KIP-707: The future of KafkaFuture

Status

Current state: *Adopted*

Discussion thread: *here*

***Vote thread**: here*

JIRA: *KAFKA-6987*

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

Motivation

The admin client makes extensive use of `KafkaFuture` in its `*Result` classes to enable clients the choice between synchronous and asynchronous programming styles. `KafkaFuture` originally had to work on versions of Java which didn't support `CompletionStage`/`CompletableFuture`, and so was intended to provide a useful subset of that missing JDK functionality. That constraint no longer applies. It would be beneficial to provide a richer API, like that of `CompletionStage`, and also compatibility with 3rd party APIs which require an actual `CompletionStage` or `CompletableFuture` instance.

This KIP proposes to add a `KafkaFuture.toCompletionStage()` method as a backward compatible solution to this problem.

Adding `toCompletionStage()` is sufficient because `CompletionStage` itself exposes `toCompletableFuture()`, so anyone who needs an actual CompletableFuture (e.g. for interoperating with 3rd party APIs that require one) can get one. However, `CompletableFuture` exposes methods for future completion which should not be called by users (only the Admin client should be completing the returned futures), so calling these will be prevented. It is expected that users wanting to block on the completion of the `KafkaFuture` would use `kafkaFuture.get()`, rather than calling `kafkaFuture.toCompletionStage().toCompletableFuture().get()`, so the need to access the `CompleteableFuture` should be rare.

Now also seems like a good opportunity to:

- Remove the `@InterfaceStability.Evolving` annotation on `KafkaFuture` to reflect the reality that changing this class incompatibly would cause of too much user code to break.
- Deprecate the static class `KafkaFuture.Function`, which already had Javadoc noting that `KafkaFuture.BaseFunction` was preferred.
- Annotating `KafkaFuture.Function`, `.BaseFunction` and `.BiFunction` with `@FunctionalInterface` , like the corresponding interfaces in `java.util`.

The methods of future admin client `*Result` classes would continue to use `KafkaFuture` for the sake of consistency.

Public Interfaces

The changes to `KafkaFuture` are summarized below:

```
/**
 * A flexible future which supports call chaining and other asynchronous programming patterns.
 *
 * <h3>Relation to {@code CompletableFuture}</h3>
 * <p>This class exists because support for a Future-like construct was once needed on Java versions predating
 * the addition of {@code CompletableFuture}. It is now possible to obtain a {@code CompletionStage} from a
 * {@code KafkaFuture} instance by calling {@link #toCompletionStage()}.
 * If converting {@link KafkaFuture#whenComplete(BiConsumer)} or {@link KafkaFuture#thenApply(BaseFunction)} to
 * {@link CompletableFuture#whenComplete(java.util.function.BiConsumer)} or
 * {@link CompletableFuture#thenApply(java.util.function.Function)} be aware that the returned
 * {@code KafkaFuture} will fail with an {@code ExecutionException}, where as a {@code CompletableFuture} fails
 * with a {@code CompletionException}.
 */
class KafkaFuture<Void> {

  // ... existing methods ...

  /**
   * Get a CompletionStage with the same completion properties as this KafkaFuture.
   * The returned instance will complete when this future completes and in the same way
   * (with the same result or exception).
   *
   * <p>Calling toCompletableFuture() on the returned instance will yield a CompletableFuture,
   * but invocation of the completion methods (complete() and other methods in the complete*()
   * and obtrude*() families) on that CompleteableFuture instance will result in
   * UnsupportedOperationException being thrown. Unlike a minimal CompletableFuture
   * the get*() and other methods of that CompletableFuture not inherited from CompletionStage
   * will work normally.
   *
   * <p>If you want to block on the completion of a KafkaFuture you should use
   * {@link #get()}, {@link #get} or {@link #getNow(Object)}, rather then calling
   * {@code .toCompletionStage().toCompletableFuture().get()} etc.

   */
  CompletionStage toCompletionStage();

  /**
   * A function which takes objects of type A and returns objects of type B.
   *
   * Prefer the functional interface {@link BaseFunction} over the class {@link Function}.  This class is here
for
   * backwards compatibility reasons and might be deprecated/removed in a future release.
   * @deprecated Replaced by the functional interface {@link BaseFunction} over the class {@link Function}.
   */
  @Deprecated // adding this
  public static abstract class Function<A, B> implements BaseFunction<A, B> { }

}
```

Proposed Changes

Some work has already been done to thoroughly test the existing `KafkaFuture` API and reimplement it using a `CompletableFuture` internally.

To get the required completion-safety properties a new (internal) `KafkaCompletableFuture` class, a subclass of `CompletableFuture`, will be introduced. This KIP will allow access to the instance of that subclass wrapped by a `KafkaFutureImpl`, and that instance will be completed within `KafkaFutureImpl` via a different method than the `complete`/`completeExceptionally` that it inherits from `CompletableFuture`.

`KafkaFutureImpl` would gain a new public constructor for wrapping a KafkaCompletableFuture, which will allow implementation of `KafkaFuture#allOf()` to be simplified.

Compatibility, Deprecation, and Migration Plan

The addition of `toCompetionStage` is backwards compatible.

As noted, `KafkaFuture.Function` will be formally deprecated. The lambda-compatible `BaseFunction` has existed and been documented as preferred for since Kafka 1.1.
The actual removal of `KafkaFuture.Function` can be done in some future major version of Kafka.

Rejected Alternatives

KafkaFuture implementing CompletionStage

`KafkaFuture` already has a `thenApply(KafkaFuture.BaseFunction)` method. Making `KafkaFuture` implement `CompletionStage` would require adding `thenApply(java.util.function.Function)`. That is not a source compatible because existing call sites using lambdas would become ambiguous, since both parameter types are SAM types. While it's easily resolved with a type cast, it's still incompatible. There would also be differences in the exception handling for `thenApply` and `whenComplete` in order to keep `KafkaFuture` compatible with its current behaviour.

Specifically, the `CompletionStage` contract states:

> *In all other cases, if a stage's computation terminates abruptly with an (unchecked) exception or error, then all dependent stages requiring its completion complete exceptionally as well, with a {@link CompletionException} holding the exception as its cause.*

But `KafkaFuture#thenApply(KafkaFuture.BaseFunction)` always fails using an `ExecutionException`. So the two `thenApply` and `whenComplete` methods would behave differently and resolving the compiler error by casting a lambda to `java.util.function.Function` could break users' exception handling.

"Admin2"

Other, more radical, possibilities include deprecating and replacing `KafkaFuture`-returning methods on a case-by-case basis (eugh), or creating a new `Admin2` client which used `CompletionStage` or `CompletableFuture` rather than `KafkaFuture` in its API, but was a thin wrapper of the existing `Admin`. These have a high cost to existing users of the admin client, who would have to change their code. They also come at a greater cost in terms of additional testing and documentation overhead for the project. And they don't offer any extra functionality beyond the solution proposed.

It is possible that this cost/benefit analysis might change in the future, for example if Project Loom's virtual threading model proves to be successful then having an `Admin2` client which supported only a synchronous programming model could make sense.