第四天

一、服务注册

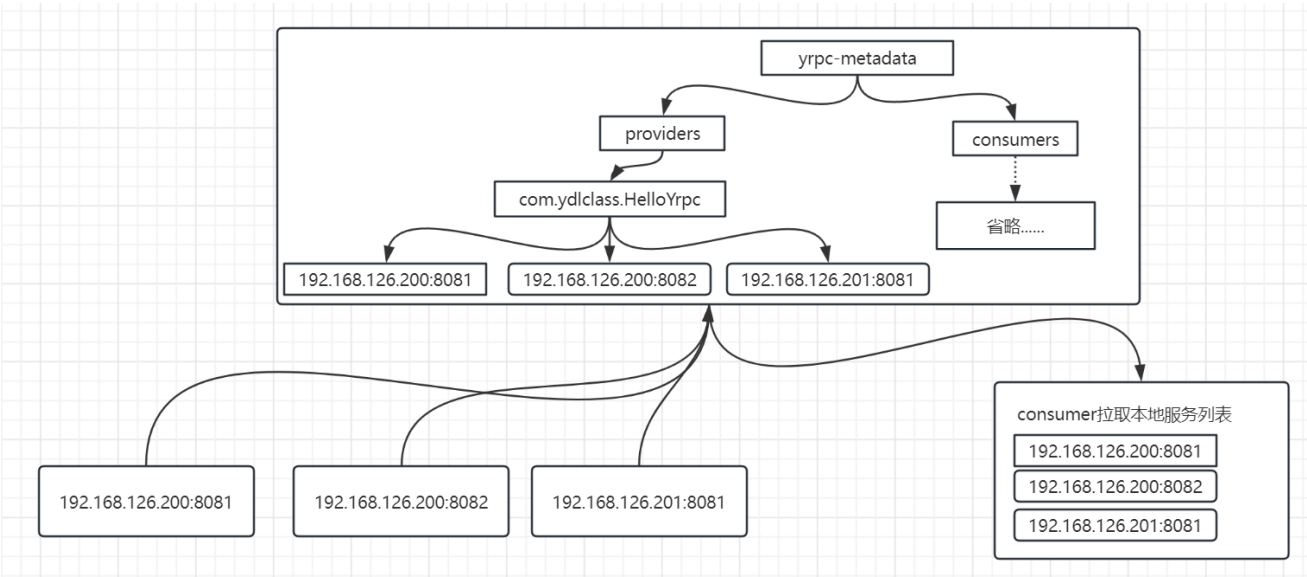
第四天我们主要完成服务注册和发现的功能，其具体流程如下：

1、服务提供方将服务注册到注册中心中。

2、消费端拉取服务列表。

3、消费端简单的选取一个可以服务（后续会进行改造，实现负载均衡）。

以上相关内容都使用zookeeper相关api实现，很简单，看视频敲代码即可。



为了让注册和拉取相对更加简单，我们封装了相应的工具类，代码如下：

```
@Slf4j
public class ZookeeperUtils {


    /**
     * 使用默认配置创建zookeeper实例
     * @return zookeeper实例
     */
    public static ZooKeeper createZookeeper(){
        // 定义连接参数
        String connectString = Constant.DEFAULT_ZK_CONNECT;
        // 定义超时时间
        int timeout = Constant.TIME_OUT;
        return createZookeeper(connectString,timeout);
    }
```

```
    public static ZooKeeper createZookeeper(String connectString,int timeout){
        CountDownLatch countDownLatch = new CountDownLatch(1);
        try {
            // 创建zookeeper实例，建立连接
            final ZooKeeper zooKeeper = new ZooKeeper(connectString, timeout, event ->
{
                // 只有连接成功才放行
                if (event.getState() == Watcher.Event.KeeperState.SyncConnected) {
                    log.debug("客户端已经连接成功。");
                    countDownLatch.countDown();
                }
            });

            countDownLatch.await();
            return zooKeeper;
        } catch (IOException  | InterruptedException e) {
            log.error("创建zookeeper实例时发生异常：",e);
            throw new ZookeeperException();
        }
    }

    /**
     * 创建一个节点的工具方法
     * @param zooKeeper zooKeeper实例
     * @param node 节点
     * @param watcher watcher实例
     * @param createMode 节点的类型
     * @return true：成功创建  false：已经存在  异常：抛出
     */
    public static Boolean createNode(ZooKeeper zooKeeper,ZookeeperNode node,Watcher
watcher,CreateMode createMode){
        try {
            if (zooKeeper.exists(node.getNodePath(), watcher) == null) {
                String result = zooKeeper.create(node.getNodePath(), node.getData(),
                    ZooDefs.Ids.OPEN_ACL_UNSAFE, createMode);
                log.info("节点【{}】，成功创建。",result);
                return true;
            } else {
                if(log.isDebugEnabled()){
                    log.info("节点【{}】已经存在，无需创建。",node.getNodePath());
                }
                return false;
            }
        } catch (KeeperException| InterruptedException e) {
            log.error("创建基础目录时发生异常：",e);
            throw new ZookeeperException();
        }
    }

    /**
     * 判断节点是否存在
     * @param zk zk实例
     * @param node  节点路径
```

```
     * @param watcher watcher
     * @return ture 存在 | false 不存在
     */
    public static boolean exists(ZooKeeper zk,String node,Watcher watcher){
        try {
            return zk.exists(node,watcher) != null;
        } catch (KeeperException | InterruptedException e) {
            log.error("判断节点[{}]是否存在时发生异常",node,e);
            throw new ZookeeperException(e);
        }
    }

    /**
     * 关闭zookeeper的方法
     * @param zooKeeper zooKeeper实例
     */
    public static void close(ZooKeeper zooKeeper){
        try {
            zooKeeper.close();
        } catch (InterruptedException e) {
            log.error("关闭zookeeper时发生问题: ",e);
            throw new ZookeeperException();
        }
    }

    /**
     * 查询一个节点的子元素
     * @param zooKeeper zk实例
     * @param serviceNode 服务节点
     * @return 子元素列表
     */
    public static List<String> getChildren(ZooKeeper zooKeeper, String
serviceNode,Watcher watcher) {
        try {
            return zooKeeper.getChildren(serviceNode, watcher);
        } catch (KeeperException | InterruptedException e) {
            log.error("获取节点【{}】的子元素时发生异常.",serviceNode,e);
            throw new ZookeeperException(e);
        }
    }
}
```

二、抽象能力

在这个功能中，我们可能会思考一些问题，没错，在当前项目中我们的确使用的是zookeeper作为我们项目的注册中心。但是，我们希望在我们的项目是**可以扩展使用其他类型的注册中心的**，如nacos，redis，甚至是自己独立开发注册中心。当然，在这个样例工程中，我们不会做这些事情，但是我们必须留出口子，为后来的扩展提供可能，这就是面向对象编程的魅力。所以在整个工程中我们再也不能单独的面向具体的对象编程，而是面向抽象，我们将抽象出【注册中心】整个抽象的概念，代码如下：

```java
public interface Registry {

    /**
     * 注册服务
     * @param serviceConfig 服务的配置内容
     */
    void register(ServiceConfig<?> serviceConfig);

    /**
     * 从注册中心拉取服务列表
     * @param serviceName 服务的名称
     * @return 服务的地址
     */
    List<InetSocketAddress> lookup(String serviceName,String group);

}
```

目录结构：

针对注册中心的核心能力，我们抽象出简洁核心的接口，这样需要其他的扩展仅仅需要完成独立的实现即可完成。

关于zookeeper的具体实现，我们就不在这里详细介绍了，大家在代码中都能看到。

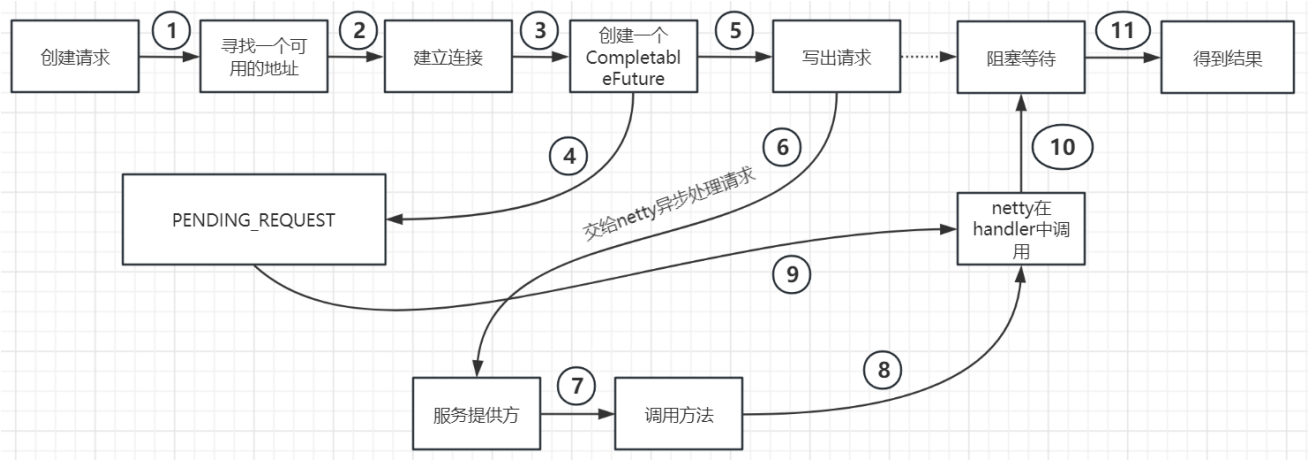
第五天

一、异步

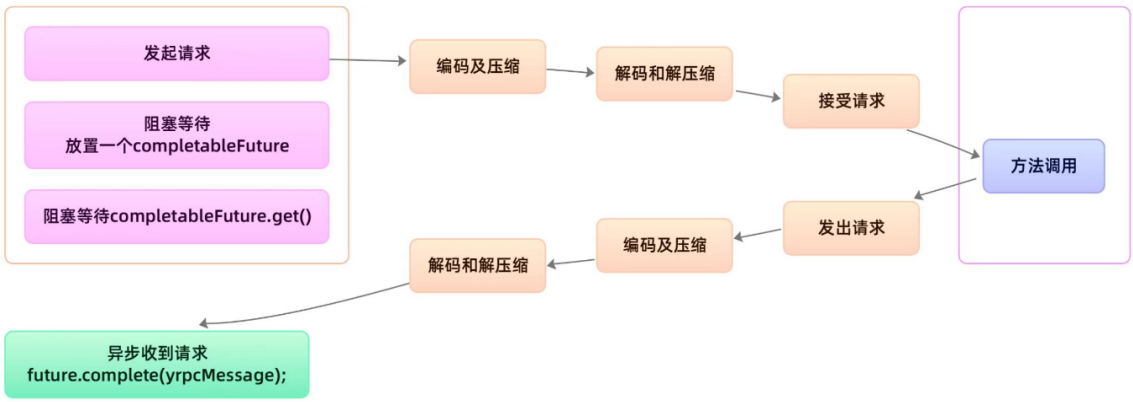
经过了第四天的学习，我们的消费端已经可以获取一个可用的连接地址，接下来要做的就是**建立消费端和服务提供方的之间的长连接**，并且进行通信。但因为netty在整个通信的过程中是**异步**的，所以我们会使用CompletableFuture来获取异步的结果。

关于CompletableFuture的用法有一个博主的文章介绍的很好，我们可以看看：https://zhuanlan.zhihu.com/p/344431341。

我画了一个简单的流程图如下，他展示了一个从创建请求到得到响应的整个流程安装。



还有一个图，你们觉得能看懂哪个算哪个？哈哈



在以上的过程中，请求发送出去，即一旦调用了writeAndFlush方法，其中网络通信、方法调用等一系列的工作就都和当前线程无关了，我们只能使用CompletableFuture.get()方法等待结果。

二、代理模式

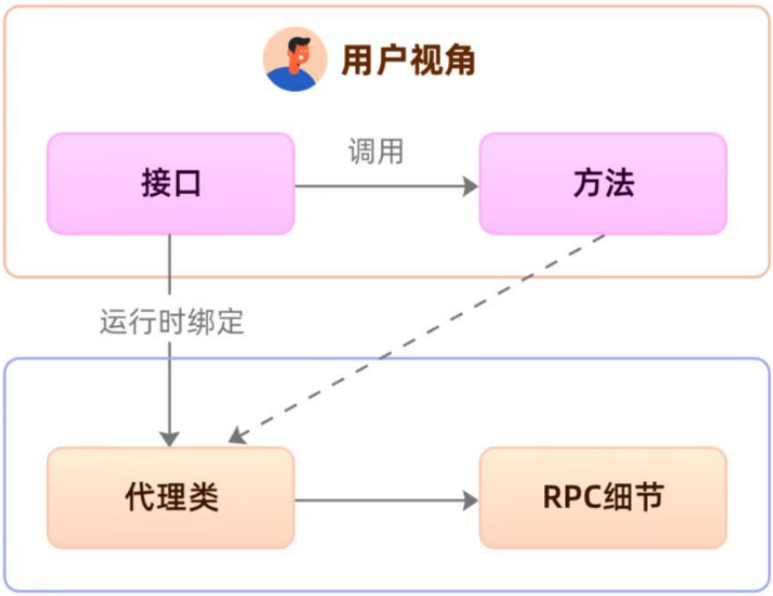
yrpc 是用来解决两个应用之间的通信，而网络则是两台机器之间的“桥梁”，只有架好了桥梁，我们才能把请求数据从一端传输另外一端。其实关于网络通信，你只要记住一个关键字就行了——可靠的传输。

对于服务端和客户端，他们做的事情都很确定：

服务端：暴露接口，等待客户端的远程访问，执行方法，返回结果。

客户端：引入接口，实现接口，在实现中编写网络请求代码和结果处理代码。

我们一定会发现，对于客户端而言，其中涉及的过程如 **封装请求、选择通道、等待响应**等功能（事实上，这个方法的功能远不止于此，后期我们还会开发**异常重试、熔断保护、负载均衡**等）都是一样的，我们不可能为每一个方法调用都编写相同的逻辑。仔细思考，这是不是在**给方法调用做增强。谈及增强我们可能第一时间想起了代理模式和装饰器模式。**



当然心细的同学可能发现了，我们目前是**既没有被代理对象，也没有被装饰的类，有的只是一个孤零零的接口。**这种情况无论是静态代理、还是装饰器都没有用武之地，只有**动态代理可以大展身手**，可以在运行期凭空捏造生成一个代理对象。

动态代理模式的详细使用大家可以看我的设计模式课程，当然百度也有，可以自行搜索。生成代理对象代码如下：

```
public T get() {
    ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
    Class<T>[] classes = new Class[]{interfaceRef};
    InvocationHandler handler = new
RpcConsumerInvocationHandler(registry,interfaceRef,group);

    // 使用动态代理生成代理对象
    Object helloProxy = Proxy.newProxyInstance(classLoader, classes, handler);

    return (T) helloProxy;
}
```

本质上调用代理对象的方法会最终落实到RpcConsumerInvocationHandler的invoke方法，并且会将方法对象、参数列表传入：

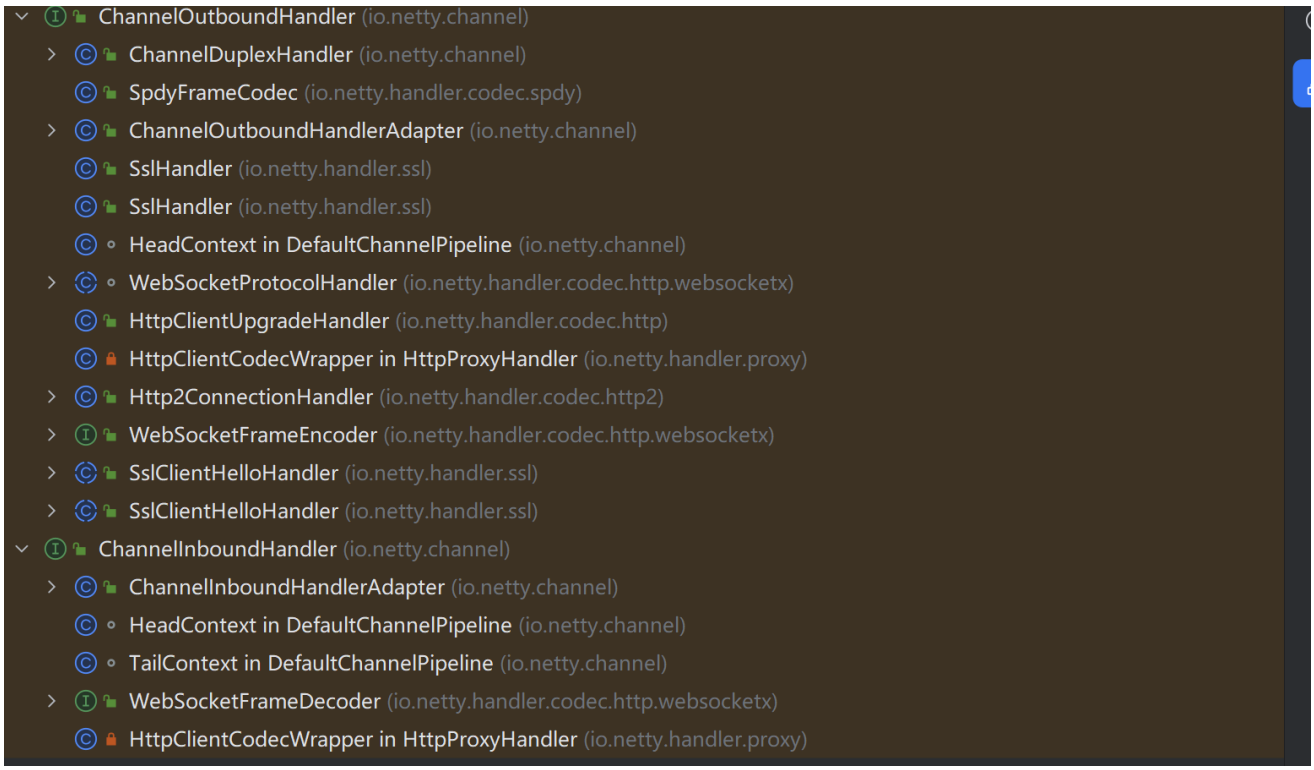
```
public class RpcConsumerInvocationHandler implements InvocationHandler {

    // 此处需要一个注册中心，和一个接口
    private final Registry registry;
    private final Class<?> interfaceRef;

    public RpcConsumerInvocationHandler(Registry registry, Class<?> interfaceRef) {
        this.registry = registry;
        this.interfaceRef = interfaceRef;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
```

```
            // 1、利用方法、参数列表封装请求负载
            // 2、封装请求对象
            // 3、选取一个服务提供方
            // 4、建立连接
            // 5、写出请求
            // 6、等待响应（网络通信和方法调用是异步的）
            // 7、获得结果返回
        }
    }
```

更详细的代码可以在源码中察看。

三、netty的pipeline

在netty中请求的处理都是使用的IO多路复用，同时他提供了非常友好的请求处理方式就是pipeline（流水线）。他提供了基本的**入站和出站**的能力，并且抽象了两个接口ChannelInboundHandler（入栈处理器）和ChannelOutboundHandler（出栈处理器），当然他们共同继承自ChannelHandler接口，同时为我们实现了**大量的通用的入站和出站处理器**。其图如下：



当我们在消费端调用writeAndFlush方法时，网络通信就开始了，大致的流程如下：

1、对于消费方，开始做**出站的工作**，中间会经历多个出站处理器，主要的核心逻辑是将**请求对象封装成报文**。

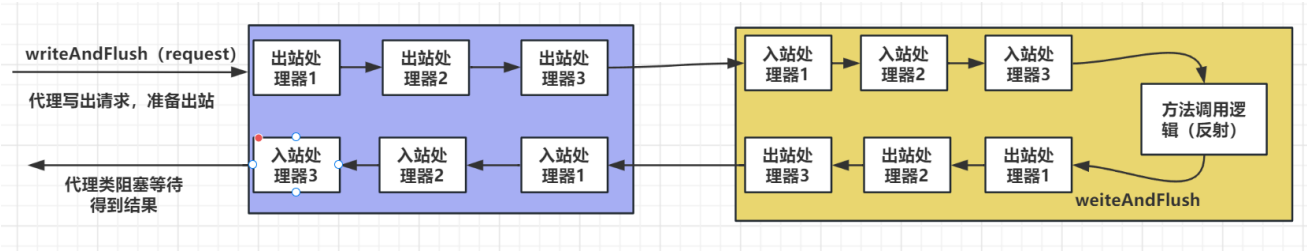
2、消息经过消费方的出站处理程序后就变成了二进制字节流报文，就会进入服务提供方，开始进入**提供方的入站逻辑，核心就是解析请求报文**。

3、得到请求的之后，提供方根据请求携带的负载选定合适的对象和方法进行**方法调用**，得到结果。

4、调用方开始封装响应，并调用writeAndFlush将响应写出，进入提供方的出站逻辑，主要就是**封装响应报文**。

5、调用方接受响应，进入入站逻辑，**解析响应，得到结果。**

下图可以清晰的表达出整个流程：



在这其中我们可以使用netty提供的原生的出入站处理器，也可以自定义，比如我们**用来解析和封装请求以及响应的处理器是需要自己编写的。**

服务端的pipeline如下，这里我们通过addLast方法添加了四个出入站处理程序：

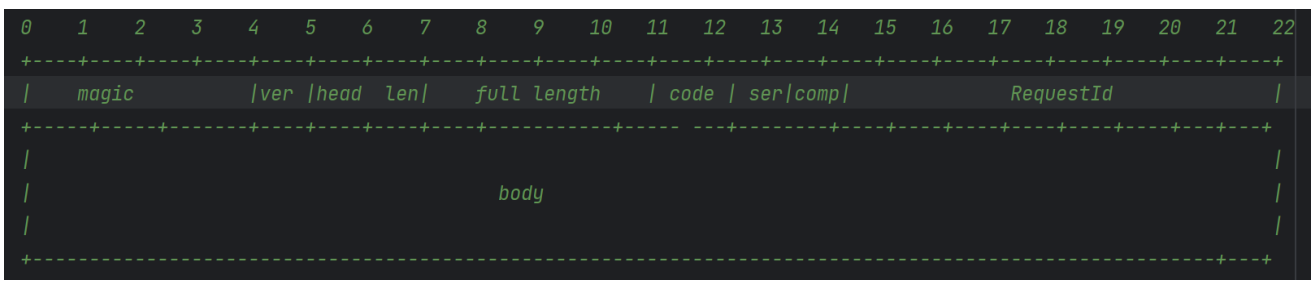
```
serverBootstrap = serverBootstrap.group(boss, worker)
    .channel(NioServerSocketChannel.class)
    .childHandler(new ChannelInitializer<SocketChannel>() {
        @Override
        protected void initChannel(SocketChannel socketChannel) throws Exception {
            // 是核心，我们需要添加很多入站和出站的handler
            socketChannel.pipeline()
                .addLast(new LoggingHandler())
                .addLast(new YrpcRequestDecoder())
                // 根据请求进行方法调用
                .addLast(new MethodCallHandler())
                .addLast(new YrpcResponseEncoder());
        }
    });
```

第六天

一、解析请求

基础的通讯的逻辑代码已经完成了，封装请求的代码如下，关于设计私有协议的内容我们在上一个课件中就讲述了，当然其中还包括对**负载的序列化和压缩**的代码逻辑：

报文格式如下：



将请求对象进行编码的代码如下，我们继承了MessageToByteEncoder类，而该类也是ChannelOutboundHandler的一个实现，用来处理出站逻辑：

```java
/**
 * 4B magic(魔数)   --->yrpc.getBytes()
 * 1B version(版本)   ----> 1
 * 2B header length 首部的长度
 * 4B full length 报文总长度
 * 1B serialize
 * 1B compress
 * 1B requestType
 * 8B requestId
 * body
 * 出站时，第一个经过的处理器
 * @author it楠老师
 * @createTime 2023-07-02
 */
@Slf4j
public class YrpcResponseEncoder extends MessageToByteEncoder<YrpcResponse> {

    @Override
    protected void encode(ChannelHandlerContext channelHandlerContext, YrpcResponse
yrpcResponse, ByteBuf byteBuf) throws Exception {
        // 4个字节的魔数值
        byteBuf.writeBytes(MessageFormatConstant.MAGIC);
        // 1个字节的版本号
        byteBuf.writeByte(MessageFormatConstant.VERSION);
        // 2个字节的头部的长度
        byteBuf.writeShort(MessageFormatConstant.HEADER_LENGTH);
        // 总长度不清楚，不知道body的长度 writeIndex(写指针)
        byteBuf.writerIndex(byteBuf.writerIndex() +
MessageFormatConstant.FULL_FIELD_LENGTH);
        // 3个类型
        byteBuf.writeByte(yrpcResponse.getCode());
        byteBuf.writeByte(yrpcResponse.getSerializeType());
        byteBuf.writeByte(yrpcResponse.getCompressType());
        // 8字节的请求id
        byteBuf.writeLong(yrpcResponse.getRequestId());
        byteBuf.writeLong(yrpcResponse.getTimeStamp());

        // 1、对响应做序列化
        byte[] body = null;
        if(yrpcResponse.getBody() != null) {
            Serializer serializer = SerializerFactory
                .getSerializer(yrpcResponse.getSerializeType()).getImpl();
            body = serializer.serialize(yrpcResponse.getBody());

            // 2、压缩
            Compressor compressor = CompressorFactory.getCompressor(
                yrpcResponse.getCompressType()
            ).getImpl();
            body = compressor.compress(body);
        }

        if(body != null){
            byteBuf.writeBytes(body);
```

```
        }
        int bodyLength = body == null ? 0 : body.length;

        // 重新处理报文的总长度
        // 先保存当前的写指针的位置
        int writerIndex = byteBuf.writerIndex();
        // 将写指针的位置移动到总长度的位置上
        byteBuf.writerIndex(MessageFormatConstant.MAGIC.length
            + MessageFormatConstant.VERSION_LENGTH +
MessageFormatConstant.HEADER_FIELD_LENGTH
        );
        byteBuf.writeInt(MessageFormatConstant.HEADER_LENGTH + bodyLength);
        // 将写指针归位
        byteBuf.writerIndex(writerIndex);

        if(log.isDebugEnabled()){
            log.debug("响应【{}】已经在服务端完成编码工作。",yrpcResponse.getRequestId());
        }
    }
}
```

经过了这个处理器，我们的**请求对象就变成了报文**，发送给服务提供方，提供方需要解析报文，解析过程如下，其中需要对负载进行**解压缩和反序列化**：

```
public class YrpcResponseDecoder extends LengthFieldBasedFrameDecoder {
    public YrpcResponseDecoder() {
        super(
            // 找到当前报文的总长度，截取报文，截取出来的报文我们可以去进行解析
            // 最大帧的长度，超过这个maxFrameLength值会直接丢弃
            MessageFormatConstant.MAX_FRAME_LENGTH,
            // 长度的字段的偏移量，
            MessageFormatConstant.MAGIC.length + MessageFormatConstant.VERSION_LENGTH +
MessageFormatConstant.HEADER_FIELD_LENGTH,
            // 长度的字段的长度
            MessageFormatConstant.FULL_FIELD_LENGTH,
            // todo 负载的适配长度
            -(MessageFormatConstant.MAGIC.length + MessageFormatConstant.VERSION_LENGTH
              + MessageFormatConstant.HEADER_FIELD_LENGTH +
MessageFormatConstant.FULL_FIELD_LENGTH),
            0);
    }

    @Override
    protected Object decode(ChannelHandlerContext ctx, ByteBuf in) throws Exception {
        Object decode = super.decode(ctx, in);
        if(decode instanceof ByteBuf byteBuf){
            return decodeFrame(byteBuf);
        }
        return null;
    }

    private Object decodeFrame(ByteBuf byteBuf) {
        // 1、解析魔数
```

```
        byte[] magic = new byte[MessageFormatConstant.MAGIC.length];
        byteBuf.readBytes(magic);
        // 检测魔数是否匹配
        for (int i = 0; i < magic.length; i++) {
            if(magic[i] != MessageFormatConstant.MAGIC[i]){
                throw new RuntimeException("The request obtained is not legitimate。");
            }
        }

        // 2、解析版本号
        byte version = byteBuf.readByte();
        if(version > MessageFormatConstant.VERSION){
            throw new RuntimeException("获得的请求版本不被支持。");
        }

        // 3、解析头部的长度
        short headLength = byteBuf.readShort();

        // 4、解析总长度
        int fullLength = byteBuf.readInt();

        // 5、请求类型
        byte responseCode = byteBuf.readByte();

        // 6、序列化类型
        byte serializeType = byteBuf.readByte();

        // 7、压缩类型
        byte compressType = byteBuf.readByte();

        // 8、请求id
        long requestId = byteBuf.readLong();

        // 9、时间戳
        long timeStamp = byteBuf.readLong();

        // 我们需要封装
        YrpcResponse yrpcResponse = new YrpcResponse();
        yrpcResponse.setCode(responseCode);
        yrpcResponse.setCompressType(compressType);
        yrpcResponse.setSerializeType(serializeType);
        yrpcResponse.setRequestId(requestId);
        yrpcResponse.setTimeStamp(timeStamp);

        // todo 心跳请求没有负载，此处可以判断并直接返回
        //        if( requestType == RequestType.HEART_BEAT.getId()){
        //            return yrpcRequest;
        //        }

        int bodyLength = fullLength - headLength;
        byte[] payload = new byte[bodyLength];
        byteBuf.readBytes(payload);
```

```
        if(payload.length > 0) {
            // 有了字节数组之后就可以解压缩，反序列化
            // 1、解压缩
            Compressor compressor =
CompressorFactory.getCompressor(compressType).getImpl();
            payload = compressor.decompress(payload);

            // 2、反序列化
            Serializer serializer = SerializerFactory
                .getSerializer(yrpcResponse.getSerializeType()).getImpl();
            Object body = serializer.deserialize(payload, Object.class);
            yrpcResponse.setBody(body);
        }

        if(log.isDebugEnabled()){
            log.debug("响应【{}】已经在调用端完成解码工作。",yrpcResponse.getRequestId());
        }

        return yrpcResponse;
    }

}
```

二、使用反射完成方法调用

服务提供方通过解析请求报文，可以获得负载数据，并根据负载数据进行方法调用，负载内容如下：

```
public class RequestPayload implements Serializable {

    // 1、接口的名字 -- com.ydlclass.HelloYrpc
    private String interfaceName;

    // 2、方法的名字 --sayHi
    private String methodName;

    // 3、参数列表，参数分为参数类型和具体的参数
    // 参数类型用来确定重载方法，具体的参数用来执行方法调用
    private Class<?>[] parametersType;  // -- {java.long.String}
    private Object[] parametersValue;   // -- "你好"

    // 4、返回值的封装 -- {java.long.String}
    private Class<?> returnType;
}
```

其中包含接口的名字，方法名字，参数类型列表，和参数列表、返回值类型几个核心参数。当我们拥有了这些核心参数之后，我们应该思考，服务提供方应该如何使用这些数据进行方法调用，很明显，我们能直接想到的技术就是【反射】。

在这其中主要涉及两个过程：

> 提供方接受到请求之后整体的执行流程如下：

1、获取和调用方的连接（channel）。

2、从请求中获取请求所携带的负载。

3、根据负载进行方法调用。

4、写出结果（writeAndFlush）到调用方。

> 根据负载进行方法调用的过程，callTargetMethod(RequestPayload requestPayload)

1、从负载中获取核心参数。

2、从已经发布的服务中获取具体的实例。（在发布服务时，会将一个具体的实例进行缓存）

3、使用反射进行方法调用。

具体的代码实现如下：

```
public class MethodCallHandler extends SimpleChannelInboundHandler<YrpcRequest> {
    @Override
    protected void channelRead0(ChannelHandlerContext channelHandlerContext,
YrpcRequest yrpcRequest) throws Exception {

        // 1、 获得通道
        Channel channel = channelHandlerContext.channel();


        // 2、获取负载内容
        RequestPayload requestPayload = yrpcRequest.getRequestPayload();

        // 3、根据负载内容进行方法调用
        try {
            Object result = callTargetMethod(requestPayload);
            if (log.isDebugEnabled()) {
                log.debug("请求【{}】已经在服务端完成方法调用。",
yrpcRequest.getRequestId());
            }
            // 4、封装响应
            YrpcResponse yrpcResponse = new YrpcResponse();
            yrpcResponse.setRequestId(yrpcRequest.getRequestId());
            yrpcResponse.setCompressType(yrpcRequest.getCompressType());
            yrpcResponse.setSerializeType(yrpcRequest.getSerializeType());
            yrpcResponse.setCode(RespCode.SUCCESS.getCode());
            yrpcResponse.setBody(result);
        } catch (Exception e){
            log.error("编号为【{}】的请求在调用过程中发生异
常。",yrpcRequest.getRequestId(),e);
            yrpcResponse.setCode(RespCode.FAIL.getCode());
        }


        // 5、写出响应
        channel.writeAndFlush(yrpcResponse);
    }
```

```
    private Object callTargetMethod(RequestPayload requestPayload) {
        String interfaceName = requestPayload.getInterfaceName();
        String methodName = requestPayload.getMethodName();
        Class<?>[] parametersType = requestPayload.getParametersType();
        Object[] parametersValue = requestPayload.getParametersValue();

        // 寻找到匹配的暴露出去的具体的实现
        ServiceConfig<?> serviceConfig = YrpcBootstrap.SERVERS_LIST.get(interfaceName);
        Object refImpl = serviceConfig.getRef();

        // 通过反射调用 1、获取方法对象  2、执行invoke方法
        Object returnValue;
        try {
            Class<?> aClass = refImpl.getClass();
            Method method = aClass.getMethod(methodName, parametersType);
            returnValue = method.invoke(refImpl, parametersValue);
        } catch (InvocationTargetException | NoSuchMethodException |
IllegalAccessException e) {
            log.error("调用服务【{}】的方法【{}】时发生了异常。", interfaceName, methodName,
e);
            throw new RuntimeException(e);
        }
        return returnValue;
    }
}
```

第七天

一、雪花算法

在当前项目中，我们需要给请求一个唯一标识，用来标识一个请求和响应的关联关系，我们要求请求的id必须唯一，且不能占用过大的空间，可用的方案如下：

1、自增id，单机的自增id不能解决不重复的问题，微服务情况下我们需要一个稳定的发号服务才能保证，但是这样做性能偏低。

2、uuid，将uuid作为唯一标识占用空间太大

3、雪花算法，最优解。接下来我们简单地介绍一下雪花算法，同时带着大家一起手写雪花算法。

1、简介

雪花算法(snowflake)最早是twitter内部使用分布式环境下的唯一ID生成算法，他使用64位long类型的数据存储id，具体如下：

> 0 - 0000000000 0000000000 0000000000 0000000000 0 - 0000000000 - 000000000000
>
> 符号位 时间戳 机器码 序列号

最高位表示符号位，其中0代表整数，1代表负数，而id一般都是正数，所以最高位为0。**当然知道了这个理论，我们甚至可以自由设定属于我们自己的雪花算法。**

- 41位存储毫秒级时间戳，**这个时间戳不是存储当前时间的时间戳，而是存储时间戳的差值（当前时间戳 - 开始时间戳) * 得到的值）**，这样我们可以存储一个相对更长的时间。
- 10位存储机器码，最多支持1024台机器，当并发量非常高，同时有多个请求在同一毫秒到达，可以根据机器码进行第二次生成。机器码可以根据实际需求进行二次划分，比如两个机房操作可以一个机房分配5位机器码。
- 12位存储序列号，当同一毫秒有多个请求访问到了同一台机器后，此时序列号就派上了用场，为这些请求进行第三次创建，最多每毫秒每台机器产生2的12次方也就是4096个id，满足了大部分场景的需求。

总的来说雪花算法有以下几个优点：

- 能满足高并发分布式系统环境下ID不重复
- 基于时间戳，可以保证基本有序递增
- 不依赖第三方的库或者中间件
- 生成效率极高

2、手撸雪花

我们了解了雪花算法的的一些基础知识后就可以独立的编写一个属于自己的雪花算法了，我们可能并没有完全按照雪花算法的定义去实现，但是这正是程序的魅力，如我们的机房不需要那么多我们可以将时间戳的位数变的多一点，但事实上雪花算法是经过twitter等公司进行的最佳实践：

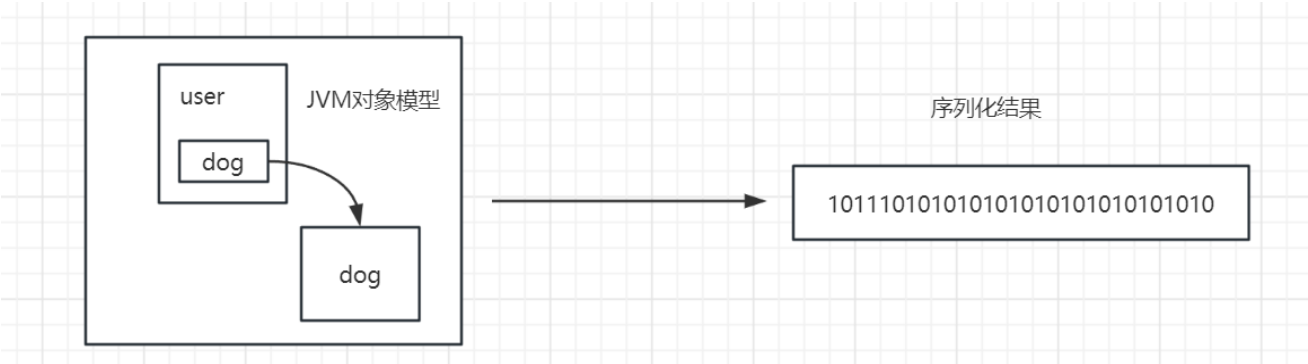
```
public class IdGenerator {

    // 雪花算法 -- 世界上没有一个片雪花是一样的
    // 机房号（数据中心） 5bit  32
    // 机器号            5bit  32
    // 时间戳（long 1970-1-1） 原本64位表示的时间，
    // 现在由41位构成，（我们整个案例中使用了42位，实时上有点问题，会出现负数，所有要看我们的业务需求）
    // 自由选择一个比较近的时间，比如我们公司成立的时间戳，项目启动的时间戳等
    // 同一个机房的同一个机器号的同一个时间可以因为并发量很大需要多个id
    // 序列号   12bit      5+5+42+12 = 64

    // 起始时间戳
    public static final long START_STAMP = DateUtil.get("2022-1-1").getTime();
    //
    public static final long DATA_CENTER_BIT = 5L;
    public static final long MACHINE_BIT = 5L;
    public static final long SEQUENCE_BIT = 12L;

    // 最大值 Math.pow(2,5) -1
    public static final long DATA_CENTER_MAX = ~(-1L << DATA_CENTER_BIT);
    public static final long MACHINE_MAX = ~(-1L << MACHINE_BIT);
    public static final long SEQUENCE_MAX = ~(-1L << SEQUENCE_BIT);


    // 时间戳 （42） 机房号 （5） 机器号 （5） 序列号 （12）
    // 10101010101010101010101010101010101010101011 10101 10101 101011010101
```

```
    public static final long TIMESTAMP_LEFT = DATA_CENTER_BIT + MACHINE_BIT +
SEQUENCE_BIT;
    public static final long DATA_CENTER_LEFT = MACHINE_BIT + SEQUENCE_BIT;
    public static final long MACHINE_LEFT = SEQUENCE_BIT;

    private long dataCenterId;
    private long machineId;
    private LongAdder sequenceId = new LongAdder();
    // 时钟回拨的问题，我们需要去处理
    private long lastTimeStamp = -1L;

    public IdGenerator(long dataCenterId, long machineId) {
        // 判断传世的参数是否合法
        if(dataCenterId > DATA_CENTER_MAX || machineId > MACHINE_MAX){
            throw new IllegalArgumentException("你传入的数据中心编号或机器号不合法.");
        }
        this.dataCenterId = dataCenterId;
        this.machineId = machineId;
    }

    public long getId(){
        // 第一步：处理时间戳的问题
        long currentTime = System.currentTimeMillis();

        long timeStamp = currentTime - START_STAMP;

        // 判断时钟回拨
        if(timeStamp < lastTimeStamp){
            throw new RuntimeException("您的服务器进行了时钟回调.");
        }

        // sequenceId需要做一些处理，如果是同一个时间节点，必须自增
        if (timeStamp == lastTimeStamp){
            sequenceId.increment();
            if(sequenceId.sum() >= SEQUENCE_MAX){
                timeStamp = getNextTimeStamp();
                sequenceId.reset();
            }
        } else {
            sequenceId.reset();
        }

        // 执行结束将时间戳赋值给lastTimeStamp
        lastTimeStamp = timeStamp;
        long sequence = sequenceId.sum();
        return timeStamp << TIMESTAMP_LEFT |  dataCenterId << DATA_CENTER_LEFT
            | machineId << MACHINE_LEFT | sequence;

    }

    private long getNextTimeStamp() {
        // 获取当前的时间戳
        long current = System.currentTimeMillis() - START_STAMP;
```

```
            // 如果一样就一直循环，直到下一个时间戳
            while (current == lastTimeStamp){
                current = System.currentTimeMillis() - START_STAMP;
            }
            return current;
        }

        public static void main(String[] args) {
            IdGenerator idGenerator = new IdGenerator(1,2);
            for (int i = 0; i < 1000; i++) {
                new Thread(() -> System.out.println(idGenerator.getId())).start();
            }
        }

    }
```

二、序列化

在需要传输的内容中我们封装了负载（payload），这是一个java对象，其实整个request就是一个java对象，我们需要将它转化为一个二进制字节流用于网络传输，这个过程就是序列化。



有人可能会问：jvm中的对象本来就是以二进制的形式存储的呀？

jvm中的对象是使用java的对象模型存储的，一个对象包含了对象头、成员变量的值、class指针、指向其他对象的引用等，这其中很多数据是不连续、而且这种模型只有java中可以被识别。如上图，这里会涉及很多的问题？

1、如何标记user属于什么类型？

2、user对象模型中的对象头等信息怎么处理？

3、user对象中指向其他对象的地址，以及dog对象怎么存储？即使存储了传递到其他的环境还能被识别吗？

所以，我们必须按照一套特定的逻辑对以上的问题进行解决，指定出一套序列化的方式，比如json将对象转化为可识别的字符序列来存储和传输，再比如很多常见的高性能的序列化方式hessian、protobuf等，都是提供的自己的序列化方案，以及现跨语言的相关实现。

1、jdk的序列化方式

jdk的序列化方式就是使用IO进行序列化，他只支持不同的jvm之间的传输，并不能跨语言，代码如下：

```
public class JdkSerializer implements Serializer {
    @Override
    public byte[] serialize(Object object) {
        if (object == null) {
            return null;
        }

        try (
            // 将流的定义写在这里会自动关闭，不需要在写finally
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            ObjectOutputStream outputStream = new ObjectOutputStream(baos);
        ) {
            outputStream.writeObject(object);

            byte[] result = baos.toByteArray();
            if(log.isDebugEnabled()){
                log.debug("对象【{}】已经完成了序列化操作，序列化后的字节数为
【{}】",object,result.length);
            }
            return result;
        } catch (IOException e) {
            log.error("序列化对象【{}】时放生异常.",object);
            throw new SerializeException(e);
        }
    }

    @Override
    public <T> T deserialize(byte[] bytes, Class<T> clazz) {
        if(bytes == null || clazz == null){
            return null;
        }
        try (
            // 将流的定义写在这里会自动关闭，不需要在写finally
            ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
            ObjectInputStream objectInputStream = new ObjectInputStream(bais);
        ) {
             Object object = objectInputStream.readObject();
            if(log.isDebugEnabled()){
                log.debug("类【{}】已经完成了反序列化操作.",clazz);
            }
            return (T)object;
        } catch (IOException | ClassNotFoundException e) {
            log.error("反序列化对象【{}】时发生异常.",clazz);
            throw new SerializeException(e);
        }
    }
}
```

2、hessian序列化

Hessian序列化是一种支持动态类型、跨语言、基于对象传输的网络协议，Java对象序列化的二进制流可以被其他语言（如，c++，python）。特性如下：

- **自描述序列化类型。**不依赖外部描述文件或者接口定义，用一个字节表示常用的基础类型，极大缩短二进制流。
- 语言无关，支持脚本语言
- 协议简单，比Java原生序列化高效 相比hessian1，hessian2中增加了压缩编码，其序列化二进制流大小是Java序列化的50%，序列化耗时是Java序列化的30%，反序列化耗时是Java序列化的20%。

样例代码如下：

```
public class HessianSerializer implements Serializer {
    @Override
    public byte[] serialize(Object object) {
        if (object == null) {
            return null;
        }
        try (
            // 将流的定义写在这里会自动关闭，不需要在写finally
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ) {
            Hessian2Output hessian2Output = new Hessian2Output(baos);
            hessian2Output.writeObject(object);
            hessian2Output.flush();
            byte[] result = baos.toByteArray();
            if(log.isDebugEnabled()){
                log.debug("对象【{}】已经完成了序列化操作，序列化后的字节数为
【{}】",object,result.length);
            }
            return result;
        } catch (IOException e) {
            log.error("使用hessian进行序列化对象【{}】时发生异常.",object);
            throw new SerializeException(e);
        }
    }

    @Override
    public <T> T deserialize(byte[] bytes, Class<T> clazz) {
        if(bytes == null || clazz == null){
            return null;
        }
        try (
            // 将流的定义写在这里会自动关闭，不需要在写finally
            ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
        ) {

            Hessian2Input hessian2Input = new Hessian2Input(bais);
            T t = (T) hessian2Input.readObject();
            if(log.isDebugEnabled()){
                log.debug("类【{}】已经使用hessian完成了反序列化操作.",clazz);
            }
            return t;
        } catch (IOException  e) {
```

```
            log.error("使用hessian进行反序列化对象【{}】时发生异常.",clazz);
            throw new SerializeException(e);
        }
    }
}
```

3、工厂模式的使用

为了让框架可以支持**更多的序列化方式**，我们设计了序列化器工厂，并对**不同的序列化器进行了缓存**，我们可以根据序列化的类型和编号轻松的获取一个序列化器。

```
public class SerializerFactory {

    private final static ConcurrentHashMap<String, ObjectWrapper<Serializer>>
SERIALIZER_CACHE = new ConcurrentHashMap<>(8);
    private final static ConcurrentHashMap<Byte, ObjectWrapper<Serializer>>
SERIALIZER_CACHE_CODE = new ConcurrentHashMap<>(8);

    static {
        ObjectWrapper<Serializer> jdk = new  ObjectWrapper<>((byte) 1, "jdk", new
JdkSerializer());
        ObjectWrapper<Serializer> json = new  ObjectWrapper<>((byte) 2, "json", new
JsonSerializer());
        ObjectWrapper<Serializer> hessian = new  ObjectWrapper<>((byte) 3, "hessian",
new HessianSerializer());
        SERIALIZER_CACHE.put("jdk",jdk);
        SERIALIZER_CACHE.put("json",json);
        SERIALIZER_CACHE.put("hessian",hessian);

        SERIALIZER_CACHE_CODE.put((byte) 1, jdk);
        SERIALIZER_CACHE_CODE.put((byte) 2, json);
        SERIALIZER_CACHE_CODE.put((byte) 3, hessian);
    }

    /**
     * 使用工厂方法获取一个SerializerWrapper
     * @param serializeType 序列化的类型
     * @return SerializerWrapper
     */
    public static  ObjectWrapper<Serializer> getSerializer(String serializeType) {
        ObjectWrapper<Serializer> serializerWrapper =
SERIALIZER_CACHE.get(serializeType);
        if(serializerWrapper == null){
            log.error("未找到您配置的【{}】序列化工具，默认选用jdk的序列化方
式。",serializeType);
            return SERIALIZER_CACHE.get("jdk");
        }

        return SERIALIZER_CACHE.get(serializeType);
    }
```

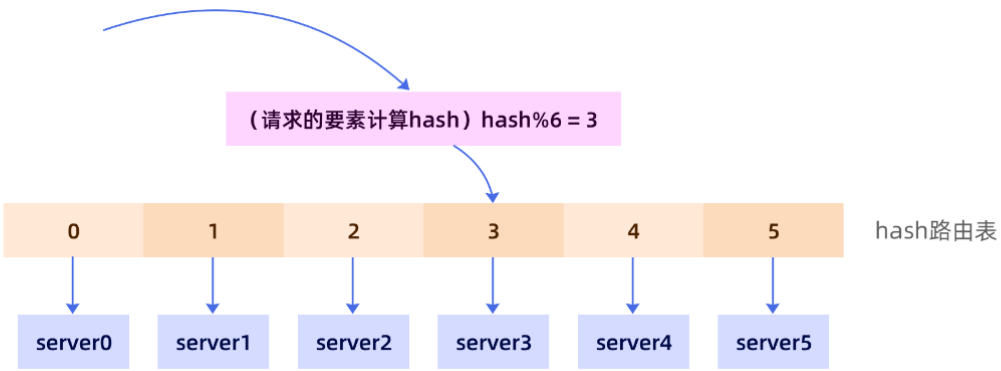
```
    public static  ObjectWrapper<Serializer> getSerializer(Byte serializeCode) {
        ObjectWrapper<Serializer> serializerWrapper =
SERIALIZER_CACHE_CODE.get(serializeCode);
        if(serializerWrapper == null){
            log.error("未找到您配置的【{}】序列化工具，默认选用jdk的序列化方
式。",serializeCode);
            return SERIALIZER_CACHE.get("jdk");
        }
        return SERIALIZER_CACHE_CODE.get(serializeCode);
    }

    /**
     * 新增一个新的序列化器
     * @param serializerObjectWrapper 序列化器的包装
     */
    public static void addSerializer(ObjectWrapper<Serializer> serializerObjectWrapper)
{

 SERIALIZER_CACHE.put(serializerObjectWrapper.getName(),serializerObjectWrapper);

 SERIALIZER_CACHE_CODE.put(serializerObjectWrapper.getCode(),serializerObjectWrapper);
    }

}
```

第八天

今天我们主要完成负载均衡，关于负载均衡的问题我们其实我们在第一个文档中已经介绍的很详细了，今天我们主要是完成负载均衡相关的一些算法实现。常见的负载均衡策略由轮询、加权轮询、随机、一致性hash、最短响应时间、最少连接数等，我们选取**轮询、一致性hash、最短响应时间**这三个比较具有代表性的算法一起来实现一下：

一、一致性hash

在yrpc中我们可以使用一致性hash算法来解决负载均衡的问题，当然在学习一致性hash之前我们需要了解普通的hash算法有什么问题，按照传统的hash算法思路，我们需要构建一张hash表，将服务器挂载在hash表中，如下图：



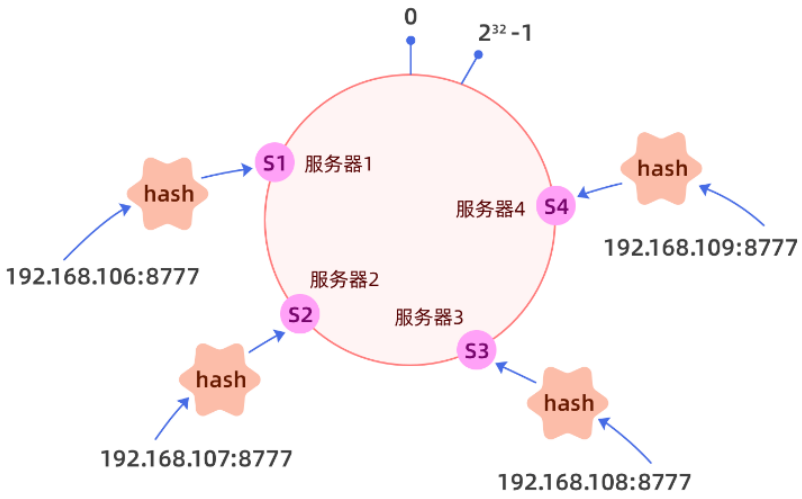
但是，这样的方式会存在很多问题，如动态扩容的问题。比如，随着业务量增长，将原有的六个服务扩容至八个，此时，我们不仅要修改路由表，还要修改hash的路由策略。

一致性hash借鉴了hash算法的部分能力做了如下的设计，

1、将hash值均匀的分布在一个区间，我们一般将区间设置为整形的取值范围（-2^{31} ~ 2^{31}-1）当然这个范围也可以是（0 ~ 2^{32}-1），只要是一个合理的容易计算的足够大的范围即可。

2、将这个区间构建成一个环，构建成环不一定必须要链表，其实很多的有序的数据结构都可以，比如数组，比如红黑树，只要加上一点点逻辑，就是数完最后一个回到第一个节点就可以了。

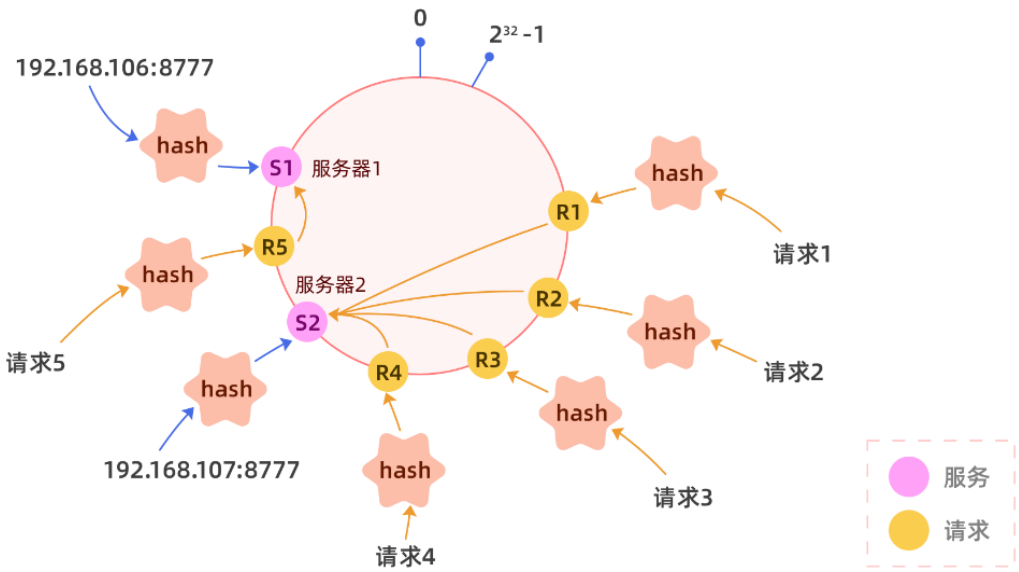
3、将服务器按照自身的特点，计算hash值，并将其挂载在hash表中。



那形成了这样的hash环后我们我们应该如何进行操作呢?

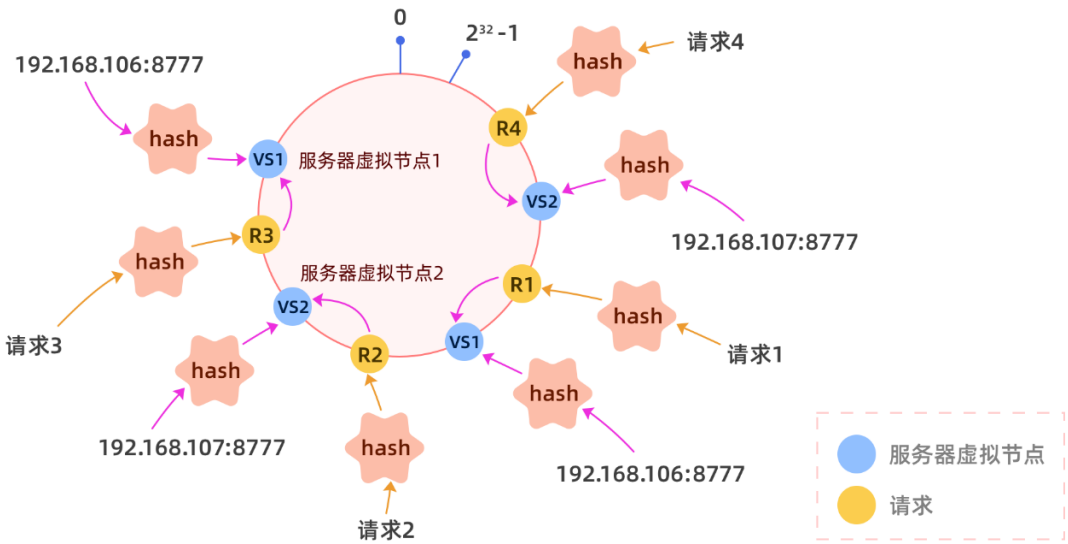
当请求进来以后，根据请求的部分特征，如url、请求id，请求来源等信息进行hash运算，看请求落在哪个范围，**然后顺时针找到第一个服务器**即可，这样最大的好处就是**当有新的服务加入集群只需要将服务挂载在hash环即**可，但是后自然会有流量进入该服务器，而不需要修改任何的逻辑，因为我们的hash环足够大，所以可以容纳的机器也很多。

问题：但是此时会出现一个问题，如果节点过少，hash分布不均匀会产生严重的流量倾斜：



为了解决这个问题，我们就需要**引入虚拟节点**的概念，我们可以将一个**真实节点化身为n个（比如128）虚拟节点**，每个虚拟节点都指向同一个服务，分别对虚拟节点进行hash，可以让一个服务的虚拟节点大致均匀的分布在hash环上，不要觉得他很神奇，代码实现其实很简单。当然虚拟节点数也可以乘以每个服务单位权重。

还是只有两个节点：



以下是代码实现：

```
@Slf4j
public class ConsistentHashBalancer extends AbstractLoadBalancer {

    @Override
```

```
    protected Selector getSelector(List<InetSocketAddress> serviceList) {
        return new ConsistentHashSelector(serviceList,128);
    }

    /**
     * 一致性hash的具体算法实现
     */
    private static class ConsistentHashSelector implements Selector{

        // hash环用来存储服务器节点
        private SortedMap<Integer,InetSocketAddress> circle= new TreeMap<>();
        // 虚拟节点的个数
        private int virtualNodes;

        public ConsistentHashSelector(List<InetSocketAddress> serviceList,int
virtualNodes) {
            // 我们应该尝试将节点转化为虚拟节点，进行挂载
            this.virtualNodes = virtualNodes;
            for (InetSocketAddress inetSocketAddress : serviceList) {
                // 需要把每一个节点加入到hash环中
                addNodeToCircle(inetSocketAddress);
            }
        }

        @Override
        public InetSocketAddress getNext() {
            // 1、hash环已经建立好了，接下来需要对请求的要素做处理我们应该选择什么要素来进行hash运算
            // 有没有办法可以获取，到具体的请求内容  --> threadLocal
            YrpcRequest yrpcRequest = YrpcBootstrap.REQUEST_THREAD_LOCAL.get();

            // 我们想根据请求的一些特征来选择服务器  id
            String requestId = Long.toString(yrpcRequest.getRequestId());

            // 请求的id做hash，字符串默认的hash不太好
            int hash = hash(requestId);

            // 判断该hash值是否能直接落在一个服务器上，和服务器的hash一样
            if( !circle.containsKey(hash)){
                // 寻找离我最近的一个节点
                SortedMap<Integer, InetSocketAddress> tailMap = circle.tailMap(hash);
                hash = tailMap.isEmpty() ? circle.firstKey() : tailMap.firstKey();
            }

            return circle.get(hash);
        }

        /**
         * 将每个节点挂载到hash环上
         * @param inetSocketAddress 节点的地址
         */
        private void addNodeToCircle(InetSocketAddress inetSocketAddress) {
            // 为每一个节点生成匹配的虚拟节点进行挂载
            for (int i = 0; i < virtualNodes; i++) {
```

```
            int hash = hash(inetSocketAddress.toString() + "-" + i);
            // 关在到hash环上
            circle.put(hash,inetSocketAddress);
            if(log.isDebugEnabled()){
                log.debug("hash为[{}]的节点已经挂载到了哈希环上.",hash);
            }
        }
    }

    private void removeNodeFromCircle(InetSocketAddress inetSocketAddress) {
        // 为每一个节点生成匹配的虚拟节点进行挂载
        for (int i = 0; i < virtualNodes; i++) {
            int hash = hash(inetSocketAddress.toString() + "-" + i);
            // 挂载到hash环上
            circle.remove(hash);
        }
    }

    /**
     * 具体的hash算法，todo 小小的遗憾，这样也是不均匀的
     * @param s
     * @return
     */
    private int hash(String s) {
        MessageDigest md;
        try {
            md = MessageDigest.getInstance("MD5");
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
        byte[] digest = md.digest(s.getBytes());
        // md5得到的结果是一个字节数组，但是我们想要int 4个字节

        int res = 0;
        for (int i = 0; i < 4; i++) {
            res = res << 8;
            if( digest[i] < 0 ){
                res = res | (digest[i] & 255);
            } else {
                res = res | digest[i];
            }
        }
        return res;
    }

    private String toBinary(int i){
        String s = Integer.toBinaryString(i);
        int index = 32 -s.length();
        StringBuilder sb = new StringBuilder();
        for (int j = 0; j < index; j++) {
            sb.append(0);
        }
        sb.append(s);
```

```
                return sb.toString();
            }
        }
    }
```

二、心跳检测

事实上netty框架本身可以实现探活，可以**自动感知channel的连接状态**。但是我们为了和大家一起体验一把（说实话就是写的时候不知道），特意将相关的内容手动实现了一下。

其核心原理十分简单，就是定期向所有的channel发送一个简单的请求即可，如果能得到回应说明连接是正常的。

其中我们要在心跳探测的过程中完成以下几项工作：

1、如果可以正常访问，记录响应时间，以备后用。

2、如果不能正常访问，则进行重试，重试三次依旧不能访问，则从健康服务列表中剔除，以后的访问不会使用该连接。

注意：重试的等待时间我们选取一个合适范围内的随机时间，这样可以避免局域网络问题导致的大面积同时重试，产生**重试风暴**。

代码如下：

```
@Override
public void run() {

    // 将响应时长的map清空
    YrpcBootstrap.ANSWER_TIME_CHANNEL_CACHE.clear();

    // 遍历所有的channel
    Map<InetSocketAddress, Channel> cache = YrpcBootstrap.CHANNEL_CACHE;
    for (Map.Entry<InetSocketAddress, Channel> entry : cache.entrySet()) {
        // 定义一个重试的次数
        int tryTimes = 3;
        while (tryTimes > 0) {
            // 通过心跳检测处理每一个channel
            Channel channel = entry.getValue();

            long start = System.currentTimeMillis();
            // 构建一个心跳请求
            YrpcRequest yrpcRequest =
                ...  // 省略
                .build();

            // 4、写出报文
            CompletableFuture<Object> completableFuture = new CompletableFuture<>();
            // 将 completableFuture 暴露出去
            YrpcBootstrap.PENDING_REQUEST.put(yrpcRequest.getRequestId(),
completableFuture);
```

```
            channel.writeAndFlush(yrpcRequest).addListener((ChannelFutureListener)
promise -> {
                if (!promise.isSuccess()) {
                    completableFuture.completeExceptionally(promise.cause());
                }
            });

            Long endTime = 0L;
            try {
                // 阻塞方法，get()方法如果得不到结果，就会一直阻塞
                // 我们想不一直阻塞可以添加参数
                completableFuture.get(1, TimeUnit.SECONDS);
                endTime = System.currentTimeMillis();
            } catch (InterruptedException | ExecutionException | TimeoutException e) {
                // 一旦发生问题，需要优先重试
                tryTimes --;
                log.error("和地址为【{}】的主机连接发生异常.正在进行第【{}】次重试......",
                          channel.remoteAddress(), 3 - tryTimes);

                // 将重试的机会用尽，将失效的地址移出服务列表
                if(tryTimes == 0){
                    YrpcBootstrap.CHANNEL_CACHE.remove(entry.getKey());
                }

                // 尝试等到一段时间后重试
                try {
                    Thread.sleep(10*(new Random().nextInt(5)));
                } catch (InterruptedException ex) {
                    throw new RuntimeException(ex);
                }

                continue;
            }
            Long time = endTime - start;

            // 使用treemap进行缓存
            YrpcBootstrap.ANSWER_TIME_CHANNEL_CACHE.put(time, channel);
            log.debug("和[{}]服务器的响应时间是[{}].", entry.getKey(), time);
            break;
        }
    }

    log.info("----------------------响应时间的treemap----------------------");
    for (Map.Entry<Long, Channel> entry :
YrpcBootstrap.ANSWER_TIME_CHANNEL_CACHE.entrySet()) {
        if (log.isDebugEnabled()) {
            log.debug("[{}]--->channelId:[{}]", entry.getKey(), entry.getValue().id());
        }
    }
}
}
```

三、最短响应时间

事实上，不同的人可能实现的方式也不同，我们手写了心跳检测，心跳检测正好可以帮助我们收集一些元数据（比如响应时间），我们利用treeMap将响应时间和对应的channel进行排序，取出响应时间最短的即可。

四、模板方法进行改造

为了支持多种负载均衡策略，我们同样抽象出了**负载均衡器**的抽象概念，形成一个接口，如下：

```
public interface LoadBalancer {

    /**
     * 根据服务名获取一个可用的服务
     * @param serviceName 服务名称
     * @return 服务地址
     */
    InetSocketAddress selectServiceAddress(String serviceName,String group);

    /**
     * 当感知节点发生了动态上下线，我们需要重新进行负载均衡
     * @param serviceName 服务的名称
     */
    void reLoadBalance(String serviceName, List<InetSocketAddress> addresses);
}
```

同时，我们也发现不同的负载均衡器只是实现的算法不同，在执行一些操作时骨架代理逻辑是一样的，所以我们想起了模板方法设计模式，将相同的骨干逻辑封装在抽象类中：

```
public abstract class AbstractLoadBalancer implements LoadBalancer {

    // 一个服务会匹配一个selector
    private Map<String, Selector> cache = new ConcurrentHashMap<>(8);

    // 骨架逻辑
    @Override
    public InetSocketAddress selectServiceAddress(String serviceName,String group) {

        // 1、优先从cache中获取一个选择器
        Selector selector = cache.get(serviceName);

        // 2、如果没有，就需要为这个service创建一个selector
        if (selector == null) {
            // 对于这个负载均衡器，内部应该维护服务列表作为缓存
            List<InetSocketAddress> serviceList = YrpcBootstrap.getInstance()

.getConfiguration().getRegistryConfig().getRegistry().lookup(serviceName,group);

            // 提供一些算法负责选取合适的节点
            selector = getSelector(serviceList);
```

```
            // 将select放入缓存当中
            cache.put(serviceName, selector);
        }

        // 获取可用节点
        return selector.getNext();
    }

    @Override
    public synchronized void reLoadBalance(String serviceName,List<InetSocketAddress>
addresses) {
        // 我们可以根据新的服务列表生成新的selector
        cache.put(serviceName,getSelector(addresses));
    }

    /**
     * 由子类进行扩展
     * @param serviceList 服务列表
     * @return 负载均衡算法选择器
     */
    protected abstract Selector getSelector(List<InetSocketAddress> serviceList);

}
```

而算法独立进行抽象：

```
public interface Selector {

    /**
     * 根据服务列表执行一种算法获取一个服务节点
     * @return 具体的服务节点
     */
    InetSocketAddress getNext();

}
```

第九天

一、实现动态上下线

当有**服务提供方动态上下线**，我们如何进行感知呢？服务上线，首先会在注册中心进行注册，调用方是无法实时感知的，合理的方式只有两种：

1、调用方**定时的去主动的拉。**

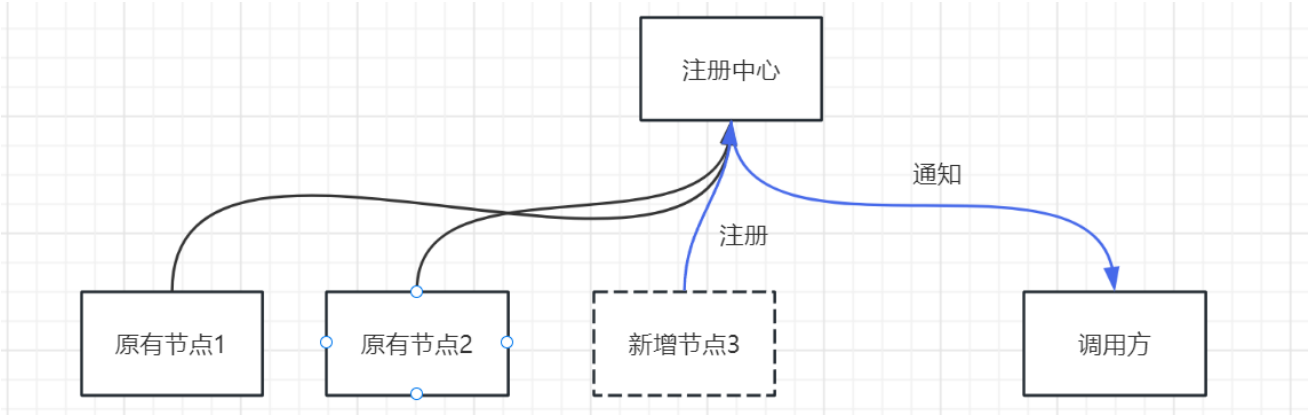
2、注册中心**主动的推送。**

在我们当前的项目中zk提供了watcher机制，我们正好可以利用他来实现动态上下线，具体步骤如下：

1、调用方拉取服务列表时，注册一个watcher关注该服务节点的变化。

2、当服务提供方上线或线下时会触发watcher机制（节点发生了变化）。

3、通知调用方，执行动态上下线的操作。



代码如下：

```
@Override
public List<InetSocketAddress> lookup(String serviceName,String group) {
    // 1、找到服务对应的节点
    String serviceNode = Constant.BASE_PROVIDERS_PATH + "/" + serviceName + "/" +group;

    // 2、从zk中获取他的子节点，这里需要注册一个watcher
    List<String> children = ZookeeperUtils.getChildren(zooKeeper, serviceNode,new
UpAndDownWatcher());
    // 获取了所有的可用的服务列表
    List<InetSocketAddress> inetSocketAddresses = children.stream().map(ipString -> {
        String[] ipAndPort = ipString.split(":");
        String ip = ipAndPort[0];
        int port = Integer.parseInt(ipAndPort[1]);
        return new InetSocketAddress(ip, port);
    }).toList();

    if(inetSocketAddresses.size() == 0){
        throw new DiscoveryException("未发现任何可用的服务主机.");
    }

    return inetSocketAddresses;
}
```

一旦节点发生了变化，UpAndDownWatcher就会被触发，会触发reloadbalance（重新进行负载均衡），代码如下：

```
public class UpAndDownWatcher implements Watcher {
    @Override
    public void process(WatchedEvent event) {

        // 当前的阶段是否发生了变化
        if (event.getType() == Event.EventType.NodeChildrenChanged){
            if (log.isDebugEnabled()){
```

```
                log.debug("检测到服务【{}】下有节点上/下线，将重新拉取服务列
表...",event.getPath());
            }
            String serviceName = getServiceName(event.getPath());
            Registry registry =
YrpcBootstrap.getInstance().getConfiguration().getRegistryConfig().getRegistry();;
            List<InetSocketAddress> addresses = registry.lookup(serviceName,

 YrpcBootstrap.getInstance().getConfiguration().getGroup());
            // 处理新增的节点
            for (InetSocketAddress address : addresses) {
                // 新增的节点   会在address 不在CHANNEL_CACHE
                // 下线的节点   可能会在CHANNEL_CACHE 不在address
                if(!YrpcBootstrap.CHANNEL_CACHE.containsKey(address)){
                    // 根据地址建立连接，并且缓存
                    Channel channel = null;
                    try {
                        channel = NettyBootstrapInitializer.getBootstrap()
                            .connect(address).sync().channel();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                    YrpcBootstrap.CHANNEL_CACHE.put(address,channel);
                }
            }

            // 处理下线的节点 可能会在CHANNEL_CACHE 不在address
            for (Map.Entry<InetSocketAddress,Channel> entry:
YrpcBootstrap.CHANNEL_CACHE.entrySet()){
                if(!addresses.contains(entry.getKey())){
                    YrpcBootstrap.CHANNEL_CACHE.remove(entry.getKey());
                }
            }

            // 获得负载均衡器，进行重新的loadBalance
            LoadBalancer loadBalancer =
YrpcBootstrap.getInstance().getConfiguration().getLoadBalancer();
            loadBalancer.reLoadBalance(serviceName,addresses);

        }
    }

    private String getServiceName(String path) {
        String[] split = path.split("/");
        return split[split.length - 1];
    }
}
```

我们**对于reloadbalance的实现十分简单**，就是根据新的服务列表，生成一个新的选择器，将原有的替换即可。

```
@Override
public synchronized void reLoadBalance(String serviceName,List<InetSocketAddress>
addresses) {
    // 我们可以根据新的服务列表生成新的selector
    cache.put(serviceName,getSelector(addresses));
}
```

注：服务上线，下线均可以依赖watcher机制，但是对于下线而言也可以通过心跳探活来实现，我们将两者皆保留。

二、实现包扫描发布服务

扫描包，进行批量发布的思路和逻辑也很简单，具体步骤如下：

1、将包名转化为文件夹

2、遍历文件夹下的所有class文件。

4、获取class文件的完整路径，转化为全限定名。

4、通过**反射进行加载**，封装成ServiceConfig对象，调用发布接口进行发布即可。

代码如下：

```
public YrpcBootstrap scan(String packageName) {
    // 1、需要通过packageName获取其下的所有的类的权限定名称
    List<String> classNames = getAllClassNames(packageName);
    // 2、通过反射获取他的接口，构建具体实现
    List<Class<?>> classes = classNames.stream()
        .map(className -> {
            try {
                return Class.forName(className);
            } catch (ClassNotFoundException e) {
                throw new RuntimeException(e);
            }
        }).filter(clazz -> clazz.getAnnotation(YrpcApi.class) != null)
        .collect(Collectors.toList());

    for (Class<?> clazz : classes) {
        // 获取他的接口
        Class<?>[] interfaces = clazz.getInterfaces();
        Object instance = null;
        try {
            instance = clazz.getConstructor().newInstance();
        } catch (InstantiationException | IllegalAccessException |
InvocationTargetException |
                 NoSuchMethodException e) {
            throw new RuntimeException(e);
        }
```

```
            // 获取分组信息
            YrpcApi yrpcApi = clazz.getAnnotation(YrpcApi.class);
            String group = yrpcApi.group();

            for (Class<?> anInterface : interfaces) {
                ServiceConfig<?> serviceConfig = new ServiceConfig<>();
                serviceConfig.setInterface(anInterface);
                serviceConfig.setRef(instance);
                serviceConfig.setGroup(group);
                if (log.isDebugEnabled()){
                    log.debug("---->已经通过包扫描，将服务【{}】发布.",anInterface);
                }
                // 3、发布
                publish(serviceConfig);
            }

        }
        return this;
    }
```

三、添加配置类

到目前为止我们的所有的配置相关的内容全部定义在了启动引导程序中，这样其实有一些不合理，事实上全局配置我们应该统一放在一个类中，如下，让这个类将会成为我们**当前工程的上下文环境**：

```
@Data
@Slf4j
public class Configuration {

    // 配置信息-->端口号
    private int port = 8094;

    // 配置信息-->应用程序的名字
    private String appName = "default";

    // 分组信息
    private String group = "default";

    // 配置信息-->注册中心
    private RegistryConfig registryConfig = new
RegistryConfig("zookeeper://127.0.0.1:2181");

    // 配置信息-->序列化协议
    private String serializeType = "jdk";

    // 配置信息-->压缩使用的协议
    private String compressType = "gzip";

    // 配置信息-->id发射器
    public IdGenerator idGenerator = new IdGenerator(1, 2);
```

```
    // 配置信息-->负载均衡策略
    private LoadBalancer loadBalancer = new RoundRobinLoadBalancer();

    // 为每一个ip配置一个限流器
    private final Map<SocketAddress, RateLimiter> everyIpRateLimiter = new
ConcurrentHashMap<>(16);
    // 为每一个ip配置一个断路器，熔断
    private final Map<SocketAddress, CircuitBreaker> everyIpCircuitBreaker = new
ConcurrentHashMap<>(16);

    // 读xml，dom4j
    public Configuration() {
        // 1、成员变量的默认配置项

        // 2、spi机制发现相关配置项
        SpiResolver spiResolver = new SpiResolver();
        spiResolver.loadFromSpi(this);

        // 3、读取xml获得上边的信息
        XmlResolver xmlResolver = new XmlResolver();
        xmlResolver.loadFromXml(this);

        // 4、编程配置项，yrpcBootstrap提供
    }
}
```

我们看到了，在配置类的构造器中，整个配置项会从三个地方加载：

1、默认项

2、spi自动发现

3、xml解析

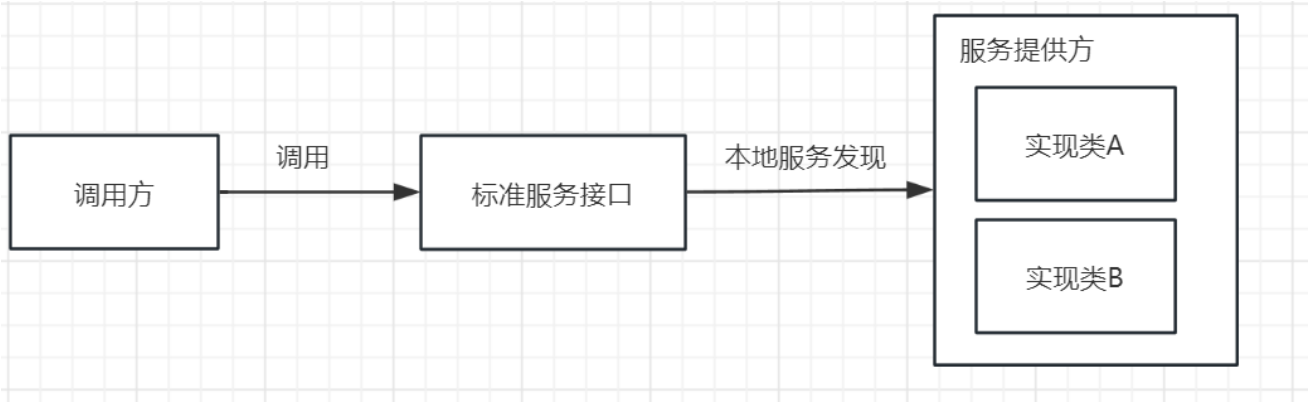
4、编程式配置

第十天

一、spi机制

SPI（Service Provider Interface），是JDK内置的一种**服务提供发现机制**，可以用来启用框架扩展和替换组件，主要是被框架的开发人员使用，比如java.sql.Driver接口，其他不同厂商可以针对同一接口做出不同的实现，MySQL和PostgreSQL都有不同的实现提供给用户，而Java的SPI机制可以为某个接口寻找服务实现。Java中SPI机制主要思想是将装配的控制权移到程序之外，在模块化设计中这个机制尤其重要，其核心思想就是**解耦。**

SPI整体机制图如下：



当服务的提供者提供了一种接口的实现之后，需要在classpath下的META-INF/services/目录里创建一个以服务接口命名的文件，这个文件里的内容就是这个接口的具体的实现类。当其他的程序需要这个服务的时候，就可以通过查找这个jar包（一般都是以jar包做依赖）的META-INF/services/中的配置文件，配置文件中有接口的具体实现类名，可以根据这个类名进行加载实例化，就可以使用该服务了。JDK中查找服务的实现的工具类是：java.util.ServiceLoader。

SPI扩展机制应用场景有很多，比如Common-Logging，JDBC，Dubbo等等。

SPI流程：

1. 有关组织和公式定义接口标准
2. 第三方提供具体实现: 实现具体方法, 配置 META-INF/services/${interface_name} 文件
3. 开发者使用

比如JDBC场景下：

- 首先在Java中定义了接口java.sql.Driver，并没有具体的实现，具体的实现都是由不同厂商提供。
- 在MySQL的jar包mysql-connector-java-6.0.6.jar中，可以找到META-INF/services目录，该目录下会有一个名字为java.sql.Driver的文件，文件内容是com.mysql.cj.jdbc.Driver，这里面的内容就是针对Java中定义的接口的实现。
- 同样在PostgreSQL的jar包PostgreSQL-42.0.0.jar中，也可以找到同样的配置文件，文件内容是org.postgresql.Driver，这是PostgreSQL对Java的java.sql.Driver的实现。

二、手写简易版本的spi

jdk的spi固然也挺好用但是有的时候不能满足我们的特定需求，我们也可手动实现一个，代码如下：

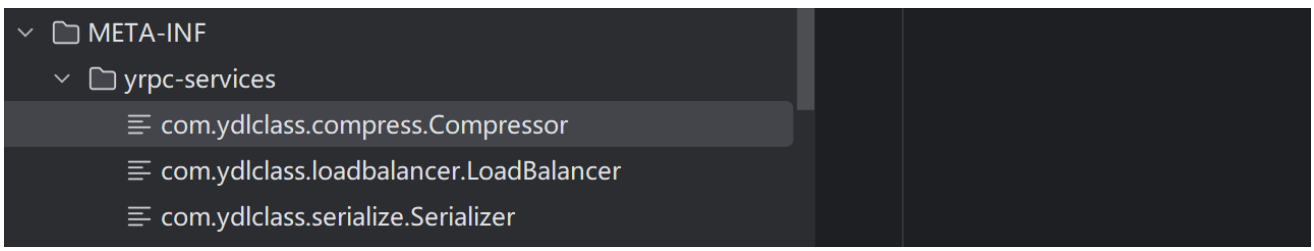
```
public class SpiHandler {

    // 定义一个basePath
    private static final String BASE_PATH = "META-INF/yrpc-services";

    // 先定义一个缓存，保存spi相关的原始内容
    private static final Map<String, List<String>> SPI_CONTENT = new
ConcurrentHashMap<>(8);

    // 缓存的是每一个接口所对应的实现的实例
```

```
    private static final Map<Class<?>,List<ObjectWrapper<?>>> SPI_IMPLEMENT = new
ConcurrentHashMap<>(32);

    // 加载当前类之后需要将spi信息进行保存，避免运行时频繁执行IO
    static {
        // todo 怎么加载当前工程和jar包中的classPath中的资源
        ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
        URL fileUrl = classLoader.getResource(BASE_PATH);
        if(fileUrl != null) {
            File file = new File(fileUrl.getPath());
            File[] children = file.listFiles();
            if(children != null) {
                for (File child : children) {
                    String key = child.getName();
                    List<String> value = getImplNames(child);
                    SPI_CONTENT.put(key, value);
                }
            }
        }
    }


    /**
     * 获取第一个和当前服务相关的实例
     * @param clazz 一个服务接口的class实例
     * @return      实现类的实例
     * @param <T>
     */
    public synchronized static <T> ObjectWrapper<T> get(Class<T> clazz) {

        // 1、优先走缓存
        List<ObjectWrapper<?>> objectWrappers = SPI_IMPLEMENT.get(clazz);
        if(objectWrappers != null && objectWrappers.size() > 0){
            return (ObjectWrapper<T>)objectWrappers.get(0);
        }

        // 2、构建缓存
        buildCache(clazz);

        List<ObjectWrapper<?>>  result = SPI_IMPLEMENT.get(clazz);
        if (result == null || result.size() == 0){
            return null;
        }

        // 3、再次尝试获取第一个
        return (ObjectWrapper<T>)result.get(0);
    }


    /**
     * 获取所有和当前服务相关的实例
     * @param clazz 一个服务接口的class实例
     * @return       实现类的实例集合
```

```
     * @param <T>
     */
    public synchronized static <T> List<ObjectWrapper<T>> getList(Class<T> clazz) {

        // 1、优先走缓存
        List<ObjectWrapper<?>> objectWrappers = SPI_IMPLEMENT.get(clazz);
        if(objectWrappers != null && objectWrappers.size() > 0){
            return objectWrappers.stream().map( wrapper -> (ObjectWrapper<T>)wrapper )
                .collect(Collectors.toList());
        }

        // 2、构建缓存
        buildCache(clazz);

        // 3、再次获取
        objectWrappers = SPI_IMPLEMENT.get(clazz);
        if(objectWrappers != null && objectWrappers.size() > 0){
            return objectWrappers.stream().map( wrapper -> (ObjectWrapper<T>)wrapper )
                .collect(Collectors.toList());
        }
        return new ArrayList<>();
    }


    /**
     * 获取文件所有的实现名称
     * @param child 文件对象
     * @return      实现类的权限定名称结合
     */
    private static List<String> getImplNames(File child) {
        try(
            FileReader fileReader = new FileReader(child);
            BufferedReader bufferedReader = new BufferedReader(fileReader)
        )
        {
            List<String> implNames = new ArrayList<>();
            while (true){
                String line = bufferedReader.readLine();
                if (line == null || "".equals(line)) break;
                implNames.add(line);
            }
            return implNames;
        } catch (IOException e){
            log.error("读取spi文件时发生异常.",e);
        }
        return null;
    }

    /**
     * 构建clazz相关的缓存
     * @param clazz 一个类的class实例
     */
    private static void buildCache(Class<?> clazz) {
```

```
        // 1、通过clazz获取与之匹配的实现名称
        String name = clazz.getName();
        List<String> implNames = SPI_CONTENT.get(name);
        if(implNames == null || implNames.size() == 0){
            return;
        }

        // 2、实例化所有的实现
        List<ObjectWrapper<?>> impls = new ArrayList<>();
        for (String implName : implNames) {
            try {
                // 首先进行分割
                String[] codeAndTypeAndName = implName.split("-");
                if(codeAndTypeAndName.length != 3){
                    throw new SpiException("您配置的spi文件不合法");
                }
                Byte code = Byte.valueOf(codeAndTypeAndName[0]);
                String type = codeAndTypeAndName[1];
                String implementName = codeAndTypeAndName[2];

                Class<?> aClass = Class.forName(implementName);
                Object impl = aClass.getConstructor().newInstance();

                ObjectWrapper<?> objectWrapper = new ObjectWrapper<>(code,type,impl);

                impls.add(objectWrapper);
            } catch (ClassNotFoundException | NoSuchMethodException |
InstantiationException | IllegalAccessException |
                     InvocationTargetException e){
                log.error("实例化【{}】的实现时发生了异常",implName,e);
            }

        }
        SPI_IMPLEMENT.put(clazz,impls);
    }

}
```

这样我们只需要将目录结构按照如下的方式建立，并按照规范填写响应的内容即可完成spi的自动发现。



第十一天

rpc 是解决分布式系统通信问题的一大利器，而分布式系统的一大特点就是**高并发**，所以说 yrpc **也可能会面临高并发的场景**，即使不会我们也要假装会。

在这样的情况下，我们提供服务的每个服务节点就都可能由于访问量过大而引起一系列的问题，比如业务处理耗时过长、CPU 飘高、频繁 Full GC 以及服务进程直接宕机等等。

但是在生产环境中，我们要保证服务的稳定性和高可用性，这时我们就需要**业务进行自我保护**，从而保证在高访问量、高并发的场景下，应用系统**依然稳定，服务依然高可用。**

那么在使用 yrpc 时，业务又如何实现自我保护呢?

最常见的方式就是限流了，简单有效，但 yrpc 框架的自我保护方式可不只有限流，并且 yrpc 框架的限流方式可以是多种多样的。

我们可以将 yrpc 框架拆开来分析，yrpc 调用包括**服务端和调用端**，调用端向服务端发起调用。下面我就分享一下服务端与调用端分别是如何进行自我保护的。

一、异常重试

1、为什么需要异常重试?

我们可以考虑这样一个场景。我们发起一次 yrpc 调用，去调用远程的一个服务，比如用户的登录操作，我们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。当我们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，导致我们的请求失败了，而这个请求我们希望它能够尽可能地执行成功，那这时我们要怎么做呢?

我们需要重新发起一次 yrpc 调用，那我们在代码中该如何处理呢？是在代码逻辑里 catch 一下，失败了就再发起一次调用吗？这样做显然不够优雅吧。这时我们就可以**考虑使用 yrpc 框架的重试机制。**

2、yrpc 框架的重试机制

那什么是 yrpc 框架的重试机制呢?

这其实很好理解，就是**当调用端发起的请求失败时，yrpc 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。**

q：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。

综上，我们可以总结出：在使用 yrpc 框架的时候，**我们要确保被调用的服务的业务逻辑是幂等的**，这样我们才能考虑根据事件情况开启 yrpc 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。**

通过上述讲解，我相信你已经非常清楚 yrpc 框架的重试机制了，这也是现在大多数 yrpc 框架所采用的重试制。当然为了解决以上的问题我们也提供了以下方案：

1、手动指定可重试的接口，可以通过注解的形式进行标记，有特定注解的接口才能重试。

2、设置重试白名单。

q：如果因为机房的网络问题导致了大量的请求被重试，而且是同时进行，会不会产生问题？会的

为了避免因网络抖动导致的重试风暴，可以采用以下策略：

1. 指数退避算法：在连续的重试中，每次重试之间的等待时间呈指数级增长。这样可以降低在短时间内发起大量重试请求的可能性，从而减轻对系统的压力。
2. 随机抖动：在指数退避算法的基础上，引入随机抖动，使得重试之间的等待时间变得不那么规律。这样可以避免多个客户端在相同时间点发起重试请求，进一步减轻服务器压力。
3. 限制重试次数和超时时间：限制单个请求的最大重试次数，以及整个重试过程的总超时时间，防止无限制地发起重试请求。
4. 请求结果缓存：如果有些请求的结果可以缓存，可以考虑在客户端或服务器端缓存请求结果。当发生重试时，直接从缓存中获取结果，以减轻服务器压力。
5. 服务熔断：在客户端或服务器端实现熔断机制，当连续失败达到一定阈值时，触发熔断，暂时阻止后续请求。熔断器在一段时间后会自动恢复，允许新的请求通过。

下面是一个使用指数退避和随机抖动的重试策略示例，实时上我们的工程并没有使用这个：

```
public class RetryPolicy {
    private int maxRetries; // 最大重试次数
    private int initialInterval; // 初始重试间隔
    private double backoffMultiplier; // 退避系数
    private double jitterFactor; // 抖动因子

    // 构造方法
    public RetryPolicy(int maxRetries, int initialInterval, double backoffMultiplier,
double jitterFactor) {
        this.maxRetries = maxRetries;
        this.initialInterval = initialInterval;
        this.backoffMultiplier = backoffMultiplier;
        this.jitterFactor = jitterFactor;
    }

    // 获取最大重试次数
    public int getMaxRetries() {
        return maxRetries;
    }

    // 获取下一次重试的间隔时间
    public long getNextRetryInterval(int retryCount) {
        double backoff = initialInterval * Math.pow(backoffMultiplier, retryCount); //
计算指数退避的时间间隔
        double jitter = backoff * jitterFactor * (Math.random() * 2 - 1); // 计算抖动时间
间隔
        return (long) (backoff + jitter); // 返回指数退避和抖动的总和
    }
}
```

我们不妨思考一下，能不能使用策略设计模式将重试机制进行抽象，提供不同的重试机制。

本项目中，我们偷了个懒，实现了最简单的重试。

定义重试接口：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TryTimes {

    int tryTimes() default 3;
    int intervalTime() default 2000;

}
```

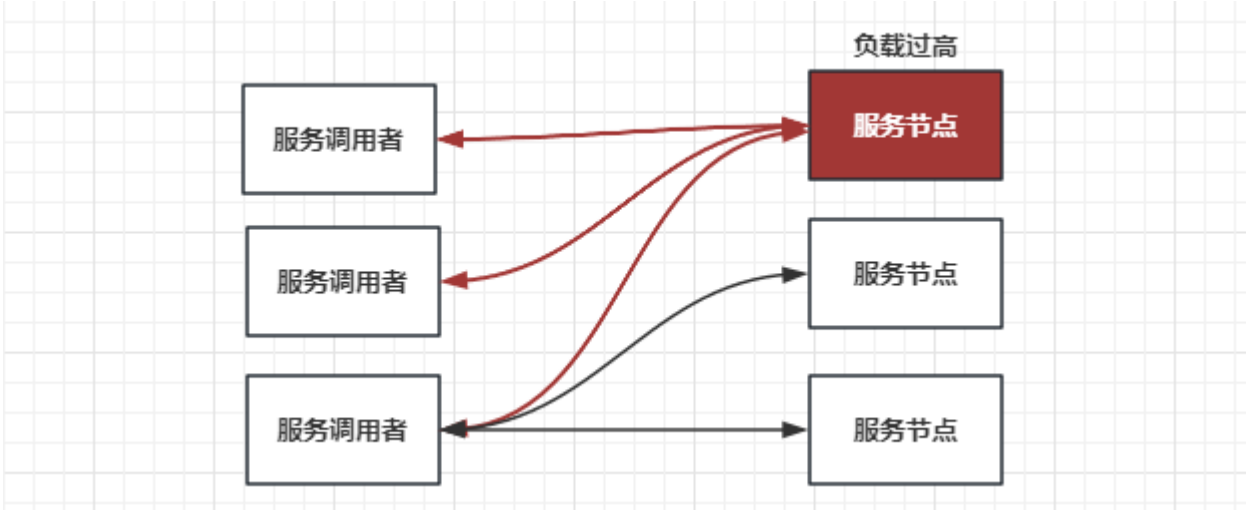
重试的核心逻辑：

```
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

    // 从接口中获取判断是否需要重试
    TryTimes tryTimesAnnotation = method.getAnnotation(TryTimes.class);

    // 默认值0,代表不重试
    int tryTimes = 0;
    int intervalTime = 0;
    if (tryTimesAnnotation != null) {
        tryTimes = tryTimesAnnotation.tryTimes();
        intervalTime = tryTimesAnnotation.intervalTime();
    }


    while (true) {
        try {
            // 执行逻辑
            break;
        } catch (Exception e) {
            // 次数减一，并且等待固定时间，固定时间有一定的问题，重试风暴
            tryTimes--;
            try {
                Thread.sleep(intervalTime);
            } catch (InterruptedException ex) {
                log.error("在进行重试时发生异常.", ex);
            }
            if (tryTimes < 0) {
                log.error("对方法【{}】进行远程调用时，重试{}次，依然不可调用",
                          method.getName(), tryTimes, e);
                break;
            }
            log.error("在进行第{}次重试时发生异常.", 3 - tryTimes, e);
        }
    }
    throw new RuntimeException("执行远程方法" + method.getName() + "调用失败。");
}
```

二、熔断限流

1、服务端的自我保护

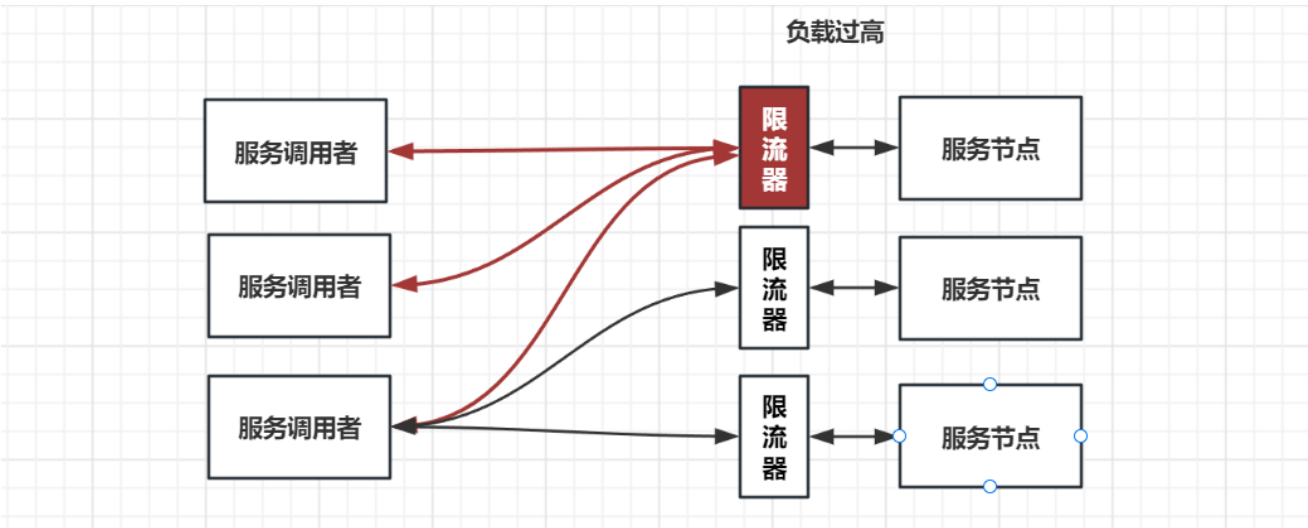
我们先看服务端，举个例子，假如我们要发布一个 yrpc 服务，作为服务端接收调用端发送过来的请求，这时服务端的某个节点负载压力过高了，我们该如何保护这个节点？



这个问题还是很好解决的，既然**负载压力高**，那就**不让它再接收太多的请求就好了**，等接收和处理的请求数量下来后，这个节点的负载压力自然就下来了。

那么就是限流吧？是的，在 yrpc 调用中**服务端的自我保护策略**就是限流，那你有没有想过我们是如何实现限流的呢？是在服务端的业务逻辑中做限流吗？有没有更优雅的方式？

限流是一个比较通用的功能，我们可以在 yrpc 框架中集成限流的功能，让使用方自己去配置限流阈值；我们还可以在服务端添加限流逻辑，当调用端发送请求过来时，服务端在执行业务逻辑之前先执行限流逻辑，如果发现访问量过大并且超出了限流的阈值，就让服务端直接抛回给调用端一个限流异常，否则就执行正常的业务逻辑。



那服务端的限流逻辑又该如何实现呢？

方式有很多，比如最简单的计数器，还有可以做到平滑限流的**滑动窗口、漏斗算法以及令牌桶算法**等等。其中**令牌桶算法最为常用**。上述这几种限流算法我就不一一讲解了，资料很多，不太清楚的话自行查阅下就可以了。

我们在项目中也手写了一个限流器，基于令牌桶算法：

```
public class TokenBuketRateLimiter implements RateLimiter {
    // 思考，令牌是个啥？令牌桶是个啥？
    // String,Object?  list? ,map?

    // 代表令牌的数量，>0 说明有令牌，能放行，放行就减一，==0,无令牌  阻拦
    private int tokens;

    // 限流的本质就是，令牌数
    private final int capacity;

    // 令牌桶的令牌，如果没了要怎么办？ 按照一定的速率给令牌桶加令牌,如每秒加500个，不能超过总数
    // 可以用定时任务去加--> 启动一个定时任务，每秒执行一次 tokens+500 不能超过 capacity （不好）
    // 对于单机版的限流器可以有更简单的操作，每一个有请求要发送的时候给他加一下就好了
    private final int rate;

    // 上一次放令牌的时间
    private Long lastTokenTime;

    public TokenBuketRateLimiter(int capacity, int rate) {
        this.capacity = capacity;
        this.rate = rate;
        lastTokenTime = System.currentTimeMillis();
        tokens = capacity;
    }

    /**
     * 判断请求是否可以放行
     * @return true 放行  false  拦截
     */
    public synchronized boolean allowRequest() {
        // 1、给令牌桶添加令牌
        // 计算从现在到上一次的时间间隔需要添加的令牌数
        Long currentTime = System.currentTimeMillis();
        long timeInterval = currentTime - lastTokenTime;
        // 如果间隔时间超过一秒，放令牌
        if(timeInterval >= 1000/rate){
            int needAddTokens = (int)(timeInterval * rate / 1000);
            System.out.println("needAddTokens = " + needAddTokens);
            // 给令牌桶添加令牌
            tokens = Math.min(capacity, tokens + needAddTokens);
            System.out.println("tokens = " + tokens);

            // 标记最后一个放入令牌的时间
            this.lastTokenTime = System.currentTimeMillis();
        }

        // 2、自己获取令牌,如果令牌桶中有令牌则放行，否则拦截
        if(tokens > 0){
            tokens --;
            System.out.println("请求被放行---------------");
            return true;
```

```
        } else {
            System.out.println("请求被拦截---------------");
            return false;
        }

    }
}
```

我们可以假设这样一个场景：我发布了一个服务，提供给**多个应用的调用方去调用**，这时有**一个应用的调用方发送过来的请求流量要比其它的应用大很多**，这时我们就应该对**这个应用下的调用端发送过来的请求流量进行限流。**所以说我们在做限流的时候要考虑**应用级别的维度，甚至是 IP 级别的维度**，这样做不仅可以让我们对一个应用下的调用端发送过来的请求流量做限流，还可以对一个 IP 发送过来的请求流量做限流。

这时你可能会想，使用方该**如何配置应用维度以及 IP 维度的限流呢？**在代码中配置是不是不大方便？我之前说过，yrpc 框架真正强大的地方在于**它的治理功能**，而治理功能大多都需要依赖一个**注册中心或者配置中心**，我们可以通过 **yrpc 治理的管理端进行配置**，再通过注册中心或者配置中心将限流阈值的配置下发到服务提供方的每个节点上，**实现动态配置。**

看到这儿，你有没有发现，在**服务端实现限流，配置的限流阈值是作用在每个服务节点上的。**比如说我配置的阈值是每秒 1000 次请求，那么就是指一台机器每秒处理 1000 次请求；如果我的服务集群拥有 10 个服务节点，那么我提供的服务限流阈值在最理想的情况下就是每秒 10000 次。

接着看这样一个场景：我提供了一个服务，而这个服务的业务逻辑依赖的是 MySQL 数据库，由于 MySQL 数据库的性能限制，我们是需要**对其进行保护。**假如在 MySQL 处理业务逻辑中，SQL 语句的能力是每秒 10000 次，那么我们提供的服务处理的访问量就不能超过每秒 10000 次，而我们的服务有 10 个节点，这时我们配置的限流阈值应该是每秒 1000 次。那如果之后因为某种需求我们对这个服务扩容了呢？扩容到 20 个节点，我们是不是就要把限流阈值调整到每秒 500 次呢？这样操作每次都要自己去计算，重新配置，显然太麻烦了。

我们可以让 yrpc 框架自己去计算，当注册中心或配置中心将限流阈值配置下发的时候，我们可以将总服务节点数也下发给服务节点，之后由服务节点自己计算限流阈值，这样就解决问题了吧？

解决了一部分，还有一个问题存在，那就是在实际情况下，一个服务节点所接收到的访问量并不是绝对均匀的，比如有 20 个节点，而每个节点限流的阈值是 500，其中有的节点访问量已经达到阈值了，但有的节点可能在这一秒内的访问量是 450，这时调用端发送过来的总调用量还没有达到 10000 次，但可能也会被限流，这样是不是就不精确了？那有没有比较精确的限流方式呢？

我刚才讲解的限流方式之所以不精确，是因为限流逻辑是服务集群下的每个节点独立去执行的，是一种**单机的限流方式**，而且每个服务节点所接收到的流量并不是绝对均匀的。

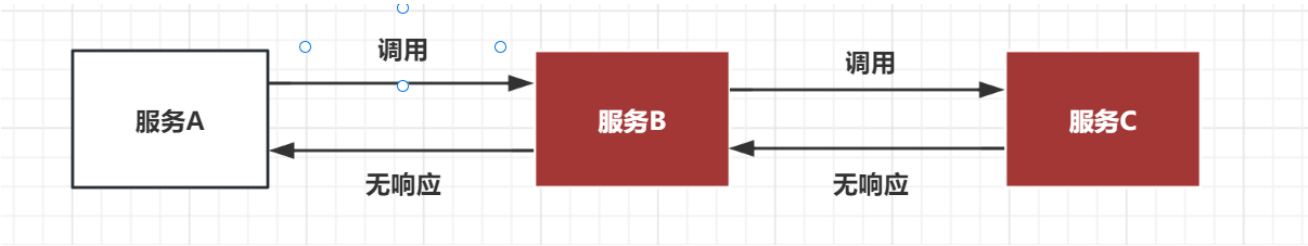
我们可以提供一个**专门的限流服务，**让每个节点都依赖一个限流服务，当请求流量打过来时，服务节点触发限流逻辑，调用这个限流服务来判断是否到达了限流阈值。我们甚至可以将**限流逻辑放在调用端，调用端在发出请求时先触发限流逻辑，调用限流服务，如果请求量已经到达了限流阈值，请求都不需要发出去，直接返回给动态代理**一个限流异常即可。

这种限流方式可以让整个服务集群的限流变得更加精确，但也由于依赖了一个限流服务，它在性能和耗时上与单机的限流方式相比是有很大劣势的。至于要选择哪种限流方式，就要结合具体的应用场景进行选择了。

2、调用端的自我保护

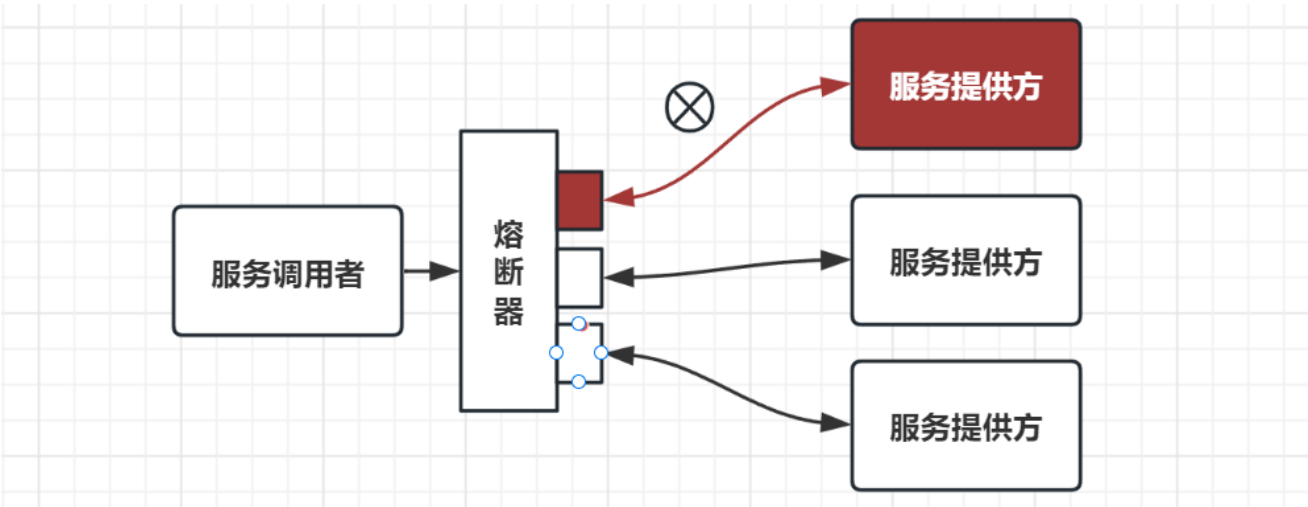
刚才我讲解了服务端如何进行自我保护，最简单有效的方式就是限流。那么调用端呢？调用端是否需要自我保护呢？

举个例子，假如我要发布一个服务 B，而服务 B 又依赖服务 C，当一个服务 A 来调用服务 B 时，服务 B 的业务逻辑调用服务 C，而这时服务 C 响应超时了，由于服务 B 依赖服务 C，C 超时直接导致 B 的业务逻辑一直等待，而这个时候服务 A 在频繁地调用服务 B，服务 B 就可能会因为堆积大量的请求而导致服务宕机。



由此可见，服务 B 调用服务 C，服务 C 执行业务逻辑出现异常时，会影响到服务 B，甚至可能会引起服务 B 宕机。这还只是 A->B->C 的情况，试想一下 A->B->C->D->……呢？在整个调用链中，只要中间有一个服务出现问题，都可能会引起上游的所有服务出现一系列的问题，甚至会引起整个调用链的服务都宕机，这是非常恐怖的。

所以说，在一个服务作为调用端调用另外一个服务时，为了防止被调用的服务出现问题而影响到作为调用端的这个服务，这个服务也需要进行自我保护。**而最有效的自我保护方式就是熔断。**



我们可以先了解下熔断机制。

熔断器的工作机制主要是关闭、打开和半打开这三个状态之间的切换。

1. 在正常情况下，熔断器是关闭的。
2. 当调用端调用下游服务出现异常时，熔断器会**收集异常指标信息进行计算**，当达到熔断条件时熔断器打开，这时调用端再发起请求是会**直接被熔断器拦截**，并快速地执行失败逻辑；
3. 当熔断器打开一段时间后，会转为**半打开状态**，这时熔断器允许调用端**发送一个请求给服务端**，如果这次请求能够正常地得到服务端的响应，则将状态置为关闭状态，否则设置为打开。

了解完熔断机制，你就会发现，在业务逻辑中加入熔断器其实是不够优雅的。**那么在 yrpc 框架中，我们该如何整合熔断器呢？**

熔断机制主要是保护调用端，调用端在发出请求的时候会先经过熔断器。

想到这里我们就自然而然想到在动态代理中加入熔断逻辑就再合理不过了

我的建议是动态代理，因为在 yrpc 调用的流程中，动态代理是 yrpc 调用的第一个关口。在发出请求时先经过熔断器，如果状态是闭合则正常发出请求，如果状态是打开则执行熔断器的失败策略。

我们自己手写的断路器代码如下，我们并没有添加半打开的状态，有兴趣的同学可以自己实现一下看看：

```
public class CircuitBreaker {

    // 理论上：标准的断路器应该有三种状态  open close half_open，我们为了简单只选取两种
    private volatile boolean isOpen = false;

    // 需要搜集指标  异常的数量   比例
    // 总的请求数
    private AtomicInteger requestCount = new AtomicInteger(0);

    // 异常的请求数
    private AtomicInteger errorRequest = new AtomicInteger(0);

    // 异常的阈值
    private int maxErrorRequest;
    private float maxErrorRate;

    public CircuitBreaker(int maxErrorRequest, float maxErrorRate) {
        this.maxErrorRequest = maxErrorRequest;
        this.maxErrorRate = maxErrorRate;
    }


    // 断路器的核心方法，判断是否开启
    public boolean isBreak(){
        // 优先返回，如果已经打开了，就直接返回true
        if(isOpen){
            return true;
        }

        // 需要判断数据指标，是否满足当前的阈值
        if( errorRequest.get() > maxErrorRequest ){
            this.isOpen = true;
            return true;
        }

        if( errorRequest.get() > 0 && requestCount.get() > 0 &&
           errorRequest.get()/(float)requestCount.get() > maxErrorRate
          ) {
            this.isOpen = true;
            return true;
        }

        return false;
    }

    // 每次发生请求，获取发生异常应该进行记录
    public void recordRequest(){
        this.requestCount.getAndIncrement();
```

```
    }

    public void recordErrorRequest(){
        this.errorRequest.getAndIncrement();
    }

    /**
     * 重置熔断器
     */
    public void reset(){
        this.isOpen = false;
        this.requestCount.set(0);
        this.errorRequest.set(0);
    }

}
```

三、流量隔离

上节课，我们介绍了 yrpc 中常用的保护手段“熔断限流”，熔断是调用方为了避免在调用过程中，服务提供方出现问题的时候，自身资源被耗尽的一种保护行为；而限流则是服务提供方为防止自己被突发流量打垮的一种保护行为。虽然这两种手段作用的对象不同，但出发点都是为了实现自我保护，所以一旦发生这种行为，业务都是有损的。

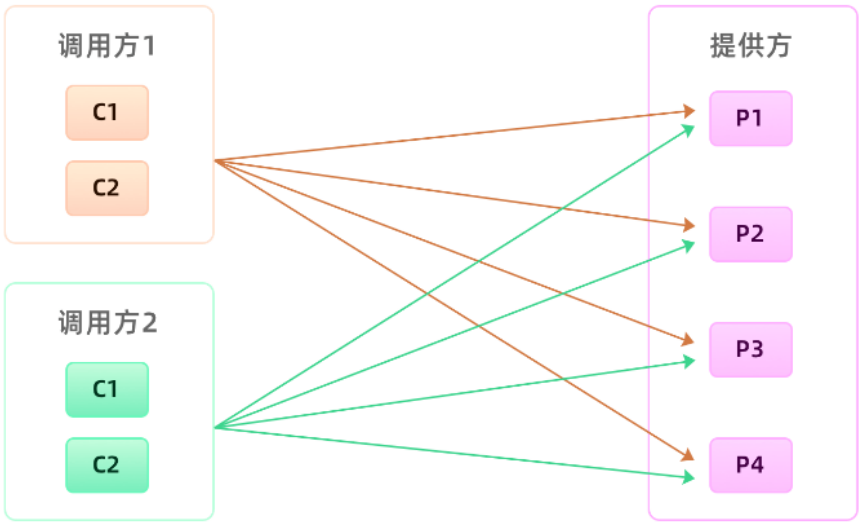
那说起突发流量，限流固然是一种手段，但其实面对复杂的业务以及高并发场景时，我们还有别的手段，可以最大限度地保障业务无损，那就是**流量隔离**。这也是我今天重点要和你分享的内容，接下来我们就一起看看分组在 yrpc 中的应用。

1、为什么需要分组?

我们举一个例子，有一条街道，很宽但是街道上没有画任何的辅助线，人和车可以随便行驶，在人少车少的情况下我们必然可以畅通无阻，但是车辆一旦变多，就很难控制，有向前开的车，有向后开的车，有横穿马路的人……

所以我们的马路是需要划分区域的，有向东的车道、向西的车道，在快速路和高速还要将道路隔离，还有非机动车道、人行道等等，立下这样的规矩后，有很多好处，高速上向东的车道堵了，不会影响向西行驶的车辆，人、非机动车、机动车的行驶互不影响。

同样的道理，我们用在 yrpc 治理上也是一样的。假设你是一个服务提供方应用的负责人，在早期业务量不大的情况下，应用之间的调用关系并不会复杂，请求量也不会很大，我们的应用有足够的能力扛住日常的所有流量。我们并不需要花太多的时间去治理调用请求过来的流量，我们通常会选择最简单的方法，就是把服务实例统一管理，把所有的请求都用一个共享的“大池子”来处理。这就类似于“简单道路时期”，服务调用方跟服务提供方之间的调用拓扑如下图所示：



后期业务发展了，调用你接口的调用方就会越来越多，流量也会渐渐多起来。可能某一天，一个“爆炸式惊喜”就来了。其中一个调用方的流量突然激增，让你整个集群瞬间处于高负载运行，进而影响到其它调用方，导致整体的业务可用性降低。

怎么样杜绝这样的事情发生呢？最好的办法就是隔离流量，将多个yrpc服务进行分组，一个调用方只能访问一个分组的服务，就是一个调用方流量爆炸也只会影响一个分组的服务，整体还是可用的。

2、怎么实现分组?

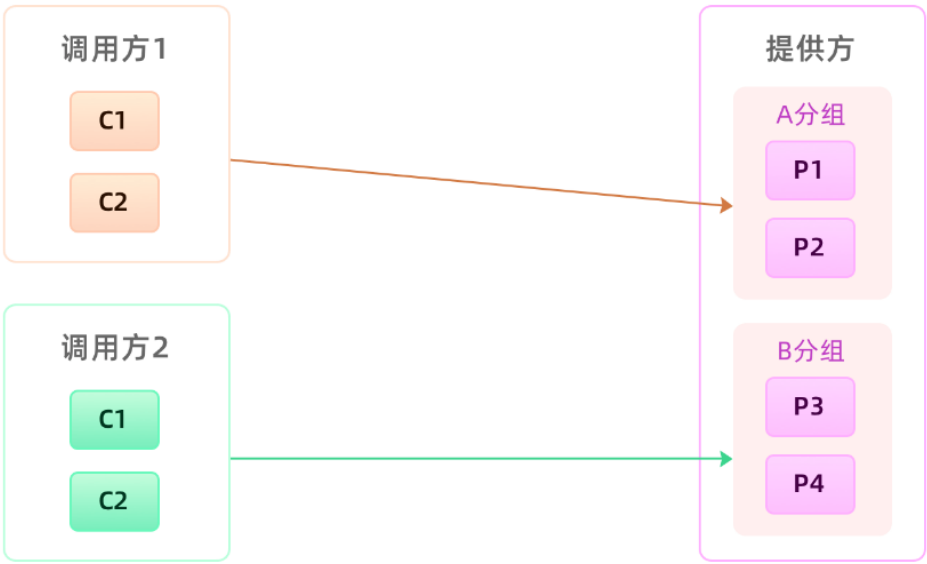
那我们在yrpc项目中怎么实现分组呢？分组的逻辑的就是让调用方可以发现一个分组的服务，那实现的逻辑就一定是服务发现的时候只能拉取同一个分组的服务，我们需要在服务发现中做一些改造。

原本服务调用方是通过**接口名去注册中心找到所有的服务节点来完成服务发现的**，那换到这里的话，这样做其实并不合适，因为这样调用方会拿到所有的服务节点。因此为了实现分组隔离逻辑，我们需要重新改造下服务发现的逻辑，调用方去**获取服务节点的时候除了要带着接口名，还需要另外加一个分组参数**，相应的服务提供方在注册的时候也要带上分组参数。

通过改造后的分组逻辑，我们可以把服务提供方所有的实例分成若干组，每一个分组可以提供给单个或者多个不同的调用方来调用。那怎么分组好呢，有没有统一的标准？

坦白讲，这个分组并没有一个可衡量的标准，但我自己总结了一个规则可以供你参考，就是按照应用重要级别划分。

非核心应用不要跟核心应用分在同一个组，核心应用之间应该做好隔离，一个重要的原则就是保障核心应用不受影响。比如提供给电商下单过程中用的商品信息接口，我们肯定是需要独立出一个单独分组，避免受其它调用方污染的。有了分组之后，我们的服务调用方跟服务提供方之间的调用拓扑就如下图所示：



通过分组的方式隔离调用方的流量，从而避免因为一个调用方出现流量激增而影响其它调用方的可用率。对服务提供方来说，这种方式是我们日常治理服务过程中一个高频使用的手段，那通过这种分组进行流量隔离，对调用方应用会不会有影响呢?

回到我们的项目中，我们的实现方案就是，给服务加上一个`@YrpcApi(group = "primary")`通过group属性对服务进行编组：

以后在服务注册和发现时，将组名作为一个参数携带上即可。

第十二天

一、优雅停机

首先，我们讨论一下暴力关闭的问题和解决方案：

我们知道，我们yrpc是服务于**分布式系统**的，到目前我们我们已经实现了基本的服务治理的能力，但是昨天还有个学生对我来了一个灵魂拷问。

q: 我启动了多个服务提供方，同时启动了服务调用方，不断的进行方法调用，然后**快速关闭几个提供方**发现就出问题了，目前我们的项目确实会有这样的问题。

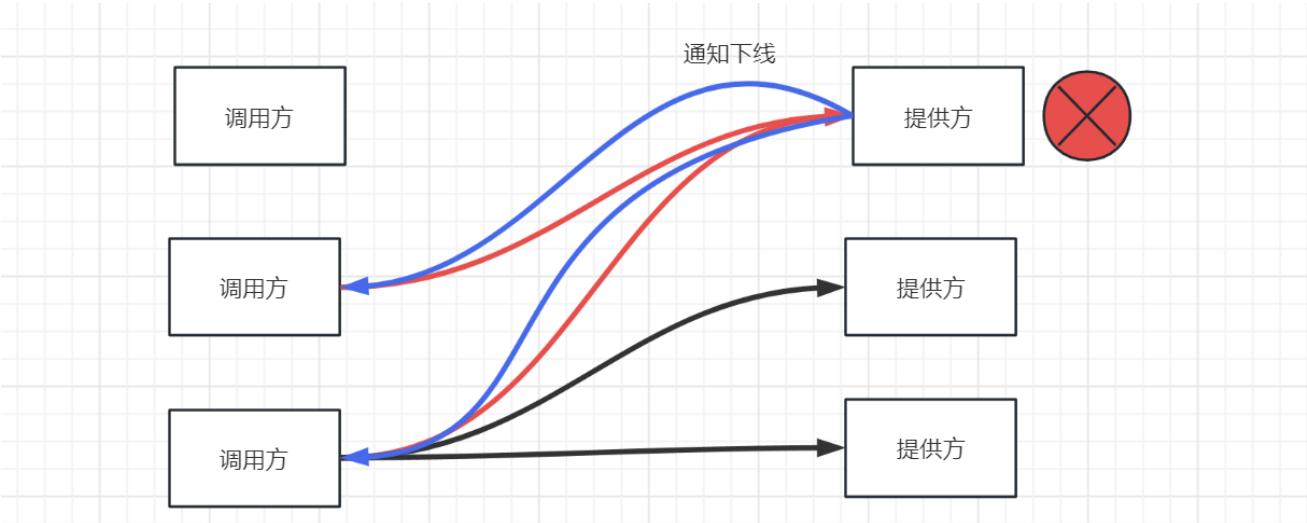
当我们**快速关闭服务提供方**时，注册中心感知、以及通过watcher机制通知调用方**一定不能做到实时**，**一定会有延时**，同时我们的心跳检测也会有**一定的时间间隔。**也就意味着当一个提供方实际上**已经下线了**，但是他依然在调用方的**健康列表中**，调用方依然认为他健康依然会给他发送消息，最后的结果就是超时等待，不断重试而已。所以如何在服务下线时快速的让调用方感知，很重要。

> 我们思考以下大概可以有以下几种解决方案:

1、通过控制台**人工通知调用方**，让他们**手动摘除要下线的机器**，这种方式很原始也很直接。但这样对于提供方上线的过程来说**太繁琐了**，每次上线都要通知到所有调用我接口的团队，整个过程既浪费时间又没有意义，显然不能被正常接受。

2、通过服务发现机制感知，这种方式我们探讨过，因为存在一定的时间差，所以会出现一定的问题。

3、**不强依赖"服务发现"来通知调用方要下线的机器**，由**服务提供方自己来通知**行不行。在yrpc里面调用方跟服务提供方之间是**长连接**，我们可以在提供方应用内存里面维护一份调用方连接集合，当服务要关闭的时候，挨个去通知调用方去下线这台机器。



实时上第三种方式已经很好了，但是依旧会出现一些问题 ，如请求的时间点跟收到服务提供方关闭通知的时间点很接近，只比关闭通知的时间早不到1ms，如果再加上网络传输时间的话，那服务提供方收到请求的时候，它应该正在处理关闭逻辑。这就说明服务提供方关闭的时候，并没有正确处理关闭后接收到的新请求。

然而我们要明白，**发生问题并不可怕，合理的解决掉问题就可以了**，了解以上的问题我们可以看看下边的优雅关闭方案

优雅停机方案:

知道了根本原因，问题就很好解决了，因为服务提供方已经开始进入关闭流程，那么很多对象就可能已经被销毁了，关闭后再收到的请求按照正常业务请求来处理，肯定没法保证能处理的。所以我们可以在关闭的时候，设置一个请求"挡板"，挡板的作用就是告诉调用方，我已经开始进入关闭流程了，我不能再处理你这个请求了。

这种场景在生活中其实很常见，举一个例子:

银行办理业务，在交接班或者有其他要事情处理的时候，银行柜台工作人员会拿出一个纸板，放在窗口前，上面写到“**该窗口已关闭**”。在该窗口排队的人虽然有一万个不愿，也只能**换到其它窗口**办理业务，因为柜台工作人员会把当前正在办理的业务处理完后正式关闭窗口。

基于这个思路，我们可以这么处理:

1、调用方发起请求，给调用方一个特殊的响应，使用响应码标记即可，就是告诉调用方我已经收到这个请求了，但是我正在关闭，并没有处理这个请求。 2、调用方收到这个异常响应后，yrpc框架把这个节点从健康列表挪出，并把请求自动重试到其他节点，因为这个请求是没有被服务提供方处理过，所以可以安全地重试至其他节点，这样就可以实现对业务无损。

问题一：那要怎么捕获到关闭事件呢?

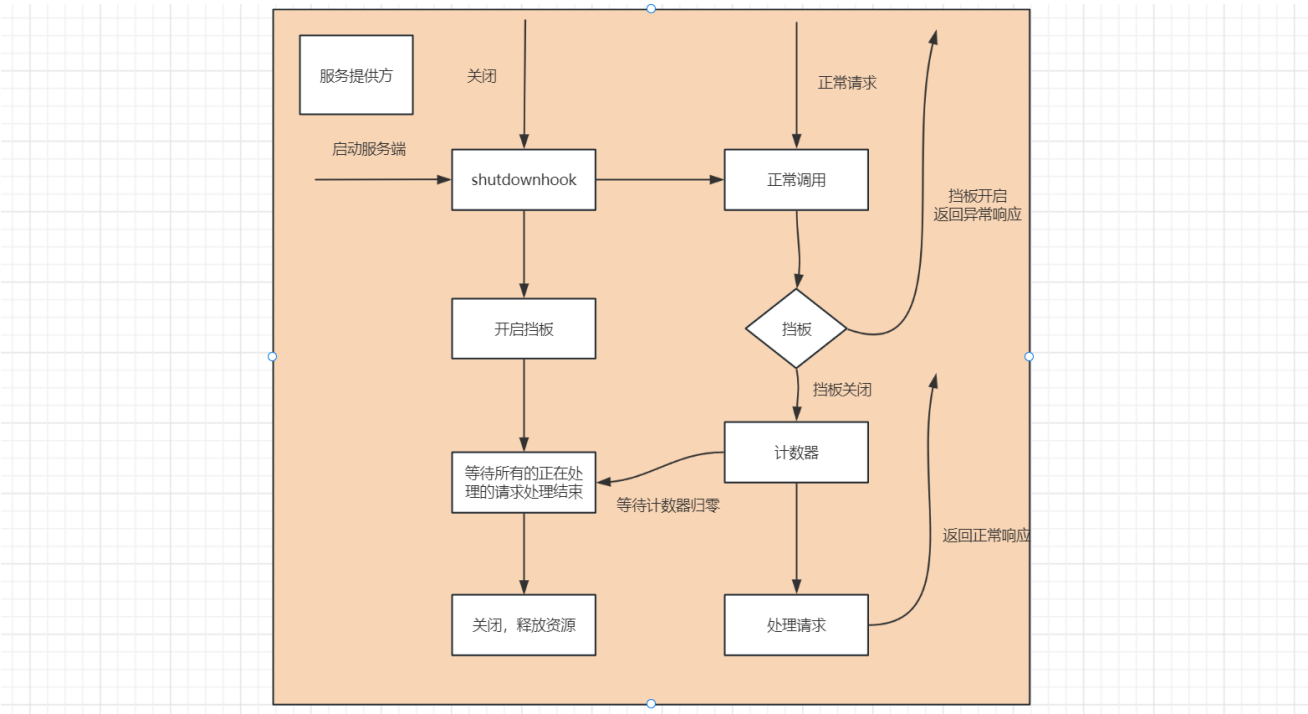
我们可以通过**捕获操作系统的进程信号来获取**，在java 语言里面，可以使用Runtime的addShutdownHook方法，可以**注册关闭的钩子。**在yrpc启动的时候，我们**提前注册关闭钩子，并在里面添加处理程序**，负责开启关闭标识和安全关闭服务，服务在关闭的时候会通知调用方下线节点。同时需要在我们调用链里面加上挡板处理器，当新的请求来的时候，会判断关闭标识，如果正在关闭，则抛出特定异常，返回特定结果。

看到这里，感觉问题已经比较好地被解决了。但细心的同学可能还会提出问题，关闭过程中已经在处理的请求会不会受到影响呢?

如果进程结束过快会造成这些请求还没有来得及应答， 此时调用方会也会抛出异常。为了尽可能地完成正在处理的请求，首先我们要把这些请求识别出来，需要有**一个标识来判断是否还有正在处理的请求**，如果没有了再关闭服务。

再举一个例子，我们经常看见停车场指示牌上提示还有多少剩余车位，这个是如何做到的呢？如果仔细观察一下，你就会发现它是每进入一辆车，剩余车位就减一，每出来一辆车剩余车位就加一。我们也可以利用这个原理，引入一个全局计数器，每开始**处理请求之前加一完成请求处理减一**，通过该计数器我们就可以快速判断是否有正在理的请求。

服务对象在关闭过程中，会拒绝新的请求，同时**根据引用计数器等待正在处理的请求全部结束之后**才会真正关闭。但考虑到有些业务请求可能处理时间长，或者存在被挂住的情况，为了避免一直等待造成应用无法正常退出，我们可以在整个 ShutdowrHook里面，加上超时时间控制， 当超过了指定时间没有结束，则强制退出应用。超时时间我建议可以设定成 10s，基本可以确保请求都处理完了。整个流程如下图所示。



关闭的钩子函数如下：

```
public class YrpcShutdownHook extends Thread {

    @Override
    public void run() {
        // 1、打开挡板    (boolean 需要线程安全)
        ShutDownHolder.BAFFLE.set(true);

        // 2、等待计数器归零（正常的请求处理结束）   AtomicInteger
        // 等待归零，继续执行  countdownLatch，最多等十秒
        long start = System.currentTimeMillis();
        while (true) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            if (ShutDownHolder.REQUEST_COUNTER.sum() == 0L
                || System.currentTimeMillis() - start > 10000) {
                break;
            }
        }
        // 3、阻塞结束后，放行。执行其他操作，如释放资源
    }
}
```

二、优雅启动

刚刚介绍了优雅停机，就是为了让服务提供方在停机应用的时候，保证所有调用方都能“安全”地切走流量，不再调用自己，从而做到对业务无损。其中实现的关键点就在于，让正在停机的服务提供方应用有状态，让调用方感知到服务提供方正在停机。

接着上一讲的内容，今天我们来聊聊优雅启动。

是不是很诧异？应用启动居然也要这么“讲究”吗？这就好比我们日常生活中的热车，行驶之前让发动机空跑一会，可以让汽车的各个部件都“热”起来，减小磨损。

换到应用上来看，原理也是一样的。运行了一段时间后的应用，执行速度会比刚启动的应用更快。这是因为在 Java 里面，在运行过程中，JVM 虚拟机会把高频的代码编译成机器码，被加载过的类也会被缓存到 JVM 缓存中，再次使用的时候不会触发临时加载，这样就使得“热点”代码的执行不用每次都通过解释，从而提升执行速度。

但是这些“临时数据”，都在我们应用重启后就消失了。重启后的这些“红利”没有了之后，如果让我们刚启动的应用就承担像停机前一样的流量，这会使应用在启动之初就处于高负载状态，从而导致调用方过来的请求可能出现大面积超时，进而对线上业务产生损害行为。

在上一讲我们说过，在微服务架构里面，上线肯定是频繁发生的，那我们总不能因为上线，就让过来的请求出现大面积超时吧？所以我们得想点办法。既然问题的关键是在于“刚重启的服务提供方因为没有预跑就承担了大流量”，那我们是不是可以通过某些方法，让应用一开始只接少许流量呢？这样低功率运行一段时间后，再逐渐提升至最佳状态。

这其实就是我今天要和你分享的重点，yrpc 里面的一个实用功能——启动预热。

1、启动预热

那什么叫启动预热呢？

简单来说，就是让刚启动的服务提供方应用不承担全部的流量，而是让它被调用的次数随着时间的移动慢慢增加，最终让流量缓和地增加到跟已经运行一段时间后的水平一样。

那在 yrpc 里面，我们该怎么实现这个功能呢？

我们现在是要控制调用方发送到服务提供方的流量。我们可以先简单地回顾下调用方发起的 yrpc 调用流程是怎样的，调用方应用通过服务发现能够获取到服务提供方的 IP 地址，然后每次发送请求前，都需要通过负载均衡算法从连接池中选择一个可用连接。那这样的话，我们是不是就可以**让负载均衡在选择连接的时候，区分一下是否是刚启动不久的应用**？对于刚启动的应用，我们**可以让它被选择到的概率特别低**，但这个概率会随着时间的推移慢慢变大，从而实现一个动态增加流量的过程。

2、延迟暴露

我们应用启动的时候都是通过 main 入口，然后顺序加载各种相关依赖的类。以 Spring 应用启动为例，在加载的过程中，Spring 容器会顺序加载 Spring Bean，如果某个 Bean 是 yrpc 服务的话，我们不光要把它注册到 Spring-BeanFactory 里面去，还要把这个 Bean 对应的接口注册到注册中心。注册中心在收到新上线的服务提供方地址的时候，会把这个地址推送到调用方应用内存中；当调用方收到这个服务提供方地址的时候，就会去建立连接发请求。

但这时候是不是存在服务提供方可能并没有启动完成的情况？因为服务提供方应用可能还在加载其它的 Bean。对于调用方来说，只要获取到了服务提供方的 IP，就有可能发起 yrpc 调用，但如果这时候服务提供方没有启动完成的话，就会导致调用失败，从而使业务受损。

那有什么办法可以避免这种情况吗？

在解决问题前，我们先看下出现上述问题的根本原因。这是因为服务提供方应用在没有启动完成的时候，调用方的请求就过来了，而调用方请求过来的原因是，服务提供方应用在启动过程中把解析到的 yrpc 服务注册到了注册中心，这就导致在后续加载没有完成的情况下服务提供方的地址就被服务调用方感知到了。

这样的话，其实我们就可以**把接口注册到注册中心的时间挪到应用启动完成后。**具体的做法就是在应用启动加载、解析 Bean 的时候，如果遇到了 yrpc 服务的 Bean，**只先把这个 Bean 注册到 Spring-BeanFactory 里面去，而并不把这个 Bean 对应的接口注册到注册中心，只有等应用启动完成后，才把接口注册到注册中心用于服务发现，从而实现让服务调用方延迟获取到服务提供方地址。**

这样是可以保证应用在启动完后才开始接入流量的，但其实这样做，我们还是没有实现最开始的目标。因为这时候应用虽然启动完成了，但并没有执行相关的业务代码，所以 JVM 内存里面还是冷的。如果这时候大量请求过来，还是会导致整个应用在高负载模式下运行，从而导致不能及时地返回请求结果。而且在实际业务中，一个服务的内部业务逻辑一般会依赖其它资源的，比如缓存数据。如果我们能在服务正式提供服务前，先完成缓存的初始化操作，而不是等请求来了之后才去加载，我们就可以降低重启后第一次请求出错的概率。

那具体怎么实现呢?

我们还是需要利用服务提供方把接口注册到注册中心的那段时间。我们可以在服务提供方应用启动后，接口注册到注册中心前，预留一个 Hook 过程，让用户可以实现可扩展的 Hook 逻辑。用户可以在 Hook 里面模拟调用逻辑，从而使 JVM 指令能够预热起来，并且用户也可以在 Hook 里面事先预加载一些资源，只有等所有的资源都加载完成后，最后才把接口注册到注册中心。整个应用启动过程如下图所示：

二、集成至springboot

我们尝试使用最原始的方法将我们的yrpc集成到springboot中，以后有时间我们进行升级，手写一个starter。

1、服务端

我们利用CommandLineRunner，等待springboot启动成功之后，让程序预热5秒钟，开始正式启动我们的服务端。

```
@Component
@Slf4j
public class YrpcStarter implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        Thread.sleep(5000);
        log.info("yrpc 开始启动...");
        YrpcBootstrap.getInstance()
            .application("first-yrpc-provider")
            // 配置注册中心
            .registry(new RegistryConfig("zookeeper://127.0.0.1:2181"))
            .serialize("jdk")
            .scan("com.ydlclass.impl")
            // 启动服务
            .start();
    }
}
```

2、消费端

我们提供了一个`@YrpcService`注解，想办法给spring的bean中注册一个相应的bean：

```
@RestController
public class HelloController {

    // 需要注入一个代理对象
    @YrpcService
    private HelloYrpc helloYrpc;

    @GetMapping("hello")
    public String hello(){
        return helloYrpc.sayHi("provider");
    }

}
```

那怎么注入呢？注入的逻辑是将一个类型为HelloYrpc的具体实现（代理对象）set进去即可。为了实现该逻辑，我们完成了代理工厂：

```
public class YrpcProxyFactory {

    private static Map<Class<?>,Object> cache = new ConcurrentHashMap<>(32);

    public static <T> T getProxy(Class<T> clazz) {

        Object bean = cache.get(clazz);
        if(bean != null){
            return (T)bean;
        }

        ReferenceConfig<T> reference = new ReferenceConfig<>();
        reference.setInterface(clazz);

        // 代理做了些什么?
        // 1、连接注册中心
        // 2、拉取服务列表
        // 3、选择一个服务并建立连接
        // 4、发送请求，携带一些信息（接口名，参数列表，方法的名字），获得结果
        YrpcBootstrap.getInstance()
            .application("first-yrpc-consumer")
            .registry(new RegistryConfig("zookeeper://127.0.0.1:2181"))
            .serialize("hessian")
            .compress("gzip")
            .group("primary")
            .reference(reference);
        T t = reference.get();
        cache.put(clazz,t);
        return t;
    }
}
```

并通过一个后置处理器，拦截所有的bean，当bean初始化结束后，判断哪些成员变量被注解YrpcService标识，则通过代理工厂获取代理对象，进行赋值操作。

```
@Component
public class YrpcProxyBeanPostProcessor implements BeanPostProcessor {

    // 他会拦截所有的bean的创建，会在每一个bean初始化后被调用
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        // 想办法给他生成一个代理
        Field[] fields = bean.getClass().getDeclaredFields();
        for (Field field : fields) {
            YrpcService yrpcService = field.getAnnotation(YrpcService.class);
            if(yrpcService != null){
                // 获取一个代理
                Class<?> type = field.getType();
                Object proxy = YrpcProxyFactory.getProxy(type);
                field.setAccessible(true);
                try {
                    field.set(bean,proxy);
                } catch (IllegalAccessException e) {
                    throw new RuntimeException(e);
                }
            }
        }

        return bean;
    }
}
```

事实上，和spring融入我们有很多更好的方法，大家也可以开洞大脑一起思考。