

Auslieferungsmethoden im Performancevergleich Zieleinlauf mit Java

Dominik Gruntz & Hans-Peter Oser
Fachhochschule Aargau/Nordwestschweiz
d.gruntz@fh-aargau.ch & h.oser@fh-aargau.ch

Im iX 3/2005 wurde mit einem kleinen Test [1] der Frage nachgegangen, ob und unter welchen Umständen Perl, PHP oder C schneller arbeiten. Was uns und anderen Lesern bei diesem Vergleich fehlte, waren Varianten mit Java. Dieser Beitrag schliesst diese Lücke und erweitert die Testreihe mit Servlets und JSP Lösungen.

In einem Projekt standen wir vor der Frage, ob wir eine Lösung auf der Basis von PHP oder Java realisieren sollen. Der Beitrag [1] im iX kam uns dabei gerade recht, nur fehlten die Zahlen für Java. An Java klebt der Makel, dass es auf Grund des Garbage-Collectors langsam sei. Dieser Test soll zeigen, ob dieses Vorurteil immer noch seine Berechtigung hat.

Der Test wird ebenfalls mit dem einfachen Beispiel aus [1] durchgeführt, in welchem ein kurzes 100-zeiliges (3K) bzw. ein 1000-zeiliges (30K) HTML Dokument generiert und ausgeliefert wird. Wir möchten die Diskussion, ob solche Tests sinnvoll sind oder nicht, in diesem Artikel nicht aufgreifen.

Die Messung wurde wiederum mit *ab2* (ApacheBench Version 2) durchgeführt, wobei wir das Testprogramm ebenfalls mit den Optionen *-c 10 -n 1000* gestartet haben. Die Tests (ApacheBench) liefen auch auf demselben Rechner wie der Webserver.

Als Testrechner haben wir einen Doppelprozessor XEON 2.4GHz mit 2GB RAM verwendet. Als Betriebssystem kam SuSE Linux (Kernel 2.6.8-24.11-smp) zum Einsatz. Als Webserver wurde der Apache (Version 2.0.50), als Servlet Container Tomcat (Version 5.0.27) eingesetzt. Wir haben den Servlet Zugriff direkt auf den Tomcat, aber auch über Apache (*mod_jk*) durchgeführt. Dabei hat sich gezeigt, dass der Zugriff über Apache eine Verschlechterung um durchschnittlich 30% bringt (Die Anfragen, die mehr Daten zurückgeben 27%, die kurzen Anfragen 35%). Die Zahlen der Messungen unserer Java Beispiele gelten daher immer für den Direktzugriff auf Tomcat.

Wir haben unsere Tests mit Java 5 (jdk-1.5.0_01) durchgeführt. Als Garbage Collector haben wir den Default GC verwendet. Damit die Resultate nicht verfälscht wurden, wurde nach jeder Testsequenz der Speicher mit einem expliziten GC Aufruf aufgeräumt.

Java Servlets

Als erste Variante haben wir ein einfaches Servlet geschrieben, welches GET Anfragen in der Methode *doGet* beantwortet (siehe Listing 1). Die generierte Seite enthält dieselben META Anweisungen wie die Beispiele in [1]. Die Zeilenschaltungen wurden direkt in den Strings encodiert, da diese Variante effizienter ist als die Verwendung der Methode *println*. Die Variable *max* gibt die Grösse des Dokumentes (10 oder 100) an. Der Wert dieser Variablen wird in der *init* Methode des Servlets gesetzt und kann im *web.xml* Deskriptor definiert werden.

Dieses Servlet liefert die kleinen Dokumente (100 Zeilen) im Durchschnitt in 4ms pro Anfrage aus, die grossen Dokumente (1000 Zeilen) in 10.4ms. Davon sind etwa 3.2ms konstanter Aufwand und der restliche Aufwand hängt linear von der Grösse der generierten Seite ab.



```

public void doGet (HttpServletRequest request, HttpServletResponse response) throws IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.print("<html><head>\n");
    out.print("<meta http-equiv=\"pragma\" content=\"no-cache\">\n");
    out.print("<meta http-equiv=\"expires\" content=\"Mon, 30 Jan 1970 01:01:01 GMT\">\n");
    out.print("<title>Test Servlet Pure</title>\n");
    out.print("</head><body>\n");
    for(int j = 0; j < max; j++){
        for(int i = 0; i < 10; i++){
            out.print("Dies ist Zeile " + j + "/" + i + "<br>\n");
        }
    }
    out.print("</body></html>\n");
    out.close();
}

```

Listing 1: Servlet

Dieses einfache Servlet enthält noch Optimierungspotential. Der geübte Java-Programmierer weiss, dass man Strings nicht mit dem +-Operator verknüpfen darf. Es ist besser, die Strings mit einem *StringBuffer* oder seit Java 5 mit einem *StringBuilder* aufzubauen. Letzterer ist effizienter, da die Zugriffsmethoden nicht synchronisiert sind. In Listing 2 ist der Code für eine optimierte Servlet-Variante angegeben, bei dem die gesamte Resultatseite zuerst in einen *StringBuilder* geschrieben wird (dessen Initialgrösse so festgelegt ist, dass die gesamte Seite platz hat). Ein weiterer Vorteil dieser Lösung ist, dass nur einmal auf den Outputstream geschrieben wird. Man ist so nicht davon abhängig, ob der Stream gepuffert ist oder nicht.

```

public void doGet (HttpServletRequest request, HttpServletResponse response) throws IOException
{
    response.setContentType("text/html");

    StringBuilder buf = new StringBuilder(240*max+100);
    buf.append("<html><head>\n");
    buf.append("<meta http-equiv=\"pragma\" content=\"no-cache\">\n");
    buf.append("<meta http-equiv=\"expires\" content=\"Mon, 30 Jan 1970 01:01:01 GMT\">\n");
    buf.append("<title>Test Servlet Pure</title>\n");
    buf.append("</head><body>\n");
    for(int j = 0; j < max; j++){
        for(int i = 0; i < 10; i++){
            buf.append("Dies ist Zeile ");
            buf.append(j);
            buf.append("/");
            buf.append(i);
            buf.append("<br>\n");
        }
    }
    buf.append("</body></html>");

    PrintWriter out = response.getWriter();
    out.println(buf);
    out.close();
}

```

Listing 2: Optimierte Servlet Implementierung

Mit dieser Variante erreicht man eine Verbesserung beim von der Grösse des Dokumentes abhängigen Aufwand um einen Faktor 1.7, was sich (zusammen mit dem konstanten Aufwand von 3.2ms) beim grossen Dokument in einer Verbesserung der Ausführungsgeschwindigkeit um etwa 40% und beim kleinen Dokument um etwa 9% niederschlägt. Verglichen zum statischen Zugriff auf das Dokument erreichen wir mit diesem Servlet einen Durchsatz von 43% beim grossen Dokument und 81% beim kleinen Dokument

JSP Java Server Pages

Die Generierung von HTML Code in einem Servlet ist unhandlich und aus Sicht der Wartbarkeit unschön. Besser ist es den HTML Code und den Java Code zu trennen. Ein erster Ansatz dazu sind die

Java Server Pages (JSP). JSP Seiten sind HTML Seiten, die Java Code enthalten können. Der Webserver wird eine JSP Seite in ein Servlet übersetzen, daher erwarten wir ähnliche Resultate wie bei der Servlet-Lösung. Eine JSP Seite für unser Beispiel ist in Listing 3 abgedruckt. Der Aufwand entspricht in etwa jenem des einfachen Servlets, bei den kleinen Dokumenten etwas schlechter, bei den grossen etwas besser.

```
<%@ page contentType="text/html" %>
<html><head>
<meta http-equiv="pragma" content="no-cache">
<meta http-equiv="expires" content="Mon, 30 Jan 1970 01:01:01 GMT">
<title>Test JSP</title>
</head><body>

<%
int max = Integer.parseInt(getInitParameter("max"));
%>

<%
for(int j=0; j<max; j++){
    for(int i=0; i<10; i++){
    %>
Dies ist Zeile <%=j%>/<%=i%><br>
<%
    }
}
%>
</body></html>
```

Listing 3: JSP Lösung

Etwas ungewohnt ist die Mischung von HTML und Java Code. Damit der Designer nur noch HTML-ähnlichen Code sieht, ist die JSP Standard Tab Library (JSTL) definiert worden, welche die Grundfunktionalität, die in vielen JSP Seiten benötigt wird (wie Schleifen, Bedingungen etc.), in einfachen Tags zur Verfügung stellt. Eine Formulierung der Lösung mit Hilfe der JSTL ist in Listing 4 angegeben. Der Aufwand um das 30K grosse Dokument zu erzeugen und auszuliefern ist etwa 7.5mal grösser als jener um das 3K grosse Dokument auszuliefern, d.h. der Aufwand steigt sehr stark mit der Grösse des Dokumentes an. Im Vergleich zum statischen Zugriff auf das Dokument mit 1000 Zeilen benötigt die JSP/JSTL Lösung etwa 30mal länger.

```
<%@ page contentType="text/html" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html><head>
<meta http-equiv="pragma" content="no-cache">
<meta http-equiv="expires" content="Mon, 30 Jan 1970 01:01:01 GMT">
<title>Test JSP</title>
</head><body>
<c:forEach var="j" begin="0" end="9">
    <c:forEach var="i" begin="0" end="9">
    Dies ist Zeile <c:out value="{j}"/><c:out value="{i}"/><br>
    </c:forEach>
</c:forEach>
</body></html>
```

Listing 4: JSP Lösung mit JSTL

Velocity

Eine andere Variante um Java-Code und HTML Code zu trennen ist die Verwendung einer Template-Engine. Wir haben in vielen Projekten Erfahrungen mit Velocity [2] gesammelt. In einer Unterklasse von *VelocityServlet* (vgl. Listing 6) wird das Template geladen und im Kontext werden die Werte abgelegt, auf die im Template zugegriffen wird (in unserem Fall nur die obere Schranke *max*). Das Template (vgl. Listing 5) sieht in diesem Fall ähnlich aus wie die JSP Lösung, ausser dass für den Zugriff auf die Java-Objekte und für die Formulierung der Kontrollstrukturen die Velocity Template Sprache (VTL) verwendet werden muss. Der Aufwand für das kleine Dokument ist mit JSP

vergleichbar, die Velocity Lösung fällt jedoch für grössere Dokumente ab. Mit der Velocity hatten wir generell Probleme bezüglich Stabilität bei hoher Last, die wir nicht lösen konnten.

```
<html><head>
<meta http-equiv="pragma" content="no-cache">
<meta http-equiv="expires" content="Mon, 30 Jan 1970 01:01:01 GMT">
<title>Test Velocity</title>
</head><body>
#foreach ($j in [0..$max])
#foreach ($i in [0..9])
Dies ist Zeile $j/$i<br>
#end
#end
</body></html>
```

Listing 5: Velocity Template

```
public Template handleRequest (HttpServletRequest request,
                                HttpServletResponse response, Context context) {
    // default content type is "text/html"
    context.put("max", new Integer(max-1));

    if (template == null){
        try {
            template = getTemplate("templates/example.vm");
        }
        catch(Exception e){
            System.out.println(e);
        }
    }
    return template;
}
```

Listing 6: Velocity Servlet

FastCGI

Wir haben die Tests auch mit PHP, Perl und CGI ausgeführt und wir sind beim Fast CGI Test bei unserer Infrastruktur auf ein interessantes Resultat gestossen. Bei einem Multiprozessor System muss unbedingt darauf geachtet werden, dass mehrere Instanzen des FastCGI Programmes geladen werden. Im Falle unseres Doppelprozessor XEON Systems hat sich gezeigt, dass bei 4 gleichzeitigen Instanzen die optimale Leistung erzielt wird (das Hyperthreading hat hier etwas gebracht). Ohne diese mehrfachen Instanzen war die FastCGI Leistung um über 30% tiefer!

Zusammenfassung

Die Messresultate haben wir in Abbildung 1 zusammengefasst. Die Performance ist dabei relativ zur Leistung des direkten Zugriffes der HTML Seite auf dem Apache angegeben (auf unserer Maschine war die Zugriffszeit auf das 3K Dokument im Durchschnitt 3ms und auf das 30K Dokument 3.2ms).

Während den Tests haben wir auch die Auslastung der Maschine überwacht. Während mit den CGI Scripts das System vollständig ausgelastet war (0% idle time), liess sich unser System mit den Servlets nur zwischen 70% (Servlet optimiert) und 85% (JSP) auslasten. Auch das Anheben der Priorität des Tomcat Prozesses hat daran nichts geändert. Möglicherweise handelt es sich um ein SMP Problem der verwendeten JVM.

Der Test hat aber deutlich gezeigt, dass sich die Java Lösungen durchaus zeigen lassen und Leistungen erzielen, die grösser sind als mit FastCGI. Die Begründung dafür dürfte bei den im FastCGI notwendigen Umschalten der Adressräume einerseits und in der Hotspot Optimierung der Servlets andererseits liegen.

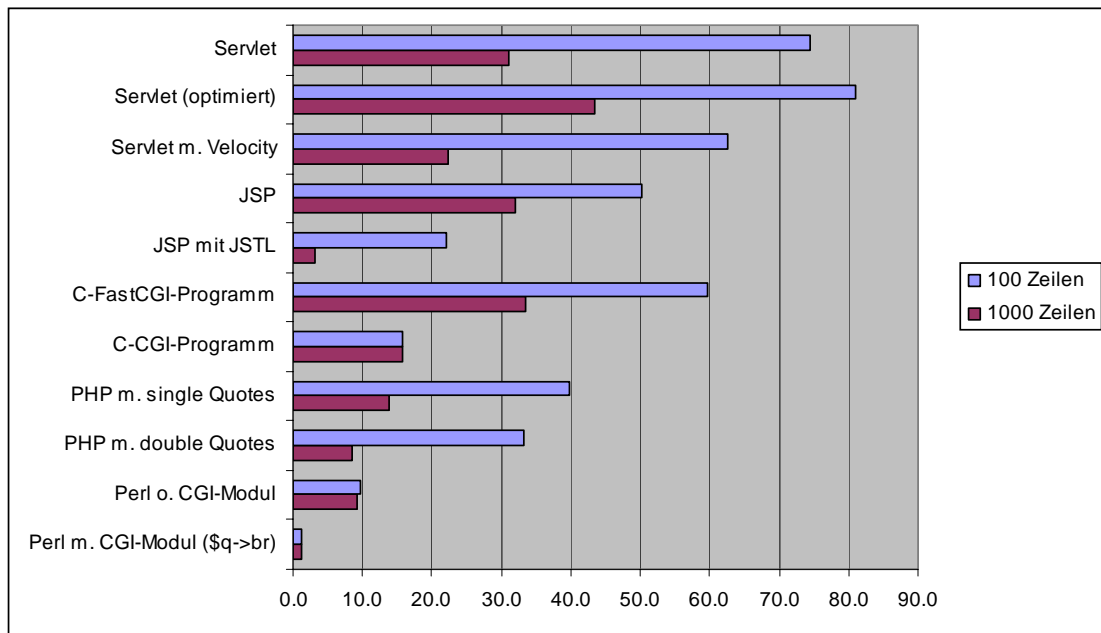


Abbildung 1: Resultate im Überblick

Referenzen:

- [1] Jürgen Seeger, Zieleinlauf: Auslieferungsmethoden im Performancevergleich, iX 3/2005, S. 94-96.
- [2] Andreas Pacek, Schraubhilfe: Apache-Projekt Velocity, iX 12/2004, S. 70-73.