第2章
流动布局

一棵大橡树被风连根拔起，然后落到了一片芦苇丛中。橡树对芦苇们说：“我很想知道，你们如此轻而弱小，却为什么没有被大风吹走呢？”芦苇们回答道：“你和大风打斗对抗，结果就被摧毁了；而我们在大风来临之前便弯下了腰，并因此保持完好而存活了下来。”

——“橡树与芦苇”伊索寓言

在“橡树与芦苇”的故事中，大橡树和芦苇都被风吹来吹去，橡树想要一直站得又高又稳，并因此与大风做着顽强的抵抗，但最终它还是被大风给打败了。

与之相反，芦苇则弯下了腰，当然也不是它们想要这样，只是它们能弯得下腰。它们不与大风对着干，而是就让自己随风摇曳。虽然被大风吹得扭曲甚至彼此缠绕在一起，但毕竟它们最后还是活了下来。

长期以来，我们其实都是在按照橡树的思维方式来建立网站的，主要表现为设计中的一些硬性规定以及使用固定宽度。它们看起来好像很不错，直到它们不可避免地遇到了我们无法预知的Web。现在的我们需要去拥抱Web，而不是与无法预知的Web也来场对抗。

这也是John Allsopp在2000年为A List Apart写的“*A Dao of Web Design*”（Web设计之道）中所极力指出的。在这个今天还是最佳实践，而明天就有可能成为笑柄的业界，Allsopp的见解后来被证明是难得的先见之明——他认为Web社区需要拥抱Web的灵活性，同时不要再将Web的不可控性视为是一种限制：

> “我相信，Web最伟大的优点常常被人们视为是限制和缺陷。灵活性是Web固有的本性，作为设计师和开发者的我们，应该去拥抱它的这一特性，并且要设计开发出同样具有灵活性的页面，使得所有的设备都可以访问。”

Allsopp认识到Web的灵活性和不可预测性不应该是我们与之斗争的对象，因为这些都是Web的特性，而不是它的漏洞。也正是由于这些特性才使得Web如此独特，以至于成为了比印刷品还要强大的媒介。

随着越来越多设备的涌现，人们越来越难以继续忽略Web所固有的灵活性和不可预测性。终于在12年之后，业界才终于赶上了Allsopp那篇文章中提出的新思潮。而作为拥抱灵活性的第一步，就是要为我们的站点创建流动布局，并藉此来对不同尺寸的设备屏幕作出不同的响应。

在本章中你将会学到：

- 四种不同的布局类型；
- 指定字体大小的几种不同方法，以及如何从中作出选择；
- 如何创建流动布局；
- 如何将固定宽度的资源（比如图片）与流动布局很好地结合起来；

- 如何利用display:table来结合使用固定宽度和流动宽度。

2.1　布局选项

要想弄清楚在哪些情况下灵活布局可能会是我们最好的选择，我们就要先知道我们还有哪些选择。只有先明白了每种选择的优缺点，我们才有可能做出正确的决定，并保证我们的站点能在各种环境中发挥出最大优势。

在广受好评的《Flexible Web Design（灵活的Web设计）》一书中，作者Zoe Mickley定义了4种布局类型：固定布局、流体布局、弹性布局和混合布局。

这里提到的每种布局都有着它们各自的优势、劣势以及面临的挑战。

2.1.1　固定布局

在固定布局中，页面宽度会被指定为特定大小的像素，其中960px是使用最为广泛的宽度。Cameron Moll在2006年发表的一篇名为“*Optimal width for 1024px resolution？*”（1024px分辨率屏幕的最佳页面宽度？）的文章中，对越来越流行的1024分辨率屏幕的“最佳”页面宽度进行了仔细的分析：在考虑了Chrome之后，他认为最佳页面宽度应该介于974px到984px之间。但相比其他数值而言，960对网格布局则更加友好（因为960可以被3、4、5、6、8、10、12和15整除，所以可以为网格布局提供多种可能），而且它也能与互动广告局（IAB）（译者注：互动广告局是一个非盈利性的广告商业组织，主要负责研究、制定广告标准，并为在线广告业提供合法支持）的标准广告大小很好地配合，因此这一宽度得到了广泛采用。

固定布局是Web布局中最为常见的实现方式，因为固定布局会给人一种对页面拥有较多控制权的错觉。能确切地知道你的站点会以多宽来显示，这使得你能创建出拥有大量图形的设计，而且即使在不同的屏幕上，它们也能看起来相当一致。

然而，固定布局最大的问题在于，你做的一切都是在许多假设前提之下的。当你决定站点应该有多宽时，你不得不去猜测什么样的尺寸可以照顾到大多数的访客，而这其中又暗藏着比从表面上看起来多得多的技巧。早在智能手机、平板电脑这些设备出现之前，就已经存在了大量使用不同大小屏幕的访客了，而这也仅

仅是个开始，因为有些人并不会全屏浏览网页，还有些人的浏览器装有带边栏的插件——这些都在很大程度上降低了站点可以利用的实际空间。

此外，固定布局带给我们的所谓“一致性”的优点其实也带有些许误导。如果你的站点是960px宽的，当访客使用屏幕较小（假设是800px宽）的设备浏览站点时，他将只能看到站点的一部分，而且还会看到一个丑陋的水平滚动条（图2.1）。

图2.1 当浏览器宽度小于站点的固定布局的宽度时，用户就会看到丑陋的水平滚动条。

大屏幕设备也同样存在问题：如果有人用大屏幕设备浏览960px宽的站点，那他就会在页面两边看到大片的、未使用的空白区域。虽然留白作为设计的一部分固然是好的，但是对于人们意料之外的大片空白，我想没有谁能从中获益。

严格的固定布局在今天广泛而多样的设备生态系统中早已问题重重，因为当今很多最新、功能最强的手机和平板电脑的浏览器都会默认显示缩小之后的网页，以此来适应它们那较小的屏幕。虽然可以在屏幕上通过手指来缩放页面（当然提供这样的放大功能总比不提供要好），但这样的操作还是会让人感到繁琐，况且这一过程也远远不能称之为是一种享受。

2.1.2　流动布局

在流动布局中，度量的单位不再是像素，而是变成了百分比，这样可使页面具有可变的特性。你可以设计一个占容器宽度60%的内容栏、一个占容器宽度30%的边栏，以及一个占容器宽度10%的分隔区域。采用这种方法定义，意味着你不用再去关心用户使用的到底是1024px宽的桌面浏览器，还是768px宽的平板电脑浏览器，因为元素的宽度会根据浏览器窗口的宽度自动进行调整。

▶注意

Gillenwater在她的分类中称为流体布局；在本书中称为流动布局。

流动布局的设计可以避免许多会在固定布局中遇到的问题，例如，在大多数情况下，流动布局中都不会出现水平滚动条。由于站点可以根据浏览器窗口的宽度来自动调节自身宽度，因此站点将可以充分利用并适应各种大小的可用空间，从而也就避免了在使用大屏幕设备浏览固定布局时，人们所不愿见到的大片空白出现。

同时，在实施诸如媒介查询、针对不同分辨率优化样式这样的响应式策略时，流动布局能使这些策略更容易实现（我们会在后面的章节中对此做详细的介绍）。由于没有那么多的问题需要修复，所以需要写的CSS也相对较少。但是，单独使用流动布局这一种方法依然不足以保证所有元素在从手机到电视机等一系列设备都能保持良好的效果。因为有些文本的行宽在大屏幕上看起来会太宽，而在小屏幕上又看起来太窄。流动布局仅仅是个开始，后面还会有为什么这本书没有到此为止的其他原因。

2.1.3 弹性布局

弹性布局与流动布局类似，只是它们的度量单位不同——通常情况下，在弹性布局中会以em来作为单位。1em相当于当前字体的大小，例如，如果body的字体大小是16px，那么此时1em就等于16px，同理2em等于32px。

弹性布局为设计师们在排版方面提供了强大的控制权。大量研究表明，理想的阅读文本的行宽介于45到70个字符之间。利用弹性布局，你可以将容器的宽度设定为55em，这样就可以使容器维持在一个合适的宽度了。

弹性布局带来的另外一个好处是随着用户增大或者减小字体，使用弹性布局的元素的宽度也会等比例地变化，我们会在后面讨论字体大小时进一步讨论这个问题。

但是，在弹性布局中也可能出现令人讨厌的水平滚动条。如果你把字体大小设置为16px，并把容器宽度设置为55em，那么就会在任何宽度小于880px（16×55）的屏幕中出现水平滚动条。弹性布局中的这个问题甚至比固定布局中的情况还要有更多的不确定性，因为如果用户把字体放大到了18px，那么你的容器宽度便会达到990px（18×55）。

2.1.4 混合布局

我们的最后一个选择就是混合布局，它结合了上面提到的两种或两种以上的布局方式。

例如，假设你有一个300px宽的广告区域，那么你可以将放置广告的边栏指定为固定的300px宽，其余各列的宽度则以百分比为单位。这样做即保证了广告的宽度是固定的300px（考虑到第三方的广告服务，这样安排是很有必要的），同时除边栏之外的其他内容也能填满剩下的整个空间。

● 视口
浏览器呈现网页时的可见区域。

但是在流动布局中使用上面这种方法会很容易使页面变得凌乱不堪，如果把边栏设置为300px宽，把主要内容栏设置为70%，那么当屏幕宽度小于1000px时，你

还是会遇到水平滚动条的问题。因为此时300px宽的边栏已经超过了分配给它的宽度，即视口宽度的30%，剩下的空间已经小于了内容栏所要求的70%。不过万幸的是，还有另外一种创建混合布局的方法，我们会在后面的章节中提到。

如果你是毫无压力地读完上面这一段的，而且没有回想起你高中的数学课，那么我可就要为你鼓掌了。

2.1.5　哪种布局是最具响应性的

那么到底哪种方式才是正确的、能对于各种设备和环境都具有响应性的呢？从根本上来说，这取决于你特定的项目，因为每一种方法都有其优势和不足。

大多数情况下，最佳答案是更具灵活性的那几种布局——流动布局、弹性布局或者混合布局——因为相比固定布局，它们对未来更加友好。

虽然能够通过使用媒介查询来在几个固定布局之间来回切换，但这样做仍旧存在限制。因为使用一个这样的“可切换”的固定布局，会使得你的站点在某几个特定分辨率的屏幕中显示得非常好，但在这几个值之间过渡的那些分辨率的屏幕中就会显得比较尴尬。这样一来，用户就会受到你的决定的支配：如果他的设备与你选定的不一致，那么他的体验还不如你什么都不做时好呢。

所以，虽然“可切换”的固定布局的方法在大方向上是对的，但这有点像是每天晨跑前，你都要花30分钟先坐在沙发上吃个冰激凌一样——虽然有总比没有强，但你没有得到你本应该得到的全部。

相反，如果你使用流动布局，至少你已经得到了很大一部分。甚至没有媒介查询的帮助，你的设计都能够在不同的视口之间作出调整，虽然会有一些不完美。

当你开始强调媒介查询的作用时，你便很容易忽视掉另外一点：其实是弹性布局或流动布局在发挥更主要的作用（参见第3章）。流动布局为你做的事情越多，你需要创建的断点、写的CSS代码就越少。当你有了一个强大的流动布局的时候，媒介查询就会弱化为一种调整设计的手段，而不必再去重新创建另外的设计。

媒介查询

媒介查询允许你根据设备的信息——诸如屏幕宽度、方向或者分辨率等属性来使用不同的样式。

断点

那些被指定的、开始应用某一新的媒介查询的点。例如，一个在980px处的断点的意思是，当浏览器宽度大于或小于这一数值时，新的媒介查询将被触发。

2.2 字体大小

要想让你的设计拥抱Web的流动性，那就意味着在你的设计中要能够灵活地改变字体大小。你可以在Web上使用任意单位来设置字体大小，但主要的方法不外乎三种：像素、em，还有百分比。

2.2.1 像素

长期以来，像素一直都是人们喜欢使用的字体大小单位，其原因很简单：无论浏览器如何设置字体大小，你都能对其进行精确的控制。如果你把字体设置为18px，那么每个浏览器都会将其准确地显示为18px。

但这样的掌控是要付出一定代价的。对于初学者而言，虽然使用像素作为字体大小单位时不需要考虑级联——父元素的字体大小对子元素没有影响，但这也意味着当你想让某些文本以不同的大小显示时就需要对它们逐一设置。这对于维护来说非常不利，因为当你想增大所有字体时，你就不得不去逐一修改每一个值。

更重要的是，以像素为单位的字体存在着潜在的可访问性问题。所有主流的浏览器都允许用户放大或缩小页面，浏览器对该操作通常有两种实现方式。

第一种方式是简单地将页面上所有的东西都放大，所以当访问者放大页面时，包括文本在内的所有的元素都会变大。在这种情况下，无论字体大小使用哪种单位，都允许用户进行缩放（图2.2）。

另外一种方式是浏览器只调整文本的大小，页面上其他元素则保持不变。长期以来这都是常见的浏览器实现方式，并且现在仍有一些浏览器采用这种方式实现缩放功能。

而且不幸的是，以像素为单位的字体在老版本的IE中是无法实现缩放的。这意味着那些使用IE9之前的、字体被设为默认大小的IE浏览器（或者在最新版中启用了字体默认大小的浏览器）的用户，是不能调节你的页面中字体的大小的。

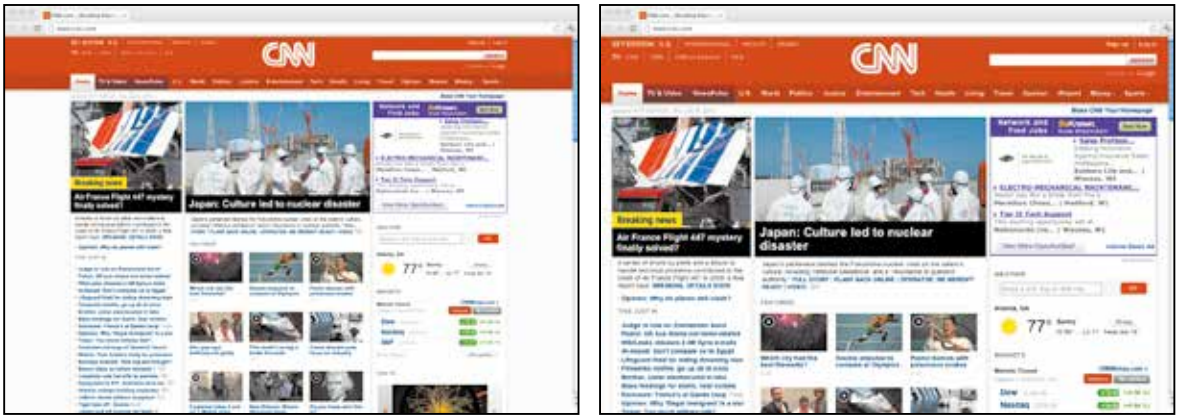


图2.2　在一些较新的浏览器中，包括字体在内的整个页面都会被放大。

无法调整字体大小的问题也同样出现在很多较老的、在触摸屏出现之前就已经存在的设备上。有时所有的页面内容都不能缩放，在这种情况下，虽然字体还是同样的大小，但部分页面也许就不太适合人们阅读了。

浏览器能够调整文本的能力给了用户更多的控制权，另外作为浏览器本来就应该实现的功能，这样做也可以提高站点的可访问性。因为对于某些访问者来说，当字体大小小于某一数值时，就会给他们的阅读造成障碍。所以允许用户增大字体就意味着他们可以继续享用你的站点里的内容。

用像素指定字体大小依然不是一种对未来特别友好的方式。由于不同的设备有着不同的屏幕大小和像素密度，因此以像素为单位的字体也许在这台设备上看起来不错，但在另外一台设备上看起来也许就会要么太大、要么太小了（本章后面“默认字体大小”对此会做进一步的讨论）。使用像素作为字体大小的单位，是与Web的灵活性原则背道而驰的做法。

2.2.2　em

使用em作为字体大小的单位是一种更加流行、更具灵活性的方法 。正如前面介绍过的那样，1em等于默认的字体大小，例如，如果内容的字体大小是16px，那么：

1em = 16px

2em = 32px

em可以跨浏览器进行缩放，而且它们也是级联的——这既可以是好事，也可以是坏事。从简化维护的角度来看这是好事，因为这样相对地指定元素的字体大小，意味着你只需要调整初始化时的基准，其余部分就会自动地进行调整，而且是按比例调整的。

但级联也会使事情变得复杂。例如，考虑一下下面的HTML：

```
<body>
     <div id="main">
          <h1>Question One <span>Please choose an answer from below.
             </span></h1>
<p>In which book did H.G. Wells write: "Great and strange ideas
    transcending experience often have less effect upon men
and women than smaller, more tangible considerations."</p>
     <ol>
          <li>The Invisible Man</li>
          <li>The War of the Worlds</li>
          <li>The World Set Free</li>
     </ol>
</div>
</body>
```

其对应的CSS如下：

```
body {
     font-size: 16px; /* base font size set in pixels */
}
h1 {
     font-size: 1.5em; /* 24px / 16px */
}
span {
     font-size: 1em; /* 16px / 16px */
}
```

在上面的例子中，字体基准被设置为了16px，h1元素的字体大小为1.5em，即24px。我们想把span元素的字体大小设置为16px，所以我们将其设置为1em。但问题在于上下文环境变了：基准已经不再是body字体的16px了，而是h1元素

的24px。因此span元素的字体大小将会显示为24px，而不是我们期望的16px。

所以我们需要调整span元素的font-size来缩小它：

```
span {
     font-size: .666666667em; /* 16px / 24px */
}
```

要试着结构化你的CSS和HTML，并保持字体大小的可预测性。例如，如果你的内容部分使用了基准字体大小，并且只改变了标题元素的大小，那么你就可以避免这些问题了。类似地，如果你的HTML是经过精心设计的，你也可以很容易地通过后代选择器来解决这些问题。

● 后代选择器
一种CSS选择器，用来匹配特定元素的子元素。

2.2.3 百分比

和以em为单位的字体一样，以百分比为单位的字体也是可缩放的，而且也是级联的。另外与em类似的是，如果基准字体大小是16px的话，那么100%相当于16px，200%相当于32px。

虽然从理论上讲，百分比和em没有太大的区别，而且em逐渐成为了Web中更受欢迎的字体度量单位，但其实这其中并没有什么技术上的原因。只有当出于em直接与文本大小相关的考虑时，使用em才更有意义。

然而，承蒙众多用户喜爱的IE浏览器，却在显示以em为单位的基准字体时会有问题。如果基准字体是以em为单位的，那么IE会夸大实际应当调整的字体大小。假如你把基准字体大小设置为了1em，并且把h1元素的设置为了2em，在绝大多数的浏览器中，h1元素会像我们预料的那样显示：它们会变大两倍，但在IE中它们会变得更大。要解决这个问题，我们得感谢下面这个小小的漏洞。

你可以通过在body元素上以百分比为单位指定基准字体大小来绕过这个问题。

```
body {
     font-size: 100%;
}
```

引人注目的是，在大多数浏览器和设备中，默认的字体大小都是16px（后面专栏

中有更多的相关信息）。通过将body的字体设置为100%，你就可以保证页面内容是可缩放的了，而且不会有夸大的现象，之后你就又可以使用em来指定其他元素的字体大小了。

默认字体大小

我们已经知道在桌面浏览器中，默认的页面字体大小是16px，所以当你把页面的font-size指定为100%时，你就能在不同的浏览器中得到相同大小的字体了。

但这一特性对于其他类型的设备来说有时就不那么奏效了。例如，当我在一台运行着黑莓6.0操作系统的黑莓手机上测试时，默认的font-size是22px，而Kindle Touch上的结果则更具戏剧性：它的初始默认大小是26px。

在你开始朝它们扔东西之前，请先听听它们这样做的原因。很多这样的新设备都有着较高的分辨率，因此16px的字体在上面看起来会非常小。大多数设备只好通过向浏览器报告说，自己有一个与实际不符的分辨率来绕过这个问题。例如，iPhone 4的分辨率为640x960，但是它会告诉浏览器自己的分辨率是320x480。

其他诸如Kindle Touch这样的设备，会向浏览器报告真实的分辨率，但会通过增大默认的font-size来进行补偿。

其实最重要的不是屏幕实际的像素大小，屏幕上文字的可读性才是真正重要的。虽然可以将基准的font-size设置为100%，但要记住用户使用的设备的基准字体大小也许不是16px（这是一个在媒介查询中使用em的好例子，我们将会在下一章讨论这方面的内容）。

2.2.4 奖励关卡：rem

还有另外一种极具潜力，并兼具灵活性的设置字体大小的方法：以rem（“root em”）作为单位。rem与em的区别在于：rem的大小与根元素——HTML元素——有关。

使用rem能够避免发生在嵌套元素中的级联问题，让我们更新一下CSS，并以rem为单位来样式化列表项：

```
html {
     font-size: 100%; /* equates to ~16px */
}
h1 {
     font-size: 1.5em; /* 24px / 16px */
}
span {
     font-size: 1rem; /* 16px / 16px */
}
```

在上面这个例子中，h1元素的字体大小依然为24px，但是span元素将会显示为16px，因为在使用root em作为单位之后，span元素将继承html元素的字体大小，而不是包含它的div元素。

但对于rem，这里有一处有关移动浏览器支持情况的告诫。通常情况下，rem在桌面电脑浏览器中都能得到很好的支持：IE9+、Firefox 3.6+、Chrome 6.0+、Safari 5.0+以及Opera 11.6+都支持rem。另外，iOS 4.0+和Android 2.1+也可以提供相应的支持。但不幸的是，在写这本书时其他移动平台（包括流行的Opera Mobile）还都不支持rem。

为了应对上面这些情况，你可以额外设置一个以像素为单位的备用方案。

```
span {
  font-size: 16px;
  font-size: 1rem;
}
```

使用上面这种方法，支持rem的浏览器将会使用rem声明，因为它是在后面声明的，而那些不支持rem的浏览器将会使用第一条使用像素的声明，并忽略rem声明。

2.2.5　哪种单位是最具响应性的

在决定要采用哪种方式时，这里有一些权衡利弊时需要考虑的因素。使用em不但可以使文字能够缩放，而且维护起来也更加容易。例如，如果你要将整个站点的字体都增大，那么只需简单地修改body上字体的百分比即可。如果使用rem，由于需要有像素声明作为补充，所以你还要更新所有的像素值。

● 提示

许多维护人员关心的事情其实都可以通过CSS预处理器来解决，比如SASS（http:// sass-lang.com）或者LESS（http://lesscss.org），并且可以在其中使用变量。

在本书后面的内容当中，我们将以百分比来作为body元素字体大小的单位，而以em来作为其他元素的单位。

2.2.6 从像素转换

你手中的每一个项目在刚开始时都新鲜无比，这当然很好，因为它们允许你从一开始就使用流动的设计。但事实是这样的可能性不大，因为大多数项目都会涉及迁移，此时你要能够将原有的固定大小的元素或者单位转换为一些更加灵活的东西。

鉴于此，让我们先来看一段完全以像素为单位的代码片段（图2.3）。

```
body {
     font-size: 16px;
     font-family: Helvetica, sans-serif;
}
h1 {
     font-size: 24px;
}
span {
     font-size: 12px;
}
```

图2.3 以像素为单位的文本虽然美观，但完全是不灵活的。

在这段代码中，body的字体大小为16px，h1元素的字体大小为24px，span元素的字体大小为12px。

将上面这段代码转换为更具灵活性的代码相对来说比较容易。我们首先来设置body的字体大小：

```
body {
     font-size: 100%;
     font-family: Helvetica, serif;
}
```

你需要记住的是，这里设置100%对于大多数浏览器来说能保证基准font-size的值为16px，此外这样做也为我们提供了一个灵活的基准，让我们能在其之上进行其他设计。

通过一个简单的数学方程式我们就能将剩下的内容轻易地转换为em了。“我知道，我知道”——如果你想做数学题，那最好还是去买本算术书吧。不过谢天谢地，你不必知道圆周率的平方根的余弦值就可以算出这里的em值了，因为方程式很简单：

```
目标 / 上下文环境 = 结果
```

以h1元素为例，对于h1元素而言，我们的目标是24px，上下文环境等于容器元素的font-size值——在这里是body的16px。所以我们用24除以16，结果是1.5em：

```
h1 {
     font-size: 1.5em; /* 24px / 16px */
}
```

注意声明后面的注释。作为一个在回顾我前一天写的代码（更不用说一个月前写的代码）时都会经常挠头的人，我强烈建议你使用注释，它们可以帮你回忆起这些数值是怎么来的。

我们可以将相同的方程式应用到span元素上。因为span元素位于h1元素内，所以上下文环境也改变了：现在是h1元素。经过计算，我们将span的font-size设置为.5em（12/24）。

更新后的代码如下所示：

```
body {
      font-size: 100%;
      font-family: Helvetica, sans-serif;
}
h1 {
      font-size: 1.5em; /* 24px / 16px */
}
span{
      font-size: .5em; /* 12px / 24px */
}
```

灵活地指定字体大小——现在你已经实现它了！现在即使你更改了body的字体大小，body的字体大小与标题元素字体大小之间的比例也不会改变（图2.4）。

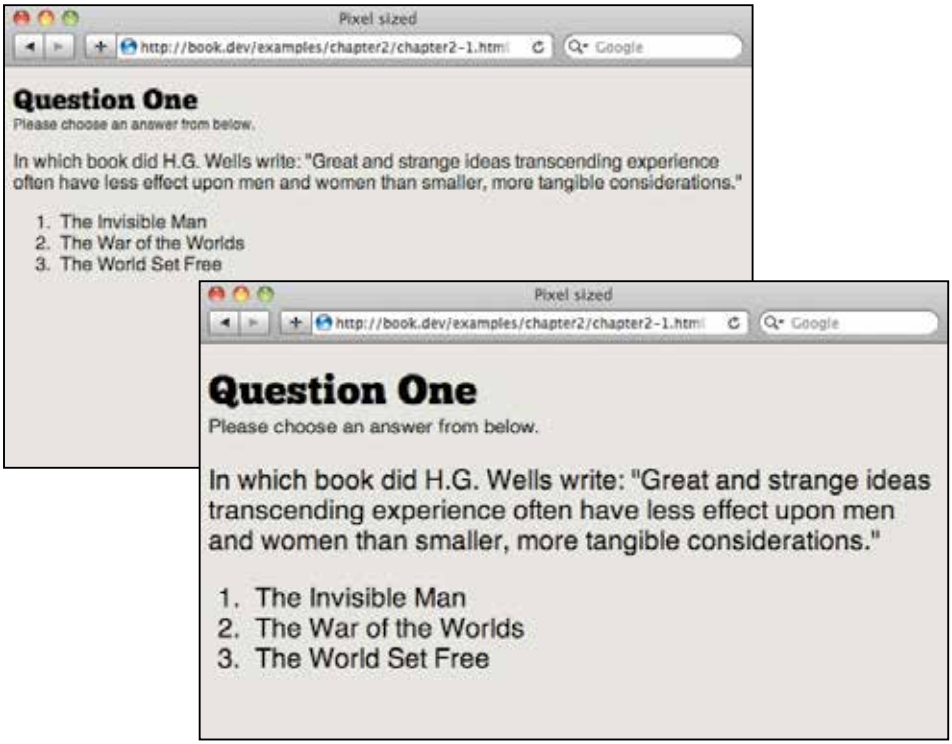


图2.4 灵活地设置的字体虽然看起来和用像素指定大小的字体完全相同，但现在当字体增大时，其比例将保持不变。

◆ 提示

可以访问http://implementingresponsivedesign.com/ex/ch2/ch2.1.html来查看实际效果。

2.3　网格布局

早在Web出现前的数十年，在设计中使用网格就已经是一种异常流行的做法了，因为网格有助于实现站点的平衡、间距以及组织结构。一个实现良好的网格系统会使你的站点井井有条，同时还可以提高页面的可读性和可浏览性。

▶注意

关于网格的更多信息，可以参阅Khoi Vinh的书籍，或者找一份Mark Boulton的系列视频来看。

在《“Ordering Disorder: Grid Principles for Web Design”（秩序之美：网页中的网格设计）》一书中，作者Khoi Vinh便强调了使用网格进行设计的四大主要优势。当你把这些优势同时运用在设计中时，你会看到它们之间更为紧密的联系。

- 网格使信息的展示变得有序、新颖、和谐。
- 网格使得用户能够预测该从哪里获得信息，这在信息交流过程中很有帮助。
- 在与原始总体设想相一致的前提下，网格使添加新的内容更为容易。
- 网格能在设计单一解决方案时促进合作，而不必对总体设想的解决方案妥协。

框架简述

在网上可以找到大量基于网格的框架，这些框架模板和CSS规则能帮你快速搭建出预先定义好的网格布局。在这些框架中，有些是灵活的，而有些则不是，它们中的大多数都介于12列至16列之间。虽然在每个新项目中都使用你最喜爱的框架是一件诱人的事，但其实我们可以做得比这更有创意。

虽然使用12列的网格并没有什么不妥，但是如果在每个站点中都使用12列的网格就会产生让人感到厌烦、可预测的布局。为了能够从网格布局那里获得最大的益处，你得针对每个项目重新考虑一下框架。

大胆地去合并那些列，并实现一个3列或者5列的网格吧！那些设计得最为漂亮的Web站点，没有一个会使用被人们用烂了的12列布局的——有时越简单越好。

2.3.1 从内容出发

使用网格布局要做的第一件事就是确定画布。在图形设计中，画布就是你的纸，它的大小决定了网格。你得先把画布划分为若干你需要的列（3列、5列、9列，甚至12列），然后才能继续后面的工作。

正如我们之前讨论过的，在Web上没有像在纸上那样准确的尺寸，你得工作在内容之外，并让内容来决定网格。

为了清楚起见，这里需要说明的一点是，当我说“内容”的时候，我并不是特指文本，我说的内容有很多种形式：广告、视频、图片、文本。这些不同类型的内容都可以决定你的网格。例如，如果你的公司是一家出版公司，并且收入主要来自于广告，那么比较明智的做法是围绕IAB的一两个特定的广告大小来确定你的网格。同样，如果你正在重新设计一个有很多遗留图片的大型网站，那么你就要根据这些图片来确定你的网格。

让你的内容来决定你站点的结构是一种很好的设计方法，而且这样做也是非常实用的。相比把遗留的图片或者广告硬塞进一个事先就定义好的网格中，不如从一开始就让你的网格来配合它们，这会让每个页面的设计都更具粘性。

已经唠叨得够多了，下面让我们来动手写一些代码吧。

2.3.2 设置网格

◆ 提示

可以访问http://implementing responsivedesign. com/ex/ch2/ ch2-start.html来查看实际效果。

让我们从一个虚构的体育刊物——《另一个体育网站》（原创的，这我知道）开始吧。我们将为文章页面设计一个网格，默认的颜色和字体样式已经写好了（出于示例的考虑，我们要等到下一章才会引入页面的header和footer）。让我们先来看一下需要处理的内容：

```
1.  <body id="top">
2.       <div id="container">
3.            <article class="main" role="main">
4.                 <h1>That guy has the ball</h1>
5.                 <p class="summary">In what has to be considered a
                   development of the utmost importance, that man over
```

```
                there now has the ball.</p>
6.              <p class="articleInfo">By Ricky Boucher |
                <time>January 1, 2012</time></p>
7.              <section>
8.                  <img src="images/football.jpg" alt="Football" />
9.                  <p>...</p>
10.             </section>
11.         </article>
12.         <aside>
13.             <section class="related">
14.                 <h2>Related Headlines</h2>
15.                 <ul>
16.                 <li>
17.                     <a href="#">That Guy Knocked Out the Other
                        Guy</a>
18.                 </li>
19.                 ...
20.                 </ul>
21.             </section>
22.             <section class="ad">
23.                 <img src="images/ad.png" alt="Boombox ad unit" />
24.             </section>
25.             <section class="article-tags">
26.                 <h2>Tagged</h2>
27.                 <ul class="tags">
28.                     <li><a href="#">Football</a></li>
29.                     ...
30.                 </ul>
31.             </section>
32.             <section class="soundbites">
33.                 <h2>Sound Bites</h2>
34.                 <blockquote>
35.                     ..this much is clear to me. If I were in his
                        shoes, I would have already won 5
                        Super Bowls.
36.                     <cite><a href="#">—Guy with big
                        ego</a></cite>
37.                 </blockquote>
38.             </section>
39.         </aside>
```

```
            <div class="more-stories">
                <h2>More in Football</h2>
                <ul class="slats">
                    <li class="group">
                        <a href="#">
                            <img src="images/ball.jpg"
                            alt="Look, it's a ball!" />
                            <h3>Kicker connects on record 13 field
                            goals</h3>
                        </a>
                    </li>
                    ...
                </ul>
            </div>
        </div><!-- /#container -->
</body>
```

我们是水平最高的开发者，所以对于要在页面上显示什么内容我们早已成竹在胸，而且还为我们的设计创建了牢固的HTML结构。在我们的页面中会包括一篇文章，每篇文章又会包括一个用h1元素表示的大标题、作者名字以及一则概述，之后便是放在section元素内的文章正文部分了。

每个文章页面还会有一个边栏，里面包含一个最近文章标题的无序列表。由于《另一个体育网站》如果不通过广告来收费的话，那么它还需要去寻找别的赚钱途径，因此每个文章页面还需要有一个300px × 250px的广告空间。这是我们遇到的第一个限制，我们可以根据该限制来设计网格。

最后，边栏中还需要有一个与文章相关的标签列表，并引用一些他人的言论。其中标签以无序列表的形式展示，而引用则要放在blockquote元素当中。

让我们先来用一种古老而传统的方法来创建我们的网格——以像素为单位。

960px，让我们尝试使用9列网格来取代经常使用的12列网格。9列网格的每一列都为84px宽，而且它们之间还有24px的间隔，共计948px。一个300px宽的广告可以很容易地放到网格的最后3列当中去，剩下的6列则留给文章内容使用。

◆ 提示

可以在https://github.com/robbiemanson/960px-Grid-Templates查看Robbie Manson在Github上的代码仓库，里面有可以在Photoshop和Fireworks中使用的各式各样的模板。

首先，我们来设置容器元素的宽度：

```
#container{
      width: 948px;
}
```

然后让文章和边栏分别向左、右浮动，并为它们指定适当的宽度：

```
aside {
      float: right;
      width: 300px;
}
.main {
      float: left;
      width: 624px;
}
```

到此为止，布局看起来已经相当可爱了。网格让设计中的元素看起来是有联系的，而且文章的宽度也非常利于阅读，这会使用户的阅读过程非常轻松。

但是，当你缩小浏览器窗口时，问题便会马上暴露：当浏览器宽度小于948px的时候，我们又看到了讨厌的水平滚动条，屏幕中又不能同时显示所有的内容了（图2.5）。

◆ 提示

可以访问http:// imp-lementing respon-sivedesign. com/ex/ch2/ ch2.2.html来查看实际效果。

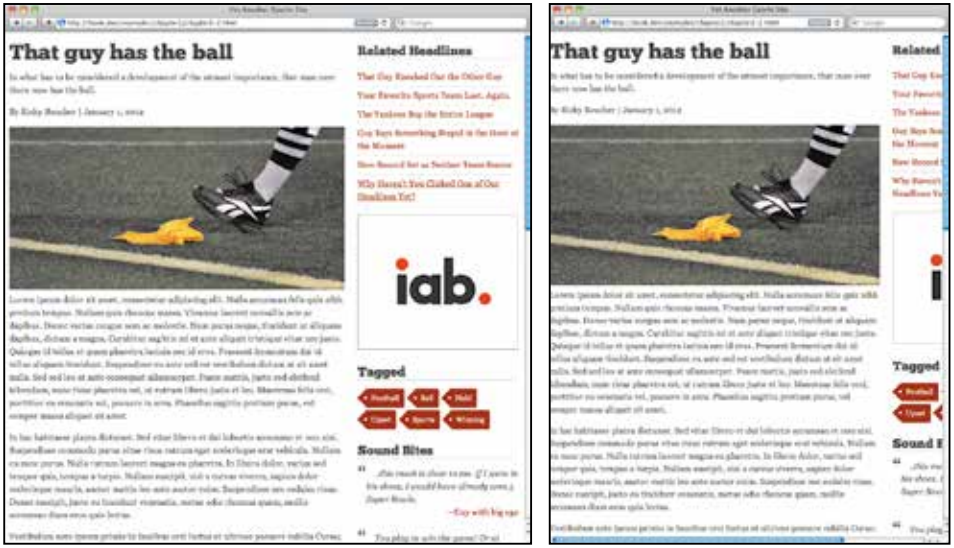


图2.5 页面在大屏幕上看起来很好，但当浏览器改变大小后问题就出现了。

虽然我们已经设计出了一个不错的网格，但它目前还只适用于一小部分用户使用，下面让我们来做一些弥补吧，好吗？

让网格变得灵活

如果此时你想起了前面让字体变灵活的方法，那么其实你可以把同样的方程式（目标/上下文=结果）应用在灵活化网格布局上来。

容器的上下文是948px，在这样的容器里实现流动布局是件很容易的事：

```
aside {
     float: right;
     width: 31.6455696%; /* 300/948 */
}
.main {
     float: left;
     width: 65.8227848%; /* 624/948 */
}
```

Box-sizing

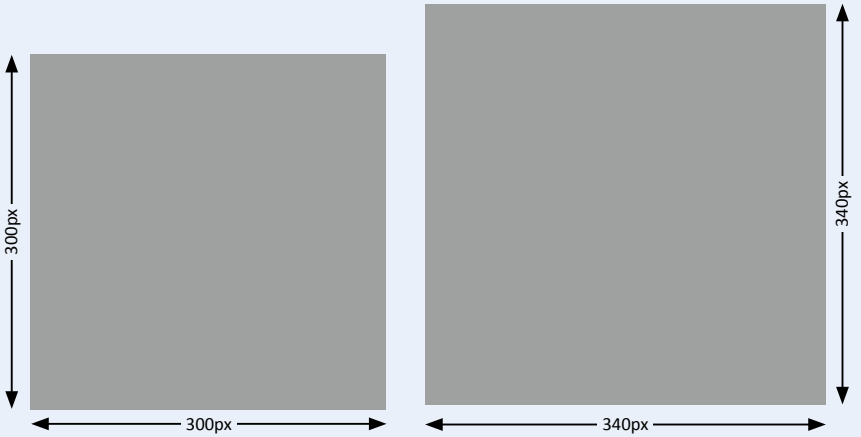
如果你看过上面这个页面的样式表，那么也许你会注意到下面这3行代码几乎应用到了所有元素上：

```
-moz-box-sizing: border-box;
-webkit-box-sizing: border-box;
box-sizing: border-box;
```

CSS中默认的盒模型会让人觉得有问题：你本来已经定义好了width，但你之后定义的所有padding又都会被算到这个width中去。例如，如果你创建了一个300px宽的列，之后在其左右各加了20px宽的padding，那么此时该列的宽度就会变成340px。 这在人们创建基于网格的流动布局时，会使人感到相当痛苦。

通过使用box-sizing: border-box，你就可以让浏览器将padding的值算在已经定义好的元素宽度内部了。通过使用这一属性，一个300px宽的列即使两边新增了

20px的padding，它的宽度依然会是300px，因为padding的宽度被算在了元素内部。这使得设计流动布局变得更加容易。



在不使用box-sizing的情况下（如右图所示），一个宽为300px且有20px padding的元素，其宽和高都会变为340px 。而在使用了box-sizing后，元素仍然是300px宽、300px高。

在包含了box-sizing的各种版本的前缀之后，这一属性就可以获得很好的浏览器支持了。在桌面及移动设备的主流浏览器中，只有IE8之前的IE浏览器对此缺乏支持。实际上，大多数浏览器对此已经支持了很长一段时间了，比如Chrome和Firefox，从第一个版本开始就支持前缀语法了。

最后，我们来让页面中的其他部分也流动起来。

```
.main {
    float: left;
    margin-right: 2.5316456%; /* 24px / 948px */
    width: 31.6455696%; /* 300/948 */
}
```

现在，内容栏和边栏都实现了流动布局，下面我们要将容器上现有的固定宽度也去掉。我们将页面容器的总宽度设置为95%来取代之前的948px，留一点padding（译者注：指剩下的那5%）能给页面一些呼吸的空间：

```
#container{
```

▶注意

为什么是95%？其实没有什么确切的原因。我试了几种不同的宽度，结果95%在不同的浏览器宽度下都显示得非常好。有时候，设计中真的就是靠看和感觉来定的。

```
        width: 95%;
        padding: .625em 1.0548523% 1.5em; /* 10px/16px, 10px/948,
        24px/16px */
        margin: auto 0;
    }
```

在这个例子中，我们使用em作为顶部和底部padding的单位，并使用百分比作为左右两边padding的单位，之所以要这样做是由上下文所决定的。因为顶部和底部的padding值是由font-size的大小决定的，所以使用em更有意义。

流动世界中的固定宽度对象

下一个要讨论的对象就是图片。当你缩放浏览器窗口时，有着固定宽度的图片就会变得极为醒目，因为与周围其他流动的内容相比，它们会显得格格不入。在较宽的列中它们只会占据一小部分；而在较窄的列中它们又太宽。不过谢天谢地，让它们变整齐也是很容易的。

我们要做的第一件事，就是使用下面的声明来让图片填满整个边栏：

```
aside img,
.main img,
.slats img {
        width: 100%;
}
```

这里需要注意的很重要的一点是，不能在HTML中设定img元素的height和width的属性值，因为如果设定了这些值，就不能按比例缩放图片了。为了使流动图片正常工作，你只能通过CSS来控制图片的大小。

然后我们需要声明max-width。通过将max-width设置为100%，我们便可以告诉浏览器不要让图片的大小超过了它的容器元素（本例中是边栏）。这样，即使浏览器窗口变窄，图片也不会溢出或被裁剪了：

```
aside img,
.main img,
.slats img {
        width: 100%;
        max-width: 100%;
}
```

至此，我们便拥有了一个流动布局——能够自适应地调整自己，而且在大部分的设备上都是可用的（图2.6）。在提升有关图片的体验方面我们还有很多事可以做，我们将在第4章中再对此做详细介绍。

◆ 提示

可以访问http://implementing responsivedesign.com/ex/ch2/ch2.3.html来查看实际效果。

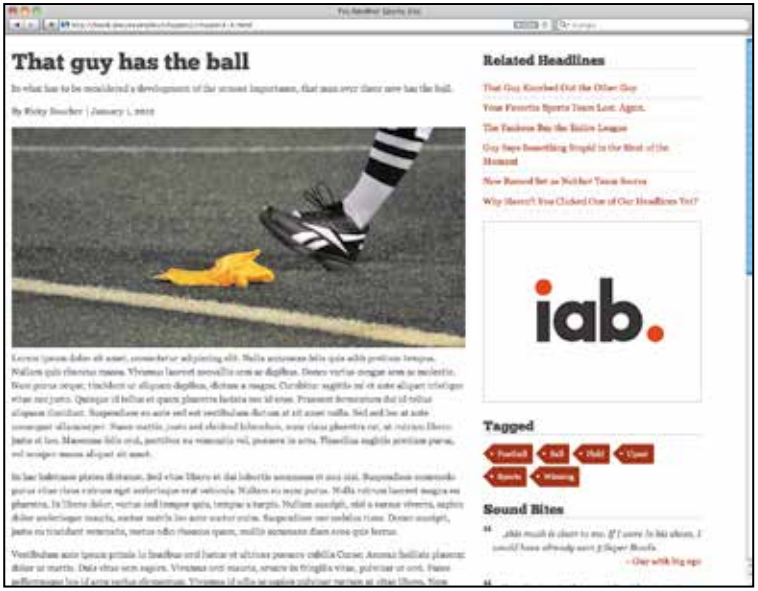


图2.6 即使屏幕大小不是预料的那样，新的流动布局看起来依然非常棒。

2.4 混合固定宽度和流动宽度

到目前为止，文章页面看起来已经很不错了，而且布局十分灵活，下面让我们来加强一下右侧的边栏。现在它看起来好像没什么问题，但是如果我们能让它保持300px宽，而只让主体列流动岂不是更好？虽然这不是必需的，但是考虑到边栏里的广告，我想这将会是一次很好的润色。

使用浮动来实现这个任务几乎是不可能的。正如我们之前讨论过的那样，主要内容栏的宽度依赖于屏幕的分辨率，如果我将边栏设置为固定的300px宽，同时保持主要内容栏为当前的63.125%，那么我们将遇到前面在浏览器宽度小于960px时遇到的同样的问题。

有一种方法可以绕过这个问题，其中会涉及CSS表格。

表格布局——正确的方法

不久以前，在一个离我们不太遥远的星球，那里多数的Web站点都是使用表格来布局的。那是一种缺乏语义的、散乱到让人看了想哭的实现方法，但它的确是有效的。后来那里出现了一场Web标准化运动，并提出了要将内容和表现相分离的理念，突出强调了使用语义标记的重要性。一场伟大的战役便随之而来，并最终以标准的获胜而告终。

相比CSS布局，表格布局的优势之一在于它简化了将站点布局为多列的实现过程。可以混合使用固定宽度和流动宽度；一行里可以包括好几列——所有这些在表格布局中实现起来都相当容易，而用CSS来实现这些样式则远没有那么简单。

其实你可以给display属性赋予许多不同的、与表格相关的值，并以此来实现上述的那些样式，因为display的一些属性值可以让元素获得与表格相关元素相似的布局效果。

表2.1 与表格相关的display值

值	相应元素	值	相应元素
table	TABLE	table-column	COL
table-row	TR	table-column-group	COLGROUP
table-row-group	TBODY	table-cell	TD, TH
table-header-group	THEAD	table-caption	CAPTION
table-footer-group	TFOOT		

如果在CSS中使用表格值的做法会让你觉得不爽，那么我想你不该因此而受到指责。毕竟，基于表格（table）的布局曾一度受到了人们猛烈的抨击，你也许非常理解那种甚至看到厨房里的桌子（table）都会感到恶心的感受。但是，在CSS中使用表格值与使用HTML表格来布局还是有很大区别的，CSS中的表格值只是给内容定义了视觉样式，而并不是说表格也是内容的一部分。

到目前为止，display属性的表格值还没有得到广泛的使用。对此你也许可以指责IE，因为Firefox、Safari以及Opera都已经支持表格值有一段时间了，而IE直到IE 8才开始支持表格值。在写这本书时，IE 6和IE 7总共的市场份额已经下降到了5%以下，所以我想现在是时候扫扫CSS表格值身上的尘土，并开始使用它们了，更何况移动平台对此的支持也相当棒。

如果将某列的display属性指定为table-cell，我们就可以混合分别使用固定宽度和流动宽度的列了：

```
.main {
     display:table-cell;
}
aside {
     display:table-cell;
     width: 300px;
}
```

这时当我们再来缩放浏览器，边栏将仍旧保持300px，而左边的主要内容栏则会进行调整以填满剩余区域。虽然现在两列之间没有了漂亮的间隔，但我们可以很容易地通过增加一些padding来让它重现：

```
.main {
     display:table-cell;
     padding-right: 2.5316456%; /* 24px / 948px */
}
```

现在，我们已经能够将固定宽度的列和流动宽度的列结合在一起使用了，这使得我们在保证布局灵活性的同时，不必再去担心在混合布局中引入浮动时会造成的混乱了（图2.7）。但主要内容栏在高分辨率的屏幕中显示时还是会有一些不妥之处（译者注：因为在高分辨率屏幕下主要内容栏会变得很宽，过宽的行不利于人们阅读），我们将在下一章探索媒介查询时再去解决这个问题。

对老版本IE的支持

对于很多站点来说，也许本章到此就可以结束了，因为IE 8之前的各种版本的IE正在迅速失去各自的市场份额。其实这仍然是由你的用户来决定的，让那些旧版本的IE以它们现有的方式来呈现网站也许是不够的。虽然可以显示网站的内容，但你的客户也许并不满足于这样的设计，这时你就要准备一些备用的样式了。

条件注释可以帮你达到上面的目标，因为条件注释能让特定版本的IE浏览器使用另外的样式表。例如，我们创建一个叫做ie.css的样式表，并且规定只在IE 7及以下版本的IE中才加载它，那我们就可以使用下面这样的条件注释：

```
<!--[if lt IE 8]>
<link rel="stylesheet" href="/css/ie.css" media="all">
<![endif]-->
```

现在，任何IE 8之前版本的IE就都会加载ie.css了。这样，我们便实现了为较早版本的IE浏览器提供备选样式表的目标。

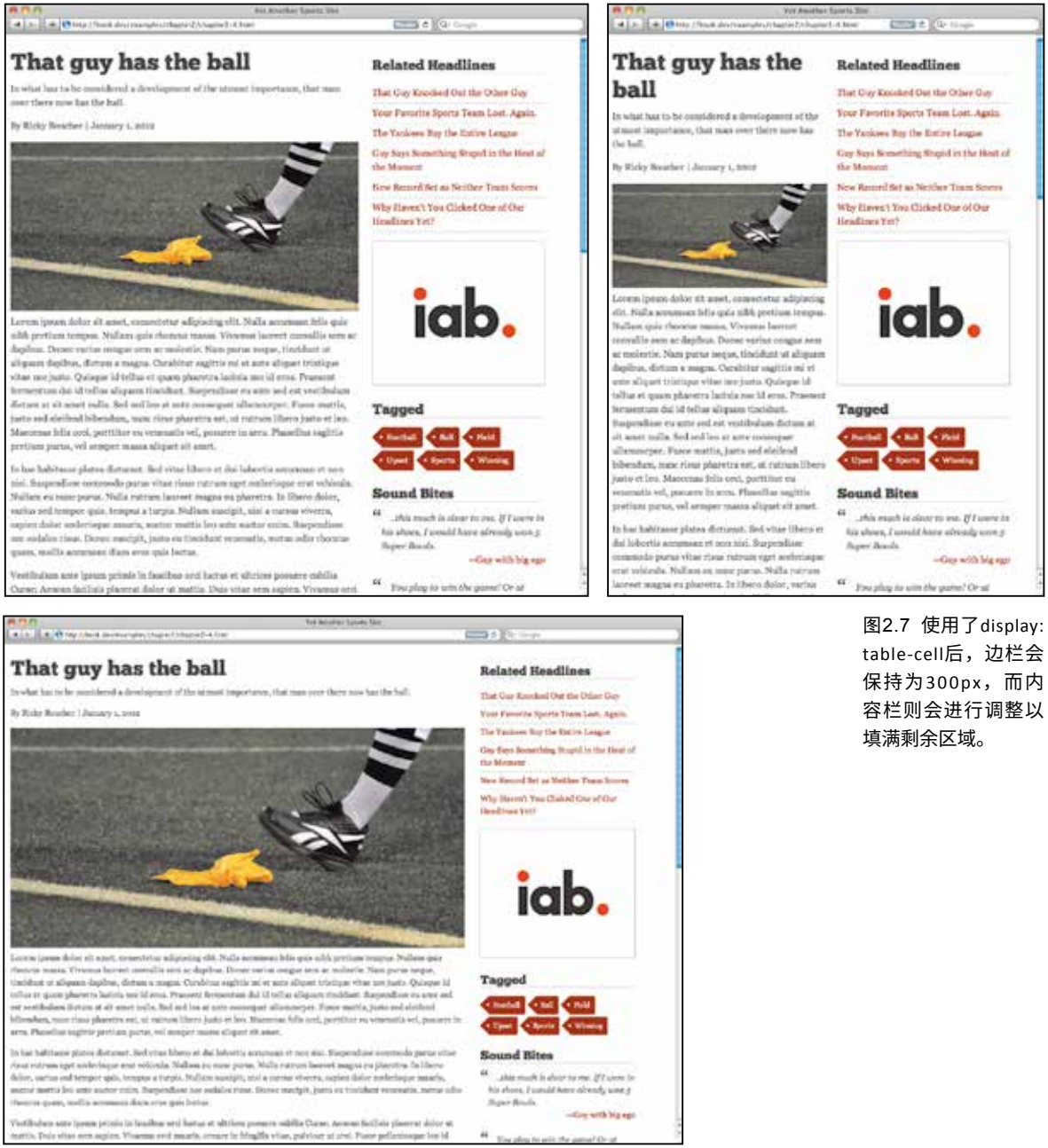


图2.7　使用了display: table-cell后，边栏会保持为300px，而内容栏则会进行调整以填满剩余区域。

Display:table的告诫以及未来

在你变得兴奋并开始在所有项目中都使用display:table之前，这里有一些你需要意识到的潜在问题 。

第一个问题是：在一个被指定为display:table-cell的元素内，你无法精确地定位元素。如果你需要精确地定位，你就不得不在表格中插入另外一个div，或者干脆就不要使用display:table。

另一个需要牢记的一点是：相对来说，表格是更加严格的。有时浮动所具有的流动性是有利的，比如如果有些内容过长的话，那么浮动可以很容易地让超出的部分折回到下面去。

Web设计中没有银弹（译者注：银弹是指那些可以解决复杂而棘手的问题的方法或者技术手段），你会看到我在后面还会提到这句话的。所以你必须在提交任何方案之前，仔细考虑你的需求，包括使用display:table。

CSS网格布局和Flexbox是两种新的布局方法，它们可以为我们提供更多种类的控制，而且值得我们关注。但现在浏览器对它们的支持还都非常有限，这也是我们使用display:table的原因。

现在唯一的问题是Windows Phone 7也会加载这个备用样式，鉴于此，我们将在下一章中利用媒介查询来针对小屏幕修改样式表，因为我们不想在手机中让备用的样式表覆盖现有的样式。幸好我们只需在条件注释中做一处小小的修改，就可以修复这个问题了（该方法最先由Jeremy Keith提出）：

```
<!--[if (lt IE 8) & (!IEMobile)]>
<link rel="stylesheet" href="/css/ie.css" media="all">
<![endif]-->
```

既然我们可以在不影响手机体验的前提下提供备选样式，那么我们就先将ie.css文件中的样式变回两列浮动的流动布局：

```
.main {
     float: left;
```

```
        width: 65.8227848%; /* 624/948 */
    }
    aside {
        float: right;
        width: 31.6455696%; /* 300/948 */
    }
```

虽然在老版本IE中达不到那些对标准支持更好的浏览器中显示时的效果，但这已经足够了。记住，站点不需要在不同设备的不同浏览器中看起来都一模一样，事实上这也是无法做到的。即便是现在这样，那些老版本IE的用户也照样可以看到一个对于他们的浏览器来说合适且良好的布局。

◆ 提示

可以访问http://implementing responsivedesign.com/ex/ch2/ ch2.4.html来查看实际效果。

2.5　结束语

大多数情况下，流动布局（以百分比为单位，因而可以根据屏幕大小来调整自身的布局）是你设计站点时的最佳选择。当宽度被字体大小限制时，你可以使用弹性布局；当宽度被百分比限制时，你可以使用流动布局。

采用更加灵活的方式来指定文字大小可以使维护工作变得更加容易，同时这样做也提升了站点的可访问性。为了达到这个目标，我们就要坚持使用百分比和em，尽管rem在未来很有潜力。

通过定义网格，有助于为你的网站提供良好的结构并增加一致性。要试着从内容出发来建立你的网格，而不是随意拿一个事先就定义好的网格来用。这就意味着我们要根据行宽、图片、广告大小或者其他标准来建立网格。

将固定单位转换为灵活的单位与计算一道除法题一样简单，因为在指定元素宽度以和字体大小时，你用的都是那个方程式。

通过使用CSS表格，你可以很容易地将固定宽度与流动宽度结合使用，现代的桌面浏览器对此特性都有着极为出色的支持，而且你还可以通过使用条件注释来为IE 7及以下版本的IE浏览器提供备用样式。

现在，“ 另一个体育网站 ”的文章页面的布局已经变得灵活了许多，而且相比采用固定布局也能够适应更多种类的分辨率了。然而它现在还不是真正响应式的，因为当浏览器变得很窄的时候，我们依然会遇到很多样式上的问题，而且如果屏幕太宽的话，我们的设计也不是特别整洁。

在下一章中，我们将使用媒介查询来解决上面这些问题。媒介查询使我们可以根据用户使用的设备来改变页面的样式，这一强大的技术将带领我们走向真正的响应式设计。

第3章
媒介查询

你必须像水那样无形：当你把水倒入杯子中，水就变成了杯子的形状；当你把水倒入瓶子中，水就变成了瓶子的形状；当你把水倒入茶壶中，水就变成了茶壶的形状。

——李小龙

你有没有吃过花生酱三明治？对，没错，花生酱三明治——两片面包之间没有果酱，只有花生酱。

这完全是可以食用的，而且至少肯定要比两片面包之间什么都没夹要好，但它也不是特别令人满意的。正如你知道的那样，它缺少了某种成分——一种能使整个事物变得更好的成分。

是的，你需要果酱。

在响应式设计中，媒介查询就是果酱（我把它想象成是草莓味的，你可以把它想象成任何你所喜欢的味道）。

流动布局是个伟大的开端，它消除了固定布局中的种种限制，并使站点能在不同分辨率的屏幕上都能漂亮地展示，但是它也只能带你走这么远。

媒介查询可以让你根据在特定环境下查询到的各种属性值——比如分辨率、色彩深度、高度和宽度——来决定应用什么样的样式。通过使用媒介查询，你可以熨平以前的布局中的所有褶皱。

在你阅读完本章之后，你将能够：

- 为你的站点设定视口；
- 使用媒介查询来调整站点的设计；
- 组织并使用媒介查询；
- 设定必要的断点；
- 为小屏幕设备提升导航栏的使用体验。

▶注意
可以访问http://implementingresponsivedesign.com/ex/ch3/ch3.1.html来查看实际效果。

我们上一次提到文章页面时，它利用display:table创建了一个流动布局：边栏采用固定宽度，而主要内容列和外层容器则都是以百分比为单位的，这样一来，页面就可以根据屏幕的尺寸做出相应调整了。

当你现在再次打开文章页面时，你会看到header和footer奇迹般地出现了，此外我们还为站点增添了其他的样式和结构，现在的文章页面看起来如图3.1所示。

图3.1 “另一个体育网站”的页面有了动感而时尚的header和footer。

对于大多数宽度来说，现有的页面已经能够很好地显示了，但是经过一番仔细检查之后，我们又发现了一些新的问题。

当浏览器窗口被调整到非常宽的时候，文章正文的行宽也会随之增大。我们把浏览器调整得越宽，每行文字的行宽就离理想的阅读宽度越远。当然也并不是所有东西都会变得很糟，至少布局保持得还相当不错。

当浏览器窗口越变越窄时，我们可爱的布局又会看起来像被砖拍扁了似的，其实窗口还不是特别窄时，有些导航栏的条目就已经折到下一行去了（图3.2）。这些虽然不是网站致命的弱点，但总显得不是特别优雅，而且当我们缩小浏览器窗口后，主要内容列的行宽也略显窄小。记住，我们的理想行宽介于45到70个字符之间，任何的多余或不足都会给阅读体验带来负面的影响。

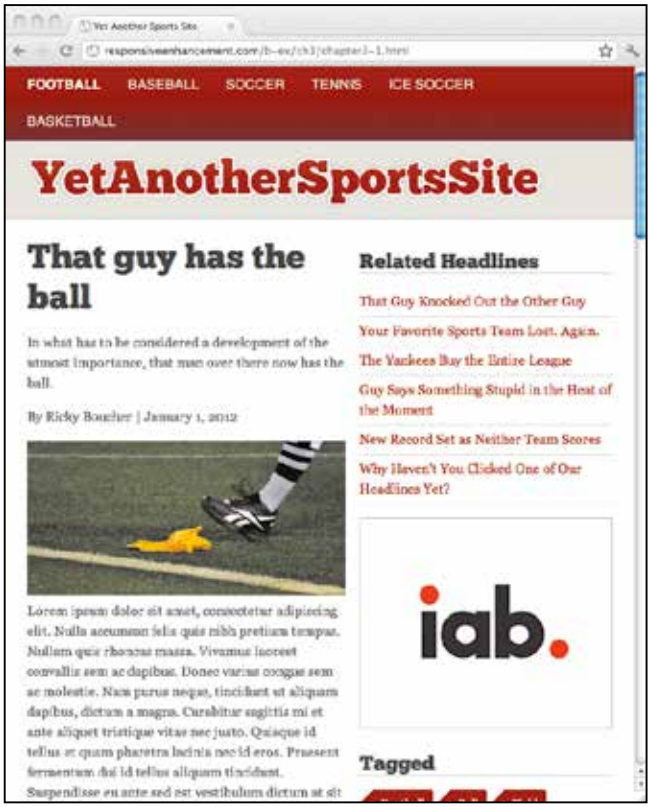


图3.2 浏览器窗口变窄导致布局被破坏了。

随着窗口继续变窄，问题也越来越严重了。当宽度缩小至大约360px时，导航栏已经乱作一团了，内容栏的每行几乎只能容纳三个字，而且边栏也非常狭窄。当然，我们需要对此采取一些对策。

你也许会认为，这样狭窄的窗口可以用来模仿访问者在移动设备上浏览网站时的情形，因为移动浏览器看起来也就是那么宽的，但是，你错了（图3.3）。

图3.3　在智能手机上浏览时，站点被缩小后的样子。

如果你在大多数的智能手机上浏览这个文章页面，你不会遇到缩放浏览器时遇到的那些问题。相反，页面会维持一开始时的布局，只不过被缩小了——整个站点都变得非常小。为了理解为什么会发生这样的事情，我们需要先凑近了好好观察一下那些屏幕中的小方块——像素。

3.1　视口

在桌面浏览器中，视口是一个非常简单的概念：视口就是浏览器的可视区域，也指浏览器的宽度。它是如此的简单，以至于实际上没有人会因为弄不懂它而感到

困惑。但是一切都因为手机而发生了改变。尽管手机的屏幕很小，但为了给用户能提供一种完整的Web体验，手机会尝试显示一个“完整的”站点，事情一下子就变得更加复杂了。

3.1.1 像素就是像素，除非它不是像素

当提到浏览器时，我们通常会涉及两种像素：设备像素和CSS像素。设备像素的行为正如你所预料的那样：如果你的屏幕的宽度是1024px，那么你可以在里面并排地放置两个512px宽的元素。

CSS像素则没有那么确定。CSS像素与屏幕无关，但与浏览器窗口内的可视区域有关，这意味着CSS像素并不一定与设备像素一一对应。在许多设备上，一个CSS像素对应一个设备像素，而在另外一些像拥有Retina这样高分辨率屏幕的iPhone等设备上，一个CSS像素实际上等于两个设备像素。等一下，这里还有更有趣的事情！

当用户缩放某一页面时，CSS像素也会跟着变化。例如，如果用户将页面放大至300%，则CSS像素的宽和高也会变为原始值的三倍大；如果用户将页面缩小至50%，则CSS像素的宽和高也会减半。但在整个过程中，设备像素——屏幕——是不会发生变化的，毕竟屏幕只能是那么宽，然而CSS像素却变化了。确切地说，是在浏览器窗口内可以被看到的像素数发生了变化。

你需要在视口中考虑这两种不同的像素。再强调一遍，你有两种不同的视口要考虑：布局视口和视觉视口。

布局视口与设备像素非常相似，因为它们的度量单位都是一样的，它们与设备的方向以及缩放的级别都没有关系。但是视觉视口是可以变化的，视觉视口指的是在屏幕上显示的那一部分页面（图3.4）。

在移动设备上，这便成为了一件非常复杂的事情。为了使用户获得“完整Web”的体验，很多设备都会给浏览器返回一个数值较大的布局视口。例如，iPhone的布局视口为980px，Opera Mobile为850px，Android WebKit为800px。这就意味着如果你创建了一个320px宽的元素，那么它在iPhone上将只占屏幕实际大小的1/3左右。

图3.4　移动设备有两种差别很大的不同视口。

3.1.2　视口标签和属性

幸好在WebKit把我们从中解救了出来之后，其他的渲染引擎也都开始纷纷效仿：通过视口元标签，我们可以控制页面在很多设备上的大小和布局视口。

视口标签的形式非常简单，只需指定使用的视口元标签，然后列出一些声明即可：

```
<meta name="viewport" content="directive,directive" />
```

该元标签需要放在HTML文件的head标签中：

```
<head>
    <meta name="viewport" content="directive,directive" />
</head>
```

让我们再来看一下视口的属性，看看我们有哪些东西可以利用。

● **渲染引擎**
浏览器中负责读取标记语言（HTML、XML等）和样式信息（CSS、XSLT等），并将格式化后的内容显示在屏幕的组件。

▶注意

请务必包括引号和嵌套等号。写成<meta name="viewport" width="device-width"/>是不规范的，你需要加上content=""。

Width

Width声明可以让你将视口设置为某一特定的宽度，或者设置为设备屏幕的宽度：

```
<meta name="viewport" content="width=device-width" />
```

在width声明中使用device-width是最佳的选择，因为这样设置之后，页面在屏幕上的布局视口将等于设备的屏幕宽度——设备像素。

但如果你指定了一个特定的宽度，例如240px，那些宽度不是240px的设备只能通过拉伸页面来匹配其屏幕。所以如果你的设备宽320px，为了尽可能整洁地显示页面，所有东西都会被放大1.33（320/240）倍（图3.5）。

图3.5 左图为width被设置为了iPhone的屏幕宽度，即320px。右图中的width被设置为了与屏幕宽度不同的另一特定宽度，我们看到所有的东西都被放大了。

因此，你最好永远都不要为width指定一个绝对的数值，而要使用device-width。

Height

与width对应的就是height，height允许你指定一个特定的高度：

```
<meta name="viewport" content="height=468px" />
```

该声明会将高度设置为468px。与width声明类似，这里也有一种万无一失的设置高度的方法，那就是将height设置为device-height：

```
<meta name="viewport" content="height=device-height" />
```

这将使布局视口的高度等于屏幕的高度。在实践中，你也许并不会经常用到height，因为height唯一让人觉得有用的时候，是在你不想让页面垂直滚动的时候，但是这种情况并不多见。

User-scalable

user-scalable声明会告诉浏览器，是否允许用户在页面上进行缩放：

```
<meta name="viewport" content="user-scalable=no" />
```

你经常会看到页面的user-scalable属性被设置成了no，尤其是在那些追求“完美到像素”的设计中更是如此。但其实这样做不仅违背了Web的本性，还损害了用户对于可访问性的需求。如果你不设置user-scalable属性，那么它的默认值会是yes，所以最好还是明确地声明一个值。

iOS方向漏洞

开发者之所以会去使用user-scalable或maximum-scale，是因为iOS中有一个长期存在的漏洞。

如果你把视口设置为任意大小并且允许用户缩放页面的话，那么当你将设备横屏之后，页面就会自动变大，这会迫使用户不得不点击屏幕两次来使页面得到正确的缩放，从而避免页面被裁剪。

如果你禁用了缩放，那么无论是使用maximum-scale还是user-scalable，问题当然就不存在了。但不幸的一点、同时也是更重要的一点是，这时你的页面就变得不是那么容易使用了。

幸好这个问题在iOS6中被修复了。对于老版本的iOS而言，来自Filament Group的Scott Jehl通过利用设备的加速度计来确定设备何时方向发生了变化，并针对该问题做了集中的修复：当设备的方向发生变化时，会暂时禁止用户缩放，直到方向不再变化时，缩放功能又会恢复。

你可以在GitHub上找到并免费下载到这一聪明的修复：https://github.com/scottjehl/ios-orientationchange-Fix。

Initial-scale

你可以通过给initial-scale赋值为0.1（10%）到10.0（1000%）之间的某个数，来设置页面初始化时的缩放层级：

```
<meta name="viewport" content="initial-scale=1, width=device-width />
```

在使用了上面的声明之后，如果设备屏幕的宽度是320px的话，那么页面也将会是320px宽；如果设备屏幕的宽度是200px，那么页面也将会是200px宽。

让我们再来看另外一个例子：

```
<meta name="viewport" content="initial-scale=.5, width=device-width" />
```

在上面的这个例子中，width属性被设置为了设备的宽度，同时initial-scale被设置为了.5（50%）。这意味着浏览器将会缩小显示所有东西：在一个320px宽的设备上，页面会显示为640px（图3.6）；在一个200px宽的设备上，页面会显示为400px。

图3.6 将initial-scale设置为1时，在320px宽的设备上页面是正常大小的（左图）；当设置为.5时，页面被缩小了（右图）。

Maximum-scale

Maximum-scale声明可以告诉浏览器允许用户放大页面到什么程度。在移动端的Safari中，其默认值是1.6（160%），但其实你可以指定一个介于0.1（10%）到

10.0（1000%）之间的任意数字。

和user-scalable一样，如果你将maximum-scale设置为了1.0，那么你也同样禁止了用户缩放页面，这样做也同样会降低站点的可访问性。

Minimum-scale

Minimum-scale声明会告知浏览器允许用户将页面缩小到什么程度。在移动端的Safari中，其默认值是0.25（25%）。你可以将minimum-scale设置为介于0.1（10%）到10.0（1000%）之间的任意数字。

如果你将minimum-scale设置为了1.0（100%），那就意味着你禁止用户缩小页面。正如你之前看到的，这同样会降低可访问性，所以应该避免这样使用。

修复视口问题

在了解了视口元标签及其声明之后，你可以通过使用设备宽度来摆脱需要“放大”页面后才能查看这种情况，给width赋值为device-width即可实现这个效果：

```
<meta name="viewport" content="width=device-width" />
```

CSS设备适配

事实证明，视口元标签实际上是不规范的。简单地说，它还不是一个确切的标准。如果你看过了W3C的文档你就会发现，其实该标准现在仍处于草案阶段，该草案同时也为浏览器整合新的@viewport语法提供了路线图。

@viewport规则可以让你在CSS中直接使用视口元标签中的那些描述符（width、zoom、orientation、resolution，等等）。例如，你可以用下面的CSS代码将视口设置为设备宽度：

```
@viewport {
     width: device-width;
}
```

目前对于该语法的支持还仅限于Opera和IE 10中带有前缀的实现，然而从视口元标签目前的发展情况来看，有关它将会被从浏览器中剔除的猜想应该是合理的，取而代之的是浏览器对@viewport规则的整合。

现在当我们再在移动设备上加载页面时，页面的行为就和我们在桌面浏览器中缩放浏览器窗口大小时的行为一样了，因为现在手机使用它自身的宽度来作为视口宽度。在这里我们没有使用任何其他声明，因为它们都不是必需的，而且我们也不想为了控制使用环境而落入降低可访问性的陷阱中去。

没有敏锐的眼光你也能发现——事实上设置视口后我们使情况变得更糟了（图3.7）！我们的站点看起来还是像被揍扁了一样，是时候呼叫我们的朋友——媒介查询——来帮忙了。

图3.7 当设置了视口宽度后，站点会像在桌面浏览器中一样显示，只不过被缩小了。

3.2 媒介查询结构

媒介查询可以通过问询浏览器来确定特定的表达式是否为真。如果为真，那么就加载一些特殊的、适用于这种情况的样式，从而达到调整布局的目的。

媒介查询的一般形式为：

```
@media [not|only] type [and] (expr) {
     rules
}
```

一条媒介查询包含以下四个基本组成部分：

- 媒介类型：特定的目标设备类型。
- 媒介表达式：测试某一特性是否为真。
- 逻辑关键词：你可以使用关键词（例如and、or、not、only）来创建出更多复杂的表达式。
- 规则：调整显示效果的基本样式。

让我们来逐个研究一下它们。

3.2.1　媒介类型

Web的众多奇妙特点之一，就是它具有可以为多种不同媒体提供内容的能力。Web的呈现方式远远不止屏幕一种：信息可以被打印出来，也可以通过触觉反馈装置、语音合成器、投影仪、电视等其他平台来访问。

媒介类型的出现为这场混乱带来了秩序。其基本用法是只使用一种媒介类型，而不是写出整个媒介序列。其实如果你之前创建过用于打印的样式表的话，那么其实你已经使用过媒介类型了。

每一种媒介类型都会告诉用户代理（例如浏览器）是否要加载特定的样式表。例如，如果你指定的是screen媒介类型，那么所有通过计算机显示器来浏览页面的用户代理都会加载相应的样式；如果你使用的是print媒介类型，那么相应的样式将会在打印或打印预览时进行加载。

CSS中定义了10种不同的媒介类型，如表3.1所示。

表3.1 媒介类型

类型	目标设备
all	所有设备（默认）
braille	盲文触觉反馈设备
embossed	分页盲文打印机
handheld	手持设备（通常为小屏幕并且可能是黑白屏幕）
print	打印或打印预览
projection	投影仪
screen	彩色计算机屏幕
speech	语音合成器
tty	使用固定字符间距的设备（终端或打印设备）
tv	电视机

样式表中的查询语句如下：

```
@media print {
}
```

此外，你也可以采用外部样式文件的形式，并在link元素内指定媒介属性：

```
<link rel="stylesheet" href="print.css" media="print" />
```

无论使用哪种方法，被引用的CSS的效果只有在打印或打印预览时才能看到。

每条媒介查询都必须包含一种媒介类型，如果没有设置媒介类型，该条查询将使用默认值all，但在不同浏览器中的实际行为是各不相同的。

在实际的使用过程中，你会发现你几乎只会用到all、screen以及print。不幸的是，很长一段时间以来开发者们都一直只用screen，而没有使用handheld或者tv，这使得大部分设备都开始支持screen，而不是它们本应该支持的媒介类型。但其实这也不能怪各个生产厂家，因为如果他们不那样做的话，大多数网站就无法在他们的设备上显示了。

就媒介类型本身而言，它只允许你指定设备的类型，但是为了对页面进行进一步

的细分，你需要缩小设备的范围，这时就该媒介表达式登场了。

3.2.2　媒介表达式

媒介查询的强大之处在于它们能利用表达式来检测出设备不同特性的真假值 。例如通过下面这条简单的声明，你就可以判断出设备视口的宽度是否大于320px：

```
@media screen and (min-width: 320px) {
}
```

这条语句检测了两样东西：它会先检测访问页面的设备是否属于screen，然后测试设备的视口宽度——这是表达式做的事，其中min-前缀会保证视口宽度至少为320px。在表3.2中，列出了你可以使用的不同特性，而且min-和max-前缀也都适用于这些特性。

表3.2 媒介特性

特性	描述	值	可以指定最小或最大
width	描述设备显示区域的宽度	〈长度〉（例如320）	是
height	描述设备显示区域的高度	〈长度〉（例如600）	是
device-width	描述设备渲染界面的宽度	〈长度〉（例如320）	是
device-height	描述设备渲染界面的高度	〈长度〉（例如600）	是
orientation	指定设备处于竖直（高度大于宽度）或者水平（宽度大于高度）状态	portrait|landscape	否
aspect-ratio	width属性与height属性的比值	〈比值〉（例如16/9）	是
device-aspect-ratio	device-width属性与 device-height属性的比值	〈比值〉（例如16/9）	是
color	设备的每个颜色分量的比特数	〈整数〉（例如1）	是
color-index	设备的颜色查找表中的条目数	〈整数〉（例如256）	是
monochrome	黑白屏幕设备每个像素的比特数（如果不是黑白屏幕将返回0）	〈整数〉（例如8）	是
resolution	设备的分辨率（像素密度），可以以点每英尺［dpi］或者点每厘米［dpcm］来表示	〈分辨率〉（例如118dpcm）	是

特性	描述	值	可以指定最小或最大
scan	“ tv ”类设备的扫描过程	progressive|interlace	否
grid	返回该设备是网格设备（1）还是位图设备（0）	〈整数〉（例如1）	否

我想你将会经常用到width、height、orientation、resolution这几个值，此外你也许有时也会用到aspect-ratio。

浏览器对于color、color-index、device- aspect-ratio的支持欠佳，目前monochrome、scan、grid也不适用于大多数设备。

3.2.3 逻辑关键词

除了媒介类型和媒介表达式外，你还可以通过使用可选的关键字，来使媒介查询语句具有更加强大的功能。

AND

你可以使用and来测试多个表达式：

```
@media screen and (color)
```

上面的例子会对设备进行测试，以确定其是否配备有彩色屏幕。

NOT

Not关键词会对整个表达式的结果取反，而不是对部分表达式的结果取反。让我们来看看下面这个例子：

```
@media not screen and (color) {...} //equates to not (screen and (color))
```

在上面的媒介查询中，对于任何带有彩色屏幕的设备而言，该表达式都将返回一个false。另外需要注意的一点是，你不能使用not关键字对单个的测试取反——它必须放在其他查询结果之前。

第四级媒介查询

在标准化额外属性方面，人们已经做了一些工作。截止写这本书时，标准中已经增加了另外3个属性：script、pointer和hover。

Script属性会检测浏览器是否支持ECMAscript，如果支持，是否处于激活状态（即没有被禁用）。Script的值可以是1（支持脚本）或者是0（不支持脚本）。

Pointer属性将会检测点击设备的准确性（比如鼠标或者手指）。如果支持多种输入方法，设备应该返回主要输入方法的测试结果。

Pointer属性的值可以是none（无可点击设备）、coarse（有限的准确性——比如手指或者基于触摸的屏幕）或者是fine（比如用鼠标或者手写笔）。

最后，hover属性将会检测设备的点击方式是否支持悬停效果，如果设备有多种点击方法，它应该返回最主要的方法的测试结果。对于一个以触摸屏为主要输入方式的设备而言，即使该设备能够连接并使用鼠标（支持悬停效果），那它也会返回0以表示不支持悬停效果。

当然，这些还都不是板上钉钉的事，因此规范日后可能还会被修改，不过看看地平线上已经露出了什么还是一件很有趣的事。

Ed Merritt

垂直媒介查询

Ed Merritt不仅是一位设计师、前端开发工程师，还是一名业余面包师，此外他也非常热爱麦芽酒。从2001年起，Ed就开始利用像素、字体并偶尔使用一些div来创建Web界面了。Ed与Headscape.co.uk另外一群可爱而有天赋的员工一起工作着，此外他还是TenByTwenty.com的创始人，TenByTwenty.com是一间设计字体、图标以及Wordpress主题的工作室。Ed现在住在英国南岸伯恩茅斯的海边。

计划

2010年年中的时候，我们正在为环境保护基金会设计一个新的网站，我就是在那时偶然读到Ethan Marcotte写的“*Responsive Web Design*”（响应式Web设计）的。我很赞同他提出的让布局去适应浏览环境的思想，这种思想在当时是非常新颖的，虽然他在设计中还取得了一些其他进展（他能够在当时提出一套能做到完全响应的方法，这其实已经超出他提出计划的范围了），但我还是渴望能在这套方法中再增加一些其他内容。

问题

通常主页都会有一个特点，即它们大多都有一个精彩的含有滚动图片的上半部分，但随后的部分就较为很普通了。这样做确实非常有效，我们发现在1024 × 768的屏幕上（据有关统计，这是访问者使用的第二多的分辨率），并且在大多数默认设置的情况下（浏览器窗口最大化，并且没有额外的工具条），可视区域到滑动图片之后也就正好戛然而止了。在测试中我们发现，大部分用户不会去看可视区域下面的部分，他们会误以为他们已经到了页面的末尾了，只有极少数人会向下滚动页面。

最近，我一直在为藏在可视区域下面的那部分页面内容而担忧，虽然有些用户会去滚动页面，但正如我们的测试结果显示的那样，有时候页面的布局其实是会误导用户的，让他们误以为自己已经看到了全部页面。在用户的浏览器中，如果内容正好在可视区域的末尾处结束，那么用户还为什么要去滚动呢？所以，我们面临的挑战是，我们需要向用户表明下面还有更多的内容。

解决方案

我已经为站点选好了两个固定的宽度：一个“宽”布局（针对那些视口大于1024px的设备）以及一个“窄”布局（针对那些视口介于800px到1024px之间的设备）。虽然这样的解决方案不是完全响应式的，但这至少在我朝着在项目中使用媒介查询的正确方向上，迈出了第一步。

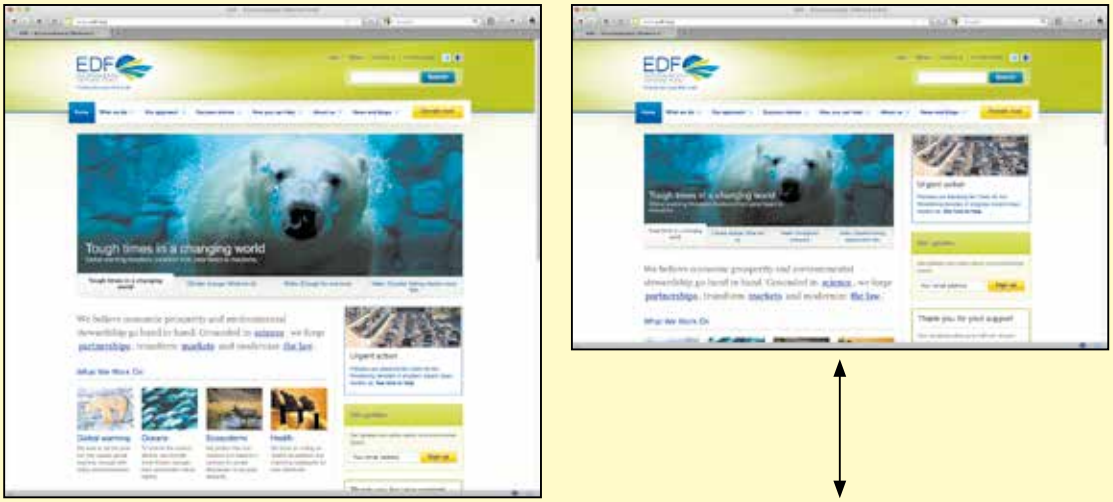
后来我意识到，其实可以通过使用垂直媒介查询在设备的垂直分辨率小于768px时修改布局来解决问题。较窄版本的设计已经将页面宽度减小到了其原始宽度的3/4，同时也等比地减小了出彩部分的高度。下面我要做的就是让较窄的版本能适应比较矮的浏览器窗口（如图所示）。

虽然窗口比较矮，但对于较宽的窗口来说，显然水平方向的空间没有得到充分利用。幸好藏在下面的页面中有一个占总宽度1/4的右边栏，那么其实我们可以将滑动图片设置为页面总宽度的3/4，然后在它旁边为边栏开辟出一块空间来。

效果

这种设计方法可以对客户端做出一些让步，同时还可以使用户意识到下面还有更多的内容。此外这里还有一个额外的奖励：这种方法还可以更高效地利用可视区域（我很少会希望滑动图片填满整个可视区域）。要达到所有这些目标，只需针对宽而矮的视口增加一条媒介查询就可以了。

虽然这是一种针对特定问题的解决方案，但其原理可以运用到其他地方。于是在后续的项目中，我经常会花一些时间坐下来问问自己，一些特定的（不同的宽或者高）视口会对描述内容产生不利影响吗？如果会，我该如何解决这个问题？



在使用了垂直媒介查询之后，在较矮的浏览器窗口中会显示多的内容，并以此来暗示用户下面还有更多的内容。

OR

其实并没有“ or ”这个关键词，但是逗号可以起到相同的作用。你可以使用逗号在一系列表达式中的某一个为真时，加载某些样式表：

```
@media screen and (color), projection and (color)
```

在上面这个例子中，如果被检测的是一台具有彩色屏幕的设备，或者是一台彩色的投影仪，那么这条查询语句就都将返回true。

ONLY

很多较老的浏览器支持媒介类型，但却不支持媒介查询。有时这会导致浏览器去尝试下载那些你不希望用户看到的样式。这时你就可以使用only关键字，对那些较老浏览器隐藏媒介查询，这样它们就不会识别出相应的样式了，而那些支持only关键字的浏览器则会继续处理带有only关键字的查询语句，这通常是个非常好的主意。

```
@media only screen and (color)
```

如果设备不支持媒介查询，那么它将会忽略上面这条查询。如果设备支持媒介查询，那么设备会将上面这条查询与下面这条查询等效对待：

```
@media screen and (color)
```

3.2.4 规则

媒介查询中的最后一块内容，就是你要应用的实际样式规则。你可以在这里写基本的CSS规则，它们唯一的特殊之处就是位于媒介查询里面：

```
@media only screen and (min-width: 320px) {
     a{
          color: blue;
     }
}
```

3.3 内嵌样式与外部样式

媒介查询既可以写在页面内部，也可以通过link元素的media属性被包含到页面中来。

你可以像下面这样将媒介查询包含到样式表中去：

```
a{
     text-decoration:none;
}
@media screen and (min-width: 1300px) {
     a{
          text-decoration: underline;
     }
}
```

在上面这个例子中，当屏幕宽度大于等于1300px时，页面中的链接将会带有下划线。

外部媒介查询则需要通过link元素来加载特定的样式表，这时你会在head元素中看到类似于下面这样的代码：

```
<link href="style.css" media="only screen and (min-width: 1300px)" />
```

到底应该选择哪种方法，这在很大程度上取决于你手中的项目，因为这两种方法都各有利弊。

在内嵌式媒介查询中，无论是否会用到，所有的样式都会被下载下来，但其好处是只需要一个HTTP请求即可。如果从性能的角度来考虑的话这非常重要，对于移动网络中的设备来说更是如此。因为移动网络有着较大的延迟。所谓延迟，是指服务器接受并处理一个从浏览器发来的请求的时间。移动网络上一次HTTP请求所需要的时间，可能是有线连接的四倍到五倍。当然内嵌样式也有其不足之处，那就是单个的CSS文件会变得非常大。因此，在你节省下了一些请求数的同时，你也创建了一个较难维护的大型文件。

你也许会非常惊讶，但事实是——外部的媒介查询也会下载所有的样式，即使它们不会被用到。这样的合理性在于，如果浏览器窗口的大小或者方向改变了，这些样式便已经各就各位了（该规则的一个例外是那些不支持媒介查询的设备，如果你在媒体查询前加上了only关键字，那么这些设备将会忽略这些外部样式）。

外部媒介查询的优势在于文件可以变得很小，进而利于维护。你也可以为设备提供一个轻量的、简化的、不支持媒介查询的样式表，而且要再一次感谢only关键字，你不必担心它们会出现在不需要它们的地方。

当然，最终选择哪种方式还是要依赖于你手头的项目，但通常我会推荐人们使用内嵌的媒介查询，因为额外的HTTP请求肯定会使站点变慢，而且性能是如此重要，以至于我们不能随意地忽视掉它。

3.4　媒介查询顺序

接下来要考虑的事情就是在你创建CSS时，要选择哪种设计思想来建立响应式的站点：是要从桌面端开始向下设计，还是从移动端开始向上设计。

3.4.1 从桌面端向下设计

正如响应式设计一开始所宣传的、以及人们通常所实现的那样，它是从桌面端向下设计的，因为它的默认布局是你在笔记本或者台式机的浏览器中所看到的样子。之后，它才利用了一系列媒介查询（例如典型的max-width）针对小屏幕简化并调整了布局。用这种设计思想创建出的样式表通常会是这样的：

▶注意
在示例代码中，…代表样式规则。有关更多编码惯例的信息请参阅本书第一章中的介绍。

```
/* base styles */
@media all and (max-width: 768px) {
...
}
@media all and (max-width: 320px) {
...
}
```

但是，从桌面端向下的设计造成了一些问题。虽然目前移动设备对媒介查询的支持有所改善，但仍旧不够完善。黑莓（6.0之前版本）、Windows Phone 7还有NetFront（驱动着前三代Kindle设备）对媒介查询都缺乏必要的支持。

每位用户都拥有最新最好的设备——这样的想象虽然有趣，但事实并非如此。在写这本书的时候，Android 4是最新的Android操作系统，但却有将近92%的Android设备还运行着Android 2.3.x或是更低版本的系统。同样的问题在较旧的黑莓设备中也同样普遍——目前有66%的黑莓用户的操作系统对媒体查询缺乏相应的支持[1]。

事实上，并不是每个人都喜欢追赶快速发展的新技术，而另外一些人则也许是因为压根就买不起这些新的技术和设备。

3.4.2 从移动端向上设计

如果你将事情反过来——优先建立移动体验，然后针对大屏幕使用媒介查询对布局作出调整——那你就可以在很大程度上规避上面所遇到的问题了。

优先建立移动体验能确保那些不支持媒介查询的移动设备也能获得一个合适的布

1 详见“Choosing a Target Device OS” http://us.blackberry.com/developers/choosingtargetos.jsp。

局。你唯一需要另外着手处理的就是IE，因为IE 9之前的IE浏览器是不支持媒介查询的，但是正如本章后面所描述的那样，这个问题也是很容易解决的。

一个采用从移动端向上的设计思想创建的样式表通常会是这样的：

```
/* base styles, for the small-screen experience, go here */
@media all and (min-width: 320px) {
...
}
@media all and (min-width: 768px) {
...
}
```

能获得浏览器更好的支持并不是从移动端向上设计方法的唯一好处，优先创建移动体验还可以降低CSS文件的复杂性。让我们以“另一个体育网站”文章页的边栏为例进行说明。当用大屏幕浏览边栏时，边栏被设置为display:table-cell并且宽为300px；而在小屏幕上浏览时，将边栏与文章正文排成一列也许会更好。如果用从桌面端向下的设计来实现这样的效果，其样式表会是这样：

```
aside{
     display:table-cell;
     width: 300px;
}
@media all and (max-width: 320px) {
     aside{
          display:block;
          width: 100%;
     }
}
```

而如果采用从移动端向上的设计，样式表则会是这样：

```
@media all and (min-width: 320px) {
     aside{
          display:table-cell;
          width: 300px;
     }
}
```

先用一个较为简单的布局作为浏览器的默认样式，然后再在上面搭建其他样式，这样做所需的CSS代码更少，同时代码结构也更加清晰。

3.5 创建核心体验

理想情况下，每个项目都应该从核心体验开始创建，而且该核心体验要尽量保持简单、合理，同时又能在尽可能多的设备上使用。覆盖面广是互联网的伟大特点之一——要尽可能发挥出它的最大优势。

在记住上面这点之后，我们将从一个简单的单列布局开始构建我们的核心体验。我们先把所有与布局相关的CSS都移到样式表的底部，并且释掉它们。

经过整合样式表中所有的浮动和display:table属性之后，样式表底部被注释掉的样式如下所示：

```
/*
.main {
     display: table-cell;
     padding-right: 2.5316456%;
}
aside {
     display: table-cell;
     width: 300px;
}
.slats li {
     float: left;
     margin-right: 2.5316456%;
     width: 31.6455696%;
}
.slats li:last-child {
     margin-right: 0;
}
nav[role="navigation"] li {
     float: left;
}
nav[role="navigation"] a {
```

▶ 注意

虽然留着也并不影响布局，但我还是把将广告宽度设置为100%的代码删掉了，因为这依赖于你与广告商之间的协定，也许你不能任意缩放广告的大小。我们将会在第4章中进一步讨论这个问题。

```
22.       float: left;
23. }
24. footer[role="contentinfo"] .top {
25.       float: right;
26. }
27. */
```

在代码被注释掉之后，页面如图3.8所示。

目前还没有涉及什么比较复杂的东西，这很好，因为这意味着我们的核心体验几乎适用于所有设备。我认为导航条目之间的距离可以再大一些，也许加个边框会有所帮助（图3.9）：

图3.8 随着样式被注释掉，页面现在有了一个简单且可访问的布局。

图3.9 在给导航条目增加了1px的边框之后，它们看起来相当锐利。

```
nav[role="navigation"] li {
     padding: .625em 2em .625em 0;
     border-top: 1px solid #333;
}
```

采用单列的布局并做了这样小小的调整之后，我们的核心体验就已经准备完毕了，现在是时候加入媒介查询来提高布局的可扩展性了。

3.6 确定断点

传统的确定断点的方法一直都是使用一些固定的标准宽度：320px（iPhone和其他一些设备属于这个宽度）、768px（iPad）和1024px。然而，依赖这些“默认的”断点存在一个问题。

当你完全根据常见设备的分辨率来确定断点时，你就会冒着为这些特定宽度开发却忽视在这些宽度之间（例如当iPhone横屏后你会遇到480px）的设备的风险，同时这种方法也不是对未来友好型的。今天流行的未必明天依旧流行，当下一代热门设备出现时，你将不得不增加新的断点。这是一场必败之仗。

3.6.1 追随内容

根据内容来决定应该在哪里设置断点以及需要设置多少个断点才是更好的方法。之后你可以再通过缩放浏览器窗口来查看还有哪里有进一步改善的空间。

在让内容来引导你的同时，其实你已经在很大程度上将布局与特定的分辨率进行了解耦，因为你允许页面内容来告诉你应该在什么时候对布局做出调整——这是明智的、也是对未来友好的一步。

为了能够确定出断点，你可以将浏览器窗口缩放至300px左右（假设你的浏览器允许你缩放到这种程度），然后缓慢地拉宽窗口直到有些东西看起来需要进行一点润色。

在大概600px时，“More in Football”中的图片占满了屏幕，因此看起来有些过大。在这里添加一条在第2章中提到过的媒介查询也许能起一些作用，从而使得图片缩小后能整齐地排成一行（图3.10）。

▶ 注意

媒介查询书签（http://seesparkbox.com/foundry/media_query_bookmarklet）是一个方便的工具，你可以在缩放窗口大小时用它来查看当前窗口的大小，也可以从中看到当前匹配了哪些媒介查询。

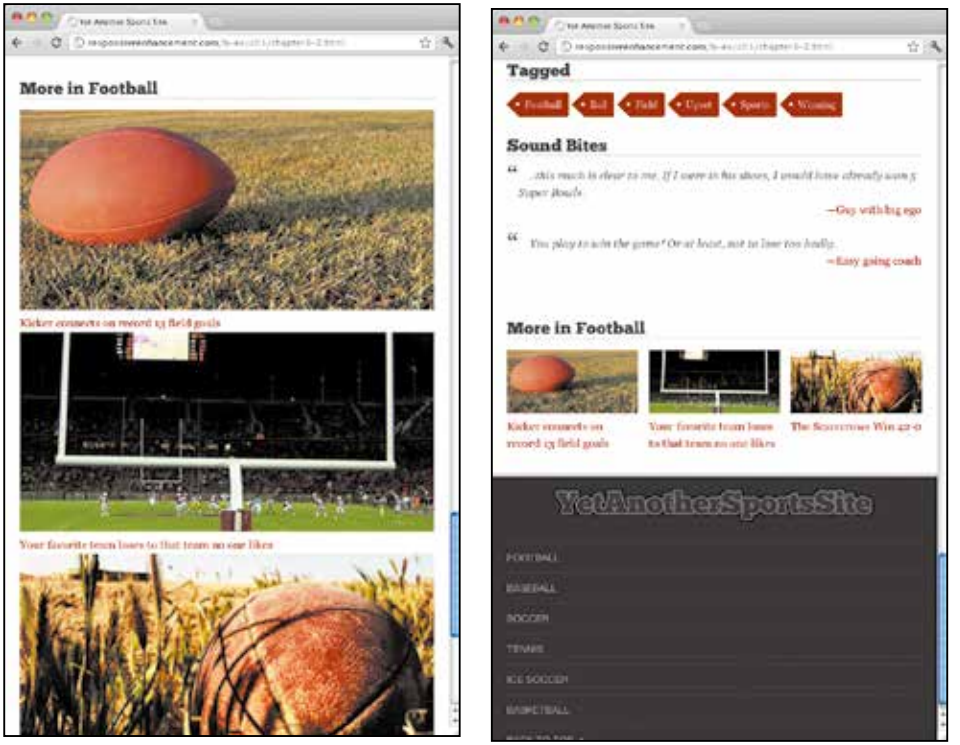


图3.10 图片几乎占满了屏幕（左图），所以在这里增加一个断点来调整设计是有意义的（右图）。

```
@media all and (min-width: 600px) {
      .slats li {
           float: left;
           margin-right: 2.5316456%; /* 24px / 948px */
           width: 31.6455696%; /* 300 / 948 */
      }
      .slats li:last-child {
           margin-right: 0;
      }
}
```

▶注意

即使你能跟得上，你也仍然应该查看一下在线示例：http://implementingresponsivedesign.com/ex/ch3/ch3.1.html

当浏览器窗口被拉长到860px左右的时候，页面的右边会让人感觉很空，但此时窗口还是太窄，不适合将边栏放在右边，因此我们可以让边栏浮动，这样它们便形成了两行（图3.11）。

```
@media all and (min-width: 860px) {
      aside{
           display: block;
```

图3.11 通过增加一个断点使得边栏在浮动后能与其他内容紧挨着，这样的布局看起来更加紧凑。

▶注意

为什么是860px？其实这里并没有什么硬性的规定。如果你认为早一点增加一个断点可以改进布局就尽管去做吧。但你需要记住的是，每个断点都会增加一小部分的复杂性，所以试着在其中找到一个平衡点吧。

```
            margin-bottom: 1em;
            padding: 0 1%;
            width: auto;
        }
        aside section{
            float: left;
            margin-right: 2%;
            width: 48%;
        }
        .article-tags{
            clear: both;
        }
        .ad{
            text-align: center;
            padding-top: 2.5em;
        }
}
```

还可以在当前的断点处让导航条目重新排成一行，而不是像现在这样堆叠成一列（图3.12），但相应的CSS被注释掉了，我们需要把它们重新放回到媒介查询中去。同时，我们还要删掉导航条目的边框：

```
@media all and (min-width: 860px) {
        ...
        nav[role="navigation"] li {
```

```
4.              float: left;
5.              border-top: 0;
6.         }
7.         nav[role="navigation"] a {
8.              float: left;
9.         }
10.        footer[role="contentinfo"] .top {
11.             float: right;
12.        }
13. }
```

图3.12 现在有了足够的空间来让导航浮动，并能使正文内容提前。

最后，当浏览器被拉伸至940px左右宽的时候，看起来边栏已经可以放回到右侧去了。边栏里面的内容不再浮动，这样才能达到占满整个边栏宽度的目的：

```
1.  @media all and (min-width: 940px) {
2.        .main {
3.             display: table-cell;
4.             padding-right: 2.5316456%; /* 24px / 948px */
5.        }
6.        aside {
7.             display: table-cell;
8.             width: 300px;
9.        }
10.       aside img {
11.            max-width: 100%;
12.       }
```

```
13.       aside section {
14.            float: none;
15.            width: 100%;
16.       }
17. }
```

现在，940px及以上宽度的布局就和页面在第2章结束时的样子非常像了（图3.13）。

图3.13 通过增加新的断点，当浏览器窗口宽度大于940px时，页面布局和我们之前的设计看起来就非常像了。

3.6.2 增强对大屏幕的支持

随着浏览器窗口被继续拉宽，文章又会变得于难以阅读。很多站点会在这里设置max-width属性来限制浏览器窗口的最大宽度，或者通过增大字体来增加行宽。

不要去限制浏览器窗口的宽度，相反，我们将使用CSS3中的多列布局来解决这个问题。

多列布局可以告诉浏览器要以多少列来显示内容（图3.14），而且浏览器对此的支持也相当不错：Opera、Firefox以及WebKit都支持。只不过要确保在Firefox、IE 10以及WebKit中都使用了正确的前缀，对于Opera和IE来说没有前缀也可以。虽然多列布局是一个很好的特性，但是对于站点来说这个功能也并不是必需的，所以对于这些支持这一特性的浏览器，我们可以采用渐进增强的方法：

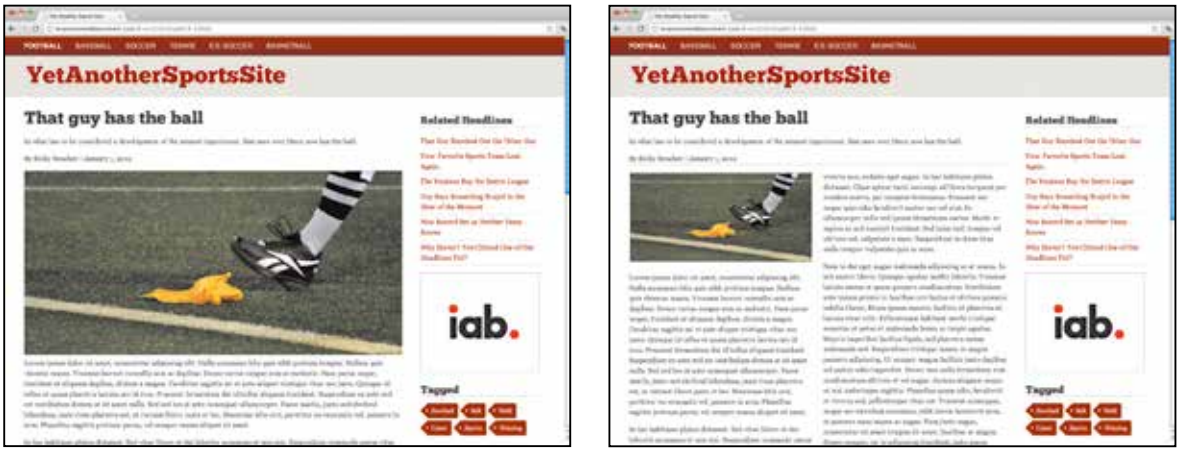


图3.14 对于较宽的浏览器窗口而言，将文章分成两列可以协助保持一个对读者来说较为友好的行宽。

```
1.  @media all and (min-width: 1300px) {
2.        .main section {
3.              -moz-column-count: 2; /* Firefox */
4.              -webkit-column-count: 2; /* Safari, Chrome */
5.              column-count: 2;
6.              -moz-column-gap: 1.5em; /* Firefox */
7.              -webkit-column-gap: 1.5em; /* Safari, Chrome */
8.              column-gap: 1.5em;
9.              -moz-column-rule: 1px dotted #ccc; /* Firefox */
10.             -webkit-column-rule: 1px dotted #ccc; /* Safari, Chrome */
11.             column-rule: 1px dotted #ccc;
12.       }
13. }
```

第3行到第5行的代码负责告诉浏览器要用多少列来显示文章。第6行到7行的代码会告诉浏览器要在列与列之间插入1.5em宽的空白间隔（24px）。最后，第9行到11行的代码会告诉浏览器在空白间隔中要加入1px宽的浅灰色点状线，以此来提供一些视觉上的分隔（图3.15）。

现在行宽已经有了更大的改善，如果能在文章和作者信息之间增加一些间隔，那么页面看着会更加舒服一些。此外，图片也可以离内容再远一些：

```
1.  @media all and (min-width: 1300px) {
2.        .main section img{
3.              margin-bottom: 1em;
```

```
            border: 3px solid #dbdbdb;
        }
        .main .articleInfo{
            border-bottom: 2px solid #dbdbdb;
        }
        ...
}
```

图3.15 增加的少量空间和边框有助于把图片与后续文本相分离。

在为图片加上边框以及额外的padding部分之后，现在整个设计看起来相当漂亮。

3.6.3 使用em为媒介查询增加灵活性

人们会使用有着不同缩放级别的浏览器来浏览Web，有些视觉有障碍的人也许会觉得很多站点的文字都难以看清，因此会将默认的缩放等级增大 。

当用户使用非默认缩放等级的浏览器浏览页面时，页面中的文字便会增大（或变小）。在Firefox和Opera中这不是什么问题，因为基于像素的媒介查询将会根据缩放等级来重新计算样式，并应用到字体上。但在其他浏览器中，我们那些之

图3.16 由于断点是以像素为单位的，所以当用户放大浏览页面时，我们精心设计的布局就会变得十分混乱。

前一直很完美的断点开始让我们失望了。忽然之间，几乎所有的内容显示得都不那么合理了，而且之前理想的行宽也无情地消失了（图3.16）。同样的问题在第2章我们讨论“流动布局”时也曾出现过，即设备具有不同的默认字体大小。例如，Kindle默认的26px就会破坏以像素为单位的媒介查询。我们可以通过使用em来解决这些问题，从而让我们的站点更加灵活。

正如我们在第2章中介绍过的那样，把以像素为单位转换为以em为单位，其实与用目标（断点）除以上下文（这里是16px，即body的字体大小）一样简单：

```
/* 600px/16px = 37.5em */
@media all and (min-width: 37.5em) {
...
}
/* 860px/16px = 53.75em */
@media all and (min-width: 53.75em) {
```

```
...
}
/* 940px/16px = 58.75em */
@media all and (min-width: 58.75em) {
...
}
/* 1300px/16px = 81.25em */
@media all and (min-width: 81.25em) {
...
}
```

▶注意

如果你是在页面加载后放大的，也许你需要刷新页面。大多数人会提前设置缩放级别，所以这通常不是问题。

在以em作为单位之后，即使站点被放大了，媒介查询也能够生效，并保证布局仍旧是优化过的（图3.17）。

图3.17 通过用em为单位来设置断点，无论访问者使用的是什么样的缩放等级，你都可以保证他们能看到一个恰当的布局。

图3.18 当你在手机上查看页面时，内容被深深地埋在了导航栏的下面。

使用以em为单位的媒介查询是另一种拥抱Web灵活性和不可预测性的途径，这样做可以让用户控制体验，同时让内容来决定布局。

3.7 导航栏

导航栏是我们在总结媒介查询这章前要处理的最后一个对象。如果用户不知道如何前往他想去的页面，那么即使站点有再好的内容也不会留住用户。不论用户使用的是什么样的屏幕，你的导航栏都应该是可访问的，而且要易于使用。

在我们目前的例子中，导航栏对于移动设备还不够友好。将导航条目一个个堆叠起来看起来虽然比较整齐，但是文章——用户来访问你站点的首要目的——却被挤到了下面（图3.18）。

我们期望的导航栏应该符合下面的一些原则：

- 它不应该占据宝贵的屏幕空间。
- 它应该是直观的，这样用户就不会觉得迷失了方向或者感到困惑。
- 它应该可以在各种设备上使用（尽管体验会根据所使用的设备能力的不同而有所不同）。

▶注意
关于更详细的创建响应式导航栏的方法，以及它们每种方法的优缺点，请参阅Brad Frost的文章http:// bradfrostweb.com/blog/web/responsive-nav-patterns/。

让我们快速浏览一下我们都有哪些选择：

- 什么都不做。就让页面保持它现在的样子，虽然它是直观的，也能在各种设备上运行良好，但是它占据了太多的屏幕空间。
- 转换为下拉列表。我们可以把导航栏转换为下拉列表，这将能够节省屏幕空间，并且在大多数设备上也具有可用性，同时这样也能在那些不支持JavaScript的设备上得到很好的降级。然而，用户熟悉的下拉列表原本是作为表单的一部分的，他们也许会因为看到下拉列表被用来导航而感到困惑。而且我们也不能样式化下拉列表，因为大多数浏览器都不允许我们这样做。
- 切换式菜单。在小屏幕上，我们可以使用JavaScript在最初时先隐藏导航栏，然后给用户提供一个按钮，通用单击该按钮来显示导航栏。这种方法能够符合我们的3个要求：它节省了屏幕空间、对用户来说比较直观，同时可以在各种设备上实现，并且在那些不支持JavaScript的设备上也能很好地降级。

切换式菜单能够满足我们所有的要求，我们就在“另一个体育网站”上使用这种方法吧。

切换

仅使用几行CSS和JavaScript代码，你就可以实现一个简单的切换式菜单了。

首先，在你的HTML文件中增加一个链接，该链接用来作为折叠按钮。你可以把这个链接放在导航列表的上面。

```
<a href="#nav" class="nav-collapse" id="nav-collapse">Menu</a>
<ul class="nav" id="nav">
```

用于切换的CSS代码

我们在CSS里创建一些样式规则来格式化折叠按钮，并先隐藏它。

```
1.  #nav-collapse{
2.       display: none;
3.       color: #fff;
4.       text-align: right;
5.       width: 100%;
```

```
6.        padding: .625em 0 .625em 0;
7.  }
8.  #nav-collapse.active {
9.        display: block;
10. }
```

第1至7行代码为切换按钮设置了一些基本样式，并且会在一开始时隐藏菜单。记住，如果浏览器不支持JavaScript，那么将会显示导航栏，在这种情况下按钮也就没有存在的必要了。

当“ active ”类被触发时，第8至10行代码将通过JavaScript来显示按钮。

在不支持JavaScript的浏览器中使用这些样式时将不会有任何改变。这正是我们想要的。即使浏览器不支持JavaScript，导航将仍然是完全可用的，虽然效果不是很理想，但至少它是可用的。

用于切换的JavaScript代码

JavaScript的部分也很简单。我们先创建一个名为yass.js的文件，并在HTML中的body闭合标签前把它包含进来。

```
<script type="text/javascript" src="yass.js"></script>
```

然后把下面的JavaScript代码拷贝到yass.js中去。

```
1.  window.onload = function() {
2.        var collapse = document.getElementById('nav-collapse');
3.        var nav = document.getElementById('nav');
4.        //toggle class utility function
5.        function classToggle( element, tclass ) {
6.              var classes = element.className,
7.                 pattern = new RegExp( tclass );
8.              var hasClass = pattern.test( classes );
9.              //toggle the class
10.             classes = hasClass ? classes.replace( pattern, '' ) :
                classes + ' ' + tclass;
11.             element.className = classes.trim();
12.       };
13.       classToggle(nav, 'hide');
14.       classToggle(collapse, 'active');
```

```
15.       collapse.onclick = function() {
16.            classToggle(nav, 'hide');
17.            return false;
18.       }
19. }
```

当页面载入时（第1行代码），浏览器会自动运行上面的JavaScript。第2至3行代码会分别获取导航栏和切换按钮，方便在后面脚本中使用它们。

第5至12行代码创建了一个名为toggleClass的简单函数，该函数会获取一个元素，并检查该元素是否有某个特定的类名。如果有就删除，如果没有就添加。

第13至14行代码为导航栏添加了一个hide类，并且为按钮添加了一个active类。

最后，第15至18行代码定义了一个函数，并会在单击切换按钮时触发。该函数能添加或删除导航栏的hide类，这样按钮就可以控制导航菜单的隐藏与显示了。

现在这段代码会在所有设备上运行，这显然不是我们想要的，我们希望在导航菜单折叠的时候才运行这段代码。虽然检测屏幕宽度比较容易，但是这意味着断点会被硬编码到两个位置：CSS文件中和JavaScript文件中。

如果使用JavaScript来检查导航条目有没有浮动，并基于此来运行代码，那我们就只需在一个地方保存断点了，这会使得之后维护断点的工作更加方便。当我们做这一修改时，可以顺便把classToggle函数也放到另外一个单独的工具对象中去，我们会在稍后再来创建这个完整的工具对象。

```
1.  var Utils = {
2.       classToggle : function(element, tclass) {
3.            ...
4.       }
5.  }
6.  window.onload = function() {
7.       var nav = document.getElementById('nav');
8.       var navItem = nav.getElementsByTagName('li');
9.
10.      //is it floated?
11.      var floated = navItem[0].currentStyle ? el.currentStyle['float'] :
```

```
        document.defaultView.getComputedStyle(navItem[0],null).
        getPropertyValue('float');
12.
13.     if (floated != 'left') {
14.         var collapse = document.getElementById('nav-collapse');
15.
16.         Utils.classToggle(nav, 'hide');
17.         Utils.classToggle(collapse, 'active');
18.
19.         collapse.onclick = function() {
20.             Utils.classToggle(nav, 'hide');
21.             return false;
22.         }
23.     }
24. }
```

让我们稍微看一下这段代码。

第8至11行代码会获得导航条目，并检查它有没有浮动。第11行代码看起来也许会有点吓人，但其实它只负责决定该获取哪种样式 。IE浏览器有点特殊，如果用户使用的是IE浏览器，我们就需要检测不同的属性。

在得到导航条目是否浮动了的信息之后，你就可以命令剩下的JavaScript只在导

图3.19 随着切换功能的实现，现在的导航栏只有在访问者需要它的时候才出现。

航条目浮动时才运行了。有了这些样式和脚本，虽然当你在大屏幕设备上刷新页面时不会看到任何改变，但是在小屏幕设备上显示时，页面加载后你就会看到切换按钮了，而且可以通过点击它来隐藏或显示导航菜单（图3.19）。

3.8 对IE的支持

我们还没有万事大吉，因为人们都喜欢使用桌面浏览器，但是IE着实还是会让我们有些头疼。

只有IE 9及其更高版本的IE浏览器才支持媒介查询，这意味着如果你是优先为移动端创建站点的，使用IE 9之前版本IE的用户就会看到一个本应属于小屏幕的布局。

我们可以通过在条件注释中加载针对IE的样式来修复这个问题。从第2章“流动布局”开始，我们就有了针对IE浏览器的ie.css，这使得问题变简单了一些。

我们先将条件注释的条件改为所有IE9之前版本的IE，虽然这会让本来支持display:table的IE 9也要使用浮动，但这一损失比起增加针对IE的样式表的复杂性来说还是值得的：

```
<!--[if (lt IE 9) & (!IEMobile)]>
<link rel="stylesheet" href=""/css/ie.css" media="all">
<![endif]-->
```

然后，我们把原来在媒介查询中的样式放到IE的样式表中去：

```
.main {
      float: left;
      width: 65.8227848%; /* 624 / 948 */
}
.slats li {
      float: left;
      margin-right: 2.5316456%; /* 24px / 948px */
      width: 31.6455696%; /* 300 / 948 */
}
.slats li:last-child {
      margin-right: 0;
```

```
}
aside{
      display: block;
      margin-bottom: 1em;
      padding: 0 1%;
      float: right;
      width: 31.6455696%; /* 300 / 948 */
}
nav[role="navigation"] li {
      float: left;
      border-top: 0;
}
nav[role="navigation"] a {
      float: left;
}
footer[role="contentinfo"] .top {
      float: right;
}
aside img {
      max-width: 100%;
}
```

现在，站点在IE上就可以很好地运行了。虽然它不是响应式的，但是至少它在大多数屏幕上都能呈现为流动布局了。

3.9 结束语

流动布局是我们迈出的第一步，但是它也只能带我们走这么远。有时候，我们需要进一步调整布局来适应不同的设备。

智能手机总是希望能为人们提供完整的Web体验，如果不使用视口元标签，那么大部分的智能手机都将会显示一个缩小版的站点。

媒介查询让我们能够检测诸如视口宽度或高度这样的特性，并据此来适当地调整CSS样式。媒介查询可以是内嵌的，也可以是从外部引用的，这两种方法都有其

优缺点，所以选择能够满足项目需求的方法就变得非常重要的。

虽然选择特定设备的宽度作为断点是常见的做法，但更好的方法是让内容来决定要在哪里增加一条媒介查询。

在媒介查询中使用em代替像素作为单位，可以使响应式站点更加灵活而且是可访问的。

一定要在真实的设备上进行测试，这样做可以提醒你也许应该针对不同的屏幕对类似于导航菜单这类组件作出相应的调整。

在下一章中我们将会看到，你可以采用哪些方法来为不同的设备提供大小合适的图片，并以此来大幅提升站点的表现。

第4章

响应式多媒体

看！我们已经可以用十七种不同的方式来理解它了，可是我们的感觉还是不太好，因为每次无论我们用哪种方式来理解，总会有人不喜欢。

——由BUDDY HACKETT饰演的BENJY BENJAMIN

电影《疯狂世界》

每当我们提到丰富的在线体验时，我们便会对它既爱又恨。一方面，漂亮的图片和有趣的视频可以让人们获得一种更加沉浸而愉悦的体验，但另一方面，在页面中包含太多的图片和视频又会延长页面的加载时间，这无疑会给用户带来挫败感。因此我们不得不谨慎地权衡并作出计划，争取为用户将这两方面都做到最好，从而为他们提供尽可能快的美妙体验。

通过使用前面3章所介绍的方法，我们已经创建出了一个响应式的站点，而且它在桌面电脑、平板电脑和智能手机上看起来都相当不错：用户可以随心所欲地缩放浏览器窗口大小，网站可以自动根据变化来调整其布局。如果交付一个响应式设计如此简单，那么这本书其实可以比现在薄上很多，但其实在使站点变得更加整洁方面还有很大的提升空间，比如图片就是个典型的例子。

本章我们将讨论以下内容：

- 为什么性能很重要
- 如何实现有条件地加载图片
- 有哪些响应式的图片解决方案可供我们使用，它们都有什么限制
- 如何在不下载许多图片的前提下显示背景图片
- 如何有条件地加载Web字体
- 响应式图片的前景如何
- 如何在保持高宽比的前提下使嵌入式视频可伸缩
- 对于响应式广告我们能够做些什么

4.1 问题是什么

当屏幕宽度一旦超过了最大断点（1300px）时，“ More in Football ”部分中的图片看起来就会变得模糊，而其他的图片则依然清晰而明快。

此外，我们还可以为小屏幕优化一下文章正文中的图片。因为如果能够对图片进行进一步的裁剪，那么图片就能在小屏幕上继续保持它在大屏幕上时的那种视觉冲击力。而现在在小屏幕上，图片中的旗子以及球员的脚都失色不少（图4.1）。

图4.1　在小屏幕上，旗帜和球员的脚都失色不少。

关键的问题不是图片看起来如何，而是如何来衡量它们，它们对性能有多大的影响。现在无论使用什么设备，下载到的都是相同的图片。这就意味着，在一个有350px宽的图片就足矣的设备上，却下载了一张624px宽的图片。这无疑严重地降低了页面的性能，而且这对用户来说也是个大问题。

性能

遗憾的是，性能在很多项目中向来扮演的都是马后炮的角色。我们来看一下数据为我们揭示出的性能本该有的重要地位。

比起普通的互联网用户，大多数从事Web相关工作的人都拥有更快的网速，这导致了这些人和普通用户体验着不一样的Web。因此，我们的用户能够更加敏锐地体会到那些性能较差的站点给他们带来的种种痛苦。

2009年，购物比较网站巨擘Shopzilla将其页面加载时间从4到6秒提高到了1.5秒，其效果是令人惊叹的：站点的转换率上升了7到12个百分点，访问量增长幅度更是高达25%。

Mozilla在将页面加载时间降低了2.2秒后，也发现了类似的结果：下载量上升了15.4个百分点，大约相当于Firefox浏览器每年新增1028万的下载量！

在手机上的情况则更加惊人。网络慢、硬件能力弱，此外还要面对种种数据混乱的限制和转码，尽管这样，用户们的期望却依旧没变。事实上，71%的移动用户期望在手机上浏览站点时能像他们在家中计算机上浏览站点时一样快，甚至更快。

这对于我们现在的站点来说是个坏消息，因为logo和文章的图片都很大：文章的图片有624px宽并且有50KB那么大。小屏幕上的布局本应有一个稍小一些的图片（大约300px宽）与之配合，而我们却给了它一张在桌面浏览器上使用的图片。为了对小屏幕做优化，减小传输的数据量就是一个重要的考虑因素，我们无法忽略掉这一点。

经过快速评估之后，我们可以对下面这些图片进行优化：

- “More in Football”部分中的图片。这些图片中的每一张都有300px宽，这在小屏幕上是多余的。实际上，它们占据了太多的屏幕空间，并因此挤占了本应属于文章正文的空间（图4.2）。在小屏幕上显示标题能够让用户获得更好的体验，而不是显示图片。
- 文章图片。文章图片的宽度达到了624px，并且有50KB那么大。其实在小屏幕上，图片有现在的一半大小就刚好合适了。因此可以对小屏幕上的图片进行进一步裁剪，从而使人们的注意力聚焦在旗子上，从而带来更强的冲击力。
- Logo。虽然logo只有10KB，相比文章的图片已经小了很多，但是它同样比实际所需的大小足足大了两倍。

图4.2 在小屏幕上，“More Football”中的图片占据了过多宝贵的屏幕空间。

4.2 有选择地为手机提供图片

下面就让我们从删减核心体验中“More in Football”里的图片开始吧。也许就使用一条display:none来结束今天的工作是个诱人的选择，但这样做并没有解决问题，我们只是把图片给藏了起来罢了。

即使将图片设置为display:none，图片也依然会在浏览器中被请求并下载，所以即使屏幕上不显示，额外的请求和文件大小的问题依然会存在。与前面一样，正确的方法还是得从移动端开始设计，然后逐渐增强大屏幕上的体验。

首先我们删除HTML里所有的图片：

```
1.  <ul class="slats">
2.       <li class="group">
3.            <a href="#">
4.                 <h3>Kicker connects on record 13 field goals</h3>
5.            </a>
6.       </li>
```

```
        <li class="group">
            <a href="#">
                <h3>Your favorite team loses to that team no one likes</h3>
            </a>
        </li>
        <li class="group">
            <a href="#">
                <h3>The Scarecrows Win 42-0</h3>
            </a>
        </li>
</ul>
```

● **自定义数据属性**

专门为页面存储自定义数据并具有data-前缀的属性，通常出于脚本编程的目的而设置。

显然，现在的HTML不会再下载图片了，在小屏幕上这样没有问题，对于较大的屏幕来说几行JavaScript就可以让图片恢复。通过巧妙地使用HTML5的data-属性可以很容易地告诉JavaScript应该去下载哪些图片：

```
<ul class="slats">
        <li data-src="images/ball.jpg" class="group">
            <a href="#">
                <h3>Kicker connects on record 13 field goals</h3>
            </a>
        </li>
        <li data-src="images/goal_post.jpg" class="group">
            <a href="#">
                <h3>Your favorite team loses to that team no one likes</h3>
            </a>
        </li>
        <li data-src="images/ball_field.jpg" class="group">
            <a href="#">
                <h3>The Scarecrows Win 42-0</h3>
            </a>
        </li>
</ul>
```

4.2.1 JavaScript

首先要在JavaScript中添加用来帮助快速选取元素的工具函数，虽然这不是必需的，但它绝对有用：

```
q : function(query) {
      if (document.querySelectorAll) {
           var res = document.querySelectorAll(query);
      } else {
           var d = document,
           a = d.styleSheets[0] || d.createStyleSheet();
           a.addRule(query,'f:b');
           for(var l=d.all,b=0,c=[],f=l.length;b<f;b++) {
                l[b].currentStyle.f && c.push(l[b]);
                a.removeRule(0);
                var res = c;
           }
           return res;
      }
}
```

如果你对JavaScript还不太熟悉，那么上面这段代码也许看起来会有些乱。没关系，函数只是负责读入一个选择器，然后返回与之匹配的元素罢了。如果你能理解每行函数当然很好，即便你不能理解每行代码，只要能知道函数有哪些功能便足矣。

有了这个函数之后，加载图片的部分就相当简单了：

```
//load in the images
var lazy = Utils.q('[data-src]');
for (var i = 0; i < lazy.length; i++) {
      var source = lazy[i].getAttribute('data-src');
      //create the image
      var img = new Image();
      img.src = source;
      //insert it inside of the link
      lazy[i].insertBefore(img, lazy[i].firstChild);
};
```

第2行代码会选取所有具有data-src属性的元素，然后第3行代码会循环遍历这些元素。第4至7行的代码负责使用data-src的值为每一个元素创建一个新的图片。最后，脚本会将这些新的图片插入到相应的链接中去。

有了这些JavaScript之后，页面就不会立刻请求图片了，相反图片会在页面加载后才加载，这正是我们想要的。现在，我们只需要告诉脚本不要在小屏幕上加载这些图片就可以了。

Guy Podjarny

响应式设计对性能的影响

Guy Podjarny（简称Guypo）作为一位Web性能方面的研究专家和布道者，他在不断追逐着让人难以捉摸并且瞬息万变的Web 。他的大部分精力都献给了手机Web性能方面的研究，并会定期深入到各种手机浏览器的研究中去。同时，Guypo也是免费手机测量工具Mobitest的作者，并且在很多开源工具中也都有贡献。之前Guypo 是Blaze.io的创始人兼CTO，在Blaze.io被Akamai收购之后，他成为了Akamai的首席产品架构师。

响应式设计能够解决很多问题，但同时它也很容易在诸如如何维护、如何成为未来友好型的，以及如何让它变得更酷这些问题中迷失方向。在这些问题中，“它能有多快”这个问题显得尤为重要。因为在你构建的用户体验中，性能是其中一个不可或缺的标准组件，而且很多案例研究表明，性能影响着你的用户的满意度，性能也是你最重要的一道防线。

现在智能手机的浏览器常常会被重定向到专门的移动站点，这些站点的内容和视觉设计显然是经过删减的，并被称为mdot（译者注：译者认为之所以会这样叫是因为很多针对手机的站点的网址都使用的是“m.”开头的子域名，所以才会称之为mdot）站点。下载更少的图片、脚本和样式表……这些转变使得网站能更快加载。原因显而易见——下载更少的字节以及发起更少的请求肯定会比原来要快。

但是响应式网站的设计却并没有遵从这个模式。我最近对347个响应式网站（所有的网站都是在2012年3月从http://mediaqueri.es/中选取的）做了性能测试，我在从320×480到1600×1200四种不同大小的Google Chrome中加载了每个网站的主页。借助于Web性能测量工具www.webpagetest.org，我对每个页面都进行了多次测量。

测量结果是令人沮丧的。尽管通过调整浏览器大小可以改变它们的外观，但是页面的大小和加载时间几乎没有改变。86%的网站即使在最小的屏幕上，也会粗暴地加载与在最大的屏幕中浏览时相同大小的页面。也就是说，尽管在小屏幕上看起来像是一个mdot的站点，但实际上它们仍然下载了所有的内容，这也导致了慢到令人发指的加载速度。

虽然每个网站的情况都各不相同，但有3种原因几乎是所有过度下载的网站所共有的：

- 下载并隐藏
- 下载并缩小
- 额外的DOM

下载并隐藏是目前最主要的原因。响应式设计的站点通常会给所有客户端一份相同的HTML，即使是使用“移动端优先”设计方法设计的站点，它的HTML里也包含了所有用在最大的屏幕

上提供丰富体验所需的内容，并在小屏幕上使用display:none隐藏了这些不该显示的内容。

可惜display:none帮不上性能任何忙：页面中被隐藏的资源依旧会被下载；被隐藏的脚本依旧会运行；DOM元素也依旧会被创建。最后的结果就是：即使页面内容被隐藏了，浏览器仍然需要遍历页面内的所有资源，并下载能找到的所有资源。

下载并缩小在概念上与下载并隐藏类似。为了能与不同大小的屏幕进行匹配，响应式Web设计中使用了流动图片。虽然这样做加强了视觉感染力，但这也意味着桌面级别大小的图片每次都会被下载下来，即使在屏幕小得多的设备上也是如此。其实在小屏幕上用户是无法欣赏那些高质量的图片的，那些多余的字节就这样被白白浪费掉了。

额外的DOM是这一些列故事中的第三集。响应式Web设计的网站会为所有浏览器都发送相同的HTML。尽管有些内容会被隐藏，但浏览器还是会解析并处理隐藏部分的DOM。这使得在小屏幕上加载一个响应式的网站变得十分复杂，其结果也偏离了人们所期待的用户体验，而且复杂的DOM还导致了更高的内存消耗、昂贵的回流以及运行得更慢的网站。

这些问题都不容易解决，因为它们是由现有的响应式Web设计和浏览器的工作模式所共同决定的。然而还是有一些方法可以帮我们将站点的性能维持在一个可控的范围之内：

- 使用响应式图片
- 优先为移动平台创建
- 测量

本书已经详细地讨论过响应式图片了，它便可以用来帮助解决“下载并缩小”的问题。由于每个页面上的图片都是成块的字节，所以从图片入手是显著减小页面大小最容易的方法。需要注意的是，CSS里的图片也应当是响应式的，并且可以用媒介查询来代替。

优先为移动平台创建意味着在手机优先的网站的基础上又向前迈进了一步，因为这实际上是在为那些你在乎的分辨率最低的设备而编码。一旦该设计得以实现，那么它应该和其他mdot站点表现得一样，同时也是轻量级的。所以我们要做的就是借助JavaScript和CSS来增强页面，从而避免出现过度下载。这样做虽然会使那些不支持JavaScript的客户端只能获得最基本的浏览体验，但这对于这些边缘设备来说已经足够好了。需要注意的是：在通过JavaScript加强站点的同时，还要保证其高性能，虽然这不是一件容易的事。业内对此的最佳实现暂时还没有出炉——这也把我带到了我要讲的下一点。

测量。要把测量当成是网站的核心来对待，在弄清楚并接受网站的性能表现之前绝不发布网站。如果手机网站的页面超过了1MB，那么你最好在采取一些措施之后再让其上线。可以用来测量的工具有很多，但我推荐在真实的设备上做测试（http://akamai.com/mobitest），同时用WebPageTest在桌面浏览器上做测试，因为你可以使用setviewportsize命令来缩放它们。

总而言之，响应式Web设计是一种强劲的、向前思考的技术，但同时它对性能也有着显著的影响。要确信你已经明白了这些挑战，并且要在设计的过程中避免它们，这样，用户在看到你那令人惊讶的视觉设计和内容前，他们就不需要再忍受什么了。

4.2.2 matchMedia介绍

在第3章“媒介查询”中，我们编写的用来在小屏幕上切换导航栏显示的脚本，会检查导航条目是否浮动了，如果浮动了就为导航添加收缩的特性。而这次，我们将使用更为简便的matchMedia()方法。

matchMedia()是JavaScript内部自带的方法，你可以将CSS媒介查询作为参数传递给它，它会返回相关媒介查询是否匹配的信息。

具体来说，函数会返回一个MediaQueryList对象，该对象具有两个属性：matches和media。matches属性的值可以是true（如果媒介查询匹配）或者false（如果媒介查询不匹配）。media属性的值就是你刚刚传递的参数，例如对于window.matchMedia("(min-width: 200px)")来说media属性将会返回"(min-width: 200px)"。

▶注意

Irish的polyfill可以在Github上下载到：https://github.com/paulirish/matchMedia.js 或者从本书网站的示例文件中也可以找到：http://www.implementingresponsivedesign.com。

● Polyfill

为那些不支持相关功能的浏览器提供支持能力的代码片段。

支持matchMedia()方法的浏览器有：Chrome、Safari 5.1+、Firefox 9、Android 3+以及iOS5+。Paul Irish为那些不支持该方法的浏览器创建了一个方便使用的polyfill。

有了matchMedia polyfill，就可以告诉所有的浏览器只有当图片比matchMedia中的第一个断点大的时候，才插入图片：

```
1.  if (window.matchMedia("(min-width: 37.5em)").matches) {
2.       //load in the images
3.       var lazy = Utils.q('[data-src]');
4.       for (var i = 0; i < lazy.length; i++) {
5.            var source = lazy[i].getAttribute('data-src');
6.            //create the image
7.            var img = new Image();
8.            img.src = source;
9.            //insert it inside of the link
10.           lazy[i].insertBefore(img, lazy[i].firstChild);
11.      };
12. }
```

现在，当在手机上或者其他小窗口浏览器中加载页面时，就不会请求图片了（图4.3），这对于小屏幕设备上的性能来说是一次巨大的胜利。现在只有3个HTTP请求，页面大小也只有大约60KB（算上那3张图片），而且标题还在那里，链接也完全可用，使用体验没有受到任何影响。

图4.3　在小屏幕上，并没有请求“ More in Football ”中的图片，这极大地提高了页面的性能。

删除了这些图片之后，我们就该专注于如何导入图片和logo了。我们希望这些图片在所有分辨率的屏幕中都要显示，尤其是logo，因此不需要有条件地加载它们，而是每次都要下载它们，只不过图片的大小要合适。这是件很麻烦的事。

4.3　响应式图片策略

据说世界上其实只有7个故事，另外其余的故事只不过是以不同的方式来讲述的罢了。同样，现在只有3种处理响应式图片的策略：和浏览器比赛、默许浏览器的行为，或者去找服务器帮忙。

4.3.1　和浏览器比赛

大部分的前端解决方案都是在试图与浏览器比赛——在浏览器下载到错误的图片前，这些方案要尽最大努力切换到应该被下载的正确的图片上。

这是一个越来越困难的任务，因为浏览器希望快速加载页面，所以它们会竭尽全力地快速下载图片。这当然是件好事——你也希望你的站点能尽快加载，只有在当你想打败它们的时候，才会觉得下载快也是件烦人的事。

4.3.2 默许浏览器的行为

第二种策略其实就是默许浏览器的默认行为，典型的例子是：先使用默认的方法下载小屏幕可以使用的图片，然后再在需要的时候为大屏幕下载较大的图片。

这样做显然是不理想的，因为大屏幕设备得在本来只需要发一次请求的地方发两次请求，所以应该尽可能避免这种情况的出现，大屏幕设备上的性能也同样重要。

4.3.3 找服务器帮忙

最后一种策略需要用到服务器——使用服务器端检测技术来决定应该下载哪一张图片。这种方法的好处是不需要去和浏览器比速度，因为所有的逻辑代码在浏览器接收到HTML之前就执行了。

但是在服务器端处理对未来也不是特别友好。随着请求你站点内容的设备的种类不断增长（这要感谢计算设备制造过程中成本的下降），维护所有设备的信息会变得越来越困难。而且越来越多的设备允许以不同的方式来查看内容，例如，通过投影仪、内嵌的网页，或者另外一个外接屏幕，导致有关设备的信息也不是那么可靠。

4.4 响应式图片的实现方法

现有的各种响应式图片的实现方法都有着各自的局限，为了描述清楚这些局限，下面我们就来看几种实现响应式图片的方法，并对它们进行评估，看看它们对于“另一个体育网站”来说是否合适。

4.4.1 Sencha.io Src

Sencha.io Src是最近似于即插即用型的响应式图片的解决方案。这项服务最初由James Pearce创建，它以你的图片作为输入，并返回缩放后的图片。要使用该服务，你只需在你的图片资源前面加上Sencha.io Src的链接即可：

```
http://src.sencha.io/http://mysite.com/images/football.jpg
```

Sencha.io Src会使用发起请求的设备的用户代理字符串来计算出设备屏幕的大小，然后根据该数值来缩放图片。默认情况下，它会将图片缩放到设备宽度的100%（尽管这样并不符合我们的要求）。

▶注意
你可以在这里找到有关Sencha.io Src的详细文档 http://docs.sencha. io/0.3.3/index.html#!/guide/src。

你可以进行大量的自定义。例如，如果你想让图片缩放到某一特定的宽度，就可以通过传递另一个参数来实现。下面这行代码将会使图片被缩放到320px宽：

```
http://src.sencha.io/320/http://mysite.com/images/football.jpg
```

Sencha.io Src足够聪明，它会缓存请求，这样便不用在每次加载页面时都要生成一次图片。

但是对于“另一个体育网站”来说，这也许并不是最佳的解决方案。将图片缩放到屏幕宽度的100%只是在小屏幕上有帮助，在大的显示器上，当文章被分为两栏时，图片却依然会是屏幕的大小，因为Sencha.io Src只看屏幕的宽度而不是容器的宽度。尽管告诉Sencha.io Src要使用容器的宽度也是可以实现的，但是这会涉及实验性质的客户端检测，而且还需要一些JavaScript技巧。

尽管现在的页面没什么问题，但是如果你除了希望能缩放图片，还想对图片进行重新裁剪的话，那么Sencha.io Src对此就会有一些限制。也许“More in Football”中需要的图片是某些点处的正方形缩略图，如果是这样的话，那么一张缩放后的图片是满足不了需求的。这里需要有一些艺术加工，但是Sencha.io Src目前还做不到。

另外，也许你也知道一些使用第三方解决方案的缺点。如果公司改变了它现有的原则或者停业了，那么你就很可能会被冷落，并且不得不再去找一整套新的解决方案。

4.4.2　自适应图片

另一个近乎于即插即用的解决方案就是由Matt Wilcox所创建的自适应图片，它会先确定屏幕的大小，然后创建并缓存一张缩放后的图片。

▶注意
关于自适应图片的代码可以在这里找到：http:// adaptive-images.com。

对于那些已有的却又没有时间去重构标签和代码的站点来说，这是一个非常合适的解决方案。使用该方案只需要下面3个简单的步骤：

（1）把.htaccess和adaptive-images.php以及需要下载的文件放在你的根目录下。

（2）创建一个ai-cache的文件夹，并赋予其写权限。

（3）将下面的JavaScript代码放到文档的头部：

```
<script>document.cookie='resolution='+Math.max(screen.width,
screen.height)+'; path=/';</script>
```

这行代码会获取屏幕的分辨率，并保存到cookie中，以便后面使用。

在adaptive-images.php中你可以配置很多选项，但通过只设置$resolutions变量能节省你大量的时间：

```
$resolutions = array(860, 600, 320); // 要使用的分辨率断点（值为屏幕的宽度，单位为像素）
```

如果仔细查看，你会发现这里的断点和“另一个体育网站”中的断点会有细微的差别。在CSS中并没有320px这个断点，而CSS中两个最大的断点——1300px和940px却没有被包含在$resolutions的数组中。这是由脚本的工作方式所决定的。

首先，脚本会为宽度小于最小断点的屏幕提供最小的图片，即这里的320px。例如，一个300px宽的屏幕会获取到一张320px宽的图片，因为320是$resolutions数组中最小的尺寸了。而一个321px宽的屏幕由于超出了320px，因此它会得到一张下一个尺寸的图片——这里是600px。如果我们将600px设置为我们的第一个断点，那么任何屏幕宽度小于600px的设备都将得到一张600px宽的图片。

其次，我们不需要用那两个最大的断点是因为虽然脚本会试着将图片缩放至断点大小，但它不会将图片放大到比实际大小还大的尺寸，所以任何比624px（图片的实际大小）大的断点都不会起作用——脚本不会根据它们去缩放图片。

图片一旦被创建，就会被保存到ai-cache文件夹（你可以更改文件夹的名字）中，这样就不必在第二次用到它们时再次生成了。这里还有另外一个配置选项可以用来指定浏览器可以缓存多久图片。

再说一次，虽然安装过程非常简单，而且对于那些已经存在的并且需要改进的站点来说这是一个伟大的解决方案，但是它也有它的缺点：由于图片是动态生成的，所以不能对图片进行艺术指导裁剪。

艺术指导和响应式图片

大部分有关响应式图片的讨论都是有关文件大小的，虽然这的确是一个需要重点考虑的因素，但它不是唯一要考虑的，因为有时为小屏幕缩放图片反而会降低性能。

考虑一下这张橄榄球头盔的照片吧。

在初始大小时，照片看起来很不错，并且构图也相当平衡。但如果我们缩小它，头盔瞬间就变得几乎难以辨认了。

这里是需要艺术指导的典型例子。单单只是缩小图片会导致图片失去它原有的视觉冲击力和可辨识度。但是经过特殊裁剪后，即使图片变小了，我们依然可以让头盔成为焦点。

对于高分辨率的屏幕而言，如果图片比较小，那么脚本也帮不上什么忙。对于“另一个体育网站”来说这也是个问题。当屏幕宽度大于1300px时，文章就会变成两列，图片应该只在其中一列中显示，而且图片应该是被缩小了的。但是如果使用自适应图片脚本的话，此时浏览器下载的依然会是最大的版本的图片。

● 内容分发网络（Content Delivery Network）
为了更高效地将内容分发给用户而在不同地点部署的服务器集群。

对于该方法，人们关心的另外一个问题是：无论请求的是多大的图片，其URL始终都是相同的，这会在内容分发网络（CDN）中造成问题。在第一次请求某个URL时，CDN可能会缓存该URL从而在下次该URL被再次请求时提高速度。但是如果多个不同大小的图片在同一个CDN中都对应同一个请求，那么CDN可能会返回缓存了的图片，而不是我们真正想要的尺寸的图片。

响应式图片的未来如何？

明确地说：依靠服务器端检测并结合使用JavaScript cookie的方法完全是一个头痛医头、脚痛医脚的方法。如果有其他更加长期的方法，那么我主张去用其他方法。但不幸的是，目前每一种响应式图片的实现方法本质上都只是一个技巧，只是治标不治本地掩盖了问题。

人们已经开始了对更长期的解决方案的讨论，例如，创建一个新的元素、新的属性或者新的图像格式。其实，如果你已经对此跃跃欲试了，在GitHub（https://github.com/scottjehl/picturefill）上有这样一个还不存在的元素，可以提供一个全功能的polyfill。但遗憾的是，这个问题还远没有解决，因为答案远远不是“什么对开发人员来说更容易使用”那么简单。

在一篇讨论两种流行的解决方案的冲突的博客文章中，Jason Grigsby在开头处便一语中的：为了提高性能，浏览器希望在知道页面的布局前，就尽可能快地加载网页；但另一方面开发者们却要依靠他们关于页面布局的知识，来确定要下载哪张图片。这真是一块难啃的骨头。

我相信，随着时间的推移，将会出现一个妥善的解决方案。但在此之前，正如前面已经提到的那样，最好的解决方案还是要取决于你手中的项目。

4.4.3 等等，答案是什么

目前并不存在明确的响应式图片的解决方案，每一种方案都有其优点和缺点。你

选取的方法取决于你手中的项目。

在我们已经讨论过的这两种方法中，自适应图像可能是比较好的选择，因为它不需要依赖任何第三方资源。

4.5　背景图片

“另一个体育网站”上的人们已经非常喜欢现在的站点了，但他们还希望在页面顶部能有一个视觉提示，以帮助他们确定自己现在还在这个网站。

经过30秒在Photoshop中令人精疲力竭的工作之后，我们为人们准备好了一张如图4.4所示的有两个橄榄球剪影的背景图片。

在大屏幕上浏览时，人们很喜欢这个图片，但是对于那些宽度小于53.75em的屏幕来说，logo就和图片叠在了一起，所以这时他们不希望有背景图片。

这里是另外一处得益于从移动端向上设计的地方，让我们来看看如果采用从桌面端向下设计的媒介查询会发生什么。

基本样式负责把图片包含进来，并且在小屏幕上你不得不用一个媒介查询来将基本样式覆盖掉。其代码大概会是这个样子：

```
1.  /* base styles */
2.  header[role="banner"] .inner{
3.       background: url('../images/football_bg.png') bottom right
    no-repeat;
4.  }
5.  ....
6.  @media all and (max-width: 53.75em) {
7.       header[role="banner"] .inner {
8.            background-image: none;
9.       }
10. }
```

图4.4　网页的头部在炫耀它新的、漂亮的背景图片。

也许在纸上这看起来不错，但是对于很多现实中的浏览器来说，这将导致那些用不到这张图片的小屏幕设备上也会下载该图片，其中最为明显的就是Android 2.x上自带的浏览器。记住，虽然在写作这本书时Android的最新版本是4，但是大约有95%的Android设备仍旧运行着之前版本的系统。这意味着在移动设备中，几乎所有的Android流量都将会下载该图片，却不会用到它。

为了避免这种浪费，一种更好的方法是像下面这样——在媒介查询中声明背景图片：

```
/* base styles */
@media all and (min-width: 53.75em) {
     header[role="banner"] .inner{
          background: url('../images/football_bg.png') bottom right
          no-repeat;
     }
}
....
@media all and (max-width: 53.75em) {
    header[role="banner"] .inner {
          background-image: none;
     }
}
```

这样就可以让背景图片在Android上良好地运行了。

▶注意

如果你想知道各种替换或者隐藏背景图片方法的细节，这里有我所做测试的结果表：http://timkadlec.com/2012/04/media-query-asset-downloading-results/。

当我使用从移动端向上的设计方法时，整个过程会简单很多，因为基本体验是不需要背景图片的，所以我们可以在之后的媒介查询中再将背景图片加入进来：

```
    /* base styles */
     @media all and (min-width: 53.75em) {
          header[role=banner] .inner{
                    background: url('../images/football _ bg.png')bottom right
                    no-repeat;
          }
    }
```

使用这种方法意味着，只有那些需要用到背景图片的浏览器才会发送请求下载它——性能问题解决了！

当然，创建从移动端向上的设计意味着IE 8及以下版本的IE在默认情况下是无法看到背景图片的。然而，托条件注释的福，我们已经有了一个针对IE的样式表，我们在那里加上相关的声明就可以了。

既然我们已经到了这里

现在，我们用Web字体来加载ChunkFive字体，并且在头部的元素中使用该字体。样式声明如下所示：

```
@font-face {
      font-family:'ChunkFiveRegular';
      src: url('Chunkfive-webfont.eot');
      src: url('Chunkfive-webfont.eot?#iefix') format
      ('embedded-opentype'),
            url('Chunkfive-webfont.woff') format('woff'),
            url('Chunkfive-webfont.ttf') format('truetype'),
            url('Chunkfive-webfont.svg#ChunkFiveRegular')
format('svg');
      font-weight: normal;
      font-style: normal;
}
```

▶ 注意

为什么字体文件都各不相同呢？这要感谢浏览器的不同。虽然浏览器对字体的支持很好，但它们好像没有在文件格式上达成一致。

上面的声明运行得还不错：浏览器抓取它需要的文件，然后渲染字体，文件大小也不错，但还是有个缺点。目前，基于WebKit的浏览器在下载好Web字体之前，是不会显示使用该Web字体格式化的文本的。这就意味着如果用户使用的是一个连接速度很慢的Android设备、黑莓或者iPhone时（或者是有线连接但连接速度也很慢的笔记本上时），需要花费一段时间来显示头部的元素，这会让用户感到迷惑，所以应该尽量避免。

虽然我们无法决定带宽（但可以先查看一下第9章“响应式体验”，看看将会提到什么），但是我们知道移动设备的网速较慢的可能性最高。如果能为用户省去不必要的麻烦并实现只在大屏幕上下载字体，我想这会是个很好的主意。

我们之前有条件地下载背景图片的方法对于字体来说也同样奏效，所以我们可以将@font-face的声明也放到媒介查询中去。这样做可以确保那些屏幕宽度低于断

点的设备不会尝试下载字体：

```
@media all and (min-width: 37.5em) {
      ...
      @font-face {
           font-family: 'ChunkFiveRegular';
           src: url('Chunkfive-webfont.eot');
           src: url('Chunkfive-webfont.eot?#iefix')
           format('embedded-opentype'),
           url('Chunkfive-webfont.woff') format('woff'),
                url('Chunkfive-webfont.ttf') format('truetype'),
                url('Chunkfive-webfont.svg#ChunkFiveRegular')
format('svg');
           font-weight: normal;
           font-style: normal;
      }
}
```

经过这个小小的调整，只有那些屏幕宽度大于37.5em（大约是600px）的设备才会去下载Web字体。虽然对于使用较慢连接速度的用户来说，仍然有可能被WebKit的加载漏洞卡在那里，但是通过为小屏幕删除字体，我们也已经删除了最有可能的受害者——那些使用移动设备的人们（图4.5）。

图4.5 出于性能的考虑，小屏幕设备将不会下载Web字体。

4.6　高分辨率屏幕

如果你觉得基于屏幕尺寸来转换图片并不是那么困难，那么至少还有另外一种情形也要求有不同大小的图片：高分辨率屏幕。虽然问题始于iPhone 4S的Retina屏幕，但是随着iPad 3和最新版本的MacBook Pro也都采用了Retina屏幕，该问题也进一步加剧了。

Retina屏幕的像素密度达到了惊人的326ppi（每英寸的像素数），而iPhone 3的屏幕只有163ppi。这样高的密度意味着图片的显示效果将会异常的细致而清晰——如果图片为此做了优化的话。如果图片没有做优化，那么它们的显示效果将会是颗粒状并且是模糊的。

● **像素密度**
特定空间内的像素数。例如，326ppi意味着每英寸的显示屏里有326个像素。

为高分辨率屏幕创建图片就意味着要创建面积较大的图片，同时也就意味着图片的文件大小也会很大。难点在于——你不想把这些大文件传输给那些不需要它们的显示器。目前还没有好的解决HTML内图片的方法，这和我们前面讨论过的根据不同的屏幕宽度来加载适当图片的问题相同。

对于CSS里的图片来说，你可以为非WebKit浏览器使用min-resolution媒介查询；对于基于WebKit的浏览器，你必须使用–webkit-min-device-pixel-ratio媒介查询。

–webkit-min-device-pixel-ratio媒介查询需要赋予其一个十进制的表示像素比的值 。对于iPhone、iPad或者新的MacBook Pro来说，你需要对其赋值至少为2才行。

min-resolution媒介查询需要两个值中的一个，第一个值是屏幕的分辨率，单位可以是每英寸的点数，也可以是每厘米的点数。这里需要用到一些数学计算，而且早期的一些实现方法并不是很准确，因此我建议你使用新的每像素的点数（dppx）作为单位。这样不仅免去了计算（它能与–webkit-min-device-pixel-ratio媒介查询可以接受的参数完美地结合），而且避免了那些过时的、不准确的实现方法。虽然浏览器对于每像素的点数这个单位的支持仍有待提高，但是，由于显示准Retina屏幕的图片是一种很好的功能上的增强，而不是一个必要的特性，所以我可以放心地使用它。

```
header[role="banner"] .inner {
     background: url('../images/football_bg_lowres.png') bottom right
     no-repeat;
}
@media only screen and (-webkit-min-device-pixel-ratio: 2),
     only screen and (min-resolution: 2dppx) {
          header[role="banner"] .inner {
               background: url('../images/football_bg_highres.png')
               bottom right no-repeat;
          }
}
```

任何像素比至少为2的设备都将会应用上面的媒介查询。第1至3行代码为低分辨率的屏幕设置了背景图片，第4至5行代码将判断屏幕的像素比是否至少为2，如果是，那么第8到10行代码会为其指定一个高分辨率的背景图片。

SVG

对于高分辨率屏幕上的显示问题，以及图片在不同尺寸屏幕上显示时的可伸缩问题，可以将可伸缩矢量图形（SVG）作为一个解决方案。SVG是使用XML来定义的矢量图形，也就是说它们可以在不增加实际文件大小的前提下自如缩放，这也同时意味着可以通过编程来改变或者调整它们。

一个关于SVG如何能提高使用体验的例子就是Yiibu，这是一个由一家位于爱丁堡的移动公司为皇家格林尼治天文台开发的应用。该公司开发的项目包括一个响应式的站点，该站点需要能够按需缩小星座图片的大小。使用正常的图片并缩小时，小屏幕上的图片便会丢失很多细节。然而在使用了SVG以及其他一些聪明的缩放技巧之后，Yiibu就可以在保留细节的同时，并为小屏幕做出一些调整了（图4.6）。

在使用SVG的过程中会遇到两个实际的问题：浏览器的支持情况不理想以及相应工具的缺乏。像以往一样，IE 8以及更低版本的IE浏览器不支持SVG，更重要的是Android 2.x——最流行的平台上默认的浏览器也不支持SVG，而那些支持SVG的浏览器其支持程度和质量也参差不齐。

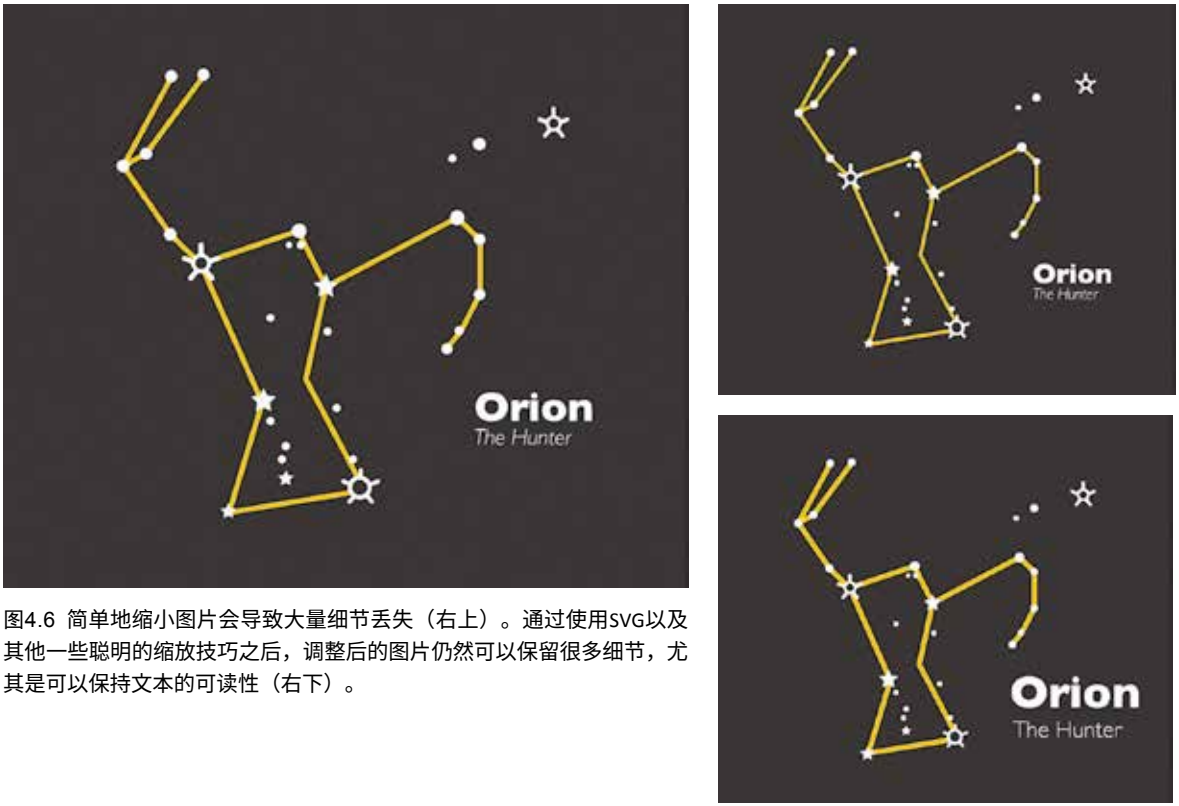


图4.6 简单地缩小图片会导致大量细节丢失（右上）。通过使用SVG以及其他一些聪明的缩放技巧之后，调整后的图片仍然可以保留很多细节，尤其是可以保持文本的可读性（右下）。

例如，像Photoshop这样最流行的图像创建和编辑软件，也并没有考虑SVG这样的矢量格式。如果你想创建SVG图像，你还需要找其他工具来实现。

但随着工具和浏览器都开始追赶上来，SVG图像也许将会成为Web开发者的工具箱中一种非常普通的工具。

4.7　其他固定宽度的内容

对于响应式站点来说，图片并不是唯一存在问题的内容。让我们来看看另外两个典型的内容：视频和广告。

4.7.1 视频

也许会令人感到意外，但是在响应式站点内嵌入视频确实要比它第一次出现时还要复杂一点。如果你使用的是HTML5的视频标签，那么问题会简单一些。你可以使用前面我们讨论过的让图片变得灵活的max-width技术：

```
1.  video{
2.        max-width: 100%;
3.        height: auto;
4.  }
```

然而，大多数的站点中会用iFrame来嵌入第三方（例如YouTube或者Vimeo）的视频。如果在这样的页面中使用上面提到的方法，虽然改变了宽度，但是高度仍然会维持原始的值，这便破坏了视频的比例（图4.7）。

图4.7 不幸的是，在嵌入式视频上使用max-width: 100%和height: auto将导致破坏视频的高宽比。

对此有一些被Thierry Koblentz称为“固定比例”的相关技巧。其基本思想是：包含视频的容器要有适合于视频的比例（例如4:3，16:9，等等），然后视频需要适应容器的尺寸。这样，当容器的宽度改变时仍然可以保持原来的比例，并且强制视频通过调整自身来适应它。

▶注意

如果你愿意，这里有一个有用的叫做FitVids的jQuery插件，它可以自动执行响应视频的过程。你可以从GitHub上下载它：https:// github.com/dava- tron5000/FitVids.js。

该方法的第一个步骤是创建一个容器元素：

```
<div class="vid-wrapper">
      <iframe></iframe>
</div>
```

作为一个包含盒，容器元素需要维持合适的比例，在这个例子中其比例为16:9。由于视频本身是绝对定位的，所以容器需要用适当的padding来维持比例。为了保持16:9的比例，我们用9除以16，得到56.25%。

```
.vid-wrapper{
      width: 100%;
      position: relative;
      padding-bottom: 56.25%;
      height: 0;
}
.vid-wrapper iframe{
      position: absolute;
      top: 0;
      left: 0;
      width: 100%;
      height: 100%;
}
```

上面的样式将容器内的iFrame设置为绝对定位，并且将宽和高设置为100%，这样它就可以通过伸缩来填充容器了（第11至12行）。而容器本身的宽度则被设置成了文章的宽度，这样它就可以根据屏幕尺寸来自动调整了。

有了这些样式之后，视频就可以在不同大小的屏幕上维持它原始的比例了。

增强使用体验

像往常一样，又到了停下来思考一下该如何增强使用体验的时候了。现在，视频会被下载到所有的设备上，这也许并不是最好的使用体验。为了提升核心体验，先只显示一个视频链接也许会是个不错的选择，然后再为那些大屏幕设备将视频包含到页面当中来。

我们先从一个简单的链接开始：

```
<a id="video" href="http://www.youtube.com/watch?v=HwbE3bPvzr4">
```

```
Video highlights</a>
```

你也可以为文字链接添加一些样式来让它们看起来不那么过时：

```
.vid{
      display: block;
      padding: .3em;
      margin-bottom: 1em;
      background: url(../images/video.png) 5px center no-repeat #e3e0d9;
      padding-left: 35px;
      border: 1px solid rgb(175,175,175);
      color: #333;
}
```

这里并不需要有多棒的想象力，我们给链接增加了一些padding和margin来让它和其他内容保持一段距离，我们还给它的左侧增加了一张有视频图标的背景图片（图4.8）。

现在，该把链接转换成为合适的嵌入式视频了。

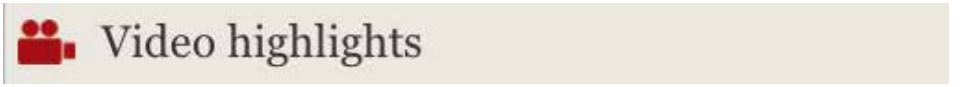


图4.8 增加了样式规则之后，视频链接与页面上其他的内容之间就更加协调了。

将下面的函数添加到yass.js中的Utils对象中去：

```
getEmbed : function(url){
      var output = '';
      var youtubeUrl = url.match(/watch\?v=([a-zA-Z0-9\-_]+)/);
      var vimeoUrl = url.match(/^http:\/\/(www\.)?vimeo\.com\
      /(clip\:)?(\d+).*$/);
      if(youtubeUrl){
            output = '<div class="vid-wrapper"><iframe src="http://
             www.youtube.com/embed/'+youtubeUrl[1]+'?rel=0"
             frameborder="0" allowfullscreen></iframe></div>';
            return output;
      } else if(vimeoUrl){
          output =  '<div class="vid-wrapper"><iframe src="http://
          player.vimeo.com/video/'+vimeoUrl[3]+'" frameborder="0">
          </iframe></div>';
           return output;
      }
}
```

让我们来看一下这个函数。

该函数只有视频的URL这唯一的一个参数，之后函数会使用正则表达式来确定该URL是否是YouTube或者Vimeo的视频（第4至5行）。根据URL的类型，函数会创建包含容器元素的嵌入元素，并作为返回值返回（第5至11行）。

有了getEmbed函数，我们就可以很容易地将视频链接转换为一个嵌入视频了。将下面的JavaScript代码放到matchMedia("(min-width:37.5em)")中进行测试：

```
1.  //load in the video embed
2.  var videoLink = document.getElementById('video');
3.  if (videoLink) {
4.        var linkHref = videoLink.getAttribute('href');
5.        var result = Utils.getEmbed(linkHref);
6.        var parent = videoLink.parentNode;
7.        parent.innerHTML = result + videoLink.parentNode.innerHTML;
8.        parent.removeChild(document.getElementById('video'));
9.  }
```

开头的两行代码获得了视频元素以及链接中href的值，链接地址在第5行处被传递给了我们创建的getEmbed函数。一旦获得了返回的结果，第6至8行代码就会把视频插入到文章中去，然后删除原有的文字链接（图4.9）。

图4.9 在大屏幕上（左）视频被嵌入到了页面里，但是在小屏幕上只能看到一个链接。

现在，嵌入式视频也成为响应式的了，并且只有当屏幕尺寸大于37.5em时才会载入视频，以此来确保在基本体验中，不会发起代价高昂的HTTP请求并嵌入视频。

4.7.2 广告

另外一个需要修补且修补起来有些困难的元素就是广告。

无论你喜不喜欢，对于很多在线公司的收入来说，广告都是关键的一部分。我们不想在这里卷入基于广告收入模式和支付内容模式的讨论，因为这样的讨论将很快变得丑陋不堪。现实情况是，对于许多企业而言，广告收入都是至关重要的。

从纯粹的技术角度来看，响应式布局中的广告并不是特别难以实现，你可以使用JavaScript基于屏幕尺寸有条件地加载广告。来自纽约的开发者Rob Flaherty为我们演示了一种基本的方法：

```
1.  // Ad config
2.  var ads = {
3.       leaderboard: {
4.            width: 728,
5.            height: 90,
6.            breakpoint: false,
7.            url: '728x90.png'
8.       },
9.       rectangle: {
10.           width: 300,
11.           height: 250,
12.           breakpoint: 728,
13.           url: '300x250.png'
14.      },
15.      mobile: {
16.           width: 300,
17.           height: 50,
18.           breakpoint: 500 ,
19.           url: '300x50.png'
20.      }
21. };
```

上面的配置代码设置了3种不同的广告（通栏广告、边栏矩形广告、手机广告），每种广告都有着各自的宽度（第4、10和16行）、高度（第5、11和17行）、URL（第7、13和19行）以及广告应该被加载处的断点值（第6、12和18行）。你可以通过使用matchMedia函数并基于断点来决定要下载哪个广告。

更棒的是广告自身是响应式的，即可以用HTML和CSS来实现可以根据不同的屏幕尺寸来调整自身大小的广告。这样的实现不仅可以消除对于JavaScript的依赖，而且借助于广告的互动性，广告还能做一些很酷的事情。

从技术的角度来来看，上面这些广告都不难实现，难的是创建并显示包含许多动态内容的广告。

现有的大部分广告都来自于第三方网络，或者广告创意是在系统外部开发并在特定的站点来发布的。目前，还没有主流的广告网络可以根据屏幕尺寸来提供不同尺寸的广告。

虽然使用内部的广告平台会更加灵活，但是如果创意研发来自于公司之外，那么你就要乐意为他们做一些培训了。创作广告的人们也许并没有跟上最新的网页发展趋势。

更重要的是，网上的广告和印刷品上的广告非常相似：都是根据广告的大小和位置来付款的。那么，当广告的大小和位置都会变化时，到底该如何付款呢？

一个解决方案是卖一组广告，而不是单个广告。例如，你可以卖一个超级广告包（或者取一个你喜欢的名字），而不是卖一个摩天楼广告。超级广告包里也许包括一个为宽度在900px以上的屏幕准备的摩天楼广告，一个为600px到900px之间屏幕准备的盒式广告，以及为更小屏幕准备的小的标题栏广告。

显然这不会是一个容易的转变。创意团队、决策层以及销售人员都需要相关的培训，来认识到为什么这种方法比买一个特定的广告更加有效。这不是一桩容易的买卖，但随着时间的推移，它应该会变得越来越容易。

这里会涉及的另外一个考虑因素是，有些公司也许只想把某一单独的平台作为其推广目标。也许他们的服务是只针对移动设备的，并且他们决定只在那些有着较小屏幕的设备上做广告。这给广告包带来了一点麻烦，使得事情在刚刚开始时就麻烦重重。

最后，我希望看到关于响应式广告是会导致减少广告的插播还是更高的广告费用的讨论。那些收入来自广告的站点经常因为大量的广告而导致页面过载，当他们尝试设计小屏幕上的体验时，情况会变得更加困难。你是要隐藏所有这些广告来限制广告商们的页面访问量，还是要让广告填满屏幕来破坏访客的体验？

要减少页面上广告的数量，而不是在页面上加载越来越多的广告。可以在页面上

提供3个每个每月4000美元的广告位，而不是10个每个每月1000美元的广告位。要让广告位值得人们追捧。这样既有利于广告商，因为没有了其他广告对人们注意力的争抢，也有利于用户，因为他们有了更好的使用体验。

不幸的是这里有个鸡和蛋的问题：目前的广告率一直走低，广告要努力获得高质量的点击率，而这样的竞争就是要看每条广告的成本能降低多少。这需要有足够大胆的人来走出第一步。

4.8 结束语

无论是什么类型的站点，性能都是一个需要重点考虑的问题。加载不需要的或者超过所需的图片会对页面加载时间产生严重的影响。

使用CSS的display:none的解决方案并不可行，虽然它从视觉上隐藏了图片，但是图片仍然会被请求并下载。如果你想让图片只在屏幕大于某一特定的点时才显示，在页面加载后有条件地加载图片会是更好的办法。

目前的响应式图片还存在着种种问题，人们尝试过的每一种解决方法都有其自身的一系列问题。你能做的最好的事情就是在每个项目开始前，仔细考虑一下对于手中的站点来说哪种方法更适用。

若要隐藏背景图片，那就压根不要去下载它，包括在媒介查询中也是一样的。在基本体验中设置背景图片再将它们隐藏，将会导致在大多数情况下图片都会被下载下来。

例如在最新的iPhone、iPad和MacBook Pro上的Retina这样的高分辨率显示器又向我们提出了新的挑战。我们的解决方案之一是基于CSS的图片，可以通过min-resolution媒介查询来实现。

视频和广告也是人们关心的重点。对于视频来说，使用固定比例的方法可以让你根据屏幕尺寸适当地缩放视频。与往常一样，要有意识地关注性能。最好能够为小屏幕用户显示视频链接，而为大屏幕用户直接显示嵌入的视频。

对于广告来说，解决技术上的挑战并不困难。如果你是从自己系统中加载广告的，JavaScript或者一些响应式的HTML和CSS都可以为在不同分辨率的屏幕上为改变广告而提供帮助。更大的问题是如何把销售团队和第三方广告网络也拉上船。