# PLA开发文档——制作插件

# 文档概述

本文档为使用本软件进行开发的编程人员提供必要的插件制作信息、插件制作示例等，可作为软件编程人员的编程使用手册使用。

# 运行环境

## 硬件运行环境

## 软件运行环境

# 应用开发环境

VS2010，QT4.8.6

# 插件制作流程

## 插件制作的基本流程

1. 新建一个空插件，实现在应用软件平台上执行菜单或工具条的命令完成用户的操作。
2. 实现矢量数据的读取，包括几何信息和属性数据的访问。

## 插件制作的实例开发过程

### 创建一个空插件

在VS201中点击【文件->新建】，打开新建对话框，如图4-1所示，选择Qt4 Projects，选择Qt Library，选择插件的位置路径，输入插件名称，如：PluginDemo。点击确定。

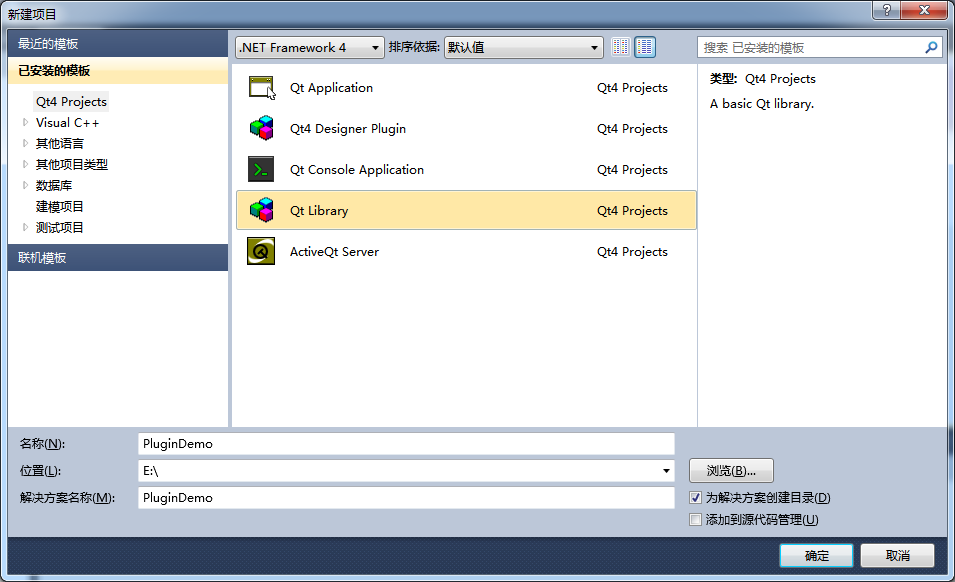


图 4-1 新建项目对话框

在弹出的对话框中，在Project Setting中根据功能需求选择勾选项，如图4-2所示。

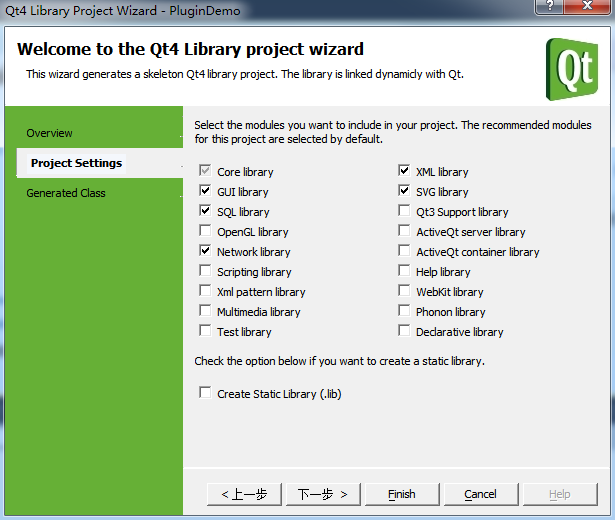


图4-2 插件工程向导

点击Finish，完成创建。系统会创建一个名称为PluginDemo的插件工程，如图4-3所示。

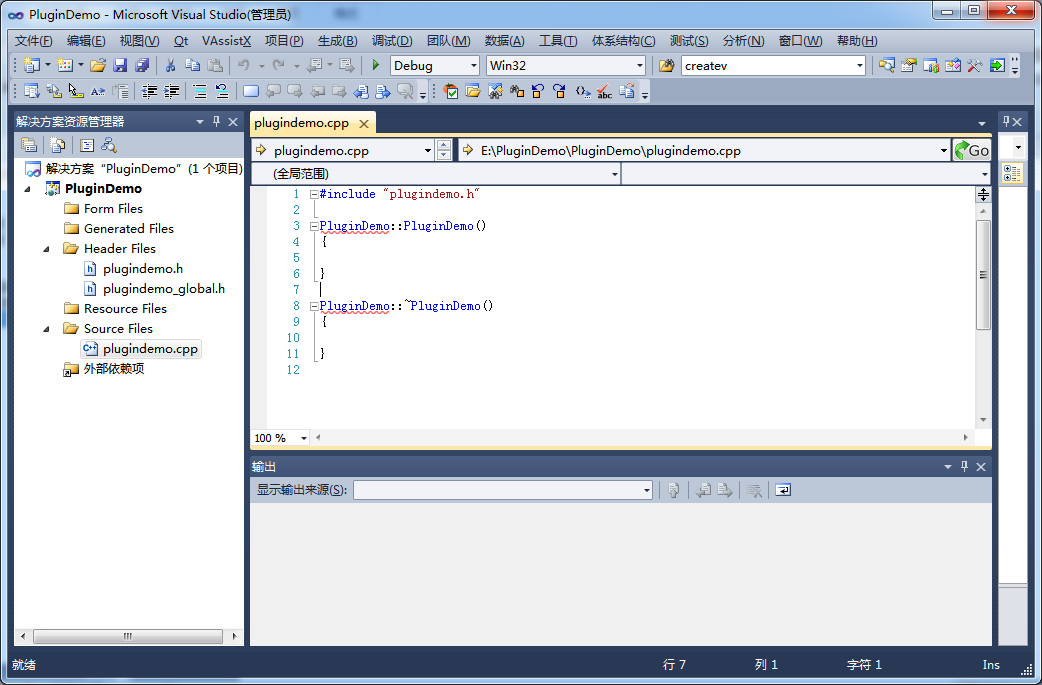


图4-3 创建插件

单击菜单，【项目->属性】，打开PluginDemo属性页，点击【C/C++->常规】，在附加包含目录中输入包含目录，如图4-4所示。

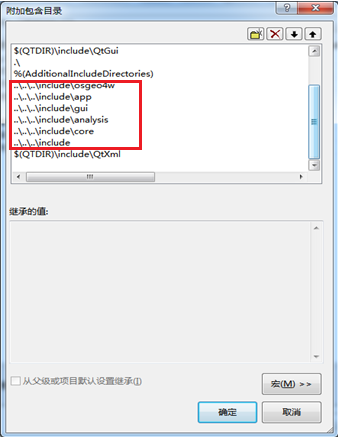


图4-4 附加包含目录

点击【C/C++->预处理器】，在预处理器定义中添加5个定义，如图4-5所示。

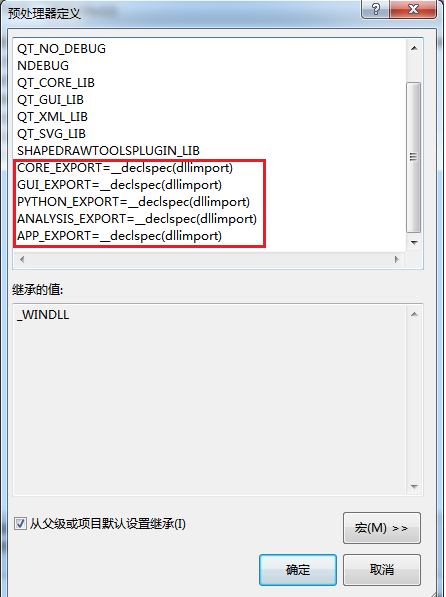


图4-5 预处理器定义

点击【链接器->常规】，在附加库目录中输入Lib库目录，如图4-6所示。

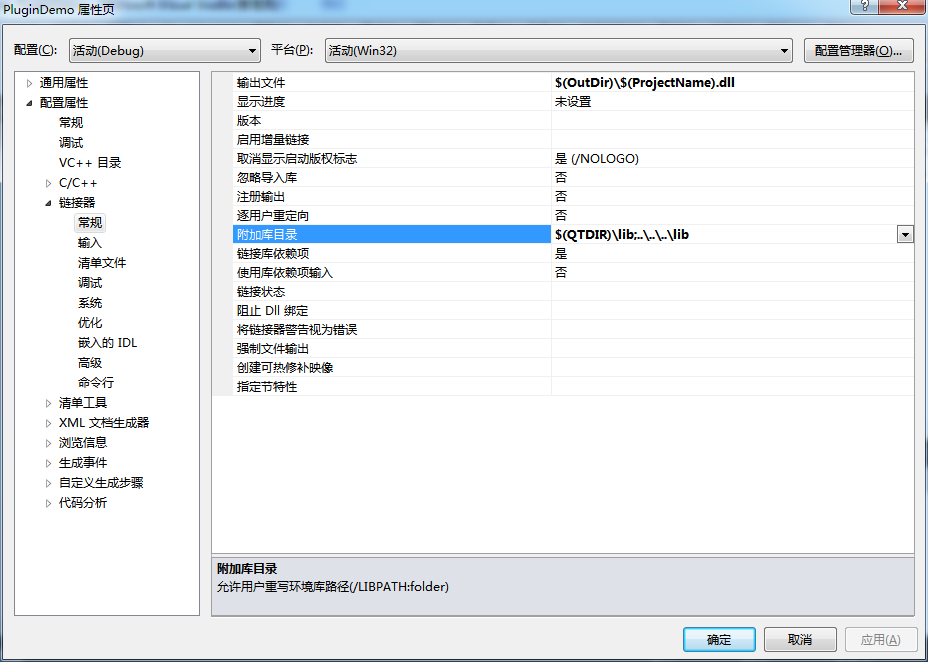


图4-6 附加库目录

点击【链接器->输入】，在附加依赖项中输入依赖lib文件，如图4-7所示。

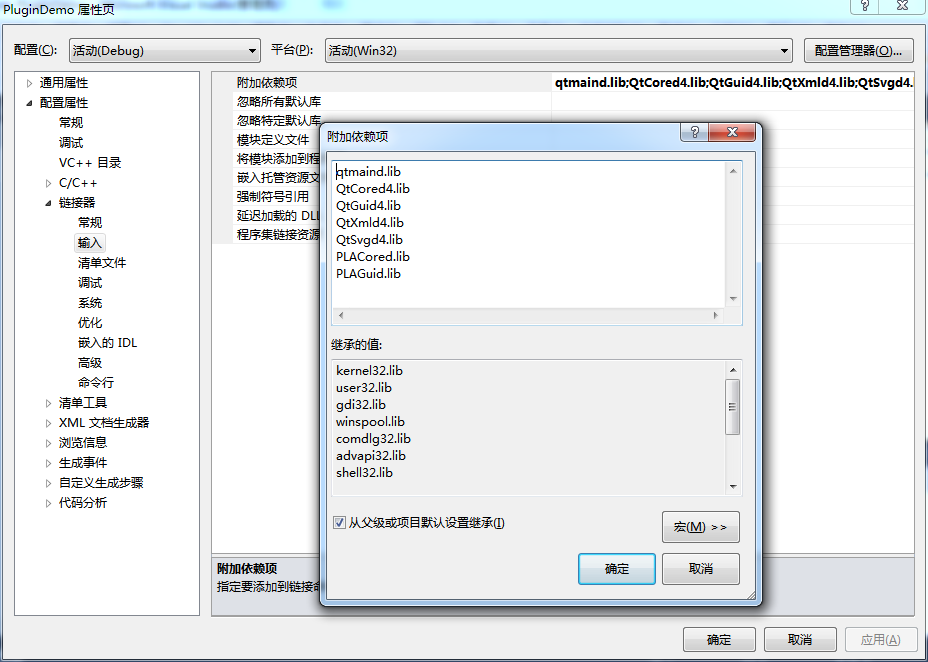


图4-7 附加依赖项

点击【配置属性->常规】，在输出目录中，输入插件的输出目录（..\..\..\bin\plugins），如图4-8所示。

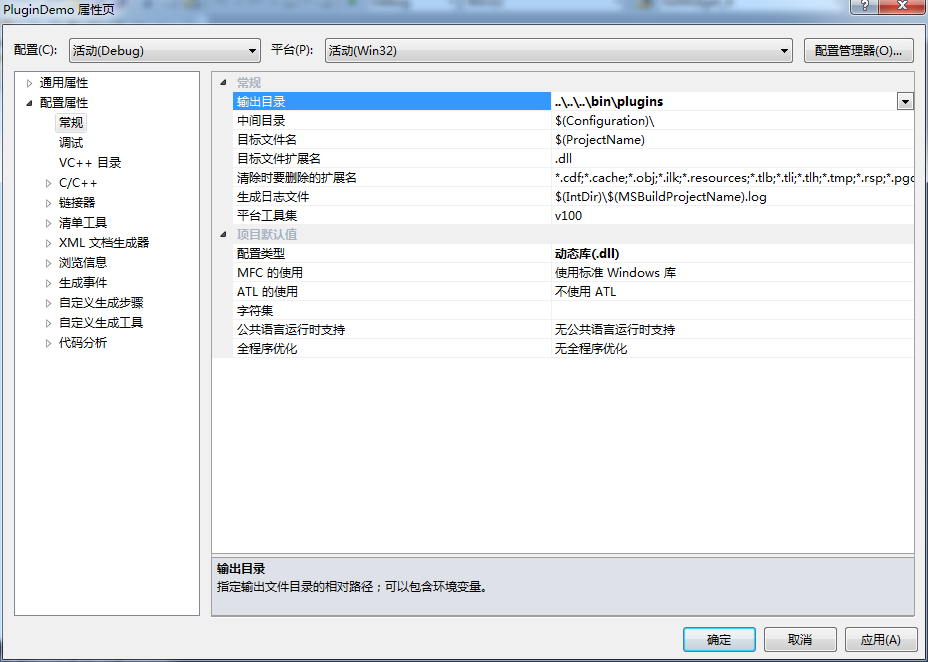


图4-8输出目录

点击【配置属性->调试】，在命令中，输入插件的输出位置（..\..\..\bin\PLAMaind.exe），如图4-9所示。

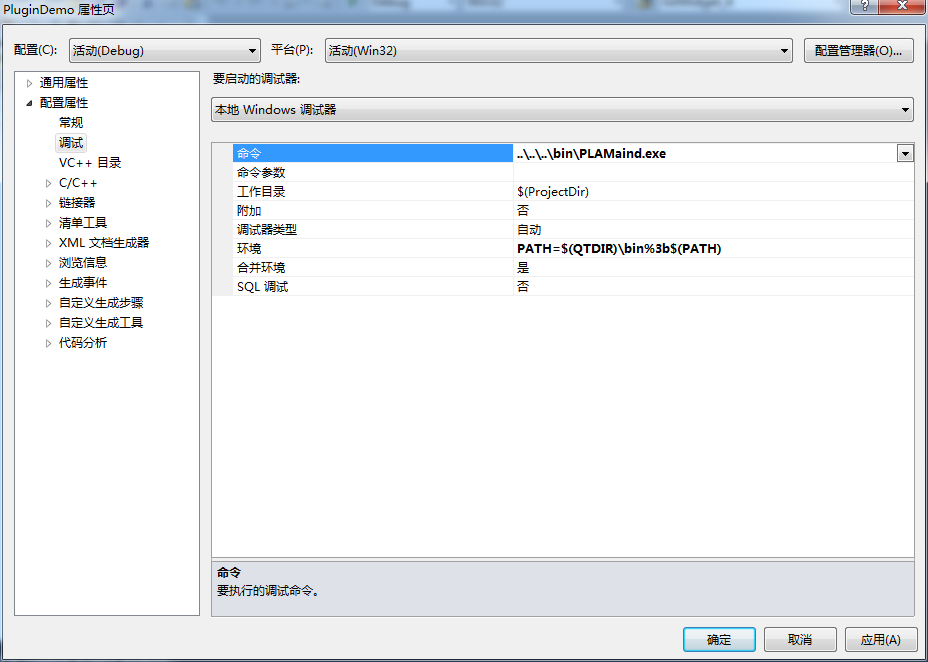


图4-9 命令

点击应用，确定。完成配置。

在plugindemo.h中添加代码，由于这里是创建一个独立的插件，所以需要对类进行修改，不需要 PLUGINDEMO\_EXPORT，在该类中定义继承关系，在该类中声明函数，并且把相应的头文件包含进来。声明一个QgisInterface类和该类指针，具体代码如下。

#include "plugindemo\_global.h"

#include <QObject>

#include "qgisplugin.h"

#include <qgisinterface.h>

class QgisInterface;

class PluginDemo: public QObject, public QgisPlugin

{

Q\_OBJECT

public:

PluginDemo(QgisInterface \* theInterface);

~PluginDemo();

public :

virtual void initGui();

void run();

void unload();

void help();

private:

QgisInterface \*mQGisIface;

}

在plugindemo.cpp中，在构造函数中进行初始化，添加槽函数的定义，具体代码如下。

PluginDemo::PluginDemo(QgisInterface \* theInterface)

: QgisPlugin(sName,sDescription,sCategory,sPluginVersion,sPluginType)

, mQGisIface(theInterface){}

void PluginDemo::initGui(){}

void PluginDemo::run(){}

void PluginDemo::help(){}

void PluginDemo::unload(){}

在plugindemo.cpp中，设置插件的名称，插件描述，插件类别，插件类型，插件版本，插件图标以及插件的销毁。

static const QString sName = QObject::tr( "插件示例" );

static const QString sDescription = QObject::tr( "插件描述" );

static const QString sCategory = QObject::tr( "Vector" );

static const QString sPluginVersion = QObject::tr( "Version 1.0" );

static const QgisPlugin::PLUGINTYPE sPluginType = QgisPlugin::UI;

static const QString sPluginIcon = ":/images/themes/default/mIconNew.png";//选定图标，存放在插件目录Resources下

QGISEXTERN QgisPlugin \* classFactory( QgisInterface \* theQgisInterfacePointer ) //框架

{

return new PluginDemo( theQgisInterfacePointer );

}

// Return the name of the plugin - note that we do not user class members as

// the class may not yet be insantiated when this method is called.

QGISEXTERN QString name() //插件名称

{

return sName;

}

// Return the description

QGISEXTERN QString description() //插件描述

{

return sDescription;

}

// Return the category

QGISEXTERN QString category() //插件类别

{

return sCategory;

}

// Return the type (either UI or MapLayer plugin)

QGISEXTERN int type() //插件类型

{

return sPluginType;

}

// Return the version number for the plugin

QGISEXTERN QString version() //插件版本

{

return sPluginVersion;

}

QGISEXTERN QString icon() //插件图标

{

return sPluginIcon;

}

// Delete ourself

QGISEXTERN void unload( QgisPlugin \* thePluginPointer ) //插件销毁

{

delete thePluginPointer;

}

至此，已经制作完成了一个新的空插件。

下面介绍如何在应用平台上添加工具图标，首先，在plugindemo.h中，声明两个类QAction和QToolBar。

class QAction;

class QToolBar;

声明一个私有的指针。

private:

QAction \* mAction

在plugindemo.cpp中，在函数initGui()中，添加代码如下：

void PluginDemo::initGui()

{

if(mQGisIface == 0){return;}

mAction = new QAction(mQGisIface);

mAction->setIcon(QIcon(tr(":/images/themes/default/mIconNew.png")));

mAction->setText("插件测试");

QMenu\* layerMenu = mQGisIface->layerMenu();

if(layerMenu != 0){layerMenu->addAction(mAction);}

mQGisIface->layerToolBar()->addAction(mAction);

}

在函数unload()中，添加代码如下：

void PluginDemo::unload()

{

mQGisIface->layerToolBar()->removeAction(mAction);

QMenu\* layerMenu = mQGisIface->layerMenu();

if(layerMenu != 0){layerMenu->removeAction(mAction);}

delete mAction;

}

运行结果如下：点击【插件->管理并安装插件】，如图4-10所示：

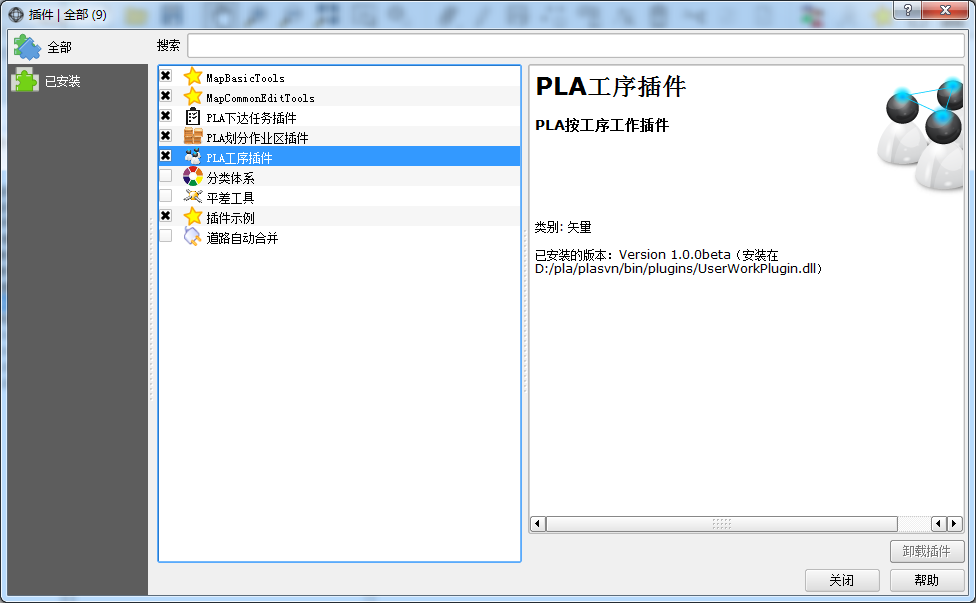


图4-10 管理并安装插件

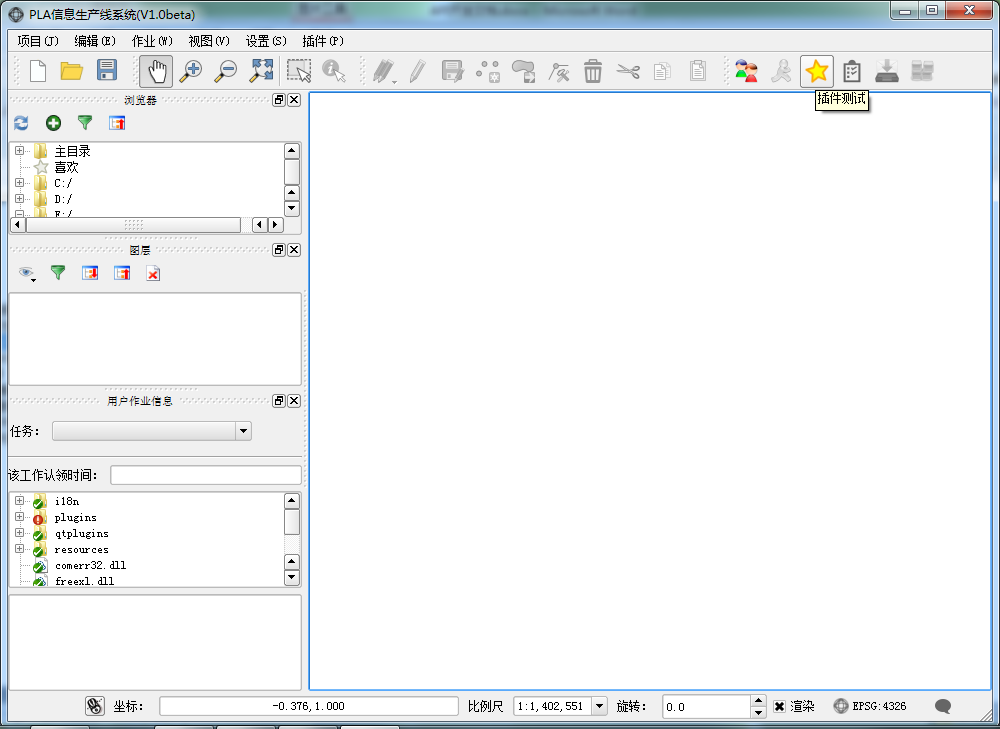


图4-11 运行结果

### 响应工具条命令

1. 设置工具条的可用性。

在plugindemo.h中声明一个私有函数。

void deactiveCombineTool();

在plugindemo.cpp的initGui()中对deactiveCombineTool()进行初始化，在.cpp文件中对deactiveCombineTool()进行定义。

void PluginDemo::deactiveCombineTool()

{

mpActionOutput->setEnabled(false); //false为工具不可用，参数为ture时为可用

}

运行结果如图4-12所示：

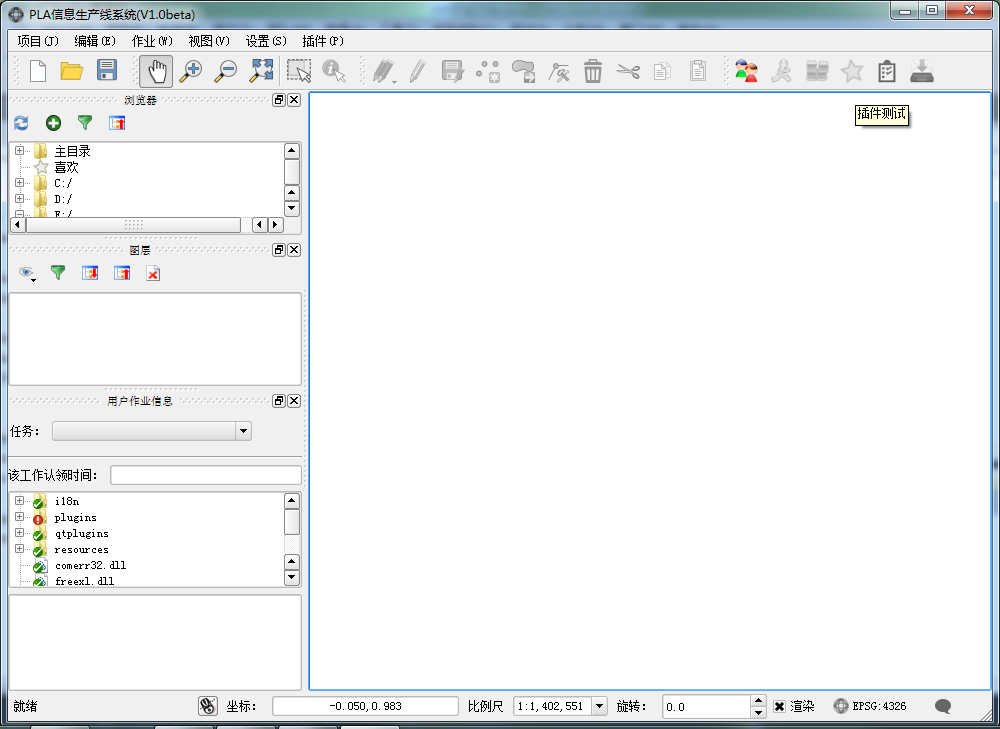


图4-12 工具不可用运行结果

当加载图层后，工具切换状态。添加信号响应行为，包含响应的头文件。

#include <qgslayertreeview.h>

connect(mQGisIface->layerTreeView(),SIGNAL(currentLayerChanged(QgsMapLayer\*)),this,SLOT(activeCombineTool(QgsMapLayer \*)));

添加函数activeCombineTool(QgsMapLayer \*layer)的实现代码。

void PluginDemo::activeCombineTool(QgsMapLayer \*layer)

{

if (layer)

{

layer = mQGisIface->layerTreeView()->currentLayer();

if (layer->type()==QgsMapLayer::VectorLayer)

{

QgsVectorLayer \*vlayer= (QgsVectorLayer \*)layer;

mAction->setEnabled(true);

}

else

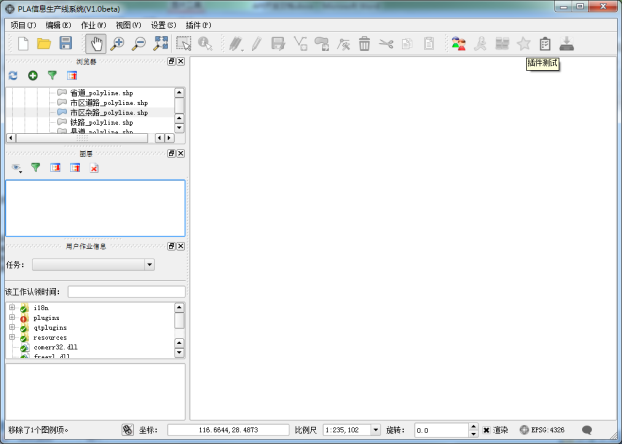
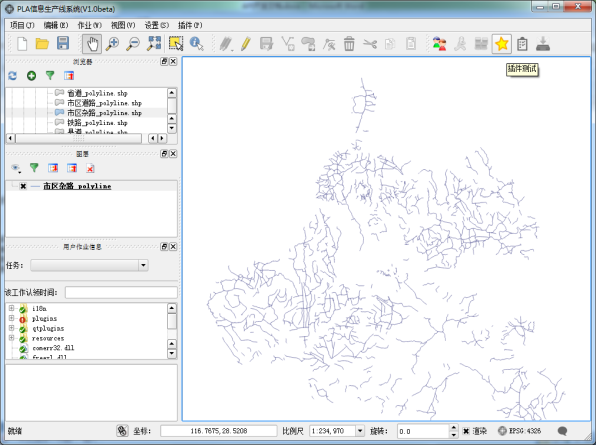
deactiveCombineTool();

}

else deactiveCombineTool();

}

运行结果图4-13所示：

1.加载图层前 2.加载图层后

图4-13 运行结果

2) 响应工具条命令

在plugindemo.h中声明一个槽函数。

void onDemo();

在plugindemo.cpp的initGui()中创建工具行为，在.cpp文件中对onDemo()进行定义，之后可以根据需求，在该函数中添加代码实现。

connect(mAction, SIGNAL( triggered()), this, SLOT( onDemo() ) );

void PluginDemo:: onDemo (){}

### 矢量数据引擎

#### 矢量数据的读取

矢量图层的读取是从文件系统中直接获取矢量图层，根据矢量图层文件的全路径可以直接创建一个矢量层对象，例如，从文件系统中获取“国道\_polyline.shp”，创建一个对象pNewLayer。具体代码执行如下：

在plugindemo.h中包含头文件，以及声明一个画布指针。

#include <qgsmaplayerregistry.h>

#include <qgsmapcanvas.h>

QgsMapCanvas \*mCanvas;

在plugindemo.cpp中，在initGui()中创建一个画布对象，mCanvas= new QgsMapCanvas(0,0)。

在onDemo()函数中添加代码如下：

void PluginDemo::onDemo()

{

QString myLayerPath ="E:\\3\\1导Ì?航?数ºy据Y"; // 此处可换成自己的矢量文件

QString myLayerBaseName = "国道\_polyline.shp ";

QString myProviderName = "ogr";

QgsVectorLayer \*pNewLayer = new QgsVectorLayer(myLayerPath,myLayerBaseName,myProviderName);

if (!pNewLayer->isValid())

{

QMessageBox::information(NULL,QString(tr("图层加载")),QString(tr("图层加载不成功")));

}

if (pNewLayer->isValid())

{

QgsMapLayerRegistry::instance()->addMapLayer(pNewLayer, TRUE);//实例化地图注册类之后，然后增加图层。

mCanvas->setExtent(pNewLayer->extent());//设置画布的extent

QList<QgsMapCanvasLayer> pLayerSet;

pLayerSet.append(QgsMapCanvasLayer(pNewLayer)); //设置画布的图层数据集

mCanvas->setLayerSet(pLayerSet);

}

else

{

return;

}

}

点击框架工具图标，运行结果如下;

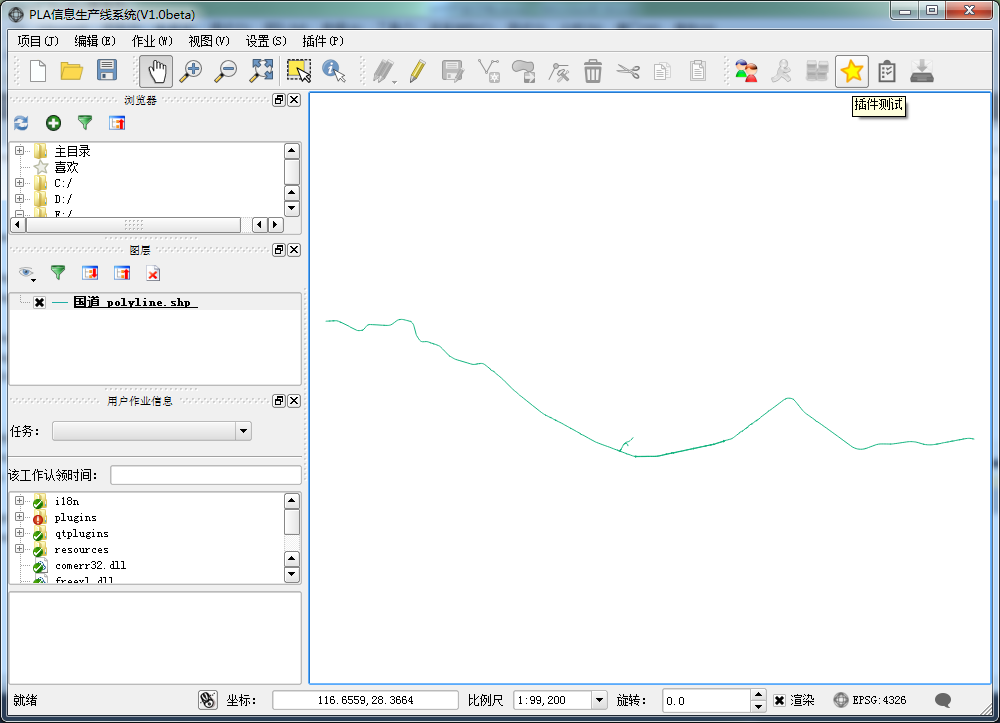


图4-13 矢量数据加载运行结果

#### 获取矢量数据信息

首先，要获取加载到画布上的图层，将获取到的图层集存放在一个列表容器中，通过对列表容器的遍历，定义一个变量指针，获取到加载的每一个图层。具体代码如下：

mCanvas = mGisInterface->mapCanvas();

QList<QgsMapLayer\*>layers;

layers = mCanvas->layers();

foreach(QgsMapLayer \*maplayer,layers){}

这里得到的是加载的画布上的所有图层，由于我们要做矢量数据引擎，所以需要对图层进行转换。

QgsVectorLayer\* vlayer = (QgsVectorLayer\*)maplayer

然后，从矢量图层中获取到要素，矢量图层要素包含几何信息和属性信息。

从图层中获取要素有两种方法：

1. 通过图层指针直接获取全部要素，得到要素列表，通过遍历要素列表，得到每一个要素。具体代码如下：

vlayer->selectAll();

QgsFeatureList featurelist = vlayer->selectedFeatures();

for (int i=0; i<featurelist.size();i++)

{

QgsFeature f = featurelist.at(i);

}

1. 通过定义一个迭代器，迭代得到图层中的全部要素。具体代码如下：

QgsFeature f;

QgsFeatureIterator fi = vlayer->getFeatures();

while(fi.nextFeature(f)){}

##### 4.2.3.2.1 获取矢量数据的几何信息

获取到要素后，得到要素的几何信息，QgsGeometry\* geo = f.geometry();。

首先对所加载的图层进行判断，是点图层，线图层还是面图层。多点组成线，多线组成面。

函数geometryType()获取图层类型。具体代码如下：

if ( vlayer->geometryType() == QGis::GeometryType::Polygon)

{

QList<QgsPoint> points;

QgsPolygon polygon = geo->asPolygon();

foreach(QgsPolyline line, polygon)

{

foreach(QgsPoint point,line)

{

points.push\_back(point);

double x = point.x();

double y = point.y();

QString X= X.number(x,false,10);

QString Y= Y.number(y,false,10);

}

}

}

if ( vlayer->geometryType() == QGis::GeometryType::Line)

{

QList<QgsPoint> points;

QgsPolyline line = geo->asPolyline();

foreach(QgsPoint point,line)

{

points.push\_back(point);

double x = point.x();

double y = point.y();

QString X= X.number(x,false,10);

QString Y= Y.number(y,false,10);

}

}

else if ( vlayer->geometryType() == QGis::GeometryType::Point)

{

QgsPoint point = geo->asPoint();

double x = point.x();

double y = point.y();

QString X= X.number(x,false,10);

QString Y= Y.number(y,false,10);

}

添加一个listWidget控件存放输出的点X和Y坐标，具体代码如下：

QListWidgetItem\* item3= new QListWidgetItem(X+" "+Y);

ui.listWidget\_3->addItem(item3);

运行结果如图4-14所示：

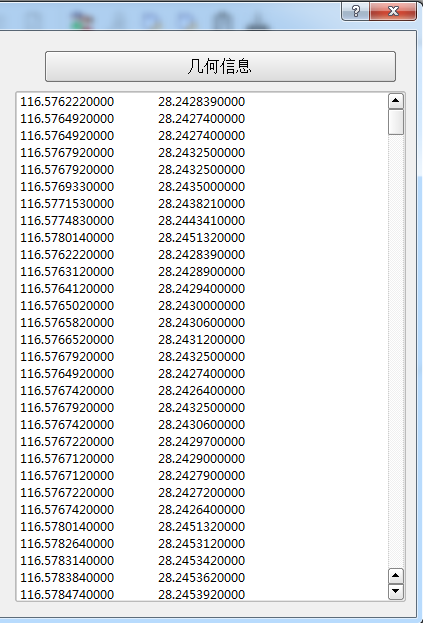


图4-14 几何信息输出结果

##### 获取矢量数据的属性信息

1. 获取到要素后，要素的属性信息包括字段（Fields）和字段值（attributes）。声明两个容器分别存放单个要素的字段名和字段值，以及图层所有要素的字段名和字段值。具体代码如下：

QMap<QString, QString> featureValue;

QList<QMap<QString, QString>> featureValues;

QgsFeature feature;

QgsFeatureIterator fi = vlayer->getFeatures();

while (fi.nextFeature(feature))

{

const QgsFields &fields = vlayer->pendingFields(); //从图层获取字段信息

QgsField field;

for(int j = 0; j <fields.count(); j++)

{

fieldName = fields[j].name(); //获取要素的字段名

fieldValue = feature.attribute(j).toString(); //获取要素的字段值

featureValue.insert(fieldName,fieldValue); //存放单个要素的字段名和字段值

}

featureValues.push\_back(featureValue);

}

添加一个listWidget控件存放字段名称和字段值的输出结果，具体代码如下：

QListWidgetItem\* item1 = new QListWidgetItem(fieldName+" "+fieldValue);

ui.listWidget->addItem(item1);

运行结果如图4-15所示：

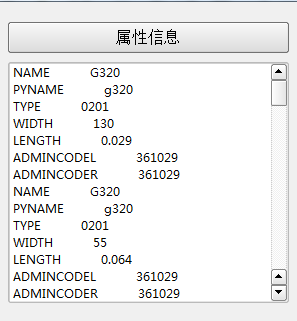


图4-14 图层的属性信息

1. 根据字段名称（Fields）获取到字段值（attributes）。应用int fieldNameIndex( const QString& fieldName ) const方法实现。例如以“NAME”字段为例，获取该字段的字段值。在要素迭代过程中添加如下代码：

QString refNamFielde[] = {"NAME"};

int nameIndexN = feature.fields()->fieldNameIndex(refNamFielde);

QVariant v1 = feature.attribute(nameIndexN);

1. 综合使用：由comobox控件来显示当前加载的图层，当选中某一个图层时，用listWidget控件来显示该图层的所有字段，当选中一个字段时，用另一个listWidget控件显示该字段对应的字段值。

首先，第一步，在comobox控件中显示当前加载的图层，将头文件包含到plugind.h中，具体代码如下：

#include "qcombobox.h"

void plugind::qvector()

{

mCanvas = mGisInterface->mapCanvas();

QList<QgsMapLayer\*>layers;

layers = mCanvas->layers();

foreach(maplayer,layers)

{

QgsVectorLayer\* vlayer =(QgsVectorLayer\*)maplayer;

ui.comboBox->addItem(vlayer->name());

}

}

第二，添加信号槽，响应comobox控件行为。

connect(ui.comboBox, SIGNAL(currentIndexChanged(int)), this, SLOT(on\_sel\_sex(int)));

第三，实现on\_sel\_sex(int)槽函数，具体代码如下：

void plugind::on\_sel\_sex(int index)

{

QgsMapLayer \*maplayer;

maplayer = mCanvas->layer(index); //索引当前选择图层

QgsVectorLayer\* vlayer =(QgsVectorLayer\*)maplayer;

if (ui.listWidget\_4) {ui.listWidget\_4->clear(); }

const QgsFields &fields = vlayer->pendingFields();

QString fieldName;

for(int j = 0; j <fields.count(); j++)

{

fieldName = fields[j].name();

ui.listWidget\_4->addItem(fieldName);

}

}

第四，根据选择的字段名称，获取对应的字段值，可以使用void uniqueValues( int index, QList<QVariant> &uniqueValues, int limit = -1 )函数实现。添加信号槽：

connect(ui.listWidget\_4,SIGNAL(clicked(QModelIndex)),this,SLOT(fieldsClicked(QModelIndex)));

实现fieldsClicked(QModelIndex)槽函数，获取指定字段的属性值，输出到另一个listWidget控件中。具体实现代码如下：

void plugind::fieldsClicked(QModelIndex index)

{

int idx = index.row();

int limit = -1;

QgsVectorLayer\* vlayer =(QgsVectorLayer\*)maplayer;

if (ui.listWidget\_2) {ui.listWidget\_2->clear();}

QList<QVariant> values;

vlayer->uniqueValues( idx, values, limit ); //获取指定字段的属性值

for ( int i = 0; i < values.size(); i++ )

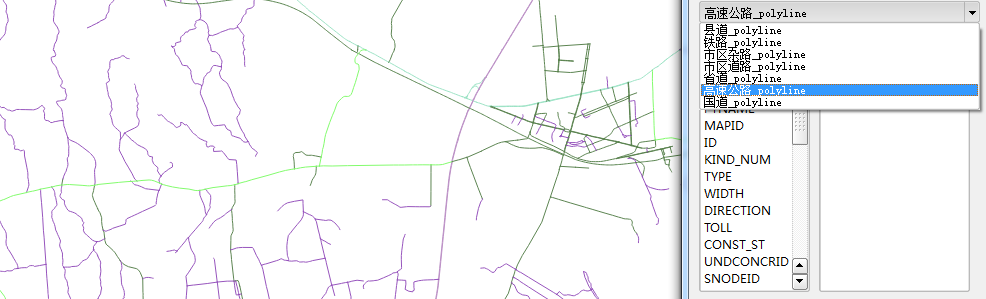
{

QListWidgetItem \*item = new QListWidgetItem(values[i].toString());

ui.listWidget\_2->addItem(item);

}

}

运行结果如图4-16所示

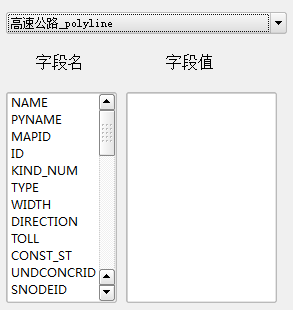
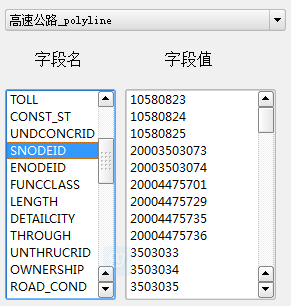
 

图4-16 运行结果

### 鼠标响应事件

#### 鼠标响应事件概述

鼠标响应事件包括，鼠标移动事件（mouseMoveEvent），鼠标按下事件（mousePressEvent），鼠标释放（ mouseReleaseEvent），鼠标按下又包括单击和双击。

通过[setMouseTracking](http://www.kuqin.com/qtdocument/qwidget.html#setMouseTracking) ()设置属性值并且通过[hasMouseTracking](http://www.kuqin.com/qtdocument/qwidget.html#hasMouseTracking) ()来获得属性值。

下面列举了鼠标的基本事件响应函数，添加两个Lable控件来显示鼠标响应事件结果，将相应的头文件包含进来。

#include "qlabel.h"

#include "QMouseEvent"

1. 鼠标响应单击事件

鼠标单击事件包括，左键单击事件，右键单击事件和中键单击事件。具体代码实现：

void MouseEvent::mousePressEvent ( QMouseEvent \* e ) //鼠标响应单击事件

{

QString str="("+QString::number(e->x())+","+QString::number(e->y())+")"; //显示屏幕坐标 if(e->button()==Qt::LeftButton)

{

ui.labelStatus->setText("左键按下"+str);

}

else if(e->button()==Qt::RightButton)

{

ui.labelStatus->setText("右键按下"+str);

}

else if(e->button()==Qt::MidButton)

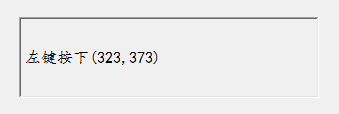
{

ui.labelStatus->setText("中键按下"+str);

}

}

运行结果如下图4-17所示：



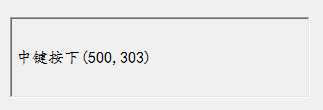
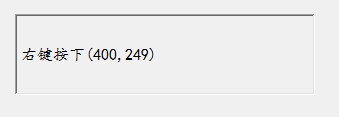


图4-17 鼠标响应单击事件运行结果

1. 鼠标响应双击事件

鼠标双击事件包括左键双击事件、右键双击事件和中键双击事件。具体代码实现：

void MouseEvent::mouseDoubleClickEvent( QMouseEvent \* e ) //鼠标响应双击事件

{

QString str="("+QString::number(e->x())+","+QString::number(e->y())+")";

if(e->button()==Qt::LeftButton)

{

ui.labelStatus->setText("左键双击"+str);

}

else if(e->button()==Qt::RightButton)

{

ui.labelStatus->setText("右键双击"+str);

}

else if(e->button()==Qt::MidButton)

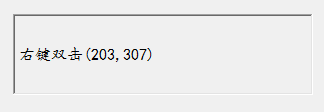
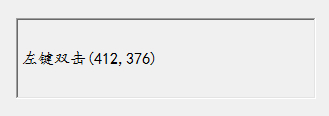
{

ui.labelStatus->setText("中键双击"+str);

}

}

运行结果如下图4-18所示：



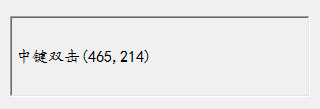


图4-18 鼠标响应双击事件运行结果

1. 鼠标响应移动事件

当鼠标在屏幕上移动时，发生事件，下面通过鼠标移动来获取鼠标位置坐标。具体代码如下：

void MouseEvent::mouseMoveEvent ( QMouseEvent \* e ) //鼠标响应移动事件

{

QString str="("+QString::number(e->x())+","+QString::number(e->y())+")";

//1.获取局部鼠标位置

ui.labelMousePos->setText("当前鼠标位置坐标"+str);

//2.获取全局鼠标位置

QPoint coursePoint;

coursePoint = QCursor::pos(); //获取当前鼠标位置

ui.labelMousePos->setText("当前鼠标位置坐标"+str);

}

运行结果如下图4-19所示：



图4-19 鼠标响应移动事件运行结果

1. 鼠标响应释放事件

鼠标释放事件包括，左键释放，右键释放和中键释放。具体实现代码如下：

void MouseEvent::mouseReleaseEvent ( QMouseEvent \* e ) //鼠标响应释放事件

{

QString str="("+QString::number(e->x())+","+QString::number(e->y())+")";

if(e->button()==Qt::LeftButton)

{

ui.labelStatus->setText(tr("鼠标左键释放：")+str);

}

else if(e->button()==Qt::RightButton)

{

ui.labelStatus->setText(tr("鼠标右键释放：")+str);

}

else if(e->button()==Qt::MidButton)

{

ui.labelStatus->setText(tr("鼠标中键释放：")+str);

}

}

运行结果如下图4-20所示：



图4-20 鼠标响应双击事件运行结果

#### 使用QgsMapTool类实现鼠标响应

类QgsMapTool是针对所有地图工具的一个抽象基类，通常在使用鼠标进行交互的时候使用。下图4-21是简单的类继承关系。



图4-21 QgsMapTool类继承关系

QgsMapTool基类提供鼠标事件方法，主要是转换点的坐标系。它的很多直接继承类重载了QgsMapTool基类的方法，QgsMapToolAdvancedDigitizing类重载了QgsMapTool基类的鼠标事件方法（[canvasMoveEvent](http://qgis.org/api/2.8/classQgsMapTool.html#aa0cab2195496cb4f3f3abf16b6d75093) (QMouseEvent \*e)，[canvasPressEvent](http://qgis.org/api/2.8/classQgsMapTool.html#ad84ed827c5f410afa99fc190ffe1209c)(QMouseEvent \*e)，[canvasDoubleClickEvent](http://qgis.org/api/2.8/classQgsMapTool.html#a4d1ce211ce8e642e453ffdf614e29a02) (QMouseEvent \*e)），并且重新定义了新的鼠标事件方法（[canvasMapMoveEvent](http://qgis.org/api/2.8/classQgsMapTool.html#aa0cab2195496cb4f3f3abf16b6d75093) (QMouseEvent \*e)，[canvasMapPressEvent](http://qgis.org/api/2.8/classQgsMapTool.html#ad84ed827c5f410afa99fc190ffe1209c)(QMouseEvent \*e)，canvasMapReleaseEvent( QgsMapMouseEvent\* e )，[canvasMapDoubleClickEvent](http://qgis.org/api/2.8/classQgsMapTool.html#a4d1ce211ce8e642e453ffdf614e29a02) (QMouseEvent \*e)）。这两种方法前者是捕捉到鼠标事件，对它进行转换成地图坐标的处理，发送到虚函数方法。后者是在前者处理后的基础上重新定义。QgsMapToolEdit继承类主要实现当前加载图层的编辑，当我们需要对加载的图层中的要素进行处理时，如：删除部分要素、移动一些要素、对要素的结点进行操作、旋转要素等，我们需要继承该类。QgsMapToolCapture继承类主要实现对当前加载图层中各种要素的捕捉，如：向图层中添加要素，填充，对要素进行分割，对图层部分进行分割等，需要继承该类。

##### 4.2.4.3.1 QgsMapTool基类鼠标事件响应

类QgsMapTool是针对所有地图工具的一个抽象基类，地图工具是用户操作地图画布的交互式工具。

1）QgsMapTool类信号函数有：

void [activated](http://qgis.org/api/2.8/classQgsMapTool.html#a6fdb2db81a67113694d9aa529ac0d293) ()； 地图工具被激活就发出信号

void [deactivated](http://qgis.org/api/2.8/classQgsMapTool.html#a0b4dda4a618e78e1a7abcbaf14459599) ()； 地图工具没有激活就发出信号

void [messageDiscarded](http://qgis.org/api/2.8/classQgsMapTool.html#a25c8b7fee757fe46de0e3c58b4321c36) ()； 发出信号清除之前的信息

void [messageEmitted](http://qgis.org/api/2.8/classQgsMapTool.html#aefffff259de04a81ff5ab101a50a6663) (QString message, [QgsMessageBar::MessageLevel](http://qgis.org/api/2.8/classQgsMessageBar.html#a74456a59f4e5f60f994c54cf3de177da)=[QgsMessageBar::INFO](http://qgis.org/api/2.8/classQgsMessageBar.html#a74456a59f4e5f60f994c54cf3de177daaad13ded9a5513a187772fca45d1b2955))； 发出信息

2）QgsMapTool类相关公有成员函数有：

virtual void [canvasDoubleClickEvent](http://qgis.org/api/2.8/classQgsMapTool.html#a4d1ce211ce8e642e453ffdf614e29a02) (QMouseEvent \*e)；鼠标双击事件

virtual void [canvasMoveEvent](http://qgis.org/api/2.8/classQgsMapTool.html#aa0cab2195496cb4f3f3abf16b6d75093) (QMouseEvent \*e)；鼠标移动事件

virtual void [canvasPressEvent](http://qgis.org/api/2.8/classQgsMapTool.html#ad84ed827c5f410afa99fc190ffe1209c) (QMouseEvent \*e)； 鼠标按下事件

virtual bool [isEditTool](http://qgis.org/api/2.8/classQgsMapTool.html#a08131fa327c7c5cfb24bfba9150c78e0) ()； 判断当前编辑状态

void  [setAction](http://qgis.org/api/2.8/classQgsMapTool.html#a861b5670efeb10e9356fea9dbac57641) (QAction \*[action](http://qgis.org/api/2.8/classQgsMapTool.html#aa3168575e1bb517a141bc19c522c0e6f)) ； 关联行为与地图工具

void [setButton](http://qgis.org/api/2.8/classQgsMapTool.html#a538f79352c9689e6d3a47cc7f12d07c5) (QAbstractButton \*[button](http://qgis.org/api/2.8/classQgsMapTool.html#a4abea5661b13ba0efacac83d66e5240f))； 关联按钮与地图工具

virtual void [setCursor](http://qgis.org/api/2.8/classQgsMapTool.html#a7e7c23ef178baef9884d874620ea6968) (QCursor cursor)； 设置用户定义的鼠标样式

QString  [toolName](http://qgis.org/api/2.8/classQgsMapTool.html#a007fe906fccf2e2cd1d12ce273333b32) ()； 设置一个地图工具

3）QgsMapTool类相关保护成员函数有：

[QgsMapTool](http://qgis.org/api/2.8/classQgsMapTool.html#a22c0614112d1f7476bd9cc8a38889969) ([QgsMapCanvas](http://qgis.org/api/2.8/classQgsMapCanvas.html) \*[canvas](http://qgis.org/api/2.8/classQgsMapTool.html#a6d7b1a04ee0a1c8c7b2a8899ecbce728))； 构造函数将地图画布作为构造参数

QPoint  [toCanvasCoordinates](http://qgis.org/api/2.8/classQgsMapTool.html#aafeabd37aff50504a5480b3be805521e) (const [QgsPoint](http://qgis.org/api/2.8/classQgsPoint.html) &point)；将地图坐标转换成屏幕坐标

[QgsPoint](http://qgis.org/api/2.8/classQgsPoint.html) [toLayerCoordinates](http://qgis.org/api/2.8/classQgsMapTool.html#a4f7916307e7098786bdb3994d358922e) ([QgsMapLayer](http://qgis.org/api/2.8/classQgsMapLayer.html) \*layer, const QPoint &point)； 将屏幕坐标转换成图层坐标

[QgsPoint](http://qgis.org/api/2.8/classQgsPoint.html)  [toLayerCoordinates](http://qgis.org/api/2.8/classQgsMapTool.html#a49ee2795fa44b2bbe0d416ffa3ce2804) ([QgsMapLayer](http://qgis.org/api/2.8/classQgsMapLayer.html) \*layer, const [QgsPoint](http://qgis.org/api/2.8/classQgsPoint.html) &point)； 将地图坐标转换成图层坐标

[QgsPoint](http://qgis.org/api/2.8/classQgsPoint.html)  [toMapCoordinates](http://qgis.org/api/2.8/classQgsMapTool.html#a3340999b5da5020491201f590e4d7ce4) (const QPoint &point)； 将屏幕坐标转换成地图坐标

[QgsPoint](http://qgis.org/api/2.8/classQgsPoint.html)  [toMapCoordinates](http://qgis.org/api/2.8/classQgsMapTool.html#a6eac842e3ccfde02f97006f40c4238e2) ([QgsMapLayer](http://qgis.org/api/2.8/classQgsMapLayer.html) \*layer, const [QgsPoint](http://qgis.org/api/2.8/classQgsPoint.html) &point)；将图层坐标转换成地图坐标。

QgsMapTool类实现实例：通过QgsMapTool类实现响应鼠标释放向面图层添加要素。

首先，在PluginDemo中添加一个类EventMouse来实现向面图层中添加要素的功能。对该类进行修改，使之继承于QgsMapTool类。具体代码如下所示：

在eventmouse.h中

包含头文件：

#include "qgisinterface.h"

#include "qgsmaptool.h"

#include <qgsvectorlayer.h>

#include "qgsfeatureaction.h"

#include "qgsvectordataprovider.h"

#include "qgsfeature.h"

#include "qgsgeometry.h"

#include "qgsmapcanvas.h"

#include "qgsmapmouseevent.h"

包含类：

class QgsMapCanvas;

class QgsMapLayer;

class QMouseEvent;

class QRect;

class QPoint;

对类EventMouse进行修改：

class EventMouse :public QgsMapTool

{

Q\_OBJECT

public:

EventMouse(QgsMapCanvas\* canvas );

~EventMouse();

void canvasReleaseEvent( QMouseEvent \* e );

bool addFeature(QgsVectorLayer \*vlayer,QgsFeature \*f,bool showModal = true);

void activate() ;

private: };

在eventmouse.cpp中

对构造函数进行初始化：

EventMouse::EventMouse(QgsMapCanvas\* canvas ) : QgsMapTool( canvas ){}

添加布尔型函数addFeature(QgsVectorLayer \*vlayer,QgsFeature \*f,bool showModal = true)的实现，判断是否成功添加要素。返回true时，添加成功，返回false时，添加失败。

bool EventMouse::addFeature(QgsVectorLayer \*vlayer,QgsFeature \*f,bool showModal )

{

QgsFeatureAction \*action = new QgsFeatureAction(tr("向面图层中添加要素"),\*f,vlayer,-1,-1,this);

bool res = action->addFeature(QgsAttributeMap(),showModal);

if (showModal)

delete action;

return res;

}

设置工具图标的激活状态，实现函数void activate()。

void EventMouse::activate()

{

QgsVectorLayer \*vlayer = qobject\_cast<QgsVectorLayer \*>(mCanvas->currentLayer());

if (vlayer && vlayer->geometryType() == QGis::NoGeometry) //判断当前激活图层以及类型

{

QgsFeature f;

addFeature(vlayer,&f,false);

return;

}

QgsMapTool::activate();

}

响应鼠标释放事件，当鼠标释放时，向激活图层中添加要素。实现函数void canvasReleaseEvent (QMouseEvent \* e )。

void EventMouse::canvasReleaseEvent( QMouseEvent \* e )

{

QgsVectorLayer \*vlayer = qobject\_cast<QgsVectorLayer \*>(mCanvas->currentLayer());

if (!vlayer){return;}

QGis::WkbType layerWKBType = vlayer->wkbType();

QgsVectorDataProvider\* provider = vlayer->dataProvider();

if (!(provider->capabilities() & QgsVectorDataProvider::AddFeatures))

{

emit messageEmitted( tr( "The data provider for this layer does not support the addition of features." ), QgsMessageBar::WARNING );

return;

}

if (!vlayer->isEditable()){return;}

QList<QPoint> ptList;

ptList.append(QPoint(e->x() - 5, e->y() - 5));

ptList.append(QPoint(e->x() + 5, e->y() - 5));

ptList.append(QPoint(e->x() + 5, e->y() + 5));

ptList.append(QPoint(e->x() - 5, e->y() + 5));

const QgsMapToPixel \*transform = mCanvas->getCoordinateTransform();

QgsPolyline polyine;

foreach(QPoint point,ptList)

{

polyine.append(transform->toMapCoordinatesF(point.x(),point.y()));

}

QgsPolygon polygon;

polygon.append(polyine);

QgsFeature f;

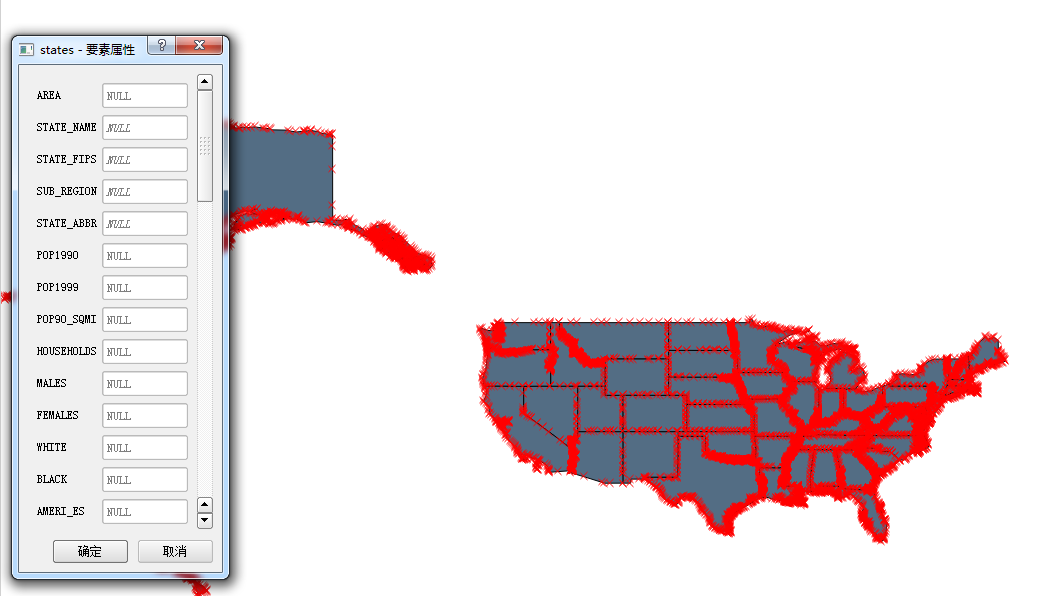
f.setFields(&vlayer->pendingFields(),true);

f.setGeometry(QgsGeometry::fromPolygon(polygon));

addFeature(vlayer,&f);

}

向画布中添加面图层，切换至编辑状态。点击向面图层中添加要素工具的运行结果，如图4-21所示：



编辑添加要素的属性

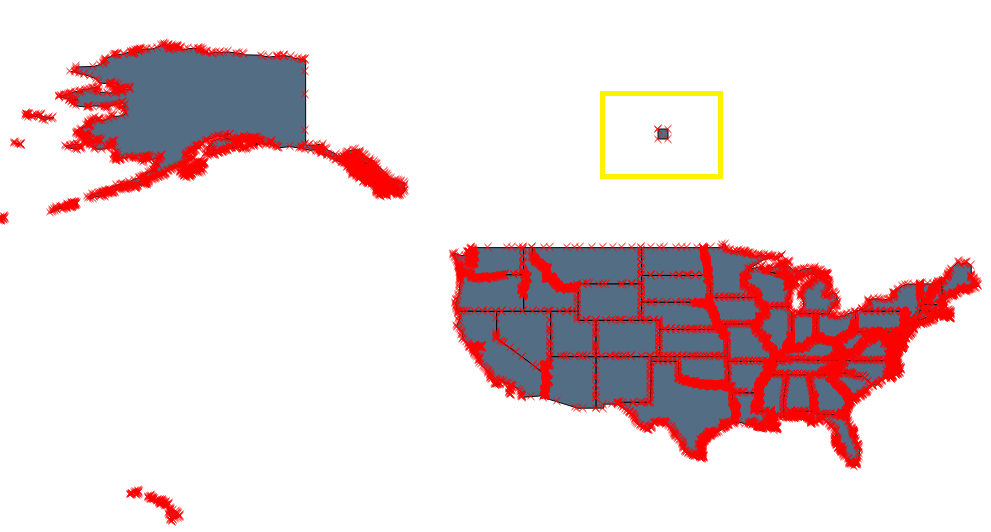


图4-21 向面图层添加要素的运行结果

##### 4.2.4.3.2 QgsMapToolEdit类鼠标事件响应

QgsMapToolEdit类是继承于QgsMapToolAdvancedDigitizing类的继承类，而QgsMapToolAdvancedDigitizing继承于QgsMapTool。QgsMapToolEdit类用于编辑地图工具。

1）QgsMapToolEdit类公有成员函数有：

virtual bool isEditTool() override； 重载是否编辑工具函数

2）QgsMapToolEdit类保护成员函数有：

QgsRubberBand\* createRubberBand( QGis::GeometryType geometryType = QGis::Line, bool alternativeBand = false )； 创建一个橡皮筋的颜色/线宽

QgsVectorLayer\* currentVectorLayer()； 获取当前矢量图层

int addTopologicalPoints( const QList<QgsPoint>& geom )；添加顶点到其他要素保持拓扑关系不变

void notifyNotVectorLayer()； 显示一个定时消息栏，当前激活图层不是矢量图层

void notifyNotEditableLayer()； 显示一个定时消息栏，当前矢量图层不能编辑

QgsMapToolEdit类实现实例：通过QgsMapToolEdit类实现响应鼠标事件移动图层要素。首先，在PluginDemo中添加一个类MouseMaptoolEdit来实现图层中移动要素的功能。对该类进行修改，使之继承于QgsMapToolEdit类。具体代码如下所示：

在mousemaptooledit.h中

包含头文件：

#include "qgsmaptool.h"

#include <qgsvectorlayer.h>

#include "qgisinterface.h"

#include "qgsvectordataprovider.h"

#include "qgsfeature.h"

#include "qgsgeometry.h"

#include "qgsmapcanvas.h"

#include "qgsmaptooledit.h"

#include "qgsrubberband.h"

#include "qgstolerance.h"

包含类：

class QgisInterface;

class QgsMapCanvas;

class QgsMapLayer;

class QMouseEvent;

class QRect;

class QPoint;

对类MouseMaptoolEdit进行修改：

class Mousemaptooledit : public QgsMapToolEdit

{

Q\_OBJECT

public:

Mousemaptooledit(QgsMapCanvas\* canvas);

~Mousemaptooledit();

virtual void canvasMoveEvent( QMouseEvent \* e ) override;

virtual void canvasPressEvent( QMouseEvent \* e ) override;

virtual void canvasReleaseEvent( QMouseEvent \* e ) override;

void deactivate() override;

private:

QgsPoint mStartPoint;

QgsRubberBand\* mRubberBand;

QgsFeatureIds mMovedFeatures;

};

在mousemaptooledit.cpp中

对构造函数进行初始化：

Mousemaptooledit::Mousemaptooledit(QgsMapCanvas\* canvas):QgsMapToolEdit(canvas) , mRubberBand( 0 ) {}

图层不激活时，释放指针。

void Mousemaptooledit::deactivate()

{

delete mRubberBand;

mRubberBand = 0;

QgsMapTool::deactivate();

}

响应鼠标移动事件，实现函数virtual void canvasMoveEvent( QMouseEvent \* e ) override。

void Mousemaptooledit::canvasMoveEvent( QMouseEvent \* e )

{

if ( mRubberBand )

{

QgsPoint point = toMapCoordinates( e->pos() ); //转换成地图坐标

double offsetX = point.x() - mStartPoint.x();

double offsetY = point.y() - mStartPoint.y();

mRubberBand->setTranslationOffset( offsetX, offsetY );

mRubberBand->updatePosition();

mRubberBand->update();

}

}

响应鼠标按下事件，实现函数virtual void canvasPressEvent( QMouseEvent \* e ) override。

void Mousemaptooledit::canvasPressEvent( QMouseEvent \* e )

{

delete mRubberBand;

mRubberBand = 0;

QgsVectorLayer\* vlayer = currentVectorLayer(); //获取当前图层

if ( !vlayer )

{

notifyNotVectorLayer();

return;

}

if ( !vlayer->isEditable() )

{

notifyNotEditableLayer();

return;

}

QgsPoint layerCoords = toLayerCoordinates( vlayer, e->pos() );

double searchRadius = QgsTolerance::vertexSearchRadius( mCanvas->currentLayer(), mCanvas->mapSettings() );

QgsRectangle selectRect( layerCoords.x() - searchRadius, layerCoords.y() - searchRadius, layerCoords.x() + searchRadius, layerCoords.y() + searchRadius );

if ( vlayer->selectedFeatureCount() == 0 )

{

QgsFeatureIterator fit = vlayer->getFeatures( QgsFeatureRequest().setFilterRect( selectRect ).setSubsetOfAttributes( QgsAttributeList() ) );

QgsGeometry\* pointGeometry = QgsGeometry::fromPoint( layerCoords );

if ( !pointGeometry ){return;}

double minDistance = std::numeric\_limits<double>::max();

QgsFeature cf;

QgsFeature f;

while ( fit.nextFeature( f ) )

{

if ( f.geometry() )

{

double currentDistance = pointGeometry->distance( \*f.geometry() );

if ( currentDistance < minDistance )

{

minDistance = currentDistance;

cf = f;

}

}

}

delete pointGeometry;

if ( minDistance == std::numeric\_limits<double>::max() ){return;}

mMovedFeatures.clear();

mMovedFeatures << cf.id();

mRubberBand = createRubberBand( vlayer->geometryType() );

mRubberBand->setToGeometry( cf.geometry(), vlayer );

}

else

{

mMovedFeatures = vlayer->selectedFeaturesIds();

mRubberBand = createRubberBand( vlayer->geometryType() );

QgsFeature feat;

QgsFeatureIterator it = vlayer->selectedFeaturesIterator( QgsFeatureRequest().setSubsetOfAttributes( QgsAttributeList() ) );

while ( it.nextFeature( feat ) )

{

mRubberBand->addGeometry( feat.geometry(), vlayer );

}

}

mStartPoint = toMapCoordinates( e->pos() );

mRubberBand->setColor( QColor( 255, 0, 0, 65 ) );

mRubberBand->setWidth( 2 );

mRubberBand->show();

}

响应鼠标释放事件，实现函数virtual void canvasReleaseEvent( QMouseEvent \* e ) override。

void Mousemaptooledit::canvasReleaseEvent( QMouseEvent \* e )

{

if ( !mRubberBand ){return;}

QgsVectorLayer\* vlayer = currentVectorLayer();

if ( !vlayer ){return;}

QgsPoint startPointLayerCoords = toLayerCoordinates(( QgsMapLayer\* )vlayer, mStartPoint );

QgsPoint stopPointLayerCoords = toLayerCoordinates(( QgsMapLayer\* )vlayer, e->pos() );

double dx = stopPointLayerCoords.x() - startPointLayerCoords.x();

double dy = stopPointLayerCoords.y() - startPointLayerCoords.y();

vlayer->beginEditCommand( tr( "移动要素" ) );

foreach ( QgsFeatureId id, mMovedFeatures )

{

vlayer->translateFeature( id, dx, dy );

}

delete mRubberBand;

mRubberBand = 0;

mCanvas->refresh();

vlayer->endEditCommand();

}

在图层中移动要素，切换至编辑状态。点击移动图层要素工具的运行结果，如图4-22所示：

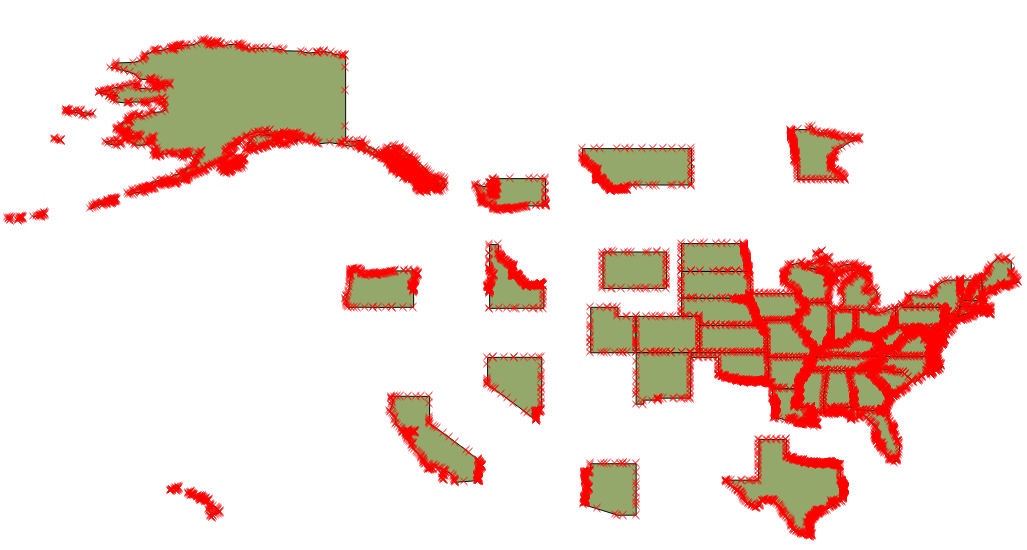


图4-22 移动图层要素的运行结果

##### 4.2.4.3.3 QgsMapToolCapture类鼠标事件响应

QgsMapToolCpture类是继承于QgsMapToolEdit类的继承类。QgsMapToolCapture类用于捕捉点，线，面。

1. QgsMapToolCpture类公有成员函数有：

QgsMapToolCapture( QgsMapCanvas\* canvas, CaptureMode mode = CaptureNone )；构造函数

virtual void canvasMapMoveEvent( QgsMapMouseEvent\* e ) override； 重载鼠标移动事件

virtual void canvasMapPressEvent( QgsMapMouseEvent \* e ) override； 重载鼠标按下事件

virtual void canvasKeyPressEvent( QKeyEvent\* e ) override； 重载键盘按下事件

virtual void deactivate() override；

1. QgsMapToolCpture类槽函数有：

void currentLayerChanged( QgsMapLayer \*layer )；

void addError( QgsGeometry::Error );

void validationFinished();

1. QgsMapToolCpture类保护成员函数有：

int nextPoint( const QgsPoint& mapPoint, QgsPoint& layerPoint );

int addVertex( const QgsPoint& point )； 添加一个点到橡皮条或捕捉列表中

void undo()； 从橡皮条和捕捉列表中移除最后一个顶点

void startCapturing()； 开始捕捉

bool isCapturing() const； 判断是否捕捉

void stopCapturing()； 停止捕捉

void deleteTempRubberBand()； 删除临时橡皮条

int size()； 获取捕捉列表的数量

const QList<QgsPoint> &points() { return mCaptureList; }

void setPoints( const QList<QgsPoint>& pointList ) { mCaptureList = pointList; }

void closePolygon();

QgsMapToolCapture类实现实例：通过QgsMapToolCapture类实现响应鼠标事件添加图层要素。首先，在PluginDemo中添加一个类QgsMapToolCapture来实现图层中添加要素的功能。对该类进行修改，使之继承于QgsMapToolCapture类。具体代码如下所示：

在mousemaptoolcapture.h中

包含头文件：

#include "qgsmaptool.h"

#include <qgsvectorlayer.h>

#include "qgisinterface.h"

#include "qgsvectordataprovider.h"

#include "qgsfeature.h"

#include "qgsgeometry.h"

#include "qgsmapcanvas.h"

#include "qgsmaptooledit.h"

#include "qgsmaptoolcapture.h"

#include "qgsmaplayerregistry.h"

#include "qgsproject.h"

#include "qgsfeatureaction.h"

包含类：

class QgisInterface;

class QgsMapCanvas;

class QgsMapLayer;

class QMouseEvent;

class QRect;

对类MouseMaptoolcapture进行修改，继承于QgsMapToolCapture

class mousemaptoolcapture : public QgsMapToolCapture

{

Q\_OBJECT

public:

mousemaptoolcapture(QgsMapCanvas\* canvas);

~mousemaptoolcapture();

void canvasMapReleaseEvent( QgsMapMouseEvent \* e ) override;

bool addFeature( QgsVectorLayer \*vlayer, QgsFeature \*f, bool showModal = true );

void activate() override;

private:

};

在mousemaptoolcapture.cpp中

对构造函数进行初始化：

mousemaptoolcapture::mousemaptoolcapture(QgsMapCanvas\* canvas): QgsMapToolCapture(canvas)

添加布尔型函数addFeature(QgsVectorLayer \*vlayer,QgsFeature \*f,bool showModal = true)的实现，判断是否成功添加要素。返回true时，添加成功，返回false时，添加失败。

bool mousemaptoolcapture::addFeature( QgsVectorLayer \*vlayer, QgsFeature \*f, bool showModal )

{

QgsFeatureAction \*action = new QgsFeatureAction( tr( "添加要素" ), \*f, vlayer, -1, -1, this );

bool res = action->addFeature( QgsAttributeMap(), showModal );

if ( showModal )

delete action;

return res;

}

设置工具图标的激活状态，实现函数void activate()。

void mousemaptoolcapture::activate()

{

QgsVectorLayer \*vlayer = qobject\_cast<QgsVectorLayer \*>( mCanvas->currentLayer() );

if ( vlayer && vlayer->geometryType() == QGis::NoGeometry )

{

QgsFeature f;

addFeature( vlayer, &f, false );

return;

}

QgsMapTool::activate();

}

响应鼠标释放事件，当鼠标释放时，向激活图层中添加要素。实现函数void canvasReleaseEvent (QMouseEvent \* e )。

void mousemaptoolcapture::canvasMapReleaseEvent( QgsMapMouseEvent\* e )

{

QgsDebugMsg( "entered." );

QgsVectorLayer \*vlayer = qobject\_cast<QgsVectorLayer \*>( mCanvas->currentLayer() ); // 获取当前图层

if ( !vlayer ) //判断如果不是矢量图层

{

notifyNotVectorLayer();

return;

}

QGis::WkbType layerWKBType = vlayer->wkbType(); //获取当前图层类型

QgsVectorDataProvider\* provider = vlayer->dataProvider(); //获取当前图层数据

if ( !( provider->capabilities() & QgsVectorDataProvider::AddFeatures ) )

{

emit messageEmitted( tr( "图层数据提供不支持要素添加" ), QgsMessageBar::WARNING );

return;

}

if ( !vlayer->isEditable() ) // 判断当前矢量图层是否在编辑状态

{

notifyNotEditableLayer();

return;

}

if ( mode() == CapturePoint ) // 判断如果当前捕捉的是点

{

if ( e->button() != Qt::LeftButton )

return;

if ( vlayer->geometryType() != QGis::Point ) //判断当前图层类型不是点图层

{

emit messageEmitted( tr( "错误的工具, 该矢量图层不能提供“捕捉点”工具" ), QgsMessageBar::WARNING );

return;

}

QgsPoint savePoint;

Try //对异常的处理

{

savePoint = toLayerCoordinates( vlayer, e->mapPoint() );

QgsDebugMsg( "savePoint = " + savePoint.toString() );

}

catch ( QgsCsException &cse ) //抛出异常

{

Q\_UNUSED( cse );

emit messageEmitted( tr( "不能转换点到图层坐标系" ), QgsMessageBar::WARNING );

return;

}

if ( provider->capabilities() & QgsVectorDataProvider::AddFeatures ) // 该图层支持要素添加

{

QgsFeature f( vlayer->pendingFields(), 0 ); //获取属性信息

QgsGeometry \*g = 0;

if ( layerWKBType == QGis::WKBPoint || layerWKBType == QGis::WKBPoint25D ) //图层类型为点图层

{

g = QgsGeometry::fromPoint( savePoint ); //点组成的几何体

}

else if ( layerWKBType == QGis::WKBMultiPoint || layerWKBType == QGis::WKBMultiPoint25D ) //图层类型为多点图层

{

g = QgsGeometry::fromMultiPoint( QgsMultiPoint() << savePoint );//多点组成几何体

}

f.setGeometry( g ); //几何信息添加到要素中

addFeature( vlayer, &f, false ); //向图层中添加要素

mCanvas->refresh();

}

}

else if ( mode() == CaptureLine || mode() == CapturePolygon ) //判断捕捉到的是线图层或者面图层

{

if ( mode() == CaptureLine && vlayer->geometryType() != QGis::Line )

{

emit messageEmitted( tr( "错误的编辑工具，该图层不能提供“捕捉线”工具" ), QgsMessageBar::WARNING );

return;

}

if ( mode() == CapturePolygon && vlayer->geometryType() != QGis::Polygon )

{

emit messageEmitted( tr( "错误的编辑工具，该图层不能提供“捕捉面”工具" ), QgsMessageBar::WARNING );

return;

}

if ( e->button() == Qt::LeftButton ) //鼠标左键按下

{

int error = addVertex( e->mapPoint() );

if ( error == 1 )

{

return;//当前图层不是矢量图层

}

else if ( error == 2 )

{

emit messageEmitted( tr( "不能转换点到图层坐标系" ), QgsMessageBar::WARNING );

return;

}

startCapturing(); //开始捕捉

}

else if ( e->button() == Qt::RightButton ) //鼠标右键按下

{

deleteTempRubberBand(); //删除临时橡皮条

if ( mode() == CaptureLine && size() < 2 )

{

stopCapturing();

return;

}

if ( mode() == CapturePolygon && size() < 3 )

{

stopCapturing();

return;

}

QgsFeature\* f = new QgsFeature( vlayer->pendingFields(), 0 );

QgsGeometry \*g;

if ( mode() == CaptureLine ) //捕捉线

{

if ( layerWKBType == QGis::WKBLineString || layerWKBType == QGis::WKBLineString25D ) 　　　　　　　　　　　 //当前矢量图层为线图层

{

g = QgsGeometry::fromPolyline( points().toVector() ); //线构成几何体

}

else if ( layerWKBType == QGis::WKBMultiLineString || layerWKBType == QGis::WKBMultiLineString25D )

{

g = QgsGeometry::fromMultiPolyline( QgsMultiPolyline() << points().toVector() );

//多线构成几何体

}

else

{

emit messageEmitted( tr( "不能添加要素，不明类型" ), QgsMessageBar::CRITICAL );

stopCapturing();

delete f;

return;

}

f->setGeometry( g );

}

else

{

if ( layerWKBType == QGis::WKBPolygon || layerWKBType == QGis::WKBPolygon25D ) //图层类型为面图层

{

g = QgsGeometry::fromPolygon( QgsPolygon() << points().toVector() );

//面构成几何体

}

else if ( layerWKBType == QGis::WKBMultiPolygon || layerWKBType == QGis::WKBMultiPolygon25D ) //图层类型为多面图层

{

g = QgsGeometry::fromMultiPolygon( QgsMultiPolygon() << ( QgsPolygon() << points().toVector() ) ); //多面构成几何体

}

else

{

emit messageEmitted( tr( "不能添加要素，不明类型" ), QgsMessageBar::CRITICAL );

stopCapturing();

delete f;

return;

}

if ( !g )

{

stopCapturing();

delete f;

return;

}

f->setGeometry( g );

}

addFeature(vlayer,f,false); //向图层中添加要素

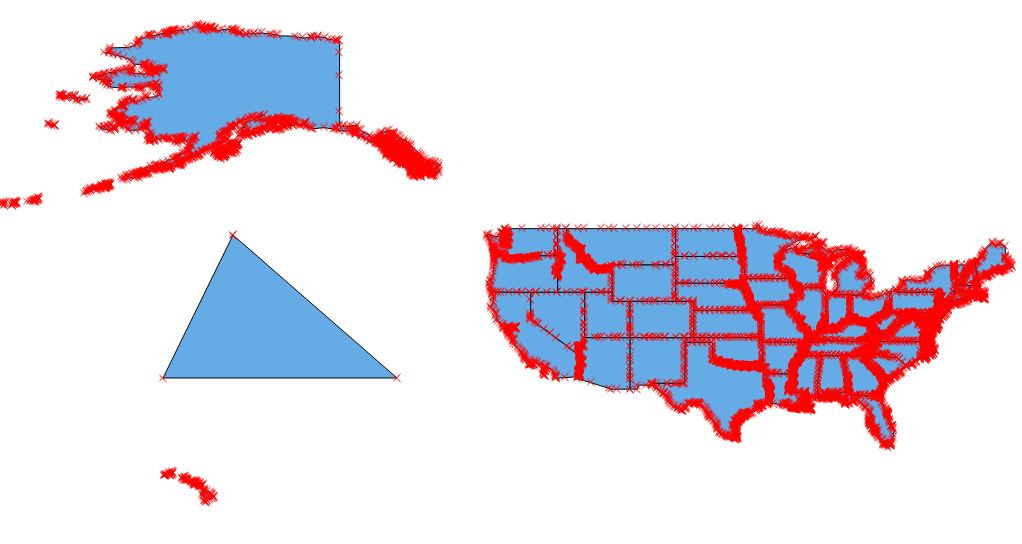
stopCapturing(); //停止编辑

}

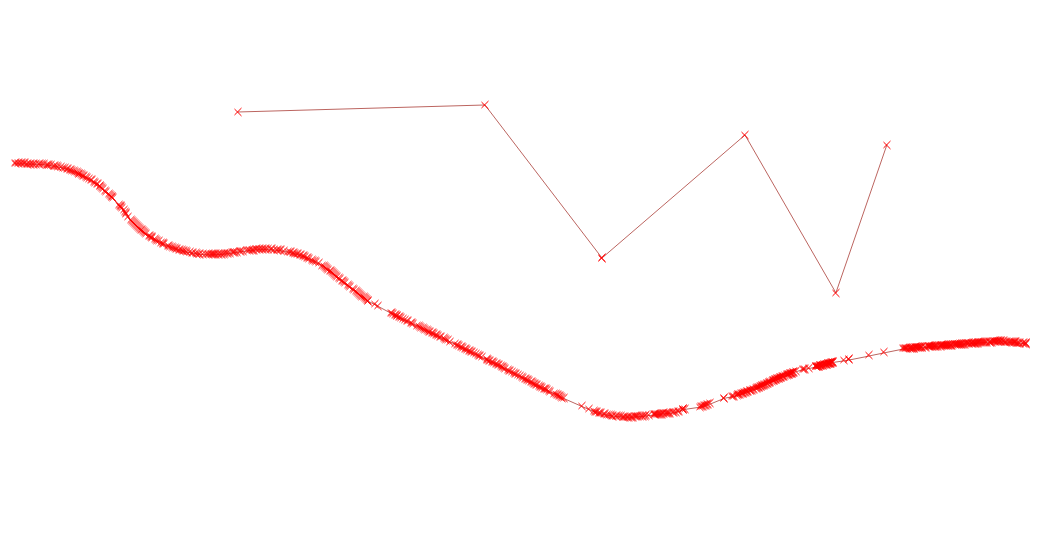
}

}

在图层中添加要素，切换至编辑状态。点击添加图层要素工具的运行结果，如图4-23所示：



（1）向面图层中添加面要素



（2）向线图层中添加线要素

图4-23 向图层添加要素运行结果