

第 21 章

Ajax 与 Comet

本章内容

- 使用 XMLHttpRequest 对象
- 使用 XMLHttpRequest 事件
- 跨域 Ajax 通信的限制

2005 年, Jesse James Garrett 发表了一篇在线文章, 题为 “Ajax: A new Approach to Web Applications” (<http://www.adaptivepath.com/ideas/essays/archives/000385.php>)。他在这篇文章里介绍了一种技术, 用他的话说, 就叫 Ajax, 是对 Asynchronous JavaScript + XML 的简写。这一技术能够向服务器请求额外的数据而无须卸载页面, 会带来更好的用户体验。Garrett 还解释了怎样使用这一技术改变自从 Web 诞生以来就一直沿用的 “单击, 等待” 的交互模式。

Ajax 技术的核心是 XMLHttpRequest 对象 (简称 XHR), 这是由微软首先引入的一个特性, 其他浏览器提供商后来都提供了相同的实现。在 XHR 出现之前, Ajax 式的通信必须借助一些 hack 手段来实现, 大多数是使用隐藏的框架或内嵌框架。XHR 为向服务器发送请求和解析服务器响应提供了流畅的接口。能够以异步方式从服务器取得更多信息, 意味着用户单击后, 可以不必刷新页面也能取得新数据。也就是说, 可以使用 XHR 对象取得新数据, 然后再通过 DOM 将新数据插入到页面中。另外, 虽然名字中包含 XML 的成分, 但 Ajax 通信与数据格式无关; 这种技术就是无须刷新页面即可从服务器取得数据, 但不一定是 XML 数据。

实际上, Garrett 提到的这种技术已经存在很长时间了。在 Garrett 撰写那篇文章之前, 人们通常将这种技术叫做远程脚本 (remote scripting), 而且早在 1998 年就有人采用不同的手段实现了这种浏览器与服务器的通信。再往前推, JavaScript 需要通过 Java applet 或 Flash 电影等中间层向服务器发送请求。而 XHR 则将浏览器原生的通信能力提供给了开发人员, 简化了实现同样操作的任务。

在重命名为 Ajax 之后, 大约是 2005 年底 2006 年初, 这种浏览器与服务器的通信技术可谓红极一时。人们对 JavaScript 和 Web 的全新认识, 催生了很多使用原有特性的新技术和新模式。就目前来说, 熟练使用 XHR 对象已经成为所有 Web 开发人员必须掌握的一种技能。

21.1 XMLHttpRequest 对象

IE5 是第一款引入 XHR 对象的浏览器。在 IE5 中, XHR 对象是通过 MSXML 库中的一个 ActiveX 对象实现的。因此, 在 IE 中可能会遇到三种不同版本的 XHR 对象, 即 MSXML2.XMLHttp、MSXML2.XMLHttp.3.0 和 MSXML2.XMLHttp.6.0。要使用 MSXML 库中的 XHR 对象, 需要像第 18 章讨论创建 XML 文档时一样, 编写一个函数, 例如:



```
//适用于 IE7 之前的版本
function createXHR(){
    if (typeof arguments.callee.activeXString != "string"){
        var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
            "MSXML2.XMLHttp"],
            i, len;

        for (i=0,len=versions.length; i < len; i++){
            try {
                new ActiveXObject(versions[i]);
                arguments.callee.activeXString = versions[i];
                break;
            } catch (ex){
                //跳过
            }
        }

        return new ActiveXObject(arguments.callee.activeXString);
    }
}
```

这个函数会尽力根据 IE 中可用的 MSXML 库的情况创建最新版本的 XHR 对象。

IE7+、Firefox、Opera、Chrome 和 Safari 都支持原生的 XHR 对象，在这些浏览器中创建 XHR 对象要像下面这样使用 XMLHttpRequest 构造函数。

```
var xhr = new XMLHttpRequest();
```

假如你只想支持 IE7 及更高版本，那么大可丢掉前面定义的那个函数，而只用原生的 XHR 实现。但是，如果你必须还要支持 IE 的早期版本，那么则可以在这个 createXHR() 函数中加入对原生 XHR 对象的支持。



```
function createXHR(){
    if (typeof XMLHttpRequest != "undefined"){
        return new XMLHttpRequest();
    } else if (typeof ActiveXObject != "undefined"){
        if (typeof arguments.callee.activeXString != "string"){
            var versions = [ "MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
                "MSXML2.XMLHttp"],
                i, len;

            for (i=0,len=versions.length; i < len; i++){
                try {
                    new ActiveXObject(versions[i]);
                    arguments.callee.activeXString = versions[i];
                    break;
                } catch (ex){
                    //跳过
                }
            }

            return new ActiveXObject(arguments.callee.activeXString);
        } else {
            throw new Error("No XHR object available.");
        }
    }
}
```

这个函数中新增的代码首先检测原生 XHR 对象是否存在，如果存在则返回它的新实例。如果原生对象不存在，则检测 ActiveX 对象。如果这两种对象都不存在，就抛出一个错误。然后，就可以使用下面的代码在所有浏览器中创建 XHR 对象了。

```
var xhr = createXHR();
```

由于其他浏览器中对 XHR 的实现与 IE 最早的实现是兼容的，因此就可以在所有浏览器中都以相同方式使用上面创建的 xhr 对象。

21.1.1 XHR 的用法

在使用 XHR 对象时，要调用的第一个方法是 open()，它接受 3 个参数：要发送的请求的类型（"get"、"post"等）、请求的 URL 和表示是否异步发送请求的布尔值。下面就是调用这个方法的例子。

```
xhr.open("get", "example.php", false);
```

这行代码会启动一个针对 example.php 的 GET 请求。有关这行代码，需要说明两点：一是 URL 相对于执行代码的当前页面（当然也可以使用绝对路径）；二是调用 open() 方法并不会真正发送请求，而只是启动一个请求以备发送。



只能向同一个域中使用相同端口和协议的 URL 发送请求。如果 URL 与启动请求的页面有任何差别，都会引发安全错误。

要发送特定的请求，必须像下面这样调用 send() 方法：

```
xhr.open("get", "example.txt", false);  
xhr.send(null);
```



[XHRExample01.htm](#)

这里的 send() 方法接收一个参数，即要作为请求主体发送的数据。如果不需要通过请求主体发送数据，则必须传入 null，因为这个参数对有些浏览器来说是必需的。调用 send() 之后，请求就会被分派到服务器。

由于这次请求是同步的，JavaScript 代码会等到服务器响应之后再继续执行。在收到响应后，响应的数据会自动填充 XHR 对象的属性，相关的属性简介如下。

- responseText：作为响应主体被返回的文本。
- responseXML：如果响应的内容类型是"text/xml"或"application/xml"，这个属性中将保存包含着响应数据的 XML DOM 文档。
- status：响应的 HTTP 状态。
- statusText：HTTP 状态的说明。

在接收到响应后，第一步是检查 status 属性，以确定响应已经成功返回。一般来说，可以将 HTTP 状态代码为 200 作为成功的标志。此时，responseText 属性的内容已经就绪，而且在内容类型正确的情况下，responseXML 也应该能够访问了。此外，状态代码为 304 表示请求的资源并没有被修改，可以直接使用浏览器中缓存的版本；当然，也意味着响应是有效的。为确保接收到适当的响应，应该像下面这样检查上述这两种状态代码：

```

xhr.open("get", "example.txt", false);
xhr.send(null);

if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
    alert(xhr.responseText);
} else {
    alert("Request was unsuccessful: " + xhr.status);
}

```

XHRExample01.htm

根据返回的状态代码，这个例子可能会显示由服务器返回的内容，也可能会显示一条错误消息。我们建议读者要通过检测 status 来决定下一步的操作，不要依赖 statusText，因为后者在跨浏览器使用时不太可靠。另外，无论内容类型是什么，响应主体的内容都会保存到 responseText 属性中；而对于非 XML 数据而言，responseXML 属性的值将为 null。



有的浏览器会错误地报告 204 状态代码。IE 中 XHR 的 ActiveX 版本会将 204 设置为 1223，而 IE 中原生的 XHR 则会将 204 规范化为 200。Opera 会在取得 204 时报告 status 的值为 0。

像前面这样发送同步请求当然没有问题，但多数情况下，我们还是要发送异步请求，才能让 JavaScript 继续执行而不必等待响应。此时，可以检测 XHR 对象的 readyState 属性，该属性表示请求/响应过程的当前活动阶段。这个属性可取的值如下。

- 0：未初始化。尚未调用 open() 方法。
- 1：启动。已经调用 open() 方法，但尚未调用 send() 方法。
- 2：发送。已经调用 send() 方法，但尚未接收到响应。
- 3：接收。已经接收到部分响应数据。
- 4：完成。已经接收到全部响应数据，而且已经可以在客户端使用了。

只要 readyState 属性的值由一个值变成另一个值，都会触发一次 readystatechange 事件。可以利用这个事件来检测每次状态变化后 readyState 的值。通常，我们只对 readyState 值为 4 的阶段感兴趣，因为这时所有数据都已经就绪。不过，必须在调用 open() 之前指定 onreadystatechange 事件处理程序才能确保跨浏览器兼容性。下面来看一个例子。



```

var xhr = createXHR();
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
};
xhr.open("get", "example.txt", true);
xhr.send(null);

```

XHRAsyncExample01.htm

以上代码利用 DOM0 级方法为 XHR 对象添加了事件处理程序，原因是并非所有浏览器都支持 DOM2 级方法。与其他事件处理程序不同，这里没有向 onreadystatechange 事件处理程序中传递 event 对象；必须通过 XHR 对象本身来确定下一步该怎么做。



这个例子在 onreadystatechange 事件处理程序中使用了 xhr 对象，没有使用 this 对象，原因是 onreadystatechange 事件处理程序的作用域问题。如果使用 this 对象，在有的浏览器中会导致函数执行失败，或者导致错误发生。因此，使用实际的 XHR 对象实例变量是较为可靠的一种方式。

另外，在接收到响应之前还可以调用 abort() 方法来取消异步请求，如下所示：

```
xhr.abort();
```

调用这个方法后，XHR 对象会停止触发事件，而且也不再允许访问任何与响应有关的对象属性。在终止请求之后，还应该对 XHR 对象进行解引用操作。由于内存原因，不建议重用 XHR 对象。

21.1.2 HTTP 头部信息

每个 HTTP 请求和响应都会带有相应的头部信息，其中有的对开发人员有用，有的也没有什么用。XHR 对象也提供了操作这两种头部（即请求头部和响应头部）信息的方法。

默认情况下，在发送 XHR 请求的同时，还会发送下列头部信息。

- ☐ Accept: 浏览器能够处理的内容类型。
- ☐ Accept-Charset: 浏览器能够显示的字符集。
- ☐ Accept-Encoding: 浏览器能够处理的压缩编码。
- ☐ Accept-Language: 浏览器当前设置的语言。
- ☐ Connection: 浏览器与服务器之间连接的类型。
- ☐ Cookie: 当前页面设置的任何 Cookie。
- ☐ Host: 发出请求的页面所在的域。
- ☐ Referer: 发出请求的页面的 URI。注意，HTTP 规范将这个头部字段拼写错了，而为保证与规范一致，也只能将错就错了。（这个英文单词的正确拼法应该是 referrer。）
- ☐ User-Agent: 浏览器的用户代理字符串。

虽然不同浏览器实际发送的头部信息会有所不同，但以上列出的基本上是所有浏览器都会发送的。使用 setRequestHeader() 方法可以设置自定义的请求头部信息。这个方法接受两个参数：头部字段的名称和头部字段的值。要成功发送请求头部信息，必须在调用 open() 方法之后且调用 send() 方法之前调用 setRequestHeader()，如下面的例子所示。



```
var xhr = createXHR();
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
}
```



```
};  
xhr.open("get", "example.php", true);  
xhr.setRequestHeader("MyHeader", "MyValue");  
xhr.send(null);
```

XHRRequestHeadersExample01.htm

服务器在接收到这种自定义的头部信息之后，可以执行相应的后续操作。我们建议读者使用自定义的头部字段名称，不要使用浏览器正常发送的字段名称，否则有可能会影响服务器的响应。有的浏览器允许开发人员重写默认的头信息，但有的浏览器则不允许这样做。

调用 XHR 对象的 `getResponseHeader()` 方法并传入头部字段名称，可以取得相应的响应头部信息。而调用 `getAllResponseHeaders()` 方法则可以取得一个包含所有头部信息的长字符串。来看下面的例子。

```
var myHeader = xhr.getResponseHeader("MyHeader");  
var allHeaders = xhr.getAllResponseHeaders();
```

在服务器端，也可以利用头部信息向浏览器发送额外的、结构化的数据。在没有自定义信息的情况下，`getAllResponseHeaders()` 方法通常会返回如下所示的多行文本内容：

```
Date: Sun, 14 Nov 2004 18:04:03 GMT  
Server: Apache/1.3.29 (Unix)  
Vary: Accept  
X-Powered-By: PHP/4.3.8  
Connection: close  
Content-Type: text/html; charset=iso-8859-1
```

这种格式化的输出可以方便我们检查响应中所有头部字段的名称，而不必一个一个地检查某个字段是否存在。

21.1.3 GET 请求

GET 是最常见的请求类型，最常用于向服务器查询某些信息。必要时，可以将查询字符串参数追加到 URL 的末尾，以便将信息发送给服务器。对 XHR 而言，位于传入 `open()` 方法的 URL 末尾的查询字符串必须经过正确的编码才行。

使用 GET 请求经常会发生的一个错误，就是查询字符串的格式有问题。查询字符串中每个参数的名称和值都必须使用 `encodeURIComponent()` 进行编码，然后才能放到 URL 的末尾；而且所有名-值对儿都必须由和号（&）分隔，如下面的例子所示。

```
xhr.open("get", "example.php?name1=value1&name2=value2", true);
```

下面这个函数可以辅助向现有 URL 的末尾添加查询字符串参数：

```
function addURLParam(url, name, value) {  
    url += (url.indexOf("?") == -1 ? "?" : "&");  
    url += encodeURIComponent(name) + "=" + encodeURIComponent(value);  
    return url;  
}
```

这个 `addURLParam()` 函数接受三个参数：要添加参数的 URL、参数的名称和参数的值。这个函数首先检查 URL 是否包含问号（以确定是否已经有参数存在）。如果没有，就添加一个问号；否则，就添加一个和号。然后，将参数名称和值进行编码，再添加到 URL 的末尾。最后返回添加参数之后的 URL。

下面是使用这个函数来构建请求 URL 的示例。

```
var url = "example.php";

//添加参数
url = addURLParam(url, "name", "Nicholas");
url = addURLParam(url, "book", "Professional JavaScript");

//初始化请求
xhr.open("get", url, false);
```

在这里使用 `addURLParam()` 函数可以确保查询字符串的格式良好，并可靠地用于 XHR 对象。

21.1.4 POST 请求

使用频率仅次于 GET 的是 POST 请求，通常用于向服务器发送应该被保存的数据。POST 请求应该把数据作为请求的主体提交，而 GET 请求传统上不是这样。POST 请求的主体可以包含非常多的数据，而且格式不限。在 `open()` 方法第一个参数的位置传入 "post"，就可以初始化一个 POST 请求，如下面的例子所示。

```
xhr.open("post", "example.php", true);
```

发送 POST 请求的第二步就是向 `send()` 方法中传入某些数据。由于 XHR 最初的设计主要是为了处理 XML，因此可以在此传入 XML DOM 文档，传入的文档经序列化之后将作为请求主体被提交到服务器。当然，也可以在此传入任何想发送到服务器的字符串。

默认情况下，服务器对 POST 请求和提交 Web 表单的请求并不会一视同仁。因此，服务器端必须有程序来读取发送过来的原始数据，并从中解析出有用的部分。不过，我们可以使用 XHR 来模仿表单提交：首先将 `Content-Type` 头部信息设置为 `application/x-www-form-urlencoded`，也就是表单提交时的内容类型，其次是以适当的格式创建一个字符串。第 14 章曾经讨论过，POST 数据的格式与查询字符串格式相同。如果需要将页面中表单的数据进行序列化，然后再通过 XHR 发送到服务器，那么就可以使用第 14 章介绍的 `serialize()` 函数来创建这个字符串：



```
function submitData(){
    var xhr = createXHR();
    xhr.onreadystatechange = function(){
        if (xhr.readyState == 4){
            if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
                alert(xhr.responseText);
            } else {
                alert("Request was unsuccessful: " + xhr.status);
            }
        }
    };

    xhr.open("post", "postexample.php", true);
    xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    var form = document.getElementById("user-info");
    xhr.send(serialize(form));
}
```

这个函数可以将 ID 为“user-info”的表单中的数据序列化之后发送给服务器。而下面的示例 PHP 文件 `postexample.php` 就可以通过 `$_POST` 取得提交的数据了：

```
<?php
    header("Content-Type: text/plain");
    echo <<<EOF
Name: {$_POST[ 'user-name' ]}
Email: {$_POST[ 'user-email' ]}
EOF;
?>
```

postexample.php

如果不设置 `Content-Type` 头部信息，那么发送给服务器的数据就不会出现在 `$_POST` 超级全局变量中。这时候，要访问同样的数据，就必须借助 `$HTTP_RAW_POST_DATA`。



与 GET 请求相比，POST 请求消耗的资源会更多一些。从性能角度来看，以发送相同的数据计，GET 请求的速度最多可达到 POST 请求的两倍。

21.2 XMLHttpRequest 2 级

鉴于 XHR 已经得到广泛接受，成为了事实标准，W3C 也着手制定相应的标准以规范其行为。XMLHttpRequest 1 级只是把已有的 XHR 对象的实现细节描述了出来。而 XMLHttpRequest 2 级则进一步发展了 XHR。并非所有浏览器都完整地实现了 XMLHttpRequest 2 级规范，但所有浏览器都实现了它规定的部分内容。

21.2.1 FormData

现代 Web 应用中频繁使用的一项功能就是表单数据的序列化，XMLHttpRequest 2 级为此定义了 `FormData` 类型。`FormData` 为序列化表单以及创建与表单格式相同的数据（用于通过 XHR 传输）提供了便利。下面的代码创建了一个 `FormData` 对象，并向其中添加了一些数据。

```
var data = new FormData();
data.append("name", "Nicholas");
```

这个 `append()` 方法接收两个参数：键和值，分别对应表单字段的名称和字段中包含的值。可以像这样添加任意多个键值对儿。而通过向 `FormData` 构造函数中传入表单元素，也可以用表单元素的数据预先向其中填入键值对儿：

```
var data = new FormData(document.forms[0]);
```

创建了 `FormData` 的实例后，可以将其直接传给 XHR 的 `send()` 方法，如下所示：

```
var xhr = createXHR();
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
            alert(xhr.responseText);
        }
    }
}
```




```
    } else {  
        alert("Request was unsuccessful: " + xhr.status);  
    }  
    }  
};  
  
xhr.open("post", "postexample.php", true);  
var form = document.getElementById("user-info");  
xhr.send(new FormData(form));
```


XHRFormDataExample01.htm

使用 FormData 的方便之处体现在不必明确地在 XHR 对象上设置请求头部。XHR 对象能够识别传入的数据类型是 FormData 的实例，并配置适当的头部信息。

支持 FormData 的浏览器有 Firefox 4+、Safari 5+、Chrome 和 Android 3+版 WebKit。

21.2.2 超时设定

IE8 为 XHR 对象添加了一个 timeout 属性，表示请求在等待响应多少毫秒之后就终止。在给 timeout 设置一个数值后，如果在规定的时间内浏览器还没有接收到响应，那么就会触发 timeout 事件，进而会调用 ontimeout 事件处理程序。这项功能后来也被收入了 XMLHttpRequest 2 级规范中。来看下面的例子。



```
var xhr = createXHR();  
xhr.onreadystatechange = function(){  
    if (xhr.readyState == 4){  
        try {  
            if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){  
                alert(xhr.responseText);  
            } else {  
                alert("Request was unsuccessful: " + xhr.status);  
            }  
        } catch (ex){  
            //假设由 ontimeout 事件处理程序处理  
        }  
    }  
};  
  
xhr.open("get", "timeout.php", true);  
xhr.timeout = 1000; //将超时设置为1秒钟 (仅适用于 IE8+)  
xhr.ontimeout = function(){  
    alert("Request did not return in a second.");  
};  
xhr.send(null);
```

XHRTIMEOUTExample01.htm

这个例子示范了如何使用 timeout 属性。将这个属性设置为 1000 毫秒，意味着如果请求在 1 秒钟内还没有返回，就会自动终止。请求终止时，会调用 ontimeout 事件处理程序。但此时 readyState 可能已经改变为 4 了，这意味着会调用 onreadystatechange 事件处理程序。可是，如果在超时终止请求之后再访问 status 属性，就会导致错误。为避免浏览器报告错误，可以将检查 status 属性的语

句封装在一个 try-catch 语句当中。

在写作本书时，IE 8+仍然是唯一支持超时设定的浏览器。

21.2.3 overrideMimeType() 方法

Firefox 最早引入了 overrideMimeType() 方法，用于重写 XHR 响应的 MIME 类型。这个方法后来也被纳入了 XMLHttpRequest 2 级规范。因为返回响应的 MIME 类型决定了 XHR 对象如何处理它，所以提供一种方法能够重写服务器返回的 MIME 类型是很有用的。

比如，服务器返回的 MIME 类型是 text/plain，但数据中实际包含的是 XML。根据 MIME 类型，即使数据是 XML，responseXML 属性中仍然是 null。通过调用 overrideMimeType() 方法，可以保证把响应当作 XML 而非纯文本来处理。

```
var xhr = createXHR();  
xhr.open("get", "text.php", true);  
xhr.overrideMimeType("text/xml");  
xhr.send(null);
```

这个例子强迫 XHR 对象将响应当作 XML 而非纯文本来处理。调用 overrideMimeType() 必须在 send() 方法之前，才能保证重写响应的 MIME 类型。

支持 overrideMimeType() 方法的浏览器有 Firefox、Safari 4+、Opera 10.5 和 Chrome。

21.3 进度事件

Progress Events 规范是 W3C 的一个工作草案，定义了与客户端服务器通信有关的事件。这些事件最早其实只针对 XHR 操作，但目前也被其他 API 借鉴。有以下 6 个进度事件。

- ☐ loadstart: 在接收到响应数据的第一个字节时触发。
- ☐ progress: 在接收响应期间持续不断地触发。
- ☐ error: 在请求发生错误时触发。
- ☐ abort: 在因为调用 abort() 方法而终止连接时触发。
- ☐ load: 在接收到完整的响应数据时触发。
- ☐ loadend: 在通信完成或者触发 error、abort 或 load 事件后触发。

每个请求都从触发 loadstart 事件开始，接下来是一或多个 progress 事件，然后触发 error、abort 或 load 事件中的一个，最后以触发 loadend 事件结束。

支持前 5 个事件的浏览器有 Firefox 3.5+、Safari 4+、Chrome、iOS 版 Safari 和 Android 版 WebKit。Opera (从第 11 版开始)、IE 8+只支持 load 事件。目前还没有浏览器支持 loadend 事件。

这些事件大都很直观，但其中两个事件有一些细节需要注意。

21.3.1 load 事件

Firefox 在实现 XHR 对象的某个版本时，曾致力于简化异步交互模型。最终，Firefox 实现中引入了 load 事件，用以替代 readystatechange 事件。响应接收完毕后将触发 load 事件，因此也就没有必要去检查 readyState 属性了。而 onload 事件处理程序会接收到一个 event 对象，其 target 属性就指向 XHR 对象实例，因而可以访问到 XHR 对象的所有方法和属性。然而，并非所有浏览器都为这个

事件实现了适当的事件对象。结果，开发人员还是要像下面这样被迫使用 XHR 对象变量。



```
var xhr = createXHR();
xhr.onload = function(){
    if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
        alert(xhr.responseText);
    } else {
        alert("Request was unsuccessful: " + xhr.status);
    }
};
xhr.open("get", "altevents.php", true);
xhr.send(null);
```

XHRProgressEventExample01.htm

只要浏览器接收到服务器的响应，不管其状态如何，都会触发 load 事件。而这意味着你必须检查 status 属性，才能确定数据是否真的已经可用了。Firefox、Opera、Chrome 和 Safari 都支持 load 事件。

21.3.2 progress 事件

Mozilla 对 XHR 的另一个革新是添加了 progress 事件，这个事件会在浏览器接收新数据期间周期性地触发。而 onprogress 事件处理程序会接收到一个 event 对象，其 target 属性是 XHR 对象，但包含着三个额外的属性：lengthComputable、position 和 totalSize。其中，lengthComputable 是一个表示进度信息是否可用的布尔值，position 表示已经接收的字节数，totalSize 表示根据 Content-Length 响应头部确定的预期字节数。有了这些信息，我们就可以为用户创建一个进度指示器了。下面展示了为用户创建进度指示器的一个示例。

```
var xhr = createXHR();
xhr.onload = function(event){
    if ((xhr.status >= 200 && xhr.status < 300) ||
        xhr.status == 304){
        alert(xhr.responseText);
    } else {
        alert("Request was unsuccessful: " + xhr.status);
    }
};
xhr.onprogress = function(event){
    var divStatus = document.getElementById("status");
    if (event.lengthComputable){
        divStatus.innerHTML = "Received " + event.position + " of " +
            event.totalSize + " bytes";
    }
};
xhr.open("get", "altevents.php", true);
xhr.send(null);
```

XHRProgressEventExample01.htm

为确保正常执行，必须在调用 open() 方法之前添加 onprogress 事件处理程序。在前面的例子中，

每次触发 progress 事件, 都会以新的状态信息更新 HTML 元素的内容。如果响应头部中包含 Content-Length 字段, 那么也可以利用此信息来计算从响应中已经接收到的数据的百分比。

21.4 跨源资源共享

通过 XHR 实现 Ajax 通信的一个主要限制, 来源于跨域安全策略。默认情况下, XHR 对象只能访问与包含它的页面位于同一个域中的资源。这种安全策略可以预防某些恶意行为。但是, 实现合理的跨域请求对开发某些浏览器应用程序也是至关重要的。

CORS (Cross-Origin Resource Sharing, 跨源资源共享) 是 W3C 的一个工作草案, 定义了在必访访问跨源资源时, 浏览器与服务器应该如何沟通。CORS 背后的基本思想, 就是使用自定义的 HTTP 头部让浏览器与服务器进行沟通, 从而决定请求或响应是应该成功, 还是应该失败。

比如一个简单的使用 GET 或 POST 发送的请求, 它没有自定义的头部, 而主体内容是 text/plain。在发送该请求时, 需要给它附加一个额外的 Origin 头部, 其中包含请求页面的源信息 (协议、域名和端口), 以便服务器根据这个头部信息来决定是否给予响应。下面是 Origin 头部的一个示例:

```
Origin: http://www.nczonline.net
```

如果服务器认为这个请求可以接受, 就在 Access-Control-Allow-Origin 头部中回发相同的源信息 (如果是公共资源, 可以回发*)。例如:

```
Access-Control-Allow-Origin: http://www.nczonline.net
```

如果没有这个头部, 或者有这个头部但源信息不匹配, 浏览器就会驳回请求。正常情况下, 浏览器会处理请求。注意, 请求和响应都不包含 cookie 信息。

21.4.1 IE 对 CORS 的实现

微软在 IE8 中引入了 XDR (XDomainRequest) 类型。这个对象与 XHR 类似, 但能实现安全可靠的跨域通信。XDR 对象的安全机制部分实现了 W3C 的 CORS 规范。以下是 XDR 与 XHR 的一些不同之处。

- cookie 不会随请求发送, 也不会随响应返回。
- 只能设置请求头部信息中的 Content-Type 字段。
- 不能访问响应头部信息。
- 只支持 GET 和 POST 请求。

这些变化使 CSRF (Cross-Site Request Forgery, 跨站点请求伪造) 和 XSS (Cross-Site Scripting, 跨站点脚本) 的问题得到了缓解。被请求的资源可以根据它认为合适的任意数据 (用户代理、来源页面等) 来决定是否设置 Access-Control-Allow-Origin 头部。作为请求的一部分, Origin 头部的值表示请求的来源域, 以便远程资源明确地识别 XDR 请求。

XDR 对象的使用方法与 XHR 对象非常相似。也是创建一个 XDomainRequest 的实例, 调用 open() 方法, 再调用 send() 方法。但与 XHR 对象的 open() 方法不同, XDR 对象的 open() 方法只接收两个参数: 请求的类型和 URL。

所有 XDR 请求都是异步执行的, 不能用它来创建同步请求。请求返回之后, 会触发 load 事件, 响应的数据也会保存在 responseText 属性中, 如下所示。



```
var xdr = new XDomainRequest();
xdr.onload = function(){
    alert(xdr.responseText);
};
xdr.open("get", "http://www.somewhere-else.com/page/");
xdr.send(null);
```

XDomainRequestExample01.htm

在接收到响应后，你只能访问响应的原始文本；没有办法确定响应的状态代码。而且，只要响应有效就会触发 load 事件，如果失败（包括响应中缺少 Access-Control-Allow-Origin 头部）就会触发 error 事件。遗憾的是，除了错误本身之外，没有其他信息可用，因此唯一能够确定的就只有请求未成功了。要检测错误，可以像下面这样指定一个 onerror 事件处理程序。

```
var xdr = new XDomainRequest();
xdr.onload = function(){
    alert(xdr.responseText);
};
xdr.onerror = function(){
    alert("An error occurred.");
};
xdr.open("get", "http://www.somewhere-else.com/page/");
xdr.send(null);
```

XDomainRequestExample01.htm



鉴于导致 XDR 请求失败的因素很多，因此建议你不要忘记通过 onerror 事件处理程序来捕获该事件；否则，即使请求失败也不会有任何提示。

在请求返回前调用 abort() 方法可以终止请求：

```
xdr.abort(); //终止请求
```

与 XHR 一样，XDR 对象也支持 timeout 属性以及 ontimeout 事件处理程序。下面是一个例子。

```
var xdr = new XDomainRequest();
xdr.onload = function(){
    alert(xdr.responseText);
};
xdr.onerror = function(){
    alert("An error occurred.");
};
xdr.timeout = 1000;
xdr.ontimeout = function(){
    alert("Request took too long.");
};
xdr.open("get", "http://www.somewhere-else.com/page/");
xdr.send(null);
```

这个例子会在运行 1 秒钟后超时，并随即调用 ontimeout 事件处理程序。

为支持 POST 请求，XDR 对象提供了 contentType 属性，用来表示发送数据的格式，如下面的例子所示。


```
var xdr = new XDomainRequest();
xdr.onload = function(){
    alert(xdr.responseText);
};
xdr.onerror = function(){
    alert("An error occurred.");
};
xdr.open("post", "http://www.somewhere-else.com/page/");
xdr.contentType = "application/x-www-form-urlencoded";
xdr.send("name1=value1&name2=value2");
```

这个属性是通过 XDR 对象影响头部信息的唯一方式。

21.4.2 其他浏览器对 CORS 的实现

Firefox 3.5+、Safari 4+、Chrome、iOS 版 Safari 和 Android 平台中的 WebKit 都通过 XMLHttpRequest 对象实现了对 CORS 的原生支持。在尝试打开不同来源的资源时，无需额外编写代码就可以触发这个行为。要请求位于另一个域中的资源，使用标准的 XHR 对象并在 open() 方法中传入绝对 URL 即可，例如：

```
var xhr = createXHR();
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if ((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
            alert(xhr.responseText);
        } else {
            alert("Request was unsuccessful: " + xhr.status);
        }
    }
};
xhr.open("get", "http://www.somewhere-else.com/page/", true);
xhr.send(null);
```

与 IE 中的 XDR 对象不同，通过跨域 XHR 对象可以访问 status 和 statusText 属性，而且还支持同步请求。跨域 XHR 对象也有一些限制，但为了安全这些限制是必需的。以下就是这些限制。

- ❑ 不能使用 setRequestHeader() 设置自定义头部。
- ❑ 不能发送和接收 cookie。
- ❑ 调用 getAllResponseHeaders() 方法总会返回空字符串。

由于无论同源请求还是跨源请求都使用相同的接口，因此对于本地资源，最好使用相对 URL，在访问远程资源时再使用绝对 URL。这样做能消除歧义，避免出现限制访问头部或本地 cookie 信息等问题。

21.4.3 Preflighted Requests

CORS 通过一种叫做 Preflighted Requests 的透明服务器验证机制支持开发人员使用自定义的头部、GET 或 POST 之外的方法，以及不同类型的主体内容。在使用下列高级选项来发送请求时，就会向服务器发送一个 Preflight 请求。这种请求使用 OPTIONS 方法，发送下列头部。

- ❑ Origin：与简单的请求相同。
- ❑ Access-Control-Request-Method：请求自身使用的方法。
- ❑ Access-Control-Request-Headers：（可选）自定义的头部信息，多个头部以逗号分隔。

以下是一个带有自定义头部 NCZ 的使用 POST 方法发送的请求。

Origin: http://www.nczonline.net
Access-Control-Request-Method: POST
Access-Control-Request-Headers: NCZ

发送这个请求后，服务器可以决定是否允许这种类型的请求。服务器通过在响应中发送如下头部与浏览器进行沟通。

- ❑ Access-Control-Allow-Origin: 与简单的请求相同。
- ❑ Access-Control-Allow-Methods: 允许的方法，多个方法以逗号分隔。
- ❑ Access-Control-Allow-Headers: 允许的头部，多个头部以逗号分隔。
- ❑ Access-Control-Max-Age: 应该将这个 Preflight 请求缓存多长时间（以秒表示）。

例如：

```
Access-Control-Allow-Origin: http://www.nczonline.net
Access-Control-Allow-Methods: POST, GET
Access-Control-Allow-Headers: NCZ
Access-Control-Max-Age: 1728000
```

Preflight 请求结束后，结果将按照响应中指定的时间缓存起来。而为此付出的代价只是第一次发送这种请求时会多一次 HTTP 请求。

支持 Preflight 请求的浏览器包括 Firefox 3.5+、Safari 4+ 和 Chrome。IE 10 及更早版本都不支持。

21.4.4 带凭据的请求

默认情况下，跨源请求不提供凭据（cookie、HTTP 认证及客户端 SSL 证明等）。通过将 withCredentials 属性设置为 true，可以指定某个请求应该发送凭据。如果服务器接受带凭据的请求，会用下面的 HTTP 头部来响应。

```
Access-Control-Allow-Credentials: true
```

如果发送的是带凭据的请求，但服务器的响应中没有包含这个头部，那么浏览器就不会把响应交给 JavaScript（于是，responseText 中将是空字符串，status 的值为 0，而且会调用 onerror() 事件处理程序）。另外，服务器还可以在 Preflight 响应中发送这个 HTTP 头部，表示允许源发送带凭据的请求。

支持 withCredentials 属性的浏览器有 Firefox 3.5+、Safari 4+ 和 Chrome。IE 10 及更早版本都不支持。

21.4.5 跨浏览器的 CORS

即使浏览器对 CORS 的支持程度并不都一样，但所有浏览器都支持简单的（非 Preflight 和不带凭据的）请求，因此有必要实现一个跨浏览器的方案。检测 XHR 是否支持 CORS 的最简单方式，就是检查是否存在 withCredentials 属性。再结合检测 XDomainRequest 对象是否存在，就可以兼顾所有浏览器了。



```
function createCORSRequest(method, url){
    var xhr = new XMLHttpRequest();
    if ("withCredentials" in xhr){
        xhr.open(method, url, true);
    } else if (typeof XDomainRequest != "undefined"){
        vxhr = new XDomainRequest();
        vxhr.open(method, url);
    }
}
```

```
    } else {  
        xhr = null;  
    }  
    return xhr;  
}  
  
var request = createCORSRequest("get", "http://www.somewhere-else.com/page/");  
if (request){  
    request.onload = function(){  
        //对 request.responseText 进行处理  
    };  
    request.send();  
}
```

CrossBrowserCORSRequestExample01.htm

Firefox、Safari 和 Chrome 中的 XMLHttpRequest 对象与 IE 中的 XDomainRequest 对象类似，都提供了够用的接口，因此以上模式还是相当有用的。这两个对象共同的属性/方法如下。

- abort(): 用于停止正在进行的请求。
- onerror: 用于替代 onreadystatechange 检测错误。
- onload: 用于替代 onreadystatechange 检测成功。
- responseText: 用于取得响应内容。
- send(): 用于发送请求。

以上成员都包含在 createCORSRequest() 函数返回的对象中，在所有浏览器中都能正常使用。

21.5 其他跨域技术

在 CORS 出现以前，要实现跨域 Ajax 通信颇费一些周折。开发人员想出了一些办法，利用 DOM 中能够执行跨域请求的功能，在不依赖 XHR 对象的情况下也能发送某种请求。虽然 CORS 技术已经无处不在，但开发人员自己发明的这些技术仍然被广泛使用，毕竟这样不需要修改服务器端代码。

21.5.1 图像 Ping

上述第一种跨域请求技术是使用标签。我们知道，一个网页可以从任何网页中加载图像，不用担心跨域不跨域。这也是在线广告跟踪浏览量的主要方式。正如第 13 章讨论过的，也可以动态地创建图像，使用它们的 onload 和 onerror 事件处理程序来确定是否接收到了响应。

动态创建图像经常用于图像 Ping。图像 Ping 是与服务器进行简单、单向的跨域通信的一种方式。请求的数据是通过查询字符串形式发送的，而响应可以是任意内容，但通常是像素图或 204 响应。通过图像 Ping，浏览器得不到任何具体的数据，但通过侦听 load 和 error 事件，它能知道响应是什么时候接收到的。来看下面的例子。



```
var img = new Image();  
img.onload = img.onerror = function(){  
    alert("Done!");  
};  
img.src = "http://www.example.com/test?name=Nicholas";
```

ImagePingExample01.htm

这里创建了一个 Image 的实例,然后将 onload 和 onerror 事件处理程序指定为同一个函数。这样无论是什么响应,只要请求完成,就能得到通知。请求从设置 src 属性那一刻开始,而这个例子在请求中发送了一个 name 参数。

图像 Ping 最常用于跟踪用户点击页面或动态广告曝光次数。图像 Ping 有两个主要的缺点,一是只能发送 GET 请求,二是无法访问服务器的响应文本。因此,图像 Ping 只能用于浏览器与服务器间的单向通信。

21.5.2 JSONP

JSONP 是 JSON with padding (填充式 JSON 或参数式 JSON) 的简写,是应用 JSON 的一种新方法,在后来的 Web 服务中非常流行。JSONP 看起来与 JSON 差不多,只不过是包含在函数调用中的 JSON,就像下面这样。

```
callback({ "name": "Nicholas" });
```

JSONP 由两部分组成:回调函数和数据。回调函数是当响应到来时应该在页面中调用的函数。回调函数的名字一般是在请求中指定的。而数据就是传入回调函数中的 JSON 数据。下面是一个典型的 JSONP 请求。

```
http://freegeoip.net/json/?callback=handleResponse
```

这个 URL 是在请求一个 JSONP 地理定位服务。通过查询字符串来指定 JSONP 服务的回调参数是很常见的,就像上面的 URL 所示,这里指定的回调函数的名字叫 handleResponse()。

JSONP 是通过动态<script>元素(要了解详细信息,请参考第 13 章)来使用的,使用时可以为 src 属性指定一个跨域 URL。这里的<script>元素与元素类似,都有能力不受限制地从其他域加载资源。因为 JSONP 是有效的 JavaScript 代码,所以在请求完成后,即在 JSONP 响应加载到页面中以后,就会立即执行。来看一个例子。

```
function handleResponse(response){
    alert("You're at IP address " + response.ip + ", which is in " +
        response.city + ", " + response.region_name);
}

var script = document.createElement("script");
script.src = "http://freegeoip.net/json/?callback=handleResponse";
document.body.insertBefore(script, document.body.firstChild);
```

[JSONPExample01.htm](#)

这个例子通过查询地理定位服务来显示你的 IP 地址和位置信息。

JSONP 之所以在开发人员中极为流行,主要原因是它非常简单易用。与图像 Ping 相比,它的优点在于能够直接访问响应文本,支持在浏览器与服务器之间双向通信。不过,JSONP 也有两点不足。

首先,JSONP 是从其他域中加载代码执行。如果其他域不安全,很可能在响应中夹带一些恶意代码,而此时除了完全放弃 JSONP 调用之外,没有办法追究。因此在使用不是你自已运维的 Web 服务时,一定得保证它安全可靠。

其次,要确定 JSONP 请求是否失败并不容易。虽然 HTML5 给<script>元素新增了一个 onerror 事件处理程序,但目前还没有得到任何浏览器支持。为此,开发人员不得不使用计时器检测指定时间内



是否接收到了响应。但就算这样也不能尽如人意，毕竟不是每个用户上网的速度和带宽都一样。

21.5.3 Comet

Comet 是 Alex Russell^①发明的一个词儿，指的是一种更高级的 Ajax 技术（经常也有人称为“服务器推送”）。Ajax 是一种从页面向服务器请求数据的技术，而 Comet 则是一种服务器向页面推送数据的技术。Comet 能够让信息近乎实时地被推送到页面上，非常适合处理体育比赛的分数和股票报价。

有两种实现 Comet 的方式：长轮询和流。长轮询是传统轮询（也称为短轮询）的一个翻版，即浏览器定时向服务器发送请求，看有没有更新的数据。图 21-1 展示的是短轮询的时间线。

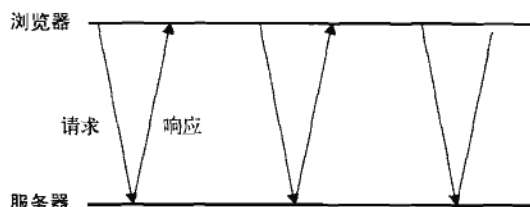


图 21-1

长轮询把短轮询颠倒了一下。页面发起一个到服务器的请求，然后服务器一直保持连接打开，直到有数据可发送。发送完数据之后，浏览器关闭连接，随即又发起一个到服务器的新请求。这一过程在页面打开期间一直持续不断。图 21-2 展示了长轮询的时间线。

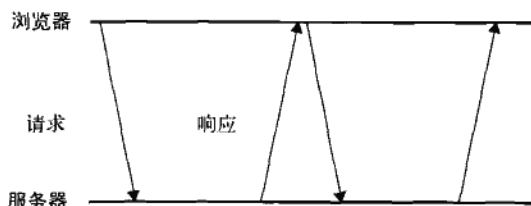


图 21-2

无论是短轮询还是长轮询，浏览器都要在接收数据之前，先发起对服务器的连接。两者最大的区别在于服务器如何发送数据。短轮询是服务器立即发送响应，无论数据是否有效，而长轮询是等待发送响应。轮询的优势是所有浏览器都支持，因为使用 XHR 对象和 `setTimeout()` 就能实现。而你要做的就是决定什么时候发送请求。

第二种流行的 Comet 实现是 HTTP 流。流不同于上述两种轮询，因为它在页面的整个生命周期内只使用一个 HTTP 连接。具体来说，就是浏览器向服务器发送一个请求，而服务器保持连接打开，然后周期性地向浏览器发送数据。比如，下面这段 PHP 脚本就是采用流实现的服务器中常见的形式。

```
<?php
    $i = 0;
    while(true){
```

^① Alex Russell 是著名 JavaScript 框架 Dojo 的创始人。


```
//输出一些数据, 然后立即刷新输出缓存
echo "Number is $i";
flush();

//等几秒钟
sleep(10);

$i++;
}
```

所有服务器端语言都支持打印到输出缓存然后刷新（将输出缓存中的内容一次性全部发送到客户端）的功能。而这正是实现 HTTP 流的关键所在。

在 Firefox、Safari、Opera 和 Chrome 中，通过侦听 `readystatechange` 事件及检测 `readyState` 的值是否为 3，就可以利用 XHR 对象实现 HTTP 流。在上述这些浏览器中，随着不断从服务器接收数据，`readyState` 的值会周期性地变为 3。当 `readyState` 值变为 3 时，`responseText` 属性中就会保存接收到的所有数据。此时，就需要比较此前接收到的数据，决定从什么位置开始取得最新的数据。使用 XHR 对象实现 HTTP 流的典型代码如下所示。



```
function createStreamingClient(url, progress, finished){
    var xhr = new XMLHttpRequest(),
        received = 0;

    xhr.open("get", url, true);
    xhr.onreadystatechange = function(){
        var result;

        if (xhr.readyState == 3){

            //只取得最新数据并调整计数器
            result = xhr.responseText.substring(received);
            received += result.length;

            //调用 progress 回调函数
            progress(result);

        } else if (xhr.readyState == 4){
            finished(xhr.responseText);
        }

    };
    xhr.send(null);
    return xhr;
}

var client = createStreamingClient("streaming.php", function(data){
    alert("Received: " + data);
}, function(data){
    alert("Done!");
});
```

[HTTPStreamingExample01.htm](#)

这个 `createStreamingClient()` 函数接收三个参数：要连接的 URL、在接收到数据时调用的函数以及关闭连接时调用的函数。有时候，当连接关闭时，很可能还需要重新建立，所以关注连接什么时

候关闭还是有必要的。

只要 `readystatechange` 事件发生, 而且 `readyState` 值为 3, 就对 `responseText` 进行分割以取得最新数据。这里的 `received` 变量用于记录已经处理了多少个字符, 每次 `readyState` 值为 3 时都递增。然后, 通过 `progress` 回调函数来处理传入的新数据。而当 `readyState` 值为 4 时, 则执行 `finished` 回调函数, 传入响应返回的全部内容。

虽然这个例子比较简单, 而且也能在大多数浏览器中正常运行 (IE 除外), 但管理 Comet 的连接是很容易出错的, 需要时间不断改进才能达到完美。浏览器社区认为 Comet 是未来 Web 的一个重要组成部分, 为了简化这一技术, 又为 Comet 创建了两个新的接口。

21.5.4 服务器发送事件

SSE (Server-Sent Events, 服务器发送事件) 是围绕只读 Comet 交互推出的 API 或者模式。SSE API 用于创建到服务器的单向连接, 服务器通过这个连接可以发送任意数量的数据。服务器响应的 MIME 类型必须是 `text/event-stream`, 而且是浏览器中的 JavaScript API 能解析格式输出。SSE 支持短轮询、长轮询和 HTTP 流, 而且能在断开连接时自动确定何时重新连接。有了这么简单实用的 API, 再实现 Comet 就容易多了。

支持 SSE 的浏览器有 Firefox 6+、Safari 5+、Opera 11+、Chrome 和 iOS 4+ 版 Safari。

1. SSE API

SSE 的 JavaScript API 与其他传递消息的 JavaScript API 很相似。要预订新的事件流, 首先要创建一个新的 `EventSource` 对象, 并传进一个入口点:

```
var source = new EventSource("myevents.php");
```

注意, 传入的 URL 必须与创建对象的页面同源 (相同的 URL 模式、域及端口)。`EventSource` 的实例有一个 `readyState` 属性, 值为 0 表示正连接到服务器, 值为 1 表示打开了连接, 值为 2 表示关闭了连接。

另外, 还有以下三个事件。

- `open`: 在建立连接时触发。
- `message`: 在从服务器接收到新事件时触发。
- `error`: 在无法建立连接时触发。

就一般的用法而言, `onmessage` 事件处理程序也没有什么特别的。

```
source.onmessage = function(event){  
    var data = event.data;  
    //处理数据  
};
```

服务器发回的数据以字符串形式保存在 `event.data` 中。

默认情况下, `EventSource` 对象会保持与服务器的活动连接。如果连接断开, 还会重新连接。这就意味着 SSE 适合长轮询和 HTTP 流。如果想强制立即断开连接并且不再重新连接, 可以调用 `close()` 方法。

```
source.close();
```

2. 事件流

所谓的服务器事件会通过一个持久的 HTTP 响应发送, 这个响应的 MIME 类型为 `text/event-`

stream。响应的格式是纯文本，最简单的情况是每个数据项都带有前缀 data:，例如：

```
data: foo

data: bar

data: foo
data: bar
```

对以上响应而言，事件流中的第一个 message 事件返回的 event.data 值为"foo"，第二个 message 事件返回的 event.data 值为"bar"，第三个 message 事件返回的 event.data 值为"foo\nbar"（注意中间的换行符）。对于多个连续的以 data: 开头的数据行，将作为多段数据解析，每个值之间以一个换行符分隔。只有在包含 data: 的数据行后面有空行时，才会触发 message 事件，因此在服务器上生成事件流时不能忘了多添加这一行。

通过 id: 前缀可以给特定的事件指定一个关联的 ID，这个 ID 行位于 data: 行前面或后面皆可：

```
data: foo
id: 1
```

· 设置了 ID 后，EventSource 对象会跟踪上一次触发的事件。如果连接断开，会向服务器发送一个包含名为 Last-Event-ID 的特殊 HTTP 头部的请求，以便服务器知道下一次该触发哪个事件。在多次连接的事件流中，这种机制可以确保浏览器以正确的顺序收到连接的数据段。

21.5.5 Web Sockets

要说最令人津津乐道的新浏览器 API，就得数 Web Sockets 了。Web Sockets 的目标是在一个单独的持久连接上提供全双工、双向通信。在 JavaScript 中创建了 Web Socket 之后，会有一个 HTTP 请求发送到浏览器以发起连接。在取得服务器响应后，建立的连接会使用 HTTP 升级从 HTTP 协议交换为 Web Socket 协议。也就是说，使用标准的 HTTP 服务器无法实现 Web Sockets，只有支持这种协议的专门服务器才能正常工作。

由于 Web Sockets 使用了自定义的协议，所以 URL 模式也略有不同。未加密的连接不再是 http://，而是 ws://；加密的连接也不是 https://，而是 wss://。在使用 Web Socket URL 时，必须带着这个模式，因为将来还有可能支持其他模式。

使用自定义协议而非 HTTP 协议的好处是，能够在客户端和服务器之间发送非常少量的数据，而不必担心 HTTP 那样字节级的开销。由于传递的数据包很小，因此 Web Sockets 非常适合移动应用。毕竟对移动应用而言，带宽和网络延迟都是关键问题。使用自定义协议的缺点在于，制定协议的时间比制定 JavaScript API 的时间还要长。Web Sockets 曾几度搁浅，就因为不断有人发现这个新协议存在一致性和安全性的问题。Firefox 4 和 Opera 11 都曾默认启用 Web Sockets，但在发布前夕又禁用了，因为又发现了安全隐患。目前支持 Web Sockets 的浏览器有 Firefox 6+、Safari 5+、Chrome 和 iOS 4+版 Safari。

1. Web Sockets API

要创建 Web Socket，先实例一个 WebSocket 对象并传入要连接的 URL：

```
var socket = new WebSocket("ws://www.example.com/server.php");
```

注意，必须给 WebSocket 构造函数传入绝对 URL。同源策略对 Web Sockets 不适用，因此可以通过它打开到任何站点的连接。至于是否会与某个域中的页面通信，则完全取决于服务器。（通过握手信息就可以知道请求来自何方。）

实例化了 WebSocket 对象后,浏览器就会马上尝试创建连接。与 XHR 类似,WebSocket 也有一个表示当前状态的 `readyState` 属性。不过,这个属性的值与 XHR 并不相同,而是如下所示。

- ❑ `WebSocket.OPENING (0)`: 正在建立连接。
- ❑ `WebSocket.OPEN (1)`: 已经建立连接。
- ❑ `WebSocket.CLOSING (2)`: 正在关闭连接。
- ❑ `WebSocket.CLOSE (3)`: 已经关闭连接。

WebSocket 没有 `readystatechange` 事件;不过,它有其他事件,对应着不同的状态。`readyState` 的值永远从 0 开始。

要关闭 Web Socket 连接,可以在任何时候调用 `close()` 方法。

```
socket.close();
```

调用了 `close()` 之后, `readyState` 的值立即变为 2 (正在关闭),而在关闭连接后就会变成 3。

2. 发送和接收数据

Web Socket 打开之后,就可以通过连接发送和接收数据。要向服务器发送数据,使用 `send()` 方法并传入任意字符串,例如:

```
var socket = new WebSocket("ws://www.example.com/server.php");  
socket.send("Hello world!");
```

因为 Web Sockets 只能通过连接发送纯文本数据,所以对于复杂的数据结构,在通过连接发送之前,必须进行序列化。下面的例子展示了先将数据序列化为一个 JSON 字符串,然后再发送到服务器:

```
var message = {  
  time: new Date(),  
  text: "Hello world!",  
  clientId: "asdfp8734rew"  
};  
  
socket.send(JSON.stringify(message));
```

接下来,服务器要读取其中的数据,就要解析接收到的 JSON 字符串。

当服务器向客户端发来消息时,WebSocket 对象就会触发 `message` 事件。这个 `message` 事件与其他传递消息的协议类似,也是把返回的数据保存在 `event.data` 属性中。

```
socket.onmessage = function(event){  
  var data = event.data;  
  
  //处理数据  
};
```

与通过 `send()` 发送到服务器的数据一样, `event.data` 中返回的数据也是字符串。如果你想得到其他格式的数据,必须手工解析这些数据。

3. 其他事件

WebSocket 对象还有其他三个事件,在连接生命周期的不同阶段触发。

- ❑ `open`: 在成功建立连接时触发。
- ❑ `error`: 在发生错误时触发,连接不能持续。
- ❑ `close`: 在连接关闭时触发。

WebSocket 对象不支持 DOM 2 级事件侦听器,因此必须使用 DOM 0 级语法分别定义每个事件处

理程序。

```
var socket = new WebSocket("ws://www.example.com/server.php");

socket.onopen = function(){
    alert("Connection established.");
};

socket.onerror = function(){
    alert("Connection error.");
};

socket.onclose = function(){
    alert("Connection closed.");
};
```

在这三个事件中，只有 close 事件的 event 对象有额外的信息。这个事件的事件对象有三个额外的属性：wasClean、code 和 reason。其中，wasClean 是一个布尔值，表示连接是否已经明确地关闭；code 是服务器返回的数值状态码；而 reason 是一个字符串，包含服务器发回的消息。可以把这些信息显示给用户，也可以记录到日志中以便将来分析。

```
socket.onclose = function(event){
    console.log("Was clean? " + event.wasClean + " Code=" + event.code + " Reason="
        + event.reason);
};
```

21.5.6 SSE 与 Web Sockets

面对某个具体的用例，在考虑是使用 SSE 还是使用 Web Sockets 时，可以考虑如下几个因素。首先，你是否有自由度建立和维护 Web Sockets 服务器？因为 Web Socket 协议不同于 HTTP，所以现有服务器不能用于 Web Socket 通信。SSE 倒是通过常规 HTTP 通信，因此现有服务器就可以满足需求。

第二个要考虑的问题是到底需不需要双向通信。如果用例只需读取服务器数据（如比赛成绩），那么 SSE 比较容易实现。如果用例必须双向通信（如聊天室），那么 Web Sockets 显然更好。别忘了，在不能选择 Web Sockets 的情况下，组合 XHR 和 SSE 也是能实现双向通信的。

21

21.6 安全

讨论 Ajax 和 Comet 安全的文章可谓连篇累牍，而相关主题的书也已经出了很多本了。大型 Ajax 应用程序的安全问题涉及面非常之广，但我们可以从普遍意义上探讨一些基本的问题。

首先，可以通过 XHR 访问的任何 URL 也可以通过浏览器或服务器来访问。下面的 URL 就是一个例子。

```
/getuserinfo.php?id=23
```

如果是向这个 URL 发送请求，可以想象结果会返回 ID 为 23 的用户的某些数据。谁也无法保证别人不会将这个 URL 的用户 ID 修改为 24、56 或其他值。因此，getuserinfo.php 文件必须知道请求者是否真的有权访问要请求的数据；否则，你的服务器就会门户大开，任何人的数据都可能被泄漏出去。

对于未被授权系统有权访问某个资源的情况，我们称之为 CSRF（Cross-Site Request Forgery，跨站点请求伪造）。未被授权系统会伪装自己，让处理请求的服务器认为它是合法的。受到 CSRF 攻击的 Ajax

程序有大有小, 攻击行为既有旨在揭示系统漏洞的恶作剧, 也有恶意的数据窃取或数据销毁。

为确保通过 XHR 访问的 URL 安全, 通行的做法就是验证发送请求者是否有权限访问相应的资源。有下列几种方式可供选择。

- 要求以 SSL 连接来访问可以通过 XHR 请求的资源。
 - 要求每一次请求都要附带经过相应算法计算得到的验证码。
- 请注意, 下列措施对防范 CSRF 攻击不起作用。
- 要求发送 POST 而不是 GET 请求——很容易改变。
 - 检查来源 URL 以确定是否可信——来源记录很容易伪造。
 - 基于 cookie 信息进行验证——同样很容易伪造。

XHR 对象也提供了一些安全机制, 虽然表面上看可以保证安全, 但实际上却相当不可靠。实际上, 前面介绍的 `open()` 方法还能再接收两个参数: 要随请求一起发送的用户名和密码。带有这两个参数的请求可以通过 SSL 发送给服务器上的页面, 如下面的例子所示。

```
xhr.open("get", "example.php", true, "username", "password"); //不要这样做!!
```



即使可以考虑这种安全机制, 但还是尽量不要这样做。把用户名和密码保存在 JavaScript 代码中本身就是极为不安全的。任何人, 只要他会使用 JavaScript 调试器, 就可以通过查看相应的变量发现纯文本形式的用户名和密码。

21.7 小结

Ajax 是无需刷新页面就能够从服务器取得数据的一种方法。关于 Ajax, 可以从以下几方面来总结一下。

- 负责 Ajax 运作的核心对象是 XMLHttpRequest (XHR) 对象。
- XHR 对象由微软最早在 IE5 中引入, 用于通过 JavaScript 从服务器取得 XML 数据。
- 在此之后, Firefox、Safari、Chrome 和 Opera 都实现了相同的特性, 使 XHR 成为了 Web 的一个事实标准。
- 虽然实现之间存在差异, 但 XHR 对象的基本用法在不同浏览器间还是相对规范的, 因此可以放心地用在 Web 开发当中。

同源策略是对 XHR 的一个主要约束, 它为通信设置了“相同的域、相同的端口、相同的协议”这一限制。试图访问上述限制之外的资源, 都会引发安全错误, 除非采用被认可的跨域解决方案。这个解决方案叫做 CORS (Cross-Origin Resource Sharing, 跨源资源共享), IE8 通过 XDomainRequest 对象支持 CORS, 其他浏览器通过 XHR 对象原生支持 CORS。图像 Ping 和 JSONP 是另外两种跨域通信的技术, 但不如 CORS 稳妥。

Comet 是对 Ajax 的进一步扩展, 让服务器几乎能够实时地向客户端推送数据。实现 Comet 的手段主要有两个: 长轮询和 HTTP 流。所有浏览器都支持长轮询, 而只有部分浏览器原生支持 HTTP 流。SSE (Server-Sent Events, 服务器发送事件) 是一种实现 Comet 交互的浏览器 API, 既支持长轮询, 也支持 HTTP 流。

Web Sockets 是一种与服务器进行全双工、双向通信的信道。与其他方案不同，Web Sockets 不使用 HTTP 协议，而使用一种自定义的协议。这种协议专门为快速传输小数据设计。虽然要求使用不同的 Web 服务器，但却具有速度上的优势。

各方面对 Ajax 和 Comet 的鼓吹吸引了越来越多的开发人员学习 JavaScript，人们对 Web 开发的关注也再度升温。与 Ajax 有关的概念都还相对比较新，这些概念会随着时间推移继续发展。



Ajax 是一个非常庞大的主题，完整地讨论这个主题超出了本书的范围。要想了解有关 Ajax 的更多信息，请读者参考《Ajax 高级程序设计（第 2 版）》。