KIP-336: Consolidate ExtendedSerializer/Serializer and ExtendedDeserializer/Deserializer

Status

Current state: Accepted (vote thread)

Discussion thread: *here*

JIRA: *KAFKA-6923*

Planned Release: 2.1.0

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

When headers were introduced by KIP-82 - Add Record Headers the change created the `ExtendedSerializer` and `ExtendedDeserializer` classes in order to keep interface compatibility but still add `T deserialize(String topic, Headers headers, byte[] data);` and `byte[] serialize(String topic, Headers headers, T data);` methods that consume the headers for serialization/deserialization. The reason for doing so was that Kafka at that time needed be compatbile with Java 7. Since we're not compiling on Java 7 anymore (KAFKA-4423) we should consolidate the way we're using these in a backward compatible fashion.

Public Interfaces

Two new interface methods will be created, which are the extra deserialize/serialize methods taken from the Extended classes:

Serializer

```
public interface Serializer<T> extends Closeable {

    void configure(Map<String, ?> configs, boolean isKey);

    byte[] serialize(String topic, T data);

    default byte[] serialize(String topic, Headers headers, T data) {                        // This is the new
method
        return serialize(topic, data);
    }

    @Override
    void close();
}
```

Deserializer

```
public interface Deserializer<T> extends Closeable {

    void configure(Map<String, ?> configs, boolean isKey);

    T deserialize(String topic, byte[] data);

    default T deserialize(String topic, Headers headers, byte[] data) {                    // This is the new
method
        return deserialize(topic, data);
    }

    @Override
    void close();
}
```

Proposed Changes

1. Introduce the method above in both interfaces
2. deprecate `ExtendedSerializer/ExtendedDeserializer` and `Wrapper` saying that new and existing implementations should use `Serializer/Deserializer`
3. leave `ExtendedSerializer/ExtendedDeserializer` as is, so existing behavior won't change.
4. get rid of the internal usages of `ExtendedSerializer/ExtendedDeserializer` & `Wrapper.ensureExtended()`

Compatibility, Deprecation, And Migration Plan

All changes should be backward compatible. The defined serialize/deserialize methods in `ExtendedSerializer/ExtendedDeserializer` will act as abstract overrides for the default methods which is valid. From the perspective of the implementors it shouldn't be a breaking change as the generated code in `ExtendedSerializer/ExtendedDeserializer` will remain the same.

Test Plan

Review existing unit tests and system tests.

Manual verification of binary compatibility: existing ExtendedSerializer implementations should work without recompiling the application.

Rejected Alternatives

The code examples here are made with the `Deserializer` class but they are completely valid for the `Serializer` as well.

Default Implementations for the "headerless" methods

There are options to provide a default implementation: return a dummy value or throw an exception as shown below. This means that we'll have an interface where there are no methods enforced. Furthermore we should somehow suggest users that which method should they implement. To achieve this we could deprecate the 2 parameter ("headerless") method but this would litter the code base with warnings as we're still using this method in a lot of places (like the serializers and deserializers).

Serializer

```
public interface Serializer<T> extends Closeable {

    void configure(Map<String, ?> configs, boolean isKey);

        @Deprecated
    default byte[] serialize(String topic, T data) {
        return new byte[0];
                // throw new UnsupportedOperationException("Method not implemented");
    }

    default byte[] serialize(String topic, Headers headers, T data) {                       // This is the new
method
        return serialize(topic, data);
    }
    @Override
    void close();
}
```

Deserializer

```
public interface Deserializer<T> extends Closeable {

    void configure(Map<String, ?> configs, boolean isKey);

        @Deprecated
    default T deserialize(String topic, byte[] data) {
        return null;
                // throw new UnsupportedOperationException("Method not implemented");
    }

    default T deserialize(String topic, Headers headers, byte[] data) {                     // This is the new
method
        return deserialize(topic, data);
    }

    @Override
    void close();
}
```

Propagate to more complicated

We could propagate the method calls to the `deserialize(String,Headers,byte[])` method. The drawback here is that it wouldn't be backward compatible as the ExtendedDeserializer just did the opposite thing: it propagated the `deserialize(String,byte[])` method from the `deserialize(String,Headers,byte[])` method.

Propagate to more complicated

```
public interface Deserializer<T> extends Closeable {

        default T deserialize(String topic, byte[] data) {
            return deserialize(topic, null, data);
        }

        default T deserialize(String topic, Headers headers, byte[] data) {
            return null;
        }
}
```

Headers as parameter

In this version we'd make sure that `setHeaders` is called right before `deserialize` (or the same goes for `serialize`). The drawback of this is that we'd force implementations to maintain state.

Setter for Headers

```
public interface Deserializer<T> extends Closeable {

    void setHeaders(Headers headers);

    default T deserialize(String topic, byte[] data) {
        return null;
    }
}
```

Circular-referencing methods

This implementation is pretty good if the user overrides one (or both) the methods, however in the case where the default implementation is used for both of them, we'll get into a circular method reference and eventually a stack overflow.

Circular referencing methods

```
public interface Deserializer<T> extends Closeable {

        default T deserialize(String topic, byte[] data) {
            return deserialize(topic, null, data);
        }

        default T deserialize(String topic, Headers headers, byte[] data) {
            return deserialize(topic, data);
        }
}
```