http://www.ibm.com/developerworks/cn/java/j-lo-javawebhiperf2/

Java Web 高性能开发，第 2 部分：前端的高性能

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。Web 的思想是通用的，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。第一部分已经讲解了部分前端优化，该部分是前端性能优化的其他内容，包括 HTTP 协议的利用、动静分离等等。合理利用这些技术将使 Web 技术更加高效。

查看本系列更多内容 | 评论：

魏 强, 研究员, IBM
王 芹华, 研究员, IBM
2013 年 3 月 28 日

- 内容

引言

在前端优化的第一部分中，主要讲解了对静态资源的一些优化措施，包括图片压缩 、CSS Sprites 技术、GZIP 压缩等。这一部分，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。

回页首

Flush 机制的使用

实际上在 Web 技术中，Flush 机制并不新鲜，它的思想是无需等到网页内容全部加载完毕，一次性写回客户端，而是可以部分逐次的返回。如果网页很大的话，一次性写回全部内容显然是个不明智的选择，因为这会造成网页的长时间空白。Flush 机制允许开

发人员将网页的内容按文档流顺序逐步返回给客户端，这样可以使得用户知道我们的系统正在工作，只是等待的时间稍长而已，这样用户也会“心甘情愿”的等下去。Flush 机制是一个经典的提高用户体验的方法，至今也一直在用。如果网页很大，这个机制也是建议使用的。在 Java Web 技术中，实现 Flush 非常简单，只要调用 HttpServletResponse.getWriter 输出流的 flush 方法，就可以将已经完成加载的内容写回给客户端。

但是是否每个网页都要使用该技术呢？笔者当然不这么建议。将网页内容加载完毕后再一次性返回客户端也有它的好处。我们知道网络传输也有最大的传输单元，内容加载完毕后一次性输出就可以最大程度的利用传输的带宽，减少分块，减少传输次数，也就是说实际上 Flush 机制会增加用户等待时间、增加浏览器渲染时间，但是对于大网页来说，降低这点效率来增强用户体验，是值得的。

回页首

动静分离

所谓的动静分离，就是将 Web 应用程序中静态和动态的内容分别放在不同的 Web 服务器上，有针对性的处理动态和静态内容，从而达到性能的提升。本文基于 Java Web 来讲解 Web 优化，而 Java Web 的主流服务器软件是 Tomcat。让人遗憾的是，Tomcat 在并发和静态资源处理的能力上较弱，这也是 Tomcat 为人诟病的地方。但是瑕不掩瑜，既然我们选择了 Java Web，那么就应该发挥我们程序员的头脑去想方设法的提高性能。而动静分离就是其中一种方法，既然 Tomcat 处理静态资源的能力较弱，那就将静态资源的处理任务交给适合的软件，而让 Tomcat 专注于处理 JSP/Servlet 的请求。

对于静态资源处理的服务器软件，我们可以选择 Nginx，它是一款俄罗斯人开发的软件，似乎比 Apache 更加优秀。它支持高并发，对静态资源处理的能力较强，这正是我们想要的不是吗？事实上，动静分离的方案很多，有人采用 Apache+Tomcat 的组合；也有人使用 Tomcat+Tomcat 的组合，不过两个 Tomcat 分别被放置于不同的主机，不同的域名。其中 Apache+Tomcat 的方案与 Nginx 的方案原理上是一样的，它们都是基于反向代理，相对于使用 Nginx 配置动静分离，Apache 的配置就显得略微复杂一些。在 Apache 里，mod_proxy 模块负责反向代理的实现。其中核心配置内容如清单 1 所示，该配置属于本人参与某项目的其中一部分。

清单 1. 动静分离的 Apache 核心配置

```
<Proxy balancer://proxy>
      BalancerMember http://192.168.1.178:8080 loadfactor=1
      BalancerMember http://192.168.1.145:8080 loadfactor=1
</Proxy>
NameVirtualHost *:80
<VirtualHost *:80>
      ServerAdmin service@xuanli365.com
      ServerName www.xuanli365.com
      DocumentRoot /www
      DirectoryIndex index.shtml
      <Directory /www>
```

```
            AllowOverride All
            AddType text/html .shtml
            AddType application/x-rar .rar
            AddHandler server-parsed .shtml
            Options +IncludesNOEXEC
      </Directory>
RewriteEngine on
ProxyRequests Off
          ProxyPass /static/!
          ProxyPass / balancer://proxy/
          ProxyPassReverse / balancer://proxy/
          ProxyPreserveHost on
</VirtualHost>
```

从 Apache 官方对 mod_proxy 模块的介绍，我们可以知道 ProxyPass 属性可以将一个远端服务器映射到本地服务器的 URL 空间中，也就是说这是一个地址映射功能。在清单 1 的配置中，当访问的路径不在 /static/ 下时（！表示非），就转发给后端的服务器（也就是 Tomcat）；否则如果是 /static/ 路径就访问本机。例如，当访问 www.xuanli365.com/static/css/index.css 时，实际处理请求的是 Apache 服务器，而访问 www.xuanli365.com/index.jsp，那么 Apache 会将请求转发到后端的 Tomcat 服务器，实际访问的页面是 http:// 192.168.1.178(或 145):8080/index.jsp，这就实现了动静分离。在清单 1 的配置中实际也包含了简单的负载均衡（loadfactor 因子）。

事实上，我们可以随便打开一个大型门户网站来看一下，我打开的是腾讯网站，任意查看其中两张图片的地址，我发现一个是：http://mat1.gtimg.com/www/iskin960/qqcomlogo.png，而另一个则是：http://img1.gtimg.com/v/pics/hv1/95/225/832/54158270.jpg。可见该网站存放图片资源使用了多个的域名，我们再用 Linux 的 host 命令查看两个域名的 IP 地址，结果如图 1 所示。

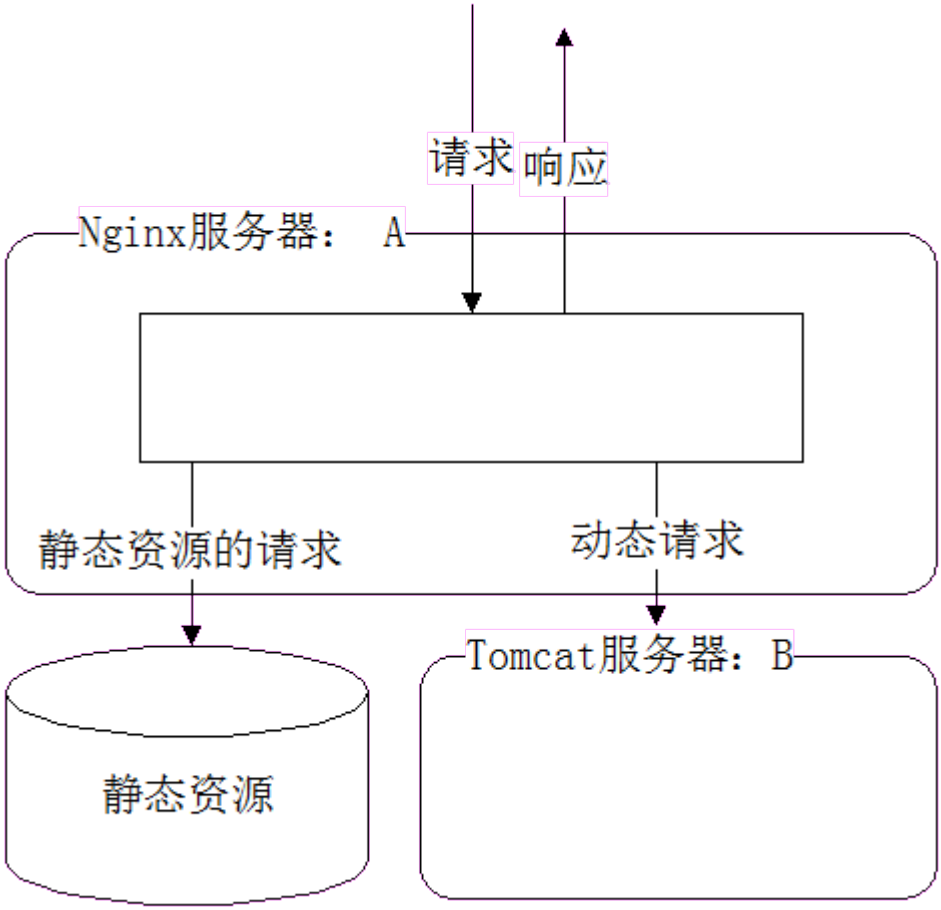
图 1. 某网站的动静分离

```
wq243221863@ubuntu:~$ host img1.gtimg.com
img1.gtimg.com has address 59.64.114.101
img1.gtimg.com has address 59.64.114.102
Host img1.gtimg.com not found: 3(NXDOMAIN)
Host img1.gtimg.com not found: 3(NXDOMAIN)
wq243221863@ubuntu:~$ host mat1.gtimg.com
mat1.gtimg.com has address 59.64.114.99
Host mat1.gtimg.com not found: 3(NXDOMAIN)
Host mat1.gtimg.com not found: 3(NXDOMAIN)
```

可以看到，通过查看 IP 地址，我们发现这些图片很可能存放在不同的主机上（为什么是很可能？因为一个主机可以拥有多个 IP），而图片内容和网页的动态内容并不在同一 IP 下，也很可能是动静分离。多个域名在前面也已经提到，可以增加浏览器的并发下载数，提高下载效率。

本文采用另一种策略对动静分离进行演示，它的大致结构如图 2 所示。

图 2. 本文设计的动静分离结构



在本文中，我们将静态资源放在 A 主机的一个目录上，将动态程序放在 B 主机上，同时在 A 上安装 Nginx 并且在 B 上安装 Tomcat。配置 Nginx，当请求的是 html、jpg 等静态资源时，就访问 A 主机上的静态资源目录；当用户提出动态资源的请求时，则将请求转发到后端的 B 服务器上，交由 Tomcat 处理，再由 Nginx 将结果返回给请求端。

提到这，可能有您会有疑问，动态请求要先访问 A，A 转发访问 B，再由 B 返回结果给 A，A 最后又将结果返回给客户端，这是不是有点多余。初看的确多余，但是这样做至少有 2 点好处。第一，为负载均衡做准备，因为随着系统的发展壮大，只用一台 B 来处理动态请求显然是是不够的，要有 B1，B2 等等才行。那么基于图 2 的结构，就可以直接扩展 B1，B2，再修改 Nginx 的配置就可以实现 B1 和 B2 的负载均衡。第二，对于程序开发而言，这种结构的程序撰写和单台主机没有区别。我们假设只用一台 Tomcat 作为服务器，那么凡是静态资源，如图片、CSS 代码，就需要编写类似这样的访问代码：<img src="{address of A}/a.jpg">，当静态资源过多，需要扩展出其他的服务器来安放静态资源时，访问这些资源就可能要编写这样的代码：<img src="{address of C}/a.jpg">、<img src="{address of D}/a.jpg">。可以看到，当服务器进行变更或扩展时，代码也要随之做出修改，对于程序开发和维护来说非常困难。而基于上面的结构，程序都只要 <img src="a.jpg">，无需关心具体放置资源的服务器地址，因为具体的地址 Nginx 为帮您绑定和选择。

按照图 2 所示的架构图，安装好需要的软件 Nginx 和 Tomcat。按照设想，对 Nginx 的配置文件 nginx.conf 进行配置，其中与本文该部分相关的配置如清单 2 所示。

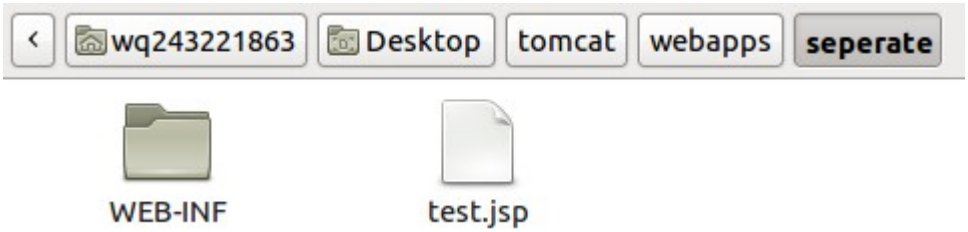
清单 2. 动静分离的 Nginx 配置

```
# 转发的服务器，upstream 为负载均衡做准备
upstream tomcat_server{
      server 192.168.1.117:8080;
}

server {
      listen       9090;
      server_name  localhost;
index index.html index.htm index.jsp;
      charset koi8-r;

      # 静态资源存放目录
      root  /home/wq243221863/Desktop/ROOT;

      access_log  logs/host.access.log  main;

# 动态请求的转发
      location ~ .*.jsp$ {
         proxy_pass http://tomcat_server;
         proxy_set_header Host $host;
      }

# 静态请求直接读取
location ~ .*\.(gif|jpg|jpeg|png|bmp|swf|css)$ {
        expires      30d;
}
……
```

清单 2 十分简洁，其目的和我们预期的一样，动态的请求（以 .jsp 结尾）发到 B（192.168.1.117：8080，即 tomcat_server）上，而静态的请求（gif|jpg 等）则直接访问定义的 root（/home/wq243221863/Desktop/ROOT）目录。这个 root 目录我直接将其放到 Linux 的桌面 ROOT 文件夹。

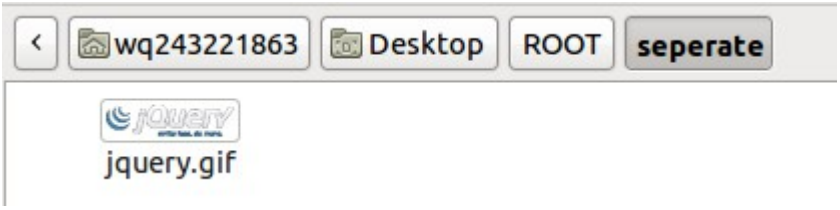
接下来在 Tomcat 中新建 Web 项目，很简单，我们只为其添加一个 test.jsp 文件，目录结构如图 3 所示。

图 3. B 上的测试项目结构



而我们定义了一张测试用的静态图片，放置在 A 的桌面 ROOT/seperate 目录下。结构如图 4 所示

图 4. A 上的静态资源文件夹结构



注意：这里的 separate 目录名是与 B 的项目文件夹同名的。

再查看图 3 中的 test.jsp 的源码。如清单 3 所示。

清单 3. test.jsp 源码

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<%@ page import="java.util.Date" %>
<%@ page import="java.text.SimpleDateFormat" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/
html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>动静分离的测试</title>
</head>
<body>
     <div>这是动态脚本处理的结果</div><br>
   <% //这是一段测试的动态脚本
   Date now=new Date();
   SimpleDateFormat f=new SimpleDateFormat("现在是"+"yyyy年MM月dd日E kk点mm分");
   %>
   <%=f.format(now)%>
   <br><br>
   <div>这是静态资源的请求结果</div><br><img alt="静态资源" src="jquery.gif">
</body>
</html>
```

清单 3 是一个非常简单的 JSP 页面，主要是使用 img 标签来访问 jquery.gif，我们知道 test.jsp 在 B 服务器上，而 jquery.gif 在 A 服务器上。用于访问 jquery.gif 的代码里不需要指定 A 的地址，而是直接使用相对路径即可，就好像该图片也在 B 上一样，这就是本结构的一个优点了。我们在 A 上访问 test.jsp 文件。结果如图 5 所示。

图 5. test.jsp 的结果

非常顺利，完全按照我们的想法实现了动静分离！
我们初步完成了动静分离的配置，但是究竟动静分离如何提高我们的程序性能我们还不得而知，我们将 Tomcat 服务器也迁移到 A 服务器上，同时将 jquery.gif 拷贝一份到 separate 项目目录下，图 3 的结构变为图 6 所示。

图 6. 拷贝 jquery.gif 的 separate 项目

我们将 Tomcat 的端口设置为 8080，Nginx 的端口依然是 9090。现在访问 http://localhost:9090/separate/test.jsp（未使用动静分离）和访问 http://localhost:8080/separate/test.jsp（使用了动静分离）的效果是一样的了。只是 8080 端口的静态资源由 Tomcat 处理，而 9090 则是由 Nginx 处理。我们使用 Apache 的 AB 压力测试工具，对 http://localhost:8080/seperate/jquery.gif、http://localhost:9090/seperate/jquery.gif、http://localhost:8080/seperate/test.jsp、http://localhost:9090/seperate/test.jsp 分别进行压力和吞吐率测试。
首先，对静态资源（jquery.gif）的处理结果如清单 4 所示。

清单 4. 静态资源的 AB 测试

测试脚本：***ab -c 100 -n 1000 http://localhost:{port}/seperate/jquery.gif***

9090 端口，也就是 ***Nginx*** 的测试结果：

```
Concurrency Level:      100
Time taken for tests:   0.441 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
```

```
Total transferred:      4497000 bytes
HTML transferred:       4213000 bytes
Requests per second:    2267.92 [#/sec] (mean)
Time per request:       44.093 [ms] (mean)
Time per request:       0.441 [ms] (mean, across all concurrent requests)
Transfer rate:          9959.82 [Kbytes/sec] received

8080 端口，也就是 Tomcat 的测试结果：
Concurrency Level:      100
Time taken for tests:   1.869 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      4460000 bytes
HTML transferred:       4213000 bytes
Requests per second:    535.12 [#/sec] (mean)
Time per request:       186.875 [ms] (mean)
Time per request:       1.869 [ms] (mean, across all concurrent requests)
Transfer rate:          2330.69 [Kbytes/sec] received
```

清单 4 的测试脚本代表同时处理 100 个请求并下载 1000 次 jquery.gif 文件，您可以只关注清单 4 的粗体部分（Requests per second 代表吞吐率），从内容上就可以看出 Nginx 实现动静分离的优势了，动静分离每秒可以处理 2267 个请求，而不使用则只可以处理 535 个请求，由此可见动静分离后效率的提升是显著的。

您还会关心，动态请求的转发，会导致动态脚本的处理效率降低吗？降低的话又降低多少呢？因此我再用 AB 工具对 test.jsp 进行测试，结果如清单 5 所示。

清单 5. 动态脚本的 AB 测试

测试脚本：`ab -c 1000 -n 1000 http://localhost:{port}/seperate/test.jsp`

```
9090 端口，也就是 Nginx 的测试结果：
Concurrency Level:      100
Time taken for tests:   0.420 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      709000 bytes
HTML transferred:       469000 bytes
Requests per second:    2380.97 [#/sec] (mean)
Time per request:       42.000 [ms] (mean)
Time per request:       0.420 [ms] (mean, across all concurrent requests)
Transfer rate:          1648.54 [Kbytes/sec] received

8080 端口，也就是 Tomcat 的测试结果：
Concurrency Level:      100
Time taken for tests:   0.376 seconds
```

```
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      714000 bytes
HTML transferred:       469000 bytes
Requests per second:    2660.06 [#/sec] (mean)
Time per request:       37.593 [ms] (mean)
Time per request:       0.376 [ms] (mean, across all concurrent requests)
Transfer rate:          1854.77 [Kbytes/sec] received
```

经过笔者的多次测试，得出了清单 5 的较为稳定的测试结果，可以看到在使用 Nginx 实现动静分离以后，的确会造成吞吐率的下降，然而对于网站整体性能来说，静态资源的高吞吐率，以及未来可以实现的负载均衡、可扩展、高可用性等，该牺牲我想也应该是值得的。

我想任何技术都是有利有弊，动静分离也是一样，选择了动静分离，就选择了更为复杂的系统架构，维护起来在一定程度会更为复杂和困难，但是动静分离也的确带来了很大程度的性能提升，这也是很多系统架构师会选择的一种解决方案。

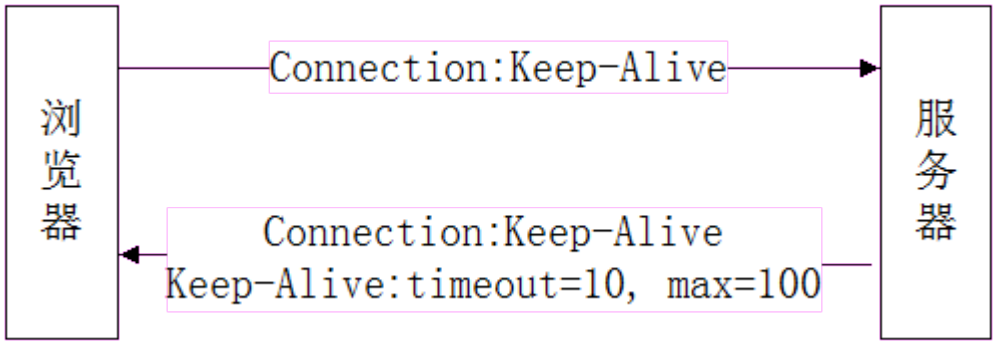
回页首

HTTP 持久连接

持久连接（Keep-Alive）也叫做长连接，它是一种 TCP 的连接方式，连接会被浏览器和服务器所缓存，在下次连接同一服务器时，缓存的连接被重新使用。由于 HTTP 的无状态性，人们也一直很清楚“一次性”的 HTTP 通信。持久连接则减少了创建连接的开销，提高了性能。HTTP/1.1 已经支持长连接，大部分浏览器和服务器也提供了长连接的支持。

可以想象，要想发起长连接，服务器和浏览器必须共同合作才可以。一方面浏览器要保持连接，另一方面服务器也不会断开连接。也就是说要想建立长连接，服务器和浏览器需要进行协商，而如何协商就要靠伟大的 HTTP 协议了。它们协商的结构图如图 7 所示。

图 7. 长连接协商



浏览器在请求的头部添加 Connection:Keep-Alive，以此告诉服务器“我支持长连接，你支持的话就和我建立长连接吧”，而倘若服务器的确支持长连接，那么就在响应头部添加“Connection:Keep-Alive”，从而告诉浏览器“我的确也支持，那我们建立长连接

吧”。服务器还可以通过 Keep-Alive:timeout=10, max=100 的头部告诉浏览器“我希望 10 秒算超时时间，最长不能超过 100 秒”。
在 Tomcat 里是允许配置长连接的，配置 conf/server.xml 文件，配置 Connector 节点，该节点负责控制浏览器与 Tomcat 的连接，其中与长连接直接相关的有两个属性，它们分别是：keepAliveTimeout，它表示在 Connector 关闭连接前，Connector 为另外一个请求 Keep Alive 所等待的微妙数，默认值和 connectionTimeout 一样；另一个是 maxKeepAliveRequests，它表示 HTTP/1.0 Keep Alive 和 HTTP/1.1 Keep Alive / Pipeline 的最大请求数目，如果设置为 1，将会禁用掉 Keep Alive 和 Pipeline，如果设置为小于 0 的数，Keep Alive 的最大请求数将没有限制。也就是说在 Tomcat 里，默认长连接是打开的，当我们想关闭长连接时，只要将 maxKeepAliveRequests 设置为 1 就可以。
毫不犹豫，首先将 maxKeepAliveRequests 设置为 20，keepAliveTimeout 为 10000，通过 Firefox 查看请求头部（这里我们访问上面提到的 test.jsp）。结果如图 8 所示。

图 8. 服务器打开长连接

接下来，我们将 maxKeepAliveRequests 设置为 1，并且重启服务器，再次请求网页后查看的结果如图 9 所示。

图 9. 服务器关闭长连接

对比可以发现，Tomcat 关闭长连接后，在服务器的请求响应中，明确标识了：Connection close, 它告诉浏览器服务器并不支持长连接。那么长连接究竟可以带来怎么样的性能提升，我们用数据说话。我们依然使用 AB 工具，它可以使用一个 -k 的参数，

模拟浏览器使用 HTTP 的 Keep-Alive 特性。我们对 http://localhost:8080/seperate/jquery.gif 进行测试。测试结果如清单 6 所示。

清单 6. AB 测试长连接

测试脚本：***ab - k -c 1000 -n 10000 http://localhost:8080/seperate/jquery.gif***

关闭长连接时：

```
Concurrency Level:      1000
Time taken for tests:   5.067 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Keep-Alive requests:    0
Total transferred:      44600000 bytes
HTML transferred:       42130000 bytes
Requests per second:    1973.64 [#/sec] (mean)
Time per request:       506.678 [ms] (mean)
Time per request:       0.507 [ms] (mean, across all concurrent requests)
Transfer rate:          8596.13 [Kbytes/sec] received
```

打开长连接时，***maxKeepAliveRequests*** 设置为 ***50***：

```
Concurrency Level:      1000
Time taken for tests:   1.671 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Keep-Alive requests:    10000
Total transferred:      44650000 bytes
HTML transferred:       42130000 bytes
Requests per second:    5983.77 [#/sec] (mean)
Time per request:       167.119 [ms] (mean)
Time per request:       0.167 [ms] (mean, across all concurrent requests)
Transfer rate:          26091.33 [Kbytes/sec] received
```

结果一定会让您大为惊讶，使用长连接和不使用长连接的性能对比，对于 Tomcat 配置的 maxKeepAliveRequests 为 50 来说，竟然提升了将近 5 倍。可见服务器默认打开长连接是有原因的。

回页首

HTTP 协议的合理使用

很多程序员都将精力专注在了技术实现上，他们认为性能的高低完全取决于代码的实现，却忽略了已经成型的某些规范、协议、工具。最典型的就是在 Web 开发上，部分开

发人员没有意识到 HTTP 协议的重要性，以及 HTTP 协议可以提供程序员另一条性能优化之路。通过简单的在 JSP 的 request 对象中添加响应头部，往往可以迅速提升程序性能，一切实现代码仿佛都成浮云。本系列文章的宗旨也在于让程序员编最少的代码，提升最大的性能。

本文提出一个这样的需求，在文章前面部分提到的 test.jsp 中，它的一部分功能是显示服务器的当前时间。现在我们希望这个动态网页允许被浏览器缓存，这似乎有点不合理，但是在很多时候，虽然是动态网页，但是却只执行一次（比如有些人喜欢将网页的主菜单存入数据库，那么他肯定不希望每次加载菜单都去读数据库）。浏览器缓存带来的性能提升已经众人皆知了，而很多人却并不知道浏览器的缓存过期时间、缓存删除、什么页面可以缓存等，都可以由我们程序员来控制，只要您熟悉 HTTP 协议，就可以轻松的控制浏览器。

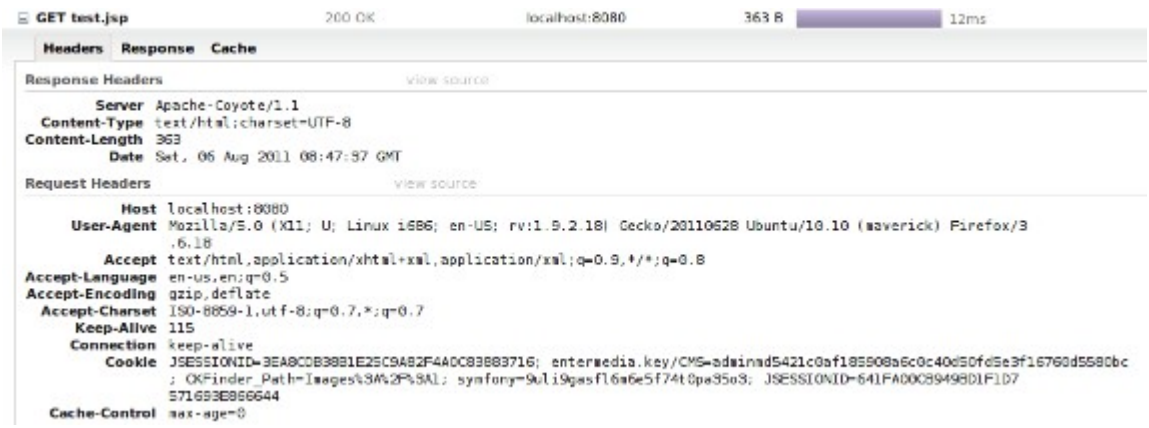
我们访问上面提及的 test.jsp。用 Firebug 查看请求情况，发现每次请求都会重新到服务器下载内容，这不难理解，因此 test.jsp 是动态内容，每次服务器必须都执行后才可以返回结果 ，图 10 是访问当前的 test.jsp 的头部情况。现在我们往 test.jsp 添加清单 7 的内容。

清单 7. 在 test.jsp 的首部添加的代码

```
<%
SimpleDateFormat f2=new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss");
String ims = request.getHeader("If-Modified-Since");
if (ims != null)
{
try
{
Date dt = f2.parse(ims.substring(0, ims.length()-4));
if (dt.after(new Date(2009, 1, 1)))
{
response.setStatus(304);
return;
}
} catch(Exception e)
{

}
}
response.setHeader("Last-Modified", f2.format(new Date(2010, 5, 5)) + " GMT");
%>
```

上述代码的意图是：服务器获得浏览器请求头部中的 If-Modified-Since 时间，这个时间是浏览器询问服务器，它所请求的资源是否过期，如果没过期就返回 304 状态码，告诉浏览器直接使用本地的缓存就可以，

图 10. 修改 test.jsp 前的访问头部情况



修改完 test.jsp 代码后，使用鼠标激活浏览器地址栏，按下回车刷新页面。这次的结果如图 11 所示。

图 11. 修改 test.jsp 后的首次访问

可以看到图 11 和图 10 的请求报头没有区别，而在服务器的响应中，图 11 增加了 Last-Modified 头部，这个头部告诉浏览器可以将此页面缓存。

按下 F5（必须是 F5 刷新），F5 会强制 Firefox 加载服务器内容，并且发出 If-Modified-Since 头部。得到的报头结果如图 12 所示 .

图 12. 修改 test.jsp 后的再次访问

可以看到，图 12 的底部已经提示所有内容都来自缓存。浏览器的请求头部多出了 If-Modified-Since，以此询问服务器从缓存时间起，服务器是否对资源进行了修改。服务器判断后发现没有对此资源（test.jsp）修改，就返回 304 状态码，告诉浏览器可以使用缓存。

我们在上面的实验中，用到了 HTTP 协议的相关知识，其中涉及了 If-Modified-Since、Last-Modified、304 状态码等，事实上与缓存相关的 HTTP 头部还有许多，诸如过期设置的头部等。熟悉了 HTTP 头部，就如同学会了如何与用户的浏览器交谈，也可以利用协议提升您的程序性能。这也是本文为何一直强调 HTTP 协议的重要性。那么对于 test.jsp 这个小网页来说，基于缓存的方案提升了多少性能呢？我们用 AB 给您答案。

AB 是个很强大的工具，他提供了 -H 参数，允许测试人员手动添加 HTTP 请求头部，因此测试结果如清单 8 所示。

清单 8. AB 测试 HTTP 缓存

测试脚本：***ab –c 1000 – n 10000 – H ' If-Modified-Since:Sun, 05 Jun 3910 00:00:00 GMT ' http://localhost:8080/seperate/test.jsp***

未修改 *test.jsp* 前 :

```
Document Path:          /seperate/test.jsp
Document Length:        362 bytes
Concurrency Level:      1000
Time taken for tests:   10.467 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      6080000 bytes
HTML transferred:       3630000 bytes
Requests per second:    955.42 [#/sec] (mean)
Time per request:       1046.665 [ms] (mean)
Time per request:       1.047 [ms] (mean, across all concurrent requests)
Transfer rate:          567.28 [Kbytes/sec] received
```

修改 *test.jsp* 后：

```
Document Path:          /seperate/test.jsp
Document Length:        0 bytes
Concurrency Level:      1000
Time taken for tests:   3.535 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Non-2xx responses:      10000
Total transferred:      1950000 bytes
HTML transferred:       0 bytes
Requests per second:    2829.20 [#/sec] (mean)
Time per request:       353.457 [ms] (mean)
Time per request:       0.353 [ms] (mean, across all concurrent requests)
Transfer rate:          538.76 [Kbytes/sec] received
```

分别对比 Document Length、Requests per second 以及 Transfer rate 这三个指标。可以发现没使用缓存的 Document Length（下载内容的长度）是 362 字节，而使用了缓存的长度为 0。在吞吐率方面，使用缓存是不使用缓存的 3 倍左右。同时在传输率方面，缓存的传输率比没缓存的小。这些都是用到了客户端缓存的缘故。

回页首

CDN 的使用

CDN 也是笔者最近才了解和接触到的东西，耳中也是多次听到 CDN 这个词了，在淘宝的前端技术报告上、在一个好朋友的创新工场创业之路上，我都听到了这个词，因此我想至少有必要对此技术了解一下。所谓的 CDN，就是一种内容分发网络，它采用智能路由和流量管理技术，及时发现能够给访问者提供最快响应的加速节点，并将访问者的请求导向到该加速节点，由该加速节点提供内容服务。利用内容分发与复制机制，CDN 客户不需要改动原来的网站结构，只需修改少量的 DNS 配置，就可以加速网络的响应速度。当用户访问了使用 CDN 服务的网站时，DNS 域名服务器通过 CNAME 方式将最终域名请求重定向到 CDN 系统中的智能 DNS 负载均衡系统。智能 DNS 负载均衡系统通过一组预先定义好的策略（如内容类型、地理区域、网络负载状况等），将当时能够最快响应用户的节点地址提供给用户，使用户可以得到快速的服务。同时，它还与分布在不同地点的所有 CDN 节点保持通信，搜集各节点的健康状态，确保不将用户的请求分配到任何一个已经不可用的节点上。而我们的 CDN 还具有在网络拥塞和失效情况下，能拥有自适应调整路由的能力。

由于笔者对 CDN 没有亲身实践，不便多加讲解，但是各大网站都在一定程度使用到了 CDN，淘宝的前端技术演讲中就提及了 CDN，可见 CDN 的威力不一般。

图 12. 淘宝的 CDN 前端优化

因此 CDN 也是不得不提的一项技术，国内有免费提供 CDN 服务的网站：http://www.webluker.com/，它需要您有备案的域名，感兴趣的您可以去试试。

回页首

小结

本文总结了 HTTP 长连接、动静分离、HTTP 协议等等，在您需要的时候，可以查看本文的内容，相信按照本文的方法，可以辅助您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。由于笔者水平有限，如有错误，请联系我批评指正。

接下来在第三部分文章中，我将介绍服务器端缓存、静态化与伪静态化、分布式缓存等，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。