1. 网络编程基础

//上节课

2. 深入BIO与NIO

//上节课

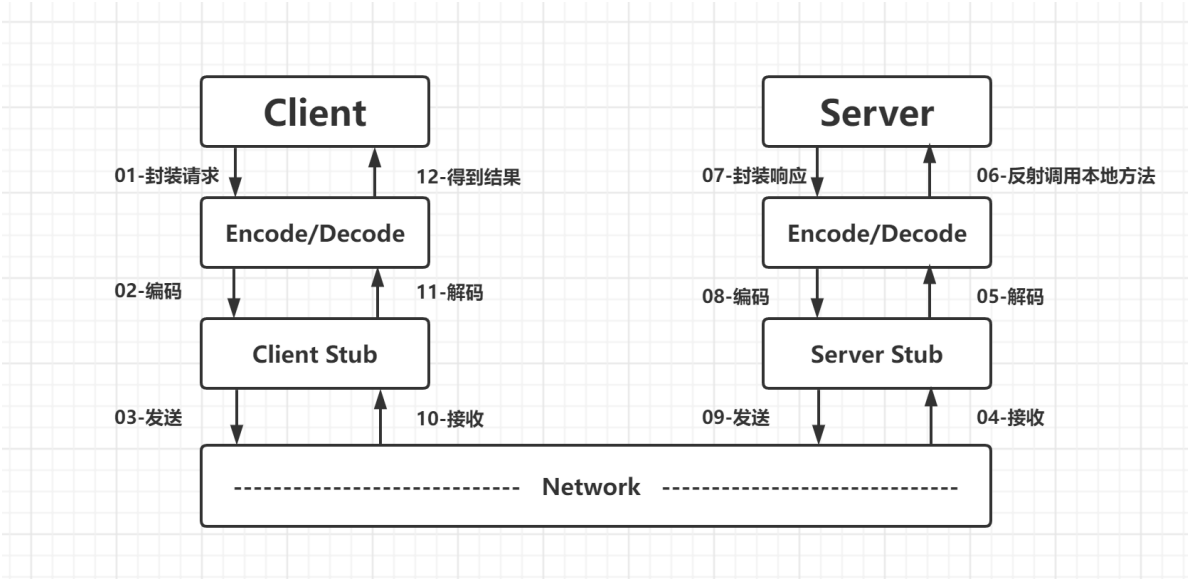
3. Netty核心技术

4. 案例01：手写一个RPC框架HeroRPC

> 先看效果：HeroRPC

4.1 RPC原理

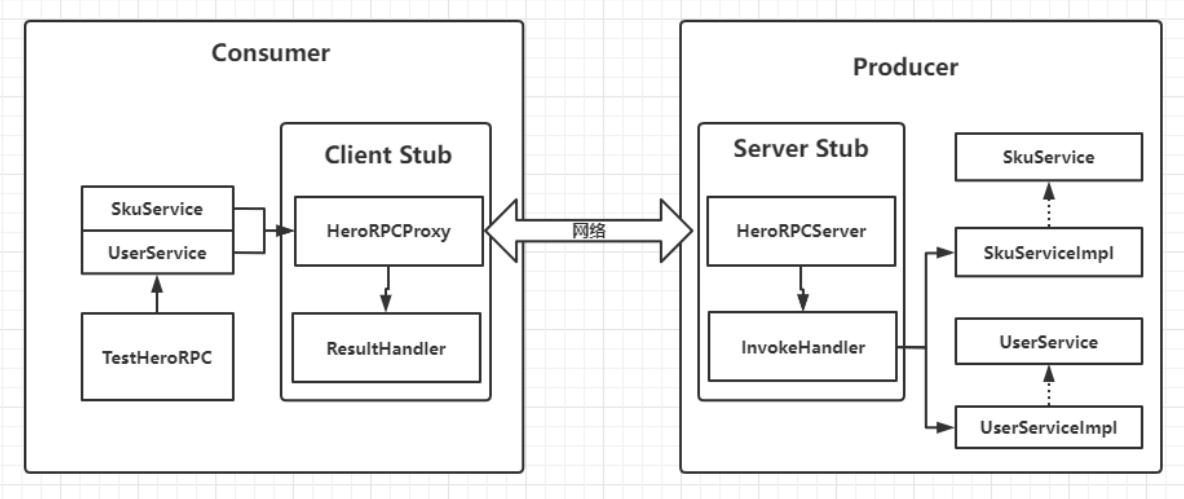
RPC（Remote Procedure Call)，即远程过程调用，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络实现的技术。常见的 RPC 框架有：阿里的 Dubbo， Spring 旗下的 Spring Cloud Feign，Google 出品的 gRPC等。



1. **服务消费方（client）以本地调用方式调用服务**
2. client stub（可以用nio，netty实现） 接收到调用后，负责将方法、参数等封装成能够进行网络传输的消息体
3. client stub 将消息进行编码并发送到服务端
4. server stub 收到消息后进行解码
5. server stub 根据解码结果调用**提供者**
6. 本地服务执行并将结果返回给 server stub
7. server stub 将返回导入结果进行编码并发送至**消费方**
8. client stub 接收到消息并进行解码
9. **服务消费方（client）得到结果**

RPC 的目标就是将 2-8 这些步骤都封装起来，用户无需关心这些细节，可以像调用本地方法一样即可完成远程服务调用。接下来我们基于 Netty 自己动手写一个 RPC，名为HeroRPC。

4.2 框架设计结构图



- 服务的调用方：两个接口【服务提供方决定】+ 一个包含 main 方法的测试类
- Client Stub: 一个客户端代理类 + 一个客户端业务处理类
  - HeroRPCProxy
  - ResultHandler
- 服务的提供方：两个接口 + 两个实现类
- Server Stub: 一个网络处理服务器 + 一个服务器业务处理类
  - HeroRPCServer
  - InvokeHandler

注意：服务调用方的接口必须跟服务提供方的接口保持一致（包路径可以不一致）

最终要实现的目标是：在 TestNettyRPC 中远程调用 SkuServiceImpl 或 UserServiceImpl中的方法

4.3 代码实现

4.3.1 Server服务的提供方：

1）SkuService接口与实现类

```
package com.hero.rpc.producer;

public interface SkuService {
    String findByName(String name);
}

```

```java
package com.hero.rpc.producer.impl;

import com.hero.rpc.producer.SkuService;

public class SkuServiceImpl implements SkuService {
    @Override
    public String findByName(String name) {
        return "sku{}:" + name;
    }
}

```

2）UserService接口与实现类

```java
package com.hero.rpc.producer;

public interface UserService {
    String findById();
}
```

```java
package com.hero.rpc.producer.impl;

import com.hero.rpc.producer.UserService;

public class UserServiceImpl implements UserService {
    @Override
    public String findById() {
        return "user{id=1,username=xiongge}";
    }
}
```

上述代码作为服务的提供方，我们分别编写了两个接口和两个实现类，供消费方远程调用。

4.3.2 Server Stub部分

1. 传输的消息封装类：

```java
package com.hero.rpc.producerStub;

//封装类信息
public class ClassInfo implements Serializable {

    private static final long serialVersionUID = 1L;

    private String className;  //类名
    private String methodName;//方法名
    private Class<?>[] types; //参数类型
    private Object[] objects;//参数列表

    public String getClassName() {
        return className;
```

```
    }

    public void setClassName(String className) {
        this.className = className;
    }

    public String getMethodName() {
        return methodName;
    }

    public void setMethodName(String methodName) {
        this.methodName = methodName;
    }

    public Class<?>[] getTypes() {
        return types;
    }

    public void setTypes(Class<?>[] types) {
        this.types = types;
    }

    public Object[] getObjects() {
        return objects;
    }

    public void setObjects(Object[] objects) {
        this.objects = objects;
    }
}
```

上述代码作为实体类用来封装消费方发起远程调用时传给服务方的数据。

2. 服务端业务处理类：Handler

```
package com.hero.rpc.producerStub;

//服务端业务处理类
public class InvokeHandler extends ChannelInboundHandlerAdapter {
    //得到某接口下某个实现类的名字
    private String getImplClassName(ClassInfo classInfo) throws Exception{
        //服务方接口和实现类所在的包路径
        String interfacePath="com.hero.rpc.producer";
        int lastDot = classInfo.getClassName().lastIndexOf(".");
        //接口名称
        String interfaceName=classInfo.getClassName().substring(lastDot);
        //接口字节码对象
        Class superClass=Class.forName(interfacePath+interfaceName);
        //反射得到某接口下的所有实现类
        Reflections reflections = new Reflections(interfacePath);
        Set<Class> ImplClassSet=reflections.getSubTypesOf(superClass);
        if(ImplClassSet.size()==0){
            System.out.println("未找到实现类");
            return null;
        }else if(ImplClassSet.size()>1){
```

```
            System.out.println("找到多个实现类，未明确使用哪一个");
            return null;
        }else {
            //把集合转换为数组
            Class[] classes=ImplClassSet.toArray(new Class[0]);
            return classes[0].getName(); //得到实现类的名字
        }
    }

    @Override  //读取客户端发来的数据并通过反射调用实现类的方法
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        ClassInfo classInfo = (ClassInfo) msg;
        Object clazz =
Class.forName(getImplClassName(classInfo)).newInstance();
        Method method =
clazz.getClass().getMethod(classInfo.getMethodName(), classInfo.getTypes());
        //通过反射调用实现类的方法
        Object result = method.invoke(clazz, classInfo.getObjects());
        ctx.writeAndFlush(result);
    }
}
```

上述代码作为业务处理类，读取消费方发来的数据，并根据得到的数据进行本地调用，然后把结果返回给消费方。

3. RPC服务端程序：HeroRPCServer

```
package com.hero.rpc.producerStub;

//RPC服务端程序
public class HeroRPCServer {
    private int port;
    public HeroRPCServer(int port) {
        this.port = port;
    }

    public void start() {
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.group(bossGroup, workerGroup)
                    .channel(NioServerSocketChannel.class)
                    .option(ChannelOption.SO_BACKLOG, 128)
                    .childOption(ChannelOption.SO_KEEPALIVE, true)
                    .localAddress(port).childHandler(
                            new ChannelInitializer<SocketChannel>() {
                                @Override
                                protected void initChannel(SocketChannel ch)
throws Exception {
                                    ChannelPipeline pipeline =
ch.pipeline();
                                    //编码器
```

```
                                    pipeline.addLast("encoder", new
ObjectEncoder());
                                    //解码器
                                    pipeline.addLast("decoder", new
ObjectDecoder(Integer.MAX_VALUE, ClassResolvers.cacheDisabled(null)));
                                    //服务端业务处理类
                                    pipeline.addLast(new InvokeHandler());
                                }
                            });
            ChannelFuture future = serverBootstrap.bind(port).sync();
            System.out.println("......Hero RPC is ready......");
            future.channel().closeFuture().sync();
        } catch (Exception e) {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }

    public static void main(String[] args) throws Exception {
        new HeroRPCServer(9999).start();
    }
}
```

上述代码是用 Netty 实现的网络服务器，采用 Netty 自带的 ObjectEncoder 和 ObjectDecoder作为编解码器（为了降低复杂度，这里并没有使用第三方的编解码器），当然实际开发时也可以采用 JSON 或 XML。

4.3.3 Client Stub部分

1）客户端业务处理类：ResultHandler

```
package com.hero.rpc.consumerStub;

//客户端业务处理类
public class ResultHandler extends ChannelInboundHandlerAdapter {

    private Object response;
    public Object getResponse() {
        return response;
    }

    @Override //读取服务端返回的数据(远程调用的结果)
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        response = msg;
        ctx.close();
    }
}
```

上述代码作为客户端的业务处理类读取远程调用返回的数据

2）RPC客户端程序：RPC远程代理HeroRPCProxy

```
package com.hero.rpc.consumerStub;

//客户端代理类
public class HeroRPCProxy {
    //根据接口创建代理对象
    public static Object create(Class target) {

        return Proxy.newProxyInstance(target.getClassLoader(), new Class[]
{target}, new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
                    throws Throwable {
                //封装ClassInfo
                ClassInfo classInfo = new ClassInfo();
                classInfo.setClassName(target.getName());
                classInfo.setMethodName(method.getName());
                classInfo.setObjects(args);
                classInfo.setTypes(method.getParameterTypes());

                //开始用Netty发送数据
                EventLoopGroup group = new NioEventLoopGroup();
                ResultHandler resultHandler = new ResultHandler();
                try {
                    Bootstrap b = new Bootstrap();
                    b.group(group)
                            .channel(NioSocketChannel.class)
                            .handler(new ChannelInitializer<SocketChannel>()
{
                                @Override
                                public void initChannel(SocketChannel ch)
throws Exception {
                                    ChannelPipeline pipeline =
ch.pipeline();
                                    //编码器
                                    pipeline.addLast("encoder", new
ObjectEncoder());
                                    //解码器  构造方法第一个参数设置二进制数据的最
大字节数  第二个参数设置具体使用哪个类解析器
                                    pipeline.addLast("decoder", new
ObjectDecoder(Integer.MAX_VALUE, ClassResolvers.cacheDisabled(null)));
                                    //客户端业务处理类
                                    pipeline.addLast("handler",
resultHandler);
                                }
                            });
                    ChannelFuture future = b.connect("127.0.0.1",
9999).sync();
                    future.channel().writeAndFlush(classInfo).sync();
                    future.channel().closeFuture().sync();
                } finally {
                    group.shutdownGracefully();
                }
                return resultHandler.getResponse();
```

```
            }
        });
    }
}
```

上述代码是用 Netty 实现的客户端代理类，采用 Netty 自带的 ObjectEncoder 和 ObjectDecoder作为编解码器（为了降低复杂度，这里并没有使用第三方的编解码器），当然实际开发时也可以采用 JSON 或 XML。

4.3.4 Client服务的调用方-消费方

```
package com.hero.rpc.consumer;

//服务调用方
public class TestHeroRPC {
    public static void main(String [] args){

        //第1次远程调用
        SkuService skuService=(SkuService)
HeroRPCProxy.create(SkuService.class);
        System.out.println(skuService.findByName("uid"));

        //第2次远程调用
        UserService userService =  (UserService)
HeroRPCProxy.create(UserService.class);
        System.out.println(userService.findById());

    }
}
```

消费方不需要知道底层的网络实现细节，就像调用本地方法一样成功发起了两次远程调用。

5. 案例02：手写一个Tomcat-V2.0

> **先看效果：herocat**

本案例，咱们手写的是一个Web 容器命名为HeroCat，类似于Tomcat 的容器，用于处理HTTP请求。

Servlet 规范很复杂，所以本 Web 容器并没有去实现JavaEE 的Servlet 规范，所以说并不算是一个 Servlet 容器。但是，其是类比着Tomcat 来写的，这里定义了自己的请求、响应及Servlet规范，分别命名为了HeroRequest， HeroResponse 与HeroServlet。

5.1 HeroCat容器需求

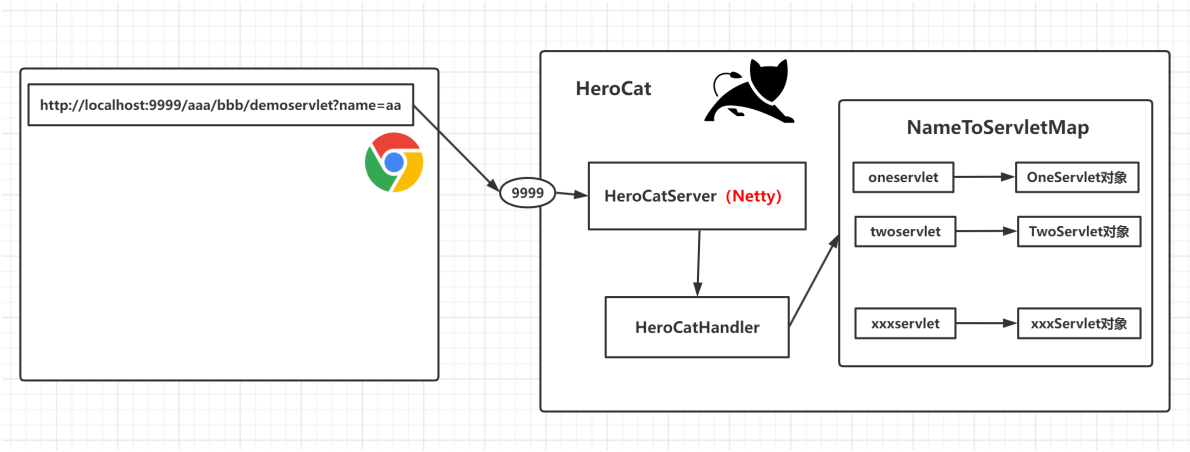
需求：软件工程师自定义一个Tomcat提供给码农使用，码农只需要按照规定步骤，即可编写出自己的应用程序发布到HeroCat中供用户使用。

5.1.1 角色

Web容器（HeroCat）相关的角色：

- **HeroCat开发者**，编写**Hero核心代码**的软件工程师，下文简称：**工程师**
- **HeroCat使用者**，应用程序**业务功能**开发的软件工程师，下文简称：**码农**
- **应用程序使用者**：用户

设计思路：



5.1.2 使用步骤-码农

码农使用HeroCat的步骤：

- 码农编写自己的应用程序：
  - 导入HeroCat依赖坐标，并编写启动类
  - 将自定义Servlet 放置到指定包下：例如`com.hero.webapp`
- 码农发布自己的服务：
  - 码农将自己的接口URL按照固定规则发布：
    - 按照后缀，`.do`、`.action`、无后缀
  - 不管用何种规则：都将映射到**自定义的Servlet**（类名映射，忽略大小写）举例

```
http://localhost:8080/aaa/bbb/userservlet?name=xiong
```

- 用户在访问应用程序：
  - 按照URL地址访问服务
  - 如果没有指定的Servlet，则访问默认的Servlet

5.1.3 HeroCat开发思路-工程师

工程师实现HeroCat思路：

- 第一步：创建HeroCat工程，导入依赖坐标
- 第二步：定义Servlet规范，HeroRequest、HeroResponse、HeroServlet
  - Servlet的规范其实是**语言层面**定义JavaEE
- 第三步：实现Servlet规范
  - HttpHeroRequest
  - HttpHeroResponse
  - DefaultHeroServlet【兜底】
- 第四步：编写HeroCat核心代码：
  - HeroCatServer基于Netty实现：Servlet容器
  - HeroCatHandler处理请求，映射到Servlet的容器的自定义Servlet（Map容器）中去
- 第五步：打包发布HeroCat

5.2 创建工程

5.2.1 创建工程

创建一个普通的Maven 的Java 工程herocat。

5.2.2 导入依赖

```
<dependencies>
    <!-- netty-all依赖 -->
    <dependency>
        <groupId>io.netty</groupId>
        <artifactId>netty-all</artifactId>
        <version>4.1.36.Final</version>
    </dependency>
    <!--lombok依赖-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.6</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.dom4j</groupId>
        <artifactId>dom4j</artifactId>
        <version>2.1.3</version>
    </dependency>
    <dependency>
        <groupId>jaxen</groupId>
        <artifactId>jaxen</artifactId>
        <version>1.1.6</version>
    </dependency>
</dependencies>
```

5.3 定义Servlet 规范

5.3.1 定义请求接口HeroRequest

```java
package com.hero.servlet;

/**
 * Servlet规范之请求规范
 */
public interface HeroRequest {
    // 获取URI，包含请求参数，即?后的内容
    String getUri();
    // 获取请求路径，其不包含请求参数
    String getPath();
    // 获取请求方法（GET、POST等）
    String getMethod();
    // 获取所有请求参数
    Map<String, List<String>> getParameters();
    // 获取指定名称的请求参数
    List<String> getParameters(String name);
    // 获取指定名称的请求参数的第一个值
    String getParameter(String name);
}

```

5.3.2 定义响应接口HeroResponse

```java
package com.hero.servlet;

/**
 * Servlet规范之响应规范
 */
public interface HeroResponse {
    // 将响应写入到Channel
    void write(String content) throws Exception;
}

```

5.3.3 定义Servlet 规范HeroServlet

```java
package com.hero.servlet;

/**
 * 定义Servlet规范
 */
public abstract class HeroServlet {
    //处理Http的get请求
    public abstract void doGet(HeroRequest request, HeroResponse response)
throws Exception;
    //处理Http的post请求
    public abstract void doPost(HeroRequest request, HeroResponse response)
throws Exception;
}

```

5.4 定义Tomcat服务器

5.4.1 定义HttpHeroRequest 类

```java
package com.hero.herocat;

/**
 * HeroCat中对Servlet规范的默认实现
 */
public class HttpHeroRequest implements HeroRequest {
    private HttpRequest request;

    public HttpHeroRequest(HttpRequest request) {
        this.request = request;
    }

    @Override
    public String getUri() {
        return request.uri();
    }

    @Override
    public String getPath() {
        QueryStringDecoder decoder = new QueryStringDecoder(request.uri());
        return decoder.path();
    }

    @Override
    public String getMethod() {
        return request.method().name();
    }

    @Override
    public Map<String, List<String>> getParameters() {
        QueryStringDecoder decoder = new QueryStringDecoder(request.uri());
```

```
        return decoder.parameters();
    }

    @Override
    public List<String> getParameters(String name) {
        return getParameters().get(name);
    }

    @Override
    public String getParameter(String name) {
        List<String> parameters = getParameters(name);
        if (parameters == null || parameters.size() == 0) {
            return null;
        }
        return parameters.get(0);
    }
}

```

5.4.2 定义HttpHeroResponse类

```
package com.hero.herocat;

/**
 * HeroCat中对Servlet规范的默认实现
 */
public class HttpHeroResponse implements HeroResponse {
    private HttpRequest request;
    private ChannelHandlerContext context;

    public HttpHeroResponse(HttpRequest request, ChannelHandlerContext
context) {
        this.request = request;
        this.context = context;
    }

    @Override
    public void write(String content) throws Exception {

        // 处理content为空的情况
        if (StringUtil.isNullOrEmpty(content)) {
            return;
        }

        // 创建响应对象
        FullHttpResponse response = new
DefaultFullHttpResponse(HttpVersion.HTTP_1_1,
                HttpResponseStatus.OK,
                // 根据响应体内容大小为response对象分配存储空间
                Unpooled.wrappedBuffer(content.getBytes("UTF-8")));

        // 获取响应头
        HttpHeaders headers = response.headers();
        // 设置响应体类型
        headers.set(HttpHeaderNames.CONTENT_TYPE, "text/json");
```

```
        // 设置响应体长度
        headers.set(HttpHeaderNames.CONTENT_LENGTH,
response.content().readableBytes());
        // 设置缓存过期时间
        headers.set(HttpHeaderNames.EXPIRES, 0);
        // 若HTTP请求是长连接，则响应也使用长连接
        if (HttpUtil.isKeepAlive(request)) {
            headers.set(HttpHeaderNames.CONNECTION,
HttpHeaderValues.KEEP_ALIVE);
        }
        // 将响应写入到Channel
        context.writeAndFlush(response);
    }
}
```

5.4.3 定义DefaultHeroServlet 类

```
package com.hero.herocat;

/**
 * HeroCat中对Servlet规范的默认实现
 */
public class DefaultHeroServlet extends HeroServlet {
    @Override
    public void doGet(HeroRequest request, HeroResponse response) throws
Exception {
        // http://localhost:8080/aaa/bbb/oneservlet?name=xiong
        // path: /aaa/bbb/oneservlet?name=xiong
        String uri = request.getUri();
        String name = uri.substring(0, uri.indexOf("?"));
        response.write("404 - no this servlet : " + name);
    }


    @Override
    public void doPost(HeroRequest request, HeroResponse response) throws
Exception {
        doGet(request, response);
    }
}
```

5.4.4 定义服务器类HeroCatServer

HeroCat的Servlet容器

- 第一个 map：key 为指定的Servlet 的名称，value 为该Servlet 实例
- 第二个 map：key 为指定的Servlet 的名称，value 为该Servlet 的全限定性类名

```
package com.hero.herocat;

/**
 * HeroCat功能的实现
 */
public class HeroCatServer {
```

```
    // key为HeroServlet的简单类名，value为对应HeroServlet实例
    private Map<String, HeroServlet> nameToServletMap = new
ConcurrentHashMap<>();
    // key为HeroServlet的简单类名，value为对应HeroServlet类的全限定性类名
    private Map<String, String> nameToClassNameMap = new HashMap<>();
    // HeroServlet存放位置
    private String basePackage;

    public HeroCatServer(String basePackage) {
        this.basePackage = basePackage;
    }

    // 启动tomcat
    public void start() throws Exception {
        // 加载指定包中的所有Servlet的类名
        cacheClassName(basePackage);
        // 启动server服务
        runServer();
    }

    private void cacheClassName(String basePackage) {
        // 获取指定包中的资源
        URL resource = this.getClass().getClassLoader()
                // com.abc.webapp  =>  com/abc/webapp
                .getResource(basePackage.replaceAll("\\.", "/"));
        // 若目录中没有任何资源，则直接结束
        if (resource == null) {
            return;
        }

        // 将URL资源转换为File资源
        File dir = new File(resource.getFile());
        // 遍历指定包及其子孙包中的所有文件，查找所有.class文件
        for (File file : dir.listFiles()) {
            if (file.isDirectory()) {
                // 若当前遍历的file为目录，则递归调用当前方法
                cacheClassName(basePackage + "." + file.getName());
            } else if (file.getName().endsWith(".class")) {
                String simpleClassName = file.getName().replace(".class",
"").trim();
                // key为简单类名，value为全限定性类名
                nameToClassNameMap.put(simpleClassName.toLowerCase(),
basePackage + "." + simpleClassName);
            }
        }
        // System.out.println(nameToClassNameMap);
    }

    private void runServer() throws Exception {

        EventLoopGroup parent = new NioEventLoopGroup();
        EventLoopGroup child = new NioEventLoopGroup();

        try {
            ServerBootstrap bootstrap = new ServerBootstrap();
```

```
            bootstrap.group(parent, child)
                    // 指定存放请求的队列的长度
                    .option(ChannelOption.SO_BACKLOG, 1024)
                    // 指定是否启用心跳机制来检测长连接的存活性，即客户端的存活性
                    .childOption(ChannelOption.SO_KEEPALIVE, true)
                    .channel(NioServerSocketChannel.class)
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch) throws Exception {
                            ChannelPipeline pipeline = ch.pipeline();
                            pipeline.addLast(new HttpServerCodec());
                            pipeline.addLast(new HeroCatHandler(nameToServletMap, nameToClassNameMap));
                        }
                    });
            int port = initPort();
            ChannelFuture future = bootstrap.bind(port).sync();
            System.out.println("HeroCat启动成功：监听端口号为:" + port);
            future.channel().closeFuture().sync();
        } finally {
            parent.shutdownGracefully();
            child.shutdownGracefully();
        }
    }
    //初始化端口
    private int initPort() throws DocumentException {
        //初始化端口
        //读取配置文件Server.xml中的端口号
        InputStream in = HeroCatServer.class.getClassLoader().getResourceAsStream("server.xml");
        //获取配置文件输入流
        SAXReader saxReader = new SAXReader();
        Document doc = saxReader.read(in);
        //使用SAXReader + XPath读取端口配置
        Element portEle = (Element) doc.selectSingleNode("//port");
        return Integer.valueOf(portEle.getText());
    }
}
```

5.4.5 定义服务端处理器HeroCatHandler

```
package com.hero.herocat;

/**
 * HeroCat服务端处理器
 *
 *    1）从用户请求URI中解析出要访问的Servlet名称
 *    2）从nameToServletMap中查找是否存在该名称的key。若存在，则直接使用该实例，否则执行第3）步
 *    3）从nameToClassNameMap中查找是否存在该名称的key，若存在，则获取到其对应的全限定性类名，
 *       使用反射机制创建相应的serlet实例，并写入到nameToServletMap中，若不存在，则直接访问默认Servlet
 *
```

```
 */
public class HeroCatHandler extends ChannelInboundHandlerAdapter {
    private Map<String, HeroServlet> nameToServletMap;
    private Map<String, String> nameToClassNameMap;

    public HeroCatHandler(Map<String, HeroServlet> nameToServletMap,
Map<String, String> nameToClassNameMap) {
        this.nameToServletMap = nameToServletMap;
        this.nameToClassNameMap = nameToClassNameMap;
    }

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        if (msg instanceof HttpRequest) {
            HttpRequest request = (HttpRequest) msg;
            String uri = request.uri();
            // 从请求中解析出要访问的Servlet名称
            //aaa/bbb/twoservlet?name=aa
            String servletName = uri.substring(uri.lastIndexOf("/") + 1,
uri.indexOf("?"));

            HeroServlet heroServlet = new DefaultHeroServlet();
            if (nameToServletMap.containsKey(servletName)) {
                heroServlet = nameToServletMap.get(servletName);
            } else if (nameToClassNameMap.containsKey(servletName)) {
                // double-check，双重检测锁
                if (nameToServletMap.get(servletName) == null) {
                    synchronized (this) {
                        if (nameToServletMap.get(servletName) == null) {
                            // 获取当前Servlet的全限定性类名
                            String className =
nameToClassNameMap.get(servletName);
                            // 使用反射机制创建Servlet实例
                            heroServlet = (HeroServlet)
Class.forName(className).newInstance();
                            // 将Servlet实例写入到nameToServletMap
                            nameToServletMap.put(servletName, heroServlet);
                        }
                    }
                }
            } //  end-else if

            // 代码走到这里，servlet肯定不空
            HeroRequest req = new HttpHeroRequest(request);
            HeroResponse res = new HttpHeroResponse(request, ctx);
            // 根据不同的请求类型，调用heroServlet实例的不同方法
            if (request.method().name().equalsIgnoreCase("GET")) {
                heroServlet.doGet(req, res);
            } else if(request.method().name().equalsIgnoreCase("POST")) {
                heroServlet.doPost(req, res);
            }
            ctx.close();
        }
    }
```

```
61
62      @Override
63      public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
    throws Exception {
64          cause.printStackTrace();
65          ctx.close();
66      }
67  }
```

5.4.6 定义启动类HeroCat

```
1  package com.hero.herocat;
2
3  public class HeroCat {
4      public static void main(String[] args) throws Exception {
5          HeroCatServer server = new HeroCatServer("com.hero.webapp");
6          server.start();
7      }
8  }
```

5.5 定义业务SkuServlet

```
1  package com.hero.webapp;
2
3  /**
4   * 业务Servlet
5   */
6  public class SkuServlet extends HeroServlet {
7      @Override
8      public void doGet(HeroRequest request, HeroResponse response) throws
    Exception {
9          String uri = request.getUri();
10         String path = request.getPath();
11         String method = request.getMethod();
12         String name = request.getParameter("name");
13
14         String content = "uri = " + uri + "\n" +
15                          "path = " + path + "\n" +
16                          "method = " + method + "\n" +
17                          "param = " + name;
18         response.write(content);
19     }
20
21     @Override
22     public void doPost(HeroRequest request, HeroResponse response) throws
    Exception {
23         doGet(request, response);
24     }
25 }
```

6. 案例03：600W+连接网络应用实战

6.1 Disruptor框架

6.1.1 什么是Disruptor?

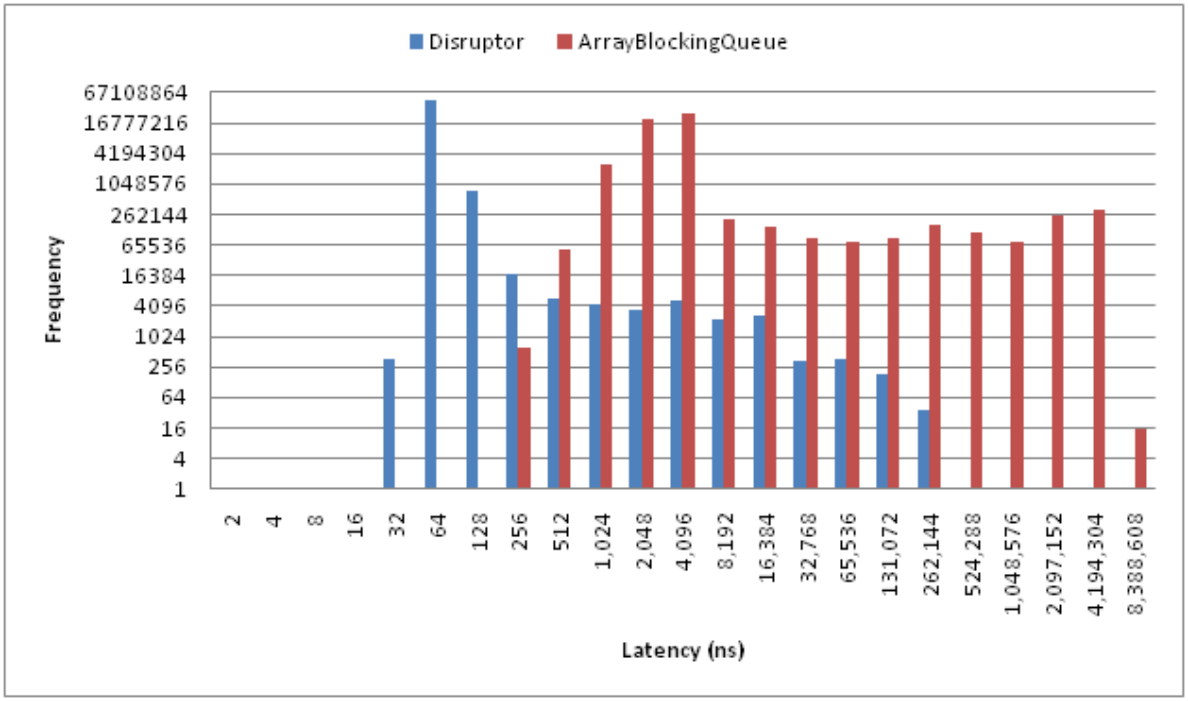
LMAX是英国外汇交易公司，目标是成为世界上最快的交易平台。为了实现这一点，这家公司的技术团队使用 Java 平台实现**非常低的延迟和高吞吐量**的交易系统。经过一系列性能测试表明，**使用队列在系统的各个阶段之间传递数据会导致延迟**，**当然吞吐量也就很难上的去**，因此他们技术团队专注于优化这个领域，所以Disruptor诞生了。

Disruptor是一个通用解决方案，用于解决并发编程中的难题（低延迟与高吞吐量）。其本质还是一个**队列（环形）**，与其他队列类似，也是基于生产者消费者模式设计，只不过这个队列很特别是一个环形队列。这个队列能够在无锁的条件下进行并行消费，也可以根据消费者之间的依赖关系进行先后次序消费。

说的简单点：生产者向RingBuffer中写入元素，消费从RingBuffer中消费元素。基于 Disruptor 开发的系统单线程能支撑每秒 600 万订单。

它与并发编程中的阻塞队列有什么不同?

- **低延时高通吐**
- 快，它实在是太快了



6.1.2 通用步骤

1. 创建工厂类，用于生产Event对象
2. 创建Consumer监听类，用于监听，并处理Event
3. 创建Disruptor对象，并初始化一系列参数：工厂类、RingBuffer大小、线程池、单生产者或多生产者、Event等待策略
4. 编写Producer组件，向Disruptor容器中去投递Event

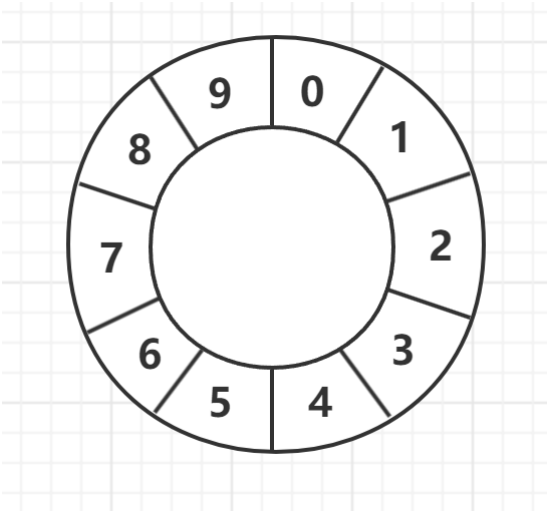
6.1.3 核心概念

1）Disruptor

它是一个辅助类，持有**RingBuffer**、消费者线程池Executor、消费者仓库ConsumerRepository等引用。

2）RingBuffer环形缓存器

RingBuffer基于数组的实现，数据结构是个首尾相接的环，用做在不同上下文（线程）间传递数据的buffer。RingBuffer拥有一个Sequencer序号器，这个序号器指向数组中下一个可用元素。



3）Sequencer序号器

Sequencer序号器是Disruptor核心。

此接口有两个实现类：

- SingleProducerSequencer 单生产者
- MultiProducerSequencer 多生产者

4）Sequence序号

Sequencer序号器中有Sequence序号，通过顺序递增的序号来编号和管理进行交换的Event。

Event的处理过程总是沿着序号逐个递增处理。

一个Sequence用于跟踪标识某个特定的事件处理者的处理进度。Producer和Consumer都有自己的Sequence，用来判断Consumer和Producer之间平衡，防止生产快，消费慢或生产慢，消费快等情况【上下游速度不一致问题】。相当于标识进度了

- 解决上下游消费速度不一致问题
- 异步提速
- 削峰填谷

5）WaitStrategy等待策略

决定一个Consumer将如何等待Producer将Event置入RingBuffer

主要策略有：

- **BlockingWaitStrategy：阻塞等待策略**，最低效的策略，但其对CPU的消耗最小并且在各种不同部署环境中能提供更加一致的性能表现。
- **SleepingWaitStrategy：休眠等待策略**，性能表现跟BlockingWaitStrategy差不多，对CPU的消耗也类似，但其对生产者线程的影响最小，适合用于异步日志类似的场景。
- **YieldingWaitStrategy：产生等待策略，**性能最好，适合用于低延迟的系统，在要求极高性能且事件处理线程数小于CPU逻辑核心数的场景中，推荐使用。是**无锁并行**

6）Event

从Producer到Consumer过程中所处理的数据单元

7）EventHandler

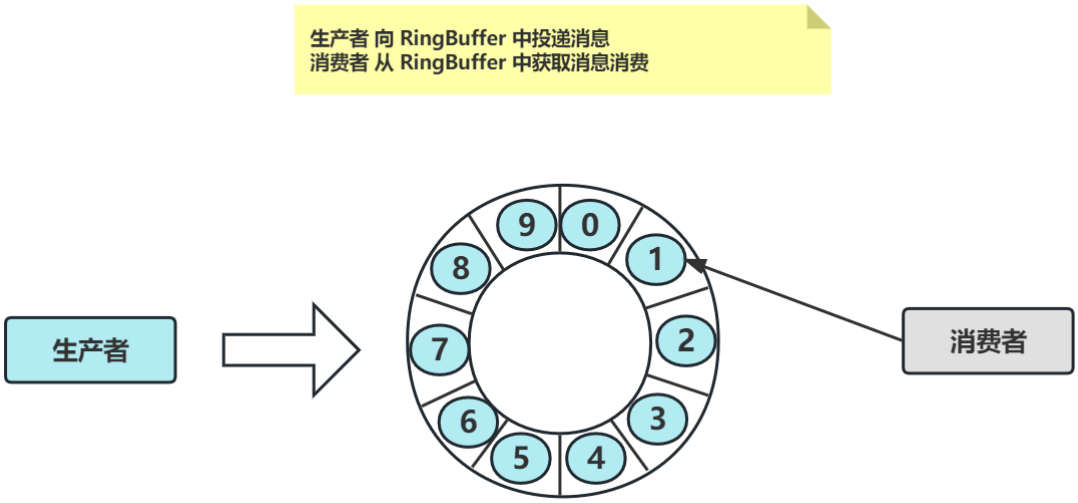
由用户实现，并且代表Disruptor中的一个消费者的接口，我们的消费者逻辑都需要写在这里。

6.2 案例04：单生产者单消费者

目标：演示Disruptor高性能队列的基本用法，创建循环100个订单消息并消费之

步骤：

1. 创建OrderEventFactory来产生OrderEvent实例对象
2. 创建Consumer处理者OrderEventHandler，当Producer投递一条条数据时此Handler进行处理
3. 编写核心类Main创建disruptor对象，让其与Consumer处理者OrderEventHandler绑定，启动disruptor
4. 通过disruptor对象获取到ringBuffer容器。
5. 创建生产者OrderEventProducer，将消息放到RingBuffer容器
6. 循环100次，通过sendData()投递消息。sendData()方法的最后将消息发布出去，只有发布出去，消费者才能收到



1）OrderEvent

定义需要处理的OrderEvent类

```
1  package com.hero.disruptor.demo;
2
3  //订单对象，生产者要生产订单对象，消费者消费订单对象
4  public class OrderEvent {
5
6      private long value; //订单的价格
7
8      public long getValue() {
9          return value;
10     }
11
12     public void setValue(long value) {
13         this.value = value;
14     }
15 }
```

2）OrderEventFactory

定义工厂类OrderEventFactory，用于创建OrderEvent对象。

```
1  package com.hero.disruptor.demo;
2
3  //建立一个工厂类，用于创建Event的实例（OrderEvent）
4  public class OrderEventFactory implements EventFactory<OrderEvent> {
5      @Override
6      public OrderEvent newInstance() {
7          return new OrderEvent(); //返回空的数据对象，不是null,OrderEvent,value属
   性还没有赋值。
8      }
9  }
10
```

3）OrderEventHandler

定义Event监听及处理类OrderEventHandler，用于处理OrderEvent

```
package com.hero.disruptor.demo;

//消费者
public class OrderEventHandler implements EventHandler<OrderEvent> {

    @Override
    public void onEvent(OrderEvent orderEvent, long l, boolean b) throws Exception {
        System.err.println("消费者:"+ orderEvent.getValue()); //取出订单对象的价格。
    }
}
```

4）TestDisruptor

定义测试类，创建Disruptor对象，并初始化一系列参数：工厂类、RingBuffer大小、线程池、单生产者或多生产者、Event等待策略。

```
package com.hero.disruptor.demo;

public class TestDisruptor {
    public static void main(String[] args) {
        //做一些准备工作
        OrderEventFactory orderEventFactory=new OrderEventFactory();
        int ringBufferSize=8;
        ExecutorService executor=Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
        /*
          1.eventFactory:消息(event)工厂对象
          2.ringBufferSize: 容器的长度
          3.executor:线程池，建议使用自定义的线程池，线程上限。
          4.ProducerType:单生产者或多生产者
          5.waitStrategy:等待策略
         */
        //1.实例化disruptor对象
        Disruptor<OrderEvent> disruptor=new Disruptor<OrderEvent>(orderEventFactory,
                ringBufferSize,
                executor,
                ProducerType.SINGLE,
                new BlockingWaitStrategy());
        //2.添加消费者的监听（去构建disruptor与消费者的一个关联关系）
        disruptor.handleEventsWith(new OrderEventHandler());
        //3.启动disruptor
        disruptor.start();
        //4.取到容器后通过生产者去生产消息
        //获取实际存储数据的容器RingBuffer
        RingBuffer<OrderEvent> ringBuffer=disruptor.getRingBuffer();
        //生产者
        OrderEventProducer producer=new OrderEventProducer(ringBuffer);
```

```
        //先初始化ByteBuffer长度为8个字节
        ByteBuffer bb=ByteBuffer.allocate(8);
        //生产100个orderEvent->value->i 0-99
        for(long i=0;i<100;i++){
            bb.putLong(0,i);
            producer.sendData(bb);
        }
        disruptor.shutdown();;
        executor.shutdown();
    }
}
```

5）OrderEventProducer

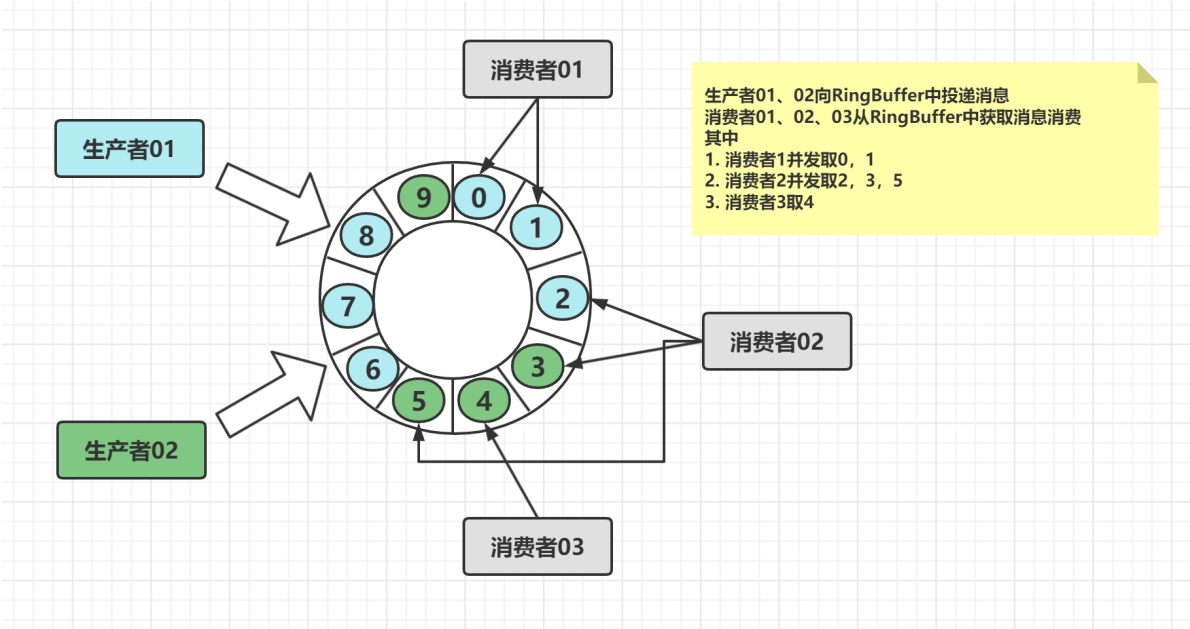
定义Producer类，向Disruptor容器中去投递数据。

```
package com.hero.disruptor.demo;

public class OrderEventProducer {

    //ringBuffer存储数据的一个容器
    private RingBuffer<OrderEvent> ringBuffer;

    public OrderEventProducer(RingBuffer<OrderEvent> ringBuffer) {
        this.ringBuffer = ringBuffer;
    }

    //生产者投递的数据
    public void sendData(ByteBuffer data) {
        //1.在生产者发送消息时，首先要从ringBuffer中找一个可用的序号。
        long sequence = ringBuffer.next();
        try {
            //2.根据这个序号找到具体的OrderEvent元素，此时获取到的OrderEvent对象是一个没有被赋值的空对象。value
            OrderEvent event = ringBuffer.get(sequence);
            event.setValue(data.getLong(0));//设置订单价格
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            //4.提交发布操作
            //生产者最后要发布消息，
            ringBuffer.publish(sequence);
        }
    }
}
```

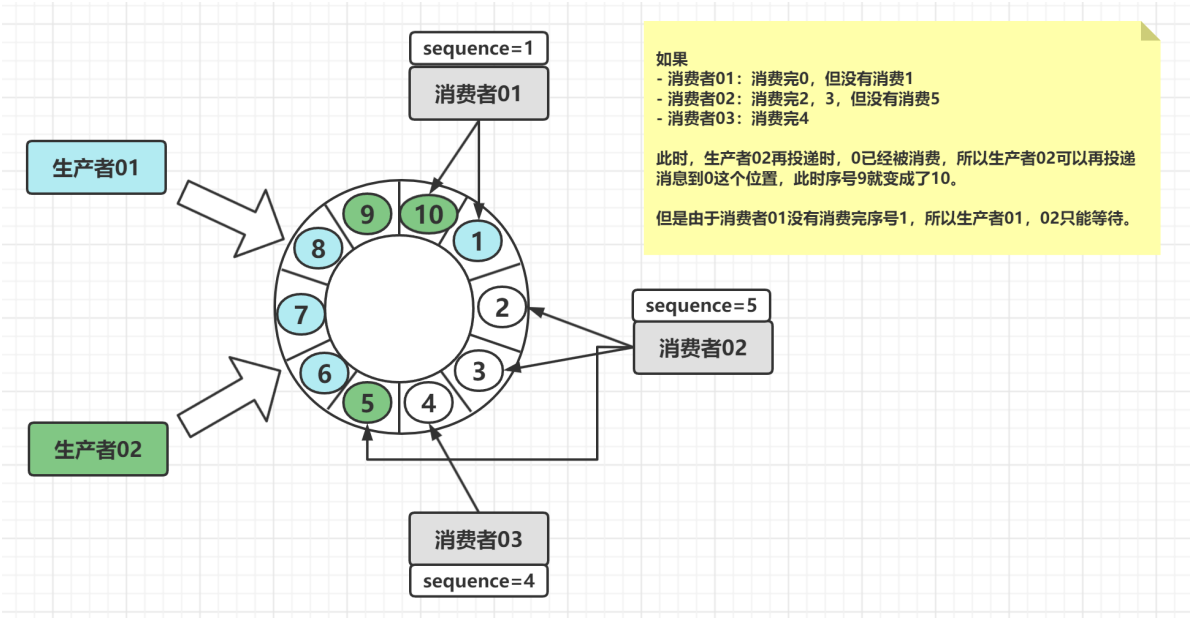
```
C:\develop\java\jdk1.8.0_171\bin\java.exe ...
消费者:0
消费者:1
消费者:2
消费者:3
消费者:4
消费者:5
消费者:6
消费者:7
消费者:8
```

6.3 案例05：多生产者和多消费者

时刻01



时刻02



1）Order

```java
package com.hero.disruptor.multi;
/**
 * Disruptor中的 Event
 */
public class Order {

    private String id;
    private String name;
    private double price;

    public Order(){
    }

    //getter、setter
}
```

2）ConsumerHandler

```java
package com.hero.disruptor.multi;

public class ConsumerHandler implements WorkHandler<Order> {

    //每个消费者有自己的id
    private String comsumerId;

    //计数统计，多个消费者，所有的消费者总共消费了多个消息。
    private static AtomicInteger count=new AtomicInteger(0);

    private Random random=new Random();

    public ConsumerHandler(String comsumerId){
        this.comsumerId=comsumerId;
    }
    //当生产者发布一个sequence，ringbuffer中一个序号，里面生产者生产出来的消息，生产者最后publish发布序号
    //消费者会监听，如果监听到，就会ringbuffer去取出这个序号，取到里面消息
    @Override
    public void onEvent(Order event) throws Exception {
        //模拟消费者处理消息的耗时，设定1-4毫秒之间
        TimeUnit.MILLISECONDS.sleep(1*random.nextInt(5));

        System.err.println("当前消费者:"+this.comsumerId+",消费信息ID:"+event.getId());
        //count计数器增加+1，表示消费了一个消息
        count.incrementAndGet();

    }

    //返回所有消费者总共消费的消息的个数。
    public int getCount(){
```

```
        return count.get();
    }
}
```

3）Producer

```
package com.hero.disruptor.multi;

public class Producer {

    private RingBuffer<Order> ringBuffer;

    //为生产者绑定ringbuffer
    public Producer(RingBuffer<Order> ringBuffer){
        this.ringBuffer=ringBuffer;
    }

    //发送数据
    public void sendData(String uuid){
        //1.获取到可用sequence
        long sequence=ringBuffer.next();
        try{
            Order order=ringBuffer.get(sequence);
            order.setId(uuid);
        }finally {
            //发布序号
            ringBuffer.publish(sequence);
        }
    }
}

```

4）TestMultiDisruptor

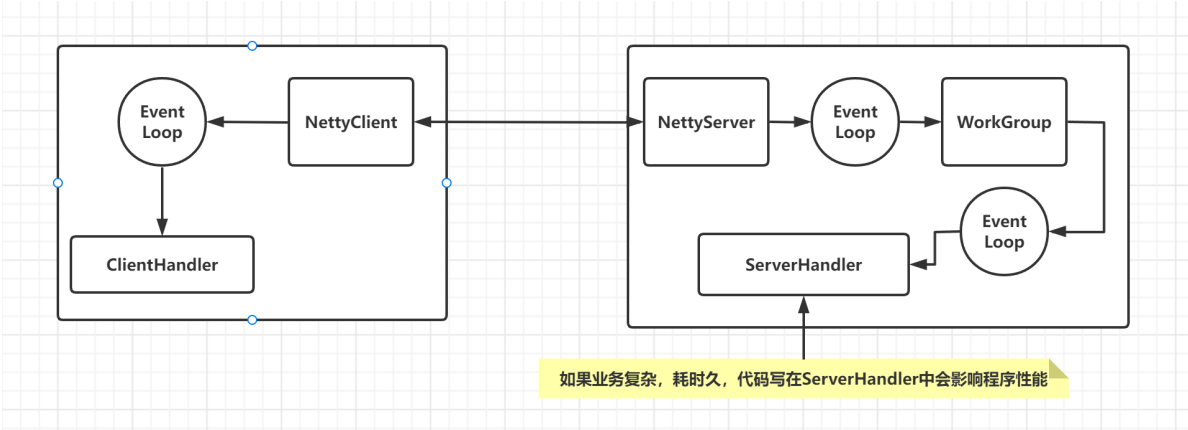
```
package com.hero.disruptor.multi;

public class TestMultiDisruptor {

    public static void main(String[] args) throws InterruptedException {
        //1.创建RingBuffer，Disruptor包含RingBuffer
        RingBuffer<Order> ringBuffer = RingBuffer.create(
                ProducerType.MULTI, //多生产者
                new EventFactory<Order>() {
                    @Override
                    public Order newInstance() {
                        return new Order();
                    }
                },
                1024 * 1024,
                new YieldingWaitStrategy());

        //2.创建ringBuffer屏障
        SequenceBarrier sequenceBarrier = ringBuffer.newBarrier();

```

```
        //3.创建多个消费者数组
        ConsumerHandler[] consumers = new ConsumerHandler[10];

        for (int i = 0; i < consumers.length; i++) {
            consumers[i] = new ConsumerHandler("C" + i);
        }

        //4.构建多消费者工作池
        WorkerPool<Order> workerPool = new WorkerPool<Order>(
                ringBuffer,
                sequenceBarrier,
                new EventExceptionHandler(),
                consumers);

        //5.设置多个消费者的sequence序号，用于单独统计消费者的消费进度。消费进度让RingBuffer知道
        ringBuffer.addGatingSequences(workerPool.getWorkerSequences());
        //6.启动workPool
        workerPool.start(Executors.newFixedThreadPool(5)); //在实际开发，自定义线程池。

        //要生产100生产者，每个生产者发送100个数据,投递10000
        final CountDownLatch latch = new CountDownLatch(1);

        //设置100个生产者向ringbuffer中去投递数据
        for (int i = 0; i < 100; i++) {
            final Producer producer = new Producer(ringBuffer);

            new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        //每次一个生产者创建后就处理等待。先创建100个生产者，创建完100个生产者后再去发送数据。
                        latch.await();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }

                    //每个生产者投递100个数据
                    for (int j = 0; j < 100; j++) {
                        producer.sendData(UUID.randomUUID().toString());
                    }
                }
            }).start();
        }
        //把所有线程都创建完
        TimeUnit.SECONDS.sleep(2);
        //唤醒，开始运行100个线程
        latch.countDown();
        //休眠10s，让生产者将100次循环走完
        TimeUnit.SECONDS.sleep(10);

        System.err.println("任务总数:" + consumers[0].getCount());

```

```
    }

    static class EventExceptionHandler implements ExceptionHandler<Order> {
        //消费时出现异常
        @Override
        public void handleEventException(Throwable throwable, long l, Order
order) {
        }
        //启动时出现异常
        @Override
        public void handleOnStartException(Throwable throwable) {
        }
        //停止时出现异常
        @Override
        public void handleOnShutdownException(Throwable throwable) {
        }
    }
}

```

6.4 案例06：使用Disruptor提升Netty应用性能

6.4.1 构建Netty网络模型



1）构建基础网络应用环境

- disruptor-netty-com是通用包
- disruptor-netty-client是客户端
- disruptor-netty-server是服务端

2）TranslatorData

传输的数据对象

```
package com.hero.entity;

public class TranslatorData implements Serializable {

   private static final long serialVersionUID = 8763561286199081881L;
```

```
6
7     private String id;
8     private String name;
9     private String message;    //传输消息体内容
10
11    public String getId() {
12       return id;
13    }
14    public void setId(String id) {
15       this.id = id;
16    }
17    public String getName() {
18       return name;
19    }
20    public void setName(String name) {
21       this.name = name;
22    }
23    public String getMessage() {
24       return message;
25    }
26    public void setMessage(String message) {
27       this.message = message;
28    }
29
30 }
31
```

3）NettyServer

```
1  package com.hero.server;
2
3  public class NettyServer {
4
5     public NettyServer() {
6        //1. 创建两个工作线程组：一个用于接受网络请求的线程组. 另一个用于实际处理业务的线
   程组
7        EventLoopGroup bossGroup = new NioEventLoopGroup();
8        EventLoopGroup workGroup = new NioEventLoopGroup();
9        //2 辅助类
10       ServerBootstrap serverBootstrap = new ServerBootstrap();
11       try {
12
13          serverBootstrap.group(bossGroup, workGroup)
14          .channel(NioServerSocketChannel.class)
15          .option(ChannelOption.SO_BACKLOG, 1024)
16          //表示缓存区动态调配（自适应）
17          .option(ChannelOption.RCVBUF_ALLOCATOR,
   AdaptiveRecvByteBufAllocator.DEFAULT)
18          //缓存区 池化操作
19          .option(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT)
20          .handler(new LoggingHandler(LogLevel.INFO))
21          .childHandler(new ChannelInitializer<SocketChannel>() {
22             @Override
23             protected void initChannel(SocketChannel sc) throws Exception {
```

```
sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingDecoder());

sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingEncoder());
                sc.pipeline().addLast(new ServerHandler());
            }
        });
        //绑定端口，同步等等请求连接
        ChannelFuture cf = serverBootstrap.bind(8765).sync();
        System.err.println("Server Startup...");
        cf.channel().closeFuture().sync();

    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        //优雅停机
        bossGroup.shutdownGracefully();
        workGroup.shutdownGracefully();
        System.err.println("Sever ShutDown...");
    }
  }

}

```

4）ServerHandler

```
package com.hero.server;

public class ServerHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
       TranslatorData request = (TranslatorData)msg;
       System.err.println("Sever端: id= " + request.getId() + ", name= " +
request.getName() + ", message= " + request.getMessage());
       //数据库持久化操作 IO读写 ---> 交给一个线程池 去异步的调用执行
       TranslatorData response = new TranslatorData();
       response.setId("resp: " + request.getId());
       response.setName("resp: " + request.getName());
       response.setMessage("resp: " + request.getMessage());
       //写出response响应信息:
       ctx.writeAndFlush(response);
     }
}

```

5）NettyClient

```
package com.hero.client;

public class NettyClient {

```

```
    public static final String HOST = "127.0.0.1";
    public static final int PORT = 8765;

    //扩展 完善 池化: ConcurrentHashMap<KEY -> String, Value -> Channel>
    private Channel channel;

    //1. 创建工作线程组: 用于实际处理业务的线程组
    private EventLoopGroup workGroup = new NioEventLoopGroup();

    private ChannelFuture cf;

    public NettyClient() {
        this.connect(HOST, PORT);
    }

    private void connect(String host, int port) {
        //2 辅助类(注意Client 和 Server 不一样)
        Bootstrap bootstrap = new Bootstrap();
        try {
            //绑定一个线程组
            bootstrap.group(workGroup)
            .channel(NioSocketChannel.class)
            //表示缓存区动态调配（自适应）
            .option(ChannelOption.RCVBUF_ALLOCATOR,
AdaptiveRecvByteBufAllocator.DEFAULT)
            //缓存区 池化操作
            .option(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT)
            .handler(new LoggingHandler(LogLevel.INFO))
            .handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel sc) throws Exception {
                    //网络传递对象，客户端和服务端都要做编码解码操作

sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingDecoder());

sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingEncoder());
                    sc.pipeline().addLast(new ClientHandler());
                }
            });
            //绑定端口，同步等等请求连接
            this.cf = bootstrap.connect(host, port).sync();
            System.err.println("Client connected...");

            //接下来就进行数据的发送，但是首先我们要获取channel:
            this.channel = cf.channel();

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    //发送数据的方法，提供给外部使用
    public void sendData(){

        for(int i =0; i <10; i++){
            TranslatorData request = new TranslatorData();
```

```
            request.setId("" + i);
            request.setName("请求消息名称 " + i);
            request.setMessage("请求消息内容 " + i);
            this.channel.writeAndFlush(request);
        }
    }

    public void close() throws Exception {
        cf.channel().closeFuture().sync();
        workGroup.shutdownGracefully();//优雅停机
        System.err.println("Sever ShutDown...");
    }

}

```

6）ClientHandler

```
package com.hero.client;

public class ClientHandler extends ChannelInboundHandlerAdapter {
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        try {
            TranslatorData response = (TranslatorData)msg;
            System.err.println("Client端: id= " + response.getId()
                    + ", name= " + response.getName()
                    + ", message= " + response.getMessage());
        } finally {
            //一定要注意 用完了缓存 要进行释放
            ReferenceCountUtil.release(msg);
        }
    }
}
```

7）启动类

服务端NettyServerApplication

```
package com.hero;

@SpringBootApplication
public class NettyServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(NettyServerApplication.class, args);
        new NettyServer();
    }
}

```

客户端NettyClientApplication

```java
package com.hero;

@SpringBootApplication
public class NettyClientApplication {
   public static void main(String[] args) {
   SpringApplication.run(NettyClientApplication.class, args);
      //建立连接 并发送消息
      new NettyClient().sendData();
   }
}

```

测试：

启动server

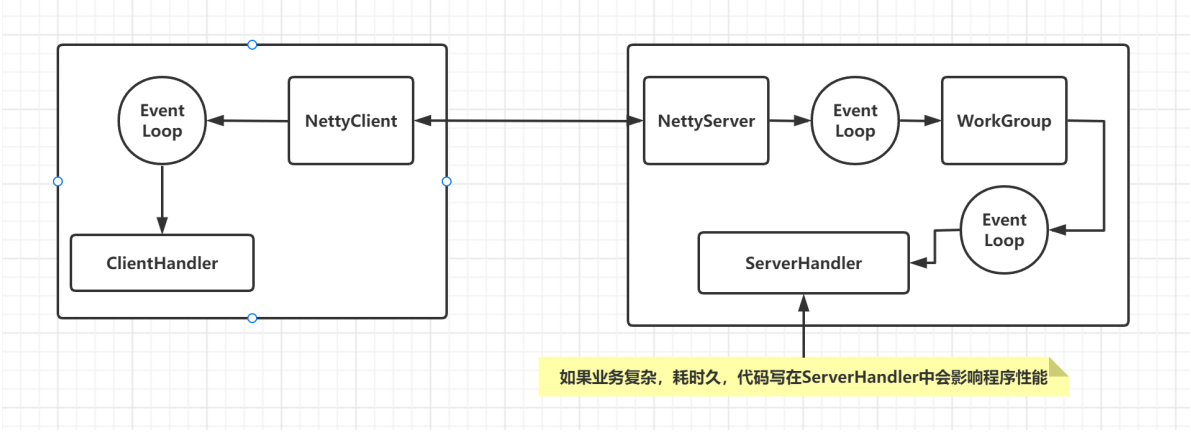
```
Console  Endpoints
[redacted] INFO 5840 --- [ntLoopGroup-2-1] io.net
[redacted] INFO 5840 --- [ntLoopGroup-2-1] io.net
Server Startup...
```

启动client，发送数据。

```
Console  Endpoints
[redacted] INFO 11396 --- [           main] com.hero.NettyClientApplication
[redacted] INFO 11396 --- [           main] com.hero.NettyClientApplication
[redacted] INFO 11396 --- [           main] s.c.a.AnnotationConfigApplicationCont
[redacted] INFO 11396 --- [           main] o.s.j.e.a.AnnotationMBeanExporter
[redacted] INFO 11396 --- [           main] com.hero.NettyClientApplication
Client connected...
Client端: id= resp: 0, name= resp: 请求消息名称 0, message= resp: 请求消息内容 0
Client端: id= resp: 1, name= resp: 请求消息名称 1, message= resp: 请求消息内容 1
Client端: id= resp: 2, name= resp: 请求消息名称 2, message= resp: 请求消息内容 2
Client端: id= resp: 3, name= resp: 请求消息名称 3, message= resp: 请求消息内容 3
Client端: id= resp: 4, name= resp: 请求消息名称 4, message= resp: 请求消息内容 4
Client端: id= resp: 5, name= resp: 请求消息名称 5, message= resp: 请求消息内容 5
Client端: id= resp: 6, name= resp: 请求消息名称 6, message= resp: 请求消息内容 6
Client端: id= resp: 7, name= resp: 请求消息名称 7, message= resp: 请求消息内容 7
Client端: id= resp: 8, name= resp: 请求消息名称 8, message= resp: 请求消息内容 8
Client端: id= resp: 9, name= resp: 请求消息名称 9, message= resp: 请求消息内容 9
```

到此为止，我们已经构建好了网络模型，可以接收和发送消息了。

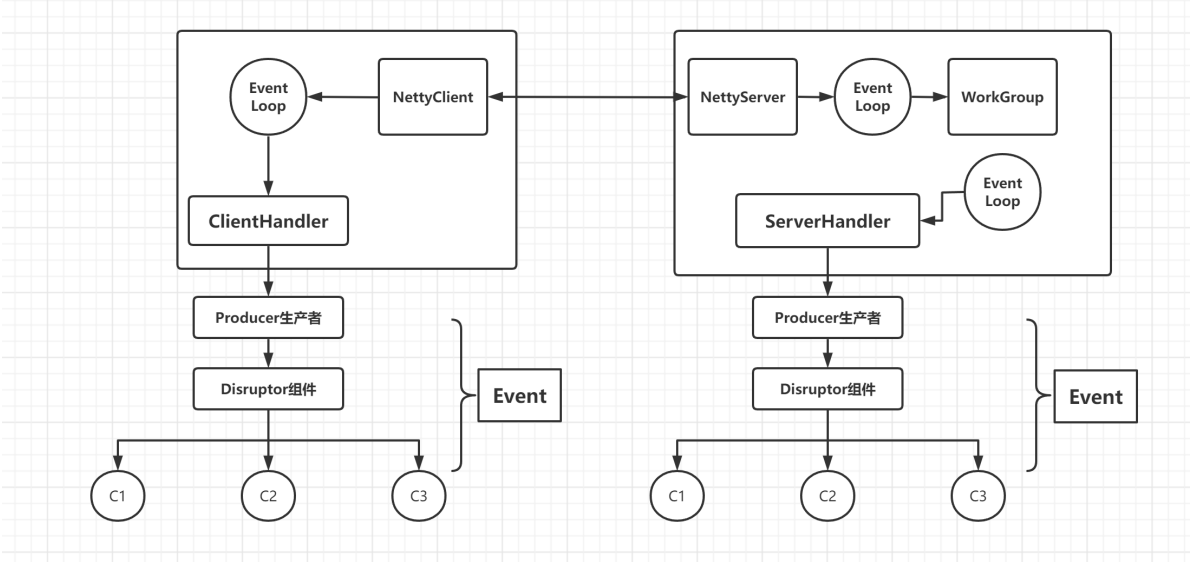
6.4.2 整合Disruptor



在使用Netty进行接收处理数据时，尽量不要在工作线程上编写自己的代理逻辑，会降低netty性能。可以利用异步机制，如使用线程池异步处理，如果使用线程池就意味使用阻塞对列，可以替换为Disruptor提高性能。

加入disruptor提升性能：

Event是客户端发到服务端的数据，serverHandler获取到Event后，不在serverHandler中对数据做处理，将Event通过生产者交给Disruptor组件。消费者c1、c2、c3通过负载均衡去消费投递过来的数据。服务端最终要返回一个响应数据给客户端。客户端这边也不是在ClientHandler中处理数据，也要构建一个生产消费者模型，有多个线程去处理。



1）TranslatorDataWapper

传输的对象包装类

```
package com.hero.entity;

public class TranslatorDataWapper {

    private TranslatorData data;

    private ChannelHandlerContext ctx;

    public TranslatorData getData() {
        return data;
    }

    public void setData(TranslatorData data) {
        this.data = data;
    }

    public ChannelHandlerContext getCtx() {
        return ctx;
    }

    public void setCtx(ChannelHandlerContext ctx) {
        this.ctx = ctx;
    }
}
```

2）MessageProducer

生产者

```
package com.hero.disruptor;

public class MessageProducer {

    private String producerId;

    private RingBuffer<TranslatorDataWapper> ringBuffer;

    public MessageProducer(String producerId,
RingBuffer<TranslatorDataWapper> ringBuffer) {
        this.producerId = producerId;
        this.ringBuffer = ringBuffer;
    }

    public void onData(TranslatorData data, ChannelHandlerContext ctx) {
        long sequence = ringBuffer.next();
        try {
            TranslatorDataWapper wapper = ringBuffer.get(sequence);
            wapper.setData(data);
            wapper.setCtx(ctx);
        } finally {
            ringBuffer.publish(sequence);
        }
    }

}

```

3）MessageConsumer

消费者

```
package com.hero.disruptor;

public abstract class MessageConsumer implements
WorkHandler<TranslatorDataWapper> {

    protected String consumerId;

    public MessageConsumer(String consumerId) {
        this.consumerId = consumerId;
    }

    public String getConsumerId() {
        return consumerId;
    }

```

```
    public void setConsumerId(String consumerId) {
        this.consumerId = consumerId;
    }
}
```

4）RingBufferWorkerPoolFactory

创建连接池工厂类

```
package com.hero.disruptor;

public class RingBufferWorkerPoolFactory {
    //单例
    private static class SingletonHolder {
        static final RingBufferWorkerPoolFactory instance = new RingBufferWorkerPoolFactory();
    }

    private RingBufferWorkerPoolFactory(){

    }

    public static RingBufferWorkerPoolFactory getInstance() {
        return SingletonHolder.instance;
    }
    //需要生产者池和消费者池管理生产和消费者。
    private static Map<String, MessageProducer> producers = new ConcurrentHashMap<String, MessageProducer>();

    private static Map<String, MessageConsumer> consumers = new ConcurrentHashMap<String, MessageConsumer>();

    private RingBuffer<TranslatorDataWapper> ringBuffer;

    private SequenceBarrier sequenceBarrier;

    private WorkerPool<TranslatorDataWapper> workerPool;
    //初始化ProducerType生产者类型，是多生产还是单生产。MessageConsumer[]多消费者
    public void initAndStart(ProducerType type, int bufferSize, WaitStrategy waitStrategy, MessageConsumer[] messageConsumers) {
        //1. 构建ringBuffer对象
        this.ringBuffer = RingBuffer.create(type,
                new EventFactory<TranslatorDataWapper>() {
                    public TranslatorDataWapper newInstance() {
                        return new TranslatorDataWapper();
                    }
                },
                bufferSize,
                waitStrategy);
        //2.设置序号栅栏
        this.sequenceBarrier = this.ringBuffer.newBarrier();
        //3.设置工作池
```

```
        this.workerPool = new WorkerPool<TranslatorDataWapper>
(this.ringBuffer,
                this.sequenceBarrier,
                new EventExceptionHandler(), messageConsumers);
        //4 把所构建的消费者置入池中
        for(MessageConsumer mc : messageConsumers){
            this.consumers.put(mc.getConsumerId(), mc);
        }
        //5 添加我们的sequences

 this.ringBuffer.addGatingSequences(this.workerPool.getWorkerSequences());
        //6 启动我们的工作池

this.workerPool.start(Executors.newFixedThreadPool(Runtime.getRuntime().avai
lableProcessors()/2));
    }

    public MessageProducer getMessageProducer(String producerId){
        //池里有直接获取生产者
        MessageProducer messageProducer = this.producers.get(producerId);
        if(null == messageProducer) {
            messageProducer = new MessageProducer(producerId,
this.ringBuffer);
            this.producers.put(producerId, messageProducer);
        }
        return messageProducer;
    }


    /**
     * 异常静态类
     */
    static class EventExceptionHandler implements
ExceptionHandler<TranslatorDataWapper> {
        public void handleEventException(Throwable ex, long sequence,
TranslatorDataWapper event) {
        }

        public void handleOnStartException(Throwable ex) {
        }

        public void handleOnShutdownException(Throwable ex) {
        }
    }
}
```

6.4.3 百万级连接接入

1）修改serverHandler

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
    TranslatorData request = (TranslatorData)msg;
    //自己的应用服务应该有一个ID生成规则
    String producerId = "code:sessionId:001";
    MessageProducer messageProducer =
RingBufferWorkerPoolFactory.getInstance().getMessageProducer(producerId);
    messageProducer.onData(request, ctx);
}

```

2）MessageConsumerImpl4Server

服务器端消费者：用来处理客户端发送来数据的逻辑

```
package com.hero.server;

public class MessageConsumerImpl4Server extends MessageConsumer {

   public MessageConsumerImpl4Server(String consumerId) {
      super(consumerId);
   }

   public void onEvent(TranslatorDataWapper event) throws Exception {
      TranslatorData request = event.getData();
      ChannelHandlerContext ctx = event.getCtx();
      //1.业务处理逻辑:
       System.err.println("Sever端: id= " + request.getId()
      + ", name= " + request.getName()
      + ", message= " + request.getMessage());

       //2.回送响应信息:
       TranslatorData response = new TranslatorData();
       response.setId("resp: " + request.getId());
       response.setName("resp: " + request.getName());
       response.setMessage("resp: " + request.getMessage());
       //写出response响应信息:
       ctx.writeAndFlush(response);
   }

}

```

3）修改clientHandler

```
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
   TranslatorData response = (TranslatorData)msg;
   //i消费者和生产者共用一个池，所以id不可以冲突,以后可以随机生成id(机器码:sessionId:标识)
   String producerId = "code:seesionId:002";
   MessageProducer messageProducer = RingBufferWorkerPoolFactory.getInstance().getMessageProducer(producerId);
   messageProducer.onData(response, ctx);
  }

```

4）MessageConsumerImpl4Client

客户端处理服务端返回数据

```
package com.hero.client;

public class MessageConsumerImpl4Client extends MessageConsumer {

   public MessageConsumerImpl4Client(String consumerId) {
      super(consumerId);
   }

   public void onEvent(TranslatorDataWapper event) throws Exception {
      TranslatorData response = event.getData();
      ChannelHandlerContext ctx = event.getCtx();
      //业务逻辑处理:
      try {
          System.err.println("Client端: id= " + response.getId()
         + ", name= " + response.getName()
         + ", message= " + response.getMessage());
      } finally {
         ReferenceCountUtil.release(response);
      }
   }

}

```

5）启动类

服务端

```
package com.hero;

@SpringBootApplication
public class NettyServerApplication {

```

```
5
6     public static void main(String[] args) {
7        SpringApplication.run(NettyServerApplication.class, args);
8        MessageConsumer[] conusmers = new MessageConsumer[4];
9        for(int i =0; i < conusmers.length; i++) {
10          MessageConsumer messageConsumer = new
MessageConsumerImpl4Server("code:serverId:" + i);
11          conusmers[i] = messageConsumer;
12       }
13
 RingBufferWorkerPoolFactory.getInstance().initAndStart(ProducerType.MULTI,
14             1024*1024,
15             new BlockingWaitStrategy(),
16             conusmers);
17
18       new NettyServer();
19    }
20 }
21
```

客户端

```
1  package com.hero;
2
3
4  @SpringBootApplication
5  public class NettyClientApplication {
6
7     public static void main(String[] args) {
8     SpringApplication.run(NettyClientApplication.class, args);
9
10          MessageConsumer[] conusmers = new MessageConsumer[4];
11       for(int i =0; i < conusmers.length; i++) {
12          MessageConsumer messageConsumer = new
MessageConsumerImpl4Client("code:clientId:" + i);
13          conusmers[i] = messageConsumer;
14       }
15
 RingBufferWorkerPoolFactory.getInstance().initAndStart(ProducerType.MULTI,
16             1024*1024,
17             //new YieldingWaitStrategy(),
18             new BlockingWaitStrategy(),
19             conusmers);
20       //建立连接 并发送消息
21       new NettyClient().sendData();
22    }
23 }
24
```

6）测试

```
//发送数据的方法，提供给外部使用
public void sendData(){

    for(int i =0; i <6000000; i++){
        TranslatorData request = new TranslatorData();
        request.setId("" + i);
        request.setName("请求消息名称 " + i);
        request.setMessage("请求消息内容 " + i);
        this.channel.writeAndFlush(request);
    }
}
```

可以看到百万级别的连接已经建立