**设计Qt风格的C++API**

作者Matthias Ettrich，译者Googol Lee

在奇趣（Trolltech），为了改进Qt的开发体验，我们做了大量的研究。这篇文章里，我打算分享一些我们的发现，以及一些我们在设计Qt4时用到的原则，并且展示如何把这些原则应用到你的代码里。

好的API的六个特性

便利陷阱

布尔参数陷阱

静态多态

命名的艺术

指针还是引用？

例子：QProgressBar

如何把API设计好

设计应用程序接口，API，是很难的。这是一门和设计语言同样难的艺术。这里可以选择太多的原则，甚至有很多原则和其他原则有矛盾。

现在，计算机科学教育把很大的力气放在算法和数据结构上，而很少关注设计语言和框架背后的原则。这让应用程序员完全没有准备去面对越来越重要的任务：创造可重用的组件。

在面向对象语言普及之前，可重用的通用代码大部分是由库提供者写的，而不是应用程序员。在Qt的世界里，这种状况有了明显的改善。在任何时候，用Qt编程就是写新的组件。一个典型的Qt应用程序至少都会有几个在程序中反复使用的自定义组件。一般来说，同样的组件会成为其他应用程序的一部分。KDE，K桌面环境，走得更远，用许多追加的库来扩展Qt，实现了数百个附加类。（一般来说，一个类就是一个可重用组件，原文这里没有写清楚。）

但是，一个好的，高效的C++ API是由什么组成的呢？是好还是坏，取决于很多因素——比如，手头的工作和特定的目标群体。好的API有很多特性，一些特性是大家都想要的，而另一些则是针对特定问题域的。

好的API的六个特性

API是面向程序员的，用来描述提供给最终用户的GUI是什么样子。API中的P带表程序员（Programmer），而不是程序（Program），用来强调API是给程序员用的，给人类的程序员用的。

我们坚信API应该是最小化且完整的，拥有清晰且简单的语义，直觉化，容易记忆，并且引导人写出易读的代码。

最小化：最小化的API是指一个类尽可能只拥有最少的公开成员且尽可能只拥有最少的类。这个原则可以让API更简单易懂，更好记，更容易除错，且更容易改变。

完整的：完整的API是指要提供所有期望的功能。这个可能与最小化原则相冲突。另外，如果一个成员函数属于一个不应该属于的类，很多潜在的使用者都会找不到这个函数。

拥有清晰且简单的语义：就像其他设计工作一样，你必须遵守最小惊奇原则（the principle of least surprise）。让常见的任务简单易行。不常见的工作可行，但不会让用户过分关注。解决特殊问题时，不要让解决方案没有必要的过度通用。（比如，Qt3中的QMimeSourceFactory可以通过调用QImageLoader来实现不同的API。）

直觉化：就像电脑上的其他东西一样，API必须是直觉化的。不同的经验和背景会导致在判断什么是直觉而什么不是时不同的感觉。如果一个中级用户不读文档就可以使用（a semi-experienced user gets away without reading the documentation，没懂这里的get away该怎么翻译），并且一个程序员不懂API就可以理解缩写的代码，这种API就是直觉化的。

易于记忆：让API易于记忆，使用统一且精确的命名方法。使用可识别的模式和概念，并且避免缩写。

引导易读的代码（Lead to readable code）：代码一经写就，会读（并且除错和修改）多次。易读的代码可能会花点时间来写，但是可以节省产品周期中的其他时间。

最后，记住，不同类型的用户会用到API的不同部分。虽然简单的实例化一个Qt类是非常直觉化的，让资深专家在试图子类化之前读一遍文档，是很合理的。

便利陷阱

这是个常见的误解：更好的API，用更少的代码完成一件事。永远记住代码一次写就，之后需要不断的阅读并理解。比如：

QSlider \*slider = new QSlider(12, 18, 3, 13, Qt::Vertical,

0, "volume");

远比下面那样难读（甚至难写）：

QSlider \*slider = new QSlider(Qt::Vertical);

slider->setRange(12, 18);

slider->setPageStep(3);

slider->setValue(13);

slider->setObjectName("volume");

布尔参数陷阱

布尔参数通常会导致不易读的代码。更进一步，给一个已经存在的函数加入一个布尔参数，这常常是个错误。在Qt里，一个传统的例子是repaint()，这个函数带有一个布尔参数，来标识是否擦除背景（默认擦除）。这让代码通常写成：

widget->repaint(false);

初学者很容易把这句话理解成“别重画”！

这样做是考虑到布尔参数可以减少一个函数，避免代码膨胀。事实上，这反而增加了代码量。有多少Qt用户真的记住了下面三行程序都是做什么的？

widget->repaint();

widget->repaint(true);

widget->repaint(false);

一个好一些的API可能看起来是这样：

widget->repaint();

widget->repaintWithoutErasing();

在Qt4里，我们重新设计了widget，使得用户不再需要不重画背景的重画widget，来解决这个问题。Qt4原生支持双缓存，废掉了这个特性。

这里还有一些例子：

widget->setSizePolicy(QSizePolicy::Fixed,

QSizePolicy::Expanding, true);

textEdit->insert("Where's Waldo?", true, true, false);

QRegExp rx("moc\_\*.c??", false, true);

一个显而易见的解决方法是，使用枚举类型代替布尔参数。这正是我们在Qt4中QString大小写敏感时的处理方法。比较：

str.replace("%USER%", user, false); // Qt 3

str.replace("%USER%", user, Qt::CaseInsensitive); // Qt 4

静态多态

相似的类应该含有相似的API。在必要的时候——就是说，需要使用运行时多态的时候——这可以通过继承实现。但是多态依旧会发生在设计时期。比如，如果你用QListBox代替QComboBox，或者用QSlider代替QSpinBox，你会发现相似的API使这种替换非常容易。这就是我们所说的“静态多态”。

静态多态也使API和程序模式更容易记忆。作为结论，一组相关类使用相似的API，有时要比给每个类提供完美的单独API，要好。

（译注：C++ 0x将要引入的concept，就是静态多态的语法层实现。这个要比单独的函数名相似更强大且易用。）

命名的艺术

命名，大概是设计API时唯一最重要的问题了。该怎么称呼这个类？成员函数该叫什么？

通用的命名规则

一些规则通常对所有名字都是有用的。首先，就像我之前提到的，别用缩写。甚至很明显的缩写，比如“prev”表示“previous”从长远看也是不划算的，因为用户必须记住哪些词是缩写。

如果API本身不一致，事情自然会变得很糟糕，比如， Qt3有activatePreviousWindow()和fetchPrev()。坚持“没有缩写”的规则更容易创建一致的API。

另一个重要但更加微妙的规则是，在设计类的时候，必须尽力保证子类命名空间的干净。在Qt3里，没有很好的遵守这个规则。比如，拿QToolButton来举例。如果你在Qt3里，对一个QToolButton调用name()、caption()、text()或者textLabel()，你希望做什么呢？你可以在Qt Designer里拿QToolButton试试：

name属性继承自QObject，表示一个对象用于除错和测试的内部名字。

caption属性继承自QWidget，表示窗口的标题，这个标题在视觉上对QToolButton没有任何意义，因为他们总是跟随父窗口而创建。

text属性继承自QButton，一般情况下是按钮上现实的文字，除非useTextLabel为真。

textLabel在QToolButton里声明，并且在useTextLabel为真时显示在按钮上。

由于对可读性的关注，name在Qt4里被称作objectName，caption变成了windowsTitle，而在QToolButton里不再有单独的textLabel属性。

给类命名

标识一组类而不是单独给每个类找个恰当的名字。比如，Qt4里所有模式感知项目的视图类（model-aware item view classes）都拥有-View的后缀（QListView、QTableView和QTreeView），并且对应基于项目的类都用后缀-Widget代替（QListWidget、QTableWidget和QTreeWidget）。

给枚举类型及其值命名

当声明枚举时，时刻记住，在C++（不像Java和C#）中，使用枚举值不需要类型信息。下面的例子演示了给枚举值起个太过常用的名字所引起的危害：

namespace Qt

{

enum Corner { TopLeft, BottomRight, ... };

enum CaseSensitivity { Insensitive, Sensitive };

...

};

tabWidget->setCornerWidget(widget, Qt::TopLeft);

str.indexOf("$(QTDIR)", Qt::Insensitive);

在最后一行，Insensitive是什么意思？一个用于命名枚举值的指导思想是，在每个枚举值里，至少重复一个枚举类型名中的元素：

namespace Qt

{

enum Corner { TopLeftCorner, BottomRightCorner, ... };

enum CaseSensitivity { CaseInsensitive,

CaseSensitive };

...

};

tabWidget->setCornerWidget(widget, Qt::TopLeftCorner);

str.indexOf("$(QTDIR)", Qt::CaseInsensitive);

当枚举值可以用“或”连接起来当作一个标志时，传统的做法是将“或”的结果作为一个int保存，这不是类型安全的。Qt4提供了一个模板类 QFlags<T>来实现类型安全，其中T是个枚举类型。为了方便使用，Qt为很多标志类名提供了typedef，所以你可以使用类型 Qt::Alignment代替QFlags<Qt::AlignmentFlag>。

为了方便，我们给枚举类型单数的名字（这样表示枚举值一次只能有一个标志），而“标志”则使用复数名字。比如：

enum RectangleEdge { LeftEdge, RightEdge, ... };

typedef QFlags<RectangleEdge> RectangleEdges;

有些情况下，“标志“类使用了单数的名字。这时，枚举类使用-Flag做后缀：

enum AlignmentFlag { AlignLeft, AlignTop, ... };

typedef QFlags<AlignmentFlag> Alignment;

（这里为啥不是把”标志“类用-Flag做后缀，而是把枚举值做后缀呢？感觉有点混淆……）

给函数和参数命名

给函数命名的一个规则是，名字要明确体现出这个函数是否有副作用。在Qt3，常数函数QString::simplifyWhiteSpace()违反了这个原则，因为它返回类一个QString实例，而不是像名字所提示的那样，更改了调用这个函数的实例本身。在Qt4，这个函数被重命名为QString::simplified()。

参数名是程序员的重要信息来源，虽然在使用API时，并不直接展示在代码里。由于现代IDE在程序员写代码时可以自动显示参数名（就是自动感知或者自动补全之类的功能），值得花时间给头文件里声明的参数一个合适的名字，并且在文档中也使用相同的名字。

给布尔值设置函数（Setter）、提取函数（Getter）和属性命名

给布尔属性的设置函数和提取函数一个合适的名字，总是非常痛苦的。提取函数应该叫做checked()还是isChecked()？scrollBarsEnabled()还是areScrollBarEnabled()?

在Qt4里，我们使用下列规则命名提取函数：

形容类的属性使用is-前缀。比如：

isChecked()

isDown()

isEmpty()

isMovingEnable()

另外，应用到复数名词的形容类属性没有前缀：

scrollBarsEnabled()，而不是areScrollBarsEnabled()

动词类的属性不使用前缀，且不使用第三人称（-s）：

acceptDrops()，而不是acceptsDrops()

allColumnsShowFocus()

名词类的属性，通常没有前缀：

autoCompletion()，而不是isAutoCompletion()

boundaryChecking()

有时，没有前缀就会引起误解，这种情况使用前缀is-：

isOpenGLAvailable()，而不是openGL()

isDialog()，而不是dialog()

（通过调用dialogue()方法，正常情况下会期望返回一个QDialog\*的实例。）

设置函数名字继承自提取函数名，只是移掉了所有前缀，并使用set-做前缀，比如：setDown()还有setScrollBarsEnabled()。属性的名字与提取函数相同，只是去掉了前缀。

指针还是引用？

传出参数的最佳选择是什么，指针还是引用？

void getHsv(int \*h, int \*s, int \*v) const

void getHsv(int &h, int &s, int &v) const

大部分C++书推荐在能用引用的地方就用引用，这是因为一般认为引用比指针更“安全且好用”。然而，在奇趣（Trolltech），我们倾向使用指针，因为这让代码更易读。比较：

color.getHsv(&h, &s, &v);

color.getHsv(h, s, v);

只有第一行能清楚的说明，在函数调用后，h、s和v将有很大几率被改动。

例子：QProgressBar

为了展示如何实际应用这些概念，我们将学习Qt3中的API QProgressBar并和Qt4里相通的API做比较。在Qt3里：

class QProgressBar : public QWidget

{

...

public:

int totalSteps() const;

int progress() const;

const QString &progressString() const;

bool percentageVisible() const;

void setPercentageVisible(bool);

void setCenterIndicator(bool on);

bool centerIndicator() const;

void setIndicatorFollowsStyle(bool);

bool indicatorFollowsStyle() const;

public slots:

void reset();

virtual void setTotalSteps(int totalSteps);

virtual void setProgress(int progress);

void setProgress(int progress, int totalSteps);

protected:

virtual bool setIndicator(QString &progressStr,

int progress,

int totalSteps);

...

};

API相当复杂，且不统一。比如，仅从名字reset()并不能理解其作用，setTotalSteps()和setProgress()是紧耦合的。

改进API的关键，是注意到QProgressBar和Qt4的QAbstractSpinBox类及其子类QSpinBox，QSlider和QDial很相似。解决方法？用minimum、maximum和value代替progress和totalSteps。加入alueChanged()信号。加入setRange()函数。

之后观察progressString、percentage和indicator实际都指一个东西：在进度条上显示的文字。一般来说文字是百分比信息，但是也可以使用setIndicator()设为任意字符。下面是新的API：

virtual QString text() const;

void setTextVisible(bool visible);

bool isTextVisible() const;

默认的文字信息是百分比信息。文字信息可以藉由重新实现text()而改变。

在Qt3 API中，setCenterIndicator()和setIndicatorFollowStyle()是两个影响对齐的函数。他们可以方便的由一个函数实现，setAlignment()：

void setAlignment(Qt::Alignment alignment);

如果程序员不调用setAlignment()，对齐方式基于当前的风格。对于基于Motif的风格，文字将居中显示；对其他风格，文字将靠在右边。

这是改进后的QProgressBar API：

class QProgressBar : public QWidget

{

...

public:

void setMinimum(int minimum);

int minimum() const;

void setMaximum(int maximum);

int maximum() const;

void setRange(int minimum, int maximum);

int value() const;

virtual QString text() const;

void setTextVisible(bool visible);

bool isTextVisible() const;

Qt::Alignment alignment() const;

void setAlignment(Qt::Alignment alignment);

public slots:

void reset();

void setValue(int value);

signals:

void valueChanged(int value);

...

};

如何把API设计好（原文是How to Get APIs Right，我总想成We do APIs right……）

API需要质量保证。第一个修订版不可能是正确的；你必须做测试。写些用例：看看那些使用了这些API的代码，并验证代码是否易读。

其他的技巧包括让别的人分别在有文档和没有文档的情况下，使用这些API；或者为API类写文档（包括类的概述和独立的函数）。

当你卡住时，写文档也是一种获得好名字的方法：仅仅是尝试把条目（类，函数，枚举值，等等呢个）写下来并且使用你写的第一句话作为灵感。如果你不能找到一个精确的名字，这常常说明这个条目不应该存在。如果所有前面的事情都失败了并且你确认这个概念的存在，发明一个新名字。毕竟，“widget”、 “event”、“focus”和“buddy”这些名字就是这么来的。