JCC Features

Table of contents

Copyright © 2009 The Apache Software Foundation. All rights reserved.

Warning:
Before calling any PyLucene API that requires the Java VM, start it by calling `initVM(classpath, ...)`. More about this function in here.

1. Installing JCC

JCC is a Python extension written in Python and C++. It requires a Java Runtime Environment (JRE) to operate as it uses Java's reflection APIs to do its work. It is built and installed via `distutils` or `setuptools`.

See installation for more information and operating system specific notes.

2. Invoking JCC

JCC is installed as a package and how to invoke it depends on the Python version used:

- python 2.7: `python -m jcc`
- python 2.6: `python -m jcc.__main__`
- python 2.5: `python -m jcc`
- python 2.4:
  - no setuptools: `python` *`site-packages`*`/jcc/__init__.py`
  - with setuptools: `python` *`site-packages`*/*`jcc egg directory`*`/jcc/__init__.py`
- python 2.3: `python` *`site-packages`*/*`jcc egg directory`*`/jcc/__init__.py`

3. Generating C++ and Python wrappers with JCC

JCC started as a C++ code generator for hiding the gory details of accessing methods and fields on Java classes via Java's Native Invocation Interface. These C++ wrappers make it possible to access a Java object as if it was a regular C++ object very much like GCJ's CNI interface.

It then became apparent that JCC could also generate the C++ wrappers for making these classes available to Python. Every class that gets thus wrapped becomes a CPython type.

JCC generates wrappers for all public classes that are requested by name on the command line or via the `--jar` command line argument. It generates wrapper methods for all public methods and fields on these classes whose return type and parameter types are found in one of the following ways:

Copyright © 2009 The Apache Software Foundation. All rights reserved.

- the type is one of the requested classes
- the type is one of the requested classes' superclass or implemented interfaces
- the type is available from one of the packages listed via the `--package` command line argument

Overloaded methods are supported and are selected at runtime on the basis of the type and number of arguments passed in.

JCC does not generate wrappers for methods or fields which don't satisfy these requirements. Thus, JCC can avoid generating code for runaway transitive closures of type dependencies.

JCC generates property accessors for a property called *`field`* when it finds Java methods named `set`*`Field`*`(value)`, `get`*`Field`*`()` or `is`*`Field`*`()`.

The C++ wrappers are declared in a C++ namespace structure that mirrors the Java classes' Java packages. The Python types are declared in a flat namespace at the top level of the resulting Python extension module.

JCC's command-line arguments are best illustrated via the PyLucene example:

```
    $ python -m jcc           # run JCC to wrap
        --jar lucene.jar      # all public classes in the lucene jar file
        --jar analyzers.jar   # and the lucene analyzers contrib package
        --jar snowball.jar    # and the snowball contrib package
        --jar highlighter.jar # and the highlighter contrib package
        --jar regex.jar       # and the regex search contrib package
        --jar queries.jar     # and the queries contrib package
        --jar extensions.jar  # and the Python extensions package
        --package java.lang   # including all dependencies found in the
                              # java.lang package
        --package java.util   # and the java.util package
        --package java.io     # and the java.io package
          java.lang.System    # and to explicitely wrap java.lang.System
          java.lang.Runtime   # as well as java.lang.Runtime
          java.lang.Boolean   # and java.lang.Boolean
          java.lang.Byte      # and java.lang.Byte
          java.lang.Character # and java.lang.Character
          java.lang.Integer   # and java.lang.Integer
          java.lang.Short     # and java.lang.Short
          java.lang.Long      # and java.lang.Long
          java.lang.Double    # and java.lang.Double
          java.lang.Float     # and java.lang.Float
          java.text.SimpleDateFormat
                              # and java.text.SimpleDateFormat
          java.io.StringReader
                              # and java.io.StringReader
          java.io.InputStreamReader
                              # and java.io.InputStreamReader
          java.io.FileInputStream
                              # and java.io.FileInputStream
```

Copyright © 2009 The Apache Software Foundation. All rights reserved.

```
          java.util.Arrays    # and java.util.Arrays
        --exclude org.apache.lucene.queryParser.Token
                              # while explicitely not wrapping
                              # org.apache.lucene.queryParser.Token
        --exclude org.apache.lucene.queryParser.TokenMgrError
                              # nor
org.apache.lucene.queryParser.TokenMgrError
        --exclude org.apache.lucene.queryParser.ParseException
                              #
nor.apache.lucene.queryParser.ParseException
        --python lucene       # generating Python wrappers into a module
                              # called lucene
        --version 2.4.0       # giving the Python extension egg version
2.4.0
        --mapping org.apache.lucene.document.Document
                  'get:(Ljava/lang/String;)Ljava/lang/String;'
                              # asking for a Python mapping protocol
wrapper
                              # for get access on the Document class by
                              # calling its get method
        --mapping java.util.Properties
                  'getProperty:(Ljava/lang/String;)Ljava/lang/String;'
                              # asking for a Python mapping protocol
wrapper
                              # for get access on the Properties class by
                              # calling its getProperty method
        --sequence org.apache.lucene.search.Hits
                   'length:()I'
                   'doc:(I)Lorg/apache/lucene/document/Document;'
                              # asking for a Python sequence protocol
wrapper
                              # for length and get access on the Hits class
by
                              # calling its length and doc methods
        --files 2             # generating all C++ classes into about 2
.cpp
                              # files
        --build               # and finally compiling the generated C++
code
                              # into a Python egg via setuptools - when
                              # installed - or a regular Python extension
via
                              # distutils or setuptools otherwise
        --module collections.py
                              # copying the collections.py module into the
egg
        --install             # installing it into Python's site-packages
                              # directory.
```

There are limits to both how many files can fit on the command line and how large a C++ file the C++ compiler can handle. By default, JCC generates one large C++ file containing the source code for all wrapper classes.

Copyright © 2009 The Apache Software Foundation. All rights reserved.

Using the `--files` command line argument, this behaviour can be tuned to workaround various limits:
for example:

- to break up the large wrapper class file into about 2 files:
  `--files 2`
- to break up the large wrapper class file into about 10 files:
  `--files 10`
- to generate one C++ file per Java class wrapped:
  `--files separate`

The `--prefix` and `--root` arguments are passed through to `distutils`' `setup()`.

4. Classpath considerations

When generating wrappers for Python, the JAR files passed to JCC via `--jar` are copied into the resulting Python extension egg as resources and added to the extension module's `CLASSPATH` variable. Classes or JAR files that are required by the classes contained in the argument JAR files need to be made findable via JCC's `--classpath` command line argument. At runtime, these need to be appended to the extension's `CLASSPATH` variable before starting the VM with `initVM(CLASSPATH)`.

To have such required jar files also automatically copied into resulting Python extension egg and added to the classpath at build and runtime, use the `--include` option. This option works like the `--jar` option except that no wrappers are generated for the classes contained in them unless they're explicitely named on the command line.

When more than one JCC-built extension module is going to be used in the same Python VM and these extension modules share Java classes, only one extension module should be generated with wrappers for these shared classes. The other extension modules must be built by importing the one with the shared classes by using the `--import` command line parameter. This ensures that only one copy of the wrappers for the shared classes are generated and that they are compatible among all extension modules sharing them.

5. Using distutils vs setuptools

By default, when building a Python extension, if `setuptools` is found to be installed, it is used over `distutils`. If you want to force the use of `distutils` over `setuptools`, use the `--use-distutils` command line argument.

6. Distributing an egg

Copyright © 2009 The Apache Software Foundation. All rights reserved.

The `--bdist` option can be used to ask JCC to invoke `distutils` with `bdist` or `setuptools` with `bdist_egg`. If `setuptools` is used, the resulting egg has to be installed with the `easy_install` installer which is normally part of a Python installation that includes `setuptools`.

7. JCC's runtime API functions

JCC includes a small runtime component that is compiled into any Python extension it produces.

This runtime component makes it possible to manage the Java VM from Python. Because a Java VM can be configured with a myriad of options, it is not automatically started when the resulting Python extension module is loaded into the Python interpreter.

Instead, the `initVM()` function must be called from the main thread before using any of the wrapped classes. It takes the following keyword arguments:

- `classpath`
  A string containing one or more directories or jar files for the Java VM to search for classes. Every Python extension produced by JCC exports a `CLASSPATH` variable that is hardcoded to the jar files that it was produced from. A copy of each jar file is installed as a resource file with the extension when JCC is invoked with the `--install` command line argument. This parameter is optional and defaults to the `CLASSPATH` string exported by the module `initVM` is imported from.

  ```
  >>> import lucene
  >>> lucene.initVM(classpath=lucene.CLASSPATH)
  ```
- `initialheap`
  The initial amount of Java heap to start the Java VM with. This argument is a string that follows the same syntax as the similar `-Xms` java command line argument.

  ```
  >>> import lucene
  >>> lucene.initVM(initialheap='32m')
  >>> lucene.Runtime.getRuntime().totalMemory()
  33357824L
  ```
- `maxheap`
  The maximum amount of Java heap that could become available to the Java VM. This argument is a string that follows the same syntax as the similar `-Xmx` java command line argument.
- `maxstack`
  The maximum amount of stack space that available to the Java VM. This argument is a string that follows the same syntax as the similar `-Xss` java command line argument.

Copyright © 2009 The Apache Software Foundation. All rights reserved.

- `vmargs`
  A string of comma separated additional options to pass to the VM startup rountine. These are passed through as-is. For example:

```
            >>> import lucene
            >>>
lucene.initVM(vmargs='-Xcheck:jni,-verbose:jni,-verbose:gc')
```

The `initVM()` and `getVMEnv()` functions return a JCCEnv object that has a few utility methods on it:

- `attachCurrentThread(name, asDaemon)`
  Before a thread created in Python or elsewhere but not in the Java VM can be used with the Java VM, this method needs to be invoked. The two arguments it takes are optional and self-explanatory.
- `detachCurrentThread()` The opposite of `attachCurrentThread()`. This method should be used with extreme caution as Python's and java VM's garbage collectors may use a thread detached too early causing a system crash. The utility of this method seems dubious at the moment.

There are several differences between JNI's `findClass()` and Java's `Class.forName()`:

- className is a '/' separated string of names
- the class loaders are different, `findClass()` may find classes that `Class.forName()` won't.

For example:

```
        >>> from lucene import *
        >>> initVM(CLASSPATH)
        >>> findClass('org/apache/lucene/document/Document')
        <Class: class org.apache.lucene.document.Document>
        >>> Class.forName('org.apache.lucene.document.Document')
        Traceback (most recent call last):
          File "<stdin>", line 1, in <module>
        lucene.JavaError: java.lang.ClassNotFoundException:
                          org/apache/lucene/document/Document
        >>> Class.forName('java.lang.Object')
        <Class: class java.lang.Object>
```

8. Type casting and instance checks

Many Java APIs are declared to return types that are less specific than the types actually returned. In Java 1.5, this is worked around with type parameters. JCC generates code to

Copyright © 2009 The Apache Software Foundation. All rights reserved.

heed type parameters unless the `--no-generics` is used. See next section for details on Java generics support.

In C++, casting the object into its actual type is supported via the regular C casting operator.

In Python each wrapped class has a class method called `cast_` that implements the same functionality.

Similarly, each wrapped class has a class method called `instance_` that tests whether the wrapped java instance is of the given type. For example:

```
if BooleanQuery.instance_(query):
    booleanQuery = BooleanQuery.cast_(query)

print booleanQuery.getClauses()
```

9. Handling generic classes

Java 1.5 added support for parameterized types. JCC generates code to heed type parameters unless the `--no-generics` command line parameter is used. Java type parameterization is a runtime feature. The same class is used for all its parameterizations. Similarly, JCC wrapper objects all use the same class but store type parameterizations on instances and make them accessible as a tuple via the `parameters_` property.

For example, an `ArrayList<Document>` instance, has `(<type 'Document'>,)` for `parameters_` and its `get()` method uses that type parameter to wrap its return values.

To allocate an instance of a generic Java class with specific type parameters use the `of_()` method. This method accepts one or more Python wrapper classes to use as type parameters. For example, `java.util.ArrayList<E>` is declared to accept one type parameter. Its wrapper's `of_()` method hence accepts one parameter, a Python class, to use as type parameter for the return type of its `get()` method, among others:

```
>>> a = ArrayList().of_(Document)
>>> a
<ArrayList: []>
>>> a.parameters_
(<type 'Document'>,)
>>> a.add(Document())
True
>>> a.get(0)
<Document: Document<>>
```

The use of type parameters is, of course, optional. A generic Java class can still be used as before, without type parameters. Downcasting from `Object` is then necessary:

Copyright © 2009 The Apache Software Foundation. All rights reserved.

```
>>> a = ArrayList()
>>> a
<ArrayList: []>
>>> a.parameters_
(None,)
>>> a.add(Document())
True
>>> a.get(0)
<Object: Document<>>
>>> Document.cast_(a.get(0))
<Document: Document<>>
```

10. Handling arrays

Java arrays are wrapped with a C++ JArray template. The `[]` is available for read access. This template, `JArray<T>`, accomodates all java primitive types, `jstring`, `jobject` and wrapper class arrays.

Java arrays are returned to Python in a `JArray` wrapper instance that implements the Python sequence protocol. It is possible to change an array's elements but not to change an array's size.

To convert a char array to a Python string use a `''.join(array)` construct.

Any Java method expecting an array can be called with the corresponding sequence object from python.

To instantiate a Java array from Python, use one of the following forms:

```
>>> array = JArray('int')(size)
# the resulting Java int array is initialized with zeroes

>>> array = JArray('int')(sequence)
# the sequence must only contain ints
# the resulting Java int array contains the ints in the sequence
```

Instead of `'int'`, you may also use one of `'object'`, `'string'`, `'bool'`, `'byte'`, `'char'`, `'double'`, `'float'`, `'long'` and `'short'` to create an array of the corresponding type.

Because there is only one wrapper class for object arrays, the `JArray('object')` type's constructor takes a second argument denoting the class of the object elements. This argument is optional and defaults to `Object`.

As with the `Object` types, the `JArray` types also include a `cast_` method. This method becomes useful when the array returned to Python is wrapped as a plain `Object`. This is the

Copyright © 2009 The Apache Software Foundation. All rights reserved.

case, for example, with nested arrays since there is no distinct Python type for every different java object array class - all java object arrays are wrapped by `JArray('object')`. For example:

```
        # cast obj to an array of ints
        >>> JArray('int').cast_(obj)
        # cast obj to an array of Document
        >>> JArray('object').cast_(obj, Document)
```

In both cases, the java type of obj must be compatible with the array type it is being cast to.

```
        # using nested array:

        >>> d = JArray('object')(1, Document)
        >>> d[0] = Document()
        >>> d
        JArray<object>[<Document: Document<>>]
        >>> d[0]
        <Document: Document<>>
        >>> a = JArray('object')(2)
        >>> a[0] = d
        >>> a[1] = JArray('int')([0, 1, 2])
        >>> a
        JArray<object>[<Object:
[Lorg.apache.lucene.document.Document;@694f12>, <Object: [I@234265>]
        >>> a[0]
        <Object: [Lorg.apache.lucene.document.Document;@694f12>
        >>> a[1]
        <Object: [I@234265>
        >>> JArray('object').cast_(a[0])[0]
        <Object: Document<>>
        >>> JArray('object').cast_(a[0], Document)[0]
        <Document: Document<>>
        >>> JArray('int').cast_(a[1])
        JArray<int>[0, 1, 2]
        >>> JArray('int').cast_(a[1])[0]
        0
```

To verify that a Java object is of a given array type, use the `instance_()` method available on the array type. This is not the same as verifying that it is assignable with elements of a given type. For example, using the arrays created above:

```
        # is d array of Object ? are d's elements of type Object ?
        >>> JArray('object').instance_(d)
        True

        # can it receive Object instances ?
        >>> JArray('object').assignable_(d)
        False
```

Copyright © 2009 The Apache Software Foundation. All rights reserved.

```
        # is it array of Document ? are d's elements of type Document ?
        >>> JArray('object').instance_(d, Document)
        True

        # is it array of Class ? are d's elements of type Class ?
        >>> JArray('object').instance_(d, Class)
        False

        # can it receive Document instances ?
        >>> JArray('object').assignable_(d, Document)
        True
```

11. Exception reporting

Exceptions that occur in the Java VM and that escape to C++ are reported as a `javaError` C++ exception. When using Python wrappers, the C++ exceptions are handled and reported with Python exceptions. When using C++ only, failure to handle the exception in your C++ code will cause the process to crash.

Exceptions that occur in the Java VM and that escape to the Python VM are reported with a `JavaError` python exception object. The `getJavaException()` method can be called on `JavaError` objects to obtain the original java exception object wrapped as any other Java object. This Java object can be used to obtain a Java stack trace for the error, for example.

Exceptions that occur in the Python VM and that escape to the Java VM, as for example can happen in Python extensions (see topic below) are reported to the Java VM as a `RuntimeException` or as a `PythonException` when using shared mode. See installation instructions for more information about shared mode.

12. Writing Java class extensions in Python

JCC makes it relatively easy to extend a Java class from Python. This is done via an intermediary class written in Java that implements a special method called `pythonExtension()` and that declares a number of native methods that are to be implemented by the actual Python extension.

When JCC sees these special extension java classes it generates the C++ code implementing the native methods they declare. These native methods call the corresponding Python method implementations passing in parameters and returning the result to the Java VM caller.

For example, to implement a Lucene analyzer in Python, one would implement first such an extension class in Java:

Copyright © 2009 The Apache Software Foundation. All rights reserved.

```
    package org.apache.pylucene.analysis;

    import org.apache.lucene.analysis.Analyzer;
    import org.apache.lucene.analysis.TokenStream;
    import java.io.Reader;

    public class PythonAnalyzer extends Analyzer {
        private long pythonObject;

        public PythonAnalyzer()
        {
        }

        public void pythonExtension(long pythonObject)
        {
            this.pythonObject = pythonObject;
        }
        public long pythonExtension()
        {
            return this.pythonObject;
        }

        public void finalize()
            throws Throwable
        {
            pythonDecRef();
        }

        public native void pythonDecRef();
        public native TokenStream tokenStream(String fieldName, Reader
reader);
    }
```

The `pythonExtension()` methods is what makes this class recognized as an extension class by JCC. They should be included verbatim as above along with the declaration of the `pythonObject` instance variable.

The implementation of the native `pythonDecRef()` method is generated by JCC and is necessary because it seems that `finalize()` cannot itself be native. Since an extension class wraps the Python instance object it's going to be calling methods on, its ref count needs to be decremented when this Java wrapper class disappears. A declaration for `pythonDecRef()` and a `finalize()` implementation should always be included verbatim as above.

Really, the only non boilerplate user input is the constructor of the class and the other native methods, `tokenStream()` in the example above.

The corresponding Python class(es) are implemented as follows:

```
        class _analyzer(PythonAnalyzer):
```

Copyright © 2009 The Apache Software Foundation. All rights reserved.

```
            def tokenStream(_self, fieldName, reader):
                class _tokenStream(PythonTokenStream):
                    def __init__(self_):
                        super(_tokenStream, self_).__init__()
                        self_.TOKENS = ["1", "2", "3", "4", "5"]
                        self_.INCREMENTS = [1, 2, 1, 0, 1]
                        self_.i = 0
                        self_.posIncrAtt =
self_.addAttribute(PositionIncrementAttribute.class_)
                        self_.termAtt =
self_.addAttribute(TermAttribute.class_)
                        self_.offsetAtt =
self_.addAttribute(OffsetAttribute.class_)
                    def incrementToken(self_):
                        if self_.i == len(self_.TOKENS):
                            return False
                        self_.termAtt.setTermBuffer(self_.TOKENS[self_.i])
                        self_.offsetAtt.setOffset(self_.i, self_.i)
self_.posIncrAtt.setPositionIncrement(self_.INCREMENTS[self_.i])
                        self_.i += 1
                        return True
                    def end(self_):
                        pass
                    def reset(self_):
                        pass
                    def close(self_):
                        pass
                return _tokenStream()
```

When an `__init__()` is declared, `super()` must be called or else the Java wrapper class will not know about the Python instance it needs to invoke.

When a java extension class declares native methods for which there are public or protected equivalents available on the parent class, JCC generates code that makes it possible to call `super()` on these methods from Python as well.

There are a number of extension examples available in PyLucene's test suite and samples.

13. Embedding a Python VM in a Java VM

Using the same techniques used when writing a Python extension of a Java class, JCC may also be used to embed a Python VM in a Java VM. Following are the steps and constraints to follow to achieve this:

- JCC must be built in shared mode. See installation instructions for more information about shared mode. Note that for this use on Mac OS X, JCC must also be built with the link flags `"-framework", "Python"` in the `LFLAGS` value.
- As described in the previous section, define one or more Java classes to be "extended" from Python to provide the implementations of the native methods declared on them.

Copyright © 2009 The Apache Software Foundation. All rights reserved.

Instances of these classes implement the bridges into the Python VM from Java.

- The `org.apache.jcc.PythonVM` Java class is going be used from the Java VM's main thread to initialize the embedded Python VM. This class is installed inside the JCC egg under the `jcc/classes` directory and the full path to this directory must be on the Java `CLASSPATH`.
- The JCC egg directory contains the JCC shared runtime library - not the JCC Python extension shared library - but a library called `libjcc.dylib` on Mac OS X, `libjcc.so` on Linux or `jcc.dll` on Windows. This directory must be added to the Java VM's shared library path via the `-Djava.library.path` command line parameter.
- In the Java VM's main thread, initialize the Python VM by calling its static `start()` method passing it a Python program name string and optional start-up arguments in a string array that will be made accessible in Python via `sys.argv`. Note that the program name string is purely informational, and is not used by the `start()` code other than to initialize that Python variable. This method returns the singleton PythonVM instance to be used in this Java VM. `start()` may be called multiple times; it will always return the same singleton instance. This instance may also be retrieved at any later time via the static `get()` method defined on the `org.apache.jcc.PythonVM` class.
- Any Java VM thread that is going to be calling into the Python VM should start with acquiring a reference to the Python thread state object by calling `acquireThreadState()` method on the Python VM instance. It should then release the Python thread state before terminating by calling `releaseThreadState()`. Calling these methods is optional but strongly recommended as it ensures that Python is not creating and throwing away a thread state everytime the Python VM is entered and exited from a given Java VM thread.
- Any Java VM thread may instantiate a Python object for which an extension class was defined in Java as described in the previous section by calling the `instantiate()` method on the PythonVM instance. This method takes two string parameters, the name of the Python module and the name of the Python class to import and instantiate from it. The `__init__()` constructor on this class must be callable without any parameters and, if defined, must call `super()` in order to initialize the Java side. The `instantiate()` method is declared to return `java.lang.Object` but the return value is actually an instance of the Java extension class used and must be downcast to it.

14. Pythonic protocols

When generating wrappers for Python, JCC attempts to detect which classes can be made iterable:

- When a class declares to implement `java.lang.Iterable`, JCC makes it iterable from Python.

Copyright © 2009 The Apache Software Foundation. All rights reserved.

- When a Java class declares a method called `next()` with no arguments returning an object type, this class is made iterable. Its `next()` method is assumed to terminate iteration by returning `null`.

JCC generates a Python mapping get method for a class when requested to do so via the `--mapping` command line option which takes two arguments, the class to generate the mapping get for and the Java method to use. The method is specified with its name followed by ':' and its Java signature.

For example, `System.getProperties()['java.class.path']` is made possible by:

```
        --mapping java.util.Properties
                  'getProperty:(Ljava/lang/String;)Ljava/lang/String;'
                              # asking for a Python mapping protocol
wrapper
                              # for get access on the Properties class by
                              # calling its getProperty method
```

JCC generates Python sequence length and get methods for a class when requested to do so via the `--sequence` command line option which takes three arguments, the class to generate the sequence length and get for and the two java methods to use. The methods are specified with their name followed by ':' and their Java signature. For example:

```
      for i in xrange(len(hits)):
          doc = hits[i]
          ...
```

is made possible by:

```
        --sequence org.apache.lucene.search.Hits
                   'length:()I'
                   'doc:(I)Lorg/apache/lucene/document/Document;'
```

Copyright © 2009 The Apache Software Foundation. All rights reserved.