**Apache Avro™ 1.8.2 Documentation**

* [Introduction](http://avro.apache.org/docs/current/#intro)
* [Schemas](http://avro.apache.org/docs/current/#schemas)
* [Comparison with other systems](http://avro.apache.org/docs/current/#compare)

**Introduction**

Apache Avro™ is a data serialization system（me：数据序列化）.

Avro provides:

* Rich data structures.
* A compact, fast, binary data format.
* A container file, to store persistent data.
* Remote procedure call (RPC)（me:avor支持rpc远程调用的数据交互）.
* Simple integration（整合） with dynamic languages（me：与动态语言简单的整合，即与像PHP、python等动态语言的简单整合）. Code generation is not required to read or write data files nor to use or implement RPC protocols（使用avro代码生成功能不需要如何写，也不需要实现RPC协议）. Code generation as an optional optimization, only worth implementing for statically typed languages（代码生成功能只是一个可选的功能，只对静态类型的语言如java/c++/c#等有价值）.

Avro提供：

丰富的数据结构

紧凑的快速的二进制数据格式

文件容器，存储持久话的数据

支持RPC

与动态类型语言整合

Avro还提供了代码自动生成的功能，但是该功能对于avro的功能实现来说不是必须的，而且该功能仅能够使用在静态类型语言中。

**Schemas**

Avro relies on *schemas(avro依赖于schema)*. When Avro data is read, the schema used when writing it is always present. This permits each datum(数据) to be written with no per-value overheads（日常费用，开销，即写数据/序列化的成本）, making serialization both fast and small. This also facilitates（促进，推进） use with dynamic, scripting languages, since data, together with its schema, is fully self-describing（完全自描述地）.

When Avro data is stored in a file, its schema is stored with it, so that files may be processed later by any program. If the program reading the data expects a different schema this can be easily resolved, since both schemas are present.

Schmea很重要，无论读和写数据（即序列化和反序列化数据都参照schema来操作），写数据的时候会同时将schema一同写入文件等序列化载体以便被之后的程序进一步处理，读取的时候需要提供schema。

When Avro is used in RPC, the client and server exchange schemas in the connection handshake. (This can be optimized so that, for most calls, no schemas are actually transmitted.) Since both client and server both have the other's full schema, correspondence between same named fields, missing fields, extra fields, etc. can all be easily resolved.

当avro被用于RPC时，client和server在handshake连接阶段会交换schemas（这个过程是被优化的，因为对于大多数的调用，实际上并没有真实的schema传递）。因为client和server都持有了另一方的完全的schema，对于同样的字段名，丢失了字段名以及额外的字段名等等都是能够被很容易的处理的。。

Avro schemas are defined with [JSON](http://www.json.org/) . This facilitates implementation in languages that already have JSON libraries.

Avro是由json定义的

**Comparison with other systems**

Avro provides functionality similar to systems such as [Thrift](http://thrift.apache.org/), [Protocol Buffers](http://code.google.com/p/protobuf/), etc. Avro differs from these systems in the following fundamental aspects（在以下几个基本方面avro不同于其他的系统）.

* *Dynamic typing（动态类型）*: Avro does not require that code be generated（代码生成对于avro来说并不是必须的）. Data is always accompanied（相伴） by a schema that permits full processing（完全处理） of that data without code generation, static datatypes, etc. This facilitates（促进了） construction of generic data-processing systems and languages.
* *Untagged data（没有标志位数据）*: Since the schema is present when data is read, considerably less type information need be encoded with data, resulting in smaller serialization size（由于在读取数据的时候需要提供schema，需要相当小的类型信息（如类信息）被一同编码进数据（即将schema一同编码如数据），所以序列化后的数据很小）.
* *No manually-assigned field IDs（不用手动生成id字段，序列化类需要id字段）*: When a schema changes, both the old and new schema are always present when processing data, so differences may be resolved symbolically, using field names（当schema改变时，旧的和新的schema会被处理以便得出一个统一的解决）.

*Apache Avro, Avro, Apache, and the Avro and Apache logos are trademarks of The Apache Software Foundation.*

# Apache Avro™ 1.8.2 Getting Started (Java)

* [Download](http://avro.apache.org/docs/current/gettingstartedjava.html#download_install)
* [Defining a schema](http://avro.apache.org/docs/current/gettingstartedjava.html#Defining+a+schema)
* [Serializing and deserializing with code generation](http://avro.apache.org/docs/current/gettingstartedjava.html#Serializing+and+deserializing+with+code+generation)
  + [Compiling the schema](http://avro.apache.org/docs/current/gettingstartedjava.html#Compiling+the+schema)
  + [Creating Users](http://avro.apache.org/docs/current/gettingstartedjava.html#Creating+Users)
  + [Serializing](http://avro.apache.org/docs/current/gettingstartedjava.html#Serializing)
  + [Deserializing](http://avro.apache.org/docs/current/gettingstartedjava.html#Deserializing)
  + [Compiling and running the example code](http://avro.apache.org/docs/current/gettingstartedjava.html#Compiling+and+running+the+example+code)
* [Serializing and deserializing without code generation](http://avro.apache.org/docs/current/gettingstartedjava.html#Serializing+and+deserializing+without+code+generation)
  + [Creating users](http://avro.apache.org/docs/current/gettingstartedjava.html#Creating+users)
  + [Serializing](http://avro.apache.org/docs/current/gettingstartedjava.html#Serializing-N101DD)
  + [Deserializing](http://avro.apache.org/docs/current/gettingstartedjava.html#Deserializing-N10206)
  + [Compiling and running the example code](http://avro.apache.org/docs/current/gettingstartedjava.html#Compiling+and+running+the+example+code-N10248)

This is a short guide for getting started with Apache Avro™ using Java. This guide only covers using Avro for data serialization; see Patrick Hunt's [Avro RPC Quick Start](https://github.com/phunt/avro-rpc-quickstart) for a good introduction to using Avro for RPC.

## Download

Avro implementations（avro的其他语言实现） for C, C++, C#, Java, PHP, Python, and Ruby can be downloaded from the [Apache Avro™ Releases](http://avro.apache.org/releases.html) page. This guide uses Avro 1.8.2, the latest version at the time of writing. For the examples in this guide, download avro-1.8.2.jar and avro-tools-1.8.2.jar. The Avro Java implementation also depends on the [Jackson](http://jackson.codehaus.org/) JSON library（使用了JSON库jackon）. From the Jackson [download page](http://wiki.fasterxml.com/JacksonDownload), download the core-asl and mapper-asl jars. Add avro-1.8.2.jar and the Jackson jars to your project's classpath (avro-tools will be used for code generation(avro-tools是被用来自动代码生成的，如果不需要此功能则可以不加载此jar包).

Alternatively, if you are using Maven, add the following dependency to your POM:

<dependency>

<groupId>org.apache.avro</groupId>

<artifactId>avro</artifactId>

<version>1.8.2</version>

</dependency>

As well as the Avro Maven plugin (for performing code generation):

<plugin>

<groupId>org.apache.avro</groupId>

<artifactId>avro-maven-plugin</artifactId>

<version>1.8.2</version>

<executions>

<execution>

<phase>generate-sources</phase>

<goals>

<goal>schema</goal>

</goals>

<configuration>

<sourceDirectory>${project.basedir}/src/main/avro/</sourceDirectory>

<outputDirectory>${project.basedir}/src/main/java/</outputDirectory>

</configuration>

</execution>

</executions>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.6</source>

<target>1.6</target>

</configuration>

</plugin>

You may also build the required Avro jars from source. Building Avro is beyond the scope of this guide; see the [Build Documentation](https://cwiki.apache.org/AVRO/Build+Documentation) page in the wiki for more information.

添加依赖：org.apache.avro

如果需要代码自动生成功能，添加插件maven-compiler-plugin

## Defining a schema

Avro schemas are defined using JSON. Schemas are composed of [primitive types](http://avro.apache.org/docs/current/spec.html#schema_primitive)（主类型） (null, boolean, int, long, float, double, bytes, and string) and [complex types](http://avro.apache.org/docs/current/spec.html#schema_complex)（复合类型） (record-对应java里的类, enum, array, map, union-union复合类型使用数组来实现，数组元素是多个类型, and fixed). You can learn more about Avro schemas and types from the specification, but for now let's start with a simple schema example, user.avsc:

{"namespace": "example.avro",

"type": "record",

"name": "User",

"fields": [

{"name": "name", "type": "string"},

{"name": "favorite\_number", "type": ["int", "null"]},

{"name": "favorite\_color", "type": ["string", "null"]}

]

}

This schema defines a record representing a hypothetical（假定的） user. (Note that a schema file can only contain a single schema definition（注意，一个schema文件只能包含一个单独的shema定义）.) At minimum, a record definition must include its type ("type": "record"), a name ("name": "User"), and fields（至少，一个记录定义必须包含一个类型、名字和字段）, in this case name, favorite\_number, and favorite\_color. We also define a namespace ("namespace": "example.avro"), which together with the name attribute（命名空间和名字属性两者定义了这个schema的全名） defines the "full name" of the schema (example.avro.User in this case).

Fields are defined via an array of objects, each of which defines a name and type (other attributes are optional, see the [record specification](http://avro.apache.org/docs/current/spec.html#schema_record) for more details). The type attribute of a field is another schema object, which can be either a primitive or complex type. For example, the name field of our User schema is the primitive type string, whereas the favorite\_number and favorite\_color fields are both unions, represented by JSON arrays（me：unions是被json数组来定义的，并非一个可写的类型，仅仅代表了可以使用列表中列出的多种类型中的一个）. unions are a complex type that can be any of the types listed in the array; e.g., favorite\_number can either be an int or null, essentially making it an optional field（本质上使得它成为一个可选的字段）.

注意：

1. Schema是基于json的
2. 一个shema文件只能定义一个单独的schema（json的object在这里叫做schema object）
3. 一个记录record定义必须包含名字、类型和字段属性
4. 字段是由数组来定义的，每一个字段定义了名字和类型以及一些其他的可选信息
5. 也可以定义命名空间（对应java中的包名），它与名字一起定义了一个schema的全名
6. 字段是由数组来定义的（每一个定义了名字和类型，其他属性是可选的）
7. Unions并不是一个这里定义的一个可写的数据类型，它代表的是一种概念，指定是能够使列表中列出的多种类型中的一个
8. 主类型：null, boolean, int, long, float, double, bytes, and string
9. 复合类型： record、enum, array, map, union, and fixed

## Serializing and deserializing with code generation

### Compiling the schema

Code generation allows us to automatically create classes based on our previously-defined schema. Once we have defined the relevant classes, there is no need to use the schema directly in our programs（me：因为自动生成的类里面包含了该schema，字段SCHEMA$）. We use the avro-tools jar to generate code as follows(在配置好的maven项目中也可以对项目编译从而自动生成代码):

java -jar /path/to/avro-tools-1.8.2.jar compile schema <schema file> <destination>

This will generate the appropriate source files in a package based on the schema's namespace in the provided destination folder. For instance, to generate a User class in package example.avro from the schema defined above, run

java -jar /path/to/avro-tools-1.8.2.jar compile schema user.avsc .

Note that if you using the Avro Maven plugin, there is no need to manually invoke the schema compiler; the plugin automatically performs code generation on any .avsc files present in the configured source directory（me：如果安装了插件后，在compile的时候会根据schema自动生成类）.

两种方法根据schema自动生成类：

1. 下载包avro-tools，手动执行java -jar /path/to/avro-tools.jar compile /path/to/schema

2） 安装必要的插件，配置avsc文件的目录和java的src目录，这样在使用构建工具构建的时候回自动根据schema生成对应的类

### Creating Users

Now that we've completed the code generation, let's create some Users, serialize them to a data file on disk, and then read back the file and deserialize the User objects.

First let's create some Users and set their fields.

User user1 = new User();

user1.setName("Alyssa");

user1.setFavoriteNumber(256);

// Leave favorite color null

// Alternate constructor

User user2 = new User("Ben", 7, "red");

// Construct via builder

User user3 = User.newBuilder()

.setName("Charlie")

.setFavoriteColor("blue")

.setFavoriteNumber(null)

.build();

As shown in this example, Avro objects can be created either by invoking a constructor directly or by using a builder. Unlike constructors, builders will automatically set any default values specified in the schema（使用builder的话会自动的为字段赋值默认值，这些默认值都是在schema中给出的）. Additionally, builders validate the data as it set（builder会对值进行有效性检验）, whereas objects constructed directly will not cause an error until the object is serialized. However, using constructors directly generally offers better performance, as builders create a copy of the datastructure before it is written.

Note that we do not set user1's favorite color. Since that record is of type ["string", "null"], we can either set it to a string or leave it null; it is essentially optional. Similarly, we set user3's favorite number to null (using a builder requires setting all fields, even if they are null).

Key Notes:

1. 可以调用构造方法或者使用构建者builder来得到对象
2. Builder是一个内部的静态类
3. 使用构建者的话会首先使用默认值为对象的字段赋值，这些默认值都是在schema中指定的
4. 而且，使用构建者模式还会对传入的值进行有效性检验
5. 但是，直接使用构造方法的话性能高于使用构建者，因为构建者会先创建一份数据结构的拷贝

### Serializing

Now let's serialize our Users to disk.

// Serialize user1, user2 and user3 to disk

DatumWriter<User> userDatumWriter = new SpecificDatumWriter<User>(User.class);

DataFileWriter<User> dataFileWriter = new DataFileWriter<User>(userDatumWriter);

dataFileWriter.create(user1.getSchema(), new File("users.avro"));

dataFileWriter.append(user1);

dataFileWriter.append(user2);

dataFileWriter.append(user3);

dataFileWriter.close();

We create a DatumWriter, which converts Java objects into an in-memory serialized format（DatumWriter可以将java的对象转换为内存序列化的形式）. The SpecificDatumWriter class is used with generated classes and extracts the schema from the specified generated type（SpecificDatumWriter 类和自动生成的类一起使用，能够从定义好的生成类型中抽取出schema）.

Next we create a DataFileWriter, which writes the serialized records, as well as the schema, to the file specified（DataFileWriter向指定的文件中写入序列化的records-对象和它的schema） in the dataFileWriter.create call. We write our users to the file via calls to the dataFileWriter.append method. When we are done writing, we close the data file.

Key Notes:

1. DatumWriter
2. DataFileWriter

### Deserializing

Finally, let's deserialize the data file we just created.

// Deserialize Users from disk

DatumReader<User> userDatumReader = new SpecificDatumReader<User>(User.class);

DataFileReader<User> dataFileReader = new DataFileReader<User>(file, userDatumReader);

User user = null;

while (dataFileReader.hasNext()) {

// Reuse user object by passing it to next(). This saves us from

// allocating and garbage collecting many objects for files with

// many items.

user = dataFileReader.next(user);

System.out.println(user);

}

This snippet will output:

{"name": "Alyssa", "favorite\_number": 256, "favorite\_color": null}

{"name": "Ben", "favorite\_number": 7, "favorite\_color": "red"}

{"name": "Charlie", "favorite\_number": null, "favorite\_color": "blue"}

Deserializing is very similar to serializing. We create a SpecificDatumReader, analogous to the SpecificDatumWriter we used in serialization, which converts in-memory serialized items into instances of our generated class, in this case User. We pass theDatumReader and the previously created File to a DataFileReader, analogous to the DataFileWriter, which reads the data file on disk.

Next we use the DataFileReader to iterate through the serialized Users and print the deserialized object to stdout. Note how we perform the iteration: we create a single User object which we store the current deserialized user in, and pass this record object to every call of dataFileReader.next. This is a performance optimization that allows the DataFileReader to reuse the same User object rather than allocating a new User for every iteration, which can be very expensive in terms of object allocation and garbage collection if we deserialize a large data file. While this technique is the standard way to iterate through a data file, it's also possible to use for (User user : dataFileReader) if performance is not a concern.

Key Notes:

1. theDatumReader
2. DataFileReader

### Compiling and running the example code

This example code is included as a Maven project in the examples/java-example directory in the Avro docs. From this directory, execute the following commands to build and run the example:

$ mvn compile # includes code generation via Avro Maven plugin

$ mvn -q exec:java -Dexec.mainClass=example.SpecificMain

## Serializing and deserializing without code generation

Data in Avro is always stored with its corresponding schema, meaning we can always read a serialized item regardless of whether we know the schema ahead of time. This allows us to perform serialization and deserialization without code generation.

Let's go over the same example as in the previous section, but without using code generation: we'll create some users, serialize them to a data file on disk, and then read back the file and deserialize the users objects.

### Creating users

First, we use a Parser to read our schema definition and create a Schema object.

Schema schema = new Schema.Parser().parse(new File("user.avsc"));

Using this schema, let's create some users.

GenericRecord user1 = new GenericData.Record(schema);

user1.put("name", "Alyssa");

user1.put("favorite\_number", 256);

// Leave favorite color null

GenericRecord user2 = new GenericData.Record(schema);

user2.put("name", "Ben");

user2.put("favorite\_number", 7);

user2.put("favorite\_color", "red");

Since we're not using code generation, we use GenericRecords to represent users. GenericRecord uses the schema to verify that we only specify valid fields. If we try to set a non-existent field (e.g., user1.put("favorite\_animal", "cat")), we'll get anAvroRuntimeException when we run the program(我们使用schema只检验我们定义好的字段，如果我们尝试添加一个不存在的字段的话，在运行阶段会抛出一个Avro的运行时异常).

Note that we do not set user1's favorite color. Since that record is of type ["string", "null"], we can either set it to a string or leave it null; it is essentially optional.

Key Notes:

1. Schema
2. GenericRecord

### Serializing

Now that we've created our user objects, serializing and deserializing them is almost identical to the example above which uses code generation. The main difference is that we use generic instead of specific readers and writers.

First we'll serialize our users to a data file on disk.

// Serialize user1 and user2 to disk

File file = new File("users.avro");

DatumWriter<GenericRecord> datumWriter = new GenericDatumWriter<GenericRecord>(schema);

DataFileWriter<GenericRecord> dataFileWriter = new DataFileWriter<GenericRecord>(datumWriter);

dataFileWriter.create(schema, file);

dataFileWriter.append(user1);

dataFileWriter.append(user2);

dataFileWriter.close();

We create a DatumWriter, which converts Java objects into an in-memory serialized format. Since we are not using code generation, we create a GenericDatumWriter. It requires the schema both to determine how to write the GenericRecords and to verify that all non-nullable fields are present.

As in the code generation example, we also create a DataFileWriter, which writes the serialized records, as well as the schema, to the file specified in the dataFileWriter.create call. We write our users to the file via calls to the dataFileWriter.appendmethod. When we are done writing, we close the data file.

### Deserializing

Finally, we'll deserialize the data file we just created.

// Deserialize users from disk

DatumReader<GenericRecord> datumReader = new GenericDatumReader<GenericRecord>(schema);

DataFileReader<GenericRecord> dataFileReader = new DataFileReader<GenericRecord>(file, datumReader);

GenericRecord user = null;

while (dataFileReader.hasNext()) {

// Reuse user object by passing it to next(). This saves us from

// allocating and garbage collecting many objects for files with

// many items.

user = dataFileReader.next(user);

System.out.println(user);

This outputs:

{"name": "Alyssa", "favorite\_number": 256, "favorite\_color": null}

{"name": "Ben", "favorite\_number": 7, "favorite\_color": "red"}

Deserializing is very similar to serializing. We create a GenericDatumReader, analogous to the GenericDatumWriter we used in serialization, which converts in-memory serialized items into GenericRecords. We pass the DatumReader and the previously created File to a DataFileReader, analogous to the DataFileWriter, which reads the data file on disk.

Next, we use the DataFileReader to iterate through the serialized users and print the deserialized object to stdout. Note how we perform the iteration: we create a single GenericRecord object which we store the current deserialized user in, and pass this record object to every call of dataFileReader.next. This is a performance optimization that allows the DataFileReader to reuse the same record object rather than allocating a new GenericRecord for every iteration, which can be very expensive in terms of object allocation and garbage collection if we deserialize a large data file. While this technique is the standard way to iterate through a data file, it's also possible to use for (GenericRecord user : dataFileReader) if performance is not a concern.

Key Notes：

1. 使用了同一个变量迭代，节省了对象创建内存分配和垃圾回收的开销

### Compiling and running the example code

This example code is included as a Maven project in the examples/java-example directory in the Avro docs. From this directory, execute the following commands to build and run the example:

$ mvn compile

$ mvn -q exec:java -Dexec.mainClass=example.GenericMain

# Apache Avro™ 1.8.2 Specification

* [Introduction](http://avro.apache.org/docs/current/spec.html#preamble)
* [Schema Declaration](http://avro.apache.org/docs/current/spec.html#schemas)
  + [Primitive Types](http://avro.apache.org/docs/current/spec.html#schema_primitive)
  + [Complex Types](http://avro.apache.org/docs/current/spec.html#schema_complex)
    - [Records](http://avro.apache.org/docs/current/spec.html#schema_record)
    - [Enums](http://avro.apache.org/docs/current/spec.html#Enums)
    - [Arrays](http://avro.apache.org/docs/current/spec.html#Arrays)
    - [Maps](http://avro.apache.org/docs/current/spec.html#Maps)
    - [Unions](http://avro.apache.org/docs/current/spec.html#Unions)
    - [Fixed](http://avro.apache.org/docs/current/spec.html#Fixed)
  + [Names](http://avro.apache.org/docs/current/spec.html#names)
  + [Aliases](http://avro.apache.org/docs/current/spec.html#Aliases)
* [Data Serialization](http://avro.apache.org/docs/current/spec.html#Data+Serialization)
  + [Encodings](http://avro.apache.org/docs/current/spec.html#Encodings)
  + [Binary Encoding](http://avro.apache.org/docs/current/spec.html#binary_encoding)
    - [Primitive Types](http://avro.apache.org/docs/current/spec.html#binary_encode_primitive)
    - [Complex Types](http://avro.apache.org/docs/current/spec.html#binary_encode_complex)
  + [JSON Encoding](http://avro.apache.org/docs/current/spec.html#json_encoding)
  + [Single-object encoding](http://avro.apache.org/docs/current/spec.html#single_object_encoding)
    - [Single object encoding specification](http://avro.apache.org/docs/current/spec.html#single_object_encoding_spec)
* [Sort Order](http://avro.apache.org/docs/current/spec.html#order)
* [Object Container Files](http://avro.apache.org/docs/current/spec.html#Object+Container+Files)
  + [Required Codecs](http://avro.apache.org/docs/current/spec.html#Required+Codecs)
    - [null](http://avro.apache.org/docs/current/spec.html#null)
    - [deflate](http://avro.apache.org/docs/current/spec.html#deflate)
  + [Optional Codecs](http://avro.apache.org/docs/current/spec.html#Optional+Codecs)
    - [snappy](http://avro.apache.org/docs/current/spec.html#snappy)
* [Protocol Declaration](http://avro.apache.org/docs/current/spec.html#Protocol+Declaration)
  + [Messages](http://avro.apache.org/docs/current/spec.html#Messages)
  + [Sample Protocol](http://avro.apache.org/docs/current/spec.html#Sample+Protocol)
* [Protocol Wire Format](http://avro.apache.org/docs/current/spec.html#Protocol+Wire+Format)
  + [Message Transport](http://avro.apache.org/docs/current/spec.html#Message+Transport)
    - [HTTP as Transport](http://avro.apache.org/docs/current/spec.html#HTTP+as+Transport)
  + [Message Framing](http://avro.apache.org/docs/current/spec.html#Message+Framing)
  + [Handshake](http://avro.apache.org/docs/current/spec.html#handshake)
  + [Call Format](http://avro.apache.org/docs/current/spec.html#Call+Format)
* [Schema Resolution](http://avro.apache.org/docs/current/spec.html#Schema+Resolution)
* [Parsing Canonical Form for Schemas](http://avro.apache.org/docs/current/spec.html#Parsing+Canonical+Form+for+Schemas)
  + [Transforming into Parsing Canonical Form](http://avro.apache.org/docs/current/spec.html#Transforming+into+Parsing+Canonical+Form)
  + [Schema Fingerprints](http://avro.apache.org/docs/current/spec.html#schema_fingerprints)
* [Logical Types](http://avro.apache.org/docs/current/spec.html#Logical+Types)
  + [Decimal](http://avro.apache.org/docs/current/spec.html#Decimal)
  + [Date](http://avro.apache.org/docs/current/spec.html#Date)
  + [Time (millisecond precision)](http://avro.apache.org/docs/current/spec.html#Time+%28millisecond+precision%29)
  + [Time (microsecond precision)](http://avro.apache.org/docs/current/spec.html#Time+%28microsecond+precision%29)
  + [Timestamp (millisecond precision)](http://avro.apache.org/docs/current/spec.html#Timestamp+%28millisecond+precision%29)
  + [Timestamp (microsecond precision)](http://avro.apache.org/docs/current/spec.html#Timestamp+%28microsecond+precision%29)
  + [Duration](http://avro.apache.org/docs/current/spec.html#Duration)

## Introduction

This document defines Apache Avro. It is intended to be the authoritative（权威的） specification. Implementations of Avro must adhere（遵守） to this document.

## Schema Declaration

A Schema is represented in [JSON](http://www.json.org/) by one of:

* A JSON string, naming a defined type.
* A JSON object, of the form:

{"type": "*typeName*" ...*attributes*...}

where typeName is either a primitive or derived type name（衍生类型名）, as defined below. Attributes not defined in this document are permitted as metadata, but must not affect the format of serialized data.

* A JSON array（me：union类型也是由json数组来定义的）, representing a union of embedded types.

1） 一个schema由一个json来定义，称为schema object

### Primitive Types

The set of primitive type names is:

* null: no value
* boolean: a binary value
* int: 32-bit signed integer
* long: 64-bit signed integer
* float: single precision (32-bit) IEEE 754 floating-point number
* double: double precision (64-bit) IEEE 754 floating-point number
* bytes: sequence of 8-bit unsigned bytes
* string: unicode character sequence

Primitive types have no specified attributes.

Primitive type names are also defined type names. Thus, for example, the schema "string" is equivalent to:

{"type": "string"}

### Complex Types

Avro supports six kinds of complex types: records, enums, arrays, maps, unions（me：使用json数组实现，是几种主类型的集合） and fixed.

#### Records

Records use the type name "record" and support three attributes:

* name: a JSON string providing the name of the record (required).
* namespace, a JSON string that qualifies the name;
* doc: a JSON string providing documentation to the user of this schema (optional).
* aliases: a JSON array of strings, providing alternate names for this record (optional).

**（复合类型的属性包括：namespace/name/type=record/fields/aliases/doc等，红色的为必须参数）**

Class类型数据使用关键字record来表示，支持的字段有：

0） type 类型，为record，必须

1） name 必须

2) namespace 类似包名

3） doc 该类型/字段的说明

4） fields 字段，数组格式，必须

5） aliases 该字段的别名，一般用在schema兼容问题的解决上

* fields: a JSON array, listing fields (required). Each field is a JSON object with the following attributes:
  + name: a JSON string providing the name of the field (required), and
  + doc: a JSON string describing this field for users (optional).
  + type: A JSON object defining a schema, or a JSON string naming a record definition (required).
  + default: A default value for this field, used when reading instances that lack this field (optional-可选字段，当读取这个实例的时候，如果没有提供这个字段的话使用者默认值). Permitted values depend on the field's schema type, according to the table below. Default values for union fields correspond to the first schema in the union（对于union类型的字段的默认值是第一个类型的默认值）. Default values for bytes and fixed fields are JSON strings, where Unicode code points 0-255 are mapped to unsigned 8-bit byte values 0-255.

|  |  |  |
| --- | --- | --- |
| **field default values** | | |
| **avro type** | **json type** | **example** |
| null | null | null |
| boolean | boolean | true |
| int,long | integer | 1 |
| float,double | number | 1.1 |
| bytes | string | "\u00FF" |
| string | string | "foo" |
| record | object | {"a": 1} |
| enum | string | "FOO" |
| array | array | [1] |
| map | object | {"a": 1} |
| fixed | string | "\u00ff" |

* + order: specifies how this field impacts sort ordering of this record (optional). Valid values are "ascending" (the default), "descending", or "ignore". For more details on how this is used, see the the [sort order](http://avro.apache.org/docs/current/spec.html#order) section below.
  + aliases: a JSON array of strings, providing alternate names for this field (optional).

For example, a linked-list of 64-bit values may be defined with:

{

"type": "record",

"name": "LongList",

"aliases": ["LinkedLongs"], // old name for this

"fields" : [

{"name": "value", "type": "long"}, // each element has a long

{"name": "next", "type": ["null", "LongList"]} // optional next element

]

}

一个fields由json的array来实现，每个field包含的属性：name\type\default\aliases\order\doc,其中红色是必须的属性：

* 1. 对于union类型的字段，默认值为第一个类型的默认值
  2. Order字段影响这个record中字段的顺序，默认升序
  3. Aliases：由一个json数组来表示，代表了这个字段的别名，用于在schema兼容的时候

#### Enums

Enums use the type name "enum" and support the following attributes:

* name: a JSON string providing the name of the enum (required).
* namespace, a JSON string that qualifies the name;
* aliases: a JSON array of strings, providing alternate names for this enum (optional).
* doc: a JSON string providing documentation to the user of this schema (optional).
* symbols: a JSON array, listing symbols, as JSON strings (required). All symbols in an enum must be unique; duplicates are prohibited（禁止）. Every symbol must match the regular expression [A-Za-z\_][A-Za-z0-9\_]\* (the same requirement as for [names](http://avro.apache.org/docs/current/spec.html#names)).

For example, playing card suits might be defined with:

{ "type": "enum",

"name": "Suit",

"symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]

}

Enum类型使用关键字enum，支持的属性有：

1. name 必须
2. namespace 类似包名，同record
3. aliases 别名，可选属性
4. doc 该字段的注释
5. symbols 列示枚举类型中的各个元素的名字，必须

#### Arrays

Arrays use the type name "array" and support a single attribute:

* items: the schema of the array's items(定义的是这个数组中元素的类型).

For example, an array of strings is declared with:

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

Array类型使用关键字array，支持的字段有：

1. type 必须，为array
2. items 指定该数组中各个元素的类型

这类复合类型的定义与主类型的不同：

{"name": "nickName","type":{"type":"array","items":"string"}}

#### Maps

Maps use the type name "map" and support one attribute:

* values: the schema of the map's values.

Map keys are assumed to be strings.

For example, a map from string to long is declared with:

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

Map类型使用关键字map,支持的属性有：

1. type 必须，且值为map
2. values 指定的是这个map的值的类型，key是固定的，为string

{"name":"score","type":{"type":"map","values":"long"}},

#### Unions

Unions, as mentioned above, are represented using JSON arrays. For example, ["null", "string"] declares a schema which may be either a null or string.

(Note that when a [default value](http://avro.apache.org/docs/current/spec.html#schema_record) is specified for a record field whose type is a union, the type of the default value must match the first element of the union. Thus, for unions containing "null", the "null" is usually listed first, since the default value of such unions is typically null.)

Unions may not contain more than one schema with the same type, except for the named types record, fixed and enum. For example, unions containing two array types or two map types are not permitted, but two types with different names are permitted. (Names permit efficient resolution when reading and writing unions.)

Unions may not immediately contain other unions.

Unions是用json数组的形式给出，表示的是这个字段支持多种类型，如果作为一个record的字段类型时，没有提供默认值的时候使用的是第一个类型的默认值，所以一般union类型的第一个类型为null

#### Fixed

Fixed uses the type name "fixed" and supports two attributes:

* name: a string naming this fixed (required).
* namespace, a string that qualifies the name;
* aliases: a JSON array of strings, providing alternate names for this enum (optional).
* size: an integer, specifying the number of bytes per value (required).

For example, 16-byte quantity may be declared with:

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

Fixed类型使用关键字fixed，支持的属性有：

1. name
2. namespace
3. aliases 可选
4. size 是一个整数，代表了这个fixed类型的每个值占用的字节大小，必须

fixed类型类似于record类型，定义的schema会生成一个类，但是该shema定义在字段fields中：

{"name": "md5","type":{"name":"myFixed","type": "fixed", "size": 16}}

### Names

Record, enums and fixed are named types. Each has a fullname that is composed of two parts; a name and a namespace. Equality of names is defined on the fullname（名字和命名空间，name是否有效要看这两个部分）.

The name portion of a fullname, record field names, and enum symbols must:

* start with [A-Za-z\_]
* subsequently contain only [A-Za-z0-9\_]

A namespace is a dot-separated sequence of such names（命名空间是由句号分隔的名字序列）. The empty string may also be used as a namespace to indicate the null namespace（可使用空的命名空间代表不使用命名空间）. Equality of names (including field names and enum symbols) as well as fullnames is case-sensitive（名字是大小写敏感的）.

命名法则（类似java中对变量的命名规则）：

1. 名字需要以大小写字母或者下划线开头
2. 名字中只能包含字母数字和下划线
3. 名字是大小写敏感的
4. 如果命名空间是””即空的，则代表不使用命名空间

In record, enum and fixed definitions, the fullname is determined in one of the following ways:

* A name and namespace are both specified. For example, one might use "name": "X", "namespace": "org.foo" to indicate the fullname org.foo.X.
* A fullname is specified. If the name specified contains a dot, then it is assumed to be a fullname, and any namespace also specified is ignored. For example, use "name": "org.foo.X" to indicate the fullname org.foo.X.
* A name only is specified, i.e., a name that contains no dots. In this case the namespace is taken from the most tightly enclosing schema or protocol. For example, if "name": "X" is specified, and this occurs within a field of the record definition of org.foo.Y, then the fullname is org.foo.X. If there is no enclosing namespace then the null namespace is used.

References to previously defined names are as in the latter two cases above: if they contain a dot they are a fullname, if they do not contain a dot, the namespace is the namespace of the enclosing definition.

Primitive type names have no namespace and their names may not be defined in any namespace（主类型没有命名空间）.

A schema or protocol may not contain multiple definitions of a fullname. Further, a name must be defined before it is used ("before" in the depth-first, left-to-right traversal of the JSON parse tree-沿着json解析树，深度优先从左至右, where the types attribute of a protocol is always deemed to come "before" the messages attribute-然而，一个协议的类型属性总是先于message属性.)

在reocrd,enum和fixed类型的定义中，fullname通常采用以下中的一种来定义：

1) 名字和namespace都指定，

2）仅仅指定fullname,如果指定的name中包含了句号.则这个name代表的是fullname，这个时候即使制定了namespace也会被忽略

3）仅仅指明了name但是不包含句号.,namespace取自最近的namespace定义

4）名字在使用之前要先定义

### Aliases

Named types and fields may have aliases. An implementation may optionally use aliases to map a writer's schema to the reader's. This faciliates both schema evolution as well as processing disparate datasets（完全不同的数据集）.

//==================2017901

Aliases function by re-writing the writer's schema using aliases from the reader's schema. For example, if the writer's schema was named "Foo" and the reader's schema is named "Bar" and has an alias of "Foo", then the implementation would act as though "Foo" were named "Bar" when reading. Similarly, if data was written as a record with a field named "x" and is read as a record with a field named "y" with alias "x", then the implementation would act as though "x" were named "y" when reading.

A type alias may be specified either as a fully namespace-qualified, or relative to the namespace of the name it is an alias for. For example, if a type named "a.b" has aliases of "c" and "x.y", then the fully qualified names of its aliases are "a.c" and "x.y".

## Data Serialization

Avro data is always serialized with its schema. Files that store Avro data should always also include the schema for that data in the same file. Avro-based remote procedure call (RPC) systems must also guarantee that remote recipients of data have a copy of the schema used to write that data.

Because the schema used to write data is always available when the data is read, Avro data itself is not tagged with type information. The schema is required to parse data.

In general, both serialization and deserialization proceed as a depth-first, left-to-right traversal of the schema, serializing primitive types as they are encountered.

### Encodings

Avro specifies two serialization encodings: binary and JSON. Most applications will use the binary encoding, as it is smaller and faster. But, for debugging and web-based applications, the JSON encoding may sometimes be appropriate.

### Binary Encoding

#### Primitive Types

Primitive types are encoded in binary as follows:

* null is written as zero bytes.
* a boolean is written as a single byte whose value is either 0 (false) or 1 (true).
* int and long values are written using [variable-length](http://lucene.apache.org/java/3_5_0/fileformats.html#VInt) [zig-zag](http://code.google.com/apis/protocolbuffers/docs/encoding.html#types) coding. Some examples:

|  |  |
| --- | --- |
| **value** | **hex** |
| 0 | 00 |
| -1 | 01 |
| 1 | 02 |
| -2 | 03 |
| 2 | 04 |
| ... | |
| -64 | 7f |
| 64 | 80 01 |
| ... | |

* a float is written as 4 bytes. The float is converted into a 32-bit integer using a method equivalent to [Java's floatToIntBits](http://java.sun.com/javase/6/docs/api/java/lang/Float.html#floatToIntBits%28float%29) and then encoded in little-endian format.
* a double is written as 8 bytes. The double is converted into a 64-bit integer using a method equivalent to [Java's doubleToLongBits](http://java.sun.com/javase/6/docs/api/java/lang/Double.html#doubleToLongBits%28double%29) and then encoded in little-endian format.
* bytes are encoded as a long followed by that many bytes of data.
* a string is encoded as a long followed by that many bytes of UTF-8 encoded character data.

For example, the three-character string "foo" would be encoded as the long value 3 (encoded as hex 06) followed by the UTF-8 encoding of 'f', 'o', and 'o' (the hex bytes 66 6f 6f):

06 66 6f 6f

#### Complex Types

Complex types are encoded in binary as follows:

##### Records

A record is encoded by encoding the values of its fields in the order that they are declared. In other words, a record is encoded as just the concatenation of the encodings of its fields. Field values are encoded per their schema.

For example, the record schema

{

"type": "record",

"name": "test",

"fields" : [

{"name": "a", "type": "long"},

{"name": "b", "type": "string"}

]

}

An instance of this record whose a field has value 27 (encoded as hex 36) and whose b field has value "foo" (encoded as hex bytes 06 66 6f 6f), would be encoded simply as the concatenation of these, namely the hex byte sequence:

36 06 66 6f 6f

##### Enums

An enum is encoded by a int, representing the zero-based position of the symbol in the schema.

For example, consider the enum:

{"type": "enum", "name": "Foo", "symbols": ["A", "B", "C", "D"] }

This would be encoded by an int between zero and three, with zero indicating "A", and 3 indicating "D".

##### Arrays

Arrays are encoded as a series of blocks. Each block consists of a long count value, followed by that many array items. A block with count zero indicates the end of the array. Each item is encoded per the array's item schema.

If a block's count is negative, its absolute value is used, and the count is followed immediately by a long block size indicating the number of bytes in the block. This block size permits fast skipping through data, e.g., when projecting a record to a subset of its fields.

For example, the array schema

{"type": "array", "items": "long"}

an array containing the items 3 and 27 could be encoded as the long value 2 (encoded as hex 04) followed by long values 3 and 27 (encoded as hex 06 36) terminated by zero:

04 06 36 00

The blocked representation permits one to read and write arrays larger than can be buffered in memory, since one can start writing items without knowing the full length of the array.

##### Maps

Maps are encoded as a series of blocks. Each block consists of a long count value, followed by that many key/value pairs. A block with count zero indicates the end of the map. Each item is encoded per the map's value schema.

If a block's count is negative, its absolute value is used, and the count is followed immediately by a long block size indicating the number of bytes in the block. This block size permits fast skipping through data, e.g., when projecting a record to a subset of its fields.

The blocked representation permits one to read and write maps larger than can be buffered in memory, since one can start writing items without knowing the full length of the map.

##### Unions

A union is encoded by first writing a long value indicating the zero-based position within the union of the schema of its value. The value is then encoded per the indicated schema within the union.

For example, the union schema ["null","string"] would encode:

* null as zero (the index of "null" in the union):

00

* the string "a" as one (the index of "string" in the union, encoded as hex 02), followed by the serialized string:

02 02 61

##### Fixed

Fixed instances are encoded using the number of bytes declared in the schema.

### JSON Encoding

Except for unions, the JSON encoding is the same as is used to encode [field default values](http://avro.apache.org/docs/current/spec.html#schema_record).

The value of a union is encoded in JSON as follows:

* if its type is null, then it is encoded as a JSON null;
* otherwise it is encoded as a JSON object with one name/value pair whose name is the type's name and whose value is the recursively encoded value. For Avro's named types (record, fixed or enum) the user-specified name is used, for other types the type name is used.

For example, the union schema ["null","string","Foo"], where Foo is a record name, would encode:

* null as null;
* the string "a" as {"string": "a"}; and
* a Foo instance as {"Foo": {...}}, where {...} indicates the JSON encoding of a Foo instance.

Note that a schema is still required to correctly process JSON-encoded data. For example, the JSON encoding does not distinguish between int and long, float and double, records and maps, enums and strings, etc.

### Single-object encoding

In some situations a single Avro serialized object is to be stored for a longer period of time. One very common example is storing Avro records for several weeks in an [Apache Kafka](http://kafka.apache.org/) topic.

In the period after a schema change this persistance system will contain records that have been written with different schemas. So the need arises to know which schema was used to write a record to support schema evolution correctly. In most cases the schema itself is too large to include in the message, so this binary wrapper format supports the use case more effectively.

#### Single object encoding specification

Single Avro objects are encoded as follows:

1. A two-byte marker, C3 01, to show that the message is Avro and uses this single-record format (version 1).
2. The 8-byte little-endian CRC-64-AVRO [fingerprint](http://avro.apache.org/docs/current/spec.html#schema_fingerprints) of the object's schema
3. The Avro object encoded using [Avro's binary encoding](http://avro.apache.org/docs/current/spec.html#binary_encoding)

Implementations use the 2-byte marker to determine whether a payload is Avro. This check helps avoid expensive lookups that resolve the schema from a fingerprint, when the message is not an encoded Avro payload.

## Sort Order

Avro defines a standard sort order for data. This permits data written by one system to be efficiently sorted by another system. This can be an important optimization, as sort order comparisons are sometimes the most frequent per-object operation. Note also that Avro binary-encoded data can be efficiently ordered without deserializing it to objects.

Data items may only be compared if they have identical schemas. Pairwise comparisons are implemented recursively with a depth-first, left-to-right traversal of the schema. The first mismatch encountered determines the order of the items.

Two items with the same schema are compared according to the following rules.

* null data is always equal.
* boolean data is ordered with false before true.
* int, long, float and double data is ordered by ascending numeric value.
* bytes and fixed data are compared lexicographically by unsigned 8-bit values.
* string data is compared lexicographically by Unicode code point. Note that since UTF-8 is used as the binary encoding for strings, sorting of bytes and string binary data is identical.
* array data is compared lexicographically by element.
* enum data is ordered by the symbol's position in the enum schema. For example, an enum whose symbols are ["z", "a"] would sort "z" values before "a" values.
* union data is first ordered by the branch within the union, and, within that, by the type of the branch. For example, an ["int", "string"] union would order all int values before all string values, with the ints and strings themselves ordered as defined above.
* record data is ordered lexicographically by field. If a field specifies that its order is:
  + "ascending", then the order of its values is unaltered.
  + "descending", then the order of its values is reversed.
  + "ignore", then its values are ignored when sorting.
* map data may not be compared. It is an error to attempt to compare data containing maps unless those maps are in an "order":"ignore" record field.

## Object Container Files

Avro includes a simple object container file format. A file has a schema, and all objects stored in the file must be written according to that schema, using binary encoding. Objects are stored in blocks that may be compressed. Syncronization markers are used between blocks to permit efficient splitting of files for MapReduce processing.

Files may include arbitrary user-specified metadata.

A file consists of:

* A file header, followed by
* one or more file data blocks.

A file header consists of:

* Four bytes, ASCII 'O', 'b', 'j', followed by 1.
* file metadata, including the schema.
* The 16-byte, randomly-generated sync marker for this file.

File metadata is written as if defined by the following [map](http://avro.apache.org/docs/current/spec.html#map_encoding) schema:

{"type": "map", "values": "bytes"}

All metadata properties that start with "avro." are reserved. The following file metadata properties are currently used:

* **avro.schema** contains the schema of objects stored in the file, as JSON data (required).
* **avro.codec** the name of the compression codec used to compress blocks, as a string. Implementations are required to support the following codecs: "null" and "deflate". If codec is absent, it is assumed to be "null". The codecs are described with more detail below.

A file header is thus described by the following schema:

{"type": "record", "name": "org.apache.avro.file.Header",

"fields" : [

{"name": "magic", "type": {"type": "fixed", "name": "Magic", "size": 4}},

{"name": "meta", "type": {"type": "map", "values": "bytes"}},

{"name": "sync", "type": {"type": "fixed", "name": "Sync", "size": 16}},

]

}

A file data block consists of:

* A long indicating the count of objects in this block.
* A long indicating the size in bytes of the serialized objects in the current block, after any codec is applied
* The serialized objects. If a codec is specified, this is compressed by that codec.
* The file's 16-byte sync marker.

Thus, each block's binary data can be efficiently extracted or skipped without deserializing the contents. The combination of block size, object counts, and sync markers enable detection of corrupt blocks and help ensure data integrity.

### Required Codecs

#### null

The "null" codec simply passes through data uncompressed.

#### deflate

The "deflate" codec writes the data block using the deflate algorithm as specified in [RFC 1951](http://www.isi.edu/in-notes/rfc1951.txt), and typically implemented using the zlib library. Note that this format (unlike the "zlib format" in RFC 1950) does not have a checksum.

### Optional Codecs

#### snappy

The "snappy" codec uses Google's [Snappy](http://code.google.com/p/snappy/) compression library. Each compressed block is followed by the 4-byte, big-endian CRC32 checksum of the uncompressed data in the block.

## 

## Protocol Declaration

Avro protocols describe RPC interfaces. Like schemas, they are defined with JSON text（avro协议描述了RPC的接口，就像schema一样，他们也是由json文本定义的）.

A protocol is a JSON object with the following attributes:

* protocol, a string, the name of the protocol (required);
* namespace, an optional string that qualifies the name;
* doc, an optional string describing this protocol;
* types, an optional list of definitions of named types (records, enums, fixed and errors). An error definition is just like a record definition except it uses "error" instead of "record". Note that forward references to named types are not permitted.
* messages, an optional JSON object whose keys are message names and whose values are objects whose attributes are described below. No two messages may have the same name.

The name and namespace qualification rules defined for schema objects apply to protocols as well.

Protocol是一个json的对象，包含以下属性：

1. protocol，字符串类型，定义这个protocol的名字，必须
2. namespace,可选的符合命名规则的字符串
3. doc，描述此protocol的可选择的文本
4. types：可选择的类型定义的list（如record、enum、fixed和errors）。一个error定义就像一个record定义除了使用error的名字。注意，这里不支持前向引用的类型
5. messages，可选择的json对象，对象的key表示message的名字，值是一个对象。这个对象的数显按照下面的要求定义好，不能出现两个同名的message。
6. 针对schema的name和namespace的规则也适用于protocol

### Messages

A message has attributes:

* a doc, an optional description of the message,
* a request, a list of named, typed parameter schemas (this has the same form as the fields of a record declaration);
* a response schema;
* an optional union of declared error schemas. The effective union has "string" prepended to the declared union, to permit transmission of undeclared "system" errors. For example, if the declared error union is ["AccessError"], then the effective union is ["string", "AccessError"]. When no errors are declared, the effective error union is ["string"]. Errors are serialized using the effective union; however, a protocol's JSON declaration contains only the declared union.
* an optional one-way boolean parameter.

A request parameter list is processed equivalently to an anonymous record. Since record field lists may vary between reader and writer, request parameters may also differ between the caller and responder, and such differences are resolved in the same manner as record field differences.

The one-way parameter may only be true when the response type is "null" and no errors are listed.

一个message包含的属性有：

1. doc，本message的描述，可选
2. request定义，包含一系列被命名和设置了类型的参数schema的集合（这部分与fields的定义相同）
3. response,一个schema
4. 定义好的schema的error的union，这里默认将string填加到这个union里面以允许传递未被定义的系统error。例如，如果error的union的定义为["AccessError"]，那么实际的union是["string", "AccessError"]，当没有error被定义时，实际的union是["string"]。Errors使用这个union来序列化。然而，protocol的json定义值包含定义的union
5. Boolean类型参数，可选

request的参数列表会被当做匿名的record处理。由于读和写的record的fields可能不同，request的参数在caller和responder之间也可能不同。这些不同也会按照处理field的不同的处理规则来处理（即使用别名来处理）。

如果resonse的类型是null，并且没有错误产生，则one-way parameter该参数也许只能为true。

注意：protocol的文件名以.avpr后缀

### Sample Protocol

For example, one may define a simple HelloWorld protocol with:

{

"namespace": "com.acme",

"protocol": "HelloWorld",

"doc": "Protocol Greetings",

"types": [

{"name": "Greeting", "type": "record", "fields": [

{"name": "message", "type": "string"}]},

{"name": "Curse", "type": "error", "fields": [

{"name": "message", "type": "string"}]}

],

"messages": {

"hello": {

"doc": "Say hello.",

"request": [{"name": "greeting", "type": "Greeting" }],

"response": "Greeting",

"errors": ["Curse"]

}

}

}

## Protocol Wire Format

### Message Transport

Messages may be transmitted via different transport mechanisms（messages也许通过不同的传输机制传递）.

To the transport, a message is an opaque byte sequence（对于传输来说，messages是一个不透明的字节序列）.

A transport is a system that supports:

* **transmission of request messages**
* **receipt of corresponding response messages**

Servers may send a response message back to the client corresponding to a request message. The mechanism of correspondance is transport-specific. For example, in HTTP it is implicit, since HTTP directly supports requests and responses. But a transport that multiplexes many client threads over a single socket would need to tag messages with unique identifiers.

Transports may be either stateless or stateful. In a stateless transport, messaging assumes no established connection state, while stateful transports establish connections that may be used for multiple messages. This distinction is discussed further in the [handshake](http://avro.apache.org/docs/current/spec.html#handshake) section below.

#### HTTP as Transport

When [HTTP](http://www.w3.org/Protocols/rfc2616/rfc2616.html) is used as a transport, each Avro message exchange is an HTTP request/response pair. All messages of an Avro protocol should share a single URL at an HTTP server. Other protocols may also use that URL. Both normal and error Avro response messages should use the 200 (OK) response code. The chunked encoding may be used for requests and responses, but, regardless the Avro request and response are the entire content of an HTTP request and response. The HTTP Content-Type of requests and responses should be specified as "avro/binary". Requests should be made using the POST method.

HTTP is used by Avro as a stateless transport.

### Message Framing

Avro messages are framed as a list of buffers.

Framing is a layer between messages and the transport. It exists to optimize certain operations（framing的存在是为了优化某些性能）.

The format of framed message data is:

* a series of buffers, where each buffer consists of:
  + a four-byte, big-endian buffer length, followed by
  + that many bytes of buffer data.
* A message is always terminated by a zero-length buffer.

Framing is transparent to request and response message formats (described below). Any message may be presented as a single or multiple buffers.

Framing can permit readers to more efficiently get different buffers from different sources and for writers to more efficiently store different buffers to different destinations. In particular, it can reduce the number of times large binary objects are copied. For example, if an RPC parameter consists of a megabyte of file data, that data can be copied directly to a socket from a file descriptor, and, on the other end, it could be written directly to a file descriptor, never entering user space.

A simple, recommended, framing policy is for writers to create a new segment whenever a single binary object is written that is larger than a normal output buffer. Small objects are then appended in buffers, while larger objects are written as their own buffers. When a reader then tries to read a large object the runtime can hand it an entire buffer directly, without having to copy it.

### Handshake

The purpose of the handshake is to ensure that the client and the server have each other's protocol definition, so that the client can correctly deserialize responses, and the server can correctly deserialize requests. Both clients and servers should maintain a cache of recently seen protocols, so that, in most cases, a handshake will be completed without extra round-trip network exchanges or the transmission of full protocol text.

RPC requests and responses may not be processed until a handshake has been completed. With a stateless transport, all requests and responses are prefixed by handshakes. With a stateful transport, handshakes are only attached to requests and responses until a successful handshake response has been returned over a connection. After this, request and response payloads are sent without handshakes for the lifetime of that connection.

The handshake process uses the following record schemas:

{

"type": "record",

"name": "HandshakeRequest", "namespace":"org.apache.avro.ipc",

"fields": [

{"name": "clientHash",

"type": {"type": "fixed", "name": "MD5", "size": 16}},

{"name": "clientProtocol", "type": ["null", "string"]},

{"name": "serverHash", "type": "MD5"},

{"name": "meta", "type": ["null", {"type": "map", "values": "bytes"}]}

]

}

{

"type": "record",

"name": "HandshakeResponse", "namespace": "org.apache.avro.ipc",

"fields": [

{"name": "match",

"type": {"type": "enum", "name": "HandshakeMatch",

"symbols": ["BOTH", "CLIENT", "NONE"]}},

{"name": "serverProtocol",

"type": ["null", "string"]},

{"name": "serverHash",

"type": ["null", {"type": "fixed", "name": "MD5", "size": 16}]},

{"name": "meta",

"type": ["null", {"type": "map", "values": "bytes"}]}

]

}

* A client first prefixes each request with a HandshakeRequest containing just the hash of its protocol and of the server's protocol (clientHash!=null, clientProtocol=null, serverHash!=null), where the hashes are 128-bit MD5 hashes of the JSON protocol text. If a client has never connected to a given server, it sends its hash as a guess of the server's hash, otherwise it sends the hash that it previously obtained from this server.
* The server responds with a HandshakeResponse containing one of:
  + match=BOTH, serverProtocol=null, serverHash=null if the client sent the valid hash of the server's protocol and the server knows what protocol corresponds to the client's hash. In this case, the request is complete and the response data immediately follows the HandshakeResponse.
  + match=CLIENT, serverProtocol!=null, serverHash!=null if the server has previously seen the client's protocol, but the client sent an incorrect hash of the server's protocol. The request is complete and the response data immediately follows the HandshakeResponse. The client must use the returned protocol to process the response and should also cache that protocol and its hash for future interactions with this server.
  + match=NONE if the server has not previously seen the client's protocol. The serverHash and serverProtocol may also be non-null if the server's protocol hash was incorrect.

In this case the client must then re-submit its request with its protocol text (clientHash!=null, clientProtocol!=null, serverHash!=null) and the server should respond with a successful match (match=BOTH, serverProtocol=null, serverHash=null) as above.

The meta field is reserved for future handshake enhancements.

### Call Format

A call consists of a request message paired with its resulting response or error message. Requests and responses contain extensible metadata, and both kinds of messages are framed as described above.

The format of a call request is:

* request metadata, a map with values of type bytes
* the message name, an Avro string, followed by
* the message parameters. Parameters are serialized according to the message's request declaration.

When the empty string is used as a message name a server should ignore the parameters and return an empty response. A client may use this to ping a server or to perform a handshake without sending a protocol message.

When a message is declared one-way and a stateful connection has been established by a successful handshake response, no response data is sent. Otherwise the format of the call response is:

* response metadata, a map with values of type bytes
* a one-byte error flag boolean, followed by either:
  + if the error flag is false, the message response, serialized per the message's response schema.
  + if the error flag is true, the error, serialized per the message's effective error union schema.

//====================20170904

## Schema Resolution

A reader of Avro data, whether from an RPC or a file, can always parse that data because its schema is provided. But that schema may not be exactly the schema that was expected. For example, if the data was written with a different version of the software than it is read, then records may have had fields added or removed. This section specifies how such schema differences should be resolved.

We call the schema used to write the data as the writer's schema, and the schema that the application expects the reader's schema. Differences between these should be resolved as follows:

* It is an error if the two schemas do not match.

To match, one of the following must hold:

* + both schemas are arrays whose item types match
  + both schemas are maps whose value types match
  + both schemas are enums whose names match
  + both schemas are fixed whose sizes and names match
  + both schemas are records with the same name
  + either schema is a union
  + both schemas have same primitive type
  + the writer's schema may be promoted to the reader's as follows:
    - int is promotable to long, float, or double
    - long is promotable to float or double
    - float is promotable to double
    - string is promotable to bytes
    - bytes is promotable to string
* **if both are records:**
  + the ordering of fields may be different: fields are matched by name.
  + schemas for fields with the same name in both records are resolved recursively.
  + if the writer's record contains a field with a name not present in the reader's record, the writer's value for that field is ignored.
  + if the reader's record schema has a field that contains a default value, and writer's schema does not have a field with the same name, then the reader should use the default value from its field.
  + if the reader's record schema has a field with no default value, and writer's schema does not have a field with the same name, an error is signalled.
* **if both are enums:**

if the writer's symbol is not present in the reader's enum, then an error is signalled.

* **if both are arrays:**

This resolution algorithm is applied recursively to the reader's and writer's array item schemas.

* **if both are maps:**

This resolution algorithm is applied recursively to the reader's and writer's value schemas.

* **if both are unions:**

The first schema in the reader's union that matches the selected writer's union schema is recursively resolved against it. if none match, an error is signalled.

* **if reader's is a union, but writer's is not**

The first schema in the reader's union that matches the writer's schema is recursively resolved against it. If none match, an error is signalled.

* **if writer's is a union, but reader's is not**

If the reader's schema matches the selected writer's schema, it is recursively resolved against it. If they do not match, an error is signalled.

A schema's "doc" fields are ignored for the purposes of schema resolution. Hence, the "doc" portion of a schema may be dropped at serialization.

## Parsing Canonical Form for Schemas

One of the defining characteristics of Avro is that a reader is assumed to have the "same" schema used by the writer of the data the reader is reading. This assumption leads to a data format that's compact and also amenable to many forms of schema evolution. However, the specification so far has not defined what it means for the reader to have the "same" schema as the writer. Does the schema need to be textually identical? Well, clearly adding or removing some whitespace to a JSON expression does not change its meaning. At the same time, reordering the fields of records clearly does change the meaning. So what does it mean for a reader to have "the same" schema as a writer?

Parsing Canonical Form is a transformation of a writer's schema that let's us define what it means for two schemas to be "the same" for the purpose of reading data written agains the schema. It is called Parsing Canonical Form because the transformations strip away parts of the schema, like "doc" attributes, that are irrelevant to readers trying to parse incoming data. It is called Canonical Form because the transformations normalize the JSON text (such as the order of attributes) in a way that eliminates unimportant differences between schemas. If the Parsing Canonical Forms of two different schemas are textually equal, then those schemas are "the same" as far as any reader is concerned, i.e., there is no serialized data that would allow a reader to distinguish data generated by a writer using one of the original schemas from data generated by a writing using the other original schema. (We sketch a proof of this property in a companion document.)

The next subsection specifies the transformations that define Parsing Canonical Form. But with a well-defined canonical form, it can be convenient to go one step further, transforming these canonical forms into simple integers ("fingerprints") that can be used to uniquely identify schemas. The subsection after next recommends some standard practices for generating such fingerprints.

### Transforming into Parsing Canonical Form

Assuming an input schema (in JSON form) that's already UTF-8 text for a valid Avro schema (including all quotes as required by JSON), the following transformations will produce its Parsing Canonical Form:

* [PRIMITIVES] Convert primitive schemas to their simple form (e.g., int instead of {"type":"int"}).
* [FULLNAMES] Replace short names with fullnames, using applicable namespaces to do so. Then eliminate namespace attributes, which are now redundant.
* [STRIP] Keep only attributes that are relevant to parsing data, which are: type, name, fields, symbols, items, values, size. Strip all others (e.g., doc and aliases).
* [ORDER] Order the appearance of fields of JSON objects as follows: name, type, fields, symbols, items, values, size. For example, if an object has type, name, and size fields, then the name field should appear first, followed by the type and then the size fields.
* [STRINGS] For all JSON string literals in the schema text, replace any escaped characters (e.g., \uXXXX escapes) with their UTF-8 equivalents.
* [INTEGERS] Eliminate quotes around and any leading zeros in front of JSON integer literals (which appear in the size attributes of fixed schemas).
* [WHITESPACE] Eliminate all whitespace in JSON outside of string literals.

### Schema Fingerprints

"[A] fingerprinting algorithm is a procedure that maps an arbitrarily large data item (such as a computer file) to a much shorter bit string, its fingerprint, that uniquely identifies the original data for all practical purposes" (quoted from [[Wikipedia](http://en.wikipedia.org/wiki/Fingerprint_(computing))]). In the Avro context, fingerprints of Parsing Canonical Form can be useful in a number of applications; for example, to cache encoder and decoder objects, to tag data items with a short substitute for the writer's full schema, and to quickly negotiate common-case schemas between readers and writers.

In designing fingerprinting algorithms, there is a fundamental trade-off between the length of the fingerprint and the probability of collisions. To help application designers find appropriate points within this trade-off space, while encouraging interoperability and ease of implementation, we recommend using one of the following three algorithms when fingerprinting Avro schemas:

* When applications can tolerate longer fingerprints, we recommend using the [SHA-256 digest algorithm](http://en.wikipedia.org/wiki/SHA-2) to generate 256-bit fingerprints of Parsing Canonical Forms. Most languages today have SHA-256 implementations in their libraries.
* At the opposite extreme, the smallest fingerprint we recommend is a 64-bit [Rabin fingerprint](http://en.wikipedia.org/wiki/Rabin_fingerprint). Below, we provide pseudo-code for this algorithm that can be easily translated into any programming language. 64-bit fingerprints should guarantee uniqueness for schema caches of up to a million entries (for such a cache, the chance of a collision is 3E-8). We don't recommend shorter fingerprints, as the chances of collisions is too great (for example, with 32-bit fingerprints, a cache with as few as 100,000 schemas has a 50% chance of having a collision).
* Between these two extremes, we recommend using the [MD5 message digest](http://en.wikipedia.org/wiki/MD5) to generate 128-bit fingerprints. These make sense only where very large numbers of schemas are being manipulated (tens of millions); otherwise, 64-bit fingerprints should be sufficient. As with SHA-256, MD5 implementations are found in most libraries today.

These fingerprints are not meant to provide any security guarantees, even the longer SHA-256-based ones. Most Avro applications should be surrounded by security measures that prevent attackers from writing random data and otherwise interfering with the consumers of schemas. We recommend that these surrounding mechanisms be used to prevent collision and pre-image attacks (i.e., "forgery") on schema fingerprints, rather than relying on the security properties of the fingerprints themselves.

Rabin fingerprints are [cyclic redundancy checks](http://en.wikipedia.org/wiki/Cyclic_redundancy_check) computed using irreducible polynomials. In the style of the Appendix of [RFC 1952](http://www.ietf.org/rfc/rfc1952.txt) (pg 10), which defines the CRC-32 algorithm, here's our definition of the 64-bit AVRO fingerprinting algorithm:

long fingerprint64(byte[] buf) {

if (FP\_TABLE == null) initFPTable();

long fp = EMPTY;

for (int i = 0; i < buf.length; i++)

fp = (fp >>> 8) ^ FP\_TABLE[(int)(fp ^ buf[i]) & 0xff];

return fp;

}

static long EMPTY = 0xc15d213aa4d7a795L;

static long[] FP\_TABLE = null;

void initFPTable() {

FP\_TABLE = new long[256];

for (int i = 0; i < 256; i++) {

long fp = i;

for (int j = 0; j < 8; j++)

fp = (fp >>> 1) ^ (EMPTY & -(fp & 1L));

FP\_TABLE[i] = fp;

}

}

Readers interested in the mathematics behind this algorithm may want to read [this book chapter.](http://www.scribd.com/fb-6001967/d/84795-Crc) (Unlike RFC-1952 and the book chapter, we prepend a single one bit to messages. We do this because CRCs ignore leading zero bits, which can be problematic. Our code prepends a one-bit by initializing fingerprints using EMPTY, rather than initializing using zero as in RFC-1952 and the book chapter.)

## Logical Types

A logical type is an Avro primitive or complex type with extra attributes to represent a derived type. The attribute logicalType must always be present for a logical type, and is a string with the name of one of the logical types listed later in this section. Other attributes may be defined for particular logical types.

A logical type is always serialized using its underlying Avro type so that values are encoded in exactly the same way as the equivalent Avro type that does not have a logicalType attribute. Language implementations may choose to represent logical types with an appropriate native type, although this is not required.

Language implementations must ignore unknown logical types when reading, and should use the underlying Avro type. If a logical type is invalid, for example a decimal with scale greater than its precision, then implementations should ignore the logical type and use the underlying Avro type.

### Decimal

The decimal logical type represents an arbitrary-precision signed decimal number of the form unscaled × 10-scale.

A decimal logical type annotates Avro bytes or fixed types. The byte array must contain the two's-complement representation of the unscaled integer value in big-endian byte order. The scale is fixed, and is specified using an attribute.

The following attributes are supported:

* scale, a JSON integer representing the scale (optional). If not specified the scale is 0.
* precision, a JSON integer representing the (maximum) precision of decimals stored in this type (required).

For example, the following schema represents decimal numbers with a maximum precision of 4 and a scale of 2:

{

"type": "bytes",

"logicalType": "decimal",

"precision": 4,

"scale": 2

}

Precision must be a positive integer greater than zero. If the underlying type is a fixed, then the precision is limited by its size. An array of length n can store at most floor(log\_10(28 × n - 1 - 1)) base-10 digits of precision.

Scale must be zero or a positive integer less than or equal to the precision.

For the purposes of schema resolution, two schemas that are decimal logical types match if their scales and precisions match.

### Date

The date logical type represents a date within the calendar, with no reference to a particular time zone or time of day.

A date logical type annotates an Avro int, where the int stores the number of days from the unix epoch, 1 January 1970 (ISO calendar).

### Time (millisecond precision)

The time-millis logical type represents a time of day, with no reference to a particular calendar, time zone or date, with a precision of one millisecond.

A time-millis logical type annotates an Avro int, where the int stores the number of milliseconds after midnight, 00:00:00.000.

### Time (microsecond precision)

The time-micros logical type represents a time of day, with no reference to a particular calendar, time zone or date, with a precision of one microsecond.

A time-micros logical type annotates an Avro long, where the long stores the number of microseconds after midnight, 00:00:00.000000.

### Timestamp (millisecond precision)

The timestamp-millis logical type represents an instant on the global timeline, independent of a particular time zone or calendar, with a precision of one millisecond.

A timestamp-millis logical type annotates an Avro long, where the long stores the number of milliseconds from the unix epoch, 1 January 1970 00:00:00.000 UTC.

### Timestamp (microsecond precision)

The timestamp-micros logical type represents an instant on the global timeline, independent of a particular time zone or calendar, with a precision of one microsecond.

A timestamp-micros logical type annotates an Avro long, where the long stores the number of microseconds from the unix epoch, 1 January 1970 00:00:00.000000 UTC.

### Duration

The duration logical type represents an amount of time defined by a number of months, days and milliseconds. This is not equivalent to a number of milliseconds, because, depending on the moment in time from which the duration is measured, the number of days in the month and number of milliseconds in a day may differ. Other standard periods such as years, quarters, hours and minutes can be expressed through these basic periods.

A duration logical type annotates Avro fixed type of size 12, which stores three little-endian unsigned integers that represent durations at different granularities of time. The first stores a number in months, the second stores a number in days, and the third stores a number in milliseconds.

Apache Avro, Avro, Apache, and the Avro and Apache logos are trademarks of The Apache Software Foundation.