搜索



# Qt创建者编码规则

**注意：**本文档正在编写中。

编码规则旨在指导Qt Creator开发人员，帮助他们编写可理解和可维护的代码，并最大限度地减少混乱和意外。

像往常一样，规则不是一成不变的。如果您有充分的理由打破一个，请这样做。但首先要确保至少有其他一些开发人员同意你的观点。

要为Qt Creator主源做出贡献，您应该遵守以下规则：

- 最重要的规则是：KISS（保持简短）。始终选择更简单的实现选项而不是更复杂的实现选项。这使得维护变得更加容易。
- 编写好的C++代码。也就是说，可读性强，必要时注释良好，面向对象。
- 充分利用Qt。不要重新发明轮子。想想你的代码的哪些部分足够通用，以至于它们可能会被合并到Qt而不是Qt Creator中。
- 使代码适应Qt Creator中的现有结构。如果您有改进想法，请在编写代码之前与其他开发人员讨论。
- 请遵循代码构造、格式设置和模式和实践中的准则。
- 文档界面。现在我们使用qdoc，但正在考虑改用doxygen。

## 提交代码

要向Qt Creator提交代码，您必须了解Qt开发背后的工具和机制以及哲学。有关如何设置开发环境以使用Qt Creator以及如何提交代码和文档以供收录的更多信息，请参阅Qt贡献指南。

## 二进制和源兼容性

以下列表描述了如何对版本进行编号，并定义了版本之间的*二进制*兼容性和*源代码兼容性*：

- Qt Creator 3.0.0是*主要*版本，Qt Creator 3.1.0是*次要版本*，Qt Creator 3.1.3是*补丁版本*。
- *向后二进制兼容性*意味着链接到库早期版本的代码仍然有效。
- *向前二进制兼容性*意味着链接到较新版本库的代码适用于较旧版本的库。
- *源代码兼容性*意味着代码无需修改即可编译。

我们目前不保证主要版本和次要版本之间的二进制或源代码兼容性。

但是，我们尝试在同一次要版本中保留补丁版本的向后二进制兼容性和向后源代码兼容性：

- 软 API 冻结：从次要版本的测试版发布后不久开始，我们开始在该次要版本中保持向后源代码兼容性，包括其补丁版

- 硬 API 冻结：从次要版本的候选版本开始，我们在次要版本中保持向后源代码兼容性，包括其补丁版本。我们也开始保持向后二进制兼容性，除了这不会反映在插件的 compatVersion 设置中。因此，针对候选版本编写的Qt Creator插件实际上不会在最终版本或更高版本中加载，即使库的二进制兼容性理论上允许这样做。请参阅下面有关插件规格的部分。
- 硬 ABI 冻结：从次要版本的最终版本开始，我们为此版本及其所有补丁版本保留向后源代码和二进制兼容性。

为了保持向后兼容性：

- 请勿添加或删除任何公共 API（例如全局函数、公共/受保护/私有成员函数）。
- 不要重新实现函数（甚至不要内联，也不要重新实现受保护或私有函数）。
- 查看二进制兼容性解决方法，了解保持二进制兼容性的方法。

有关二进制兼容性的详细信息，请参阅C++ 的二进制兼容性问题。

从插件元数据的角度来看，这意味着：

- 补丁版本中的Qt Creator插件将具有次要版本。例如，版本 3.1.2 中的插件将具有。`compatVersioncompatVersion="3.1.0"`
- 次要版本的预发布版本（包括候选版本）仍将具有自身，这意味着针对最终版本编译的插件将不会在预版本中加载。`compatVersion`
- Qt Creator插件开发人员可以通过设置相应的时间来决定他们的插件是否需要其他Qt Creator插件的某个补丁版本（或更高版本），或者他们是否适用于此次要版本的所有补丁版本。Qt项目提供的Qt Creator插件的默认设置是需要最新的补丁版本。`version`

例如，Qt Creator 3.1 beta（内部版本号3.0.82）中的插件将具有`Find`

```
<plugin name="Find" version="3.0.82" compatVersion="3.0.82">
  <dependencyList>
    <dependency name="Core" version="3.0.82"/>
    ....
```

Qt Creator 3.1.0 final中的插件将具有`Find`

```
<plugin name="Find" version="3.1.0" compatVersion="3.1.0">
  <dependencyList>
    <dependency name="Core" version="3.1.0"/>
    ....
```

Qt Creator 3.1.1补丁版本中的插件将具有3.1.1版本，将向后二进制兼容插件版本3.1.0（），并且需要与插件版本3.1.1二进制向后兼容的插件：`FindFindcompatVersion="3.1.0"CoreCore`

```
<plugin name="Find" version="3.1.1" compatVersion="3.1.0">
  <dependencyList>
    <dependency name="Core" version="3.1.1"/>
    ....
```

# 代码构造

```
++T;
--U;

-NOT-

T++;
U--;
```

- 尝试一遍又一遍地尽量减少对相同代码的评估。这特别针对循环：

```
Container::iterator end = large.end();
for (Container::iterator it = large.begin(); it != end; ++it) {
        ...;
}

-NOT-

for (Container::iterator it = large.begin();
     it != large.end(); ++it) {
        ...;
}
```

# 格式

## 大写标识符

在标识符中使用驼峰大小写。

将标识符中的第一个单词大写，如下所示：

- 类名以大写字母开头。
- 函数名称以小写字母开头。
- 变量名称以小写字母开头。
- 枚举名称和值以大写字母开头。无作用域枚举值包含枚举类型名称的某些部分。

## 空白

- 使用四个空格进行缩进，没有制表符。
- 在适当的情况下使用空行将语句组合在一起。
- 始终仅使用一个空行。

### 指针和引用

对于指针或引用，始终在星号 (*) 或与号 (&) 之前使用单个空格，但切勿在之后使用。尽可能避免使用 C 型石膏：

```
char *blockOfMemory = (char *)malloc(data.size());
char *blockOfMemory = reinterpret_cast<char *>(malloc(data.size()));
```

当然，在这种特殊情况下，使用可能是一个更好的选择。new

### 运算符名称和括号

运算符名称和括号之间不要使用空格。公式标记 （==） 是运算符名称的一部分，因此，空格使声明看起来像表达式：

```
operator==(type)

-NOT-

operator == (type)
```

### 函数名称和括号

不要在函数名称和括号之间使用空格：

```
void mangle()

-NOT-

void mangle ()
```

### 关键字

始终在关键字后和大括号之前使用单个空格：

```
if (foo) {
}

-NOT-

if(foo){
}
```

### 评论

通常，在“//”之后放置一个空格。要在多行批注中对齐文本，可以插入多个空格。

## 括号

作为基本规则，将左大括号放在与语句开头相同的行上：

```
if (codec) {
}

-NOT-
```

例外：函数实现和类声明始终在行首使用左大括号：

```
static void foo(int g)
{
    qDebug("foo: %i", g);
}

class Moo
{
};
```

当条件语句的主体包含多行时，以及单行语句有些复杂时，请使用大括号。否则，请省略它们：

```
if (address.isEmpty())
    return false;

for (int i = 0; i < 10; ++i)
    qDebug("%i", i);

-NOT-

if (address.isEmpty()) {
    return false;
}

for (int i = 0; i < 10; ++i) {
    qDebug("%i", i);
}
```

例外 1：如果父语句包含多行或换行，也使用大括号：

```
if (address.isEmpty()
        || !isValid()
        || !codec) {
    return false;
}
```

**注意：**这可以重写为：

```
if (address.isEmpty())
    return false;

if (!isValid())
    return false;
```

例外 2：在 if-then-else 块中也使用大括号，其中 if 代码或 else 代码涵盖几行：

```
if (address.isEmpty()) {
    --it;
} else {
    qDebug("%s", qPrintable(address));
    ++it;
}

-NOT-

if (address.isEmpty())
    --it;
else {
    qDebug("%s", qPrintable(address));
    ++it;
}
```

```
if (a) {
    if (b)
        ...
    else
        ...
}

-NOT-

if (a)
    if (b)
        ...
    else
        ...
```

当条件语句的主体为空时，请使用大括号：

```
while (a) {}

-NOT-

while (a);
```

## 括弧

使用括号对表达式进行分组：

```
if ((a && b) || c)
```

```
(a + b) & c

-NOT-

a + b & c
```

## 换行符

- 保持行短于 100 个字符。
- 如有必要，插入换行符。
- 逗号位于虚线的末尾。
- 操作员从新生产线的开头开始。

```
if (longExpression
    || otherLongExpression
    || otherOtherLongExpression) {
}

-NOT-

if (longExpression ||
    otherLongExpression ||
    otherOtherLongExpression) {
}
```

## 声明

- 对类的访问部分使用此顺序：公共、受保护、私有。公共部分对类的每个用户都很有趣。私有部分仅对类的实现者（您）感兴趣。
- 避免在类的声明文件中声明全局对象。如果对所有对象使用相同的变量，请使用静态成员。
- 使用代替。一些编译器将该差异混入符号名称中，并在结构声明后跟类定义时发出警告。为了避免从一个到另一个的持续更改，我们声明首选方式。`classstructclass`

### 声明变量

- 避免类类型的全局变量以排除初始化顺序问题。如果无法避免，请考虑使用。`Q_GLOBAL_STATIC`
- 将全局字符串文本声明为

```
const char aString[] = "Hello";
```

- 尽可能避免使用短名称（例如，a，rbarr，nughdeget）。仅对计数器和临时变量使用单字符变量名称，其中变量的用途很明显。
- 在单独的行上声明每个变量：

```
-NOT-

QString a = "Joe", b = "Foo";
```

> **注意：**正式调用由字符串文本构造的临时复制构造函数。因此，它可能比直接建造更昂贵。但是，允许编译器省略副本（即使这有副作用），现代编译器通常会这样做。考虑到这些相等的成本，Qt Creator代码倾向于使用"="习惯用法，因为它符合传统的C式初始化，它不会被误认为是函数声明，并且它减少了更多初始化中的嵌套参数级别。`QString a = "Joe"QString a("Joe")`

- 避免缩写：

```
int height;
int width;
char *nameOfThis;
char *nameOfThat;

-NOT-

int a, b;
char *c, *d;
```

- 等待声明变量，直到需要它。当同时完成初始化时，这一点尤其重要。

## 命名空间

- 将左大括号放在与关键字相同的行上。namespace
- 不要缩进内部的声明或定义。
- 可选，但如果命名空间跨多行，则建议使用：在右大括号后添加注释，重复命名空间。

```
namespace MyPlugin {

void someFunction() { ... }

    }  // namespace MyPlugin
```

- 作为例外，如果命名空间中只有一个类声明，则所有声明都可以放在一行上：

```
namespace MyPlugin { class MyClass; }
```

- 不要在头文件中使用 using 指令。
- 在定义类和函数时不要依赖 using 指令，而是在正确命名的声明性区域中定义它。
- 访问全局函数时不要依赖 using 指令。
- 在其他情况下，建议您使用 using-指令，因为它们可以帮助您避免使代码混乱。更喜欢将所有 using-指令放在文件顶部附近，毕竟包含。

```
...
#include <utils/filename.h>
...
using namespace Utils;

namespace Foo {
namespace Internal {

void SomeThing::bar()
{
    FilePath f;              // or Utils::FilePath f
    ...
}
...
} // namespace Internal      // or only // Internal
} // namespace Foo           // or only // Foo

-NOT-

[in foo.h]
...
using namespace Utils;       // Wrong: no using-directives in headers

class SomeThing
{
    ...
};

-NOT-

[in foo.cpp]
...
using namespace Utils;

#include "bar.h"             // Wrong: #include after using-directive

-NOT-

[in foo.cpp]
...
using namepace Foo;

void SomeThing::bar()        // Wrong if Something is in namespace Foo
{
    ...
}
```

# 模式和实践

## 命名空间

阅读Qt In Namespace，请记住，所有Qt Creator都是*命名空间感知*代码。

Qt Creator中的命名空间策略如下：

- 导出以供其他库或插件使用的库或插件的类/符号位于特定于该库/插件的命名空间中，例如MyPlugin

### 传递文件名

Qt Creator API需要可移植格式的文件名，也就是说，即使在Windows上，也要使用斜杠（/）而不是反斜杠（\）。要将文件名从用户传递到 API，请先使用QDir：：fromNativeSeparators进行转换。若要向用户显示文件名，请使用QDir：：toNativeSeparators 将其转换回本机格式。考虑使用 Utils：：FilePath：：fromUserInput（QString） 和 Utils：：FilePath：：toUserOutput（） 来完成这些任务。

比较文件名时使用 Utils：：FilePath，因为这会考虑区分大小写。还要确保比较干净的路径（QDir：：cleanPath（））。

## 要使用的类和不使用的类

Qt Creator代码的很大一部分处理与开发机器不同的设备上的数据。这些可能在路径分隔符、行尾、进程启动详细信息等方面有所不同。

但是，一些基本的Qt类假定Qt应用程序只关注与开发机器类似的机器。

因此，这些类不适合在Qt Creator中涉及非本地代码的部分使用。相反，Qt Creator的Utils库提供了替代品，导致以下规则：

- 将 Utils：：FilePath 用于语义上是文件或目录的任何QString，另请参阅传递文件名。
- 更喜欢使用 Utils：：FilePath，而不是使用QDir和QFileInfo。
- 更喜欢使用 Utils：：QtcProcess 而不是QProcess。
- 如果 Utils：：FilePath 或 Utils：：QtcProcess 功能不足以满足您的目的，最好增强它们而不是回退到QString或QProcess。
- 避免平台#ifdefs，除非本地执行的代码绝对需要它们，即使这样，也更喜欢 Utils：：HostInfo 而不是#ifdefs。

## 插件扩展点

插件扩展点是由一个插件提供的接口，由其他插件实现。然后，插件检索接口的所有实现并使用它们。也就是说，它们*扩展了*插件的功能。通常，接口的实现在插件初始化期间放入全局对象池中，插件在插件初始化结束时从对象池中检索它们。

例如，Find 插件提供了 FindFilter 接口供其他插件实现。使用 FindFilter 界面，可以添加其他搜索范围，这些范围显示在"**高级搜索**"对话框中。Find 插件从全局对象池中检索所有 FindFilter 实现，并将它们显示在对话框中。该插件将实际的搜索请求转发到正确的 FindFilter 实现，然后执行搜索。

## 使用全局对象池

您可以通过 ExtensionSystem：:P luginManager：：addObject（） 将对象添加到全局对象池中，并通过 ExtensionSystem：:P luginManager：：getObject（） 再次检索特定类型的对象。这应该主要用于插件扩展点的实现。

> **注意：**不要将单例放入池中，也不要从那里检索它。请改用单一实例模式。

## C++特点

- 优先于头卫。`#pragma once`
- 不要使用例外，除非您知道自己在做什么。
- 不要使用 RTTI（运行时类型信息;即 typeinfo 结构、dynamic_cast或 typeid 运算符，包括引发异常），除非您知道自己要做什么。
- 不要使用虚拟继承，除非您知道自己在做什么。
- 明智地使用模板，不仅仅是因为您可以。

  提示：使用编译自动测试可以查看测试服务器场中的所有编译器是否都支持C++功能。

- 所有代码仅是 ASCII（仅限 7 位字符，如果不确定，则运行）`man ascii`

- 对于字符串：使用 \nnn（其中 nnn 是您希望字符串使用的任何区域设置的八进制表示形式）或 \xnn（其中 nn 是十六进制）。例如：QStrings =QString：：fromUtf8（“\213\005”）；
- 对于文档中的变音符号或其他非 ASCII 字符，请使用 qdoc 命令或使用相关宏。例如：对于 ü。 `\unicode\uuml`

- 尽可能使用静态关键字而不是匿名命名空间。使用静态本地化到编译单元的名称保证具有内部链接。对于在匿名命名空间中声明的名称，不幸的是，C++标准要求外部链接（ISO/IEC 14882，7.1.1/6，或在gcc邮件列表中查看有关此内容的各种讨论）。

### 空指针

对空指针常量使用 nullptr。

```
void *p = nullptr;

-NOT-

void *p = NULL;

-NOT-

void *p = '\0';

-NOT-

void *p = 42 - 7 * 6;
```

**注意：**作为例外，导入的第三方代码以及与本机 API 接口的代码 （src/support/os_*） 可以使用 NULL 或 0。

## C++11 和 C++14 功能

代码应使用 Microsoft Visual Studio 2013、g++ 4.7 和 Clang 3.1 进行编译。

### 拉姆达斯

使用 lambda 时，请注意以下事项：

- 不必显式指定返回类型。如果您没有使用前面提到的编译器之一，请注意这是一个 C++14 功能，您可能需要在编译器中启用 C++14 支持。

```
[]() {
    Foo *foo = activeFoo();
    return foo ? foo->displayName() : QString();
});
```

- 如果使用 lambda 所在的类中的静态函数，则必须显式捕获。否则，它不会使用 g++ 4.7 及更早版本进行编译。 `this`

```
void Foo::something()
{
    ...
    [this]() { Foo::someStaticFunction(); }
```

```
-NOT-

void Foo::something()
{
    ...
    []() { Foo::someStaticFunction(); }
    ...
}
```

根据以下规则格式化 lambda：

- 将捕获列表、参数列表、返回类型和左大括号放在第一行，将正文缩进放在后面的行上，将右大括号放在新行上。

```
[]() -> bool {
    something();
    return isSomethingElse();
}

-NOT-

[]() -> bool { something();
somethingElse(); }
```

- 将封闭函数调用的右括号和分号放在与 lambda 的右大括号相同的行上。

```
foo([]() {
    something();
});
```

- 如果您在“if”语句中使用 lambda，请在新行上开始 lambda，以避免混淆 lambda 的左大括号和“if”语句的左大括号。

```
if (anyOf(fooList,
        [](Foo foo) {
            return foo.isGreat();
        }) {
    return;
}

-NOT-

if (anyOf(fooList, [](Foo foo) {
            return foo.isGreat();
        }) {
    return;
}
```

- 或者，如果适合，将 lambda 完全放在一行上。

```
foo([] { return true; });
```

```
    }
```

### auto关键词

（可选）可以在以下情况下使用 the关键字。如果有疑问，例如，如果使用可能会降低代码的可读性，请不要使用。请记住，代码的读取频率比编写代码的频率高得多。autoautoauto

- 当它避免在同一语句中重复类型时。

```
auto something = new MyCustomType;
auto keyEvent = static_cast<QKeyEvent *>(event);
auto myList = QStringList({ "FooThing",  "BarThing" });
```

- 分配迭代器类型时。

```
auto it = myList.const_iterator();
```

### 作用域枚举

可以在不需要将无作用域枚举隐式转换为 int 或附加作用域很有用的地方使用作用域枚举。

### 委派构造函数

如果多个构造函数使用基本相同的代码，请使用委托构造函数。

### 初始值设定项列表

使用初始值设定项列表初始化容器，例如：

```
const QVector<int> values = {1, 2, 3, 4, 5};
```

### 使用大括号初始化

如果使用带大括号的初始化，请遵循与圆括号相同的规则。例如：

```
class Values // the following code is quite useful for test fixtures
{
    float floatValue = 4; // prefer that for simple types
    QVector<int> values = {1, 2, 3, 4, integerValue}; // prefer that syntax for initializer lists
    SomeValues someValues{"One", 2, 3.4}; // not an initializer_list
    SomeValues &someValuesReference = someValues;
    ComplexType complexType{values, otherValues} // constructor call
}

object.setEntry({"SectionA", value, doubleValue}); // calls a constructor
object.setEntry({}); // calls default constructor
```

将非静态数据成员初始化用于简单的初始化，公共导出类除外。

### 默认和已删除的函数

考虑使用 and 来控制特殊功能。`=default=delete`

### 覆盖

建议在覆盖虚函数时使用关键字。不要在被覆盖的函数上使用虚拟。`override`

确保类对所有重写的函数或无函数一致地使用。`override`

### 空普特

所有编译器都支持，但对使用它没有达成共识。如有疑问，请询问模块的维护者是否更喜欢使用。`nullptrnullptr`

### 基于范围的 for 循环

您可以使用基于范围的 for 循环，但要注意虚假分离问题。如果 for 循环仅读取容器，并且容器是 const 还是非共享并不明显，则 use 确保容器不会不必要地分离。`std::cref()`

## 使用 QObject

- 请记住将Q_OBJECT宏添加到依赖于元对象系统的QObject子类中。元对象系统相关功能是信号和时隙的定义、使用等。另请参阅转换。`qobject_cast<>`
- 首选 Qt5 样式调用而不是 Qt4 样式调用。`connect()`
- 使用 Qt4 样式调用时，规范化连接语句中信号和槽的参数，以安全地使信号和时隙查找速度加快几个周期。可以使用 $QTDIR/util/normalize 来规范化现有代码。有关更多信息，请参阅QMetaObject：：normalizedSignature。`connect()`

## 文件头

如果您创建一个新文件，文件的顶部应该包含一个标题注释，该注释与Qt Creator的其他源文件中的标题注释相同。

## 包括标题

- 使用以下格式包含 Qt 标头：不要包含该模块，因为它可能在 Qt4 和 Qt5 之间发生了变化。`#include <QWhatEver>`
- 按从特定到通用的顺序排列包含，以确保标头是独立的。例如：
  - `#include "myclass.h"`
  - `#include "otherclassinplugin.h"`
  - `#include <otherplugin/someclass.h>`
  - `#include <QtClass>`
  - `#include <stdthing>`
  - `#include <system.h>`
- 将其他插件的标头括在尖括号 （<>） 而不是引号 （“”） 中，以便更容易发现源代码中的外部依赖项。
- 在*长对等*标头块之间添加空行，并尝试在块内按字母顺序排列标头。

## 铸造

- 避免 C 转换。首选C++转换（ ）。Both和 C 样式转换很危险，但至少不会删除 const 修饰符

么。dynamic_castqobject_casttype()

## 编译器和特定于平台的问题

- 使用问号运算符时要格外小心。如果返回的类型不相同，某些编译器会生成在运行时崩溃的代码（您甚至不会收到编译器警告）：

```
QString s;
// crash at runtime - QString vs. const char *
return condition ? s : "nothing";
```

- 对齐时要格外小心。

  每当强制转换指针以增加目标所需的对齐方式时，在某些体系结构上，生成的代码可能会在运行时崩溃。例如，如果 ais 转换为 a，它将在整数必须在两字节或四字节边界对齐的机器上崩溃。const char *const int *

  使用联合强制编译器正确对齐变量。在下面的示例中，您可以确保 AlignHelper 的所有实例都以整数边界对齐：

```
union AlignHelper
{
    char c;
    int i;
};
```

- 坚持使用整型、整型数组及其结构，用于标头中的静态声明。例如，在大多数情况下，不会在每个插件中优化和复制，最好避免它。static float i[SIZE_CONSTANT];
- 任何具有构造函数或需要运行要初始化的代码的内容都不能用作库代码中的全局对象，因为它未定义何时运行该构造函数或代码（首次使用时、库加载时、之前或根本不运行）。main()

  即使为共享库定义了初始值设定项的执行时间，在插件中移动该代码或静态编译库时也会遇到麻烦：

```
// global scope

-NOT-

// Default constructor needs to be run to initialize x:
static const QString x;

-NOT-

// Constructor that takes a const char * has to be run:
static const QString y = "Hello";

-NOT-

QString z;

-NOT-

// Call time of foo() undefined, might not be called at all:
static const int i = foo();
```

```
// global scope
// No constructor must be run, x set at compile time:
static const char x[] = "someText";

// y will be set at compile time:
static int y = 7;

// Will be initialized statically, no code being run.
static MyStruct s = {1, 2, 3};

// Pointers to objects are OK, no code needed to be run to
// initialize ptr:
static QString *ptr = 0;

// Use Q_GLOBAL_STATIC to create static global objects instead:

Q_GLOBAL_STATIC(QString, s)

void foo()
{
    s()->append("moo");
}
```

**注意：**函数作用域中的静态对象没有问题。构造函数将在首次输入函数时运行。但是，该代码不是可重入的。

- A是有符号还是无符号取决于体系结构。如果您明确需要有符号或无符号字符，请使用 signedor。例如，以下代码将在 PowerPC 上中断：charcharuchar

```
// Condition is always true on platforms where the
// default is unsigned:
if (c >= 0) {
    ...
}
```

- 避免使用 64 位枚举值。嵌入的 AAPCS（ARM 体系结构的过程调用标准）将所有枚举值硬编码为 32 位整数。
- 不要混合使用常量和非常量迭代器。这将在损坏的编译器上静默崩溃。

```
for (Container::const_iterator it = c.constBegin(); it != c.constEnd(); ++it)

-NOT-

for (Container::const_iterator it = c.begin(); it != c.end(); ++it)
```

- 不要在导出的类中内联虚拟析构函数。这会导致依赖插件中出现重复的 vtables，这也可能破坏 RTTI。请参阅QTBUG-45582。

## 美学

- 首选无作用域枚举来定义 const 而不是静态 const int 或定义。枚举值将在编译时由编译器替换，从而加快代码速度。

## 从模板或工具类继承

从模板或工具类继承具有以下潜在缺陷：

- 析构函数不是虚拟的，这可能会导致内存泄漏。
- 符号不会导出（并且主要是内联的），这可能会导致符号冲突。

例如，库 A 有类，库 B 有类。突然，QList符号从两个库中导出，从而导致冲突。`Q_EXPORT X: public QList<QVariant> {};Q_EXPORT Y: public QList<QVariant> {};`

## 继承与聚合

- 如果存在明确的*is-a*关系，请使用继承。
- 使用聚合重用正交构建基块。
- 如果有选择，首选聚合而不是继承。

## 公共头文件的约定

我们的公共头文件必须经受住某些用户的严格设置。所有已安装的标头都必须遵循以下规则：

- 无 C 样式转换 （） 。使用，或者，对于基本类型，使用构造函数形式：代替。有关详细信息，请参阅强制转换。`-Wold-style-caststatic_castconst_castreinterpret_castint(a)(int)a`
- 没有浮点比较 （） 。用于将值与增量进行比较。用于检查浮点数是否为二进制 0，而不是将其与 0.0 进行比较，或者最好将此类代码移动到实现文件中。`-Wfloat-equalqFuzzyCompareqIsNull`
- 不要隐藏子类中的虚函数（{-Woverload-virtual}）。如果基类 A 有一个虚拟的重载，子类 B 有一个同名的重载，则隐藏 Afunction。使用关键字使其再次可见，并为损坏的编译器添加以下愚蠢的解决方法：`int val()int val(int x)valusing`

```
class B: public A
{
#ifdef Q_NO_USING_KEYWORD
inline int val() { return A::val(); }
#else
using A::val;
#endif
};
```

- 不要隐藏变量 （） 。`-Wshadow`
- 尽可能避免类似的事情。`this->x = x;`
- 不要为变量指定与类中声明的函数相同的名称。
- 若要提高代码的可读性，请始终在探测预处理器变量的值 （） 之前检查是否定义了预处理器变量。`-Wundef`

```
#if defined(Foo) && Foo == 0

  -NOT-

#if Foo == 0

-NOT-

#if Foo - 0 == 0
```

- 使用运算符检查预处理器定义时，请始终在括号中包含变量名称。defined

```
#if defined(Foo)

  -NOT-

  #if defined Foo
```

## 类成员名称

我们使用“m_”前缀约定，除了公共结构成员（通常在*私有类和非常罕见的真正公共结构中）。指针不受“m_”规则的约束。dq

指针（“Pimpls”）被命名为“d”，而不是“m_d”。指针 inis 的类型，其中在与相应的 {Internal} 命名空间中声明或导出的 ifis 位于同一命名空间中。ddclass FooFooPrivate *FooPrivateFooFoo

如果需要（例如当私有对象需要发出适当类的信号时），可以成为朋友。FooPrivateFoo

如果私有类需要对真实类的反向引用，则指针命名，其类型为。（与Qt中的约定相同：“q”看起来像一个倒置的“d”。qFoo *

不要使用智能指针来保护指针，因为它会施加编译和链接时间开销，并使用更多符号创建更胖的目标代码，例如导致减慢调试器启动速度：d

```
############### bar.h

#include <QScopedPointer>
//#include <memory>

struct BarPrivate;

struct Bar
{
    Bar();
    ~Bar();
    int value() const;

    QScopedPointer<BarPrivate> d;
    //std::unique_ptr<BarPrivate> d;
};

############### bar.cpp

#include "bar.h"

struct BarPrivate { BarPrivate() : i(23) {} int i; };

Bar::Bar() : d(new BarPrivate) {}

Bar::~Bar() {}

int Bar::value() const { return d->i; }

############### baruser.cpp

#include "bar.h"
```

```
############### baz.h

struct BazPrivate;

struct Baz
{
    Baz();
    ~Baz();
    int value() const;

    BazPrivate *d;
};

############### baz.cpp

#include "baz.h"

struct BazPrivate { BazPrivate() : i(23) {} int i; };

Baz::Baz() : d(new BazPrivate) {}

Baz::~Baz() { delete d; }

int Baz::value() const { return d->i; }
```



```
#include "baz.h"

int bazUser() { Baz b; return b.value(); }

############### main.cpp

int barUser();
int bazUser();

int main() { return barUser() + bazUser(); }
```

结果:

```
Object file size:

 14428 bar.o
  4744 baz.o

  8508 baruser.o
  2952 bazuser.o

Symbols in bar.o:

    00000000 W _ZN3Foo10BarPrivateC1Ev
    00000036 T _ZN3Foo3BarC1Ev
    00000000 T _ZN3Foo3BarC2Ev
    00000080 T _ZN3Foo3BarD1Ev
    0000006c T _ZN3Foo3BarD2Ev
    00000000 W _ZN14QScopedPointerIN3Foo10BarPrivateENS_21QScopedPointerDeleterIS2_EEEC1EPS2_
```

```
                 U _ZN9qt_assertEPKcS1_i
      00000094 T _ZNK3Foo3Bar5valueEv
      00000000 W _ZNK14QScopedPointerIN3Foo10BarPrivateENS_21QScopedPointerDeleterIS2_EEEptEv
                 U _ZdlPv
                 U _Znwj
                 U __gxx_personality_v0

  Symbols in baz.o:

      00000000 W _ZN3Foo10BazPrivateC1Ev
      0000002c T _ZN3Foo3BazC1Ev
      00000000 T _ZN3Foo3BazC2Ev
      0000006e T _ZN3Foo3BazD1Ev
      00000058 T _ZN3Foo3BazD2Ev
      00000084 T _ZNK3Foo3Baz5valueEv
                 U _ZdlPv
                 U _Znwj
                 U __gxx_personality_v0
```

## 文档

文档是从源文件和头文件生成的。您为其他开发人员编写文档，而不是为自己编写文档。在头文件中，文档界面。也就是说，函数做什么，而不是实现。

在.cpp文件中，如果实现不明显，则可以记录实现。

‹ 用户界面文本指南

Qt 创建者 API 参考 ›