第15章 Ajax

1999年，微软公司发布IE5，第一次引入新功能：允许JavaScript脚本向服务器发起HTTP请求。这个功能当时并没有引起注意，直到2004年Gmail发布和2005年Google Map发布，才引起广泛重视。2005年2月，Ajax这个词第一次正式提出，指围绕这个功能进行开发的一整套做法。从此，Ajax成为脚本发起HTTP通信的代名词，W3C也在2006年发布了它的国际标准。

本章我们将学习如下内容：

- Ajax介绍
- 原生Ajax的实现
- Fetch API介绍

15-1 Ajax基本介绍

15-1-1 什么是Ajax

当万维网开始时，网页包含静态内容。对页面上的内容所做的任何更改都需要重新加载整个页面，这样通常会导致在加载页面时屏幕变空白。

1999年，微软公司在IE5中实现了XMLHttp ActiveX控件。它最初是为了outlook Web客户端开发的，允许在后台使用JavaScript异步发送数据。随后其他浏览器也实现了这种技术，不过它仍然是一个相对未知的功能，很少使用。

2004年和2005年，Google公司相继推出了Gmail和Google地图，从这个时候开始异步加载技术开始备受关注。这些Web应用程序使用异步加载技术，通过更改页面部分内容而不进行全面刷新来增强了用户的体验。让用户感觉更像是一个桌面应用程序。

Ajax这个术语是由Jesse James Garrett于2005年在"Ajax：一种新的Web应用程序方法"一文中所创造的，他提到了Google在其最近的Web应用程序中使用到的技术。Ajax是一个简洁的首字母缩写词，指的是该技术在使用的过程中所涉及到的不同部分，全称：Asynchronous JavaScript and XML。

- Asynchronous：翻译成中文是异步的意思，当发送数据请求时，程序不必停下来等待响应。它可以继续运行，等待响应收到时触发事件。通过使用回调来管理这种过程，程序能够以有效的方式运行，避免了数据来回传输带来的延迟。
- JavaScript：利用JavaScript我们可以接收来自服务器端返回的数据，并将这些数据实时的更新到页面上。

- XML：最开始术语Ajax被创造时，经常用XML文档来返回数据。但是实际上可以发送许多不同类型的数据。到目前为止，在Ajax中最常用的是JSON，它比XML更轻量且更易于解析。JSON还具备被JavaScript原生支持的优点，所以我们可以处理JavaScript对象，而不必使用DOM方法来解析XML文件。

在Garrett的文章发表以后，Ajax的使用真正开始起飞。现在用户不必刷新页面，就可以在网页上看到新的内容。例如购物篮的数据在后台更新，但是页面上的内容却可以无缝加载。还有诸如动态加载照片库等，基本上可以这么说，当需要网页部分页面更新时，不太可能不用Ajax。公共API的爆炸式增长，也意味着Ajax比以往任何时候都更多地在站点之间来回传输数据。

当然Ajax也不是只有优点没有缺点，关于Ajax的优缺点总结如下：

优点：

- 页面无刷新，在页面内与服务器通信，减少用户等待的时间，增强了用户体验。
- 使用异步方式与服务器通信，响应速度快。
- 可以把一些原本服务器的工作转接到客户端，利用客户端闲置的能力来处理，减轻了服务器和宽带的负担，节约空间和宽带租用成本。
- 基于标准化的并被广泛支持的技术，不需要下载插件或者小程序。

缺点

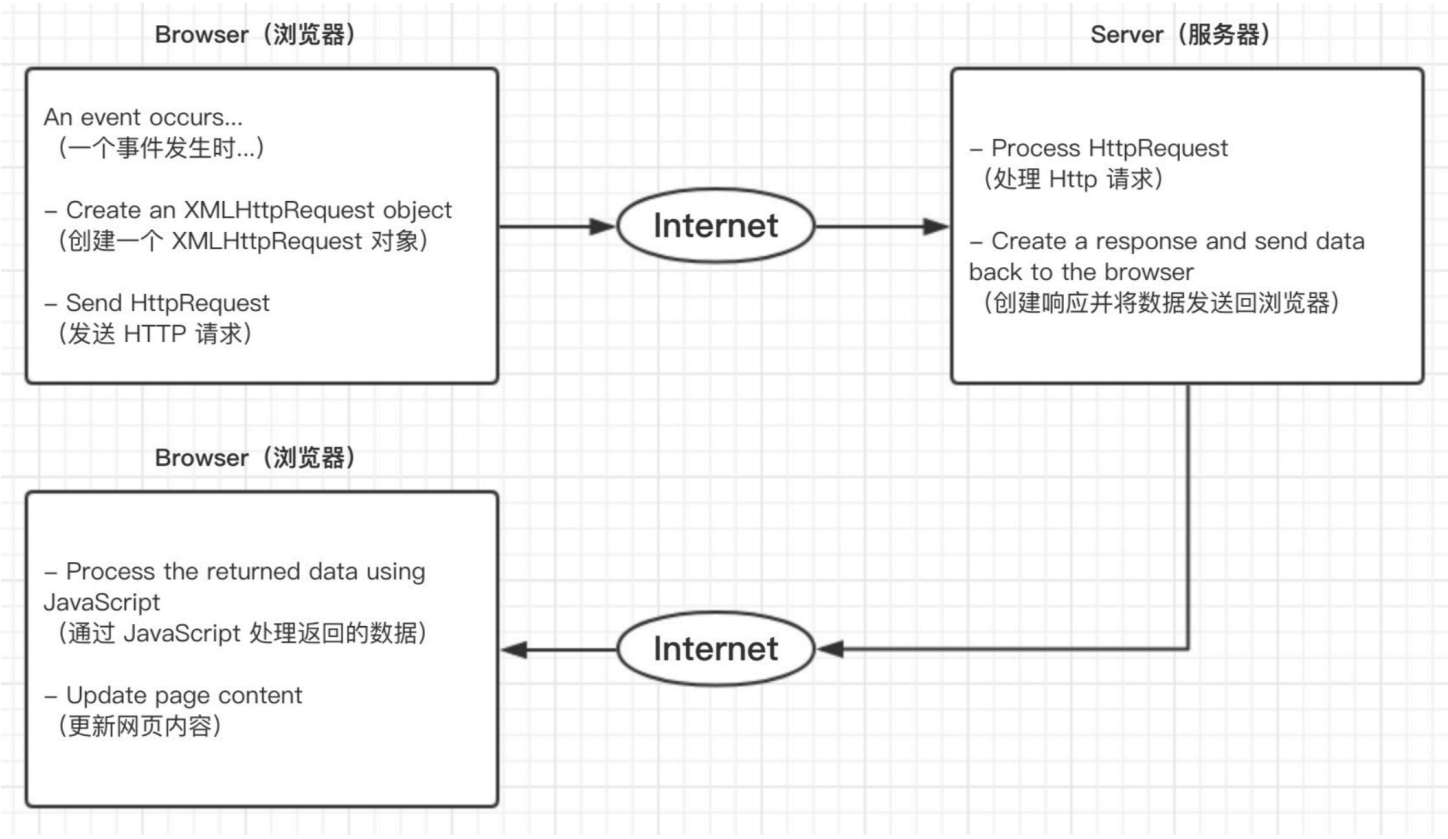
- 无法进行操作的后退，即不支持浏览器的页面后退。
- 对搜索引擎的支持比较弱。
- 可能会影响程序中的异常处理机制。
- 安全问题，对一些网站攻击，如csrf，xxs，sql注入等不能很好的防御。

15-1-2 原生Ajax的实现

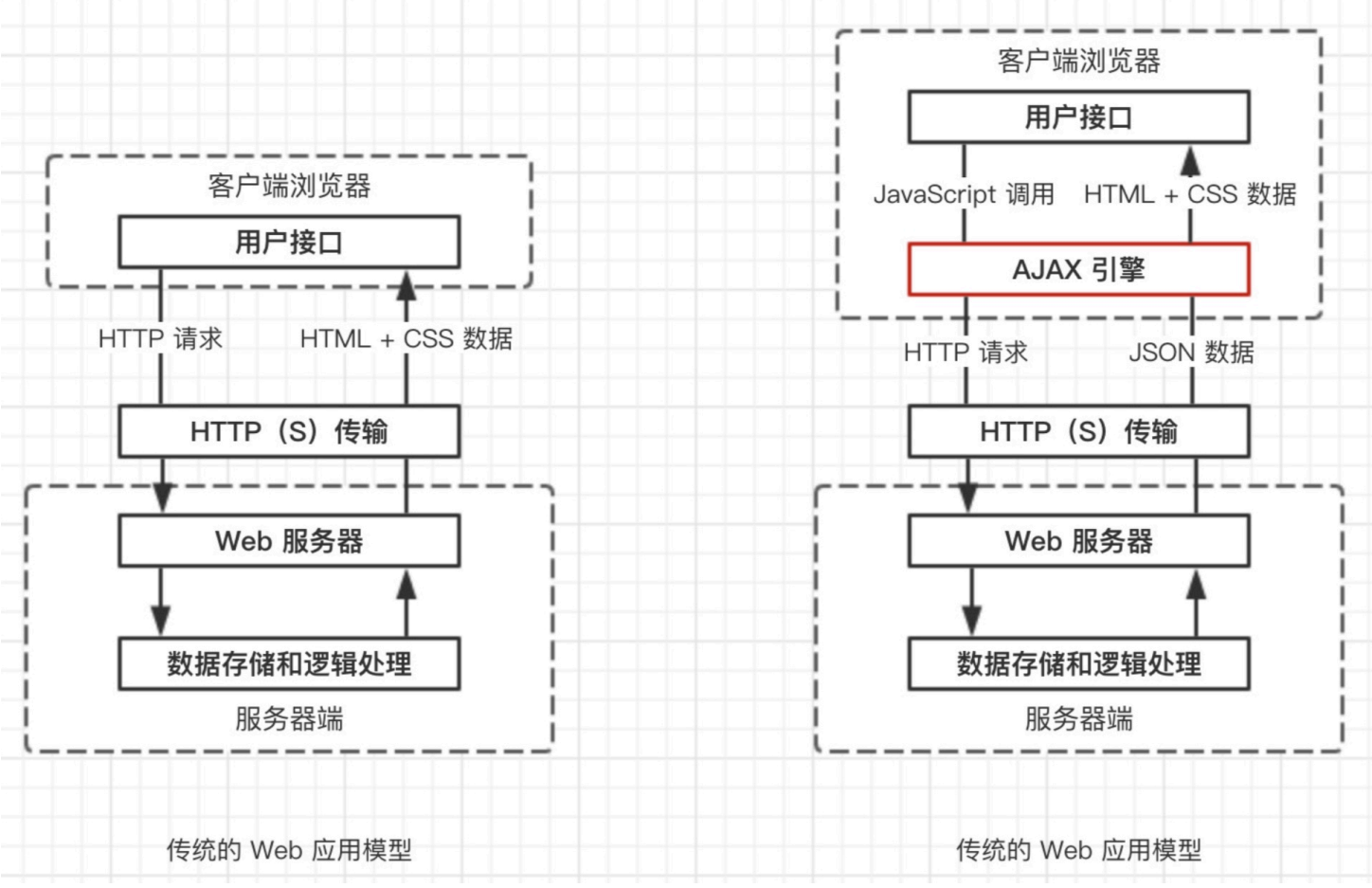
使用原生的Ajax大致包括以下几步骤：

1. 创建XMLHttpRequest对象
2. 发出HTTP请求
3. 接收服务器传回的数据
4. 更新网页数据

概括起来，就是一句话，Ajax通过原生的XMLHttpRequest对象发出HTTP请求，得到服务器返回的数据后，再进行处理。下面我们通过一张图来解释Ajax的工作原理：



下面的图解释了Ajax和传统方式的区别，如下：



大致了解了Ajax的工作原理之后，接下来我们来具体看一下每一个部分。

1. 创建XMLHttpRequest对象

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。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

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，而IE7+及其他标准浏览器都支持原生的XHR对象

创建一个XHR对象，也叫实例化一个XHR对象，因为XMLHTTPRequest()是一个构造函数。下面是创建XHR对象的兼容写法。

```
<body>
    <script>
        let xhr;
        if (window.XMLHttpRequest)
        {
            xhr = new XMLHttpRequest();
        }
        else {
            xhr = new ActiveXObject('Microsoft.XMLHTTP');
        }
    </script>
</body>
```

2. 发送请求

open()方法

在使用XHR对象时，要调用的第一个方法是open()，如下所示，该方法接受3个参数

```
xhr.open(method,url,async)
```

参数说明：

- method：指定发送请求的方式，字符串类型，取值为 GET 或者 POST(不区分大小写，但通常使用大写字母。)
- URL：文件在服务器上的位置。该URL相对于执行代码的当前页面，且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。
- async：表示是否异步处理请求。布尔值类型，默认为true，表示异步，false表示同步。

> 一般来讲，都会产用异步的方式来发送请求，不然Ajax将变得毫无意义。

例如：

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

send()方法

`send()`方法接收一个参数，即要作为请求主体发送的数据：

```
xhr.send(string)
```

参数说明：

- 如果发送请求的方式是 GET，则`send()`方法无参数或者参数为null。
- 如果发送请求的方式是 POST，则`send()`方法的参数为"要发送的数据"。

一般情况下，使用Ajax提交的参数多是些简单的字符串，这种情况下我们可以直接使用 GET 方法将要提交的参数写到`open()`方法的`url`参数中，此时`send()`方法的参数为null。

例如：

```
var url = "login.php?user=XXX&pwd=XXX";
xhr.open("GET",url,true);
xhr.send(null);
```

此外，也可以使用`send()`方法传递参数。使用`send()`方法传递参数使用的是 POST 方法，需要设定`Content-Type`头信息，模拟`HTTP POST`方法发送一个表单，这样服务器才会知道如何处理上传的内容。参数的提交格式和 GET 方法中`url`的写法一样。设置头信息前必须先调用`open()`方法。

例如：

```
xhr.open("POST","login.php",true);
xhr.setRequestHeader("Content-Type","application/x-www-form-urlencoded;charset=UTF-8");
xhr.send("user="+username+"&pwd="+password);
```

GET 和 POST

和 POST 相比，GET 的请求方式更简单也更快，并且在大部分情况下都能使用。但是，在以下

情况中，请使用 POST 请求：

- 无法使用缓存文件(更新服务器上的文件或者数据库)
- 向服务器发送大量数据(POST没有数据量的限制)
- 发送包含未知字符的用户输入时，POST 比 GET 更稳定也更可靠

3. 接收数据

一个完整的HTTP响应由状态码、响应头集合和响应主体组成。在收到响应后，这些都可以通过XHR对象的属性和方法使用，主要有以下4个属性：

- responseText: 作为响应主体被返回的文本(文本形式)
- responseXML: 如果响应的内容类型是`text/xml`或`application/xml`，这个属性中将保存着响应数据的XML DOM文档(document形式)
- status: HTTP状态码(数字形式)
- statusText: HTTP状态说明(文本形式)

在接收到响应后，第一步是检查`status`属性，一般来说，可以将HTTP状态码为200作为成功的标志。此时，`responseText`属性的内容已经就绪。在内容类型正确的情况下，`responseXML`也可以访问。无论内容类型是什么，响应主体的内容都会保存到`responseText`中，对于非XML数据而言，`responseXML`属性的值将为null。

除了上面所说的要对`status`属性进行判断以外，一般还要对XHR对象的`readyState`属性进行判断，该属性会在整个Ajax工作过程中有4个状态，如下：

状态值	描述
0(UNSENT)	未初始化。尚未调用open()方法
1(OPENED)	启动。已经调用open()方法，但尚未调用`send()`方法
2(HEADERS_RECEIVED)	发送。已经调用`send()`方法，且接收到头信息
3(LOADING)	接收。已经接收到部分响应主体信息
4(DONE)	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

理论上，只要`readyState`属性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。但是通常我们只会对`readyState`值为4的阶段感兴趣，因为这时所有数据都已就绪。

所以，接收数据比进行处理的代码一般会写作如下形式：

```
xhr.onreadystatechange = function(){
    if(xhr.readyState === 4)
    {
        if(xhr.status == 200)
        {
            //对返回的数据进行处理
        }
    }
}
```

好了，到目前为止，我们已经介绍了如何创建一个XMLHttpRequest对象，利用这个对象向服务器发送HTTP请求，然后在什么时候接收从服务器传回来的数据。接下来让我们来看一个完整的Ajax示例，这里我们需要创建一个服务器来返回数据。

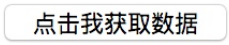
```
<?php
    echo "这是从服务器端获取的数据";
```

这个php文件用于向客户端返回数据。返回的数据也很简单，就是一段"这是从服务器端获取的数据"的文本。接下来我们来书写我们的html文件，代码如下：

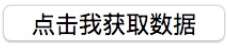
```
<body>
    <button id="btn">点击我获取数据</button>
    <div id="test"></div>
    <script>
        //以兼容的方式创建XMLHttpRequest对象
        let createXHR = function(){
            let xhr;
            if(window.XMLHttpRequest)
            {
                xhr = new XMLHttpRequest();
            }
            else{
                xhr = new ActiveXObject('Microsoft.XMLHTTP');
            }
            return xhr;//返回创建好的XMLHttpRequest对象
        }
        btn.addEventListener("click",function(){
            let xhr = createXHR();//创建XMLHttpRequest对象
            xhr.open("GET","./1.php",true);//以异步的方式打开1.php
            xhr.send(null);//因为是get的方式放松，所以不需要发送数据
            xhr.onreadystatechange = function(){
                if(xhr.readyState === 4)
```

```
                {
                    if(xhr.status === 200)
                    {
                        //将从服务器端取回来的数据添加到div里面
                        test.innerText = xhr.responseText;
                    }
                }
            }
        },false);
    </script>
</body>
```

效果：页面上有一个获取数据的按钮



点击按钮以后会从1.php里面获取数据，然后动态的更新`<div>`的内容



这是从服务器端获取的数据

当然也可以使用后面我们介绍的Mock来拦截Ajax请求，如下：

```
<body>
    <button id="btn">点击我从服务器获取数据</button>
    <span id="test"></span>
    <script>
        btn.onclick = function () {
            // 首先第一步 创建xhr
            let xhr = null;
            if (window.XMLHttpRequest) {
                xhr = new XMLHttpRequest();
            } else {
                xhr = new ActiveXObject('Microsoft.XMLHTTP');
            }
            // 接下来来做第二步 发送请求
            xhr.open('GET', '1.php', true);
            xhr.send(null);
            // 第三步 监听readyState的值 当值为4的时候说明数据已经回来 可以将数据通过J
S添加到页面上面
            xhr.onreadystatechange = function () {
                if (xhr.readyState === 4 && xhr.status === 200) {
                    test.innerHTML = JSON.parse(xhr.responseText).name;
                }
            }
        }
        // 接下来我们使用Mock来截取Ajax请求
        Mock.mock(/\.php/, 'get', function () {
```

```
            return Mock.mock({
                'name': '@cname'
            });
        });
    </script>
</body>
```

15-2 Fetch API

在上一小节中，我们介绍了原生Ajax的实现。可以看到原生的Ajax的实现还是比较复杂的。要根据浏览器的不同然后以不同的方式创建XMLHttpRequest对象，还要进行各种状态的判断。现代的Ajax书写已经很少使用原生的方式来书写了，而是采用各种各样的库中所提供的方式。其中最为出名的就是采用JQuery库中所提供的`$.ajax()`方法来进行Ajax的编写。

现在，书写Ajax又有了新的方式。Fetch API是当前通过网络异步请求和发送数据的现存标准，它使用Promise来避免回调地狱，并简化了许多在使用XMLHttpRequest对象时变得麻烦的概念。

下面是Fetch API的一个基本应用：

```
<body>
    <button id="btn">点击我获取数据</button>
    <div id="test"></div>
    <script>
        btn.addEventListener("click", function () {
            fetch("./1.php")
                .then((response) => response.text())
                .then(text => {
                    test.innerText = text;
                })
                .catch(error => console.log("出错了"));
        }, false);
    </script>
</body>
```

效果：和前面的Ajax示例的效果是一样的，点击获取数据的按钮后会从1.php里面获取数据，然后动态的更新`<div>`的内容。但是，我们可以明显的看到使用Fetch API来书写的Ajax代码要比原生的简单的多。

在上面的代码中，`fetch()`方法返回一个promise，它解析从作为参数提供的URL返回的响应。在上面的示例中，当从URL`./1.php`中接收到返回的数据时，promise将会被解析。因为这是一个promise，我们还可以在最后使用`catch`语句来处理可能发生的任何错误。

15-2-1 Response接口

Fetch API引入了Response接口，该接口处理履行承诺时返回的对象。

1. Response对象的属性和方法

Response对象有许多让我们可以有效处理响应的属性和方法，如下：

- OK：用于检查响应是否成功。
- statusText：状态信息。
- headers：包含了与响应有关的所有HTTP头的一个Headers对象。
- url：一个包含了响应的URL的字符串。
- redirected：一个布尔值，指定响应的结果是否是重定向。
- type：一个字符串值，取值包括`basic`，`cors`，`error`或者`opaque`。`basic`值表示来自于同一个域的响应。`cors`值表示来自不同域的响应。`opaque`表示来自另一个域的非CORS请求的响应，这意味着对数据的访问将被严格限制。`error`表示发生了网络错误。

除了上面所介绍的那些属性以外，还包含了许多返回promise的方法，如下：

- text()：用于处理文本格式的Ajax响应。它从响应中获取文本流，将其读完，然后返回一个被解决为string对象的promise。
- blob()：用于处理二进制文件格式(比如图片或者电子表格)的Ajax响应。它读取文件的原始数据，一旦读取完整个文件，就返回一个被解决为blob对象的promise。
- json()：用于处理JSON格式的Ajax的响应。它将JSON数据流转换为一个被解决为JavaScript对象的promise。
- redirect()：可以用于重定向到另一个URL。它会创建一个新的promise，以解决来自重定向的URL的响应。

> 目前为止还没有浏览器支持`redirect()`方法。

2. 创建响应对象

一般来讲，响应对象都是由服务器来返回的。不过，我们还可以使用`Response()`构造器函数创建一个我们自己的响应对象，如下：

```
let respone = new Response("这是模拟从服务器端获取的数据",{
    ok : true,
    status : 200,
    statusText : 'OK',
    type : 'cors',
    url : "./1.php"
});
```

这里我们就使用`Response()`构造器函数来创建了一个响应对象。其中第一个实参是要返回的数据(例如一个文本，文件或者JSON数据)。第二个实参是可以用来给上面列出的任何属性提供值的对象。

如果我们在创建一个需要发送一个响应的API，或者需要发送一个测试用途的虚拟响应，那么自

定义的响应对象就会很有用。

15-2-2 Request接口

如果我们要对请求进行更细粒度的控制，可以给`fetch()`方法提供一个Request对象来作为实参。这就让我们可以对请求设置很多选项。

请求对象可以用`Request()`构造函数来创建，该对象包含如下属性：

- url：被请求的资源的URL(唯一必须的属性)。
- method：指定请求所有的HTTP方法的字符串。默认是GET。
- headers：这是一个Headers对象(后面会具体介绍)，它提供请求头的细节。
- mode：指定是否用CORS。默认是启用CORS。
- cache：指定请求如何使用浏览器的缓存。例如我们可以强制请求一个资源，并用结果更新缓存，或者强制只在缓存中查找资源。
- credentials：指定请求中是否可以带有cookie。
- redirect：指定如果响应返回一个重定向该如何做。有三个可选值：follow(跟随重定向)，error(抛出一个错误)，manual(用户必须点击链接以跟随重定向)。

下面是一个创建新的请求对象的示例，如下：

```
let request = new Request("./1.php",{
    method : "GET",
    mode : "cors",
    redirect : "follow",
    cache : "no-cache"
});
```

url属性是第一个实参，并且是必须的。第二个实参是一个对象，有上面列出的其他属性组成。一旦Request对象被赋值给一个变量，那么该变量就可以被用作为`fetch()`方法的参数：

```
fetch(request)
    .then(//处理响应)
    .catch(//处理错误)
```

Headers接口

HTTP头用于传递请求或响应的附加信息。HTTP头中包含的典型信息包括资源的文件类型，cookie信息，身份验证信息以及资源最后修改的时间等等。

Fetch API引入了`Headers`接口，用于创建一个`Headers`对象，该对象可以作

为`Request`和`Response`对象的一个属性。新的`Headers`对象可以使用`Headers()`构造函数来进行创建，如下：

```
let header = new Headers({
    "Content-Type" : "text/plain",
    "Accept-Charset" : "utf-8",
    "Accept-Encoding" : "gzip,deflate"
});
```

在上面的代码中我们利用`Headers()`构造函数来创建了一个`Headers`对象，构造函数中可以将HTTP头信息作为`Headers`对象的属性。

`Headers`对象包含如下的属性和方法，可以用来访问HTTP的头信息：

- has()：用来检测headers对象是否包含作为参数提供的HTTP头。例如：

```
header.has("Content-Type");
```

- get()：返回作为参数提供的HTTP头的值。例如：

```
header.get("Content-Type");
```

- set()：用于设置一个HTTP头的值。例如：

```
header.set("Content-Type","application/json");
```

- append()：给Headers对象添加一个新的HTTP头。例如：

```
header.append("Accept-Encoding","gzip,deflate");
```

- delete()：删除一个HTTP头。例如：

```
header.delete("Accept-Encoding");
```

- keys()、values()和entries()：用来遍历HTTP头，值以及头和值的迭代器。例如：

```
for(let i of header.entries())
{
```

```
    console.log(i);
}
```

15-2-3 接收信息

接下来，让我们将前面所介绍的Response对象，Request对象和Headers对象综合起来，书写一个完整的例子，如下：

```
<body>
    <button id="btn">点击我获取数据</button>
    <div id="test"></div>
    <script>
        //创建headers对象
        let header = new Headers({
            "Content-Type" : "text/plain",
            "Accept-Charset" : "utf-8",
            "Accept-Encoding" : "gzip,deflate"
        });
        //创建request对象
        let request = new Request("./1.php",{headers : header});
        btn.addEventListener("click", function () {
            fetch(request)
                .then(function(response){
                    if(response.ok)
                    {
                        return response;
                    }
                    throw Error(response.statusText);
                })
                .then((response) => response.text())//处理响应
                //动态添加到页面上面
                .then(text => {
                    test.innerText = text;
                })
                .catch(error => console.log("出错了"));
        }, false);
    </script>
</body>
```

15-2-4 发送信息

除了通过Ajax从服务器端获取数据以外，我们也可以利用Ajax向服务器端无刷新地提交数据。信息可以有多种格式，不过通常是一个JSON字符串。示例如下：

1.html

```
<body>
    <input type="text" name="userName" id="userName" placeholder="用户名">
    <input type="text" name="userPwd" id="userPwd" placeholder="密码">
    <button id="btn">提交信息</button>
    <script>
        //给按钮添加点击事件
        btn.addEventListener("click",function(){
            //获取控件里面的数据
            let data = {
                userName : userName.value,
                userPwd : userPwd.value
            };
            data = JSON.stringify(data);//将获取到的数据变为一个字符串
            //构建header 注意类型一定要为application/json
            let header = new Headers({
                "Content-type": "application/json",
                "Accept" : "application/json"
            });
            let req = new Request("./1.php",{
                method : "post",
                headers : header,
                body : data
            })
            fetch(req)
                .then(function(response){
                    if(response.ok)
                    {
                        return response;
                    }
                    throw Error(response.statusText);
                })
                .then(response => response.json())
                .then(data => console.log(data.userName,data.userPwd))
                .catch(error => console.log("出错了"));
        },false);
    </script>
</body>
```

1.php

```
<?php
    //"php://input"可以读取没有处理过的POST数据
    //file_get_contents()用于把文件读入一个字符串
    $data = file_get_contents("php://input");
```

```
    echo $data;
```

在本例中，用户首先通过`1.html`中的表单控件来填写信息，然后通过Fetch API将数据传递到了`1.php`，该服务器端文件在获取到了数据后，再将数据传回给1.html，最后在控制台打印出用户输入的内容。

FormData接口

在上面的例子中，我们向服务器端发送消息时要手动去获取表单里面的数据。在Fetch API中还提供给我们一个FormData的接口，让我们获取表单中的信息变得更加容易。FormData对象也可以通过构造函数来进行创建，如下：

```
let data = new FormData();
```

FormData对象，会自动序列化所有数据，然后使用Ajax发送。示例如下：

```

```

我们还可以使用`append()`方法，将数据作为键值对添加到FormData对象里面，如下：

```

```

总结

1. Ajax并不是一门崭新的技术，而是当时所存在的几种技术的结合。
2. 使用原生Ajax的步骤大致可以分为：创建XMLHttpRequest对象，发出HTTP请求，接收服务器传回的数据和更新网页的数据。
3. 根据浏览器的不同，创建XMLHttpRequest对象的方式也不同。
4. 发送请求时要用到open方法和send方法。
5. readyState属性有5种状态值，一般我们只对值为4的时候采取相应的操作。
6. Fetch API是当前通过网络异步请求和发送数据的现存标准。
7. Fetch API中提供了诸如Response，Request以及Headers等接口来简化许多在使用XMLHttpRequest对象时变得麻烦的概念。

15-3 Mock.js

在我们的实际生产中，后端的接口往往是较晚才会出来，并且还要写接口文档，于是我们的前端的许多开发都要等到接口给我们才能进行，这样对于我们前端来说显得十分的被动，于是有没有可以制造假数据来模拟后端接口呢，答案是肯定的。今天我们来介绍一款非常强大的插件Mock.js，可以非常方便的模拟后端的数据，也可以轻松的实现增删改查这些操作。

Mock.js拥有以下的特点：

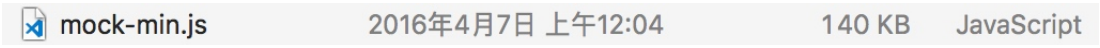
- 前后端分离：让前端工程师独立于后端进行开发。
- 开发无侵入：不需要修改既有的代码，就可以拦截Ajax请求，返回模拟的响应数据。
- 数据类型丰富：支持生成随机的文本、数字、布尔值、日期、邮箱、链接、图片和颜色等。
- 增加单元测试的真实性：通过随机数据，模拟各种场景。
- 用法简单：符合直觉的接口。
- 方便扩展：支持扩展更多数据类型，支持自定义函数和正则。

安装

最简单的方式就是使用npm进行安装，如下：

```
npm install mockjs
```

但是我安装老是出问题，所以这里我选择把它下载下来。下载的是压缩后的文件，大小也就140KB



语法规范

Mock.js的语法规范包括两个部分：

- 数据模板定义规范(Data Template Definition，DTD)
- 数据占位符定义规范(Data Placeholder Definition，DPD)

数据模板定义规范 DTD

数据模板中的每个属性由 3 部分构成：属性名、生成规则、属性值：

```
// 属性名   name
```

```
// 生成规则 rule
// 属性值   value
'name|rule': value
```

注意：

属性名和生成规则 之间用竖线 `|` 分隔。生成规则是可选的。生成规则有 7 种格式：

- 'name|min-max': value
- 'name|count': value
- 'name|min-max.dmin-dmax': value
- 'name|min-max.dcount': value
- 'name|count.dmin-dmax': value
- 'name|count.dcount': value
- 'name|+step': value

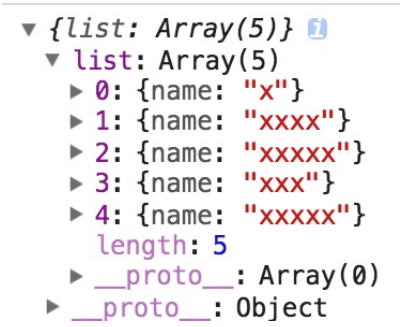
生成规则的含义需要依赖属性值的类型才能确定。属性值中可以含有 `@` 占位符。属性值还指定了最终值的初始值和类型。生成规则和示例：

1. 属性值是字符串 String

'name|min-max': string
通过重复 string 生成一个字符串，重复次数大于等于 min，小于等于 max。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        // 生成5个x长度为1-5的数据
        let data = Mock.mock({
            'list|5':[{'name|1-5': 'x'}],
        });
        console.log(data);
    </script>
</body>
```

效果：



'name|count': string
通过重复 string 生成一个字符串，重复次数等于 count。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        // name为属性名
        // x为属性值
        // 1-5为重复次数
        let data = Mock.mock({
            'name|3': 'x'
        });
        console.log(data); // {name: "xxx"}
    </script>
</body>
```

2. 属性值是数字 Number

'name|+1': number
属性值自动加 1，初始值为 number。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        // 生成5个数据
        let data = Mock.mock({
            'list|5':[{'age|+1': 20}],
        });
        console.log(data);
    </script>
</body>
```

效果：

```
▼ {list: Array(5)}
  ▼ list: Array(5)
    ▶ 0: {age: 20}
    ▶ 1: {age: 21}
    ▶ 2: {age: 22}
    ▶ 3: {age: 23}
    ▶ 4: {age: 24}
      length: 5
    ▶ __proto__: Array(0)
  ▶ __proto__: Object
```

'name|min-max': number
生成一个大于等于 min、小于等于 max 的整数，属性值 number 只是用来确定类型。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        // 如果是字符串类型，那么值就为1-5个'1'
        let data = Mock.mock({
            'list|5':[{'age|1-5': '1'}],
        });
        // 如果是数值类型，那么值就1-5之间
        let data2 = Mock.mock({
            'list|5':[{'age|1-5': 2}],
        });
        console.log(data);
        console.log(data2);
    </script>
</body>
```

效果：

```
▼ {list: Array(5)}
  ▼ list: Array(5)
    ▶ 0: {age: "11111"}
    ▶ 1: {age: "1111"}
    ▶ 2: {age: "11"}
    ▶ 3: {age: "11111"}
    ▶ 4: {age: "111"}
      length: 5
    ▶ __proto__: Array(0)
  ▶ __proto__: Object
▼ {list: Array(5)}
  ▼ list: Array(5)
    ▶ 0: {age: 3}
    ▶ 1: {age: 1}
    ▶ 2: {age: 3}
    ▶ 3: {age: 2}
    ▶ 4: {age: 4}
      length: 5
    ▶ __proto__: Array(0)
  ▶ __proto__: Object
```

'name|min-max.dmin-dmax': number

生成一个浮点数，整数部分大于等于 min、小于等于 max，小数部分保留 dmin 到 dmax 位。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let data = Mock.mock({
            'number1|1-100.1-10': 1,
            'number2|123.1-10': 1,
            'number3|123.3': 1,
            'number4|123.10': 1.123
        });
        console.log(data);
    </script>
</body>
```

效果：

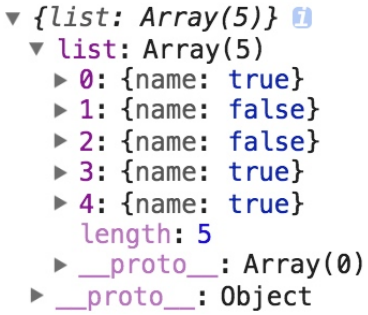
```
▼ {number1: 1.8391537, number2: 123.730576, number3: 123.181, number4: 123.1238897713}
    number1: 1.8391537
    number2: 123.730576
    number3: 123.181
    number4: 123.1238897713
  ▶ __proto__: Object
```

3. 属性值是布尔型 Boolean

'name|1': boolean
随机生成一个布尔值，值为 true 的概率是 1/2，值为 false 的概率同样是 1/2。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        // name右边的1是指概率1/2
        // 这个代码表示生成true1/2  非true1/2
        let data = Mock.mock({
            'list|5':[{'name|1': true}],
        });
        console.log(data);
    </script>
</body>
```

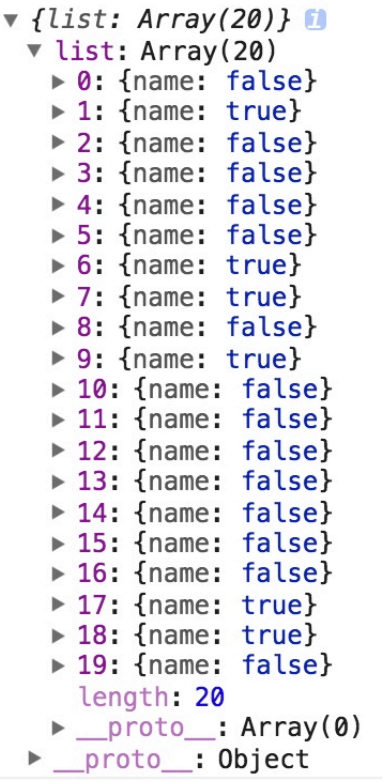
效果：



'name|min-max': value
随机生成一个布尔值，值为 value 的概率是 min / (min + max)，值为 !value 的概率是 max / (min + max)。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        // 生成true的机率是 1/4
        // 生成非true的机率 3/4
        let data = Mock.mock({
            'list|20':[{'name|1-3': true}],
        });
        console.log(data);
    </script>
</body>
```

效果：

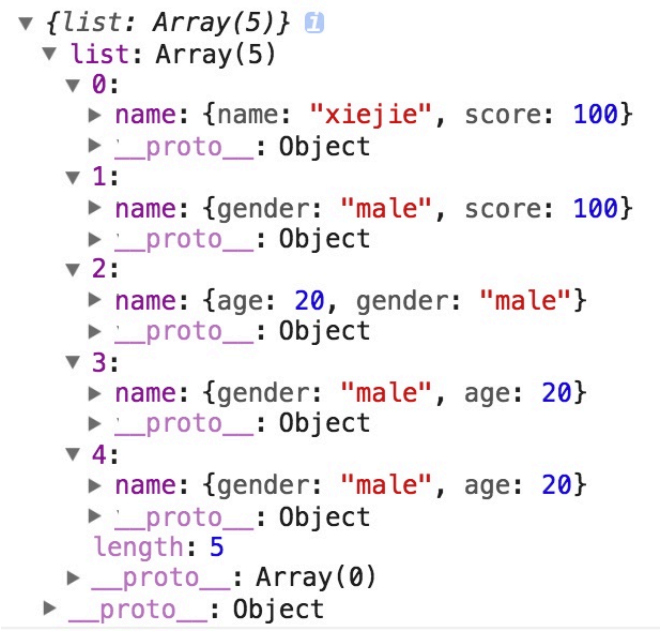


4. 属性值是对象 Object

'name|count': object
从属性值 object 中随机选取 count 个属性。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let data = Mock.mock({
            'list|5':[{'name|2': {'name':'xiejie','age':20,'gender':'male','
score':100}}],
        });
        console.log(data);
    </script>
</body>
```

效果：



'name|min-max': object
从属性值 object 中随机选取 min 到 max 个属性。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let data = Mock.mock({
            'name1|1-3': {'name':'xiejie','age':20,'gender':'male','score':1
00},
            'name2|1-3': {'name':'xiejie','age':20,'gender':'male','score':1
00},
```

```
            'name3|1-3': {'name':'xiejie','age':20,'gender':'male','score':1
00},
        });
        console.log(data);
    </script>
</body>
```

效果：

```
▼ {name1: {…}, name2: {…}, name3: {…}}
  ▶ name1: {gender: "male", name: "xiejie"}
  ▶ name2: {age: 20, gender: "male", name: "xiejie"}
  ▶ name3: {score: 100, age: 20}
  ▶ __proto__: Object
```

5. 属性值是数组 Array

'name|1': array

从属性值 array 中随机选取 1 个元素，作为最终值。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let data = Mock.mock({
            'list|5':[{'name1|1': [1,2,3,4,5,6,7,8,9,10]}]
        });
        console.log(data);
    </script>
</body>
```

效果：

```
▼ {list: Array(5)}
  ▼ list: Array(5)
    ▶ 0: {name1: 5}
    ▶ 1: {name1: 2}
    ▶ 2: {name1: 9}
    ▶ 3: {name1: 3}
    ▶ 4: {name1: 5}
      length: 5
    ▶ __proto__: Array(0)
  ▶ __proto__: Object
```

'name|+1': array

从属性值 array 中顺序选取 1 个元素，作为最终值。

```
<body>
    <script src="./mock-min.js"></script>
```

```
    <script>
        // 从数组第一位开始，然后每隔4个选取一个元素
        let data = Mock.mock({
            'list|5':[{'name1|+4': [1,2,3,4,5,6,7,8,9,10]}]
        });
        console.log(data);
    </script>
</body>
```

效果：

```
▼ {list: Array(5)} i
  ▼ list: Array(5)
    ▶ 0: {name1: 1}
    ▶ 1: {name1: 5}
    ▶ 2: {name1: 9}
    ▶ 3: {name1: 3}
    ▶ 4: {name1: 7}
      length: 5
    ▶ __proto__: Array(0)
  ▶ __proto__: Object
```

'name|min-max': array

通过重复属性值 array 生成一个新数组，重复次数大于等于 min，小于等于 max。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        // 将后面的数组重复1-3次
        let data = Mock.mock({
            'list|3':[{'name1|1-3': [1,2,3,4,5,6,7,8,9,10]}]
        });
        console.log(data);
    </script>
</body>
```

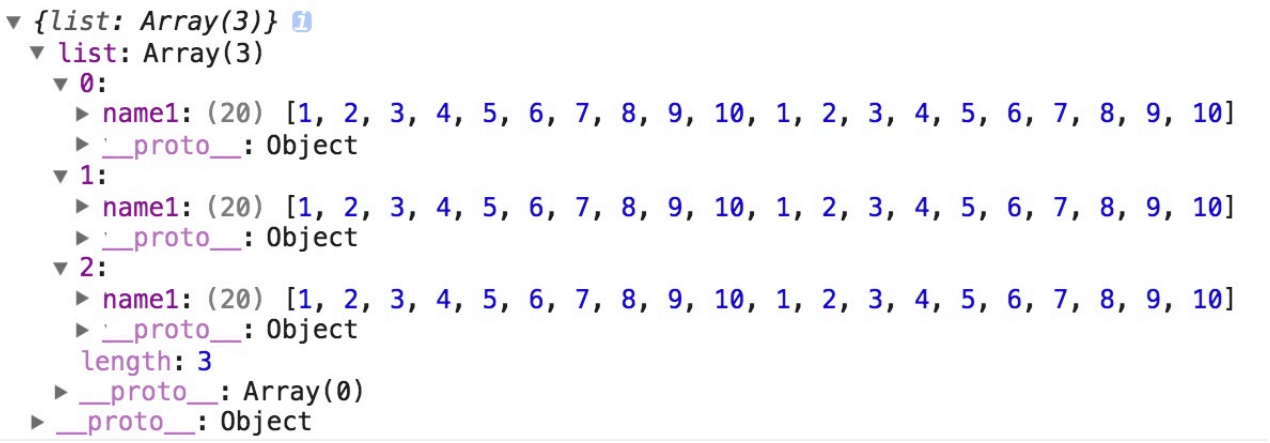
效果：

```
▼ {list: Array(3)} i
  ▼ list: Array(3)
    ▼ 0:
      ▶ name1: (30) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      ▶ __proto__: Object
    ▼ 1:
      ▶ name1: (20) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      ▶ __proto__: Object
    ▼ 2:
      ▶ name1: (10) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      ▶ __proto__: Object
      length: 3
    ▶ __proto__: Array(0)
  ▶ __proto__: Object
```

'name|count': array
通过重复属性值 array 生成一个新数组，重复次数为 count。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        // 将后面的数组重复2次
        let data = Mock.mock({
            'list|3':[{'name1|2': [1,2,3,4,5,6,7,8,9,10]}]
        });
        console.log(data);
    </script>
</body>
```

效果：

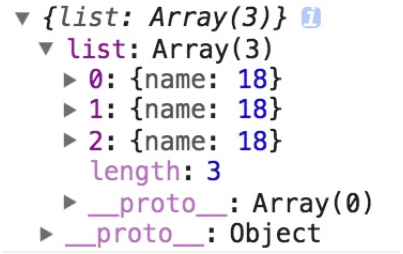


6. 属性值是函数 Function

'name': function
执行函数 function，取其返回值作为最终的属性值，函数的上下文为属性 'name' 所在的对象。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        // 将函数的返回值作为属性值
        let data = Mock.mock({
            'list|3':[{'name': function(){
                return 18;
            }}]
        });
        console.log(data);
    </script>
</body>
```

效果：

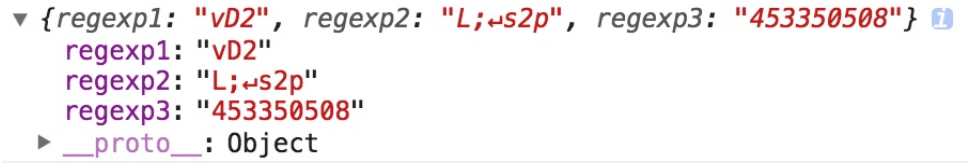


7. 属性值是正则表达式 RegExp

'name': regexp
根据正则表达式 regexp 反向生成可以匹配它的字符串。用于生成自定义格式的字符串。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let data = Mock.mock({
            'regexp1': /[a-z][A-Z][0-9]/,
            'regexp2': /\w\W\s\S\d\D/,
            'regexp3': /\d{5,10}/
        });
        console.log(data);
    </script>
</body>
```

效果：



数据占位符定义规范 DPD

占位符只是在属性值字符串中占个位置，并不出现在最终的属性值中。

占位符 的格式为：

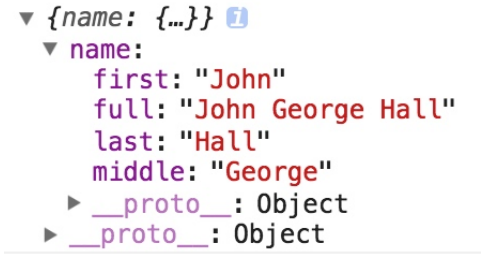
```
@占位符
@占位符(参数 [, 参数])
```

注意：

- 用 `@` 来标识其后的字符串是占位符。
- 占位符引用的是 `Mock.Random` 中的方法。
- 通过 `Mock.Random.extend()` 来扩展自定义占位符。
- 占位符也可以引用数据模板中的属性。
- 占位符会优先引用数据模板中的属性。
- 占位符支持相对路径和绝对路径。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        // 占位符引用的是 Mock.Random 中的方法
        let data = Mock.mock({
            name: {
                first: '@FIRST',
                middle: '@FIRST',
                last: '@LAST',
                full: '@first @middle @last'
            }
        });
        console.log(data);
    </script>
</body>
```

效果：



Mock.mock()

对于这个方法，我们已经不再陌生了。上面的例子中，我们都是使用的 `Mock.mock()` 来生成的数据。该方法还可以根据模板生成模拟数据，看下面的例子：

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let template = {
            'id|+1':1,
            'name' : '@cname',
```

```
            'age|20-30' : 1,
            'gender' : /[男女]/,
            'score|60-100' : 1,
            'isMarry|1' : true
        }
        let data = Mock.mock({
            'list|5' : [template]
        });
        console.log(data);
    </script>
</body>
```

效果：其实就是里面的对象单独提取了出来而已

```
▼ {list: Array(5)}
  ▼ list: Array(5)
    ▶ 0: {id: 1, name: "张洋", age: 29, gender: "女", score: 83, …}
    ▶ 1: {id: 2, name: "周涛", age: 20, gender: "男", score: 81, …}
    ▶ 2: {id: 3, name: "赵桂英", age: 27, gender: "女", score: 68, …}
    ▶ 3: {id: 4, name: "易刚", age: 23, gender: "女", score: 94, …}
    ▶ 4: {id: 5, name: "冯军", age: 25, gender: "女", score: 79, …}
      length: 5
    ▶ __proto__: Array(0)
  ▶ __proto__: Object
```

接下来重头戏来了，使用`Mock.mock()`可以拦截Ajax请求，语法如下：

```
Mock.mock( rurl?, rtype?, template|function( options ) )
```

各参数的含义如下：

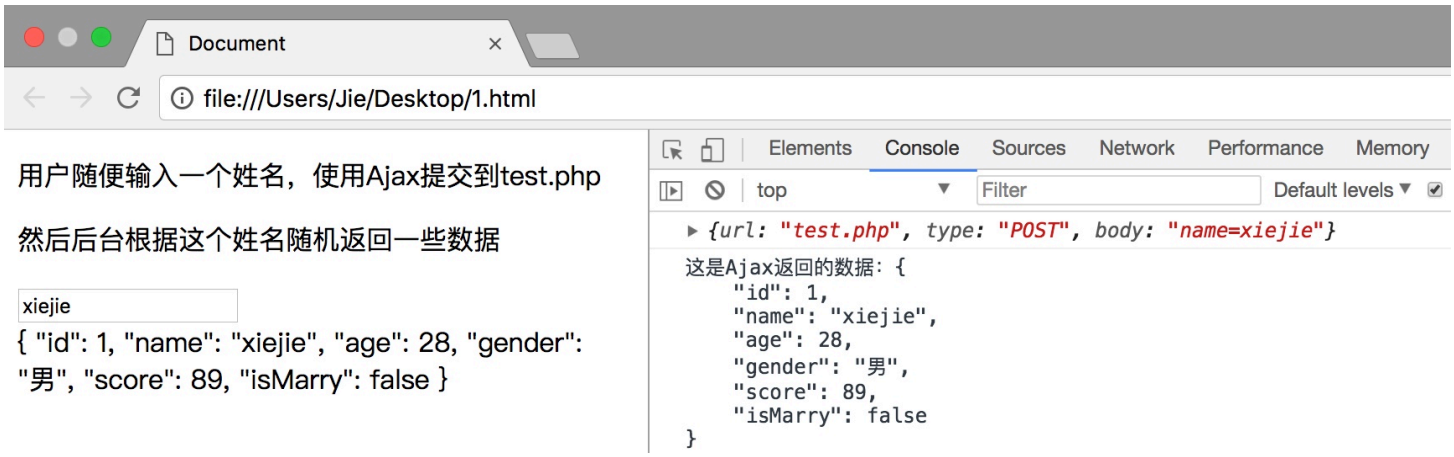
- rurl：可选。表示需要拦截的 URL，可以是 URL 字符串或 URL 正则。例如`/\/domain\/list\.json/`、`'/domian/list.json'`。
- rtype：可选。表示需要拦截的 Ajax 请求类型。例如 GET、POST、PUT、DELETE 等。
- template：可选。表示数据模板，可以是对象或字符串。例如`{ 'data|1-10':[{}] }`、`'@EMAIL'`。
- function(options)：可选。表示用于生成响应数据的函数。
- options：指向本次请求的 Ajax 选项集，含有 url、type 和 body 三个属性

来看一个具体的示例，如下：

```
<body>
    <p>用户随便输入一个姓名，使用Ajax提交到test.php</p>
    <p>然后后台根据这个姓名随机返回一些数据</p>
    <input type="text" name="" id="" placeholder="随便输入点东西">
    <div id="box"></div>
    <script src="./jquery-1.12.4.min.js"></script>
```

```
    <script src="./mock-min.js"></script>
    <script>
        // 当文本框失去焦点时将文本框里面的内容发送到test.php
        $('input').blur(function(){
            let info = $('input').val();
            $.ajax({
                type:'POST', //这里可以换成 GET
                url:'test.php',
                data:{ name : info },
                success:function(response,stutas,xhr){
                    console.log(`这是Ajax返回的数据：${response}`);
                    $('#box').html(response);
                }
            });
        });
        let template = {
            'id|+1':1,
            'name':'@cname',
            'age|20-30' : 1,
            'gender' : /[男女]/,
            'score|60-100' : 1,
            'isMarry|1' : true
        }
        let data = Mock.mock(/\.php/,'post',function(option){
            // 拦截下来的Ajax请求如下
            console.log(option);
            // 讲option.body提取出来，只取=后面的部分
            let arr = option.body.split('=');
            // 随机返回一些数据
            return Mock.mock({
                'id|+1':1,
                'name': arr[1],
                'age|20-30' : 1,
                'gender' : /[男女]/,
                'score|60-100' : 1,
                'isMarry|1' : true
            });
        });
    </script>
</body>
```

渲染出来的效果如下：



不同的组合方式所代表的不同含义具体如下：

- `Mock.mock( template )`：根据数据模板生成模拟数据。
- `Mock.mock( rurl, template )`：记录数据模板。当拦截到匹配 `rurl` 的 Ajax 请求时，将根据数据模板 `template` 生成模拟数据，并作为响应数据返回。
- `Mock.mock( rurl, function( options ) )`：记录用于生成响应数据的函数。当拦截到匹配 `rurl` 的 Ajax 请求时，函数 `function(options)` 将被执行，并把执行结果作为响应数据返回。
- `Mock.mock( rurl, rtype, template )`：记录数据模板。当拦截到匹配 `rurl` 和 `rtype` 的 Ajax 请求时，将根据数据模板 template 生成模拟数据，并作为响应数据返回。
- `Mock.mock( rurl, rtype, function( options ) )`：记录用于生成响应数据的函数。当拦截到匹配 `rurl` 和 `rtype` 的 Ajax 请求时，函数 `function(options)` 将被执行，并把执行结果作为响应数据返回。

基本大同小异，这里不再做具体示例。

Mock.set()

`Mock.setup( settings )`：配置拦截 Ajax 请求时的行为。支持的配置项有：timeout。

- settings：必选。配置项集合。
- timeout：可选。指定被拦截的 Ajax 请求的响应时间，单位是毫秒。值可以是正整数，例如 400，表示 400 毫秒 后才会返回响应内容；也可以是横杠 '-' 风格的字符串，例如 '200-600'，表示响应时间介于 200 和 600 毫秒之间。默认值是'10-100'。

```
Mock.setup({
    timeout: 400
```

```
})
Mock.setup({
    timeout: '200-600'
})
```

> 目前，接口`Mock.setup( settings )`仅用于配置 Ajax 请求，将来可能用于配置 Mock 的其他行为。

Mock.Random

`Mock.Random`是一个工具类，用于生成各种随机数据。`Mock.Random`的方法在数据模板中称为`占位符`，书写格式为`@占位符(参数 [, 参数])`。来看一个具体的示例：

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        var Random = Mock.Random
        console.log(Random.email()); // t.wwqn@eghfyprjm.bj

        console.log(Mock.mock('@email')); // n.mpkotnsf@wqrohq.az

        console.log(Mock.mock({ email: '@email' })); // {email: "j.sxnjliq@k
ofmbqnwt.tf"}
    </script>
</body>
```

方法：

`Mock.Random`提供的完整方法（占位符）如下：

Type	Method
Basic	boolean, natural, integer, float, character, string, range, date, time, datetim e, now
Image	image, dataImage
Color	color
Text	paragraph, sentence, word, title, cparagraph, csentence, cword, ctitle
Name	first, last, name, cfirst, clast, cname
Web	url, domain, email, ip, tld

Address	area, region
Helper	capitalize, upper, lower, pick, shuffle
Miscellaneous	guid, id

> 后面会有具体的示例

扩展

`Mock.Random`中的方法与数据模板的`@占位符`一一对应，在需要时还可以为`Mock.Random`扩展方法，然后在数据模板中通过`@扩展方法`引用。例如：

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        Random.extend({
            constellation: function (date) {
                var constellations = [
                    '白羊座', '金牛座', '双子座', '巨蟹座', '狮子座',
                    '处女座', '天秤座', '天蝎座', '射手座', '摩羯座',
                    '水瓶座', '双鱼座'
                ];
                return this.pick(constellations);
            }
        })
        console.log(Random.constellation());
        console.log(Mock.mock('@CONSTELLATION'));
        console.log(Mock.mock({
            constellation: '@CONSTELLATION'
        }));
    </script>
</body>
```

效果：

```
水瓶座
巨蟹座
▶ {constellation: "双鱼座"}
```

Random方法详解

Basic

Random.boolean(min?, max?, current?)

- Random.boolean()
- Random.boolean(min, max, current)

返回一个随机的布尔值。

各参数说明如下：

- min：可选。指示参数 current 出现的概率。概率计算公式为 `min / (min + max)`。该参数的默认值为 1，即有 50% 的概率返回参数 current。
- max：可选。指示参数 current 的相反值 `!current` 出现的概率。概率计算公式为 `max / (min + max)`。该参数的默认值为 1，即有 50% 的概率返回参数 `!current`。
- current：可选。可选值为布尔值 true 或 false。如果未传入任何参数，则返回 true 和 false 的概率各为 50%。该参数没有默认值。在该方法的内部，依据原生方法 `Math.random()` 返回的（浮点）数来计算和返回布尔值，例如在最简单的情况下，返回值是表达式 `Math.random() >= 0.5` 的执行结果。

来看一个具体的示例，如下：

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.boolean()); // true
        console.log(Random.boolean(1, 9, true)); // false
        console.log(Random.bool()); // true
        console.log(Random.bool(1, 9, false)); // true
    </script>
</body>
```

Random.natural(min?, max?)

- Random.natural()
- Random.natural(min)
- Random.natural(min, max)

返回一个随机的自然数（大于等于 0 的整数）。

各参数说明如下：

- min：可选。指示随机自然数的最小值。默认值为 0。
- max：可选。指示随机自然数的最大值。默认值为 9007199254740992。

来看一个具体的示例，如下：

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.natural()); // 2723769199764888
        console.log(Random.natural(10000)); // 2637724598982232
        console.log(Random.natural(60, 100)); // 95
    </script>
</body>
```

Random.integer(min?, max?)

- Random.integer()
- Random.integer(min)
- Random.integer(min, max)

返回一个随机的整数。

各参数说明如下：

- min：可选。指示随机整数的最小值。默认值为 -9007199254740992。
- max：可选。指示随机整数的最大值。默认值为 9007199254740992。

具体示例如下：

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.integer()); // 599659465188144
        console.log(Random.integer(10000)); // 1871808516812238
        console.log(Random.integer(60, 100)); // 77
    </script>
</body>
```

Random.float(min?, max?, dmin?, dmax?)

- Random.float()

- Random.float(min)
- Random.float(min, max)
- Random.float(min, max, dmin)
- Random.float(min, max, dmin, dmax)

返回一个随机的浮点数。

各个参数说明如下：

- min：可选。整数部分的最小值。默认值为 -9007199254740992。
- max：可选。整数部分的最大值。默认值为 9007199254740992。
- dmin：可选。小数部分位数的最小值。默认值为 0。
- dmin：可选。小数部分位数的最大值。默认值为 17。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.float()); // 2011032664849092.2
        console.log(Random.float(0)); // 4604281202306217
        console.log(Random.float(60, 100)); // 77.313882
        console.log(Random.float(60, 100, 3)); // 68.54644312712652
        console.log(Random.float(60, 100, 3, 5)); // 92.349
    </script>
</body>
```

Random.character(pool?)

- Random.character()
- Random.character('lower/upper/number/symbol')
- Random.character(pool)

返回一个随机字符。

各参数说明如下：

- pool：可选。字符串。表示字符池，将从中选择一个字符返回。如果传入了 'lower' 或 'upper'、'number'、'symbol'，表示从内置的字符池从选取：

```
{
    lower: "abcdefghijklmnopqrstuvwxyz",
    upper: "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
    number: "0123456789",
```

```
    symbol: "!@#$%^&*()[]"
}
```

如果未传入该参数，则从 `lower + upper + number + symbol` 中随机选取一个字符返回。

具体示例如下：

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.character()); // 8
        console.log(Random.character('lower')); // u
        console.log(Random.character('upper')); // E
        console.log(Random.character('number')); // 3
        console.log(Random.character('symbol')); // [
        console.log(Random.character('aeiou')); // e
    </script>
</body>
```

Random.string(pool?, min?, max?)

- Random.string()
- Random.string(length)
- Random.string(pool, length)
- Random.string(min, max)
- Random.string(pool, min, max)

返回一个随机字符串。

各参数说明如下：

- pool：可选。字符串。表示字符池，将从中选择一个字符返回。如果传入 'lower' 或 'upper'、'number'、'symbol'，表示从内置的字符池从选取：

```
{
    lower: "abcdefghijklmnopqrstuvwxyz",
    upper: "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
    number: "0123456789",
    symbol: "!@#$%^&*()[]"
}
```

如果未传入该参数，则从 lower + upper + number + symbol 中随机选取一个字符返回。

- min：可选。随机字符串的最小长度。默认值为 3。
- max：可选。随机字符串的最大长度。默认值为 7。

具体示例如下：

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.string()); // ER0U37
        console.log(Random.string(5)); // gz76P
        console.log(Random.string('lower', 5)); // glpfb
        console.log(Random.string(7, 10)); // 3B3K4r)u3
        console.log(Random.string('aeiou', 1, 3)); // oi
        console.log(Random.string('壹贰叁肆伍陆柒捌玖拾', 3, 5)); // 陆贰叁
    </script>
</body>
```

Random.range(start?, stop, step?)

- Random.range(stop)
- Random.range(start, stop)
- Random.range(start, stop, step)

返回一个整型数组。

各个参数说明如下：

- start：必选。数组中整数的起始值。
- stop：可选。数组中整数的结束值（不包含在返回值中）。
- step：可选。数组中整数之间的步长。默认值为 1。

具体示例如下：

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.range(10)); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        console.log(Random.range(3, 7)); // [3, 4, 5, 6]
        console.log(Random.range(1, 10, 2)); // [1, 3, 5, 7, 9]
        console.log(Random.range(1, 10, 3)); // [1, 4, 7]
    </script>
</body>
```

Date

Random.date(format?)

- Random.date()
- Random.date(format)

返回一个随机的日期字符串。

参数说明如下：

- format：可选。指示生成的日期字符串的格式。默认值为 `yyyy-MM-dd` 。

可选的占位符参考自 `Ext.Date` ，如下所示：

Format	Description	Example
yyyy	A full numeric representation of a year, 4 digits	1999 or 2003
yy	A two digit representation of a year	99 or 03
y	A two digit representation of a year	99 or 03
MM	Numeric representation of a month, with leading zeros	01 to 12
M	Numeric representation of a month, without leading zeros	1 to 12
dd	Day of the month, 2 digits with leading zeros	01 to 31
d	Day of the month without leading zeros	1 to 31
HH	24-hour format of an hour with leading zeros	00 to 23
H	24-hour format of an hour without leading zeros	0 to 23
hh	12-hour format of an hour without leading zeros	1 to 12
h	12-hour format of an hour with leading zeros	01 to 12
mm	Minutes, with leading zeros	00 to 59
m	Minutes, without leading zeros	0 to 59
ss	Seconds, with leading zeros	00 to 59
s	Seconds, without leading zeros	0 to 59
SS	Milliseconds, with leading zeros	000 to 999

S	Milliseconds, without leading zeros	0 to 999
A	Uppercase Ante meridiem and Post meridiem	AM or PM
a	Lowercase Ante meridiem and Post meridiem	am or pm
T	Milliseconds, since 1970-1-1 00:00:00 UTC	759883437303

接下来我们来看一下具体的示例，如下：

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.date()); // 1993-04-04
        console.log(Random.date('yyyy-MM-dd')); // 2015-11-21
        console.log(Random.date('yy-MM-dd')); // 00-12-13
        console.log(Random.date('y-MM-dd')); // 78-03-08
        console.log(Random.date('y-M-d')); // 79-10-2
    </script>
</body>
```

Random.time(format?)

- Random.time()
- Random.time(format)

返回一个随机的时间字符串。

各参数代表的意思如下：

- format：可选。指示生成的时间字符串的格式。默认值为 HH:mm:ss。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.time()); // 07:49:26
        console.log(Random.time('A HH:mm:ss')); // AM 06:38:51
        console.log(Random.time('a HH:mm:ss')); // am 08:54:21
        console.log(Random.time('HH:mm:ss')); // 08:55:51
        console.log(Random.time('H:m:s')); // 11:59:6
    </script>
</body>
```

Random.datetime(format?)

- Random.datetime()
- Random.datetime(format)

返回一个随机的日期和时间字符串。

参数说明：

- format：可选。指示生成的日期和时间字符串的格式。默认值为 `yyyy-MM-dd HH:mm:ss` 。

具体示例如下：

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.datetime()); // 1987-11-17 16:50:12
        console.log(Random.datetime('yyyy-MM-dd A HH:mm:ss')); // 1990-10-27
 PM 23:09:24
        console.log(Random.datetime('yy-MM-dd a HH:mm:ss')); // 72-12-02 pm
20:36:10
        console.log(Random.datetime('y-MM-dd HH:mm:ss')); // 96-05-29 15:30:
35
        console.log(Random.datetime('y-M-d H:m:s')); // 85-12-14 6:5:15
    </script>
</body>
```

Random.now(unit?, format?)

- Ranndom.now(unit, format)
- Ranndom.now()
- Ranndom.now(format)
- Ranndom.now(unit)

返回当前的日期和时间字符串。

各参数说明如下：

- unit：可选。表示时间单位，用于对当前日期和时间进行格式化。可选值有：year、month、week、day、hour、minute、second、week，默认不会格式化。
- format：可选。指示生成的日期和时间字符串的格式。默认值为 yyyy-MM-dd HH:mm:ss。

具体示例如下：

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.now()); // 2018-04-19 13:38:50
        console.log(Random.now('day', 'yyyy-MM-dd HH:mm:ss SS')); // 2018-04
-19 00:00:00 000
        console.log(Random.now('day')); // 2018-04-19 00:00:00
        console.log(Random.now('yyyy-MM-dd HH:mm:ss SS')); // 2018-04-19 13:
38:50 934
        console.log(Random.now('year')); // 2018-01-01 00:00:00
        console.log(Random.now('month')); // 2018-04-01 00:00:00
        console.log(Random.now('week')); // 2018-04-15 00:00:00
        console.log(Random.now('day')); // 2018-04-19 00:00:00
        console.log(Random.now('hour')); // 2018-04-19 13:00:00
        console.log(Random.now('minute')); // 2018-04-19 13:38:00
        console.log(Random.now('second')); // 2018-04-19 13:38:50
    </script>
</body>
```

Image

Random.image(size?, background?, foreground?, format?, text?)

- Random.image()
- Random.image(size)
- Random.image(size, background)
- Random.image(size, background, text)
- Random.image(size, background, foreground, text)
- Random.image(size, background, foreground, format, text)

生成一个随机的图片地址。

> Random.image() 用于生成高度自定义的图片地址，一般情况下，应该使用更简单的 Random.dataImage()。

参数说明：

- size：可选。指示图片的宽高，格式为 '宽x高'。默认从下面的数组中随机读取一个：

```
[
    '300x250', '250x250', '240x400', '336x280',
    '180x150', '720x300', '468x60', '234x60',
    '88x31', '120x90', '120x60', '120x240',
```

```
    '125x125', '728x90', '160x600', '120x600',
    '300x600'
]
```

- background：可选。指示图片的背景色。默认值为 '#000000'。
- foreground：可选。指示图片的前景色（文字）。默认值为 '#FFFFFF'。
- format：可选。指示图片的格式。默认值为 'png'，可选值包括：'png'、'gif'、'jpg'。
- text：可选。指示图片上的文字。默认值为参数 size。

```
Random.image()
// => "http://dummyimage.com/125x125"
Random.image('200x100')
// => "http://dummyimage.com/200x100"
Random.image('200x100', '#fb0a2a')
// => "http://dummyimage.com/200x100/fb0a2a"
Random.image('200x100', '#02adea', 'Hello')
// => "http://dummyimage.com/200x100/02adea&text=Hello"
Random.image('200x100', '#00405d', '#FFF', 'Mock.js')
// => "http://dummyimage.com/200x100/00405d/FFF&text=Mock.js"
Random.image('200x100', '#ffcc33', '#FFF', 'png', '!')
// => "http://dummyimage.com/200x100/ffcc33/FFF.png&text=!"
```

Random.dataImage(size?, text?)

- Random.dataImage()
- Random.dataImage(size)
- Random.dataImage(size, text)

生成一段随机的 Base64 图片编码。

> 如果需要生成高度自定义的图片，请使用 Random.image()。

参数说明：

- size：可选。指示图片的宽高，格式为 '宽x高'。默认从下面的数组中随机读取一个：

```
[
    '300x250', '250x250', '240x400', '336x280',
    '180x150', '720x300', '468x60', '234x60',
    '88x31', '120x90', '120x60', '120x240',
    '125x125', '728x90', '160x600', '120x600',
    '300x600'
]
```

- text：可选。指示图片上的文字。默认值为参数 size。

```
Random.dataImage()
Random.dataImage('200x100')
Random.dataImage('200x100', 'Hello Mock.js!')
```

Color

Random.color()

- Random.color()：随机生成一个有吸引力的颜色，格式为 '#RRGGBB'。示例如下：

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.color()); // #799af2
    </script>
</body>
```

Random.hex()

- Random.hex()：随机生成一个有吸引力的颜色，格式为 '#RRGGBB'。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.hex()); // #79f2d8
    </script>
</body>
```

Random.rgb()

- Random.rgb()：随机生成一个有吸引力的颜色，格式为 'rgb(r, g, b)'。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.rgb()); // rgb(176, 242, 121)
    </script>
```

```
</body>
```

Random.rgba()

- Random.rgba()：随机生成一个有吸引力的颜色，格式为 'rgba(r, g, b, a)'。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.rgba()); // rgba(242, 231, 121, 0.50)
    </script>
</body>
```

Random.hsl()

- Random.hsl()：随机生成一个有吸引力的颜色，格式为 'hsl(h, s, l)'。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.hsl()); // hsl(265, 82, 71)
    </script>
</body>
```

Text

Random.paragraph(min?, max?)

- Random.paragraph()
- Random.paragraph(len)
- Random.paragraph(min, max)

随机生成一段文本。

参数说明：

- len：可选。指示文本中句子的个数。默认值为 3 到 7 之间的随机数。
- min：可选。指示文本中句子的最小个数。默认值为 3。
- max：可选。指示文本中句子的最大个数。默认值为 7。

```
Random.paragraph()
Random.paragraph(2)
Random.paragraph(1, 3)
```

Random.cparagraph(min?, max?)

- Random.cparagraph()
- Random.cparagraph(len)
- Random.cparagraph(min, max)

随机生成一段中文文本。

> 参数的含义和默认值同 Random.paragraph(min?, max?)

```
Random.cparagraph()
// => "给日数时化周作少情者美制论。到先争劳今已美变江以好较正新深。族国般建难出就金感基酸转。任部四那响成族利标铁导术一或已于。省元切世权往着路积会其区素白思断。加把他位间存定国工取除许热规先法方。"
Random.cparagraph(2)
// => "去话起时为无子议气根复即传月广。题林里油步不约认山形两标命导社干。"
Random.cparagraph(1, 3)
// => "候无部社心性有构员其深例矿取民为。须被亲需报每完认支这明复几下在铁需连。省备可离展五斗器就石正队除解动。"
```

Random.sentence(min?, max?)

- Random.sentence()
- Random.sentence(len)
- Random.sentence(min, max)

随机生成一个句子，第一个单词的首字母大写。

参数说明：

- len：可选。指示句子中单词的个数。默认值为 12 到 18 之间的随机数。
- min：可选。指示句子中单词的最小个数。默认值为 12。
- max：可选。指示句子中单词的最大个数。默认值为 18。

```
Random.sentence()
// => "Jovasojt qopupwh plciewh dryir zsqsvlkga yeam."
Random.sentence(5)
// => "Fwlymyyw htccsrgdk rgemfpyt cffydvvpc ycgvno."
Random.sentence(3, 5)
```

```
// => "Mgl qhrprwkhb etvwfbixm jbqmg."
```

Random.csentence(min?, max?)

- Random.csentence()
- Random.csentence(len)
- Random.csentence(min, max)

随机生成一段中文文本。

> 参数的含义和默认值同 Random.sentence(min?, max?)

```
Random.csentence()
// => "第任人九同段形位第律认得。"
Random.csentence(2)
// => "维总。"
Random.csentence(1, 3)
// => "厂存。"
```

Random.word(min?, max?)

- Random.word()
- Random.word(len)
- Random.word(min, max)

随机生成一个单词。

参数说明：

- len：可选。指示单词中字符的个数。默认值为 3 到 10 之间的随机数。
- min：可选。指示单词中字符的最小个数。默认值为 3。
- max：可选。指示单词中字符的最大个数。默认值为 10。

```
Random.word()
// => "fxpocl"
Random.word(5)
// => "xfqjb"
Random.word(3, 5)
// => "kemh"
```

> 目前单词中的字符是随机的小写字母，未来会根据词法生成『可读』的单词。

Random.cword(pool?, min?, max?)

- Random.cword()
- Random.cword(pool)
- Random.cword(length)
- Random.cword(pool, length)
- Random.cword(min, max)
- Random.cword(pool, min, max)

随机生成一个汉字。

参数说明：

- pool：可选。汉字字符串。表示汉字字符池，将从中选择一个汉字字符返回。
- min：可选。随机汉字字符串的最小长度。默认值为 1。
- max：可选。随机汉字字符串的最大长度。默认值为 1。

```
Random.cword()
// => "干"
Random.cword('零一二三四五六七八九十')
// => "六"
Random.cword(3)
// => "别金提"
Random.cword('零一二三四五六七八九十', 3)
// => ""七七七""
Random.cword(5, 7)
// => "设过证全争听"
Random.cword('零一二三四五六七八九十', 5, 7)
// => "九七七零四"
```

Random.title(min?, max?)

- Random.title()
- Random.title(len)
- Random.title(min, max)

随机生成一句标题，其中每个单词的首字母大写。

参数说明：

- len：可选。指示单词中字符的个数。默认值为 3 到 7 之间的随机数。
- min：可选。指示单词中字符的最小个数。默认值为 3。
- max：可选。指示单词中字符的最大个数。默认值为 7。

```
Random.title()
// => "Rduqzr Muwlmmlg Siekwvo Ktn Nkl Orn"
Random.title(5)
// => "Ahknzf Btpehy Xmpc Gonehbnsm Mecfec"
Random.title(3, 5)
// => "Hvjexiondr Pyickubll Owlorjvzys Xfnfwbfk"
```

Random.ctitle(min?, max?)

- Random.ctitle()
- Random.ctitle(len)
- Random.ctitle(min, max)

随机生成一句中文标题。

> 参数的含义和默认值同 Random.title(min?, max?)

说明：

- len：可选。指示单词中字符的个数。默认值为 3 到 7 之间的随机数。
- min：可选。指示单词中字符的最小个数。默认值为 3。
- max：可选。指示单词中字符的最大个数。默认值为 7。

```
Random.ctitle()
// => "证构动必作"
Random.ctitle(5)
// => "应青次影育"
Random.ctitle(3, 5)
// => "出料阶相"
```

Name

Random.first()

- Random.first()：随机生成一个常见的英文名。

```
Random.first()
// => "Nancy"
```

Random.last()

- Random.last()：随机生成一个常见的英文姓。

```
Random.last()
// => "Martinez"
```

Random.name(middle?)

- Random.name()
- Random.name(middle)

随机生成一个常见的英文姓名。

参数说明：

- middle：可选。布尔值。指示是否生成中间名。

```
Random.name()
// => "Larry Wilson"
Random.name(true)
// => "Helen Carol Martinez"
```

Random.cfirst()

- Random.cfirst()：随机生成一个常见的中文名。

```
Random.cfirst()
// => "曹"
```

Random.clast()

- Random.clast()：随机生成一个常见的中文姓。

```
Random.clast()
// => "艳"
```

Random.cname()

- Random.cname()：随机生成一个常见的中文姓名。

```
Random.cname()
// => "袁军"
```

Web

Random.url(protocol?, host?)

- Random.url()
- Random.url(protocol, host)

随机生成一个 URL。

参数说明：

- protocol：指定 URL 协议。例如 http。
- host：指定 URL 域名和端口号。例如 nuysoft.com。

```
Random.url()
// => "mid://axmg.bg/bhyq"
Random.url('http')
// => "http://splap.yu/qxzkyoubp"
Random.url('http', 'nuysoft.com')
// => "http://nuysoft.com/ewacecjhe"
```

Random.protocol()

- Random.protocol()：随机生成一个 URL 协议。返回以下值一：

```
'http'、'ftp'、'gopher'、'mailto'、'mid'、'cid'、'news'、'nntp'、'prospero'、'telnet'、'rlogin'、'tn3270'、'wais'。
```

```
Random.protocol()
// => "ftp"
```

Random.domain()

- Random.domain()：随机生成一个域名。

```
Random.domain()
// => "kozfnb.org"
```

Random.tld()

- Random.tld():随机生成一个顶级域名（Top Level Domain）。

```
Random.tld()
// => "net"
```

Random.email(domain?)

- Random.email()
- Random.email(domain)

随机生成一个邮件地址。

参数说明：

- domain：指定邮件地址的域名。例如 nuysoft.com。

```
Random.email()
// => "x.davis@jackson.edu"
Random.email('nuysoft.com')
// => "h.pqpneix@nuysoft.com"
```

Random.ip()

- Random.ip()：随机生成一个 IP 地址。

```
Random.ip()
// => "34.206.109.169"
```

Address

Random.region()

- Random.region()：随机生成一个（中国）大区。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.region()); // 华东
    </script>
</body>
```

Random.province()

- Random.province()：随机生成一个（中国）省（或直辖市、自治区、特别行政区）。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.province()); // 贵州省
    </script>
</body>
```

Random.city(prefix?)

- Random.city()
- Random.city(prefix)

随机生成一个（中国）市。

参数说明：

- prefix：可选。布尔值。指示是否生成所属的省。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.city()); // 天津市
        console.log(Random.city(true)); // 四川省 南充市
    </script>
</body>
```

Random.county(prefix?)

- Random.county()
- Random.county(prefix)

随机生成一个（中国）县。

参数说明：

- prefix：可选。布尔值。指示是否生成所属的省、市。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
```

```
        let Random = Mock.Random;
        console.log(Random.county()); // 通道侗族自治县
        console.log(Random.county(true)); // 吉林省 辽源市 东辽县
    </script>
</body>
```

Random.zip()

- Random.zip()：随机生成一个邮政编码（六位数字）。

```
<body>
    <script src="./mock-min.js"></script>
    <script>
        let Random = Mock.Random;
        console.log(Random.zip()); // 867792
    </script>
</body>
```

Helper

Random.capitalize(word)

- Random.capitalize(word)：把字符串的第一个字母转换为大写。

```
Random.capitalize('hello')
// => "Hello"
```

Random.upper(str)

- Random.upper(str)：把字符串转换为大写。

```
Random.upper('hello')
// => "HELLO"
```

Random.lower(str)

- Random.lower(str)：把字符串转换为小写。

```
Random.lower('HELLO')
// => "hello"
```

Random.pick(arr)

- Random.pick(arr)：从数组中随机选取一个元素，并返回。

```
Random.pick(['a', 'e', 'i', 'o', 'u'])
// => "o"
```

Random.shuffle(arr)

- Random.shuffle(arr)：打乱数组中元素的顺序，并返回。

```
Random.shuffle(['a', 'e', 'i', 'o', 'u'])
// => ["o", "u", "e", "i", "a"]
```

Miscellaneous

Random.guid()

- Random.guid()：随机生成一个 GUID。

```
Random.guid()
// => "662C63B4-FD43-66F4-3328-C54E3FF0D56E"
```

Random.id()

- Random.id()：随机生成一个 18 位身份证。

```
Random.id()
// => "420000200710091854"
```

Random.increment(step?)

- Random.increment()
- Random.increment(step)

生成一个全局的自增整数。

参数说明：

- step：可选。整数自增的步长。默认值为 1。

```
Random.increment()
// => 1
Random.increment(100)
// => 101
Random.increment(1000)
// => 1101
```