第8章 QObject，QApplication，信号和槽

QObject 是 Qt 库中许多重要的类的基类，如 QEvent，QApplication，QLayout 和 QWidget。我们会把任何 QObject 派生类的对象都看作是一个 QObject 对象。一个 QObject 可以有一个父对象和一些子对象，这是组合模式(Composite pattern)的另一种实现方式。它可以使用信号和槽，即观察者模式(Observer pattern)的一种实现，与其他 QObject 通信。QObject 使基于事件的编程成为可能，其中用到了 QApplication 和 Qt 的事件循环。

下面先简单看一下 QObject 的定义。

```
class QObject {
  public:
    explicit QObject(QObject* parent=0);
    QObject * parent () const;
    QString objectName() const;
    void setParent ( QObject * parent );
    const ObjectList & children () const;
    // ... more ...
};
```

QObject 没有公有的复制构造函数或复制赋值运算符。向下到 QObject 类定义的结尾处有一个宏 Q_DISABLE_COPY(QObject)，它显式地确保任何 QObject 都不能被复制。QObject 也不是设计用于复制的。通常来说，QObject 会用来代表具有唯一身份的对象，也就是说，它们对应于现实世界中具有同样某种永久身份的东西。这种不带复制构造函数策略的一个直接后果就是永远无法通过值传递方式向函数传递 QObject。尽管把一个 QObject 对象的数据成员复制给另外一个 QObject 对象是可能的，但这两个 QObject 对象仍被认为是不同的。

显式构造函数

QObject(及其派生类)的单参数构造函数应当予以显式(explicit)声明，以避免意外的隐式转换的发生[①]。QObject 不被用来持有一个临时值，而且也不应该有从一个指针或单值生成另一个对象的可能。

每个 QObject 都可以有(至多)一个父 QObject，且可以拥有任意数量的子 QObject。换句话说，每个子对象的类型必须是 QObject 或者必须派生自 QObject。每个 QObject 都将指向各个子对象的指针存放在一个 QObjectList 中[②]。这一列表自身是通过一种松散的风格创建的，以尽量降低那些不带子孙的对象的开销。因为每个子对象都是一个 QObject，也可以有任意数量的子对象，所以很容易明白为什么不允许在 QObject 对象之间进行复制。

① 在 2.12 节曾第一次讨论到关键字 explicit。

② QObjectList 是 QList<QObject *>的别名，只是通过 typedef 进行了重定义而已。

子对象的概念有助于我们理解身份的概念和 QObject 之间不允许复制的策略。如果把一个一个的人看成是 QObject，则每个 QObject 的身份都有所不同这一点就很容易理解了。同样，每个 QObject 都可以有子对象的概念也非常清楚了。而对于每个 QObject 都至多有一个父对象的规则也就可以看成是一种简化类的实现的方式。最后，也就可以看出 QObject 不允许复制策略的必要性了。即使有可能去“克隆”一个人(也就是，将一个 QObject 对象的数据成员复制到另一个 QObject 对象中)，也无法去处理这个人所拥有的子孙问题，因此可以得到一个非常清楚的答案：克隆对象是一个拥有不同身份、完全分离的、不同的对象。

每个 QObject 父对象都会管理自己的子对象。这就意味着，在调用 QObject 的析构函数时会自动销毁该对象的子对象。

子对象列表会在各个 QObject 对象之间建立一种双向的关联：

- 每个父对象都知道它的子对象的地址。
- 每个子对象都知道其父对象的地址。

给某个对象设置父对象，将会隐含地把此对象添加到父对象的子对象列表之中，例如：

```
objA->setParent(objB);
```

会把 objA 的指针添加到 objB 的子列表中。如果随后再运行语句

```
objA->setParent(objC);
```

那么 objA 的指针就会从 objB 的子对象列表中移除，然后添加到 objC 的子对象列表中。这一行为称作重父化(reparenting)。

父对象与基类的比较

不应将父对象和基类混为一谈。父—子关系是为了描述对象运行时的约束和管理关系。基类派生关系是编译时各个类进行判定的一种静态关系。

当然，父对象也可能是其某些子对象的基类的实例。但这两种关系仍是不同的，务必不要混淆，尤其是考虑到许多类会直接或间接地派生自 QObject 时更是如此。图 8.1 应该可以澄清这一思想。

正如之前看到的那样，图形用户界面(GUI)中的所有窗件都派生自 QWidget，而 QWidget 则从 QObject 派生而来。像对待 QObject 一样，我们会把任意的 QWidget 派生类的对象都看作是一个 QWidget 对象(或者在某些时候，仅仅是看作一个窗件)。在 GUI 中，父—子关系通常是可见的：子窗件会显示在父窗件中。图 8.1 中，对话框窗件有数个子对象，包括：一个标签窗件，一个行编辑窗件以及两个按钮窗件。它还有一个标题栏窗件，既可以是该对话框的父对象也可以是其兄弟对象。标题栏窗件通常是由窗口管理器提供且含有数个子窗件，包括几个可以让用户能够最小化、最大化或者关闭该对话框的按钮窗件。

从图 8.1 中也可以看出对子对象管理需求的必要性。关闭对话框时(例如，通过单击右上角处的 X 按钮)，希望整个对话框窗口可以从屏幕上消失掉(也就是说，要销毁掉)。并不希望将对话框窗口打散开(例如，剩下按钮或标签窗件)而仍旧停留在屏幕上，也并不希望把这项清理工作的重担压在编程人员的身上。这就是为什么要在调用 QObject 的析构函数时要由它负责销毁自己所有的子对象的原因。这是一个自然递归的过程。当销毁 QObject 时，它首先会调用自己每个子对象的析构函数，每个子对象必须再调用自己每个子对象的析构函数，等等。

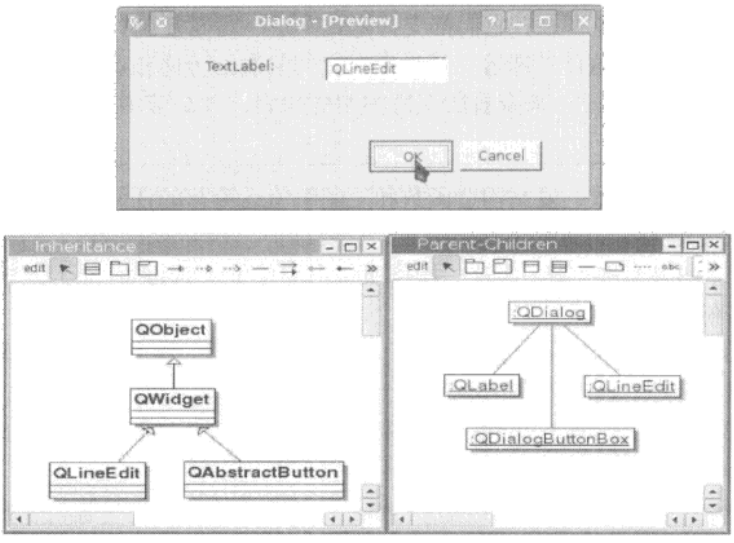

图 8.1　父-子对象和基于派生的类

现在可以来考虑一下，假设要复制 QObject，将会引起一些什么样的问题。例如，副本对象会和原对象拥有同一个父亲吗？副本对象应该(在某种意义上)拥有原对象的子对象吗？子对象列表的浅复制(shallow copy)将无法工作，因为如果那样可行的话，则每个子对象都将会拥有两个父对象。在那种情况下，如果需要销毁副本对象(例如，如果该副本是一个函数调用中的值参数)，那么每一个子对象也需要销毁，这样即使采用资源共享的方法也会遇到一些严重的问题。而在子对象列表非常庞大且指向的对象很大的情况下，子对象列表的深复制(deep copy)操作会付出高昂的代价。因为每个子对象同样也可以拥有任意数量的子对象，这种本就问题多多的方法自然会遇到更为严重的问题。

通常而言，没有父对象的 QObject 应当在程序栈区(stack)中进行定义，而那些有父对象的 QObject 则应当在堆区(heap)动态创建出来。这样可有助于确保发生正确的析构操作：位于栈区的对象是没有父对象的，在它超出作用域时就会被销毁掉(例如，当它在函数中定义而函数结束返回时，或者在控制流离开它所定义的地方时)。然后，在被析构之前，该(栈)对象会销毁它(堆区)的子对象。

8.1　值和对象

C++类型可以分成两类：值(value)类型和对象(object)类型。

值类型的实例通常相对简单，占用相邻的内存空间，而且可以进行快速复制或者比较。值类型的例子有：*Anything**，int，char，QString，QDate 和 QVariant。

QVariant

QVariant 是一种特殊的联合体类型，可保存所有可复制的内置类型和编程人员定义的类型。有关 QVariant 以及 QVariant 已支持的诸如 QList, QImage, QString, QMap,

QHash 等各类型的一件趣事是：它们都支持隐式共享、写时复制(copy on write)和引用计数(reference counting)。这就意味着，通过值对它们复制、传递和返回的代价要相对低些。可参阅 QMetaType::Type 来查看所支持类型的列表，11.5 节中也讲解了实现隐式共享的更多细节。

带有公有默认构造函数、复制构造函数和复制赋值运算符的任何类都是值类型的。

相反，对象类型的实例通常要复杂得多，它要维护一些类型的身份。对象类型通常很少进行复制(克隆)。如果允许克隆，为这一操作所付出的代价也会很大，而且生成的新对象(图)与原来对象的身份也不会相同。

QObject 的设计人员毫不犹豫地采用了“无复制”策略，该方法就是将赋值运算符与复制构造函数设置为 private 类型。这种设计方法有效地阻止了编译器给 QObject 派生类生成赋值运算符和复制构造函数。采用这一机制后，如果试图通过值传递的方式向函数传递或者从函数返回 QObject 派生类对象，都会导致编译时错误。

提示

从来没有令人信服的要在堆区创建 QList，QString，QHash，QImage 或者其他与 QVariant 相关类型的理由，也不要那样做。还是让 Qt 为你完成引用计数和内存管理吧。

8.2 组合模式：父对象和子对象

根据[Gamma 95]，组合模式(composite pattern)的意图是通过将部分—整体的层次结构表示成树状结构，以便于使用较为简单(组件)的部分来创建出复杂(复合)的对象。组合模式的使用场景是，客户(client)无须区分简单的组成部分与由简单部分所组成(例如，包含)的较为复杂的部分。

图 8.2 描述的是组合模式。在这个示意图中，有两个不同的类可以方便地描述上述两种角色：

- 复合对象是包含可以包含子对象的类。
- 组件对象是可以拥有一个父对象的类。

许多 Qt 类都用到了组合模式：QObject，QWidget，QTreeWidgetItem，QDomNode，QHelpContentItem 和 QResource。在任何基于树的结构体中都可以找到组合模式。

图 8.3 中，可以看出 QObject 既是复合对象也是组件对象。可以把 QObject 对象之间的整体—部分关系表达成父-子关系。在这样的一棵树中，最高层的(即最为“复合”的)QObject 对象(即树的根)将会有若干个子对象而没有父对象。最简单的 QObject(即这棵树的叶子节点)都有一个父对象而无子对象。客户代码可以递归地处理此树中的每一个节点。

下面给出一个可能会用到组合模式的例子。萨福克大学由 Gleason Archer 在 1906 年创建，当时他决定开始向一些想要成为律师的零售商讲授法律的基本原则。最开始时，他只有一位秘书，后来又雇佣了一些讲师。这个学校的组织图非常简单：一间办公室、几个任务分工不同的雇员。随着企业不断发展壮大，组织图开始变得越来越复杂，比如逐渐开始增加了新的办公室和公寓。而在 100 多年后的今天，当时的法律学校得到了发展，已拥有一个艺术与科学学院、一个商学院、一个艺术与设计学院、海外学校以及许多专用的办公室，因而使组织

图变得异常复杂起来，同时也注定以后还会越来越复杂。图 8.4 所示为目前萨福克大学的一个简化过的、简明的组织图。

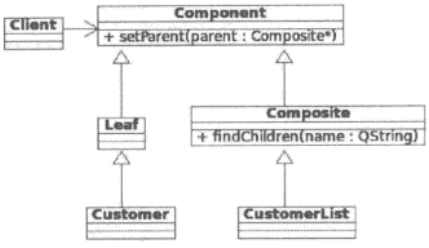

图 8.2　组件和复合

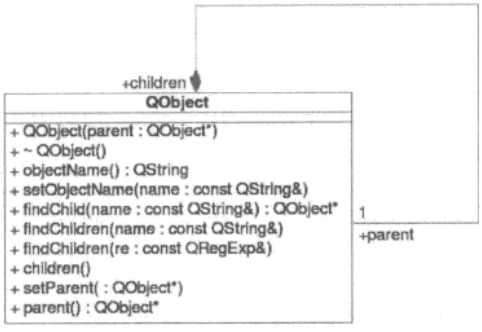

图 8.3　QObject：复合和组件

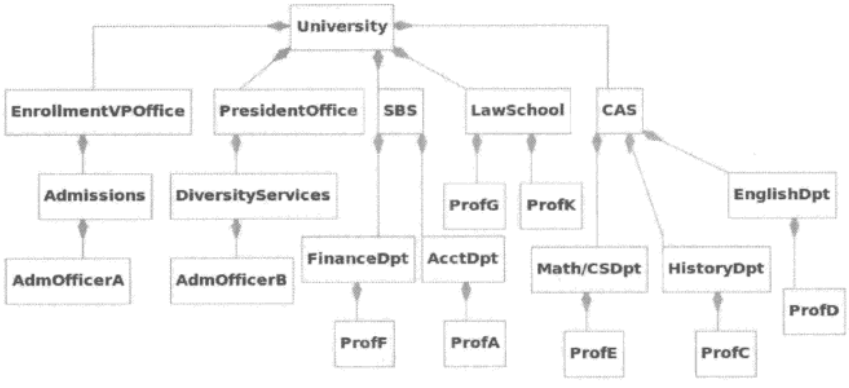

图 8.4　萨福克大学的组织图

这幅图中的每个矩形框都是一个组件。它可能是一个复合组件并拥有子组件，而反过来，这些子组件又可能是复合组件或者是单独的一个组件。例如，PresidentOffice 有单独的雇员(例如，校长及其助手)和子办公室(例如，DiversityServices)。这棵树的叶子是整个组织的个人雇员。

可以使用组合模式来对这所大学进行建模。树中的每一个节点都可以使用下面的类的一个 OrgUnit 型对象来进行表示。

```
class OrgUnit : public QObject {
  public:
    QString getName();
    double getSalary();
  private:
    QString m_Name;
    double m_Salary;
};
```

通过 QObject 的公有接口，可以构建一个类似于树的表达方式来描述这所大学的组织结构，然后编码实例化一个 OrgUnit 并调用 setParent()将其添加到合适的子对象列表中。

对于树中的每一个 OrgUnit 指针 ouptr，可以按照下面的规则来初始化其 m_Salary 数据成员。

- 如果 ouptr 指向一个单一雇员，那么使用此雇员的实际工资。
- 否则，将其初始化为 0。

可以像下面这样来实现 getSalary() 方法。

```
double OrgUnit::getSalary() {
  QList<OrgUnit*> childlst = findChildren<OrgUnit*>();
  double salaryTotal(m_Salary);
  if(!childlst.isEmpty())
    foreach(OrgUnit* ouptr, childlst)
      salaryTotal += ouptr->getSalary();
  return salaryTotal;
}
```

可以从任意特定的节点调用 getSalary() 方法，返回的结果是以此节点为根的子树所代表的大学中相应部门的工资总和。例如，如果 ouptr 指向 University，那么 ouptr->getSalary() 会返回整个大学的总工资；如果 ouptr 指向 EnglishDpt，那么 ouptr->getSalary() 会返回英语系的总工资；如果 ouptr 指向 ProfE，那么 ouptr->getSalary() 仅返回 ProfE 的个人工资。

8.2.1 查找子对象

每个 QObject 都可以有不限数量的 QObject 子对象。这些子对象的地址会存放在一个特殊的 QObject 指针容器内[①]。QObject 有一个成员函数，可以返回一个指向主对象中全部子对象的指针列表。这个函数的函数原型是

```
const QObjectList& QObject::children() const
```

子对象在该列表中的出现次序是它们在添加到该列表中时的(最初)次序。有一些特定的运行时操作可以改变该次序[②]。

QObject 还提供两个名称为 findChildren() 的重载(递归)函数。每个都会返回一个满足特定条件的子对象列表。该函数被重载之后的一种函数原型具有形式

```
QList<T> parentObj.findChildren<T> ( const QString& name = QString() ) const
```

该函数返回一个类型为 T 的子对象列表，其对象名与 *name* 相等。如果 *name* 是空字符串，那么将会起到一个类过滤器的作用，返回结果是一个 QList，其中包含指向全部子对象的指针，而这些子对象都可以通过类型转换变成类型 T 的对象[③]。

示例 8.1 说明如何来调用这个函数。必须在函数名称之后提供一个模板参数。

示例 8.1 src/findchildren/findchildren.cpp

```
[ . . . . ]
/* Filter on Customer* */
    QList<Customer*> custlist = parent.findChildren<Customer*>();
    foreach (const Customer* current, custlist) {
        qDebug() << current->toString();
    }
[ . . . . ]
```

① 回忆一下：从 QObject 派生的任何对象都称为 QObject。也就是说，QObject 指针也就可以保存一个派生对象的地址。
② 例如，提升(或者降低)QWidget 子对象，会将该窗口部件置于所有部分重叠同级窗口部件的前面(或者后面)。
③ 19.7 节将讨论类型变换和强制类型转换。

这是一个简单的示例，因此为了让它更真实，假定父对象是 SalesManager，其子列表中或许就会包含指向 Supplier、SupportStaff 和 SalesPerson 对象的指针，相应地，这些对象拥有的子列表或许会包含指向 Customer、TravelAgent 及其他相关对象的指针。

8.2.2　QObject 的子对象管理

示例 8.2 给出的是一个 QObject 派生类[①]。

示例 8.2　src/qobject/person.h

```
[ . . . . ]

class Person : public QObject {
 public:
    explicit Person(QString name, QObject* parent = 0);
    virtual ~Person();
};
[ . . . . ]
```

该类的全部实现如示例 8.3 所示。值得注意的是，~Person() 的确没有做任何显式的对象删除。它只是显示了即将要销毁的对象的名称(出于教学目的)。

示例 8.3　src/qobject/person.cpp

```
#include "person.h"
#include <QTextStream>

static QTextStream cout(stdout);

Person::Person(QString name, QObject* parent)
         : QObject(parent) {
    setObjectName(name);
    cout << QString("Constructing Person: %1").arg(name)
         << endl;
}

Person::~Person() {
    cout << QString("Destroying Person: %1").arg(objectName())
         << endl;
}
```

示例 8.4 中显示的是 growBunch()，它创建了一些对象，将它们添加到其他对象中，然后退出。而当 growBunch() 返回时，会销毁它的所有本地对象。

示例 8.4　src/qobject/bunch.cpp

```
[ . . . . ]
void growBunch() {
    qDebug() << "First we create a bunch of objects." << endl;
    QObject bunch;
    bunch.setObjectName("A Stack Object");
    /* other objects are created on the heap */
```

① 这个例子以一部风行于 1969 年至 1974 年的美国家庭情景喜剧(脱线家族)为基础，该剧曾连续风靡过几十年。如果你出生比较晚而没看过这部电视剧，可从维基文章 http://en.wikipedia.org/wiki/The_Brady_Bunch 中寻找一些有趣的事。

```
    Person* mike = new Person("Mike", &bunch);
    Person* carol = new Person("Carol", &bunch);
    new Person("Greg", mike);                                    2
    new Person("Peter", mike);
    new Person("Bobby", mike);
    new Person("Marcia", carol);
    new Person("Jan", carol);
    new Person("Cindy", carol);
    new Person("Alice");                                         3
    qDebug() << "\nDisplay the list using QObject::dumpObjectTree()"
             << endl;
    bunch.dumpObjectTree();                                      4
    cout << "\nReady to return from growBunch() -"
         << " Destroy all local stack objects." << endl;
int main(int , char**) {
    growBunch();
    cout << "We have now returned from growBunch()."
         <<  "\nWhat happened to Alice?" << endl;
    return 0;
}
[ . . . . ]
```

1 本地栈对象——不是指针。

2 无须让指针指向子对象，因为可以借助对象导向而抵达它们那里。

3 Alice 没有父对象——内存泄漏?

4 只有使用的 Qt 库自身被编译时启用了调试选项，dumpObjectTree()的输出才会在屏幕上出现。

以下是这个程序的输出。

```
src/qobject> ./qobject
First we create a bunch of objects.
Constructing Person: Mike
Constructing Person: Carol
Constructing Person: Greg
Constructing Person: Peter
Constructing Person: Bobby
Constructing Person: Marcia
Constructing Person: Jan
Constructing Person: Cindy
Constructing Person: Alice

Display the list using QObject::dumpObjectTree()
QObject::A Stack Object
    QObject::Mike
        QObject::Greg
        QObject::Peter
        QObject::Bobby
    QObject::Carol
        QObject::Marcia
        QObject::Jan
        QObject::Cindy
```

```
Ready to return from growBunch() - Destroy all local stack objects.
Destroying Person: Mike
Destroying Person: Greg
Destroying Person: Peter
Destroying Person: Bobby
Destroying Person: Carol
Destroying Person: Marcia
Destroying Person: Jan
Destroying Person: Cindy
We have now returned from growBunch().
There is no way to access Alice.
src/qobject>
```

8.2.2.1　练习：QObject 的子对象管理

1. 示例 8.4 中，当 `growBunch()` 返回时，哪些本地栈中的对象会被销毁掉?
2. 需要注意的是，Alice 并没有在 `dumpObjectTree()` 的输出中出现。Alice 是何时销毁的?
3. 在 `main.cpp` 中，写出你自己的 `void showTree(QObject* theparent)` 函数。在全部对象创建后，这个函数的输出应具有形式

```
Member: Mike - Parent: A Stack Object
Member: Greg - Parent: Mike
Member: Peter - Parent: Mike
Member: Bobby - Parent: Mike
Member: Carol - Parent: A Stack Object
Member: Marcia - Parent: Carol
Member: Jan - Parent: Carol
Member: Cindy - Parent: Carol
```

4. 修改你的 `showTree()` 函数，以便让它能够与 `dumpObjectTree()` 函数产生同样的输出。

8.3　QApplication 和事件循环

具有 GUI 的交互式 Qt 应用程序与控制台应用程序和过滤器应用程序[①]的控制流有所不同。这是因为它们是基于事件的。在这些应用程序中，对象之间频繁地通过间接对象相互发送消息。这就使得通过手动线性地跟踪整个代码变得异常复杂。

观察者模式

在编写事件驱动的程序时，GUI 视图需要对数据模型对象的状态变化做出响应，以便它们可以显示最新信息。

当任意数据模型对象发生状态改变时，就需要一种间接的方式来提醒(并且可能向外发送额外的信息)观察者。观察者就是一些正在监听(并且响应)状态变化事件的对象。使用这种消息传递机制的设计模式就称为观察者模式(Observer pattern，有时也称为“发布-预定”模式，publish-subscribe pattern)。

① 过滤器应用程序都不是交互的。它通常从标准输入设备读入数据，然后将其写入标准输出设备。

这种模式有许多种不同的实现，但是它们都具有一些共同的特征：

1. 允许实体观察者类与实体主体类之间解耦。
2. 支持广播风格(一对多)的通信。
3. 所采用的从主体向观察者发送信息的机制完全由主体的基类给定。

Qt 的 QEvent 类封装了底层事件的概念。QEvent 类是若干特定的事件类的基类，例如 QActionEvent, QFileOpenEvent, QHoverEvent, QInputEvent, QMouseEvent 等。QEvent 对象可以由窗口系统创建以响应用户的动作(例如，QMouseEvent)，或按照指定的时间间隔(QTimerEvent)完成创建，也可以由应用程序显式地创建。成员函数 type() 会返回一个枚举，其中含有近百个特定的值，以区分不同种类的各式事件(例如，关闭、DragEnter、DragMove、放下、输入、GrabMouse、HoverEnter、KeyPress、MouseButtonDblClick、MouseMove、Resize 等)。

事件循环是一个程序结构，它能够将事件划分优先级，排队并分派给一些对象。编写一个基于事件的应用程序就是实现由函数组成的被动接口，且这些函数仅仅在某些特定事件发生时才会得到调用，如鼠标单击、触摸手势、敲击按键、发射信号、各类窗口管理器事件或来自其他程序的消息等。事件循环通常会一直运行，直到遇到某个终止事件。(例如，用户发出了退出的动作，才会关闭最后的窗口，等等。)

一个典型的 Qt 程序会创建对象，连接各个对象，然后再告诉应用程序开始 exec()。在运行时，应用程序就进入了事件循环。各个对象之间可以通过各种方式相互发送信息。典型的 main() 函数具有如下形式：

```
int main(int argc, char ** argv) {
  QApplication app(argc, argv);
  FancyWidget fwidg;
  fwidg.show();          // returns immediately
  return app.exec();     // enters event loop
}
```

在 main() 的最后，在 return 语句中才出现对 QApplication::exec() 函数的调用。应用程序的整个工作部分开始于该函数的调用，终止于该函数的返回。详细部分位于 FancyWidget 类的定义和实现中。那是相当典型的一个 Qt 事件循环应用程序。

事件与信号和槽的比较

事件可认为是低级消息，目标是某个特定的对象。信号可以认为是高级消息，很有可能会连接到许多槽上。只有在事件循环，特别是由 QApplication::exec() 进入的事件循环中，信号才能发送到槽上。这是因为信号和槽在其外表之下是使用事件循环来传递消息的。

这里的信号是指对事件进行封装的信号。

8.4 Q_OBJECT 和 moc 一览表

QObject 支持一些普通 C++对象通常所没有的特性：

- 信号和槽(参见 8.5 节)。
- 元对象、元属性、元方法(参见第 12 章)。

- qobject_cast(参见 12.2 节)。

这些特性中的某些特性只有借助生成的代码才能够实现。元对象编译器，即 moc，会针对每个使用 Q_OBJECT 宏的 QObject 派生类生成额外的函数，生成的代码可以在名称为 moc_filename.cpp 的文件中找到。

这也就意味着，当 moc 无法发现或处理工程中某个类时，编译器或者链接器会报出的一些莫名其妙含混不清的错误。为了保证 moc 能够处理所编写的全部 QObject 派生类，下面是编写 C++代码和 qmake 工程文件时应该遵守的一些指导原则。

- 每个类的定义都应该放在各自对应的.h 文件中。
- 每个类的实现都应当放在相应的.cpp 文件中。
- 为避免头文件的多次包含，头文件应该“封装”起来(例如，用#ifndef)。
- 每个.cpp 源文件都应当列举在工程文件的 SOURCES 变量中，否则，它将不会被编译。
- 每个头文件都应当列举在.pro 工程文件的 HEADERS 变量中。否则，moc 将不会对其进行预处理。
- Q_OBJECT 宏必须出现在每个 QObject 派生类定义的头文件中，以便让 moc 知道要为其生成代码。

注意

因为每个 Q_OBJECT 宏都会产生代码，所以它需要使用 moc 进行预处理。moc 是在这种假设下工作的：只会从 QObject 类派生了一次。更进一步说，QObject 类应该是其基类列表中的第一个基类。如果在实际应用中不小心多次继承了 QObject，或者它不是继承列表中的第一个基类，那么就可能会从 moc 生成的代码中发现一些非常奇怪的错误。

注意

如果定义了一个 QObject 派生类，构建了一个应用程序，然后才意识到需要在类的定义中添加一个 Q_OBJECT 宏，而且是在该工程是使用了旧的 Makefile 文件构建之后添加的，此时必须使用 qmake 更新一下该 Makefile。

不然，make 并不能智能到在 Makefile 中对这样的文件添加 moc 的步骤。通常执行 clean rebuild 命令并不能修复这种问题。该问题经常使得经验不足的 Qt 开发者非常头痛。关于该错误消息的更多信息，可参阅附录 C 的 C.3.1 节。

8.5　信号和槽

信号是在类定义中给出的类似于 void 函数声明的一种消息。它有参数列表却没有函数体。信号是一个类的接口的一部分。它看起来像函数，但不用同样的方式进行调用——它被此类的对象发射。

槽通常是一个 void 成员函数。它可以像普通的成员函数一样进行调用，或者可以由 QMetaObject 系统进行间接调用。

一个对象的信号可以与一个或者多个对象的槽相连接，前提是这些对象存在并且参数列表从信号到槽都是赋值兼容的[①]。连接语句的语法是

```
bool QObject::connect(senderQObjectPtr,
                      SIGNAL(signalName(argumentList)),
                      receiverQObjectPointer,
                      SLOT[②](slotName(argumentList))
                      optionalConnectionType);
```

任何拥有信号的 QObject 都可以发射出那样的信号。这就会引起对全部连接的槽的间接调用。

类似于函数调用，在发射语句中传递的参数可以在槽函数内通过参数进行访问。参数列表是从一个对象向另一个对象传递信息的方式。optionalConnectionType 让你可以明确说明，你是否希望从发射点处同步(阻塞)或者非同步(排队)地调用目标槽[③]。

在 1.11 节，我们讨论过一个用到了 QInputDialog 窗件(见图 8.5)的例子。在运行那个应用程序时，用户通过输入一个值，然后再通过左键单击 Cancel 按钮或者 OK 按钮，实现与第一个对话框的交互。

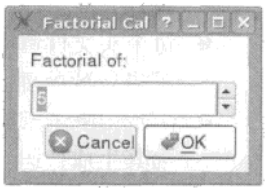

图 8.5 QInputDialog

鼠标左键的释放事件，是鼠标单击中的最后一步，引起选中按钮窗件发射 clicked()信号。那个信号是一系列说明鼠标指针(在按钮的矩形框内)位置的鼠标底层事件的集合，并可以确保鼠标按钮操作的正确次序。例如，鼠标左键按下然后再释放时，可能仍旧还在该矩形框内。换句话说，鼠标事件已经组合起来构成了 clicked()信号。设计良好的窗件 API 应该适当包含一组信号，使得用户不必与底层事件打交道，除非是要开发自定义窗件。


提示

如果有多个信号连接到同一个槽上且需要知道是哪一个 QObject 发射的信号，则可以在该槽中调用 sender()，它会返回一个指向那个对象的指针。

8.5.1 更多探讨

与 Qt 的 QObject 模型类似，还有其他一些信号和槽的开源实现。其中之一称为 XLObject[④]。与 Qt 相比，它不要求任何 moc 风格的预处理，但非常依赖模板，因此只有(2002 年以后的)现代 C++编译器才支持它。Boost 库[⑤]也包含一种信号和槽的实现。

8.6 QObject 的生命周期

这一节将给出一些很好地管理 QObject 生命周期的实践经验。

① 当列表是赋值兼容时，意味着对应的参数必须是兼容的。在 Qt 中，槽必须至少有与信号一样多的参数。槽可以忽略多余的参数。

② 或者 SIGNAL——从一个信号连接另一个信号也是可行的。

③ 连接不会局限于当前线程(参见 17.2 节)。

④ 参见http://sourceforge.net/projects/xlobject。

⑤ 参见http://www.boost.org。

警告

确保每一个 QObject 在 QApplication 之后创建，在 QApplication 销毁前销毁，这一点至关重要。静态存储区创建的对象将在 main() 返回后才被销毁，这就太迟了。这就意味着绝不应该定义静态存储类的 QObject。

栈还是堆

一般情况下，没有父对象的 QObject 应当在栈上创建，或者定义成另一个类的子对象。有父对象的 QObject 不应在栈上创建。因为那样的话，它有可能会被删除两次。在堆上创建的所有 QObject 都应当或者是有父对象的，或者是由其他对象进行管理的。

提示

不推荐直接删除 QObject。在带有事件循环的程序中，最好是利用 QObject::deleteLater() 来删除 QObject。这样做，可以在应用程序处理事件并在当前槽返回之后就安排该对象的销毁。

希望能够在槽内删除信号的 sender()，这么做实际上也是必须的(参见示例 17.20)。

8.7 QTestLib

编写测试代码最常用的方式是将其组织在一个基于单元的框架内。Qt 4 引入了由辅助宏和测试运行器构成的 QTestLib 框架，用来简化使用 Qt 编写的应用程序和库的单元测试的编写。在这个框架中，所有的公有方法都位于 QTest 命名空间中。下面的几个示例会用到这个框架。

TestClass 是一个派生自 QObject 的类，有一些用于一个或多个测试函数的私有槽。测试用例就是一系列将要执行的测试函数。每个测试用例都必须在其自己的工程中，该工程有一个含有 QTEST_MAIN() 宏的 source 模块。QTEST_MAIN() 用来说明哪个类是这个测试的入口点。它会扩展成一个创建实例并依照声明次序执行其私有槽的 main() 函数。还可以提供一些用于初始化和清理的其他方法，分别是 initTestCase() 和 cleanupTestCase()，会分别在测试用例的开头和结尾处调用它们。

为了演示 QTestLib 的用法，我们为 QCOMPARE 和 QVERIFY 写一个测试，这两个宏会在 Qt 的测试用例中用作断言。这些宏仅能用在测试类中，而 Q_ASSERT 则可用于任何地方。当断言失败时，它们能比 Q_ASSERT 提供更多的信息。

任何用到了 QTestLib 模块的工程，都必须在 .pro 文件中用下面这行代码进行启用：

```
CONFIG += qtestlib
```

如示例 8.5 所示，第一步要定义一个含有测试函数的 QObject 派生类。包含 QtTest 头文件并把各个测试函数声明成私有槽是必要的。

示例 8.5　src/libs/tests/assert/testassertequals.h

```
[ . . . . ]

#include <QtTest>
```

```
class TestAssertEquals:public QObject {
    Q_OBJECT
private slots:
    void test ();
};

[ . . . . ]
```

这个测试会试验所有各种验证表达式的类型。示例 8.6 是处理 bool 表达式的部分代码。

示例 8.6 src/libs/tests/assert/testassertequals.cpp

```
[ . . . . ]

void TestAssertEquals::test () {
    qDebug() << "Testing bools";
    bool boolvalue = true;
    QVERIFY (1);
    QVERIFY (true);
    QVERIFY (boolvalue);
    qDebug () << QString ("We are in file: %1  Line: %2").
            arg (__FILE__).arg (__LINE__);
    QCOMPARE (boolvalue, true);        1
```

1 用布尔值测试 EQUALS。

示例 8.7 是处理 QString 表达式部分的实现代码。

示例 8.7 src/libs/tests/assert/testassertequals.cpp

```
[ . . . . ]

    qDebug() << "Testing QStrings";
    QString string1 = "apples";                                    1
    QString string2 = "oranges";
    QString string3 = "apples";
    QCOMPARE ("apples", "apples");                                 2
    QCOMPARE (string1, QString("apples"));
    QCOMPARE (QString("oranges"), string2);
    QCOMPARE (string1, string3);
    QVERIFY (string2 != string3);
```

1 用字符串值测试 EQUALS。

2 测试 QString 和 char *的比较。

示例 8.8 用来处理 QDate 和 QVariant 表达式。通常，test()函数会在第一个测试失败的地方停下，当 QVERIFY(*condition*)碰到 *condition* 的值是 false 或者是碰到 QCOMPARE(*actual, expected*)在 *actual* 和 *expected* 不相等时，就会导致失败。因此，示例 8.8 中故意包含了一个 QCOMPARE()错误。要继续测试，就需要在这个故意失败的前面放上 QEXPECT_FAIL()宏。

示例 8.8 src/libs/tests/assert/testassertequals.cpp

```
[ . . . . ]

    qDebug() << "Testing QDates";
    QString datestr ("2010-11-21");
```

```
    QDate dateobj = QDate::fromString (datestr, Qt::ISODate);
    QVERIFY (dateobj.isValid ());
    QVariant variant (dateobj);
    QString message(QString ("comparing datestr: %1 dateobj: %2 variant: %3")
           .arg (datestr).arg (dateobj.toString ()).arg (variant.toString ()));
    qDebug() << message;
    QCOMPARE (variant, QVariant(dateobj));                          1
    QCOMPARE (QVariant(dateobj), variant);
    QCOMPARE (variant.toString(), datestr);                         2
    QCOMPARE (datestr, variant.toString());
    QEXPECT_FAIL("","Keep going!", Continue);
    QCOMPARE (datestr, dateobj.toString());                         3
```

1 比较 QDate 和 QVariant。

2 比较 QVariant 和 String。

3 比较 QDate 和 QString。

示例 8.9 处理含有 int，long 和 double 项的表达式。在示例 8.9 中我们插入了两个 QVERIFY()失败。我们在第一个失败的前面放一个 QEXPECT_FAIL()宏。允许第二个失败可以停止该测试。注意位于函数定义下面的宏 QTEST_MAIN，它将生成适当的 main()函数代码。

示例 8.9　src/libs/tests/assert/testassertequals.cpp

```
[ . . . . ]

    qDebug() << "Testing ints and doubles";
    int i = 4;                                                      1
    QCOMPARE (4, i);
    uint u (LONG_MAX + 1), v (u / 2);
    QCOMPARE (u, v * 2);
    double d (2. / 3.), e (d / 2);
    QVERIFY (d != e);
    QVERIFY (d == e*2);
    double f(1./3.);
    QEXPECT_FAIL("","Keep going!", Continue);
    QVERIFY (f * 3 == 2);
    qDebug() << "Testing pointers";
    void *nullpointer = 0;
    void *nonnullpointer = &d;
    QVERIFY (nullpointer != 0);
    qDebug() << "There is one more item left in the test.";
    QVERIFY (nonnullpointer != 0);
}

// Generate a main program
QTEST_MAIN(TestAssertEquals)
```

1 整数测试。

示例 8.10 给出了这一测试的输出。值得注意的是，并没有看到最后的 qDebug()的输出信息。

示例 8.10　src/libs/tests/assert/testassert.txt

```
********* Start testing of TestAssertEquals *********
Config: Using QTest library 4.6.2, Qt 4.6.2
```

```
PASS   : TestAssertEquals::initTestCase()
QDEBUG : TestAssertEquals::test() Testing bools
QDEBUG : TestAssertEquals::test() "We are in file:
testassertequals.cpp  Line: 15"
QDEBUG : TestAssertEquals::test() Testing QStrings
QDEBUG : TestAssertEquals::test() Testing QDates
QDEBUG : TestAssertEquals::test() "comparing datestr: 2010-11-21
dateobj: Sun Nov 21 2010 variant: 2010-11-21"
XFAIL  : TestAssertEquals::test() Keep going!
   Loc: [testassertequals.cpp(46)]
QDEBUG : TestAssertEquals::test() Testing ints and doubles
XFAIL  : TestAssertEquals::test() Keep going!
   Loc: [testassertequals.cpp(59)]
QDEBUG : TestAssertEquals::test() Testing pointers
FAIL!  : TestAssertEquals::test() 'nullpointer != 0' returned FALSE.
()
   Loc: [testassertequals.cpp(63)]
PASS   : TestAssertEquals::cleanupTestCase()
Totals: 2 passed, 1 failed, 0 skipped
********* Finished testing of TestAssertEquals *********
```

8.8 练习：QObject, QApplication，信号和槽

1. 重写 4.4 节中的 Contact 和 ContactList，以使它们都从 QObject 派生出来。当在 ContactList 中添加一个 Contact 时，要让 Contact 成为 ContactList 的子对象。
2. 用你在本练习中编写的客户代码将 Contact 和 ContactList 替换成新版本。

8.9 复习题

1. 当 QObject A 是 QObject B 的父对象时，含义是什么？
2. 哪些 QObject 不需要父对象？
3. 当一个 QObject 重父化(reparent)时，会发生什么事情？
4. 为什么 QObject 的复制构造函数不是公有的？
5. 什么是组合模式？
6. 在什么情况下，QObject 既可以是复合对象又可以是组件对象？
7. 如何访问一个 QObject 的子对象？
8. 什么是事件循环？它是如何启动的？
9. 什么是信号？如何调用一个信号？
10. 什么是槽？如何调用一个槽？
11. 信号和槽是如何进行连接的？
12. 在多个信号连接到同一个槽的情况下，如何判定是哪一个 QObject 发射的信号？
13. 信息是如何从一个对象传递给另外一个对象的？
14. 一个类从 QObject 派生多次可以引起一些问题。什么情况下会意外地发生这种情况？
15. 值类型和对象类型之间有何区别？试举出几个例子。