main函数

// Don't touch redirected streams here

// GetStdHandle should be called before AttachConsole

HANDLE hStdIn = GetStdHandle(STD\_INPUT\_HANDLE);

HANDLE hStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);

HANDLE hStdErr = GetStdHandle(STD\_ERROR\_HANDLE);

FILE \*fIn, \*fOut, \*fErr;

// Enable console output if available

if (AttachConsole(ATTACH\_PARENT\_PROCESS))

{

if (!hStdIn)

{

freopen\_s(&fIn, "CONIN$", "r", stdin);

}

if (!hStdOut)

{

freopen\_s(&fOut, "CONOUT$", "w", stdout);

}

if (!hStdErr)

{

freopen\_s(&fErr, "CONOUT$", "w", stderr);

}

}

// Make Qt's debug message handlers work

qInstallMessageHandler(consoleMessageHandler);

非规格化浮点数视为零

// Set denormal protection for this thread.

void inline disable\_denormals()

{

#ifdef \_\_SSE\_\_

/\* Setting DAZ might freeze systems not supporting it \*/

if (can\_we\_daz()) {

\_MM\_SET\_DENORMALS\_ZERO\_MODE( \_MM\_DENORMALS\_ZERO\_ON );

}

/\* FTZ flag \*/

\_MM\_SET\_FLUSH\_ZERO\_MODE( \_MM\_FLUSH\_ZERO\_ON );

#endif // \_\_SSE\_\_

}

在 Windows 平台上尝试将 LMMS 进程的优先级设置为高优先级。

你提供的代码片段是 LMMS `main.cpp` 文件中的一部分，它的作用是 \*\*在 Windows 平台上尝试将 LMMS 进程的优先级设置为高优先级\*\*。

\*\*代码解释:\*\*

- `#ifdef LMMS\_BUILD\_WIN32`: 这是一个预处理指令，表示这段代码只会在 Windows 平台上编译。

- `SetPriorityClass`: 这是一个 Windows API 函数，用于设置进程的优先级。

- `GetCurrentProcess()`: 这个函数返回当前进程的句柄。

- `HIGH\_PRIORITY\_CLASS`: 这是一个常量，表示高优先级。

- `if( !SetPriorityClass(...) )`: 如果 `SetPriorityClass` 函数调用失败，则执行 `if` 语句块内的代码。

\*\*功能:\*\*

这段代码尝试将 LMMS 进程的优先级设置为高优先级，以便为音频处理分配更多 CPU 资源，从而提高音频性能，减少卡顿和延迟。

\*\*注意:\*\*

- 这段代码只在 Windows 平台上有效。

- 设置进程优先级需要管理员权限，如果用户没有管理员权限，`SetPriorityClass` 函数调用可能会失败。

- 将 LMMS 进程的优先级设置为高优先级可能会影响系统中其他程序的性能。

\*\*总结:\*\*

这段代码是为了在 Windows 平台上提高 LMMS 的音频性能而添加的，它尝试将 LMMS 进程的优先级设置为高优先级，以便为音频处理分配更多 CPU 资源。

#ifdef LMMS\_BUILD\_WIN32

if( !SetPriorityClass( GetCurrentProcess(), HIGH\_PRIORITY\_CLASS ) )

{

printf( "Notice: could not set high priority.\n" );

}

#endif

音频引擎和界面成为两个模块。

# 注意

如果用低版本的QT打开工程，然后出错。重新克隆工程，并且重新cmake。

# ConfigManager

管理各种各样的配置。

# LMMS::GUI

## ModelView

MVP架构的P基类，把model和View链接起来

## AutomatableModelView

这个类更专注于处理可自动化的参数控制，比如在LMMS中的：

音量控制

效果器参数

合成器参数

等需要实时调节和自动化的场景

## AutomatableModelViewSlots

弹出一些设置。

execConnectionDialog()弹出MIDI控制器窗口。

## ControllerConnectionDialog

里面很多单独的小控件也是用了MVC架构。同时这些小控件也是继承了ModelView基类。

## AutomatableButton

## LcdWidget

Q\_PROPERTY( QColor textColor READ textColor WRITE setTextColor )

Q\_PROPERTY( QColor textShadowColor READ textShadowColor WRITE setTextShadowColor )

自定义控件，通过读取图片在pianterEvent里面绘制。

滚轮事件 void wheelEvent( QWheelEvent \* \_we ) override;

void contextMenuEvent( QContextMenuEvent \* \_me ) override;

通过设置模型，视图，LcdWidget作为视图，当数值有更新时，更新到模型引擎里。

m\_tempoSpinBox->setModel( &m\_song->m\_tempoModel );

## LcdSpinBox



基类是LcdWidget

**像LcdSpinBox这样的类，在界面中就是一个调节声音等属性大小的控件，为什么要搞这么复杂，要继承IntModelView，做MVP架构，直接在这个LcdSpinBox增加一个int property作为属性不行吗。**

这是一个很好的问题。让我来解释为什么LMMS选择使用MVP架构而不是直接在控件中存储属性：

1. \*\*数据与视图分离\*\*

- MVP架构将数据(Model)和视图(View)分离，这样同一个数据可以被多个不同的视图控件共享和显示

- 例如，同一个音量值可能需要在LcdSpinBox、滑块、自动化轨道等多个地方显示和修改

- 如果直接在控件中存储属性，就无法实现这种数据共享

2. \*\*自动化支持\*\*

- LMMS是一个音乐制作软件，需要支持自动化功能

- 通过Model类，可以方便地实现自动化控制

- 例如`IntModel`可以连接到自动化轨道，实现参数随时间变化

- 如果直接在控件中存储属性，就无法实现这种自动化功能

3. \*\*撤销/重做支持\*\*

- 通过Model类，可以方便地实现撤销/重做功能

- 所有的数据修改都通过Model进行，可以记录修改历史

- 如果直接在控件中存储属性，就无法实现这种撤销/重做功能

4. \*\*数据持久化\*\*

- Model类可以方便地实现数据的保存和加载

- 项目文件只需要保存Model的数据，而不需要关心UI状态

- 如果直接在控件中存储属性，数据持久化会变得复杂

5. \*\*信号/槽机制\*\*

- Qt的信号/槽机制在MVP架构中发挥更大作用

- Model的数据变化可以通知所有相关的View更新

- View的用户输入可以通知Model更新数据

- 如果直接在控件中存储属性，这种双向通信会变得复杂

6. \*\*测试和维护\*\*

- MVP架构使代码更容易测试和维护

- Model和View可以独立测试

- 修改UI不会影响数据逻辑，修改数据逻辑不会影响UI

- 如果直接在控件中存储属性，测试和维护会变得困难

所以虽然看起来使用MVP架构增加了代码复杂度，但对于LMMS这样的专业音乐制作软件来说，这种架构带来的好处远大于其复杂性。它提供了更好的可扩展性、可维护性和功能支持。

## MeterDialog

同LcdWidget一样，也是设置模型来设置相应数据。



## TimeDisplayWidget

时间显示。

通过单例调用，更新时间

void TimeDisplayWidget::updateTime()

## AutomatableSlider

音量滑块（工具栏）。看到AutomatableSlider。

也是通过单例更新波形图。

同样使用MVC架构，去更新model。

在 Qt 框架中，MVC 的角色常常并不是完全分开的，而是通过信号与槽机制将控制逻辑嵌入到视图中。这种结构在 GUI 框架中较为常见。

addDefaultActions是一个公有的右击菜单。

widget()->setCursor( QCursor( embed::getIconPixmap( "hand" ), 3, 3 ) );可以设置经过时变成手型。

AutomatableModelViewSlots

弹出一些设置。

execConnectionDialog()弹出MIDI控制器。

弹出对话框选择项

// TODO[pg]: Display a dialog with list of controllers currently in the song

// in addition to any system MIDI controllers

AutomatableModel\* m = m\_amv->modelUntyped();

m->displayName();

gui::ControllerConnectionDialog d( getGUI()->mainWindow(), m );

if( d.exec() == 1 )

{

// Actually chose something

if( d.chosenController() )

{

// Update

if( m->controllerConnection() )

{

m->controllerConnection()->setController( d.chosenController() );

}

// New

else

{

auto cc = new ControllerConnection(d.chosenController());

m->setControllerConnection( cc );

//cc->setTargetName( m->displayName() );

}

}

// no controller, so delete existing connection

else

{

removeConnection();

}

}

## Oscilloscope

音乐波形图。

Q\_PROPERTY( QColor leftChannelColor READ leftChannelColor WRITE setLeftChannelColor )

Q\_PROPERTY( QColor rightChannelColor READ rightChannelColor WRITE setRightChannelColor )

Q\_PROPERTY( QColor otherChannelsColor READ otherChannelsColor WRITE setOtherChannelsColor )

Q\_PROPERTY( QColor clippingColor READ clippingColor WRITE setClippingColor )

自定义控件画波形图。

void Oscilloscope::paintEvent( QPaintEvent \* )

{

QPainter p( this );

p.drawPixmap( 0, 0, m\_background );

if( m\_active && !Engine::getSong()->isExporting() )

{

AudioEngine const \* audioEngine = Engine::audioEngine();

float masterOutput = audioEngine->masterGain();

const fpp\_t frames = audioEngine->framesPerPeriod();

SampleFrame peakValues = getAbsPeakValues(m\_buffer, frames);

auto const leftChannelClips = clips(peakValues.left() \* masterOutput);

auto const rightChannelClips = clips(peakValues.right() \* masterOutput);

p.setRenderHint( QPainter::Antialiasing );

// now draw all that stuff

int w = width() - 4;

const qreal xd = static\_cast<qreal>(w) / frames;

const qreal half\_h = -(height() - 6) / 3.0 \* static\_cast<qreal>(masterOutput) - 1;

int x\_base = 2;

const qreal y\_base = height() / 2 - 0.5;

qreal const width = 0.7;

for( ch\_cnt\_t ch = 0; ch < DEFAULT\_CHANNELS; ++ch )

{

QColor color = ch == 0 ? (leftChannelClips ? clippingColor() : leftChannelColor()) : // Check left channel

ch == 1 ? (rightChannelClips ? clippingColor() : rightChannelColor()) : // Check right channel

otherChannelsColor(); // Any other channel

p.setPen(QPen(color, width));

for (auto frame = std::size\_t{0}; frame < frames; ++frame)

{

sample\_t const clippedSample = AudioEngine::clip(m\_buffer[frame][ch]);

m\_points[frame] = QPointF(

x\_base + static\_cast<qreal>(frame) \* xd,

y\_base + ( static\_cast<qreal>(clippedSample) \* half\_h ) );

}

p.drawPolyline( m\_points, frames );

}

}

else

{

p.setPen( QColor( 192, 192, 192 ) );

p.setFont(adjustedToPixelSize(p.font(), 10));

p.drawText( 6, height()-5, tr( "Click to enable" ) );

}

}

## SideBarWidget

## PluginBrowser继承自SideBarWidget

左侧展开的树形乐器选择。（第一个）

使用QSplitter进行伸缩布局。

除了第一个乐器选择，都是使用PluginBrowser定义。

## FileBrowser继承自SideBarWidget

左侧展开的文件选择。

这个是双击加载音乐文件。

void FileBrowserTreeWidget::activateListItem(QTreeWidgetItem \* item,

int column )

mousePressEvent左侧树形控件的类。

handleFile

根据不同的Item有不同的动作。

reloadTree

读取文件

## PluginDescWidget

树形乐器选择的控件。

摁下拖动选择控件。

void PluginDescWidget::mousePressEvent( QMouseEvent \* \_me )

{

Engine::setDndPluginKey(&m\_pluginKey);

if ( \_me->button() == Qt::LeftButton )

{

new StringPairDrag("instrument",

QString::fromUtf8(m\_pluginKey.desc->name), m\_logo, this);

leaveEvent( \_me );

}

}

右击事件

void PluginDescWidget::contextMenuEvent(QContextMenuEvent\* e)

{

QMenu contextMenu(this);

contextMenu.addAction(

tr("Send to new instrument track"),

[=]{ openInNewInstrumentTrack(m\_pluginKey.desc->name); }

);

contextMenu.exec(e->globalPos());

}

## FileBrowserTreeWidget

mousePressEvent

预览插件声音

previewFileItem

会有一个TextFloat窗口打开，但是可能是因为加载太快没看到。

processEvents让QT程序等待

## Plugin

插件类，里面嵌套子插件类。

## SideBar

左侧的工具栏

## SideBarButton

左侧工具栏的按钮

自定义样式比QStyle更简单的办法。

void paintEvent( QPaintEvent \* ) override

{

QStylePainter p( this );

QStyleOptionToolButton opt;

initStyleOption( &opt );

if( orientation() == Qt::Vertical )

{

const QSize s = sizeHint();

p.rotate( 270 );

p.translate( -s.height(), 0 );

opt.rect = QRect( 0, 0, s.height(), s.width() );

}

p.drawComplexControl( QStyle::CC\_ToolButton, opt );

}

## GuiApplication

继承QObject，所有界面类成员都放在这。没用智能指针。包含所有界面类的初始化。

lmms的单例模式没有做线程保护。

## LmmsStyle

这几个样式设置探索一下用来干什么。

auto lmmsstyle = new LmmsStyle();

QApplication::setStyle(lmmsstyle);

auto lmmspal = new LmmsPalette(nullptr, lmmsstyle);

auto lpal = new QPalette(lmmspal->palette());

QApplication::setPalette( \*lpal );

LmmsStyle::s\_palette = lpal;

## ToolButton继承QToolButton

connect可以传指针代表槽函数

ToolButton::ToolButton( const QPixmap & \_pixmap, const QString & \_tooltip,

QObject \* \_receiver, const char \* \_slot,

QWidget \* \_parent ) :

QToolButton( \_parent )

{

setAutoFillBackground( false );

if( \_receiver != nullptr && \_slot != nullptr )

{

connect( this, SIGNAL(clicked()), \_receiver, \_slot );

}

setToolTip(\_tooltip);

setIcon( \_pixmap );

}

connect( this, SIGNAL(clicked()), \_receiver, \_slot );

## MainWindow

## TrackContainer

## Song

同样使用MVC架构，去更新model。

继承JournallingObject序列化

loadProject

加载工程

clearProject

关闭工程

## TrackContainerView

Public QWidget、public ModelView,public JournallingObject,public SerializingObjectHook

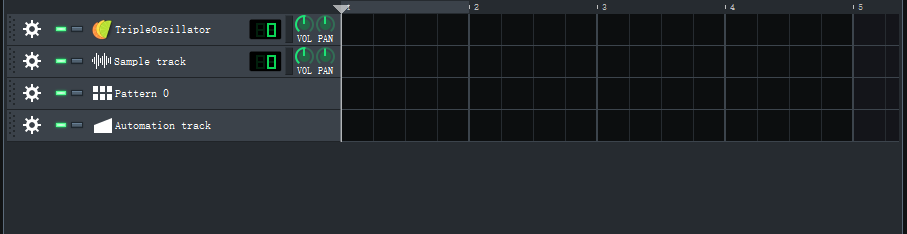
鼠标拖控件到View里面。

void dropEvent( QDropEvent \* \_de ) override;

void dragEnterEvent( QDragEnterEvent \* \_dee ) override;

## SongEditor

这个窗口。



**m\_tempoSpinBox是SongEditor的成员，为什么在SongEditor初始化并添加到主窗口的工具栏。而且SongEditor并没有使用到m\_tempoSpinBox好像（MeterDialog等工具栏的控件同理了）**

让我分析一下`m\_tempoSpinBox`的设计：

1. \*\*位置问题\*\*

- `m\_tempoSpinBox`虽然是`SongEditor`的成员，但被添加到主窗口工具栏

- 这是因为速度(tempo)是一个全局参数，需要随时可以访问和修改

- 放在主窗口工具栏比放在`SongEditor`窗口内更方便用户操作

2. \*\*使用情况\*\*

- 虽然`SongEditor`类中没有直接使用`m\_tempoSpinBox`

- 但它通过`m\_song->m\_tempoModel`间接使用：

```cpp

m\_tempoSpinBox->setModel( &m\_song->m\_tempoModel );

```

- 这样当用户修改`m\_tempoSpinBox`的值时，会直接修改`Song`的tempo

3. \*\*设计考虑\*\*

- 这种设计体现了MVP架构的思想：

- Model: `m\_song->m\_tempoModel` (数据)

- View: `m\_tempoSpinBox` (显示)

- Presenter: `SongEditor` (控制)

- 虽然View放在主窗口，但数据和控制逻辑仍在`SongEditor`

4. \*\*代码组织\*\*

- 虽然`m\_tempoSpinBox`的UI位置在主窗口

- 但它的生命周期管理仍在`SongEditor`中

- 这样可以保持相关代码的集中管理

5. \*\*实际用途\*\*

- `m\_tempoSpinBox`用于：

- 显示当前歌曲速度

- 允许用户修改速度

- 与其他速度相关功能联动

6. \*\*改进建议\*\*

- 这种设计确实有些混乱

- 更好的做法可能是：

- 将`m\_tempoSpinBox`移到主窗口类

- 或者创建一个专门的`TempoController`类

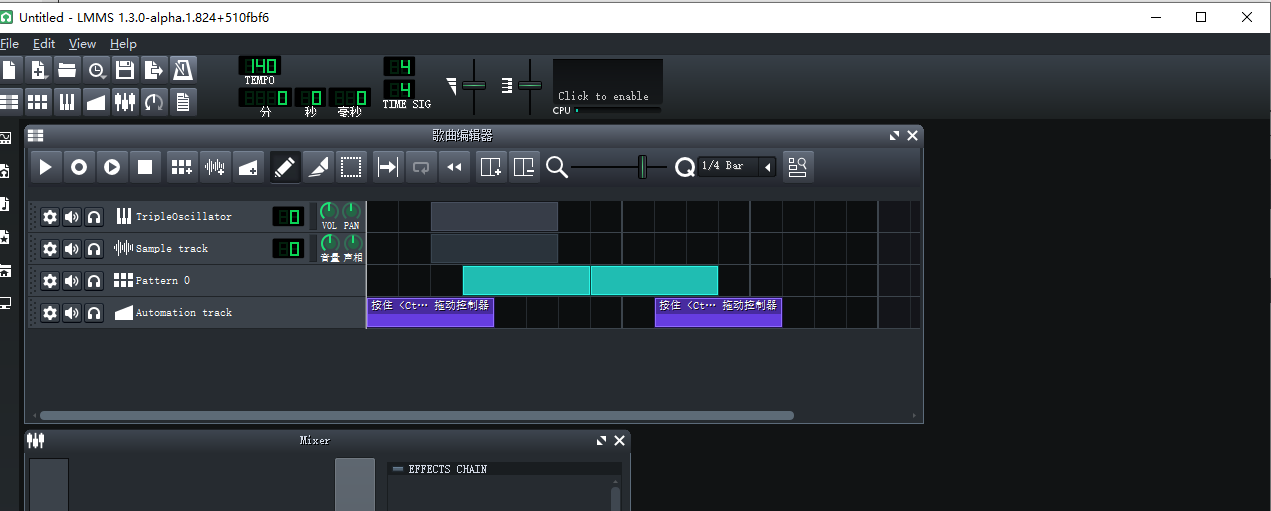
- 或者使用信号槽机制解耦

所以虽然`m\_tempoSpinBox`的当前设计有些不太合理，但它是基于实际使用需求和历史原因形成的。这种设计虽然不够优雅，但在功能上是有效的。

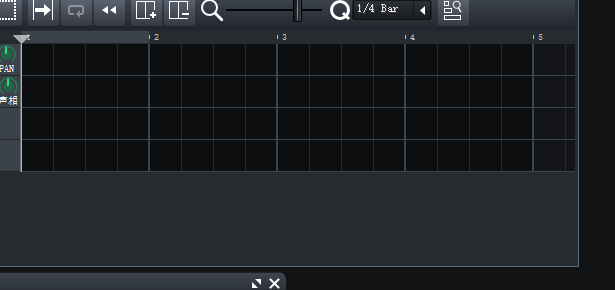
## TimeLineWidget

图块类上面的时间标线。不属于TrackView的一部分。

无painterEvent



有painterEvent



所以只是上面有数字的一部分。

addToolButtons解耦

1. 设计上的解耦与职责分离

TimeLineWidget 之所以有 addToolButtons，主要是为了解耦工具栏和时间线控件的实现。虽然 \_tool\_bar 物理上属于外部窗口（比如主窗口或 SongEditor），但哪些按钮、按钮的行为、按钮和时间线的交互，这些细节只有 TimeLineWidget 最清楚。

TimeLineWidget 负责告诉外部“我需要哪些按钮、这些按钮要怎么响应”。

外部窗口 只负责把工具栏传进来，具体加什么按钮、按钮怎么连槽，由 TimeLineWidget 决定。

这样做的好处是：如果以后 TimeLineWidget 需要增加、删除或修改按钮，只需要改 TimeLineWidget 的代码，不用动外部窗口的逻辑。

2. 为什么不直接在 SongEditor 里做？

你说得没错，理论上也可以在 SongEditor 里直接添加按钮并连接到 TimeLineWidget 的槽。但这样会有几个问题：

SongEditor 必须了解 TimeLineWidget 的内部细节（比如有哪些槽、按钮的用途），这会导致强耦合。

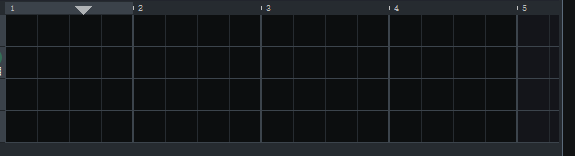
如果 TimeLineWidget 的按钮需求变了，SongEditor 也要跟着改，维护成本高。

违反了“高内聚、低耦合”的设计原则。

## PositionLine

播放位置标线。

没有painterEvent



## Editor

Editor 类是一个基类（superclass），专门为 LMMS 中带有工具栏的编辑器窗口设计。它是一个抽象类，继承自 QMainWindow，为多个具体的编辑器（如 Song Editor、Automation Editor、Beat+Bassline Editor 和 Piano Roll Editor）提供了通用功能和结构。简单来说，它的作用是：

提供统一的工具栏管理：通过封装工具栏相关的功能（如播放、停止、录制等按钮），让子类可以专注于实现各自独特的编辑逻辑，而无需重复编写工具栏代码。

定义通用的播放控制接口：包括播放、暂停、停止等操作的虚函数，子类可以根据自己的需求重写这些函数。

事件处理和信号机制：通过信号和槽机制，处理用户交互（如按下空格键播放/停止），并允许与 LMMS 的其他部分通信。

### Lambda表达式的高级应用。

auto addButton = [this](QAction\* action, QString objectName) {

m\_toolBar->addAction(action);

m\_toolBar->widgetForAction(action)->setObjectName(objectName);

};

// Set up play and record actions

m\_playAction = new QAction(embed::getIconPixmap("play"), tr("Play (Space)"), this);

m\_stopAction = new QAction(embed::getIconPixmap("stop"), tr("Stop (Space)"), this);

m\_recordAction = new QAction(embed::getIconPixmap("record"), tr("Record"), this);

m\_recordAccompanyAction = new QAction(embed::getIconPixmap("record\_accompany"), tr("Record while playing"), this);

m\_toggleStepRecordingAction = new QAction(embed::getIconPixmap("record\_step\_off"), tr("Toggle Step Recording"), this);

// Set up connections

connect(m\_playAction, SIGNAL(triggered()), this, SLOT(play()));

connect(m\_recordAction, SIGNAL(triggered()), this, SLOT(record()));

connect(m\_recordAccompanyAction, SIGNAL(triggered()), this, SLOT(recordAccompany()));

connect(m\_toggleStepRecordingAction, SIGNAL(triggered()), this, SLOT(toggleStepRecording()));

connect(m\_stopAction, SIGNAL(triggered()), this, SLOT(stop()));

new QShortcut(Qt::Key\_Space, this, SLOT(togglePlayStop()));

new QShortcut(QKeySequence(Qt::SHIFT + Qt::Key\_Space), this, SLOT(togglePause()));

new QShortcut(QKeySequence(Qt::SHIFT + Qt::Key\_F11), this, SLOT(toggleMaximize()));

// Add actions to toolbar

addButton(m\_playAction, "playButton");

if (record)

{

addButton(m\_recordAction, "recordButton");

addButton(m\_recordAccompanyAction, "recordAccompanyButton");

}

if(stepRecord)

{

addButton(m\_toggleStepRecordingAction, "stepRecordButton");

}

addButton(m\_stopAction, "stopButton");

## SongEditorWindow

SongEditorWindow

标题栏名称为"Song-Editor"的窗口的类。

上面是QAction按钮。

下面是 SongEditor\* m\_editor;

## 控件

### NStateButton

自定义的带状态、图片的toolbutton。

## TextFloat

拉动音量条时，显示的文本。

# 功能类

## DataFile

写xml的东西。

由于使用序列化的方式，所以文本不能直接打开。

## JournallingObject

## SerializingObjectHook

## InstrumentLoaderThread

音频加载的现场子类

# Engine

# LMMS模块

## Model

为继承类提供基础设施来管理名称、父子关系和数据变化等功能。它为 LMMS 中的各种对象（如音频、乐器、效果等）提供了一个统一的框架，并且通过信号机制支持响应数据变化。

## AutomatableModel

主要用于管理和控制可以自动化的模型对象

## TypedAutomatableModel（模板类）

包括三个实例化

FloatModel

IntModel

BoolModel

## ThreadableJob

线程工作基类

## PresetPreviewPlayHandle

同样继承自ThreadableJob

## SamplePlayHandle

样品处理句柄。