Java batch jobs on z/OS and OS/390

Steve Goetze
Kirk Wolf

March 10, 2005

Java™ has many uses on z/OS® and OS/390® outside the confines of a Web server environment. Kirk Wolf and Steve Goetze explain how you can effectively run Java programs in a z/OS or OS/390 batch environment.

Introduction

The IBM® Developer Kit (for OS/390, Java 2 Technology Edition, Version 1.3.1 and Software Development Kit (SDK) for z/OS, Version 1.4) provides the JVM and runtime environment for all Java-enabled subsystems on z/OS, including WebSphere® Application Server for z/OS, CICS®, DB2®, and Information Management System (IMS). The Java Developer Kit also provides the standard Java command, which can be run from a UNIX System Services shell environment.

The availability of a robust Java 2 Platform, Enterprise Edition (J2EE) environment, such as WebSphere Application Server for z/OS, is important. However, as Java technology continues to permeate the enterprise, the ability to seamlessly integrate Java into existing z/OS batch subsystems and schedules might be key to wide-scale adoption.

Here are some potential uses for Java in z/OS batch jobs:

- Passing datasets created by traditional job steps to Java programs, which convert the data to XML.
- Using Java programs to access APIs such as SOAP/Web services, WebSphere MQ client API, Java Database Connectivity (JDBC) databases, custom Transmission Control Protocol/ Internet Protocol (TCP/IP) socket services, and so forth.
- Employing long-running started tasks for Java that periodically query a database to find new work to process.
- Migrating systems written in COBOL to Java in order to reuse Java programming skills and class libraries.

Whereas in the past Java might have been considered slow or too resource intensive, the modern JVMs, with their just-in-time translation (JIT) to machine code, have excellent performance characteristics for many tasks. IBM has recently provided considerable financial incentives to using Java on z/OS with the announcement of the zSeries® Application Assist Processor (zAAP).

In this article, we describe some of the challenges you might encounter when using Java in batch jobstreams and how to seamlessly integrate Java into MVS Job Control Language (JCL). We refer to the operating system as z/OS, even though most of the information applies as well to OS/390, except that SDK 1.4 and above are only supported on z/OS.

Java batch job requirements

Before diving into some JCL for running Java in a z/OS batch job or started task, let's discuss some essential goals:

Flexible environment configuration
: You need a way to configure the environment variables and parameters that are required to invoke the JVM. As with other Java environments, the Java SDKs for z/OS require that directories and JAR files containing classes be listed in the CLASSPATH environment variable. These files are required to be in the Hierarchical File System (HFS) or the zSeries File System (zFS). Therefore, Java batch jobs actually straddle the fence, making extensive use of the UNIX System Services. Setting up and diagnosing problems in this environment can be painful, so you will want some flexibility in configuring the CLASSPATH and other environment variables that control how the JVM behaves.

Output routing
: This might seem too obvious to mention, but you would like to be able to send the output from your batch job to normal MVS datasets, including JES SYSOUT datasets. For Java, this would include the System.out and System.err output. As with any job, you would probably like to use tools like System Display and Search Facility (SDSF) to monitor the output of the job while it's running.

Control output encoding
: You also need to control the character encoding of your job's output, since you'll often run the JVM with ASCII as the default encoding. To understand this requirement, we first need to explain a little bit about encoding in Java: Internally, Java always runs with characters encoded in Unicode, but the *file.encoding* system property specifies the character set that is used when converting Java characters to and from bytes for input and output. On z/OS, the *LANG* environment variable determines the default encoding, which is normally *IBM-1047* (EBCDIC). Some Java applications, however, incorrectly assume that the default encoding is an ASCII code page such as *ISO-8859-1*. In order for these applications to run properly, you must set the *file.encoding* property to ISO-8859-1. If there were no way to separately control the encoding of a Java system output, the system output would also be written in ASCII. Therefore, you'll need a way to control the encoding of a job's output separately from the default JVM encoding.

Condition code passing
: It would be nice if Java job steps played nicely with other programs and utilities in the same job. Specifically, you would like to use the Java System.exit() method to terminate your Java program and pass the exit code as a condition code from the job step.

Use of MVS datasets and DD statements
: The standard *java.io* package only works with HFS or zFS files, but it would be nice to be able to use traditional MVS datasets, either by name or by using DD cards in the JCL. Having this

feature allows you to create and pass temporary datasets between Java and non-Java steps in the job.

Execute the JVM in the same address space

It is often surprising to new users of z/OS UNIX Systems Services to find that each new UNIX process can create a separate *OMVS* address space. This behavior often causes problems in the traditional MVS batch world. First of all, address spaces do not share DD names, SYSOUT files, and other dataset allocations. If the JVM is launched into a separate address space, the goals presented here can become difficult to achieve. Also, job monitoring and performance management can be tricky if multiple address spaces are used. Accounting data, such as Service Management Framework (SMF) type 30 records, will be created for each address space, which is often undesirable.

Communicating with the MVS job log and system console

It is common for batch jobs to use the write-to-operator (WTO) macro to write messages to the job log and system console. For long-running jobs and started tasks, the system operator might wish to gracefully stop the program using the MVS STOP (P) command, or send it arbitrary commands using the MODIFY (F) command.

Running Java batch jobs with BPXBATCH

You can use the BPXBATCH utility, part of Unix System Services, to launch any UNIX shell command, including the Java shell command that is part of the SDK. Listing 1 below illustrates a job that uses BPXBATCH to start a Java program and send its output to JES SYSOUT datasets:

Listing 1. Running Java batch jobs with BPXBATCH

```
//BPXBATCH JOB (999,XXX),'JAVA BPXBATCH',CLASS=A,MSGLEVEL=(1,1)
//   MSGCLASS=X,REGION=0M,NOTIFY=&SYSUID
//*****************************************************************
//* Run Java under a UNIX System Service shell
//*****************************************************************
//STEP2 EXEC PGM=BPXBATCH,
// PARM='SH java com.foo.MyClass arg1 arg2'
//STDIN  DD DUMMY
//STDOUT DD PATH='/tmp/&SYSUID..bpxbatch.out',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU
//STDERR DD PATH='/tmp/&SYSUID..bpxbatch.err',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU
//STDENV DD *
CLASSPATH=/u/myuid/classes
//*****************************************************************
//* Copy HFS output files to SYSOUT, since BPXBATCH can only write
//* STDOUT and STDERR to HFS files.
//*****************************************************************
//STEP3 EXEC PGM=IKJEFT01,DYNAMNBR=300,COND=EVEN
//SYSTSPRT DD SYSOUT=*
//HFSOUT DD PATH='/tmp/&SYSUID..bpxbatch.out'
//HFSERR DD PATH='/tmp/&SYSUID..bpxbatcherr'
//STDOUTL DD SYSOUT=*,DCB=(RECFM=VB,LRECL=133,BLKSIZE=137)
//STDERRL DD SYSOUT=*,DCB=(RECFM=VB,LRECL=133,BLKSIZE=137)
//SYSPRINT DD SYSOUT=*
//SYSTSIN DD *
OCOPY INDD(HFSOUT) OUTDD(STDOUTL)
OCOPY INDD(HFSERR) OUTDD(STDERRL)
//
```

So, how well does BPXBATCH meet the stated requirements?

- **Flexible environment configuration**: Environment variables, such as CLASSPATH, can be specified in the user's default *.profile* login script or by using the //STDENV DD. The //STDENV dataset, however, does not allow for variable substitution or scripting of any kind. See Environment variables used by Java for more information.
- **Output routing**: Java System.out and System.err streams are sent to //STDOUT and //STDERR. BPXBATCH only supports HFS files for these DDs, so a separate step is required to copy these files to SYSOUT datasets. This also means that you cannot view the output until the job completes.
- **Control output encoding**: Default output encoding cannot be controlled separately from the default *file.encoding* system property, so a separate step is required to do conversion.
- **Condition code passing**: You cannot pass Java condition codes using System.exit() to subsequent steps.
- **Use of MVS datasets and DD statements**: The Java Record IO (JRIO) package, included with the IBM Developer Kit for the Java Platform allows for record I/O access. Although initially supporting only EBCDIC code pages, since 4Q 2003, JRIO supports all code pages that are supported by z/OS. JRIO supports converting dsname and pathnames but customer data, including keys, are not translated. In order to use job step DD statements, you must launch the JVM in the same address space (see below). (See Reading and Writing MVS datasets for more information.)
- **Execute the JVM in the same address space**: By default, BPXBATCH starts the Java JVM in a separate address space, so that the Java program does not have access to DD dataset allocations in the job step.
  You can use a special version (entry point) to the BPXBATCH utility, BPXBATSL, to cause the shell to spawn in the same address space to overcome this problem. BPXBATSL has a limitation that prevents it from running a logon shell unless run as root, so it must be executed as follows:
  ```
  //STEP2 EXEC PGM=BPXBATSL,
  // PARM='PGM /bin/sh -c java com.foo.MyClass arg1 arg2'
  ```
  This also requires that your //STDENV includes settings for all environment variables, since /etc/profile and .profile will not be run. For this reason, it's often better to execute a shell script to handle this rather than invoking Java directly.
- **Communicating with the job log and system console**: No facility exists in BPXBATCH or the Java SDK to write WTO messages or respond to operator STOP or MODIFY commands. A Java program could make a JNI call to a C program which can then call the _console or _console2 C functions to issue WTOs or interact with STOP and MODIFY. See Communication with the MVS system console for more information.

Running Java batch jobs with a custom JVM launcher

Rather than executing Java as a shell command under UNIX System Services, a custom Java VM launcher can be written using the Java Native Interface (JNI) launcher interface. Listing 2 demonstrates a JCL using the free JZOS Batch Launcher, which is a custom JVM launcher for z/OS:

Listing 2. Running Java batch jobs with a custom JVM launcher

```
//JZOSBAT JOB (999,XXX),'JAVA JZOS',CLASS=A,MSGLEVEL=(1,1)
//   MSGCLASS=X,REGION=0M,NOTIFY=&SYSUID
//JAVAJVM  EXEC PGM=JZOSVM14,
//   PARM='com.foo.MyClass arg1 arg2'
//STEPLIB  DD DSN=JZOS.LIBRARY,DISP=SHR
//SYSPRINT DD SYSOUT=*          < System stdout
//SYSOUT   DD SYSOUT=*          < System stderr
//STDOUT   DD SYSOUT=*          < Java System.out
//STDERR   DD SYSOUT=*          < Java System.err
//STDENV   DD *
. /etc/profile
. ~/.profile
export CLASSPATH=~/myapp
for i in ~/myapp/lib/*.jar; do
    export CLASSPATH=$i:$CLASSPATH
    done
//
```

The JZOS Batch Launcher, not accidentally, does better with the stated requirements:

- **Flexible environment configuration**: The //STDENV file points to a UNIX shell script, which exports any environment variables that it wishes to set. The ability to run a shell script, which itself might call other programs or shell scripts, allows for flexibility and reuse when configuring the environment. In the example above, new JARs added to the lib directory would automatically be added to the CLASSPATH.
  Note: By design, the JZOS Batch Launcher doesn't automatically run a logon shell, so if you want /etc/profile and the user's login profile to run, you have to explicitly execute them. See Environment variables used by Java for more information.
- **Output routing**: STDOUT and STDERR files might route directly to JES SYSOUT datasets (or MVS datasets). You can monitor the output while the job is running.
- **Control output encoding**: Output encoding defaults to the codepage implied by the "LANG" environment variable, regardless of the default file.encoding system property. If the JZOS_OUTPUT_ENCODING environment variable is defined, its value is used to change the default output encoding.
- **Condition code passing**: Java condition codes are passed from the System.exit() to subsequent steps.
- **Use of MVS datasets and DD statements**: The JZOS ZFile class provides a flexible JNI wrapper for the standard C library I/O routines. See Reading and writing MVS datasets for more information.
- **Execute the JVM in the same address space**: The JZOS VM launcher invokes the JVM under the original address space, so that the Java program does have access to DD dataset allocations in the job step. Since a secondary address space is not used, the Java program runs under the original WLM group.
- **Communicating with the job log and system console**: The ZOS ZUtil class provides methods for writing console messages and responding to operator commands. See Communication with the MVS system console for more information.

Environment variables used by Java

The following are environment variables commonly used in Java applications:

- **PATH**: List of directories, separated by a colon (":"), used to look up executable binaries. This should include $JAVA_HOME/bin.
- **LIBPATH**: List of directories, separated by a colon, used to look up shared libraries. This should include any libraries required by any JNI libraries used by your application. When running the JZOS Launcher, this must include:
  ```
  $JAVA_HOME/bin:$JAVA_HOME/bin/classic:$JZOS_HOME
  ```
- JZOS Batch Launcher **CLASSPATH**: List of directories and JAR/ZIP files, separated by a colon, from which to load Java classes. This should include the directories or individual JAR files used by your Java application. When running the JZOS Batch Launcher, this must also include $JZOS_HOME/jzos.jar.
- **IBM_JAVA_OPTIONS**: Space-separated list of options to the JVM. Type `Java -help` from a z/OS or OMVS shell to display a list of options. For example:
  ```
  IBM_JAVA_OPTIONS="-Xms64m -Xmx128m -Djzos.home=/u/jzos"
  ```

Reading and writing MVS datasets from Java

You can use the normal java.io package on z/OS to read and write HFS (UNIX) files, but not MVS datasets. The JRIO package, included with the IBM SDK for z/OS, Java 2 Technology Edition, provides the ability to work with records. As with any of the IBM Java products, no source code is available.

As an alternative for using JRIO, the JZOS toolkit (containing the JZOS ZFile class) includes:

- A JNI wrapper for the z/OS C file IO APIs (fopen, fclose, fread, fwrite, and so forth).
- Support for all IO models provided by the C library (both "record" and "stream" modes). See z/OS C/C++ Programming Guide for more information.
- Java source code.

Listing 3 is an example of reading and writing MVS datasets in record mode:

Listing 3. Reading and writing MVS datasets in record mode

```
ZFile inZFile = new ZFile("//DD:INPUT", "rb,type=record,noseek");
ZFile outZFile = new ZFile("//DD:OUTPUT", "wb,type=record,noseek");
try {
    byte[] recBuf = new byte[inZFile.getLrecl()];
    int nRead = 0;
    while((nRead = inZFile.read(recBuf)) > 0) {
        outZFile.write(recBuf, 0, nRead);
    }
} finally {
   inZFile.close();
   outZFile.close();
}
```

Listing 4 is an example of reading and writing MVS datasets in stream mode:

Listing 4. Reading and writing MVS datasets in stream mode

```
ZFile inZFile = new ZFile("//DD:INPUT", "rt");
ZFile outZFile = new ZFile("//DD:OUTPUT", "wt");
```

```
    BufferedReader brdr;
    BufferedWriter bwtr;
    try {
        InputStream istream = inZFile.getInputStream();
        InputStreamReader rdr =
             new InputStreamReader(istream, ZFile.DEFAULT_EBCDIC_CODE_PAGE);
        brdr = new BufferedReader(irdr);

        OutputStream ostream = outZFile.getOutputStream();
        OutputStreamWriter wtr =
             new OutputStreamWriter(ostream, ZFile.DEFAULT_EBCDIC_CODE_PAGE);
        bwtr = new BufferedWriter(bwtr);

        String line;
        while ((line = brdr.readLine()) != null) {
            bwtr.write(line);
            bwtr.newLine();
        }
    } finally {
        if (brdr != null) brdr.close();
        if (bwtr != null) bwtr.close();
    }
```

Since the ZFile class is only available when running on z/OS or OS/390, it's often desirable to write code that only uses ZFile classes when using MVS datasets and to use normal java.io classes in other cases. The JZOS toolkit provides the FileFactory class for just that purpose, as shown in Listing 5 below.

Listing 5. FileFactory class

```
    BufferedReader brdr = FileFactory.newBufferedReader(inputFileName);
    BufferedWriter bwtr = FileFactory.newBufferedWriter(outputFileName);
    try {
        String line;
        while ((line = brdr.readLine()) != null) {
            bwtr.write(line);
            bwtr.newLine();
        }
    } finally {
        if (brdr != null) brdr.close();
        if (bwtr != null) bwtr.close();
    }
```

In the above example, the inputFileName and outputFileName variables might be configured at runtime, depending on the target platform. If the file names start with "//", then the FileFactory class uses a ZFile in stream mode, otherwise a normal java.io class will be used. This allows code to be developed and tested on your workstation and later deployed to the mainframe.

For more information, see JZOS ZFile class API reference.

Communicating with the MVS system console

It is common for batch jobs to display messages to the MVS system console. The JZOS toolkit also has this feature.

```
  ZUtil.wto("FOO1233E processing terminated",
              0x0020,   // routecde
              0x4000);  // descriptor code
```

If the message doesn't fit into a single-line WTO (limited to 125 characters), it's broken on word boundaries into a multi-line WTO.

Also, it's sometimes desirable to control batch jobs using MVS operator commands. The JZOS Batch Launcher allows a batch Java application to respond the following console commands:

- **START**: If a Java job runs as an MVS started task, it might access options specified on this command.
- **STOP (P)**: This command causes the JZOS Batch Launcher to issue a Java System.exit(), which calls any Java shutdown hooks and terminates the JVM.
- **MODIFY (F)**: The JZOS Batch Launcher allows Java applications to register a hook to intercept MVS MODIFY commands.

For more information, see the JZOS ZUtil class API reference.

Summary: Comparison of tools for launching batch Java jobs

	BPXBATCH	BPXBATSL	Custom JVM launcher (JZOS)
Flexible configuration of environment variables	**Yes, variable substitution and scripting are not allowed**	**Yes, variable substitution and scripting are not allowed**	**Yes**
Route output directory to SYSOUT datasets	No	No	Yes
Control output encoding separately from default JVM encoding	Yes, using iconv	Yes, using iconv	Yes
Condition-code passing between Java and non-Java steps	No	No	Yes
Use MVS datasets and DD statements	No	Yes	Yes
JVM runs in same address space	No	Yes	Yes
Communication with MVS console	No	Yes, using _console function from a JNI routine	Yes

Java is present on the mainframe in many forms -- most visibly in WebSphere -- but also playing an important role for products like CICS, DB2, and IMS. Until recently, however, Java has not been very visible in the batch environment, arguably z/OS's most ubiquitous workload type. With the recent zAPP announcement, expect that to change.

Related topics

- IBM Redbook Integrating Java with Existing Data and Applications on OS/390, SG24-5142-00 by Alex Louwe Kooijmans gives you a comprehensive overview of how to get started with Java in an OS/390 Server environment.
- Visit the JZOS site and get information on a set of free tools (JZOS batch JVM launcher and toolkit) designed to make Java easier to use on the mainframe.
- Learn more about the zSeries Application Assist Processor (zAAP).
- For more information on IBM WebSphere Application Server for z/OS, visit the z/OS main page.
- Get the z/OS UNIX System Services users guide: Using BPXBATCH.
- Learn more about the Java Record IO package, which is part of the IBM SDK for z/OS, Java 2 Technology Edition.
- The z/OS C/C++ Programming Guide provides information about implementing programs that are written in C and C++.
- Sun documentation of Java JNI Launcher interface provides a programming interface for writing Java native methods.
- Learn how to write a JNI program on OS/390.