Table of Contents

Chapter 1. Repeat

1.1. RepeatTemplate

Batch processing is about repetitive actions, either as a simple optimization or as part of a job. To strategize and generalize the repetition and to provide what amounts to an iterator framework, Spring Batch has the `RepeatOperations` interface. The `RepeatOperations` interface has the following definition:

```
public interface RepeatOperations {

    RepeatStatus iterate(RepeatCallback callback) throws RepeatException;

}
```

The callback is an interface, shown in the following definition, that lets you insert some business logic to be repeated:

```
public interface RepeatCallback {

    RepeatStatus doInIteration(RepeatContext context) throws Exception;

}
```

The callback is executed repeatedly until the implementation determines that the iteration should end. The return value in these interfaces is an enumeration that can either be `RepeatStatus.CONTINUABLE` or `RepeatStatus.FINISHED`. A `RepeatStatus` enumeration conveys information to the caller of the repeat operations about whether there is any more work to do. Generally speaking, implementations of `RepeatOperations` should inspect the `RepeatStatus` and use it as part of the decision to end the iteration. Any callback that wishes to signal to the caller that there is no more work to do can return `RepeatStatus.FINISHED`.

The simplest general purpose implementation of `RepeatOperations` is `RepeatTemplate`, as shown in the following example:

```
RepeatTemplate template = new RepeatTemplate();

template.setCompletionPolicy(new SimpleCompletionPolicy(2));

template.iterate(new RepeatCallback() {

    public RepeatStatus doInIteration(RepeatContext context) {
        // Do stuff in batch...
        return RepeatStatus.CONTINUABLE;
    }

});
```

In the preceding example, we return `RepeatStatus.CONTINUABLE`, to show that there is more work to do. The callback can also return `RepeatStatus.FINISHED`, to signal to the caller that there is no more work to do. Some iterations can be terminated by considerations intrinsic to the work being done in the callback. Others are effectively infinite loops as far as the callback is concerned and the completion decision is delegated to an external policy, as in the case shown in the preceding example.

1.1.1. RepeatContext

The method parameter for the `RepeatCallback` is a `RepeatContext`. Many callbacks ignore the context. However, if necessary, it can be used as an attribute bag to store transient data for the duration of the iteration. After the `iterate` method returns, the context no longer exists.

If there is a nested iteration in progress, a `RepeatContext` has a parent context. The parent context is occasionally useful for storing data that need to be shared between calls to `iterate`. This is the case, for instance, if you want to count the number of occurrences of an event in the iteration and remember it across subsequent calls.

1.1.2. RepeatStatus

`RepeatStatus` is an enumeration used by Spring Batch to indicate whether processing has finished. It has two possible `RepeatStatus` values, described in the following table:

Table 1. RepeatStatus Properties

Value	*Description*
CONTINUABLE	There is more work to do.
FINISHED	No more repetitions should take place.

`RepeatStatus` values can also be combined with a logical AND operation by using the `and()` method in `RepeatStatus`. The effect of this is to do a logical AND on the continuable flag. In other words, if either status is `FINISHED`, then the result is `FINISHED`.

1.2. Completion Policies

Inside a `RepeatTemplate`, the termination of the loop in the `iterate` method is determined by a `CompletionPolicy`, which is also a factory for the `RepeatContext`. The `RepeatTemplate` has the responsibility to use the current policy to create a `RepeatContext` and pass that in to the `RepeatCallback` at every stage in the iteration. After a callback completes its `doInIteration`, the `RepeatTemplate` has to make a call to the `CompletionPolicy` to ask it to update its state (which will be stored in the `RepeatContext`). Then it asks the policy if the iteration is complete.

Spring Batch provides some simple general purpose implementations of `CompletionPolicy`. `SimpleCompletionPolicy` allows execution up to a fixed number of times (with `RepeatStatus.FINISHED` forcing early completion at any time).

Users might need to implement their own completion policies for more complicated decisions. For example, a batch processing window that prevents batch jobs from executing once the online systems are in use would require a custom policy.

1.3. Exception Handling

If there is an exception thrown inside a `RepeatCallback`, the `RepeatTemplate` consults an `ExceptionHandler`, which can decide whether or not to re-throw the exception.

The following listing shows the `ExceptionHandler` interface definition:

```
public interface ExceptionHandler {

    void handleException(RepeatContext context, Throwable throwable)
        throws Throwable;

}
```

A common use case is to count the number of exceptions of a given type and fail when a limit is reached. For this purpose, Spring Batch provides the `SimpleLimitExceptionHandler` and a slightly more flexible `RethrowOnThresholdExceptionHandler`. The `SimpleLimitExceptionHandler` has a limit property and an exception type that should be compared with the current exception. All subclasses of the provided type are also counted. Exceptions of the given type are ignored until the limit is reached, and then they are rethrown. Exceptions of other types are always rethrown.

An important optional property of the `SimpleLimitExceptionHandler` is the boolean flag called `useParent`. It is `false` by default, so the limit is only accounted for in the current `RepeatContext`. When set to `true`, the limit is kept across sibling contexts in a nested iteration (such as a set of chunks inside a step).

1.4. Listeners

Often, it is useful to be able to receive additional callbacks for cross-cutting concerns across a number of different iterations. For this purpose, Spring Batch provides the `RepeatListener` interface.

The `RepeatTemplate` lets users register `RepeatListener` implementations, and they are given callbacks with the `RepeatContext` and `RepeatStatus` where available during the iteration.

The `RepeatListener` interface has the following definition:

```
public interface RepeatListener {
    void before(RepeatContext context);
    void after(RepeatContext context, RepeatStatus result);
    void open(RepeatContext context);
    void onError(RepeatContext context, Throwable e);
    void close(RepeatContext context);
}
```

The `open` and `close` callbacks come before and after the entire iteration. `before`, `after`, and `onError` apply to the individual `RepeatCallback` calls.

Note that, when there is more than one listener, they are in a list, so there is an order. In this case, `open` and `before` are called in the same order while `after`, `onError`, and `close` are called in reverse order.

1.5. Parallel Processing

Implementations of `RepeatOperations` are not restricted to executing the callback sequentially. It is quite important that some implementations are able to execute their callbacks in parallel. To this end, Spring Batch provides the `TaskExecutorRepeatTemplate`, which uses the Spring `TaskExecutor` strategy to run the `RepeatCallback`. The default is to use a `SynchronousTaskExecutor`, which has the effect of executing the whole iteration in the same thread (the same as a normal `RepeatTemplate`).

1.6. Declarative Iteration

Sometimes there is some business processing that you know you want to repeat every time it happens. The classic example of this is the optimization of a message pipeline. It is more efficient to process a batch of messages, if they are arriving frequently, than to bear the cost of a separate transaction for every message. Spring Batch provides an AOP interceptor that wraps a method call in a `RepeatOperations` object for just this purpose. The `RepeatOperationsInterceptor` executes the intercepted method and repeats according to the `CompletionPolicy` in the provided `RepeatTemplate`.

The following example shows declarative iteration using the Spring AOP namespace to repeat a service call to a method called `processMessage` (for more detail on how to configure AOP interceptors, see the Spring User Guide):

```
<aop:config>
    <aop:pointcut id="transactional"
        expression="execution(* com..*Service.processMessage(..))" />
    <aop:advisor pointcut-ref="transactional"
        advice-ref="retryAdvice" order="-1"/>
</aop:config>

<bean id="retryAdvice" class="org.spr...RepeatOperationsInterceptor"/>
```

The following example demonstrates using java configuration to repeat a service call to a method called `processMessage` (for more detail on how to configure AOP interceptors, see the Spring User Guide):

```
@Bean
public MyService myService() {
    ProxyFactory factory = new ProxyFactory(RepeatOperations.class.getClassLoader());
    factory.setInterfaces(MyService.class);
    factory.setTarget(new MyService());

    MyService service = (MyService) factory.getProxy();
    JdkRegexpMethodPointcut pointcut = new JdkRegexpMethodPointcut();
    pointcut.setPatterns(".*processMessage.*");

    RepeatOperationsInterceptor interceptor = new RepeatOperationsInterceptor();

    ((Advised) service).addAdvisor(new DefaultPointcutAdvisor(pointcut, interceptor));

    return service;
}
```

The preceding example uses a default `RepeatTemplate` inside the interceptor. To change the policies, listeners, and other details, you can inject an instance of `RepeatTemplate` into the interceptor.

If the intercepted method returns `void`, then the interceptor always returns `RepeatStatus.CONTINUABLE` (so there is a danger of an infinite loop if the `CompletionPolicy` does not have a finite end point). Otherwise, it returns `RepeatStatus.CONTINUABLE` until the return value from the intercepted method is `null`, at which point it returns `RepeatStatus.FINISHED`. Consequently, the business logic inside the target method can signal that there is no more work to do by returning `null` or by throwing an exception that is re-thrown by the `ExceptionHandler` in the provided `RepeatTemplate`.