参考：<http://shift-alt-ctrl.iteye.com/blog/2217425>

[**Avro与JAVA**](http://shift-alt-ctrl.iteye.com/blog/2217425)

**博客分类：**

* [架构](http://shift-alt-ctrl.iteye.com/category/339420)
* [JAVA](http://shift-alt-ctrl.iteye.com/category/272662)

    我们已经接触过很多序列化框架 （或者集成系统），比如protobuf、hessian、thrift等，它们各有优缺点以及各自的实用场景，Avro也是一个序列化框架，它的设计思想、编程模式都和thirft非常相似，也都是Apache的顶级项目。Avro还提供了RPC机制，可以不需要生成额外的API代码即可使用Avro来存储数据和RPC交互，“代码生成”是可选的，这一点区别于protobuf和thrift。此外Hadoop平台上的多个项目正在使用（或者支持）Avro作为数据序列化的服务。

    Avro尽管提供了RPC机制，事实上Avro的核心特性决定了它通常用在“大数据”存储场景（Mapreduce），即我们通过借助schema将数据写入到“本地文件”或者HDFS中，然后reader再根据schema去迭代获取数据条目。它的schema可以有限度的变更、调整，而且Avro能够巧妙的兼容，这种强大的可扩展性正是“文件数据”存储所必须的。

    Avro是基于schema（模式），这和protobuf、thrift没什么区别，在schema文件中（.avsc文件）中声明数据类型或者protocol（RPC接口），那么avro在read、write时将依据schema对数据进行序列化。因为有了schema，那么Avro的读、写操作将可以使用不同的平台语言。Avro的schema是JSON格式，所以编写起来也非常简单、可读性很好。目前Avro所能支持的平台语言并不是很多，其中包括JAVA、C++、Python。

    当Avro将数据写入文件时，将会把schema连同实际数据一同存储，此后reader将可以根据这个schema处理数据，如果reader使用了不同的schema，那么Avro也提供了一些兼容机制来解决这个问题。

    在RPC中使用Avro，Client和server端将会在传输数据之前，首先通过handshake交换Schema，并在Schema一致性上达成统一（me：与数据反序列化时比较两个schema判断兼容性的功能相同）。

**一、Java与Avro实例**

    接下来我们简单描述一下如何在Java平台上对Avro进行序列化、反序列化。我们以如下schema为例子(user.avsc)：

**Java代码  [收藏代码](javascript:void())**

1. {"namespace": "com.test.avro",
2. "type": "record",
3. "name": "User",
4. "fields": [
5. {"name": "name", "type": "string"},
6. {"name": "age",  "type": ["int", "null"]},
7. {"name": "email", "type": ["string", "null"]}
8. ]
9. }

    上述schema表示User类有三个field：“name”、“age”、“email”；“type”用来声明field的数据类型，比如“email”的type为“["string","null"]”，则表示类型可以为“string”或者为null。稍后我们会详细介绍。

    首先我们需要在pom.xml文件中增加如下配置：

**Java代码  [收藏代码](javascript:void())**

1. <dependency>
2. <groupId>org.apache.avro</groupId>
3. <artifactId>avro</artifactId>
4. <version>1.7.7</version>
5. </dependency>
7. <dependency>
8. <groupId>org.apache.avro</groupId>
9. <artifactId>avro-tools</artifactId>
10. <version>1.7.7</version>
11. </dependency>

   avro和avro-tools两个依赖包，是avro开发的必备的基础包。如果你的项目需要让maven来根据.avsc文件生成java代码的话，还需要增加如下avro-maven-plugin依赖（me：也可以说使用avro-tools.jar来生成，插件也是调用这个jar包，java –jar avro-tools.jar 查看帮助），否则此处是不需要的。

**Java代码  [收藏代码](javascript:void())**

1. <plugin>
2. <groupId>org.apache.avro</groupId>
3. <artifactId>avro-maven-plugin</artifactId>
4. <version>1.7.7</version>
5. <executions>
6. <execution>
7. <phase>generate-sources</phase>
8. <goals>
9. <goal>schema</goal>
10. </goals>
11. <configuration>
12. <sourceDirectory>${project.basedir}/src/main/resources/avro/</sourceDirectory>
13. <outputDirectory>${project.basedir}/src/main/java/</outputDirectory>
14. </configuration>
15. </execution>
16. </executions>
17. </plugin>
18. <plugin>
19. <groupId>org.apache.maven.plugins</groupId>
20. <artifactId>maven-compiler-plugin</artifactId>
21. <configuration>
22. <encoding>utf-8</encoding>
23. </configuration>
24. </plugin>

**1、基于“代码生成”**

    这种方式是比较常见的，即根据Avro schema生成JAVA代码，然后根据JAVA API来进行数据操作。如果你在pom.xml配置了avro-maven-plugin插件，那么只需要只需要执行：

**Java代码  [收藏代码](javascript:void())**

1. maven compile

    此后插件将会根据user.avsc文件在project指定的package目录下生成java代码。此外如果你的项目不是基于maven的，或者你不希望maven来做这件事，也可以使用如下命令生成java代码，此后只需要将代码copy到项目中即可：

**Java代码  [收藏代码](javascript:void())**

1. java -jar /path/to/avro-tools-1.7.7.jar compile schema <schema file> <code destination>
3. //例如
4. java -jar avro-tools-1.7.7.jar compile schema user.avsc ./

    序列化、反序列化Java代码样例

**Java代码  [收藏代码](javascript:void())**

1. User.Builder builder = User.newBuilder();
2. builder.setName("张三");
3. builder.setAge(30);
4. builder.setEmail("zhangsan@\*.com");
5. User user = builder.build();
7. //序列化
8. File diskFile = **new** File("/data/users.avro");
9. DatumWriter<User> userDatumWriter = **new** SpecificDatumWriter<User>(User.**class**);
10. DataFileWriter<User> dataFileWriter = **new** DataFileWriter<User>(userDatumWriter);
11. //指定schema
12. dataFileWriter.create(User.getClassSchema(), diskFile);
13. dataFileWriter.append(user);
14. dataFileWriter.fSync();//多次写入之后，可以调用fsync将数据同步写入磁盘(IO)通道
15. user.setName("李四");
16. user.setEmail("lisi@\*.com");
17. dataFileWriter.append(user);
18. dataFileWriter.close();
20. //反序列化
21. DatumReader<User> userDatumReader = **new** SpecificDatumReader<User>(User.**class**);
22. // 也可以使用DataFileStream
23. // DataFileStream<User> dataFileStream = new DataFileStream<User>(new FileInputStream(diskFile),userDatumReader);
24. DataFileReader<User> dataFileReader = **new** DataFileReader<User>(diskFile, userDatumReader);
25. User \_current = **null**;
26. **while** (dataFileReader.hasNext()) {
28. //注意:avro为了提升性能，\_current对象只会被创建一次，且每次遍历都会重用此对象
29. //next方法只是给\_current对象的各个属性赋值，而不是重新new。
30. \_current = dataFileReader.next(\_current);
31. //toString方法被重写，将获得JSON格式
32. System.out.println(\_current);
33. }
34. dataFileReader.close();

    代码很简单，描述了将User通过avro schema写入文件，我们可以通过二进制文本编辑器查看这个结果文件，会发现文件的开头部分是schema，后续是逐个user序列化的二进制结果。我们在稍后介绍encoding format，你会知道文件的数据结构。

    在reader迭代读取数据时，“\_current”对象不能在while循环之外的其他地方使用，这样是不安全的。

**原理：**代码生成时，User的Schema信息已经作为一个静态常量（me:json）写入了User.java中，同时根据schema中fields的列表严格顺序，显式的生成Fields数组，数组的index为schema中filed的声明的位置，比如name字段为0，age为1，email为2。这个严格有序性，保证了writer按照field的顺序依次编码，同时reader也按照此顺序依次解码；这也意味着开发工程者不能随意更改filed在Schema中的顺序，这个特性和protobuf、thrift都一样。

    在writer写入实际数据之前，首先把schema作为header写入文件，这个header将作为Avro数据文件的合法性校验提供帮助，如果reader和writer使用的schema无法兼容（通过此Header校验），将导致数据文件无法读取；当user对象被写入文件时，将会依次遍历user的每个filed，并根据filed的数据类型对值进行encoder，然后将bytes写入通道，编码由BinaryEncoder来实现，具体format稍后详解。

    在创建reader时会指定Schema，这个Schema称为“expected”，那么writer写入文件中还有一个Schema，这个称为“actual”，那么在reader读取数据时，究竟哪个生效呢？如果“expected”没有指定，那么将使用“actual”，否则最终的Schema将会是这两个Schema互相兼容的结果：**Avro约定，Filed顺序不能更改**，即相应的index上Filed的Type必须兼容（一致，或者兼容，比如long兼容int），如果对应index上的Filed名称不同，那么它们应该可以通过“别名”（aliase）互相兼容，即filed的name可以不一样，但是允许使用aliase来声明它的曾用名。

    那么在迭代时（next方法）也需要传递一个“\_current”对象，如果\_current与“expected”的Schema不一样（==）,即它们的Class类型不同，如果\_current为"expected"的父类，则使用\_current类型，否则尝试根据其实际的class类型与expected，通过反射机制的方式构建一个实例。next方法的迭代过程，将根据上述“兼容”后的Schema，使用BinaryDecoder逐个解析Field。

    由此可见，保持Avro Schema的设计严谨性是非常重要的，这可以避免解析过程带来的困惑。如果Schema有了巨大的变化，我们通常将数据写入新的文件，并更新解析器的API。（而不是将它们混淆在一起）

**2、非“代码生成”情况**：无需通过Schema生成java代码，开发者需要在运行时指定Schema。

**Java代码  [收藏代码](javascript:void())**

1. //user.avsc放置在“resources/avro”目录下
2. InputStream inputStream = ClassLoader.getSystemResourceAsStream("avro/user.avsc");
3. Schema schema = **new** Schema.Parser().parse(inputStream);
5. GenericRecord user = **new** GenericData.Record(schema);
6. user.put("name", "张三");
7. user.put("age", 30);
8. user.put("email","zhangsan@\*.com");
10. File diskFile = **new** File("/data/users.avro");
11. DatumWriter<GenericRecord> datumWriter = **new** GenericDatumWriter<GenericRecord>(schema);
12. DataFileWriter<GenericRecord> dataFileWriter = **new** DataFileWriter<GenericRecord>(datumWriter);
13. dataFileWriter.create(schema, diskFile);
14. dataFileWriter.append(user);
15. dataFileWriter.close();
17. DatumReader<GenericRecord> datumReader = **new** GenericDatumReader<GenericRecord>(schema);
18. DataFileReader<GenericRecord> dataFileReader = **new** DataFileReader<GenericRecord>(diskFile, datumReader);
19. GenericRecord \_current = **null**;
20. **while** (dataFileReader.hasNext()) {
21. \_current = dataFileReader.next(\_current);
22. System.out.println(user);
23. }
25. dataFileReader.close();

    这种情况下，没有生成JAVA API，那么序列化过程就需要开发者预先熟悉Schema的结构（me：fields），创建User的过程就像构建JSON字符串一样，通过put操作来“赋值”。反序列化也是一样，需要指定schema。

    GenericRecord接口提供了根据“field”名称获取值的方法：Object get(String fieldName)；不过需要声明，这内部实现并不是基于map，而是一个数组，数组和Schema声明的Fileds按照index对应。put操作根据field名称找到对应的index，然后赋值；get反之。那么在对待Schema兼容性上和“代码生成”基本一致。

**3、其他：**将Avro信息序列化到文件，这是我们在大数据存储和“数据迁移”、“分析”时常用的手段，有可能数据文件是多次append的结果（但不可能是多个线程同时append），那么开发者需要注意这一点，那么每次append时，我们需要首先seek到文件的尾部。Avro提供了内部的SeekableInput类，可以封装File。（其实最常用的办法是，每次都新建数据文件）

**Java代码  [收藏代码](javascript:void())**

1. File diskFile = **new** File("/data/users.avro");
2. **long** length = diskFile.length();
3. DatumWriter<User> userDatumWriter = **new** SpecificDatumWriter<User>(User.**class**);
4. DataFileWriter<User> dataFileWriter = **new** DataFileWriter<User>(userDatumWriter);
5. **if**(length == 0) {
6. dataFileWriter.create(user.getSchema(), diskFile);//如果是新文件，则插入Schema
7. }**else** {
8. dataFileWriter.appendTo(diskFile);//对于现有文件，则直接追加到文件的尾部
9. }
10. //....

**二、Avro数据结构**

     通过上述实例，我们已经知道了Avro Schema格式是JSON，所以编写起来非常简单，只需要了解Avro的规范即可，接下来简单介绍一些Avro的数据结构。

    Primitive Types（原生类型） ：“null”，“boolean”，“int”，“long”，“float”，“double”，“bytes”，“string”；这些数据类型和JAVA基本没有太大区别。

     复合类型：包括6种“record”，“enum”，“array”，“map”，“union”，“fixed”。

**1、record**：这个和java中的“class”有同等的意义，它支持如下属性：

        1）name：必要属性，表示record的名称，在java生成代码时作为类的名称。

        2）namespace：限定名，在java生成代码时作为package的名字。其中namespace + name最终构成record的全名。

        3）doc：可选，文档信息，备注信息。

        4）aliases：别名,一个数组

        5）fields：field列表，严格有序。每个filed又包括“name”、“doc”、“type”、“default”、“order”、“aliases”几个属性。

    其中在fields列表中每个filed应该拥有不重复的name，“type”表示field的数据类型。

    “default”很明显，用来表示field的默认值，当reader读取时，当没有此field时将会采用默认值；“order”：可选值，排序（ascending、descending、ignore），在Mapreduce集成时有用。

    “aliases”别名，JSON Array，表示此field的别名列表(me：schema迭代兼容schema时有用)。

**Java代码  [收藏代码](javascript:void())**

1. {
2. "type": "record",
3. "name": "User",
4. "namespace":"com.test.avro",
5. "aliases": ["User1","User2"],
6. "fields" : [
7. {"name": "age", "type": "int","default":10},
8. {"name": "email", "type": ["null", "string"]}
9. ]
10. }

**2、enum**：枚举类型。

**Java代码  [收藏代码](javascript:void())**

1. { "type": "enum",
2. "name": "Suit",
3. "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]
4. }

    “symbols”属性即为enum的常量值列表。值不能重复。

**3、array**：数组，和java中的数组没有区别。其items属性表示数组的item的类型。

**Java代码  [收藏代码](javascript:void())**

1. {"type":"array","items":"string"}

**4、map**：跟java中的map一样，只是key必须为string（无法声明key的类型），“values”属性声明value的类型。

**Java代码  [收藏代码](javascript:void())**

1. {"type":"map","values":"long"}

**5、union**：集合，数学意义上的“集合”（me：类型的集合。即可能为多个类型），集合中的数据不能重复。如果对“type”使用union类型，那么其default值必须和union的第一个类型匹配。

**Java代码  [收藏代码](javascript:void())**

1. {"type":["int","null"],"default":10}

**6、fixed**：表示field的值的长度为“固定值”，“size”属性表示值的字节长度。

**Java代码  [收藏代码](javascript:void())**

1. {"type":"fixed","size":16,"name":"md5"}

    record、enum、fixed、field属性，都可以声明aliases，别名--曾用名，这在schema兼容机制中非常重要。对于reader而言可以使用aliases来映射writer的schema，就像模式演变一样来处理不同的数据集。比如writer schema中有个filed命名为“Foo”，reader schema中有个filed为“Bar”并且有个别名为“Foo”,那么在reader处理数据时Bar将可以与数据中“Foo”映射并正常处理。如果schema中有多个Field重名，那么可以借助“namespace”来组合成全限定名（full namespace-qualified）。

**三、编码**

    Avro实现了两种encoding：BinaryEncoder、JsonEncoder（有兴趣可以看看其java实现），对于数据存储或者RPC通常使用BinaryEncoder，这意味着数据尺寸小而且处理更加快速；不过对于debugging或者基于web的应用，JsonEncoding通常比较便捷（即数据格式采用JSON，借助jackson）。

**Binary Encoding：**

    对于primitive类型，binary编码规则如下：“null”值将会写入0字节，“boolean”写入一个单独的字节表示0或者1；“int”和“long”采用“varint”编码技巧；“float”为4个字节，“double”为8个字节，“bytes”的编码是[long数字表示长度] + 字节数组；“string”编码类似于bytes，只是字节是UTF-8编码之后的。比较有意思的是，avro编码并不会把字段的索引号、field类型输出到流中，这一点区别于protobuf（me：因为他的编码信息已经包含了这些类型信息）。

    对于复合类型record，则不会将record的结构编码，只会依次编码filed，即record编码的结果其实就是所有fields依次编码的整合体。解析的过程将交给schema逻辑，而非将record结构信息编码到结果中。enum有一序列symbols，那么编码时只需要将enum的值所在的index作为结果即可，比如enum有["A","b","C"]三个可选值，如果其值为A，那么只需要用0来表示即可，所以这也要求开发者不能随意变更enum类型的值列表。

    array类型的编码稍有复杂，一个array可能包含多个item值，那么这些item将会被编码成一序列的blocks，每个block包含几个item，数量不等，具体一个block包含几个item，有writer的buffer决定，即writer的buffer满时将会写入一个block。所以对于reader而言，是不能预先知道array有多少个item。一个block是由一个long计数值和多个item构成，这个long计数值表示当前block中item的个数；如果一个block的long计数值为0，则表示array的结束。item的type决定它使用何种具体的编码（参见primitive）。

    对于union编码，首先union的schema声明中包含多个值比如{"type":["null","string"]}，那么编码是，首先输出值所属type在union数组的位置（起始位0），然后输出值的二进制。比如“null”则会输出“00”，如果值为“a”，则会输出“02 02 61”（02表示string在union的索引位置为1，第二个02表示string的长度为1，这上string的编码结构）。

    fixed：这个很简单，因为fixed本身不是一种数据结构，仅仅表示字节长度，那么直接生成将bytes输出即可。

**JSON Encoding：**将数据输出为json结构的字符串，key-value结构。需要注意一下，null将会输出“null”字符串，这个是由json决定。

    avro输出的数据文件，也是格式严谨的。它由header和多个file data block构成。其中header包括“magic”、“meta”、“sync”三个属性，magic通常为魔法数字：四个字节，ASII  '0' 'b' 'j' 然后紧跟一个数字1（参看源码）；meta即为schema信息，sync为同步点，目前为16个字节的随机sync标记。

    file data block包含：1）此block中包含的avro对象的个数（object counts），long型 2） 此block中data序列化的长度（block size），long型 3）序列化的对象列表，如果制定了codec，那么此对象列表是经过压缩的。 4）一个16字节的sync标记。

    之所以将数据以block的方式组织，可以非常高效的“提取”或者skipped某个block（由sync判定）而不需要反序列化它的全部内容。将“block size”、“object counts”、“sync标记”组合在一起，也可以帮助检测block是否损坏，以确保数据的完整性。（参见DataFileReader，nextBlock等API）

    目前支持的codec有“null”、“deflate”，可选“snappy”（需要手动安装和指定）。

**四、Protocol与RPC**

    Protocol就像Java中interface，定义RPC操作。一个protocol包含如下属性：

    1）protocol：protocol的名字，必要属性，对于java而言就是interface的名字。

    2）namespace：可选属性，类似于package名称。

    3）doc：备注，注释

    4）types：用于定义protocol中所涉及到的数据类型（me：也只是当类型是复合类型时定义，即相当于定义一个自定义的类型，以被方法使用）（包括record，enum，fixed，error）。error的定义和record一样，其实它在语义上也是一个record，用来表示protocol的exception信息。

    5）messages：类似于JAVA中的方法（method）列表。它有几个属性：doc、request、response、error。

其中doc还是表示注释；request为JSON数组，用于声明此message的请求参数列表，结构类似于record；response表示响应的数据类型，如果为“null”表示无需响应值；error是一个可选项，声明可能的exception类型列表，union类型的数据结构。

    代码实例，首先需要增加一个avro-ipc依赖，同时修改插件的配置（pom.xml）：

**1、pom.xml**

**Java代码  [收藏代码](javascript:void())**

1. <dependency>
2. <groupId>org.apache.avro</groupId>
3. <artifactId>avro-ipc</artifactId>
4. <version>1.7.7</version>
5. </dependency>

**Java代码  [收藏代码](javascript:void())**

1. <plugin>
2. <groupId>org.apache.avro</groupId>
3. <artifactId>avro-maven-plugin</artifactId>
4. <version>1.7.7</version>
5. <executions>
6. <execution>
7. <phase>generate-sources</phase>
8. <goals>
9. <goal>schema</goal>
10. <goal>protocol</goal>
11. <goal>idl-protocol</goal>
12. </goals>
13. <configuration>
14. <sourceDirectory>${project.basedir}/src/main/resources/avro/</sourceDirectory>
15. <outputDirectory>${project.basedir}/src/main/java/</outputDirectory>
16. </configuration>
17. </execution>
18. </executions>
19. </plugin>

**2、helloworld.avpr**：此文件用于声明protocol schema

**Java代码  [收藏代码](javascript:void())**

1. {
2. "namespace": "com.test.avro.rpc",
3. "protocol": "HelloWorld",
4. "doc": "Protocol Greetings",
6. "types": [
7. {"name": "Greeting", "type": "record", "fields": [
8. {"name": "message", "type": "string"}]},
9. {"name": "Curse", "type": "error", "fields": [
10. {"name": "message", "type": "string"}]}
11. ],
13. "messages": {
14. "hello": {
15. "doc": "Say hello.",
16. "request": [{"name": "greeting", "type": "Greeting" }],
17. "response": "Greeting",
18. "errors": ["Curse"]
19. }
20. }
21. }

    然后像普通的avro生成代码一样，执行“maven compile”即可生成protocol所需要的java代码。

**3、HelloWorldImpl.java**：我们声明了一个Portocol为HelloWorld，代码生成后，那么HelloWorld就是一个接口，我们需要继承此接口实现真正的业务逻辑。

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** HelloWorldImpl **implements** HelloWorld {
3. @Override
4. **public** Greeting hello(Greeting greeting) **throws** AvroRemoteException, Curse {
5. System.out.println(greeting.getMessage());
6. greeting.setMessage("From Server");
7. **return** greeting;
8. }
9. }

    4、测试代码

**Java代码  [收藏代码](javascript:void())**

1. **public** **static** **void** main(String[] args) **throws** Exception{
3. Server server = **new** NettyServer(**new** SpecificResponder(HelloWorld.**class**, **new** HelloWorldImpl()), **new** InetSocketAddress(8080));
4. server.start();
5. Thread.sleep(3000);
7. NettyTransceiver client = **new** NettyTransceiver(**new** InetSocketAddress(8080));
8. // client code - attach to the server and send a message
9. HelloWorld proxy = (HelloWorld) SpecificRequestor.getClient(HelloWorld.**class**, client);
10. Greeting request = **new** Greeting();
11. request.setMessage("From client");
12. Greeting response = proxy.hello(request);
13. System.out.println(response.getMessage());
15. client.close();
16. server.close();
18. }

    过程非常简单，我们看到Avro-ipc其实是依赖了netty的相关jar，其实通过netty来实现底层的IO通讯是一个不错的选择。avro-ipc还有多种方式，比如HttpServer，SaslSocketServer，DatagramServer，大家可以根据实际情况选择合适的通讯方式。 它们的内部实现基本类似，基于动态代理 + 反射机制 ，因为RPC都是“交互式”操作，如果在production环境中使用，通常开发者还需要考虑对client端、server端使用连接池机制，以提高吞吐能力，不过这些在avro-ipc中并没有提供，需要开发者自己实现；同时需要注意NettyTransceiver本身不能在多线程环境中使用，开发者需要将请求队列化，或者为每个request分配一个唯一的ID，以避免消息的错乱。

    当使用Http协议时，Avro通过request、response来交换消息，一个protocol通常由一个URL表达，Http的Content-type需要为“avro/binary”，并且client端需要通过post方式发送。

    “handshake”：“握手”的主要目的就是确保client和server端都能够持有对方的protocol声明，那么client可以正确的反序列化response，server端可以正确的反序列化request。client、server在进行实际操作之前，首先会通过handshake交换（确认）protocol schema，对于Http而言，是无状态的，那么也意味着每次请求都会进行handshake。对于有状态的通道，比如TCP，handshake只需要在connection建立之后进行一次即可，那么protocol schema将会被双方缓存起来。

**Java代码  [收藏代码](javascript:void())**

1. {
2. "type": "record",
3. "name": "HandshakeRequest", "namespace":"org.apache.avro.ipc",
4. "fields": [
5. {"name": "clientHash",
6. "type": {"type": "fixed", "name": "MD5", "size": 16}},
7. {"name": "clientProtocol", "type": ["null", "string"]},
8. {"name": "serverHash", "type": "MD5"},
9. {"name": "meta", "type": ["null", {"type": "map", "values": "bytes"}]}
10. ]
11. }
12. {
13. "type": "record",
14. "name": "HandshakeResponse", "namespace": "org.apache.avro.ipc",
15. "fields": [
16. {"name": "match",
17. "type": {"type": "enum", "name": "HandshakeMatch",
18. "symbols": ["BOTH", "CLIENT", "NONE"]}},
19. {"name": "serverProtocol",
20. "type": ["null", "string"]},
21. {"name": "serverHash",
22. "type": ["null", {"type": "fixed", "name": "MD5", "size": 16}]},
23. {"name": "meta",
24. "type": ["null", {"type": "map", "values": "bytes"}]}
25. ]
26. }

    上述即为handshake的schema，即client和server端交换protocol schema时所使用的“schema”，一个是client request，一个是server response。

    client请求时将会发送其本地protocol的hash值（clientHash），如果它已经获得过server端的hash值，此时也会传递过去（serverHash）。此后server端将会响应，如果client发送的hash值与server端计算的一致，返回结构为（match=BOTH,serverProtocol=null,serverHash=null）；如果不一致，这就意味着client和server端schema不同，那么server将会把自己的protocol反馈给client，响应结果为（match=CLIENT,serverProtocol!=null,serverHash!=null），client必须使用server返回的protocol处理此后的响应并将此protocol缓存起来。

    上述中提到需要client与server端的schema必须一致，MD5值一样；这个计算过程则需要非常的精细。如果断言2个schema在语义上是一致的？比如client schema中包含doc属性，而server端没有，但是这个doc并不影响语义上的解析，在逻辑上这两个schema仍然是“一致的”，但是MD5的计算值则不同（字面值不同）。这就引入了“解析范式”。（比较复杂，参见官网详解）

    至此，我们基本了解了avro的核心特性，以及如何使用avro实现简单的应用。我个人认为avro在RPC层面和thrift还有很大的差距，在使用thrift做RPC应用时非常简单而且是production级别可用。本人更加倾向于使用avro做数据存储和解析。