Netty

> Netty是Java领域有名的开源网络库，特点是高性能和高扩展性，因此很多流行的框架都是基于它来构建的，比如我们熟知的Dubbo、Rocketmq、Hadoop等，针对高性能RPC，一般都是基于Netty来构建，比如sock-bolt。

1 高级功能

1.1 异步任务调度

1.1.1 TaskQueue

任务队列 TaskQueue 的任务 Task 应用场景 :

① **自定义任务 :** 自己开发的任务 , 然后将该任务提交到任务队列中 ;

② **自定义定时任务 :** 自己开发的任务 , 然后将该任务提交到任务队列中 , 同时可以指定任务的执行时间 ;

③ **其它线程调度任务**

1.1.2 Handler同步异步

用户自定义的 Handler 处理器 , 该处理器继承了 ChannelInboundHandlerAdapter 类 , 在重写的 public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception 方法中 , 执行的业务逻辑要注意以下两点 :

- **同步操作 :** 如果在该业务逻辑中只执行一个短时间的操作 , 那么可以直接执行 ;
- **异步操作 :** 如果在该业务逻辑中执行访问数据库 , 访问网络 , 读写本地文件 , 执行一系列复杂计算等耗时操作 , 肯定不能在该方法中处理 , 这样会阻塞整个线程 ; 正确的做法是将耗时的操作放入任务队列 TaskQueue , 异步执行 ;

1.1.3 自定义任务

多任务执行 : 如果用户连续向任务队列中放入了多个任务 , NioEventLoop 会按照顺序先后执行这些任务 , 注意任务队列中的任务 **是先后执行 , 不是同时执行** ;

顺序执行任务 (不是并发) : 任务队列任务执行机制是顺序执行的 ; 先执行第一个 , 执行完毕后 , 从任务队列中获取第二个任务 , 执行完毕之后 , 依次从任务队列中取出任务执行 , 前一个任务执行完毕后 , 才从任务队列中取出下一个任务执行 ;

1.1.4 自定义定时任务

用户自定义定时任务 与 用户自定义任务流程基本类似:

① **调度方法 :**

- **定时异步任务**使用 schedule 方法进行调度
- 普通异步任务使用 execute 方法进行调度

② **任务队列 :**

- **定时异步任务**提交到 ScheduleTaskQueue 任务队列中 ;
- 普通异步任务提交到 TaskQueue 任务队列中 ;

1.1.5 向其它线程调度任务

在服务器中使用 Map 集合管理该 Channel 通道 , 需要时根据用户标识信息 , 获取该通道 , 向该客户端通道对应的 NioEventLoop 线程中调度任务 ;

```
public class GroupChatServerHandler extends
SimpleChannelInboundHandler<String> {
    private static ChannelGroup channelGroup = new
DefaultChannelGroup(GlobalEventExecutor.INSTANCE);

    ...
     @Override
    public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
```

```
        Channel channel = ctx.channel();
        channelGroup.writeAndFlush("[客户端]" + channel.remoteAddress() + " 加入
聊天" + sdf.format(new java.util.Date()) + " \n");
        channelGroup.add(channel);
    }

    .............
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, String msg) throws
Exception {

        //获取到当前channel
        Channel channel = ctx.channel();
        //这时我们遍历channelGroup, 根据不同的情况，回送不同的消息

        channelGroup.forEach(ch -> {
            if (channel != ch) { //不是当前的channel,转发消息
            ch.writeAndFlush("[客户]" + channel.remoteAddress() + " 发送了消息" +
msg);
            }
        });
    }

  ....
```

1.2 异步线程池

在 Netty 中做耗时的，不可预料的操作，比如数据库，网络请求，会严重影响 Netty 对 Socket 的处理速度。而解决方法就是将耗时任务添加到异步线程池中。但就添加线程池这步操作来讲，可以有2种方式，而且这2种方式实现的区别也蛮大的。

第一种方式—**handler 中加入线程池**

第二种方式—**Context 中添加线程池**

1.2.1 handler 中加入线程池

```
    private static final EventExecutorGroup group = new
DefaultEventExecutorGroup(8);
```

源码

```
abstract class AbstractChannelHandlerContext implements ChannelHandlerContext,
ResourceLeakHint {
  ....
    private void write(Object msg, boolean flush, ChannelPromise promise) {
       ...
```

```
        EventExecutor executor = next.executor();
        if (executor.inEventLoop()) {
            if (flush) {
                next.invokeWriteAndFlush(m, promise);
            } else {
                next.invokeWrite(m, promise);
            }
        } else {
            final WriteTask task = WriteTask.newInstance(next, m, promise,
flush);
            if (!safeExecute(executor, task, promise, m, !flush)) {
                // We failed to submit the WriteTask. We need to cancel it so
we decrement the pending bytes
                // and put it back in the Recycler for re-use later.
                //
                // See https://github.com/netty/netty/issues/8343.
                task.cancel();
            }
        }
    }
```

1.2.2 Context 中添加线程池

在调用addLast方法添加线程池后，handler将优先使用这个线程池，如果不添加，将使用IO线程。

```
    private static final EventExecutorGroup group =new
DefaultEventExecutorGroup(2);

                            pipeline.addLast(group,new NettyServerHandler());
```

源码

```
abstract class AbstractChannelHandlerContext implements ChannelHandlerContext,
ResourceLeakHint {
  ....

static void invokeChannelRead(final AbstractChannelHandlerContext next, Object
msg) {
        final Object m = next.pipeline.touch(ObjectUtil.checkNotNull(msg,
"msg"), next);
        EventExecutor executor = next.executor();
        if (executor.inEventLoop()) {
            next.invokeChannelRead(m);
        } else {
            executor.execute(new Runnable() {
                @Override
```

```
                public void run() {
                    next.invokeChannelRead(m);
                }
            });
        }
    }
```

两种方式比较

1、第一种方式在handler中添加异步，可能更加的自由，比如如果需要访问数据库，那我就异步，如果不需要就不异步，异步会拖长接口响应时间。因为需要将任务放进task中，如果IO时间很短，task很多，可能一个循环下来，都没时间执行整个task，导致响应时间不达标。

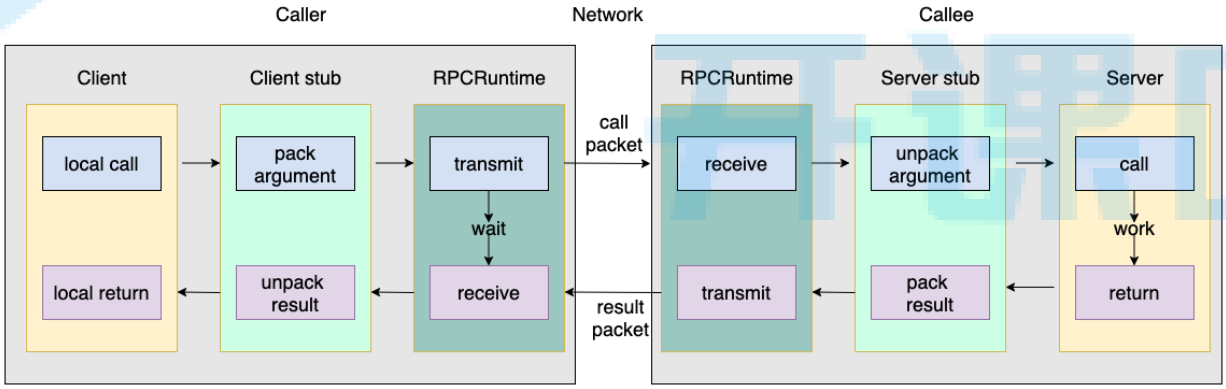
2、第二中方式是Netty标准方式即加入到队列，但是这么做会将整个handler都交给业务线程池，不论耗时不耗时都加入队列，不够灵活。

1.3 Netty实现Dubbo RPC

1.3.1 RPC 基本介绍

RPC（Remote Procedure Call）—远程过程调用，是一个计算机通信协议。该协议允许运行于一台计算机的程序调用另一台计算机的子程序，而程序员无需额外地为这个交互作用编程

两个或多个应用程序都分布在不同的服务器上，它们之间的调用都像是本地方法调用一样



PRC 调用流程说明

1. 服务消费方（client）以本地调用方式调用服务
2. `client stub` 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` 根据解码结果调用本地的服务
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

总结：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

1.3.2 实现RPC

设计说明

1. 创建一个接口，定义抽象方法。用于消费者和提供者之间的约定。
2. 创建一个提供者，该类需要监听消费者的请求，并按照约定返回数据。
3. 创建一个消费者，该类需要透明的调用自己不存在的方法，内部需要使用netty请求提供者返回数据

代码实现

1、定义统一的接口

```
package com.kkb.demo.netty.example.dubborpc.api;

public interface CityService {
    ....
}
```

2、定义协议标识

```
package com.kkb.demo.netty.example.dubborpc.api;

public class Constant {
    ......
}
```

3、提供方:接口实现

```
package com.kkb.demo.netty.example.dubborpc.provider;

public class CityServiceImpl implements CityService {

    ....
}
```

4、Netty的服务端的处理handler

```
package com.kkb.demo.netty.example.dubborpc.netty;


public class NettyServerHandler extends ChannelInboundHandlerAdapter {
    ....
}
```

5、Netty服务端启动

```
package com.kkb.demo.netty.example.dubborpc.netty;


public class NettyServer {
    ...
}
```

6、服务提供者启动类

```
package com.kkb.demo.netty.example.dubborpc.provider;


import com.kkb.demo.netty.example.dubborpc.netty.NettyServer;

public class ServerBootstrap {
    public static void main(String[] args) {

        //代码代填..
        NettyServer.startServer("127.0.0.1", 7000);
    }
}
```

7、Netty的客户端的处理handler

```
package com.kkb.demo.netty.example.dubborpc.netty;

public class NettyClientHandler extends ChannelInboundHandlerAdapter
implements Callable {
...........
}
```

8、Netty客户端

```
package com.kkb.demo.netty.example.dubborpc.netty;

public class NettyClient {
...
}
```

9、服务消费者启动类

```
package com.kkb.demo.netty.example.dubborpc.customer;

public class ClientBootstrap {
..
}
```

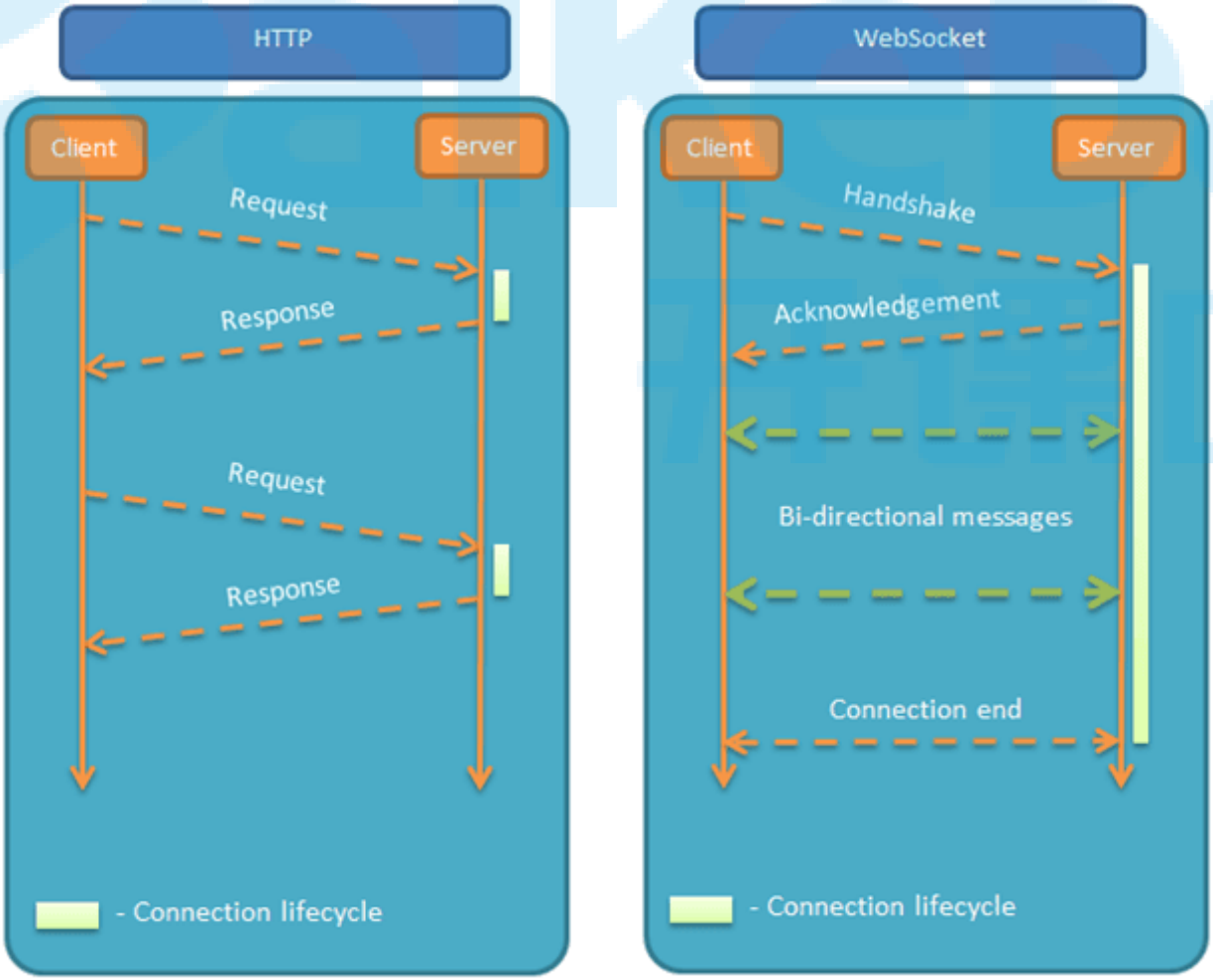
1.4 Netty实现Websocket

1.4.1 WebSocket协议介绍

HTTP协议的主要弊端:

- 半双工协议：可以在客户端和服务端2个方向上传输，但不能同时传输。同一时刻，只能在一个方向传输。
- HTTP消息冗长：相比于其他二进制协议，有点繁琐。
- 黑客攻击，例如长时间轮询。现在很多网站的消息推送都是使用轮询，即客户端每隔1S或者其他时间给服务器发送请求，然后服务器返回最新的数据给客户端。HTTP协议中的**Header**非常冗长，因此会占用很多的带宽和服务器资源。

为了解决这个问题，HTML5定义的WebSocket协议。



WebSocket协议

在WebSocket API中，浏览器和服务器**只需要一个握手**的动作，然后，浏览器和服务器之间就形成了一条快速通道，两者就可以直接互相传送数据了。

WebSocket基于**TCP双向全双工协议**，即在同一时刻，即可以发送消息，也可以接收消息，相比于HTTP协议，是一个性能上的提升。

特点：

- 单一的TCP连接，全双工；
- 对代理、防火墙和路由器透明；
- 无头部信息、Cookie和身份验证；
- 无安全开销；
- 通过"ping/pong"帧保持链路激活;
- 服务器可以主动传递消息给客户端，不再需要客户端轮询；

拥有以上特点的WebSocket就是为了取代轮询和Comet技术，使得客户端浏览器具备像C/S架构下桌面系统一样的实时能力。

浏览器通过js建立一个WebSocket的请求，连接建立后，客户端和服务器端可以通过TCP直接交换数据。

因为WebSocket本质上是一个TCP连接，稳定，所以在Comet和轮询比拥有性能优势，

1.4.2 实现WebSocket

1、服务端启动

```
package com.kkb.demo.netty.example.webs;


public class MyWebSocketServer {
...
}
```

2、服务端ChannelInitializer

```
package com.kkb.demo.netty.example.webs;
....

public class WebSocketServerInitializer extends
ChannelInitializer<SocketChannel> {
  ....
}
```

3、服务端处理Handler

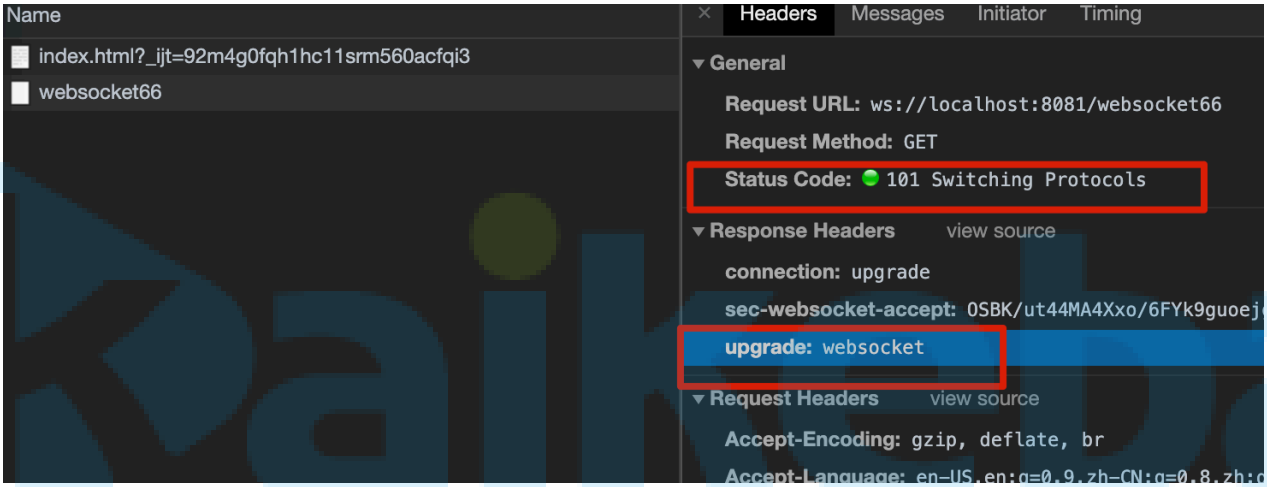
```
package com.kkb.demo.netty.example.webs;



public class WebSocketRequestHandler extends
SimpleChannelInboundHandler<Object> {
  ...
}
```

4、客户端(页面)

```
index.html
```



5、服务端推送消息

```
package com.kkb.demo.netty.example.webs;


public class ChannelSupervise {
...
}
```

补充:

- netty的websocket协议是在HTTP协议基础之上完成的，要使用WebSocket协议，需要将HTTP请求头中添加`Upgrade:WebSocket`
- WebSocket相关的编解码

```
public abstract class WebSocketServerHandshaker {
....
  //WebSocket相关的编解码
public final ChannelFuture handshake(Channel channel, FullHttpRequest req,
                                     HttpHeaders responseHeaders, final
ChannelPromise promise) {
```

```
        if (logger.isDebugEnabled()) {
            logger.debug("{} WebSocket version {} server handshake", channel,
version());
        }
  //构造握手响应
        FullHttpResponse response = newHandshakeResponse(req,
responseHeaders);
  //下面将channelpipeline中的HttpObjectAggregator和HttpContentCompressor移除，并且添加WebSocket编解码器newWebSocketEncoder和newWebsocketDecoder
        ChannelPipeline p = channel.pipeline();
        if (p.get(HttpObjectAggregator.class) != null) {
            p.remove(HttpObjectAggregator.class);
        }
        if (p.get(HttpContentCompressor.class) != null) {
            p.remove(HttpContentCompressor.class);
        }
        ChannelHandlerContext ctx = p.context(HttpRequestDecoder.class);
        final String encoderName;
        if (ctx == null) {
            // this means the user use an HttpServerCodec
            ctx = p.context(HttpServerCodec.class);
            if (ctx == null) {
                promise.setFailure(
                        new IllegalStateException("No HttpDecoder and no
HttpServerCodec in the pipeline"));
                return promise;
            }
            p.addBefore(ctx.name(), "wsencoder", newWebSocketEncoder());
            p.addBefore(ctx.name(), "wsdecoder", newWebsocketDecoder());
            encoderName = ctx.name();
        } else {
            p.replace(ctx.name(), "wsdecoder", newWebsocketDecoder());

            encoderName = p.context(HttpResponseEncoder.class).name();
            p.addBefore(encoderName, "wsencoder", newWebSocketEncoder());
        }
  //将response消息返回给客户端
        channel.writeAndFlush(response).addListener(new
ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) throws
Exception {
                if (future.isSuccess()) {
                    ChannelPipeline p = future.channel().pipeline();
                    p.remove(encoderName);
                    promise.setSuccess();
                } else {
                    promise.setFailure(future.cause());
                }
```

```
            }
        });
        return promise;
    }
  ...
```

1.5 Netty实现Tomcat

1.5.1 编解码器

```
public class HttpHelloWorldServerInitializer extends
ChannelInitializer<SocketChannel> {
...
}
```

1.5.2 GET解析

```
if (method.equals(HttpMethod.GET)){
                QueryStringDecoder queryDecoder = new QueryStringDecoder(uri,
CharsetUtil.UTF_8);
                ...
            }
```

1.5.3 POST解析

FullHttpRequest 包含了 HttpRequest 和 FullHttpMessage，是一个 HTTP 请求的完全体。

解析Content-Type

```
private void dealWithContentType() throws Exception{
        String contentType = getContentType();
        //可以使用HttpJsonDecoder
        if(contentType.equals("application/json")){
            String jsonStr =
fullRequest.content().toString(CharsetUtil.UTF_8);
            JSONObject obj = JSON.parseObject(jsonStr);
            for(Map.Entry<String, Object> item : obj.entrySet()){
                logger.info(item.getKey()+"="+item.getValue().toString());
            }

        }else if(contentType.equals("application/x-www-form-urlencoded")){
            //方式一：使用 QueryStringDecoder
            String jsonStr =
fullRequest.content().toString(CharsetUtil.UTF_8);
            QueryStringDecoder queryDecoder = new QueryStringDecoder(jsonStr,
false);
```

```
            Map<String, List<String>> uriAttributes =
queryDecoder.parameters();
            for (Map.Entry<String, List<String>> attr :
uriAttributes.entrySet()) {
                for (String attrVal : attr.getValue()) {
                    logger.info(attr.getKey() + "=" + attrVal);
                }
            }

        }else if(contentType.equals("multipart/form-data")){
            //TODO 用于文件上传
        }else{
            //do nothing...
        }
    }
```

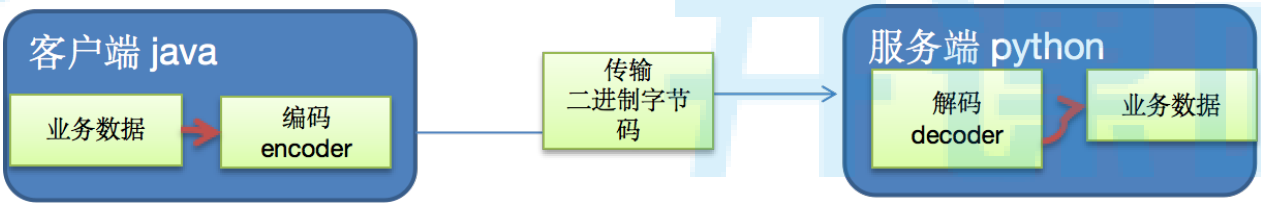
2 Google Protobuf

2.1 编码和解码的基本介绍

编写网络应用程序时，因为数据在网络中传输的都是二进制字节码数据，在发送数据时就需要编码，接收数据时就需要解码[示意图]

codec（编解码器）的组成部分有两个：decoder（解码器）和 encoder（编码器）；

encoder 负责把业务数据转换成字节码数据，decoder 负责把字节码数据转换成业务数据



2.2 Netty 本身的编码解码的机制和问题分析

1. Netty 自身提供了一些 codec(编解码器)
2. Netty 提供的编码器 StringEncoder，对字符串数据进行编码 ObjectEncoder，对java对象进行编码。
3. Netty 提供的解码器 StringDecoder，对字符串数据进行解码 ObjectDecoder，对 java 对象进行解码。
4. Netty 本身自带的 ObjectDecoder 和 ObjectEncoder 可以用来实现 POJO 对象或各种业务对象的编码和解码，底层使用的仍是Java序列化技术,而Java序列化技术本身效率就不高，存在如下问题
    - 无法跨语言
    - 序列化后的体积太大，是二进制编码的5倍多。
    - 序列化性能太低

引出新的解决方案[Google 的 Protobuf]

2.3 Protobuf

Protobuf是由谷歌开源而来，在谷歌内部久经考验。它将数据结构以.proto文件进行描述，通过代码生成工具可以生成对应数据结构的POJO对象和Protobuf相关的方法和属性。

特点如下:

- 结构化数据存储格式(XML,JSON等)
- 高效的编解码性能
- 语言无关、平台无关、扩展性好

数据交互xml、json、protobuf格式比较

1. json: 一般的web项目中，最流行的主要还是json。因为浏览器对于json数据支持非常好，有很多内建的函数支持。
2. xml: 在webservice中应用最为广泛，但是相比于json，它的数据更加冗余，因为需要成对的闭合标签。json使用了键值对的方式，不仅压缩了一定的数据空间，同时也具有可读性。
3. protobuf:是后起之秀，是谷歌开源的一种数据格式，适合高性能，对响应速度有要求的数据传输场景。因为profobuf是二进制数据格式，需要编码和解码。数据本身不具有可读性。因此只能反序列化之后得到真正可读的数据。

相对于其它protobuf更具有优势

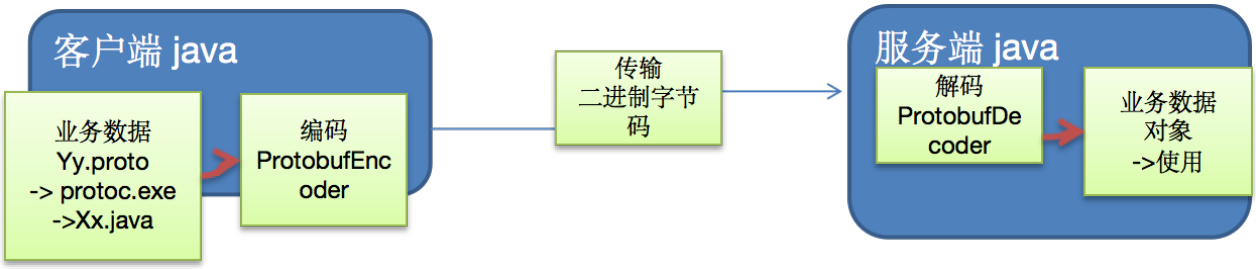
1. 序列化后体积相比Json和XML很小，适合网络传输
2. 支持跨平台多语言
3. 消息格式升级和兼容性还不错
4. 序列化反序列化速度很快，快于Json的处理速速

结论： 在一个需要大量的数据传输的场景中，如果数据量很大，那么选择protobuf可以明显的减少数据量，减少网络IO，从而减少网络传输所消耗的时间。

因而，对于打造一款高性能的通讯服务器来说，protobuf 传输格式，是最佳的解决方案。

参考文档：https://developers.google.com/protocol-buffers/docs/proto 语言指南

protobuf 使用示意图



2.4 示例操作步骤

1、idea安装插件Protobuf Support

2、安装protobuf编译器

https://github.com/protocolbuffers/protobuf/releases

3、添加依赖

```
<!-- https://mvnrepository.com/artifact/com.google.protobuf/protobuf-java -->
    <dependency>
      <groupId>com.google.protobuf</groupId>
      <artifactId>protobuf-java</artifactId>
      <version>3.14.0</version>
    </dependency>
```

4、编写文件Student.proto

```
syntax = "proto3";
option optimize_for = SPEED; // 加快解析
option java_package="com.kkb.demo.netty.example.codec";   //指定生成到哪个包下
option java_outer_classname="MyDataInfo"; // 外部类名, 文件名

//protobuf 可以使用message 管理其他的message
message MyMessage {

    //定义一个枚举类型
    enum DataType {
        StudentType = 0; //在proto3 要求enum的编号从0开始
        WorkerType = 1;
    }

    //用data_type 来标识传的是哪一个枚举类型
    DataType data_type = 1;

    //表示每次枚举类型最多只能出现其中的一个, 节省空间
    oneof dataBody {
        Student student = 2;
        Worker worker = 3;
    }

}


message Student {
    int32 id = 1;//Student类的属性
    string name = 2; //
}
message Worker {
    string name=1;
    int32 age=2;
}
```

5、编写 .proto 文件，生成java文件

```
➜  codec git:(master) ✗ ~/protobuf/bin/protoc --version
libprotoc 3.14.0

➜  java git:(master) ✗ pwd
/Users/hadoop/handout/Netty/netty-example/src/main/java
➜  java git:(master) ✗ ~/protobuf/bin/protoc --java_out=.
com/kkb/demo/netty/example/codec/Student.proto
```

6、Netty客户端 Handler发送

```
public class NettyClientHandler extends ChannelInboundHandlerAdapter {
    //当通道就绪就会触发该方法
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {

//        MyDataInfo.MyMessage message =
//                MyDataInfo.MyMessage.newBuilder().setDataType(
MyDataInfo.MyMessage.DataType.StudentType)
//
.setStudent(MyDataInfo.Student.newBuilder().setId(6).setName("北
京").build()).build();
//        ctx.writeAndFlush(message);
        MyDataInfo.MyMessage message =
                MyDataInfo.MyMessage.newBuilder().setDataType(
MyDataInfo.MyMessage.DataType.WorkerType)

.setWorker(MyDataInfo.Worker.newBuilder().setAge(20).setName("杭
州").build()).build();
        ctx.writeAndFlush(message);
    }

  .....
}
```

7、Netty客户端添加编码器

```
 //设置相关参数
            bootstrap.group(group) //设置线程组
                    .channel(NioSocketChannel.class) // 设置客户端通道的实现类(反
射)
                    .handler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch) throws
Exception {
                            ChannelPipeline channelPipeline = ch.pipeline();
                            channelPipeline.addLast("encoder",new
ProtobufEncoder());
                            channelPipeline.addLast(new NettyClientHandler());
//加入自己的处理器
                        }
                    });
```

8、Netty服务端添加解码器

```
.childHandler(new ChannelInitializer<SocketChannel>() {//创建一个通道初始化对象(匿
名对象)
                        //给pipeline 设置处理器
                        @Override
                        protected void initChannel(SocketChannel ch) throws
Exception {
                            System.out.println("客户socketchannel hashcode=" +
ch.hashCode()); //可以使用一个集合管理 SocketChannel， 再推送消息时，可以将业务加入到各
个channel 对应的 NIOEventLoop 的 taskQueue 或者 scheduleTaskQueue
                            ChannelPipeline pipeline = ch.pipeline();
                            pipeline.addLast("decoder",new
ProtobufDecoder(MyDataInfo.MyMessage.getDefaultInstance()));
                            pipeline.addLast(new NettyServerHandler());
                        }
                    });
```

9、Netty服务端Handler接收消息

```
public class NettyServerHandler extends
SimpleChannelInboundHandler<MyDataInfo.MyMessage> {


    //读取数据实际(这里我们可以读取客户端发送的消息)

    @Override
    public void channelRead0(ChannelHandlerContext ctx, MyDataInfo.MyMessage
msg) throws Exception {

        MyDataInfo.MyMessage.DataType dataType = msg.getDataType();
        if(dataType==MyDataInfo.MyMessage.DataType.StudentType){
```

```
            MyDataInfo.Student student = msg.getStudent();
            System.out.println("student
id="+student.getId()+",name="+student.getName());

        }else if(dataType==MyDataInfo.MyMessage.DataType.WorkerType){
            MyDataInfo.Worker worker = msg.getWorker();
            System.out.println("woker
age="+worker.getAge()+",name="+worker.getName());
        }else {
            System.out.println("类型不存在");
        }
    }

  ...
}
```

kaikeba 开课吧