# 使用DTK开发应用程序

在前面几章的内容中，我们介绍了如何使用Qt开发应用程序，以及如何在扩展性、健壮性和安全性等方面对开发的应用程序进行优化。本章我们将开始介绍如何使用DTK开发库提供的接口和图形控件，加强Qt应用程序的表现力和提高开发效率。

## DTK简介

DTK（Deepin Toolkit）是深度操作系统提供的一套应用程序开发框架。其前身为Deepin UI项目以及后来的DUI项目，这两个项目均为在深度操作系统的桌面环境、原创应用开发的过程中积累的一些常见工具类和控件库。

其中，Deepin UI项目为Python项目，基于GTK+2.0图形开发框架，主要是支撑深度操作系统1.0和2.0版本的开发；DUI项目则为C++项目，基于Qt5图形开发框架，主要是支撑深度操作系统3.0版本的开发；在深度操作系统4.0及之后版本的开发中，为了规范DUI项目的开发并且重新梳理其结构，将DUI项目改名为DTK，并分拆成了以dtk开头的几个模块，一直开发维护至今。

所以，狭义上的DTK是指dtkcore、dtkwidget和dtkwm等项目的合称，其功能主要包括新的控件和控件样式、程序日志、文件监控、汉字转拼音等；而广义的DTK则是指深度操作系统提供的一系列应用程序开发接口，除了前面提到的几个项目，还有libdframeworkdbus、任务栏插件、文件管理器插件等用来扩展桌面环境功能的开发库。

本章除了介绍狭义概念上DTK各方面的使用，也会对任务栏插件、文件管理器插件等方面的使用进行说明。

## DTK的结构

DTK按照功能模块划分成了三个项目，即dtkcore、dtkwidget和dtkwm，它们主要的功能如下表所示：

表 7.1 DTK模块功能描述表

|  |  |
| --- | --- |
| 项目 | 功能描述 |
| dtkcore | 提供应用程序开发中的工具类，如程序日志、文件系统监控、格式转换等工具类 |
| dtkwidget | 提供一些额外的控件和控件样式 |
| dtkwm | 封装了窗口管理相关信息获取、设置的类 |

作为C++开发库，DTK的每个模块有自己的命名空间，如dtkcore的命名空间为Dtk::Core，在程序中一般使用宏DCORE\_BEGIN\_NAMESPACE、DCORE\_END\_NAMESPACE以及DCORE\_USE\_NAMESPACE来分别表示开始Dtk::Core命名空间、结束Dtk::Core命名空间和使用Dtk::Core命名空间。之所以这样做，一方面是为了方便IDE进行一次性补全，提升开发效率；另一方面，则是在开发之初对于命名空间的命名方式意见不太统一，为了方便以后修改方便，统一在使用命名空间的时候使用宏而非具体的命名空间。另外两个项目dtkwidget和dtkwm也是类似规则，分别使用Dtk::Widget和Dtk::Wm命名空间。

在每个模块内部，公开类均使用D-Pointer模式，以保证开发库的二进制兼容性，也就是在开发库不发生API变化的情况下，尽量保证应用程序无需重新编译、链接即可使用新的动态库正常执行。

简单来说，使用D-Pointer就是将类的私有变量进行封装，放在一个只有内部访问的类中。这样对应用程序来说，即使类内部增加或者减少了私有变量，也不影响程序链接之初定义的符号偏移量，从而保证了库的二进制兼容性。

## 创建DTK程序

因为DTK基于C++/Qt5开发的特点，所以DTK本身与QtCreator有比较好的集成，我们在QtCreator中可以直接创建DTK程序（如非深度操作系统环境，则需安装qtcreator-template-dtk包支持），如下图所示：

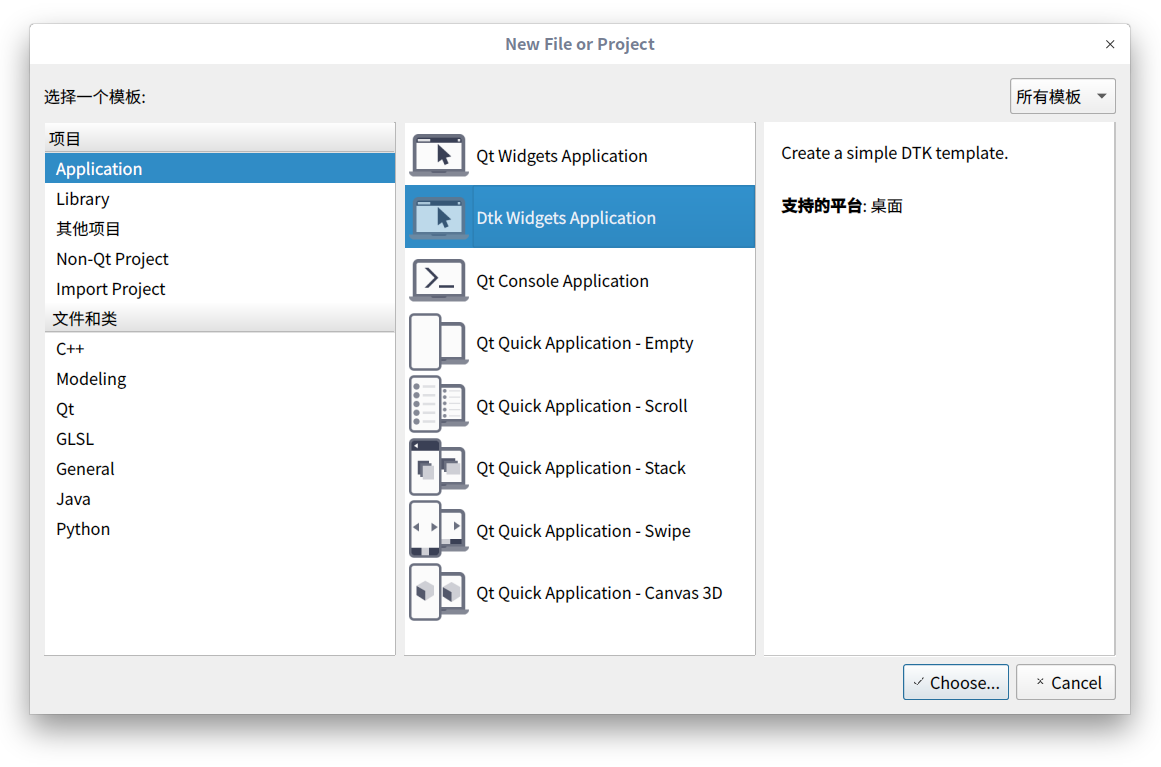


图 7.1 新建DTK项目

在QtCreator创建项目的对话框中选择“Dtk Widgets Application”一项，后面的步骤根据提示进行选择即可，与创建普通的Qt程序相同，这里不再赘述。

程序创建完成后，可以直接按下快捷键Ctrl+R或者点击程序运行按钮编译运行程序，新创建的DTK程序默认包含一个主窗口，显示效果如下图所示：

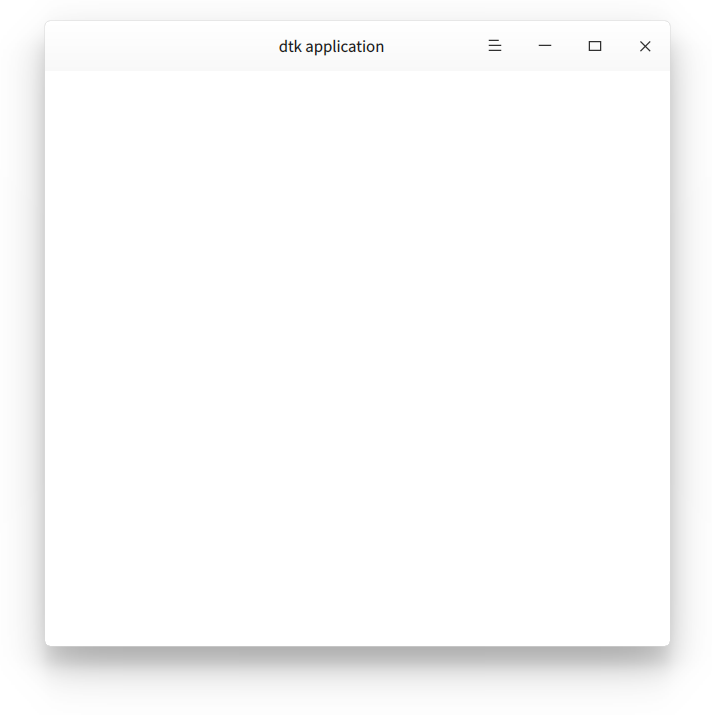


图 7.2 运行新建的DTK程序

打开程序的.pro文件，内容如下：

QT += core gui

greaterThan(QT\_MAJOR\_VERSION, 4): QT += widgets

TARGET = dtk-gallery

TEMPLATE = app

CONFIG += c++11 link\_pkgconfig

PKGCONFIG += dtkwidget

SOURCES += \

main.cpp

RESOURCES += resources.qrc

可以看到，文件中的内容与普通Qt程序的.pro文件没有太多不同，只是默认开启pkg-config支持，并且添加了dtkwidget库为依赖。另外，文件中默认添加了名为resources.qrc的资源文件，这个文件是为了方便添加DTK程序需要的一些图标资源文件，如刚才提到的程序关于对话框中的图标以及标题栏左侧的应用程序图标等。

在main.cpp文件中，DApplication和DMainWindow替代了Qt中常见的QApplication和QMainWindow，构成了一个最基本的DTK程序。

其中，DApplication继承自QApplication，在使用方式上没有太大不同，DApplication在QApplication的基础上，加入了对桌面环境特性（如窗口背景模糊、窗口圆角等）的融合。需要注意的是使用DTK的控件一般会要求程序使用DApplication，否则效果可能相差甚大；DMainWindow则继承自QMainWindow，虽然两者是继承关系，但是由于两者在表现上差异较大，因此在使用上面也有些不同，后面会详细进行说明。

main.cpp文件的内容如下：

#include <DApplication>

#include <DMainWindow>

#include <DWidgetUtil>

DWIDGET\_USE\_NAMESPACE

int **main**(int argc, char \*argv[])

{

DApplication::loadDXcbPlugin();

DApplication a(argc, argv);

a.setAttribute(Qt::AA\_UseHighDpiPixmaps);

a.setTheme("light");

a.setOrganizationName("deepin");

a.setApplicationName("dtk application");

a.setApplicationVersion("1.0");

a.setProductIcon(QIcon(":/images/logo.svg"));

a.setProductName("Dtk Application");

a.setApplicationDescription("This is a dtk template application.");

DMainWindow w;

w.setMinimumSize(500, 500);

w.show();

Dtk::Widget::moveToCenter(&w);

return a.exec();

}

在上面的代码中，DApplication对象还设置了一系列的程序、产品相关的属性，这些属性主要是在命令行参数、产品关于对话框中使用。

## 修改关于界面

点击标题栏中的菜单按钮，默认有两项内容，一个是帮助（关于）；另外一个是退出（Exit）。选择退出项目，程序会直接退出；如果点击关于选项，则会显示程序默认的关于对话框，如下图所示：

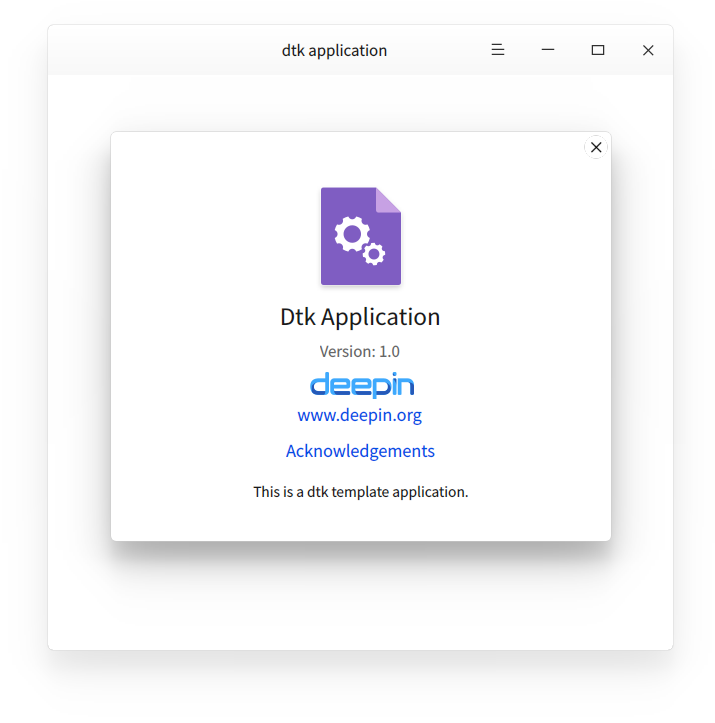


图 7.3 DTK程序关于对话框

关于对话框中的内容一般包含程序图标、版本号、开发商logo、官网地址、鸣谢以及程序描述等内容，应用开发者可以根据自己的需求进行修改和定制。

修改程序关于对话框内容有两种方式，一种方式是通过设置DApplication相关的属性进行修改，修改后的属性会被程序关于对话框识别并使用；另一种方式则是通过DAppliation获取程序的关于对话框DAboutDialog对象，然后再通过DAboutDialog类提供的相关函数接口进行信息的设置。

这里推荐使用第一种方式进行关于对话框信息的修改，虽然现在DTK中除了DAboutDialog还没有其他部分会使用这些信息，但是将来可能对此功能进行扩展，这些信息也可以被复用。

dtk-gallery作为本章的例子程序，主要是用来介绍DTK各个方面的使用，以及展示DTK中特殊控件的样式。根据这些信息，我们修改例子程序中设置关于对话框中相关属性，代码如下：

// 设置组织名称

a.setOrganizationName("deepin");

// 设置应用名称

a.setApplicationName("dtk-gallery");

// 设置应用版本号

a.setApplicationVersion("0.1");

// 设置应用图标

a.setProductIcon(QIcon(":/images/logo.svg"));

// 设置应用产品名称

a.setProductName("DTK Widgets Gallery");

// 设置应用描述信息

a.setApplicationDescription("DTK widgets gallery is an demo application to "

"demonstrate how DTK works.");

// 不显示鸣谢链接

a.setApplicationAcknowledgementVisible(false);

// 设置程序授权

1. setApplicationLicense("GPLv3");

修改后的关于对话框展示效果如下图所示：

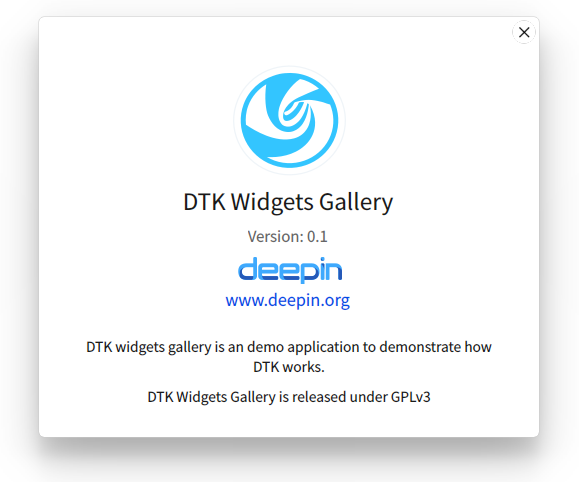


图 7.4 修改后的关于对话框

当然，除了修改代码以外，程序资源文件中的logo.svg文件也需要替换为程序图标的文件，本例为方便起见，直接使用deepin的logo作为程序的图标。

## 程序单实例

在构建深度桌面环境和众多应用的过程中，我们发现很多情况下，程序需要确保同时运行的实例只有一个，也就是我们平常所说的程序单实例，所以我们将程序单实例的功能融入了DTK中。因此，在使用DApplication的程序中实现单实例效果非常简单，只需要使用DApplication类的setSingleInstance方法即可：

DApplication a(argc, argv);

if (!a.setSingleInstance("dtk-gallery"))

return 0;

上面几行代码实现的效果为：如果当前程序已经有一个实例在运行，则DApplication::setSingleInstance函数会返回false，新实例直接退出程序。

如果程序需要感知新实例的启动，也可以监听DApplication::newInstanceStarted信号，在信号处理的函数中执行一定的动作，如常见的将主窗口置顶等。

## 日志文件

日志文件是记录程序运行过程中的一些关键日志信息的文件，是为了方便对程序运行过程中出现的一些问题进行回溯和追踪的一种调试手段。在Qt程序中，只提供了在终端打印日志和调试信息的工具，如果有将日志保存至文件的需求，则需要通过qInstallMessageHandler注册日志处理函数来进行处理。

为了方便使用以及统一日志文件的存放路径，我们在DTK中添加了日志文件处理的工具类DLogManager，用于对日志相关事项统一进行处理。

DLogManager类属于dtkcore模块，使用时需引入头文件DLog，并且使用宏DCORE\_USE\_NAMESPACE声明使用其所在命名空间。使用时只需在main函数中DApplication构造后添加如下代码即可：

// 设置终端日志和日志文件处理

DLogManager::registerConsoleAppender();

DLogManager::registerFileAppender();

编译运行程序，此时程序运行中使用qDebug等方法打印的日志除了会出现在终端上，也写入~/.cache/deepin/dtk-gallery/dtk-gallery.log文件。日志文件的路径主要取决于前面给DApplication设置的相关属性，如本例路径中的deepin为DApplication::organizationName属性，dtk-gallery则是DApplication::applicationName属性的值。

## 主窗口

一个程序的主窗口是图形应用程序的内容展示区域，因此显得尤为重要。向DMainWindow中添加内容的方式跟向QMainWindow中添加内容的方式并无不同，都是通过setCentralWidget设置主窗口的内容。

为了方便构建主窗口的“复杂”界面结构，我们一般会创建一个新类继承自DMainWindow，然后在这个类的基础上构建应用程序的主界面。在本例中，新建MainWindow类继承自DMainWindow，代码如下：

MainWindow::**MainWindow**()

: DMainWindow()

{

// 构建主窗口界面内容

initUI();

// 初始化信号槽连接

initConnections();

}

void MainWindow::**initUI**()

{

// 设置主窗口内容

QFrame \*content = new QFrame;

content->setStyleSheet("background: red");

setCentralWidget(content);

}

void MainWindow::**initConnections**()

{

}

代码中的initUI和initConnections函数是DTK程序中约定成俗的一种“模式”：在纯界面的类代码中，initUI主要用于构建图形界面；initConnections则主要用于初始化控件信号和槽函数的连接。

在MainWindow类的initUI函数中，我们创建了一个QFrame对象，用作主窗口的内容控件，并使用setCentralWidget函数设置给了MainWindow。为了更加清晰地展示主窗口内容区域，给content对象设置了背景色为红色的样式，运行效果如下图所示：



图 7.5 主窗口内容区域

## 自定义标题栏

在DTK风格的应用程序中，一般主窗口都包含一个标题栏。严格意义上来说，标题栏也属于主窗口的内容，而不仅仅是上面例子程序中标红的部分。不过，对于一般的程序来说，主要关注的是非标题栏区域所展示的界面内容，所以我们一般说主窗口的区域即指上面标红的那部分区域。

主窗口包含的标题栏在DTK中使用DTitlebar类来表示。DTitlebar本身具有非常高的定制性，甚至可以将一些控件摆在标题栏上，而不仅仅只是用来显示窗口图标和标题。因此，在DTK风格的程序中，为了方便用户快捷操作，程序往往会对标题栏进行自定义，放置一些快捷操作控件在标题栏上（如深度操作系统的看图应用）。

在dtk-gallery例子程序中，为了向应用开发者更清晰地展示DTK中的控件和类，主窗口区域需要分两个部分进行切换展示：Controls和Effects。其中Controls部分主要展示DTK中特有的控件；Effects部分主要展示DTK中辅助实现特殊效果的一些类。因此，为了方便用户快速进行界面切换，我们将在标题栏上放置一个类似标签页的控件DSegmentedControl。

修改MainWindow类的代码，在其initUI中添加如下代码：

m\_seg = new DSegmentedControl;

// 为DSegmentedControl中添加选项

m\_segControlsIndex = m\_seg->addSegmented("Controls");

m\_segEffectsIndex = m\_seg->addSegmented("Effects");

// 在标题栏中显示DSegmentedControl控件

titlebar()->setCustomWidget(m\_seg, true);

并在其initConnections函数中，添加如下代码：

connect(m\_seg, &DSegmentedControl::currentChanged, this, [this] (int index){

qDebug() << m\_seg->getText(index);

});

经过上面的修改后，运行程序展示效果如下：

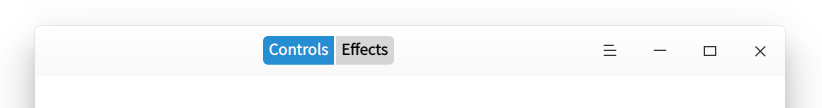


图 7.6 用DSegmentedControl自定义标题栏

点击控件中的选项，程序就会在日志中打印对应选项的标题。

## DTK中的控件

### Controls页面

在Controls页面，dtk-gallery程序将会展示DTK中特有的图形控件，这部分界面内容如下图所示：

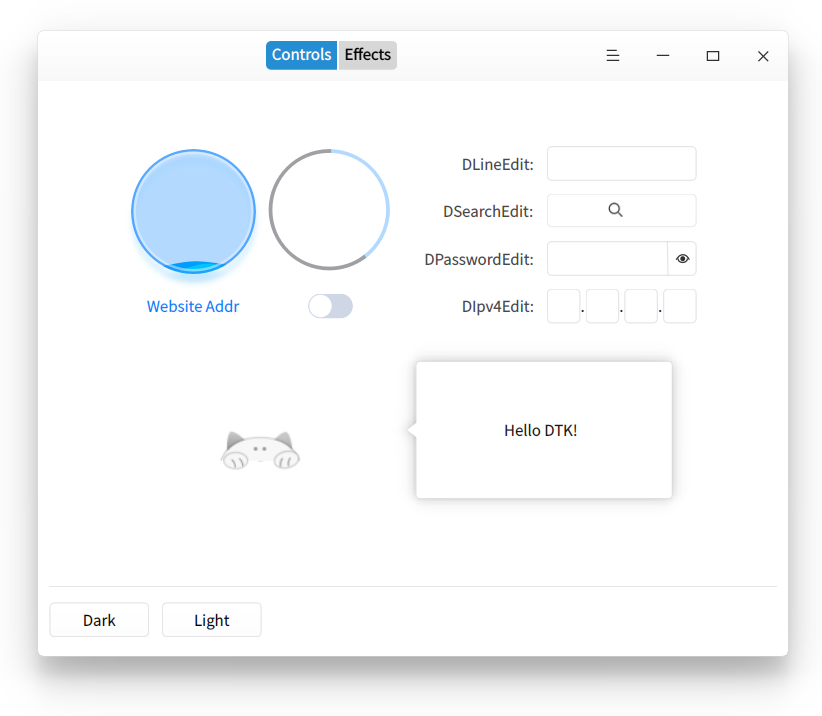


图 7.7 Control页面

下面会对上图中涉及到的控件分别进行说明。

#### 进度控件

DTK中用来展示进度的控件主要有两个：DWaterProgress和DCircleProgress，即上图中两个圆形控件，左边为DWaterProgress，右边为DCircleProgress。两者在主要功能使用上并无太多不同：创建控件实例后，使用setValue即能修改其进度值。以下代码将两个控件的进度设置为10%。

m\_waterProgress->setValue(10);

m\_circleProgress->setValue(10);

不同的是，DWaterProgress为动态显示的控件，在调用了DWaterProgress::start方法以后控件即开始水纹波动的动画；DCircleProgress则具有一定的样式定制性，总量环形和当前进度的部分可以分别设置颜色。

// 设置总量环形颜色为灰色

m\_circleProgress->setBackgroundColor(Qt::gray);

// 设置当前进度部分颜色为浅蓝色

m\_circleProgress->setChunkColor(QColor("#B3D9FE"));

以上代码将DCircleProgress进度控件的总量环形设置颜色为灰色，同时设置当前进度部分的颜色为浅蓝色。

#### 按钮控件

进度控件下面的三个控件为DTK特有的按钮控件，它们分别是DLinkButton、DSwitchButton和DImageButton。

其中，DLinkButton继承自QPushButton，除了默认样式不同外，其余的功能基本一致。在目前的DTK版本中，使用DLinkButton时，只能设置按钮的展示文字，如果需要点击按钮打开链接，需要监听按钮的clicked信号，在槽函数中打开相应链接，如下面代码所示：

// DLinkButton点击的时候使用QProcess执行命令打开特定网址

connect(m\_linkButton, &DLinkButton::clicked, this,

[this] ()

{

QProcess::startDetached("xdg-open https://www.deepin.org");

m\_message->setText("Link opened!");

});

DSwitchButton是一个包含两种状态（checked和非checked）的开关类控件，在使用时通过setChecked设置其状态，通过监听checkedChanged信号获取其状态切换的时机，如下代码所示：

// DSwitchButton打开的时候设置两个进度条的进度为50%，否则设置为0%

connect(m\_switchButton, &DSwitchButton::checkedChanged, this,

[this] (bool checked)

{

int progress = checked ? 50 : 0;

m\_waterProgress->setValue(progress);

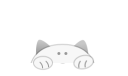
m\_circleProgress->setValue(progress);

m\_message->setText(QString("Progress set to %1%").arg(progress));

});

DImageButton控件则是纯图像显示的按钮，通过给这个按钮设置三种状态（normal、hover、pressed）需要使用的图片，按钮根据鼠标的动作自动切换状态并更新图片显示，适合实现表现比较随意、灵活的按钮。

在dtk-gallery程序中，DImageButton的展现形态为一只小猫，正常状态下小猫的头为缩回状态；鼠标移动到按钮上小猫的头会漏出一点；如果点击按钮，则小猫会将头完全伸出。这个过程如下图所示：



normal

hover

pressed

图 7.8 DImageButton实现的小猫动画

设置DImageButton的代码如下：

// 为DImageButton设置三种状态的图片资源

m\_catButton->setNormalPic(":/images/cat\_normal.png");

m\_catButton->setHoverPic(":/images/cat\_hover.png");

m\_catButton->setPressPic(":/images/cat\_pressed.png");

如果要在按钮点击时时做逻辑处理，则同样监听按钮的clicked信号，在本例中我们在按钮点击时显示消息“Moew~~~”，代码如下：

// DImageButton点击的时候，在右侧的信息框中显示提示信息

connect(m\_catButton, &DImageButton::clicked, this,

[this]()

{

m\_message->setText("Meow~~~");

});

#### 输入控件

为了实现输入框在表现形式以及行为上的一些特殊需求，DTK引入了自己的输入控件：DLineEdit、DSearchEdit、DPasswordEdit和DIpv4LineEdit。

DLineEdit在QLineEdit的基础上加入了警告，即可以通过setAlert方法设置DLineEdit的警告状态，并且通过showAlertMessage显示警告信息。如下代码在输入完成时检查输入框内容，如果为空则提供用户输入不能为空，并且在用户重新输入的时候撤销警告状态：

// DLineEdit编辑完成时，在右侧信息框中显示输入内容

connect(m\_lineEdit, &DLineEdit::editingFinished, this, [this] {

if (m\_lineEdit->text().length() != 0) {

m\_message->setText("Typed: " + m\_lineEdit->text());

} else {

//警告用户输入不能为空

m\_lineEdit->setAlert(true);

m\_lineEdit->showAlertMessage("Can't be empty");

}

});

// 用户重新编辑时，撤销警告状态

connect(m\_lineEdit, &DLineEdit::textChanged, this, [this] {

m\_lineEdit->setAlert(false);

});

另外三个类型的输入框控件DSearchEdit、DPasswordEdit和DIpv4Edit除了表现形式与QLineEdit不同外，其他用法大同小异，此处不再详细介绍。本例中在输入控件编辑完成后显示相应内容的代码如下：

// DLineEdit编辑完成时，在右侧信息框中显示输入内容

connect(m\_searchEdit, &DSearchEdit::editingFinished, this, [this] {

m\_message->setText("Searched: " + m\_searchEdit->text());

});

// DPasswordEdit编辑完成时，在右侧信息框中显示输入内容

connect(m\_passwordEdit, &DPasswordEdit::editingFinished, this, [this] {

m\_message->setText("Guess what, I know your password: \n" + m\_passwordEdit->text());

});

// DIpv4LineEdit编辑完成时，在右侧信息框中显示输入内容

connect(m\_ipv4Edit, &DIpv4LineEdit::editingFinished, this, [this] {

m\_message->setText("I got IP address: " + m\_ipv4Edit->text());

});

#### 其他控件

DTK中还有一些比较特殊的控件，如类似安卓开发中Toast的DToast提醒控件和类似聊天消息框形态的DArrowRectangle控件等。

DToast控件在使用时比较简单，只需将控件移动到特定的位置，调用pop方法显示进行显示：

// 设置DToast的提示信息内容

m\_toast->setText("Welcome back");

// 移动DToast的位置到窗口中下方的位置并显示

m\_toast->move((width() - m\_toast->width()) / 2.0,

height() - 20 - m\_toast->height());

m\_toast->pop();

显示效果如下图所示，其中显示“Welcome back”字样的就是DToast控件：

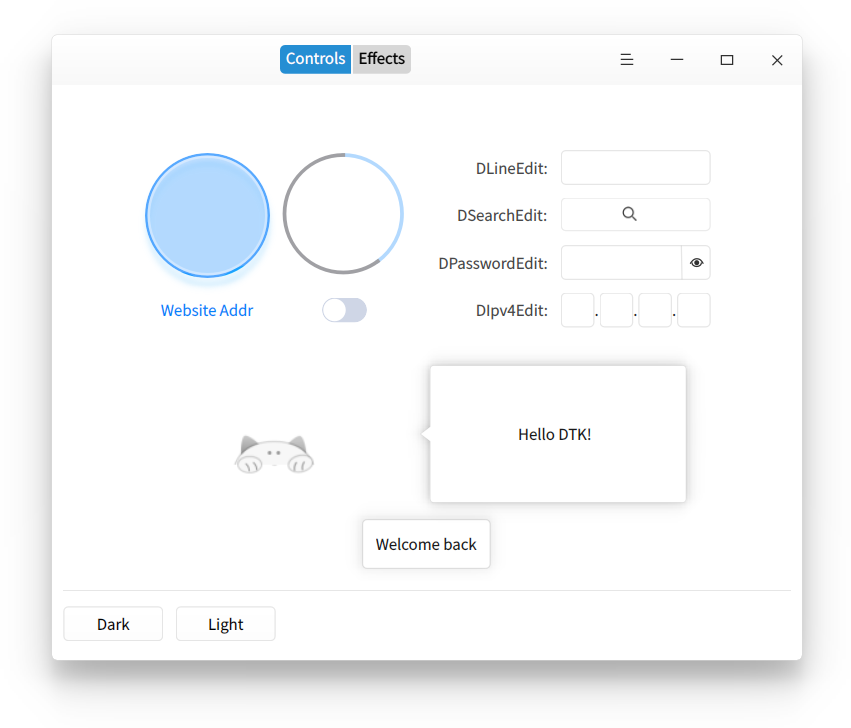


图 7.9 DToast效果

但是，在使用DToast时需要注意的是DToast在创建时需设定父控件，否则控件会直接以窗口状态显示。

DArrowRectangle控件一般用来构造如上面提到的DLineEdit的警告提醒、Popover提醒类窗口等，接口比较贴近底层，一般不会使用到，这里不再详述。

### Effects页面

在Effects页面，总共展示了四种类型的特殊效果，从左到右、从上到下分别为DBlurEffectWidget窗口内混合模式、DBlurEffectWidget窗口背景混合、DGraphicsClipEffect和DGraphicsGlowEffect。

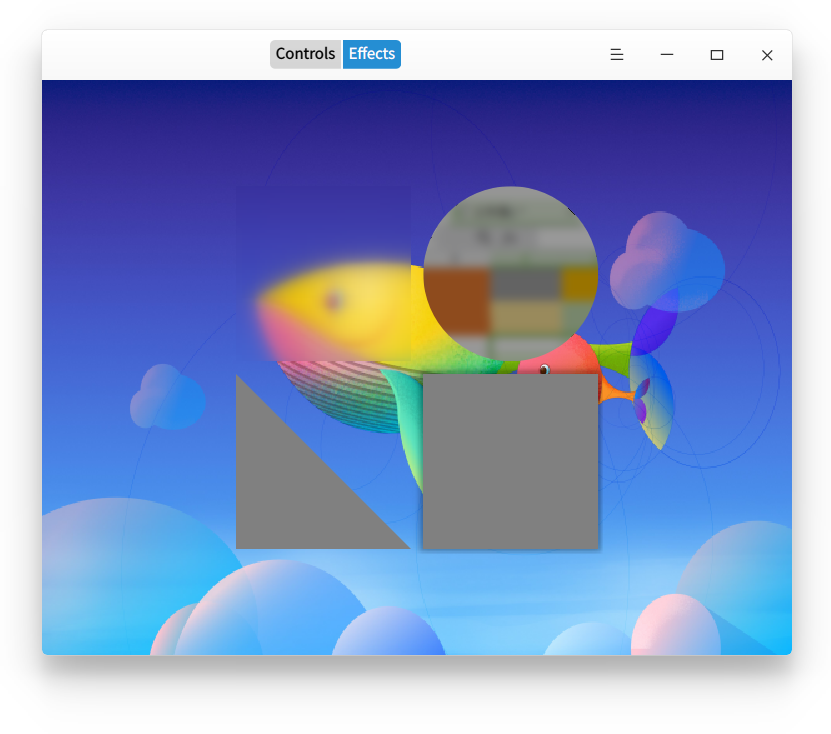


图 7.10 Effects页面

其中，DBlurEffectWidget主要用来实现背景模糊的功能，它本身包含两种混合模式：窗口内混合，即控件跟窗口内容模糊、混合后显示；窗口背景混合，即控件跟窗口背后的内容模糊、混合后显示。本例子实现效果代码如下：

// DBlurEffectWidget窗口内混合模式

m\_blurInner->setFixedSize(140, 140);

m\_blurInner->setBlendMode(DBlurEffectWidget::InWindowBlend);

// 设置模糊半径

m\_blurInner->setRadius(30);

// DBlurEffectWidget窗口背景混合模式

m\_blurBackground->setFixedSize(140, 140);

m\_blurBackground->setBlendMode(DBlurEffectWidget::BehindWindowBlend);

// 设置遮罩层颜色为暗色

m\_blurBackground->setMaskColor(DBlurEffectWidget::DarkColor);

// 设置控件X和Y方向的圆角半径

m\_blurBackground->setBlurRectXRadius(70);

m\_blurBackground->setBlurRectYRadius(70);

DGraphicsClipEffect和DGraphicsGlowEffect都继承自QGraphicsEffect，兼容QWidget::setGraphicsEffect函数，使用起来非常方便：

// 创建切割路径

QPainterPath path;

path.moveTo(0, 0);

path.lineTo(0, 140);

path.lineTo(140, 140);

path.closeSubpath();

// 设置切割路径

m\_clipEffect->setClipPath(path);

// 设置外发光X轴的偏移量

m\_glowEffect->setXOffset(10);

// 设置外放光Y轴的偏移量

m\_glowEffect->setYOffset(10);

QFrame \*cliped = new QFrame;

cliped->setStyleSheet("background: grey");

cliped->setFixedSize(140, 140);

cliped->setGraphicsEffect(m\_clipEffect);

QFrame \*glowing = new QFrame;

glowing->setStyleSheet("background: grey");

glowing->setFixedSize(140, 140);

glowing->setGraphicsEffect(m\_glowEffect);

## 切换主题

早在Deepin UI的时代，深度控件库就开始提供换肤功能，每个使用Deepin UI的程序都可以自由的进行风格自定义。DTK同样提供切换主题风格的功能，并且内置两款主题：浅色风格的light和深色风格的dark，DApplication默认使用light风格的主题。

为了演示主题切换的功能，我们对gtk-gallery项目进行修改，增加两个按钮“Light”和“Dark”，点击“Light”按钮即将程序切换为浅色风格，点击“Dark”按钮将程序切换为深色风格，代码如下：

connect(m\_darkMode, &QPushButton::clicked, this, [this] () {

DApplication \*app = qobject\_cast<DApplication\*>(qApp);

if (app) {

app->setTheme("dark");

}

});

connect(m\_lightMode, &QPushButton::clicked, this, [this]() {

DApplication \*app = qobject\_cast<DApplication\*>(qApp);

if (app) {

app->setTheme("light");

}

});

从上面的代码可以看到，DApplication提供了setTheme方法用来设置程序的主题风格，使用起来非常简单。切换至深色风格的dtk-gallery显示效果如下图所示：

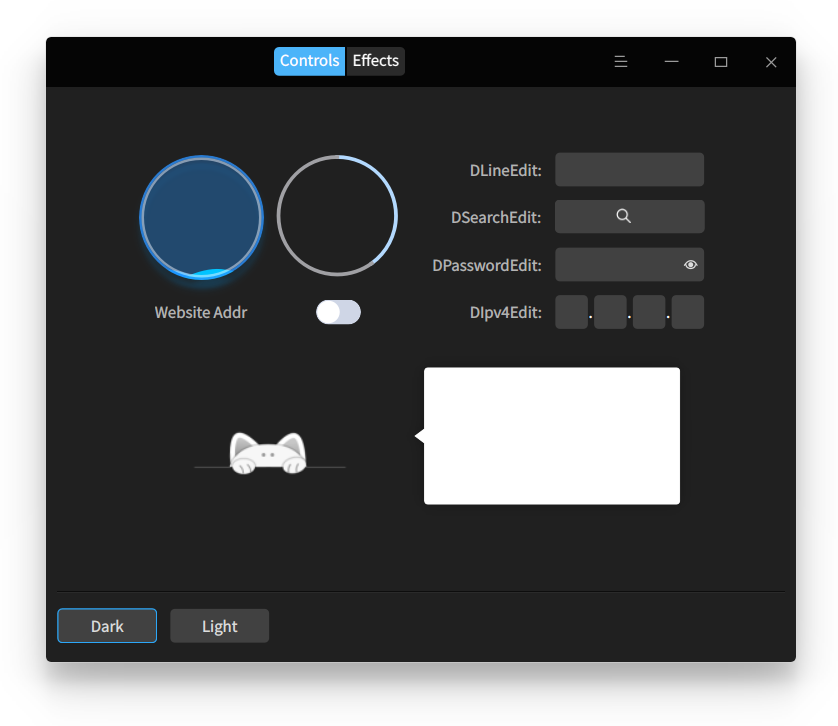


图 7.11 暗色主题样式

## 添加设置窗口

在所有深度系列应用中，窗口标题栏都提供了程序选项按钮，点击这个按钮会弹出一个菜单，并且包含一个设置菜单项，用户点击设置菜单项就能打开程序的设置窗口。这是所有DTK风格应用程序需要遵循的设计风格。本小节会就此内容进行说明，并通过修改dtk-gallery程序演示如何给应用添加设置窗口，以保证应用跟其他DTK风格应用保持一致。

首先，在标题栏的程序选项菜单中添加设置菜单项，代码如下：

QMenu\* MainWindow::**createSettingsMenu**()

{

QMenu \*menu = new QMenu;

QAction \*settingsAction = menu->addAction("Settings");

connect(settingsAction, &QAction::triggered, this, [this] {

m\_settingsDialog->show();

});

return menu;

}

// 给标题栏添加设置菜单

QMenu \*settingsMenu = createSettingsMenu();

titlebar()->setMenu(settingsMenu);

createSettingsMenu用来创建选项菜单，里面包含程序选项菜单中除“关于”、“退出”两个菜单项以外的所有菜单项，即本例中的“Settings”菜单项。“关于”、“退出”以及之后要提到的“帮助”菜单项由DTitlebar自动创建和添加。

创建好菜单后，通过DTitlebar::setMenu方法设置此菜单为程序选项菜单。此时点击程序选项按钮，显示菜单内容如下图所示：

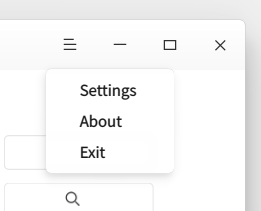


图 7.12 添加设置菜单项

为了方便应用开发者构建设置窗口界面，DTK开发了一套通过json文件创建设置界面的机制。在构建设置窗口界面的时候，应用开发者无需再一行代码一行代码构建界面，而是通过一个json文件描述设置界面内容，经过DSettings加载，传递给DSettingsDialog构建界面即可：

SettingsDialog::**SettingsDialog**(QWidget \*parent)

: DSettingsDialog(parent)

{

// 构造配置文件路径

const QString confDir = DStandardPaths::writableLocation(

QStandardPaths::AppConfigLocation);

const QString confPath = confDir + QDir::separator() + "dtk-gallery.conf";

// 创建设置项存储后端

QSettingBackend \*backend = new QSettingBackend(confPath);

// 通过json文件创建DSettings对象

DSettings \*settings = DSettings::fromJsonFile(":/data/settings.json");

// 设置DSettings存储后端

settings->setBackend(backend);

// 通过DSettings对象构建设置界面

updateSettings(settings);

connect(settings, &DSettings::valueChanged,

this, &SettingsDialog::settingsChanged);

}

在上面的代码中，我们新建了一个类SettingsDialog，继承自DSettingsDialog，用于dtk-gallery应用的设置界面。

其中，DSettings需要一个存储后端用来存放设置数据，DTK提供了两个可选选项，一个是上例子中用到的QSettingsBackend，另外一个是GSettingsBackend。从名字上可以看出，QSettingsBackend是使用QSettings作为存储后端，而GSettingsBackend是使用GSettings作为存储后端。GSettings相较于QSettings，优点在于可以有系统和用户级两套配置，在系统定制时比较方便，而且其本身有一套通知机制；但是，缺点也比较明显，就是它在使用上比较麻烦。本例为了方便演示使用QSettings作为存储后端。

存储后端构建完成后，通过DSettings::setBackend设置给DSettings对象，DSettingsDialog则通过DSettingsDialog::updateSettings方法加载DSettings的内容。

DSettings从json文件加载内容时，则是使用DSettings::fromJsonFile方法，在本例中settings.json文件的内容如下：

{

"groups": [

{

"key": "basic",

"name": "Basic",

"groups": [

{

"key": "select\_multiple",

"name": "Checkbox",

"options": [

{

"key": "checkbox\_1",

"text": "Check list 1",

"type": "checkbox",

"default": 1

},

{

"key": "checkbox\_2",

"text": "Check list 2",

"type": "checkbox"

}

]

},

{

"key": "select\_single",

"name": "Radiogroup",

"options": [

{

"key": "radiogroup",

"name": "",

"type": "radiogroup",

"items": [

"Option 1",

"Option 2"

],

"default": 1

}

]

},

{

"key": "slider",

"name": "Sliders",

"options": [

{

"key": "slider",

"type": "slider",

"name": "Opacity",

"min": 0,

"max": 100,

"default": 50

}

]

}

]

},

{

"key": "advanced",

"name": "Advanced",

"groups": [

{

"key": "combo",

"name": "Combobox",

"options": [

{

"key": "combobox",

"name": "Combobox",

"type": "combobox",

"items": [

"hello", "world"

],

"default": "hello"

}

]

},

{

"key": "spin",

"name": "SpinButton",

"options": [

{

"key": "spin",

"name": "Change the value",

"type": "spinbutton",

"default": 10

}

]

},

{

"key": "shortcuts",

"name": "Shortcuts",

"options": [

{

"key": "open\_file",

"name": "Open file",

"type": "shortcut",

"default": "Ctrl+o"

},

{

"key": "open\_folder",

"name": "Open folder",

"type": "shortcut",

"default": "Ctrl+f"

}

]

}

]

}

]

}

生成设置界面内容如下：

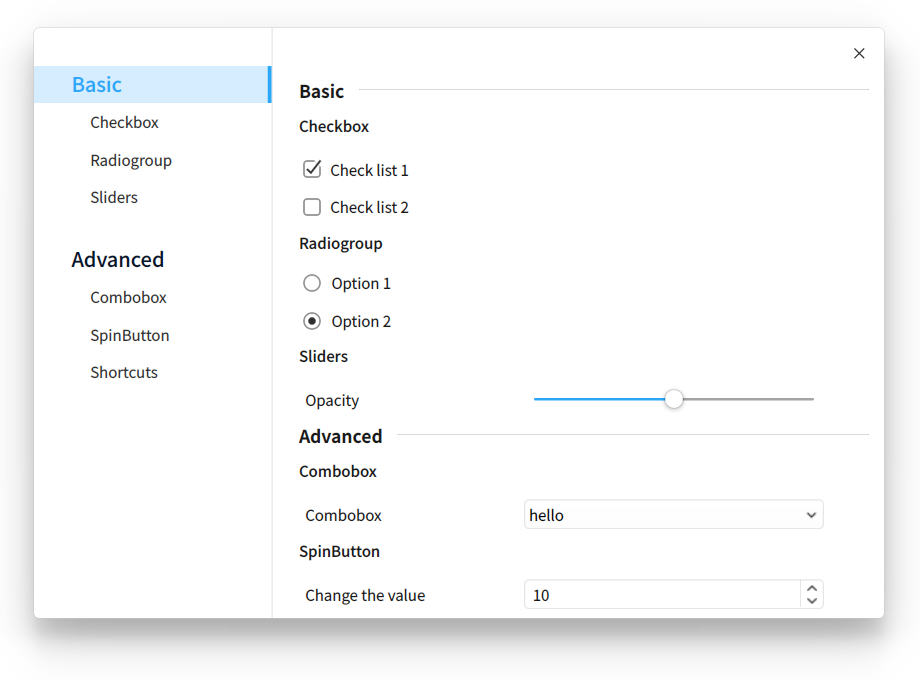


图 7.13 生成设置界面1

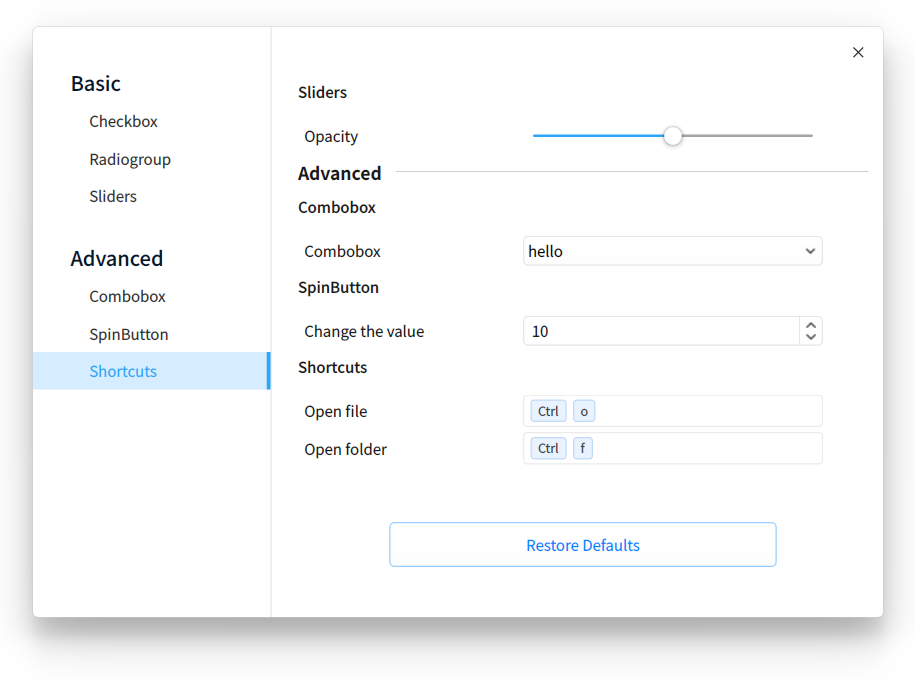


图 7.14 生成设置界面 2

由DSettings生成的设置对话框分为左右两个部分，左侧是导航栏，右侧为设置内容，在设置内容最下方是重置按钮，用户可以通过这个按钮将程序所有的设置项重置。

json文件最外层是一个无名对象，包含属性“groups”，这是一级导航，例如图中的“Basic”；一级导航下层是二级导航，例如图中的“Checkbox”，二级导航在json文件中以一个对象的形式出现，包含属性“key”、”name”和”options“。其中，”key”为字符串，表示设置项目的唯一ID，json文件中除最顶层的对象外其余的对象均需设置”key“属性；”name“属性同样为字符串，表示标题的名称；”options“属性为数组，包含当前二级标题下的所有设置项。每个设置项都从属于一个特定的二级导航，每个二级导航都从属于一个一级导航。

每个设置项可以对应于不同的控件类型，在设置项的json对象中使用”type“属性来标识，DSettings目前支持的”type“属性类型和其对应生成的控件关系如下表所示：

表 7.2 DSettings支持类型表

|  |  |  |
| --- | --- | --- |
| type字段 | 生成控件 | json属性 |
| checkbox | QCheckBox | text |
| radiogroup | QRadioButton组成的QButtonGroup | items |
| slider | QSlider | name、min、max |
| combobox | QComboBox | name、items |
| spinbutton | QSpinBox | name |
| shortcut | ShortcutEdit | name |
| lineedit | QLineEdit | name、text |

除了上面列出的属性外，每个控件可选提供一个”default“属性，表示设置项默认值，用于重置按钮点击后的设置重置处理，本书建议所有控件都设置一个默认值。

## 添加帮助手册

前面提到程序选项菜单中”帮助“一项同”关于“、”退出“都是DTitlebar自动添加的，但是dtk-gallery到目前为止还没有出现这一项菜单，这主要是因为我们还没有给dtk-gallery添加帮助手册内容。

为了让DTitlebar能识别到针对你应用的帮助手册，你需要将应用程序的帮助手册内容放置在/usr/share/deepin-manual/manual/下，并且保证手册目录的名称跟你应用的程序名称对应（即前面通过DApplication::setApplicationName设置的名称）。

以dtk-gallery为例，你需要编写合适的手册内容，并将其放置在/usr/share/deepin-manual/manual/dtk-gallery/目录下。此时点击程序选项按钮，弹出的菜单就会包含”Help“一项，如下图所示：

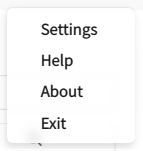


图 7.15 帮助菜单项

点击”Help“菜单项，会使用帮助手册应用打开dtk-gallery的帮助手册内容。帮助手册文档的格式见deepin-manual项目wiki，此处限于篇幅不再展开描述。

## 编写任务栏插件

在第6章的内容中，我们曾介绍过使用Qt提供的插件机制来扩展程序的功能。深度桌面环境的任务栏即提供插件机制，任务栏通知区域中的很多项目都是透过插件机制开发的。外部的开发者也可以通过提供任务栏插件来增强任务栏的功能。

由于任务栏常驻用户界面的特点，也比较适合用于功能扩展，常见的任务栏插件类型主要有天气预报、资源监控、状态指示等。

### 插件加载

任务栏启动时，会先启动一个线程去检测/usr/lib/任务栏/plugins目录下的所有文件，并检测每个文件是否是一个正常的动态库文件，如果是则尝试作为插件加载。尝试加载主要分为两个方面：

1. 检查插件的版本等元数据信息；
2. 检查动态库是否实现了任务栏插件要求的接口。

其中，第1点由任务栏完成，主要是通过定义的接口版本号等静态信息，对插件进行过滤，避免插件加载导致的任务栏崩溃等问题；第2点则由Qt插件框架处理，这部分主要保证插件实现了任务栏定义的接口，以及依赖的其他符号都存在等。

如果插件加载成功，任务栏会调用插件的初始化接口进行资源初始化，获取插件的ID、名称、图标、控件等将插件“安装”在任务栏上。

### 插件接口

任务栏主要提供了两个接口类，作为任务栏和插件的交互接口：

* PluginItemInterface 中包含插件需要实现、重写的函数；
* PluginProxyInterface 中包含任务栏主体提供给插件调用的函数集合；

两个接口类中包含的接口见下面两个表格的说明：

表 7.3 任务栏插件PluginItemInterface接口说明表

|  |  |  |
| --- | --- | --- |
| 名称 | 说明 | 必须实现 |
| pluginName | 返回插件名称，用于在任务栏内部管理插件时使用 | ✓ |
| pluginDisplayName | 返回插件名称，用于在界面上显示 |  |
| init | 插件初始化入口函数 | ✓ |
| itemWidget | 返回插件主控件，用于显示在任务栏面板上 | ✓ |
| itemTipsWidget | 返回鼠标悬浮在插件主控件上时显示的提示框控件 |  |
| itemPopupApplet | 返回鼠标左键点击插件主控件后弹出的控件 |  |
| itemCommand | 返回鼠标左键点击插件主控件后要执行的命令数据 |  |
| itemContextMenu | 返回鼠标右键点击插件主控件后要显示的菜单数据 |  |
| invokedMenuItem | 菜单项被点击后的回调函数 |  |
| itemSortKey | 返回插件主控件的排序位置 |  |
| setSortKey | 重新设置主控件新的排序位置（用户拖动了插件控件后） |  |
| itemAllowContainer | 返回插件控件是否允许被收纳 |  |
| itemIsInContainer | 返回插件是否处于收纳模式（仅在 itemAllowContainer 为 true 时有作用） |  |
| setItemIsInContainer | 新插件是否处于收纳模式的状态（仅在 itemAllowContainer 主 true 时有作用） |  |
| pluginIsAllowDisable | 返回插件是否允许被禁用（默认不允许被禁用） |  |
| pluginIsDisable | 返回插件当前是否处于被禁用状态 |  |
| pluginStateSwitched | 当插件的禁用状态被用户改变时此接口被调用 |  |
| displayModeChanged | 任务栏 显示模式发生改变时此接口被调用 |  |
| positionChanged | 任务栏位置变化时时此接口被调用 |  |
| refreshIcon | 当插件控件的图标需要更新时此接口被调用 |  |
| displayMode | 用于插件主动获取任务栏当前的显示模式 |  |
| position | 用于插件主动获取任务栏当前的位置 |  |

表 7.4 任务栏插件PluginProxyInterface接口说明表

|  |  |
| --- | --- |
| 名称 | 说明 |
| itemAdded | 向任务栏添加新的主控件（一个插件可以添加多个主控件它们之间使用ItemKey区分） |
| itemUpdate | 通知任务栏有主控件需要更新 |
| itemRemoved | 从任务栏移除主控件 |
| requestWindowAutoHide | 设置任务栏是否允许隐藏，通常被用在任务栏被设置为智能隐藏或始终隐藏而插件又需要让任务栏保持显示状态来显示一些重要信息的场景下 |
| requestRefreshWindowVisible | 通知任务栏更新隐藏状态 |
| requestSetAppletVisible | 通知任务栏显示或隐藏插件的弹出面板（鼠标左键点击后弹出的控件） |
| saveValue | 统一的配置保存函数 |
| getValue | 统一的配置读取函数 |

## 编写文件管理器插件

深度文件管理器作为一款文件管理工具，常常遇到的需求是针对一个或者多个选中的文件做执行特定的操作，所以我们开放了上下文菜单的接口，方便用户对菜单进行自定义扩展。

文件管理器的上下文菜单可以通过两种方式进行扩展：json形式和插件形式，因为json形式的上下文菜单实现起来比较简单、更为使用，这里只针对使用json形式扩展文件管理器的上下文菜单做出说明。插件形式的上下文菜单如果以后开放更多接口，会在深度的技术博客（https://docs.deepin.io）进行更新说明。

默认情况下，文件管理器会在启动时尝试加载~/.config/deepin/dde-file-manager/menuextensions/目录下的json文件作为上下文菜单扩展， 对于一个有效的上下文菜单扩展，内容应该是一个菜单对象的列表，每个菜单对象应当包含有如下的通用字段：

表 7.5 上下文菜单插件json字段说明表

|  |  |
| --- | --- |
| 字段 | 说明 |
| MenuType | 菜单类型，指定菜单项出现的时机。可选类型为：SingleFile（选择单个文件时出现）；MultiFiles（选择多个文件时出现）；SingleDir（选择单个目录时出现）；MultiDirs（选择多个目录时出现）；MultiFileDirs（选择多个文件和目录时出现）；EmptyArea（在空白区域点击时出现） |
| Icon | 菜单项图标，图标的绝对路径，可选。 |
| Text | 菜单项文字，支持Text[local]形式的国际化。 |
| Exec | 点击菜单项会执行的可执行文件，不包含参数。 |
| Args | 执行Exec字段命令需要的参数，形式如[“arg1”, “arg2”] |
| SubMenu | 子菜单项，可以无限嵌套菜单对象。 |
| MimeType | 过滤文件的mimetype，只有在MenuType为SingleFile或者MultiFiles时有效，形式如MimeType:text/plain;image/jpeg |
| Suffix | 过滤文件的后缀名，同样只有在MenuType为SingleFile或者MultiFiles时有效，形式为Suffix: md;txt |

下面的例子为文件管理器的上下文菜单添加了一个“使用VSCode打开”的菜单项，用户点击这个菜单项后，所选中的文件会在VSCode中打开：

[

{

"MenuType": "EmptyArea",

"Text[zh\_CN]": "在 VSCode 中打开",

"Text[en\_US]": "Open with VSCode",

"Exec": "code",

"Args": ["."]

}

]

生成菜单项如下图所示：

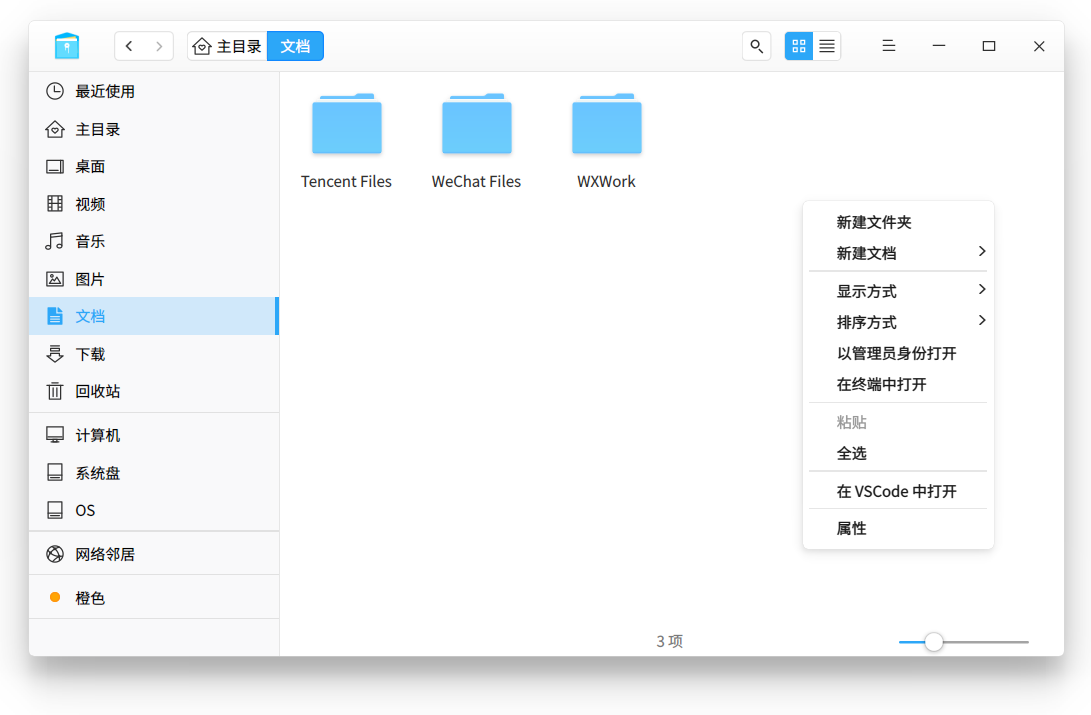


图 7.16 上下文菜单插件添加的菜单项

## 小结

本章介绍了DTK开发框架的历史和由来，并通过dtk-gallery例子程序对DTK中的方方面面进行了讲解和说明，包括程序创建、设置单实例、添加日志到控件使用，以及切换主题、生成设置窗口界面等特性。另外，本章也对广义DTK中包含的任务栏插件和文件管理器插件的接口和编写进行了介绍。

## 练习题

为文件管理器添加上下文菜单插件，增加菜单项，实现功能和要求如下：

* 点击菜单项，弹出图形界面，开始对选中单个文件夹进行扫描，扫描内容为超过500M大小的大文件；
* 扫描过程需有图形界面展示，扫描进度等；
* 扫描完成后，使用列表展示扫描结果。