Principles of Java Batch

Draft Edition 1.0

Disclaimer

This document should in no way be construed as a draft of the JSR 352 specification.

Purpose of this Document

The purpose of this document is to provide a conceptual framework and basic objectives for developing a batch application specification under JSR 352, Batch Applications for the Java Platform.

Audience

The principle audience of this document is the JSR 352 Expert Group, although observers of this JSR may also find the content of this document interesting.

Objectives

This documents intends to convey basic principles of batch processing and offer preliminary ideas on how to organize, represent, and expose those principles in the context of the Java Platform.

This document will be treated as a work in progress and will be extended and revised to facilitate the work of the JSR 352 Expert Group at the group's discretion.

Introduction

Batch processing is a pervasive workload pattern, expressed by a distinct application organization and execution model. It is found across virtually every industry, applied to such tasks as statement generation, bank postings, risk evaluation, credit score calculation, inventory management, portfolio optimization, and on and on. Nearly any bulk processing task from any business sector is a candidate for batch processing.

Batch processing is typified by bulk-oriented, non-interactive, background execution. Frequently long-running, it may be data or computationally intensive, execute sequentially or parallel, and may be initiated through various invocation models, including ad hoc, scheduled, and on-demand.

Batch applications have common requirements, including logging, checkpoint, and parallelization. Batch workloads have common requirements, especially operational control, which allow for initiation of, and interaction with, batch instances; such interactions include stop and restart.

Batch Concepts

Batch Application Anatomy

Batch applications implement business and data logic. The smallest unit of business logic is called a 'step'. Good architecture dictates a distinct separation between business and data logic: a step uses a 'reader' and 'writer' to access batch data. A step is the basic unit of composition and reuse. A batch 'job' is composed of one or more steps. A job is the basic unit of batch execution. A job describes a sequence of steps, that taken together, accomplish a particular business goal. Different jobs may reuse one or more steps in different sequences to accomplish different business goals. So a batch application is a collection of steps, their readers and writers, and one or more jobs that specify particular step sequences. The relationships among these application artifacts are illustrated in the following diagram:

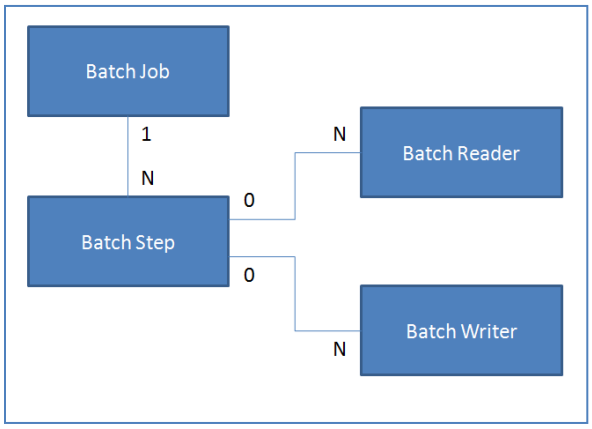

Diagram 1. Relationships among application artifacts

Batch Runtime Anatomy

A good architecture addresses separation of concerns. Batch applications should concern themselves with domain-specific business function; generic concerns such as logging, checkpoints, parallel task management, and operational control (e.g. commands) should be delivered by a batch runtime ('container') that supplies these capabilities to all batch applications. This approach supports the basic notion that an external agent (i.e. a user, an application, etc) can submit a job to a runtime that can then run that job, manage logging, checkpoints, and parallelization as required by the job. Each job instance run by the container can be thought of as a "Batch Execution", to give it a name. The relationships among these runtime artifacts are illustrated in the following diagram:

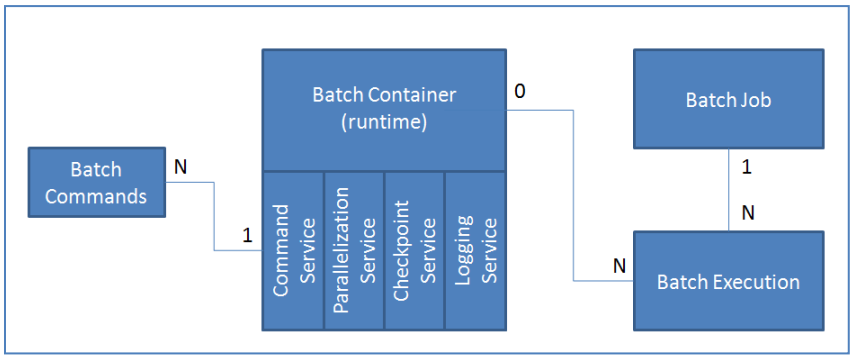

Diagram 2. Relationship among runtime artifacts

The idea is that the Batch Container supports a plug-in architecture that allows the various services to be plugged in, thus allowing adaptation of the container to various runtime environments.

Batch Checkpoints

For data intensive batch applications - particularly those that may run for long periods of time - checkpoint/restart is a common design requirement. Checkpoints allow a batch execution to periodically bookmark its current progress to enable restart from the last point of consistency, following a planned or unplanned interruption.

An effective batch runtime should support checkpoint/restart in a generic way that can be exploited by any batch step that has this requirement.

Since progress during a batch execution is really a function of the current position of the input/output data, natural placement of function suggests the knowledge for saving/restoring current position is a reader/writer responsibility.

Since managing batch execution is a container responsibility, the batch container must necessarily understand batch execution lifecycle, including initial start, execution end states, and restart.

Since checkpoint frequency has a direct effect on lock hold times, for lockable resources, tuning checkpoint interval size can have a direct bearing on overall system throughput. Ideally, this should be adjustable operationally and not explicitly controlled by the batch application itself.

All of this implies a relationship between the batch container and the batch application for the purpose of performing checkpoints. The following diagram depicts the essential flow among this relationship:

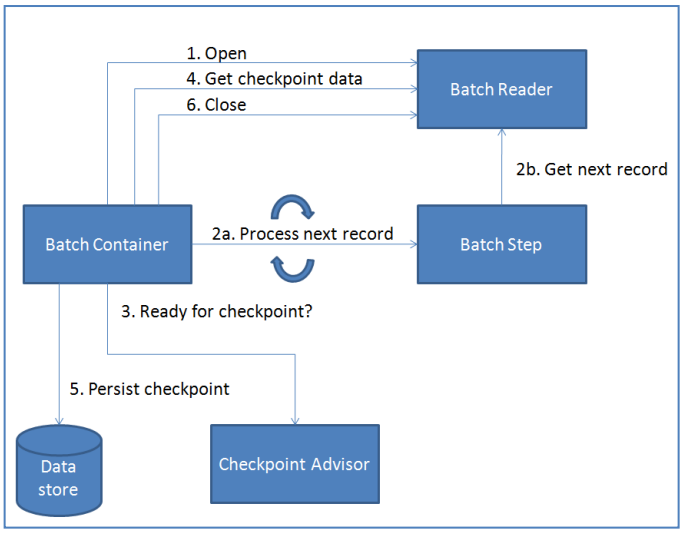

Diagram 3. Checkpoint flow between container and application

In diagram 3, the illustrated flow depicts the container responsibility for interacting with the application to support checkpoint. The essential steps of the flow are:

1. open reader

   This gives an opportunity to the reader to perform any sort of first time processing. E.g. open a file, issue an SQL SELECT statement, etc.

2. batch loop

   The batch loop processes a number of input records by the container repeatedly invoking it to process the next record until the next checkpoint interval is reached. After a checkpoint is taken, the batch loop continues. This pattern continues until all records are processed.

   a. process next record

      This is where the batch step business logic resides. It processes the next logical record and then returns to the container.

   b. get next record

      The batch step directly invokes the batch reader to acquire the next record to process. A batch step should be able to read from one or more readers in order to acquire the data it requires.

3. check if ready for checkpoint

   After each logical record is processed, the container would interrogate an application (or runtime) supplied algorithm to determine if it is time to checkpoint the current batch execution. The decision to take a checkpoint could be affected by any number of factors including: number of records processed, amount of time elapsed, etc.

4. get checkpoint data

   Once time to take a checkpoint, the container queries the reader to obtain the reader's current position.

5. persist checkpoint data

   The container would then persist the checkpoint data in a runtime-managed store.

6. close

   After all records are processed, the container calls the reader for close processing.

Supporting checkpoints allows the container to also support restart. The restart use case demands a job be restartable after a planned or unplanned outage. An effective batch runtime is able to support this and could do so with a flow typified by the following diagram:

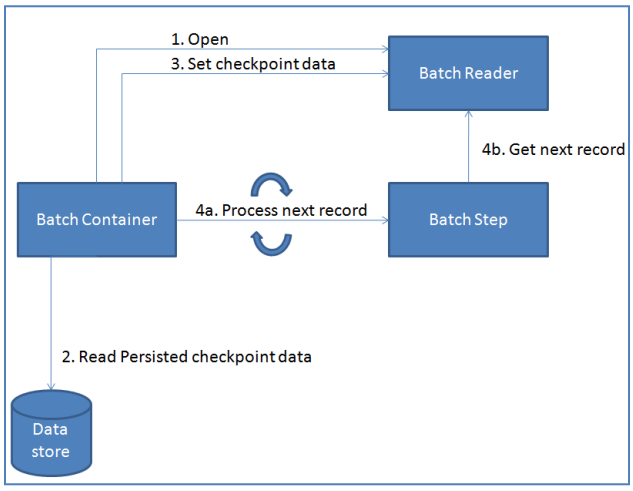

Diagram 4. Container and application flow during restart

The preceding diagram is similar to diagram 3 except that since this depicts restart processing, the container reads the data from the last checkpoint and passes it to the reader after open processing so the reader can reposition itself to the last record that was successfully processed.

Note the flows described for the batch reader are applicable also to the batch writer. Moreover, nothing should preclude a batch step from using multiple batch readers and/or batch writers.

Other Types of Batch Steps

While data intensive batch steps are common. Other types of batch steps also exist:

1. computationally intensive

   This is a long running step type that does not require checkpoint services. Typically, such a batch step does minimal data access relative to the amount of computation.

2. batch utilities

   This is a batch step that run a variable amount of time. It may be a short running or longer running batch step. Any of a variety of different utility functions could fall into this category: background FTP, data compression or other actions, execution of a shell script, and so forth.

3. agent or daemon

   This is a batch step that run an indefinite period of time. It may provide an agent or daemon sort of function. It may expose a private command interface to facilitate its control.

An effective batch programming model and runtime should accommodate a wide range of step types. It should be possible to mix any batch step types together in the same batch job.

Batch Job Identity

A batch job should have a name. This name need not be unique. A batch execution should have a unique identifier - e.g. a sequence number. This identifier must be unique among all batch executions.

Batch Step End States

It should be possible for a batch step to indicate to the batch container a variety of end states, such as: normal, restartable, failed.

Batch Step Return Codes

In addition to an end state, a batch step should be able to specify an application-specific return code. It should be possible within a job step for batch step execution to be conditioned upon the return codes of other batch steps within the same batch job, which has already run.

Error Handling

The batch container should handle any exception that is not handled by a batch step and assign an appropriate end state. It is useful for a batch container to offer skip and retry capabilities to a batch step such that common errors can be configured and handled by the batch container.

Transaction Management

It should be possible for a batch step - particularly, a data intensive step that requires checkpoints - to select between JTA global and resource manager local transactions. For checkpoints, it should be possible for transactional work done by the batch step to be rolled back to the most recent checkpoint if the batch step ends prematurely due to planned or unplanned reasons.

Transaction management should be a pluggable service of the batch container. See 'Pluggable Runtime' topic in the 'Design Principles' section.

Job Logs and Logging

It is highly effective for problem determination and other operational needs for a batch runtime to provide a distinct log for each batch execution. Such a log is commonly referred to as a "job log". Batch steps should be able to use java.util.logging to write to a job log in a natural way.

The backend side of the logging should be a pluggable service of the batch container so that different technologies are possible for storage mediums. See 'Pluggable Runtime' topic in the 'Design Principles' section.

Usage Accounting

An effective batch runtime should be able to calculate the amount of elapsed and/or CPU time a batch step consumes and be able to expose it to various interfaces. An approach for this could be a System Programming Interface (SPI) that allows a customized handler to be added to the batch container to interpose on job and step lifecycle events. See 'Pluggable Runtime' topic in the 'Design Principles' section.

Persistence

The batch runtime should be capable of maintaining job state, including running jobs, restartable jobs, and jobs in final state.

The specific persistence mechanism used by a batch container should be pluggable. See 'Pluggable Runtime' topic in the 'Design Principles' section.

Security

The batch runtime should address three distinct security aspects:

1. authentication

   It should be possible for a batch container running in a secure environment to require its invokers to be authenticated. It is reasonable for a batch container to depend on its hosting environment to provide authentication services.

2. authorization

   It should be possible for a batch container running in a secure environment to discriminate among its invokers with regard to actions they are able to perform. it is reasonable for a batch container to depend on its hosting environment to provide access control or role based authorization services.

3. execution context

   It should be possible for a batch container running in a secure environment to impose different "run as" policies, choosing, for example, between running a batch job under "submitter" identity or system identity. It is reasonable for a batch container to depend on its hosting environment to provide "run as" services.

Provisions for security employed by the batch container should be pluggable. See 'Pluggable Runtime' topic in the 'Design Principles' section.

Parallel processing

A powerful batch runtime should provide for parallel execution. This may come in several forms:

1. parallel steps

   A parallel step is simply a step that runs on multiple threads in the same process or across multiple processes. This technique is typically employed in divide and conquer schemes.

2. concurrent steps

   Concurrent steps within a job are steps that run simultaneously on separate threads in the same process or multiple processes. This technique is typically employed in consumer/producer schemes.

3. parallel jobs

   A parallel job is a variation of a parallel in step in that an entire multi-step job runs in parallel.

As the method of distributing work to other processes (machine images) may vary based on the environment hosting a batch container, the means for doing this should be through a pluggable service of the batch container. See 'Pluggable Runtime' topic in the 'Design Principles' section.

Commands

A batch container should provide a set of operational commands that allow manipulation of batch jobs, including, but not limited to:

1. submit - initiates a new batch execution
2. cancel - terminates a particular batch execution
3. restart - re-initiates a particular batch execution
4. getJobReturnCode - obtain return code from a particular batch execution
5. getJobLog - obtain the job log from a particular batch execution

Notification

A batch container should provide a mechanism for communicating job and step lifecycle events to some form of listener. Notification are most likely delivered asynchronously. Notifications should be provided for the following events:

1. job start
2. step start
3. step end
4. job end

Java SE

A batch container should accommodate usage in a Java SE environment. This implies that the core implementation requires no services outside of Java SE.

Java EE

A batch container should accommodate usage in a Java EE environment. This implies a layer around the core implementation that exposes the batch container to the Java EE environment in an appropriate manner. For example, that may include providing a reference to a batch container through JNDI and exposing batch container commands through remote interfaces. The following diagram depicts this concept:

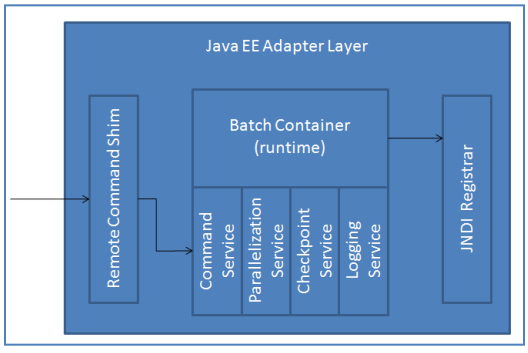

Diagram 5. Java EE Layer

Design Principles

Annotations and Interfaces

Java interfaces are a traditional basis for prescribing new programming models. More recently annotations provide a flexible and attractive mechanism for defining programming models. A recommendation for a Java batch programming is to employ a combination:

- annotations

  Annotations are a good choice for defining programming artifacts provided by the application developer.

- interfaces

  Interfaces provide a natural way for a batch container to expose optional runtime services to an application.

Static and Dynamic Jobs

A straight forward programming model based on only Java and annotations implies that job definitions are static, defined in code. This is a natural and reasonable starting point. A more powerful batch container would support a job control language (e.g. JCL, JSDL, etc).

Since a follow-on JSR for batch scheduling was proposed by this very JSR, it may be prudent to defer specification of a "JCL" to such a follow-on JSR. However, it would be helpful for this JSR to give some consideration to the implications of a future dynamic job definition ability during the course of defining the base specification represented by this JSR.

Pluggable Runtime

A powerful batch container would be adaptable to multiple execution environments. A well defined services layer for key quality of services duties performed by the container would serve as the basis for this adaptability.

Services, and a mechanism for wiring them into the container, should be specified for the following functions:

1. configuration
2. transaction control
3. persistence
4. logging

5. parallel distribution

Appendix - Proposed Annotations

Simple Batch Step with Checkpointing Example

```
package jsr352.test;

import javax.batch.annotation.*;

@Step(name="Checkpoint",txMode=TxMode.GLOBAL)
@Properties({
      @Property(name="inputJNDIName",value=""),
      @Property(name="outputFileName",value="")
})
@Checkpoint(policy=CheckpointPolicy.RECORDs,count=1000)
public class CheckpointStep {

      @Reader(name="input") BatchJDBCReader inputRecords;
      @Writer(name="output") BatchFileWriter outputRecords;

      @BeginStep
      public void init(boolean isRestart) {
      }

      @RunStep
      public StepDirective processNextRecord () {
            BatchRecord record= inputRecords.readNext();
            if (record != null ) {
                  outputRecords.writeNext(record);
                  return StepDirective.CONTINUE;
            }
            else return StepDirective.END;
      }

      @GetReturnCode
      public  int getReturnCode() {
            return 0;
      }

      @EndStep
      public void cleanup() { }
}
```

Simple Batch Step without Checkpointing Example

```
package jsr352.test;

import javax.batch.annotation.*;

@Step(name="NoCheckpoint")
@Properties({
      @Property(name="inputFileName",value=""),
      @Property(name="outputFileName",value="")
})
public class NoCheckpointStep {

      @Reader(name="input") BatchFileReader inputRecords;
      @Writer(name="output") BatchFileWriter outputRecords;

      @BeginStep
      public void init(boolean isRestart) {
      }

      @RunStep
      public StepDirective doFileMerge () {
            return StepDirective.END;
      }

      @GetReturnCode
      public  int getReturnCode() {
            return 0;
      }

      @EndStep
      public void cleanup() { }
```

Simple Multi-step Job

```
package jsr352.test;

import javax.batch.annotation.*;

@Job(name="TestAnnotations",txMode=TxMode.GLOBAL)
public class TestAnnotationsJob {

	// Job Steps ...
	@Step(name="Step1",txMode=TxMode.GLOBAL) Step1 _step1;
	@Step(name="Step2",parallel=true) Step2 _step2;
	@Step(name="Step3",concurrencyGroup="concurrent") Step3 _step3;
	@Step(name="Step4",concurrencyGroup="concurrent") Step4 _step4;
	@Step(name="Step5") Step5 _step5;

	@BeginJob
	public void init(boolean isRestart) {
	}

	@BeginStep
	public void beginStep(String stepName) { }

	@EndStep
	public void endStep(String stepName, int stepRC) {
	}

	@GetReturnCode
	public  int getReturnCode() {
		return 0;
	}

	@EndJob(txMode=TxMode.GLOBAL)
	public void sendNotification() { }
}
```

Readers and Writers

Reader Example

```
package jsr352.test;

import javax.batch.annotation.*;

public class BatchFileReader {

	public BatchFileReader(String fileName) {}

	@Open
	public void open() {
		// file open logic here
	}

	@Close
	public void close() {
		// file close logic here
	}

	@GetCheckpoint
	public String getCheckpointData() {
		return null; // return current file position
	}

	@SetCheckpoint
	public void setCheckpointData(String lastPosition) {
		// move file pointer to last position
	}

	// Return next record or null for EOF
	public BatchRecord readNext() { return null; }
}
```

Writer Example

```
package jsr352.test;

import javax.batch.annotation.*;

public class BatchFileWriter {

	public BatchFileWriter(String fileName) {}

	@Open
	public void open() {
		// file open logic here
	}

	@Close
	public void close() {
		// file close logic here
	}

	public void writeNext(BatchRecord record) { }

}
```