第17章 并 发

QProcess 和 QThread 提供了两种并发方式。本章将讨论进程和线程的创建及其通信方式，同时还会介绍进程与线程的监视和调试技术。

17.1 QProcess 和进程控制

QProcess 是一个能够非常方便(而且跨平台)的用于启动和控制其他进程的类。它从 QObject 派生而来，可充分利用信号和槽来简化与其他 Qt 类的“交互”。

现在考虑一个简单的例子，启动一个进程并观察其持续运行的输出结果[①]。示例 17.1 中给出了一个 QProcess 的简单派生类的定义。

示例 17.1 src/logtail/logtail.h

```
[ . . . . ]
#include <QObject>
#include <QProcess>
class LogTail : public QProcess {
    Q_OBJECT
  public:
    LogTail(QString fn = QString());
    ~LogTail();
  signals:
    void logString(const QString &str);

  public slots:
    void logOutput();
};
[ . . . . ]
```

一个 QProcess 可以使用 start() 函数来启动[②]另外一个进程。新进程将会成为一个子进程并且在父进程终止时而随之终止[③]。示例 17.2 给出了 LogTail 类构造函数和析构函数的实现。

示例 17.2 src/logtail/logtail.cpp

```
[ . . . . ]

LogTail::LogTail(QString fn) {
```

① tail -f 将会始终运行，随时显示所有添加到文件中的内容，这对于显示一个运行进程的日志文件的内容是非常有用的。

② 强调 QProcess API 的跨平台价值是因为这样一个事实，一个进程用于启动另一个进程的机制在两大主流操作系统家族中是非常不同的。有关这一机制在*nix 系统中的做法的更多信息，可参阅维基百科中的文章 http://en.wikipedia.org/wiki/Fork_%28operating_system%29。而描述 Microsoft Windows 中的方法位于维基百科中的文章 http://en.wikipedia.org/wiki/Spawn_%28computing%29。

③ 也可以使用 startDetached() 函数来启动一个在父进程结束之后仍能存活的进程。

```
    connect (this, SIGNAL(readyReadStandardOutput()),
        this, SLOT(logOutput()));                                    1
    QStringList argv;

    argv << "-f" << fn;                                              2
    start("tail", argv);                                             3
}
LogTail::~LogTail() {
    terminate();                                                     4
}
```

1 当输入准备好时，会调用这个槽。

2 `tail -f` 文件名。

3 立即返回，并且现在会有一个“依附”于当前进程的子进程在独立运行。当调用进程退出时，新产生的子进程也会终止。

4 试图终止此进程。

子进程可以看成是一个预先定义了两个输出通道的顺序 I/O 设备，这两个输出通道分别代表了两个独立的数据流：`stdout` 和 `stderr`。父进程可以使用函数 `setReadChannel()` 来选择一个输出通道(默认是 `stdout`)。当子进程中被选中通道的数据可用时，它将会发射出信号 `readyRead()`。此时父进程就可以通过调用函数 `read()`，`readLine()` 或者 `getChar()` 来读取其输出结果。如果子进程启用了标准输入，那么父进程就可以使用 `write()` 函数向其发送数据。

示例 17.3 给出了 `logOutput()` 槽的具体实现，这个槽与 `readyReadStandardOutput()` 信号相连接，并且使用了 `readAllStandardOutut()` 方法，因此它仅仅关注 `stdout`。

示例 17.3 src/logtail/logtail.cpp

```
[ . . . . ]

// tail sends its output to stdout.
void LogTail::logOutput() {                                          1
    QByteArray bytes = readAllStandardOutput();
    QStringList lines = QString(bytes).split("\n");
    foreach (QString line, lines) {
        emit logString(line);
    }
}
```

1 无论什么时候有要读取的输入，都会调用该槽。

各个信号的消除用法是需要有一个读循环。当没有更多的输入数据要读入时，就不会再调用槽。信号和槽使得并行代码的可读性更好，原因是它们隐藏了事件处理和派发的代码。示例 17.4 给出了一些相关的客户代码。

示例 17.4 src/logtail/logtail.cpp

```
[ . . . . ]

int main (int argc, char* argv[]) {
    QApplication app(argc, argv);
```

```
    QStringList al = app.arguments();
    QTextEdit textEdit;
    textEdit.setWindowTitle("Debug");
    textEdit.setWindowTitle("logtail demo");
    QString filename;
    if (al.size() > 1) filename = al[1];
    LogTail tail(filename);                                          1
    tail.connect (&tail, SIGNAL(logString(const QString&)),
        &textEdit, SLOT(append(const QString&)));
    textEdit.show();
    return app.exec();
}
```

1 创建对象，同时启动进程。

一旦数据行在指定的日志文件中出现，这个应用程序就会在 `QTextEdit` 中进行追加。为了演示 `LogTail` 这一应用程序，需要一个诸如某种活动日志的文本文件，它可以随着向其添加行而不断增长。如果无法找到一个这样的文件，可以使用诸如 top 这样的工具自行创建一个，`top` 是一个在典型 *nix 主机上都可用的一个实用工具。通常情况下，不带命令行参数的 `top` 会产生一个纯文本、格式化的屏幕列表，其中会按照资源的使用情况来降序排列当时使用系统资源最多的 25 个正在运行的进程。显示的开始位置是系统使用情况的总体说明，并且每隔几秒都会更新整个显示。`top` 会一直运行，直到它被用户终止。在这个例子中，我们可以用以下命令行参数来启动 `top`：

- `-b`，将其置为批处理模式，以便可以重定向其输出。
- `-d 1.0`，指明两次更新之间的间隔秒数。
- `> toplog`，将输出重定向到 `toplog` 文件中。
- `&`，将 top 作为后台进程运行。

然后，对所得的结果文件运行 `logtail` 例子：

```
top -b -d 1.0 > toplog &
./logtail toplog
```

图 17.1 是该程序运行过程中的屏幕截图。

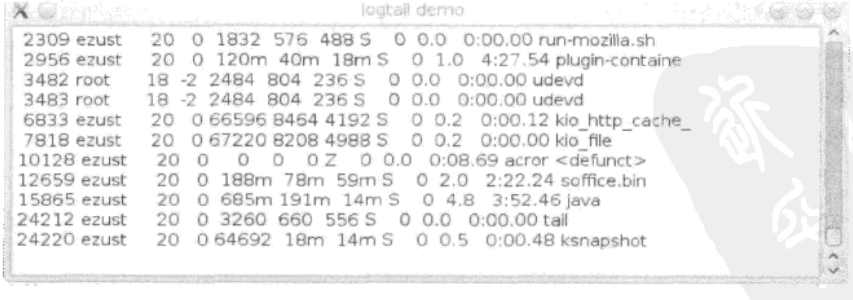

图 17.1 使用中的 `LogTail`

这个示例程序会一直运行直至被迫终止，随后必须杀死 `top` 作业，它的作业编号 (job number) 和进程 ID 在启动后就会显示出来。使用 `bash` 时，杀死作业时只需使用作业编号 `%1` 即可。

```
src/logtail> top -b -d 1.0 > toplog &
[1] 24209
src/logtail> ./logtail toplog
      [[ logtail was terminated here. ]]
QProcess: Destroyed while process is still running.
src/logtail> kill 24209
src/logtail>
```

17.1.1 进程和环境

环境变量是<name, value>这样的名/值字符串对，可以非常容易地存储在一个映射或者哈希表中。环境变量名必须是合法的标志符，按照惯例，通常不会含有小写字母。每个正在运行的进程都有一个由环境变量的集合所构成的环境。大多数编程语言都会支持一种获取和设置这些变量的方法。

最常用的一些环境变量如下。

1. PATH。用于搜索可执行文件(在 Windows 上也包括动态链接库)的一个目录列表。
2. HOME。主目录的位置。
3. CPPLIBS。来自本书源代码示例程序的 C++库的安装位置[①]。
4. HOSTNAME(*nix)或者 COMPUTERNAME (win32)。通常用于获取机器名。
5. USER(*nix)或者 USERNAME (Win32)。通常用于获取当前登录的用户 ID。

环境变量和它们的值通常由父进程设置。依赖特殊变量或者值的程序通常是不可移植的，但在一定程度上也取决于它们的父进程。环境变量为进程间交流信息提供了一种便利的跨语言机制。

操作系统允许用户给进程及其将来的子进程设置环境变量。下面是一些例子。

- Microsoft Windows 桌面——通过 Start->Settings->System->Advanced->Environment Variables 可以得到类似图 17.2 的设置对话框。
- Microsoft 命令提示符——可以设置 VARIABLE = value 和 echo %VARIABLE%。在较新版本的 Windows 中也可以看到 setx 命令。
- Bash 命令行——利用 export VARIABLE = value 和 echo $VARIABLE。

许多编程语言也支持环境变量的获取和设置，列举如下。

- C++/Qt：QProcess::environment()函数和 setEnvironment()函数。
- Python：os.getenv()函数和 os.putenv()函数。
- Java：ProcessBuilder.environment()函数。
- C：<cstdlib>中的 getenv()函数和 putenv()函数(参见附录 B)。

所运行的任何程序涉及的进程树都可能有好几层深。这是因为，典型的桌面操作系统环境是由许多并发运行的进程一起组成的。图 17.3 中，缩进排列的层次显示了进程之间的父-子关系。在同一个缩进层次上的进程之间是兄弟关系(即它们拥有相同的父对象)。图 17.3 给出了在典型的 KDE/Linux 系统上该 environment 例子从 konsole 运行时各个进程的生命周期和进程之间的父-子关系。

① 参见 http://www.distancecompsci.com/dist/。

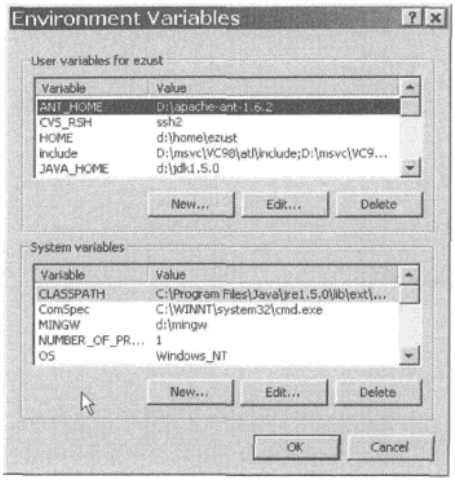

图 17.2 Windows 环境变量

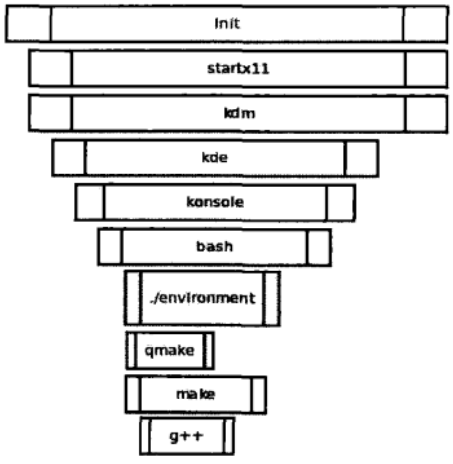

图 17.3 KDE/Linux 的进程层次

只要程序 A 启动了程序 B，进程 A 就是进程 B 的父亲。在创建进程 B 时，进程 B 会继承(作为一个副本)进程 A 的环境。在进程 B 内部对环境的改变将会影响进程 B 及其子孙进程，而这些修改对于进程 A 来说是永远不可见的。

示例 17.5 展示了提供给 `setenv()` 函数的值可传播到其子孙进程中。

示例 17.5 src/environment/setenv.cpp

```
#include <QCoreApplication>
#include <QTextStream>
#include <QProcess>
#include <QCoreApplication>
#include <QTextStream>
#include <QStringList>
#include <cstdlib>

class Fork : public QProcess {
    public:
    Fork(QStringList argv = QStringList() ) {
        execute("environment", argv);
    }
    ~Fork() {
        waitForFinished();
    }
};

QTextStream cout(stdout);
int main(int argc, char* argv[]) {

    QCoreApplication qca(argc, argv);
    QStringList al = qca.arguments();
    al.removeAt(0);
    bool fork=al.contains("-f");
    if(fork) {
       int i = al.indexOf("-f");
```

```
        al.removeAt(i);
    }

    QStringList extraVars;
    if (al.count()  > 0) {
        setenv("PENGUIN", al.first().toAscii(), true);
    }
    cout << " HOME=" << getenv("HOME") << endl;
    cout << " PWD=" << getenv("PWD") << endl;
    cout << " PENGUIN=" << getenv("PENGUIN") << endl;

    if (fork) {
        Fork f;
    }
}
```

1 将同一应用程序作为子进程运行。

当这个程序运行时，其输出结果如下所示。

```
src/environment> export PENGUIN=tux
src/environment> ./environment -f
HOME=/home/lazarus
PWD=src/environment
PENGUIN=tux
HOME=/home/lazarus
PWD=src/environment
PENGUIN=tux
src/environment> ./environment -f opus
HOME=/home/lazarus
PWD=src/environment
PENGUIN=opus
HOME=/home/lazarus
PWD=src/environment
PENGUIN=opus
src/environment> echo $PENGUIN
tux
src/environment>
```

17.1.2 Qonsole：用 Qt 编写一个 Xterm

命令行 shell 从用户读入命令，然后打印程序的输出结果。这个例子中用一个 QTextEdit 来给另外一个正在运行的进程提供输出结果视图。在这里这个进程是指 bash，它是大多数 *nix 系统中默认的命令行 shell 解释程序(Windows 上是 cmd)。Qprocess 是一个模型，代表一个正在运行的进程。图 17.4 所示为 Qonsole 运行时的一个截图，这是我们在图形用户界面(GUI)中提供命令 shell 第一次尝试[①]。

因为 Qonsole 把信号连接到了槽上并以此来处理用户交互问题，所以可以把它看成是一个控制器。而又因为它继承自 QMainWindow 类，所以其中也包含了一些视图代码。图 17.5 中的 UML 框图给出了此应用程序中的各个类之间的关系。

① 本书作者在 Qonsole 发音上尚存在一些分歧。两个选择或许应当是“Chon-sole”(使用标准的中文发音惯例，就好像 Qing)和“Khonsole”(使用标准的阿拉伯发音方法，就好像 Qatar)。

图 17.4　Qonsole1

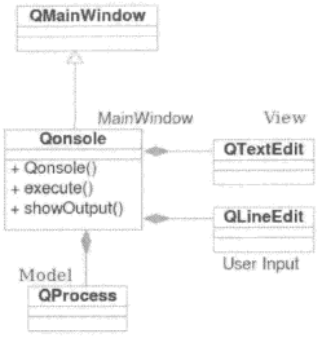

图 17.5　Qonsole UML：模型和视图

示例 17.6 给出了 Qonsole 类的定义。

示例 17.6　src/qonsole/qonsole1/qonsole.h

```
[ . . . . ]
class Qonsole : public QMainWindow {
    Q_OBJECT
 public:
    Qonsole();
 public slots:
    void execute();
    void showOutput();
 private:
    QTextEdit* m_Logw;
    QLineEdit* m_InputArea;
    QProcess* m_Shell;
};
[ . . . . ]
```

示例 17.7 中，我们可以看到构造函数是如何创建 Qonsole 的框架的，同时还可以看到 Qonsole 组件之间的一些重要连接。

示例 17.7　src/qonsole/qonsole1/qonsole.cpp

```
[ . . . . ]

Qonsole::Qonsole() {
   m_Logw = new QTextEdit();
   m_Logw->setReadOnly(true);
   setCentralWidget(m_Logw);
   m_InputArea = new  QLineEdit();
   QDockWidget* qdw = new QDockWidget("Type commands here");
   qdw->setWidget(m_InputArea);
   addDockWidget(Qt::BottomDockWidgetArea, qdw);
   connect (m_InputArea, SIGNAL(returnPressed()),
                         this, SLOT(execute()));

   m_Shell = new QProcess(this);
   m_Shell->setReadChannelMode(QProcess::MergedChannels);
   connect (m_Shell, SIGNAL(readyReadStandardOutput()),
```

```
                        this, SLOT(showOutput()));
#ifdef Q_OS_WIN
   m_Shell->start("cmd", QStringList(), QIODevice::ReadWrite);
#else
   m_Shell->start("bash", QStringList("-i"), QIODevice::ReadWrite);          2
#endif

}
```

1 将 stdout 和 stderr 合二为一。

2 以交互方式运行 bash。

不管 shell 输出什么结果，Qonsole 都会将其发送到 QTextEdit。而无论用户何时按下回车键，Qonsole 都会捕捉 QLineEdit 中的所有文本并将其发送给 Shell，由 shell 将这些文本作为一个命令进行解释，正如示例 17.8 所示。

示例 17.8 src/qonsole/qonsole1/qonsole.cpp

```
[ . . . . ]

void Qonsole::showOutput() {                                                 1
    QByteArray bytes = m_Shell->readAllStandardOutput();
    QStringList lines = QString(bytes).split("\n");
    foreach (QString line, lines) {
        m_Logw->append(line);
    }
}

void Qonsole::execute() {
    QString cmdStr = m_InputArea->text() + "\n";
    m_InputArea->setText("");
    m_Logw->append(cmdStr);
    QByteArray bytes = cmdStr.toUtf8();                                      2
    m_Shell->write(bytes);                                                   3
}
```

1 只要输入准备好，就会调用该槽。

2 8 位的 Unicode 传输模式。

3 将数据发送到 Shell 子进程的 stdin 流。

示例 17.9 给出了启动这个应用程序的客户代码。

示例 17.9 src/qonsole/qonsole1/qonsole.cpp

```
[ . . . . ]

#include <QApplication>

int main(int argc, char* argv[]) {
   QApplication app(argc, argv);
   Qonsole qon;
   qon.show();
   return app.exec();
}
```

17.1.3 带有键盘事件的 Qonsole

在先前的例子中，Qonsole 为用户输入提供一个单独的窗件。为了提供更为真实的终端经验，用户应该能够在命令输出窗口中输入命令。为了满足这一基本需求，Qonsole 需要捕捉键盘事件。其中的第一步，就是重写 QObject 基类的 eventFilter() 方法。示例 17.10 中修改的类定义中可以看到这一点。

示例 17.10 src/qonsole/keyevents/qonsole.h

```
[ . . . . ]
class Qonsole : public QMainWindow {
    Q_OBJECT
 public:
    Qonsole();
 public slots:
    void execute();
    void showOutput();
    bool eventFilter(QObject *o, QEvent *e)  ;
 protected:
    void updateCursor();
 private:
    QString m_UserInput;
    QTextEdit* m_Logw;
    QProcess* m_Shell;
};
[ . . . . ]
```

正如 8.3 节中讨论的那样，事件是一个从 QEvent 派生的对象。在一个应用程序的上下文中，这样一个 QEvent 与作为预期接收者的 QObject 相关联。负责接收的对象会有一个专门处理此事件的函数。事件过滤器首先对 QEvent 进行检查，然后决定是否允许其接收者进行处理。我们给修改后的 Qonsole 应用程序提供了一个 eventFilter() 函数，它的作用是过滤来自 m_Logw 的键盘事件，其中 m_Logw 扩展自 QTextEdit。Qonsole 扩展自 QmainWindow，它是所有这些事件的预期接收者。此函数的具体实现会在示例 17.11 中给出。

示例 17.11 src/qonsole/keyevents/qonsole.cpp

```
[ . . . . ]

bool Qonsole::eventFilter(QObject* o, QEvent* e) {
    if (e->type() == QEvent::KeyPress) {
        QKeyEvent* k = static_cast<QKeyEvent*> (e);
        int key = k->key();
        QString str = k->text();
        m_UserInput.append(str);
        updateCursor();
        if ((key == Qt::Key_Return) || (key == Qt::Key_Enter) ) {
#ifdef Q_WS_WIN                                                    1
            m_UserInput.append(QChar(0x000A));
#endif
            execute();
            return true;                                           2
```

```
        }
        else {
            m_Logw->insertPlainText(str);
            return true;
        }
    }
    return false;                                                    3
}
```

1 Windows 进程需要的是一个回车+换行，并非只是一个回车。

2 我们处理了这个事件。这样，其他的窗件也就无法再感知此事件。

3 不要处理(touch)该事件。

当按下回车键时，就会调用成员函数 `execute()`，以便可以让命令字符串发送到 Shell 并随之得到重置。示例 17.12 给出了这两个函数的具体实现。

示例 17.12 src/qonsole/keyevents/qonsole.cpp

```
[ . . . . ]

void Qonsole::updateCursor() {
    QTextCursor cur = m_Logw->textCursor();
    cur.movePosition(QTextCursor::End, QTextCursor::KeepAnchor);
    m_Logw->setTextCursor(cur);
}

void Qonsole::execute() {
    QByteArray bytes = m_UserInput.toUtf8();
    m_Shell->write(bytes);
    m_UserInput = "";
}
```

剩下的事情就是在 `m_Logw` 上调用其基类函数 `installEventFilter()`，该窗件的事件也正是我们想要捕捉的事件。这部分动作是在构造函数中完成的，如示例 17.13 所示。

示例 17.13 src/qonsole/keyevents/qonsole.cpp

```
[ . . . . ]

Qonsole::Qonsole() {
   m_Logw = new QTextEdit;
   setCentralWidget(m_Logw);
   m_Logw->installEventFilter(this);                                 1
   m_Logw->setLineWrapMode(QTextEdit::WidgetWidth);
   m_Shell = new QProcess();
   m_Shell->setReadChannelMode(QProcess::MergedChannels);
   connect (m_Shell, SIGNAL(readyReadStandardOutput()),
            this, SLOT(showOutput()));
#ifdef Q_WS_WIN
   m_Shell->start("cmd", QStringList(), QIODevice::ReadWrite);
#else
   m_Shell->start("bash", QStringList("-i"), QIODevice::ReadWrite);
#endif
}
```

1 截取送往 `QTextEdit` 的事件。

17.1.4 练习：QProcess 和进程控制

1. 在 Qonsole 的这一版本中，尚无法恰当地处理退格键。增加一个事件处理器，让它可以对退格键进行适当的响应。
2. 修改 Qonsole，使其能够在单独的标签中支持多个并行终端。
3. 根据美国国家标准与技术研究所[①](NIST)的研究成果，哈希函数可接受称为消息的二进制数据，并可生成一种称为消息摘要(message digest)的简明表示。加密哈希函数是一个旨在达到一定的安全属性的哈希函数。联邦信息处理标准(Federal Information Processing Standard) 180-2，即安全哈希标准(Secure Hash Standard)，给出了 5 个用在计算方面的加密哈希函数算法：SHA-1, SHA-224, SHA-256, SHA-384 和 SHA-512。
   对于任意给定的数据块，一个好的加密哈希函数必须能够可靠地生成实质唯一的摘要。也就是说，绝不可能有其他任何数据块可以使用该函数得到同样的摘要。哈希法是一种单向操作。也就是说，这一过程通常是不可逆的，也不可能从该摘要中生成数据块。
   通过仅对每个密码的摘要进行存储即可处理安全密码。当用户登录并输入密码，那个字符串会即刻得到散列处理，其结果摘要会和存储的摘要加以比对。如果两者匹配，用户就是合法的。否则，登录就不成功。用户的密码永远不会被存储并且也只是在计算密码摘要时存在于内存中。
   Qt 有一个 QCryptographicHash 类，它提供了一个用于计算给定 QByteArray 加密哈希值的哈希函数。Qt 4.7 中提供了 SHA-1, MD4 和 MD5[②]。
   a. 编写一个带有两个命令行参数的简单应用程序：一个是用于哈希处理的字符串，一个用来说明所要使用的算法。该应用程序应当把摘要的处理结果发送到标准输出终端上。例如：

      ```
      crhash "my big secret" md5
      ```

      将输出由二进制数据所组成的摘要。
   b. 把上一题中的 crhash 应用程序用作单独进程使用，编写一个管理俱乐部成员数据的应用程序，包括用户的 ID、密码、电子邮件地址、街道地址、城市、州、邮政编码和电话号码。密码应当只存储成摘要。确保对组成成员的数据进行过适当的序列化处理。

17.2 QThread 和 QtConcurrent

在 Qt 之前，对于 C++开源程序开发人员来说，通常是没有跨平台多线程可用的，因为多线程是相对较新的概念，并且在每个操作系统中多线程的处理方式都有所不同。现在，在大多数的操作系统和许多现代编程语言中都提供多线程。多核处理器也已相当普遍，因此，来自同一进程中的多个线程都可被现代操作系统分配到不同的内核上运行。

① 参见http://csrc.nist.gov/groups/ST/hash/index.html。

② MD4 和 MD5 是由 Ron Rivest 在 SHA-1 之前设计用于消息—摘要的算法，而 SHA-1 已经被 SHA-2 的系列哈希函数所取代。更多详细情况，可以参阅http://en.wikipedia.org/wiki/MD5。

Qt 的线程模型允许线程的优先次序和控制。QThread 是一个低级(low-level)类，适合用于显式地构建长期运行的线程。

QtConcurrent 是一个命名空间，提供了用于编写并发软件的更高层次的类和算法。该命名空间中有一个重要的类，QThreadPool，这是一个管理线程池的类。每个 Qt 应用程序都有一个 QThreadPool::globalInstance()函数，它带有一个推荐的最大线程数，在大多数系统上，处理核的数量就是该值的默认值。

借助于 QtConcurrent 中函数式的 map/filter/reduce 算法(它们可将函数并行用到容器中的每个项)，通过将进程分布在由线程池管理的多个线程上，可编写一个能够自动利用系统多核的程序。另外，在命令模式和利用 QtConcurrent:: run()工作时可把 QRunnable 用作基类。在这些情况下，无须显式地创建线程或者直接管理它们，只需简单地把工作片段描述为具有正确接口的对象即可。

为什么使用线程

有时候，使用线程给软件所带来的复杂程度会超过所带来的性能优势。如果一个程序的性能受限于输入/输出，则将 CPU 的工作分散于多个线程将不会为程序的整体性能带来明显改善。然而，如果程序需要做相当数量的高级计算工作，并且也有空闲的处理核，多线程就可以提高性能。

线程指南

一般情况下，要尽可能避免使用线程，而是用 Qt 事件循环与 QTimer、非阻塞 I/O 操作、信号以及短持续时间槽相结合的方法来代替。此外，可以在主线程中长期运行的循环调用 QApplication::processEvents()，以使执行工作时图形用户界面可以保持响应[①]。

要驱动动画(animation)，建议使用 QTimer，QTimeLine 或者动画框架(Animation Framework)[②]。这些 API 并不需要额外创建其他线程。它们允许访问动画代码中的 GUI 对象而且不会妨碍图形用户界面的响应。

如果要完成 CPU 密集型工作并希望将其分配给多个处理核，可以把工作分散到 QRunnable 并通过以下这些推荐做法来实现线程的安全。

- 无论何时，都尽可能使用 QtConcurrent 算法把 CPU 密集型计算工作分散给多线程，而不是自己编写 QThread 代码。
- 除了主线程以外，不要从其他任何线程访问图形用户界面(这也包括那些由 QWidget 派生的类、QPixmap 和其他与显卡相关的类)。这包括读取操作，比如查询 QLineEdit 中输入的文本。
- 要其他线程中处理图像，使用 QImage 而不是 QPixmap。
- 不要调用 QDialog::exec()或者从除主线程之外的任何线程创建 QWidget 或 QIODevice 的子类。

① 参见http://doc.trolltech.com/qq/qq27-responsive-guis.html#manualeventprocessing。
② 在 Qt 4.6 中引入。

- 使用 QMutex，QReadWriteLock 或者 QSemaphore 以禁止多个线程同时访问临界变量。
- 在一个拥有多个 return 语句的函数中使用 QMutexLocker(或者 QReadLocker，QWriteLocker)，以确保函数从任意可能的执行路径均可释放锁。
- 创建 QObject 的线程，也称线程关联(thread affinity)，负责执行那个 QObject 的槽。
- 如果各 QObject 具有不同的线程关联，那么就不能以父-子关系来连接它们。
- 通过从 run() 函数直接或者间接地调用 QThread::exec()，可以让线程进入事件循环。
- 利用 QApplication:: postEvent() 分发事件，或使用队列式的信号/槽连接，都是用于线程间通信的安全机制——但需要接收线程处于事件循环中。
- 确保每个跨线程连接的参数类型都用 qRegisterMetaType() 注册过。

17.2.1 线程安全和 QObject

可重入(reentrant)函数就是一个可以由多个线程同时调用的函数，其中任意的两次调用都不会试图访问相同的数据。线程安全的方法在任何时间都可以同时由多个线程调用，因为任何共享数据都会在某种程度上(例如，通过 QMutex)避免被同时访问。如果一个类的所有非静态函数都是可重入的或者是线程安全的，那么它就是可重入的或者是线程安全的。

一个 QObject 在它所“属于”或者有关联的线程中被创建。其各子对象也必须属于同一线程。Qt 禁止存在跨线程的父—子关系。

- QObject::thread() 可返回它的所有者线程，或者是其关联线程。
- QObject::moveToThread() 可将其移动到另一个线程。

moveToThread(this)

由于 QThread 是一个 QObject 而且在需要额外的线程时才会创建 QThread，因此，即使你会认为 QThread 和线程是可以相互指代的，也是可以理解的。尽管如此，那个额外的线程在调用 QThread::start() 之前实际上都不会被创建，这使得问题更难于理解。

回想一下，每个 QThread 的本质都是一个 QObject，这决定了它与其创建的线程存在关联，而不是与它启动的线程存在关联。

正是因为这个原因，有人说 QThread 并不是线程本身，而是该线程的管理器。这或许也可以有助于理解这一方式。实际上，QThread 是一个底层线程 API 的封装器，也是一个基于 java.lang.thread API 的管理单个线程的管理器。

这就意味着，当信号连接到这个 QThread 的槽上时，槽函数的执行是在其创建线程，而不是在其管理的线程进行的。

一些程序通过改变 QThread 的定义使它可表示其管理的线程并在该线程内执行它的槽。这些程序使用一种变通方法：在 QThread 的构造函数中使用 moveToThread(this)。这一变通方法的主要问题是，在线程退出后，通过 post 方式派发给该对象的事件如何处理留下不确定性。

有一篇文章[①]详细地讨论了这个问题。要说明的是，尽管 Qt 自身文档和一些旧示例程

① 参见http://labs.qt.nokia.com/2010/06/17/youre-doing-it-wrong/。

序都使用了这种做法，它仍是不再被推荐的。一方面，在托管的线程终止时，对于事件和信号会发生什么并不确定。另一方面，它混淆了 QRunnable 和 QThread 的目的，并将太多的职责加到单一的类中。

线程安全的对象就是一个可以由多个线程同时访问并且可确保处于“有效”状态的对象。默认情况下，QObject 不是线程安全的。为了让一个对象线程安全，可以利用的方法有许多种。这里列出了一些，但推荐去更深入地了解 Qt 线程支持方面的文档[①]。

1. QMutex 用于保证互斥，可与 QMutexLocker 一起使用，它允许一个单独的线程 T 保护(锁住)一个对象或者一段代码，使其在线程 T 释放(解锁)之前不能被其他的线程访问。
2. QWaitCondition 与 QMutex 结合使用，可以把某个线程置于一种不忙的阻塞状态，这种状态下，可让其等待另外一个线程将其唤醒。
3. QSemaphore 是一个广义的 QMutex，可以用在一个线程在开始工作之前需要锁住不止一个资源的各种情况下。信号量使其能够保证线程仅在要进行工作所需的资源全部满足的情况下才锁住资源。

有更多的 Qt 示例展示了如何使用 QtConcurrent： $QTDIR/examples/qtconcurrent。

volatile 的合理性

volatile 就像 const 一样，可由编译器用于为程序强制确保一定的线程安全。如果对象是 volatile 的，则只有标记为 volatile 的方法才可以调用它。volatile 在 Qt 中用于实现一些原子操作，这些操作反过来又用于实现诸如 QMutex，QSharedPointer 和 QReadWriteLock 这样的更高级结构中。有关 volatile 的更多信息，可参考下列两篇文章。

- *Using Volatile with User Defi ned Types*(在用户定义的类型中使用 volatile)[②]。
- *Volatile Almost Useless for Multithreaded Programming*(volatile 对多线程编程几乎没有什么用)[③]。

17.2.2 并行素数计算器

这一节介绍两种不同的计算素数的方法，其工作共享于多个线程。

第一种方法是生产者—消费者模型(producer-consumer model)，带有一个负责收集结果的调停对象(mediator object)。

示例 17.14 给出了一个生产者类 PrimeServer。

示例 17.14 src/threads/PrimeThreads/primeserver.h

```
[ . . . . ]
class PrimeServer : public QObject
{
    Q_OBJECT
public:
```

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

② 参见http://www.drdobbs.com/184403766。

③ 参见http://software.intel.com/en-us/blogs/2007/11/30/volatile-almost-useless-for-multi-threaded-programming/。

```
    explicit PrimeServer(QObject* parent =0);
    void doCalc(int numThreads, int highestPrime, bool concurrent = false);
    int nextNumberToCheck();
    void foundPrime(int );
    bool isRunning() const;
public slots:
    void cancel();
private slots:
    void handleThreadFinished();
signals:
    void results(QString);
private:
    int m_numThreads;
    bool m_isRunning;
    QList<int> m_primes;
    int m_nextNumber;
    int m_highestPrime;
    QTime m_timer;
    QMutex m_nextMutex;
    QMutex m_listMutex;
    QSet<QObject*> m_threads;
private slots:
    void handleWatcherFinished();
    void doConcurrent();
private:
    bool m_concurrent;
    int m_generateTime;                                          1
    QFutureWatcher<void> m_watcher;
};
[ . . . . ]
```

1 花在生成输入数据上的时间。

PrimeServer 创建 PrimeThreads(消费者)来执行实际的工作。示例 17.15 中创建并启动了各个 PrimeThreads 子对象。

示例 17.15　src/threads/PrimeThreads/primeserver.cpp

```
[ . . . . ]

void PrimeServer::
doCalc(int numThreads, int highestPrime, bool concurrent) {
    m_isRunning = true;
    m_numThreads = numThreads;
    m_concurrent = concurrent;
    m_highestPrime = highestPrime;
    m_primes.clear();
    m_primes << 2 << 3;
    m_threads.clear();
    m_nextNumber = 3;
    m_timer.start();
    if (!concurrent) {
        for (int i=0; i<m_numThreads; ++i) {
            PrimeThread *pt = new PrimeThread(this);             1
            connect (pt, SIGNAL(finished()), this,
                     SLOT(handleThreadFinished()));
```

```
            m_threads << pt;
            pt->start();                                    2
        }
    }
    else doConcurrent();
}
```

1 子线程还没有开始。

2 子线程执行 run()。

如示例 17.16 所示，PrimeThread 是一个重写了 run()的自定义 QThread。

示例 17.16 src/threads/PrimeThreads/primethread.h

```
#ifndef PRIMETHREAD_H
#define PRIMETHREAD_H

#include <QThread>
#include "primeserver.h"

class PrimeThread : public QThread
{
    Q_OBJECT
public:
    explicit PrimeThread(PrimeServer *parent);
    void run();                                             1
private:
    PrimeServer *m_server;

};

#endif // PRIMETHREAD_H
```

1 需要重写。

如示例 17.17 所示，在一个紧凑的循环中，run()函数在调用 PrimeServer 的两个带有 QMutexLocker 的方法中间进行一个素数测试。

示例 17.17 src/threads/PrimeThreads/primethread.cpp

```
[ . . . . ]
PrimeThread::PrimeThread(PrimeServer *parent)
: QThread(parent), m_server(parent) { }

void PrimeThread::run() {
    int numToCheck = m_server->nextNumberToCheck();
    while (numToCheck != -1) {
        if (isPrime(numToCheck))
            m_server->foundPrime(numToCheck);
        numToCheck = m_server->nextNumberToCheck();
    }
}
[ . . . . ]
```

如示例 17.18 所示，PrimeServer 使用 QMutexLocker 来锁住 QMutex，这使得程序进入和离开封闭的块作用范围时都可以安全地锁定和解锁 QMutex。最初，这个程序使用一个简

单的互斥量来保护这两种方法，但因为要访问的数据是独立的，为每种方法都使用独立的互斥量可以增加并行的可能性。

示例 17.18 src/threads/PrimeThreads/primeserver.cpp

```
[ . . . . ]

int PrimeServer::nextNumberToCheck() {
    QMutexLocker locker(&m_nextMutex);                    1
    if (m_nextNumber >= m_highestPrime) {
        return -1;
    }
    else {
        m_nextNumber+= 2;
        return m_nextNumber;
    }
}

void PrimeServer::foundPrime(int pn) {
    QMutexLocker locker(&m_listMutex);                    2
    m_primes << pn;
}
```

1 基于作用范围的互斥量工作于有多个返回点的情况。

2 这个方法也必须要线程安全。

这些方法都是线程安全的，因为从多个线程同时调用会相互阻塞。这是序列化访问临界共享数据的一种方式。

示例 17.19 src/threads/PrimeThreads/primeserver.cpp

```
[ . . . . ]

void PrimeServer::cancel() {
    QMutexLocker locker(&m_nextMutex);
    m_nextNumber = m_highestPrime+1;
```

示例 17.19 中给出的 `cancel` 方法打算以非阻塞的方式来得到调用，以便让一个 GUI 可以在 `PrimeThread` 安全退出其循环并从 `run()` 返回时持续对事件作出响应。

示例 17.20 中，服务器会清空每个完成的线程并在它们全部完成时报告其结果。它使用了 `QObject::sender()` 来获取信号发送者并用 `deleteLater()` 安全地将其删除。这是一种推荐用于终止和清理线程的安全方式。

示例 17.20 src/threads/PrimeThreads/primeserver.cpp

```
[ . . . . ]

void PrimeServer::handleThreadFinished() {
    QObject* pt = sender();                               1
    m_threads.remove(pt);
    pt->deleteLater();
    if (!m_threads.isEmpty()) return;                     2
    int numPrimes = m_primes.length();
    QString result = QString("%1 mutex'd threads %2 primes in %3"
```

```
                                "miliseconds. ").arg(m_numThreads)
                                .arg(numPrimes).arg( m_timer.elapsed());
    QString r2 = QString(" %1 kp/s")
                    .arg(numPrimes / m_timer.elapsed());
    qDebug() << result << r2;
    emit results(result + r2);
    m_isRunning = false;
}
```

1 QThread 是发送者。

2 其他仍在运行中。

图 17.6 中对测试 100 000 000 个号码的结果进行了总结。标记有 Mutex'd 的行给出了在 *n* 个工作者线程上运行生产者—消费者算法时的加速比因子(speedup factor)。正如所看到的，最佳加速比因子是在 3 时获得的，之后性能就降了下来。这可能是因为还有一个生产者线程相当繁忙而在图中没有计算进来。

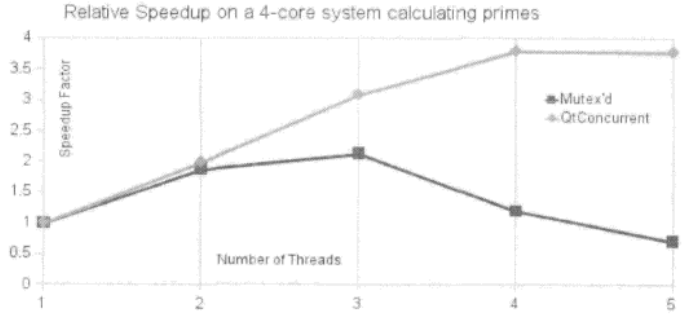

图 17.6 PrimeThreads 的加速比因子

标有 QtConcurrent 的另一行给出了几乎是最佳的加速比因子(1:1)，在 4 核系统上达到了 4，并在超过该值而继续增加线程数时并没有显著的退化。如示例 17.22 所示，这来自于对(前面)同一个 isPrime() 函数的并发使用。

这个曲线图主要关注在一个单独线程上运行相同算法时的相对加速因子的比较，但它的确给出的是绝对速度。如果尝试运行这个例子，会发现在所有测试中 QtConcurrent 方法总体至少会快 10 倍。

并行算法的效率取决于让所有核都忙着做计算，而不是等待对方的同步。相对越多的时间是花费在同步方面，随着线程增加得更多，算法的性能会变得越差。

QtConcurrent 方法

可以使用 QtConcurrent 的 filter()算法来从一系列的数中过滤非素数，而不是创建和管理自己的线程。QtConcurrent 算法可自动把工作分发给任意数量的线程，这可通过全局的 QThreadPool 给定。这些算法会接受一个容器和一个可作用于容器中每一项的函数指针或者仿函数。

示例17.21给出了在PrimeServer中并行算法用到的数据成员。可以使用QFutureWatcher以非阻塞的方式来等待计算的结束。

示例 17.21 src/threads/PrimeThreads/primeserver.h

```
[ . . . . ]

private slots:
    void handleWatcherFinished();
    void doConcurrent();
private:
    bool m_concurrent;
    int m_generateTime;                                          1
    QFutureWatcher<void> m_watcher;
};
```

1 用在生成输入数据上的时间。

示例 17.22 给出了一个函数式编程风格的解决方案。非阻塞的 `filtering()` 函数会立即返回一个 `QFuture` 类型的值。可以将其发送给 `QFutureWatcher` 来监测计算的进展情况。

示例 17.22 src/threads/PrimeThreads/primeserver.cpp

```
[ . . . . ]

void PrimeServer::doConcurrent() {
    QThreadPool::globalInstance()->setMaxThreadCount(m_numThreads);
    m_primes.clear();
    m_primes << 2;
    for (m_nextNumber=3; m_nextNumber<=m_highestPrime;
         m_nextNumber += 2) {
       m_primes << m_nextNumber;
    }
    m_generateTime = m_timer.elapsed();
    qDebug() << m_generateTime << "Generated "
             << m_primes.length() << " numbers";
    connect (&m_watcher, SIGNAL(finished()), this,
             SLOT(handleWatcherFinished()));
    m_watcher.setFuture(                                         1
        QtConcurrent::filter(m_primes, isPrime));                2
}

void PrimeServer::handleWatcherFinished() {
    int numPrimes = m_primes.length();
    int msecs = m_timer.elapsed();
    QString result =
      QString("%1 thread pool %2 primes in %4/%3 milliseconds"
              "(%5% in QtConcurrent).") .arg(m_numThreads)
              .arg(numPrimes).arg(msecs).arg(msecs-m_generateTime)
              .arg((100.0 * (msecs-m_generateTime)) / msecs);
    QString r2 = QString(" %1 kp/s").arg(numPrimes / msecs);
    qDebug() << result << r2;
    m_watcher.disconnect(this);
    emit results(result + r2);
    m_isRunning = false;
}
```

1 QFutureWatcher 用于检测进展情况。

2 非阻塞，此 filter()会返回一个 QFuture。

17.2.3 并发映射/规约示例

重新看一下 9.10 节中“康威的游戏人生”(Conway's Game of Life)示例，我们将使用 QtConcurrent 的 MapReduce()算法并行化这一计算。对于这项工作，必须先将问题分成较小的块。把每一块都定义成一个 LifeSlice，如示例 17.23 所示。

示例 17.23 src/threads/life/lifeslice.h

```
[ . . . . ]
struct LifeSlice {
    LifeSlice() {};
    LifeSlice(QRect r, QImage i) : rect(r), image(i) {}
    QRect rect;
    QImage image;
};
[ . . . . ]
```

一个 LifeSlice 由一个用来说明 QRect 是来自哪一块的 QImage 构成。这是映射函数的参数和返回类型(示例 17.24)。不要在 LifeSlice 中使用 QPixmap[①]。

示例 17.24 src/threads/life/lifemainwindow.cpp

```
[ . . . . ]
struct LifeFunctor : public std::unary_function<LifeSlice, LifeSlice> {
    LifeSlice operator() (LifeSlice slice);
};

LifeSlice LifeFunctor::operator()(LifeSlice slice) {                    1
    QRect rect = slice.rect;
    QImage image = slice.image;
    QImage next = QImage(rect.size(), QImage::Format_Mono);
    next.fill(DEAD);
    int h = rect.height();  int w = rect.width();

    for (int c=0; c<w; ++c) {
        for (int r=0; r<h; ++r) {
            int x = c+rect.x();
            int y = r+rect.y();
            bool isAlive = (image.pixelIndex(x, y) == ALIVE);
            int nc = neighborCount(image, x, y);
            if (!isAlive && nc == 3)
                 next.setPixel(c, r, ALIVE);
            if (!isAlive) continue;
            if (nc == 2 || nc == 3)
                next.setPixel(c,r, ALIVE);
        }
    }
    slice.image = next;
    return slice;
}
```

1 映射函数。

① 17.2 节中讨论过这一限制。通常情况下，在 GUI(主)线程之外使用 QPixmap 是不安全的。

LifeFunctor 派生自一个一元函数对象，来自标准库模板的 unary_function 会给仿函数附加额外的类型“特征”（traits），可被泛型算法用来获得仿函数的参数及返回类型。这个仿函数会定义一个带有 LifeSlice 参数并返回 LifeSlice 值的 operator()。

映射函数的返回类型必须是示例 17.25 中的规约函数的(第二个)参数类型。

示例 17.25 src/threads/life/lifemainwindow.cpp

```
[ . . . . ]
void stitchReduce(QImage& next, const LifeSlice &slice) {
    if (next.isNull())
        next = QImage(boardSize, QImage::Format_Mono);
    QPainter painter(&next);
    painter.drawImage(slice.rect.topLeft(), slice.image);          1
}
```

1 在一个图片上绘制另一个图片的一部分。

规约函数必须以某种方式共同带有每个工作者线程产生的部分结果并重新组合成一个连贯的 QImage。在这个拼图练习中，映射函数使用高层次的 QPainter API 来把一幅画绘制到另一幅画上，避免了单像素赋值对嵌套循环的需求。

示例 17.26 中的主循环必须将问题分成更小的问题，把它们发送到 QtConcurrent 的 blockingMappedReduced()，并且把结果发送到 LifeWidget。

示例 17.26 src/threads/life/lifemainwindow.cpp

```
[ . . . . ]

void LifeMainWindow::calculate() {
    int w = boardSize.width();
    // This might not be optimal, but it seems to work well...
    int segments = QThreadPool::globalInstance()->maxThreadCount() * 2;
    int ws = w/segments;                                                    1
    LifeFunctor functor;                                                    2
    while (m_running) {
        qApp->processEvents();                                              3
        m_numGenerations++;
        QList<LifeSlice> slices;                                            4
        for (int c=0; c<segments; ++c) {
            int tlx = c*ws;
            QRect rect(tlx, 0, ws, boardSize.height());
            LifeSlice slice(rect, m_current);
            slices << slice;                                                5
        }
        m_current = QtConcurrent::blockingMappedReduced(slices, functor,
                    stitchReduce, QtConcurrent::UnorderedReduce );          6
        m_lifeWidget->setImage(m_current);
    }
}
```

1 段的宽度。

2 映射仿函数。

3 确保 GUI 仍旧可以响应。

4 分成更小的块。

5 把小块添加到一个以并行方式处理的集合中。

6 开始并行工作。当其准备好后，可在每个块上都可以调用 stitchReduce。

这个循环中的 qApp->processEvents()是必须的，以便主事件循环来接收和处理其他 GUI 事件。如果注释掉这一行并试着运行应用程序，就会注意到，一旦计算开始就没有办法停止或退出。

在 1024 × 768 的版幅上，这个程序使用 4 个线程时可获得每秒 10 帧(frames per second，fps)，而用一个线程相对可获得 4 fps。这里并不是一个因子 4，但使用更大的幅面时，可能会观察到更好的改善。记住，在计算完每一代之后，都不可避免地有一个同步点。

17.3 练习：QThread 和 QtConcurrent

1. 在不使用 QtConcurrent 的情况下，编写一个多线程的游戏人生的例子，其中的 LifeServer 创建并管理 LifeWorker 线程(可多达 QThreadPool::maxThreadCount()个)，同时把计算分配到这些线程以获得更快的速度。让用户可以从 QSpinBox 中设置线程的数目。

   使用 QMutex 或 QReadWriteLock 对共享数据的访问进行同步。以示例 17.14 中 PrimeThreads 的生产者与消费者的例子为指南，并可复用 17.2.4 节中 Life 例子的任意代码。图 17.7 给出了一个用 UML 表示的可能的高层次设计。

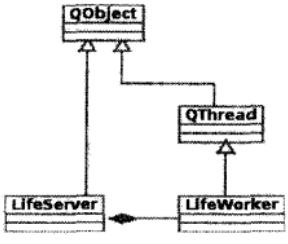

图 17.7 多线程的 Life UML

2. 从 17.2.2 节中给出的应用程序开始，在 GUI 中添加一个可使这两种算法工作的进度条。

3. 在这个练习中，重新回顾一下那些在 10.5.1 节中介绍过的图像处理技术并将其用于并行，通过多个线程来产生随机拼图。

   编写一个应用程序，让用户可从磁盘中选择尽可能多的图片，然后使用 QtConcurrent 算法在随机数量的图像副本上随机使用图像处理功能(如果适当，可以带一些随机参数)。

   在图像操作完成后，对每个处理过的图像使用 QtConcurrent 算法进行随机缩减，并将其绘制到拼图最初空白图像的随机位置。

   在拼图生成并保存到磁盘后，在屏幕上显示这个拼图。图 17.8 是使用 28 张照片产生的，其中的每个都被复制了 1 至 5 次，经操纵、缩放并插入到一个(默认的)640 × 480 图像中。最终拼图的大小可以用命令行参数设置。

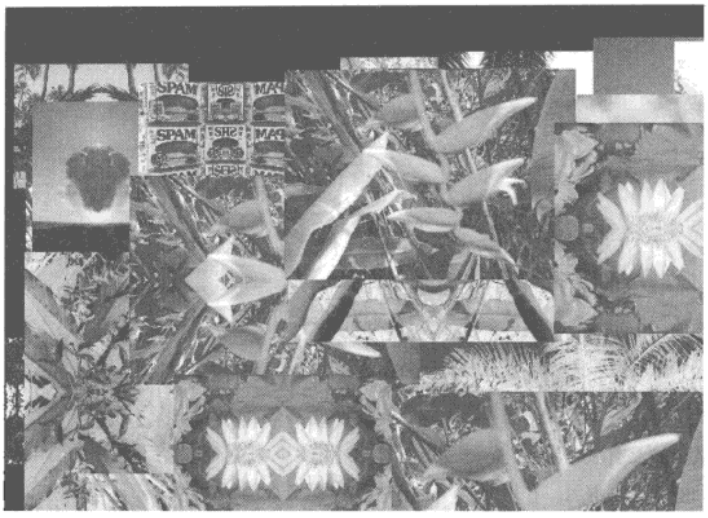

图 17.8　拼图示例

17.4　复习题

1. 列举并解释至少两种父进程可以用来向其子进程传递信息的机制。
2. 列举并解释至少两种线程之间彼此同步的机制。
3. 在什么情况下可以使用一个 `QTimer` 来代替一个 `QThread`？为什么要这么做？
4. 对于函数来说，线程安全意味着什么？
5. 哪个类可用于非 GUI 型线程？是 `QImage` 还是 `QPixmap`？
6. 对于函数来说，可重入(reentrant)意味着什么？
7. 怎样让一个非 GUI 线程进入事件循环？
8. 无须使用其他额外线程，在执行一个长期运行的循环同时，如何保持 GUI 的响应能力？