

Qt Quick 全面导引

Good Luck

2019 年 1 月 31 日

目录

目录	i
前言	1
第 1 章 Qt Quick 入门导引	3
1.1 搭建开发环境	3
1.1.1 在 Windows 平台下搭建开发环境	3
1.1.1.1 在 Windows 平台下安装 Qt	3
1.1.1.2 在 Windows 平台下安装 Boost	4
1.1.1.3 在 Windows 平台下 MinGW 配置 jemalloc	5
1.1.2 在 Linux 平台下搭建开发环境	6
1.1.2.1 在 Linux 平台下安装 Qt	6
1.1.2.2 在 Linux 平台下安装 Boost	6
1.2 qmake 入门	7
1.2.1 使用 qmake 构建 Hellow World!	7
1.2.2 使用 qmake 创建动态链接库	9
1.2.3 qmake 高级用法	12
1.2.4 qmake 生成 Visual Studio 工程	18
1.2.5 获得更多 qmake 帮助	19
1.3 第一个程序	20
1.3.1 本书的工程项目	20
1.3.2 Qt Quick 运行常用设置	22
1.3.3 使用 QQuickView 加载 Qt Quick 程序	24
1.3.4 使用 QQuickWidget 加载 Qt Quick 程序	25
1.3.5 使用 QQmlApplicationEngine 加载 Qt Quick 程序	26
1.4 你好世界!	27
1.5 初识 Qt Quick 控件	30
1.6 在 Qt Quick 中使用着色器	32
1.7 使用 C++ 扩展 Qt Quick	36
第 2 章 Qt Quick 基础	41
2.1 QML 语法	41
2.1.1 文件编码	41

2.1.2	注释与帮助	41
2.1.3	属性	42
2.1.3.1	基本类型	42
第 3 章	从 C++ 扩展 Qt Quick	45
第 4 章	状态机及动画	47
第 5 章	粒子系统	49
第 6 章	特效	51
6.1	导引	51
6.2	Blend	53
6.3	BrightnessContrast	59
6.4	ColorOverlay	61
6.5	Colorize	63
6.6	Desaturate	64
6.7	GammaAdjust	65
6.8	HueSaturation	66
6.9	LevelAdjust	68
6.10	ConicalGradient	69
6.11	LinearGradient	70
6.12	RadialGradient	71
6.13	Displace	73
6.14	DropShadow	74
6.15	InnerShadow	75
6.16	FastBlur	77
6.17	GaussianBlur	78
6.18	MaskedBlur	79
6.19	RecursiveBlur	80
6.20	DirectionalBlur	82
6.21	RadialBlur	83
6.22	ZoomBlur	84
6.23	Glow	85
6.24	RectangularGlow	87
6.25	OpacityMask	88
6.26	ThresholdMask	89
第 7 章	多媒体	93
第 8 章	富文本及图表	95

目录	iii
第 9 章 控件	97
第 10 章 模型视图	99
10.1 自定义表模型	99
10.1.1 使用 QtCreator 快速创建模型	99
10.2 自定义树模型	101
附录	103
图片索引	103
表格索引	103
源码索引	103
命令索引	104
路径索引	104

前言

Qt往往被认为是一套跨平台的图形界面开发架构。诚然，Qt对于图形界面支持的很好，并且，这一方面被越来越多的团队所接纳。但Qt并不仅限于开发图形界面，它其实是一种更加通用的客户端开发架构。Qt几乎提供了用于构建一个客户端所需的所有模块，包括但不限于蓝牙模块、串口模块、音频模块、网络模块、多媒体模块、数据库模块……

更加令用户愉悦的是，由于Qt本身是被广泛使用的开源产品，用户可以轻松的享受到来自整个开源社区（其中包括整个C/C++社区）的加持。也就是说，即使Qt本身并未提供一些方面的支持（或者Qt自身提供的支持无法满足要求），用户也可以轻松的找到免费或付费的解决方案。即使有些情况下用户无法找到解决麻烦的现成并有效的手段，但至少通过社区，用户可以获得一些走出困境的灵感。

随着新的硬件设备的广泛采用和开发者观念的变更。完全采用C++这类静态计算机语言开发图形界面变得越来越笨手笨脚，并且最终效果亦不佳，很多由动态语言轻松可以达到的效果往往用静态计算机语言难以实现。所幸的是，Qt一直没有停下前行的脚步。Qml以及基于Qml的Qt Quick被引入和大力推广。Qml本身被设计为一种简单而优雅的脚本语言，并且，Qml天然支持一个JavaScript子集¹。用户可以安心的用C++做基础模块，而利用Qt Quick将一切快速的组织起来。

Qt Quick比传统的Qt Widgets不仅仅更加有效利用CPU多核资源（Qt Quick可以异步渲染）。更令人高兴的是，Qt Quick完全是在显卡端完成渲染。即使某些设备不支持显卡渲染，Qt自身也可以通过软件模拟达到效果。这一切并不受限于某几个平台，而是几乎所有平台。智能手机、个人电脑、嵌入式设备，它们都受到支持。用户可以使用Qt Quick敏捷的构造出美观、高效、稳健并跨平台的一流产品。

Qt公司为用户提供了大量的辅助工具。用这些工具，用户可以迅速的编写、测试、调试、部署、以及调优和美化。除了Qt公司直接提供的工具外，由于Qt的广泛使用，很多第三方工具链也支持Qt。虽然，到目前为止，这些第三方支持主要是面向传统Qt C++。但仅仅来自Qt自身的工具链对于Qt Quick的支持也不会令用户失望。

本书是一本完整介绍Qt Quick的书。通过本书，读者可以完整的掌握整个Qt Quick的全貌。但限于篇幅和个人精力所限，一些细节可能被舍弃。

读者在阅读本书之前应当对于Qt C++、JavaScript和OpenGL具有一定了解，并具备一定的图形学相关知识。为了避免本书变成数千页的大部头，本书并不会对上述细节多做解释。

¹ 主要在Qml中禁用了JavaScript中this这个语义模糊的关键字，并禁用JavaScript中的全局变量。

基于 Qt Qml 的另一个模块是 Qt 3D。Qt 3D 和 Qt Quick 是两个几乎不关联的模块，虽然它们都基于 Qt Qml。本书并不介绍 Qt 3D。

本书采用 C++ 17 标准编写，Qt 最低版本为 Qt 5.12.0，FFMPEG 版本最低为 4.1，Boost 版本为 1.69.0，如果读者在编译本书代码出现了问题，请尝试查询当前开发环境是否正确。本书只支持桌面 Windows 和桌面 Linux，这两个平台已经足够涵盖绝大多数读者，并能够完整诠释所有技术细节。对于刚刚接触 Qt Quick 的读者，一方面太多平台细节会成为干扰读者统揽全局的噪音；另一方面对于一个具体的平台，本书采用的一些技术特性可能不被支持²，这些细节将极大拖慢本书的写作进度和涵盖的范围。为了向读者展示一个全面的并且现代的 Qt，本书不得不舍弃过多的平台适配而轻装上阵。

本书第 1 章带领读者纵览整个 Qt Quick，对于 Qt Quick 不太熟悉的读者可能读起来有些吃力。对于第 1 章，初次阅读起来有些困难的地方直接跳过即可。

本书第 2 章介绍 Qt Qml 基础语法以及 Qt Quick 基本元素，第 3 章介绍如何使用 C++ 扩展 Qt Quick，第 9 章介绍 Qt Quick 基础控件，这三章是主干章节。

第 4 章介绍 Qt Quick 动画和状态机，第 5 章介绍 Qt Quick 粒子系统，第 6 章介绍一些常见特效，第 8 章介绍 Qt Quick 的图文表模块，第 10 章介绍 Qt Quick 的模型视图模块。第 7 章本书介绍如何结合 FFMPEG 构建多媒体模块。这些章节各有主题，读者根据需要选读即可。

为了行文方便，本书转义了一些特殊符号如下表：

特殊符号	意义
ʌ	一个符合上下文的集合
⊓	一个符合特定集合的非空元素

² 一些平台可能只支持有限的 C++ 17 标准和 OpenGL 特性，另一方面在一些平台下 FFMPEG 也难以编译，而且 Qt 自带的多媒体库在特定平台下如何部署也千差万别，甚至有些平台可能根本不支持 Qt 5.12.0 及其后续版本而只支持一些老的 Qt 版本。

第1章 Qt Quick入门导引

1.1 搭建开发环境

1.1.1 在 Windows 平台下搭建开发环境

1.1.1.1 在 Windows 平台下安装 Qt

读者可以到 <http://download.qt.io/archive/> 下载最新的 Qt 运行环境。然而，遗憾的是，从 Qt 5.12.0 开始从此网址下载的 Windows 平台下的 Qt 开发环境并不完整。读者不得不访问 Qt 官网 <https://www.qt.io>，注册 Qt 帐号，然后按照流程下载在线安装包。不得不说，这对初学者很不友好。幸运的是，目前 Qt 网站有一个漏洞。读者可以直接访问 <https://www.qt.io/download-thank-you>，点击“here”下载在线安装包。如图 1.1.1。

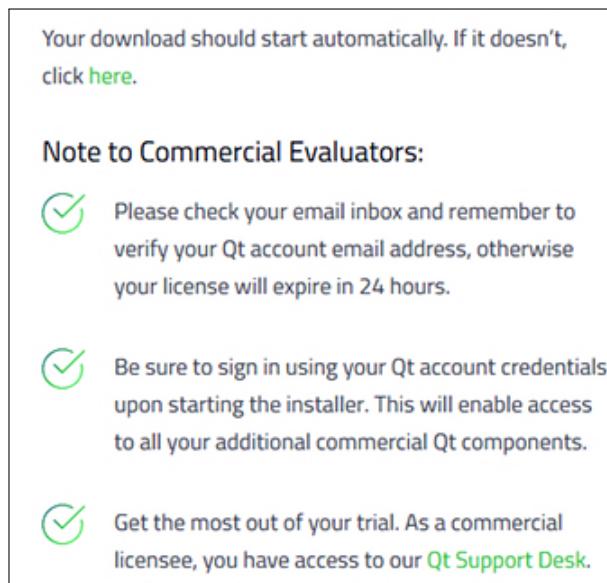


图 1.1.1 : Qt 在线安装包下载路径

以管理员身份运行在线安装包，选择安装路径时请不要选择包含空格和中文字符的路径。虽然现代开发环境对于空格和中文字符支持良好，但是，很多第三方辅助工具未必支持空格和中文字符。包括本书自带的辅助工具也不保证支持空格和中文。

在 Windows 平台下，建议读者选择安装“MSVC 2017 64-bit”或以上版或者“MinGW 7.3.0 64-bit”或以上版本。Qt 选择 5.12.0 或以上版本。安装的时候最好选择安装“Sources”、“Qt Charts”、“Qt WebEngine”以及“Qt Debug Information Files”

这些模块。在“Tools”选项下组好安装“CDB”以及对应的“MinGW”。本书建议最小安装如图 1.1.2。

图 1.1.2

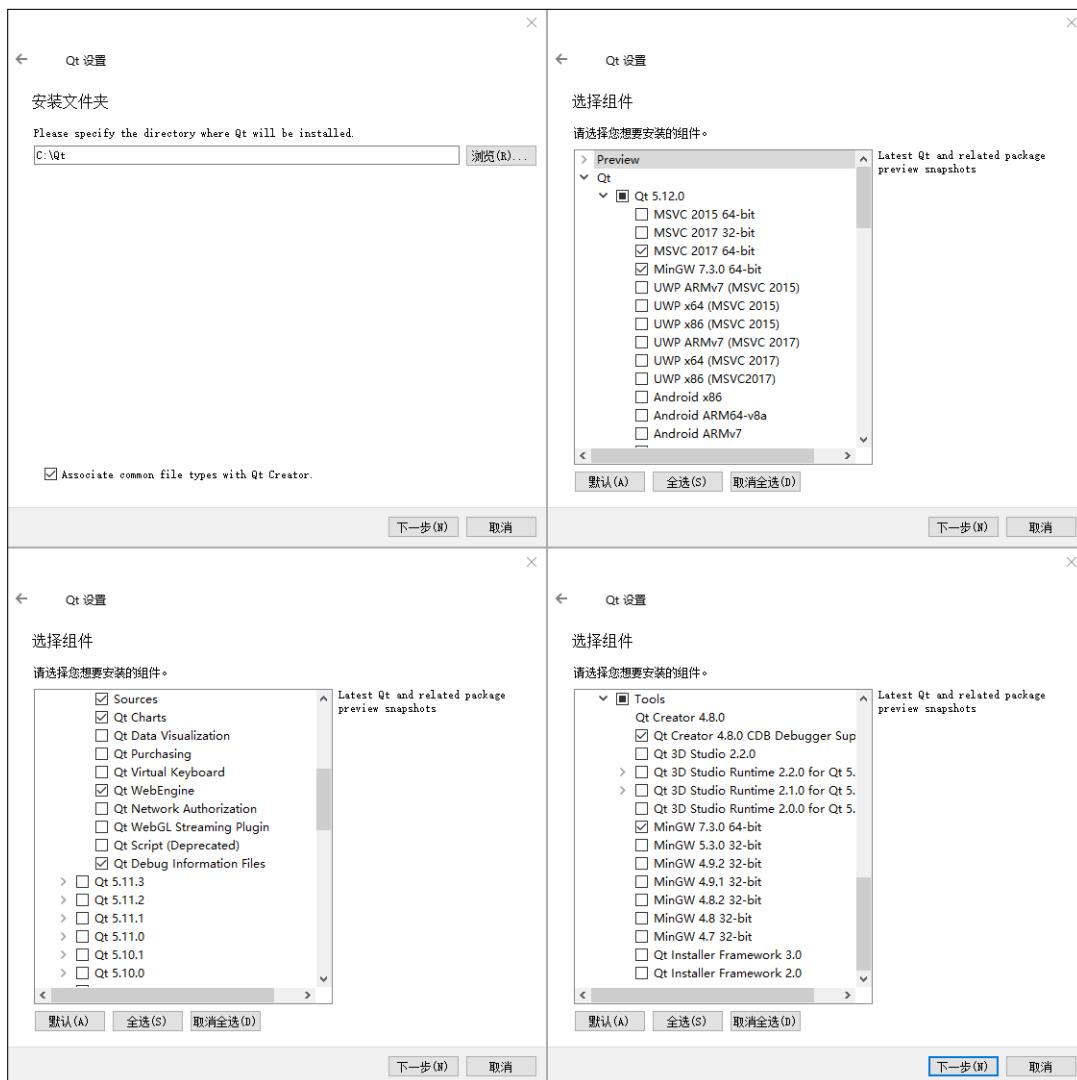


图 1.1.2 : Qt 在线安装建议安装组件

1.1.1.2 在 Windows 平台下安装 Boost

读者需要到 Boost 官网 <https://www.boost.org/> 下载最新 Boost 稳定版。解压缩，将“boost”文件夹复制到 Qt Include 路径。

比如，用户的 Qt Include 路径为：

C:\Qt\Qt5.12.0\5.12.0\msvc2017_64\include

复制完 boost 之后，应当存在路径：

C:\Qt\Qt5.12.0\5.12.0\msvc2017_64\include\boost

当然，读者也可以采用“mklink”建立链接代替拷贝。

如果读者使用的是 Visual Studio 自带的编译器，则需要使用“Visual Studio 命令提示符”运行命令 1.1.1。并将编译结果的“*.lib”文件拷贝到 Qt 根目录下的 lib 文件夹，将“*.dll”文件拷贝到 Qt 根目录下的 bin 文件夹。

命令 1.1.1

```
1 cd /D < Boost 源代码路径 >
```

```

2 bootstrap.bat
3 bjam —build-type=complete
    —toolset=< MSVC 版本比如: msvc-14.1 >
    address-model=64
    Link=shared
    runtime-link=shared
    threading=multi

```

命令 1.1.1

之后，读者需要根据编译输出更新“QtQmlBook/msvc_boost.pri”文件。如源码 1.1.1 所示：

源码 1.1.1

```

CONFIG(debug,debug|release){
    LIBS += "boost_atomic-vc141-mt-gd-x64-1_68.lib"
    .....
} else{
    LIBS += "boost_atomic-vc141-mt-x64-1_68.lib"
    .....
}

```

源码 1.1.1

如果读者使用的是 MinGW 环境，则需要使用“MinGW 命令提示符”运行命令 1.1.2。并将编译结果的“*.a”文件拷贝到 Qt 根目录下的 lib 文件夹，将“*.dll”文件拷贝到 Qt 根目录下的 bin 文件夹。

命令 1.1.2

```

1 cd /D < Boost 源代码路径 >
2 bootstrap.bat
3 bjam —build-type=complete
    —toolset=gcc
    address-model=64
    Link=shared
    runtime-link=shared
    threading=multi

```

命令 1.1.2

之后，读者需要根据编译输出更新“QtQmlBook/mingw_boost.pri”文件。如源码 1.1.2 所示：

源码 1.1.2

```

CONFIG(debug,debug|release){
    LIBS += "libboost_atomic-mgw73-mt-d-x64-1_68.dll"
    .....
} else{
    LIBS += "libboost_atomic-mgw73-mt-x64-1_68.dll"
    .....
}

```

源码 1.1.2

1.1.1.3 在 Windows 平台下 MinGW 配置 jemalloc

对于 C++ 来说，小对象的内存碎片问题向来很棘手。一般而言，使用 tcmalloc 或 jemalloc 可以有效避免内存碎片问题。

在 Linux 平台或类似平台下，可以使用“LD_PRELOAD”或类似的技术轻松的覆盖动态链接库中的函数。因而，在 Linux 平台下，使用 tcmalloc 或 jemalloc 替换 C 库中的内存分配函数是简易的。

而在 Windows 平台下，覆盖动态库中的函数相当复杂。为了能够使得本书的示

例代码不是玩具,本书在 Windows 平台下使用 jemalloc 克服小对象内存碎片。

当使用 MSVC 编译器的时候,本书直接嵌入 jemalloc 源代码,因而读者不必特别操心。但是,当在 Windows 下使用 MinGW 编译器时,读者需要自己静态编译 jemalloc¹。并将编译结果放置到:

QtQmlBook\ssstd_library\memory\libs

文件夹下。并将文件重命名为“jemalloc_win64_mingw_730.a”。如果读者实在无法静态编译 jemalloc,读者可以找到:

QtQmlBook\ssstd_library_ssstd_library_memory.pri

并将此文件内容清空²。

1.1.2 在 Linux 平台下搭建开发环境

Linux 有众多发行版,如果读者是首次在 Linux 平台下搭建 Qt 开发环境,建议读者使用 Ubuntu 等使用者较多的版本。

1.1.2.1 在 Linux 平台下安装 Qt

如命令 1.1.3,所示:

- 第 1 行命令用于安装基本 C++ 开发环境;
- 第 2 行命令用于安装 OpenGL 环境;

命令 1.1.3

```
1 sudo apt-get install build-essential
2 sudo apt-get install libgl1-mesa-dev
```

读者可以参照 1.1.1.1 节相关内容下载最新的 Qt 开发包并安装,如命令 1.1.4 所示:

- 第 1 行命令用于赋予 Qt 安装包执行权限;
- 第 2 行运行安装包;

命令 1.1.4

```
1 chmod +x qt-opensource-Linux-x64-5.12.0.run
2 ./qt-opensource-Linux-x64-5.12.0.run
```

在不同 Linux 发行版本这些命令有所不同,即使是同一发行版本,随着时间推移命令也会有所变化。

读者可以访问 <https://wiki.qt.io/Main> 获得更加详细的帮助。

1.1.2.2 在 Linux 平台下安装 Boost

在 Linux 下安装 Boost 极其简单,只需要执行命令 1.1.5 即可。

¹ 如果读者使用 MinGW 7.3 64 bit 版本的编译器,本书已经将对应版本的 jemalloc 编译好了,读者不需要再次编译。

² 注意不要删除这个文件,而只是删除此文件内容。

命令 1.1.5

```
1 sudo apt-get install libboost-all-dev
```

命令 1.1.5

1.2 qmake 入门

qmake 类似于 cmake，但 qmake 比 cmake 更加简洁清晰。如果读者希望写一个跨平台的通用库的话，或许 cmake 是比 qmake 更加优异的选择。但读者明确是写一个特定的应用程序的话，qmake 就比 cmake 优秀的多。qmake 比 cmake 功能较少，但从另一个角度，qmake 比 cmake 更加专注。通过本节，读者会发现只需要学习可怜的一点内容，就可以使用 qmake 搭建出复杂的程序架构。不过，本书毕竟是一门专门写 Qt Quick 的书，不可能介绍 qmake 的每一个细节。

1.2.1 使用 qmake 构建 Hellow World！

读者新建一个目录³，在此文件夹下新建一个“hellow_world.pro”文件，输入文件内容如 源码 1.2.1。在此文件夹下建立“main.cpp”文件，输入内容如 源码 1.2.2。

源码 1.2.1

```
1 QT -= gui
2 QT -= core
3
4 CONFIG += console
5
6 CONFIG(debug, debug|release){
7     TARGET = hellow_word_debug
8 }else{
9     TARGET = hellow_word
10 }
11
12 TEMPLATE = app
13
14 win32-msvc*{
15     QMAKE_CXXFLAGS += /std:c++latest
16 }else{
17     CONFIG += c++17
18 }
19
20 SOURCES += $$PWD/main.cpp
21 DESTDIR = $$PWD
22
23 DEFINES *= NUMBER=1
24 DEFINES *= HELLOW=\\"Hello\\\""
25 DEFINES += QT_DEPRECATED_WARNINGS
```

源码 1.2.1

源码 1.2.2

```
1 #include <iostream>
2
3 int main(int , char **) {
4     if constexpr(NUMBER) {
5         std::cout << HELLOW " World! "
```

³ 本书所有目录都要求不包含空格和中文，以后不再赘述。

源码 1.2.2

```

6 }           << std::endl;
7 }
8 }
```

使用 Qt Creator 打开“hellow_world.pro”，运行此项目。

现在来分析一下源码 1.2.1：

- 第 1~2 行表示不使用 Qt 库；
- 第 4 行表示这是一个控制台应用程序；
- 第 6~10 行表示在 debug 模式下输出目标名称是“hellow_world_debug”，在 release 模式下输出目标名称是“hellow_world”；
- 第 12 行表示输出的是一个应用程序；
- 第 14~18 行表示使用 C++ 17 标准；
- 第 20 行将“main.cpp”加入编译过程；
- 第 21 行规定输出目录就是当前“pro”文件所在目录；
- 第 23 行定义了一个叫“NUMBER”的宏，宏的值是一个数字；
- 第 24 行定义了一个叫“HELLOW”的宏，宏的值是一个字符串；
- 第 25 行定义了一个叫“QT_DEPRECATED_WARNINGS”的宏，这个宏没有定义值；

不难发现 qmake 的语法十分简单：

- “=” 代表赋值；
- “+=” 代表向变量中增加元素；
- “-=” 代表从变量中删除元素；
- “*=” 代表如果变量中不存在则加入元素否则忽略；
- “~=” 代表替换变量中的值；
- “\$\$” 代表当 qmake 运行时，变量的字面值；
- “\$” 代表当 qmake 生成 Makefile 后，变量的字面值；
- “#” 代表注释；
- “SOURCES” 代表需要编译的 C/C++ 源代码变量；
- “HEADERS” 代表 C/C++ 头文件变量；
- “DEFINES” 代表 C/C++ 宏变量；
- “TARGET” 代表输出对象名称；
- “CONFIG” 用来加入和检查 Qt 中预定义的编译选项；
- “QMAKE_CXXFLAGS” 代表 qmake 生成 Makefile 时需要加入的编译器参数；
- “TEMPLATE” 决定此项目的模板类型，本案例是使用应用程序模板“app”，顾名思义此模板的目标是生成应用程序。后续章节会介绍更多模板；

第 6~10 行和 14~18 虽然写法不同，实际上都是检查“CONFIG”中是否定义了特定项。读者可以尝试一下向文件“hellow_world.pro”文件最后加入源码 1.2.3，分别去掉源码 1.2.3 第一行和保留第一行，观察 Qt Creator 的“概要信息”输出什么。

源码 1.2.3

```
1 CONFIG += mydebug
```

```

2 mydebug{
3     message("find my debug")
4 }else{
5     message("can not find my debug")
6 }

```

源码 1.2.3

1.2.2 使用 qmake 创建动态链接库

绝大多数项目的项目结构都很复杂，从这一节开始读者要开始接受这一事实。本节示例的项目结构如路径 1.2.1 所示。

路径 1.2.1

```

.
├── import_library.pro
└── test_library
    ├── import_test_library.pri
    ├── TestLibrary.cpp
    ├── TestLibrary.hpp
    └── test_library.pro
└── the_app
    ├── main.cpp
    └── the_app.pro

```

路径 1.2.1

先来看看“import_library.pro”文件，如源码 1.2.4 所示。

此文件启用了一个新的模版，“subdirs”。

“subdirs”模版可以将一系列孤立的工程组织起来⁴，并要求它们按照一定先后顺序编译。比如本节采用的“CONFIG += ordered”就要求项目按照定义顺序编译。

源码 1.2.4

```

1 #import_library.pro
2 TEMPLATE = subdirs
3
4 CONFIG += ordered
5
6 test_library.file = $$PWD/test_library/test_library.pro
7 SUBDIRS += test_library
8
9 the_app.file = $$PWD/the_app/the_app.pro
10 SUBDIRS += the_app

```

源码 1.2.4

再来看看“the_app.pro”文件，如源码 1.2.5 所示。它采用了“app”模版。比起上一节，它多了一些新的知识点。

- 第 21~23 行更改了在非 Windows 平台下程序的链接参数，它要求程序运行时将其所在目录加入动态库搜索路径；
- 第 28 行将另一个文件引入此文件，它和 C/C++ 的“#include”工作原理一致；

源码 1.2.5

```

1 #the_app.pro
2 QT += gui
3 QT += core
4

```

⁴ 最好不要嵌套引用 subdirs，某些 IDE 并不支持。

```

5 CONFIG += console
6
7 CONFIG(debug, debug|release){
8     TARGET = the_app_debug
9 }else{
10    TARGET = the_app
11 }
12
13 TEMPLATE = app
14
15 win32-msvc*{
16     QMAKE_CXXFLAGS += /std:c++latest
17 }else{
18     CONFIG += c++17
19 }
20
21 !win32 {
22     QMAKE_LFLAGS += -Wl,-rpath .
23 }
24
25 DESTDIR = $$PWD/../bin
26
27 SOURCES += $$PWD/main.cpp
28 include($$PWD/../../test_library/import_test_library.pri)

```

源码 1.2.5

接下来是“import_test_library.pri”文件，如源码 1.2.6 所示。它也引入了一些新的知识。

- 第 2 行使用“INCLUDEPATH”变量将当前目录加入 C/C++ 包含路径搜索路径；
- 第 3~7 行使用“LIBS”变量导入 C/C++ 链接库，“-L”后面是库所在路径，“-l”后面紧跟库的名称；

源码 1.2.6

```

1 #import_test_library.pri
2 INCLUDEPATH += $$PWD
3 CONFIG(debug, debug|release){
4     LIBS += -L$$PWD/../bin -ltest_libraryd
5 }else{
6     LIBS += -L$$PWD/../bin -ltest_library
7 }

```

源码 1.2.6

然后，我么来看一下如何使用 qmake 定义一个动态链接库。一切与定义应用程序没什么不同，只是将“TEMPLATE = app”改成了“TEMPLATE = lib”，如源码 1.2.7 第 13 行所示。

源码 1.2.7

```

1 #test_library.pro
2 QT += gui
3 QT += core
4
5 CONFIG += console
6
7 CONFIG(debug, debug|release){
8     TARGET = test_libraryd
9 }else{
10    TARGET = test_library
11 }

```

```

11 }
12
13 TEMPLATE = lib
14
15 win32-msvc*{
16     QMAKE_CXXFLAGS += /std:c++latest
17 }else{
18     CONFIG += c++17
19 }
20
21 !win32 {
22     QMAKE_LFLAGS += -Wl,-rpath .
23 }
24
25 SOURCES += $$PWD/TestLibrary.cpp
26 HEADERS += $$PWD/TestLibrary.hpp
27
28 DESTDIR = $$PWD/../bin
29 DEFINES *= D_TEST_LIBRARY

```

源码 1.2.7

剩下的是“TestLibrary.hpp”(如源码 1.2.8),“TestLibrary.cpp”(如源码 1.2.9)和“main.cpp”(如源码 1.2.10)。都是标准 C++, 本书不赘述。

源码 1.2.8

```

1 /*TestLibrary.hpp*/
2 #pragma once
3
4 #include <QtCore/qglobal.h>
5
6 #ifndef D_TEST_LIBRARY
7 #define TEST_LIBRARY_EXPORT Q_DECL_IMPORT
8 #else
9 #define TEST_LIBRARY_EXPORT Q_DECL_EXPORT
10#endif
11
12 class TEST_LIBRARY_EXPORT TestClass {
13 public:
14     TestClass();
15     ~TestClass();
16     void foo();
17 };

```

源码 1.2.8

源码 1.2.9

```

1 /*TestLibrary.cpp*/
2 #include "TestLibrary.hpp"
3 #include <iostream>
4
5 TestClass::TestClass() {
6 }
7
8 TestClass::~TestClass() {
9 }
10
11 void TestClass::foo() {
12     std::cout << __func__ << std::endl;
13 }

```

源码 1.2.9

源码 1.2.10

```

1 /*main.cpp*/

```

```

2 #include <TestLibrary.hpp>
3
4 int main(int, char **) {
5     TestClass varClass;
6     varClass.foo();
7     return 0;
8 }
```

源码 1.2.10

1.2.3 qmake 高级用法

qmake 远比读者想象的要复杂的多，本节向读者展示一些常见功能如何使用 qmake 实现。

图 1.2.1

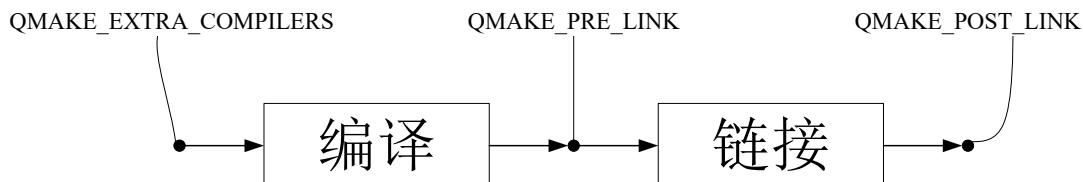


图 1.2.1 : qmake 对 C/C++ 编译链接过程中的控制点

如图 1.2.1 一个 C/C++ 程序编译至少可以抽象出三个节点,源代码编译前,链接前以及链接后。这三个时刻分别对应于 qmake 变量:QMAKE_EXTRA_COMPILERS, QMAKE_PRE_LINK 以及 QMAKE_POST_LINK。

使用这三个控制变量,用户可以在这三个时刻执行自定义命令。

本节代码树如路径 1.2.2 所示,“advance_use_qmake.pro”文件如源码 1.2.11 所示。

路径 1.2.2

```

.
├── advance_use_qmake.pro
├── after_run
│   ├── after_run.pro
│   └── main.cpp
├── before_run
│   ├── before_run.pro
│   └── main.cpp
├── new_moc
│   ├── main.cpp
│   └── new_moc.pro
└── the_run
    ├── main.cpp
    ├── test1.hpp
    ├── test2.hpp
    └── the_run.pro

```

路径 1.2.2

源码 1.2.11

```

1 #advance_use_qmake.pro
2 TEMPLATE = subdirs
3
4 CONFIG += ordered
5
6 new_moc.file = $$PWD/new_moc/new_moc.pro
7 SUBDIRS += new_moc
8
```

```

9 before_run.file = $$PWD/before_run/before_run.pro
10 SUBDIRS += before_run
11
12 after_run.file = $$PWD/after_run/after_run.pro
13 SUBDIRS += after_run
14
15 the_run.file = $$PWD/the_run/the_run.pro
16 SUBDIRS += the_run

```

源码 1.2.11

本案例向读者展示：

- 在编译开始前, qmake 调用“程序 new_moc”自动生成 cpp 文件并加入编译过程；
- 在链接前 qmake 调用“程序 before_run”,“程序 before_run”向“the_run 文件夹”下建立一个“before_run.txt 文件”；
- 在链接完成后 qmake 调用“程序 after_run”,“程序 after_run”向“the_run”文件夹下建立一个“after_run.txt 文件”；

主要分析一下“the_run.pro”(如源码 1.2.12)。

- 第 28~41 行展示了如何使用“QMAKE_EXTRA_COMPILERS”。
“QMAKE_EXTRA_COMPILERS”往往用于自定义一种“编译时编译”规则，实际上 Qt 的 moc 就是这么实现的。读者可以用此技术实现自定义代码生成器，不过这需要读者有编译原理相关知识。
- 第 44~49 行展示了如何使用“QMAKE_PRE_LINK”。
实际上，对于一般用户，“QMAKE_PRE_LINK”并不常用。除非读者要实现类似将其它编译器编译的二进制文件加入本次编译过程的功能。
- 第 52~57 行展示了如何使用“QMAKE_POST_LINK”。
“QMAKE_POST_LINK”往往用来自定义“make install”。虽然 qmake 有默认的“make install”规则。不过，本书并不准备介绍。因为，一个实际应用程序的“make install”往往不是简单的拷贝，而是需要对文件进行加密、压缩或者对文件进行语法检查等额外的任务。而利用 C++ 17 的 filesystem 模块自己实现一个单纯的拷贝程序并不复杂。因而，本书介绍更加通用的“QMAKE_POST_LINK”，而不介绍 qmake 的专用语法。

源码 1.2.12

```

1 #the_run.pro
2 QT -= gui
3 QT -= core
4
5 CONFIG += console
6
7 CONFIG(debug,debug|release){
8     TARGET = the_run_debug
9 }else{
10     TARGET = the_run
11 }
12
13 TEMPLATE = app
14
15 win32-msvc*{

```

```

16     QMAKE_CXXFLAGS += /std:c++latest
17 }else{
18     CONFIG += c++17
19     LIBS += -lstdc++fs
20 }
21
22 SOURCES += $$PWD/main.cpp
23 DESTDIR = $$PWD/../bin
24
25 DEFINES += QT_DEPRECATED_WARNINGS
26
27 #when before build new_moc will call ...
28 new_moc.dependency_type = TYPE_C
29 new_moc.variable_out = SOURCES
30 new_moc.output = moc_new_${QMAKE_FILE_BASE}.cpp
31 CONFIG(debug,debug|release){
32     new_moc.commands = \
33 $$${DESTDIR}/new_moc_debug ${QMAKE_FILE_NAME} ${QMAKE_FILE_OUT}
34 }else{
35     new_moc.commands = \
36 $$${DESTDIR}/new_moc ${QMAKE_FILE_NAME} ${QMAKE_FILE_OUT}
37 }
38 NEW_MOC_HEADERS = test2.hpp test1.hpp
39 new_moc.input = NEW_MOC_HEADERS
40 QMAKE_EXTRA_COMPILERS += new_moc
41 export(QMAKE_EXTRA_COMPILERS)
42
43 #when link started before_run will call ...
44 CONFIG(debug,debug|release){
45     QMAKE_PRE_LINK += $$${DESTDIR}/before_run_debug $$PWD
46 }else{
47     QMAKE_PRE_LINK += $$${DESTDIR}/before_run $$PWD
48 }
49 export(QMAKE_PRE_LINK)
50
51 #when link finished after_run will call ...
52 CONFIG(debug,debug|release){
53     QMAKE_POST_LINK += $$${DESTDIR}/after_run_debug $$PWD
54 }else{
55     QMAKE_POST_LINK += $$${DESTDIR}/after_run $$PWD
56 }
57 export(QMAKE_POST_LINK)

```

源码 1.2.12

其余的,“before_run.pro”(如源码 1.2.13)、“before_run/main.cpp”(如源码 1.2.14)、“after_run.pro”(如源码 1.2.15)、“after_run/main.cpp”(如源码 1.2.16)、“new_moc.pro”(如源码 1.2.17)、“new_moc/main.cpp”(如源码 1.2.18)和“the_run/main.cpp”(如源码 1.2.19)没有新知识点,本书不赘述。

源码 1.2.13

```

1 #before_run.pro
2 QT -= gui
3 QT -= core
4
5 CONFIG += console
6
7 CONFIG(debug,debug|release){
8     TARGET = before_run_debug
9 }else{
10     TARGET = before_run

```

```

11 }
12
13 TEMPLATE = app
14
15 win32-msvc*{
16     QMAKE_CXXFLAGS += /std:c++latest
17 }else{
18     CONFIG += c++17
19     LIBS += -lstdc++fs
20 }
21
22 SOURCES += $$PWD/main.cpp
23 DESTDIR = $$PWD/../bin
24
25 DEFINES += QT_DEPRECATED_WARNINGS

```

源码 1.2.13

源码 1.2.14

```

1 /*main.cpp*/
2 #if __has_include(<filesystem>)
3 #include <filesystem>
4 namespace fs = std::filesystem;
5 #else
6 #include <experimental/filesystem>
7 namespace fs = std::experimental::filesystem;
8 #endif
9
10 #include <iostream>
11 #include <fstream>
12 #include <chrono>
13
14 class OStream final : public std::ofstream {
15     using Super = std::ofstream;
16 public:
17     template<typename T,
18             typename = std::enable_if_t<
19                 std::is_constructible_v<Super, T && > > >
20             inline OStream(T && arg) :
21                 Super(std::forward<T>(arg)) {
22         }
23     template<typename T,
24             typename = void,
25             typename = std::enable_if_t<
26                 !std::is_constructible_v<Super, T && > > >
27             inline OStream(T && arg) :
28                 Super(std::forward<T>(arg).string()) {
29         }
30     };
31
32 /* 在特定文件夹下建立一个before_run.txt
33 * 并输出程序运行时时间戳 */
34 int main(int argc, char ** argv) {
35     std::cout << "before_run : "
36     << argc << std::endl;
37     if (argc < 2) {
38         return -1;
39     }
40     fs::path varPath{ argv[1] };
41     OStream stream{ varPath / "before_run.txt" };
42     stream << std::chrono::
43         high_resolution_clock::now()

```

源码 1.2.14

```

44     .time_since_epoch().count());
45     stream << std::endl;
46     return 0;
47 }
```

源码 1.2.15

```

1 #after_run.pro
2 QT -= gui
3 QT -= core
4
5 CONFIG += console
6
7 CONFIG(debug, debug|release){
8     TARGET = after_run_debug
9 }else{
10     TARGET = after_run
11 }
12
13 TEMPLATE = app
14
15 win32-msvc*{
16     QMAKE_CXXFLAGS += /std:c++latest
17 }else{
18     CONFIG += c++17
19     LIBS += -lstdc++fs
20 }
21
22 SOURCES += $$PWD/main.cpp
23 DESTDIR = $$PWD/../bin
24
25 DEFINES += QT_DEPRECATED_WARNINGS
```

源码 1.2.15

源码 1.2.16

```

1 /*main.cpp*/
2 #if __has_include(<filesystem>)
3 #include <filesystem>
4 namespace fs = std::filesystem;
5 #else
6 #include <experimental/filesystem>
7 namespace fs = std::experimental::filesystem;
8 #endif
9
10 #include <iostream>
11 #include <fstream>
12 #include <chrono>
13
14 class OStream final : public std::ofstream {
15     using Super = std::ofstream;
16 public:
17     template<typename T,
18             typename = std::enable_if_t<
19                 std::is_constructible_v<Super, T && > > >
20             inline OStream(T && arg) :
21             Super(std::forward<T>(arg)) {
22     }
23     template<typename T,
24             typename = void,
25             typename = std::enable_if_t<
26                 !std::is_constructible_v<Super, T && > > >
```

```

27     inline OStream(T && arg) :
28         Super(std::forward<T>(arg).string()) {
29     }
30 };
31
32 /* 在特定文件夹下建立一个after_run.txt
33 * 并输出程序运行时时间戳 */
34 int main(int argc, char ** argv) {
35     std::cout << "after_run : "
36     << argc << std::endl;
37     if (argc < 2) {
38         return -1;
39     }
40     fs::path varPath{ argv[1] };
41     OStream stream{ varPath / "after_run.txt" };
42     stream << std::chrono::
43         high_resolution_clock::now()
44         .time_since_epoch().count();
45     stream << std::endl;
46     return 0;
47 }
```

源码 1.2.16

源码 1.2.17

```

1 #new_moc.pro
2 QT -= gui
3 QT -= core
4
5 CONFIG += console
6
7 CONFIG(debug, debug|release){
8     TARGET = new_moc_debug
9 }else{
10     TARGET = new_moc
11 }
12
13 TEMPLATE = app
14
15 win32-msvc*{
16     QMAKE_CXXFLAGS += /std:c++latest
17 }else{
18     CONFIG += c++17
19     LIBS += -lstdc++fs
20 }
21
22 SOURCES += $$PWD/main.cpp
23 DESTDIR = $$PWD/../bin
24
25 DEFINES += QT_DEPRECATED_WARNINGS
```

源码 1.2.17

源码 1.2.18

```

1 /*main.cpp*/
2 #include <iostream>
3 #include <fstream>
4
5 #if __has_include(<filesystem>)
6 #include <filesystem>
7 namespace fs = std::filesystem;
8 #else
9 #include <experimental/filesystem>
```

```

10 namespace fs = std::experimental::filesystem;
11 #endif
12
13 /*生成一个用于测试的.cpp,用于在控制台输出“Good Luck! ” */
14 int main(int argc, char ** argv) {
15     std::cout << "new_moc : "
16         << argc << std::endl;
17     if (argc < 3) {
18         return -1;
19     }
20     std::ifstream varInput{ argv[1] };
21     std::ofstream varOutput{ argv[2] };
22     varOutput << "/******";
23     varOutput << std::endl;
24     varOutput << "#include \"";
25     varOutput << argv[1];
26     varOutput << "\"";
27     varOutput << std::endl;
28     varOutput << u8R"(inline static int a = [](){"
29         std::cout << "Good Luck!" << std::endl;
30         return 12;
31     }(); )";
32     varOutput << std::endl;
33     return 0;
34 }
```

源码 1.2.18

源码 1.2.19

```

1 /*main.cpp*/
2 #if __has_include(<filesystem>)
3 #include <filesystem>
4 namespace fs = std::filesystem;
5 #else
6 #include <experimental/filesystem>
7 namespace fs = std::experimental::filesystem;
8 #endif
9 #include <iostream>
10
11 int main(int, char **) {
12     std::cout << "the_run" << std::endl;
13     return 0;
14 }
```

源码 1.2.19

1.2.4 qmake 生成 Visual Studio 工程

使用 qmake 生成 Visual Studio 工程十分简单，其核心指令只有一条，如命令 1.2.1：

命令 1.2.1

命令 1.2.1

```
1 qmake -r -tp vc < 工程名称 >
```

在 Windows 平台下读者如果想在命令行下运行此命令需要设置好运行环境。

读者可以在 Qt 安装目录下找到“qtenv2.bat”文件。其中一个合法路径是：

“C:\Qt\Qt5.12.0\5.12.0\msvc2017_64\bin\qtenv2.bat”

读者要修改“qtenv2.bat”文件。32 位开发环境将“vcvarsall.bat”或 64 位开发环境将“vcvar64.bat”引入并执行。

如源码 1.2.20 第 5 行所示：

源码 1.2.20

```

1 @echo off
2 echo Setting up environment for Qt usage...
3 set PATH=C:\Qt1\5.12.0\msvc2017_64\bin;%PATH%
4 cd /D C:\Qt1\5.12.0\msvc2017_64
5 call "C:/Program Files (x86)/Microsoft Visual Studio/2017/Enterprise/VC/Auxiliary/Build/
    vcvars64.bat"
6 echo Remember to call vcvarsall.bat to complete environment setup!

```

源码 1.2.20

以后读者在 Windows 平台下运行“qtenv2.bat”就可以得到一个完整的运行环境了。

1.2.5 获得更多 qmake 帮助

本书所介绍的知识已经足够帮助读者搭建绝大多数大型复杂应用程序。但软件项目如此复杂，读者可能需要更进一步的知识才能解决手头的问题。

Qt 的帮助系统一向被认为是各个软件项目中最好的之一。读者只需要打开 Qt Creator，在帮助的索引搜索栏里面输入“qmake”，一切读者需要的信息就出现了。

- qmake 的所有控制变量

要获得 qmake 的所有控制变量帮助信息，只需要单击“qmake Variable Reference”即可，如图 1.2.2。

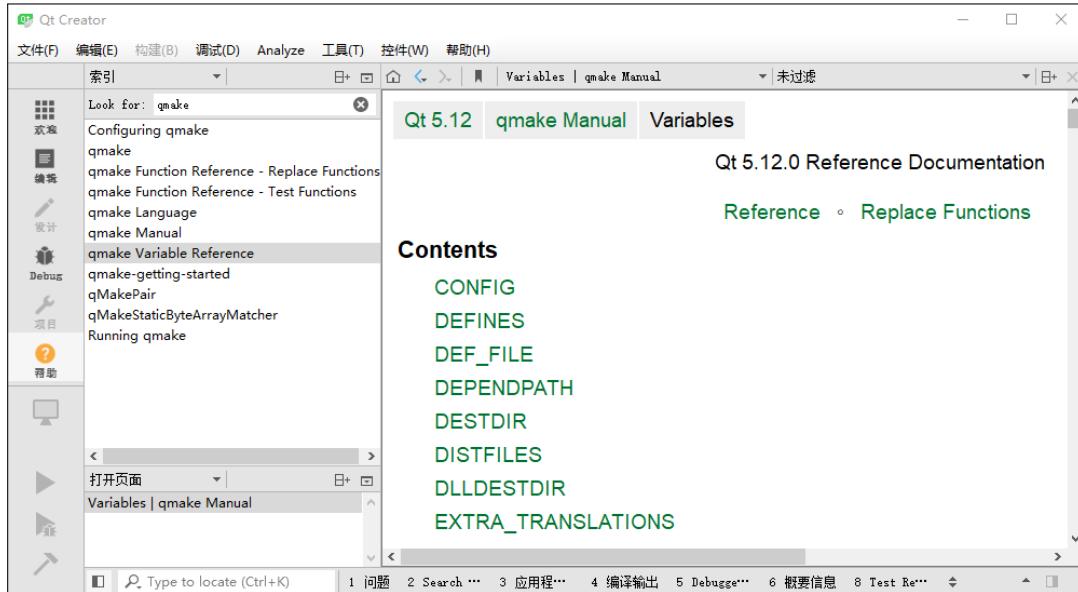


图 1.2.2

图 1.2.2 : qmake Variable Reference

- qmake 控制台运行参数

要获得 qmake 的所有控制台运行参数相关信息，只需要单击“Running qmake”即可，如图 1.2.3。

- qmake 的完整语法

要完整的了解 qmake 的所有语法，只需要单击“qmake Language”即可，如图 1.2.4。

图 1.2.3

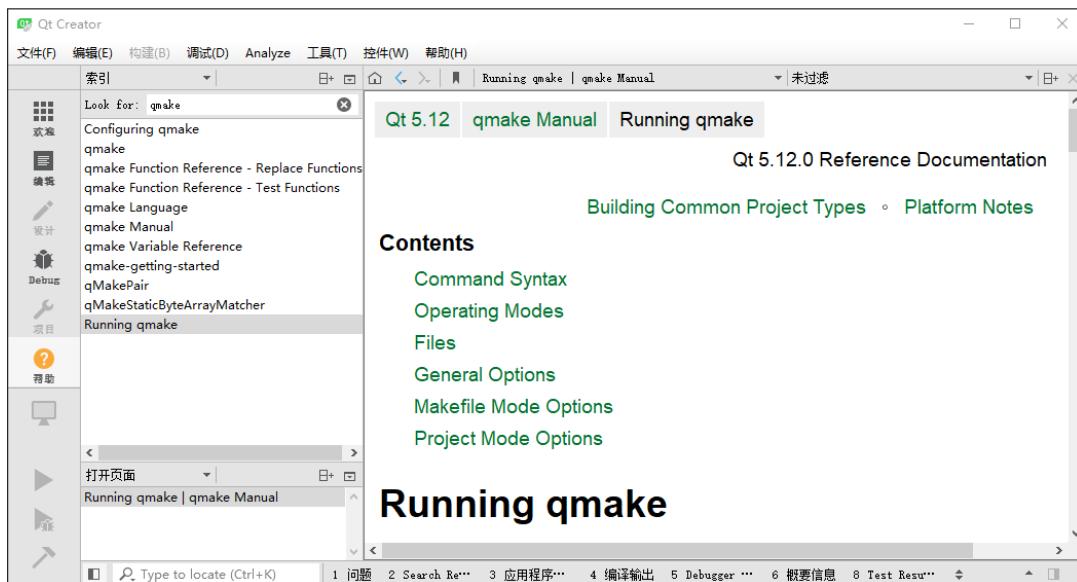


图 1.2.3 : Running qmake

图 1.2.4

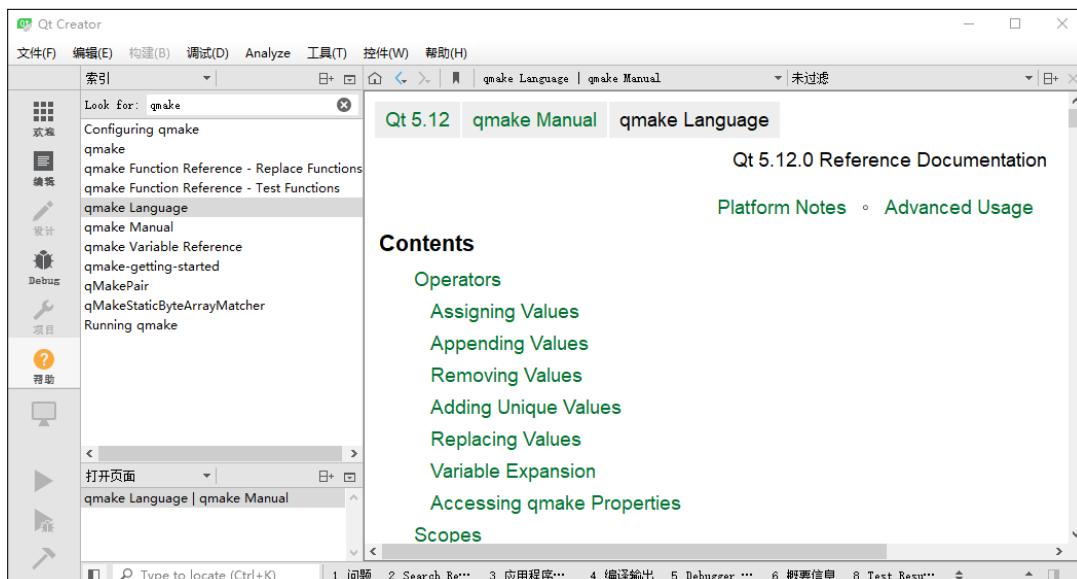


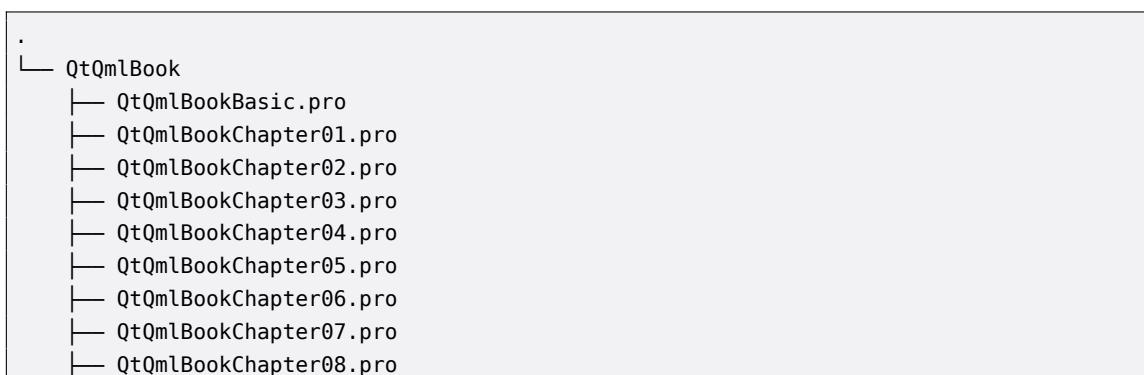
图 1.2.4 : qmake Language

1.3 第一个程序

1.3.1 本书的工程项目

本书的项目结构如路径 1.3.1 :

路径 1.3.1



```

├── QtQmlBookChapter09.pro
├── QtQmlBookChapter10.pro
└── QtQmlBookTest.pro

```

路径 1.3.1

- “QtQmlBookBasic.pro”此项目必须优先编译,其他所有项目依赖于此项目;
- “QtQmlBookChapter01.pro”~“QtQmlBookChapter10.pro”分别对应本书第 1 章到第 10 章随书代码;
- “QtQmlBookTest.pro”对应于本书的测试文件;

对于初学者可以使用 Qt Creator 分别打开上述工程文件进行编译。

在 Windows 平台下,也可以修改“build_msvc.bat”,从而使用 Visual Studio。

如源码 1.3.1 :

源码 1.3.1

```

1 call "C:/Qt/Qt5.12.0/5.12.0/msvc2017_64/bin/qtenv2.bat"
2 cd /D "E:/QtQmlBookMsvc"
3 qmake -r -tp vc "E:/QtQmlBook/QtQmlBookBasic.pro"
4 qmake -r -tp vc "E:/QtQmlBook/QtQmlBookChapter01.pro"
5 qmake -r -tp vc "E:/QtQmlBook/QtQmlBookChapter02.pro"
6 qmake -r -tp vc "E:/QtQmlBook/QtQmlBookChapter03.pro"
7 qmake -r -tp vc "E:/QtQmlBook/QtQmlBookChapter04.pro"
8 qmake -r -tp vc "E:/QtQmlBook/QtQmlBookChapter05.pro"
9 qmake -r -tp vc "E:/QtQmlBook/QtQmlBookChapter06.pro"
10 qmake -r -tp vc "E:/QtQmlBook/QtQmlBookChapter07.pro"
11 qmake -r -tp vc "E:/QtQmlBook/QtQmlBookChapter08.pro"
12 qmake -r -tp vc "E:/QtQmlBook/QtQmlBookChapter09.pro"
13 qmake -r -tp vc "E:/QtQmlBook/QtQmlBookChapter10.pro"
14 qmake -r -tp vc "E:/QtQmlBook/QtQmlBookTest.pro"
15 qmake -r -tp vc "E:/QtQmlBook/TheBook/TheBook.pro"
16 cmd

```

源码 1.3.1

- 第 1 行用于设置 Qt 运行环境;
- 第 2 行用于设置 Visual Studio 工程文件输出目录;
- 第 3~15 行用于指明将哪些 qmake 项目转为 Visual Studio 项目;

运行完上述命令后,读者即可使用 Visual Studio 打开相应的“sln”文件。

本节示例代码位于文件夹“QtQmlBook/chapter01/firstapplication”下。

源码 1.3.2 展示了该项目的工程文件:

- 第 11 行引入“outdirpath.pri”文件,此文件定义了输出路径。
- 第 13 行引入“cplusplus.pri”文件,此文件定义了标准 C++ 相关控制项。
- 第 15 行引入“import_sstd_library.pri”文件,此文件引入“sstd_library”库。“sstd_library”库用于引入和补充标准 C++ 库。
- 第 17 行引入“import_sstd_qt_and_qml_library.pri”文件,此文件引入“sstd_qt_and_qml_library”库。“sstd_qt_and_qml_library”库用于引入和补充 Qt 库。
- 第 44~48 行将 Qml 文件加入工程。这是一种惯用法,用于实现 Qml 国际化。

本书的所有工程项目大同小异,以后不再赘述。

源码 1.3.2

```

1 #firstapplication.pro
2 TEMPLATE = app
3
4 CONFIG(debug,debug|release){
5     TARGET = firstapplication_debug
6 }else{
7     TARGET = firstapplication
8 }
9
10 #define out put dir
11 include($$PWD/../../outdirpath.pri)
12 #define cplusplus environment
13 include($$PWD/../../cplusplus.pri)
14 #import sstd_library
15 include($$PWD/../../sstd_library/import_sstd_library.pri)
16 #import sstd_qt_and_qml_library
17 include($$PWD/../../sstd_qt_and_qml_library/import_sstd_qt_and_qml_library.pri)
18
19 !win32 {
20     QMAKE_LFLAGS += -Wl,-rpath .
21 }
22
23 win32-msvc*{
24     CONFIG += console
25 }
26
27 DEFINES += CURRENT_DEBUG_PATH=\\"$$PWD\\\""
28
29 DESTDIR = $$RootDestDir
30
31 SOURCES += $$PWD/main.cpp
32
33 CONFIG(debug,debug|release){
34     QMAKE_POST_LINK += $$DESTDIR/build_install_debug $$PWD "myqml"
35 }else{
36     QMAKE_POST_LINK += $$DESTDIR/build_install $$PWD "myqml"
37 }
38 export(QMAKE_POST_LINK)
39
40 QMLSOURCE += $$PWD/myqml/firstapplication/main1.qml
41 QMLSOURCE += $$PWD/myqml/firstapplication/main2.qml
42 QMLSOURCE += $$PWD/myqml/firstapplication/main3.qml
43
44 lupdate_only{
45     SOURCES += $$QMLSOURCE
46 }
47
48 DISTFILES += $$QMLSOURCE

```

源码 1.3.2

1.3.2 Qt Quick 运行常用设置

Qt Quick 路径识别依靠 `QUrl` 类。`QUrl` 既可以表达一个网络路径也可以表达一个本地路径。

使用 `QUrl` 表达一个本地绝对路径，Windows 平台下需要在开头加“`file:///`”，POSIX 平台下需要在开头加“`file://`”。

比如，在 Windows 平台下一个本地绝对路径是“`C:/main.qml`”，那么用 `QUrl` 表

达则为“file:///C:/main.qml”;在 Linux 平台下一个本地绝对路径是“/usr/main.qml”,那么用 QUrl 表达则为“file:///usr/main.qml”。

源码 1.3.3 展示如何用 C++ 实现这一功能。

源码 1.3.3

```

6 inline QUrl getLocalFileFullPath(
7     const QString & argFileName,
8     const QString & argBase) {
9     const QDir varRootDir{ argBase };
10    const auto varAns = varRootDir.absoluteFilePath(argFileName);
11    if (varAns.startsWith(QChar('/'))) {
12        return QStringLiteral(R"(file://)") + varAns;
13    } else {
14        return QStringLiteral(R"(file:///") + varAns;
15    }
16 }
```

源码 1.3.3

Qt Quick 依赖于 OpenGL。源码 1.3.4 展示常用 OpenGL 参数设置:

源码 1.3.4

```

18 inline QSurfaceFormat getDefaultQSurfaceFormat() {
19     auto varFormat = QSurfaceFormat::defaultFormat();
20     varFormat.setVersion(4, 6);
21     varFormat.setProfile(QSurfaceFormat::CoreProfile);
22     varFormat.setSamples(4);
23     varFormat.setAlphaBufferSize(8);
24     varFormat.setBlueBufferSize(8);
25     varFormat.setRedBufferSize(8);
26     varFormat.setGreenBufferSize(8);
27     varFormat.setDepthBufferSize(24);
28     varFormat.setSwapBehavior(QSurfaceFormat::DoubleBuffer);
29     varFormat.setRenderableType(QSurfaceFormat::OpenGL);
30     varFormat.setSwapInterval(0)/*关闭垂直同步*/;
31 #if defined(ENABLE_GL_DEBUG)
32     varFormat.setOption(QSurfaceFormat::DebugContext, true);
33 #else
34     varFormat.setOption(QSurfaceFormat::DebugContext, false);
35 #endif
36     return varFormat;
37 }
```

源码 1.3.4

Qt 库依靠事件队列实现消息循环。因而在使用绝大多数 Qt 组件之前都要先构造 QCoreApplication 或其子类。

在构造 QCoreApplication 之前,需要设置一些参数。源码 1.3.5 展示了一些常见设置。

源码 1.3.5

```

39 inline void beforeApplication() {
40     {
41         /*初始化随机种子*/
42         ::srand(static_cast<unsigned>(::time(nullptr)));
43     }
44     {
45         /*高分屏支持*/
46         QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
47     }
48 }
```

```

49     /* 设置默认opengl环境*/
50     QSurfaceFormat::setDefaultFormat(::getDefaultQSurfaceFormat());
51 }
52 }
```

源码 1.3.5

在构造完 `QCoreApplication` 之后,还需要加载一些组件。比如 `QImage` 组件是依靠插件支持的。对于一些外存较差的设备而言,加载插件可能会造成几百毫秒甚至数秒的卡顿。为了程序运行平滑,这些组件都是预加载的。

源码 1.3.6 展示了如何实现这一操作。

源码 1.3.6

```

54 inline void afterApplication() {
55 {
56     QImage varImage{QStringLiteral("test.png")};
57     (void)varImage;
58 }
59 }
```

源码 1.3.6

源码 1.3.7 展示了如何在工程文件中引入常用 Qt 库。

源码 1.3.7

```

1 QT += gui
2 QT += qml
3 QT += core
4 QT += quick
5 QT += widgets
6 QT += concurrent
7 QT += quickwidgets
8 QT += quickcontrols2
```

源码 1.3.7

1.3.3 使用 `QQuickView` 加载 Qt Quick 程序

`QQuickView` 自 Qt 5.0 引入。

`QQuickView` 被设计用来提供一个在显示器上呈现 Qml 渲染结果的一个集成环境。如果读者需要在移动设备上呈现 Qt Quick, `QQuickView` 是一个好的选择。

源码 1.3.8 展示了使用 `QQuickView` 加载 Qml 的 C++ 代码⁵; 源码 1.3.9 展示了一个简单显示一个红色窗口的 Qml 代码。

源码 1.3.8

```

67 beforeApplication();
68 QGuiApplication varApp{ argc,argv };
69 afterApplication();
70 QQuickView varView;
71 varView.setResizeMode(QQuickView::SizeMode::SizeViewToRootObject);
72 #ifdef _DEBUG
73 varView.setSource(
74     getLocalFileFullPath(
75         QStringLiteral("myqml/firstapplication/main1.qml"),
76         CURRENT_DEBUG_PATH));
77 #else
78 varView.setSource(
79     getLocalFileFullPath(
80         QStringLiteral("myqml/firstapplication/main1.qml"),
```

⁵ 将宏 `QML_USE_WINDOW_TYPE` 的值改为 1。

```

81     qApp->applicationDirPath());
82 #endif
83 if (varView.status() != QQuickView::Status::Ready) {
84     qWarning() << QStringLiteral("can not load : main1.qml");
85     return -1;
86 }
87 varView.show();
88 return varApp.exec();

```

源码 1.3.8

源码 1.3.9

```

1 /*main1.qml*/
2 import QtQuick 2.9
3
4 Rectangle{
5
6     width: 512 ;
7     height: 512 ;
8     color: Qt.rgba(1,0,0,1);
9
10 }/*Rectangle*/

```

源码 1.3.9

如源码 1.3.8 的 72~82 行所示，本书在 Debug 模式下从当前项目目录下载入 Qml 文件，而在 Release 模式下从应用程序目录下载入 Qml 文件。

将当前项目目录文件拷贝到应用程序目录是依靠本书自带的一个小工具“build_install”达成的。此工具对于非 qml 文件仅仅是拷贝，而对于 qml 文件做了更多处理。后续章节会有介绍。

1.3.4 使用 QQuickWidget 加载 Qt Quick 程序

QQuickWidget 自 Qt 5.3 引入。

引入 QQuickWidget 的目的就是混用 Qt Quick 和 Qt Widgets。但实际上，使用 QQuickWidget 加载 Qml 和使用 QQuickView 加载 Qml 的根本区别在于：QQuickWidget 是同步渲染的，而 QQuickView 是异步渲染的。

对于一些小型应用程序，异步渲染是不必要的。而更重要的是，在一些设备上，由于硬件限制，异步渲染是低效甚至不可实现的。即使可以实现，也可能有一些莫名其妙的 BUG。

对于以上情形，使用 QQuickWidget 比使用 QQuickView 能够获得更好的效果，甚至能消除一些 BUG。

源码 1.3.10 展示了使用 QQuickWidget 加载 Qml 的 C++ 代码⁶；源码 1.3.11 展示了一个简单显示一个红色窗口的 Qml 代码。

源码 1.3.10

```

90 beforeApplication();
91 QApplication varApp{ argc, argv };
92 afterApplication();
93 QQuickWidget varWidget;
94 varWidget.setResizeMode(QQuickWidget::SizeMode::SizeViewToRootObject);
95 #ifdef _DEBUG
96 varWidget.setSource(getLocalFileFullPath(

```

⁶ 将宏 QML_USE_WINDOW_TYPE 的值改为 2。

```

97     QStringLiteral("myqml/firstapplication/main2.qml"),
98     CURRENT_DEBUG_PATH));
99 #else
100 varWidget.setSource(getLocalFileFullPath(
101     QStringLiteral("myqml/firstapplication/main2.qml"),
102     qApp->applicationDirPath()));
103#endif
104 if (varWidget.status() != QQuickWidget::Status::Ready) {
105     qWarning() << QStringLiteral("can not load : main2.qml");
106     return -1;
107 }
108 varWidget.show();
109 return varApp.exec();

```

源码 1.3.10

源码 1.3.11

```

1 /*main2.qml*/
2 import QtQuick 2.9
3
4 Rectangle{
5
6     width: 512 ;
7     height: 512 ;
8     color: Qt.rgba(1,0,0,1);
9
10 /*begin:debug*/
11     border.width: 9 ;
12 /*end:debug*/
13
14
15 }/*Rectangle*/

```

读者可以分别在 Debug 和 Release 模式下运行此程序，会发现 Debug 模式比 Release 模式多了一个边框。这一切都是靠本书自带的小工具“build_install”实现的。

如源码 1.3.11 的 10~12 行所示。本书自带的小工具“build_install”在拷贝 qml 文件时会将/*begin:debug*/和/*end:debug*/之间的内容替换为注释。

读者采用此小工具可以达到一些调试效果。

1.3.5 使用 QQmlApplicationEngine 加载 Qt Quick 程序

QQmlApplicationEngine 自 Qt 5.1 引入。

引入 QQmlApplicationEngine 的目的是为了能够实现 Qt Quick Controls。

QQmlApplicationEngine 是对 QQuickView 的进一步包装和扩展。Qt Quick Controls 一些控件比如“对话框”需要一些全局支持。QQmlApplicationEngine 就是被设计用来提供这些全局支持的。

如果读者不需要传统对话框的话，QQuickView 或 QQuickWidget 也是足够使用的。但如果使用类似于对话框这样的功能，还是建议读者使用 QQmlApplicationEngine。

源码 1.3.12 展示了使用 QQuickWidget 加载 Qml 的 C++ 代码⁷；源码 1.3.13 展示了一个简单显示一个红色窗口的 Qml 代码。

⁷ 将宏 QML_USE_WINDOW_TYPE 的值改为 3。

源码 1.3.12

```

111 beforeApplication();
112 QGuiApplication varApp{ argc,argv };
113 afterApplication();
114 #ifdef _DEBUG
115 QQmlApplicationEngine varEngine(getLocalFileFullPath(
116     QStringLiteral("myqml/firstapplication/main3.qml"),
117     CURRENT_DEBUG_PATH));
118#else
119 QQmlApplicationEngine varEngine(getLocalFileFullPath(
120     QStringLiteral("myqml/firstapplication/main3.qml"),
121     qApp->applicationDirPath()));
122#endif
123 if (varEngine.rootObjects().isEmpty()) {
124     qWarning() << QStringLiteral("can not load : main3.qml");
125     return -1;
126 }
127 return varApp.exec();

```

源码 1.3.12

源码 1.3.13

```

1 /*main3.qml*/
2 import QtQuick 2.9
3 import QtQuick.Controls 2.5
4
5 ApplicationWindow {
6
7     width: 512 ;
8     height: 512 ;
9     visible: true ;
10
11     Rectangle{
12         anchors.fill: parent ;
13         color: Qt.rgba(1,0,0,1);
14
15         /*begin:debug*/
16         border.width: 9 ;
17         /*end:debug*/
18
19     }/*Rectangle*/
20 }

```

源码 1.3.13

1.4 你好世界！

绝大多数介绍计算机语言的书籍都有一个“Hello World!”的案例，本书也不能免俗。

本章的 C++ 代码如源码 1.4.1：

图 1.4.1

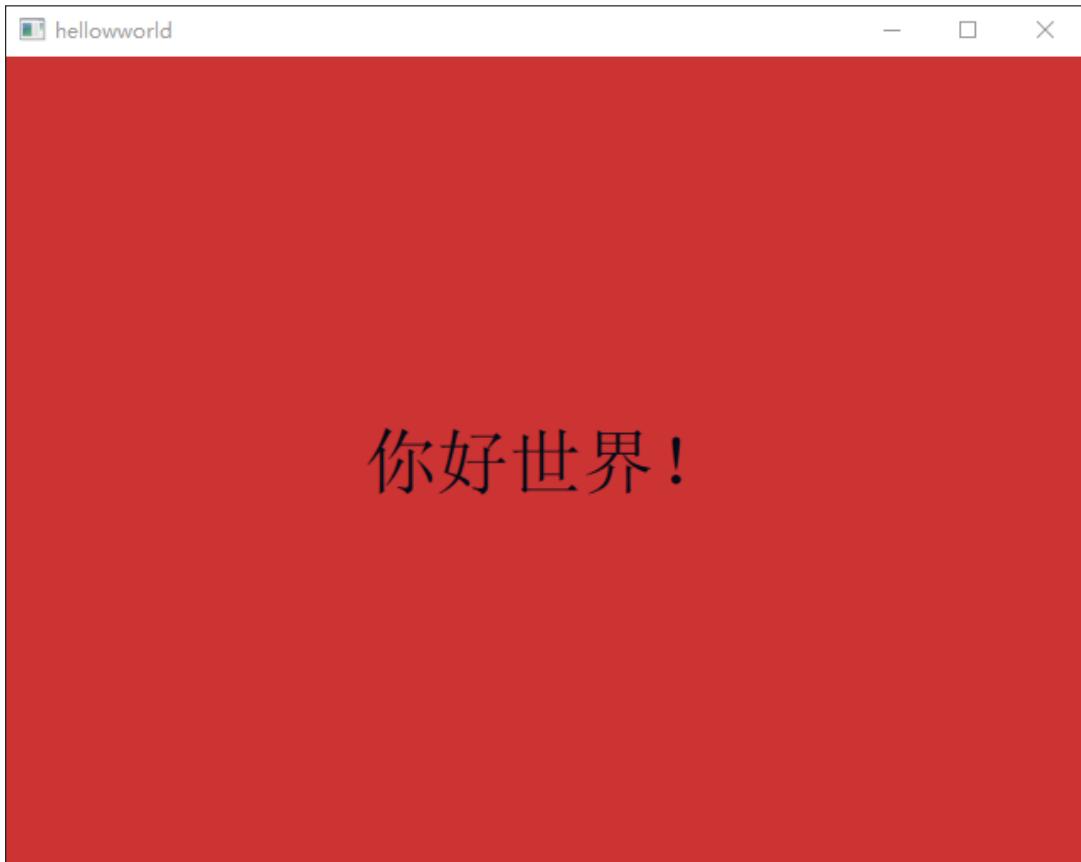


图 1.4.1：你好世界！

源码 1.4.1

```
1 #include <sstd_qt_and_qml_library.hpp>
2
3 int main(int argc, char ** argv) {
4
5     /*初始化程序*/
6     auto varApp = sstd_make_unique< sstd::Application >(argc, argv);
7     /*初始化Qml/Quick引擎*/
8     auto varWindow = sstd_make_unique< sstd::DefaultRoowWindow >();
9
10    /*获得Qml文件绝对路径*/
11    auto varFullFileName = sstd::getLocalFileFullPath(
12        QSL("myqml/helloworld/main.qml"));
13    /*加载Qml文件*/
14    varWindow->load(varFullFileName);
15    /*检查并报错*/
16    if (varWindow->status() != sstd::LoadState::Ready) {
17        qWarning() <<
18            QSL("can not load : ")
19            << varFullFileName;
20        return -1;
21    }
22    varWindow->show();
23
24
25    return varApp->exec();
26
27 }
```

源码 1.4.1

本书将大量的程序细节隐藏到了“sstd_qt_and_qml_library”库里面。

- `sstd::Application` 用于构造 `QApplication`, 并初始化 Qt Quick 运行所需的参数;
- `sstd::DefaultRoowWindow` 在 Debug 模式下继承自 `QQuickWidget`, 在 Release 模式下继承自 `QQuickView`;
- `sstd::getLocalFileFullPath` 在 Debug 模式以当前文件目录作为根目录, 在 Release 模式下以应用程序目录作为根目录;

本书以后章节的“`main.cpp`”都大同小异, 以后不再赘述。

“`main.qml`”如源码 1.4.2 所示:

源码 1.4.2

```

1 /*main.qml*/
2 import QtQuick 2.9
3 import "main_private" as MainPrivate
4
5 Rectangle {
6
7     width: 640
8     height: 480
9     color: Qt.rgba(0.8, 0.8, 0.8, 1)
10
11     MainPrivate.MainText {
12         z: 1
13         anchors.fill: parent
14     } /*~MainText*/
15
16     MainPrivate.MainRectangle {
17         z: 0
18         anchors.fill: parent
19     } /*~MainRectangle*/
20 } /*~Rectangle*/

```

源码 1.4.2

- 第 3 行展示了如何引入其他目录的 Qml 文件。其语法如源码 1.4.3 所示:

源码 1.4.3

```
1 import "ResourceURL" as Qualifier
```

源码 1.4.3

- 第 5 行定义了一个 `Rectangle`:

- 第 7 行定义了 `Rectangle` 的宽度;
- 第 8 行定义了 `Rectangle` 的高度;
- 第 9 行定义了 `Rectangle` 的颜色;

- 第 11~19 行定义了两个子对象。和 Qt Widgets 一样, 子对象在父对象之上。兄弟对象之间的关系是, 后出现的对象在先出现的对象之上。也可以调整 `z` 属性调整兄弟对象的上下关系。

读者可以尝试注释掉第 12 行和 17 行观察程序输出结果。

- 第 13 行和第 18 行使用“`anchors`”确保子对象完全覆盖父对象。

“`MainRectangle.qml`”如源码 1.4.4 所示:

源码 1.4.4

```

1 /*main_private/MainRectangle.qml*/
2 import QtQuick 2.9
3

```

源码 1.4.4

```

4 Rectangle {
5     color: Qt.rgba(0.8, 0.2, 0.2, 1)
6 }
```

Qml 是一门大小写敏感的计算机语言。读者使用 import 命令引入 Qml 定义的对象时，文件名必须以大写开头；而引入 JavaScript 文件时，文件名应当以小写开头。

“MainText.qml”如源码 1.4.5 所示：

源码 1.4.5

```

1 /*main_private/MainText.qml*/
2 import QtQuick 2.9
3
4 Text {
5     text: qsTr("你好世界！")
6     color: Qt.rgba(Math.random() / 10, Math.random() / 10,
7                     Math.random() / 10, 1)
8     font.pointSize: 32
9     verticalAlignment: Text.AlignVCenter
10    horizontalAlignment: Text.AlignHCenter
11 }
```

- 第 5 行展示了如何在 Qml 中实现国际化，读者只需要用 qsTr 包装字符串即可；
- 第 6~7 行展示了可以在 Qml 中直接使用 JavaScript；

1.5 初识 Qt Quick 控件

图 1.5.1

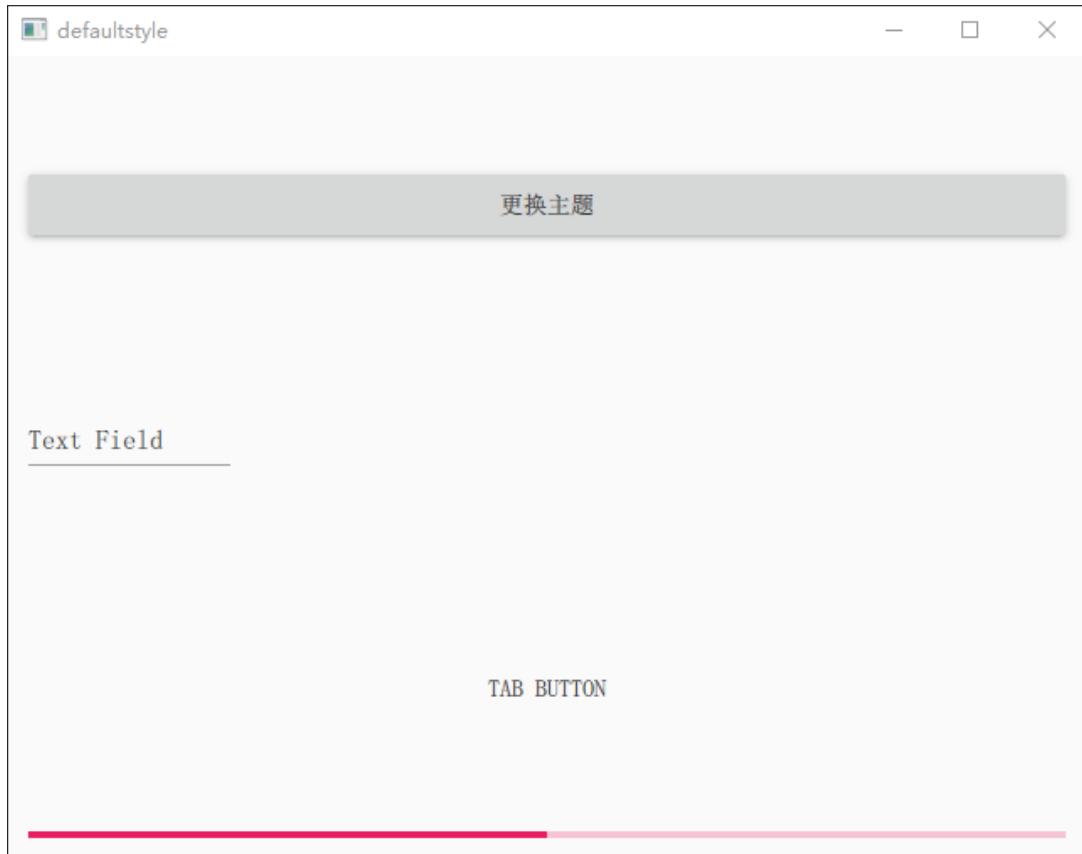


图 1.5.1：Qt Quick 控件及样式！

Qt Quick Controls 2 自 Qt 5.7 引入。

本书不加特别说明,提到 Qt Quick Controls 就是指 Qt Quick Controls 2。

Qt Quick Controls 1 更多的是沿用传统桌面的设计风格;而 Qt Quick Controls 2 更加现代化并更适用于移动设备。并且,Qt Quick Controls 2 对于主题和样式提供了专门的语法支持。

基于这些语法,读者可以轻松的实现样式、内容和结构分离。即使读者不想在样式上太花费心思,Qt Quick Controls 2 也默认提供了数个艺术级的样式模板。

除了维护老项目,没有什么理由不采用 Qt Quick Controls 2。

本项目的“main.qml”如源码 1.5.1 所示:

源码 1.5.1

```
1 /*defaultstyle/main.qml*/
2 import QtQuick 2.9
3 import QtQuick.Controls 2.3
4 import QtQuick.Layouts 1.3
5 import QtQuick.Controls.Material 2.12
6
7 Pane {
8
9     id : idRoot
10    width: 640;
11    height: 480;
12
13    function changeTheme(){
14        if(idRoot.Material.theme === Material.Dark ){
15            idRoot.Material.theme = Material.Light;
16        }else{
17            idRoot.Material.theme = Material.Dark;
18        }
19    }
20
21    ColumnLayout {
22        id: idColumn
23        anchors.fill: parent
24
25        Button {
26            id: idButton
27            text: qsTr("更换主题")
28            Layout.fillWidth : true
29            onClicked: {
30                idRoot.changeTheme();
31            }
32        }
33
34        TextField {
35            id: idTextField
36            text: qsTr("Text Field")
37            Layout.fillWidth : true
38        }
39
40        TabButton {
41            id: idTabButton
42            text: qsTr("Tab Button")
43            Layout.fillWidth : true
44        }
45
46        ProgressBar {
47            id: idProgressBar
```

```

48         value: 0.5
49         Layout.fillWidth : true
50     }
51 }
52 }
53 }/*~Pane*/

```

源码 1.5.1

- 第 21 行展示了如何使用 Layout;
- 第 28 行、第 37 行、第 43 行、第 49 行展示了使用关联属性⁸;
- 第 29~31 行展示了如何关联一个 Qml 信号到一个 JavaScript 函数;
- 第 13~19 行展示了如何在一个 Qml 对象里面使用 JavaScript 定义一个槽函数;

本案例演示了如何使用 Qt Quick Control 自带的“Material”样式。

要使用 Qt Quick Control 自带的样式，需要在 QApplication 构造之前调用如源码 1.5.2 所示的 C++ 代码，以载入配置文件。

源码 1.5.2

```
1 ::qputenv("QT_QUICK_CONTROLS_CONF","defaultstyle_qtquickcontrols2.conf");
```

配置文件内容如源码 1.5.3 所示：

源码 1.5.3

```

1 [Controls]
2 Style=Material
3 FallbackStyle=Material
4
5 [Material]
6 Theme=Dark

```

源码 1.5.3

Qt Quick Control 的样式具有继承性。

子控件的样式与父控件一致，只需要更改父控件的样式，则子控件的样式跟随父控件变化。

1.6 在 Qt Quick 中使用着色器

Qt Quick 本身是使用 OpenGL 达成渲染的，Qt Quick 原生支持 GLSL。

不过，考虑到硬件兼容性。目前在 Qml 中使用 GLSL，只支持顶点着色器和片段着色器。

如果读者需要使用计算着色器、几何着色器或分型着色器，读者需要使用 C++ 扩展 Qml。

如源码 1.6.1 第 14~46 行展示了如何使用“ShaderEffect”，在 Qt Quick 中使用 GLSL。

⁸ Attached Properties。

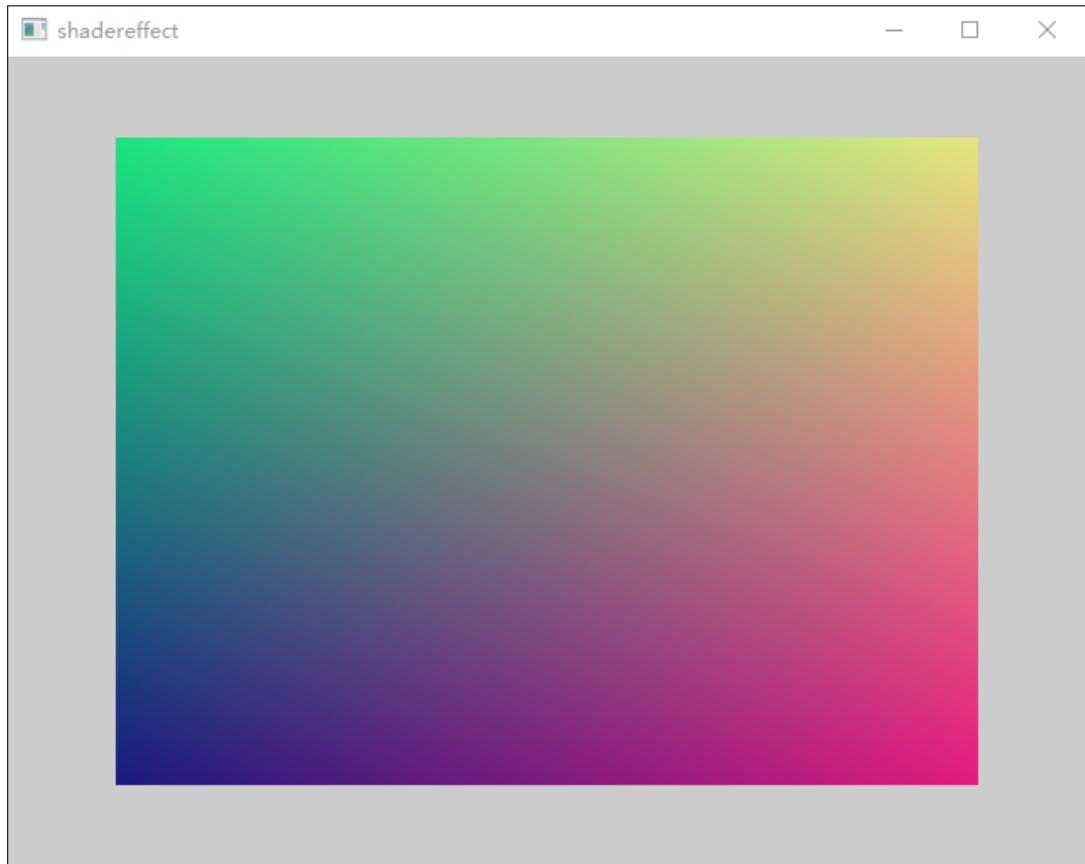


图 1.6.1

图 1.6.1 : Qt Quick 中使用着色器

源码 1.6.1

```
1 /*shadereffect/main.qml*/
2 import QtQuick 2.9
3
4 Rectangle {
5
6     width: 640;
7     height: 480;
8     color: Qt.rgba(0.8,0.8,0.8,1);
9
10    Rectangle{
11        anchors.centerIn: parent      ;
12        width: parent.width * 0.8   ;
13        height: parent.height * 0.8 ;
14        ShaderEffect{
15            anchors.fill: parent ;
16            fragmentShader:"
17 /*片段着色器*/
18 #version 460
19
20 in vec2 qt_TexCoord0/*纹理坐标*/ ;
21 out vec4 fragColor /*输出值*/ ;
22
23 uniform float qt_Opacity/*透明度*/ ;
24
25 void main() {
26     vec4 varColor = vec4( qt_TexCoord0.x , qt_TexCoord0.y , 0.5 , 1);
27     fragColor = varColor * qt_Opacity;
28 }
29
30 "
```

```

31     vertexShader :"
32 /*顶点着色器*/
33 #version 460
34
35 in vec4 qt_Vertex/*输入点坐标*/ ;
36 out vec2 qt_TexCoord0/*纹理坐标*/ ;
37
38 uniform mat4 qt_Matrix/*投影矩阵*/ ;
39
40 void main() {
41     gl_Position = qt_Matrix * qt_Vertex;
42     qt_TexCoord0 = gl_Position.xy*0.5 + 0.5 ;
43 }
44
45 "
46     }
47 }
48
49 }/*~Rectangle*/

```

源码 1.6.1

使用“ShaderEffect”导入纹理是极为简单的。读者只需要在“ShaderEffect”自定义一个代表“Image”的属性，在 GLSL 中就可以直接使用了。

如源码 1.6.2 所示：

- 第 14~18 行定义了一个 Image；
- 第 20 行定在“ShaderEffect”中自定义了一个名为的“source”属性，此属性指向 Image 对象；
- 第 30 行在片段着色器中将“source”属性作为一个纹理载入；

图 1.6.2

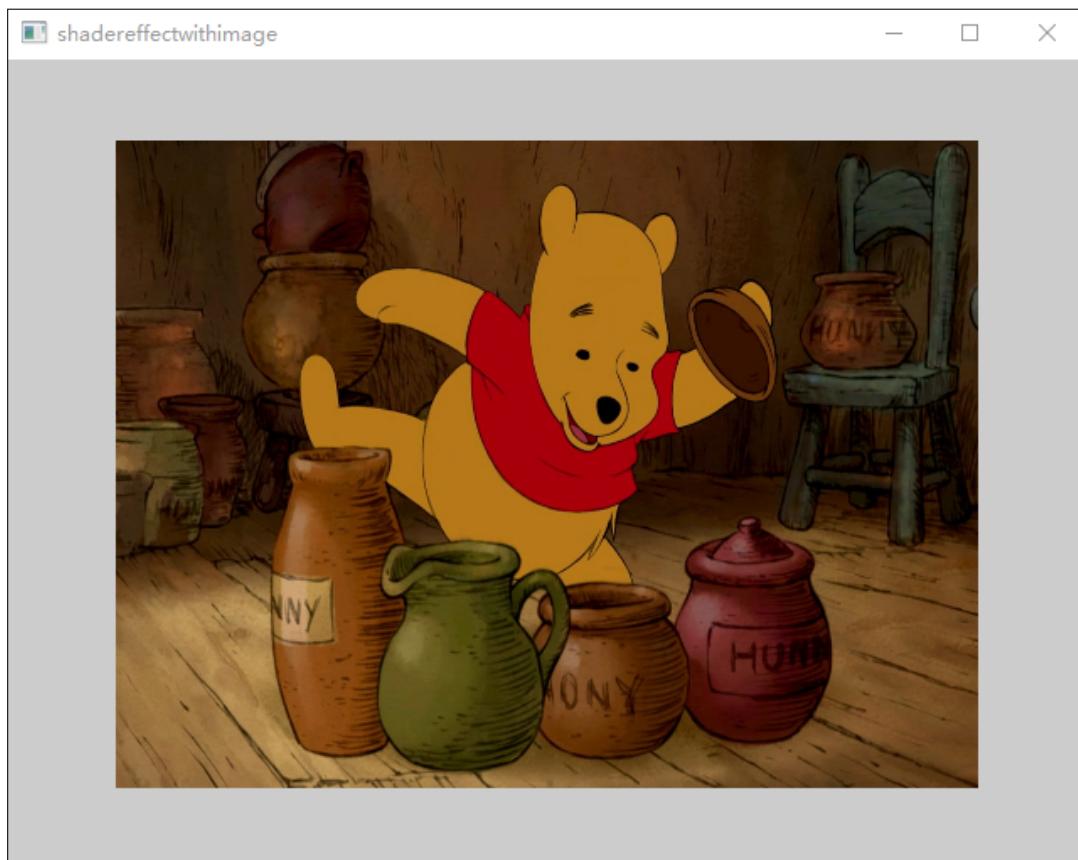


图 1.6.2：Qt Quick 着色器中使用纹理

源码 1.6.2

```
1 /*shadereffectwithimage/main.qml*/
2 import QtQuick 2.9
3
4 Rectangle {
5
6     width: 640;
7     height: 480;
8     color: Qt.rgba(0.8,0.8,0.8,1);
9
10    Rectangle{
11        anchors.centerIn: parent      ;
12        width: parent.width * 0.8   ;
13        height: parent.height * 0.8 ;
14        Image{
15            id : idSourceImage;
16            source: "0000.jpg";
17            visible: false      ;
18        }
19        ShaderEffect{
20            property variant source: idSourceImage/*the image...*/
21            anchors.fill: parent ;
22            fragmentShader:"
23 /*片段着色器*/
24 #version 460
25
26 in vec2 qt_TexCoord0;
27
28 out vec4 fragColor;
29
30 uniform sampler2D source/*the image...*/;
31 uniform float qt_Opacity;
32
33 void main() {
34     fragColor = texture(source, qt_TexCoord0) * qt_Opacity;
35 }
36
37 "
38         vertexShader :"
39 /*顶点着色器*/
40 #version 460
41
42 in vec4 qt_Vertex;
43 in vec2 qt_MultiTexCoord0;
44
45 out vec2 qt_TexCoord0;
46
47 uniform mat4 qt_Matrix;
48
49 void main() {
50     qt_TexCoord0 = qt_MultiTexCoord0;
51     gl_Position = qt_Matrix * qt_Vertex;
52 }
53
54 "
55     }
56 }
57
58 }/*~Rectangle*/
```

1.7 使用 C++ 扩展 Qt Quick

读者可能认为使用 Qt Quick 只需要 Qml 足以,但不久读者就会失望。即使退一步,希望 Qt Quick 可以满足绝大多数需求,这也是难以达成。

Qt Quick 不是一种全面代替 Qt C++ 无所不包的解决方案。Qt Quick 只是导出 Qt C++ 的一套接口规范。

当读者面对一个具体的问题,在 Qt Quick 中无法找到现成的组件或者无法通过简单修改现有 Qt Quick 组件达成目的时。使用 Qt C++ 自定义组件就势在必行。

- 如果所需的组件不需要几何逻辑,比如实现一个本地文件监视器,那么只继承自 QObject 即可;
- 如果所需的组件需要几何逻辑但无需渲染,比如实现一个鼠标监视器,那么只需要继承自 QQuickItem;
- 如果所需的组件需要渲染,那么需要继承自 QQuickItem,并在构造函数中设置 QQuickItem::ItemHasContents 标志位;
- 如果希望使用 QPainter 实现渲染,那么需要继承自 QQuickPaintedItem 是一个好的选择;
- 如果仅需要一个简易的 OpenGL 离屏渲染环境,那么继承自 QQuickFramebufferObject 是一个好的选择;

如图 1.7.1 ,列出了 C++ 导出 Qt Quick 基类继承关系图。

图 1.7.1

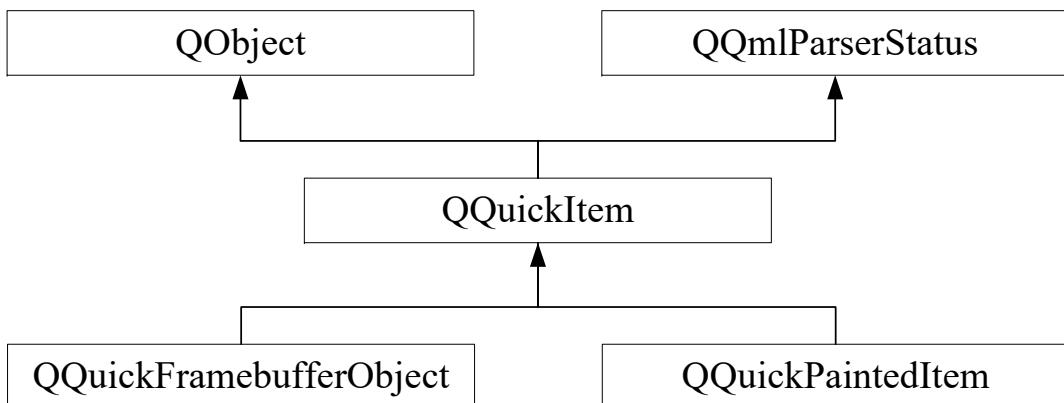


图 1.7.1 : C++ 导出 Qt Quick 所需基类

本节展示直接使用 C++ 调用 OpenGL 绘制,并将其导出到 Qt Quick。

本节示例位于目录“QtQmlBook/chapter01/directdrawbyopengl”。

先来看看 Qt Quick 代码,如源码 1.7.1 。

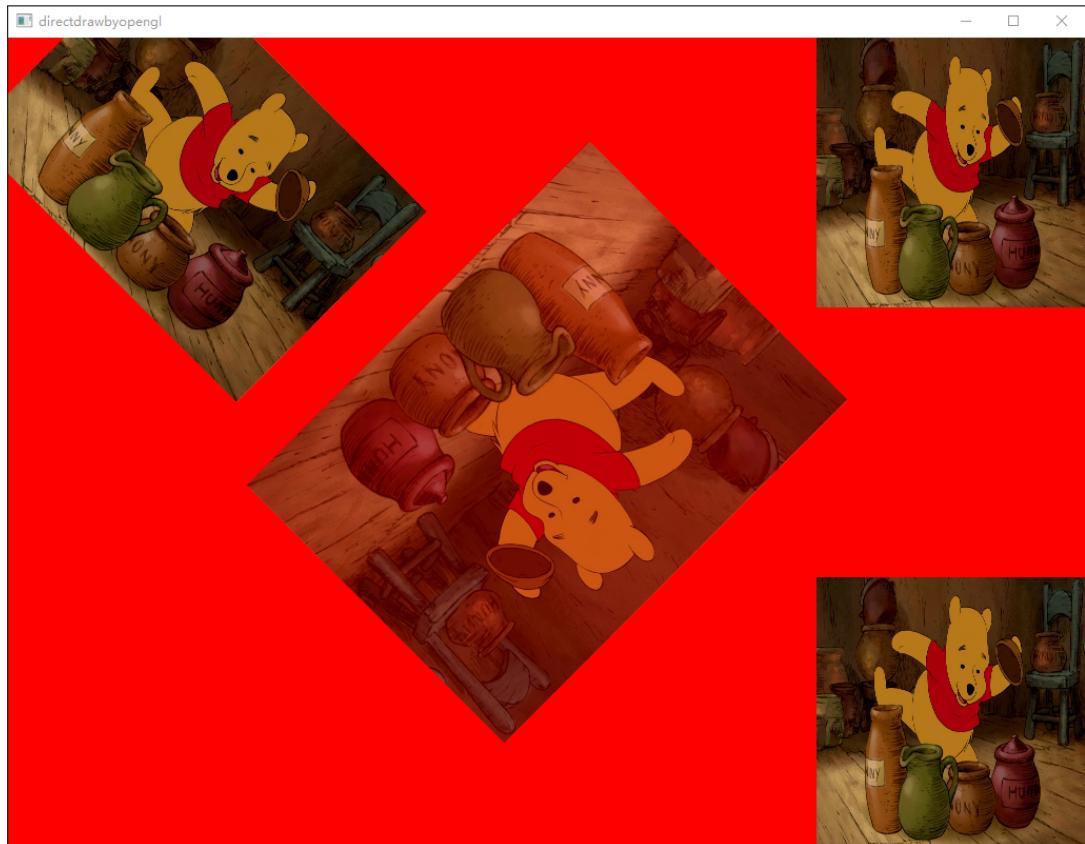


图 1.7.2

图 1.7.2：直接使用 OpenGL 绘制

源码 1.7.1

```
1 /*directdrawbyopengl/main.qml*/
2 import QtQuick 2.9
3 import sstd.quick 1.0
4
5 Rectangle{
6     id : root_object
7     objectName: "root_object";
8     width: 1024 ;
9     height: 768 ;
10    color: Qt.rgba(1,0,0,1);
11
12    DrawImageItemRaw {
13
14        width: parent.width /2 ;
15        height: parent.height /2;
16        anchors.centerIn: parent;
17        transformOrigin: Item.Center ;
18        rawImage : Qt.resolvedUrl( "0000.jpg" );
19
20        Timer{
21            repeat: true;
22            running: true;
23            interval : 500 ;
24            triggeredOnStart: true;
25            onTriggered: {
26                parent.rotation += 15 ;
27                if(parent.rotation>360){
28                    parent.rotation -= 360 ;
29                }
30                parent.opacity = Math.random() * 0.3 + 0.7 ;
31                parent.scale = Math.random() * 0.3 + 0.7 ;
32            }
33        }
34    }
35}
```

```

32         }
33     }
34   }
35 }
36
37
38 Component.onCompleted : {
39     Qt.createQmlObject(
40 "
41 import QtQuick 2.9
42 import sstd.quick 1.0
43
44 DrawImageItemRaw {
45     width: 256 ;
46     height: 256 ;
47     anchors.top: parent.top ;
48     anchors.right : parent.right ;
49     rawImage : Qt.resolvedUrl( '0000.jpg' ) ;
50 }
51
52 " , root_object ) ;
53 }
54
55 }/*Rectangle*/

```

源码 1.7.1

- 第 3 行引入自定义模块。其对应的 C++ 代码如源码 1.7.2：

源码 1.7.2

```

1 static inline void register_this() {
2     qmlRegisterType<DrawImageItem>(
3         "sstd.quick",
4         1, 0,
5         "DrawImageItemRaw");
6 }
7 Q_COREAPP_STARTUP_FUNCTION(register_this)

```

源码 1.7.2

本节的示例依然是静态加载，实际上 Qt Quick 也支持以插件的形式动态加载组件。

- 第 12~35 行演示了如何使用自定义模块中的类“DrawImageItemRaw”。
- 使用 C++ 自定义的类与 Qt Quick 原生元素没有什么不同。信号槽、平移、旋转、缩放……
- 不难发现，使用 Qt Quick 可以将一切从极高的抽象层上迅速的组装起来。
- 第 38~53 行演示了直接在 Qt Quick 里面编译 Qt Quick 源码并以此创建对象。正如读者所见，Qt Quick 拥有脚本语言的所有特性，并可以与 Qt C++ 无缝通信。

源码 1.7.3

```

1 /*directdrawbyopengl/main.cpp*/
2 #include <sstd_qt_and_qml_library.hpp>
3 #include "DrawImageItem.hpp"
4
5 int main(int argc, char ** argv) {
6
7     /*初始化程序*/
8     auto varApp = sstd_make_unique< sstd::Application >(argc, argv);

```

```
9  /*初始化Qml/Quick引擎*/
10 auto varWindow = std::make_unique< std::DefaultRoowWindow >();
11 {
12     /*获得Qml文件绝对路径*/
13     auto varFullFileName = std::getLocalFileFullPath(
14         QStringLiteral("myqml/directdrawbyopengl/main.qml"));
15
16     /*加载Qml文件*/
17     varWindow->load(varFullFileName);
18     /*检查并报错*/
19     if (varWindow->status() != std::LoadState::Ready) {
20         qWarning() << QStringLiteral("can not load : ")
21             << varFullFileName;
22         return -1;
23     }
24
25 }
26
27 varWindow->show();
28
29 {
30     /*运行时由C++端添加对象*/
31     auto varRootObject = varWindow->getRootObject();
32     assert(varRootObject);
33     assert(varRootObject->objectName() == QStringLiteral("root_object"));
34     auto varItem = std::new<DrawImageItem>();
35     varItem->setParent(varRootObject);
36     const auto varImage = QImage(std::getLocalFileFullPath(
37         QStringLiteral("myqml/directdrawbyopengl/0000.jpg")));
38     varItem->setImage(varImage);
39     varItem->setWidth(360);
40     varItem->setHeight(256);
41     varItem->setTransformOrigin(QQuickItem::Center);
42     varItem->setRotation(45);
43     varItem->setParentItem(varRootObject);
44 }
45
46 {
47     /*运行时由C++编译QML对象*/
48     const auto varQmlCode = u8R"+++
49
50 import QtQuick 2.9
51 import std.quick 1.0
52
53 DrawImageItemRaw {
54
55     width: 256
56     height: 256
57     rawImage : Qt.resolvedUrl( "0000.jpg" );
58     anchors.bottom: parent.bottom
59     anchors.right : parent.right
60
61 }
62
63 )+++sv;
64
65     QQmlComponent varComponent{ varWindow->getEngine() };
66     auto varContex = QQmlEngine::contextForObject( varWindow->getRootObject() );
67     varComponent.setData(
68         QByteArray( varQmlCode.data(), static_cast<int>(varQmlCode.size()) ),
```

```
69         varContex->baseUrl()
70     );
71     auto varObject = sstd_runtime_cast<DrawImageItem>(
72         varComponent.beginCreate( varContex ) );
73     assert(varObject);
74     varObject->setParent(varWindow->getRootObject());
75     varObject->setParentItem(varWindow->getRootObject());
76     varComponent.completeCreate();
77
78 }
79
80 return varApp->exec();
81
82 }
```

源码 1.7.3

第 2 章 Qt Quick 基础

2.1 QML 语法

2.1.1 文件编码

自从 Qt 5 开始,Qt 所有源代码只接受一种编码,那就是 UTF-8。

无论是 C++、QML、JavaScript 或者是 qmake 工程文件,都应当只采用 UTF-8 编码。

特别的是,有些 UTF-8 文件开头会有三个特殊字符:

0xEF0xBB0xBF

以上三个特殊字符开头的 UTF-8 文件一般被称为 UTF-8 with BOM。

Qt 集成开发环境所有编译器都识别 BOM,除了 qmake。

也就是读者在书写 qmake 工程文件 (*.pro 或者 *.pri 文件) 时一定不能给文件添加 BOM。实际上,qmake 只识别 ASCII 码的 127 个字符。不应当在 qmake 工程文件中使用超出 ASCII 码的字符。

但是在书写 C++、QML、JavaScript 文件时,可以给文件添加 BOM。

在不同的操作系统下,系统的默认文件编码会有所不同。由于历史原因和编程习惯,一些计算机语言不要求在源码中指定源码的编码。这就会造成,如果不给这些计算机语言的源码添加 BOM,编译器会错误的识别源代码编码,造成编译错误和运行逻辑错误。

对于这类计算机语言,最好对它们的源码添加 BOM。

而对于一些现代计算机语言比如 Html,一般的在源码中就指定了源码的编码。因而,对于这些计算机语言,最好不要对它们的源码添加 BOM。

本书对 C++ 源文件,QML 源文件,以及 JavaScript 源文件添加 BOM 以最大程度降低平台差异性。

2.1.2 注释与帮助

C++、QML 以及 JavaScript 的注释语法都是一致的:

- //单行注释
- /*多行注释*/

而 qmake 的工程文件只支持单行注释:

- # 单行注释

Qt 集成开发环境自带一个工具可以自动提取 C++ 与 QML 中的注释并生成帮助文件。

此工具被称为 QDoc。

QDoc 是一套庞大的体系, 读者可以在 QtCreator 中输入 qdoc 以获得更多帮助。如图 2.1.1。

图 2.1.1

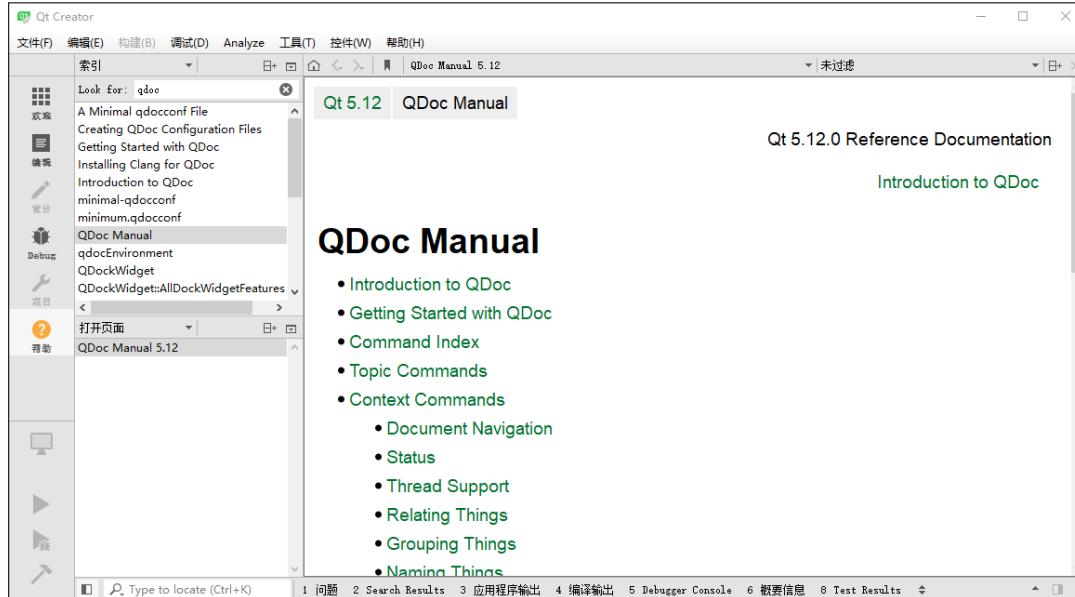


图 2.1.1 : QDoc

2.1.3 属性

2.1.3.1 基本类型

QML 被设计为一门可以支持 JavaScript 的弱类型语言。

虽然绝大多数 JavaScript 代码可以在 QML 环境之中正常运行。但并不意味着 JavaScript 语义与 QML 语义完全一致。

一般而言, QML 对 JavaScript 做了适当扩展。

如果读者完全按照 JavaScript 语义理解 QML 类型系统, 倒也不算错。但损失大量效率是免不了的。

表 2.1.1 展示了 QML 中的基本类型。

表 2.1.1

类型	说明
bool	布尔类型
int	整型
double	浮点型
real	浮点型
enumeration	枚举型
string	字符型
url	统一资源定位符
var	$\forall \{ \text{任意类型} \}$

类型	说明
表 2.1.1 : QML 基本类型	

表 2.1.1 中的基本类型是直接内嵌在 QML 引擎中的。Qt 并没有提供接口可以让读者直接定义像 int, double 这样的原生类型。读者如果需要扩展 QML 类型, 只能继承自 QObject 类或其子类。

- **bool**

QML 中的布尔类型根绝大多数计算机语言一致, 只有 true 和 false 两个值。

- **int**

QML 中的整型对应于 C++ 中的 int, 其安全使用范围是 -2000000000~2000000000。
很多时候应用程序需要的是 int64, 这时候应当使用 var。

- **double 与 real**

QML 中的 double 与 real 没有什么区别, 对应于 IEEE 754 标准中规定的 64 位双精度浮点数。

- **enumeration**

QML 中的枚举类型, 既可来源于 C++ 的导出, 也可来源于 QML 中的定义。

- **url**

QML 中的路径都是 url 类型, 它与 QUrl 一致。

- **string**

QML 中的字符串除了可以使用 JavaScript 中的所有方法之外, 还可以使用 QString 中的 arg 函数。

- **list**

QML 中 list 被设计用于包装 QML 对象, 如果需要基本类型容器, 应当使用 var。

- **var**

QML 中的 var 用于代表一切合法类型, 包括信号槽, 容器, 基本类型以及一切可以被 QVariant 识别的类型……

在一些时候读者会发现 variant 这个词, 它与 var 是一致的。支持这个词仅仅是为了兼容老版本的 QML。

第 3 章 从 C++ 扩展 Qt Quick

第 4 章 状态机及动画

第 5 章 粒子系统

Qt Quick 自带一个艺术级的粒子系统模块。读者可以用它实现一些绚丽的效果,比如:模拟烟雾、模拟流星、模拟火焰……

不过读者如果需要工业级或者类似的要求十分严苛的粒子仿真系统,Qt Quick 自带的粒子系统显然不是一个好的选择。

第 6 章 特效

由于 Qt Quick 的所有渲染最终都依靠现代 3D 可编程渲染管线实现。这样读者就可以在原有管线的基础上任意添加新的渲染管线。

毫无疑问，读者可以不受限的将所有现代渲染技术应用到应用程序之中。

对于常见图形特效，Qt Quick 提供了“Qt Graphical Effects”模块。

本章将带领读者纵览 Qt Graphical Effects 提供的 25 种图形特效。

6.1 导引



图 6.1.1

图 6.1.1：文字阴影

要使用 Qt Graphical Effects 只需要在 Qml 文件开头增加一行：

```
import QtGraphicalEffects 1.12
```

源码 6.1.1 展示了使用“DropShadow”和“InnerShadow”模拟光照，以达到模拟立体文字的效果。

值得注意的是, 图形特效不应当是其源对象(source)的子对象。否则, 上一次图形特效渲染出的结果会加入下一次运算。这会造成渲染出现闭环。在绝大多数情况下这种闭环会导致错误的渲染结果。

源码 6.1.1

```

1 /*firsteffect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6
7     id : idRoot
8     width: 640;
9     height: 480;
10    color: Qt.rgba(0.8,0.8,0.8,1);
11
12    RenderText{
13        id : idText
14        text: "Good!"
15        font.pointSize: 128
16        font.bold: true
17        anchors.centerIn: parent
18        color: Qt.darker( idInnerShadow.color ) ;
19    }
20
21    DropShadow{
22        anchors.fill: idText
23        horizontalOffset: 3
24        verticalOffset: 3
25        radius: 8.0
26        samples: 17
27        color: Qt.rgba(0.1,0.1,0.1,1)
28        source: idText
29    }
30
31    InnerShadow{
32        id : idInnerShadow
33        anchors.fill: idText
34        radius: 8.0
35        samples: 16
36        horizontalOffset: 5.5
37        verticalOffset: -2.8
38        color: Qt.rgba(1,0.9,0.8,1)
39        source: idText
40    }
41
42 }/*~Rectangle*/

```

源码 6.1.1

Qt Graphical Effects 共提供了 25 种图形特效, 如表 6.1.1 所示:

表 6.1.1

类名	分类	简介
Blend	Blend	混合
BrightnessContrast	Color	亮度对比度
ColorOverlay	Color	颜色叠加
Colorize	Color	着色
Desaturate	Color	饱和度

类名	分类	简介
GammaAdjust	Color	伽玛调整
HueSaturation	Color	色相饱和度
LevelAdjust	Color	色阶
ConicalGradient	Gradient	锥形渐变
LinearGradient	Gradient	线性渐变
RadialGradient	Gradient	辐射渐变
Displace	Distortion	变形
DropShadow	Drop Shadow	外阴影
InnerShadow	Drop Shadow	内阴影
FastBlur	Blur	快速模糊
GaussianBlur	Blur	高斯模糊
MaskedBlur	Blur	遮罩模糊
RecursiveBlur	Blur	递归模糊
DirectionalBlur	Motion Blur	方向模糊
RadialBlur	Motion Blur	径向模糊
ZoomBlur	Motion Blur	缩放模糊
Glow	Glow	发光
RectangularGlow	Glow	矩形发光
OpacityMask	Mask	不透明遮罩
ThresholdMask	Mask	阈值遮罩

表 6.1.1 : Qt Graphical Effects 特效概述

6.2 Blend

Blend 特效常用属性如表 6.2.1 :

属性	类型	默认值	范围	简介
cached	bool	false	[false, true]	缓存
mode	string	normal	见表 6.2.2	混合模式
source	variant	\exists	$\forall \{ \text{可渲染对象} \}$	混合源(基色)
foregroundSource	variant	\exists	$\forall \{ \text{可渲染对象} \}$	前混合源(混合色)

表 6.2.1

表 6.2.1 : Blend 特效属性

混合模式如表 6.2.2¹ :

模式	描述 [公式]
normal	结果是 foregroundSource 的 rgb, alpha 取二者中 alpha 的较大者
addition	结果是 source 与 foregroundSource 的和
average	结果是 source 与 foregroundSource 的均值

表 6.2.2

¹ a 代表 foregroundSource, b 代表 source, 0 代表黑色, 1 代表白色, v 代表最终结果。

模式	描述 [公式]
saturation	结果是 source 的亮度值与色调, foregroundSource 的饱和度
hue	结果是 source 的亮度值与饱和度, foregroundSource 的色调
color	结果是 source 的亮度值, foregroundSource 的色调与饱和度
lightness	结果是 foregroundSource 的亮度值, source 的色调与饱和度
colorBurn	颜色加深 $[v = 1 - (1 - b)/a]$
colorDodge	颜色减淡 $[v = b/(1 - a)]$
darken	变暗 $[v = \min(a, b)]$
lighten	变亮 $[v = \max(a, b)]$
darkerColor	深色 $v = \begin{cases} a, & a_r + a_g + a_b < b_r + b_g + b_b \\ b, & a_r + a_g + a_b > b_r + b_g + b_b \end{cases}$
lighterColor	浅色 $v = \begin{cases} a, & a_r + a_g + a_b > b_r + b_g + b_b \\ b, & a_r + a_g + a_b < b_r + b_g + b_b \end{cases}$
difference	变暗 $[v = b - a]$
divide	划分 $[v = b/a]$
multiply	正片叠底 $[v = b \times a]$
negation	否定 $[v = 1 / b - a]$
exclusion	排除 $[v = a + b - (a \times b) / 2]$
hardLight	强光 $v = \begin{cases} 2ab, & a < 0.5 \\ 1 - 2(1 - a)(1 - b), & a \geq 0.5 \end{cases}$
softLight	柔光 $v = \begin{cases} 2ab + a^2(1 - 2b), & a < 0.5 \\ 2a(1 - b) + \sqrt{a}(2a - 1), & a \geq 0.5 \end{cases}$
screen	滤色 $[v = 1 - (1 - a)(1 - b)]$
subtract	减去 $[v = b - a]$

表 6.2.2 : 类型属性

如源码 6.2.1 展示了 Blend 的常见用法。

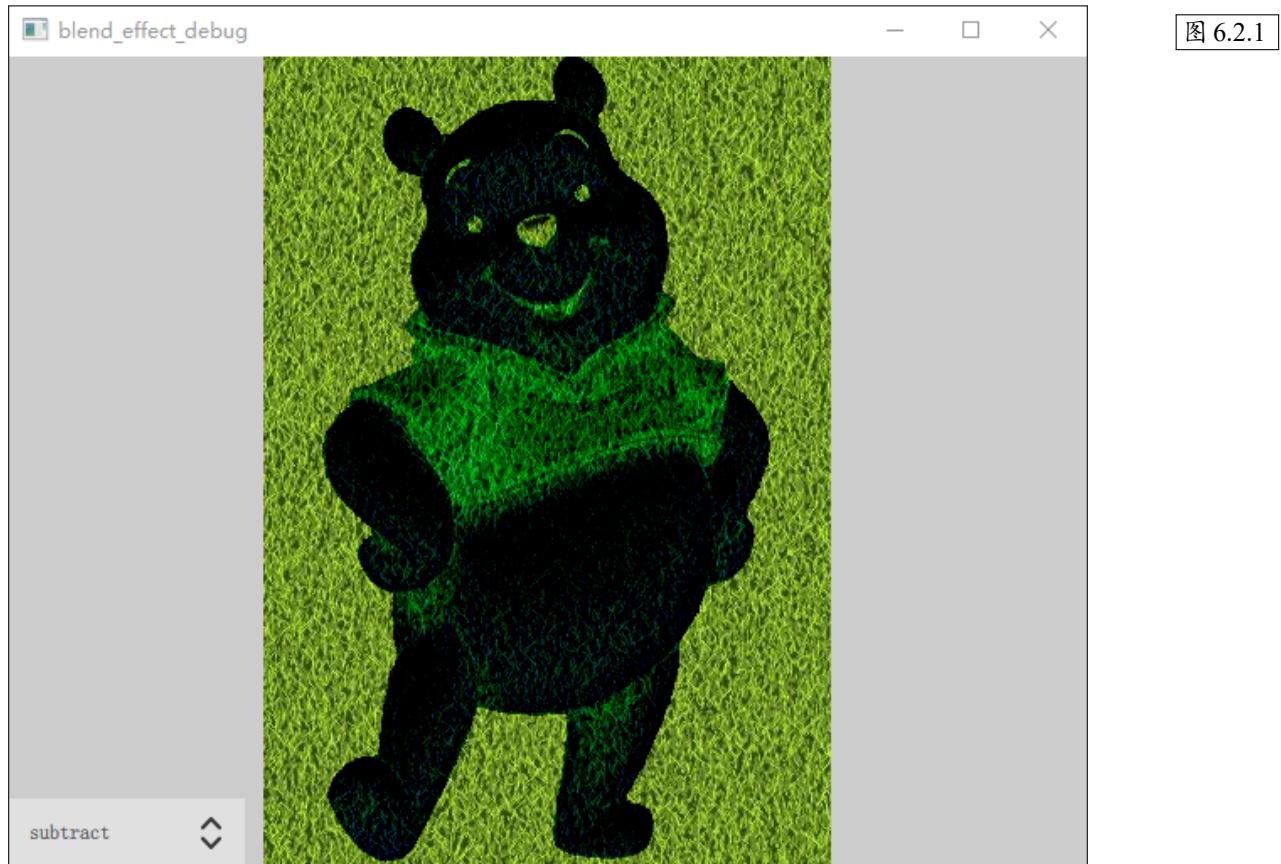


图 6.2.1 : Blend

源码 6.2.1

```
1 /*blend_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11    Image{
12        anchors.fill: idBear;
13        source: "grass.jpg"
14        fillMode: Image.Tile
15        id : idGrass
16        visible: false
17    }
18
19    Image{
20        anchors.centerIn: parent;
21        source: "bear.png"
22        fillMode: Image.Stretch
23        id : idBear
24        visible: false
25    }
26
27    Blend{
28        source: idGrass
29        foregroundSource: idBear
30        mode: idBlendControl.blendModeComboBox.currentText
31    }
32}
```

图 6.2.1

```

31     anchors.centerIn: parent;
32     width: idBear.width
33     height: idBear.height
34 }
35
36 BlendControl {
37     id : idBlendControl
38 }
39
40 }
```

源码 6.2.1

对于一些跳读本书的读者可能会对 Qt Quick 如何实现 Blend 特效感到好奇。这一切没有什么秘密,Qt 本身就是开源的。读者可以下载 Qt 源代码,并找到 Blend.qml 文件,它一般位于如下位置:

[Qt/qtgraphicaleffects/src/effects/Blend.qml](#)

其源代码如源码 6.2.2。本书为了便于印刷删除了注释与空行并调整了源码格式。

源码 6.2.2

```

1 import QtQuick 2.12
2 import QtGraphicalEffects.private 1.12
3 Item {
4     id: rootItem
5     property variant source
6     property variant foregroundSource
7     property string mode: "normal"
8     property bool cached: false
9     SourceProxy {
10         id: backgroundSourceProxy ;
11         input: rootItem.source ; }
12     SourceProxy {
13         id: foregroundSourceProxy ;
14         input: rootItem.foregroundSource ; }
15     ShaderEffectSource {
16         id: cacheItem
17         anchors.fill: parent ; visible: rootItem.cached ;
18         smooth: true ; sourceItem: shaderItem ;
19         live: true ; hideSource: visible ; }
20     ShaderEffect {
21         id: shaderItem
22         property variant backgroundSource: backgroundSourceProxy.output
23         property variant foregroundSource: foregroundSourceProxy.output
24         property string mode: rootItem.mode ; anchors.fill: parent ;
25         fragmentShader: fragmentShaderBegin + blendModeNormal + fragmentShaderEnd
26         function buildFragmentShader() {
27             var shader = fragmentShaderBegin
28             switch (mode.toLowerCase()) {
29                 case "addition" : shader += blendModeAddition; break;
30                 case "average" : shader += blendModeAverage; break;
31                 case "color" : shader += blendModeColor; break;
32                 case "colorburn" : shader += blendModeColorBurn; break;
33                 case "colordodge" : shader += blendModeColorDodge; break;
34                 case "darker" : shader += blendModeDarken; break;
35                 case "darkercolor" : shader += blendModeDarkerColor; break;
36                 case "difference" : shader += blendModeDifference; break;
37                 case "divide" : shader += blendModeDivide; break;
38                 case "exclusion" : shader += blendModeExclusion; break;
39                 case "hardlight" : shader += blendModeHardLight; break;
}
```

```

40         case "hue" : shader += blendModeHue; break;
41         case "lighten" : shader += blendModeLighten; break;
42         case "lightercolor" : shader += blendModeLighterColor; break;
43         case "lightness" : shader += blendModeLightness; break;
44         case "negation" : shader += blendModeNegation; break;
45         case "normal" : shader += blendModeNormal; break;
46         case "multiply" : shader += blendModeMultiply; break;
47         case "saturation" : shader += blendModeSaturation; break;
48         case "screen" : shader += blendModeScreen; break;
49         case "subtract" : shader += blendModeSubtract; break;
50         case "softlight" : shader += blendModeSoftLight; break;
51         default: shader += "gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);"; break; }
52         shader += fragmentShaderEnd ;
53         fragmentShader = shader ;
54         backgroundSourceChanged() ; }
55 Component.onCompleted: {buildFragmentShader() ; }
56 onModeChanged: {buildFragmentShader() ;}
57 property string blendModeAddition: "result.rgb = min(rgb1 + rgb2, 1.0);"
58 property string blendModeAverage: "result.rgb = 0.5 * (rgb1 + rgb2);"
59 property string blendModeColor: "result.rgb = HSLtoRGB(vec3(RGBtoHSL(rgb2).xy,
60             RGBtoL(rgb1)));"
61 property string blendModeColorBurn:
62     "result.rgb = clamp(1.0 - ((1.0 - rgb1) / max(vec3(1.0 / 256.0), rgb2)), vec3
63         (0.0), vec3(1.0));"
64 property string blendModeColorDodge:
65     "result.rgb = clamp(rgb1 / max(vec3(1.0 / 256.0), (1.0 - rgb2)), vec3(0.0), vec3
66         (1.0));"
67 property string blendModeDarken: "result.rgb = min(rgb1, rgb2);"
68 property string blendModeDarkerColor:
69     "result.rgb = 0.3 * rgb1.r + 0.59 * rgb1.g + 0.11 * rgb1.b > 0.3 * rgb2.r + 0.59
70         * rgb2.g + 0.11 * rgb2.b ? rgb2 : rgb1;"
71 property string blendModeDifference: "result.rgb = abs(rgb1 - rgb2);"
72 property string blendModeDivide: "result.rgb = clamp(rgb1 / rgb2, 0.0, 1.0);"
73 property string blendModeExclusion:
74     "result.rgb = rgb1 + rgb2 - 2.0 * rgb1 * rgb2;"
```

```
92     (GraphicsInfo.profile === GraphicsInfo.OpenGLCoreProfile ?
93      "#version 150 core
94      #define varying in
95      #define texture2D texture
96      out vec4 fragColor;
97      #define gl_FragColor fragColor " : "")"
98      property string fragmentShaderBegin: fragmentCoreShaderWorkaround + "
99      varying mediump vec2 qt_TexCoord0;
100     uniform highp float qt_Opacity;
101     uniform lowp sampler2D backgroundSource;
102     uniform lowp sampler2D foregroundSource;
103     highp float RGBtoL(highp vec3 color) {
104         highp float cmin = min(color.r, min(color.g, color.b));
105         highp float cmax = max(color.r, max(color.g, color.b));
106         highp float l = (cmin + cmax) / 2.0;
107         return l; }
108     highp vec3 RGBtoHSL(highp vec3 color) {
109         highp float cmin = min(color.r, min(color.g, color.b));
110         highp float cmax = max(color.r, max(color.g, color.b));
111         highp float h = 0.0;
112         highp float s = 0.0;
113         highp float l = (cmin + cmax) / 2.0;
114         highp float diff = cmax - cmin;
115         if (diff > 1.0 / 256.0) {
116             if (l < 0.5) { s = diff / (cmin + cmax); }
117             else { s = diff / (2.0 - (cmin + cmax)); }
118             if (color.r == cmax) {
119                 h = (color.g - color.b) / diff; }
120             else if (color.g == cmax) {
121                 h = 2.0 + (color.b - color.r) / diff; }
122             else {
123                 h = 4.0 + (color.r - color.g) / diff; }
124             h /= 6.0; }
125             return vec3(h, s, l); }
126     highp float hueToIntensity(highp float v1, highp float v2, highp float h) {
127         h = fract(h);
128         if (h < 1.0 / 6.0) {
129             return v1 + (v2 - v1) * 6.0 * h; }
130         else if (h < 1.0 / 2.0) { return v2; }
131         else if (h < 2.0 / 3.0) {
132             return v1 + (v2 - v1) * 6.0 * (2.0 / 3.0 - h); }
133         return v1; }
134     highp vec3 HSLtoRGB(highp vec3 color) {
135         highp float h = color.x;
136         highp float l = color.z;
137         highp float s = color.y;
138         if (s < 1.0 / 256.0) {
139             return vec3(l, l, l); }
140         highp float v1;
141         highp float v2;
142         if (l < 0.5) {
143             v2 = l * (1.0 + s); }
144         else {
145             v2 = (l + s) - (s * l); }
146         v1 = 2.0 * l - v2;
147         highp float d = 1.0 / 3.0;
148         highp float r = hueToIntensity(v1, v2, h + d);
149         highp float g = hueToIntensity(v1, v2, h);
150         highp float b = hueToIntensity(v1, v2, h - d);
151         return vec3(r, g, b); }
```

```

152     lowp float channelBlendHardLight(lowp float c1, lowp float c2) {
153         return c2 > 0.5 ?
154             (1.0 - (1.0 - 2.0 * (c2 - 0.5)) * (1.0 - c1))
155             : (2.0 * c1 * c2); }
156     void main() {
157         lowp vec4 result = vec4(0.0);
158         lowp vec4 color1 = texture2D(backgroundSource, qt_TexCoord0);
159         lowp vec4 color2 = texture2D(foregroundSource, qt_TexCoord0);
160         lowp vec3 rgb1 = color1.rgb / max(1.0/256.0, color1.a);
161         lowp vec3 rgb2 = color2.rgb / max(1.0/256.0, color2.a);
162         highp float a = max(color1.a, color1.a * color2.a); "
163         property string fragmentShaderEnd: "
164             gl_FragColor.rgb = mix(rgb1, result.rgb, color2.a);
165             gl_FragColor.rgb *= a;
166             gl_FragColor.a = a;
167             gl_FragColor *= qt_Opacity; } } }
```

源码 6.2.2

通过阅读源码 6.2.2，不难发现，Blend 仅仅是 ShaderEffect 的一个具体应用罢了。读者也可以结合本书第 2 章的内容写自己的特效。

6.3 BrightnessContrast

BrightnessContrast 用于调整亮度和对比度。其常见属性见表 6.3.1。

属性	类型	默认值	范围	简介
brightness	real	0.0	[−1, 1]	亮度
contrast	real	0.0	[−1, 1]	对比度
cached	bool	false	[false, true]	缓存
source	variant	⋮	∀ {可渲染对象}	源

表 6.3.1

表 6.3.1：类型属性

BrightnessContrast 的关键代码见源码 6.3.1。

源码 6.3.1

```

1 /*+glslcore/brightnesscontrast.frag*/
2 #version 150 core
3 in vec2 qt_TexCoord0;
4 uniform float qt_Opacity;
5 uniform sampler2D source;
6 uniform float brightness;
7 uniform float contrast;
8 out vec4 fragColor;
9
10 void main() {
11     vec4 pixelColor = texture(source, qt_TexCoord0);
12     pixelColor.rgb /= max(1.0/256.0, pixelColor.a);
13     float c = 1.0 + contrast;
14     float contrastGainFactor = 1.0 + c * c * c * c * step(0.0, contrast);
15     pixelColor.rgb =
16         ((pixelColor.rgb - 0.5) * (contrastGainFactor * contrast + 1.0)) + 0.5;
17     pixelColor.rgb =
18         mix(pixelColor.rgb, vec3(step(0.0, brightness)), abs(brightness));
19     fragColor = vec4(pixelColor.rgb * pixelColor.a, pixelColor.a) * qt_Opacity;
```

源码 6.3.1

20 }

本节案例源码见源码 6.3.2。

图 6.3.1

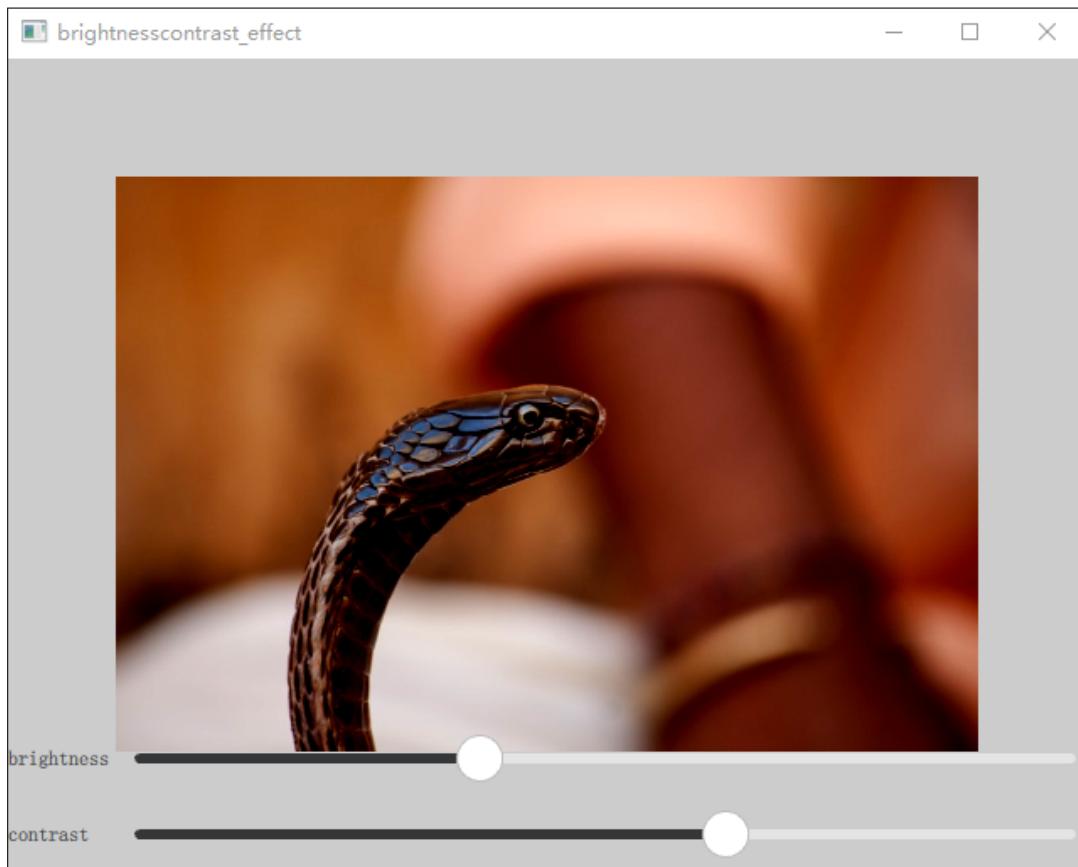


图 6.3.1 : BrightnessContrast

源码 6.3.2

```
1 /*brightnesscontrast_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6
7     id : idRoot
8     width: 640;
9     height: 480;
10    color: Qt.rgba(0.8,0.8,0.8,1);
11
12
13    Image{
14        width: parent.width * 0.8;
15        height: parent.height * 0.8;
16        anchors.centerIn: parent
17        source: "image.jpg"
18        visible: false
19        fillMode: Image.PreserveAspectFit
20        id : idImage
21    }
22
23    BrightnessContrast{
24        anchors.fill: idImage
25        source: idImage
26        contrast: idControl.contrastItem.value
27    }
28}
```

```

27     brightness: idControl.brightnessItem.value
28 }
29
30 BrightnessContrastControl{
31     id:idControl
32 }
33
34 }
```

源码 6.3.2

6.4 ColorOverlay

ColorOverlay 用于在可渲染对象上叠加一个颜色。其常见属性见表 6.3.1。

属性	类型	默认值	范围	简介
color	color	transparent	$\forall \{ \text{RGBA 颜色} \}$	覆盖颜色
cached	bool	false	$[false, true]$	缓存
source	variant	\exists	$\forall \{ \text{可渲染对象} \}$	源

表 6.4.1

表 6.4.1 : 类型属性

ColorOverlay 的关键代码见源码 6.4.1。

源码 6.4.1

```

1 /*+glslcore/coloroverlay.frag*/
2 #version 150 core
3 in vec2 qt_TexCoord0;
4 uniform float qt_Opacity;
5 uniform sampler2D source;
6 uniform vec4 color;
7 out vec4 fragColor;
8 void main() {
9     vec4 pixelColor = texture(source, qt_TexCoord0);
10    fragColor = vec4(
11        mix(pixelColor.rgb/max(pixelColor.a, 0.00390625),
12            color.rgb/max(color.a, 0.00390625),
13            color.a) * pixelColor.a,
14            pixelColor.a) * qt_Opacity;
15 }
```

源码 6.4.1

本节案例源码见源码 6.4.2。

图 6.4.1

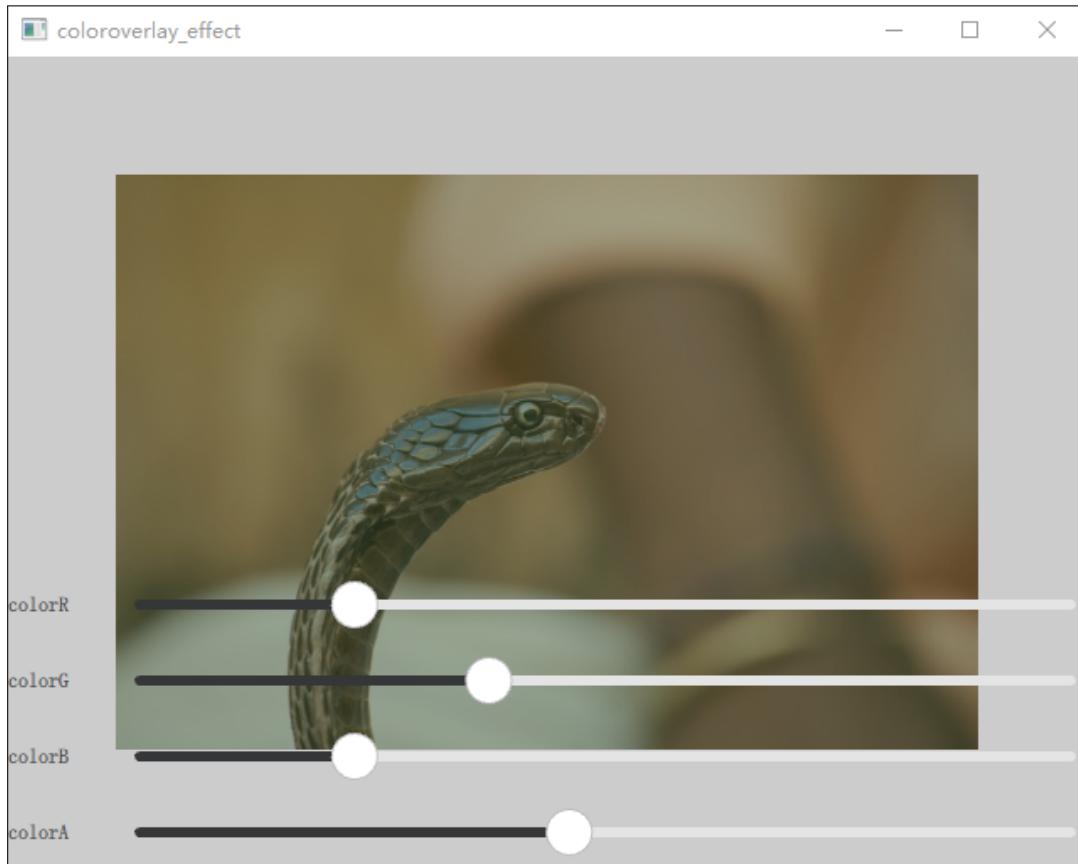


图 6.4.1 : ColorOverlay

源码 6.4.2

```
1 /*coloroverlay_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6
7     id : idRoot
8     width: 640;
9     height: 480;
10    color: Qt.rgba(0.8,0.8,0.8,1);
11
12    Image{
13        width: parent.width * 0.8;
14        height: parent.height * 0.8;
15        anchors.centerIn: parent
16        source: "image.jpg"
17        visible: false
18        fillMode: Image.PreserveAspectFit
19        id : idImage
20    }
21
22    ColorOverlay{
23        anchors.fill: idImage
24        source: idImage
25        color: idColoroverlayControl.applyColor
26    }
27
28    ColoroverlayControl{
29        id : idColoroverlayControl
30    }
```

```
31 }  
32 }
```

源码 6.4.2

6.5 Colorize

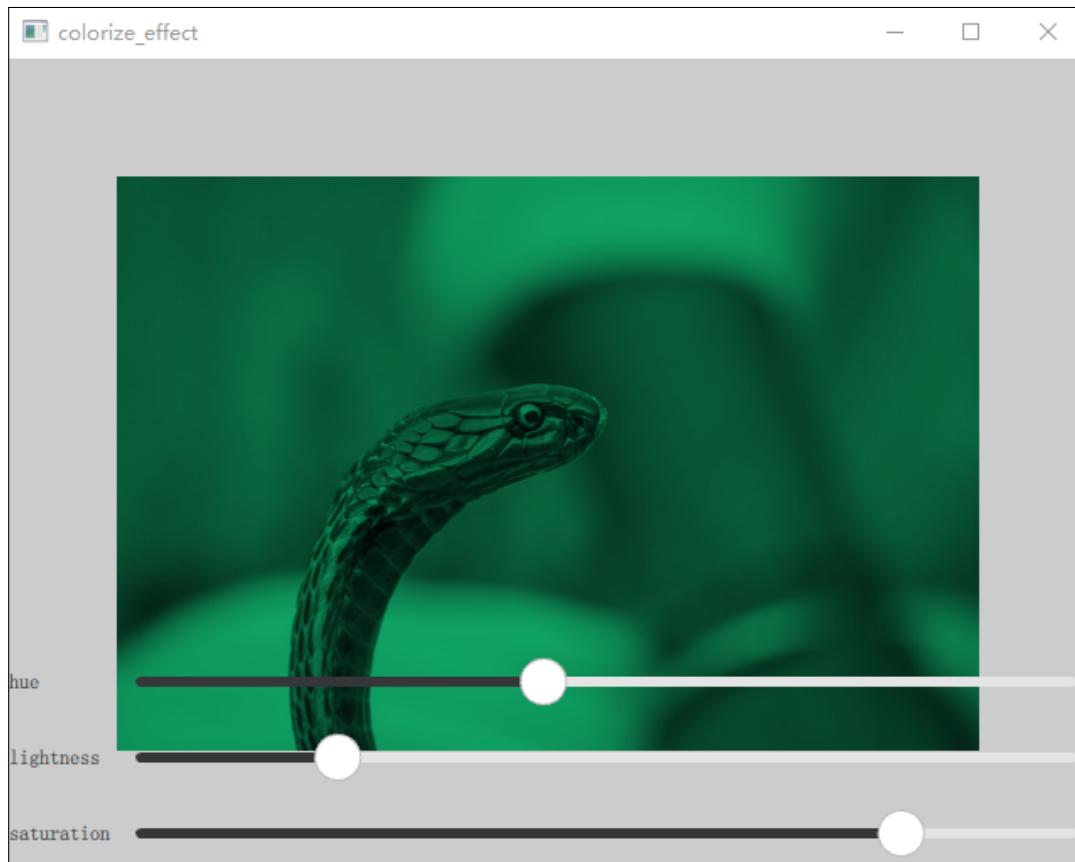


图 6.5.1

图 6.5.1 : Colorize

源码 6.5.1

```
1 /*colorize_effect/main.qml*/  
2 import QtQuick 2.9  
3 import QtGraphicalEffects 1.12  
4  
5 Rectangle {  
6     id : idRoot  
7     width: 640;  
8     height: 480;  
9     color: Qt.rgba(0.8,0.8,0.8,1);  
10  
11     Image{  
12         width: parent.width * 0.8;  
13         height: parent.height * 0.8;  
14         anchors.centerIn: parent  
15         source: "image.jpg"  
16         visible: false  
17         fillMode: Image.PreserveAspectFit  
18         id : idImage  
19     }  
20  
21     Colorize {  
22 }
```

```
23     anchors.fill: idImage
24     source: idImage
25     hue: idColorizeControl.hueItem.value;
26     lightness: idColorizeControl.lightnessItem.value;
27     saturation: idColorizeControl.saturationItem.value;
28 }
29
30 ColorizeControl{
31     id : idColorizeControl
32 }
33
34 }
```

源码 6.5.1

未完待续

6.6 Desaturate

图 6.6.1

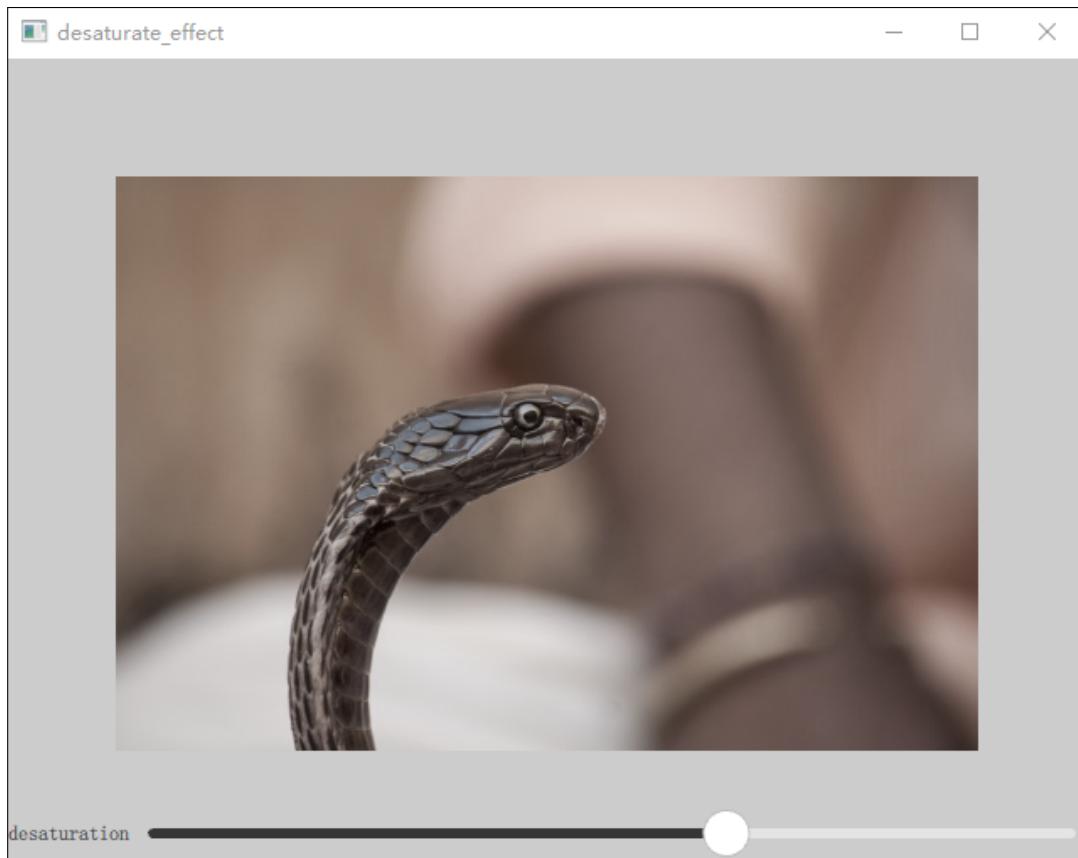


图 6.6.1 : Desaturate

源码 6.6.1

```
1 /*desaturate_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6
7     id : idRoot
8     width: 640;
9     height: 480;
10    color: Qt.rgba(0.8,0.8,0.8,1);
```

```
11 Image{  
12     width: parent.width * 0.8;  
13     height: parent.height * 0.8;  
14     anchors.centerIn: parent  
15     source: "image.jpg"  
16     visible: false  
17     fillMode: Image.PreserveAspectFit  
18     id : idImage  
19 }  
20  
21 Desaturate{  
22     anchors.fill: idImage  
23     source: idImage  
24     desaturation: thisControl.desaturationItem.value  
25 }  
26  
27 DesaturateControl{  
28     id : thisControl  
29 }  
30  
31 }  
32 }
```

源码 6.6.1

未完待续

6.7 GammaAdjust

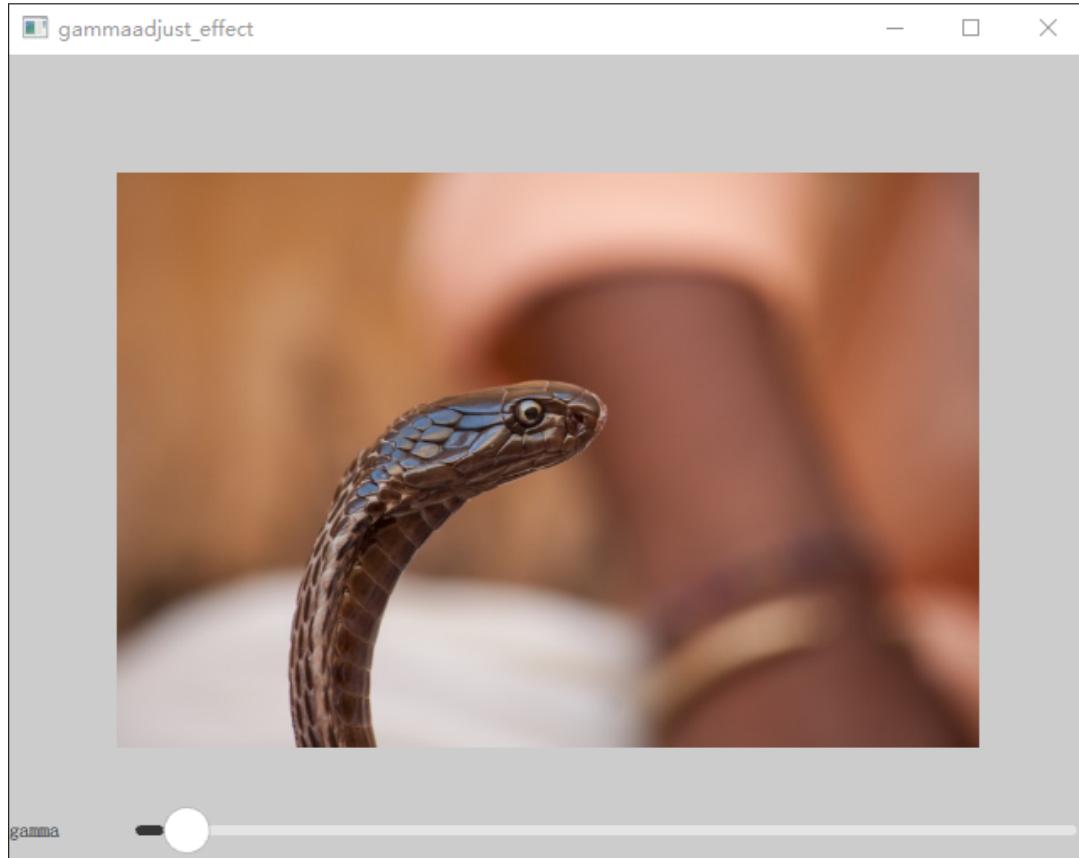


图 6.7.1

图 6.7.1 : GammaAdjust

源码 6.7.1

```
1 /*gammaadjust_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11    Image{
12        width: parent.width * 0.8;
13        height: parent.height * 0.8;
14        anchors.centerIn: parent
15        source: "image"
16        visible: false
17        fillMode: Image.PreserveAspectFit
18        id : idImage
19    }
20
21    GammaAdjust{
22        anchors.fill: idImage
23        source: idImage
24        gamma: idThisControl.gammaItem.value
25    }
26
27    GammaAdjustControl{
28        id : idThisControl
29    }
30
31 }
```

源码 6.7.1

未完待续

6.8 HueSaturation

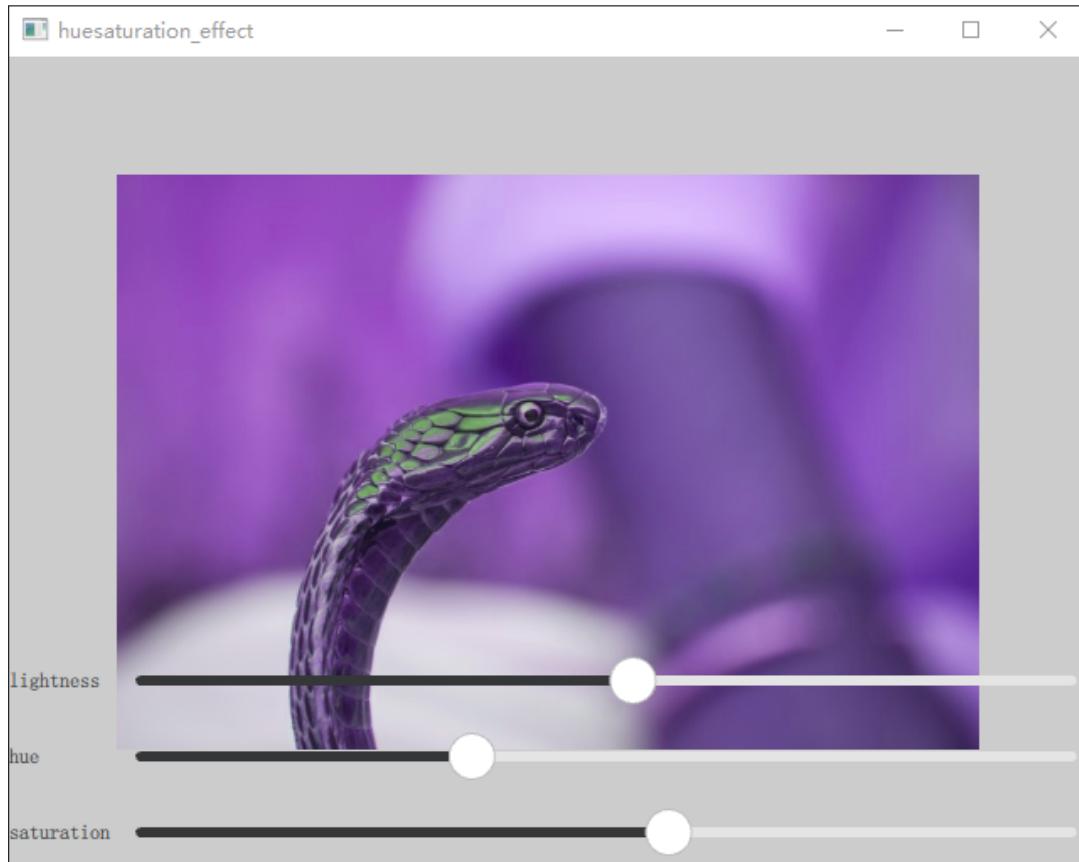


图 6.8.1

图 6.8.1 : HueSaturation

源码 6.8.1

```
1 /*huesaturation_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11     Image{
12         width: parent.width * 0.8;
13         height: parent.height * 0.8;
14         anchors.centerIn: parent
15         source: "image"
16         visible: false
17         fillMode: Image.PreserveAspectFit
18         id : idImage
19     }
20
21     HueSaturation{
22         anchors.fill: idImage
23         source: idImage
24         hue : thisControl.hueItem.value
25         lightness: thisControl.lightnessItem.value
26         saturation: thisControl.saturationItem.value
27     }
28
29     HuesaturationControl{
30         id : thisControl
```

源码 6.8.1

```
31     }
32 }
33 }
```

未完待续

6.9 LevelAdjust

图 6.9.1

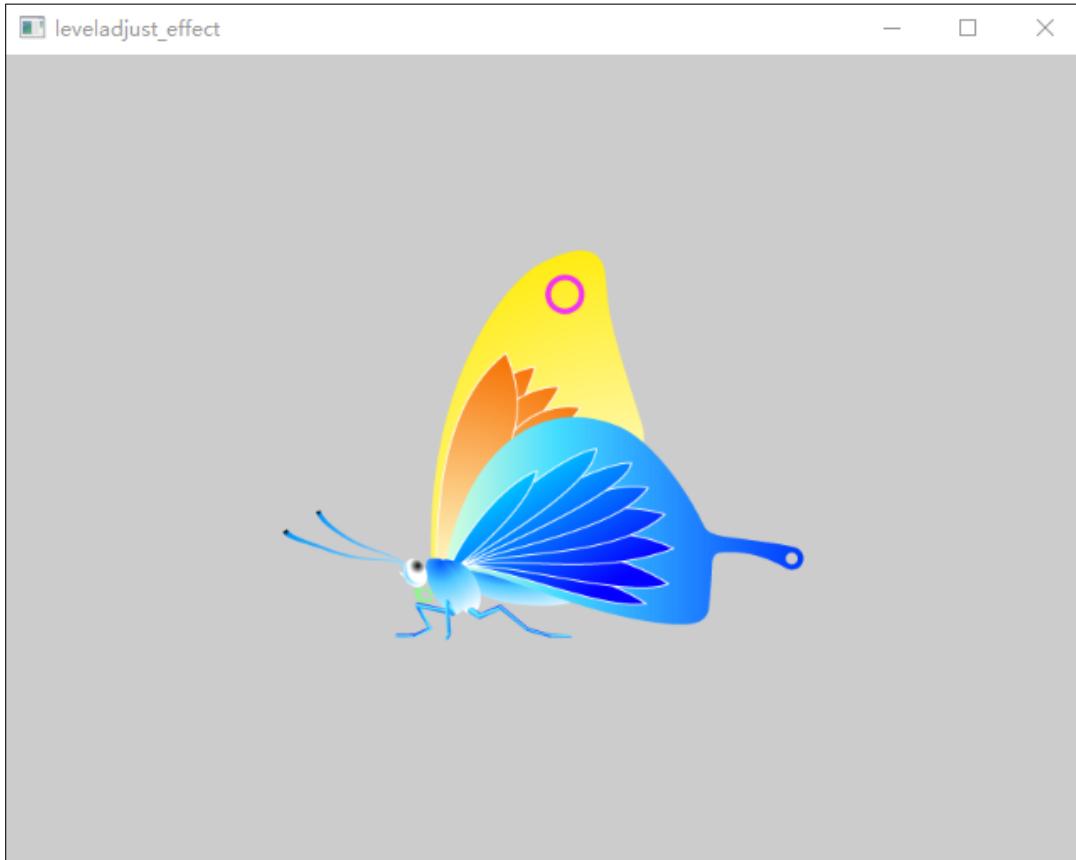


图 6.9.1 : LevelAdjust

源码 6.9.1

```
1 /*leveladjust_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11    Image{
12        width: parent.width * 0.8;
13        height: parent.height * 0.8;
14        anchors.centerIn: parent
15        source: "image"
16        visible: false
17        fillMode: Image.PreserveAspectFit
18        id : idImage
19    }
```

```
20  /*将RGB反向，A不变*/
21  LevelAdjust {
22      anchors.fill: idImage
23      source: idImage
24      minimumOutput: Qt.rgba(1,1,1,0)
25      maximumOutput: Qt.rgba(0,0,0,1)
26  }
27
28 }
29 }
```

源码 6.9.1

未完待续

6.10 ConicalGradient



图 6.10.1

图 6.10.1 : ConicalGradient

源码 6.10.1

```
1 /*conicalgradient_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9
10    Image{
11        width: parent.width * 0.8;
12        height: parent.height * 0.8;
```

```
13     anchors.centerIn: parent
14     source: "image.png"
15     visible: false
16     fillMode: Image.PreserveAspectFit
17     id : idImage
18 }
19
20 ConicalGradient {
21     anchors.fill: idImage
22     source: idImage
23     angle: 0.0
24     gradient: Gradient {
25         GradientStop { position: 0.0; color: "red" }
26         GradientStop { position: 1.0; color: "black" }
27     }
28 }
29
30 }
```

源码 6.10.1

未完待续

6.11 LinearGradient

图 6.11.1

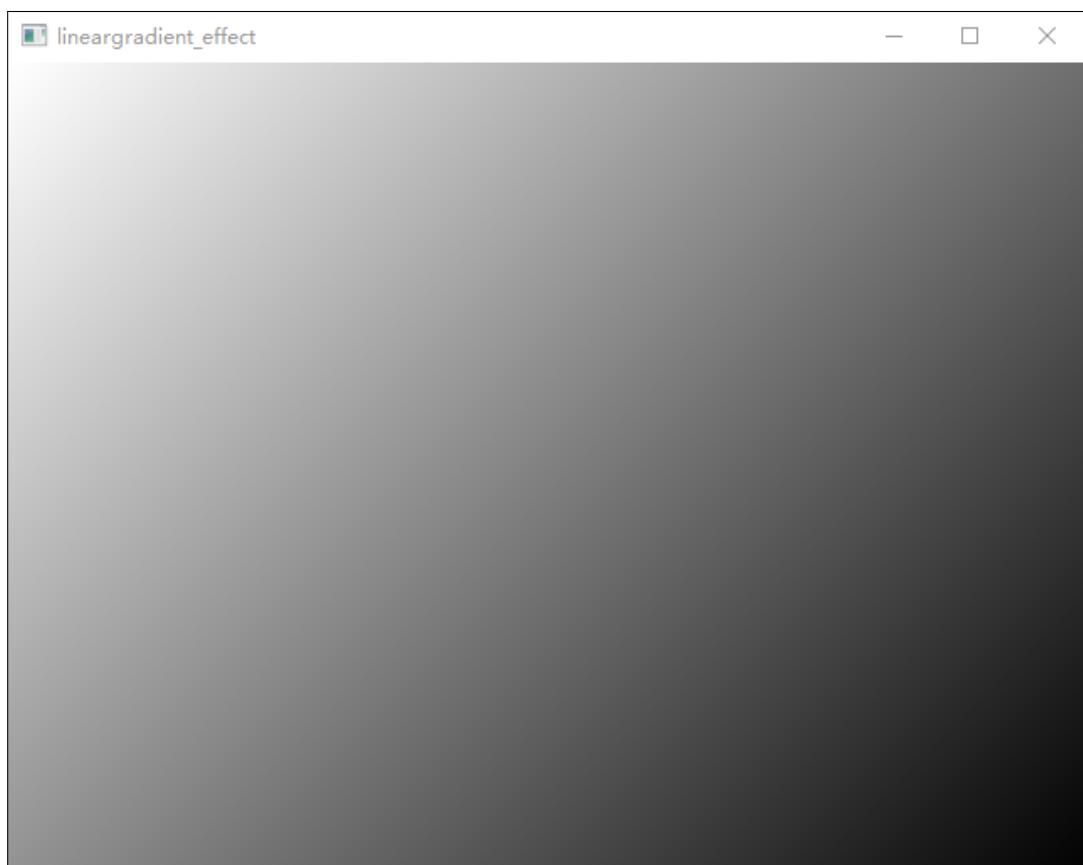


图 6.11.1 : LinearGradient

源码 6.11.1

```
1 /*lineargradient_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
```

```
5 Rectangle {  
6     id : idRoot  
7     width: 640;  
8     height: 480;  
9     color: Qt.rgba(0.8,0.8,0.8,1);  
10    }  
11    LinearGradient {  
12        id : idEffect  
13        anchors.fill: parent  
14        start: Qt.point(parent.x, parent.y)  
15        end: Qt.point(parent.x+parent.width,  
16                        parent.y+parent.height)  
17        gradient: Gradient {  
18            GradientStop { position: 0.0; color: "white" }  
19            GradientStop { position: 1.0; color: "black" }  
20        }  
21    }  
22}  
23}
```

源码 6.11.1

未完待续

6.12 RadialGradient

图 6.12.1

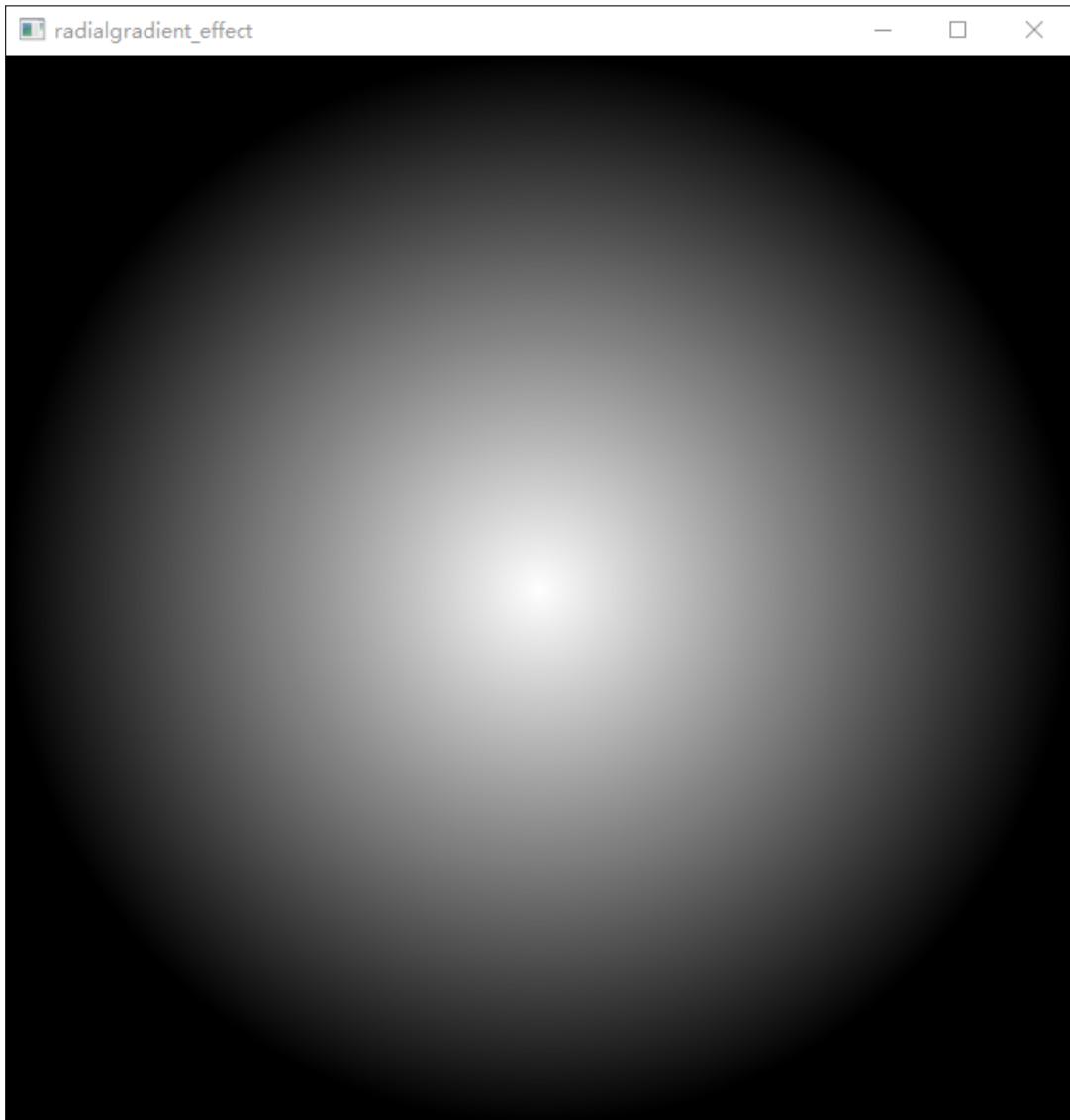


图 6.12.1 : RadialGradient

源码 6.12.1

```
1 /*radialgradient_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 640;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11     RadialGradient {
12         anchors.fill: parent
13         gradient: Gradient {
14             GradientStop { position: 0.0; color: "white" }
15             GradientStop { position: 0.5; color: "black" }
16         }
17     }
18 }
19 }
```

源码 6.12.1

未完待续

6.13 Displace

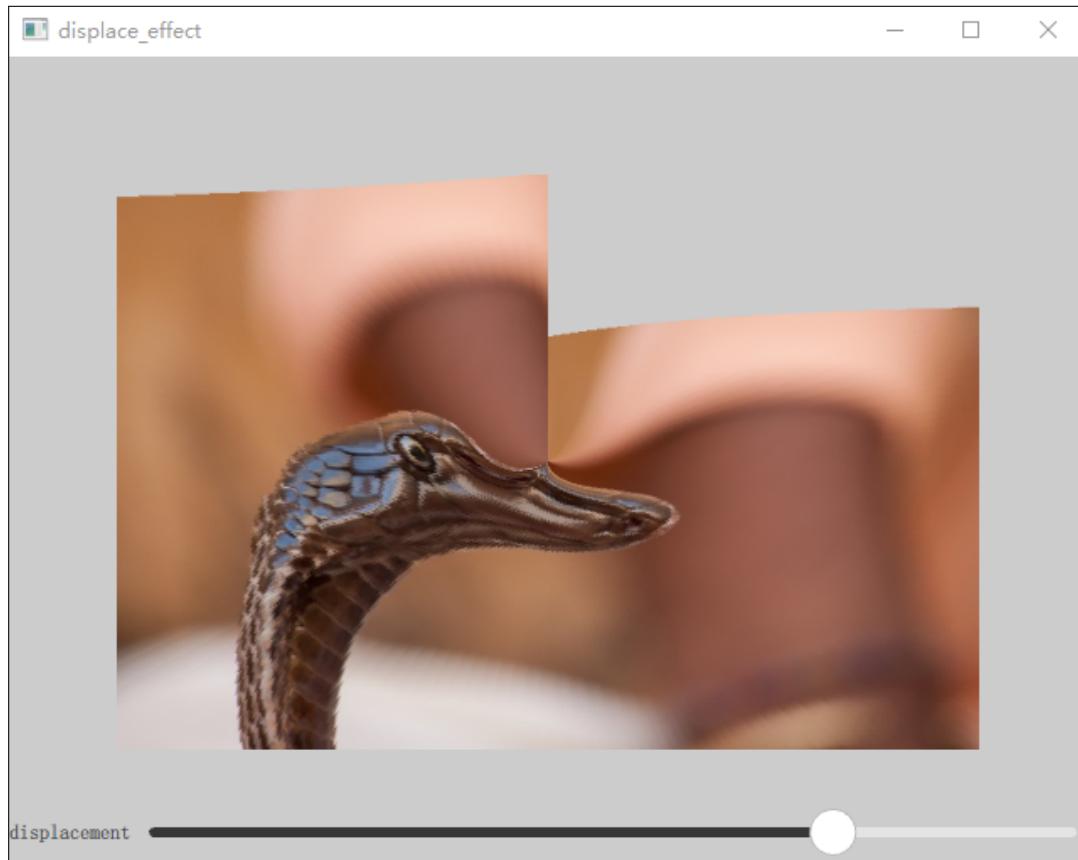


图 6.13.1

图 6.13.1 : Displace

源码 6.13.1

```
1 /*displace_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11    Image{
12        width: parent.width * 0.8;
13        height: parent.height * 0.8;
14        anchors.centerIn: parent
15        source: "image.jpg"
16        visible: false
17        fillMode: Image.PreserveAspectFit
18        id : idImage
19    }
20
21    ConicalGradient {
22        anchors.fill: idImage
23        source: idImage
24        angle: 0.0
25        gradient: Gradient {
26            GradientStop { position: 0.0; color: "red" }
```

```
27     GradientStop { position: 1.0; color: "green" }
28 }
29 visible: false
30 id : idRedGreenDisplacement
31 }
32
33 Displace{
34     anchors.fill: idImage
35     source: idImage
36     displacementSource: idRedGreenDisplacement
37     displacement : thisControl.displacementItem.value
38 }
39
40 DisplaceControl{
41     id : thisControl
42 }
43
44 }
```

源码 6.13.1

未完待续

6.14 DropShadow

图 6.14.1

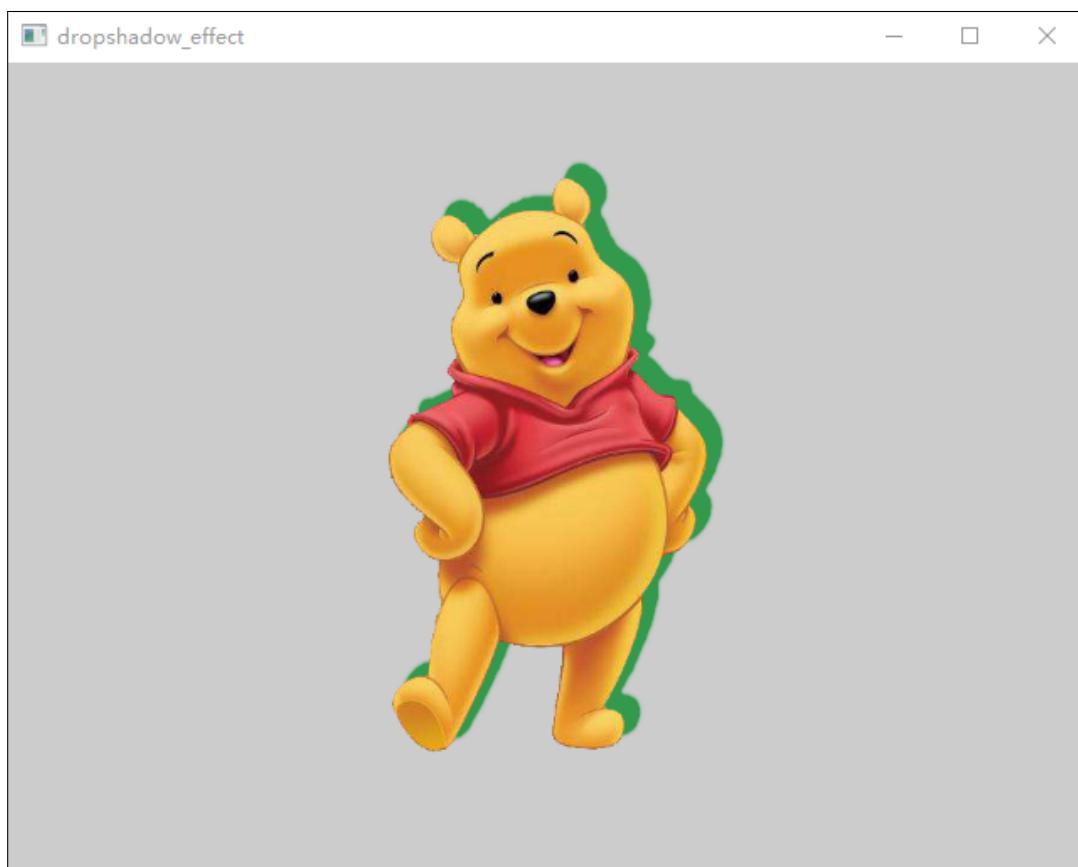


图 6.14.1 : DropShadow

源码 6.14.1

```
1 /*dropshadow_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
```

```
5 Rectangle {  
6     id : idRoot  
7     width: 640;  
8     height: 480;  
9     color: Qt.rgba(0.8,0.8,0.8,1);  
10    }  
11    Image{  
12        width: parent.width * 0.8;  
13        height: parent.height * 0.8;  
14        anchors.centerIn: parent  
15        source: "image"  
16        visible: false  
17        fillMode: Image.PreserveAspectFit  
18        id : idImage  
19    }  
20    }  
21    DropShadow{  
22        color: Qt.rgba(0.2,0.6,0.3,1)  
23        horizontalOffset: 8  
24        verticalOffset: -8  
25        radius: 8  
26        samples: 36  
27        source: idImage  
28        spread: 0.5  
29        anchors.fill: idImage  
30    }  
31    }  
32 }
```

源码 6.14.1

未完待续

6.15 InnerShadow

图 6.15.1

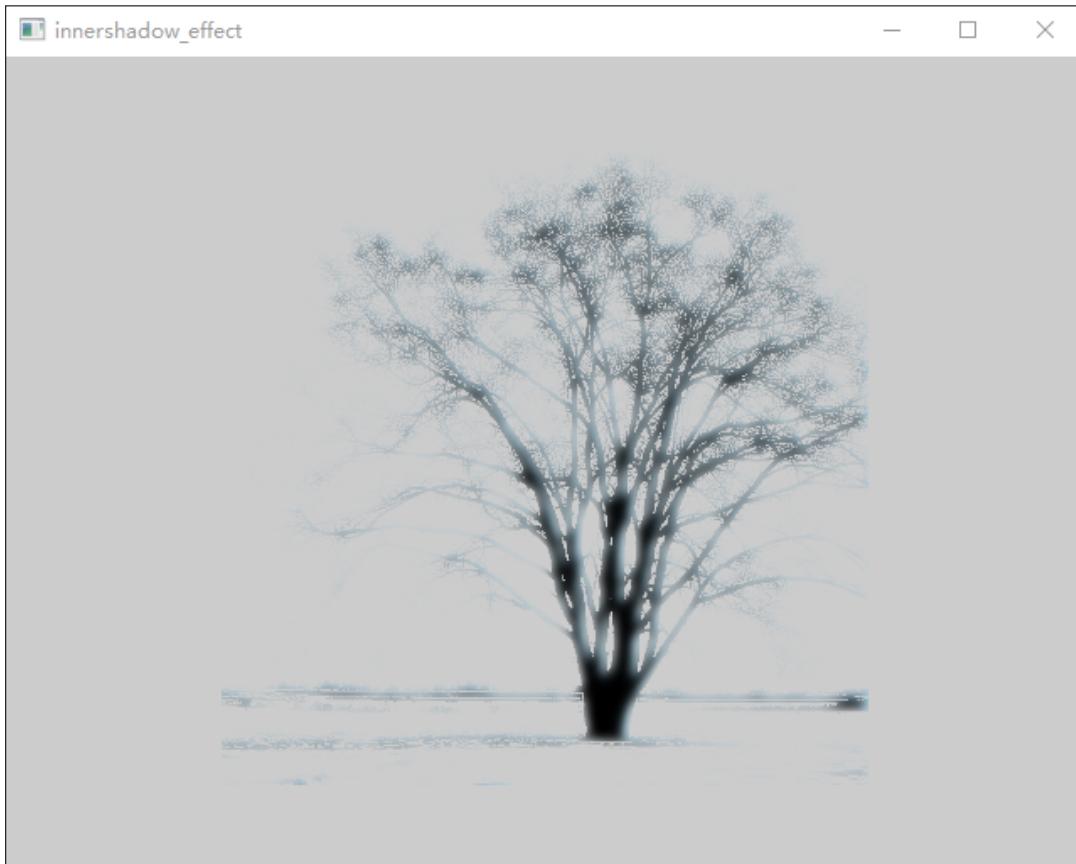


图 6.15.1 : InnerShadow

源码 6.15.1

```
1 /*innershadow_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11    Image{
12        width: parent.width * 0.8;
13        height: parent.height * 0.8;
14        anchors.centerIn: parent
15        source: "image"
16        visible: false
17        fillMode: Image.PreserveAspectFit
18        id : idImage
19    }
20
21    InnerShadow{
22        anchors.fill: idImage
23        source: idImage
24        radius: 8
25        samples: 16
26        horizontalOffset: -3
27        verticalOffset: 3
28        color: Qt.rgba( 0.6,0.8,0.9,1 );
29    }
30}
```

31 }

源码 6.15.1

未完待续

6.16 FastBlur

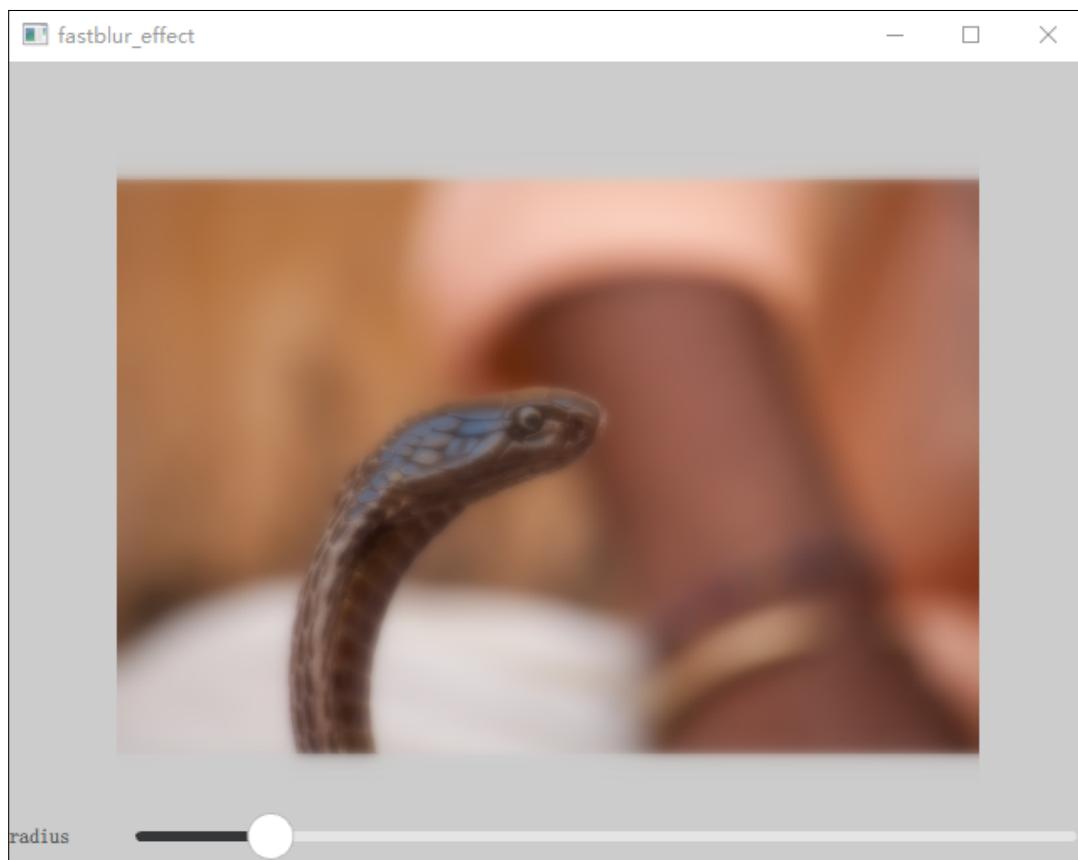


图 6.16.1

图 6.16.1 : FastBlur

源码 6.16.1

```
1 /*fastblur_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11    Image{
12        width: parent.width * 0.8;
13        height: parent.height * 0.8;
14        anchors.centerIn: parent
15        source: "image"
16        visible: false
17        fillMode: Image.PreserveAspectFit
18        id : idImage
19    }
20
21    FastBlur{
```

```
22     anchors.fill: idImage
23     source: idImage
24     radius: thisControl.radiusItem.value
25 }
26
27 FastBlurControl{
28     id : thisControl
29 }
30
31 }
```

源码 6.16.1

未完待续

6.17 GaussianBlur

图 6.17.1

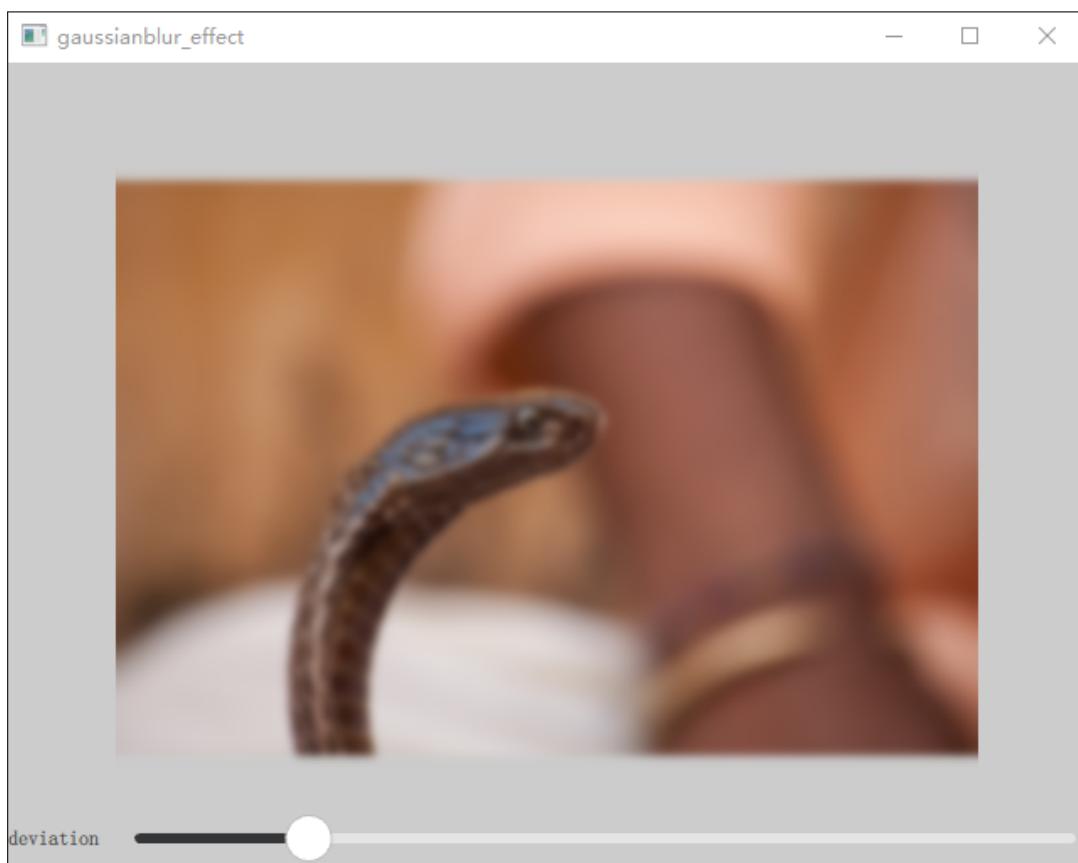


图 6.17.1 : GaussianBlur

源码 6.17.1

```
1 /*gaussianblur_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11    Image{
12         width: parent.width * 0.8;
```

```
13     height: parent.height * 0.8;
14     anchors.centerIn: parent
15     source: "image"
16     visible: false
17     fillMode: Image.PreserveAspectFit
18     id : idImage
19 }
20
21 GaussianBlur{
22     anchors.fill: idImage
23     source: idImage
24     radius: 8
25     samples: 16
26     deviation: idThisControl.deviationItem.value
27 }
28
29 GaussianBlurControl{
30     id : idThisControl
31 }
32 }
33 }
```

源码 6.17.1

未完待续

6.18 MaskedBlur

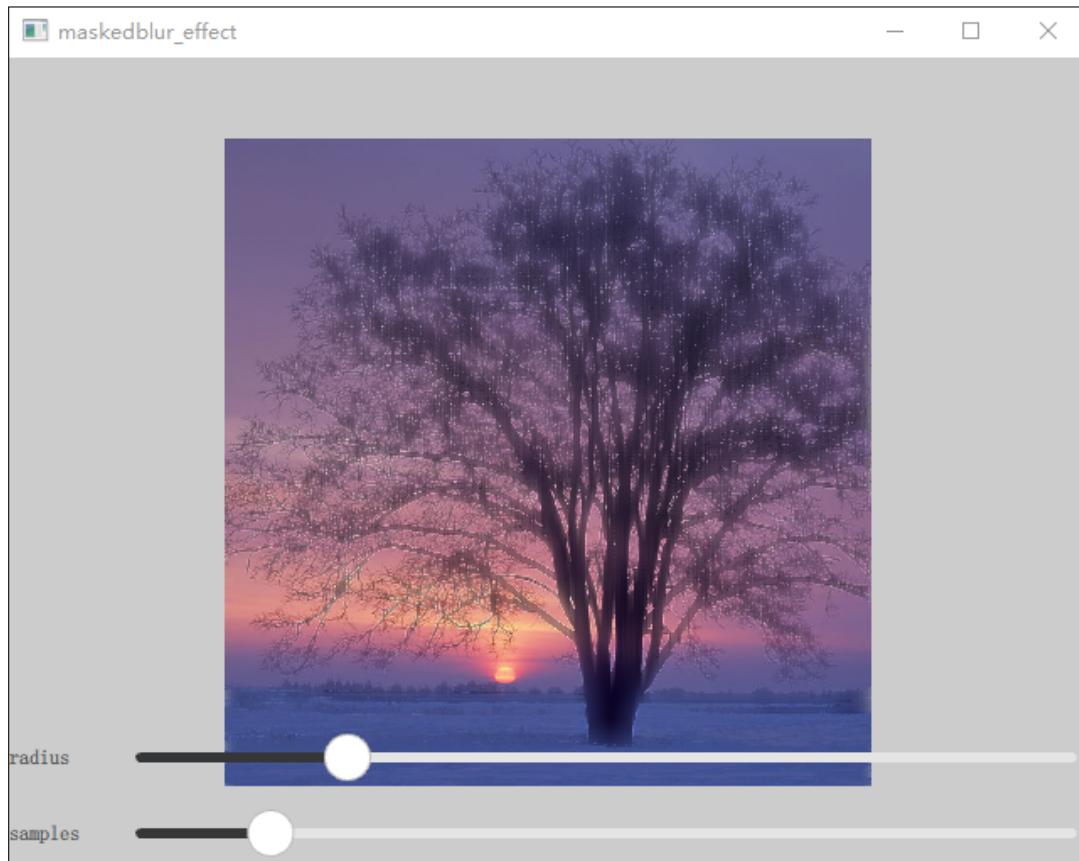


图 6.18.1

图 6.18.1 : MaskedBlur

源码 6.18.1

1 /*maskedblur_effect/main.qml*/

```
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11    Image{
12        width: parent.width * 0.8;
13        height: parent.height * 0.8;
14        anchors.centerIn: parent
15        source: "image.png"
16        visible: false
17        fillMode: Image.PreserveAspectFit
18        id : idMask
19    }
20
21    Image{
22        width: parent.width * 0.8;
23        height: parent.height * 0.8;
24        anchors.centerIn: parent
25        source: "image.jpg"
26        visible: false
27        fillMode: Image.PreserveAspectFit
28        id : idImage
29    }
30
31    MaskedBlur{
32        source: idImage
33        maskSource: idMask
34        anchors.fill: idImage
35        radius: idThisControl.radiusItem.value
36        samples: idThisControl.samplesItem.value
37    }
38
39    MaskedBlurControl{
40        id : idThisControl
41    }
42}
43}
```

源码 6.18.1

未完待续

6.19 RecursiveBlur

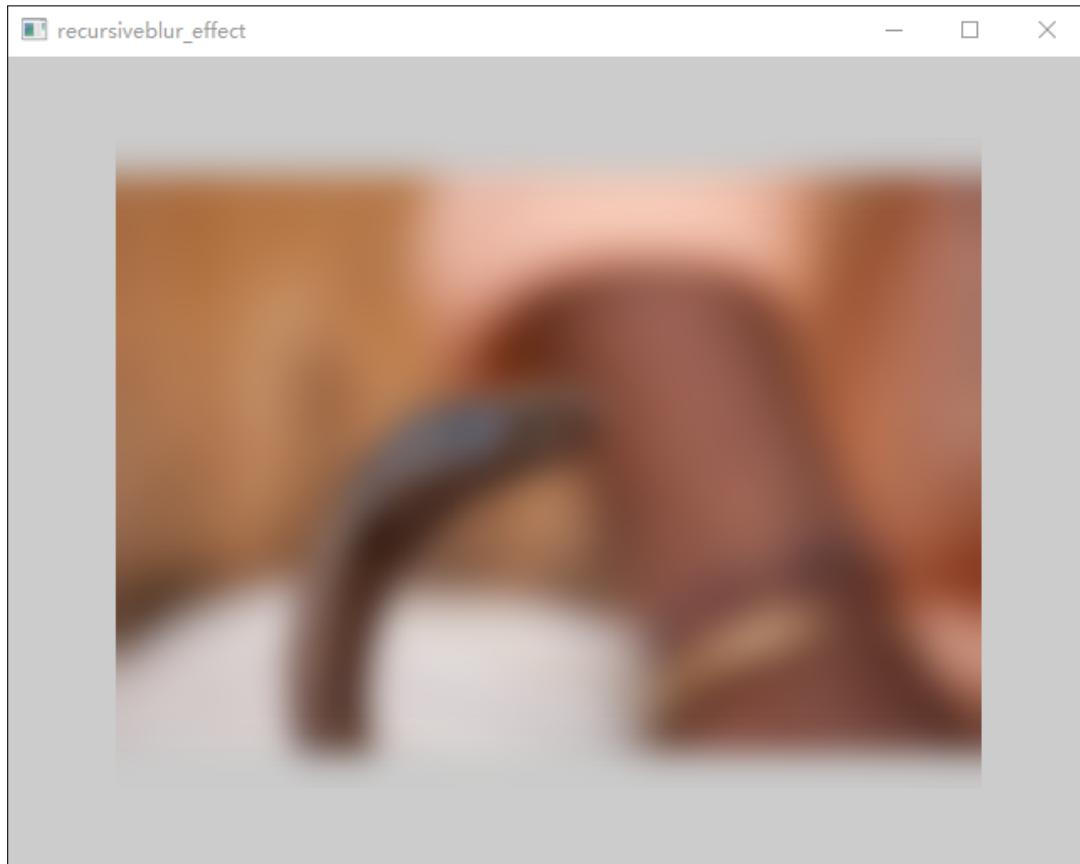


图 6.19.1

图 6.19.1 : RecursiveBlur

源码 6.19.1

```
1 /*recursiveblur_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11     Image{
12         width: parent.width * 0.8;
13         height: parent.height * 0.8;
14         anchors.centerIn: parent
15         source: "image.jpg"
16         visible: false
17         fillMode: Image.PreserveAspectFit
18         id : idImage
19     }
20
21     RecursiveBlur{
22         anchors.fill: idImage
23         source: idImage
24         loops: 8
25         radius: 8
26     }
27 }
```

源码 6.19.1

6.20 DirectionalBlur

图 6.20.1

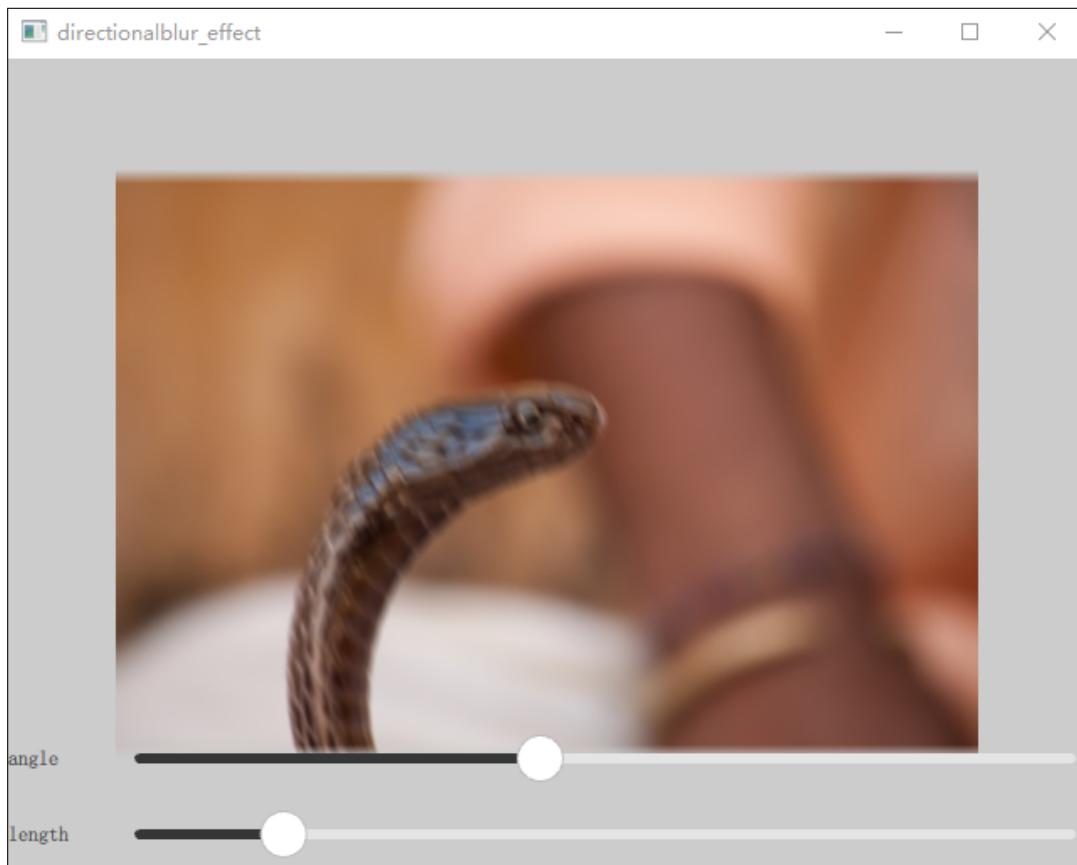


图 6.20.1 : DirectionalBlur

源码 6.20.1

```
1 /*directionalblur_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11     Image{
12         width: parent.width * 0.8;
13         height: parent.height * 0.8;
14         anchors.centerIn: parent
15         source: "image.jpg"
16         visible: false
17         fillMode: Image.PreserveAspectFit
18         id : idImage
19     }
20
21     DirectionalBlur{
22         anchors.fill: idImage
23         source: idImage
24         samples: 64
25         length: idThisControl.lengthItem.value
26         angle: idThisControl.angleItem.value
27     }
28 }
```

```
27     transparentBorder : false
28     id:idEffect
29   }
30
31 DirectionalblurControl{
32     id : idThisControl
33     lengthItem.to: idEffect.samples
34   }
35
36 }
```

源码 6.20.1

未完待续

6.21 RadialBlur

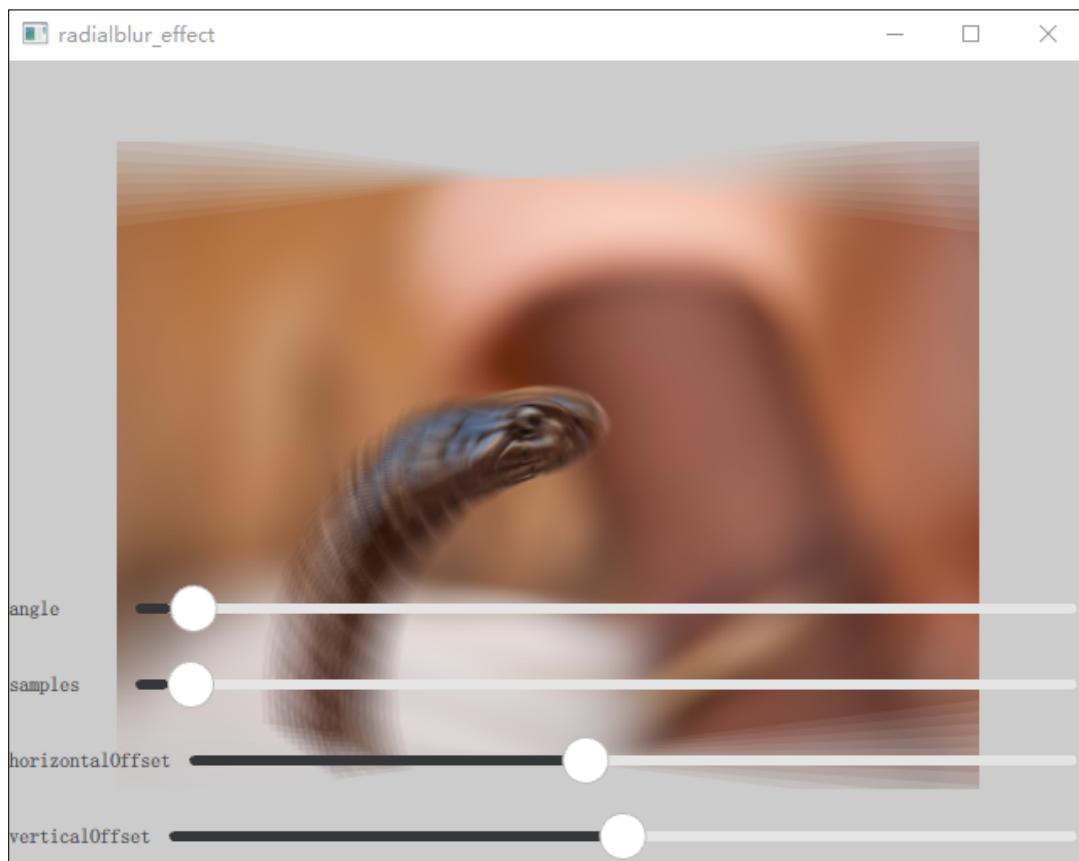


图 6.21.1

图 6.21.1 : RadialBlur

源码 6.21.1

```
1 /*radialblur_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11    Image{
12        width: parent.width * 0.8;
```

```
13     height: parent.height * 0.8;
14     anchors.centerIn: parent
15     source: "image.jpg"
16     visible: false
17     fillMode: Image.PreserveAspectFit
18     id : idImage
19 }
20
21 RadialBlur{
22     anchors.fill: idImage
23     source: idImage
24     angle: thisControl.angleItem.value
25     samples: thisControl.samplesItem.value
26     verticalOffset:
27         thisControl.verticalOffsetItem.value
28     horizontalOffset:
29         thisControl.horizontalOffsetItem.value
30 }
31
32 RadialBlurControl{
33     id : thisControl
34 }
35
36 }
```

源码 6.21.1

未完待续

6.22 ZoomBlur

图 6.22.1

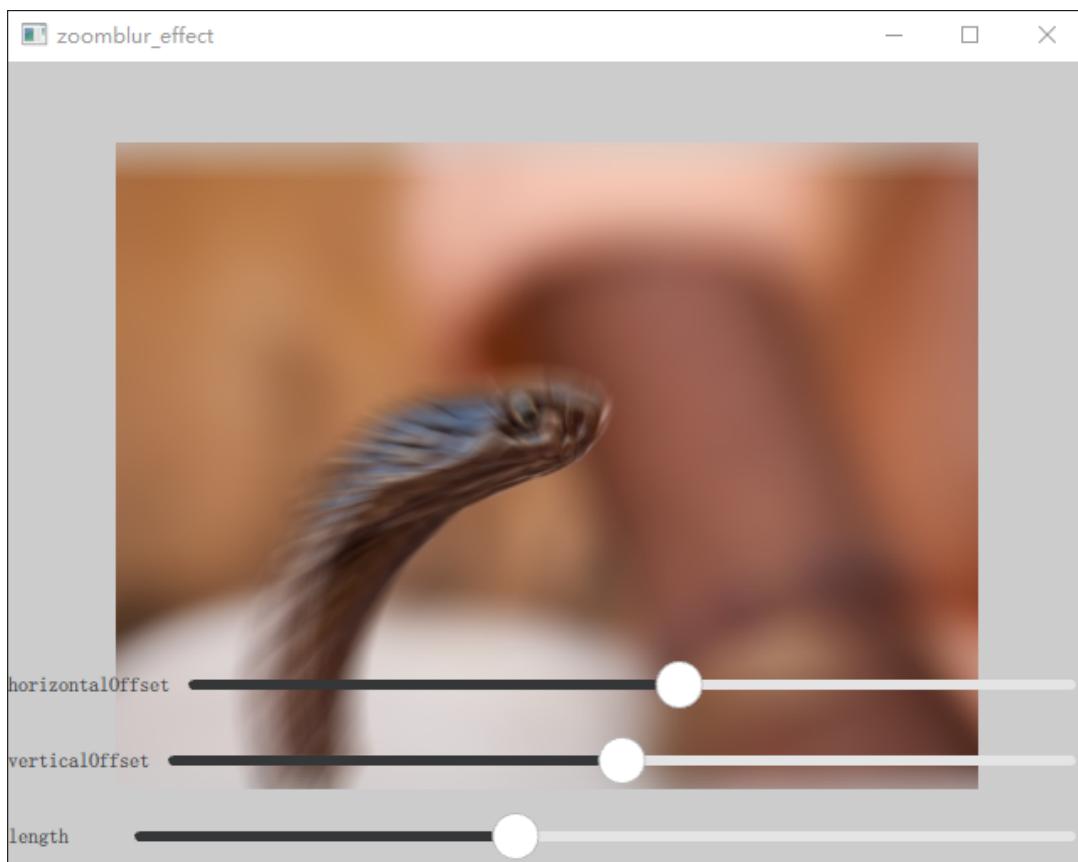


图 6.22.1 : ZoomBlur

源码 6.22.1

```
1 /*zoomblur_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11     Image{
12         width: parent.width * 0.8;
13         height: parent.height * 0.8;
14         anchors.centerIn: parent
15         source: "image"
16         visible: false
17         fillMode: Image.PreserveAspectFit
18         id : idImage
19     }
20
21     ZoomBlur {
22         anchors.fill: idImage
23         source: idImage
24         samples: 24
25         length:
26             idThisControl.lengthItem.value
27         verticalOffset:
28             idThisControl.verticalOffsetItem.value
29         horizontalOffset :
30             idThisControl.horizontalOffsetItem.value
31     }
32
33     ZoomBlurControl{
34         id : idThisControl
35     }
36
37 }
```

源码 6.22.1

未完待续

6.23 Glow

图 6.23.1

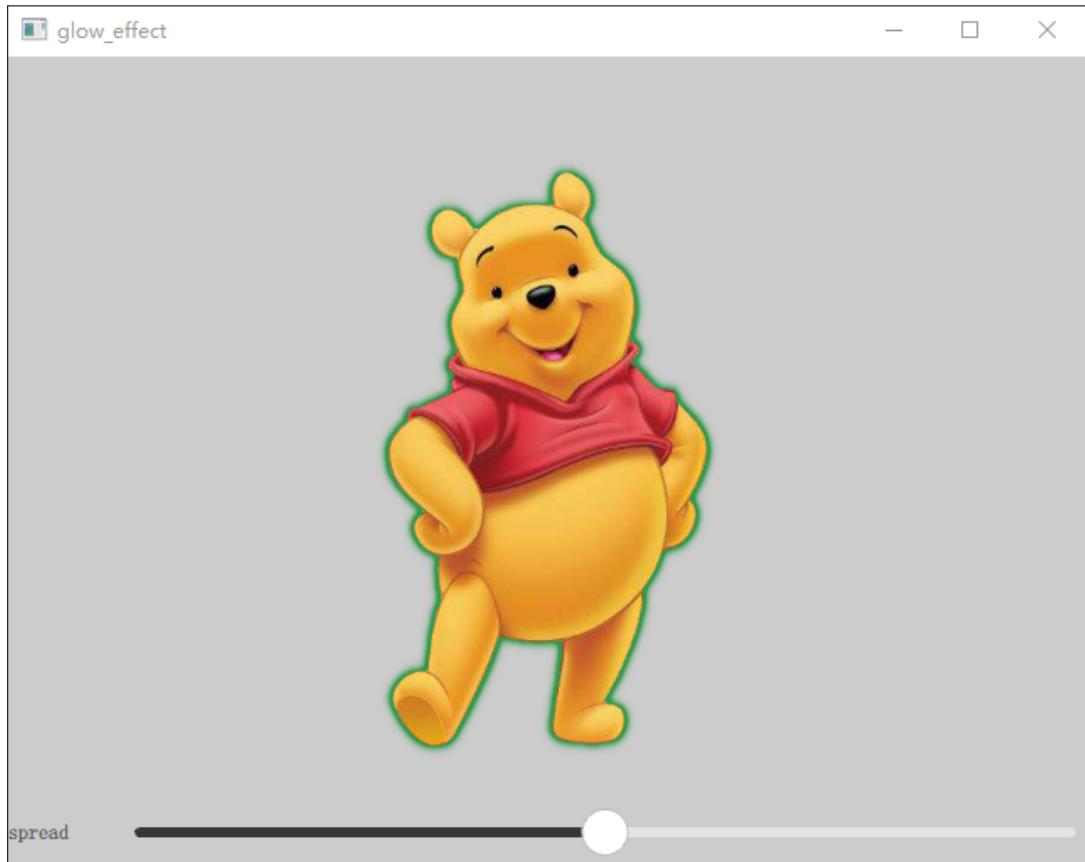


图 6.23.1 : Glow

源码 6.23.1

```
1 /*glow_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11    Image{
12        width: parent.width * 0.8;
13        height: parent.height * 0.8;
14        anchors.centerIn: parent
15        source: "image"
16        visible: false
17        fillMode: Image.PreserveAspectFit
18        id : idImage
19    }
20
21    Glow{
22        color: Qt.rgba(0.2,0.6,0.3,1)
23        radius: 8
24        samples: 16
25        spread: idThisControl.spreadItem.value
26        anchors.fill: idImage
27        source: idImage
28    }
29
30    GlowControl{
```

```
31     id : idThisControl  
32 }  
33  
34 }
```

源码 6.23.1

未完待续

6.24 RectangularGlow

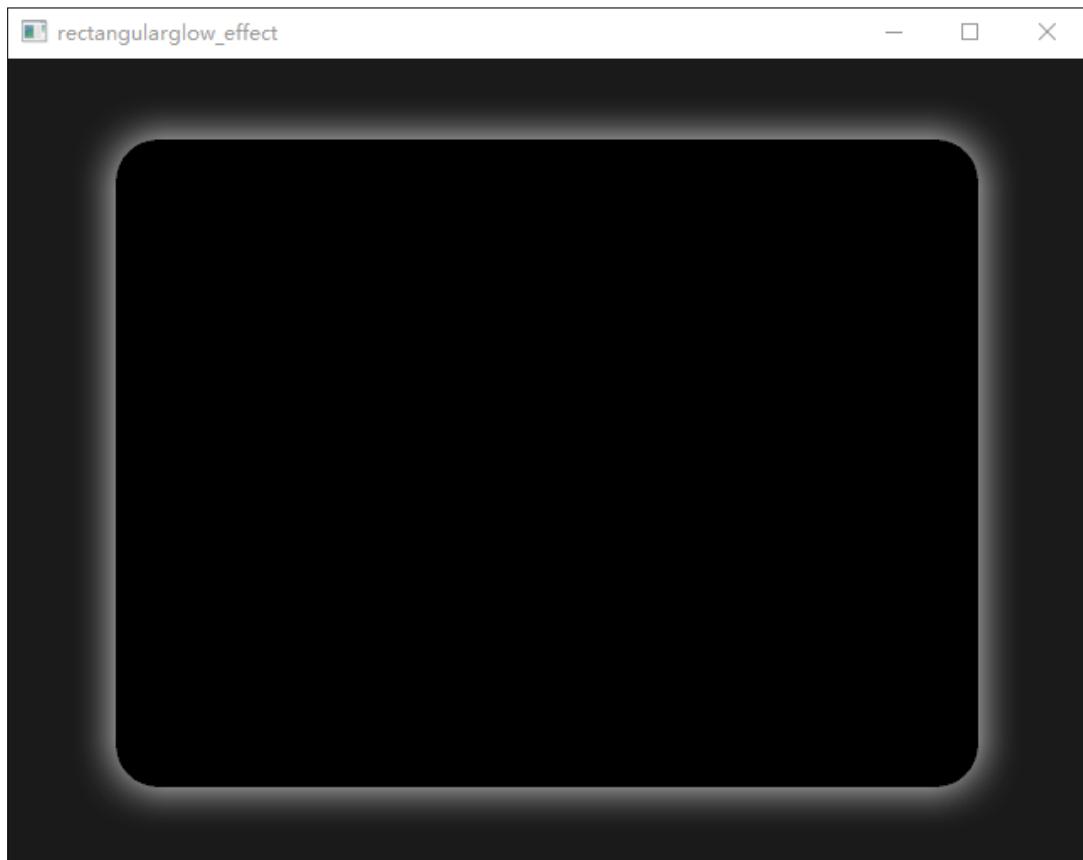


图 6.24.1

图 6.24.1 : RectangularGlow

源码 6.24.1

```
1 /*rectangularglow_effect/main.qml*/  
2 import QtQuick 2.9  
3 import QtGraphicalEffects 1.12  
4  
5 Rectangle {  
6     id : idRoot  
7     width: 640;  
8     height: 480;  
9     color: Qt.rgba(0.1,0.1,0.1,1);  
10  
11     RectangularGlow {  
12         id: idEffect  
13         anchors.fill: idRect  
14         glowRadius: 10  
15         spread: 0.2  
16         color: "white"  
17         cornerRadius: glowRadius + idRect.radius  
18     }  
19 }
```

```
19 Rectangle {  
20     id: idRect  
21     color: "black"  
22     width: parent.width * 0.8  
23     height: parent.height * 0.8  
24     anchors.centerIn: parent  
25     radius: 25  
26 }  
27  
28 }  
29 }
```

源码 6.24.1

未完待续

6.25 OpacityMask

图 6.25.1

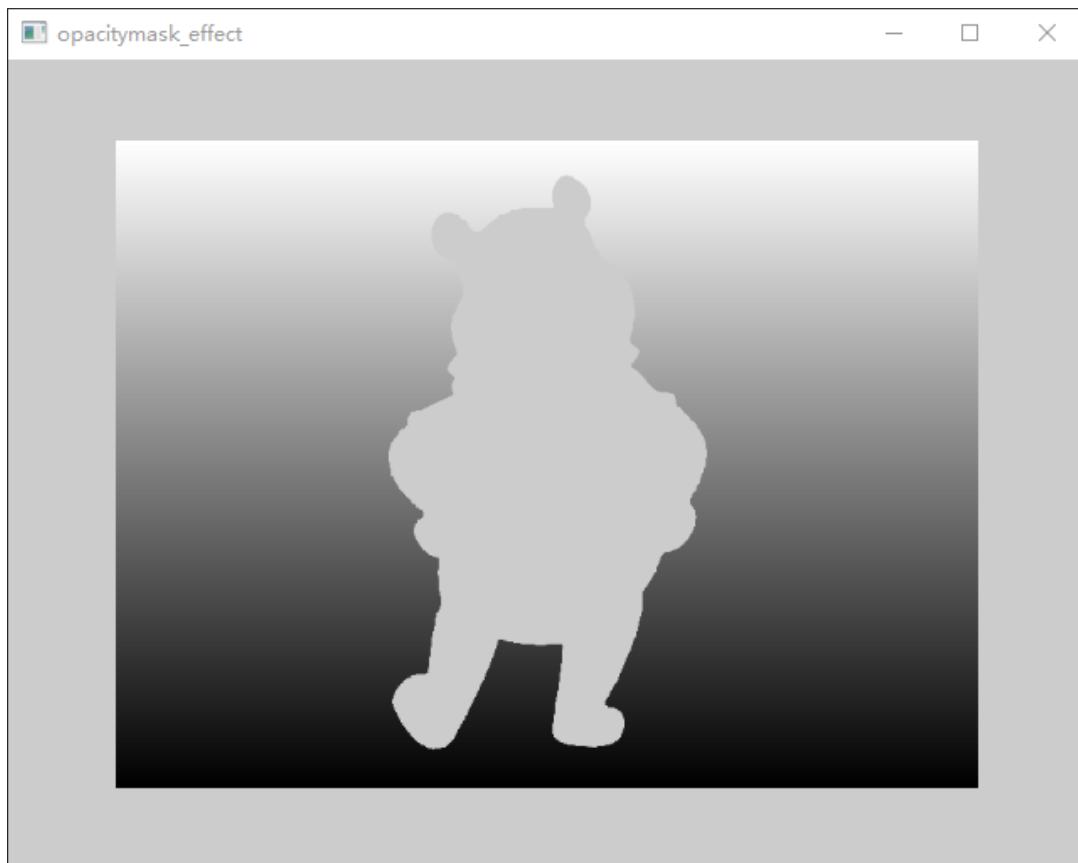


图 6.25.1 : OpacityMask

源码 6.25.1

```
1 /*opacitymask_effect/main.qml*/  
2 import QtQuick 2.9  
3 import QtGraphicalEffects 1.12  
4  
5 Rectangle {  
6     id : idRoot  
7     width: 640;  
8     height: 480;  
9     color: Qt.rgba(0.8,0.8,0.8,1);  
10  
11     Image{
```

```
12     width: parent.width * 0.8;
13     height: parent.height * 0.8;
14     anchors.centerIn: parent
15     source: "image"
16     visible: false
17     fillMode: Image.PreserveAspectFit
18     id : idMask
19 }
20
21 LinearGradient{
22     id : idSource
23     anchors.fill: idMask
24     gradient: Gradient {
25         GradientStop { position: 0.0; color: "white" }
26         GradientStop { position: 1.0; color: "black" }
27     }
28     visible: false
29 }
30
31 OpacityMask{
32     anchors.fill : idSource
33     invert: true
34     maskSource: idMask
35     source: idSource
36 }
37
38 }
```

源
码 6.25.1

未完待续

6.26 ThresholdMask

图 6.26.1

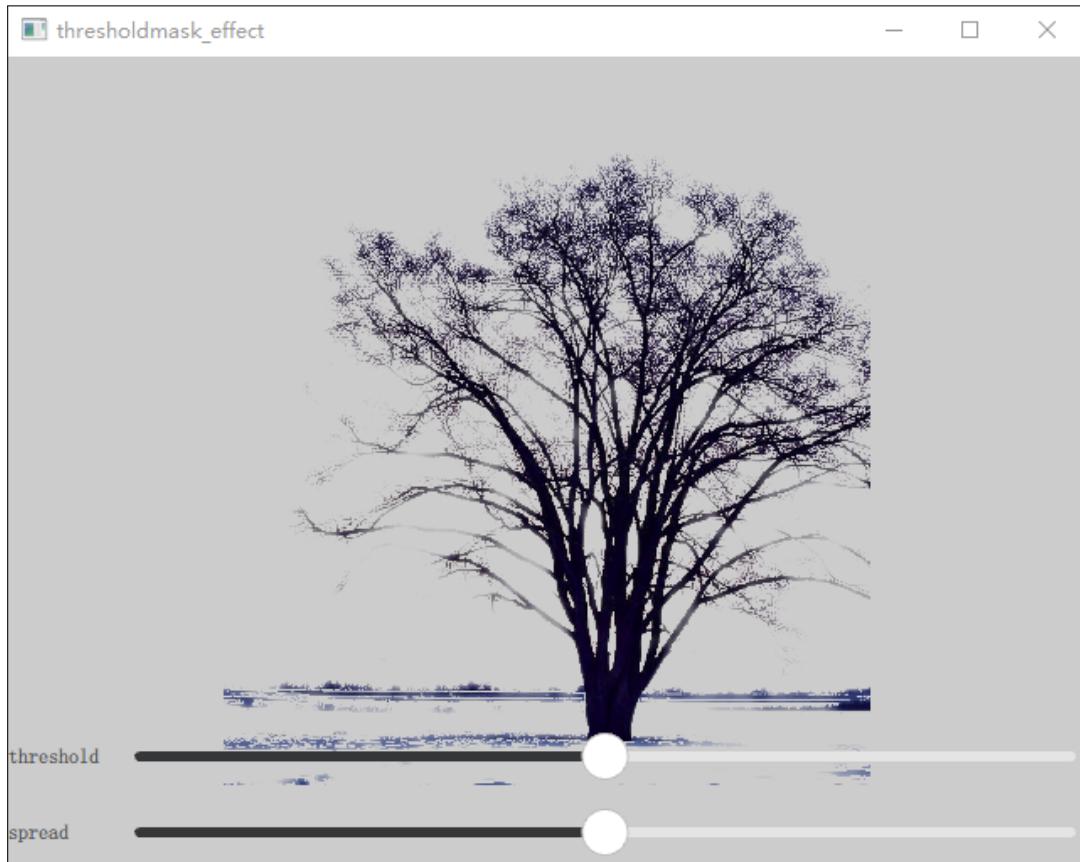


图 6.26.1 : ThresholdMask

源码 6.26.1

```
1 /*thresholdmask_effect/main.qml*/
2 import QtQuick 2.9
3 import QtGraphicalEffects 1.12
4
5 Rectangle {
6     id : idRoot
7     width: 640;
8     height: 480;
9     color: Qt.rgba(0.8,0.8,0.8,1);
10
11    Image{
12        width: parent.width * 0.8;
13        height: parent.height * 0.8;
14        anchors.centerIn: parent
15        source: "image.jpg"
16        visible: false
17        fillMode: Image.PreserveAspectFit
18        id : idImage
19    }
20
21    Image{
22        width: parent.width * 0.8;
23        height: parent.height * 0.8;
24        anchors.centerIn: parent
25        source: "image.png"
26        visible: false
27        fillMode: Image.PreserveAspectFit
28        id : idImageMask
29    }
30}
```

```
31     ThresholdMask{  
32         anchors.fill: idImage  
33         source: idImage  
34         maskSource: idImageMask  
35         spread: thisControl.spreadItem.value  
36         threshold: thisControl.thresholdItem.value  
37     }  
38  
39     ThresholdMaskControl{  
40         id : thisControl  
41     }  
42 }  
43 }
```

源
码 6.26.1

未完待续

第 7 章 多媒体

第8章 富文本及图表

第 9 章 控件

第 10 章 模型视图

Qt Widgets 中的模型视图非常难用。

在 Qt Widgets 中的模型视图架构中,每一个单独的项不是一个单独的 QWidget。而是通过 QPainter 进行绘制,并通过一些函数响应鼠标键盘事件。

这就造成了如果读者想实现一些复杂的动态效果就不得不自己实现一个超级复杂的状态机系统或者将每一个数据项用诸如 QListWidgetItem 进行包装,这样会浪费大量内存。

而在 Qt Quick 中一切可视元素都是一致的。

视图是 QQuickItem 的子类,视图中的每一项也是 QQuickItem 的子类。在 Qt Quick 体系中一切都是对象!

读者可以使用 Qt Quick 的模型视图架构,将数据与渲染和控制完全分离。而且,这种实现是极其高效的,即使模型拥有高达数亿元素。Qt Quick 的模型视图架构也可以快速布局。最终,只有当前可见部分被渲染,而当前不可见部分除了布局信息之外,不耗费任何额外资源。

10.1 自定义表模型

10.1.1 使用 QtCreator 快速创建模型

无论是对于初学者还是老手,通过继承自 QAbstractItemModel 创建自己的模型都是一件麻烦的事。

所幸的是 QtCreator 可以帮助我们快速的创建出我们所需的模型的架构。

1. 如图 10.1.1 所示。在项目名称单击右键,选择“Add New...”。
2. 如图 10.1.2 所示。选择“Qt Item Model”模板。

图 10.1.1

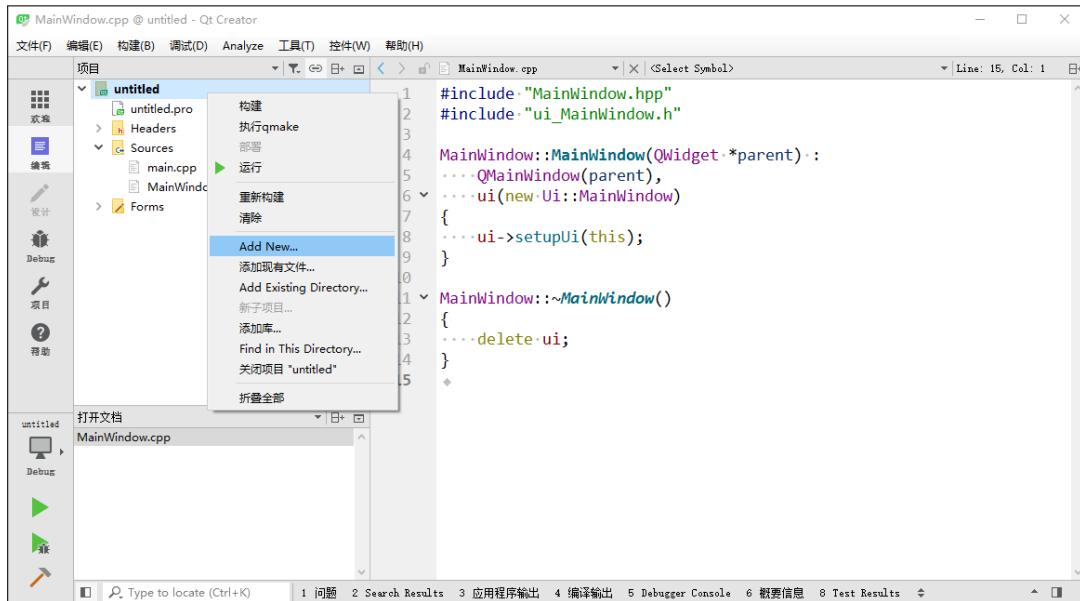


图 10.1.1：使用 QtCreator 创建模型

图 10.1.2

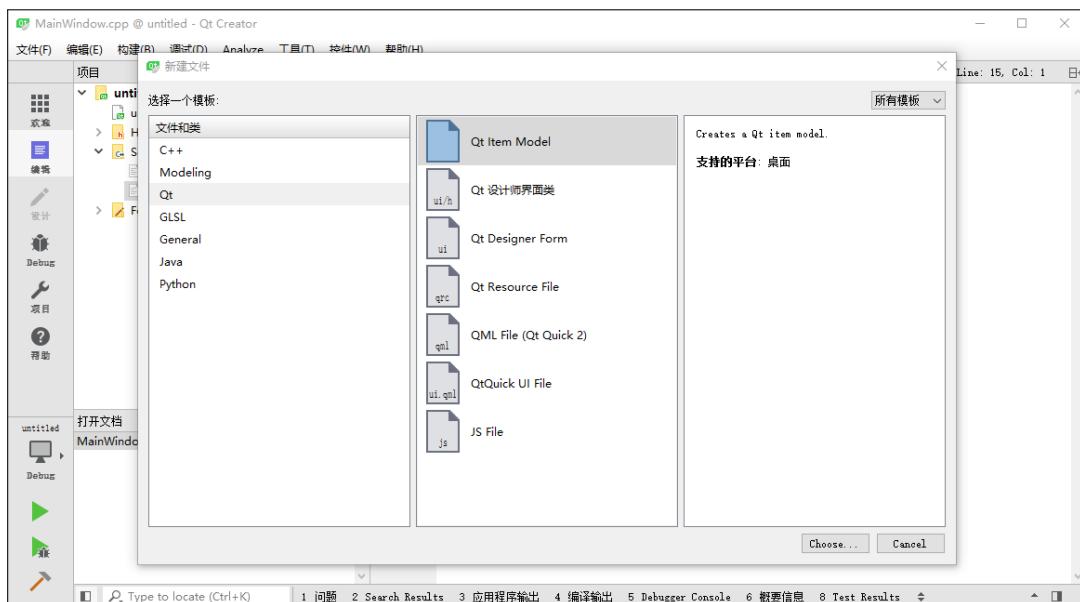


图 10.1.2：使用 QtCreator 创建模型

3. 如图 10.1.3 所示。输入类名，并选择所需功能。如果没有选择任何功能，则创建一个只读的模型。

- | | |
|-----------------------------------|----------|
| • Customize header row | 自定义标题 |
| • Items are editable | 数据元素可编辑 |
| • Rows and columns can be added | 可以添加行或列 |
| • Rows and columns can be removed | 可以删除行或列 |
| • Fetch data dynamically | 下拉获得更多数据 |

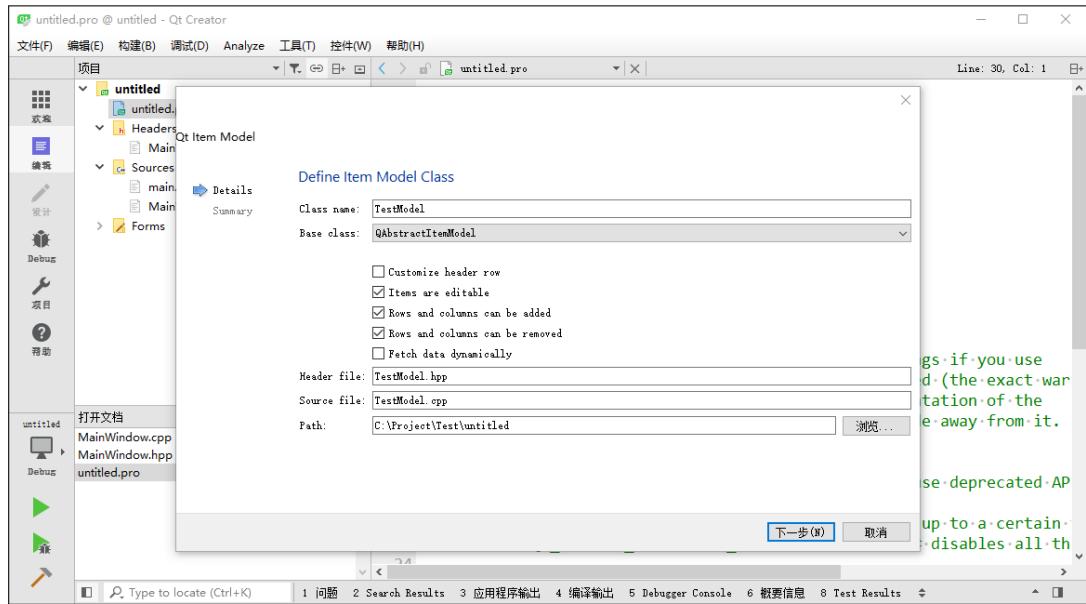


图 10.1.3

图 10.1.3：使用 QtCreator 创建模型

10.2 自定义树模型

附录

图片索引

图 1.1.1 , 3

图 1.1.2 , 4

图 1.2.1 , 12

图 1.2.2 , 19

图 1.2.3 , 20

图 1.2.4 , 20

图 1.4.1 , 28

图 1.5.1 , 30

图 1.6.1 , 33

图 1.6.2 , 34

图 1.7.1 , 36

图 1.7.2 , 37

图 2.1.1 , 42

图 6.1.1 , 51

图 6.2.1 , 55

图 6.3.1 , 60

图 6.4.1 , 62

图 6.5.1 , 63

图 6.6.1 , 64

图 6.7.1 , 65

图 6.8.1 , 67

图 6.9.1 , 68

图 6.10.1 , 69

图 6.11.1 , 70

图 6.12.1 , 72

图 6.13.1 , 73

图 6.14.1 , 74

图 6.15.1 , 76

图 6.16.1 , 77

图 6.17.1 , 78

图 6.18.1 , 79

图 6.19.1 , 81

图 6.20.1 , 82

图 6.21.1 , 83

图 6.22.1 , 84

图 6.23.1 , 86

图 6.24.1 , 87

图 6.25.1 , 88

图 6.26.1 , 90

图 10.1.1 , 100

图 10.1.2 , 100

图 10.1.3 , 101

表格索引

表 2.1.1 , 43

表 6.1.1 , 53

表 6.2.1 , 53

表 6.2.2 , 54

表 6.3.1 , 59

表 6.4.1 , 61

源码索引

源码 1.1.1 , 5

源码 1.1.2 , 5

源码 1.2.1 , 7

源码 1.2.2 , 7

源码 1.2.3 , 8

源码 1.2.4 , 9

源码 1.2.5 , 9

源码 1.2.6 , 10

源码 1.2.7 , 10

源码 1.2.8 , 11

源码 1.2.9 , 11

源码 1.2.10 , 11

源码 1.2.11 , 12

源码 1.2.12 , 13

源码 1.2.13 , 14

源码 1.2.14 , 15

源码 1.2.15 , 16

源码 1.2.16 , 16

源码 1.2.17 , 17

源码 1.2.18 , 17

源码 1.2.19 , 18

源码 1.2.20 , 19

源码 1.3.1 , 21

源码 1.3.2 , 21

源码 1.3.3 , 23

源码 1.3.4 , 23

源码 1.3.5 , 23

源码 1.3.6 , 24

源码 1.3.7 , 24

源码 1.3.8 , 24

源码 1.3.9 , 25

源码 1.3.10 , 25

源码 1.3.11 , 26

源码 1.3.12 , 26

源码 1.3.13 , 27

源码 1.4.1 , 27

源码 1.4.2 , 29

源码 1.4.3 , 29

源码 1.4.4 , 29

源码 1.4.5 , 30

源码 1.5.1 , 31

源码 1.5.2 , 32	源码 6.8.1 , 66	源码 6.25.1 , 88
源码 1.5.3 , 32	源码 6.9.1 , 68	源码 6.26.1 , 89
源码 1.6.1 , 32	源码 6.10.1 , 69	
源码 1.6.2 , 34	源码 6.11.1 , 70	命令索引
源码 1.7.1 , 36	源码 6.12.1 , 71	
源码 1.7.2 , 38	源码 6.13.1 , 73	命令 1.1.1 , 4
源码 1.7.3 , 38	源码 6.14.1 , 74	命令 1.1.2 , 5
源码 6.1.1 , 52	源码 6.15.1 , 75	命令 1.1.3 , 6
源码 6.2.1 , 54	源码 6.16.1 , 77	命令 1.1.4 , 6
源码 6.2.2 , 56	源码 6.17.1 , 78	命令 1.1.5 , 6
源码 6.3.1 , 59	源码 6.18.1 , 79	命令 1.2.1 , 18
源码 6.3.2 , 60	源码 6.19.1 , 80	
源码 6.4.1 , 61	源码 6.20.1 , 82	路径索引
源码 6.4.2 , 61	源码 6.21.1 , 83	
源码 6.5.1 , 63	源码 6.22.1 , 84	路径 1.2.1 , 9
源码 6.6.1 , 64	源码 6.23.1 , 85	路径 1.2.2 , 12
源码 6.7.1 , 65	源码 6.24.1 , 87	路径 1.3.1 , 20