# QSettings 介绍

# 简介

QSettings类提供了持久的跨平台应用程序设置。

用户通常期望应用程序记住它的设置（窗口大小、位置等）所有会话。这些信息通常存储在Windows系统注册表，OS X和iOS的属性列表文件中。在Unix系统中，在缺乏标准的情况下，许多应用程序(包括KDE应用程序)使用INI文本文件。

QSettings围绕这些抽象技术，使我们能够以便携的方式保存和恢复应用程序设置。它还支持自定义存储格式。

QSettings API基于QVariant，可以保存很多基础的类型，比如 QString、QRect、QImage等。

如果你需要的是一个非持久性的基于内存结构，可以考虑使用QMap<QString, QVariant>代替。

# 基本用法

当创建一个QSettings对象时，必须通过指定公司或组织名称以及产品名称，例如：公司名称为：MySoft，产品名为：Star Runner，那么可以用下列方式来构造QSettings对象：

QSettings settings("MySoft", "Star Runner");

QSettings对象既可以创建在栈上，也可以创建在堆（即使用new）上，构建和销毁也非常快。

如果你的应用程序在很多地方使用QSettings，则可以使用QCoreApplication::setOrganizationName() 和 QCoreApplication::setApplicationName()来指定组织名和应用名，然后使用默认的QSettings构造函数：

QCoreApplication::setOrganizationName("MySoft");

QCoreApplication::setOrganizationDomain("mysoft.com");

QCoreApplication::setApplicationName("Star Runner");

...

QSettings settings;

QSettings可以存储一系列设置。每个设置包括指定设置名称（键）的一个字符串和一个与该键关联的QVariant存储数据。使用setValue()可以实现一个设置。例如：

settings.setValue("editor/wrapMargin", 68);

如果存在相同的设置键，现有的值将被新值覆盖。为了提高效率，这些变化可能不会被立即保存到永久存储（可以随时调用sync()来提交更改）。

可以使用value()得到一个设置的值：

int margin = settings.value("editor/wrapMargin").toInt();

如果没有指定键对应的设置，QSettings将会返回一个空QVariant(可转换为整数0)。这时，我们可以通过另一个参数来指定默认值：

int margin = settings.value("editor/wrapMargin", 80).toInt();

* contains() 判断一个指定的键是否存在
* remove() 删除相关的键
* allKeys() 获取所有键
* clear() 删除所有键

void QSettings::beginGroup(const QString & prefix)

为当前组附加前缀。

当前组会自动追加到指定QSettings所有键。此外，查询功能，如childGroups()、childKeys()、allKeys() 也是基于组的。默认情况下，不存在组设置。

组是有用的,以避免输入同样的设置路径。例如：

settings.beginGroup("mainwindow")*;*

settings.setValue("size", win->size())*;*

settings.setValue("fullScreen", win->isFullScreen())*;*

settings.endGroup()*;*

settings.beginGroup("outputpanel")*;*

settings.setValue("visible", panel->isVisible())*;*

settings.endGroup()*;*

这将生成三个设置值:

* mainwindow/size
* mainwindow/fullScreen
* outputpanel/visible

int QSettings::beginReadArray(const QString & prefix)

为当前组添加前缀，并开始从数组中读取。返回数组的大小。   
例：

struct Login {

QString userName;

QString password;

};

QList<Login> logins;

...

QSettings settings;

int size = settings.beginReadArray("logins");

for (int i = 0; i < size; ++i) {

settings.setArrayIndex(i);

Login login;

login.userName = settings.value("userName").toString();

login.password = settings.value("password").toString();

logins.append(login);

}

settings.endArray();

void QSettings::beginWriteArray(const QString & prefix, int size = -1)

为当前组添加前缀，并开始写大小为size的数组。如果大小为-1（默认值），系统会自动根据索引的数目确定。

如果有许多出现一定的键集，可以使用数组实现更容易。例如，假设想要保存的用户名和密码的长度可变的列表。然后，你可以写：

struct Login {

QString userName;

QString password;

};

QList<Login> logins;

...

QSettings settings;

settings.beginWriteArray("logins");

for (int i = 0; i < logins.size(); ++i) {

settings.setArrayIndex(i);

settings.setValue("userName", list.at(i).userName);

settings.setValue("password", list.at(i).password);

}

settings.endArray();

生成的结果如下：

* logins/size
* logins/1/userName
* logins/1/password
* logins/2/userName
* logins/2/password
* logins/3/userName
* logins/3/password
* …

enum QSettings::Format

这个枚举类型指定QSettings所使用的存储格式。

| **常量** | **值** | **描述** |
| --- | --- | --- |
| QSettings::NativeFormat | 0 | 使用平台最合适的存储格式设置。在Windows中，使用系统注册表；OS X和iOS中，使用的是CFPreferences API；在Unix中，使用的是INI格式的文本配置文件。 |
| QSettings::IniFormat | 1 | 存储在INI文件中的设置。 |
| QSettings::InvalidFormat | 16 | registerFormat()返回的值 |

Unix中，NativeFor​​mat和IniFormat意思是一样的，只是文件扩展名不同（NativeFor​​mat为.conf，IniFormat 为.ini）。

enum QSettings::Scope

该枚举指定设置是否用户特定或同一系统的所有用户共享。

| **常量** | **值** | **描述** |
| --- | --- | --- |
| QSettings::UserScope | 0 | 在一个位置存储特定于当前用户的设置（例如，用户的主目录）。 |
| QSettings::SystemScope | 1 | 在一个全局位置存储设置，以便在相同机器上所有用户访问同一组的设置。 |

void QSettings::setPath(Format format, Scope scope, const QString & path)

为给定格式和范围设置用来存储的路径。对于路径而言，该格式可以是自定义格式。

下表总结了默认值：

| **平台** | **格式** | **范围** | **路径** |
| --- | --- | --- | --- |
| Windows | IniFormat | 1.UserScope 2.SystemScope | 1.%APPDATA% 2.%COMMON\_APPDATA% |
| Unix | NativeFormat, IniFormat | 1.UserScope 2.SystemScope | 1.$HOME/.config 2./etc/xdg |
| Qt for Embedded Linux | NativeFormat, IniFormat | 1.UserScope 2.SystemScope | 1.$HOME/Settings 2./etc/xdg |
| OS X and iOS | IniFormat | 1.UserScope 2.SystemScope | 1.$HOME/.config 2./etc/xdg |

在Windows、OS X、iOS中设置NativeFormat没有任何效果。

警告：此功能不会影响现有QSettings对象。

# QVariant和GUI类型

因为QVariant是Qt Core模块的一部分，它不能提供转换功能到数据类型-例如：QColor、QImage、 QPixmap，因为这是Qt GUI的一部分。换句话说，QVariant中没有toColor()、toImage()、toPixmap()等接口。

相反，可以使用QVariant::value()或qVariantValue()模板函数。 例如：

QSettings settings("MySoft", "Star Runner");

QColor color = settings.value("DataPump/bgcolor").value<QColor>();

逆转换（例如，从QColor到QVariant）是自动通过QVariant支持的所有数据类型，包括GUI相关类型：

QSettings settings("MySoft", "Star Runner");

QColor color = palette().background().color();

settings.setValue("DataPump/bgcolor", color);

自定义类型注册使用qRegisterMetaType()和qRegisterMetaTypeStreamOperators()可以使用QSettings存储。

# 重点说明

设置中的键可以包含任何Unicode字符。Windows注册表和INI文件使用对键不区分大小写，而在OS X和iOS的CFPreferences API使用区分大小写。为了避免可移植性问题，需要遵循这些简单的规则：

1. 在相同情况下使用相同的键。例如：代码中的一个位置使用”text fonts”，不要在别的地方使用”Text Fonts”。
2. 避免键名相同除了这种情况，例如：有一个名为”MainWindow”的键，不要试图用”mainwindow”保存另一个键。
3. 不要使用斜线（’/’和’\’）作为键名，反斜杠字符用于分隔子键（见下文）。在Windows中，’\’被QSettings转换为’/’，这使得它们相同。

可以使用’ / ‘字符作为分隔符形成分层键，类似于Unix文件路径。例如：

settings.setValue("mainwindow/size", win->size());

settings.setValue("mainwindow/fullScreen", win->isFullScreen());

settings.setValue("outputpanel/visible", panel->isVisible());

如果想保存或恢复具有相同前缀的一些设置，可以使用beginGroup()来指前缀，结束时调用endGroup()。下面和上面的例子相同，但这时使用组机制：

settings.beginGroup("mainwindow")*;*

settings.setValue("size", win->size())*;*

settings.setValue("fullScreen", win->isFullScreen())*;*

settings.endGroup()*;*

settings.beginGroup("outputpanel")*;*

settings.setValue("visible", panel->isVisible())*;*

settings.endGroup()*;*

如果一个组使用beginGroup()设置，大多数功能的行为变化，组可以递归地设置。

# 后备机制

假设你用组织名MySoft、应用名Star Runner创建了一个QSettings对象，当查看一个值时，依次搜索四个地方：

1. 一个供Star Runner应用程序的用户特定位置
2. 一个供MySoft所有应用程序的用户特定位置
3. 一个供Star Runner应用程序的系统范围位置
4. 一个供MySoft所有应用程序的系统范围位置

如果一个键不能在第一位置被找到时，继续在第二的位置搜索，依此类推。这确保能够存储系统范围或组织范围内的设置，并在每个用户或每个应用程序覆盖它们，使用setFallbacksEnabled(false)可以关闭这一机制。

虽然可以读取来自所有四个位置的键，仅第一个文件（用户特定的应用程序）用于写入访问。要写入任何其他文件，省去了程序名和指定QSettings:: SystemScope（相对于QSettings:: UserScope，默认值）。

用一个例子看看：

QSettings obj1("MySoft", "Star Runner");

QSettings obj2("MySoft");

QSettings obj3(QSettings::SystemScope, "MySoft", "Star Runner");

QSettings obj4(QSettings::SystemScope, "MySoft");

下表总结了QSettings对象访问的位置。”X”表示该位置相关联的QSettings对象的主要位置，既可以读取也可以写入，”O”是指读操作时被占用当做后备。

| **Locations** | **obj1** | **obj2** | **obj3** | **obj4** |
| --- | --- | --- | --- | --- |
| User, Application | X |  |  |  |
| User, Organization | o | X |  |  |
| System, Application | o |  | X |  |
| System, Organization | o | o | o | X |

这种机制的优点在于它可以在Qt支持的所有平台运行，它仍然给我们一个很大的灵活性，无需指定任何文件名或注册表路径。

如果想在所有平台上不使用原生API来使用 INI文件，可以通过 QSettings::IniFormat格式作为QSettings构造函数的第一个参数，其次是范围，组织名，以及应用名：

QSettings settings(QSettings::IniFormat, QSettings::UserScope, "MySoft", "Star Runner");

可以参考：Settings Editor例子（可以体验不同的设置-回退、打开、关闭）。

# 存储GUI程序状态

QSettings通常用于存储GUI程序的状态。下面的例子演示了如何使用QSettings保存和恢复应用程序的主窗口的几何形状。

void MainWindow::writeSettings()

{

QSettings settings("Moose Soft", "Clipper")*;*

settings.beginGroup("MainWindow")*;*

settings.setValue("size", size())*;*

settings.setValue("pos", pos())*;*

settings.endGroup()*;*

}

void MainWindow::readSettings()

{

QSettings settings("Moose Soft", "Clipper")*;*

settings.beginGroup("MainWindow")*;*

resize(settings.value("size", QSize(400, 400)).toSize())*;*

move(settings.value("pos", QPoint(200, 200)).toPoint())*;*

settings.endGroup()*;*

}

可以参考： Window Geometry。为什么调用QWidget::resize()和QWidget::move()更好，而不是QWidget::setGeometry()来恢复窗口的几何形状。

必须在MainWindow构造函数和关闭事件处理程序中调用readSettings()和writeSettings()函数，如下:

MainWindow::MainWindow()

{

...

readSettings();

}

void MainWindow::closeEvent(QCloseEvent \*event)

{

if (userReallyWantsToQuit()) {

writeSettings();

event->accept();

} else {

event->ignore();

}

}

可以参考自带的Application程序示例使用QSettings例子。

# 同时从多个线程或进程访问QSettings

QSettings是可重入的，意味着可以同时在不同的线程中使用不同的QSettings对象，这保证QSettings对象操作同一磁盘上的文件（或在系统注册表中的相同条目）。如果通过QSettings对象修改了一个设置，那么对于操作在同一位置和存在相同的进程的其他QSettings对象来说，更改会立即可见。

QSettings可以由不同的进程（其可以是应用程序同时运行的不同实例或完全不同的应用程序）安全地使用-在相同的系统位置上进行读写，它使用劝告式文件锁和智能合并算法以确保数据的完整性，需要注意的是sync()由其他进程所做的更改。

# 特定平台

## 应用程序设置的存储位置

如上所提到的，在后背机制部分，QSettings为应用程序存储的设置多达四个位置，这取决于设置是否是特定于用户或系统范围的，设置是否特定于应用或组织范围的。为简单起见，我们假设该组织被称为MySoft并且应用程序被称为Star Runner。

在Unix系统中，如果文件格式是NativeFormat，默认使用以下文件：

1. $HOME/.config/MySoft/Star Runner.conf (Qt for Embedded Linux: $HOME/Settings/MySoft/Star Runner.conf)
2. $HOME/.config/MySoft.conf (Qt for Embedded Linux: $HOME/Settings/MySoft.conf)
3. /etc/xdg/MySoft/Star Runner.conf
4. /etc/xdg/MySoft.conf

在Mac OS X版本10.2和10.3中，这些文件所使用的默认值：

1. $HOME/Library/Preferences/com.MySoft.Star Runner.plist
2. $HOME/Library/Preferences/com.MySoft.plist
3. /Library/Preferences/com.MySoft.Star Runner.plist
4. /Library/Preferences/com.MySoft.plist

在Windows上，NativeFormat设置存储在注册表路径如下：

1. HKEY\_CURRENT\_USER\Software\MySoft\Star Runner
2. HKEY\_CURRENT\_USER\Software\MySoft\OrganizationDefaults
3. HKEY\_LOCAL\_MACHINE\Software\MySoft\Star Runner
4. HKEY\_LOCAL\_MACHINE\Software\MySoft\OrganizationDefaults

如果文件格式是IniFormat,，以下文件用于在Unix、Mac OS X，、和iOS：

1. $HOME/.config/MySoft/Star Runner.ini (Qt for Embedded Linux: $HOME/Settings/MySoft/Star Runner.ini)
2. $HOME/.config/MySoft.ini (Qt for Embedded Linux: $HOME/Settings/MySoft.ini)
3. /etc/xdg/MySoft/Star Runner.ini
4. /etc/xdg/MySoft.ini

在Windows中，使用以下文件：

1. %APPDATA%\MySoft\Star Runner.ini
2. %APPDATA%\MySoft.ini
3. %COMMON\_APPDATA%\MySoft\Star Runner.ini
4. %COMMON\_APPDATA%\MySoft.ini

%APPDATA%路径通常为：C:\Documents and Settings\All Users\Application Data   
%COMMONAPPDATA%路径通常为：C:\Documents and Settings\All Users\Application Data。

在黑莓手机只有一个文件。如果文件格式是IniFormat，这时”Settings/MySoft/Star Runner.ini”在应用程序的主目录。

对于在.ini和conf文件的路径，可以使用的setPath()来改变。在Unix、Mac OS X、iOS中，用户可以通过设置XDG\_CONFIG\_HOME环境变量替代他们。

## 访问INI和.plist文件

有时候，想在一个特定的文件或注册表路径中访问存储设置。在所有平台上，如果想直接读取INI文件，可以使QSettings构造函数的第一个参数为文件名，第二个参数为QSettings::IniFormat。例如：

QSettings settings("/home/petra/misc/myapp.ini", QSettings::IniFormat);

然后，就可以对文件进行读写设置。   
在OS X和iOS中，可以通过指定第二个参数为QSettings::NativeFormat访问属性列表的.plist文件。例如：

QSettings settings("/Users/petra/misc/myapp.plist", QSettings::NativeFormat);

## 访问Windows注册表

在Windows中，QSettings可以在系统注册表访问由QSettings写入的设置（或设置支持的格式，如字符串数据）。通过使用一个注册表路径和QSettings::NativeFormat来构建一个QSettings对象。例如：

QSettings settings("HKEY\_CURRENT\_USER\\Software\\Microsoft\\Office", QSettings::NativeFormat);

所有出现在指定的路径下的注册表条目，可以通过QSettings对象像往常一样进行读写(使用斜杠而不是反斜杠)。例如：

settings.setValue("11.0/Outlook/Security/DontTrustInstalledFiles", 0);

注意，反斜线字符，如前所述，使用QSettings分割为子项。这样一来，不能读写包含斜线或反斜线Windows注册表项，如果需要的话，应该使用Windows API。

## 访问Windows上常见的注册表设置

在Windows上，有可能存在一个键既有值又存在子键。其默认值是通过使用”Default”或”.” 来代替子键。

settings.setValue("HKEY\_CURRENT\_USER\\MySoft\\Star Runner\\Galaxy", "Milkyway");

settings.setValue("HKEY\_CURRENT\_USER\\MySoft\\Star Runner\\Galaxy\\Sun", "OurStar");

settings.value("HKEY\_CURRENT\_USER\\MySoft\\Star Runner\\Galaxy\\Default"); // returns "Milkyway"

# 平台限制

尽管QSettings试图支持​​不同平台，但还存在着一些差异，我们应该知道，当移植应用程序：

Windows系统注册表有以下限制：一个子项不能超过255个字符，一个条目的值不得超过16,383个字符，一个键的所有值不得超过65,535个字符。要解决这些局限性的一种方法是使用IniFormat代替NativeFor​​mat存储设置。

在OS X和iOS中，allKeys()将返回适用于所有应用程序的全局设置一些额外的键。这些键可以使用value()读取，但不能改变，只能跟踪。调用setFallbacksEnabled(false) 将隐藏这些全局设置。

在OS X和iOS，使用QSettings的CFPreferences API预计互联网域名而不是组织名。为了提供一个统一的API，QSettings源于该组织名提供一个假域​​名（除非组织名已经是一个域名，如：OpenOffice.org）。该算法追加”.com”到公司名，并用连字符替换空格和其他非法字符。如果你想指定不同的域名，在main()函数中调用QCoreApplication::setOrganizationDomain()、QCoreApplication::setOrganizationName()、QCoreApplication::setApplicationName()，然后使用默认的QSettings构造函数。   
另一种解决方案是使用预处理指令，例如：

#ifdef Q\_OS\_MAC

QSettings settings("grenoullelogique.fr", "Squash");

#else

QSettings settings("Grenoulle Logique", "Squash");

#endif