**序列化和反序列化**

**摘要**

　　序列化和反序列化几乎是工程师们每天都要面对的事情，但是要精确掌握这两个概念并不容易：一方面，它们往往作为框架的一部分出现而湮没在框架之中；另一方面，它们会以其他更容易理解的概念出现，例如加密、持久化。然而，序列化和反序列化的选型却是系统设计或重构一个重要的环节，在分布式、大数据量系统设计里面更为显著。恰当的序列化协议不仅可以提高系统的通用性、强健性、安全性、优化系统性能，而且会让系统更加易于调试、便于扩展。本文从多个角度去分析和讲解“序列化和反序列化”，并对比了当前流行的几种序列化协议，期望对读者做序列化选型有所帮助。

**简介**

　　文章作者服务于美团推荐与个性化组，该组致力于为美团用户提供每天billion级别的高质量个性化推荐以及排序服务。从Terabyte级别的用户行为数据，到Gigabyte级别的Deal/Poi数据；从对实时性要求毫秒以内的用户实时地理位置数据，到定期后台job数据，推荐与重排序系统需要多种类型的数据服务。推荐与重排序系统客户包括各种内部服务、美团客户端、美团网站。为了提供高质量的数据服务，为了实现与上下游各系统进行良好的对接，序列化和反序列化的选型往往是我们做系统设计的一个重要考虑因素。

　　本文内容按如下方式组织：

* 第一部分给出了序列化和反序列化的定义，以及其在通讯协议中所处的位置。
* 第二部分从使用者的角度探讨了序列化协议的一些特性。
* 第三部分描述在具体的实施过程中典型的序列化组件，并与数据库组建进行了类比。
* 第四部分分别讲解了目前常见的几种序列化协议的特性，应用场景，并对相关组件进行举例。
* 最后一部分，基于各种协议的特性，以及相关benchmark数据，给出了作者的技术选型建议。

**一、定义以及相关概念**

　　互联网的产生带来了机器间通讯的需求，而互联通讯的双方需要采用约定的协议，序列化和反序列化属于通讯协议的一部分。通讯协议往往采用分层模型，不同模型每层的功能定义以及颗粒度不同，例如：TCP/IP协议是一个四层协议，而OSI模型却是七层协议模型。在OSI七层协议模型中展现层（Presentation Layer）的主要功能是把应用层的对象转换成一段连续的二进制串，或者反过来，把二进制串转换成应用层的对象--这两个功能就是序列化和反序列化。一般而言，TCP/IP协议的应用层对应与OSI七层协议模型的应用层，展示层和会话层，所以序列化协议属于TCP/IP协议应用层的一部分。本文对序列化协议的讲解主要基于OSI七层协议模型。

* 序列化： 将数据结构或对象转换成二进制串的过程
* 反序列化：将在序列化过程中所生成的二进制串转换成数据结构或者对象的过程

**数据结构、对象与二进制串**

　　不同的计算机语言中，数据结构，对象以及二进制串的表示方式并不相同。

　　数据结构和对象：对于类似Java这种完全面向对象的语言，工程师所操作的一切都是对象（Object），来自于类的实例化。在Java语言中最接近数据结构的概念，就是POJO（Plain Old Java Object）或者Javabean－－那些只有setter/getter方法的类。而在C++这种半面向对象的语言中，数据结构和struct对应，对象和class对应。

　　二进制串：序列化所生成的二进制串指的是存储在内存中的一块数据。C++语言具有内存操作符，所以二进制串的概念容易理解，例如，C++语言的字符串可以直接被传输层使用，因为其本质上就是以'\0'结尾的存储在内存中的二进制串。在Java语言里面，二进制串的概念容易和String混淆。实际上String 是Java的一等公民，是一种特殊对象（Object）。对于跨语言间的通讯，序列化后的数据当然不能是某种语言的特殊数据类型。二进制串在Java里面所指的是byte[]，byte是Java的8中原生数据类型之一（Primitive data types）。

**二、序列化协议特性**

　　每种序列化协议都有优点和缺点，它们在设计之初有自己独特的应用场景。在系统设计的过程中，需要考虑序列化需求的方方面面，综合对比各种序列化协议的特性，最终给出一个折衷的方案。

**通用性**

　　通用性有两个层面的意义：  
　　第一、技术层面，序列化协议是否支持跨平台、跨语言。如果不支持，在技术层面上的通用性就大大降低了。  
　　第二、流行程度，序列化和反序列化需要多方参与，很少人使用的协议往往意味着昂贵的学习成本；另一方面，流行度低的协议，往往缺乏稳定而成熟的跨语言、跨平台的公共包。

**强健性/鲁棒性**

　　以下两个方面的原因会导致协议不够强健：  
　　第一、成熟度不够，一个协议从制定到实施，到最后成熟往往是一个漫长的阶段。协议的强健性依赖于大量而全面的测试，对于致力于提供高质量服务的系统，采用处于测试阶段的序列化协议会带来很高的风险。  
　　第二、语言/平台的不公平性。为了支持跨语言、跨平台的功能，序列化协议的制定者需要做大量的工作；但是，当所支持的语言或者平台之间存在难以调和的特性的时候，协议制定者需要做一个艰难的决定--支持更多人使用的语言/平台，亦或支持更多的语言/平台而放弃某个特性。当协议的制定者决定为某种语言或平台提供更多支持的时候，对于使用者而言，协议的强健性就被牺牲了。

**可调试性/可读性**

　　序列化和反序列化的数据正确性和业务正确性的调试往往需要很长的时间，良好的调试机制会大大提高开发效率。序列化后的二进制串往往不具备人眼可读性，为了验证序列化结果的正确性，写入方不得同时撰写反序列化程序，或提供一个查询平台--这比较费时；另一方面，如果读取方未能成功实现反序列化，这将给问题查找带来了很大的挑战--难以定位是由于自身的反序列化程序的bug所导致还是由于写入方序列化后的错误数据所导致。对于跨公司间的调试，由于以下原因，问题会显得更严重：  
　　第一、支持不到位，跨公司调试在问题出现后可能得不到及时的支持，这大大延长了调试周期。  
　　第二、访问限制，调试阶段的查询平台未必对外公开，这增加了读取方的验证难度。

　　如果序列化后的数据人眼可读，这将大大提高调试效率， XML和JSON就具有人眼可读的优点。

**性能**

　　性能包括两个方面，时间复杂度和空间复杂度：  
　　第一、空间开销（Verbosity）， 序列化需要在原有的数据上加上描述字段，以为反序列化解析之用。如果序列化过程引入的额外开销过高，可能会导致过大的网络，磁盘等各方面的压力。对于海量分布式存储系统，数据量往往以TB为单位，巨大的的额外空间开销意味着高昂的成本。  
　　第二、时间开销（Complexity），复杂的序列化协议会导致较长的解析时间，这可能会使得序列化和反序列化阶段成为整个系统的瓶颈。

**可扩展性/兼容性**

　　移动互联时代，业务系统需求的更新周期变得更快，新的需求不断涌现，而老的系统还是需要继续维护。如果序列化协议具有良好的可扩展性，支持自动增加新的业务字段，而不影响老的服务，这将大大提供系统的灵活度。

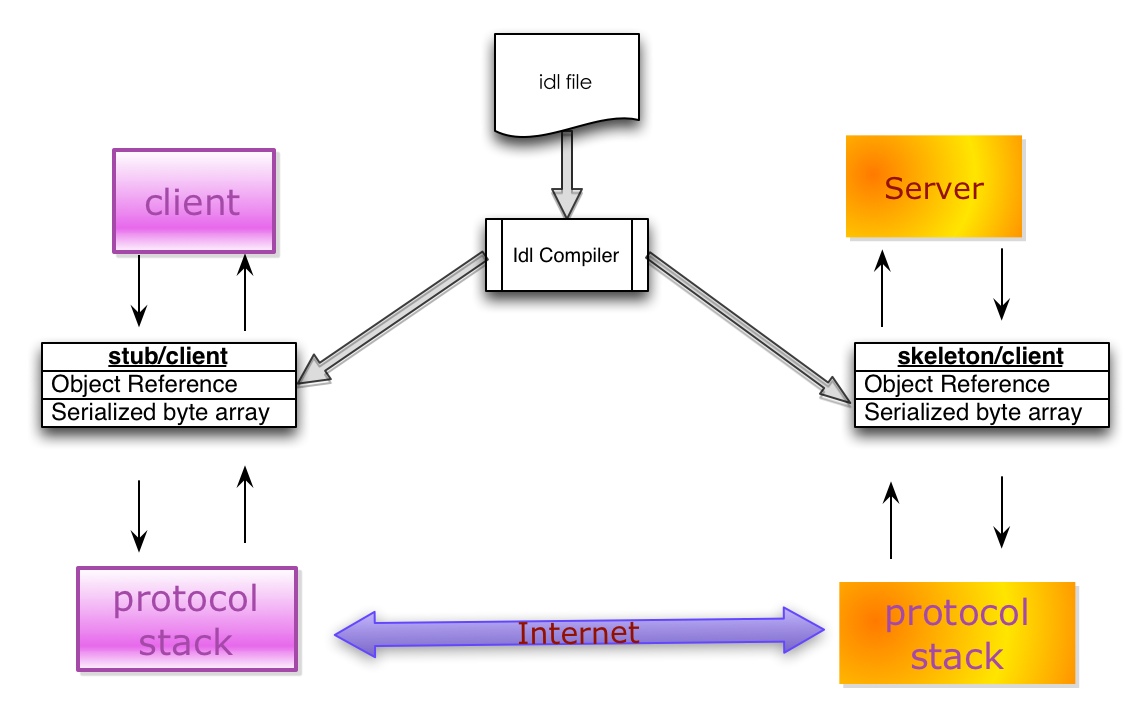
**安全性/访问限制**

　　在序列化选型的过程中，安全性的考虑往往发生在跨局域网访问的场景。当通讯发生在公司之间或者跨机房的时候，出于安全的考虑，对于跨局域网的访问往往被限制为基于HTTP/HTTPS的80和443端口。如果使用的序列化协议没有兼容而成熟的HTTP传输层框架支持，可能会导致以下三种结果之一：  
　　第一、因为访问限制而降低服务可用性。  
　　第二、被迫重新实现安全协议而导致实施成本大大提高。  
　　第三、开放更多的防火墙端口和协议访问，而牺牲安全性。

**三、序列化和反序列化的组件**

　　典型的序列化和反序列化过程往往需要如下组件：

* IDL（Interface description language）文件：参与通讯的各方需要对通讯的内容需要做相关的约定（Specifications）。为了建立一个与语言和平台无关的约定，这个约定需要采用与具体开发语言、平台无关的语言来进行描述。这种语言被称为接口描述语言（IDL），采用IDL撰写的协议约定称之为IDL文件。
* IDL Compiler：IDL文件中约定的内容为了在各语言和平台可见，需要有一个编译器，将IDL文件转换成各语言对应的动态库。
* Stub/Skeleton Lib：负责序列化和反序列化的工作代码。Stub是一段部署在分布式系统客户端的代码，一方面接收应用层的参数，并对其序列化后通过底层协议栈发送到服务端，另一方面接收服务端序列化后的结果数据，反序列化后交给客户端应用层；Skeleton部署在服务端，其功能与Stub相反，从传输层接收序列化参数，反序列化后交给服务端应用层，并将应用层的执行结果序列化后最终传送给客户端Stub。
* Client/Server：指的是应用层程序代码，他们面对的是IDL所生存的特定语言的class或struct。
* 底层协议栈和互联网：序列化之后的数据通过底层的传输层、网络层、链路层以及物理层协议转换成数字信号在互联网中传递。



**序列化组件与数据库访问组件的对比**

　　数据库访问对于很多工程师来说相对熟悉，所用到的组件也相对容易理解。下表类比了序列化过程中用到的部分组件和数据库访问组件的对应关系，以便于大家更好的把握序列化相关组件的概念。

| **序列化组件** | **数据库组件** | **说明** |
| --- | --- | --- |
| IDL | DDL | 用于建表或者模型的语言 |
| DL file | DB Schema | 表创建文件或模型文件 |
| Stub/Skeleton lib | O/R mapping | 将class和Table或者数据模型进行映射 |

**四、几种常见的序列化和反序列化协议**

　　互联网早期的序列化协议主要有COM和CORBA。

　　COM主要用于Windows平台，并没有真正实现跨平台，另外COM的序列化的原理利用了编译器中虚表，使得其学习成本巨大（想一下这个场景， 工程师需要是简单的序列化协议，但却要先掌握语言编译器）。由于序列化的数据与编译器紧耦合，扩展属性非常麻烦。

　　CORBA是早期比较好的实现了跨平台，跨语言的序列化协议。COBRA的主要问题是参与方过多带来的版本过多，版本之间兼容性较差，以及使用复杂晦涩。这些政治经济，技术实现以及早期设计不成熟的问题，最终导致COBRA的渐渐消亡。J2SE 1.3之后的版本提供了基于CORBA协议的RMI-IIOP技术，这使得Java开发者可以采用纯粹的Java语言进行CORBA的开发。

　　这里主要介绍和对比几种当下比较流行的序列化协议，包括XML、JSON、Protobuf、Thrift和Avro。

**一个例子**

　　如前所述，序列化和反序列化的出现往往晦涩而隐蔽，与其他概念之间往往相互包容。为了更好了让大家理解序列化和反序列化的相关概念在每种协议里面的具体实现，我们将一个例子穿插在各种序列化协议讲解中。在该例子中，我们希望将一个用户信息在多个系统里面进行传递；在应用层，如果采用Java语言，所面对的类对象如下所示：

class Address

{

private String city;

private String postcode;

private String street;

}

public class UserInfo

{

private Integer userid;

private String name;

private List<Address> address;

}

**XML&SOAP**

　　XML是一种常用的序列化和反序列化协议，具有跨机器，跨语言等优点。 XML历史悠久，其1.0版本早在1998年就形成标准，并被广泛使用至今。XML的最初产生目标是对互联网文档（Document）进行标记，所以它的设计理念中就包含了对于人和机器都具备可读性。 但是，当这种标记文档的设计被用来序列化对象的时候，就显得冗长而复杂（Verbose and Complex）。 XML本质上是一种描述语言，并且具有自我描述（Self-describing）的属性，所以XML自身就被用于XML序列化的IDL。 标准的XML描述格式有两种：DTD（Document Type Definition）和XSD（XML Schema Definition）。作为一种人眼可读（Human-readable）的描述语言，XML被广泛使用在配置文件中，例如O/R mapping、 Spring Bean Configuration File 等。

　　SOAP（Simple Object Access protocol） 是一种被广泛应用的，基于XML为序列化和反序列化协议的结构化消息传递协议。SOAP在互联网影响如此大，以至于我们给基于SOAP的解决方案一个特定的名称--Web service。SOAP虽然可以支持多种传输层协议，不过SOAP最常见的使用方式还是XML+HTTP。SOAP协议的主要接口描述语言（IDL）是WSDL（Web Service Description Language）。SOAP具有安全、可扩展、跨语言、跨平台并支持多种传输层协议。如果不考虑跨平台和跨语言的需求，XML的在某些语言里面具有非常简单易用的序列化使用方法，无需IDL文件和第三方编译器， 例如Java＋XStream。

**自我描述与递归**

　　SOAP是一种采用XML进行序列化和反序列化的协议，它的IDL是WSDL. 而WSDL的描述文件是XSD，而XSD自身是一种XML文件。 这里产生了一种有趣的在数学上称之为“递归”的问题，这种现象往往发生在一些具有自我属性（Self-description）的事物上。

**IDL文件举例**

　　采用WSDL描述上述用户基本信息的例子如下：

<xsd:complexType name='Address'>

<xsd:attribute name='city' type='xsd:string' />

<xsd:attribute name='postcode' type='xsd:string' />

<xsd:attribute name='street' type='xsd:string' />

</xsd:complexType>

<xsd:complexType name='UserInfo'>

<xsd:sequence>

<xsd:element name='address' type='tns:Address'/>

<xsd:element name='address1' type='tns:Address'/>

</xsd:sequence>

<xsd:attribute name='userid' type='xsd:int' />

<xsd:attribute name='name' type='xsd:string' />

</xsd:complexType>

**典型应用场景和非应用场景**

　　SOAP协议具有广泛的群众基础，基于HTTP的传输协议使得其在穿越防火墙时具有良好安全特性，XML所具有的人眼可读（Human-readable）特性使得其具有出众的可调试性，互联网带宽的日益剧增也大大弥补了其空间开销大（Verbose）的缺点。对于在公司之间传输数据量相对小或者实时性要求相对低（例如秒级别）的服务是一个好的选择。

　　由于XML的额外空间开销大，序列化之后的数据量剧增，对于数据量巨大序列持久化应用常景，这意味着巨大的内存和磁盘开销，不太适合XML。另外，XML的序列化和反序列化的空间和时间开销都比较大，对于对性能要求在ms级别的服务，不推荐使用。WSDL虽然具备了描述对象的能力，SOAP的S代表的也是simple，但是SOAP的使用绝对不简单。对于习惯于面向对象编程的用户，WSDL文件不直观。

**JSON（Javascript Object Notation）**

　　JSON起源于弱类型语言Javascript， 它的产生来自于一种称之为"Associative array"的概念，其本质是就是采用"Attribute－value"的方式来描述对象。实际上在Javascript和PHP等弱类型语言中，类的描述方式就是Associative array。JSON的如下优点，使得它快速成为最广泛使用的序列化协议之一：  
　　1、这种Associative array格式非常符合工程师对对象的理解。  
　　2、它保持了XML的人眼可读（Human-readable）的优点。  
　　3、相对于XML而言，序列化后的数据更加简洁。 来自于的以下链接的研究表明：XML所产生序列化之后文件的大小接近JSON的两倍。<http://www.codeproject.com/Articles/604720/JSON-vs-XML-Some-hard-numbers-about-verbosity>  
　　4、它具备Javascript的先天性支持，所以被广泛应用于Web browser的应用常景中，是Ajax的事实标准协议。  
　　5、与XML相比，其协议比较简单，解析速度比较快。  
　　6、松散的Associative array使得其具有良好的可扩展性和兼容性。

**IDL悖论**

　　JSON实在是太简单了，或者说太像各种语言里面的类了，所以采用JSON进行序列化不需要IDL。这实在是太神奇了，存在一种天然的序列化协议，自身就实现了跨语言和跨平台。然而事实没有那么神奇，之所以产生这种假象，来自于两个原因：  
　　第一、Associative array在弱类型语言里面就是类的概念，在PHP和Javascript里面Associative array就是其class的实际实现方式，所以在这些弱类型语言里面，JSON得到了非常良好的支持。  
　　第二、IDL的目的是撰写IDL文件，而IDL文件被IDL Compiler编译后能够产生一些代码（Stub/Skeleton），而这些代码是真正负责相应的序列化和反序列化工作的组件。 但是由于Associative array和一般语言里面的class太像了，他们之间形成了一一对应关系，这就使得我们可以采用一套标准的代码进行相应的转化。对于自身支持Associative array的弱类型语言，语言自身就具备操作JSON序列化后的数据的能力；对于Java这强类型语言，可以采用反射的方式统一解决，例如Google提供的Gson。

**典型应用场景和非应用场景**

　　JSON在很多应用场景中可以替代XML，更简洁并且解析速度更快。典型应用场景包括：  
　　1、公司之间传输数据量相对小，实时性要求相对低（例如秒级别）的服务。  
　　2、基于Web browser的Ajax请求。  
　　3、由于JSON具有非常强的前后兼容性，对于接口经常发生变化，并对可调式性要求高的场景，例如Mobile app与服务端的通讯。  
　　4、由于JSON的典型应用场景是JSON＋HTTP，适合跨防火墙访问。

　　总的来说，采用JSON进行序列化的额外空间开销比较大，对于大数据量服务或持久化，这意味着巨大的内存和磁盘开销，这种场景不适合。没有统一可用的IDL降低了对参与方的约束，实际操作中往往只能采用文档方式来进行约定，这可能会给调试带来一些不便，延长开发周期。 由于JSON在一些语言中的序列化和反序列化需要采用反射机制，所以在性能要求为ms级别，不建议使用。

**IDL文件举例**

　　以下是UserInfo序列化之后的一个例子：

{"userid":1,"name":"messi","address":[{"city":"北京","postcode":"1000000","street":"wangjingdonglu"}]}

**Thrift**

　　Thrift是Facebook开源提供的一个高性能，轻量级RPC服务框架，其产生正是为了满足当前大数据量、分布式、跨语言、跨平台数据通讯的需求。 但是，Thrift并不仅仅是序列化协议，而是一个RPC框架。相对于JSON和XML而言，Thrift在空间开销和解析性能上有了比较大的提升，对于对性能要求比较高的分布式系统，它是一个优秀的RPC解决方案；但是由于Thrift的序列化被嵌入到Thrift框架里面，Thrift框架本身并没有透出序列化和反序列化接口，这导致其很难和其他传输层协议共同使用（例如HTTP）。

**典型应用场景和非应用场景**

　　对于需求为高性能，分布式的RPC服务，Thrift是一个优秀的解决方案。它支持众多语言和丰富的数据类型，并对于数据字段的增删具有较强的兼容性。所以非常适用于作为公司内部的面向服务构建（SOA）的标准RPC框架。

　　不过Thrift的文档相对比较缺乏，目前使用的群众基础相对较少。另外由于其Server是基于自身的Socket服务，所以在跨防火墙访问时，安全是一个顾虑，所以在公司间进行通讯时需要谨慎。 另外Thrift序列化之后的数据是Binary数组，不具有可读性，调试代码时相对困难。最后，由于Thrift的序列化和框架紧耦合，无法支持向持久层直接读写数据，所以不适合做数据持久化序列化协议。

**IDL文件举例**

struct Address

{

1: required string city;

2: optional string postcode;

3: optional string street;

}

struct UserInfo

{

1: required string userid;

2: required i32 name;

3: optional list<Address> address;

}

**Protobuf**

　　Protobuf具备了优秀的序列化协议的所需的众多典型特征：  
　　1、标准的IDL和IDL编译器，这使得其对工程师非常友好。  
　　2、序列化数据非常简洁，紧凑，与XML相比，其序列化之后的数据量约为1/3到1/10。  
　　3、解析速度非常快，比对应的XML快约20-100倍。  
　　4、提供了非常友好的动态库，使用非常简介，反序列化只需要一行代码。

　　Protobuf是一个纯粹的展示层协议，可以和各种传输层协议一起使用；Protobuf的文档也非常完善。 但是由于Protobuf产生于Google，所以目前其仅仅支持Java、C++、Python三种语言。另外Protobuf支持的数据类型相对较少，不支持常量类型。由于其设计的理念是纯粹的展现层协议（Presentation Layer），目前并没有一个专门支持Protobuf的RPC框架。

**典型应用场景和非应用场景**

　　Protobuf具有广泛的用户基础，空间开销小以及高解析性能是其亮点，非常适合于公司内部的对性能要求高的RPC调用。由于Protobuf提供了标准的IDL以及对应的编译器，其IDL文件是参与各方的非常强的业务约束，另外，Protobuf与传输层无关，采用HTTP具有良好的跨防火墙的访问属性，所以Protobuf也适用于公司间对性能要求比较高的场景。由于其解析性能高，序列化后数据量相对少，非常适合应用层对象的持久化场景。

　　它的主要问题在于其所支持的语言相对较少，另外由于没有绑定的标准底层传输层协议，在公司间进行传输层协议的调试工作相对麻烦。

**IDL文件举例**

message Address

{

required string city=1;

optional string postcode=2;

optional string street=3;

}

message UserInfo

{

required string userid=1;

required string name=2;

repeated Address address=3;

}

**Avro**

　　Avro的产生解决了JSON的冗长和没有IDL的问题，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，JSON格式方便测试阶段的调试。 Avro支持的数据类型非常丰富，包括C++语言里面的union类型。Avro支持JSON格式的IDL和类似于Thrift和Protobuf的IDL（实验阶段），这两者之间可以互转。Schema可以在传输数据的同时发送，加上JSON的自我描述属性，这使得Avro非常适合动态类型语言。 Avro在做文件持久化的时候，一般会和Schema一起存储，所以Avro序列化文件自身具有自我描述属性，所以非常适合于做Hive、Pig和MapReduce的持久化数据格式。对于不同版本的Schema，在进行RPC调用的时候，服务端和客户端可以在握手阶段对Schema进行互相确认，大大提高了最终的数据解析速度。

**典型应用场景和非应用场景**

　　Avro解析性能高并且序列化之后的数据非常简洁，比较适合于高性能的序列化服务。

　　由于Avro目前非JSON格式的IDL处于实验阶段，而JSON格式的IDL对于习惯于静态类型语言的工程师来说不直观。

**IDL文件举例**

protocol Userservice {

record Address {

string city;

string postcode;

string street;

}

record UserInfo {

string name;

int userid;

array<Address> address = [];

}

}

　　所对应的JSON Schema格式如下：

{

"protocol" : "Userservice",

"namespace" : "org.apache.avro.ipc.specific",

"version" : "1.0.5",

"types" : [ {

"type" : "record",

"name" : "Address",

"fields" : [ {

"name" : "city",

"type" : "string"

}, {

"name" : "postcode",

"type" : "string"

}, {

"name" : "street",

"type" : "string"

} ]

}, {

"type" : "record",

"name" : "UserInfo",

"fields" : [ {

"name" : "name",

"type" : "string"

}, {

"name" : "userid",

"type" : "int"

}, {

"name" : "address",

"type" : {

"type" : "array",

"items" : "Address"

},

"default" : [ ]

} ]

} ],

"messages" : { }

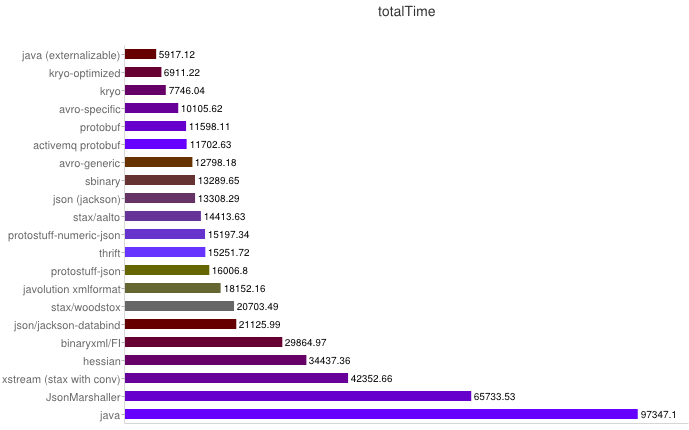
}

**五、Benchmark以及选型建议**

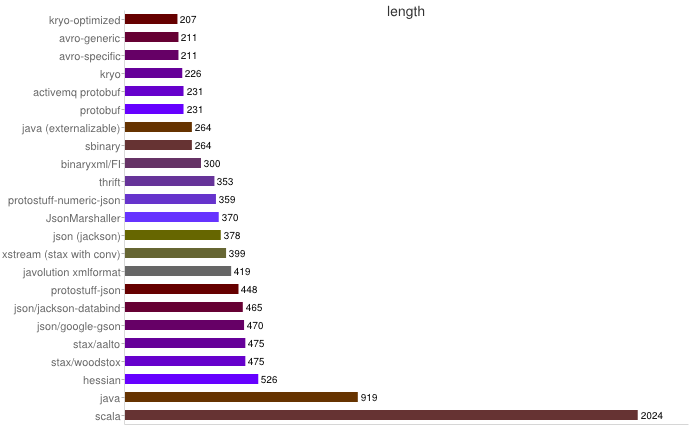
**Benchmark**

　　以下数据来自<https://code.google.com/p/thrift-protobuf-compare/wiki/Benchmarking>

**解析性能**



**序列化之空间开销**



　　从上图可得出如下结论：  
　　1、XML序列化（Xstream）无论在性能和简洁性上比较差。  
　　2、Thrift与Protobuf相比在时空开销方面都有一定的劣势。  
　　3、Protobuf和Avro在两方面表现都非常优越。

**选型建议**

　　以上描述的五种序列化和反序列化协议都各自具有相应的特点，适用于不同的场景：  
　　1、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。  
　　2、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。  
　　3、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。  
　　4、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。  
　　5、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。  
　　6、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。  
　　7、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。  
　　8、如果需要提供一个完整的RPC解决方案，Thrift是一个好的选择。  
　　9、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。

序列化应用场合

主要用于存储对象状态为另一种通用格式，比如存储为二进制、xml、json等等，把对象转换成这种格式就叫序列化，而反序列化通常是从这种格式转换回来。  
  
 使用序列化主要是因为跨平台和对象存储的需求，因为网络上只允许字符串或者二进制格式，而文件需要使用二进制流格式，如果想把一个内存中的对象存储下来就必须使用序列化转换为xml（字符串）、json（字符串）或二进制（流）

http://blog.csdn.net/abc6368765/article/details/51365838