Storage of pickled Scala signatures in class files

LAMP/EPFL — Gilles Dubochet — gilles.dubochet@epfl.ch

This document describes the method through which pickled Scala signatures are stored in Java class files. Its first part is destined to all interested person and should be accessible to most Scala developers. It gives an overview of pickled signatures, explains the Scala 2.8 implementation with respect to the legacy implementation and describes possible compatibility issues. The second part is primarily intended for developer of tools that have to read Scala signatures in class files.

You may safely ignore this document if you have not been using prerelease versions of Scala 2.8 and are not developing a tool that directly accesses Scala signatures.

1. What are pickled Scala signatures?

Pickled Scala signatures are compressed representations of the interfaces of Scala entities — classes, traits and objects.

The Java class file format itself can only represent Scala entities as simplified Java-like classes. This is sufficient for the JVM to run Scala programs. Other tools require full Scala signatures. For example, the compiler requires for type checking the full signatures of referenced libraries. You can get an idea of the information contained in class files and in pickled Scala signatures by decompiling class files of Scala programs using `javap` and `scalap` respectively.

The interface constructed during compilation of the original source file is compressed by the Scala pickler into a very efficient binary representation. This is then attached to class files using the method described below. Normal Java tools do not recognize this information, but Scala-specific tools can use it to retrieve full signatures.

2. How are pickled Scala signatures stored in class files?

Given a Scala signature, the Scala pickler returns an array of bytes encoding it. The Java class file format was not designed with these signatures in mind and does not directly support them. However, it does offer extension mechanisms that can be used to store pickled signatures in class files.

Prior to Scala 2.8, pickled signature bytes were stored as class file attributes called "`ScalaSig`". Attributes allow arbitrary content in their body so that the pickled bytes can simply be copied in the class file. Whilst parsing class files, a Java tool conveniently skips over any attribute it does not know.

The Scala 2.8 storage method uses another extension mechanism: annotations. An annotation is a special class whose instances can be attached to syntactic elements of a Java or Scala program, such as definitions. Some classes of annotations — runtime annotations — can be stored in Java class files with their constructor arguments.

The `scala.reflect.ScalaSignature` runtime annotation class contains the bytes of pickled Scala signature. Instead of `ScalaSig` attributes, the 2.8 Scala compiler adds `ScalaSignature` annotations to the declaration of all relevant top-level classes.

It should be noted that, for reasons of performance due to the curious design of storing annotation attributes in class files, `ScalaSignature` annotations do not directly store the pickled signatures as an array of bytes but instead encode it as a string. Sections 5 and 6 provide details on this encoding and on how to decode it.

3. Why does Scala 2.8 store signatures differently?

The legacy method for storing signatures as attributes is simultaneously more elegant, more compact (about 15%) and simpler than that using annotations. However, to access the pickled signature in attributes requires obtaining and parsing the entire class file. Because annotations are recognized by the JVM, the new method allows retrieving pickled signature bytes directly from within a running Scala program by using Java reflection.

The new method is part of an ongoing development to write a good Scala reflection library. Java reflection views Scala programs in a simplified Java-centric way. To give a Scala-centric view of the reflected program, information provided by Java reflection must be completed with information found in Scala signatures. The legacy storage method would have required all class files to be parsed again — the JVM has parsed them already and exposes most of their content through reflection. The new storage method allows accessing pickled Scala signatures in the same way than other reflective information is obtained, making the new Scala reflection library simpler and faster.

Whilst the Scala reflection library will not be part of the 2.8 release, it is desirable that existing 2.8 class files are compatible with the new reflection library when it becomes available. Furthermore, since 2.8 class files are incompatible with those of 2.7 in any case, changing the method of storing signatures in 2.8 will not require another binary-incompatible class file format change shortly after.

4. What impact will the change have?

For users who will switch from Scala 2.7 to 2.8 final directly:

This change does not modify the upgrade process: existing Scala 2.7 class files must be recompiles from source to be compatible with 2.8.

For users who have been using prerelease versions of Scala 2.8:

The Scala compiler and other tools (including the Scala plugins for IntelliJ IDEA, Eclipse and NetBeans) can understand both the old and new methods for signature storage. This means that class files compiled with older prerelease versions of Scala 2.8 will be compatible with the latest tools. Other tools that read Scala class files and that have not been updated for the new storage method may not work immediately (we are not aware of any such tools). Once updated, they will provide a similar level of compatibility to that of the Scala compiler.

For developers of tools that parse class files to read pickled Scala signatures:

You must modify your tool to support the new storage method. The specification of the method for signature storage is in section 5; recommendations for its implementation are in section 6. You can contact Gilles Dubochet for any questions regarding the specification or implementation.

5. Specification of the signatures storage method

The following class files contain pickled signatures:

- Class files of top-level Scala classes or traits. Their signatures define the corresponding class or trait, its companion object if it exists, as well as all of their inner entities.
- Class files of the static forwarder class for objects without a companion class. Their signatures define the object an all of its inner entities.
- Class files of package classes. Their signatures define only non-entity members of the package (that is: `val`, `var`, `def` and `type` declarations). Entity members are defined in separate class files.

Class files that contain a pickled signature have the following two properties:

1. They contain a class file attribute (class file 5 spec. §4.8) named "`ScalaSig`"
2. Their top-level class is annotated with a `RuntimeVisibleAnnotation` (class file 5 spec. §4.8.15) of type "`scala.reflect.ScalaSignature`" or "`ScalaLongSignature`".

Note that class files that correspond to modules (name ends with "`$`") or to inner entities (name contains an inner "`$`") never contain a signature. Instead they contain an empty class file attribute named "`Scala`" to indicate that they are defined as part of a pickled signature in another class file.

The `ScalaSig` attribute

The `ScalaSig` attribute serves two purposes:

1. To mark the class file as containing a `ScalaSignature` annotation without requiring that all annotations be parsed.
2. So that previous versions of the Scala compiler or tools can read the version of the signature and correctly report that they are not compatible with it.

The `ScalaSig` attribute contains the first 3 elements of a pickled Scala signature, namely:

1. the major version number,
2. the minor version number,
3. the number of entries, which is always equal to "0" (as the actual entries are defined in the `ScalaSignature` annotation, described below).

The specification of pickled Scala signatures describes the exact encoding of these numbers. If both version numbers are inferior to `0x7f` (127), the length of the `ScalaSig` attribute info is 3 bytes. In any case, the last byte is equal to `0x00`.

The `ScalaSignature` or `ScalaLongSignature` annotation

The annotation is included as a `RuntimeVisibleAnnotation` attached to the top-level class in the class file. In most cases, the annotation is of class `scala.reflect.ScalaSignature`. It is instantiated with "`bytes`", a string that encodes the bytes of a pickled Scala signature.

If the length of the string that encodes the bytes of a pickled Scala signature exceeds 65535 characters, the usual annotation is replaced by a `ScalaLongSignature` annotation. It is instantiated with an array of strings which concatenation encodes the bytes of a pickled Scala signature. The last string in the array has a length of at most 65535 characters; all other strings have a length of exactly 65535 characters.

The bytes of a Scala signature are encoded as a UTF-8 string because Java class files have a very inefficient way of storing arrays of bytes in annotation (1 byte in the array is stored as 8 bytes in the class file). Still, UTF-8 strings encode one character as one byte only if its value is between `0x01` and `0x7f`. In order to minimize the number of UTF-8 bytes required to encode the pickled signature, the latter is encoded as follows:

1. The original 8-bit bytes are split into bytes of 7 bits. That means that the first 7-bit byte will contain the first 7 bits of the first 8-bit byte. The second 7-bit byte will contain the last bit of the first 8-bit byte as well as the first 6 bits of the second 8-bit byte. And so on. The new bytes obtained by the 7-bit transformation are in the range `0x00`–`0x7f`.
2. Each new byte is incremented by one (within 7 bits), so that `0x00` becomes `0x01` and `0x7f` become `0x00`. This is done so that value `0x00` that will be encoded as 2 bytes (see below) corresponds to the less common value `0x7f` of the 7-bit encoding.

3. The 7-bit bytes are then encoded in modified UTF-8 format (used by Java class files) as if each 7-bit byte were a character. The effect of this encoding is to replaces any 7-bit byte which value is `0x00` by two bytes: `0xc0` and `0x80`.

6. Recommendations for the implementation of tools

These recommendations mostly relate to developers of tools that read Scala class files. Such tool may include IDEs, debuggers, reflection tools and libraries, etc.

Use ByteCodecs to decode UTF-8 strings to bytes

Object `scala.reflect.generic.ByteCodecs` is part of the standard Scala library and provides utility methods to encode and decode arrays of bytes into UTF-8 string, as described in the specification above.

- Method "`encode`" takes an array of 8-bit bytes and returns a new array of bytes encoding the corresponding UTF-8 string. The resulting bytes can directly be written in a class file or transformed into a String object to be used to instantiate the annotation by using the `String(bytes)` constructor.
- Method "`decode`" takes an array of bytes encoding a UTF-8 string. Once decode returns, the array passed to the method has been changed and starts with the bytes decoded from the UTF-8 bytes. The return value of "`decode`" is the number of bytes in the decoded array that are relevant (there are less decoded bytes than encoded bytes).

Parse the ScalaSig attribute if the annotation is missing

So that you tool is backward compatible, you should continue to support class files in which the pickled signature is stored as an attribute.

1. The `ScalaSig` attribute is present in all Scala class files containing a signature, irrespective of storage method. Read the third number of the pickled signature (by using a `scala.reflect.generic.PickleBuffer`, or simply by accessing `bytes(2)`)
2. If the third number is `0x00` (the signature contains no entries), the class file is encoded in the new format: you must read the signature stored in the annotation.
3. If the third number is greater than `0x00`, the class file is encoded in the old format: you can build the signature from the bytes in the attribute and need not check for an annotation.

Consider using a Java ClassLoader to access pickled Scala signatures

Your tool may no longer need to parse class files. It is now recommended that tools that need to access pickled signatures use the JVM's reflection infrastructure.

Implementation is to be described…