Table of Contents

Chapter 1. Retry

To make processing more robust and less prone to failure, it sometimes helps to automatically retry a failed operation in case it might succeed on a subsequent attempt. Errors that are susceptible to intermittent failure are often transient in nature. Examples include remote calls to a web service that fails because of a network glitch or a `DeadlockLoserDataAccessException` in a database update.

1.1. `RetryTemplate`

The retry functionality was pulled out of Spring Batch as of 2.2.0. It is now part of a new library, Spring Retry.

To automate retry operations Spring Batch has the `RetryOperations` strategy. The following interface definition for `RetryOperations`:

```
public interface RetryOperations {

    <T, E extends Throwable> T execute(RetryCallback<T, E> retryCallback) throws E;

    <T, E extends Throwable> T execute(RetryCallback<T, E> retryCallback,
RecoveryCallback<T> recoveryCallback)
        throws E;

    <T, E extends Throwable> T execute(RetryCallback<T, E> retryCallback, RetryState
retryState)
        throws E, ExhaustedRetryException;

    <T, E extends Throwable> T execute(RetryCallback<T, E> retryCallback,
RecoveryCallback<T> recoveryCallback,
        RetryState retryState) throws E;

}
```

The basic callback is a simple interface that lets you insert some business logic to be retried, as shown in the following interface definition:

```
public interface RetryCallback<T, E extends Throwable> {

    T doWithRetry(RetryContext context) throws E;

}
```

The callback runs and, if it fails (by throwing an `Exception`), it is retried until either it is successful or the implementation aborts. There are a number of overloaded `execute` methods in the `RetryOperations` interface. Those methods deal with various use cases for recovery when all retry attempts are exhausted and deal with retry state, which allows clients and implementations to store

information between calls (we cover this in more detail later in the chapter).

The simplest general purpose implementation of `RetryOperations` is `RetryTemplate`. It can be used as follows:

```
RetryTemplate template = new RetryTemplate();

TimeoutRetryPolicy policy = new TimeoutRetryPolicy();
policy.setTimeout(30000L);

template.setRetryPolicy(policy);

Foo result = template.execute(new RetryCallback<Foo>() {

    public Foo doWithRetry(RetryContext context) {
        // Do stuff that might fail, e.g. webservice operation
        return result;
    }

});
```

In the preceding example, we make a web service call and return the result to the user. If that call fails, then it is retried until a timeout is reached.

1.1.1. `RetryContext`

The method parameter for the `RetryCallback` is a `RetryContext`. Many callbacks ignore the context, but, if necessary, it can be used as an attribute bag to store data for the duration of the iteration.

A `RetryContext` has a parent context if there is a nested retry in progress in the same thread. The parent context is occasionally useful for storing data that need to be shared between calls to `execute`.

1.1.2. `RecoveryCallback`

When a retry is exhausted, the `RetryOperations` can pass control to a different callback, called the `RecoveryCallback`. To use this feature, clients pass in the callbacks together to the same method, as shown in the following example:

```
Foo foo = template.execute(new RetryCallback<Foo>() {
    public Foo doWithRetry(RetryContext context) {
        // business logic here
    },
  new RecoveryCallback<Foo>() {
    Foo recover(RetryContext context) throws Exception {
          // recover logic here
    }
});
```

If the business logic does not succeed before the template decides to abort, then the client is given the chance to do some alternate processing through the recovery callback.

1.1.3. Stateless Retry

In the simplest case, a retry is just a while loop. The `RetryTemplate` can just keep trying until it either succeeds or fails. The `RetryContext` contains some state to determine whether to retry or abort, but this state is on the stack and there is no need to store it anywhere globally, so we call this stateless retry. The distinction between stateless and stateful retry is contained in the implementation of the `RetryPolicy` (the `RetryTemplate` can handle both). In a stateless retry, the retry callback is always executed in the same thread it was on when it failed.

1.1.4. Stateful Retry

Where the failure has caused a transactional resource to become invalid, there are some special considerations. This does not apply to a simple remote call because there is no transactional resource (usually), but it does sometimes apply to a database update, especially when using Hibernate. In this case it only makes sense to re-throw the exception that called the failure immediately, so that the transaction can roll back and we can start a new, valid transaction.

In cases involving transactions, a stateless retry is not good enough, because the re-throw and roll back necessarily involve leaving the `RetryOperations.execute()` method and potentially losing the context that was on the stack. To avoid losing it we have to introduce a storage strategy to lift it off the stack and put it (at a minimum) in heap storage. For this purpose, Spring Batch provides a storage strategy called `RetryContextCache`, which can be injected into the `RetryTemplate`. The default implementation of the `RetryContextCache` is in memory, using a simple `Map`. Advanced usage with multiple processes in a clustered environment might also consider implementing the `RetryContextCache` with a cluster cache of some sort (however, even in a clustered environment, this might be overkill).

Part of the responsibility of the `RetryOperations` is to recognize the failed operations when they come back in a new execution (and usually wrapped in a new transaction). To facilitate this, Spring Batch provides the `RetryState` abstraction. This works in conjunction with a special `execute` methods in the `RetryOperations` interface.

The way the failed operations are recognized is by identifying the state across multiple invocations of the retry. To identify the state, the user can provide a `RetryState` object that is responsible for returning a unique key identifying the item. The identifier is used as a key in the `RetryContextCache` interface.

Be very careful with the implementation of `Object.equals()` and `Object.hashCode()` in the key returned by `RetryState`. The best advice is to use a business key to identify the items. In the case of a JMS message, the message ID can be used.

When the retry is exhausted, there is also the option to handle the failed item in a different way, instead of calling the `RetryCallback` (which is now presumed to be likely to fail). Just like in the stateless case, this option is provided by the `RecoveryCallback`, which can be provided by passing it in to the `execute` method of `RetryOperations`.

The decision to retry or not is actually delegated to a regular `RetryPolicy`, so the usual concerns about limits and timeouts can be injected there (described later in this chapter).

1.2. Retry Policies

Inside a `RetryTemplate`, the decision to retry or fail in the `execute` method is determined by a `RetryPolicy`, which is also a factory for the `RetryContext`. The `RetryTemplate` has the responsibility to use the current policy to create a `RetryContext` and pass that in to the `RetryCallback` at every attempt. After a callback fails, the `RetryTemplate` has to make a call to the `RetryPolicy` to ask it to update its state (which is stored in the `RetryContext`) and then asks the policy if another attempt can be made. If another attempt cannot be made (such as when a limit is reached or a timeout is detected) then the policy is also responsible for handling the exhausted state. Simple implementations throw `RetryExhaustedException`, which causes any enclosing transaction to be rolled back. More sophisticated implementations might attempt to take some recovery action, in which case the transaction can remain intact.

Failures are inherently either retryable or not. If the same exception is always going to be thrown from the business logic, it does no good to retry it. So do not retry on all exception types. Rather, try to focus on only those exceptions that you expect to be retryable. It is not usually harmful to the business logic to retry more aggressively, but it is wasteful, because, if a failure is deterministic, you spend time retrying something that you know in advance is fatal.

Spring Batch provides some simple general purpose implementations of stateless `RetryPolicy`, such as `SimpleRetryPolicy` and `TimeoutRetryPolicy` (used in the preceding example).

The `SimpleRetryPolicy` allows a retry on any of a named list of exception types, up to a fixed number of times. It also has a list of "fatal" exceptions that should never be retried, and this list overrides the retryable list so that it can be used to give finer control over the retry behavior, as shown in the following example:

```
SimpleRetryPolicy policy = new SimpleRetryPolicy();
// Set the max retry attempts
policy.setMaxAttempts(5);
// Retry on all exceptions (this is the default)
policy.setRetryableExceptions(new Class[] {Exception.class});
// ... but never retry IllegalStateException
policy.setFatalExceptions(new Class[] {IllegalStateException.class});

// Use the policy...
RetryTemplate template = new RetryTemplate();
template.setRetryPolicy(policy);
template.execute(new RetryCallback<Foo>() {
    public Foo doWithRetry(RetryContext context) {
        // business logic here
    }
});
```

There is also a more flexible implementation called `ExceptionClassifierRetryPolicy`, which allows the user to configure different retry behavior for an arbitrary set of exception types though the `ExceptionClassifier` abstraction. The policy works by calling on the classifier to convert an exception into a delegate `RetryPolicy`. For example, one exception type can be retried more times before failure than another by mapping it to a different policy.

Users might need to implement their own retry policies for more customized decisions. For instance, a custom retry policy makes sense when there is a well-known, solution-specific classification of exceptions into retryable and not retryable.

1.3. Backoff Policies

When retrying after a transient failure, it often helps to wait a bit before trying again, because usually the failure is caused by some problem that can only be resolved by waiting. If a `RetryCallback` fails, the `RetryTemplate` can pause execution according to the `BackoffPolicy`.

The following code shows the interface definition for the `BackOffPolicy` interface:

```
public interface BackoffPolicy {

    BackOffContext start(RetryContext context);

    void backOff(BackOffContext backOffContext)
        throws BackOffInterruptedException;

}
```

A `BackoffPolicy` is free to implement the backOff in any way it chooses. The policies provided by Spring Batch out of the box all use `Object.wait()`. A common use case is to backoff with an exponentially increasing wait period, to avoid two retries getting into lock step and both failing (this is a lesson learned from ethernet). For this purpose, Spring Batch provides the `ExponentialBackoffPolicy`.

1.4. Listeners

Often, it is useful to be able to receive additional callbacks for cross cutting concerns across a number of different retries. For this purpose, Spring Batch provides the `RetryListener` interface. The `RetryTemplate` lets users register `RetryListeners`, and they are given callbacks with `RetryContext` and `Throwable` where available during the iteration.

The following code shows the interface definition for `RetryListener`:

```
public interface RetryListener {

    <T, E extends Throwable> boolean open(RetryContext context, RetryCallback<T, E>
callback);

    <T, E extends Throwable> void onError(RetryContext context, RetryCallback<T, E>
callback, Throwable throwable);

    <T, E extends Throwable> void close(RetryContext context, RetryCallback<T, E>
callback, Throwable throwable);
}
```

The `open` and `close` callbacks come before and after the entire retry in the simplest case, and `onError` applies to the individual `RetryCallback` calls. The `close` method might also receive a `Throwable`. If there has been an error, it is the last one thrown by the `RetryCallback`.

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

1.5. Declarative Retry

Sometimes, there is some business processing that you know you want to retry every time it happens. The classic example of this is the remote service call. Spring Batch provides an AOP interceptor that wraps a method call in a `RetryOperations` implementation for just this purpose. The `RetryOperationsInterceptor` executes the intercepted method and retries on failure according to the `RetryPolicy` in the provided `RetryTemplate`.

The following example shows a declarative retry that uses the Spring AOP namespace to retry a service call to a method called `remoteCall` (for more detail on how to configure AOP interceptors, see the Spring User Guide):

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

<bean id="retryAdvice"
    class="org.springframework.batch.retry.interceptor.RetryOperationsInterceptor"/>
```

The following example shows a declarative retry that uses java configuration to retry a service call to a method called `remoteCall` (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(".*remoteCall.*");

    RetryOperationsInterceptor interceptor = new RetryOperationsInterceptor();

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

    return service;
}
```

The preceding example uses a default `RetryTemplate` inside the interceptor. To change the policies or listeners, you can inject an instance of `RetryTemplate` into the interceptor.