第7章　库与设计模式

库是一组代码模块，它按照可复用的方式组织。本章将探讨如何构建、复用和设计库。也会介绍和探讨设计模式。

术语“平台”指的是硬件体系结构与软件架构的一种特定组合，前者尤指中央处理单元(CPU)[①]，后者通常指操作系统(OS)[②]。每一种计算机系统都只能执行以其自己低级的、特定于平台的语言编写的代码。这种低级的机器语言与任何自然界的人类语言都不相同，几乎没有程序员能够方便地直接使用它。

为了最优化编程资源的使用(尤其是在程序员的时间)，我们需要使用高级语言(例如，C++)来编写程序，以便以尽量接近于自然语言(例如，英语)的形式来表达并分享思想和精确的指令。将高级语言代码翻译成机器语言以便能够在特定的计算机平台上执行的工作由编译器[③]负责。

对代码复用的价值的广泛认知，导致了对代码库(及其产品)的急剧需求。代码库中保存的是有用的、可复用的、编译后的代码，这样，程序员不必处理代码库的任何源代码就能够利用它的功能。当用“`#include`”指令包含库模块的头文件时，就可以复用这个库模块。这个指令会在适当的源代码模块中指定应用编程接口(API)。前面已经复用过来自于标准库和Qt的几个库模块，前者如`iostream`和`string`，后者如`QString`，`QTextStream`和`QList`。当复用来自于库中的任何模块时，其工作由链接器(linker)在链编(build)过程完成，以在编译后代码中的项目引用与编译后库代码中的项目定义之间构建适当的链接，得到的可执行文件必须在运行时找到并动态地链接到编译后的库(称为运行时库)。编译后的库代码并不需要集成到可执行文件中，因为它能够在运行时动态地链接。这样就可以得到更小的可执行文件，并能更高效地使用内存。

库(lib)是一个文件，它包含一个或者多个编译后文件(称为目标文件)，并对其进行了索引，以便链接器能够更容易地找到符号(例如，类的名称、类成员、函数、变量等)以及它们的定义。将多个目标文件集成在一个库中，能够显著提速链接过程。

C++库能够以多种途径被打包：

- 开源包
- `dev`包
- 运行时库

开源包通常以压缩的档案文件形式发布，它包含全部的源代码、头文件以及链编脚本和文档。`dev`包在Linux包管理程序中有时被称为“`-devel`”包，它通常以档案文件的形式发布，包含一个库以及相关的头文件。这种格式使得在发布库时可以不带源代码，而其他程序

---

① 例如，Intel Core 2 Duo或者SPARC64“Venus”。

② 例如，Linux, Windows7, Mac_OS_X。

③ 通过将Java代码编译成称为字节码的中间状态，Java编译器能够产生独立于平台的代码。然后，用特定于平台的Java虚拟机就可以将字节码翻译成机器语言。这种安排利用了平台独立性的许多优点，但在性能上有所损失。

员依然可以编译他们的应用。运行时库由 lib 文件组成，没有相关联的头文件，所以它只能用于执行已经用这个库链编过的应用。

能够用来组织和打包 C++代码的各种方式总结如下，表 7.1 中定义了描述代码容器的一些术语。

表 7.1 可复用组件

术 语	可视化属性	描 述
类	class Classname{ body } ;	函数、数据成员以及其生命周期管理描述(构造函数和析构函数)的集合
命名空间	namespace name{ body } ;	类、函数及其静态成员的声明的集合，可能会分布在多个文件中
头文件	.h	类定义、模板定义、函数声明(带有默认实参定义)、inline 函数定义、静态对象声明
源代码模块	.cpp	函数定义、静态对象定义
编译后“对象”模块	.o 或者.obj	每一个.cpp 模块都被编译进一个二进制模块中，作为链编库或者可执行文件的中间一步
库	.lib 或者.la(如果是动态库，则也可以是.so 或者.dll)	对象文件的索引集合被连接在一起。库中的任何代码模块不必存在 main()函数
devel 包	.lib 加上头文件	库与相关联的头文件的结合
应用程序	Windows 下的扩展名为.exe，*nix 下没有特定的扩展名	目标文件与库相连接的一个集合，形成一个应用程序。只能包含一个称为 main()的函数定义

7.1 建立并复用库

本书中的许多示例都连接了各种各样的库。可以下载 tarball 文件 src.tar.gz，它包含 dist 目录下使用的代码和库[①]。解包这个 tarball 文件并创建一个 shell/环境变量 CPPLIBS，使其包含到 src/libs 目录的绝对路径。

注意

当设置复用这些库的工程时，总是假定 shell/环境变量 CPPLIBS(或者 Windows 系统下的 %CPPLIBS%)已经被正确地设置成包含 libs 根目录。

这个变量有两个作用：它是库的全部 C++源代码的父目录，同时也是这些库的已编译过的共享目标代码的目标目录。

qmake 可以在工程文件里访问 CPPLIBS 环境变量，其访问语法是$$(CPPLIBS)。qmake 也可以包含其他的工程文件(片断)。例如，示例 7.1 中的工程文件包含文件 common.pri，它位于示例 1.6 中，用于常见的应用链编设置。

示例 7.1 src/xml/domwalker/domwalker.pro

```
# include common qmake settings
include (../../common.pri)

# this project depends on libdataobjects:
LIBS += -ldataobjects
```

① URL 可以在参考文献中找到。

```
# this directory contains the libraries:
LIBS += -L$$(CPPLIBS)

# Search here for headers:
INCLUDEPATH += . $$(CPPLIBS)/dataobjects

QT += xml gui

CONFIG += console
TEMPLATE = app

SOURCES += main.cpp slacker.cpp domwalker.cpp xmltreemodel.cpp
HEADERS += slacker.h domwalker.h xmltreemodel.h
```

此外，这个工程还为 LIBS 和 INCLUDEPATH 变量添加了一些值，以使它能够找到所依赖的库和头文件。

命令：

```
qmake -project
```

产生的工程文件所包含的信息，只会以当前工作目录下的内容为基础。特别地，qmake 无法知道用来构建工程所需要的外部库。如果工程依赖于某个外部库，则必须编辑这个工程文件，将它的值赋予变量 INCLUDEPATH 和 LIBS。然后，再次运行 qmake -project 会彻底破坏这些改变，所以不要这么做！

例如，假设应用需使用 dataobjects 库，其头文件位于$CPPLIBS/dataobjects 下，而共享目标文件的库位于$CPPLIBS 下，那么必须将如下这些行添加到工程文件中：

```
INCLUDEPATH += $$(CPPLIBS)/dataobjects  # the source header files
LIBS += -L$$(CPPLIBS)              # add this to the lib search path
LIBS += -ldataobjects                   # link with libdataobjects.so
```

给变量 LIBS 赋值通常包含两种直接传递给编译器和链接器的开关。关于链接器开关的更多信息，请参见 C.3 节。

7.1.1　组织库：依赖性管理

如果某个程序元素复用了另一个程序元素，则它们之间就存在依赖性。也就是说，构建、使用或者测试某个程序元素(复用者)时，要求另一个程序元素(被复用者)存在并且是正确的。对于类，只要被复用者类的接口发生改变，就使得复用者类的实现必须改变，则它们之间就存在依赖性。

描述这种关系的另一种方法是：如果构建 ProgElement1 时需要 ProgElement2，则就称 ProgElement1 依赖于 ProgElement2。

如果编译 ProgElement2.cpp 时必须包含 ProgElement1.h，则称这种依赖性为编译时依赖性；如果目标文件 ProgElement2.o 包含 ProgElement1.o 中定义的符号，则称其为连接时依赖性。

图 7.1　依赖性

图 7.1 中通过 UML 框图给出了复用者 ClassA 和复用者 ClassB 之间的依赖性。

ClassA 与 ClassB 之间的依赖性能够以各种形式出现。在下面各种情形下，ClassB 接口的变化会导致 ClassA 的实现也必须改变。

- ClassA 的数据成员为 ClassB 的对象或者指针。
- ClassA 派生自 ClassB。
- ClassA 的函数具有 ClassB 类型的参数。
- ClassA 的函数使用 ClassB 的静态成员。
- ClassA 向 ClassB 发送消息(例如，信号)[①]。

在每一种情况下，都必须在 ClassA 的实现文件中包含指令#include ClassB。

图 7.2 给出的包框图中,列出了一部分 libs 库集合。其中存在直接依赖性和间接依赖性。这一节将关注库之间的依赖性(用虚线箭头表示)。

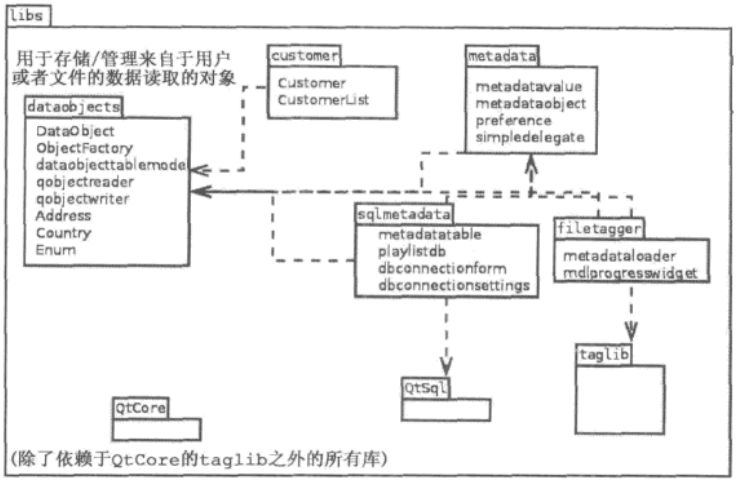

图 7.2 库及其依赖性

如果希望复用图 7.2 中的任何一个库，就需要确保它的所有依赖库也是工程的一部分。例如，如果使用 filetagger 库，则会存在一个依赖链，要求也使用 dataobjects 库(例如，派生自 DataObject 的那些 MetaData 类)和 taglib 库(例如，filetagger 使用 taglib 来加载元数据)。如果希望使用 sqlmetadata，则需要 QtSql，即 Qt 的 SQL 模块。

代码复用，这是一个有价值且重要的目标，但总会生成依赖。当设计类和库时，需要确保尽可能地减少不必要的或者无意的依赖性，因为它们会延长编译时间，降低类和库的可复用性。每一个#include 指令都会产生一种依赖性，所以要仔细检查，以保证它确实是必要的。这一点对头文件尤其重要：每一次用#include 指令包含头文件时，都会带入这个头文件中所包含的其他头文件，这样它们之间的依赖性就会相应地增多。

类的前置声明(forward declaration)将它的名称声明成一个有效的类名称，但不给出类的定义。这会使得类名称能够被用作指针和引用的类型，在遇到类的定义之前，不会将这些指针和引用进行解引用操作。对于类而言，前置声明使得头文件之间可以具有环形关系而没有环形依赖性(编译器不允许这样)。

① 8.5 节中将探讨信号和槽。

对于类定义头文件，需遵循的一条规则是：如果可以使用前置声明，就不要使用#include指令。例如，头文件 classa.h 的内容可能是这样的：

```
#include "classb.h"
#include "classd.h"
// other #include directives as needed
class ClassC;    // forward declaration
class ClassA : public ClassB {
  public:
    ClassC* f1(ClassD);
  // other stuff that does not involve ClassC
};
```

这个定义中，存在(至少)两个故意设置的复用依赖性：ClassB 和 ClassD，所以两个#include 指令都是必要的。但是，ClassC 有前置声明就足够了，因为类定义只使用了这个类的指针。

依赖性管理是一个重要的问题，它是许多文章的主题，且已经为它开发出了各种工具，其中的两个开源工具如下。

- cinclude2dot[①]，一种 Perl 脚本，它会分析 C/C++代码并产生一个依赖性图。
- Makedep[②]，一个用于大型软件工程的 C/C++依赖性生成程序，它会分析目录树下的全部源文件，并会构造出一个大型的依赖性文件用于包含在 Makefile 中。

7.1.2　安装库

当编写并测试完库之后，在链编过程结束后它将被安装到由 qmake 变量 DESTDIR 指定的目录下。例如，dataobjects 库的工程文件包含如下相关的行：

```
TEMPLATE = lib      # Build this as a library, not as an application
DESTDIR=$$(CPPLIBS) # Place the compiled shared object code here
```

对于库模板，qmake 将产生一个包含 install 目标的 Makefile。这样，在成功链编之后，命令

```
make install
```

将会把这个库复制到某个特定的位置。例如，在*nix 平台上，可以在 dataobjects 库的工程文件中添加如下的行：

```
target.path=/usr/lib
INSTALLS += target
```

然后，如果对目标位置有写权限，则命令

```
make install
```

会将 libdataobjects.so 文件以及与它们相关联的符号链接复制到/usr/lib 目录，使得这个库能被登录到计算机的任何人使用。

如果要迁移库，则过程会根据平台的不同而有所不同。在 Windows 系统中，可以将它的.dll 文件复制到 PATH 变量中给出的合适目录下。在*nix 系统中，可以将共享的目标文件以及相关的符号链接复制到/etc/ld.so.conf 中列出的目录下，也可以复制到 LD_LIBRARY_PATH 变量中列出的目录下。

① 参见 http://www.flourish.org/cinclude2dot/。
② 参见 http://sourceforge.net/projects/makedep。

在开发过程中，通常只需在 CPPLIBS 目录下编译并安装库，并相应调整 LD_LIBRARY_PATH 设置即可。在*nix 系统中，如果遵循了上面的建议，并且一直使用了 CPPLIBS 环境变量，则只需在.bashrc 文件中添加下面的一行：

```
export LD_LIBRARY_PATH=$CPPLIBS
```

将这一行置于定义 CPPLIBS 环境变量的那一行的下面即可。在*nix 平台下部署库时，理想的安装目录是/usr/local，它是一个全部用户都能够访问的系统级位置。这个操作要求有超级用户权限。

注意

Qt 中并不要求 QTDIR 环境变量，但本书中有时在不同的地方会将其作为解压的 Qt tarball 的基本目录。它提供了一种方便的途径来引用 Qt 的示例、教程、二进制文件以及库。但是，在诸如 Ubuntu 或者 Debian 的系统中，Qt 被分成了多个文件夹并被安装在不同的位置，所以不应当定义 QTDIR。这种情况下，可以将 QTDIR 的使用理解成“Qt 二进制文件的父目录”或者“Qt 示例的父目录”（可以有两个不同的目录）。

提示

在 Microsoft 平台上建立动态链接库(DLL)要复杂得多，因为需要为每一个库定义唯一的“导出者”宏。

根据头文件是否从自己的库里包含还是从外部程序包含，这个宏会扩展成适当的 declspec()导出或者导入。

可以使用在<qglobal.h>中定义的预处理器宏 Q_DECL_EXPORT，它能够有条件地开启或者关闭 declspec，并可以在合适的时刻导入或者导出这个标志符。

例如，libdataobjects 中定义了一个名称为 DOBJS_EXPORT 的导出宏，见示例 7.2。

示例 7.2 src/libs/dataobjects/dobjs_export.h

```
#include <QtGlobal>
#ifndef DOBJS_EXPORT
/* This macro is only for building DLLs on Windows. */
#ifndef Q_OS_WIN
#define DOBJS_EXPORT
#elif defined(DATAOBJECTS_DLL)
#define DOBJS_EXPORT Q_DECL_EXPORT
#else
#define DOBJS_EXPORT Q_DECL_IMPORT
#endif
#endif
```

只有当 dataobjects.pro 中定义了 DATAOBJECTS_DLL 时，这个宏才会导出符号：

```
win32 {
   CONFIG(dll) {
      DEFINES += DATAOBJECTS_DLL
   }
}
```

现在，对于希望导出到 DLL 的任何类，只需将这个宏置于 class 关键字与类定义中的类名称之间即可：

```
class DOBJS_EXPORT DataObject : public QObject {
  [ . . . ]
}
```

当构建第一个 DLL 时，qtcentre.org 提供了关于这个主题的一些思路[①]，它能帮助节省许多时间。

注意

QLibrary 类中的几个方法可用来以独立于平台的方式在运行时加载或者卸载动态库。

7.2　练习：安装库

本书中的大量示例使用了库中的一些类，这些库是专门为这本书而编写的。可以下载这些类的源代码[②]，其中包含用 Doxygen 生成的 API 文档。这个练习中将看到如何建立并安装一些库。

随后，将给出在*nix 平台上安装库供本书示例使用的指导。在 Windows 系统上用 MinGW 安装 Qt 和 MySQL 的帮助，请查看 QtCentre Wiki[③]。

正如 7.1 节和 7.1.2 节中建议的那样，需依次执行如下步骤：

- 创建一个专门用于 C++/Qt 的目录，例如~/oop/projects/。
- 从 dist 目录下载 src.tar.gz[④]。
- 将这个 tarball 文件解压到一个新目录。结果应当是一个 libs 目录和许多的子目录，其中包括 libs/dataobjects 和 libs/customer。
- 在 libs 目录下检查名称为 libs.pro 的 subdirs 工程文件。
- 这个工程文件是用来建立库并进行测试的。
- 可随意将那些不准备使用的库注释掉，但是不要改变它们的顺序。
- 创建一个名称为 CPPLIBS 的 shell/环境变量，使之包含新的 libs 目录的绝对路径。为了方便，可以将这个环境变量的定义放入 shell 脚本中，如示例 7.3 所示。

注意

如果决定要将 libs.pro 中的某个特定目录注释掉，也应将 libs/tests/tests.pro 中相应的测试目录注释掉(否则不会建立库的测试部分)。

需提醒的是：在工程文件中某一行的开头插入一个#字符，就能将它注释掉。

① 参见 http://www.qtcentre.org/forum/showthread.php? t=1080。

② 来自于 dist 目录。

③ 参见 http://wiki.qtcentre.org/index.php? title=Building_the_QMYSQL_plugin_on_Windows_using_MinGW。

④ URL 可以在参考文献中找到。

libs.pro

```
TEMPLATE = subdirs
CPPLIBS=$$(CPPLIBS)
isEmpty(CPPLIBS) {
   error("Define CPPLIBS environment variable to point to this location.")
   }
SUBDIRS += dataobjects \
           actioneditor \
           customer \
#          metadata \
#          sqlmetadata
```

从 libs 目录建立库需要两个步骤:

1. qmake -recursive //在$CPPLIBS 和每一个子目录下创建 Makefile。
2. make //建立库并测试(每一个 Hc)[①]。

验证这些库已经建立,且共享的目标文件(例如,libdataobjects.so[②])位于 CPPLIBS 目录下。下面列出的是一个典型的 Linux 系统下 libs 目录下的内容:

```
libs> ls -l
drwxr-xr-x  5 dataobjects
lrwxrwxrwx  1 libactioneditor.so -> libactioneditor.so.1.0.0
lrwxrwxrwx  1 libactioneditor.so.1 -> libactioneditor.so.1.0.0
lrwxrwxrwx  1 libactioneditor.so.1.0 -> libactioneditor.so.1.0.0
-rwxrwxr-x  1 libactioneditor.so.1.0.0
[...]
lrwxrwxrwx  1 libdataobjects.so -> libdataobjects.so.1.0.0
lrwxrwxrwx  1 libdataobjects.so.1 -> libdataobjects.so.1.0.0
lrwxrwxrwx  1 libdataobjects.so.1.0 -> libdataobjects.so.1.0.0
-rwxr-xr-x  1 libdataobjects.so.1.0.0
-rw-r--r--  1 libs.pro
-rw-r--r--  1 Makefile
libs>
```

以 drwxr-xr-x 开头的行是目录,以 lrwxrwxrwx 开头的行是符号链接。可以通过搜索引擎找出为什么每一个共享目标文件都有三个符号链接的答案。

固定链接器路径

- 用合适的语法,更新 shell/环境变量 LD_LIBRARY_PATH(*nix)或者 PATH(win32),以包含 CPPLIBS。
- 创建一个 projects/tests 目录,它就是保存用于测试各种库组件代码的地方。
- 运行随 libs tarball 文件一起提供的测试程序,它们位于与所建立库对应的 libs/tests 子目录下。

① 在建立某些库之前,可能还需要安装一下专门的 Qt 包。例如,phononmetadata 库就要求 libphonon-dev 包。链接器会提示这些依赖性。

② 在 Windows 系统下为 libdataobjects.lib 和 dataobjects.dll,后者必须位于 PATH 中列出的目录下,所以一定要在 PATH 中包含%CPPLIBS%。

注意

在*nix 平台上，通常用一个 shell 脚本来定义环境变量。示例 7.3 中给出了处理这项工作的一个 bash shell 脚本。

示例 7.3　src/bash/env-script.sh

```
export CPPLIBS=$HOME/oop/projects/libs
export LD_LIBRARY_PATH=$CPPLIBS
```

应注意这个脚本中的 bash 语法细节：

- 环境变量 HOME 包含到个人主目录的绝对路径。也可以使用符号“~”。
- 位于等于号左边的环境变量没有美元符号前缀。
- 位于等于号右边的环境变量必须有美元符号前缀。
- 如果环境变量是环境的一部分，而不仅仅是针对这个脚本文件的，则需要使用 export 命令。

输入如下两种命令之一，都可以运行这个脚本：

```
source env-script.sh
```

或者

```
. env-script.sh
```

注意后一个命令开始处的点号。在 bash shell 中，点号与 source 命令等价。

如果希望在启动每一个 shell 时能够被自动设置，则可以将这个脚本放入~/.bashrc，只要启动 bash(例如，启用终端或者控制台)，它就会自动运行。

关于如何编写 shell 脚本，Hamish Whittal 提供了一个很好的在线指南[①]。

7.3　框架与组件

如何对类进行组织，已经超出了简单的继承的范围。对这种框架进行仔细设计，可以使查找和复用组件变得更加容易。所有大型软件工程都建立在框架的基础之上，这里将探讨一些当今流行的框架。

在现代编程技术中，代码复用具有最高优先级。过去，计算机时间较为昂贵而程序员时间相对便宜，现在，形势逆转了。如今，所有的软件都是基于块构建的，块本身也可以看成是一个软件。几乎没有从头开始进行应用设计的，因为这样做是将程序员时间浪费在重复设计和实现那些已经由公认的专家设计、实现、优化和测试过的模块。

框架是一个(通常非常大的)通用(或者针对特定领域的)类与约定的集合，其目的是提高设计的一致性。框架通常被用来创建图形化应用、数据库应用或者其他复杂的软件。

框架一般都具有文档丰富的公共 API。API 是库中公共函数、类和接口的描述。为了实现框架，可以采用设计模式。用设计模式进行开发涉及寻找合适的对象和可能的层次。所使用的类和模式都采用富于描述性的名称，这样可以真正做到一次定义、处处复用。7.3 节中将简要讨论设计模式。

① 参见 http://www.cc.puv.fi/~jd/course/Linux/Shell_Scripting/index.html。

Qt 是许多开源的面向对象框架中的一种，它提供一组可复用的组件，用于创建跨平台的应用。其他一些知名的类似工具如下。

- boost[①]，一种开源的跨平台 C++工具类库。
- mono[②]，Microsoft.NET 的一种开源实现，它是用于 C#的 API，以 libgtk 为基础而创建。
- libgtk, libgtk++，定义了一些窗件的库，这些窗件用于 Gnome 桌面、Mozilla、Dia、GAIM、GIMP、Evolution、OpenOffice 以及许多其他的开源程序。
- wxWidgets[③]，另一种 C++跨平台的窗件工具集。
- Wt[④]，一种类似于 Qt 的框架，用于用 boost 和 AJAX[⑤]创建 Web 应用。

利用 Qt 这样的多平台框架，就可以从其他人的创造性工作中获得大量的好处。严格使用 Qt 创建的软件，其基础是将那些已经在 Windows, Linux 和 Mac OS/X 上经过大量程序员测试过的组件。

Qt(以及跨平台的 Gnu ToolKit，Gtk++)这样的工具集，在不同平台下有不同的实现。这就是为什么基于 Qt 的应用在 Linux 上看起来与 KDE 应用类似、在 Windows 上看起来与 Windows 应用类似的原因。

7.4 设计模式

对于面向对象的软件设计所面对的共同问题而言，设计模式是一种高效且精炼的解决方案。设计模式是一种高度抽象的模板，可以将它们应用到不同类型的设计问题。

在 Erich Gamma, Richard Helm, Ralph Johnson 和 John Vlissides(他们经常被戏称为“四人组”)所著的图书 *Design Patterns*[Gamma95]中，分析了 23 种特定的模式。每一种模式都用专门的一个小节进行讲解，描述了下面这些事情：

- 模式名称。
- 对可以使用该模式的某种问题的描述。
- 对关于设计问题以及如何获得解决方案的抽象描述。
- 应用该模式后能够得到什么结果以及关于得失的探讨。

设计模式可用于许多不同场合，其中的大多数都描述如何根据职责来区分代码。这些模式被分成三种类别：创建模式(Creational)、结构模式(Structural)和行为模式(Behavioral)。结构模式描述如何组织对象并连接它们；行为模式描述如何组织代码；创建模式描述如何组织那些用于管理对象创建的代码。

“四人组”宣称：设计模式是“在特定环境下用于解决某类设计问题的类与对象间通信关系的描述”。当在后面继续用 Qt 开发应用时，将会看到几个设计模式的描述和示例。

当使用更多流行的编程语言和 API 时，就会遇到工作中常用的一些设计模式。另一方面，类似于 Python 之类的现代的、动态的编程语言，已经具有对几种设计模式的内置支持，所以

① 参见 http://www.boost.org。
② 参见 http://www.mono-project.com/。
③ 参见 http://www.wxwidgets.org。
④ 参见 http://www.webtoolkit.eu/wt。
⑤ AJAX 是 Asynchronous JavaScript And XML 的首字母缩写，一种 JavaScript 和 XML 的客户系统，在 Web 页面中提供列表/树/表格视图且具有 GUI 行为。

在软件中实现它们是学习这种语言的一种天然收益。Qt 中也是如此，后面将会遇到几种 Qt 类，它们都实现了一些流行的设计模式。

7.4.1　序列化器模式：QTextStream 和 QDataStream

序列化器模式

序列化器(serializer)是一种只负责读取或者写入对象的对象。Qt 的 QTextStream 序列化器用于读写人可读的文件，而 QDataStream 序列化器用于读写结构化的二进制数据。这些类是用序列化器模式实现的，被用于 C++和 Qt 中[Martin98]。

利用 QDataStream，就可以序列化和解序列化(deserialize) QVariant 支持的全部类型，包括 QList, QMap, QVector 以及其他类型[①]。利用 QTextStream，如果希望在定制的类型上将提取运算符(>>)用于插入运算符(<<)的输出，则必须为字符串类型定义正确的字段和记录分隔符，并要用样本数据测试这些运算符的正确性。

由于这两种流都可以从 QIODevice 创建，且存在许多使用 QIODevice 进行通信的其他 Qt 类，所以这两种运算符能够将对象发送到网络、管道或者数据库。

示例 7.4 给出了 MetaDataValue 的输入/输出运算符函数的友元声明，MetaDataValue 是用来存储歌曲元数据的一个类。这些运算符不是成员函数，因为它的左操作数是一个类无法修改的 QDataStream 或者 QTextStream。此外，运算符还不能是 MetaDataValue 的成员函数，因为序列化器模式的思想是将 I/O 代码与类本身分开。METADATAEXPORT 宏有利于在 Windows 平台上复用这些代码[②]。

示例 7.4　src/libs/metadata/metadatavalue.h

```
[ . . . . ]
class METADATAEXPORT MetaDataValue {
public:

     friend METADATAEXPORT QTextStream& operator<< (QTextStream& os,
                                          const MetaDataValue& mdv);
     friend METADATAEXPORT QTextStream& operator>> (QTextStream& is,
                                          MetaDataValue& mdv);
     friend METADATAEXPORT QDataStream& operator<< (QDataStream& os,
                                          const MetaDataValue& mdv);
     friend METADATAEXPORT QDataStream& operator>> (QDataStream& is,
                                          MetaDataValue& mdv);
     friend METADATAEXPORT bool operator==(const MetaDataValue&,
                                          const MetaDataValue&);
[ . . . . ]
    virtual QString fileName() const ;
    virtual Preference preference() const ;
    virtual QString genre() const;
    virtual QString artist() const;
    virtual QString albumTitle() const;
    virtual QString trackTitle() const;
    virtual QString trackNumber() const;
```

① 参见 http://doc .qt.nokia.com/latest/datastreamformat.html。

② 7.1.2 节的“提示”小节中探讨过这类宏。

```
    virtual const QImage &image() const;
    virtual QTime trackTime() const;
    virtual QString trackTimeString() const;
    virtual QString comment() const;
[ . . . . ]
protected:
    bool m_isNull;
    QUrl m_Url;
    QString m_TrackNumber;
    QString m_TrackTitle;
    QString m_Comment;
    Preference m_Preference;
    QString m_Genre;
    QString m_Artist;
    QTime m_TrackTime;
    QString m_AlbumTitle;
    QImage m_Image;
};
Q_DECLARE_METATYPE(MetaDataValue);                                    1
[ . . . . ]
```

1 添加到 QVariant 类型系统。

每一个运算符都处理一个 MetaDataValue 对象。operator<<()将它的数据插入到输出流中，operator>>()从输入流中提取它的数据。每一个运算符都返回一个左操作数的引用，以便能够进行链式操作。

示例 7.5 中的 QTextStream 运算符需要考虑到空白符和分隔符，因为进行流式输出时全部内容都是以字符串表示的。

示例 7.5 src/libs/metadata/metadatavalue.cpp

```
[ . . . . ]


QTextStream& operator<< (QTextStream& os, const MetaDataValue& mdv) {
    QStringList sl;
    sl << mdv.url().toString() << mdv.trackTitle() << mdv.artist() << mdv.
albumTitle()
            << mdv.trackNumber() << mdv.trackTime().toString("m:ss")
            << mdv.genre() << mdv.preference().toString() << mdv.comment();
    os << sl.join("\t") << "\n";                                      1
    return os;
}

QTextStream& operator>> (QTextStream& is, MetaDataValue& mdv) {
    QString line = is.readLine();
    QStringList fields = line.split("\t");                            2
    while (fields.size() < 9) {
        fields << "";
    }
    mdv.m_isNull = false;
    mdv.setUrl(QUrl::fromUserInput(fields[0]));
    mdv.setTrackTitle(fields[1]);
```

```
    mdv.setArtist(fields[2]);
    mdv.setAlbumTitle(fields[3]);
    mdv.setTrackNumber(fields[4]);
    QTime t = QTime::fromString(fields[5], "m:ss");
    mdv.setTrackTime(t);
    mdv.setGenre(fields[6]);
    Preference p(fields[7]);
    mdv.setPreference(p);
    mdv.setComment(fields[8]);
    return is;
}
```

1 输出到 TSV(制表符分隔的值)。

2 以 TSV 读取。

示例 7.6 中的 QDataStream 运算符使用起来要简单得多，因为它们不需要程序员来区分数据项。

示例 7.6　src/libs/metadata/metadatavalue.cpp

```
[ . . . . ]

QDataStream& operator<< (QDataStream& os, const MetaDataValue& mdv) {
    os << mdv.m_Url << mdv.trackTitle() << mdv.artist() << mdv.albumTitle()
            << mdv.trackNumber() << mdv.trackTime() << mdv.genre()
            << mdv.preference() << mdv.comment() << mdv.image();
    return os;
}

QDataStream& operator>> (QDataStream& is, MetaDataValue& mdv) {
    is >> mdv.m_Url >> mdv.m_TrackTitle >> mdv.m_Artist >> mdv.m_AlbumTitle
            >> mdv.m_TrackNumber >> mdv.m_TrackTime >> mdv.m_Genre
            >> mdv.m_Preference >> mdv.m_Comment >> mdv.m_Image;
    mdv.m_isNull= false;
    return is;
}
```

示例 7.7 中给出了如何将这些运算符用于不同的流。使用 QDataStream 时唯一的缺点是结果文件为二进制的(即人不可读的)。

示例 7.7　src/serializer/testoperators/tst_testoperators.cpp

```
[ . . . . ]
void TestOperators::testCase1()
{
    QFile textFile("playlist1.tsv");
    QFile binaryFile("playlist1.bin");
    QTextStream textStream;
    QDataStream dataStream;

    if (textFile.open(QIODevice::ReadOnly)) {
        textStream.setDevice(&textFile);
    }
    if (binaryFile.open(QIODevice::WriteOnly)) {
        dataStream.setDevice(&binaryFile);
```

```
    }
    QList<MetaDataValue> values;
    while (!textStream.atEnd()) {
        MetaDataValue mdv;
        textStream >> mdv;                                    1
        values << mdv;                                        2
        dataStream << mdv;                                    3
    }
    textFile.close();
    binaryFile.close();
    textFile.setFileName("playlist2.tsv");
    if (binaryFile.open(QIODevice::ReadOnly)) {
        dataStream.setDevice(&binaryFile);
        for (int i=0; i<values.size(); ++i) {
            MetaDataValue mdv;
            dataStream >> mdv;                                4
            QCOMPARE(mdv, values[i]);                         5
        }
    }
}
[ . . . . ]
```

1 以 TSV 读取。

2 添加到列表。

3 写入 `binaryFile`。

4 读取二进制数据。

5 它与以前读取的相同吗?

7.4.2 反模式

反模式(antiPattern)最早是由[Koenig95]提出的一个术语，它常用来描述那些已经被证明是没有效果的、低效率的或者达不到预期目标的编程实践。对于挑选出来的一些反复出现的问题，当其解决方案在学生以及无经验的程序员之间传递时，几种反模式就出现了。在Wikipedia[①]中有关于这一主题的一篇信息丰富、讲解透彻的文章，它罗列并简要描述了大量的反模式。研究一些反模式的例子，可以帮助程序员避免类似的陷阱。以下是来自于 Wikipedia 文章 antiPatterns 的一小部分内容。

- 软件设计反模式
  - 输入杂乱：没有指定和实现如何对可能的无效输入进行处理。
  - 接口膨胀：接口的功能强大而复杂，以至于难以复用或者实现。
  - 竞争风险：没有注意事件的不同顺序所造成的后果。
- 面向对象设计反模式
  - 循环依赖性：在对象或者软件模块之间引入了不必要的直接或者间接彼此依赖性。
  - “上帝”对象：具有太多信息或者太多责任的对象。这可能是由于在一个类中包含太多函数而导致的。这种对象可以在许多种情况下出现，但经常发生的情形是：将模型和视图的代码组合在同一个类中。

① 参见 http://en.wikipedia.org/wiki/AntiPattern#Programming_antiPatterns。

- 编程反模式
  - 难以编码：在实现中嵌入了关于系统环境的假设。
  - 魔幻数字：算法中包含未解释的数字。
  - 魔幻字符串：代码中包含直接的字符串作为事件类型用于比较。
- 方法学反模式
  - 复制—粘贴编程：复制并修改已有的代码，而不是创建更通用的解决方案。
  - 一切从头开始：不采用已有的解决方案，而是采用(执行起来表现要差得多的)定制的解决方案。

图 7.3 中，Customer 包含的成员函数用于以 XML 格式导入和导出它的每一个数据成员。

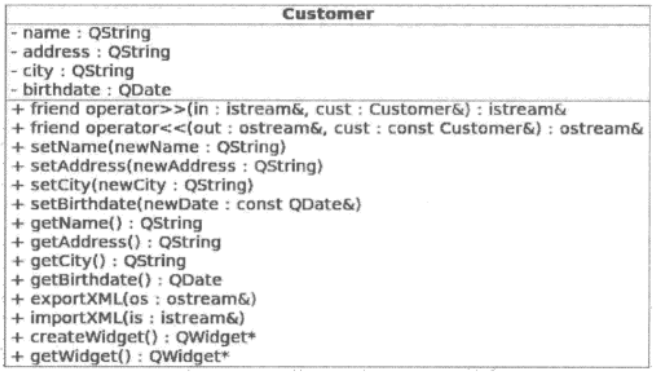

图 7.3　反模式示例

getWidget()提供了一个特殊的 GUI 窗件，用户可以用它来从图形化应用输入数据。此外，还存在几个通过 iostream 进行输入/输出的友元函数[①]。

这个类是一个模型，因为它拥有数据并表示一些抽象实体。但是，这个类也包含了一些视图代码，因为存在 createWidget()和 getWidget()成员函数。此外，它还包含专门针对特定 I/O 流的序列化代码[②]。如此看来，这个类对数据模型承担了太多的责任。这是接口膨胀(以及其他)反模式的一个例子。

当实现其他的数据模型类，比如 Address，ShoppingCart，Catalog，CatalogItem 等，接口膨胀反模式导致的问题就会立即出现。这些类还需要如下这些方法：

- createWidget()
- importXML()
- exporXML()
- operator<<()
- operator>>()

这会导致另一种反模式“复制—粘贴编程”的使用。一旦改变了数据结构，就需要相应改变全部的数据表示以及 I/O 方法。当维护这样的代码时，就有可能出现 bug。如果 Customer 是反射的(reflective)，则意味着它对自己的成员具有判断能力(例如，有多少个属性？属性的

① 2.6 节中探讨了友元。

② 序列化是一种将对象的数据转换成某一种形式的过程，这种形式使得数据能够保存到文件中或者通过网络传输，以便今后可以重构这个对象。7.3.1 节中探讨了序列化器模式。

名称是什么？属性的类型是什么？如何加载/存储这些属性？子对象是什么？)，然后，就可以定义一种通用的方法来读写用于 Customer 的对象以及任何其他相似的反射类。第 12 章中探讨了一个例子，它给出了如何用反射编写更为通用的代码。

7.4.2.1 关于设计模式的更多探讨

有数不清的 Web 站点探讨并给出了关于设计模式的例子。除了 Wikipedia 之外，还可以访问如下的站点：

- Vince Huston 的 Web 页面[①]。
- Douglas Schmidt 的设计模式教程[②]。
- Wiki 图书 C++ Programming/Code/Design Patterns[③]的站点。

术语“反模式”已经被 Pattern Community(模式社区)[④]采纳，这个社区维护着一个反模式的目录[⑤]。

7.5 复习题

1. 什么是平台？你使用的是什么平台？在你的学校或者工作地有哪些平台？
2. 什么是代码复用？如何实现它？其优点是什么？
3. 编译器的角色是什么？
4. 链接器扮演的角色是什么？
5. 给出三种 C++库的名称。
6. 什么是 CPPLIBS？为什么需要它？
7. 对下面的每一个项，判断它是否会正常出现在某个头文件(.h)或者某个实现文件(.cpp)中，并给出理由。
   a. 函数定义
   b. 函数声明
   c. static 对象声明
   d. static 对象定义
   e. 类定义
   f. 类声明
   g. inline 函数定义
   h. inline 函数声明
   i. 默认实参指示符
8. 编译时依赖性与链接时依赖性有什么不同？
9. 什么是框架？你在使用它吗？
10. 什么是设计模式？最常见的设计模式有哪些？
11. 什么是反模式？

① 参见 http://www.vincehuston.org/dp/。
② 参见 http://www.cs.wustl.edu/~schmidt/tutorials-patterns.html。
③ 参见 http://en.wikibooks.org/wiki/C++_Programming/Code/Design_Patterns。
④ 参见 http://c2.com/cgi/wiki? PatternCommunity。
⑤ 参见 http://c2.com/cgi/wiki? AntiPatternsCatalog。