Some Jython Internals

Frank Wierzbicki

Jython Core

- Antlr 3.1 for parsing
- AST nearly identical to CPython using _ast
- ASM 3.1 for emitting bytecodes
- Libraries written in Java and Python
- Using jna-posix from the JRuby project for platform specific features

Jython Roadmap

- Currently 2.5 alphas are out
- 2.5 Beta out in October
- 2.5 release in January
- Start 2.6 and 3.0 in January

Jython 3.0

- As with CPython 3.0 some evolution will be permitted
- Some backwards incompatibility allowed
- Will be able to *require* JDK 1.7
- Lots of older code can go
- We can take full advantage of whatever parts of MLVM make it into JDK 1.7!

Jython Threading

- Probably over-synchronized
- Multi-threaded programs written for C implementation of Python often rely on GIL
- Need a Python Memory Model analogous to Java Memory Model
- Define behaviors based on happens-before
- Jython core dicts now use CHM

Import in Jython

- Modules compiled on first import
- Python has user definable “import hooks” that allow special behavior upon import
- It is at import time that Java classes are proxied with “JavaImporter”
- import hooks allow all sorts of user manipulation at import time

Jython Compiler

- Always compiled to bytecode even eval()
- Not always saved to disk
- Table of offsets pointing to functions
- Uses a ClassLoader per generated class at runtime

vikings.py

```
class Vikings(object):

    def __init__(self):
        self.yell("spam!", 3)
        self.yell("eggs!", 1)
        self.yell("spam!", 1)

    def yell(self, word, times):
        for time in xrange(times):
            print word

Vikings()
```

function signatures

```
public PyObject f$0(PyFrame);
public PyObject Vikings$1(PyFrame);
public PyObject __init__$2(PyFrame);
public PyObject yell$3(PyFrame);
public vikings$py(String);
public PyCode getMain();
public static void main(String[]);
public PyObject call_function(int, PyFrame);
```

call_function detail

```
public PyObject call_function(int, PyFrame);
    0:aload_0
    1:aload_2
    2:iload_1
    3:tableswitch{ //0 to 3
     0: 32;
     1: 36;
     2: 40;
     3: 44;
     default: 48 }
    32: invokevirtual #190; //Method f$0
    35: areturn
    36: invokevirtual #192; //Method Vikings$1
    39: areturn
    40: invokevirtual #194; //Method __init__$2
    43: areturn
    44: invokevirtual #196; //Method yell$3
    47: areturn
    48: aconst_null
    49: areturn
```

PyFrame

- Jython currently creates a custom Python-specific frame for each call.
- Python allows a program to grab, inspect, and manipulate the current frame.
- However this is considered an "implementation detail" of the C version

The Future of PyFrame

- We are experimenting with a number of adjustments to our use of PyFrame
- In many cases we can prove that a particular module has no need for frames (no “from x import *”, etc)
- Also we can think about fast path/slow path where “slow paths” occur when frames are manipulated

Builtin Classes

- Written in Java
- Classes and methods are annotated
- Annotations are used to generate companion bytecodes at build time
- May be able to use these same annotations to emit in an invokedynamic style

PyInteger.java excerpt

```
@ExposedType(name = "int")
public class PyInteger extends PyObject {
    ...
    public String toString() {
        return int_toString();
    }

    @ExposedMethod(names = {"__str__", "__repr__"})
    final String int_toString() {
        return Integer.toString(getValue());
    }
    ...
}
```

PyInteger$PyExposer

```
public PyInteger$PyExposer();
  Code:
    ...
    7:   ldc #13; //int 45
    9:   anewarray   #15; //class PyBuiltinMethod
    12:  astore_1
    13:  aload_1
    14:  ldc #16; //int 0
    16:  new #18; //PyInteger$int_toString_exposer
    19:  dup
    20:  ldc #20; //String __str__
    22:  invokespecial   #23; //Method PyInteger
$int_toString_exposer."<init>":(LString;)V
    25:  aastore
    ...
```

PyInteger$int_toString_exposer __call__ method

```
public PyObject __call__();
  Code:
    0:  aload_0
    1:  getfield #30; //Field self:LPyObject;
    4:  checkcast #32; //class PyInteger
    7:  invokevirtual #36; //Method PyInteger.int_toString:
()LString;
    10: invokestatic #42; //Method Py.newString:
(LString;)LPyString;
    13: areturn
```

Derived Builtins

- Each Builtin type also has a special generated Java class that handles subclasses
- Very repetitious and verbose code.
- May be another good place for method handles and anonymous classes

PyIntegerDerived

```
public PyString __repr__() {
    PyType self_type=getType();
    PyObject impl=self_type.lookup("__repr__");
    if (impl!=null) {
        PyObject res=impl.__get__(this,self_type).__call__();
        if (res instanceof PyString)
            return(PyString)res;
        throw Py.TypeError("__repr__ returned non-string");
    }
    return super.__repr__();
}
```

Calling Java from Jython

- Proxies generated to wrap Java code
- arguments coerced into Object[] with adapter classes
- Java code called through proxy
- return value wrapped in PyObject

HelloWorld.java

```
package com.foo;
public class HelloWorld {
    public void hello() {
        System.out.println("Hello World");
    }
    public void hello(String name) {
        System.out.printf("Hello %s!\n", name);
    }
}
```

Call hello from Jython

```
public PyObject hello();
  Code:
    0: aload_0
    1: ldc #93; //String hello
    3: invokestatic #56; //Method Py.jgetattr:
(LPyProxy;LString;)LPyObject;
    6: getstatic#40; //Field Py.EmptyObjects:[LPyObject;
    9: invokevirtual#62; //Method PyObject._jcall:
([Object;)LPyObject;
    12:  areturn
```

Call Jython from Java

- Old method: jythonc, but it used its own compiler and isn't useful anymore
- PythonInterpreters can be created and used, but that's too hard for simple cases
- Need a simple way to specify Java signatures for Jython classes.

jythonc replacement

- Need a way for users to specify Java signatures for Jython classes and methods
- The backend implementation should be fairly straightforward
- The client interface will be harder to get right
- May just have a _jythonc.py to start with

Jython Decorators

```
class Simple(object):
    @javamethod
    def __init__(self):
        pass

    @javamethod(String, String)
    def firstWord(self, param):
        return param.split(' ')[0]
```

Jython Decorators

```
//Produce bytecodes corresponding to:
public class Simple {
    public Simple() {
    }
    public String firstWord(String arg0) {
        //Call Jython firstWord
    }
}
```

“with” statement

```
with VAR as EXPR:
    BLOCK

#roughly translates into this:
VAR = EXPR
VAR.enter()
try:
    BLOCK
finally:
    VAR.exit()
```

Where to find out more

- http://www.jython.org
- http://wiki.python.org/jython
- http://fwierzbicki.blogspot.com
- Twitter: fwierzbicki
- http://www.python.org