# package org.apache.avro.Schema

**//Changes in Schema class – new introduced code is in brown**

public abstract class Schema extends JsonProperties {

public enum Type {

RECORD, ENUM, ARRAY, MAP, UNION, FIXED, STRING, BYTES,

INT, LONG, FLOAT, DOUBLE, BOOLEAN, **BITMAP**, NULL; //TODO OF Changes: Type.BITMAP

private String name;

private Type() { this.name = this.name().toLowerCase(); }

public String getName() { return name; }

};

**// parsing**

if (PRIMITIVES.containsKey(type)) {

result = create(PRIMITIVES.get(type));

} else if (type.equals("enum")) {

JsonNode symbolsNode = schema.get("symbols");

**JsonNode itemsNode = schema.get("items");**

**JsonNode listNode = schema.get("list");**

result = null;

if ((symbolsNode == null || !symbolsNode.isArray()) &&

(itemsNode == null || listNode == null))

throw new SchemaParseException("Enum has neither symbols nor items: "+schema);

if (symbolsNode != null) {

LockableArrayList<String> symbols = new LockableArrayList<String>();

for (JsonNode n : symbolsNode)

symbols.add(n.getTextValue());

result = new EnumSchema(name, doc, symbols);

if (name != null) names.add(result);

}

**else if (itemsNode != null && listNode != null) {**

**Schema itemsSchema = parse(itemsNode, names);//String itemsType = itemsNode.get;**

**LockableArrayList<JsonNode> list = new LockableArrayList<JsonNode>();**

**LockableArrayList<String> symbols = new LockableArrayList<String>();**

**Iterator<JsonNode> i = listNode.getElements();**

**while (i.hasNext()) {**

**JsonNode n = i.next();**

**symbols.add(n.get("name").getTextValue());**

**list.add(n);**

**}**

result = new EnumSchema(name, doc, itemsSchema, list);

if (name != null) names.add(result);

}

}

**//Changes in Enum schema, newly introduced code is in brown**

private static class EnumSchema extends NamedSchema {

private final List<String> symbols;

private final Map<String,Integer> ordinals;

**private final Map<String, JsonNode> items;**

**private final Schema itemsSchema;**

public EnumSchema(Name name, String doc,

LockableArrayList<String> symbols) {

super(Type.ENUM, name, doc);

this.symbols = symbols.lock();

this.ordinals = new HashMap<String,Integer>();

this.items = null;

this.itemsSchema = null;

int i = 0;

for (String symbol : symbols)

if (ordinals.put(validateName(symbol), i++) != null)

throw new SchemaParseException("Duplicate enum symbol: "+symbol);

}

public EnumSchema(Name name, String doc, Schema itemsSchema,

LockableArrayList<JsonNode> list) {

super(Type.ENUM, name, doc);

this.ordinals = null;

**this.itemsSchema = itemsSchema;**

this.symbols = new ArrayList<String>();

**this.items = new HashMap<String, JsonNode>();**

**for (JsonNode n : list) {**

**JsonNode defaultValue = n.get("default");**

**if (defaultValue.isObject()) {**

**IOperation operation = new Operation(defaultValue);**

**defaultValue = itemsSchema.getDefault(operation.result());**

**}**

**items.put(n.get("name").getTextValue(), defaultValue);**

**this.symbols.add(n.get("name").getTextValue());**

**}**

return;

}

public List<String> getEnumSymbols() { return symbols; }

public boolean hasEnumSymbol(String symbol) {

return ordinals.containsKey(symbol); }

public int getEnumOrdinal(String symbol) { return ordinals.get(symbol); }

**public JsonNode getEnumItem(String symbol) { return items.get(symbol); }**

**public Schema getEnumItemsSchema() { return this.itemsSchema; }**

public boolean equals(Object o) {

if (o == this) return true;

if (!(o instanceof EnumSchema)) return false;

EnumSchema that = (EnumSchema)o;

return equalCachedHash(that)

&& equalNames(that)

&& symbols.equals(that.symbols)

&& props.equals(that.props);

}

@Override int computeHash() { return super.computeHash() + symbols.hashCode(); }

void toJson(Names names, JsonGenerator gen) throws IOException {

if (writeNameRef(names, gen)) return;

gen.writeStartObject();

gen.writeStringField("type", "enum");

writeName(names, gen);

if (getDoc() != null)

gen.writeStringField("doc", getDoc());

gen.writeArrayFieldStart("symbols");

for (String symbol : symbols)

gen.writeString(symbol);

gen.writeEndArray();

writeProps(gen);

aliasesToJson(gen);

gen.writeEndObject();

}

}

**/\***

**\* New classes for Operations and Bitmap**

**\*/**

**Operation:**

public interface IOperation {

JsonNode result();

};

protected interface IOperationBuilder {

Operation build(JsonNode ops);

};

public class Operation implements IOperation {

protected String name;

protected List<IOperation> operands = null;

protected JsonNode result = null;

protected Map<String, IOperationBuilder> operations = null;

protected class OperationOrBuilder implements IOperationBuilder{

public Operation build(JsonNode ops) {

return new OperationOr (ops);

}

}

protected class OperationXorBuilder implements IOperationBuilder{

public Operation build(JsonNode ops) {

return new OperationXor (ops);

}

}

protected class OperationAndBuilder implements IOperationBuilder{

public Operation build(JsonNode ops) {

return new OperationAnd (ops);

}

}

protected class OperationShiftBuilder implements IOperationBuilder{

public Operation build(JsonNode ops) {

return new OperationShift(ops);

}

}

protected class OperationBitsBuilder implements IOperationBuilder{

public Operation build(JsonNode ops) {

return new OperationBits(ops);

}

}

protected void init () {

if (operations == null) {

operations = new HashMap<String, IOperationBuilder>();

operations.put("or", new OperationOrBuilder());

operations.put("xor", new OperationXorBuilder());

operations.put("and", new OperationAndBuilder());

operations.put("shift", new OperationShiftBuilder());

operations.put("set\_bits", new OperationBitsBuilder());

}

}

public Operation (String n, List<IOperation> ops) {

if (operations == null)

init();

this.name = n;

this.operands = ops;

}

public Operation (JsonNode n) {

if (operations == null)

init();

if (n.isInt()) {

this.name = "nop";

this.result = n;

} else {

this.operands = new ArrayList<IOperation>();

this.name = n.getFieldNames().next();

JsonNode ops = n.getElements().next();

JsonNode operand = null;

Iterator <JsonNode> it = ops.getElements();

while (it.hasNext()) {

operand = it.next();

this.operands.add(new Operation(operand));

}

}

}

public Operation () {

this.name = "nop";

this.operands = new ArrayList<IOperation> ();

}

@Override

public JsonNode result() {

if (this.name.equalsIgnoreCase("nop")) {

return result;

} else {

if (this.name.equalsIgnoreCase("and")) {

return resultAnd();

} else if (this.name.equalsIgnoreCase("or")) {

return resultOr();

} else if (this.name.equalsIgnoreCase("xor")) {

return resultXor();

} else if (this.name.equalsIgnoreCase("shift")) {

return resultShift();

} else return null;

}

}

private JsonNode resultOr () {

int result = 0;

for ( IOperation operand: operands) {

result |= operand.result().getValueAsInt();

}

return new IntNode (result);

}

private JsonNode resultAnd() {

int result = 255;

for ( IOperation operand: operands) {

result &= operand.result().getValueAsInt();

}

return new IntNode (result);

}

private JsonNode resultXor() {

int result = 0;

for ( IOperation operand: operands) {

result ^= operand.result().getValueAsInt();

}

return new IntNode (result);

}

private JsonNode resultShift() {

int result = operands.get(0).result().getValueAsInt();

int factor = operands.get(1).result().getValueAsInt();

return new IntNode (result << factor);

}

};

public class OperationValue extends Operation {

IntNode value;

public OperationValue(IntNode val) {

super();

this.value = val;

}

@Override

public JsonNode result() {

return this.value;

}

}

public class OperationOr extends Operation {

public OperationOr (List<IOperation> ops) {

super ("or", ops);

}

public OperationOr (JsonNode ops) {

super ();

name = "or";

JsonNode operand = null;

Iterator <JsonNode> it = ops.getElements();

while (it.hasNext()) {

operand = it.next();

if (operand.isInt() ) {

this.operands.add(new OperationValue((IntNode) operand));

} else {

String opName = operand.getFieldNames().next();

operands.add(operations.get(opName).build(operand.getElements().next()));

}

}

}

@Override

public JsonNode result() {

int result = 0;

for ( IOperation operand: operands) {

result |= operand.result().getValueAsInt();

}

return new IntNode (result);

}

}

public class OperationAnd extends Operation {

public OperationAnd (List<IOperation> ops) {

super ("and", ops);

}

public OperationAnd (JsonNode ops) {

super ();

name = "and";

JsonNode operand = null;

Iterator <JsonNode> it = ops.getElements();

while (it.hasNext()) {

operand = it.next();

if (operand.isInt() ) {

this.operands.add(new OperationValue((IntNode) operand));

} else {

String opName = operand.getFieldNames().next();

operands.add(operations.get(opName).build(operand.getElements().next()));

}

}

}

@Override

public JsonNode result() {

int result = 255;

for ( IOperation operand: operands) {

result &= operand.result().getValueAsInt();

}

return new IntNode (result);

}

}

public class OperationXor extends Operation {

public OperationXor (List<IOperation> ops) {

super ("xor", ops);

}

public OperationXor (JsonNode ops) {

super ();

name = "xor";

JsonNode operand = null;

Iterator <JsonNode> it = ops.getElements();

while (it.hasNext()) {

operand = it.next();

if (operand.isInt() ) {

this.operands.add(new OperationValue((IntNode) operand));

} else {

String opName = operand.getFieldNames().next();

operands.add(operations.get(opName).build(operand.getElements().next()));

}

}

}

@Override

public JsonNode result() {

int result = 0;

for ( IOperation operand: operands) {

result ^= operand.result().getValueAsInt();

}

return new IntNode (result);

}

}

public class OperationShift extends Operation {

public OperationShift (List<IOperation> ops) {

super ("shift", ops);

}

public OperationShift (JsonNode ops) {

super ();

name = "shift";

JsonNode operand = null;

Iterator <JsonNode> it = ops.getElements();

while (it.hasNext()) {

operand = it.next();

if (operand.isInt() ) {

this.operands.add(new OperationValue((IntNode) operand));

} else {

String opName = operand.getFieldNames().next();

operands.add(operations.get(opName).build(operand.getElements().next()));

}

}

}

@Override

public JsonNode result() {

int result = operands.get(0).result().getValueAsInt();

int factor = operands.get(1).result().getValueAsInt();

return new IntNode (result << factor);

}

}

public class OperationBits extends Operation {

public OperationBits (List<IOperation> ops) {

super ("set\_bits", ops);

}

public OperationBits (JsonNode ops) {

super ();

name = "set\_bits";

JsonNode operand = null;

Iterator <JsonNode> it = ops.getElements();

while (it.hasNext()) {

operand = it.next();

if (operand.isInt() ) {

this.operands.add(new OperationValue((IntNode) operand));

} else {

String opName = operand.getFieldNames().next();

operands.add(operations.get(opName).build(operand.getElements().next()));

}

}

}

@Override

public JsonNode result() {

int result = operands.get(0).result().getValueAsInt();

return new IntNode (result);

}

}

**BitmapSchema**

private static class BitmapSchema extends NamedSchema { //TODO OF Changes: BitmapSchema

private final int size;

boolean isError;

private IOperation defaultValue = null;

public IOperation getOperation() {

if (defaultValue == null)

throw new AvroRuntimeException("Schema operation not set yet");

return defaultValue;

}

public BitmapSchema(Name name, JsonNode sizeNode, JsonNode defaultValue, String doc, int size, boolean isError) {

super(Type.BITMAP, name, doc);

this.isError = isError;

this.size = size;

if (defaultValue != null) {

if (defaultValue.isInt()) {

this.defaultValue = new OperationValue((IntNode)defaultValue);

} else {

String opName = defaultValue.getFieldNames().next();

JsonNode operands = defaultValue.getElements().next();

if (opName.equalsIgnoreCase("or"))

this.defaultValue = new OperationOr(operands);

else if (opName.equalsIgnoreCase("and"))

this.defaultValue = new OperationAnd(operands);

else if (opName.equalsIgnoreCase("xor"))

this.defaultValue = new OperationXor(operands);

else if (opName.equalsIgnoreCase("shift"))

this.defaultValue = new OperationShift(operands);

else if (opName.equalsIgnoreCase("set\_bits"))

this.defaultValue = new OperationBits(operands);

else if (opName.equalsIgnoreCase("or"))

this.defaultValue = new OperationOr(operands);

}

JsonNode result = this.defaultValue.result();

}

}

}

# org.apache.avro.io.parsing.ResolvingGrammarGenerator.java

public static void encode(Encoder e, Schema s, JsonNode n)

throws IOException {

switch (s.getType()) {

... .. .. .. ..

case FIXED:

if (n.isTextual()) {

byte[] bb = n.getTextValue().getBytes("ISO-8859-1");

if (bb.length != s.getFixedSize()) {

bb = Arrays.copyOf(bb, s.getFixedSize());

e.writeFixed(bb);

}

} else

**if (n.isArray()) {**

**int ii = 0;**

**byte[] bb = new byte[n.size()];**

**ArrayList<Byte> a = new ArrayList<Byte>();**

**Iterator<JsonNode> it = n.getElements();**

**while (it.hasNext()) {**

**JsonNode tn = it.next();**

**bb[ii++] = (byte)tn.getIntValue();**

**}**

**e.writeFixed(bb);**

}

else

throw new AvroTypeException("Non-string and non-array default value for fixed: "+n);

.. .. .. ..

}