# 优化Qt应用软件

在前面的几个章节里面，我们介绍了如何使用Qt开发图形界面应用、以及如何使用Qt开发程序的一些扩展功能，但是在实际的开发过程中，除了满足功能性的要求，我们对应用程序还有更多方面的要求。本章我们会分别介绍Qt的插件系统、Qt单元测试和polkit鉴权系统等方面的内容，并就这几个方面进行展开，介绍如何在Qt开发的过程中进行优化和设计，满足程序在扩展性、安全性和可靠性等方面的要求。

## Qt插件系统

插件作为扩展程序最常见的方式之一，在软件开发的过程中有着极大的便利，它带来的好处主要有：

1. 扩展性，程序可以动态地得到扩展功能，而无需重新开发和编译；
2. 模块化，主程序和插件之间主要通过接口进行沟通，保证了两个模块间尽可能少的耦合，有利于程序架构设计更加清晰；
3. 合理分工，主程序的开发者和每个插件的开发者可以是不同的人和团队，有利于并行协作和合理分工。

基于以上几点优势，一些比较大型的项目如Chromium、Firefox、甚至Linux内核都提供了各式各样的插件机制。

Qt作为一个功能丰富的跨平台开发库，也提供非常方便的扩展接口，为应用软件的开发和扩展提供了便利，这主要表现在两个方面：

1. Qt提供了一套完整的插件开发框架

这套框架包含了插件识别、元数据提供、插件加载、插件销毁等各个方面，应用开发者可以根据自己的需要对应用中需要扩展的部分进行封装，灵活根据需要进行插件的加载和编排，从而实现扩展程序的目的。

例如deepin系统的任务栏，定义了托盘插件的接口，系统设置相关的托盘由任务栏自身提供，磁盘挂载的插件则由文件管理器提供，这样既满足了系统挂载插件显示在任务栏的需求，也可以保证任务栏项目中不需要混入挂载相关的大量代码。

1. Qt本身支持使用插件进行扩展

Qt本身很多功能，包括数据库驱动、图像格式、文本编码、自定义风格、甚至跨平台支持等，都是基于插件完成的，这些插件接口在1）的基础上进行了一定的封装，定义了不同类型插件需要提供的不同接口，通过这些插件接口，开发者可以方便地开发各种各样的扩展功能给所有基于Qt的应用软件。

再以deepin系统中使用的输入法框架Fcitx为例，它就是通过Qt的输入法插件为基于Qt的应用程序提供统一的输入法。

对于应用开发者来说，主要使用到的是Qt提供的插件开发框架，以实现应用功能的扩展，因此，对于扩展Qt本身的插件我们这里不再做详细介绍。另外，需要说明的是Qt提供静态插件和动态插件两种使用方式，分别对应于静态库和动态库，这里提到的插件主要是指动态插件。

### 使用

Qt的插件系统使用起来非常方便，相较于使用纯粹的C++动态库来说，无需太多关心底层的符号加载、查找等，另外，插件通过提供额外的基于json的元数据，可提供强大的静态数据信息。

创建和使用插件，从主程序和插件两个角度看，分别有不同的工作需要完成，对于主程序来说，需要：

1. 定义接口类；
2. 通过Q\_DECLARE\_INTERFACE宏来声明插件接口；
3. 使用QPluginLoader加载插件，加载成功后通过QPluginLoader::instance()来获取插件实例；
4. 最后，通过qobject\_cast来将加载的插件实例转换成需要的接口类型进行使用。

其中，需要注意的是接口类必须是纯虚类，否则在插件加载的过程中可能出现“符号未定义”的错误。

主程序以上的工作完成以后，对于插件来说，需要完成的工作有：

1. 创建一个基于QObject的类，同时实现插件接口中定义的函数；
2. 使用Q\_INTERFACES宏声明插件类实现了哪些已经定义的接口；
3. 同样地，在插件类中，使用Q\_PLUGIN\_METADATA宏声明插件的元数据信息（元数据以json文件的方式提供）。

可以看到，在一个插件系统中，接口是主程序和插件之间用来交互的桥梁，也是一个插件系统的核心。接口最初定义得够合理，接口才容易保持稳定，插件开发才能发挥真正的力量；相反，如果插件接口更换比较频繁，插件的开发者会逐渐失去耐心，导致插件系统失去实际的意义。

了解了以上信息，下面会创建一个例子程序，以此来说明一下如何在实际的应用场景用使用到Qt的插件技术。

### 实例

在这个例子中，我们将制作一个非常简单的“画板”程序，这个画板程序本身只提供画布和画笔，实际的绘画操作则由插件来完成。用户运行程序后，程序首先显示一个背景为黑色的窗口，用户可以通过右键弹出菜单，选择图形画笔进行图形绘制。其中每个画笔都是一个插件，用户增加或者删除一个插件，重新启动程序后，菜单中的选项会相应地增加或者删除。

#### 创建项目

首先，创建一个名为ch06\_drawing、类型为subdirs的项目，并创建它的两个子项目，一个叫app，作为我们的主程序；另外一个叫plugins，也是一个subdirs类型的项目，用来放置扩展主程序的插件项目。

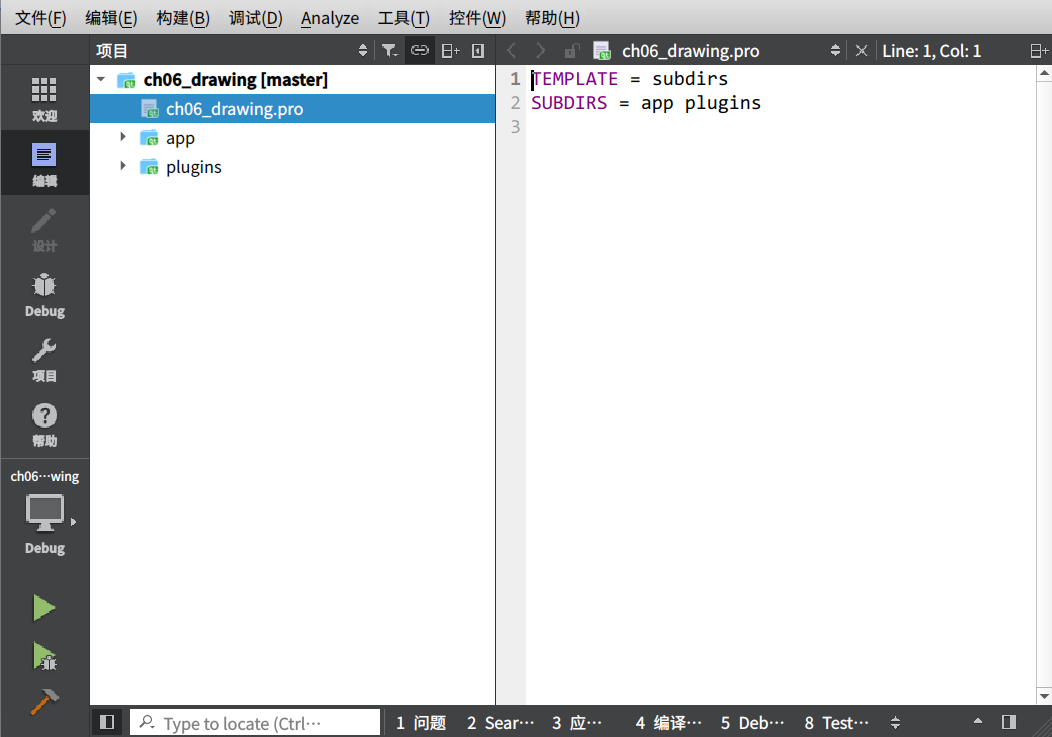


图 6.1 ch06\_drawing项目

#### 定义接口

在编写主程序的其他代码之前，首先需要定义主程序和插件之前的交互接口。在这个例子中主程序会主动调用插件来完成绘制操作，插件不需要主动调用主程序的接口，所以我们只需要定义一个接口即可：

class PainterInterface {

public:

// 画笔的绘制函数，每个插件都需要有自己的实现

virtual void ***paint***(QPainter \*painter, const QPoint &pos) = 0;

virtual ~***PainterInterface***() = default;

};

// 接口ID，全局唯一

#define PainterInterface\_iid "com.deepin.examples.ch06\_drawing"

// 声明接口

Q\_DECLARE\_INTERFACE(PainterInterface, PainterInterface\_iid)

其中，paint方法接受一个QPainter类型的指针和一个QPoint类型的引用常量两个参数，用来做实际的绘制操作；PainterInterface\_iid作为接口的唯一标识符，我们选择了域名倒置的方式进行定义。

接口定义文件interface.h作为app和plugins共用的内容，在项目顶层目录新建include目录存放。

注：在一般的项目结构中，include目录中的文件会被打包，最终安装到系统/usr/include/或者其他常见的头文件存放目录下，作为开发头文件使用，在我们的例子项目中为了方便演示，会直接加到各个子项目的INCLUDEPATH中。

#### 编写主程序

在主程序中，我们使用一个自定义的Widget类来提供主窗口，并为他设置了Qt::WA\_OpaquePaintEvent，表示控件自己控制绘制内容的清除工作，不需要Qt在窗口发生大小改变等情况下自动的清理动作。并且为窗口添加了右键菜单，每个菜单项对应于插件提供的一种画笔，从加载的插件中动态获取。在控件的mousePressEvent中，记录用户点击的位置，并调用更新函数update()，这样在paintEvent函数中就可以直接使用用户选择的画笔进行图形绘制了，代码如下：

#include "widget.h"

#include "pluginmanager.h"

#include <QAction>

Widget::**Widget**(QWidget \*parent)

: QWidget(parent),

m\_painter(nullptr)

{

// 设置窗口绘制和清理由程序自己完全负责，Qt无需自动处理

// 这样做以后窗口背景默认变为黑色

setAttribute(Qt::WA\_OpaquePaintEvent);

// 添加右键菜单

setContextMenuPolicy(Qt::ActionsContextMenu);

// 获取所有画笔插件，每个画笔作为一个菜单项加入到菜单中

QList<PainterPlugin\*> painters = PluginManagerInstance->painters();

for (PainterPlugin \*painter : painters) {

QAction \*action = new QAction(painter->name());

action->setCheckable(true);

addAction(action);

// 设置默认画笔

if (!m\_painter) {

setPainter(painter);

}

connect(action, &QAction::triggered, this, [this, painter] {

setPainter(painter);

});

}

}

Widget::~***Widget***()

{

}

void Widget::***paintEvent***(QPaintEvent \*)

{

QPainter painter(this);

// 设置反锯齿

painter.setRenderHint(QPainter::Antialiasing);

// 设置画笔颜色为白色

painter.setPen(Qt::white);

// 在用户点击的位置，使用当前画笔绘图

if (m\_painter && m\_clickPoint != QPoint(0, 0)) {

m\_painter->painter()->*paint*(&painter, m\_clickPoint);

}

painter.end();

}

void Widget::***mousePressEvent***(QMouseEvent \*event)

{

// 在用户点击时记录点击位置

if (event->button() == Qt::LeftButton) {

m\_clickPoint = event->pos();

update();

}

QWidget::*mousePressEvent*(event);

}

PainterPlugin \*Widget::**painter**() const

{

return m\_painter;

}

void Widget::**setPainter**(PainterPlugin \*painter)

{

m\_painter = painter;

// 菜单项的互斥效果实现

for (QAction \*action : actions()) {

action->setChecked(action->text() == painter->name());

}

}

#### 编写插件

在plugins项目中新建子项目square，用来提供一个绘制方形的插件，pro文件如下：

QT += core gui

TARGET = square

TEMPLATE = lib

CONFIG += plugin

INCLUDEPATH += ../../include

DESTDIR = ../

SOURCES = \

squarepainter.cpp

HEADERS = \

squarepainter.h

DISTFILES += square.json

插件项目的配置区别于普通项目主要有两点：项目的模板需使用 lib而不是常见的app、CONFIG中需添加plugin配置项。square.json文件为本插件提供元数据信息，内容如下：

{

"Name" : "Square"

}

在本例子中的使用非常简单，只是提供了插件的名称。

另外，在这个项目中，因为需要引用 interface.h 头文件，所以将其所在的include目录加入到了INCLUDEPATH中方便使用。将DESTDIR设置为上层目录则主要是需要将生成的插件放置到 plugins 目录方便加载。

在插件类中，新定义一个SquarePainter类，继承自QObject，并且实现了PainterInterface接口中的paint 方法：

class SquarePainter : public QObject, public PainterInterface

{

Q\_OBJECT

// 给插件添加静态元数据，这些数据在插件加载的过程中可以用作静态检查项

Q\_PLUGIN\_METADATA(IID PainterInterface\_iid FILE "square.json")

// 声明实现的接口

Q\_INTERFACES(PainterInterface)

public:

// 实现PainterInterface中的paint方法

void ***paint***(QPainter \*painter, const QPoint &pos) Q\_DECL\_OVERRIDE;

};

需要注意的是插件类中 Q\_OBJECT、Q\_PLUGIN\_METADATA、Q\_INTERFACES 三个宏的使用缺一不可。

在插件类的实现中，添加 paint 函数的实现，在paint函数中，以 pos 为中心绘制一个边长为 40px 的正方形，代码如下：

static const int Radius = 20;

void SquarePainter::***paint***(QPainter \*painter, const QPoint &pos)

{

QRect r(pos.x() - Radius, pos.y() - Radius, Radius \* 2, Radius \* 2);

painter->drawRect(r);

}

完成后，通过 QtCreator 编译 ch06\_drawing 项目，在编译目录的 plugins 目录下，就会生成一个名为 libsquare.so 的动态库，这就是我们新编写的插件。

#### 加载插件

在主程序中，另外创建一个PluginManager类，用于加载绘图插件，因为这个类可以全局使用，所以在本例子中简单地使用 Q\_GLOBAL\_STATIC 宏为这个类创建一个全局静态实例，方便使用：

QList<PainterPlugin \*> PluginManager::**painters**()

{

if (m\_initialized)

return m\_painters;

// 设置插件所在的目录

QDir pluginsDir("../plugins/");

// 遍历目录中的所有文件，尝试加载

const QStringList files = pluginsDir.entryList(QDir::Files);

for (const QString file : files)

{

// 过滤非动态库的内容

if (!QLibrary::isLibrary(file)) {

qWarning() << "file's not plugin type: " << file;

continue;

}

// 使用QPluginLoader尝试将文件作为插件加载

QPluginLoader loader(pluginsDir.absoluteFilePath(file));

bool status = loader.load();

if (!status) {

qWarning() << "failed to load file as plugin: " << file;

qWarning() << "error message: " << loader.errorString();

continue;

}

// 尝试从加载成功的插件中获取到入口对象

QObject \*instance = loader.instance();

if (!instance) {

qWarning() << "failed to get object from plugin: " << file;

continue;

}

// 将入口对象转换为我们所需的接口

PainterInterface \*painter = qobject\_cast<PainterInterface\*>(instance);

if (!painter) {

qWarning() << "failed to convert plugin to PaitnerInterface: " << file;

instance->deleteLater();

continue;

}

// 获取静态元数据

const QJsonObject meta = loader.metaData().value("MetaData").toObject();

const QString name = meta.value("Name").toString();

// 构造封装类PainterPlugin的实例

PainterPlugin \*plugin = new PainterPlugin(painter, name);

m\_painters << plugin;

}

return m\_painters;

}

插件加载的过程，分为以下几个步骤：

1. 在实际的加载动作发生之前，首先使用QLibrary::isLibrary过滤掉了一些非动态库的文件；
2. 使用QPluginLoader来尝试加载插件，如果加载失败，可以使用QPluginLoader::errorString获取出错信息，以此来排查插件的加载问题；
3. 插件加载成功后，QPluginLoader::instance会获取到一个QObject类的指针，使用这个对象之前，需要通过qobject\_cast来转换为我们需要的接口类型；
4. 通过QPluginLoader::metaData获取插件的静态元数据，此函数返回一个QJsonObject对象，如果想获取到square.json中的内容，仍然需要从这个QJsonObject对象中获取其“MetaData”对象的值；
5. 数据封装。

一般情况下，对于获取到的元数据信息，只需在插件加载的时候使用，例如声明插件加载所需依赖和环境等，但是在本例中我们需要对这些元数据进行保存，因此将插件和插件的元数据封装为一个新的PainterPlugin数据类型，代码如下：

class PainterPlugin

{

public:

PainterPlugin() = default;

PainterPlugin(PainterInterface \*iface, QString name);

~PainterPlugin();

PainterInterface \***painter**() const;

QString **name**() const;

private:

PainterInterface \*m\_iface;

QString m\_name;

};

PainterPlugin::**PainterPlugin**(PainterInterface \*iface, QString name):

m\_iface(iface),

m\_name(name)

{

}

PainterPlugin::~PainterPlugin()

{

if (m\_iface)

delete m\_iface;

}

PainterInterface \*PainterPlugin::**painter**() const

{

return m\_iface;

}

QString PainterPlugin::**name**() const

{

return m\_name;

}

#### 实际运行

以上内容完成后，对整个项目进行编译和运行，右键选择Square并点击左键进行绘制，程序显示如下：

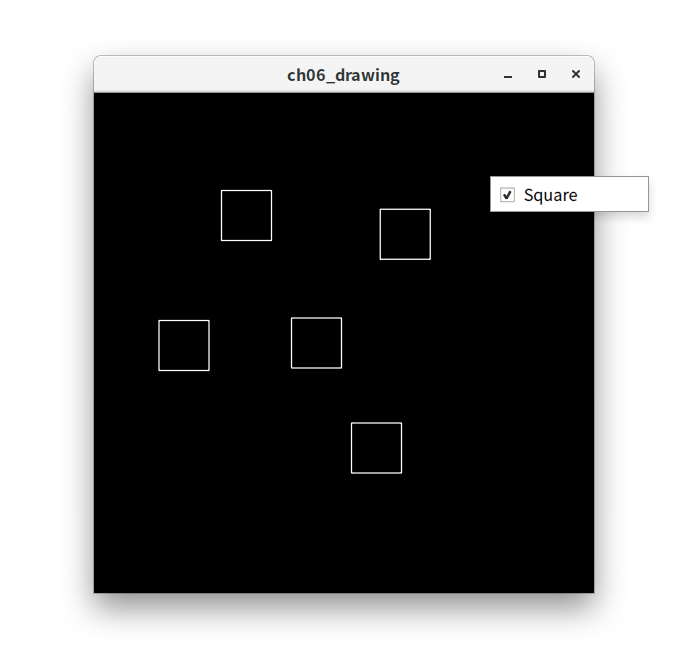


图 6.2 Square画笔

按照同样的步骤，创建一个新的circle插件项目，用来绘制圆形，编译后产生libcircle.so文件，将文件跟libsquare.so放置在一起，运行主程序，右键选择画笔并进行绘制，产生效果如下：

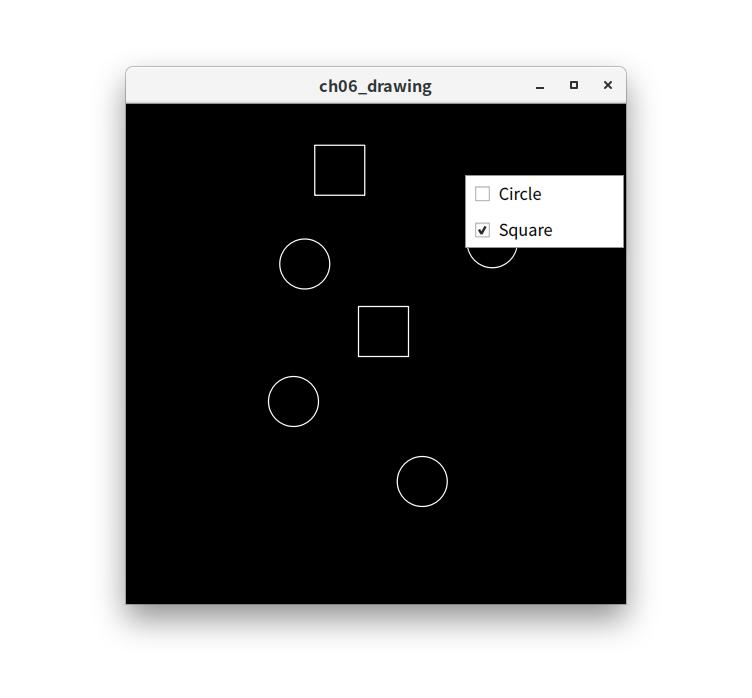


图 6.3 Square和Circle画笔

删除其中任何一个或者全部插件，重启主程序后右键菜单中相应地选项也会消失。

## Qt单元测试

我们都知道，程序等于数据结构加算法①，为了完成一件复杂的工作，一个程序中可能涉及多种数据结构，围绕着每种数据结构又存在着多个算法实现，每一个程序都是由这样的最小单元组合起来完成复杂工作的。为了保证程序整体的健壮性，最好的做法就要保证每个最小单元工作的正确性，这就是单元测试出现的背景。

单元测试（英语：Unit Testing）又称为模块测试, 是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作。程序单元是应用的最小可测试部件。在过程化编程中，一个单元就是单个程序、函数、过程等；对于面向对象编程，最小单元就是方法，包括基类（超类）、抽象类、或者派生类（子类）中的方法②。

简单来说，单元测试就是通过编写一些额外的测试代码，在代码中对程序中的最小单元给定不同的输入，检测相应的输出是否正确，以及时知道程序代码修改的可用性。为了方便地执行和展示，围绕着单元测试的概念，出现了很多比较有名的单元测试框架，例如Java中主要使用的JUnit和Spock、Python标准库中的unittest模块，Go语言则内建了单元测试的支持等。对于一个主要由C++和Qt开发的应用程序，在单元测试框架的选择上主要有两个可选方案：googletest和QTestLib。

#### googletest

googletest 是由谷歌开发的用于C++单元测试的框架，基于xUnit架构设计，支持单元测试的同时，也支持基准测试和简单的压力测试，在运行环境方面，可以支持在POSIX系统和Windows系统中运行，在Chromium、LLVM、Protocol Buffers、OpenCV、Gromacs 等项目中均有应用。

#### QTestLib

QTestLib是Qt公司专门为Qt打造的一套单元测试框架，从其官网的介绍来看，它主要有以下特性：

表 6.1 QTestLib单元测试框架的特性

|  |  |
| --- | --- |
| 特性 | 说明 |
| 轻量级 | QTestLib只有6000多行代码，导出的符号也只有约60个 |
| 自包含 | 对于非图形化的测试来说，QTestLib只依赖Qt Core中少量的几个符号 |
| 速度快 | QTestLib不需要特殊的测试执行程序，不需要为测试而进行特殊的注册 |
| 数据驱动测试 | 同一个测试用例可以无需修改而使用不同的数据重复执行 |
| 简单的图形测试 | QTestLib支持模拟简单的鼠标和键盘事件，用于图形化控件的单元测试 |
| 基准测试 | QTestLib支持基准测试，并且可以支持不同的基准测试后端 |
| IDE友好 | QTestLib的输出可以被QtCreator、Visual Studio和KDevelop等IDE识别和展示 |
| 线程安全 | 错误输出线程安全 |
| 类型安全 | 加强的模板可以很好地防止隐式类型转换导致的错误 |
| 易于扩展 | 自定义的数据类型可以很方便地添加到测试数据和测试结果中 |

相比较而言，googletest更偏向通用，而QTestLib则与Qt有更好的结合，且支持图形控件的单元测试，所以本章介绍的单元测试主要是指使用QTestLib做Qt程序的单元测试。

### 使用

#### 编写测试用例

针对一个特定单元的测试叫做测试用例。在QTestLib中，一个测试用例对应于测试类中的一个测试函数，针对一个类的测试用例一般放在同一个测试类中。需要注意的是测试类需继承自QObject，每个测试函数需声明为私有槽。

在测试函数中，给予待测试单元固定的输入，并期待特定的输出，写下断言。例如我们需要测试QString::toUpper函数，那么测试代码便是：

#include <QtTest/QtTest>

class TestQString: public QObject

{

Q\_OBJECT

private slots:

void **toUpper**();

};

void TestQString::toUpper()

{

QString str = "Hello";

QVERIFY(str.toUpper() == "HELLO");

}

在上面的代码中，我们给QString::toUpper的输入为“Hello”，期望输出为“HELLO”，使用宏QVERIFY断言待测试函数的输出与期望相等，如果测试用例执行失败，则说明QString::toUpper 函数实现有问题，需要修正。

在单元测试中，断言的使用非常多，Qt提供了一些用来做断言或者辅助断言的宏：

* QCOMPARE(actual, expected)
* QEXPECT\_FAIL(dataIndex, comment, mode)
* QFAIL(message)
* QFETCH(type, name)
* QFINDTESTDATA(filename)
* QSKIP(description)
* QTRY\_COMPARE(actual, expected)
* QTRY\_COMPARE\_WITH\_TIMEOUT(actual, expected, timeout)
* QTRY\_VERIFY2(condition, message)
* QTRY\_VERIFY(condition)
* QTRY\_VERIFY2\_WITH\_TIMEOUT(condition, message, timeout)
* QTRY\_VERIFY\_WITH\_TIMEOUT(condition, timeout)
* QVERIFY2(condition, message)
* QVERIFY(condition)
* QVERIFY\_EXCEPTION\_THROWN(expression, exceptiontype)
* QWARN(message)

其中，比较常用的是 QVERIFY、QVERIFY2、QCOMPARE和QFAIL，前两者接收一个返回值是布尔类型的表达式，QCOMPARE用来对比预期结果和实际结果，QFAIL可以随时让一个测试用例失败掉，从而制造更灵活的控制。使用 QCOMPARE 和 QFAIL 替换上面例子中的 QVERIFY 实现相同的效果，代码分别为：

void TestQString::toUpper()

{

QString str = "Hello";

QCOMPARE(str.toUpper(), "HELLO");

}

和

void TestQString::toUpper()

{

QString str = "Hello";

if (str.toUpper() != "HELLO")

QFAIL("toUpper has unexpected output");

}

#### 执行单元测试

单元测试项目与普通项目的pro文件并没有什么特别的不同，只是需要额外添加一下依赖的testlib 模块：

QT += testlib

不同的地方主要在于单元测试项目一般没有显式的main.cpp文件和main函数，而是使用QTEST\_MAIN宏生成需要的程序入口函数，在QTEST\_MAIN后需要包含源码生成的moc文件，例如在上述的例子中我们创建的类名为TestQString、文件名为tst\_testqstring.cpp，那么这部分代码就应该是：

// 生成main函数

QTEST\_MAIN(TestQLineEdit)

// 包含moc生成文件

#include "tst\_TestQLineEdit.moc"

添加完入口函数以后就可以运行测试用例了，有两种方式，第一种是直接运行生成的二进制，这种模式下项目输出纯文本的测试结果数据：

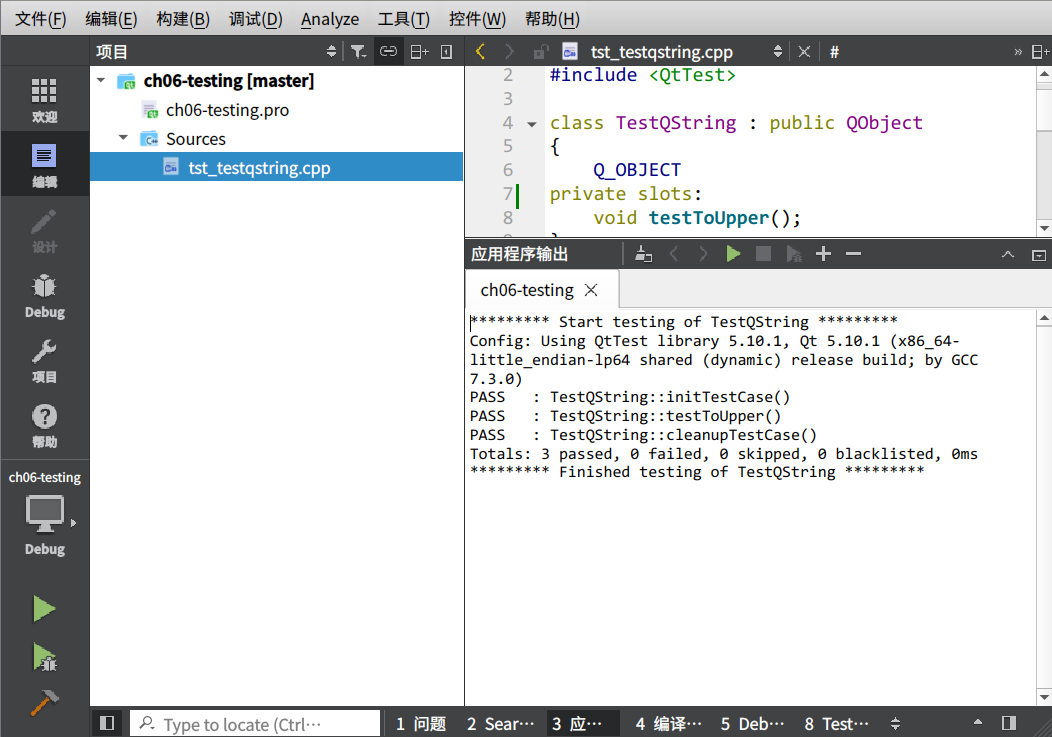


图 6.4 运行单元测试

PASS 表示测试通过，代码功能正常。

还有一种方式是使用QtCreator提供的图形化方式，需要在QtCreator中打开“Test Results”面板，面板中两个运行按钮一个是“运行所有测试”，另外一个是“运行所选测试”。点击“运行所有测试”的按钮，可以得到如下显示结果：

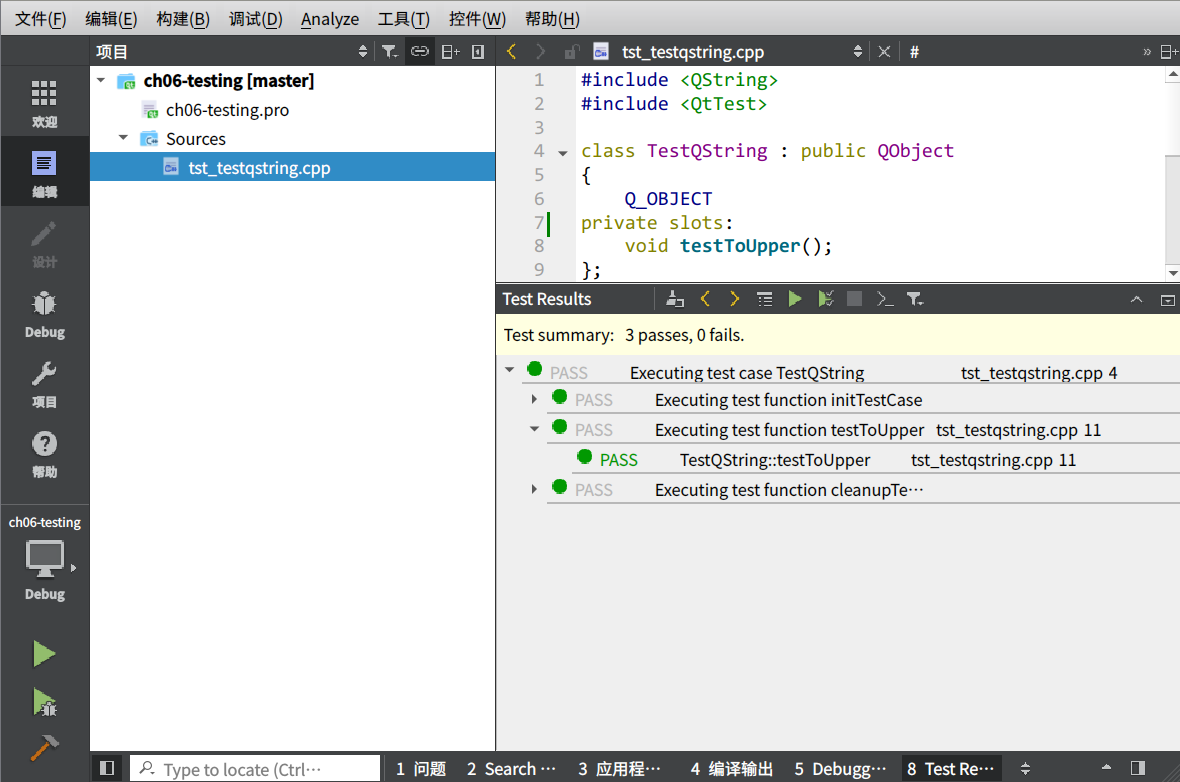


图 6.5 图形化测试结果展示

所有的测试用例都会以图形化的方式展示是否运行通过，更加直观。如果在一个项目里面有多个测试用例，可以在QtCreator的资源管理器选择“Tests”视图：

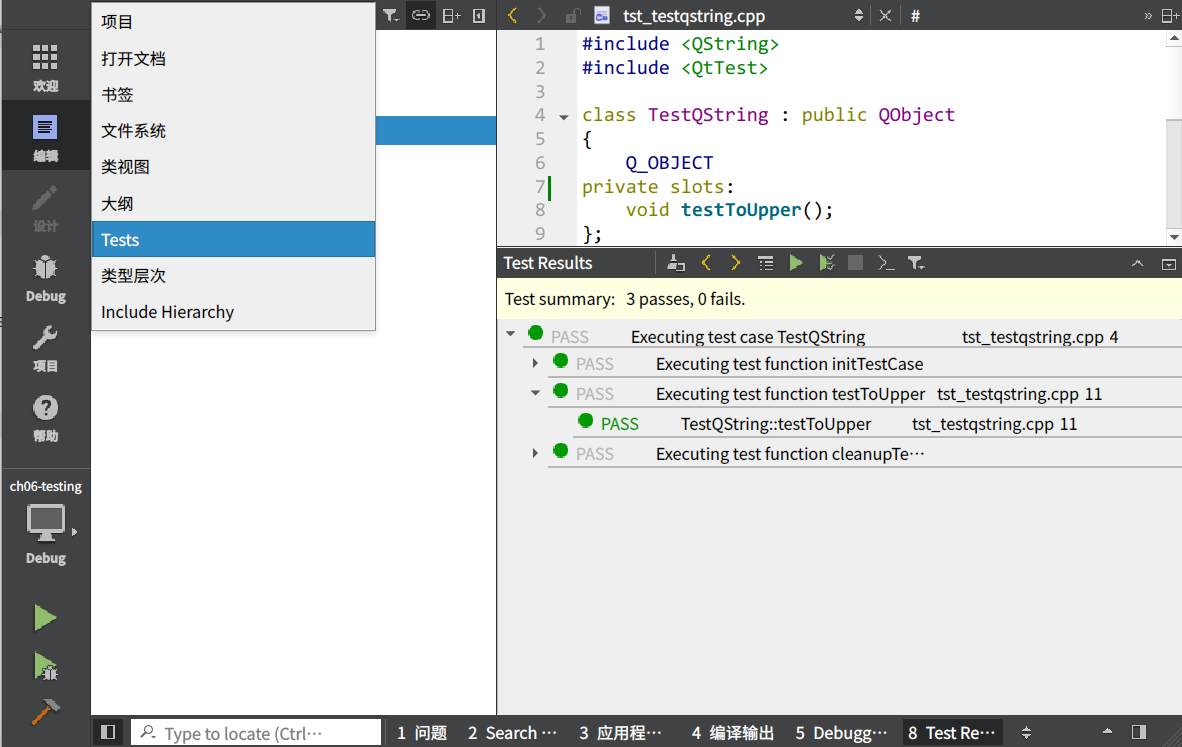


图 6.6 选择Tests视图

这时候在资源管理器中可以看到项目中有哪些测试用例，并且可以勾选其中的一部分测试用例来执行。挑选完成后，在刚才打开的“Test Results”面板，点击“运行所选测试”按钮来运行我们挑选的测试用例。这个功能在重构或者修改某个功能函数的时候比较有用，因为这个时候我们一般只关心这个函数对应的测试用例是否能通过。

#### 测试用例的生命周期

在上面的测试结果中，除了TestQString::toUpper用例，还有两个函数TestQString::initTestCase和TestQString::cleanupTestCase，这两个函数并不是我们编写的，为什么会存在呢？这就涉及到了单元测试的生命周期。

单元测试的生命周期主要涉及四个特殊的函数：

表 6.2 单元测试中的特殊函数

|  |  |
| --- | --- |
| 函数名 | 说明 |
| initTestCase() | 在所有的测试用例运行之前执行 |
| cleanupTestCase() | 在所有的测试用例运行完成后执行 |
| init() | 在每个测试用例运行之前执行 |
| cleanup() | 在每个测试用例运行完成之后执行 |

我们可以重写这几个函数，用于在测试用例执行前后完成一些准备和清理的工作，所以，这些函数在需要准备的内容耗时较长而每个测试用例又会使用到的情况下会比较有用。如果没有提供相应的函数，QTestLib会为我们生成空的initTestCase和cleanupTestCase函数。

#### 数据驱动测试

在前面测试QString::toUpper的测试用例中，我们只是测试了输入为“hello”这种输入是全小写的情况，还有一些其他预期的情况，例如全大写的“HELLO”、混合大小写的“Hello”等情况没有测试，假如我们以前发现QString::toUpper函数可能存在类似的潜在问题，那么就需要在测试用例中包含这些情况：

void TestQLineEdit::**testToUpper**()

{

// 测试全小写的情况

QString str = "hello";

QVERIFY(str.toUpper() == "HELLO");

// 测试全大写的情况

str = "HELLO";

QVERIFY(str.toUpper() == "HELLO");

// 测试混合大小写的情况

str = "Hello";

QVERIFY(str.toUpper() == "HELLO");

}

添加相关的代码到测试用例中运行，发现结果都仍然符合预期。

这种添加测试数据的方式虽然也可以完成工作，但是可以发现其中有不少重复代码，假如待测试的接口发生改变，我们需要修改不少地方才能让测试用例可以重新工作起来，所以，这种方式并不是一种好的添加测试数据的方法。

Qt提供了一种更好的方式来满足这种需要提供多套测试数据的测试需求，在这种方式中，编写测试函数的时候，不是写死测试数据，而是假设有一个“测试数据表格”，在这个表格中每一行都是一组测试数据，每一列代表了每组测试数据中各个数据所负责的角色。测试函数通过Qt提供的宏每次从表格中取出每一组数据来使用，测试数据则由单独的数据函数来提供。下面举个例子来说明。

假如我们除了要测试QString::toUpper外，还要测试QString::toLower函数的功能，并且这次需要测试输入为全小写、全大写、混合大小写的三种情况。首先，需要创建一个数据函数，这个函数同样为私有槽函数，命名规则为在所需提供数据的测试函数名后加上“\_data”后缀：

void TestQLineEdit::**testToLower\_data**()

{

// 添加input列

QTest::addColumn<QString>("input");

// 添加expected列

QTest::addColumn<QString>("expected");

// 添加全小写的测试数据行，数据需要按照上面数据列的顺序进行添加

QTest::newRow("all lower") << "hello" << "hello";

// 添加混合大小写的测试数据行

QTest::newRow("mixed") << "Hello" << "hello";

// 添加全大写的测试数据行

QTest::newRow("all upper") << "HELLO" << "hello";

}

在代码中，先使用QTest::addColumn来添加表格所有的列，然后使用QTest::newRow添加数据行，函数参数为行名称，会在测试用例执行结果中显示，方便定位出问题的数据。

数据函数完成后，测试函数可以写作：

void TestQLineEdit::**testToLower**()

{

// 获取input列测试数据

QFETCH(QString, input);

// 获取expected列测试数据

QFETCH(QString, expected);

// 断言

QCOMPARE(input.toLower(), expected);

}

在这个函数中，使用QFETCH宏从数据表中取出测试数据，需要注意的QFETCH宏的两个参数，一个是数据列的类型，另外一个是数据列的名称，都需要与数据函数中QTest::addColumn提供的数据类型和名称一致。取出数据后便可以前面的测试函数一样使用、测试。运行测试，结果如下：

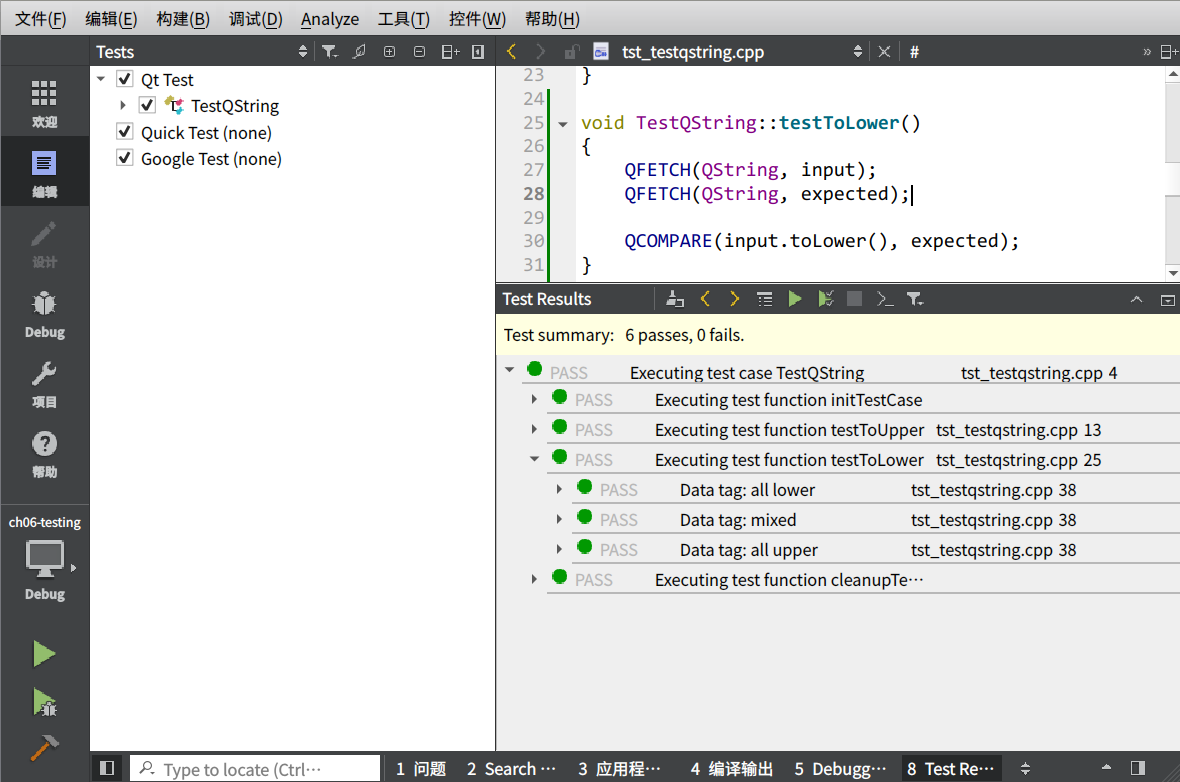


图 6.7 测试结果

#### 图形化测试

在图形界面程序中，自定义控件或者组合控件的情况是很常见的，作为界面的一个最小单元模块，对这些控件的单元测试需求也长期存在，所以Qt提供了简单的图形化测试，可以很方便地满足对单个控件的测试需求。

在下面的例子中，我们新建了一个测试测试项目用来对 QLineEdit 的功能进行验证，测试类的代码如下：

class TestQString : public QObject

{

Q\_OBJECT

private slots:

void **testInput**();

};

void TestQString::**testInput**()

{

QLineEdit lineEdit;

// 模拟键盘输入

QTest::keyClicks(&lineEdit, "hello world");

// 断言

QCOMPARE(lineEdit.text(), QString("hello world"));

}

// 生成main函数

QTEST\_MAIN(TestQString)

// 包含moc生成文件

#include "tst\_testqlineedit.moc"

上面的代码中，我们使用QTest::keyClicks向创建的QLineEdit控件发送了一系列的键盘输入，组成了“hello world”输入串，QLineEdit控件接收到这些事件以后按照预期应该显示“hello world”在输入框中，所以使用 QCOMPARE 宏断言控件的内容和我们的预期应该相同，这样就完成了一个简单的图形化测试。

除了QTest::keyClicks外，QTest类还提供了一系列的函数用来模拟其他类型的输入事件，例如keyClick用来模拟单次键盘点击、keyPress模拟键盘按下、keyRelease模拟键盘松开、mouseClick模拟鼠标点击、mousePress和mouseRelease分别模拟鼠标按下和松开等，基本可以满足日常输入事件的模拟需求。

同样的，对于同一个控件的一个行为，可能需要使用不同的输入事件去验证其行为是否正确。在Qt中也可以使用前文提到的数据提供方式来测试图形程序，只是需要在数据函数中使用QTestEventList作为事件传输的“载体”，如下面的代码所示：

void TestQString::**testInputAndDelete**()

{

// 获取events列测试数据

QFETCH(QTestEventList, events);

// 获取expected列测试数据

QFETCH(QString, expected);

QLineEdit lineEdit;

// 模拟将测试事件数据发送到lineEdit对象

events.simulate(&lineEdit);

// 断言

QCOMPARE(lineEdit.text(), expected);

}

void TestQString::**testInputAndDelete\_data**()

{

// 添加events列

QTest::addColumn<QTestEventList>("events");

// 添加expected列

QTest::addColumn<QString>("expected");

// 构造测试事件数据

QTestEventList list1;

list1.addKeyClick('a');

// 添加测试数据行

QTest::newRow("char") << list1 << "a";

// 构造测试事件数据

QTestEventList list2;

list2.addKeyClick('a');

list2.addKeyClick(Qt::Key\_Backspace);

// 添加测试数据行

QTest::newRow("there and back again") << list2 << "";

}

在这个测试用例中，除了测试输入功能，还测试了输入后删除输入的动作。执行单元测试，用例全部通过，说明控件的行为符合我们的预期。

## polkit鉴权系统

在开发linux应用程序的过程中，很多情况下我们会需要获取系统级别的权限，以实现应用程序需要的功能，例如修改系统配置、开关系统服务等，随之也出现了各种各样的工具和方法来实现这个目标，比如最古老的setuid的方式，可以让程序以其他用户的身份运行，这种权限控制一刀切的做法很容易导致权限被滥用，所以，后来出现了linux capabilities，capabilities特性可以让程序附带一定的特殊权限，如访问网络接口、发送kill信号等，这样程序可以只附带自己需要的特权，而无需直接以系统最高权限执行。

还有一些工具本身带有比较高的用户权限，可以用来辅助我们的程序获取高权限，例如sudo、gksu、kdesu等，使用这些工具启动我们的程序，会先鉴定执行者的凭证，比如在终端中使用sudo会要求用户输入正确的管理员密码；启动一些图形应用程序的时候，在程序命令前加上gksu或者kdesu，它们也会弹出一个对话框让用户输入正确的密码才能继续执行，验证通过后，我们的程序便以高权限执行了。

以上两种方式是在Linux中最常见的两种提权方式，但是，对于图形应用程序来说，却并不是很实用，主要原因有以下两点：

图形应用程序一般都是非特权的，这是因为我们需要获取系统级别权限大多数是比较临时的情况，完成操作后就不需要再保留有那么高的权限了，否则一旦程序出现了问题，很容易导致系统级别的安全隐患，另外，由于图形程序是用户操作电脑最直接的界面，如果用户忘记程序已经是高权限而胡乱操作，很容易对系统造成破坏。

capabilities虽然可以让程序拥有部分特殊能力，但是这些特殊能力的数量是有限的，所以比较适合一些功能比较固定的系统级工具，例如ping等，对于应用程序来说仍然不够灵活。

所以，我们需要一种将需要特权执行的代码分离开来的机制，这就是polkit要做的事情。

polkit是Linux环境下的一套鉴权系统，通过这个框架提供的API，低权限的用户程序可以向拥有高权限的系统程序发出请求，通过用户权限鉴定后，执行特定的动作，在polkit的术语中，这种低权限的用户程序叫Subject，高权限的系统程序叫Mechanism，而用来跟用户做鉴权交互的程序叫做Agent，统一由提供polkit服务的后台程序polkitd调度和控制，polkitd的对外接口主要以提供DBus的system bus服务为主，他们之间的架构图如下：

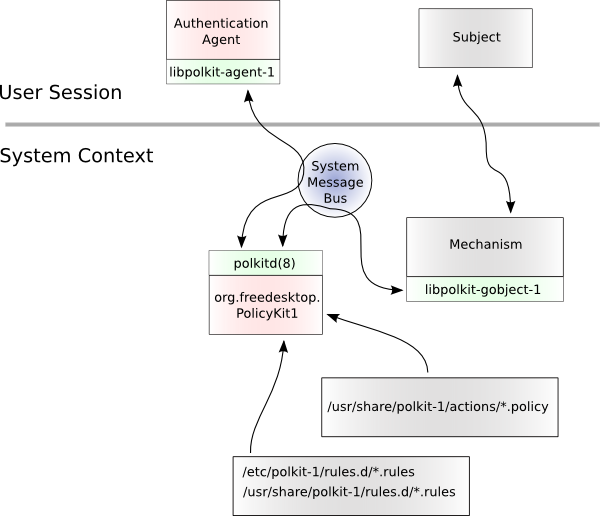


图 6.8 架构图③

在上图中，Subject和Agent都工作来用户会话中，拥有普通的用户权限，polkitd和与Subject对应的Mechanism则工作在系统层面，拥有系统权限。Subject需要执行高权限动作的时候，会执行一下步骤：

1. 透过DBus向Mechanism发出DBus调用；
2. Mechanism接到请求以后开始做用户权限鉴定，即向polkitd发出请求；
3. polkitd会检查系统中安装的rules文件和policy文件，以确定继续验证请求还是直接拒绝，拒绝后Subject调用直接返回失败；如果是继续验证请求，polkitd则透过DBus向Agent发送消息要求开始鉴权；
4. 按照规则，用户需要输入自己的密码或者选择一个管理员进行密码验证，验证通过后Agent会告知polkitd鉴权结果，polkitd再转而反馈给Mechanism；
5. Mechanism收到验证结果后决定执行请求动作还是返回失败。

在图形界面下，Agent一般是一个对话框：

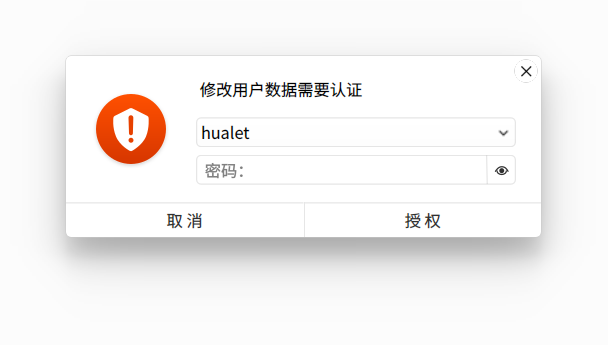


图 6.9 鉴权对话框

### 声明动作

在上面流程介绍中，只是介绍了特权动作的使用过程，动作在被使用之前还需要进行声明，声明其实就是使用xml文件对一个或者多个动作进行一定的描述，这种xml文件以.policy结尾，通常放置在 /usr/share/polkit-1/actions/目录中，它们的根节点是policyconfig，子节点可以包含 vendor、vendor\_url、icon\_name和action等，分别用于描述这个文件中一系列动作的开发商、开发商网址、图标和各个动作，其中，action作为主要内容，又可以包含 description、message、defaults、annotate、vendor、vendor\_url、icon\_name，分别用来表示这个动作的描述信息、动作在Agent上的提示信息、默认鉴权行为、注解、开发商、开发商网址、图标等。

每个action可以设置不同的鉴权行为策略，也就是上面提到的defaults项，它可以包含对三种情况的鉴权行为配置项：allow\_any表示针对所有请求者的鉴权行为、allow\_inactive表示在当前终端下的非活跃会话中的请求者、allow\_active则表示在当前终端下的活跃会话中的请求者，这三种情况下分别可以设置不同的鉴权策略：

表 6.3 polkit鉴权策略

|  |  |
| --- | --- |
| 配置项 | 说明 |
| no | 直接失败 |
| yes | 直接通过 |
| auth\_self | 使用自己的凭证鉴权，用来验证动作执行的请求者确实是所在会话的拥有者 |
| auth\_admin | 使用管理员的凭证鉴权 |
| auth\_self\_keep | 跟auth\_self类似，但是会为请求者保留一段时间的鉴权状态（例如5分钟内的特权动作请求无需再验证） |
| auth\_admin\_keep | 跟auth\_admin类似，但是会为请求者保留一段时间的鉴权状态（例如5分钟内的特权动作请求无需再验证） |

另外，每个动作除了可以使用 vendor、vendor\_url、icon\_name等覆盖总体的这些信息外，还可以使用annotate添加一些键值对形式额外的注解，这些注解可以在Agent中进行展示或者使用，主要用于扩展，例如比较常见的注解是：org.freedesktop.policykit.exec.path ， 它会被pkexec识别，用来表示这个声明的动作内容可以用来代替默认的Agent展示内容。

### 定义规则

polkit规则是一系列的JavaScript脚本，通常放置在 /etc/polkit-1/rules.d 和 /usr/share/polkit-1/rules.d 中。通过暴露的polkit对象，这些脚本可以为polkit添加一些特殊的鉴权规则，通常使用 polkit.addRule 或者 polkit.addAdminRule 来完成，前者用来添加在鉴权过程中执行的回调函数；后者用来添加在挑选管理员的时候执行的回调函数。这两种回调函数都接收两个参数，一个是action，表示一个动作；另外一个是subject，表示一个请求的发起者。从两个对象中，我们又可以获取各自更多的信息，用来做出动态的判断，例如

polkit.addRule(function(action, subject) {

if (action.id == "org.freedesktop.accounts.user-administration" &&

subject.isInGroup("admin")) {

return polkit.Result.YES;

}

});

上面的代码添加了一个规则，用来表示在admin组中的用户可以执行 org.freedesktop.accounts.user-administration 的动作。

polkit.addAdminRule(function(action, subject) {

return ["unix-group:wheel"];

});

则表示在wheel这个组中的用户才算作管理员。总之，polkit规则可以很方便地让系统管理员动态配置polkit鉴权过程中的一些条件和结果，用以辅助系统管理上的一些特殊需求，不过对于应用开发者来说使用不多，不再详细说明。

下面我们通过一个例子来具体说明polkit在应用开发中的使用方式。

### 例子

在这个例子中，我们将实现一个非常简单的系统环境变量修改器，通过这个修改器，无需再手工编辑 /etc/environment 文件即可完成配置系统环境变量的需求。

例子项目有两个子项目，分别是helper和editor，editor是一个图形化的编辑界面，运行起来的样子如下图所示：

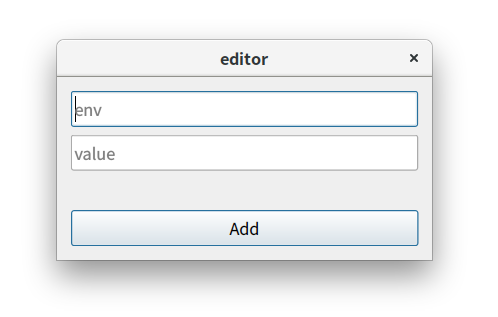


图 6.10 编辑器

用户在操作的时候，分别在env行和value行填写入要设置的环境变量的名称和相应的环境变量的值，然后点击Add即可添加环境变量设置到 /etc/environment 文件，如果添加成功，在输入框下面按钮上面会显示“success”，如果失败则会显示详细的错误信息，在editor的代码中只有界面代码，实际的写入文件的操作通过DBus调用helper提供的接口来完成，其主要代码如下：

#include "dialog.h"

#include <QLabel>

#include <QLineEdit>

#include <QPushButton>

#include <QVBoxLayout>

#include <QDBusReply>

#include <QDBusInterface>

Dialog::**Dialog**(QWidget \*parent)

: QDialog(parent)

{

// 设置对话框大小

setFixedWidth(300);

// 搭建对话框内容控件

QVBoxLayout \*layout = new QVBoxLayout;

m\_key = new QLineEdit;

m\_key->setPlaceholderText("env");

m\_value = new QLineEdit;

m\_value->setPlaceholderText("value");

QPushButton \*btn = new QPushButton("Add");

m\_msg = new QLabel;

m\_msg->setWordWrap(true);

layout->addWidget(m\_key);

layout->addWidget(m\_value);

layout->addWidget(m\_msg);

layout->addWidget(btn);

setLayout(layout);

connect(btn, &QPushButton::clicked, this, &Dialog::buttonClicked);

}

Dialog::~***Dialog***()

{

}

void Dialog::**buttonClicked**()

{

if (m\_key->text().isEmpty() || m\_value->text().isEmpty())

return;

// 使用用户输入的数据作为参数，调用DBus方法

QDBusInterface iface("com.deepin.examples.ch06.sysenvhelper",

"/com/deepin/examples/ch06/sysenvhelper",

"com.deepin.examples.ch06.SysEnvHelper",

QDBusConnection::systemBus());

QDBusReply<bool> reply = iface.call("SetEnv", m\_key->text(), m\_value->text());

// 在界面上反馈调用结果

if (reply.isValid() && reply.value()) {

m\_msg->setText("success");

} else {

m\_msg->setText("err: " + reply.error().message());

}

}

与之相对的，helper程序提供系统级接口，供editor调用，本身不提供界面。在helper项目中的 SysEnv 类，暴露了一个槽函数SetEnv，用来提供DBus方法：

bool SysEnv::**SetEnv**(const QString &name, const QString &val)

{

// 检查输入是否合法

if (name.isEmpty())

return false;

// 写入内容到/etc/environment中，这里使用system函数调用系统命令完成

QString cmd = QString("echo %1=%2 >> /etc/environment").arg(name).arg(val);

int ret = system(cmd.toUtf8());

return ret == 0;

}

并在main函数中将注册系统DBus服务和对象：

// DBus服务名

#define Service "com.deepin.examples.ch06.sysenvhelper"

// DBus对象路径

#define Path "/com/deepin/examples/ch06/sysenvhelper"

// DBus接口

#define Interface "com.deepin.examples.ch06.SysEnvHelper"

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

{

QCoreApplication a(argc, argv);

SysEnv sysenv;

QDBusConnection systemBus = QDBusConnection::systemBus();

bool status;

// 注册DBus服务

status = systemBus.registerService(Service);

if (!status) {

qWarning() << "failed to register service" << systemBus.lastError().message();

return -1;

}

// 将sysenv实例注册为DBus对象，并暴露实例所有的槽函数为DBus方法

status = systemBus.registerObject(Path,

Interface,

&sysenv,

QDBusConnection::ExportAllSlots);

if (!status) {

qWarning() << "failed to register object";

return -2;

}

return a.exec();

}

编译helper项目，并以sudo执行（通常系统级的服务会包装成一个systemd的服务，这里为了例子简便，直接以root权限执行），会有如下报错：

failed to register service "Connection \":1.811\" is not allowed to own the service \"com.deepin.examples.ch06.sysenvhelper\" due to security policies in the configuration file"

这主要是出于安全性的考虑，系统级DBus服务的注册需要先完成一定的配置，一般情况下，只需要在 /usr/share/dbus-1/system.d/ 目录中添加一个对应于服务的配置文件即可：

<?xml version="1.0" encoding="UTF-8"?> <!-- -\*- XML -\*- -->

**<!DOCTYPE** busconfig PUBLIC

"-//freedesktop//DTD D-BUS Bus Configuration 1.0//EN"

"http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd"**>**

<busconfig>

<policy user="root">

<allow own="com.deepin.examples.ch06.sysenvhelper"/>

</policy>

<policy context="default">

<allow send\_destination="com.deepin.examples.ch06.sysenvhelper" />

</policy>

</busconfig>

配置文件中的policy节点用于定义不同的规则，一般是通过policy节点的属性来设置过滤条件，在policy节点的内容中通过allow或者deny和其属性来进一步完成一系列的规则配置，这里可以将DBus理解为一个网络服务，而这些配置文件就像是防火墙规则一样。

在上面的配置文件中，我们通过第一个policy指定root用户可以拥有com.deepin.examples.ch06.sysenvhelper服务，这是因为这个服务会以root身份运行，指明用户为root后，其中的allow条件才会被执行；在第二个policy规则中，指定context为default，表示默认情况就会加载这个policy中的规则，其中的allow节点表示发送向com.deepin.examples.ch06.sysenvhelper是被允许的。如果我们只配置第一个规则，虽然服务可以正常注册，但是在editor进行DBus调用的时候会找不到相应的服务而失败。

添加了上面的配置文件后，helper就可以正常注册服务并且提供相应的接口了，editor也可以正常的接入，这时候好像跟polkit还没什么关系，但是要注意的是helper提供的服务是非常危险的操作，现在任何人在任何时候都可以调用这个接口，从而更改系统环境变量，从而影响其他程序的正常行为；如果服务提供了更多特殊的操作，例如可以安装软件，那么这个系统就成了最好的病毒传播者。

所以，我们必须在服务调用的时候检查调用者的权限。一般情况下，不直接使用polkit服务提供的DBus接口，所以，在Qt5的项目中一般使用 polkit-qt5-1 这个库，它提供了的对polkit库的Qt5封装，可以在Qt5程序中方便地使用。具体的验证代码如下：

bool SysEnv::**doAuth**()

{

// 获取DBus调用者的PID

const QDBusMessage msg = message();

const QString service = msg.service();

const qint64 pid = connection().interface()->servicePid(service);

// 构造Subject实例

UnixProcessSubject sub(pid);

// 构造Authority实例

Authority \*auth = Authority::instance();

// 做权限鉴定

Authority::Result result = auth->checkAuthorizationSync(

"com.deepin.examples.ch06.setsysenv",

sub,

PolkitQt1::Authority::AllowUserInteraction);

if (result != Authority::Yes) {

qWarning() << "authorization failed: " << auth->errorDetails();

}

return result == Authority::Yes;

}

在上面的代码中，我们通过DBus的接口获取到调用方的一些信息，例如进程PID，其中使用到的 message()函数、connection()函数均来自QDBusContext，为了获取到这些信息，需要让SysEnv类继承了QDBusContext，这样被DBus调用的槽函数中，就可以获取一些额外的关于DBus调用的信息。

拿到PID后，可以用来构造UnixProcessSubject对象，这种类型的Subject表示鉴权是针对这个进程的，这是最常用的Subject类型，还有两种类型的Subject：UnixSessionSubject和SystemBusNameSubject，分别表示鉴权针对一个会话、鉴权针对一个系统服务，鉴权成功后，polkitd会根据相应动作配置的鉴权行为判断是否为这个进程、会话还是系统服务保留一定时间的鉴权成功状态。

Authority::checkAuthorizationSync函数用来发起一次鉴权检查，第一个参数是动作的ID，这个动作设定的鉴权行为是只允许在活跃会话中的使用管理员凭证进行鉴权，并且不维持鉴权成功状态。第二个参数是刚构造的Subject对象，第三个参数表示需要用户交互。根据定义的动作内容：

<?xml version="1.0" encoding="UTF-8"?>

**<!DOCTYPE** policyconfig PUBLIC

"-//freedesktop//DTD PolicyKit Policy Configuration 1.0//EN"

"http://www.freedesktop.org/standards/PolicyKit/1.0/policyconfig.dtd"**>**

<policyconfig>

<vendor>Deepin</vendor>

<vendor\_url>http://www.deepin.org/</vendor\_url>

<action id="com.deepin.examples.ch06.setsysenv">

<description>Change system environment variables</description>

<message>Changing system environment variables need authorization.</message>

<defaults>

<allow\_any>no</allow\_any>

<allow\_inactive>no</allow\_inactive>

<allow\_active>auth\_admin</allow\_active>

</defaults>

</action>

</policyconfig>

在每次鉴权的时候，系统都会弹出如下对话框让用户输入管理员凭证（密码等），用户鉴权成功后动作才会正确执行。

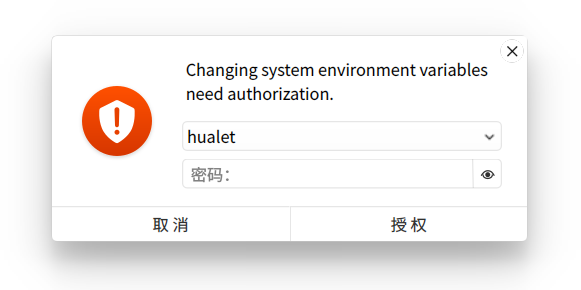


图 6.11 鉴权对话框

在上面的例子中，这个对话框会在用户点击Add按钮后弹出，填入正确的管理员密码，需要设置的环境变量就会按照预期出现在 /etc/environment 文件中。

## 小结

本章主要介绍了在Qt开发的过程中，我们如何利用Qt以及系统提供的相关技术对我们的程序进行改造和优化，满足程序在扩展性、健壮性和安全性等方面的需求。在介绍这几个方面的同时，我们又使用、回顾了前面几个章节讲到的内容，如DBus、动态库、qmake等。此外，我们还通过几个例子对相关的知识进行了演示和详细讲解，希望读者对本章的内容有更深刻的认识。

## 练习题

对最后一节中的例子进行改造，使其支持插件系统，可以选择“添加环境变量”和“添加host”功能，来进行相关的设置。

## 引用内容

1. 来自Niklaus Wirth的书《Algorithms + Data Structures = Programs》
2. 定义来自维基百科
3. 图片来自https://www.freedesktop.org/software/polkit/docs/latest/polkit.8.html