于GDS数据结构，层列表中包含的是属于该层的Element数据列表。Struct列表中包含的Struct数据列表，Struct数据又包含属于该Cell的所有Element数据列表。
3. Element数据结构本身不包含具体的图元数据(如圆，多边形等图元)，具体的图元对象做为Element的子节点而存在（因为它也从复合节点CGeoMulripler派生）。例如多义线就是CGeoLWPolyline对象为它的子节点。
4. Element对象从CGeoTransform矩阵对象派生，所以对Element对象中具体的图元对象的旋转/移动/缩放，都只操作矩阵，不要去修改具体的图元坐标数据。注意：移动指移动图元，而不是移动整个版图便于查看。
5. 每个Element数据节点都有一个指向所属的Cell的父引用，通过该引用可以返回到对应的Struck节点。
6. Element数据不一定跟GDS文件里面的Element类型完全对应。例如GDS文件中的SREF和AREF可能只对应着一个类型，在读取数据时，转换参数可能整合到矩阵中。

# 版图文件读写模块

## 文件读写模块类结构描述图样

版图文件数据读写模块的代码逻辑结构



## 注意点描述

文件数据读取基类CGeoFileBase代码在GeoFile.h中，任何格式的文件实现代码必须派生与CGeoFileBase类，必须实现两个基本的虚函数ReadFile()和WriteFile()，前者是把文件数据解析成场景数据CGeoScene，后者把场景数据保存为指定格式的文件。除了两个基本的虚函数，以后可能还要添加其他的基础函数，例如GetErrorInf()，用于读取或者写入文件失败后，获取更详细的错误信息。

一个派生类至少要实现一种文件格式的读取，当然也可以同时实现几种格式的读取。对于任何格式文件的文件解析，必须实现读取函数ReadFile()用于读取指定格式的文件数据。但是根据实际情况是否特别需要，可以有选择的实现WriteFile ()函数。

任何派生类都必须有缺省构造函数，因为文件读写操作/遍历器根据不同格式生成对应派生类对象时调用的是缺省构造函数。

文件读写操作/遍历器子程序包括一系列的宏定义和两个读写函数，具体列举如下：

1. 全局函数ReadDataFromFile()和WriteDataToFile()
2. 宏定义DECLARE\_FILEFORMAT(class\_name)声明文件格式操作链接
3. BEGIN\_REGISTRATION(class\_name)和END\_REGISTRATION(class\_name) 宏定义用于实现文件格式操作链接
4. REGISTRATION\_EXT(class\_name,ext\_name)宏定义用于添加该类支持哪些格式解析。

有了这些宏定义后，以后扩展或者删除其他格式解析的类代码时，只需要针对具体的格式解析类做相关的操作，其他的代码都不需要修改。因为有了这些宏的定义，ReadDataFromFile()和WriteDataToFile()函数就跟自动遍历所支持的格式类并且生成相应的类对象然后操作这些对象进行读取或者写入操作。

如ReadDataFromFile("aaa.dxf")。则内部根类型名找到CGeoDXFFile类并生成CGeoDXFFile对象，然偶把文件名参数以及其他的必要参数传递进入后调用CGeoDXFFile::ReadFile()把文件数据读取到CGeoScene对象中。WriteFile()也类似。

这些宏定义是自动遍历的基础，如果没有这些宏定义，每次增加新的格式解析类或者删除已有的格式解析类， ReadDataFromFile()和WriteDataToFile()函数代码都要相应的修改。

宏定义的使用方法：

1. DECLARE\_FILEFORMAT(class\_name)宏定义添加在每个派生的具体格式文件解析类的定义末尾，其中class\_name为派生类的类名。如下：

class CGeoDXFFile : public CGeoFileBase

{

…… //其他代码

DECLARE\_FILEFORMAT(CGeoDXFFile)

};

1. 在派生类的实现文件文件中的开头添加剩下的三个宏定义，BEGIN\_REGISTRATION，REGISTRATION\_EXT，END\_REGISTRATION。示例如下：

include "GeoDxfFile.h"

… //其他包含文件以及其他数据

BEGIN\_REGISTRATION(CGeoDXFFile)

REGISTRATION\_EXT(CGeoDXFFile, TEXT("DXF"))

REGISTRATION\_EXT(CGeoDXFFile, TEXT("DWG"))

END\_REGISTRATION(CGeoDXFFile)

1. 支持一种格式，就使用REGISTRATION\_EXT(class\_name,ext\_name) 添加。如果支持多种格式，则每种格式都调用一次，如上的示例代码同时支持dxf和dwg格式。class\_name表示派生类名，ext\_name表示支持的格式名称。

**注意：REGISTRATION\_EXT必须在BEGIN\_REGISTRATION和END\_REGISTRATION宏定义的中间。**

# 功能CheckList

1、完善版图基础图元基础数据结构和文件数据结构定义

2、在Qt中实现版图基本图元自画。基本图元包括点，线，圆，圆弧，椭圆，多义线(区域)

3、实现基本图元的绘制数据从基础数据结构获取

4、在Qt中实现基本图元的选取，要注意多个图元有重叠的情况，这种情况要选取最上面的一个

5、实现对自画的基本图元移动缩放等处理

6、实现把GDS文件读入版图基础数据结构

7、实现版图基础数据结构数据写入到把GDS文件

8、实现dxf文件基本的数图元据读取

9、替换原先的Qt预定义图元和文档对象中的数据结构

10、命令行中画图调用的函数与主绘图中调用函数统一

# 功能描述

根据以上的checklis，把各个功能的开发以及一些功能实现的方案描述和工作安排

1. 完善版图基础图元数据结构和文件数据结构定义以及相关的数据读取功能，要求达到的最终目的如下：
2. 基础数据结构完全兼容GDS2文件格式数据、大部分兼容DXF文件格式。
3. 完善现有的GDS2文件数据读取，使之能够完整的解析出GDS2文件数据。
4. 把读取的GDS2文件的数据转化为场景数据结构
5. 实现把版图场景数据写入到GDS2文件中
6. 实现把基本DXF文件数据读取到版图场景数据中
7. 该功能估计工作量2~3周。
8. 实现基本图元的绘制数据从最新的版图场景数据结构中获取

该功能要求绘图的数据源必须从最新定义的版图场景数据结构里面获取

以画线为例：

CGeoPt ptStart,ptEnd;

CGeoLine\* temp = static\_cast<CGeoLine\*>(pObj);

ptStart = temp->GetFirstPt();

ptEnd = temp->GetSecondPt();

QgraphicsLineItem \*item = new QgraphicsLineItem (ptStart.dx, ptStart.dy, ptEnd.dx, ptEnd.dy);

item->setFlags(QGraphicsItem::ItemIsSelectable | QGraphicsItem::ItemIsMovable);

item->setPen(pen);

addItem(item);

item->setPos(0,0);

这个功能中的自定义在下一个功能描述中也会涉及到

这个功能由于这一步只是做一些数据源的替换实现。为了简化，可以把现有的绘制场景代码复制过来修改其中传入的参数，实现可以参考第四步。当然也可以与第四步的场景渲染结构合并一块进行，如果不合并，这个工作量比较少，估计2~3天。

1. 在Qt中实现版图基本图元自画。

这个功能要求把以前在Qt中通过QGraphicsItem类的子类对象画基本图元全部改成自画，例如上一步描述的就是通过QgraphicsLineItem实现画线，经过这一步后不再使用QGraphicsItem类的子类对象实现，也不需要通过addItem(item)之类的代码把该图元添加到场景中。可以减少实现数据渲染依赖具体UI对象。

还是以上面的画线为例，修改后画图的示例代码如下：

CGeoPt ptStart,ptEnd;

CGeoLine\* temp = static\_cast<CGeoLine\*>(pObj);

ptStart = temp->GetFirstPt();

ptEnd = temp->GetSecondPt();

QpointF tmpStart(ptStart.dx, ptStart.dy);

QpointF tmpEnd(ptEnd.dx, ptEnd.dy)

QPainter painter(this->viewport());

QPen pen(Qt::black);

painter.setPen(pen);

painter.setBrush(Qt::black);

painter.drawLine(tmpStart, tmpEnd);

注意：以上代码只是一个简单的演示代码。对于实际上的场景画图，画图时不要直接画到显存上，而是要先画到位图上，然后把位图复制给显存。这些绘制可以在paintEvent事件里面画图，也可以独立，根据实际情况来。

为了方便场景数据的绘制，定义了一个场景渲染接口类CSceneRender用于封装与UI绘制无关的绘制操作，在其子类里面实现与具体的UI接口关联绘制，接口定义在SceneRender.h文件中。该接口类定义了一些基础图元绘制函数接口，由子类实现绘制。

接口定义如下，以后也许会增加或者修改接口定义

virtual void DrawScene();

virtual void DrawPrimitive(CGeoLine\* pData)=0;

virtual void DrawPrimitive(CGeoCircle\* pData)=0;

virtual void DrawPrimitive(CGeoArc\* pData)=0;

virtual void DrawPrimitive(CGeoLWPolyline\* pData)=0;

virtual void DrawPrimitive(CGeoEllipse\* pData)=0;

virtual void DrawPrimitive(CGeoText\* pData)=0;

virtual void DrawPrimitive(CGeoMulripler\* pData)=0;

在上面的定义中，DrawScene()函数接口用于绘制整个场景，它的实现是遍历整个场景所有图层中的所有图元（如果某个图元有子图元，则迭代遍历，直达基本图元的叶节点），然后分别画出基本图元。

DrawPrimitive()函数分别对应着不同的基本图元实现不同的绘制方法。例如DrawPrimitive(CGeoLine\* pData)函数的实现绘制直线，可能是如下的情况：

DrawPrimitive(CGeoLine\* pData)

{

CGeoPt ptStart,ptEnd;

ptStart = pData ->GetFirstPt();

ptEnd = pData ->GetSecondPt();

如果没有实现自定义绘制，则应该是如下的代码

QgraphicsLineItem \*item = new QgraphicsLineItem (ptStart.dx, ptStart.dy, ptEnd.dx, ptEnd.dy);

item->setFlags(QGraphicsItem::ItemIsSelectable | QGraphicsItem::ItemIsMovable);

item->setPen(pen);

addItem(item);

如果采用的自定义绘图，则是类似如下的代码(仅仅是示例用来表明方案，实现上根据实际情况可能采用效率更高的绘制)

QpointF tmpStart(ptStart.dx, ptStart.dy);

QpointF tmpEnd(ptEnd.dx, ptEnd.dy)

QPainter painter(this->viewport());

QPen pen(Qt::black);

painter.setPen(pen);

painter.setBrush(Qt::black);

painter.drawLine(tmpStart, tmpEnd);

}

场景接口类必须与具体的UI操作结合来实现绘图，绘图操作与场景渲染接口类CsceneRender结合的代码如下：

以Qt的视图绘制类为例：

class CView:public QgraphicsView, public CSceneRender

{

Q\_OBJECT

….

virtual void DrawScene();

virtual void DrawPrimitive(CGeoLine\* pData);

virtual void DrawPrimitive(CGeoCircle\* pData);

virtual void DrawPrimitive(CGeoArc\* pData);

virtual void DrawPrimitive(CGeoLWPolyline\* pData);

virtual void DrawPrimitive(CGeoEllipse\* pData);

virtual void DrawPrimitive(CGeoText\* pData);

virtual void DrawPrimitive(CGeoMulripler\* pData);

}

该场景类也定义了一些基本的场景图元添加删除更新操作，方便命令行功能模块调用这些接口函数实现场景相关的操作诸如添加直线等。如

AddLine()

DelObject()

命令行功能如果接收到输入添加直线的命令，解析后直接调用AddLine()，该函数自动把新增的直线添加到当前场景数据中，如果同时需要把添加的直线在视图上显示出来，则调用DrawPrimitive()函数即可。

场景渲染结构示意图如下：

1. 在Qt中实现自画基本图元的选取

选取基本图元的功能是基于自定义画图，如果采取Qt的预定义绘图，则内置了选取移动等功能，选取功能不需要再次实现。

进行图元选取之前，先必须建立基于坐标位置的图元索引，建立图元索引的方案描述如下：

1. 把整个场景分成n\*m个栅格区域，每个栅格的编号都是根据实际坐标范围来映射，每个编号本身的含义就对应着一定的栅格区域范围。每个栅格的大小固定的。
2. 计算栅格编号的方法：先计算出图形区域的最小坐标点(Xmin,Ymin)和最大坐标点(Xmax,Ymax)。最小坐标点向下取整，最大坐标点向上取整。为了冗余，也可以在最小坐标点再向下偏移几个单位，最大坐标点再向上偏移几个单位。然后计算出栅格的区域范围，其中宽度数据width=(Xmax- Xmin)/n，高度数据为height=(Ymax- Ymin)/m。计算出来的宽度和高度向上取整。对于任意坐标(X,Y),则X方向的编号为NOx = (X- Xmin)/width, Y方向的编号为NOy = (Y- Ymin)/ height。两者编号都取整。实际编号就是两者编号的组合，如NO = (NOx<<16)+ NOy。(设定编号为32为无符号整形)。
3. 如果图元位于栅格里面、与栅格相交、包含栅格。则这个图元就属于这个栅格。因此一个图元可能对应着多个栅格。建立一个编号与图元的多重映射，关键字为栅格编号，值为图元列表。
4. 当图元发生平移或者选择时，需要更新该图元所在的栅格信息。

选取的功能描述如下；

1. 把选取的屏幕坐标转化为图形坐标
2. 根据图形坐标计算该坐标所在的栅格ID
3. 根据栅格ID找出位于该栅格的所有图元，然后继续下一步找出满足条件的图元。
4. 把这个栅格内的所有图元的区域与选取的坐标点进行比较，如果坐标点位于区域内或者区域边界，则满足要求。如果有多个图元满足要求，则选取的原则为：就小不就大，当前层优先。意思如果多个图元都包含选取点时，则选取区域最小的那个，如果区域相同的有多个，则选取当前层的那个。另外如果当前层也有区域相同的多个(正常情况下这应该是错误，因为这是重复图元)则选列表中的第一个。
5. 选定了基本图元，把图元的边界突出显示
6. 对自画图元的平移旋转等基本操作

平移旋转操作本身不是难事，现在主要说明在鼠标操作的情况:

1. 选定需要操作的图元
2. 捕获鼠标的点击与移动事件，计算出图元要平移或者旋转的屏幕距离
3. 根据平移或者旋转操作把计算出来的屏幕距离转换成实际的平移距离或者选择的角位移
4. 调用平移或者旋转函数
5. 要求刷新显示

如果是命令行，则只有后面的4，5两步。

目前需要把针对图元旋转平移的功能封装起来。

1. 彻底替换以前的采用Qt预定义图元操作以及文档中的场景数据