Java API Internals

NOTE: This Wiki is obsolete as of November 2016 and is retained for reference only.

This page documents the design and internals of Spark's Java API and is intended for those developing Spark itself; if you are a user and want to learn to use Spark from Java, please see the Java programming guide.

Why a Java API?

Scala and Java are fairly interoperable, but there are several subtleties that make it difficult to directly call Spark's Scala APIs from Java:
- Spark uses Scala's implicit conversions to define additional operations on RDDs of key-value pairs and doubles, such as `reduceByKey`, `join`, and `stdev`.
  Since Java doesn't support implicit conversions, users have to manually instantiate the `PairRDDFunctions` and `DoubleRDDFunctions` classes to access these methods.
- Many Spark functions take implicit `ClassManifest` arguments; users have to manually pass `ClassManifest` instances when calling these functions from Java.
- To express user-defined functions in Java, users have to subclass Scala's internal function classes, which can be confusing.
- Many of Spark's methods accept or return Scala collection types; this is inconvenient and often results in users manually converting to and from Java types.

These difficulties made for an unpleasant user experience. To address this, the Spark 0.7 release introduced a Java API that hides these Scala <-> Java interoperability concerns.

Implementation

The Java API is implemented as a thin wrapper over the equivalent Scala APIs. The bulk of this wrapper is actually implemented using Scala, except we use Java-friendly types and hide Scala-specific features, like `ClassManifest`.

Spark defines additional operations on RDDs of key-value pairs and doubles, such as `reduceByKey`, `join`, and `stdev`. In the Scala API, these methods are automatically added using Scala's implicit conversions mechanism. In the Java API, the extra methods are defined in the `JavaPairRDD` and `JavaDoubleRDD` classes.

Function Classes

Transformations like `map` return different RDDs depending on the type of function that's applied to the RDD's elements. Due to type erasure, this overloading is ambiguous and won't work:

```scala
def map[R](f: Function1[T, R]): JavaRDD[R]
def map(f: Function1[T, Double]): JavaDoubleRDD
def map[K, V](f: Function1[T, (K, V)]): JavaPairRDD[K, V]
```

Instead, we define a hierarchy of Java Function classes that allow functions like map to be properly overloaded:

```scala
def map[R](f: Function[T, R]): JavaRDD[R]
def map[R](f: DoubleFunction[T]): JavaDoubleRDD
def map[K2, V2](f: PairFunction[T, K2, V2]): JavaPairRDD[K2, V2]
```

This works because `PairFunction`, `DoubleFunction`, and `Function` aren't subclasses of each other. Rather, this is the Function class hierarchy:

```scala
AbstractFunction1 (scala.runtime)
  WrappedFunction1 (org.apache.spark.api.java.function)
  DoubleFunction (org.apache.spark.api.java.function)
  PairFlatMapFunction (org.apache.spark.api.java.function)
  PairFunction (org.apache.spark.api.java.function)
  DoubleFlatMapFunction (org.apache.spark.api.java.function)
  Function (org.apache.spark.api.java.function)

AbstractFunction2 (scala.runtime)
  WrappedFunction2 (org.apache.spark.api.java.function)
  Function2 (org.apache.spark.api.java.function)
```
ClassManifests

Many Spark methods take implicit `ClassManifest` arguments that are used by the compiler to preserve type information for instantiating Arrays at runtime. To hide ClassManifests from users, the Java API generates dummy ClassManifests by casting `ClassManifest[Object]` to the appropriate type. Users of the Java API have to work with RDDs of Java objects, since Java generics can't be parameterized with generic types, so this works fine.

Methods with signatures like

```scala
def transformationName[R: ClassManifest](args)
```

are equivalent to

```scala
def transformationName[R](args)(implicit cm: ClassManifest[R])
```

and will produce Java methods that accept explicit ClassManifest objects, making them difficult to call from Java:

```scala
returnType transformationName(args, ClassManifest<? extends Object> evidence$1)
```

Here's an excerpt from `JavaRDDLike.java`, showing how we generate dummy manifests. In this example, the user-defined `PairFunction` has `keyType()` and `valueType()` methods to get its arguments' ClassManifests. To produce the `ClassManifest`, we just cast an `Object` class manifest:

```scala
/**
 * Return a new RDD by applying a function to all elements of this RDD.
 */
def map[K2, V2](f: PairFunction[T, K2, V2]): JavaPairRDD[K2, V2] = {
  def cm = implicitly[ClassManifest[AnyRef]].asInstanceOf[ClassManifest[Tuple2[K2, V2]]]
  new JavaPairRDD(rdd.map(f)(cm))(f.keyType(), f.valueType())
}
```

Scala -> Java Types

The Java API exposes standard Java collections types, instead of Scala ones. These substitutions include:

- `scala.collection.Map` -> `java.util.Map`
- `scala.collection.Seq` -> `java.util.List` (or in some cases, array)
- `scala.collection.Iterator` -> `java.util.Iterator`
- `scala.collection.mutable.Queue` -> `java.util.Queue`
- `scala.Option` -> `com.google.common.base.Optional`

Most of these conversions are performed using Scala's `JavaConverters` package.

Workarounds for compiler bugs

**SI-6050 / SI-3452: problems using implementation traits**

Many RDD transformations obey a "same-result-type" principle where they derive RDDs of the same type. For example, applying 'filter' to a JavaPairRDD should yield a new JavaPairRDD, while applying 'filter' to JavaDoubleRDD yields a JavaDoubleRDD.

Many operations in the Scala collections library obey this same principle. To avoid code duplication, the Scala collections define implementation traits, like `TraversableLike`, that contain methods that are parameterized by the implementor's type. This allows them to define methods like 'filter' with types like "function from a predicate defined on elements of this collection to a new concrete collection of the same type."

Ideally, we could use this same technique to factor out all of the common operations in JavaRDD, JavaPairRDD, and JavaDoubleRDD into the JavaRDDLike class. Unfortunately, this doesn't seem to work properly due to Scala compiler bugs; in our case, the compiler produced code that compiled fine but threw `NoSuchMethodErrors` at runtime (`SI-6050` is a minimal example that reproduces this problem).

A Scala pull request proposed a workaround for this bug. When compiling the code with the `-Ycheck:genjvm` flag, as suggested in that pull request, the compiler emits warnings when compiling the Scala code and compile-time typechecking errors when calling that Scala code from Java. This is an improvement, since it makes the problem explicit during compilation, but it doesn't fix the problem.

To work around this issue, methods that must return RDDs of the same type (like filter and distinct) are redundantly implemented in each Java*RDD class. As a result, these methods aren't defined in JavaRDDLike. This limits users' abilities to define functions that accept JavaRDDLike and perform transformations on them, since transformations like filter and distinct won't be available via that interface.

When these compiler bugs are fixed, we should refactor the code to remove these redundancies and define a proper implementation trait.
**SI-6057: NoSuchMethodErrors due to type parameter naming**

TODO: this section doesn't precisely state the conditions for this problem to occur.

In some rare cases, type parameters in different scopes can interfere in a way that causes problems when applying methods. For example, see SI-6057.

This affected a few methods in the Java API, including `JavaRDDLike.map(PairFunction)`.

## Keeping the Java API up-to-date

### Discovering which methods to add

To discover methods that need to be added to the Java API, Spark includes a `JavaAPICompletenessChecker` script that identifies public methods in the Scala API, searches for each method's equivalent Java method, and reports the ones that are missing. For example, see its output for Spark 0.7.3 (this output contains a few false-positives).

Internally, this script uses reflection to discover the Scala methods, custom code for parsing their type signatures, and a set of rewrite rules that convert Scala types into their Java API equivalents. For more details, see this pull request.

In Spark 0.7.*, this tool can be run with the command

```
./run spark.tools.JavaAPICompletenessChecker
```

In Spark 0.8.1+, it can be run with

```
./spark-class org.apache.spark.tools.JavaAPICompletenessChecker
```

### Guidelines for adding methods to the Java API

- **No user-facing ClassManifests**: Java API methods should never contain ClassManifest in their signatures, including type bounds.
- **Methods added to JavaRDD should also be added to Java(Double|Pair)RDD**: All of JavaRDD's methods obey the "same-result-type" principle (if they didn't, they should have been implemented in JavaRDDLike), so they should also be implemented in JavaDoubleRDD and JavaPairRDD.
- **Handling default parameters**: Java methods don't support optional parameters with default values. For Scala methods with optional arguments, like `def textFile(path: String, minSplits: Int = defaultMinSplits)`, you can either define two separate methods (1- and 2-parameter versions) or make the optional argument into a required one.
- **Unit tests**: for non-trivial additions that might be prone to compile-time or run-time problems if untested, please add new unit tests to JavaAPISuite.

### References and Resources

- Twitter's Scala School "Java + Scala" guide provides a nice overview of Java interoperability issues and explores some of the details of how Scala constructs are exposed in Java.