Qt 编程惯例

翻　译：wd007　XChinux

审　校：齐　亮

◆ C++ 特性

■ 不要使用异常

■ 不要使用 rtti（运行时类型识别;主要包括,typeinfo 结构,dynamic 或者 typeid 操作符，包括抛出异常等）

■ 慎明的使用模板类,只有当必需时才使用它,而不是仅仅因为你掌握了它的使用方法。

提示：使用编译器的自动测试功能来确定某个 C++特性是否被支持。

◆ Qt 源代码书写惯例

■ 所有代码都仅仅使用 ascii 格式 7 位字符 如果不确定,则可以运行 man ascii 来验证 ）

● 原理:我们已经有了太多的新的语系作品以及 UTF-8 和 Latin1 混合的不健壮的系统。当你在喜欢使用的编辑器中使用字符位数大于 127 的系统时，往往还没有来得及保存，系统就会崩溃掉。

● 对于字符串：使用\nnn（这里 nnn 是八进制的字符，在任何系统中都可以代表你要输入的字符）或者使用\xnn（这里 xnn 是十六进制的字符）。

举例：

```
QString s = QString::fromUtf8(”\213\005”);
```

● 对于文档中的变音，或者其他非 ASCII 的字符，或者使用 qdoc 的 unicode 命令或者使用适当的宏；举例：对ü使用\uuml。

■ 所有 QObject 的子类都必须包含一个 Q_OBJECT 的宏,即便它并没有使用信号或者槽，否则的话，qobject_cast 就会运行失败。

■ 在连接语句中，规范化信号和槽中的参数（可以查看 QMetaObject::normalizedSignature 以获得规范化的信息）以获得更快速的信号/槽遍

历。你也可以使用$QTDIR/util/normalize 来规范已有的代码

◎ 包含的头文件

■ 在公用的头文件中，头文件包含总是使用这样的形式：

```
#include <QtCore/qwhatever.h>
```

对于 Mac OS X 系统框架而言，QtCore 这样的库前缀是必需的，这样的写法对非 qmake 项目也是非常便利的

■ 在源文件中，先包含特定的头文件，再包含常用的头文件（如系统文件）

举例：

```
#include <qstring.h>   // Qt 的类
#include <new>         // 标准模板库中的模块
#include <limits.h>   // 系统文件
```

■ 如果要包含 qplatformdefs.h，总是第一个包含它

■ 如果要包含 qt_x11_p.h，总是最后一个包含它

◎ 强制转换

■ 避免使用 C 风格的转换，使用 C++风格的转换（static_cast, const_cast, reinterpret_cast）

● 原理：reinterpret_cast 以及 C 风格的转换都是危险的，但好在前者在转换过程中并不去掉 const 修饰符

■ 不要使用 dynamic_cast，对 QObject 及其子类对象使用 qobject_cast 或者重构你的设计，例如引入一个 type()方法（参考 QListWidgetItem）

■ 对于简单的变量类型转换，使用构造函数实现。注意形式是：int(myFloat)而不是(int)myFloat

● 原理：当重构代码时，编译器会立即告诉开发者这种转换是否会带来危险

◎ 编译器/平台特定问题

■ 当使用有疑问的操作符时，要格外的小心。如果返回值类型不一致，某些编译器生成的代码在运行时会崩溃（你甚至不会得到编译器给出的警告信息）

```
QString s;
return condition ? s : ”nothing”;  //运行时崩溃
```

■ 对字节对齐问题要格外的小心

● 无论何时，进行一个指针转换以致于需要目标进行对齐而导致字节增加，其结果可能致使在某些系统架构上运行时产生崩溃。举个例子，如果一个 const char *类型转换为一个 const int *，这会在整型必须被对齐到两个或四个字节边界的机器上导致崩溃。

● 使用一个联合体来强制编译器正确的对变量进行对齐。在下面的例子中，你能够确定 AlignHelper 的所有实例在整数的边界处对齐。

```
union AlignHelper [1]
{
    char c;
    int i;
};
```

■ 在库代码中，任何具有一个构造函数或需要运行一段代码来初始化的变量不能被用做全局对象。因为其构造函数或这段代码什么时候运行是未定义的（在第一次使用时，在库加载时，在 main()函数之前或者其它场合）。甚至是在共享库中已定义了的初始化过程执行时，在将这段代码移到一个插件中或者这个库是被静态编译的时候会遇到麻烦。

```
// 全局域
static const QString x;           // 错误 - 缺省构造函数需要在初始化 x 变量时运行
static const QString y = "Hello"; // 错误 - 必须要运行以 const char *类型为参数的构造函数
QString z;                        // 大错特错
static const int i = foo();       // 错误 - 调用时 foo()是尚未被定义的，务必不要这样使用
```

可以这样做：

```
// 全局域
static const char x[] = "someText"; // 可以的 - 没有构造函数必须被运行，x 在编译时被赋值。
static int y = 7;                   // 可以的 - y 会在编译时赋值
static MyStruct s = {1, 2, 3};      // 可以的 - 静态初始化，没有代码要执行
static QString *ptr = 0;            // 指向对象的指针是可以的 - 初始化指针时不需要运行任
                                    // 何代码
```

使用 Q_GLOBAL_STATIC 宏来创建静态全局对象：

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

注意：在局部范围内使用静态对象是没有问题的，当进入这个局部时，构造函数会首次被运行。但是这段代码是不可重入的。

■ char 的值是否是有符号的,取决于系统架构。所以如果你要明确的使用有符号或者是无符号字符时，请使用 signed char 或者 是 uchar 来表示。下面的代码在 ppc[2]上运行时会崩溃掉：

```
char c = 'A';
if (c < 0) { ... }  // 错误-但当所在平台的默认型式（指的是 char）是 unsigned char 时，
                    // 该句的判断条件总是正确的
```

■ 试着去避免使用 64 位的枚举类型

● aapcs embedded ABI[3]将所有的枚举类型都规定死为 32 位整型了[4]。

◎ 程序编码美学[5]

■ 尽量使用枚举来定义常量而不是使用类似 static const int 这样的用法，也不要使用宏定义。原因如下：

● 枚举变量会在编译时被编译器替换掉，这会使程序的运行速度更快。

● 宏定义并不是命名空间安全的（并且看上去很丑陋）。

■ 在头文件中尽量使用详尽的参数名字

● 绝大多数的 IDE 会在其“代码自动补全”功能处显示出参数的名称

● 在文档中看上去更清楚

● 味道很坏的型式：

```
doSomething(QRegion rgn, QPoint p)
```

请使用下面的型式来代替：

```
doSomething(QRegion clientRegion, QPoint gravitySource)
```

◎ 需要避免的事情

■ 不要从模板/工具类进行继承

● 析构函数不是虚函数，这可能会引起内存泄露

● 标识符不是输出的（绝大多数是内联的），这会导致有意思的标识符冲突

● 举例来说，库 A 内有类定义：class Q_EXPORT X: public QList<QVariant> {}；库 B 内有类定义：class Q_EXPORT Y: public QList<QVariant> {}；这时如果 QList<QVariant>中的标识符要从这两个库中输出，则将产生标识符冲突。

■ 不要混合使用常量和非常量迭代器。在某些编译器上这会导致“悄悄的”崩溃

```
for (Container::const_iterator it = c.begin(); it != c.end(); ++it)           // 错误
for (Container::const_iterator it = c.constBegin(); it != c.constEnd(); ++it) // 正确
```

■ Q[Core]Application 是一个单例模式的类。一次只能有一个实例。但是，可以在销毁这个实例的同时创建一个新的实例，这就像在一个<nop>ActiveQt 或者是一个浏览器插件中一样。像下面这样的代码就很容易导致程序崩溃：

```
static QObject *obj = 0;
if (!obj)
obj = new QObject(QCoreApplication::instance());
```

如果基于 QCoreApplication 的应用程序已经销毁，obj 就会变成一个“野指针”。使用 Q_GLOBAL_STATIC 来创建静态全局实例，或者使用 qAddPostRoutine 来执行清理工作。

■ 如果可能的话，避免使用匿名名字空间，这有利于静态的关键字。一个带有 static 的定位到编译单元中的名字，会保证自己与编译单元之间有一个内部连接。然而不幸的是，对于在匿名名字空间中声明的名字，C++标准要求使用外部连接。(7.1.1/6, 或通过在 gcc 邮件列表中查阅各种关于匿名名字空间的讨论来了解这方面的内容)

◎ 二进制以及源代码的兼容性

■ 规定：

● Qt 4.0.0 是一个主要的发布（版本），Qt 4.1.0 是一个次要的发布（版本），Qt 4.1.1 是一个分支发布（版本）

● 向后的二进制兼容性：链接到旧的库的代码会运行良好

● 向前的二进制兼容性：链接到新的库的代码在链接到旧的库时会运行良好

● 源代码兼容性：若对源代码未作更改，则会保持兼容

■ 在次要的发布中保持向后的二进制以及源代码的兼容性

■ 在分支发布中保持向后和向前的二进制以及源代码的兼容性

● 不要增加/移除公用的 API（例如全局函数，公用的/受保护的/私有的方法）

● 不要重新实现方法（甚至包括内联的和受保护的/私有的方法）

● 在《Binary Compatibility Workarounds》[6]一文中对照检查，找到保持二进制兼容的途径

■ 关于二进制兼容的更多信息请参阅：http://techbase.kde.org/Policies/Binary_Compatibility_Issues_With_C++ [techbase.kde.org]++

■ 当子类化一个QWidget时，记着要总是重写它的event()，即便该函数体是空的。这就能确保该部件在不破坏二进制兼容性的情况下保持固定

■ 所有从 Qt 中导出的方法必须以字母‘q’或者‘Q’开头。使用这些“标识符”的自动测试功能来找出违反之处

◎ 名字空间

阅读《Qt In Namespace》[7]，并确保牢记在 Qt 中除了测试和 Webkit 模块以外，都是“名字空间化”的。

◆ 公用头文件的编程惯例

我们使用的公用头文件必须适应某些用户的严格设定，所有安装的头文件必须遵循下面的规则：

■ 不要使用 C 风格的强制转换（GCC 命令行参数是：-Wold-style-cast）

- 使用 static_cast, const_cast 或者是 reinterpret_cast
- 对于基础类型，使用的构造形式是：int(a)而不是(int)a
- 阅读“强制转换”一节来获得更多信息

■ 不要对 float 的值进行比较（GCC 命令行参数是：-Wfloat-equal）

- 使用 qFuzzyCompare 来对 float 的值进行比较
- 使用 qisNull[8]来验证一个 float 值是否是二进制的 0，而不要通过与 0.0 的比较来验证

■ 不要在子类中隐藏虚方法（GCC 命令行参数是：-Woverloaded-virtual）

- 举例：假如基类 A 有一个虚方法 virtual int val()，而子类 B 重载了这个方法为 int val(int x),A 的 val()方法被隐藏了。要使用关键字 using 来使得它再次可见，并且要通过加上下列这样的代码来迂回避免编译器崩溃：

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

■ 不要采取“遮蔽”变量名的做法（GCC 命令行参数：-Wshadow）

- 避免出现这样的用法：

```
this->x = x;
```

- 不要给你的类中的方法和变量取一样的名字

■ 总是在使用一个预处理器变量名字的值之前，对它的定义进行检测（GCC 命令行参数：-Wundef）

```
#if Foo == 0                    // 错误
```

```
#if defined(Foo) && Foo == 0   // 正确
#if Foo - 0 == 0               // 这种做法似是而非，并不易读。使用上面的做法更好
```

注释：

[1] 原文示例源代码的格式为：

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

但我们并不赞同 Qt 推荐的这种格式，原因是目前绝大多数的编码规范都建议“括号独占一行，前后相对呼应”。

[2] ppc 的原文为：programming planning and control，意为“程序设计与控制”，这里可以理解为在程序实际的运行环境中。

[3] aapcs 即 ARM 汇编语言与嵌入式 C 语言进行相互调用标准。

[4] 本句原文为：

```
The aapcs embedded ABI hard codes all enum values to a 32-bit integer.
```

其中 hard code 表示手工编码的意思，相对的是比如工具自动生成代码等等。在这里应理解为“程序中写死”、“强制限定”的意思。

[5] 原文单词为：aestethics，但这个词在词典中找不到，推断应该是 aesthetic，即程序编码美学的意思。

[6] 参见后续译校的《实现二进制兼容的变通方法》一文。

[7] 参见后续译校的《Qt 中的的名字空间》一文。