第 5 章

UI层的松耦合

在 Web 开发中，用户界面（User Interfaoe，UI）是由三个彼此隔离又相互作用的层定义的。

- HTML 用来定义页面的数据和语义。
- CSS 用来给页面添加样式，创建视觉特征。
- JavaScript 用来给页面添加行为，使其更具交互性。

UI 层次关系是这样的，HTML 在最底层，CSS 和 JavaScript 在高层，如图 5-1 所示。

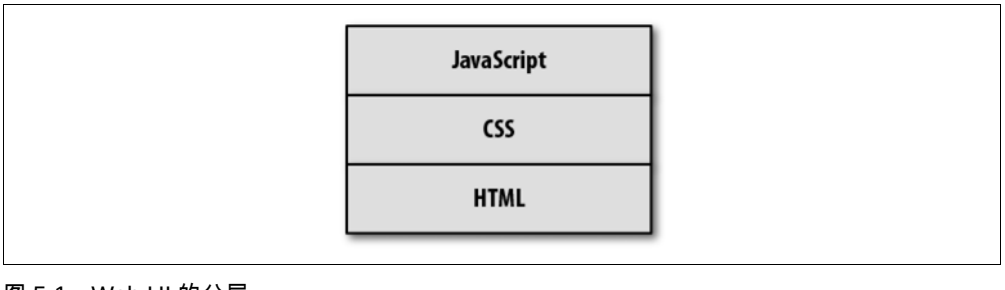

图 5-1　Web UI 的分层

在实际场景中，CSS 和 JavaScript 更像是兄弟关系而非依赖关系（JavaScript 依赖 CSS）。一个页面很可能只有 HTML 和 CSS 而没有 JavaScript，同样，一个页面也

可以只有 HTML 和 JavaScript 而没有 CSS。我更愿意以图 5-2 所示的分层来理解三者的关系。

为了更多地增加分层的合理性和消除依赖性，我们认为在所有 Web UI 中，CSS 和 JavaScript 是同等重要的。比如，JavaScript 的正确运行不应当依赖 CSS——在缺少 CSS 的情况下也要能够正确运行，尽管两者之间可能有些互动。

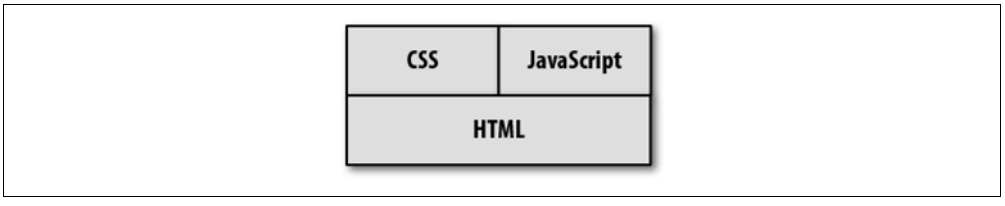

图 5-2　更新后的 Web UI 分层

5.1　什么是松耦合

很多设计模式就是为了解决紧耦合的问题。如果两个组件耦合太紧，则说明一个组件和另一个组件直接相关，这样的话，如果修改一个组件的逻辑，那么另外一个组件的逻辑也需修改。比如，假设有一个名为 error 的 CSS 类名，它是贯穿整个站点的，它被嵌入到 HTML 之中。如果有一天你觉得 error 的取名并不合适，想将它改为 warning，你不仅需要修改 CSS 还要修改用到这个 className 的 HTML。HTML 和 CSS 紧耦合在一起。这只是一个简单的例子。想象一下，如果一个系统包含上百个组件，那这简直就是一场噩梦。

当你能够做到修改一个组件而不需要更改其他的组件时，你就做到了松耦合。对于多人大型系统来说，有很多人参与维护代码，松耦合对于代码可维护性来说至关重要。你绝对希望开发人员在修改某部分代码时不会破坏其他人的代码。

当一个大系统的每个组件的内容有了限制，就做到了松耦合。本质上讲，每个组件需要保持足够瘦身来确保松耦合。组件知道的越少，就越有利于形成整个系统。

有一点需要注意：在一起工作的组件无法达到“无耦合”（no coupling）。在所有系统中，组件之间总要共享一些信息来完成各自的工作。这很好理解，我们的目标是确保对一个组件的修改不会经常性地影响其他部分。

如果一个 Web UI 是松耦合的，则很容易调试。和文本或结构相关的问题，通过查找 HTML 即可定位。当发生了样式相关的问题，你知道问题出现在 CSS 中。最后，对于那些行为相关的问题，你直接去 JavaScript 中找到问题所在，这种能力是 Web 界面的可维护性的核心部分。

5.2 将JavaScript从CSS中抽离

在 IE8 和更早版本的浏览器中有一个特性让人爱少恨多，即 CSS 表达式（CSS expression）。CSS 表达式允许你将 JavaScript 直接插入到 CSS 中，这样可以在 CSS 代码中直接执行运算或其他操作。比如，下面这段代码设置了元素宽度以匹配浏览器宽度。

```
/* 不好的写法 */
.box {
    width: expression(document.body.offsetWidth + "px");
}
```

CSS 表达式包裹在一个特殊的 expression()函数中，可以给它传入任意 JavaScript 代码。浏览器会以高频率重复计算 CSS 表达式，这严重影响了性能，甚至 Steve Souder 的那本《高性能网站建设指南》也特意提到这一点，要避免使用 CSS 表达式（规则 7：避免使用 CSS 表达式）。

除了性能问题之外，在 CSS 中嵌入 JavaScript 代码对于代码维护来说亦是一场噩梦。这方面我有着切身的体会。在 2005 年，有一个 JavaScript bug 分配到我这里，从一开始就很棘手。这个 bug 只在 IE 下才有，并且只有当浏览器窗口大小发生几次改变的时候才会出现。在当时 IE 下，最好的 JavaScript 调试器就是 Visual Studio，但我无论如何也没办法找到错误。我花了一整天时间来设置断点和插入 alert()语句试图找到问题的根源。

在快下班的时候，我被逼无奈使用了我最不喜欢的调试方法：逐段删除代码。我一个接一个地删除 JavaScript 代码，尝试复现这个问题。我很快就删除了页面中所有的 JavaScript 代码，但问题依然存在，这让我困惑至极。我简直不敢相信自己的眼睛。页面中没有任何 JavaScript 代码，却报 JavaScript 错误——真的是见鬼了。

直到今天，我依然很纳闷是什么原因促使我去关注 CSS。那时我甚至不知道要去 CSS

中查什么。只是小心翼翼地从头开始看，一点点地向下滚动我的代码，试图找出一些不一样之处。最终，我发现了症结所在：CSS 表达式。当我将其删除，JavaScript 错误也随之不见了。

正是这次经历促使我提炼出了本章所提到的准则。我花了一整天时间在 JavaScript 中查找脚本错误，罪魁祸首竟然是 CSS。解决这个错误并不难，但是在我认为不会出错的代码里找错误，这着实是一件可笑的浪费时间的事。

幸运的是，IE9 不再支持 CSS 表达式了，但是老版本的 IE 依然可以运行 CSS 表达式。尽管我们很少用 CSS 表达式去实现一些罕见的功能，来让低版本浏览器里也达到和高级浏览器一致的表现，但还是要克制住这份冲动，避免不必要地浪费时间和精力。将 JavaScript 从 CSS 中抽离出来。

5.3 将CSS从JavaScript中抽离

有时候，保持 CSS 和 JavaScript 之间清晰的分离是很有挑战的。这两门语言相互协作得很不错，所以我们经常将样式数据和 JavaScript 混写在一起。通过脚本修改样式最流行的一种方法是，直接修改 DOM 元素的 style 属性。style 属性是一个对象，包含了可以读取和修改的 CSS 属性。比如，你可以像下面这样修改元素里的文本颜色。

```
// 不好的写法
element.style.color = "red";
```

我们经常看到使用这种方法来修改多个样式属性的代码段，比如：

```
// 不好的写法
element.style.color = "red";
element.style.left = "10px";
element.style.top = "100px";
element.style.visibility = "visible";
```

这种方法是有问题的，因为样式信息是通过 JavaScript 而非 CSS 来承载的。当出现了样式问题，你通常首会先去查找 CSS。直到你精疲力竭地排除了所有可能性，才会去 JavaScript 中查找样式信息。

开发者修改 style 对象还有一种方式，给 cssText 属性赋值整个 CSS 字符串，看下

面这个例子。

```
    // 不好的写法
    element.style.cssText = "color: red; left: 10px; top: 100px; visibility:
hidden";
```

使用 cssText 属性只是一次性设置多个 CSS 属性的一种快捷写法。这种模式同样有问题，比如在设置单个属性时：将样式信息写入 JavaScript 带来了可维护性问题。

将 CSS 从 JavaScript 中抽离意味着所有的样式信息都应当保持在 CSS 中。当需要通过 JavaScript 来修改元素样式的时候，最佳方法是操作 CSS 的 className，比如，我想在页面中显示一个对话框，在 CSS 中的样式定义是像下面这样的。

```
    .reveal {
        color: red;
        left: 10px;
        top: 100px;
        visibility: visible;
    }
```

然后，在 JavaScript 中像这样将样式添加至元素。

```
    // 好的写法 - 原生方法
    element.className += " reveal";

    // 好的写法 - HTML5
    element.classList.add("reveal");

    // 好的写法 - YUI
    Y.one(element).addClass("reveal");

    // 好的写法 - jQuery
    $(element).addClass("reveal");

    // 好的写法 - Dojo
    dojo.addClass(element, "reveal");
```

由于 CSS 的 className 可以成为 CSS 和 JavaScript 之间通信的桥梁。在页面的生命周期中，JavaScript 可以随意添加和删除元素的 className。而 className 所定义的样式则在 CSS 代码之中。任何时刻，CSS 中的样式都是可以修改的，而不必更新 JavaScript。JavaScript 不应当直接操作样式，以便保持和 CSS 的松耦合。

有一种使用 style 属性的情形是可以接收的：当你需要给页面中的元素作定位，使其相对于另外一个元素或整个页面重新定位。这种计算是无法在 CSS 中完成的，因此这时是可以使用 style.top、style.left、style.bottom 和 style.right 来对元素作正确定位的。在 CSS 中定义这个元素的默认属性，而在 JavaScript 中修改这些默认值。

5.4 将JavaScript从HTML中抽离

很多人学习 JavaScript 之初所做的第一件事是，将脚本嵌入到 HTML 中来运行。有很多种方式。第一种方式是使用 on 属性（比如 onclick）来绑定一个事件处理程序。

```
<!-- 不好的写法 -->
<button onclick="doSomething()" id="action-btn">Click Me</button>
```

这种写法在 2000 年的时候非常流行，多数网站都采用了这种写法。HTML 代码中放满了 onclick 和其他的事件处理程序，很多元素都包含这样的属性。尽管这种代码在多数场景下是正常工作的，但却是两个 UI 层（HTML 和 JavaScript）的深耦合，因此这种写法是有一些问题的。

首先，当按钮上发生点击事件时，doSomething()函数必须存在。那些在 2000 年左右开发过网站的人对这个问题非常熟悉。包含 doSomething()的代码可能是从外部文件载入或存在于 HTML 文件的后面某处。不管哪种情形，都有可能出现用户点击按钮时这个函数还不存在，这时就会报 JavaScript 错误。页面会弹出错误信息或者点击事件不会有任何响应。两种情形都是我们所不希望发生的。

第二个问题在于可维护性方面。如果你修改了 doSomething()的函数名，将会发生什么事情？如果这时点击按钮调用了别的函数又会怎样呢？在这两个例子中，你需要同时修改 JavaScript 和 HTML 代码。这就是典型的紧耦合的代码。

你的绝大多数（并非所有的）JavaScript 代码都应当包含在外部文件中，并在页面中通过<script>标签来引用。在 HTML 代码中，也不应当直接给 on 属性挂载事件处理程序。相反，一旦外部脚本文件加载至页面，则使用 JavaScript 方法来添加事件处理程序。对于支持 2 级 DOM 模型的浏览器来说，用如下这段代码可以完成上

一个例子中的功能。

```
function doSomething() {
    // 代码
}

var btn = document.getElementById("action-btn");
btn.addEventListener("click", doSomething, false);
```

这种方法的优势在于，函数 doSomething()的定义和事件处理程序的绑定都是在一个文件中完成的。如果函数名称需要修改，则只需修改一个文件；如果点击发生时想额外做一些动作，也只需在一处做修改。

IE8 及其更早的版本不支持 addEventListener()函数，因此你需要一个标准的函数将这些差异性做封装。

```
function addListener(target, type, handler) {
    if (target.addEventListener) {
        target.addEventListener(type, handler, false);
    } else if (target.attachEvent) {
        target.attachEvent("on" + type, handler);
    } else {
        target["on" + type] = handler;
    }
}
```

这个函数可以做到在各种浏览器中给一个元素添加事件处理程序，甚至可以降级到支持给 0 级 DOM 模型对象的 on 属性赋值处理程序（只有在非常古老的浏览器，比如 Netscape 4 中，才会执行这一步，因此这段代码可以在所有情形下都正常工作）。我们常常像下面这样来使用这个方法。

```
function doSomething() {
    // 代码
}

var btn = document.getElementById("action-btn");
addListener(btn, "click", doSomething);
```

如果你用了 JavaScript 类库，可以使用类库提供的方法来给元素挂载事件处理程序。这里给出一些流行的类库中的实例代码。

```
// YUI
Y.one("#action-btn").on("click", doSomething);
```

```
// jQuery
$("#action-btn").on("click", doSomething);

// Dojo
var btn = dojo.byId("action-btn");
dojo.connect(btn, "click", doSomething);
```

在 HTML 中嵌入 JavaScript 的另一种方法是使用<script>标签，标签内包含内联的脚本代码。

```
<!-- 不好的写法 -->
<script>
    doSomething();
</script>
```

最好将所有的 JavaScript 代码都放入外置文件中，以确保在 HTML 代码中不会有内联的 JavaScript 代码。这样做的原因是出于紧急调试的考虑。当 JavaScript 报错，你的下意识的行为应当是去 JavaScript 文件中查找原因。如果在 HTML 中包含 JavaScript 代码，则会阻断你的工作流（workflow interruption）。你必须要首先确定 JavaScript 代码是在文件里（通常如此）还是在 HTML 中。之后你才会开始调试。

这一点看起来无关紧要，尤其是在有很多优秀且强大的调试工具的今天，但它实际上是可维护性难题的一个重要方面。“可预见性”（Predictability）会带来更快的调试和开发，并确信（而非猜测）从何入手调试 bug，这会让问题解决得更快、代码总体质量更高。

5.5 将HTML从JavaScript中抽离

正如我们需要将 JavaScript 从 HTML 中抽离一样，最好也将 HTML 从 JavaScript 中抽离。就像上文提到的，当需要调试一个文本或结构性的问题时，你更希望从 HTML 开始调试。在我的职业生涯中，当我在 HTML 中查找问题原因时，最终却发现真正的问题被深埋在 JavaScript 代码之中，这种情形发生过太多太多次了。

在 JavaScript 中使用 HTML 的情形往往是给 innerHTML 属性赋值时，比如：

```
// 不好的写法
```

```
var div = document.getElementById("my-div");
div.innerHTML = "<h3>Error</h3><p>Invalid e-mail address.</p>";
```

将 HTML 嵌入在 JavaScript 代码中是非常不好的实践。原因有几个。第一，正如上文提到的，它增加了跟踪文本和结构性问题的复杂度。调试上面这段标签的典型方法是首先去浏览器调试工具中的 DOM 树里查找，然后打开页面的 HTML 源码对比其不同。一旦 JavaScript 做了除简单 DOM 操作之外的事情，追踪 bug 就变得很成问题了。

用这种方法（将 JavaScript 从 HTML 中抽离）来解决第二个问题则提高了代码的可维护性。如果你希望修改文本或标签，你只希望去一个地方：可以控制你 HTML 代码的地方。这可能在 PHP 代码中、JSP 文件中或者甚至 Mustache 或 Handlebars 模板中去修改。不管你用了哪种方法，你总是希望你的标签出现在一处，以便很方便地更新它们。嵌入在 JavaScript 代码中的 HTML 标签则不便被修改，因为（通常）你无法（下意识地）想到：如果大部分标签都放置于一个目录下的模板文件里，你何以想到会去 JavaScript 中去修改 HTML 标签呢？

相比于修改 JavaScript 代码，修改标签通常不会引发太多错误。当 HTML 和 JavaScript 混淆在一起时，则将这个问题变得复杂化了。JavaScript 字符串需要对引号做适当的转义，这样做则会导致它和模板语言的原生语法略有差异。

因为多数 Web 应用本质上都是动态的，而在页面的生命周期内，JavaScript 通常用来修改 UI，必然需要通过 JavaScript 向页面插入或修改标签。有多种方法可以以低耦合的方式完整这项工作。

5.5.1 方法 1：从服务器加载

第一种方法是将模板放置于远程服务器[①]，使用XMLHttpRequest对象来获取外部标签。相比于多页应用（multiple-page applications），这种方法对单页应用（single-page applications）带来更多的便捷。例如，点击一个链接，希望弹出一个新对话框，代码可能如下。

```
function loadDialog(name, oncomplete) {
```

① 译注：这种方法（从服务器获取模板）很容易造成 XSS 漏洞，需要服务器对模板文件做适当转义处理，比如<和>以及双引号等，当然前端也应当给出与之匹配的渲染规则，总之这种方法需要一揽子前后端的转码和解码策略来尽可能地封堵 XSS 漏洞。

```
    var xhr = new XMLHttpRequest();
    xhr.open("get", "/js/dialog/" + name, true);

    xhr.onreadystatechange = function() {

        if (xhr.readyState == 4 && xhr.status == 200) {

            var div = document.getElementById("dlg-holder");
            div.innerHTML = xhr.responseText;
            oncomplete();

        } else {
            // 处理错误
        }
    };

    xhr.send(null);
}
```

这里没有将 HTML 字符串嵌入在 JavaScript 里，而是向服务器发起请求获取字符串，这样可以让 HTML 代码以最合适的方式注入到页面中。JavaScript 类库将这个操作做了封装，使得直接给 DOM 元素挂载内容变得非常方便。对此 YUI 和 jQuery 都提供了简单的 API。

```
// YUI
function loadDialog(name, oncomplete) {
    Y.one("#dlg-holder").load("/js/dialog/" + name, oncomplete);
}
// jQuery
function loadDialog(name, oncomplete) {
    $("#dlg-holder").load("/js/dialog/" + name, oncomplete);
}
```

当你需要注入大段 HTML 标签到页面中时，使用远程调用的方式来加载标签是非常有帮助的。出于性能的原因，将大量没用的标签存放于内存或 DOM 中是很糟糕的做法。对于少量的标签段，你可以考虑采用客户端模板。

5.5.2 方法 2：简单客户端模板

客户端模板是一些带“插槽”的标签片段，这些“插槽”会被 JavaScript 程序替换为数据以保证模板的完整可用。比如，一段用来添加数据项的模板看起来就像下面这样。

```
<li><a href="%s">%s</a></li>
```

这段模板中包含%s 占位符，这个位置的文本会被程序替换掉（这个格式和 C 语言中的 sprintf()一模一样）。JavaScript 程序会将这些占位符替换为真实数据，然后将结果注入 DOM。下面这段代码给出了这样一个函数及其用法。

```
function sprintf(text) {
    var i=1, args=arguments;
    return text.replace(/%s/g, function() {
        return (i < args.length) ? args[i++] : "";
    });
}

// 用法
var result = sprintf(templateText, "/item/4", "Fourth item");
```

将模板文本传入 JavaScript 是这个过程的重要一环。本质上讲，你一点也不希望在 JavaScript 中嵌入模板文本，而是将模板放置于他处。通常我们将模板定义在其他标签之间，直接存放于 HTML 页面里，这样可以被 JavaScript 读取，用两种方法之一即可做到。第一种方法是在 HTML 注释中包含模板文本。注释是和元素及文本一样的 DOM 节点，因此可以通过 JavaScript 将其提取出来，比如：

```
<ul id="mylist"><!--<li id="item%s"><a href="%s">%s</a></li>-->
    <li><a href="/item/1">First item</a></li>
    <li><a href="/item/2">Second item</a></li>
    <li><a href="/item/3">Third item</a></li>
</ul>
```

在这个用法里，这段注释作为列表的第一个子节点，被恰当地放置于上下文中。下面这段 JavaScript 代码则将模板文本从注释中提取出来。

```
var mylist = document.getElementById("mylist"),
    templateText = mylist.firstChild.nodeValue;
```

一旦提取出了模板文本，紧接着需要将它格式化并插入 DOM。通过下面这个函数可以完成这些操作。

```
function addItem(url, text) {
    var mylist = document.getElementById("mylist"),
        templateText = mylist.firstChild.nodeValue,
        result = sprintf(template, url, text);
```

```
    div.innerHTML = result;
    mylist.insertAdjacentHTML("beforeend", result);
}

// 用法
addItem("/item/4", "Fourth item");
```

我们给这个方法传入了一些数据信息，用它们来处理模板文本，并用 insertAdjacentHTML()将结果注入 HTML。这一步操作将 HTML 字符串转换为一个 DOM 节点，并将它作为子节点插入到<ul>里。

将模板数据嵌入到 HTML 页面里的第二个方法是使用一个带有自定义 type 属性的<script>元素。浏览器会默认地将<script>元素中的内容识别为 JavaScript 代码，但你可以通过给 type 赋值为浏览器不识别的类型，来告诉浏览器这不是一段 JavaScript 脚本，比如：

```
<script type="text/x-my-template" id="list-item">
    <li><a href="%s">%s</a></li>
</script>
```

你可以通过<script>标签的 text 属性来提取模板文本。

```
var script = document.getElementById("list-item"),
    templateText = script.text;
```

这样 addItem()函数就会变为这样。

```
function addItem(url, text) {
    var mylist = document.getElementById("mylist"),
        script = document.getElementById("list-item"),
        templateText = script.text,
        result = sprintf(template, url, text),
        div = document.createElement("div");

    div.innerHTML = result.replace(/^\s*/, "");
    list.appendChild(div.firstChild);
}

// 用法
addItem("/item/4", "Fourth item");
```

这个版本的函数有一处修改，即去掉了模板文本中的前导空格。之所以会出现这个

多余的前导空格，是因为模板文本总是在<script>起始标签的下一行。如果将模板文本原样注入，则会在 里创建一个文本节点，这个文本节点的内容是一个空格，而这个文本节点最终会代替<li>被添加进列表之中。

5.5.3 方法 3：复杂客户端模板

上几节中介绍的模板格式都非常简单，并无太多转义。如果你想用一些更健壮的模板，则可以考虑诸如 Handlebars（http://handlebarsjs.com/）所提供的解决方案。Handlebars 是专为浏览器端 JavaScript 设计的完整的客户端模板系统。

在 Handlebar 的模板中，占位符使用双花括号来表示。我们将上一节中的模板表示为 Handlebars 版本如下。

```
<li><a href="{{url}}">{{text}}</a></li>
```

在 Handlebars 模板中，占位符都标记为一个名称，以便可以在 JavaScript 中设置其映射。Handlebars 建议将模板嵌入 HTML 页面中，并使用 type 属性为"text/x-handlebars-template"的<script>标签来表示。

```
<script type="text/x-handlebars-template" id="list-item">
    <li><a href="{{url}}">{{text}}</a></li>
</script>
```

要想使用这个模板，你首先必须将 Handlebars 类库引入页面，这个类库会创建一个名为 Handlebars 的全局变量，用来将模板文本编译为一个函数。

```
var script = document.getElementById("list-item"),
    templateText = script.text,
    template = Handlebars.compile(script.text);
```

这时，变量 template 包含了一个函数，当执行这个函数时则返回一个格式化好的字符串。你需要做的仅仅是传入一个包含 name 和 url 属性的对象。

```
var result = template({
    text: "Fourth item",
    url: "/item/4"
});
```

参数会自动做 HTML 转义，转义操作也是格式化的一部分。转义是为了增强模板

的安全性，并确保简单的文本值不会破坏你的标签结构。比如，字符&会自动转义为&。

现在将它们合并入一个单独的函数中。

```
function addItem(url, text) {
    var mylist = document.getElementById("mylist"),
        script = document.getElementById("list-item"),
        templateText = script.text,
        template = Handlebars.compile(script.text),
        div = document.createElement("div"),
        result;

    result = template({
        text: text,
        url: url
    });

    div.innerHTML = result;
    list.appendChild(div.firstChild);
}

// 用法
addItem("/item/4", "Fourth item");
```

这个简单的例子并未真正体现 Handlebar 的灵活性。除了简单的占位符替换之外，Handlebars 模板同样支持一些简单的逻辑和循环。

假设你想渲染整个列表而非一条记录，但你又想仅在实际存在可渲染的记录时才生成一条记录。你可以像下面这样创建一条 Handlebars 模板。

```
{{#if items}}
<ul>
    {{#each items}}
    <li><a href="{{url}}">{{text}}</a></li>
    {{/each}}
</ul>
{{/if}
```

在这段模板中，{{#if}}代码段确保了只有 item 数组中至少存在一条记录时才会真正渲染完整的标签。{{#each}}代码段紧接着遍历数组中的每条记录，因此，你将模板编译为一个函数，然后给这个函数传入一个包含 items 属性的对象，比如下面这段代码。

```
// 返回一个空字符串
var result = template({
    items: []
});

// 返回包含两个记录的 HTML 列表
var result = template({
    items: [
        {
              text: "First item",
              url: "/item/1"
        },
        {
              text: "Second item",
              url: "/item/2"
        }
    ]
});
```

Handlebar 包含其他很多种逻辑原语，因此它也是一款强大的 JavaScript 模板引擎。