



# NUI Engine

## 入门篇



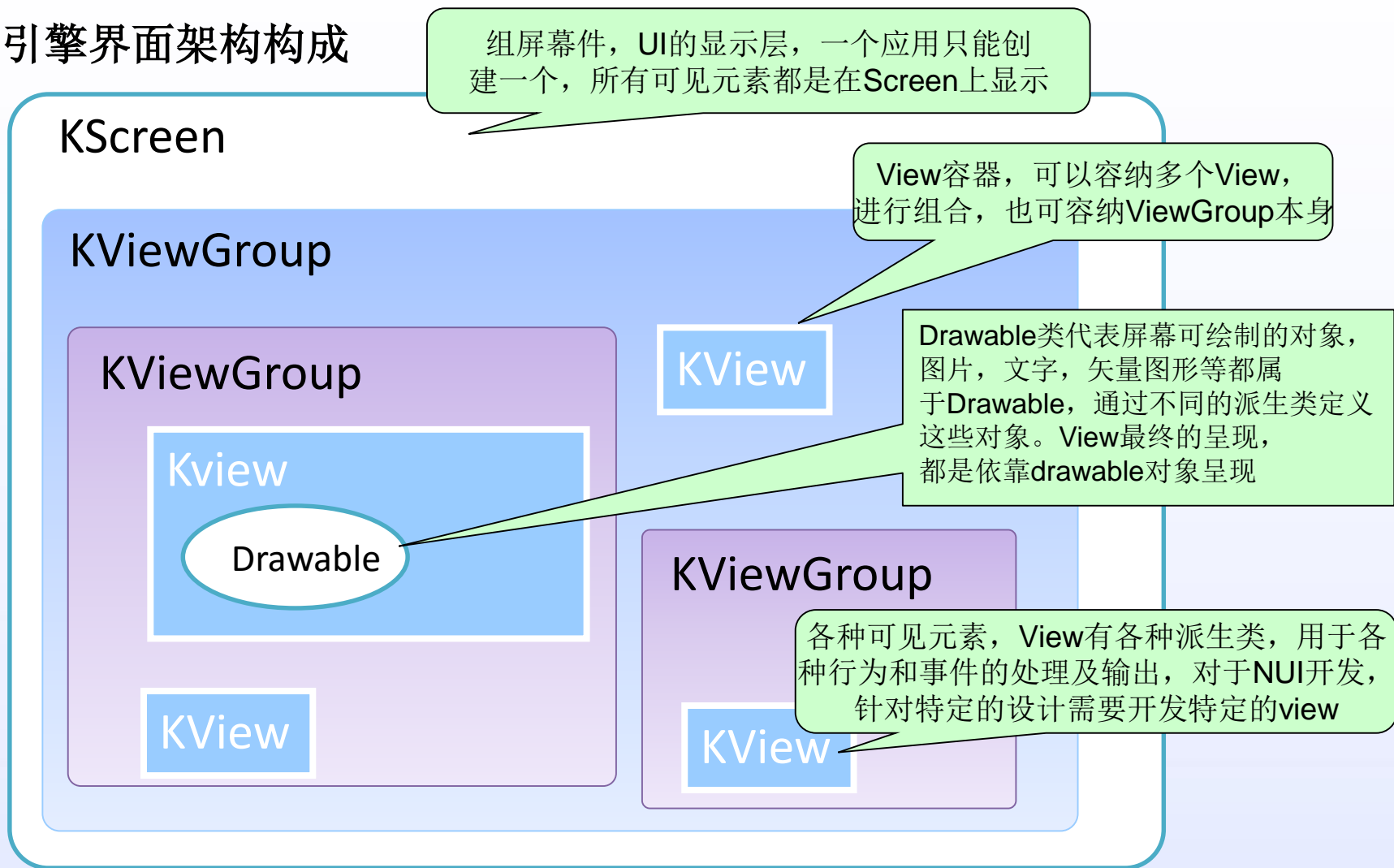
# 课程目标

- 1.掌握NUI引擎使用基本流程
- 2.能够使用基本控件制作简单界面
- 3.实现简单动画效果



# 基本使用流程介绍

## 引擎界面架构构成





font、string和surface  
管理类

辅助类  
(KFontManager  
KStringManager  
KSurfaceManager)

封装系统级消息处理及执行  
核心渲染架构，界面上屏的  
跨平台处理

屏幕抽象层  
(KScreen)

动画控制层  
(CAnimationThread/NUIAnimation)

通过View对象及其派生  
类封装具体的UI对象，  
如按钮等

UI层  
(KView与KViewGroup)

封装动画线程及数据交  
换结构，可方便实现界  
面动画

绘制对象封装层  
(drawable)

通过Drawable对象及其派  
生类封装具体的显示对象，  
如图片，文字等

渲染引擎抽象层  
(surface架构)

渲染抽象层封装底层绘制引  
擎的接口，保证绘制引擎可  
替换性

绘制引擎  
(Skia)

使用免费的第三方开源绘  
制引擎Skia（绘制引擎可  
替换）



## 创建NUI引擎实例

```
KNUIInitPara para;  
para.m_w = g_iScreenWidth;  
para.m_h = g_iScreenHeight;  
para.m_wnd = hWnd;  
para.m_render_mode = SCREEN_RENDER_MULTI_THREAD;  
para.m_device_type = WIN_DEVICE;  
g_nui_instance.create(para );
```

通过NUI实例设置渲染标志位（进入程序第一次Draw）

```
g_nui_instance.getScreen()->SetRenderFlag(TRUE);
```

创建主窗口对象，大小与Screen大小一致

```
KMainWindow_PTR main_view = KMainWindow_PTR(new KMainWindow());  
main_view->Create(0, 0, g_iScreenWidth, g_iScreenHeight);
```



通过实例获取屏幕**Screen**，将主窗口添加到**Screen**。主窗口可以添加子窗口，也可以添加子控件，一般窗口会用一个**Init()**函数来组合窗口和控件。需要刷新主窗口，因为引擎是渲染标志位置为**true**时才会需要刷新。

```
g_nui_instance.getScreen()->AddView(main_view);  
main_view->Init();  
main_view->InvalidateView();
```

## 主窗口添加控件及控件的初始化和属性设置

```
KTextView* view = new KTextView();
```

错误，绝不允许


**智能指针**采用引用计数的方法将一个计数器与类指向的对象相关联，引用计数跟踪共有多少个类对象共享同一指针。

### 基本用法

```
class KView
```

```
{.....}
```

```
typedef boost::shared_ptr<KView> KView_PTR;
```



智能指针-> 和.

```
KView_PTR pView = KView_PTR(new KView);
```

```
If(pView) //判断是否为空
```

```
pView.reset(); //清空该智能指针对象的内部指针，菜鸟不要用
```

```
pView.get(); //取指针，不建议使用
```

```
If(pView == NULL) 错误
```

```
delete pView 错误
```

```
m_hello_world_text = KTextView_PTR(new KTextView());
```

```
m_hello_world_text->Create(0, 0, 120, 120 );
```

```
AddView(m_hello_world_text);
```

初始化，添加到父窗口

```
m_hello_world_text->SetFontSize(20);
```

```
m_hello_world_text->SetTextColor(RE_ColorRED);
```

```
m_hello_world_text->SetFont(GetFontManagerSingleton()->GetFontFromName("Microsoft YaHei"));
```

```
m_hello_world_text->setTextAlign(REPaint::kCenter_Align);
```

```
m_hello_world_text->SetText( _T("Hello World"));
```

```
m_hello_world_text->SetViewChangeType(KVIEW_LRMethod_Parent_Left, KVIEW_LRMethod_Parent_Left,  
KVIEW_BTMethod_Parent_Top, KVIEW_BTMethod_Parent_Top);
```

属性设置



# 智能指针 $\neq$ 指针

智能指针对象内部访问自己的智能指针

`shared_from_this()`

```
void KWindowMove::OnMove(kn_int x, kn_int y, KMessageMouse* pMsg)
{
    if(m_b_mouse_picked)
    {
        CPropertyPos* p = new CPropertyPos( shared_from_this(), x -
            m_rect.width() / 2, y - m_rect.height() / 2);
        GetScreen()->addProperty(p);
        UpdateUI();
    }
}
```





## 主窗口添加窗口及窗口的初始化和属性设置

```
m_hello_world_group = KViewGroup_PTR(new KViewGroup());  
m_hello_world_group->Create(0, 0, m_rect.width(), m_rect.height()*3/4);  
AddView(m_hello_world_group);  
m_hello_world_group->SetShow(false);  
m_hello_world_group->SetViewChangeType(KVIEW_LRMethod_Parent_Left,  
KVIEW_LRMethod_Parent_Right, KVIEW_BTMethod_Parent_Top, KVIEW_BTMethod_Parent_Bottom);  
KImageDrawable_PTR bk_drawable = KImageDrawable_PTR(new  
KImageDrawable(_T("./resource/hello_world.png")));  
bk_drawable->setRect(0, 0, view_group->GetRect().width(), view_group->GetRect().height());  
m_hello_world_group->addDrawable(bk_drawable);  
bk_drawable->SetViewChangeType(KVIEW_LRMethod_Parent_Left, KVIEW_LRMethod_Parent_Right,  
KVIEW_BTMethod_Parent_Top, KVIEW_BTMethod_Parent_Bottom);
```



## 回调处理、信号槽的使用

NUI引擎对于控件事件回调处理使用了可以对事件实现动态的绑定处理的信号槽机制。老的事件处理机制，对于控件的事件要绑定到行为上，需要使用固定的回调函数（胶水代码），用于连接事件消息和具体的事件处理。

```
void KDlgMainMenu::OnBtnClock(void* p, void*
pParam)
{
    KDlgMainMenu* pThis = (KDlgMainMenu*)p;
    pThis->OnBtnClock(pParam);
}
void KDlgMainMenu::OnBtnClock(void *pParam)
{
    Effect(enu_normal);
}
```

m\_hello\_world\_text的点击响应事件绑定

及实现如下，这样当有点击事件时，就会去处理onHelloWorldClick函数。

```
m_hello_world_text->m_clicked_signal.connect(this,
&KMainWindow::onHelloWorldClick);
void KMainWindow::onHelloWorldClick(KView_PTR)
{
    m_hello_world_group->SetShow(true);
}
```

胶水代码



信号槽机制，通过connect函数可以在运行时指定某一事件的处理函数。

注意：信号的参数必须与绑定的处理函数的参数保持一致。

有时候引擎提供的信号并不能完全满足我们的功能需求，这时候就需要自定义信号去实现，例如：

public:

```
sigslot::signal1<bool> m_timeline_change_signal;
```

signal1: 一个参数  
依次signal2: 两个参数

处理函数的绑定类似，如下：

```
m_time_bar->m_timeline_change_signal.connect(this,  
&CTimeBarPage::OnTimelineChange);
```

当我们需要消息触发时，通过信号的emit函数（可选择是否传递参数）触发信号槽，这样处理函数就会得到响应，例如：

```
m_timeline_change_signal.emit(false);
```

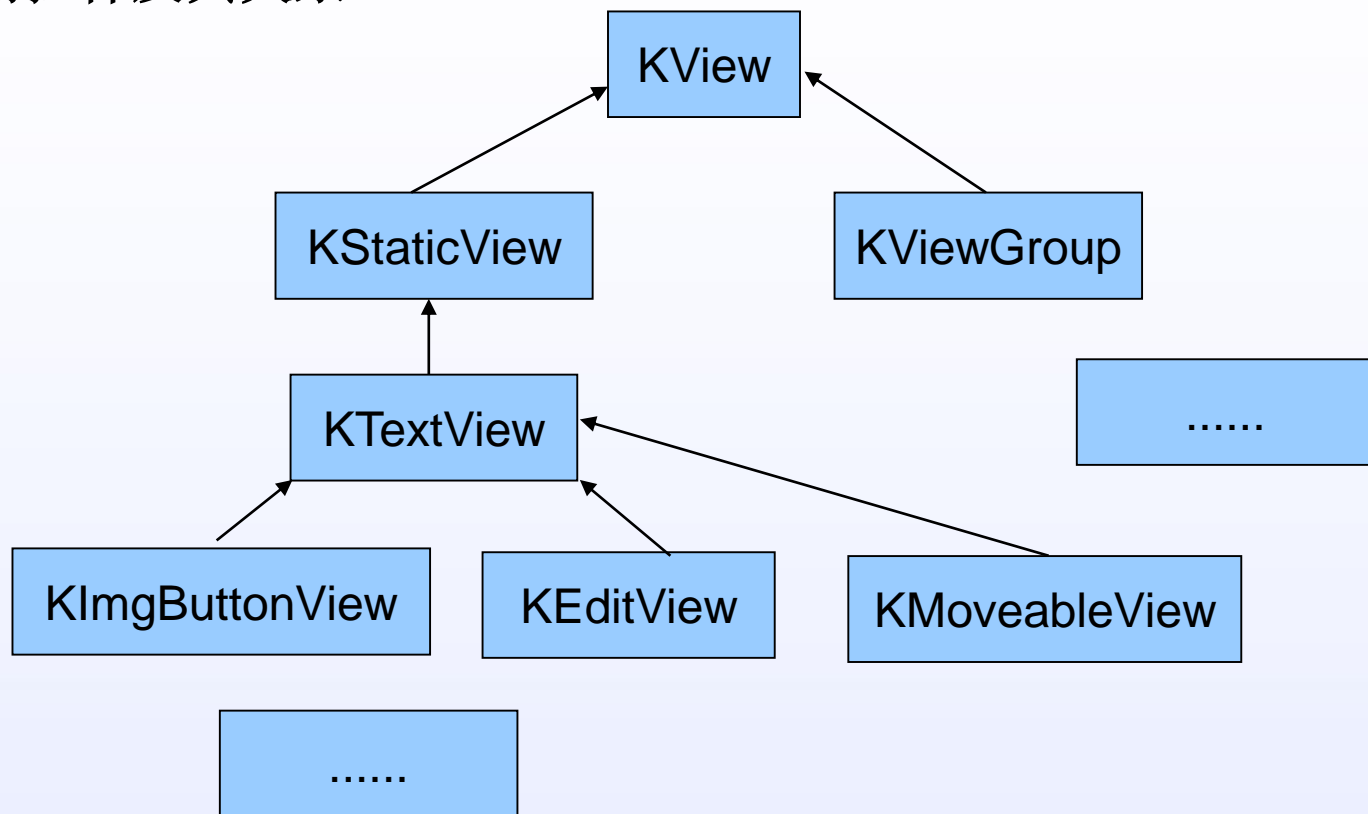
这种机制，可以在运行时刻定义甚至改变信号的处理函数，基于这种特性，可实现纯粹的应用MVC架构（界面和实现分离）

**不用为事件编写事件函数，直接绑定到功能处理函数**



# 能够使用基本控件制作简单界面

## 常用控件及其关系



**注意：不能在构造函数中初始化成员控件，准确地说不能调用AddView**



## KView与KViewGroup

KViewGroup中有m\_lst\_view (LSTVIEW: vector<KView\_PTR>), 因此KViewGroup可以容纳多个View和ViewGroup, 将它们进行组合。

Question1: 如何在ViewGroup添加KView或ViewGroup呢?

通过AddView()函数添加到m\_lst\_view, 示例:

```
m_bk = KStaticView_PTR(new KStaticView());  
m_bk->Create(0, 0, m_rect.width, m_rect.height());  
AddView(m_bk);
```

ViewGroup添加View

```
KViewGroup_PTR edit_view_group = KViewGroup_PTR(new KViewGroup());  
edit_view_group->Create(20, h/2 - 80, w - 20*2, 80*2);  
AddView(edit_view_group);
```

ViewGroup添加ViewGroup

Question2: 如何在ViewGroup删除KView或ViewGroup呢?

AddViewToDel()函数可以从m\_lst\_view中移去KView或KViewGroup, 示例:

```
AddViewToDel(m_bk);  
m_bk.reset();
```

AddViewToDel只是将它从m\_lst\_view中移除,  
reset才是将智能指针的内容清空



## View的快速创建

```
void createTextViewHelper(KTextView_PTR* view, const kn_string& imgPath, kn_int x, kn_int y);  
void createImageHelper(KStaticView_PTR* view, const kn_string& path, kn_int x, kn_int y);  
void createTextHelper(KStaticView_PTR* view, const kn_string& txt, int fontsize, kn_int x, kn_int y, kn_int w, kn_int h);  
void createEditViewHelper(KEditView_PTR* view, int fontsize, kn_int x, kn_int y, kn_int w, kn_int h);
```

示例：

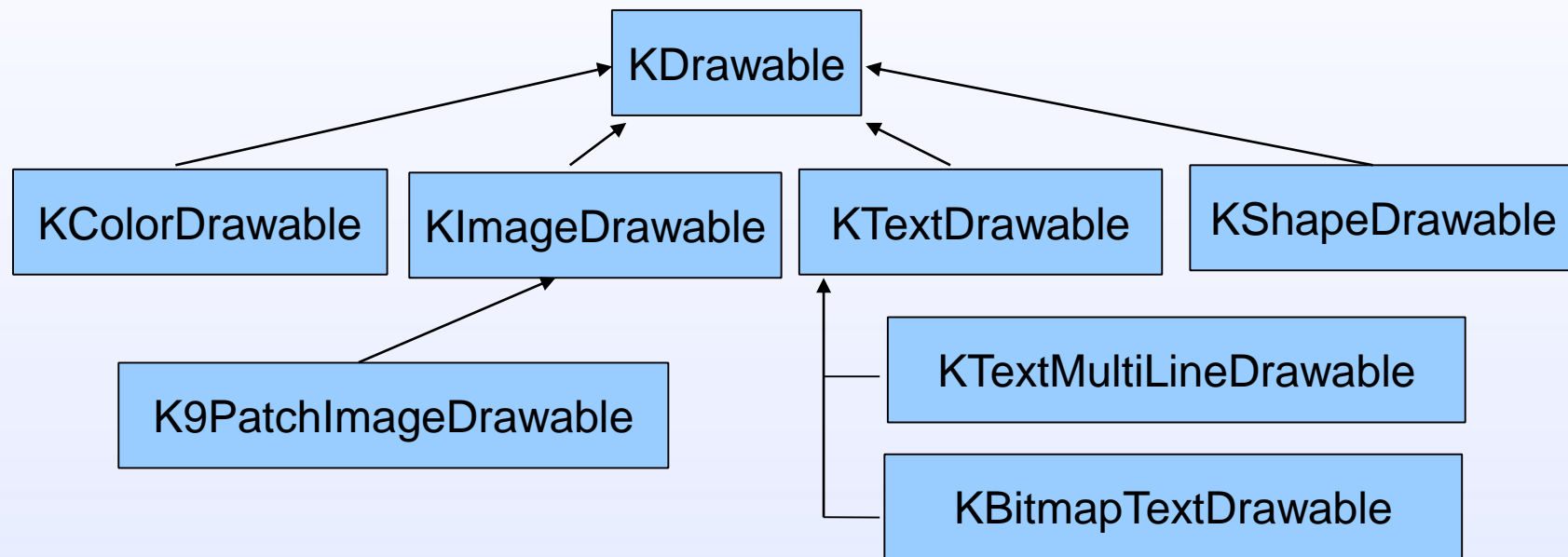
```
createTextViewHelper(&m_anction_view, _T("./resource/action_bk.9.png"),10, 10 );  
m_anction_view->setRect(10, 10, 100, 100 );
```

无需初始化及AddView， 在createTextViewHelper()中会创建和添加View



## 各种drawable的使用

定义Drawable的优点，在于将绘制处理进行了封装，这样控件层（view及其派生类），只用根据自己的显示需要，加入不同的drawable对象，而不用处理具体的绘制操作。由于简化了具体的绘制操作，View的对象只用在基类实现draw函数，子类不允许重载，因为局部刷新，异步渲染等特性，对draw的处理流程有一定规范要求，如果子类随意重载，不按处理流程实现，可能造成处理异常。目前引擎的Drawable有如下几种，之间的关系如下图所示：





如下给出一些Drawable的使用示例：

```
KColorDrawable_PTR panel_bk_drawable = KColorDrawable_PTR(new  
KColorDrawable(ARGB(255, 0, 0, 0)));  
panel_bk_drawable->setRect(0, 0, w, h);//设置drawable相对于所添加于对  
象（KView或KViewGroup）的位置  
panel_bk_drawable->setOpacity(255* 0.2);  
addDrawable(panel_bk_drawable);
```

```
m_stop_btn = KImgButtonView_PTR(new KImgButtonView());  
m_stop_btn->Create(0, 0, m_rect.width(), m_rect.height());  
AddView(m_stop_btn);  
KImageDrawable_PTR stop_icon_drawable = KImageDrawable_PTR(new  
KImageDrawable(_T("./resource/simulation_stop.png")));  
m_stop_btn->setIconDrawable(stop_icon_drawable);  
m_stop_btn->setIconCenter();
```





# 实现简单动画效果

为方便应用层使用的，引擎封装了动画参数处理的函数，一个函数即可完成动画设置，将动画的相关内容添加到动画属性列表中：

```
void addAnimationHelper(KView_PTR, kn_int PARA_ID, double v, kn_int duration, kn_int wait_time, kn_int loop_wait_time, KEasingCurve::Type = KEasingCurve::InOutCirc);  
void addAnimationHelper(KView_PTR, kn_int PARA_ID, double v1, double v2, kn_int duration, kn_int wait_time, kn_int loop_wait_time, KEasingCurve::Type = KEasingCurve::InOutCirc);  
void addRectAnimationHelper(KView_PTR, RERect rect, kn_int duration, kn_int wait_time, kn_int loop_wait_time, KEasingCurve::Type = KEasingCurve::InOutCirc);
```

引擎动画的属性参数定义目前有：

<b>#define PropertyPARA_ID_POS_X</b>	<b>1</b>
<b>#define PropertyPARA_ID_POS_Y</b>	<b>2</b>
<b>#define PropertyPARA_ID_Opacity</b>	<b>3</b>
<b>#define PropertyPARA_ID_RotateAngle</b>	<b>4</b>
<b>#define PropertyPARA_ID_ScaleX</b>	<b>5</b>
<b>#define PropertyPARA_ID_ScaleY</b>	<b>6</b>
<b>#define PropertyPARA_ID_USER</b>	<b>1000 //应用层从这个开始定义</b>



下面提供一些一个对象的单一属性动画的示例：

```
m_ani_thread.addAnimationHelper(m_simu_navi_ctrl_panel, PropertyPARA_ID_POS_X,  
30, 300, 0, 0, KEasingCurve::Linear);//位置变幻
```

```
m_ani_thread.addRectAnimationHelper(m_fm_small_circle_view_ptr, small_circle_rect,  
300, 50, 0, KEasingCurve::OutSine);//矩形变换
```

```
m_icon_view_ptr->SetScalePoint( m_icon_view_ptr->GetRect().width()/2,  
m_icon_view_ptr->GetRect().height()/2 );// 设置缩放的中心点  
m_ani_thread.addAnimationHelper(m_icon_view_ptr, PropertyPARA_ID_ScaleX, 0.0, 1.0,  
200, start_time, 0, KEasingCurve::OutSine);// X轴缩放动画  
m_ani_thread.addAnimationHelper(m_icon_view_ptr, PropertyPARA_ID_ScaleY, 0.0, 1.0,  
200, start_time, 0, KEasingCurve::OutSine);// Y轴缩放动画
```

```
m_ani_thread.setFrameTime(40); //设置帧长  
m_ani_thread.setStopMsg(ANIMATION_PALY_OVER_MSG);//动画结束后发送的消息  
m_ani_thread.Start();//启动动画
```

添加到动画属性列表

设置动画的相关属性，启动动画