

Editorial

Liebe Leserin, lieber Leser,

Hier sind wir wieder mit einer neuen Sammlung von Artikeln mit dem technischen Fokus dynamisches Laden von Klassen.

So wie sich die Tage in diesem Frühling verlängern, so wird auch unser Angebot für Ihre eingebetteten und echtzeit-Anforderungen größer. Wichtig wird dabei auch die immer weiter gehende Zusammenarbeit mit Software- und Hardware-Partnern. So bietet ORB-express von Objective Interface Systems eine hochperformante CORBA-Lösung für Ihre JamaicaVM Anwendungen.

Wir wünschen Ihnen viel Freude bei der Lektüre. Bitte zögern Sie nicht uns mitzuteilen, wenn Sie ein besonderes Thema in dieser Veröffentlichung behandelt sehen möchten.

Ihr

Dr. James J. Hunt

BMW Scientific Award für aicas' Echtzeittechnologie

BMW verleiht seinen angesehensten Preis für aicas' Echtzeit-Java Technologie.

Mehr als 200 Absolventen nahmen unter dem Motto "Passion for Innovation" am Scientific Award BMW Group 2003 teil, der mit 70.000€ höchstdotierte internationale Preis für Diplom- und Doktorarbeiten. Unter den 6 Preisträgern hat die Arbeit von Dr. Fridtjof Siebert, Director of Development der aicas GmbH, den zweiten Platz in der Kategorie Dissertation erreicht.

Mit seiner Arbeit schaffte Dr. Siebert die technologische Basis, die den Einsatz moderner Softwareentwicklungswerkzeuge basierend auf Java im immer komplexer werdenden Bereich der eingebetteten Systeme ermöglicht. Dr. Siebert entwickelte den hierfür nötigen Algorithmus für die Speicherverwaltung, den sogenannten "Garbage Collector" (Müllsammelner). Der Erfolg der

neuen Technologie zeigt sich am Erfolg von aicas' Echtzeit-Java Lösung JamaicaVM.

Im Rahmen der Preisverleihung mit Vertretern aus Wirtschaft, Wissenschaft wie Forschung sagte Prof.

Dr. Göschel, Vorstand Entwicklung bei BMW und Schirmherr des Preises: "Wer in Netzwerken denkt und handelt, treibt Innovationen schneller voran, schafft damit die Basis für neue Arbeitsplätze und sichert bestehende und damit unsere gemeinsame Zukunft. Es wird kein Wachstum und keine Beschäftigung morgen ohne Bildung und Innovation heute geben". (aw)



Scientific Award

Neuigkeiten

JamaicaVM in Japan

Januar 2004: Microsystems Co. Ltd. Japan und aicas sind eine Vertriebskooperation eingegangen. Microsystems, die im japanischen Echtzeitumfeld bekannt sind, werden künftig JamaicaVM in Japan vertreiben sowie Kundensupport anbieten. "Japan ist einer unserer strategischen Märkte", so Andy Walter, Vertriebsleiter der aicas GmbH. "Durch die Kooperation mit Microsystems können nun auch in Japan muttersprachliche Ingenieure unsere Kunden betreuen." Weitere Informationen:

Izuru Oki,
microsystems Co.,Ltd.,
www.microsystems-ltd.com

Java Hotline

Das Forschungszentrum Informatik Karlsruhe stellt Entwicklern ab sofort eine kostenlose Expertenberatung im Internet zur Verfügung. Interessenten finden die FZI Java Hotline unter www.fzi.de/javaexperte.

Neue RCE- und VRCE-Version

aicas hat eine neue Beta-Version von RCE 4.0 und VRCE 4.0 veröffentlicht. Die neue API-Version enthält das neue

streaming interface. Mit diesen Funktionen können Entwickler streams verarbeiten. Hierdurch wird einerseits die Anwendung des API weiter vereinfacht und andererseits die Leistung des API weiter erhöht. Die grafische Benutzeroberfläche von VRCE wurde ebenfalls überarbeitet und verbessert. Unter anderem ist es mit der neuen Version des Differenz-Dialoges nun möglich die Unterschiede verschiedener Versionen einer Datei einfacher und schneller mit einander zu vergleichen. RCE und VRCE 4.0 erscheint im März 2004.

Dynamisch Programmieren mit eigenen ClassLoadern

Eine der großen Stärken von Java ist es, Code dynamisch nachzuladen. Dies erfordert einen benutzerdefinierten *ClassLoader*, dessen Funktionsweise wird hier erklärt.

Das dynamische Nachladen von Code durch benutzerdefinierte *ClassLoader* in Java ermöglicht eine enorme Flexibilität bei der Softwareentwicklung. Dabei geht das dynamische Laden von Klassen über einen eigenen *ClassLoader* sehr viel weiter als das bloße Laden von Klassen über die Funktion *Class.forName()*. Insbesondere ermöglicht die Definition eines eigenen *ClassLoaders*

- das Laden von Klassen aus einer vom Benutzer frei definierten Quelle (etwa eine Netzwerkadresse, Flash-Karten, usw.)
- Die Definition eines neuen Namensraums. D.h., alle über einen *ClassLoader* geladenen Klassen können dieselben Namen verwenden, wie Klassen die über einen anderen *ClassLoader* geladen wurden.
- Das automatische Freigeben des Speichers von nicht mehr benötigten Klassen. Diese Klas-

sen werden automatisch aus dem System entfernt, sobald der *ClassLoader*, der sie geladen hat, nicht mehr benötigt wird. Dies wird als Classfile Garbage Collection bezeichnet.

Damit können Anwendungen so entwickelt werden, dass zur Laufzeit beliebiger neuer Code hinzugefügt werden kann. Zudem können voneinander unabhängige Module erstellt werden, die voneinander getrennt in der gleichen Java-Umgebung ablaufen.

Schließlich sorgt die Classfile Garbage Collection dafür, dass auch in lang laufenden Applikationen, die häufig neuen Code über eigene *ClassLoader* nachladen, der Speicher für diesen nachgeladenen Code immer wieder automatisch freigegeben wird.

Beispiel

Das Erstellen eines eigenen *ClassLoaders* ist leicht und soll hier an

einem kleinen Beispiel illustriert werden. Bild 1 und 2 zeigen die Klassen *Receiver* und *Sender* einer kleinen Netzwerkanwendung. Die *Receiver* wartet auf eine Netzwerkverbindung, während *Sender* diese Verbindung herstellt.

Über die statischen Felder *host* und *port* wird der Name und Port des Zielrechners, auf dem *Receiver* läuft, festgelegt. Hier wird Port 1024 verwendet, und als Zielrechner ist schlicht "localhost" eingetragen, so dass *Sender* und *Receiver* für Testzwecke auf demselben System laufen können. Andere Werte für *host* und *port* können über die Properties *HOST* und *PORT* angegeben werden.

Sender und *Receiver* kennen zwei Kommandos: *STOP* zum Beenden der Anwendung, und *UPLOAD*, um neue Klassen von *Sender* zum *Receiver* zu schicken, diese über einen benutzerdefinierten *ClassLoader* zu laden und auszuführen.

Als Beispiel soll die Klasse *Demo.class* über das Netzwerk übertragen, von *Receiver* geladen und ausgeführt werden. Dazu muss *Receiver* zunächst gestartet werden, etwa mit folgender Kommandozeile:

```
> javacvm Receiver
Waiting on server socket port 1024 ...
```

In einem anderen Eingabefenster kann nun die Klasse *Demo.class* mit Hilfe von *Sender* zu *Receiver* übertragen werden. Bild 3 zeigt den Java-Quelltext von *Demo*.

```
> javacvm Sender Demo
```

Die so übertragene Klasse *Demo.class* wird auf der *Receiver*-Seite geladen und ausgeführt:

```
Hello dynamically loaded World!
```

So funktioniert's

Damit solch eine Netzwerkanwendung entwickelt werden kann, muss natürlich zunächst ein Netzwerkprotokoll, das sowohl *Receiver* and auch *Sender* kennen, definiert werden. In unserem Beispiel besteht das Protokoll aus zwei möglichen Kommandos *Stop* und *Upload* (Bild 4).

```
import java.io.*;
import java.net.*;
import java.util.Hashtable;
public class Receiver {
    static final int port = Integer.parseInt(System.getProperty("PORT", "1024"));
    public static void main(String[] args) {
        try {
            ServerSocket server = new ServerSocket(port);
            System.out.println("Waiting on server socket port "+port+" ...");
            boolean done;
            do {
                Socket s = server.accept(); /* wait for next connection */
                DataInputStream in = DataInputStream(s.getInputStream());
                String msg = in.readUTF(); /* read message type */
                done = msg.equals("STOP"); /* handle message "STOP" */
                if (msg.equals("UPLOAD")) { /* handle message "UPLOAD" */
                    String main = in.readUTF(); /* main class name */
                    Hashtable classes = new Hashtable();
                    for(int i=in.readInt(); i>0; i--) { /* receive classes */
                        String name = in.readUTF();
                        byte[] data = new byte[in.readInt()]; in.readFully(data);
                        classes.put(name, data); /* add name/data to hashtable */
                    }
                    ClassLoader l = new DataClassLoader(classes);
                    Class c = l.loadClass(main); /* load main class */
                    Object o = c.newInstance(); /* call constructor */
                }
                in.close(); s.close();
            } while (!done);
            server.close();
        } catch (Exception e) { e.printStackTrace(); }
    }

    static class DataClassLoader extends ClassLoader { /* user class loader */
        Hashtable classes; /* hashtable of names/data for classes */
        DataClassLoader(Hashtable classes) { this.classes = classes; }

        /* overwrite findClass() to search for class in hashtable */
        public Class findClass(String name) throws ClassNotFoundException {
            byte[] data = (byte[]) classes.get(name);
            if (data==null) throw new ClassNotFoundException(name);
            return defineClass(name, data, 0, data.length);
        }
    }
}
```

Bild 1: Klasse *Receiver* zum Laden von Klassen über einen *ClassLoader*.

```
import java.io.*;
import java.net.*;

public class Sender {

    static final String host = System.getProperty("HOST", "localhost");
    static final int port = Integer.parseInt(System.getProperty("PORT", "1024"));

    public static void main(String[] args) {
        try {
            if (args.length==0) {
                System.out.println("usage: jamaicavm Sender STOP | <class> {<class>}");
            } else {
                Socket s = new Socket(host, port);
                DataOutputStream o = new DataOutputStream(s.getOutputStream());
                if (args[0].equals("STOP")) {
                    o.writeUTF("STOP"); /* send STOP command */
                } else {
                    o.writeUTF("UPLOAD"); /* send UPLOAD command */
                    o.writeUTF(args[0]); /* main class name */
                    o.writeInt(args.length); /* number of classes to send */
                    for(int i=0; i<args.length; i++) {
                        byte[] data = loadFile(args[i]+".class");
                        o.writeUTF(args[i]); /* class name */
                        o.writeInt(data.length); /* classfile data length */
                        o.write(data,0,data.length); /* classfile data */
                    }
                }
                o.close();
                s.close();
            }
        } catch (IOException e) { e.printStackTrace(); }

        /* read file name into a byte array */
        public static byte[] loadFile(String name) throws IOException {
            File f = new File(name);
            DataInputStream i = new DataInputStream(new FileInputStream(f));
            byte[] d = new byte[(int) f.length()];
            i.readFully(d);
            i.close();
            return d;
        }
    }
}
```

Bild 2: Klasse Sender sendet Klassen dynamisch an Receiver

```
public class Demo {
    public Demo() {
        System.out.println("Hello dynamically loaded World!");
    }
}
```

Bild 3: Dynamisch nachgeladene Klasse Demo

Kommando	Wert	Typ
Stop:	"STOP"	String
Upload:	"UPLOAD"	String
	Hauptklasse	String
	Klassenzahl n	int
	Klasse 1: name	String
	len	int
	data	byte[len]
	Klasse 2: name	String
len	int	
data	byte[len]	
:		
Klasse n: name	String	
len	int	
data	byte[len]	

Bild 4: Protokoll für ClassLoader Beispiel

Sender generiert beim Hochladen von Klassen ein Upload-Kommando, das den Namen der Hauptklasse und die Namen und Classfile-Daten der einzelnen Klassen enthält. Receiver empfängt diese Daten entsprechend und speichert die geladenen Klassendaten in einer Hashtabelle, die als Schlüssel die Namen der Klassen verwendet, und als Wert jeweils

die dazugehörigen Classfile-Daten.

Diese Hashtabelle kann nun vom benutzerdefinierten *DataClassLoader* in *Receiver* verwendet werden, um diese Klassen zu Laden. Der *ClassLoader* implementiert die Methode *findClass()*, die von der VM aufgerufen wird, um Klassen zu Laden. In diesem Beispiel sucht *findClass()* die Daten der angeforderten Klasse in der Hashtabelle und generiert durch einen Aufruf von *defineClass()* daraus eine Klasse.

Nach dem Erzeugen eines *DataClassLoaders* mit den aus dem Upload-Kommando empfangenen Daten kann die Hauptklasse durch einen Aufruf von *loadClass()* in diesem *ClassLoader* geladen werden. Durch den Aufruf von *newInstance()* wird schließlich ein Objekt der Klasse erzeugt und der Konstruktor ausgeführt.

Stolpersteine

Gewöhnlich reicht es nicht, eine einzige Klasse zu übertragen, oft wird eine komplexe Kompo-

Kurse und Vorträge

Forschungszentrum Informatik

Das Forschungszentrum Informatik Karlsruhe hat sein Angebot erweitert. Das bewährte Schulungsangebot wurde um neue Themen, wie die Echtzeitprogrammierung, Automatisierung, sowie die Integration von Legacy-Systemen in Java Umgebungen ergänzt. Das aktuelle Schulungsprogramm des FZI finden Sie unter www.fzi.de/ajc/.

Java in Wireless und Embedded Umgebungen

12.-16. Juli 2004, 5-tägige Schulung. Aus dem Inhalt: Editionen, Konfigurationen und Profile; Anforderungen an Embedded Platforms, Knackpunkte: Java-Umgebung und Java-Anwendung; Wireless- und Embedded-Szenarien; Echtzeit-OS, VM und Tools; Java und Echtzeit; Java und Performance; Java und Co-Degröße. Dieser Kurs wird zusammen mit rpCube und Microconsult angeboten. Kontakt: info@aicas.com

nente aus mehreren Klassen im Zielsystem benötigt. Hierfür ist es erforderlich, dass der *ClassLoader* alle von der Hauptklasse referenzierten dynamisch geladenen Klassen laden kann. In diesem Beispiel ist dies möglich, indem dem *Sender* eine Liste an Klassen übergeben wird, wobei die erste Klasse die Hauptklasse ist.

Die Beispiele in Bild 1 und 2 haben keinerlei Fehlerbehandlung für die möglichen *Exceptions*, die durch die Netzwerkoperationen, oder beim Klassenladen und -instanzieren entstehen können. Eine vollständige Implementierung braucht entsprechende Fehlerbehandlung und *finally*-Anweisungen für ein sauberes Schließen und Freigeben aller Ressourcen im Falle einer Exception. Um die Beispiele übersichtlich zu halten wird dies hier nicht getan.

Ausblick

In der nächsten Ausgabe: So funktioniert Echtzeit Garbage Collection in JamaicaVM. (fs)

Veranstaltungen

RTS Embedded Systems, Paris

30. März - 1. April 2004. Die Show für Echtzeitleösungen und Embedded Software. Paris Expo - Porte de Versailles - Halle 3. aicas präsentiert seine echtzeit Java Technologien JamaicaVM und AERO-VM. Besuchen Sie uns an Stand N25.

SSTC, Salt Lake City, USA

19.-22. April 2004. Die Systems & Software Technology Conference. Besuchen Sie aicas auf der führenden, vom Department of Defense gesponsorten Softwarekonferenz.

Electronica, München

9.-12. November 2004. Die zweijährige Weltleitmesse electronica findet zusammen mit der *Embedded in Munich* statt. aicas ist selbstverständlich wieder dabei.

JamaicaVM auf NetSilicon's NS9750

Jamaica für NetSilicons neuesten NET+ARM Prozessor.

Basierend auf dem ARM 926EJ-S, ARMs leistungsfähigster ARM 9 Core, bietet der NS9750 die höchste Performance- und Integrationsebene für eingebettetes Networking. Der NS9750 zielt durch die Integration einer breiten Menge an Peripherie, wie 10/100 Ethernet, PCI, USB (host or device), I2C, 1284, serial ports, GPIO, und einem kosteneffektiven LCD Controller, auf flexible Connectivity-Lösungen. Dank DSP Instruktionen, Java byte code Beschleunigung und einer MMU, ist der NS9750 ein high-performance 32-Bit Prozessor der Geschwindigkeiten bis 200MHz ermöglicht. Er bietet voll duplex 10/100Base-T ethernet mit mehr als genügend zusätzlicher Prozessorleistung und Bandbreite für die anspruchvollsten embedded Anwendungen.

JamaicaVM ist eine perfekte Ergänzung zu NetSilicons Entwicklungsumgebung. Es ermöglicht statisches und dynamisches Laden von echtzeit object-orientierten Applikationen auf netzgebundene eingebettete Geräte. (fs)

Führende CORBA Technologie ORBexpress

aicas und Objective Interface Systems bieten Benutzern von JamaicaVM die CORBA Lösung ORBexpress ST für Java.

ORBexpress ist die führende CORBA Implementierung für eingebettete, high-performance und Echtzeit-Systeme. ORBexpress ist die Grundlage für viele missionskritische Systeme in den Bereichen Verteidigung, Aerospace, Telekommunikation, Prozesskontrolle, Transport, Medizin und Industrieautomatisierung.

Objective Interface Systems ist ein Pionier für fortgeschrittene echtzeit CORBA Technologie und bekannt für schnelle, qualitativ hochwertige Kommunikationsprodukte.

Die Common Object Request Broker Architecture (CORBA) ist ein Standard der Object Management Group (OMG) und bietet Interoperabilität zwischen verteilten Objekten. CORBA ist die weltweit führende Middleware Standard für den Austausch von Information, unabhängig von Hardwareplattform, Programmiersprache und Betriebssystem. CORBA erleichtert die Entwicklung von Sprach- und Orts-unabhängigen Schnittstellen für verteilte Objekte.

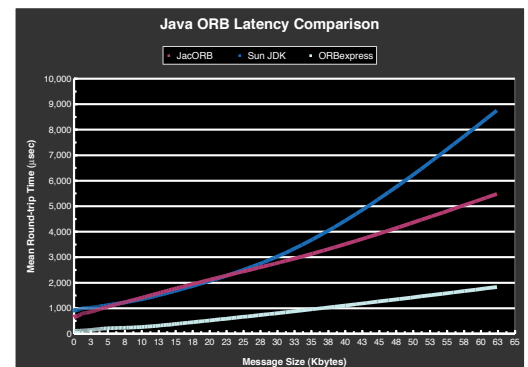
Dienste via OSGi

Dynamische Funktionen mit Java auf das Zielsystem bringen

Javas dynamisches Klassenladen ermöglicht zusätzliche Dienste. Die Open Service Gateway Initiative (OSGi) definiert ein Standard Framework für solche Dienste in vernetzten Umgebungen. Mitglieder von Branchen wie Heimelektronik, Telekommunikation, Automobil zusammen mit Software zielen auf immer vernetztere Umgebungen. Das OSGi Framework ermöglicht es Herstellern von Software Diensten sich auf den Dienst zu konzentrieren, und nicht auf Hardwareabhängigkeiten. Dienste werden in einfache Java Archive (JAR) gepackt mit zusätzlichen Informationen über diesen Dienst im Manifest.

Bitte kontaktieren Sie aicas für Informationen zu verfügbaren OSGi Lösungen für JamaicaVM. (ip)

2003 hat Objective Interface Systems ORBexpress ST for Java veröffentlicht, eine schnelle, skalierbare Implementierung des CORBA Standards für Java. Dieses high-performance ORB verringert für Java Entwickler das Risiko beim Bau verteilter, heterogener Anwendungen mit voller Interoperabilität und einheitlicher Unterstützung von einem Anbieter.



ORBexpress Leistungsvergleich

ORBexpress ST for Java übertrifft sogar die C++ ORBs vieler anderer Anbieter. Lockheed Martin's Advanced Technology Labs haben Benchmarks für eine Reihe an Java ORBs durchgeführt (s. www.atl.external.lmco.com/projects/QoS/). Wie im Diagramm zu sehen ist, demonstrieren diese unabhängigen Tests, dass ORBexpress deutlich schneller ist als andere Java ORBs; für kleine Datengrößen mehr als 8 Mal schneller als Sun's JDK ORB. Damit wird ORBexpress zusammen mit JamaicaVM zur idealen Lösung für Ihre verteilten Anforderungen. (jjh)

Kontakt

Redakteure:

Dr. James J. Hunt (jjh), Ingo Prötzel (ip), Dr. Torsten Rupp (tr), Roman Schnider (rs), Dr. Fridtjof Siebert (fs), Andy Walter (aw)

aicas GmbH

Hoepfner Burg
Haid-und-Neu-Str. 18
76131 Karlsruhe
Germany

tel +49.721.663.968-0
fax +49.721.663.968-99
email info@aicas.com
web www.aicas.com