Java in Macro, SCL und Data Step

Carsten Zaddach
BDE Business Datawarehouse
Engineering GmbH
Landsberger Str. 218
12623 Berlin
cz@bde-gmbh.de

Zusammenfassung

Die Verwendung von Java-Klassen und Objekten sowie der direkte Aufruf von Methoden dieser Klassen und Objekte sind nur innerhalb eines Data Steps möglich. Durch die begrenzte Auswahl an möglichen Parametertypen (String und double) wird die Einsatzmöglichkeit von Java weiter eingeschränkt.
Durch die Verwendung eines entsprechenden Proxys und der SAS-Funktionen module(n,c) lassen sich diese Einschränkungen jedoch aufheben.

Schlüsselwörter: modulen, modulec, Java, Proxy, DLL, C++

1 Einführung

Die Verwendung von Java innerhalb eines SAS-Programms bietet die Möglichkeit, schnell und ohne großen Aufwand neue Funktionen hinzuzufügen. Standardmäßig erfolgt der Zugriff auf Java-Klassen und Objekte durch den Datentyp `JavaObj`. Die Verwendung dieses Datentyps ist jedoch nur innerhalb eines Data Steps möglich. Dies führt dazu, dass die Aufrufe von Methoden entsprechender Java-Objekte in SCL und Makro-Programmen in der Regel wie folgt aussehen:

```
data _null_;
     declare JavaObj obj;
     …
run;
```

Neben der Beschränkung auf Data Steps gibt es in den Augen des Autors noch weitere Einschränkungen, die einer optimalen Verwendung von Java innerhalb von SAS-Programmen entgegenstehen. Im weiteren Verlauf soll auf diese Einschränkungen näher eingegangen werden, da sie Motivation zur Realisierung dieses Projektes waren. Folgende Einschränkungen kristallisierten sich bei der täglichen Verwendung von Java in SAS heraus:

- Java nur innerhalb eines Data Steps verfügbar
- Lebenzzeit von Java-Objekten auf Laufzeit des Data Steps begrenzt
- Eingeschränkte Auswahl an Methodenaufrufen

Java nur innerhalb eines Data Steps: Der mit Version 9 eingeführte Datentyp `JavaObj` ist nur innerhalb eines Data Steps verfügbar. Um Methoden von Java-Klassen und Objekten auch in Makros und SCL-Programmen aufzurufen, muss die oben beschriebene „data-_null_-Krücke“ verwendet werden.

Lebenszeit von Java-Objekten: Ein weiterer großer Nachteil der aktuellen Java-Integration innerhalb von SAS ist die Lebenszeit der Java-Objekte. Diese beschränkt sich auf die Laufzeit des Data Steps, in der sie erstellt wurden. Somit können Java-Objekte nicht zum Austausch von Daten (z.B. Hashes, Schlüssel etc.) zwischen einzelnen Data Steps verwendet werden.

Eingeschränkte Auswahl an Methodenaufrufen: Alle numerischen Daten werden in SAS mit nur einem Datentyp abgebildet (8-Byte bzw. 64-Bit Gleitkommazahl). Dieser Datentyp entspricht in Java dem Typ double. Dadurch unterstützt `JavaObj` als Parameter der Java-Methoden nur die Typen `String` und `double`. Methoden, deren Parameter vom Typ `int, float, boolean` etc. sind, werden von SAS nicht gefunden und können somit nicht aufgerufen werden. Die einzige Möglichkeit, auch auf diese Methoden zuzugreifen ist die Verwendung eines individuellen Wrappers, der die double-Daten in den entsprechenden Datentyp castet und anschließend die entsprechende Methode aufruft.

Aufgrund dieser Einschränkungen wurde nach einer Möglichkeit gesucht, mit einfachen und den in SAS zur Verfügung stehenden Mitteln das Potenzial von Java auszuschöpfen.

2 Idee

Grundgedanke dieser Idee ist die Möglichkeit, Funktionen von externen Bibliotheken aus SAS heraus mittels `call module()` aufzurufen. Die Java Virtual Machine (JVM) wiederum stellt eine solche Bibliothek mit den entsprechenden Funktionen zur Verfügung.
Bei der Umsetzung der Idee traten jedoch noch einige Probleme und Hindernisse auf, die beseitigt werden mussten. Eines dieser Probleme war die Tatsache, dass die Erstellung und Initialisierung des Java Native Interface (JNI) zu komplex ist, um sie direkt mittels `call module()` aufrufen zu können. Der nachfolgende C++-Quelltext gibt auszugsweise einen Einblick darüber, wie eine Instanz vom JNI erstellt wird:

```
CreateJVMFunc CreateJVM = (CreateJVMFunc)GetProcAddress(jvmDll,
"JNI_CreateJavaVM");

JavaVMOption *options = new JavaVMOption[2];
size_t requiredSize;

std::string option = "-Djava.class.path=";
option.append(classpath);
```

```
options[0].optionString = _strdup(option.c_str());
options[1].optionString = "-verbose:jni";

odprintf("using classpath = %s", options[0].optionString);
vm_args.version = JNI_VERSION_1_6;
vm_args.nOptions = 2;
vm_args.options = options;
vm_args.ignoreUnrecognized = 0;

JNIEnv *jnienv;
jint rc = CreateJVM(&jvm, (void**)&jnienv, &vm_args);
```

Somit musste eine Art Proxy erstellt werden, der die komplexen Aufrufe der JVM-Bibliothek kapselt.

Das nächste Problem war die Lebensdauer des Proxys. Beim ersten Aufruf einer externen Funktion über `call module()` wird die entsprechende Bibliothek durch SAS geladen und nach Beendigung wieder entladen. Dies ist aber nicht gewünscht, da die Java-Klassen und Objekte über die Lebensdauer eines Data Steps hinaus verfügbar sein sollen. Um zu verhindern, dass die Bibliothek wieder automatisch entladen wird, wird sie einfach ein zweites Mal durch sich selbst geladen. Damit wird verhindert, dass der interne Betriebssystemzähler (zählt von wie vielen die Bibliothek verwendet wird) nach der Freigabe der Bibliothek durch SAS den Wert 0 annimmt und diese damit aus dem Speicher entfernt wird.

Da das Problem mit den beiden SAS-Datentypen weiterhin besteht, aber durch den Proxy ein größerer Umfang an Methoden aufgerufen werden soll, entstand die Idee, die Datentypen der Methoden beim Aufruf als Parameter mitzugeben. Anhand dieser Information kann der Proxy die gewünschte Methode ermitteln, die Datenkonvertierung vornehmen und anschließend die Methode aufrufen. Jeder Typ wird dabei durch ein eigenes Zeichen repräsentiert. Tabelle 1 zeigt die Datentypen und ihr entsprechendes Zeichen.

Tabelle 1: Java-Datentypen und ihre Codierung

Java-Datentyp	Codierung
boolean	Z
byte	B
char	C
double	D
float	F
integer	I
long	J
short	S
java.lang.String	X
Java object	O

3 Verwendung des Proxys

Die aktuelle Version des Proxys kann kostenfrei unter der URL:
https://www.bde-gmbh.de/proxy4jvm/
heruntergeladen werden. Wichtig ist darauf zu achten, dass sowohl SAS, Proxy als auch die verwendete Java-Runtime-Environment (JRE) die gleiche Architektur (32-oder 64-Bit) haben. Die Architektur des Betriebssystems spielt dabei aber keine Rolle.

3.1 Installation

Nach dem Entpacken des Archives muss der Pfad entweder zur Umgebungsvariable PATH hinzugefügt werden, oder die DLL wird in einen Pfad kopiert, der bereits in der PATH-Variablen vorhanden ist.
Die zweite Datei enthält die Informationen, welche Funktionen in welcher Bibliothek durch `call module()` aufgerufen werden sollen. Nachfolgend ist beispielhaft die Definition der Funktion `SetupJVMInstance` dargestellt.

```
routine SetupJVMInstance
  minarg=2
  maxarg=2
  module=Proxy4JVM;
  arg 1 input char format=$cstr1000.;
  arg 2 input char format=$cstr1000.;
```

Damit SAS diese Informationen finden und auswerten kann, muss diese Datei über ein Filename eingebunden werden. Der Name des Filenames muss zwingend `sascbtbl` sein. Um die JVM-Instance zu erstellen ist dann folgender Aufruf notwendig:

```
filename sascbtbl "path-to-sascbtbl/sascbtbl_arch.dat";
data _null_;
    call module("SetupJVMInstance", "path-to-jvm.dll\jvm.dll",
         "insert-your-classpath");
run;
```

Der entsprechende Aufruf mit SAS/Macro-Funktionen sieht wie folgt aus:

```
%let func=SetupJVMInstance;
%let jvm=%quote(path-to-jvm.dll\jvm.dll);
%let cp = insert-your-classpath;
%syscall module(func, jvm, cp);
```

Wird beim Setup kein Wert für den `classpath` angegeben, so wird die entsprechende Umgebungsvariable verwendet. Wird ebenfalls kein Pfad zur JVM angegeben, so sucht die Routine im aktuellen Pfad nach der Bibliothek.

3.2 Laden von Klassen

Damit Klassen verwendet werden können, müssen diese zuerst in die JVM geladen werden. Dabei spielt es keine Rolle, ob ein neues Objekt der Klasse erstellt oder eine statische Methode der Klasse aufgerufen werden soll. Um eine Klasse zu laden, muss folgendes aufgerufen werden:

```
data _null_;
     length classref 8;
     classref = modulen("LoadClass", "testClass");
run;
```

`testClass` ist der Name der Klasse und dieser wird im aktuellen Klassenpfad gesucht. `classref` ist die Referenz auf die geladene Klasse. Mit dieser Referenz kann auf statische Felder und Methoden der Klasse zugegriffen werden und es können damit neue Objekte der Klasse erstellt werden. Ein Wert von 0 in `classref` bedeutet, dass die entsprechende Klasse nicht gefunden wurde.

3.3 Zugriff auf statische und nicht-statische Felder

Für den Zugriff auf statische und nicht-statische Felder wird durch den Proxy eine Reihe von Methoden zur Verfügung gestellt. Dabei wird sowohl das Auslesen der Felder als auch das Modifizieren von Feldern unterstützt. Für numerische Felder stehen folgende Funktionen zur Verfügung

```
value = modulen("GetStatic***type***Field", classref, <Feldname>);
value = modulen("Get***type***Field", objectref, <Feldname>);
```

und für alphanumerische Felder (String und char) sind die Funktionen wie folgt:

```
value = modulec("GetStatic***type***Field", classref, <Feldname>);
value = modulec("Get***type***Field", objectref, <Feldname>);
```

<Feldname> ist der Name der Feldes, auf das in der Klasse oder im Objekt zugegriffen werden soll. Als ***type*** muss der Typ des Feldes angegeben werden, auf das zugegriffen werden soll. Folgende Typen stehen zur Verfügung: `Int, Long, Float, Double, Object, Short, Char, Boolean, String, Byte`.
Der folgende Code illustriert den Zugriff auf ein statisches Feld am Beispiel eines Integer-Feldes.

Der Java-Code sieht wie folgt aus:

```
public class testClass {

    public static int intValue = 1;
}
```

Und das SAS-Programm um auf das Feld `intValue` zuzugreifen sieht dann entsprechend so aus:

```
data _null_;
     length classref 8;

     classref = modulen("LoadClass", "testClass");
     value = modulen("GetStaticIntField", classref, "intValue");
run;
```

Entsprechend kann natürlich auch auf Felder von Objekten zugegriffen werden:

```
data _null_;
     . . .
     value = modulen("GetIntField", objref, "intValue");
run;
```

3.4 Aufruf von Methoden von Klassen und Objekten

In Java ist es möglich, dass mehrere Methoden mit dem gleichen Namen in einer Klasse existieren. Welche von diesen Methoden aufgerufen wird, entscheidet der Compiler anhand der Signaturen der Methoden und der Typen der Argumente, mit denen die Methode aufgerufen werden soll.
Da SAS aber nur zwei Typen kennt (String und Gleitkommazahl (double)), muss beim Aufruf einer Methode die Signatur der gewünschten Methode mit angegeben werden. Durch die Angabe der Signatur können fast alle möglichen Methoden einer Klasse aufgerufen werden. In Tabelle 1 sind die Java-Typen und ihre Codierung angegeben.

Um eine Methode aufzurufen, die einen numerischen Wert zurückliefert, ist folgender Aufruf erforderlich:

```
value = modulen("CallStatic type Method", classref, <Name der Methode>, Signature, <argument-1>,<argument-2>,… );
value = modulen("Call type Method", objectref, <Name der Methode>, Signature, <argument-1>,<argument-2>,… );
```

type gibt den Typ des Rückgabewertes der Methode an. Entsprechend ist der Aufruf für Methoden, die alphanumerische Werte zurückliefern:

```
value = modulec("CallStatic type Method", classref, <Name der Methode>, Signature, <argument-1>,<argument-2>,… );
value = modulec("Call type Method", objectref, <Name der Methode>, Signature, <argument-1>,<argument-2>,… );
```

Für Methoden, die keinen Wert zurückliefern (void), muss folgender Aufruf verwendet werden:

```
call module("CallStaticVoidMethod", classref, <Name der Methode>,
Signature, <argument-1>,<argument-2>,… );
call module("CallVoidMethod", objectref, <Name der Methode>,
Signature, <argument-1>,<argument-2>,… );
```

Das folgende Beispiel zeigt den Aufruf der statischen Methode `myIntMethod`. Die Methode ist in der Klasse `testClass` wie folgt definiert:

```
public static int myIntMethod(long arg1, boolean arg2, String arg3)
{
   // weiterer Code
}
```

Nach dem Laden der Klasse im Data-Step durch

```
classref = modulen("LoadClass", "testClass");
```

kann dann anschließend die Methode wie folgt aufgerufen werden:

```
value = modulen("CallStaticIntMethod", classref, "myIntMethod",
"JZX", 20, 1, "this is a string");
```

Die Signature für den Aufruf ist folgendermaßen codiert: J=long, Z=Boolean and X=String

Mit Verwendung des Proxys zum Aufruf von Methoden können auch Objekte als Argumente verwendet werden. Bei abgeleiteten Objekten sucht der Proxy nach der Klasse, die mit der Klasse der Methodensignatur in der Java-Klasse übereinstimmt.

3.5 Erzeugen von Java-Objekten

Die Erzeugung eines neuen Objektes erfolgt auf die gleiche Weise wie der Aufruf einer „normalen“ Methode. Auch bei der Instanziierung muss die gewünschte Signatur und die entsprechenden Werte mit übergeben werden. Für die Erstellung einer neuen Objektinstanz muss die Funktion `NewObject` aufgerufen werden. Der folgende Code zeigt den Aufruf der Methode:

```
objectref = modulen("NewObject", classref, signature, <argument-
1>,<argument-2>,… );
```

3.6 Einschränkungen

Zum Zeitpunkt der Erstellung dieses Artikels ist es nicht möglich, Arrays als Übergabeparameter zu verwenden. Wir hoffen aber, dieses Problem in absehbarer Zeit zu lösen und eine entsprechende Version bereitstellen zu können.