9. 重试机制 - 手写 RPC 框架项目教程 - 编程导航教程

> 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、需求分析现在我们的 RPC 框架已经可以从注册中。

一、需求分析

现在我们的 RPC 框架已经可以从注册中心获取到服务提供者的注册信息了，同一个服务可能会有多个服务提供者，但是目前我们消费者始终读取了第一个服务提供者节点发起调用，不仅会增大单个节点的压力，而且没有利用好其他节点的资源。

我们完全可以从服务提供者节点中，选择一个服务提供者发起请求，而不是每次都请求同一个服务提供者，这个操作就叫做 负载均衡。

本节教程，我们就为 RPC 框架支持服务消费者的负载均衡。

二、负载均衡

什么是负载均衡?

让我们把这个词拆开来看：

1）何为负载？可以把负载理解为要处理的工作和压力，比如网络请求、事务、数据处理任务等。

2）何为均衡？把工作和压力平均地分配给多个工作者，从而分摊每个工作者的压力，保证大家正常工作。

用个比喻，假设餐厅里只有一个服务员，如果顾客非常多，他可能会忙不过来，没法及时上菜、忙中生乱；而且他的压力会越来越大，最严重的情况下就累倒了无法继续工作。而如果有多个服务员，大家能够服务更多的顾客，即使有一个服务员生病了，其他服务员也能帮忙顶上。

所以，负载均衡是一种用来分配网络或计算负载到多个资源上的技术。它的目的是确保每个资源都能够有效地处理负载、增加系统的并发量、避免某些资源过载而导致性能下降或服务不可用的情况。

回归到我们的 RPC 框架，负载均衡的作用是从一组可用的服务提供者中选择一个进行调用。

常用的负载均衡实现技术有 Nginx（七层负载均衡）、LVS（四层负载均衡）等。推荐阅读鱼皮之前写过的一篇负载均衡入门文章：
https://www.codefather.cn/%E4%BB%80%E4%B9%88%E6%98%AF%E8%B4%9F%E8%BD%BD%E5%9D%87%E8%A1%A1/

常见负载均衡算法

负载均衡学习的重点就是它的算法 —— 按照什么策略选择资源。

不同的负载均衡算法，适用的场景也不同，一定要根据实际情况选取，主流的负载均衡算法如下：

1）轮询（Round Robin）：按照循环的顺序将请求分配给每个服务器，适用于各服务器性能相近的情况。

假如有 5 台服务器节点，请求调用顺序如下：

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

2）随机（Random）：随机选择一个服务器来处理请求，适用于服务器性能相近且负载均匀的情况。

假如有 5 台服务器节点，请求调用顺序如下：

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

3）加权轮询（Weighted Round Robin）：根据服务器的性能或权重分配请求，性能更好的服务器会获得更多的请求，适用于服务器性能不均的情况。

假如有 1 台千兆带宽的服务器节点和 4 台百兆带宽的服务器节点，请求调用顺序可能如下：

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

4）加权随机（Weighted Random）：根据服务器的权重随机选择一个服务器处理请求，适用于服务器性能不均的情况。

假如有 2 台千兆带宽的服务器节点和 3 台百兆带宽的服务器节点，请求调用顺序可能如下：

```
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;
}
```

5）最小连接数（Least Connections）：选择当前连接数最少的服务器来处理请求，适用于长连接场景。

6）IP Hash：根据客户端 IP 地址的哈希值选择服务器处理请求，确保同一客户端的请求始终被分配到同一台服务器上，适用于需要保持会话一致性的场景。

当然，也可以根据请求中的其他参数进行 Hash，比如根据请求接口的地址路由到不同的服务器节点。

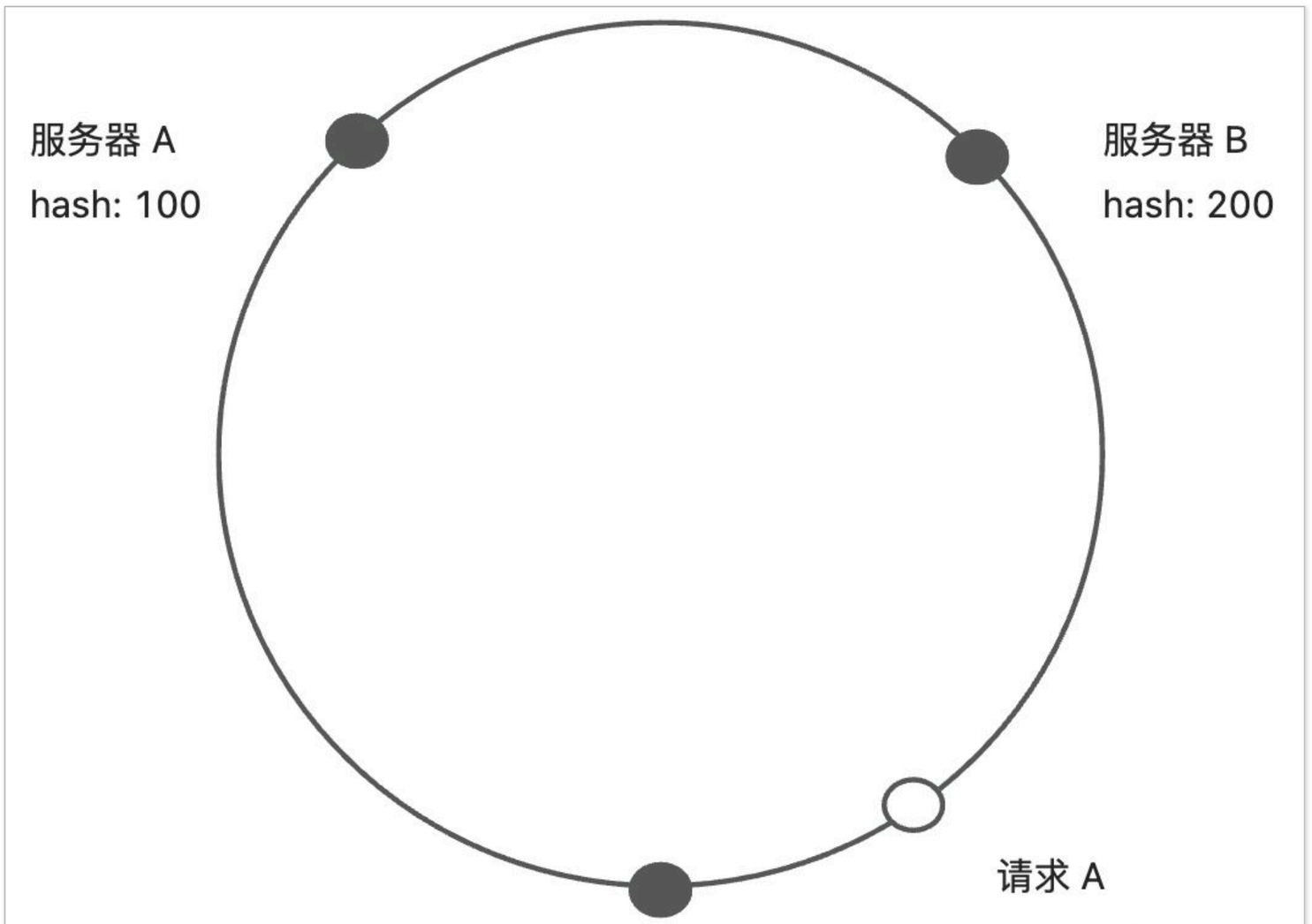
下面，再给大家分享一个很重要的分布式知识点：一致性 Hash。

一致性 Hash

一致性哈希（Consistent Hashing）是一种经典的哈希算法，用于将请求分配到多个节点或服务器上，所以非常适用于负载均衡。

它的核心思想是将整个哈希值空间划分成一个环状结构，每个节点或服务器在环上占据一个位置，每个请求根据其哈希值映射到环上的一个点，然后顺时针寻找第一个大于或等于该哈希值的节点，将请求路由到该节点上。

一致性哈希环结构如图：



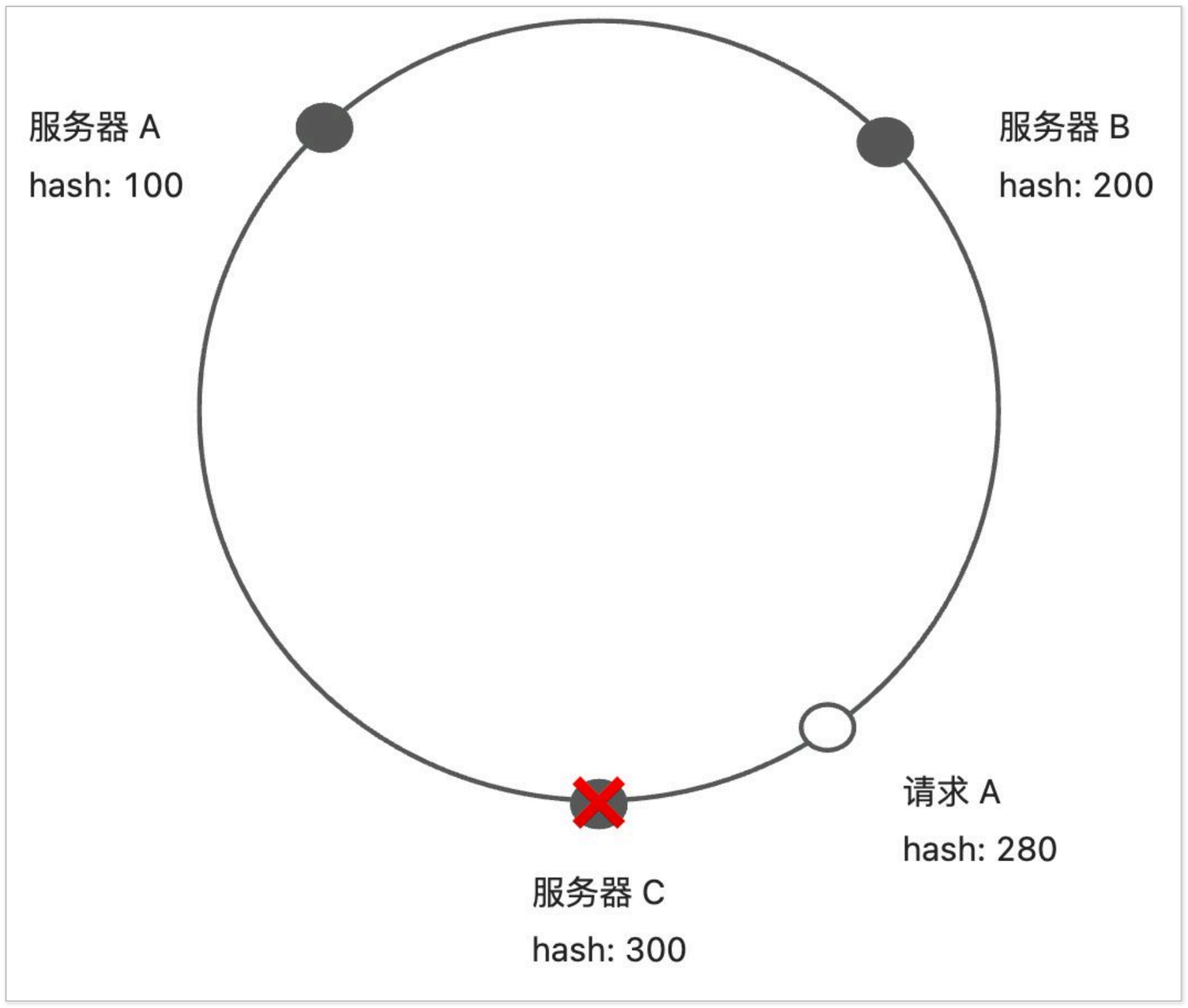
上图中，请求 A 会交给服务器 C 来处理。

好像也没什么特别的啊？还整个环？

其实，一致性哈希还解决了 节点下线 和 倾斜问题。

1）节点下线：当某个节点下线时，其负载会被平均分摊到其他节点上，而不会影响到整个系统的稳定性，因为只有部分请求会受到影响。

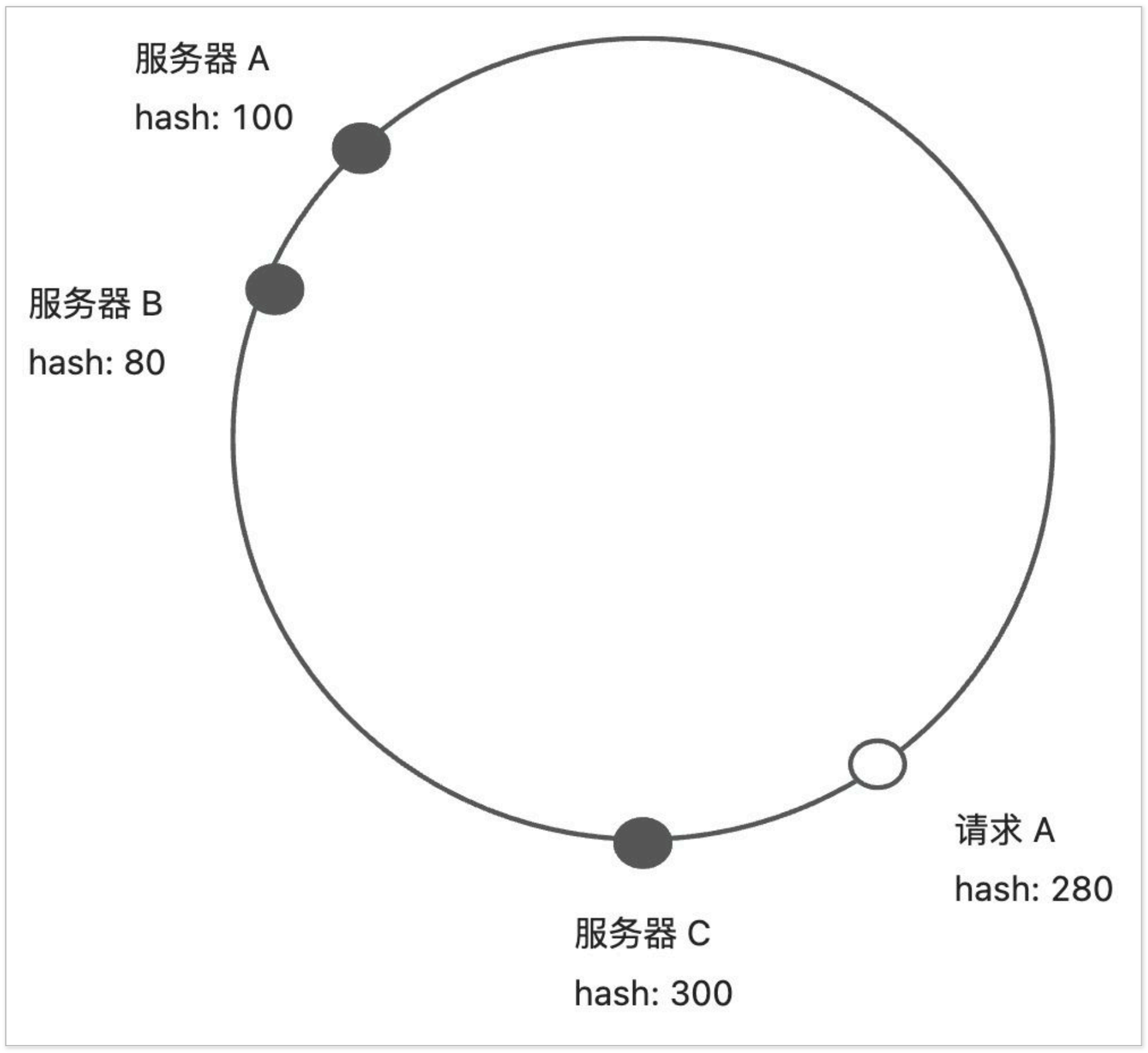
如下图，服务器 C 下线后，请求 A 会交给服务器 A 来处理（顺时针寻找第一个大于或等于该哈希值的节点），而服务器 B 接收到的请求保持不变。



如果是轮询取模算法，只要节点数变了，很有可能大多数服务器处理的请求都要发生变化，对系统的影响巨大。

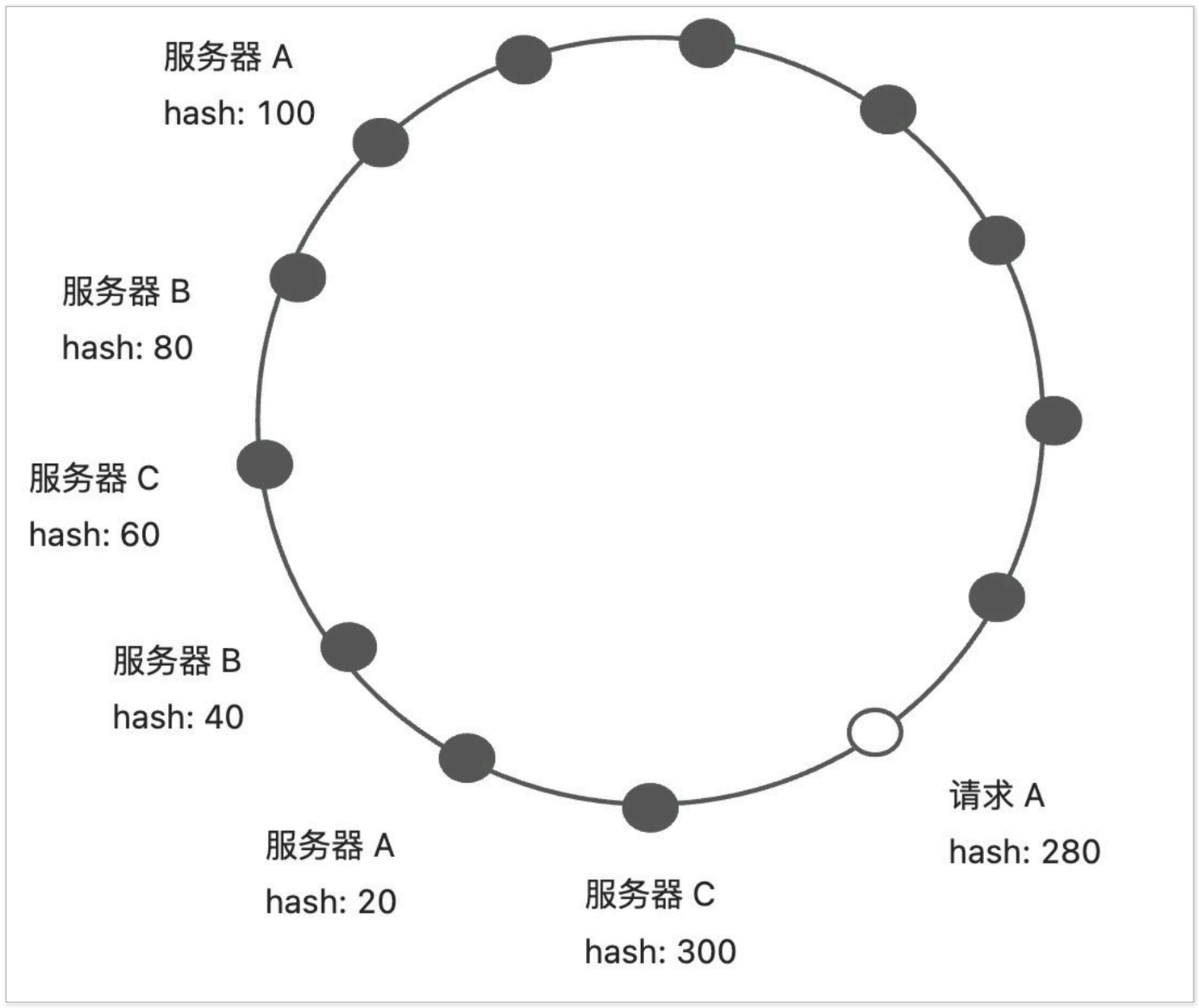
2）倾斜问题：通过虚拟节点的引入，将每个物理节点映射到多个虚拟节点上，使得节点在哈希环上的 分布更加均匀，减少了节点间的负载差异。

举个例子，节点很少的情况下，环的情况可能如下图：



这样就会导致绝大多数的请求都会发给服务器 C，而服务器 A 的“领地”非常少，几乎不会有请求。

引入虚拟节点后，环的情况变为：



这样一来，每个服务器接受到的请求会更容易平均。

理解了负载均衡算法后，我们来开发实现。

三、开发实现

1、多种负载均衡器实现

大家学习负载均衡的时候，可以参考 Nginx 的负载均衡算法实现，此处鱼皮带大家实现轮询、随机、一致性 Hash 三种负载均衡算法。

在 RPC 项目中新建 `loadbalancer` 包，将所有负载均衡器相关的代码放到该包下。

1）先编写负载均衡器通用接口。提供一个选择服务方法，接受请求参数和可用服务列表，可以根据这些信息进行选择。

代码如下：

```
<!-- https://github.com/rholder/guava-retrying -->
<dependency>
    <groupId>com.github.rholder</groupId>
    <artifactId>guava-retrying</artifactId>
    <version>2.0.0</version>
</dependency>
```

2）轮询负载均衡器。

使用 JUC 包的 `AtomicInteger` 实现原子计数器，防止并发冲突问题。

代码如下：

```
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 Exception {
        return callable.call();
    }

}
```

3）随机负载均衡器。

使用 Java 自带的 Random 类实现随机选取即可，代码如下：

```
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 ExecutionException, RetryException {
        Retryer<RpcResponse> retryer = RetryerBuilder.<RpcResponse>newBuilder()
                .retryIfExceptionOfType(Exception.class)
                .withWaitStrategy(WaitStrategies.fixedWait(3L, TimeUnit.SECONDS))
                .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);
    }

}
```

4）实现一致性 Hash 负载均衡器。

可以使用 TreeMap 实现一致性 Hash 环，该数据结构提供了 ceilingEntry 和 firstEntry 两个方法，便于获取符合算法要求的节点。

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();
        }
    }
}
```

上述代码中，注意两点：

1. 根据 requestParams 对象计算 Hash 值，这里鱼皮只是简单地调用了对象的 hashCode 方法，大家也可以根据需求实现自己的 Hash 算法。
2. 每次调用负载均衡器时，都会重新构造 Hash 环，这是为了能够即时处理节点的变化。

大家一定要自己手敲这段代码！

2、支持配置和扩展负载均衡器

一个成熟的 RPC 框架可能会支持多个负载均衡器，像序列化器和注册中心一样，我们的需求是，让开发者能够填写配置来指定使用的负载均衡器，并且支持自定义负载均衡器，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）负载均衡器常量。

在 loadbalancer 包下新建 `LoadBalancerKeys` 类，列举所有支持的负载均衡器键名。

代码如下：

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

```java
/**
 * 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 获取负载均衡器对象实例。

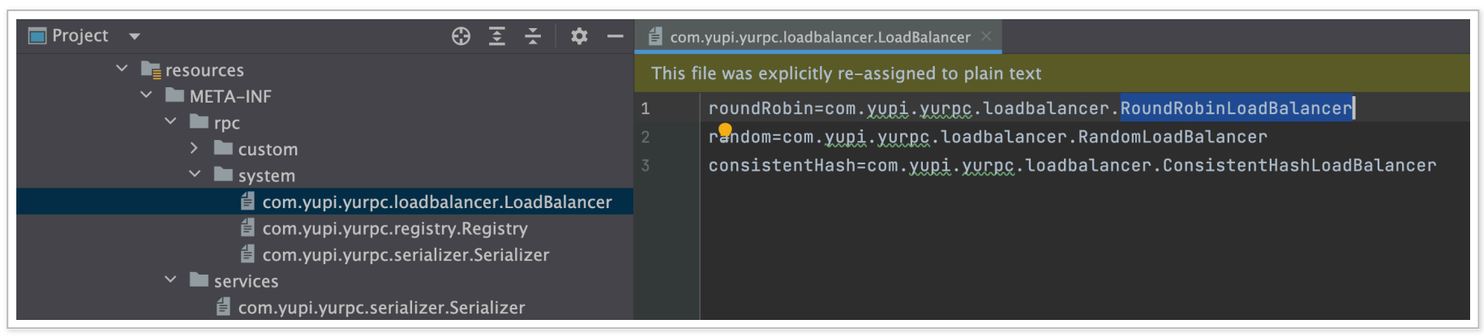
在 loadbalancer 包下新建 `LoadBalancerFactory` 类，代码如下：

```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 NoRetryStrategy();

    /**
     * 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` 目录下编写负载均衡器接口的 SPI 配置文件，文件名称为 `com.yupi.yurpc.loadbalancer.LoadBalancer` 。

如图：



代码如下：

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

4）为 RpcConfig 全局配置新增负载均衡器的配置，代码如下：

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

3、应用负载均衡器

现在，我们就能够愉快地使用负载均衡器了。修改 ServiceProxy 的代码，将“固定调用第一个服务节点”改为“调用负载均衡器获取一个服务节点”。

修改后的代码如下：

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

上述代码中，我们给负载均衡器传入了一个 requestParams HashMap，并且将请求方法名作为参数放到了 Map 中。如果使用的是一致性 Hash 算法，那么会根据 requestParams 计算 Hash 值，调用相同方法的请求 Hash 值肯定相同，所以总会请求到同一个服务器节点上。

四、测试

1、测试负载均衡算法

首先编写单元测试类 `LoadBalancerTest` ，代码如下：

```
/**
 * 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 Throwable {
        // 指定序列化器
        final Serializer serializer = SerializerFactory.getInstance(RpcApplication.getRpcConfig().getSerializer());

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

            // 负载均衡
            LoadBalancer loadBalancer = LoadBalancerFactory.getInstance(rpcConfig.getLoadBalancer());
            // 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(requestParams, serviceMetaInfoList);

            // rpc 请求
            // 使用重试机制
            RetryStrategy retryStrategy = RetryStrategyFactory.getInstance(rpcConfig.getRetryStrategy());
            RpcResponse rpcResponse = retryStrategy.doRetry(() ->
                    VertxTcpClient.doRequest(rpcRequest, selectedServiceMetaInfo)
            );
            return rpcResponse.getData();
        } catch (Exception e) {
            throw new RuntimeException("调用失败");
        }
    }
}
```

可以替换 loadBalancer 对象为不同的负载均衡器实现类，然后观察效果。

2、测试负载均衡调用

首先在不同的端口启动 2 个服务提供者，然后启动服务消费者项目，通过 Debug 或者控制台输出来观察每次请求的节点地址。

五、扩展

1）实现更多不同算法的负载均衡器

参考思路：比如最少活跃数负载均衡器，选择当前正在处理请求的数量最少的服务提供者。

2）自定义一致性 Hash 算法中的 Hash 算法

参考思路：比如根据请求客户端的 IP 地址来计算 Hash 值，保证同 IP 的请求发送给相同的服务提供者。