Spring Retry

快速学习手册

May 14, 2019

liuliu

什么时候用可以重试

- 远程调用失败的可以重试
- 参校对失败不应该重试
- 只读操作可以重试
- 幂等写操作可以重试
- 非幂等写操作不能重试（重试可能导致脏写，或产生重复数据）

无状态 (Stateless) 重试

具体用法请查看样例..

- 无状态重试，是在一个循环中执行完重试策略，即重试上下文保持在一个线程上下文中，在一次调用中进行完整的重试策略判断。非常简单的情况，如远程调用某个查询方法时是最常见的无状态重试。

- 如果远程方法调用是没有事务，远程方法调用的时候不需要设置。

有状态(Stateful)重试

具体用法请查看样例..

- 有状态重试，有两种情况需要使用有状态重试，事务操作需要回滚或者熔断器模式。 事务操作需要回滚场景时，当整个操作中抛出的是数据库异常DataAccessException，异常会往外抛，使事务回滚，这里不能进行重试，而抛出其他异常则可以进行重试。

Spring Retry 注解

@EnableRetry

- @EnableRetry能否重试。当proxyTargetClass属性为true时，使用CGLIB代理。默认使用标准JAVA注解。

```
@SpringBootApplication
@EnableRetry
//proxyTargetClass属性为true时，使用CGLIB代理。默认使用标准JAVA注解。
@EnableAspectJAutoProxy(proxyTargetClass = true)
public class SpringretryApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringretryApplication.class, args);
    }

}
```

Maven dependency

```
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
</dependency>
```

Spring Retry 注解

@Retryable

注解需要被重试的方法

- **value：** 需要进行重试的异常，和参数includes是一个意思。默认为空，当参数exclude也为空时，所有异常都将要求重试。
- **include：** 需要进行重试的异常，默认为空。当参数exclude也为空时，所有异常都将要求重试。
- **exclude：** 不需要重试的异常。默认为空，当参include也为空时，所有异常都将要求重试。
- **stateful** 标明重试是否是有状态的，异常引发事物失效的时候需要注意这个。该参数默认为false。远程方法调用的时候不需要设置，因为远程方法调用是没有事物的；只有当数据库更新操作的时候需要设置该值为true，特别是使用Hibernate的时候。抛出异常时，异常会往外抛，使事物回滚；重试的时候会启用一个新的有效的事物。
- **maxAttempts：** 最大重试次数，默认为3。包括第一次失败。
- **backoff：** 回避策略，默认为空。该参数为空时是，失败立即重试，重试的时候阻塞线程。
- **exceptionExpression：** SimpleRetryPolicy.canRetry()返回true时该表达式才会生效，触发重试机制。如果抛出多个异常，只会检查最后那个。表达式举例："message.contains('you can retry this')“ 并且 "@someBean.shouldRetry(#root)“

```
@Retryable(value = {
        RemoteAccessException.class }, maxAttempts = 3, backoff = @Backoff(delay = 500l, multiplier = 1))
public void call() throws Exception {
    logger.info(LocalTime.now() + " do something...");
    throw new RemoteAccessException("RPC调用异常");
}
```

使用了@Retryable的方法里面不能使用try...catch包裹，要在发放上抛出异常，不然不会触发。

Spring Retry 注解

@Backoff

重试回退策略（立即重试还是等待一会再重试）

- **value：**重试延迟时间，单位毫秒，默认值1000，即默认延迟1秒。当未设置multiplier时，表示每隔value的时间重试，直到重试次数到达maxAttempts设置的最大允许重试次数。当设置了multiplier参数时，该值作为幂运算的初始值。
- **delay：**等同value参数，两个参数设置一个即可。
- **maxDelay：**两次重试间最大间隔时间。当设置multiplier参数后，下次延迟时间根据是上次延迟时间乘以multiplier得出的，这会导致两次重试间的延迟时间越来越长，该参数限制两次重试的最大间隔时间，当间隔时间大于该值时，计算出的间隔时间将会被忽略，使用上次的重试间隔时间。
- **multiplier：**作为乘数用于计算下次延迟时间。公式：delay = delay * multiplier
- **random：**是否启用随机退避策略，默认false。设置为true时启用退避策略，重试延迟时间将是delay和maxDelay间的一个随机数。设置该参数的目的是重试的时候避免同时发起重试请求，造成Ddos攻击。

```
@Retryable(value = { Exception.class }, maxAttempts = 3, backoff = @Backoff(delay = 2000, multiplier = 1.5))
public int minGoodsnum(int num) throws Exception {
    logger.info("minGoodsnum开始" + LocalTime.now());

    if (num <= 0) {
        throw new Exception("数量不对");
    }
    logger.info("minGoodsnum执行结束");
    return totalNum - num;
}
```

Spring Retry 注解

@Recover

用于方法。用于**@Retryable**失败时的“兜底”处理方法。

- 该注解用于恢复处理方法，当全部尝试都失败时执行。返回参数必须和@Retryable修饰的方法返回参数完全一样。第一个参数必须是异常，其他参数和@Retryable修饰的方法参数顺序一致。
- 要触发@Recover方法，那么在@Retryable方法上不能有返回值，只能是void才能生效。

```
@Retryable(value = { RemoteAccessException.class,
        TimeoutException.class }, maxAttempts = 5, backoff = @Backoff(delay = 5001, multiplier = 1))
public void retryTest02(String arg01) throws Exception {
    System.out.println("do something...");
    throw new TimeoutException("TimeoutException....");
}

@Recover
public void recover(RemoteAccessException e,String arg01) {
    System.out.println(e.getMessage());
    System.out.println("RemoteAccessException recover...."+arg01);
}
```

Spring Retry 注解

@CircuitBreaker

用于方法，实现熔断模式。

- **include:** 指定处理的异常类。默认为空
- **exclude:** 指定不需要处理的异常。默认为空
- **vaue:** 指定要重试的异常。默认为空
- **maxAttempts:** 最大重试次数。默认3次
- **openTimeout:** 配置熔断器打开的超时时间，默认5s，当超过openTimeout之后熔断器电路变成半打开状态（只要有一次重试成功，则闭合电路）
- **resetTimeout:** 配置熔断器重新闭合的超时时间，默认20s，超过这个时间断路器关闭

```
@Service
class ShakyBusinessService {

    @Recover
    public int fallback(BoomException ex) {
        return 2;
    }

    @CircuitBreaker(include = BoomException.class, openTimeout = 20000L, resetTimeout = 5000L, maxAttempts = 1)
    public int desireNumber() throws Exception {
        System.out.println("calling desireNumber()");
        if (Math.random() > .5) {
            throw new BoomException("Boom");
        }
        return 1;
    }
}
```

RetryTemplate

什么时候使用RetryTemplate?

- 不使用spring容器的时候，使用了@Retryable，@CircuitBreaker的方法不能在本类被调用，不然重试机制不会生效。也就是要标记为@Service，然后在其它类使用@Autowired注入或者@Bean去实例才能生效。
- 需要使用复杂策略机制和异常场景时
- 使用有状态重试,且需要全局模式时建议使用
- 需要使用监听器Listener的场景
- 需要使用Retry统计分析

RetryPolicy 重试策略

- **NeverRetryPolicy**：只允许调用RetryCallback一次，不允许重试；
- **AlwaysRetryPolicy：**允许无限重试，直到成功，此方式逻辑不当会导致死循环；
- **SimpleRetryPolicy**：固定次数重试策略，默认重试最大次数为3次，RetryTemplate默认使用的策略；
- **TimeoutRetryPolicy：**超时时间重试策略，默认超时时间为1秒，在指定的超时时间内允许重试；
- **CircuitBreakerRetryPolicy**：有熔断功能的重试策略，需设置3个参数openTimeout、resetTimeout和delegate，稍后详细介绍该策略；
- **CompositeRetryPolicy：**组合重试策略，有两种组合方式，乐观组合重试策略是指只要有一个策略允许重试即可以，悲观组合重试策略是指只要有一个策略不允许重试即可以，但不管哪种组合方式，组合中的每一个策略都会执行。

BackOffPolicy
退避策略

- **NoBackOffPolicy：**无退避算法策略，即当重试时是立即重试；
- **FixedBackOffPolicy：**固定时间的退避策略，需设置参数sleeper和backOffPeriod，sleeper指定等待策略，默认是Thread.sleep，即线程休眠，backOffPeriod指定休眠时间，默认1秒；
- **UniformRandomBackOffPolicy：**随机时间退避策略，需设置sleeper、minBackOffPeriod和maxBackOffPeriod，该策略在[minBackOffPeriod,maxBackOffPeriod之间取一个随机休眠时间，minBackOffPeriod默认500毫秒，maxBackOffPeriod默认1500毫秒；
- **ExponentialBackOffPolicy：**指数退避策略，需设置参数sleeper、initialInterval、maxInterval和multiplier，initialInterval指定初始休眠时间，默认100毫秒，maxInterval指定最大休眠时间，默认30秒，multiplier指定乘数，即下一次休眠时间为当前休眠时间*multiplier；
- **ExponentialRandomBackOffPolicy：**随机指数退避策略，引入随机乘数，之前说过固定乘数可能会引起很多服务同时重试导致DDos，使用随机休眠时间来避免这种情况。

DEMO
RetryTemplate

```java
/**
 *  TimeoutRetryPolicy策略，TimeoutRetryPolicy超时时间默认是1秒。
 *  TimeoutRetryPolicy超时是指在execute方法内部，从open操作开始到调用TimeoutRetryPolicy的canRetry方法这之间所经过的时间。
 *  这段时间未超过TimeoutRetryPolicy定义的超时时间，那么执行操作，否则抛出异常。
 *  当重试执行完闭，操作还未成为，那么可以通过RecoveryCallback完成一些失败事后处理。
 */
public class RetryTemplate01 {

    public static void main(String[] args) throws Exception {
        RetryTemplate template = new RetryTemplate();
        TimeoutRetryPolicy policy = new TimeoutRetryPolicy();
        template.setRetryPolicy(policy);

        String result = template.execute(new RetryCallback<String, Exception>() {
            public String doWithRetry(RetryContext arg0) throws Exception {
                return "Retry";
            }
        });
        System.out.println(result);
    }
}
```

DEMO
RetryTemplate

```java
/**
 *   代码重试两次后，仍然失败，RecoveryCallback被调用，返回"recovery callback"。
 *   如果没有定义RecoveryCallback，那么重试2次后，将会抛出异常。
 */
public class RetryTemplate02 {

    public static void main(String[] args) throws Exception {

        RetryTemplate template = new RetryTemplate();
        SimpleRetryPolicy policy = new SimpleRetryPolicy();
        policy.setMaxAttempts(2);
        template.setRetryPolicy(policy);

        String result = template.execute(new RetryCallback<String, Exception>() {
            public String doWithRetry(RetryContext arg0) throws Exception {
                throw new NullPointerException("nullPointerException");
            }
        }, new RecoveryCallback<String>() {
            public String recover(RetryContext context) throws Exception {
                return "recovery callback";
            }
        });
        System.out.println(result);
    }
}
```

DEMO
RetryTemplate

```java
/**
 *   通过监听器，可以在重试操作的某些位置嵌入调用者定义的一些操作，以便在某些场景触发。
 *   代码注册了两个Listener，Listener中的三个实现方法，onError,open,close会在执行重试操作时被调用，
 *   在RetryTemplate中doOpenInterceptors,doCloseInterceptors,doOnErrorInterceptors会调用监听器对应的open,close,onError方法。
 *   doOpenInterceptors方法在第一次重试之前会被调用，如果该方法返回true，则会继续向下直接，如果返回false，则抛出异常，停止重试。
 *   doCloseInterceptors 会在重试操作执行完毕后调用。
 *   doOnErrorInterceptors 在抛出异常后执行，
 * 当注册多个Listener时，open方法按会按Listener的注册顺序调用，而onError和close则按Listener注册的顺序逆序调用。
 */
public class RetryTemplate06 {

    public static void main(String[] args) throws Exception {

        RetryTemplate template = new RetryTemplate();

        ExponentialRandomBackOffPolicy exponentialBackOffPolicy = new ExponentialRandomBackOffPolicy();
        exponentialBackOffPolicy.setInitialInterval(1500);
        exponentialBackOffPolicy.setMultiplier(2);
        exponentialBackOffPolicy.setMaxInterval(6000);

        CompositeRetryPolicy policy = new CompositeRetryPolicy();
        RetryPolicy[] polices = { new SimpleRetryPolicy(), new AlwaysRetryPolicy() };

        policy.setPolicies(polices);
        policy.setOptimistic(true);

        template.setRetryPolicy(policy);
        template.setBackOffPolicy(exponentialBackOffPolicy);

        template.registerListener(new RetryListener() {
            public <T, E extends Throwable> boolean open(RetryContext context, RetryCallback<T, E> callback) {
                System.out.println("open");
                return true;
            }
            public <T, E extends Throwable> void onError(RetryContext context, RetryCallback<T, E> callback,
                    Throwable throwable) {
                System.out.println("onError");
            }
            public <T, E extends Throwable> void close(RetryContext context, RetryCallback<T, E> callback,
                    Throwable throwable) {
                System.out.println("close");
            }
        });
```

DEMO
RetryTemplate

```
/**
* 当把状态放入缓存时，通过该key查询获取，全局模式 DataAccessException进行回滚
 */
public class RetryTemplate07 {

    public static void main(String[] args) throws Exception {
        RetryTemplate template = new RetryTemplate();
        Object key = "mykey";
        boolean isForceRefresh = true;
        BinaryExceptionClassifier rollbackClassifier = new BinaryExceptionClassifier(
                Collections.<Class<? extends Throwable>>singleton(DataAccessException.class));
        RetryState state = new DefaultRetryState(key, isForceRefresh, rollbackClassifier);
        String result = template.execute(new RetryCallback<String, RuntimeException>() {
            @Override
            public String doWithRetry(RetryContext context) throws RuntimeException {
                System.out.println("retry count:" + context.getRetryCount());
                throw new TypeMismatchDataAccessException();
            }
        }, new RecoveryCallback<String>() {
            @Override
            public String recover(RetryContext context) throws Exception {
                return "default";
            }
        }, state);
        System.out.println(result);
    }
}
```

XML Configuration

xml配置可以在不修改原来代码的情况下通过，添加spring retry的功能。

```java
@SpringBootApplication
@EnableRetry
@EnableAspectJAutoProxy
@ImportResource("classpath:/retryadvice.xml")
public class XmlApplication {
    public static void main(String[] args) {
        SpringApplication.run(XmlApplication.class, args);
    }
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.xsd
                        http://www.springframework.org/schema/aop
                        http://www.springframework.org/schema/aop/spring-aop.xsd">
    <aop:config>
        <aop:pointcut id="transactional" expression="execution(*XmlRetryService.xmlRetryService(..))" />
        <aop:advisor pointcut-ref="transactional" advice-ref="taskRetryAdvice" order="-1" />
    </aop:config>
    <bean id="taskRetryAdvice" class="org.springframework.retry.interceptor.RetryOperationsInterceptor">
        <property name="RetryOperations" ref="taskRetryTemplate" />
    </bean>
    <bean id="taskRetryTemplate" class="org.springframework.retry.support.RetryTemplate">
        <property name="retryPolicy" ref="taskRetryPolicy" />
        <property name="backOffPolicy" ref="exponentialBackOffPolicy" />
    </bean>
    <bean id="taskRetryPolicy" class="org.springframework.retry.policy.SimpleRetryPolicy">
        <constructor-arg index="0" value="5" />
        <constructor-arg index="1">
            <map>
                <entry key="org.springframework.remoting.RemoteAccessException" value="true" />
            </map>
        </constructor-arg>
    </bean>
    <bean id="exponentialBackOffPolicy"
        class="org.springframework.retry.backoff.ExponentialBackOffPolicy">
        <property name="initialInterval" value="300" />
        <property name="maxInterval" value="30000" />
        <property name="multiplier" value="2.0" />
    </bean>
</beans>
```

DEMO
RetryTemplate

更多样例请查看：https://git.sha.mastercard.int/stash/projects/AP/repos/ap-starter-samples/browse/springretry