# 基本概念

Dubbo 是一款高性能、轻量级的开源 RPC 框架，提供服务自动注册、自动发现等高效服务治理方案，可以和 Spring 框架无缝集成。

# 使用场景

## 1、透明化的远程方法调用

就像调用本地方法一样调用远程方法，只需简单配置，没有任何 API的侵入。

## 2、软负载均衡及容错机制

可在内网替代 F5 等硬件负载均衡器，降低成本减少单点。

## 3、服务自动注册与发现

不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的 IP 地址，并且能够平滑添加或删除服务提供者。

# 核心功能

1、Remoting: 网络通信框架，提供对多种 NIO 框架抽象封装，包括“同步转

异步”和“请求-响应”模式的信息交换方式。

2、Cluster: 服务框架，提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持。

3、Registry: 服务注册，基于注册中心目录服务，使服务消费方能动态的找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

# 核心组件

## 1、Provider

暴露服务的服务提供方

## 2、Consumer

调用远程服务的消费方

## 3、Registry

服务注册与发现的注册中心

## 4、Monitor

监控中心和访问调用统计

## 5、Container

服务运行的容器

# Dubbo服务器注册与发现的流程

1、Provider (提供者)绑定指定端口并启动服务。

2、提供者连接注册中心，并发本机 IP、端口、应用信息和提供服务信息发送至注册中心存储。

3、Consumer (消费者)连接注册中心，并发送应用信息、所求服务信息至

注册中心。

4、注册中心根据消费者所求服务信息匹配对应的提供者列表发送至 Consumer应用缓存。

5、Consumer 在发起远程调用时基于缓存的消费者列表一起发起调用。

6、Provider 状态变更会实时通知注册中心，由注册中心实时推送至Consumer。

# Dubbo推荐什么协议

推荐使用dubbo协议

# Dubbo有哪些注册中心

Dubbo 能够支持的注册中心有：consul、etcd、nacos、sofa、zookeeper、 redis、multicast 等。 而一般Dubbo 的注册中心选用zookeeper，因为除了数据存储外，zookeeper 还提供了watcher 机制，即只要服务发生变化，调用方就能感知到。

# Dubbo 的注册中心集群挂掉，发布者和订阅者之间还能通信么

可以通讯。启动 Dubbo 时，消费者会从 Zookeeper 拉取注册的生产者的地址接口等数据，缓存在本地。每次调用时，按照本地存储的地址进行调用。

# Dubbo 使用的是什么通信框架

默认使用 Netty 作为通讯框架。

# Dubbo 集群提供了哪些负载均衡策略

# Dubbo 支持哪些序列化方式

# Dubbo 和 Spring Cloud 有什么哪些区别?

# Dubbo 和 Spring Cloud 有什么关系?

# RPC异常重试

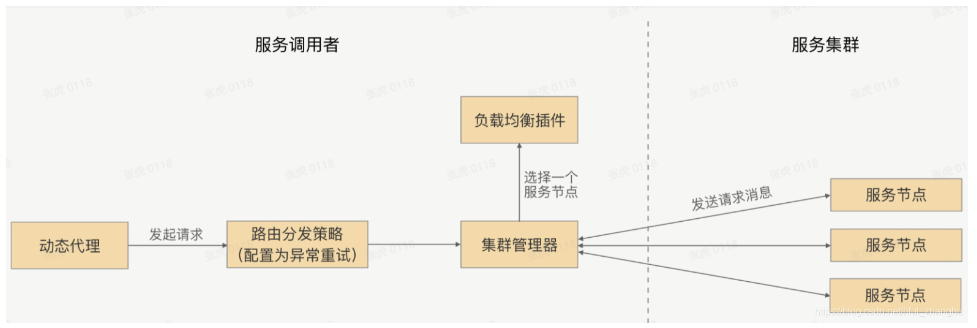
## 为什么需要异常重试

一次 RPC 调用，去调用远程的一个服务，比如用户的登录操作，会先对用户的用户名以及密码进行验证，验证成功之后会获取用户的基本信息。当通过远程的用户服务来获取用户基本信息的时候，恰好网络出现了问题，比如网络突然抖了一下，导致请求失败了，而这个请求希望它能够尽可能地执行成功，那这时要怎么做呢？

需要重新发起一次 RPC 调用，那代码中该如何处理呢？这时就可以考虑使用 RPC 框架的重试机制。

## RPC框架的重试机制

就是当调用端发起的请求失败时，RPC 框架自身可以进行重试，再重新发送请求，用户可以自行设置是否开启重试以及重试的次数。那这个机制是如何实现的呢？



调用端在发起 RPC 调用时，会经过负载均衡，选择一个节点，之后它会向这个节点发送请求信息。当消息发送失败或收到异常消息时，我们就可以捕获异常，根据异常触发重试，重新通过负载均衡选择一个节点发送请求消息，并且记录请求的重试次数，当重试次数达到用户配置的重试次数的时候，就返回给调用端动态代理一个失败异常，否则就一直重试下去。

RPC 框架的重试机制就是调用端发现请求失败时捕获异常，之后触发重试，那是不是所有的异常都要触发重试呢？当然不是了，因为这个异常可能是服务提供方抛回来的业务异常，它是应该正常返回给动态代理的，所以要在触发重试之前对捕获的异常进行判定，只有符合重试条件的异常才能触发重试，比如网络超时异常、网络连接异常等。了解了 RPC 框架的重试机制，那用户在使用异常重试时需要注意哪些问题呢？当网络突然抖动了一下导致请求超时了，但这个时候调用方的请求信息可能已经发送到服务提供方的节点上，也可能已经发送到服务提供方的服务节点上，那如果请求信息成功地发送到了服务节点上，那这个节点就要执行业务逻辑了。

那如果这个时候发起了重试，业务逻辑是否会被执行呢？会的。那如果这个服务业务逻辑不是幂等的，比如插入数据操作，那触发重试的话会不会引发问题呢？会的。

总结出：在使用 RPC 框架的时候，要确保被调用的服务的业务逻辑是幂等的，这样才能考虑根据事件情况开启 RPC 框架的异常重试功能。这一点你要格外注意，这算是一个高频误区了。

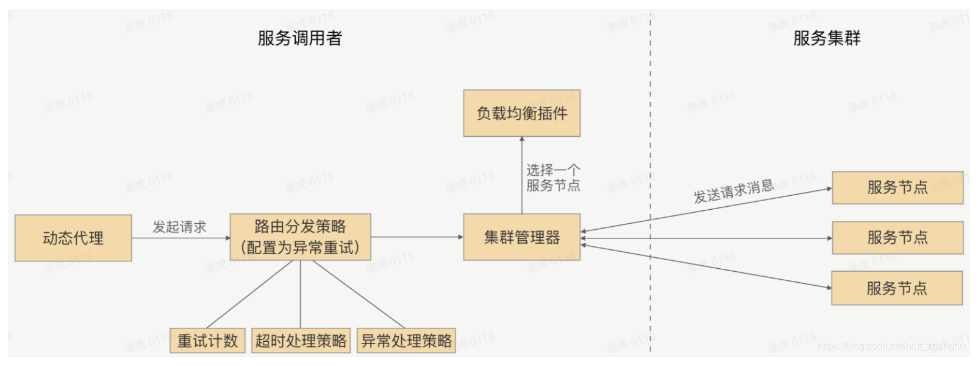
## 如何在约定时间内安全可靠地重试

连续的异常重试可能会出现一种不可靠的情况，那就是连续的异常重试并且每次处理的请求时间比较长，最终会导致请求处理的时间过长，超出用户设置的超时时间。解决这个问题最直接的方式就是，在每次重试后都重置一下请求的超时时间。当调用端发起 RPC 请求时，如果发送请求发生异常并触发了异常重试，可以先判定下这个请求是否已经超时，如果已经超时了就直接返回超时异常，否则就先重置下这个请求的超时时间，之后再发起重试。

当调用端设置了异常重试策略，发起了一次 RPC 调用，通过负载均衡选择了节点，将请求消息发送到这个节点，这时这个节点由于负载压力较大，导致这个请求处理失败了，调用端触发了重试，再次通过负载均衡选择了一个节点，结果恰好仍选择了这个节点，那么在这种情况下，重试的效果是否受影响了呢？当然有影响。因此，需要在所有发起重试、负载均衡选择节点的时候，去掉重试之前出现过问题的那个节点，以保证重试的成功率。

RPC 框架的异常重试机制，是调用端发送请求之后，如果发送失败会捕获异常，触发重试，但并不是所有的异常都会触发重试的，只有 RPC 框架中特定的异常才会如此，比如连接异常、超时异常。而像服务端业务逻辑中抛回给调用端的异常是不能重试的。那么请你想一下这种情况：服务端的业务逻辑抛给调用端一个异常信息，而服务端抛出这个异常是允许调用端重新发起一次调用的。比如这个场景：服务端的业务逻辑是对数据库某个数据的更新操作，更新失败则抛出个更新失败的异常，调用端可以再次调用，来触发服务端重新执行更新操作。那这个时候对于调用端来说，它接收到了更新失败异常，虽然是服务端抛回来的业务异常，但也是可以进行重试的。

RPC 框架是不会知道哪些业务异常能够去进行异常重试的，我们可以加个重试异常的白名单，用户可以将允许重试的异常加入到这个白名单中。当调用端发起调用，并且配置了异常重试策略，捕获到异常之后，我们就可以采用这样的异常处理策略。如果这个异常是 RPC 框架允许重试的异常，或者这个异常类型存在于可重试异常的白名单中，我们就允许对这个请求进行重试。



## 总结

RPC 框架的重试机制，还有如何在约定时间内进行安全可靠地重试。这个机制是当调用端发起的请求失败时，如果配置了异常重试策略，RPC 框架会捕捉异常，对异常进行判定，符合条件则进行重试，重新发送请求。

在重试的过程中，为了能够在约定的时间内进行安全可靠地重试，在每次触发重试之前，需要先判定下这个请求是否已经超时，如果超时了会直接返回超时异常，否则我们需要重置下这个请求的超时时间，防止因多次重试导致这个请求的处理时间超过用户配置的超时时间，从而影响到业务处理的耗时。

在发起重试、负载均衡选择节点的时候，应该去掉重试之前出现过问题的那个节点，这样可以提高重试的成功率，并且允许用户配置可重试异常的白名单，这样可以让 RPC 框架的异常重试功能变得更加友好。在使用 RPC 框架的重试机制时，确保被调用的服务的业务逻辑是幂等的，这样才能考虑是否使用重试，这一点至关重要。