

QGIS 底层代码的分析

一、目录

目录

一、目录.....	1
二、作业内容、提纲及标准.....	1
三、安装部署 QGIS.....	1
四、Windows 程序 Main 函数及消息过程函数.....	3
五、GDI 绘图功能的封装（各类绘制函数及画笔画刷）.....	9
六、视图、窗口变换代码.....	13
七、点、线、面几何类.....	16
八、要素、符号、图层、地图类.....	23
九、空间分析功能：点与面的关系判断函数，面缓冲区算法.....	28
十、地图数据（geojson 格式）的存取.....	31

二、作业内容、提纲及标准

（1）作业内容

1、安装部署开源 GIS 平台 QGIS（原称 Quantum GIS），现在源代码，搭建 C++环境，实现 QGIS 平台的编译和运行。

<https://www.qgis.org/en/site/>

2、了解 QGIS 的源代码，根据实验报告提纲，在开源项目中寻找各项功能的关键核心代码，摘抄填入实验报告的对应章节。

（2）实验报告内容提纲

- 1、 Windows 程序 Main 函数及消息过程函数。
- 2、GDI 绘图功能的封装（各类绘制函数及画笔画刷）。
- 3、视图、窗口变换代码。
- 4、点、线、面几何类。
- 5、要素、符号、图层、地图类。
- 6、空间分析功能：点与面的关系判断函数，面缓冲区算法。
- 7、地图数据（geojson 格式）的存取。

（3）作业要求

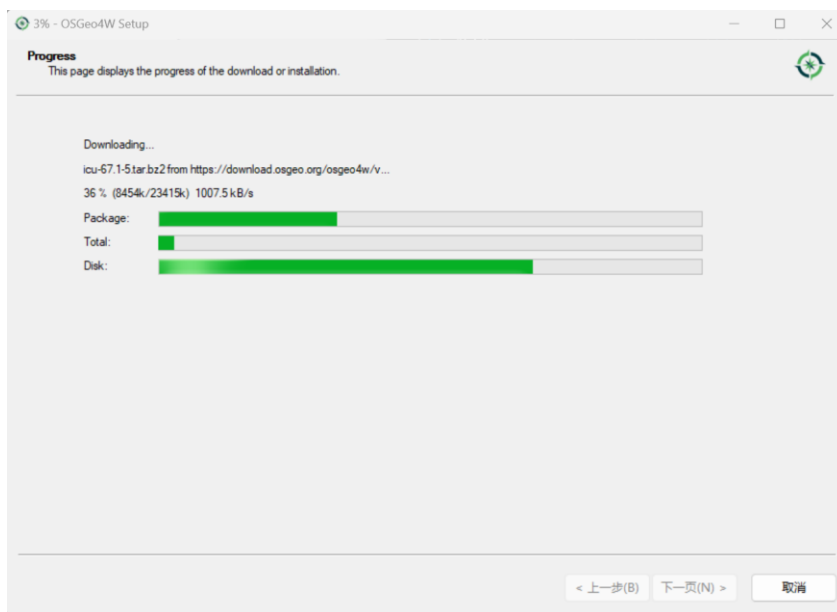
- 1、每人提交一份实验报告文档。
- 2、实验报告不少于 15 页，版式参照学校统一的实验报告。
- 3、内容需要涵盖报告提纲中的所有内容。
- 4、针对每一项内容，需要包括：原理介绍，关键代码（可截图），类名、方法名、代码所在文件和行数，与课程上编写的 C++代码的对应关系。

三、安装部署 QGIS

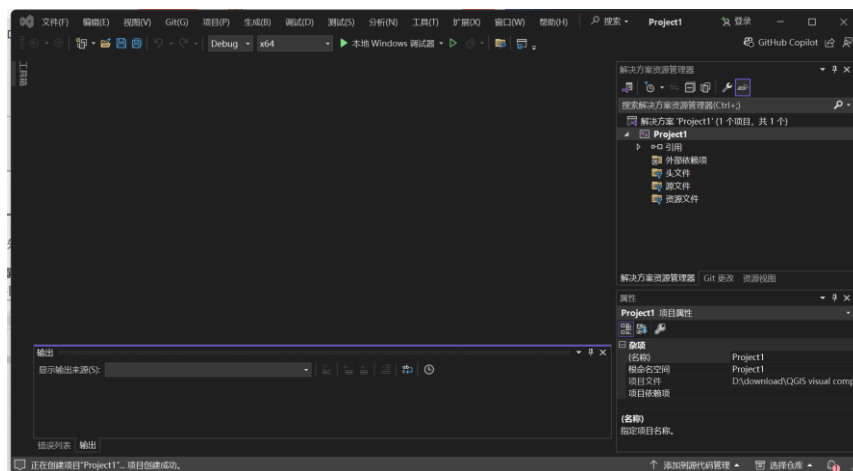
(1) 下载 OSGeo4W



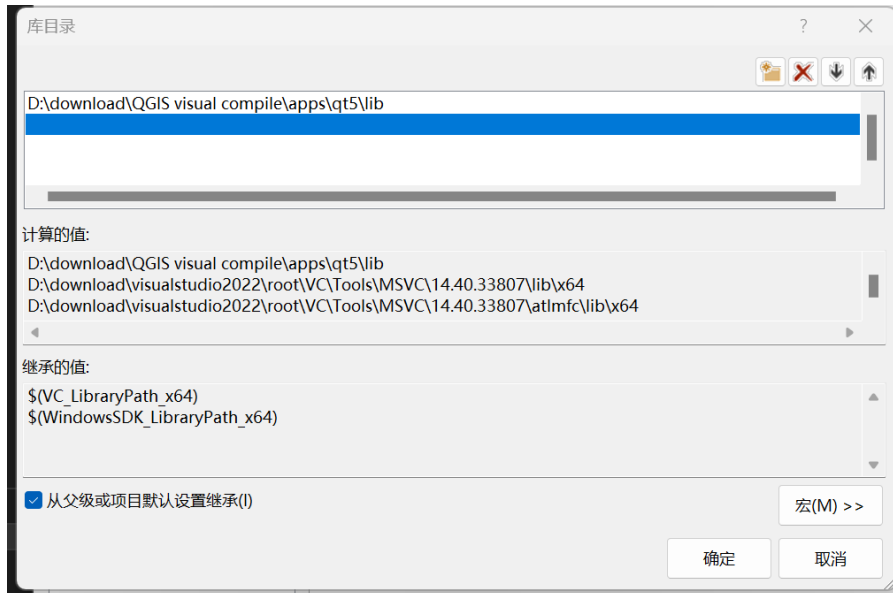
(2) 安装 OSGeo4W



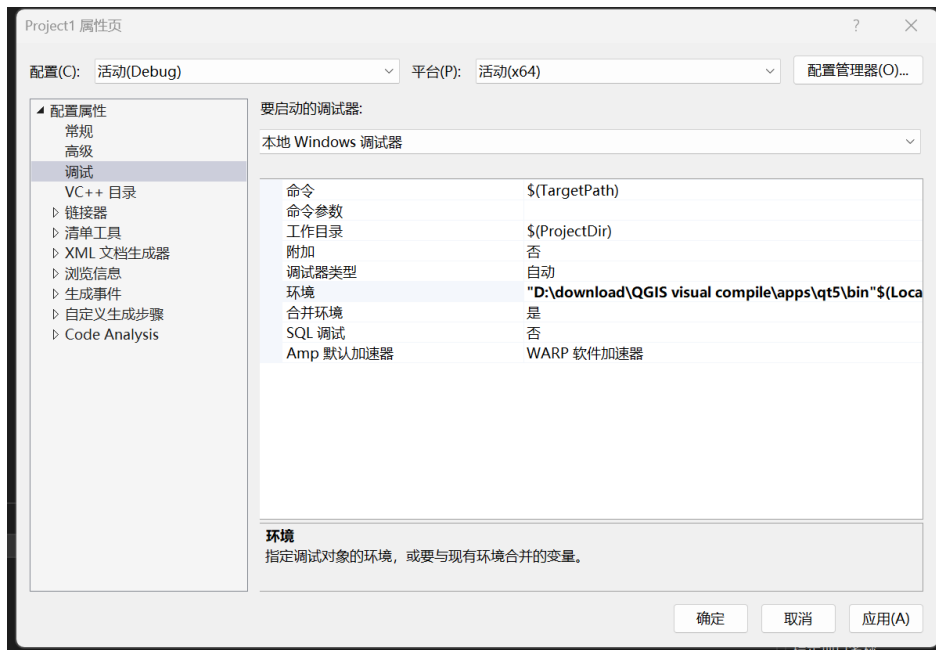
(3) 新开一个 vs 的空项目



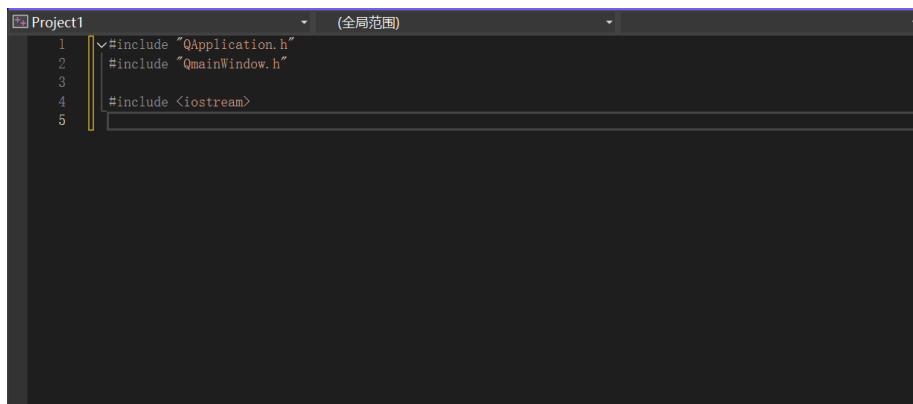
(4) 依次在 vs 环境中添加头文件目录、库目录、配置连接器



(5) 同时添加调试环境、定义预处理器



(6) 运行调试，引入 QGIS 相关的头文件，发现没有报错，说明已经成功搭建了编译环境



四、Windows 程序 Main 函数及消息过程函数

找到了几个相关的文档，分别有 main.cpp (.h)、qgisapp.cpp (.h)、mainwin.cpp,

各自的路径:

1.main.cpp 和 main.h

文件路径

1.QGIS-master\QGIS-master\src\app\main.cpp

2.QGIS-master\QGIS-master\src\providers\oracle\ocispatial\main.h

main 是程序的入口点, 包含 **Main** 函数, 负责初始化应用程序和主窗口, 并启动消息循环。主要负责整个应用程序的初始化和启动工作, 通常不涉及具体业务逻辑或界面设计, 主要是设置和启动整个应用程序的基本框架。发现这个 **main.h** 里面只有一个名为 **QOCISpatialDriverPlugin** 的类, 用于创建 **Oracle Spatial** 数据库的 **SQL** 驱动对象。这个类是数据库驱动插件, 并不是与 **Qt** 应用程序的 **main** 函数或消息循环直接相关的类。但是 **main.cpp** 中有比较丰富的信息, 找到了几个类:

QgsApplication 类: 这是 **QGIS** 应用程序的主要入口点, 用于初始化和应用程序的生命周期。它扩展了 **QCoreApplication** 类, 并添加了特定于 **QGIS** 的功能。

QgisApp 类: 这是 **QGIS** 主窗口的实现, 它创建了一个地图浏览器界面, 用户可以与之交互。**QgisApp** 类通常在 **QgsApplication** 初始化之后创建, 并在其中设置用户界面。

main 函数: 这是程序的入口点, 它创建了一个 **QgsApplication** 对象和一个 **QgisApp** 对象, 并启动 **Qt** 事件循环。这个函数处理命令行参数, 加载项目文件, 并根据需要执行其他初始化任务。

QApplication::exec(): 这是 **Qt** 应用程序的主事件循环, 它等待并处理用户界面的事件。在 **main** 函数中, 创建 **QgisApp** 对象后, 调用 **QgsApplication::exec()** 来启动事件循环。

QgsApplication::exit(): 用于退出应用程序的函数。在处理某些信号或在特定条件下, 可能会调用此函数来结束应用程序。

QgsApplication::processEvents(): 这个静态函数用于处理等待的事件。在需要强制更新用户界面或在特定时间点刷新应用程序状态时, 可能会使用此函数

QSplashScreen 类: 用于显示启动画面。在 **main** 函数中, 创建了一个 **QSplashScreen** 对象, 并在初始化 **QgisApp** 对象之前显示它。

QgsCustomization 类: 用于处理 **QGIS** 用户界面的定制。在 **main** 函数中, 加载了自定义设置, 并根据需要应用它们。

QgsSettings 类: 用于访问和修改 **QGIS** 的配置设置。在 **main** 函数中, 使用 **QgsSettings** 来获取和设置应用程序的偏好。

```
int main( int argc, char *argv[] )
{
    //log messages written before creating QgsApplication
    QStringList preApplicationLogMessages;
    QStringList preApplicationWarningMessages;

#ifdef Q_OS_UNIX
    // Increase file resource limits (i.e., number of allowed open files)
    // (from code provided by Larry Biehl, Purdue University, USA, from 'MultiSpec' project)
    // This is generally 256 for the soft limit on Mac
    // NOTE: setrlimit() must come *before* initialization of stdio strings,
    //       e.g. before any debug messages, or setrlimit() gets ignored
    // see: http://stackoverflow.com/a/17726104/2865523

```

Main 函数中, 启动了程序之后, 使用了一个登录验证的命令, 并且还有一个可以传出警告信息的命令, 之后用了一个预编译命令来判断本系统是否为 **UNIX** 系统, 这里变灰就说明本系统不是 **UNIX**, 所以被跳过了。

```

////////////////////////////////////
// Command line options 'behavior' flag setup
////////////////////////////////////

//
// Parse the command line arguments, looking to see if the user has asked for any
// special behaviors. Any remaining non command arguments will be kept aside to
// be passed as a list of layers and / or a project that should be loaded.
//

// This behavior is used to load the app, snapshot the map,
// save the image to disk and then exit
QString mySnapshotFileName;
QString configLocalStorageLocation;
QString profileName;
int mySnapshotWidth = 800;
int mySnapshotHeight = 600;

bool myHideSplash = false;
bool settingsMigrationForce = false;
bool mySkipVersionCheck = false;
bool hideBrowser = false;
#ifdef ANDROID
QgsDebugMsgLevel( QStringLiteral( "Android: Splash hidden" ), 2 );

```

从这里开始有了新的提示，这里的代码主要解析命令行参数，查看用户是否请求了任何特殊行为，然后对加载的图层列表进行处理，还有加载应用程序和对地图进行快照，随后保存和退出。

```

bool myRestoreDefaultWindowState = false;
bool myRestorePlugins = true;
bool mySkipBadLayers = false;
bool myCustomization = true;

QString dxfOutputFile;
Qgis::FeatureSymbologyExport dxfSymbologyMode = Qgis::FeatureSymbologyExport::PerSymbolLayer;
double dxfScale = 50000.0;
QString dxfEncoding = QStringLiteral( "CP1252" );
QString dxfMapTheme;
QgsRectangle dxfExtent;

bool takeScreenShots = false;
QString screenShotsPath;
int screenShotsCategories = 0;

// This behavior will set initial extent of map canvas, but only if
// there are no command line arguments. This gives a usable map
// extent when qgis starts with no layers loaded. When layers are
// loaded, we let the layers define the initial extent.
QString myInitialExtent;
if ( argc == 1 )
    myInitialExtent = QStringLiteral( "-1,-1,1,1" );

// This behavior will allow you to force the use of a translation file
// which is useful for testing
QString translationCode;

// The user can specify a path which will override the default path of custom
// user settings ( ~/.qgis ) and it will be used for QgsSettings INI file
QString authdbdirectory;

QString pythonfile;
QStringList pythonArgs;

QString customizationfile;
QString globalsettingsfile;

```

随后又紧跟了一些提示，这里主要是设置地图画布的初始范围，并且制定了一个自定义的默认路径，允许用户指定编译文件

```

QStringList args;

{
    QCoreApplication coreApp( argc, argv );
    ( void ) QgsApplication::resolvePkgPath(); // trigger storing of application path in QgsApplication

    if ( !bundleclicked( argc, argv ) )
    {
        // Build a local QCoreApplication from arguments. This way, arguments are correctly parsed from their native locale
        // It will use QString::fromLocal8Bit( argv ) under Unix and GetCommandLine() under Windows.
        args = QCoreApplication::arguments();

        for ( int i = 1; i < args.size(); ++i )
        {
            const QString &arg = args[i];

            if ( arg == QLatin1String( "--help" ) || arg == QLatin1String( "-?" ) )
            {
                usage( args[0] );
                return EXIT_SUCCESS;
            }
            else if ( arg == QLatin1String( "--version" ) || arg == QLatin1String( "-v" ) )
            {
                version();
                return EXIT_SUCCESS;
            }
            else if ( arg == QLatin1String( "--nologo" ) || arg == QLatin1String( "-n" ) )
            {
                myHideSplash = true;
            }
            else if ( arg == QLatin1String( "--version-migration" ) )
            {

```

根据提示和后面的代码来看，这段代码主要是根据用户的行为选择存储的方式

```

}

////////////////////////////////////
// If no --project was specified, parse the args to look for a      //
// .qgs file and set myProjectFileName to it. This allows loading    //
// of a project file by clicking on it in various desktop managers    //
// where an appropriate mime-type has been set up.                  //
////////////////////////////////////
if ( sProjectFileName.isEmpty() )
{
    // check for a .qgs/z
    for ( int i = 0; i < args.size(); i++ )
    {
        QString arg = QDir::toNativeSeparators( QFile::fromRawBuffer( args[i] ).absoluteFilePath() );
        if ( arg.endsWith( QLatin1String( ".qgs" ), Qt::CaseInsensitive ) ||
            arg.endsWith( QLatin1String( ".qgz" ), Qt::CaseInsensitive ) )
        {
            sProjectFileName = arg;
            break;
        }
    }
}

```

在 main 函数的最后，又给了几行提示，如果用户没有在命令行中指定 --project 参数，程序将遍历所有提供的参数，查找以 .qgs 或 .qgz 结尾的文件。如果找到这样的文件，程序将其路径赋值给 sProjectFileName 变量，这样 QGIS 就可以加载这个项目文件了。这是为了方便用户在桌面环境中双击项目文件图标来启动 QGIS 并加载项目

2.qgis.cpp 和 qgis.h

文件路径：

3.QGIS-master\QGIS-master\src\app\qgisapp.cpp

4.QGIS-master\QGIS-master\src\app\qgisapp.h

在 `qgisapp.h` 中，有超多的类和函数，一一列举如下：

1. `QgsMapCanvas`: 地图画布，用于显示地图和执行地图相关的绘制操作。
2. `QgsLayerTreeView`: 图层树视图，显示和管理地图中的图层。
3. `QgsAdvancedDigitizingDockWidget`: 高级数字化工具箱，提供 CAD 工具。
4. `QgsVertexEditor`: 顶点编辑器，用于编辑矢量图层的顶点。
5. `QgsTaskManagerStatusBarWidget`: 任务管理器状态栏控件，显示后台任务的进度。
6. `QgsStatusBarScaleWidget`: 状态栏比例尺控件。
7. `QgsStatusBarMagnifierWidget`: 状态栏放大镜控件。
8. `QgsStatusBarCoordinatesWidget`: 状态栏坐标控件。
9. `QgsDoubleSpinBox`: 用于输入旋转角度的双精度浮点数输入框。
10. `QgsMapTip`: 地图提示，显示鼠标悬停位置的地图信息。
11. `QgsInternalClipboard`: QGIS 内部剪贴板，用于存储矢量要素。
12. `QgsPythonUtils`: Python 工具，提供 Python 脚本运行和相关功能。
13. `QgsUndoWidget`: 撤销操作界面。
14. `QgsBrowserDockWidget`: 浏览器面板，提供数据源管理和浏览功能。
15. `QgsDataSourceManagerDialog`: 数据源管理对话框。
16. `QgsPluginManager`: 插件管理器，管理 QGIS 插件。
17. `QgsUserProfileManager`: 用户配置文件管理器。
18. `QgsVectorLayerTools`: 矢量图层工具，提供矢量图层相关操作。
19. `QgsMapCanvasTracer`: 地图追踪器，用于追踪地图要素。
20. `QgsSnappingUtils`: 捕捉工具，提供捕捉功能。
21. `QgsLocatorWidget`: 定位器小部件，提供地理定位和搜索功能。
22. `QgsMessageLogViewer`: 消息日志查看器，显示应用程序日志信息。

函数：

1. 构造函数和析构函数：初始化和销毁 QGIS 应用程序的主窗口。
2. `mapCanvas()`: 获取地图画布的指针。
3. `layerTreeView()`: 获取图层树视图的指针。
4. `addDockWidget()` 和 `removeDockWidget()`: 添加和移除停靠窗口。
5. `addToolBar()`: 添加工具栏。
6. `openProject()`: 打开 QGIS 项目文件。
7. `saveMapAsImage()`: 将当前视图保存为图像。
8. `exit()`: 退出应用程序。
9. `toggleEditing()`: 切换当前图层的编辑状态。
10. `saveEdits()` 和 `rollbackEdits()`: 保存和回滚图层编辑。
11. `cutSelectionToClipboard()` 和 `copySelectionToClipboard()`: 剪切和复制选中的要素到剪贴板。
12. `pasteFromClipboard()`: 从剪贴板粘贴要素到图层。
13. `copyStyle()` 和 `pasteStyle()`: 复制和粘贴图层样式。
14. `zoomIn()`, `zoomOut()`, `pan()`: 视图缩放和平移。
15. `identify()`: 识别地图上的要素。
16. `measure()`: 测量距离或面积。
17. `addFeature()`, `moveFeature()` 等: 编辑矢量要素，如添加、移动、删除等操作。
18. `showOptionsDialog()`: 显示选项对话框。
19. `checkUnsavedLayerEdits()`: 检查未保存的图层编辑。
20. `saveDirty()`: 保存脏项目。

21. triggerCrashHandler(): 触发崩溃处理程序（调试用）。
22. setActiveLayer(): 设置当前活动的图层。
23. fileNew(), fileOpen(), fileSave() 等：文件操作，如新建、打开、保存项目。
24. event(): 事件处理，用于拦截特定的系统事件。
25. keyPressEvent(): 键盘按键事件处理。
26. closeEvent(): 关闭窗口事件处理。
27. dragEnterEvent() 和 dropEvent(): 拖拽进入和释放事件处理。

随后看了一下 qgisapp.cpp，里面有两万多行代码，主要就是 qgisapp.h 的实现，这里面包含 QGIS 应用程序的主要逻辑，包括消息处理、主窗口的初始化、菜单和工具栏的设置等。这些文件是 QGIS 主程序的核心，负责实现 QGIS 的具体功能和用户界面的各种操作。它们与 main.cpp 通常有较紧密的联系，但功能更为具体和细化。

3.mainwin.cpp

5. QGIS-master\QGIS-master\src\app\mainwin.cpp

mainwin.cpp 是一个 Windows 应用程序的入口点文件，它包含了 Windows 特有的程序启动逻辑。

```
int CALLBACK WinMain( HINSTANCE /*hInstance*/, HINSTANCE /*hPrevInstance*/, LPSTR /*lpCmdLine*/, int /*nCmdShow*/ )
{
    std::string exename( moduleExeBaseName() );
    std::string basename( exename.substr( 0, exename.size() - 4 ) );

    if ( getenv( "OSGEO4W_ROOT" ) && __argc == 2 && strcmp( __argv[1], "--postinstall" ) == 0 )
    {
        std::string envfile( basename + ".env" );

        // write or update environment file
        if ( _access( envfile.c_str(), 0 ) < 0 || _access( envfile.c_str(), 2 ) == 0 )
        {
            std::list<std::string> vars;

            try
            {
                std::ifstream varfile;
                varfile.open( basename + ".vars" );

                std::string var;
                while ( std::getline( varfile, var ) )
                {
                    vars.push_back( var );
                }

                varfile.close();
            }
            catch ( std::ifstream::failure &e )
            {
                std::string message = "Could not read environment variable list " + basename + ".vars" + " [" + e.what() + "];";
                showError( message, "Error loading QGIS" );
            }
        }
    }
}
```

在这里找到了 WinMain 函数的入口，看样子主要是进行 window 系统特有的配置，有以下几个功能：
获取程序名称：从可执行文件路径中提取程序名（不包含扩展名）。

检查安装后模式：如果程序是以 --postinstall 参数运行，并且 OSGEO4W_ROOT 环境变量存在，说明可能正处于安装后的配置阶段。

环境变量处理：检查是否存在 .env 文件，若不存在或文件可写，则尝试创建或更新该文件读取 .vars 文件中的环境变量列表。对每个环境变量，检查系统是否已设置该变量，若已设置，则将其写入 .env 文件。

异常处理：在读取或写入文件过程中，如果发生错误，将通过 `showError` 函数显示错误信息，并终止程序。

退出：如果程序是作为安装后配置运行的，并且成功处理了环境变量文件，则正常退出。

总的来看，这段 `winmain` 代码主要是和 `main` 函数是相互联动和配合的关系，`main` 中主要是针对于普适性的配置，但是 `winmain` 主要是针对于 `window` 系统中独有的变量进行配置。

五、GDI 绘图功能的封装（各类绘制函数及画笔画刷）

研究了一下，QGIS 本身是基于 QT 框架进行编写的，其中的 GDI 绘画部分大多都被封装到了 QT 之中，因此，在 QGIS 的所有代码中并没有直接使用 GDI 绘图的底层代码。但是，这不代表没有各类绘制函数及画笔画刷。QT 框架使用 `QPainter` 绘图类进行操作。

在 "QGIS-master\QGIS-master\src\core\maprender\qgsmaprendercustompainterjob.cpp" 中，使用了大量地绘制函数及画笔画刷进行地图渲染，

```
void QgsMapRendererAbstractCustomPainterJob::preparePainter( QPainter *painter, const QColor &backgroundColor )
{
    // clear the background
    painter->fillRect( 0, 0, mSettings.deviceOutputSize().width(), mSettings.deviceOutputSize().height(), backgroundColor );

    painter->setRenderHint( QPainter::Antialiasing, mSettings.testFlag( Qgs::MapSettingsFlag::Antialiasing ) );
    painter->setRenderHint( QPainter::SmoothPixmapTransform, mSettings.testFlag( Qgs::MapSettingsFlag::HighQualityImageTransforms ) );
    painter->setRenderHint( QPainter::LosslessImageRendering, mSettings.testFlag( Qgs::MapSettingsFlag::LosslessImageRendering ) );

#ifdef QT_NO_DEBUG
    QPaintDevice *paintDevice = painter->device();
    const QString errMsg = QStringLiteral( "pre-set DPI not equal to painter's DPI (%1 vs %2)" )
        .arg( paintDevice->logicalDpiX() )
        .arg( mSettings.outputDpi() );
    Q_ASSERT_X( qgsDoubleNear( paintDevice->logicalDpiX(), mSettings.outputDpi(), 1.0 ),
        "Job::startRender()", errMsg.toLatin1().data() );
#endif
}
```

设置背景颜色：在 `preparePainter` 函数中，使用 `QPainter` 的 `fillRect` 方法填充背景色，并设置 `QPainter` 的渲染提示

```

if ( ! job.cached )
{
    QElapsedTimer layerTime;
    layerTime.start();

    if ( job.previewRenderImage && !job.previewRenderImageInitialized )
    {
        job.previewRenderImage->fill( 0 );
        job.previewRenderImageInitialized = true;
    }

    if ( job.img )
    {
        job.img->fill( 0 );
        job.imageInitialized = true;
    }

    job.completed = job.renderer->render();

    if ( job.picture )
    {
        job.renderer->renderContext()->painter()->end();
    }

    job.renderingTime += layerTime.elapsed();
}

if ( ! hasSecondPass && job.img )
{
    // If we flattened this layer for alternate blend modes, composite it now
    mPainter->setOpacity( job.opacity );
    mPainter->drawImage( 0, 0, *job.img );
    mPainter->setOpacity( 1.0 );
}

if ( mainElevationMap && job.context()->elevationMap() )
{
    const QgsElevationMap &layerElevationMap = *job.context()->elevationMap();
}

```

在 `doRender` 函数中，调用图层的 `render` 方法进行绘制，绘制图层图像到 `QPainter` 对象上，调用 `drawLabeling` 函数绘制标签，设置 `QPainter` 的透明度，如果有活动图层渲染，使用 `drawImage` 绘制组合后的图像。使用 `setCompositionMode` 设置合成模式，并绘制图像。绘制 `QPicture` 对象。

```

static void _fixQPictureDPI( QPainter *p )
{
    // QPicture makes an assumption that we drawing to it with system DPI.
    // Then when being drawn, it scales the painter. The following call
    // negates the effect. There is no way of setting QPicture's DPI.
    // See QTBUG-20361
    p->scale( static_cast< double >( qt_defaultDpiX() ) / p->device()->logicalDpiX(),
             static_cast< double >( qt_defaultDpiY() ) / p->device()->logicalDpiY() );
}

//
// QgsMapRendererAbstractCustomPainterJob
//

QgsMapRendererAbstractCustomPainterJob::QgsMapRendererAbstractCustomPainterJob( const QgsMapSettings &settings )
: QgsMapRendererJob( settings )
{
}

void QgsMapRendererAbstractCustomPainterJob::preparePainter( QPainter *painter, const QColor &backgroundColor )
{
    // clear the background
    painter->fillRect( 0, 0, mSettings.deviceOutputSize().width(), mSettings.deviceOutputSize().height(), backgroundColor );

    painter->setRenderHint( QPainter::Antialiasing, mSettings.testFlag( Qgs::MapSettingsFlag::Antialiasing ) );
    painter->setRenderHint( QPainter::SmoothPixmapTransform, mSettings.testFlag( Qgs::MapSettingsFlag::HighQualityImageTrans

```

这里也有 `QPicture` 的相关代码，主要是进行画笔的相关配置

在这个 `cpp` 对应的头文件里，定义了如下的函数和类：

类定义和继承：

`QgsMapRendererAbstractCustomPainterJob` 是一个抽象基类，为使用自定义画笔的地图渲染作业提供了基础。`QgsMapRendererCustomPainterJob` 类继承自上述抽象类，并实现了具体的渲染逻辑。

构造函数和析构函数: `QgsMapRendererCustomPainterJob::QgsMapRendererCustomPainterJob(const QgsMapSettings &settings, QPainter *painter)`: 构造函数, 接收地图设置和自定义 `QPainter` 对象。

`~QgsMapRendererCustomPainterJob() override`: 析构函数, 清理资源。

公共方法

`cancel()`, `cancelWithoutBlocking()`, `waitForFinished()`, `isActive()`, `usedCachedLabels()`, `takeLabelingResults()`: 这些方法用于控制和查询渲染作业的状态。`waitForFinishedWithEventLoop()`: 在等待渲染完成时保持事件循环运行, 这对于需要处理 GUI 事件的插件很有用。`renderSynchronously()`: 同步渲染地图, 阻塞调用线程直到渲染完成。`prepare()` 和 `renderPrepared()`: 用于准备和渲染预配置的渲染作业, 通常用于后台线程。

保护方法

`preparePainter(QPainter *painter, const QColor &backgroundColor = Qt::transparent)`: 准备 `QPainter` 对象, 设置背景颜色

私有槽和方法

`futureFinished()`: 当异步渲染完成时调用。`staticRender()` 和 `doRender()`: 在工作线程中调用的静态方法和实例方法, 用于执行实际的渲染工作

私有成员变量

`mPainter`: 指向自定义 `QPainter` 对象的指针。`mFuture` 和 `mFutureWatcher`: 用于异步渲染的 `QFuture` 和 `QFutureWatcher` 对象。`mLabelingEngineV2`: 用于标签渲染的引擎。

`mActive`, `mLayerJobs`, `mLabelJob`, `mSecondPassLayerJobs`: 表示渲染作业的状态和层级渲染作业的列表。`mRenderSynchronously`, `mPrepared`, `mPrepareOnly`: 控制渲染行为的标志。

宏和导出

`CORE_EXPORT`: 用于导出类符号, 确保在动态链接库中可见。

此外, 我们继续搜索, 在相同目录下的另外一个渲染文件"

`QGIS-master\QGIS-master\src\core\maprenderer\qgsmaprendererjob.cpp`", 也是显著地反映了 QGIS 的绘图封装情况:

```
~#include "qgsmaprendererjob.h"

#include <QPainter>
#include <QElapsedTimer>
#include <QTimer>
#include <QtConcurrentMap>

#include <QPicture>

#include "qgslogger.h"
#include "qgsrendercontext.h"
#include "qgsmaplayer.h"
#include "qgsmaplayerrenderer.h"
#include "qgsmaprenderercache.h"
#include "qgsrasterlayer.h"
#include "qgsmessagelog.h"
#include "qgsalllabeling.h"
#include "qgsexception.h"
#include "qgslabelingengine.h"
#include "qgsmaplayerlistutils_p.h"
#include "qgsvectorlayerlabeling.h"
#include "qgsexpressioncontextutils.h"
#include "qgsvectorlayerutils.h"
#include "qgsmaplayertemporalproperties.h"
#include "qgsmaplayerrelevationproperties.h"
#include "qgsmaplayerstyle.h"
```

在头文件引用中, 可以发现, 调用了 `QPainter` 作为主要的绘制外界库, 再次说明了 QGIS 并不使用直接的 windows 底层绘图函数进行操作, 而是借助 QT 框架下的绘图函数进行操作。

```

~QgsElevationMap *QgsMapRendererJob::allocateElevationMap( QString layerId )
{
    std::unique_ptr<QgsElevationMap> elevationMap = std::make_unique<QgsElevationMap>( mSettings.deviceOutputSize(), mSettings.devicePixelRatio() );
    ~
    if ( !elevationMap->isValid() )
    {
        mErrors.append( Error( layerId, tr( "Insufficient memory for elevation map %1x%2" ).arg( mSettings.outputSize().width() ).arg( mSettings.outputSize().height() ) ) );
        return nullptr;
    }
    return elevationMap.release();
}

~QPainter *QgsMapRendererJob::allocateImageAndPainter( QString layerId, QImage *&image, const QgsRenderContext *context )
{
    QPainter *painter = nullptr;
    image = allocateImage( layerId );
    ~
    if ( image )
    {
        painter = new QPainter( image );
        context->setPainterFlagsUsingContext( painter );
    }
    return painter;
}

~QgsMapRendererJob::PictureAndPainter QgsMapRendererJob::allocatePictureAndPainter( const QgsRenderContext *context )
{
    std::unique_ptr<QPicture> picture = std::make_unique<QPicture>();
    QPainter *painter = new QPainter( picture.get() );
    context->setPainterFlagsUsingContext( painter );
    return { std::move( picture ), painter };
}

~std::vector<LayerRenderJob> QgsMapRendererJob::prepareJobs( QPainter *painter, QgsLabelingEngine *labelingEngine2, bool deferredPainterSet )
{
    std::vector< LayerRenderJob > layerJobs;
}

```

继续往下，可以发现，里面有很多的函数涉及到了地图的绘制，其中被划分为了图片和非图片类，如果是图片类会规定一定的尺寸和大小，并在固定区域内绘制出图片全貌，如果是非地图类，会跳转到其它函数进行分类处理。

整理之后，会有不同的类和函数执行绘图功能：

QgsMapRendererJob：这是主要的地图渲染工作类，负责协调整个渲染过程。它创建了用于绘制的 QPainter 对象，管理图层绘制任务，以及处理标签绘制。

LayerRenderJob：这个类表示单个图层的渲染任务。它包含了图层的绘制上下文、图像、以及与绘制过程相关的各种参数。

QgsRenderContext：这个类存储了渲染过程中需要的上下文信息，比如坐标转换、显示范围、以及绘制设置等。

QgsMapLayer：代表地图上的一个图层，可以是矢量图层、栅格图层或其他类型。

QgsMapLayerRenderer：负责具体渲染图层内容的类。每个图层类型都有自己的渲染器实现。

QPainter：这是 Qt 框架中的一个类，用于绘制 2D 图形。在这段代码中，QPainter 被用来绘制图层内容到图像上。

QImage：表示一个图像，用于存储渲染结果。在渲染过程中，每个图层可能首先被渲染到自己的 QImage，然后合成到最终的地图图像中。

QgsElevationMap：用于处理和存储高程数据的类，它可以与栅格图层结合使用，为地图添加高程渲染效果。

QgsLabelingEngine：负责处理地图上的标签绘制。它根据图层的标签设置和地图的渲染上下文来渲染标签。

QgsMapRendererCache：用于缓存渲染结果，以提高渲染效率，避免重复渲染相同的内容。

QGIS-master > src > core > maprenderer

















在 maprenderer 中搜索

↑↓ 排序

≡ 查看

...

详细信息

名称	修改日期	类型	大小
 qgsmaprenderercache.cpp	2024/7/7 15:47	C++ Source	10 KB
 qgsmaprenderercache.h	2024/7/7 15:47	C/C++ Header	8 KB
 qgsmaprenderercustompainterjob.cpp	2024/7/7 15:47	C++ Source	15 KB
 qgsmaprenderercustompainterjob.h	2024/7/7 15:47	C/C++ Header	6 KB
 qgsmaprendererjob.cpp	2024/7/7 15:47	C++ Source	55 KB
 qgsmaprendererjob.h	2024/7/7 15:47	C/C++ Header	25 KB
 qgsmaprendererparalleljob.cpp	2024/7/7 15:47	C++ Source	13 KB
 qgsmaprendererparalleljob.h	2024/7/7 15:47	C/C++ Header	4 KB
 qgsmaprenderersequentialjob.cpp	2024/7/7 15:47	C++ Source	5 KB
 qgsmaprenderersequentialjob.h	2024/7/7 15:47	C/C++ Header	3 KB
 qgsmaprendererstagedrenderjob.cpp	2024/7/7 15:47	C++ Source	8 KB
 qgsmaprendererstagedrenderjob.h	2024/7/7 15:47	C/C++ Header	5 KB
 qgsmaprenderertask.cpp	2024/7/7 15:47	C++ Source	20 KB
 qgsmaprenderertask.h	2024/7/7 15:47	C/C++ Header	6 KB
 qgsrendereditemresults.cpp	2024/7/7 15:47	C++ Source	8 KB
 qgsrendereditemresults.h	2024/7/7 15:47	C/C++ Header	5 KB

经过综合研究调查发现，制图包装的类基本都位于这个文件夹下，这个文件夹的文件主要是用于做渲染工作的，其中的 `qgsmapcanvas.cpp` 用于显示画布，`QgsRasterLayer` 用于矢量图层和栅格图层的绘制和管理，`qgslayout` 用于布局元素，`qgsrenderer` 用于可视化表示地图要素，`qgsmaprenderers` 用于进行渲染顺序调整。

六、视图、窗口变换代码

视图找到的 `cpp` 文件和 `h` 文件路径为"
\\QGIS-master\\QGIS-master\\src\\core\\project\\qgsprojectviewsettings.cpp"和"
\\QGIS-master\\QGIS-master\\src\\core\\project\\qgsprojectviewsettings.h"

```

~ QVector<double> QgsProjectViewSettings::mapScales() const
{
    return mMapScales;
}

~ void QgsProjectViewSettings::setUseProjectScales( bool enabled )
{
    if ( enabled == useProjectScales() )
        return;

    mUseProjectScales = enabled;
    emit mapScalesChanged();
}

~ bool QgsProjectViewSettings::useProjectScales() const
{
    return mUseProjectScales;
}

~ double QgsProjectViewSettings::defaultRotation() const
{
    return mDefaultRotation;
}

~ void QgsProjectViewSettings::setDefaultRotation( double rotation )
{
    mDefaultRotation = rotation;
}

```

在 cpp 文件中，提供了很多对于视图的定义和变换操作，如图显示了其中对于视图的地图适应、旋转变换等变换操作。

```

private:

    QgsProject *mProject = nullptr;
    QVector<double> mMapScales;
    bool mUseProjectScales = false;
    QgsReferencedRectangle mDefaultViewExtent;
    QgsReferencedRectangle mPresetFullExtent;
    double mDefaultRotation = 0;
;

```

在 h 文件中，可以发现定义了很多私有变量来定义视图的属性，视图的变换基本就是围绕这些变量展开的，其中的变量有：

mProject：指向 QGIS 项目的指针。

mMapScales：存储项目地图比例尺的列表。

mUseProjectScales：一个布尔值，指示是否使用项目比例尺。

mDefaultViewExtent：项目的默认视图范围。

mPresetFullExtent：项目的预设完整视图范围。

mDefaultRotation：地图的默认旋转角度。

类和函数：

类：**QgsProjectViewSettings**：包含与 QGIS 项目在地图画布中显示相关的设置和属性，例如地图比例尺和默认视图范围

部分函数：**defaultViewExtent() const**：获取默认视图范围。**setDefaultViewExtent(const QgsReferencedRectangle &extent)**：设置默认视图范围。

presetFullExtent() const：获取项目的预设完整视图范围。**setPresetFullExtent(const QgsReferencedRectangle &extent)**：设置项目的预设完整视图范围。

fullExtent() const：计算并返回项目的完整视图范围。

setMapScales(const QVector<double> &scales): 设置项目地图的比例尺列表。**mapScales() const:** 获取项目地图的比例尺列表。

setUseProjectScales(bool enabled): 设置是否使用项目的比例尺。**useProjectScales() const:** 获取是否使用项目的比例尺。

defaultRotation() const: 获取默认地图旋转角度。

setDefaultRotation(double rotation): 设置默认地图旋转角度。

接下来，就是窗口变换的代码了，经过研究发现，这里和渲染的代码位置差不多，路径为"

\\QGIS-master\\src\\gui\\qgsmapcanvasutils.cpp"

因为在渲染的过程中就需要随时兼顾窗口变换的内容，因此窗口变换的主流代码都在地图渲染的代码中了。但是窗口变换实际有一个核心的属性保存源码处理的地方，在我们之前提到的 **mainwin.cpp** 文件中，里面有实时存储窗口属性和动态的地方，但是主要的变换操作都是渲染实现的。

```
void QgsMapCanvas::setWheelFactor( double factor )
{
    mWheelZoomFactor = std::max( factor, 1.01 );
}

void QgsMapCanvas::zoomIn()
{
    // magnification is already handled in zoomByFactor
    zoomByFactor( zoomInFactor() );
}

void QgsMapCanvas::zoomOut()
{
    // magnification is already handled in zoomByFactor
    zoomByFactor( zoomOutFactor() );
}

void QgsMapCanvas::zoomScale( double newScale, bool ignoreScaleLock )
{
    zoomByFactor( newScale / scale(), nullptr, ignoreScaleLock );
}

void QgsMapCanvas::zoomWithCenter( int x, int y, bool zoomIn )
{
    double scaleFactor = ( zoomIn ? zoomInFactor() : zoomOutFactor() );

    // transform the mouse pos to map coordinates
    QgsPointXY center = getCoordinateTransform()->toMapCoordinates( x, y );

    if ( mScaleLocked )
    {
        ScaleRestorer restorer( this );
        setMagnificationFactor( mapSettings().magnificationFactor() / scaleFactor, &center );
    }
}
```

在 **cpp** 文件中找到了关于窗口缩放的代码，这里的代码可以实现窗口的放大、缩小、大小调换、中心调换等等操作，总览整个 **cpp** 文件，有很多类似的视图操作，包括：

resizeEvent: 响应窗口大小变化事件，调整相应的视图大小

refresh: 刷新地图视图，涉及到重绘和窗口变换。

panAction: 处理整体平移操作的开始、移动和结束，并带动整体窗口信息变化。

moveCanvasContents: 移动地图内容以实现平移效果，并带动整体窗口信息变化。

setMagnificationFactor: 设置窗口的放大倍数，影响缩放级别。

magnificationFactor: 获取当前窗口的放大倍数。

还有很多，就不一一列举了。


```

void QgsMapCanvas::showContextMenu( QgsMapMouseEvent *event )
{
    const QgsPointXY mapPoint = event->originalMapPoint();

    QMenu menu;

    QMenu *copyCoordinateMenu = new QMenu( tr( "Copy Coordinate" ), &menu );
    copyCoordinateMenu->setIcon( QgsApplication::getThemeIcon( QStringLiteral( "/mActionEditCopy.svg" ) ) );

    auto addCoordinateFormat = [ &, this]( const QString identifier, const QgsCoordinateReferenceSystem & crs )
    {
        const QgsCoordinateTransform ct( mSettings.destinationCrs(), crs, mSettings.transformContext() );
        try
        {
            const QgsPointXY transformedPoint = ct.transform( mapPoint );

            // calculate precision based on visible map extent -- if user is zoomed in, we get better precision!
            int displayPrecision = 0;
            try
            {
                QgsCoordinateTransform extentTransform = ct;
                extentTransform.setBallparkTransformsAreAppropriate( true );
                QgsRectangle extentReproj = extentTransform.transformBoundingBox( extent() );
                const double mapUnitsPerPixel = ( extentReproj.width() / width() + extentReproj.height() / height() ) * 0.5;
                if ( mapUnitsPerPixel > 10 )
                    displayPrecision = 0;
                else if ( mapUnitsPerPixel > 1 )
                    displayPrecision = 1;
                else if ( mapUnitsPerPixel > 0.1 )
                    displayPrecision = 2;
                else if ( mapUnitsPerPixel > 0.01 )
                    displayPrecision = 3;
            }
            catch ( ... )
            {
                displayPrecision = 0;
            }
        }
        catch ( ... )
        {
            displayPrecision = 0;
        }
    };
}

```

这里有部分核心代码，涉及到依据地图大小弹出二级菜单的操作，里面就有很多窗口变换的内容，包括如何进行窗口的调整。

七、点、线、面几何类

这个比较好找，有一个文件夹统一存储了 QGIS 的各种地理元素类，即

QGIS-master\QGIS-master\src\core\geometry。

对于点的几何类而言，QGIS 中专门有一个对应的 cpp 和 h 文件存储相关源码，路径分别为"

QGIS-master\QGIS-master\src\core\geometry\qgspoint.cpp"和"

QGIS-master\QGIS-master\src\core\geometry\qgspoint.h"


```

QgsPoint::QgsPoint( double x, double y, double z, double m, Qgis::WkbType wkbType )
{
    : mX( x )
    , mY( y )
    , mZ( z )
    , mM( m )
    {
    if ( wkbType != Qgis::WkbType::Unknown )
    {
        Q_ASSERT( QgsWkbTypes::flatType( wkbType ) == Qgis::WkbType::Point );
        mWkbType = wkbType;
    }
    else if ( std::isnan( z ) )
    {
        if ( std::isnan( m ) )
            mWkbType = Qgis::WkbType::Point;
        else
            mWkbType = Qgis::WkbType::PointM;
    }
    else if ( std::isnan( m ) )
        mWkbType = Qgis::WkbType::PointZ;
    else
        mWkbType = Qgis::WkbType::PointZM;
    }

QgsPoint::QgsPoint( const QgsPointXY &p )
{
    : mX( p.x() )
    , mY( p.y() )
    , mZ( std::numeric_limits<double>::quiet_NaN() )
    , mM( std::numeric_limits<double>::quiet_NaN() )
    {
        mWkbType = Qgis::WkbType::Point;
    if ( p.isEmpty() )
    {
        mX = std::numeric_limits<double>::quiet_NaN();
        mY = std::numeric_limits<double>::quiet_NaN();
    }
}

```

发现 QGIS 中的地理元素基本都是用 WKB 格式存储的，对于一个点而言，有 x,y,z,m 四个分量，在这个文件中，给予了这个点类的各种不同操作，包括存储、显示等等。

```

QgsPoint *QgsPoint::snappedToGrid( double hSpacing, double vSpacing, double dSpacing, double mSpacing, bool ) const
{
    // helper function
    auto gridifyValue = []( double value, double spacing, bool extraCondition = true ) -> double
    {
        if ( spacing > 0 && extraCondition )
            return std::round( value / spacing ) * spacing;
        else
            return value;
    };

    // Get the new values
    const auto x = gridifyValue( mX, hSpacing );
    const auto y = gridifyValue( mY, vSpacing );
    const auto z = gridifyValue( mZ, dSpacing, QgisWkbTypes::hasZ( mWkbType ) );
    const auto m = gridifyValue( mM, mSpacing, QgisWkbTypes::hasM( mWkbType ) );

    // return the new object
    return new QgsPoint( mWkbType, x, y, z, m );
}

QgsPoint *QgsPoint::simplifyByDistance( double ) const
{
    return clone();
}

```

这里进行了点与格网的操作，包括附着格网等

```

        mM = mM * mScale + mTranslate;
    }

    ~bool QgsPoint::dropZValue()
    {
        if ( !is3D() )
            return false;

        mWkbType = QgsWkbTypes::dropZ( mWkbType );
        mZ = std::numeric_limits<double>::quiet_NaN();
        clearCache();
        return true;
    }

    ~bool QgsPoint::dropMValue()
    {
        if ( !isMeasure() )
            return false;

        mWkbType = QgsWkbTypes::dropM( mWkbType );
        mM = std::numeric_limits<double>::quiet_NaN();
        clearCache();
        return true;
    }

    ~void QgsPoint::swapXy()
    {
        std::swap( mX, mY );
        clearCache();
    }

    ~bool QgsPoint::convertTo( Qgis::WkbType type )

```

还有一大堆对于点的增删改查工作，这里就不一一列举了，都是一些基础功能。整理了一下，由于类中函数众多，只列举一些比较典型的：

QgsPoint 类：

构造函数：

QgsPoint(double x, double y, double z, double m, Qgis::WkbType wkbType): 创建一个具有给定坐标（x, y, z, m）和类型（wkbType）的点。

QgsPoint(const QgsPointXY &p): 根据 QgsPointXY 对象创建一个点，这是一个二维点坐标。

QgsPoint(QPointF p): 根据 QPointF 对象创建一个点，它是 Qt 框架中的二维点。

QgsPoint(Qgis::WkbType wkbType, double x, double y, double z, double m): 根据提供的参数和类型创建点。

成员函数：

clone(): 返回点的深拷贝。

snappedToGrid(): 返回一个新的点，其坐标已根据给定的网格间距调整。

simplifyByDistance(): 根据指定的距离简化点。对于 QgsPoint，它只是返回克隆。

removeDuplicateNodes(): 尝试移除重复的节点，但对单个点来说总是返回 false。

fromWkb(): 从 Well-Known Binary 格式解析点。

fromWkt(): 从 Well-Known Text 格式解析点。

wkbSize(): 返回点的 WKB 表示的大小。

asWkb(): 返回点的 WKB 格式的字节序列。

asWkt(): 返回点的 WKT 格式的字符串。

辅助函数:

isEmpty(): 检查点是否为空 (即坐标是否为 NaN)。

boundingBox3D(): 返回点的三维边界框。

geometryType(): 返回点的几何类型名称。

dimension(): 返回点的维度。

childCount(): 返回子点的数量。

后面, 继续在这个文件夹中寻找, 找到了描述线段的 cpp 文件和 h 文件, 路径分别为

QGIS-master\QGIS-master\src\core\geometry\qgslinesegment.cpp 和

QGIS-master\QGIS-master\src\core\geometry\qgslinesegment.h

```
#include "qgslinesegment.h"
#include "qgsgeometryutils.h"

int QgsLineSegment2D::pointLeftOfLine( const QgsPointXY &point ) const
{
    return QgsGeometryUtilsBase::leftOfLine( point.x(), point.y(), mStart.x(), mStart.y(), mEnd.x(), mEnd.y() );
}
```

Cpp 中的内容十分简单, 就只有几行代码, 表示一个二维空间中的线段, 有起点和终点相关的内容。

```
QgsLineSegment2D( double x1, double y1, double x2, double y2 ) SIP_HOLDGIL
mStart( QgsPointXY( x1, y1 ) )
, mEnd( QgsPointXY( x2, y2 ) )
{}

/**
 * Returns the length of the segment.
 * \see lengthSquared()
 */
double length() const SIP_HOLDGIL
{
    return mStart.distance( mEnd );
}

/**
 * Returns the squared length of the segment.
 * \see length()
 */
double lengthSquared() const SIP_HOLDGIL
{
    return mStart.sqrDist( mEnd );
}

/**
 * Returns the segment's start x-coordinate.
 * \see start()
 * \see startY()
 */
double startX() const SIP_HOLDGIL
{
    return mStart.x();
}
```

这是头文件中的代码, 有一些对于线段的普通操作, 包括求长度, 求长度平方之类的功能。线段的源代码之所以非常简单, 是因为线段本身就是由点连接而成的, 因此很多都只是点的拓展, 不需要写很多多余的内容。

这个头文件的函数和类主要有:

class QgsLineSegment2D: 定义了一个名为 QgsLineSegment2D 的类, 它是一个核心类 (由 CORE_EXPORT 指示)。

QgsLineSegment2D(const QgsPointXY &start, const QgsPointXY &end): 使用给定的起点和终点创建线段。

QgsLineSegment2D(double x1, double y1, double x2, double y2): 使用坐标创建线段, 起点为 (x1, y1), 终点为 (x2, y2)。

double length() const: 返回线段的长度。

double lengthSquared() const: 返回线段长度的平方, 这个函数通常用于避免平方根计算, 可以用于比较长度而不需要计算实际的长度值。

double startX() const、double startY() const、double endX() const、double endY() const: 分别返回起点和终点的 x 和 y 坐标。

QgsPointXY start() const、QgsPointXY end() const: 返回起点和终点的 QgsPointXY 对象。

void setStartX(double x)、void setStartY(double y)、void setEndX(double x)、void setEndY(double y): 分别设置起点和终点的 x 和 y 坐标。

void setStart(const QgsPointXY &start)、void setEnd(const QgsPointXY &end): 设置线段的起点和终点。

int pointLeftOfLine(const QgsPointXY &point) const: 判断一个点是否在直线的左侧, 返回 -1 表示点在左侧, +1 表示在右侧, 0 表示点在线上或测试不成功。

void reverse(): 反转线段, 交换起点和终点。

bool operator==(const QgsLineSegment2D &other) const 和 bool operator!=(const QgsLineSegment2D &other) const: 重载等号和不等号操作符, 用于比较两个线段是否相等或不相等。

值得一提的是, 线的种类不只有线段一种, 还有曲线以及其它类型的线条, 我在其中都找到了相关的源代码, 比如说曲线的源代码, 其 cpp 和 h 文件的路径有

"\\QGIS-master\\QGIS-master\\src\\core\\geometry\\qgscurve.cpp"和

"\\QGIS-master\\QGIS-master\\src\\core\\geometry\\qgscurve.h"

```
bool QgsCurve::operator==( const QgsAbstractGeometry &other ) const
{
    const QgsCurve *otherCurve = qgsgeometry_cast< const QgsCurve * >( &other );
    if ( !otherCurve )
        return false;

    return equals( *otherCurve );
}

bool QgsCurve::operator!=( const QgsAbstractGeometry &other ) const
{
    return !operator==( other );
}

bool QgsCurve::isClosed2D() const
{
    if ( numPoints() == 0 )
        return false;

    //don't consider M-coordinates when testing closedness
    const QgsPoint start = startPoint();
    const QgsPoint end = endPoint();

    return qgsDoubleNear( start.x(), end.x() ) &&
           qgsDoubleNear( start.y(), end.y() );
}

bool QgsCurve::isClosed() const
{
    bool closed = isClosed2D();
```

在 cpp 文件中, 有着非常多的对于曲线的计算操作, 这里显示了曲线的重载操作符以及起点和终点判定, 另外, 这段代码中还有分割曲线的函数、计算曲线三维边界框等功能, 下面列举了这个文件中的主要函数:

snapToGridPrivate: 将曲线的点吸附到指定的网格上, 可能移除冗余点。childCount 和 childPoint: 获取子点的数量和指定索引的子点。

straightDistance2d: 计算曲线起点到终点的直线距离。vertexCount, ringCount, partCount: 返回曲线的顶点数、环数和部件数。asQPainterPath: 将曲线转换为 QPainterPath 对象, 用于绘图。

isClosed2D 和 isClosed: 检查曲线是否在二维空间和三维空间中闭合。

还有很多, 这里就不一一列举了。

```
virtual int indexOf( const QgsPoint &point ) const = 0;

/**
 * Returns a reversed copy of the curve, where the direction of the curve has been flipped.
 */
virtual QgsCurve *reversed() const = 0 SIP_FACTORY;

QgsAbstractGeometry *boundary() const override SIP_FACTORY;

QString asKml( int precision = 17 ) const override;

/**
 * Returns a geometry without curves. Caller takes ownership
 * \param tolerance segmentation tolerance
 * \param toleranceType maximum segmentation angle or maximum difference between approximation and curve
 */
QgsCurve *segmentize( double tolerance = M_PI_2 / 90, SegmentationToleranceType toleranceType = MaximumAngle ) const override;

int vertexCount( int part = 0, int ring = 0 ) const override;
int ringCount( int part = 0 ) const override;
int partCount() const override;
QgsPoint vertexAt( QgsVertexId id ) const override;
QgsCurve *toCurveType() const override SIP_FACTORY;
void normalize() final SIP_HOLDGIL;

QgsBox3D boundingBox3D() const override;
bool isValid( QString &error SIP_OUT, Qgs::GeometryValidityFlags flags = Qgs::GeometryValidityFlags() ) const override;

/**
 * Returns the x-coordinate of the specified node in the line string.
 * \param index index of node, where the first node in the line is 0
 * \returns x-coordinate of node, or 0.0 if index is out of bounds
 */
virtual double xAt( int index ) const = 0;
```

这个是对应的头文件, 里面主要就是 cpp 文件中函数的声明, 这里就不赘述了。

最后, 就是 QGIS 的多边形类了, 可以选取最典型的多边形源代码文件, 其 cpp 文件和 h 文件路径分别为

"QGIS-master\QGIS-master\src\core\geometry\qgpspolygon.cpp"和"
QGIS-master\QGIS-master\src\core\geometry\qgpspolygon.h"

```

QgsAbstractGeometry *QgsPolygon::boundary() const
{
    if ( !mExteriorRing )
        return nullptr;

    if ( mInteriorRings.isEmpty() )
    {
        return mExteriorRing->clone();
    }
    else
    {
        QgsMultiLineString *multiLine = new QgsMultiLineString();
        int nInteriorRings = mInteriorRings.size();
        multiLine->reserve( nInteriorRings + 1 );
        multiLine->addGeometry( mExteriorRing->clone() );
        for ( int i = 0; i < nInteriorRings; ++i )
        {
            multiLine->addGeometry( mInteriorRings.at( i )->clone() );
        }
        return multiLine;
    }
}

```

在 cpp 文件中，都是对多边形的一系列计算和操作，这个函数比较典型，它主要是进行多边形边界的求解，其中有着一些数学的原理。

Cpp 有着如下的函数：

构造函数：

QgsPolygon()：默认构造函数，初始化 mWkbType 为 Polygon。

QgsPolygon(QgsLineString *exterior, const QList<QgsLineString *> &rings)：带参数的构造函数，允许设置外环和内环。

成员函数：

geometryType()：返回多边形的类型名称。

createEmptyWithSameType()：创建一个与当前多边形类型相同的空多边形。

clone()：克隆当前多边形对象。

clear()：清除多边形的所有环，并重置 mWkbType。

fromWkb()：从 WKB（Well-Known Binary）格式的地理空间数据中读取多边形。

wkbSize()：计算多边形的 WKB 表示所需的字节数。

asWkb()：将多边形转换为 WKB 格式。

asWkt()：将多边形转换为 WKT（Well-Known Text）格式。

addInteriorRing()：添加内环。

setExteriorRing()：设置外环。

boundary()：返回多边形的边界。

pointDistanceToBoundary()：计算点到多边形边界的最小距离。

surfaceToPolygon()：将表面转换为多边形（在这种情况下，直接返回克隆对象，因为 QgsPolygon 已经是多边形类型）。

toCurveType()：将多边形转换为曲线类型。

私有成员变量：

mWkbType：存储多边形的 WKB 类型。

mExteriorRing：存储多边形的外环。

mInteriorRings：存储多边形的内环列表。

```

class CORE_EXPORT QgsPolygon: public QgsCurvePolygon
{
public:

    /**
     * Constructor for an empty polygon geometry.
     */
    QgsPolygon() SIP_HOLDGIL;

    /**
     * Constructor for QgsPolygon, with the specified \a exterior ring and interior \a rings.
     *
     * Ownership of \a exterior and \a rings is transferred to the polygon.
     */
    /** \since QGIS 3.14 */
    QgsPolygon( QgsLineString *exterior SIP_TRANSFER, const QList< QgsLineString * > &rings SIP_TRANSFER = QList< QgsLineString * >() ) SIP_HOLDGIL;

    QString geometryType() const override SIP_HOLDGIL;
    QgsPolygon *clone() const override SIP_FACTORY;
    void clear() override;
    bool fromWkb( QgsConstWkbPtr &wkb ) override;
    int wkbSize( QgsAbstractGeometry::WkbFlags flags = QgsAbstractGeometry::WkbFlags() ) const override;
    QByteArray asWkb( QgsAbstractGeometry::WkbFlags flags = QgsAbstractGeometry::WkbFlags() ) const override;
    QString asWkt( int precision = 17 ) const override;
    QgsPolygon *surfaceToPolygon() const override SIP_FACTORY;

    /**
     * Returns the geometry converted to the more generic curve type QgsCurvePolygon
     * \returns the converted geometry. Caller takes ownership
     */
    QgsCurvePolygon *toCurveType() const override SIP_FACTORY;

    void addInteriorRing( QgsCurve *ring SIP_TRANSFER ) override;
};

```

H 文件中主要就是之前 cpp 文件的声明，但是声明的函数比 cpp 中要少一些，应该是有一些被其它文件实现了。

另外，除了这种多边形，还有多边形集合类、三角形类，实现的底层逻辑都和这个差不多，就不过多赘述了。

八、要素、符号、图层、地图类

要素类比较好找，QGIS 都统一存在以要素命名的 cpp 和 h 文件中，路径分别为"

QGIS-master\QGIS-master\src\core\pal\feature.cpp"和" QGIS-master\QGIS-master\src\core\pal\feature.h"

```

std::vector<std::unique_ptr<LabelPosition> > createCandidates( Pal *pal );

/**
 * Generate candidates for point feature, located around a specified point.
 * \param x x coordinate of the point
 * \param y y coordinate of the point
 * \param lPos pointer to an array of candidates, will be filled by generated candidates
 * \param angle orientation of the label
 * \returns the number of generated candidates
 */
std::size_t createCandidatesAroundPoint( double x, double y, std::vector<std::unique_ptr<LabelPosition> > &lPos, double angle );

/**
 * Generate one candidate over or offset the specified point.
 * \param x x coordinate of the point
 * \param y y coordinate of the point
 * \param lPos pointer to an array of candidates, will be filled by generated candidate
 * \param angle orientation of the label
 * \returns the number of generated candidates (always 1)
 */
std::size_t createCandidatesOverPoint( double x, double y, std::vector<std::unique_ptr<LabelPosition> > &lPos, double angle );

/**
 * Generate one candidate centered over the specified point.
 * \param x x coordinate of the point
 * \param y y coordinate of the point
 * \param lPos pointer to an array of candidates, will be filled by generated candidate
 * \param angle orientation of the label
 * \returns the number of generated candidates (always 1)
 */
std::size_t createCandidateCenteredOverPoint( double x, double y, std::vector<std::unique_ptr<LabelPosition> > &lPos, double angle );

```


因为这个相关的 `cpp` 比较复杂，先从 `h` 文件开始分析。`h` 中提供了主要的声明，其中提供了一些对于要素的操作，包括提取坐标点、内部集合类型判断之类的。根据说明可以发现这个要素类主要是依赖于 `libpal` 库进行自动化地图标签放置。

从上述截图的这段代码来看，`h` 文件中提供了大量的注释供读者阅读，这里主要是说明这里的函数适用于返回类型 `std::vector<std::unique_ptr<LabelPosition>>`，生成一系列位于指定点周围的标签位置候选项、在指定点上方或偏移位置生成一个标签位置候选项，在正上方生成一个标签位置候选项等等。

```
std::size_t FeaturePart::createCandidatesOutsidePolygon( std::vector<std::unique_ptr<LabelPosition>> &IPos, Pal *pal )
{
    // calculate distance between horizontal lines
    const std::size_t maxPolygonCandidates = mLF->layer()->maximumPolygonLabelCandidates();
    std::size_t candidatesCreated = 0;

    double labelWidth = getLabelWidth();
    double labelHeight = getLabelHeight();
    double distanceToLabel = getLabelDistance();
    const QgsMargins &visualMargin = mLF->visualMargin();

    /*
    * From Rylov & Reimer (2016) "A practical algorithm for the external annotation of area features":
    *
    * The list of rules adapted to the
    * needs of externally labelling areal features is as follows:
    * R1. Labels should be placed horizontally.
    * R2. Label should be placed entirely outside at some
    * distance from the area feature.
    * R3. Name should not cross the boundary of its area
    * feature.
    * R4. The name should be placed in way that takes into
    * account the shape of the feature by achieving a
    * balance between the feature and its name, emphasizing their relationship.
    * R5. The lettering to the right and slightly above the
    * symbol is prioritized.
    */
}
```

接下来再回来分析这个 `cpp` 文件，这个截图显示了其中要素类对于标签尺寸的处理，并且提供了大量的注释来解释标签规范。这个 `cpp` 文件提供了大量的实现，包括从 `GEOS` 几何体中提取坐标、生成标签候选位置、计算标签的尺寸和角度等。里面的类和函数主要有：构造函数和析构函数：

`FeaturePart::FeaturePart(QgsLabelFeature *feat, const GEOSGeometry *geom)`：构造函数，接受一个 `QgsLabelFeature` 指针和一个 `GEOS` 几何体指针。`FeaturePart::FeaturePart(const FeaturePart &other)`：复制构造函数。`FeaturePart::~~FeaturePart()`：析构函数，清理分配的资源。

成员函数：

`extractCoords(const GEOSGeometry *geom)`：从 `GEOS` 几何体中提取坐标点。

`Layer *layer()`：返回要素所属的图层。

`QgsFeatureId featureId() const`：返回要素的唯一 ID。

`std::size_t maximumPointCandidates() const` 等函数：返回生成标签候选位置的最大数量。

`bool hasSameLabelFeatureAs(FeaturePart *part) const`：判断两个 `FeaturePart` 是否属于同一个 `QgsLabelFeature`。

`std::size_t createCandidateCenteredOverPoint(double x, double y, std::vector<std::unique_ptr<LabelPosition>> &IPos, double angle)` 等函数：生成标签候选位置。

`std::unique_ptr<LabelPosition> createCandidatePointOnSurface(PointSet *mapShape)`：使用“点在表面上”算法创建单个候选位置。

`std::size_t createCandidatesAtOrderedPositionsOverPoint(double x, double y, std::vector<std::unique_ptr<LabelPosition>> &IPos, double angle)`：生成有序预定义位置的标签候选位置。

私有成员变量：

`QgsLabelFeature *mLF`：指向 `QgsLabelFeature` 的指针。

`QList<FeaturePart *> mHoles`：存储内环要素的列表。

接下来来找符号类，找到的 cpp 和 h 路径如下：

\\QGIS-master\\QGIS-master\\src\\core\\symbolology\\qgssymbol.cpp"和"

\\QGIS-master\\QGIS-master\\src\\core\\symbolology\\qgssymbol.h"和

```
/**
 * Constructor for a QgsSymbol of the specified \a type.
 *
 * Ownership of \a layers will be transferred to the symbol.
 */
QgsSymbol( Qgs::SymbolType type, const QgsSymbolLayerList &layers SIP_TRANSFER ); // can't be instantiated

/**
 * Creates a point in screen coordinates from a QgsPoint in map coordinates
 */
static inline QPointF _getPoint( QgsRenderContext &context, const QgsPoint &point )
{
    QPointF pt;
    if ( context.coordinateTransform().isValid() )
    {
        double x = point.x();
        double y = point.y();
        double z = 0.0;
        context.coordinateTransform().transformInPlace( x, y, z );
        pt = QPointF( x, y );
    }
    else
        pt = point.toQPointF();

    context.mapToPixel().transformInPlace( pt.rx(), pt.ry() );
    return pt;
}

/**
```

由于 cpp 文件中内容较多，先来看 h 文件中的内容，截图中的代码显示了符号类中对于点、线等不同的内容的不同操作。总的来说，这个文件中主要提供了渲染符号的方法，包括符号的创建、属性设置、动画支持、渲染过程管理等

```
void QgsSymbol::startFeatureRender( const QgsFeature &feature, QgsRenderContext &context, const int layer )
{
    if ( layer != -1 )
    {
        QgsSymbolLayer *symbolLayer = mLayers.value( layer );
        if ( symbolLayer && symbolLayer->enabled() )
        {
            symbolLayer->startFeatureRender( feature, context );
        }
        return;
    }
    else
    {
        const QList< QgsSymbolLayer * > layers = mLayers;
        for ( QgsSymbolLayer *symbolLayer : layers )
        {
            if ( !symbolLayer->enabled() )
                continue;

            symbolLayer->startFeatureRender( feature, context );
        }
    }
}

void QgsSymbol::stopFeatureRender( const QgsFeature &feature, QgsRenderContext &context, int layer )
{
    if ( layer != -1 )
    {
        QgsSymbolLayer *symbolLayer = mLayers.value( layer );
        if ( symbolLayer && symbolLayer->enabled() )
        {
            symbolLayer->stopFeatureRender( feature, context );
        }
    }
}
```

随后我们来分析 cpp 文件中的内容，cpp 的实现内容比较丰富。截图中显示的代码是用来让符号在开始渲染要素和停下渲染要素时进行的不同的操作，可以见到进行了一个图层的循环，每个图层都进行了操作。

总的来说，有如下的类和函数

类列表：

QgsSymbol - 抽象基类，用于渲染地理要素符号。

QgsSymbolAnimationSettings - 包含与符号动画相关的设置。

QgsSymbolLayer - 符号层的基类，用于构建符号。

QgsMarkerSymbol - 表示点要素的符号。

QgsLineSymbol - 表示线要素的符号。

QgsFillSymbol - 表示面要素的符号。

QgsGeometryGeneratorSymbolLayer - 用于生成符号的几何层。

QgsPaintEffect - 用于绘制效果的类。

QgsExpressionContext - 表达式上下文，用于符号渲染时的表达式计算。QgsRenderContext - 渲染上下文，包含渲染所需的信息。

主要的函数（由于太多，只列部分）

符号层管理：

QgsSymbol::symbolLayer(int layer) - 返回指定索引的符号层。QgsSymbol::symbolLayers() - 返回符号的所有层。QgsSymbol::symbolLayerCount() - 返回符号层的数量。QgsSymbol::insertSymbolLayer(int index, QgsSymbolLayer *layer) - 在指定索引插入符号层。

QgsSymbol::appendSymbolLayer(QgsSymbolLayer *layer) - 在符号层列表末尾追加符号层。

QgsSymbol::deleteSymbolLayer(int index) - 删除指定索引的符号层。

QgsSymbol::takeSymbolLayer(int index) - 移除并返回指定索引的符号层。

QgsSymbol::changeSymbolLayer(int index, QgsSymbolLayer *layer) - 更换指定索引的符号层。

颜色和透明度：

QgsSymbol::setColor(const QColor &color) - 设置符号的颜色。QgsSymbol::color() const - 返回符号的颜色。

QgsSymbol::setOpacity(qreal opacity) - 设置符号的不透明度。QgsSymbol::opacity() const - 返回符号的不透明度。

在图层方面，只找到了总的 cpp 文件，路径为" QGIS-master\QGIS-master\src\core\qgslayerdefinition.h"

```
case Qgis::LayerType::Plugin:
{
    const QString typeName = layerElem.attribute( QStringLiteral( "name" ) );
    layer = QgsApplication::pluginLayerRegistry()->createLayer( typeName );
    break;
}

case Qgis::LayerType::Mesh:
    layer = new QgsMeshLayer();
    break;

case Qgis::LayerType::VectorTile:
    layer = new QgsVectorTileLayer;
    break;

case Qgis::LayerType::PointCloud:
    layer = new QgsPointCloudLayer();
    break;

case Qgis::LayerType::TiledScene:
    layer = new QgsTiledSceneLayer;
    break;

case Qgis::LayerType::Group:
    layer = new QgsGroupLayer( QString(), QgsGroupLayer::LayerOptions( QgsCoordinateTransformContext() ) );
    break;

case Qgis::LayerType::Annotation:
    break;
}

if ( layer )
```

这段代码实现了图层的索引构建，根据不同的情况分别选取了网格、点云、群等方式组织了索引。其中的类和函数有：

QgsLayerDefinition - 主要类，提供加载和导出层定义的功能。QgsLayerDefinition::DependencySorter - 内部类，用于对层进行依赖排序。

loadLayerDefinition - 重载函数，用于从文件或 DOM 文档加载层定义。

exportLayerDefinition - 重载函数，用于导出选定的层树节点到 QLR 文件。
exportLayerDefinitionLayers - 导出多个层到一个新的 QDomDocument 对象
loadLayerDefinitionLayers - 加载 QLR 文件中的层定义，返回一个层列表。
loadLayerDefinitionLayersInternal - 内部函数，用于从 QDomDocument 中加载层定义并返回层列表
DependencySorter::init - 初始化依赖排序器，构建依赖图并排序层
DependencySorter::DependencySorter - 构造函数，初始化依赖排序器

最后，就是地图类了，找到 cpp 路径和 h 路径为 "QGIS-master\QGIS-master\src\core\qgsmapsettings.cpp" 和 "QGIS-master\QGIS-master\src\core\qgsmapsettings.h"

```
void setTextRenderFormat( Qgis::TextRenderFormat format )
{
    mTextRenderFormat = format;
    // ensure labeling engine setting is also kept in sync, just in case anyone accesses QgsMapSettings::labelingEngineSettings().defaultTextRenderFormat()
    // instead of correctly calling QgsMapSettings::textRenderFormat(). It can't hurt to be consistent!
    mLabelingEngineSettings.setDefaultTextRenderFormat( format );
}

//! sets format of internal QImage
void setOutputImageFormat( QImage::Format format ) { mImageFormat = format; }
//! format of internal QImage, default QImage::Format_ARGB32_Premultiplied
QImage::Format outputImageFormat() const { return mImageFormat; }

//! Check whether the map settings are valid and can be used for rendering
bool isValidSettings() const;
//! Returns the actual extent derived from requested extent that takes output image size into account
QgsRectangle visibleExtent() const;

/**
 * Returns the visible area as a polygon (may be rotated)
 */
QPolygonF visiblePolygon() const;
```

这里的代码是地图类头文件中负责检查的地方，会检查地图中的格式和内容是否合格，否则会返回错误。

```
if 1 // set visible extent taking rotation in consideration
{
    if ( mRotation )
    {
        const QgsPointXY p1 = mMapToPixel.toMapCoordinates( QPoint( 0, 0 ) );
        const QgsPointXY p2 = mMapToPixel.toMapCoordinates( QPoint( 0, myHeight ) );
        const QgsPointXY p3 = mMapToPixel.toMapCoordinates( QPoint( myWidth, 0 ) );
        const QgsPointXY p4 = mMapToPixel.toMapCoordinates( QPoint( myWidth, myHeight ) );
        dxmin = std::min( p1.x(), std::min( p2.x(), std::min( p3.x(), p4.x() ) ) );
        dymin = std::min( p1.y(), std::min( p2.y(), std::min( p3.y(), p4.y() ) ) );
        dxmax = std::max( p1.x(), std::max( p2.x(), std::max( p3.x(), p4.x() ) ) );
        dymax = std::max( p1.y(), std::max( p2.y(), std::max( p3.y(), p4.y() ) ) );
        mVisibleExtent.set( dxmin, dymin, dxmax, dymax );
    }
}
endif

QgsDebugMsgLevel( QStringLiteral( "Map units per pixel (x,y) : %1, %2" ).arg( qgsDoubleToString( mapUnitsPerPixelX ), qgsDoubleToString( mapUnitsPerPixelY ), 5 ) );
QgsDebugMsgLevel( QStringLiteral( "Pixmap dimensions (x,y) : %1, %2" ).arg( qgsDoubleToString( mSize.width(), qgsDoubleToString( mSize.height(), 5 ) );
QgsDebugMsgLevel( QStringLiteral( "Extent dimensions (x,y) : %1, %2" ).arg( qgsDoubleToString( mExtent.width(), qgsDoubleToString( mExtent.height(), 5 ) );
QgsDebugMsgLevel( mExtent.toString(), 5 );
QgsDebugMsgLevel( QStringLiteral( "Adjusted map units per pixel (x,y) : %1, %2" ).arg( qgsDoubleToString( mVisibleExtent.width() / myWidth ), qgsDoubleToString( mVi
QgsDebugMsgLevel( QStringLiteral( "Recalced pixmap dimensions (x,y) : %1, %2" ).arg( qgsDoubleToString( mVisibleExtent.width() / mMapUnitsPerPixel ), qgsDoubleToStr
QgsDebugMsgLevel( QStringLiteral( "Scale (assuming meters as map units) = 1:%1" ).arg( qgsDoubleToString( mScale ), 5 ) );
QgsDebugMsgLevel( QStringLiteral( "Rotation: %1 degrees" ).arg( mRotation ), 5 );
QgsDebugMsgLevel( QStringLiteral( "Extent: %1" ).arg( mExtent.asWktCoordinates(), 5 );
QgsDebugMsgLevel( QStringLiteral( "Visible Extent: %1" ).arg( mVisibleExtent.asWktCoordinates(), 5 );
QgsDebugMsgLevel( QStringLiteral( "Magnification factor: %1" ).arg( mMagnificationFactor ), 5 );
```

这里的代码是地图类 cpp 文件中的一个核心位置，其它的代码比较少，但是这个比较突出，主要是在计算地图考虑旋转之后的可见范围并进行调试信息输出。

其中的类和函数：

类：QgsMapSettings 管理地图渲染的配置，包括图层、显示范围、比例尺、DPI、旋转角度等。

成员变量：

mDpi：默认 DPI 设置。

mSize：输出图像的尺寸。

mBackgroundColor 和 mSelectionColor：地图的背景色和选中时的特征颜色。

mFlags：渲染标志，控制渲染行为。

mExtent 和 mExtentBuffer：地图显示范围和缓冲区。

mRotation 和 mMagnificationFactor：地图的旋转角度和放大因子。

成员函数（部分内容）：

构造函数：初始化地图设置并计算派生值。

setMagnificationFactor(): 设置地图的放大因子，并调整 DPI 和范围。

extent() 和 setExtent(): 获取和设置地图的范围。

rotation() 和 setRotation(): 获取和设置地图的旋转角度。

updateDerived(): 更新基于当前设置的派生值，如可见范围和比例尺。outputSize() 和 setOutputSize(): 获取和设置输出图像的尺寸。QgsScaleCalculator: 用于计算比例尺。

QgsMapToPixel: 用于将地图坐标转换为像素坐标。

QgsCoordinateTransform: 用于执行坐标参考系统之间的坐标转换。QgsCoordinateReferenceSystem: 表示坐标参考系统。

九、空间分析功能：点与面的关系判断函数，面缓冲区算法

QGIS 将点与面的关系判断函数统一纳入地理要素间关系之中了，找到的源文件的 cpp 和 h 是"

QGIS-master\QGIS-master\src\core\geometry\qgsgeometry.cpp"和"

QGIS-master\QGIS-master\src\core\geometry\qgsgeometry.h"

```
bool QgsGeometry::intersects( const QgsRectangle &r ) const
{
    // fast case, check bounding boxes
    if ( !boundingBoxIntersects( r ) )
        return false;

    const Qgis::WkbType flatType { QgisWkbTypes::flatType( d->geometry->wkbType() ) };
    // optimise trivial case for point intersections -- the bounding box test has already given us the answer
    if ( flatType == Qgis::WkbType::Point )
    {
        return true;
    }

    // Workaround for issue GH #51429
    // in case of multi polygon, intersection with an empty rect fails
    if ( flatType == Qgis::WkbType::MultiPolygon && r.isEmpty() )
    {
        const QgsPointXY center { r.xMinimum(), r.yMinimum() };
        return contains( QgsGeometry::fromPointXY( center ) );
    }

    QgsGeometry g = fromRect( r );
    return intersects( g );
}
```

```

✓bool QgsGeometry::intersects( const QgsGeometry &geometry ) const
{
✓  if ( !d->geometry || geometry.isNull() )
  {
    return false;
  }

  QgsGeos geos( d->geometry.get() );
  mLastError.clear();
  return geos.intersects( geometry.d->geometry.get(), &mLastError );
}

✓bool QgsGeometry::boundingBoxIntersects( const QgsRectangle &rectangle ) const
{
✓  if ( !d->geometry )
  {
    return false;
  }

  return d->geometry->boundingBoxIntersects( rectangle );
}

✓bool QgsGeometry::boundingBoxIntersects( const QgsGeometry &geometry ) const
{
✓  if ( !d->geometry || geometry.isNull() )
  {
    return false;
  }

  return d->geometry->boundingBoxIntersects( geometry.constGet()->boundingBox() );
}

```

```

✓bool QgsGeometry::contains( const QgsPointXY *p ) const
{
✓  if ( !d->geometry || !p )
  {
    return false;
  }

  QgsGeos geos( d->geometry.get() );
  mLastError.clear();
  return geos.contains( p->x(), p->y(), &mLastError );
}

✓bool QgsGeometry::contains( double x, double y ) const
{
✓  if ( !d->geometry )
  {
    return false;
  }

  QgsGeos geos( d->geometry.get() );
  mLastError.clear();
  return geos.contains( x, y, &mLastError );
}

```

```

bool QgsGeometry::contains( const QgsGeometry &geometry ) const
{
    if ( !d->geometry || geometry.isNull() )
    {
        return false;
    }

    QgsGeos geos( d->geometry.get() );
    mLastError.clear();
    return geos.contains( geometry.d->geometry.get(), &mLastError );
}

```

在 cpp 的一千五百多行这样找到了判断点与面关系的函数实现，它定义了几种不同的方法来判断几何对象之间的关系，如相交、包含、不相交等，我这里找的函数实际上包含了不同地理元素之间的元素，不只是点与面。

`QgsGeometry::intersects(const QgsRectangle &r) const`, 这个方法检查 `QgsGeometry` 对象是否与一个矩形区域 `QgsRectangle` 相交

`QgsGeometry::intersects(const QgsGeometry &geometry) const` 这个方法检查两个 `QgsGeometry` 对象是否相交

`QgsGeometry::boundingBoxIntersects(const QgsRectangle &rectangle) const` 检查当前几何对象的边界框是否与给定的矩形相交

`QgsGeometry::boundingBoxIntersects(const QgsGeometry &geometry) const`

检查当前几何对象的边界框是否与另一个几何对象的边界框相交

`QgsGeometry::contains(const QgsPointXY *) const` 和 `QgsGeometry::contains(double x, double y) const` 这两个方法检查几何对象是否包含一个点（通过 `QgsPointXY` 结构或坐标 `x` 和 `y`）

`QgsGeometry::contains(const QgsGeometry &geometry) const` 检查当前几何对象是否包含另一个几何对象

`QgsGeometry::disjoint(const QgsGeometry &geometry) const` 检查当前几何对象是否与另一个几何对象不相交

在 QGIS 的分析模块找到了 QGIS 统一进行缓冲区分析的文件，其 cpp 和 h 分别为"

QGIS-master\QGIS-master\src\analysis\processing\qgsalgorithmbuffer.cpp"和"

QGIS-master\QGIS-master\src\analysis\processing\qgsalgorithmbuffer.h"

```

void QgsBufferAlgorithm::initAlgorithm( const QVariantMap & )
{
    addParameter( new QgsProcessingParameterFeatureSource( QStringLiteral( "INPUT" ), QObject::tr( "Input layer" ) ) );

    auto bufferParam = std::make_unique< QgsProcessingParameterDistance >( QStringLiteral( "DISTANCE" ), QObject::tr( "Distance" ), 10, QStringLiteral( "INPUT" ) );
    bufferParam->setDynamic( true );
    bufferParam->setDynamicPropertyDefinition( QgsPropertyDefinition( QStringLiteral( "Distance" ), QObject::tr( "Buffer distance" ), QgsPropertyDefinition::Double ) );
    bufferParam->setDynamicLayerParameterName( QStringLiteral( "INPUT" ) );
    addParameter( bufferParam.release() );

    auto segmentParam = std::make_unique< QgsProcessingParameterNumber >( QStringLiteral( "SEGMENTS" ), QObject::tr( "Segments" ), Qgs::ProcessingNumberParameterType::Integer, 5, false, 1 );
    segmentParam->setHelp( QObject::tr( "The segments parameter controls the number of line segments to use to approximate a quarter circle when creating rounded offsets." ) );
    addParameter( segmentParam.release() );

    addParameter( new QgsProcessingParameterEnum( QStringLiteral( "END_CAP_STYLE" ), QObject::tr( "End cap style" ), QStringList() << QObject::tr( "Round" ) << QObject::tr( "Flat" ) << QObject::tr( "Square" ), false, 0 ) );
    addParameter( new QgsProcessingParameterEnum( QStringLiteral( "JOIN_STYLE" ), QObject::tr( "Join style" ), QStringList() << QObject::tr( "Round" ) << QObject::tr( "Miter" ) << QObject::tr( "Bevel" ), false, 0 ) );
    addParameter( new QgsProcessingParameterNumber( QStringLiteral( "MITER_LIMIT" ), QObject::tr( "Miter limit" ), Qgs::ProcessingNumberParameterType::Double, 2, false, 1 ) );

    addParameter( new QgsProcessingParameterBoolean( QStringLiteral( "DISSOLVE" ), QObject::tr( "Dissolve result" ), false ) );

    auto keepDisjointParam = std::make_unique< QgsProcessingParameterBoolean >( QStringLiteral( "SEPARATE_DISJOINT" ), QObject::tr( "Keep disjoint results separate" ), false );
    keepDisjointParam->setFlags( keepDisjointParam->flags() | Qgs::ProcessingParameterFlag::Advanced );
    keepDisjointParam->setHelp( QObject::tr( "If checked, then any disjoint parts in the buffer results will be output as separate single-part features." ) );
    addParameter( keepDisjointParam.release() );

    addParameter( new QgsProcessingParameterFeatureSink( QStringLiteral( "OUTPUT" ), QObject::tr( "Buffered" ), Qgs::ProcessingSourceType::VectorPolygon, QVariant(), false, true, true ) );
}

```

这段代码实现了缓冲区的计算操作，首先，它添加了一个输入要素源参数。接着，定义了一个可动态变化的缓冲距离参数，并允许用户设置距离属性。此外，还添加了控制圆弧近似线段数的参数，以及端点样式和连接样式的枚举参数，后者包括圆滑、平直和斜接选项。还提供了一个斜接限制的数值参数，仅适用于斜接连接样式。算法还包括一个布尔参数，用于选择是否溶解缓冲结果，以及一个高级选项，用于决定是否将不相连的部分作为单独要素输出。最后，定义了输出参数，指定输出为多边形要素汇。

总的来说，这段代码通过计算，得到了一个地理单元的缓冲区。

十、地图数据（geojson 格式）的存取

在 QGIS 的核心文件夹中找到了 QGIS 地图数据存取的源代码其 cpp 和 h 路径如下："D:\通用文件夹\GIS 工程\QGIS-master\QGIS-master\src\core\qgsjsonutils.cpp"和"D:\通用文件夹\GIS 工程\QGIS-master\QGIS-master\src\core\qgsjsonutils.h"

需要说明的是，QGIS 中并没有直接进行 geojson 转换的代码，主要是使用了 gdal 外界库帮助其进行 geojson 的转换，QGIS 中主要还是 json 的处理，然后再做 geojson 的二次加工。

```
std::unique_ptr< QgsAbstractGeometry > parseGeometryFromGeoJson( const json &geometry )
{
    if ( !geometry.is_object() )
    {
        QgsDebugError( QStringLiteral( "JSON geometry value must be an object" ) );
        return nullptr;
    }

    if ( !geometry.contains( "type" ) )
    {
        QgsDebugError( QStringLiteral( "JSON geometry must contain 'type'" ) );
        return nullptr;
    }

    const QString type = QString::fromStdString( geometry["type"].get< std::string >() );
    if ( type.compare( QLatin1String( "Point" ), Qt::CaseInsensitive ) == 0 )
    {
        if ( !geometry.contains( "coordinates" ) )
        {
            QgsDebugError( QStringLiteral( "JSON Point geometry must contain 'coordinates'" ) );
            return nullptr;
        }

        const json &coords = geometry["coordinates"];
        return parsePointFromGeoJson( coords );
    }
    else if ( type.compare( QLatin1String( "MultiPoint" ), Qt::CaseInsensitive ) == 0 )
    {
        if ( !geometry.contains( "coordinates" ) )
        {
            QgsDebugError( QStringLiteral( "JSON MultiPoint geometry must contain 'coordinates'" ) );
            return nullptr;
        }
    }
}
```

这里的 cpp 代码直接提供了 geojson 类型的判断和转换，根据不同的地理元素类型选取各自的存取方式。


```

/**
 * Converts JSON \a jsonString to a QVariant, in case of parsing error an invalid QVariant is returned
 * and the \a error argument is populated accordingly.
 *
 * \note Not available in Python bindings
 * \since QGIS 3.24
 */
static QVariant parseJson( const std::string &jsonString, QString &error ) SIP_SKIP;

/**
 * Converts JSON \a jsonString to a QVariant, in case of parsing error an invalid QVariant is returned.
 * \note Not available in Python bindings
 * \since QGIS 3.8
 */
static QVariant parseJson( const QString &jsonString ) SIP_SKIP;

/**
 * Converts a JSON \a value to a QVariant, in case of parsing error an invalid QVariant is returned.
 * \note Not available in Python bindings
 * \since QGIS 3.36
 */
static QVariant jsonToVariant( const json &value ) SIP_SKIP;

/**
 * Add \a crs information entry in \a json object regarding old GeoJSON specification format
 * if it differs from OGC:CRS84 or EPSG:4326.
 * According to new specification RFC 7946, coordinate reference system for all GeoJSON coordinates
 * is assumed to be OGC:CRS84 but when user specifically request a different CRS, this method
 * adds this information in the JSON output
 */
static void addCrsInfo( json &value, const QgsCoordinateReferenceSystem &crs ) SIP_SKIP;

```

在声明中，可以看到提供了几个函数进行 json 和 geojson 的转化和存取。

总的来说，相关的类和函数有：

QgsJsonExporter 类：

构造函数：初始化精度和向量图层，并设置坐标参考系统（CRS）。setVectorLayer 和 vectorLayer 方

法：设置和获取向量图层。

setSourceCrs 和 sourceCrs 方法：设置和获取源 CRS。

exportFeature 和 exportFeatureToJsonObject 方法：将单个地理要素（Feature）导出为 JSON 对象或字符串。

exportFeatures 和 exportFeaturesToJsonObject 方法：将多个地理要素导出为 JSON 对象或字符串。

setDestinationCrs 方法：设置目标 CRS。

QgsJsonUtils 类：

提供了一系列静态方法，用于将字符串转换为地理要素列表、字段列表，以及将 QVariant 转换为 JSON 格式的字符串。

parseArray 方法：解析 JSON 数组并将其转换为 QVariantList。

encodeValue 和 exportAttributes 方法：用于将属性值编码为 JSON 格式。geometryFromGeoJson 方法：将 GeoJSON 格式的几何对象转换为 QgsGeometry 对象。jsonFromVariant 和 jsonToVariant 方法：在 QVariant 和 JSON 之间进行转换。

parseJson 方法：解析 JSON 字符串并将其转换为 QVariant。

exportAttributesToJsonObject 方法：将属性导出为 JSON 对象。

addCrsInfo 方法：向 JSON 对象添加 CRS 信息。