高性能web

# 前端的优化

## 背景

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。

## 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

### 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。

// 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();

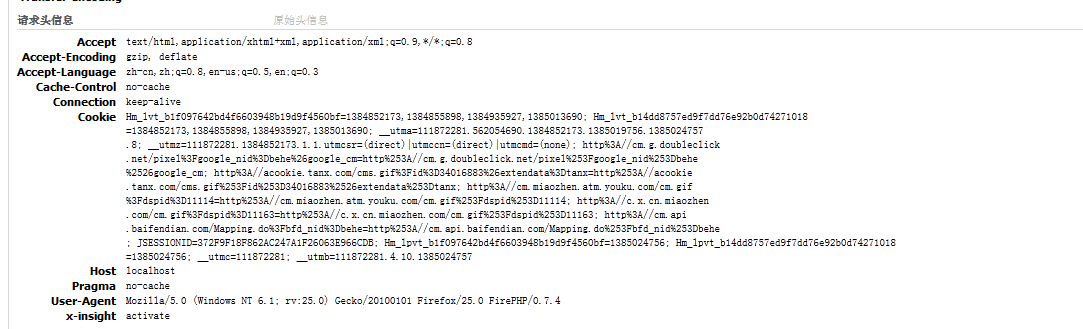
}

……

}

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。

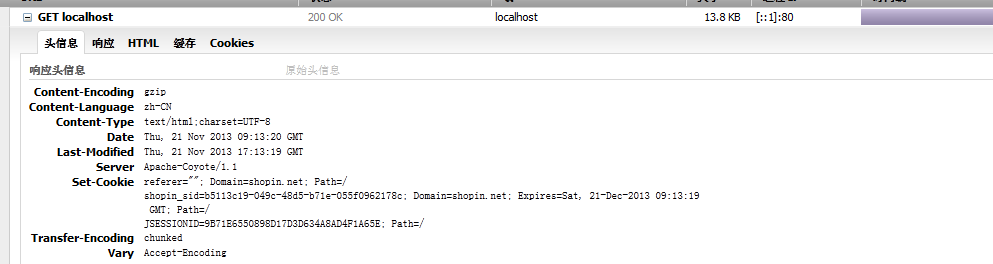
#### **请求头信息**



#### 不压缩响应



#### 压缩响应



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。

### Nginx的Gzip压缩配置

tcp\_nopush on;

keepalive\_timeout 120;

tcp\_nodelay on;

add\_header Vary Accept-Encoding;

gzip on;

gzip\_static on;

gzip\_http\_version 1.0;

gzip\_proxied any;

gzip\_min\_length 1000;

gzip\_disable "MSIE [1-6].";

gzip\_comp\_level 6;



## 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](http://www.smushit.com/ysmush.it)**。**

## 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。

### 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。

<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。

单域名多图片的效果



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。

<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>

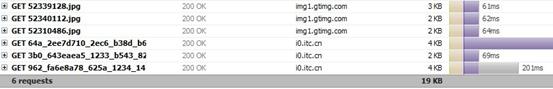
<img src="http://i0.itc.cn/20110624/3b0\_643eaea5\_1233\_b543\_82b7\_9c7273c7f97c\_1.jpg"><br>

<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。



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。

## 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。

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](http://www.appelsiini.net/projects/lazyload)。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。

## 如何加载JavaScript

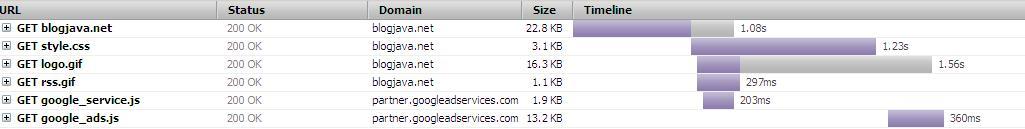
### **外部JS的阻塞下载**

所有浏览器在下载JS的时候，会阻止一切其他活动，比如其他资源的下载，内容的呈现等等。至到JS下载、解析、执行完毕后才开始继续并行下载其他资源并呈现内容。

**有人会问**：为什么JS不能像CSS、image一样并行下载了?这里需要简单介绍一下浏览器构造页面的原理，

当浏览器从服务器接收到了HTML文档，并把HTML在内存中转换成DOM树，在转换的过程中如果发现某个节点(node)上引用了CSS或者 IMAGE，就会再发1个request去请求CSS或image,然后继续执行下面的转换，而不需要等待request的返回，当request返回 后，只需要把返回的内容放入到DOM树中对应的位置就OK。但当引用了JS的时候，浏览器发送1个js request就会一直等待该request的返回。因为浏览器需要1个稳定的DOM树结构，而JS中很有可能有代码直接改变了DOM树结构，比如使用 document.write 或 appendChild,甚至是直接使用的location.href进行跳转，浏览器为了防止出现JS修改DOM树，需要重新构建DOM树的情况，所以 就会阻塞其他的下载和呈现.

阻塞下载图：下图是访问blogjava首页的时间瀑布图，可以看出来开始的2个image都是并行下载的，而后面的2个JS都是阻塞下载的(1个1个下载)。



### **嵌入JS的阻塞下载**

嵌入JS是指直接写在HTML文档中的JS代码。上面说了引用外部的JS会阻塞其后的资源下载和其后的内容呈现，哪嵌入的JS又会是怎样阻塞的了，