QtCore

# Qt属性系统(Qt’s Property System)

Qt提供了一个绝妙的属性系统。跟那些由编译器提供的属性差不多。然而，作为一个独立于编译器和平台的库，Qt不依赖于非标准的编译特性，比如\_\_property 或[property]。Qt可以在任何平台上的标准编译器下编译。Qt属性系统基于元数据对象系统－－就是那个提供了对象内置信号和槽通讯机制的家伙。

## 声明属性需要什么

要声明一个属性，需在继承自QObject的类中使用Q\_PROPERTY()宏。

Q\_PROPERTY(type name

(READ getFunction [WRITE setFunction] |

MEMBER memberName [(READ getFunction | WRITE setFunction)])

[RESET resetFunction]

[NOTIFY notifySignal]

[REVISION int]

[DESIGNABLE bool]

[SCRIPTABLE bool]

[STORED bool]

[USER bool]

[CONSTANT]

[FINAL])

下面是一些典型的声明属性的示例：

Q\_PROPERTY(bool focus READ hasFocus)

Q\_PROPERTY(bool enabled READ isEnabled WRITE setEnabled)

Q\_PROPERTY(QCursor cursor READ cursor WRITE setCursor RESET unsetCursor)

下面示例展示了如何使用MEMBER关键字将成员变量作为Qt属性导出。 请注意，必须指定一个NOTIFY信号来允许QML属性绑定。

Q\_PROPERTY(QColor color MEMBER m\_color NOTIFY colorChanged)

Q\_PROPERTY(qreal spacing MEMBER m\_spacing NOTIFY spacingChanged)

Q\_PROPERTY(QString text MEMBER m\_text NOTIFY textChanged)

...

signals:

void colorChanged();

void spacingChanged();

void textChanged(const QString &newText);

private:

QColor m\_color;

qreal m\_spacing;

QString m\_text;

一个属性的行为就像类的数据成员，但是它还具有附加的特性，这些特性可以被元数据对象系统操作。这些特性是：

* 如果没有指定MEMBER变量，那么必须需要一个READ访问器函数。用于读属性的值。理想情况下，有一个常量函数用于此目的，并且它必须返回属性的类型的值或常量引用。例如，QWidget::focus是一个只读的属性，它对应一个读函数：QWidget::hasFocus()。
* 一个可选的WRITE访问器函数。它用于设置属性的值。它必须返回空并且至少具有一个参数，参数是属性类型的值或指针或引用。例如：QWidget::enabled具有WRITE函数QWidget::setEnable()。只读属性不需要写函数。例如，QWidget::focus没有对应的写函数。
* 如果没有指定READ访问器函数，那么必须需要一个MEMBER变量。这使得给定的member变量可读和可写，而不需要创建READ和WRITE访问器函数。如果你需要控制变量访问，除了MEMBER变量关联之外(但不能同时使用)，仍然可以使用READ和WERITE函数。
* 一个可选的RESET函数。用于设置属性的值到它的默认值。例如：QWidget::cursor具有典型的READ和WRITE函数，QWidget::cursor()和QWidget::setCursor()，并且它也具有一个RESET函数，QWidget::unsetCursor()。RESET函数必须返回void并且不带有任何参数。
* 一个可选的NOTIFY信号。如果被定义了，信号将在属性的值改变时发出。信号必须带有零个或一个参数，这个参数的类型必须与属性相同；参数保存的是属性的新值。NOTIFY信号只能在属性发生变化时才发出，以避免绑定在QML中被不必要地重新评估。Qt在需要MEMBER属性时自动发出这个信号，这个属性没有一个明确的setter。
* REVISION是可选的。如果包含该变量，它将定义该属性及其notify信号被用于API的特定修订（通常用于QML）。如果不包含，默认值为0。
* 一个DESIGNABLE变量表明此属性是否在界面设计器的属性编辑器中出现。大多数属性是可见的，除了为这个变量传入true或false，你还可以指定一个bool型的成员函数。
* SCRIPTABLE变量表明这个属性是否可以被一个脚本引擎操作（默认是true）。你也可以赋予它true或false或bool型函数。
* STORED变量表明了属性是否被认为是独立存在还是依赖于其它的值而存在。它也表明是否在保存对象状态时保存此属性的值。大多数属性都是需要保存的，但是，如QWidget::minimumWidth()就是不被保存的，因为它的值是从另一个属性QWidget::minimumSize()得来的。
* USER变量表明属性是否被设计为面向用户的或用户可修改的类属性。通常，每个类只有一个USER属性（默认值是false）。例如，QAbstractButton::checked是按钮类的用户可修改属性。注意QItemDelegate获取和设置widget的USER属性。
* CONSTANT的出现表明属性的值是不变的。对于一个object实例，常量属性的READ方法在每次被调用时必须返回相同的值。此常量值可能在不同的object实例中不相同。一个常量属性不能具有WRITE方法或NOYIFY信号。
* FINAL变量的出现表明属性不能被派生类所重写。有些情况下，这可以用于效率优化，但不是被moc强制的。程序员必须永远注意不能重写一个FINAL属性。

READ，WRITE和RESET函数都可以被继承。它们也可以是虚函数。当它们在被多重继承中被继承时，它们必须出现在第一个被继承的类中。

属性的类型可以是被QVariant支持的所有类型，也可以是用户定义的类型。在下面的例子中，类QDate被当作用户自定义类型。

Q\_PROPERTY(QDate data READ getDate WRITE setDate)

因为QDate是用户定义的，你必须包含<QDate>头文件。

对于QMap,QList和QValueList属性，属性的值是一个QVariant，它包含整个list或map。注意Q\_PROPERTY字符串不能包含逗号，因为逗号会划分宏的参数。因此，你必须使用QMap作为属性的类型而不是QMap<QString,QVariant>。为了保持一致性，也需要用QList和QValueList而不是QList<QVariant>和QValueList<QVariant>。

## 通过元数据对象系统读写属性

一个属性可以使用常规函数QObject::property()和QObject::setProperty()进行读写，不用知道属性所在类的任何细节，除了属性的名字。在下面的小代码片段中，调用QAbstractButton::setDown()和QObject::setProperty()都把属性设置为“down”。

QPushButton \*button = new QPushButton;

QObject \*object = button;

button->setDown(true);

object->setProperty("down", true);

通过WRITE操作器来操作一个属性是上面两者中更好的，因为它快并且在编译时给于更好的诊断帮助，但是以这种方式设置属性要求你必须在编译时了解其类。通过名字来操作属性使你可以操作在编译器你不了解的类。你可以在运行时通过查询它的QObject,QMetaObject和QMetaProperties发现一个类的属性集，。

QObject \*object = ...

const QMetaObject \*metaobject = object->metaObject();

int count = metaobject->propertyCount();

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

QMetaProperty metaproperty = metaobject->property(i);

const char \*name = metaproperty.name();

QVariant value = object->property(name);

...

}

在上面的代码片段中，QMetaObject::property()被用于获取未知类中的属性的metadata。从metadata中获取属性名然后传给QObject::property()来获取当前对象的值。

## 一个简单例子

假设我们有一个类MyClass，它从QObject派生并且在它的private区使用了Q\_OBJECT宏。我们想在MyClass类中声明一个属性来持续追踪一个Priorty值。属性的值叫做priority，并且它的类型是一个在类MyClass中定义的叫做Priority的枚举。

我们在类的private区使用Q\_PROPERTY()来声明属性。READ函数叫做priority,并且我们包含一个WRITE函数叫做setPriority。枚举类型必须使用Q\_ENUMS()注册到元数据对象系统中。注册一个枚举类型使得枚举的名字可以在调用QObject::setProperty()时使用。我们还必须为READ和WRITE函数提供我们自己的声明。MyClass的声明看起来应该是这样的：

class MyClass : public QObject

{

Q\_OBJECT

Q\_PROPERTY(Priority priority READ priority WRITE setPriority NOTIFY priorityChanged)

public:

MyClass(QObject \*parent = 0);

~MyClass();

enum Priority { High, Low, VeryHigh, VeryLow };

Q\_ENUM(Priority)

void setPriority(Priority priority)

{

m\_priority = priority;

emit priorityChanged(priority);

}

Priority priority() const

{ return m\_priority; }

signals:

void priorityChanged(Priority);

private:

Priority m\_priority;

};

READ函数是const的并且返回属性的类型。WRITE函数返回void并且具有一个属性类型的参数。元数据对象编译器强制做这些事情。

在有了一个指向MyClass实例的指针或一个指向表示MyClass实例的QObject的指针时，我们有两种方法来设置priority属性：

MyClass \*myinstance = new MyClass;

QObject \*object = myinstance;

myinstance->setPriority(MyClass::VeryHigh);

object->setProperty("priority", "VeryHigh");

在此例子中，枚举类型在MyClass中声明并被使用Q\_ENUMS()注册到元数据对象系统中。这使得枚举值可以在调用setProperty()时做为字符串使用。如果枚举类型是在其它类中声明的，那么我们就需要用枚举的全名（如OtherClass::Priority)，并且这个其它类也必须从QObject中派生并且也要注册枚举类型。

另一个简单的Q\_FLAGS()也是可用的。就像Q\_ENUMS()，它注册一个枚举类型，但是它把枚举类型作为一个flag的集合，也就是，值可以用OR操作来合并。一个I/O类可能具有枚举值Read和Write并且QObject::setProperty()可以接受 Read|Write。此时应使用Q\_FLAGS()来注册枚举值。

## 动态属性

Qobject::setProperty()也可以用来在运行时向一个类的实例添加新的属性。当使用一个名字和值调用它时，如果一个对应的属性已经存在，并且如果值的类型与属性的类型兼容，那么值就被存储到属性中，然后返回true。如果值类型不兼容，属性的值就不会发生改变，就会返回false。但是如果对应名字的属性不存在，那么一个新的属性就诞生了，以传入的名字为名，以传入的值为值，但是依然会返回false。这表示返回值不能用于确定一个属性是否被设置值，除非你已经知道这个属性已经存在于QObject中了。

注意动态属性被添加到单个实现的基础中，也就是，被添加到QObject，而不是QMetaObject。一个属性可以从一个实例中删除，通过传入属性的名字和非法的QVariant值给QObject::setProperty()。默认的QVariant构造器构造一个非法的QVariant。

动态属性可用QObject::property()来查询，就像使用Q\_PROPERTY()声明的属性一样。

## 属性和自定义类型

被属性使用的自定义类型需要使用Q\_DECLARE\_METATYPE()宏注册，以使它们的值能被保存在QVariant对象中。这使得它们可以用于被Q\_PROPERTY()声明的静态类型中，也可以被用于动态类型中。

## 将附加信息添加到类

连接到属性系统的是一个额外的宏Q\_CLASSINFO（），它可以用来附加名称 - 值对到类的元对象，例如：

[Q\_CLASSINFO](qthelp://org.qt-project.qtcore.550/qtcore/qobject.html#Q_CLASSINFO)("Version", "3.0.0")

像其他的元数据一样，在运行时可以通过元对象访问到类信息。