FLIP-28: Long-term goal of making flink-table Scala-free

Reason	Subsumed by the bigger vision described in FLIP-32.

Status

Current state: *"Discarded"*

Discussion thread: http://apache-flink-mailing-list-archive.1008284.n3.nabble.com/DISCUSS-Long-term-goal-of-making-flink-table-Scala-free-td22761.html

JIRA: FLINK-11063

Released: <Flink Version>

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

Motivation

Currently, the Table & SQL API is implemented in Scala. This decision was made a long-time ago when the initial code base was created as part of a master's thesis. The community kept Scala because of the nice language features that enable a fluent Table API like table.select('field.trim()) and because Scala allows for quick prototyping (e.g. multi-line comments for code generation). The committers enforced not splitting the code-base into two programming languages.

However, nowadays the flink-table module more and more becomes an important part in the Flink ecosystem. Connectors, formats, and SQL client are actually implemented in Java but need to interoperate with flink-table which makes these modules dependent on Scala. As mentioned in an earlier mail thread, using Scala for API classes also exposes member variables and methods in Java that should not be exposed to users. Java is still the most important API language and right now we treat it as a second-class citizen. For example, we just noticed that you even need to add Scala if you just want to implement a ScalarFunction because of method clashes between the basic methods `public String toString()` and `public scala.Predef.String toString()`.

Goal

This FLIP aims to contain guidelines for the future structuring of the `flink-table` module. It also coordinates Scala-to-Java porting efforts.

Proposed Changes

Given the size of the current code base, reimplementing the entire `flink-table` code in Java is a goal that we might never reach. However, we should at least treat the symptoms and have this as a long-term goal in mind.

With the recent support of Scala 2.12, we needed proper suffixes of modules that depend on Scala. We introduced a flink-table-common module in order to avoid Scala-dependencies in format modules. However, table connectors still have Scala dependencies because the interfaces transitively pull in almost the entire Table API.

Our suggestion would be to split the code base further into multiple modules:

flink-table [moved out of flink-libraries as a top-level parent module]

- **flink-table-common**
  Contains interfaces and common classes that need to be shared across different Flink modules. This module was introduced in Flink 1.7 and its name integrates nicely with the existing Flink naming scheme.

  Connectors, formats, and UDFs can use this without depending on the entire Table API stack or Scala. The module contains interface classes such as descriptors, table sink, table source. It will also contain the table factory discovery service such that connectors can discover formats.

  The module should only contain Java classes and should have no external dependencies to other modules. It contains a `flink-core` dependency for common classes such as `TypeInformation`.

  In the future, we might need to add some basic expression representations (for <, >, ==, !=, field references, and literals) in order to push down filter predicates into sources without adding a dependency on `flink-table-api-base` or Calcite.

  Currently, we cannot add interfaces for connectors into the module as classes such as `StreamTableSource` or `BatchTableSource` require a dependency to `DataStream` or `DataSet` API classes. This might change in the future once other modules such as `flink-streaming-java` have been reworked. For now, extension points for connectors are located in `flink-table-api-*` and integrate with the target API.

- **flink-table-api-base**
  Contains API classes such as expressions, TableConfig and base classes for Table and TableEnvironment. It contains most classes from `org.apache.flink.table.api.*` plus some additional classes. It contains subclasses of `org.apache.flink.table.plan.logical.LogicalNode`.

  This module will be used by language-specific modules. If at all, it will have only Calcite as external dependency (+ shaded Calcite dependencies) since expressions need to be converted into `RexCall`s and nodes need to be converted into `RelNode`s. However, we should aim to not expose Calcite through the API. Only `flink-table-planner` should require Calcite.

  Additionally, the module will depend on `flink-table-common`.

  This module should only contain Java classes.

- **flink-table-api-java**
  Contains API classes with interfaces targeted to Java users, i.e. `BatchTableEnvironment` and `StreamTableEnvironment` extending some base class.

  The module should only contain Java classes. It will only dependent on `flink-table-api-base` and `flink-streaming-java`.

- **flink-table-api-scala**
  Contains API classes with interfaces targeted to Scala users, i.e. `BatchTableEnvironment` and `StreamTableEnvironment` extending some base class.

  The module should only contain Scala classes. It will only dependent on `flink-table-api-base` and `flink-streaming-scala`.

  There were opinions about letting `flink-table-api-scala` depend on `flink-table-api-java` and removing the base module. However, the problem with this approach is that classes such as `BatchTableEnvironment` or `Tumble` would be twice in the classpath. In the past, this led to confusion because people were not always paying attention to their imports and were mixing Java API with Scala API. The current module structure avoids ambiguity.

- **flink-table-planner**
  Contains the main logic for converting a logical representation into DataStream/DataSet program that only relies on `flink-table-runtime`. The planner module bridges `api` and `runtime` module similar to how it is done in the DataSet API of Flink. A user has to add `flink-table-api-scala/java` and `flink-table-planner` in order to execute a program in an IDE.

  This module contains the original `flink-table` module classes. It will gradually be converted into Java and some classes will be distributed to their future location in `flink-table-runtime` or `flink-table-api-*`. This might take a while because it contains a large set of rules, code generation, and translation logic and would be the biggest migration effort.

  For example, code generation is currently using Scala features such as multiline strings and string interpolation. Doing this in Java might not be as convenient. Either we wait until Java has better support (e.g. raw string literals look promising) or (better) we change the code generation to a programmatic approach *Expr.if(Expr.assign(var, value), Expr.throw(exception))* or something similar.

  We could make this module pretend to be Scala-free by only loading Scala dependencies into a separate classloader. A dedicated `Planner` class could be the interface between API and planning modules. Such a signature could look similar to:

  DataStream<OUT> translateStream(PlannerContext context, RelNode plan)
  DataSet<OUT> translateBatch(PlannerContext context, RelNode plan)

  The module will depend on the Calcite as external dependency. Internally, it will also require `flink-streaming-java` and `flink-table-runtime`.

- **flink-table-runtime**
  Contains the main logic for executing a table program. It aims to make JAR files that need to be submitted to the cluster small.

  The module will be a mixed Scala/Java project until we converted all classes to Java. However, compared to `flink-table-planner` this should be an achievable goal as runtime classes don't use a lot of Scala magic and are usually pretty compact.

  The module will depend on the Janino compiler and `flink-streaming-java`.

  Currently, we use some Calcite functions during runtime. Either we have to find alternatives (e.g. for time conversion) or we need to add a Calcite dependency to this module as well.

- **flink-sql-client**
  The SQL Client logically belongs to `flink-table` and should be moved under this module.

Public Interfaces

Currently, it is hard to estimate API breaking changes. However, the Table API is not marked as API stable. Thus, public interfaces might change without further notice. Of course we try to avoid this as much as possible.

One of the biggest changes for users will be the new module structure and that we will not expose protected methods and member variables in Java anymore.

Compatibility, Deprecation, and Migration Plan

We suggest the following steps to unblock people from developing new features but also start porting existing code either when touching the corresponding class or as a voluntary contribution.

Migration of Existing Classes

In order to clarify the terms here, "migration" means that Scala code is rewritten to Java code without changing the original logic. Breaking existing APIs should be avoided.

Due to different class member visibility principles in Scala and Java, it might be necessary to adapt class structures. For example, `private[flink]` is used quite often and would be `public` in Java which is not always intended, thus, we need to find a reasonable abstraction for these cases.

Since migrating code is a good chance for a code base review, an implementer should pay attention to code deduplication, exposing methods/fields, and proper annotations with `@Internal`, `@PublicEvolving` when performing the migration.

Tests should be migrated in a separate commit. This makes it possible to validate the ported code first before touching test classes.

Development of New Classes

New classes should always be implemented in Java if the surrounding code does not force Scala-specific code.

Examples:

A runtime class that only depends on `ProcessFunction` should be implemented in Java.

A new planner rule or node that only depends on Calcite and runtime classes should be implemented in Java.

If the surrounding code requires Scala, we leave it up to the implementer and committer to decide if related classes should be adapted or even migrated for Java code. If they are not adapted/migrated, a Jira issue should track such a shortcoming and the new class can still be implemented in Scala.

Examples:

A new class needs to implement a trait that requires a Scala collection or `Option` in parameters. The Java code should not depend on Scala classes. Therefore, the trait should be adapted to require Java collections or Java `Optional`. This should happen in a separate commit. If adapting the signature for this trait is too much work because it touches a lot of classes and thus is out of scope for the actual issue, implement a Scala class for now. But open an issue for it to track bigger migration blockers.

A new code generating class needs to be implemented but there are no utility methods for Java so far. Doing multiline code generation with variables and expressions inside is inconvenient in Java. We need to introduce proper tooling for this first, it is acceptable to implement this in Scala for now.

Migration Roadmap

The following steps should enable a smooth migration from Java to Scala.

1. **Setup new module structure**
   Move all files to their corresponding modules as they are. No migration happens at this stage. Modules might contain both Scala and Java classes. Classes that are in Scala so far remain in `flink-table-planner` for now.
2. **Migrate UDF classes to `flink-table-common`**
   All UDF interfaces have little dependencies to other classes.
3. **Migrate `flink-table-runtime` classes**
   All runtime classes have little dependencies to other classes.

4. **Migrate main Table API classes to `flink-table-api-base`**
   The most important API classes such as TableEnvironments and Table are exposing a lot of protected methods in Java. Migrating those classes makes the API clean and the implementation ready for a major refactoring for the new catalog support. We can also think about a separation of interface and implementation; e.g. `Table` & `TableImpl`. However, the current API design makes this difficult as we are using constructors of interfaces `new Table(...)`.
5. **Migrate connector classes to `flink-table-api-*`**
   Once we implemented improvements to the unified connector interface, we can also migrate the classes. Among others, it requires a refactoring of the timestamp extractors which are the biggest blockers because they transitively depending on expressions.
6. **Migrate remaining `flink-table-common` classes**
   While doing tasks for the new external catalog integration or improvements to the unified connector interfaces, we can migrate the remaining classes.
7. **Migrate remaining `flink-table-api-base` classes**
   This includes expressions, logical nodes etc.
8. **Load Scala in `flink-table-planner` into a separate classloader**
   After this stage, `flink-table` would be Scala-free from a dependency perspective.
9. **Migrate `flink-table-planner` classes**
   Final goal of Scala-free `flink-table`.

Test Plan

We already have a good test coverage for operators and the Scala Table API. We might need to add additional Java Table API tests. As mentioned before, tests should be migrated in a separate commit. This makes it possible to validate the ported code first before touching test classes.

Rejected Alternatives

See discussion thread on the mailing list.