Fachhochschule Bonn–Rhein–Sieg
Fachbereich Angewandte Informatik
Grantham-Allee 20
53757 Sankt Augustin

Verteilte und Parallele Systeme I
Prof. Dr. Rudolf Berrendorf

Vortrag über

Java Platform Profiling Architecture

am 03. Dezember 2003
von
Marc Berendes und Marco Krause

1 Einleitung

Die JPPA ist ein Projekt im Rahmen eines Java Specification Requests innerhalb des Java Community Process – Programms (http://jcp.org/en/jsr/detail?id=163).
Die Leitung dieser Spezifizierungs- – Kommission liegt bei Robert Field von Sun Microsystems, Inc. Die Mitglieder dieses Expertenkonsortiums bilden zahlreiche internationale Unternehmen, darunter u.a. Apple, Compaq, Hewlett-Packard, Hitachi, IBM, Motorola, Oracle, SAP und Sun Microsystems.

2 Bezug zur JVMPI

Ziel dieses Programms ist es, eine Nachfolger – API für die veraltete JVMPI zu spezifizieren. Die Motivation hierfür liegt in mehreren Nachteilen, die dem JVMPI heutzutage behaftet sind:

- Es ist sehr schwierig, mit dem JVMPI für moderne JVM zuverlässige Implementierungen zu produzieren.
- Implementierungen des JVMPI stören den eigentlichen Programmablauf wesentlich und führen sehr oft zu einer signifikanten Verlangsamung des Programmablaufes.
- Der Heap Profiling Mechanismus des JVMPI ist nicht skalierbar und hat einen starren Aufbau. Deshalb ist er nicht mehr zu den heutigen Garbage Kollektoren kompatibel. Er hat sehr oft komplett inkompatible Implementierungen verschuldet, weil er nur unzureichend spezifiziert wurde.
- Das JVMPI Interface ist nicht plattformunabhängig. Deshalb sind für die Anbieter von Implementierungen des JVMPI hohe zusätzliche Entwicklungskosten nötig, um ihre Produkte für andere Plattformen anbieten zu können. Aus diesem Grund werden meistens nur die häufigsten Plattformen wie Windows, Linux oder Solaris unterstützt.

Das Programm versucht deshalb, eine neue, plattformunabhängige, skalierbare und zuverlässige Spezifikation zu entwerfen. So soll die JPPA auch mit schon vorhandenen Interfaces Interoperieren können, z.B. mit der Java™ Platform Debugger Architecture (JPDA).

Die JPPA ist jedoch noch nicht fertig spezifiziert. Ein Release ist erstmals in der J2se Version 1.5 geplant. Von dort an soll sie die JVMPI komplett ersetzen, die dann als obsolete gekennzeichnet werden wird. Die Spezifizierungsbemühungen begannen am 22.1.2002.
Im Moment steht das Projekt auf dem Stand eines ersten öffentlichen Reviews.
In diesem sind 5 Entwürfe enthalten, die mehr oder weniger unverändert in den endgültigen Release eingehen sollen:

- Java Virtual Machine Tool Interface(JVMTI): Ein natives Interface, um Profiling- Agents zu implementieren.
- java.lang.management: Eine Monitoring - und Management - API
- java.lang.instrument: Eine prozessinterne Instrumentierungs- - API
- java.lang.Thread.getId(): Zur Unterstützung von übergeordneten APIs
- JVM-MANAGEMENT-MIB.mib: Remote SNMP Monitoring - und Management - Unterstützung (MIB Module)

3 Java Native Interface

Wenn eine Anwendung nicht komplett in Java geschrieben werden kann, dann wird das Java Native Interface (kurz: JNI) eingesetzt, um ein Schnittstelle zu anderen Programmiersprachen zu schaffen. Gründe, die den Einsatz von JNI rechtfertigen sind z.B.

- Es existiert bereits ein Programm, dass nicht in Java geschrieben wurde, sondern z.B. in C++. Damit das existierende Programm nicht neu geschrieben werden muss, kann man bestehende Funktionalitäten über JNI in Java einbauen.
- Zeitkritische Anwendungen, die unter der Java Virtual Machine zu langsam laufen würden, können in einer maschinennahen Sprache (z.B. C oder Assembler) programmiert werden und über JNI in Java eingebunden werden.
- Es werden betriebssystemspezifische Routinen benötigt.

Unter Benutzung von JNI kann man in so genanntem „native Code“, also dem Code einer weiteren Programmiersprache wie z.B. C, C++ oder Assembler auf mehrere Weisen mit dem Java Programm kommunizieren:

- Objekte (inklusive Arrays und Strings) können erstellt, inspiziert und verändert werden.
- Java Methoden können von extern ausgeführt werden.
- Exceptions können abgefangen und geworfen werden.
- Klassen können geladen werden.

Das Java Native Interface (JNI) ist der Nachfolger zur Native Method Interface (NMI) aus dem Java Development Kid (JDK) 1.0. Mit der Version 2 des JDK wurde NMI durch JNI abgelöst. Zudem wurde mit diesem Release ein Standard gesetzt, der die verschiedenen Native Interfaces der verschiedenen Hersteller (Sun, Netscape, Microsoft) zu einem Standard vereinen soll. Das hat den Vorteil, dass Programme, die JNI benutzen nicht nur auf der Virtual Machine laufen, auf der sie entwickelt wurden, sondern Herstellerunabhängig auf jeder Virtual Machine ausgeführt werden können.

Beispiel:
Nachfolgend ist ein Hallo Welt Beispielprogramm in C mit Java Anbindung aufgeführt.

Zuerst schreibt man das Java Programm und deklariert in diesem den native code, bzw. die native Methode. Danach kompiliert man den Java Quellcode.

Der Java Quellcode

```
class HelloWorld {
    public native void displayHelloWorld();

    static {
        System.loadLibrary("hello");
    }

    public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
    }
}
```

Danach generiert man mit <javah> und dem Parameter –jni aus dem Java Programm eine Headerdatei mit der formalen Signatur der C Methode.

Die mit <javah> generierte Headerdatei

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloWorld */

#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloWorld
 * Method:    displayHelloWorld
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Hat man diese Headerdatei erstellt, schreibt man das eigentliche C Programm und kompiliert die Headerdatei und das C Programm zusammen in

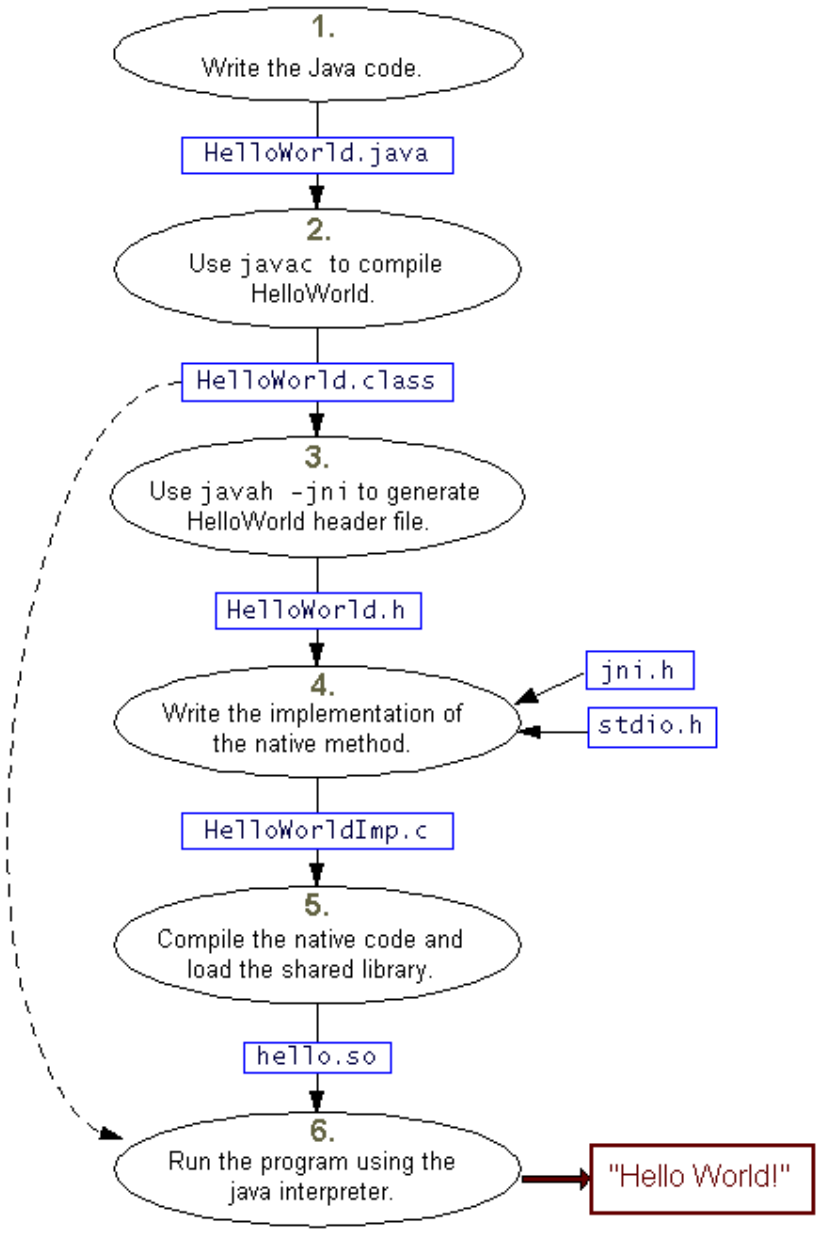
eine Shared Library Datei. Danach kann das Java Programm ausgeführt werden und es benutzt den native code des C Programms.

Der C Quellcode

```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject obj)
{
    printf("Hello world!\n");
    return;
}
```

In der folgenden Grafik, die wie der vorhergehende Quellcode aus einem JNI Tutorial von Sun (http://java.sun.com/docs/books/tutorial/native1.1/stepbystep/) kopiert ist, wird dieser Ablauf noch einmal graphisch verdeutlicht.



4 Java Virtual Machine Tool Interface (JVMTI)

Das Java Virtual Machine Tool Interface ist ein Interface zur Entwicklung und Leistungsanalyse. Es kann sowohl den Status einer aktiven Anwendung, die unter der Java Virtual Machine läuft, kontrollieren als auch die Ausführung dieser Anwendung selbst.

Die Idee dahinter ist ein Virtual Machine(VM)-Interface anzubieten, das ein breites Spektrum an Werkzeugen besitzt, die unter anderem folgende Funktionen beinhalten: profiling, debugging, Thread Analyse, monitoring und coverage Analyse.
Bei dem JVMTI handelt es sich um ein Zwei-Wege Interface, das bedeutet ein Client des Interfaces, im folgenden Agent genannt, kann über interessante Ereignisse durch Events benachrichtigt werden oder das Interface kann Anwendungen selbst durch viele Funktionen abfragen und kontrollieren, entweder durch Events oder unabhängig von ihnen.
Agenten kommunizieren direkt, durch ein herkömmliches Interface, mit der VM, die die zu überprüfende Anwendung ausführt. Diese Kommunikation geschieht durch ein herkömmliches Interface. Das Interface, das im gleichen Prozess läuft erlaubt maximale Kontrolle durch gleichzeitiges minimales Eingreifen durch das Tool. Typischer Weise sind Agenten sehr kompakt und können durch einen separaten Prozess kontrolliert werden, der den Hauptteil der Funktionen eines Werkzeuges implementiert ohne die Ausführung der zu analysierende Anwendung zu behindern.
Werkzeuge zur Leistungsanalyse können entweder direkt für das JVMTI entwickelt werden oder indirekt für das übergeordnete Interface der Java Platform Tool Architektur. Die Java Platform Tool Architektur ist eine Weiterentwicklung der Java Platform Debugging Architektur, welche auch höhere Ebenen des „**out-of-process**“ Debugger-Interfaces enthält. Die Interfaces dieser höheren Ebene sind für viele Werkzeuge besser geeignet, als die der JVMTI

4.1 Agenten

Agenten können in jeder herkömmlichen Sprache implementiert werden, die native Code unterstützt.
Die Funktions-, Event-, Datentypen-, und Konstantendefinitionen, die benötigt werden um das JVMTI zu nutzen, sind in der include Datei jvmti.h definiert. Um diese Definitionen zu nutzen muss man J2SE™ samt Directory in den eigenen Pfad einbinden und *#include <jvmti.h>* zu seinem Code hinzufügen.

JVMTI Agent Command Line Options

Die folgenden beiden Kommandozeilen Argumente können benutzt werden um Agenten korrekt zu starten und laufen zu lassen. Dieses Argument kennzeichnet sowohl die Bibliothek, die den Agenten enthält als auch einen Options-String, der beim Start entsprechend weitergereicht wird.

-agentlib:<agent-lib-name>=<options>

Der Name, der *–agentlib:* folgt, ist der Name der Bibliothek die geladen werden soll. Typischer Weise ist der Name erweitert um ein Betriebssystemspezifischen Dateinamen. Die Option <*options*> wird beim Start an den Agenten geleitet.
Bsp: *-agentlib:foo=opt1,opt2* – die VM wird versuchen die Bibliothek foo.dll aus dem System-Pfad unter Windows oder libfoo.so von der LD_Library_Path unter Solaris zu laden.

-agentpath: <path-to-agent>=<options>

Der Pfad, der <path-to-agent> folgt, ist der absolute Pfad von wo die Bibliothek geladen werden soll. Eine Dateinamenerweiterung wird nicht verwendet, die Optionen werden jedoch auch hier beim Start an den Agenten übergeben.
Bsp.: -agentpath:c:\myLibs\foo.dll=opt1,opt2 – die VM wird versuchen die Bibliothek c:\myLibs\foo.dll zu laden.

Starten des Agenten

Die Bibliothek muss eine Startfunktion mit dem folgenden Prototyp ausführen:

```
JNIEXPORT jint JNICALL
Agent_OnLoad (JavaVM *vm, char *options, void *reserved)
```

Diese Funktion wird von der VM aufgerufen, wenn die Bibliothek geladen wird. Die VM sollte die Bibliothek frühstmöglich in der VM-Initialisierung laden, so dass:

- System Eigenschaften gesetzt werden können bevor sie beim start der VM benutzt werden
- alle Funktionalitäten noch verfügbar sind
- kein Bytecode ausgeführt wurde
- keine Klassen geladen wurden
- keine Objekt erstellt worden.

Der frühe Start der Agenten ist also erforderlich, damit sie die gewünschten Funktionalitäten setzen können.
Im JVMDI unterstützt das Kommandozeilen Flag *–Xdebug* nur eine sehr grobe Kontrolle der Funktionalitäten. Die JVMPI Implementation benutzt verschiedene Tricks um verschiedene JVMPI-Schalter anzubieten. Kein sinnvolles Kommandozeilen Flag kann eine solch feingranulare Kontrolle zur Verfügung stellen, wie sie benötigt wird um die Funktionalitäten und die Leistung in der Waage zu halten. Der frühe Start ist also erforderlich, damit die Agenten die Ausführung der Umgebung kontrollieren können.

Der Rückgabewert von Agent_OnLoad wird genutzt um Fehler anzuzeigen. Jeder Wert, der nicht 0 ist, ist ein Fehler und beendet die Ausführung der VM.

Shutdown der Agenten

Die Bibliothek kann optional eine Shutdown-Funktion mit dem folgenden Prototyp ausführen:

```
JNIEXPORT void JNICALL
Agent_OnUnload (JavaVM *vm)
```

Diese Funktion wird von der VM aufgerufen, wenn die Bibliothek beendet wird. Diese Funktion wird aufgerufen, wenn einige Plattform spezifische Mechanismen das Beenden veranlassen oder die Bibliothek durch den Shutdown der VM beendet wird. Dabei ist egal ob die VM normal oder durch einen Fehler beendet wurde.

4.2 Funktionen

Aufrufen der JVMTI-Funktionen

Native Code kann Features des JVMTI durch aufrufen der JVMTI-Funktionen ausführen. Das Aufrufen dieser Funktionen geschieht ähnlich wie bei der JNI durch einen Interface Zeiger. Der Zeiger wird auch environment pointer genannt und besitzt den Datentyp jvmtiEnv*.
Der Erste Werte beim aufrufen der Umgebung ist ein Zeiger auf die Funktionstabelle. Die Funktionstabelle ihrerseits ist ein Array mit Zeigern auf die Funktionen, von denen jede an einer vordefinierten Speicherstelle im Array abgelegt ist. Der Environment Pointer liefert den Kontext und ist der erste Parameter jeder Funktion.
Beispiel:

```
JvmtiEnv *jvmti;
...
jvmtiError err -> (*jvmti)->GetLoadedClasses(jvmti, &class_count,
&classes);
```

Ein JVMTI Umfeld kann durch das Aufrufen der Funktion GetEnv erstellte werden:

```
jvmtiEnv *jvmti;
...
(*jvm) ->GetEnv (jvm, &jvmti, JVMTI_VERSION_1_0);
```

Jeder Aufruf der GetEnv Funktion erstellt eine neue JVMTI-Verbindung und folglich eine neue JVMTI-Umgebung. Das Versions-Argument muss eine gültige JVMTI Version sein. Die zurückgegebene Umgebung kann eine andere Version haben, wie angefordert wurde muss aber kompatibel sein. Ist keine kompatible Version verfügbar wird *JNI_EVERSION* zurückgegeben. Dies ist der Fall, wenn z.B. JVMTI in der laufenden VM nicht unterstützt wird. Um spezielle JVMTI-Umgebungen zu erstellen können andere Interfaces hinzugefügt werden. Jede Umgebung besitzt einen eigenen Status (z.B. angeforderte Events, Event Handling Funktionen und Fähigkeiten). Es unterscheidet sich vom JNI also in dem Punkt, dass in JNI pro Thread nur eine Umgebung existiert während JVMTI über Threads hinweg arbeiten kann und dynamisch erzeugt wird.

Rückgabewerte

JVMTI Funktionen geben immer einen Error Code mit Hilfe des Rückgabewertes der *jvmtiError* Funktion zurück. Einige Funktionen können zusätzliche Werte zurückgeben indem sie Zeiger benutzen. Das kann dazu führen, dass in machen Fällen Funktionen zusätzlichen Speicher reservieren, den man dann in seinen Programmen explizit freigeben muss. Das ist aber in den Beschreibungen der entsprechenden Funktionen festgehalten.
Für den Fall, das eine Funktion einen Fehler erhält (jeder Rückgabewert der nicht *JVMTI_ERROR_NONE* ist) ist der Wert des referenzierten Speichers zwar einerseits unbestimmt, es sind aber jedoch weder Speicher noch globale Variablen reserviert worden. Tritt ein Fehler auf Grund von ungültigen Eingaben auf, wird keine Aktion ausgeführt.
Leere Listen, Arrays, Sequenzen, ect. werden als NULL zurückgegeben.

JNI Objekt Referenzen

JVMTI Funktionen identifizieren Objekte mit den JNI-Referenzen (jobject und jclass) und ihren Vererbungen (jthread und jthreadGroup). Referenzen, die an JVMTI-Funktionen übergeben werden, können

entweder global oder lokal sein, sie müssen nur streng referenziert sein. Alle Referenzen, die von JVMTI-Funktionen zurückgegeben werden sind lokal. Diese lokalen Referenzen werden erstellt während das JVMTI aufgerufen wird. Wenn Threads von native Code zurückgegeben werden sind alle lokalen Referenzen freigegeben. Einige Threads, wie z.B. auch die Agenten, werden nicht von native Code zurückgegeben. Ein Thread kann, 16 lokale Referenzen zu erzeugen ohne dass ein explizites Management benötigt wird. Solange sie also nur eine begrenzte Anzahl an JVMTI-Aufrufen ausführen wird kein explizites Management benötigt. Jedoch benötigen lange laufende Threads ausdrückliches lokales Bezugsmanagement normalerweise mit den JNI Funktionen *Push-LocalFrame* und *PopLocalFrame*. Andererseits, um Referenzen über die Rückgabe der maschinenabhängigen Sprache hinaus zu behalten, müssen sie in globale Referenzen umgewandelt werden.

Im Folgenden werden ein paar Funktionen aufgezählt und erläutert, die zu dem JVMTI gehören:

- Speichermanagement
  - um Speicherbereiche zu reservieren und wieder freizugeben
- Thread
  - Funktionen wie Statusmeldung, Thread stoppen und weiter laufen, ect.
- ThreadGroup
  - beinhaltet nur Get-Methoden auf den Thread Gruppen
- Stack Frame
  - liefert Informationen über den Stacks eines Threads
- Heap
  - Methoden zum analysieren des Heaps und dessen Inhalt
- Local Variable
  - Funktionen um Werte lokaler Variablen abzufragen bzw. zu setzen

- Breakpoint
  - setzten und löschen von Breakpoints
- Watched Field
  - überwachen von Feldzugriffen
- Class
  - liefert Informationen über div. Klassen
- Object
  - liefert Informationen der Objekte
- Field
  - liefert Informationen der Felder
- Method
  - liefert Informationen der Methoden
- JNI Funktionsinterpreter
  - abfangen und zurücksenden der JNI Aufrufe
- Event Management
  - Generierung, Aktivierung und Deaktivierung von Events
- Extension Mechanism
  - erlaubt JVMTI Implementierung um weitere Funktionen und Events zur Verfügung zu stellen
- Capability
  - erlaubt das Ändern der Funktionalität von JVMTI
- Timers
  - detail Informationen über Timer, wie Maximalwert, ect.
- System Properties
  - setzten und Rückgabe von System Eigenschaften der Virtual Machine

Fehlermeldungen

Es ist in der Verantwortlichkeit der Agenten die JVMTI-Funktionen mit gültigen Parametern und im richtigen Kontext aufzurufen. Für eine Implementation kann es schwer, uneffizient oder gar unmöglich sein Fehlerursachen zu ergründen. Aus diesem Grund gibt, wie bereits

erwähnt, jede JVMTI-Funktion einen *jvmtiError* Error code zurück. Die Fehlermeldungen sind in der Liste *Function Specific Required Errors* aufgeführt und müssen von der Implementation eingebunden werden. Alle anderen Fehlermeldungen sind empfohlene Antworten.

Allgemeine Fehlermeldungen

Die folgenden Fehlermeldungen können von jeder beliebigen Funktion zurückgegeben werden.

JVMTI_ERROR_NONE (0)

Kein Fehler ist aufgetreten. Das ist der Error code der nach erfolgreichem Abschluss der Funktion zurückgegeben wird.

JVMTI_ERROR_NULL_POINTER (100)

Ungültiger Zeiger.

JVMTI_ERROR_OUT_OF_MEMORY (110)

Die Funktion versucht Speicher zu reservieren, es ist aber keiner verfügbar.

JVMTI_ERROR_ACCESS_DENIED (111)

Die gewünschte Funktionalität ist nicht in dieser Virtual Machine verfügbar.

JVMTI_ERROR_UNATTACHED_THREAD (115)

Der Thread, der genutzt wurde um eine Funktion aufzurufen ist nicht in die VM eingebunden. Aufrufe müssen durch eingebunden Threads gemacht werden.

JVMTI_ERROR_INVALID_ENVIRONMENT (116)

Keine Verbindung zur JVMTI-Umgebung vorhanden oder es ist keine Umgebung.

JVMTI_ERROR_WRONG_PHASE (112)

Die gewünschte Funktionalität ist in der gegenwärtigen Phase nicht verfügbar. Wird immer zurückgegeben, wenn die VM zu Ende gelaufen ist.

JVMTI_ERROR_INTERNAL (113)

Ein unerwarteter Fehler ist aufgetreten.

Function Specific Required Errors

Die folgenden Fehlermeldungen werden von einigen JVMTI-Funktionen zurückgegeben, wenn die Bedingung eintritt.

JVMTI_ERROR_INVALID_PRIORITY (12)

Ungültige Priorität.

JVMTI_ERROR_THREAD_NOT_SUSPENDED (13)

Der Thread wurde nicht suspendiert.

JVMTI_ERROR_THREAD_SUSPENDED (14)

Der Thread ist bereits suspendiert.

JVMTI_ERROR_THREAD_NOT_ALIVE (15)

Diese Operation erfordert einen lebenden Thread – er muss gestartet und darf noch nicht wieder gestorben sein.

JVMTI_ERROR_CLASS_NOT_PREPARED (22)

Die Klasse wurde geladen aber noch nicht zum Start vorbereitet.

JVMTI_ERROR_TYPE_MISMATCH (34)

Die Variable ist nicht der Typ, der in der Funktion benutzt wird.

JVMTI_ERROR_INVALID_SLOT (35)

Ungültiger Slot.

JVMTI_ERROR_DUPLICATE (40)

Item bereits gesetzt.

JVMTI_ERROR_NOT_FOUND (41)

Gewünschtes Element (z.B. Field oder Breakpoint) nicht gefunden.

JVMTI_ERROR_NOT_MONITOR_OWNER (51)

Der Thread gehört nicht dem raw monitor.

JVMTI_ERROR_INTERRUPT (52)

Der Aufruf wurde unterbrochen bevor er beendet zu Ende war.

JVMTI_ERROR_NOT_AVAILABLE (98)

Die Funktionalität ist in dieser VM nicht verfügbar.

JVMTI_ERROR_ABSENT_INFORMATION (101)

Die gewünschte Information ist nicht verfügbar.

JVMTI_ERROR_INVALID_EVENT_TYPE (102)

Die spezifiziert Event Typen-ID wurde nicht erkannt.

Function Specific Agent Errors

Die folgenden Fehlermeldungen werden nur von einigen JVMTI-Funktionen bei ungültigen Parametern für die Agenten oder ungültigem Kontext zurückgegeben. Um diese Fehlermeldungen zu erkennen ist eine Implementation nicht erforderlich.

JVMTI_ERROR_INVALID_THREAD (10)

Der übergebene Thread ist ungültig.

JVMTI_ERROR_INVALID_FIELDID (25)

Ungültiges Field.

JVMTI_ERROR_INVALID_METHODID (23)

Ungültige Methode.

JVMTI_ERROR_INVALID_LOCATION (24)

Ungültige Position.

JVMTI_ERROR_INVALID_OBJECT (20)

Ungültiges Objekt.

JVMTI_ERROR_INVALID_CLASS (21)

Ungültige Klasse.

JVMTI_ERROR_MUST_POSSESS_CAPABILITY (99)

Die genutzte Funktionalität ist in dieser Umgebung ungültig.

JVMTI_ERROR_INVALID_THREAD_GROUP (11)

Ungültige Thread Group.

JVMTI_ERROR_INVALID_MONITOR (50)

Ungültiger raw monitor

JVMTI_ERROR_ILLEGAL_ARGUMENT (103)

Nicht erlaubtes Argument.

JVMTI_ERROR_INVALID_TYPESTATE (65)

Der Status des Threads wurde geändert und ist nun inkonsistent.

JVMTI_ERROR_UNSUPPORTED_VERSION (68)

Eine neue Klasse besitzt eine Versionsnummer, die in dieser VM nicht unterstützt wird.

JVMTI_ERROR_INVALID_CLASS_FORMAT (60)

Das Format einer neuen Klasse ist ungültig (die VM gibt einen *ClassFormatError* zurück).

JVMTI_ERROR_UNSUPPORTED_REDEFINITION_METHOD_ADDED (63)

Eine neue Klasse erfordert eine zusätzliche Methode.

JVMTI_ERROR_UNSUPPORTED_REDEFINITION_SCHEMA_CHANGED (64)

Eine neue Klassenversion ändert ein Field.

JVMTI_ERROR_FAILS_VERIFICATION (62)

Klasse kann nicht überprüft werden.

.

JVMTI_ERROR_UNSUPPORTED_REDEFINITION_METHOD_DELETED (67)

Eine Methode, die in der alten Klassenversion definiert wurde, wird nicht in der neuen definiert.

JVMTI_ERROR_NAMES_DONT_MATCH (69)

Der neue Klassenname ist unterschiedlich zum alten Klassenobjekt.

4.3 Events

EventHandling

Agenten können über viele Ereignisse informiert werden, die in Anwendungsprogrammen vorkommen.

Um Events zu benutzen, wird eine Gruppe von callback-Funktionen mit SetEventCallbacks versehen. Für jedes Ereignis wird die entsprechende callback-Funktion aufgerufen. Argumente der callback-Funktionen sorgen für Zusatzinformation zu dem Event. Die callback-Funktion wird üblicherweise innerhalb von Applikations-Threads aufgerufen. Da die JVMTI-Implementierung diese Events aber in keiner Weise puffert, müssen Event-callback´s vorsichtig benutzt und geschrieben werden. Hier sind

einige allgemeine Richtlinien. Für weitere Informationen sollten die individuellen Event-Beschreibungen angesehen werden.

- Eine während der Durchführung geworfene Exception eines Event-callback´s kann eine aktuelle Exception in einem aktuellen Applikations-Thread überschreiben. Es muss nur sichergestellt werden, dass eine Exception eines JNI-Anrufs durch ein Event-callback erzeugt werden kann.
- Die JVMTI-Implementierung puffert keine Events. Wenn ein Agent Events gleichzeitig bearbeiten muss, kann er einen raw monitor in den Event-callback-Funktionen benutzen, um Event-callback-Funktionen zu serialisieren.

Einige JVMTI-Ereignisse identifizieren Gegenstände mit JNI-Referenzen. Alle Referenzen in JVMTI-Events sind JNI-Lokalreferenzen und werden ungültig, nachdem das Event-callback sie zurückgibt. Speicher, der durch Zeiger referenziert ist, ist nachdem ein Event zurückkommt nicht mehr referenzieren.

Events können mit der Funktion *SetEventNotificationMode* freigegeben und gesperrt werden. Alle Events sind anfänglich disabled. Folgende Punkte müssen beachtet werden, um ein Event zu empfangen:

- Wenn ein Event eine Funktionalität benötigt, muss diese Funktionalität mit *AddCapabilities* hinzugefügt werden.
- Ein callback für ein Event muss mit *SetEventCallbacks* gesetzt werden.
- Ein Event muss mit *SetEventNotificationMode* freigegeben werden.

Ein Thread, der ein Event erzeugt, verändert nicht seinen Status (z.B. kann ein entsprechendes Event den Thread nicht suspendieren). Wenn jedoch ein Agent dies verlangt, kann er den Thread mit *SuspendThread* suspendieren.

Wenn ein Event in mehreren Umgebungen freigegeben ist, wird das Event zu jedem Agenten in der Reihenfolge gesandt in der die Umgebungen geschaffen wurden.

Multiple Co-located Events

In vielen Situationen ist es möglich, dass an einer Stelle im Thread mehrere Events ausgelöst werden. Wenn dies geschieht, werden alle Events durch die Event-callbacks in der, in diesem Abschnitt spezifizierten Ordnung, gemeldet.
Wenn die gegenwärtige Stelle an der Einsprungstelle einer Methode ist, wird das *MethodEntry*-Event vor jedem anderen Event, an dieser Stelle in dem gleichen, Thread gemeldet.

Wenn ein singleStep- oder Breakpoint-Event eintritt, wird das Event sofort ausgelöst, bevor der Code an dem entsprechenden Stelle ausgeführt ist. Diese beiden Events werden vor den anderen Events ausgeführt, die vom Code an der gleichen Stelle in dem gleichen Thread hervorgerufen werden. Wenn sowohl ein Step- als auch Breakpoint-Event im gleichen Thread und an der gleichen Stelle im Code ausgelöst werden, wird das Step-Event vor dem Breakpoint-Event ausgeführt.

Wenn die aktuelle Stelle im Code der Austrittspunkt einer Methode ist (kurz bevor man einen Wert dem Aufrufenden zurückgibt), werden die Events *MethodExit* und *FramePop* (wenn verlangt) nach allen anderen Events an dieser Stelle im gleichen Thread ausgeführt. Im Gegensatz zu den eben zwei genannten Events gibt es hier keine spezifizierte Reihenfolge der Ausführung auf einander.

Co-located Events können während der Ausführung durch einige andere Events vom Agenten an der gleichen Stelle im gleichen Thread ausgelöst werden. Wenn solch ein Event, vom Typ Y, während seiner Ausführung von einem Event des Typen X ausgelöst wird, und X geht Y voraus, wird das Co-located Event Y ausgeführt. Wenn X nicht vorausgeht, wird Y nicht

ausgeführt. Zum Beispiel wenn ein Breakpoint während der Ausführung von SingleStep gesetzt wird, wird dieser Breakpoint gemeldet, bevor der Thread sich außerhalb des Codestücks bewegt.

5 java.lang.management

Das java.lang.management Paket wird von Sun selber unter der Überschrift „Monitoring and Management for the Java Virtual Machine“ beschrieben. Die Zielsetzung des Paketes liegt darin, eine Standard API zu generieren, mit der man den Zustand einer Virtual Machine zur Laufzeit observieren kann und deren Umgebungsparameter zu verwalten. Setzt man das Paket ein, kann man den Zustand seines Programms jederzeit überwachen und Schlüsse auf die Performance und auf ggf. auftretende Fehler und Engpässe ziehen. Dies ist vor allem im Produktionseinsatz hilfreich, da das Programm dafür nicht beendet werden muss, wenn es entsprechend angelegt wurde. Das API wurde für die folgenden Szenarien entwickelt und abgestimmt:

- Ein Java Programm benötigt eine Möglichkeit während der Laufzeit die Virtual Machine, auf der das Programm läuft zu analysieren und zu überwachen.
- Entwicklung von Systemverwaltungsprogrammen, die mehrere Instanzen der Virtual Machine auf einer Maschine oder in einem verteilten System überwachen können.
- Überwachung des zugrunde liegenden Betriebssystems

Bei der Entwicklung einer solchen API stehen einige Punkte im Vordergrund, die auf jeden Fall beachtet werden müssen, damit das Produkt auch brauchbar eingesetzt werden kann. Dies sind unter anderem:

- Eine gute *Java API*, die es erlaubt, eine plattformunabhängige Komponente in die Software mit einzubauen, um das eigene Programm zu überwachen.

- Sehr *geringe Nutzung von Systemressourcen*, damit die Monitoring Funktion nicht das Endergebnis verfälscht. Es darf also kaum Overhead für die Monitoring API erzeugt werden, denn es soll die Performance des Programms und nicht der Monitoring Funktionen gemessen werden.
- *Mehrere Clients* müssen in der Lage sein, eine einzige virtual machine zu überwachen. Dies kann z.B. bei Systemweiten Monitoring Programmen hilfreich sein, wenn ein Systemadministrator die Performance der laufenden Anwendungen überprüfen möchte und statistische Werte erfragen möchte.

Eine Virtual Machine kann eigene plattformspezifische Erweiterungen in das management interface einbauen, indem plattformabhängige Interfaces definiert werden, die das standard management interface erweitern. In der *ManagementFactory* Klasse können diese sogenannten *MBeans* dann zurückgegeben werden.

Die *ManagementFactory* Klasse ist, wie der Name bereits verlauten lässt, eine Factory Klasse. Sie besteht aus statischen Methoden, die entweder eine Instanz oder eine Liste aller Instanzen, die im Management Interface einer Komponente der virtual machine implementiert sind, zurückgibt. Eine Instanz, die ein Management Interface implementiert nennt sich *managed bean* oder kurz *MBean*.

Eine Anwendung kann auf drei verschiedene Arten die Messergebnisse einer virtual machine überwachen:

- Die Methoden des MBeans direkt aufrufen.
- Über einen MBeanServer die Methoden ansprechen.
- Einen MBean Proxy erzeugen und dann dessen Methoden ansprechen.

Die beiden Methoden über den MBeanServer und den MBeanProxy benötigen die Registrierung der MBeans an einem MBeanServer. Dazu steht die Funktion *registerBeans* in der ManagementFactory Klasse zur Verfü-

gung. Jedes MBean hat einen eindeutigen Objektnamen, mit dem es beim MBeanServer registriert ist.

Ein kleines Beispiel verdeutlicht diese Zusammenhänge:

Wir haben von einem Hersteller *xyz* eine virtual machine Implementation, mit einem plattformspezifischen Interface, welches wir *com.xyz.java.management.XYZRuntimeMBean* nennen. Es ist eine Erweiterung zur standard *RuntimeMBean*.
Die Methode *ManagementFactory.getRuntimeMBean()* gibt in diesem Fall *XYZRuntimeMBean* zurück, die Zugriff auf die standard virtual machine spezifischen Attribute und die *xyz* herstellerspezifischen, plattformabhängigen Attribute erlaubt.
Die nachfolgenden Codeschnipsel in Pseudocode stellen die oben genannten drei Arten auf MBeans zuzugreifen dar. Dabei nehmen wir an, dass wir neben dem Standard Attribut *VmVendor* noch ein herstellerspezifisches Attribut *XYZFoo* haben, das durch die getter Methode *getXYZFoo* angesprochen werden kann und im *XYZRuntimeMBean* Interface spezifiziert ist.

Das MBean direkt ansprechen:

```
// Get the standard attribute "VmVendor"
String vendor = mbean.getVmVendor();

if (mbean instanceof XYZRuntimeMBean) {
    // Type cast to the platform-specific interface
    XYZRuntimeMBean xyzMBean = (XYZRuntimeMBean) mbean;

    // Get platform-specific attribute "XYZFoo"
    int foo = xyzMBean.getXYZFoo();
}
```

Das MBean über einen MBeanServer ansprechen:

```
MBeanServerConnection mbs;

// Connect to a running JVM (or itself) and get MBeanServerCon-
nection
// that has the JVM MBeans registered in it
...
```

```
   try {
       // Assuming the RuntimeMBean has been registered in mbs
       ObjectName        oname        =        new        Object-
Name(MBeanNames.RUNTIME_MBEAN);

       // Get standard attribute "VmVendor"
       String  vendor  =  (String)  mbs.getAttribute(oname,  "Vm-
Vendor");

       // Get platform-specific attribute "XYZFoo"
       Integer foo = (Integer) mbs.getAttribute(oname, "XYZFoo");
   } catch (....) {
       // Catch the exceptions thrown by ObjectName constructor
       // and MBeanServer.getAttribute method
       ...
   }
```

Das MBean über einen MBeanProxy ansprechen:

```
   MBeanServerConnection mbs;

   // Connect to a running JVM (or itself) and get MBeanServerCon-
nection
   // that has the JVM MBeans registered in it
   ...

   // Assuming the RuntimeMBean has been registered in mbs
   ObjectName oname = new ObjectName(MBeanNames.RUNTIME_MBEAN);

   // Get a MBean proxy for RuntimeMBean interface
   RuntimeMBean proxy = (RuntimeMBean)
       MBeanServerInvocationHandler.newProxyInstance(mbs,
                                                     oname,
                                                     Runtime-
MBean.class,
                                                     false);
   // Get standard attribute "VmVendor"
   String vendor = proxy.getVmVendor();

   // Access MBean proxy as it were a managed object obtained from
   // the factory method in ManagementFactory class
   if (mbs.isInstanceOf(oname, XYZRuntimeMBean.class)) {
       XYZRuntimeMBean proxy = (XYZRuntimeMBean)
           MBeanServerInvocationHandler.newProxyInstance(mbs,
                                                         oname,
                                                         XYZRun-
timeMBean.class,
                                                         false);
       // Get platform-specific attribute "XYZFoo"
       int foo = xyzProxy.getXYZFoo();
   }
```

5.1 Klassen

Das Paket java.lang.management besteht im Einzelnen aus den folgenden *Klassen*:

```
class java.lang.Object
 class         java.util.EventObject         (implements
                                    java.io.Serializable)
     class javax.management.Notification
       class java.lang.management.SensorNotification
```

Diese Klasse dient der Verwaltung und Überprüfung von Triggern. Ein MBean wird angetriggert, wenn ein bestimmter Event auftritt. Hier die Anzahl der Aufrufe eines Triggers gespeichert.

```
class java.lang.management.ManagementFactory
```

Dies ist eine der wichtigsten Klassen (zuvor bereits angesprochen). Sie ist eine factory Klasse, die managed Beans (MBeans) verwaltet.

```
class java.lang.management.MbeanNames
```

Diese Klasse beinhaltet String Konstanten, die bestimmte Objektnamen von MBeans definiert.

```
class java.lang.management.MemoryType (implements
                                  java.io.Serializable)
```

Diese Klasse definiert die Speichertypen, die MBeans observieren können. Ein Beispiel ist der gemeinsame Heap, den mehrere Threads in einer virtual machine benutzen.

```
class java.lang.management.MemoryUsage (implements
                                java.io.Serializable)
```

Diese Klasse stellt Methoden bereit, die über den Speicher informieren. Hier wird zwischen initSize (Zu Beginn initialisierter Speicher), user (Benutztes Speicher), commited (Zugesicherter Speicher) und maxSize (Maximal verfügbarer Speicher) unterschieden.

```
class java.security.Permission (implements
                java.security.Guard, java.io.Serializable)
     class java.security.BasicPermission (implements
                                java.io.Serializable)
         class java.lang.management.ManagementPermission
```

Diese Klasse verwaltet Rechte, die ein SecurityManager prüft, wenn Code mit einem SecurityManager über das Management Interface ausgeführt wird. Es gibt Rechte für Controling und Monitoring.

```
class java.lang.management.ThreadInfo (implements
                                java.io.Serializable)
```

Diese Klasse stellt Methoden bereit, um Informationen zu Threads angezeigt zu bekommen. Die Informationen werden in drei Rubriken unterteilt. In General, Execution und Synchronisation Informationen.

```
class java.lang.management.ThreadState (implements
                                java.io.Serializable)
```

Die ThreadState Klasse beschreibt die Zustände, die ein Thread haben kann. Diese sind betriebssystemunabhängig, da sie nur auf der Java virtual machine laufen.

```
class java.lang.Throwable (implements
                  java.io.Serializable)
  class java.lang.Exception
    class java.lang.RuntimeException
      class java.lang.management.AlreadyRegisteredException
```

Diese Exceptions werden von der Klasse ManagementFactory.registeredMBeans geworfen, wenn versucht wird ein MBean zu registrieren, dass bereits bei einem MBeanServer registriert ist.

5.2 Interfaces

Des Weiteren besteht das Paket java.lang.management im Einzelnen aus den folgenden *Interfaces*:

```
interface java.lang.management.ClassLoadingMBean
```

Dies ist das management Interface für das Class Loading System der virtual machine.

```
interface java.lang.management.CompilationMBean
```

Dieses Interface ist das management Interface für das compilation system der Java virtual machine.

```
interface java.lang.management.MemoryManagerMBean
  interface java.lang.management.GarbageCollectorMBean
```

MemoryManagaerMBean ist das management Interface für einen Speichermanager. Ein Speichermanager benötigt Zugriff auf einen oder mehrere Speicherbereiche in der virtual machine.
Das GarbageCollectorMBean Interface stellt ein management Interface für die Garbage Collection dar. Ein Garbage Collector gibt Speicher von unerreichbaren, nicht mehr referenzierten, Objekten wieder frei.

```
interface java.lang.management.MemoryMBean
```

Dies ist das management Interface für das Speichersystem der virtual machine.

```
interface java.lang.management.MemoryPoolMBean
```

Dies ist das management Interface für Speicherbereiche in der virtual machine.

```
interface java.lang.management.OperatingSystemMBean
```

Dies ist das management Interface für das zugrunde liegende Betriebssystem und die Rechnerarchitektur.

```
interface java.lang.management.RuntimeMBean
```

Dies ist das management Interface für das runtime System der virtual machine. Hier werden Funktionen wie getuptime() und ähnliche Informationen deklariert.

```
interface java.lang.management.ThreadMBean
```

Dieses Interface ist das management Interface zur Thread Verwaltung der virtual machine. Hier werden z.B. Informationen wie die Thread ID (positiver long Wert) oder die Thread CPU time (Rechenzeit, die ein Thread verbraucht hat) in ms definiert.

6 java.lang.instrument

Das Package java.lang.instrument besteht nur aus einer einzigen funktionalen Klasse, einer exception Klasse und zwei Interfaces. Diese werden hier nicht genauer beschrieben, da das Package noch nicht implementiert ist und hier nur die Funktionalität des Packages erläutert werden soll.

Das Package erlaubt es, laufende Programme zu steuern/erweitern. Dies wird durch eine Veränderung des Bytecodes der jeweiligen Funktion bewirkt.

Dazu muss man einen Agent programmieren, der dann mit folgender Option an die Java virtual machine übergeben wird:

```
-javaagent classname=option
```

Es können mehrere dieser Optionen an eine virtual machine übergeben werden. Jeder Agent hat seinen eigenen classname und seine eigenen Optionen. Allerdings darf ein classname von mehr als einem Agenten benutzt werden.

Die Klasse, die im classname benannt wird, muss eine public static premain Methode haben, die wie folgt deklariert wird:

```
public static void premain(String agentArgs,
                           Instrumentation inst);
```

Nach der Initialisierung der virtual machine wird die premain Methode vor dem Aufruf der main Methode aufgerufen. Die premain Methode muss sich selbst beenden, bevor die main Methode gestartet wird. Die premain

Methode wird mit den selben Rechten und den gleichen classloader rules wie die main Methode ausgeführt.
Jeder Agent bekommt seine Parameter als String übergeben. Sollen zusätzliche Parameter übergeben werden, müssen die Parameter in einen String zusammengefügt und der übergebene String von der premain Methode geparsed werden, die dann die benötigten Daten extrahiert.

Wird eine nicht abgefangene Exception in der premain Methode geworfen, terminiert die virtual machine automatisch.

7 Zusammenfassung und Ausblick

Die JPPA besitzt viele gute Möglichkeiten um ein effizientes und gutes Profiling von Java-Code durchzuführen. Die Architektur ist im Gegensatz zum JVMPI sehr viel durchdachter. Ob die JPPA alle diese Features auch sinnvoll umsetzt, wird sich aber erst dann zeigen, wenn die an der Spezifikation beteiligten Unternehmen erste marktreife Implementierungen auf den Markt bringen. Ob die JPPA letztlich ihre Ziele konsequent umsetzen kann und damit die grundlegendes Probleme und Hindernisse beim Profiling von Java – Applikationen beseitigt, bleibt abzuwarten.

8 Literaturverzeichnis

Sun(Hrsg.): JSR 163 Java Platform Profiling Architecture, http://jcp.org/en/jsr/detail?id=163 ,2003