第 10 章 Ajax导引 10

Ajax是Adaptive Path的Jesse James Garrett提出的一个名词，他在解释XMLHttpRequest对象（大部分现代浏览器都提供了这个对象）所进行的异步的客户端到服务器端通信时，使用了这一名词。作为异步JavaScript与XML（Asynchronous JavaScript and XML）的缩写，它其实只是对创建动态Web应用程序所必备技术的一个统称。而且Ajax技术的个别组成部分并非不可替换，比如使用HTML来替代XML，就根本不会影响它的正确性。

在本章里，你将了解到组成整个 Ajax 过程的细节（围绕从浏览器发送请求到服务器这个中心）。我们讨论的内容从具体的请求到 JavaScript 的交互和完成工作的必要的数据操作。它包括：

- 分析不同类型的HTTP请求，决定以何种方式将数据对象发送到服务器最合适。
- 观察整个HTTP响应过程，尝试处理所有可能发生的错误，包括服务器的超时。
- 读取、遍历并操作来自服务器响应的结果数据。

这样你就能够完整地理解整个 Ajax 过程是如何运作的，应该如何实现它，从而能掌握它的各种应用场景——从常见的工作到完整的应用程序。在第 11 章～第 13 章里，你还能看到一系列运用 Ajax 技巧的实例分析。

10.1 使用 Ajax

创建一个简单的 Ajax 不需要很多代码，虽然这样的实现已能给你提供非常多的特性了。比如，在提交了表单之后，不需要用户请求一个完整的新页面，而是异步地进行提交，在提交完成后加载表单结果所期望的那一小部分内容即可。再比如，搜索可购买域名的过程通常麻烦又费时，每次需要查找一个新域名的时候都要往表单里输入请求和提交它，等待页面加载完成。而在 Ajax 的帮助下，你可以获得即时的输出，类似在线应用站点 Instant Domain Search（http://instantdomainsearch.com/），如图 10-1 所示。

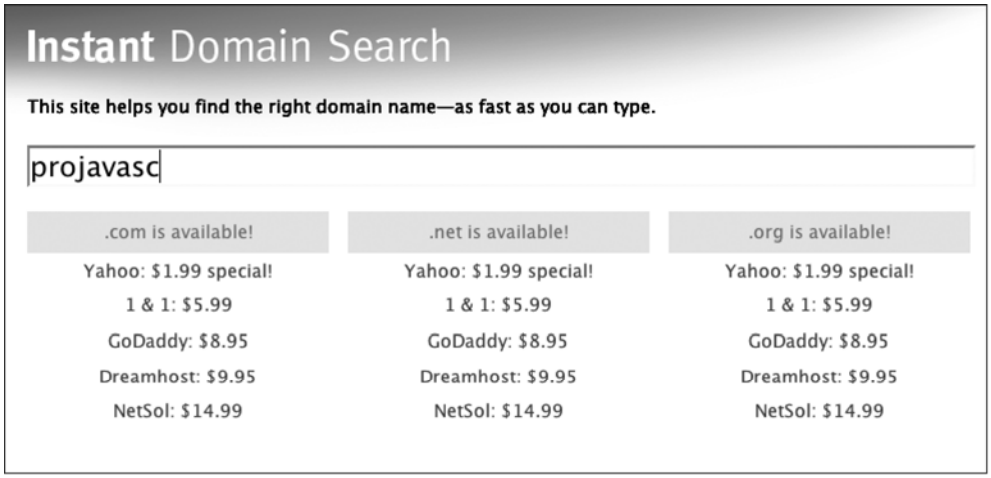

图10-1 Instant Domain Search随输入域名进行查询的一个例子

10.1.1 HTTP 请求

Ajax 中最重要也是最固定的部分是 HTTP 请求。HTTP（Hypertext Transfer Protocol，超文本传输协议）就是为了传输 HTML 文档和相关文件设计的一种协议。令人欣慰的是，所有的浏览器都支持一种使用 JavaScript 来动态建立 HTTP 连接的方式。这在开发更具丰富响应的 Web 应用程序中显得尤为有用。

归根结底，Ajax 的目的是异步地向服务器发送数据并接收数据。至于数据是什么格式，取决于你的具体文档，将在 10.2 节详细讨论。

在下面几个小节里，你将看到如何格式化数据，以便使用不同的 HTTP 请求传输到服务器上。你还能看到如何与服务器建立基本的连接，以及在跨浏览器环境中处理这些的必要细节。

1．建立连接

Ajax的关键在于创建到服务器的链接，实现这一目的有许多方法。不过这里只考虑一种发送和接受数据都很方便的方法，这种方法通常叫做“使用XMLHttpRequest对象”。

根据用户浏览器的不同，XMLHttpRequest对象的数据通信通常可用两种方法实现：

(1) IE是这种基于浏览器通信技术的创始者，使用ActiveXObject来创建连接（ActiveX对象的版本根据IE的版本而异）。幸运的是，IE 7 对XMLHttpRequest对象已经有了直接的支持。

(2) 其他所有的现代浏览器支持直接使用XMLHttpRequest对象。包括Firefox、Opera和Safari。 216

令人欣慰的是，尽管IE创建XMLHttpRequest对象的方法和其他现代浏览器不一样，但创建出来的对象还是支持同一套功能的。XMLHttpRequest对象有一系列用于建立连接、读取数据的方法。代码清单 10-1 就展示了如何向服务器发送一个基本的GET请求。

代码清单10-1 一种向服务器发送HTTP GET请求的跨浏览器方法

```
// 如果使用了IE，需要在XMLHttpRequest对象外包装一层
```

```
if ( typeof XMLHttpRequest == "undefined" )
    XMLHttpRequest = function(){
        // Internet Explorer使用ActiveXObject来创建新的
        // XMLHttpRequest对象
        return new ActiveXObject(
            // IE 5 使用的XMLHTTP对象和IE 6 不同
            navigator.userAgent.indexOf("MSIE 5") >= 0 ?
            "Microsoft.XMLHTTP" : "Msxml2.XMLHTTP"
        );
    };

//创建新的请求对象
var xml = new XMLHttpRequest();

//初始化请求①
xml.open("GET", "/some/url.cgi", true);

//与服务器建立连接并发送数据
xml.send();
```

如你所见，与服务器建立连接所需的代码非常简单。困难之处在于你需要高级特性（比如检查超时或修改过的数据）的时候，将在 10.1.2 节中介绍这些细节。

Ajax 方法中最重要的特性是允许数据在客户端（即 Web 浏览器）和服务器端间传输。鉴于这样的情况，让我们看看如何包装数据来发给服务器。

2．数据串行化

要发送一系列数据到服务器上，第一步就是整理它的格式，使服务器易于读取，这一过程称为“串行化（serialization）”。串行化有两种不同的情况，但都能满足多种不同的传输需求。

(1) 传输一个常规 JavaScript 对象，其中可能包含键/值的对（值可能是字符串也可能是数字）。

(2) 从一系列表单的输入栏中提交。这种情况和第一种不同之处在于提交的元素要按顺序排列，而第一种情况可以任意排序。

217

让我们看看一些例子，分别发送不同类型的数据，以及由它们转换得到的服务器友好的、串行的输出（如代码清单 10-2 所示）。

代码清单10-2 将原生JavaScript对象转换为串行形式的例子

```
// 一个包含“键/值”对的例子
{
    name: "John",
    last: "Resig",
    city: "Cambridge",
    zip: 02140
}

// 串行形式
```

① 原文为open the socket（打开套接字），这是错误的。考虑XHR对象完全不需要考虑到socket编程的细节，Microsoft、Apple和Mozilla的文档也无一提到过open和socket有任何关系，都只是说open用于初始化一个准备发起仍在“pending”状态中的请求。——译者注

```
name=John&last=Resig&city=Cambridge&zip=02140

// 另一组数据，包含许多值
[
    { name: "name", value: "John" },
    { name: "last", value: "Resig" },
    { name: "lang", value: "JavaScript" },
    { name: "lang", value: "Perl" },
    { name: "lang", value: "Java" }
]

// 这个数据的串行形式
name=John&last=Resig&lang=JavaScript&lang=Perl&lang=Java

// 最后，让我们找些表单输入元素（用我们在第5章创建的 id() 方法）
[
    id( "name" ),
    id( "last" ),
    id( "username" ),
    id( "password" )
]

// 将其串行化为字符串
name=John&last=Resig&username=jeresig&password=test
```

这种用于串行化上述数据的格式是发送 HTTP 请求的标准格式，你很可能曾见过这样的 HTTP GET 标准请求：

```
http://someurl.com/?name=John&last=Resig
```

218

这样的数据也可以通过 POST 请求（而且比 GET 允许传输的数据量要多得多）来传输，你将在接下来的一节看到它们的区别。

现在，让我们建立一个将代码清单10-2中的数据结构串行化的标准方法。在代码清单10-3可以看到这么一个函数，它能对大多数的表单元素进行串行化，不过多选按钮除外。

代码清单10-3 一个标准函数，将数据结构串行化为兼容HTTP的参数模式

```
// 串行化一系列数据。支持两种不同的对象：
//  - 表单输入元素的数组
//  - 键/值对的散列表
// 本函数返回串行化后的字符串
function serialize(a) {
    // 串行化结果的集合
    var s = [];

    // 若传入的参数是数组，假定它们是表单元素的数组
    if ( a.constructor == Array ) {

        // 串行化表单元素
        for ( var i = 0; i < a.length; i++ )
            s.push( a[i].name + "=" + encodeURIComponent( a[i].value ) );
```

```
        // 否则，假定这是一个键值对对象
        } else {

            // 串行化键值对
            for ( var j in a )
                s.push( j + "=" + encodeURIComponent( a[j] ) );

        }

        // 返回串行化结果
        return s.join("&");
    }
```

现在数据都转成了串行形式的字符串，接下来看看如何用 GET 或者 POST 请求发送数据到服务器上。

3．发送GET请求

让我们重新看看用XMLHttpRequest发送GET请求到服务器的方法，不过这次要连带发送串
219 行数据了。代码清单 10-4 是一个简单的例子。

代码清单10-4 一种跨浏览器的向服务器发送HTTP GET请求的方法（并且不读取任何结果数据）

```
// 创建请求对象
var xml = new XMLHttpRequest();

// 初始化异步 GET 请求
xml.open("GET", "/some/url.cgi?" + serialize( data ), true);

// 与服务器建立连接
xml.send();
```

要注意的是串行数据是附在服务器URL后面的，用?字符分隔。所有的Web服务器和应用程序框架都知道如何将?后面这部分数据解析为键值对。我们会在 10.2 节讨论如何处理响应，这些响应是服务器根据你提交的数据而返回的。

4．发送POST请求

使用XMLHttpRequest发送HTTP请求的另一种方式是POST，这是一种与GET截然不同的方式。POST支持发送任意格式、任意长度的数据，而不仅限于串行化字符串。

用于发送串行化格式数据的 MIME 类型（content type）通常是 application/x-www-form-urlencoded。这意味着你还能以 text/xml 或 application/xml 的形式给服务器直接发送 XML，甚至以 application/json 的形式发送 JavaScript 对象。

发送这种请求并传送附加串行数据的一个简单例子如代码清单 10-5 所示。

代码清单10-5 一种跨浏览器的向服务器发送HTTP POST请求的方法（并且不读取任何结果数据）

```
// 创建请求对象
var xml = new XMLHttpRequest();
```

```
// 初始化异步 POST 请求
xml.open("POST", "/some/url.cgi", true);

// 设置 content-type 首部，告知服务器如何解析我们发送的数据
xml.setRequestHeader(
    "Content-Type", "application/x-www-form-urlencoded");

// 保证浏览器发送的串行化数据长度正确 -
// 基于 Mozilla 的浏览器有时处理这个会碰到问题
if ( xml.overrideMimeType )
    xml.setRequestHeader("Connection", "close");

// 与服务器建立连接，并发送串行化数据
xml.send( serialize( data ) );
```

220

为了展开讨论先前的观点，我们来看一种不以“串行化”格式发送数据的情形（如代码清单10-6所示）。

代码清单10-6 将XML数据发送到服务器的例子

```
// 创建请求对象
var xml = new XMLHttpRequest();

// 初始化异步 POST 请求
xml.open("POST", "/some/url.cgi", true);

// 设置 content-type 首部，告知服务器如何解析我们发送的数据
xml.setRequestHeader( "Content-Type", "text/xml");

// 保证浏览器发送的串行化数据长度正确 -
// 基于 Mozilla 的浏览器有时处理这个会碰到问题
if ( xml.overrideMimeType )
    xml.setRequestHeader("Connection", "close");

// 与服务器建立连接，并发送串行化数据
xml.send( "<items><item id='one'/><item id='two'/></items>" );
```

这种发送大量数据的能力是非常重要的，它和 GET 请求依浏览器不同最多只能发几 KB 数据不同，而不用限制数据的长度。使用它你也可以实现许多不同通信协议，比如 XML-RPC 或 SOAP。

尽管如此，为简单起见，考虑 HTTP 响应时，不妨只限定在几种最常见，也最有用的数据格式的范围内。

10.1.2 HTTP 响应

创建、使用XMLHttpRequest要比其他简单的单向通信优越之处在于，它能够从服务器读取不同形式的文本数据。这包括Ajax的基石之一：XML。不过并没有规定必须用到XML才算一个Ajax应用程序，10.2 节介绍了其他的替代数据格式。

让我们先来看一个非常浅显的处理服务器响应数据的例子，如代码清单 10-7 所示。

221

代码清单10-7 与服务器建立连接并读取结果数据

```
// 创建请求对象
var xml = new XMLHttpRequest();

// 初始化异步 GET 请求
xml.open("GET", "/some/url.cgi", true);

// 在文档的状态更新时调用
xml.onreadystatechange = function(){
    // 等到数据完整加载
    if ( xml.readyState == 4 ) {

        // xml.responseXML 包含 XML 文档 (如果返回的是 XML)
        // xml.responseText 包含返回的文本
        // (如果返回的不是 XML)

        // 为避免内存泄漏，清理文档
        xml = null;
    }
};

// 建立到服务器的连接
xml.send();
```

在这个例子里，你能看到如何从HTTP响应中获取不同部分的数据。responseXML和responseText这两个属性将分别包含对应格式的数据。比如，如果服务器返回的是XML文档，那responseXML里存储的就是DOM文档，而其他所有的响应和结果都存放在responseText中。

在对响应数据进行实际的处理、遍历与操作之前，我们先创建一个更健壮的onreadystatechange函数（代码清单 10-7 里出现过的），用来处理服务器错误和连接超时。

1. 处理错误

如果XMLHttpRequest对象有内建服务器错误处理机制的话，会大大节省我们的时间，不幸的是它没有。不过只要花点工夫，就可以创建自己的错误处理机制。你需要检查的是以下这些请求状况，以判断服务器在处理请求时是否遇到了问题：

- *成功响应代码*：可以通过HTTP规范里定义的响应状态码来检查错误，通过读取状态码，客户端能够知道服务器的情形。状态码在200到300之间的属于成功的请求。
- *未修改响应*：服务器返回的文档可能会加上“Not Modified（未修改）”的标记，也就是状态码304。说明服务器返回的数据和浏览器的缓存内容一致，并未修改过。这其实不算是个错误，因为客户端仍能读出正确的数据。
- *本地存储的文件*：如果你在本机上直接执行Ajax应用程序，而不通过Web服务器，就算请求成功了，也不会得到任何返回的状态码。这意味着，在执行本地文件且得不到状态码时，你应该把这种情况算做成功的响应。
- Safari与未修改状态：如果文档自上次请求（或者通过浏览器明确地发送一个IF-MODIFIED-SINCE首部，指定上次修改过的时间给服务器）未曾修改过，Safari返回的状态码会是"undefined"。这是一个比较怪异的情形，也让人难于调试。

考虑了上述这些情形，我们可以看看代码清单 10-8，实现了刚才概述的响应检查。

代码清单10-8 用于检查服务器HTTP响应的成功状态（Success State）的一个函数

```
// 检查 XMLHttpRequest 对象是否有 'Success' 状态。
// 此函数需要一个 XMLHttpRequest 对象作为参数。
function httpSuccess(r) {
    try {
        // 如果得不到服务器状态，且我们正在请求本地文件，认为成功
        return !r.status && location.protocol == "file:" ||

            // 所有 200 到 300 间的状态码表示成功
            ( r.status >= 200 && r.status < 300 ) ||

            // 文档未修改也算成功
            r.status == 304 ||

            // Safari 在文档未修改时返回空状态
            navigator.userAgent.indexOf("Safari") >= 0 &&
                typeof r.status == "undefined";
    } catch(e){}

    // 若检查状态失败，就假定请求是失败的
    return false;
}
```

检查 HTTP 响应的成功状态是非常重要的一步，不作检查可能会导致许多难以预料的结果，比如服务器返回的其实是 HTML 错误页面，而非 XML 文档。

我们将把这个函数集成在 10.3 节给出的完整 Ajax 方案中。

2．检查超时

在XMLHttpRequest的默认实现中缺乏的另一个有用功能是如何判断请求超时。 223

这一功能的实现并不那么直接，但花点工夫（像上一节那样）判断请求的成功状态还是可行的。代码清单 10-9 展示了如何检查自己程序中的请求超时。

代码清单10-9 检查请求超时的一个例子

```
// 创建请求对象
var xml = new XMLHttpRequest();

// 初始化异步 GET 请求
xml.open("GET", "/some/url.cgi", true);

// 我们在请求后等 5 秒，然后放弃
var timeoutLength = 5000;

// 记录请求是否成功完成
var requestDone = false;

// 初始化一个 5 秒后执行的回调函数，用于取消请求（如果尚未完成的话）。
setTimeout(function(){
    requestDone = true;
```

```
}, timeoutLength);

// 监听文档状态的更新
xml.onreadystatechange = function()
{
    // 保持等待，直到数据完全加载，并保证请求并未超时
    if ( xml.readyState == 4 && !requestDone ) {

        // xml.responseXML 包含 XML 文档 (如果返回的是 XML)
        // xml.responseText 包含返回的文本
        // (如果返回的不是 XML)

        // 为免内存泄漏，清理文档
        xml = null;
    }
};

// 与服务器建立连接
xml.send();
```

在考虑了服务器通信的细节，并处理了许多可能的出错后，现在可以来看看如何处理服务器的响应数据了。

224

10.2 处理响应数据

迄今为止的所有例子中，你都只用一个符号来代替服务器的响应数据，理由很简单——服务器返回的数据格式有无限种可能性。不过现实中XMLHttpRequest只处理基于文本的数据格式。而且它在处理某些格式（XML）上要胜过另一些（JSON）。在本章里，你将看到服务器可能会返回的 3 种数据格式，并用客户端来读取并操作它们：

- XML：幸运的是，所有的现代浏览器都提供了原生的XML文档处理支持，自动将它们转换为可用的DOM文档
- HTML：和XML文档的区别在于，它通常以纯文本字符串的形式存在，存放一个HTML片段。
- JavaScript/JSON：这包括两种格式——原始的可执行JavaScript代码和JSON（JavaScript Object Notation，JavaScript对象表示）格式。

这 3 种格式都各有其适合的用途。比如有时返回 HTML 要比返回 XML 更有意义。

获取HTTP响应数据的重点是XMLHttpRequest对象的两个属性：

- responseXML：如果服务器返回的是XML文档，这个属性包含到预处理后DOM文档的引用，它是XML文档的表达。只在服务器明确指定其内容首部（content header）是"Content-type:text/xml"或类似的XML数据类型时，这一点才起作用。
- responseText：这一属性包含到服务器返回的原始文本数据的引用。HTML和JavaScript类型的数据都依赖这一方法来获得。

通过这两个属性，开发从 HTTP 响应中确定性地获取数据的通用函数就很简单了（甚或判断正在处理的数据是 XML 还是纯文本）。代码清单 10-10 展示的就是这样一个函数。

代码清单10-10 从HTTP服务器响应中解析正确数据的一个函数

```
// 从 HTTP 响应中解析数据的函数
// 有两个参数：一个 XMLHttpRequest 对象和一个可选参数——期望从服务器得到的数据类型
// 正确的值包括：xml, script, text, 或 html - 默认是 ""，根据 content-type 的首部得到
function httpData(r, type) {
    // 获取 content-type 首部
    var ct = r.getResponseHeader("content-type");

    // 若没有提供默认的类型，判断服务器返回的是否是 XML 形式
    var data = !type && ct && ct.indexOf("xml") >= 0;

    // 若是，获得 XML 文档对象，否则返回文本内容
    data = type == "xml" || data ? r.responseXML : r.responseText;

    // 若指定类型是 "script"，则以 JavaScript 形式执行返回文本
    if ( type == "script" )
        eval.call( window, data );

    // 返回响应数据（或为 XML 文档或为文本字符串）
    return data;
}
```

225

随着这个数据解析函数的完成，你现在拥有构建一套完整 Ajax 函数的所有组件了，可以实现常用的 Ajax 调用。我们在下一节能看到这样一个函数的完整实现。

10.3 完整的 Ajax 程序包

运用迄今为止学到的所有概念，你就可以构建出一个处理所有 Ajax 请求和相关响应的通用函数了。这个函数可以作为后续章节里你所有 Ajax 开发的基础，让你能迅速地从服务器查获信息。

这个完整的 Ajax 函数如代码清单 10-11 所示。

代码清单10-11 能够执行必要的Ajax相关任务的一个完整函数

```
// 执行 Ajax 请求的通用函数
// 带一个参数，是包含一系列选项的对象，这些选项在下面的注释中简述
function ajax( options ) {

    // 如果用户没有提供某个选项的值，就用默认值替代
    options = {
        // HTTP 请求的类型
        type: options.type || "POST",
        // 请求的 URL
        url: options.url || "",

        // 请求超时的时间
        timeout: options.timeout || 5000,
        // 请求失败、成功或完成 （不管成功还是失败都会调用的） 时执行的函数
        onComplete: options.onComplete || function(){},
        onError: options.onError || function(){},
        onSuccess: options.onSuccess || function(){},
```

226

```
    // 服务器将会返回的数据类型，这一默认值用于判断服务器返回的数据
    // 并作相应动作
    data: options.data || ""
};

// 创建请求对象
var xml = new XMLHttpRequest();

// 初始化异步请求
xml.open(options.type, options.url, true);

// 我们在请求后等待 5 秒，超时则放弃
var timeoutLength = options.timeout;

// 记录请求是否成功完成
var requestDone = false;

// 初始化一个 5 秒后执行的回调函数，用于取消请求（如果尚未完成的话）。
setTimeout(function(){
    requestDone = true;
}, timeoutLength);

// 监听文档状态的更新
xml.onreadystatechange = function(){
    // 保持等待，直到数据完全加载，并保证请求并未超时
    if ( xml.readyState == 4 && !requestDone ) {

        // 检查是否请求成功
        if ( httpSuccess( xml ) ) {

            // 以服务器返回的数据作为参数调用成功回调函数
            options.onSuccess( httpData( xml, options.type ) );

        // 否则就发生了错误，执行错误回调函数
        } else {
            options.onError();
        }

        // 调用完成回调函数
        options.onComplete();

        // 为避免内存泄漏，清理文档
        xml = null;
    }
};

// 建立与服务器的连接
xml.send();

// 判断 HTTP 响应是否成功
function httpSuccess(r) {
try {
    // 如果得不到服务器状态，且我们正在请求本地文件，认为成功
    return !r.status && location.protocol == "file:" ||
```

```
            // 所有 200 到 300 间的状态码表示成功
            ( r.status >= 200 && r.status < 300 ) ||

            // 文档未修改也算成功
            r.status == 304 ||

            // Safari 在文档未修改时返回空状态
            navigator.userAgent.indexOf("Safari") >= 0
                && typeof r.status == "undefined";
    } catch(e){}

        // 若检查状态失败，就假定请求是失败的
        return false;
    }

    // 从 HTTP 响应中解析正确数据
    function httpData(r,type) {
        // 获取 content-type 的首部
        var ct = r.getResponseHeader("content-type");

        // 若没有提供默认的类型，判断服务器返回的是否是 XML 形式
        var data = !type && ct && ct.indexOf("xml") >= 0;

        // 若是，获得 XML 文档对象，否则返回文本内容
        data = type == "xml" || data ? r.responseXML : r.responseText;

        // 若指定类型是 "script"，则以 JavaScript 形式执行返回文本
        if ( type == "script" )
            eval.call( window, data );

        // 返回响应数据 （或为 XML 文档或为文本字符串）
        return data;
    }
}
```

228

需要注意的是，请求与本页面在不同域下的数据是不可能的，因为所有的现代浏览器都有安全限制（避免别人试图获得你的个人信息）。现在你已经拥有了这个强大的函数，是时候通过一些例子试试你新开发的 Ajax 函数的能力了。

10.4 数据的不同用途

从根本上说，每次进行的简单 Ajax 请求之间并没有什么差别，不一样的是服务器返回的数据。根据希望实现的目的选择不同的数据格式完成任务会很有帮助。这也是接下来我将展示给你如何根据几种不同数据格式执行一些常见任务的原因。

10.4.1 基于 XML 的 RSS Feed

目前服务器返回的数据格式中最受欢迎的是XML，这不是毫无理由的。所有现代的浏览器都对XML文档有直接支持，能即时将它们转换为DOM的表达形式。因为服务器把解析的重任完

成了，你所需要做的只是像遍历其他DOM文档一样的遍历它。不过需要注意的是，通常无法直接用getElementById函数来遍历远程返回的XML文档，这是因为浏览器对非HTML的XML文档没有预备好的唯一ID属性选择器，所以也就无法仅通过ID来选择纯XML元素。尽管如此，还是有办法在XML文档中做有效的遍历。

代码清单 10-12 展示了一个用返回的 XML 来给你的网站创建动态 RSS Feed 控件的例子。

代码清单10-12 从基于XML的远程RSS Feed加载项目的标题

```
<html>
<head>
    <title>Dynamic RSS Feed Widget</title>
    <!— 载入我们的通用 Ajax 函数 -->
    <script src="ajax.js"></script>
    <script>
        // 等待文档完整加载
        window.onload = function(){
        // 然后使用 Ajax 载入 RSS feed
        ajax({
            // RSS feed 的 URL
            url: "rss.xml",

            // 这是一个 XML 文档
            type: "xml",

            // 此函数会在请求结束后执行
            onSuccess: function( rss ) {
                // 我们将把所有 RSS 项目的标题都插入到 id 为 "feed" 的 <ol> 中
                var feed = document.getElementById("feed");

                // 获取 RSS XML 文档中所有的标题
                var titles = rss.getElementsByTagName("title");

                // 遍历每个匹配的标题
                for ( var i = 0; i < titles.length; i++ ) {
                    // 创建一个 <li> 元素来存放标题
                    var li = document.createElement("li");

                    // 将其内容设为项目标题
                    li.innerHTML = titles[i].firstChild.nodeValue;

                    // 将其添加到 DOM 的 <ol> 元素中
                    feed.appendChild( li );
                }
            }
        });
    };
    </script>
</head>
<body>
    <h1>Dynamic RSS Feed Widget</h1>
    <p>Check out my RSS feed:</p>
    <!--这里将插入 RSS feed -->
```

```
    <ol id="feed"></ol>
</body>
</html>
```

你可以看到，一旦把 Ajax 请求/响应过程的复杂性解决了，剩下的其实并不复杂。此外，考虑到浏览器使 XML 文档的遍历非常方便，XML 确实是一种从服务器端快速传输数据到客户端的好方法。

230

10.4.2 HTML 注入器

可以用 Ajax 实现的另一个有用的技巧是，动态地将 HTML 片段载入到文档中。这个技巧和前面讨论的 XML 文档的方法区别在于，你不必解析或遍历从服务器收到的数据，就可以把它马上插入到文档中。这种快刀斩乱麻的方法能让你的页面更新得简便而快捷。它的一个例子如代码清单 10-13 所示。

代码清单10-13 从远程文件中载入一段HTML代码，注入到当前网页

```
<html>
  <head>
      <title>HTML Sports Scores Loaded via Ajax</title>
      <!-- 载入我们的通用 Ajax 函数 -->
      <script src="ajax.js"></script>
      <script>
          // 等待文档完全加载
          window.onload = function(){

              // 然后使用  Ajax 载入 RSS feed
              ajax({
                  // HTML sports score 的 URL
                  url: "scores.html",

                  // 是 HTML 文档
                  type: "html",

                  // 此函数会在请求结束后执行
                  onSuccess: function( html ) {
                      // 我们将插入到 id 为 'scores' 的 div 中
                      var scores = document.getElementById("scores");

                      // 将新的 HTML 注入文档
                      scores.innerHTML = html;
                  }
              });
          };
      </script>
  </head>
  <body>
      <h1> HTML Sports Scores Loaded via Ajax </h1>
       <!--这里插入 Sports Score -->
      <div id="scores"></div>
  </body>
```

231

```
</html>
```

对这种动态 HTML 技巧来说，最重要的一点在于，你仍然可以在服务器端代码里使用所有应用程序级别的模板工具，这使得模板代码集中也易于维护。

尽管这很简单，但也不要小看这种载入 HTML 文件的功能，它是创建更具用户响应 Web 应用程序的最便捷的方法。

10.4.3 JSON 与 JavaScript：远程执行

最后（在第 13 章 wiki 的例子中将会遇到）我们要讨论的数据格式是从 JSON 数据串到纯 JavaScript 代码的转换。对 JSON 化数据的转换可以作为从服务器到客户端传输 XML 文档的一种轻量级替代。此外，从服务器提供纯 JavaScript 代码则是一个构建动态多用户 Web 应用程序的好方法。为了简化，让我们看看如何将一个远程 JavaScript 文件载入你的应用程序中，如代码清单 10-14 所示。

代码清单10-14 动态载入并执行一个远程JavaScript文件

```
<html>
<head>
    <!-- 载入我们的通用 Ajax 函数 -->
    <script src="ajax.js"></script>
    <script>
        // 载入一个远程 Javascript 文档
        ajax({
            // JavaScript 文件的 URL
            url: "myscript.js",

            // 强制以 JavaScript 的形式执行
            type: "script"
        });
    </script>
</head>
<body></body>
</html>
```

10.5 小结

尽管看似简单，Ajax Web 应用程序的概念却是非常强大的。通过动态地把其他信息载入到基于 JavaScript 的运行中应用程序里，你可以创建响应更为丰富的程序界面。

在本章里，你了解到 Ajax 的基本概念，包括 HTTP 请求和响应的规范、错误处理、数据格式化与解析。我们同时得到了一个可供重用的通用函数，用它可以很简单地给任何 Web 应用程序赋予动态特性。你会在接下来的 3 章里使用这个函数来构建一系列的动态 Ajax 交互。

232