10_容错机制

> 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

上节教程中，我们给 RPC 框架增加了重试机制，提升了服务消费端的可靠性和健壮性。

但如果重试超过了一定次数仍然失败，我们又该怎么处理呢？

或者说当调用出现失败时，我们一定要重试么？有没有其他的策略呢？

本节教程，鱼皮带大家实现另一种提高服务消费端可靠性和健壮性的机制 —— 容错机制。

二、设计方案

容错机制

容错是指系统在出现异常情况时，可以通过一定的策略保证系统仍然稳定运行，从而提高系统的可靠性和健壮性。

在分布式系统中，容错机制尤为重要，因为分布式系统中的各个组件都可能存在网络故障、节点故障等各种异常情况。要顾全大局，尽可能消除偶发 / 单点故障对系统带来的整体影响。

打个比方，将分布式系统类比为一家公司，如果公司某个优秀员工请假了，需要 “触发容错” ，让另一个普通员工顶上，这本质上是容错机制的一种 **降级** 策略。

容错机制一般都是在系统出现错误时才触发的，这点没什么好讲的，我们需要重点学习的是容错策略和容错实现方式。

容错策略

容错策略有很多种，常用的容错策略主要是以下几个：

1）Fail-Over 故障转移：一次调用失败后，切换一个其他节点再次进行调用，也算是一种重试。

2）Fail-Back 失败自动恢复：系统的某个功能出现调用失败或错误时，通过其他的方法，恢复该功能的正常。可以理解为降级，比如重试、调用其他服务等。

3）Fail-Safe 静默处理：系统出现部分非重要功能的异常时，直接忽略掉，不做任何处理，就像错误没有发生过一样。

4）Fail-Fast 快速失败：系统出现调用错误时，立刻报错，交给外层调用方处理。

容错实现方式

容错其实是个比较广泛的概念，除了上面几种策略外，很多技术都可以起到容错的作用。

比如：

1）重试：重试本质上也是一种容错的降级策略，系统错误后再试一次。

2）限流：当系统压力过大、已经出现部分错误时，通过限制执行操作（接受请求）的频率或数量，对系统进行保护。

3）降级：系统出现错误后，改为执行其他更稳定可用的操作。也可以叫做 “兜底” 或 “有损服务”，这种方式的本质是：即使牺牲一定的服务质量，也要保证系统的部分功能可用，保证基本的功能需求得到满足。

4）熔断：系统出现故障或异常时，暂时中断对该服务的请求，而是执行其他操作，以避免连锁故障。

5）超时控制：如果请求或操作长时间没处理完成，就进行中断，防止阻塞和资源占用。

注意，在实际项目中，根据对系统可靠性的需求，我们通常会结合多种策略或方法实现容错机制。

容错方案设计

回归到我们的 RPC 框架，之前已经给系统增加重试机制了，算是实现了一部分的容错能力。

现在，我们可以正式引入容错机制，通过更多策略来进一步增加系统可靠性。

容错方案的设计可以是很灵活的，建议大家先自己思考。

这里鱼皮提供 2 种方案：

1）先容错再重试。

当系统发生异常时，首先会触发容错机制，比如记录日志、进行告警等，然后可以选择是否进行重试。

这种方案其实是把重试当做容错机制的一种可选方案。

2）先重试再容错。在发生错误后，首先尝试重试操作，如果重试多次仍然失败，则触发容错机制，比如记录日志、进行告警等。

但其实大家能不能想到，这 2 种方案其实完全可以结合使用！

系统错误时，先通过重试操作解决一些临时性的异常，比如网络波动、服务端临时不可用等；如果重试多次后仍然失败，说明可能存在更严重的问题，这时可以触发其他的容错策略，比如调用降级服务、熔断、限流、快速失败等，来减少异常的影响，保障系统的稳定性和可靠性。

举个具体的例子：

1. 系统调用服务 A 出现网络错误，使用容错策略 - 重试。
2. 重试 3 次失败后，使用其他容错策略 - 降级。
3. 系统改为调用不依赖网络的服务 B，完成操作。

下面，鱼皮就带大家实现这种方案。

三、开发实现

整个容错机制的实现方式几乎和重试机制一模一样，能力较强的同学可以先尝试自己实现。

1、多种容错策略实现

下面鱼皮带大家实现 2 种最基本的容错策略：Fail-Fast 快速失败、Fail-Safe 静默处理。

在 RPC 项目中新建 `fault.tolerant` 包，将所有容错相关的代码放到该包下。

1）先编写容错策略通用接口。提供一个容错方法，使用 Map 类型的参数接受上下文信息（可用于灵活地传递容错处理需要用到的数据），并且接受一个具体的异常类参数。

由于容错是应用到发送请求操作的，所以容错方法的返回值是 RpcResponse（响应）。

代码如下：

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

import com.yupi.yurpc.model.RpcResponse;

import java.util.Map;

/**
 * 容错策略
 *
 * @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 TolerantStrategy {

    /**
     * 容错
     *
     * @param context 上下文，用于传递数据
     * @param e       异常
     * @return
     */
    RpcResponse doTolerant(Map<String, Object> context, Exception e);
}
```

2）快速失败容错策略实现。

很好理解，就是遇到异常后，将异常再次抛出，交给外层处理。

代码如下：

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

import com.yupi.yurpc.model.RpcResponse;

import java.util.Map;

/**
 * 快速失败 - 容错策略（立刻通知外层调用方）
 *
 * @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 class FailFastTolerantStrategy implements TolerantStrategy {

    @Override
    public RpcResponse doTolerant(Map<String, Object> context, Exception e) {
        throw new RuntimeException("服务报错", e);
    }
}
```

3）静默处理容错策略实现。

也很好理解，就是遇到异常后，记录一条日志，然后正常返回一个响应对象，就好像没有出现过报错。

代码如下：

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

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

import java.util.Map;

/**
 * 静默处理异常 - 容错策略
 *
 * @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 FailSafeTolerantStrategy implements TolerantStrategy {

    @Override
    public RpcResponse doTolerant(Map<String, Object> context, Exception e) {
        log.info("静默处理异常", e);
        return new RpcResponse();
    }
}
```

4）其他容错策略。

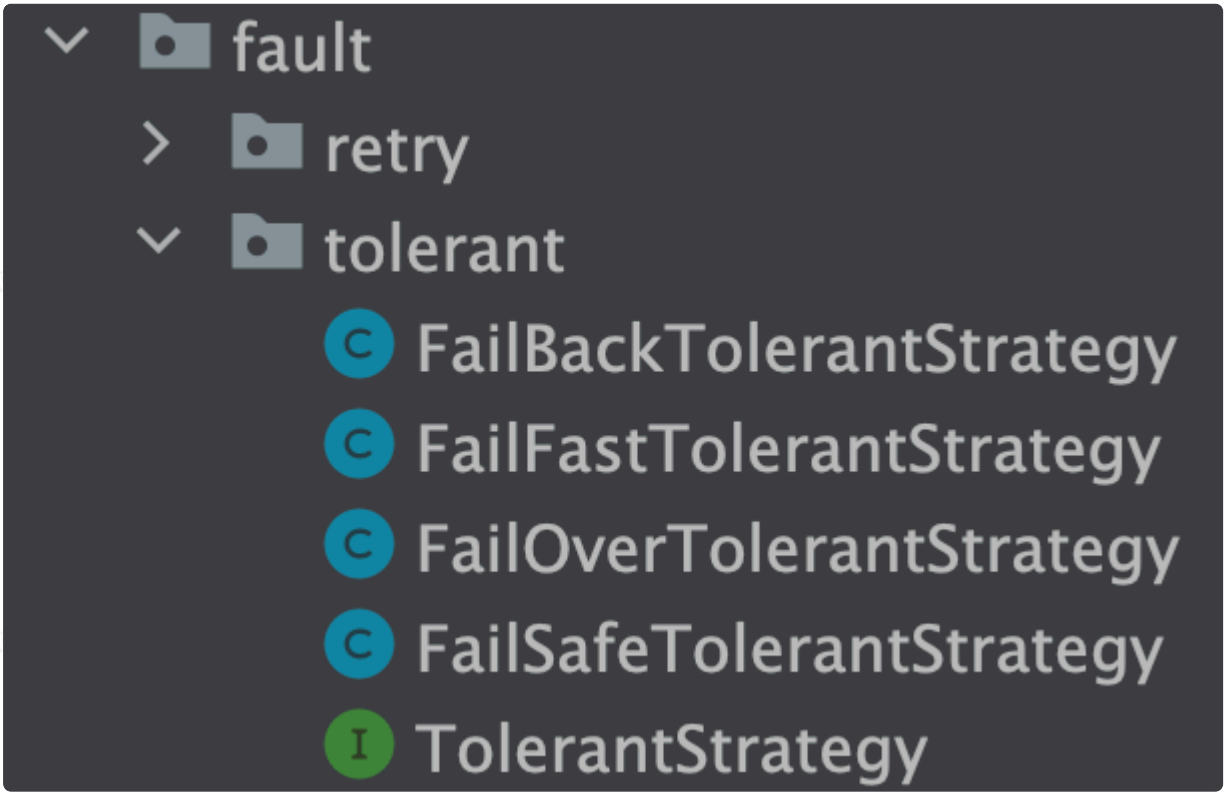
还可以自行实现更多的容错策略，比如 `FailBackTolerantStrategy` 故障恢复策略：

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

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

import java.util.Map;

/**
 * 降级到其他服务 - 容错策略
 *
 * @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 FailBackTolerantStrategy implements TolerantStrategy {

    @Override
    public RpcResponse doTolerant(Map<String, Object> context, Exception e) {
        // todo 可自行扩展，获取降级的服务并调用
        return null;
    }
}
```

还有 `FailOverTolerantStrategy` 故障转移策略：

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

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

import java.util.Map;

/**
 * 转移到其他服务节点 - 容错策略
 *
 * @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 FailOverTolerantStrategy implements TolerantStrategy {

    @Override
    public RpcResponse doTolerant(Map<String, Object> context, Exception e) {
        // todo 可自行扩展，获取其他服务节点并调用
        return null;
    }
}
```

当前的容错机制目录如下：



2、支持配置和扩展容错策略

一个成熟的 RPC 框架可能会支持多种不同的容错策略，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容错策略，并且支持自定义容错策略，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）容错策略常量。

在 `fault.tolerant` 包下新建 `TolerantStrategyKeys` 类，列举所有支持的容错策略键名。

代码如下：

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

/**
 * 容错策略键名常量
 *
 * @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 TolerantStrategyKeys {

    /**
     * 故障恢复
     */
    String FAIL_BACK = "failBack";

    /**
     * 快速失败
     */
    String FAIL_FAST = "failFast";

    /**
     * 故障转移
     */
    String FAIL_OVER = "failOver";

    /**
     * 静默处理
     */
    String FAIL_SAFE = "failSafe";

}
```

2）使用工厂模式，支持根据 key 从 SPI 获取容错策略对象实例。

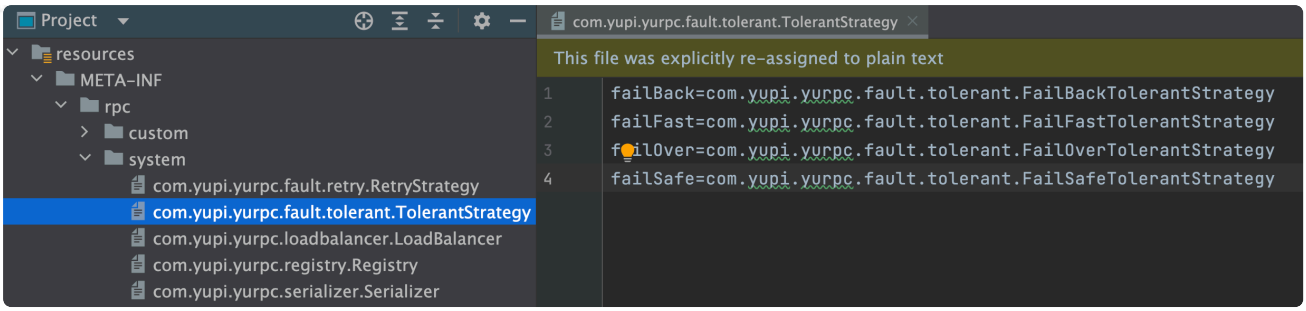
在 `fault.tolerant` 包下新建 `TolerantStrategyFactory` 类，代码如下：

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

import com.yupi.yurpc.spi.SpiLoader;

/**
 * 容错策略工厂（工厂模式，用于获取容错策略对象）
 *
 * @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 class TolerantStrategyFactory {

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

    /**
     * 默认容错策略
     */
    private static final TolerantStrategy DEFAULT_RETRY_STRATEGY = new FailFa

    /**
     * 获取实例
     *
     * @param key
     * @return
     */
    public static TolerantStrategy getInstance(String key) {
        return SpiLoader.getInstance(TolerantStrategy.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` 目录下编写容错策略接口的 SPI 配置文件，文件名称为 `com.yupi.yurpc.fault.tolerant.TolerantStrategy` 。

如图：



代码如下：

```
failBack=com.yupi.yurpc.fault.tolerant.FailBackTolerantStrategy
failFast=com.yupi.yurpc.fault.tolerant.FailFastTolerantStrategy
failOver=com.yupi.yurpc.fault.tolerant.FailOverTolerantStrategy
failSafe=com.yupi.yurpc.fault.tolerant.FailSafeTolerantStrategy
```

4）为 RpcConfig 全局配置新增容错策略的配置，代码如下：

```
@Data
public class RpcConfig {

    /**
     * 容错策略
     */
    private String tolerantStrategy = TolerantStrategyKeys.FAIL_FAST;
}
```

3、应用容错功能

容错功能的应用非常简单，我们只需要修改 ServiceProxy 的部分代码，在重试多次抛出异常时，从工厂中获取容错策略并执行即可。

修改的代码如下：

```
// rpc 请求
// 使用重试机制
RpcResponse rpcResponse;
try {
    RetryStrategy retryStrategy = RetryStrategyFactory.getInstance(rpcConfig.
    rpcResponse = retryStrategy.doRetry(() ->
            VertxTcpClient.doRequest(rpcRequest, selectedServiceMetaInfo)
    );
} catch (Exception e) {
    // 容错机制
    TolerantStrategy tolerantStrategy = TolerantStrategyFactory.getInstance(r
    rpcResponse = tolerantStrategy.doTolerant(null, e);
}
return rpcResponse.getData();
```

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
        // 构造请求
        String serviceName = method.getDeclaringClass().getName();
        RpcRequest rpcRequest = RpcRequest.builder()
                .serviceName(serviceName)
                .methodName(method.getName())
                .parameterTypes(method.getParameterTypes())
                .args(args)
                .build();

        // 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.getRegistry
        ServiceMetaInfo serviceMetaInfo = new ServiceMetaInfo();
        serviceMetaInfo.setServiceName(serviceName);
        serviceMetaInfo.setServiceVersion(RpcConstant.DEFAULT_SERVICE_VERSION
        List<ServiceMetaInfo> serviceMetaInfoList = registry.serviceDiscovery
        if (CollUtil.isEmpty(serviceMetaInfoList)) {
            throw new RuntimeException("暂无服务地址");
        }

        // 负载均衡
        LoadBalancer loadBalancer = LoadBalancerFactory.getInstance(rpcConfig
        // 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(request
        // rpc 请求
        // 使用重试机制
        RpcResponse rpcResponse;
        try {
```

```
            RetryStrategy retryStrategy = RetryStrategyFactory.getInstance(rp
            rpcResponse = retryStrategy.doRetry(() ->
                    VertxTcpClient.doRequest(rpcRequest, selectedServiceMetaI
            );
        } catch (Exception e) {
            // 容错机制
            TolerantStrategy tolerantStrategy = TolerantStrategyFactory.getIn
            rpcResponse = tolerantStrategy.doTolerant(null, e);
        }
        return rpcResponse.getData();
    }
}
```

我们会发现，即使引入了容错机制，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。等待多次重试后，就可以看到容错策略的执行。

五、扩展

1）实现 Fail-Back 容错机制。

参考思路：可以参考 Dubbo 的 Mock 能力，让消费端指定调用失败后要执行的本地服务和方法。

2）实现 Fail-Over 容错机制。

参考思路：可以利用容错方法的上下文参数传递所有的服务节点和本次调用的服务节点，选择一个其他节点再次发起调用。

3）实现更多容错方案。（较难）

参考思路：比如限流、熔断、超时控制等。或者将重试机制作为容错机制的一种策略来实现。