第16章　更多的设计模式

本章将介绍来自于两个类别的设计模式：创建模式(creational)和结构模式(structural)。

16.1　创建模式

使用管理对象创建模式，可以获得某种灵活性，使得在运行时能够挑选或者改变所创建或使用的对象的种类，并可用来管理对象的删除，尤其是在大型软件系统中更是如此。正确管理对象的创建，是管理不同的代码层的关键部分，也是确保对象可被正确销毁的重要一环。

C++中，工厂是一种程序组件(通常为一个类)，它负责对象的创建。工厂的思想是将对象创建与对象使用分离开来。

工厂类通常具有一个函数，它获得用于新对象的动态内存，并返回一个指向这个对象的(基类)指针。这种方法使得能够在不改变使用工厂的代码的情况下引入新的派生类型。

这一节将探讨几种使用工厂的设计模式，并会给出这些设计模式的一些例子。

当创建堆对象的责任委托给一个虚函数时，就称它是一个工厂方法。

如果让工厂方法为纯虚方法，并且编写了具体的派生工厂类，就使工厂基类成为了抽象工厂。

> 抽象工厂模式提供的接口用于定义一些工厂，它们共享抽象的特性但有不同的具体细节。客户代码可以实例化某个特定的子工厂，然后用抽象接口来创建对象。

通过强制使用防止直接实例化类的创建规则，就可以迫使客户代码只能使用工厂方法来创建所有的实例。例如，可能需要限制一次可以存在的某个特定类的对象数量。事实上，所用到的类只被严格地实例化一次是相当常见的。

> 单一模式(singleton pattern)可将类限制成只能创建一个实例。实现方式是让其构造函数为 `private` 类型或者 `protected` 类型，并且提供一个 `instance()` 成员函数。如果类的实例不存在，则这个成员函数返回一个指向新实例的指针，否则返回一个指向已存在实例的指针。

8.3 节中已经看到过并使用过单一模式。使用事件循环的 Qt 应用会首先实例化 `QApplication`，创建另外一些对象，然后调用 `QApplication::exec()`，启动事件循环。为了在程序的其他地方引用这个对象，可以使用 `qApp` 宏，它会返回一个指向这个单一 `QApplication` 模式的指针[①]。16.1.4 节中将更进一步探讨单一模式。

对象创建

考虑如下创建派生的 `QObject` 类 `Customer` 实例的三种方式：

---

① `QApplication` 派生自 `QCoreApplication`。当包含<QApplication>时，就是包含了宏`#define qApp (static_cast< QApplication *>(QCoreApplication ::instance()))`。`QCoreApplication: :instance()` 返回一个已定义实例的指针，或者如果没有创建实例的话，则返回 `null`。

- `Customer* c1 = new Customer(name);`
- `QMetaObject meta = Customer::staticMetaObject; Customer* cust = qobject_cast<Customer*>(meta.newInstance());`
- `Customer* cust = CustomerFactory::instance()->newCustomer(name);`

第一种情形中，类名称被硬编码到构造函数调用中，对象将在内存中用默认的堆存储创建。在客户代码中硬编码类名称，会限制代码的可复用性和灵活性。

第二种情形中，调用的是 QMetaObject::newInstance()。这是一个抽象工程，在每一个 QMetaObject 派生类中都会被重写，由 moc.newInstance()返回的 QObject 指向 Customer 的一个新实例。

第三种情形中，使用一个名称为 CustomerFactory::newCustomer()的专门化工厂方法，间接构造了一个 Customer 对象。与 Qt 库相比，这个接口使用起来会更加方便，但它只不过是另一个工厂的封装器。

在 QMetaObject::newInstance()之前[①]，有必要编写一条 switch 语句来处理工厂中所支持的每一个类。

16.1.1 抽象工厂

AbstractFactory(抽象工厂)的定义见示例 16.1，它是一个简单类。

示例 16.1 src/libs/dataobjects/abstractfactory.h

```
[ . . . . ]
class DOBJS_EXPORT AbstractFactory
{
  public:
    virtual QObject* newObject (QString className,
                                QObject* parent=0) = 0;
    virtual ~AbstractFactory();

};
[ . . . . ]
```

newObject()为一个纯虚方法，因此必须在派生类中重写它。示例 16.2 中给出了一些派生自 AbstractFactory 的具体类。

示例 16.2 src/libs/dataobjects/objectfactory.h

```
[ . . . . ]
class DOBJS_EXPORT  ObjectFactory : public AbstractFactory {
  public:
    ObjectFactory();
    virtual QObject* newObject (QString className, QObject* parent=0);
 protected:
    QHash<QString, QMetaObject> m_knownClasses;
};
[ . . . . ]
```

① 它是在 Qt 4.5 中引入的。

程序员网址导航www.daocode.com

QMetaObject::newInstance()

通过将某个构造函数标记为 Q_INVOKABLE，就可以用 QMetaObject::newInstance() 创建派生自 QObject 类的实例。QMetaObject::newInstance() 方法本身也是抽象工厂模式的一个例子。

前面已经定义了 ObjectFactory，这样它就知道如何创建两种具体类型。由于能够提供给 newObject() 的可以是任何 QString，所以 ObjectFactory 需要处理传递的类未知的情况。这种情况下，它返回一个泛型 QObject 的指针并会设置动态属性 className，以便其他函数（例如，XML 导出例程）知道这个对象正在“欺骗”另一个类中的对象。

示例 16.3　src/libs/dataobjects/objectfactory.cpp

```
[ . . . . ]

ObjectFactory::ObjectFactory() {
    m_knownClasses["UsAddress"] = UsAddress::staticMetaObject;
    m_knownClasses["CanadaAddress"] = CanadaAddress::staticMetaObject;
}

QObject* ObjectFactory::newObject(QString className, QObject* parent) {
    QObject* retval = 0;
    if (m_knownClasses.contains(className)) {
        const QMetaObject& mo = m_knownClasses[className];
        retval = mo.newInstance();                                    1
        if (retval == 0) {
            qDebug() << "Error creating " << className;
            abort();
        }
    } else {
        qDebug() << QString("Generic QObject created for new %1")
                    .arg(className);
        retval = new QObject();
        retval->setProperty("className", className);
    }
    if (parent != 0) retval->setParent(parent);
    return retval;
}
```

1 要求 Qt 4.5 或者更高的版本。

示例 16.4 中给出的例子将 Q_INVOKABLE 宏用于 UsAddress 类的一个构造函数中。

示例 16.4　src/libs/dataobjects/address.h

```
[ . . . . ]

class DOBJS_EXPORT UsAddress : public Address {
    Q_OBJECT
  public:
    Q_PROPERTY( QString State READ getState WRITE setState );
    Q_PROPERTY( QString Zip READ getZip WRITE setZip );
    explicit Q_INVOKABLE UsAddress(QString name=QString(), QObject* parent=0)
```

```
                                    : Address(name, parent) {}
  protected:
    static QString getPhoneFormat();
  public:
[ . . . . ]

  private:
    QString m_State, m_Zip;
};
```

16.1.2 抽象工厂和库

现在考虑两个库：libdataobjects 和 libcustomer，它们都具有自己的(具体)对象工厂，如图 16.1 中的 UML 框图所示。示例 16.5 中定义的 CustomerFactory 扩展了 ObjectFactory 的功能性。

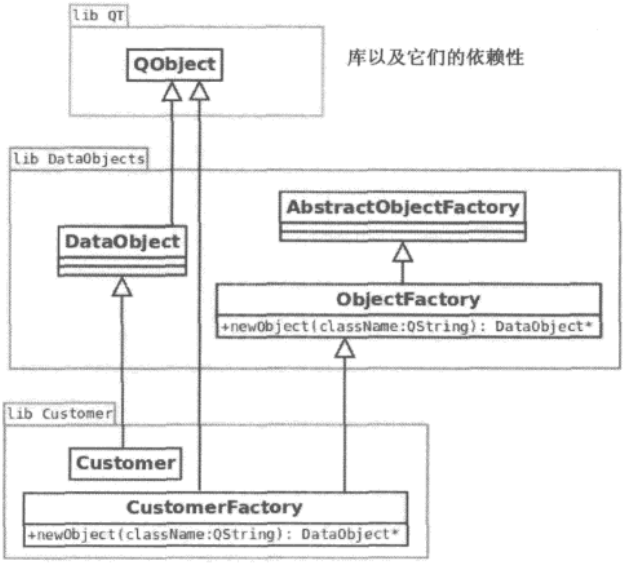

图 16.1 库与工厂

示例 16.5 src/libs/customer/customerfactory.h

```
[ . . . . ]
class CUSTOMER_EXPORT CustomerFactory :
                            public QObject, public ObjectFactory {
  public:
    static CustomerFactory* instance();                          1
    Customer* newCustomer(QString name, QObject* parent=0);      2
    Address* newAddress(QString countryType = "USA", QObject* parent=0);
  private:
    CustomerFactory(QObject* parent=0);
};
[ . . . . ]
```

1 单一工厂方法。

2 常规的工厂方法，不要求类型转换。

程序员网址导航www.daocode.com

CustomerFactory 从 ObjectFactory 继承了创建 Address 对象的能力。此外，它还知道如何创建 Customer 对象。CustomerFactory 只需要在构造函数中将一些 QMetaObject 添加到 ObjectFactory::m_knownClasses 中(这是可能的，因为这个容器为 protected 类型)。基类 newObject()方法足够聪明，能够处理由 Customer 添加的类，只要它们被正确地在容器中被注册了。

示例 16.6　src/libs/customer/customerfactory.cpp

```
[ . . . . ]


CustomerFactory::CustomerFactory(QObject* parent) : QObject(parent) {
    m_knownClasses["Customer"] = Customer::staticMetaObject;
    m_knownClasses["CustomerList"] = Customer::staticMetaObject;
}
```

16.1.3　qApp 和单一模式

正如以前所讨论的，单一模式是一种专门化的工厂，用于希望限制所创建的实例数量或者类型的情形。

示例 16.7 中定义的 CustomerFactory::instance()方法，就是单一模式的一个例子。在需要时它会创建一个对象，但是只有在首次调用这个方法时才会这样做。对它的后续调用，会总是返回一个指向同一对象的指针。

示例 16.7　src/libs/customer/customerfactory.cpp

```
[ . . . . ]

CustomerFactory* CustomerFactory::instance() {
    static CustomerFactory* retval = 0;
    if (retval == 0) retval = new CustomerFactory(qApp);          1
    return retval;
}
```

1 当 QApplication 退出时，应确保这个对象以及它的全部子对象都被清除了。

当处理堆对象时，千万注意不要产生内存泄漏。可以利用 QObject 的父-子关系来防止这种情况出现。

正如前面提到的，qApp 是一个指向 QApplication 单一实例的指针，这个实例假定是在 main()中创建的。QApplication 实例的存在时间与应用运行的时间一致。

为什么不使用 static 父对象

如果堆对象的父对象为 static QObject，则它的子对象将在 QApplication 被销毁之后才销毁。除非存在足够的理由，否则在销毁 QApplication 之后，程序不应当对 QObject 有任何动作，包括对象清理。对于使用多个文件的应用程序而言，来自于不同文件的 static 对象是按照它们的链接器依赖性顺序被销毁的。这种销毁顺序可能导致无意的副作用(例如，终止时的段错误)。更多细节，请参见 8.6 节。

16.1.4 使用工厂的好处

工厂模式的好处之一是：可以在一个池(pool)中管理所创建的对象(复用可以被复用的那些对象)。

间接的对象创建还可以在运行时决定要创建哪些类对象。这样就使得可以进行替换类的“插入”，而不必要求客户代码有任何改变。16.2 节中给出的一个方法示例，它根据 XML 文件的内容使用工厂对象来创建链接的、客户定义的对象树。在 src/libs/metadata/abstractmetadataloader.h 中有工厂方法的另一个示例，它管理 MetaDataLoader 的单一实例，从而不必改变代码就能够轻易编写出在派生的元数据加载器之间切换的程序。

库和插件

建立大型系统时，一种好的做法是将库设计成包含共享某些共性或者需要一起使用的类。大量的应用都会使用来自于多个库的组件，这些库中的一些是由开发团队提供的，而另一些是由第三方开发人员提供的（例如，诺基亚的 Qt）。只有库类的公共接口才能出现在复用它们的客户代码中。库设计人员在改动库类的实现时，不应当导致客户代码的破坏。

插件（plugin）是一种集成的软件组件集，它为大型应用添加特定的功能。插件的一个例子是 Adobe Flash Player，它使得各种 Web 浏览器都能够显示某些类型的视频。好的插件支持使得用户能够定制应用的功能性。许多库通过发布接口、提供如何实现的文档，从而能够在类之外被插入。通过提供一个工厂基类，专门化的工厂类根据需要从这个工厂基类派生，这样就使得库方便了插件类的创建。

工厂方法的另一个好处（即通常所说的间接对象创建）是：可以强制对象在构造函数之后被初始化，包括 virtual 函数（虚函数）的调用。

来自于构造函数的多态

在构造函数完成执行之前，对象不会被认为是已经“完全构造的”。在构造函数执行完毕之前，对象的虚指针(vpointer)不会指向正确的虚表(vtable)。因此，从构造函数对这种对象调用方法不能使用多态。

当在对象初始化期间需要多态行为时，就要求使用工厂方法。示例 16.8 中就是这种情况。

示例 16.8 src/ctorpoly/ctorpoly.cpp

```
#include <iostream>
using namespace std;

class A {
public:
    A() {
        cout << "in A ctor" << endl;
        foo();
    }
    virtual void foo() {
        cout << "A's foo()" << endl;
    }
};
```

程序员网址导航www.daocode.com

```
class B: public A {
public:
    B() {
        cout << "in B ctor" << endl;
    }
    void foo() {
        cout << "B's foo()" << endl;
    }
};

class C: public B {
public:
    C() {
        cout << "in C ctor" << endl;
    }

    void foo() {
        cout << "C's foo()" << endl;
    }
};

int main() {
    C* cptr = new C;
    cout << "After construction is complete:" << endl;
    cptr->foo();
    return 0;
}
```

其输出见示例 16.9。

示例 16.9　src/ctorpoly/ctorpoly-output.txt

```
src/ctorpoly> ./a.out
in A ctor
A's foo()
in B ctor
in C ctor
After construction is complete:
C's foo()
src/ctorpoly>
```

值得注意的是，当构造一个新的 C 对象时，调用的是错误的 foo()版本。22.1 节中将详细探讨虚表。

16.1.5　练习：创建模式

1. 完成 Address，Customer 以及 CustomerList 类的实现。
   将 7.4.1 节中讲解的思想用于编写 CustomerWriter 类。确保使用了 Customer 和 Address 的 Q_PROPERTY 特性，以便当改变 Customer 类的实现时不必重写 CustomerWriter 类。
   要记住的是，Address 对象是作为 Customer 的子对象保存的。以下是值得考虑的一种输出格式：

```
Customer {
   Id=83438
   DateEstablished=2004-02-01
   Type=Corporate
   objectName=Bilbo Baggins
   UsAddress {
       Line1=52 Shire Road
       Line2=Suite 6
       City=Brighton
       Phone=1234567890
       State=MA
       Zip=02201
       addressName=home
   }
}
```

如果使用 Dataobject ::toString()函数，则会出现另一种情况。

2. 编写一个 CustomerReader 类，它通过复用所提供的 CustomerFactory 类来创建全部的新对象。

   编写一个 CustomerListWriter 类和一个 CustomerListReader 类，它们分别序列化和解序列化 Customer 对象的列表。编写它们时能够复用多少 CustomerWriter 类和 CustomerReader 类的代码?

   编写客户代码，测试这些类。

16.2 备忘录模式

这一节将 QMetaObject 与 SAX2 解析器结合起来，以展示如何编写通用的 XML 编码/解码工具，让其用于带有定义良好的 Q_PROPERTY 的 QObject 及其子对象。这样就可以将元对象模式与备忘录模式(Memento 模式)结合在一起使用。而且由于 XML 和 QObject 都能够表示层次结构，所以可以将这些想法与组合(Composite)模式以及抽象工厂模式结合在一起，以保存和加载全部的多态对象树。

这一节的目标是为许多不同类型的类提供序列化器和解序列化器，其中的编码和解码是由能够操作 QMetaObject 的方法处理的，处理时按照模型逻辑进行区分。

为了将 QObject 树编码和解码成 XML，必须定义一个映射模式，这种映射不仅必须获得 QObject 的属性、类型和值，而且还要知晓对象及其子对象之间已有的关系，每一个子对象以及它们的子对象之间的关系，等等。

XML 元素之间的父-子关系可以自然地映射成 QObject 的父-子关系。这些关系就定义了一种树结构。

前面说过，XML 和 QObject 都已经使用了组合模式，以支持对象的树状层次。图16.2 中给出的 Customer 和 CustomerList 都派生自 QObject。这里使用组合模式，将 QObject 子对象映射成 XML 中对应的子元素。

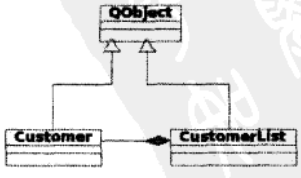

图 16.2 CustomerList 的 UML 框图

示例 16.10 中给出了一种所期望的 XML 格式，用于保存 CustomerList 的数据。

程序员网址导航www.daocode.com

示例 16.10　src/xml/propchildren/customerlist.xml

```
<object class="CustomerList" name="Customers" >

   <object class="Customer" name="Simon" >
     <property name="Name" type="QString" value="Simon" />
     <property name="Date" type="QDate" value="1963-11-22" />
     <property name="LuckyNumber" type="int" value="834" />
     <property name="State" type="QString" value="WA" />
     <property name="Zip" type="QString" value="12345" />
     <property name="FavoriteFood" type="QString" value="Donuts" />
     <property name="FavoriteDrink" type="QString" value="YooHoo"/>
   </object>
   <object class="Customer" name="Raja" >
     <property name="Name" type="QString" value="Raja" />
     <property name="Date" type="QDate" value="1969-06-15" />
     <property name="LuckyNumber" type="int" value="62" />
     <property name="State" type="QString" value="AZ" />
     <property name="Zip" type="QString" value="54321" />
     <property name="FavoriteFood" type="QString" value="Mushrooms" />
     <property name="FavoriteDrink" type="QString" value="Jolt" />
   </object>

</object>
```

如果输入文件包含的是这种类型的信息，则不仅完全可以重新构造对象的属性以及类型，而且还可以重新定义 CustomerList 中各种 QObject 之间的父-子关系树结构。

16.2.1　导出到 XML

示例 16.11 中给出了一个反射递归函数 toString()，它为每一个对象的属性构造一个字符串，然后对该对象的子对象进行迭代，递归地对每个子对象调用 toString()函数。

> **备忘录模式**
>
> 如果对象的内部状态被捕获且被固定以便今后能够恢复时，这就是备忘录模式的一种实现。

示例 16.11　src/libs/dataobjects/qobjectwriter.cpp

```
[ . . . . ]
QString QObjectWriter::
toString(const QObject* obj, int indentLevel) const {
    QStringList result;
    QString indentspace;
    indentspace.fill(' ', indentLevel * 3 );
    QString className = obj->metaObject()->className();
    QString objectName = obj->objectName();
    QStringList propnames = propertyNames(obj);

    foreach (const QString &propName, propnames) {
        if (propName == "objectName") continue;
        QVariant qv = obj->property(propName.toAscii());
```

```
        if (propName == "className") {
                className = qv.toString();
                continue;
        }
        const QMetaObject* meta = obj->metaObject();
        int idx = meta->indexOfProperty(propName.toAscii());
        QMetaProperty mprop = meta->property(idx);

        result <<
        QString("%1  <property name=\"%2\" type=\"%3\" value=\"%4\" />")
            .arg(indentspace).arg(propName).
            arg(qv.typeName()).arg(toString(qv, mprop));
    }
    /* Query over QObjects */
    if (m_children) {
        QList<QObject*> childlist =
                qFindChildren<QObject*>(obj, QString());

        foreach (const QObject* child, childlist) {
            if (child->parent() != obj) {
      //          qDebug() << "This is not my child!!";
                continue;
            }
            if (child != 0) {
                result << toString(child, indentLevel+1);
            }
        }
    }

    result.insert(0, QString("\n%1<object class=\"%2\" name=\"%3\" >")
        .arg(indentspace).arg(className).arg(objectName));
    result << QString("%1</object>\n").arg(indentspace);
    return result.join("\n");
}
[ . . . . ]
```

示例 16.11 中给出的 `toString()` 函数利用了 Qt 的属性和 `QMetaObject` 的便利性来反射类。迭代时，它会将每一行追加到 `QStringList` 的末尾。当迭代完成时，会关闭<object>。然后，通过调用 `QStringList::join("\n")`，可以快速获得返回的 `QString`。

16.2.2 导入具有抽象工厂的对象

先修课程

要求先阅读 15.2 节。

导入例程要比导出例程复杂得多，而且它具有几个有趣的特性。

- 它使用 SAX 解析器解析 XML。
- 它根据输入的情况来创建对象。

程序员网址导航www.daocode.com

- 对象的数量和类型以及它们之间的父-子关系必须根据文件中的信息重新构造。

示例 16.12 中包含了 QObjectReader 类的定义。

示例 16.12　src/libs/dataobjects/qobjectreader.h

```
[ . . . . ]
#include "dobjs_export.h"
#include <QString>
#include <QStack>
#include <QQueue>
#include <QXmlDefaultHandler>

class AbstractFactory;
class DOBJS_EXPORT  QObjectReader : public QXmlDefaultHandler {
  public:
    explicit QObjectReader (AbstractFactory* factory=0) :
                          m_Factory(factory), m_Current(0) { }
    explicit QObjectReader (QString filename, AbstractFactory* factory=0);
    void parse(QString text);
    void parseFile(QString filename);
    QObject* getRoot();
    ~QObjectReader();
    // callback methods from QXmlDefaultHandler
    bool startElement( const QString& namespaceURI,
                       const QString& name,
                       const QString& qualifiedName,
                       const QXmlAttributes& attributes );
    bool endElement(  const QString& namespaceURI,
                      const QString& localName,
                      const QString& qualifiedName);
    bool endDocument();
  private:
    void addCurrentToQueue();
    AbstractFactory* m_Factory;
    QObject* m_Current;
    QQueue<QObject*> m_ObjectList;
    QStack<QObject*> m_ParentStack;
};
[ . . . . ]
```

图 16.3 中给出了这个示例中各种类之间的关系。

QObjectReader 派生自 QXmlDefaultHandler，后者是用于 QXmlSimpleReader 的一个插件。AbstractFactory 是用于 QObjectReader 的一个插件。当创建 QObjectReader 时，必须将它应用到 ObjectFactory 或者 DataObjectFactory 的具体实例。

现在，QObjectReader 就完全与它能够创建的对象的特定类型分开了。为了将 QObjectReader 用于自己的类型，只需从 AbstractFactory 派生出一个工厂即可。

要注意的是，当阅读根据示例 16.10 中的 XML 输出文件构造对象的代码时，应从示例 16.13 开始。

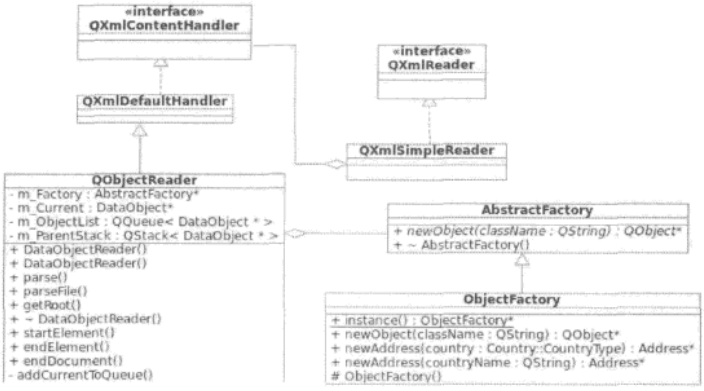

图 16.3 QObjectReader 以及它的相关类

示例 16.13 src/libs/dataobjects/qobjectreader.cpp

```
[ . . . . ]

bool QObjectReader::startElement( const QString&,
     const QString& elementName, const QString&,
     const QXmlAttributes& atts) {                          1
    if (elementName == "object") {
        if (m_Current != 0)                                 2
            m_ParentStack.push(m_Current);                  3
        QString classname = atts.value("class");
        QString instancename = atts.value("name");
        m_Current = m_Factory->newObject(classname);
        m_Current->setObjectName(instancename);
        if (!m_ParentStack.empty()) {                       4
            m_Current->setParent(m_ParentStack.top());
        }
        return true;
    }
    if (elementName == "property") {
        QString fieldType = atts.value("type");
        QString fieldName = atts.value("name");
        QString fieldValue = atts.value("value");
        QVariant qv = QVariant(fieldValue);
        m_Current->setProperty(fieldName.toAscii(), qv);

    }
    return true;
}
```

1 不需要命名未使用的参数。

2 判断是否在某个对象里。

3 将前一个流压入栈中。

4 ParentStack 栈顶或者前一个流应为父对象。

当 SAX 解析器遇到 XML 元素的初始标记时，就会调用 startElement() 函数。这个函数的参数包含创建对象所需的全部信息。在 startElement() 函数与对应的 endElement()

程序员网址导航www.daocode.com

函数之间遇到的所有其他对象，都是 m_Current 的子对象。当到达 endElement() 函数时，这个对象就算“完成”了，如示例 16.14 所示。

示例 16.14　src/libs/dataobjects/qobjectreader.cpp

```
[ . . . . ]

bool QObjectReader::endElement( const QString& ,
                                const QString& elementName,
                                const QString& ) {
    if (elementName == "object") {
        if (!m_ParentStack.empty())
            m_Current = m_ParentStack.pop();
        else {
            addCurrentToQueue();
        }
    }
    return true;
}
```

QObjectReader 使用抽象工厂来进行实际的对象创建工作。

回调函数 newObject(QString className) 创建的对象能够保存 className 中描述的全部属性。ObjectFactory 会为具有 QMetaObject 的类型正确地创建对象。对于其他的类，它会创建常规的 QObject，但会通过设置它的 className 动态属性来“模仿”这种类型。为了支持自己的类型，可以编写一个具体派生工厂，将正确的字符串映射成 QMetaObject。

16.3　Façade 模式

使用 Façade 模式的类“为子系统中的接口集提供一个统一的接口。Façade 定义的高级接口使得更容易(且更安全地)使用子系统”[Gamma95]。如果类接口由于太复杂而无法有效地使用时(导致难以调试错误)，或者没有使用与大型框架相适应的编程风格，则应使用 Façade 模式。Façade 模式的另一种用法是将使用某个特定库的代码与应用的其他部分区别开，以达到减少库间依赖性的目的。Façade 是一个类(或者类集)，它具有清晰、简单的接口，封装且隐藏了复杂的类集或者函数集。两种紧密相关的设计模式是封装器封装器(wrapper)和适配器(adaptor)，其中一种设计模式的某些例子，可以用另一种设计模式体现。

许多核心 Qt 类都是具有一些用于不同平台的原始类的 Façade 设计模式，它们的实现要比表象复杂得多。例如，QString 是一个具有可增长的、被隐式共享的字符数组的封装器；QWidget 是一个具有原始窗件的封装器；而 QThread，QFile，QProcess 和 QSqlDatabase 是具有低级库的封装器，这些库根据平台的情况有相当不同的实现。如果只通过 Qt API 使用这些封装器，则代码就可以运行于全部平台上。

从复用具有不同接口的类中获得的经验，能够为设计自己的类中成熟的、友好的、有用的接口提供有价值的帮助。下面的这个示例探讨了用于多媒体元数据系统的 Qt 封装器。

多媒体文件包含音视频内容：声音效果、音乐、图形图像、动画、电影等。元数据是描述多媒体文件音视频内容的信息。元数据经常被称为标签数据，或简称为标签。

多媒体文件通常将元数据保存在包含媒体的同一个文件中。通过标签信息，媒体播放器可以显示关于当前要播放或即将播放的文件的相关信息(例如，标题、艺术家、风格等)。也可以用它来将文件组织成集合，或者创建播放列表。

有许多库都能用了读取文件中的元数据。本书的前一个版本使用的是 id3lib，但它已经不再被支持了。现在使用的是 Phonon，它是在 Qt 4.4 中增加的。Phonon 支持多种文件格式，但是不能用于 Windows 下的 MP3 文件，而且它只能读取而不能回写标签数据。还有一个 Taglib 1.6 库，它与 id3lib 类似但支持现代的文件格式，且可用于 Windows, Linux 和 Mac OS/X 系统。许多开源工具都使用 TagLib，比如 Amarok 和 kid3。

本书中的代码已经被组织成根据应用的情况能够在 Phonon 和 TagLib 之间轻易切换。图 16.4 中给出了 MetaDataLoader 和 MetaDataValue 彼此相互适应的方式，只需在应用与所使用的底层元数据库之间提供一个 Façade 封装器即可。如果希望在程序中以某种方式动态地挑选其中之一(也许需根据平台的不同或者所使用的文件格式)，则可以进一步将 phononmetadata 和 filetagger 包装到插件中。

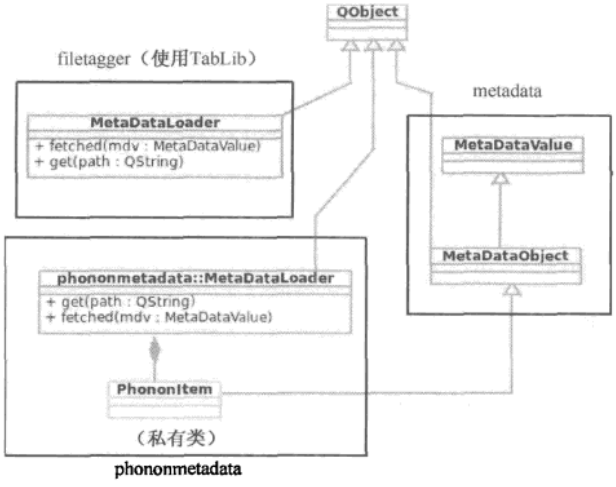

图 16.4 MetaDataLoader 和 MetaDataValue

16.15 中的 MetaDataValue 是用于操作元数据值的一个基类接口。

示例 16.15 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&);
```

程序员网址导航www.daocode.com

```
[ . . . . ]
    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;
    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。

MetaDataValue 对象可以来自于磁盘、数据库或者用户的标签。不同的库都提供了一个自定义的 MetaDataLoader 类，它在一个信号参数中包含了这种类型的一个值。

示例 16.16　src/libs/metadata/abstractmetadataloader.h

```
[ . . . . ]
namespace Abstract {
class METADATAEXPORT MetaDataLoader : public QObject {
    Q_OBJECT
public:
    explicit MetaDataLoader(QObject *parent = 0)
        : QObject(parent) {}
    static MetaDataLoader* instance();
    virtual MetaDataLoader* clone(QObject* parent=0) = 0;
    virtual ~MetaDataLoader();
    virtual const QStringList &supportedExtensions() = 0;
    virtual void get(QString path) = 0;
    virtual void get(QStringList path) = 0;
    virtual bool isRunning() const = 0;
public slots:
    virtual void cancel() = 0;

signals:
```

```
    void fetched(const MetaDataValue & mdv);
    void progressValueChanged(int);
    void progressRangeChanged(int, int);
    void finished();

};
}

#endif // AMETADATALOADER_H
```

示例 16.16 中的 AbstractMetaDataLoader 具有一个简单的、无阻断的 get()方法，它会立即返回。它释放出一个 fetched(MetaDataValue)信号，这个信号能够被连接到另一个对象的槽，比如 PlayListModel。示例 16.17 中给出了使用 TagLib 的 MetaDataLoader 具体实现。

示例 16.17 src/libs/filetagger/tmetadataloader.h

```
[ . . . . ]
class FILETAGGER_EXPORT MetaDataLoader
                          : public Abstract::MetaDataLoader {
    Q_OBJECT
public:
    typedef Abstract::MetaDataLoader SUPER;
    explicit MetaDataLoader(QObject *parent = 0);
    static MetaDataLoader* instance();
    virtual ~MetaDataLoader() {}
    const QStringList &supportedExtensions() ;
    MetaDataLoader* clone(QObject *parent) ;
    void get(QString path);
    void get(QStringList path);
    bool isRunning() const {return m_running;}
public slots:
    void cancel();
private slots:
    void checkForWork();

private:
    bool m_running;
    bool m_canceled;
    int m_processingMax;
    QStringList m_queue;
    QTimer m_timer;
};
}

[ . . . . ]
```

示例 16.18 中给出的 checkForWork()方法在一个循环中执行操作，它调用 qApp->processEvents()方法，使得在一个长时间运行的循环中 GUI 依然能够保持响应性，比如这里的情况。

程序员网址导航www.daocode.com

示例 16.18　src/libs/filetagger/tmetadataloader.cpp

```
[ . . . . ]

TagLib::MetaDataLoader::MetaDataLoader(QObject *parent) :
    SUPER(parent) {
    m_processingMax = 0;
    m_running = false;
    qDebug() << "TagLib::MetaDataLoader created.";
    connect (this, SIGNAL(finished()), this, SLOT(checkForWork()),
             Qt::QueuedConnection);
}

void TagLib::MetaDataLoader::get(QString path) {
    m_queue << path;
    m_timer.singleShot(2000, this, SLOT(checkForWork()));
}

void TagLib::MetaDataLoader::checkForWork() {
    MetaDataFunctor functor;
    if (m_queue.isEmpty() && !m_running) {
        m_processingMax = 0;
        return;
    }
    if (m_running ) return;
    m_running = true;
    m_canceled = false;
    QStringList sl = m_queue;
    m_queue = QStringList();
    m_processingMax = sl.length();
    emit progressRangeChanged(0, m_processingMax);
    for (int i=0; i<m_processingMax;++i) {
        if (m_canceled) break;
        emit fetched(functor(sl[i]));
        emit progressValueChanged(i);
        qApp->processEvents();                                    1
    }
    m_running = false;
    emit finished();
}
```

1 使得 GUI 能够处理事件(且信号也能被传送)。

示例 16.19 中能够找到实际的 TagLib 代码，这些代码被放在一个仿函数(functor)中，因为这里是第一次在 QtConcurrent 算法中使用它。经过一些测试后，我们发现仿函数并不是线程安全的，所以这里从循环中依次调用它。这里还完成了 TagLib 类型与 Qt 类型之间的全部转换。客户代码无须关心数据是如何被取出的，也不必担心所使用的字符串库，就可以继续使用来自于 libmetadata 的 MetaDataValue 接口。

示例 16.19　src/libs/filetagger/tmetadataloader.cpp

```
[ . . . . ]

MetaDataValue MetaDataFunctor::operator ()(QString path) {
```

```
    using namespace TagLib;
    MetaDataValue retval;
    FileRef f(path.toLocal8Bit().constData());
    const Tag* t = f.tag();
    Q_ASSERT( t != NULL ) ;
    retval.setFileName(path);
    retval.setTrackTitle(toQString(t->title()));
    retval.setArtist(toQString(t->artist()));
    retval.setAlbumTitle(toQString(t->album()));
[ . . . . ]

    QTime time(0,0,0,0);
    const AudioProperties* ap = f.audioProperties();
    time = time.addSecs(ap->length());
    retval.setTrackTime(time);
    return retval;
}
```

注意

在利用与 MetaDataValue 参数的排队连接发出信号之前，必须先调用 qRegisterMetaType<MetaDataValue>("MetaDataValue")。这是因为，在底层 QMetaType::construct()会动态创建另一个实例。

16.4 复习题

1. 创建模式如果能够管理对象的销毁?
2. 如何利用属性来编写更通用的写入器(Writer)?
3. 如何利用抽象工厂来编写更通用的读取器(Reader)?
4. 哪些 Qt 类是 Façade 模式的实现? 解释为什么它们是 Façade 模式或者封装器。

16.4.1 更多探讨

- 给出另一个 UserType,它会针对 InputField 的新类型而被添加到 QVariant 中。
- 关于使用元对象的 moc 对象以及 marshalling 对象的更多探讨，请参见 Qt 季刊(Qt Quarterly)[①]。

① 参见 http://doc.trolltech.com/qq/qq14-metatypes.html。

程序员网址导航www.daocode.com