第 12 章　元对象，属性和反射编程

本章将引入反射(reflection)的基本思想。所谓反射，就是指对象成员的自我检查。使用反射编程(reflective programming)，就可以编写出通用的操作，可对具有各种不同结构的类进行操作。使用通用的值存储器 QVariant，就可以按照一种统一的方式来对基本类型和其他普通类型进行操作。

12.1　QMetaObject——元对象模式

所谓元对象(meta object)，就是描述另一个对象结构的对象[①]。

元对象模式

QMetaObject 是元对象模式(MetaObject Pattern)的一个 Qt 实现，它提供了一个 QObject 对象所拥有的属性和方法的信息。元对象模式有时也称为反射模式(Reflection Pattern)。

一个拥有元对象的类就可以支持反射。这是一个许多面向对象语言都所具有的性质。虽然 C++中不存在反射，但 Qt 的元对象编译器(MetaObject compiler，moc)可以为 QObject 生成支持这种机制的代码。

像图 12.1 中的 Customer 和 Address 一样，只要满足一定的条件[②]，每个派生自 QObject 的类都会拥有一个由 moc 为其生成的 QMetaObject。QObject 拥有一个成员函数，它能够返回指向对象的 QMetaObject 的指针。这个函数的函数原型是

```
QMetaObject* QObject::metaObject() const [virtual]
```

可以使用 QMetaObject 的下面这些方法来获取一个 QObject 的信息。

- className()，它会将类的名称以 const char*格式返回。
- superClass()，如果存在基类的 QMetaObject，则返回其指针（如果不存在，则返回 0)。
- methodCount()，返回类的成员函数的个数。
- 另外还有几个非常有用的函数，将会在本章的后面进行讨论。

信号和槽机制也同样需要依赖于 QMetaObject。图 12.1 给出了各个 Qt 基类、QObject 的派生类以及由 moc 为其生成的元类(meta class)之间的继承关系。

通过使用 QMetaObject 和 QMetaProperty，就可以编写足够通用的代码来处理所有具有自我描述能力的类。

---

① meta，这个拉丁语词根的含义是“关于”，这里取其字面含义。

② 每个类都必须在头文件中定义，并且列举在工程文件的 HEADERS 中，同时此类的定义中还必须包括 Q_OBJECT 宏。

12.2　类型识别和 qobject_cast

RTTI，全称 Run Time Type Identification（运行时类型识别），如同其名称所显示的一样，是一个用来在运行时决定一个你可能仅仅拥有其基类指针的对象的实际类型的系统。

除了 C++的 RTTI 运算符 dynamic_cast 和 typeid(参见 19.10 节)之外，Qt 还提供了两种运行时的类型识别机制：

1. qobject_cast
2. QObject::inherits()

qobject_cast 是一个 ANSI 风格的类型转换运算符(参见 19.8 节)。ANSI 类型转换看起来很像模板函数：

```
DestType* qobject_cast<DestType*> ( QObject* qoptr )
```

类型转换运算符根据类型和语言的特定规则与约束将表达式从一种类型转换为另一种类型。如同其他转换运算符一样，qobject_cast 把目标类型看做模板参数，它返回指向同一个对象的 DestType 的指针。如果在运行时，实际的指针类型无法转换成 *DestType**，那么转换就会失败，此时返回值是 NULL。

如同 qobject_cast 的字面意思一样，它的参数类型受限于 ObjectType*，其中 ObjectType 类是 QObject 的派生类并且该类完全由 moc 进行处理（这需要其类定义中有一个 Q_OBJECT 宏)。

qobject_cast 实际是一个向下转换运算符，类似于 dynamic_cast。qobject_cast 允许把一个更为常规的指针和引用转换成某种特定的类型。取决于所使用的编译器，你或许可以发现，qobject_cast 的运行速度要比 dynamic_cast 快 5 到 10 倍。

拥有指向派生类的基类指针时，向下转换允许调用在基类接口中不存在的派生类方法。

找到 qobject_cast 的常见地方是在 QAbstractItemDelegate 的实际实现中，比如像示例 12.1 这样。大多数虚函数都把 QWidget*作为参数，所以可以执行类型检查来确定它究竟是何种类型的窗件。

示例 12.1　src/modelview/playlists/stardelegate.cpp

```
[ . . . . ]

void StarDelegate::
    setEditorData(QWidget* editor,
                  const QModelIndex& index) const {
    QVariant val = index.data(Qt::EditRole);
    StarEditor* starEditor = qobject_cast<StarEditor*>(editor);          1
    if (starEditor != 0) {
        StarRating sr = qVariantValue<StarRating>(val);                  2
        starEditor->setStarRating(sr);
        return;
    }
```

```
    TimeDisplay* timeDisplay = qobject_cast<TimeDisplay*>(editor);          3
    if (timeDisplay != 0) {
        QTime t = val.toTime();
        timeDisplay->setTime(t);
        return;
    }
    SUPER::setEditorData(editor, index);                                    4
    return;
}
```

1 动态类型检查。

2 从 QVariant 中抽取用户类型值。

3 动态类型检查。

4 让基类处理其他类型。

注意

qobject_cast 的实现没有使用 C++ RTTI，该运算符的代码是由元对象编译器生成的。

注意

把 qobject_cast 用于非 QObject 的基类时，需要把每个基类都放到一个形如 Q_INTERFACES(BaseClass1 BaseClass2)的代码行内，并把它放到类定义中 Q_OBJECT 宏的后面。

QObject 还提供了一个不再建议使用的、Java 风格的类型检查函数 inherits()。与 qobject_cast 不同，inherits()按树接收一个 char *类型名，而不是类型表达式。因为该运算符需要进行额外的哈希表查找操作，所以该函数比 qobject_cast 要慢一些，但是如果需要输入驱动(input-driven)型的类型检查，这个函数就非常有用了。示例 12.2 给出了一些使用 inherits()函数的客户代码。

示例 12.2 src/qtrtti/qtrtti.cpp

```
[ . . . . ]

//  QWidget* w = &s;                                                    1

    if (w->inherits("QAbstractSlider"))  cout << "Yes, it is ";
    else cout << "No, it is not";
    cout << "a QAbstractSlider" << endl;

    if (w->inherits("QListView")) cout << "Yes, it is ";
    else  cout << "No, it is not ";
    cout << "a QListView" << endl;

    return 0;
}
```

1 指向某个窗件的指针。

12.3　Q_PROPERTY 宏——描述 QObject 的属性

属性功能使得我们可以选择访问数据成员的方式：

- 直接访问，通过经典的获取函数和设置函数（getter/setter）。速度更快，更为有效。
- 间接访问，通过 QObject/QMetaObject 接口（可让代码复用性更好）。

通过省略 WRITE 函数，可以对一些属性给定只读访问。此外，也可以提供一个在属性发生更改时会发出的 NOTIFY 信号。

示例 12.3 中对 Customer 类的每个数据成员都定义了一个 Qt 属性。QVariant::Type 中列出了可用于属性的可能类型，还有一些加在 Q_DECLARE_METATYPE（参见 12.6 节）中的用户类型。这里采用了在程序开发中获得的经验，给每个属性赋予一个基于对应成员名的名称。例如，如果数据成员的名称是 m_DataItem，那么对应的属性就应当命名为 dataItem。

示例 12.3　src/properties/customer-props.h

```
[ . . . . ]
class Customer : public QObject {
    Q_OBJECT                                                          1

    /* Each property declaration has the following syntax:

    Q_PROPERTY( type name READ getFunction [WRITE setFunction]
    [RESET resetFunction] [NOTIFY notifySignal] [DESIGNABLE bool]
    [SCRIPTABLE bool] [STORED bool] )
    */

    Q_PROPERTY( QString id READ getId WRITE setId NOTIFY valueChanged);
    Q_PROPERTY( QString name READ getName WRITE setName
                NOTIFY valueChanged);
    Q_PROPERTY( QString address READ getAddress WRITE setAddress
                NOTIFY addressChanged);
    Q_PROPERTY( QString phone READ getPhone WRITE setPhone
                NOTIFY phoneChanged);
    Q_PROPERTY( QDate dateEstablished READ getDateEstablished );      2
    Q_PROPERTY( CustomerType type READ getType WRITE setType
                NOTIFY valueChanged);

  public:
    enum CustomerType
    { Corporate, Individual, Educational, Government };               3
    Q_ENUMS( CustomerType ) ;                                         4

    explicit Customer(const QString name = QString(),                5
        QObject* parent = 0);

    QString getId() const {
        return m_id;
    }
[ . . . . ]
```

```
    // Overloaded, so we can set the type two different ways:
    void setType(CustomerType newType);
    void setType(QString newType);
signals:
    void addressChanged(QString newAddress);
    void phoneChanged(QString newPhone);
    void typeChanged(CustomerType type);
    void valueChanged(QString propertyName,
        QVariant newValue, QVariant oldValue = QVariant());
private:
    QString m_id, m_name, m_address, m_phone;
    QDate m_date;
    CustomerType m_type;
};
[ . . . . ]
```

1 moc 预处理类所需要的宏。

2 只读属性。

3 枚举类型定义必须与 Q_ENUMS 宏的定义出现在同一类定义中。

4 特殊的宏可以实现生成字符串到枚举之间的转换功能；必须在同一个类中。

5 之所以声明为 explicit，是因为不希望从 QString 转换到 Customer 时出现意外。

值得注意的是，在 class Customer 的 public 部分定义的 enum CustomerType-defined，其中的 Q_ENUMS 宏告诉 moc 在 QMetaProperty 中为该属性生成一些函数来辅助字符串到枚举值的转换。

示例 12.4 中给出了设置函数和获取函数的定义，它们的实现都采用了常规的方式。

示例 12.4 src/properties/customer-props.cpp

```
[ . . . . ]
Customer::Customer(const QString name, QObject* parent)
:QObject(parent) {
    setObjectName(name);
}

void Customer::setId(const QString &newId) {
    if (newId != m_id) {
        QString oldId = m_id;
        m_id = newId;
        emit valueChanged("id", newId, oldId);
    }
}
[ . . . . ]
void Customer::setType(CustomerType theType) {
    if (m_type != theType) {
        CustomerType oldType = m_type;
        m_type = theType;
        emit valueChanged("type", theType, oldType);
    }
}

/* Method for setting enum values from Strings. */
```

```
void Customer::setType(QString newType) {                              1

    static const QMetaObject* meta = metaObject();                     2
    static int propindex = meta->indexOfProperty("type");
    static const QMetaProperty mp = meta->property(propindex);

    QMetaEnum menum = mp.enumerator();                                 3
    const char* ntyp = newType.toAscii().data();
    CustomerType theType =
                  static_cast<CustomerType>(menum.keyToValue(ntyp));

    if (theType != m_type) {                                           4
        CustomerType oldType = m_type;
        m_type = theType;
        emit valueChanged("type", theType, oldType);
    }
}
QString Customer::getTypeString() const {
    return property("type").toString();
}
[ . . . . ]
```

1 重载的版本，能够接收一个字符串作为参数。如果不清楚该如何设置，则可以将其设置为-1。

2 因为它们是静态局部变量，所以初始化操作仅仅执行了一次。

3 这些代码每次都会执行。

4 始终检测是否需要 valueChanged 信号。

重载函数 setType(QString)的实现充分利用了 QMetaProperty 的 Q_ENUMS 宏，它把 QString 转换成适当的枚举值。为了获得与一个枚举(enum)对应的正确的 QMetaProperty 对象，首先获得了 QMetaObject 对象，然后调用 indexOfProperty()函数和 property()函数进行查找。QMetaProperty 有一个称为 enumerator()的函数，可以用其将字符串转换成枚举值。如果给定的 QString 参数与任何一个枚举值都不匹配，那么 keyToValue()函数将返回-1。

静态局部变量

仔细观察下，我们在代码中是将三个局部(块作用域)变量——meta，propindex 和 mp——定义成静态变量的。静态局部变量仅初始化一次，这也是我们的目标——对此函数的重复调用将会使用同一个的 QMetaProperty 对象来进行转换工作。在一个函数中，这样使用静态局部变量可以极大地提高此函数的运行时性能[①]。

12.4 QVariant 类：属性访问

可以通过下面的函数来获得任意属性的值。

① 当然，这需要取决于创建对象的昂贵程度以及函数调用的频繁程度。

```
QVariant QObject::property(QString propertyName);
```

QVariant 是一个联合体[①]的封装，其中包含了所有基本类型和所允许的全部 Q_PROPERTY 类型。可以把 QVariant 创建为另外一个有类型值的封装。QVariant 会记住自己的类型，并且拥有获取和设置其值的成员函数。

QVariant 中包含丰富的函数来进行数据转换和合法性检查，尤其是有一个 toString() 函数能够为它支持的许多类型返回其 QString 表示[②]。这个类大大简化了属性接口。

示例 12.5 中给出了如何借助直接的获取和设置函数，或者借助间接的 property()和 setProperty()函数获得和设置同一个属性值的做法。

示例 12.5 src/properties/testcustomerprops.cpp

```
[ . . . . ]

#include "customer-props.h"
void TestCustomerProps::test() {
    Customer cust;
    cust.setObjectName("Customer");                                    1
    cust.setName("Falafal Pita");                                      2
    cust.setAddress("41 Temple Street; Boston, MA; 02114");
    cust.setPhone("617-555-1212");
    cust.setType("Government");                                        3
    QCOMPARE(cust.getType(), Customer::Government);                    4
    QString originalid = "834";                                        5
    cust.setId(originalid);
    QVariant v = cust.property("id");                                  6
    QString str = v.toString();
    QCOMPARE(originalid, str);
    QDate date(2003, 7, 15);
    cust.setProperty("dateEstablished", QVariant(date));               7
    QDate anotherDate = cust.getDateEstablished();                     8
    QEXPECT_FAIL("", "These are different dates", Continue);
    QCOMPARE(date, anotherDate);
    cust.setId(QString("anotherId"));
    qDebug() << objToString(&cust);
    cust.setType(Customer::Educational);
    qDebug() << " Educational=" << cust.getType();
    cust.setType("BogusType");
    qDebug() << " Bogus= " << cust.getType();
    return;
}

QTEST_MAIN(TestCustomerProps)
```

1 QObject 函数调用。

2 设置一些简单的属性。

3 用字符串来设置枚举属性。

① 在 C 和 C++中，联合体是一个数据结构，它声明了两种或者两种以上的数据成员，但将其分配在同一个地址。也就是说，联合体将会占据足够容纳最大已声明数据成员的内存。在初始化时，联合体仅仅可以为其中一个已声明成员存储值。

② 更多详情，可以参阅 QVariant::canConvert()。

4 与枚举值比较。
5 设置一个字符串属性。
6 通过 QObject 基类方法把值作为一个 QVariant 获取回来。
7 设置日期属性，并封装到 QVariant 中。
8 通过类型相关的获取函数取回日期。

示例 12.6 中给出了一个反射的方法 objToString()，它能够对任何定义了 Qt 属性的类进行操作。该函数的工作原理是通过对所有的 property()索引值进行迭代，这种方法与 java.lang.reflect 接口相对应。只有 canConvert(QVariant ::String)变量类型可被打印。

示例 12.6　src/properties/testcustomerprops.cpp

```
[ . . . . ]

QString objToString(const QObject* obj) {
    QStringList result;
    const QMetaObject* meta = obj->metaObject();                    1
    result += QString("class %1 : public %2 {")
        .arg(meta->className())
        .arg(meta->superClass()->className());
    for (int i=0; i < meta->propertyCount(); ++i) {
        const QMetaProperty qmp = meta->property(i);                2
        QVariant value = obj->property(qmp.name());
        if (value.canConvert(QVariant::String))
            result += QString("  %1 %2 = %3;")
            .arg(qmp.typeName())
            .arg(qmp.name())
            .arg(value.toString());
    }
    result += "};";
    return result.join("\n");
}
```

1 通过 QMetaObject 来深入地分析对象。
2 每个属性都有一个 QMetaProperty。

要构建这个程序，需要在工程文件中包含代码

```
CONFIG += qtestlib
```

该程序会以 C++的风格输出一个对象的状态，见示例 12.7。

示例 12.7　src/properties/output.txt

```
********* Start testing of TestCustomerProps *********
Config: Using QTest library 4.6.2, Qt 4.6.2
PASS   : TestCustomerProps::initTestCase()
QDEBUG : TestCustomerProps::test() "class CustProps : public QObject {
  QString objectName = Customer;
  QString id = anotherId;
  QString name = Falafal Pita;
  QString address = 41 Temple Street; Boston, MA; 02114;
  QString phone = 617-555-1212;
```

```
  QString phone = 617-555-1212;
  QDate dateEstablished = 2003-07-15;
  CustomerType type = 3;
};"
QDEBUG : TestCustomerProps::test()  Educational= 2
QDEBUG : TestCustomerProps::test()  Bogus=  -1
PASS   : TestCustomerProps::test()
PASS   : TestCustomerProps::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped
********* Finished testing of TestCustomerProps *********
```

12.5 动态属性

即使未用 Q_PROPERTY 在类中定义，在 QObject 中加载和存储一些属性也是可能的。

迄今为止，我们已经对用 Q_PROPERTY 宏定义的那些属性进行了专门处理。把这些属性对该类的 QMetaObject 是可知的，且有 QMetaProperty 与之对应。同一类的所有对象会共享同一个 metaObject，因而会有相同元属性组。

另一方面，在运行时会获得动态属性，这些属性对于获得它们的对象而言也是很特别的。换句话说，同一个类的两个对象会具有相同的元属性列表，但对于动态属性可以有不同的动态属性列表。示例 12.8 中定义了一个包含单一 Q_PROPERTY 的类，并给出了一个属性为 someString 的类。

示例 12.8 src/properties/dynamic/dynoprops.h

```
[ . . . . ]
class DynoProps : public QObject {
    Q_OBJECT
    Q_PROPERTY(QString someString READ someString WRITE setSomeString);
 public:
    friend QDataStream& operator<<(QDataStream& os, const DynoProps& dp);
    friend QDataStream& operator>>(QDataStream& is, DynoProps& dp);
    QString someString() { return m_someString; }
    void setSomeString(QString ss) { m_someString = ss; }
    QString propsInventory();
 private:
    QString m_someString;
};
[ . . . . ]
```

示例 12.9 中，propsInventory() 的实现给出了一种显示固定属性和动态属性的方法。固定属性的清单来自 QMetaObject。使用 QMetaProperty::read() 或者 QObject::property() 可以查询属性的值。propertyCount() 函数会设置对 QMetaProperty 清单进行遍历的条件。

QMetaObject 无法知道动态属性。相反，要知道动态属性必须得使用 QObject 的一些方法。可以对利用 QObject::dynamicPropertyNames() 返回的 QList 清单中的名称进行遍历，并可用 QObject::property() 获得其值。

示例 12.9 src/properties/dynamic/dynoprops.cpp

```
[ . . . . ]

QString DynoProps::propsInventory() {
```

```
    static const QMetaObject* meta = metaObject();
    QStringList res;
    res << "Fixed Properties:";
    QString propData;
    for(int i = 0; i < meta->propertyCount(); ++i) {
       res << QString("%1\t%2").arg(QString(meta->property(i).name()))
              .arg(meta->property(i).read(this).toString());        1
    }
    res << "Dynamic Properties:";
    foreach(QByteArray dpname, dynamicPropertyNames()) {
       res << QString("%1\t%2").arg(QString(dpname))
              .arg(property(dpname).toString());
    }
    return res.join("\n");
}
```

1 这里本来也是可以使用 `property(propName)` 的。

除了访问它们时稍有不便之外，动态属性的使用在很大程度上还是与固定属性相同的，例如，可以对它们进行序列化。示例 12.10 中给出了两个序列化运算符的实现方法。在序列化中使用了一种与用于 `propsInventory()` 函数中相类似的技术。

示例 12.10　src/properties/dynamic/dynoprops.cpp

```
[ . . . . ]
QDataStream& operator<< (QDataStream& os, const DynoProps& dp) {
  static const QMetaObject* meta = dp.metaObject();
  for(int i = 0; i < meta->propertyCount(); ++i) {
       const char* name = meta->property(i).name();
       os << QString::fromLocal8Bit(name)                          1
          << dp.property(name);
  }
  qint32 N(dp.dynamicPropertyNames().count());                     2
  os << N;
  foreach(QByteArray propname, dp.dynamicPropertyNames()) {
     os << QString::fromLocal8Bit(propname) << dp.property(propname);
  }
  return os;
}

QDataStream& operator>> (QDataStream& is, DynoProps& dp) {
  static const QMetaObject* meta = dp.metaObject();
  QString propname;
  QVariant propqv;
  int propcount(meta->propertyCount());
  for(int i = 0; i < propcount; ++i) {
     is >> propname;
     is >> propqv;
     dp.setProperty(propname.toLocal8Bit(), propqv);              3
  }
  qint32 dpcount;
  is >> dpcount;
  for(int i = 0; i < dpcount; ++i) {
```

```
        is >> propname;
        is >> propqv;
        dp.setProperty(propname.toLocal8Bit(), propqv);
    }
    return is;
}
```

1 把 char *序列化成 QString。

2 序列化 int。

3 用 QString 逆向转换来反序列化 char*。

示例 12.11 中给出了说明动态属性用法的客户代码。

示例 12.11 src/properties/dynamic/dynoprops-client.cpp

```
#include <QtCore>
#include "dynoprops.h"

int main() {
    QTextStream cout(stdout);
    DynoProps d1, d2;
    d1.setObjectName("d1");
    d2.setObjectName("d2");
    d1.setSomeString("Washington");
    d1.setProperty("AcquiredProp", "StringValue");
    d2.setProperty("intProp", 42);
    d2.setProperty("realProp", 3.14159);
    d2.setProperty("dateProp", QDate(2012, 01, 04));
    cout << d1.propsInventory() << endl;
    cout << d2.propsInventory() << endl;
    cout << "\nNow we save both objects to a file, close the file,\n"
            "reopen the file, read the data from the file, and use it\n"
            "to create new DynoProps objects.\n" << endl;
    QFile file("file.dat");
    file.open(QIODevice::WriteOnly);
    QDataStream out(&file);
    out << d1 << d2;
    file.close();
    DynoProps nd1, nd2;
    file.open(QIODevice::ReadOnly);
    QDataStream in(&file);
    in >> nd1 >> nd2;
    file.close();
    cout << "Here are the property inventories for the new objects.\n";
    cout << nd1.propsInventory() << endl;
    cout << nd2.propsInventory() << endl;
}
```

示例 12.12 中给出了这个程序的输出结果。

示例 12.12 src/properties/dynamic/output.txt

```
Fixed Properties:
objectName      d1
someString      Washington
```

```
Dynamic Properties:
AcquiredProp    StringValue
Fixed Properties:
objectName      d2
someString
Dynamic Properties:
intProp 42
realProp        3.14159
dateProp        2012-01-04

Now we save both objects to a file, close the file,
reopen the file, read the data from the file, and use it
to create new DynoProps objects.

Here are the property inventories for the new objects.
Fixed Properties:
objectName      d1
someString      Washington
Dynamic Properties:
AcquiredProp    StringValue
Fixed Properties:
objectName      d2
someString
Dynamic Properties:
intProp 42
realProp        3.14159
dateProp        2012-01-04
```

12.6　元类型，声明和注册

QMetaType 是一个用于值类型的辅助类(helper class)。对于 60 多种内置类型，QMetaType 为每个类型 ID 关联了一个类型名，从而使构造和析构可以在运行时动态发生。有一个名称为 QMetaType::Type 的公共枚举，它有所有 QVariant 兼容类型的值。在 QMetaType:: Type 中的枚举值与 QVariant:: Type 中的枚举值一样。

通过使用 Q_ENUMS 宏，我们已在 QVariant 系统中加入了一些自定义的枚举类型。使用 Q_DECLARE_METATYPE(*MyType*)宏也有可能把自己的值类型加到 QMetaType 列表中。如果 *MyType* 有公共的默认复制构造函数和公共的复制构造函数以及一个公共的析构函数，Q_DECLARE_METATYPE 宏使得它可用作 QVariant 中的自定义类型。

示例 12.13 中，我们将一个新的值类型 Fraction，引入到包含它定义的程序的已知元类型库。没有必要明确定义默认构造函数和复制构造函数，也没有必要明确定义析构函数，因为编译器会生成这些函数，它们会进行逐一复制或者逐一赋值，这正是我们所需要的。把这个宏放到头文件类定义的下面是一种标准的做法。

示例 12.13　src/metatype/fraction.h

```
[ . . . . ]
class Fraction : public QPair<qint32, qint32> {
public:
    Fraction(qint32 n = 0, qint32 d = 1) : QPair<qint32,qint32>(n,d)
```

```
    { }
};

Q_DECLARE_METATYPE(Fraction);
[ . . . . ]
```

12.6.1 qRegisterMetaType()

要注册的元类型必须已经用 Q_DECLARE_METATYPE 声明过。模板函数 qRegisterMetaType<T>()会注册类型 T 并返回由 QMetaType 使用的内部 ID。这个函数有一个重载版本，qRegisterMetaType<T>(const char* *name*)，它可以让你注册一个名称作为类型 T 的名称。对于这个函数的调用必须早早地出现在主程序中，一定要在任何试图以一种动态方式尝试使用该注册的类型之前。

声明了元类型之后，存储一个值到 QVariant 中是可能的。示例 12.14 展示了如何存储以及从 QVariant 中取回已声明元类型的值。

示例 12.14 src/metatype/metatype.cpp

```
[ . . . . ]

int main (int argc, char* argv[]) {
    QApplication app(argc, argv);
    qRegisterMetaType<Fraction>("Fraction");
    Fraction twoThirds (2,3);
    QVariant var;
    var.setValue(twoThirds);
    Q_ASSERT (var.value<Fraction>() == twoThirds);

    Fraction oneHalf (1,2);
    Fraction threeQuarters (3,4);

    qDebug() << "QList<Fraction> to QVariant and back."

    QList<Fraction> fractions;
    fractions << oneHalf << twoThirds << threeQuarters;
    QFile binaryTestFile("testMetaType.bin");
    binaryTestFile.open(QIODevice::WriteOnly);
    QDataStream dout(&binaryTestFile);
    dout << fractions;
    binaryTestFile.close();
    binaryTestFile.open(QIODevice::ReadOnly);
    QDataStream din(&binaryTestFile);
    QList<Fraction> frac2;
    din >> frac2;
    binaryTestFile.close();
    Q_ASSERT(fractions == frac2);
    createTest();
    qDebug() << "If this output appears, all tests passed.";
}
```

已注册类型的值可以借助 QMetaType:: construct()动态构造，如示例 12.15 所示。

示例 12.15　src/metatype/metatype.cpp

```
[ . . . . ]

void createTest() {
    static int fracType = QMetaType::type("Fraction");
    void* vp = QMetaType::construct(fracType);
    Fraction* fp = reinterpret_cast<Fraction*>(vp);                    1
    fp->first = 1;
    fp->second = 2;
    Q_ASSERT(*fp == Fraction(1,2));
}
```

1 注意：这是第一次在本书中使用 reinterpret_cast!

QMetaType::construct()会返回一个 void 指针，所以应使用 reinterpret_cast 将它转换成一个 Fraction 指针。

12.7　invokeMethod()

Qt 把信号连接到槽需要一种机制：通过名称以类型安全的方式来间接调用这些槽。当调用槽时，实际是由 invokeMethod()完成的。示例 12.16 显示了它是如何接收一个作为函数名称的字符串的。除了槽，标记为 Q_INVOKABLE 的常规成员也可以用这种方法来间接调用。

示例 12.16　src/refl ection/invokemethod/autosaver.cpp

```
void AutoSaver::saveIfNecessary() {
    if (!QMetaObject::invokeMethod(parent(), "save")) {
        qWarning() << "AutoSaver: error invoking save() on parent";
    }
}
```

与 QObject::connect()类似，invokeMethod()接受一个可选参数 Qt::ConnectionType，该参数可让你来决定是要用同步调用还是要用异步调用。默认情况下为 Qt::AutoConnection，表示在发射者和接收者处于同一个线程中时会同步执行一个槽。

要通过 invokeMethod()向函数传递类型参数，可以用示例 12.17 中的 Q_ARG 宏创建一些值，这样会返回一个 QGenericArgument，它封装了单个参数的类型和值信息。

示例 12.17　src/refl ection/invokemethod/arguments.cpp

```
QByteArray buffer= ... ;
const bool b = QMetaObject::invokeMethod(m_thread, "calculateSpectrum",
                    Qt::AutoConnection,
                    Q_ARG(QByteArray, buffer),
                    Q_ARG(int, format.frequency()),
                    Q_ARG(int, bytesPerSample));
```

12.8　练习：反射

1. 编写一个可创建下列各个类实例的程序——QSpinBox, QProcess, QTimer——并向用户显示一个属性列表（以及一些值，如果它们可以转换成 QString），外加一个所有函数名称的清单。

2. 重写之前的某个 GUI 应用程序，从 `main()` 中使用 `invokeMethod()` 而不是通过直接调用来 `show()` 初始化窗件。

12.9 复习题

1. 从 `QMetaObject` 中可以获得何种类型的信息?
2. 用来实现数据反射的 Qt 类有哪些?
3. 每个 `QObject` 派生类的 `QMetaObject` 代码是如何生成的?
4. 什么是向下转换? 在何种情况下你愿意使用向下转换?
5. 什么是 RTTI? Qt 是如何提供 RTTI 的?
6. 讨论 `dynamic_cast` 和 `qobject_cast` 运算符的区别。
7. 什么是 `QVariant`? 应该如何使用它?
8. 使用 `property()` 和 `setProperty()`，比直接使用获得函数和设置函数有何好处?
9. `property()` 函数会返回什么? 如何才能获得实际的存储值?
10. 请解释向 `QObject` 添加新的可在运行时获得的属性是如何可能的。
11. 说明动态属性是如何实现序列化的。