**9.1 重试模板**

**请注意** 这个重试功能在Spring Batch 2.2.0里面退出，现在它是Spring Retry的一部分.

为了让这个进程更稳定，更小的失败性。有时它帮助自动重试一个失败的操作以防止它可能在后续的尝试成功。本质上，这种处理会导致误差。例如，远程调用网络服务或RMI服务失败是由于在短暂的数据更新后，网络故障或冻结异常aDeadLockLoserException.例如重试这种自动化操作，Spring Batch有重试操作的策略。 重试操作接口如下:

public interface RetryOperations {

<T> T execute(RetryCallback<T> retryCallback) throws Exception;

<T> T execute(RetryCallback<T> retryCallback, RecoveryCallback<T> recoveryCallback)

throws Exception;

<T> T execute(RetryCallback<T> retryCallback, RetryState retryState)

throws Exception, ExhaustedRetryException;

<T> T execute(RetryCallback<T> retryCallback, RecoveryCallback<T> recoveryCallback,

RetryState retryState) throws Exception;

}

基本回调是一个简单的接口,允许您插入一些业务逻辑重试：

public interface RetryCallback<T> {

T doWithRetry(RetryContext context) throws Throwable;

}

执行回调如果失败了（抛出异常），它会重试直至成功或者执行决定终止。在重试界面，有一些超载的操作方法，专门处理恢复各种用户实例。也可以在重试阶段,允许客户端和实现信息存储之间的调用（后面详细讨论)

最简单通用的实施RetryOperations就是Retry Template。就像这样

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;

}

});

在这个例子中我们执行web服务调用和返回结果给用户。如果调用失败然后重试,直到超时。

**9.1.1 重试上下文**

RetryCallback的方法参数是RetryContext.许多回调会简单地忽略上下文,但必要时它可以作为一个属性包为迭代的持续时间存储数据。

在同一线程中,如果一个嵌套在重试过程中, RetryContext会有一个母本。这个母本偶尔在存储数据用于调用和执行共享很有帮助

**9.1.2 恢复回调**

当一个重试耗尽这个RetryOperations可以传递控制一个不同的回调，是恢复回调。要使用该功能的客户只是通过同样的方法回调在一起,例如:

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

}

});

在这个模板决定终止之前如果业务逻辑没有执行成功，然后客户端有机会做一些交替处理通过恢复回调处理

**9.1.3 无状态的重试**

在一个简单的实例中，重试只是一个while循环，RetryTemplate可以不断尝试,直到成功或失败，RetryContext包含一些状态来确定是否重试或中止，但这种状态在堆栈上没有必要在任何时候地方存储，所以我们称之为无状态的重试，无状态和有状态重试之间的区别是包含在实施RetryPolicy中(RetryTemplate可以同时处理)，在一个无状态的重试,执行回调总是在同一线程上耗尽而失败

**9.1.4 状态性重试**

有一些需要特别考虑的是,当失败引起事务资源无效，这并不适用于一个简单的远程调用,因为没有事务资源(通常)，但它有时适用于数据库更新，尤其是使用Hibernate，这个案例中，我们重新抛出这个异常，我们称之为立即失效，这样的话，事物资源可以回滚，我们可以启用一个新的。

在这些情况下无状态重试还不够好，因为re-throw和回滚必然涉及离开RetryOperations.execute()方法和潜在损失的上下文堆栈。为了避免失去它我们必须引入存储策略提升了堆栈并把它(至少)放在堆存储中，为此Spring Batch提供了一个存储策略RetryContextCache可以注入RetryTemplate。内存中有RetryContextCache默认的实现,使用一个简单的Map。多个进程的高级用法在集群环境中也会考虑实现集群缓存的RetryContextCache(不过,在集群环境中这可能是过度)。

RetryOperations的责任之一就是在一个执行新任务时候识别错误操作(通常是包裹在一个新的事务)。为了使它更方便,Spring Batch提供抽象的RetryState。RetryOperations结合特殊执行操作方法

识别错误的操作方法是通过识别跨多个调用的重试。辨别出这个状态,用户可以提供RetryState对象返回一个唯一键识别项。标识符用作RetryContextCache的一个关键。

**警告** 实现Object的equals方法和hashCode方法要非常小心，关键是在返回重试状态。最好的建议是使用一个业务主键来标识这个项目，在JMS 消息的message Id可以使用的情况下。

当重试停止也有选择以不同的方式处理失败的项，而不是调用RetryCallback(假定现在可能失败)。就像在无状态的情况下，这个选项是RecoveryCallback提供的,也可以用越过RetryOperations的执行方法提供。

重试与否实际上是委托给一个普通的RetryPolicy,所以通常的使自己关心的限制和超时是可以注入的(见下文)。

**9.2 重试策略**

在RetryTemplate里面，执行excuter方法是重试还是失败是由RetryPolicy决定的,这也是一个RetryContext工厂.这个RetryTemplate有责任使用当前的策略创建一个RetryContext并且把它注入到RetryCallback在每一次尝试中。回调失败后RetryTemplate必须由RetryPolicy决定使其更新状态(存储在RetryContext中)，然后它询问策略是否会作出其他决定。如果不能作出其他决定,（例如达到极限或超时），政策有责任处理这个状态。简单的操作将会抛出异常它会引起封闭事务的回滚。更多更复杂的操作可能会采取一些回复操作，在这种情况下，事务可以保持不变。

**提示** 失败是固有的,无论重试与否，如果在业务逻辑中总是抛出相同的异常。这不能帮助重试它。所以不要重试所有异常类型,试着只关注那些你期望可能重试的异常，它虽然对业务逻辑重试变得更具侵略性,但是它没有危害性。如果你事先知道它会失败还去花些时间来证明，只是浪费时间。

Spring Batch提供给无状态RetryPolicy一些简单通用的实现,例如SimpleRetryPolicy,和在上面的示例中使用的TimeoutRetryPolicy。

SimpleRetryPolicy只允许重试一些异常类型的命名列表，它也有一个“fatal”的异常列表,不应重试.并且这个列表重写了retryable的列表，以便它可以用来很好地控制重试行为:

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

}

});

还有一个更灵活的实现叫ExceptionClassifierRetryPolicy。它允许用户配置不同的重试行为为了一组固定的异常类型集合，虽然ExceptionClassifier抽象类型，分类器上的政策是通过调用异常转换为一个委托RetryPolicy,举个例子,一个异常类型能比映射不通的策略重试更多的次数。

用户需要为更多的客户定制的决定实施他们自己重试策略。例如，如果有一个清楚的详细的解决方案，是归类到可以重试还是不可以重试

**9.3 补偿策略**

在瞬时失效之后做一个尝试它常常有助于在再试一次之前等待一会。因为通常失败是有一些问题引起的它只能通过等待来解决这些问题，如果RetryCallback失败,RetryTemplate可以暂停执行取决于BackoffPolicy在适当的位置

public interface BackoffPolicy {

BackOffContext start(RetryContext context);

void backOff(BackOffContext backOffContext)

throws BackOffInterruptedException;

}

backoffPolicy可以不受约束的以任何选择的方式实现backoff。spring batch 这个政策创造性的使用了object.wait()。通常情况下等待期间以指数方式上升。避免两个重试锁定步骤和失败。这是网上学到的，为了这个目的spring batch提供了ExponentialBackoffPolicy.

**9.4 监听器**

常常它是有用的能够接受附加的回调为了切割关注点穿过一些不同的重试 为了这个目的Spring Batch 提供了RetryListene 接口，RetryTemplate 允许使用者注册RetryListene，并且他们将发送回调随从RetryContext和Throwable,在迭代期间可用。

这个接口看起来像这样:

public interface RetryListener {

void open(RetryContext context, RetryCallback<T> callback);

void onError(RetryContext context, RetryCallback<T> callback, Throwable e);

void close(RetryContext context, RetryCallback<T> callback, Throwable e);

}

open 和 close 回调在整个的重试前后调用在简单的实例和onError 适用于个体的RetryCallback调用，close方法可能也需要接收一个Throwable.如果已经有一个错误它是最后一个在RetryCallback抛出。

注意当有多余一个监听的时候，它们是在一个列表。所以它们是顺序的，在这种情况下，open方法将被顺序访问执行，onError 和 close方法被倒序执行

**9.5 声明式重试**

有时一些业务逻辑的发生你每次都想重试它。最经典的例子就是远程调用，为了这个目的Spring Batch提供了一个Aop拦截器用来封装一个方法调用RetryOperations 。RetryOperationsInterceptor依据RetryPolicy 提供的RepeatTemplate 执行拦截方法和重试失败。

下面是一个示例使用Spring AOP命名空间的声明式迭代重复一个服务调用的一个方法称为remoteCall (AOP如何配置拦截器,更多细节见Spring用户指南):

<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"/>

在上面的示例中在拦截器内使用了一个默认的RetryTemplate 。改变策略和监听，你只需要在拦截器中注入一个RetryTemplate 实例。