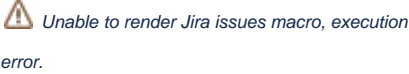


FLIP-214: Support Advanced Function DDL

Discussion thread	https://lists.apache.org/thread/7m5md150qgodgz1wlp5plx15j1nowx8
Vote thread	https://lists.apache.org/thread/ht8cbh9r2gpt2lt611d61j2qhr79pznz
JIRA	
Release	1.16

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

Motivation

This proposal is a continuation of [FLIP-79](#) in which Flink Function DDL is defined. Until now it is partially released as the flink function DDL with remote resources is not clearly discussed and implemented. It is an important feature for SQL platform engineers to scale their UDF management as the UDF can be released separately for different SQL users.

Proposed Change

Syntax

```
CREATE [TEMPORARY|TEMPORARY SYSTEM] FUNCTION [IF NOT EXISTS] [catalog_name.db_name.]function_name AS class_name [LANGUAGE  
JAVA|SCALA|PYTHON] [USING JAR 'resource_path' [, JAR 'resource_path']*];
```

The red part is what we will discuss in the proposal. This statement allow user to create a function that is implemented by the class_name. Jars which need to be added to the table environment can be specified with the USING clause; when the function is referenced for the first time by a Flink session, these resources will be added to the table environment as if [ADD JAR](#) had been issued.

Use Cases

Use Local Resource

```
CREATE TEMPORARY FUNCTION catalog1.db1.NestedOutput AS 'com.xxx.udf.NestedOutputUDF' LANGUAGE JAVA USING JAR 'file:///xxx-udf/xxx-  
udf-1.0.1-20180502.011548-12.jar'
```

Use Remote Resource

```
CREATE TEMPORARY FUNCTION catalog1.db1.NestedOutput AS 'com.xxx.udf.NestedOutputUDF' LANGUAGE JAVA USING JAR 'hdfs:///xxx-udf/1.  
0.1-SNAPSHOT/xxx-udf-1.0.1-20180502.011548-12.jar'
```

New or Changed Public Interfaces

Execution Flow

The advanced ddl syntax allows the Flink SQL user or Table API user to use udf defined in local file or remote resource. Thus, it requires the class to be loaded correctly in the stages of calcite validation, code generation and distributed execution.

[blocked URL](#)

As shown in the diagram above, the class in remote resources needs to be loaded as a catalog function in the table environment for calcite sql validation and code generation, then the path needs to be registered into the execution environment, so that it can be used in distributed runtime. The deployment model plays an important role in setting the context of where the resource is. It could be in blobstore or just a path item in the classpaths of the job graph in a different deployment model. Thus, we may first introduce the API changes required on the left side in this section, then discuss the resource shipping in different deployment models.

Public API Changes

ResourceUri

```
/** Description of function resource information. */
@PublicEvolving
public class ResourceUri {

    /** ResourceType. */
    public enum ResourceType {
        FILE,
        JAR,
        ARCHIVE
    }

    private final ResourceType resourceType;
    private final String uri;

    public ResourceUri(ResourceType resourceType, String uri) {
        this.resourceType = resourceType;
        this.uri = uri;
    }

    public ResourceType getResourceType() {
        return resourceType;
    }

    public String getUri() {
        return uri;
    }
}
```

CatalogFunction

This API will provide `getFunctionResources` method which is used to get resource information of UDF, and deprecate the `isGeneric` method.

```
public interface CatalogFunction {
    /**
     * Get a detailed resource description of the function.
     *
     * @return an {@link ResourceUri} list of the function
     */
    List<ResourceUri> getFunctionResources();

    /**
     * Distinguish if the function is a generic function.
     *
     * @deprecated This method is currently only used in hive to determine
     * if a function is a generic function. The behavior should be implemented
     * by hive itself, instead of providing a public api, so we deprecate it.
     *
     * @return whether the function is a generic function
     */
    @Deprecated
    boolean isGeneric();
}
```

FunctionDefinitionFactory

For UDF function created with custom resources, a default method is provided here to load the class and get function definition using the user class loader.

```

public interface FunctionDefinitionFactory {
    /**
     * Creates a {@link FunctionDefinition} from given {@link CatalogFunction}. If the
     * {@link CatalogFunction} is created by user defined resource, the user of
     * {@link FunctionDefinitionFactory} needs to override this method explicitly.
     *
     * <p> It is recommended to use the given user classloader to load the function class,
     * because if the function has a resource URL, the framework will help load
     * the resource into the given classloader.
     *
     * @param name name of the {@link CatalogFunction}
     * @param catalogFunction the catalog function
     * @param userClassLoader the class loader is used to load user defined function's class
     * @return
     */
    default FunctionDefinition createFunctionDefinition(
        String name,
        CatalogFunction catalogFunction,
        ClassLoader userClassLoader) {
        if (!CollectionUtil.isNullOrEmpty(catalogFunction.getFunctionResources())) {
            throw new UnsupportedOperationException(
                String.format("%s need to override default createFunctionDefinition for "
                    + "loading user defined function class", this.getClass().getSimpleName()));
        } else {
            return createFunctionDefinition(name, catalogFunction);
        }
    }
}

```

TableEnvironment

Providing some methods that are used to register UDF for Table API user.

```

@PublicEvolving
public interface TableEnvironment {

    /**
     * Registers a {@link UserDefinedFunction} class as a temporary system function by the specific
     * class name and user defined resource uri.
     *
     * <p>Compared to {@link #createTemporarySystemFunction(String, Class)}, this method allow
     * registering a user defined function by only provide a full path class name and an available
     * resource which may be local or remote. User doesn't need to initialize the function instance
     * in advance.
     *
     * <p>Temporary functions can shadow permanent ones. If a permanent function under a given name
     * exists, it will be inaccessible in the current session. To make the permanent function
     * available again one can drop the corresponding temporary system function.
     *
     * @param name The name under which the function will be registered globally.
     * @param className The class name of UDF to be registered.
     * @param resourceUri The udf resource uri in local or remote.
     */
    void createTemporarySystemFunction(String name, String className, ResourceUri resourceUri);

    /**
     * Registers a {@link UserDefinedFunction} class as a catalog function in the given path by the
     * specific class name and user defined resource uri.
     *
     * <p>Compared to {@link #createFunction(String, Class)}, this method allow registering a user
     * defined function by only provide a full path class name and an available resource which may
     * be local or remote. User doesn't need to initialize the function instance in advance.
     *
     * <p>Compared to system functions with a globally defined name, catalog functions are always
     * (implicitly or explicitly) identified by a catalog and database.
     *
     * <p>There must not be another function (temporary or permanent) registered under the same
     * path.
     */
}

```

```

*
* @param path The path under which the function will be registered. See also the {@link
*      TableEnvironment} class description for the format of the path.
* @param className The class name of UDF to be registered.
* @param resourceUri The udf resource uri in local or remote.
*/
void createFunction(String path, String className, ResourceUri resourceUri);

/**
* Registers a {@link UserDefinedFunction} class as a catalog function in the given path by the
* specific class name and user defined resource uri.
*
* <p>Compared to {@link #createFunction(String, Class)}, this method allow registering a user
* defined function by only provide a full path class name and an available resource which may
* be local or remote. User doesn't need to initialize the function instance in advance.
*
* <p>Compared to system functions with a globally defined name, catalog functions are always
* (implicitly or explicitly) identified by a catalog and database.
*
* <p>There must not be another function (temporary or permanent) registered under the same
* path.
*
* @param path The path under which the function will be registered. See also the {@link
*      TableEnvironment} class description for the format of the path.
* @param className The class name of UDF to be registered.
* @param resourceUri The udf resource uri in local or remote.
* @param ignoreIfExists If a function exists under the given path and this flag is set, no
*      operation is executed. An exception is thrown otherwise.
*/
void createFunction(
    String path, String className, ResourceUri resourceUri, boolean ignoreIfExists);

/**
* Registers a {@link UserDefinedFunction} class as a temporary catalog function in the given
* path by the specific class name and user defined resource uri.
*
* <p>Compared to {@link #createTemporaryFunction(String, Class)}, this method allow registering
* a user defined function by only provide a full path class name and an available resource uri
* which may be local or remote. User doesn't need to initialize the function instance in
* advance.
*
* <p>Compared to {@link #createTemporarySystemFunction(String, String, ResourceUri)} with a
* globally defined name, catalog functions are always (implicitly or explicitly) identified by
* a catalog and database.
*
* <p>Temporary functions can shadow permanent ones. If a permanent function under a given name
* exists, it will be inaccessible in the current session. To make the permanent function
* available again one can drop the corresponding temporary function.
*
* @param path The path under which the function will be registered. See also the {@link
*      TableEnvironment} class description for the format of the path.
* @param className The class name of UDF to be registered.
* @param resourceUri The udf resource uri in local or remote.
*/
void createTemporaryFunction(String path, String className, ResourceUri resourceUri);
}

```

StreamExecutionEnvironment

To pass the resource path information from table environment to execution environment, we will use **pipeline.jars** option. Currently `StreamExecutionEnvironment.configure` method doesn't override **pipeline.jars** option from dynamic configuration when generate Jobgraph, this will result in the user jar used in the query not being uploaded to the blobstore and a `ClassNotFoundException` will be thrown at distributed runtime, so the **pipeline.jars** option needs to be overridden in this method.

We can use the **pipeline.jars** option have the precondition is that the resource will be downloaded to local during client compile phase whether the resource is local or remote.

Implementation Plan

Supported Resource Type

As described in the use case section, the advanced function DDL is going to support both local resource and remote resource. Using local resource is very convenient for user to develop and debug jobs. In a production environment, user often need to reuse a resource across jobs or sessions. In addition, user basically submit jobs through the SQL development platform, and their local resources and development platform are not together, so using remote resources is a better way. The effort for these two types of resource are different. Thus, we will discuss them separately.

Local Resource

For local resource, the local path is added to the user class loader during job compilation phase to ensure that the corresponding class can be loaded during calcite validation, code generation. It is also added to the user jar path of JobGraph to distribute the resources to all machines in the cluster at distributed execution time.

Remote Resource

For remote resource there are different storage scheme such as HDFS, S3, OSS etc. During the compilation period of the query, it will first download the remote resource to a local temporary directory based on its path, and then add the local path to the user class loader. The next behavior is the same as local resource section.

We will use Flink's FileSystem abstraction to download remote resource, which can support all types of file system, including OSS, S3, HDFS, etc, rather than binding to a specific implementation.

Note: Currently, `ADD JAR` syntax only supports adding local resources. With the release of this advanced function DDL feature, `ADD JAR` syntax also supports adding remote resources.

Supported Deployment Mode/Resource Type Mapping

Mode/Type	Local	Remote
Per Job Mode		
Session Mode		
Application Mode		

Limitation: For application mode, the use of local resources is special, user need to put the resources into flink `usrlib` folder before submit job, only then the resources are available when running query, please refer to [Application Mode](#) for details.

Proposed Design

UDF Registration Process

If the user uses `CREATE FUNCTION ... USING JAR` statement to register the UDF, the general process is very simple, we only create the UDF in catalog by class name and store the path information of UDF resources. we neither check for resource available nor validity of the class exists because of `CREATE FUNCTION` are pure metadata catalog operations in Flink.

UDF Usage Process

If the user uses a UDF registered by `CREATE FUNCTION ... USING JAR` statement in query, the general process is as follows:

1. Parsing the SQL to find the UDF used in the query, if it is found that the UDF is registered with the Jar resource, first determine whether the Jar resource has been registered.
2. If the Jar is not registered before, determine whether the Jar resource path is in the remote. If in the remote, will first download the Jar resource to the local temporary directory which will be generated using UUID.
3. Then the local path will be registered into the user class loader and loaded the class into JVM for calcite validation and code generation.
4. Finally, when the job is submitted, the Jar resource is uploaded to the blobstore at the same time, so that it is available at distributed runtime.

Core Code Design

As already mentioned in [FLINK-15635](#), we had a couple of class loading issues in the past because people forgot to use the right class loader in flink-table. The SQL Client executor code hacks a class loader into the planner process by using `wrapClassLoader` that sets the threads context class loader. Instead we should allow passing a class loader to environment settings. This class loader can be passed to the planner and can be stored in table environment, table config, etc. to have a consistent class loading behavior.

In addition, due to the problem of class loader, the **execution** of `ADD JAR/REMOVE JAR` syntax is implemented currently inside the SQL Client, we will move the execution to table environment side, so that the syntax is more general, for example Table API user can also use it.

In the current code implementation, the table environment holds a `ClassLoader` object, but this is not enough. If the `REMOVED JAR` syntax is used before using UDF, the corresponding Jar information will be removed from the class loader, which leads the old `ClassLoader` will be closed and a new `ClassLoader` will be constructed. As a result, `ClassLoader` object in other objects such as `CatalogManager` throughout the flink-table module will no longer be available because the latest `ClassLoader` object is not obtained.

Comprehensive consideration of the above, based on [FLINK-15635](#), in this FLIP, the overall code design is as follows:

1. Proposing a user class loader called **MutableURLClassLoader** which contains a parent classloader, owner classloader and URL list registered by user. This classloader supports add and remove jar url, so can modify the urls in it dynamically.
2. Proposing an internal interface **UserResourceManager**, this entity performs bookkeeping of used/loaded resources. This entity should also provide a `MutableURLClassLoader` which parent classloader originated from the class loader given via `EnvironmentSettings`. `UserResourceManager` is responsible for manager all used resources.
3. The table environment directly refers `UserResourceManager` instead of `ClassLoader` object, to manager all jar resources get from related resource SQL syntax and add or remove the specific jar url to `MutableURLClassLoader`.
4. `FunctionCatalog` also needs to refer `UserResourceManager` which is used to manager jar get from `CREATE FUNCTION ... USING JAR` syntax.

Migration Plan and Compatibility

It is a new feature for Function DDL, there is no migration needed.

Test Plan

1. For local resource, changes will be verified by UT.
2. For remote resource, changes will be verified by manually.

Rejected Alternatives

N/A

References

1. [Add advanced function DDL syntax "USING JAR/FILE/ACHIVE"](#)
2. [Allow passing a ClassLoader to EnvironmentSettings](#)
3. [LanguageManual DDL#Create/Drop/ReloadFunction](#)