Shim

Less Frustration with Private Spark APIs

Chris Twiner

Table of contents

1. Shim - 0.0.1-SNAPSHOT

verage

Statement	0.00 %	Branch	0.00 %

1.1 Provides shims over private Apache Spark APIs

Using the private Spark APIs can allow for improved functionality (e.g. Frameless encoder derivation) and performance (e.g. Quality HOF compilation or Plan rules) but it comes at a cost: lots of workarounds to support different runtimes.

Shim aims to bring those workarounds in a separate runtime library shim and let the library authors focus on useful functionality.

Last update: February 27, 2024 13:33:44

Created: February 27, 2024 13:33:44

2. Getting Started

2.1 Building and Setting Up

2.1.1 Building The Library

- fork,
- use the Scala dev environment of your choice,
- or build directly using Maven

Building via commandline

For OSS versions (non Databricks runtime - dbr):

```
mvn --batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true -DskipTests install -P Spark321
```

but dbr versions will not be able to run tests from the command line (typically not an issue in intellij):

```
mvn --batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true -DskipTests clean install -P 10.4.dbr
```

2.1.2 Build tool dependencies

Shim is cross compiled for different versions of Spark, Scala *and* runtimes such as Databricks. The format for artifact's is:

```
shim_[compilation|runtime]_RUNTIME_SPARKCOMPATVERSION_SCALACOMPATVERSION-VERSION.jar
```

e.g.

```
shim_runtime_3.4.1.oss_3.4_2.12-0.1.3.jar
```

The build poms generate those variables via maven profiles, but you are advised to use properties to configure e.g. for Maven:

```
<dependency>
    <groupId>com.sparkutils</groupId>
    <artifactId>shim_runtime_${qualityRuntime}${sparkShortVersion}_${scalaCompatVersion}</artifactId>
    <version>${shimRuntimeVersion}</version>
</dependency>
```

The "compilation" artefacts are only needed if you rely on the internal apis used in them (e.g. Frameless doesn't, Quality does), you should attempt the use of runtime only first.

The full list of supported runtimes is below:

Spark Version	sparkShortVersion	qualityRuntime	scalaCompatVersion
2.4.6	2.4		2.11
3.0.3	3.0		2.12
3.1.3	3.1		2.12
3.1.3	3.1	9.1.dbr_	2.12
3.2.0	3.2		2.12
3.2.1	3.2	3.2.1.oss_	2.12
3.2.1	3.2	10.4.dbr_	2.12
3.3.2	3.3	3.3.2.oss_	2.12
3.3.2	3.3	11.3.dbr_	2.12
3.3.2	3.3	12.2.dbr_	2.12
3.3.2	3.3	13.1.dbr_	2.12
3.4.1	3.4	3.4.1.oss_	2.12
3.4.1	3.4	13.1.dbr_	2.12
3.4.1	3.4	13.3.dbr_	2.12
3.5.0	3.5	3.5.0.oss_	2.12
3.5.0	3.5	14.0.dbr_	2.12
3.5.0	3.5	14.3.dbr_	2.12

2.4 support is deprecated and will be removed in a future version. 3.1.2 support is replaced by 3.1.3 due to interpreted encoder issues.

Databricks 13.x support

13.0 also works on the 12.2.dbr_ build as of 10th May 2023, despite the Spark version difference. 13.1 requires its own version as it backports 3.5 functionality. The 13.1.dbr quality runtime build also works on 13.2 DBR. 13.3 has backports of 4.0 functionality which requires it's own runtime.

Databricks 14.x support

Due to back-porting of SPARK-44913 frameless 0.16.0 (the 3.5.0 release) is not binary compatible with 14.2 and above which has back-ported this 4.0 interface change. Similarly, 4.0 / 14.2 introduces a change in resolution so a new runtime version is required upon a potential fix for 44913 in frameless. 14.2 is not directly supported but has been tested and works with the 14.3 LTS release. Use the 14.3 version on 14.3, 14.0.dbr will not work

Developing for a Databricks Runtime

As there are many compatibility issues that Shim works around between the various Spark runtimes and their Databricks equivalents you will need to use two different runtimes when you do local testing (and of course you *should* do that):

```
<properties>
    <shimRuntimeVersion>0.1.3</shimRuntimeVersion>
    <shimRuntimeTest>3.4.1.oss_</shimRuntimeTest>
    <shimRuntimeDatabricks>13.1.dbr_</shimRuntimeDatabricks>
    <sparkShortVersion>3.4</sparkShortVersion>
    <scalaCompatVersion>2.12</scalaCompatVersion>
```

```
</properties>

<dependencies>
    <dependency>
        <groupId>com.sparkutils.</groupId>
        <artifactId>shim_runtime_${shimRuntimeTest}${sparkShortVersion}_${scalaCompatVersion}</artifactId>
        <version>${shimRuntimeVersion}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>com.sparkutils</groupId>
        <artifactId>shim_runtime_${shimRuntimeDatabricks}${sparkShortVersion}_${scalaCompatVersion}</artifactId>
        <version>${shimRuntimeVersion}</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

That horrific looking "." on the test groupId is required to get Maven 3 to use different versions many thanks for finding this Zheng.

It's safe to assume better build tools like gradle / sbt do not need such hackery.

The known combinations requiring this approach is below:

Spark Version	sparkShortVersion	qualityTestPrefix	qualityDatabricksPrefix	scalaCompatVersion
3.2.1	3.2	3.2.1.oss_	10.4.dbr_	2.12
3.3.0	3.3	3.3.0.oss_	11.3.dbr_	2.12
3.3.2	3.3	3.3.2.oss_	12.2.dbr_	2.12
3.4.1	3.4	3.4.1.oss_	13.1.dbr_	2.12
3.4.1	3.4	3.4.1.oss_	13.3.dbr_	2.12
3.5.0	3.5	3.5.0.oss_	14.0.dbr_	2.12
3.5.0	3.5	3.5.0.oss_	14.3.dbr_	2.12

Developing a library against internal APIs changed by Databricks

In this scenario, similar to Quality, it is assumed you want to use internal apis covered in the version specific "compilation" source. The approach taken is to force Databricks runtime compatible interfaces higher up in the classpath than the OSS equivalents (or indeed provide them where the OSS version doesn't have them - like backported code from as yet unreleased OSS versions).

This approach requires your build tool environment to support runtime ordering in the build, if it does then you may simply depend on the shim_compilation artefact as provided scope. The scala maven plugin does not maintain order from maven, which is fine for most usages, just not this one....

In order to support maven some config is needed - for a working complete build see Quality's - namely to use the sources classifier with the dependency and build helper plugins:

```
<project>
...

<dependencies>
    <dependency>
        <groupId>com.sparkutils</groupId>
        <artifactId>shim_compilation_${shimCompilationRuntime}_${sparkCompatVersion}_${scalaCompatVersion}</artifactId>
        <version>${shimCompilationVersion}</version>
        <scope>provided</scope>
        <classifier>sources</classifier>
        <exclusions>
            <exclusion>
                <groupId>org.apache.spark</groupId>
                <artifactId>spark-sql_${scalaCompatVersion}</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>com.sparkutils</groupId>
        <artifactId>shim_runtime_${shimRuntime}_${sparkCompatVersion}_${scalaCompatVersion}</artifactId>
        <version>${shimRuntimeVersion}</version>
    </dependency>
```

```
</dependencies>

<plugins>
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <version>${buildHelperPluginVersion}</version>
    <executions>
        <execution>
            <id>add-source</id>
            <phase>generate-sources</phase>
            <goals>
                <goal>add-source</goal>
            </goals>
            <configuration>
                <sources>
                    <source>src/main/scala</source>
                    <source>src/main/${profileDir}-scala</source>
                    <source>${project.build.directory}/shim_compilation_${shimCompilationRuntime}_${sparkCompatVersion}_${scalaCompatVersion}</source>
                </sources>
            </configuration>
        </execution>
...
    </executions>
</plugin>

<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <version>${dependencyPluginVersion}</version>
    <executions> <!-- maven scala plugin uses a set to store classpath, it doesn't follow maven's so we need to use the source -->
        <execution>
            <id>unpack</id>
            <phase>initialize</phase>
            <goals>
                <goal>unpack</goal>
            </goals>
            <configuration>
                <artifactItems>
                    <artifactItem>
                        <groupId>com.sparkutils</groupId>
                        <artifactId>shim_compilation_${shimCompilationRuntime}_${sparkCompatVersion}_${scalaCompatVersion}</artifactId>
                        <version>${shimCompilationVersion}</version>

                        <classifier>sources</classifier>
                        <type>jar</type>

                        <overWrite>true</overWrite>
                        <outputDirectory>${project.build.directory}/shim_compilation_${shimCompilationRuntime}_${sparkCompatVersion}_${scalaCompatVersion}</
outputDirectory>
                    </artifactItem>
                </artifactItems>
            </configuration>
        </execution>
    </executions>
</plugin>
</plugins>
</project>
```

This, at project initialization phase, downloads and unpacks the shim_compilation correct version to the target directory (the dependency plugin configuration) and then, at source generation phase, adds the directory as source with the build-helper plugin.

how will I know if I need this?

You'll get strange errors, incompatible implementations or linkages, missing methods etc. Hopefully they are already covered by the current code, if not raise an issue and we'll see if there is a solution to it.

Last update: February 27, 2024 13:33:44

Created: February 27, 2024 13:33:44

2.2 Running Shim on Databricks

In short, you should generally be able to use the OSS shim_runtime jars to build against and use the appropriate DBR version to run against - unless you are using internal apis that Databricks has changed then you need the shim_compilation approach (see here for details).

The shims are currently tested under Quality tests (which tests most, but not all Frameless encoding) and, as such, there is currently no direct tests for the shims themselves (this is tracked under #4)

Last update: February 27, 2024 13:33:44
Created: February 27, 2024 13:33:44

3. About

3.1 History

3.1.1 Why Shim?

In short - both the OSS and Vendor Spark teams want to innovate without needing to think about internal api changes that shouldn't affect 95% of the user base.

Frameless and Quality users likely feel the same, but changes made in Spark's internal apis can break both libraries through linkage errors, missing functions etc. and more insidious issues such as changes in decimal type handling.

The straw that broke the camels proverbial was a change to StaticInvoke made in Spark 4, which was reasonably added to the Databricks 14.2, the only 3.5 build. This change added a new parameter which works fine if you recompile but breaks at runtime if you built against 3.5. So although both Frameless and Quality work flawlessly on 14.0 and 14.1, they don't work on 14.2.

Shim aims to alleviate this pain by swapping out the different runtime implementations as needed.

Last update: February 27, 2024 13:33:44

Created: February 27, 2024 13:33:44

3.2 Changelog

0.0.1 20th March, 2024

#1 - Quality support

#2 - Frameless support

#3 - 14.3 LTS support

Last update: February 27, 2024 13:33:44
Created: February 27, 2024 13:33:44