Java Web 高性能开发，第 1 部分: 前端的高性能

魏 强, 研究生, 东北大学

简介: Web 发展的速度让许多人叹为观止，层出不穷的组件、技术，只需要合理的组合、恰当的设置，就可以让 Web 程序性能不断飞跃。所有 Web 的思想都是通用的，它们也可以运用到 Java Web。这一系列的文章，将从各个角度，包括前端高性能、反向代理、数据库高性能、负载均衡等等，以 Java Web 为背景进行讲述，同时用实际的工具、实际的数据来对比被优化前后的 Java Web 程序。第一部分，主要讲解网页前端的性能优化，这一部分是最直接与用户接触的。事实证明，与其消耗大量时间在服务器端，在前端进行的优化更易获得用户的肯定。

发布日期: 2011 年 10 月 24 日
级别: 中级
访问情况: 17642 次浏览
评论: 10 (查看 | 添加评论 - 登录)

平均分 (92个评分)
为本文评分

引言

前端的高性能部分，主要是指减少请求数、减少传输的数据以及提高用户体验，在这个部分，图片的优化显得至关重要。许多网站的美化，都是靠绚丽的图片达到的，图片恰恰是占用带宽的元凶。每个 img 标签，浏览器都会试图发起一个下载请求。本文就详细介绍了图片优化的几种方式，介绍了使用的工具以及优化后的结果。

图片压缩

减少图片的大小，可以明显的提高性能，而对于已有图片，要想减少图片的大小，只能改变图片的格式，这里推荐的是 PNG8 的格式，它可以在基本保持清晰度的情况下，减少图片的大小。知道这个原理以后，可以用 Windows 的画图工具、以及 PhotoShop 工具逐个的改变。但是这样做的缺点是单张处理，效率太慢。本文推荐一个在线转换工具 Smush.it，可以批量的进行压缩与转换。它的地址是：www.smushit.com/ysmush.it。打开后效果如下图所示。

图 1. Yahoo 提供的在线压缩工具

我们上传了一张大小为 3790K 的图片，待在线程序处理完毕后，点击 Download Smushed Images 下载查看结果。下载界面如下图所示。

图 2. 压缩后的结果

打开下载下来的压缩包，查看结果可以看到，图片从 3790 减少到了 3344，就如下图所示。对于大批量的图片网站，这个方法会帮助快速实现批量图片压缩。

图 3. 压缩后的结果

图像合并实现 CSS Sprites

CSS Sprites 是一个吸引人的技术，它其实就是把网页中一些背景图片整合到一张图片文件中，再利用 CSS 的“background-image”，“background- repeat”，“background-position”的组合进行背景定位，background-position 可以用数字能精确的定位出背景图片的位置。利用 CSS Sprites 能很好地减少网页的 HTTP 请求，从而大大的提高了页面的性能，这也是 CSS Sprites 最大的优点，也是其被广泛传播和应用的主要原因。CSS Sprites 能减少图片的字节，由于图像合并后基本信息不用重复，那么多张图片合并成 1 张图片的字节往往总是小于这些图片的字节总和。同时 CSS Sprites 解决了网页设计师在图片命名上的困扰，只需对一张集合的图片上命名就可以了，不需要对每一个小元素进行命名，从而提高了网页的制作效率。更换风格方便，只需要在一张或少张图片上修改图片的颜色或样式，整个网页的风格就可以改变。维护起来更加方便。同时，由于将图片合并到一张图片，因此图片的请求数就被缩减到 1 个。其他的请求都可以用到本地缓存，不需要访问服务器。下图是一个合并以后的图片。它将很多小图标都拼到了一起。

图 4. 合并后的图片

这里介绍一个小工具 ---“CSS Sprites 样式生成工具 2.0”，可以从 这里下载。这是一个简单免费的小工具，用该工具打开上面的图片，选中图片中的某块。如下图的“绿色大拇指”部分，工具会计算出这个部分的长、宽、距离左上角的距离。勾选复制类名、复制宽、复制高，再点击“复制当前样式”按钮。这样生成的样式会被复制到剪切板上。

图 5. 小工具的使用

生成的 CSS 代码如清单 1 所示。

清单 1. 小工具生成的 CSS 代码

```
.div_6148{width:18px;height:20px;background-position:-17px -209px;}
```

将这段代码运用在网页上，它的代码如下清单所示。

清单 2. 测试 CSS Sprites 代码

```
<html>
<head>
    <style>
          .div_6148
          {
             width:18px;
             height:20px;
             background-image:url(css-sprites-source.gif);
             background-position:-17px -209px;
  }
    </style>
</head>
<body>
<div class="div_6148"></div>
</body>
</html>
```

打开测试网页显示结果如下图所示。

图 6. 测试网页效果

可以看到，网页只显示工具选择的“绿色大拇指”部分，这样的代码可以运用在网页的多个部分，而图片只需要下载一次，这就是该技术的最大优势，减少了因为小图片引起的多个请求。

多域名请求

有时候，图片数据太多，一些公司的解决方法是将图片数据分到多个域名的服务器上，这在一方面是将服务器的请求压力分到多个硬件服务器上。另一方面，是利用了浏览器的特性。一般来说，浏览器对于相同域名的图片，最多用 2-4 个线程并行下载。不同浏览器的并发下载数，都是不同的，并发数如下清单所示。

清单 3. 各浏览器的并发下载数

```
Browsers                    HTTP/1.1    HTTP/1.0
IE6,7                        2           4
IE8                          6           6
FireFox 2                    2           8
FireFox 3                    6           6
Safari 3,4                   4           4
Chrome 1,2                   6           ?
Chrome 3                     4           4
Opera 9.63,10.00alpha        4           4
```

而相同域名的其他图片，则要等到其他图片下载完后才会开始下载。 这里我做了一个测试，选择了多个相同域名的图片在同一网页上。代码如清单 5 所示。

清单 4. 单域名的多图片下载

```
<html>
<body>
<img src="http://img1.gtimg.com/news/pics/hv1/123/231/804/52339128.jpg"><br>
<img src="http://img1.gtimg.com/news/pics/hv1/87/235/804/52340112.jpg"><br>
<img src="http://img1.gtimg.com/finance/pics/hv1/41/119/804/52310486.jpg"><br>
<img src="http://img1.gtimg.com/sports/pics/hv1/246/198/804/52330836.jpg"><br>
<img src="http://img1.gtimg.com/ent/pics/hv1/101/54/805/52358996.jpg"><br>
<img src="http://img1.gtimg.com/blog/pics/hv1/169/226/804/52337899.jpg">
</body>
</html>
```

接下来，使用 FireFox 的 Firebug 插件监控网络。结果如下图所示。

图 7. 单域名多图片的监控效果

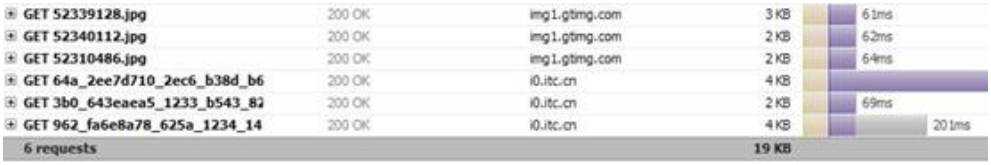
可以看到，相同域名的多张图片，它们下载的起始点是存在延迟的。它们并不是并行下载。当我们将其中的 3 张图片换成别的域名图片。如清单 6 所示。

清单 5. 多域名多图片下载

```
<html>
<body>
<img src="http://img1.gtimg.com/news/pics/hv1/123/231/804/52339128.jpg"><br>
<img src="http://img1.gtimg.com/news/pics/hv1/87/235/804/52340112.jpg"><br>
<img src="http://img1.gtimg.com/finance/pics/hv1/41/119/804/52310486.jpg"><br>
<img src="http://i0.itc.cn/20110624/64a_2ee7d710_2ec6_b38d_b678_dc3af28392be_1.jpg"><br>
<img src="http://i0.itc.cn/20110624/3b0_643eaea5_1233_b543_82b7_9c7273c7f97c_1.jpg"><br>
<img src="http://i0.itc.cn/20110623/962_fa6e8a78_625a_1234_147f_3a627fe17033_1.jpg">
</body>
</html>
```

再次查看网络监控，可以看到，这些图片是并行下载的。

图 8. 多域名多图片测试结果



多域名的下载固然很好，但是太多域名并不太好，一般在 2-3 个域名下载就差不多。

图像的 BASE64 编码

不管如何，图片的下载始终都要向服务器发出请求，要是图片的下载不用向服务器发出请求，而可以随着 HTML 的下载同时下载到本地那就太好了。而目前，浏览器已经支持了该特性，我们可以将图片数据编码成 BASE64 的字符串，使用该字符串代替图像地址。假设用 *S*代表这个 BASE64 字符串，那么就可以使用 <img src="*S*"> 来显示这个图像。可以看出，图像的数据包含在了 HTML 代码里，无需再次访问服务器。那么图像要如何编码成 BASE64 字符串呢？可以使用 在线的工具---“Base64 Online”，这个工具可以上传图片将图片转换为 BASE64 字符串。当然，如果读者有兴趣，完全可以自己实现一个 BASE64 编码工具，比如使用 Java 开发，它的代码就如清单 7 所示。

清单 6. BASE64 的 Java 代码

```
public static String getPicBASE64(String picPath) {
        String content = null;
        try {
            FileInputStream fis = new FileInputStream(picPath);
            byte[] bytes = new byte[fis.available()];
            fis.read(bytes);
            content = new sun.misc.BASE64Encoder().encode(bytes); // 具体的编码方法
            fis.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return content;
}
```

本文编码了一个图像，并且将编码获得的 BASE64 字符串，写到了 HTML 之中，如下清单 8 所示。

清单 7. 嵌入 BASE64 的测试 HTML 代码

```
<html>
<body>
<img src="
iVBORw0KGgoAAAANSUhEUgAAAeQAAAB8BAMAAABKwt5QAAAAA3NCSVQICAjb4U/gAAAAGFBMVEX/
……（省略了大部分编码）… BJRU5ErkJggg==">
</body>
</html>
```

由于图片数据包含在了 BASE64 字符串中，因此无需向服务器请求图像数据，结果显示如下图所示。

图 9. BASE64 显示图像

然而这种策略并不能滥用，它适用的情况是浏览器连接服务器的时间 > 图片下载时间，也就是发起连接的代价要大于图片下载，那么这个时候将图片编码为 BASE64 字符串，就可以避免连接的建立，提高效率。如果图片较大的话，使用 BASE64 编码虽然可以避免连接建立，但是相对于图像下载，请求的建立只占很小的比例，如果用 BASE64，对于动态网页来说图像缓存就会失效（静态网页可以缓存），而且 BASE64 字符串的总大小要大于纯图片的大小，这样一算就非常不合适了。因此，如果你的页面已经静态化，图像又不是非常大，可以尝试 BASE64 编

码，客户端会将网页内容和图片的 BASE64 编码一起缓存；而如果你的页面是动态页面，图像还较大，每次都要下载 BASE64 字符串，那么就不能用 BASE64 编码图像，而正常引用图像，从而使用到浏览器的图像缓存，提高下载速度。从现实我们接触的角度看，如一些在线 HTML 编辑器，里面的小图标，如笑脸等，都使用到了 BASE64 编码，因为它们非常小，数量多，BASE64 可以帮助网页减少图标的请求数，提高效率。

GZIP 压缩

为了减少传输的数据，压缩是一个不错的选择，而 HTTP 协议支持 GZIP 的压缩格式，服务器响应的报头包含 Content-Encoding: gzip，它告诉浏览器，这个响应的返回数据，已经压缩成 GZIP 格式，浏览器获得数据后要进行解压缩操作。这在一定程度可以减少服务器传输的数据，提高系统性能。那么如何给服务器响应添加 Content-Encoding: gzip 报头，同时压缩响应数据呢？如果你用的是 Tomcat 服务器，打开 $tomcat_home$/conf/server.xml 文件，对 Connector 进行配置，配置如清单 9 所示。

清单 8. TOMCAT 配置清单

```
<Connector  port ="80"  maxHttpHeaderSize ="8192"
maxThreads ="150"  minSpareThreads ="25"  maxSpareThreads ="75"
enableLookups ="false"  redirectPort ="8443"  acceptCount ="100"
connectionTimeout ="20000"  disableUploadTimeout ="true"  URIEncoding ="utf-8"
compression="on"
compressionMinSize="2048"
noCompressionUserAgents="gozilla, traviata"
compressableMimeType="text/html,text/xml" />
```

我们为 Connector 添加了如下几个属性，他们意义分别是：

compression="on" 打开压缩功能

compressionMinSize="2048" 启用压缩的输出内容大小，这里面默认为 2KB

noCompressionUserAgents="gozilla, traviata" 对于以下的浏览器，不启用压缩

compressableMimeType="text/html,text/xml, image/png" 压缩类型

有时候，我们无法配置 server.xml，比如如果我们只是租用了别人的空间，但是它并没有启用 GZIP，那么我们就要使用程序启用 GZIP 功能。我们将需要压缩的文件，放到指定的文件夹，使用一个过滤器，过滤对这个文件夹里文件的请求。

清单 9. 自定义 Filter 压缩 GZIP

```
// 监视对 gzipCategory 文件夹的请求
@WebFilter(urlPatterns = { "/gzipCategory/*" })
public class GZIPFilter implements Filter {

@Override
public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
String parameter = request.getParameter("gzip");
// 判断是否包含了 Accept-Encoding 请求头部
HttpServletRequest s = (HttpServletRequest)request;
String header = s.getHeader("Accept-Encoding");
//"1".equals(parameter) 只是为了控制，如果传入 gzip=1，才执行压缩，目的是测试用
if ("1".equals(parameter) && header != null && header.toLowerCase().contains("gzip")) {
HttpServletResponse resp = (HttpServletResponse) response;
final ByteArrayOutputStream buffer = new ByteArrayOutputStream();

HttpServletResponseWrapper hsrw = new HttpServletResponseWrapper(
resp) {

@Override
public PrintWriter getWriter() throws IOException {
return new PrintWriter(new OutputStreamWriter(buffer,
getCharacterEncoding()));
}

@Override
public ServletOutputStream getOutputStream() throws IOException {
return new ServletOutputStream() {
```

```
@Override
public void write(int b) throws IOException {
buffer.write(b);
}
};
}

};

chain.doFilter(request, hsrw);
byte[] gzipData = gzip(buffer.toByteArray());
resp.addHeader("Content-Encoding", "gzip");
resp.setContentLength(gzipData.length);
ServletOutputStream output = response.getOutputStream();
output.write(gzipData);
output.flush();
} else {
chain.doFilter(request, response);
}
}
// 用 GZIP 压缩字节数组
private byte[] gzip(byte[] data) {
ByteArrayOutputStream byteOutput = new ByteArrayOutputStream(10240);
GZIPOutputStream output = null;
try {
output = new GZIPOutputStream(byteOutput);
output.write(data);
} catch (IOException e) {
} finally {
try {
output.close();
} catch (IOException e) {
}
}
return byteOutput.toByteArray();
}
……
}
```

该程序的主体思想，是在响应流写回之前，对响应的字节数据进行 GZIP 压缩，因为并不是所有的浏览器都支持 GZIP 解压缩，如果浏览器支持 GZIP 解压缩，会在请求报头的 Accept-Encoding 里包含 gzip。这是告诉服务器浏览器支持 GZIP 解压缩，因此如果用程序控制压缩，为了保险起见，还需要判断浏览器是否发送 accept-encoding: gzip 报头，如果包含了该报头，才执行压缩。为了验证压缩前后的情况，使用 Firebug 监控请求和响应报头。

清单 10. 压缩前请求

```
GET /testProject/gzipCategory/test.html HTTP/1.1
Accept: */*
Accept-Language: zh-cn
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
Host: localhost:9090
Connection: Keep-Alive
```

清单 11. 不压缩的响应

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
ETag: W/"5060-1242444154000"
Last-Modified: Sat, 16 May 2009 03:22:34 GMT
Content-Type: text/html
Content-Length: 5060
Date: Mon, 18 May 2009 12:29:49 GMT
```

清单 12. 压缩后的响应

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
ETag: W/"5060-1242444154000"
Last-Modified: Sat, 16 May 2009 03:22:34 GMT
Content-Encoding: gzip
Content-Type: text/html
Content-Length: 837
Date: Mon, 18 May 2009 12:27:33 GMT
```

可以看到，压缩后的数据比压缩前数据小了很多。压缩后的响应报头包含 Content-Encoding: gzip。同时 Content-Length 包含了返回数据的大小。GZIP 压缩是一个重要的功能，前面提到的是对单一服务器的压缩优化，在高并发的情况，多个 Tomcat 服务器之前，需要采用反向代理的技术，提高并发度，而目前比较火的反向代理是 Nginx（这在后续的文章会进行详细的介绍）。对 Nginx 的 HTTP 配置部分里增加如下配置。

清单 13. Nginx 的 GZIP 配置

```
gzip  on;
gzip_min_length  1000;
gzip_buffers     4 8k;
gzip_types       text/plain application/x-javascript text/css text/html application/xml;
```

由于 Nginx 具有更高的性能，利用该配置可以更好的提高性能。在高性能服务器上该配置将非常有用。

懒加载与预加载

预加载和懒加载，是一种改善用户体验的策略，它实际上并不能提高程序性能，但是却可以明显改善用户体验或减轻服务器压力。

预加载原理是在用户查看一张图片时，就将下一张图片先下载到本地，而当用户真正访问下一张图片时，由于本地缓存的原因，无需从服务器端下载，从而达到提高用户体验的目的。为了实现预加载，我们可以实现如下的一个函数。

清单 14. 预加载函数

```
function  preload(callback) {
var imageObj = new Image();
images = new Array();
images[0]="pre_image1.jpg";
images[1]=" pre_image2.jpg";
images[2]=" pre_image3.jpg";
for(var i=0; i<=2; i++) {
  imageObj.src=images[i];
 if (imageObj.complete) { // 如果图片已经存在于浏览器缓存，直接调用回调函数
     callback.call(imageObj);
  } else {
imageObj.onload = function () {// 图片下载完毕时异步调用 callback 函数
       callback.call(imageObj);// 将回调函数的 this 替换为 Image 对象
   };
}
}
}

function callback()
{
 alert(this.src + "已经加载完毕 , 可以在这里继续预加载下一组图片");
}
```

上面的代码，首先定义了 Image 对象，并且声明了需要预加载的图像数组，然后逐一的开始加载（.src=images[i]）。如果已经在缓存里，则不做其他处理；如果不在缓存，监听 onload 事件，它会在图片加载完毕时调用。

而懒加载则是在用户需要的时候再加载。当一个网页中可能同时有上百张图片，而大部分情况下，用户只看其中的一部分，如果同时显示上百张，则浪费了大量带宽资源，因此可以当用户往下拉动滚动条时，才去请求下载被查看的图像，这个原理与 word 的显示策略非常类似。

在 JavaScript 中，它的基本原理是首先要有一个容器对象，容器里面是 img 元素集合。用隐藏或替换等方法，停止 img 的加载，也就是停止它去下载图像。然后历遍 img 元素，当元素在加载范围内，再进行加载（也就是显示或插入 img 标签）。加载范围一般是容器的视框范围，即浏览者的视觉范围内。当容器滚动或大小改变时，再重新历遍元素判断。如此重复，直到所有元素都加载后就完成。当然对于开发来讲，选择已有的成熟组件，并不失为一个上策，Lazy Load Plugin for jQuery 是基于 JQuery 的懒加载组件，它有自己的 官方网站。这是一个不错的免费插件。可以帮助程序员快速的开发懒加载应用。

小结

本文总结了前端图片高性能优化的几种方式，将它们归结起来，在读者需要的时候，可以查看本文的内容，相信按照本文的方法，可以辅助读者进行前端的高性能优化。笔者将继续写后续的部分，包括数据库的优化、负载均衡、反向代理等。包括由于笔者水平有限，如有错误，请联系我批评指正。

接下来在第二部分文章中，我将介绍前端的 BigPipe 技术、Flush 机制、动静分离、HTTP 持久连接、HTTP 协议灵活应用、静态化与伪静态化、页面缓存（整体、部分）等，并且将它们应用到 Java Web 的开发中。使用这些技术可以帮助提高 Java Web 应用程序的性能。

参考资料

学习

- 查看本系列的 第 2 部分：本文将讲解前端优化里重要的 Flush 机制、动静分离、HTTP 持久连接、HTTP 协议灵活应用、CDN 等。结合这些技术或思想，相信会使 Java Web 应用程序的性能更上一层楼。
- 参考“CSS Sprites 图片切割术与图片优化”，具体了解 CSS Sprites 图片切割术与图片优化的技术。
- 了解 “GZIP 压缩详解”，了解 GZIP 压缩的详细原理。
- 查看“深入理解 HTTP 协议”，了解 HTTP 协议，更加理解 HTTP 的含义。
- 查看“CSS Sprite”，了解 CSS Sprite 的发源和原理。
- 查看“当前浏览器的并行下载数”，查看浏览器的并行下载数。
- 查看“浅谈 BASE64”，了解 BASE64 的基本原理。
- 查看“预加载与懒加载”，大致了解预加载与懒加载的原理和概念。
- developerWorks Web development 专区：通过专门关于 Web 技术的文章和教程，扩展您在网站开发方面的技能。
- developerWorks Ajax 资源中心：这是有关 Ajax 编程模型信息的一站式中心，包括很多文档、教程、论坛、blog、wiki 和新闻。任何 Ajax 的新信息都能在这里找到。
- developerWorks Web 2.0 资源中心，这是有关 Web 2.0 相关信息的一站式中心，包括大量 Web 2.0 技术文章、教程、下载和相关技术资源。您还可以通过 Web 2.0 新手入门 栏目，迅速了解 Web 2.0 的相关概念。
- 查看 HTML5 专题，了解更多和 HTML5 相关的知识和动向。

讨论

- 加入 developerWorks 中文社区。查看开发人员推动的博客、论坛、组和维基，并与其他 developerWorks 用户交流。

关于作者

魏强，东北大学软件学院硕士研究生，现在主要从事 Eclipse 插件的开发，同时热爱着 Web 技术，尤其对 Java Web 相关技术，更是情有独钟。他的邮箱是：neuswc20063500@gmail.com。