* 前言:

Web 应用的信息交互过程通常是:

客户端通过浏览器发出一个请求，服务器端接收和审核完请求后进行处理并返回结果给客户端，然后客户端浏览器将信息呈现出来。

这种机制对于信息变化不是特别频繁的应用尚能相安无事，但是对于那些实时要求比较高的应用来说就显得捉襟见肘了。

我们需要一种高效节能的双向通信机制来保证数据的实时传输。有web TCP之称的WebSocket应运而生，给开发人员提供了一把强有力的武器来解决疑难杂症。

* 简介:

WebSocket有web TCP之称,既然是TCP，肯定是用来做通信的。但是和TCP的不同之处在于:

WebSocket作为HTML5中新增的一种通信协议，由通信协议和编程API组成，它能够在浏览器和服务器之间建立双向连接，以基于事件的方式，赋予浏览器原生的实时通信能力，来扩展我们的web应用，增加用户体验，提升应用的性能。

何谓双向？服务器端和客户端可以同时发送并响应请求，而不再像HTTP的请求和响应。

* 为什么使用WebSocket

在WebSocket出现之前，我们有一些其它的实时通讯方案，比较常用的有轮询，长轮询，流，还有基于Flash的交换数据的方式，接下来，我们一一分析一下，各种通信方式的特点。

* + 轮询:

这是最早的一种实现实时web应用的方案；原理比较简单易懂，就是客户端以一定的时间间隔向服务器发送请求，以频繁请求的方式来保持客户端和服务器端的数据同步。

但是问题也很明显：当客户端以固定频率向服务器端发送请求时，服务器端的数据可能并没有更新，这样会带来很多无谓的请求，浪费带宽，效率低下。

* + 长轮询:

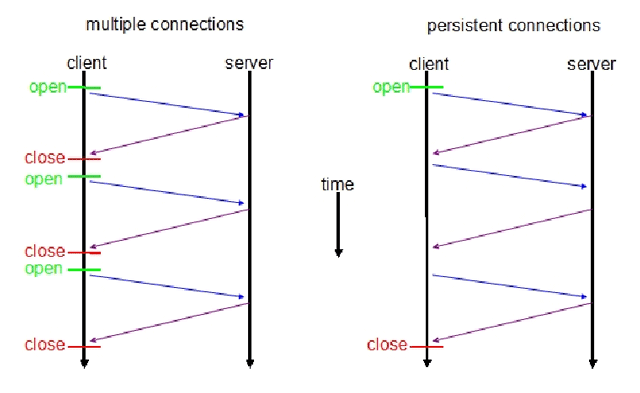
长轮询是对定时轮询的改进和提高，目地是为了降低无效的网络传输。

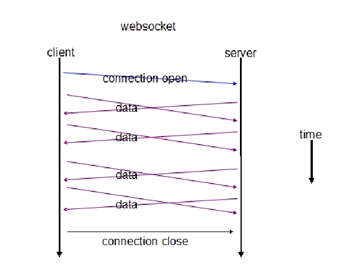
当服务器端没有数据更新的时候，连接会保持一段时间周期直到数据或状态改变或者时间过期，通过这种机制来减少无效的客户端和服务器间的交互。当然，如果服务端的数据变更非常频繁的话，这种机制和定时轮询比较起来没有本质上的性能的提高。

* + 流:
  + 基于Flash的实时通信方式:

Flash有自己的socket实现，这为实时通信提供了可能。我们可以利用Flash完成数据交换，再利用Flash暴露出相应的接口，方便JavaScript调用，来达到实时传输数据的目的。这种方式比前面三种方式都要高效，而且应用场景比较广泛；

因为flash本身的安装率很高；但是在当前的互联网环境下，移动终端对flash的支持并不好，以IOS为主的系统中根本没有flash的存在，而在android阵营中，虽然有flash的支持，但实际的使用效果差强人意，即使是配置较高的移动设备，也很难让人满意。就在前几天(2012年6月底)，Adobe官方宣布，不在支持android4.1以后的系统，这基本上宣告了flash在移动终端上的死亡。



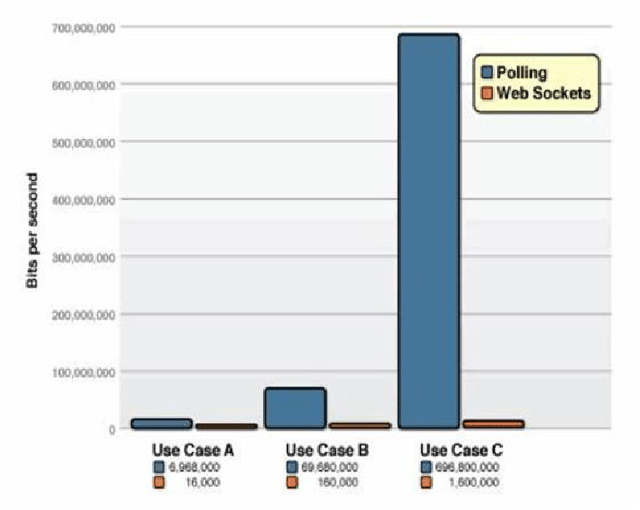


* + 对比:

除了基于Flash的方案外，其它方式都是使用AJAX方式来模拟实时的效果，每次客户端和服务器端时，都是一次完整的HTTP请求和应答的过程，而每一次的HTTP请求和应答都带有完整的HTTP头信息，这就增加了每次的数据传输，而且这些方案中客户端和服务器端的编程实现比较复杂。

* + 为什么要使用WebSocket？

高效节能，简单易用。



在流量和负载增大的情况下，WebSocket方案相比传统的Ajax轮询方案有较大的性能优势;

而在开发方面，也十分简单，我们只需要实例化WebSocket,创建连接，查看是否连接成功，然后就可以发送和相应信息了。

* + 搭建WebSocket服务器

在服务器的选择上很广，基本上，主流语言都有WebSocket的服务器端实现，而作为前端开发工程师，当然要选择现在比较火热的NodeJS作为我们的服务器端环境了。

NodeJS本身并没有原生的WebSocket支持，但是有第三方的实现(大家要是有兴趣的话，完全可以参考WebSocket协议来做自己的实现)，我们选择了“ws”作为我们的服务器端实现。

* WebSocket:
  + 参数原型:

WebSocket构造器方法接受一个必须的参数和一个可选的参数：

WebSocket WebSocket(

in DOMString url,

in optional DOMString protocols

);

WebSocket WebSocket(

in DOMString url,

in optional DOMString[] protocols

);

* + 参数说明:

**url**

表示要连接的URL。这个URL应该为响应WebSocket的地址。

**protocols**可选

可以是一个单个的协议名字字符串或者包含多个协议名字字符串的数组。这些字符串用来表示子协议，这样做可以让一个服务器实现多种WebSocket子协议（例如你可能希望通过制定不同的协议来处理不同类型的交互）。如果没有制定这个参数，它会默认设为一个空字符串。

构造器方法可能抛出以下异常：

**SECURITY\_ERR**

试图连接的端口被屏蔽。

* + 方法:

|  |
| --- |
| void [close](https://developer.mozilla.org/zh-CN/docs/Web/API/WebSocket#close())(in optional unsigned long code, in optional DOMString reason); |
| void [send](https://developer.mozilla.org/zh-CN/docs/Web/API/WebSocket#send())(in DOMString data); |

* + 属性:

|  |  |  |
| --- | --- | --- |
| **属性名** | **类型** | **描述** |
| binaryType | [DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString) | 一个字符串表示被传输二进制的内容的类型。取值应当是"blob"或者"arraybuffer"。  "blob"表示使用DOM[Blob](https://developer.mozilla.org/zh-CN/docs/Web/API/Blob) 对象，而"arraybuffer"表示使用 [ArrayBuffer](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer) 对象。 |
| bufferedAmount | [unsigned long](https://developer.mozilla.org/en/unsigned_long) | 调用 [send()](https://developer.mozilla.org/zh-CN/docs/Web/API/WebSocket#send()) 方法加入到队列中等待传输，但是还没有发出的字节数。该值会在所有队列数据被发送后重置为 0。而当连接关闭时不会设为0。如果持续调用[send()](https://developer.mozilla.org/zh-CN/docs/Web/API/WebSocket#send())，这个值会持续增长。**只读**。 |
| extensions | [DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString) | 服务器选定的扩展。目前这个属性只是一个空字符串，或者是一个包含所有扩展的列表。 |
| onclose | [EventListener](https://developer.mozilla.org/zh-CN/docs/Web/API/EventListener) | 用于监听连接关闭事件监听器。当WebSocket对象的readyState状态变为CLOSED时会触发该事件。这个监听器会接收一个叫close的[CloseEvent](https://developer.mozilla.org/en/WebSockets/WebSockets_reference/CloseEvent)对象。 |
| onerror | [EventListener](https://developer.mozilla.org/zh-CN/docs/Web/API/EventListener) | 当错误发生时用于监听error事件的事件监听器。会接受一个名为error的event对象。 |
| onmessage | [EventListener](https://developer.mozilla.org/zh-CN/docs/Web/API/EventListener) | 一个用于消息事件的事件监听器，这一事件当有消息到达的时候该事件会触发。这个Listener会被传入一个名为"message"的[MessageEvent](https://developer.mozilla.org/en/WebSockets/WebSockets_reference/MessageEvent)对象。 |
| onopen | [EventListener](https://developer.mozilla.org/zh-CN/docs/Web/API/EventListener) | 一个用于连接打开事件的事件监听器。当readyState的值变为OPEN的时候会触发该事件。该事件表明这个连接已经准备好接受和发送数据。这个监听器会接受一个名为"open"的事件对象。 |
| protocol | [DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString) | 一个表明服务器选定的子协议名字的字符串。这个属性的取值会被取值为构造器传入的protocols参数。 |
| readyState | [unsigned short](https://developer.mozilla.org/en/unsigned_short) | 连接的当前状态。取值是 [Ready state constants](https://developer.mozilla.org/zh-CN/docs/Web/API/WebSocket#Ready_state_constants)之一。**只读**。 |
| url | [DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString) | 传入构造器的URL。它必须是一个绝对地址的URL。**只读**。 |

* + 常量

### Ready state constants

这些常量是readyState属性的取值，可以用来描述WebSocket连接的状态。

|  |  |  |
| --- | --- | --- |
| **Constant** | **Value** | **Description** |
| CONNECTING | 0 | 连接还没开启。 |
| OPEN | 1 | 连接已开启并准备好进行通信。 |
| CLOSING | 2 | 连接正在关闭的过程中。 |
| CLOSED | 3 | 连接已经关闭，或者连接无法建立。 |

* + 方法:

### close()

关闭WebSocket连接或停止正在进行的连接请求。如果连接的状态已经是closed，这个方法不会有任何效果。

void close(

in optional unsigned short code,

in optional DOMString reason

);

###### 参数

**code**可选

一个数字值表示关闭连接的状态号，表示连接被关闭的原因。如果这个参数没有被指定，默认的取值是1000 （表示正常连接关闭）。

**reason**可选

一个可读的字符串，表示连接被关闭的原因。这个字符串必须是不长于123字节的UTF-8 文本（**不是**字符）。

###### 可能抛出的异常

**INVALID\_ACCESS\_ERR**

选定了无效的code。

**SYNTAX\_ERR**

reason 字符串太长或者含有unpaired surrogates。

### send()

通过WebSocket连接向服务器发送数据。

void send(

in DOMString data

);

void send(

in ArrayBuffer data

);

void send(

in Blob data

);

###### 参数:

**data**

要发送到服务器的数据。

###### 可能抛出的异常

**INVALID\_STATE\_ERR**

当前连接的状态不是OPEN。

**SYNTAX\_ERR**

数据是一个包含unpaired surrogates的字符串。

* 类型化数组

类型化数组是[JavaScript](http://lib.csdn.net/base/javascript)操作二进制数据的一个接口。

这要从WebGL项目的诞生说起，所谓WebGL，就是指浏览器与显卡之间的通信接口，为了满足JavaScript与显卡之间大量的、实时的数据交换，它们之间的数据通信必须是二进制的，而不能是传统的文本格式。

比如，以文本格式传递一个32位整数，两端的JavaScript脚本与显卡都要进行格式转化，将非常耗时。这时要是存在一种机制，可以像[C语言](http://lib.csdn.net/base/c)那样，直接操作字节，然后将4个字节的32位整数，以二进制形式原封不动地送入显卡，脚本的性能就会大幅提升。

类型化数组（Typed Array）就是在这种背景下诞生的。它很像C语言的数组，允许开发者以数组下标的形式，直接操作内存。有了类型化数组以后，JavaScript的二进制数据处理功能增强了很多，接口之间完全可以用二进制数据通信。

* + 分配内存

类型化数组是建立在ArrayBuffer对象的基础上。它的作用是:

分配一段可以存放数据的连续内存区域。

var buf = new ArrayBuffer(32);

其中，ArrayBuffer对象的byteLength属性，返回所分配的内存区域的字节长度。

var buffer = new ArrayBuffer(32);

buffer.byteLength//= 32

如果要分配的内存区域很大，有可能分配失败（因为没有那么多的连续空余内存），所以有必要检查是否分配成功。

if (buffer.byteLength === n) {

// 成功

} else {

// 失败

}

ArrayBuffer对象有一个slice方法，允许将内存区域的一部分，拷贝生成一个新的ArrayBuffer对象。

slice方法其实包含两步:

* + - 先分配一段新内存;
    - 将原来的ArrayBuffer对象拷贝过去;

slice方法接受两个参数:

* + - 参数一:表示拷贝开始的字节序号;
    - 参数二: 表示拷贝截止的字节序号,如果缺省第二个参数，则默认到原来的ArrayBuffer对象的结尾;

除了slice方法，ArrayBuffer对象不提供任何直接读写内存的方法，只允许在其上方建立视图，然后通过视图读写。

* + 视图
    - 回顾:
      * Int: 4字节
      * Short: 2字节
      * Long: 8字节
      * Byte: 1字节
      * Character:2字节
      * Float:4字节
      * Double:8字节
      * 一个字节 8位。
    - 视图的生成:

ArrayBuffer作为内存区域，可以存放多种类型的数据。不同数据有不同的存储方式，这就叫做“视图”。

目前，JavaScript提供以下类型的视图：

* **Int8Array**：8位有符号整数，长度1个字节。
* **Uint8Array**：8位无符号整数，长度1个字节。
* **Int16Array**：16位有符号整数，长度2个字节。
* **Uint16Array**：16位无符号整数，长度2个字节。
* **Int32Array**：32位有符号整数，长度4个字节。
* **Uint32Array**：32位无符号整数，长度4个字节。
* **Float32Array**：32位浮点数，长度4个字节。
* **Float64Array**：64位浮点数，长度8个字节。

每一种视图都有一个BYTES\_PER\_ELEMENT常数，表示这种数据类型占据的字节数。

**Int8Array.BYTES\_PER\_ELEMENT // 1**

**Uint8Array.BYTES\_PER\_ELEMENT // 1**

**Int16Array.BYTES\_PER\_ELEMENT // 2**

**Uint16Array.BYTES\_PER\_ELEMENT // 2**

**Int32Array.BYTES\_PER\_ELEMENT // 4**

**Uint32Array.BYTES\_PER\_ELEMENT // 4**

**Float32Array.BYTES\_PER\_ELEMENT // 4**

**Float64Array.BYTES\_PER\_ELEMENT // 8**

每一种视图都是一个构造函数，有多种方法可以生成：

1. **在ArrayBuffer对象之上生成视图:**

同一个ArrayBuffer对象之上，可以根据不同的数据类型，建立多个视图。

**// 创建一个8字节的ArrayBuffer**

**var b = new ArrayBuffer(8);**

**// 创建一个指向b的Int32视图，开始于字节0，直到缓冲区的末尾**

**var v1 = new Int32Array(b);**

**// 创建一个指向b的Uint8视图，开始于字节2，直到缓冲区的末尾**

**var v2 = new Uint8Array(b, 2);**

**// 创建一个指向b的Int16视图，开始于字节2，长度为2**

**var v3 = new Int16Array(b, 2, 2);**

上面代码在一段长度为8个字节的内存（b）之上，生成了三个视图：v1、v2和v3。视图的构造函数可以接受三个参数：

* 第一个参数：视图对应的底层ArrayBuffer对象，该参数是必需的。
* 第二个参数：视图开始的字节序号，默认从0开始。
* 第三个参数：视图包含的数据个数，默认直到本段内存区域结束。

1. **直接生成:**

视图还可以不通过ArrayBuffer对象，直接分配内存而生成。

**var f64a = new Float64Array(8);**

**f64a[0] = 10;**

**f64a[1] = 20;**

**f64a[2] = f64a[0] + f64a[1];**

上面代码生成一个8个成员的Float64Array数组（共64字节），然后依次对每个成员赋值。这时，视图构造函数的参数就是成员的个数。可以看到，视图数组的赋值操作与普通数组的操作毫无两样。

1. **将普通数组转为视图数组**

将一个数据类型符合要求的普通数组，传入构造函数，也能直接生成视图。

**var typedArray = new Uint8Array( [ 1, 2, 3, 4 ] );**

上面代码将一个普通的数组，赋值给一个新生成的8位无符号整数的视图数组。 视图数组也可以转换回普通数组。

**var normalArray = Array.apply( [], typedArray );**

1. 视图的操作:

建立了视图以后，就可以进行各种操作了。这里需要明确的是，视图其实就是普通数组，语法完全没有什么不同，只不过它直接针对内存进行操作，而且每个成员都有确定的数据类型。所以，视图就被叫做“类型化数组”。

* 1. 数组操作:

普通数组的操作方法和属性，对类型化数组完全适用。

**var buffer = new ArrayBuffer(16);**

**var int32View = new Int32Array(buffer);**

**for (var i=0; i<int32View.length; i++) {**

**int32View[i] = i\*2;**

**}**

上面代码生成一个16字节的ArrayBuffer对象，然后在它的基础上，建立了一个32位整数的视图。由于每个32位整数占据4个字节，所以一共可以写入4个整数，依次为0，2，4，6。

如果在这段数据上接着建立一个16位整数的视图，则可以读出完全不一样的结果。

**var int16View = new Int16Array(buffer);**

**for (var i=0; i<int16View.length; i++) {**

**console.log("Entry " + i + ": " + int16View[i]);**

**}**

**// Entry 0: 0**

**// Entry 1: 0**

**// Entry 2: 2**

**// Entry 3: 0**

**// Entry 4: 4**

**// Entry 5: 0**

**// Entry 6: 6**

**// Entry 7: 0**

由于每个16位整数占据2个字节，所以整个ArrayBuffer对象现在分成8段。然后，由于x86体系的计算机都采用小端字节序（little endian），相对重要的字节排在后面的内存地址，相对不重要字节排在前面的内存地址，所以就得到了上面的结果。

比如，一个占据四个字节的16进制数0x12345678，决定其大小的最重要的字节是“12”，最不重要的是“78”。小端字节序将最不重要的字节排在前面，储存顺序就是78563412；大端字节序则完全相反，将最重要的字节排在前面，储存顺序就是12345678。

目前，所有个人电脑几乎都是小端字节序，所以类型化数组内部也采用小端字节序读写数据，或者更准确的说，按照本机[**操作系统**](http://lib.csdn.net/base/operatingsystem)设定的字节序读写数据。

这并不意味大端字节序不重要，事实上，很多网络设备和特定的操作系统采用的是大端字节序。这就带来一个严重的问题：如果一段数据是大端字节序，类型化数组将无法正确解析，因为它只能处理小端字节序！为了解决这个问题，JavaScript引入DataView对象，可以设定字节序。

**// 假定某段buffer包含如下字节 [0x02, 0x01, 0x03, 0x07]**

**// 计算机采用小端字节序**

**var uInt16View = new Uint16Array(buffer);**

**// 比较运算**

**if (bufView[0]===258) {**

**console.log("ok");**

**}**

**// 赋值运算**

**uInt16View[0] = 255; // 字节变为[0xFF, 0x00, 0x03, 0x07]**

**uInt16View[0] = 0xff05; // 字节变为[0x05, 0xFF, 0x03, 0x07]**

**uInt16View[1] = 0x0210; // 字节变为[0x05, 0xFF, 0x10, 0x02]**

总之，与普通数组相比，类型化数组的最大优点就是可以直接操作内存，不需要数据类型转换，所以速度快得多。

* + 使用DataView和Int8Array写入相同数据，在内存中的显示如下:



* + - 实现:

**var** arrayBuffer= **new** ArrayBuffer(8);  
**var** dataView= **new** DataView(arrayBuffer);  
**var** buf32= **new** Int32Array(arrayBuffer);  
**var** buf8= **new** Int8Array(arrayBuffer,4);  
dataView.setInt32(0, 698273947);

**console**.log(**"dataView.getInt8= "**+dataView.getInt8(0)+**" "**+dataView.getInt8(1)+**" "**+ dataView.getInt8(2)+**" "**+dataView.getInt8(3));  
  
buf32[1]= 698273947;  
**console**.log(**"buf= "**,buf8.join(**" "**));  
**console**.log(**"dataView.getInt8= "**+dataView.getInt8(0)+**" "**+dataView.getInt8(1)+**" "**+ dataView.getInt8(2)+**" "**+dataView.getInt8(3));

* + - 利用Int8Array等视图，为程序赋值时，采用的是小端字节序（little endian），儿使用DataView视图，操纵数据是，采用的是大端字节序（big endian）;
  1. **buffer属性**

类型化数组的buffer属性，返回整段内存区域对应的ArrayBuffer对象。该属性为只读属性。

**var a = new Float32Array(64);**

**var b = new Uint8Array(a.buffer);**

上面代码的a对象和b对象，对应同一个ArrayBuffer对象，即同一段内存。

* 1. **byteLength属性和byteOffset属性**

**var b = new ArrayBuffer(8);**

**var v1 = new Int32Array(b);**

**var v2 = new Uint8Array(b, 2);**

**var v3 = new Int16Array(b, 2, 2);**

**v1.byteLength // 8**

**v2.byteLength // 6**

**v3.byteLength // 4**

**v1.byteOffset // 0**

**v2.byteOffset // 2**

**v3.byteOffset // 2**

注意将byteLength属性和length属性区分，前者是字节长度，后者是成员长度。

**var a = new Int16Array(8);**

**a.length // 8**

**a.byteLength // 16**

* 1. **set方法**

类型化数组的set方法用于复制数组，也就是将一段内容完全复制到另一段内存。

**var a = new Uint8Array(8);**

**var b = new Uint8Array(8);**

**b.set(a);**

上面代码复制a数组的内容到b数组，它是整段内存的复制，比一个个拷贝成员的那种复制快得多。set方法还可以接受第二个参数，表示从b对象哪一个成员开始复制a对象。

**var a = new Uint16Array(8);**

**var b = new Uint16Array(10);**

**b.set(a,2)**

上面代码的b数组比a数组多两个成员，所以从b[2]开始复制。

* 1. **subarray方法**

subarray方法是对于类型化数组的一部分，再建立一个新的视图。

**var a = new Uint16Array(8);**

**var b = a.subarray(2,3);**

**a.byteLength // 16**

**b.byteLength // 2**

subarray方法的第一个参数是起始的成员序号，第二个参数是结束的成员序号（不含该成员），如果省略则包含剩余的全部成员。所以，上面代码的a.subarray(2,3)，意味着b只包含a[2]一个成员，字节长度为2。

* 1. **ArrayBuffer与字符串的互相转换、**

ArrayBuffer转为字符串，或者字符串转为ArrayBuffer，有一个前提，即字符串的编码方法是确定的。假定字符串采用UTF-16编码（JavaScript的内部编码方式），可以自己编写转换函数。

**// ArrayBuffer转为字符串，参数为ArrayBuffer对象**

**function ab2str(buf) {**

**return String.fromCharCode.apply(null, new Uint16Array(buf));**

**}**

**// 字符串转为ArrayBuffer对象，参数为字符串**

**function str2ab(str) {**

**var buf = new ArrayBuffer(str.length\*2); // 每个字符占用2个字节**

**var bufView = new Uint16Array(buf);**

**for (var i=0, strLen=str.length; i<strLen; i++) {**

**bufView[i] = str.charCodeAt(i);**

**}**

**return buf;**

**}**

1. DataView视图

如果一段数据包括多种类型（比如服务器传来的HTTP数据），这时除了建立ArrayBuffer对象的复合视图以外，还可以通过DataView视图进行操作。

DataView视图提供更多操作选项，而且支持设定字节序。本来，在设计目的上，ArrayBuffer对象的各种类型化视图，是用来向网卡、声卡之类的本机设备传送数据，所以使用本机的字节序就可以了;

而DataView的设计目的，是用来处理网络设备传来的数据，所以大端字节序或小端字节序是可以自行设定的。

DataView本身也是构造函数，接受一个ArrayBuffer对象作为参数，生成视图。

**DataView(ArrayBuffer buffer [, 字节起始位置 [, 长度]]);**

下面是一个实例。

**var buffer = new ArrayBuffer(24);**

**var dv = new DataView(buffer);**

DataView视图提供以下方法读取内存：

* **getInt8**：读取1个字节，返回一个8位整数。
* **getUint8**：读取1个字节，返回一个无符号的8位整数。
* **getInt16**：读取2个字节，返回一个16位整数。
* **getUint16**：读取2个字节，返回一个无符号的16位整数。
* **getInt32**：读取4个字节，返回一个32位整数。
* **getUint32**：读取4个字节，返回一个无符号的32位整数。
* **getFloat32**：读取4个字节，返回一个32位浮点数。
* **getFloat64**：读取8个字节，返回一个64位浮点数。

这一系列get方法的参数都是一个字节序号，表示从哪个字节开始读取。

**var buffer = new ArrayBuffer(24);**

**var dv = new DataView(buffer);**

**// 从第1个字节读取一个8位无符号整数**

**var v1 = dv.getUint8(0);**

**// 从第2个字节读取一个16位无符号整数**

**var v2 = dv.getUint16(1);**

**// 从第4个字节读取一个16位无符号整数**

**var v3 = dv.getUint16(3);**

上面代码读取了ArrayBuffer对象的前5个字节，其中有一个8位整数和两个十六位整数。

如果一次读取两个或两个以上字节，就必须明确数据的存储方式，到底是小端字节序还是大端字节序。默认情况下，DataView的get方法使用大端字节序解读数据，如果需要使用小端字节序解读，必须在get方法的第二个参数指定true。

**// 小端字节序**

**var v1 = dv.getUint16(1, true);**

**// 大端字节序**

**var v2 = dv.getUint16(3, false);**

**// 大端字节序**

**var v3 = dv.getUint16(3);**

DataView视图提供以下方法写入内存：

* **setInt8**：写入1个字节的8位整数。
* **setUint8**：写入1个字节的8位无符号整数。
* **setInt16**：写入2个字节的16位整数。
* **setUint16**：写入2个字节的16位无符号整数。
* **setInt32**：写入4个字节的32位整数。
* **setUint32**：写入4个字节的32位无符号整数。
* **setFloat32**：写入4个字节的32位浮点数。
* **setFloat64**：写入8个字节的64位浮点数。

这一系列set方法，接受两个参数，第一个参数是字节序号，表示从哪个字节开始写入，第二个参数为写入的数据。对于那些写入两个或两个以上字节的方法，需要指定第三个参数，false或者undefined表示使用大端字节序写入，true表示使用小端字节序写入。

**// 在第1个字节，以大端字节序写入值为25的32位整数**

**dv.setInt32(0, 25, false);**

**// 在第5个字节，以大端字节序写入值为25的32位整数**

**dv.setInt32(4, 25);**

**// 在第9个字节，以小端字节序写入值为2.5的32位浮点数**

**dv.setFloat32(8, 2.5, true);**

如果不确定正在使用的计算机的字节序，可以采用下面的判断方式。

**var littleEndian = (function() {**

**var buffer = new ArrayBuffer(2);**

**new DataView(buffer).setInt16(0, 256, true);**

**return new Int16Array(buffer)[0] === 256;**

**})();**

如果返回true，就是小端字节序；如果返回false，就是大端字节序。

1. 应用:
   * + **Ajax**

传统上，服务器通过Ajax操作只能返回文本数据。XMLHttpRequest 第二版允许服务器返回二进制数据，这时分成两种情况。如果明确知道返回的二进制数据类型，可以把返回类型（responseType）设为arraybuffer；如果不知道，就设为blob。

**xhr.responseType = 'arraybuffer';**

* + 字节序

字节序，顾名思义字节的顺序，再多说两句就是大于一个字节类型的数据在内存中的存放顺序。

其实大部分人在实际的开发中都很少会直接和字节序打交道。唯有在跨平台以及网络程序中，字节序才是一个应该被考虑的问题。在所有的介绍字节序的文章中都会提到字节序分为两类：Big-Endian和Little-Endian。

1) Little-Endian就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。  
2) Big-Endian就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。  
3) 网络字节序：TCP/IP各层协议将字节序定义为Big-Endian，因此TCP/IP协议中使用的字节序通常称之为网络字节序。

* + - 如何检查处理器是big-endian还是little-endian?

由于联合体union的存放顺序是所有成员都从低地址开始存放，利用该特性就可以轻松地获得了CPU对内存采用Little-endian还是Big-endian模式读写。例如：  
   int checkCPUendian(){  
        union {  
             unsigned int a;  
             unsigned char b;              
        }c;  
        c.a = 1;  
        return (c.b == 1);         
   }   /\*return 1 : little-endian, return 0:big-endian\*/

* + - 实例:Cocos 2D-JS中，利用WebSocket与服务进行交互
      * 网络传输方式一:字符串
        + 初始化:

**this**.**host** = **"ws://10.10.0.188:8099"**;*//目标WebSocket服务器* **this**.**socket** = **new** WebSocket(**this**.**host**);*//创建  
 //连接成功之后，会自动打开onopen方法* **this**.**socket**.onopen = **function**(evt){  
   
 };  
  
 *//连接成功之后，前台获取后台的信息* **this**.**socket**.onmessage = **function**(evt){  
   
 };  
  
 *//Socket连接失败时，会自动调用该函数* **this**.**socket**.onerror = **function**(evt){  
   
 };  
  
 *//Socket关闭时，自动调用该函数* **this**.**socket**.onclose = **function**(evt){

};

* 发送消息:

**this**.**socket**.send(“sdhfksdhfg”);

* 接受信息:

在初始化的**onmessage(evt)**方法中，就是后台传递过来的字符串。

* 关闭连接:

**this**.**socket**.close();  
**this**.**socket** = **null**;

* + - * 网络传输方式二:二进制数组【ArrayBuffer】
    - 遇到的坑:

1. 如何分别解析二进制流中的某一类型的数据【Byte、Int、Short、Long、UTF8、UTF16】?

* 困难:
  1. 后台传递过来的数据类型是ArrayBuffer,这种数据类型，不可直接操作读写;
  + 解决:

可以使用DataView和Int8Array类型化数组，可以读取数据;

* 1. 在网络上传输方式是大端字节序（Big Endian）,而使用Int8Array等类型化数组，读取数据时是以littleEndian方式读取;
  + 解决:

方法一:使用DataView类型化数组的getInt16()等方法，其中第二参数，不要设置或者设置为false，这种方法读取是按照Big Endian方式读取。

方法二:使用一个缓冲区(普通类型数组)和Int8Array类型化数组，将类型化数组中的内容按照字节大小逐步拷贝到缓冲区中(因为一个字节占用8位);

* 1. 如何将数据转换为对应数据类型？

网络中传输的数据是字节流(例如:0111111111111),将字节流存储到缓冲区(普通数组)中，每8位占用数组中的一个数据;

* + 方法一:使用DataView类型的getInt8()、getInt16()等方法(整数类型未测试)

**var** buffer= **new** DataView(arrBuffer);

/\*\*\*\*\*\*\*\*\*\*\*\*Int类型\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**buffer.getInt8();**

/\*\*\*\*\*\*\*\*\*\*\*\*Int类型\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**buffer.getInt32();**

/\*\*\*\*\*\*\*\*\*\*\*\*Short类型\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**buffer.getInt16();**

/\*\*\*\*\*\*\*\*\*\*\*\*Long类型\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**buffer.getInt32(0)\* (4294967295)+ buffer.getInt32(1);**

/\*\*\*\*\*\*\*\*\*\*\*\*UTF8类型\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**buffer.getUint8();//在传递字符串之前，传递一个字符串的长度，用于截取**

/\*\*\*\*\*\*\*\*\*\*\*\*UTF16类型\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**buffer.getUint16();//在传递字符串之前，传递一个字符串的长度，用于截取**

* + 方法二:使用缓冲区和parseInt()方法

/\*\*\*\*\*\*\*\*\*\*\*\*Int类型\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**var** val = **this**.**pool**[**this**.**position**++];  
**if** (val > 255) {  
 val &= 255;  
}  
**return** val;

/\*\*\*\*\*\*\*\*\*\*\*\*Int类型\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**var** end = **""**;  
**for** (**var** i = 0; i < 4; i++) {  
 end += **this**.**pool**[**this**.**position**++].toString(16)  
}  
**return** parseInt(end, 16);

/\*\*\*\*\*\*\*\*\*\*\*\*Short类型\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**var** end = **""**;  
**for** (**var** i = 0; i < 2; i++) {  
 end += **this**.**pool**[**this**.**position**++].toString(16)  
}  
**return** parseInt(end, 16);

/\*\*\*\*\*\*\*\*\*\*\*\*Long类型\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**var** end = **""**;  
**for** (**var** i = 0; i < 2; i++) {  
 end += **this**.**pool**[**this**.**position**++].toString(16)  
}  
**return** parseInt(end, 16);

/\*\*\*\*\*\*\*\*\*\*\*\*UTF16类型\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**var** len= **this**.readShort();  
*//其中，2个是后台传递前台时，多传了两个字符串结束符(EOF)  
//小端字节和大端字节的转换，EOF放在字节流的前面  
//将arrayBuffer按照DataView视图读取方式读取***var** binary= **new** Uint8Array(**this**.**arrayBuffer**.slice(**this**.**position**+ 2, **this**.**position**+ len));  
**this**.**position**+= len;  
**return this**.binaryArrayToString(binary);

* 1. 需要向后台传递的数据是ArrayBuffer类型的，数据长度不固定，但是ArrayBuffer特性是长度固定，不能动态分配;
  + 解决:

将需要传递的数据，按照一个字节(8位)存储到普通数组(Array)中，在普通数组中，可以使用push()方法追加，向后台发送数据时，将普通数据转为ArrayBuffer类型。

* 1. 如何将Array普通类型转换为ArrayBuffer类型？
  + 解决:

没有直接转换的方法，可以采用折中方法，逐步遍历，为ArrayBuffer赋值。

* 1. 如何将Int等整数类型转换为8位表示的二进制
  + 解决:

使用DataView视图化数组中的setInt8、setInt16、setInt32分别将数值转为为ArrayBuffer数组中的一个数值(Byte、Short、Int)。

* 1. 如何将Long类型在二进制流中表示？
  + 思路:

DataView中没有setInt64和getInt64类型，因此在做的时候，怎么样从二进制流的读取和把Long类型写入数据流，成了问题。

利用取余和取模的方法(Long类型数据对Int操作)

**if**(value<= ***INT\_RANGE***){  
 x.setInt32(0, 0);  
 x.setInt32(4, value);  
}**else**{  
 x.setInt32(0, **Math**.floor(value/***INT\_RANGE***));  
 x.setInt32(4, **Math**.ceil(value%***INT\_RANGE***));  
}

* 思路一:
  1. 解决方法
* 思路二:
  + - Bug:

1. 获取后台传过来的字符串UTF16和本地的字符串，打印出来的结果一致，但是长度大一倍

环境:

虽然，打印出来的结果一致，但是在使用(window.location.href或者post数据时)，就会出现跳不过的现象。

解决:

读取字符串数组(一个字节为一位)，遇到数值为0时，不做任何操作。

* + - ArrayBuffer的缺陷:
      * 长度固定，不能动态存储;
      * 没有使用Long、字符串类型;
      * 不能由Array和ArrayBuffer之间没有直接
  + 摘要:
    - <http://blog.csdn.net/lichwei1983/article/details/43893025>
    - <http://blog.163.com/hlz_2599/blog/static/1423784742013101211841751/>

问题:

如何使用WebSocket?

如何传递Text字符串?

如何传递接受Text字符串？

如何传递、接受二进制数据？

前后台传递时，前后顺序颠倒

每种数据类型，所占用的字节数

使用Int32Array数据类型