Netty 高级

Netty 快速入门

什么是 Netty

Netty 是一个基于 JAVA NIO 类库的异步通信框架，它的架构特点是：异步非阻塞、基于事件驱动、高性能、高可靠性和高可定制性。

Netty 应用场景

1.分布式开源框架中 dubbo、Zookeeper，RocketMQ 底层 rpc 通讯使用就是 netty。
2.游戏开发中，底层使用 netty 通讯。

为什么选择 netty

在本小节，我们总结下为什么不建议开发者直接使用 JDK 的 NIO 类库进行开发的原因：
1) NIO 的类库和 API 繁杂，使用麻烦，你需要熟练掌握 Selector、ServerSocketChannel、SocketChannel、ByteBuffer 等；
2) 需要具备其它的额外技能做铺垫，例如熟悉 Java 多线程编程，因为 NIO 编程涉及到 Reactor 模式，你必须对多线程和网路编程非常熟悉，才能编写出高质量的 NIO 程序；
3) 可靠性能力补齐，工作量和难度都非常大。例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处 n 理等等，NIO 编程的特点是功能开发相对容易，但是可靠性能力补齐工作量和难度都非常大；
4) JDK NIO 的 BUG，例如臭名昭著的 epoll bug，它会导致 Selector 空轮询，最终导致 CPU 100%。官方声称在 JDK1.6 版本的 update18 修复了该问题，但是直到 JDK1.7 版本该问题仍旧存在，只不过该 bug 发生概率降低了一些而已，它并没有被根本解决。该 BUG 以及与该 BUG 相关的问题单如下：

Netty 服务器端

```
class ServerHandler extends SimpleChannelHandler {



    /**
     * 通道关闭的时候触发
     */
```

```
	@Override
	public void channelClosed(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
		System.out.println("channelClosed");
	}

	/**
	 * 必须是连接已经建立,关闭通道的时候才会触发.
	 */
	@Override
	public void channelDisconnected(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
		super.channelDisconnected(ctx, e);
		System.out.println("channelDisconnected");
	}

	/**
	 * 捕获异常
	 */
	@Override
	public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) throws Exception {
		super.exceptionCaught(ctx, e);
		System.out.println("exceptionCaught");

	}

	/**
	 * 接受消息
	 */
	public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) throws Exception {
		super.messageReceived(ctx, e);
//		System.out.println("messageReceived");
		System.out.println("服务器端收到客户端消息:"+e.getMessage());
		//回复内容
		ctx.getChannel().write("好的");
	}

}
// netty 服务器端
public class NettyServer {

	public static void main(String[] args) {
		// 创建服务类对象
		ServerBootstrap serverBootstrap = new ServerBootstrap();
		// 创建两个线程池 分别为监听监听端口 , nio监听
		ExecutorService boos = Executors.newCachedThreadPool();
```

```
            ExecutorService worker = Executors.newCachedThreadPool();
            // 设置工程 并把两个线程池加入中
            serverBootstrap.setFactory(new NioServerSocketChannelFactory(boos, worker));
            // 设置管道工厂
            serverBootstrap.setPipelineFactory(new ChannelPipelineFactory() {

                  public ChannelPipeline getPipeline() throws Exception {
                        ChannelPipeline pipeline = Channels.pipeline();
                        //将数据转换为string类型.
                        pipeline.addLast("decoder", new StringDecoder());
                        pipeline.addLast("encoder", new StringEncoder());
                        pipeline.addLast("serverHandler", new ServerHandler());
                        return pipeline;
                  }
            });
            // 绑定端口号
            serverBootstrap.bind(new InetSocketAddress(9090));
            System.out.println("netty server启动....");
      }


}
```

Netty 客户端

```
package com.itmayiedu;

import java.net.InetSocketAddress;
import java.util.Scanner;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import org.jboss.netty.bootstrap.ClientBootstrap;
import org.jboss.netty.channel.Channel;
import org.jboss.netty.channel.ChannelFuture;
import org.jboss.netty.channel.ChannelHandlerContext;
import org.jboss.netty.channel.ChannelPipeline;
import org.jboss.netty.channel.ChannelPipelineFactory;
import org.jboss.netty.channel.ChannelStateEvent;
import org.jboss.netty.channel.Channels;
import org.jboss.netty.channel.ExceptionEvent;
```

```
import org.jboss.netty.channel.MessageEvent;
import org.jboss.netty.channel.SimpleChannelHandler;
import org.jboss.netty.channel.socket.nio.NioClientSocketChannelFactory;
import org.jboss.netty.handler.codec.string.StringDecoder;
import org.jboss.netty.handler.codec.string.StringEncoder;
class ClientHandler extends SimpleChannelHandler {



	/**
	 * 通道关闭的时候触发
	 */
	@Override
	public void channelClosed(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
		System.out.println("channelClosed");
	}

	/**
	 * 必须是连接已经建立,关闭通道的时候才会触发.
	 */
	@Override
	public void channelDisconnected(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
		super.channelDisconnected(ctx, e);
		System.out.println("channelDisconnected");
	}

	/**
	 * 捕获异常
	 */
	@Override
	public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) throws Exception {
		super.exceptionCaught(ctx, e);
		System.out.println("exceptionCaught");

	}

	/**
	 * 接受消息
	 */
	public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) throws Exception {
		super.messageReceived(ctx, e);
//		System.out.println("messageReceived");
		System.out.println("服务器端向客户端回复内容:"+e.getMessage());
		//回复内容
//		ctx.getChannel().write("好的");
```

```
	}


}
public class NettyClient {


	public static void main(String[] args) {
		System.out.println("netty client启动...");
		// 创建客户端类
		ClientBootstrap clientBootstrap = new ClientBootstrap();
		// 线程池
		ExecutorService boos = Executors.newCachedThreadPool();
		ExecutorService worker = Executors.newCachedThreadPool();
		clientBootstrap.setFactory(new NioClientSocketChannelFactory(boos, worker));
		clientBootstrap.setPipelineFactory(new ChannelPipelineFactory() {


			public ChannelPipeline getPipeline() throws Exception {
				ChannelPipeline pipeline = Channels.pipeline();
				// 将数据转换为string类型.
				pipeline.addLast("decoder", new StringDecoder());
				pipeline.addLast("encoder", new StringEncoder());
				pipeline.addLast("clientHandler", new ClientHandler());
				return pipeline;


			}
		});
		//连接服务端
		ChannelFuture connect = clientBootstrap.connect(new InetSocketAddress("127.0.0.1", 9090));
		Channel channel = connect.getChannel();
		System.out.println("client start");
		Scanner scanner= new Scanner(System.in);
		while (true) {
			System.out.println("请输输入内容...");
			channel.write(scanner.next());
		}
	}



}
```

Maven 坐标

```
<dependency>
		<groupId>io.netty</groupId>
		<artifactId>netty</artifactId>
		<version>3.3.0.Final</version>
	</dependency>
```

Netty5.0 用法

创建服务器端

```
class ServerHandler extends ChannelHandlerAdapter {
	/**
	 * 当通道被调用,执行该方法
	 */
	@Override
	public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
		// 接收数据
		String value = (String) msg;
		System.out.println("Server msg:" + value);
		// 回复给客户端 “您好!”
		String res = "好的...";
		ctx.writeAndFlush(Unpooled.copiedBuffer(res.getBytes()));
	}

}

public class NettyServer {

	public static void main(String[] args) throws InterruptedException {
		System.out.println("服务器端已经启动....");
		// 1.创建2个线程,一个负责接收客户端连接， 一个负责进行 传输数据
		NioEventLoopGroup pGroup = new NioEventLoopGroup();
		NioEventLoopGroup cGroup = new NioEventLoopGroup();
		// 2. 创建服务器辅助类
		ServerBootstrap b = new ServerBootstrap();
		b.group(pGroup, cGroup).channel(NioServerSocketChannel.class).option(ChannelOption.SO_BACKLOG,
1024)
				// 3.设置缓冲区与发送区大小
```

```
                    .option(ChannelOption.SO_SNDBUF, 32 * 1024).option(ChannelOption.SO_RCVBUF, 32 * 1024)
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel sc) throws Exception {
                            sc.pipeline().addLast(new StringDecoder());
                            sc.pipeline().addLast(new ServerHandler());
                        }
                    });
            ChannelFuture cf = b.bind(8080).sync();
            cf.channel().closeFuture().sync();
            pGroup.shutdownGracefully();
            cGroup.shutdownGracefully();

        }

}
```

创建客户端

```
class ClientHandler extends ChannelHandlerAdapter {

    /**
     * 当通道被调用,执行该方法
     */
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        // 接收数据
        String value = (String) msg;
        System.out.println("client msg:" + value);
    }


}

public class NettyClient {

    public static void main(String[] args) throws InterruptedException {
        System.out.println("客户端已经启动....");
        // 创建负责接收客户端连接
        NioEventLoopGroup pGroup = new NioEventLoopGroup();
```

```
            Bootstrap b = new Bootstrap();
            b.group(pGroup).channel(NioSocketChannel.class).handler(new ChannelInitializer<SocketChannel>() {
                  @Override
                  protected void initChannel(SocketChannel sc) throws Exception {
                        sc.pipeline().addLast(new StringDecoder());
                        sc.pipeline().addLast(new ClientHandler());
                  }
            });
            ChannelFuture cf = b.connect("127.0.0.1", 8080).sync();
             cf.channel().writeAndFlush(Unpooled.wrappedBuffer("itmayiedu".getBytes()));
             cf.channel().writeAndFlush(Unpooled.wrappedBuffer("itmayiedu".getBytes()));
            // 等待客户端端口号关闭
            cf.channel().closeFuture().sync();
            pGroup.shutdownGracefully();

      }

}
```

Maven 坐标

```
      <dependencies>
            <!-- https://mvnrepository.com/artifact/io.netty/netty-all -->
            <dependency>
                  <groupId>io.netty</groupId>
                  <artifactId>netty-all</artifactId>
                  <version>5.0.0.Alpha2</version>
            </dependency>

            <!-- https://mvnrepository.com/artifact/org.jboss.marshalling/jboss-marshalling -->
            <dependency>
                  <groupId>org.jboss.marshalling</groupId>
                  <artifactId>jboss-marshalling</artifactId>
                  <version>1.3.19.GA</version>
            </dependency>
            <!-- https://mvnrepository.com/artifact/org.jboss.marshalling/jboss-marshalling-serial -->
            <dependency>
                  <groupId>org.jboss.marshalling</groupId>
                  <artifactId>jboss-marshalling-serial</artifactId>
                  <version>1.3.18.GA</version>
                  <scope>test</scope>
            </dependency>
```

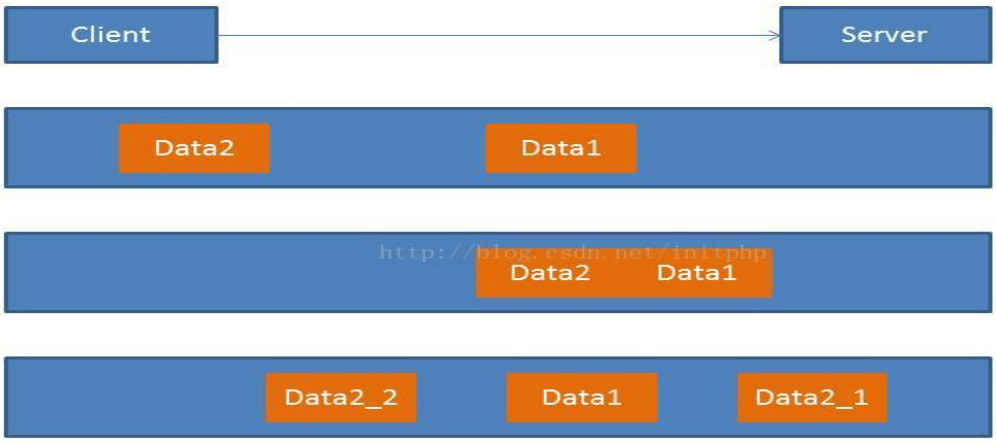
```
</dependencies>
```

TCP 粘包、拆包问题解决方案

什么是粘包/拆包

一个完整的业务可能会被 TCP 拆分成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送，这个就是 TCP 的拆包和封包问题。

下面可以看一张图，是客户端向服务端发送包：



1. 第一种情况，Data1 和 Data2 都分开发送到了 Server 端，没有产生粘包和拆包的情况。
2. 第二种情况，Data1 和 Data2 数据粘在了一起，打成了一个大的包发送到 Server 端，这个情况就是粘包。
3. 第三种情况，Data2 被分离成 Data2_1 和 Data2_2，并且 Data2_1 在 Data1 之前到达了服务端，这种情况就产生了拆包。

由于网络的复杂性，可能数据会被分离成 N 多个复杂的拆包/粘包的情况，所以在做 TCP 服务器的时候就需要首先解决拆包/

解决办法

消息定长，报文大小固定长度，不够空格补全，发送和接收方遵循相同的约定，这样即使粘包了通过接收方编程实现获取定长报文也能区分。

```
sc.pipeline().addLast(new FixedLengthFrameDecoder(10));
```

包尾添加特殊分隔符，例如每条报文结束都添加回车换行符（例如 FTP 协议）或者指定特殊字符作为报文分隔符，接收方通过特殊分隔符切分报文区分。

```
ByteBuf buf = Unpooled.copiedBuffer("_mayi".getBytes());
sc.pipeline().addLast(new DelimiterBasedFrameDecoder(1024, buf));
```

将消息分为消息头和消息体，消息头中包含表示信息的总长度（或者消息体长度）的字段

序列化协议与自定义序列化协议

序列化定义

序列化（serialization）就是将对象序列化为二进制形式（字节数组），一般也将序列化称为编码（Encode），主要用于网络传输、数据持久化等；
反序列化（deserialization）则是将从网络、磁盘等读取的字节数组还原成原始对象，以便后续业务的进行，一般也将反序列化称为解码（Decode），主要用于网络传输对象的解码，以便完成远程调用。

序列化协议“鼻祖”

我知道的第一种序列化协议就是 Java 默认提供的序列化机制，需要序列化的 Java 对象只需要实现 Serializable / Externalizable 接口并生成序列化 ID，这个类就能够通过 ObjectInput 和 ObjectOutput 序列化和反序列化，若对 Java 默认的序列化协议不了解，或是遗忘了，请参考：序列化详解
但是 Java 默认提供的序列化有很多问题，主要有以下几个缺点：
无法跨语言：我认为这对于 Java 序列化的发展是致命的“失误”，因为 Java 序列化后的字节数组，其它语言无法进行反序列化。；
序列化后的码流太大:：相对于目前主流的序列化协议，Java 序列化后的码流太大；
序列化的性能差：由于 Java 序列化采用同步阻塞 IO，相对于目前主流的序列化协议，它的效率非常差。
影响序列化性能的关键因素
序列化后的码流大小（网络带宽的占用）；
序列化的性能（CPU 资源占用）；
是否支持跨语言（异构系统的对接和开发语言切换）。

几种流行的序列化协议比较

XML

（1）定义：
XML（Extensible Markup Language）是一种常用的序列化和反序列化协议， 它历史悠久，从 1998 年的 1.0 版本被广泛使用至今。
（2）优点

人机可读性好

可指定元素或特性的名称

（3）缺点

序列化数据只包含数据本身以及类的结构，不包括类型标识和程序集信息。

类必须有一个将由 XmlSerializer 序列化的默认构造函数。

只能序列化公共属性和字段

不能序列化方法

文件庞大，文件格式复杂，传输占带宽

（4）使用场景

当做配置文件存储数据

实时数据转换

JSON

（1）定义：

JSON(JavaScript Object Notation, JS 对象标记) 是一种轻量级的数据交换格式。它基于 ECMAScript (w3c 制定的 js 规范)的一个子集， JSON 采用与编程语言无关的文本格式,但是也使用了类 C 语言(包括 C， C++， C#, Java， JavaScript， Perl， Python 等）的习惯，简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言。

（2）优点

前后兼容性高

数据格式比较简单，易于读写

序列化后数据较小，可扩展性好，兼容性好

与 XML 相比，其协议比较简单，解析速度比较快

（3）缺点

数据的描述性比 XML 差

不适合性能要求为 ms 级别的情况

额外空间开销比较大

（4）适用场景（可替代ＸＭＬ）

跨防火墙访问

可调式性要求高的情况

基于 Web browser 的 Ajax 请求

传输数据量相对小，实时性要求相对低（例如秒级别）的服务

Fastjson

（1）定义

Fastjson 是一个 Java 语言编写的高性能功能完善的 JSON 库。它采用一种假定有序快速匹配”的算法,把 JSON Parse 的性能提升到极致。

（2）优点

接口简单易用

目前 java 语言中最快的 json 库

（3）缺点

过于注重快，而偏离了“标准”及功能性
代码质量不高，文档不全
（4）适用场景
协议交互
Web 输出
Android 客户端

Thrift

（1）定义：
Thrift 并不仅仅是序列化协议，而是一个 RPC 框架。它可以让你选择客户端与服务端之间传输通信协议的类别，即文本(text)和二进制(binary)传输协议，为节约带宽，提供传输效率，一般情况下使用二进制类型的传输协议。
（2）优点
序列化后的体积小，速度快
支持多种语言和丰富的数据类型
对于数据字段的增删具有较强的兼容性
支持二进制压缩编码
（3）缺点
使用者较少
跨防火墙访问时，不安全
不具有可读性，调试代码时相对困难
不能与其他传输层协议共同使用（例如 HTTP）
无法支持向持久层直接读写数据，即不适合做数据持久化序列化协议
（4）适用场景
分布式系统的 RPC 解决方案

Avro

（1）定义：
Avro 属于 Apache Hadoop 的一个子项目。 Avro 提供两种序列化格式：JSON 格式或者 Binary 格式。Binary 格式在空间开销和解析性能方面可以和 Protobuf 媲美，Avro 的产生解决了 JSON 的冗长和没有 IDL 的问题
（2）优点
支持丰富的数据类型
简单的动态语言结合功能
具有自我描述属性
提高了数据解析速度
快速可压缩的二进制数据形式
可以实现远程过程调用 RPC
支持跨编程语言实现
（3）缺点
对于习惯于静态类型语言的用户不直观
（4）适用场景

在 Hadoop 中做 Hive、Pig 和 MapReduce 的持久化数据格式

Protobuf

（1）定义

protocol buffers 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。

（2）优点

序列化后码流小，性能高

结构化数据存储格式（XML JSON 等）

通过标识字段的顺序，可以实现协议的前向兼容

结构化的文档更容易管理和维护

（3）缺点

需要依赖于工具生成代码

支持的语言相对较少，官方只支持 Java 、C++ 、Python

（4）适用场景

对性能要求高的 RPC 调用

具有良好的跨防火墙的访问属性

适合应用层对象的持久化

其它

protostuff 基于 protobuf 协议，但不需要配置 proto 文件，直接导包即

Jboss marshaling 可以直接序列化 java 类， 无须实 java.io.Serializable 接口

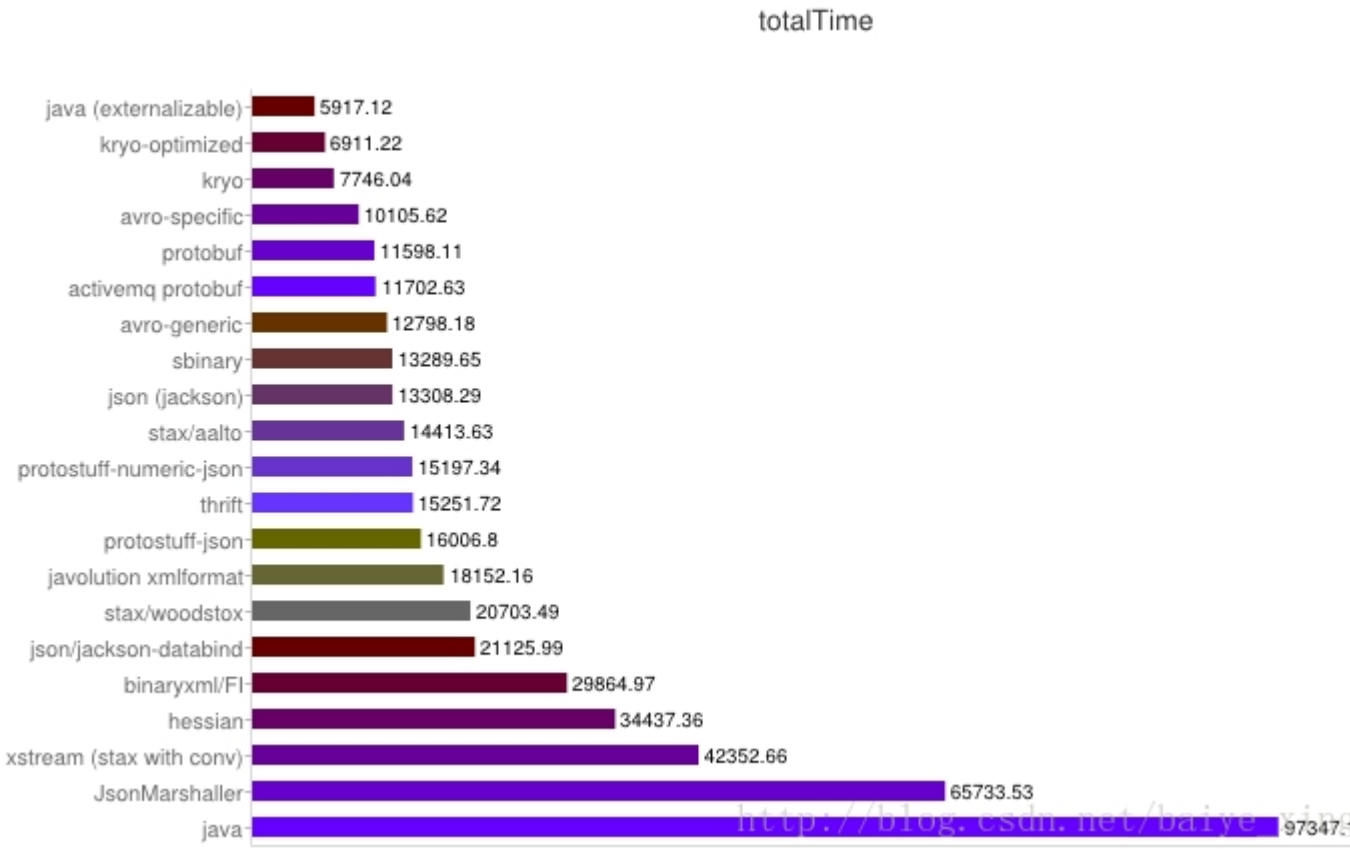
Message pack 一个高效的二进制序列化格式

Hessian 采用二进制协议的轻量级 remoting onhttp 工具

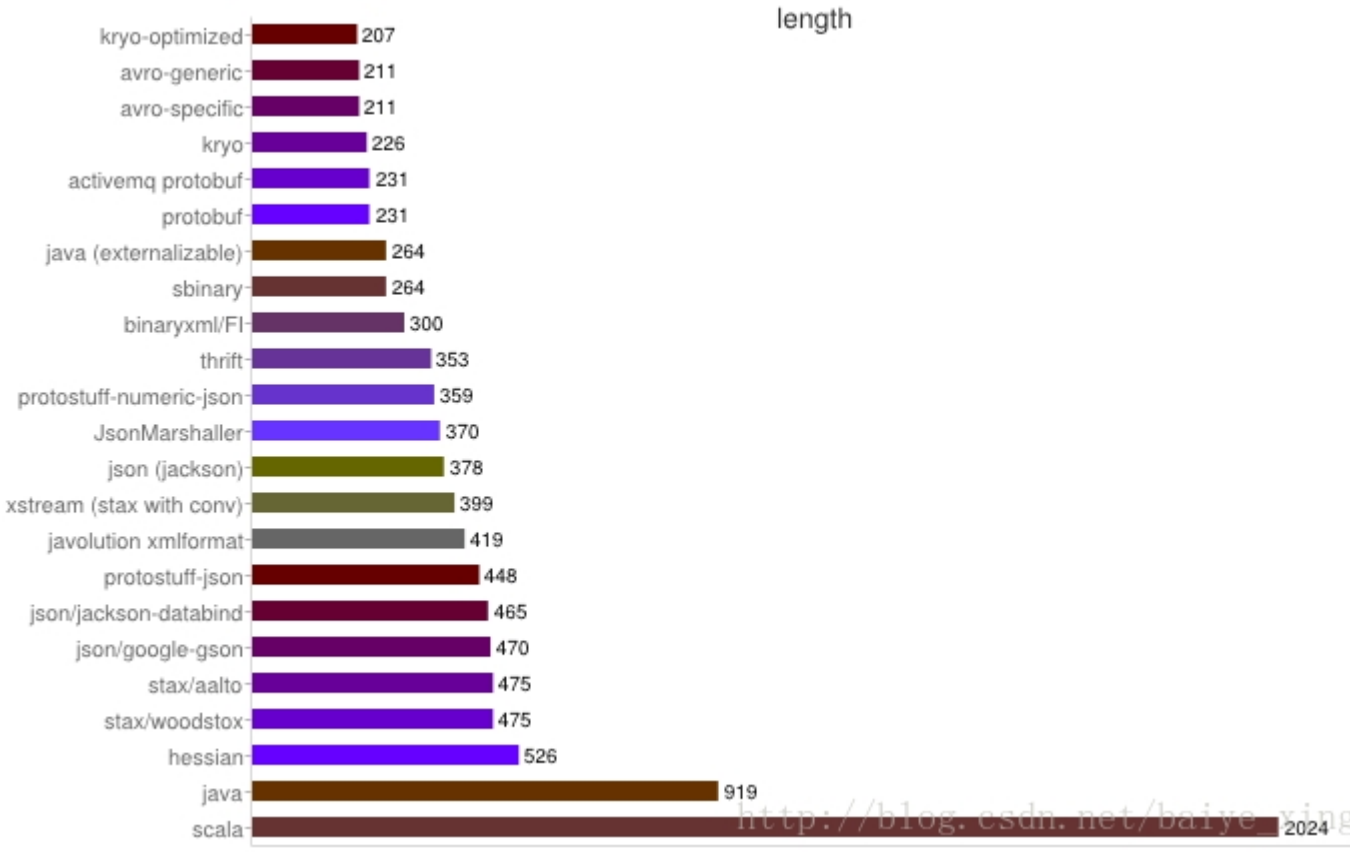
kryo 基于 protobuf 协议，只支持 java 语言,需要注册（Registration），然后序列化（Output），反序列化（Input）

性能对比图解

时间


totalTime
java (externalizable) 5917.12
kryo-optimized 6911.22
kryo 7746.04
avro-specific 10105.62
protobuf 11598.11
activemq protobuf 11702.63
avro-generic 12798.18
sbinary 13289.65
json (jackson) 13308.29
stax/aalto 14413.63
protostuff-numeric-json 15197.34
thrift 15251.72
protostuff-json 16006.8
javolution xmlformat 18152.16
stax/woodstox 20703.49
json/jackson-databind 21125.99
binaryxml/FI 29864.97
hessian 34437.36
xstream (stax with conv) 42352.66
JsonMarshaller 65733.53
java 97347

空间


length
kryo-optimized 207
avro-generic 211
avro-specific 211
kryo 226
activemq protobuf 231
protobuf 231
java (externalizable) 264
sbinary 264
binaryxml/FI 300
thrift 353
protostuff-numeric-json 359
JsonMarshaller 370
json (jackson) 378
xstream (stax with conv) 399
javolution xmlformat 419
protostuff-json 448
json/jackson-databind 465
json/google-gson 470
stax/aalto 475
stax/woodstox 475
hessian 526
java 919
scala 2024

分析上图知：

XML 序列化（Xstream）无论在性能和简洁性上比较差。

Thrift 与 Protobuf 相比在时空开销方面都有一定的劣势。

Protobuf 和 Avro 在两方面表现都非常优越。

选型建议

不同的场景适用的序列化协议：

对于公司间的系统调用，如果性能要求在 100ms 以上的服务，基于 XML 的 SOAP 协议是一个值得考虑的方案。

基于 Web browser 的 Ajax，以及 Mobile app 与服务端之间的通讯，JSON 协议是首选。对于性能要求不太高，或者以动态类型语言为主，或者传输数据载荷很小的的运用场景，JSON 也是非常不错的选择。

对于调试环境比较恶劣的场景，采用 JSON 或 XML 能够极大的提高调试效率，降低系统开发成本。

当对性能和简洁性有极高要求的场景，Protobuf，Thrift，Avro 之间具有一定的竞争关系。

对于 T 级别的数据的持久化应用场景，Protobuf 和 Avro 是首要选择。如果持久化后的数据存储在 Hadoop 子项目里，Avro 会是更好的选择。

由于 Avro 的设计理念偏向于动态类型语言，对于动态语言为主的应用场景，Avro 是更好的选择。

对于持久层非 Hadoop 项目，以静态类型语言为主的应用场景，Protobuf 会更符合静态类型语言工程师的开发习惯。

如果需要提供一个完整的 RPC 解决方案，Thrift 是一个好的选择。

如果序列化之后需要支持不同的传输层协议，或者需要跨防火墙访问的高性能场景，Protobuf 可以优先考虑。

Marshalling 编码器

```
public final class MarshallingCodeCFactory {

    /**
     * 创建Jboss Marshalling解码器MarshallingDecoder
     */
    public static MarshallingDecoder buildMarshallingDecoder() {
        final MarshallerFactory marshallerFactory = Marshalling.getProvidedMarshallerFactory("serial");
        final MarshallingConfiguration configuration = new MarshallingConfiguration();
        configuration.setVersion(5);
        UnmarshallerProvider provider = new DefaultUnmarshallerProvider(marshallerFactory, configuration);
        MarshallingDecoder decoder = new MarshallingDecoder(provider, 1024);
        return decoder;
    }

    /**
     * 创建Jboss Marshalling编码器MarshallingEncoder
     */
    public static MarshallingEncoder buildMarshallingEncoder() {
        final MarshallerFactory marshallerFactory = Marshalling.getProvidedMarshallerFactory("serial");
```

```
            final MarshallingConfiguration configuration = new MarshallingConfiguration();
            configuration.setVersion(5);
            MarshallerProvider provider = new DefaultMarshallerProvider(marshallerFactory, configuration);
            MarshallingEncoder encoder = new MarshallingEncoder(provider);
            return encoder;
        }

}
```