

Symbian OS: 创建自定义控件

版本 1.0

2005 年 9 月 28 日

S
Y
M
B
I
A
N
O
S

法律声明

版权©诺基亚公司 20055。版权所有。

Nokia 和 **Nokia Connecting People** 是诺基亚公司的注册商标。**Java** 以及基于 **Java** 的商标是 **Sun Microsystems** 公司的注册商标。本文中提到的其它产品和公司名称可能是其相应公司的商标或商号。

否认声明:

本文内容按“现状” (**as is**) 提供, 即没有任何形式的保证, 包括对产品可销售、适合特定目的以及其它由本文任何建议、规范和范例衍生出来的任何保证。另外, 本文提供的信息是初级的, 因此在最终版本确定之前其可能有很大改动。本文目的仅是提供信息参考。

诺基亚公司不承诺承担任何责任, 包括对任何所有权的侵害责任, 尽管这些所有权与实施本文给出的内容有关。诺基亚公司不保证或声称使用本文内容不会侵害上述所有权。

诺基亚保留对本文, 在未经事先通知的情况下, 随时进行变更的权力。

许可声明:

允许对本文进行仅用于个人使用目的的下载和打印。在此没有许可任何其它知识产权。

目录

1.	简介	5
2.	为 Symbian OS 手机创建自定义控件.....	6
2.1	复合控件.....	6
2.2	窗口	6
2.3	输入事件.....	8
3.	CblinkText 范例.....	10
4.	控件类型	13
4.1	顶层控件	13
4.2	视图控件	13
4.3	绘制控件	14
5.	自定义控件	16
5.1	资源	16
6.	总结	18
7.	术语与缩写	19
8.	参考	20
9.	文档评价	21

修订记录

2005 年 9 月 28 日	版本 1.0	初始文档版本

1. 简介

本文介绍了自定义控件的设计、使用以及与这些控件有关的设计问题。虽然本文的重点是自定义控件的设计，但同时也介绍了控件结构体系的基础知识；另外本文对控件的派生与复合控件也进行了简要的介绍。文中未包括接口描述；它们可以在相关的 SDK 文档中找到。本文通过范例重点阐述了如何设计自定义控件。本文最后介绍了如何在视图中使用自定义控件以及如何在资源中定义自定义控件。文中所涉及的问题可以应用于任何 Symbian OS 平台，因为它们都具有类似的 UI 框架。

2. 为 Symbian OS 手机创建自定义控件

Symbian OS库提供了大量的UI标准控件。此外，它允许开发者创建自己的控件。当标准控件无法支持应用程序的UI时，则需要设计自定义控件。大多数Symbian OS UI控件派生自 `CCoeControl`类，`CCoeControl`类提供了一个控件框架并同时提供丰富的功能。`CCoeControl`通过将窗口服务器及其事件封装为公共Symbian OS应用框架的一部分，即控件环境，而隐藏了相关的复杂性。

`CCoeControl`是控件的基类，直接由它派生自定义控件可能比较繁琐；幸运的是，一些标准控件也可以用作自定义控件的基类。例如，`CEikEdwin`是单一文本编辑器（`CEikSecretEditor`除外）的基类，`CEikListBox`是所有列表框控件的基类。一些标准控件派生自`CEikAlignedControl`、`CEikBorderedControl`、`CEikMfne`和`CEikButtonBase`这些控件抽象类：它们的纯虚方法需要在派生类实例化前实现。

在大多数情况下，专用控件通过继承创建。例如，一般来说，对话框是由`CEikDialog`派生的控件。`CEikDialog`类实现了对话框的标准框架，相关特定函数需要派生类实现，如在设置控件大小之前调用的`CEikDialog::PreDynLayoutInitL`。此函数的缺省实现为空，但派生的对话框可以覆盖它来设置子控件的属性，并且正确计算它的大小。

一个控件可以包括其它控件，形成所谓的复合控件，它可以包含一个或多个子控件。复合结构通常可以递归，它可以使用子控件实现复杂的功能。值得注意的一点是，复合控件不在屏幕上绘制任何内容，因为通过子控件在UI上绘制。（复合控件可以绘制其子控件的背景。`Series 60 SDK`包含解释此方法的范例。）

2.1 复合控件

小组件的聚合是面向对象编程的核心，其设计思想在复合控件的设计中得以体现，而复合控件正是由子控件组成的。创建复合控件的主要原因是多个控件可以作为一个控件处理。由于复合控件也是由`CCoeControl`继承而来，因此它们可以和所有其它控件是一样的。例如，子控件的位置与父控件的位置有关，这样父控件的调整也会为所有子控件设置新的绝对位置。

复合控件实现的函数允许控件框架在初始化和绘制控件时访问子控件。`CCoeControl::CountComponentControls`可以返回子控件的数目，`CCoeControl::ComponentControl`方法可以返回一个指向子控件的指针。任何时候，所有子控件都可以通过`CCoeControl::ComponentControl`被访问，无论它们是否可见。应该通过`CCoeControl::MakeVisible`方法调整控件的可视性。复合控件拥有子控件，因此它负责子控件的创建与销毁。复合控件还应该设置其子控件的位置和大小；必须确保所有子控件都位于复合控件矩形内，并且可视的子控件矩形不能互相重叠。

2.2 窗口

`RWindow`类代表设备屏幕上窗口服务器支持的区域。控件能够被绘制到窗口中，并且在窗口具有焦点时从窗口服务器接收事件。所有控件都通过使用一个窗口访问屏幕很重要。控件可以拥有它们自己的窗口或使用其它控件的窗口。一种常用的设计方法是顶层复合控件创建一个窗口，然后它的所有子控件共享该窗口。

在控件之间共享窗口可以节约资源，因此Symbian建议控件不创建它们自己的窗口。出于性能考虑，窗口应尽可能少，因为它们需要消耗资源。窗口作为窗口服务器的一个客户端，窗口越少越意味着上下文切换和进程间通信的减少。

控件窗口通过 `CCoeControl::CreateWindowL` 方法创建。它有一些重载方法，但最常用的是没有参数的重载方法，它基于控件的内存地址创建主窗口及其句柄。另一个有用的重载方法是使用 `CCoeControl` 作为参数，从而为控件的窗口创建子窗口，例如控件使用窗口滚动或剪切时需要操作子窗口——作为一些特例，为子控件创建窗口而不是使用父控件的窗口是需要的。

如果一个控件没有自己的窗口，则应该通过调用 `CCoeControl::SetContainerWindowL` 方法设置它需要使用的窗口。该方法有一些重载，但对于复合控件，使用 `CCoeControl` 作为参数的重载最为有用。该参数可以是拥有窗口的控件，也可以是与其它控件共享窗口的控件。

当一个控件在另一个控件的窗口中绘制时，其位置与窗口相对。如果一个控件拥有的是一个子窗口，该控件的位置与父窗口相对；然而，如果该控件是复合控件，则其子控件的坐标与其父控件窗口是相对的。顶层窗口所拥有的控件的显示位置与屏幕相对，即使用显示屏的物理坐标。

为了解释以上内容，假设有三个控件，A、B 和 C（如图 1 所示）。A 是顶层控件，它拥有一个主窗口。在第一个范例中，子控件 B 和 C 不创建它们自己的窗口。

A 的子控件 B 不创建窗口，它拥有一个子控件 C，B 的窗口通过调用 `CCoeControl::SetContainerWindowL(A)` 方法设置，C 的窗口通过调用 `CCoeControl::SetContainerWindowL(B)` 方法设置。C 的位置 (p) 与 A 是相对的，因为它是真正的窗口所有者。A 的位置 (m) 与屏幕位置是相对的。

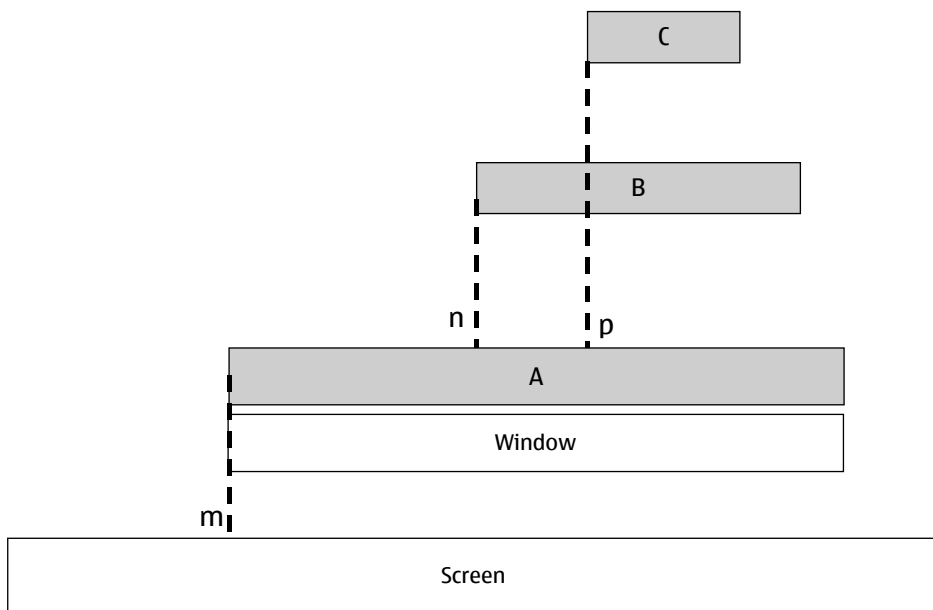


图 1：顶层父控件拥有窗口时，三个控件的相对位置

然而，如果 B 是 A 的一个子控件，但它有自己的窗口，即 A 窗口的子窗口（如图 2 所示）。如果 C 是 B 的子控件，并且通过调用 `CCoeControl::SetContainerWindowL(B)` 设置它的窗口，C 的位置 (p') 是与 B 的窗口相对的。

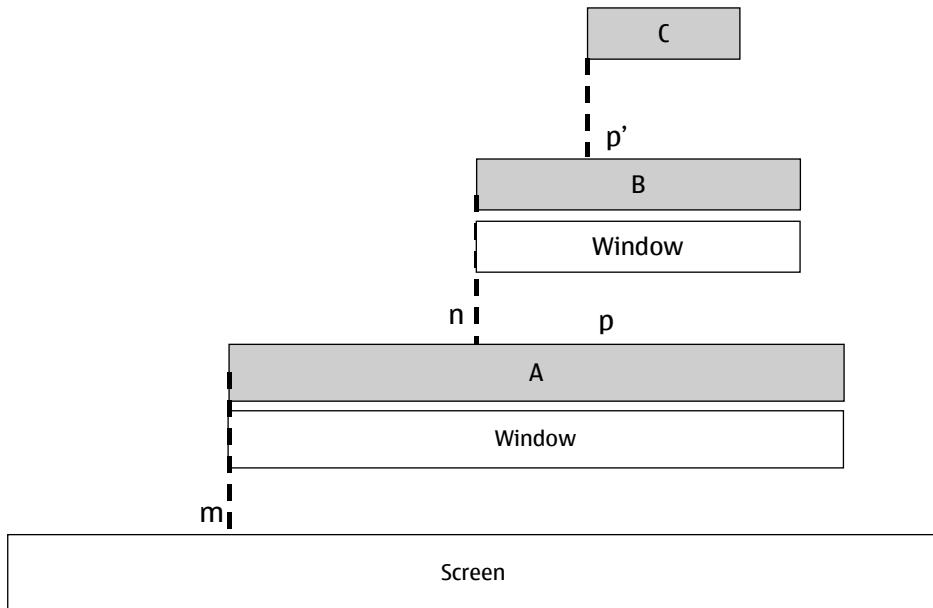


图 2：子控件拥有窗口时，三个控件的相对位置

如本例所述，控件位置取决于绘制该控件的窗口；因此，设计者需要知道各个控件的绘制窗口。在设计UI布局时，这是一个重要问题。一些标准控件拥有自己的窗口，如菜单、对话框和滚动条等浮动控件。这些控件动态地调整它们的绘制，且坐标调整对设计者透明。

`CCoeControl::OwnsWindow`调用可以检测控件是否拥有窗口。

各控件拥有自己的区域或边界矩形；通常所期望的行为是控件在边界矩形内绘制。控件在边界矩形之外绘制时会发生剪切。剪切使一些绘制效果易于实现，例如，移动控件位置的滚动，任何时候只有一部分控件内容可见。

如果控件需要剪切，它必须拥有一个窗口，因为控件本身不具备剪切功能——剪切在控件的窗口中执行。非拥有窗口的控件拥有一个边界矩形，它可以在边界矩形以外的位置上绘制，只要该位置在它正使用的窗口中。因此，控件的`CCoeControl::Draw`方法应该只绘制它自己的边界矩形，任何子控件应该在它们的复合控件边界框内绘制。否则，重新绘制可能导致混乱，例如对于非拥有窗口的控件调用`CCoeControl::DrawDeferred`的情况，复合控件边界框的重绘使控件窗口的边界矩形部分失效，边界矩形以外的像素将不重新绘制——它们将被剪切；但是如果整个窗口被重新绘制，控件边界框以外的像素则被绘制——不会被剪切。由于控件绘制方式不同，因此依赖于绘制发起方式会导致画面错误。

2.3 输入事件

控件可以接收按键或指针事件。窗口服务器产生事件，应用框架使用活动调度器循环侦听事件。当发生事件时，它将被发送到具有焦点的控件，框架然后根据事件类型调用控件的处理方法；例如，为按键事件调用`CCoeControl::OfferKeyEventL`，为指针事件调用`CCoeControl::HandlePointerEventL`。

应用框架仅向压入事件堆栈中的控件提供按键输入事件。通过`CCoeAppUi::AddToStackL`方法向事件堆栈压入输入接收控件。事件堆栈中可以有多于一个控件。在发生事件时，按照压入到堆栈中的顺序为控件调用`CCoeControl::OfferKeyEventL`方法，即首先提供事件给最后压入的控件。事件被依次传给堆栈中的各个控件，直到控件返回

TKeyResponse::EKeyWasConsumed。 **TKeyResponse**的一种常用策略是，如果控件使用事件执行某些内容，则该事件将被消耗掉。通常一个顶层复合控件被压入到堆栈中，顶层控件将事件分配给它的子控件。然而，无论父控件是否在事件堆栈中，事件都不会自动提供给子控件。如果子控件需要事件，复合控件应执行**OfferKeyEventL**，将按键事件传递给子控件。该方法使事件流程的实现更加准确、清晰，而且更不容易出现错误。

按一次按钮可产生三个事件：**EEventKeyDown**、**EEventKey**和**EventKeyUp**，因此**CCoeControl::OfferKeyEventL**将被调用三次，在按下、保持和释放按钮时各调用一次。在大多数情况下，控件只需要**EEventKey**，无需处理其它事件；然而，需要更准确输入的游戏和其它应用可能使用所有按键事件。**TKeyEvent**保存真正的事件，其**iCode**成员中定义为**TKeyCode**枚举值或**UNICODE**字符编码值。例如，如果应用需要检测按钮‘a’是否按下，则可以通过确定**iCode**是否等于值‘a’（**UNICODE**映射中为97）来实现。一些标准按键是在**uikon.hrh**中定义的。应用框架本身也拥有一个内嵌的键事件捕捉方法，相关键事件直接会被应用程序捕捉，如**keylock**，而不提供给控件堆栈中的控件。

如果控件需要指针事件，则需要实现**CCoeControl::HandlePointerEventL**方法。窗口服务器仅向拥有窗口的控件提供指针事件。在缺省情况下，当指针事件坐标位于边界矩形内时，控件才获得事件。如果控件不拥有自己的窗口，则通过控件框架提供事件。这里，**CCoeControl::HandlePointerEventL**的缺省实现将检查各子控件，如果坐标位于子控件的边界矩形内，则调用其**HandlePointerEventL**方法。当复合控件覆盖**HandlePointerEventL**方法时，它应该调用缺省的实现，因为子控件也会消耗指针事件。

3. CBlinkText 范例

为了说明第 2 章所讨论的复合控件功能，本章将给出一个简单的范例，*CBlinkText*——一个闪烁文本(*blinker*)的基本范例。闪烁文本是被设定为周期性可见和不可见的文本行。阅读闪烁文本比较困难，但它是强调消息重要性的一种有用方法。

*CBlinkText*说明了通过组合标准控件创建新功能的简便性。本例中，复杂的文本绘制由 *CEikLabel* 组件处理，而 *CBlinkText* 用作一个容器，设置周期可见性来绘制 *CEikLabel*。

CBlinkText 类由 *CCoeControl* 公有派生，以适用控件框架。公有方法包括闪烁文本特有 API，用于设置和获取控件特定属性的公有 *CCoeControl* API。

```
class CBlinkText : public CCoeControl
{
public:
    void ConstructL(CCoeControl& aParent);
    CEikLabel& Label() const;
    void Start(TTimeIntervalMicroSeconds32 aInterval);
    void Stop();
    ~CBlinkText();
    TSize MinimumSize();
private:
    TInt CountComponentControls() const;
    CCoeControl* ComponentControl(TInt aIndex) const;
    void SizeChanged();
    static TInt Tick(TAny* aThis);
    void DoTick();
private:
    CEikLabel* iLabel;
    CPeriodic* iTicker;
};

void CBlinkText::ConstructL(CCoeControl& aParent)
{
    SetContainerWindowL(aParent);
    iLabel = new (ELeave) CEikLabel();
    iLabel->SetContainerWindowL(*this);
    iTicker = CPeriodic::NewL(CActive::EPriorityIdle);
    Start();
}
```

*ConstructL*是Symbian OS编程中的第二阶段构造函数。构造函数创建*CEikLabel*与*CPeriodic*实例。*CPeriodic*是一个简单的定时器，用于触发周期性函数调用。定时器使用最低优先级创建，因为该控件没有实时要求。构造函数同时为本身和标签控件设置窗口，因为闪烁文本控件没有自己的窗口。一些控件（类似*CBlinkText*所做的）自动设置它们的容器窗口，但另一些控件像*CEikLabel*需要显式设置。如果构造函数使用父控件作为参数，则很可能有此设置要求，因为它也需要设置容器窗口，但通常了解要求的唯一方法是检查API文档。

```
CEikLabel& CBlinkText::Label() const
{
    return *iLabel;
}
```

*Label*方法返回*CEikLabel*引用，它可以用于设置文本及其属性。提供这些访问函数并非最佳的面向对象设计，但在此例中它使得程序较为简单。

```
void CBlinkText::Start(TTimeIntervalMicroSeconds32 aInterval)
{
    iTicker->Start(aInterval, aInterval, TCallBack(Tick, this));
}
```

Start启动绘制。与桌面PC相比，Symbian OS中的绘制相对较慢。一个 **TTimeIntervalMicroSeconds32**非常短，比实际系统的定时器短得多。在大多数设备中，硬件定时器的准确性为几毫秒，软件中定义的值无法更精确；因此当**aInternal** 的值比 1000 微秒小时，实际定时器周期通常比 1000 微秒大。

```
void CBlinkText::Stop()
{
    iTicker->Cancel();
}
```

Stop 终止绘制。在移动设备中，可用的电能有限，因此开发者应该避免使用长时间运行、快速跳动的定时器。在控件不可见或系统进入空闲模式时，**Stop**应该被调用。跳动的定时器使系统无法进入节能模式，从而快速消耗电能。

```
CBlinkText::~CBlinkText()
{
    if(iTicker)
        iTicker->Cancel();
    delete iTicker;
    delete iLabel;
}
```

析构函数清除实例。定时器必须在被删除前停止，并且应该进行检查以确保它不为 NULL，例如，第二阶段构造函数可能由于内存不足错误而失败。

```
TInt CBlinkText::CountComponentControls() const
{
    return 1;
}
```

CountComponentControls 返回复合控件中的子控件数。在递归遍历控件树时，需要子控件数；例如，当初始化应用时、在顶层控件调用 **ActivateL** 方法后、或在进行绘制操作时均需要子控件数。

```
CCoeControl* CBlinkText::ComponentControl(TInt /*aIndex*/) const
{
    return iLabel;
}
```

控件框架使用 **ComponentControl** 方法访问子控件，通过 **CountComponentControls** 返回的子控件数以确定需要进行多少次访问。

```
void CBlinkText::SizeChanged()
{
    iLabel->SetExtent(Position(), Size());
}
```

当 **SetRect**、**SetSize** 或 **SetExtent** 方法被调用时，**SizeChanged** 将被调用。**SetSize** 方法是复合控件调整布局的便利之处。闪烁文本中只存在一个填充了整个复合控件的子控件。

```
TSize CBlinkText::MinimumSize()
{
    return iLabel->MinimumSize();
}
```

MinimumSize 提供了控件边界框所需的最小面积。在没有任何更多有关控件内容信息的情况下，复合控件需要此最小面积信息。

```
TInt CBlinkText::Tick(TAny* aThis)
{
    static_cast<CScrollText*>(aThis)->DoTick();
    return 0;
}
```

```
}
```

Tick 是一个静态回调函数，定时器在到期时调用它。**Timer** 是一个活动对象，并且从它的 **RunL** 方法调用回调函数。由于活动对象为非抢先式，因此回调函数应该快速返回——其它回调函数（包括输入）直到它返回后才进行处理。

```
void CBlinkText::DoTick()  
{  
    iLabel->MakeVisible(!iLabel->IsVisible());  
}
```

DoTick 仅仅设置标签的可见性。

4. 控件类型

4.1 顶层控件

大多数 Symbian OS 应用都有一个状态面板，一个命令面板和一个应用视图。应用视图是应用的主窗口，它被实现为一个拥有窗口的控件——顶层控件。顶层控件可能是文本编辑应用的一个文本控件，或者仅仅是为图像浏览应用绘制位图。顶层控件从 `CCoeControl` 派生，在对话框体系结构中，顶层控件从 `CEikDialog` 派生。如果顶层控件处理事件，则通过 `CCoeAppUi::AddToStackL` 方法将其添加到事件堆栈中。控件被销毁之前，必须使用 `CCoeAppUi::RemoveFromStack` 方法将其从事件堆栈中删除。在顶层控件的生命周期中，可以根据应用设计者的需要，向事件堆栈添加控件或从事件堆栈删除控件。

创建顶层控件不同于创建其它控件。以下是创建顺序的一个基本范例：

- `CreateWindowL();`

顶层控件必须创建一个窗口，用于该控件及其子控件的绘制。

- `Window().SetShadowDisabled(ETrue);`

一个应用顶层控件没有阴影。对话框可以有阴影；该阴影具有三维效果，它使得对话框看起来像置于应用之上。

子控件同时在这一步创建——即窗口创建之后，窗口矩形设置之前。

- `SetRect(iEikonEnv->EikAppUi()->ClientRect());`

顶层控件矩形被设置为框架为应用指定的区域。调用 `SetRect` 将会调用 `SizeChanged` 方法，这里应该设置所有子控件的位置和大小，从而调整 UI 的控件布局。

- `SetBlank();`

如果设置了空白标志，缺省的 `CCoeControl::Draw` 方法则使用缺省的单色填充应用程序区域。由于复合控件的 `CCoeControl::Draw` 方法先于子控件的 `CCoeControl::Draw` 方法被调用，因此如果背景被设为空白，即使子控件未实现 `Draw` 方法，子控件也无须为防止控件之下的应用程序可见而覆盖整个应用程序区域。

- `ActivateL();`

对于一些控件，一些初始化操作可能无法在此阶段之前执行。因此，`ActivateL` 为所有子控件的实际初始化提供了最后的地方。缺省实现为控件的绘制做准备，如果对此方法进行覆盖，需要调用缺省实现。

- `iEikonEnv->EikAppUi()->AddToStackL(this);`

如果应用控件需要处理事件，它应该将自身加入到应用事件堆栈。

4.2 视图控件

从 Symbian OS v6.1 及更高版本开始，应用框架可以利用视图体系结构的优势。任何应用都可以提供视图，这些视图可以从应用内部或外部被调用。视图通常被实现为一个从

MCoeView 接口派生的顶层控件。**CCoeAppUi::RegisterViewL** 向视图服务器注册视图。应用可以实现并注册多个顶层控件，这些控件通过视图服务器激活和去活。

每个视图控件具有一个 ID。当激活视图时，需要提供应用 UID（唯一标识符，任何应用或库都必须包含的 32 位受控值）和一个视图 UID。视图服务器启动与 UID 匹配的应用（如果已经运行，则将其送到前台）；在创建期间，应用的所有视图都被注册到视图服务器，视图服务器寻找正确的视图 ID 并将其送到前台。

快速的视图激活很重要；提供视图的任何应用必须快速响应，启动要快。这是由于视图服务器是单线程服务器，同一时刻只能处理一个激活请求；因此，如果存在其它请求，它们必须等待，直到第一个激活请求完成。一个行为较差的视图提供者可能会使整个设备停止工作，直到此行为完成或视图服务器强行关闭应用。如果发生设备工作停止，显示屏将不会更新，应用也不会响应输入。

4.3 绘制控件

当更新屏幕上的控件区域时，控件框架调用 **CCoeControl::Draw** 方法。缺省实现在控件设置空白标志时，会使用背景颜色填充区域；否则不执行任何动作。复合控件本身通常不实现 **Draw**，而是由子控件完成所有绘制。控件框架首先调用父控件的 **Draw** 方法，然后递归调用各子控件的 **Draw** 方法。

在大多数情况下，控件通过屏幕设备图形上下文在屏幕上绘制，并且通过 **CCoeControl::SystemGc()** 方法访问图形上下文。图形上下文提供一系列 GDI（图形设备接口—通用 Symbian OS 图形 API）绘制接口，用于屏幕上的实际绘制。GDI API 的速度不是特别快，通常性能问题会影响软件体系结构设计。

永远不要直接调用 **Draw** 方法。它是一个屏幕更新时调用的回调函数。在 **CCoeControl::ActivateL** 方法被调用后，任何时候都可以调用 **Draw**。窗口服务器可以确定何时更新屏幕区域，或者何时通过调用 **CCoeControl::DrawDeferred** 或 **CCoeControl::DrawNow** 函数进行绘制。

CCoeControl::DrawNow 能够直接启动绘制，并且 **CCoeControl::Draw** 方法会在 **DrawNow** 方法返回前被调用。**CCoeControl::DrawDeferred** 能够发起异步操作；它能够立即返回，但实际绘制会在优先级较低的空闲对象运行时执行——这样，更急迫的任务，如输入处理将首先被执行。绘制始终是一项繁重的操作；它给设备带来压力，频繁的画面更新通常会影晌性能。因此应该避免不必要的绘制。在大多数情况下，**DrawDeferred** 是推荐的绘制方法：在屏幕缓存区中，控件矩形区域被标识为无效，用于随后的更新。在屏幕缓存区被更新前，调用 **CCoeControl::DrawDeferred**，或者由于其它原因标识区域失效，则重叠更新的绘制无效。

所有绘制应该在控件的 **Draw** 方法内中执行，除此之外不应执行其它工作。**Draw** 方法应该尽可能快。例如，动态创建字体并在绘制时读取位图或资源是较差的设计。根据经验，好的设计在 **Draw** 方法不应该有一个陷阱处理程序；任何可以预先执行的费时功能应该进行缓存。

对于执行大量绘制操作的控件，应该对绘制操作进行缓存，即双缓存操作。双缓存操作首先对内存上下文执行绘制，接着在 **Draw** 方法中，仅将上下文的位图传送给屏幕。在 Symbian OS 中，实现双缓存最简便的方法是创建一个 **CFbsBitmap**，然后和图形上下文绑定——这样就可以使用屏幕上下文的 GDI 接口。在内存位图缓存中执行绘制，当窗口服务器更新失效屏幕区域时，内存缓存将被拷贝到其中。游戏中经常使用双缓存，但也可以用于绘制控件性能非常重要的任何应用中。

以下是有关如何创建和使用双缓存的一个简短的范例：

```
iGcBmp = new (ELeave) CWsBitmap(iEikonEnv->WsSession());
User::LeaveIfError(iGcBmp->Create(aClientRect.Size(), iEikonEnv-
>ScreenDevice()->DisplayMode()));
iGcDevice = CFbsBitmapDevice::NewL(iGcBmp);
User::LeaveIfError(iGcDevice->CreateBitmapContext(iGc));
```

`iGcBmp` 是一个 `CWsBitmap` 指针，指向位图内存缓存区，它被创建为具有与顶层控件相同的宽度和高度以及与相同的显示色位深度。`iGcDevice` 是一个指向 `CBitmapDevice` 设备的指针，而上下文 `iGc` 指向 `CbitmapContext` 的实例。当控件绘制自身时，使用的是 `iGc` 而非从 `CCoeControl::SystemGc()` 方法获得的 `CScreenGc`，双缓存绘制应该在 `CCoeControl::Draw` 方法之外执行，需要时可以直接调用。只有在离屏绘制的结束时，内存缓存才通过调用 `CCoeControl::DrawDeferred()` 传到屏幕。

```
void CMyDrawingExample::Draw(const TRect& /*aRect*/) const
{
    SystemGc().BitBlt(TPoint(0, 0), iGcBmp);
}
```

5. 自定义控件

5.1 资源

如果对话框重用控件，那么创建资源文件更方便，这样可以通过资源来创建控件。

这里给出了一个非常简单的范例，它介绍了如何扩展 *CBlinkText* 以便从资源文件定义它。通过 Symbian OS 的资源编译器 *rcomp*，可以将资源文件编译成二进制资源。资源文件和 C 语言的语法相似。

资源必须在使用前引入。通常资源定义在 **.rh* 文件中，与 C++ 文件共享的常量定义在 **.hrh* 文件中。对于 *CBlinkText*，只有一个常量：类型标识符；这是一个唯一的整数值，不能与其它控件的类型标识符冲突。如需了解更多信息，参见平台的 *.hrh* 文件。

常量按如下方式在 **.hrh* 文件中定义：

```
enum {KBlinkTextType = KAknCtLastControlId};
```

在 **.rh* 文件中有一个对资源声明的定义：

```
STRUCT BLINKTEXT
{
    LTEXT txt;
    LONG period = 2000000;
}
```

STRUCT 关键字定义了闪烁文本的资源参数结构。一些内置的类型可以被使用。**LTEXT** 保存 **UNICODE** 文本，**LONG** 保存 32 位数值。给参数赋值时，它们成为缺省值。

在资源文件中定义自定义控件与任何普通控件相似；下列是从资源文件中摘取的一个简短范例，说明了如何在对话框中使用闪烁文本：

```
RESOURCE DIALOG r_blinker_dialog
{
    flags = EEikDialogFlagWait;
    title = "Blinker";
    buttons = R_EIK_BUTTONS_CANCEL_OK;
    items =
    {
        DLG_LINE
        {
            type = KBlinkTextType ;
            control = BLINKTEXT
            {
                txt="hello world";
                period=1000000;
            };
        };
    };
}
```

控件框架仅识别普通控件，控件工厂 (factory) 方法通过标识符创建它们。由于自定义控件具有它们自己的标识符，因此这些控件需要通过对话框代码创建。

CEikDialog::CreateCustomControlL 是一个类工厂方法，闪烁文本对话框的最小实现包含以下内容：

```
class CBlinkDialog : public CEikDialog
{
```

```

    SEikControlInfo CreateCustomControlL(TInt aControlType);
};

SEikControlInfo CBlinkDialog::CreateCustomControlL(TInt aControlType)
{
    if(aControlType == KblinkTextType )
    {
        SEikControlInfo sinfo;
        sinfo.iControl = new (ELeave) CBlinkText();
        return sinfo;
    }
    else
        return CEikDialog::CreateCustomControlL(aControlType);
}

```

CBlinkText 必须覆盖 **ConstructFromResourceL** 方法以便能够由资源创建；对话框框架能够在构造其间调用它。**CBlinkText::ConstructFromResourceL** 采用与 **ConstructL** 方法相同的方式创建控件，但使用 **TresourceReader** 作为参数。资源读取器是一个工具类，它从资源文件中读取字节；因此，读取顺序和数据长度必须与资源结构的声明相匹配。此控件的窗口上下文由对话框框架设置，但它仍需调整子控件的窗口上下文。

```

void CBlinkText::ConstructFromResourceL(TResourceReader& aReader)
{
    iLabel = new (ELeave) CEikLabel();
    iLabel->SetContainerWindowL(*this);
    iTicker = CPeriodic::NewL(CActive::EPriorityIdle);
    TPtrC label = aReader.ReadTPtrC16();
    iLabel->SetTextL(label);
    const TInt interval = aReader.ReadInt32();
    Start(interval);
}

```

6. 总结

Symbian OS 手机上的所有用户界面都是通过控件创建。鼓励开发者在界面设计中使用普通标准控件，因为它们为应用提供一致的外观感受。标准控件可以自动与框架交互，从而为应用开发者排除了布局设计的问题。如果需要为用户提供更好的用户体验，可以利用自定义控件来实现。通常，新控件应该与平台总体的 UI 风格一致。但也可以通过自定义控件创建不同于标准 UI 风格但与特殊 UI 风格相匹配的接口。例如，采用仿效物理 CD 播放器上的控件为音乐播放器应用提供一个界面，可以使用户快速学习如何使用。这意味可以使用其它平台或物理世界中用户理解熟悉的界面。然而，UI 设计者应该注意避免过度使用自定义控件，因为它们可能会给用户带来困惑。在使用自定义控件时，建议整个应用、整个屏幕保持一致且有针对性。应该特别谨慎地处理自定义控件与标准控件的混合使用。此外，自定义控件可能需要与 UI 风格进行适配。一些 UI 风格包括主题支持。

本文讨论了控件体系结构，给出了对所有 UI 控件运行环境的概述。然后详细介绍了如何创建复合控件，接着给出了一个简单的闪烁控件的范例。在讨论如何绘制控件，如何从资源文件和对话框资源创建自定义控件之前，介绍了顶层控件和视图。

Symbian OS 控件框架允许开发者在 UI 控件设计中百花齐放。虽然灵活性会明显带来一些复杂度，但是基于对常用复合控件概念的理解，任何开发者都很容易创建自定义控件。在阅读完本文档后，开发者应该能够设计自定义控件，并将它们与应用引擎相集成。

7. 术语与缩写

术语与缩写	含义
边界矩形	控件的边界矩形是一个定义控件所控制的可显示区域的大小和位置的矩形。
控件	屏幕的一个矩形区域，可以响应用户输入事件。
子控件	构成复合控件的控件。子控件通常由复合控件创建和销毁。
复合控件	封装了一个或多个子控件的控件，因此用户界面是这些子控件的集合，并且与子控件的编程接口也是通过复合控件实现的。
自定义控件	应用开发者所写的控件，而非 SDK 所提供的控件。
控件环境	为窗口服务器的异步服务提供活动对象接口，并且为控件和应用 UI 提供框架。
工厂(Factory)模式	工厂(Factory)是一个接口类，其派生类能够创建对象。此模式在类（创建器）预先不知道它要创建的所有子类时使用。相反，由各子类（具体创建器）创建实际对象实例。
窗口服务器	管理用户输入和绘制屏幕的 Symbian OS 线程。

8. 参考

Series 60 Developer Platform: Application UI Customization,
<http://www.forum.nokia.com/documents>

9. 文档评价

In order to improve the quality of documentation, we kindly ask you to fill in the [document survey](#).

为了提高文档质量，我们衷心邀请您填写[文档调查](#)。