Variablen manipulieren per JDI

Zusammenfassung

Jede moderne Java IDE verfügt über eine mächtige und dennoch meist einfach zu bedienende Benutzeroberfläche die das finden von Fehlern in lokalen oder entfernt laufenden Anwendungen ermöglicht. Weshalb soll man sich überhaupt mit JDI, dem Java Debugger Interface auseinandersetzen?

Im konkreten Fall sollte eine Anwendung auf einem Server analysiert werden, und es war unmöglich eine zuverlässige TCP-Verbindung zwischen IDE und JVM Instanz aufzubauen. Ebenso war es nicht möglich die IDE auf dem Server selbst laufen zu lassen.

Die Lösung bestand darin, eine kurzes JDI basierendes und auf die konkrete Aufgabe zugeschnittenes Java-Programm zu schreiben und auf dem Server auszuführen. Zuvor wurde ein Test Programm geschrieben, um Erfahrungen mit JDI zu sammeln. Darüber wird hier berichtet.

Hinweis

Um die `com.sun.jdi` Klassen verwenden zu können muss `tools.jar` zum Klassenpfad hinzugefügt werden. Diese jar-Datei ist üblicherweise nur im JDK enthalten, jedoch nicht im JRE, und befindet sich meist auch nicht im Default-Klassenpfad.

Die JDI Bausteine

Verbinden mit der JVM

Der Anker ist die `Bootstrap` Klasse mit deren statischen Methode man Zugriff zu einem `VirtualMachineManager` erhält. Dieser wird benötigt um die Verbindungen zu einer JVM zu erstellen.

```
VirtualMachineManager vmm=Bootstrap.virtualMachineManager();
```

Als nächstes muss einen `Connector` gefunden werde, der geeignet ist, um sich mit JVM der Applikation zu verbinden. JDI unterscheidet zwischen drei Hauptarten.

Ein `LaunchingConnector` startet selbst die Anwendung und kontrolliert sie dann.

Ein `AttachingConnector` verbindet sich mit einer schon laufenden Anwendung.

Ein `ListeningConnector` erlaubt es, dass sich eine später gestartete Anwendung mit dem Debugger verbindet.

Zu jedem `Connector` kann es verschiedenen Transportmechanismen für die Verbindung zwischen Anwendung und Debugger geben.

In diesem Beispiel soll die JVM mit folgendem `agentlib` Argument gestartet werden.

```
-agentlib:jdwp=transport=dt_socket,server=y,address=8000,suspend=n
```

In der JDI Anwendung wird ein `AttachingConnector` mit Transport `dt_socket` benötigt um mit der JVM zu sprechen. Diese Methode sucht sich den passenden `Connector` und baut die Verbindung auf.

```
public static VirtualMachine socketAttach(final VirtualMachineManager vmm,final String host,
```

```
 final int port) throws IOException,IllegalConnectorArgumentsException {
 final Iterator<? extends AttachingConnector> it=vmm.attachingConnectors().iterator();
 AttachingConnector ac=null;
 while ((ac==null)&&it.hasNext()) {
 final AttachingConnector c=it.next();
 if (c.transport().name().equals("dt_socket")) {
 ac=c;
 }
 }
 if (ac==null) {
 throw new NullPointerException("No dt_socket connector found");
 }
 final Map<String,Argument> acArgs=ac.defaultArguments();
 acArgs.get("hostname").setValue(host);
 acArgs.get("port").setValue(Integer.toString(port));
 final VirtualMachine vm=ac.attach(acArgs);
 return vm;
}
```

Die Verbindung wird dann mit diesem Aufruf erstellt.

```
final VirtualMachine vm=socketAttach(Bootstrap.virtualMachineManager(),"localhost",8000);
```

Setzen eines Haltepunkt

Diese Methode sucht die gewünschte Klasse und darin den Ort der einer Zeile im Quellcode entspricht. Voraussetzung ist natürlich, dass die Klasse mit den nötigen Hilfsinformationen übersetzt wurde (`javac` Option `-g`).

```
public static void setBreakpoint(final VirtualMachine vm,final String className,
 final int lineNumber) throws AbsentInformationException {
 final List<ReferenceType> classes=vm.classesByName(className);
 assert classes.size()==1;
 final ReferenceType clas=classes.get(0);
 final List<Location> locations=clas.locationsOfLine(lineNumber);
 assert locations.size()==1;
 final BreakpointRequest
breakpoint=vm.eventRequestManager().createBreakpointRequest(locations.get(0));
 breakpoint.setSuspendPolicy(EventRequest.SUSPEND_ALL);
 breakpoint.enable();
 System.out.println(breakpoint);
}
```

Das setzen eines Haltepunkt geschieht dann ganz einfach mit

```
setBreakpoint(vm,"ch.inodes.examples.Counter",24);
```

Der Haltepunkt ist so gesetzt worden, dass die ganze JVM blockiert wird (`SUSPEND_ALL`) wenn er erreicht wird. Es wäre auch möglich nur den involvierten Thread anzuhalten.

Warten auf Haltepunkt

Zuletzt wartet man darauf, dass der Haltepunkt erreich wird, und modifiziert die gewünschten Variablen. Die `ThreadReference` wir immer wieder benötigt und deshalb in der Variable `thread` bereitgehalten.

```
final EventQueue eq=vm.eventQueue();
EventSet es;
try {
 es=eq.remove();
 for (final Event event:es) {
  if (event instanceof BreakpointEvent) {
   final BreakpointEvent be=(BreakpointEvent)event;
```

```
   final ThreadReference thread=be.thread();
   /*
    * Haltepunkt erreicht, hier können variable modifiziert werden
    */
   be.virtualMachine().resume();
  } else {
   System.out.println("UNEXPECTED "+event);
  }
 }
} catch (final Exception e) {
 e.printStackTrace();
 System.exit(1);
}
```

Hilfsmethoden für Zugriff auf Variablen

Hiermit erhält man eine Referenz auf `this` im Kontext des Haltepunkts.

```
public static ObjectReference getThis(final ThreadReference thread)
  throws IncompatibleThreadStateException {
 return thread.frame(0).thisObject();
}
```

Hat man eine Objektreferenz, z.B. mit `getThis()` erhalten, dann liefert die nachfolgenden Methoden den Wert eines Feldes davon.

```
public static Value readField(final ObjectReference obj,final String name) {
 return obj.getValue(obj.referenceType().fieldByName(name));
}
```

Ein Feld kann auch überschrieben werden, diese Methode zeigt wie das geschieht.

```
public static void writeField(final ObjectReference obj,final String name,final Value value)
 throws InvalidTypeException,ClassNotLoadedException {
 obj.setValue(obj.referenceType().fieldByName(name),value);
}
```

Der Zugriff auf lokale Variablen der Methode an der die JVM gehalten hat funktioniert etwas anders. Diese Methode zeigt, wie man eine solche Variable lesen kann.

```
public static Value readLocalVariable(final ThreadReference thread,final String name)
  throws IncompatibleThreadStateException,AbsentInformationException {
 final StackFrame stack=thread.frame(0);
 return stack.getValue(stack.visibleVariableByName(name));
}
```

Diese Methode zeigt wie man ihr einen neuen Wert gibt.

```
public static void writeLocalVariable(final ThreadReference thread,final String name,
 final Value value) throws IncompatibleThreadStateException,InvalidTypeException,
 ClassNotLoadedException,AbsentInformationException {
 final StackFrame stack=thread.frame(0);
 stack.setValue(stack.visibleVariableByName(name),value);
}
```

Schlussendlich ist es möglich eine Methode auszuführen.

```
public static Value invokeMethod(final ThreadReference thread,
 final ObjectReference objectInstance,final String name,final String signature,
 final Value... args) throws InvalidTypeException,ClassNotLoadedException,
 IncompatibleThreadStateException,InvocationException {
 return objectInstance.invokeMethod(thread,
((ClassType)objectInstance.referenceType()).concreteMethodByName(name,signature),Arrays.asList(arg
s),
```

```
 ObjectReference.INVOKE_SINGLE_THREADED);
}
```

Ebenso kann man eine neue Instanz eines Objektes erstellen. Hiermit wird ein Konstruktor aufgerufen.

```
public static Value invokeConstructor(final ThreadReference thread,final ClassType type,
  final String signature,final Value... args) throws InvalidTypeException,
  ClassNotLoadedException,IncompatibleThreadStateException,InvocationException {
 return type.newInstance(thread,type.concreteMethodByName("<init>",signature),Arrays.asList(args),
   ObjectReference.INVOKE_SINGLE_THREADED);
}
```

Diese Variante akzeptiert direkt eine Objektreferenz und ermittelt daraus selbst den Typ.

```
public static Value invokeConstructor(final ThreadReference thread,
 final ObjectReference objectInstance,final String signature,
 final Value... args) throws InvalidTypeException,ClassNotLoadedException,
 IncompatibleThreadStateException,InvocationException {
 return invokeConstructor(thread,(ClassType)objectInstance.referenceType(),signature,args);
}
```

Der Polymorphismus in Java erlaubt es mehrere Methoden mit demselben Namen zu haben. Deshalb muss die Signatur der Methode beim Aufruf mitgegeben werden. Tabelle 3-2 auf http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html beschreibt das Format. Ist man sich nicht sicher wie die Signatur genau auszusehen hat, dann kann man sich am einfachsten die Signaturen aller Methoden mit z.B. diesen Zeilen für `java.lang.String` ausgeben

```
for (Method m:vm.classesByName("java/lang/String").get(0).methods()) {
 System.out.println(m.name()+" "+m.signature());
}
```

Beispiele für Variablenzugriffe

Die Beispiele beziehen sich auf diese Test-Anwendung.

```
package ch.inodes.examples;

import java.awt.Point;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

/**
 * @author <a href='mailto:info@inodes.ch'>iNodes AG, 8048 Zürich</a>
 */
public class Counter {
 List<Date> list=new ArrayList<Date>();
 Date[] arrayD=new Date[2];
 int[] arrayI=new int[10];

 private void p() throws InterruptedException {
 int i=0;
 final Point p=new Point(1,2);
 final Date now=new Date();
 list.add(now);
 arrayD[1]=now;
 while (true) {
 i++;
 System.out.println(i+" "+list.get(0)+" "+arrayD[1]+" "+arrayI.length+" "+p);
 Thread.sleep(1000);
 }
 }
```

```
 public static void main(final String[] args) throws InterruptedException {
 final Counter c=new Counter();
 c.p();
 }
}
```

Die lokalen Variablen i und p sowie die Felder `list`, `arrayD` und `arrayI` sollen verändert werden, wenn die Zeile 24 mit der `System.out.println` Anweisung erreicht wird.

Anzeigen und ändern der lokalen Variable `i`. Das entspricht der Anweisung `System.out.println(i); i=-i;`

```
final IntegerValue i=(IntegerValue)readLocalVariable(thread,"i");
System.out.println(i);
writeLocalVariable(thread,"i",vm.mirrorOf(-i.intValue()));
```

Anzeigen und ändern eines Feldes eines lokalen Objekts. Das entspricht der Anweisung `System.out.println(p.y); p.y=p.y+1;`

```
final ObjectReference p=(ObjectReference)readLocalVariable(thread,"p");
final IntegerValue y=(IntegerValue)readField(p,"y");
System.out.println(y);
writeField(p,"y",vm.mirrorOf(y.intValue()+1));
```

Die folgenden Beispiele verändern die globale Felder. Dazu wird überall die spezielle Variable „`this`“ benötigt.

```
final ObjectReference thisAtBP=getThis(thread);
```

Das erste Element von `list` soll modifiziert werden, es soll das Äquivalent der Anweisung `System.out.println(list.get(0)); list.set(0,new Date());` ausgeführt werden.

```
final ObjectReference dateList=(ObjectReference)readField(thisAtBP,"list");
final ObjectReference
date=(ObjectReference)invokeMethod(thread,dateList,"get","(I)Ljava/lang/Object;",vm.mirrorOf(0));
System.out.println(((StringReference)invokeMethod(thread,date,"toString","()Ljava/lang/String;")).
value());
final ObjectReference newDate=(ObjectReference)invokeConstructor(thread,date,"()V");
invokeMethod(thread,dateList,"set","(ILjava/lang/Object;)Ljava/lang/Object;",vm.mirrorOf(0),newDat
e);
```

Die erste Zeile holt sich die Referenz auf die Liste. Die zweite ruft `get(0)` auf um auf das erste Element der Liste zuzugreifen. Die dritte Zeile ruft `toString()` auf um den Wert des `Date`-Elements zu erhalten und dann auszugeben. Die vierte Zeile ruft den Konstruktor von `Date` auf um eine neue Instanz zu erstellen. Die letzte Zeile speichert es in das erste Element der Liste.

Das nächste Beispiel zeigt das Ändern eines Array-Elementes, es wird die Anweisung `System.out.println(arrayD[1].toString()); arrayD[1]=newDate;` ausgeführt, wobei das Date-Element vom vorhergehenden Beispiel wiederverwendet wird.

```
final ArrayReference arrayD=(ArrayReference)readField(thisAtBP,"arrayD");
System.out.println(((StringReference)invokeMethod(thread,
(ObjectReference)arrayD.getValue(1),"toString","()Ljava/lang/String;")).value());
arrayD.setValue(1,newDate);
```

Zuletzt soll ein ganzes Array ersetzt werden. Dies entspricht den Anweisungen `System.out.println(arrayI.length()); arrayI=new int[arrayI.length()+1];`

```
final ArrayReference arrayI=(ArrayReference)readField(thisAtBP,"arrayI");
System.out.println(arrayI.length());
writeField(thisAtBP,"arrayI",((ArrayType)arrayI.referenceType()).newInstance(arrayI.length()+1));
```

Fügt man all diese Anweisungen aus nachdem der Haltepunkt erreich wurde, und startet dann den selbstgebauten „Debugger“ während dem die Test-Anwendung läuft, so kann man vom „Debugger“ eine Ausgabe wie diese beobachten.

```
9
2
Sun Feb 24 01:28:54 CET 2013
Sun Feb 24 01:28:54 CET 2013
10
```

Man sieht, dass die Werte von `i`, `p.y`, `list.get(0).toString()`, `arrayD.get(1).toString()` und `arrayI.length()` ausgegeben werden, so, wie sie die Test-Anwendung als nächstes im System.out schreiben würde, wenn man nichts tun würde.

Die Test-Anwendung dagegen wird dies ausgeben.

```
7 Sun Feb 24 01:28:54 CET 2013 Sun Feb 24 01:28:54 CET 2013 10 java.awt.Point[x=1,y=2]
8 Sun Feb 24 01:28:54 CET 2013 Sun Feb 24 01:28:54 CET 2013 10 java.awt.Point[x=1,y=2]
-9 Sun Feb 24 01:29:02 CET 2013 Sun Feb 24 01:29:02 CET 2013 11 java.awt.Point[x=1,y=3]
Listening for transport dt_socket at address: 8000
-8 Sun Feb 24 01:29:02 CET 2013 Sun Feb 24 01:29:02 CET 2013 11 java.awt.Point[x=1,y=3]
-7 Sun Feb 24 01:29:02 CET 2013 Sun Feb 24 01:29:02 CET 2013 11 java.awt.Point[x=1,y=3]
```

Nach der Ausgabe der Zeile die mit 8 beginnt, müsste eigentlich

```
9 Sun Feb 24 01:28:54 CET 2013 Sun Feb 24 01:28:54 CET 2013 10 java.awt.Point[x=1,y=2]
```

folgen. Doch das „Debugger“ Programm hat die Werte verändert. Es hat wie erwartet die lokale Variable `i` negiert, das Date-Element in `list` und `arrayD` ersetzt, `arrayI` ersetzt durch eines, dass ein Element länger ist, und den Wert von `p.y` um eins erhöht.

Die Zeile „Listening ...“ wird von der JVM ausgegeben, um anzuzeigen, dass der Debug-Port wieder frei ist, nachdem sich der „Debugger“ davon getrennt hat.