原文链接：API Design Principles - Qt Wiki
基于Gary的影响力上 *Gary Gao* 的译文稿：C++的API设计指导
译文发在酷壳 - CoolShell：API设计原则， 2017-07-25

API设计原则 - Qt官网的设计实践总结

🍎 译序

Qt的设计水准在业界很有口碑，一致、易于掌握和强大的API是Qt最著名的优点之一。此文既是Qt官网上的API设计指导准则，也是Qt在API设计上的实践总结。虽然Qt用的是C++，但其中设计原则和思考是具有普适性的（如果你对C++还不精通，可以忽略与C++强相关或是过于细节的部分，仍然可以学习或梳理关于API设计最有价值的内容）。整个篇幅中有很多示例，是关于API设计一篇难得的好文章。

需要注意的是，这篇Wiki有一些内容并不完整，所以，可能会有一些阅读上的问题，我们对此做了一些相关的注释。

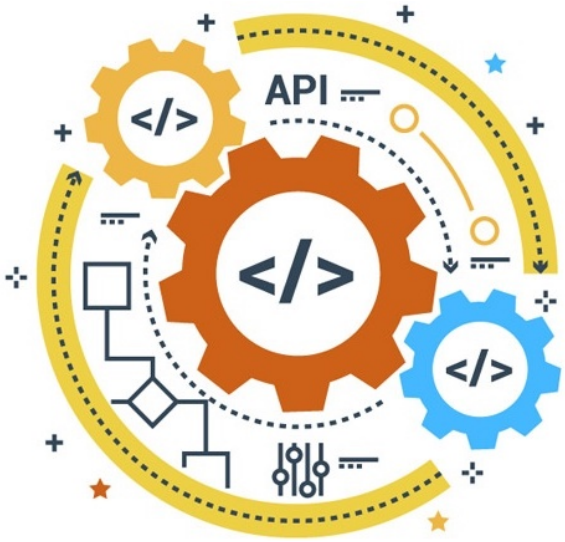
感谢酷壳博主 陈皓的全文审校，并对难点和要注意的地方给出贴心的说明和译注。

API设计原则

一致、易于掌握和强大的API是Qt最著名的优点之一。此文总结了我们在设计Qt风格API的过程中所积累的诀窍（know-how）。其中许多是通用准则；而其他的则更偏向于约定，遵循这些约定主要是为了与已有的API保持一致。

虽然这些准则主要用于对外的API（public API），但在设计对内的API（private API）时也推荐遵循相同的技巧（techniques），作为开发者之间协作的礼仪（courtesy）。

如有兴趣也可以读一下 *Jasmin Blanchette* 的Little Manual of API Design (PDF) 或是本文的前身 *Matthias Ettrich* 的Designing Qt-Style C++ APIs。



1. 好API的6个特质

API之于程序员就如同图形界面之于普通用户（end-user）。API中的『P』实际上指的是『程序员』（Programmer），而不是『程序』（Program），强调的是API是给程序员使用的这一事实。

在第13期Qt季刊，*Matthias* 的关于API设计的文章中提出了观点：API应该极简（minimal）且完备（complete）、语义清晰简单（have clear and simple semantics）、符合直觉（be intuitive）、易于记忆（be easy to memorize）和引导API使用者写出可读代码（lead to readable code）。

1.1 极简

极简的API是指每个class的public成员尽可能少，public的class也尽可能少。这样的API更易理解、记忆、调试和变更。

1.2 完备

完备的API是指期望有的功能都包含了。这点会和保持API极简有些冲突。如果一个成员函数放在错误的类中，那么这个函数的潜在用户就会找不到，这也是违反完备性的。

1.3 语义清晰简单

就像其他的设计一样，我们应该遵守最少意外原则（the principle of least surprise）。好的API应该可以让常见的事完成的更简单，并有可以完成不常见的事的可能性，但是却不会关注于那些不常见的事。解决的是具体问题；当没有需求时不要过度通用化解决方案。（举个例子，在Qt 3中，如果不是考虑到『解决具体问题』这点，QMimeSourceFactory就会命名成QImageLoader并有不一样的API。）

1.4 符合直觉

就像计算机里的其他事物一样，API应该符合直觉。对于什么是符合直觉的什么不符合，不同经验和背景的人会有不同的看法。API符合直觉的测试方法：经验不很丰富的用户不用阅读API文档就能搞懂API，而且程序员不用了解API就能看明白使用API的代码。

1.5 易于记忆

为使API易于记忆，API的命名约定应该具有一致性和精确性。使用易于识别的模式和概念，并且避免用缩写。

1.6 引导API使用者写出可读代码

代码只写一次，却要多次的阅读（还有调试和修改）。写出可读性好的代码有时候要花费更多的时间，但对于产品的整个生命周期来说是节省了时间的。

最后，要记住的是，不同的用户会使用API的不同部分。尽管简单使用单个Qt类的实例应该符合直觉，但如果是要继承一个类，让用户事先看好文档是个合理的要求。

2. 静态多态

相似的类应该有相似的API。在继承（inheritance）合适时可以用继承达到这个效果，即运行时多态。然而多态也发生在设计阶段。例如，如果你用QProgressBar替换QSlider，或是用QString替换QByteArray，你会发现API的相似性使的替换很容易。这即是所谓的『静态多态』（static polymorphism）。

静态多态也使记忆API和编程模式更加容易。因此，一组相关的类有相似的API有时候比每个类都有各自的一套API更好。

一般来说，在Qt中，如果没有足够的理由要使用继承，我们更倾向于用静态多态。这样可以减少Qt public类的个数，也使刚学习Qt的用户在翻看文档时更有方向感。

2.1 好的案例

QDialogButtonBox与QMessageBox，在处理按钮（addButton()、setStandardButtons()等等）上有相似的API，不需要继承某个QAbstractButtonBox类。

2.2 差的案例

QTcpSocket与QUdpSocket都继承了QAbstractSocket，这两个类的交互行为的模式（mode of interaction）非常不同。似乎没有什么人以通用和有意义的方式用过QAbstractSocket指针（或者 ***能*** 以通用和有意义的方式使用QAbstractSocket指针）。

2.3 值得斟酌的案例

QBoxLayout是QHBoxLayout与QVBoxLayout的父类。好处：可以在工具栏上使用QBoxLayout，调用setOrientation()使其变为水平/垂直。坏处：要多一个类，并且有可能导致用户写出这样没什么意义的代码，((QBoxLayout *)hbox)->setOrientation(Qt::Vertical)。

3. 基于属性的API

新的Qt类倾向于用『基于属性（property）的API』，例如：

```
QTimer timer;
timer.setInterval(1000);
timer.setSingleShot(true);
timer.start();
```

这里的 ***属性*** 是指任何的概念特征（conceptual attribute），是对象状态的一部分 —— 无论它是不是Q_PROPERTY。在说得通的情况下，用户应该可以以任何顺序设置属性，也就是说，属性之间应该是正交的（orthogonal）。例如，上面的代码可以写成：

```
QTimer timer;
timer.setSingleShot(true);
timer.setInterval(1000);
timer.start();
```

> 【译注】：正交性是指改变某个特性而不会影响到其他的特性。《程序员修炼之道》中讲了关于正交性的一个直升飞机坠毁的例子，讲得深入浅出很有画面感。

为了方便，也写成：

```
timer.start(1000);
```

类似地，对于QRegExp会是这样的代码：

```
QRegExp regExp;
regExp.setCaseSensitive(Qt::CaseInsensitive);
regExp.setPattern(".");
regExp.setPatternSyntax(Qt::WildcardSyntax);
```

为实现这种类型的API，需要借助底层对象的延迟创建。例如，对于QRegExp的例子，在不知道模式语法（pattern syntax）的情况下，在setPattern()中去解释"."就为时过早了。

属性之间常常有关联的；在这种情况下，我们必须小心处理。思考下面的问题：当前风格（style）提供的『默认的图标尺寸』属性 vs. QToolButton的『iconSize』属性：

```
toolButton->setStyle(otherStyle);
toolButton->iconSize();    // returns the default for otherStyle
toolButton->setIconSize(QSize(52, 52));
toolButton->iconSize();    // returns (52, 52)
toolButton->setStyle(yetAnotherStyle);
toolButton->iconSize();    // returns (52, 52)
```

提醒一下，一旦设置了iconSize，设置就会一直保持，即使改变当前风格。这 ***很好***。但有的时候需要能重置属性。有两种方法：

1. 传入一个特殊值（如QSize()、-1或者Qt::Alignment(0)）来表示『重置』
2. 提供一个明确的重置方法，如resetFoo()和unsetFoo()

对于iconSize，使用QSize()（比如 QSize(–1, -1)）来表示『重置』就够用了。

在某些情况下，getter方法返回的结果与所设置的值不同。例如，虽然调用了widget->setEnabled(true)，但如果它的父widget处于disabled状态，那么widget->isEnabled()仍然返回的是false。这样是OK的，因为一般来说就是我们想要的检查结果（父widget处于disabled状态，里面的子widget也应该变为灰的不响应用户操作，就好像子widget自身处于disabled状态一样；与此同时，因为子widget记得在自己的内心深处是enabled状态的，只是一直等待着它的父widget变为enabled）。当然诸如这些都必须在文档中妥善地说明清楚。

4. C++相关

4.1 值 vs. 对象

4.1.1 指针 vs. 引用

指针（pointer）还是引用（reference）哪个是最好的输出参数（out-parameters）？

```
void getHsv(int *h, int *s, int *v) const;
void getHsv(int &h, int &s, int &v) const;
```

大多数C++书籍推荐尽可能使用引用，基于一个普遍的观点：引用比指针『更加安全和优雅』。与此相反，我们在开发Qt时倾向于指针，因为指针让用户代码可读性更好。比较下面例子：

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

只有第一行代码清楚表达出h、s、v参数在函数调用中非常有可能会被修改。

这也就是说，编译器并不喜欢『出参』，所你应该在新的API中避免使用『出参』，而是返回一个结构体，如下所示：

```
struct Hsv { int hue, saturation, value };
Hsv getHsv() const;
```

> 【译注】：函数的『入参』和『出参』的混用会导致API接口语义的混乱，所以，使用指针，在调用的时候，实参需要加上“&”，这样在代码阅读的时候，可以看到是一个『出参』，有利于代码阅读。（但是这样做，在函数内就需要判断指针是否为空的情况，因为引用是不需要判断的，所以，这是一种 trade-off）
>
> 另外，如果这样的参数过多的话，最好使用一个结构体来把数据打包，一方面，为一组返回值取个名字，另一方面，这样有利用接口的简单。

4.1.2 按常量引用传参 vs. 按值传参

如果类型大于16字节，按常量引用传参。

如果类型有重型的（non-trivial）拷贝构造函数（copy-constructor）或是重型的析构函数（destructor），按常量引用传参以避免执行这些函数。

对于其它的类型通常应该按值传参。

示例：

```
void setAge(int age);
void setCategory(QChar cat);
void setName(QLatin1String name);

// const-ref is much faster than running copy-constructor and destructor
void setAlarm(const QSharedPointer<Alarm> &alarm);

// QDate, QTime, QPoint, QPointF, QSize, QSizeF, QRect
// are good examples of other classes you should pass by value.
```

> 【译注】：这是传引用和传值的差别了，因为传值会有对像拷贝，传引用则不会。所以，如果对像的构造比较重的话（换句话说，就是对像里的成员变量需要的内存比较大），这就会影响很多性能。所以，为了提高性能，最好是传引用。但是如果传入引用的话，会导致这个对象可能会被改变，所以传入const reference。

4.2 虚函数

在C++中，当类的成员函数声明为virtual，主要是为了通过在子类重载此函数能够定制函数的行为。将函数声明为virtual的目的是为了让对这个函数已有的调用变成执行实际实例的代码路径。对于没有在类外部调用的函数声明成virtual，你应该事先非常慎重地思考过。

```
// QTextEdit in Qt 3: member functions that have no reason for being virtual
virtual void resetFormat();
virtual void setUndoDepth( int d );
virtual void setFormat( QTextFormat *f, int flags );
virtual void ensureCursorVisible();
virtual void placeCursor( const QPoint &pos;, QTextCursor **c = 0 );
virtual void moveCursor( CursorAction action, bool select );
virtual void doKeyboardAction( KeyboardAction action );
virtual void removeSelectedText( int selNum = 0 );
virtual void removeSelection( int selNum = 0 );
virtual void setCurrentFont( const QFont &f );
virtual void setOverwriteMode( bool b ) { overWrite = b; }
```

QTextEdit从Qt 3移植到Qt 4的时候，几乎所有的虚函数都被移除了。有趣的是（但在预料之中），并没有人对此有大的抱怨，为什么？因为Qt 3没用到QTextEdit的多态行为 —— 只有你会；简单地说，没有理由去继承QTextEdit并重写这些函数，除非你自己调用了这些方法。如果在Qt在外部你的应用程序你需要多态，你可以自己添加多态。

> 【译注】：『多态』的目的只不过是为了实践 —— 『依赖于接口而不是实现』，也就是说，接口是代码抽像的一个非常重要的方式（在Java/Go中都有专门的接口声明语法）。所以，如果没有接口抽像，使用『多态』的意义也就不大了，因为也就没有必要使用『虚函数』了。

4.2.1 避免虚函数

在Qt中，我们有很多理由尽量减少虚函数的数量。每一次对虚函数的调用会在函数调用链路中插入一个未掌控的节点（某种程度上使结果更无法预测），使得bug修复变得更复杂。用户在重写的虚函数中可以做很多疯狂的事：

- 发送事件
- 发送信号
- 重新进入事件循环（例如，通过打开一个模态文件对话框）
- 删除对象（即触发『delete this』）

还有其他很多原因要避免过度使用虚函数：

- 添加、移动或是删除虚函数都带来二进制兼容问题（binary compatibility/BC）
- 重载虚函数并不容易
- 编译器几乎不能优化或内联（inline）对虚函数的调用

- 虚函数调用需要查找虚函数表（v-table），这比普通函数调用慢了2到3倍
- 虚函数使得类很难按值拷贝（尽管也可以按值拷贝，但是非常混乱并且不建议这样做）

经验告诉我们，没有虚函数的类一般bug更少、维护成本也更低。

一般的经验法则是，除非我们以这个类作为工具集提供而且有很多用户来调用某个类的虚函数，否则这个函数九成不应该设计成虚函数。

> 【译注】：
>
> 1. 使用虚函数时，你需要对编译器的内部行为非常清楚，否则，你会在使用虚函数时，觉得有好些『古怪』的问题发生。比如在创建数组对象的时候。
> 2. 在C++中，会有一个基础类，这个基础类中已经实现好了很多功能，然后把其中的一些函数放给子类去修改和实现。这种方法在父类和子类都是一组开发人员维护时没有什么问题，但是如果这是两组开发人员，这就会带来很多问题了，就像Qt这样，子类完全无法控制，全世界的开发人员想干什么就干什么。所以，子类的代码和父类的代码在兼容上就会出现很多很多问题。所以，还是上面所说，其实，虚函数应该声明在接口的语义里（这就是设计模式的两个宗旨——依赖于接口，而不是实现；钟爱于组合，而不是继承。也是为什么Java和Go语言使用interface关键字的原因，C++在多态的语义上非常容易滥用）

4.2.2 虚函数 vs. 拷贝

多态对象（polymorphic objects）和值类型的类（value-type classes）两者很难协作好。

包含虚函数的类必须把析构函数声明为虚函数，以防止父类析构时没有清理子类的数据，导致内存泄漏。

如果要使一个类能够拷贝、赋值或按值比较，往往需要拷贝构造函数、赋值操作符（operator =）和相等操作符（operator ==）。

```
class CopyClass {
public:
    CopyClass();
    CopyClass(const CopyClass &other);
    ~CopyClass();
    CopyClass &operator =(const CopyClass &other);
    bool operator ==(const CopyClass &other) const;
    bool operator !=(const CopyClass &other) const;

    virtual void setValue(int v);
};
```

如果继承CopyClass这个类，预料之外的事就已经在编码时酝酿了。一般情况下，如果没有虚成员函数和虚析构函数，就不能创建出可以多态的子类。然而，如果存在虚成员函数和虚析构函数，这突然变成了

要有子类去继承的理由，而且开始变得复杂了。***起初认为只要简单声明上虚操作符重载函数（virtual operators）。*** 但其实是走上了一条混乱和毁灭之路（破坏了代码的可读性）。看看下面的这个例子：

```
class OtherClass {
public:
    const CopyClass &instance() const; // 这个方法返回的是什么？可以赋值什么？
};
```

（这部份还未完成）

> 【译注】：因为原文上说，这部份并没有完成，所以，我也没有搞懂原文具体也是想表达什么。不过，就标题而言，原文是想说，在多态的情况下拷贝对象所带来的问题？？

4.3 关于const

C++的关键词const表明了内容不会改变或是没有副作用。可以应用于简单的值、指针及指针所指的内容，也可以作为一个特别的属性应用于类的成员函数上，表示成员函数不能修改对象的状态。

然而，const本身并没有提供太大的价值 —— 很多编程语言甚至没有类似const的关键词，但是却并没有因此产生问题。实际上，如果你不用函数重载，并在C++源代码用搜索并删除所有的const，几乎总能编译通过并且正常运行。尽量让使用的const保持实用有效，这点很重要。

让我们看一下在Qt的API设计中与const相关的场景。

4.3.1 输入参数：const指针

有输入指针参数的const成员函数，几乎总是const指针参数。

如果函数声明为const，意味着既没有副作用，也不会改变对象的可见状态。那为什么它需要一个没有const限定的输入参数呢？记住const类型的函数通常被其他const类型的函数调用，接收到的一般都是const指针（只要不主动const_cast，我们推荐尽量避免使用const_cast）

以前：

```
bool QWidget::isVisibleTo(QWidget *ancestor) const;
bool QWidget::isEnabledTo(QWidget *ancestor) const;
QPoint QWidget::mapFrom(QWidget *ancestor, const QPoint &pos) const;
```

QWidget声明了许多非const指针输入参数的const成员函数。注意，这些函数可以修改传入的参数，不能修改对象自己。使用这样的函数常常要借助const_cast转换。如果是const指针输入参数，就可以避免这样的转换了。

之后：

```
bool QWidget::isVisibleTo(const QWidget *ancestor) const;
bool QWidget::isEnabledTo(const QWidget *ancestor) const;
QPoint QWidget::mapFrom(const QWidget *ancestor, const QPoint &pos) const;
```

注意，我们在QGraphicsItem中对此做了修正，但是QWidget要等到Qt 5:

```
bool isVisibleTo(const QGraphicsItem *parent) const;
QPointF mapFromItem (const QGraphicsItem *item, const QPointF &point) const;
```

4.3.2 返回值：const值

调用函数返回的非引用类型的结果，称之为右值（R-value）。

非类（non-class）的右值总是无cv限定类型（cv-unqualified type）。虽然从语法上讲，加上const也可以，但是没什么意义，因为鉴于访问权限这些值是不能改变的。多数现代编译器在编译这样的代码时会提示警告信息。

> 【译注】：cv-qualified的类型（与cv-unqualified相反）是由const或者volatile或者volatile const限定的类型；详见cv (const and volatile) type qualifiers - C++语言参考。

当在类类型（class type）右值上添加const关键字，则禁止访问非const成员函数以及对成员的直接操作。

不加const则没有以上的限制，但几乎没有必要加上const，因为右值对象生存时间（life time）的结束一般在C++清理的时候（通俗的说，下一个分号地方），而对右值对象的修改随着右值对象的生存时间也一起结束了（也就是本条语句的执行完成的时候）。

示例：

```
struct Foo {
    void setValue(int v) { value = v; }
    int value;
};

Foo foo() {
    return Foo();
}

const Foo cfoo() {
    return Foo();
}

int main() {
    // The following does compile, foo() is non-const R-value which
    // can't be assigned to (this generally requires an L-value)
    // but member access leads to a L-value:
    foo().value = 1; // Ok, but temporary will be thrown away at the end of the full-exp

    // The following does compile, foo() is non-const R-value which
    // can't be assigned to, but calling (even non-const) member
    // function is fine:
    foo().setValue(1); // Ok, but temporary will be thrown away at the end of the full-e

    // The following does _not_compile, foo() is ''const'' R-value
    // with const member which member access can't be assigned to:
    cfoo().value = 1; // Not ok.

    // The following does _not_compile, foo() is ''const'' R-value,
    // one cannot call non-const member functions:
    cfoo().setValue(1); // Not ok
}
```

> 【译注】：上述的代码说明，如果返回值不是const的，代码可以顺利编译通过，然而并没有什么卵用，因为那个临时对像马上就被抛弃了。所以，这样的无用的代码最好还是在编译时报个错，以免当时头脑发热想错了，写了一段没用但还以为有用的代码。

4.3.3 返回值：非const的指针还是有const的指针

谈到const函数应该返回非const的指针还是const指针这个话题时，多数人发现在C++中关于『const正确性』（const correctness）在概念上产生了分歧。*问题起源是：**const函数本身不能修改对象自身的状态，却可以返回成员的非const指针**。返回指针这个简单动作本身既不会影响整个对象的可见状态，当然也不会改变这个函数职责范围内涉及的状态。但是，这却使得程序员可以间接访问并修改对象的状态。*

下面的例子演示了通过返回非const指针的const函数绕开const约定（constness）的诸多方式中的一种：

```
QVariant CustomWidget::inputMethodQuery(Qt::InputMethodQuery query) const {
    moveBy(10, 10); // doesn't compile!
    window()->childAt(mapTo(window(), rect().center()))->moveBy(10, 10); // compiles!
}
```

返回const指针的函数正是保护以避免这些（可能是不期望的/没有预料到的）副作用，至少是在一定程度上。但哪个函数你会觉得更想返回const指针，或是不止一个函数?

若采用const正确（const-correct）的方法，每个返回某个成员的指针（或多个指向成员的指针）的const函数必须返回const指针。在实践中，很不幸这样的做法将导致无法使用的API：

```
QGraphicsScene scene;
// … populate scene

foreach (const QGraphicsItem *item, scene.items()) {
    item->setPos(qrand() % 500, qrand() % 500); // doesn't compile! item is a const poir
}
```

QGraphicsScene::items()是一个const函数，顺着思考看起来这个函数只应该返回const指针。

在Qt中，我们几乎只有非const的使用模式。我们选择的是实用路子：
相比滥用非const指针返回类型带来的问题，返回const指针更可能招致过分使用const_cast的问题。

4.3.4 返回值：按值返回 还是 按const引用返回?

若返回的是对象的拷贝，那么返回const引用是更直接的方案；
然而，这样的做法限制了后面想要对这个类的重构（refactor）。
（以d-point的惯用法（idiom）为例，我们可以在任何时候改变Qt类在内存表示（memory representation）；但却不能在不破坏二进制兼容性的情况下把改变函数的签名，返回值从const QFoo &变为QFoo。）
基于这个原因，除去对运行速度敏感（speed is critical）而重构不是问题的个别情形（例如，QList::at()），我们一般返回QFoo而不是const QFoo &。

> 【译注】：参看《Effective C++》中条款23：Don't try to return a reference when you must return an object

4.4.5 const vs. 对象的状态

const正确性（const correctness）的问题就像C圈子中vi与emacs的讨论，因为这个话题在很多地方都存在分歧（比如基于指针的函数）。

但通用准则是const函数不能改变类的可见状态。『状态』的意思是『自身以及涉及的职责』。这并不是指非const函数能够改变自身的私有成员，也不是指const函数改变不了。而是指函数是活跃的并存在可见的副作用（visible side effects）。const函数一般没有任何可见的副作用，比如：

```
QSize size = widget->sizeHint(); // const
widget->move(10, 10); // not const
```

代理（delegate）负责在其它对象上绘制内容。
它的状态包括它的职责，因此包括在哪个对象做绘制这样的状态。
调用它的绘画行为必然会有副作用；
它改变了它绘制所在设备的外观（及其所关联的状态）。鉴于这些，paint()作为const函数并不合理。
进一步说，任何paint()或QIcon的paint()的视图函数是const函数也不合理。
没有人会从内部的const函数去调用QIcon::paint()，除非他想显式地绕开const这个特性。
如果是这种情况，使用const_cast会更好。

```
// QAbstractItemDelegate::paint is const
void QAbstractItemDelegate::paint(QPainter **painter, const QStyleOptionViewItem &optior

// QGraphicsItem::paint is not const
void QGraphicsItem::paint(QPainter *painter, const QStyleOptionGraphicsItem option, QWic
```

const关键字并不能按你期望的样子起作用。应该考虑将其移除而不是去重载const/非const函数。

5. API的语义和文档

当传值为-1的参数给函数，函数会是什么行为？有很多类似的问题……

是警告、致命错误还是其它？

API需要的是质量保证。API第一个版本一定是不对的；必须对其进行测试。
以阅读使用API的代码的方式编写用例，且验证这样代码是可读的。

还有其他的验证方法，比如

- 让别人使用API（看了文档或是先不看文档都可以）
- 给类写文档（包含类的概述和每个函数）

6. 命名的艺术

命名很可能是API设计中最重要的一个问题。类应该叫什么名字？成员函数应该叫什么名字？

6.1 通用的命名规则

有几个规则对于所有类型的命名都等同适用。第一个，之前已经提到过，不要使用缩写。即使是明显的缩写，比如把previous缩写成prev，从长远来看是回报是负的，因为用户必须要记住缩写词的含义。

如果API本身没有一致性，之后事情自然就会越来越糟；例如，Qt 3 中同时存在activatePreviousWindow()与fetchPrev()。恪守『不缩写』规则更容易地创建一致性的API。

另一个时重要但更微妙的准则是在设计类时应该保持子类名称空间的干净。在Qt 3中，此项准则并没有一直遵循。以QToolButton为例对此进行说明。如果调用QToolButton的 name()、caption()、text()或者textLabel()，你觉得会返回什么？用Qt设计器在QToolButton上自己先试试吧：

- name属性是继承自QObject，返回内部的对象名称，用于调试和测试。
- caption属性继承自QWidget，返回窗口标题，对QToolButton来说毫无意义，因为它在创建的时候parent就存在了。
- text函数继承自QButton，一般用于按钮。当useTextLabel不为true，才用这个属性。
- textLabel属性在QToolButton内声明，当useTextLabel为true时显示在按钮上。

为了可读性，在Qt 4中QToolButton的name属性改成了objectName，caption改成了windowTitle，删除了textLabel属性因为和text属性相同。

当你找不到好的命名时，写文档也是个很好方法：要做的就是尝试为各个条目（item）（如类、方法、枚举值等等）写文档，并用写下的第一句话作为启发。如果找不到一个确切的命名，往往说明这个条目是不该有的。如果所有尝试都失败了，并且你坚信这个概念是合理的，那么就发明一个新名字。像widget、event、focus和buddy这些命名就是在这一步诞生的。

> 【译注】：写文档是一个非常好的习惯。写文档的过程其实就是在帮你梳理你的编程思路。很多时候，文档写着写着你就会发现要去改代码去了。除了上述的好处多，写文档还有更多的好处。比如，在写文档的过程中，你发现文字描述过于复杂了，这表明着你的代码或逻辑是复杂的，这就倒逼你去重构你的代码。所以 —— **写文档其实就是写代码**。

6.2 类的命名

识别出类所在的分组，而不是为每个类都去找个完美的命名。例如，所有Qt 4的能感知模型（model-aware）的item view，类后缀都是View（QListView、QTableView、QTreeView），而相应的基于item（item-based）的类后缀是Widget（QListWidget、QTableWidget、QTreeWidget）。

6.3 枚举类型及其值的命名

声明枚举类型时，需要记住在C++中枚举值在使用时不会带上类型（与Java、C#不同）。下面的例子演示了枚举值命名得过于通用的危害：

```
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？命名枚举类型的一个准则是在枚举值中至少重复此枚举类型名中的一个元素：

```
namespace Qt {
    enum Corner { TopLeftCorner, BottomRightCorner, ... };
    enum CaseSensitivity { CaseInsensitive, CaseSensitive };
    ...
};

tabWidget->setCornerWidget(widget, Qt::TopLeftCorner);
str.indexOf("$(QTDIR)", Qt::CaseInsensitive);
```

当对枚举值进行或运算并作为某种标志（flag）时，传统的做法是把或运算的结果保存在int型的值中，但这不是类型安全的。Qt 4提供了一个模板类QFlags<T>，其中的T是枚举类型。为了方便使用，Qt用typedef重新定义了QFlag类型，所以可以用Qt::Alignment代替QFlags<Qt::AlignmentFlag>。

习惯上，枚举类型命名用单数形式（因为它一次只能『持有』一个flag），而持有多个『flag』的类型用复数形式，例如：

```
enum RectangleEdge { LeftEdge, RightEdge, ... };
typedef QFlags<RectangleEdge> RectangleEdges;
```

在某些情形下，持有多个『flag』的类型命名用单数形式。对于这种情况，持有的枚举类型名称要求是以Flag为后缀：

```
enum AlignmentFlag { AlignLeft, AlignTop, ... };
typedef QFlags<AlignmentFlag> Alignment;
```

6.4 函数和参数的命名

函数命名的第一准则是可以从函数名看出来此函数是否有副作用。在Qt 3中，const函数QString::simplifyWhiteSpace()违反了此准则，因为它返回了一个QString而不是按名称暗示的那样，改变调用它的QString对象。在Qt 4中，此函数重命名为QString::simplified()。

虽然参数名不会出现在使用API的代码中，但是它们给程序员提供了重要信息。因为现代的IDE都会在写代码时显示参数名称，所以值得在头文件中给参数起一个恰当的名字并在文档中使用相同的名字。

6.5 布尔类型的getter与setter方法的命名

为bool属性的getter和setter方法命名总是很痛苦。getter应该叫做checked()还是isChecked()？scrollBarsEnabled()还是areScrollBarEnabled()？

Qt 4中，我们套用以下准则为getter命名：

- 形容词以is为前缀，例子：
  - isChecked()
  - isDown()
  - isEmpty()
  - isMovingEnabled()
- 然而，修饰名词的形容词没有前缀：
  - scrollBarsEnabled()，而不是areScrollBarsEnabled()
- 动词没有前缀，也不使用第三人称(-s)：
  - acceptDrops()，而不是acceptsDrops()
  - allColumnsShowFocus()
- 名词一般没有前缀：
  - autoCompletion()，而不是isAutoCompletion()
  - boundaryChecking()
- 有的时候，没有前缀容易产生误导，这种情况下会加上is前缀：
  - isOpenGLAvailable()，而不是openGL()
  - isDialog()，而不是dialog()
    （一个叫做dialog()的函数，一般会被认为是返回QDialog。）

setter的名字由getter衍生，去掉了前缀后在前面加上了set；例如，setDown()与setScrollBarsEnabled()。

7. 避免常见陷阱

7.1 简化的陷阱

一个常见的误解是：实现需要写的代码越少，API就设计得越好。应该记住：代码只会写上几次，却要被反复阅读并理解。例如：

```
QSlider *slider = new QSlider(12, 18, 3, 13, Qt::Vertical, 0, "volume");
```

这段代码比下面的读起来要难得多（甚至写起来也更难）：

```
QSlider *slider = new QSlider(Qt::Vertical);
slider->setRange(12, 18);
slider->setPageStep(3);
slider->setValue(13);
slider->setObjectName("volume");
```

> 【译注】：在有IDE的自动提示的支持下，后者写起来非常方便，而前者还需要看相应的文档。

7.2 布尔参数的陷阱

布尔类型的参数总是带来无法阅读的代码。给现有的函数增加一个bool型的参数几乎永远是一种错误的行为。仍以Qt为例，repaint()有一个bool类型的可选参数用于指定背景是否被擦除。可以写出这样的代码：

```
widget->repaint(false);
```

初学者很可能是这样理解的，『不要重新绘制！』，能有多少Qt用户真心知道下面3行是什么意思：

```
widget->repaint();
widget->repaint(true);
widget->repaint(false);
```

更好的API设计应该是这样的：

```
widget->repaint();
widget->repaintWithoutErasing();
```

在Qt 4中，我们通过移除了重新绘制（repaint）而不擦除widget的能力来解决了此问题。Qt 4的双缓冲使这种特性被废弃。

还有更多的例子：

```
widget->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Expanding, true);
textEdit->insert("Where's Waldo?", true, true, false);
QRegExp rx("moc_***.c??", false, true);
```

一个明显的解决方案是bool类型改成枚举类型。我们在Qt 4的QString中就是这么做的。对比效果如下：

```
str.replace("%USER%", user, false);               // Qt 3
str.replace("%USER%", user, Qt::CaseInsensitive); // Qt 4
```

> 【译注】：关于这个条目可以看看CoolShell这篇文章一些展开的讨论： 千万不要把 BOOL 设计成函数参数。

8. 案例研究

8.1 QProgressBar

为了展示上文各种准则的实际应用。我们来研究一下Qt 3中QProgressBar的API，并与Qt 4中对应的API作比较。在Qt 3中：

```
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);
    ...
};
```

该API相当的复杂和不一致；例如，reset()、setTotalSteps()、setProgress()是紧密联系的，但方法的命名并没明确地表达出来。

改善此API的关键是抓住QProgressBar与Qt 4的QAbstractSpinBox及其子类QSpinBox、QSlider、QDail之间的相似性。怎么做？把progress、totalSteps替换为minimum、maximum和value。增加一个valueChanged()消息，再增加一个setRange()函数。

进一步可以观察到progressString、percentage与indicator其实是一回事，即是显示在进度条上的文本。通常这个文本是个百分比，但是可通过setIndicator()设置为任何内容。以下是新的API：

```
virtual QString text() const;
void setTextVisible(bool visible);
bool isTextVisible() const;
```

默认情况下，显示文本是百分比指示器（percentage indicator），通过重写text()方法来定制行为。

Qt 3的setCenterIndicator()与setIndicatorFollowsStyle()是两个影响对齐方式的函数。他们可被一个setAlignment()函数代替：

```
void setAlignment(Qt::Alignment alignment);
```

如果开发者未调用setAlignment()，那么对齐方式由风格决定。对于基于Motif的风格，文字内容在中间显示；对于其他风格，在右侧显示。

下面是改善后的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);
    ...
};
```

8.2 QAbstractPrintDialog & QAbstractPageSizeDialog

Qt 4.0有2个幽灵类QAbstractPrintDialog和QAbstractPageSizeDialog，作为QPrintDialog和QPageSizeDialog类的父类。这2个类完全没有用，因为Qt的API没有是QAbstractPrint-或是-PageSizeDialog指针作为参数并执行操作。通过篡改qdoc（Qt文档），我们虽然把这2个类隐藏起来了，却成了无用抽象类的典型案例。

这不是说，***好***的抽象是错的，QPrintDialog应该是需要有个工厂或是其它改变的机制 —— 证据就是它声明中的#ifdef QTOPIA_PRINTDIALOG。

8.3 QAbstractItemModel

关于模型/视图（model/view）问题的细节在相应的文档中已经说明得很好了，但作为一个重要的总结这里还需要强调一下：抽象类不应该仅是所有可能子类的并集（union）。这样『合并所有』的父类几乎不可能是一个好的方案。QAbstractItemModel就犯了这个错误 —— 它实际上就是个QTreeOfTablesModel，结果导致了错综复杂（complicated）的API，而这样的API要让 ***所有本来设计还不错的子类*** 去继承。

仅仅增加抽象是不会自动就把API变得更好的。

8.4 QLayoutIterator & QGLayoutIterator

在Qt 3，创建自定义的布局类需要同时继承QLayout和QGLayoutIterator（命名中的G是指Generic（通用））。QGLayoutIterator子类的实例指针会包装成QLayoutIterator，这样用户可以像和其它的迭代器（iterator）类一样的方式来使用。通过QLayoutIterator可以写出下面这样的代码：

```
QLayoutIterator it = layout()->iterator();
QLayoutItem **child;
while ((child = it.current()) != 0) {
    if (child->widget() == myWidget) {
        it.takeCurrent();
        return;
    }
    ++it;
}
```

在Qt 4，我们干掉了QGLayoutIterator类（以及用于盒子布局和格子布局的内部子类），转而是让QLayout的子类重写itemAt()、takeAt()和count()。

8.5 QImageSink

Qt 3有一整套类用来把完成增量加载的图片传递给一个动画 —— QImageSource/Sink/QASyncIO/QASyncImageIO。由于这些类之前只是用于启用动画的QLabel，完全过度设计了（overkill）。

从中得到的教训就是：对于那些未来可能的还不明朗的需求，不要过早地增加抽象设计。当需求真的出现时，比起一个复杂的系统，在简单的系统新增需求要容易得多。