KIP-776: Add Consumer#peek for debugging/tuning

This page is meant as a template for writing a KIP. To create a KIP choose Tools->Copy on this page and modify with your content and replace the heading with the next KIP number and a description of your issue. Replace anything in italics with your own description.

Status

Current state: *"Under Discussion"*

Discussion thread: *here*

JIRA: *here*

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

Motivation

The current `Consumer#poll(Duration)` method is designed to block until data is available or the provided poll timeout expires. This implies, that if fetch requests fail the consumer retries them internally and eventually returns an empty set of records. – Thus, from a user point of view, returning an empty set of records can mean that no data is available at broker side or that the broker cannot be reached.

Besides, we sometimes wants to "peek" the incoming records, to do some testing, without affecting the offsets, like the "peek" method provided in many data structures (ex: java Queue). So, in this "peek" method, we won't increase the position offset in the partition. That means, after peek, the next "poll"ed records will still include the records returned by `peek`. Under the `enable.auto.commit = true` (default setting) case, because the offsets are not incremented, so it won't affect the committed offsets. That means, after the consumer restarted or rebalanced, the next poll will always start from the offset before operating peek methods. (of course if user manually commit the offsets, the offsets will be incremented)

Use cases:

Imagine we have brokers up now, and producers are producing records. We're a team developing consumers to consume the data, and feed into another integration process. Before this KIP, we need to do a polling loop, to retrieve the data, and see if the integration works as expected. If luckily yes, then, we can seek the offset to the beginning and start the new consumers to do the work, if no, we might need to poll more data, and do more troubleshooting cycle, but once the data are not producing fast enough, we might run into a situation that there are no more data in the brokers and we need to seek back to the beginning and restart again. After this KIP, the issues can be easily achieved via peek method, and also, if there's any connection issue between consumers and brokers, we can get the exception thrown via this peek testing.

So, we will have a `consumer#peek()` to allow consumers to:

1. peek what records existed at broker side and no increasing the position offsets.
2. throw exceptions when there is connection error existed between consumer and broker (or other exceptions will be thrown by "poll")

Public Interfaces

Add a `peek` method into `Consumer` interface

```
/**
 * @see KafkaConsumer#poll(long)
 */
ConsumerRecords<K, V> peek();

```

```
/**
 * Peek data for the topics or partitions specified using one of the subscribe/assign APIs.
 * It is an error to not have subscribed to any topics or partitions before polling for data.
 *
 * <p>
 * On each peek, consumer will try to use the last consumed offset as the starting offset and fetch
sequentially. The last
 * consumed offset can be manually set through {@link #seek(TopicPartition, long)} or automatically set as
the last committed
 * offset for the subscribed list of partitions.
 *
 * <p>
 * This method returns immediately if there are no records available or exception thrown.
 * Otherwise, it will await the passed timeout.
 * If the timeout expires, and there is no other exceptions, an empty record set will be returned.
 * Note that this method may block beyond the timeout in order to execute custom
 * {@link ConsumerRebalanceListener} callbacks.
 *
 * <p>
 * Note: The difference between #peek and #poll is that, peek won't increment the offsets, but poll will.
That is, after #peek,
 * when you do #poll, the returned records will include the records returned by previous #peek. Also, if
there's any IOException
 * while fetch records, the exception will be thrown during #peek, but not during #poll
 *
 * @param timeout The maximum time to block (must not be greater than {@link Long#MAX_VALUE} milliseconds)
 *
 * @return map of topic to records since the last fetch for the subscribed list of topics and partitions
 *
 * @throws java.io.IOException if unexpected error during I/O  <-- different from #poll
 * @throws org.apache.kafka.clients.consumer.InvalidOffsetException if the offset for a partition or set of
 *             partitions is undefined or out of range and no offset reset policy has been configured
 * @throws org.apache.kafka.common.errors.WakeupException if {@link #wakeup()} is called before or while this
 *             function is called
 * @throws org.apache.kafka.common.errors.InterruptException if the calling thread is interrupted before or
while
 *             this function is called
 * @throws org.apache.kafka.common.errors.AuthenticationException if authentication fails. See the exception
for more details
 * @throws org.apache.kafka.common.errors.AuthorizationException if caller lacks Read access to any of the
subscribed
 *             topics or to the configured groupId. See the exception for more details
 * @throws org.apache.kafka.common.KafkaException for any other unrecoverable errors (e.g. invalid groupId or
 *             session timeout, errors deserializing key/value pairs, your rebalance callback thrown
exceptions,
 *             or any new error cases in future versions)
 * @throws java.lang.IllegalArgumentException if the timeout value is negative
 * @throws java.lang.IllegalStateException if the consumer is not subscribed to any topics or manually
assigned any
 *             partitions to consume from
 * @throws java.lang.ArithmeticException if the timeout is greater than {@link Long#MAX_VALUE} milliseconds.
 * @throws org.apache.kafka.common.errors.InvalidTopicException if the current subscription contains any
invalid
 *             topic (per {@link org.apache.kafka.common.internals.Topic#validate(String)})
 * @throws org.apache.kafka.common.errors.UnsupportedVersionException if the consumer attempts to fetch
stable offsets
 *             when the broker doesn't support this feature
 * @throws org.apache.kafka.common.errors.FencedInstanceIdException if this consumer instance gets fenced by
broker.
 */
@Override
public ConsumerRecords<K, V> peek(final Duration timeout) { }


* @param partitions The partitions to fetch records from. If the partitions provided are not subscribed by
this consumer,
                   exception will be thrown.
* @param timeout The maximum time to block (must not be greater than {@link Long#MAX_VALUE} milliseconds)
@Override
public ConsumerRecords<K, V> peek(final Set<TopicPartition> partitions, final Duration timeout) { }
```

Proposed Changes

Provided a new method `peek(timeout)` in Consumer to allow user to:

1. peek what records existed at broker side and no increasing the position offsets.
2. throw exceptions when there is connection error existed between consumer and broker (or other exceptions will be thrown by "poll")

Compatibility, Deprecation, and Migration Plan

This is a new added method in Consumer interface. There will be no impact to the existing users.

Rejected Alternatives

1. Could be easily realized on the user side by using manual offset commit + offset position rewind

That's true.
But I have the same thoughts as Sagar, which is that, it's for advanced users.
Another reason is for simplicity. If you've ever used the peek API from java collection (ex: Queue#peek), you should know what I'm talking about. When you have data in a queue, if you want to know what the first data is in the queue, you'd use peek(). You can also achieve it by remove() the 1st element from queue, and then added it back to the right position, but I believe that's not what you'd do.