第15章 网 络

互联网的蓬勃发展从1995年开始，距今已有将近20年。2008年6月，iPhone 3G发布，开启移动互联网的新纪元，距今已有7年。网络像小雨一样，随风潜入夜，润物细无声，已经成为我们感知世界的神经元了。在Qt Quick的世界里，网络操作被封装在各个类库里，多数时候你什么都不用干，只要传递一个URL，那些个生具网络属性的对象就把你伺候得好好的。而如果你对需求的渴望像安踏一样永不止步，那还可以请出C++与QML混合编程的神器，利用Qt C++强悍的网络类库帮助你完成使命。

15.1 大珠小珠落玉盘——支持网络的对象

还记得5.11.1节介绍的url类型吗？

当我开始偷偷地想你——url，当你研究Qt Quick提供的类库时，如果发现某个元素的某个属性，帮助里标注类型为url，那它多半就可以处理网络链接，从网络加载它需要的资源。

15.1.1 Image

想必你已经想起我们的好朋友Image类了，它的source属性就是url类型的，可以接受HTTP协议的链接地址，显示网络图片。例如：

```
Image {
    source: "http://www.baidu.com/img/baidu_jgylogo3.gif";
}
```

类似的，还有AnimatedImage、AnimatedSprite等。

15.1.2 Qt.createComponent

全局对象Qt的createComponent（url, mode, parent）方法，第一个参数是url类型。这个方法可以根据一个URL创建一个组件，支持从网络加载。例如：

```
Qt.createComponent("http://xxx.yyy.com/YourComponent.qml");
```

15.1.3　Loader 对象

Loader 类的 source 属性从指定的 URL 加载组件，可以接受 HTTP 链接。

假设你的 text.qml 位于 http://127.0.0.1:8081/text.qml，remote.qml 可以这样：

```
import QtQuick 2.2
import QtQuick.Window 2.1
Window {
    id: root;
    visible: true;
    width: 300;
    height: 200;
    Loader {
        anchors.fill: parent;
        source: "http://127.0.0.1:8081/text.qml";
    }
}
```

15.1.4　QQmlApplicationEngine

当你基于 Qt SDK 5.3.1 使用 Qt Creator 3.1.2 创建 Qt Quick App 项目时，默认生成的 main.cpp 是这样子的：

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>

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

    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:///main.qml")));

    return app.exec();
}
```

如你所见，QQmlApplicationEngine 的 load()方法，其参数类型是 QUrl，能够接受网络地址，你修改成下面这样，也可以正常运行：

```
engine.load(QUrl("http://127.0.0.1:8081/main.qml"));
```

15.1.5　QQuickView

在 Qt SDK 5.2 中，新建项目向导生成的 Qt Quick App，使用 QQuickView 来加载应用的主 QML，4.1.2 节我们提供了一个 main.cpp 示例：

```
#include <QGuiApplication>
#include <QQuickView>

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

    QQuickView viewer;
    viewer.setResizeMode(QQuickView::SizeRootObjectToView);
    viewer.setSource(QUrl("qrc:///main.qml"));
    viewer.show();
```

```
    return app.exec();
}
```

QQuickView 的 setSource()方法的参数类型是 QUrl，你把 setSource()调用修改成下面的样子就可以从网络加载 QML 了：

```
viewer.setSource(QUrl("http://127.0.0.1:8081/main.qml"));
```

15.1.6 MediaPlayer

MediaPlayer 类的 source 属性用于指定待播放的媒体文件，支持 HTTP 等协议，我们在 14 章介绍过了，你可以这样用：

```
MediaPlayer {
    id: player;
    source: "http://192.168.3.3/music/123.mp3";
}
```

15.2 QML 里的 HTTP

我是 C++程序员，先学 Qt C++，后学 Qt Quick，在我执行网络请求的时候，总是优先使用 Qt C++里的网络类库。这仅仅是个人习惯，其实呢，不依赖 C++，QML 也能完成常见的 HTTP 操作。提供这种能力的，就是 XMLHttpRequest 类。

XMLHttpRequest 是与宿主环境相关的对象，现在多数浏览器都支持它，AJAX 也表示离不开它，但它还只有事实标准而没有正式标准。

XMLHttpRequest 类提供了对 HTTP 协议的访问支持，你能够用它完成 GET、POST、HEAD 等请求。XMLHttpRequest 可以同步或异步地返回 Web 服务器的响应，并且能够以文本或者一个 DOM 文档的形式返回内容。

不要被 XMLHttpRequest 傲娇的外表迷惑，以为它只接受 XML 文档，其实，它什么都能处理，txt、html、json、binary……它温柔、坚定、强悍无比。

15.2.1 XMLHttpRequest 类介绍

先说属性吧。

readyState

readyState 属性保存 HTTP 请求的状态。XMLHttpRequest 对象刚创建时，这个属性的值从 0 开始，直到接收到完整的 HTTP 响应，这个值增加到 4。

表 15-1 中列出了 readyState 所代表的 5 种状态的细节。

表 15-1 XMLHttpRequest 对象的状态

状 态	名 称	描 述
0	Uninitialized	初始化状态。XMLHttpRequest 对象已创建或已被 abort()方法重置
1	Open	open()方法已调用，但是 send()方法未调用，请求还没有被发送
2	Sent	send()方法已调用，HTTP 请求已发送到 Web 服务器。未接收到响应
3	Receiving	所有响应头部都已经接收到。响应体开始接收但未完成
4	Loaded	HTTP 响应已经接收完毕

readyState 的值不会递减，除非你调用 abort()或 open()方法终止了一个正在处理的请求。每次这个属性的值增加的时候，都会触发 onreadystatechange 回调（话说这个名字真够别扭的，完全不能和 Qt 优雅的命名风格相提并论……）。

responseText

这个属性保存到目前为止从服务器接收到的响应体（不包括头部）。没收到数据时是空字符串。

如果 readyState 小于 3，这个属性就是一个空字符串。当 readyState 为 3 时，这个属性返回目前已经接收的响应部分。如果 readyState 为 4，这个属性保存了完整的响应体（body）。

如果 HTTP 头部指定了响应体（body）的字符编码；数据就使用该编码；否则就假定文本内容使用 UTF-8 编码。

responseXML

XMLHttpRequest 会判断收到的数据是否为 XML 格式，如果是，则将其解析为一个 DOM 对象，保存到 responseXML 属性中；如果不是 XML 格式的数据或者在解析 XML 的过程中发生错误，responseXML 属性的值为 null。

status

由服务器返回的 HTTP 状态码，如 200 表示成功，而 404 表示“Not Found”错误。当 readyState 小于 3 时读取这个属性可能会导致异常。

statusText

HTTP 状态码对应的文本描述，比如状态码 200 对应“OK”，404 对应“Not Found”。你应当在 readyState 大于或等于 3 时读取它。

现在来了解 XMLHttpRequest 的方法。

onreadystatechange

这个蹩脚的家伙，咱先说它。它是个回调函数，每次 readyState 属性改变时都会调用它。当 readyState 为 3 时，它也可能被调用多次。

open()

open()方法用于初始化 HTTP 请求参数，但并不发送请求。其函数原型如下：

```
open(method, url, async, username, password)
```

method 参数是个字符串，指定 HTTP 方法，可以是 GET、POST、HEAD。

url 参数指定地址，字符串，形如“http://www.baidu.com:80/xxx.asp”。

async 是个布尔值，取 false 时请求是同步的，后续对 send()的调用将阻塞，直到响应完全接收。如果这个参数是 true 或省略，请求是异步的，此时你需要设置 onreadystatechange 回调来获知请求何时完成。

username 和 password 参数是可选的，URL 需要授权访问时你就得提供。

setRequestHeader()

HTTP 协议定义了很多头部，比如 User-Agent、Content-Type、Accept-Encoding 等，你可以在 open()之后、send()之前调用这个方法设置必要的头部。函数原型如下：

```
setRequestHeader(name, value)
```

name 指定要设置的头部名称，value 指定头部对应的值（不能有换行哦）。

send()

send(body)方法用于发送 HTTP 请求，open()和 setRequestHeader()两个方法组装 HTTP 头部，send()是真正干大事儿的，它连接服务器，发送组装好的头部和 body 参数指定的数据。

body 只有在 open()指定了 POST、PUT 方法时才有效，它可以是一个字符串或者一个 DOM 对象。当使用 GET 方法或者没有数据要发送时，应当传递 null 给它。

abort()

这个方法重置 XMLHttpRequest 对象的状态，readyState 归 0，并且取消所有未决的网络活动。例如，如果请求用了太长时间，用户是个急性子等不起，你就可以调用它。

getResponseHeader()

这个方法返回指定的 HTTP 头部的值，请在 readyState 大于等于 3 时调用它。函数原型如下：

```
getResponseHeader(name)
```

name 指定头部名字，是个字符串，不区分大小写。

要是指定的头部没收到（比如服务器没发送），这个方法返回一个空串。

getAllResponseHeaders()

这个方法返回原始的 HTTP 头部信息，头部作为单个的字符串返回，一行一个头部，每行用换行符“\r\n”隔开。请在 readyState 大于等于 3 时调用它，这句话说了多遍了，烦死了吧。

15.2.2 GET 小示例

写了一个简单的示例，xmlhttp_get.qml，代码如下：

```
import QtQuick 2.2
import QtQuick.Controls 1.2

Rectangle {
    id: root;
    width: 480;
    height: 300;
    color: "black";
    property var xmlhttp: null;
    function onResultReady(){
        if(xmlhttp.readyState == 4){
            result.append("Status Code: %1\n\n
                    Response Headers:".arg(xmlhttp.status));
            result.append("  Content-Type: %1".arg(
                 xmlhttp.getResponseHeader("Content-Type")));
            result.append("  Content-Length: %1".arg(
                    xmlhttp.getResponseHeader("Content-Length")));
            result.append("  Content-Encoding: %1".arg(
                    xmlhttp.getResponseHeader("Content-Encoding")));
            result.append("  Transfer-Encoding: %1".arg(xmlhttp.
                    getResponseHeader("Transfer-Encoding")));
            result.append("  Server: %1\n".arg(
                    xmlhttp.getResponseHeader("Server")));
```

```
        if(xmlhttp.responseXML != null){
            result.append("XML Contents:");
            var doc = xmlhttp.responseXML.documentElement;
            result.append("  root elementname: %1".arg(doc.nodeName));
        }
        xmlhttp.abort();
    }
}
function get(url){
    if(xmlhttp == null){
        xmlhttp = new XMLHttpRequest();
        xmlhttp.onreadystatechange = onResultReady;
    }
    if(xmlhttp.readyState == 0){
        result.remove(0, result.length);
        xmlhttp.open("GET", url, true);
        xmlhttp.setRequestHeader("Accept-Encoding","gzip,deflate,sdch");
        xmlhttp.setRequestHeader("User-Agent", "QML");
        xmlhttp.send(null);
    }
}

TextEdit {
    id: result;
    anchors.margins: 4;
    anchors.bottom: searchBox.top;
    anchors.top: parent.top;
    anchors.left: parent.left;
    anchors.right: parent.right;
    readOnly: true;
    color: "steelblue";
}

Text {
    id: searchClue;
    text: "URL:";
    font.pointSize: 11;
    verticalAlignment: Text.AlignVCenter;
    height: 30;
    anchors.left: parent.left;
    anchors.leftMargin: 4;
    anchors.verticalCenter: searchBox.verticalCenter;
    color: "lightgray";
}

Rectangle {
    id: searchBox;
    z: 2;
    border.width: 1;
    border.color: "gray";
    color: "black";
    anchors.bottom: parent.bottom;
    anchors.bottomMargin: 8;
    anchors.left: searchClue.right;
    anchors.right: parent.right;
    anchors.margins: 4;
    height: 40;

    TextInput {
        id: searchEdit;
        anchors.margins: 2;
        anchors.fill: parent;
        font.pointSize: 13;
        verticalAlignment: TextInput.AlignVCenter;
```

```
            color: "white";
            activeFocusOnTab: true;
            onAccepted: root.get(text);
        }
    }
}
```

我设计了一个编辑框，接收 URL，按回车键则调用 get()方法，根据得到的 URL 发出 HTTP 请求。

get()方法以异步方式调用 XMLHttpRequest 的 open()方法，设置了两个 HTTP 头部，在 onResultReady()回调中提取了服务器返回的一些头部信息显示在 TextEdit 中。如果 URL 代表的资源是 XML 格式，则输出 XML 文档的根节点的名字。

我使用 qmlscene 工具加载 xmlhttp_get.qml。

图 15-1 是访问 CSDN 博客首页拿到的信息。

图 15-1　使用 XMLHttpRequest 访问 CSDN

我在笔记本电脑上搭建了一个简单的 HTTP 服务器，放了一个 XML 文档，图 15-2 是访问这个文档时的情况。

图 15-2　使用 XMLHttpRequest 访问 XML 文档

15.2.3　POST 数据

使用 XMLHttpRequest 执行 POST 操作与 GET 类似，不同之处有两点：一是 open()的第一个参数需要指定 HTTP 方法为 POST；二是 send()方法的参数不为 null。

示例代码如下：

```
import QtQuick 2.2
import QtQuick.Window 2.1
import QtQuick.Controls 1.2

Window {
    id: root;
    visible: true;
    width: 300;
    height: 200;
    color: "black";

    property var xmlhttp: null;
    function onResultReady(){
        if(xmlhttp.readyState == 4){
            console.log("Status Code: ", xmlhttp.status);
            xmlhttp.abort();
        }
    }
    function post(){
        if(xmlhttp == null){
            xmlhttp = new XMLHttpRequest();
            xmlhttp.onreadystatechange = onResultReady;
        }
        if(xmlhttp.readyState ==0){
            xmlhttp.open("POST", "http://127.0.0.1:8081/t");
            xmlhttp.send("test post");
        }
    }

    Button {
        anchors.centerIn: parent;
        text: "Send";
        onClicked: root.post();
    }
}
```

15.3　使用 C++代码完成复杂的网络操作

如果你像我一样更倾向于使用 Qt C++的网络类库，或者你发现 XMLHttpRequest 不能满足你高冷的需求，那你可以重回 QNetworkAccessManager 柔软、温暖的怀抱。

我们以 HTTP 下载为例来说明如何使用 QNetworkAccessManager、QNetworkReply、QNetworkRequest、QUrl 这一组类库。先科普一下：

- QNetworkAccessManager 是中心，它提供了发送请求的各种 API，以及网络配置相关的方法。
- QNetworkRequest 代表一个网络请求，包括 URL、HTTP 访问用的 User-Agent、用户名、密码等都通过它设置。
- QNetworkReply 是 QNetworkAccessManager 根据你提交的 QNetworkRequest 构造的一

个应答对象，一旦请求发出，你就可以通过它获知网络操作过程中的各种状态。

- QUrl 是构造请求（QNetworkRequest 对象）时常用的类，它可以编解码 URL，提取 URL 中的主机、路径、文件、用户名、密码等信息。

使用 QNetworkAccessManager 进行 HTTP 编程的基本步骤如下：

① 根据 URL 生成 QUrl 对象，然后根据 QUrl 创建一个网络请求 QNetworkRequest，必要时可设置一些 header。

② 调用 QNetworkAccessManager 的 get()、post()等方法中的一个。

③ 使用 QNetworkAccessManager 返回的 QNetworkReply 实例来跟踪应答的各种状态反馈。

示例代码片段：

```
QString strUrl("http://www.baidu.com");
QUrl qurl(strUrl);
QNetworkRequest req(qurl);
m_reply = m_nam.get(req);
connect(m_reply,SIGNAL(error(QNetworkReply::NetworkError)),
       this, SLOT(onEror(QNetworkReply::NetworkError)));
connect(m_reply, SIGNAL(finished()),this, SLOT(onFinished()));
```

其中 m_reply、m_nam 的定义如下：

```
class ProbeVideoPrivate : public QObject
{
    Q_OBJECT
public:
    ...
protected slots:
    void onEror(QNetworkReply::NetworkError code);
    void onFinished();

private:
    ...
    QNetworkAccessManager m_nam;
    QNetworkReply *m_reply;
};
```

onFinished()槽的代码如下：

```
void ProbeVideoPrivate::onFinished()
{
    int status = m_reply->attribute(QNetworkRequest::
                 HttpStatusCodeAttribute).toInt();
    if(status != 200)
    {
        QByteArray data = m_reply->readAll();
        ...
    }
}
```

想把 C++代码中网络请求的结果导出到 QML 环境中，请参看第 11 章。完整的示例，请参考 13.5 节的实例“股票跟踪”和 13.8 节的实例“找图看”。

有时你可能还会基于套接字来开发某些网络功能，此时就只能使用低阶的 QTcpSocket 或 QUdpSocket 类了，必须要在 C++中完成相关模块。具体实例可以参考第 18 章的聊哈。