WBoard

WBoard 是一款在 Windows 平台基于 VS+Qt 开发的一款开放源码的白板教学软件，主要用于学校和大学的交互式电子白板。 它既可以与交互式白板一起使用，也可以在双屏幕场景中通过笔、平板显示器和光束进行使用。 主要有演示板、网页、文档和桌面四大界面。

编译环境

编译器安装

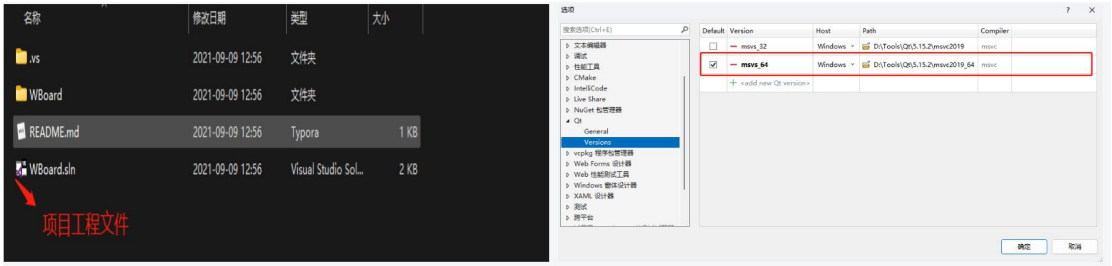
1.安装 Visual Studio: VS2017 及以上　Qt: Qt5

2.Visual Studio 官网下载地址：
https://visualstudio.microsoft.com/zh-hans/downloads/
3.Qt5.15.2 下载地址
https://download.qt.io/official_releases/online_installers/
4.Visual Studio 配置 Qt 教程博客

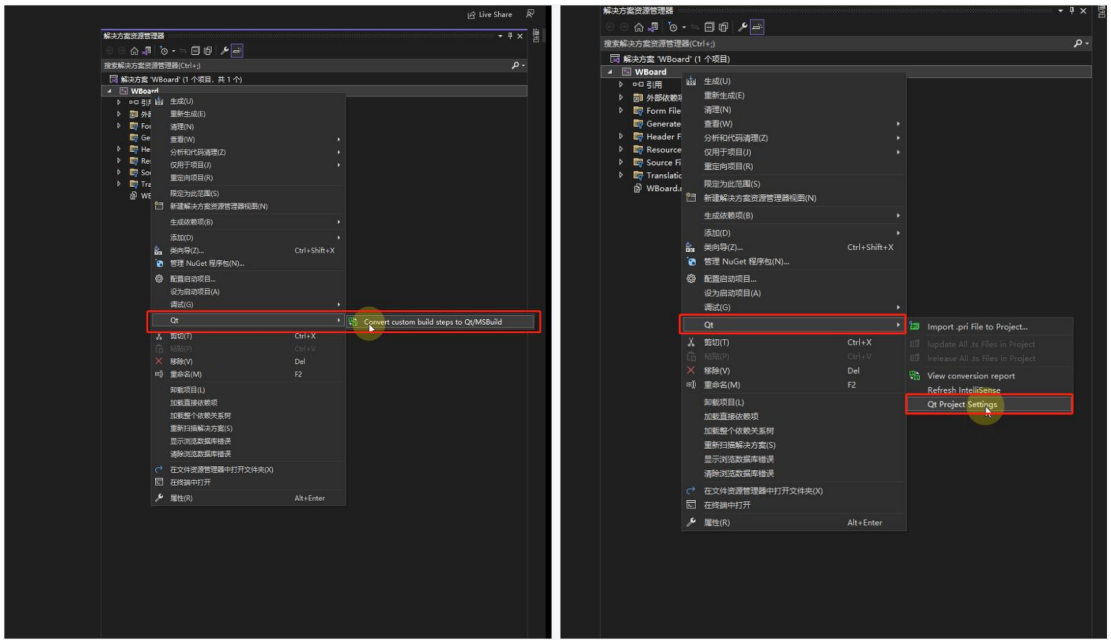
[vs2019 + Qt5.15.2 开发环境搭建_vsaddin-msvc2019-2.7.0.vsix 下载-CSDN 博客](https://blog.csdn.net/u014552102/article/details/118346113)

配置项目

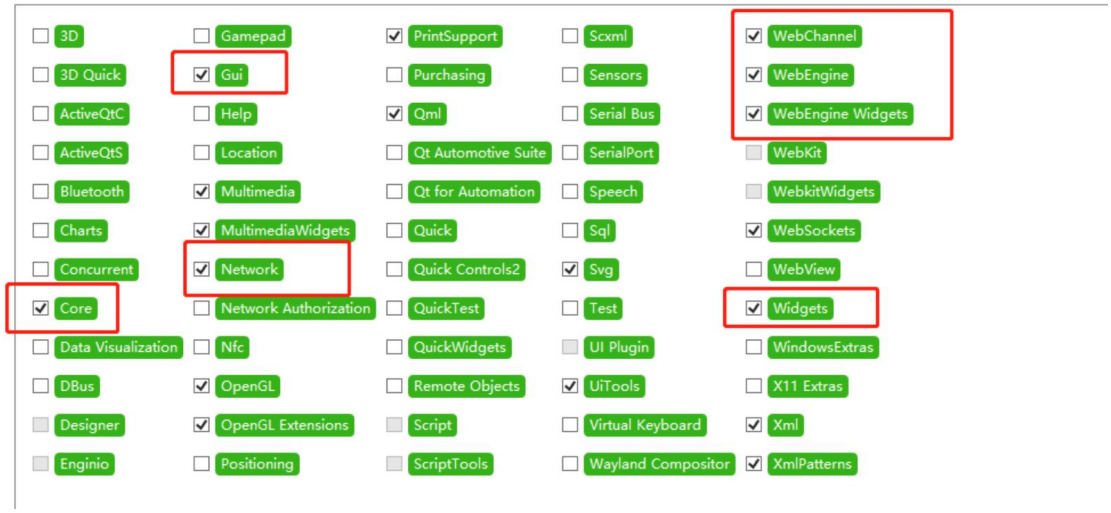
1.解压项目并打开项目工程文件,并配置 Qt 编译环境



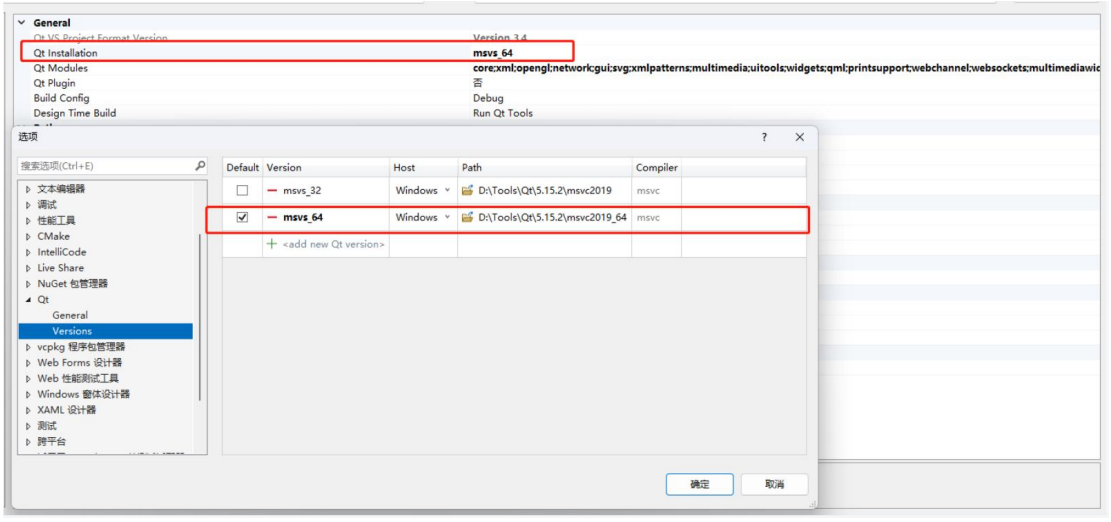
2.选择 Qt 构建项目并选择 Qt 项目设置



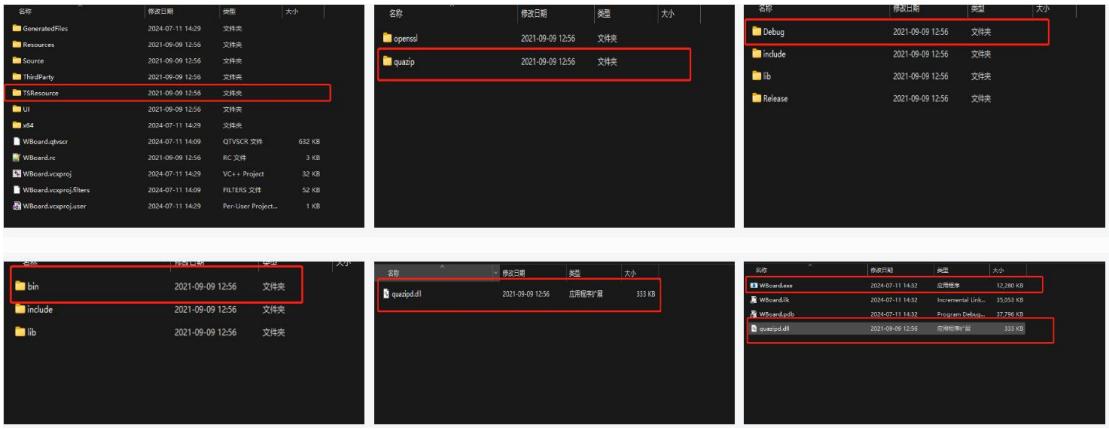
1. 查看 Qt 模块，是否含有以下模块，如没有去 Qt 安装目录找到 MaintenanceTool.exe 安装缺失模块并安装，详细安装步骤参考文献：[解决“Project ERROR: Unknown module(s) in QT: webengine”以及“Your MaintenanceTool appears to be older than 3.0.2. .” 的办法 - The_Reborn - 博客园 (cnblogs.com)



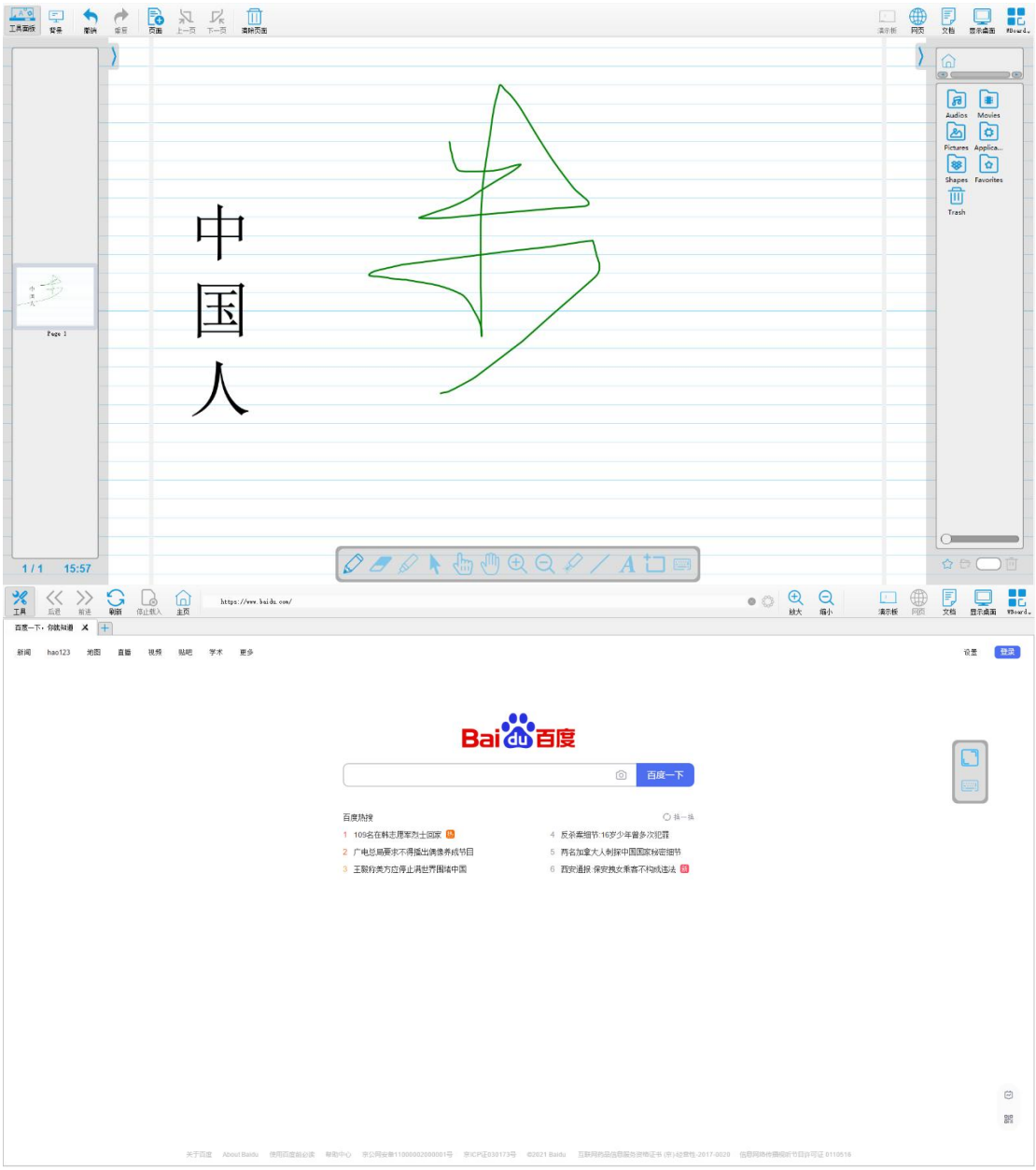
4.检查 Qt Version 配置与项目属性是否一致。

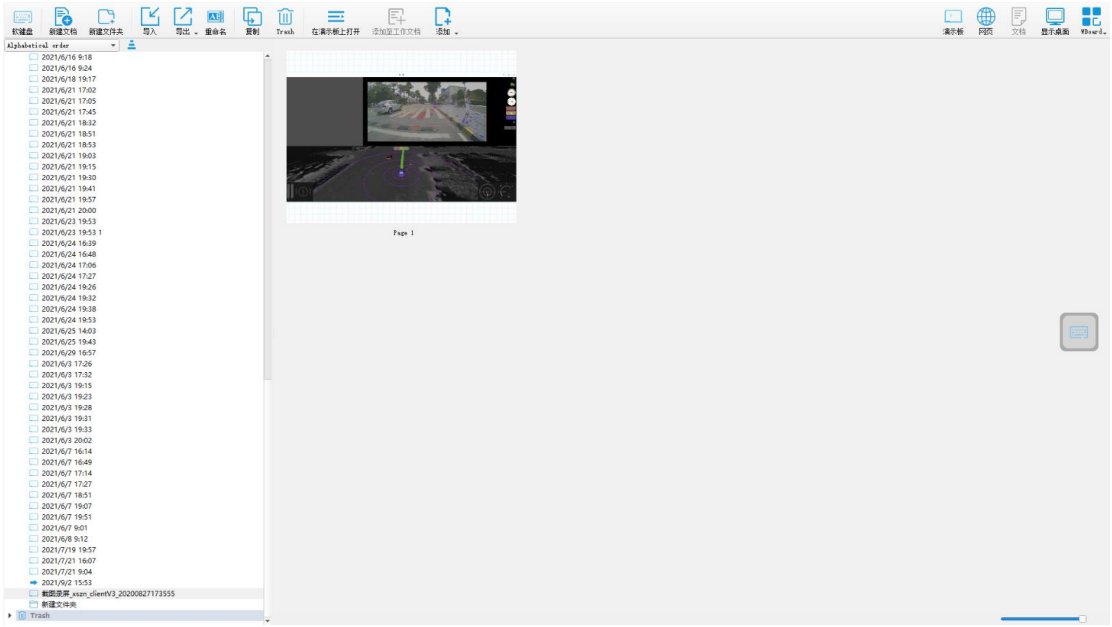


5.编译程序完成之后配置依赖，找到项目文件中的三方库 ThirdParty 文件夹，找到依赖动态库 quazipd.dll，再将 quazipd.dll 放在可执行程序下。


GeneratedFiles
Resources
Source
ThirdParty
TSResource
UI
x64
WBoard.qhvsor
WBoard.rc
WBoard.vcxproj
WBoard.vcxproj.filters
WBoard.vcxproj.user
openssl
quazip
Debug
include
lib
Release
bin
include
lib
quazipd.dll
WBoard.exe
WBoard.ilk
WBoard.pdb
quazipd.dll

6.项目运行截图


中
国
人
Audios
Movies
Pictures
Shapes
Favorites
Trash
1 / 1
15:57
Baidu百度
百度一下
百度热搜



项目模块解读

图形管理模块

图形管理模由图形动作、图形场景、图形形状、操作委托组成。

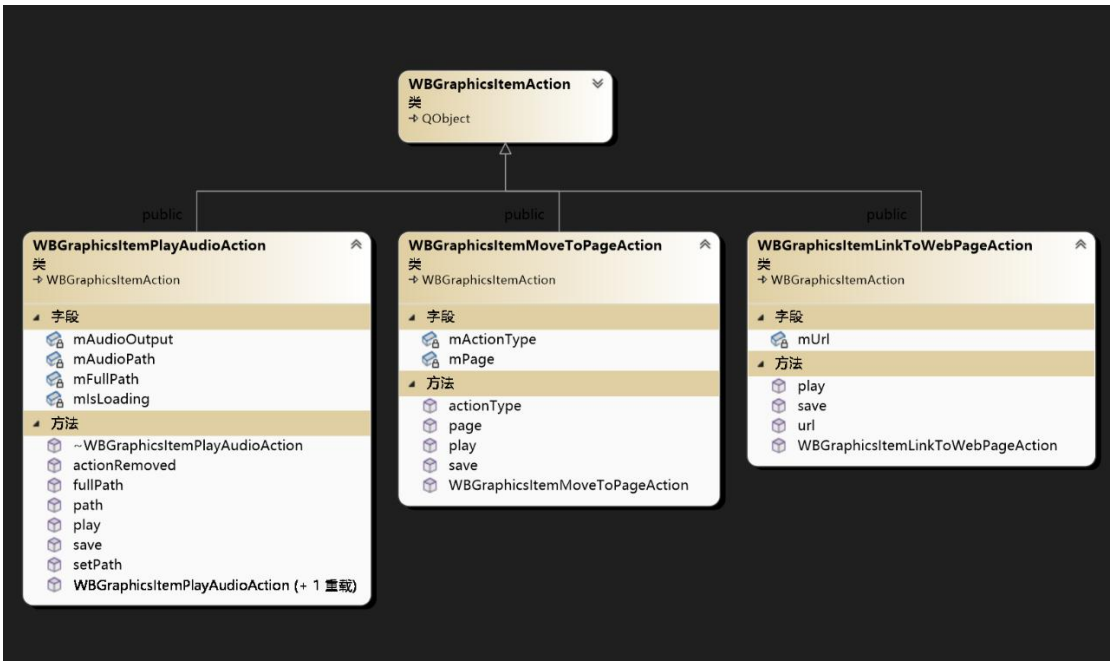
1.图形动作管理：负责管理和控制图形对象的各种动作，例如图形的创建、移动、缩放、旋转、删除等基本操作，以及更复杂的动画效果、交互动作的设置与执行

2.图形场景管理：主要对写字板项目中的图形展示场景进行管理。包括场景的布局设置、背景颜色与样式的定义、不同图形元素在场景中的层次与排列顺序管理，以及对场景视图的缩放、平移、旋转等操作的控制。

3.图形形状管理：专注于对图形的形状进行创建、编辑和管理。涵盖了基本图形（如矩形、圆形、三角形等）的绘制与参数设置，自定义图形的绘制工具与功能，对已绘制图形的形状修改（如顶点编辑、曲线调整等），以及图形的填充、描边样式的设置与管理。

4.图形操作委托(代理)：用于处理图形操作的任务分配与权限管理。当用户发起图形操作请求时，操作委托机制会根据预设的规则和权限，将操作任务分配给相应的处理模块或组件。同时，还负责监控和管理操作的执行过程，处理可能出现的错误与异常情况，确保图形操作的安全性和稳定性。

图形动作管理



WBGraphicsItemActions:　改基类主要用于处理图形项的各种操作和动作,定义了图形项操作的基本接口,声明了纯虚函数 play 用于执行操作、save 用于保存操作相关的数据、actionRemoved 用于处理操作被移除的情况,对应代码如下

```
class WBGraphicsItemAction : public QObject
{
    Q_OBJECT
public:
    WBGraphicsItemAction(eWBGraphicsItemLinkType linkType,QObject* parent = 0);
    virtual void play() = 0;
    virtual QStringList save() = 0;
    virtual void actionRemoved();
    virtual QString path() {return "";}
    eWBGraphicsItemLinkType linkType() { return mLinkType;}
private:
    eWBGraphicsItemLinkType mLinkType;
};
```

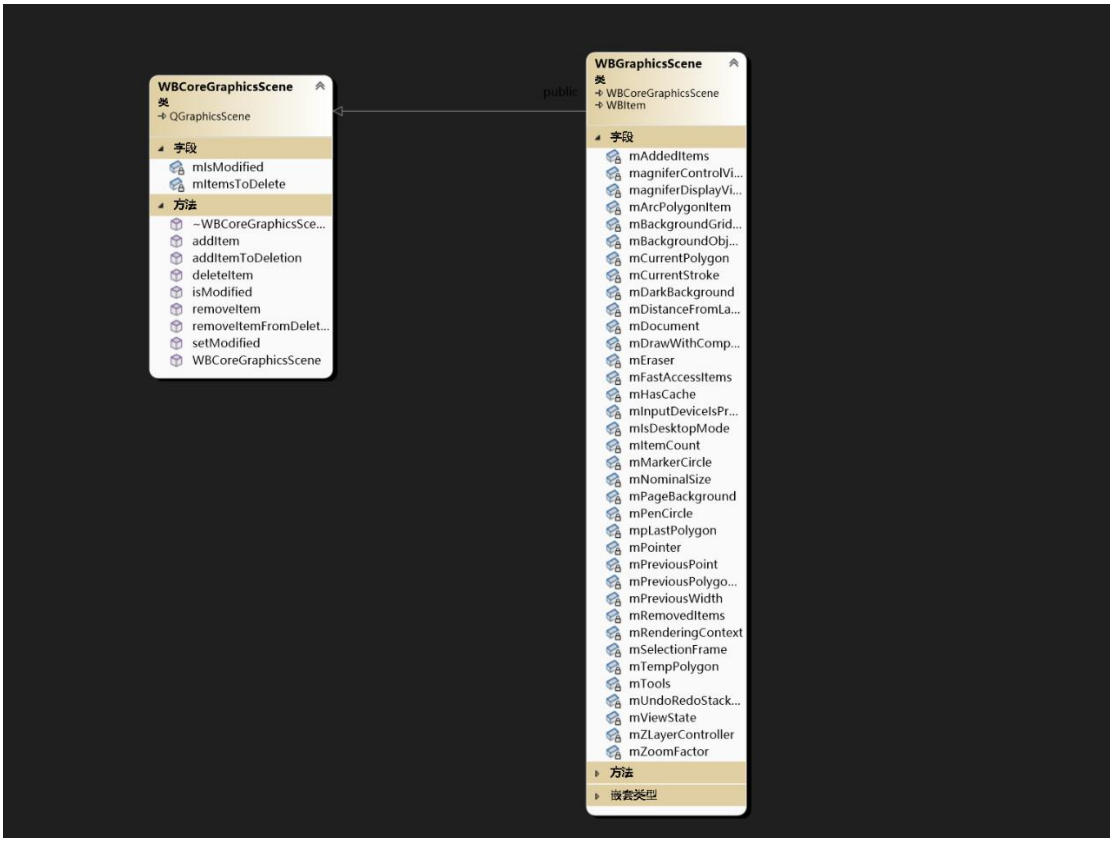
WBGraphicsItemPlayAudioAction：是用于播放音频的图形项操作类,存储了音频文件的路径、媒体播放器对象等,实现了 `play` 方法来播放音频,实现了 `save` 方法来保存相关数据，定义了 `onSourceHide` 槽函数，可能用于处理音频源隐藏的情况。

WBGraphicsItemMoveToPageAction: 是用于移动到特定页面的图形项操作类。包含了移动页面的操作类型（如移动到首页、末页、上一页、下一页或指定页）和目标页面的索引,实现了 `play` 方法来执行移动操作,实现了 `save` 方法来保存相关信息。

WBGraphicsItemLinkToWebPageAction: 是用于链接到网页的图形项操作类。

存储了链接的网页 URL 。实现了 `play` 方法来执行打开网页的操作。实现了 `save` 方法来保存链接相关的数据。

图形场景管理



WBCoreGraphicsScene: 该类是对`QGraphicsScene` 的扩展,在 QGraphicsScene 的基础上增加接口来`对图形项进行添加删除管理和场景修改状态的跟踪功能`，接口如下

```
class WBCoreGraphicsScene : public QGraphicsScene
{
    public:
        WBCoreGraphicsScene(QObject * parent = 0);
        virtual ~WBCoreGraphicsScene();
        virtual void addItem(QGraphicsItem* item);
        virtual void removeItem(QGraphicsItem* item, bool forceDelete = false);
        virtual bool deleteItem(QGraphicsItem* item);
        void removeItemFromDeletion(QGraphicsItem* item);
        void addItemToDeletion(QGraphicsItem *item);
        bool isModified() const{return mIsModified;}
        void setModified(bool pModified){mIsModified = pModified;}
    private:
        QSet<QGraphicsItem*> mItemsToDelete;
        bool mIsModified;
};
```

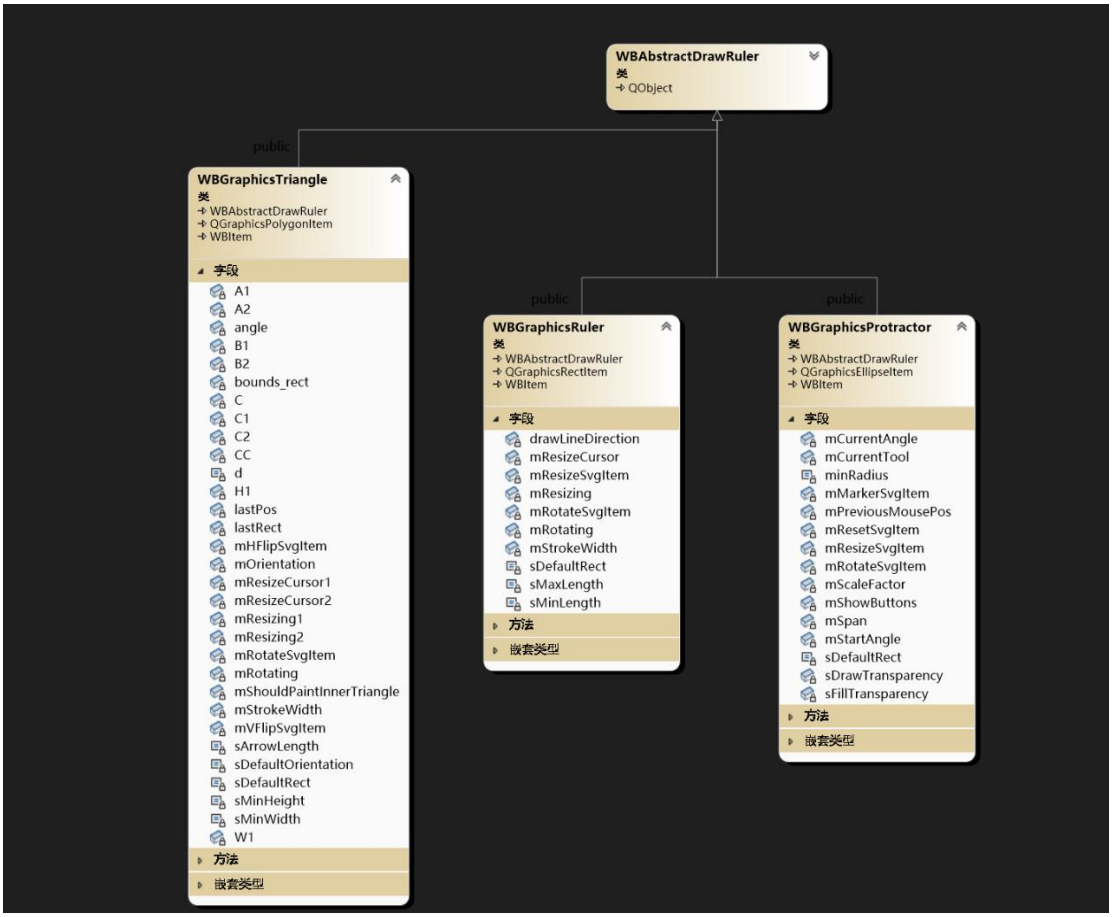
WBGraphicsScene: 是一个自定义的图形场景类，具有丰富的功能用于`管理图形项`、`处理输入事件、执行绘制操作`等。

1.提供了多种方法用于管理`撤销/重做栈的状态,清除场景内容（如项目、注释等）`、处理输入设备的操作（按下、移动、释放。

2.添加和移除图形项等方法,`能够创建和添加各种类型的图形项`，如小部件、媒体、SVG 图像、文本、组等,并且提供了一些方法用于处理背景对象的设置和获取，以及对`图形项进行缩放、适应文档大小`等操作。

3.包含一些与绘制相关的方法，如`画线、画弧、画曲线`等，并能`处理橡皮擦、指针、标记圆`等的绘制,还能支持获取场景的视图状态、设置渲染质量、获取依赖项、处理选择框的更新等。

图形形状管理



WBAbstractDrawRuler: 这个抽象类为具体的绘制尺子类提供了基本的框架和一些通用的操作，子类需要实现特定于场景和旋转等操作的细节。定义了一些虚函数，如 `StartLine`、`DrawLine`、`EndLine`，用于开始、绘制和结束线条。`paint` 方法用于绘制操作。包含一些纯虚函数，如 `scene`、`rotateAroundCenter` 等，需要子类具体实现。有一些保护成员变量用于控制显示、存储 SVG 项、比例等信息。定义了一系列的辅助函数用于获取不同的光标、颜色、字体等。包含一些静态常量用于颜色、边距、角度单位等的定义，具体定义如下图。

```
class WBAbstractDrawRuler : public QObject
{
    Q_OBJECT
public:
    WBAbstractDrawRuler();
    ~WBAbstractDrawRuler();
    void create(QGraphicsItem& item);
    virtual void StartLine(const QPointF& position, qreal width);
    virtual void DrawLine(const QPointF& position, qreal width);
    virtual void EndLine();
protected:
    void paint();
    QCursor moveCursor() const;
    QCursor rotateCursor() const;
    QCursor closeCursor() const;
    QCursor drawRulerLineCursor() const;
    QColor   drawColor() const;
    QColor   middleFillColor() const;
    QColor   edgeFillColor() const;
```

```
QFont     font() const;
virtual WBGraphicsScene* scene() const = 0;
virtual void rotateAroundCenter(qreal angle) = 0;
virtual QPointF rotationCenter() const = 0;
virtual QRectF closeButtonRect() const = 0;
virtual void paintGraduations(QPainter *painter) = 0;
bool mShowButtons;
QGraphicsSvgItem* mCloseSvgItem;
qreal mAntiScaleRatio;
QPointF startDrawPosition;
 };
```

WBGraphicsTriangle: 用于在图形场景中创建和操作三角形图形项，能够处理各种交互操作、方向设置、绘制细节等，并具有复制和自定义属性的功能。

1.重写了一系列图形项相关的方法，如绘制（`paint`）、形状计算（`shape`）、鼠标事件处理（`mousePressEvent`等)、旋转相关方法等。

2.包含了一些私有方法用于计算内部的点、变换、游标更新、边界框等。定义了一些私有成员变量来存储状态、方向、SVG 项、点的坐标等。定义了一些常量用于控制图形的默认值和一些尺寸限制。

WBGraphicsRuler: 用于在图形场景中创建和操作一个尺子图形项，能够响应各种交互事件并进行相应的绘制和状态更新。

1.重写了一些方法来`处理线条的起始、绘制和结束`，以及各种图形项相关的`事件处理方法（如鼠标事件、悬停事件）和绘制方法`。

2.包含一些私有成员变量来表示状态（如是否正在调整大小、旋转）和一些 SVG 项、游标等。定义了一些私有方法用于内部的绘制和操作，如填充背景、绘制旋转中心、更新调整大小的游标等。

3.定义了一些常量用于控制默认矩形和一些长度限制。

WBGraphicsProtractor: 用在图形场景中创建和操作一个量角器图形项，能够处理各种交互操作并进行相应的绘制和状态更新。

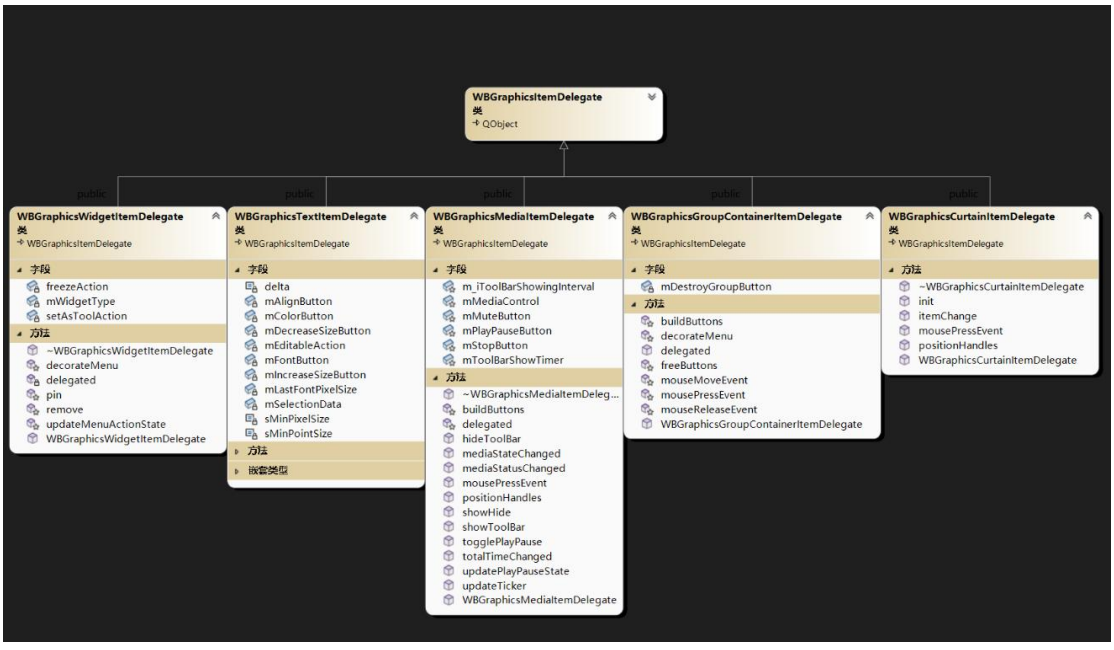
重写了多种图形项相关的方法，包括绘制（`paint`）、处理图形项变化（`itemChange`）、鼠标事（`mousePressEvent`、`mouseMoveEvent`、`mouseReleaseEvent`等）以及形状计算（`shape`、`boundingRect`）等。

2.包含一些私有方法用于处理内部的绘制细节（如绘制按钮、角度标记)、确定工具类型、计算反缩放等。

3.定义了一些私有成员变量来存储工具状态、角度值、缩放因子、SVG 项等。包含一些常量用于控制绘制的透明度和默认的矩形范围。

图形操作委托

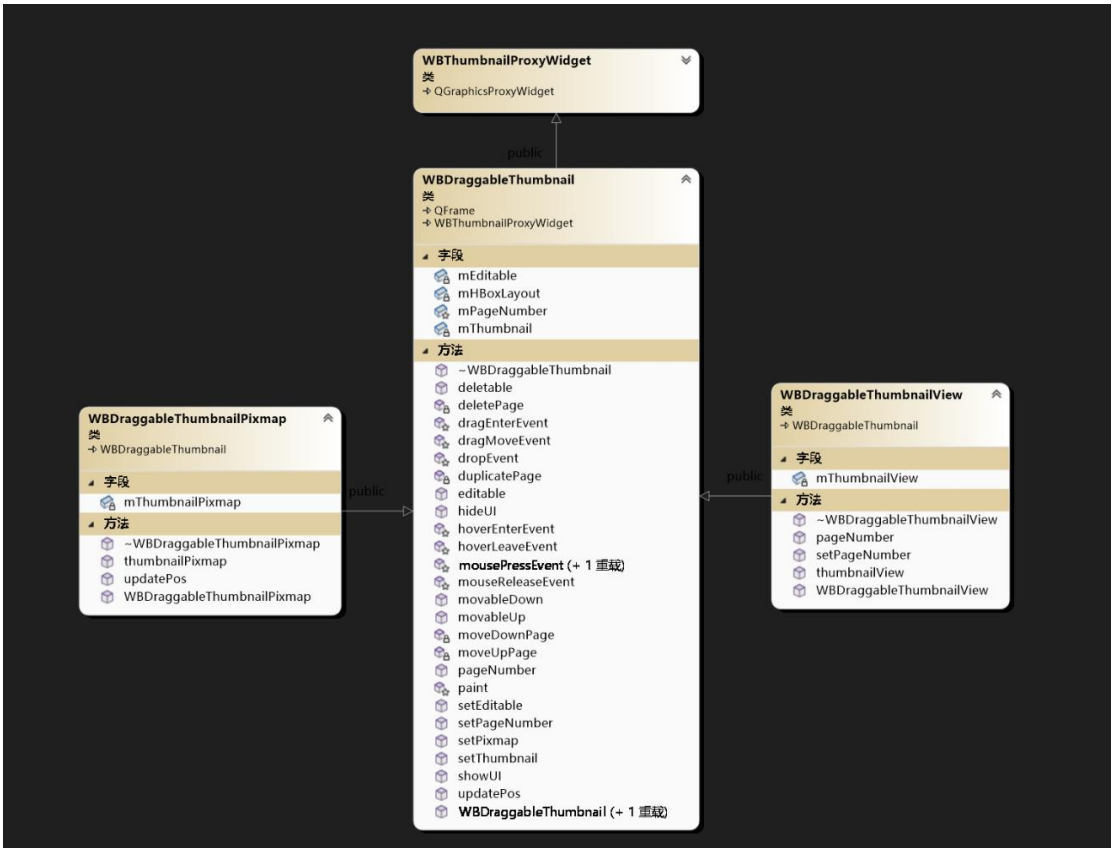
一：



WBGraphicsItemDelegate: 主要用于处理图形项的委托操作和相关控制,充当了图形项的代理，负责处理与图形项相关的各种交互操

作、状态管理和控制元素的显示与操作。
1.提供了一系列方法用于创建、释放和显示控制元素，处理各种鼠标、键盘和悬停事件，处理图形项的变化，执行撤销操作，以及处理图形项的锁定、显示隐藏、复制、顺序调整等操作。
2.包含了一些信号，用于通知显示状态和锁定状态的改变。公共槽函数用于执行移除、显示菜单、显示隐藏、锁定、复制、调整顺序、处理缩放变化等操作。
3.保护方法用于构建和释放按钮、装饰菜单、更新菜单动作状态，以及处理递归的显示隐藏和锁定操作。私有方法用于更新框架、按钮和相关数据。
3.成员变量用于存储被委托的图形项、各种按钮、菜单、框架、变换信息、标志、拖放相关的数据等。
WBGraphicsWidgetItemDelegate: 是 `WBGraphicsItemDelegate` 的派生类，专门用于处理 `WBGraphicsWidgetItem` 的委托操作,`WBGraphicsWidgetItem` 提供了特定的委托操作，包括菜单装饰、操作状态更新、移除处理以及特定的操作响应。
`decorateMenu`：用于为相关菜单添加特定于小部件的操作选项。
2. `updateMenuActionState`：用于更新菜单中操作的状态（例如是否可用、是否选中等）。
3.`remove`：处理小部件的移除操作，可能考虑是否支持撤销。
WBGraphicsTextItemDelegate: 是 `WBGraphicsItemDelegate` 的派生类，专门用于处理 `WBGraphicsTextItem` 的委托操作。`WBGraphicsTextItem` 提供了更具体和定制化的委托操作，包括与文本编辑相关的各种功能和交互处理。
1.提供了判断文本是否可编辑、缩放文本大小、重新着色等方法。
2.重写了 `itemChange` 方法以处理图形项的变化。实现了创建控件、处理槽函数（如内容改变、设置可编辑性、移除等）。
3.保护方法用于装饰菜单、更新菜单动作状态、释放按钮以及处理鼠标和键盘事件。>3.私有方法和成员变量用于处理字体选择、颜色选择、文本大小调整、对齐按钮状态更新、处理选择数据、创建默认字体等操作。
WBGraphicsMediaItemDelegate: 是 `WBGraphicsItemDelegate` 的派生类，用于处理 `WBGraphicsMediaItem` 的委托操作。`WBGraphicsMediaItem` 提供了委托操作，包括处理鼠标事件、媒体状态变化、操作按钮的构建和响应，以及工具栏的显示和隐藏控制。
1. 公共槽函数：`toggleMute` ：用于切换静音状态。`updateTicker` ：可能用于更新时间显示。 `showHide` ：显示或隐藏相关元素。`mediaStatusChanged` 和 `mediaStateChanged` ：用于响应媒体的状态变化。
2.保护槽函数：`remove` ：执行移除操作。 `togglePlayPause` ：切换播放/暂停状态。`updatePlayPauseState` ：更新播放/暂停的状态显示。`totalTimeChanged` ：当总时间改变时进行处理。`hideToolBar` ：隐藏工具栏。
3. 保护方法 `buildButtons` 用于构建相关的按钮。
4. 成员变量：`mPlayPauseButton` 、 `mStopButton` 、 `mMuteButton` ：与播放/暂停、停止、静音操作相关的按钮。`mMediaControl` ：可能用于媒体控制的相关组件。 `mToolBarShowTimer` ：工具栏显示的定时器。`m_iToolBarShowingInterval` ：工具栏显示的时间间隔。
5. 私有槽函数：`freeze`：处理小部件的冻结操作。`pin`：可能用于执行将小部件固定或类似的操作。
WBGraphicsGroupContainerItemDelegate: 类是 `WBGraphicsItemDelegate` 的派生类，用于处理 `WBGraphicsGroupContainerItem` 的委托操作。`WBGraphicsGroupContainerItem` 提供了特定的委托操作，包括菜单装饰、按钮管理和鼠标事件处理，以及实现了销毁分组的功能。
`decorateMenu` 用于装饰相关的菜单。`buildButtons` 用于构建特定的按钮。`freeButtons` 用于释放按钮相关的资源。重写了鼠标事件处理方法，用于处理在分组容器上的鼠标操作。`destroyGroup` 可能用于执行销毁分组的操作。 `mDestroyGroupButton` 用于存储与销毁分组操作相关的按钮
WBGraphicsCurtainItemDelegate: 用于为 `WBGraphicsCurtainItem` 提供特定的事件处理和操作逻辑。

二：



WBThumbnailProxyWidget: 作为`WBThumbnailWidget`类的代理，处理`WBThumbnailWidget`的委托操作，用于处理缩略图的选择、布局、鼠标事件、大小调整等。

```
class WBThumbnailProxyWidget : public QGraphicsProxyWidget
{
public:
    WBThumbnailProxyWidget(WBDocumentProxy* proxy, int index)
        : mDocumentProxy(proxy)
        , mSceneIndex(index){}
    WBDocumentProxy* documentProxy(){return mDocumentProxy;}
    void setSceneIndex(int i){mSceneIndex = i;}
    int sceneIndex(){return mSceneIndex;}
private:
    WBDocumentProxy* mDocumentProxy;
    int mSceneIndex;
};
```

WBThumbnailPixmap：基于 `QGraphicsPixmapItem` ，处理缩略图的图像显示。

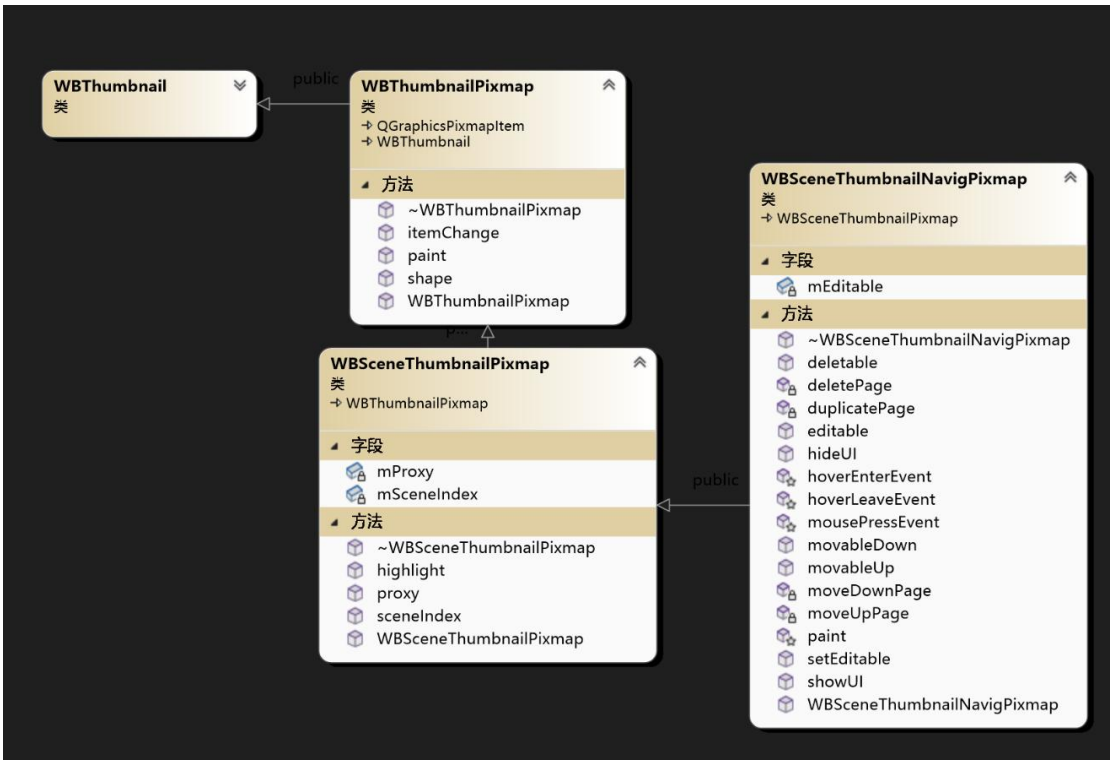
```
class WBDraggableThumbnailPixmap : public WBDraggableThumbnail
{
    Q_OBJECT
public:
    WBDraggableThumbnailPixmap(WBThumbnailPixmap* thumbnailPixmap, WBDocumentProxy* documentProxy, int index)
        : WBDraggableThumbnail(documentProxy, index)
        , mThumbnailPixmap(thumbnailPixmap)
    {
        setFlag(QGraphicsItem::ItemIsSelectable, true);
        setAcceptDrops(true);
```

```
    }
    ~WBDraggableThumbnailPixmap()
    {}
    WBThumbnailPixmap* thumbnailPixmap()
    { return mThumbnailPixmap;}
    void updatePos(qreal w, qreal h);
private:
    WBThumbnailPixmap* mThumbnailPixmap;
};
```

WBDraggableThumbnail: 是一个可拖动的缩略图框架。使得显示的缩略图具有可拖动的特性，方便在支持拖放操作的界面中进行交互和操作。`setThumbnail` 和 `setPixmap` 方法用于设置显示的缩略图。`dragEnterEvent` ： 处理拖放操作进入时的事件。`dragMoveEvent` ： 处理拖放操作移动时的事件。`dropEvent` ： 处理拖放操作放下时的事件。 `mousePressEvent` ： 处理鼠标按下事件，可能用于启动拖动操作。

```
class WBDraggableThumbnail : public QFrame
{
    public:
        WBDraggableThumbnail(QWidget* parent =0, const QPixmap& pixmap = QPixmap(":images/libpalette/notFound.png"));
        void setThumbnail(const QPixmap &pixmap);
        void setPixmap(const QPixmap & pixmap);
    protected:
        void dragEnterEvent(QDragEnterEvent *event);
        void dragMoveEvent(QDragMoveEvent *event);
        void dropEvent(QDropEvent *event);
        void mousePressEvent(QMouseEvent *event);
    private:
        QLabel* mThumbnail;
        QHBoxLayout* mHBoxLayout;
};
```

三：



WBThumbnail：用于处理与缩略图相关的通用属性和操作。
WBThumbnailPixmap: 基于 `QGraphicsPixmapItem` ，处理缩略图的图像显示。
WBSceneThumbnailPixmap: 特定于场景的缩略图图像。
WBSceneThumbnailNavigPixmap: 具有更多导航相关的操作和属性。

窗口管理模块

窗口管理分别有窗口面板、浮动调色板、编辑控件。
1.窗口面板：负责管理写字板项目中的各个窗口面板。包括打开、关闭、切换不同的面板，调整面板的大小、位置和显示状态。还可以对面板中的内容进行组织和布局，例如将不同功能的工具、选项、属性设置等分组放置在不同的面板中，以提供清晰和便捷的用户操作界面。
2.浮动调色板：在写字板项目中用于颜色的选择和管理。用户可以通过浮动调色板轻松选取所需的颜色，对绘制的图形、文字等元素进行颜色填充或描边。调色板可能支持自定义颜色的添加、删除和编辑，以及颜色方案的保存和加载，以满足不同的创作需求。
3.编辑控件：是写字板项目中用于文本编辑和图形元素编辑的功能组件。对于文本编辑，编辑控件可能包括字体、字号、颜色、对齐方式、缩进等设置功能；对于图形元素编辑，编辑控件可能用于调整图形的大小、形状、旋转角度、层次顺序等。此外，编辑控件还可能包含撤销、重做、复制、粘贴、剪切等基本编辑操作的按钮或功能选项。

窗口面板

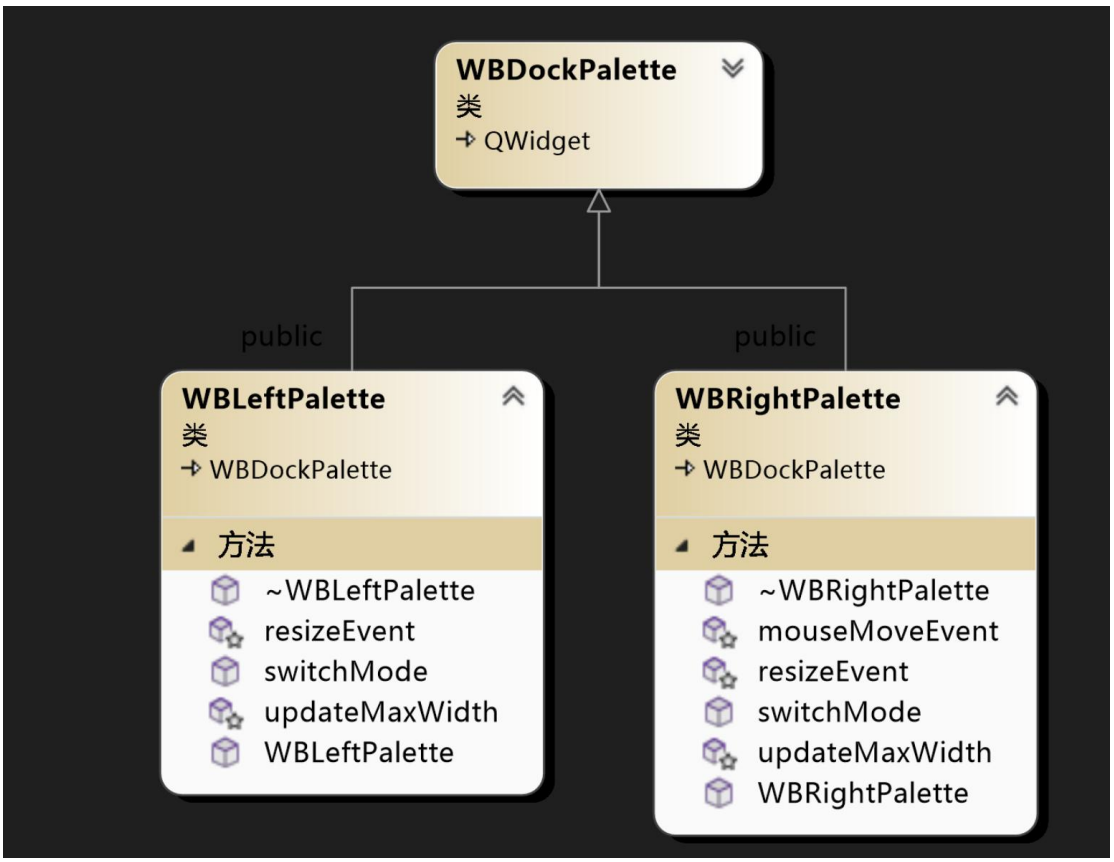
WBDockPaletteWidget: 是一个用于停靠面板的基础部件类。为停靠面板中的具体部件提供了基本的框架和一些通用的功能，而具体的可见性逻辑由派生类根据不同的模式来实现
`iconToRight` 和 `iconToLeft` ：可能用于获取左右方向的图标, `name` ：获取部件的名称。`registerMode` ：注册部件支持的模式。 `visibleInMode` ：要求派生类实现根据不同模式决定是否可见的逻辑。
WBFeaturesWidget：继承自 `WBDockPaletteWidget` ，包含了与功能控制器、视图、布局、操作栏等相关的成员变量。定义了一些信号和私有槽函数，用于处理各种操作和交互事件，如文件加载、元素选择、搜索等。
WBDockDownloadWidget: 是一个从 `WBDockPaletteWidget` 派生的自定义部件，用于处理下载相关的功能展示。为了在特定

模式下展示与下载相关的界面和功能。

WBCachePropertiesWidget：是一个自定义的窗口部件，用于显示和操作图形缓存的相关属性。提供了一个用户界面，用于展示图形缓存的属性，并允许用户进行颜色、形状和大小等方面的设置和调整。

WBPageNavigationWidget：它继承自 `WBDockPaletteWidget` ，调整网页大小，设置网页页码，网页刷新。

二：

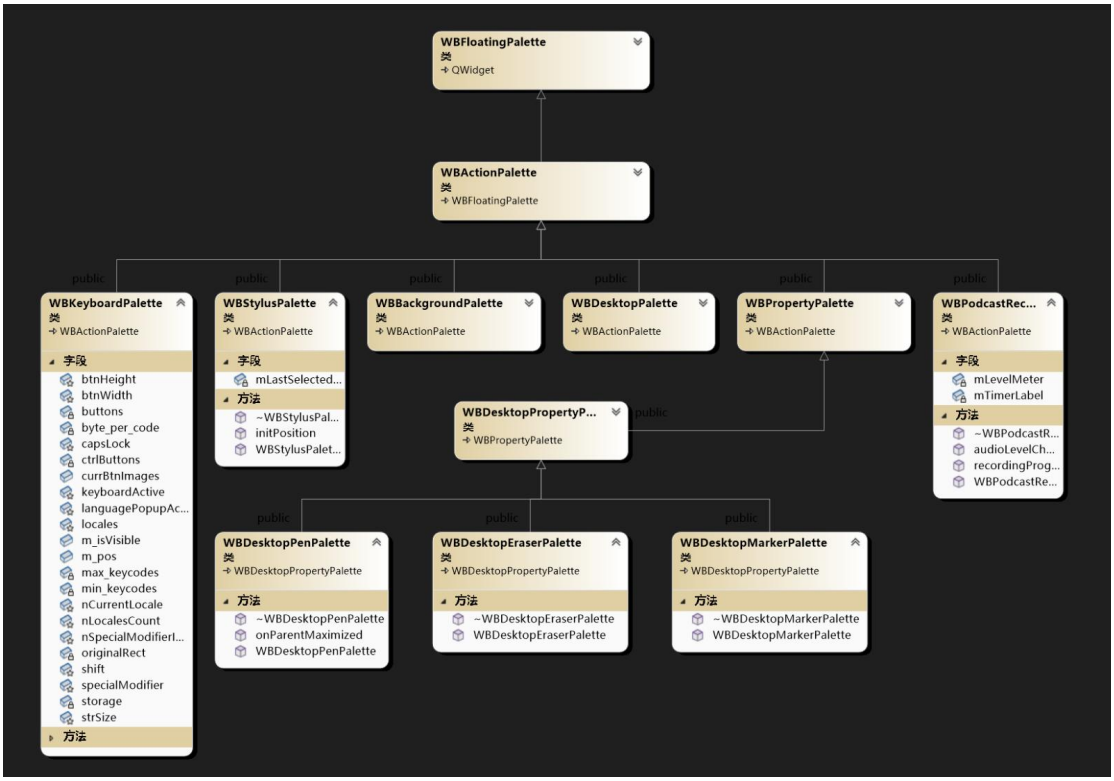


WBDockPalette: 是一个自定义的停靠面板类，用于管理和展示一系列相关的部件。提供了一个可定制的停靠面板框架，用于组织和管理多个相关的部件，并处理它们的显示、交互和布局调整等功能。

WBLeftPalette：当需要在窗口左侧显示特定的调色板功能，并根据不同情况切换模式、处理文档设置以及响应窗口大小变化时，可以使用 `WBLeftPalette` 类。当有文档设置的相关操作时，会调用 `onDocumentSet` 槽函数进行相应的处理。

WBRightPalette：当需要在窗口右侧显示特定的调色板功能，并根据不同情况切换模式、处理文档设置以及响应窗口大小变化时，可以使用 `WBRightPalette` 类。当有文档设置的相关操作时，会调用 `onDocumentSet` 槽函数进行相应的处理。

浮动调色板



WBFloatingPalette: 该类为可浮动的调色板或窗口组件。定义了一个枚举类型 `eMinimizedLocation`，用于表示调色板最小化时的位置选项。继承自 `QWidget` 重写了一些鼠标事件处理函数，如 `mouseMoveEvent`、`mousePressEvent` 和 `mouseReleaseEvent`，可能用于实现窗口的拖动功能。提供了一些方法来管理关联的调色板、调整大小和位置、设置自定义位置、背景刷、抓取状态、最小化权限等。重写了一些保护成员函数，如 `enterEvent`、`showEvent` 和 `paintEvent`，可能用于处理鼠标进入、显示和绘制事件。具有一些私有方法用于内部管理，如移除所有关联的调色板、最小化调色板等。定义了一些信号，用于通知外部关于鼠标进入、最小化开始、最大化开始、已最大化和正在移动等状态的变化。

WBActionPalette: 是一个自定义的动作调色板类，用于管理和展示一系列动作,`WBActionPalette` 类提供了一个可定制的、可包含多个动作的调色板界面，并处理与调色板的显示、布局、动作管理和用户交互相关的功能。

WBKeyboardPalette: 可以作为一个虚拟键盘的界面组件，通过处理各种事件和信号，与应用的其他部分进行交互，实现键盘输入的功能。当用户点击某个键盘按钮时，会触发相应的 `onPress` 和 `onRelease` 方法来处理按键操作。

WBStylusPalette：它继承自 `WBActionPalette`，当触笔工具被双击时，会触发 `stylusToolDoubleClicked` 槽函数，并通过信号将相关信息传递给其他连接的对象，以执行相应的操作。

WBBackgroundPalette：是从 `WBActionPalette` 派生的类，用于处理与背景相关的操作和显示，为用户提供了与背景设置和操作相关的交互界面。

WBDesktopPalette：是一个与桌面操作相关的调色板类。这个类主要负责管理和控制与桌面操作相关的调色板的显示、隐藏、按钮操作以及与其他组件的交互。

```
class WBDesktopPropertyPalette : public WBPropertyPalette
{
    Q_OBJECT

    public:
        WBDesktopPropertyPalette(QWidget *parent, WBRightPalette* _rightPalette);
    private:
        WBRightPalette* rightPalette;
    protected:
        virtual int getParentRightOffset();
};
```

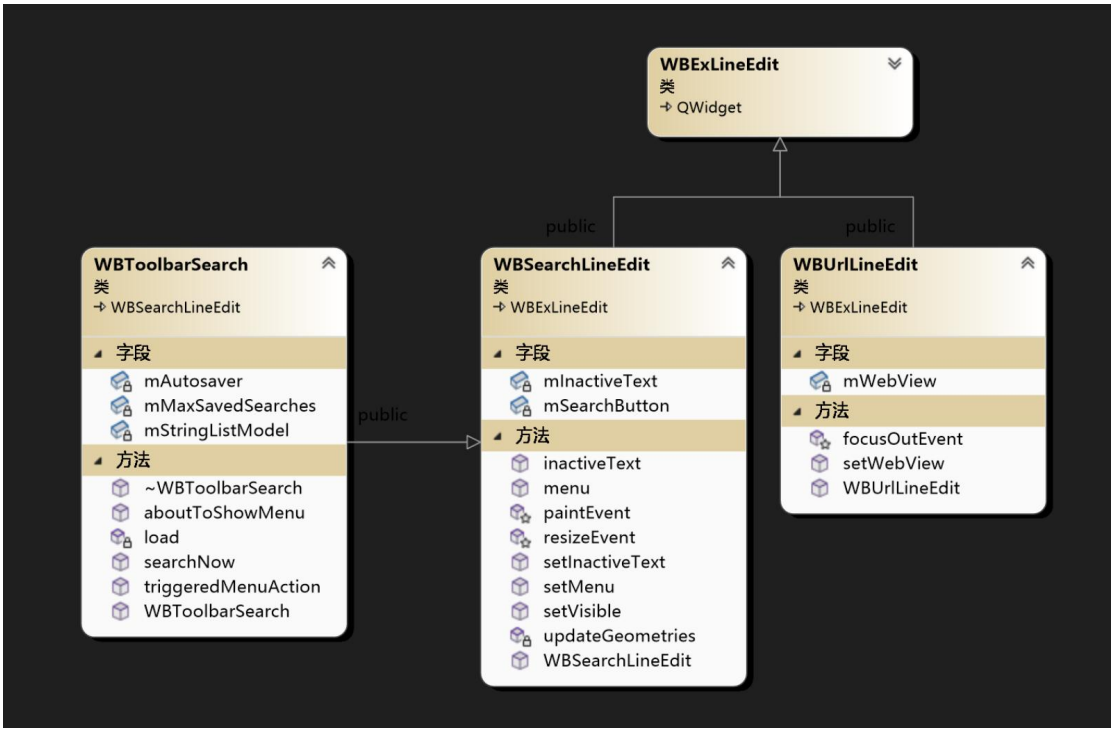
```
class WBDesktopPenPalette : public WBDesktopPropertyPalette
{
    Q_OBJECT
    public:
        WBDesktopPenPalette(QWidget *parent, WBRightPalette* rightPalette);
        virtual ~WBDesktopPenPalette(){}
    public slots:
        void onParentMinimized();
        void onParentMaximized();

    private slots:
        void onButtonReleased();

};
```

WBPodcastRecordingPalette：用于在播客录制过程中展示相关的状态和信息，响应录制状态、录制进度和音频电平的变化。
WBDesktopPropertyPalette：它是 `WBPropertyPalette` 的子类，为桌面相关的属性调色板提供了一些基础功能。定义了获取父窗口右边偏移量的方法。
WBDesktopPenPalette：继承自 `WBDesktopPropertyPalette` 。包含了处理父窗口最小化和最大化的槽函数，以及按钮释放的私有槽函数。
WBDesktopEraserPalette：继承自 `WBDesktopPropertyPalette` ，可能用于处理与桌面橡皮擦相关的属性设置。
WBDesktopMarkerPalette：继承自 `WBDesktopPropertyPalette` ，可能用于处理与桌面标记相关的属性设置。

编辑控件



WBExLineEdit：继承自 QWidget 实现了一个自定义的编辑控件。包含了一些方法来管理左右的部件、处理各种事件（如焦点事件、键盘事件、绘制事件等），提供了方法来更新部件的几何形状和初始化样式选项。

```
class WBExLineEdit : public QWidget
{
    Q_OBJECT;
```

```
public:
    WBExLineEdit(QWidget *parent = 0);
    inline QLineEdit *lineEdit() const { return mLineEdit; }
    void setLeftWidget(QWidget *widget);
    QWidget *leftWidget() const;
    QSize sizeHint() const;
    QVariant inputMethodQuery(Qt::InputMethodQuery property) const;
    void setVisible(bool pVisible);
protected:
    void focusInEvent(QFocusEvent *event);
    void focusOutEvent(QFocusEvent *event);
    void keyPressEvent(QKeyEvent *event);
    void paintEvent(QPaintEvent *event);
    void resizeEvent(QResizeEvent *event);
    void inputMethodEvent(QInputMethodEvent *e);
    bool event(QEvent *event);
protected:
    void updateGeometries();
    void initStyleOption(QStyleOptionFrameV2 *option) const;
    QWidget*            mLeftWidget;
    QLineEdit*          mLineEdit;
    WBClearButton*      mClearButton;
};
```

WBSearchLineEdit：继承自 `WBExLineEdit`， 提供了获取和设置非活动状态文本的方法。可以获取和设置菜单，并处理可见性的设置。重写了 `resizeEvent` 和 `paintEvent` 方法来处理大小调整和绘制事件。定义了一个信号 `textChanged` ，用于在文本变化时发出信号。包含一些私有方法，如 `updateGeometries` ，用于更新几何形状。

WBUrlLineEdit：继承自 `WBExLineEdit` 。可以设置关联的 `WBWebView` ，并在焦点失去事件和 `WebView` 的 URL 变化时进行相应处理。专门用于处理与 URL 相关的输入和与 `WBWebView` 的交互。

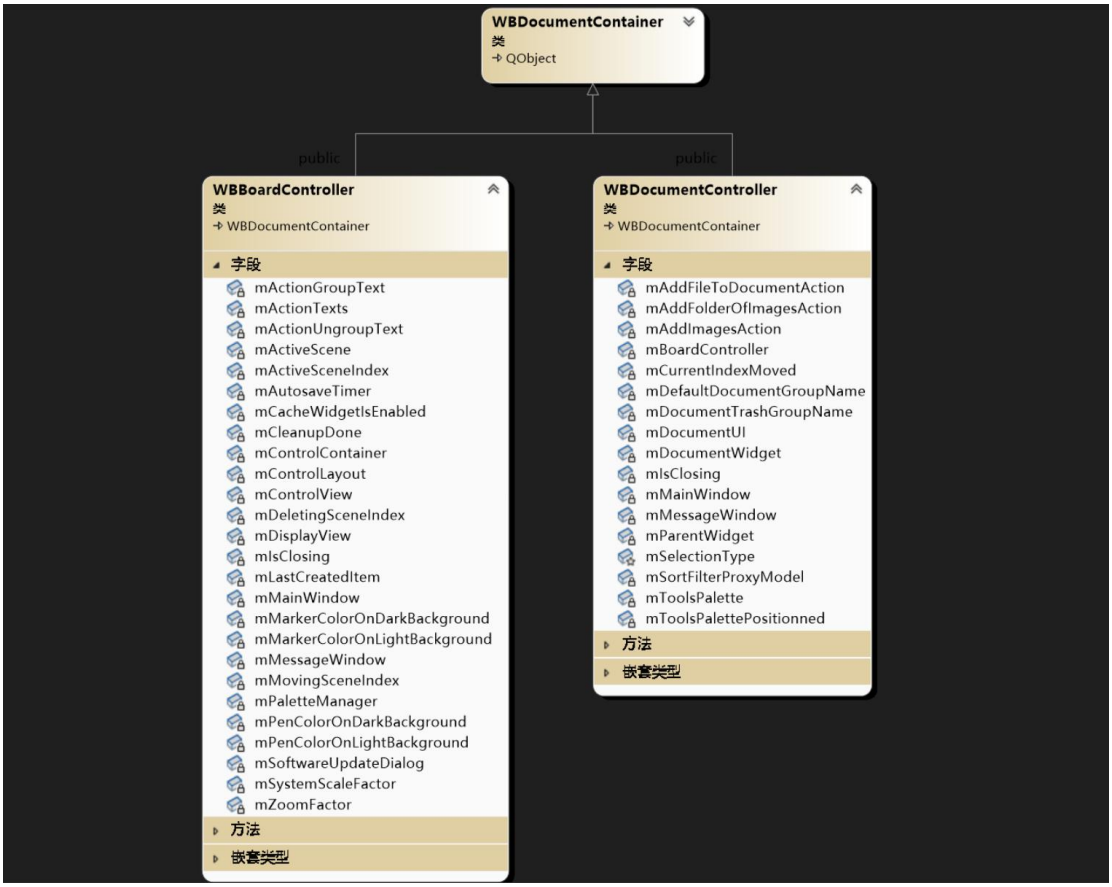
WBToolbarSearch：它继承自 `WBSearchLineEdit` 。用于在工具栏中实现搜索功能，能够保存搜索历史并根据用户操作执行相应的搜索操作。

文档管理模块

文档管理模块分别有文档导出、文档加载。

1.文档导出：负责将在写字板中创建和编辑的文档内容以指定的格式进行输出。用户可以选择将文档导出为常见的文件格式，如 PDF、DOCX、TXT 等，以便于在不同的设备和软件中查看、打印或进一步编辑。在文档导出过程中，可能会涉及对文档的格式设置、页面布局调整、字体嵌入、图像质量优化等操作，以确保导出的文档能够准确呈现原始文档的内容和样式。用于将外部的文档文件加载到写字板应用程序中进行查看和编辑。它支持多种文档格式的识别和读取，当用户选择打开一个文档文件时，文档加载模块会解析文件的内容和格式，并将其转换为写字板内部可编辑的格式进行显示。此外，文档加载模块还可能需要处理文档的加密、权限验证等安全相关的问题，以确保只有授权的用户能够访问和编辑相应的文档。

2.文档加载：用于将外部的文档文件加载到写字板应用程序中进行查看和编辑。它支持多种文档格式的识别和读取，当用户选择打开一个文档文件时，文档加载模块会解析文件的内容和格式，并将其转换为写字板内部可编辑的格式进行显示。此外，文档加载模块还可能需要处理文档的加密、权限验证等安全相关的问题，以确保只有授权的用户能够访问和编辑相应的文档。



WBDocumentContainer：主要用于管理文档相关的操作和数据。包含了对文档缩略图的一系列操作方法，如初始化、更新、添加、删除、插入等。通知其他组件关于文档的设置、页面的更新、缩略图的操作等情况。

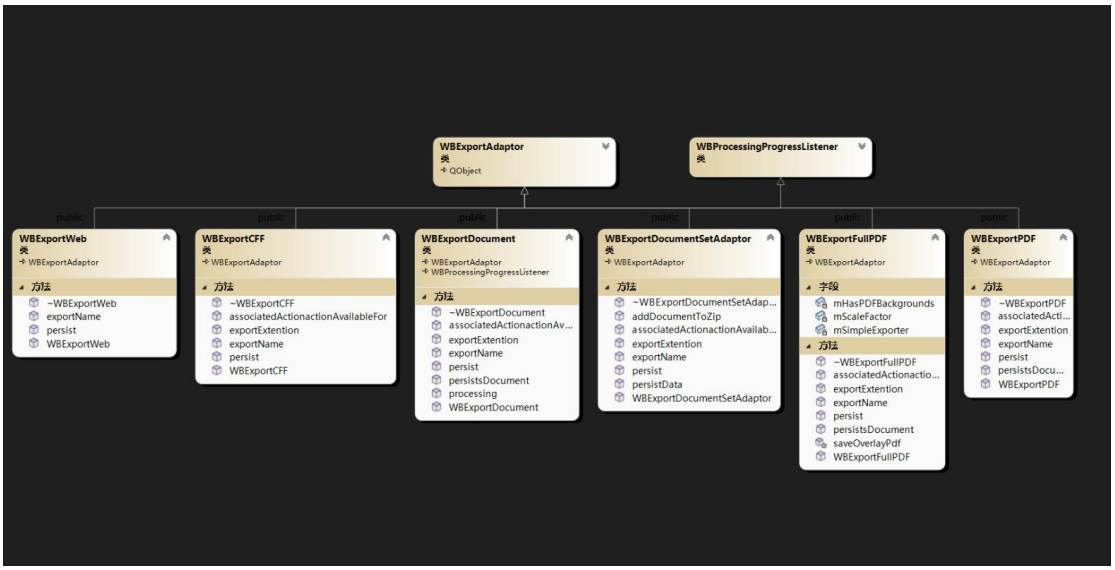
WBDocumentController：主要负责对文档的全面管理和控制。包含多个公共槽函数，用于响应各种用户操作，如创建新文档、刷新日期列、重新排序文档、创建新文档组、删除选中项等。

设置视图、工具栏、调色板等，并进行一些相关的检查和获取操作，如检查是否可以打开文档、获取选中的文档代理等。

处理一些内部的操作和响应，如文档缩放滑块值改变、选择元素类型改变、导出文档等。

WBBoardController：主要负责管理和控制与白板相关的各种操作和属性。定义了一些信号，用于与其他组件进行通信，如通知新页面添加、活动场景改变、缩放改变等。用于内部的处理逻辑，如更新操作状态、处理自动保存超时、处理应用主模式改变等。这个类充当了白板应用的核心控制器，协调各种操作、处理状态变化，并与其他相关组件进行交互，以实现白板的各种功能和行为。

文档导出



WBExportAdaptor：是一个基于 QObject 的抽象基类，主要用于定义与文档导出相关的通用接口和一些辅助功能。这个类为不同的导出适配器提供了一个统一的接口框架，使得在处理各种不同类型的文档导出操作时，可以遵循相同的基本结构和方法，同时允许每个具体的导出适配器根据自身需求实现特定的导出逻辑和行为

```
class WBExportAdaptor : public QObject
{
    Q_OBJECT

    public:
        WBExportAdaptor(QObject *parent = 0);
        virtual ~WBExportAdaptor();
        virtual QString exportName() = 0;
        virtual QString exportExtention() { return "";}
        virtual void persist(WBDocumentProxy* pDocument) = 0;
        virtual bool persistsDocument(WBDocumentProxy* pDocument, const QString& filename);
        virtual bool associatedActionactionAvailableFor(const QModelIndex &selectedIndex) {Q_UNUSED(selectedIndex);
return false;}
        QAction *associatedAction() {return mAssociatedAction;}
        void setAssociatedAction(QAction *pAssociatedAction) {mAssociatedAction = pAssociatedAction;}
        virtual void setVerbose(bool verbose)
        {
            mIsVerbose = verbose;
        }
        virtual bool isVerbose() const
        {
            return mIsVerbose;
        }
    protected:
        QString askForFileName(WBDocumentProxy* pDocument, const QString& pDialogTitle);
        QString askForDirName(WBDocumentProxy* pDocument, const QString& pDialogTitle);
        virtual void persistLocally(WBDocumentProxy* pDocumentProxy, const QString &pDialogTitle);
        void showErrorsList(QList<QString> errorsList);
        bool mIsVerbose;
        QAction* mAssociatedAction;
};
```

以下是对其作用的详细分析：

- `exportName()`要求子类必须实现，用于指定导出操作的名称。
- `exportExtention()`默认返回空字符串，子类可以重写以提供特定的导出文件扩展名。
- `persist(WBDocumentProxy* pDocument)`要求子类必须实现，执行实际的文档持久化（导出）操作。
- `persistDocument(WBDocumentProxy* pDocument, const QString& filename)`默认实现，可能提供了一些通用的文档持久化逻辑。
- `associatedActionactionAvailableFor(const QModelIndex &selectedIndex)`默认返回 false ，子类可以重写以根据特定条件判断相关操作是否可用。
- askForFileName 和 askForDirName ：用于向用户请求文件名和目录名。
- persistLocally ：由子类重写以实现本地持久化的具体逻辑。
- showErrorsList ：用于显示错误列表。

总的来说，这个类为不同的导出适配器提供了一个统一的接口框架，使得在处理各种不同类型的文档导出操作时，可以遵循相同的基本结构和方法，同时允许每个具体的导出适配器根据自身需求实现特定的导出逻辑和行为。

例如，假设我们有一个子类 PDFExportAdaptor 继承自 WBExportAdaptor ，它可以重写 exportName 函数返回 "PDF Export" ，重写 persist 函数实现将文档导出为 PDF 格式的具体逻辑，并根据 PDF 导出的特点重写其他相关函数。

文件加载

WBImportAdaptor：作为所有导入适配器的基类。用于文件导入接口定义

```
class WBImportAdaptor : public QObject
{
    Q_OBJECT;
    protected:
        WBImportAdaptor(bool _documentBased, QObject *parent = 0);
        virtual ~WBImportAdaptor();
    public:
        virtual QStringList supportedExtentions() = 0;
        virtual QString importFileFilter() = 0;
        bool isDocumentBased(){return documentBased;}
    private:
        bool documentBased;
};
```

WBPageBasedImportAdaptor：继承自 WBImportAdaptor，是基于页面的导入适配器的抽象类。

WBDocumentBasedImportAdaptor：同样继承自 WBImportAdaptor，是基于文档的导入适配器的抽象类。

```
class WBPageBasedImportAdaptor : public WBImportAdaptor
{
     protected:
        WBPageBasedImportAdaptor(QObject *parent = 0);
     public:
        virtual QList<WBGraphicsItem*> import(const QUuid& uuid, const QString& filePath) = 0;
        virtual void placeImportedItemToScene(WBGraphicsScene* scene, WBGraphicsItem* item) = 0;
        virtual const QString& folderToCopy() = 0;
};
class WBDocumentBasedImportAdaptor : public WBImportAdaptor
{
```

```
    protected:
        WBDocumentBasedImportAdaptor(QObject *parent = 0);
    public:
    virtual WBDocumentProxy* importFile(const QFile& pFile, const QString& pGroup) = 0;
    virtual bool addFileToDocument(WBDocumentProxy* pDocument, const QFile& pFile) = 0;
};
```

动作响应管理

动作响应模块分别有撤回操作、键盘操作。

1.撤回操作：允许用户撤销最近执行的操作，恢复到上一个操作前的文档状态。当用户在写字板中进行了误操作，如误删除文本、误绘制图形或错误地修改了格式等，通过撤回操作可以快速返回到之前的正确状态。撤回操作通常具有一定的可回溯次数和历史记录，以满足用户在不同场景下的需求。

2.键盘操作：主要处理用户通过键盘与写字板应用程序的交互。用户可以使用键盘上的按键来输入文本内容、执行快捷键操作、切换工具或功能、控制光标的移动和选择等。键盘操作还包括对特殊功能键（如 Ctrl、Shift、Alt 等）组合的响应，以实现诸如复制、粘贴、全选、查找替换等功能，为用户提供高效便捷的文档编辑体验。

撤回操作

WBPageSizeUndoCommand：是一个用于处理页面大小更改的撤销命令类。

```
class WBPageSizeUndoCommand : public WBUndoCommand
{
    public:
        WBPageSizeUndoCommand(WBGraphicsScene* pScene, const QSize& previousSize, const QSize& newSize);
        virtual ~WBPageSizeUndoCommand();
        virtual int getType() { return WBUndoType::undotype_PAGESIZE; };
    protected:
        virtual void undo();
        virtual void redo();
```

```
    private:
        WBGraphicsScene* mScene;
        QSize mPreviousSize;
        QSize mNewSize;
        bool mFirstRedo;
};
```

`getType` 方法返回特定的撤销命令类型，这里是与页面大小相关的类型。`undo` 方法用于执行撤销操作，将页面大小恢复为之前的状态。

`redo` 方法用于执行重做操作，将页面大小设置为新的状态。

`mScene` ： 关联的图形场景。

`mPreviousSize` ： 页面更改前的大小。

`mNewSize` ： 页面更改后的大小。

`mFirstRedo` ： 标记是否是第一次重做。

WBGraphicsItemGroupUndoCommand：是一个用于处理图形项分组操作的撤销命令类。

```
class WBGraphicsItemGroupUndoCommand : public WBUndoCommand
{
public:
    WBGraphicsItemGroupUndoCommand(WBGraphicsScene *pScene, WBGraphicsGroupContainerItem *pGroupCreated);
    virtual ~WBGraphicsItemGroupUndoCommand();
    virtual int getType() const { return WBUndoType::undotype_GRAPHICSGROUPITEM; }
protected:
    virtual void undo();
    virtual void redo();
private:
    WBGraphicsScene *mScene;
    WBGraphicsGroupContainerItem *mGroup;
    QList<QGraphicsItem*> mItems;
    bool mFirstRedo;
};
```

`getType` 方法返回该撤销命令的类型，以便在撤销系统中进行识别和处理。

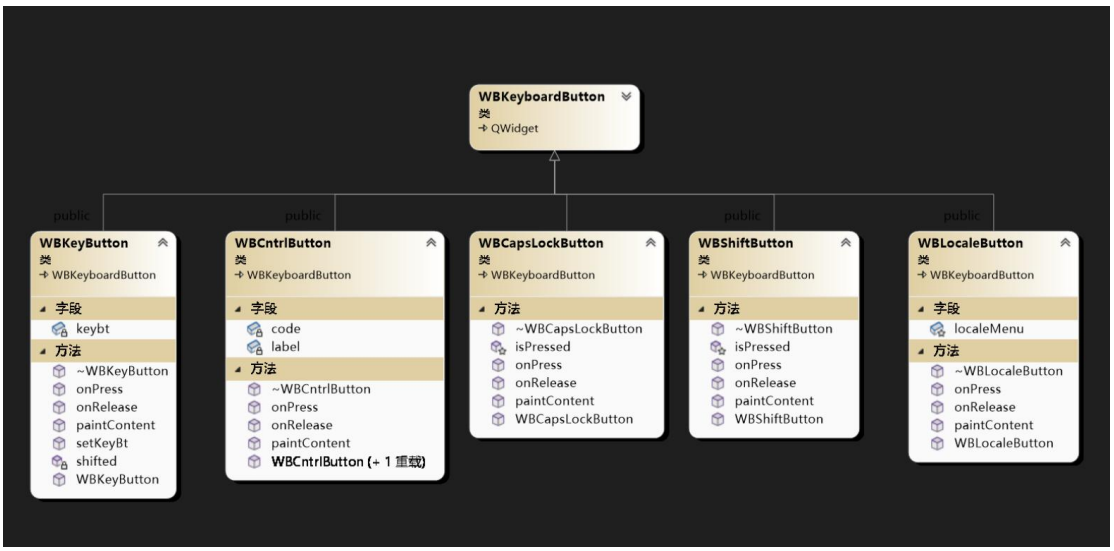
`undo` 方法执行撤销操作，可能会将分组操作还原，即将分组中的图形项恢复到分组前的状态。

`redo` 方法执行重做操作，重新应用分组操作。

WBGraphicsItemTransformUndoCommand：是一个特定的撤销命令类，用于处理图形项的变换操作（如位置、大小、旋转、Z 值等）的撤销和重做。

WBGraphicsTextItemUndoCommand：类是一个专门用于处理 `WBGraphicsTextItem` 文本项的撤销命令类。

键盘操作

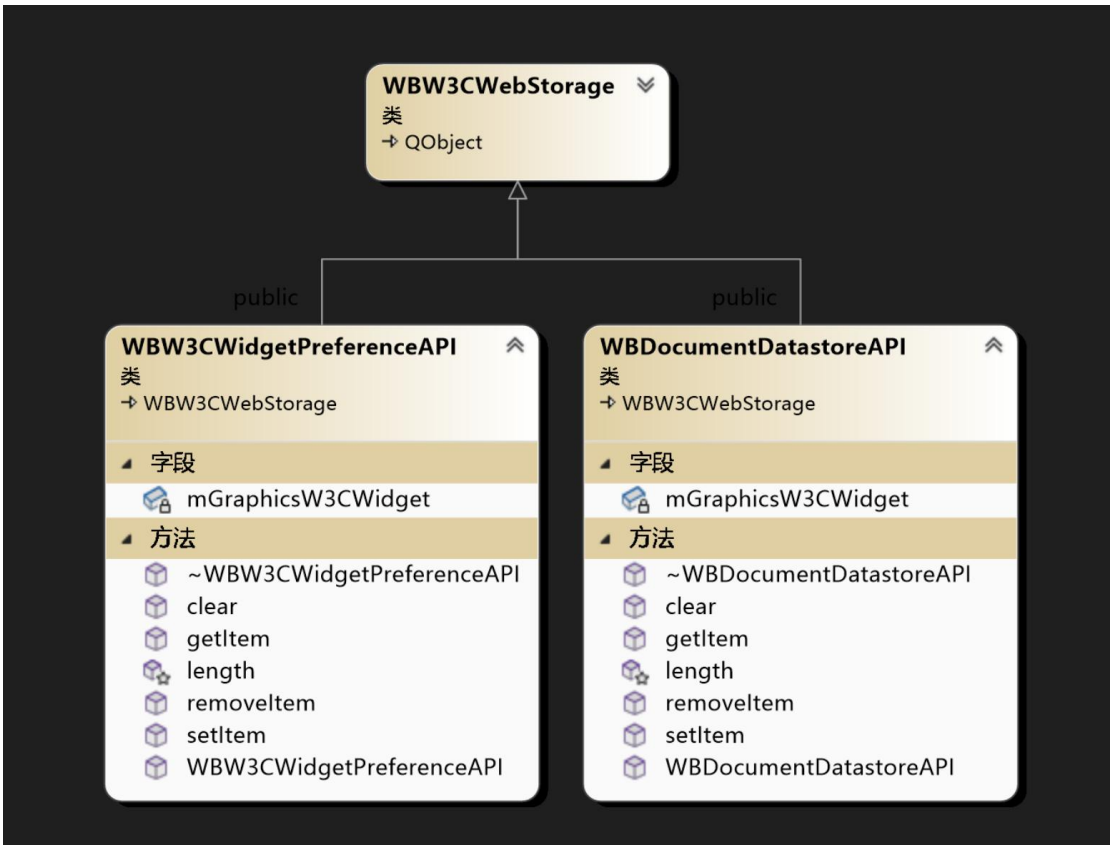


WBKeyboardButton 类及其子类（WBKeyButton、WBCntrlButton、WBCapsLockButton、WBShiftButton、WBLocaleButton）：这些类继承自 `WBKeyboardButton`，并分别实现了不同类型键盘按钮的特定行为和绘制方式。`WBKeyboardPalette` 可以作为一个虚拟键盘的界面组件，通过处理各种事件和信号，与应用的其他部分进行交互，实现键盘输入的功能。当用户点击某个键盘按钮时，会触发相应的 `onPress` 和 `onRelease` 方法来处理按键操作。

```
class WBKeyboardButton : public QWidget
{
    Q_OBJECT
public:
    WBKeyboardButton(WBKeyboardPalette* parent, QString contentImagePath);
    ~WBKeyboardButton();
protected:
    WBKeyboardPalette* m_parent;
    ContentImage *imgContent;
    QString m_contentImagePath;
    void paintEvent(QPaintEvent *event);
    virtual void    enterEvent ( QEvent * event );
    virtual void    leaveEvent ( QEvent * event );
    virtual void    mousePressEvent ( QMouseEvent * event );
    virtual void    mouseReleaseEvent ( QMouseEvent * event );
    virtual void onPress() = 0;
    virtual void onRelease() = 0;
    virtual void paintContent(QPainter& painter) = 0;
    virtual bool isPressed();
    WBKeyboardPalette* keyboard;
    void sendUnicodeSymbol(KEYCODE keycode);
    void sendControlSymbol(int nSymbol);
private:
    bool bFocused;
    bool bPressed;
};
```

偏好设置与文档操作接口

提供偏好设置接口，方便根据个人需求和使用习惯对写字板应用程序进行个性化配置，及其文档操作接口提供了一系列用于管理文档的操作入口和功能按钮。



WBW3CWebStorage：是一个抽象基类，用于定义与网络存储相关的基本接口。

```
class WBW3CWebStorage : public QObject
{
    Q_OBJECT
    Q_PROPERTY(int length READ length SCRIPTABLE true);
    public:
        WBW3CWebStorage(QObject *parent = 0)
        : QObject(parent){/* NOOP */}
        virtual ~WBW3CWebStorage(){/* NOOP */}
    public slots:
        virtual QString key(int index) = 0;
        virtual QString getItem(const QString& key) = 0;
        virtual void setItem(const QString& key, const QString& value) = 0;
        virtual void removeItem(const QString& key) = 0;
        virtual void clear() = 0;
    protected:
        virtual int length() = 0;
};
```

WBW3CWidgetPreferenceAPI：继承自 WBW3CWebStorage ，它与 WBGraphicsW3CWidgetItem 相关联，实现了获取键、获取项、设置项、移除项和清空等操作，用于处理该控件的偏好设置的存储和操作。

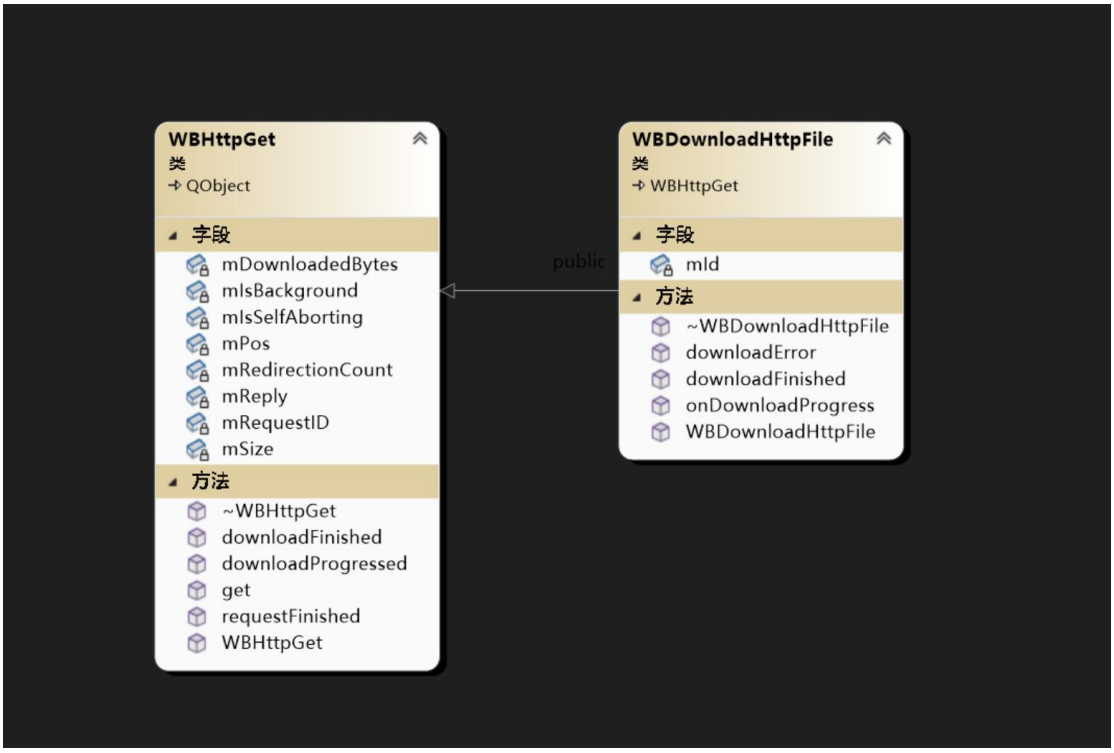
例如，在一个应用中，可以创建 WBW3CWidgetAPI 对象来获取和设置 WBGraphicsW3CWidgetItem 的各种属性，通过 preferences 获取偏好设置并进行操作。而 WBW3CWidgetPreferenceAPI 则专门负责处理偏好设置的具体存储和修改逻辑。

WBDocumentDatastoreAPI：继承自 `WBW3CWebStorage` ，专门处理与文档相关的数据存储的具体操作，如获取键、获取项、

设置项、移除项和清空等。
例如，在一个图形应用中，可以使用 `WBWidgetUniboardAPI` 对象来控制场景和部件的各种属性和操作，通过 `WBDatastoreAPI` 和 `WBDocumentDatastoreAPI` 来管理和操作与部件相关的数据存储。

网络模块

网络模块为用户精心打造了通过 HTTP 请求进行访问的功能特性。当用户发起 HTTP 请求时，能够与网络中的服务器及相关资源建立连接，实现数据的快速交互、文件的传输、信息的共享等操作，满足用户多样化的网络访问需求。



WBHttpGet：主要用于处理 HTTP 的 GET 请求，并提供了相关的功能和信号来处理请求过程中的各种事件和状态。

```
class WBHttpGet : public QObject
{
    Q_OBJECT
public:
    WBHttpGet(QObject* parent = 0);
    virtual ~WBHttpGet();
    QNetworkReply* get(QUrl pUrl, QPointF pPoint = QPointF(0, 0), QSize pSize = QSize(0, 0), bool isBackground = false);
signals:
    void downloadProgress(qint64 bytesReceived, qint64 bytesTotal);
    void downloadFinished(bool pSuccess, QUrl sourceUrl, QString pContentTypeHeader
            , QByteArray pData, QPointF pPos, QSize pSize, bool isBackground);
private slots:
    void readyRead();
    void requestFinished();
    void downloadProgressed(qint64 bytesReceived, qint64 bytesTotal);
private:
    QByteArray mDownloadedBytes;
    QNetworkReply* mReply;
    QPointF mPos;
```

```
    QSize mSize;
    bool mIsBackground;
    int mRequestID;
    int mRedirectionCount;
    bool mIsSelfAborting;
};
```

- `get(QUrl pUrl, QPointF pPoint = QPointF(0, 0), QSize pSize = QSize(0, 0), bool isBackground = false)` 方法用于发起一个 HTTP GET 请求，并可以指定一些额外的参数，如位置信息（`QPointF`）、尺寸信息（`QSize`）以及是否为后台请求。
- `downloadProgress(qint64 bytesReceived, qint64 bytesTotal)` 用于在下载过程中报告进度，提供已接收的字节数和总字节数。
- `downloadFinished(bool pSuccess, QUrl sourceUrl, QString pContentTypeHeader, QByteArray pData, QPointF pPos, QSize pSize, bool isBackground)` 用于在下载完成时发出信号，包含下载是否成功、源 URL、内容类型头、下载的数据、位置、尺寸以及是否为后台请求等信息。

项目流程

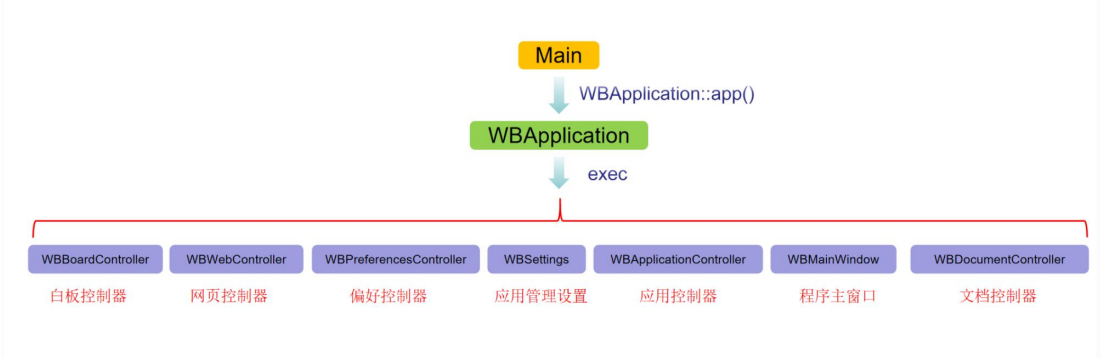
WBoard 项目使用典型的`MVC 设计模式`作为软件开发中的架构模式。
Model（模型）：
- 负责处理数据和业务逻辑。
- 包含数据的存储、检索、更新和验证等操作。

View（视图）：
- 负责数据的展示和用户交互界面。
- 它以用户友好的方式呈现 Model 中的数据。

Controller（控制器）：
- 作为 Model 和 View 之间的桥梁。
- 接收用户的输入请求，处理请求并调用相应的 Model 方法进行数据操作，然后选择合适的 View 来呈现结果。

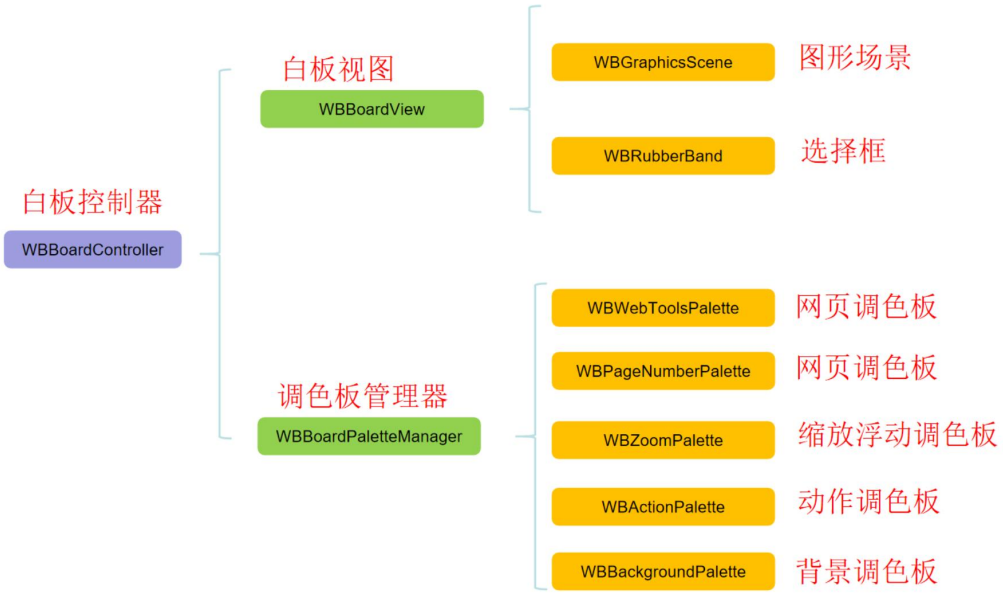
程序运行流程

1.项目入口点，从`Main 函数`进入之后，`创建 WBApplication 实例`，通过 WBApplication 实例对象初始化主程序。
2.WBApplication 初始化期间`会创建`WBMainWindow 主窗口开始进行子窗口布局`，`初始化视图(View)层`，再依次`创建控制层对象 (Controller)WBBoardController` 、 `WBWebController` 、 `WBPreferencesController` 、 `WBApplicationController` 、 `WBDocumentController`分别控制程序面板、网页面板、偏好设置面板、应用主程序、文档管理。
3.创建完成之后，会在`控制层中创建模型层(Model)来进行数据管理`，处理模型数据`通过代理模式更新视图`。如下图所示。

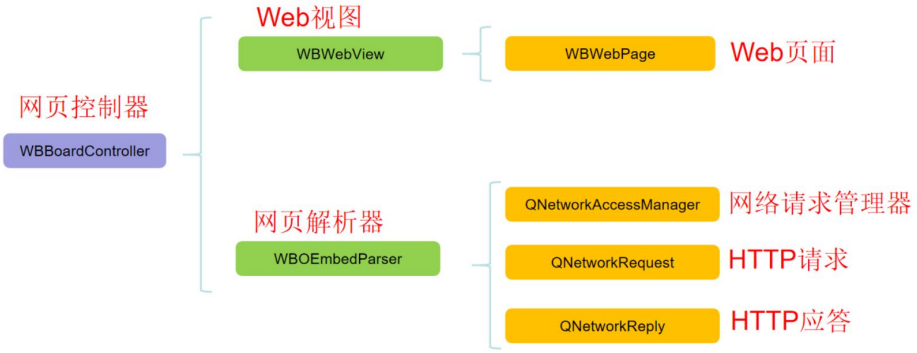


程序主控制器及数据模型

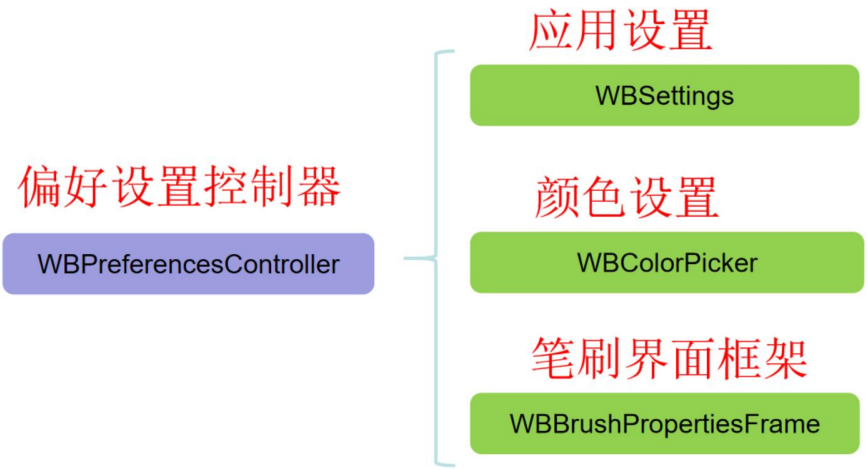
一：白板控制器



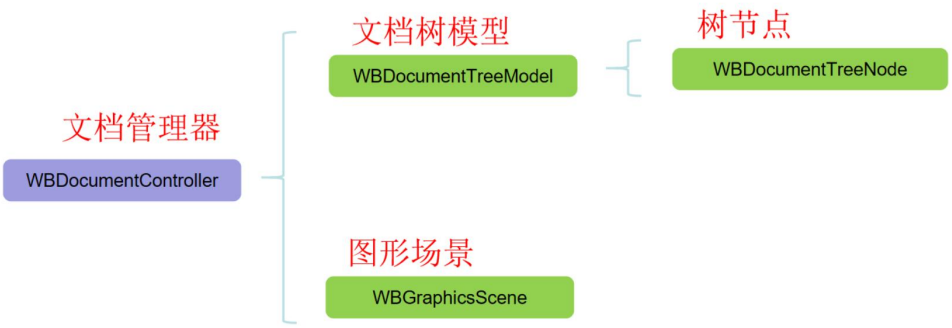
二：网页控制器



三：偏好设置控制器

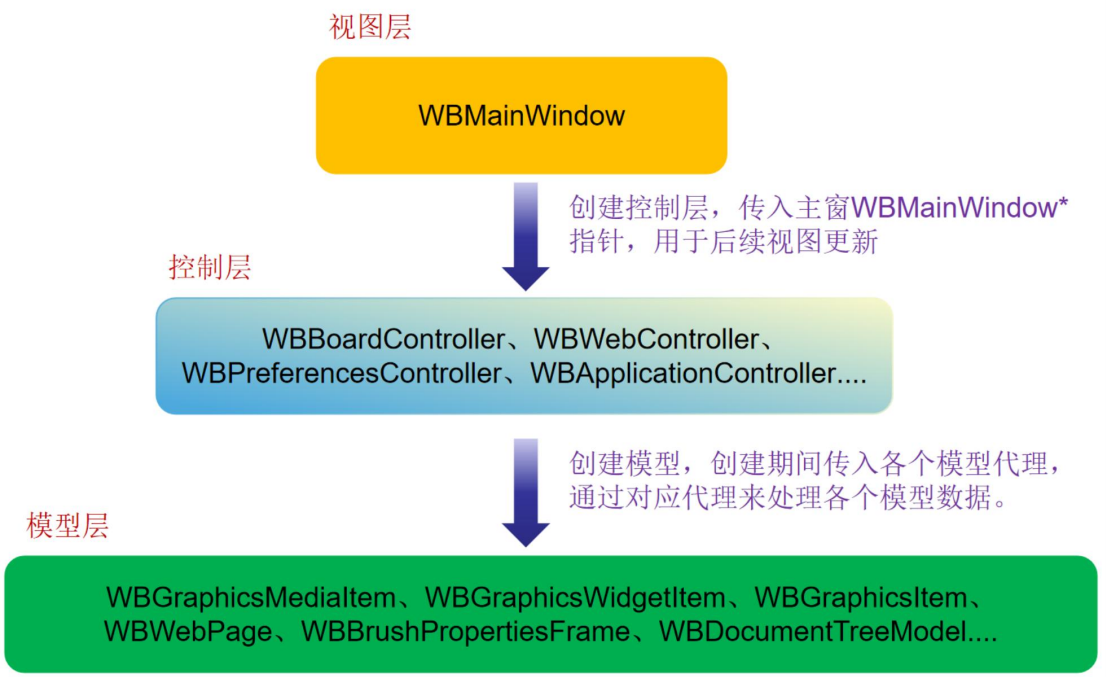


四：文档管理器



项目亮点

MVC 架构



程序怎么更新数据模型，直接调用？回调函数？观察者模式？消息队列？

答案是：WBoard 通过代理模型进行模型数据更新。通过代理更新数据模型具有以下几个显著的好处：

解耦和封装：

- 代理将数据更新的逻辑与数据模型的核心逻辑分离，减少了直接对模型的依赖，增强了系统的封装性。

增强安全性：

- 代理可以对更新操作进行额外的权限检查和访问控制，防止未经授权的访问和修改。

统一的更新接口：

- 为不同的数据源或更新方式提供了统一的接口，便于管理和维护。

可扩展性：

- 当需要添加新的更新规则、处理逻辑或支持新的数据来源时，只需修改代理的实现，而无需改动数据模型本身。