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();

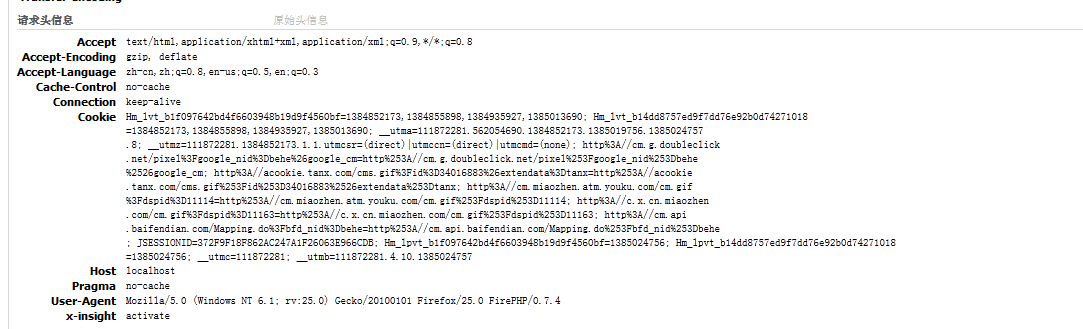
}

……

}

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。

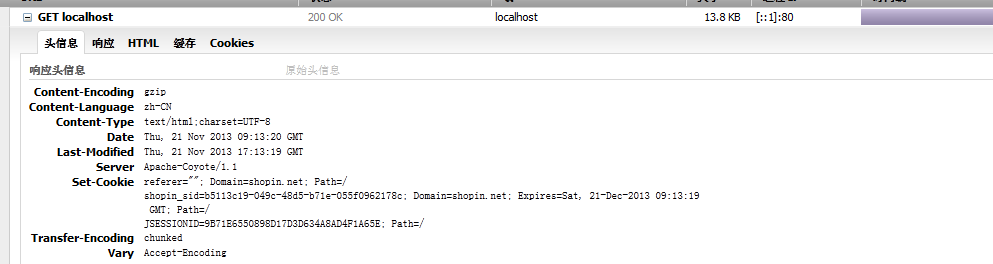
#### **请求头信息**



#### 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