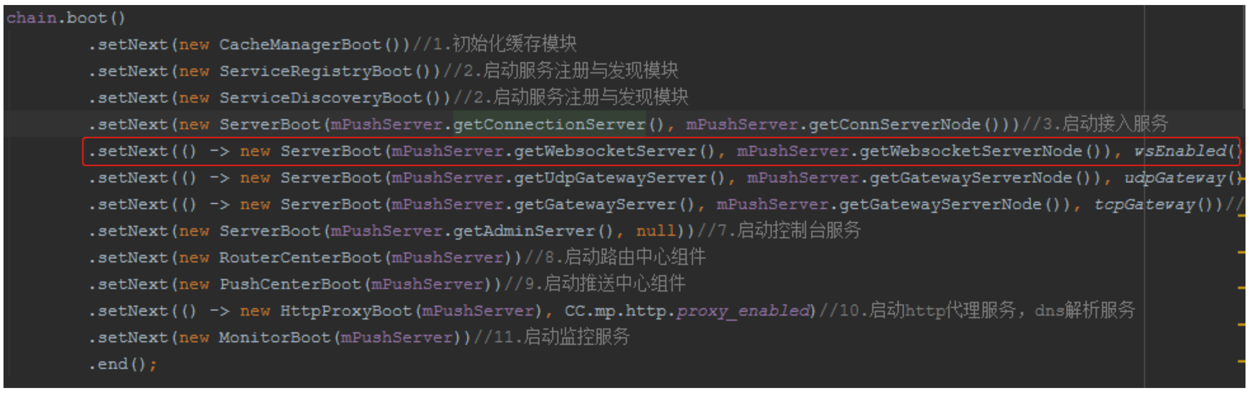
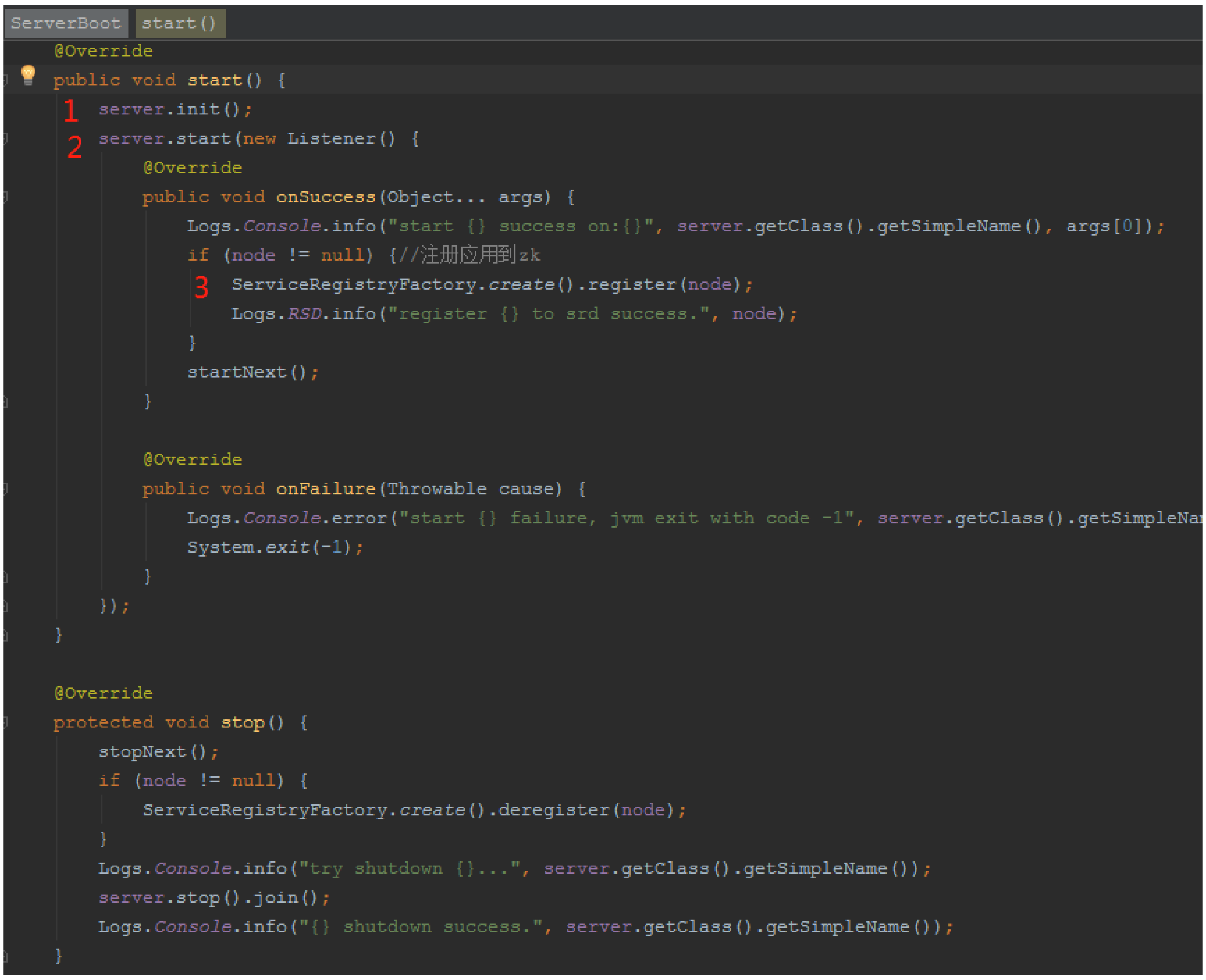
初始化websocket服务



```java
chain.boot()
        .setNext(new CacheManagerBoot())//1.初始化缓存模块
        .setNext(new ServiceRegistryBoot())//2.启动服务注册与发现模块
        .setNext(new ServiceDiscoveryBoot())//2.启动服务注册与发现模块
        .setNext(new ServerBoot(mPushServer.getConnectionServer(), mPushServer.getConnServerNode()))//3.启动接入服务
        .setNext(() -> new ServerBoot(mPushServer.getWebsocketServer(), mPushServer.getWebsocketServerNode()), wsEnabled()
        .setNext(() -> new ServerBoot(mPushServer.getUdpGatewayServer(), mPushServer.getGatewayServerNode()), udpGateway()
        .setNext(() -> new ServerBoot(mPushServer.getGatewayServer(), mPushServer.getGatewayServerNode()), tcpGateway())//
        .setNext(new ServerBoot(mPushServer.getAdminServer(), null))//7.启动控制台服务
        .setNext(new RouterCenterBoot(mPushServer))//8.启动路由中心组件
        .setNext(new PushCenterBoot(mPushServer))//9.启动推送中心组件
        .setNext(() -> new HttpProxyBoot(mPushServer), CC.mp.http.proxy_enabled)//10.启动http代理服务，dns解析服务
        .setNext(new MonitorBoot(mPushServer))//11.启动监控服务
        .end();
```

服务启动

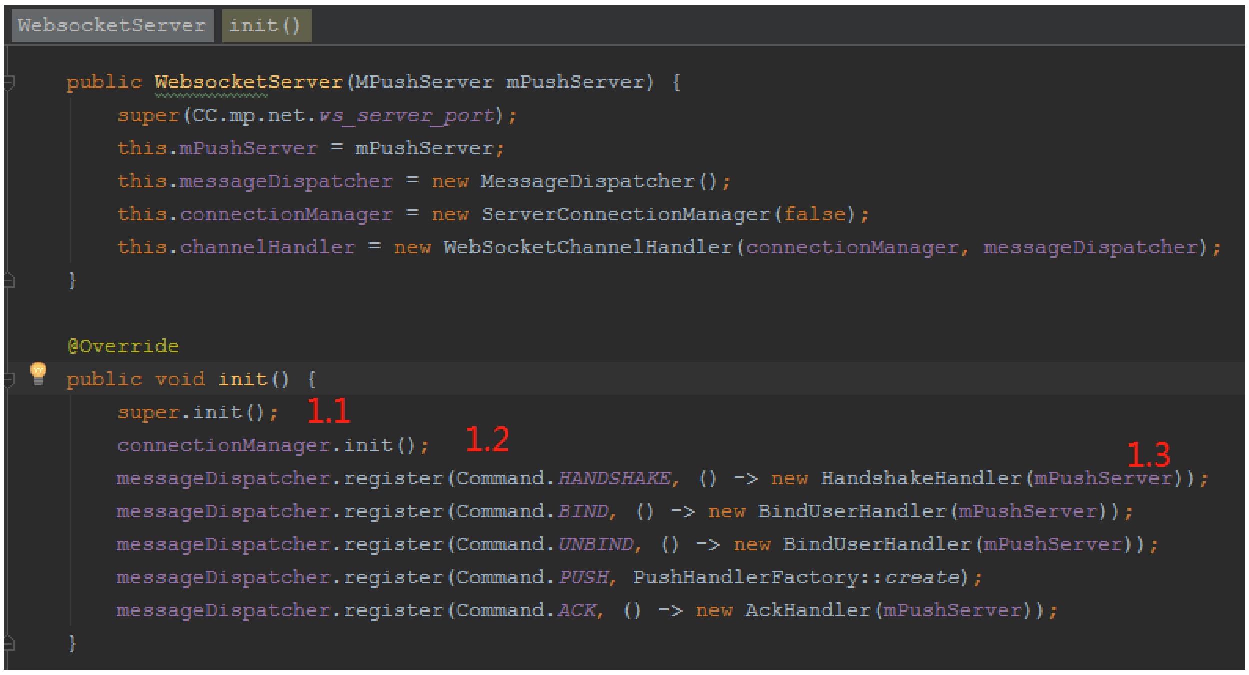


```java
    @Override
    public void start() {
1       server.init();
2       server.start(new Listener() {
            @Override
            public void onSuccess(Object... args) {
                Logs.Console.info("start {} success on:{}", server.getClass().getSimpleName(), args[0]);
                if (node != null) {//注册应用到zk
3                   ServiceRegistryFactory.create().register(node);
                    Logs.RSD.info("register {} to srd success.", node);
                }
                startNext();
            }

            @Override
            public void onFailure(Throwable cause) {
                Logs.Console.error("start {} failure, jvm exit with code -1", server.getClass().getSimpleNa
                System.exit(-1);
            }
        });
    }

    @Override
    protected void stop() {
        stopNext();
        if (node != null) {
            ServiceRegistryFactory.create().deregister(node);
        }
        Logs.Console.info("try shutdown {}...", server.getClass().getSimpleName());
        server.stop().join();
        Logs.Console.info("{} shutdown success.", server.getClass().getSimpleName());
    }
```

1、调用WebsocketServer#init()

2、调用WebsocketServer的父类NettyTCPServer#start()

3、将WS节点信息注册到Zookeeper



```java
public WebsocketServer(MPushServer mPushServer) {
    super(CC.mp.net.ws_server_port);
    this.mPushServer = mPushServer;
    this.messageDispatcher = new MessageDispatcher();
    this.connectionManager = new ServerConnectionManager(false);
    this.channelHandler = new WebSocketChannelHandler(connectionManager, messageDispatcher);
}

@Override
public void init() {
    super.init();
    connectionManager.init();
    messageDispatcher.register(Command.HANDSHAKE, () -> new HandshakeHandler(mPushServer));
    messageDispatcher.register(Command.BIND, () -> new BindUserHandler(mPushServer));
    messageDispatcher.register(Command.UNBIND, () -> new BindUserHandler(mPushServer));
    messageDispatcher.register(Command.PUSH, PushHandlerFactory::create);
    messageDispatcher.register(Command.ACK, () -> new AckHandler(mPushServer));
}
```

1.1 调用NettyTCPServer#init()

主要是利用AtomicReference<State>的cas判断netty服务是不是启动，如果已经启动则抛出ServiceException异常；

1.2 调用ServerConnectionManager#init()

这里主要是正常连接管理，该方法里面不做任何事情；

1.3 注册各种消息的处理类

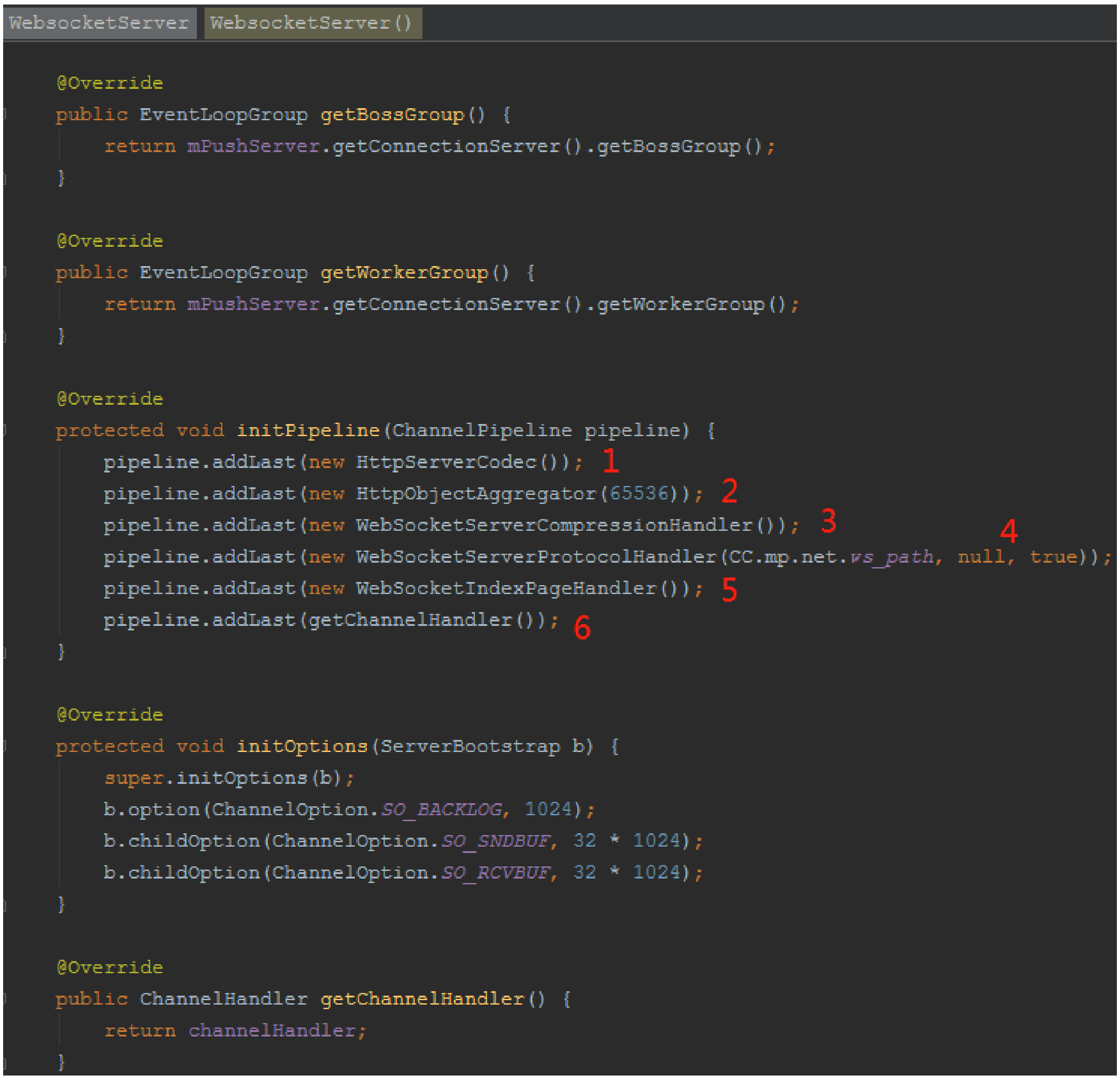
2 调用NettyTCPServer#start()方法，创建ServerBootstrap启动Netty长连接服务；

NettyTCPServer创建netty ServerBootstrap服务时，会调用其子类ConnectionServer中方法：

initPipeline()：重新创建并加入HTTP/websocket编解码相关处理类，不使用父类中的pipeline

initOptions(): 设置发送、接收BUF缓冲区大小

getChannelHandler(): 设置netty 事件处理类WebSocketChannelHandler，处理建连、消息、断连、异常事件；



```java
    @Override
    public EventLoopGroup getBossGroup() {
        return mPushServer.getConnectionServer().getBossGroup();
    }

    @Override
    public EventLoopGroup getWorkerGroup() {
        return mPushServer.getConnectionServer().getWorkerGroup();
    }

    @Override
    protected void initPipeline(ChannelPipeline pipeline) {
        pipeline.addLast(new HttpServerCodec()); 1
        pipeline.addLast(new HttpObjectAggregator(65536)); 2
        pipeline.addLast(new WebSocketServerCompressionHandler()); 3            4
        pipeline.addLast(new WebSocketServerProtocolHandler(CC.mp.net.ws_path, null, true));
        pipeline.addLast(new WebSocketIndexPageHandler()); 5
        pipeline.addLast(getChannelHandler()); 6
    }

    @Override
    protected void initOptions(ServerBootstrap b) {
        super.initOptions(b);
        b.option(ChannelOption.SO_BACKLOG, 1024);
        b.childOption(ChannelOption.SO_SNDBUF, 32 * 1024);
        b.childOption(ChannelOption.SO_RCVBUF, 32 * 1024);
    }

    @Override
    public ChannelHandler getChannelHandler() {
        return channelHandler;
    }
```

HttpServerCodec：双端HTTP编解码处理类，包含decode和encode

HttpObjectAggregator：Http消息聚合，把多个HTTP请求中的数据组装成一个

WebSocketServerCompressionHandler：WebSocket数据压缩

WebSocketServerProtocolHandler：指定web访问路径，处理除TextWebSocketFrame以外的消息事件

WebSocketIndexPageHandler：首页处理index.html

WebSocketChannelHandler：处理TextWebSocketFrame消息事件