FLIP-12: Asynchronous I/O Design and Implementation

Status

<table>
<thead>
<tr>
<th>Discussion thread</th>
<th>Vote thread</th>
</tr>
</thead>
</table>

JIRA

- FLINK-4391 - Provide support for asynchronous operations over streams

Release 1.2

Google Doc: https://docs.google.com/document/d/1Lr9UYXEz6x6R_3PWg3bZQLF3upGaNEkc0rQCFSzaYDi/edit

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

Motivation

I/O access, for the most case, is a time-consuming process, making the TPS for single operator much lower than in-memory computing, particularly for streaming job, when low latency is a big concern for users. Starting multiple threads may be an option to handle this problem, but the drawbacks are obvious: The programming model for end users may become more complicated as they have to implement thread model in the operator. Furthermore, they have to pay attention to coordinate with checkpointing.

Terms

- AsyncFunction: Async I/O will be triggered in AsyncFunction.
- AsyncWaitOperator: An StreamOperator which will invoke AsyncFunction.
- AsyncCollector: For each input streaming record, an AsyncCollector will be created and passed into user's callback to get the async i/o result.
- AsyncCollectorBuffer: A buffer to keep all AsyncCollectors.
- Emitter Thread: A working thread in AsyncCollectorBuffer, being signalled while some of AsyncCollectors have finished async i/o and emitting results to the following opeartors.

Public Interfaces

A helper class, named AsyncDataStream, is added to provide methods to add AsyncFunction, which will do async i/o operation, into FLINK streaming job.
### AsyncDataStream.java

```java
class AsyncDataStream {
    public static DataStream<OUT> unorderedWait(DataStream<IN> in, AsyncFunction<IN, OUT> func, int bufSize);
    public static DataStream<OUT> unorderedWait(DataStream<IN> in, AsyncFunction<IN, OUT> func);

    public static DataStream<OUT> orderedWait(DataStream<IN> in, AsyncFunction<IN, OUT> func, int bufSize);
    public static DataStream<OUT> orderedWait(DataStream<IN> in, AsyncFunction<IN, OUT> func);
}
```

### Proposed Changes

#### Overview

The following diagram illustrates how the streaming records are processed while:

- arriving at AsyncWaitOperator
- recovering from task failover
- snapshotting state
- being emitted by Emitter Thread
Sequence Diagram
AsyncFunction

AsyncFunction works as a user function in AsyncWaitOperator which looks like StreamFlatMap operator, having open()/processElement (StreamRecord<IN> record)/processWatermark(Watermark mark).

For user's concrete AsyncFunction, the asyncInvoke(IN input, AsyncCollector<OUT> collector) has to be overriden to supply codes to start an async operation.
AsyncFunction.java

```java
public interface AsyncFunction<IN, OUT> extends Function, Serializable {
    /**
     * Trigger async operation for each stream input.
     * The AsyncCollector should be registered into async client.
     * @param input Stream Input
     * @param collector AsyncCollector
     */
    void asyncInvoke(IN input, AsyncCollector<OUT> collector) throws Exception;
}
```

```java
public abstract class RichAsyncFunction<IN, OUT> extends AbstractRichFunction
    implements AsyncFunction<IN, OUT> {
    @Override
    public abstract void asyncInvoke(IN input, AsyncCollector<OUT> collector) throws Exception;
}
```

For each input stream record of AsyncWaitOperator, they will be processed by `AsyncFunction.asyncInvoke(IN input, AsyncCollector<OUT> cb)`. Then `AsyncCollector` will be appended into `AsyncCollectorBuffer`. We will cover `AsyncCollector` and `AsyncCollectorBuffer` later.

AsyncCollector

AsyncCollector is created by AsyncWaitOperator, and passed into AsyncFunction, where it should be added into user’s callback. It acts as a role to get results or errors from user codes and notify the AsyncCollectorBuffer to emit results.

The functions specific for the user is the `collect`, and they should be called when async operation is done or errors are thrown out.

AsyncCollector.java

```java
public class AsyncCollector<OUT> {
    private List<OUT> result;
    private Throwable error;
    private AsyncCollectorBuffer<OUT> buffer;

    /**
     * Set result
     * @param result A list of results.
     */
    public void collect(List<OUT> result) {
        this.result = result;
        buffer.mark(this);
    }

    /**
     * Set error
     * @param error A Throwable object.
     */
    public void collect(Throwable error) {
        this.error = error;
        buffer.mark(this);
    }

    /**
     * Get result. Throw RuntimeException while encountering an error.
     * @return A List of result.
     */
    public List<OUT> getResult() throws RuntimeException {
        // ...
    }
}
```

How is it used
Before calling AsyncFunction.asyncInvoke(IN input, AsyncCollector<OUT> collector), AsyncWaitOperator will try to get an instance of AsyncCollector from AsyncCollectorBuffer. Then it will be taken into user’s callback function. If the buffer is full, it will **wait** until some of ongoing callbacks has finished.

Once async operation has done, the AsyncCollector.collect() will take results or errors and AsyncCollectorBuffer will be notified.

AsyncCollector is implemented by FLINK.

**AsyncCollectorBuffer**

AsyncCollectorBuffer keeps all AsyncCollectors, and emit results to the next nodes.

When AsyncCollector.collect() is called, a mark will be placed in AsyncCollectorBuffer, indicating finished AsyncCollectors. A working thread, named Emitter, will also be **signalled** once a AsyncCollector gets results, and then try to **emit** results depending on the ordered or unordered setting.

For simplicity, we will refer task to AsyncCollector in the AsyncCollectorBuffer in the following text.

### Ordered and Unordered

Based on the user configuration, the order of output elements will or will not be guaranteed. If not guaranteed, the finished AsyncCollectors coming later will be emitted earlier.

**Emitter Thread**

The Emitter Thread will wait for finished AsyncCollectors. When it is signalled, it will process tasks in the buffer as follow:

- **Ordered Mode**
  
  If the first task in the buffer is finished, then Emitter will collect its results, and then proceed to the second task. If the first task is not finished yet, just **wait** for it again.

- **Unordered Mode**
  
  Check all finished tasks in the buffer, and collect results from those tasks which are prior to the oldest Watermark in the buffer.

The Emitter Thread and Task Thread will access **exclusively** by acquiring/releasing the lock.

**Signal** Task Thread when all tasks have finished to notify it that all data has been processed, and it is OK to close the operator.

**Signal** Task Thread after removing some tasks from the buffer.
Propagate Exceptions to Task Thread.

Task Thread
Access AsyncCollectorBuffer exclusively against the Emitter Thread.
Get and Add a new AsyncCollector to the buffer, wait while buffer is full.

Watermark
All watermarks will also be kept in AsyncCollectorBuffer. A watermark will be emitted if and only if after all AsyncCollectors coming before current watermark have been emitted.

State, Failover and Checkpoint

State and Checkpoint
All input StreamRecords will be kept in state. Instead of storing each input stream records into state one by one while processing, AsyncWaitOperator will put all input stream records in AsyncCollectorBuffer into state while snapshotting operator state. Old data in the state will be cleared before persisting those records.
When all barriers have arrived at the operator, checkpoint can be carried out immediately.

Failover
While restoring the operator’s state, the operator will scan all elements in the state, get AsyncCollectors, call AsyncFunction.asyncInvoke() and insert them back into AsyncCollectorBuffer.

Notes
Async Resource Sharing
For the case to share async resources (like connection to hbase, netty connections) among different slots(task workers) in the same TaskManager(a.k.a the same JVM), we can make the connection static so that all threads in the same process can share the same instance.
Of course, please pay attention to thread safety while using those resources.

Example
Example.java

```java
/***  ***/
public class HBaseAsyncFunction implements AsyncFunction<String, String> {
    // initialize it while reading object
    transient Connection connection;

    @Override
    public void asyncInvoke(String val, AsyncCollector<String> c) {
        Get get = new Get(Bytes.toBytes(val));
        Table ht = connection.getTable(TableName.valueOf(Bytes.toBytes("test")));
        // UserCallback is from user’s async client.
        ((AsyncableHTableInterface) ht).asyncGet(get, new UserCallback(c));
    }
}

// create data stream
public void createHBaseAsyncTestStream(StreamExecutionEnvironment env) {
    DataStream<String> source = getDataStream(env);
    DataStream<String> stream = AsyncDataStream.unorderedWait(source, new HBaseAsyncFunction());
    stream.print();
}
```

For ListenableFuture
import com.google.common.util.concurrent.FutureCallback;
import com.google.common.util.concurrent.Futures;
import com.google.common.util.concurrent.MoreExecutors;
import com.google.common.util.concurrent.ListenableFuture;

public class HBaseAsyncFunction implements AsyncFunction<String, String> {
    // initialize it while reading object
    transient Connection connection;

    @Override
    public void asyncInvoke(String val, AsyncCollector<String> c) {
        Get get = new Get(Bytes.toBytes(val));
        Table ht = connection.getTable(TableName.valueOf(Bytes.toBytes("test")));

        ListenableFuture<Result> future = ht.asyncGet(get);
        Futures.addCallback(future,
            new FutureCallback<Result>() {
                @Override public void onSuccess(Result result) {
                    List ret = new ArrayList<String>();
                    ret.add(result.get(...));
                    c.collect(ret);
                }
            },
            MoreExecutors.newDirectExecutorService()
        );
        },

    @Override public void onFailure(Throwable t) {
        c.collect(t);
    }

    public void createHBaseAsyncTestStream(StreamExecutionEnvironment env) {
        DataStream<String> source = getDataStream(env);
        DataStream<String> stream = AsyncDataStream.unorderedWait(source, new HBaseAsyncFunction());
        stream.print();
    }
}

Compatibility, Deprecation, and Migration Plan
No compatibility and migration issues for the users' current FLINK codes.

Test Plan
Add Unittest cases / Function test cases

Rejected Alternatives
N/A