9_重试机制

> 仅供 编程导航 <https://www.code-nav.cn/post/1816420035119853569> 内部成员观看，请勿对外分享！

一、需求分析

目前，如果使用 RPC 框架的服务消费者调用接口失败，就会直接报错。

调用接口失败可能有很多原因，有时可能是服务提供者返回了错误，但有时可能只是网络不稳定或服务提供者重启等临时性问题。这种情况下，我们可能更希望服务消费者拥有自动重试的能力，提高系统的可用性。

本节教程，鱼皮就带大家实现服务消费端的重试机制。

二、设计方案

重试机制

重试的概念我相信大家都能理解，不必多说，就是 “不行再来” 呗。

我们需要掌握的是 “如何设计重试机制”，重试机制的核心是 **重试策略**，一般来说，包含以下几个考虑点：

1. 什么时候、什么条件下重试？
2. 重试时间（确定下一次的重试时间）
3. 什么时候、什么条件下停止重试？
4. 重试后要做什么？

重试条件

首先是什么时候、什么条件下重试？

这个比较好思考，如果我们希望提高系统的可用性，当由于网络等异常情况发生时，触发重试。

重试时间

重试时间（也叫重试等待）的策略就比较丰富了，可能会用到一些算法，主流的重试时间算法有：

1）固定重试间隔（Fixed Retry Interval）：在每次重试之间使用固定的时间间隔。

比如近 5 次重试的时间点如下：

```
1s
2s
3s
4s
5s
```

2）指数退避重试（Exponential Backoff Retry）：在每次失败后，重试的时间间隔会以指数级增加，以避免请求过于密集。

比如近 5 次重试的时间点如下：

```
1s
3s（多等 2s）
7s（多等 4s）
15s（多等 8s）
31s（多等 16s）
```

3）随机延迟重试（Random Delay Retry）：在每次重试之间使用随机的时间间隔，以避免请求的同时发生。

4）可变延迟重试（Variable Delay Retry）：这种策略更 “高级” 了，根据先前重试的成功或失败情况，动态调整下一次重试的延迟时间。比如，根据前一次的响应时间调整下一次重试的等待时间。

值得一提的是，以上的策略是可以组合使用的，一定要根据具体情况和需求灵活调整。比如可以先使用指数退避重试策略，如果连续多次重试失败，则切换到固定重试间隔策略。

停止重试

一般来说，重试次数是有上限的，否则随着报错的增多，系统同时发生的重试也会越来越多，造成雪崩。

主流的停止重试策略有：

1）最大尝试次数：一般重试当达到最大次数时不再重试。

2）超时停止：重试达到最大时间的时候，停止重试。

重试工作

最后一点是重试后要做什么事情？一般来说就是重复执行原本要做的操作，比如发送请求失败了，那就再发一次请求。

需要注意的是，当重试次数超过上限时，往往还要进行其他的操作，比如：

1. 通知告警：让开发者人工介入
2. 降级容错：改为调用其他接口、或者执行其他操作

重试方案设计

回归到我们的 RPC 框架，消费者发起调用的代码如下：

```
try {
    // rpc 请求
    RpcResponse rpcResponse = VertxTcpClient.doRequest(rpcRequest, selectedSer
    return rpcResponse.getData();
} catch (Exception e) {
    throw new RuntimeException("调用失败");
}
```

我们完全可以将 `VertxTcpClient.doRequest` 封装为一个可重试的任务，如果请求失败（重试条件），系统就会自动按照重试策略再次发起请求，不用开发者关心。

对于重试算法，我们就选择主流的重试算法好了，Java 中可以使用 Guava-Retrying 库轻松实现多种不同的重试算法，非常简单，后文直接带大家实战。

鱼皮之前专门写过一篇 Guava-Retrying 的教程文章：
https://cloud.tencent.com/developer/article/1752086
<https://cloud.tencent.com/developer/article/1752086>

和序列化器、注册中心、负载均衡器一样，重试策略本身也可以使用 SPI + 工厂的方式，允许开发者动态配置和扩展自己的重试策略。

最后，如果重试超过一定次数，我们就停止重试，并且抛出异常。在下节教程中，还会给大家分享重试失败后的另一种选择 —— 容错机制。

三、开发实现

1、多种重试策略实现

下面鱼皮带大家实现 2 种最基本的重试策略：不重试、固定重试间隔。

没错，不重试也是一种重试策略哈哈！

在 RPC 项目中新建 `fault.retry` 包，将所有重试相关的代码放到该包下。

1）先编写重试策略通用接口。提供一个重试方法，接受一个具体的任务参数，可以使用 Callable 类代表一个任务。
代码如下：

```java
package com.yupi.yurpc.fault.retry;

import com.yupi.yurpc.model.RpcResponse;

import java.util.concurrent.Callable;

/**
 * 重试策略
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>

 * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>

 * @from <a href="https://yupi.icu">编程导航学习圈</a>

 */
public interface RetryStrategy {

    /**
     * 重试
     *
     * @param callable
     * @return
     * @throws Exception
     */
    RpcResponse doRetry(Callable<RpcResponse> callable) throws Exception;
}
```

2）引入 Guava-Retrying 重试库，代码如下：

```xml
<!-- https://github.com/rholder/guava-retrying -->
<dependency>
    <groupId>com.github.rholder</groupId>

    <artifactId>guava-retrying</artifactId>

    <version>2.0.0</version>

</dependency>

```

3）不重试策略实现。

就是直接执行一次任务，代码如下：

```java
package com.yupi.yurpc.fault.retry;

import com.yupi.yurpc.model.RpcResponse;
import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.Callable;

/**
 * 不重试 - 重试策略
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>

 * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>

 * @from <a href="https://yupi.icu">编程导航学习圈</a>

 */
@Slf4j
public class NoRetryStrategy implements RetryStrategy {

    /**
     * 重试
     *
     * @param callable
     * @return
     * @throws Exception
     */
    public RpcResponse doRetry(Callable<RpcResponse> callable) throws Excepti
        return callable.call();
    }

}
```

4）固定重试间隔策略实现。

使用 Guava-Retrying 提供的 `RetryerBuilder` 能够很方便地指定重试条件、重试等待策略、重试停止策略、重试工作等。

代码如下：

```java
package com.yupi.yurpc.fault.retry;

import com.github.rholder.retry.*;
import com.yupi.yurpc.model.RpcResponse;
import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;

/**
 * 固定时间间隔 - 重试策略
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>

 * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>

 * @from <a href="https://yupi.icu">编程导航学习圈</a>

 */
@Slf4j
public class FixedIntervalRetryStrategy implements RetryStrategy {

    /**
     * 重试
     *
     * @param callable
     * @return
     * @throws ExecutionException
     * @throws RetryException
     */
    public RpcResponse doRetry(Callable<RpcResponse> callable) throws Executi
        Retryer<RpcResponse> retryer = RetryerBuilder.<RpcResponse>newBuilder
                .retryIfExceptionOfType(Exception.class)
                .withWaitStrategy(WaitStrategies.fixedWait(3L, TimeUnit.SECON
                .withStopStrategy(StopStrategies.stopAfterAttempt(3))
                .withRetryListener(new RetryListener() {
                    @Override
                    public <V> void onRetry(Attempt<V> attempt) {
                        log.info("重试次数 {}", attempt.getAttemptNumber());
                    }
                })
                .build();
        return retryer.call(callable);
    }

}
```

3190字 上述代码中，重试策略如下：

- 重试条件：使用 retryIfExceptionOfType 方法指定当出现 Exception 异常时重试。
- 重试等待策略：使用 withWaitStrategy 方法指定策略，选择 fixedWait 固定时间间隔策略。
- 重试停止策略：使用 withStopStrategy 方法指定策略，选择 stopAfterAttempt 超过最大重试次数停止。
- 重试工作：使用 withRetryListener 监听重试，每次重试时，除了再次执行任务外，还能够打印当前的重试次数。

5）可以简单编写一个单元测试，来验证不同的重试策略，这是最好的学习方式。

单元测试代码如下：

```
package com.yupi.yurpc.fault.retry;

import com.yupi.yurpc.model.RpcResponse;
import org.junit.Test;

/**
 * 重试策略测试
 */
public class RetryStrategyTest {

    RetryStrategy retryStrategy = new NoRetryStrategy();

    @Test
    public void doRetry() {
        try {
            RpcResponse rpcResponse = retryStrategy.doRetry(() -> {
                System.out.println("测试重试");
                throw new RuntimeException("模拟重试失败");
            });
            System.out.println(rpcResponse);
        } catch (Exception e) {
            System.out.println("重试多次失败");
            e.printStackTrace();
        }
    }
}
```

2、支持配置和扩展重试策略

一个成熟的 RPC 框架可能会支持多种不同的重试策略，像序列化器、注册中心、负载均衡器一样，我们的需求是，让开发者能够填写配置来指定使用的重试策略，并且支持自定义重试策略，让框架更易用、更利于扩展。

要实现这点，开发方式和序列化器、注册中心、负载均衡器都是一样的，都可以使用工厂创建对象、使用 SPI 动态加载自定义的注册中心。

1）重试策略常量。

在 `fault.retry` 包下新建 `RetryStrategyKeys` 类，列举所有支持的重试策略键名。

代码如下：

```
package com.yupi.yurpc.fault.retry;

/**
 * 重试策略键名常量
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>

 * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>

 * @from <a href="https://yupi.icu">编程导航学习圈</a>

 */
public interface RetryStrategyKeys {

    /**
     * 不重试
     */
    String NO = "no";

    /**
     * 固定时间间隔
     */
    String FIXED_INTERVAL = "fixedInterval";

}
```

2）使用工厂模式，支持根据 key 从 SPI 获取重试策略对象实例。

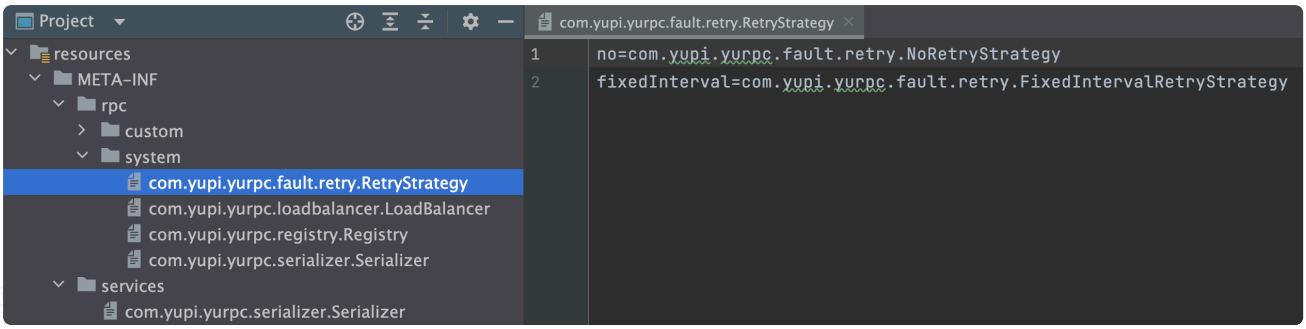
在 `fault.retry` 包下新建 `RetryStrategyFactory` 类，代码如下：

```java
package com.yupi.yurpc.fault.retry;

import com.yupi.yurpc.spi.SpiLoader;

/**
 * 重试策略工厂（用于获取重试器对象）
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>

 * @learn <a href="https://codefather.cn">编程宝典</a>

 * @from <a href="https://yupi.icu">编程导航知识星球</a>

 */
public class RetryStrategyFactory {

    static {
        SpiLoader.load(RetryStrategy.class);
    }

    /**
     * 默认重试器
     */
    private static final RetryStrategy DEFAULT_RETRY_STRATEGY = new NoRetrySt

    /**
     * 获取实例
     *
     * @param key
     * @return
     */
    public static RetryStrategy getInstance(String key) {
        return SpiLoader.getInstance(RetryStrategy.class, key);
    }

}
```

这个类可以直接复制之前的 SerializerFactory，然后略做修改。可以发现，只要跑通了一次 SPI 机制，后续的开发就很简单了~

3）在 `META-INF` 的 `rpc/system` 目录下编写重试策略接口的 SPI 配置文件，文件名称为 `com.yupi.yurpc.fault.retry.RetryStrategy` 。

如图：



代码如下：

```
no=com.yupi.yurpc.fault.retry.NoRetryStrategy
fixedInterval=com.yupi.yurpc.fault.retry.FixedIntervalRetryStrategy
```

4）为 RpcConfig 全局配置新增重试策略的配置，代码如下：

```
@Data
public class RpcConfig {
    /**
     * 重试策略
     */
    private String retryStrategy = RetryStrategyKeys.NO;
}
```

3、应用重试功能

现在，我们就能够愉快地使用重试功能了。修改 ServiceProxy 的代码，从工厂中获取重试器，并且将请求代码封装为一个 Callable 接口，作为重试器的参数，调用重试器即可。

修改的代码如下：

```
// 使用重试机制
RetryStrategy retryStrategy = RetryStrategyFactory.getInstance(rpcConfig.getRe
RpcResponse rpcResponse = retryStrategy.doRetry(() ->
        VertxTcpClient.doRequest(rpcRequest, selectedServiceMetaInfo)
);
```

上述代码中，使用 Lambda 表达式将 `VertxTcpClient.doRequest` 封装为了一个匿名函数，简化了代码。

修改后的 ServiceProxy 的完整代码如下：

```java
/**
 * 服务代理（JDK 动态代理）
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>

 * @learn <a href="https://codefather.cn">编程宝典</a>

 * @from <a href="https://yupi.icu">编程导航知识星球</a>

 */
public class ServiceProxy implements InvocationHandler {

    /**
     * 调用代理
     *
     * @return
     * @throws Throwable
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws T
        // 指定序列化器
        final Serializer serializer = SerializerFactory.getInstance(RpcApplic

        // 构造请求
        String serviceName = method.getDeclaringClass().getName();
        RpcRequest rpcRequest = RpcRequest.builder()
                .serviceName(serviceName)
                .methodName(method.getName())
                .parameterTypes(method.getParameterTypes())
                .args(args)
                .build();
        try {
            // 从注册中心获取服务提供者请求地址
            RpcConfig rpcConfig = RpcApplication.getRpcConfig();
            Registry registry = RegistryFactory.getInstance(rpcConfig.getRegi
            ServiceMetaInfo serviceMetaInfo = new ServiceMetaInfo();
            serviceMetaInfo.setServiceName(serviceName);
            serviceMetaInfo.setServiceVersion(RpcConstant.DEFAULT_SERVICE_VER
            List<ServiceMetaInfo> serviceMetaInfoList = registry.serviceDisco
            if (CollUtil.isEmpty(serviceMetaInfoList)) {
                throw new RuntimeException("暂无服务地址");
            }

            // 负载均衡
            LoadBalancer loadBalancer = LoadBalancerFactory.getInstance(rpcCo
            // 将调用方法名（请求路径）作为负载均衡参数
            Map<String, Object> requestParams = new HashMap<>();
            requestParams.put("methodName", rpcRequest.getMethodName());
            ServiceMetaInfo selectedServiceMetaInfo = loadBalancer.select(req

```

```
51                // rpc 请求
52                // 使用重试机制
53                RetryStrategy retryStrategy = RetryStrategyFactory.getInstance(rp
54                RpcResponse rpcResponse = retryStrategy.doRetry(() ->
55                        VertxTcpClient.doRequest(rpcRequest, selectedServiceMetaI
56                );
57                return rpcResponse.getData();
58            } catch (Exception e) {
59                throw new RuntimeException("调用失败");
60            }
61        }
62    }
```

我们会发现，即使引入了重试机制，整段代码并没有变得更复杂，这就是可扩展性设计的巧妙之处。

四、测试

首先启动服务提供者，然后使用 Debug 模式启动服务消费者，当服务消费者发起调用时，立刻停止服务提供者，就会看到调用失败后重试的情况。

五、扩展

1）新增更多不同类型的重试器。

参考思路：比如指数退避算法的重试器。